КОМПЮТЪРНА ГРАФИКА И ГЕОМЕТРИЧНО МОДЕЛИРАНЕ

ЧАСТ I. В РАВНИНАТА

Евгений Лукипудис

Книгата е одобрена като учебно пособие за курса по "Компютърна графика и приложения" от катедрения съвет на катедра "Компютърна информатика" при Факултета по математика и информатика на СУ "св. Климент Охридски".

Редактор и коректор: **Бойко Б. Банчев**

Художник на корицата: **Стефан Лютаков**

ISBN 954-8935-01-5 (ч. 1)

Предговор

Това е първата част от монография по "Компютърна графика", предназначена да служи за основен текст на студентите от специалности като "информатика" и "изчислителна техника" и особено на разработчиците на приложни графични програми и системи за моделиране. Надявам се, че тя ще бъде полезна и на всички програмисти, на които се налага да решават геометрични задачи.

Тази книга е резултат на шестгодишна подготовка на курса по "Компютърна графика и приложения", който чета във Факултета по математика и информатика на Софийския университет. Този курс претърпя много и най-вече структурни промени, а настоящата книга е отражение на моето виждане за организацията му. Разделянето на "двумерна" и "тримерна" графика се наложи от ударението, което тук се поставя на геометричното моделиране. Компютърната графика отдавна вече не се отъждествява само със синтезирането на изображения. В практиката много по-често се налага да се търсят модели за представяне на геометричната форма и методи за извършване на разнообразни обработки и анализи върху тях. В тази връзка съм се постарал да обоснова необходимостта и да изясня причините, поради които в компютърната графика се прилагат определени математически формулировки. На тази именно основа геометричните трансформации и представянето им чрез хомогенни координати се разглеждат в главата за геометрично моделиране, а не като нещо самостоятелно.

В тази част са разгледани основните алгоритми, модели и методи само на т.нар. "двумерна компютърна графика". Това дава възможност за представяне на различни подходи при решаването на един и същ проблем. Самите проблеми са пряко свързани с практическите нужди при разработката на графични програми. Поради тази причина са включени и такива задачи, които не присъстват в литературата по компютърна графика, каквато е например тази за растеризирането на дъга от окръжност. Основните използвани източници са монографиите на Фоли, Ван Дам, Фейнер и Хюз 1990 и на Роджърс и Адамс 1990, но са включени и много допълнителни алгоритми, като например този на порциите, предложен от Брезенхам. Голяма част от алгоритмите са представени във вид на завършени програмни фрагменти. Езикът за програмиране C бе избран при кодирането на алгоритмите единствено от съображения за компактност на текста.

Всяка от главите е сравнително самостоятелна. Единствената зависимост е на четвърта, пета и шеста глави от представения в трета глава Базов Графичен Пакет (БГП). Читателите, имащи опит с базово графично програмно осигуряване, не е нужно да се спират подробно на БГП, тъй като лесно ще разпознаят използваните графични примитиви. Особено внимание е отделено на средствата за структуриране на изображението, типични за съвременните графични системи. Като илюстрация тук е използван моделът на системата PHIGS. За разбиране на моделирането на йерархични структури чрез сегментни мрежи, представено в пета глава, е необходимо първо да се прочете последната част от трета глава.

Материалът в настоящата книга е разчетен за едносеместриален курс с хорариум 45 часа. Убеждението ми е, че такъв задълбочен курс по двумерна графика създава предпоставки за по-нататъшни специализирани курсове по "Реалистична визуализация и анимация в 3D", "Геометрично моделиране на пространствени форми", "Моделиране на криви и повърхнини", "Графични стандарти", "Изчислителна геометрия" и др., някои от които са и основните раздели във втората част на тази монография.

Искам да благодаря на колегите си от катедра "Компютърна информатика" на ФМИ и най-много на бившите си колеги (но винаги приятели) Николай Манев и Бойко Банчев, с които работих дълго време във вече закритата "Лаборатория по компютърна графика" към Института по математика на БАН. Особено съм задължен на Бойко Банчев за редакцията на текста и извънредно ценните му поправки и забележки. Благодаря също и на Николай Манев, Владимир Владимиров и Николай Николов за техните препоръки и корекции. На Николай Николов принадлежи идеята за интерактивния метод за задаване на правоъгълник с фиксирани пропорции, както и фигурата, представяща обемното сечение на 3 и D, използвана за илюстрация на корицата.

Задължен съм и на всички мои бивши и настоящи студенти, работата с които ми е дала болшинството от идеите за структурирането и излагането на материала. Моите благодарности на Владимир Владимиров, Ивайло Пеев, Веселка Боева, Димитър Козалиев, Стефан Лютаков и Любомир Европейски за помощта при окончателното оформяне на книгата, а на Елена за грижите и подкрепата по време на писането ѝ.

Ще бъда признателен за всички отзиви, предложения, поправки и препоръки, които биха подобрили едно евентуално следващо издание. Електронният адрес за връзка е evgueni@fmi.uni-sofia.bg

Е. Н. Лукипудис

Съдържание

Глава 1

Въведение в компютърната графика

Компютърната графика (често наричана у нас и *машинна графика* заради буквалния превод от руски език) започва своето развитие много скоро след създаването на компютрите и първите устройства за извеждане на текст. Програми, които отпечатват изображения върху постоянен носител (хартия, паус и др.) са били писани още в зората на програмирането. За разлика от другите информатични дисциплини, компютърната графика е била винаги много зависима от наличните технически средства, така че нейното развитие е следвало в общи линии прогреса в разработването на компютърните периферни устройства.

Създадената през 1950 год. в Масачузетския технологичен институт компютърна система Whirlwind вече е разполагала с графичен екран на принципа на електронно-лъчевата тръба, използван само за извеждане на графична информация. Разработената скоро след това система за противовъздушна отбрана SAGE е известна като първата изчислителна система, снабдена с графичен екран за управление на работата ѝ. В тази система изобразяваните върху екрана въздушни цели е можело да бъдат посочени от оператора чрез светлинно перо (диалогово устройство, отчитащо светлинни пулсации).

Началото на съвременната компютърна графика поставя Айвън Садърланд (Ivan Sutherland) с разработената от него система Sketchpad през 1962 год. като дисертационен труд в Масачузетския технологичен институт. В своята работа Садърланд поставя основите на много от интерактивните похвати и методи, използвани широко днес при проектирането на взаимодействието на човека с компютъра чрез графични средства - *графичния диалог*. Той предлага и структури от данни за представянето на графичната информация и разработва методи за моделиране на геометричните характеристики на обектите. Много скоро след това мощните концерни в автомобилостроенето и самолетостроенето осъзнават огромния потенциал на компютърната графика и геометричното моделиране при автоматизирането на дейности като чертане, проектиране и дори производство. В средата на шестдесетте години General Motors, Boeing, McDonnell-Douglas, а в Европа (малко по-късно) Renault и Matra Datavision започват практически да експлоатират интерактивната компютърна графика и методите за моделиране при компютърната автоматизация на ин-

женерните дейности.

На конгреса на IFIP през 1965 год. Садърланд излага своя проект за създаване на *съвършен дисплей*, извежданите образи върху който (при помощта и на различни допълнителни средства) да не могат да се различават от действителните обекти. Три години по-късно той демонстрира прототип на своя дисплей, който представлява каска, в която са разположени два малки екрана, оптически свързани поотделно с всяко око и допълнителна специализирана система за отчитане на положението и ориентацията на каската във всеки момент. Това полага основите на ново направление в компютърната графика - системите за симулиране на реална обстановка или *виртуална реалност* (Virtual reality). В компютърната графика се оформя специален дял за генериране на реалистични изображения с отчитане на разнообразни оптически ефекти като сенки и полусенки, разсеяна и отразена светлина, прозрачност и полупрозрачност, както и методи за стереоскопично визуализиране.

Въпреки бурния старт в началото на 60-те години, компютърната графика остава сравнително тясна област до края на 70-те, главно поради високите цени на графичните устройства и необходимостта от значителни изчислителни ресурси - за обслужване на графичното взаимодействие и за съхранението и обработката на геометричните модели. Друга причина е необходимостта да се пишат интерактивни графични програми и моделиращи системи при липса на каквото и да е базово графично програмно осигуряване, всеки път по различен начин за всяка нова създадена компютърна конфигурация. Проблемът с нарастващото изобилие от различни по тип графични устройства предизвиква усилена работа по разработването на т.нар. *графични стандарти*, осигуряващи мобилност на програмното осигуряване, първите от които се появяват още през 1977 год.

В тези години се създават много от разглежданите в тази книга алгоритми за визуализация, както на равнинни, така и на пространствени обекти; много от методите за моделирането на тези обекти; започва систематизирането на интерактивните методи и похвати за да се появят първите системи за управление на графичния диалог. Повечето от алгоритмите и методите са резултат на решаването на конкретни приложни задачи: управление на процеси; разработване на *тренажори за летци*; проектиране на формата на самолети, автомобили и кораби; проектирането на машиностроителни детайли и интегрални схеми; симулирането на поведението на обекти и др.

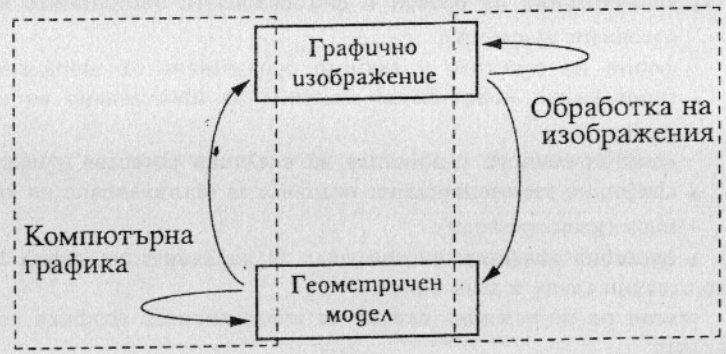
С масовото навлизане на сравнително евтините персонални компютри, снабдени с растерни дисплеи и устройства за графичен вход в началото на 80-те години разработването на приложни графични програми минава на друг етап. Графичното взаимодействие става предпочитан и сравнително лесно осъществим начин за комуникация между потребителя и приложните програми от най-различни области. Това постепенно води до организирането на интерактивните похвати в стройни системи, налагащи определен стил на работа. В същото време геометричното моделиране отива далеч отвъд представянето на равнинни фигури и сцени от пространствени обекти за визуализация. Самата компютърна графика престава да бъде само средство за създаване на образи. Все по-голямо внимание започва да се отделя на използваните

модели, интеграцията им и ефективната им обработка. Нещо повече, изследванията се насочват и към представяне на измененията в геометричната форма и поведението на сложни пространствени обекти. По този начин компютърната графика става все по-тясно свързана със симулацията и анимацията на обектите за постигане не само на визуално реалистично изобразяване, но и на физически правилно представяне.

Днес компютърната графика има вече много дялове и все по тясно се интегрира и с останалите области на информатиката. За да подчертаем важността на моделирането, което се оформи като самостоятелен дял на компютърната графика, тази книга е озаглавена *"Компютърна графика и геометрично моделиране"*. Ние не твърдим, че това са отделни области, а по-скоро желаем да съсредоточим вниманието на читателя върху тези дялове от компютърната графика, които са най-тясно свързани с моделирането. Убеждението ни е, че именно знанията в тези дялове ще бъдат най-полезни на разработчиците на програмно осигуряване.

1.1 ПРЕДМЕТ НА КОМПЮТЪРНАТА ГРАФИКА

Компютърната графика е един от дяловете на информатиката, свързани със създаването, съхранението и обработката на визуална информация. Предметът на компютърната графика е синтезирането на изображения на обекти на базата на създадени за тези обекти подходящи описания - компютърни модели. Обратният процес - анализирането на образи, създаването на модел на един обект въз основа на един или няколко негови графични образа - е предмет на *обработката на изображения* (Image Processing). Обработката на изображения включва много отделни дисциплини като *подобряване на изображения* (чрез подходяща филтрация и отстраняване на "шум"); *разпознаване на образи* (или по-точно на образци и символи); *анализ на сцени*; *реконструиране на обекти* (най-често от няколко различни техни образи); *компютърно зрение* и др.



Фиг. 1-1

Самата компютърна графика е наука за методите на съхранение, създаване и обработване на модели и техните визуални образи с помощта на ком-

пютър. Тези методи днес са изключително *интерактивни*: потребителят на една графична система може динамично да управлява вида, структурата и начина на изобразяване на обектите, използвайки средствата на т.нар. *графичен диалог*. Ето защо напоследък рядко се използва терминът *интерактивна компютърна графика* - в него вече се крие тавтология. Другият вид - *пасивната* графика е само една нейна част, която засяга методите за визуализация на обектите.

Някои от по-важните дялове на компютърната графика са обособени и като отделни глави в тази книга:

- растерна графика: алгоритмите за визуализация върху растерни дисплеи (Raster Graphics);
- графично взаимодействие: методите за организиране на комуникацията *човек-компютър* с помощта на визуални средства (Graphical User Interface);
- геометрично моделиране: методите за представяне на геометричната форма на обектите и тяхното обработване от приложните програми (Geometric Modelling);
- визуализация на образи и фотореализъм: алгоритмите и методите за изобразяване на обекти, създаване на реалистични картини с премахнати невидими линии и повърхнини, при отчитане на различни оптически ефекти, осветление, оцветяване и т.н. (Image Rendering);
- моделиране на криви и повърхнини: математическите методи за представяне и обработване на фигури с произволни криволинейни граници (Computer-Aided Geometric Design);
- изчислителна геометрия: алгоритмите за решаване на геометрични комбинаторни задачи (Computational Geometry) и др.

Ние сме разделили темите по компютърна графика на две части - *в равнината* и *в пространството*. Това разделение е направено единствено за полесно навлизане на читателя във всеки от отделните дялове. Основните принципи и в двата случая - двумерния и тримерния - са много подобни. Двумерните методи и алгоритми обаче могат да се реализират много по-лесно, така че разглеждането им отделно позволява пряко сблъскване на читателя с практическите проблеми при писането на графични програми.

Трябва да отбележим, че компютърната графика е тясно свързана с математически дисциплини като аналитична геометрия, дескриптивна и проективна геометрия, диференциална геометрия, числени методи и комбинаторика. В тази книга се разчита на математическите знания на читателя, особено по аналитична геометрия.

1.2 ТЕХНИЧЕСКИ СРЕДСТВА ЗА ГРАФИЧЕН ИЗХОД

Техническите средства за графичен изход са интересни за нас само от гледна точка на принципите, на които те са построени, тъй като тяхната архитектура пряко влияе както на използваните алгоритми в графичните програми, така и на структурирането на тези програми.

1.2.1 Графични дисплеи

Интерактивността в компютърната графика изисква визуализация върху устройства, образите върху които да могат да се променят бързо. Най-широко използвани за тази цел са графичните дисплеи, а от тях пък са тези с електронно-лъчева тръба, които твърде малко се отличават от телевизионните приемници. Много от качествата на тези дисплеи ги правят най-важните устройства в компютърната графика и гарантират масовото им използване и в бъдеще.

Електронно-лъчевата тръба (ЕЛТ) е построена на следния принцип: сноп от електрони, излъчван от нагрят катод и ускоряван от високо напрежение към вътрешността на екран, която е покрита с фосфор (луминофор) се фокусира от специална система по магнитен път, за да достигне определена точка от това фосфорно покритие. Цветните ЕЛТ разполагат с покритие от три различни цвята луминофор: най-често червен, зелен и син - RGB. Екранът представлява съвкупност от малки цветни точки, групирани по три в т.нар. триади. Три отделни катода (за всеки от цветовете) излъчват електрони, които минават през метална маска преди да достигнат фосфорното покритие. Тази маска осигурява попадението на трите лъча само върху три съседни луминофорни точки от различни цветове (една единствена триада), които пък осветени едновременно изобразяват цвета, съответстващ на смесването на трите интензитета.

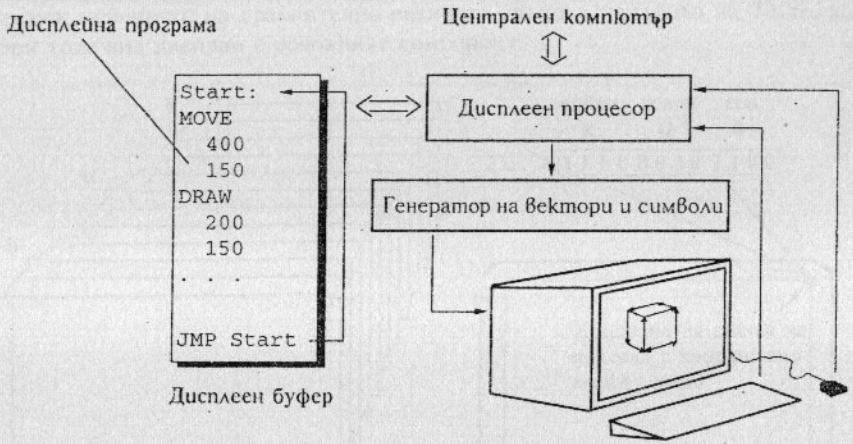
Не е нужно да се спираме по-подробно върху детайлите на горния процес. Това, което е важно за нас е, че светлинният импулс на луминофора затихва експоненциално с времето, така че образът се нуждае от непрекъснато прерисуване дори и тогава, когато е статичен. При честота на прерисуване типично 60Hz вече се създава илюзията за постоянно изображение. За постигане на напълно нетрептящ образ (flicker-free image) обаче в повечето случаи е необходима честота 70-90Hz. В зависимост от начина за регенерация на образа дисплеите с ЕЛТ биват два вида: *векторни* и *растерни*.

Тук няма да разглеждаме такива средства като дисплеите с течни кристали или плазмените дисплеи, защото от гледна точка на програмиста те са също растерни устройства.

ВЕКТОРНИ ДИСПЛЕИ. Тези дисплеи са били разработени в средата на 60-те години и са били широко използвани до масовото навлизане на персоналните компютри и растерните устройства. Те се наричат често още и *дисплеи с произволно регенериране* (random-scan display) заради това, че прерисуването се извършва, като се изчертава поредица от вектори, които могат да бъдат разположени произволно в изображението. В този смисъл визуализирането на един вектор не зависи от неговото положение върху екрана. Архитектурата на векторния дисплей е показана на фиг. 1-2.

Изображението (във вид на вектори и символи) се съхранява в памет, наречена *дисплеен буфер* като последователност от команди за специализиран процесор, наречен *дисплеен процесор*. Тази последователност често се нарича и *дисплейна програма*. Дисплейният процесор интерпретира командите и за

всеки вектор изпраща координатите на краищата му към *генератор на вектори*, който го преобразува в напрежения, които управляват движението на електронния лъч от началото до края на вектора. Последната команда в дисплейната програма е винаги безусловен преход към нейното начало, което осигурява непрекъснатата регенерация на образа.

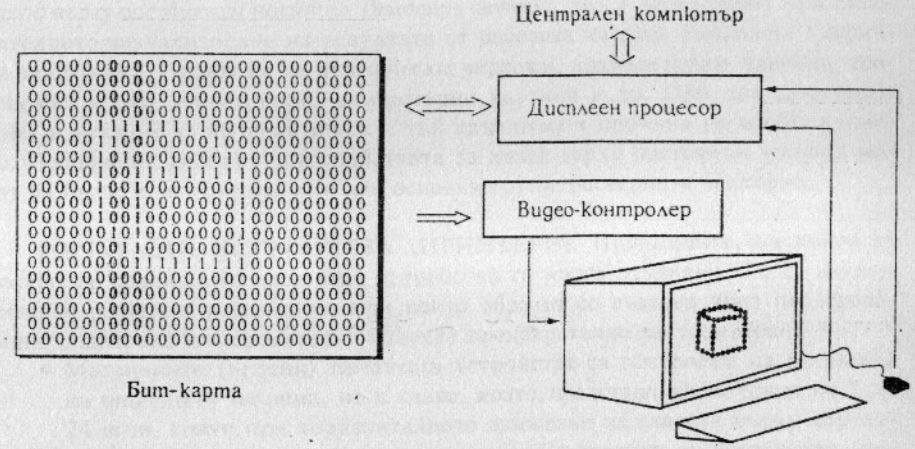


Фиг. 1-2

Очевидно е, че при такава архитектура честотата на прерисуване силно зависи от сложността на образа. При качеството на дисплеите от 60-те години няколко хиляди вектора са били горната граница на визуализация без видимо трептене. С увеличаването на бързодействието в следващите години, дисплейните процесори се усложняват значително и започват да обработват структурирани дисплейни програми, да изпълняват геометрични трансформации, отсичане и дори успоредни и перспективни проекции за визуализиране на пространствени обекти. Необходимостта от циклично изпълнение на дисплейната програма води до ограничение в нейната дължина. Това пък от своя страна изисква структуриране на образа така, че да не се повтаря описанието на еднакви елементи. От тези години датира *сегментацията* на изображението (разгледана в трета глава) и матричното представяне на геометрични трансформации и проекции.

ДИСПЛЕИ СЪС ЗАПОМНЯЩА ЕЛЕКТРОННО-ЛЪЧЕВА ТРЪБА. За да се реши проблемът с трептенето на образа на векторните дисплеи при голям брой вектори, в края на 60-те години се разработват дисплеи, при които луминофорът има много дълго време на светлинно затихване (повече от 1 час). При тези дисплеи практически няма нужда от регенерация, а промяната на изображението се извършва при пълно изтриване на екрана и визуализирането изцяло на новия образ. Като резултат се получава стабилно изображение за сметка на някои ограничения върху интерактивността и възможностите за промяна на образа. Тази технология позволява създаване на сравнително евтини графични дисплеи, които веднага стават масовото графично устройство. Дори и днес много графични системи предлагат емулация на дисплеи като Tektronix 4010, за който като първи достъпен дисплей от този вид са написани извънредно много програми.

РАСТЕРНИ ДИСПЛЕИ. В началото на 70-те години започва разработването на евтините растерни дисплеи, основани на принцип, близък до този на телевизионната развивка.



Фиг. 1-3

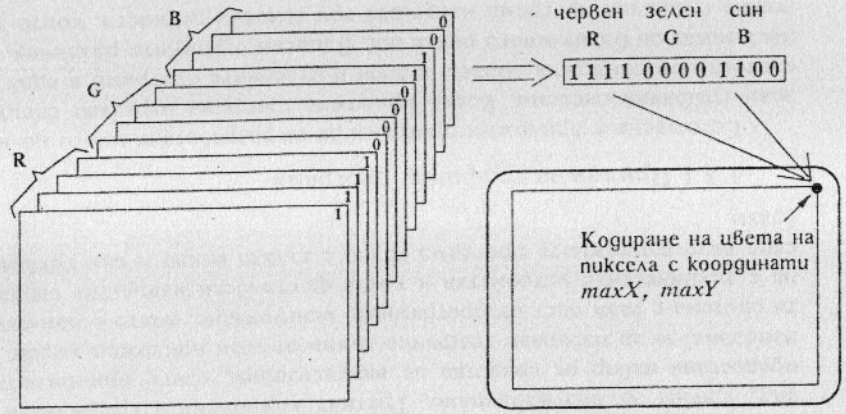
При черно-белите дисплеи дисплейният буфер се заменя от правоъгълна матрица (растер), която представя цялата област на екрана във вид на редове и стълбове от нули и единици, като всяка нула отбелязва точка, която не е осветена, а всяка единица - светеща точка. Тази матрица често се нарича *бит-карта* на екрана (bitmap).

В цветните дисплеи на всяка точка от екрана (наричана *пиксел* - pixel) се съпоставят по няколко бита, превръщайки по този начин матрицата на растера в тримерна матрица с дълбочина от 8 до 96 бита. 24 бита са достатъчни за кодирането на 16 милиона различни цвята. Тримерната матрица от нули и единици носи името *пикселна карта* (pixmap) за да се отличава от бит-картата, която е характерна за дисплеите с един бит за пиксел. Броят на битовете, отделени за всеки пиксел (третата размерност в тази матрица) често се нарича *дълбочина на дисплея* или брой равнини на дисплея. На фиг. 1-4 е показана пикселната карта на един растерен дисплей с 12 равнини - по четири бита за всеки от основните цветове (червен, зелен и син) - и начинът, по който е кодиран цветът в тази карта на точката, намираща се в горния ляв ъгъл на екрана.

При растерните дисплеи, дисплейният процесор се опростява неимоверно. Една част - наричана просто *видео-контролер* по аналогия с контролерите на други периферни устройства - се отделя от него и започва да изпълнява една единствена задача: да сканира ред след ред (отгоре-надолу и после отново) матрицата на растера и да изпраща аналогови сигнали към катода на ЕЛТ за

емисия на електрони за всеки елемент от един ред, който е различен от нула.

Този тип дисплей често е наричан и *дисплей с последователно регенериране* (raster-scan display), тъй като пътят на електронния лъч е фиксиран, следвайки обхождането на растерната матрица, а интензитетът му зависи от съдържанието на съответната растерна клетка. Успехът на растерните дисплеи се дължи най-много на сравнително евтината памет в началото на 70-те, което при този вид дисплеи е основният компонент.



Фиг. 1-4

Предимствата пред векторните дисплеи са цената, независимостта на регенерацията от сложността на изображението и не на последно място възможността за ефективно запълване на области (дори и с образци), което се оказва много полезно при визуализацията на пространствени обекти. Недостатъците са главно в тяхната дискретност: един вектор трябва първо да бъде превърнат в поредица от пиксели, при което изчезва структурираността на елементите - изображението се състои само от пиксели. Това оказва определено влияние при разработването на програмно осигуряване. Алгоритмите за получаване на растеризирани образи на геометрични обекти са предмет на растерната графика, която ние ще разгледаме в следващата глава. Друга последица от дискретността е стъпаловидността на изображението, за визуалното отстраняване на което започват да се разработват допълнителни програмни средства.

Растерните дисплеи се различават по разделителната способност на устройството. Тя е пряко свързана с размерността на матрицата на растера, а оттам и с необходимата дисплейна памет. Съвременните масово използвани растерни дисплеи имат разделителна способност 1280×1024 пиксела при размери между 14 и 21 инча по диагонал. Много от специализираните модели имат апаратно реализирани възможности за визуализация на пространствени обекти с премахване на невидимите линии и повърхности; осветяване с отчитане на оптически ефекти като сенки, полусенки, прозрачност и др.; изпълнение на пространствени трансформации над моделите им; визуализиране на произволни повърхнини чрез рационални сплайн-функции и др.

1.2.2 Устройства за изход върху постоянен носител

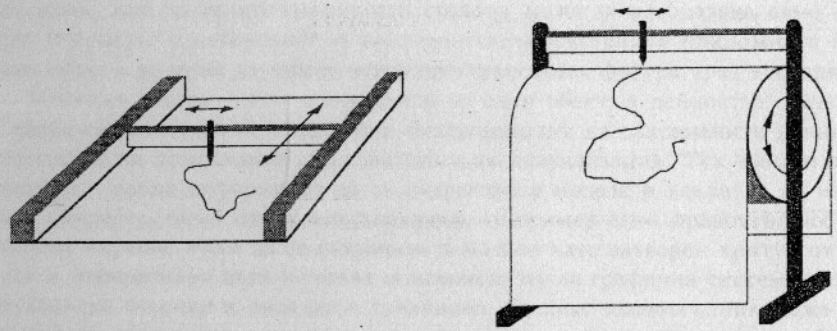
Тук ще направим съвсем кратък преглед на устройствата, които се използват за постоянен графичен изход, наричани с общото име *устройства за изход върху постоянен носител* (hardcopy devices). Те са необходими при окончателното визуализиране на резултата от работата на една приложна графична програма - създаване на технически чертежи, архитектурни планове, географски карти, илюстрации, компютърни картини и др. При тях естествено няма изискване за интерактивност, тъй като няма и промяна на изображението. Подобно на дисплеите устройствата за изход върху постоянен носител могат да бъдат класифицирани в две основни групи: растерни и векторни.

ПЕЧАТАЩИ УСТРОЙСТВА (ПРИНТЕРИ). Принципите, на които са построени различните печатащи устройства са много различни. Те са изключително растерни устройства, при които образът се създава чрез последователно сканиране и следователно изисква предварително растеризиране.

- Матричните (иглени) печатащи устройства са построени на принципа на пишещата машина, но с глава, която представлява матрица от 7 до 24 игли, които при хоризонталното движение на главата върху хартията отпечатват няколко реда от растера. За разлика от растерните дисплеи, тук един ред може да бъде отпечатан няколко пъти с различни отмествания, при което могат да се разработят допълнителни методи за премахване на стъпаловидния ефект. Цветен печат се постига с използването на няколко ленти, но резултатът рядко е удовлетворителен.
- Принтерите с впръскване на мастило са евтини устройства за цветен печат. Една глава с три мастилени инжектора сканира листа хартия и впръсква на пиезоелектричен принцип комбинация от трите основни цвята, при смесването на които се получава желаният.
- Лазерните принтери са също растерни устройства, при които лазерен лъч сканира въртящ се, положително зареден селенов барабан. Точките, в които лъчът докосва барабана губят положителния си товар и в крайна сметка положително заредени остават само тези, които трябва да са черни. Прахообразно мастило (тонер) залепва върху положително заредените области и после се нанася върху хартия. Лазерният принтер е снабден с микропроцесор, който извършва растеризирането. Някои принтери могат да интерпретират и програми, написани на специализиран език: PCL, HP-GL, PostScript.
- Електростатичните принтери зареждат отрицателно специална хартия и след това нанасят течен, положително зареден тонер върху нея. Те също, подобно на лазерните принтери, може да имат микропроцесор за извършване на растеризация и интерпретация на специализирани програми.

- Термопринтерите са подобни на електростатичните, само че принципът се прилага към специална лента, покрита с восък, която при нагряване го разтопява и отделя полепналото върху нея мастило върху обикновен лист хартия.

ЧЕРТОЖНИ УСТРОЙСТВА (ПЛОТЕРИ). Плотерите са най-често векторни устройства, при които специален държач движи писалка, която чертае върху хартиен лист. Предимствата на плотерите са най-вече в по-голямата им точност (разделителната им способност е типично 40 точки на милиметър, което е повече от три пъти по-добро от принтерите с 300 точки на инч) както при позициониране, така и при използване на точна дебелина на линиите. Друго предимство са техните размери (съответно и на листа, върху който се изчертава изображението), които са значително по-големи от тези на принтерите. Движението на писалката при изчертаването на един вектор не е плавно. То се разбива на много стъпкови движения най-често в осем посоки, но поради голямата разделителна способност векторите изглеждат гладки.



Планшетен плотер Барабанен плотер

Фиг. 1-5

Плотерите са снабдени с микропроцесор, който интерпретира програма на специализиран език за описание на векторни образи: BGL, HP-GL, Gerber. Основните команди на този език са:

- вдигане на перото от хартиения лист;
- спускане на перото върху хартиения лист;
- придвижване от текущата позиция до определена точка.

Допълнителните команди включват: чертане на дъги от окръжности и елипси, символи с произволен наклон и размер, задаване на типове линии, щриховане и др.

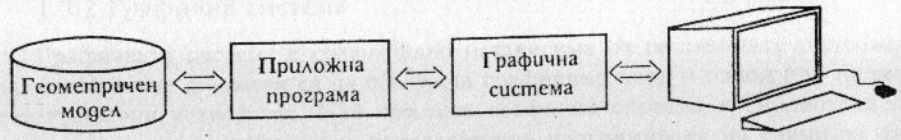
Използват се най-често два вида плотери (фиг. 1-5):

- плоски (планшетни, flatbed plotter), при които листът хартия е неподвижен, едно рамо се движи в едната посока (най-често по ширината на листа), а по това рамо в другата посока се движи държачът на писалката. Началото на координатната система е най-често в единия от ъглите на листа.

- барабанни (drum plotter), при които хартията се движи в едната посока посредством въртене върху барабан, а писалката се движи от държача в другата посока. Тези плотери са удобни при големите формати A1-A0. Тук трябва да отбележим, че високата разделителна способност и големите размери налагат използване на цели числа, които излизат извън интервала $0 \div 32767$. Поради тази причина координатното начало може да е разположено в центъра на листа хартия (за да се използват и отрицателните двубайтови цели числа за координати на точките).

1.3 ПРОГРАМНО ОСИГУРЯВАНЕ ЗА КОМПЮТЪРНА ГРАФИКА

На фиг. 1-6 е представена концептуалната схема на програмна система за интерактивна графика.



Фиг. 1-6

Една такава система има три компонента: моделът, който се създава, обработва и визуализира; приложната програма, която се грижи за създаването му, извършването на операции върху него, както и за организирането му във вид, удобен за визуализиране. Третият компонент - графичната система - предоставя набор от средства за визуализация, които приложната програма използва, за да изобрази графично данни от този приложен модел. Графичната система е тази част от програмното осигуряване, която е най-тясно свързана с техническите устройства и която фактически извършва визуализацията, след като приложната програма точно е задала какво и как трябва да се изобрази.

1.3.1 Приложна графична програма

Естествено е приложни програми да се разработват много по-често от базови графични системи. Всяка приложна програма отразява спецификата на съответната приложна област, а дори и отделните програми в една и съща област могат да бъдат много различни. Въпреки огромните различия, всяка приложна графична програма извършва три основни дейности, които за нас имат определено значение:

- моделиране,
- описание на модела за графичната система и
- интерактивна работа.

МОДЕЛИРАНЕ. Всяка приложна програма решава конкретна задача: изобразяване на молекулната структура на различни химични вещества; про-

ектирането на вътрешната архитектура на една сграда; анализ на движението на различни механизми; пресмятане и визуализиране на векторни полета в електромагнетиката, хидродинамиката и др. За решаването на тази задача приложната програма построява и използва *геометричен модел* на обектите, с които работи. Огромна част от програмистките труд се влага именно в проектирането, създаването, обработването и съхраняването на този модел. Моделите биват най-различни (ние ще разгледаме най-много използваните видове и начините, по които те се обработват в отделна глава), но всички те малко или много описват геометричната форма на обектите, с които приложната програма и нейният потребител боравят.

ОПИСАНИЕ НА МОДЕЛА ЗА ГРАФИЧНАТА СИСТЕМА. Описанието на геометричната форма в модела е пряко свързано с поставения приложен проблем и е напълно независимо от графичната система. За да бъде визуализиран приложния модел или някаква част от него, е необходимо той да бъде представен в термините на графичната система, която извършва самото изобразяване. Ако например графичната система може да изобразява само вектори, а моделът е съвкупност от пространствени фигури, то приложната програма трябва да може да опише всяка пространствена фигура чрез вектори.

Моделът е това, което е съхранено за един обект, а дейността *описание за графичната система* е един вид интерпретация на съхранените данни за получаване на тези, които са необходими за визуализация. Тук винаги стои дилемата - каква информация да се съхранява в модела и каква да се изчислява непосредствено преди визуализация. Например един правоъгълник със заоблени върхове може да се съхранява в модела като затворен контур от отсечки и допирателни дъги и тогава описанието му за графична система, която визуализира отсечки и дъги ще е тривиално. Същият правоъгълник може също да бъде представен чрез краищата на диагонала си и радиуса на заобляне на върховете му. Тогава модулът за описанието на модела за същата графична система ще трябва да изчисли краищата на отсечките и дъгите на този заоблен правоъгълник по данните в модела.

Приложната програма трябва също така да може да представи за визуализация само определена част от модела или някакъв определен негов *изглед*. При визуализирането на модел от пространствени фигури с двумерна графична система например трябва да се зададе проекцията и да се генерира плоският образ на пространствената сцена съобразно тази проекция.

ИНТЕРАКТИВНА РАБОТА. Една графична програма взаимодейства с потребителя най-често по графичен начин - чрез графични образи (графичен вход и изход или само графичен изход).

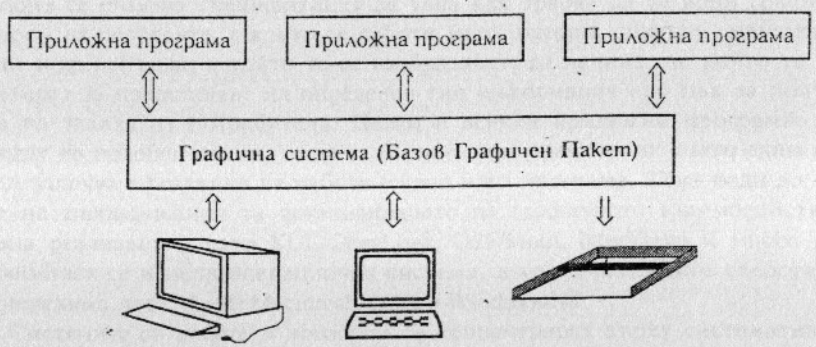
В някои приложения не се налага много често намеса на потребителя, а в други приложните програми са организирани като набор от програмни модули, всеки от които се активира и извършва определена дейност след интерактивно избиране на *команда* от потребителя. По-голяма част от интерактивната работа протича по цикличната схема: избор на команда; изпълнение на командата; предоставяне на възможност за избор на нова команда.

Фиг. 1-7

Взаимодействието между приложната програма и потребителя с графични средства е същността на т.нар. *графичен диалог*. На въпросите на графичния диалог - как да се организира, кой да играе водеща роля (приложната програма или потребителя), какви средства и похвати да се използват в типични ситуации - е посветена четвърта глава на тази книга.

1.3.2 Графична система

Графичната система е сравнително независима от решаваните приложни задачи. Нейните функции са да обслужва графичния вход и изход при наличните графични устройства. Най-простата графична система е библиотека от подпрограми, които могат да се използват при програмиране на езици на високо ниво като С, Паскал и ФОРТРАН за визуализиране на прости геометрични елементи: отсечки, дъги, символи и др. Една такава библиотека, наречена "*Базов Графичен Пакет*" ние представяме в трета глава. Нейната основна цел е да освободи приложната програма от детайлите по визуализацията върху конкретните устройства, каквито са растерните алгоритми.



Фиг. 1-8

Растеризирането на отсечки и дъги за растерните устройства е само една част от това, което графичната система извършва. Тя решава още следните глобални задачи:

- Да предостави на приложната програма възможност за работа в координатна система, типична за геометричния модел, който тя обработва;
- Да се грижи да изобразява само тази част от модела, която в определен момент е необходима за работата на приложната програма;

- Еднотипно да определя положението на видимата част от модела върху всички налични графични устройства;
- Да предостави еднотипни функции за визуализиране на основните графични примитиви, независими от вида на конкретното графично устройство;
- Да предостави еднотипни функции за установяване на визуалните атрибути - цвят, тип на линията, дебелина на линията, образец на запълване и др. - на изобразяваните графични примитиви;
- Да предостави независимост от устройствата за графичен вход.

Това са само основните задачи на графичната система. Все по-голямо внимание в съвременните графични системи се отделя на предоставяне на средства за по-лесно описание на моделите: визуализация на пространствени обекти, реалистична визуализация, наличие на примитиви като рационални криви и произволни повърхнини, както и предоставяне на средства за структуриране в приложни термини и дори геометрично моделиране.

ПРЕНОСИМОСТ И ГРАФИЧНИ СТАНДАРТИ. Програмното осигуряване на първите графични устройства е било на сравнително ниско ниво, разработено от самите производители на устройствата и естествено твърде зависимо от техните конкретни особености. Може би най-популярен графичен пакет от началото на 70-те години става PLOT-10, разработен като базово програмно осигуряване на графичните дисплеи Tektronix 4010, за чиято популярност споменахме по-горе. За да може да се осигури преносимост на приложните графични програми от една платформа на друга, както и преносимост на знанията на самите приложни програмисти, в средата на 70-те започва разработването на спецификация за графична система. Като резултат през 1977 год. се създава документацията на системата *Core*, която е обновена отново през 1979 год. Въпреки че не става официален стандарт, много производители създават нейни реализации, които се използват широко до 1985 год., когато се въвежда като официален стандарт системата *GKS* (*Graphics Kernel System*). За разлика от Core, GKS е само двумерна система и според много автори тя е една изчистена от излишества система.

В GKS за пръв път се въвежда концепцията за работна станция, дефинира се система от абстрактни входни устройства (ние ще разгледаме подробно тези устройства в четвърта глава), която се използва широко и днес и се въвежда спецификация за обмен на графични данни (графичен метафайл). В тази система се въвеждат и групи от логически свързани графични примитиви, наречени "сегменти", които не могат да се съдържат един в друг. През 1988 год. GKS-3D става също официален стандарт, но бива почти моментално изместен от много по-сложната, също тримерна, но с много по-големи възможности система *PHIGS* (*Programmer's Hierarchical Interactive Graphics System*). Тя е първата система, която е ориентирана не само към визуализация, но и към моделиране на визуализираните обекти. Групите от логически свързани примитиви (наречени *структури*) вече могат да са вложени една в друга, да бъдат редактирани и визуализирани, при което системата практически предлага на приложния програмист една графична йерархична база от данни. Всяка съхранена структура може да съдържа и неграфични данни, което цели да даде на програмистите всички необходими средства за геометрично моделиране. Примитивите на PHIGS включват и рационални криви и повърхнини освен стандартните отсечки, дъги, окръжности, многоъгълници и текст. Скоро след това се появява *PHIGS+*, в който са реализирани средства за псевдореалистична визуализация върху растерни дисплеи. Реализациите на тази система са много големи програмни продукти и тяхното ефективно използване е възможно най-вече на компютри с достатъчно бързодействие и графични устройства с апаратно реализирани отсичане, трансформации, отстраняване на невидими линии и стени. Успоредно с тези системи се разработват и други графични системи, каквато е например библиотеката *OpenGL* на фирмата Silicon-Graphics.

С появата на персоналните компютри, снабдени с евтини растерни дисплеи, започва разработването и на прости графични пакети за растерна графика. Най-популярни стават пакетите за компютри от типа на Apple и IBM-PC. В средата на 80-те вече се появяват и първите системи за работа в прозорци, при които графичното взаимодействие е много важен елемент. За да се осигури преносимост и на програми, които не са непременно графични, но използват графичен диалог, между отделни компютри и устройства, а също и между отделни операционни системи през 1984 год. в Масачузетския технологичен институт се създава системата X Window System. Проектирана на принципа *клиент-сървер*, тя позволява изпълнение на графични програми в мрежа от различни по тип и операционни системи компютри.

На базата на тази и подобни на нея системи като Apple Macintosh и MS-Windows се създава спецификация за това как трябва да се води графичният диалог - от ресурсите, с които се работи (като бутони, списъци, менюта, прозорци и др.) и вида, в който те се изобразяват, до начина, по който те се организират за представяне на определен тип информация или пък за получаването на такава от потребителя. Целта е всички приложни програми да изглеждат по подобен начин, за да се минимизира времето, за което един потребител започва ефективно да работи с една нова програма. Това води до създаване на спецификации за организирането на графичното взаимодействие (и редица реализации) като XUI, OpenLook, OSF/Motif, InterViews и много други. Разработват се и цели операционни системи, в които описаните средства стават неделима част: Apple Macintosh, OS/2 и Windows-NT.

Системите за работа в прозорци се концентрират върху систематизиране на интерактивните възможности и се ограничават с визуализация, която е растерна и двумерна. При тях липсват сегменти или структури, потребителски координатни системи и елементи от моделирането. Не след дълго идва и съчетаването на тези системи с първите, които имат тези възможности. Появява се спецификацията и някои реализации на системата PEX (*PHIGS Extension to X Window System*), чиято цел е да съчетае възможностите за моделиране на PHIGS и интерактивните средства на X Window System.

Друго направление в стандартизацията става обменът на графична ин-

формация. Реализираният в GKS метафайл се оказва недостатъчен за обмен на данни между такива приложни програми като системите за автоматизирано проектиране и чертане (*Computer-Aided Design and Drafting Systems* - CADD). Появяват се други стандарти като IGES (*Initial Graphics Exchange Specification*), който има може би най-много реализации, но много рядко съответстващи на пълната спецификация. Наред с тези стандарти започва да се обръща особено внимание и на съхраняването на растерни изображения (GIF, TIFF, JPEG), както с компресиране, така и с възможност за представяне на изменения за нуждите на анимацията - MPEG.

Основният недостатък при описанието на един геометричен модел се оказва невъзможността да се опишат цялостно връзките между отделните елементи в геометричния модел, както и екземпляри от отделни негови етапи на създаване. Опит да се направи това е разработването на стандарта Step, някои части от който вече са приети официално, какъвто е например езикът за описание на продукти - EXPRESS.

1.4 ПРИЛОЖЕНИЯ НА КОМПЮТЪРНАТА ГРАФИКА

Достъпността на растерните дисплеи значително разшири обхвата на приложенията на компютърната графика. Днес те варират от визуализация на икономическа, статистическа, математическа и физическа информация до споменатите по-горе системи за автоматизирано проектиране. Тук ще изброим само по-важните от тях:

- **Визуализация на неграфична информация:** (*Presentation graphics*) Икономическите, статистическите, демографските и др. модели по принцип не са графични, нито имат отношение към геометрията. Визуализиране на резултатите от много анализи и изчисления във вид на схеми, диаграми, карти и др. се оказва много полезно за по-бързото възприемане на информацията, която тези резултати представят.
- **Визуализация на резултати от изследвания:** (*Scientific visualisation*) Тези приложения показват графично резултати от изследвания или поведение на обекти, в които формата играе важна роля. Такива са визуализацията на абстрактни математически структури, векторни полета, деформация на механични структури, поток на флуиди, ядрени и химически реакции, физиологични системи, функциониране на органи и др.
- **Управление на процеси:** (*Process control and simulation*) Тези приложения не само визуализират поведението на определени реални или моделирани процеси, но дават възможност и потребителят да ги управлява. Управление на работата на заводи, електростанции, комуникационни мрежи, космически кораби, компютърни мрежи са само някои от тези приложения. Особено място заемат симулираните процеси, каквито са например тренажорите за летци, компютъризираното извършване на хирургически операции и др.
- **Автоматизирано проектиране и производство:** (*Computer-Aided Design and Manufacturing* - CAD/CAM) Това е може би най-важното приложение на компютърната графика и геометричното моделиране. След първите системи за автоматизиране на чертожната дейност, днес целта е автоматизиране на целия цикъл на проектиране и производство в области като машиностроенето (включващи автоматизираното производство на детайлите и сглобяването им чрез управление на роботи), автомобилостроенето, самолетостроенето (включващи симулация на обтичане с флуиди и деформация на формата), текстилната и обувна промишленост, електрониката (с тестване на работата на интегрални схеми и печатни платки), електротехниката (симулация работата на електрически инсталации), строителството (анализ на деформациите и оребряването на конструкции) и много други.
- **Бюротика:** (*Office automation*) С навлизането на персоналните компютри във всеки кабинет много хора използват техните средства за текстообработка, управление на документи, планиране на дейностите, счетоводство, комуникация и електронни конференции, предпечатна подготовка и др.
- **Изкуство, реклама и анимация:** (*Computer art*) Средствата на компютърната графика могат да се използват както за създаването на компютърни картини, така и при създаване на междинни сцени в мултипликационните филми. Много от съвременните игрални филми използват компютърно генерирани сцени и образи.
- **Геодезия и картография:** (*Geographic Information Systems - GIS*) Приложенията на компютърната графика в геодезията и картографията започват да имат все по-голям дял в нея напоследък. Спецификата на геометричната информация и извънредно големият обем данни водят до създаването на специални системи за обработката им. Тук ще спомена приложения като създаването на схематични и точни географски карти, градски кадастър, географските информационни системи, подготовка на информация за сондиране, метеорологични системи, океанографски системи и др.

КЛАСИФИКАЦИЯ НА ПРИЛОЖЕНИЯТА. Изброените приложения могат да бъдат класифицирани по най-различни критерии. Най-често те се отличават едно от друго по размерността на обектите, с които се работи. В този смисъл приложните програми се разделят на двумерни и тримерни. Двумерните приложни програми могат от своя страна да се разделят на такива, които визуализират контури - отсечки и дъги; на такива, които визуализират черно-бели запълнени плоски фигури и най-сетне на такива, които работят с цветни области. Някои тримерни системи могат да работят с пространствени обекти, които се състоят само от ребра, а в други обектите могат да са съставени от плоски стени. В трети пространствените обекти могат да имат за граници произволни повърхнини.

Друга класификация се прави в зависимост от това, дали тези обекти са статични или динамични. Динамичните обекти имат фактически една размерност повече от статичните. Обектите също така могат да бъдат реални - да са

модели на реално съществуващи продукти или на такива, които предстои да бъдат произведени. Те могат също така да бъдат напълно абстрактни, каквито са някои математически структури или компютърно синтезираните картини.

Много важно разделение се прави между различните приложения в зависимост от ролята на визуалните образи в тях. В някои приложения като визуализацията на данни, картографията, изкуството и рекламата, визуализираният обект е крайният продукт на съответните програми. В други приложения като автоматизираното проектиране и управлението на процеси, визуализираният обект само представя някаква геометрична информация, която е достатъчна на потребителя да вземе решение за по-нататъшните си действия.

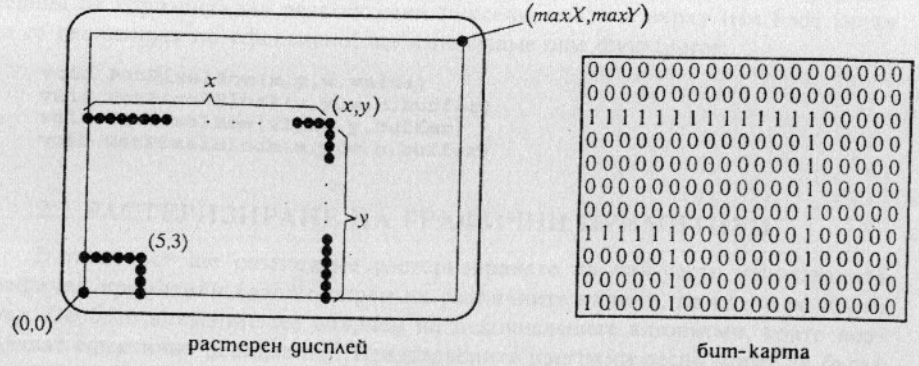
Приложенията могат да се класифицират и по вида на тяхната интерактивност - нуждата им от взаимодействие с потребителя. Тук възможностите са много, вариращи от приложни програми, в които потребителят задава поредица от данни и след това получава визуалния образ на резултата до т.нар. *интерактивно проектиране*, при което потребителят започва работа върху празен екран, създава нови графични елементи, включва стари такива, асемблира ги и ги структурира, така че в резултат да получи образа на желания обект. Едни от най-силно интерактивните приложения са именно системите за автоматизирано проектиране, които са най-тясно свързани с компютърната графика и геометричното моделиране.

Глава 2

Растерна графика

Алгоритмите за растерна графика имат за цел да обслужват визуализацията върху растерни устройства. Необходимостта от специални алгоритми за генериране и манипулиране на растерни изображения се налага поради широкото използване на растерните дисплеи за силно динамични задачи като анимация, симулация в реално време, т.нар. *симулация на реална среда* и др. Тези задачи изискват изпълнение на операциите с голяма скорост, която се постига и когато самите алгоритми са много ефективни и отчитат особеностите на растерните дисплеи. Търсенето на бързодействие е причината и едно от съвременните направления в изследванията в тази област да е създаването на паралелни алгоритми за растерна графика.

В тази глава ще разгледаме най-често използваните алгоритми за генериране на растерни изображения на основните геометрични обекти (графичните примитиви) - отсечка, окръжност, дъга и текстов символ, както и начините за запълване на области, зададени с многоъгълници и окръжности. Ще се спрем накратко на методите за отстраняване на стъпаловидността на обектите (antialiasing), удебеляването и използването на типове линии и образци за запълване. Тук не е отделено внимание на такива теми като растерни трансформации, филтриране, съхраняване и компресиране на растерни изображения, тъй като те са свързани повече с обработката, а не с генерирането на изображения.



Фиг. 2-1

При представянето на растерните дисплеи по-горе казахме, че изображението се описва в т.нар. *бит-карта* или *пикселна карта*, която е една правоъгълна матрица. Елементите на тази матрица (пикселите) са фиксиран брой битове, които задават цвета и интензитета на съответната точка от екрана. Адресацията на всеки пиксел в тази растерна матрица става чрез двойка целочислени координати, всяка от които варира във фиксиран целочислен интервал [0,*maxX*] и [0,*maxY*] съответно. Числата *maxX*+1 и *maxY*+1 са различни за всеки растерен дисплей и отразяват броя на пикселите по всяка от координатните оси.

Координатната система, определена върху растерната мрежа за повечето растерни дисплеи е с координатно начало в горния ляв ъгъл на растера (съответно и на екрана). Навсякъде в тази книга ние обаче ще считаме, че координатната система на растера е декартова, т.е. координатното начало е в долния ляв ъгъл на устройството.

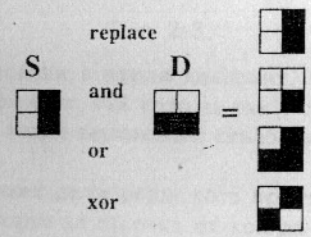
От гледна точка на програмиста обслужването на едно растерно устройство е сравнително елементарно. Пикселите се адресират с техните декартови координати и във всеки пиксел може да се запише дадена стойност, както и да се прочете тази, която вече е записана там. Записването на нова стойност в бит-картата става в няколко различни режима. Режимът определя типа на булевата операция, която се извършва между стойността **S**, която предстои да се запише и стойността **D**, която пикселът има преди записването.

$D \leftarrow S$ BoolOp **D**

От 16-те възможни булеви операции се използват най-често само:

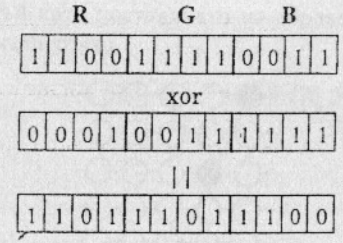
- **replace**: стойността, която пикселът получава е точно тази, която се записва, а предишното състояние на пиксела се игнорира;
- **and**, **or**, **xor**: стандартните булеви "и", "или" и "изключващо или".

В огромна част от случаите записването в бит-картата се извършва с операцията **replace**. Затова и в повечето растерни системи се използва *текущ режим на записване*, вместо изрично да се задава операцията, с която всеки пиксел се модифицира. За растерни дисплеи, пикселите на които могат да имат само една от двете стойности 0 и 1 ("черна" и "бяла"), режимите за записване могат да се илюстрират така:



Фиг. 2-2

При растерни дисплеи с *n* състояния за всеки пиксел булевите операции се извършват побитово. Долният пример показва това за една система, в която за всеки от основните цветове (червен - R, зелен - G и син - B) са отделени по четири бита:



Споменатите режими се използват в следните случаи:

replace: за записване на растеризацията на примитив, както и за изтриването му;

and: за избирателно изтриване на пиксели в дадена правоъгълна област (блок от пиксели);

or: за добавяне на блок от пиксели към изображението без да се изтриват тези от тях, за които в блока има записана 1;

xor: за инвертиране на образа.

При повторно изпълнение на последната операция образът възвръща началното си състояние: $\mathbf{D} \equiv \mathbf{S}$ xor (**S** xor **D**). Това е важно свойство, което се използва в много интерактивни похвати, както ще видим в четвърта глава. За нуждите на тази глава можем да смятаме, че разполагаме със следния набор от функции, с които се осъществява казаното дотук:

```
void SetWritingMode(REPLACE|AND|OR|XOR)
int  GetPixel(x,y)
void PutPixel(x,y,value)
```

Последната функция извършва записването на стойността `value` в зададения пиксел съобразно текущия режим на записване. Пикселите в пикселната карта са разположени по редове и затова е подходящо и обработката на група от съседни пиксели да става по редове. Тъй като операциите върху поредица от хоризонтално разположени пиксели, както и върху цял блок могат да се реализират по-ефективно, ще използваме още функциите:

```
void PutPixelRow(x,y,w,value)
void PutPixelBlock(x,y,w,h,buffer)
void GetPixelRow(x1,x2,y,buffer)
void GetPixelBlock(x,y,w,h,buffer)
```

2.1 РАСТЕРИЗИРАНЕ НА ГРАФИЧНИ ПРИМИТИВИ

В тази част ще разгледаме растеризирането на най-често използваните графични примитиви като се спрем на различните начини, по които се прави това. Особено внимание ще обърнем на целочислените алгоритми, които позволяват ефективна реализация. Представените програми лесно могат да бъдат оптимизирани, използвайки ефективни езикови конструкции за постигане на

максимално бърза визуализация, което често е основна цел при разработването на интерактивни системи. Тук целта е по-скоро яснота, затова и не всички програми са написани оптимално.

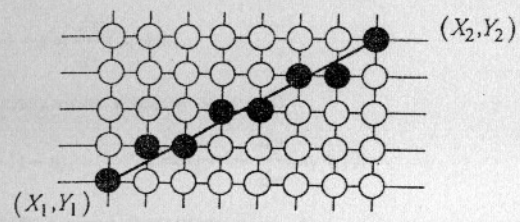
2.1.1 Растеризиране на отсечка

Основната задача при растеризирането на отсечка е да се намерят координатите на точките от растера, които лежат най-близо до зададената отсечка и съвкупността от които дава най-добро визуално приближение за нея. Нека една отсечка е зададена с координатите на началната и крайната си точки (X_1,Y_1), (X_2,Y_2), които принадлежат на растера. Уравнението на една такава невертикална отсечка тогава би било:

$$y = m(x - X_1) + Y_1, \qquad m = \frac{dy}{dx} = \frac{Y_2 - Y_1}{X_2 - X_1} \qquad [2.1]$$

РАСТЕРИЗИРАНЕ СЪС ЗАКРЪГЛЯВАНЕ. Ако отсечката има наклон, близък до хоризонтален, т.е. $-1 < m < 1$, то във всеки вертикален стълб на растера между двата ѝ крайни пиксела ще има само по един пиксел от нея. За да намерим всеки от тези пиксели, можем да даваме на x стойности X_1, X_1+1, X_1+2,..., X_2, при което от [2.1] за y ще получим последователността Y_1, Y_1+m, Y_1+2m,..., Y_2. Тъй като m е реално число, то и всички числа от тази редица ще бъдат реални. За да намерим целочислената y-координата, ще трябва да закръглим всяко от реалните числа до най-близкото цяло. Тогава отсечката би могла да бъде представена с пикселите:

$$(x_i, y_i), \qquad x_i = X_1 + i, \qquad y_i = \lfloor Y_1 + i \cdot m + 0.5 \rfloor, \qquad i = 0,1,\ldots,dx$$

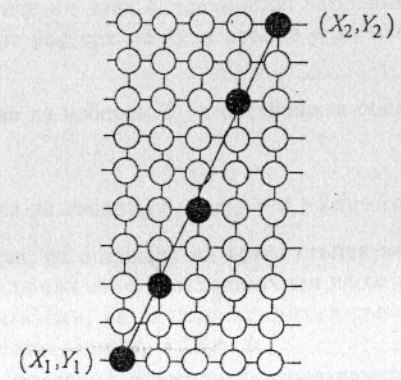


Фиг. 2-3

Растеризацията на отсечки в първи квадрант, чиито наклон е $m > 1$, по този начин би била неприемлива, тъй като за тях y се мени по-бързо от x и не можем да твърдим, че във всеки вертикален стълб има само по един пиксел от отсечката (фиг. 2-4).

Този проблем лесно може да се реши като просто се разменят местата на x и y. И накрая за да обобщим за отсечка от кой да е квадрант е необходимо да отчитаме, че стъпката с която се изменя i може да бъде и отрицателна (когато $X_2 < X_1$). Би било добре да отбележим, че не е желателно да се разменят началната и крайната точки на отсечката при положение, че $X_2 < X_1$ с цел винаги да растеризираме с положително нарастване по x. Проблем възниква

например при растеризиране на пунктирана отсечка, която е част от начупена линия. Тогава размяната на краищата ѝ би довела до нежелателно прекъсване на пунктира, тъй като поставянето на образеца е ориентирано винаги от началната към крайната точка.



Фиг. 2-4

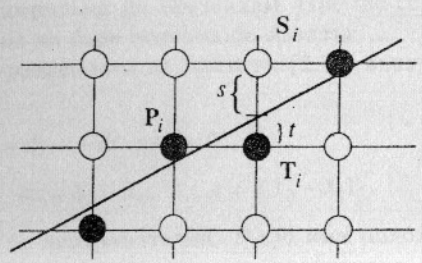
Следната функция ще осъществи растеризирането по описаната схема:

```
void SimpleLine(X1,Y1,X2,Y2,value)
int X1,Y1,X2,Y2; int value;
{int d,dx,dy,x,inty;
 int incX,n,reverse; float y, incY;
    dx=abs(X2-X1); dy=abs(Y2-Y1);
    if (reverse=(dx<dy)) {
       d=X1; X1=Y1; Y1=d;   /* разменяме местата на X и Y */
       d=X2; X2=Y2; Y2=d;   /* ако наклонът е по-голям от */
       d=dx; dx=dy; dy=d;   /* 45 градуса */
    }
    incX=(X1<=X2)?1:-1;     /* нарастването по X */
    incY=((float)dy)/dx;    /* може да е отрицателно */
    x=X1; y=Y1;
    n=dx+1;
    while (n--) {
          inty=(int)y;      /* закръгляване на реалното Y */
          if (reverse) PutPixel(inty,x,value);
                  else PutPixel(x,inty,value);
          x+=incX; y+=incY;
    }
}
```

Използването на числа с плаваща запетая не е най-ефективният начин за растеризиране на отсечка. Естествено е да търсим алгоритъм, който работи само с цели числа, тъй като и всички координати на пиксели са целочислени.

АЛГОРИТЪМ НА БРЕЗЕНХАМ ЗА ОТСЕЧКА. Следният алгоритъм, предложен от Брезенхам (Bresenham) през 1965 год. за управление на плотер при растеризиране на вектори е на практика един от най-често прилаганите целочислени алгоритми. Идеята на този алгоритъм е следната:

Нека за простота разгледаме отсечка, чийто наклон е $0 < m < 1$. Да видим как се осъществява избора на точка от растера на i-тата стъпка в този алгоритъм (фиг. 2-5). Ако на дадена стъпка е била избрана точката $\mathbf{P}_i = (x_i, y_i)$, то следващият избор за отсечка с такъв наклон може да бъде или $\mathbf{T}_i = (x_i+1, y_i)$ или $\mathbf{S}_i = (x_i+1, y_i+1)$.



Фиг. 2-5

Ясно е, че ако $(t-s) < 0$, то трябва да изберем $\mathbf{T}_i$, а в противен случай - $\mathbf{S}_i$. Това означава, че можем да използваме тази разлика в качеството на "оценка" за избор на следващия пиксел на всяка стъпка от растеризирането на отсечката. Тя за съжаление е отново реално число, но ние ще покажем, че можем да намерим друга *целочислена* оценка, за която можем да изведем рекурентна зависимост и че тази целочислена оценка е пряко свързана с разликата $(t-s)$.

За да опростим, нека да транслираме отсечката, зададена с уравнение [2.1] така, че началната точка (X_1, Y_1) да съвпадне с (0,0). Тогава можем да запишем следното:

$$y_i + t = y_i + 1 - s = \frac{dy}{dx}(x_i + 1),$$

което би дало следните изрази за s и t:

$$t = \frac{dy}{dx}(x_i + 1) - y_i, \qquad s = y_i + 1 - \frac{dy}{dx}(x_i + 1).$$

За разликата $t-s$ ще получим:

$$t - s = 2\frac{dy}{dx}(x_i + 1) - 2y_i - 1.$$

Тъй като ще оценяваме дали тази разлика е положителна или не, вместо нея можем да разгледаме $dx(t-s)$, защото $dx > 0$. По този начин ще получим целочислена оценка. Нека означим с d_i тази оценка:

$$d_i = dx(t-s) = 2dy(x_i + 1) - 2y_i dx - dx. \qquad [2.2]$$

Използвайки горната формула, можем да запишем каква ще бъде оценката и на $(i+1)$-вата стъпка:

$$d_{i+1} = 2dy(x_{i+1} + 1) - 2y_{i+1}dx - dx.$$

За да получим рекурентна зависимост за оценката, нека извадим горните две равенства и използваме, че $x_{i+1} - x_i = 1$:

$$d_{i+1} = d_i + 2dy(x_{i+1} - x_i) - 2dx(y_{i+1} - y_i) = d_i + 2dy - 2dx(y_{i+1} - y_i).$$

Този резултат вече ни дава основание да направим следното заключение за избора на точка от растера на тази стъпка и за извеждането на оценката за следващата:

Ако $d_i \le 0$, трябва да изберем $\mathbf{T}_i$, а следващата оценка ще е:

$$d_{i+1} = d_i + 2dy.$$

Ако $d_i > 0$, трябва да изберем $\mathbf{S}_i$ и $d_{i+1} = d_i + 2dy - 2dx$.

От горното следва, че оценката на всяка стъпка може да се пресмята целочислено, което изключва вече необходимостта да се използват числа с плаваща запетая. Използвайки, че началният пиксел съвпада с (0,0), от [2.2] за началната оценка ще получим: $d_0 = 2dy - dx$.

Показаната тук програма илюстрира използването на този алгоритъм в общия случай. Допълнителното, което е направено, за да се растеризира една произволно наклонена отсечка е следното:

- да се осигури $dx > 0$ и $dy > 0$;
- да се разменят координатите на крайните пиксели, както и на тези, които се получават при растеризирането, ако отсечката има наклон по-голям от 45 градуса;
- да се предвиди възможността x и y да намаляват вместо само да нарастват - в случай, че някоя от разликите $(X_2 - X_1)$ или $(Y_2 - Y_1)$ е отрицателна.

```
void BresenhamLine(X1,Y1,X2,Y2,value)
int X1,Y1,X2,Y2; int value;
{int x,y,dx,dy,incX,incY;
 int d,incUP,incDN,reverse,n;
    dx=abs(X2-X1);
    dy=abs(Y2-Y1);
    if (reverse=(dx<dy))
        ExchangeXY(X1,X2,dx,Y1,Y2,dy);
    incUP=-2*dx+2*dy;       /* нарастване при избор на S */
    incDN= 2*dy;            /* нарастване при избор на T */
    incX=(X1<=X2)?1:-1;
    incY=(Y1<=Y2)?1:-1;
    d=-dx+2*dy;
    x=X1; y=Y1;
    n=dx+1;
    while (n--) {
          if (reverse) PutPixel(y,x,value);
                  else PutPixel(x,y,value);
          x+=incX;
          if (d>0) { d+=incUP; y+=incY;
          } else     d+=incDN;
    }
}
```

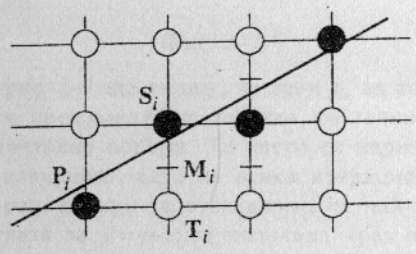
АЛГОРИТЪМ НА СРЕДНАТА ТОЧКА. Ван Аакен (Van Aken) разработва един интересен подход към растеризирането - т.нар. *принцип на средната точка* - използването на който позволява създаването на алгоритми за целочислено растеризиране не само на отсечки и окръжности, но и на конични сечения. Прилагането на този принцип за отсечки води до получаването на същия код като при алгоритъма на Брезенхам. Ние ще се спрем на тази формулировка защото тя ще ни бъде полезна по-нататък.

Друга форма на уравнението на правата [2.1], върху която лежи отсечката е следната:

$$F(x,y) = ax + by + c = dy \cdot x - dx \cdot y + c = 0, \qquad [2.3]$$

където $dy = Y_2 - Y_1$; $dx = X_2 - X_1$; $c = Y_1X_2 - X_1Y_2$.

Тъй като dx и dy са неотрицателни, $F(x,y)$ има положителна стойност за всяка точка разположена под отсечката и отрицателна за точките над нея. Ако на i-тата стъпка в този алгоритъм е била избрана точката $\mathbf{P}_i = (x_i, y_i)$, то може да се каже, че следващият избор - $\mathbf{T}_i = (x_i+1, y_i)$ или $\mathbf{S}_i = (x_i+1, y_i+1)$ зависи от положението на средната им точка $\mathbf{M}_i = (x_i+1, y_i+1/2)$ - фиг. 2-6.



Фиг. 2-6

Положението на $\mathbf{M}_i$ може веднага да се определи като се пресметне $F(\mathbf{M}_i)$ от [2.3] – $F(x_i+1, y_i+1/2)$. Това означава, че е удобно да изберем за оценка числото:

$$d_i = F\left(x_i + 1, y_i + \frac{1}{2}\right) = dy \cdot (x_i + 1) - dx \cdot \left(y_i + \frac{1}{2}\right) + c. \qquad [2.4]$$

Можем да кажем, че ако $d_i \le 0$, трябва да изберем $\mathbf{T}_i$, а ако $d_i > 0$, ще изберем $\mathbf{S}_i$. При това, ако изберем $\mathbf{T}_i$, т.е. $\mathbf{P}_{i+1} = \mathbf{T}_i$ то оценката на $(i+1)$-вата стъпка ще е:

$$d_{i+1} = F\left(x_i + 2, y_i + \frac{1}{2}\right) = dy \cdot (x_i + 2) - dx \cdot \left(y_i + \frac{1}{2}\right) + c. \qquad [2.5]$$

Рекурентната зависимост ще получим, като извадим d_i от d_{i+1} зададени с [2.4] и [2.5]: $d_{i+1} = d_i + dy$.

Ако пък изберем $\mathbf{S}_i$, т.е. $\mathbf{P}_{i+1} = \mathbf{S}_i$ то новата оценка ще е:

$$d_{i+1} = F\left(x_i + 2, y_i + \frac{3}{2}\right) = dy \cdot (x_i + 2) - dx \cdot \left(y_i + \frac{3}{2}\right) + c = d_i + dy - dx.$$

Остава да намерим стойността в първата средна точка, която е:

$$d_0 = F\left(x_0 + 1, y_0 + \frac{1}{2}\right) = dy \cdot (x_0 + 1) - dx \cdot \left(y_0 + \frac{1}{2}\right) + c =$$

$$= F(x_0, y_0) + dy - \frac{1}{2}dx = dy - \frac{1}{2}dx$$

За да избегнем дробта в горната формула, можем да умножим навсякъде по 2, което не би променило нищо в разсъжденията ни, но би позволило да използваме целочислена оценка. С други думи, можем да приемем, че уравнението на правата е:

$$F(x,y) = 2(ax + by + c) = 2(dy \cdot x - dx \cdot y + c) = 0. \qquad [2.6]$$

Получихме същия резултат като в алгоритъма на Брезенхам, но чрез различни разсъждения. Те дават по-ясна представа за смисъла на оценката, която въвеждаме и използваме на всяка стъпка от алгоритъма. Нещо повече, тази оценка може да бъде използвана в алгоритмите за отстраняване на стъпаловидността на отсечките, както ще видим по-късно.

АЛГОРИТЪМ НА ПОРЦИИТЕ. Разгледаните дотук алгоритми решават задачата за намиране на една от координатите на един пиксел като знаем другата му координата, така че този пиксел да е най-близко разположен до растеризираната отсечка. В този смисъл всяка итерация води до намирането само на един пиксел от растеризацията.

Ако разгледаме една отсечка в първи октант (т.е. в тази част от I квадрант, в която $x > y$), ще видим, че растерният ѝ образ се състои от поредица от хоризонтални участъци (фиг. 2-7). Тези хоризонтални участъци Брезенхам нарича *порции*. Един друг подход към растеризирането на отсечка е да се получат всички такива порции, като на всяка итерация в алгоритъма се намира съответната поредица от пиксели, а не само един единствен.

Нека за удобство да разгледаме отсечка от първи октант с начало точката (0,0) и крайна точка (H,V). Алгоритъмът на средната точка на практика решава системата от $H+1$ неравенства:

$$y - \frac{1}{2} < \frac{V}{H} \cdot x \le y + \frac{1}{2}, \qquad x = 0,1,2,\ldots,H \qquad [2.7]$$

относно неизвестните y. Нека да модифицираме тази система като я решим относно x. Тя ще се преобразува в система от $V+1$ неравенства спрямо x:

$$\frac{H}{2V}(2y-1) < x \le \frac{H}{2V}(2y+1), \qquad y = 0,1,2,\ldots V,$$

което може да се запише и като:

$$\frac{2H}{2V}y - \frac{H}{2V} < x \le \frac{2H}{2V}y + \frac{H}{2V}, \qquad y = 0,1,2,\ldots V \qquad [2.8]$$

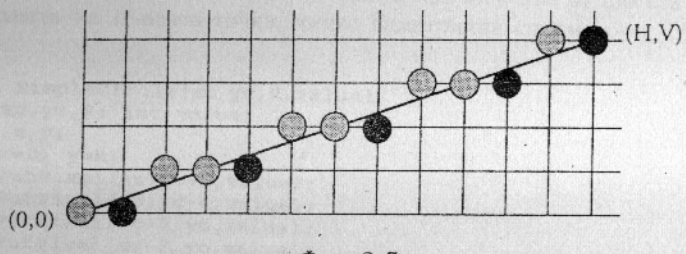
Нека да представим дробите в горната система от неравенства като цяла

част и остатък по модул $2V$, за да можем да извършим нейното решаване в цели числа. Ще получим следното:

$$\frac{H}{2V} = c_0 + \frac{r_0}{2V}, \qquad \frac{2H}{2V} = c_1 + \frac{r_1}{2V} \qquad [2.9]$$

Използвайки [2.9], системата [2.8] може да бъде записана по следния начин:

$$c_1 y - c_0 + \frac{r_1 y - r_0}{2V} < x \le c_1 y + c_0 + \frac{r_1 y + r_0}{2V}, \qquad y = 0,1,2,\ldots V \qquad [2.10]$$



Фиг. 2-7

Ако разгледаме фиг. 2-7 ще видим, че тези x, за които десните неравенства в тази система се превръщат в равенства, са точно крайните десни пиксели във всяка хоризонтална порция. Те често се наричат *преходни пиксели*. Растеризирането ще извършим, като на всяка итерация намираме преходния пиксел в съответния ред. Можем за отбележим, че най-малкото x, което удовлетворява неравенствата за $y = j+1$, се получава чрез прибавяне на единица към x-координатата на преходния пиксел в неравенството за $y = j$.

Нека разгледаме кога се получава равенство в дясното неравенство от [2.10] за $y = 0$. При решаване на уравнението

$$x = c_1 y + c_0 + \frac{r_1 y + r_0}{2V}$$

в цели числа ще видим, че първият преходен пиксел има координати точно $(c_0, 0)$.

Всеки следващ преходен пиксел може да бъде получен използвайки същото уравнение след предварително изчисляване на коефициентите в целочисленото разлагане от [2.9]. За целта е необходимо само да въведем една променлива (в представената програма сме използвали името mod), в която да съхраняваме стойността на числителя $r_1 y + r_0$ на дробта в споменатото уравнение и да коригираме цялата част на x всеки път когато тази дроб стане по-голяма от единица.

В така предложения алгоритъм могат да се направят допълнителни подобрения за намаляване на броя на операциите и дори за намаляване на итерациите. Ние ще предоставим възможност на читателя да подобри представения вариант. Много по-сериозно подобрение е да се извлече закономерността в

образуването на порциите, за намирането на която напоследък има разработени интересни алгоритми. В тях основно се анализира делимостта на числата H и V.

```
void DrawBresenhamSliceLine(H,V,value)
int H,V,value;
(int r1,c1,y;
 int startX,              /* началната точка на порцията */
 int endX,                /* координатата на преходната точка */
 int mod;                 /* числителят в дробта за изчисляване на x */
   r1 = (H+H)%(V+V);
   c1 = (H+H-r1)/(V+V);
   y = 0; startX = 0;
   mod = H%(V+V);
   endX = (H-mod)/(V+V);
   while(1) {
     if (endX<H) {        /* записване на порцията */
        PutPixelRow(startX,endX,y,value);
     } else {             /* това е последната порция */
        PutPixelRow(startX,H,y,value);
        return;
     }
     startX = endX+1;
     endX += c1;
     mod += r1;           /* корекция на числителя */
     if (mod>V+V) { endX++; mod -= V+V; }
     y++;
   }
}
```

2.1.2 Растеризиране на окръжност

Задачата за растеризиране на окръжност е почти толкова важна в компютърната графика, колкото и тази за растеризиране на отсечка поради често налагащото се визуализиране на окръжности. Въпреки, че са разработени алгоритми, приложими за всякакви плоски криви с уравнения $F(x,y) = 0$ с непрекъснати първи производни окръжността заслужава специално внимание поради симетричността си. Първо нека разгледаме стандартния алгоритъм, използващ числа с плаваща запетая, след което ще се спрем по-подробно и на целочислените.

Без да ограничаваме общността навсякъде по-долу ще считаме, че работим с централна окръжност: $x^2 + y^2 = R^2$, чийто радиус е цяло число.

РАСТЕРИЗИРАНЕ СЪС ЗАКРЪГЛЯВАНЕ. Нека най-напред разгледаме само тази част от една централна окръжност, която лежи в първи квадрант. За нея лесно можем да получим явен израз за y:

$$y = \sqrt{R^2 - x^2}, \qquad x \in [0, R].$$

Ако даваме на x последователно стойности 0,1,...,R и намираме y от горното уравнение, след закръгляването му до най-близкото цяло число ще имаме последователност от точки на растера, които апроксимират окръжността.

Както може да се очаква, втората половина от тази четвърт-окръжност изглежда неприемливо поради голямото разстояние между апроксимиращите точки (фиг. 2-8). Това лесно може да се преодолее като се използва симетрията на окръжността относно координатните оси и правите $x=y$ и $x=-y$.

Използвайки тази симетрия може да се генерира само 1/8 от окръжността по този алгоритъм, както е показано с програмата по-долу. В нея е отделен случаят $x = y$, защото тогава за всяка от съответните точки ще има по две обръщения към функцията PutPixel, което в режим на записване **xor** (изключващо "или") би довело до различна осветеност на тези точки. Забележете, че е възможно пикселът с координати (x,y), когато $x = y$ да не е точка от растеризацията на окръжността. По същата причина вън от цикъла е изнесено и записването на първата точка, чиято симетрична спрямо оста Oy е същата точка.

```
void SimpleCircle(xc,yc,R,value)
int xc,yc,R; int value;
{int x,y;
    x=0; y=R;
    PutPixel(xc,yc+R,value);
    PutPixel(xc,yc-R,value);
    PutPixel(xc+R,yc,value);
    PutPixel(xc-R,yc,value);
    while (x<y) {
        x++;
        y=(int)sqrt((double)(R*R-x*x));
        EightSymmetric(xc,yc,x,y,value);
    }
    if (x==y) FourSymmetric(xc,yc,x,y,value);
}

void EightSymmetric(xc,yc,x,y,value)
int xc,yc,x,y; int value;
{   FourSymmetric(xc,yc,x,y,value);
    FourSymmetric(xc,yc,y,x,value);
}

void FourSymmetric(xc,yc,x,y,value)
int xc,yc,x,y; int value;
{   PutPixel(xc+x,yc+y,value);
    PutPixel(xc-x,yc-y,value);
    PutPixel(xc-x,yc+y,value);
    PutPixel(xc+x,yc-y,value);
}
```

Поради наличието на умножение и извличане на квадратен корен, представеният алгоритъм не е много по-добър от този, който генерира последователност от стойности за координатите чрез вариране на параметъра θ в параметричното уравнение:

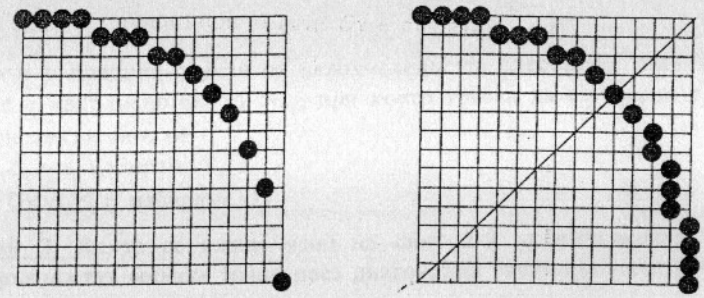
$$\begin{cases} x = X_C + R\cdot\cos\theta \\ y = Y_C + R\cdot\sin\theta \end{cases}$$

Споменатият начин не е за предпочитане не само поради наличието на тригонометрични функции, но и поради независимостта на параметъра от

растера. Използването на тригонометрични функции може да се избегне като се направи следната субституция: $t = \mathrm{tg}(\theta/2)$. Получената нова параметризация:

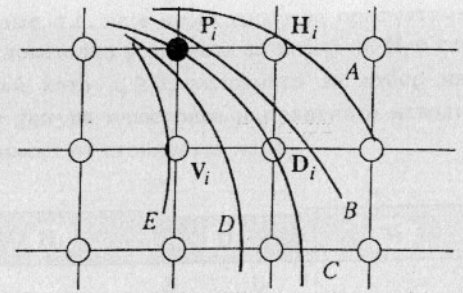
$$x = X_C + R\cdot\frac{1-t^2}{1+t^2} \qquad y = Y_C + R\cdot\frac{2t}{1+t^2}$$

може да се използва за решаването на някои геометрични задачи, но също като горната не може да се свърже директно с растера. Още повече разпределението на генерираните пиксели не е равномерно по протежение на нито една от координатните оси.



Фиг. 2-8

АЛГОРИТЪМ НА БРЕЗЕНХАМ ЗА ОКРЪЖНОСТ. Алгоритъмът на Брезенхам е целочислен и се стреми да даде оценка за грешката, която се прави на всяка стъпка при избор на една или друга апроксимираща точка от растера. Нека отново разгледаме само първата четвърт от централна окръжност, намираща се в първи квадрант. Да приемем, че на i-тата стъпка в този алгоритъм е била избрана точката $\mathbf{P}_i = (x_i, y_i)$. Следващият избор може да бъде само една от точките: $\mathbf{H}_i = (x_i+1, y_i)$, $\mathbf{V}_i = (x_i, y_i-1)$ или $\mathbf{D}_i = (x_i+1, y_i-1)$, показани на фиг 2-9. На същата фигура са показани възможните начини (A,B,C,D,E), по които разглежданата част от четвърт-окръжността може да е разположена спрямо точките от растера.



Фиг. 2-9

Ще въведем оценка за грешката, която се прави при избирането на диагонално разположената точка чрез ориентираното ѝ разстояние до истинската окръжност:

$$\Delta_i = D(\mathbf{D}_i) = (x_i+1)^2 + (y_i-1)^2 - R^2 \qquad [2.11]$$

Случай 1. Нека първо разгледаме какъв избор правим при $\Delta_i < 0$. Тогава окръжността е разположена както в случаите *A* и *B* от горната фигура и е естествено да се избере или $\mathbf{H}_i = (x_i+1, y_i)$, или $\mathbf{D}_i = (x_i+1, y_i-1)$. Ще въведем още една оценка, за да разграничим тези две възможности:

$$\delta_i = D(\mathbf{H}_i) + D(\mathbf{D}_i) = 2(x_i+1)^2 + (y_i-1)^2 + y_i^2 - 2R^2 \qquad [2.12]$$

а/ В случая *A* трябва да се избере винаги $\mathbf{H}_i$ и както се вижда от фигурата тогава $D(\mathbf{H}_i) \le 0$ и $D(\mathbf{D}_i) < 0$, а следователно и $\delta_i < 0$;

б/ В случай *B* е изпълнено $D(\mathbf{H}_i) > 0$ и $D(\mathbf{D}_i) < 0$ - т.е. двата члена на сумата [2.12] имат различни знаци и избраната точка би била $\mathbf{H}_i$ само ако $D(\mathbf{H}_i) \le D(\mathbf{D}_i)$.

От казаното тук можем да заключим, че при $\Delta_i < 0$ е необходимо разграничението:

- при $\delta_i < 0$ избираме $\mathbf{H}_i$
- при $\delta_i > 0$ избираме $\mathbf{D}_i$

Тук не използваме равенство, защото $\delta_i \neq 0$: уравнението $\delta_i = 0$, както ще видим по-късно, няма подходящо решение в цели числа. Нека сега запишем неравенството $\delta_i > 0$ като използваме [2.12], допълним до точния квадрат на $y_i - 1$ с прибавяне и изваждане на $(-2y_i + 1)$ и след това прегрупираме подходящо членовете му:

$$\delta_i = 2\left[(x_i+1)^2 + (y_i-1)^2 - R^2\right] + 2y_i - 1 = 2(\Delta_i + y_i) - 1 > 0,$$

като за израза в квадратните скоби използваме дефиницията [2.11]. Следователно можем да кажем, че в разглеждания случай ($\Delta_i < 0$) ще избираме $\mathbf{D}_i$ само ако:

$$\Delta_i > -y_i + \frac{1}{2}.$$

От това неравенство можем да заключим, че $\delta_i \neq 0$. Тъй като неизвестните в неравенството са цели числа, то би било изпълнено само ако $\Delta_i > -y_i$. Да обобщим този случай:

- при $\Delta_i \le -y_i$ избираме $\mathbf{H}_i$
- при $-y_i < \Delta_i < 0$ избираме $\mathbf{D}_i$

Случай 2. Нека сега приемем, че $\Delta_i > 0$. Това съответства на разположенията *D* и *E* от фиг. 2-9. Както в предния случай, да въведем една нова оценка, за да разграничим тези две възможности:

$$\varepsilon_i = D(\mathbf{D}_i) + D(\mathbf{V}_i) = (x_i+1)^2 + 2(y_i-1)^2 + x_i^2 - 2R^2 \qquad [2.13]$$

Аналогично анализирайки възможностите за избор при *D* и *E*, ще заключим, че:

- при $\varepsilon_i > 0$ избираме $\mathbf{V}_i$, а
- при $\varepsilon_i < 0$ избираме $\mathbf{D}_i$.

Тук отново можем да изпуснем знака за равенство.

Замествайки в първото неравенство с [2.13] и както преди допълвайки до точен квадрат с прибавяне и изваждане на $(2x_i + 1)$ получаваме:

$$\varepsilon_i = 2\left[(x_i+1)^2 + (y_i-1)^2 - R^2\right] + 2x_i - 1 = 2(\Delta_i - x_i) - 1 > 0,$$

или

$$\Delta_i > x_i + \frac{1}{2}$$

Както и в предния случай от целочислеността следва, че горното неравенство ще е изпълнено ако $\Delta_i > x_i$, при което трябва да се избере $\mathbf{V}_i$. Обобщавайки ще заключим, че:

- при $\Delta_i > x_i$ избираме $\mathbf{V}_i$, а
- при $0 < \Delta_i \le x_i$ избираме $\mathbf{D}_i$.

Случай 3. Както се вижда ясно на фиг. 2-9 (разположение *C*), при $\Delta_i = 0$, окръжността минава точно през диагонално разположената точка и тя именно трябва да бъде избрана.

След като разгледахме всички възможни случаи за избор на следваща точка на *i*-тата стъпка в растеризирането на окръжността, сега можем да обобщим трите разгледани възможности по следния начин:

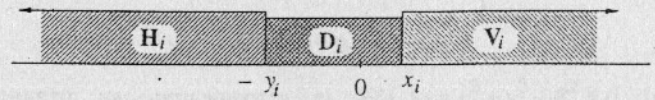
- Ако $\Delta_i < 0$ и $\Delta_i \le -y_i$ избираме точката $\mathbf{H}_i$;
- Ако $\Delta_i > 0$ и $\Delta_i > x_i$ избираме точката $\mathbf{V}_i$;
- Във всички останали случаи ще избираме $\mathbf{D}_i$

Ако вземем предвид, че $y_i \ge 0$ за всяка точка от първата четвърт на окръжността, условието за избор на $\mathbf{H}_i$ се свежда до следните две:

- $\Delta_i \le -y_i$ за $y_i > 0$ и
- $\Delta_i < 0$ за $y_i = 0$.

Но $y_i = 0$ само когато е достигнат краят на дъгата, а тогава не е необходимо да продължаваме, т.е. не е необходимо да пресмятаме оценката за следващата стъпка. Следователно условието за избор на $\mathbf{H}_i$ е само $\Delta_i \le -y_i$.

Аналогично, тъй като $x_i \ge 0$, условието за избор на $\mathbf{V}_i$ се свежда до $\Delta_i > x_i$. Следващата фигура илюстрира направените изводи за избор на следваща точка в зависимост от стойността на Δ_i:



Фиг. 2-10

За оценката Δ_i ще изведем лесно рекурентна зависимост по формулата [2.11]. В началото $x_0=0$ и $y_0=R$ и следователно $\Delta_0=2(1-R)$. На всяка стъпка новата стойност на оценката зависи от предишния избор:

а/Изборът на точката $\mathbf{H}_i$ означава, че $x_{i+1}=x_i+1$ и $y_{i+1}=y_i$, а оттам и оценката е:

$$\begin{aligned}\Delta_{i+1} &= (x_{i+1}+1)^2+(y_{i+1}-1)^2-R^2=\left[(x_i+1)+1\right]^2+(y_i-1)^2-R^2\\ &= (x_i+1)^2+(y_i-1)^2-R^2+2(x_i+1)+1\\ &= \Delta_i+2(x_i+1)+1=\Delta_i+2x_{i+1}+1\end{aligned} \qquad [2.14]$$

б/Изборът на $\mathbf{V}_i$ отговаря на $x_{i+1}=x_i$ и $y_{i+1}=y_i-1$, а оценката е:

$$\begin{aligned}\Delta_{i+1} &= (x_{i+1}+1)^2+(y_{i+1}-1)^2-R^2=(x_i+1)^2+\left[(y_i-1)-1\right]^2-R^2\\ &= (x_i+1)^2+(y_i-1)^2-R^2-2(y_i-1)+1\\ &= \Delta_i-2(y_i-1)+1=\Delta_i-2y_{i+1}+1\end{aligned} \qquad [2.15]$$

в/При избор на точката D_i - $x_{i+1}=x_i+1$ и $y_{i+1}=y_i-1$, а оттам и оценката е:

$$\begin{aligned}\Delta_{i+1} &= (x_{i+1}+1)^2+(y_{i+1}-1)^2-R^2=\left[(x_i+1)+1\right]^2+\left[(y_i-1)-1\right]^2-R^2\\ &= \Delta_i+2(x_i+1)-2(y_i-1)+1=\Delta_i+2x_{i+1}-2y_{i+1}+1\end{aligned} \qquad [2.16]$$

Следващата програма растеризира цялата окръжност по представения начин, като използва симетрията ѝ относно координатните оси. Случаят $y=0$ е изнесен вън от цикъла по същата причина, поради която направихме това за алгоритъма, използващ числа с плаваща запетая - за да се записва всеки пиксел само по веднъж.

Ако искаме да използваме осемстранната симетрия, бихме могли да растеризираме само една осма от окръжността. Тогава вместо три възможности за избор на точки от растера: $\mathbf{H}_i$, $\mathbf{D}_i$ и $\mathbf{V}_i$, ще имаме само две: $\mathbf{H}_i$ и $\mathbf{D}_i$. Оценката в такъв случай вместо Δ_i ще бъде δ_i, дефинирана чрез [2.12].

```
void DrawBresenhamCircle(xc,yc,R,value)
int xc,yc,R; int value;
{int x,y,d;
    x=0;y=R;
    d=2-2*R;
    PutPixel(xc,yc+R,value);
    PutPixel(xc,yc-R,value);
    PutPixel(xc+R,yc,value);
    PutPixel(xc-R,yc,value);
    while (1) {
          if (d >-y) {y--; d+=1-2*y;}
          if (d <=x) {x++; d+=1+2*x;}
          if (y) return;
          FourSymmetric(xc,yc,x,y,value);
    }
}
```


Използвайки метода на Брезенхам, Михенер (Michener) предлага алгоритъм само за една осма от окръжността с използване на тази именно оценка. Рекурентните зависимости се извеждат по аналогичен на горния начин, а началната ѝ стойност е:

$$\delta_0=2(x_0+1)^2+(y_0-1)^2+y_0^2-2R^2=2+R^2-2R+1+R^2-2R^2=3-2R,$$

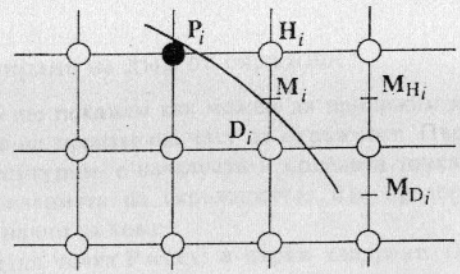
имайки пред вид, че началната точка е $(0,R)$.

Програмата, показана по-долу, осъществява растеризирането на пълна окръжност по този именно алгоритъм. В нея е възможно някои от пикселите да се запишат по два пъти. Читателят видя как може да се избягва това когато е необходимо в програмата `SimpleCircle` а и в по-горната реализация на алгоритъма на Брезенхам.

```
void DrawMichenerCircle(xc,yc,R,value)
int xc,yc,R; int value;
{int x,y,d;
    d=3-2*R; y=R;
    EightSymmetric(xc,yc,0,R,value);
    for (x=0; x<y; x++) {
         if (d >= 0) d+=10+4*x-4*(y--);
              else   d+= 6+4*x;
         EightSymmetric(xc,yc,x,y,value);
    }
}
```

АЛГОРИТЪМ НА СРЕДНАТА ТОЧКА ЗА ОКРЪЖНОСТ. Принципът на средната точка, който въведохме в 2.1.1 може да се приложи и при намиране на оценка за избор на следваща точка при дискретизация на окръжност. При това отново ще видим, че полученият алгоритъм почти не се различава от резултата на Брезенхам. Ще разгледаме растеризирането само на дъгата във втори октант на централна окръжност, т.е. когато x се мени от 0 до $R\sqrt{2}$.

Ако избраният пиксел на i-тата стъпка е $\mathbf{P}_i=(x_i,y_i)$, следващият избор може да бъде или $\mathbf{H}_i=(x_i+1,y_i)$ или $\mathbf{D}_i=(x_i+1,y_i-1)$. Отново ще оценим положението на средната точка, за да определим избора.



Фиг. 2-11

Уравнението на окръжността е: $F(x,y)=x^2+y^2-R^2=0$. Функцията $F(x,y)$ има положителни стойности за точките извън окръжността и отрица-

телни за тези във вътрешността ѝ. Ще вземем за оценка стойността на функцията в средната точка:

$$d_i = F\left(x_i+1, y_i+\frac{1}{2}\right) = (x_i+1)+\left(y_i+\frac{1}{2}\right)^2 - R^2.$$

При $d_i \geq 0$, трябва да изберем $\mathbf{D}_i$ и оценката на (i+1)-вата стъпка ще е:

$$\begin{aligned} d_{i+1} &= F\left(x_i+2, y_i-\frac{3}{2}\right) = (x_i+2)^2+\left(y_i-\frac{3}{2}\right)^2 - R^2 \\ &= (x_i+1)^2+\left(y_i-\frac{1}{2}\right)^2 - R^2 + 2(x_i+1) - 2\left(y_i-\frac{1}{2}\right)+2 \\ &= F\left(x_i+1, y_i-\frac{1}{2}\right)+2x_i-2y_i+5 = d_i+2x_i-2y_i+5 \end{aligned} \qquad [2.17]$$

Ако пък $d_i < 0$, трябва да изберем $\mathbf{H}_i$, а тогава новата оценка ще е:

$$\begin{aligned} d_{i+1} &= F\left(x_i+2, y_i-\frac{1}{2}\right) = (x_i+1)^2+\left(y_i-\frac{1}{2}\right)^2 - R^2 + 2(x_i+1)+1 \\ &= F\left(x_i+1, y_i-\frac{1}{2}\right)+2x_i+3 = d_i+2x_i+3 \end{aligned} \qquad [2.18]$$

Остава да намерим и началната стойност на оценката в първата средна точка:

$$d_0 = F\left(1, R-\frac{1}{2}\right) = 1-\left(R^2-R-\frac{1}{4}\right)-R^2 = \frac{5}{4}-R$$

Дотук разсъжденията не се отличават от тези направени в извеждането на алгоритъма за отсечка. За да избегнем използването на дроби можем да въведем нова оценка $e = d-\frac{1}{4}$, която има целочислена начална стойност и се изменя целочислено.

Сравненията трябва да се заменят с $e \geq \frac{1}{4}$ и $e < \frac{1}{4}$, но тъй като e е цяло число, то те могат да се извършват и спрямо 0. Забележете, че ако положим

$$e = 2d+\frac{1}{2}$$

и извършваме сравненията спрямо 0, а не спрямо -1/2, ще получим точно алгоритъма на Михенер, показан по-горе.

ЧАСТНИ РАЗЛИКИ ОТ ВТОРИ РЕД. Алгоритмите, които разгледахме дотук, използват частни разлики от първи ред. Така се наричат разликите в изменението на функцията от уравнението на примитива в две съседни точки. Когато уравнението на примитива е линейно (напр. на отсечката), тогава тези частни разлики са константи, но тъй като окръжността се задава с уравнение от втори ред, тези нараствания са линейни функции на x и y - [2.17], [2.18]. При растеризиране на окръжност е по-удобно да се използват частни

разлики от втори ред, които пък са разликите в изменението на първите частни разлики. В този случай те са константи.

Нека означим с d_i^H изменението на d_i при избор на $\mathbf{H}_i$, а с d_i^D нараства-нето на d_i при избор на $\mathbf{D}_i$. Това са всъщност и частните разлики от втори ред за окръжността. В точката $\mathbf{P}_i$:

$$d_i^H = 2x_i+3, \text{ а } d_i^D = 2x_i-2y_i+5. \qquad [2.19]$$

Когато се избира $\mathbf{H}_i$, въведените втори частни разлики променят стойностите си по следния начин:

$$\begin{aligned} d_{i+1}^H &= 2(x_i+1)+3 = d_i^H+2, \\ d_{i+1}^D &= 2(x_i+1)-2y_i+5 = d_i^D+2. \end{aligned}$$

Аналогично, при избор на $\mathbf{D}_i$ тези разлики са:

$$\begin{aligned} d_{i+1}^H &= 2(x_i+1)+3 = d_i^H+2, \\ d_{i+1}^D &= 2(x_i+1)-2(y_i-1)+5 = d_i^D+4. \end{aligned}$$

Началните стойности d_0^H и d_0^D могат да се определят от [2.19] за точката с координати $(0,R)$. Функцията за растеризиране на окръжност, която показваме тук, е написана съобразно изведените формули и използва алгоритъма на средната точка.

```
void DrawMidpoint2OrderDiffCircle(xc,yc,R,value)
int xc,yc,R; int value;
{int x,Y, d,dH,dD;
    d=1-R; y=R;
    dH=3; dD=5-2*R;
    EightSymetric(xc,yc,0,R,value);
    for (x=0; x<y; x++) {
        if (d<0) {d+=dH; dH+=2; dD+=2;}
          else  {d+=dD; dH+=2; dD+=4; y--;}
        EightSymetric(xc,yc,x,y,value);
    }
}
```

2.1.3. Растеризиране на дъга от окръжност

В този параграф ще покажем как можем да приложим метода на Брезенхам за растеризиране на произволна част от окръжност. Първото нещо, което е необходимо да си осигурим, е началната и крайната точка на дъгата да бъдат точки от растеризацията на окръжността. Ще представим един много прост, но ефективен начин за това:

Да разгледаме една точка $\mathbf{P} = (x,y)$ в първи квадрант. Отдалечеността на тази точка от окръжността (грешката, която се допуска, като се използва тя за апроксимираща точка) се задава като $e(P) = F(x,y) = x^2+y^2-R^2$. При изграждането на алгоритъма на Брезенхам в 2.1.2 видяхме, че грешката в съседните точки се задава като:

$$e(x+1,y) = e(x,y)+2x+1$$
$$e(x,y+1) = e(x,y)+2y+1$$

Да започнем да се движим от т. P в посока към окръжността правейки по една стъпка (само по x или само по y) и пресмятайки новата грешка дотогава докато тази грешка намалява. Посоката на стъпката ще зависи от знака на грешката - ако е положителна (точката е извън окръжността), трябва да се движим наляво и надолу, а ако е отрицателна - надясно и нагоре.

Естествено е да правим стъпка по тази координатна ос, по която бихме получили по-малка нова грешка. Например ако точката е във вътрешността на окръжността, тогава трябва да избираме между $(x+1,y)$ и $(x,y+1)$. Ще изберем $(x+1,y)$ ако:

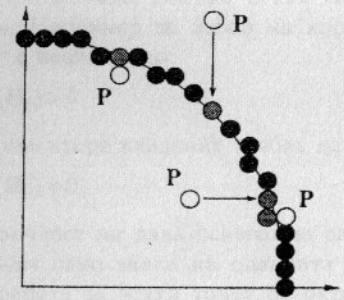
$$|e(x+1,y)| < |e(x,y+1)|$$

Нека приемем, че двете съседни точки, за които пресмятаме грешката, са също във вътрешността на окръжността. В такъв случай можем да се освободим от абсолютните стойности, тъй като и двете грешки ще бъдат отрицателни:

$$-e(x+1,y) < -e(x,y+1) \quad \Rightarrow$$
$$e(x+1,y) > e(x,y+1) \quad \Rightarrow$$
$$e(x,y)+2x+1 > e(x,y)+2y+1 \quad \Rightarrow \quad x > y$$

Аналогично ще изберем $(x,y+1)$ ако $x < y$.

Ние си опростихме задачата приемайки, че оценката няма да сменя знака си при движение към окръжността. Дори и с тази уговорка този метод дава добри резултати, ако точката е достатъчно близко до окръжността, което е вярно в огромната част от случаите (фиг. 2-12). Оставяме на читателя да докаже, че намерената по този начин точка винаги е част от растеризацията на окръжността. За по-отдалечени точки е необходимо да се предвиди придвижване и по диагонал. Представената функция позиционира една точка върху окръжността в общия случай, което налага да сравняваме абсолютните стойности на координатите. Използваният макрос `sign` дава знака на едно число, ако то не е 0, и 0 в противен случай.



Фиг. 2-12


```
void PutPointOnCentralCircle(x,y,R)
int *x,*y,R;
{int e,newe,dx,dy,dir;
    e=(*x)*(*x)-R*R+(*y)*(*y);
    dir=-sign(e);
    while (1) {
          if (abs(*y)>abs(*x)) {
              dx=0; dy=dir*sign(*y);
          } else if (*x) {
              dx=dir*sign(*x); dy=0;
          } else {
              dx=1; dy=0;
          }
          newe=e+2*(*x)*sign(dx)+2*(*y)*sign(dy)+1;
          if (abs(newe)>abs(e)) return;
          (*x)+=dx; (*y)+=dy;
          e=newe;
    }
}
```

За да извършим растеризирането на произволна част от окръжност е необходимо да следваме нарастването на ъгъла, без да използваме симетрията относно координатните оси, тъй като в общия случай дъгите не са симетрични.

Нарастването по x и по y е различно в различните квадранти, както това може да се види от стойностите на `incx` и `incy` в таблицата:

квадрант	*incx*	*incy*
I	-1	1
II	-1	-1
III	1	-1
IV	1	1

Оценката, която се използва в алгоритъма на Брезенхам е стойността на функцията в диагонално разположената точка спрямо текущо избраната, което може да се обобщи за произволен квадрант като:

$$\Delta_i = D(\mathbf{D}_i) = (x_i + incx)^2 + (y_i + incy)^2 - R^2$$

Чрез разсъждения подобни на тези за I квадрант, можем да обобщим и рекурентните зависимости за тази оценка:

$\Delta_{i+1} = \Delta_i + ddx$, $\quad ddx = 2 \cdot incx \cdot x + 1$, което е аналогично на [2.14]

$\Delta_{i+1} = \Delta_i + ddy$, $\quad ddy = 2 \cdot incy \cdot y + 1$, аналогично на [2.15] и

$\Delta_{i+1} = \Delta_i + ddx + ddy$, което пък е аналогично на [2.16].

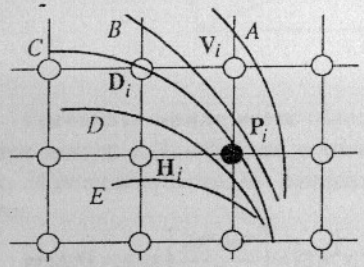
Тези нараствания съответстват на стъпка или само по x, или само по y, или по диагонал. Необходимо е да отбележим, че ролите на x и y в нашите разсъждения ще са разменени, защото растеризирането ще извършваме по посока на нарастването на ъгъла, което в I квадрант съвпада с нарастването на y и намаляването на x - обратно на това, което разгледахме в 2.1.2. Допълнителната оценка, която въведохме с [2.12] сега е:

$$\delta_i = 2(x_i+incx)^2+(y_i+incy)^2+x_i-2R$$
$$= 2\left[(x_i+incx)^2+(y_i+incy)^2-2R\right]-2\cdot incx\cdot x_i-1 = 2\Delta_i-(2\cdot incx\cdot x_i+1)$$
$$= 2\Delta_i - ddx$$

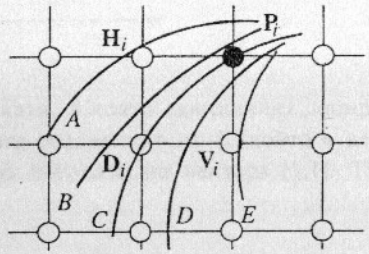
Тъй като $\Delta_i = D(\mathbf{D}_i) = D(\mathbf{P}_i)+ddx+ddy$, неравенството $\delta_i < 0$ ще се преобразува в:

$$2D(\mathbf{P}_i)+2ddx+2ddy < ddy \quad \Leftrightarrow \quad D(\mathbf{P}_i)+ddx+ddy < -D(\mathbf{P}_i)-ddx.$$

Сравненията за определяне на посоката, в която се прави всяка следваща стъпка в I и III квадранти, са еднакви. Например при $\delta_i < 0$ и в двата нечетни квадранта трябва да се избере вертикално отместеният пиксел. Не е така обаче в четните квадранти, както може да се види на фиг. 2-13.



I-ви квадрант



II-ри квадрант

$\Delta_i < 0 \quad \delta_i = D(\mathbf{D}_i)+D(\mathbf{V}_i) < 0 \rightarrow \mathbf{V}_i$
$\quad > 0 \rightarrow \mathbf{D}_i$
$\Delta_i < 0 \quad \varepsilon_i = D(\mathbf{D}_i)+D(\mathbf{H}_i) > 0 \rightarrow \mathbf{H}_i$
$\quad < 0 \rightarrow \mathbf{D}_i$

$\Delta_i < 0 \quad \delta_i = D(\mathbf{D}_i)+D(\mathbf{H}_i) < 0 \rightarrow \mathbf{H}_i$
$\quad > 0 \rightarrow \mathbf{D}_i$
$\Delta_i < 0 \quad \varepsilon_i = D(\mathbf{D}_i)+D(\mathbf{V}_i) > 0 \rightarrow \mathbf{V}_i$
$\quad < 0 \rightarrow \mathbf{D}_i$

Фиг. 2-13

Да разгледаме дефинициите на оценките и сравненията за първите два квадранта. Вижда се, че дефинициите на δ_i и ε_i трябва да са разменени за правилната работа на алгоритъма. Заедно с тях са сменени сравненията за оценката на Брезенхам. Например за избор на хоризонтално разположената точка в първи квадрант е необходимо:

$$\Delta_i > 0 \quad D(\mathbf{D}_i)+D(\mathbf{H}_i) > 0$$

докато за същия избор във втори квадрант трябва да е изпълнено:

$$\Delta_i < 0 \quad D(\mathbf{D}_i)+D(\mathbf{H}_i) < 0$$

Тази пълна симетричност ни дава основание да запазим дефинициите и сравненията, а да сменим само знака на оценката във втори квадрант. Ето защо алгоритъмът ще работи за всяка точка от окръжността, като ако тя се намира в четен квадрант, нейната оценка ще бъде с обратен знак. Остава да

видим какво трябва да се промени когато на някоя итерация се достигне до коя да е от координатните оси, т.е. когато се сменя четността на квадранта. Нека да разгледаме прехода от първи във втори квадрант. Най-напред трябва да сменим знака на самата оценка, на втората частна разлика и на нарастването при вертикално движение. Да видим как се променят първите частни разлики *ddx* и *ddy*:

$$ddx' = -2\cdot incx'\cdot x-1 = -2\cdot incx\cdot x-1 = -ddx$$
$$ddy' = -2\cdot incy'\cdot y-1 = -2\cdot(-incy)\cdot y-1 = ddy-2$$

В представената програма е използвана тази именно модификация на алгоритъма на Брезенхам, а нарастването на оценката се пресмята чрез втори частни разлики.

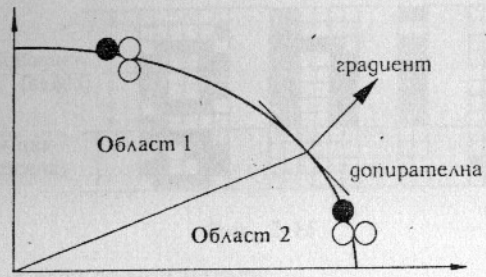
```
void DrawBresenhamArc(x,y,xc,yc,xe,ye,R,value)
int xc,yc,x,y,xe,ye,R; int value;
{int d,ddx,ddy,dd2,incx,incy;
    x-=xc; y-=yc;
    xe-=xc; ye-=yc;
    PutPointOnCentralCircle(&x,&y,R);
    PutPointOnCentralCircle(&xe,&ye,R);
     /* определяне на нарастванията в началния квадрант */
    incx=y?-sign(y):-sign(x);
    incy=x?sign(x):-sign(y);
     /* началните стойности на оценката и нейните нараствания */
    d=x*x+y*y-R*R;
    ddx=2*x*incx+1;
    ddy=2*y*incy+1;
    dd2=2;
    /* в II и IV квадранти разглеждаме оценката с обратен знак */
    if (incx==incy) {d=-d; ddy=-ddy; ddx=-ddx; dd2=-dd2;}
    while (1) {
          PutPixel(xc+x,yc+y,value);
          /* избор на следваща точка и обновяване на оценката */
          if (d+ddx+ddy>-d-ddx) {x+=incx; d+=ddx; ddx+=dd2;}
          if (d+ddx+ddy<-d-ddy) {y+=incy; d+=ddy; ddy+=dd2;}
          if (!x) {     /* достигната е оста Ox */
             d=-d; ddy-=dd2; incy=-incy; ddx=-ddx; dd2=-dd2;
          } else if (!y) {
                        /* достигната е оста Oy */
             d=-d; ddx-=dd2; incx=-incx; ddy=-ddy; dd2=-dd2;
          }
          if (x==xe && y==ye ) return;
    }
}
```

2.1.4 Растеризиране на елипса

Както споменахме по-горе, принципът на средната точка може да се приложи и за конични сечения. Ще покажем как може да се направи това за елипса. Уравнението на една централна елипса е:

$$F(x,y) = \left(\frac{x}{a}\right)^2+\left(\frac{y}{b}\right)^2 = 1, \quad \text{или} \quad F(x,y) = b^2x^2+a^2y^2-a^2b^2 = 0. \qquad [2.20]$$

За разлика от окръжността, тук можем да използваме само 4-странна симетрия. От друга страна, трябва да разделим първи квадрант на две области: в първата, изборът може да се прави между хоризонтално $\mathbf{H}_i$ и диагонално $\mathbf{D}_i$ разположените съседни пиксели, а във втората - между разположените диагонално и вертикално $\mathbf{D}_i$ и $\mathbf{V}_i$. В първата област допирателният вектор $[x,y]$ е такъв, че $x > y$, а във втората: $x < y$.



Фиг. 2-14

Границата между двете области е в точката, в която единичният допирателен вектор е [1,–1]. Това е точката, в която градиентът на F (който е вектор, перпендикулярен на допирателния) има посоката на вектора [1,1]. Тъй като

$$\operatorname{grad} F(x,y) = \left[\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}\right] = \left[2b^2x, 2a^2y\right],$$

за граничната точка (x,y) е изпълнено $b^2x = a^2y$.

Подобно на окръжността, функцията $F(x,y)$ има положителни стойности за точките извън елипсата и отрицателни за тези във вътрешността ѝ. Нека разгледаме само първата област. В нея оценката на всяка итерация е стойността на функцията в средната точка спрямо текущо избраната:

$$d_i = F\left(x_i+1, y_i-\frac{1}{2}\right) = b^2(x_i+1)^2 + a^2\left(y_i-\frac{1}{2}\right)^2 - a^2b^2.$$

При $d_i \geq 0$, трябва да изберем $\mathbf{D}_i$ и оценката на $(i+1)$-вата стъпка ще е:

$$d_{i+1} = F\left(x_i+2, y_i-\frac{3}{2}\right) = b^2(x_i+2)^2 + a^2\left(y_i-\frac{3}{2}\right)^2 - a^2b^2$$
$$= d_i + b^2(2x_i+3) + a^2(-y_i+2).$$

При $d_i < 0$ - избираме $\mathbf{H}_i$. Рекурентната зависимост за оценката в този случай е:

$$d_{i+1} = F\left(x_i+2, y_i-\frac{1}{2}\right) = b^2(x_i+2)^2 + a^2\left(y_i-\frac{1}{2}\right)^2 - a^2b^2 = d_i + b^2(2x_i+3).$$

Началната стойност на оценката в началото на първата област е:

$$d_0 = F\left(1, b-\frac{1}{2}\right) = b^2 + a^2\left(b-\frac{1}{2}\right)^2 - a^2b^2 = b^2 + a^2\left(-b+\frac{1}{4}\right).$$

Аналогични разсъждения можем да направим и за втората област. За да можем да работим с целочислена оценка, нека да умножим навсякъде по 4, т.е. да използваме оценката $d_i = 4F(\mathbf{M}_i)$. В показаната програма тази оценка се изчислява директно при започване на растеризирането във всяка от областите, а нейното нарастване се пресмята чрез първи частни разлики. Читателят може да кодира алгоритъма по-ефективно, ако използва втори частни разлики.

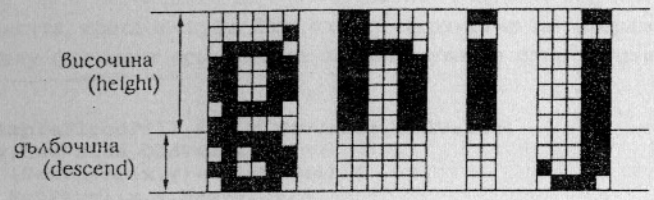
```
void DrawMidpointEllipse(xc,yc,a,b,value)
int xc,yc,a,b; int value;
{int x,y,d, asq,bsq;
    asq=a*a; bsq=b*b;
    x=0; y=b; d=4*bsq-asq*(1-4*b);
    while (asq*y>bsq*x) {                          /* Област 1 */
          FourSymmetric(xc,yc,x,y,value);
          if (d>=0) {
              d+=4*bsq*(2*x+3)+asq*(-2*y+2);
              y--;
          } else d+=4*bsq*(2*x+3);
          x++;
    }
    d=2*bsq*(2*x+1)*(2*x+1)+4*asq*(y-1)*(y-1)-4*asq*bsq;
    while (y>0) {                                  /* Област 2 */
          FourSymmetric(xc,yc,x,y,value);
          if (d>=0) {
             d+=4*bsq*(2*x+2)+asq*(-2*y+3);
             x++;
          } else d+=4*asq*(-2*y+3);
          y--;
    }
}
```

Представеният алгоритъм може да се приложи само за елипси, чиито оси съвпадат с координатните. През 1967 год. Питуей (Pitteway) предлага общ алгоритъм за растеризиране на произволно ориентирано конично сечение. В него се разглежда общото уравнение на крива от втора степен и за оценка се взема отново средната точка между двата възможни избора на всяка стъпка. Вторите частни разлики са линейни функции, които могат да се изведат по начин, подобен на този, който прилагахме дотук. Допълнителното, което се прави, е изразяването на компонентите на градиентния вектор чрез първите и втори частни разлики на оценката. Това позволява бързо определяне на границите на октантите, в които става превключване на нарастванията по всяка от координатните оси. Ние няма да се спираме тук на този алгоритъм, но препоръчваме изучаването му от тези читатели, които имат определен интерес към растерната графика.

2.1.5 Растеризиране на символи

Съществуват няколко различни начина за изобразяване на текстови символи върху растерни дисплеи. Изборът на най-подходящия начин зависи от

конкретното приложение и от изчислителната мощност на графичната система. В повечето съвременни работни станции символите се задават с контурите си, представени чрез сплайн-криви на базата на средствата, описани в шеста глава. Изобразяването чрез сплайни е доста трудоемко и затова в персоналните компютри, а и в някои работни станции се използват по-прости методи.



Фиг. 2-15

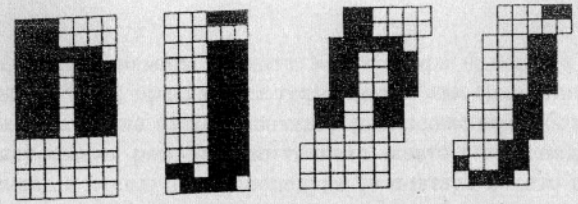
Да разгледаме един от тези методи:

Всеки шрифт се задава с набор от растерни макети за всеки един от символите, от които този шрифт се състои. Тези макети са малки правоъгълни матрици, които често имат фиксирани размери. Височината на всички символни макети е почти винаги еднаква, но ширината им може да е различна. Това е така за т.нар. *пропорционални шрифтове* - тези, за които ширината на буквите например "i" и "W" е различна. Глобални параметри са височината и дълбочината на символа (за тези символи, част от които е разположена под хоризонталния ред).

Там, където в макета стои 1, съответният пиксел трябва да бъде осветен. Вижда се, че е удобно всички символи в шрифта да са кодирани в един общ двумерен масив, като са известни ширините на всеки отделен символ. Изобразяването на един символ може да стане с показаната по-долу програма:

```
typedef struct FontDef {
                char height;
                char descent;
                int width[MAXSYMB];
                int pattern[MAXLEN][MAXHEGHT];
        } FontDef;
FontDef font;
. . .
void DrawTextSymbol(x,y,char,value)
int x,y,value; char char;
{int i,j;
    char-=FirstPrintableASCII;
    for (j=0; j<font.descend+font.height; j++)
        for (i=0; i<font.width[char]; i++)
            if (font.pattern[char+i][j])
               PutPixel(x+i,y-font.descend+j,value);
}
```

Обикновено за удебелени символи или такива в курсив се използва отделен, нов шрифт, но сравнително приемливи резултати могат да се получат чрез изобразяване с отместване (за удебеляване) и чрез трансформиране на макета (за курсив).



Фиг. 2-16

Друг, сравнително прост начин за представяне на символи е всеки от тях да се задава с последователност от вектори, като при изобразяване всеки вектор се растеризира по някой от начините, разгледани в 2.1.1. Този начин е подходящ за приложни програми, в които при мащабиране на текста не е желателно удебеляването му. Такива са например текстовите шрифтове, с които се анотират машинни чертежи. Този метод представлява един елементарен начин за геометрично моделиране, който ще разгледаме по-подробно в пета глава.

2.2 ЗАПЪЛВАНЕ НА ОБЛАСТИ И ГРАФИЧНИ ПРИМИТИВИ

В тази част ще се спрем на запълването на графични примитиви и растерни области, зададени чрез набор от пиксели. Необходимо е да обърнем внимание на разликата между тези две понятия. *Растерна област* или само *област* ще наричаме всяко множество от съседни пиксели, а от графичните примитиви ще разгледаме само затворените - многоъгълник и окръжност.

2.2.1 Запълване на области

В зависимост от това как е дефинирано понятието *съседни пиксели* има два типа области - 4-свързани и 8-свързани. 4-свързана е тази област, всяка точка на която може да бъде съединена с коя да е друга точка от областта (без да се напуска областта) с последователност от пиксели, всеки два съседни от които са разположени непосредствено един над друг или са хоризонтално един до друг (съседи или в реда или в стълба от растера, на който принадлежат). В 8-свързаните области един пиксел има за съседни пиксели още и четирите диагонално разположени пиксела.

В зависимост от това как са зададени, областите биват: *вътрешноопределени* и *граничноопределени*. Всяка област се задава спрямо някакъв фиксиран пиксел P. Вътрешноопределената област е максималното свързано (4- или 8-свързано) множество от пиксели имащи стойността на P. Може да се каже, че ако P има стойност OldValue, то пикселите по границата на областта имат стойности, различни от OldValue. Алгоритмите за запълване на такива области се споменават често под името "flood-fill" алгоритми. Областите, които са зададени чрез стойността BoundaryValue на пикселите по границата се наричат *граничноопределени*. Това е максималното свързано множество от съседни пиксели, съдържащо пиксела P, всеки от които има стойност, различна от предварително зададената BoundaryValue. За тях често се налага ограни-

чението всички пиксели от областта да имат стойност, различна от тази, с която се запълва.

ЗАПЪЛВАНЕ С ИЗПОЛЗВАНЕ НА РЕКУРСИЯ. Най-простите алгоритми за запълване от гледна точка на реализация са тези с използване на рекурсия. За тях е необходимо да е известна поне една точка, която принадлежи на областта, която и служи като входен параметър за програмата. Показаният по-долу фрагмент осъществява запълването на една вътрешноопределена област.

```
void SimpleFloodFill_4(x,y,NewValue,OldValue)
int x,y,NewValue,OldValue;
{   if (GetPixel(x,y)==OldValue) {
       PutPixel(x,y,NewValue);
       SimpleFloodFill_4(x-1,y,NewValue,OldValue);
       SimpleFloodFill_4(x+1,y,NewValue,OldValue);
       SimpleFloodFill_4(x,y-1,NewValue,OldValue);
       SimpleFloodFill_4(x,y+1,NewValue,OldValue);
    }
}
```

При 8-свързана област е необходимо рекурсивно обръщение и за диагонално разположените пиксели. Обърнете внимание, че този метод не би работил, ако старата и новата стойност на запълване съвпадат. Аналогично на програмата за запълване на вътрешноопределена област, тази за запълване на граничноопределена област има рекурсивно обръщение за всички пиксели, които още не са запълнени и които не принадлежат на границата ѝ.

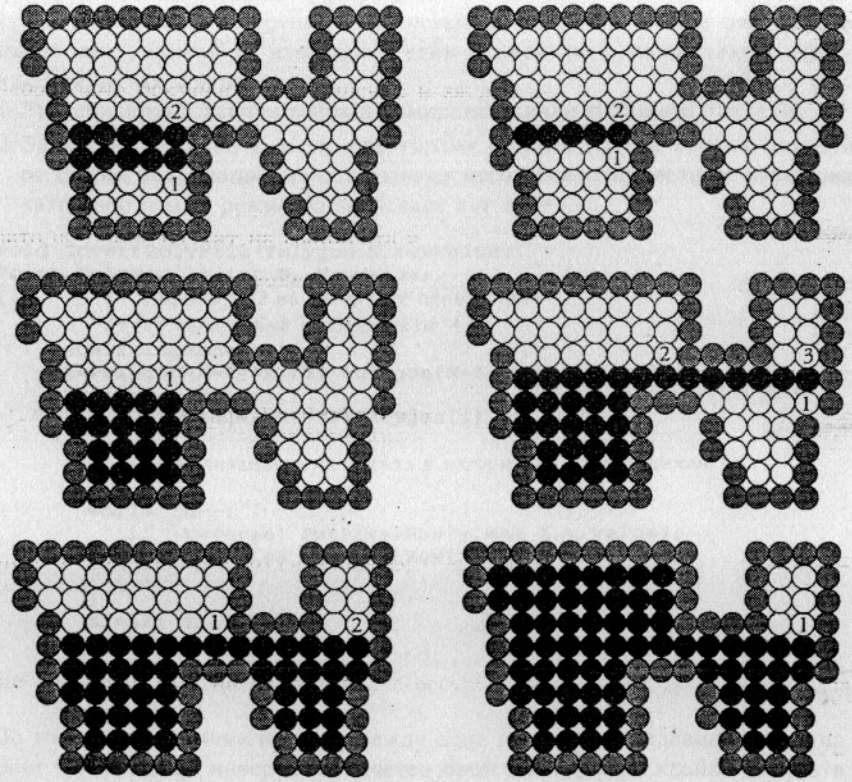
```
void SimpleBoundaryFill_4(x,y,NewValue,BorderValue)
int x,y,NewValue,BorderValue;
{int value;
    value=GetPixel(x,y);
    if (value!=NewValue && value!=BorderValue) {
       PutPixel(x,y,NewValue);
       SimpleBoundaryFill_4(x-1,y,NewValue,BorderValue);
       SimpleBoundaryFill_4(x+1,y,NewValue,BorderValue);
       SimpleBoundaryFill_4(x,y-1,NewValue,BorderValue);
       SimpleBoundaryFill_4(x,y+1,NewValue,BorderValue);
    }
}
```

Вижда се, че запълването може да се обобщи като се използва множество от няколко вътрешни и няколко гранични цвята.

ЗАПЪЛВАНЕ С ОПТИМАЛНА ДЪЛБОЧИНА НА СТЕКА. Горните алгоритми далеч не са оптимални по отношение на използваните ресурси на компютърната система. Дълбочината на стека на рекурсията може да се намали съществено с предложения по-долу алгоритъм за граничноопределена област. Принципът, на който той работи е следният:

Първо се запълва целият хоризонтален ред, съдържащ началната точка и се определят неговите две граници `Xleft` и `Xright`. После се разглежда редът, който се намира непосредствено под вече запълнения ред. Започвайки от точката, разположена точно под тази с x-координата `Xleft` и движейки се

се отляво надясно се намира точката, която опира надясно в гранична точка. Тя се записва в стека, прескача се граничната (или граничните) точки, докато се намери нова група от хоризонтално разположени необработени точки от областта върху същия ред. Отново точката, която опира надясно в гранична точка се записва в стека. Тази процедура се повтаря докато разгледаме всички точки наляво от `Xright`. Същото се прави и за реда, намиращ се непосредствено над разглеждания ред. На фиг. 2-17 е показана последователността в запълването на една област и редът, в които точките се поставят в стека. Ще отбележим, че най-дясно разположената точка, която се записва в стека, не е непременно опряна отдясно в границата. Това е възможно в случай, че пикселът, намиращ се точно под (или над) дясната граница `Xright` на текущия ред е незапълнен пиксел от областта, а отдясно на него не стои граничен пиксел. Затова именно запълването на всяка група съседни пиксели от един ред става в двете посоки, започвайки от текущата точка.



Фиг. 2-17

Ще обърнем внимание и на това, че след като се извлече точка от стека, трябва да се провери дали тя не е вече запълнена от предишна итерация. Това може да се случи когато запълваме области, на които контурите са вло-

жени един в друг. По този начин ще се избегне неколкократно запълване на едни и същи пиксели. Забележете, че така предложеният алгоритъм е приложим само за 4-свързана област, тъй като съседните редове се разглеждат само в интервала под най-левия и най-десния запълнени пиксели на текущия ред.

В програмната реализация най-десният от всяка група незапълнени пиксели се намира, като се търси двойката *незапълнен-граничен* пиксели. Ако такава двойка се намери, незапълненият пиксел се вкарва в стека. Отделено е разглеждането на пикселите под и над дясната граница. Ако някой от тях е незапълнен, той със сигурност принадлежи на група пиксели, на която още не е намерен най-десният. Следователно, този пиксел трябва също да се сложи в стека.

```
void StackedBoundaryFill_4(x,y,NewValue,BorderValue)
int x,y;
{int nexty;
    pushPoint(x,y);
    while (!isPointStackEmpty()) {
       popPoint(&x,&y);
       if (GetPixel(x,y)==NewValue)
           continue; /* този пиксел вече е запълнен */
       Xleft=Xright=x;
       /* намираме лявата граница на реда */
       while (GetPixel(Xleft-1,y)!=BorderValue ) Xleft--;
       /* намираме дясната граница на реда */
       while (GetPixel(Xright+1,y)!=BorderValue) Xright++;
       PutPixelRow(Xleft,Xright,y,NewValue);
       /* разглеждаме редовете непосредствено под и над текущия ред */
       for (nexty=y-1; nexty<y+2; nexty+=2) {
          p1=GetPixel(Xleft,nexty);
          for (x=Xleft;x<Xright;x++) {
             p2=GetPixel(x+1,nexty);
             /* дали това е двойка "незапълнен-граничен" пиксели */
             if (p1!=BorderValue && p1!=NewValue &&
                 p2==BorderValue) pushPixel(x,nexty);
             p1=p2;
          }
       }
    }
}
```

2.2.2 Запълване на многоъгълник

Запълването на пикселите, заградени от многоъгълник, зададен с последователност от върхове е задача, която се налага да се решава извънредно често в компютърната графика. Например визуализирането на едноцветна стена на обемно тяло се свежда до запълването на проекцията на тази стена, която представлява многоъгълник.

При разработването на ефективни алгоритми за запълване е важно да се подхожда по различен начин в зависимост от това какъв е многоъгълникът. Ако този многоъгълник е правоъгълник със страни, успоредни на координатните оси, неговото запълване би било елементарно и би било неразумно да се използват общи алгоритми. Това е важен принцип в базовите алгоритми за визуализация - да се използват по-прости методи за по-простите случаи. Например в системата PHIGS се прави разлика между обработката на изпъкнали и неизпъкнали многоъгълници. Приложният програмист указва типа на многоъгълника при създаването му, за да може графичната система правилно да подбира най-ефективния алгоритъм за работа с него - в частност за запълването му.

ЗАПЪЛВАНЕ ЧРЕЗ ИНВЕРТИРАНЕ НА ПИКСЕЛИ. Ако разгледаме един ред от растера, можем да кажем, че в общия случай пресечните точки на този ред с ребрата на многоъгълника са четен брой. Всяка двойка такива точки (нечетна и следващата я четна) загражда група от пиксели, които са вътрешни за многоъгълника. Ще наречем пикселите в тези пресечни точки *гранични пиксели*. Задачата ни тогава се свежда до намирането на последователността от двойките гранични пиксели за всеки ред от растера.

Граничните пиксели можем да получим като растеризираме ребрата по някой от стъпковите алгоритми, разгледани в 2.1.1. Ако пикселите от вътрешността на многоъгълника трябва да се установят в новия цвят в режим "изключващо или", можем да използваме един елементарен, но не твърде ефективен начин за запълване:

1. Растеризираме всички ребра и намираме всички гранични пиксели;
2. За всеки граничен пиксел инвертираме стойностите на всички пиксели от същия ред, разположени вдясно от него. Инвертирането се извършва като използваме режим на записване xor с S= 1.

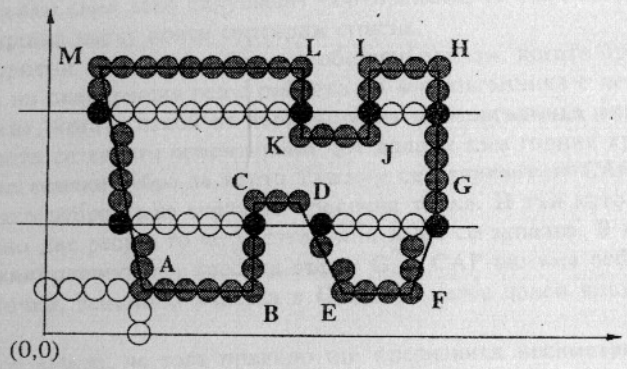
```
void InvertPolyFill(Polygon,N,NewValue)
Point Polygon[]; int N, NewValue;
{. . .  /* дефиниране на локалните променливи */
        /* от алгоритъма на Брезенхам */
    SetWritingMode(XOR);
    X1=Polygon[N-1].x; Y1=Polygon[N-1].y;
    for (i=0;i<N;i++) {
       X2=Polygon[i].x; Y2=Polygon[i].y;
       . . .
       /* инициализацията на цикъла в алгоритъма на Брезенхам */
       . . .
       while (n--) {
          if (reverse) PutPixelRow(y,MAX_X,x,value);
                       else PutPixelRow(x,MAX_X,y,value);
       . . .
       }
       X1=X2; Y1=Y2;
    }
    SetWritingMode(REPLACE);
}
```

По този начин всички пиксели между една нечетна и следващата я четна гранични точки ще се инвертират нечетен брой пъти (т.е. в крайна сметка в тях ще се установи новата стойност), а всички пиксели между една четна и следващата я нечетна ще се инвертират четен брой пъти. Тъй като прилагането на "изключващо или" (xor) с една и съща стойност четен брой пъти

води до възстановяване на началното състояние, то пикселите, които не принадлежат на вътрешността на многоъгълника, ще останат непроменени. Остават особените случаи по границата на многоъгълника, но за тях можем да въведем отделно правило, както ще видим по-долу.

Този алгоритъм може да се кодира като инвертирането на реда се прави при последователното получаване на всеки граничен пиксел, растеризирайки контура. Показаната програма е проста модификация на алгоритъма на Брезенхам.

АЛГОРИТЪМ НА СКАНИРАЩИЯ РЕД. По-ефективно е, разбира се, запълването между двойките гранични пиксели да се извършва като итерациите се правят съобразно редовете от растера, а не за всяко едно от ребрата поотделно. В тази част ще разгледаме един алгоритъм за запълване, който се основава на тази именно идея и който може да се приложи не само за многоъгълник, но и за произволна едносвързана област, зададена с вложени контури, както и за самопресичащ се контур. Тъй като запълването се извършва по редове, той носи името *алгоритъм на сканиращия ред*.



Фиг. 2-18

Нека преди да започнем запълването да направим следното:

1. Отстраняваме всички хоризонтални ребра;
2. Подготвяме следната информация за всяко ребро:

`Ymin`: най-малката y-координата на реброто (това е y-координатата на по-долния му край);

`Ymax`: най-голямата y-координата на реброто (това е y-координатата на по-горния край);

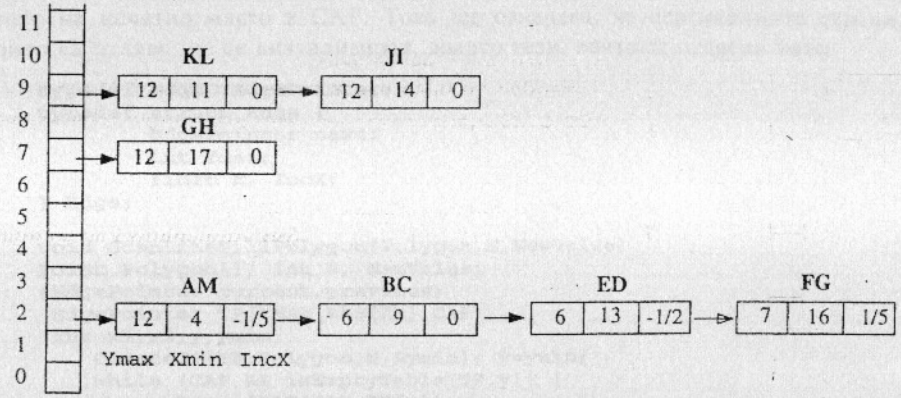
`Xmin`: x-координата на по-ниския край на реброто (това е x-координатата на този край, чиято y-координата е `Ymin`);

`IncX`: изменение по x при нарастването на y с единица.

Последното можем да получим директно от уравнението на правата, носеща реброто, тъй като то не е хоризонтално. Ще използваме уравнение, подобно на [2.1], но спрямо x:

$$x = IncX(y - Y_1) + X_1, \qquad IncX = \frac{dx}{dy} = \frac{X_2 - X_1}{Y_2 - Y_1}$$

3. Тази информация за ребрата подреждаме в таблица, в която за всяко y се съдържа сортиран списък спрямо `Xmin` от всички ребра, чиито `Ymin`=y. Ще наречем тази таблица *таблица на ребрата* - ТР. Отчитайки, че точката A има координати (4,2), многоъгълникът от фиг. 2-18 ще има следната ТР:



Фиг. 2-19

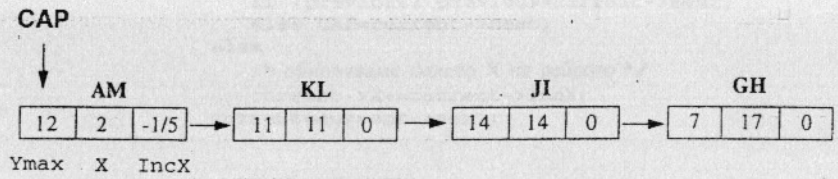
Алгоритъмът на сканиращия ред използва тази таблица, за да поддържа на всяка стъпка (за всеки текущо обработван ред от растера) един списък от всички ребра, които имат пресечни точки с този ред. Този списък ще наричаме *списък на активните ребра* - САР. Всеки елемент от него характеризира по едно ребро и съдържа данни, подобни на тези в ТР:

`Ymax`: най-голямата y-координата на реброто (също като в ТР);

`X` : x-координатата на пресечната точка на реброто с реда;

`IncX`: също като в ТР.

За сканиращия ред `y`=10 САР за същия многоъгълник (от фиг. 2-18) ще има вида, показан на фиг. 2-20.



Фиг. 2-20

Можем да получим списъка на следващия ред от растера - `y`=11 - директно от показания САР като за всяко ребро увеличим `X` с `IncX`. Тъй като е възможно от следващия ред да започва ново ребро, то е необходимо да добавим и съответния подсписък от ТР за новото `y`. Разбира се, ще е необходимо

след това да сортираме списъка, особено ако желаем алгоритъмът да работи за области с неизпъкнали контури. Някои от ребрата могат да не достигат до следващия ред, което лесно можем да установим от техните Ymax стойности. Те пък ще трябва да се изтрият от CAP.

Ето и самия алгоритъм:

1. Построяваме TP за многоъгълника;
2. Инициализираме CAP като празен списък;
3. Даваме на y стойността на най-малкото y в TP;
4. Повтаряме до момента, в който и TP, и CAP станат празни:
   4.1. Добавяме към CAP списъка от TP за текущото y. Това е сливане на два сортирани списъка и може да се осъществи доста ефективно;
   4.2. Запълваме пикселите между всяка двойка от последователни нечетна-четна пресечни точки от CAP;
   4.3. Увеличаваме y с 1;
   4.4. Изтриваме от CAP всички ребра за които Ymax=y;
   4.5. За всяко ребро в CAP увеличаваме X с IncX;
   4.6. Сортираме CAP по нарастване на x. Тъй като предишните две стъпки само леко нарушават сортирането, то тази операция се извършва върху почти сортиран списък.

Този алгоритъм се справя добре с особените случаи, когато броят на пресечните точки на сканиращия ред с ребрата на многоъгълника е нечетен. Това се случва тогава, когато някой от върховете на многоъгълника лежи на реда. Тъй като ребрата са винаги ориентирани (от долния към горния край), а според алгоритъма всички ребра за които Ymax=y се изтриват от CAP, то горният връх на всяко ребро не се счита за пресечна точка. И тъй като всеки връх е край на точно две ребра, то четността винаги ще се запазва. В горния пример, когато сканиращият ред пресича върха **G**, в CAP ще има отбелязана една пресечна точка, защото G участва в **GH** като негов долен връх, но не и в **FG**.

Веднага се вижда, че това правило ще предизвика несиметрично третиране на върхове, които са едновременно долни и едновременно горни краища на своите ребра. Връх, който е долен край и на двете ребра, които се събират в него ще се запълни, докато такъв, който е горен край на две съседни ребра няма да бъде запълнен. Аналогично, хоризонталните ребра ще се растеризират в зависимост от това дали вътрешността на многоъгълника се намира от долната или от горната им страна. Например ребрата **AB**, **EF** и **KJ** от същия пример ще се растеризират, докато **CD**, **ML** и **IH** няма.

Такова разграничение е необходимо да се прави особено когато се визуализира група от многоъгълници, които имат общи страни. При използване на представения алгоритъм общите им хоризонтални ребра ще се растеризират само по веднъж. Това обаче не е вярно за останалите, нехоризонтални ребра. Причината е, че за растеризирането им се използва метод, пригоден за отсечки, в който не се отчита от коя страна е разположена вътрешността на многоъгълника.

За решаването на този проблем първо трябва да се дадат отговори на следните въпроси:

- Кой пиксел да изберем за граничен, когато пресечната точка на едно ребро със сканиращия ред е пиксел от растера? Винаги ли самият пиксел-пресечна точка е най-правилният избор?
- Кой пиксел да изберем за граничен, когато пресечната точка на едно ребро със сканиращия ред НЕ е пиксел от растера, т.е. когато е необходимо закръгляване?

За да отговорим на първия въпрос можем да приемем правилото да включваме всички пиксели, които са начало на интервал за запълване, т.е. стоят на нечетно място в CAP. Това ще означава, че вертикалните страни, които са отляво ще се визуализират, докато тези, които са отдясно няма.

```
typedef Edge *EdgePointer;
typedef struct Edge {
        EdgePointer next;
        int Ymax;
        float X, IncX;
} Edge;

void ScanLineFillPolygon(Polygon,N,NewValue)
Point Polygon[]; int N, NewValue;
{EdgePointer current,previous;
 EdgePointer TP[YMAX_RASTER],CAP;
 int x1,x2,y,ymin;
    CreateTP(TP,Polygon,N,&ymin); y=ymin;
    while (CAP && isEmptyTable(TP,y)) {
          AppendCAP(CAP,TP[y]);
          SortCAPbyX(CAP);
          current=CAP;
          while (current) {
            /* закръгляваме към вътрешността */
                x1=ceil(current->X); current=current->next;
                x2=floor(current->X);
            /* изключваме дясната граница ако е част от растеризацията*/
                if (current->X==x2) x2--;
                if (x1<=x2) PutPixelRow(x1,x2,y,NewValue);
                current=current->next;
          }
          y++; current=CAP; previous=NULL;
          while (current) {
                if (current->Ymax==y)
                   /* изключваме реброто от CAP */
                   if (previous) previous=current->next;
                   else CAP=current->next;
                else
                   /* обновяваме полето X на реброто */
                   current->X+=current->IncX;
                current=current->next;
          }
    }
}
```

Закръгляването ще извършваме в зависимост от това дали пресечната точка е начало на интервал за запълване или не. Ако тя е отляво на такъв

интервал, т.е. стои на нечетно място в CAP - закръгляваме нагоре (към вътрешността), ако пък тази пресечна точка стои на четно място - закръгляваме надолу (отново към вътрешността). Показаната програма илюстрира приетите правила.

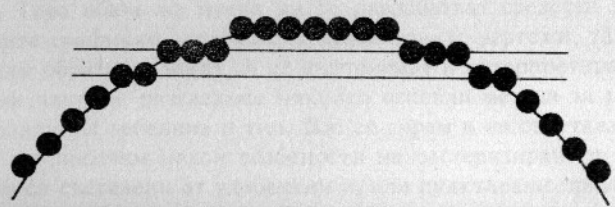
Тъй като IncX е рационално число, алгоритъмът може да се преобразува в целочислен, като се съхраняват отделно числителят и знаменателят на това число. Закръгляването надолу тогава извършваме, като за всяка пресечна точка X пазим и числителя на дробната ѝ част. На всяка стъпка ще прибавяме към него числителя на нарастването и ако той е по-голям от знаменателя, това означава, че цялата част се увеличава с единица, а дробната се намалява със знаменателя на нарастването.

В началото на този параграф отбелязахме, че е необходимо да се опростяват алгоритмите винаги, когато това е възможно. Запълването на изпъкнал многоъгълник е удачно да се прави без да се използват толкова сложни структури от данни, тъй като всеки сканиращ ред пресича изпъкналия многоъгълник точно два пъти. Това важи и за окръжности и елипси, както ще видим по-долу.

Може да се каже, че тъй като всеки многоъгълник може да се представи като множество от изпъкнали, запълването му може да се сведе до запълването на тези многоъгълници. Същото се постига и чрез триангулация на многоъгълника - разбиването му на триъгълници. Това обаче са известни задачи от изчислителната геометрия, чието решаване не е тривиално.

2.2.3 Запълване на окръжност и елипса

Запълването на окръжност може да се извърши, като отново се използва идеята на *сканиращия ред*. Както казахме, в този случай сканиращият ред пресича окръжността точно два пъти и при това двете пресечни точки са симетрично разположени спрямо центъра ѝ.



Фиг. 2-21

Можем да използване алгоритъма на Брезенхам за растеризиране на окръжността, при който на всяка стъпка се осветява пиксел или от същия ред или от този под него. Трябва да вземем предвид, че при генериране на най-горната (и респективно най-долната) ѝ част - когато $|y| > x$ - на един ред могат да се получат по няколко пиксела от всяка страна както е показано на фиг. 2-21.

От тях за десен граничен пиксел трябва да изберем този пиксел от дясната група, който е разположен най-отдясно и се намира във вътрешността на

окръжността. Левият граничен пиксел можем да получим чрез симетрия. За да определим дали пикселът е вътрешен за окръжността, можем да проверим знака на $F(x,y)$ в тази точка. В алгоритъма на Брезенхам стойността на тази функция можем да получим директно от оценката, дефинирана с [2.11]. От тази дефиниция директно следва:

$$F(x_i, y_i) = d_i - 2x_i + 2y_i - 2.$$

Може да се случи така, че от цялата група десни гранични пиксели нито един да не е вътрешен. Тогава за граничен пиксел в сканиращия ред трябва да изберем този от двата съседни пиксела, който има най-голяма отрицателна стойност F.

В случаите, когато граничният пиксел лежи точно върху окръжността, ще приложим правилото, прието в предния параграф, а именно да включваме левите и да изключваме десните граници. Това може да е неприемливо за някои приложения, тъй като не се запазва симетричността, която е важна характеристика на окръжността.

2.2.4 Запълване с образец

Образецът е най-често малка правоъгълна $M \times N$-матрица от нули и единици, подобна на тази, използвана за визуализиране на текстови символи. Образецът може да съдържа не само 0 и 1, но и стойности, с които се дефинира цвят, макар че това се прави много рядко. Запълването на една област с образец се извършва като всеки неин пиксел се осветява съобразно стойността в съответен нему елемент от образеца. Това съответствие е различно в различните растерни системи. Възможни са следните варианти:

- *Образецът е свързан с екрана* (или с текущия прозорец върху екрана). Това означава, че елементът с координати (0,0) от образеца съответства на началото на координатната система. Всеки вътрешен пиксел от областта с координати (x,y) ще има за съответен елемента със същите координати, но взети всяка по модул размера на образеца по тази ос, а именно: (x mod M, y mod N);
- *Образецът е свързан с областта*, т.е. началото му съвпада с някоя характерна нейна точка. Типичен проблем в този случай е изборът на такава опорна точка, особено когато става дума за произволен многоъгълник. Най-лесно е да се вземе първият връх от списъка върхове или най-левият и най-долен такъв. Съответствието между пикселите тогава ще е:

$$(x,y) \rightarrow ((x-x_0) \bmod M,\ (y-y_0) \bmod N)$$

Запълването е различно и в зависимост от това дали всички пиксели от областта трябва да получат нова стойност. То може да бъде:

- Прозрачно запълване: Запълват се само тези пиксели, чиито съответни елементи от образеца са ненулеви:

```
if (pattern[x%M][y%N]) PutPixel(x,y,value);
```

- Плътно запълване: Запълват се всички пиксели - тези, за които в образеца стои 0, се запълват със стойността на фона (в нашия пример - стойността на променливата background), а останалите - с основния цвят (foreground):

```
PutPixel(x,y,pattern[x%M][y%N]?foreground:background);
```

При плътно запълване е възможно да се записват пикселите не един по един, а на групи от по цял хоризонтален ред от образеца.

Един друг начин за генериране на примитив, запълнен с образец е първо да се растеризира той в правоъгълна работна област с размерите на правоъгълната му обвивка (минималния изправен правоъгълник, които изцяло обхваща примитива). След запълването в тази работна област тя се визуализира върху екрана. Това е два пъти по-сложно от разгледания преди малко начин, но е полезно ако:

- съответният примитив ще се генерира много често, какъвто е случаят с текстови символи и икони; и/или
- графичната система има ефективна реализация на функцията за запис на блок от пиксели върху екрана; и/или
- запълване с два цвята, всеки от които е различен от фоновия.

2.3 ВИЗУАЛНИ АТРИБУТИ НА ГРАФИЧНИТЕ ПРИМИТИВИ

Дотук разгледахме растеризирането на отсечка и окръжност с плътна линия, която има дебелина един пиксел. В чертожните системи също толкова често се използват пунктирани, тънки и дебели линии за улесняване на интерпретацията на чертежа.

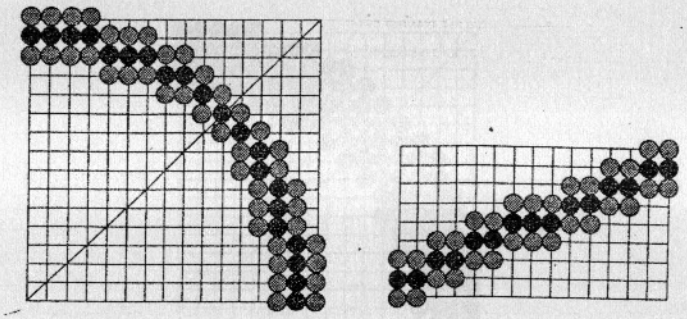
Възможностите на растерните дисплеи за визуализация на чертежи съвсем не отговарят на възможностите, които един чертожник има върху простия бял лист. Това обаче не пречи да се разработват средства за имитиране на най-важните графични атрибути от инженерните чертежи, така че създаваните визуални образи да могат да се възприемат и интерпретират като чертежи.

В тази част ще разгледаме няколко основни метода за растеризиране на линии с различна дебелина и тип. Ще се спрем и на съчетаването на тези атрибути и ще посочим някои особености на растеризирането на начупени линии, които са съставени от удебелени и/или пунктирани примитиви.

2.3.1 Удебеляване на примитиви

Има няколко основни метода за удебеляване на примитиви. Най-важните от тях са: повторение на пиксели; имитиране на следата, оставяна от движещо се перо с определено сечение; запълване на областта между два образа на примитива, отместени един от друг на разстояние, съответстващо на неговата дебелина. Всеки от тези методи има своите предимства и недостатъци, но основната разлика е в това как се прави балансът между *добър визуален резултат* и *приемлива изчислителна сложност*.

УДЕБЕЛЯВАНЕ ЧРЕЗ ПОВТОРЕНИЕ НА ПИКСЕЛИ. Това е най-простият метод, който има и минимална изчислителна сложност, защото е елементарно разширение на алгоритъма за растеризиране. Нека да разгледаме отсечка, чийто наклон е не по-голям от 45 градуса. Във всеки стълб на растера има най-много по един пиксел от тази отсечка. Естествено удебеляване би се получило, ако при растеризирането ѝ вместо по един пиксел осветяваме симетрично и неговите съседни във всеки стълб. Аналогично, когато отсечката има наклон по-голям от 45 градуса ще повтаряме пикселите по редове. При окръжността повторението ще се извършва в реда или в стълба в зависимост от това в кой квадрант се намира пикселът.



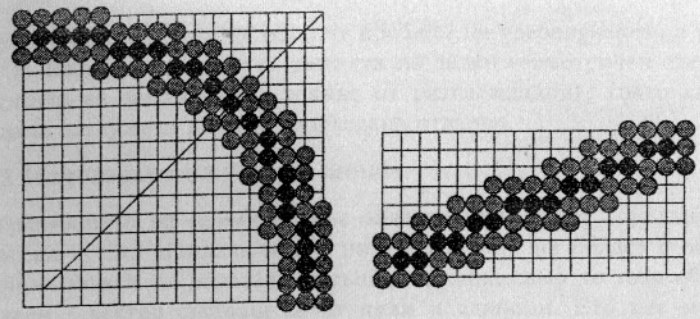
Фиг. 2-22

Типични проблеми при този подход са следните:

- Краищата на примитивите са хоризонтално или вертикално отрязани.
- Има забележимо изтъняване при увеличаване на наклона (от 0 към 45 градуса), което е особено ярко изразено в точките на превключване на повторението от ред към стълб в окръжността (в границите на октантите).
- Върховете на един удебелен по този начин многоъгълник няма да изглеждат добре. Например ъглите на един правоъгълник, страните на който са удебелени чрез повторение на пиксели ще изглеждат поръбени, защото хоризонталните страни са удебелени вертикално, а вертикалните - хоризонтално.
- Удебеляването няма да е симетрично, ако зададената дебелина определя четен брой пиксели във всеки стълб (или ред).

ДВИЖЕЩО СЕ ПЕРО. Имитирането на следата, която оставя перо със зададен профил при движението си по пикселите на растеризацията е много често използван подход. На всяка стъпка профилът се разполага така, че центърът му да е в пиксела, получен при растеризирането. Тук естествено възникват няколко въпроса:

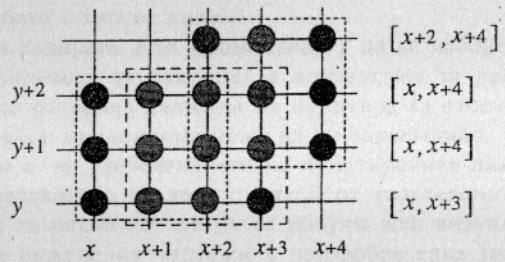
- Какво е най-подходящото сечение на перото?
- Трябва ли сечението да се ориентира спрямо примитива, който се удебелява?



Фиг. 2-23

Обикновено се избира правоъгълно сечение, което има винаги една и съща ориентация, т.е. не е свързано с примитива. За разлика от предния подход, това би довело до по-добре оформени краища, макар и значително удебелени. Самото удебеляване отново не е равномерно за всички наклони, като този път най-тънки са хоризонталните и вертикални участъци. Последното може да се избегне, като се ориентира сечението или се избере сечение с формата на кръг, което е симетрично. Допълнителен проблем при използването на перо с кръгло сечение е, че алгоритъмът трябва да е пригоден за запълване на кръг с радиус, който не е цяло число, а именно ($R=W/2$).

Този метод може лесно да се съчетае с алгоритъма на Брезенхам, като на всяка стъпка от растеризирането се запълва сечение с център генерирания пиксел. Тогава възниква проблемът, че ще има пиксели, които ще се запълват многократно. Както казахме и преди, това трябва да се избягва, особено в режимите **and**, **or** и **xor**. Едно разрешение е да се използва принципът на сканиращия ред, т.е. записването на пиксели да се извършва ред по ред, след като за всеки ред се намерят лявата и дясна граници на групата съседни пиксели.

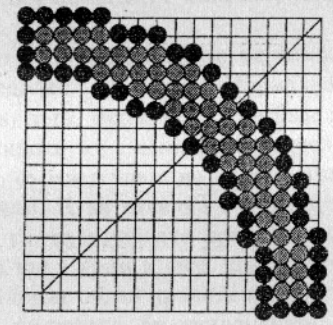


Фиг. 2-24

При удебеляване на отсечка на всяка стъпка ще получаваме интервали за няколко съседни реда, които трябва да обединяваме с интервалите, получени при налагане на сечението върху предходните пиксели от растеризацията. При окръжност, както и при многоъгълник, за всеки сканиращ ред ще има набор от интервали, което води до използването на същите структури като в алгоритъма на сканиращия ред.

УДЕБЕЛЯВАНЕ ЧРЕЗ ЗАПЪЛВАНЕ. Този метод дава най-добри визуални резултати и може да се реализира чрез създаване на област, която впоследствие се запълва по някой от разгледаните в предната част методи. Контурът на областта се получава при отместване на примитива по посоката на нормалата му на $W/2$ и $-W/2$. Ако примитивът е граница на друга област, то граници на удебеляването могат да бъдат самият той и образът, отместен на W към вътрешността на тази област. Така би се решил и проблемът за четното удебеляване.

Генерирането на отместени образи на елипси води до решаване на уравнения от 8-ма степен, затова разширяването и свиването им се апроксимира чрез модифициране на всяка от полуосите с $W/2$.



Фиг. 2-25

КРАЙНИ ТОЧКИ И ВЪРХОВЕ НА МНОГОЪГЪЛНИЦИ. Важен проблем на удебеляването е оформянето на крайните точки на примитивите и върховете на многоъгълниците. На фигурата по-долу са показани някои от възможностите да се направи това за една отсечка. Изборът на типа на удебеляването на крайната точка зависи както от нуждите на приложната програма, така и от това дали примитивът е самостоятелен или е свързан с други. В повечето съвременни графични системи се предлагат по няколко възможности, от които потребителят или приложният програмист избира тази, която е най-подходяща за конкретния случай.

Фиг. 2-26

Оформянето на върховете на един многоъгълник може естествено да се контролира чрез избора на типа на краищата или чрез отсичане на удебеляването по ъглополовящата на върха. Последното се нуждае от допълнителен контрол - например при скосяване на много остър връх.

Накрая ще отбележим, че общият проблем на удебеляването е подобен на задачата за построяване на еквидистанта на даден геометричен елемент (елемент, отстоящ на зададено разстояние от разглеждания), която като аналитична задача заслужава да бъде разгледана отделно.

2.3.2 Използване на типове линии

Изобразяването на примитив със зададен тип линия (пунктирана, централна, осева и т.н.) прилича много на запълването на област с образец. По тази причина можем да използваме същия термин, само че този образец сега ще е линеен - вектор състоящ се от нули и единици. Ето как могат да се представят няколко различни типа линии: пунктирана (от дълги тирета), точкова, осева (тире и точка) и пунктирана (от къси тирета). Дължината на вектора-образец в показания пример е 12 пиксела:

```
Pattern dashed    = {1,1,1,1,1,1,0,0,0,0,0,0},
        dotted    = {1,0,1,0,1,0,1,0,1,0,1,0},
        dashdot   = {1,1,1,1,1,0,0,0,1,0,0,0},
        shortdash = {1,1,1,0,0,0,1,1,1,0,0,0};
```

Аналогично на запълването на област с образец, типовете линии могат да се използват в два режима: прозрачно запълване и плътно запълване, които се реализират по следния начин:

```
if (pattern[i%N]) PutPixel(x,y,value)
```

или

```
PutPixel(x,y,pattern[i%N]?foreground:background);
```

В практиката най-често се използва прозрачно запълване, което означава, че пикселите, намиращи се между тиретата на една пунктирана отсечка няма да бъдат променяни. Много растерни системи предлагат само едната възможност (като пример ще посочим, че Windows-NT предлага само плътно запълване при работа с типове линии).

Най-лесно за кодиране е за променлива i да се избере нарастването (по x или по y в зависимост от наклона) в алгоритъма за растеризиране. Това обаче би довело до различна дължина на пунктира за отсечки с различни наклони, което за някои приложения може да е неприемливо.

Друг проблем е, че удебеляването на пунктираните линии не може да се извърши при съчетаването на този с някой от разгледаните по-горе методи без да се рискува за някои случаи да се получи лош визуален резултат. При визуализиране на инженерни чертежи е най-добре една пунктирана отсечка да се разбие на множеството от малки отсечки, съставящи пунктира, и всяка от тях да се удебелява поотделно.

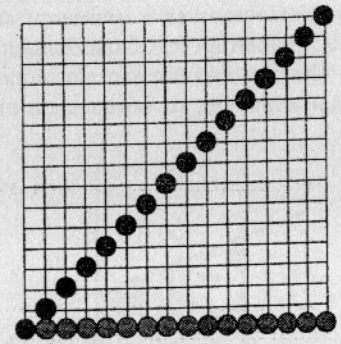
За разлика от образеца при запълване на области, този се привързва към обекта - почти винаги към неговата начална точка, а не към екрана. Единственото изключение са начупените линии, при които образецът трябва се привързва към началото на начупената, а не към началото на всяка нейна отсечка.

2.4 ИЗГЛАЖДАНЕ НА РАСТЕРИЗАЦИЯТА

Всички разгледани дотук алгоритми генерират примитиви, които имат малко или повече стъпаловидна форма. Особено отчетливо това се вижда върху дисплеи с ниска разрешаваща способност (със сравнително малък брой пиксели върху единица площ). Причината за това е, че алгоритмите са основани на принципа "всичко или нищо", т.е. пикселите са или осветени или не. Това, разбира се, е и единствената възможност за устройства, в чиято пиксел-на карта има отделен само по един бит за пиксел.

Тъй като използването на дисплеи с повече от една *равнини* е вече масово, тук ще разгледаме няколко метода за получаване на *изгладени* примитиви чрез осветяване на поредицата от пиксели, така че всеки от тях да има различен интензитет.

Първо е необходимо да отбележим, че не само пикселите, но и самите примитиви светят с определен интензитет. Нормално е интензитетът на един линеен елемент - отсечка, дъга, окръжност - да бъде пропорционален на неговата дължина (ако дебелината му е само един пиксел). Запълнените елементи пък ще имат интензитет, съответстващ на площта им. Това далеч не е така в разгледаните досега случаи. Една отсечка с наклон 45 градуса например ще има същия брой пиксели както и нейната проекция върху оста x, т.е. интензитетът и на двете отсечки ще е един и същ. Дължините им обаче съвсем не са равни и за да бъде визуалният образ правилен и по-точно - за да не изглежда хоризонталната отсечка по-дебела от наклонената, е необходимо всеки от пикселите на последната да свети с интензитет $\sqrt{2}$.

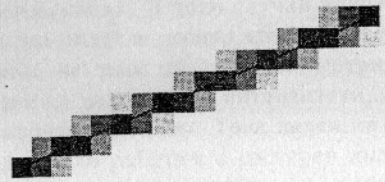


Фиг. 2-27

Това определено няма да е достатъчно за да изглежда тя по-плътна. За да се отстрани стъпаловидността ѝ ще е необходимо във всеки вертикален стълб от растера да се осветят по няколко пиксела. Интензитетът на всеки един от тях трябва да е пропорционален на частта от отсечката, която припокрива този пиксел. За да прецизираме последното е необходимо първо да изясним няколко неща:

- Каква е формата и големината на един пиксел?
- Каква е формата на един примитив с дебелина един пиксел?

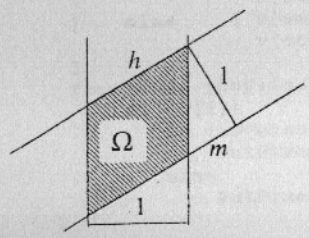
Възможни са няколко различни отговора на тези въпроси. На фигурите досега изобразявахме пикселите като кръгове, за да може да ги отличаваме от координатната мрежа. По-правилно е да третираме всеки пиксел като квадрат със страна 1, с чиито координати (x,y) ще отбелязваме центъра му. Това означава, че хоризонталната и вертикалната отсечки са за нас правоъгълници с ширина един пиксел. По този начин естествено стигаме до извода да разглеждаме всяка растерна отсечка като такъв правоъгълник. Аналогично, растерната окръжност ще е пръстенът, заграден между две концентрични окръжности, разликата между радиусите на които е 1.



Фиг. 2-28

2.4.1 Изглаждане чрез оценка на припокритата площ

Така дефинираните растерни примитиви вече имат площи и следователно интензитетът на един пиксел може да се определи като площта на тази част, която примитивът припокрива от него. Нека вземем една отсечка с наклон между 0 и 45 градуса. Да оставим настрана въпроса за растеризирането на крайните ѝ точки и да разгледаме кой да е друг неин вътрешен пиксел. От всеки вертикален стълб на растера отсечката отсича определена площ, която трябва да се разпредели между два или три пиксела от този стълб. Тази площ можем да изразим чрез наклона на отсечката:



$$\Omega = 1 \cdot h = \sqrt{m^2+1} = \frac{\sqrt{dx^2+dy^2}}{dx}$$

Фиг. 2-29

Ще се опитаме да изразим площта, която се припокрива от отделните пиксели, чрез оценката $d = F(\mathrm{M})$ от алгоритъма на средната точка, където функцията на отсечката е дефинирана с [2.6]. Това би ни помогнало да използваме представените вече алгоритми и за изглаждане.

Първо нека се спрем на случая, в който отсечката припокрива само 2 пиксела от един растерен стълб. Разстоянието от средната точка до отсечката (по-точно до средната линия на правоъгълника, представящ отсечката) е пряко свързано със стойността на оценката:

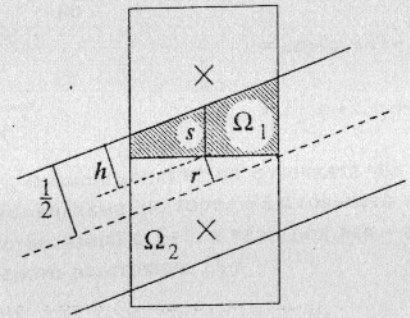
$$r = \frac{F\left(x_i+1, y_i+\frac{1}{2}\right)}{2\sqrt{dx^2+dy^2}} = \frac{d}{2\sqrt{dx^2+dy^2}}.$$

Площта на трапеца, който се отсича от горния пиксел, е равна на дължината на средната му отсечка (тъй като височината му е 1 пиксел), която пък можем да изразим чрез наклона на отсечката, която изглаждаме:

$$\Omega_1 = 1 \cdot s = \frac{\sqrt{dx^2+dy^2}}{dx} h = \frac{\sqrt{dx^2+dy^2}}{dx}\left(\frac{1}{2}-r\right) = \frac{\sqrt{dx^2+dy^2}}{2dx}\left(1-\frac{d}{\sqrt{dx^2+dy^2}}\right)$$

$$\Omega_2 = \frac{\sqrt{dx^2+dy^2}}{2dx}\left(1+\frac{d}{\sqrt{dx^2+dy^2}}\right)$$

Доста по-сложен е случаят, когато отсечката се разполага върху три пиксела от един растерен стълб. Тогава припокритите площи няма да имат трапецовидна форма. Зависимостта между оценката d и тези площи ще бъде квадратична, което би затруднило пресмятането им. Поради тази причина можем да разпределяме площта само между двата пиксела **T** и **S** - (виж фиг. 2-6), които се разглеждат на всяка стъпка от алгоритъма и да включваме трети пиксел само ако някоя от площите стане по-голяма от 1 (може да се види, че това е по-силно от условието допълващата площ да е по-малка от 0). Просто правило, което можем да приемем, е да осветяваме пиксела над разглежданата двойка с интензитет пропорционален на площта, с която по-горният пиксел превишаваща 1. Аналогично ще осветяваме пиксела под двойката пропорционално на площта, с която по-долният от двойката надхвърля 1.



Фиг. 2-30

Тъй като припокритите площи се получават в интервала $[0,\sqrt{2}]$, за да ги изобразим върху множеството на възможните интензитети $[0,J_{\max}]$ трябва да разделим на $\sqrt{2}$ и умножим по $J_{\max}$ изведените формули за Ω_1 и Ω_2.

Това дава достатъчно добра апроксимация на интензитетите на припокритите площи:

$$I_{1,2} = \frac{J_{max} \cdot \Omega}{2\sqrt{2}} \left(1 \pm \frac{d}{\sqrt{dx^2 + dy^2}} \right).$$

Промяната, която е необходимо да се направи в главния цикъл на програмата, реализираща алгоритъма на средната точка е показана във фрагмента по-долу. Оценката на припокритата площ е удобен метод за изглаждане на контура на запълнен многоъгълник. В този случай трябва да се оцени площта или само на един или на два пиксела, които едно ребро от контура пресича.

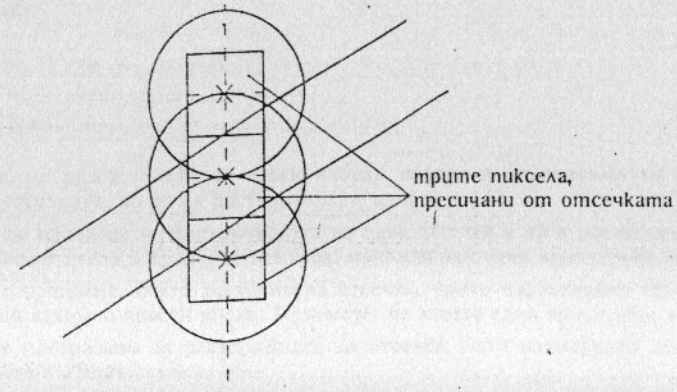
Накрая ще отбележим, че тази сравнително нетривиална апроксимация има смисъл само ако броят на различните интензитети, които можем да зададем на един пиксел е сравнително голям. Това далеч не е така при конвенционалните растерни дисплеи. Ако работим с растерен дисплей, в който за всеки пиксел са отделени по 2 бита (4 възможности), най-удачно е за апроксимация на интензитета да се използва самата оценка d.

```
. . .
norm=1.0/sqrt((double)(dx*dx+dy*dy));
area=1/(norm*dx);
sqrt2=sqrt(2.0);
while (n--) {
      x+=incX;
      if (d>0) { v[1]=area*(1+d*norm)/2;
                 d+=incUP; y+=incY; other=2; empty=0;
      }  else  { v[1]=area*(1-d*norm)/2;
                 d+=incDN; other=0; empty=2;
      }
      if (v[1]>1) { v[empty]=v[1]-1;
                    v[1]=1-v[empty];
                    v[other]=area-v[1]-v[empty];
      }   else    { v[empty]=0;
                    v[other]=area-v[1];
      }
      for (i=0, yc=y+incY; i<3; i++, yc-=incY) {
          if (v[i])
             if (reverse)
                PutPixel(yc,x,(int)(MAXINTENS*(v[i]/sqrt2)));
             else
                PutPixel(x,yc,(int)(MAXINTENS*(v[i]/sqrt2)));
      }
}
. . .
```

2.4.2 Изглаждане чрез отчитане на разстоянието до примитива

Можем да кажем, че съществува пряко пропорционална зависимост между осветеността на един пиксел и неговата близост до елемента, който той апроксимира. Друг начин за определяне на интензитета на един пиксел е да намерим разстоянието от неговия център до елемента и да му зададем интензитет обратно пропорционален на това разстояние. Естествено не бива да проверяваме разстоянието на всички пиксели от растера, затова първата задача е

да изберем максималното разстояние, което ще предизвиква осветяване на пиксела. Обикновено се избира това разстояние да е 1 пиксел. Можем да си мислим, че в центъра на всеки пиксел е разположен кръг с радиус 1 и пикселът ще бъде осветен съобразно площта, която отсечката припокрива от този кръг.



Фиг. 2-31

На фиг. 2-31 се вижда, че отсечката под ъгъл не повече от 45 градуса ще пресича най-много три такива кръга. Освен избраната точка в алгоритъма за растеризиране, отсечката пресича и кръговете на двата съседни пиксела от същия растерен стълб. Както и в предния случай, ще изразим разстоянието от избрания пиксел до отсечката чрез оценката d.

Нека на i-тата стъпка предстои да се избере пикселът T (фиг. 2-6). Разстоянието от неговия център ще е нормираната стойност на функцията на отсечката в този пиксел:

$$D_T = \frac{F(x_i + 1, y_i)}{2\sqrt{dx^2 + dy^2}}, \quad \text{но} \quad F(x_i + 1, y_i) = 2dy(x_i + 1) - 2dx \cdot y_i + 2c$$

$$= 2dy(x_i + 1) - 2dx\left(y_i + \frac{1}{2}\right) + dx + 2c$$

$$\Rightarrow \quad D_T = \frac{d + dx}{2\sqrt{dx^2 + dy^2}}. \qquad = F\left(x_i + 1, y_i + \frac{1}{2}\right) + dx = d + dx$$

Освен този пиксел, правоъгълникът на отсечката ще пресича и двата му съседни пиксела, както видяхме по-горе. Разстоянието от съседните й две точки (тази над нея е отбелязана с T+1, а тази под нея - с T–1) можем да изразим чрез вече полученото разстояние D_T:

$$D_{T+1} = \frac{F(x_i + 1, y_i + 1)}{2\sqrt{dx^2 + dy^2}} = \frac{d - dx}{2\sqrt{dx^2 + dy^2}} = D_T - \frac{dx}{\sqrt{dx^2 + dy^2}}$$

$$D_{T-1} = \frac{F(x_i + 1, y_i - 1)}{2\sqrt{dx^2 + dy^2}} = \frac{d + 3dx}{2\sqrt{dx^2 + dy^2}} = D_T + \frac{dx}{\sqrt{dx^2 + dy^2}}$$

Аналогично, ако пикселът избран на i-тата стъпка е S, ще получим следното:

$$D_S = \frac{d-dx}{2\sqrt{dx^2+dy^2}}, \quad D_{S+1} = D_S - \frac{d-dx}{2\sqrt{dx^2+dy^2}}, \quad D_{S-1} = D_S + \frac{d-dx}{2\sqrt{dx^2+dy^2}}$$

Трябва да отбележим, че всички изчислени по този начин разстояния са ориентирани и затова е необходимо да вземаме тяхната абсолютна стойност при определянето на интензитета.

Естествено е, интензитетът, съответстващ на определено разстояние, да не се изчислява всеки път - за всеки пиксел и за всяка нова отсечка. Тъй като броят на различните интензитети е краен и дори ограничен, възможно е да се състави таблица, в която на всяко от предварително фиксиран набор от разстояния е съпоставен интензитет. Това е и основната идея в алгоритъма на Гупта-Спрул (Gupta-Sproull), където подходящо закръгленото разстояние е входен индекс в таблица от предварително изчислени интензитети. В този алгоритъм, точно както и в алгоритъма на Брезенхам, на всяка стъпка получаваме само по един пиксел (който избираме съобразно знака на оценката), но вместо да осветяваме него, ще осветим пиксела, който трябва да изберем като следващ, заедно с неговите два вертикални съседа.

```
#define DistPixel(x,y,D) \\
        PutPixel(x,y,IntTable(RoundDist(fabs(D))))

void SampledDistanceAntialiasedLine(X1,Y1,X2,Y2)
int X1,Y1, X2,Y2;
{. . . /* променливите в алгоритъма на Брезенхам */
 float D,diff,norm;
    . . .
    /* инициализацията в алгоритъма на Брезенхам */
    . . .
    n=dx;
    norm=1/(2.0*sqrt((double)(dx*dx+dy*dy)));
    diff=2*dx*norm;
    if (reverse) DistPixel(Y1,X1,0.);
       else      DistPixel(X1,Y1,0.);
    while (n--) {
          x+=incX;
          if (d>0) { D=(d-dx)*norm;
                     d+=incUP; y+=incY;
          }   else { D=(d+dx)*norm;
                     d+=incDN;
          }
          if (reverse) {
             DistPixel(y,x,D);
             DistPixel(y+1,x,D-diff);
             DistPixel(y-1,x,D+diff);
          } else {
             DistPixel(x,y,D);
             DistPixel(x,y+1,D-diff);
             DistPixel(x,y-1,D+diff);
          }
    }
}
```


Подобна стратегия може да се приложи почти буквално за изглаждане на окръжност използвайки симетрията ѝ, както това бе показано при растеризирането ѝ. Когато се изглажда контурът на многоъгълник, трябва да се отчете от коя страна се намира вътрешността му, за да не се разглеждат пикселите, попадащи там.

Задачи

2.1 Докажете, че разстоянието от всеки избран пиксел в алгоритъма на средната точка до отсечката, която се растеризира, е винаги < 1/2.

2.2 Може ли да използвате симетричността на една отсечка и да я растеризирате едновременно от двата ѝ края към центъра, използвайки една единствена оценка?

2.3 Напишете програма, която растеризира отсечка, чиито нараствания по всяка от осите не са взаимно прости числа. Приемете, че знаете един техен общ делител.

2.4 Напишете програмата за растеризация на отсечка, като използвате алгоритъма на порциите в общия случай.

2.5 Напишете програма, която растеризира контура на многоъгълник, така че никой пиксел да не се записва по два пъти. Това може да е извънредно важно при използване на режим на растерно записване "изключващо или".

2.6 Напишете програма за ефективна растеризация на окръжност използвайки параметричното ѝ уравнение, без да пресмятате на всяка стъпка тригонометрични функции.

2.7 Напишете програма за растеризация на окръжност чрез растеризация на отсечките (използвайки алгоритъма на Брезенхам) на правилния многоъгълник, който апроксимира тази окръжност.

2.8 Използвайки казаното в началото на 2.1.3 предложете алгоритъм за позициониране на точка върху окръжност, като анализирате грешката и в диагонално разположената точка и отчетете възможността за смяна на знака на грешката при движение.

2.9 Напишете програма за растеризация на дъга от елипса с оси, успоредни на координатните.

2.10 Напишете програма за растеризиране на изправен правоъгълник със заоблени върхове като знаете координатите на две диагонално разположени точки и радиуса на заобляне. Осигурете записването на стойността на всеки пиксел само по веднъж.

2.11 Напишете програма за запълването на изправен правоъгълник със заоблени върхове.

2.12 Добавете към програмата за запълване на многоъгълници възможност за отстраняване стъпаловидността на контура му.

2.13 Модифицирайте алгоритъма за запълване на многоъгълник, така че границите му да се растеризират независимо от това в какво положение спрямо вътрешността му се намират.

2.14 При запълване на многоъгълник, който има много близки един до друг ръбове, които при растеризация имат съвпадащи точки, но спрямо които вътрешността се намира от различни страни по правилото, прието в алгоритъма на сканиращия ред може да се случи в един ред началната точка да се намира след крайната. Модифицирайте програмата, така че да се справя и с този частен случай.

2.15 Напишете програма за запълване на многоъгълник с образец, който е свързан с многоъгълника, а не с екрана.

2.16 Напишете програма за запълване на окръжност, използвайки алгоритъма на Брезенхам за получаване на началната и крайната точка на всеки сканиращ ред.

2.17 Напишете програма за растеризиране на отсечка с определена дебелина като използвате принципа на движещото се перо, така че пикселите да се записват само по веднъж.

2.18 Напишете програма за растеризиране на отсечка със зададен тип на линията като образец и определена дебелина.

2.19 Напишете програма за удебеляване на окръжност, като се постарате да отстраните стъпаловидния ефект.

Глава 3

Визуализация на равнинни обекти

В предната глава разгледахме подробно основните алгоритми за изобразяване на графични примитиви върху растерни устройства. Тези алгоритми се кодират на възможно най-ниско ниво - програмистите могат да гледат на тях като на неразделна част от работната станция, дисплея или плотера, които те използват. Освен всичко, тази обвързаност с конкретното устройство се определя и от факта, че алгоритмите работят в координатната система на растера, размерите и организацията на поддържането на който са специфични за всяко устройство.

При проектирането и създаването на една графична система се поставя целта възможностите за визуализация да са:

- възможно най-малко зависими от конкретното графично устройство;
- колкото се може по-общи и по-близки до обектите, които се изобразяват.

В тази глава ние ще разгледаме общите принципи на графичните системи, илюстрирайки ги с представения *"Базов Графичен Пакет"*. Така ще наричаме съвкупността от ограничен брой графични функции, които могат да се използват за основа при разработването на приложни графични програми. Съставът на БГП сме избрали без да се придържаме към който и да е стандарт за графична система, макар че въведените тук понятия могат да бъдат открити във всеки един от използваните в момента графични пакети. В тази глава ние ще се спрем подробно само на обслужването на графичния изход за нуждите на приложните програми. Графичният вход, който е основата на графичния диалог, е обект на следващата глава.

Накрая на тази глава ще се спрем и на някои средства за структуриране на изображението, макар че те едва ли са типични за простите графични пакети. Целта ни е да подготвим читателя за използване на възможностите за моделиране с графичните системи, на което пък е посветена пета глава.

3.1 ПОТРЕБИТЕЛСКИ И ЧЕРТОЖНИ КООРДИНАТИ

Една от задачите на графичната система, които изброихме във въведението, е да предостави на приложната програма възможност за работа в координатна система, типична за модела, който тя обработва.

Пространството, в което е дефиниран геометричният модел на една приложна програма е обикновено пространството на самите моделирани обекти - пространството на реалния свят. Координатната система, в която се определя положението, ориентацията и размерите на тези обекти, може да бъде много различна в различните приложения. В огромна част от случаите тя е правоъгълна (декартова). Изборът на началото и ориентацията ѝ зависят от естеството на моделираните обекти.

Координатното пространство, в което работи една приложна програма ще наричаме *потребителско пространство*. Често използван термин е и *моделно пространство*, тъй като това е пространството, в което е дефиниран геометричният модел. Координатите на обектите от това пространство ще наричаме *потребителски координати*. В тази книга ще избягваме буквалния превод на термините *World Space* и *World Coordinates* като *световно пространство* и *световни координати*, макар че те добре отразяват факта, че това е пространството, в което обектите реално съществуват.

Когато приложната програма визуализира модели на физически обекти, а не абстрактни математически структури, потребителските координати имат и размерност. Това са единиците, в които се измерват разстоянията в моделите. Приложенията за машиностроенето работят обикновено в милиметри, архитектурните приложни програми могат да работят в инчове или сантиметри, програмите за проектиране на интегрални схеми - в микрони и нанометри, а астрономичните - в светлинни години. Общото за всички тях от гледна точка на програмиста е, че това са координати, които трябва да се представят с числа с плаваща запетая. При визуализация ние не се интересуваме от размерността на потребителските координати - тази размерност има значение за моделирането (например при пресмятане на физични величини за обекти в модела като маса, инерчен момент и др.)

Да вземем конкретен пример. Една програма за проектиране на вътрешно обзавеждане на стаи най-често дефинира декартова координатна система, чието начало съвпада с някой от ъглите на стаята, а осите *Ox* и *Oy* съвпадат със стените, които се срещат в този ъгъл. Ако използваните единици за измерване са милиметри, то двойката потребителски координати на една точка в така дефинираното пространство отговарят на разстоянията в милиметри от тази точка до двете стени, които лежат върху координатните оси.

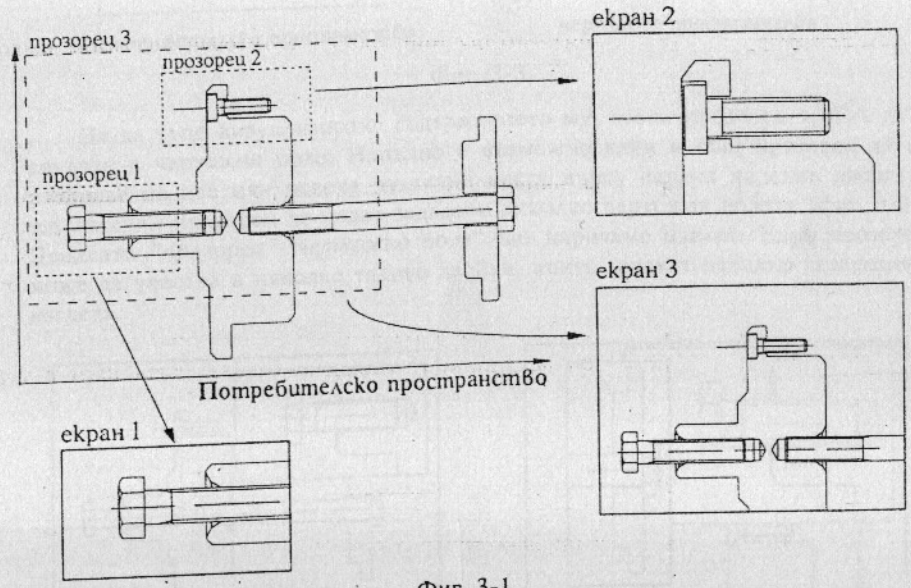
От друга страна, алгоритмите, които разгледахме в предишната глава извършват растеризиране в координатното пространство на графичното устройство. То се дефинира от координатна система, която също е декартова, но е целочислена и началото ѝ е най-често в началото на *работното поле* на устройството. Под *работно поле* ще разбираме максималната област, която графичното устройство може да използва за визуализация. Това са: целият екран на графичния дисплей; тази част от хартиения лист, на който плотера може да рисува и т.н. Координатното пространство, което е определено върху работното поле на графичното устройство ще наричаме *чертожно пространство*, а координатите в него - *чертожни координати*. Съответните английски термини, които се използват в графичните стандарти са *Device Space* и *Device Coordinates*. Често могат да се чуят още и термините *екранно пространство* и

съответно *екранни координати*, но трябва да имаме пред вид, че графичното устройство може да не е непременно дисплей с екран.

3.1.1 Потребителски прозорец

Приложната графична програма не винаги визуализира всички обекти, зададени в потребителското пространство. В повечето случаи работата на потребителя на такава програма е съсредоточена в определена област от това пространство. Ако задачата на програмата например е да създаде план на един град, то не е нужно да се изобрази целият план, ако в даден момент се моделира един сравнително малък жилищен комплекс. Чертожникът би искал в тази ситуация да вижда само част от плана, а именно един правоъгълник, съдържащ споменатия жилищен комплекс в по-едър мащаб.

Правоъгълникът от потребителското пространство, обектите в който в определен момент се визуализират от приложната програма и в който са съсредоточени графичните операции, се нарича *потребителски прозорец*. Всички геометрични примитиви или частите от тях, които се намират във вътрешността му са видими, докато всички останали не се включват в изображението.



Фиг. 3-1

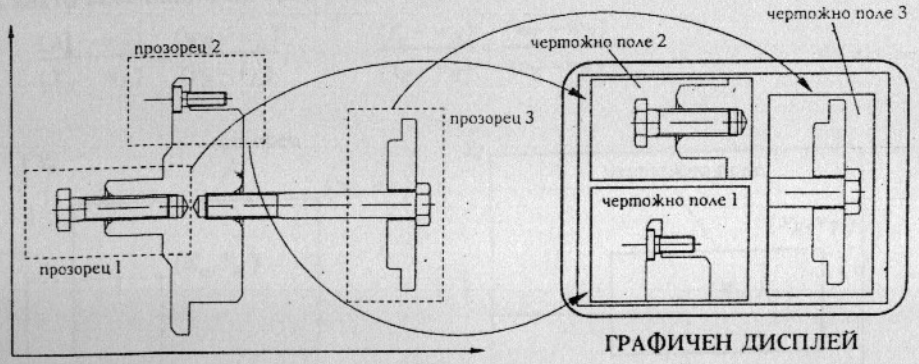
На фиг. 3-1 е показан геометричният модел на един машинен детайл. В него са дефинирани няколко различни потребителски прозореца, които са напълно независими един от друг. Можем да си представим, че всеки от тях е изобразен върху целия екран на три различни графични дисплея, давайки възможност на различни потребители да извършват различни дейности с отделни части от този модел. Самите прозорци не задават това къде и как да

бъдат изобразени върху графичното устройство. Те определят само коя част от модела ще бъде визуализирана и затова и границите им се задават в координатното пространство на модела - в потребителски координати.

Тук е важно да отбележим разликата между това понятие и широко използвания термин *прозорец* (означаващ *прозорец на процес*) в многозадачните системи с управление чрез прозорци (Window Management Systems). Това са две съвсем различни неща, които за нещастие носят еднакви имена. Ние ще се спрем подробно на последното в следващата глава, а тук ще се условим да използваме определението *потребителски* всеки път, когато има опасност от двусмислие. Ако такава опасност няма, *прозорец* ще означава *потребителски прозорец*. Двусмислието идва от английския термин *window*, който също се използва за обозначаването и на двете неща.

3.1.2 Чертожно поле. Изглед

За да визуализираме съдържащите се в един потребителски прозорец обекти върху конкретно графично устройство е необходимо да зададем в каква част от работното поле на това устройство да се изобрази този прозорец. В примера по-горе казахме, че всеки от прозорците е показан върху целия екран на три отделни дисплея (фиг. 3-1). Ако искаме трите прозореца да се виждат едновременно върху екрана на един и същ дисплей, тогава е необходимо да зададем как се разпределя работното поле на устройството между трите прозореца. Това разпределение ще зададем като за всеки прозорец дефинираме по един нов правоъгълник (този път върху работното поле на дисплея), в който да се покажат видимите части от модела за всеки прозорец (фиг. 3-2).

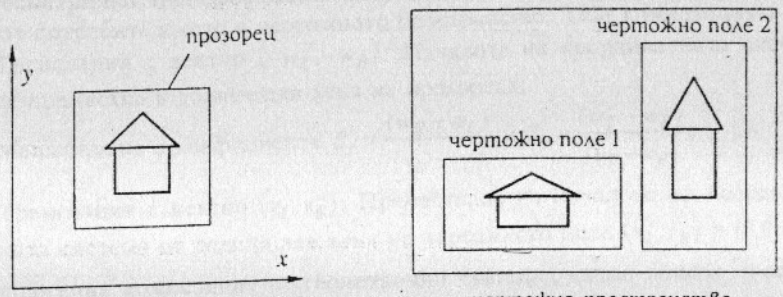


Фиг. 3-2

Правоъгълникът от работното поле на графичното устройство, в който се изобразява съдържанието на един прозорец ще наричаме *чертожно поле*. Често се използват имена като *поле за визуализиране*, *чертожна област*, *област на индикация* и др., които са различните преводи на приетия английски термин *viewport*. Границите на чертожното поле се задават в целочислените чертожни координати и са пряко свързани с конкретното устройство. Размерите

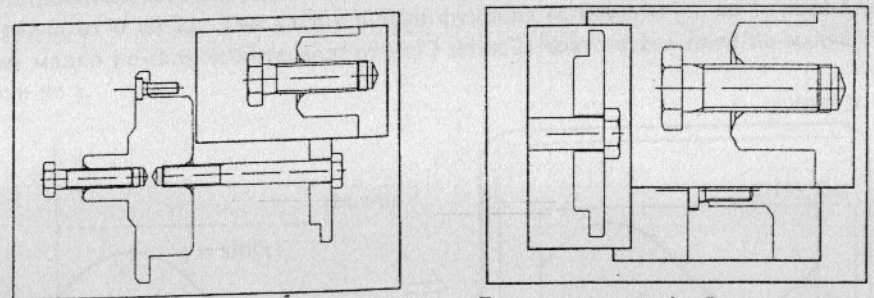
на чертожното поле естествено не бива да превишават размерите на работното поле на самото устройство. В частност, чертожното поле може да съвпада с целия екран на един графичен дисплей.

Необходимо е да кажем, че ако графичната система е част от система за управление чрез прозорци, то чертожната координатна система ще бъде свързана не с екрана, а със съответния *прозорец на процеса*. На някои особености при визуализацията в този случай ще се спрем в следващата глава. Тук ще считаме, че ако работим с графичен дисплей, то разполагаме с целия му екран.



Фиг. 3-3

За да бъде визуализирано съдържанието му, всеки прозорец трябва да е свързан с чертожно поле. Напълно е възможно един и същ прозорец да се изобрази на две или повече различни места върху екрана на един дисплей, т.е. за един прозорец да бъдат зададени няколко чертожни полета (фиг. 3-3). Двойката "*прозорец - чертожно поле*" ще наричаме *изглед*. Един прозорец може да участва в няколко такива двойки, които задават няколко независими изгледа.



Фиг. 3-4

На фиг. 3-4 са показани няколко изгледа на машиностроителния детайл от горните примери. На лявата рисунка върху екрана са показани едновременно два изгледа. Първият се състои от прозорец, обхващащ целия детайл, а

чертожното поле е целият екран. Този изглед е припокрит частично от втори изглед с по-малък прозорец от потребителското пространство, обхващащ болта в лявата част на детайла. Чертожното поле на този изглед заема част в горния десен ъгъл на екрана.

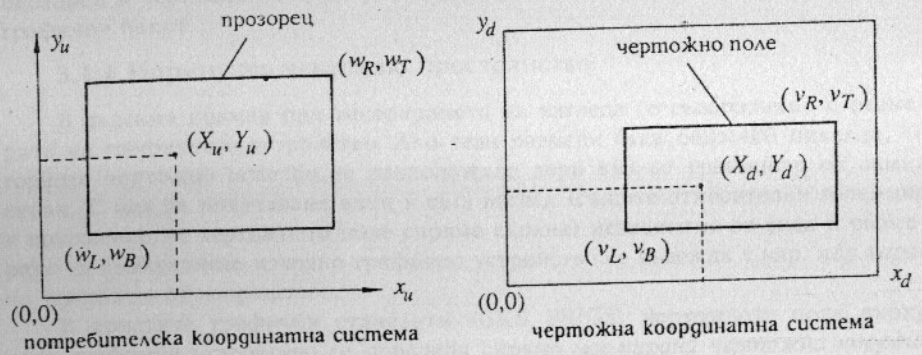
На рисунката отдясно три различни прозореца са визуализирани в три припокриващи се чертожни полета. Да отбележим, че в общия случай, когато отношението на ширина към височина на правоъгълниците на прозореца и чертожното поле са различни, визуализираните обекти ще бъдат мащабирани различно по x и по y както това се вижда на фиг. 3-3.

3.1.3 Трансформация на изгледа

Всеки изглед определя координатна трансформация, която съпоставя на всяка точка от потребителското пространство съответна точка от чертожното пространство. Ще покажем начина, по който се определя тази трансформация.

Нека (X_u, Y_u) е точка от потребителското пространство, която се намира във вътрешността на потребителския прозорец и нека образът ѝ в чертожното поле е (X_d, Y_d). Относителните координати на първата точка спрямо долния ляв ъгъл на прозореца ще бъдат $(X_u - w_L, Y_u - w_B)$, а относителните координати на втората спрямо същия ъгъл на чертожното поле - $(X_d - v_L, Y_d - v_B)$. Относителните x-координати на двете точки се отнасят една към друга както ширината на прозореца $(w_R - w_L)$ към ширината на чертожното поле $(v_R - v_L)$. Аналогично, относителните y-координати на точките се отнасят една към друга както височините на прозореца и чертожното поле.

$$\frac{(X_u - w_L)}{(X_d - w_L)} = \frac{(w_R - w_L)}{(v_R - v_L)}, \qquad \frac{(Y_u - w_B)}{(Y_d - v_B)} = \frac{(w_T - w_B)}{(v_T - v_B)}$$



Фиг. 3-5

От тези две зависимости можем веднага да получим формулите за преобразуване на всяка точка от потребителското в чертожното пространство:

$$X_d = \frac{(v_R - v_L)}{(w_R - w_L)}(X_u - w_L) + v_L, \qquad Y_d = \frac{(v_T - v_B)}{(w_T - w_B)}(Y_u - w_B) + v_B \qquad [3.1]$$

и обратно - от чертожното в потребителското пространство:

$$X_u = \frac{(w_R - w_L)}{(v_R - v_L)}(X_d - v_L) + w_L, \qquad Y_u = \frac{(w_T - w_B)}{(v_T - v_B)}(Y_d - v_B) + w_B.$$

Всяка от тези две координатни трансформации е композиция от три основни геометрични трансформации. Да разгледаме първата от тях - изобразяването от потребителското в чертожното пространство. Тя е композицията от:

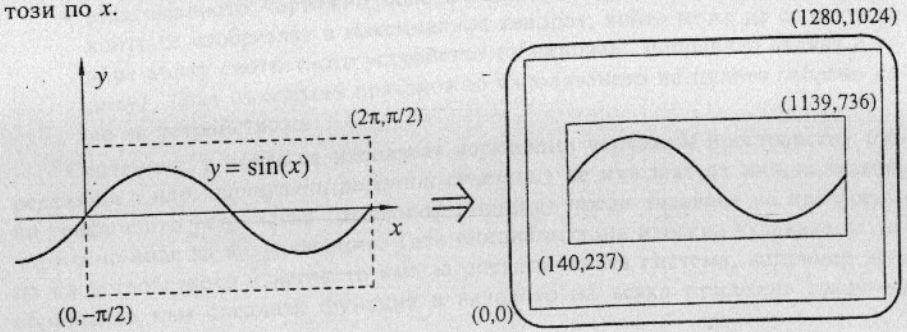
1. Транслация с вектор $(-w_L, -w_B)$: Началото на координатната система се премества в долния ляв ъгъл на прозореца;
2. Мащабиране с коефициенти $S_x = \frac{(w_R - w_L)}{(v_R - v_L)}$ и $S_y = \frac{(w_T - w_B)}{(v_T - v_B)}$;
3. Транслация с вектор (v_L, v_B): Преместване на началото на координатната система от долния ляв ъгъл на чертожното поле (v_L, v_B) в (0,0).

Аналогична композиция от геометрични трансформации описва преобразуването на чертожни в потребителски координати.

От казаното дотук става ясно, че за да се дефинира един изглед и съответните му координатни трансформации е необходимо задаването на осемте параметъра, които определят границите на прозореца и чертожното поле: $w_L, w_B, w_R, w_T, v_L, v_B, v_R, v_T$.

Да разгледаме един прост пример.

Нека задачата ни е да визуализираме функцията $y = \sin(x)$ в някакъв интервал на x върху екрана на графичен дисплей, чийто размери са 1280 на 1024 пиксела. Координатната система, в която е зададена функцията $y = \sin(x)$ е потребителската координатна система. За x можем да изберем например интервала от 0 до 2π. Тъй като y в тази функция се мени от –1 до 1, нека вземем малко по-широк интервал: от $-\pi/2$ до $\pi/2$, който е два пъти по-малък от този по x.



Фиг. 3-6

Така вече сме определили границите на прозореца:

$$w_L = 0, \quad w_R = 2\pi \qquad w_B = -\pi/2 \qquad w_T = \pi/2.$$

За да запазим пропорциите на изображението (мащабирането по всяка от осите да е еднакво), нека да изберем чертожно поле с ширина например 1000 пиксела и височина, която е два пъти по-малка: 500 пиксела. За да разположим това поле в средата на екрана трябва да изберем следните граници (вземайки пред вид и включването на крайните пиксели):

$$v_L = 1280/2 - 1000/2 = 140 \qquad v_R = (1000-1)+140 = 1139$$
$$v_B = 1024/2 - 500/2 = 237 \qquad v_T = (500-1)+237 = 736$$

Задаването на изглед е абсолютно необходимо преди започване на каквато и да е визуализация. За това е необходимо в приложната програма да се зададат потребителският прозорец и чертожното поле на този изглед. Прозорецът и чертожното поле не са неизменни по време на работата на една приложна графична програма. В процеса на взаимодействието с потребителя много често се налага да се промени едното и/или другото, определяйки по този начин нов изглед. Някои от случаите, в които се прави това са следните:

- *Уголемяване на изображението*: визуализация на по-малък прозорец (съдържащ се в текущия) в същото чертожно поле, при което се вижда по-малка част от модела, но с по-голяма детайлност;
- *Съсредоточаване на работата върху друга част от модела*: дефиниране на нов прозорец, обхващащ новата част и изобразяването му в същото или ново чертожно поле;
- *Общ изглед на модела*: дефиниране на прозорец, обхващащ целия модел и визуализирането му най-често в цялото работно поле на устройството.

Особеностите на програмната реализация на функциите за задаване на прозорец и чертожно поле ще разгледаме по-късно при описанието на базовия графичен пакет.

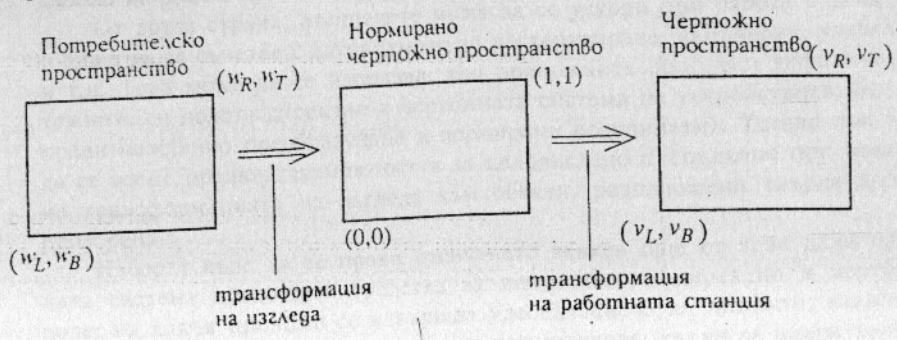
3.1.4 Нормирано чертожно пространство

В дадения пример при генерирането на изгледа се съобразихме с размерите на графичното устройство. Ако тези размери бяха 640×400 пиксела, то горното чертожно поле би се разположило дори вън от границите на самия екран. С цел да получаваме един и същ изглед (същите относителни големина и положение на чертожното поле спрямо екрана) независимо от вида и размерите на конкретното изходно графично устройство се въвежда т.нар. *нормирано чертожно пространство*.

В приетите графични стандарти (GKS, PHIGS) чертожното поле върху едно графично устройство се определя спрямо *нормирана чертожна координатна система* (Normalised Device Space), в която координатите по всяка от осите се изменят от 0 до 1. С това практически се дефинира едно виртуално графично устройство, чиито размери са винаги едни и същи. Така се дава възможност всички графични устройства да се третират по един и същ начин

от приложните графични програми.

Графичната система е тази, която определя трансформацията, която трябва да се извърши при преход от нормираната чертожна система към чертожната система на конкретното устройството. Тя носи името *трансформация на работната станция* (Workstation Transformation).



Фиг. 3-7

Тук има една особеност, на която заслужава да се обърне внимание. За повечето от графичните устройства максималното възможно чертожно поле (работното поле) далеч не е квадрат. Изобразяването на единичния квадрат в екрана на един графичен дисплей би довело до изкривяване на образа, тъй като мащабирането по осите би било различно. За да се избегне това, в различните системи се приемат различни условности:

- Размерите на максималното чертожно поле са 1 по x, а за размера по y се взема отношението на височината към ширината на работното поле:

  $$\frac{d_T - d_B}{d_R - d_L},$$

  което е най-често по-малко от единица. В този случай е необходимо приложната програма да може да получава информация от графичната система за конкретните параметри на изходните ѝ устройства;
- Максималното чертожно поле е винаги квадрат (единичния квадрат), който се изобразява в максималния квадрат, който може да се визуализира върху съответното устройство (обикновено подравнен отляво и отдолу). Това пък прави невъзможно използването на цялото работно поле на устройството.

Системите, в които се използват нормирани чертожни пространства (определящи т.нар. *виртуални работни станции*) се нуждаят от инициализация на графичното устройство (работната станция) преди задаване на прозорец и чертожно поле за визуализация. Тази инициализация изисква указване на типа на устройството и неговото име за операционната система, например чрез обръщение към следната функция в началото на всяка приложна графична програма:

```
OpenWorkstation(device_id, device_type)
```

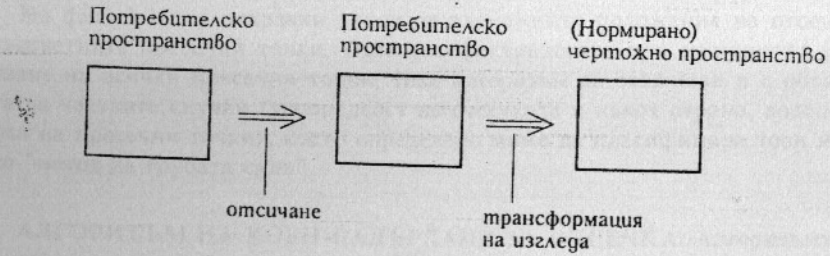
Това предизвиква генерирането на подходяща трансформация на работната станция и избора на съответния драйвер за обслужването на графичния изход. По-долу са показани няколко възможности за инициализация на графичното устройство: екран; плотер; печатащо устройство; файл за плотер и *прозорец на процес* в X Window System:

```
OpenWorkstation(SCREEN, VESA_SVGA_DISPLAY)
OpenWorkstation("COM1", HPGL_PLOTTER_A4)
OpenWorkstation("LPT1", COLOUR_POSTSCRIPT_PORTRAIT)
OpenWorkstation("/pub/pictures/myplot.plt", HPGL_PLOTTER)
OpenWorkstation(XtWidgetofWindow(dpy,main), MOTIF_WIDGET)
```

3.2 ОТСИЧАНЕ НА ГРАФИЧНИ ПРИМИТИВИ

Следващата задача, която трябва да се реши от графичната система, след като е определена трансформацията на изгледа, е да се осигури визуализацията само на тази част от модела, която попада в дефинирания прозорец. Тази задача е известна като задача за *вътрешно отсичане на графични примитиви* или само *отсичане* (clipping). Отсичане се налага да се прави най-често спрямо правоъгълник със страни успоредни на координатните оси, затова и тук ще обърнем по-голямо внимание на алгоритмите за намиране на тази част от един примитив (отсечка, дъга, многоъгълник), която лежи във вътрешността на изправен правоъгълник. В края на тази част ще се спрем накратко и на задачата за отсичане спрямо многоъгълник, което се налага при визуализация на обекти в чертожно поле, част (или части) от което са припокрити от други чертожни полета или от други *прозорци на процеси* (вж. фиг. 3-4). В такива случаи трябва да се решава и задачата за *външно отсичане*, която е обратна на горната, а именно: намирането на тази част от обект, която е вън от дадена област.

Отсичането може да се извършва в потребителското пространство - тогава отсичащият правоъгълник е този на прозореца. То може също така да се извършва и в чертожното пространство (или нормираното чертожно пространство) - тогава отсичането е спрямо правоъгълника на чертожното поле.

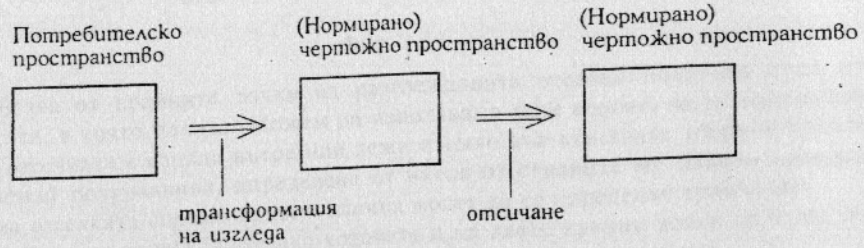


Фиг. 3-8

В какви координати да се извършва отсичането зависи от конкретната графична система. Ако в нея се използва нормирано чертожно пространство, тогава за предпочитане е отсичането да се прави в потребителски координати. Тъй като в този случай винаги се работи с числа с плаваща запетая, отсичането на по-ранен стадий в процеса на визуализация ще е по-икономично. Последователността в изпълнението на операциите за визуализация тогава е показана на фиг. 3-8.

От друга страна, отсичането може да се ускори при работа с цели числа и даже да се съчетае с алгоритмите за растеризиране, запълване, удебеляване и т.н. Това може да се използва, ако приложните програми дефинират чертожните си полета директно в чертожната система на устройството (без да се прави междинно преобразуване в нормирани координати). Тогава пък трябва да се вземе предвид възможността за целочислено препълване при прилагане на трансформацията на изгледа към обекти, разположени твърде далеч от прозореца.

Изборът къде да се прави отсичането зависи още от това дали графичната система предоставя средства за визуализация директно в чертожното поле; на какъв принцип се извършва удебеляването на линиите; какви алгоритми се ползват за растеризиране на примитивите; какви са апаратните възможности на устройството и т.н. Тук ще се спрем на методите за решаването на тази задача без да конкретизираме в кое пространство тя се извършва. Няма да отчитаме особеностите при растерно отсичане, а ще разгледаме задачата като чисто аналитична.



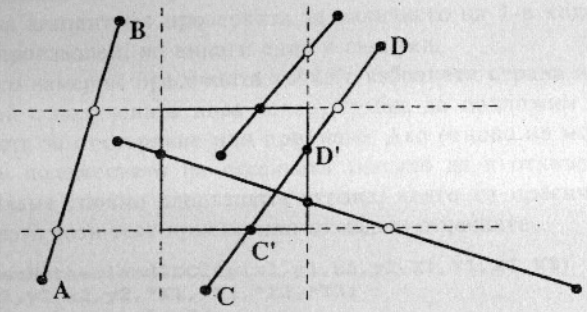
Фиг. 3-9

3.2.1 Отсичане на отсечки от правоъгълник

Отсичането на отсечки е най-важната задача в този раздел. Представянето на всички видове примитиви чрез отсечки (многоъгълници, дъги, окръжности, криви) е характерно за много от по-простите графични пакети. При тях отсичането на отсечките, които представят всички такива примитиви е достатъчно за да се реши проблемът изцяло. Тук ще разгледаме няколко алгоритъма за намиране на частта от отсечка, която попада във вътрешността на правоъгълник със страни, успоредни на координатните оси (изправен правоъгълник). По-основна от тази задача е само задачата за определяне положението на точка спрямо такъв правоъгълник, която се прилага за крайните точки на всяка отсечка. Решаването на последната е тривиално: Точката с координати (x,y) е във вътрешността на правоъгълника, чиито граници са x_{min}, x_{max} и y_{min}, y_{max}, ако са изпълнени неравенствата:

$$x_{min} \le x \le x_{max} \quad \text{и} \quad y_{min} \le y \le y_{max}.$$

За растерни устройства има един елементарен, но достатъчно общ начин за отстраняване на тази част от един примитив, която не попада в чертожното поле. Той се състои в проверка за видимост на всеки пиксел (с използване на горните неравенства), който се генерира от растеризиращия алгоритъм и записване на стойността му (чрез обръщение към `PutPixel`) само ако той попада в отсичащия правоъгълник. По този начин могат да се отсичат всички примитиви, но това е приложимо само за растерни устройства и то когато отсичането става в чертожни координати.



Фиг. 3-10

Друг прост начин за отсичане, който също като горния е неефективен, се състои в следното:

1. Намираме пресечните точки на отсечката с всяка от правите, носещи четирите страни на правоъгълника;
2. Отстраняваме тези пресечни точки, които не попадат във вътрешността на интервалите x_{min}, x_{max} и y_{min}, y_{max};
3. Ако след горната стъпка са останали две пресечни точки, то те са крайщата на видимата част на отсечката. Ако пресечната точка е една, то тя заедно с този край, който се намира във вътрешността, образуват новата отсечка. Ако не - отсечката може да е изцяло вън или вътре.

На фиг. 3-10 са показани някои от възможните положения на отсечката и съответните пресечни точки. Освен неефективността при излишното изчисляване на всички пресечни точки, този алгоритъм се отежнява и с обработката на частните случаи (успоредност на отсечката и някоя страна, водещо до липса на пресечни точки), което определено може да класифицира този метод като "метод на грубата сила".

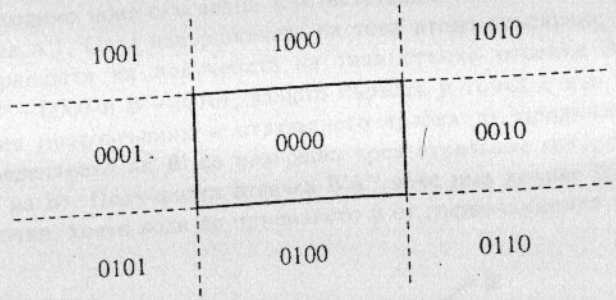
АЛГОРИТЪМ НА КОЕН-САДЪРЛАНД ЗА ОТСЕЧКА. Алгоритъмът на Коен-Садърланд (Cohen-Sutherland) е може би най-често разглежданият в литературата и най-много използван при разработването на базови графични пакети метод. Огромното достойнство на този алгоритъм е възможността:

- бързо да се отхвърлят голяма част от отсечките, които нямат пресечни точки със страните на правоъгълника;
- бързо да се приемат тези, които попадат изцяло във вътрешността му.

Обработката на тези два случая се нарича *тривиален тест за отхвърляне или приемане* и се осъществява по следния начин:

Правите, на които лежи всяка от страните на правоъгълника разделят равнината на 9 области, всяка от които може да се характеризира еднозначно от едно 4-битово число. Всеки бит от това число съответства на положението на областта спрямо всяка от четирите прави:

1-ви бит: 1, ако областта е над горната права - $y > y_{max}$
2-ри бит: 1, ако областта е под долната права - $y < y_{min}$
3-ти бит: 1, ако областта е отдясно на дясната права - $x > x_{max}$
4-ти бит: 1, ако областта е отляво на лявата права - $x < x_{min}$



Фиг. 3-11

Всяка от крайните точки на разглежданата отсечка получава кода на областта, в която попада. Можем да използваме тези кодове, за да определим дали отсечката е изцяло вътре или лежи изцяло във външната (спрямо правоъгълника) полуравнина, определена от някоя от страните му. Някои положения на отсечката спрямо правоъгълника могат да се определят тривиално:

1. Веднага се вижда, че ако кодовете и на двете крайни точки са нули, то отсечката лежи изцяло във вътрешността на правоъгълника.
2. Ако кодовете и на двете крайни точки съдържат единица в един и същ бит, то цялата отсечка лежи в невидимата полуравнина, определена от съответната страна. Такъв е случаят с отсечката **AB** на Фиг. 3-10. И двете точки имат 1 като 4-ти бит на кодовете си, което показва, че отсечката ще е изцяло отляво на правоъгълника. Аналогично, ако и двете крайни точки имат 2-ри бит 1, то отсечката ще е под правоъгълника и отново можем да я отхвърлим.

Да обобщим като кажем, че ако побитовото логическо "и" (**and**) на двата кода е различно от 0 (което съответства на наличие на 1 в един и същ бит от двата кода), то отсечката може да бъде отхвърлена.

Във всички останали случаи отсечката потенциално пресича правоъгълника. За да определим дали наистина има пресичане, да разделим отсечката на две части, и то такива, че едната със сигурност да е вън от правоъгълника. За останалата част ще приложим същите разсъждения от самото начало. Разделянето можем да направим като отрежем тази част, която лежи от външната страна на някоя от страните на правоъгълника. Изборът на отсича-

щата страна се определя от това дали в съответния бит на кода на някоя от крайните точки стои 1.

Например краят **C** на отсечката **CD** от същата фигура има код 0100, което позволява, имайки предвид интерпретацията на втория бит, да отсечем тази част, която се намира под правоъгълника. По този начин ще получим отсечката **C'D**.

При наличието на повече от една единица в кода на някоя от точките, скъсяването можем да извършим спрямо коя да е от съответните прави. При кодирането на алгоритъма проверката за наличието на 1 в кода можем да извършваме в произволен, но винаги един и същ ред.

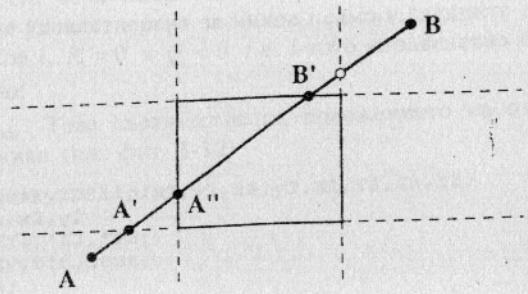
След като намерим пресечната точка с избраната страна и заменим съответния ѝ край с получената нова точка, трябва да подложим новата отсечка отново на теста за отхвърляне или приемане. Ако отново не можем тривиално да определим положението на отсечката (изцяло да я отхвърлим или приемем), я отсичаме спрямо следващата страна, която тя пресича. Ще продължим така докато този тест приеме или отхвърли отсечката.

```
int CohenSutherland2DClip(x1,y1,x2,y2,X1,Y1,X2,Y2)
float x1,y1,x2,y2,*X1,*Y1,*X2,*Y2;
{float s; char b,o1,o2;
   o2=(y2>Ymax)<<3+(y2<Ymin)<<2+(x2>Xmax)<<1+(x2<Ymin);
   while (1) {
      o1=(y1>Ymax)<<3+(y1<Ymin)<<2+(x1>Xmax)<<1+(x1<Xmin);
      if (!o1 && !o2) {
          *X1=x1;*Y1=y1;*X2=x2;*Y2=y2;
          return 1;      /* отсечката е изцяло във вътрешността */
      }
      if (o1 & o2) return 0;       /* отсечката е изцяло отвън */
      if (!o1) {
           b=o1; o1=o2; o2=b;
           s=x1; x1=x2; x2=s;
           s=y1; y1=y2; y2=s;
      }
      if (o1&8) {           /* отсичане отгоре */
          x1=x1+(x2-x1)*(Ymax-y1)/(y2-y1); y1=Ymax;
      } else if(o1&4){      /* отсичане отдолу */
          x1=x1+(x2-x1)*(Ymin-y1)/(y2-y1); y1=Ymin;
      } else if(o1&2){      /* отсичане отдясно */
          y1=y1+(y2-y1)*(Xmax-x1)/(x2-x1); x1=Xmax;
      } else {              /* отсичане отляво */
          y1=y1+(y2-y1)*(Xmin-x1)/(x2-x1); x1=Xmin;
      }
   }
}
```

Представената функция показва един от начините за реализиране на този алгоритъм. Отсичащият правоъгълник е зададен с глобалните променливи `Xmin,Ymin,Xmax,Ymax`. В тази функция двата края на отсечката се разменят, ако първият попада във вътрешността на правоъгълника, а вторият - не. Това позволява отрязването винаги да се извършва чрез скъсяване на първия край. В такъв случай изчисляването на кода на първия край е необходимо да се прави след всяко отрязване, тъй като междувременно той се променя.

Размяната на точките тук се извършва само за да се съкрати записа на алгоритъма. Той може да се реализира и с отделен анализ на кода на всяка от точките, което би направило текста на програмата малко по-дълъг (но малко по-малко времето за изпълнението ѝ).

Ще илюстрираме работата на алгоритъма като разгледаме примера, даден на фиг. 3-12. Двата края на отсечката **AB** имат кодове съответно: **A** – 0101 и **B** – 1010. И двата са различни от нула и логическото им "и" е 0. На първата стъпка ще се открие 1 във 2-рия бит на кода на **A**, което ще предизвика скъсяване отдолу до получаване на точката **A'**. Новата отсечка **A'B** (с кодове **A'** – 0001 и **B** – 1010) не може нито да се приеме, нито да се отхвърли напълно и ще е необходимо ново скъсяване (съответстващо на наличието на 1 в 4-тия бит в кода на **A'**). След извършването на това второ скъсяване ще трябва да разменим краищата на получената на тази стъпка отсечка **A"B** (нейните кодове са **A"** – 0000 и **B** – 1010), защото първата ѝ точка е във вътрешността на отсичащия правоъгълник и отрязването трябва да продължи от другата страна. Определянето на **B'** се извършва чрез отрязване отгоре (1 в първия бит на кода на **B**). Получената отсечка **B'A"** вече има кодове 0000 и за двете си крайни точки, което води до приемането ѝ от първоначалния тест.



Фиг. 3-12

Алгоритъмът на Коен-Садърланд не е най-бързият възможен алгоритъм, но популярността му се дължи освен всичко и на факта, че неговото обобщение за отсичане в пространството е елементарно.

АЛГОРИТЪМ НА ЛЯН-БАРСКИ ЗА ОТСЕЧКА. Алгоритъмът на Лян-Барски (Liang-Barsky) е разработен значително по-късно (1984 год.) и има за основа по-общия алгоритъм на Сайръс-Бек (Cyrus-Beck), който ще разгледаме при отсичането спрямо изпъкнал многоъгълник в 3.2.1. Той също може да бъде разширен, за да решава и съответната тримерна задача, но освен това е приложим и за многоъгълници.

В основата на този алгоритъм стои параметричното представяне на отсечката, зададена с крайните си точки $(x_1,y_1),(x_2,y_2)$:

$$\begin{aligned} x &= x_1 + dx\cdot t \\ y &= y_1 + dy\cdot t \end{aligned} \quad \text{където} \quad dx = x_2 - x_1; \qquad dy = y_2 - y_1; \qquad 0 \le t \le 1. \qquad [3.2]$$

Намирането на тази част от нея, която попада в правоъгълника, се свежда до намирането на двете стойности t_{in}, t_{out} на параметъра t, които съответстват на т.нар. входна и изходна точки. Това са двете точки $\mathbf{P}_{in} = \mathbf{P}(t_{in})$ и $\mathbf{P}_{out} = \mathbf{P}(t_{out})$, в които отсечката влиза и напуска отсичащия правоъгълник.

Да видим как можем най-ефективно да намерим търсените стойности t_{in}, t_{out} на параметъра. Точките от тази част на отсечката, които попадат във вътрешността на правоъгълника, удовлетворяват следните неравенства за параметъра:

$$\begin{aligned} x_{\min} &\le x_1 + dx \cdot t \le x_{\max} \\ y_{\min} &\le y_1 + dy \cdot t \le y_{\max} \end{aligned} \qquad [3.3]$$

което може да бъде записано и по следния начин:

$$\begin{aligned} -dx \cdot t &\le x_{\max} - x_1 \quad dx \cdot t \ge x_1 - x_{\min} \\ -dy \cdot t &\le y_{\max} - y_1 \quad dy \cdot t \ge y_1 - y_{\min} \end{aligned}$$

Още един еквивалентен запис на тези четири неравенства е:

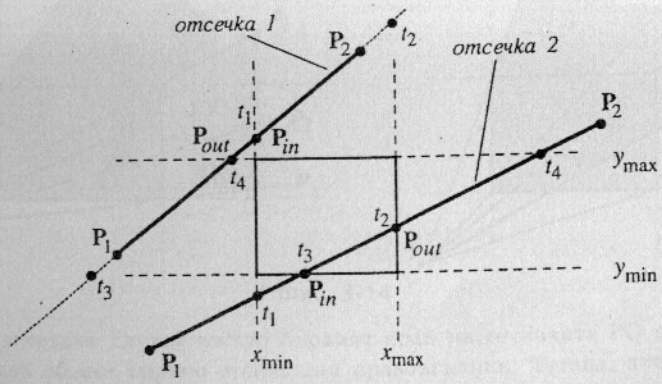
$$R_i \cdot t \le Q_i, \qquad i = 1,2,3,4 \qquad [3.4]$$

където:

$$\begin{aligned} R_1 &= -dx \quad Q_1 = x_1 - x_{\min} \\ R_2 &= dx \quad Q_2 = x_{\max} - x_1 \\ R_3 &= -dy \quad Q_3 = y_1 - y_{\min} \\ R_4 &= dy \quad Q_4 = y_{\max} - y_1 \end{aligned}$$

Нека приемем за момент, че $R_i \neq 0$, $i = 1,2,3,4$. Тогава можем да кажем, че системата неравенства [3.4] ще бъде удовлетворена за стойностите на t от интервала:

$$\max\left(\left\{t_i \middle| R_i < 0\right\}\right) \le t \le \min\left(\left\{t_i \middle| R_i > 0\right\}\right) \text{ където } t_i = \frac{Q_i}{R_i}$$



Фиг. 3-13

Вижда се, че при стойностите t_i на параметъра се получават пресечните точки на правата, върху която лежи отсечката с четирите прави, носещи страните на правоъгълника, както е показано на фиг. 3-13. За да си осигурим пресечните точки на тази права да са пресечни точки и на самата отсечка, t трябва да е в интервала [0,1]. Това позволява да запишем горния интервал окончателно като:

$$\max\left(\left\{t_i \middle| R_i < 0\right\}, 0\right) \le t \le \min\left(\left\{t_i \middle| R_i > 0\right\}, 1\right).$$

Намирайки интервала, в който неравенствата [3.4] са удовлетворени, определихме и търсените неизвестни t_{in}, t_{out}:

$$t_{in} = \max\left(\left\{t_i \middle| R_i < 0\right\}, 0\right), \qquad t_{out} = \min\left(\left\{t_i \middle| R_i > 0\right\}, 1\right)$$

По-горе приехме, че $R_i \neq 0$, $i = 1,2,3,4$. Ако сега вземем пред вид възможността някое $R_i = 0$, това ще означава, че съответното неравенство е изпълнено или за всяко или за никое t. То няма да бъде изпълнено само ако $Q_i < 0$, което означава, че отсечката е или хоризонтална, или вертикална и се намира във външната полуравнина, определена от някоя от ограничителните прави на правоъгълника. Сега вече можем да обобщим, че системата от неравенства [3.4] няма да бъде удовлетворена за никое t само в следните два случая:

1. Ако за някое i, $R_i = 0$ и $Q_i < 0$ т.е. i-тото неравенство в [3.4] никога не е изпълнено;
2. Ако $t_{in} > t_{out}$. Това съответства на положението на *отсечка 1* спрямо правоъгълника (вж. фиг. 3-12).

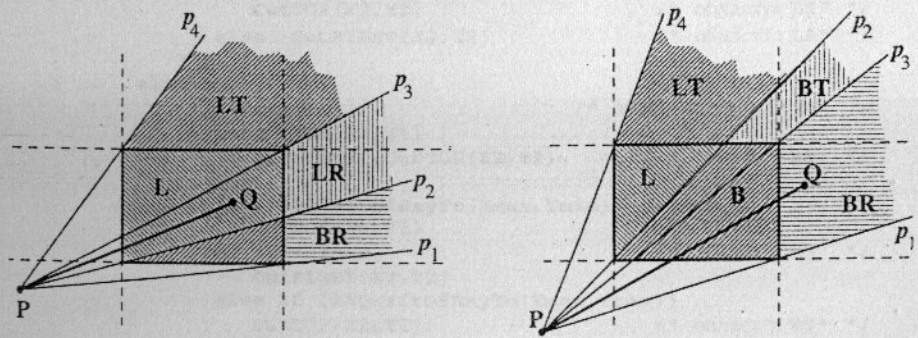
```
int LiangBarsky2DClip(x1,y1,x2,y2,X1,Y1,X2,Y2)
float x1,y1,x2,y2
float *X1,*Y1,*X2,*Y2;
{float dx,dy,tin,tout;
   dx=x2-x1;
   dy=y2-y1;
   tin=0;
   tout=1;
   if (CalcT(-dx,Xmax-x1,tin,tout))
      if (CalcT(dx,x1-Xmin,tin,tout))
         if (CalcT(-dy,Ymax-y1,tin,tout))
            if (CalcT(dy,y1-Ymin,tin,tout)) {
               if (tin>0) {
                  *X1=x1+tin*dx;
                  *Y1=y1+tin*dy;
               } else {
                  *X1=x1; *Y1=y1;
               }
               if (tout<1) {
                  *X2=x1+tout*dx;
                  *Y2=y1+tout*dy;
               } else {
                  *X1=x1; *Y1=y1;
               }
               return 1;
            }
}
```

При реализацията на алгоритъма използваме помощната функция CalcT за определяне стойността на t_i, която освен това връща 0 ако се установи наличието на някой от горните два случая. Това означава, че трябва да се отхвърли цялата отсечка. Естествено е да прекъснем изчисляването на останалите t_i ако резултатът при някое от обръщенията към функцията е 0.

```
int CalcT(R,Q,tin,tout)
float R,Q,tin,tout;
{float t;
    if (R>0) {
       t=Q/R; if (t<tin) return 0;    /* новото tout<tin */
       tout=MIN(t,tout);
    } else if (R<0) {
       t=Q/R; if (t>tout) return 0;   /* новото tin>tout */
       tin=MAX(t,tin);
    } else if (Q<0) return 0;         /* R=0 и Q<0 */
    return 1;
}
```

АЛГОРИТЪМ НА НИКОЛ-ЛИЙ-НИКОЛ. Ще разгледаме още един алгоритъм, които не притежава общността на горните два (не може аналогично да се приложи в тримерния случай), но който е най-ефективен от разработените до момента методи. Той е предложен от Никол, Лий и Никол (Nicholl, Lee, Nicholl) през 1987 год. и включва 3 пъти по-малко сравнения от този на Коен-Садърланд и 2 пъти по-малко от този на Лян-Барски. Освен това в него се използват точно толкова операции делене, колкото е броят на пресечните точки.

Общата идея е да се раздели равнината на повече от 9-те области, разгледани в алгоритъма на Коен-Садърланд в зависимост от това как двете крайни точки са разположени една спрямо друга. Това налага анализирането на всеки случай на взаимно разположение поотделно, въпреки че обработката на всички е аналогична.



Фиг. 3-14

Да разгледаме случая когато първият край на отсечката **PQ** се намира в долната лява област спрямо отсичащия правоъгълник. Тогава, ако Q е вляво от правоъгълника $(x_2 < X_{\min})$ или е под него $(y_2 < Y_{\min})$, то отсечката е невидима. Това съответства на теста за отхвърляне в алгоритъма на Коен-Садърланд. Ако никое от горните две неравенства не е изпълнено, то отсечката ще пресича долната страна на правоъгълника само ако Q лежи в областта между лъчите p_1 и p_2, а лявата страна - само ако Q лежи в областта, заключена между лъчите p_2 и p_4, както е показано на фиг. 3-14.

Тези два случая могат да се разграничат в зависимост от това от коя страна на лъча p_2 се намира точката Q. На същата фигура равнината е разделена на няколко области. Това са областите, в които може да се намира вторият край на отсечката. Можем да кажем, че ако вторият край на отсечката попада в някоя от изброените области, то ние знаем и кои страни на правоъгълника тя пресича. Означението на областите има следния смисъл:

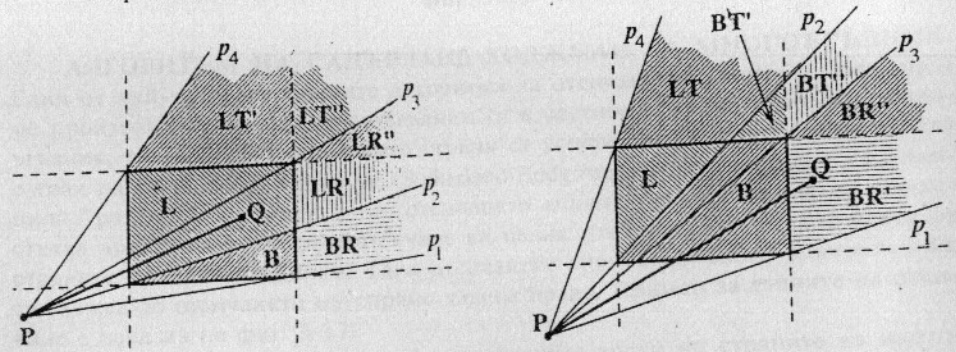
L - отсечката пресича лявата страна;
LR - отсечката пресича лявата и дясната страни;
LT - отсечката пресича лявата и горната страни;
B - отсечката пресича долната страна;
BR - отсечката пресича долната и дясната страни;
BT - отсечката пресича долната и горната страни;

Ако точката **Q** е отляво на p_2, тя може да попада в някоя от следните области:

- ако $x_2 \le X_{\max}$ и $y_2 \le Y_{\max}$, тогава $\mathbf{Q} \in \mathbf{L}$;
- ако горното не е вярно и Q е вдясно от p_3, тогава $\mathbf{Q} \in \mathbf{LR}$;
- ако горните две не са изпълнени и Q е вдясно от p_4, тогава $\mathbf{Q} \in \mathbf{LT}$;
- ако нито едно от горните условия не е изпълнено, отсечката не пресича правоъгълника.

Аналогични са възможностите и когато **Q** е отдясно на p_2:

- ако $x_2 \le X_{\max}$ и $y_2 \le Y_{\max}$, тогава $\mathbf{Q} \in \mathbf{B}$;
- ако горното не е вярно и Q е вляво от p_3, тогава $\mathbf{Q} \in \mathbf{BT}$;
- ако горните две не са изпълнени и Q е вляво от p_1, тогава $\mathbf{Q} \in \mathbf{BR}$;
- в противен случай отсечката няма видима част.



Фиг. 3-15

Допълнително ускоряване на работата на алгоритъма може да се постигне, ако областите LT, LR, BT и BR се разбият на по две части - съответно: LT', LT'', LR', LR'', BT', BT'', BR' и BR'', както това е показано на фиг. 3-15. Ползата от това разделяне е, че проверката за принадлежност към LT', LR', BT' или BR', е значително по-проста. В този случай се анализира само едната координата, а не положението спрямо лъча.

Как се извършва скъсяването вече видяхме в алгоритъма на Коен-Садърланд. За да установим дали точката Q е вляво от един лъч PR е необходимо да проверим дали синуса на ъгъла между лъча и вектора PQ има положителен знак. Това е еквивалентно на условието знакът на векторното произведение на PQ и PR да е положителен:

$$(y_Q - y_P)(x_R - x_P) - (x_Q - x_P)(y_R - y_P) > 0$$

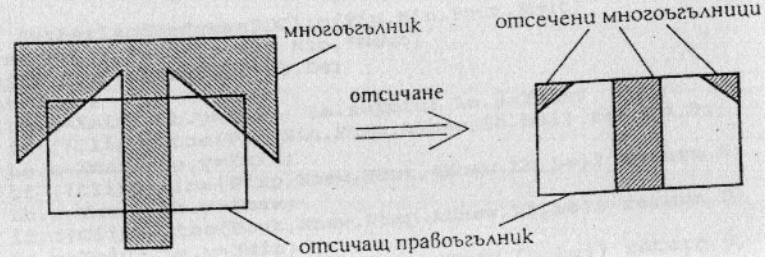
```
#define isQLeftOfRayTo(x,y)  dy*(x-x1)>dx*(y-y1)

int NichollLeeNicholl2DClip(x1,y1,x2,y2,X1,Y1,X2,Y2)
float x1,y1,x2,y2, *X1,*Y1,*X2,*Y2;
{float dx,dy;
   if (x1<Xmin && y1<Ymin) {
      if (x2<Xmin) return 0;
      if (y2<Ymin) return 0;
      dx=x2-x1; dy=y2-y1;
      *X1=x1; *Y1=y1; *X2=x2; *Y2=y2;
      if (isQLeftOfRayTo(Xmin,Ymin))
         if (y2<=Ymax) {                    /* областите L или LR' */
            CutLEFT(X1,Y1)
            if (x2>Xmax) CutRIGHT(X2,Y2)       /* областта LR' */
         } else {
            if (isQLeftOfRayTo(Xmin,Ymax)) return 0;
            CutLEFT(X1,Y1)
            if (x2<=Xmax)                      /* областта LT' */
               CutTOP(X2,Y2)
            else if (isQLeftOfRayTo(Xmax,Ymax))
               CutTOP(X2,Y2)                   /* областта LT" */
            else  CutRIGHT(X2,Y2)              /* областта LR" */
         }
      else
         if (x2<=Xmax) {                   /* областите B или BT' */
            CutBOTTOM(X1,Y1)
            if (y2>Ymax) CutTOP(X2,Y2)         /* областта BT' */
         } else {
            if (!isQLeftOfRayTo(Xmax,Ymin)) return 0;
            CutBOTTOM(X1,Y1)
            if (y2<=Ymax)                      /* областта BR' */
               CutRIGHT(X2,Y2)
            else if (isQLeftOfRayTo(Xmax,Ymax))
               CutTOP(X2,Y2)                   /* областта BT" */
            else  CutRIGHT(X2,Y2)              /* областта BR" */
         }
   } else . . . обработка на останалите случаи за разположение на P
   return 1;
}
```


В програмата е показана обработката само на случая, който анализирахме. Останалите възможности трябва да се разгледат поотделно, което прави функцията, реализираща алгоритъма значително по-дълга от тези на разгледаните два метода, но по-бърза от тях. За по-лесно четене скъсяванията на отсечката спрямо всяка от страните на отсичащия многоъгълник са записани като отделни макроси.

```
#define CutLEFT(X,Y)   { *X=Xmin; *Y=y1+(Xmin-x1)*dy/dx; }
#define CutBOTTOM(X,Y) { *Y=Ymin; *X=x1+(Ymin-y1)*dx/dy; }
#define CutRIGHT(X,Y)  { *X=Xmax; *Y=y1+(Xmax-x1)*dy/dx; }
#define CutTOP(X,Y)    { *Y=Ymax; *X=x1+(Ymax-y1)*dx/dy; }
```

3.2.2 Отсичане на многоъгълници от правоъгълник

Отсичането на многоъгълници от правоъгълник е по-сложна задача от тази, която разгледахме дотук. Основната разлика е, че многоъгълниците са истински двумерни фигури и след отсичането им те трябва да останат такива. Това означава, че е необходимо като резултат от отсичането да се получи отново многоъгълник, а не просто последователност от несвързани отсечки, т.е. задачата не може тривиално да се сведе до отсичане на страните му като отделни отсечки. Това е особено важно, когато полученият многоъгълник трябва да бъде запълнен. Някои разположения на многоъгълника спрямо отсичащия правоъгълник може да са такива, че отсичането да води до получаване не на един, а на няколко многоъгълника (фиг. 3-16).



Фиг. 3-16

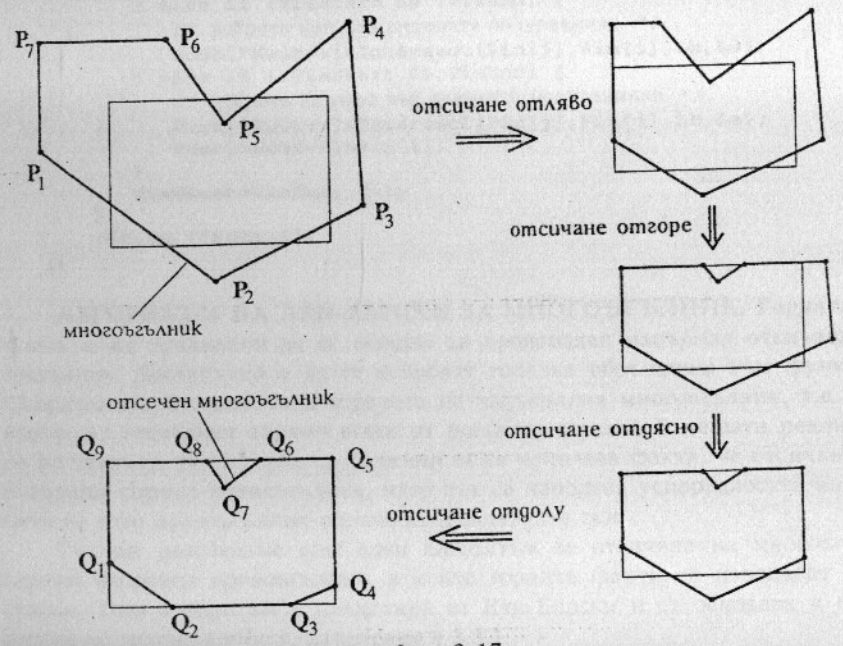
АЛГОРИТЪМ НА САДЪРЛАНД-ХОДЖМАН ЗА МНОГОЪГЪЛНИК.

Един от най-разпространените алгоритми за отсичане на многоъгълник спрямо произволен изпъкнал многоъгълник (и в частност спрямо изправен правоъгълник, т.е. правоъгълник чиито страни са успоредни на координатните оси) е този на Садърланд-Ходжман (Sutherland-Hodgman). Той е построен на принципа "разделяй и владей", т.е. отсичането спрямо правоъгълника се осъществява чрез последователно отсичане на целия многоъгълник спрямо всяка от страните на правоъгълника. Така отсичането спрямо изпъкнал многоъгълник се свежда до отсичането му спрямо дадена права. Пример за етапите на отсичане е показан на фиг. 3-17.

От примера се вижда, че освен пресечните точки на страните на многоъгълника с отсичащия правоъгълник резултатът съдържа като върхове и ня-

кои от върховете на правоъгълника - такива са Q_5 и Q_9. Получаването на тези върхове става възможно именно при последователното отсичане. Редът, в който то се прави, не е от значение.

Нека първо да разгледаме как се извършва отсичането на многоъгълник спрямо права. Всяка права разделя равнината на две полуравнини, едната от които условно можем да наречем *видима*, а другата *невидима*. *Видима* е тази полуравнина, която съдържа отсичащия правоъгълник. Очевидно е, че такова разделяне може да се направи не само спрямо страните на правоъгълник, но спрямо тези на произволен изпъкнал многоъгълник. Всеки връх P_i от многоъгълника, който предстои да бъде отсечен, може да се нарече видим или невидим спрямо правата в зависимост от това в коя полуравнина лежи. Върховете, лежащи точно върху правата, ще считаме за видими.

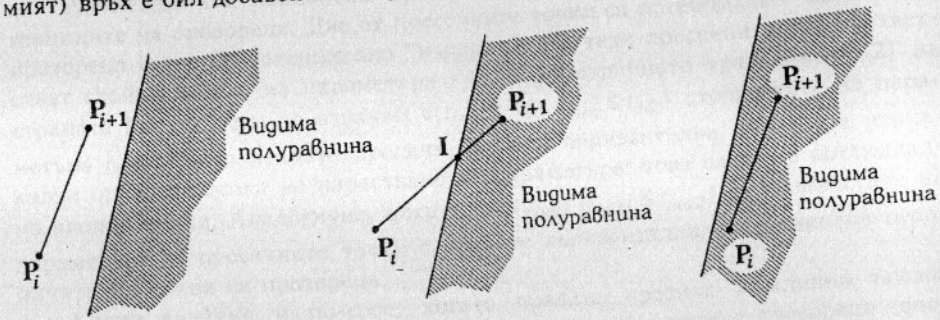


Фиг. 3-17

Ще обходим списъка от върхове, с който се задава многоъгълникът, като на всяка стъпка ще разглеждаме по една двойка точки. Те са краища на някое ребро на многоъгълника. Всяко ребро може да бъде разположено по следните няколко начина спрямо отсичащата права в зависимост от видимостта на краищата му:

- *и двете точки са видими*: реброто е напълно видимо и следователно и двете му точки ще участват в изходния многоъгълник;
- *и двете точки са невидими*: реброто е напълно невидимо и нито една от двете му точки няма да е връх на изходния многоъгълник;
- *едната точка е видима, а другата е невидима*: реброто пресича правата и видимата точка заедно с пресечната точка I са върховете на изходния многоъгълник.

Ще образуваме нов списък от върхове, към който ще добавяме всеки видим връх в последователността, в която те описват зададения многоъгълник. Освен тях ще включваме и всяка пресечна точка на ребро, което навлиза във видимата полуравнина, т.е. такова ребро, на което първата точка е невидима, а втората видима, като първо записваме пресечната точка, а след това видимата. Когато едно ребро излиза от видимата полуравнина, към изходния многоъгълник ще се добавя само пресечната му точка, тъй като началният (видимият) връх е бил добавен като краен връх на предното ребро.



Фиг. 3-18

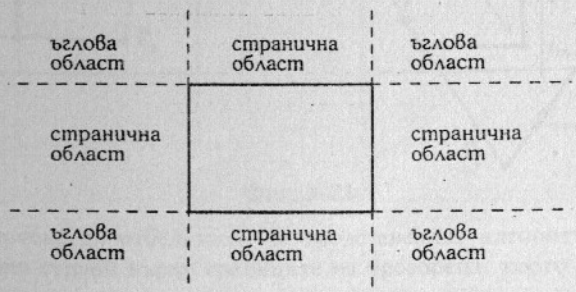
```
int SutherlandHodgman2DClip(Pin,Nin,Pout,Nout)
Point Pin[],Pout[]; int Nin,*Nout;
{Point Ptmp[MAXPNTS], Lb,Le;
 int Nnew;
   Lb.x=Xmin; Lb.y=Ymin; Le.x=Xmin; Le.y=Ymax;
   if (!ClipByLine(Pin,Nin,Ptmp,&Nnew,Lb,Le)) return 0;
   Le.x=Xmax; Le.y=Ymin;
   if (!ClipByLine(Ptmp,Nnew,Pout,&Nnew,Lb,Le)) return 0;
   Lb.x=Xmax; Lb.y=Ymax;
   if (!ClipByLine(Pout,Nnew,Ptmp,&Nnew,Lb,Le)) return 0;
   Le.x=Xmin; Le.y=Ymin;
   if (!ClipByLine(Ptmp,Nnew,Pout,&Nnew,Lb,Le)) return 0;
   return 1;
}
```

Използваната функция `ClipByLine` извършва отсичането на многоъгълника `Pin[Nin]` спрямо правата, определена от точките `Lb,Le` и връща резултата в многоъгълника `Pout[Nout]`. Тя има стойност 1, ако многоъгълникът е видим (т.е. лежи изцяло или частично във видимата полуравнина) и 0 ако е изцяло извън нея. Ако на някой от етапите на отсичане при последователното обръщение към тази функция тя върне 0, трябва да прекратим отсичането. Видимостта на една точка определяме с макроса `isVisible` като разчитаме, че върховете на правоъгълника са зададени в посока, обратна на часовниковата стрелка. Тогава видимата полуравнина е отляво и можем да използваме свойството на векторното произведение както в предишния алгоритъм. Забележете, че видимостта може да бъде определена и по-просто, ако вземем предвид, че страните на правоъгълника са успоредни на осите.

```
#define isVisible(P) (P.y-Lb.y)*dx > (P.x-Lb.x)*dy

int ClipByLine(Pin,Nin,Pout,Nout,Lb,Le)
Point Pin[],Pout[],Lb,Le;
int Nin,*Nout;
{int i,j,VisStart,VisEnd;
 float dx,dy;  /* векторът на отсичащото ребро */
   dx=Le.x-Lb.x; dy=Le.y-Lb.y;
   *Nout=0;
   /* определяне на видимостта на началната точка */
   j=Nin-1; VisStart=isVisible(Pin[j]);
   for (i=0; i<Nin; i++) {
      VisEnd=isVisible(Pin[i]);
      if (VisStart && VisEnd) {
         /* и двата края на реброто са видими */
         Pout[*Nout++]=Pin[i];
      } else if (VisStart && !VisEnd) {
         /* реброто напуска видимата полуравнина */
         Pout[*Nout++]=Intersect(Pin[j],Pin[i],Lb,Le);
      } else if (!VisStart && VisEnd) {
         /* реброто навлиза във видимата полуравнина */
         Pout[*Nout++]=Intersect(Pin[j],Pin[i],Lb,Le);
         Pout[*Nout++]=Pin[i];
      }
      VisStart=VisEnd; j=i;
   }
   return (*Nout>0);
}
```

АЛГОРИТЪМ НА ЛЯН-БАРСКИ ЗА МНОГОЪГЪЛНИК. Горната програма може тривиално да се обобщи за произволен изпъкнал отсичащ многоъгълник. Достатъчно е да се направят толкова обръщения към функцията `ClipByLine`, колкото са и страните на изпъкналия многоъгълник, т.е. да се извършат отсичания спрямо всяка от неговите страни. В нашата реализация на алгоритъма на Садърланд-Ходжман не се използва факта, че отсичането се извършва спрямо правоъгълник, нито пък се използва успоредността на страните на този правоъгълник спрямо координатните оси.

Тук ще разгледаме още един алгоритъм за отсичане на многоъгълник спрямо изправен правоъгълник, в който горните факти се използват съществено. Този алгоритъм е предложен от Лян-Барски и се основава в голяма степен на разсъжденията, направени в 3.3.1.



Фиг. 3-19

Ако разделим равнината на 9 части спрямо границите на отсичащия правоъгълник, то тези части, които са извън правоъгълника можем да класифицираме или като *странични области* (които граничат със страна на прозореца) или *ъглови области* (обхващащи външните ъгли на правоъгълника).

Нека разгледаме една от страните $\mathbf{P}_i\mathbf{P}_{i+1}$ на многоъгълника и нека за момент да се условим, че тя не е нито вертикална, нито хоризонтална. При това условие правата, върху която тази страна лежи, започва от ъглова област и завършва в ъглова област. За нея има две възможности: тя или пресича прозореца; или пресича още една ъглова област.

Разглежданата наклонена права пресича и четирите прави, задаващи границите на прозореца. Две от пресечните точки са потенциално "входни" за прозореца и две са потенциално "изходни". На тези пресечни точки съответстват стойностите t_i на параметъра t от параметричното уравнение [3.2] на страната $\mathbf{P}_i\mathbf{P}_{i+1}$. Нека да означим с t_{in1} и t_{in2} $(t_{in1} \le t_{in2})$ стойностите на параметъра t, за които правата пресича първата хоризонтална и първата вертикална прави в посока на нарастване на параметъра: това са двете потенциално входни точки. Аналогично, нека t_{out1} и t_{out2} $(t_{out1} \le t_{out2})$ са стойностите на параметъра за пресечните точки с вторите хоризонтална и вертикална ограничителни прави на прозореца.

Както видяхме и по-горе, когато правата пресича междинна ъглова област (фиг. 3-20а) е изпълнено $t_{out1} < t_{in2}$, а когато пресича прозореца (фиг. 3-20б): $t_{in2} \le t_{out1}$. Като вземем предвид, че двата края на разглежданата страна се получават за стойностите на параметъра: $t = 0$ и $t = 1$, можем да кажем, че тя ще има видима част само ако:

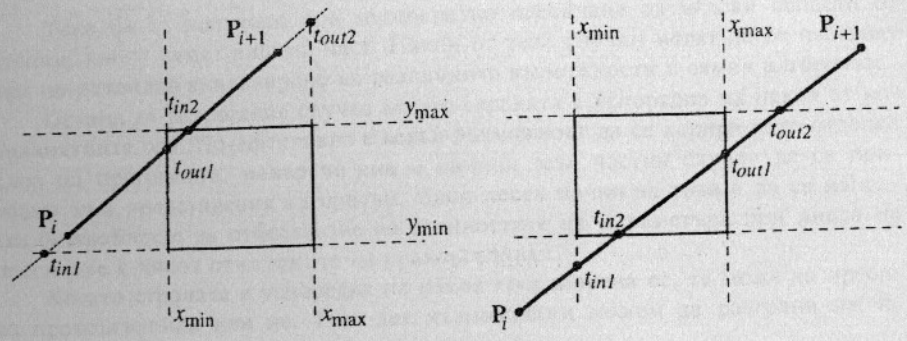
1. $t_{in2} \le t_{out1}$ (правата ѝ пресича прозореца) и
2. двата интервала $[t_{in2},t_{out1}]$ и $[0,1]$ имат обща част. Това може да се запише като условието:

$$(0 < t_{out1}) \wedge (t_{in2} \le 1),$$

което означава, че началото на страната е преди носещата права да напусне прозореца и краят ѝ е след като съответната ѝ права навлезе в прозореца.

Изходният многоъгълник (този, който се получава като резултат от отсичането) ще се образува, като на всяка стъпка (за всяка страна) към него добавяме един или повече върхове. Кои върхове да добавяме ще зависи от взаимното разположение на t_{in1}, t_{in2}, t_{out1} и t_{out2}; 0 и 1 върху числовата ос. Ако разглежданата страна има видима част, обработката на списъка ще се извърши по следния начин:

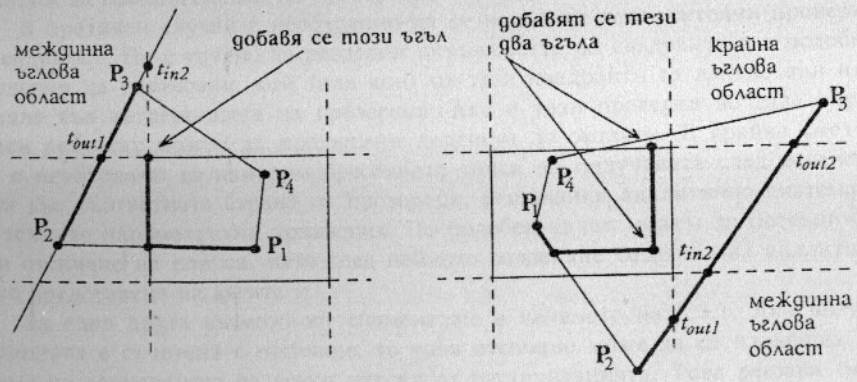
- Ако началото на страната е преди t_{in2} (т.е. $0 < t_{in2}$), то трябва в списъка да се добави пресечната точка за параметъра t_{in2}. В противен случай началото ѝ е вътрешно за прозореца и е било добавено на предишната итерация (като крайна точка на предната страна).
- Ако краят на страната е след t_{out1} (т.е. $t_{out1} < 1$), трябва да се добави пресечната точка в t_{out1}, а иначе да се добави самият край на страната.



Фиг. 3-20 а,б

Освен това има една особеност, която заслужава специално внимание: Независимо дали разглежданата страна има видима част, ако тя преминава през *ъглова област*, то съответният ъгъл на прозореца трябва да се включи в списъка на изходния многоъгълник. Алгоритмично това включване може да се осъществи или когато страната навлезе в такава област, или когато я напуска (но не и в двата случая). В представения тук вариант ще включваме ъгъла всеки път, когато страната навлезе в такава област.

Всяка страна може да навлезе в *крайна* и/или *междинна* ъглова област. *Крайна ъглова област* е ъглова област, която съдържа втория край на страната, а *междинна* е тази, през която страната преминава, когато не пресича правоъгълника (фиг. 3-20а). При положение, че навлиза и в двете, към списъка на върхове за изходния многоъгълник е необходимо да се добавят и двата ъгъла. На фиг. 3-21 са илюстрирани тези две възможности за страната P_2P_3. С по-дебела линия е показана частта от изходния многоъгълник, създадена до момента на разглеждане на тази страна.



Фиг. 3-21

Накрая трябва да отбележим, че представеният алгоритъм може да генерира излишни страни върху границите на прозореца, което не би било особена пречка при запълване, изчисляване на площи и др.


```
int LiangBarskyPolygonClip(Pin,Nin,Pout,Nout)
Point Pin[],Pout[]; int Nin,*Nout;
(float dx,dy, xIn,xOut, yIn,yOut;
 float t,tIn2,tInX,tInY,tOut1,tOut2;
 int Xfirst, i, j;
   (*Nout)=0;
   for (i=0; i<Nin; i++) { j=(i+1)%Nin;
      dx=Pin[j].x-Pin[i].x; dy=Pin[j].y-Pin[i].y;
      /* намиране на пресечните точки с правите на прозореца */
      if (dx>0 || (dx==0 && Pin[i].x>Xmax)) {
                xIn=Xmin; xOut=Xmax;
      } else { xIn=Xmax; xOut=Xmin; }
      if (dy>0 || (dy==0 && Pin[i].y>Ymax)) {
                yIn=Ymin; yOut=Ymax;
      } else { yIn=Ymax; yOut=Ymin; }
      /* намиране стойностите на t при изход от прозореца */
      if (dx!=0) tOut1 = (xOut-Pin[i].x)/dx;
      else if (Xmin<=Pin[i].x && Pin[i].x<=Xmax)
         tOut1 = INFINITE;
      else tOut1 = -INFINITE;
      if (dy!=0) tOut2 = (yOut-Pin[i].y)/dy;
      else if (Ymin<=Pin[i].y && Pin[i].y<=Ymax)
         tOut2 = INFINITE;
      else tOut2 = -INFINITE;
      if (tOut2<tOut1){ t=tOut1; tOut1=tOut2; tOut2=t; }
      /* анализиране на разположението на входните и изходни t */
      if (tOut2>0) {                    /* възможно е да се добави връх */
         /* намиране стойностите на t при вход в прозореца */
         tInX=dx?((xIn-Pin[i].x)/dx):-INFINITE;
         tInY=dy?((yIn-Pin[i].y)/dy):-INFINITE;
         Xfirst=tInX<tInY;
         tIn2=MAX(tInX,tInY);
         if (tOut1<tIn2) {              /* няма видима част */
            if (0<tOut1 && tOut1<=1){   /* ъглова област */
               Pout[*Nout].x=Xfirst?xOut:xIn;
               Pout[*Nout].y=Xfirst?yIn:yOut; (*Nout)++;
            }
         } else if (0<tOut1 && tIn2<=1) { /* пресича прозореца */
               if (0<tIn) {             /* пресичане при вход */
                  Pout[*Nout].x=Xfirst?(Pin[i].x+tInY*dx):xIn;
                  Pout[*Nout].y=Xfirst?yIn:(Pin[i].y+tInX*dy);
                  (*Nout)++;
               }
               if (tOut1<1) {           /* пресичане при изход */
                  Pout[*Nout].x=Xfirst?(Pin[i].x+tInY*dx):xIn;
                  Pout[*Nout].y=Xfirst?yIn:(Pin[i].y+tInX*dy);
                  (*Nout)++;
               } else                   /* крайната точка е в прозореца */
                  Pout[(*Nout)++]=Pin[i+1];
         }
         if (0<tOut2 && tOut2<=1) {
            Pout[*Nout].x=xOut;   /* завършва в ъглова област */
            Pout[*Nout].y=yOut; (*Nout)++;
         }
      }
   }
   return (*Nout>0);
}
```

Това би се получило при многократно пресичане на ъглови области от страни, които нямат видима част. Някои от тези случаи могат да се избегнат при по-детайлно анализиране на различните възможности в самия алгоритъм.

Остана да разгледаме случая когато страната е успоредна на някоя от координатните оси. По-ефективно е всяка възможност да се кодира като отделен клон на програмата, макар че има и начини тези частни случаи да се приобщят към представения алгоритъм. Един лесен начин за това е да се използва *безкрайност* за отбелязване на стойностите на параметъра при липса на пресичане с някоя от страните на правоъгълника.

Когато страната е успоредна на някоя координатна ос, тя може да пресича правоъгълника или не. Тези две възможности можем да разграничим по следния начин:

- Ако страната пресича правоъгълника, за изходна стойност на параметъра ще вземем $t_{out2} = \infty$.
- Ако тя не пресича правоъгълника, ще установим $t_{out1} = -\infty$.

Входната стойност на параметъра и в двата случая ще установим като $t_{in1} = -\infty$. По този начин подредбата на стойностите на параметъра в първия случай ще бъде правилна: $t_{in1} < t_{out2}$, докато във втория страната ще бъде класифицирана като невидима, тъй като няма да бъде изпълнено $t_{in2} \le t_{out1}$.

3.2.3 Отсичане на окръжности, елипси, дъги и символи.

Една от често решаваните задачи при визуализация на равнинни обекти е отсичане на окръжност и елипса от правоъгълник. Като първа стъпка в намирането на видимата част на една окръжност е необходимо да се анализира разположението на минималния квадрат, обхващащ тази окръжност спрямо отсичащия правоъгълник. Това е квадрат със страна, чиято големина е равна на големината на диаметъра ѝ. Ако този квадрат е изцяло видим или изцяло невидим за правоъгълника на прозореца, то същото важи и за окръжността.

В противен случай е необходимо да се направят допълнителни проверки за видимост. Не е трудно да разделим окръжността на квадранти и с подобни проверки да установим, кой (или кои) от тези квадранти са изцяло вън или изцяло във вътрешността на прозореца. Ако и тази проверка не даде определен резултат можем да продължим деленето до октанти. В крайна сметка ще е необходимо да намерим пресечната точка на получената след деленето дъга със съответната страна от прозореца, решавайки аналитично системата от техните параметрични уравнения. По подобен начин можем да постъпим и при отсичане на елипса, като след нейното разделяне се използва аналитичното представяне на дъгите ѝ.

За една друга възможност споменахме в началото на 3.3.1. Ако визуализацията е съчетана с отсичане, то това отсичане може да се извършва по време на записването на всеки пиксел от растеризацията. Това решава (макар и доста неикономично) проблема за всички графични примитиви, които се растеризират - включително и за символи.

Символите също се отсичат като първо се проверява взаимното разположение на правоъгълната им обвивка и прозореца. Най-тривиалното отсичане

на символ е да се изобразяват само тези символи, чиито правоъгълни обвивки са изцяло видими, а всички останали да се считат за невидими. Това естествено не би позволило да се изобразяват видимите части на символите, попадащи на границата на прозореца.

Такова решение може да бъде взето единствено в графични системи, в които текстовата информация се използва само за анотация. Отсичането на частично видими символи зависи от начина, по който те са дефинирани:

- символи, дефинирани чрез растерен образец, се отсичат най-често чрез проверка на видимостта на всеки пиксел от образеца.
- векторните символи (които са съставени от последователност от отсечки) се отсичат, като се отсичат векторите, които ги съставят.
- символите, представляващи запълнен сложен контур (сплайн или многоъгълник) се отсичат, като се отсича самият контур.

3.2.4 Отсичане спрямо многоъгълници

В тази част ще разгледаме отсичането спрямо многоъгълници, което не е толкова често решавана задача при визуализацията на равнинни обекти както отсичането спрямо изправен правоъгълник. Затова тук ще се спрем само на отсичането на отсечка. Както видяхме в 3.3.1 някои алгоритми за отсичане спрямо правоъгълници естествено могат да бъдат обобщени и спрямо произволни изпъкнали многоъгълници.

АЛГОРИТЪМ НА САЙРЪС-БЕК ЗА ИЗПЪКНАЛИ МНОГОЪГЪЛНИЦИ. Алгоритъмът, предложен от Сайръс-Бек (Cyrus-Beck) през 1978 год. за намиране на видимата част на отсечка спрямо изпъкнал многоъгълник е в основата на разработения по-късно вариант от Лян и Барски специално за изправен правоъгълник. Ще отбележим, че в преводната руска литература този алгоритъм е наречен "алгоритъм на Кирус-Бек". Основното, което се използва в него, както и в разгледания по-горе вариант на Лян-Барски е, че една отсечка, пресичаща такъв многоъгълник, има винаги най-много една входна и една изходна точки. Тези точки отговарят на стойностите t_{in} и t_{out} на t от параметричното уравнение на правата, върху която лежи отсечката. Взаимното разположение на стойностите t_{in}, t_{out}, 0 и 1 (отбелязващи краищата на отсечката) определя еднозначно видимата ѝ част.

Основното в този алгоритъм е начинът, по който се определят t_{in} и t_{out}. Нека $\mathbf{Q}_1\mathbf{Q}_2$ е отсечката, чието параметрично уравнение е:

$$\mathbf{Q}(t) = \mathbf{Q}_1 + t \cdot (\mathbf{Q}_2 - \mathbf{Q}_1) \qquad 0 \le t \le 1 \qquad [3.5]$$

Нека разгледаме произволна страна $\mathbf{P}_i\mathbf{P}_{i+1}$ от многоъгълника. Нейната вътрешна нормала $\mathbf{n}_i$ е векторът, който е перпендикулярен на тази страна и има посока към вътрешността на многоъгълника. Нека $\mathbf{Q}(t)$ е произволна точка от правата, върху която лежи $\mathbf{Q}_1\mathbf{Q}_2$, а $\mathbf{T}$ е вектор с начало първата точка на страната и край в $\mathbf{Q}(t)$:

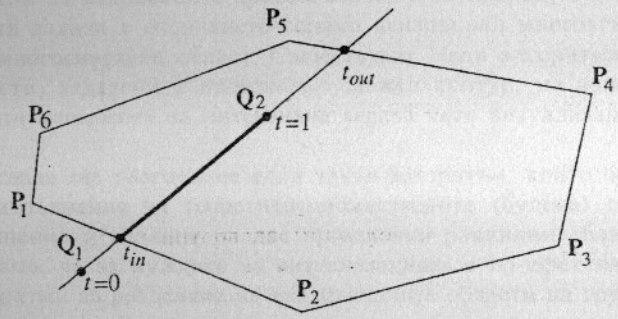
$$\mathbf{T} = \mathbf{Q}(t) - \mathbf{P}_i$$

Можем да твърдим, че за всяка точка $\mathbf{Q}(t)$ от правата знакът на скаларното произведение

$$s = \mathbf{T}\cdot\mathbf{n}_i = \left[\mathbf{Q}(t)-\mathbf{P}_i\right]\cdot\mathbf{n}_i$$

има следния смисъл:

- ако $s>0$: точката $\mathbf{Q}(t)$ лежи от вътрешната страна на $\mathbf{P}_i\mathbf{P}_{i+1}$;
- ако $s<0$: точката $\mathbf{Q}(t)$ лежи от външната страна на $\mathbf{P}_i\mathbf{P}_{i+1}$;
- ако $s=0$: точката $\mathbf{Q}(t)$ лежи върху страната $\mathbf{P}_i\mathbf{P}_{i+1}$.



Фиг. 3-22

И наистина, този знак се определя от знака на косинуса на ъгъла между двата вектора, който е положителен за ъгли от 0 до $\pi/2$ и отрицателен за тези между $\pi/2$ и π (фиг. 3-23).

Като приложим този резултат за всяка от страните на многоъгълника, подобно на [3.4] можем да запишем, че точките от видимата част на правата, върху която отсечката лежи, ще удовлетворяват неравенствата:

$$\mathbf{n}_i\cdot\mathbf{T} = \mathbf{n}_i\cdot\left[\mathbf{Q}(t)-\mathbf{P}_i\right]\geq 0, \quad i=1,\ldots,n,$$

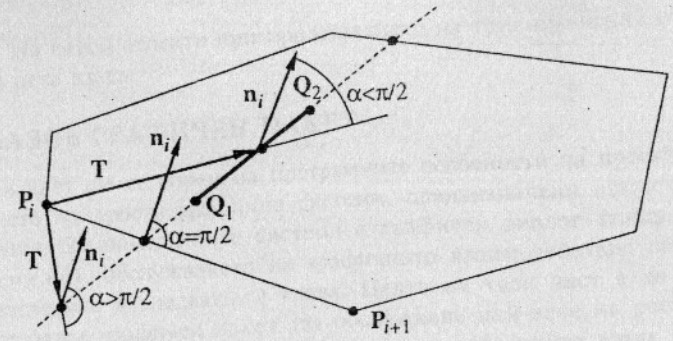
в които след заместване на Q(t) от [3.4] ще получим:

$$\begin{aligned}&\mathbf{n}_i\cdot\left[\mathbf{Q}_1+t\cdot[\mathbf{Q}_2-\mathbf{Q}_1]-\mathbf{P}_i\right]=\\ &\mathbf{n}_i\cdot\left[\mathbf{Q}_1-\mathbf{P}_i\right]+t\cdot\mathbf{n}_i\cdot\left[\mathbf{Q}_2-\mathbf{Q}_1\right]\geq 0, \quad i=1,\ldots,n\end{aligned} \qquad [3.6]$$

Ако въведем следните означения за векторите от горните неравенства: $\mathbf{V}_i=\left[\mathbf{Q}_1-\mathbf{P}_i\right]$ и $\mathbf{D}=\left[\mathbf{Q}_2-\mathbf{Q}_1\right]$, системата, решена спрямо t ще изглежда така:

$$\left\{-\frac{\mathbf{n}_i\cdot\mathbf{V}_i}{\mathbf{n}_i\cdot\mathbf{D}}\middle|\mathbf{n}_i\cdot\mathbf{D}>0\right\}\leq t\leq\left\{-\frac{\mathbf{n}_j\cdot\mathbf{V}_i}{\mathbf{n}_j\cdot\mathbf{D}}\middle|\mathbf{n}_j\cdot\mathbf{D}<0\right\}. \qquad [3.7]$$

Знакът на неравенството [3.6] се обръща в зависимост от знака на скаларното произведение $s_i=\mathbf{n}_i\cdot\mathbf{D}$. То ще е 0 само ако отсечката е успоредна на страната. В такъв случай съответното неравенство [3.6] ще бъде нарушено само ако $\mathbf{n}_i\cdot\mathbf{V}_i<0$.



Фиг. 3-23

Да отбележим още, че векторът $\mathbf{D}=\left[\mathbf{Q}_2-\mathbf{Q}_1\right]$ ще е нула само ако началото и краят на отсечката съвпадат. За всички останали неизродени случаи изведените формули ще дадат желания резултат.

Като вземем предвид, че параметърът може да взема стойности само в интервала [0,1], то от [3.7] можем да намерим интервала, за който параметърът t удовлетворява системата [3.6], с което определяме и търсените неизвестни t_{in}, t_{out}:

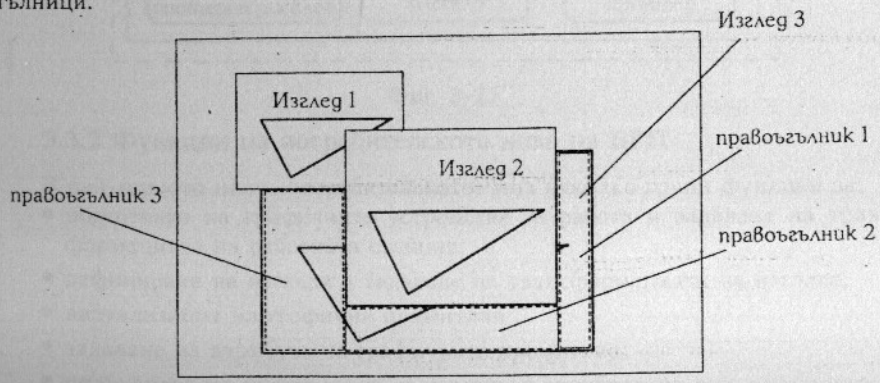
$$t_{in}=\max\left(\left\{-\frac{\mathbf{n}_i\cdot\mathbf{V}_i}{s_i}\middle|s_i>0\right\},0\right), \qquad t_{out}=\min\left(\left\{-\frac{\mathbf{n}_i\cdot\mathbf{V}_i}{s_i}\middle|s_i<0\right\},1\right).$$

```
int CyrusBeck2DClip(X1,Y1,X2,Y2,P,n,x1,y1,x2,y2)
float X1,Y1,X2,Y2;
Point P[]; int n;
float *x1,*y1,*x2,*y2;
{Vector D,V,N;
 float tin,tout,nv,s; int i;
   tin=0; tout=1;
   D.x=X2-X1;
   D.y=Y2-Y1;
   for (i=0; i<n; i++) {
      N.x=-P[(i+1)%n].y+P[i].y;
      N.y= P[(i+1)%n].x-P[i].x;
      V.x=X1-P[i].x; V.y=Y1-P[i].y;
      nv=DotProduct(N,V)
      if (s=DotProduct(N,D)) {
         if (s>0)
            tin =MAX(tin,-nv/s);
         else
            tout=MIN(tout,-nv/s);
      } else if (nv<0) return 0;
   }
   if (tout<tin) return 0;
   (*x1)=X1+tin*(X2-X1);
   (*y1)=Y1+tin*(Y2-Y1);
   (*x2)=X1+tout*(X2-X1);
   (*y2)=Y1+tout*(Y2-Y1);
   return 1;
}
```

Функцията, с която реализираме този алгоритъм, е обобщение на тази, която използвахме за алгоритъма на Лян-Барски. Забележете, че игнорираме тези страни, които са успоредни на отсечката, защото те не оказват влияние при определянето на границите на интервала на параметъра когато $\mathbf{n}_i \cdot \mathbf{V}_i \geq 0$.

НЕИЗПЪКНАЛИ МНОГОЪГЪЛНИЦИ. Както споменахме преди, при визуализацията на графични примитиви в системи, които позволяват работа в няколко прозореца на процеси се налага да се прави отсичане и спрямо по-сложни области от изправените правоъгълници и изпъкналите многоъгълници. Най-сложната задача е отсичането спрямо неизпъкнал многоъгълник и дори произволна многосвързана област. Съществуват общи алгоритми за отсичане спрямо области, зададени с произволен сложен контур, но използването на такива сложни алгоритми за интензивна задача като визуализацията е неоправдано.

В пета глава ще разгледаме един такъв алгоритъм, който обикновено се прилага за изпълнение на теоретико-множествените (булеви) операции: сечение, обединение и разлика на две произволни равнинни области. Тук ще отбележим само, че за нуждите на визуализацията е по-ефективно да се използват алгоритми за разделяне на неизпъкналите области на група от изпъкнали многоъгълници. Естеството на видимата област позволява в повечето случаи такова разбиване да се направи дори на група от изправени правоъгълници.



Фиг. 3-24

В общия случай се налага да се решават две основни задачи от изчислителната геометрия:

- определяне на това дали един многоъгълник е изпъкнал или не;
- разделяне на многоъгълник на изпъкнали многоъгълници.

Последната задача може да се реши и като се разбие многоъгълникът на триъгълници, които със сигурност са изпъкнали. Това всъщност е задачата за *триангулация на област*, която е важна задача в системите за автоматизация на инженерната дейност, където се прилага методът на крайните елементи. Тъй като това са по-крупни проблеми, обсъждането им заслужава специално внимание. На някои аспекти при представянето на триангулирани области ще се спрем в пета глава.

3.3 БАЗОВ ГРАФИЧЕН ПАКЕТ

В тази част ще се спрем на програмните особености на проектирането и изграждането на прости графични системи, предназначени изключително за визуализация. За по-сложните системи с графичен диалог (т.нар. *интерактивни системи*), обслужването на графичното взаимодействие ще бъде по-дробно разгледано в следващата глава. Целта на тази част е да представи един прост базов графичен пакет (за обслужване най-вече на растерни дисплеи); начините, по които в него се прилагат разгледаните дотук алгоритми; основните му компоненти и взаимодействието между отделните му части и приложните графични програми. Многообразието на подобни пакети ни кара да не се придържаме към никоя конкретна система, а по-скоро да покажем принципите на построяването ѝ, така че да са изпълнени изискванията, които си поставихме в началото на тази глава.

Съществуват най-общо казано два типа базови графични системи, които грубо можем да наречем:

- системи без запомняне на визуализираните обекти и
- системи със запомняне на визуализираните обекти.

При първите, които са и най-простите, графичната информация получена от приложните програми се предава незабавно към избраните графични устройства, без тя междинно да се съхранява в графичната система. Вторият тип системи притежават свои вътрешни структури от данни, в които получаваната информация се съхранява по подходящ начин преди изпращането ѝ към графичните устройства. Представяният тук базов графичен пакет (БГП) принадлежи към първия тип системи. На особеностите при създаването и използването на системи със запомняне ще се спрем в края на тази глава.

3.3.1 Структура на Базовия Графичен Пакет

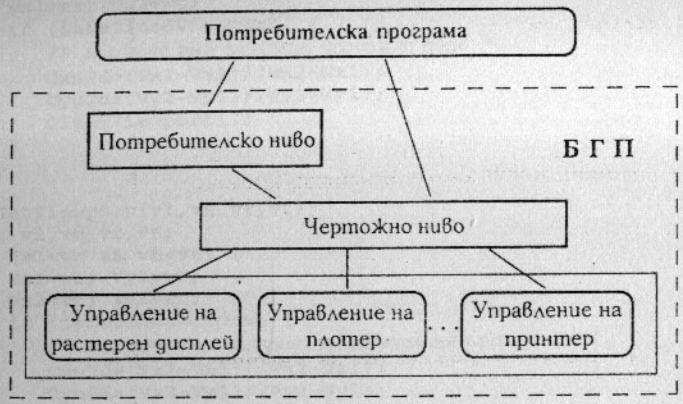
Приложният програмист има достъп до базовия графичен пакет чрез набор от функции, към които той може да прави обръщения от своята графична програма. Те са организирани в няколко основни групи. Едната включва функции, които осигуряват чертането в потребителска координатна система и установяването на параметри на визуализацията, които са свързани с потребителски размери. БГП разполага още и с функции за осъществяване на подобни графични действия и установяване на атрибути в чертожното пространство, както и с набор от глобални за БГП променливи, които най-често са недостъпни за приложния програмист. Можем да гледаме на БГП като на пакет, който има няколко нива.

Тук ще покажем една примерна структура на БГП, предложена съобразно разделянето, което направихме:

- *потребителско ниво*: функции, използвани директно от приложните програми за дефиниране на изгледи; визуализация на всеки от прос-

тите графични елементи (графичните примитиви: отсечка, дъга, символ, област и др.) и установяване на атрибутите на визуализацията (режим на растерно записване, цвят, дебелина, тип линия и др.);

- *чертожно ниво*: функции, извършващи подготовката на графичните примитиви за генериране върху графичното устройство: напр. отсичане, преобразуване на координати, визуализиране в нормирани чертожни координати и др.;
- *драйверно ниво*: управляващи програми, реализиращи ограничен набор от функции за визуализация върху всяко конкретно графично устройство.



Фиг. 3-25

3.3.2 Функции на потребителското ниво на БГП

В най-горното ниво (потребителското) има няколко групи функции за:

- подготовка на графичните устройства за работа и задаване на трансформацията на работната станция;
- дефиниране на изгледи и задаване на трансформацията на изгледа;
- визуализация на графични примитиви;
- задаване на атрибути на графичните примитиви;
- получаване на информация за текущите атрибути на визуализацията и състоянието на графичната система.

ПОДГОТОВКА НА ГРАФИЧНИТЕ УСТРОЙСТВА ЗА РАБОТА. Основно понятие в БГП е т.нар. *работна станция*. Това е устройството за графичен изход, с което приложната графична програма (а също и съответната изчислителна система, която ползваме) разполага. Една изчислителна система може да разполага с няколко работни станции: един или два графични дисплея, плотер, принтер и др. Преди да започне визуализацията на каквито и да е обекти е необходимо да се инициализира тази работна станция, която ще използваме в момента. Тази инициализация задава трансформацията на работната станция, която разгледахме в 3.1.4 (фиг. 3-7). В БГП има две функции за работа с *работната станция*: за инициализация и за завършване на работата върху нея:

```
void OpenWorkstation(device, type)
void CloseWorkstation()
```

Простите пакети (като този, който описваме) позволяват работа само върху една работна станция в определен момент. За да визуализираме даден обект върху екрана на дисплея и после да го изрисуваме върху плотер е необходимо да следваме последователността:

```
OpenWorkstation(SCREEN, VGA_DISPLAY);
DrawMyObjects();
CloseWorkstation();
OpenWorkstation(PLOTTER, HP7586);
DrawMyObjects();
CloseWorkstation();
```

При персоналните компютри най-често работната станция е и конзолата на компютъра, която служи за въвеждане на команди и данни предимно в текстов режим. В този случай инициализацията е необходима, за да превключва този терминал в графичен режим, а `CloseWorkstation()` за да го връща обратно в неговия текстов режим.

ЗАДАВАНЕ НА ИЗГЛЕДИ В БГП. Преди в приложната графична програма да се направи обръщение към каквато и да е функция, водеща до генериране на изображение върху устройството, е необходимо да се зададе изгледът, в който това изображение се намира. Във всеки момент БГП поддържа само един изглед - *текущия изглед*, зададен от последно дефинираните прозорец и чертожно поле. Прозорецът и чертожното поле от своя страна се задават поотделно чрез функциите:

```
int SetWindow(wl,wb,wr,wt)
int SetViewport(vl,vb,vr,vt)
```

Всяко ново предефиниране на прозореца или чертожното поле, определя нов изглед, като информацията за стария се загубва. В някои графични пакети има възможност да се зададат няколко изгледа, визуализацията в които да се извършва едновременно. Това е особено необходимо за тримерните системи, където един изглед може да не е достатъчен за потребителя за да добие той представа за формата на визуализираните пространствени обекти.

Чертожното поле в БГП се дефинира в *нормирана координатна система*, като приложният програмист може да получи информация за това какво е отношението на височината към ширината на устройството чрез функцията:

```
float GetDrawingAreaRatio()
```

За приложната програма, използваща БГП, изгледът е напълно определен от осемте параметъра на горните две функции. В самия БГП един изглед се характеризира и с двойката трансформационни коефициенти, които се из-

ползват във формулите [3.1] за задаване на *трансформацията на изгледа*.

В някои системи чертожното поле се дефинира винаги след прозореца и функцията SetViewport се грижи да установи тези коефициенти. Това не е голямо ограничение, но в нашия БГП задаването на прозореца и чертожното поле може да става в произволен ред и всяко обръщение към SetWindow или SetViewport предефинира текущия изглед. Чертожното поле на новия изглед се изчиства с обръщение към функцията ClearViewport.

```
int SetWindow(wl,wb,wr,wt);
float wl,wb,wr,wt;
{ if (wl<wr && wb<wt) {
     Wx1=wl; Wy1=wb;
     Wx2=wr; Wy2=wt;
     if (isDefinedViewport) {
        /* ако вече има зададено чертожно поле */
        Cxu2d=(Vx2-Vx1)/(Wx2-Wx1);
        Cyu2d=(Vy2-Vy1)/(Wy2-Wy1);
        ClearViewport();
     }
}

int SetViewport(vl,vb,vr,vt);
float vl,vb,vr,vt;
{ if (vl<vr && vb<vt) {
     Vx1=wl; Vy1=wb;
     Vx2=wr; Vy2=wt;
     if (isDefinedWindow) {
        /* ако вече има зададен потребителски прозорец */
        Cxu2d=(Vx2-Vx1)/(Wx2-Wx1);
        Cyu2d=(Vy2-Vy1)/(Wy2-Wy1);
        ClearViewport();
     }
}
```

Приложният програмист може да получи информация за текущите размери на прозореца и чертожното поле чрез функциите:

```
GetWindow(&wl,&wb,&wr,&wt)
GetViewport(&vl,&vb,&vr,&vt)
```

Тази информация може да бъде използвана за реализирането на такива операции като: увеличаване или намаляване на изображението (zoom-in, zoom-out), подобно на възможността за "приближаване" и "отдалечаване" от обекта при кино- и видео-камерите; хоризонтално или вертикално преместване на прозореца при запазване на чертожното поле (т.нар. scroll) и др. Следната програма показва увеличаване два пъти на частта от изображението, разположена в центъра на изгледа.

```
void ZoomInTwice()
{float wl,wb,wr,wt;
 float cx,cy, xhdim, yhdim;
   GetWindow(&wl,&wb,&wr,&wt);
   cx = (wr+wl)/2; xhdim = (wr-wl)/2;
   cy = (wt+wb)/2; yhdim = (wt-wb)/2;
   SetWindow(cx-xhdim,cy-yhdim,cx+xhdim,cy+yhdim);
}
```


Приложният програмист разполага и с функции за преобразуване на координати от потребителски в чертожни и обратно. Тези функции използват трансформационните формули [3.1]. Преобразуването на разстояния (радиуси, дължини, ширини, дебелини и др.) се извършва, като се преобразуват съответните вектори, тъй като в общия случай мащабирането по осите може да е различно.

```
void UserToNorm(X,Y,xp,yp);
void NormToUser(xp,yp,X,Y);
```

ГРАФИЧНИ ПРИМИТИВИ В БГП. Представените по-долу функции извършват визуализацията на няколко различни типа прости геометрични фигури, които наричаме *графични примитиви* на БГП. Техният набор може да варира значително, а ние сме се ограничили с по-важните от тях: отсечка, окръжност, дъга от окръжност, елипса (с оси успоредни на координатните), маркер (с формата на кръгче, кръстче или плюс), текстов низ, както и такива, които представляват затворени контури и следователно могат да бъдат запълвани: многоъгълник, окръжност, елипса и едносвързана област (зададена с един външен и множество от непресичащи се вътрешни контури).

```
void MoveTo(X,Y)                    void FillCircle(X,Y,R)
void DrawTo(X,Y)                    void FillEllipse(X,Y,A,B)
void Circle(X,Y,R)                  void FillPolygon(P,n)
void Ellipse(X,Y,A,B)               void FillArea(pP,P,nset)
void Arc(XC,YC,R,AB,AE)
void Marker(X,Y)
void Text(X,Y,string,angle)
```

Положението на всеки от горните примитиви се задава в потребителското пространство. Това важи за всички метрични параметри: както за координатите на точките, така и за големините на радиусите на окръжността и дъгата.

Първите две функции са много често използвани и са в основата на създаване на функции за примитиви като отсечка, правоъгълник и начупена линия: Line, Rectangle и Polyline. Тези две функции са въведени по аналогия с управлението на едни от първите чертожни устройства - плотерите. Подобно на плотера, можем да считаме, че чертането се извършва с операции зададени на въображаем писец:

- премести до нова точка без да оставяш следа - MoveTo и
- премести до нова точка, чертаейки отсечката до нея - DrawTo.

Преводът на тези два примитива на езика HP-GL за управление на плотер би бил тривиален:

```
MoveTo(X,Y) = PU; PA X Y;      вдигни перото от листа и премести до (X,Y)
DrawTo(X,Y) = PD; PA X Y;      свали перото до листа и премести до (X,Y)
```

Използвайки тези две функции, можем лесно се напишем функциите, генериращи и по-сложни геометрични фигури, както и да реализираме някои от останалите линейни примитиви. По-долу е показано изобразяването на една начупена линия:

```
void Polyline(P,n)
Point P[]; int n;
{int i;
   MoveTo(P[0].x,P[0].y);
   for (i=1;i<n;i++)
        DrawTo(P[i].x,P[i].y);
}
```

От останалите изброени функции само `Marker` променя положението на въображаемия писец, като го поставя в точката, зададена като входен параметър при обръщението към функцията. Подобно на начупената линия, в БГП може лесно да се реализира и такъв примитив като съвкупност от маркери `PolyMarker`, което е удобно средство за визуализиране на графики на таблично зададени функции, резултати от измервания и др.

Окръжностите и дъгите в БГП се задават с техните геометрични атрибути, каквито са радиусът и началният и крайният ъгли на една дъга. В много системи окръжностите не са отделен графичен примитив. Вместо тях се предоставя възможност за визуализация на елипса, която се задава с центъра и дължините на двете полуоси или чрез краищата на диагонала на правоъгълната си обвивка. И в двата случая главната ос на елипсата съвпада с оста Ox. Друг начин за задаване на окръжност е като дъга, чийто начален и краен ъгъл съвпадат. Както видяхме във втора глава, при растеризирането на една дъга не са необходими ъглите, а по-скоро координатите на крайните ѝ точки.

В някои системи е приет този именно начин за задаване на дъгите (чрез крайните точки и центъра), което прави по-лесно преобразуването на параметрите във входни данни за растеризиращите алгоритми. Последният начин дава възможност и да се използва принципът на въображаемия писец, а именно - началната точка на дъгата да не се задава изрично, а за тази цел да се използва положението, в което се намира въображаемият писец. Естествено е след изчертаването на дъгата писецът да се установи в нейната крайна точка. Този начин дава възможност за непрекъснато чертане на последователност от дъги и отсечки без да се вдига писалката на плотера. Една дъга би се визуализирала чрез обръщение към двойката функции:

```
MoveTo(Xstart,Ystart);
Arc(Xend,Yend,Xc,Yc);
```

Нека сега да се върнем към примера за визуализация на функцията $y = \sin(x)$ от фиг. 3-6, за да видим как можем да направим това използвайки БГП. Най-простият начин е да апроксимираме графиката на функцията с начупена линия, върховете на която да получим като увеличаваме x от 0 до 2π с някаква достатъчно малка стъпка и изчисляваме другата координата като стойността на тази функция в съответното x. В показания фрагмент сме взели максималното по големина чертожно поле, което съответства по пропорции на избрания прозорец:

```
SetWindow(0,-PI/2,2*PI,PI/2);
SetViewport(0.0,0.0,1.0,0.5);
MoveTo(0.0,0.0);
for (x=0; x<=2*PI; x+=0.005)
     DrawTo(x,sin(x));
```


АТРИБУТИ НА ВИЗУАЛИЗАЦИЯТА В БГП. Всеки от горните примитиви се визуализира съобразно текущите стойности на определен набор от атрибути, които управляват начина на изобразяване съобразно възможностите на съответното устройство. Някои от тези атрибути са приложими за всички примитиви, а някои - само за определен тип от тях. В БГП всеки примитив се характеризира със следните общи атрибути:

- цвят,
- тип на линията,
- дебелина на линията и
- режим на записване.

Смисълът на последния атрибут разгледахме в началото на втора глава. Той влияе на визуализацията само върху растерни графични дисплеи и е неприложим при използване на изходни устройства върху постоянен носител, каквито са плотерите и печатащите устройства. Затова пък неговото наличие за реализирането на различни интерактивни похвати е от голямо значение.

Специфични атрибути са тези, които определят начина за визуализиране само на определена група примитиви. В БГП това са:

- вид и големина на маркера,
- текстов шрифт, височина на символа и подравняване на текста и
- образец за запълване.

Установяването на текущата стойност на всеки атрибут се извършва поотделно чрез обръщение към някоя от функциите:

```
void SetColour(colorindex)
void SetLineType(type: {LT_SOLID | LT_DASHED | LT_DOTTED})
void SetLineWidth(width)
void SetWritingMode(mode: {REPLACE | XOR | AND | OR})
void SetFillPattern(type: {FP_HOLLOW | FP_SOLID | FP_HATCH})
void SetTextFont(fontname)
void SetTextAlignment(type: {TA_LEFT, TA_CENTRED, TA_RIGHT})
void SetCharHeight(height)
void SetMarkerType(type: {MR_CROSS | MR_CIRCLE | MR_PLUS})
void SetMarkerSize(size)
```

Тук сме показали само един начин за задаване на стойностите на атрибутите. В много системи само за задаване на цвят се използват няколко различни начина в зависимост от конкретните нужди:

- чрез индекс в предварително зададена таблица на цветовете;
- чрез наредена тройка числа, задаващи цвета в някоя цветова координатна система - RGB, CMY и др.;
- чрез използване на имена на цветове, които системата предлага.

Цветът е важен атрибут дори в случай, че визуализацията се извършва върху черно-бял дисплей. В простите графични системи, какъвто е предстаяният тук базов графичен пакет, не се предлагат специални функции за изтриване на примитив от изображението. Това се постига чрез визуализиране на примитива с цвят, който съвпада с фоновия. В системите, поддържащи таблица на цветовете, това съответства на цвета с индекс 0. Изтриването на един многоъгълник например приложната програма може да извърши със

следната последователност от обръщения към функции на БГП.

```
SetWritingMode(REPLACE);
SetColour(0);
Polyline(P,n);
```

В някои графични системи всички графични атрибути се наричат с общото име графичен контекст (*Graphic context*) и всеки примитив има допълнителен параметър, който задава графичния контекст, в който се изобразява примитива:

```
void DrawLine(x1,y1,x2,y2,gr_context)
```

Самият графичен контекст е най-често указател към структура, в отделните полета на която се съхраняват определени стойности за цвета, типа линия, режима на записване, шрифта и т.н., с които се визуализира примитивът. Този род системи обикновено дават възможост да се дефинират няколко различни контекста. Те най-често се получават, като се започва от един контекст и отделни негови атрибути се променят, след което резултатът се записва като нов графичен контекст. Приложната графична програма дефинира всички графични контексти (или по-голямата част от тях), които ще използва още в самото начало, а не преди визуализацията на всеки отделен примитив. С това се постига повишаване на бързодействието за сметка на паметта, която графичната система използва.

Трябва да отбележим, че графичните атрибути в системите без запомняне на визуализираните обекти са атрибути на визуализацията, а не на обектите, които се рисуват. Това означава, че не е възможно програмно да проверим една отсечка с какъв тип линия е нарисувана, както и да променим нейния тип след като тя веднъж е изобразена. Съхраняването на графичните атрибути като част от примитивите е вече един от елементите на моделирането за разлика от обикновената визуализация.

ПОЛУЧАВАНЕ НА ИНФОРМАЦИЯ ЗА СЪСТОЯНИЕТО НА ГРАФИЧНАТА СИСТЕМА. Приложният програмист много често се нуждае от динамична информация (по време на изпълнението на приложната програма) за това какви са текущите стойности и параметрите на някои величини в графичната система, от които визуализацията непосредствено зависи. Една от тези функции - `GetDrawingAreaRatio` (за получаване на съотношението височина към ширина на работното поле на графичното устройство) е необходима при задаване на изгледа. В общия случай една графична система може да дава информация за това:

- с какъв набор от устройства разполага компютърната система;
- какви са техните възможности: растерни, векторни, наличие на специален графичен хардуер, апаратни възможности за визуализация на криви и повърхнини, за визуализация на пространствени обекти с премахване на невидими линии и отчитане на светлинни ефекти, наличие на текстови шрифтове и др;
- информация за текущия изглед и текущата трансформация на работната станция и изгледа (в някои системи и за всички дефинирани изгледи);
- къде се намира въображаемият писец, който се управлява от `MoveTo` и `DrawTo`;
- какви са текущите стойности на атрибутите за визуализация: цвят, тип на линията, образец на запълване, или съответните полета в един графичен контекст и др.

3.3.3 Чертожно ниво на БГП

В чертожното ниво на базовия графичен пакет са реализирани функции, подобни на тези, разгледани в 3.3.2, но с тази разлика, че всички координати и размери се задават в нормираното чертожно пространство. Приложната програма може да използва функции от следните няколко групи:

- за визуализиране на графични примитиви в нормираното чертожно пространство;
- за задаване на стойностите на метрични визуални атрибути в него;
- за обслужване на графичния вход.

Наборът от графични примитиви е същият като в потребителското ниво, предоставяйки по този начин възможност на приложния програмист да работи еднотипно и в двете координатни пространства.

```
void MoveToNORM(X,Y)
void DrawToNORM(X,Y)
void CircleNORM(X,Y,R)
void EllipseNORM(X,Y,A,B)
void ArcNORM(XC,YC,R,AB,AE)
void MarkerNORM(X,Y)
void TextNORM(X,Y,string,angle)
void FillPolygonNORM(P,n)
void FillCircleNORM(X,Y,R)
void FillEllipseNORM(X,Y,A,B)
void FillAreaNORM(pP,P,nset)
```

Задаването на метрични атрибути има свой вариант и на това ниво:

```
SetLineWidthNORM(width)
SetCharHeightNORM(char_heigth)
SetMarkerSizeNORM(size)
```

Тук ще се опитаме да дадем аргументация на това защо се предлагат аналогични функции и в нормираното чертожно пространство. Да вземем като конкретен пример използването на символи в чертожните системи, т.е. приложните програми за автоматизиране на чертожната дейност. В повечето чертежи е необходимо да се разграничават следните два вида символи:

- *графични символи*: това са символи, които са съществена част от чертежа - тяхното положение и големина се определят от потребителските размери и всяка промяна на изгледа се отразява пряко върху тяхната големина. Това означава, че ако изображението се увеличи два пъти, то по същия начин ще се увеличат и символите. Те могат да бъдат не само текстови символи, но и такива, които обозначават специфични инженерни компоненти: транзистори, диоди и др. Естествено би било такива символи да се визуализират в потребителското пространство, т.е. тяхното положение и големина да зависят от размерите на модела.
- *символи за анотация*: това са такива символи, чиито размер и поло-

жение зависят само от големината на чертожното поле. Такива са текстовете за анотация, които не са съществена част от модела, а само подпомагат правилната интерпретация на чертежа. Поради тази причина тези символи се визуализират в чертожното пространство.

За първия тип ще използваме функциите Text и SetCharHeight, а за втория - TextNORM и SetCharHeightNORM.

Възможността приложните програми да могат да чертаят и в нормираното пространство може да се използва не само за символи. Много от останалите графични примитиви също се използват за анотация и тяхното положение и размери зависят по-скоро от големината на чертожното поле, отколкото от размерите в модела. Такива са например някои от оразмеряванията в един чертеж, щрих-линиите и различните технически символи: за допуск, за начин на обработка, за успоредност и перпендикулярност спрямо определени оси и т.н.

Друго приложение на функциите за визуализиране в чертожни координати е реализацията на някои от интерактивните похвати, които са само част от обслужването на графичния вход и които ще представим подробно в четвърта глава.

3.3.4 Реализиране на функциите в БГП

Функциите за визуализация в потребителското пространство се реализират чрез съответните функции за визуализация в чертожни координати. За целта е необходимо всяка от тях първо да извърши необходимите координатни преобразувания. В БГП отсичането се извършва спрямо чертожното поле, т.е. в нормирани чертожни координати от функциите от чертожното ниво. Причина за това е основно предоставената възможност на приложната програма да чертае примитиви и в нормирани координати, които също трябва да бъдат отсечени. И тъй като чертането в потребителски координати се осъществява с вътрешно обръщение към функциите за работа в нормираното чертожно пространство, то отсичане ще се извършва само веднъж. В потребителското ниво са дефинирани като глобални променливи коефициентите на трансформацията на изгледа, чиито стойности се установяват от SetWindow и SetViewport. Ето как се реализира например чертането на окръжност като се използват глобалните променливи, задаващи мащабирането:

```
void Circle(X,Y,R)
float X,Y,R;
{float xp,yp,a,b;
   UserToNorm(X,Y,&xp,&yp);
   a = R*Cxu2d;
   if (Cxu2d==Cyu2d) CircleNORM(xp,yp,a);
   else {
      b = R*Cyu2d; EllipseNORM(xp,yp,a,b);
   }
}
```

Тук радиусът на окръжността се пресмята чрез мащабния коефициент по оста *Ox*. В общия случай - когато мащабирането по двете оси не е еднакво - окръжността се чертае като елипса, двата радиуса на която се пресмятат чрез

двата мащабни коефициента. По-сложна е реализацията на функциите от вътрешното ниво. Необходимо е да се вземат предвид текущите стойности на графичните атрибути и да се извърши обръщение към няколко функции, които имат еднакъв интерфейс, но различна реализация за всяко отделно графично устройство. В БГП на това най-ниско ниво (което наричаме *драйверно ниво*) са реализирани следните функции:

```
LineOnDevice(x1,y1,x2,y2,pattern,width)
ArcOnDevice(x1,y1,xc,yc,x2,y2,pattern,width)
TextOnDevice(x1,y1,string,height)
FillOnDevice(P,n,fill_pattern)
NormToDevice(xp,yp,XD,YD)
DeviceToNorm(XD,YD,xp,yp)
```

В драйверното ниво на един растерен графичен дисплей са включени и специфичните функции, които изброихме в началото на втора глава. Те са пряко свързани с операции върху растера и затова са неприложими за други устройства:

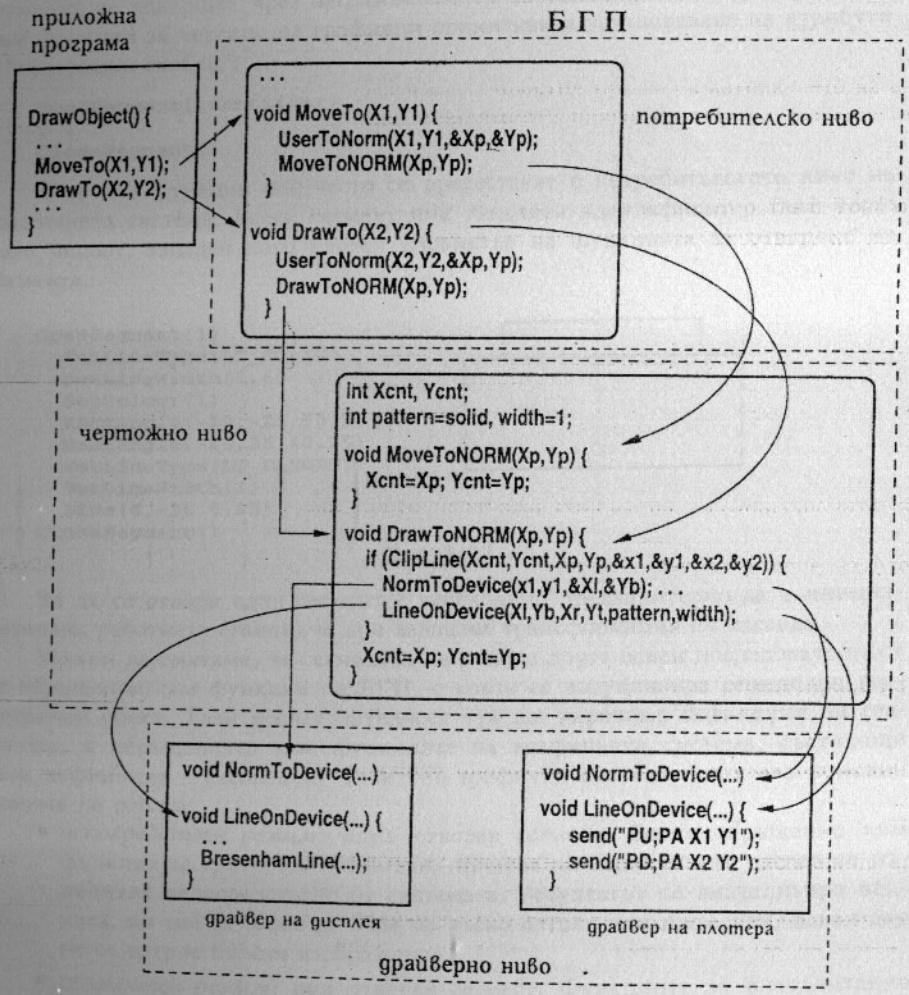
```
void SetWritingMode(REPLACE|AND|OR|XOR)
int  GetPixel(x,y)
void PutPixel(x,y,value)
void PutPixelRow(x,y,w,value)
void PutPixelBlock(x,y,w,h,buffer)
void GetPixelRow(x1,x2,y,buffer)
void GetPixelBlock(x,y,w,h,buffer)
```

На фиг. 3-26 е показан начинът, по който се извършва чертането на една отсечка върху растерен дисплей и върху плотер с една и съща приложна програма, без да се използват специфичните за растерния дисплей възможности. Във всяко от нивата са показани и променливите, които влияят на визуализацията. В чертожното ниво се поддържат стойностите на текущите атрибути на визуализацията, както и положението на въображаемия писец. Последните се съхраняват винаги в нормирани чертожни координати.

Вижда се, че на чертожното ниво на БГП се извършва по-голямата част от действията, а именно:

- отсичане на примитива спрямо чертожното поле;
- преобразуване на координатите и размерите до такива, в каквито работи съответното устройство;
- свеждане на чертането на по-сложните примитиви до чертане чрез отсечки и дъги: например функцията MarkerNORM ще сведе чертането на текущия тип маркер до обръщение към двойката отсечки или окръжност, от които този маркер е образуван.
- избиране на подходящ образец съобразно текущия тип линия с необходимото подравняване, когато съответната линия е част от по-сложен примитив: например отсечка, която е част от пунктирана начупена линия;
- избиране на подходящ начин на удебеляване: ако примитивът е отсечка, удебеляването да се извърши на драйверно ниво, ако той е многоъгълник, неговото удебеляване да се извърши чрез построяване на

поредица от еквидистанти (отместени образи), които да се визуализират чрез многократно обръщение към драйверната функция за чертане на отсечка.



Фиг. 3-26

Показаният БГП е само един примерен начин за реализиране на проста графична система. В него стандартизирането на работата с различни устройства се извършва в голяма степен за сметка на ефективността. В реалните системи всяко от нивата има много повече компоненти от тези, които по-казахме по-горе.

3.4 СТРУКТУРИРАНЕ НА ИЗОБРАЖЕНИЕТО

Разгледаният базов графичен пакет е проста графична система, в която визуализираните обекти не се съхраняват. Тя е много удобна в случай, че обектите, които се визуализират, няма впоследствие да бъдат променяни. В интерактивните приложни програми много често се налага да се правят промени в изображението съобразно действията на потребителя.

Да вземем простия пример с изтриването на един примитив. За да извърши това, приложната програма прави обръщение към функция на БГП за визуализиране на примитива с цвета на фона. Ако този примитив пресича друг примитив от изображението, то след изтриването на първия ще бъдат изтрити и пресечните му точки с този, който остава. Това обикновено е нежелателно и може да доведе до влошаване на качеството на изображението при многократно изтриване на елементи от него.

Един начин да се избегне този ефект е да се прерисуват всички примитиви освен този, който трябва да се изтрие. За тази цел обаче е необходимо наличието на списък от примитивите, които са визуализирани до този момент. Този списък може да се поддържа или от приложната програма (като част от нейния геометричен модел) или от базовата графична система. Най-ефективно би било прерисуването да се извършва на възможно най-ниско ниво (дори апаратно) и затова вторият начин е за предпочитане. Графична система, която поддържа списък на визуализираните обекти, се нарича *система със запомняне на изображението*.

Исторически, системите със запомняне на изображението са свързани с обслужването на векторните дисплеи, в които информацията за изображението във вида, получен от приложната програма (а именно като вектори) се съхранява в *дисплейния файл* като *дисплейна програма*. Векторните дисплеи се нуждаят от дисплейния файл за регенериране на изображението, а графичната система има възможност да модифицира този файл. Системите със запомняне на изображението поддържат аналог на дисплейния файл, до който приложните програми имат достъп чрез набор от специални функции. Елементите на този файл са отделните функции за визуализация: графичните примитиви, задаването на стойности на атрибутите и др. За да се улесни оперирането на приложните програми с дисплейния файл той може да бъде допълнително структуриран.

Най-простото структуриране е разбиването на дисплейния файл на *сегменти* - дисплейни подпрограми, които не могат да бъдат вложени една в друга. Системата GKS (Graphics Kernel System), приета за стандарт от международния институт за стандартизация ISO е типичен пример на система за структуриране на изображението във вид на сегменти. В някои по-съвременни графични системи структурирането може да се извършва на няколко нива, а именно няколко групи от графични примитиви да съставят нова група от по-високо ниво. Тези групи се наричат *структури*. По този начин изображението е йерархично организирано и съответства по-точно на изискванията на приложните графични програми. Пример за такава система е системата PHIGS, за която споменахме във въведението на тази книга. На тази система ще се

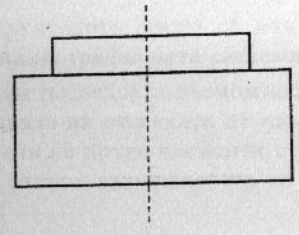
спрем по-късно, тъй като в нея освен частта за визуализация е реализирана и една *система за геометрично моделиране*.

По аналогия с дисплейния файл на векторните дисплеи, дефинирането на сегменти се извършва чрез заграждането на последователност от обръщения към функции за чертане на графични примитиви и установяване на атрибути с "програмни скобки":

```
OpenSegment(identifier)
. . .
CloseSegment()
```

Тези две функции най-често се предоставят в потребителското ниво на графичната система. Всеки сегмент има уникален идентификатор (най-често цяло число), зададен като входен параметър на функцията за отваряне на сегмента.

```
OpenSegment(1)
  SetLineType(LT_SOLID)
  SetLineWidth(1.5)
  SetColour(1)
  Rectangle(-50,-25,50,25)
  Rectangle(-40,25,40,35)
  SetLineType(LT_DASHED)
  SetLineWidth(1)
  Line(0,-28,0,45)
CloseSegment()
```



Фиг. 3-27

За да се отвори един сегмент е необходимо предварително да е инициализирана работната станция и да е зададена трансформация на изгледа.

Можем да смятаме, че сегментът не е нищо друго освен последователност от обръщения към функции на БГП, с които се визуализира определен геометричен обект. Тази последователност тук ще наричаме *дефиниция на сегмента*, а обръщенията към функциите на графичната система, съставящи тази дефиниция - *елементи*. Този тип графични системи имат два основни режима на работа:

- *непосредствен режим*: няма отворен сегмент. Всяко обръщение към функция за графичен примитив, промяна на атрибут или изглед се изпълнява непосредствено от системата. Резултатът се визуализира веднага, но той не може да бъде по-късно изтрит или променен, освен ако не се изтрие цялото изображение.
- *сегментен режим*: има отворен сегмент. Функциите за визуализация (елементите на сегмента) не се изпълняват незабавно, а изграждат дефиницията на сегмента.

След като сегментът е затворен, той може да бъде визуализиран само като цяло - да се изпълнят всичките му елементи - при обръщение към функция за включването му в текущия изглед:

```
PostSegment(segmentID)
```

Обръщение към тази функция може да се извърши само в непосредствен режим, т.е. тя не може да бъде елемент на някакъв (пък дори и същия) сегмент. Изтриването на един сегмент като цяло от изображението е също толкова просто. Приложният програмист трябва да се обърне в непосредствен режим към функцията:

```
UnpostSegment(segmentID)
```

По този начин сегментът ще бъде изключен от изображението без нежелателния ефект на изтрити пресечни точки, за който говорихме в началото на тази част. Горната функция не изтрива дефиницията на сегмента от *дисплейния файл*. Тя само обявява сегмента за временно *невидим* или изключен от изображението. Можем да смятаме, че двете представени функции само променят видимостта на един сегмент.

Изтриването на сегмент както от екрана, така и на неговата дефиниция се извършва със следната последователност от функции в непосредствен режим:

```
EmptySegment(segmentID)
PostSegment(segmentID)
```

В системите, които поддържат няколко текущи изгледа, в които визуализацията се извършва едновременно, функциите за промяна на видимостта имат още един допълнителен параметър, който задава изгледа, в който ще се включи или изключи сегментът.

Сегментите не са фиксирани структури. Техните елементи (функциите, от които са изградени) могат да бъдат променяни. Тази промяна става като отново се отвори същият сегмент и се извърши обръщение към новите функции, които ще се добавят към неговата дефиниция, определяйки по този начин нови елементи за този сегмент. Промяната на един сегмент не означава само добавяне на нови функции. Тя се управлява от така наречения *режим на редактиране* - начинът, по който елементите се включват в отворения сегмент. Обикновено има два режима: на *заместване* и на *добавяне*. Техният смисъл е близък до режимите INSERT и OVERWRITE (вмъкване и заместване) в текстовите редактори. Установяването на подходящ режим става непосредствено преди отварянето на сегмента и се отнася за включването на всички следващи елементи.

```
SetSegmentEditing(mode: {SE_REPLACE | SE_INSERT})
```

Подобно на текстовия курсор, показващ къде да се извърши вмъкването или заместването, в сегментите има *указател в сегмента*. По подразбиране той сочи в началото на сегмента (към неговия първи елемент) ако режимът е *заместване* или след последния елемент ако режимът на включване е *добавяне*. При включване на нов елемент и в двата режима указателят се премества, за да сочи към елемента непосредствено след включения. Графичните системи предоставят и средства, с които позицията на този указател може да се задава явно. Това може да става по един от следните начини:

- Чрез задаване на поредния номер на елемента:

```
AdvanceToElement(number)
```

- Чрез определяне на отместването спрямо текущия указател:

```
OffsetElementPointer(offset)
```

- Чрез етикети. Има и две други функции, които позволяват адресация независимо от поредния номер на елемента:
  - функция, която маркира определено място в дефиницията на сегмента, давайки на съответното обръщение към функцията някакъв *етикет*:

```
PutLabel(label)
```

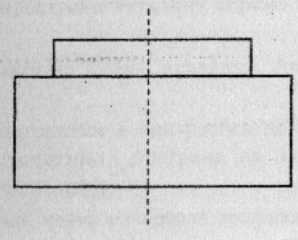
  - функция, която установява мястото, откъдето да започне добавянето или заместването с обръщения към нови функции (нови елементи):

```
AdvanceToLabel(label, offset)
```

Отместването `offset` спрямо маркираното място се измерва в брой елементи - обръщения към функции на графичната система.

Ще дадем един пример, за да поясним последната възможност. Ако искаме да променяме динамично типа на линията на отсечката от горния сегмент, без това да се влияе от различните промени на други елементи от него, то самата дефиниция на сегмента трябва да отчита възможността за такова действие:

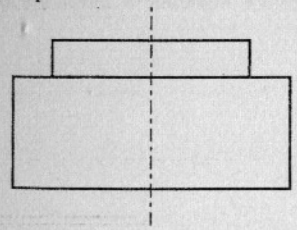
```
OpenSegment(1)
   SetLineType(LT_SOLID)
   SetLineWidth(1.5)
   SetColour(1)
   Rectangle(-50,-25,50,25)
   Rectangle(-40,25,40,35)
   SetLabel(my_label)
   SetLineType(LT_DASHED)
   SetLineWidth(1)
   Line(0,-28,0,45)
CloseSegment()
```



Фиг. 3-28

Отбелязаният с точка елемент носи етикета `my_label`. Смяната на типа на линията тогава ще се извърши с последователността, показана на фиг. 3-29. Отместването трябва да е 1, а не 0, в противен случай ще бъде заменено обръщението към самата функция за маркиране.

```
UnpostSegment(1)
SetSegmentEditing(SE_REPLACE)
OpenSegment(1)
   AdvanceToLabel(my_label,1)
   SetLineType(LT_DASHDOT)
CloseSegment()
PostSegment(1)
```



Фиг. 3-29

Елементите на сегмента могат не само да бъдат добавяни и заменяни.

Други функции, които модифицират съдържанието на сегмента, са тези за изтриване на елементи: на този, към който сочи указателя в сегмента; на всички елементи между два етикета; на всички елементи в определен интервал:

```
DeleteElement()
DeleteBetweenLabels(start_label,end_label)
DeleteElementRange(start_number,end_number)
```

Освен възможност за разнообразно променяне на съдържанието на сегмента графичните системи предоставят функции за промяна на идентификатора на сегмента. По този начин приложните програми могат да бъдат донякъде независими от замяната на един сегмент с друг в изображението.

```
ChangeSegmentIdentifier(oldID,newID)
```

Показаните по-горе средства дават възможност изображението да бъде структурирано и променяно както на ниво крупни групи от обекти - сегменти, така и на най-ниското възможно ниво - функциите за визуализация. В много системи се предоставят допълнителни функции, подпомагащи дефинирането на нови сегменти, като тази за копиране на всички елементи от един сегмент в дефиницията на друг.

```
CopyAllElementsOf(segmentID)
```

Копирането на елементите не създава връзка на вложеност на сегмента, чието съдържание се копира. Двата сегмента продължават да съществуват независимо един от друг. Вложеност е възможна само в графичните системи, които предлагат йерархично структуриране на изображението. Йерархичното структуриране обаче е средство за моделиране, а не за визуализация, поради което неговото използване ще разгледаме в пета глава.

Задачи

3.1 Използвайки функциите на БГП, напишете програма, която увеличава изгледа два пъти с център някаква точка от потребителското пространство. Как ще модифицирате тази програма, за да намалите изгледа два пъти.

3.2 Модифицирайте програмата, реализираща алгоритъма на Коен-Садърланд, така че да се елиминират изчисленията на наклона на отсечката в цикъла.

3.3 Напишете програма, която извършва отсичане на една отсечка спрямо изправен правоъгълник чрез рекурсивно разделяне на отсечката на две спрямо средната ѝ точка. За всяка от двете получени отсечки проверете дали е изцяло извън или изцяло вътре, а ако не е нито едно от двете - продължете деленето. Кога има край рекурсията? В какви координати е добре да се извършва такова отсичане?

3.4 Сравнете ефективността на алгоритъма на Коен-Садърланд с тази на алгоритъма от задача 3.3.

3.5 Какъв е максималният брой върхове на многоъгълника, който се получава при отсичане на един изпъкнал N-ъгълник от правоъгълник. А какъв е минималният им брой?

3.6 Напишете програма, за извършване на "външно отсичане" на отсечка от изправен правоъгълник, т.е. намиране на тези части от нея които лежат извън зададения правоъгълник.

3.7 Съставете програма, която намира пресечната точка на две отсечки, координатите на краищата на които са зададени като числа с плаваща запетая. Има ли разлика в алгоритъма, ако координатите на точките са целочислени?

3.8 Напишете програма, която реализира алгоритъма на Лян-Барски, като разгледате случаите на липса на пресечна точка отделно.

3.9 Допишете програмата, реализираща алгоритъма на Никол-Лий-Никол, като разгледате всички възможни случаи.

3.10 Съставете програма, която извършва отсичането на окръжност от изправен правоъгълник. Какъв е максималният брой дъги, които се получават?

3.11 Напишете програма, която намира пресечната точка (или точки) на отсечка и дъга от окръжност.

3.12 Даден е неизпъкнал многоъгълник, на който всичките страни са или хоризонтални, или вертикални и неориентираният ъгъл между всеки две съседни е 90 градуса. Съставете алгоритъм за разбиването на този многоъгълник на сума от непресичащи се и допълващи се правоъгълници.

3.13 Напишете програма, извършваща отсичане спрямо произволен изпъкнал многоъгълник, основана на алгоритъма на Садърланд-Ходжман.

3.14 Като използвате алгоритъма на Садърланд-Ходжман, напишете програма за извършване на "външно отсичане" на прост многоъгълник спрямо изправен правоъгълник.

3.15 Сравнете ефективността на алгоритмите на Коен-Садърланд, Лян-Барски и Никол-Лий-Никол.

3.16 Кой от представените в тази глава алгоритми е най-удобен за реализация чрез паралелна обработка. Напишете и съответната програма на някой от езиците, позволяващи дефинирането на паралелни процеси.

3.17 Използвайки БГП, напишете програма, която изчертава графика на функция на една променлива, зададена таблично, като дефинирате подходящо изгледа така, че графиката на функцията да е изцяло видима, без непременно мащабирането по двете оси да е еднакво.

3.18 Решете горната задача, но като запазите пропорциите на функцията по двете оси.

3.19 Добавете към горната програма и визуализация на потребителските координатни оси през т.0 от потребителското пространство и маркери върху всяка от осите през зададени интервали.

Глава 4

Графичен диалог

В тази глава ще разгледаме принципите за организиране на потребителското взаимодействие с една приложна графична програма. До много скоро при проектирането на графични програми най-голямо внимание се обръщаше на ефективността на използваните алгоритми и гъвкавостта и общността на създаваните модели на обектите, с които се работи. Изискванията към взаимодействието не бяха особено големи - най-важното бе потребителят да има достъп до всички функции на приложната програма, най-често използвайки някакъв набор от команди.

Едва през миналото десетилетие усилията на много разработчици се насочиха към увеличаване на *потребителската ефективност*, а не само на *компютърната ефективност*. Това естествено не може да не бъде за сметка на нещо друго, но е оправдано и наложено от следните условия:

- нарасналите изчислителни възможности на съвременните компютърни системи и масовото използване на растерни дисплеи;
- достъпът на широк кръг потребители до компютри и програми, взаимодействието на потребителя с които се извършва с графични средства;
- наситеността на пазара от аналогични един на друг програмни продукти, върху почти идентични компютърни платформи.

Последното условие кара много разработчици да привличат потребители чрез предоставяне именно на средства за ефективно потребителско взаимодействие, често наричани "*user-friendly*". Качеството на потребителския интерфейс в много случаи е най-важното условие за търговския успех на една програма. Докато в началото проектирането и използването на диалогови средства се е основавало най-вече на евристичен (ad-hoc) подход, то в последните години се създадоха методи за проектиране на *системи с графично взаимодействие*, които станаха съществена част на интерактивната компютърна графика.

Тук ние ще се спрем подробно на основните елементи на потребителското взаимодействие, на най-често налагащите се интерактивни дейности (позициониране, избор, задаване на непрекъснати величини, текст), както и на начините по които те се осъществяват с използването на различните видове интерактивни устройства (физически и абстрактни). Ще се спрем и на реализирането на различни интерактивни похвати, които повишават съществено качеството на взаимодействието с потребителя.

Особено внимание в тази глава е отделено на проектирането на графични диалог, използвайки целия арсенал от интерактивни похвати и потребителски метафори за нуждите на силно интерактивни приложни програми, както и на вида, в който те се предлагат и реализират от съвременните базови графични системи.

4.1 ВХОДНИ ДИАЛОГОВИ УСТРОЙСТВА

В тази част ще представим най-широко използваните устройства за графично взаимодействие и тяхната приложимост за осъществяване на различните интерактивни дейности в една графична система. Целта ни тук е не да разгледаме техните технически особености, а тяхната функционалност в контекста на графичния диалог и обслужването им от графичната система. Както видяхме в предишната глава, една от основните задачи на графичната система е да осигури независимост на създаваните приложни програми от наличните физически устройства. Това важи както за изходните устройства - дисплей, плотер, принтер и др., така и за входните диалогови устройства.

Поради по-голямото им разнообразие от това на изходните устройства, породено от различните по тип интерактивни дейности, които се обслужват, те са групирани в няколко класа. Графичните системи предлагат на приложните програми възможност за работа с няколко типа виртуални входни устройства, които съответстват именно на споменатите класове. Ще ги наричаме *абстрактни входни устройства* (Logical Input Devices).

4.1.1 Абстрактни входни устройства

Всяко абстрактно входно устройство съответства точно на определена интерактивна дейност. Пълният набор от тези дейности е следният:

- *позициониране*: задаване на координати, местоположение на обект в някаква координатна система: потребителска, чертожна, нормирана, двумерна, тримерна и др. Съответното абстрактно устройство носи името *локатор* (locator);
- *избор от възможности*: определяне на една от определен ограничен брой възможности, осъществявано от абстрактното устройство *избор* (choice);
- *определяне на стойност*: задаване на стойност от непрекъснат интервал от реални числа. Съответното абстрактно устройство носи името *валюатор* (valuator);
- *въвеждане на текстов низ*: Абстрактното устройство е добре познатата на читателя *клавиатура* (keyboard);
- *посочване*: избор на един от създадените до този момент графични обекти, които са предмет на обслужваната приложна графична програма. Съответното абстрактно устройство носи името *селектор* (pick);
- *задаване на поредица от позиции*: задаване на произволно дълга последователност от координати. В много от системите тази дейност се осъществява с устройството *локатор*, но в някои системи като GKS и PHIGS съществува отделно абстрактно устройство - *поредица* (stroke).

Различните графични системи предлагат различен брой входни абстрактни устройства, както и различни правила за тяхното дефиниране. Докато в някои системи се среща само подмножество от целия набор абстрактни устройства, в други е възможно дори дефинирането едновременно на няколко устройства от един и същ абстрактен тип - GKS и PHIGS. Системи от типа на X Window System са малко по-зависими от конкретните физически устройства, предоставяйки на приложния програмист възможност да използва повече спецификата на конкретната конфигурация. Това естествено е за сметка на по-голяма обвързаност и по-голяма сложност на приложните програми.

По-нататък в тази част ще разгледаме различните физически устройства, разделени в класове съобразно тяхната принадлежност към най-близкия абстрактен клас. Както ще видим по-късно, много рядко една конфигурация съответствя на всяко абстрактно устройство отделно физическо. На методите за моделиране на абстрактни устройства ще се спрем като на един от многото интерактивни похвати.

4.1.2 Локатори

Локаторите са най-многообразните диалогови устройства, поради различните особености, които притежават. Те могат да бъдат както *абсолютни*, така и *относителни* (в зависимост от типа координати, чието въвеждане позволяват); *равнинни* или *пространствени* (работещи в двумерна или тримерна координатна система); *непрекъснати* или *дискретни* (в зависимост от това дали позволяват или не задаването на плавни движения) и т.н.

Нека най-напред се спрем на няколко типични локаторни устройства, след което ще разгледаме техните предимства и недостатъци.

ТАБЛЕТ (tablet-digitizer). Устройството *таблет* (наричано също *дигитайзер* или *планшет*) е с плоска повърхност с размери от този на обикновения лист А4 до този на най-големите плотери. Таблетът е снабден с перо или позициониращ уред, чието положение върху повърхността му може да бъде отчетено от таблета и изпратено към компютъра. Най-често за тази цел се използва електрическа сензорна система. В общия случай компютърът може да получи от таблета следната информация:

- дали перото (или позициониращият уред) се намира близо до повърхността на таблета или е далече от нея. Това е еднобитова информация, наричана *близост на перото* (pen proximity);
- дали някой от бутоните е натиснат и ако това е така - неговият код;
- местоположението на перото (или позициониращия уред) върху повърхността на таблета в неговата физическа координатна система.

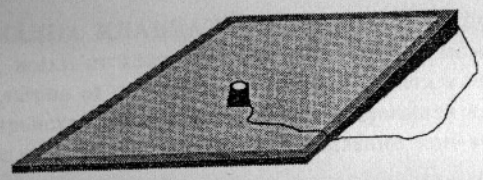
Таблетът е *абсолютно* позициониращо устройство, тъй като предаваните от него координати са тези на собствената му координатна система. Началото на тази координатна система има фиксирано разположение върху повърхността му (долния ляв ъгъл на работното му поле), а осите ѝ са успоредни на правоъгълника на това поле. Единиците, в които се работи, се определят от

неговата *разделителна способност*. Типична мерна единица е 1/40 част от милиметъра.

Таблетът се използва за "сваляне" на координати от съществуващи чертежи например при оцифроване на карти, които се закрепват върху повърхността му и се "прерисуват" с перото.

Наред с описания типичен таблет съществуват следните разновидности:

- *сонорни таблети*, при които положението на позиционират уред се определя по звуков път от система микрофони, разположени в ъглите на повърхността на таблета;
- таблети, чието перо може да отчита наклона и натиска спрямо повърхността на таблета;



Фиг. 4-1

МИШКА (mouse). Това широко разпространено устройство отчита само относителни премествания, най-често посредством търкалянето върху плоскост на малка топка, затворена в удобна за хващане кутия с бутони. Такъв тип мишка се нарича *механична мишка*, защото движението се предава по механичен път. Друга разновидност е *оптичната мишка*, която се движи върху специална повърхност с начертани хоризонтални и вертикални линии. Движението се отчита по оптичен път от фотосензори в зависимост от броя на пресечените линии.

И в двата случая двойката координати, които компютърната система получава от мишката, се интерпретират като отмествания спрямо зададена *текуща позиция*. При такова позициониране се разчита на обратната връзка - визуализиране на текущата позиция чрез графичен показалец върху дисплея. Преместванията на механичната мишка, когато тя не е в контакт с повърхността, както и движенията на оптичната мишка върху друга повърхност (не специалната оптично-чувствителна такава) не се отчитат и предават към компютъра.

Правени са опити и с т.нар. *крачна мишка* (foot mouse), която се управлява с крак и дава възможност на потребителя да използва и двете си ръце при едновременна работа с клавиатурата.

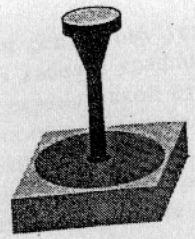
Мишките се отличават по броя на бутоните, като тези с 3 бутона са най-широко използвани, поради факта, че с три бутона могат да се зададат $2^3 - 1 = 7$ различни комбинации.

СЛЕДЯЩА ТОПКА (trackball). Това устройство може най-лесно да се опише като обърната мишка. Особеното в този случай е, че движенията се извършват с длан, което не дава възможност на потребителя да държи пръстите

си непрекъснато върху разположените върху устройството бутони. Също като мишката, топката се използва като относително устройство, защото тя няма своя физически определена координатна система.

РЪЧКА (joystick). Това устройство позволява премествания в четирите основни посоки, а в някои варианти и по диагонал. Някои ръчки имат и допълнителна степен на свобода, като позволяват въртенето на самата ръчка около оста си. За разлика от мишката и топката ръчката може да се използва като *абсолютно* устройство, като се приеме нейното изправено положение за начало на координатната система. Много от тези устройства имат пружини, които връщат ръчката в това "начално състояние".

Абсолютното позициониране с това устройство е много неточно и затова то се използва по-често за управление на т.нар. *графичен показалец*, изобразяван върху екрана на дисплея, задавайки движението му в дадена посока. Това движение не винаги е равномерно - продължително задържане на ръчката в определена посока може да води до увеличаване на скоростта на движение на графичния показалец в съответното направление.



Фиг. 4-2

ПРОСТРАНСТВЕНА ТОПКА (space-ball). Това е устройство, съчетаващо възможностите на ръчката и топката и отчитащо относителни премествания с шест степени на свобода. Използва се най-често за задаване на местоположението и ориентацията на тримерни обекти, разчитайки на незабавната обратна връзка (промяната на положението на обекта в наблюдавания изглед върху екрана).

ДИСПЛЕЙ С ДИРЕКТНО ПОЗИЦИОНИРАНЕ (touch-sensitive display). Това е дисплей, върху който местоположението може да се задава директно - при физически контакт на пръста на потребителя с екрана на дисплея. Дисплеите с директно позициониране са основани на различни технически принципи:

1. При *светлинните дисплеи* определянето на положението става чрез инфрачервени емитери и сензори, разположени вертикално и хоризонтално по краищата на екрана. Положението по всяка от осите се определя по прекъснатите инфрачервени връзки между съответен емитер и сензор.

2. Някои дисплеи са покрити с тънко стъкло-проводник, докосването до който се отчита по електрически път. В друг вариант дисплеят е покрит с два слоя стъклоподобен материал - проводник и резистор съответно, които при натиск се допират и променят пада на напрежение, по който се установява позицията. Силата на натиска може да бъде допълнително отчетена по контактната площ.
3. Дисплеите, базирани на сонорен принцип, наподобяват мрежата от инфрачервени лъчи при първия тип, с тази разлика, че в този случай разделителната способност може по-лесно да се увеличи.

4.1.3 Устройства за избор

ФУНКЦИОНАЛНА КЛАВИАТУРА (functional keyboard). Това е система от клавиши, всеки от които съответства в даден момент на определена възможност, предлагана от системата. Функционалната клавиатура може да е както отделно устройство, така и част от традиционната клавиатура, в която набор от бутони е специално заделен за използване най-вече от приложните програми.

Функционалните клавиши могат да бъдат с постоянни имена-функции, а могат да имат и панел от течен кристал (разположен близо до тях или като част от самите тях), в който се изписва тяхното текущо предназначение.

УПРАВЛЕНИЕ С ГЛАС (voice control). Това е устройство за разпознаване на определен набор от предварително въведени звукови честотни мостри на подбрани ключови думи. Гласовата връзка е все още нетипична за съвременните графични системи, въпреки очевидните предимства, свързани с освобождаването на ръцете за работа с другите входни устройства. Недостатък на гласовата връзка е невъзможността на потребителя да използва комуникация с глас за други цели например за разговор с други потребители.

4.1.4 Валюатори

ПОТЕНЦИОМЕТЪР (potentiometer). Това устройство прилича на използваните в практиката потенциометри, даващи възможност за плавно увеличаване или намаляване на дадена величина. Аналогично на потенциометрите за усилване и намаляване на звука в аудио-системите, тези устройства биват *ротационни* или *линейни* (наричани още *плъзгачи*).

Линейните плъзгачи са винаги ограничени. Те имат фиксирана минимална и максимална позиция, в границите на които управляваната величина се мени, което ги прави абсолютни устройства.

Ротационните потенциометри могат да бъдат както ограничени - т.е. да задават минимална стойност при завъртане крайно наляво и максимална при крайно завъртане надясно - така и неограничени. Последните са удобни при задаване на периодични величини какъвто е например ъгълът на въртене на едно тяло спрямо фиксирана ос. В графичните системи се използват изключително ротационни потенциометри и то най-често неограничени.

4.1.5 Устройства за въвеждане на текст

КЛАВИАТУРА (keyboard). Това е най-често използваното входно устройство поради наличието на голям набор от приложения, изискващи въвеждане на текст. Съвременните клавиатури, макар и в различни разновидности, се основават на клавиатурата на пишещата машина и съпоставят на всеки текстов символ по един клавиш. По аналогия с пишещите машини комбинацията със специални клавиши (долен и горен регистър) осигурява получаването на алтернативни символи - напр. главни букви и кодове за управление на работата на процесите (Shift, Control, Alt). Някои клавиатури позволяват *композиране на символи* за някои азбуки, в които се използват ударения. Всички те обаче предават по един код към компютъра.

За разлика от тях, *акордовите клавиатури* предават на компютъра кодовете на всички едновременно натиснати клавиши. Те използват т.нар. *акорди* от малък брой клавиши - например 5, по един за всеки пръст - за въвеждането на текстови символи. Естествено, този начин на въвеждане изисква предварително обучение на оператора.

4.1.6 Селектори

СВЕТЛИННО ПЕРО (light pen). Това устройство има формата на писалка, свързана най-често чрез кабел с графичния дисплей. С нея потребителят посочва директно върху екрана някой от изобразените там обекти. На върха на светлинното перо се намира фотодиод, които при откриване на светлинен пулс изпраща сигнал за прекъсване към дисплея. Графичният дисплей, който от своя страна в момента извършва регенерация на изображението, изпраща към графичната система намиращите се в този момент стойности в *X* и *Y* регистрите на видео-контролера.

Светлинното перо е било удачно решение за векторните дисплеи, където прекъсването на регенерацията на изображението е давало достъп до примитива и даже до сегмента от дисплейната програма, чието изобразяване в момента на прекъсването се отчита от светлинното перо. Това устройство се използва вече много рядко, тъй като се влияе от допълнителни източници на светлина и продължителната му употреба води до физическа умора. Това е единственото физическо устройство за посочване на обекти, а не на позиции (локатор), тъй като не позволява задаване на положение върху екрана, което не е осветено.

4.1.7 Режими на работа на входните устройства

Всяко от разгледаните абстрактни устройства може да работи в няколко различни режима, които се предоставят за избор на приложния програмист от всяка графична система. Различните режими са свързани с осъществяването на различни дейности от приложните графични програми, както ще видим по-нататък.

Когато една приложна програма заяви използването на дадено абстрактно устройство, тя трябва точно да определи как графичният вход с това устройство ще бъде съобразен с действията на потребителя и с предаването на

управлението в приложната програма след завършване на въвеждането. Това се задава с режима на работа на устройството: ЗАЯВКА, СЪСТОЯНИЕ, СЪБИТИЕ (REQUEST, SAMPLE, EVENT).

```
SetLocatorMode(LocatorID, IM_REQUEST)
SetValuatorMode(ValuatorID, IM_SAMPLE)
SetChoiceMode(ChoiceID, IM_EVENT)
```

REQUEST (ЗАЯВКА) - при графичен вход в този режим приложната програма изчаква потребителя да завърши въвеждането на информацията, след което тя отново получава управлението. Завършването на входа се отбелязва например с натискането на бутон на локатора, клавиша RETURN при въвеждане на текстов низ и т.н. Този режим е удобен в случай, че потребителският вход е определящ за по-нататъшното изпълнение на приложната програма и работата ѝ не може да продължи без потребителска намеса.

Приложната програма получава пълната информация за графичния вход - позиция, натиснат клавиш, посочен сегмент или примитив, зададена стойност, избрана възможност и т.н.

```
RequestLocator(LocatorID, &X, &Y, &status)
RequestPick(PickID, &segmentID)
```

Можем да считаме, че във всяка от горните функции е реализиран цикъл, в който се проверява състоянието на устройството и от този цикъл се излиза само ако това състояние съответства на завършен графичен вход.

SAMPLE (СЪСТОЯНИЕ) - при вход в този режим приложната програма не изчаква потребителя за осъществяване на графичен вход. Графичната система проверява състоянието на входното устройство и изпраща веднага резултата към приложната програма, която получава веднага и управлението. Този режим се използва в случай, че интерактивната дейност се организира от самата приложна програма или тя може да продължи работата си, ако потребителят не се намеси (има отнапред ясно действие по подразбиране).

И в този случай приложната програма отново получава пълната информация за състоянието на устройството - текущата позиция, натиснат клавиш (ако има такъв), текущата зададена стойност и т.н.

```
SampleLocator(LocatorID, &X, &Y, &status)
SampleValuator(ValuatorID, &value, &status)
```

EVENT (СЪБИТИЕ) - в този режим всички потребителски действия с входното устройство се регистрират като *асинхронни събития*: натискане на бутон или клавиш, промяна на текуща стойност или позиция, избор на възможност и др. Те се записват във *входна опашка*, която се поддържа от графичната система и до която потребителят има достъп чрез обръщение към функцията:

```
AwaitEvent(timeout, deviceID, event_type)
```

Тази функция проверява дали има събитие от определен тип във входната опашка на съответното устройство. Ако има, тя зарежда данните за последното настъпило такова събитие във временна структура, откъдето приложната програма може да получи данни за състоянието на входното устройство по време на настъпилото събитие чрез обръщение към някоя от функциите:

```
GetLocator(LocatorID, &x, &y, &status)
GetKeyboard(KeyboardID, string)
GetValuator(ValuatorID, &value, &status)
```

Ако входната опашка е празна, графичната система изчаква определено време `timeout` за получаването на събитие. Ако такова събитие не постъпи в опашката, управлението се предава на приложната програма заедно с информация, че входната опашка е празна.

Както видяхме в горните примери, при всяко обръщение към абстрактното устройство се указва и режимът, който се използва за съответното обръщение. По този начин е възможно от едно устройство, дефинирано да работи в режим ЗАЯВКА по подразбиране, да се получи информация за текущото му състояние (графичен вход в режим СЪСТОЯНИЕ).

4.2 ОСНОВНИ ИНТЕРАКТИВНИ ДЕЙНОСТИ И ПОХВАТИ

Описаните абстрактни устройства се използват за осъществяване на набор от основни *интерактивни дейности*. Тъй като в повечето случаи графичната система не разполага с всички представени по-горе абстрактни устройства, тя трябва да ги моделира с наличните физически. В тази част ще разгледаме средствата, които една графична система предоставя за моделирането на всяко от абстрактните устройства.

От друга страна, приложните програми могат да използват различни *интерактивни похвати*, които улесняват осъществяването на определена интерактивна дейност. Ние ще разгледаме тези похвати в контекста на всяка конкретна дейност. Интерактивните похвати, които ще представим се основават на определени принципи за организиране на диалога, които ще обобщим по-късно в тази глава.

4.2.1 Позициониране

Тук ще разгледаме особеностите при задаването на позиция в равнината. Това е двойка координати (x,y), необходима на приложната програма в потребителската координатна система. Трансформацията от физическата координатна система на локатора (или на устройството, което го симулира) в потребителски координати се извършва най-често от графичната система. Каква да бъде тази трансформация се определя, като се отговори на следните въпроси:

1. Дали координатите от локатора се преобразуват направо в потребителски координати или се преобразуват първо в чертожни (или нормирани) координати и след това чрез трансформацията на изгледа (представена в предната глава) се преобразуват в потребителски? Отговорът на този въпрос зависи от конкретните нужди и вида на локатора. Ако се използва таблет за въвеждане на топографска информация, втората възможност би влошила много точността, тъй като един таблет има типично 20000 × 20000 точки, а един дисплей -

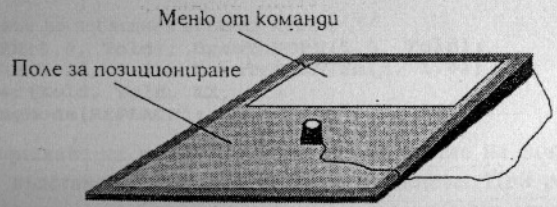
най-много 4096 × 4096. От друга страна първата възможност би усложнила *обратната връзка*, необходимостта от която ще покажем малко по-късно. Удобно решение е да се използва преобразуване до нормирани координати. По този начин, обратната връзка може да се извършва без загуба на точност чрез функциите от чертожното ниво на графичната система.

2. Каква част от работното поле на локатора може да се използва за позициониране? При всички абсолютни позициониращи локатори работното поле може да се използва както изцяло за задаване на координати, така и да се отдели само някаква негова част за тази дейност. Последното се прилага много често по две основни причини:

- работното поле при таблетите е най-често квадрат, докато екранът на дисплея и/или растерната му матрица може да не е;
- много често определена част от таблета се използва само като устройство за избор (най-често за избор на команди от постоянно меню) и дори за въвеждане на текст.

3. Какво е съотношението на разделителната способност на локатора към разделителната способност на екрана на дисплея по всяка от осите? Почти никога разделителната способност на едно абсолютно устройство не съвпада с тази на екрана. Това не е така дори при дисплеите с директно позициониране, които привидно са едно и също входно и изходно устройство. При тях входната разделителна способност е типично няколко порядъка по-малка от тази на визуализацията.

Поради различни причини, една от които представихме по-горе, е възможно еднакви движения на локатора по X и Y да предизвикват различни по големина премествания в съответните посоки по всяка от осите. Това е неприемливо единствено при точно дигитализиране. В останалите случаи потребителят разчита на обратната връзка за точно позициониране.



Фиг. 4-3

В следващите примери на тази глава ще считаме, че локаторът връща позиция в нормирани координати. Трансформацията до нормирани координати се дефинира при инициализацията на локатора по начин, подобен на дефиницията на един изглед.

Входните параметри на инициализацията в общия случай (особено когато локаторът е свързан с определен изглед) могат да бъдат доста различни в отделните графични системи:

- идентификаторът на устройството;
- началните координати в потребителското пространство;
- идентификаторът на изгледа (ако системата позволява едновременна обработка в няколко изгледа);
- границите на работното поле за позициониране в координатите на устройството.
- размерът и форматът на данните, които се получават при всяка заявка.

В разширението на БГП за обслужване на графичния вход това са само идентификаторът на устройството и границите на полето за позициониране върху него:

```
void InitLocator(LocatorID, XminD, YminD, XmaxD, YmaxD)
```

ОБРАТНА ВРЪЗКА. От особена важност за изпълнението на тази дейност е наличието на средства за визуализиране на действията на потребителя докато той извършва позиционирането. Това се прави като се използва специален графичен елемент, наречен *графичен показалец*, движенията на който са отражение на движенията, които потребителят извършва с локатора (писалката на таблета, мишката и т.н.).

Обратната връзка по правило се осигурява от графичната система, а не от приложните графични програми. В съвременните графични дисплеи тя се извършва от вградените им апаратни средства. Приложният програмист може само да я управлява чрез:

- дефиниране на вида на графичния показалец;
- допълнително визуализиране на текущите стойности на потребителските координати на показалеца в определено поле от екрана и др.

Тъй като в мнозинството системи локаторът се използва не само за позициониране, удобно е за се избере специален тип показалец, който да подсказва на потребителя, че системата очаква задаване на позиция. Най-разпространени типове показалци за тази цел са малко кръстче или кръст, образуван от една хоризонтална и една вертикална линии, ограничени от чертожното поле и пресичащи се в текущата позиция, зададена от локатора (известен като "crosshair cursor").

Графичните системи реализират движението на графичния показалец (наричан често и *курсор*) по различен начин в зависимост от вида на показалеца и от типа на дисплея, върху който се извършва визуализацията. Тук ще представим два от възможните начини за това:

1. *Векторно прерисуване*: Този сравнително прост начин се използва, когато графичният показалец има векторен вид (може да се получи чрез изрисуването на ограничен брой отсечки). Типичен пример за това е кръст, образуван от две ограничени от границите на екрана хоризонтална и вертикална линии.

Векторите в този случай се визуализират чрез обръщение към функции от чертожното ниво на БГП. Режимът на изобразяването им върху екрана е `xor` (изключващо или). Това позволява многократно прерисуване на показалеца, без да се променя изображението върху екрана. Преди започване на гра-

фичния вход показалецът се визуализира в режим **xor** ("изключващо или") в текущата си позиция върху екрана. Тъй като по-голямата част от един чертеж е с цвета на фона (напр. черен), при противоположен цвят на показалеца (напр. бял), по-голямата част от показалеца ще се запази (в случая - бяла). Там, където векторите на показалеца пресичат елементи от изображението, цветът на пресечните точки ще е "изключващото или" на двата цвята.

Графичната програма периодично проверява състоянието на локатора и ако има преместване в нова позиция, старият образ на показалеца се прерисува в същия режим, при което изображението под него се възстановява. Показалецът се изобразява в същия режим в новата позиция, получена от локатора и този цикъл продължава, докато се получи сигнал от локатора, че въвеждането е завършило.

```
. . .
SetWritingMode(XOR);
SampleLocator(LOC, &Xold, &Yold, &status);
/* изрисуване на първия курсор */
MoveToNORM(0.0, Yold); DrawToNORM(1.0, Yold);
MoveToNORM(Xold, 0.0); DrawToNORM(Xold, 1.0);
while (status != BUTTON_PRESSED) {
   /* получаване на новата позиция от локатора */
   SampleLocator(LOC, &Ynew, &Ynew, &status);
   if (Xnew!=Xold || Ynew!=Yold) {
      /* изтриване на стария курсор */
      MoveToNORM(0.0, Yold); DrawToNORM(1.0, Yold);
      MoveToNORM(Xold, 0.0); DrawToNORM(Xold, 1.0);
      /* изрисуване на курсора в новото положение на локатора */
      Xold=Xnew; Yold=Ynew;
      MoveToNORM(0.0, Yold); DrawToNORM(1.0, Yold);
      MoveToNORM(Xold, 0.0); DrawToNORM(Xold, 1.0);
   }
}
/* изтриване на последния останал курсор */
MoveToNORM(0.0, Yold); DrawToNORM(1.0, Yold);
MoveToNORM(Xold, 0.0); DrawToNORM(Xold, 1.0);
NormToUser(Xold, Yold, &X, &Y)
SetWritingMode(REPLACE);
```

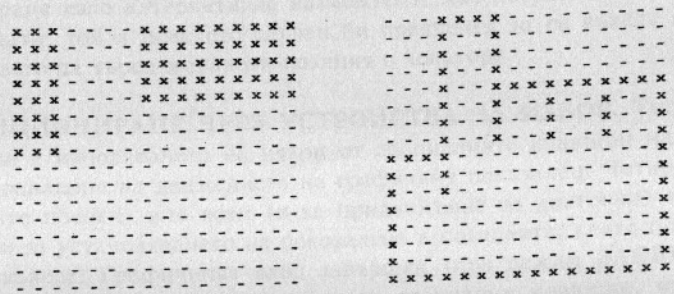
След завършване на входа последното положение на локатора се прерисува, за да се възстанови окончателно изображението. При реализирането на този прост алгоритъм трябва да се вземат под внимание няколко особености, които биха подобрили качеството на обратната връзка:

- преди да се прерисува старото положение на локатора и да се изрисува той на новото трябва да се провери дали новата позиция на локатора не е точно същата като старата. Ако това не се прави, показалецът започва да "мига" щом се остави локаторът неподвижен.
- при завършване на графичния вход е възможно последната визуализирана позиция на локатора да не съвпада с тази от последното обръщение към функцията, която го обслужва (когато е натиснат бутон например). В мнозинството от случаите е по-добре да се предаде към приложната програма последната визуализирана, а не последната получена от локаторната функция позиция, защото входът на потребителя зависи в голяма степен от обратната връзка.

По-горе се вижда примерна реализация на този алгоритъм за един чисто векторен показалец.

2. *Растерен образец*: Този начин се използва в повечето от съвременните растерни графични системи. Той е удобен, когато графичният показалец може да се представи растерно като сравнително малка матрица - например с 32 реда и 32 стълба. Графичната система съхранява изображението преди да визуализира курсора в съответната по големина временна памет и след преместване на курсора възстановява съхранената част от образа. Един начин да се дефинира курсорът е чрез два правоъгълни растерни образеца, които се налагат върху екрана един след друг - първият в режим **and**, а вторият в режим **or** - в позицията, получена от локаторното устройство. Използването на два образеца дава възможност да се визуализират курсори с повече цветове и със запълнени области в тях. В много системи приложните програмисти могат сами да извършват определянето вида на показалеца. По-долу е показано задаването на курсор във формата на ръка с протегнат показалец.

```
CursorMask PointingHand
= {     /* образец, който се записва върху екрана в режим AND */
    {0xE1FF,0xE1FF,0xE1FF,0xE1FF,0xE1FF,0xE000,0xE000,0xE000,
     0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000
    },  /* образец, който се записва върху екрана в режим OR */
    {0x1E00,0x1200,0x1200,0x1200,0x1200,0x13FF,0x1249,0x1249,
     0x1249,0xF001,0x9001,0x9001,0x8001,0x8001,0x8001,0xFFFF
    }
};
```



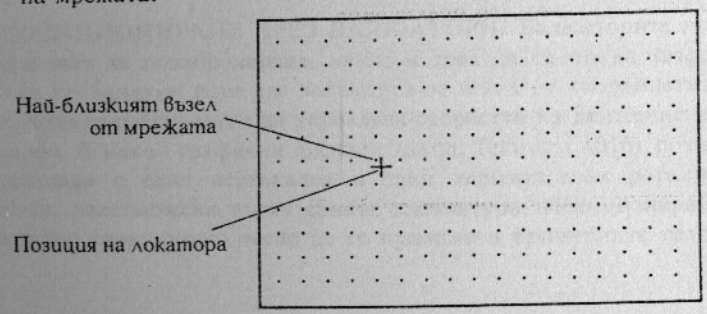
КООРДИНАТНА МРЕЖА. Полезно средство за позициониране е използването на координатна мрежа. Вместо текущите координати на локатора, графичната система предава на приложната програма най-близко разположения възел от тази мрежа чрез подходящо закръгляване на тези координати. Някои приложни програми дават възможност за позициониране само чрез координатна мрежа, т.е. не е възможно въвеждане на други точки, освен тези от координатната мрежа. Такива са пакетите за проектиране на печатни платки и интегрални схеми, в които разделителната способност е фиксирана поради характера на този вид приложения.

При проектиране на диалога на по-общите приложни графични програми

трябва да се предостави избор за позициониране с или без координатна мрежа. Началото и разстоянията между възлите по всяка от осите на координатната мрежа могат да бъдат задавани от приложната програма и е естествено да зависят както от големината на прозореца, така и от големината на чертожното поле.

Координатната мрежа е най-често правоъгълна и равномерна, но не е изключено използването и на неравномерни и ротационни мрежи. Съществуват следните разновидности в зависимост от видимостта на мрежата:

- *Невидима мрежа*. Координатната мрежа не е визуализирана върху екрана, но позиционирането се извършва само измежду нейните възли.
- *Видима мрежа*. Всеки от възлите, попадащи в границите на прозореца е отбелязан със светеща (с определен цвят на мрежата) точка. Задачата за позиционирането в този случай прилича много на посочването (с тази разлика, че посочването не е на обекти от модела). Приложните програмисти трябва да отчитат възможността прозорецът да бъде толкова голям, че изгледът да се покрие почти изцяло от изображението на мрежата.



Фиг. 4-4

В зависимост от това, как се интерпретират координатите на точките, получени след обръщението към локаторната функция, координатните мрежи биват:

- Мрежа, в която позиционирането може да бъде само върху възлите. Движенията на показалеца могат в този случай да бъдат скокообразни - от възел на възел, макар че това практически би нарушило обратната връзка при силно разредени възли. По-добро решение е плавно движение на показалеца, но с особено осветяване на най-близкия възел или визуализиране на координатите му. Такива мрежи обикновено могат да бъдат "изключвани", за да позволят задаване на произволни координати. Те са най-често видими, за да се прави разлика между отделните състояния.
- Мрежа, при която позиционирането може да бъде както измежду възлите, така и на определено разстояние от тях. В този случай възлите на мрежата играят ролята на притегателни центрове. Когато задаваната от локатора позиция е в обсега на даден възел (попада в кръг с център този възел и предварително фиксиран радиус), приложната програма получава координатите на възела, а в противен случай - истинското положение на локатора.

Какъв тип да бъде мрежата зависи от конкретните приложни задачи, възможностите за обратна връзка и от гъстотата на възлите.

ПОЗИЦИОНИРАНЕ ЧРЕЗ ВЪВЕЖДАНЕ НА ТЕКСТ. Задаването на двойка числа във вид на текстов низ се налага не само когато графичната система не разполага с локатор. Това е удобно да се прави и в случай, че координатната система, в която се извършва позиционирането, не може директно да се свърже с физическата координатна система на локатора. Типични са следните случаи на позициониране чрез въвеждане на текст:

- на точка в недекартови координати (напр. полярни координати);
- в относителни координати спрямо някоя характеристична точка на графичен примитив от модела;
- в координатна система, определена от елемент на изображението - например въвеждане на точка по продължението на отсечка от модела на определено разстояние от началната точка на тази отсечка;
- при използване на математически формули за пресмятане на координати;
- при използване на имена на променливи (и участието им в математически изрази), чиито стойности са предварително измерени координатни величини.

Особено важно е една приложна програма да предоставя този начин за позициониране като алтернативна възможност. Ако потребителят знае точните координати, той в повечето случаи би предпочел да ги въведе от клавиатурата, вместо да търси желаната позиция с локатора.

ПОЗИЦИОНИРАНЕ ЧРЕЗ УСТРОЙСТВА ЗА ИЗБОР. Типичен пример за това е използването на някои от специалните клавиши на клавиатурата за управление на движението на графичния показалец. Четири (по един за основните посоки) или осем (и за придвижване по диагонал) клавиша са необходими за установяването на показалеца в позицията, която потребителят желае да въведе. Графичният вход завършва (при режим ЗАЯВКА) когато потребителят натисне един от няколкото специални клавиша, отбелязващи края на позиционирането. Например за да се имитира напълно работа с мишка с два бутона са необходими 3 допълнителни клавиша, защото това е броят на различните комбинации от натиснати бутони на мишката.

При използване на устройства за избор е необходимо потребителят да може да управлява скоростта на движението на показалеца, а от там и точността на позиционирането. Ето няколко често използвани начина за това:

- два допълнителни клавиша (например "+" и "-") увеличават или намаляват два пъти стъпката, с която се извършва движението;
- допълнителен клавиш (например Shift, Control) в комбинация с някой

от клавишите за придвижване предизвиква по-бързо движение в съответната посока;

- задържането на някой от клавишите за придвижване увеличава стъпката на придвижване съобразно продължителността на задържането.

Един от основните недостатъци на този вид позициониране е сравнително бавното придвижване до желаната точка (увеличаването на скоростта е за сметка на точността). Поради тази причина често се използва *циклично преместване*. Когато показалецът достигне някоя от границите на екрана, той продължава движението си в същата посока, но от противоположната страна на екрана.

Придвижването може също така да се управлява с глас, с команди като НАГОРЕ, НАЛЯВО и т.н. То може да се извършва и едновременно с локатор и устройство за избор, като локаторът се използва за грубо позициониране, а точната позиция се избира с допълнителни придвижвания, които се командват от устройството за избор. Комбинирането на двете възможности не винаги е тривиално, особено при използване на локатор с абсолютни координати.

ПОЗИЦИОНИРАНЕ ЧРЕЗ ВАЛЮАТОРИ. Валюаторите могат също да се използват за позициониране, макар и това да се прави твърде рядко. За целта са необходими поне два валюатора за всяка от координатите. Допълнителен потенциометър може да управлява скоростта на движение на графичния показалец. В някои графични дисплеи (напр. Tektronix 4016) позиционирането се извършва с един вертикален и един хоризонтален ротационен потенциометър, разположени върху самата клавиатура. Позиционирането с валюатори може сравнително лесно да се приложи в тримерните графични системи.

ПОЗИЦИОНИРАНЕ ЧРЕЗ СЕЛЕКТОР (ГРАВИТАЦИОННО ПОЛЕ). Както видяхме преди, светлинното перо като типичен селектор не може да се използва директно за позициониране, защото не може да задава точка от екрана, която не е осветена. В зората на интерактивната компютърна графика, когато това устройство е било най-широко използвано, са били разработени ефективни методи за осъществяване на това. Тук ще разгледаме позициониране, за осъществяване на което се използват резултати от дейността *посочване*.

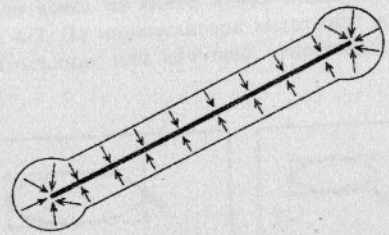
При изготвянето на всеки чертеж много често се налага една отсечка да има за крайна точка някой от върховете на фигура в създадения до момента чертеж. Ако координатите на този връх не са съхранени в някакви променливи и въведената точка не е възел от координатната мрежа, то нейното повторно интерактивно позициониране може да не е никак лесно. Читателят може да си представи каква трудност би представлявало интерактивното задаване на две концентрични окръжности, ако единственото средство, с което разполага е абсолютен локатор.

За да се реши този проблем се приема, че всички *характеристични точки* на всички елементи от изображението играят ролята на притегателни

центрове (имат *гравитационно поле*) по начин, подобен на възлите от координатната мрежа. Разликата тук е, че резултатът от позиционирането ще бъдат координатите на характеристична точка вместо директната позиция на локатора, само ако последната е достатъчно близо до точката. Както координатната мрежа, така и гравитационното поле са интерактивни похвати, разработвани в самите приложни програми, а не в базовите графични системи.

Гравитационното поле обхваща не само характеристичните точки, но и самите примитиви. Това дава възможност да се позиционира върху даден елемент - отсечка, дъга, окръжност, като позицията на локатора се проектира върху елемента, в случай че тя е достатъчно близко до него.

Гравитационното поле е дефинирано в чертожната координатна система, което означава, че привличането зависи от близостта на графичния показалец до графичното изображение на примитива, а не от разстоянието в потребителски координати между въведената позиция и елемента в геометричния модел. Това позволява на потребителя да управлява точността на въвеждане чрез подходящо увеличаване и намаляване на изображението (промяна на потребителския прозорец). Съществува обаче опасност гравитационните полета на различни елементи да се припокрият, ако потребителският прозорец е много голям. Това на практика се избягва от самите потребители, които почти никога не позиционират върху изображение в много дребен мащаб.



Фиг. 4-5

Почти всички интерактивни графични програми използват този интерактивен похват. Разбира се, той има различни нюанси, свързани например с това, кои точки да се считат за характеристични. Възникват следните няколко въпроса:

- Трябва ли да считаме пресечните точки на примитивите за характеристични?
- Ако центровете на окръжностите и дъгите не се визуализират от приложната програма, трябва ли те да се считат за характеристични? Ако пък те не са характеристични точки, как ще решим проблема с концентричните окръжности?
- Трябва ли да считаме центъра на една окръжност, която не попада в текущия прозорец, за характеристична точка, ако самата точка попада в него?
- Трябва ли да считаме управляващите точки на кривите за характеристични? Отговора на този последен въпрос можем да търсим едва когато

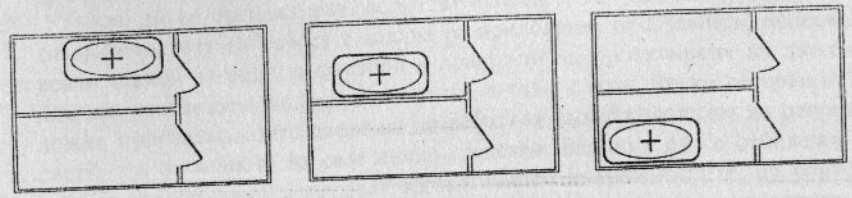
си изясним как се представят равнинните криви в един геометричен модел, на което е посветена част от шеста глава.

ДИНАМИЧНО МАНИПУЛИРАНЕ. Представените по-горе интерактивни похвати и особености на *позиционирането* са приложими при задаване на точка в потребителското пространство. За нуждите на много приложни графични програми е необходимо въвеждането не само на точка, а на позицията и на по-крупни графични обекти, както и преместването или модифицирането им чрез репозициониране на техни характеристични точки. Във всеки от тези случаи е за предпочитане обратната връзка при действието да не е само графичен показалец, а динамичното изменение на положението и/или формата на обекта в съответствие с движението на локатора. Този тип обратна връзка е известен като *динамично манипулиране*. За осъществяването ѝ се използва най-вече *векторно прерисуване*, както показахме, че се прави това за графичния показалец. Ако обектът е прекалено сложен за динамично прерисуване, то е възможно да се прерисува векторно някаква съществена част от него - негова скица. Ето няколко конкретни примера:

1. *"Влачене"* или *"буксировка"* (drag-and-drop): Използва се при задаването на преместването на един обект в ново положение. След като обектът е избран, натискане на бутон на локатора отбелязва началото на *влачението*. Когато локаторът се използва и за дейността *посочване* (което ще разгледаме малко по-късно), това натискане на бутон активира и избирането на обекта, който ще се премества. Докато бутонът е натиснат, всяко преместване на локатора води и до съответното преместване на избрания обект. Щом бутонът се освободи, действието завършва и обектът остава в новото положение (вж. фиг. 4-6).

```
. . .
SampleLocator(LOC, &Xold, &Yold, &status);
if (status == LEFTBUTTON_DOWN) {
   SetWritingMode(XOR);
   NormToUser(Xold, Yold, &Xu, &Yu);
   DrawObject(Xu,Yu);
   while (status == LEFTBUTTON_DOWN) {
      SampleLocator(LOC, &Ynew, &Ynew, &status);
      if (Xnew!=Xold || Ynew!=Yold) {
         DrawObject(Xu,Yu);
         Xold=Xnew; Yold=Ynew;
         NormToUser(Xold, Yold, &Xu, &Yu);
         DrawObject(Xu,Yu);
      }
   }
   DrawObject(Xu,Yu);
   SetWritingMode(REPLACE);
}
. . .
```

При реализирането на този интерактивен похват в показания фрагмент сме предположили, че изтриването и постоянното включване в изображението (в новото местоположение) се осъществяват от други функции на приложната програма. Показаната част обслужва само обратната връзка.



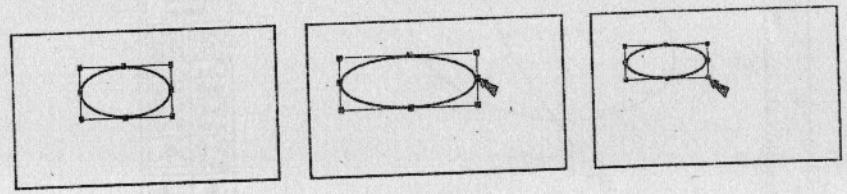
Фиг. 4-6

2. *"Динамично въртене"* и *"динамично мащабиране"*: Тези два примера на динамично манипулиране се реализират по същия начин като *влачението* по схемата:

натискане на бутон ⇒ движение ⇒ освобождаване на бутон

Разликата е, че след като обектът е посочен е необходимо потребителят да зададе и центъра на ротация или хомотетия. Възможно е да се приеме по подразбиране, че този център съвпада с центъра на тежестта на обекта, което би улеснило графичния вход.

3. ***Модифициране чрез манипулатори***: Удобен начин за модифициране на един обект е чрез репозициониране на определени точки от него (или от неговата правоъгълна обвивка), наречени *манипулатори* (handles). Осемте точки от правоъгълната обвивка дават възможност обектът да бъде мащабиран едновременно или поотделно по всяка от осите и във всяка нейна посока, както това е показано на фиг. 4-7. За манипулатори могат още да бъдат избрани върховете на един многоъгълник или начупена, управляващите точки на една крива и др.



Фиг. 4-7

4.2.2 Избор от възможности

Изборът от възможности е дейност, която се налага при работа с всички по-големи програмни пакети. В графичните програми тази дейност може многократно да се улесни най-вече чрез подходящо моделиране на това абстрактно устройство и чрез ефективна обратна връзка.

Тук е необходимо да поясним разликата между *избор от възможности* и *посочване*, които като интерактивни дейности си приличат. *Посочването* се отнася само до обектите, които са основният предмет на създаване и работа на приложната програма - елементите на нейния геометричен модел. Ако това е програма за автоматизиране на инженерното чертане, то посочването се из-

вършва измежду елементите на чертежа. *Изборът от възможности* може също да се извърши графично, когато тези възможности са визуализирани върху екрана на дисплея във вид на ключови думи или графични символи (икони). От потребителска гледна точка тези специални символи, макар и част от изображението, са визуализирани само за да се осъществи избор на някой от тях графично и вън от този избор те нямат смисъл за приложната програма. От гледна точка на приложния програмист посочването е свързано с модела, който се създава и обработва, докато изборът от възможности е само част от интерфейса на програмата.

Тъй като устройствата за избор са доста различни по тип, приложните програми най-често трябва да укажат какъв клас (от тези, поддържани от графичната система) устройства за избор желаят да използват. Това се прави при инициализацията на устройството където се задават параметри като:

- идентификаторът на устройството;
- класът устройство (функционална клавиатура, бутони на мишка, меню върху екрана, бутони върху екрана и др.),
- началната стойност на избора;
- имената на всички възможности за избор;
- границите на полето за обратна връзка в нормирани координати и др.

```
void InitChoice(ChoiceID, type, init_value, Items)
```

Функционалната клавиатура има неудобството, че разполага с фиксиран брой клавиши за избор, който почти винаги е недостатъчен за нуждите на приложната програма. Поради тази причина се използват редица прийоми за изкуствено повишаване на броя на избираните възможности, най-често предоставяни от самата графична система:

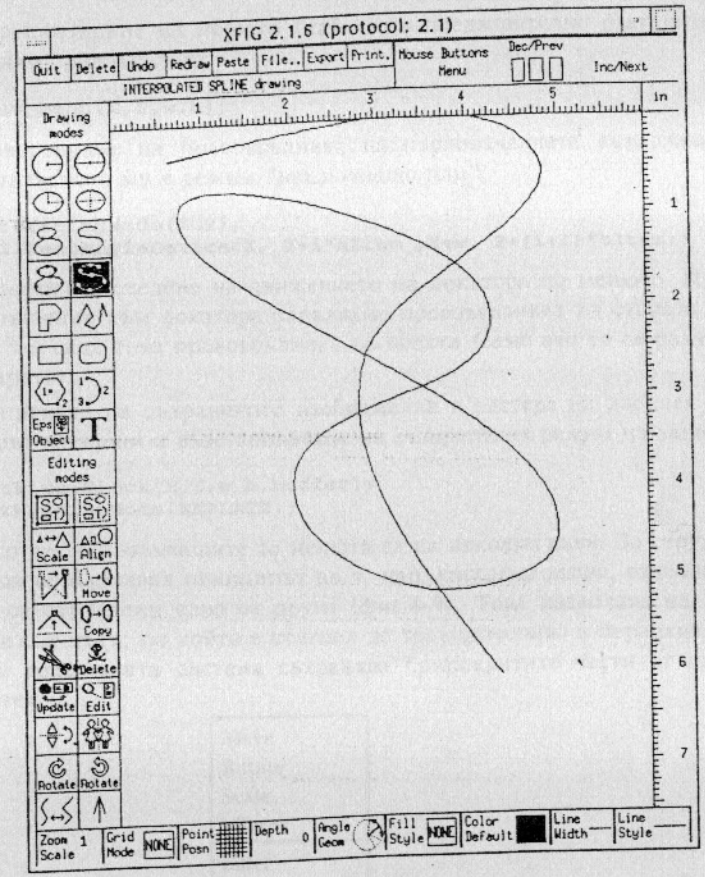
- комбиниране на функционален клавиш с модифициращ клавиш на традиционната клавиатура: `Shift-F1, Alt-PF2`;
- комбиниране на модифициращ клавиш на клавиатурата с натиснат бутон на локатора: `Shift-LEFT_BUTTON, Control-RIGHT_BUTTON`;
- последователно натискане на един и същ клавиш или бутон в определен, достатъчно кратък интервал от време: `DOUBLECLICKBUTTON`.

Всички тези начини се нуждаят от добра стандартизация (еднотипно значение в различни подобни приложни програми) и от необходимостта потребителят да запамети предлаганите комбинации. Много по-удобно е всички възможности да са визуализирани по подходящ начин в момента на избора. В повечето съвременни системи изборът се реализира чрез графично изобразяване на всички възможности в малка правоъгълна област от екрана, наречена *графично меню*.

ИЗБОР ОТ МЕНЮ. Този интерактивен похват осигурява съвместното използване на едно и също физическо устройство за локатор и за извършване избор. Това е извънредно удобно за приложни програми, в които изборът е съчетан с позициониране, така че голямата част от действията на потребителя

са съсредоточени върху едно единствено устройство.

Как да се организират, подредят и представят елементите на едно меню (възможностите за избор) е задача на приложния програмист, решаването на която зависи от редица фактори и принципи на проектиране на диалога. Тях ние ще разгледаме подробно по-късно в тази глава. Често се срещат и приложни програми, които дават на потребителя възможност сам да реорганизира системата от менюта за свое лично удобство. Важно е да се отбележи, че менюта е удачно да се използват тогава, когато възможностите, от които се избира, са сравнително постоянни (на брой и на вид), т.е съществени изменения в тях не настъпват в процеса на работата на приложната програма.



Фиг. 4-8

Много графични системи предлагат средства за организиране на избор от меню, но не са изключени и случаи, в които това се прави изцяло от приложните програми чрез подходяща визуализация и обратна връзка. Менютата биват няколко вида, всеки от които предполага различно обслужване:

1. *Статично меню*: Това са набор от възможности, изборът измежду които се налага сравнително често, които не се променят по време на работата на приложната програма и имат смисъл на почти всички етапи от нейното изпълнение. Типичен е примерът на потвърждаване или отказване от последната изпълнена операция на приложната програма.

Когато споменатите условия са налице, възможностите могат да бъдат организирани в меню, което е постоянно достъпно за потребителя по един от следните няколко начина:

- възможностите са отпечатани (например върху лист) и постоянно разположени върху повърхността на таблета, както това е показано на фиг. 4-3. Отделните възможности се избират чрез посочване с перото на таблета към желаната възможност. Този начин позволява голям брой възможности да се предоставят едновременно за избор (например командите на една приложна графична програма), без това да заема ценно място от графичния дисплей.
- менюто е разположено върху допълнителен екран (при дисплеи с два екрана например в системата Intergraph) или върху по-малък допълнителен дисплей.
- менюто е разположено в специално отделена област на основния дисплей, която не се използва за позициониране. Посочването с локатора в част от тази област води до избора на разположената там възможност (фиг. 4-8).
- менюто се намира в долната част на екрана, непосредствено над функционалната част от традиционната клавиатура. Изборът се осъществява чрез натискане на съответния функционален клавиш.
- менюто се намира в някаква част на екрана и схематично изобразява мишката и различните функции на нейните бутони (като в програмата XFIG - вж. горния десен ъгъл на фиг. 4-8).

В последните два случая менюто само подсказва как да се осъществи избора, без да дава възможност за директен избор чрез посочване върху екрана. По принцип статичните менюта са само на едно ниво, т.е. всички представени в тях елементи са възможности и не предлагат от своя страна избор от друго меню. Съществуват и програми със статични йерархични менюта, като всяко ново ниво от йерархията се разполага на мястото на предишното. Това създава опасност потребителят да загуби представа в кое точно ниво на йерархията се намира в определен момент. За да се избегне това, пътят от корена до текущото меню може се изписва в отделно поле от екрана.

Програмно изборът от графично меню се осъществява достатъчно просто: когато локаторът зададе позиция, която попада в правоъгълника на менюто, се проверява в кой от правоъгълниците на възможностите точно се намира тя. Тези проверки за правоъгълни области са тривиални и следователно сравнително бързи.

2. ***Появяващо се меню*** (pop-up menu): Това е меню, което се появява върху екрана само по специална заявка на потребителя или когато приложната програма очаква от него избор, за да може да продължи. Менюто се появява най-често на мястото, където в момента се намира графичният показалец и при завършване на избора припокритата от него част от изображението се възстановява. Този тип меню спестява място от екрана на дисплея. Осъществяването му е много удобно в съвременните растерни дисплеи и се извършва по следния начин:

а/Съхраняване на правоъгълника от растера в някакъв временен буфер, в който ще се изобрази менюто:

```
GetPixelBlock(X,Y,w,h,&buffer);
```

б/Визуализиране на менюто (най-често предварително растеризирано) в същия правоъгълник:

```
Draw_Menu(X,Y,w,h);
```

в/Инвертиране на правоъгълника на първоначалната възможност, чрез запълването му в режим "изключващо или";

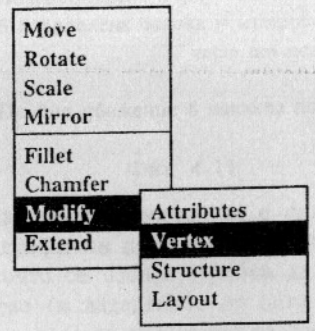
```
SetWritingMode(XOR);
FillRectangleDevice(X, Y+i*hItem ,X+w, Y+(i+1)*hItem);
```

г/Динамично следене на движението на локатора по менюто. При всяко обръщение към локатора запълваме правоъгълника на старата възможност и след това правоъгълника на новата (само ако тя се различава от старата);

д/Записване на съхраненото изображение в растера на дисплея щом изборът завърши и възстановяване на стандартния режим на записване:

```
PutPixelBlock(X,Y,w,h,buffer);
SetWritingMode(REPLACE);
```

Много често появяващите се менюта са на няколко нива. За структурирането може да се използва принципът на т. нар. *каскадно меню*, отделните нива на което са отместени едно от друго (фиг.4-9). Това позволява на потребителя да вижда пътя, по който е стигнал до текущото ниво в йерархията. В този случай графичната система съхранява припокритите части от изображението в стек.

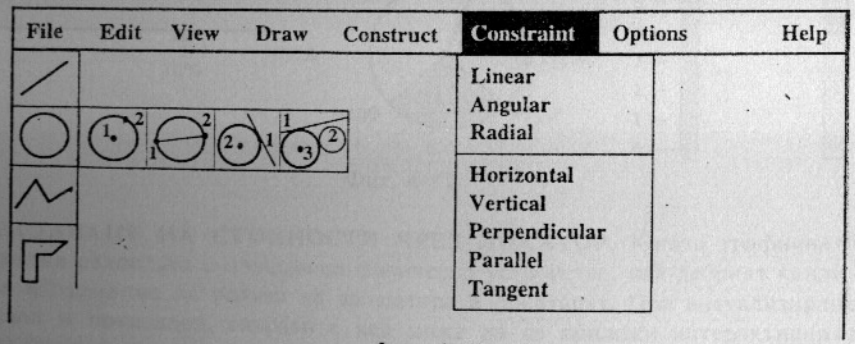


Фиг. 4-9

3. *Падащо и разгъващо се меню* (pull-down, pull-out menu): Това са менюта, които се състоят от две части: едната е статична (разположена постоянно в горната част на екрана в случай на падащо меню или отстрани при разгъващите се менюта), а другата е появяваща се. Всеки елемент от статичната част не е възможност, а представя меню, от което могат да се избират възможности. Това е прост, но ефективен начин за структуриране на командите в групи, които потребителят може лесно да запомни и използва. Мястото, заемано от този вид меню, е сравнително малко. Има два вида меню съобразно начина му на активиране:

- С *явно посочване*: потребителят премества графичния показалец до елемент от статичното меню и натиска бутон на локатора, което води до появяване на групата възможности, "прикачена" за този елемент. Докато бутонът е натиснат, потребителят може да се избира от представените възможности. При отпускане на бутона подменюто изчезва. Друг вариант на същия тип меню: При кратко натискане на бутон върху елемент от статичната част подменюто се появява и се визуализира за избор дотогава, докато потребителят натисне втори път бутон в неговите граници (избрана е възможност) или извън тях (изборът е прекратен).
- При преместване на показалеца в областта на статичната част (без да се натиска бутон), съответното подменю се появява, а изчезва или когато показалецът напусне неговата област, или когато е избрана възможност чрез натискане на бутон.

Най-често в разгъващите се менюта отделните възможности са изобразени като графични символи (*икони*), докато падащите имат текстов вид. Използването на икони спестява място върху дисплея при всички случаи, в които е лесно да се обясни графично възможността за избор, която се предлага. Типичен пример са начините за конструиране на примитиви, показани в разгъващото се меню на фиг. 4-10. За всеки от тях би било необходимо твърде дълго текстово описание.



Фиг. 4-10

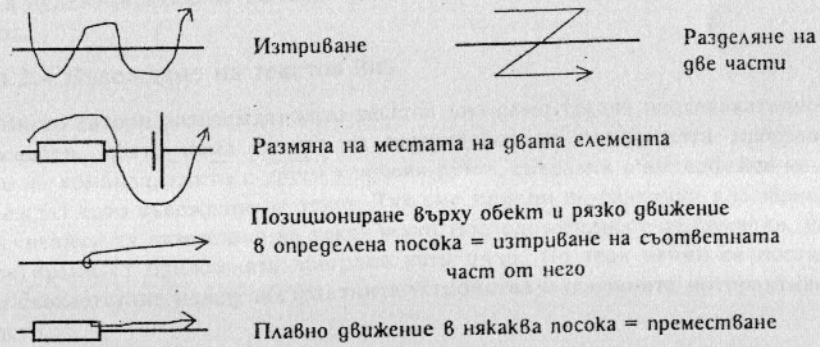
За улесняване на взаимодействието с потребителя се препоръчва появяващата се част от тези менюта да е само на едно ниво. В много програми някои от падащите менюта са на две нива, което често затруднява и забавя избора. При твърде сложна система от менюта е желателно потребителят да може сам да ги аранжира или да може да избира между работа с пълно меню и кратко меню (от най-нужните и често използвани възможности).

ИЗБОР ЧРЕЗ ВЪВЕЖДАНЕ НА ТЕКСТ. Този начин се използва много често в приложни програми, работата с които се управлява от т. нар. *команден език* (по подобие на класическия потребителски интерфейс на една операционна система). Въвеждането на името на една команда като текстов низ илюстрира този вид избор. Макар и съвсем не графичен, той може да бъде удобен в случай, че потребителят е принуден в определен период да работи с клавиатурата. Например ако потребителят трябва да разположи текстов низ в определена позиция от своя чертеж, той може да въведе от клавиатурата текста:

```
Command> PLACE TEXT "Ground Floor Elevation = 80mm"  300,300
Command>
```

вместо да избере с локатора командата от някакво меню, да остави локатора, за да въведе желания текст, а после отново да използва локатора за разполагането на този текст.

ИЗБОР ЧРЕЗ ДВИЖЕНИЯ НА ЛОКАТОРА. Друг начин за използване на локатора (или селектора) за осъществяване на избор е чрез разпознаване на определени движения с него. Това, разбира се, е възможно само за устройства, позволяващи непрекъснато позициониране (таблет, мишка, светлинно перо).



Фиг. 4-11

Този начин за задаване на команди се е прилагал много удачно още в първите интерактивни графични програми. Допълнително негово предимство е, че ако командата, която се избира, трябва да се изпълни върху някакъв обект, същото устройство (и зададените от него позиции по време на движението) може да се използва за посочване на обекта. Движенията трябва да

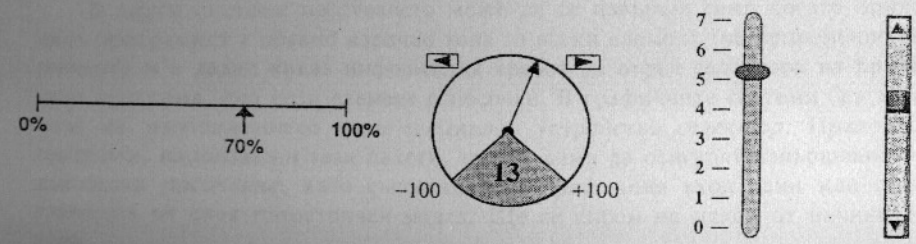
бъдат ограничен брой и да съответстват на естествени движения, характерни за конкретната приложна дейност. Например удобно е да се използват във вид на движения знаците, които употребяват коректорите на текст.

Въпреки очевидните предимства прилагането на този начин за избор е сравнително ограничено. Графичните системи не предлагат готови средства за реализирането му, поради което всеки приложен програмист трябва изцяло да разработи анализирането на движенията. Тази задача е част от общата задача за *разпознаване на образи*, на която тук няма да се спираме.

4.2.3 Задаване на стойност от непрекъснато множество

Тази дейност е много подобна на позиционирането с тази разлика, че е необходимо да се въведе линейна стойност. Координатната система е определена от границите на областта от допустими стойности. Както и при позиционирането, от особено значение за извършването ѝ е наличието и качеството на обратната връзка.

ОБРАТНА ВРЪЗКА. Минималната обратна връзка е визуализирането на текущата стойност на валюатора. Ако потребителят знае новата стойност, която иска да зададе, най-удачният начин е въвеждането ѝ като текстов низ. В много случаи (по-често дори отколкото при позициониране) потребителят желае само да промени текущата стойност като я намали или увеличи. Ако величината директно влияе на визуализираните обекти, тяхното динамично изменение е най-добрата обратна връзка. При използване на потенциометър за задаване на ъгъла на въртене на даден графичен обект относно определен център самото въртене на обекта е достатъчно ефективна обратна връзка. В повечето от останалите случаи изобразяването на линейна или кръгова скала в определена област от екрана е за предпочитане.



Фиг. 4-12

ЗАДАВАНЕ НА СТОЙНОСТИ ЧРЕЗ ЛОКАТОР. Когато графичната система не разполага с подходящо физическо устройство, най-добрият кандидат за изпълнение на ролята на валюатора е локаторът. При визуализиране на скала и показалец, свързан с нея може да се приложи интерактивният похват *динамично преместване* на този показалец чрез локатора. Това се извършва по представения по-горе начин: позициониране върху показалеца на скалата, натискане на бутон и преместване при задържан бутон до новото положение, освобождаване на бутона.

Приложният програмист може сам да избере вида на скалата - *кръгова* или *линейна*. Кръговата скала е най-удобна при задаване на периодични стойности, но може да бъде използвана и в други случаи, особено когато е желателна по-голяма точност на задаване. По-голямата точност се постига чрез движения на графичния показалец далече от центъра на скалата, които се предават като много по-малки изменения на ъгъла, който всъщност се отчита. Линейните скали са по-удобни за визуално анализиране и съпоставяне на стойности в случай, че се работи с няколко валюатора едновременно. Посоките на увеличаване и намаляване на стойността във всички скали трябва да са еднакви. Стъпката на промяната може да съответства на движението на графичния показалец, но може да бъде и по-малка при линейни скали с недостатъчна дължина.

Освен скалата и показалеца, приложната програма може да визуализира и графични бутони, позиционирането на локатора върху които да води до увеличаване и намаляване с фиксирана стъпка. Всички скали са по принцип с линейно изменение, но не са изключени логаритмични и експоненциални скали, удобни за някои приложения като акустика и обработка на сигнали.

ЗАДАВАНЕ НА СТОЙНОСТИ ЧРЕЗ УСТРОЙСТВА ЗА ИЗБОР. Макар и рядко, непрекъснати стойности могат да се задават и чрез дискретни устройства. Определяне на няколко клавиша за увеличаване, намаляване и управление на стъпката, с която става това, са достатъчни за извършването на тази дейност. За много от величините е желателно да се даде лесен достъп до някои ключови стойности от скалата, като за това се използват определени клавиши или бутони. Типичен пример е установяването на симетричен валюатор в нулево положение по подобие на баланс-регулатора на стерео усилвателите.

4.2.4 Въвеждане на текстов низ

Много автори разглеждат като текстов низ само такава последователност от символи, която няма смисъл за интерфейса на приложната програма. Името на команда, както и други ключови думи, свързани с интерфейса не се разглеждат като въвеждане на текст. Тук сме приели по-различна класификация, считайки за въвеждане на текст всяка последователност от символи, интерпретирани от приложната програма като цяло. По този начин се постига пълно съответствие между абстрактните устройства и основните интерактивни дейности.

ВЪВЕЖДАНЕ НА ТЕКСТ ЧРЕЗ ЛОКАТОР. Въвеждането на текст чрез локатор е оправдано тогава, когато почти целият графичен вход се извършва чрез локатора. Този начин е приемлив при използване на таблет и фиксирано върху неговата повърхност меню. Част от менюто може да е отделена само за латинските букви, цифри, както и за символи от други азбуки: букви с ударения, гръцки букви, кирилица, технически символи и др. Стандартните клавиатури нямат на разположение специалните символи, които се използват в

техническото чертане. За задаването на текст, отбелязващ определена размерност в даден чертеж би било по-удобно да се имитира числова клавиатура с добавени символи за диаметър, градус и допуск, което би улеснило много анотирането на чертежа.

РАЗПОЗНАВАНЕ НА СИМВОЛИ. Този начин се използва в *компютърите-бележници*, входът при които е основан на разпознаването на графично изписвани от потребителя символи. За разлика от разпознаването на сканиран текст, това е по-лесно, защото може да се извършва по определени образци от поредица движения с определени посоки. Разпознаването на ръкописен текст е много по-сложно от това на ръкописно изписани печатни букви, тъй като във втория случай образците на движенията са по-прости и всяка от буквите е отделена от останалите. За съжаление писането с печатни букви е доста по-бавно от писането с ръкописни букви.

4.2.5 Посочване

Посочване се налага при всяка модификация на обект от модела, който потребителят на приложна програма създава интерактивно. Например за да се премести една отсечка в ново положение е необходимо потребителят да разполага с удобни средства, за да укаже на програмата коя именно отсечка желае да се премести. Някои графични системи обслужват дейността *посочване* на ниво графични сегменти. Всеки сегмент може да бъде посочен от потребителя, а като резултат графичната система изпраща на приложната програма идентификатора на този сегмент.

```
PostSegment(sample);
. . .
RequestPick(PickID, &segmentId);
UnpostSegment(segmentId);
```

В други системи посочването може да се извърши само когато приложният програмист е обявил изрично това за всеки елемент (не непременно само сегмент) и е задал каква информация трябва да върне селектора на приложната програма, ако този елемент е посочен. В графичните системи без запомняне на изображението няма специално устройство *селектор*. Приложните програми, използващи тези пакети, трябва сами да осигурят извършването на дейността *посочване*, като съпоставят на графичния вход един или повече елементи от своя геометричен модел. Ще се спрем на някои от начините за това.

ПОСОЧВАНЕ ПО ИМЕ. Това е най-простият начин за посочване, при който всеки създаван графичен елемент (най-често сегмент, а не примитив) получава име или автоматично, или задавано от потребителя, което той може да използва, за да го идентифицира. В следните няколко случая посочването по име може да бъде желателно:

- Когато потребителят използва уникални обекти, всеки от които има смислено за потребителя име.

- Когато графичното посочване е трудно поради многото елементи в изображението, за регенерирането на което в увеличен вид е необходимо значително време.

Идентифицирането по име има допълнителното предимство, че при подходящо именоване използването на символи за обобщаване (wild-card characters) би позволило едновременното посочване на група от обекти. При неточно въвеждане програмата може да предложи избор от най-близките до въведеното име възможности. Неточното въвеждане може да се открие по замяна на близко разположени върху клавиатурата букви, допълнително натискане на съседни клавиши и др. При всеки от тези случаи е необходима обратна връзка с потребителя, за да може той да коригира всяко неточно подразбиране. Този метод може да се използва и при наличието на устройства за гласова връзка, като така се избягва необходимостта да се набират имената от клавиатура.

ПОСОЧВАНЕ С ЛОКАТОР. В един реален чертеж броят на примитивите е сравнително голям и тяхното удачно именоване е практически невъзможно. Посочването им с локатора е най-удобно за потребителя. Това е най-разпространеният начин за посочване, който позволява лесно графично идентифициране на обектите при използването винаги на едно и също устройство. По същество задачата се решава в следната последователност:

1. Въвеждане от потребителя на координатите на точка чрез локатора;
2. Преобразуване на тези координати в потребителски;
3. Търсене: обхождане на елементите от приложния модел и намиране на този примитив в него, който е най-близко до въведената точка;
4. Визуализиране на резултата от търсенето.

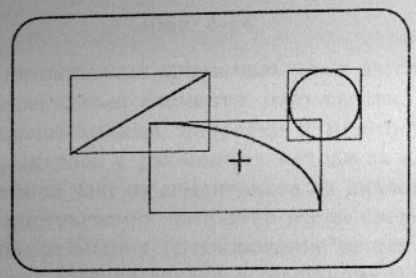
Няколкото начина за осъществяването на посочването на ниво графични примитиви се различават по реализирането на третата и четвъртата стъпки в горната последователност. Търсенето може да бъде:

- *По характеристичните точки на примитивите.* За такива се считат краищата на отсечките, краищата и центровете на дъгите, центровете на окръжности и елипси и др. Търсенето се свежда до намиране на въведената точка и характеристичните точки с най-късото разстояние между тях. Проблемите тук са свързани най-вече с факта, че една точка е характеристична на повече от един примитив; един примитив може да е видим, а неговите характеристични точки да не попадат в изгледа и т.н.
- *По правоъгълните обвивки на примитивите.* Търсенето се свежда до намирането на правоъгълник, във вътрешността на който попада въведената точка. В повечето случаи правоъгълните обвивки се припокриват (фиг. 4-13).

```
for (i=Nprimitives-1; i>=0; i--) {
  PrimitiveRectangleHull(i,&X1,&Y1,&X2,&Y2);
  if (X1<=X && Y1<=X && X<=X2 && Y<=Y2) break;
}
```

В някои програми е въведен приоритет, като моделът се преглежда в определен ред, най-често от последния създаден примитив към първия. Удачно е да се търси само между примитивите, които не са изцяло извън правоъгълника на прозореца.

- *По типове примитиви*. Търсенето се извършва само между определен тип (напр. измежду всички окръжности) и се отчита разстоянието до самите елементи, а не до техните характеристични точки.
- *Глобално търсене*. Търси се минималното разстояние на въведената точка до всеки примитив. То може да води до забавяне при сложни модели. Затова е удачно да се търси само измежду примитивите, в чиято правоъгълна обвивка попада въведената с локатора точка.



Фиг. 4-13

Изброените начини за търсене могат да се реализират и едновременно. Когато най-простият не даде еднозначен резултат, да се прибегне към следващия по сложност и т.н. докато се стигне до глобално търсене.

Обратната връзка - визуализирането на резултата от търсенето - е важна за гарантирането на правилен избор. Тя може да бъде или оцветяване на посочения примитив в някакъв особен цвят, или мигане на примитива (непрекъснатото му прерисуване в режим на записване хог през подходящ интервал от време) дотогава, докато потребителят предприеме следващото действие. При всички случаи потребителската програма трябва да се грижи да възстанови нормалния цвят на примитива щом посочването му завърши.

ПОСОЧВАНЕ В ЙЕРАРХИЧНИ СТРУКТУРИ. Някои от начините за посочване на примитиви могат да доведат до двусмислие, но това двусмислие може да бъде преодоляно без да се изисква намеса на потребителя. Когато обектите от модела са йерархично структурирани е необходима допълнителна информация от потребителя кое ниво от йерархията иска да идентифицира. Ако нивата на йерархията са фиксирани, например изображението се състои от сегменти, всеки от които се състои само от примитиви, то е необходимо потребителят да зададе само дали идентифицира сегмент или примитив. Това може дори да е ясно от контекста на командата, за нуждите на която се извършва посочването. В общия случай е достатъчен един допълнителен параметър, който да показва в кое ниво от йерархията се търси в момента. Например потребителят може да зададе, че ще извършва идентификация само на между примитиви.

Друга възможност е нивото в йерархията да се задава едновременно с посочването. Един начин е чрез няколко последователни натискания на бутон на локатора. В някои текстови редактори например посочването в йерархичната структура: "символ - дума - изречение - абзац", която има фиксирана дълбочина, се извършва така:

1. посочване с едно натискане на бутон - посоченият символ;
2. посочване с две натискания - думата, към която символът принадлежи;
3. посочване с три натискания - изречението, към което символът принадлежи;
4. посочване с четири натискания - абзацът, в който се намира символът.

При йерархични сегменти с произволна дълбочина на влагане потребителят трябва да има средства, които да му позволяват да се движи по йерархията. Това могат да бъдат например два допълнителни клавиша НАГОРЕ (към по-общия обект) и НАДОЛУ (към примитива).

Още по-сложен е случаят, в който един примитив може да принадлежи едновременно на два сегмента. Повечето модели избягват предоставянето на тази възможност, тъй като тя води до усложняване на много от операциите с тях. В някои случаи обаче това е необходимо - при т.нар. *сегментни мрежи* - и тогава резултатът от посочването е специален идентификатор, зададен от приложния програмист. Ние ще разгледаме по-подробно този случай във връзка с йерархичните геометрични модели в пета глава.

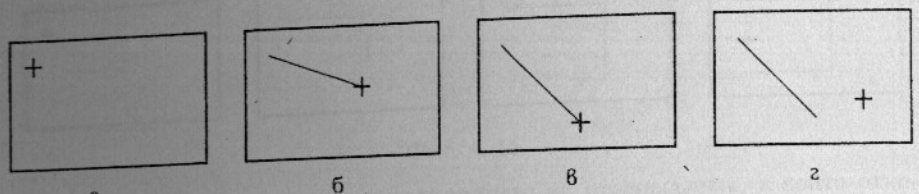
ДИНАМИЧНА НАВИГАЦИЯ. Ще поясним този начин за посочване с един пример. Когато показалецът, управляван чрез локатора се доближи до даден обект, приложната програма го оцветява по специален начин, за да покаже кой обект би бил избран, ако потребителят натисне в този момент бутон на локатора. Динамичната навигация е общ интерактивен принцип, отразяващ необходимостта от отгатване намеренията на потребителя. Графичните програми, построени на този принцип (например I-DEAS на SDRC) работят на много мощни компютърни системи.

4.2.6 Задаване на поредица от позиции

В тази част ще се спрем на някои интерактивни похвати, които обслужват задаването на поредица от точки. Тази дейност в някои системи се осъществява от отделно абстрактно устройство, а в други - от самите приложни програми. Типичен пример за тази дейност е въвеждането на върховете на многоъгълник.

РАЗТЕГАТЕЛЕН ПРИМИТИВ. При задаване на една отсечка обратната връзка за позиционирането на вторя връх може да не е само графичният показалец, но и следяща движението му отсечка. Тази отсечка наричаме *разтегателна*. Последователността от действията на потребителя е най-често следната:

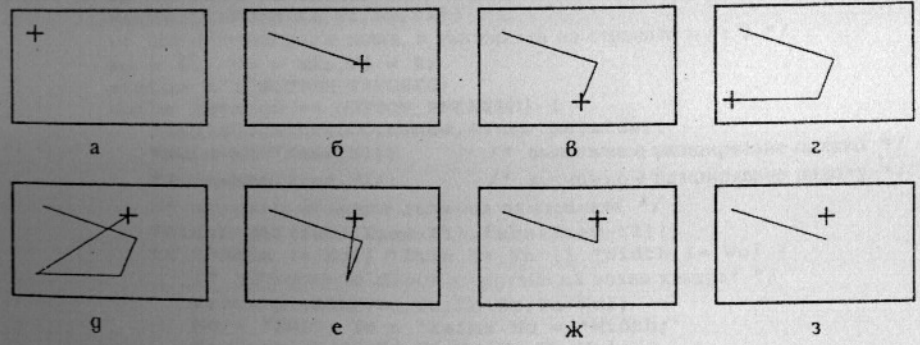
1.Натискане на бутон в началната точка на отсечката (фиг. 4-14а);
2.Докато бутонът е натиснат движението на локатора се следи от *разтегателната отсечка* (rubber line) - виж фиг. 4-14б,в;
3.След освобождаване на бутона отсечката се фиксира и позиционирането на втората точка завършва (фиг. 4-14г).



Фиг. 4-14

Принципът на *разтегателния примитив* може да се използва при задаването на различни геометрични елементи: окръжности, елипси, дъги от окръжности и елипси, правоъгълници, начупени и многоъгълници. Начинът за задаването им е пряко свързан с различните методи за конструиране на графични примитиви, които са част от семантиката на диалога.

При задаване на многоъгълник например всеки следващ връх се свързва динамично с отсечка до предишния (с изключение на случая на позициониране на първия връх). Тази отсечка заедно с графичния показалец следят движението на локатора докато положението на новия връх се фиксира. При задаване на начупена и многоъгълник е по-удобно вторият и следващите върхове да се задават не при отпускане на бутон, а при натискането му. Краят на въвеждането на върхове може да се отбелязва по няколко начина: с двойно натискане на бутона на локатора; с натискане на друг бутон на локатора; с натискане на специален графично изобразен бутон; с позициониране в началната точка за конструиране на затворен многоъгълник и др. Много важно изискване при задаване на многоъгълници е потребителят да може във всеки момент да се откаже от последния въведен връх.



Фиг. 4-15

Повтарянето на това отказване е желателно да може да продължава по

обратния ред на въвеждането на върховете до връщане към позициониране на първия връх. На фиг. 4-15 е показана една примерна последователност от действия, които илюстрират казаното:

1. Позициониране на първия връх (фиг. 4-15а)
2. Задаване на следващите върхове чрез разтегателна отсечка (фиг. 4-15б,в,г)
3. Отказване от последния въведен връх (фиг. 4-15д,е)
4. Натискане клавиш за завършване на въвеждането (фиг. 4-15ж)

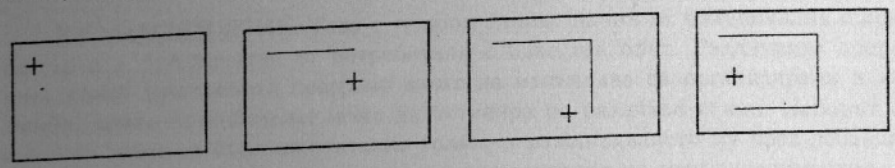
Програмно този похват се реализира подобно на графичния показалец. Ще покажем фрагмент, с които се осъществява *разтегателна отсечка*. След завършване на въвеждането последната визуализирана отсечка съответства на положението на локатора, в което бутонът е бил отпуснат.

```
SampleLocator(LOC, &Xold, &Yold, &status);
if (status == BUTTON_PRESSED) {
   /* въведена е началната точка */
   SetWritingMode(XOR);
   DrawCursor(Xold, Yold);
   /* първата разтегателна отсечка е само точка */
   MoveToNORM(Xold, Yold); DrawToNORM(Xold, Yold);
   Xbeg=Xold; Ybeg=Yold;
   while (status == BUTTON_PRESSED) {
   /* докато бутонът е натиснат анализираме новата позиция на локатора */
      SampleLocator(LOC, &Ynew, &Ynew, &status);
      if (Xnew!=Xold || Ynew!=Yold) {
         /* изтриване на старата отсечка и стария курсор */
         MoveToNORM(Xbeg, Ybeg); DrawToNORM(Xold, Yold);
         DrawCursor(Xold, Yold);
         /* визуализиране на новия курсор и новата отсечка */
         Xold=Xnew; Yold=Ynew;
         DrawCursor(Xold, Yold);
         MoveToNORM(Xbeg, Ybeg); DrawToNORM(Xold, Yold);
      }
   }
   DrawCursor(Xold, Yold);
   SetWritingMode(REPLACE);
}
```

ОГРАНИЧЕНИЯ. Друг често използван интерактивен похват при въвеждането на последователност от позиции е налагането на ограничения или зависимости от предишните позиционирания от същата последователност. Един типичен пример е въвеждането на хоризонтални, вертикални и наклонени под 45 градуса отсечки. В този случай първата въведена точка налага ограничение върху входа на втората, при което само една от нейните координати се взема под внимание за чертането на примитива. Ще илюстрираме казаното с един често използван метод за конструиране.

В много приложения чертежите се състоят изключително от хоризонтални и вертикални отсечки. Да разгледаме изготвянето на един план на сграда. За начертаването на стените в него е удобно да се даде на потребителя възможност да чертае многоъгълник, чиито страни са само хоризонтални и вертикални, като всеки две съседни образуват прав ъгъл. За целта той може да

извърши действието *задаване на поредица от позиции*, върху всяка от които (с изключение на първата) е установено ограничение - запазване на правия ъгъл с предишната страна. Тук също може да се използва разтегателен примитив, които обаче се визуализира съобразно текущото ограничение.



Фиг. 4-16

Друг вид ограничение е конструирането на правоъгълник, в който отношението на ширината към височината е предварително фиксирано. Тази задача трябва да се решава много често при задаване на нов потребителски прозорец, който да запази пропорциите на текущия. И тук трябва да се използва само една от получените от локатора координати, а другата да се изчисли по даденото съотношение. При кодирането на този интерактивен похват трябва да се вземе под внимание възможността за смяна на наредбата на двойките съответни координати на въведените точки. Това води до третиране на първата въведена точка като долен ляв ъгъл на квадрата, когато координатите на втората са по-големи и съответно горен десен ъгъл, когато и двете координати на втората са по-малки от тези на първата.

Тук представяме едно друго решение за въвеждане на квадрат, при което координатите на първата точка не се фиксират при нейното въвеждане. Това позволява квадратът да бъде разтяган във всички посоки при движение навън от всяка негова страна по време на позиционирането на втория му ъгъл.

```
DigitizeSquareNORM(Xmin,Ymin,Width)
float *Xmin, *Ymin, *Width;
{float X1,Y1, Xo,Yo,Wo, Xnew,Ynew;
 int status;
   SampleLocator(LOC,&X1,&Y1,&status);
   SetWritingMode(XOR);
   RectangleNORM(X1,Y1,X1,Y1);
   /* първият квадрат е точка, а дължината на страната му е 0 */
   Xo = X1; Yo = Y1; Wo = 0;
   status = ! BUTTON_PRESSED;
   while (status == BUTTON_PRESSED) {
      SampleLocator(LOC,&Xnew,&Ynew,&status);
      *Xmin=MIN(Xnew,X1);        /* възможно е разширяване наляво */
      *Ymin=MIN(Ynew,Y1);        /* възможно е разширяване надолу */
      /* намиране на новата дължина на страната */
      *Width=MAX(fabs(Xnew-X1),fabs(Ynew-Y1));
      if (*Xmin != Xo|| *Ymin != Yo || *Width != Wo) {
         /* изтриване на стария и чертане на новия квадрат */
         RectangleNORM(Xo,Yo,Xo+Wo,Yo+Wo);
         Xo = *Xmin; Yo = *Ymin; Wo = *Width;
         RectangleNORM(Xo,Yo,Xo+Wo,Yo+Wo);
      }
   }
}
```


4.3 ПРОЕКТИРАНЕ НА ГРАФИЧНИЯ ДИАЛОГ

В тази част ще се спрем на организирането и проектирането на диалога на една интерактивна графична програма. Приложният програмист трябва да вземе пред вид както общите принципи, на които се основава ефективното взаимодействие, така и да подбере правилно типа диалог съобразно нуждите на приложението. Графичният диалог трябва да бъде проектиран така, че да отговаря на изискванията за:

- *пълнота*: да позволява изразяването на всяка идея в областта;
- *ефективност*: да може сравнително бързо да се изрази тази идея;
- *разширяемост*: да позволява добавянето на нови диалогови елементи;
- *естествена граматика*: да се подчинява на минимален брой лесни за заучаване правила.

Графичният диалог като метод за комуникация е построен по аналогия с естествените езици. Поради тази причина неговото проектиране се извършва на няколко различни нива:

- ***концептуално ниво***: проектиране на общата схема и на потребителския модел на диалога, например имитиране на реалните инструменти и методи, използвани от чертожниците, имитиране на работно място, математически методи - теоретико-множествени булеви операции и др.;
- ***семантично ниво***: избор на типовете обекти, връзките и възможните операции с тях, набора от команди за осъществяване на операциите, възможните грешки при тяхното изпълнение и как тези грешки ще се обработват;
- ***синтактично ниво***: избиране на начина, по който отделните интерактивни дейности образуват изреченията на графичния език, последователността от интерактивни дейности за изпълнение на дадена команда, определянето на глобални операции и специални режими, разпределението на екрана на дисплея за отделните полета, структурирането на диалога и т.н.;
- ***лексично ниво***: избор на набор от средства, с които се осигурява изпълнението на основните интерактивни дейности, интерактивните похвати, типа на менютата и елементите им и определянето на начините за осъществяването на обратна връзка, т.е. думите в графичния диалог.

4.3.1 Стилове в графичния диалог

При проектиране на концептуалната схема и семантиката на диалога, които са тясно свързани с конкретната приложна област, приложният програмист трябва да избере и типа на диалога. Тук ще разгледаме няколко различни стила, които се срещат не само в графичните системи.

ДИАЛОГ "ВЪПРОС-ОТГОВОР". Водещата роля в този вид диалог има приложната програма, а потребителят трябва на всяка стъпка да избира една от предоставяните му възможности. Този тип диалог не е за предпочитане в графични програми поради фиксираната последователност, грешка при която

усложнява значително работата на потребителя. За да се коригира всяка такава грешка е необходимо да се визуализира цялата поредица от въпроси и отговори, довели до всяко определено състояние. Скоростта на работа е ниска, а възможността за грешка - твърде висока.

КОМАНДЕН ЕЗИК. Това е традиционният начин за комуникация с компютър и е предпочитан от потребители с известен опит. Различните операции, които приложната програма може да изпълнява са организирани в *команди*, които потребителят може да активира по различен начин. Наборът от команди може да бъде сравнително голям, а разширяването му чрез добавяне на нови команди от потребителя е важно предимство на този диалогов стил.

Взаимодействието чрез команден език предполага наличието на различни *режими* в диалога - ситуации, в които само определено подмножество от думи на диалога са възможни като вход. Режимите не са свързани само с този стил, но те са особено важни при проектирането на командния език. При добре проектиран синтаксис на езика ефективността на използването му може да бъде много висока при сравнително кратко предварително обучение. Лесната разширяемост осигурява и възможност за настройване към конкретни потребителски нужди и изисквания чрез използването на командни процедури (user-defined macros). Задаването на команди на естествен език е много подходящ вариант на този вид диалог, което допълнително увеличава скоростта и ефективността на диалога.

СИСТЕМА ОТ МЕНЮТА. Използването на системи от менюта е типично за широк кръг от приложни програми, които не винаги са свързани с компютърната графика. Основното предимство на менютата е, че диалогът се ръководи от потребителя, който може лесно да разпознае предлаганите му възможности по набор от ключови думи или визуални изображения на определени действия. Този стил на комуникация е особено привлекателен за начинаещи потребители, които не е необходимо да запаметяват команди и могат да видят всички достъпни в определен момент възможности. Менютата позволяват и визуализиране на текущите стойности на някои параметри, смяната на които се налага сравнително рядко - тип, дебелина и цвят на линията за чертане, вида на краищата на векторите, типа на текстовия шрифт и размера му и др.

Наборът от възможности, които се предлагат във вид на менюта е сравнително ограничен и не е толкова лесно разширяем, колкото командния език. Структурирането на менютата (тяхното групиране и разделяне на появяващи се, падащи и разгъващи се) се нуждае от особено внимание.

ИКОНИ. В компютърната графика така се наричат всички изображения, които потребителят може лесно да асоциира с определено действие, свойство, обект или група от обекти. За разлика от използването на текстови пояснения *иконите* не са свързани с никой определен език и могат да заемат значително по-малко място върху дисплея. При проектиране на иконите за една приложна програма се използват няколко различни стратегии:

- *образи на използваните средства в реалния свят*: лупа за действието "увеличение", молив за действието "чертане", ножица за действието "отрязване" и др.
- *образите на обекта преди и след действието*: при мащабиране, ротация, заобляне на връх, скосяване на връх и др.
- *абстрактни образи или езикови аналогии*: изобразяване на ботуш за обозначаване на действието презареждане (reboot), пътния знак стоп, кошче за боклук и др.
- *композиции от икони*: в системата Macintosh например са избрани определени икони за файл, програма, текст, графика, които при смислено композиране могат да обозначават "текстов файл", "текстообработваща програма", чертеж, графична програма.

Важно изискване при проектирането на икони е те да са разпознаваеми (лесно да се свързват с представяната от тях концепция), да могат да се запомнят и да могат да се отличават една от друга.

ДИРЕКТНО МАНИПУЛИРАНЕ. Името на този стил не бива да се бърка с типа обратна връзка *динамично манипулиране*, който разгледахме по-горе. Директното манипулиране е начин на организиране на диалога, при който с представените на потребителя обекти могат да се извършват операции, зададени с преки действия между техните икони. Например ако всички налични обекти: файлове, дискове, принтери, плотери са показани с икони върху екрана, преместването на иконата на един файл върху принтера ще предизвика отпечатването му. Този стил е много удобен за начинаещи потребители, но може да е неприемлив за някои нужди особено за опитните програмисти. Читателят може да си представи поредицата от действия, които трябва да извършти чрез директно манипулиране за да архивира всички текстови файлове от всички свои директории. Много по-удобно би било да въведе само една команда включваща т.нар. *символи за обобщение*. В системата OpenVMS тази команда би била:

```
$ copy DISK1:[MYROOT...]*.txt;* TAPE1:[000000]*.txt
```

Най-добри резултати се получават, когато директното манипулиране се съчетае с използването на команден език. Това компенсира трудната разширяемост на този тип диалог.

Заслужава да отбележим, че директното манипулиране не е непременно свързано с използването на икони. Този стил е приложим и когато обектите са отбелязани с текстови описания.

4.3.2 Принципи на графичния диалог

Тук ще обобщим принципите, на които се основава проектирането на графичния диалог. На тези принципи са основани и всички основни интерактивни похвати, които разгледахме в началото на тази глава. Те обаче не се отнасят само до графичния вход, а по-скоро до двустранното взаимодействие *човек-компютър*.

ЕДНОТИПНОСТ. Графичният диалог трябва да се основава на ограничен брой общи правила, които са приложими в почти всички ситуации при избягване на изключения и специални случаи. Основната причина да се съблюдава този принцип е да се позволи на потребителя да усвои лесно правилата на диалога и да прилага знанията, получени при попадане в една ситуация в почти всички останали. Този е най-лесният начин за потребителя да схване логиката на една система, без да стига до ситуации, в които не може да се ориентира. Някои наричат този принцип *закон за минималното учудване*. Ще дадем някои примери на елементи на диалога, съобразени с него:

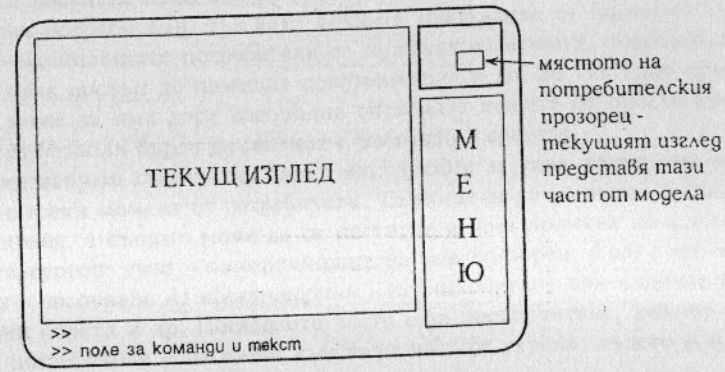
- винаги един и същ цвят за изобразяване на посочен обект;
- постоянно място на екрана за визуализиране на състоянието;
- фиксирано място за елементите от менюто;
- еднотипно значение на набор от клавиши във всички ситуации: `ESC`, `RETURN`, `F1`, `DEL` и т.н.;
- достъп до определена група глобални команди по всяко време: `HELP` (за допълнителна информация), `UNDO` (за анулиране на последното действие), `REDRAW` (за прерисуване на изображението) и др.
- прилагане на достъпните команди към всеки тип обекти при предвидим резултат;
- един и същ стил в подобни ситуации например динамична манипулация при всички геометрични трансформации;
- една и съща последователност от графични дейности за извършване на подобни действия: MOVE-Object-vector, MIRROR-Object-axis;

ОСИГУРЯВАНЕ НА ОБРАТНА ВРЪЗКА. Важността на обратната връзка беше показана при разглеждането на основните интерактивни дейности. Обратната връзка там бе разгледана главно на лексично ниво при подпомагане за извършване на основните дейности. Това става с използването на различните интерактивни похвати: графичен показалец, координатите на положението, в което той се намира, графичните скали и др. Обратна връзка е необходима на всички нива на графичния диалог.

На синтактично ниво приложната програма е необходимо да визуализира резултата от извършването на всяка интерактивна дейност: отбелязване на въведената позиция, мигане на посочения обект, изписване в определено поле на избраната команда или възможност и др. Друг вид синтактична обратна връзка е определен сигнал към потребителя, че въведената поредица от лексеми е достатъчна за извършването на определено действие.

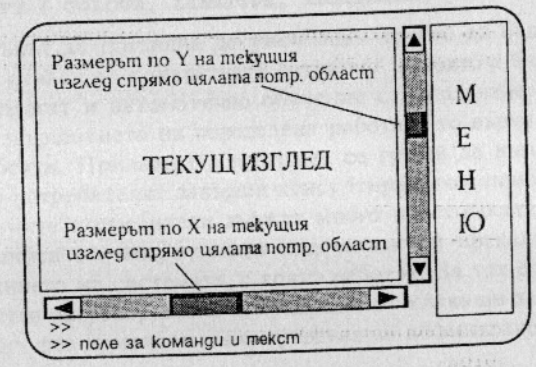
Най-важна е информацията, която потребителя получава за резултата от своята работа - семантичната обратна връзка. Естествено е след изпълнение на някаква геометрична трансформация обектът да се визуализира в новото му положение. Ако операцията, която се изпълнява е времеемка, тогава графичното представяне на някакъв брояч на извършената работа е особено нужен. В една графична приложна програма има нужда от представяне и на допълнителна информация, която подпомага потребителя в работата му:

- *Каква част от модела е визуализирана в момента?* (или по-точно къде е разположен потребителският прозорец на текущия изглед спрямо границите на потребителското пространство). Един начин за това е да се отдели специално поле от екрана, което съответства на цялото потребителско пространство и в което един правоъгълник да отбелязва мястото на потребителския прозорец (фиг. 4-17). Този метод позволява бърза промяна на текущия изглед чрез преместване на правоъгълника на прозореца върху желаната за визуализиране част.



Фиг. 4-17

Друг начин е да се визуализират един хоризонтален и един вертикален плъзгач, изобразяващи границите на потребителското пространство по двете оси, както това е показано на фиг. 4-18. Маркерите върху всеки от плъзгачите да има големина, съответстваща на размера по съответната координата на потребителския прозорец.



Фиг. 4-18

- *Какви са текущите стойности на определен набор от параметри, които имат графичен смисъл?* (и нужда от подходяща визуализация): тип, дебелина и цвят на линията, образец на запълване и др.

ВЪЗСТАНОВЯВАНЕ СЛЕД ГРЕШКА. Грешките при графичния вход са нещо естествено. Приложният програмист може да предотврати голяма част от допусканите грешки като направи недостъпно избирането на команди за извършването на операции над неподходящ тип обекти, направи недостъпни за избиране обекти, за които дадена операция е неприложима и др. Въпреки това, потребителят ще продължава да прави грешки и всяка приложна програма трябва да осигурява възможности за възстановяване на състоянието преди извършената грешка.

Най-удобно е възстановяването да се извършва чрез връщане назад в поредицата от въвеждания, които потребителят е направил. Това може да става на всичките нива в диалога: *семантично* (анулиране на последната изпълнена команда); *синтактично* (анулиране на последния графичен вход при задаване на параметрите на една команда) и т.н. Споменатото анулиране (UNDO) може да бъде реализирано на едно или на много нива. Някои приложни програми запазват цялата история на графичния диалог (в стек) и позволяват анулиране на всяка стъпка в обратен ред. При връщане назад повече от необходимото потребителят може да анулира самото връщане назад чрез аналогична команда (REDO). Самите команди UNDO и REDO не се записват в стека. При използване на система от менюта тези команди могат да носят и името на командата, към която те ще бъдат приложени например UNDO_Copy или REDO_Format. Ще отбележим, че горните две команди действат изключително на семантично ниво. Те не бива да се използват за анулиране на позиционирането на последния въведен връх на многоъгълник, защото това е синтактично анулиране. В този случай е по-удобно да се използва специален бутон на локатора.

Друго средство на семантично ниво е *явното потвърждаване* на резултата от една команда. При завършване на изпълнението на командата, приложната програма визуализира резултата и задава въпрос на потребителя дали го приема. Потребителят от своя страна трябва да даде отговор на този въпрос преди да предприеме каквото и да е по-нататъшно действие. Това значително увеличава броя на диалоговите лексеми за изпълнение на една последователност от команди. Друга алтернатива е *неявното потвърждаване*, при което изборът на нова команда автоматично потвърждава резултата от предишната. Това е предпочитано средство, особено когато е съчетано с възможност за анулиране.

Много важно е също потребителят да може да прекъсне работата на дадена команда, когато е ясно, че резултатът от нея не може да е удовлетворителен. Това може да става по два начина: или чрез специална команда CANCEL (достъпна по всяко време), или при избор на нова команда, когато текущата очаква вход на данни. При последната възможност въведените данни трябва да се игнорират, което може да доведе до неочаквани от потребителя ситуации.

Накрая ще отбележим, че при разработване на индустриални системи, където е недопустима загуба на каквато и да е информация, трябва да се отчете възможност за грешка не само на потребителя, но и на изчислителната система и нейното програмно осигуряване. Някои мощни CAD системи съхраняват в т.нар. *журнал* (journal file) или *протокол* всеки интерактивен жест на потребителя. Журналът служи за преиграване на цялата поредица от интерактивни действия, в случай че възникне системна грешка. Изискванията за постоянна памет на такива системи естествено са големи. Любопитно е да отбележим, че журналът при моделирането на дизеловия двигател DARPA със системата Pro/Engineer е надхвърлил 300MB.

СЪОБРАЗЯВАНЕ С НИВОТО НА ПОТРЕБИТЕЛЯ. Ефективен диалог може да се постигне само ако приложният програмист предвиди средства за всички нива потребители, тъй като техните изисквания са понякога коренно различни. Начинаещите потребители се нуждаят от менюта, подсказващи общения, пряк достъп до помощна документация и др. За тях една приложна програма може да има дори специална система с менюта от прости команди, което да им позволи бързо да свикнат с работата в средата.

Подсказващите съобщения дават инструкции за това какъв тип вход се очаква във всеки момент от потребителя. Те могат да се изписват в специално поле от екрана, а същото може да се постигне и чрез промяна на формата на графичния курсор: *лупа* - позициониране на нов прозорец; *длан с протегнат показалец* - посочване на обект; *мерник* - позициониране при наличие на гравитационни полета и др. Последното често е за предпочитане, защото вниманието на потребителя е насочено към тази част от екрана, където в момента се намира курсора.

Друг вид подсказване е *динамичната навигация*: представяне на възможността (или възможностите), която приложната програма смята че потребителят желае да избере. Този интерактивен похват ние разгледахме при посочването, но тя може да се използва дори и в командния език. Ако потребителят напише MOD системата автоматично допълва до MODIFY и предлага възможностите:

```
MODIFY { COLOUR, LINETYPE, THICKNESS, LAYER, VERTEX }
```

Достъпът до помощна документация трябва да бъде контекстно зависим, т.е. да се отнася до конкретната ситуация, в която е бил поискан. Някои системи предлагат и автоматично обучение с използване на анимация, за да се покаже извършването на определена работа и то върху създадените от потребителя обекти. Приложната програма се грижи да възстанови изображението след като потребителят завърши консултирането с помощните документи.

Опитните потребители държат много на скоростта на диалога и предпочитат липсата на непрекъснато подсказване и прекалено много информация за състоянието на системата, с която работят. За тях е удобно да се предоставят средства за ускоряване: комбинация от клавиши за активиране на команди, диалог чрез команден език, средства за дефиниране на командни процедури или макроси.

Автоматичното съхраняване на поредица от действия, които потребителят предприема във вид на командна процедура или макрос, е особено предпочитано средство. Това е елементарен начин за дефиниране на диалогови последователности чрез примери.

МИНИМИЗИРАНЕ НА НЕОБХОДИМОСТТА ОТ ЗАПАМЕТЯВАНЕ. Много приложни програми принуждават потребителя за запомня голям обем информация, която не винаги е нужна. Минимизирането на тази информация е една от основните задачи на приложния програмист при проектирането на диалога. Ще дадем само някои основни идеи как може да се постигне това:

- *Стимулиране на мускулната памет* като се разполагат менюта и бутони на фиксирани места върху екрана и таблета. Това означава и да не се разместват елементите на менюто дори ако измежду тях има такива, които не са достъпни в определен момент.
- *Структуриране на диалога* чрез подходящо групиране на елементите в менюто и командите от езика.
- *Подходящо използване на подразбиране* например повторение на една команда за нов обект, изпълнение на нова операция за последния създаден или модифициран обект и др.
- *Минимизиране нуждата да се помни реда на параметрите* на командите и операциите чрез еднотипни последователности, произволен ред на въвеждане на параметри, подсказване и "динамична навигация".
- *Запазване на графичното изображение* при представяне на допълнителна информация. Особено необходимо е това при достъп до помощна документация, която е в текстов вид. Не е нужно потребителят да запомни цялото изображение, което представя ситуацията, за която желае помощ. Типична крайност е смяната на графичен и текстов екран.
- *Извършване на посочването графично*, а не по име. Извършване на избор по графичен начин винаги, когато е възможно: например зареждане на един чертеж от файл чрез въвеждането на името му е нежелано от всеки потребител.
- *Структуриране на изображението* на модела чрез подходящо степенуване на елементите му по важност и детайлност при използване на цвят, типове линии, помощни обекти и др.

4.3.3 Проектиране на семантиката на диалога

Както подчертахме по-горе, семантиката на графичния диалог зависи от конкретната приложна област и специалните изисквания на потребителите в нея. Основните въпроси, на които приложният програмист трябва да си отговори са:

- Какви са обектите, с които потребителят ще работи?
- Какви са техните атрибути?
- Какви методи за създаването на тези обекти ще се предоставят?
- Какви операции ще може да извършва потребителят с обектите?
- Как ще се управлява визуализацията им?

ИЗБОР НА НАБОРА ОТ ОБЕКТИ. Тук под обекти разбираме елементарните графични единици, в термините на които потребителят ще изразява една или друга идея от своята приложна област. Изборът на този набор до голяма степен се определя и от вида на геометричния модел, който приложната програма използва. Ще изброим накратко някои варианти при решаването на тази задача:

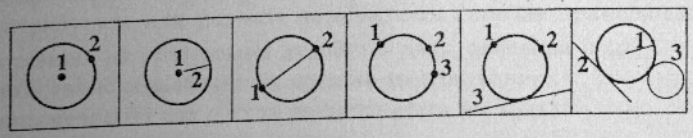
- *графични примитиви*: Най-простият начин е да се даде на потребителя възможност да работи със същите примитиви, с които работи и графичната система: отсечка, дъга, окръжност, начупена, текст. Така се постъпва в повечето приложни програми за автоматизиране на чертожната дейност.
- *геометрични обекти*: обекти, които имат по-скоро аналитично, отколкото параметрично математическо представяне: безкрайна права, окръжност, елипса, хипербола, парабола, криви от по-висока степен и др.
- *затворени плоски обекти*: обекти, които заграждат равнинна област - квадрат, правоъгълник, окръжност, сектор, сегмент, многоъгълник и др. Тяхното използване е оправдано в приложения, в които потребителят работи с площи, а не с контури.
- *специални графични символи*: като пример можем да вземем символи, представящи транзистори, резистори, диоди. Те са удобни при всеки случай, в който потребителят естествено борави с елементи, които са неделими и не се налага те да се уголемяват, намаляват, да им се променя формата и т.н.
- *специални помощни обекти*: такива като осеви линии, оразмерителни линии, щрих-линии и др.

Атрибутите на тези обекти могат изцяло да съвпадат с графичните: цвят, дебелина, тип, а могат и да нямат пряко отношение към визуализацията: чертожен пласт, ниво на детайлност и др., които ще разгледаме подробно при геометричното моделиране.

МЕТОДИ ЗА СЪЗДАВАНЕ НА ОБЕКТИТЕ. Една приложна програма може да предостави коренно различни начини за създаване на един и същ обект. Методите за създаване на обектите до голяма степен се определят още при концептуалното проектиране на диалога. Тук ще посочим само някои от най-често използваните методи:

- *конструиране чрез набор от математически методи*: Най-разпространеният начин за задаването на един графичен примитив и по-общо на геометричен обект е да се предостави набор от методи, всеки от които има различен брой и вид параметри, но води до създаването на един и същ тип обект. По този начин потребителят във всеки момент от интерактивната работа може да избере този метод, за който е най-лесно да зададат параметрите. Например една окръжност може да се зададе по всеки от следните начини (фиг. 4-19):
  - по зададени център и радиус
  - по зададени център и точка, през която тя минава
  - по зададени две диаметрално разположени точки върху нея
  - по зададени 3 точки, през които тя минава
  - по две точки, през които минава и елемент, до който се допира

- по радиус и два други обекта, до които тя се допира и др.

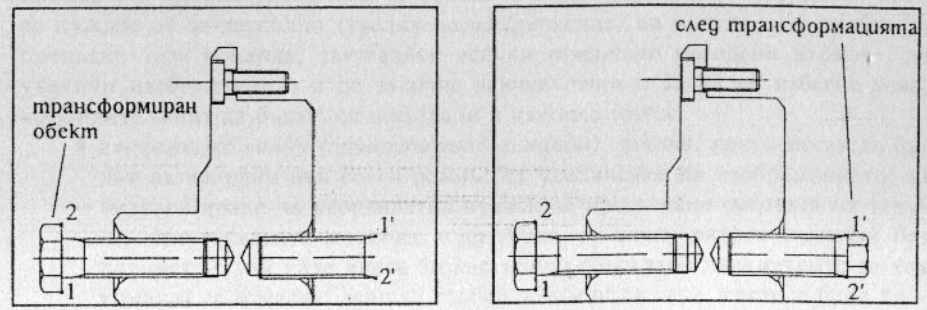


Фиг. 4-19

- *конструиране с използване на помощни линии*: Този начин до голяма степен имитира действията на чертожниците в тяхната работа без използване на компютър. Чертожниците първо нанасят някои отправни линии, които не са част от основния чертеж, но които се ползват при създаването на основните обекти. Те често се наричат *помощни линии* или *линии на конструиране* (construction lines). Например пресечните точки на тази линии могат да служат за характеристични точки на основните елементи. Преди начертаване на една дъга може да е по-лесно да се начертае първо окръжността, върху която тя лежи. За тази цел са необходими команди като:
  - задаване на елемент, лежащ върху друг елемент
  - задаване на обект с краища в определени пресечни точки и др.
- *конструиране по условия*: Един графичен примитив може да бъде конструиран и като се зададе система от условия, които той трябва да удовлетворява. Условията са най-различни геометрични релации, които имат подходящо аналитично представяне. Те често се наричат и *ограничения* (constraints), защото всяко ново условие ограничава степените на свобода на обекта, който се конструира. Следователно условията се задават дотогава, докато се фиксират всичките му степени на свобода. Често използвани условия са тези за успоредност, допирателност, инцидентност, хоризонталност, вертикалност, фиксирана x-координата, фиксирана дължина и др. Ето един пример за набор от условия, наложени при конструирането на отсечка:
  - да е хоризонтална
  - първата точка да лежи върху определен елемент (инцидентност)
  - да има определена дължина
- *скициране и ескизиране*: прост начин за задаване на начупени е следенето на движението на локатора чрез последователно запомняне на позиции или през определен интервал от време, или при преместване на локатора, по-голямо от определена стойност.

ОПЕРАЦИИ С ОБЕКТИТЕ. Конструирането е само една от операциите в една приложна графична програма, макар и най-важната. За да могат обектите, които се създават да бъдат променяни, преместени и пренареждани, потребителят трябва да разполага с необходимите интерактивни средства. Това могат да бъдат:

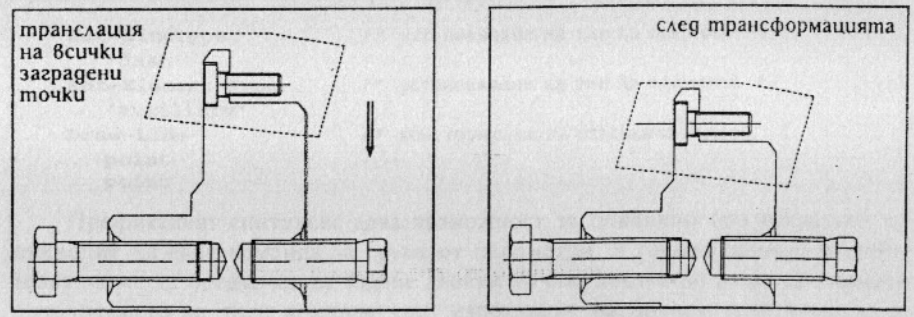
- *геометрични трансформации*: транслация, ротация, хомотетия, мащабиране и осева симетрия, както и композиции от тях. Освен преместване или движение, един обект може също да бъде трансформиран и при запазване на неговия първообраз, т.е. създаване на негово копие. Удобен начин за задаването на една обща трансформация е като се посочат две точки от обекта и техните образи след изпълнението на трансформацията. Зададената по този начин трансформация на фиг. 4-20 включва транслация, ротация на 180 градуса и хомотетия.



Фиг. 4-20

Друга удобна трансформация е многократно копиране по продължение на права, в матрица и в кръг. Това спестява много интерактивни действия в приложни програми, където еднотипността е често срещана.

- *специални трансформации*: това са трансформации, които не се представят чрез стандартните геометрични трансформации:
  - построяване на еквидистанти, т.е. отместени образи;
  - модифициране чрез трансформиране на част от характеристичните точки на един обект, което води до скъсяване или удължаване на обекта при транслация - фиг. 4-21, свиване или уголемяване при мащабиране и др.



Фиг. 4-21

- *теоретико-множествени операции*: обединение, сечение и разлика, наричани още *булеви* операции с множества - типични при работа с равнинни области.

- *модификация на примитиви*: Това са операции, които променят някои от атрибутите или формата на обектите. Типични примери са:
  - промяна на графичните атрибути: цвят, дебелина и др.;
  - скосяване и заобляне на връх на многоъгълник;
  - скъсяване на два обекта до пресечната им точка;
  - разделяне на обект на части;
  - промяна на подредбата на обектите и др.
- *структуриране*: операции, които задават включването на един обект в друг за създаването на групи, с които да се работи като с един елемент - изброяване на съставните елементи; загражданe на елементите, които трябва да образуват група с многоъгълник или правоъгълник и др.

УПРАВЛЕНИЕ НА ВИЗУАЛИЗАЦИЯТА НА ОБЕКТИТЕ. Важна част от проектирането на интерфейса е да се реши какви средства ще се дадат на потребителя да управлява изобразяването на обектите. Това определя до голяма степен ефективността на обратната връзка при взаимодействието. За тази цел приложният програмист трябва да реши:

- как да се задават потребителските прозорци, изгледи и чертожни полета - например с команди за увеличаване и намаляване на мащаба на изображението, за показване на панорамен изглед и др.
- как да се управлява детайлността на изображението - кои типове примитиви да са видими и кои не; как да се визуализират помощните обекти; целият ли модел да се визуализира при панорамен изглед или само негова скица; как да се визуализират атрибутите, които нямат пряк графичен смисъл и др.

4.3.4 Режими в диалога. Проектиране на диалоговия синтаксис

Както казахме по-горе, *режимът* е състояние на диалога, при което само определени негови думи могат да се въвеждат от потребителя. За да се изпълни някаква операция потребителят трябва да избере съответната команда и да зададе параметрите ѝ. Например за да се нарисува една отсечка, като параметри са необходими началната и крайната ѝ точки. По време на въвеждането на тези точки диалогът се намира в определен режим, в който се очаква главно действието *позициониране*.

В някои системи се използва терминът *модалност* за отбелязване на елемент на диалога, който трябва да бъде изпълнен преди каквото и да било друго действие. Типичен пример са т.нар. *модални диалогови прозорци*, които наподобяват бланки с определени полета за избор или попълване на текст и които принуждават потребителя да работи само в техните граници, без да избира елементи от менюто вън от тези прозорци или да задава допълнителни команди.

Синтаксисът на командния език определя в голяма степен възможните режими в диалога. Функционално една команда може да се запише така:

```
Command (Param1, Param2, ..., ParamN )
```


ПРЕФИКСЕН СИНТАКСИС. При префиксния синтаксис първо се избира командата, а след това един след друг и нейните параметри. Този синтаксис определя два режима: *избор на команди* и *задаване на данни*.

В първия режим потребителят може само да избира команди, а във втория - техните параметри: обекти, местоположения и др. Стриктното придържане към тези два режима може да създаде много трудности в диалога. Например ако потребителят иска да зададе многоъгълник, но за определен връх се нуждае от по-детайлно (увеличено изображение) на модела, той трябва да прекъсне тази команда, загубвайки всички предишни въведени върхове, да увеличи изображението и да започне всичко отново. За да се избегне това, командите могат да бъдат организирани в няколко групи:

- *глобални команди (приоритетни команди)*: такива, които могат да бъдат активирани във всеки режим: за увеличение на изображението, за визуализиране на координатна мрежа, за прекъсване работата на текущо изпълняваната команда и др. Това са почти винаги команди без параметри, тъй като иначе би настъпило объркване параметрите за коя команда се отнасят: дали за глобалната или за тази, която е била временно прекъсната.
- *общи команди*: които могат да бъдат активирани, само когато програмата очаква избор на команда.
- *специални команди*: такива, които могат да бъдат активирани само в определен режим на работа на приложната програма - например при автоматично генериране на командна процедура.

Повечето команди реално се нуждаят от голям набор от параметри. Дори за да се нарисува само една отсечка, трябва да се зададат както крайните ѝ точки, така и атрибутите ѝ: цветът; типът на линията; дебелината ѝ; оформянето на краищата ѝ; чертожният пласт, към който тя принадлежи, какъв елемент от модела е: (основен, помощен, оразмерителен) и т.н. За да се ускори диалогът, за атрибутите се приемат текущи стойности, които могат да бъдат променяни във всеки момент чрез приоритетна команда.

```
Set-Linetype                /* установяване на тип на линията */
   'dash'
Set-Element-Type            /* установяване на тип на елемента */
   'auxiliary'
Draw-Line                   /* конструиране на отсечката */
   point
   point
```

Префиксният синтаксис дава възможност за циклично (многократно) изпълнение на една команда за група от параметри. В горния пример потребителят може да продължи да задава двойки точки, докато не реши да завърши изпълнението на тази команда чрез използване на приоритетна команда за преминаване в режим на избор на команди. Приложната програма може и да съчетава двата основни режима като позволи при завършена последователност от параметри да се продължи с нови параметри или с избор на нова команда.

ПОСТФИКСЕН СИНТАКСИС. В много случаи е удобно първо да се зададе обектът и след това действието, което да се извърши с него. Този вид синтаксис не налага такова разграничение на режимите на диалога. Той от своя страна дава възможност над един обект да се извършат няколко последователни модификации. Постфиксният синтаксис е най-удобен именно при последователни трансформации и се реализира с въвеждането на *текущ обект*, за избирането на който е определена специална команда SELECT.

```
SELECT
   object                  /* посочване на текущия обект */
Scale-Object               /* мащабиране на текущия обект */
   X factor, Y factor
Mirror-Horizontal          /* осева симетрия относно хоризонтална ос */
   point
Modify-Attributes          /* промяна на атрибутите на текущия обект */
```

Всички команди в горната последователност се отнасят за първоначално избрания обект. Този синтаксис, от друга страна, не позволява лесното изпълнение на една и съща команда за поредица от обекти. Идеята за текущ обект може да се продължи до използване на *текущо положение*, *текущ вектор*, *текущ ъгъл*, *текущ коефициент* и т.н., което на практика превръща командите в команди без параметри. Трябва, разбира се, да се въведат правила за установяване на текущия обект, от които да е ясно при изтриване на текущия обект кой е новият и има ли въобще такъв; при създаване на нов обект променя ли се текущият; могат ли няколко обекта да са едновременно текущи и т.н.

СВОБОДЕН СИНТАКСИС. Друга възможност при префиксния синтаксис на командите е техните параметри да могат да се въвеждат в произволен ред. Например ако командата е `Tangent-Circle-Pnt`: задаване на окръжност, която е допирателна до даден елемент и има за център определена точка, елементът и точката да могат да бъдат зададени в произволен ред:

```
Tangent-Circle-Pnt          или      Tangent-Circle-Pnt
   object                               point
   point                                object
```

Това не винаги е възможно, тъй като някои параметри са еднакви като лексеми (имат един и същ тип), но имат различен смисъл в командата. При създаване на окръжност с център и точка, през която тя минава, редът на въвеждане трябва да е фиксиран, освен ако няма допълнителни средства, които да задават смисъла на лексемите.

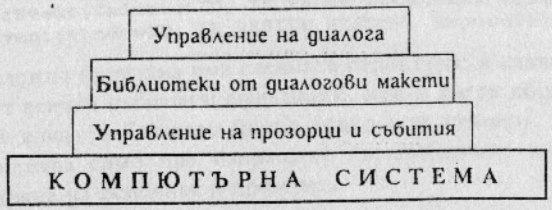
4.4 СИСТЕМИ ЗА УПРАВЛЕНИЕ НА ДИАЛОГА

Дотук разгледахме основните интерактивни дейности и похвати и тяхното място при проектирането на графичния диалог. Съвременните графични системи предлагат голяма част от тези похвати наготово, така че тяхното разработване на базово ниво рядко се налага. В тази част ще разгледаме накратко структурата на една система, която поема голяма част от управлението на

диалога, както и ролята и задачите на приложната графична програма в този случай. Съвременните графични системи се отличават главно със следните особености:

- възможност за паралелна интерактивна работа на няколко приложни програми, всяка от които има собствен прозорец на своя процес;
- управление на графичния вход асинхронно чрез входни опашки и събития, които активират определени компоненти на приложните програми;
- управление на паралелността, общите ресурси и взаимодействието между отделните приложения;
- наличие на библиотеки от интерактивни макети: средства от високо ниво, които приложният програмист може да използва за реализиране на различните интерактивни похвати: менюта, бутони, диалогови прозорци и др.
- възможност за отделяне на диалога от съдържанието на приложната програма, така че неговите елементи и структурирането им да могат да се променят независимо от нейната функционалност;
- възможност за дефиниране на диалоговият синтаксис независимо от функционалността на приложната програма.

Общата структура на една такава система е следната:



Фиг. 4-22

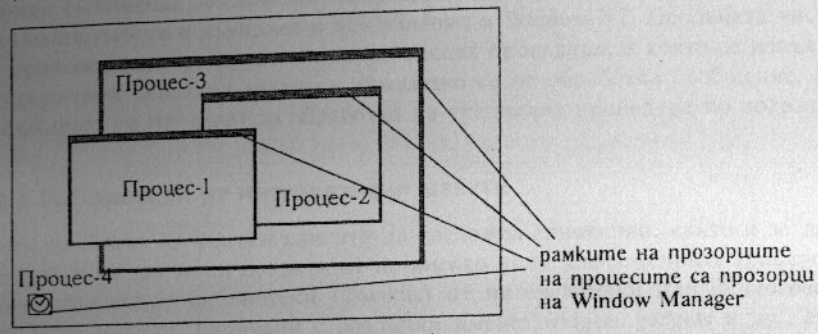
По-нататък ще се спрем на особеностите на всяко от тези нива и изискванията, които те налагат на приложните графични програми.

4.4.1 Системи за работа в прозорци

Такъв тип система може да се разглежда като надстройка на операционната система, която управлява диалоговите ресурси на компютъра. Един от тези ресурси е екранът на графичния дисплей. На всяка приложна програма се предоставя част от екрана, наречена *прозорец на процеса*, който е независим от прозорците на останалите паралелни процеси.

Системата за работа в прозорци е построена най-често на принципа "клиент-сървер" (X Window System, NeWS, DEC VWS). Програмата-сървер е тази част от системата, която се грижи за създаването на прозорци, промяната на размерите, местоположението и наредбата им, визуализирането на информацията в тях и унищожаването им. Клиенти са всички приложни графични програми, които ползват диалоговите услуги на сървера. Един специа-

лен клиент, наречен *програма за управление на прозорците* (Window Manager) обслужва работата на потребителя по избирането на прозорец, в който да работи - текущ или активен прозорец; разместването и промяната на размерите им; превръщането на прозорците в икони и др. Прозорците на този клиент са тънки рамки около останалите прозорци, в които има различни бутони. С натискането на тези бутони потребителят променя размера, подредбата, временно затваря (изобразява във вид на икона) прозорците на процесите.



Фиг. 4-23

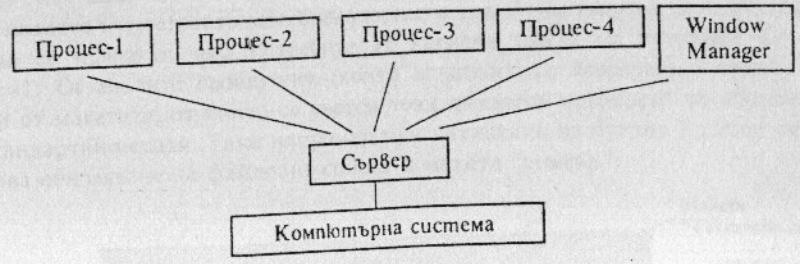
Връзката между отделните процеси в показания пример е представена на фиг. 4-24. За да се ускори комуникацията между клиентите и сървера, тя се извършва не чрез пряко обръщение към функции на сървера, а като се изпращат съобщения асинхронно. Тези съобщения са както изходни (нарисувай отсечка, създай прозорец), така и входни. Входните съобщения се генерират от сървера при работата на потребителя с входните диалогови устройства. За да може диалогът да е ефективен, в повечето системи приложните програми нямат простата линейна структура на последователност от операции.

Тъй като входните устройства са ресурси, които се ползват от няколко клиента конкурентно, приложната програма не осъществява директно обръщение към абстрактното устройство. Вместо това, в нея се декларира кои събития, свързани с това устройство са от интерес за приложния програмист и от кои точно процедури на приложната програма те ще бъдат обработвани. За всяко от тези събития в приложната програма има специален вид процедури наричани *callback-процедури* (за тях в графичната система е дефиниран определен тип). Към тях се прави обръщение от графичната система (и никога от самата приложна програма директно) всеки път, когато съответното събитие настъпи. Входните събития във всяка графична система и за всяко абстрактно устройство могат да бъдат различни, но като типични примери могат да се посочат:

- натискане на клавиш от клавиатурата;
- промяна на текущата стойност на валюатора;
- промяна на текущата позиция на локатора;
- натискане или освобождаване на бутон на локатора;
- избор на възможност и др.

Събитията могат да се отнасят не само пряко до входните устройства, а и до промени в прозорците, предизвикани от работата на програмата за управление на прозорци:

- променен е размерът на прозореца;
- променено е мястото на прозореца;
- променена е наредбата на прозорците и др.



Фиг. 4-24

Показаните примери илюстрират свързването на определен тип събития с дефинираните от приложния програмист функции за обработката им.

```
SetLocatorEvent(LocatorID, IE_POSITION_CHANGE, MoveLocProc)
SetKeyboardEvent(KeyboardID, IE_KEY_PRESSED, KeyPressProc)
SetValuatorEvent(ValuatorID, IE_VALUE_INCREASE, IncrProc)
SetWindowEvent(WindowID, IE_RESIZE_WINDOW, ExposeProc)
```

Самата приложна програма има следната структура: в главната ѝ програма се декларират всички събития и функциите, които ще ги обработват и след това програмата влиза в безкраен цикъл (явно или неявно), излизането от който се осъществява само при изпращане на съобщение към графичната система за затваряне на основния ѝ прозорец.

```
main()
{   . . .
    /* свързване на събитието за движение на локатора с MoveProc */
    SetLocatorEvent(LocatorID, IE_POSITION_CHANGE, MoveProc);
    /* свързване на събитието за направен избор с ChoiceProc */
    SetChoiceEvent(ChoiceID, IE_ITEM_SELECTED, ChoiceProc);
    /* свързване на събитието за промяна на прозореца с ExposeProc */
    SetWindowEvent(WindowID, IE_RESIZE_WINDOW, ExposeProc);
    . . .
    /* програмата влиза в безкраен цикъл */
    ApplicationLoop();
}

CallbackProc MoveProc(event)
{    /*  този код се изпълнява при всяко движение на локатора  */
     /*  функцията получава всички параметри на съобщението */
}

CallbackProc ExposeProc(event)
{    /*  този код се изпълнява при всяка промяна на прозореца  */
}
```

```
CallbackProc ChoiceProc(event)
{    /*  този код се изпълнява при всеки избор с устройството  */
   switch (event->item_selected) { . . .
      /*   предприемани действия при всеки отделен избор */
   }
}
```

Друг начин на работа е вместо за всяко събитие да се отдели процедура за обработката му, да се дефинира една единствена процедура, която да получава всички *съобщения за събития*, свързани с определен прозорец на процес. Този начин на работа е възприет в MS-Windows и Window-NT. Основната част от една приложна програма там е именно такава процедура, в която се изпълняват определени действия за всяко нуждаещо се от обработка съобщение, а всички останали се предават за обработка на специална процедура по подразбиране.

4.4.2 Библиотеки от интерактивни макети

С цел диалогът на всички клиенти да изглежда еднотипно, както и за да се използват интерактивни похвати от по-високо ниво, съвременните графични системи предлагат библиотеки (Toolkits) от интерактивни или *диалогови макети*. Тези макети, наричани с различни имена: widgets, gadgets и др., са основните потребителски метафори в диалога като бутони, менюта, полета за редактиране, плъзгачи, прозорци и др. Приложната програма може да използва тези макети както за изпълнението на основните интерактивни дейности, така и за организирането им в по-сложни диалогови елементи, каквито са диалоговите прозорци. Една графична система може да предлага няколко различни библиотеки. Типичен пример е X Window System, в която приложният програмист може да използва различни по мощност библиотеки: Athena Toolkit, XtIntrinsics, OSF/Motif, OPEN LOOK, InterViews и др.

Диалоговите макети са прозорци, които могат да съдържат в себе си други прозорци, които спазват определен стил на визуално представяне. Когато потребителят взаимодейства с един такъв макет, приложната програма, която го използва получава определен набор от съобщения, че са настъпили входни събития в използвания тип макет. Един графичен бутон е прост диалогов макет, за който са характерни съобщенията:

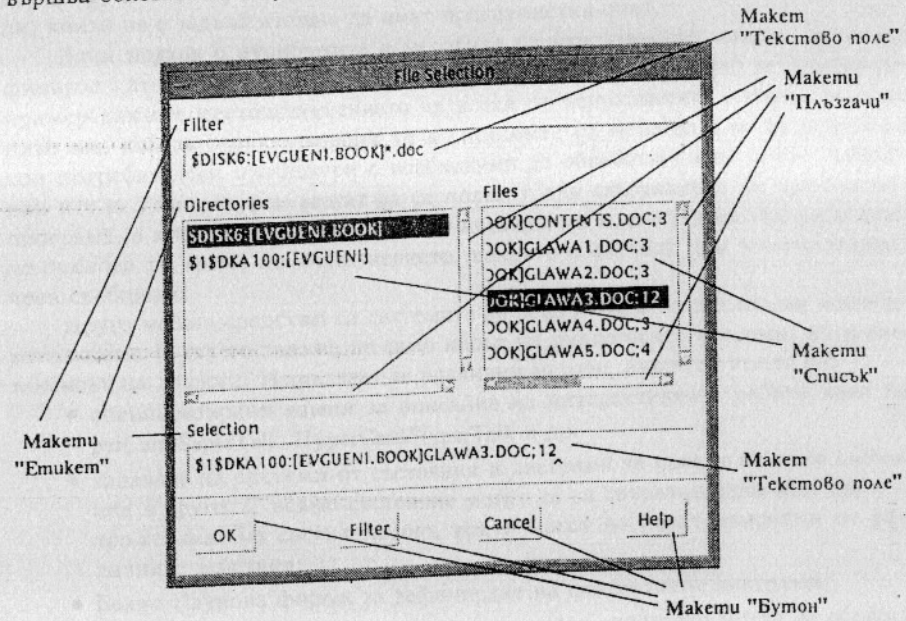
- IM_BUTTON_PRESSED - бутонът е натиснат
- IM_BUTTON_DOUBLECLICKED - бутонът е натиснат два пъти
- IM_BUTTON_DEPRESSED - бутонът е отпуснат и др.

Макетът MainWindow (основен прозорец) пък е по-сложен и се състои от макет за работна област, макет за падащо меню, макет за командно поле и др.

Библиотеките са много често обектно-ориентирани. Всеки макет принадлежи към клас (или класове) и наследява от него определени методи и атрибути. При използването на определен макет приложната програма може да декларира специфични методи да обработка на някои съобщения, но може и да използва тези, които са дефинирани за целия клас. Следният набор от интерактивни макети се среща в почти всяка от този тип библиотеки:

- бутон, бутон "включен-изключен", система от радио-бутони;
- етикет, текстово поле, списък, плъзгач, текстово поле със списък;
- прозорец за съобщение, прозорец за отговор на въпрос, диалогов прозорец и др.

Ще илюстрираме казаното с един конкретен макет: *диалогов прозорец за избор на файл* (фиг. 4-25). По-долу е показан фрагмент от програма, използваща този макет от библиотеката OSF/Motif. В главната програма е обявено включването му, неговото местоположение и атрибути, както и методите, които ще се прилагат, ако потребителят натисне някой от бутоните OK или Cancel. Останалите съобщения (които всъщност са извънредно много - за всеки от макетите, от които се състои този диалогов прозорец) се обработват от стандартни методи. Така например при натискане на бутона Filter се извършва обновяване на файловия списък в макета "списък".



Фиг. 4-25

В показаната програма диалоговият прозорец ще се появи, когато потребителят избере съответния елемент от главното меню, което става при натискането на бутона, зададен с макета file_menu_item.

Повечето системи дават допълнителни възможности да дефиниране на макети от приложния програмист, което почти никога не е лека задача. Дори самото задаване на всички прозорци и всички атрибути на използваните макети може да е много трудоемко. Поради тази причина съвременните системи предлагат различни (и най-вече интерактивни) средства на приложния програмист за дефинирането на използваните диалогови елементи. В резултат се получава автоматично програма, в която се правят нужните обръщения към

съответните диалогови макети с атрибути и методи, които програмистът интерактивно е определил.

```
Widget mainwin, file, file_menu_item;

main()
{ . . .
   /* задаване на макета на бутона, активиращ диалоговия прозорец */
   file_menu_item = XmCreatePushButtonGadget(...);
   /* свързване на натискането на този бутон с openfile */
   XtAddCallback(file_menu_item,
                 XmNactivateCallback, openfile, NULL);
   /* дефиниране на параметрите на макета за избор на файл */
   XtSetArg(arg[0], XmNx, 90); XtSetArg(arg[1], XmNy, 90);
   XtSetArg(arg[2], XmNtitle, "File Selection");
   XtSetArg(arg[3], XmNdirMask, "$disk6:[evgueni.book]*.doc");
   /* включване на макета за избор на файл */
   file = XmCreateFileSelectionDialog(mainwin, "File", arg, 4);
   /* дефиниране на процедурите, които ще обработват
      събитията "файлът е избран" и "отказ" */
   XtAddCallback(file, XmNcancelCallback, FILEcanceled, NULL);
   XtAddCallback(file, XmNokCallback, FILEselected, NULL);
   . . .
   XtAppMainLoop(app_context);
}

/* показване на диалоговия прозорец за избор на файл */
void openfile(w, client_data, call_data)
Widget w; XtPointer client_data, call_data;
{  XtManageChild(file);
}

/* изтриване от екрана на диалоговия прозорец за избор на файл */
void FILEcanceled(w, client_data, call_data)
Widget w; XtPointer client_data, call_data;
{  XtUnmanageChild(file);
}

/* при избор на файл, извличане на името му и предаването му на
   приложна процедура за визуализирането му */
void FILEselected(w, client_data, call_data)
Widget w; XtPointer client_data;
XmFileSelectionBoxCallbackStruct *call_data;
{  if (call_data) {
      if ( !XmStringGetLtoR(call_data->value,
            XmSTRING_DEFAULT_CHARSET, &fileselected)) return;
        /* приложна функция за обработка на избрания файл */
      XtUnmanageChild(file);
      DisplayDocument(fileselected);
   }
}
```

Друг начин за улесняване работата по дефинирането на диалоговите компоненти е да се свържат определени променливи с определени диалогови макети: низова променлива с текстово поле, булева променлива с бутон "включен-изключен", променлива от изброим тип с радио-бутон, реална променлива с плъзгач и т.н.

4.4.3 Отделяне на диалога от функционалността на програмите

Горните средства дават възможност за създаването на приложни програми, в които диалогът се управлява изключително от потребителя. Нещо повече, те създават предпоставки диалоговата част на една приложна програма да може да се отдели и проектира независимо от основната ѝ дейност. Ако това се постигне, то би било лесно да се строят различни потребителски интерфейси за една и съща програма, като те се съобразяват и с нуждите на някой конкретен потребител при пълно запазване на функционалността на тази програма.

Системите за управление на потребителския интерфейс (User Interface Management Systems) позволяват проектирането на диалога да се извършва от друга група в програмния колектив и дори от подходящо обучени потребители, които не е задължително да имат програмистки опит.

Един подход е атрибутите и методите на използваните макети да се дефинират като *ресурси* в отделен файл (*файл от ресурси*). В него се описва например какво е местоположението на всеки от използваните макети, големините им, как са структурирани те в системите от менюта, как са оцветени, кои потребителски процедури е необходимо да обработват кои точно съобщения и т.н. Тези ресурси могат да се ползват при свързването на приложната програма, а могат и да се извличат от независимо компилирана *спецификация на диалога* по време на изпълнението, какъвто е случаят при комуникацията чрез съобщения.

Друго мощно средство са системите за дефиниране на диалогови последователности. С тях се задава не само видът на диалоговите лексеми, но и синтаксисът на диалога. Използват се различни методи, някои от които са:

- специализирани езици за описание на интерактивната работа като SuperCard/SuperTalk, HyperCard/HyperTalk и др.
- задаване на система от състояния и диаграми за преход от едно състояние в друго. С всяко състояние могат да са свързани една или няколко *променливи на състоянието*, които могат да бъдат изменяни от различните действия;
- Бекус-Наурова форма за дефиниране на синтактични диаграми;
- чрез *интерактивни примери*. Това е често използван метод за обобщаване на диалога, а именно чрез интерактивно демонстриране как една команда трябва да се избере, как да се задават параметрите и т.н., наричан още "watch-what-I-do"-метод. Системата съхранява примера и по аналогия с него строи целия диалог.

Задачи

4.1 Напишете програма за намиране на разстоянието между точка и отсечка. Напишете програма за намирането на разстоянието от точка до дъга от окръжност.

4.2 Дадена е координатна мрежа, определена от координатите на един от възлите на

мрежата и разстоянията по всяка от осите между съседните възли. Всички те са представени като числа с плаваща запетая. Напишете програма, която определя най-близкия връх от мрежата до дадена точка.

4.3 Даден е масив от отсечки, всяка зададена с координатите на двата си края. Напишете програма, която осъществява въвеждане на точка при налично гравитационно поле на всяка от отсечките и краищата им, както е показано на фиг.4-5.

4.4 Добавете към горната задача възможност за посочване и на пресечната точка на две от отсечките в масива.

4.5 Предложете начин за третиране на характеристичните точки на геометричните елементи, отговаряйки на въпросите на стр. 143-144.

4.6 Напишете програма, която реализира валюатор плъзгач чрез устройството локатор. Визуализирайте плъзгача като отсечка и маркер, който следи движението на локатора върху тази отсечка. Обърнете внимание, че маркерът не бива да излиза извън границите на отсечката.

4.7 Напишете програма, която реализира ротационен валюатор чрез локатор. Визуализирайте го като окръжност и отсечка от центъра ѝ, която показва текущата му стойност.

4.8 Напишете програма, която осъществява въвеждане на начупена чрез разтегателни отсечки, като на всяка стъпка потребителят може да зададе нова точка (получаване на BUTTON_1 като код от локатора) или да се откаже от предишния вход (получаване на BUTTON_2 като код от локатора). Как ще укаже потребителят, че е завършил въвеждането?

4.9 Решете горната задача за начупена с прави ъгли между съседните ръбове като показаната на фиг. 4-16.

4.10 Напишете програма за въвеждане на правоъгълник с предварително фиксирано отношение ширина/височина, като не променяте първата въведена точка. Използвайте разтегателен примитив.

4.11 Напишете програма за въвеждане на дъга по три точки, които лежат на нея, първата и последната от които задават краищата ѝ. Осигурете възможност за отказване на потребителя от последната въведена точка подобно на задача 4.8.

4.12 Проектирайте набор от методи за задаване на отсечка и дъга от окръжност.

4.13 Предложете интерактивен метод за задаване на правоъгълник, чиито страни може да не са успоредни на координатните оси.

4.14 Напишете програма за позициониране на точка при следене на локатора с курсор, като имате разпределението на екрана, показано на фиг. 4-18. При натискане на бутон върху някой от плъзгачите извършете съответното предефиниране на потребителския прозорец. Обърнете внимание на правилното изобразяване на курсора след прерисуването на изображението в потребителския прозорец.

4.15 Напишете прост команден интерпретатор на фиксиран набор от ключови думи, който допълва недописаната част от ключовата дума веднага щом няма двусмислен избор. Как бихте интерпретирали натискане на клавиша backspace?

4.16 Напишете програмата за разтегателна отсечка с обработване на двете съобщения BUTTON_DOWN и BUTTON_UP от една и съща функция като два варианта на оператор switch. Първото съобщение предизвиква начало на следенето и обновяването на положението, а второто - въвеждане на края на отсечката.

ГЕОМЕТРИЧНО МОДЕЛИРАНЕ НА РАВНИННИ ОБЕКТИ

Една от основните задачи при съставянето на приложна графична програма е да се избере наборът от обекти и начинът, по който те и връзките между тях да се представят, за да бъдат обработвани от нея. Докато обектите, с които работи графичната система, са сравнително ограничени: графични примитиви, атрибути и сегменти, то обектите във всяка приложна област могат да бъдат коренно различни. В архитектурното проектиране основните компоненти могат да бъдат стени, врати и прозорци; в проектирането на метални конструкции - греди, колони, планки и отвори; в машиностроенето - болтове, гайки, винтове, лагери; в електротехниката - транзистори, резистори, бобини и т.н.

Общото между всички тези обекти от гледна точка на приложния програмист е, че те имат определена геометрична форма (т.е. могат да бъдат схематично начертани) и тяхното местоположение и свързаност с останалите компоненти е от съществено значение за потребителя. Техническият чертеж е един прост начин за некомпютърно представяне на тази информация. Геометричният модел пък е именно компютърното представяне на формата, разположението, ориентацията и размерите на обектите, с които една приложна програма работи. В геометричния модел се съдържа още информация за връзките между отделните компоненти и други техни характеристики, които може и да нямат пряко отношение към формата им. Както всеки модел, геометричният модел се създава, за да може върху него приложната програма да извършва разнообразни анализи и изследвания, вместо това да се прави с моделираните обекти от реалния свят. В проектирането тези реални обекти дори още не съществуват. Ето защо в модела присъстват и данни, необходими за тези изследвания.

Ако целта на една приложна програма е да решава електромагнитни задачи и обекти в нея са двумерните сечения на разнообразни магнитни материали и проводници, то освен тяхната форма в геометричния модел трябва да има данни за типа на материала, магнитната му плътност, плътността на тока и др. Ще обобщим, като кажем, че геометричният модел се състои от определен тип обекти, които имат своя:

- *метрика (геометрия)*: разположението, формата, ориентацията и размерите на всеки от тези обекти. Това са различните им метрични ха-

рактеристики: координати, дължини, радиуси, ъгли и др.;

- *топология*: свързаността и структурираността на обектите в модела, тяхната взаимна зависимост;
- *атрибути на обекта*: каква роля изпълнява обектът в модела - основна или помощна, с каква степен на детайлност трябва да се визуализират неговите компоненти и др. Това са тези характеристики, които пряко влияят на формирането на атрибутите на визуализация - цвят, тип на линията, дебелина и др.;
- *неграфични данни*: данни за различните специфични изследвания, които ще се извършват върху тези обекти - маса, плътност, цена и др.;
- *обработващи алгоритми*: специфични обработки върху обектите като изпълнение на теоретико-множествени (булеви) операции, триангулация и др.

Да разгледаме един прост пример. Геометричният модел на един град е, грубо казано, планът на този град, в който основните обекти са сгради, улици, паркове, електрически кабели, водопроводни участъци и др. Метриката на тези обекти е формата на сградите, улиците и т.н., техните координати в плана на града. Свързването на група от сгради в жилищни комплекси, на комплекси и улици в квартали, на водопроводни участъци във водопроводна мрежа представляват топологията на този модел. Атрибут на тези обекти е например типът им, неграфични данни са броят жители, година на построяване, а обработващи алгоритми - свързването на два водопроводни участъка, намиране на разстоянието между две точки по уличната мрежа и др.

Особено важно място в геометричния модел заемат метриката и топологията на обектите. Конкретните особености на тяхното проектиране ще разгледаме при представянето на всеки от видовете геометрични модели. Тук само ще изтъкнем някои техни по-общи черти.

МЕТРИКА. При избора на начина за съхраняване на метричните характеристики е необходимо да се отчитат някои специфични особености на работата с геометрични форми:

- *възможност за работа с обекти с различна големина*. Отношението между размерите на реално използваните обекти може да бъде няколко десетични порядъка. Приложният програмист трябва да се грижи за запазването на точността при представянето на формата независимо от големината на обектите при изпълнение на разнообразните обработващи алгоритми.
- *необходимост от различна детайлност на работа*. Визуализирането на формата на един и същ обект може да бъде различно в различните случаи: с всичките му детайли, само с по-важните особености или само негова скица.
- *много динамично изменение на метриката*. Интерактивността на приложните графични програми изисква максимално бърза визуализация на резултата от всички операции и обработващи алгоритми.

ТОПОЛОГИЯ. Интерактивността налага подобно изискване и към топологията на обектите в геометричния модел. Освен че обектите могат да бъдат от съвсем различен тип: графични, текстови, процедури и др., връзките между тях могат също да бъдат най-различни, като такива за включване, разнообразни геометрични релации и др. Представянето на тези връзки до голяма степен определя и типа на самия геометричен модел. Основно се използват два типа модели:

- ***конотативни*** (connotative, intensive) - за първични се приемат атрибутите на обектите, така че всеки обект се представя като структура, в която има определено място за всеки от възможните атрибути. Например обектът CAR (автомобил) се представя като състоящ се от двигател, колела, шаси и др.:

```
CAR = { ENGINE, WHEELS, BODY, ... }
```

- ***денотативни*** (denotative, extensive) - първични са всички обекти, а връзките между тях са техни атрибути, зададени например като техни предикати:

```
consists_of(CAR,ENGINE); consists_of(CAR,WHEELS);
consists_of(CAR,BODY); . . .
```

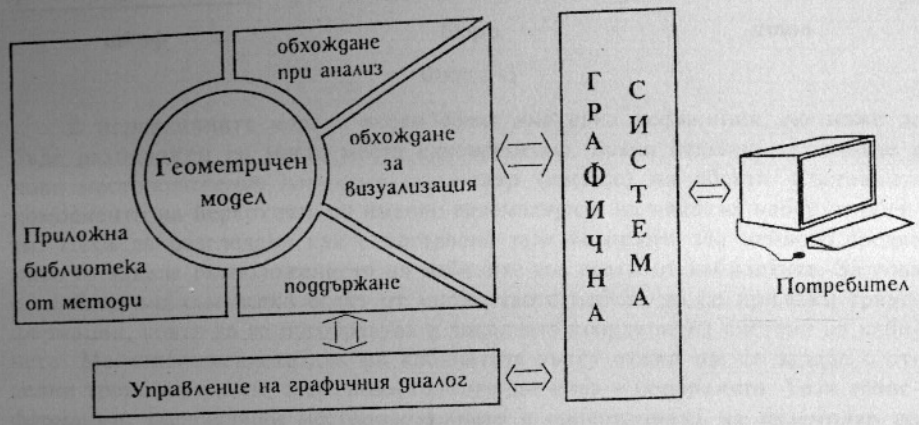
СИСТЕМИ ЗА ГЕОМЕТРИЧНО МОДЕЛИРАНЕ. От показания по-горе пример става ясно, че геометричният модел се проектира от приложния програмист. Графичните системи могат само да предлагат средства за моделиране (и то само някои), а приложната програма е тази, която създава модела. Приложната програма извършва следните основни дейности, в центъра на които стои геометричният модел:

- *поддържане*: създаване, изтриване и модифициране на елементи и връзки в модела;
- *обхождане за визуализация*: представяне на геометричната информация в модела за графичната система. Тази дейност може да бъде елементарна за модели, чиито обекти са самите графични примитиви на графичната система: отсечки, дъги и начупени. Тя може да бъде и сравнително сложна, ако обектите не се съхраняват във вид на графични примитиви, а всеки обект е сам по себе си алгоритъм за получаването на тези примитиви;
- *обхождане при търсене и анализ*: тази дейност е свързана с изпълнението на обработващите алгоритми, както и на някои специфични анализи върху него.

Поради важността на модела и при наличието на мощни средства за неговото поддържане по-сложните интерактивни приложни програми се наричат *системи за геометрично моделиране*. Освен изискванията за ефективно извършване на горните три дейности, в системите за геометрично моделиране се спазват и следните общи принципи:

- *правилност*: моделът да отразява вярно свойствата на реалните обекти, да не съдържа противоречиви данни;

- *мощност*: системата да позволява моделирането на достатъчно сложни реални обекти;
- *пълнота*: наличната информация да позволява изчисляването на разнообразни геометрични характеристики: площ, въртящ момент и др.
- *компактност*: възможност за ефективна програмна реализация;
- *отвореност*: възможност за добавяне на нови типове обекти, връзки и обработващи процедури.



Фиг. 5-1

Погрешно е да се мисли, че геометричният модел са само структурите от данни, които се използват за представянето на формата на компонентите. Структурите от данни, в които е описан един модел, са само част от неговото физическо ниво или *как* е извършено представянето. В тази глава ние ще обърнем по-голямо внимание на това *какви* са тези компоненти и връзките им (т.е. *какво* ще стои в тези структури от данни) и *какви* са възможните операции с тях, като се спрем по-подробно на някои от по-важните обработващи алгоритми.

Геометричните модели не са единствените модели в компютърната графика. Много често срещани са различните количествени модели: икономически, математически (системи от уравнения за описване на поведението на обектите), демографски, социологически и др. Друг вид модели са организационните: структури на организации, блок-схеми на процеси и др. Тези модели не съдържат геометрични данни. Те обаче могат да бъдат визуализирани така, че да може потребителят по-лесно да възприеме информацията, която те носят.

5.1 ГРАФИЧНИ СИСТЕМИ, ПРЕДОСТАВЯЩИ СРЕДСТВА ЗА МОДЕЛИРАНЕ

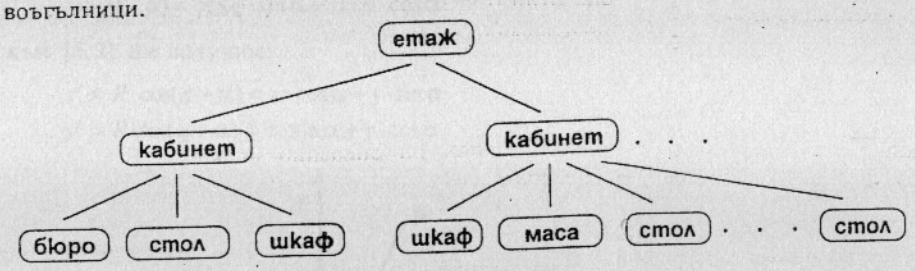
Показаната по-горе схема е малко по-различна когато една приложна програма е написана за графична система, в която има средства за моделиране. В този специален случай повечето от геометричната информация - метрика и топология - се съхранява от графичната система. В тази част на настоящата глава ще се спрем на средствата за геометрично моделиране в някои системи със запомняне на визуализираните обекти и на това как приложните графични програми използват тези средства.

5.1.1 Йерархични геометрични модели

Обектите в реалния свят рядко са неделими. Дори и в случай, че те са монолитни, ние ги моделираме като съставени от части, всяка от които има определена функционалност. Човешкото тяло е неделимо, но ние го описваме като структура от ръце, крака, глава и т.н. Особено ясно йерархичното разделяне на части се вижда в обектите, които човек сам създава. В проектирането тази йерархия е най-добре обособена, като там дори всяко ниво от йерархията носи отделно име: детайл, възел, агрегат. Използването на йерархично структурирани компоненти в един модел дава редица предимства:

- възможност за създаване на всеки от компонентите поотделно и независимо един от друг;
- сложните обекти се създават чрез просто сглобяване на съставни компоненти;
- всеки компонент може да бъде описан само веднъж, а включен като съставна част на много други обекти;
- възможност да се управлява изменението на обектите чрез изменението на техните компоненти.

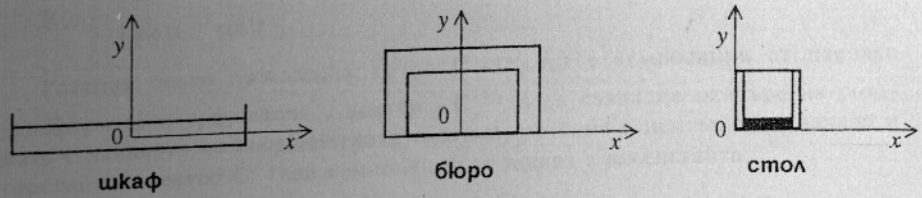
Нека да разгледаме един прост пример. При проектирането на вътрешното обзавеждане на административни сгради наборът от обекти, с които се работи, може да бъде ограничен до следните мебели: **маса**, **бюро**, **стол**, **шкаф** и **преграда**. Екземпляр от всеки отделен обект може да бъде разположен в обекта **кабинет**. Нека считаме, че един **етаж** от административната сграда се състои само от кабинети и в този пример тези кабинети са само изправени правоъгълници.



Фиг. 5-2

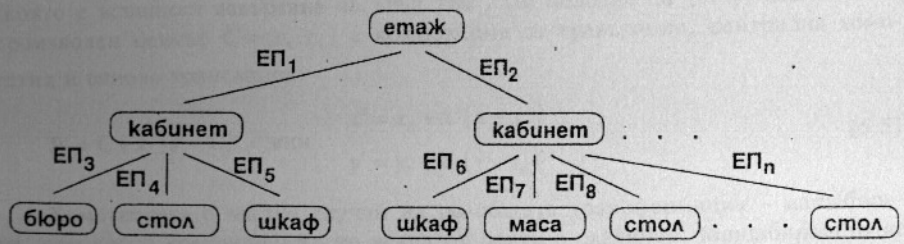
Йерархията в случая е с фиксирана дълбочина: **етаж - кабинет - мебели**. Всяко от листата може да се представи за нуждите на визуализацията чрез съвкупност от графични примитиви - отсечки и многоъгълници. Както казахме по-горе, всеки от компонентите е сравнително независим и неговото описание в графични примитиви не зависи от конкретното му местоположе-

ние. Ето защо всеки от мебелите трябва да се зададе в собствена координатна система: локална координатна система на обекта. Описанието на обект в термините на графични примитиви ще наричаме *дефиниция* на този обект.



Фиг. 5-3

В йерархичните модели всеки обект има една дефиниция, но може да бъде разположен на много места едновременно. Всяко отделно включване в ново местоположение наричаме *екземпляр* (instance) на обекта. Съставните компоненти на йерархията са именно екземплярите на някакъв набор от обекти. Нека да разгледаме как е построена тази йерархия. На нейното средно ниво се задава разположението на мебелите във всеки от кабинетите. За това е необходимо към всеки обект от множеството мебели да се приложи трансформация, която да го преобразува в локалната координатна система на кабинета. Местоположението пък на кабинетите върху етажа ще се зададе с отделни трансформации, свързващи горните две нива в йерархията. Тези трансформации, определящи местоположението и ориентацията на екземпляр на един обект в обект от по-горното ниво в йерархията наричаме *екземплярни преобразувания*. В някои системи се използва и името *моделна трансформация*. Това е трансформация само между две съседни нива.



Фиг. 5-4

Координатната система на най-горното ниво съвпада с потребителската координатна система. За да се визуализира един обект от листата (например **бюро**) в потребителското пространство е необходимо към графичните примитиви, които го изграждат и които са зададени в неговата локална координатна система да бъде приложена веригата от трансформации, които водят от него до корена на йерархията. Екземплярните преобразувания могат да бъдат най-разнообразни геометрични трансформации. Нека първо се спрем на тяхното представяне в геометричния модел.

5.1.2 Геометрични трансформации. Представяне

Читателят познава повечето геометрични трансформации от аналитичната геометрия: транслация, ротация, мащабиране, осева симетрия. Тук ние ще разгледаме тяхното представяне по еднотипен начин така, че да се улесни обхождането на йерархичния модел за нуждите на визуализацията.

ТРАНСЛАЦИЯ. Транслацията е успоредно преместване, което запазва дължините и ъглите на обектите. Транслацията се задава от вектор в равнината, който определя движение в определена посока и големина. Транслираният образ $\mathbf{P}'=(x',y')$ на една точка $\mathbf{P}=(x,y)$ се получава като сумираме с вектора $\mathbf{T}=(t_x,t_y)$ на транслацията:

$$\mathbf{P}'=\mathbf{P}+\mathbf{T}(t_x,t_y) \quad \text{или} \quad \begin{aligned} x' &= x+t_x \\ y' &= y+t_y \end{aligned} \qquad [5.1]$$

РОТАЦИЯ. Ротацията е движение, при което даден обект се завърта на определен ъгъл около зададен център. Ротацията също запазва размерите на обектите. Формулите за ротацията с център началото на координатната система можем да изведем сравнително просто.

Нека изразим координатите на една точка и нейния образ чрез радиус-векторите им, ъглите, които те сключват с оста Ox и ъгъла на ротация. Тъй като радиус-векторът и на двете точки е един и същ, както се вижда на фиг. 5-5, в сила са следните равенства:

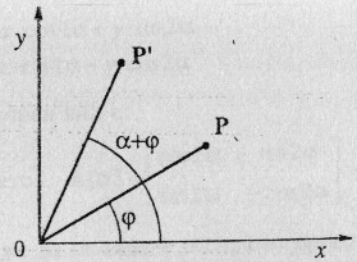
$$\begin{aligned} x &= R\cdot\cos\varphi & x' &= R\cdot\cos(\varphi+\alpha) \\ y &= R\cdot\sin\varphi & y' &= R\cdot\sin(\varphi+\alpha) \end{aligned} \qquad [5.2]$$

Като приложим тригонометричните тъждества:

$$\begin{aligned} \cos(\varphi+\alpha) &= \cos\varphi\cdot\cos\alpha-\sin\varphi\cdot\sin\alpha \\ \sin(\varphi+\alpha) &= \cos\varphi\cdot\sin\alpha+\sin\varphi\cdot\cos\alpha \end{aligned} \qquad [5.3]$$

към [5.2] ще получим:

$$\begin{aligned} x' &= R\cdot\cos(\varphi+\alpha)=x\cdot\cos\alpha-y\cdot\sin\alpha \\ y' &= R\cdot\sin(\varphi+\alpha)=x\cdot\sin\alpha+y\cdot\cos\alpha \end{aligned}$$



Фиг. 5-5

Горните формули имат и векторен израз: $\mathbf{P}' = \mathbf{R}(\alpha)\cdot\mathbf{P}$, в който ротацията се задава в матрична форма:

$$\mathbf{R}(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}$$

Ротация около произволна точка $\mathbf{C} = (c_x, c_y)$ е композиция от няколко трансформации: транслация с вектор $-\mathbf{C}$ - за да съвпадне центъра на ротацията с началото на координатната система, ротация с център това начало и транслация с вектор $\mathbf{C}$. Тази композиция се задава с равенствата:

$$\mathbf{P}' = \mathbf{C} + \mathbf{R}(\alpha)\cdot(\mathbf{P}-\mathbf{C}) \quad \text{или} \quad \begin{aligned} x' &= x_c + (x-x_c)\cdot\cos\alpha - (y-y_c)\cdot\sin\alpha \\ y' &= y_c + (x-x_c)\cdot\sin\alpha + (y-y_c)\cdot\cos\alpha \end{aligned} \qquad [5.4]$$

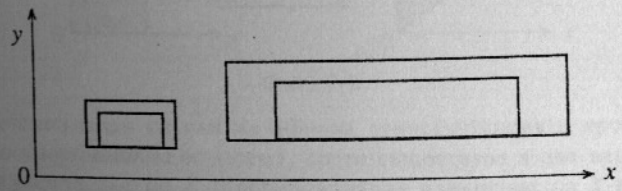
ХОМОТЕТИЯ И МАЩАБИРАНЕ. Хомотетията е преобразуване, което променя размерите на обекта - разтяга или свива обекта по протежение на координатните оси съобразно стойността на даден коефициент k. Тя обаче запазва съотношението на дължините в обекта и неговия трансформиран образ. Когато разтягането или свиването се извършва спрямо центъра на координатната система образът на една точка се получава при просто умножение на всяка от координатите с коефициента на хомотетия:

$$\mathbf{P}' = k\cdot\mathbf{P} \quad \text{или} \quad \begin{aligned} x' &= k\cdot x \\ y' &= k\cdot y \end{aligned}$$

Очевидно хомотетията запазва размерите на обектите ако $|k| = 1$. При $k = 1$ тя е идентично преобразуване, а при $k = -1$ е централна симетрия (която е всъщност завъртане на ъгъл 180°. По подобие на [5.4], хомотетия с произволен център $\mathbf{C} = (c_x, c_y)$ е композиция от транслация, централна хомотетия и отново транслация:

$$\mathbf{P}' = \mathbf{C} + k\cdot(\mathbf{P}-\mathbf{C}) \quad \text{или} \quad \begin{aligned} x' &= x_c + k\cdot(x-x_c) \\ y' &= y_c + k\cdot(y-y_c) \end{aligned} \qquad [5.5]$$

Хомотетията е частен случай на по-общата трансформация - мащабиране, при която коефициентът по всяка от осите е различен. Мащабирането не запазва съотношението на дължините - чрез него една окръжност се трансформира в елипса.



Фиг. 5-6

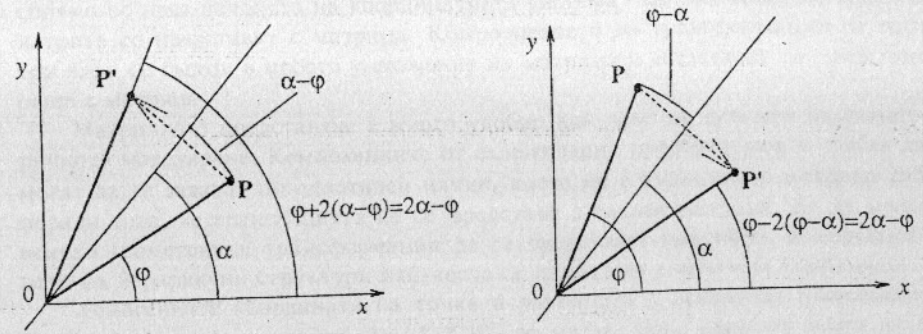
При наличието на два коефициента мащабирането, подобно на ротацията, се задава с матрица:

$$\mathbf{P}' = \mathbf{S}(k_x, k_y)\cdot\mathbf{P} \quad \text{където} \quad \mathbf{S}(k_x, k_y) = \begin{bmatrix} k_x & 0 \\ 0 & k_y \end{bmatrix}, \quad \text{или} \quad \begin{aligned} x' &= k_x\cdot x \\ y' &= k_y\cdot y \end{aligned}$$

Формулите за мащабиране с произволен център са подобни на [5.5]:

$$\mathbf{P}' = \mathbf{C} + \mathbf{S}(k_x, k_y)\cdot(\mathbf{P}-\mathbf{C}) \quad \text{или} \quad \begin{aligned} x' &= x_c + k_x\cdot(x-x_c) \\ y' &= y_c + k_y\cdot(y-y_c) \end{aligned} \qquad [5.6]$$

ОСЕВА СИМЕТРИЯ. Осевата симетрия е преобразуване, което запазва дължините в обекта и при което се създава огледален образ спрямо зададена ос в равнината. Нека разгледаме ос, която минава през началото на координатната система и сключва определен ъгъл α с оста Ox:



Фиг. 5-7

Ако представим една точка и нейния образ при такава осева симетрия чрез ъглите, които те сключват с оста Ox, подобно на [5.2] ще получим:

$$\begin{aligned} x &= R\cdot\cos\varphi & x' &= R\cdot\cos(2\alpha-\varphi) \\ y &= R\cdot\sin\varphi & y' &= R\cdot\sin(2\alpha-\varphi) \end{aligned} \qquad [5.7]$$

Използвайки отново тъждествата [5.3] ще получим следните формули за образа на точката при такава осева симетрия:

$$\begin{aligned} x' &= R\cdot\cos(2\alpha-\varphi) = x\cdot\cos 2\alpha + y\cdot\sin 2\alpha \\ y' &= R\cdot\sin(2\alpha-\varphi) = x\cdot\sin 2\alpha - y\cdot\cos 2\alpha \end{aligned},$$

което записано във векторен вид е:

$$\mathbf{P}' = \mathbf{A}(\alpha)\cdot\mathbf{P} \quad \text{където} \quad \mathbf{A}(\alpha) = \begin{bmatrix} \cos 2\alpha & \sin 2\alpha \\ \sin 2\alpha & -\cos 2\alpha \end{bmatrix}. \qquad [5.8]$$

Осева симетрия може да се зададе и спрямо произволна ос. Нека за удобство тази ос е зададена с точка $\mathbf{C} = (c_x, c_y)$, през която тя минава и единичен

вектор $\mathbf{Q}=(a,b)$ върху тази ос. Елементите на матрицата A можем да изразим чрез координатите на единичния вектор, използвайки тъждествата за тригонометричните функции от двоен ъгъл:

$$\cos 2\alpha = \cos^2\alpha - \sin^2\alpha = a^2 - b^2$$
$$\sin 2\alpha = 2\sin\alpha\cdot\cos\alpha = 2ab$$

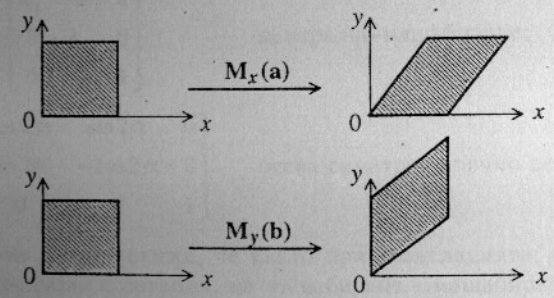
Подобно на [5.4], [5.5] и [5.6] ще изразим произволната осева симетрия като композиция от транслация, за да разположим оста през началото, осевата симетрия, зададена с [5.8] и обратната транслация, за да върнем оста в изходното ѝ положение:

$$\mathbf{P}' = \mathbf{C} + \mathbf{A}(a,b)\cdot(\mathbf{P}-\mathbf{C}), \quad \text{където} \quad \mathbf{A}(a,b) = \begin{bmatrix} a^2-b^2 & 2ab \\ 2ab & b^2-a^2 \end{bmatrix}. \qquad [5.9]$$

ДРУГИ ТРАНСФОРМАЦИИ. Най-общата линейна трансформация е *афинното преобразуване*. То се задава с уравненията:

$$\begin{aligned} x' &= a\cdot x + b\cdot y + e \\ y' &= c\cdot x + d\cdot y + f \end{aligned} \quad \text{или} \quad \mathbf{P}' = \mathbf{M}\cdot\mathbf{P} + \mathbf{T}, \quad \text{където} \quad \mathbf{M} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}. \qquad [5.10]$$

Тъй като в компютърната графика се използват обратими преобразувания, се поставя условието детерминантата на матрицата **M** да е различна от нула. Това осигурява обратимостта на трансформацията (съществуването на обратна на **M** матрица). Тази трансформация има много разновидности и се използва често в такива области като компютърната анимация и илюстративната графика. В общия случай тя не запазва дължините и ъглите на трансформираните обекти, но запазва успоредността на правите. Афинните преобразувания в йерархичните модели рядко се използват самостоятелно - те по-скоро се получават при композирането на по-простите трансформации, които представихме по-горе.



Фиг. 5-8

Един по-специален случай на афинна трансформация е *пропорционално разтягане по координатна ос* (shear), което съществува в два варианта - *пропорционално разтягане по X* (пропорционално изменение на *X*-координатите

като функция на *Y*) и *пропорционално разтягане по Y* (изменение на *Y* като функция на *X*). Подобно на ротацията с център началото и централното мащабиране, тези трансформации се задават матрично, а матриците им са съответно:

$$M_x(a) = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \quad \text{и} \quad M_y(b) = \begin{bmatrix} 1 & 0 \\ b & 1 \end{bmatrix}$$

където параметрите a и b задават коефициента на разтягането по всяка от координатните оси (фиг. 5-8).

ПРЕДСТАВЯНЕ В ХОМОГЕННИ КООРДИНАТИ. Горните трансформации са най-често срещаните в моделирането. Те могат да се композират по разнообразен начин, както показахме при ротация с произволен център, мащабиране с произволен център и осева симетрия спрямо произволна ос. Трите трансформации - централна ротация, централно мащабиране и осева симетрия спрямо ос през началото на координатната система - си приличат по това, че и трите се представят с матрица. Композирането на трансформации от тези три вида се състои в просто умножение на матрици и естествено се представя също с матрица.

Матричното представяне е много удобно най-вече за нуждите на геометричното моделиране. Композициите от екземплярни преобразувания трябва да могат да се задават по еднотипен начин, което не е възможно с матрица 2×2 поради това, че транслацията не се представя с такава матрица. За да могат всички геометрични трансформации да се представят матрично, в моделирането на йерархични структури най-често се използват хомогенни координати.

Хомогенните координати на точка в равнината с декартови координати $\mathbf{P}=(x,y)$ е наредена тройка $\mathbf{P}=(X,Y,W)$ от числа, поне едно от които не е нула, свързани с нехомогенните координати на точката със следните равенства:

$$x = \frac{X}{W}, \qquad y = \frac{Y}{W}, \qquad \text{при} \quad W \neq 0 \qquad [5.11]$$

От тези равенства се вижда, че всяка точка в равнината има безброй много такива наредени тройки, координатите на които са кратни. Най-просто можем да преобразуваме нехомогенните (декартови) координати на една точка в хомогенни чрез добавяне на единица за нейна хомогенна координата – $\mathbf{P}=(x,y,1)$, което оставя в сила равенствата [5.11]. Когато хомогенната координата на една точка е нула, тя се нарича *безкрайна точка* и представя снопа успоредни прави имащи направлението, зададено векторно от първите две координати. Безкрайните точки имат някои интересни приложения в изчислителната геометрия, но тук ние ще разглеждаме хомогенни координати само на точки, които имат декартови координати.

Ще отбележим, че в моделирането точките се представят в хомогенни координати само за изпълнението на геометричните трансформации, а се съхраняват в структурите от данни в своите нехомогенни координати. Трансла-

цията, която няма матрично представяне в нехомогенни координати, се задава със следната матрица:

$$\mathbf{T}(t_x,t_y)=\begin{bmatrix}1 & 0 & t_x\\ 0 & 1 & t_y\\ 0 & 0 & 1\end{bmatrix} \quad \text{защото} \quad \begin{bmatrix}1 & 0 & t_x\\ 0 & 1 & t_y\\ 0 & 0 & 1\end{bmatrix}\cdot\begin{bmatrix}x\\ y\\ 1\end{bmatrix}=\begin{bmatrix}x+t_x & y+t_y & 1\end{bmatrix},$$

което е точно образът на точката при транслация, изразен в хомогенни координати. Обратната матрица е точно матрицата на транслацията с противоположния вектор, което можем лесно да проверим.

$$\mathbf{T}(t_x,t_y)^{-1}=\mathbf{T}(-t_x,-t_y)=\begin{bmatrix}1 & 0 & -t_x\\ 0 & 1 & -t_y\\ 0 & 0 & 1\end{bmatrix} \quad \text{защото}$$

$$\begin{bmatrix}1 & 0 & -t_x\\ 0 & 1 & -t_y\\ 0 & 0 & 1\end{bmatrix}\cdot\begin{bmatrix}1 & 0 & t_x\\ 0 & 1 & t_y\\ 0 & 0 & 1\end{bmatrix}=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

Матриците на останалите три трансформации, които се дефинират като умножение с матрица се задават с просто допълване на съответната нехомогенна матрица с нули и поставяне на единица като елемент на последния ред и последния стълб в нея. Читателят може сам лесно да провери валидността на техните матрични уравнения в хомогенни координати.

$$\mathbf{R}(\alpha)=\begin{bmatrix}\cos\alpha & -\sin\alpha & 0\\ \sin\alpha & \cos\alpha & 0\\ 0 & 0 & 1\end{bmatrix}$$ ротация с център началото;

$$\mathbf{S}(k_x,k_y)=\begin{bmatrix}k_x & 0 & 0\\ 0 & k_y & 0\\ 0 & 0 & 1\end{bmatrix}$$ централно мащабиране;

$$\mathbf{A}(\alpha)=\begin{bmatrix}\cos 2\alpha & \sin 2\alpha & 0\\ \sin 2\alpha & -\cos 2\alpha & 0\\ 0 & 0 & 1\end{bmatrix}$$ осева симетрия спрямо ос през началото.

Лесно може да се покаже, че както при транслацията, обратната трансформация на ротация е ротация, на мащабиране - мащабиране и на осева симетрия - също осева симетрия. Вече е сравнително лесно да представим всяка от трансформациите, зададени с [5.4], [5.6] и [5.9], като уравнение от вида $\mathbf{P}'=\mathbf{M}\cdot\mathbf{P}$, използвайки умножение на матрици.

Нека покажем това за ротация с произволен център (виж уравнение [5.4]):

$$\mathbf{P}'=\mathbf{C}\cdot\mathbf{R}(\alpha)\cdot\mathbf{C}^{-1}\cdot\mathbf{P}=\begin{bmatrix}1 & 0 & c_x\\ 0 & 1 & c_y\\ 0 & 0 & 1\end{bmatrix}\cdot\begin{bmatrix}\cos\alpha & -\sin\alpha & 0\\ \sin\alpha & \cos\alpha & 0\\ 0 & 0 & 1\end{bmatrix}\cdot\begin{bmatrix}1 & 0 & -c_x\\ 0 & 1 & -c_y\\ 0 & 0 & 1\end{bmatrix}\cdot\mathbf{P}=$$

$$=\begin{bmatrix}\cos\alpha & -\sin\alpha & c_x-c_x\cos\alpha+c_y\sin\alpha\\ \sin\alpha & \cos\alpha & c_y-c_x\sin\alpha+c_y\cos\alpha\\ 0 & 0 & 1\end{bmatrix}\cdot\mathbf{P}$$

Матричната форма на представянето е удобна и поради факта, че дава общ метод за намиране на обратната трансформация - трансформацията с обратната матрица.

Композирането на геометричните трансформации не е винаги комутативно, тъй както не е комутативно и умножението на матрици. Само някои трансформации комутират (могат да бъдат приложени в произволен ред) - две транслации, две ротации с общ център, две мащабирания с общ център и др. За останалите, последователността при прилагането им - реда на умножаване на матриците е от съществено значение.

Всички разгледани дотук хомогенни матрици, представящи геометрични трансформации, имат специален вид: техният последен ред е винаги един и същ - [0 0 1]. Това може да се използва при кодирането на алгоритмите за работа с тези матрици, тъй като трансформирането с такава матрица може да се извършва не като общо умножение на матрица и вектор, а като афинна трансформация с уравненията [5.10] за матрица от вида:

$$\mathbf{M}=\begin{bmatrix}a & b & e\\ c & d & f\\ 0 & 0 & 1\end{bmatrix}$$

Геометричните трансформации могат да се представят и по други начини - например с линейни комплексни уравнения, което е удобно за решаването на някои задачи в изчислителната геометрия. Ние тук ще се ограничим само с разглеждането на хомогенното матрично представяне, тъй като то е основното представяне на трансформациите в йерархичните модели.

5.1.3 Функции за моделиране в графичните системи

В графичните системи, които предлагат средства за изграждане на йерархични геометрични модели, всеки отделен обект се дефинира в собствена координатна система. Разполагането на екземпляр от него в обект от по-горно ниво се задава с неговото *екземплярно преобразуване*, което всъщност е трансформация от локалната координатната система на включвания обект в тази на обекта от по-високото ниво.

В края на трета глава разгледахме възможностите в една графична система за структуриране на изображението. Когато структурирането се извършва само на едно ниво, системата предлага възможности за дефиниране и работа със сегменти. Ако графичната система позволява включването на сег-

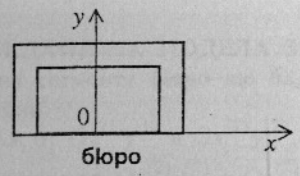
ментите един в друг, те могат да се използват за представянето на обектите от йерархичните структури, които разглеждаме тук. За целта са необходими основно две допълнителни функции:

- за включване на екземпляр от един сегмент в друг сегмент;
- за задаване на неговото екземплярно преобразуване.

ФУНКЦИИ ЗА СТРУКТУРИРАНЕ НА СЕГМЕНТИ. Тук ще запазим името *сегмент* за група от графични примитиви, която може да включва в себе си други такива групи, макар че в подобни системи (например PHIGS) се използва името *структури*, за да се отличават те от неструктурираните сегменти (например тези в системата GKS). Функциите, които приложният програмист може да използва за осъществяването на горните две действия са:

```
IncludeSegmentInstance(SegId)
SetInstanceTransformation(matrix)
```

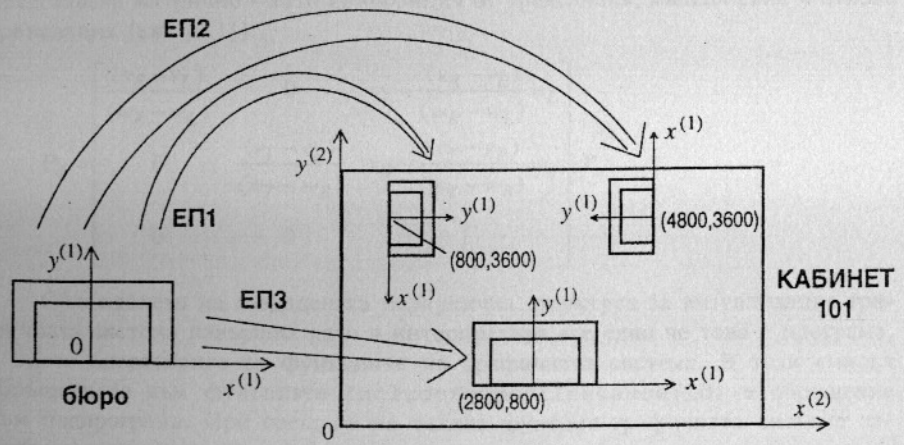
Нека покажем тяхното използване с един конкретен пример. Ако обектът **бюро** е дефиниран като сегмент, състоящ се от два правоъгълника по следния начин:




```
OpenSegment(DESK);
  Rectangle(-90,0,90,90);
  Rectangle(-60,0,60,60);
CloseSegment();
```

Фиг. 5-9

то за да се разположи екземпляр от този обект в един кабинет е необходимо да се извърши трансформация между координатните системи на **бюрото** и **кабинета** както е показано на фиг. 5-10.



Фиг. 5-10

Екземплярното преобразуване ЕП1 в случая е композиция от хомотетия, ротация на 270° и транслация, и може да бъде представено като 3×3 матрица. То преобразува координатите на всички точки (в случая върховете на правоъгълниците) от координатната система $Ox^{(1)}y^{(1)}$ в $Ox^{(2)}y^{(2)}$.

Тази трансформация не е част от дефиницията на обекта **бюро**. Тя се отнася до неговия екземпляр и задава как този екземпляр изгражда един конкретен обект от по-горното ниво – нека го наречем **кабинет 101**. Този обект е също сегмент и горното екземплярно преобразуване е част от неговата дефиниция. Ако в този кабинет са разположени три бюра, дефиницията му ще изглежда по следния начин:

```
OpenSegment(Room101);
  Rectangle(0,0,6000,4500);
  SetInstanceTransformation(EP1);
  IncludeSegmentInstance(DESK);
  SetInstanceTransformation(EP2);
  IncludeSegmentInstance(DESK);
  SetInstanceTransformation(EP3);
  IncludeSegmentInstance(DESK);
CloseSegment();
```

Матриците на горните трансформации се задават в хомогенни координати (3×3) директно при използването на специален тип променливи `Matrix`, или като се използват допълнителни функции за работа с матрици:

- получаване на матриците на основните геометрични трансформации чрез задаване на параметрите на трансформацията - вектора на транслация, ъгъла на ротация, оста на симетрия:

```
FormTranslateMatrix(tx,ty,EP)
FormRotateMatrix(xc,yc,ang,EP)
FormScaleMatrix(xc,yc,kx,ky,EP)
FormMirrorMatrix(x1,y1,x2,y2,EP)
```

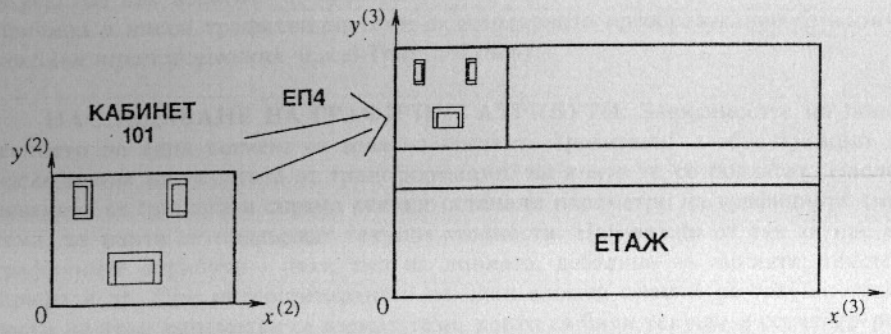
- композиция на изображения - умножаване на матрици:

```
MultiplyMatrix(M1,M2,Mresult)
```

Да дефинираме сегмента **кабинет 101** с помощта на тези функции:

```
. . .
  FormRotateMatrix(0,0,270,&M1);          /* задаване на ЕП1 */
    FormScaleMatrix(0,0,0.6,0.6,&M2);
    FormTranslateMatrix(800,3600,&M3);
    MultiplyMatrix(M2,M1,&EP1);             /* мащабиране с М2 */
    MultiplyMatrix(M3,EP1,&EP1);            /* транслация с М3 */
  FormRotateMatrix(0,0,90,&M1);           /* задаване на ЕП2 */
    FormTranslateMatrix(4800,3600,&M3);
    MultiplyMatrix(M2,M1,&EP2);             /* мащабиране с М2 */
    MultiplyMatrix(M3,EP2,&EP2);            /* транслация с М3 */
  FormScaleMatrix(0,0,0.8,0.8,&M1);       /* задаване на ЕП3 */
    FormTranslateMatrix(2800,800,&M2);
    MultiplyMatrix(M2,M1,&EP3);             /* транслация с М2 */
OpenSegment(Room101);
. . .
CloseSegment();
```

Екземпляр от така дефинирания сегмент **кабинет 101** може да бъде разположен в обекта **етаж**, който в разглеждания случай (фиг. 5-2) е най-горното ниво в йерархията. Задаването на местоположението му става с ново екземплярно преобразуване, което този път трансформира точките от координатната система $Ox^{(2)}y^{(2)}$ в $Ox^{(3)}y^{(3)}$.



Фиг. 5-11

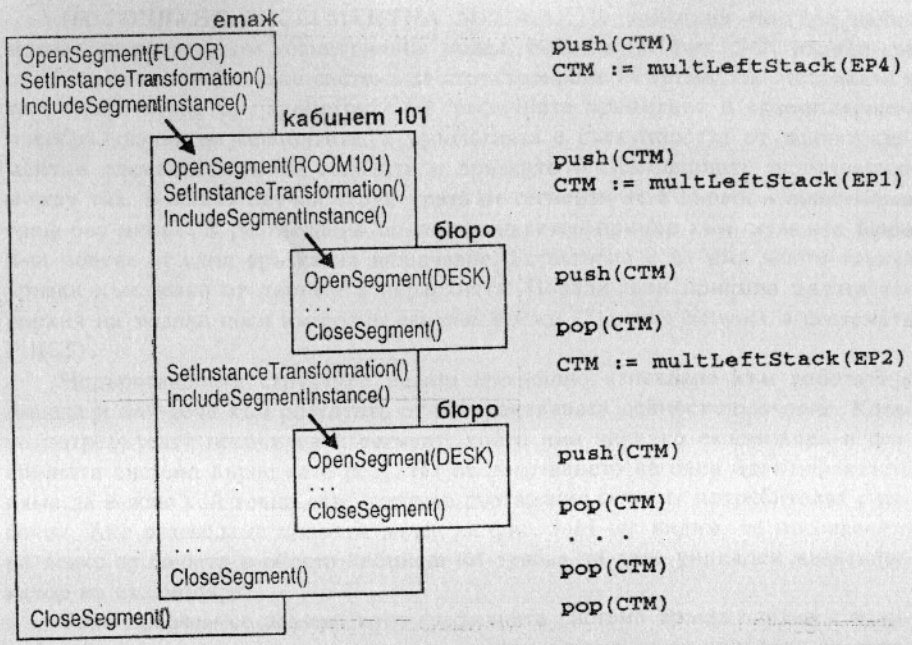
ОБХОЖДАНЕ НА МОДЕЛА ЗА ВИЗУАЛИЗАЦИЯ. Точките от всеки екземпляр на сегмента **бюро** ще бъдат подложени на двете последователни трансформации:

- веднъж от $Ox^{(1)}y^{(1)}$ в $Ox^{(2)}y^{(2)}$, за да се разположи той в **кабинета**,
- а след това от $Ox^{(2)}y^{(2)}$ в $Ox^{(3)}y^{(3)}$ - за да се разположи и в **етажа**.

Последната координатна система е потребителската, така че за окончателната визуализация остава координатите на двата правоъгълника на бюрото да се трансформират до нормираната координатна система на дисплея. Тази трансформация е *трансформацията на изгледа*, която също може да бъде представена матрично - като композиция от транслация, мащабиране и отново транслация (вж. [3.1]):

$$\mathbf{P}_d = \begin{bmatrix} \frac{(v_R - v_L)}{(w_R - w_L)} & 0 & v_L - \frac{(v_R - v_L)}{(w_R - w_L)} w_L \\ 0 & \frac{(v_T - v_B)}{(w_T - w_B)} & v_B - \frac{(v_T - v_B)}{(w_T - w_B)} w_B \\ 0 & 0 & 1 \end{bmatrix} \cdot \mathbf{P}_u$$

Обхождането на създадената йерархична структура за визуализация графичната система извършва като я интерпретира все едно че това е програма, в която операторите са функциите на графичната система. В този смисъл обръщението към функцията `IncludeSegmentInstance(id)` е обръщение към подпрограма. При срещане на такава функция графичната система започва да интерпретира дефиницията на вложения сегмент.



Фиг. 5-12

За осигуряване на разполагането на екземпляри на сегментите съобразно техните екземплярни преобразувания графичната система поддържа т.нар. *матрица на текущото преобразуване* (Current Transformation Matrix - CTM) и стек на преобразуванията, който се управлява по следния начин:

- При започване на обхождането в CTM се записва матрицата на трансформацията на изгледа;
- При обръщение към функция за отваряне на сегмент CTM се записва в стека;
- При обръщение към функция за затваряне на сегмент от върха на стека се извлича матрицата и се записва в CTM;
- При обръщение към функция за задаване на екземплярно преобразуване матрицата на екземплярното преобразуване се умножава отляво с матрицата, която се намира на върха на стека (без да се променя самият стек) и резултатът се записва в CTM.
- Всеки графичен примитив се подлага на трансформацията, зададена от CTM преди да се визуализира.

На фиг. 5-12 е показана интерпретацията на структурата, която представя примерния геометричен модел от фиг. 5-11.

Горният алгоритъм осигурява подходящо трансформиране на всички обекти в зависимост от тяхното местоположение в йерархичната структура. Съхраняването на CTM при започване на обработката на нов сегмент и възстановяването ѝ след завършване на интерпретацията му осигурява възмож-

ност за разполагане на отделните компоненти на всеки сегмент независимо един от друг. Нещо повече, екземплярните преобразувания променят положението на всички сегменти, които са вложени в даден сегмент (неговите "деца"), но не могат по никакъв начин да повлияят на сегментите, които са разположени по-горе от него в йерархията или се намират в други клонове от нея. В този смисъл екземплярното преобразуване е локално за един сегмент и участва във веригата от преобразувания само на неговите "деца". По тази причина в някои графични системи екземплярното преобразуване се нарича *локална трансформация* (Local Transformation).

НАСЛЕДЯВАНЕ НА ГРАФИЧНИ АТРИБУТИ. Зависимостта на положението на един сегмент от това на неговите "родители" е общ принцип за *наследяване* на веригата от трансформации, на които те се подлагат. Наследяването се прилага и спрямо всички останали параметри на графичната система, за които се поддържат текущи стойности. Най-важни от тях за нас са графичните атрибути - цвят, тип на линията, дебелина на линията, текстов шрифт и др. При интерпретирането на един вложен сегмент за текущи стойности на тези параметри се вземат тези, които са били текущи в сегмента-родител непосредствено преди започване на интерпретацията на вложения сегмент. Промяната на стойността им във вложен сегмент обаче не се отразява на сегмента-родител, а остава локална. Атрибутите в графичната система имат аналог в програмирането - подобни свойства имат *динамичните променливи* в езика Lisp.

Начинът, по който се реализира наследяването на графичните атрибути, прилича на работата с СТМ. При отваряне на нов вложен сегмент състоянието на графичната система се запазва в стек, от който то се възстановява след завършване на интерпретацията на този сегмент. По този начин се постига естествено наследяване на един атрибут - той е глобален за всички вложени в него екземпляри на сегменти, но е локален за всички останали. Съхраняването на състоянието на графичната система може да изисква много ресурси, поради което в някои графични системи се дава алтернатива на функцията `IncludeSegmentInstance` за включване на сегмент:

```
ReferSegmentInstance(Segid)
```

При използване на тази функция `OpenSegment` и `CloseSegment` на включения сегмент не запазват и съответно не възстановяват състоянието на графичната система. В този случай графичните атрибути са глобални променливи. Това може да се използва, за да се позволи промяната на графичен атрибут в сегмента-дете да се отрази на сегмента-родител.

Накрая ще отбележим, че графичните системи за моделиране предлагат и допълнителни функции за задаване на разнообразни неграфични данни за всеки от дефинираните сегменти. Тези данни могат да бъдат извличани при обхождане на йерархията при извършване на анализ върху геометричния модел. За тях се прилага същият принцип за наследяване както и за графичните атрибути.

ПОСОЧВАНЕ В СЕГМЕНТНА МРЕЖА. По описания по-горе начин можем да представим геометричния модел, показан на фиг. 5-2, изцяло със средствата на графичната система за структуриране на сегменти. Метриката в това представяне са параметрите на графичните примитиви и екземплярните преобразувания на сегментите, а топологията е съвкупността от всички сегментни елементи, самите сегменти и връзките (екземплярните включвания) между тях. В общия случай структурата от сегменти не е дърво, а ориентиран граф без цикъл. В разгледания по-горе конкретен пример към сегмента **бюро** има повече от една връзка на включване. Естествено е да има много такива връзки към всяко от листата в йерархията. Поради тази причина цялата йерархия на модела носи името *сегментна мрежа* (Structure Network в системата PHIGS).

Недървовидната структура налага специално изискване към работата с модела и най-вече към резултата от интерактивната дейност *посочване*. Когато потребителят посочи един сегмент, който има няколко екземпляра и графичната система върне като резултат от посочването неговия идентификатор, няма да е ясно кой точно екземпляр на съответния сегмент потребителят е посочил. Ако разгледаме примера даден на фиг. 5-11 ще видим, че посочването на всяко от бюрата в обекта **кабинет 101** трябва да даде уникален идентификатор на екземпляра.

Този проблем се решава като графичната система връща заедно с идентификатора на сегмента и *пътя в йерархията* (веригата от родители от корена до него), по който се е стигнало до този екземпляр. Това обаче не е достатъчно, ако в един сегмент са включени екземпляри на един и същ сегмент. Всяко от бюрата в горния пример ще има един и същ път:

```
FLOOR -> ROOM101 -> DESK
```

За да се отличават отделните екземпляри на един и същ сегмент, които се намират на едно и също ниво в йерархията, е необходимо освен идентификаторите на всички родители да се даде и номерът на елемента във всяка от дефинициите на по-горното ниво. По този начин горните пътища ще имат вида:

```
FLOOR (1) -> ROOM101 (1) -> DESK
FLOOR (1) -> ROOM101 (2) -> DESK
FLOOR (1) -> ROOM101 (3) -> DESK
```

където в скоби е отбелязан поредният номер на посочения вложен сегмент. Този номер е същият, който се използва при обръщение към функциите за редактиране на сегмент.

Този начин би бил удобен, ако сегментната мрежа няма да се променя. Ако обаче от горната йерархия премахнем екземпляра на бюрото, разположено с екземплярно преобразуване ЕП1, то *идентифициращите пътища* на останалите две бюра биха се променили. Ето защо е по-удобно приложният програмист сам да може да задава идентификатора, който иска да получава при посочването на даден елемент. Функцията

```
SetPickIdentifier(pickID)
```

задава този идентификатор и го свързва с елемента, непосредствено следващ тази функция в дефиницията на сегмента по подобие на функцията за задаване на *етикети*, която разгледахме в края на трета глава. Когато следващата функция е тази за включване - IncludeSegmentInstance, приложният програмист задава *идентификатора при посочване* (pick identifier) на съответния екземпляр на сегмент. Идентификаторите не се наследяват като графичните атрибути - те са уникални етикети на сегментите и графичните примитиви. Някои графични системи връщат както идентификатора, така и *идентифициращия път* при посочване на елемент.

5.1.4 Несъвършенство на моделирането със сегменти

Моделирането с представените средства има едно голямо предимство. Не само метриката, но и топологията на сегментната мрежа може да се променя динамично, тъй като всяка от функциите за включване е елемент от дефиницията на сегмента, в който се извършва включването. В края на трета глава показахме начините, по които елементите на един сегмент могат да бъдат изтривани и заменяни. В случай, че тези елементи са екземплярни включвания (т.е. съществената част от топологията на модела), тяхното изтриване или заменяне е еквивалентно на изтриване и създаване на нови връзки между компонентите на модела.

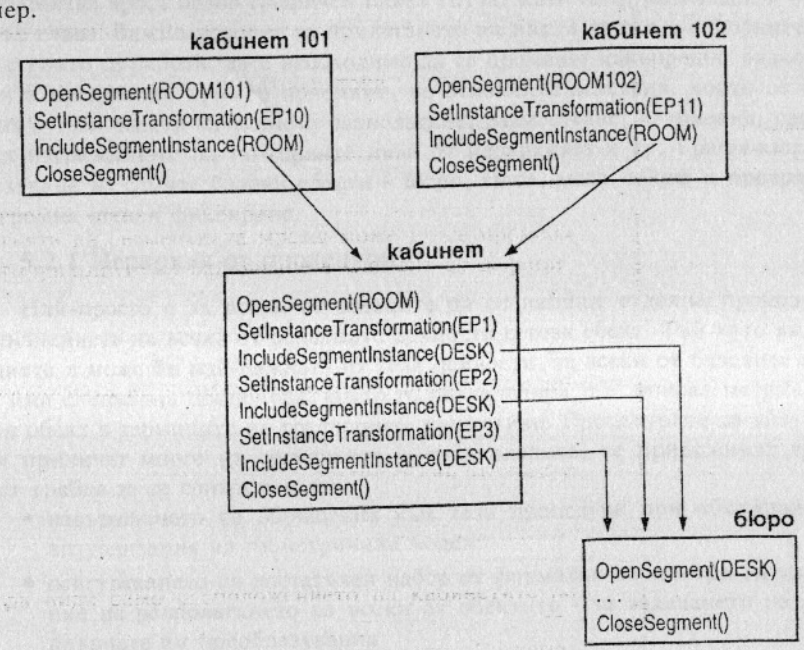
Моделирането със сегменти има и редица недостатъци. Един от тях е свързан непосредствено с визуализацията и то с една от особеностите на метриката, на която обърнахме внимание в началото на тази глава - визуализирането ѝ с различно ниво на детайлност. Ако при работа с модела от горния пример потребителят желае да види само разположението на кабинетите на етажа, без да се интересува как всеки от тях е обзаведен, приложната програма трябва да може да визуализира само част от сегментната мрежа. Системата, която описахме няма да позволи това да се направи без да се промени геометричният модел. Очевидно промяна на геометричния модел тук не е нужна, а по скоро е нужно в геометричния модел да се укаже кои сегменти да не се визуализират при определени обстоятелства или да се визуализират само схематично.

Освен избягване на претрупването на изображението, това би позволило значително ускоряване на визуализацията на целия геометричен модел. Някои системи (например PHIGS+) предлагат функции за условно включване на сегмент. При обхождане за визуализация такъв сегмент и всички включени в него сегменти ще се интерпретират само ако са изпълнени зададени условия.

В много случаи визуализацията на сегментите трябва да се избягва автоматично, без приложният програмист да е взел специални мерки за това при създаването на модела. Типичен пример е визуализацията на обекти, които са толкова малки, че се редуцират до група от пиксели върху един растерен дисплей. Удобно е също вместо избягване на визуализацията въобще, да се покаже правоъгълната обвивка на сегмента, който не е нужно да се визуализира напълно.

Едно от най-големите неудобства на моделирането със структурирани

сегменти е при представянето на еднотипни обекти. Със средствата, които описахме тук, не е възможно да създадем например обобщено описание на обекта **кабинет**, който да се състои от определени мебели, а отделните екземпляри **кабинет 101**, **кабинет 102** и т.н. да се отличават само по вътрешното разположение на тези мебели. Ще поясним казаното с един още по-прост пример.



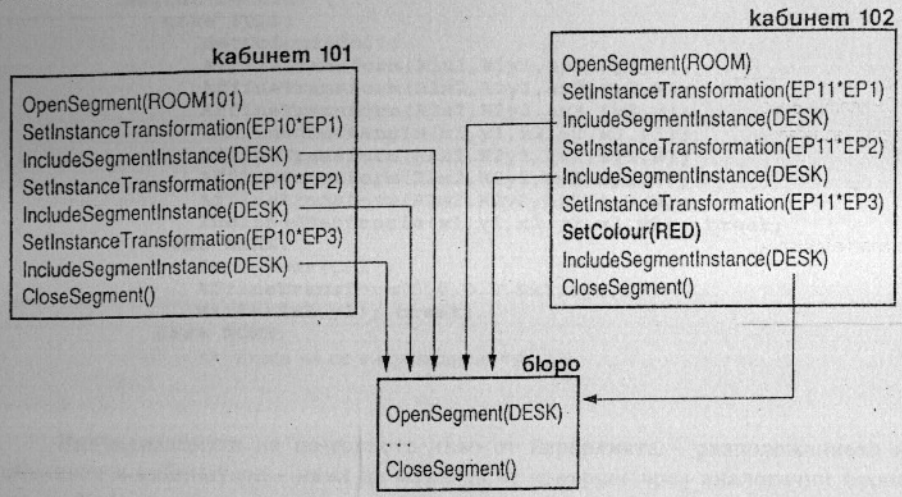
Фиг. 5-13

Нека броят, видът и разположението на мебелите в два обекта - **кабинет 101** и **кабинет 102** - да са едни и същи. За да представи тази еднаквост, приложният програмист трябва да дефинира специален сегмент **кабинет**, разположението на мебелите в който е фиксирано, а сегментите **кабинет 101** и **кабинет 102** да включват екземпляр от сегмента **кабинет** с различно екземплярно преобразуване. Сегментната мрежа ще изглежда по начина, показан на фиг. 5-13.

Ако впоследствие потребителят пожелае да запази тази еднаквост с едно единствено изключение - да промени цвета на едно от бюрата в единия от кабинетите, сегментната мрежа ще трябва да се промени коренно. Всяка индивидуална промяна би била невъзможна, ако не се промени самата структура на сегментите. Такава промяна би била много трудна, ако приложната програма не поддържа известна информация за създадените вече сегменти.

Това неудобство се дължи на факта, че сегментите са своеобразни "черни кутии" и когато техни екземпляри се включват в други сегменти, последните по никакъв начин не могат да управляват отделни техни атрибути. Макар мрежата от сегменти много да прилича на програмен модул, в който се правят

обръщения от една програма към друга (без пряка или косвена рекурсия), в тази мрежа липсва един важен компонент, а именно *предаването на параметри*. При наличието на параметри решаването на горната задача не би било трудно. Тези атрибути, които се очаква да се променят, могат да се обявят за формални параметри и всеки отделен екземпляр на сегмента би могъл да наложи различни атрибути на всеки от съставните си елементи, при това разположени произволно дълбоко в йерархията. Ето защо по-правилно би било да гледаме на сегментната мрежа като на своеобразна структура от данни, отколкото на модул от графични процедури.



Фиг. 5-14

С други думи, невъзможно е да зададем връзка от типа "Обектът *A* е същият като обекта *B*, само че елементът му *Xa* е различен". Този тип връзки приличат повече на разделянето на обектите на класове и наследяването на общи признаци, което е характерно за обектноориентираните системи. При непрекъснатото увеличаване на скоростта на съвременния графичен хардуер не е изключено обектноориентираните среди да заемат много важно място в бъдещите графични системи.

Моделирането със сегменти не е твърде удобно и за задаването на един друг тип връзки - релации между отделни компоненти на модела. Не е лесно например да се представи закономерността, че всяко от включваните бюра в един кабинет трябва да се ориентира успоредно на някоя от стените на кабинета. Подобни геометрични релации нарушават донякъде йерархичната структура на сегментите и могат да доведат до редундантност на метричната информация. Един естествен начин за преодоляването на тези недостатъци е да се използва йерархия от процедури, йерархия от данни или просто друг вид геометричен модел. По-важните видове модели, които се създават и поддържат без средствата на графичните системи, както и изискванията при тяхното проектиране ще разгледаме в останалите части на тази глава.

5.2 ЙЕРАРХИЧНИ МОДЕЛИ В ПРИЛОЖНИТЕ ПРОГРАМИ

Нека първо да разгледаме няколко начина за представяне на йерархията без използване на средствата, които предоставят графичните системи със запомняне на визуализираните обекти. Тези начини са подходящи в случай, че се използва прост базов графичен пакет (БГП) като този, разгледан в края на трета глава. Важно условие за прилагането на някой от тях е основните обекти, с които се работи, да е необходимо да се променят извънредно рядко. Както и в предишния пример приемаме, че основните действия, които се извършват с тези обекти са тяхното разполагане, разместване, изтриване, групиране в изграждането на по-горните нива от йерархията и др. Графичното изобразяване на самите базови обекти - **бюро**, **стол**, **маса**, **шкаф** и **преграда** - е в огромна степен фиксирано.

5.2.1 Йерархия от процедури

Най-просто е за всеки от обектите да се напише отделна процедура за изпълнението на всяка от основните дейности с този обект. Тъй като визуализацията е може би най-важната от тези дейности, за всеки от базовите обекти ще има специална процедура, която го визуализира, т.е. описва метриката на този обект в термините на графичните примитиви. Процедурите за визуализация приличат много на сегментите, с тази разлика, че приложният програмист трябва да се грижи сам за:

- извършването на обръщения към тези процедури при обхождането за визуализация на геометричния модел;
- осигуряването на достатъчен набор от формални параметри за управление на разполагането на всеки от обектите - за задаването на екземплярните им преобразувания;
- включване на атрибутите, които могат да се променят (като цвят, дебелина на линията, различна степен на детайлност и др.) като формални параметри.

Нека да разгледаме един примерен начин за написване на функцията за визуализация на обекта **бюро**, на която като параметри се задават матрицата на афинната трансформация, цвета му и параметър за управление на т.нар. *ниво на детайлност*. Нивото на детайлност е степента на подробност, с която един обект се визуализира. Тук ще разгледаме само три нива на детайлност: `FULL` - пълна визуализация, `MARK` - визуализация само на маркер в центъра на локалната координатна система и `NONE` - обектът не се визуализира. Пълната визуализация, както и в по-горния пример, се състои от два правоъгълника, а метриката (координатите на тези правоъгълници) са кодирани в самата процедура `display_DESK`.

При написването ѝ трябва да вземем пред вид, че изправените правоъгълници могат да се преобразуват в наклонени. В показания пример смятаме, че наклонените се изобразяват с примитива `InclinedRectangle`, чиито параметри са трите върха на правоъгълника, обхождайки го в посока, обратна на часовниковата стрелка.

```
void AffineTransform(x,y,X,Y,M)
float x,y,*X,*Y; Matrix M;
{    *X=M.a*x+M.b*y+M.e;
     *Y=M.c*x+M.d*y+M.f;
}

void display_DESK(M,Col,Detail)
Matrix M; int Col, Detail;
{float R1x1=-90;R1y1=0;R1x2=90;R1y2=90;
 float R2x1=-60;R2y1=0;R2x2=60;R2y2=60;
 float x1,y1,x2,y2,x3,y3;
    switch(Detail) {
       case FULL:
          SetColour(Col);
          AffineTransform(R1x1,R1y1,&x1,&y1,M);
          AffineTransform(R1x2,R1y1,&x2,&y2,M);
          AffineTransform(R1x2,R1y2,&x3,&y3,M);
          InclinedRectangle(x1,y1,x2,y2,x3,y3);
          AffineTransform(R2x1,R2y1,&x1,&y1,M);
          AffineTransform(R2x2,R2y1,&x2,&y2,M);
          AffineTransform(R2x2,R2y2,&x3,&y3,M);
          InclinedRectangle(x1,y1,x2,y2,x3,y3); break;
       case MARK:
          SetColour(Col);
          AffineTransform(0.0,0.0,&x1,&y1,M);
          Marker(x1,y1); break;
       case NONE:
          /* нищо не се визуализира */
    }
}
```

Визуализацията на по-горното ниво от йерархията - разположението на обектите в кабинетите - няма да може да се извърши чрез аналогични процедури. Тъй като броят, видът и разположението на мебелите във всеки кабинет е различен и още повече ще се променя динамично, тази част от модела може да се реализира като един нелинеен списък.

На всеки кабинет ще съответства елемент от списъка, в полетата на който са зададени: размерите на правоъгълника на кабинета, неговото разположение върху етажа (екземплярното му преобразуване) и подсписък от мебели. В един елемент от подсписъка ще се съдържа информация за всеки включен в кабинета обект:

- тип на обекта - това показва коя процедура за визуализация ще е необходимо да се извика, за да се изобрази този обект;
- екземплярното преобразуване на този обект, зададено с матрицата на афинната трансформация, която го привежда в координатната система на кабинета с център долния ляв ъгъл на правоъгълника му;
- цвят на обекта.

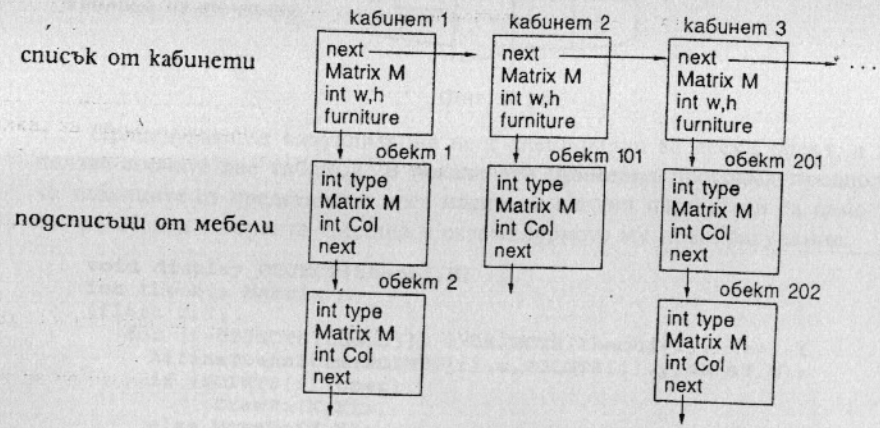
Процедурата за визуализация на кабинет ще използва информацията в подсписъка за този кабинет. Тя ще има за параметри само указател към съответния елемент от списъка на кабинети и степента на детайлност, с която обектите в кабинета трябва да се изобразят.

```
typedef struct Object *ptrObject;
typedef struct Room *ptrRoom;
```



```
typedef struct Object {ptrObject next;
                       int type;
                       Matrix M;
                       int Col; } Object;

typedef struct Room {ptrRoom next;
                     Matrix M;
                     float w,h;
                     ptrObject furniture;} Room;

void display_ROOM(theRoom,Detail)
ptrRoom theRoom; int Detail;
{Matrix M,CTM; ptrObject i;
 float x1,y1,x2,y2,x3,y3;
    M=theRoom->M;
    AffineTransform(0.0,0.0,&x1,&y1,M);
    AffineTransform(theRoom->w,0.0,&x2,&y2,M);
    AffineTransform(theRoom->w,theRoom->h,&x3,&y3,M);
    InclinedRectangle(x1,y1,x2,y2,x3,y3);
    i=theRoom->furniture;
    while (i) {
       MultiplyMatrix(M,i->M,&CTM);
       switch (i->type) {
         case DESK:  display_DESK(CTM,i->Col,Detail); break;
         case CHAIR: display_CHAIR(CTM,i->Col,Detail); break;
         case TABLE: display_TABLE(CTM,i->Col,Detail); break;
         case BOOKCASE: display_BOOKCASE(CTM,i->Col,Detail);
                        break;
         case PARTITION: display_PARTITION(CTM,i->Col,Detail);
       }
       i=i->next;
    }
}
```

Представената структура е достатъчно удобна за извършването на споменатите по-горе операции. Изтриването и добавянето на нови елементи в списък е лесно, а промяната на местоположението на всеки от отделните обекти се извършва чрез промяна на съответното екземплярно преобразуване.



Фиг. 5-15

Горните процедури извършват само визуализацията на предложената структура от данни. За всяка от останалите операции с този модел е необходимо да се напишат отделни процедури. Да вземем един конкретен пример. Типичен въпрос, който потребителят на една такава приложна програма може да зададе е "Каква е свободната (незаета от мебели) площ във всеки от кабинетите?" За да може да се даде отговор на този въпрос, приложният програмист трябва да напише:

- функция, която изчислява каква е заетата площ от всяка от отделните мебели съобразно нейното екземплярно преобразуване;
- функция, която изчислява свободната площ в една стая на базата на изчислените площи на включените мебели.

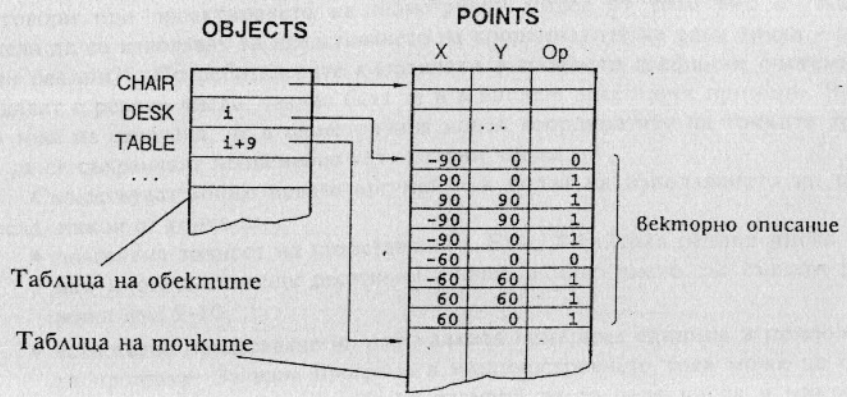
От този пример се вижда един основен недостатък на избрания подход - редундантността на данните. Всяка отделна процедура за един и същ обект поддържа собствена информация за размерите на обекта. Това изисква повишено внимание при кодиране на всяка нова процедура, тъй като данните за обекта не са общи. За да могат данните да бъдат общи е необходимо всеки обект да се представя еднотипно чрез един и същ вид графични примитиви.

При кодирането на тези функции използваме, че площта на един наклонен правоъгълник е модула на векторното произведение на векторите върху две съседни негови страни.

```
float area_DESK(M)
Matrix M;
{float R1x1=-90;R1y1=0; R1x2=90;R1y2=120;
 float x1,y1,x2,y2,x3,y3;
    AffineTransform(R1x1,R1y1,&x1,&y1,M);
    AffineTransform(R1x2,R1y1,&x2,&y2,M);
    AffineTransform(R1x1,R1y2,&x3,&y3,M);
    return(NormVectorProduct(x2-x1,y2-y1,x3-x1,y3-y1));
}

float area_ROOM(theRoom)
ptrRoom theRoom;
{Matrix M,CTM; ptrObject i;
 float x1,y1,x2,y2,x3,y3,Area;
   M=theRoom->M;
   AffineTransform(0.0,0.0,&x1,&y1,M);
   AffineTransform(theRoom->w,0.0,&x2,&y2,M);
   AffineTransform(0.0,theRoom->h,&x3,&y3,M);
   Area=NormVectorProduct(x2-x1,y2-y1,x3-x1,y3-y1);
   i=theRoom->furniture;
   while (i) {
         MultiplyMatrix(M,i->M,&CTM);
         switch (i->type) {
           case DESK: Area=Area-area_DESK(CTM); break;
           case CHAIR: Area=Area-area_CHAIR(CTM); break;
           case TABLE: Area=Area-area_TABLE(CTM); break;
           case BOOKCASE: Area=Area-area_BOOKCASE(CTM); break;
           case PARTITION: Area=Area-area_PARTITION(CTM);
         }
         i=i->next;
   }
   return(Area);
}
```


5.2.2 Просто векторно представяне

Един прост начин за уеднаквяване на представянето е всички обекти да се задават като последователност от вектори, което би направило всички процедури за визуализация еднакви. Всички те биха използвали един и същ графичен примитив - отсечка - и обща структура от данни, в която има информация кой обект от каква последователност от отсечки се състои. Това представяне можем да наречем *проста йерархия на данните*. Ще покажем една примерна структура, в която за всеки обект е отделен по един ред от таблица на обектите, който съдържа начален адрес в друга таблица, в която са записани последователностите от точки, векторите между които са графичният образ на този обект. Всяка точка е зададена с три компонента: двете координати в локалната координатна система на обекта и един трети елемент, който задава коя от двете графични функции MoveTo или DrawTo ще се използва при изобразяването на точката. Този прост начин позволява лесното кодиране както на начупени, така и на отделни несвързани отсечки в едно общо описание. Обектът **бюро**, за визуализирането на който използвахме два правоъгълника, може да бъде кодиран със следната последователност от точки:



Фиг. 5-16

Процедурата за визуализация не е специфична за всеки обект, а ще използва горните две таблици. В показаната примерна програма предполагаме, че таблиците са представени чрез масиви и входни параметри са само индексът на обекта в първата таблица и екземплярното му преобразуване:

```
void display_OBJECT(theobj,M)
int theobj; Matrix M;
{float X,Y;
   for (i=OBJECTS[theobj]; i<OBJECTS[theobj+1]; i++) {
     AffineTransform(POINTS[i].x,POINTS[i].y,&X,&Y,M);
     if (POINTS[i].oper)
          DrawTo(X,Y);
     else MoveTo(X,Y);
   }
}
```

Този начин позволява изменението на данните да става без да се променят процедурите за визуализация. Той обаче не дава възможност за визуализация на обекти при използването на дъги и окръжности. Последният недостатък може да се преодолее ако обектите се описват не чрез вектори, а чрез дъги или криви, както ще видим по-късно в тази глава.

Извършването на анализи в такъв модел става, като се използва метричната информация в структурата от данни, която въведохме. Процедурите за тези анализи ще бъдат също общи, както и при визуализация, но самото обобщаване може да не е никак лесно. Читателят може като пример да сравни сложността при написването на функция, която намира площта на един обект, зададен по горния начин с тривиалната процедура, която написахме за отделния обект - **бюро**.

5.3 КОНТУРНИ МОДЕЛИ

В тази част ще разгледаме няколко примерни геометрични модела на една приложна програма, в която основните елементи, с които се работи са близки до графичните примитиви. При автоматизирането на чертожната дейност потребителят работи в термини като отсечка, дъга, правоъгълник, окръжност, начупена и др. Естествено е основните елементи в геометричния модел да са подмножество на тези обекти. Ние ще се спрем на това какво е и как се избира това подмножество в тези приложни програми, които работят главно с едномерна информация - контури, а не равнинни области, които имат площ. Такива са на практика голяма част от т.нар. *чертожни системи*.

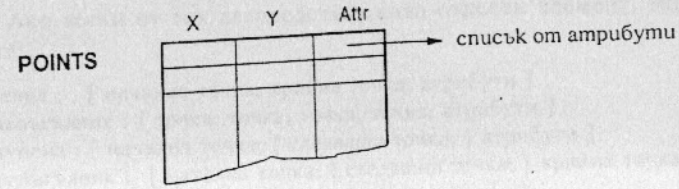
5.3.1 Представяне на точки

Най-простите елементи в една чертожна система са точките. Чертожниците рядко работят с индивидуални точки, но тяхното обособяване като отделни елементи на модела има редица причини и много предимства при извършването на разнообразни обработки. Някои от по-важните причини са следните:

- наличието на точки, които са общи за два или повече различни по тип примитива. В реалните чертежи рядко може да се видят отсечки или дъги, които да не са свързани с други примитиви. Тъй като повечето от примитивите се представят с крайните си точки, многократното съхраняване на една и съща точка би било много неикономично и не би отразявало свързаността на отделните елементи (инцидентността на краищата им).
- честото търсене само измежду точките. Типичен пример е използването на интерактивния похват *гравитационно поле* при позициониране.
- възможност за модифициране на обектите чрез трансформиране само на някои техни точки, при което се постигат разнообразни ефекти на разтягане и скъсяване, както бе показано в четвърта глава (фиг. 4-21).
- наличието на атрибути, специфични за една точка, както и на релации между отделни точки.

Точките се представят в потребителската координатна система. Освен двете координати X и Y, много често за всяка точка е необходимо да се знае и определен набор от негеометрични характеристики, които ние ще наречем с общото име *атрибути на точката*. Атрибутите на една точка могат да бъдат: наличието на маркер в тази точка, неговият тип, големината му, цвета му и др. Съхраняването на координатите и указател към структура от атрибути в един масив би било удачно поради директния достъп, който се налага сравнително често.

```
struct {Coord X,Y; ptrAttrib Attr;} POINTS[];
```



Фиг. 5-17

Първият основен въпрос, на който всеки приложен програмист трябва да отговори при проектирането на геометричен модел от този вид е "Какви числа да се използват за представянето на координатите на една точка - цели или реални?". Потребителските координати в повечето графични системи се задават с реални числа, такива бяха те и в нашите досегашни примери. Всичко това не означава, че в геометричния модел координатите на точките трябва да се съхраняват непременно като реални числа.

Съществуват много повече аргументи в полза на използването на цели числа, някои от които са:

- по-голяма точност на представянето. Едно 4-байтово реално число има най-много 6 значещи десетични цифри, а цяло число със същата дължина има 9-10.
- естествено представяне на най-малката измерима единица в приложната програма. За една програма в машиностроенето това може да бъде един микрон, а оттам и всички размери да са цели числа в микрони. По този начин се предотвратява наличието на много точки като отделни елементи на горния масив, които са практически идентични, но координатите на които не са тъждествено равни.
- представяне на координатите като размерни величини. Всички величини, в които един потребител - чертожник, проектант - мисли и работи са размерни. Не само дължините, но и физични величини като плътност, сила, работа и др. са свързани с размерността на координатите.

Ако потребителят работи в милиметри, а минималното различимо разстояние е 1 микрон, то на целочисленото представяне може да се гледа като на число с фиксирана точка. В някои езици за програмиране има пълен набор от средства за работа с такива числа, а в други те лесно могат да се симулират.

5.3.2 Представяне на контурни елементи

Нека разгледаме типичните геометрични обекти, които се използват в чертожните системи и параметрите, които ги определят. Един сравнително пълен набор от обекти, за които чертожните системи предлагат методи за конструиране е следният:

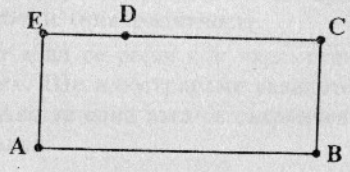
- отсечка
- правоъгълник
- начупена (линия)
- многоъгълник
- окръжност
- дъга от окръжност
- елипса
- дъга от елипса
- крива

Кривите в геометричните модели са най-често апроксимиращи криви от различен тип: криви на Безие, B-сплайн криви, рационални B-сплайн криви и др., математическите методи за представянето на които ще разгледаме в следващата глава. Тук ще кажем само, че те се задават с поредица от точки и метода за интерполирането или апроксимирането им. Извънредно рядко в практиката се налага използването на общи криви от втора степен (с изключение на елипсите, които се получават при мащабиране или афинно трансформиране на окръжности), затова в горния списък не са включени дъги от парабола и хипербола. Ограничаването на броя на елементите в показания списък спестява на приложните програмисти кодирането на различни алгоритми за обработването на всеки от видовете елементи.

Правоъгълникът, многоъгълникът и отсечката са частни случаи на начупената. Макар в чертожните системи да се предлагат отделни диалогови средства за създаването им, те рядко са отделни елементи в геометричния модел. Специалните методи за тяхното конструиране само улесняват чертожната работа и са фактически част от диалога на приложната програма.

За да реши дали да ги представя като отделни елементи в модела, приложният програмист трябва да знае отговорите на въпросите:

- Ще се налага ли търсене на обекти само от този тип и ще има ли модификации, които могат да се извършват само върху обекти от него?
- Има ли операции, които да променят типа на обекта - изтриване на част от елемент, добавяне на връх, афинни трансформации и др.?
- Трябва ли трансформациите да запазват типа на обекта - например окръжността да се преобразува в окръжност независимо от трансформацията?
- Доколко визуалното представяне трябва да съответства на представянето в геометричния модел? На фиг. 5-18 е показан един пример на елемент, който визуално е правоъгълник, но всъщност е представен като петоъгълник с две съседни колинеарни страни.



Фиг. 5-18

ПРАВОЛИНЕЙНИ ОБЕКТИ. Нека първо се спрем на праволинейните обекти. Ако всеки от тях се представя като отделен елемент, моделът би бил следният:

```
отсечка :   [ начална точка; крайна точка; атрибути ]
правоъгълник : [ точка; точка; точка; точка; атрибути ]
начупена : [ начална точка; { следваща точка; } атрибути ]
многоъгълник :  [ начална точка; { следваща точка; } крайна точка; атрибути ]
начална точка :  [ точка ]
следваща точка : [ точка ]
крайна точка :  [ точка ]
```

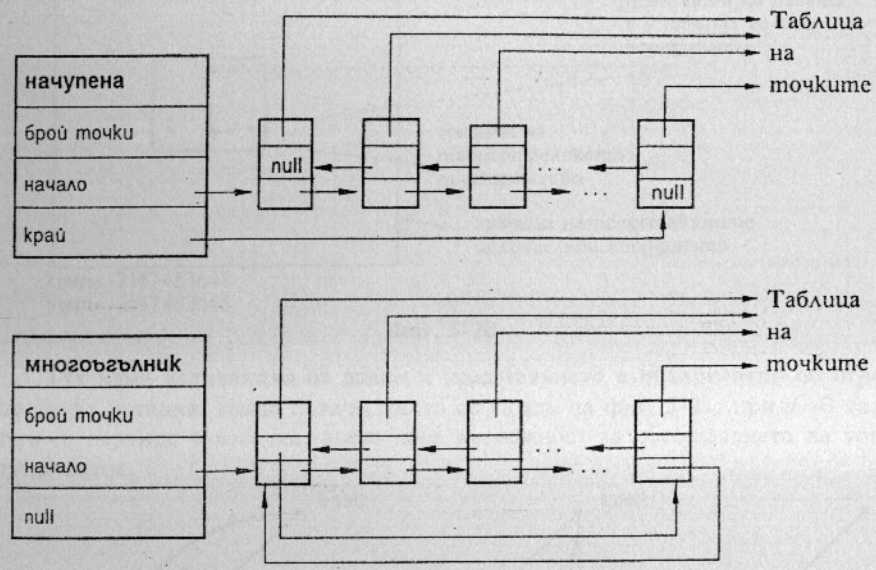
Подобно на точките, всеки елемент има набор от атрибути, които освен начина на визуализация определят важността, ролята и интерпретацията му от гледна точка на приложната програма. На някои типични атрибути ще се спрем, след като разгледаме представянето на контурните елементи.

Изобилието на типове елементи в геометричния модел може да усложни значително обработката му и затова в по-голямата част от случаите в контурните модели за елементи се избират само отсечки и начупени. Правоъгълникът и многоъгълникът се представят като начупени, което спестява преобразуването на типове при операции като: добавяне на връх, скосяване на връх, афинна трансформация и др. По този начин донякъде се избягва двусмислието, което показахме при представянето на елемента от фиг. 5-18.

```
отсечка :  [ точка; точка; атрибути ]
начупена :  [ брой точки; начална точка; { следваща точка; } атрибути ]
```

Отделянето на отсечките е обосновано при честото им самостоятелно използване, тъй като те могат да се кодират в много по-проста структура от данни от начупената. Друга проста алтернатива е всички праволинейни обекти да се представят чрез отсечки, но това би било неикономично, защото ще се удвои броят на указателите към точките. Освен това ще липсва информация за свързаността им, освен ако от точките няма обратни указатели към елементите, в които те участват. Последното би направило структурата прекалено тежка за обработка и поддържане. Свързаността е необходима дори само при визуализация - при удебеляване и оформяне на върховете на свързаните елементи. Броят на точките на една начупена е по-добре да се съхранява в явен вид, особено ако за физическото ѝ представяне се използват масиви. Един примерен начин за използване на масиви показахме на фиг. 5-14. Друг възможен начин е използването на двусвързан списък, защото обхожда-

нето на начупената често се извършва и в двете посоки, а освен това за много операции е необходимо да се знаят съседните върхове на даден връх.



Фиг. 5-19

В обектноориентираните среди праволинейните елементи могат да се дефинират като клас, обектите в който са отсечки, правоъгълници, начупени и многоъгълници. В някои случаи е подходящо използването на обект като безкрайната права. Освен с коефициентите в аналитичното си представяне, тя може да се зададе и само с две точки, през които минава. По този начин тя ще има същото представяне като отсечка, но с различна интерпретация.

Кривите също могат да се приобщят към този клас, защото, както казахме по-горе, те се представят чрез поредицата от управляващи точки и типа на интерполацията или апроксимацията, която се прилага към тези точки. Изключение правят нерационалните криви, при които за всяка точка е необходима още една координата, както ще видим при нерационалните B-сплайн криви.

крива : [тип; начална точка; { следваща точка; } атрибути]
тип : ['B-СПЛАЙН' | 'КРИВА НА БЕЗИЕ' | 'КРИВА НА ФЪРГЮСЪН']

ОКРЪЖНОСТИ, ЕЛИПСИ И ДЪГИ. Нека сега разгледаме представянето на окръжността, елипсата и дъгите от тях. Основните изисквания при избора на представянето им са следните:

- минимизиране на редундантността на данните;
- наличие на всички данни, необходими като параметри на най-често налагащите се операции с тях;

- свеждане на трансформациите над тях до трансформации над характеристичните им точки (инвариантност).

Задачата всъщност е да се реши кои параметри ще се съхраняват и кои ще се извеждат чрез тях. Ще илюстрираме казаното с избора на представяне на дъга от окръжност. Ако за една дъга се съхраняват центърът, радиусът, началният и крайният ѝ ъгли:

дъга : [център; радиус; начален ъгъл; краен ъгъл; атрибути]
център : [точка]
радиус : [скалар]
начален ъгъл : [скалар]
краен ъгъл : [скалар]

то няма да има излишество от данни и информацията, необходима при обработване с използване на параметричното уравнение на окръжност е налице. Липсват обаче някои много важни параметри, каквито са началната и крайната точки на дъгата, които са необходими при: посочване, гравитационно поле при позициониране, определяне на инцидентност на елементи.

Друго неудобство е, че това представяне не позволява трансформация върху дъгата да се сведе до трансформиране само на точките ѝ, както това може да става при праволинейните елементи и кривите. При ротация например трябва да се променят ъглите, а при хомотетия - радиусът ѝ. При мащабиране с различни коефициенти дъгата от окръжност се преобразува в дъга от елипса. Това е силен аргумент в полза на общото представяне на дъгите от окръжности и елипси, както и на целите окръжности и елипси.

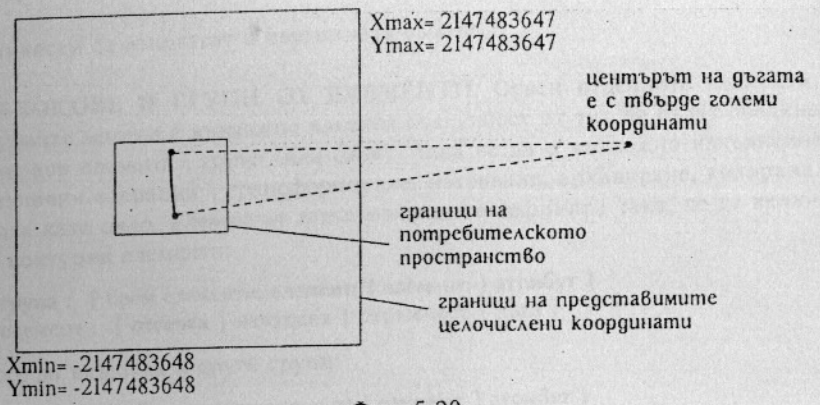
Добавянето на крайните точки към горното представяне би увеличило твърде много излишеството от данни без да реши въпроса с инвариантността му. Една възможна алтернатива е да се съхраняват центърът и крайните точки на дъгата, а радиусът и ъглите ѝ да се извеждат при нужда чрез координатите на тези точки.

дъга : [център; начална точка; крайна точка; атрибут]

Тук отново има излишество на данни, защото разстоянието от центъра до всяка от точките трябва да е еднакво. При мащабиране или при независима трансформация на някоя от точките тази зависимост ще се разруши. Горното представяне има и едно неудобство при дъги с много малка кривина (т.е. с относително голям радиус). Ако използваме цели числа за съхраняване на координатите на точките, може да се случи така, че центърът на една дъга, която се намира изцяло в квадрата, определен от целочислените граници, да е извън този квадрат (фиг. 5-20).

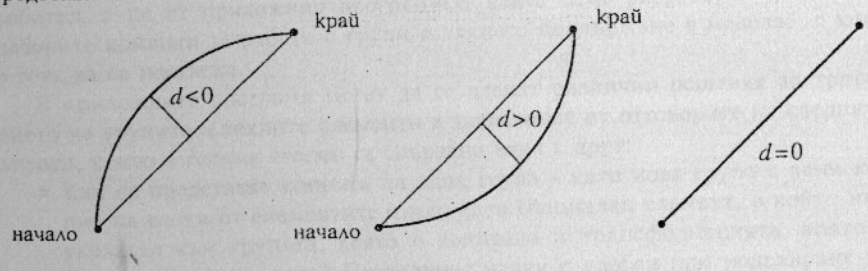
Това неудобство може да се избегне, като се съхраняват началната и крайната точки на дъгата и ориентираната височина на сегмента, който тази дъга определя. Тук използваме "сегмент" в неговия геометричен, а не графичен смисъл:

дъга : [начална точка; крайна точка; височина; атрибути]
височина : [скалар]



Фиг. 5-20

Тук няма излишество от данни и представянето е инвариантно по отношение на ротация. Нещо повече, както се вижда на фиг. 5-21, при $d=0$ тази дъга се изражда в отсечка, което дава възможност за обобщаването на това представяне.



Фиг. 5-21

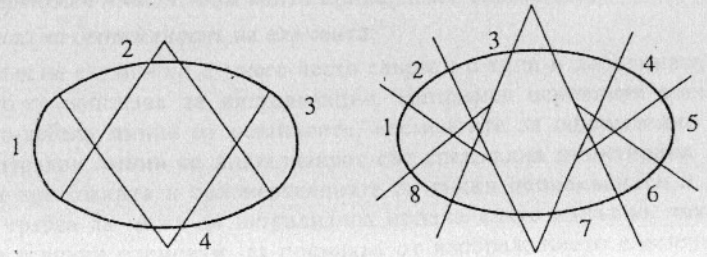
Окръжностите могат да се представят само чрез центъра и радиуса си, а елипсите - с център, дължина на голямата и малка полуоси и ъгъла, задаващ наклона на голямата ос спрямо *Ox*.

окръжност : [център; радиус; атрибути]
елипса : [център; голяма полуос; малка полуос; наклон; атрибути]
голяма полуос : [скалар]
малка полуос : [скалар]
наклон : [скалар]

Ъгълът, задаващ наклона на главната ос на елипсата, може да се избегне, за да може представянето да е инвариантно по отношение на ротация. Ако вместо центъра на елипсата, полуосите и наклона ѝ се съхраняват двата ѝ фокуса и сумарното разстояние на всяка точка до тях, представянето ѝ ще бъде толкова инвариантно по отношение на трансформации, колкото и това на окръжността.

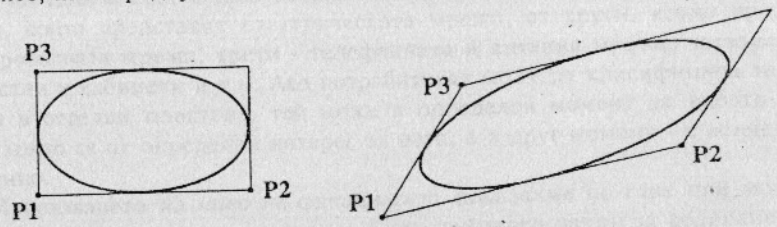
елипса : [фокус; фокус; сумарно разстояние; атрибути]
фокус : [точка]

В много системи вътрешното представяне на елипсите се извършва приближено - чрез няколко дъги от окръжности, както е показано на фиг. 5-22. Това спестява разработването на специални алгоритми за обработване на елипси, които в някои случаи (дори само за намирането на пресечни точки) далеч не са тривиални от изчислителна гледна точка. Приближеното представяне е удачно в случай, че не се изисква голяма точност при пресмятане на характеристиките на елементите.



Фиг. 5-22

Един общ начин за представяне на елипса (и дъга от елипса), чиито оси са успоредни на координатните е чрез три върха от правоъгълната ѝ обвивка (плюс двата параметрични ъгъла за дъга, които в общия случай са различни от ъглите, които радиус-векторите сключват с оста *Ox*) - фиг. 5-23.



Фиг. 5-23

Използването на параметричните ъгли осигурява инвариантност по отношение на ротация. Това представяне позволява върху точките да се изпълняват афинни преобразувания, както и независими трансформации на някоя от тях, при което трите върха задават успоредника, получен при трансформирането на правоъгълната обвивка.

елипса : [точка P1; точка P2; точка P3; атрибут]
дъга от елипса : [точка P1; точка P2; точка P3;
начален ъгъл; краен ъгъл; атрибут]
точка P1 : [точка]
точка P2 : [точка]
точка P3 : [точка]

Друг подход към елипсите и дъгите от елипси е да се поддържат само такива, чиито оси са успоредни на координатните. Това е оправдано в голяма част от приложенията, тъй като това са именно тези елипси и дъги, които

практически се използват в чертожната работа.

БЛОКОВЕ И ГРУПИ ОТ ЕЛЕМЕНТИ. Освен отделните елементи, в контурните модели е възможно някаква съвкупност от тях да бъдат обединени в един нов елемент - *група* (или *блок*), така че да е възможно изпълнението на различни операции - трансформиране, изтриване, архивиране, копиране на групата като цяло. Елементът *група* може да се дефинира така, че да включва само контурни елементи:

група : [брой елементи; елемент; { елемент; } атрибут]
елемент : [отсечка | начупена | окръжност | дъга]

а може да включва и други групи:

група : [брой елементи; елемент; { елемент; } атрибут]
елемент : [отсечка | начупена | окръжност | дъга | група]

без, разбира се, да се допуска рекурсия. Влагането на групите дава възможност за йерархично организиране на модела. За разлика от разгледаните по-горе йерархични модели, в този модел йерархията се построява от самия потребител, а не от приложния програмист, който само разработва средствата (наборите команди за работа с групи и тяхното поддържане в модела), с които това да се постигне.

В приложните програми могат да се вземат различни решения за третирането на групите и техните елементи в зависимост от отговорите на следните въпроси, които в голяма степен са свързани един с друг:

- Как се представят копията на една група - като нова група с явни копия на всеки от елементите ѝ или като специален елемент, в който има указател към групата, която е копирана и трансформацията, която е приложена върху него? Последният начин е удобен при моделиране на йерархия от фиксирани обекти, но липсата на елементи в явен вид може да е определено неудобство - нито съставните елементи, нито точките им могат да бъдат посочени при просто обхождане на модела. Освен това, индивидуални промени в определено копие ще бъдат невъзможни.
- Позволена ли е трансформация, изтриване и модификация на един елемент, ако той принадлежи на група?
- Уеднаквяват ли се атрибутите на елементите при свързването им в група? Ако ДА, кой от тях става атрибут на групата?

АТРИБУТИ НА ЕЛЕМЕНТИТЕ В ГЕОМЕТРИЧНИЯ МОДЕЛ. Нека сега да се спрем на видовете атрибути, които е необходимо да се съхраняват за всеки от елементите в контурния модел. Тук няма да разглеждаме специфична за конкретна област информация, която приложният програмист може да асоциира с елементите от модела. Ще се спрем само на тези характеристики, които непосредствено влияят на визуализацията, а това са на първо място графичните атрибути (предлагани от графичната система): цвят, тип и дебелина на линията.

В приложните програми елементите могат да играят различна роля - да имат различна важност, различна необходимост от визуализация във всеки конкретен момент. За тази цел една приложна графична програма може да предложи следния допълнителен набор от атрибути:

- *тип на елемента*: основен, помощен (спомагателен), за конструиране, за оформление и др.;
- *чертожен пласт*, към който принадлежи елементът;
- *ниво на детайлност* на елемента.

Типът на елемента е много често свързан с типа и дебелината на линията, която се използва за визуализация. Например основните елементи са с малко по-дебела линия от останалите, елементите за оформление на чертеж като централни линии се визуализират със специална пунктирана линия, линиите от щриховката и оразмеряванията са тънки непрекъснати и др. Потребителят трябва да може да визуализира модела както напълно, така и само с неговите основни елементи, да премахва от изображението елементите, свързани само с оформлението и др.

В чертожните системи чертежът се разделя на *пластове* - все едно, че много чертежи са насложени един върху друг, които могат да се визуализират заедно, поотделно и в произволна комбинация. Целта е да се позволи съвместяването на разнородна по вид информация, която отразява различни аспекти на един и същ модел. Планът на една сграда може да се състои от елементи, които представят електрическата мрежа, от други, които представят водопроводната мрежа, трети - телефонната и антенна мрежа, четвърти - самите стаи и кабинети и т.н. Ако потребителят може да класифицира тези елементи в отделни пластове, той може в определен момент да работи само с тези, които са от определен интерес за него, а в друг момент - с всички едновременно.

Използването на *ниво на детайлност* показахме по-горе при моделирането на йерархията чрез процедури. Един примерен начин за кодирането му е с номер в някакво наредено множество например целите числа от 1 до 16. При визуализация с ниво на детайлност N ще се визуализират всички елементи, чието ниво на детайлност е по-малко или равно на N. По този начин всички по-крупни елементи от модела ще имат малък номер, а всички по-дребни детайли ще имат сравнително голям номер. Най-незначителните детайли ще имат *ниво на детайлност* = 16.

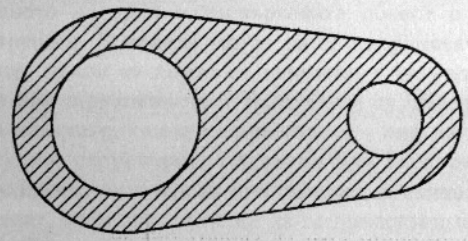
Да отбележим, че една приложна програма може да поддържа голям набор от атрибути, управляващи визуализацията, но не всички те да се съхраняват като атрибути на елементите. Важни въпроси, които се отнасят до работата с атрибути са:

- как се задават атрибутите и кога се свързват те с елементите?
- може ли да се променя атрибутът на елемент?
- имат ли групите атрибути?
- кой е атрибутът на една група - този на елементите или този на самата група?

5.3.3 Граничноопределени модели

Разгледаните дотук модели са удобни за чертожните системи, тъй като те са начин за представяне само на контурите в един чертеж. За извършването дори на най-прости анализи върху модела, който се представя на този чертеж е необходимо приложната програма да разполага и с много друга допълнителна информация. Да разгледаме детайла, показан на фиг. 5-24.

Той не е обикновена група от дъги, отсечки и окръжности. Този детайл може да представлява двумерното сечение на реален физически обект от определен тип материал. В показания чертеж има достатъчно информация за това, какви са границите на обекта, от коя страна на границите му се намира материалът, каква е площта му и дори какъв е пътят на инструмента на една машина, която трябва да го произведе. В тази част ще се спрем на такива модели, които позволяват представянето на тази информация и някои от обработващите алгоритми в тях.



Фиг. 5-24

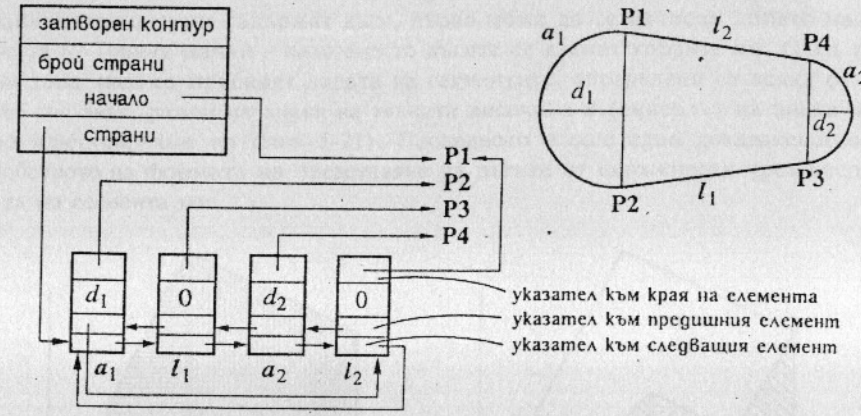
ЗАТВОРЕНИ КОНТУРИ. ОБЛАСТИ. Моделите, които разглеждаме тук се наричат *граничноопределени*, защото основните елементи в тях са равнинни фигури, които имат площ и които са зададени с ограждащите ги контури. Тези контури са затворени - заграждат определена част от равнината - и от своя страна се състоят от последователност от свързани контурни елементи: отсечки, начупени и дъги или са една от равнинните фигури: многоъгълник, окръжност или елипса. Един затворен контур може да бъде представен в такъв геометричен модел по следния начин:

затворен контур : [многоъгълник | окръжност | елипса |
отворен елемент; { отворен елемент; }]
отворен елемент : [отсечка | начупена | дъга | дъга от елипса]

Последователността от *отворени елементи* трябва непременно да бъде свързана, т.е. краят на всеки да е същата точка, която е начало на следващия. Един начин да се спести наличието на два указателя към всяка точка, която е обща на два съседни елемента в такава последователност, е да използваме представянето на дъгите от фиг. 5-21. Ако се абстрахираме от елипсите и дъгите от тях, можем да въведем елемента *затворен контур*, който е обобщение на многоъгълник, всяка страна на който се характеризира с височината на сегмента (както видяхме той е 0 при отсечки).

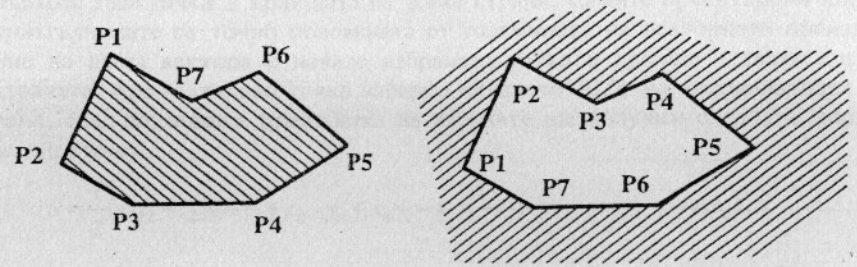
затворен контур : [начална точка; страна; { страна; } атрибути]
страна : [крайна точка; височина]

Окръжността също може да се представи като затворен контур, чиито страни са двете ѝ половинки-дъги. Външният контур на детайла, показан на фиг. 5-24, ще бъде описан по следния начин:



Фиг. 5-25

Каквото и представяне да се избере на затворения контур, в него непременно трябва да има информация за това от коя негова страна се намира вътрешността му - материалът на обекта, чиято граница е той. За една окръжност или елипса това трябва допълнително да се укаже чрез някакъв атрибут на представянето. При затворените контури и многоъгълниците може да се приеме, че последователността от страни (или върхове) е съхранена по такъв начин, че при обхождането ѝ, материалът винаги се намира *отляво* на съответната страна. На фиг. 5-26 е показана интерпретацията на двете възможни наредби на върховете на един многоъгълник. За правилното подреждане на върховете може да се използва знака на ориентираното лице на затворения контур.



Фиг. 5-26

Една равнинна фигура може да има за граници повече от един затворен контур. Ако тя има само един външен контур и множество от непресичащи се

и несъдържащи се един в друг вътрешни контури, тя се нарича *едносвързана област*. Това е област, всеки две вътрешни точки на която могат да бъдат съединени с непрекъсната линия, която е изцяло във вътрешността на областта. Ако това условие не е изпълнено, областта се нарича *многосвързана*. В моделирането много рядко се използват многосвързани области, затова и ние няма да се спираме на тяхното представяне.

Едносвързаните области, които за кратко нека наричаме само *области*, могат да се представят като съвкупност от затворени контури, като е известно къде се намира вътрешността спрямо всеки от тях. Използването на наредбата на върховете за получаването на тази информация дава възможност да се представят дори многосвързани области.

област : [{ затворен контур; } атрибути]

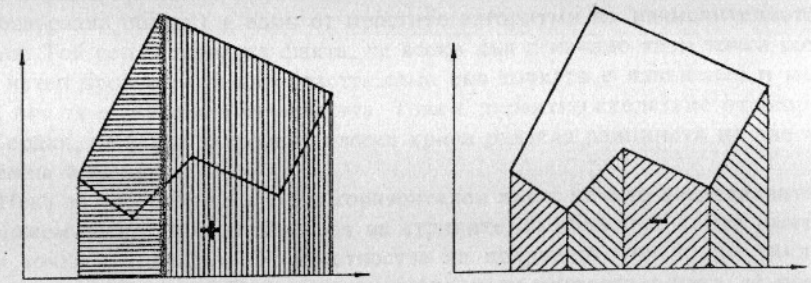
Много от обработките в модела се улесняват, ако първият затворен контур в представянето на една едносвързаната област е външният, а всички следващи са вътрешни. В модела могат да се съхраняват разнообразни атрибути на областите, някои от които са свързани с визуализацията им (например запълването или щрихованието им), а други са определени физически характеристики на обектите, които те представят - вид на материала, плътност, негови температурни, структурни, магнитни или електрически характеристики и др. Ще отбележим, че в много приложения, в които обработващите алгоритми го позволяват, една област може да се представи и като един единствен затворен контур. Това се постига като отделните контури се съединят подходящо с отсечки и се извърши обхождане, при което през съединителните отсечки се преминава по два пъти (в двете различни посоки). Припомнете си, че точно в този вид се получават многосвързаните области след отсичане на неизпъкнали многоъгълници в алгоритмите на Садърланд-Ходжман и Лян-Барски.

Използването на наредбата на върховете за кодиране на информацията за вътрешността на областта позволява ефективно реализиране на много обработващи алгоритми. Например една от дейностите, които често се извършват е изчисляването на площта на една област.

ПЛОЩ НА ГРАНИЧНООПРЕДЕЛЕНА ОБЛАСТ. Да разгледаме първо един многоъгълник. Неговата площ можем да намерим, като изчислим ориентираните площи на трапеците, образувани от всяка страна на многоъгълника и двете вертикални отсечки от крайните ѝ точки до оста Ox. Можем лесно да видим, че ако сумираме тези лица, ще получим ориентираното лице на самия многоъгълник. Ориентираното лице на самите трапеци ще вземем за положително, ако x-координатата на втория му връх е по-малка от x-координатата на първия спазвайки посоката на обхождане на страните му. На фиг. 5-27 с наклонени щрихи са отбелязани тези лица, които имат отрицателен знак, а с вертикални или хоризонтални - тези, които са положителни. Вижда се, че общата им сума дава точно лицето на областта, при това полученото лице е ориентирано, т.е. съответства на наредбата на върховете на многоъгълника. То ще бъде положително, ако ориентацията на многоъгълника е такава, че

вътрешността се намира отляво на контура му, и отрицателно в обратния случай.

По този начин лицето на една област ще се получи като сума от ориентираните лица на всички нейни контури, т.е. от лицето на най-външния многоъгълник ще се извадят лицата, заградени от вътрешните ѝ контури. Ако граничните контури съдържат дъги, първо може да се изчисли лицето на областта по горния начин - като вместо дъгите се вземат хордите им. След това към това лице се прибавят лицата на сегментите, определени от всяка от дъгите със знак, равен на знака на тяхната височина d (смисълът на знака на d ние илюстрирахме на фиг. 5-21). Последното е още едно доказателство за удобството на формата на представяне на дъгите от окръжности чрез височината на сегмента им.



Фиг. 5-27

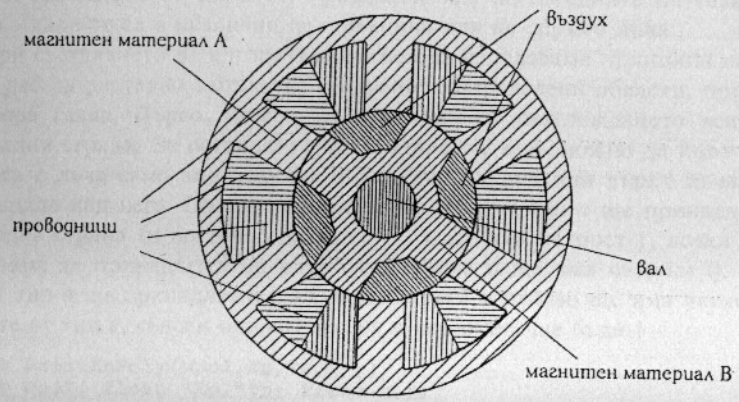
Предложеният метод на изчисление може да доведе до сериозни грешки от закръгляване при многоъгълници, които са разположени твърде далеч от оста Ox. Това може да се преодолее, като се извърши проста вертикална транслация, докато една от точките на многоъгълника попадне върху хоризонталната координатна ос.

Друг начин за намиране на лицето на многоъгълник е да се избере една произволна точка и да се пресметнат ориентираните лица на триъгълниците с върхове тази точка и краищата на всяка страна. Самите ориентирани лица на триъгълниците са точно половината от големината на векторното произведение на двата вектора с начало избраната точка и краища - двата края на страната. Ако за такава точка изберем началото (0,0) на координатната система, след подходяща преработка на изразите ще получим следната формула за лицето му:

$$S = \frac{1}{2}\left(x_1y_2 + x_2y_3 + \cdots + x_{n-1}y_n + x_ny_1 - x_2y_1 - x_3y_2 - \cdots - x_ny_{n-1} - x_1y_n\right).$$

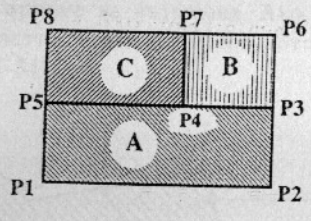
КОМПЛИМЕНТАРНИ (ДОПЪЛВАЩИ СЕ) ОБЛАСТИ. В някои приложения (като например тези, извършващи анализ по метода на крайните елементи) моделът се състои от множество неприпокриващи се области, които представляват отделните допълващи се части на едно най-често двумерно сечение. В този случай, вътрешните контури (или части от тях) на едни обла-

сти са външни контури (или части от тях) на други области. Всеки елемент на затворен контур (освен на най-външния) принадлежи точно на два контура на две съседни области (фиг. 5-28).



Фиг. 5-28

Ако всяка област се представя с контури, които са независими от останалите контури в модела, промяна в един от тях няма да предизвиква автоматично променяне на границите на съседните области и моделът би бил физически неправилен. От друга страна, за да могат общите граници на една област да се представят само по веднъж, те трябва да са разделени на части съобразно съседството на тази област с останалите в геометричния модел. На следващата фигура са показани три съседни области, границите на които са изправени правоъгълници. Правоъгълникът А обаче ще се представя като петоъгълник, защото горната му страна е разделена на две отсечки, всяка от които има различни съседни страни.



Фиг. 5-29

При такова представяне, в модела трябва да се предвиди съхраняването на информация за това един контур на кои две области принадлежи. Както казахме по-горе, повечето модели от този тип се използват за симулация с крайни елементи, а самите крайни елементи са триъгълници и четириъгълници, при което истинската геометрия се представя чрез области, чиито граници са многоъгълници. Това улеснява донякъде представянето и кодирането на редица алгоритми за обработка. Ние ще спрем на най-често използваните алгоритми за многоъгълници, като оставим на читателя да ги обобщи за произволни области.

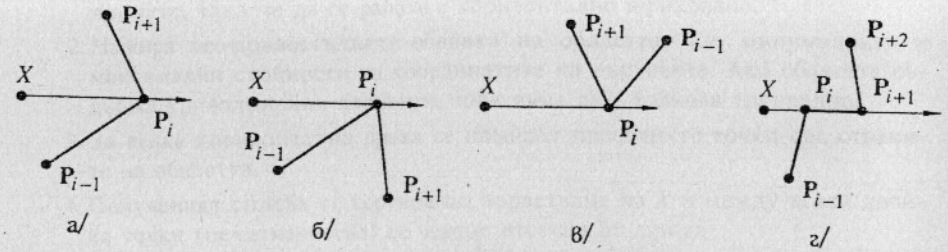
ОПРЕДЕЛЯНЕ НА ПОЛОЖЕНИЕТО НА ТОЧКА СПРЯМО ОБЛАСТ.

Това се налага при изпълнението на най-различни операции в граничноопределен модел. Ако разгледаме модела на фиг. 5-28 ще видим, че ще бъде много трудно да посочим една област, ако посочването се извършва само чрез съставящите я затворени контури. Тъй като всеки от тях (освен външният) принадлежи на точно две области, посочването на контур не определя областта. Много по-лесно е да посочим вътрешността на областта, с което ще избегнем всякакво двусмислие.

Алгоритъмът на Жордан (Jordan) за определяне на положението на една точка спрямо произволен многоъгълник (и дори многоъгълници, представящи многосвързана област) е един от простите алгоритми на изчислителната геометрия. Той се основава на факта, че всеки лъч с начало тази точка ще пресича четен брой страни на областта само ако точката е извън нея и нечетен брой, ако тя е вътрешна за областта. Това е директно следствие от теоремата на Жордан, че всяка затворена плоска крива разделя равнината на две части: вътрешна и външна.

Нека за удобство да вземем хоризонтален лъч с начало изследваната точка. Можем лесно да намерим броя на страните на областта, които имат пресечни точки с този лъч. От нечетността на полученото число ще заключим къде лежи тази точка. Преди да кодираме този алгоритъм нека да обърнем внимание на някои особености. В разсъжденията си от тук нататък ще говорим за многоъгълник, макар че читателят лесно може да се убеди, че резултатите са приложими за произволна област.

- Ще считаме ли за пресечна точка преминаването на лъча през някой от върховете на многоъгълника?
- Колко пресечни точки има лъчът с такава страна от многоъгълника, която лежи върху този лъч?
- Ако точката лежи върху страна от многоъгълника, вътрешна ли е тя за него или не?



Фиг. 5-30

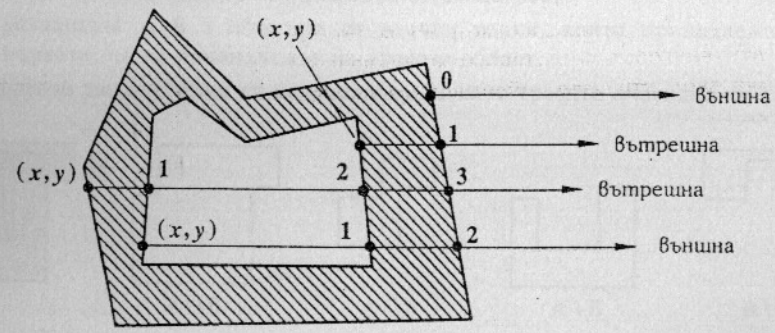
Отговорът на първите два въпроса можем лесно да дадем, ако разгледаме внимателно възможните случаи на пресичане, показани на фиг. 5-30. Ако лъчът пресича връх, образуван от страни, разположени от едната му страна

(случаи *б* и *в*) - пресичане реално няма. Ако пък двете страни са разположени в двете различни полуравнини спрямо този лъч (случай *а*) - има едно пресичане. Ако една от страните лежи на лъча (случай *г*), то наличието на пресечна точка зависи от взаимното разположение на съседните на тази страна страни - дали те са в различни полуравнини или не спрямо лъча.

При съставянето на алгоритъма можем да използваме принципа на *сканиращия ред* за растерно запълване на граничноопределени области, представен във втора глава. Първо, можем да изключим от разглеждането всички хоризонтални страни. За останалите ще смятаме, че е възможно да имат пресечна точка с лъча само ако долните им краища са под или върху лъча, а горните изцяло над него. По този начин всеки връх от тип *а* ще принадлежи само на една страна (и по-точно на долната) и ще има четност 1, всеки връх от тип *б* няма да принадлежи на нито една страна и ще има четност 0, а всеки връх от тип *в* ще принадлежи и на двете страни и отново ще има четност 0. В случаите от тип *г*, съвсем аналогично на *а* четността ще бъде 1.

```
int PointInPoly(Npol,Xp,Yp,x,y)
int Npol; float *Xp,*Yp; float x,y;
{int i,j,cnt;
   cnt=0;
   for (i=0; i<Npol; i++) {
      j=(i+1)%Npol;
      if ((Yp[i]<=y && y<Yp[j] ||
           Yp[j]<=y && y<Yp[i]) &&
           x<(Xp[j]-Xp[i])*(y-Yp[i])/(Yp[j]-Yp[i])+Xp[i])
           cnt=!cnt;
   }
   return cnt;
}
```

Остава да дадем отговор на третия въпрос, за да решим задачата напълно. Горният алгоритъм не дава еднозначен отговор в различните случаи. Ако например точката лежи върху лявата граница на областта, поради строгото неравенство за x в условния оператор броят на пресечните точки ще бъде нечетен и точката ще се приеме за вътрешна. Ако обаче точката лежи върху дясната граница на областта, броят на пресечните точки ще е 0 и тя ще се счита за външна (фиг. 5-31).



Фиг. 5-31

Един прост начин да се реши този проблем е да се определи четността на пресичанията както на лъч в положителна посока, така и на лъч в отрицателна посока. Ако точката не принадлежи на контура, тези две четности ще са еднакви, а ако принадлежи - те ще бъдат различни. Това увеличава работата на алгоритъма, но пък гарантира правилното определяне на положението дори когато точката лежи върху хоризонтална страна.

Един възможен начин за реализиране на казаното е със следната програма:

```
int PointInPoly(Npol,Xp,Yp,x,y)
int Npol; float *Xp,*Yp; float x,y;
{int i,j,cntLeft,cntRight; float Xt;
   cntLeft=cntRight=0;
   for (i=0; i<Npol; i++) {
      j=(i+1)%Npol;
      if (Yp[i]<=Y && Y<Yp[j] || Yp[j]<=Y && Y<Yp[i]) {
         Xt=(Xp[j]-Xp[i])*(y-Yp[i])/(Yp[j]-Yp[i])+Xp[i];
         if (x<Xt) cntRight=!cntRight;
         else if (x>Xt) cntLeft=!cntLeft;
      }
   }
   return cntLeft==cntRight?cntLeft:ON_THE_BORDER;
}
```

Задачата за определянето на положението на точка спрямо многоъгълник може да се реши и по други начини, но представеният тук е най-ефективен. Друг възможен подход е да се изчисли сумата от ориентираните ъгли, които се сключват между вектора от разглежданата точка до началото на всяка страна и този от същата точка до края на страната. При пълно обхождане на многоъгълника тази сума би била 0°, ако точката е вън от многоъгълника и 360°, ако тя е във вътрешността му.

ЕЛЕМЕНТИ НА ОФОРМЛЕНИЕТО. ЩРИХОВАНЕ НА ОБЛАСТИ.

Щриховането на една област е задача, която е близка до горната. Тя се състои в намирането на множество от двойки точки, представляващи краищата на отсечки с определено направление, които изцяло принадлежат на вътрешността на тази област. За целта се извършва следното:

1. Областта се подлага на ротация с ъгъл, обратен на наклона на щрихлиниите, така че да се работи с хоризонтално щриховане.
2. Намира се правоъгълната обвивка на областта, т.е. минималните и максимални стойности на координатите на върховете. Ако областта съдържа криволинейни елементи, това няма да е толкова тривиално.
3. За всяка хоризонтална права се намират пресечните точки със страните на областта.
4. Полученият списък се сортира по нарастване на X и между всяка двойка точки (нечетна-четна) се чертае отсечка от щриха.
5. Координатите на отсечките се подлагат на ротация с ъгъл наклона на щрих-линиите и се прави обръщение към графичните примитиви за визуализация.

Един от основните въпроси, свързан със щриховката на областите е как да се съхранява тази щриховка в модела. Ако се съхраняват явно отсечките, които я образуват, промяната на областта няма да води до промяна на щриховката ѝ, което в повечето случаи е нежелателно. По-добро решение е да се съхраняват параметрите на щриховката - наклон, разстояние между щрих-линиите, тип на щрих-линиите, цвят и др. - като един от атрибутите на една област и при всяка визуализация на модела да се изпълнява описаният алгоритъм за генериране на щрих-линиите. Промяната на областта ще води и до промяна на щрихованието ѝ и няма да се претрупва модела със съхраняването на информация, нужна само за визуализация. Този принцип често се прилага и към други елементи на оформлението, каквито са оразмеряванията и текстът за анотации.

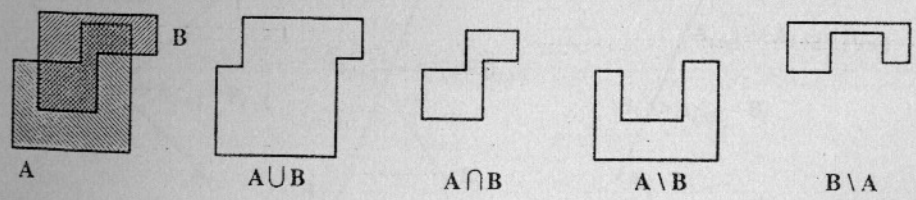
5.4 МОДЕЛИРАНЕ ЧРЕЗ БУЛЕВИ ОПЕРАЦИИ

Апаратът на *теоретико-множествените* (булеви) операции е много удобно средство за геометрично моделиране. Макар основното прилагане на тези операции да е в пространствените модели, те са много полезни и при работа с равнинни модели. Причината за предпочитанието на тези операции е от една страна техният ограничен брой: обединение, сечение и разлика, а от друга - близостта им до реалните действия, които инженерите извършват в своята практика: отрязване, слепване и т.н. Булевите операции могат да бъдат прилагани върху произволни обекти - дори и върху такива, които имат безкрайна площ - например полуравнината, лежаща от определена страна на една права. Ние тук ще разгледаме приложението им главно върху едносвързани области от вида, представен по-горе.

Дефиницията на булевите операции читателят знае от теория на множествата. Тук вместо множества ще говорим за области. Можем да смятаме, че една област е множеството от всички нейни вътрешни и контурни точки и тогава:

- *сечението* **A∩B** на две области е областта от всички точки, които принадлежат едновременно и на двете области;
- *обединението* **A∪B** на две области е областта от всички точки, всяка от които принадлежи на поне една от областите;
- *разликата* **A\B** е областта от всички точки, които принадлежат на първата, но не принадлежат на втората област.

Първите две операции са комутативни, докато третата очевидно не е.



Фиг. 5-32

По принцип моделирането чрез булеви операции не е пряко свързано с модела на приложната програма. То е елемент от концептуалното ниво на графичния диалог, както видяхме в предната глава. Реализирането на булеви операции обаче е различно за всеки различен геометричен модел. Изпълнението на булеви операции върху области, които са зададени чрез своите граници - елементите на граничноопределените модели – е твърде усложнено и се нуждае от специални алгоритми. Други модели – булевите конструктивни модели, моделите с равнинна декомпозиция и др. – имат структура, която е много по-подходяща за изпълнението на тези операции, поради което за тях те се реализират тривиално. В геометричното моделиране се използват т.нар. *регуляризирани булеви операции*, резултатът от които е винаги затворена област, т.е. границите ѝ принадлежат на самата област. По този начин се избягва получаването на висящи ръбове.

5.4.1 Булеви операции върху граничноопределени модели

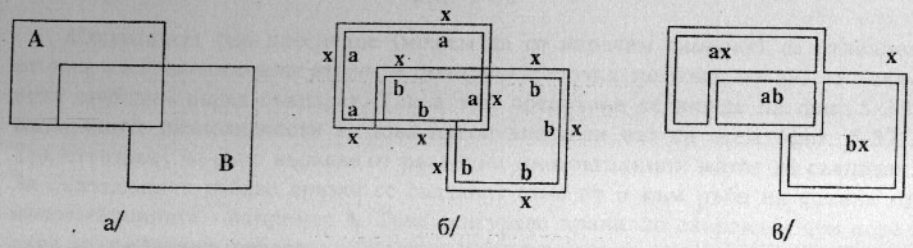
Съществуват редица алгоритми за изпълнение на булеви операции върху области, зададени чрез границите си. Сложността на алгоритмите зависи от сложността на областите, които се обработват. Тук ще се ограничим в разглеждането на области, чиито граници са само многоъгълници. Читателят ще види, че дори в този случай общата задача за две произволни едносвързани области не е тривиална. Тя е една от задачите на изчислителната геометрия, имаща много общо с отсичането, което разгледахме в трета глава. Алгоритмите на Садърланд-Ходжман, на Лян-Барски и на Сайръс-Бек за отсичане на практика извършват операцията сечение между многоъгълник и правоъгълник или между произволен и изпъкнал многоъгълник. Тук ще разгледаме един алгоритъм, който може да се приложи за произволни граничноопределени области (дори многосвързани). С него могат да се осъществят всички булеви операции: сечение, обединение и разлика.

АЛГОРИТЪМ НА УЕЙЛЪР. Алгоритъмът на Уейлър (Weiler), разработен като част от дисертацията на Кевин Уейлър пред 1980 год., е подобрение на първоначалния алгоритъм, предложен от Уейлър и Адертън (Atherton) още през 1977 год. Нека първо да направим някои общи наблюдения, преди да разгледаме детайлите на този алгоритъм.

Нека са дадени два произволни многоъгълника **A** и **B**. Нека за момент да си мислим, че ръбовете на многоъгълниците са двойни. На фиг. 5-33 двойните ръбове са нарисувани леко раздалечени само за да можем да илюстрираме алгоритъма. В действителност те съвпадат напълно. Всеки от двойните ръбове представлява отделна негова *страна* и може да бъде отбелязан с етикет в зависимост от положението му спрямо многоъгълника – този откъм вътрешността на съответния многоъгълник ще бъде отбелязан с **a** или **b** (фиг. 5-33б), а този от външната страна ще е отбелязан с **x**. Последователността от свързани страни тук ще наричаме *контури*. Контурите, получени при удвояването (и фиктивното раздалечаване на страните) се пресичат. Идеята на алгоритъма е да се свържат страните в нови контури, така че никои два контура да не се

пресичат, както е показано на фиг. 5-33в. Всеки от новополучените контури определя област, която ще има етикет, образуван от етикетите на ръбовете, които участват в него. Тези области съответстват точно на резултата от изпълнението на четирите булеви операции:

- $A \cap B$ е областта с етикет **ab**
- $A \cup B$ е областта с етикет **x**
- $A \setminus B$ е областта с етикет **ax**
- $B \setminus A$ е областта с етикет **bx**.



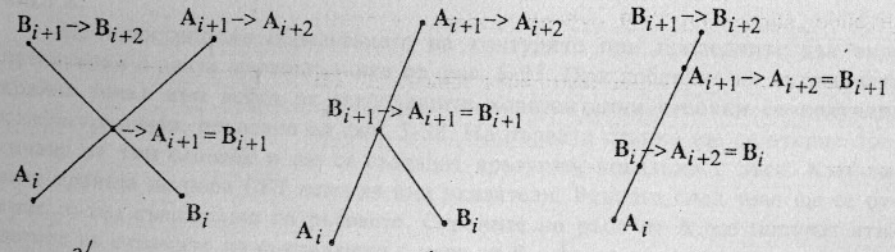
а/ б/ в/

Фиг. 5-33

Този алгоритъм има няколко стъпки:

1. Намиране на пресечните точки на двата многоъгълника и разширяване на списъците от върховете им така, че да включват и получените пресечни точки.
2. Удвояване на ръбовете на многоъгълниците за получаване на двойки *страни*, свързани в пресичащи се контури.
3. Свързване на страните в непресичащи се контури.
4. Отделяне на тези контури, които съответстват на желаната булева операция.

Първата стъпка може да се извърши самостоятелно. За намирането на пресечните точки на множество от отсечки в равнината в изчислителната геометрия са разработени бързи алгоритми. Може би по-ефективно е тази стъпка да се съчетае със следващите две (2-рата и 3-тата). Най-простата реализация би била да се пресече всеки ръб от единия многоъгълник с всеки ръб от другия и всяка намерена пресечна точка да се добави и към двата многоъгълника.



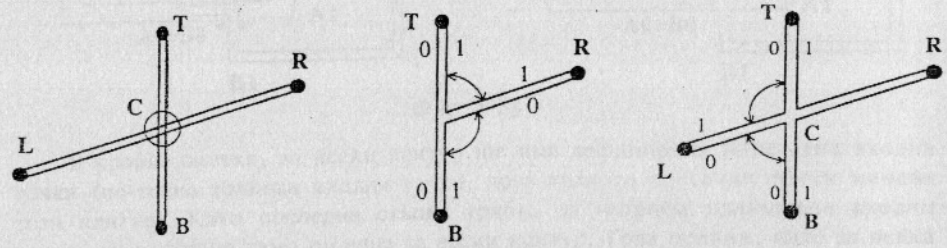
а/ б/ в/

Фиг. 5-34

Ако два ръба се пресичат в една точка, то тази точка се добавя между двете крайни точки на всеки от двата ръба - фиг. 5-34а. Ако пресечната точка е крайна точка само на един от ръбовете, то тя се добавя само в списъка на върхове на другия многоъгълник - фиг. 5-34б. Ако два ръба са колинеарни, всяка крайна точка на единия ръб, която принадлежи на другия, се добавя като негов връх - фиг. 5-34в.

Всеки ръб има две страни. Удвояването на ръбовете може да се направи, като се създадат две двойки списъци, елементи в които да са тези страни. Тези списъци съответстват на двете двойки първоначални контури, показани на фиг. 5-33б. Информацията за върховете на всяка от страните на един ръб ще е една и съща, но ще се различава свързаността им. Ще покажем по-долу обща структура за съхраняването на тази информация.

За да свържем подходящо страните в желаните контури, трябва да сменим подредбата на страните във всяка от пресечните точки на двата многоъгълника. Съществуват няколко типа пресечни точки и всеки от тях трябва да разгледаме поотделно. Най-често срещаният тип пресечна точка е получена от *кръстосани ръбове*. Една такава пресечна точка е точката **C**, получена от пресичането на ръбовете **BT** и **RL**, както е показано на фиг. 5-35.

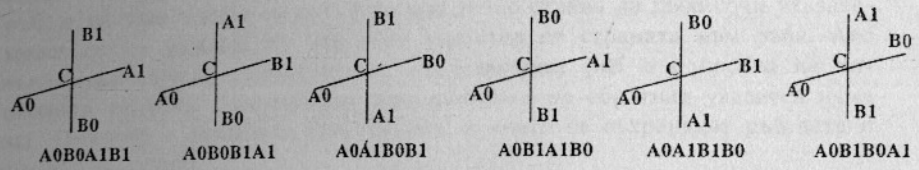


Фиг. 5-35

Пренареждането на страните за всеки един ръб става поотделно. Нека страните са номерирани - страна 0 на един ръб е тази от лявата му страна (спрямо посоката, зададена от крайните му точки - първоначалната ориентация на многоъгълника), а страна 1 е тази от дясната му страна. За да разберем към кои страни на вертикалните ръбове **BC** и **CT** да присъединим страните на ръба **RC** е необходимо само да определим положението на точката **R** спрямо отсечката **BT**. Тя лежи от дясната им страна (страна 1) и точките образуват наредена тройка **BRT** по посока, обратна на часовниковата стрелка. Това означава, че страна 0 на ръба **RC** трябва да се свърже със страна 1 на **BT**, а страна 1 на ръба **RC** трябва да се свърже със страна 1 на ръба **CT**. Аналогично ще постъпим и при свързването на ръба **CL**.

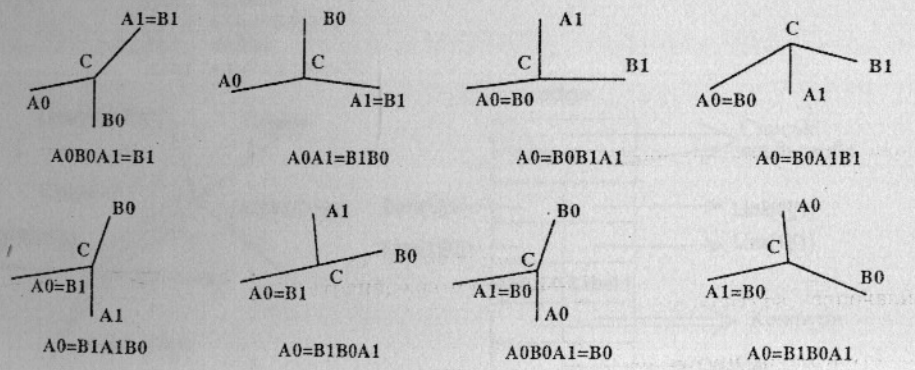
Друг вид пресечна точка е *допирната точка*. Тя прилича в голяма степен на предишната с тази разлика, че и двата ръба на единия многоъгълник лежат в една и съща полуравнина спрямо другия многоъгълник. Свързването на всяка страна става при определянето на положението ѝ спрямо останалите страни, които се събират в тази точка. Двата типа пресечни точки могат да се обобщят в една, като се разгледат всички възможни подредби на четирите

свободни върха **A0, A1, B0, B1** (фиг. 5-36) обикаляйки около пресечната точка по посока, обратна на часовниковата стрелка. Възможните подредби са шест и за всяка от тях има определен макет на свързване.



Фиг. 5-36

Следващият тип пресичане (можем да го наречем *сливане*) се отличава от това на *кръстосаните ръбове* и *допирните точки* по това, че два от четирите свободни върха съвпадат. Такъв тип пресичане се вижда на фиг. 5-33. Възможните разновидности на това пресичане този път са осем (фиг. 5-37). Тук отчитаме, че само върхове от различни многоъгълници могат да съвпадат. За съвпадащите ръбове връзки се създават само от и към ръба на единия от многоъгълниците - например **A**. Това осигурява правилно свързване при поредица от съвпадащи ръбове.

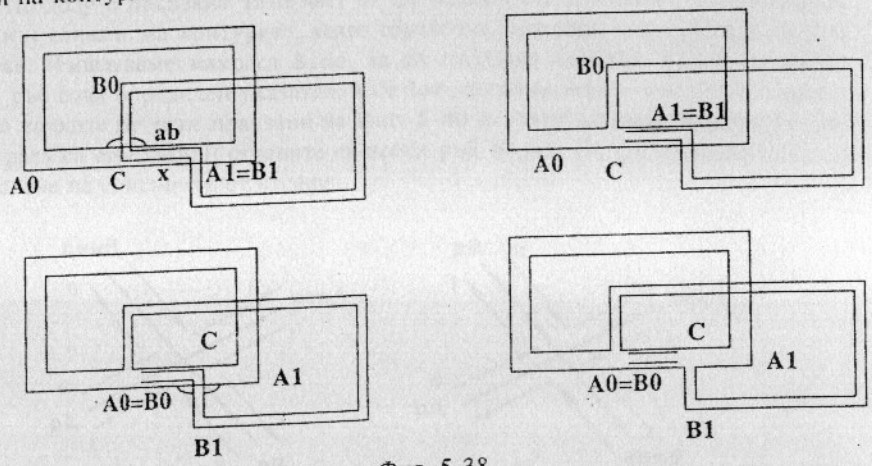


Фиг. 5-37

Пълното съвпадение на два ръба е друг особен вид пресичане. То се обработва сравнително лесно. Етикетите на всяка от страните на ръба, принадлежащ на многоъгълника **B**, се добавят към етикетите на ръба от многоъгълника **A**.

Ще илюстрираме образуването на контурите при последните два вида пресичания с двата многоъгълника от фиг. 5-33. При добавяне на вътрешната крайна точка към всяка от съвпадащите хоризонтални отсечки се получава конфигурацията, показана на фиг. 5-38. На първата стъпка ще се открие пресичане от тип *сливане* и ще се създадат връзките, показани с дъги. Към левите краища на ръба **CB1** няма да има указатели. Веднага след това ще се открие пълно съвпадение на ръбовете. Страните на ръба от **A** ще получат етикетите на страните на съвпадащия с него от **B** – **bx** за вътрешната страна и **x** за външната. На третата стъпка ще се създадат връзките, отбелязани с дъги и ръбът **B0C** ще се окаже несвързан в никой контур.

Нека сега да видим как да отделим желаните контури. При обработването на всяка пресечна точка ще дефинираме по една входна точка за всеки от свързаните контури. При пресечна точка от *кръстосани* или *допиращи се* ръбове създаваме четири входни точки за четирите контура, в които свързваме страните. Входната точка е просто един указател към страна, която принадлежи на контура.



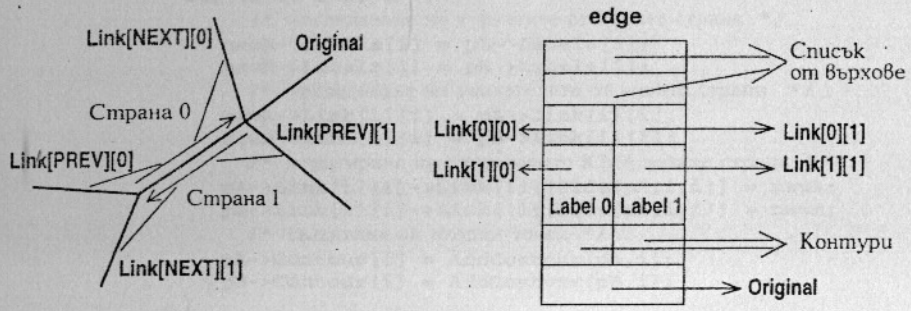
Фиг. 5-38

В крайна сметка, за всеки контур ще има дефинирана поне една входна точка (по-точно толкова входни точки, през колкото пресечни точки минава този контур). Като последна стъпка трябва да изтрием излишните входни точки, за да остане само по една за всеки контур. Това правим, като за всяка страна, която е входна точка, поддържаме указател към контура, на който тя принадлежи. Когато обхождаме контура, започвайки от произволна негова входна точка, проверяваме дали страните, през които минаваме, не са входни точки за други контури. Тези входни точки изтриваме и след пълното обхождане ще останем само с една единствена входна точка към контура. По време на обхождането образуваме и етикета на самия контур от етикетите на съставящите го страни.

Нека да обобщим сега какво е необходимо да се съхранява за всеки ръб, за да може да се извършват изброените обработки. Освен двата си върха, един ръб има две страни, които ще участват в различни контури и следователно ще бъдат независими една от друга. За всяка от тях може да се поддържа двусвързан списък (т.е. указатели към следващия и предишния ръб в контура), етикета ѝ и указател към контура, на който тя евентуално е входна точка. За страните, които не излизат от пресечни точки, този указател е винаги NULL. Двусвързаният списък е необходим за промяната както на указателите от страната, така и на указателите към нея, когато нейното свързване трябва да се промени. Тъй като е възможно създаването на контурите да се

извърши едновременно с намирането на пресечните точки при едно единствено обхождане на двата многоъгълника, то е удобно да се поддържа и още един елемент в структурата на ръба - следващият го ръб в списъка на многоъгълника. Забележете, че указателите в двусвързаните списъци са от страна към ръб, а не към страна от ръб. Към коя точно страна на ръба сочи указателят може да се разбере по обратния указател от страната към ръба. Ако двусвързаният списък е ориентиран - следващият ръб е в посока на часовниковата стрелка - двусмислие при намиране на обратния указател няма да има. По-долу е показана структурата, в която се съхраняват ръбовете и получените контури:

```
typedef struct vert *vptr;
typedef struct edge *eptr;
typedef struct cont *cptr;
typedef struct vert {float x,y; ptr next;} vert;

typedef struct edge {
          vptr Vert[2];
          eptr Link[2][2],
          int  Labels[2];
          cptr Contour[2];
          eptr Original;} edge;

typedef struct cont {
          int  Labels;
          eptr startEdge;
          int  side;
          cptr next;} cont;
```



Фиг. 5-39

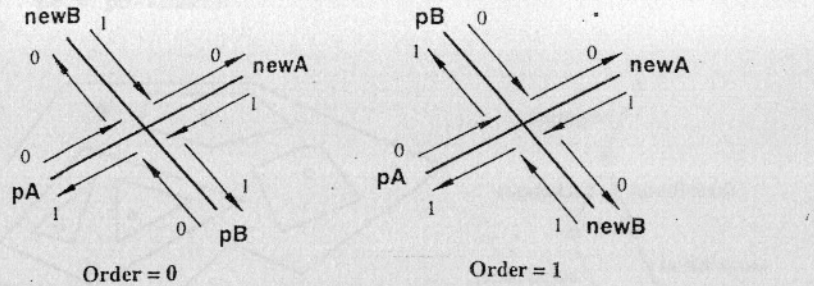
Реализирането на алгоритъма ще извършим по малко по-различен начин от описания дотук. Можем да съвместим някои от стъпките, посочени по-горе, като например намирането на пресечните точки и сливането на контурите. За целта първо зареждаме двата многоъгълника в предложените структури от данни. След това извършваме проверка за пресичане (по най-тривиалния алгоритъм: всеки срещу всеки ръб) и за всяка намерена пресечна точка сливаме контурите и създаваме входни точки към тях. Като последна стъпка изтриваме излишните входни точки и получаваме окончателния списък от контури.

Няма да се спираме подробно на всички етапи, а само на по-съществените от тях. Читателят може сравнително леко да напише сам функцията за зареждане на всеки от многоъгълниците в структурата от данни. Тази функция

връща указател към началния ръб на контура, а освен многоъгълника получава и етикета, който трябва да постави на всяка "страна 1" от ръбовете на многоъгълниците. Всички "страни 0" и на двата многоъгълника ще имат етикет LABEL_X.

```
pedge1 = LoadPolygon(NA,PolyA,LABEL_A);
pedge2 = LoadPolygon(NB,PolyB,LABEL_B);
IntersectPolygons(pedge1,pedge2);
DeleteExtraContours();
```

По-долу е показана тази част от функцията за намиране на пресечните точки и сливане на контурите, която обработва пресечна точка от кръстосани ръбове. Използваме макроса Side, за да получим към коя точно страна от един ръб сочи определен указател, и Order, за да намерим коя точно подредба на точките от тези показани на фиг. 5-40 е разглежданата в момента. На фигурата са отбелязани страните на всеки ръб (0 или 1), както и посоките на свързване на списъците от страни.



Фиг. 5-40

Следващата стъпка - изтриването на излишните входни точки и формирането на етикетите на контурите - се извършва при просто обхождане на списъка от получените контури. Движението по всеки контур се извършва отново с помощта на макроса Side.

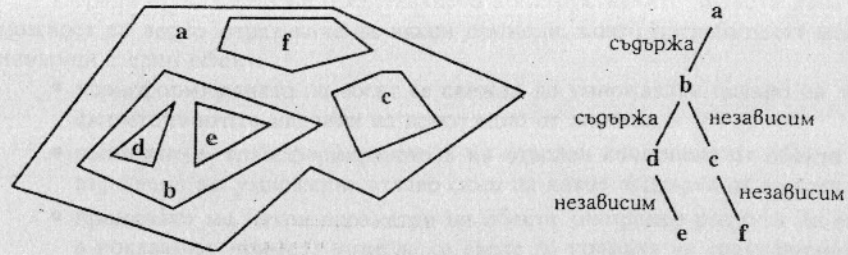
Казаното дотук е достатъчно за изпълнение на всички булеви операции над многоъгълници, но само ако двата многоъгълника имат поне една пресечна точка. При липса на пресечна точка, а също така и при прилагането на този алгоритъм за области (едносвързани и многосвързани), чиито граници са многоъгълници, резултатът от някоя булева операция може да не са няколко независими един от друг контура. Възможно е някои от тях да са вложени един в друг, образувайки по този начин границите на област.

За да намерим резултата от операцията е необходимо първо да променим дефиницията на списъка от контури, като му добавим още един указател към контур, превръщайки цялата структура в двоично дърво. Единият клон (например левият) на дървото ще представлява свързване на два независими един от друг контура, а другият (например десният) - влагане на един контур в друг. На фиг. 5-41 е показано едно примерно влагане на контури и дървото, което му съответства.

```
 /* намиране на страната, към която сочи указателят e->Link[i][side] */
#define Side(e,i,side)  e->Link[i][side]->Link[!i][1] == e
pctr ptop;        /* начало на списъка от получени контури */

void IntersectPolygons(pstartA,pstartB)
eptr pstartA,pstartB;  /* началните ръбове на двата многоъгълника */
{eptr pA,pB, newA, newB;
 int ord,i; Point ip;
   pA = pstartA;
   while (pA) {
     pB = pstartB;
     while (pB) {
         /* проверка за пресичане и намиране на типа на пресичането */
       switch (IntersectEdges(pA->Vert,pB->Vert,&ip)) {
       case CROSS:       /* кръстосани ръбове */
           /* намиране ориентацията на пресичането */
         ord = Order(pA->Vert[0],pA->Vert[1],pB->Vert[1]);
           /* създаване на новите ръбове, получени при пресичането */
         newA = malloc(sizeof(edge));
         newB = malloc(sizeof(edge));
         /* вмъкване на новите ръбове в списъците на многоъгълниците */
         newA->Original = pA->Original; pA->Original = newA;
         newB->Original = pB->Original; pB->Original = newB;
         /* записване на пресечната точка и указателите към нея */
         newA->Vert[1] = pA->Vert[1];
         newB->Vert[1] = pB->Vert[1];
         newA->Vert[0] = pA->Vert[1] =
               newB->Vert[0] = pB->Vert[1] = AddVertex(ip);
         for (i=0; i<2; i++) {
             /* наследяване на етикетите от новите страни */
           newA->Labels[i] = pA->Labels[i];
           newB->Labels[i] = pB->Labels[i];
             /* наследяване на указателите от новите страни  */
           newA->Link[i][i] = pA->Link[i][i];
           newB->Link[i][i] = pB->Link[i][i];
             /* коригиране на указателите КЪМ новите страни */
           pA->Link[i][i]->Link[!i][Side(pA,i,i)] = newA;
           pB->Link[i][i]->Link[!i][Side(pB,i,i)] = newB;
             /* създаване на входни точки */
           pA->Contour[i] = AddContour(pA,i);
           pB->Contour[i] = AddContour(pB,i);
         }
         /* сливане на контурите в четирите ъгъла на пресечната точка */
         pA->Link[ord][ord] = newA->Link[!ord][ord] = newB;
         pA->Link[!ord][!ord] = newA->Link[ord][!ord] = pB;
         pB->Link[ord][ord] = newB->Link[!ord][ord] = pA;
         pB->Link[!ord][!ord] = newB->Link[ord][!ord] = newA;
         break;
       case TOUCH: . . .        /* допиращи се ръбове */
         break;
       case COINCIDE: . . .     /* съвпадащи ръбове */
       }
       pB = pB->Original;      /* следващият ръб от B */
     }
     pA = pA->Original;        /* следващият ръб от A */
   }
}
```



```
void DeleteExtraContours()
{cptr pc;
 eptr ep;
 int side, i;
   pc = ptop;
   while(pc) {                /* за всяка входна точка */
        /* намиране на втората страна на контура */
      side = Side(pc->startEdge,NEXT,pc->side);
      ep = pc->startEdge->Link[NEXT][pc->side];
      while(ep != pc->startEdge){
         /* изтриване на входна точка, ако тя сочи към текущия контур */
         if (ep->Contour[side] != pc)
               DeleteContour(ep->Contour[side]);
         /* добавяне на етикета към множеството етикети на контура */
         pc->Labels |= ep->Labels[side];
         /* намиране на следващата страна на контура */
         i = Side(ep,NEXT,side);
         ep = ep->Link[NEXT][side];
         side= i;
      }
      pc = pc->next;
   }
}
```



Фиг. 5-41

Като последна стъпка на този алгоритъм е необходимо да се състави дървото на влагане за получените контури. При това съставяне трябва да се решава задачата за взаимно разположение на два контура, което може тривиално да се сведе до задачата за принадлежност на точка към област, която разгледахме по-горе. От това дърво лесно могат да се конструират няколко независими едносвързани области с един външен и няколко вътрешни граници - един ляв клон и всички последователни негови десни.

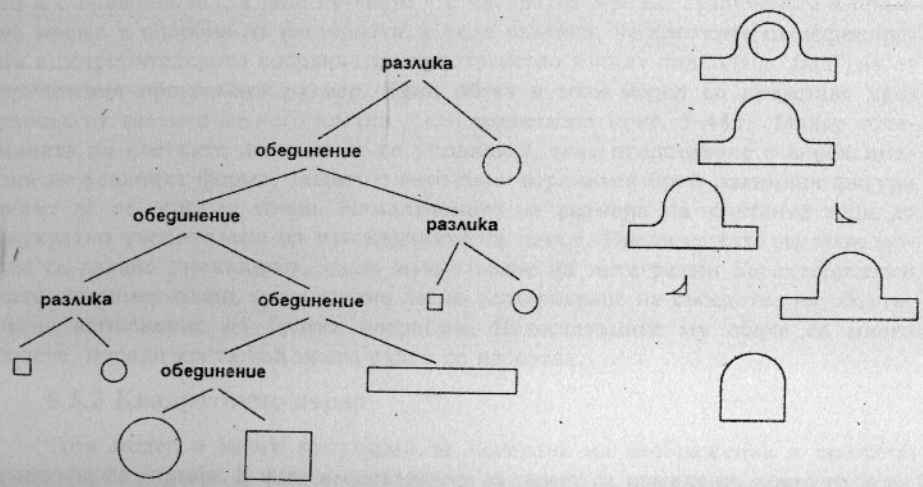
5.4.2 Булеви конструктивни модели

Както видяхме, реализирането на булеви операции върху граничноопределени обекти не е тривиално, особено когато трябва да се разгледат всички възможности и особени случаи на разположение и тип на обектите. Изпълнението на булевите операции по този начин е оправдано само ако границите на обектите са важен компонент от модела и се налага потребителят непрекъснато да работи с тях. Ако тези граници се използват само за генериране на изображение, то могат да се намерят по лесни начини за визуализация на резултата от една булева операция. При изобразяване върху растерни дисплеи

например не винаги е нужно да се знаят границите на обекта. Най-тривиален е случаят, когато обектите, с които се работи, са запълнени. Обединението на два обекта от този тип може да се визуализира като просто се запълнят двете области с еднакъв цвят.

Друго неудобство на използването на булеви операции в граничноопределени модели е невъзможността да се възстановят обектите след като върху тях е изпълнена поредица от булеви операции. Това може да се осъществи само ако се съхраняват границите на първоначалните обекти на всички междинни етапи или по-просто - да се съхраняват границите на първоначалните обекти и поредицата от булеви операции, която е приложена към тях.

Тези изисквания водят естествено до въвеждането на специален вид модели - *булеви конструктивни модели* (наричани при моделиране на пространствени форми Constructive Solid Geometry), които представят начина на създаването на даден обект чрез прилагането на булеви операции върху определен малък набор от примитиви. Историята (процедурата) на създаване е съставена само от булеви операции и създаване на примитиви. Тъй като операциите са двуместни, даден обект може да бъде представен като двоично дърво - *конструктивно дърво*, възлите в което са булеви операции, а листата - създаване на екземпляри от примитивите.



Фиг. 5-42

На фиг. 5-42 е показано конструктивното дърво на един обект, като от дясната страна са изобразени всички междинни обекти, които се получават при изпълнението на булевите операции на всяко ниво в йерархията. Моделът е изключително прост за сметка на усложняването на някои от процедурите, които го обработват. Например за да се визуализира обект от такъв модел върху векторно устройство, каквото е плотерът, е необходимо да бъдат изпълнени операциите в дървото, прилагайки току-що разгледания алгоритъм, за да се получат контурите на тази област.

Сложността на обработките зависи в голяма степен и от това колко са

сложни примитивите, които се намират в листата на дървото. В показания пример това са само правоъгълник и окръжност. В много случаи те могат да се окажат недостатъчни. Повечето моделиращи системи предлагат работа с прости двумерни фигури, защото потребителят може по-лесно да си представи създаването на желания обект чрез тях. Той може да асоциира действията си с реални операции: разлика - отрязване с определен вид инструмент; обединение - слепване на обекти и т.н. Въпреки това не са изключени случаи и на използването на полуравнини като примитиви, което позволява моделирането на по-широк клас от обекти, дори и на такива с безкрайна площ.

Създаването на екземпляр от един примитив се извършва чрез задаването на набор от стойности за параметрите му (радиуса на окръжността; ширината и височината на правоъгълника и др.), които се съхраняват в листата на конструктивното дърво. Често предпочитан начин е да се съхранява трансформацията, на която е подложен фиксиран (непараметричен) примитив преди използването му като аргумент на булева операция. В показания пример това може да се постигне, като примитиви са само единичната окръжност и единичният квадрат, а се съхранява афинната трансформация - например като 3x3 матрица - за всеки използван екземпляр.

Поради простотата на представянето конструктивните дървета дават възможност за лесно отразяване на някои промени, които потребителят може да извърши с един обект:

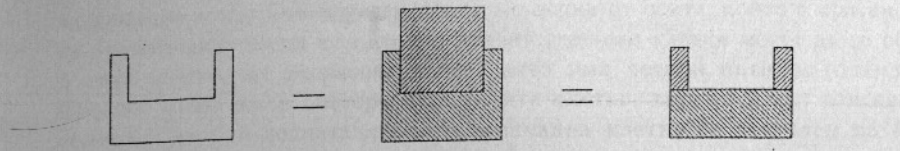
- трансформирането на обект се свежда до умножаване отляво на трансформационните матрици на всяко едно от листата;
- възможно е трансформирането и на отделен компонент от обекта - извършване на умножение отляво само на някое поддърво от модела;
- промяната на някои параметри на обекта (например радиуса на отвора в показания пример) може да се сведе до промяна на трансформационната матрица на едно или няколко листа;
- премахването на даден компонент от обекта може да се извърши като просто се изтрие определен клон от дървото.

Много моделиращи системи предлагат на потребителя средства за модификация пряко върху *конструктивното дърво*, след което системата визуализира обекта, който то описва.

Конструктивните дървета имат и много недостатъци, най-важните от които са:

- *недостатъчна мощност*: При фиксиран набор от примитиви те не могат да представят произволен обект;
- *неяснозначност*: Един и същ обект може да се представи с няколко различни конструктивни дървета, както се вижда от примера, показан на фиг. 5-43.

В практиката (особено при моделиране на пространствени обекти), този тип модели най-често се съчетават с едновременно използване и на гранични или контурни модели. Както се вижда, използването на равнинни примитиви, които имат площ, има доста удобства. Фигурите с площ са в основата на следващите модели, които ще разгледаме в тази глава.



Фиг. 5-43

5.5 МОДЕЛИ С РАВНИННА ДЕКОМПОЗИЦИЯ

Моделите, построени на принципа на т.нар. *равнинна декомпозиция* са няколко различни вида. При всеки от тях една равнинна фигура се представя като множество от съседни, непресичащи се базови плоски елементи. Наборът от тези елементи и начините на структурирането им е различен за всеки модел. Тук ще разгледаме по-важните от тях.

5.5.1 Модел на заетите клетки

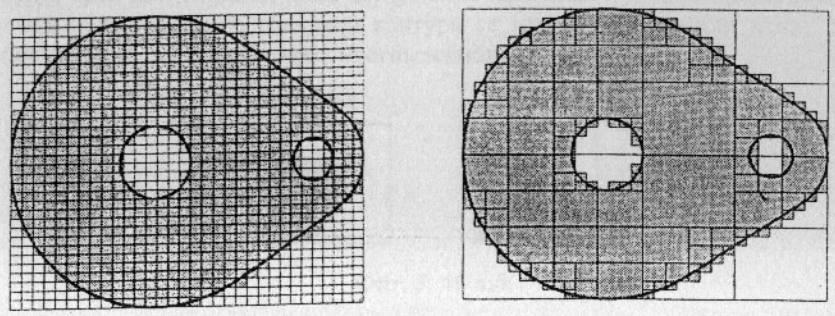
В този тип модел равнината се разделя на *клетки* (cells) чрез покриването ѝ с правоъгълна, а още по-често - с квадратна мрежа. Получената клетъчна мрежа е подобна на растерната, с тази разлика, че клетките са дефинирани в потребителското координатно пространство и имат подходящо подбран от приложния програмист размер. Един обект в този модел се представя чрез списък от заетите от него клетки - cell enumeration (фиг. 5-44а). Макар големината на клетките да може да се управлява, това представяне е апроксимация на реалната форма, защото с него само ограничен брой равнинни фигури могат да се опишат точно. Намаляването на размера на клетките води до двукратно увеличаване на изискванията за памет. Предимствата на този модел са главно уникалност, лесно изчисляване на интегрални характеристики като например площ, сравнително лесно установяване на съседство на обекти, лесно изпълнение на булеви операции. Недостатъците му обаче са много повече, поради което той много рядко се използва.

5.5.2 Квадратично дърво

Този модел е много популярен за кодиране на изображения и прилича донякъде на горния. В него изискванията за памет са намалени, което го прави удобен за много приложения. Основната идея е прилагането на принципа "разделяй и владей" към разделянето на равнината на части. *Квадратичното дърво* (quadtree) се получава при последователно разделяне на равнината (или на правоъгълника, задаващ границите на потребителското пространство) на квадранти (всеки от които е с еднаква големина), по начина показан на фиг. 5-44б. За опростяване в конкретния пример е прието, че обектът покрива цялото потребителско пространство, което е разделено на правоъгълни квадранти.

Всеки от квадрантите може да бъде *празен* (**E**), *пълен* (**F**) или *частично запълнен* (**P**) от обекта, който искаме да представим. Частично запълнените

квадранти от своя страна се разделят на нови четири квадранта, които отново се класифицират по показания начин. Подразделянето продължава дотогава, докато в листата няма частично запълнени квадранти или големината им е достигнала някаква приемлива за приложната програма минимална стойност. Тази стойност задава и максималната дълбочина на влагането на квадранти.

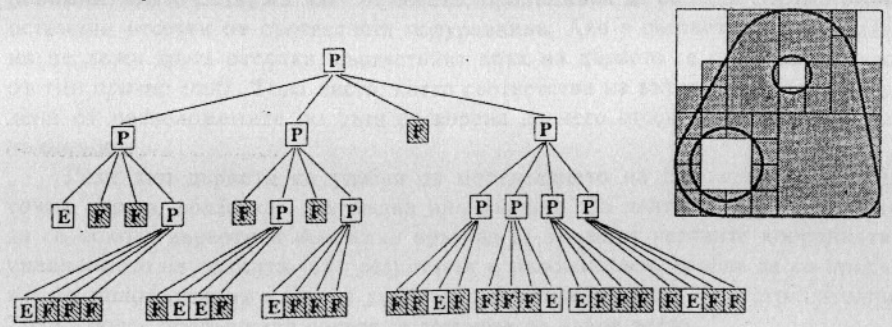


Фиг. 5-44 а,б

Може да се покаже, че броят на върховете в квадратичното дърво на един обект е пропорционален на периметъра на контура му. Този модел също е апроксимиращ в общия случай, като апроксимирането може да се извърши по два начина:

- представяне на площта с излишък (контурът е покрит изцяло от *пълни* клетки) или
- представяне на площта с недостиг (контурът е покрит от *празни* клетки).

На фиг. 5-45 е показано квадратичното дърво на един обект (със сравнително груба апроксимация - само четири нива на подразделяне), при което обхождането на квадрантите е в посока на часовниковата стрелка, започвайки от този в горния ляв ъгъл.



Фиг. 5-45

Едно квадратично дърво може да се състави директно от модела на заетите клетки по принципа "отдолу-нагоре". Ако четири съседни клетки, долна-

та лява от които има пореден номер по всяка от осите, който е кратен на две, са изцяло запълнени или изцяло празни, то тези клетки могат да се обединят в една клетка от по-високо ниво, която има четири пъти по-голяма площ. Същото може да се повтори и за новите клетки, като този път изискването е номерът на най-долната лява първоначална клетка да е кратен на 4. Продължавайки по този начин за всички по-горни нива и увеличавайки кратността по степените на 2, ще получим всички *пълни* и *празни* клетки в дървото.

Читателят познава различни методи за представяне на дървета. Квадратичното дърво има някои особености, които позволяват неговото удобно представяне в линеен списък. Най-важната от тези особености е фиксираната дълбочина. Това дава възможност всеки връх да се кодира с число, което има толкова квадратични разряда, колкото са и нивата в дървото. Едно линеаризирано квадратично дърво е списък от пълни върхове в сортиран вид. Всеки елемент от списъка е адресът на (или пътят за достигане до) съответния връх. При кодирането на този адрес сме използвали номерацията на квадрантите, приета по-горе: в посока на часовниковата стрелка, започвайки от този в горния ляв ъгъл.

За да се представят листа, които не са разположени на последното ниво в дървото, се въвежда специален терминален символ X, изисквайки по този начин три двоични разряда за всяка цифра от адреса. Линеаризацията на показаното на предишната фигура дърво ще бъде:

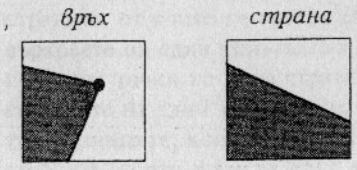
```
01X, 02X, 031, 032, 033,
10X, 110, 113, 12X, 131, 132, 133,
2X,
300, 301, 303, 310, 311, 312,
321, 322, 323, 330, 332, 333
```

Квадратичните дървета позволяват лесно изчисляване на интегралните характеристики на обектите, които представят. Булевите операции - сечение, обединение и разлика се изпълняват също много лесно. Съществуват и сравнително прости алгоритми за намиране на съседната на дадена клетка (или клетки) в определена посока. Както и при предишното представяне, само някои трансформации се изпълняват лесно. Това са само: хомотетия с коефициент, който е степен на 2, ротация на ъгъл, кратен на 90° и осева симетрия относно координатните оси. Всички останали се реализират нетривиално. Намирането на образа на едно квадратично дърво при афинна трансформация е нелек за решаване проблем.

Справянето с този проблем, както и осигуряването на точност на представянето без непременно намаляване на размера на минималната клетка е довел до някои модификации в квадратичните дървета. Най-интересната от тях е използването на *квадратични дървета от прости многоъгълници* (polytrees). При тези дървета се запазва общата структура на представянето с тази разлика, че към стандартните три вида листа - пълни, празни и частично запълнени, се добавени и две нови - *връх* и *страна*. Една клетка е от тип *връх* ако в нея се намира само един връх и две страни, които се събират в него. За такава клетка се съхраняват най-често двете прави на страните и информация

за това от коя страна на всяка от тях се намира вътрешността на обекта (фиг. 5-46а). Клетките от тип *страна* съдържат само една страна, както е показано на фиг. 5-46б.

Образуването на тези дървета става чрез последователно разделяне на равнината на клетки, докато всички те са от един от четирите вида: *пълна*, *празна*, *връх* или *страна*. Това допълнение позволява точното представяне на всички области, чиито гранични контури са многоъгълници или могат да се апроксимират подходящо чрез многоъгълници.



Фиг. 5-46 а,б

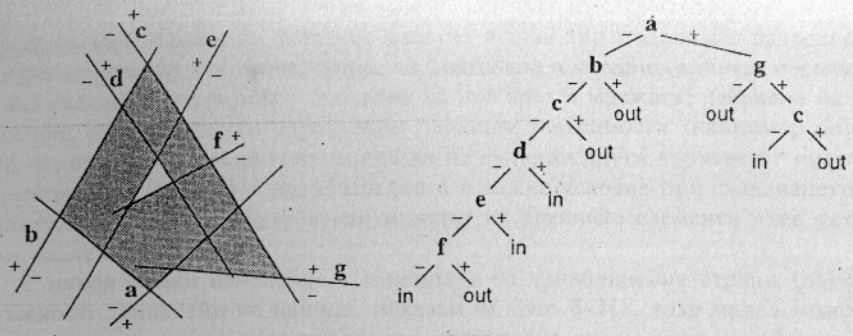
5.5.3 Дървета с двоично делене на равнината

За разлика от квадратичните дървета, при които равнината се разделя на части, независими от вида на обектите, при т.нар *дървета с двоично делене на равнината* (Binary Space Partitioning – BSP-дървета), тя се разделя на части от елементите на самия обект. Този модел е удобен за представяне на области, зададени с контури от многоъгълници. Обектите в него са инвариантни по отношение на прилаганите към тях трансформации. Всяка отсечка от граничните контури се представя с носещата я права, която разделя равнината на две полуравнини - "положителна" и "отрицателна" спрямо посоката на нормалния вектор на правата. Ако всяка права се задава с алгебричното си уравнение $Ax+By+C=0$, този нормален вектор е векторът (A,B).

BSP-дървото на една област се създава, като се започне от една отсечка и се раздели равнината спрямо нея на две полуравнини. Всяка от тези полуравнини, която съдържа част от обекта, продължава да се дели спрямо всички останали отсечки от съответната полуравнина. Ако в съответната полуравнина не лежи друга отсечка, съответният връх на дървото се превръща в листо от тип *празно* (out). Това листо, което съответства на вътрешна област, заградена от разположените по пътя от корена до него отсечки, се маркира като *пълно* (in).

Този тип дървета са удобни за определянето на положението на една точка спрямо областта - вътрешна или външна. За целта е необходимо само да се обходи дървото и във всеки връх да се заместят неговите координати в уравнението на правата. Ако резултатът е положителен, трябва да се продължи по положителния клон на дървото, в противен случай - по отрицателния. Този процес продължава докато се достигне до някое листо.

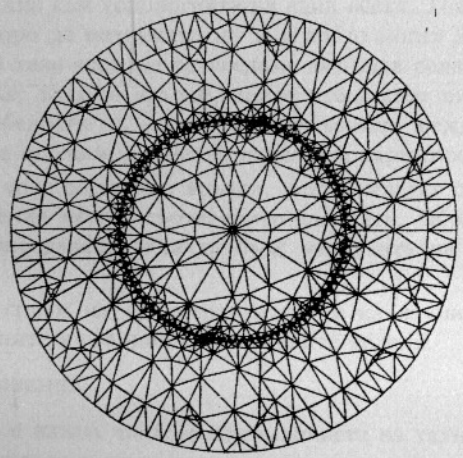
Описаният модел дава възможност за представяне и на области, които не са затворени, което за някои приложения е удобно, а за други е напълно неприемливо. Друг недостатък е, че обектите в тях нямат уникално представяне, което може да бъде проблем при извършването на някои видове анализи.



Фиг. 5-47

5.5.4 Триангулачна мрежа

Съществуват и много други модели, които се основават на разделянето на равнината на съставни компоненти. Едни използват зададен набор от плоски фигури: триъгълник, правоъгълник, сегмент, сектор, окръжност, които могат да бъдат "слепвани" една за друга, задавайки по този начин вътрешността на обекта, който се моделира. В други модели фигурите могат да бъдат наслагвани една върху друга, всяка нова припокривайки предишните. Във всеки от тях обаче представянето на един и същ обект в общия случай може да бъде извършено по много различни начини. Тази неуникалност ги прави неудобни за голяма част от приложенията.



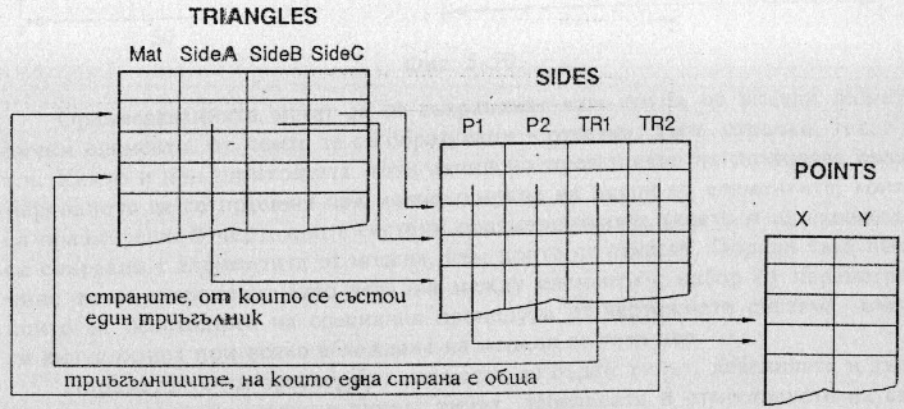
Фиг. 5-48

Един специален модел от този вид е *триангулачната мрежа*. Всяка област при него се разбива на триъгълници, а всички области заедно (които са в повечето случаи комплиментарни) образуват мрежа от съседни един за друг триъгълници. Това е типичен модел при решаване на физически задачи по метода на крайните елементи, търсеното скаларно или векторно поле в които

се задава с частни диференциални уравнения. Удобството на този модел е в много малкия набор от обекти, с които се работи и лесното извършване на повечето операции с тях. Това е най-вече за сметка на точността на представянето, за увеличаването на която е необходимо значително увеличаване на броя на елементите в модела.

На фиг. 5-48 е представена една триангулация на областите, показани на фиг. 5-28, при което дъгите от окръжности са апроксимирани с начупени линии. Геометричните задачи, които се решават в такъв модел са най-вече задачи за съседство:

1. намиране на страните, от които се състои един триъгълник,
2. намиране на върховете на един триъгълник,
3. намиране на крайните точки на една страна,
4. намиране на съседните на един триъгълник триъгълници,
5. намиране на триъгълниците, които имат общ връх в дадена точка,
6. намиране на страните, които имат за връх дадена точка,
7. намиране на точките, които са съседни на дадена точка и др.



Фиг. 5-49

Една примерна организация на данните е показана на фиг. 5-49. От всеки триъгълник има указатели към трите му страни и описание на характеристиките на материала на този триъгълник. Всяка от страните пък е зададена с двойката си крайни точки и двата триъгълника, на които принадлежи. Точките се задават само с двойката си координати. Обособяването на страните като отделни елементи на модела е необходимо за директното решаване на първите четири задачи за съседство. Останалите се решават чрез обхождане на показаните структури. Ето например как може да се реализира обхождането за намиране на всички точки, които са съседни на дадена точка:

```
for (i=0,i<NSIDES,i++) {
    if (Sides[i].P1==ThePoint) Neighbor[n++]=Sides[i].P2;
    else if (Sides[i].P2==ThePoint) Neighbor[n++]=Sides[i].P1;
}
```

Друг вид обработки, които се налагат в този тип модели са: разделянето на една страна на две части; смяна на диагонала в четириъгълника, образуван от два съседни триъгълника; добавяне на нов връх в мрежата; добавяне на нов връх при осигуряване на определени глобални зависимости (например описаната окръжност на всеки триъгълник да не съдържа други върхове от мрежата - критерий на Делоне) и др. Последното е важно условие при създаването на триангулачни мрежи, върху които методът на крайните елементи дава устойчиви числени резултати.

С някои малки изменения - въвеждане на криволинейни страни (дъги от окръжности, зададени по начина, показан на фиг. 5-21), този модел може да се използва за точно представяне на извънредно голям клас от обекти без непременно увеличаване на броя на елементите в модела.

5.6 МОДЕЛИРАНЕ НА РЕЛАЦИИ МЕЖДУ ЕЛЕМЕНТИТЕ

В разгледаните дотук модели връзките между отделните компоненти бяха само топологични - кой елемент от кои други елементи се състои. Това са връзките на подчиненост в йерархичните модели; указателите от един елемент към точките, от които той се определя; указателите от една област към граничните ѝ контури; указателите от един квадрант към четирите негови части в квадратичното дърво и т.н. Всички тези връзки са еднопосочни, затова и почти всички модели, които разгледахме по-горе, са йерархични.

В моделирането, много често се налага да се представят и други връзки, които нямат отношение към топологията на един обект. Това са релации, които се отнасят по-скоро до метриката му и в които могат да влизат един или няколко елемента. В тази част ще разгледаме два вида релации:

- *оразмерявания*: това са връзки, които представят визуално някои геометрични особености на модела - разстоянието между две точки, ъгълът между две отсечки, диаметърът на една окръжност и др.
- *ограничения*: това са такива връзки, които налагат определени геометрични отношения между съответните елементи. Обратно на оразмеряванията, те определят каква да бъде метриката на един геометричен обект.

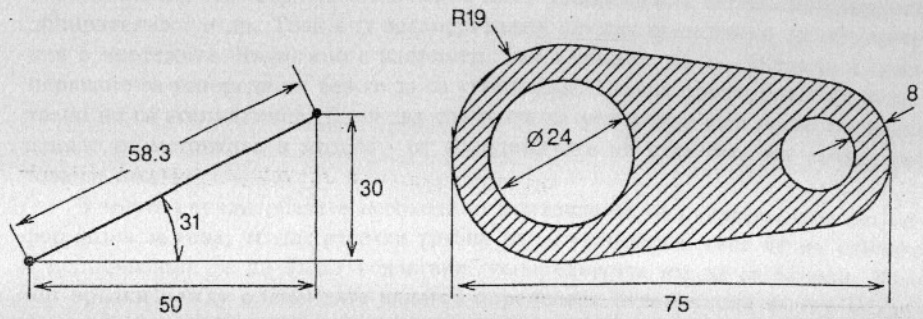
Нека първо се спрем на оразмеряванията и на начина, по който те се представят в контурните и граничноопределените модели.

5.6.1 Оразмерявания

Оразмеряването е важна част от оформлението на техническите чертежи и затова на него се отделя особено внимание при разработването на чертожни системи. Ние ще се спрем на представянето на оразмеряването в геометричния модел, а не на неговото изобразяване. Последното е обект на дисциплини като *техническо чертане*, както и на различните стандарти за изготвяне на техническа документация.

Чертожниците работят със следните видове оразмерявания (фиг.5-50):

- *линейно оразмеряване*: то изобразява хоризонтално или вертикално разстояние, както и разстояние по продължението на произволна ос между два елемента. Тези елементи могат да бъдат точки или контурни елементи (най-вече криволинейни).
- *ъглово оразмеряване*: това е ъгълът, който сключват две отсечки; наклонът на отсечка спрямо хоризонталната или вертикална ос; ъгълът между три точки; централният ъгъл на една дъга и др.
- *диаметрално оразмеряване*: то показва диаметъра (или радиуса) на окръжност или дъга от окръжност.



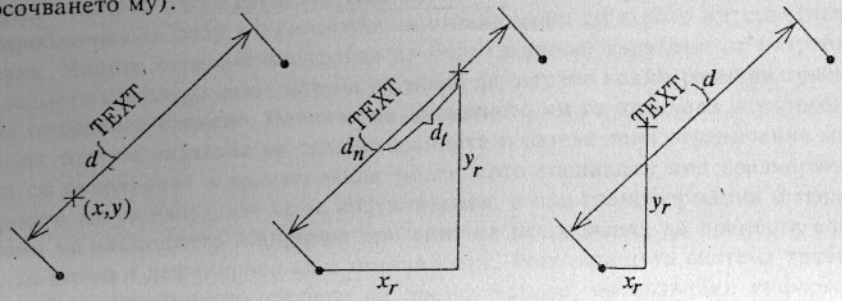
Фиг. 5-50

Оразмеряванията могат да се съхраняват като група от всички геометрични елементи, от които те са образувани - отсечки, дъги, стрелки, текст и т.н. Както и при щриховката, този начин на представяне не позволява оразмеряването да се променя при модифициране на някой от елементите, които са оразмерени. В чертожните системи оразмеряванията (както и щриховката) са свързани с елементите от модела, към които се отнасят. Поради тази причина те се съхраняват като релации между елементи с набор от параметри, които са необходими на специална процедура от чертожната система, която ги визуализира при всяко обхождане на модела за тази цел.

Параметрите на оразмеряването могат да бъдат: типът, дебелината и дължината на ограничителните линии; типът, дебелината и отместването на самата оразмерителна линия; мястото на текста, неговият наклон, положението му спрямо оразмерителната линия (под нея, над нея, прекъсвайки я и т.н.); видът на краищата на оразмерителната линия (тип на стрелките) и др. Текстът може да бъде съхраняван като низ от символи, а може да се генерира и чрез измерване на истинското разстояние между оразмерените елементи непосредствено преди визуализацията му. Последният вид оразмерявания често се наричат *асоциативни*. Самите оразмерени елементи са най-важните параметри на оразмеряването - при един и същ тип оразмеряване те могат да бъдат точки, отсечки, дъги и др.

Друг важен параметър е местоположението на оразмеряването (фиг.5-51). Най-често за тази цел се съхраняват координатите на точка, през която минава оразмерителната линия или позицията, в която е разположен оразмерителният текст. Съхранените координати могат да бъдат както абсолютни, така и относителни - спрямо определена характеристична точка на някой от оразмерените елементи (или спрямо средната точка на две характеристични

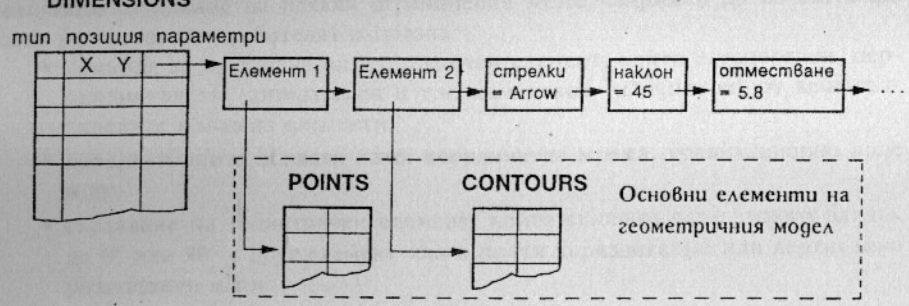
точки от двата оразмерени елемента). В първия случай посочването на оразмеряване ще бъде улеснено, но при преместване на оразмерените елементи трябва изрично да се променят и абсолютните координати. Обратно, съхраняването на относителни координати би довело в голяма част от случаите до адекватна трансформация на оразмеряването при положение, че се модифицират оразмерените елементи, но пък би затруднило неговата идентификация (посочването му).



Фиг. 5-51

Изборът на точка (или точки), спрямо които да се съхраняват относителните координати може да не е тривиален, особено при оразмеряване на криволинейни елементи. И в двата случая местоположението е по-добре да се съхранява в потребителски, отколкото в чертожни координати. Ето един възможен начин за представяне на линейно оразмеряване:

```
линейно оразмеряване :  [ тип оразмеряване;  оразмерен елемент;
                           оразмерен елемент; параметри ]
тип оразмеряване :  [ 'хоризонтално' | 'вертикално' | 'по_ос' ]
оразмерен елемент :  [ точка | елемент ]
параметри :  [ позиция текст; отместване на линията; тип оразм. линия;
               вид стрелки; височина на огран. линии; тип огран. линии;
               центриране на текста; наклон на текста ]
```



Фиг. 5-52

Предложената схема може да се реализира и като се приеме набор от стойности за параметрите по подразбиране (или такива, които са общи за

всички оразмерявания и се променят интерактивно от потребителя), а за всяко конкретно оразмеряване има списък от параметри, които са специфични за него. Подобен механизъм се предоставя наготово от обектноориентираните езици при структуриране в класове и подкласове.

5.6.2 Геометрични ограничения

В много системи се поддържат и друг вид оразмерявания - те изобразяват определен вид геометрични отношения като успоредност, перпендикулярност, допирателност и др. Този вид оразмерявания служат единствено за обозначения в чертежите. Възможно е например, два елемента да са свързани с оразмеряване за успоредност без те да са успоредни. Такива оразмерявания естествено не са асоциативни. Дали два елемента са успоредни или не се определя изцяло от метриката в модела - от координатите на крайните им точки, ако това са отсечки.

В много случаи обаче е необходимо в модела да се съхрани изрично информация за това, че две отсечки трябва да са успоредни, така че на каквито и модификации те да бъдат подлагани, успоредността им да се запази. Този тип връзки между елементите налагат определени ограничения върху метриката им и оттам носят и името си - *ограничения* (Geometric constraints). В този смисъл, на ограниченията може да се гледа като на част от метриката или на нейна алтернатива. Представянето на ограниченията в един граничноопределен модел неизбежно води до редундантност на метричната информация. Затова и обработките в такъв модел са по-сложни.

Ограниченията биват няколко вида:

- *явни*: това са ограничения, които включват допълнителен параметър - скалар, чиято стойност се въвежда и управлява от потребителя. Те съответстват в голяма степен на оразмеряванията, които разгледахме накратко по-горе:
  - зададено хоризонтално разстояние между два елемента;
  - зададено вертикално разстояние между два елемента;
  - зададено разстояние по наклонена ос между два елемента;
  - зададен ъгъл между два елемента;
  - зададен радиус или диаметър на окръжност или дъга от окръжност и др.
- *неявни*: това са ограничения в метриката на един или повече елемента, които не включват допълнителен параметър:
  - инцидентност;
  - успоредност;
  - перпендикулярност;
  - допирателност;
  - хоризонталност, вертикалност;
  - равни дължини между две двойки точки и др.

Ограниченията са удобно средство за моделиране, както ще видим по-късно в тази глава, защото те най-точно отразяват изискванията на потребителя при моделирането на геометричните особености на един обект. За да се

избегне редундантността на метричната информация при използване на ограничения, най-често се прибягва до използване на ограниченията в качество на метрика за геометричния модел. Това са съвсем различни по тип модели, които дават възможност за изменяне на формата съобразно наложените върху нея ограничения.

АВТОМАТИЧНО ГЕНЕРИРАНЕ НА ОГРАНИЧЕНИЯ. Един от сериозните проблеми при работа с геометрични ограничения е тяхното интерактивно задаване. Явните ограничения трябва да бъдат *изрично* зададени от потребителя, защото моделиращата система не може да отгатне какви размери трябва да има определен елемент. Начинът на задаването им се отличава извънредно малко от този на задаване на оразмеряванията и затова тези ограничения могат да се съхраняват в геометричния модел като специален вид оразмеряване, които ще се използват не за визуализация, а при трансформации и модификации на елементите. Например при опит на потребителя да премести елемент, за който е дефинирано явно ограничение, моделиращата система трябва да ограничи степените на свобода на преместването, използвайки информацията, съхранена в модела, така че зададеното оразмеряване-ограничение да остане в сила.

Съвсем друг е подходът при работа с неявни ограничения. Възможно е те да бъдат съхранявани в модела по същия начин като явните, т.е да се съвместят с оразмеряванията. Ограничението за перпендикулярност например може да се съхранява като явно ограничение за ъгъл между съответните точки, който е равен на 90°. В този случай обаче приложната програма трябва да може да направи разлика между ограничение за перпендикулярност и такова за ъгъл, който е 90°, т.е. зададен е явно от потребителя и той може впоследствие да промени стойността на параметъра.

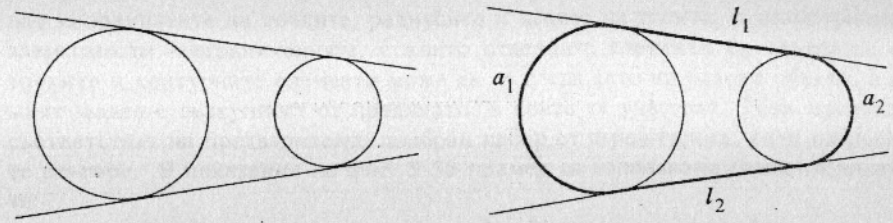
Неявните ограничения е по-добре да се отделят от явните не само защото за първите не трябва да се предоставя достъп до никакъв параметър. Друга причина е, че явните ограничения не е нужно винаги да бъдат задавани изрично от потребителя - те могат да се генерират в модела автоматично. Автоматичното създаване на неявни ограничения може например да се активира всеки път когато потребителят използва:

- команда за създаване на геометричен елемент, който е успореден, перпендикулярен, допирателен и т.н. към някой (или някои) от вече създадените в модела елементи;
- интерактивните похвати като: координатна мрежа, гравитационно поле и др.
- създаване на геометричен елемент, който сключва ъгъл, много близък до 0° или 90° с друг елемент или е почти хоризонтално или вертикално разположен и т.н.

Автоматичното генериране на ограничения е много полезно средство при проектиране на реални обекти, в които броят на неявните ограничения може да бъде извънредно голям. То естествено крие и рискове за създаване на ограничения там, където това не е нужно.

5.6.3 Използване на линии на конструиране

Друг начин за представяне на геометрични релации между елементите в даден модел е да се използват т.нар. *линии на конструиране* (construction lines). Линиите на конструиране са важен инструмент на неавтоматизираното техническо чертане. Да се върнем към обекта, представен на фиг. 5-24 и точно на неговия външен контур. За да го начертае, един чертожник би създал първо двете окръжности, върху които лежат дъгите, а след това би начертал двете допирателни прави към тях (фиг. 5-53). Както окръжностите, така и правите не са елементи на обекта, който се чертае. Те са помощни елементи, чрез които може да се определят самите елементи, от които е съставен обектът. Тези помощни линии (най-често отбелязвани с пунктир в реалните чертежи) се наричат *линии на конструиране* в системите за геометрично моделиране.



Фиг. 5-53

Линиите на конструиране са най-често само прави и елипси (в частен случай - окръжности), които се представят в модела с коефициентите на аналитичните си уравнения. Те могат да бъдат използвани по два начина:

- *като част от графичния диалог*: В този случай те се създават само за да улеснят конструирането на елементите на обекта, но вече конструирани, тези елементи се съхраняват като напълно независими от линиите на конструиране.
- *като част от метриката на геометричния модел*: В този случай те са важна част от модела. Създадените с тяхна помощ елементи се представят в модела чрез указатели към получените от линиите за конструиране характеристични точки. Тези елементи могат да съществуват само ако съществуват и линиите на конструиране, с помощта на които са получени.

Във втория случай представянето на елементите е различно от разгледаните досега. Един такъв елемент например ще се представи като указател към линия на конструиране и две стойности за параметъра в аналитичното уравнение на тази линия, които съответстват на двете му крайни точки. Ако линията на конструиране, към която сочи указателят от елемента е права, то самият елемент е отсечка; ако пък тази линия е окръжност, елементът е дъга от окръжност. Точките в такъв геометричен модел, които се получават като пресечни точки на линии на конструиране, могат да се съхраняват не с явни координати, а с указатели към линиите на конструиране, при чието пресичане

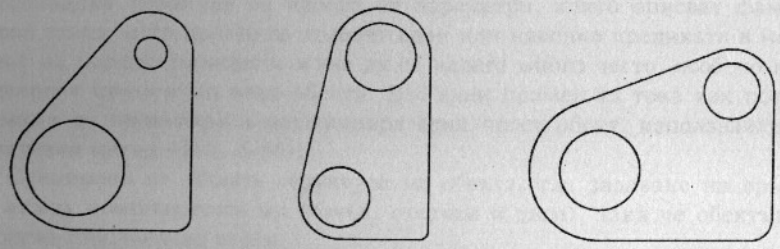
те се получават.

Два елемента могат да бъдат свързани в определени геометрични релации, ако указателите им сочат към една и съща линия на конструиране. От друга страна, самите линии на конструиране могат да са свързани една с друга ако делят общи коефициенти. Това е възможно, ако коефициентите се представят в отделен *списък от скалари*, а от линиите на конструиране има указатели към елементи от този списък. Така например две успоредни прави могат да имат еднакви коефициенти a и b, а да се различават само по третия.

Тези модели позволяват по-голяма гъвкавост при модификация, тъй като метриката на елементите зависи от тази на линиите на конструиране. При подходящо представяне е възможно обектите да се променят при запазване на връзките между тях чрез промяна само на линиите на конструиране, с помощта на които те са създадени. Най-гъвкави в този смисъл са параметричните модели, в които е заложена възможността за модификация с минимална промяна в метриката и топологията им.

5.7 ПАРАМЕТРИЧНИ МОДЕЛИ

Параметричните модели заемат важно място в геометричното моделиране. Те позволяват представяне на обекти, чиито степени на свобода съответстват на зададен от потребителя набор от величини, които имат определен физически смисъл за него. Моделираните по този начин обекти са по-скоро класове или фамилии от обекти, отделните екземпляри на които се получават при фиксиране на стойностите на тези величини. Самите величини, дефинирани от потребителя, са параметри на представяната фамилия от обекти, поради което тя често се нарича *параметричен обект*. Три различни екземпляра на един и същ параметричен обект са показани на фиг. 5-54.



Фиг. 5-54

Параметрични са тези модели, които не представят само един или няколко избрани екземпляра от фамилията, а параметричния обект като едно цяло. По този начин първо се спестява място, което може да бъде много силен аргумент, особено ако броят на използваните екземпляри е сравнително голям. Второ, замяната на един екземпляр с друг от същата фамилия изисква само промяната на стойността на някой от нейните параметри. Очевидно за представянето на параметричен обект са необходими много по-гъвкави структури, отколкото тези, които разгледахме.

Параметричните модели са удобни за представяне на кинематиката на един обект, когато за параметри се дефинират тези характеристики на обекта, които се променят с времето. Отделните екземпляри на параметричния обект тогава са образите му на различните етапи от неговото движение.

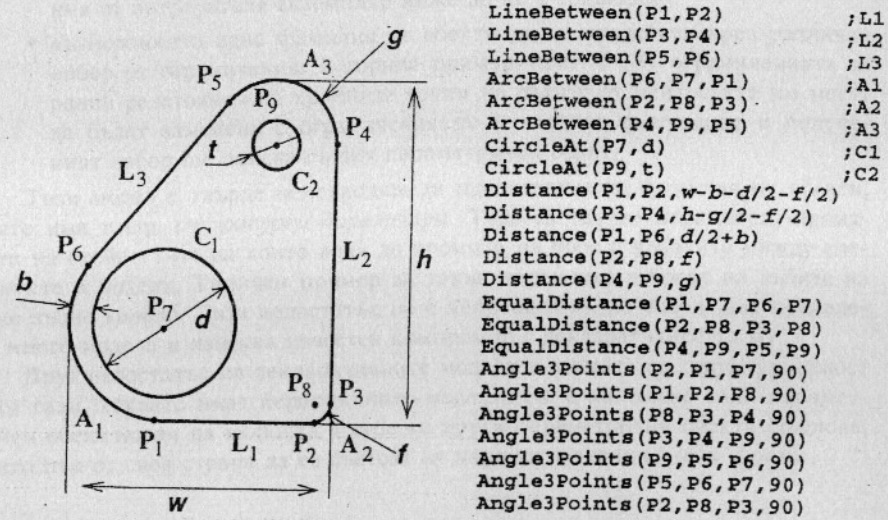
5.7.1 Моделиране чрез ограничения

Апаратът на ограниченията е много удобен за моделиране на фамилия от обекти. От една страна, екземплярите на една фамилия удовлетворяват едни и същи ограничения (както явни, така и неявни). В огромна част от случаите разликата е само в стойностите на допълнителните параметри на едни и същи явни ограничения. От друга страна, параметрите на една фамилия могат лесно да се свържат с тези на явните ограничения.

ДЕКЛАРАТИВНИ МОДЕЛИ. В декларативните модели не се съхраняват координатите на точките, радиусите и ъглите на дъгите, а геометричните зависимости - ограниченията, с които отделните елементи са свързани. На точките и контурните елементи може да се гледа като на базови обекти, а самият модел е съвкупност от предикати, в които те участват. Тези предикати съответстват на предварително подбран набор от ограничения, като изброените по-горе. В показания на фиг. 5-55 пример са използвани само предикатите:

- за инцидентност: `LineBetween, ArcBetween, CircleAt;`
- за разстояние и ъгъл между три точки: `Distance, Angle3Points;` и
- за равни разстояния между точки (неявно): `EqualDistance`.

Параметрите на фамилията `SampleObject` са дадени в получер курсив.



Фиг 5-55

Тези модели се наричат *декларативни* или *ограничителни*, защото редът, в който са зададени предикатите, не е съществен. Важно е само наборът от предикати да е достатъчен за фиксиране на всички степени на свобода на обекта, като при това няма конфликти между отделните ограничения, както и преограничаване на геометрията му. Важно условие за успешното използване на такива модели е наличието на средства за *коректно ограничаване* на модела.

Визуализацията на един екземпляр от зададена по този начин фамилия при фиксиране на стойностите на параметрите ѝ не е толкова просто колкото при моделите, чиято метрика са координатите на точките и елементите. Тъй като за визуализация се използват графични примитиви, всеки екземпляр трябва да се представи с координатите на елементите, от които е съставен. Най-често за получаването на тези координати се използват методи като:

- *алгебрично решаване* на система от нелинейни уравнения относно неизвестните координати на точките - т.нар. метод на вариационната геометрия;
- чрез *аналитични преобразувания* върху полиномиални системи;
- *конструктивно решаване* на задачата от ограничения: използване на методи като конструиране с линийка и пергел, съставяне на граф на зависимостите, декомпозиране и ориентирането му за получаване на процедура за конструиране и др.

Някои от тези методи, като например *вариационната геометрия*, ще разгледаме малко по-подробно.

Декларативните модели имат някои определени предимства, които ги правят предпочитани от различни категории потребители. Те позволяват проектиране на обектите по много естествен за човека начин: задаване на общата структура и задаване на условията, на които нейните компоненти се подчиняват. Тези модели са извънредно гъвкави - в тях е лесно да се извърши репараметризация (промяна на набора от параметри, които описват фамилията). Това става, като просто се заменят един или няколко предиката в модела. Промяна на параметризацията може да се налага много често, особено когато се моделират принципно нови обекти. Ето един пример за това как потребителят може да проектира и модифицира един прост обект, използвайки този декларативен метод (фиг. 5-56):

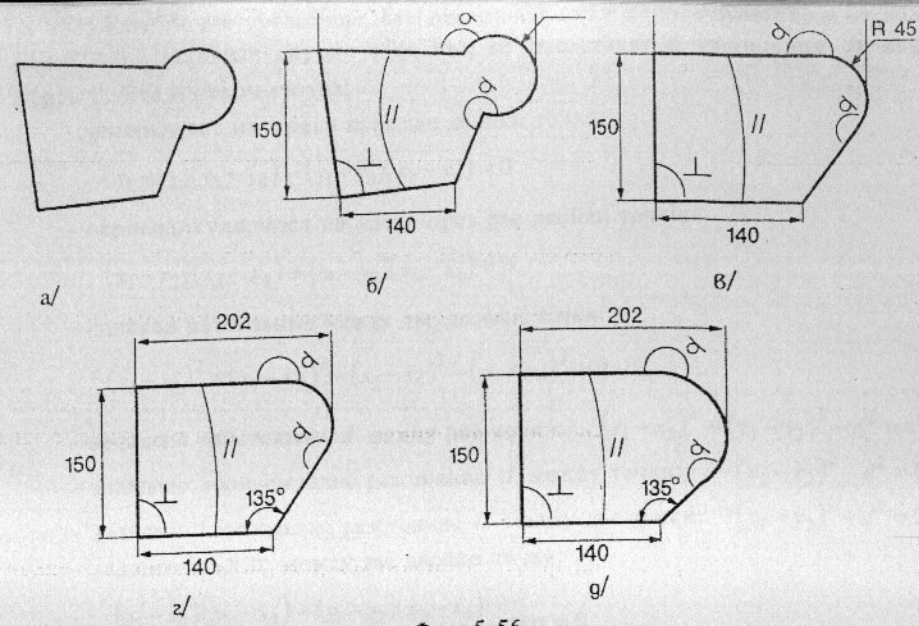
а/Скициране на общата структура на обекта, т.е. задаване на връзките между компонентите му (точки, отсечки и дъги), така че обектът да е само топологично верен;

б/Задаване на ограниченията между компонентите на обекта - фиксирани разстояния, успоредност, перпендикулярност и допирателност;

в/Решаване на задачата за получаване на желаната метрика;

г/Замяна на едно ограничение с друго - нова параметризация на обекта: ограничението за големина на радиуса се заменя с ограничение за ъгъл между отсечки;

д/Ново решаване на ограничителната задача.



Фиг. 5-56

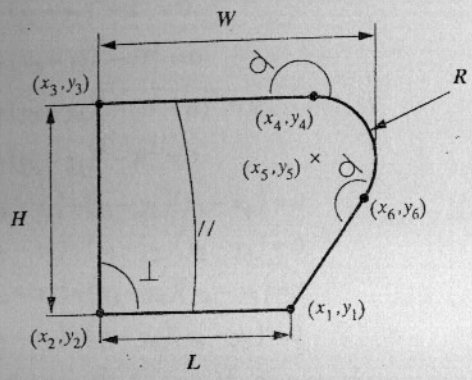
При всичките си предимства декларативните модели имат и някои недостатъци. Първият е, че те са неуникални. Неуникалността се изразява в:

- наличието на няколко екземпляра, които могат да се получат от един и същ набор от стойности на параметрите при коректно ограничени модели. Това е директно следствие от нелинейността на алгебричните уравнения, с които се представят ограниченията. Намирането на желания от потребителя екземпляр може да не е тривиално.
- възможността една фамилия от обекти да се представи чрез различен набор от ограничения. В горния пример (фиг. 5-55) ограниченията за равни разстояния на крайните точки на дъгите от центровете им могат да бъдат заменени с ограничения за фиксирано разстояние и получеият набор ще описва същия параметричен обект.

Този модел е твърде неподходящ за представяне на фамилия от обекти, която има т.нар. *структурни параметри*. Това са такива параметри, промяната на стойностите на които води до промяна на броя и връзките между елементите в модела. Типичен пример за такъв параметър е броят на зъбите на едно зъбно колело. Този недостатък не е непреодолим, но усложнява прекалено много модела и изисква известен компромис с декларативността му.

Друг недостатък на декларативните модели е липсата на структурираност като тази, каквато имат йерархичните модели. Не е възможно един параметричен обект *носач* да включва в себе си други параметрични обекти *болтове*, които пък от своя страна да се състоят от параметризирани *глава* и *резба*.

ВАРИАЦИОННА ГЕОМЕТРИЯ. Един от най-устойчивите методи за решаване на задачата за ограничения е методът на *вариационната геометрия*. При него се използва числено решаване на система от нелинейни уравнения, всяко от които представя съответно ограничение. На самото решаване на системата няма да се спираме подробно. Целта ни тук е по-скоро да покажем как удовлетворяването на наложените ограничения може да се сведе до решаването на чисто математически проблем.



Фиг. 5-57

При вариационната геометрия се приема, че структурата на даден обект (неговата топология) е фиксирана. Както видяхме по-горе, тази топология се съхранява в модела като набор от предикати за принадлежност. Всеки предикат дефинира един елемент от топологията на обекта - отсечка, дъга или окръжност и параметрите, от които този елемент се определя напълно. Да разгледаме обекта, показан на фиг. 5-57. Той се състои от четири отсечки и една дъга. Тези елементи, както и свързването между тях се задават чрез предикатите:

```
LineBetween(x1,y1,x2,y2)
LineBetween(x2,y2,x3,y3)
LineBetween(x3,y3,x4,y4)
ArcBetween(x4,y4,x5,y5,x6,y6)
```

За пълното определяне на местоположението на отсечките и дъгата е достатъчно да знаем координатите на шестте характеристични точки, свързани с горните предикати. Това са и неизвестните в тази задача. Наредената последователност от координатите на характеристичните точки:

$$\mathbf{g} = [x_1, y_1, \ldots x_n, y_n]$$

се нарича *геометричен вектор*. Аналогично, всички параметри на явните ограничения съставят т.нар. *вектор на размерите*:

$$\mathbf{d} = [d_1, d_2, \ldots d_m]$$

Всяко ограничение - както явно така и неявно - може да бъде представено чрез алгебрично уравнение. Ако ограничението е явно, тогава то е от вида $F_i(\mathbf{g},\mathbf{d}) = 0$. Неявните ограничения пък се представят с уравнения от вида $F_i(\mathbf{g}) = 0$. Ето и някои от тях:

- успоредност на прави през две двойки точки:

$$(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4) = 0$$

- перпендикулярност на прави през две двойки точки:

$$(x_1 - x_2)(x_3 - x_4) + (y_1 - y_2)(y_3 - y_4) = 0$$

- еднакво разстояние между две двойки точки:

$$(x_1 - x_2)^2 + (y_1 - y_2)^2 - (x_3 - x_4)^2 - (y_3 - y_4)^2 = 0$$

- зададено разстояние d между две точки: $(x_1 - x_2)^2 + (y_1 - y_2)^2 - d^2 = 0$
- зададено хоризонтално разстояние d между точки: $(x_1 - x_2)^2 - d^2 = 0$
- зададено вертикално разстояние d между две точки: $(y_1 - y_2)^2 - d^2 = 0$
- зададен ъгъл α между две двойки точки:

$$\frac{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}{(x_1 - x_2)(x_3 - x_4) + (y_1 - y_2)(y_3 - y_4)} - \operatorname{tg}\alpha = 0$$

Ограничението за допирателност на права до крайната точка на дъга може да се представи като ограничение за перпендикулярност на правата и централната права през тази крайна точка.

Нека сега да съставим системата от уравнения, която съответства на показания пример. В него има зададени 4 явни и 4 неявни ограничения. Освен тях, системата трябва да се допълни и с ограничението за еднакво разстояние между центъра и крайните точки на дъгата - ограниченията стават общо 9. Броят на неизвестните е 12 (6 двойки координати). Липсващите ограничения са точно трите степени на свобода на една плоска фигура в равнината - нейното положение и ъгълът на завъртането ѝ.

Положението на обекта можем да зададем като фиксираме една от неговите точки - например първата. Ъгълът на завъртане може да фиксираме като поискаме правата, която съдържа първата отсечка, да е хоризонтална. Ще получим системата, показана по-долу. Като заместим параметрите с желаните стойности, ще получим нелинейна система, за чието решаване е най-добре да се използва итерационен метод. За начални стойности на неизвестните могат да се вземат текущите стойности, които имат координатите на характеристичните точки на грубо скицирания обект (фиг. 5-56б).

Удобен итерационен метод е този на Нютон-Рафсон (Newton-Raphson), при който последователно се решава линейната система, чиято матрица е якобиянът на първоначалната и чието решение е вектор от корекции, който се добавя към геометричния вектор на всяка итерация. Самият якобиян е силно разредена матрица, тъй като всяко уравнение включва най-много осем не-

известни. Разработени са достатъчно удобни и бързи методи за решаването
даже на сравнително големи системи уравнения с такива матрици. Ще посо-
чим обаче, че намирането на желаното от потребителя решение може да се
окаже твърде трудно. Точката на сходимост на итерационния метод зависи
извънредно много от началните стойности на неизвестните и при големи про-
мени в стойностите на параметрите старите координати на характеристичните
точки могат да водят до нежелано решение.

$$
\begin{cases}
(x_1-x_2)^2+(y_1-y_2)^2-L^2=0\\
(x_2-x_3)^2+(y_2-y_3)^2-H^2=0\\
(x_3-x_4)^2+(y_3-y_4)^2-(W-R)^2=0\\
(x_4-x_5)^2+(y_4-y_5)^2-R^2=0\\
(x_1-x_2)(y_3-y_4)-(y_1-y_2)(x_3-x_4)=0\\
(x_1-x_2)(x_2-x_3)+(y_1-y_2)(y_2-y_3)=0\\
(x_3-x_4)(x_4-x_5)+(y_3-y_4)(y_4-y_5)=0\\
(x_1-x_6)(x_6-x_5)+(y_1-y_6)(y_6-y_5)=0\\
(x_4-x_5)^2+(y_4-y_5)^2-(x_6-x_5)^2-(y_6-y_5)^2=0\\
x_1=0\\
y_1=0\\
y_1-y_2=0
\end{cases}
$$

Използването на числен метод има и много предимства. То позволява например да се използват ограничения, които имат отношение към геометрията, но нямат пряк израз в оразмерявания. На практика могат да се използват всички ограничения, които могат да бъдат изразени чрез алгебрични уравнения например: "площта на фигурата да има определена стойност".

5.7.2 Процедурни параметрични модели

Един прост начин за представяне на фамилия от обекти е като се напише процедурата, с която се създава всеки отделен екземпляр от нея. Операторите, от които се състои тази процедура, могат да бъдат: дефиниране на геометрията на контурни елементи, обръщение към процедури, създаващи екземпляри на вложени параметрични обекти и др.

Да вземем за пример параметричния обект "правилен N-ъгълник". Вместо всички отсечки, които го съставят, моделът може да бъде описание как да се получат тези отсечки при зададени стойности на няколко параметъра: центърът на многоъгълника, броят и дължината на страните му. В този смисъл тези модели са *конструктивни*. Те приличат много и на йерархичните модели като основната разлика е, че процедурите имат параметри, които влияят на структурата на обекта - на всички компоненти на метриката и топологията на екземплярите му.

Процедурите могат да бъдат написани на езика, на който е написана и самата моделираща система при използване на функциите за моделиране на графичната система или само нейните графични примитиви. В този случай разликата от йерархичните модели, разгледани преди, е наистина минимална. Процедурата на "правилния N-ъгълник" може да създава само правилен N-ъгълник и нищо друго, т.е. потребителят може да работи само с параметричните обекти, които са били подготвени и включени в приложната програма от нейните разработчици. Такъв модел може да не е пълен, а със сигурност не е и разширяем.

Друго решение е в моделиращата система (в приложната програма) да се разработи интерпретатор от специализиран процедурен език, на който потребителят да може сам да описва параметричните обекти, с които работи. В този език трябва да има подходящ набор от оператори за дефиниране и манипулиране на геометрични обекти, за структуриране и др. По този начин параметричните модели стават достатъчно общи, за да покрият произволен клас от обекти дори такива, които имат структурни параметри. Приложните програми могат да предоставят средства и за графичното дефиниране на такива процедурни описания. Най-често се използват методи като графично *програмиране чрез примери* (Programming-by-Example), при които интерактивната потребителска работа по създаването на един екземпляр от фамилията (т.нар. *пример*) се обобщава и автоматично превежда в процедура за конструиране на произволен екземпляр от нея.

Моделиращите системи, използващи процедурни параметрични модели, най-често поддържат и контурен модел, в който се съхраняват екземплярите, а при *програмиране чрез примери* и екземпляра-пример, който служи за обобщаване на конструирането на фамилията.

5.7.3 Моделиране чрез признаци

Един специален вид процедурни модели са тези, основани на *признаци* (feature-based models). В тях един обект се представя като съвкупност от определени характеристики, с които потребителят и моделиращата система работят и които се наричат с общото име *признаци* (features). Този модел е извънредно удобен за тримерните системи, където признаците могат да се дефинират така, че да са близки до термините, с които инженерите работят: отвори, джобове, фаски, планки и др. Ние тук ще разгледаме основния принцип, на който тези модели се основават.

Идеята е сравнително проста. Да разгледаме обекта от фиг. 5-55. Дъгата с радиус f не е много съществен елемент от този обект. Тя се е получила в резултат на заобляне на един връх. Това заобляне не е важна част от обекта. То е просто един негов признак. Обектът не би се променил коренно, ако този признак липсва. Макар декларативният модел да е сравнително гъвкав, читателят може да си представи какво трябва да се промени в него, ако потребителят пожелае този връх да не е заоблен. За разлика от останалите видове модели, тези, основани на признаци позволяват както лесното добавяне, така и лесното премахване на един признак.

Моделът от признаци е конструктивен, процедурен модел, основните компоненти на който не са геометрични примитиви, а признаци. Всеки признак се определя от набор от параметри и съответства на някаква операция, която се изпълнява от моделиращата система с тези параметри. Това може да бъде операция за създаване на определен вид елемент (отвор, джоб), булева операция, операция за заобляне или скосяване на връх и др. По този начин значително се опростява моделът за сметка на това, че интерпретацията на всеки отделен признак се разработва като моделна операция, т.е. усложнява се самата моделираща система. Например ако в модела се съхранява признакът *заобляне*, то при визуализация системата трябва да изпълни операцията заобляне за получаване на дъгата, която в крайна сметка е графичният примитив, който ще се визуализира. Отделните признаци може да са свързани един с друг чрез ограничения между техните параметри и тогава моделиращата система трябва да може да решава и задачи за ограничения.

Моделите от признаци позволяват представянето на параметрични обекти със структурни параметри, като се въвеждат такива признаци като:

- *матрица*: разполагане на признаци, подредени в правоъгълна матрица с параметри: броят по всяка от осите и разстоянията между тях;
- *кръгов шаблон*: разполагане на признаци в кръг при зададен център, брой на признаците и ъгълът между тях.

За съжаление, моделите от признаци могат също като конструктивните дървета да бъдат неуникални, т.е. един и същ обект да се опише като съставен от различни признаци. Достатъчно е да се върнем за малко към обекта от фиг. 5-43, за да видим, че това ще бъде винаги така, когато се използват допълващи се (комплиментарни) признаци.

Задачи

5.1 Използвайки възможностите за структуриране, представени в края на трета глава, дефинирайте обекта "кабинет 101" от фиг. 5-9 така, че да има възможност за промяна на местоположението на бюрата в него.

5.2 Представете трансформацията на изгледа, разгледана в трета глава като композиция от геометрични трансформации.

5.3 Докажете, че композиция от две ротации с един център е ротация на ъгъл, който е сумата от ъглите на двете ротации (свойство "адитивност").

5.4 Намерете трансформацията, която се определя от две двойки точки в равнината. Това е преобразуване, при което първата двойка от точки се трансформира във втората (фиг. 4-20).

5.5 Напишете програма, която изпълнява ротация на зададен ъгъл спрямо зададен център на дъга, която е представена чрез центъра, радиуса, началния и крайния си ъгъл. Напишете друга програма, която изпълнява осева симетрия спрямо зададена ос върху същата дъга.

5.6 Намерете израз за радиуса и центъра на дъгата, представена с двете си крайни точки и ориентираната височина на сегмента, който тя определя (фиг. 5-21).

5.7 Напишете програма, която добавя точка към списък от точки, чиито координати са числа с плаваща запетая, като преди добавянето проверявате дали такава точка вече не фигурира в списъка (използвайте някакъв допуск при сравненията).

5.8 Предложете начин за моделиране на текстови шрифтове, буквите в които могат да се изобразяват и слято (свързани една с друга).

5.9 Напишете програма, която проверява дали един многоъгълник е самопресичащ се или не.

5.10 Напишете програма за щриховане на изпъкнал многоъгълник, като е зададен наклонът на щрих-линиите и разстоянието между тях.

5.11 Напишете програма, която намира площта на област, заградена от един затворен несамопресичащ се контур, който се състои от отсечки и дъги от окръжности, по начина показан на фиг. 5-25.

5.12 Напишете програма, която проверява каква е подредбата на точките на един триъгълник - в посока на часовниковата стрелка или обратна на нея. Използвайте дефиницията за векторно произведение.

5.13 Напишете програма, която проверява каква е подредбата на точките на един произволен прост многоъгълник.

5.14 Съставете алгоритъм, който открива дали един многоъгълник е изпъкнал или не.

5.15 Съставете програма за разбиване на изпъкнал многоъгълник на триъгълници, без да добавяте нови върхове. Запълнете структурата от данни, показана на фиг. 5-49. Какъв алгоритъм ще използвате за посочване един от тези триъгълници?

5.16 Решете горната задача, но като евентуално добавите нова точка (или точки). Коя точка е най-удобна за добавяне?

5.17 Предложете структури от данни за съхраняването на обекти, зададени като обединение на допълващи се области (фиг. 5-28)?

5.18 Напишете функцията `LoadPolygon` от алгоритъма на Уейлър.

5.19 Допишете програмата `IntersectPolygons` от алгоритъма на Уейлър, като разгледате случаите на допиране и съвпадане на ръбове.

5.20 Съставете алгоритъм за намиране на съседната на дадена клетка на едно квадратично дърво спрямо една от четирите основни посоки.

5.21 Напишете програма за изпълнение на булеви операции с квадратични дървета.

5.22 Напишете програма за изпълнение на булеви операции над "квадратични дървета от прости многоъгълници" с листа като тези, показани на фиг. 5-46.

5.23 Напишете програма, която проверява дали един обект, зададен със своето BSP-дърво е нулев или не (дали има или не площ).

5.24 Напишете програма за съставяне на BSP-дърво на неизпъкнал многоъгълник.

5.25 Напишете програма, която определя положението на точка спрямо произволна област, зададена с BSP-дърво.

5.26 Предложете структури от данни за съхраняването на триангулачна мрежа на обект с криволинейни граници (дъги и окръжности), които се представят без апроксимация.

5.27 За обект, представен в структурата от данни от фиг. 5-49, намерете контура на многоъгълника, образуван от всички триъгълници, които имат общ връх в дадена точка.

5.28 Използвайки БГП, напишете програма, която начертава линейното оразмеряване между две точки и използва като вход някой от наборите параметри, показани на фиг. 5-51.

5.29 Предложете минимален набор от геометрични релации, с които могат да се опишат всички геометрични зависимости между отсечки и дъги от окръжности в равнината. Запишете техните алгебрични уравнения.

5.30 Съставете процедурен параметричен модел (програма на C, използваща БГП) за обекта от фиг. 5-55.

5.31 Как бихте представили параметричния обект "правоъгълник със заоблени върхове" по всеки от начините, разгледани в тази глава?

5.32 Сравнете разгледаните в тази глава модели, прилагайки следните критерии: точност на представяне, мощност на представяне, разширяемост, еднозначност, възможност за верификация, компактност, ефективност при визуализация и ефективност при обработка.

5.33 Анализирайте предимствата на обектноориентираните среди за геометрично моделиране, както и техните недостатъци. Как бихте представили един параметричен обект в такава среда. А релациите между отделните елементи?

5.34 Предложете набор от признаци и начин за представяне чрез тях на обекта от фиг. 5-42.

Криви в равнината

В много от приложенията на компютърната графика се налага да се изобразяват гладки равнинни криви. Има два основни случая, в които е необходимо да се използват криви:

- когато се моделират криволинейните граници на съществуващи обекти. За тези обекти най-често са налице или резултати от измервания, или някакво математическо описание на формата им.
- когато се моделират нови обекти, чиито контури имат т.нар. *свободна форма* (freeform curves). Тогава точната форма на една крива не е предварително известна. Проектантът има по-скоро идея как тя трябва да изглежда, а окончателният ѝ вид зависи от резултата на различни анализи върху нея, прилагащи както строго определени, така и чисто естетически и евристични критерии.

Очевидно е, че изискванията към математическото описание на една крива във всеки от тези два случая могат да бъдат много различни. И при двата обаче е възможно кривите да бъдат приближени от начупени линии, отделните страни на които да имат достатъчно малка дължина. От гледна точка на съхраняването им, това би било твърде неикономично, особено ако се изисква висока точност, т.е. приближаване чрез начупени линии с извънредно голям брой върхове. Такова представяне би затруднило много и интерактивното задаване и модифициране на кривата. Ето защо се налага да търсим както по-компактни, така и по-удобни за интерактивно моделиране форми за представянето на гладки криви.

Съществуват много математически методи за задаване на криви в равнината. Особено предимство сред тях има параметричното представяне $\mathbf{r} = \mathbf{r}(u)$, защото то дава възможност лесно да се получи последователност от точки върху кривата. Както споменахме, начупената линия, свързваща такива точки, може да даде достатъчно добра визуална апроксимация на истинската крива. Чрез вариране на параметъра u (от 0 до 1 със стъпка s) от векторното параметрично уравнение:

$$\mathbf{r} = \mathbf{r}(u) \quad \text{т.е.} \quad \begin{cases} x = x(u) \\ y = y(u) \end{cases} \qquad u = 0, s, 2s, \ldots, 1$$

начупената през получената поредица от точки ще се приближава до кривата

толкова по-добре, колкото по-малка е стъпката s. Възможно е, разбира се, в качеството на параметър да се използва някоя от координатите, така че представянето да е от вида $y = y(x)$ или $x = x(y)$. Това по същество е задаване чрез *явно уравнение*, което създава някои неудобства, най-важни от които са:

- Невъзможно е да се получат няколко стойности на функцията за една и съща стойност на параметъра, което налага окръжности и елипси да се описват чрез няколко сегмента;
- Трудно е представянето на криви с вертикални производни;
- Параметризираните по този начин криви не са инвариантни по отношение на ротация.

Неявно зададените криви $f(x,y)=0$ също не са приемлива алтернатива, особено когато неявното уравнение има няколко решения при фиксирана една координата. Например за да се представи горната половина от единичната окръжност е необходимо към неявното уравнение $x^2+y^2-1=0$ да се прибави и неравенството $y \geq 0$. Неявното уравнение може да се използва, за да се определи дали дадена точка принадлежи на кривата (както и като мярка за отдалечеността на точката от кривата), но не е пригодно за намиране на точки върху нея, освен ако не може да се сведе до явно уравнение.

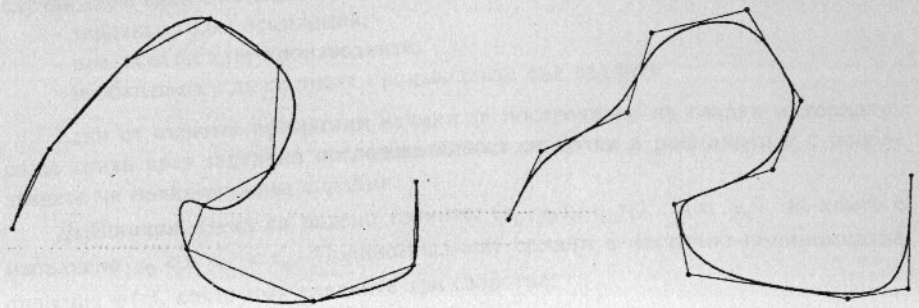
В практиката, освен за визуализация, математическото описание на кривите се използва и за изчисляването на различни техни характеристики: нормали, тангенти, кривини и др. Параметричните тангентни и нормални вектори са по-удобни за използване от геометричните наклони, които, както отбелязахме, могат да бъдат безкрайни. Тези изчисления изискват диференциране на функцията, задаваща кривата. Използването на полиноми в качеството на параметрична функция значително опростява намирането на производни и на условия за тяхната непрекъснатост.

Полиномиалните функции от висока степен обаче имат значителен брой коефициенти, чиито геометричен смисъл е трудно разбираем. Друго неудобство на използването на полиноми от висока степен е тяхната склонност към осцилация. Практиката показва, че полиномите от трета степен са най-подходящи поради наличието едновременно на достатъчно степени на свобода за контролиране на формата на кривата, а също така и заради възможността да се придаде конкретен геометричен смисъл на полиномиалните коефициенти. Поради тези именно причини тук ще разглеждаме най-вече полиномиални параметрични криви.

В зависимост от това за какви цели се генерират, съществуват два типа криви - *интерполиращи* и *апроксимиращи*.

Първият тип е характерен за задачите за визуализация: по предварително зададено наредено множество от точки да се намери гладка крива, която минава през всички точки от множеството. Това се прави например при визуализация на експериментални данни.

Апроксимиращата крива, от друга страна, минава най-близко до зададени точки, без непременно тези точки да лежат върху нея. С други думи, зададените точки управляват формата на кривата, която се стреми да се приближи до начупената, която наредената множество от точки определя, като същевременно удовлетворява и различни допълнителни условия: за непрекъснатост, гладкост, определена кривина в зададени точки и т.н. Би следвало да уточним, че понятието апроксимация не е обичайното в математиката: приближаване на функция, множество от точки или друга система по оптимален път съобразно някакъв критерий.



интерполираща крива апроксимираща крива

Фиг. 6-1

Задачи за построяване на апроксимиращи криви се решават в моделирането, където един проектант синтезира нови обекти, за които отнапред са известни само определени изисквания. За типичен пример може да служи проектирането в области като автомобилостроене, самолетостроене, корабостроене и др. В други случаи е възможно тези изисквания да са чисто естетически - графичен дизайн, проектиране на текстови шрифтове, проектиране на обувки, дрехи и др.

В тази глава ще разгледаме както начините за интерполация на последователност от точки, така и методите за построяването на апроксимиращи криви, което е важна задача в геометричното моделиране.

6.1 ИНТЕРПОЛИРАЩИ КРИВИ

6.1.1 Интерполация с конични сечения

Нека преди да преминем към общата задача за намиране на интерполираща крива през наредено множество от точки, да се спрем накратко на един прост метод за намиране на коничното сечение през зададени точки. Той е предложен още през 1944 год. от Лайминг (Liming) за нуждите на проектирането на напречните сечения на тялото на самолет.

Общото неявно уравнение на едно конично сечение е:

$$ax^2+2bxy+cy^2+2dx+2ey+f=0$$

Нека да предположим, че са ни дадени четири точки **A**, **B**, **C** и **D**, а правите, върху които лежат страните на образувания от тях четириъгълник да означим с l_1, l_2, l_3 и l_4. Уравненията на тези прави са:

$$l_i(x,y)=a_ix+b_iy+c=0 \qquad i=1,2,3,4$$

Тогава уравнението

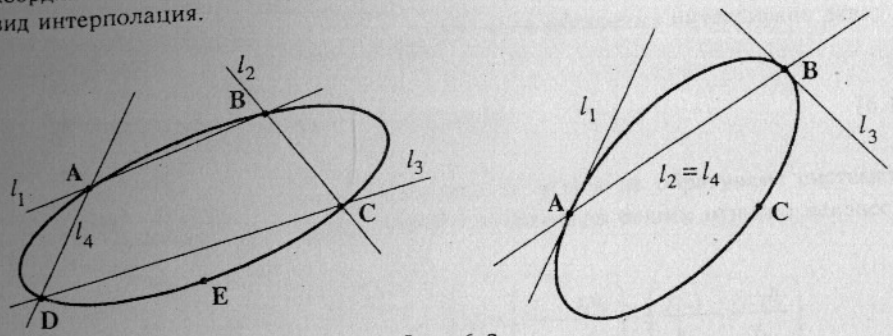
$$(1-\lambda)l_1 \cdot l_3 + \lambda l_2 \cdot l_4 = 0$$

представя фамилия от конични сечения, всяко от които минава през четирите точки. Ако знаем още една допълнителна точка **E**, можем да получим стойността на λ, за която крива от тази фамилия интерполира всичките пет точки **A**, **B**, **C**, **D** и **E**. Пет точки (или по-общо) пет условия са достатъчни за определянето на коефициентите на крива от втора степен.

Като се използва този метод, може да се намери и такова конично сечение, което да има две зададени допирателни в две точки **A** и **B**, и което минава през трета точка **C**. Това можем да постигнем, като приемем, че две от горните четири прави съвпадат $(l_2 \equiv l_4)$. Уравнението на фамилията тогава ще има вида:

$$(1-\lambda)l_1 \cdot l_3 + \lambda l_2^2 = 0$$

Стойността на λ можем да определим, като заместим в това уравнение с координатите на трета точка **C**, която често се нарича и *опорна точка* в този вид интерполация.



Фиг. 6-2

Този метод има и много други приложения, едно от които е конструиране на частично-квадратична крива, сегментите на която са конични сечения и се съединяват гладко. Това се постига, като се изисква допирателните към отделните сегменти в точките на съединяване да съвпадат.

6.1.2 Интерполация с полиномиални сплайни

Класическата интерполация при зададени $n+1$ точки в равнината се осъществява с използването на полином на Лагранж (Lagrange) от степен n, който минава през тези точки. Както бе отбелязано по-горе, тенденцията на полинома към осцилация расте с увеличаване на степента му, т.е. с увеличаване на броя на зададените точки. За да се избегне това, често се използва частично-полиномиална функция, т.е. функция, която е дефинирана върху няколко съседни интервала и е полином (но може различен) във всеки отделен интервал. Това се прави като се разделят точките на групи и за всяка група се съставя полином на Лагранж от по-ниска степен (толкова, колкото са точките в

групата минус 1). При този подход е възможна прекъснатост на производните, което почти винаги е нежелателно. За да се реши този проблем, се прибягва до използване на полиноми на Ермит (Hermite), за съставянето на които обаче е необходимо да се зададат стойности на производните във възлите.

За да обобщим ще кажем, че класическата интерполация има във всеки случай поне един от следните недостатъци:

- тенденция към осцилация;
- прекъснатост на производните;
- необходимост да се знаят производните във възлите.

Един от първите прилагани начини за построяване на гладка интерполираща крива през зададена последователност от точки в равнината е с използването на полиномиални сплайни.

Дефиниция. Нека са дадени точките: $(x_0,y_0),(x_1,y_1),\ldots,(x_n,y_n)$, за които е изпълнено $x_0 < x_1 < \cdots < x_n$. Полиномиалният сплайн е частично-полиномиална функция $\psi(x)$, която има следните три свойства:

- $\psi(x_i) = y_i$, $i = 1,2,\ldots n$;
- във всеки подинтервал $[x_{i-1},x_i]$, $i = 1,2,\ldots n$, $\psi(x)$ е полином от степен m;
- всички производни на $\psi(x)$ до $m-1$ ред са непрекъснати във възлите x_i.

Такава функция се нарича полиномиален сплайн от степен m. Ние ще разгледаме подробно кубичните сплайни. От казаното по-горе следва, че ако $\psi(x)$ е кубичен сплайн, то $\psi(x)$, $\psi'(x)$ и $\psi''(x)$ са непрекъснати в x_i. Кубични сплайни могат да се построят по няколко различни начина:

ПЪРВИ НАЧИН:

Нека са известни $\psi'(x_0) = y_0'$, $\psi''(x_0) = y_0''$. Ще покажем, че това е достатъчно за да се построи целият сплайн при положение, че знаем точките, през които той трябва да минава. Тъй като знаем още, че $\psi(x_0) = y_0$ и $\psi(x_1) = y_1$, то тези четири условия са достатъчни за да определим $\psi(x)$ като кубичен полином в интервала $[x_0,x_1]$.

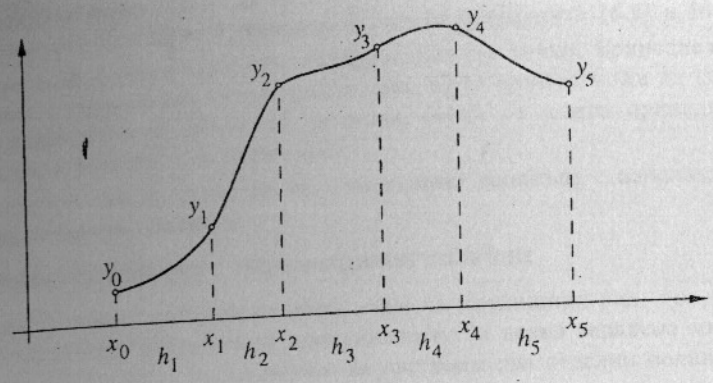
Знаейки $\psi(x)$ в интервала $[x_0,x_1]$, ще намерим стойностите на първата и втората производни на този полином в двата края $\psi'(x_1) = y_1'$, $\psi''(x_1) = y_1''$. Използвайки тези стойности, по аналогичен начин намираме $\psi(x)$ в интервала $[x_1,x_2]$. Можем да повтаряме този процес до последния интервал $[x_{n-1},x_n]$.

Този метод обаче е неудобен по две причини: Първо, възможно е натрупването на значителна грешка при последователното изчисляване на производните. Второ, необходимо е да се знае втората производна в началото на кривата. Следващият метод преодолява тези недостатъци.

ВТОРИ НАЧИН:

Ще се опитаме да използваме непрекъснатостта на вторите производни, за да получим за тях система уравнения, без да е необходимо да свеждаме задачата до явно намиране на изрази за тях. Целта е да се елиминира необходимостта да се знае втората производна в началото и да се намерят други допълнителни условия, които биха били по-лесно измерими.

Нека разгледаме $\psi(x)$ в интервала $[x_{i-1},x_i]$ и нека означим $h_i = x_i - x_{i-1}$.



Фиг. 6-3

Тъй като $\psi(x)$ е кубичен полином в този интервал, $\psi''(x)$ е линейна функция. Тогава ако положим $\psi''(x_{i-1}) = s_{i-1}$, $\psi''(x_i) = s_i$, формата на $\psi(x)$ ще е следната:

$$\psi''(x) = \frac{s_{i-1}(x_i - x) + s_i(x - x_{i-1})}{h_i}, \text{ което след двукратно интегриране дава:}$$

$$\psi(x) = \frac{s_{i-1}(x_i - x)^3 + s_i(x - x_{i-1})^3}{6h_i} + C_1 x + C_2 \qquad [6.1]$$

Нека сега да използваме полученото за $\psi(x)$ и да образуваме системата от двете уравнения $\psi(x_{i-1}) = y_{i-1}$ и $\psi(x_i) = y_i$, която да решим относно неизвестните C_1 и C_2.

$$\left\| \begin{aligned} y_{i-1} &= \frac{s_{i-1}}{6}h_i^2 + C_1 x_{i-1} + C_2 \\ y_i &= \frac{s_i}{6}h_i^2 + C_1 x_i + C_2 \end{aligned} \right. \Rightarrow \left\| \begin{aligned} C_1 &= \left(\frac{y_i}{h_i} - \frac{s_i h_i}{6}\right) - \left(\frac{y_{i-1}}{h_i} - \frac{s_{i-1}h_i}{6}\right) \\ C_2 &= \left(\frac{y_{i-1}}{h_i} - \frac{s_{i-1}h_i}{6}\right)x_i - \left(\frac{y_i}{h_i} - \frac{s_i h_i}{6}\right)x_{i-1} \end{aligned} \right.$$

Като заместим с получените изрази за C_1 и C_2 в уравнение [6.1], след подходящо прегрупиране ще изразим $\psi(x)$ чрез неизвестните s_i по следния начин:

$$\psi(x) = \frac{s_{i-1}}{6h_i}(x_i - x)^3 + \frac{s_i}{6h_i}(x - x_{i-1})^3 + \left(\frac{y_{i-1}}{h_i} - \frac{s_{i-1}h_i}{6}\right)(x_i - x) + \left(\frac{y_i}{h_i} - \frac{s_i h_i}{6}\right)(x - x_{i-1}) \qquad [6.2]$$

Този полином дефинира интерполиращата функция в интервала $[x_{i-1}, x_i]$. За да намерим стойността на първата производна в десния край на интервала нека да диференцираме [6.2]:

$$\psi'(x) = -\frac{s_{i-1}}{2h_i}(x_i - x)^2 + \frac{s_i}{2h_i}(x - x_{i-1})^2 + \frac{y_i}{h_i} - \frac{y_{i-1}}{h_i} + \frac{s_{i-1}h_i}{6} - \frac{s_i h_i}{6}, \qquad x \in [x_{i-1}, x_i]$$

Като заместим с x_i и използваме, че $h_i = x_i - x_{i-1}$ ще получим:

$$\psi'(x_i) = \frac{s_i h_i}{2} + \frac{y_i - y_{i-1}}{h_i} + \frac{s_{i-1}h_i}{6} - \frac{s_i h_i}{6} \quad \Rightarrow \quad \psi'(x_i) = \frac{y_i - y_{i-1}}{h_i} + \frac{s_{i-1}h_i}{6} + \frac{s_i h_i}{3}$$

Сега да разгледаме интерполиращата функция в интервала $[x_i, x_{i+1}]$. Това става като просто заместим с $i+1$ в [6.2] на местата, където се среща i. Нека диференцираме и заместим отново с x_i - левия край на интервала $[x_{i+1}, x_i]$. Както и преди ще използваме, че $h_{i+1} = x_{i+1} - x_i$:

$$\psi'(x) = -\frac{s_i}{2h_{i+1}}(x_{i+1} - x)^2 + \frac{s_{i+1}}{2h_{i+1}}(x - x_i)^2 + \frac{y_{i+1}}{h_{i+1}} - \frac{y_i}{h_{i+1}} + \frac{s_i h_{i+1}}{6} - \frac{s_{i+1}h_{i+1}}{6}$$

$$\psi'(x_i) = -\frac{s_i h_{i+1}}{2} + \frac{y_{i+1} - y_i}{h_{i+1}} + \frac{s_i h_{i+1}}{6} - \frac{s_{i+1}h_{i+1}}{6} = \frac{y_{i+1} - y_i}{h_{i+1}} - \frac{s_{i+1}h_{i+1}}{6} - \frac{s_i h_{i+1}}{3}$$

Като приравним изразите за $\psi'(x_i)$ за двата съседни интервала и прегрупираме неизвестните s_i ще получим $n-1$ уравнения от вида:

$$s_{i-1}h_i + 2s_i(h_{i+1} + h_i) + s_{i+1}h_{i+1} = 6\left(\frac{y_{i+1} - y_i}{h_{i+1}} - \frac{y_i - y_{i-1}}{h_i}\right), \quad i = 1, \ldots, n-1 \qquad [6.3]$$

Това е тридиагонална система относно неизвестните s_i, т.е. необходими са още две уравнения, за да се определи тя напълно. Следователно, при подходящо избрани две допълнителни условия, след решаване на горната система можем по изразите [6.2] да намерим функцията във всеки интервал. Най-често се задават следните условия:

- Сплайн със свободни краища: $s_0 = s_n = 0$, наричан още естествен сплайн;
- Сплайн с квадратични крайни участъци: $s_0 = s_1$ и $s_{n-1} = s_n$;
- Сплайн със зададени допирателни в краищата: $\psi'(x_0) = y_0'$, $\psi'(x_n) = y_n'$;
- Комбинация от някои от горните.

Сплайните са удобно средство за конструиране на криволинейни форми. Както видяхме, техните преимущества са основно:

- непрекъснатостта на производните, което в случая на непрекъсната втора производна води до непрекъснатост и на кривината им;
- неголямата информация за производните, необходима за тяхното построяване.

Показаната формулировка носи всички недостатъци на явното представяне, което освен очевидната невъзможност да описва криви, които минават през различни точки с една и съща x-координата, имат и други ограничения, най-важните от които са:

- не е възможна локална модификация: при промяна на една от интерполираните точки целият сплайн трябва да се преизчисли;

- неприложимост при наличие на вертикални производни - това би предизвикало делене на 0, както се вижда от формулите [6.2] и [6.3];
- възможна е осцилация при интерполация на точки, принадлежащи на крива с прекъснати втори производни. Като пример може да се посочи последователност от точки, първата група от които принадлежи на права, а втората - на окръжност.

Последните два недостатъка се преодоляват донякъде с използването на т.нар. *параметрични сплайни*.

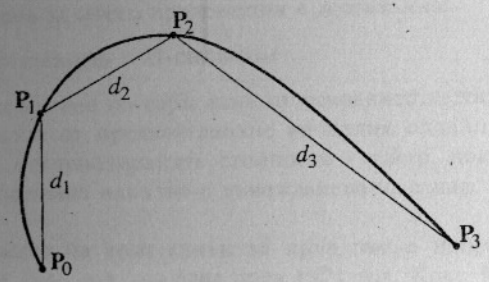
6.1.3 Интерполация с параметрични сплайни

Параметричните сплайни са обобщение на предишния вид, при които се използва параметър различен от координатите. За всяко зададено множество от точки (x_0,y_0), (x_1,y_1),..., (x_n,y_n) можем да построим два отделни полиномиални сплайна $x = x(u)$ и $y = y(u)$, които да интерполират множествата от точки съответно:

(u_0,x_0), (u_1,x_1),..., (u_n,x_n) и

(u_0,y_0), (u_1,y_1),..., (u_n,y_n).

Последователността от стойности на параметъра u_0, u_1,..., u_n ще наричаме *възли*. Изборът на тези възли е важен, тъй като те определят в голяма степен параметризацията на кривата.



Фиг. 6-4

В този случай най-естествено би било за параметър u да се избере дължината на дъгата на кривата, но тази дължина не може да се изчисли преди да се конструира самата крива. Оказва се, че ако в качеството на параметър се използва натрупаната дължина на хордите (дължините на отсечките, свързващи точките), се постига почти същият резултат (фиг. 6-4).

$$u_0 = 0, \qquad u_{i+1} = u_i + d_{i+1}, \qquad d_i = \sqrt{(x_i - x_{i-1})^2 + (y_i + y_{i-1})^2}, \qquad i = 1,2,\ldots,n-1$$

Друг подход при избор на параметъра u е да се използват целочислени стойности във възлите $u_i = i$, а именно:

$(0,x_0)$, $(1,x_1)$,..., (n,x_n) и $(0,y_0)$, $(1,y_1)$,..., (n,y_n).

Това обаче може да доведе до усложнения при положение, че възлите са разположени много неравномерно в равнината. В такива случаи е възможна осцилация и дори самопресичане на кривата.

Очевидно е, че при параметричните сплайни не възниква проблем с вертикалните производни. Още повече, както и при всяко параметрично представяне от типа $\mathbf{r} = \mathbf{r}(u)$, резултатите могат директно да се приложат за криви в пространството.

6.1.4 Интерполация с криви на Оверхаузер

Използването на параметрични сплайни не изключва напълно възможността за осцилация. Това е довело до предлагането на най-различни формулировки, целта на които е била преодоляването на този недостатък.

Вместо тях обаче много по-често се прилага един прост метод, който има своите предимства. Това е интерполацията с параметрични криви, предложени от Оверхаузер (Overhauser) през 1968 год. Множеството от точки, което се интерполира (и за които се предполага, че са сравнително равномерно отдалечени една от друга) се разделя на групи от по три съседни точки, като всяка точка участва точно в три различни групи. За всяка от тези групи, които се припокриват, се съставя параметрична квадратична крива, която ги интерполира.

Получените квадратични криви-сегменти се комбинират в една обща крива чрез линейна функция спрямо параметрите на сегментите. Тази линейна функция се избира такава, че да е единица в средата на всеки сегмент и 0 в краищата му. Да видим по-подробно как се постига това.

Да разгледаме две такива застъпващи се групи от точки **KLM** и **LMN** (фиг. 6-5). Квадратичните сегменти, които интерполират тези две групи са:

$$\mathbf{q}(s): \quad \begin{aligned} x(s) &= a_{11} + b_{11}s + c_{11}s^2 \\ y(s) &= a_{12} + b_{12}s + c_{12}s^2 \end{aligned}, \qquad 0 \le s \le 1 \quad \text{в } \mathbf{KLM}$$

като $s = 0$ в т. **K**, $s = 0.5$ в т. **L** и $s = 1$ в т. **M** и

$$\mathbf{r}(t): \quad \begin{aligned} x(t) &= a_{21} + b_{21}t + c_{21}t^2 \\ y(t) &= a_{22} + b_{22}t + c_{22}t^2 \end{aligned}, \qquad 0 \le t \le 1 \quad \text{в } \mathbf{LMN}$$

като $t = 0$ в т. **L**, $t = 0.5$ в т. **M** и $t = 1$ в т. **N**.

Резултантната крива от смесването на горните два сегмента в интервала между точките **L** и **M** ще се задава параметрично така:

$$x = \alpha\left(a_{11} + b_{11}s + c_{11}s^2\right) + \beta\left(a_{21} + b_{21}t + c_{21}t^2\right)$$
$$y = \alpha\left(a_{12} + b_{12}s + c_{12}s^2\right) + \beta\left(a_{22} + b_{22}t + c_{22}t^2\right)$$

Параметрите на двата сегмента са свързани с равенството $s = t + 0.5$, а коефициентите можем да определим от условията:

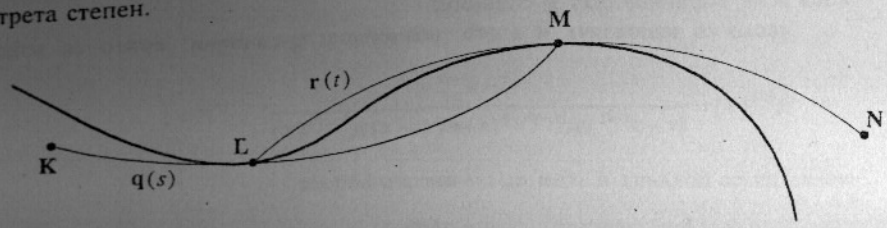
в т. **L** $\quad \alpha = 1 \quad \beta = 0$

в т. **M** $\quad \alpha = 0 \quad \beta = 1$

Да изразим α и β чрез параметъра t (защото изискването беше функциите на смесване да бъдат линейни по отношение на параметрите на сегментите). В участъка между точките L и M от кривата t се мени в интервала [0,1/2], което означава, че стойностите на коефициентите са:

$$\alpha = 1 - 2t \qquad \beta = 2t.$$

Ясно е, че в участъците между първата и втората точки, както и между последната и предпоследната точки от множеството няма смесване на сегменти. Във всички останали участъци резултантната крива на Оверхаузер е от трета степен.



Фиг. 6-5

Едно от големите предимства на тази форма е, че промяна на някоя от интерполираните точки води до промяна само на ограничена част от кривата. Кривата на Оверхаузер има гарантирана непрекъснатост само на първата производна, но това за много приложения е достатъчно.

6.1.5 Интерполация с B-сплайни

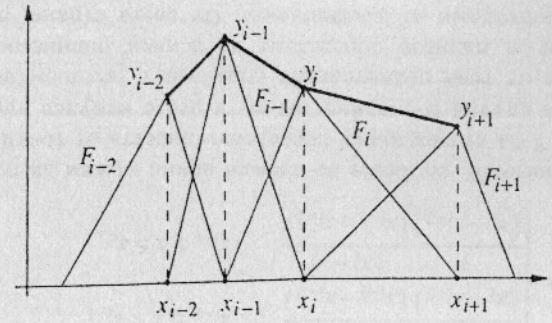
Както бе подчертано по-горе, един от основните недостатъци на сплайните е необходимостта от преизчисляване на целия сплайн при промяна дори само на една от интерполираните стойности - т.нар. локална модификация. Този проблем се решава напълно с въвеждането на т.нар. *B-сплайни* (от Basis spline).

Формулировката на тези криви за пръв път е направена от Шьонберг (Schoenberg) през 1946 год., но едва през 1971 год. Кокс (Cox) и Де Бур (Carl de Boor) независимо един от друг предлагат рекурсивно представяне, която се оказва удобно за компютърна обработка. Преди да дадем формалната дефиниция на B-сплайните и покажем тяхното приложение за интерполация, нека първо да започнем с един елементарен пример, който ще ни покаже какъв е техният механизъм.

Нека са дадени точките (x_0,y_0), (x_1,y_1),..., (x_n,y_n), които искаме да интерполираме. Най-елементарната интерполация - линейната, която е просто начупената линия през тези точки, може да се направи и по следния начин:

Да приемем, че във всяка точка x_i е дефинирана по една триъгълна функция $F_i(x) = y_i B_i(x)$, която има стойност y_i (B_i има стойност 1) в точката x_i и 0 в съседните точки x_{i-1} и x_{i+1}. Тогава начупената през дадените точки ще се зададе с уравнението:

$$\psi(x) = \sum_i F_i(x) = \sum_i y_i B_i(x) \qquad x \in [x_0, x_n]$$



Фиг. 6-6

Наистина, за всяка от интерполираните точки е изпълнено:

$$\psi(x_i) = \sum_k F_k(x_k) = F_i(x_i) = y_i$$

а във всеки интервал $x_{i-1} < t < x_i$ функцията има вида:

$$\psi(t) = \sum_k F_k(t) = F_{i-1}(t) + F_i(t) = y_{i-1}\frac{t - x_{i-1}}{x_i - x_{i-1}} + y_i \frac{x_i - t}{x_i - x_{i-1}},$$

което е уравнение точно на отсечката, свързваща (x_{i-1},y_{i-1}) и (x_i,y_i). Функциите $B_i(x)$ в случая са прост пример за B-сплайни. Сега можем да формулираме и по-ясно задачата: да намерим базисни функции от типа на $B_i(x)$, но от произволна степен m, които са ненулеви в по-широк интервал от (x_{i-1},x_{i+1}). Ако пак използваме линейна комбинация на тези функции, в която коефициенти са изрази, в които участват височините y_i на съответните интерполирани точки, то ще получим уравнение от степен m. Тогава положението на всяка интерполирана точка ще оказва влияние на формата на кривата само в тези няколко интервала, в които базисната ѝ функция е ненулева.

Ще дадем и формалната дефиниция на B-сплайн:

Дефиниция. Нека е зададена последователност от точки: (x_0,y_0), (x_1,y_1),..., (x_n,y_n), като $x_0 < x_1 < \ldots < x_n$. Тогава $M_{m,i}(x)$ се нарича B-сплайн, ако $M_{m,i}(x)$ е сплайн от $m-1$ степен, построен върху тази последователност, който е различен от нула само в m-те интервала $x_{i-m} \le x < x_i$ и удовлетворява уравнението:

$$\int_{x_{i-m}}^{x_i} M_{m,i}(x)dx = \frac{1}{m} \qquad [6.4]$$

B-сплайн от степен $m-1$ се нарича още B-сплайн от ред m. Нека сега поясним тази дефиниция с няколко прости примера.

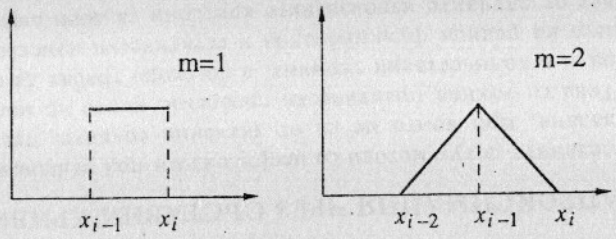
Да разгледаме първо B-сплайна $M_{1,i}(x)$ от степен 0 или ред 1. Той очевидно е константа, т.е. $M_{m,i}(x) = C_1$. Стойността на тази константа за всеки ин-

тервал $x_{i-1} \le x < x_i$ се получава от уравнение [6.4]: $C_1 = (x_i - x_{i-1})^{-1}$.

В-сплайнът $M_{2,i}(x)$ от степен 1 ще е различен от нула само за $x_{i-2} < x < x_i$ и трябва да е C^0-непрекъснат (от дефиницията за сплайн). Това означава, че

$$M_{2,i}(x_{i-2}) = M_{2,i}(x_i) = 0, \qquad [6.5]$$

тъй като вън от границите на разглеждания интервал $M_{2,i}(x) = 0$. Тъй като сплайнът е от първа степен, той се състои от две линейни части (две отсечки) за всеки един от подинтервалите $[x_{i-2}, x_{i-1}]$ и $[x_{i-1}, x_i]$ (подобно на триъгълната функция, която въведохме в началото на този параграф).



Фиг. 6-5

Условието за непрекъснатост в точката x_{i-1} заедно с [6.5] са три зависимости за общо четирите неизвестни коефициента: по два за всяка линейна част. Последната зависимост, необходима за пълното определяне на системата е отново [6.4]:

$$\int_{x_{i-2}}^{x_i} M_{2,i}(x)dx = \frac{1}{2}M_{2,i}(x_{i-1})(x_{i-1}-x_{i-2}) + \frac{1}{2}M_{2,i}(x_{i-1})(x_i - x_{i-1})$$

$$= \frac{1}{2}M_{2,i}(x_{i-1})(x_i - x_{i-2}) = \frac{1}{2}.$$

откъдето следва, че $M_{2,i}(x_{i-1}) = (x_i - x_{i-2})^{-1}$. Тогава сплайнът от степен 1 (или ред 2) се задава с формулата:

$$M_{2,i}(x) = \begin{cases} \dfrac{1}{(x_i - x_{i-2})}\dfrac{(x - x_{i-2})}{(x_{i-1} - x_{i-2})} & x_{i-2} \le x < x_{i-1} \\ \dfrac{1}{(x_i - x_{i-2})}\dfrac{(x_i - x)}{(x_{i-1} - x_{i-2})} & x_{i-1} \le x \le x_i \end{cases}$$

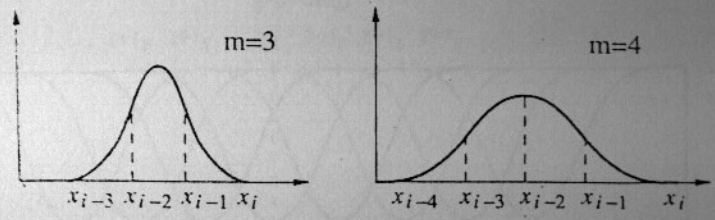
По аналогичен, но малко по-сложен начин можем да получим В-сплайните от 2-ра и 3-та степен върху последователността от точки x_i.

В общия случай В-сплайнът от m-ти ред е ненулев върху m подинтервала. За всеки от тези подинтервали сплайнът е полином от $(m-1)$-ва степен, следователно за пълното определяне на всички полиноми, от които той се състои са необходими m^2 коефициента. От друга страна, от дефиницията на

сплайн от $(m-1)$-ва степен следва, че във всеки от $(m-1)$-те вътрешни възли има по $m-1$ условия, наложени за непрекъснатостта на сплайна и производните му. Тъй като продълженията на сплайна в двата му края са 0, а непрекъснатостта там също трябва да бъде запазена, в тези два края има по още $m-1$ условия от вида:

$$M_{m,i}(x_{i-m}) = M'_{m,i}(x_{i-m}) = M''_{m,i}(x_{i-m}) = \ldots = M^{(m-2)}_{m,i}(x_{i-m}) = 0$$

$$M_{m,i}(x_i) = M'_{m,i}(x_i) = M''_{m,i}(x_i) = \ldots = M^{(m-2)}_{m,i}(x_i) = 0.$$



Фиг. 6-8

Това прави общо $(m-1)^2 + 2(m-1) = m^2 - 1$ условия. Последното, необходимо за да се определят напълно всички коефициенти е [6.4]. От това следва, че съществува точно един В-сплайн от ред m върху m съседни интервала. Има няколко начина да се получи стойността на В-сплайн от даден ред в определена точка. Най-удобно това става с т.нар. *формули на Кокс-Де Бур*, които са рекурентни зависимости, получени по начина, по който и ние започнахме разсъжденията си за В-сплайните от първи и втори ред. От формулата за В-сплайна от първи ред:

$$M_{1,i}(x) = \begin{cases} (x_{i-1} - x_i)^{-1} & x_{i-1} \le x < x_i \\ 0 & x \notin [x_{i-1}, x_i) \end{cases}$$

могат да се получат и тези от по-високи редове:

$$M_{\mu,i}(x) = \frac{(x - x_{i-\mu})M_{\mu-1,i-1}(x) + (x_i - x)M_{\mu-1,i}(x)}{x_i - x_{i-\mu}}, \qquad \mu > 1$$

Често се използват и т.нар. *нормирани В-сплайни*, които се изразяват чрез вече дефинираните В-сплайни:

$$N_{m,i}(x) = (x_i - x_{i-m})M_{m,i}(x)$$

Формулите на Кокс-Де Бур за нормираните В-сплайни се получават веднага от горните:

$$N_{1,i}(x) = \begin{cases} 1 & x_{i-1} \le x < x_i \\ 0 & x \notin [x_{i-1}, x_i) \end{cases}$$

$$N_{\mu,i}(x)=\frac{x-x_{i-\mu}}{x_{i-1}-x_{i-\mu}}N_{\mu-1,i-1}(x)+\frac{x_i-x}{x_i-x_{i-\mu+1}}M_{\mu-1,i}(x),\qquad \mu>1$$

ПРЕДСТАВЯНЕ НА СПЛАЙН КАТО ЛИНЕЙНА КОМБИНАЦИЯ НА B-СПЛАЙНИ. Практическото значение на B-сплайните се обяснява с факта, че всеки сплайн от степен $m-1$ (или ред m) върху множеството от възли x_0, $x_1,\ldots, x_n$ може бъде изразен като линейна комбинация на B-сплайните от m-ти ред върху същото множество от възли, разширено в двата края с по още $m-1$ възела, които могат да се изберат произволно:

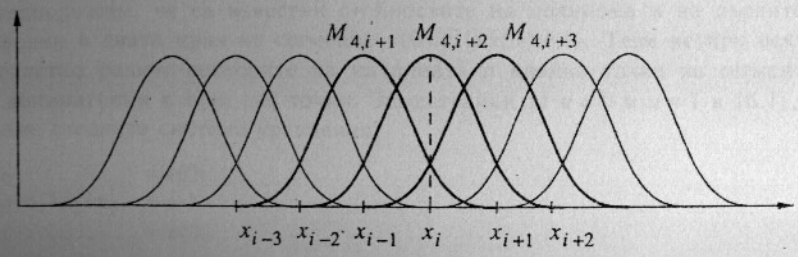
$$x_{-m+1}, x_{-m+2},\ldots, x_{-1}, x_0, x_1\ldots, x_n, x_{n+1}, x_{n+1},\ldots, x_{n+m-1} \qquad [6.6]$$

$$\psi(x)=\sum_{i=1}^{m+n-1} c_i M_{m,i}(x)$$

Всеки сплайн може още да бъде изразен и като линейна комбинация на нормираните сплайни върху същото множество. Този факт е от особено значение, защото B-сплайните са само локално ненулеви и промяната на един от коефициентите в линейната комбинация води до промяна на сплайна само върху m интервала. По-нататък ще обърнем основно внимание на B-сплайните от четвърти ред (или 3-та степен), макар че в практиката често се използват сплайни и от по-високи редове.

Нека сега разгледаме един сплайн $\psi(x)$ от трета степен, дефиниран върху множеството от точки $x_0, x_1,\ldots, x_n$. Да допълним интервала с по три произволни възела в двата края, които да не съвпадат (защото тогава би се получило делене на 0 във формулите на Кокс-Де Бур).

Както се вижда на фиг. 6-9, в точката x_i ненулеви са само B-сплайните $M_{4,i+1}(x)$, $M_{4,i+2}(x)$ и $M_{4,i+3}(x)$.



Фиг. 6-9

Ако интерполираната стойност в точката x_i е y_i, от [6.6] ще получим следното уравнение:

$$y_i=\psi(x_i)=c_{i+1}M_{4,i+1}(x_i)+c_{i+2}M_{4,i+2}(x_i)+c_{i+3}M_{4,i+3}(x_i),$$

където неизвестни са коефициентите c_i. Такива уравнения можем да запишем за всеки от възлите $x_0, x_1,\ldots, x_n$. Тъй като на разширеното множество има $n+3$ B-сплайна, неизвестните коефициенти $c_1, c_2,\ldots, c_{n+3}$ са също $n+3$. Както и при интерполацията със сплайни са необходими още две допълнителни условия за пълното определяне на $\psi(x)$. Както бе показано по-горе, обикновено се изисква втората производна в двата края да е 0:

$$y_0''=\psi''(x_0)=c_1M''_{4,1}(x_0)+c_2M''_{4,2}(x_0)+c_3M''_{4,3}(x_0)=0$$
$$y_n''=\psi''(x_n)=c_{n+1}M''_{4,n+1}(x_n)+c_{n+2}M''_{4,n+2}(x_n)+c_{n+3}M''_{4,n+3}(x_n)=0$$

Производните на B-сплайните в горните формули могат да бъдат намерени по формулата на Кокс-Де Бур. След като се реши тридиагоналната система относно c_i, стойността на линейната комбинация [6.6] може да бъде получена за всяко x.

Аналогично на параметричните сплайни, интерполирането чрез B-сплайни може да се извърши и параметрично. Необходимо е само въвеждането на независим параметър, което ще доведе както и преди до образуването на две множества от точки, всяко от които се интерполира поотделно:

$$\mathbf{r}(u)=\sum_{i=1}^{m+n-1}\mathbf{c}_i M_{m,i}(u)$$

където векторните коефициенти $\mathbf{c}_i=(c_{1,i},c_{2,i})$ са решения на системата:

$$\begin{aligned}
&c_{11}M''_{4,1}(u_0)+c_{12}M''_{4,2}(u_0)+c_{13}M''_{4,3}(u_0)=0\\
&c_{1,i+1}M_{4,i+1}(u_i)+c_{1,i+2}M_{4,i+2}(u_i)+c_{1,i+3}M_{4,i+3}(u_i)=x_i \qquad i=0,\ldots,n\\
&c_{1,n+1}M''_{4,n+1}(u_n)+c_{1,n+2}M''_{4,n+2}(u_n)+c_{1,n+3}M''_{4,n+3}(u_n)=0\\
&c_{21}M''_{4,1}(u_0)+c_{22}M''_{4,2}(u_0)+c_{23}M''_{4,3}(u_0)=0\\
&c_{2,i+1}M_{4,i+1}(u_i)+c_{2,i+2}M_{4,i+2}(u_i)+c_{2,i+3}M_{4,i+3}(u_i)=y_i \qquad i=0,\ldots,n\\
&c_{2,n+1}M''_{4,n+1}(u_n)+c_{2,n+2}M''_{4,n+2}(u_n)+c_{2,n+3}M''_{4,n+3}(u_n)=0
\end{aligned}$$

6.2 АПРОКСИМАЦИЯ ЧРЕЗ СЪСТАВНИ КРИВИ

Разгледаните дотук методи са неприложими при решаването на един друг клас от задачи, при които не са от значение точките, през които кривата минава. Това са такива практически проблеми, които не могат да се формулират изцяло в количествени термини, а по-скоро трябва да се решат при съблюдаването на редица функционални и естетически изисквания. Тъй като в голяма част от случаите използваните критерии са дори евристични, математическата формулировка трябва да е пригодна за лесно управление на формата на кривата чрез пряк контрол върху такива характеристики като кривината ѝ, наклона и изменението на допирателната към нея и др. Ето защо разбиването на кривата на *сегменти*, всеки от които да може да се променя сравнително независимо от останалите при запазване на глобалните характеристики на кривата, е един от най-удобните методи за моделирането на криви с произволна форма. Тук ще разгледаме апроксимацията чрез такива именно *съставни криви*, всеки отделен сегмент от които апроксимира някаква група

от последователни точки. Първо нека разгледаме какви са най-разпространените начини за представяне на сегментите, след което ще видим и как се получават самите съставни криви. Тъй като ще разглеждаме изключително полиномиални криви от трета степен, във формулите по-нататък често ще използваме матричен запис за кривата:

$$\mathbf{r}(\mathbf{U}) = \mathbf{U}\cdot\mathbf{A}\cdot\mathbf{G}$$

където $\mathbf{U} = [1\ u\ u^2\ u^3]$ е параметричният вектор, **G** е т.нар. *геометричен вектор* или *вектор от геометрични коефициенти*, а матрицата **A** е специфична за всяко конкретно представяне. Основната цел на следващите представяния е да се намери подходящ геометричен вектор, чрез който удобно да се задава формата на моделираната крива.

6.2.1 Представяне на сегменти от криви

Ще разгледаме по-подробно само два от начините за представяне на сегменти от криви. Задачата, която се поставя е: при известни крайни точки, да се намерят още няколко подходящи параметъра (задаващи напълно споменатия геометричен вектор), които участват явно в уравнението на кривата и за които е ясно как изменението на стойностите им влияе върху формата ѝ.

СЕГМЕНТИ ОТ КУБИЧНИ КРИВИ ВЪВ ФОРМАТА НА ФЪРГЮСЪН. Първото използване на полиноми от трета степен за параметрично представяне на криви е на Фъргюсън (Ferguson) за компанията Boeing през 1963 год. Тази форма е известна още като Ермитова форма. Един сегмент се задава с уравнението:

$$\mathbf{r} = \mathbf{r}(u) = \mathbf{a}_0 + u\mathbf{a}_1 + u^2\mathbf{a}_2 + u^3\mathbf{a}_3, \qquad [6.7]$$

където u се изменя в интервала [0,1]. За да се определи напълно **r** в това представяне е необходимо да са известни четирите векторни коефициента $\mathbf{a}_i$. Да предположим, че са известни стойностите на полинома и на първите му производни в двата края на сегмента $\mathbf{r}(0), \mathbf{r}(1), \dot{\mathbf{r}}(0), \dot{\mathbf{r}}(1)$. Тези четири вектора са съответно радиус-векторите на началната и крайна точка на сегмента и двете допирателни в тези две точки. Заместваики за $u = 0$ и $u = 1$ в [6.7], получаваме следната система уравнения:

$$\begin{aligned}
\mathbf{a}_0 &= \mathbf{r}(0)\\
\mathbf{a}_0+\mathbf{a}_1+\mathbf{a}_2+\mathbf{a}_3 &= \mathbf{r}(1)\\
\mathbf{a}_1 &= \dot{\mathbf{r}}(0)\\
\mathbf{a}_1+2\mathbf{a}_2+3\mathbf{a}_3 &= \dot{\mathbf{r}}(1)
\end{aligned}$$

От нея определяме коефициентите $\mathbf{a}_i$:

$$\begin{aligned}
\mathbf{a}_0 &= \mathbf{r}(0)\\
\mathbf{a}_1 &= \dot{\mathbf{r}}(0)\\
\mathbf{a}_2 &= 3[\mathbf{r}(1)-\mathbf{r}(0)]-2\dot{\mathbf{r}}(0)-\dot{\mathbf{r}}(1)\\
\mathbf{a}_3 &= 2[\mathbf{r}(0)-\mathbf{r}(1)]+\dot{\mathbf{r}}(0)+\dot{\mathbf{r}}(1)
\end{aligned}$$

и след заместване в [6.7] получаваме уравнението:

$$\mathbf{r}(u) = \mathbf{r}(0)(1-3u^2+2u^3)+\mathbf{r}(1)(3u^2-2u^3)+\dot{\mathbf{r}}(0)(u-2u^2+u^3)+\dot{\mathbf{r}}(1)(-u^2+u^3) \qquad [6.8]$$

То може да бъде записано в матрична форма спрямо вектора U от степените на параметъра по следния начин:

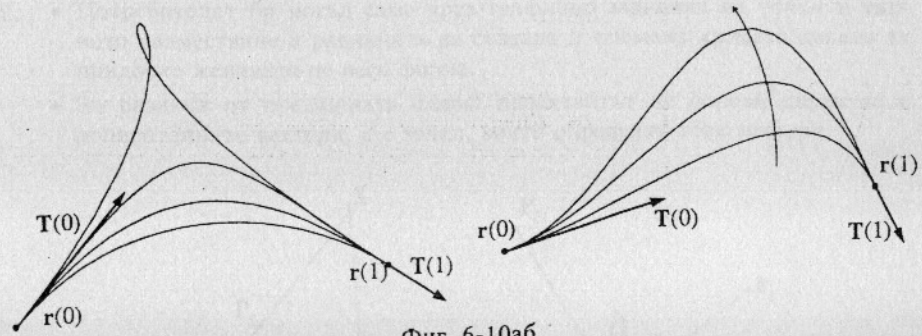
$$\mathbf{r}(\mathbf{U}) = \mathbf{U}\cdot\mathbf{F}\cdot\mathbf{R}$$

$$\mathbf{r}(\mathbf{U}) = \underbrace{\begin{bmatrix}1 & u & u^2 & u^3\end{bmatrix}}_{\mathbf{U}}\cdot\underbrace{\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ -3 & 3 & -2 & -1\\ 2 & -2 & 1 & 1\end{bmatrix}}_{\mathbf{F}}\cdot\underbrace{\begin{bmatrix}\mathbf{r}(0)\\ \mathbf{r}(1)\\ \dot{\mathbf{r}}(0)\\ \dot{\mathbf{r}}(1)\end{bmatrix}}_{\mathbf{R}}$$

Ползата от направеното дотук е, че сега можем да представим сегмента като полином с векторни коефициенти $\mathbf{R} = [\mathbf{r}(0)\ \mathbf{r}(1)\ \dot{\mathbf{r}}(0)\ \dot{\mathbf{r}}(1)]$, които имат конкретен геометричен смисъл, а именно: първите два са радиус-векторите на краищата на сегмента, а последните два са пропорционални на единичните допирателни вектори $\mathbf{T}(0)$ и $\mathbf{T}(1)$ в същите точки:

$$\begin{aligned}
\dot{\mathbf{r}}(0) &= k_0\mathbf{T}(0)\\
\dot{\mathbf{r}}(1) &= k_1\mathbf{T}(1)
\end{aligned} \qquad [6.9]$$

Интерпретацията на коефициентите в [6.9] е следната: когато те растат едновременно, кривата се извива, отдалечавайки се от отсечката, свързваща двата ѝ края. При големи стойности на тези коефициенти кривата може да започне и да се самопресича.



Фиг. 6-10аб

Когато само един от коефициентите расте, кривата в по-голямата си част започва да следва допирателната в съответната точка. За равнинни криви е възможно коефициентите k_0 и k_1 да се определят така, че кривината к да има определени стойности в двата края на сегмента.

СЕГМЕНТИ ОТ КУБИЧНИ КРИВИ ВЪВ ФОРМАТА НА БЕЗИЕ. Друг начин за параметрично представяне на криви предлага Пиер Безие (P.

E. Bézier) в началото на 60-те години. Създадената от него форма е приложена най-напред в системата UNISURF, разработена за фирмата Renault, а същата система по-късно става основа на небезизвестната EUCLID на Matra Datavision. Интересно е да се отбележи, че през 1959 год. независимо от Безие, Де Кастельо (de Casteljau) за фирмата Citroën построява пълната математическа формулировка на същия тип криви. Безие използва следния полином за съставяне на параметричното уравнение:

$$\mathbf{r} = \mathbf{r}(u) = (1-u)^3\mathbf{r}_0+3(1-u)^2u\mathbf{r}_1+3(1-u)u^2\mathbf{r}_2+u^3\mathbf{r}_3, \quad 0 \le u \le 1 \qquad [6.10]$$

За да разберем какъв е геометричният смисъл на векторните коефициенти $\mathbf{r}_i$, нека да изразим чрез тях $\mathbf{r}(0)$, $\mathbf{r}(1)$, $\dot{\mathbf{r}}(0)$, $\dot{\mathbf{r}}(1)$. Това правим, като заместваме в [6.10] с 0 и 1, а за втората двойка диференцираме и заместваме отново.

$$\begin{aligned} \mathbf{r}(0) &= \mathbf{r}_0 \\ \mathbf{r}(1) &= \mathbf{r}_3 \\ \dot{\mathbf{r}}(0) &= 3(\mathbf{r}_1-\mathbf{r}_0) \\ \dot{\mathbf{r}}(1) &= 3(\mathbf{r}_3-\mathbf{r}_2) \end{aligned} \qquad [6.11]$$

По подобие на предишното представяне, матрична форма е: $\mathbf{r}(\mathbf{U}) = \mathbf{U}\cdot\mathbf{B}\cdot\mathbf{R}$.

$$\mathbf{r}(\mathbf{U}) = \underbrace{\begin{bmatrix} 1 & u & u^2 & u^3 \end{bmatrix}}_{\mathbf{U}} \cdot \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}}_{\mathbf{B}} \cdot \underbrace{\begin{bmatrix} \mathbf{r}(0) \\ \mathbf{r}(1) \\ \dot{\mathbf{r}}(0) \\ \dot{\mathbf{r}}(1) \end{bmatrix}}_{\mathbf{R}} \qquad [6.12]$$

Следователно, ако считаме че $\mathbf{r}_0$, $\mathbf{r}_1$, $\mathbf{r}_2$ и $\mathbf{r}_3$ са радиус-векторите на четири точки в равнината, разглежданият сегмент има за крайни точки $\mathbf{r}_0$ и $\mathbf{r}_3$, а допирателните в двата му края са колинеарни с векторите $\mathbf{r}_1-\mathbf{r}_0$ и $\mathbf{r}_3-\mathbf{r}_2$. Нещо повече, дължината на последните два вектора определя и големината на производните на сегмента в двата му края.

Такова представяне има много важно приложение в моделирането, тъй като векторните коефициенти в параметричната форма служат директно за управляващи точки на кривата. Начупената, образувана от тези точки, се нарича *характеристична начупена* или *управляваща начупена на Безие*.

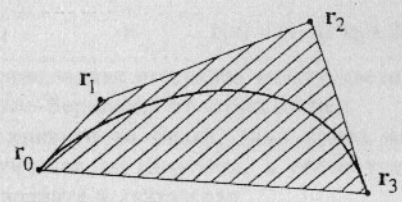
Ще разгледаме някои свойства на кривите на Безие:

<u>Свойство 1.</u> Свойство на изпъкналата обвивка: Всяка крива на Безие лежи изцяло в изпъкналата обвивка на характеристичната си начупена (фиг. 6-11). Наистина, ако разгледаме уравнението [6.10] ще видим, че $\mathbf{r}(u)$ е линейна комбинация на векторите $\mathbf{r}_i$ и стойностите на коефициентите в тази линейна комбинация са винаги между 0 и 1, а сумата им е 1 за всяко u в интервала [0,1]:

$$\mathbf{r}(u) = \sum_i \lambda_i(u)\mathbf{r}_i, \quad \text{където} \quad 0 \le \lambda_i(u) \le 1 \text{ и}$$

$$\sum_i \lambda_i(u) = [(1-u)+u]^3 = 1 \qquad \forall u \in [0,1]$$

Да си припомним, че изпъкналата обвивка е често използвана за бързо осъществяване на отсичане при визуализация на обекти.



Фиг. 6-11

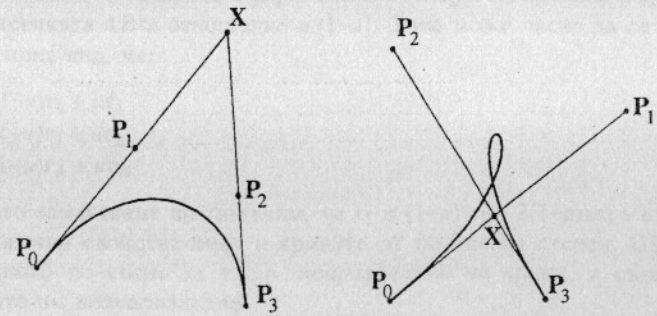
<u>Свойство 2.</u> Ако върховете на характеристичната начупена са колинеарни, то кривата на Безие се изражда в отсечка - това е просто следствие от горното свойство.

<u>Свойство 3.</u> Нека $\mathbf{P}_0$,$\mathbf{P}_1$,$\mathbf{P}_2$ и $\mathbf{P}_3$ са точките с радиус-вектори съответно $\mathbf{r}_0$, $\mathbf{r}_1$,$\mathbf{r}_2$ и $\mathbf{r}_3$, а $\mathbf{X}$ е пресечната точка на правите, носещи векторите $\mathbf{r}_1-\mathbf{r}_0$ и $\mathbf{r}_3-\mathbf{r}_2$. Ако означим с α и β коефициентите в равенствата $\mathbf{P}_0\mathbf{P}_1 = \alpha\mathbf{P}_0\mathbf{X}$ и $\mathbf{P}_2\mathbf{P}_3 = \beta\mathbf{X}\mathbf{P}_3$, то кривата се самопресича, ако са изпълнени трите условия:

$$(\alpha-\frac{4}{3})(\beta-\frac{4}{3}) > \frac{4}{9}, \qquad \alpha > 1, \qquad \beta > 1$$

Това представяне дава по-удобни средства за моделиране, а именно:

- Потребителят би могъл само чрез графично задаване на точки и тяхното разместване в равнината да създава и променя кривата докато тя придобие желаната от него форма.
- За разлика от предишната форма проектантът не борави директно с допирателните вектори, а с точки, които определят тези вектори.



Фиг. 6-12

Формата на Безие е удобна за изчисляване на кривината на кривата в

крайните ѝ точки. Това често се налага при генериране на криви в самолетостроенето и корабостроенето. *Кривината* κ е реципрочна на т.нар. *радиус на кривината* R, който за всяка точка от кривата (x,y) е радиусът на окръжност, имаща най-висока степен на допиране до кривата в тази точка. Центърът на кривината в тази точка е центърът на тази окръжност. Нека да използваме формулите на Френе от диференциалната геометрия:

$$\frac{d\mathbf{r}}{ds}=\mathbf{T}, \qquad \frac{d\mathbf{T}}{ds}=\kappa\mathbf{N}, \qquad \mathbf{N}=\mathbf{B}\times\mathbf{T},$$

където **N**, **B** и **T** са единичните вектори на главната нормала, бинормалата и допирателната съответно, а кривата **r** е параметризирана относно дължината на дъгата s. Ако преминем към независим параметър, тези формули ще приемат вида:

$$\frac{\dot{\mathbf{r}}}{\dot{s}}=\mathbf{T}, \qquad \frac{\ddot{\mathbf{r}}}{\dot{s}^2}=\kappa\mathbf{N}, \qquad \dot{s}=|\dot{\mathbf{r}}|,$$

и тогава за векторното произведение на първата и втората производни можем да направим следния извод:

$$\dot{\mathbf{r}}\times\ddot{\mathbf{r}}=\kappa\dot{s}^3\mathbf{T}\times\mathbf{N}=\kappa\dot{s}^3\mathbf{B} \qquad \text{откъдето} \quad \kappa\mathbf{B}=\frac{\dot{\mathbf{r}}\times\ddot{\mathbf{r}}}{|\dot{\mathbf{r}}|^3}.$$

Тъй като **B** е единичен вектор, формулата за кривината е:

$$\kappa=\frac{|\dot{\mathbf{r}}\times\ddot{\mathbf{r}}|}{|\dot{\mathbf{r}}|^3}.$$

След диференциране на [6.10] и заместване с 0 и 1 получаваме следните стойности за вторите производни в двата края на сегмента:

$$\ddot{\mathbf{r}}(0)=6(\mathbf{r}_0-2\mathbf{r}_1+\mathbf{r}_2)=6(\mathbf{r}_2-\mathbf{r}_1)-6(\mathbf{r}_1-\mathbf{r}_0)$$
$$\ddot{\mathbf{r}}(1)=6(\mathbf{r}_1-2\mathbf{r}_2+\mathbf{r}_3)=6(\mathbf{r}_3-\mathbf{r}_2)-6(\mathbf{r}_2-\mathbf{r}_1)$$

Като използваме отново зависимостите [6.11] и свойствата на векторното произведение, за кривината в двата края получаваме следните формули:

$$\kappa(0)=2\frac{|(\mathbf{r}_1-\mathbf{r}_0)\times(\mathbf{r}_2-\mathbf{r}_1)|}{3|\mathbf{r}_1-\mathbf{r}_0|^3}, \qquad \kappa(1)=2\frac{|(\mathbf{r}_2-\mathbf{r}_1)\times(\mathbf{r}_3-\mathbf{r}_2)|}{3|\mathbf{r}_3-\mathbf{r}_2|^3}. \qquad [6.13]$$

Този резултат за кривината ще използваме по-късно при съединяване на отделните сегменти в съставна крива.

СЕГМЕНТИ ОТ КРИВИ ВЪВ ФОРМА НА БЕЗИЕ-БЕРНЩАЙН. Формата на Безие е частен случай на един по-общ метод, основаващ се на полиномите на Бернщайн (Bernstein). Наистина полиномът, които използвахме погоре [6.10], е полином на Бернщайн от трета степен. В общия случай при известни радиус-вектори $\mathbf{r}_i$ на $n+1$ характеристични точки в равнината, параметричното уравнение на сегмента би било:

$$\mathbf{r}(u)=\sum_{i=0}^{n}\frac{n!}{i!(n-i)!}u^i(1-u)^{n-i}\mathbf{r}_i$$

При $n=1$ и $n=2$ се получават съответно:

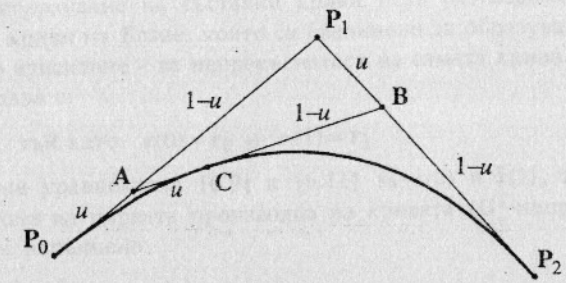
$$\mathbf{r}(u)=(1-u)\mathbf{r}_0+u\mathbf{r}_1 \qquad \text{и} \qquad \mathbf{r}(u)=(1-u)^2\mathbf{r}_0+2(1-u)u\mathbf{r}_1+u^2\mathbf{r}_2$$

Първото уравнение задава отсечката между двете точки, а второто представя кривата на Безие-Бернщайн от втора степен.

Аналогично на кривите на Безие, тази крива минава през началото и края на характеристичната си начупена, а освен това двете двойки крайни точки задават производните в двата края:

$$\mathbf{r}(0)=\mathbf{r}_0 \qquad \dot{\mathbf{r}}(0)=n(\mathbf{r}_1-\mathbf{r}_0)$$
$$\mathbf{r}(1)=\mathbf{r}_n \qquad \dot{\mathbf{r}}(1)=n(\mathbf{r}_n-\mathbf{r}_{n-1})$$

Сегментите на Безие-Бернщайн имат едно интересно свойство, което лесно се илюстрира за тези от втора степен. На това именно свойство се основава методът за получаване на криви на Безие, разработен от Де Кастельо:



Фиг. 6-13

Нека **A** е точката, разделяща отсечката $\mathbf{P}_0\mathbf{P}_1$ в отношение u:$(1-u)$, а **B** разделя в същото отношение $\mathbf{P}_1\mathbf{P}_2$. Тогава точката **C**, лежаща върху кривата, разделя отсечката **AB** в отношение u:$(1-u)$. Това може лесно да се покаже, като се има пред вид, че:

$$\mathbf{r}_A=(1-u)\mathbf{r}_0+u\mathbf{r}_1$$
$$\mathbf{r}_B=(1-u)\mathbf{r}_1+u\mathbf{r}_2$$
$$\mathbf{r}_C=(1-u)\mathbf{r}_A+u\mathbf{r}_B$$

След просто заместване ще получим, че $\mathbf{r}_C=(1-u)^2\mathbf{r}_0+2(1-u)u\mathbf{r}_1+u^2\mathbf{r}_2$.

Аналогични свойства имат и кривите от по-висока степен. Ще използваме показаното по-късно за т.нар. *подразделяне на криви*, а също и за по-ефективното им визуализиране.

6.2.2 Построяване на съставни криви

Нека сега да видим как сегментите се свързват за образуването на съставна крива. Задачата за построяването на съставни криви е следната: При

зададено множество от сегменти от криви с еднакво представяне, които апроксимират групи от точки, да се намерят параметрите на всеки сегмент, така че получената съставна крива да има определени свойства. Тук ще дефинираме накратко по-често използваните свойства.

Във всички случаи съставната крива трябва да е непрекъсната при преход от един сегмент към друг. Използвайки термините на математическия анализ, ще казваме че кривата трябва да е C^0-непрекъсната. Когато кривата има непрекъснат наклон в точките на съединяване, т.е. посоката на допирателните в тези точки съвпада (без непременно големините им да са равни), ще казваме, че кривата е G^1-непрекъсната. Тук използваме G за да означим *геометрична непрекъснатост*. Когато производните са равни като вектори - и по посока, и по големина, кривата има стандартната C^1-непрекъснатост, т.е. непрекъснатост по параметъра u. По аналогия, под G^n-непрекъснатост ще разбираме непрекъснатост на посоката на n–тата производна, а под C^n-непрекъснатост - непрекъснатост на n–тата производна по параметъра u. Очевидно е, че една крива може да е G^n-непрекъсната, без непременно да е C^n-непрекъсната.

В общия случай C^n-непрекъснатост гарантира и G^n-непрекъснатост, макар че има едно особено изключение: когато допирателните вектори в точката на съединяване са едновременно 0. Това може да се случи, когато производната по параметъра намалява до 0 в единия сегмент, а в следващия започва да расте от нула.

G^1-кривите са често използвани в моделирането поради добрите си геометрични характеристики и наличието на повече степени на свобода от съответните C^1-криви.

Друго често налагано условие е в точките на съединяване кривината да е също непрекъсната. Нека разгледаме сега как се използват горните две форми за представяне на сегментите при тяхното съединяване.

СЪСТАВНИ КРИВИ ОТ СЕГМЕНТИ НА ФЪРГЮСЪН. Нека са дадени n сегмента $\mathbf{r}^{(1)}(u)$, $\mathbf{r}^{(2)}(u)$,..., $\mathbf{r}^{(n)}(u)$, чиито краища алтернативно съвпадат, т.е. за които е изпълнено:

$$\mathbf{r}^{(1)}(1) = \mathbf{r}^{(2)}(0), \quad \mathbf{r}^{(2)}(1) = \mathbf{r}^{(3)}(0),..., \quad \mathbf{r}^{(n-1)}(1) = \mathbf{r}^{(n)}(0). \qquad [6.14]$$

Да приемем, че се изисква съставната крива да бъде C^2-непрекъсната. Тогава в точките на съединяване, първата и втората производни на всеки два сегмента трябва да съвпадат. От тези условия се получава системата:

$$\begin{aligned} &\dot{\mathbf{r}}^{(i)}(1) = \dot{\mathbf{r}}^{(i+1)}(0) \quad \text{и} \\ &\ddot{\mathbf{r}}^{(i)}(1) = \ddot{\mathbf{r}}^{(i+1)}(0) \quad \text{за} \quad i=1,2,...,n-1 \end{aligned} \qquad [6.15]$$

допълнена с условието $k_0 = k_1$ от [6.9]. Формулата за вторите производни можем да получим, като диференцираме два пъти параметричното уравнение [6.8] на сегмента, а после заместим с 0 и 1:

$$6\mathbf{r}^{(i)}(0) - 6\mathbf{r}^{(i)}(1) + 2\dot{\mathbf{r}}^{(i)}(0) + 4\dot{\mathbf{r}}^{(i)}(1) = -6\mathbf{r}^{(i+1)}(0) + 6\mathbf{r}^{(i+1)}(1) - 4\dot{\mathbf{r}}^{(i+1)}(0) - 2\dot{\mathbf{r}}^{(i+1)}(1)$$

Приемайки, че знаем радиус-векторите $\mathbf{r}_i$ на точките, през които съставната крива ще премине, можем да положим $\mathbf{t}_0 = \dot{\mathbf{r}}^{(1)}(0)$, $\mathbf{t}_i = \dot{\mathbf{r}}^{(i)}(1)$, $i=1,...,n$ и като използваме зависимостите [6.14] и [6.15], ще получим следната система за неизвестните първи производни:

$$\mathbf{t}_{i-1} + 4\mathbf{t}_i + \mathbf{t}_{i+1} = 3(\mathbf{r}_{i+1} - \mathbf{r}_{i-1}), \quad i=1,2,...,n-1 \qquad [6.16]$$

Броят на неизвестните $\mathbf{t}_i$ за определянето на съставната крива е $n+1$. Системата [6.16], допълнена със стойностите на производните в началото и края на съставната крива $\mathbf{t}_0 = \dot{\mathbf{r}}^{(1)}(0)$ и $\mathbf{t}_n = \dot{\mathbf{r}}^{(n)}(1)$ е достатъчна за намирането на $\dot{\mathbf{r}}^{(i)}$, с които пък става възможно определянето на всеки от отделните сегменти, образуващи кривата.

Следователно, за проектанта е достатъчно да зададе точките, през които трябва да мине кривата и стойностите на допирателните в двата й края, за да може по този метод да се генерира една гладка крива. Методът на Фъргюсън добре подхожда за автоматично построяване на съставна крива, когато зададените точки са разположени достатъчно равномерно.

СЪСТАВНИ КРИВИ ОТ СЕГМЕНТИ НА БЕЗИЕ. По подобие на горния начин за образуване на съставни криви нека разгледаме два сегмента $\mathbf{r}^{(1)}(u)$, $\mathbf{r}^{(2)}(u)$ от криви на Безие, които са съединени за образуване на съставна крива. Първото изискване - за непрекъснатост на самата крива (C^0-непрекъснатост) - се задава с:

$$\mathbf{r}_3^{(1)} = \mathbf{r}_0^{(2)}, \quad \text{тъй като} \quad \mathbf{r}(0) = \mathbf{r}_0 \quad \text{и} \quad \mathbf{r}(1) = \mathbf{r}_3$$

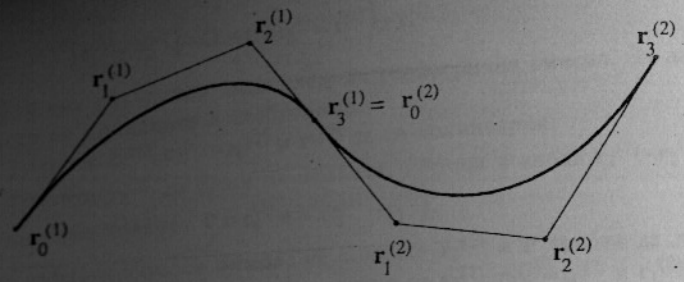
Използвайки уравненията [6.9] и [6.11] за $\dot{\mathbf{r}}(0)$ и $\dot{\mathbf{r}}(1)$, за непрекъснатостта на наклона на първата производна на кривата (G^1-непрекъснатост), е необходимо да е изпълнено:

$$\frac{3}{k_0}\left(\mathbf{r}_3^{(1)} - \mathbf{r}_2^{(1)}\right) = \frac{3}{k_1}\left(\mathbf{r}_1^{(2)} - \mathbf{r}_0^{(2)}\right) = \mathbf{T} \qquad [6.17]$$

където k_0 и k_1 от [6.9] са дължините на допирателните вектори в съвпадащите краища, които в общия случай могат да имат различни стойности. От горните две равенства следва, че точките $\mathbf{r}_2^{(1)}$, $\mathbf{r}_3^{(1)} = \mathbf{r}_0^{(2)}$ и $\mathbf{r}_1^{(2)}$ трябва да лежат на една права.

По подобие на предишната форма можем да поискаме и C^2-непрекъснатост. Следвайки описаната по-горе процедура на диференциране на параметричното уравнение и заместване с 0 и 1, това би довело до подобна система уравнения за неизвестните векторни коефициенти $\mathbf{r}_1^{(i)}$ и $\mathbf{r}_2^{(i)}$.

Много често кривите на Безие се използват, като се задават интерактивно всички точки, управляващи сегментите, а дали ще има и каква непрекъснатост се контролира единствено от положението на точките $\mathbf{r}_1^{(i)}$ и $\mathbf{r}_2^{(i)}$, през които кривата не минава. При зададена характеристична начупена с точките `P[i]`, $i = 0,1,...,n$ следната функция ще визуализира кривата при подходящо избрана стойност на стъпката `step`. Очевидно е, че броят на точките не е произволен, а именно $n = 3k+1$, където k е броят на сегментите.



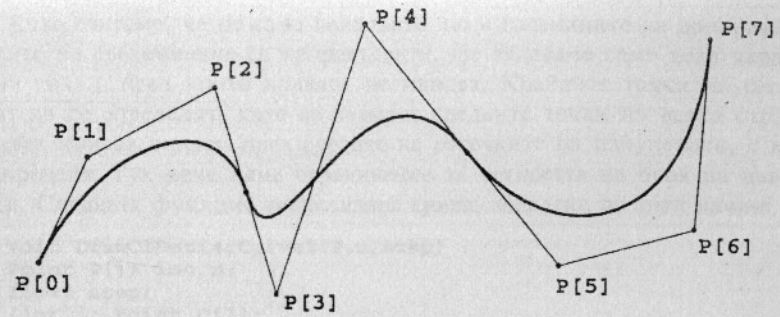
Фиг. 6-14

```
void DrawBezierSegment3(P,i,step)
Point P[]; int i;
float step;
{float x,y,u;
 float ax,bx,cx,dx, ay,by,cy,dy;
    ax= P[i].x;
    ay= P[i].y;
    bx=-3*(P[i].x-P[i+1].x);
    by=-3*(P[i].y-P[i+1].y);
    cx= 3*(P[i].x-2*P[i+1].x+P[i+2].x);
    cy= 3*(P[i].y-2*P[i+1].y+P[i+2].y);
    dx=-P[i].x+3*(P[i+1].x-P[i+2].x)+P[i+3].x;
    dy=-P[i].y+3*(P[i+1].y-P[i+2].y)+P[i+3].y;
    for (u=0; u<1; u+=step) {
        x=(ax+u*(bx+u*(cx+dx*u)));
        y=(ay+u*(by+u*(cy+dy*u)));
        DrawTo(x,y);
    }
}

void DrawBezierCurve3(P,n,step)
Point P[]; int n;
float step;
{int i;
    MoveTo(P[0].x,P[0].y);
    for (i=0; i<n; i+=3)
        DrawBezierSegment3(P,i,step);
    DrawTo(P[n].x,P[n].y);
}
```

За да намалим броя на умноженията в програмата за чертане на един сегмент от кривата, използваме правилото на Хорнер (Horner) за пресмятане на стойността на полином в точка:

$$f(u) = au^3+bu^2+cu+d = ((au+b)u+c)u+d$$

За постигане на C^1-непрекъснатост независимо от задаваните данни (при положение, разбира се, че се изключва наличието на съвпадащи съседни характеристични точки) може входният масив да съдържа само точки, съответстващи на $\mathbf{r}_1^{(i)}$ и $\mathbf{r}_2^{(i)}$, а за точките $\mathbf{r}_0^{(i)} = \mathbf{r}_3^{(i-1)}$, да се смята че лежат на средата на отсечките $|\mathbf{r}_1^{(i)}-\mathbf{r}_2^{(i-1)}|$, т.е. да се приеме, че $k_0 = k_1$ от [6.9]. С това естествено ще се намалят степените на свобода на кривата.



Фиг. 6-13

Броят на задаваните точки в този случай трябва да е четен: $n = 2k+2$, където k отново е броят на сегментите.

Нека сега разгледаме случая, в който се изисква кривината в точките на съединяване да е непрекъсната. Можем да използваме формулите [6.13] за кривината в краищата на един сегмент, като ги приравним като вектори, защото кривите, които разглеждаме, са равнинни и векторното произведение в числителя има постоянна посока. Посоката на векторите в знаменателя пък съвпада заради непрекъснатостта на посоката на първата производна. Тогава ще получим:

$$\frac{(\mathbf{r}_1^{(2)}-\mathbf{r}_0^{(2)})\times(\mathbf{r}_2^{(2)}-\mathbf{r}_1^{(2)})}{(\mathbf{r}_1^{(2)}-\mathbf{r}_0^{(2)})^3} = \frac{(\mathbf{r}_2^{(1)}-\mathbf{r}_1^{(1)})\times(\mathbf{r}_3^{(1)}-\mathbf{r}_2^{(1)})}{(\mathbf{r}_3^{(1)}-\mathbf{r}_2^{(1)})^3}$$

Ако използваме зависимостите [6.17] ще получим:

$$\mathbf{T}\times\left(\mathbf{r}_2^{(2)}-\mathbf{r}_1^{(2)}\right) = \left(\frac{k_1}{k_0}\right)^2\left(\mathbf{r}_2^{(1)}-\mathbf{r}_1^{(1)}\right)\times\mathbf{T}$$

Това уравнение се удовлетворява (от свойствата на векторното произведение) тогава, когато:

$$\mathbf{r}_2^{(2)}-\mathbf{r}_1^{(2)} = -\left(\frac{k_1}{k_0}\right)^2\left(\mathbf{r}_2^{(1)}-\mathbf{r}_1^{(1)}\right)+c\mathbf{T} = -\left(\frac{k_1}{k_0}\right)^2\left(\mathbf{r}_2^{(1)}-\mathbf{r}_1^{(1)}\right)+\frac{3c}{k_0}\left(\mathbf{r}_3^{(1)}-\mathbf{r}_2^{(1)}\right)$$

където c е произволен скалар. Като използваме отново [6.17], за да елиминираме $\mathbf{r}_1^{(2)}$ и като положим $\lambda = k_1/k_0$, $\mu = 3c/k_0$, ще получим:

$$\mathbf{r}_2^{(2)} = -\lambda^2\left(\mathbf{r}_2^{(1)}-\mathbf{r}_1^{(1)}\right)+\mu\left(\mathbf{r}_3^{(1)}-\mathbf{r}_2^{(1)}\right)+\lambda\left(\mathbf{r}_3^{(1)}-\mathbf{r}_2^{(1)}\right)+\mathbf{r}_3^{(1)} \quad \text{или}$$

$$\mathbf{r}_2^{(2)} = \lambda^2\mathbf{r}_1^{(1)}-(\lambda^2+\lambda+\mu)\mathbf{r}_2^{(1)}+(\lambda+\mu+1)\mathbf{r}_3^{(1)}$$

С това успяхме да изразим точките на характеристичната начупена на втория сегмент чрез тези на първия, а именно:

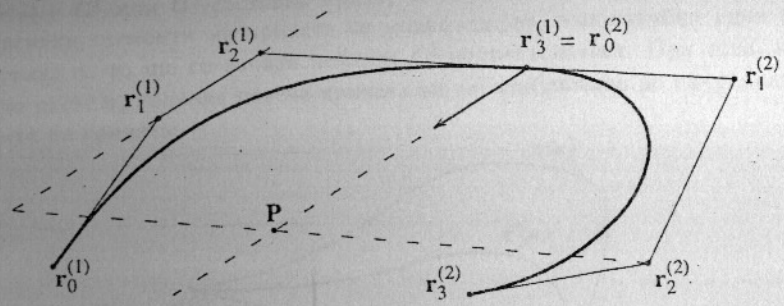
$$\mathbf{r}_1^{(2)} = -\lambda\mathbf{r}_2^{(1)}+(\lambda+1)\mathbf{r}_3^{(1)}$$

$$\mathbf{r}_2^{(2)} = \lambda^2\left(\mathbf{r}_1^{(1)} - \mathbf{r}_2^{(1)}\right) + (\lambda+\mu)\left(\mathbf{r}_3^{(1)} - \mathbf{r}_2^{(1)}\right) + \mathbf{r}_3^{(1)}$$

Тези две уравнения имат следния геометричен смисъл. За да има непрекъснатост на кривината е необходимо:

1. Точките $\mathbf{r}_2^{(1)}, \mathbf{r}_3^{(1)} = \mathbf{r}_0^{(2)}$ и $\mathbf{r}_1^{(2)}$ да са колинеарни;
2. Точката $\mathbf{r}_2^{(2)}$ да лежи на правата, носеща вектора $\mathbf{r}_3^{(1)} - \mathbf{r}_2^{(1)}$ и минаваща през точката $\mathbf{P} = \mathbf{r}_3^{(1)} + \lambda^2\left(\mathbf{r}_1^{(1)} - \mathbf{r}_2^{(1)}\right)$.

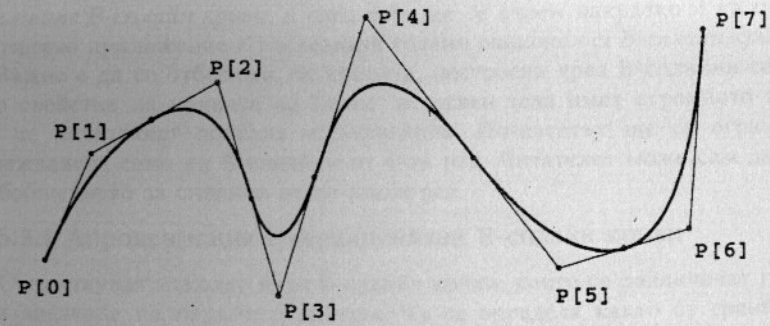
От дефиницията се вижда, че точките $\mathbf{r}_2^{(2)}$ и $\mathbf{P}$ трябва да лежат в тази полуравнина спрямо правата, определена от $\mathbf{r}_2^{(1)}, \mathbf{r}_3^{(1)} = \mathbf{r}_0^{(2)}$ и $\mathbf{r}_1^{(2)}$, в която лежи и $\mathbf{r}_1^{(1)}$.



Фиг. 6-16

Сега можем да построим съставната крива, започвайки от първия сегмент, като всеки следващ се задава с крайните си точки и въведените два скалара. Коефициентите λ и μ са степените на свобода на всеки сегмент от кривата. Точката $\mathbf{r}_3^{(2)}$ може да се избере произволно.

СЪСТАВНИ КРИВИ ОТ СЕГМЕНТИ НА БЕЗИЕ-БЕРНЩАЙН. Най-често използваните криви от този вид са тези от втора степен. За тях е удачно да се приложи механизмът на неявно задаване на възлите, представен в предишния параграф.



Фиг. 6-17

Като считаме, че не само наклоните, но и големините на допирателните в точките на съединяване са непрекъснати, ще задаваме само тези характеристични точки, през които кривата не минава. Крайните точки на сегментите могат да се определят, като се намерят средните точки на всяка страна, т.е. кривата минава винаги през средите на отсечките на начупената, с която тя се определя. Тук вече няма ограничение за четността на броя на въведените точки. Следната функция визуализира крива, зададена по този начин:

```
void DrawC1BezierCurve2(P,n,step)
Point P[]; int n;
float step;
{int i; Point Q[3];
    P[n+1].x=P[n].x;
    P[n+1].y=P[n].y;
    Q[0].x  =P[0].x;
    Q[0].y  =P[0].y;
    MoveTo(P[0].x,P[0].y);
    for (i=1; i<=n; i++) {
        Q[1].x=P[i].x;
        Q[1].y=P[i].y;
        Q[2].x=(P[i].x+P[i+1].x)/2;
        Q[2].y=(P[i].y+P[i+1].y)/2;
        DrawBezierSegment2(Q,0,step);
        Q[0].x=Q[2].x; Q[0].y=Q[2].y;
    }
    DrawTo(P[n].x,P[n].y);
}
```

В практиката една съставна крива на Безие често се изгражда от сегменти, всеки от които е от различна степен. В системата EUCLID например се използват съставни криви на Безие до 9-та степен.

6.2.3 Локална модификация на съставни криви

Едно от неудобствата на използването на криви в моделирането е, че промяната в една нейна част води до промяна и на останалите ѝ части. В практиката често се случва да се генерира форма, която удовлетворява изискванията на проектанта с изключение само на определен фрагмент от нея. Тогава е необходимо такова представяне, което би позволило промяна на формата само в желаната част, като кривата остане същата в останалите участъци. При това новата крива трябва да удовлетворява същите изисквания като първичната - за непрекъснатост, определена кривина и т.н. Представянето на съставните криви дава възможност при подходящи условия модификацията да бъде само локална.

Тук ще разгледаме един от методите, които позволяват това за G^1 и G^2-съставни криви, който е предложен от Кунс (Coons) през 1977 год. Освен метода на Кунс, представянето на сплайните чрез база от B-сплайни също осигурява локалност, както ще видим по-късно. Кунс разглежда полинома:

$$\mathbf{q}(u) = (6u^5 - 15u^4 + 10u^3)[\mathbf{r}(1) - \mathbf{r}(0)] + \mathbf{r}(0), \qquad 0 \le u \le 1$$

за който са изпълнени съотношенията:

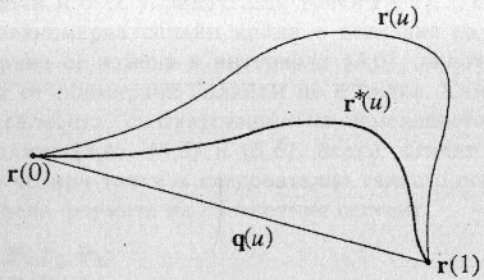
$\mathbf{q}(0) = \mathbf{r}(0), \quad \dot{\mathbf{q}}(0) = 0, \quad \ddot{\mathbf{q}}(0) = 0,$

$\mathbf{q}(1) = \mathbf{r}(1), \quad \dot{\mathbf{q}}(1) = 0, \quad \ddot{\mathbf{q}}(1) = 0.$

Това е уравнение на отсечка с краища $\mathbf{r}(0)$ и $\mathbf{r}(1)$, тъй като полиномът $(6u^5-15u^4+10u^3)$ е монотонно растяща функция в интервала $[0,1]$. Ако $\mathbf{r}(u)$ е някой сегмент от кривата, а α е произволен скалар, следният сегмент има същите крайни точки като $\mathbf{r}(u)$:

$$\mathbf{r}^*(u) = (1-\alpha)\mathbf{r}(u)+\alpha\mathbf{q}(u)$$

Векторите на първата и втората производни на $\mathbf{r}(u)$ и $\mathbf{r}^*(u)$ в краищата съвпадат по посока, а големините им се различават с $(1-\alpha)$. Следователно, ако $\mathbf{r}(u)$ е G^1 или G^2-съставна крива, то $\mathbf{r}^*(u)$ ще запази това свойство. Ако пък всички сегменти на кривата се модифицират, използвайки една и съща константа α, то ще се запази нейната C^2-непрекъснатост. При това, колкото повече расте α, толкова повече кривата ще се приближава до начупената през възлите на кривата.



Фиг. 6-18

6.3 АПРОКСИМАЦИЯ С B-СПЛАЙН КРИВИ

В този параграф ще разгледаме апроксимацията на точки чрез параметрични криви, построени на базата на дефинираните в 6.1.5 B-сплайни. В същия параграф показахме, че всеки сплайн може да се представи като линейна комбинация на B-сплайни. Ще използваме този резултат, за да въведем *нерационалните B-сплайн криви*, а след това ще се спрем накратко и на придобилите голямо приложение в последните години *рационални B-сплайн криви*.

Важно е да се отбележи, че кривите, построени чрез B-сплайни са подобни по свойства на кривите на Безие, но освен това имат огромното предимство, че позволяват локална модификация. По-нататък ще се ограничим с разглеждането само на B-сплайни от 4-ти ред. Читателят може сам да извършши обобщението за сплайни от по-висок ред.

6.3.1 Апроксимация с нерационални B-сплайн криви

Съществуват няколко вида B-сплайн криви, които се различават по начина на задаване на параметризацията. Тя се определя както от границите, в които се мени параметърът, така и от стойностите на неговите възли. Да при-

помним, че възлите са ненамаляваща редица, върху която се дефинират самите B-сплайни. При зададени възли:

$$u_0 \le u_1 \le u_2 \le \cdots \le u_n$$

нормираният B-сплайн $N_{4,i}(u)$ е ненулев само в интервала (u_{i-4}, u_i). Нека освен това да допуснем съществуването на кратни възли, като приемем, че във формулите на Кокс-Де Бур е изпълнено $0/0 = 0$. По този начин, дори и при повтарящи се възли ще можем да използваме тези формули.

Възлите биват три вида и във връзка с това сплайните, които се получават при апроксимацията, могат да се класифицират по следния начин:

- *периодични равномерни*
- *непериодични равномерни*
- *неравномерни*.

ПЕРИОДИЧНИ СПЛАЙНИ. Равномерно периодично е такова множество от възли, дължините на интервалите между съседните от които са равни, например:

$\mathbf{k} = [\ 0, 1, 2, 3, \ldots, n\ ]$
$\mathbf{k} = [\ 0, 0.1, 0.2, 0.3, \ldots, 1\ ]$
$\mathbf{k} = [\ -5/2, -3/2, -1/2, \ldots, n/2\ ]$ и др.

Нека е дадено едно равномерно множество от $n+1$ възела, разстоянието между съседните от които е 1, т.е. $u_i = i$, $i = 0,1,\ldots,n$. Върху това множество да разгледаме и нормираните B-сплайни от четвърти ред $N_{4,i}(u)$. Броят на тези сплайни е $n-3$, тъй като $N_{4,i}(u)$ е ненулев само за $i-4 < u < i$, т.е. B-сплайните $N_{4,0}(u)$, $N_{4,1}(u)$, $N_{4,2}(u)$ и $N_{4,3}(u)$ не са дефинирани върху даденото множество от възли. Следователно, върху избраното множество от възли са определени само $N_{4,4}(u)$, $N_{4,5}(u)$, ..., $N_{4,n}(u)$.

Нека освен това знаем и $n-3$ векторни коефициента $\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{n-4}$, които са радиус-векторите на точките $\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_{n-4}$. Линейната комбинация на нормираните B-сплайни с тези векторни коефициенти ще представя един параметричен сплайн, а неговият параметър ще се мени само в границите $3 \le u \le n-3$.

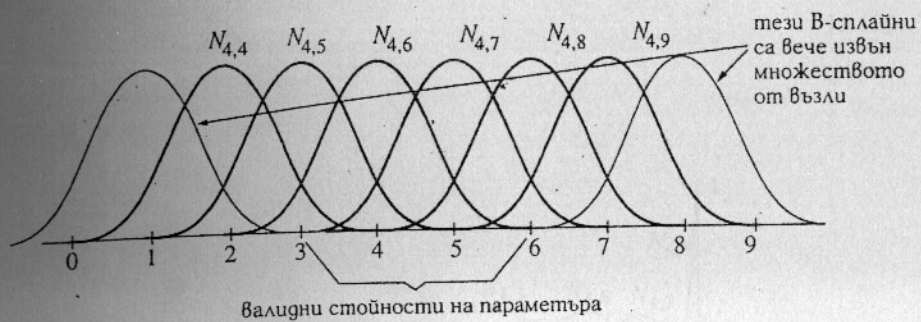
$$\mathbf{r}(u) = \sum_{i=0}^{n-4} \mathbf{r}_i N_{4,i+4}(u) \qquad 3 \le u \le n-3$$

Полученият сплайн носи името *периодична равномерна B-сплайн крива* или *PUNRBS-крива* (от Periodic Uniform Non-Rational B-Spline Curve), която апроксимира множеството от характеристични точки $\mathbf{P}_i$. Тя не минава през нито една от тези точки, а те по-скоро играят роля, подобна на управляващата начупена при кривите на Безие.

Да вземем един конкретен пример. Нека броят на възлите е 10, т.е. *възловият вектор* е:

$\mathbf{k} = [\ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\ ]$.

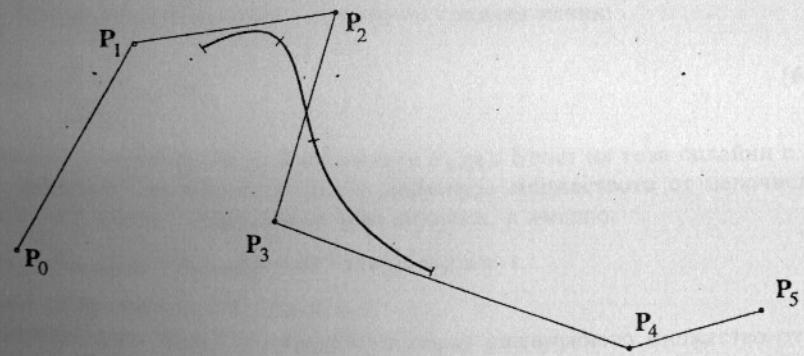
Върху тези възли е дефинирано множеството от 6 нормирани B-сплайни, показано на фиг. 6-19.



Фиг. 6-19

Нека са дадени и 6-те управляващи точки $\mathbf{P}_0$, $\mathbf{P}_1$,..., $\mathbf{P}_5$. Апроксимиращата ги периодична равномерна сплайн крива е показана на фиг. 6-20. Параметърът на тази крива се изменя в интервала [3,6], защото извън него линейната комбинация от нормирани сплайни не е пълна. Самата крива се състои от три отделни сегмента, съответстващи на изменението на параметъра във всеки от интервалите [3,4), [4,5) и [5,6]. Всеки сегмент е претеглената комбинация на само четири точки и следователно тяхното положение е единственото, което определя формата на съответния сегмент:

в [3,4] - $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$;
в [4,5] - $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3, \mathbf{P}_4$;
в [5,6] - $\mathbf{P}_2, \mathbf{P}_3, \mathbf{P}_4, \mathbf{P}_5$



Фиг. 6-20

Можем да пресметнем всеки от B-сплайните $N_{4,i}(u)$ директно. Тъй като възлите са равномерни, всички B-сплайни са еднакви и са само отместени на цели интервали един от друг (фиг. 6-19). Ако приложим формулите на Кокс-Де Бур, явният вид на всеки от тях ще се задава с:

$$N_{4,i+4}(u)=\begin{cases}\frac{1}{6}(u-i)^3 & i\le u<i+1\\ \frac{1}{6}\left[1+3(u-i)+3(u-i)^2-3(u-i)^3\right] & i+1\le u<i+2\\ \frac{1}{6}\left[4-6(u-i)^2+3(u-i)^3\right] & i+2\le u<i+3\\ \frac{1}{6}\left[1-3(u-i)+3(u-i)^2-(u-i)^3\right] & i+3\le u\le i+4\end{cases}$$

Тогава всеки сегмент от кривата (между всеки два възела i и i+1) може да се представи еднообразно спрямо параметъра $(u-i)$. Аналогично на матричния запис на сегментите на Фъргюсън и Безие, можем лесно да изведем съответното уравнение и за B-сплайн крива:

$$\mathbf{r}(\overline{\mathbf{U}})=\begin{bmatrix}1 & u-i & (u-i)^2 & (u-i)^3\end{bmatrix}\cdot\frac{1}{6}\begin{bmatrix}1 & 4 & 1 & 0\\ -3 & 0 & 3 & 0\\ 3 & -6 & 3 & 0\\ -1 & 3 & -3 & 1\end{bmatrix}\cdot\begin{bmatrix}\mathbf{r}_i\\ \mathbf{r}_{i+1}\\ \mathbf{r}_{i+2}\\ \mathbf{r}_{i+3}\end{bmatrix},\quad i=\lfloor u\rfloor,\quad [6.18]$$

където с i отбелязваме цялата част на параметъра u, който се мени в интервала $[3,n-3]$.

Периодичните равномерни сплайни са неудобни поради това, че не започват и не завършват в крайните си характеристични точки. Това може да се преодолее, като се повиши кратността на крайните характеристични точки, но е по-удобно да се промени множеството от възли, като се въведе кратност при тях. Така се получава непериодично множество от възли, което се използва много по-често. Ние ще продължим разглеждането на тези именно сплайни, като отбележим, че много от техните свойства са общи свойства на всички равномерни B-сплайн криви: както периодични, така и непериодични.

НЕПЕРИОДИЧНИ СПЛАЙНИ. Малко по-горе допуснахме възможността да съществуват кратни възли, приемайки, че $0/0=0$ във формулите на Кокс-Де Бур. Естествено, при кратни възли се намалява гладкостта на сплайна в тях. Кубичният сплайн има непрекъснати втори производни във всеки прост (некратен) възел, но има прекъсната втора производна във възел, който се повтаря (двукратен възел), прекъсната първа производна в трикратен и самият сплайн е прекъснат в четирикратен.

Да приемем, че нормираните B-сплайни $N_{4,i}(u)$ са дефинирани върху множеството от възли $u_i=i$, $i=0,1,...,n$. Нека да разширим това множеството с по 3 във всеки от двата края, като просто направим крайните възли четирикратни:

$u_{-3}=u_{-2}=u_{-1}=u_0=0$

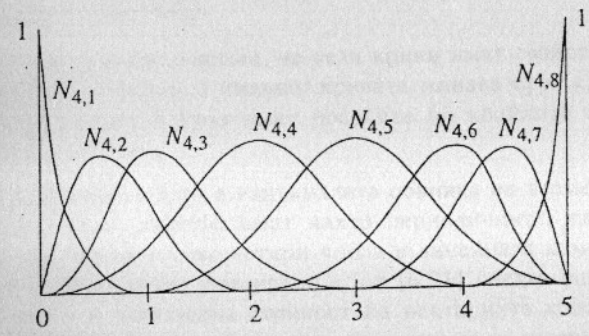
$u_n=u_{n+1}=u_{n+2}=u_{n+3}=n$.

На фиг. 6-21 са показани нормираните B-сплайни върху такова множество от възли при $n=5$. Ясно си личи прекъснатостта на производните и на крайните сплайни в краищата на интервала. Следователно върху множество от n+1 некратни възли (от 0 до n), или общо n+7 заедно с добавените в двата

края, има определени n+3 B-сплайна. В общия случай за B-сплайни от произволен ред са валидни следните равенства:

$$n_{\text{възли}} = n_{\text{некратни възли}}+2(m-1)$$
$$n_{\text{сплайни}} = n_{\text{възли}}-m, \qquad [6.19]$$
$$n_{\text{некратни възли}} = n_{\text{сплайни}}-m+2$$

където m е редът на сплайните.



Фиг. 6-21

По-горе бе казано, че всеки сплайн може да се представи като линейна комбинация на B-сплайни. Въвеждането на кратни възли оставя в сила този резултат. Ще използваме това за да зададем параметрично (чрез векторно уравнение) сплайн-крива, апроксимираща дадена начупена както при метода на Безие.

Нека са дадени n+1 векторни коефициента $\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_n$ (както и преди ще считаме, че това са радиус-векторите на n+1 точки). Като използваме [6.6] можем да представим кривата векторно по следния начин:

$$\mathbf{r}(u) = \sum_{i=0}^{n} \mathbf{r}_i N_{4,i+1}(u) \qquad [6.20]$$

като линейна комбинация на B-сплайните $N_{4,i}(u)$. Броят на тези сплайни е n+1, което съобразно зависимостта [6.19] дефинира множеството от целочислени възли, върху които са определени тези сплайни, а именно:

$$n_{\text{некратни възли}} = n_{\text{сплайни}}-m+2 = (n+1)-4+2 = n-1,$$

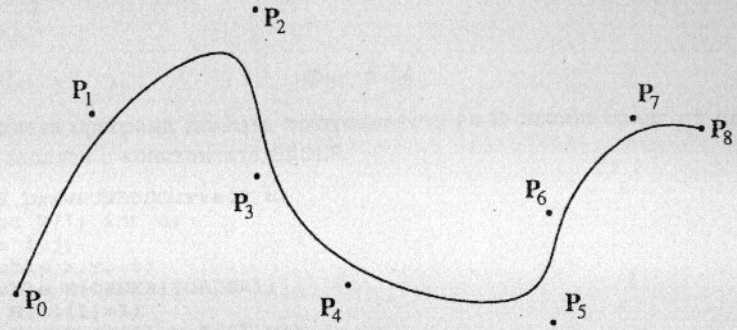
т.е. това са възлите: 0, 1, 2, ..., n–3, n–2.

Следователно $N_{4,i}(u)$ са определени върху разширеното множество от целочислени възли 0,0,0,0,1,2,...,n–3, n–2, n–2, n–2, n–2. При $0 \le u \le n-2$ уравнение [6.20] задава т.нар. *непериодична равномерна B-сплайн крива* или накратко *NPUNRBS-крива* (от Non–Periodic Uniform Non–Rational B–Spline Curve). Това съкращение показва, че е използвана линейна комбинация (нерационална) от B-сплайни върху равномерно множество от възли, като крайните възли имат

кратност, съответстваща на реда на B-сплайна. Някои автори наричат тези криви още *отворени равномерни сплайни*.

Кривата, получена върху такова множество от възли, апроксимира начупената, зададена с точките $\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_n$ с радиус-вектори $\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_n$, както това е показано на фиг 6-22. За разлика от предишния вид, тя започва в точката $\mathbf{P}_0$ и завършва в т. $\mathbf{P}_n$.

Възловият вектор тук е $\mathbf{k}$ = [0,0,0,0,1,2,...,n–3, n–2, n–2, n–2, n–2].



Фиг. 6-22

Да разгледаме някои свойства на *непериодичните равномерни сплайни*. Те могат да се обобщят и за други B-сплайн-криви, а не само за кубични.

Свойство 1. Кривата апроксимира характеристичната си начупена подобно на кривата на Безие. Сравнително лесно може да се покаже, че:

$$\sum_{i=0}^{n} N_{4,i+1}(u) = 1, \qquad 0 \le u \le n-2 \qquad [6.21]$$

Ако положим u=0, от [6.20] ще получим:

$$\mathbf{r}(0) = \sum_{i=0}^{n} \mathbf{r}_i N_{4,i+1}(0) = \mathbf{r}_0,$$

защото, както е показано на предната фигура (фиг. 6-21), само $N_{4,1}(u)$ е различен от 0 при u = 0 и $N_{4,1}(0)$ = 1. Аналогично, $\mathbf{r}(n) = \mathbf{r}_n$. Нека да разгледаме производната в т.0:

$$\dot{\mathbf{r}}(0) = \sum_{i=0}^{n} \mathbf{r}_i \dot{N}_{4,i+1}(0)$$

Ако диференцираме [6.21] ще получим:

$$\sum_{i=0}^{n} \mathbf{r}_i \dot{N}_{4,i+1}(u) = 0 \qquad 0 \le u \le n-2$$

и тъй като в тази сума при $u = 0$ само $\dot{N}_{4,1}(0)$ и $\dot{N}_{4,2}(0)$ са различни от нула, ще получим, че $\dot{N}_{4,1}(0) = -\dot{N}_{4,2}(0)$. От това пък следва, че:

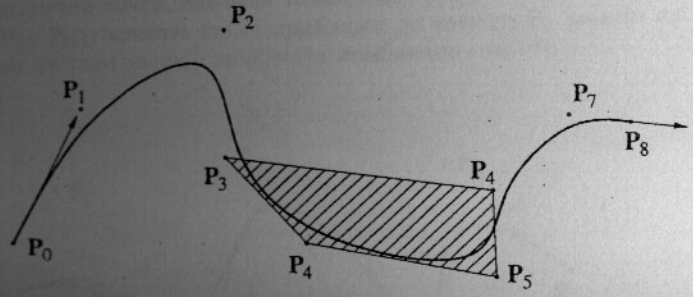
$$\dot{\mathbf{r}}(0) = \dot{N}_{4,2}(0)(\mathbf{r}_1 - \mathbf{r}_0).$$

По аналогичен начин можем да получим формулата за производната и в другия край:

$$\dot{\mathbf{r}}(n) = \dot{N}_{4,n+1}(n-2)(\mathbf{r}_n - \mathbf{r}_{n-1}).$$

Това, което получихме показва, че тези криви имат свойства, аналогични на тези на кривите на Безие, а именно: кривата минава през крайните точки, а допирателните в двата й края имат посоките на крайните отсечки от характеристичната начупена.

Свойство 2. Кривата лежи в изпъкналата обвивка на всеки четири последователни точки. Това свойство имат както периодичните, така и непериодичните сплайни. Тъй като само четири члена в линейната комбинация [6.19] са различни от нула за всяко фиксирано u, от [6.21] следва още, че за това u точката от кривата е претеглена стойност на векторните коефициенти пред ненулевите B-сплайни. Можем да считаме, че кривата се състои от множество сегменти $\mathbf{Q}_i$, всеки от които съответства на изменението на параметъра между два съседни възела $i-1 \le u < i$. Във всеки такъв сегмент формата на кривата се управлява от четирите характеристични точки, които имат ненулеви коефициенти в линейната комбинация. Изпъкналата обвивка на тези именно четири точки съдържа изцяло съответния сегмент.



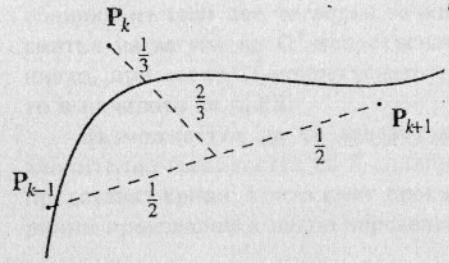
Фиг. 6-23

Като следствие от това свойство се вижда, че ако някои четири точки са колинеарни, то и кривата в този участък се изражда в отсечка.

Свойство 3. Кривата минава близко до средните точки на всяка страна на начупената с изключение на крайните две.

Свойство 4. При $k = 2,3,\ldots,n-2$ кривата минава през точките, които лежат на една трета разстояние от $\mathbf{r}_k$ по правата, съединяваща я със средата на отсечката между $\mathbf{r}_{k-1}$ и $\mathbf{r}_{k+1}$:

$$\frac{1}{6}\mathbf{r}_{k-1} + \frac{2}{3}\mathbf{r}_k + \frac{1}{6}\mathbf{r}_{k+1} = \frac{2}{3}\mathbf{r}_k + \frac{1}{3}\left[\frac{1}{3}\left(\mathbf{r}_{k-1} - \mathbf{r}_{k+1}\right)\right]$$



Фиг. 6-24

Следната програма показва построяването на B-сплайн крива от произволен ред, зададен с константата ORDER.

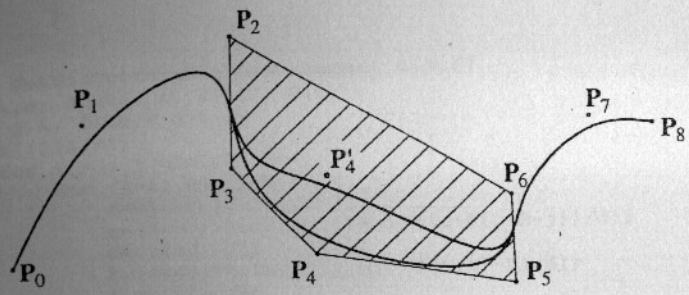
```
void DrawNUNRBSCurve(P,n)
Point P[]; int n;
{int i,j;
 double x,y, t;
 double N[ORDER][ORDER];
    N[1][1]=1;
    MoveTo(P[0].x,P[0].y);
    for (t=INC; t<n+2-ORDER; t+=INC) {
        j=(int)t;
        GetBSplineValues(j,t,N,n);
        for (i=0, x=y=0; i<ORDER; i++) {
            x+=P[j+i].x*N[ORDER][i+1];
            y+=P[j+i].y*N[ORDER][i+1];
        }
        DrawTo(x,y);
    }
    DrawTo(P[n].x,P[n].y);
}

void GetBSplineValues(i,t,N,n)
int i,n; double t, N[][];
{int j,k,ilow,ihigh;
 double a,b;
    for (j=2; j<=ORDER; j++)
        for (k=1; k<=j; k++) {
            ilow =(i+k-j<0)?0:(i+k-j);
            ihigh=(i+k-1>n-2)?n-2:(i+k-j);
            a=(ihigh<=ilow)?0:
              (N[j-1][k-1]*(t-ilow)/(ihigh-ilow));
            ilow =(i+k-j+1<0)?0:(i+k-j+1);
            ihigh=(i+k>n-2)?n-2:(i+k);
            b=(ihigh<=ilow)?0:
              (N[j-1][k]*(ihigh-t)/(ihigh-ilow));
            N[j][k] = a+b;
        }
}
```

В горната програма използваме функцията GetBSplineValues, за да получим стойността на сплайните $N_{m,i}(u)$, $N_{m,i+1}(u)$,..., $N_{m,i+m}(u)$ в точката t. За тази цел прилагаме директно формулите на Кокс-Де Бур, като вземем пред вид кратността на крайните възли.

Да отбележим, че характеристичните точки трябва да са не по-малко от реда на сплайна, иначе кривата ще се превърне в отсечка, свързваща първата и последната точки.

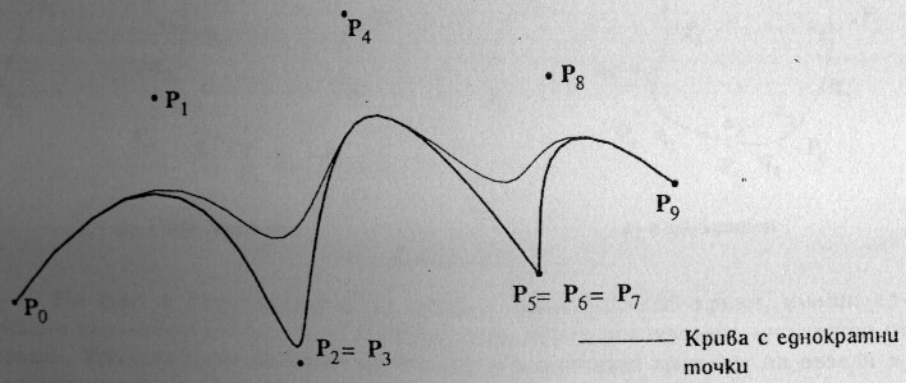
Както казахме и в началото на този параграф, най-важното свойство на кривите, построени с помощта на B-сплайни, е възможността им за локална модификация. Това позволява да се променя само един сегмент без изменения му да влияят на останалата част от кривата.

Това следва от факта, че самите B-сплайни са само частично ненулеви и промяната на един коефициент в линейната комбинация (еквивалентно на преместване на една от апроксимираните точки) ще промени формата на кривата само в ограничен участък около преместената точка.



Фиг. 6-25

B-сплайн кривите могат да минават и по-близко до избрани междинни характеристични точки, ако тези точки имат кратност, различна от 1. Например ако $\mathbf{P}_i = \mathbf{P}_{i+1}$, кривата ще се приближи до точката $\mathbf{P}_i$, защото ще се увеличи теглото на тази точка в линейната комбинация [6.19].



Фиг. 6-26

Ако за една кубична B-сплайн крива кратността на една характеристична точка е 3 ($\mathbf{P}_i = \mathbf{P}_{i+1} = \mathbf{P}_{i+2}$), кривата ще интерполира тази точка и дори ще се изроди в отсечка в двата сегмента, които се управляват от точките $\mathbf{P}_{i-1}$, $\mathbf{P}_i$, $\mathbf{P}_{i+1}$, $\mathbf{P}_{i+2}$ и $\mathbf{P}_i$, $\mathbf{P}_{i+1}$, $\mathbf{P}_{i+2}$, $\mathbf{P}_{i+3}$ съответно. Причината за това е, че изпъкналите обвивки на тези две четворки точки са отсечки. Интерполацията ще стане за сметка на загуба на G^1-непрекъснатостта в кратната точка. Това е изключението, при което C^1-непрекъснатост не поражда G^1-непрекъснатост, споменато в началото на 6.2.2.

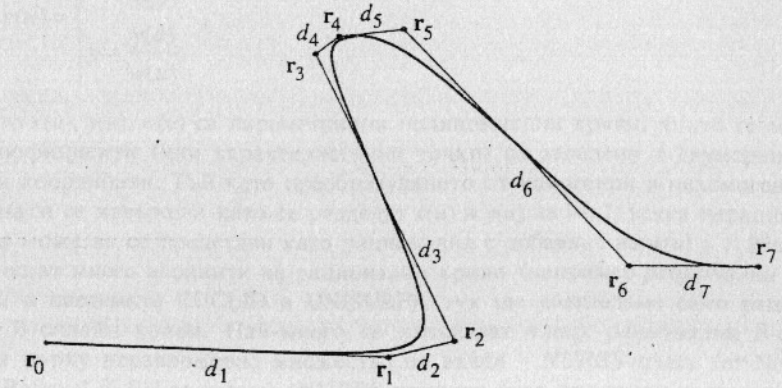
Възможността да се задават кратни характеристични точки увеличава значително гъвкавостта на B-сплайн кривите. Това позволява еднотипно да се представят криви, които имат прекъснати както втори, така и първи параметрични производни в някои определени точки.

НЕРАВНОМЕРНИ СПЛАЙНИ. Когато върховете на характеристичната начупена са разположени неравномерно, по-добре е да се използва неравномерна параметризация. Един често използван метод е да се избере за параметър натрупаното разстояние между отделните характеристични точки.

Възловият вектор в такъв случай ще бъде

$\mathbf{k}=[0,0,0,0, t_1, t_2,..., t_{n-3}, t_{n-2}, t_{n-2}, t_{n-2}, t_{n-2}]$, където

$$t_i = \sum_{k=1}^{i} d_k, \qquad d_k = |\mathbf{r}_k - \mathbf{r}_{k-1}|$$



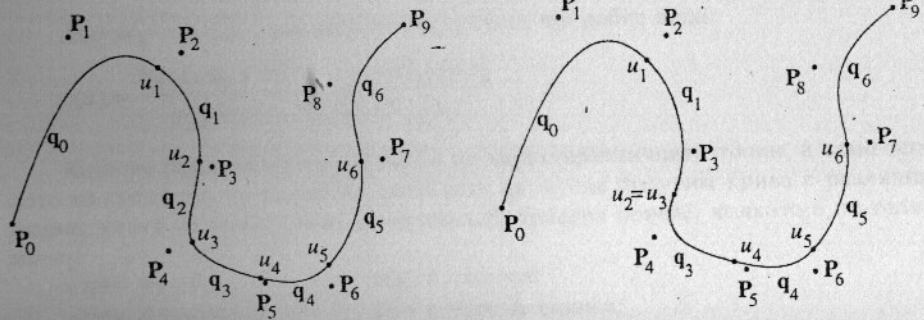
Фиг. 6-27

Полученото множество от възли вече не е равномерно, затова и сплайните се наричат *неравномерни B-сплайн криви* или само *NUNRBS-криви* (от Non-Uniform Non-Rational B-Spline Curves).

Разгледаните дотук равномерни криви нямат повтарящи се възли, освен първия и последния. При неравномерните сплайни изискването е последователността от възли да е само ненамаляваща, което позволява някои от възлите да са кратни, като кратността на един възел не надвишава реда на сплайна. Подобно на повтарящите се характеристични точки може да се постигне приближаване и дори интерполиране на характеристични точки. Естествено, в този случай е необходимо програмите да се напишат така, че да се отчита възможността за различно разстояние между възлите, както и за евентуално

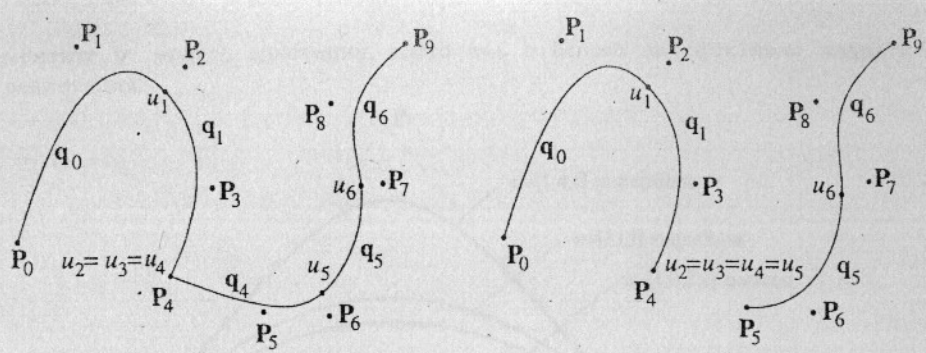
прекъсване на кривата. Един от начините да се направи това е показан с програмата по-долу. Масивът Knots съдържа разширеното множество от възли - с по четири еднакви възела в двата края.

```
void DrawGeneralNUNRBSCurve(P,n,Knots)
Point P[]; int n; double Knots[];
{int i,j; double x,y, t; double N[ORDER][ORDER];
    for (j=0; j<n+4; j++)
        for (t=Knots[j]; t<Knots[j+1]+INC/2; t+=INC) {
            GetNonUniformBSplineValues(j,t,N,Knots);
            for (i=0, x=y=0; i<ORDER; i++) {
                x+=P[j+i].x*N[ORDER][i+1];
                y+=P[j+i].y*N[ORDER][i+1];
            }
            if (i==Knots[j]) MoveTo(x,y); else  DrawTo(x,y);
        }
}

void GetNonUniformBSplineValues(i,t,N,U)
int i,n; double t, N[][], U[];
{int j,k; double a,b,d;
    N[1][1]=1;
    for (j=2; j<=ORDER; j++)
        for (k=1; k<=j; k++) {
            d=U[i+k-1]-U[i+k-j];
            a=(d<=0)?0:(N[j-1][k-1]*(t-U[i+k-j])/d);
            d=U[i+k]-U[i+k-j+1];
            b=(d<=0)?0:(N[j-1][k]*(U[i+k]-t)/d);
            N[j][k]=a+b;
        }
}
```



Фиг. 6-28а

На фиг. 6-28а,б са показани няколко общи NUNRBS–криви, имащи еднакви характеристични точки и дефинирани върху целочислено множество от възли. Единственото различие между тях е различната кратност на възела u_2 за всяка една от тях. Очевидно е изменението на повеdението на кривата около характеристичните точки P_3, P_4, P_5. За по-голяма яснота са отбелязани местата както на всички възли и характеристични точки на кривите, така и всички отделни сегменти (с q).



Фиг. 6-28б

6.3.2 Апроксимация с рационални B-сплайни

Рационалните криви са често използвано средство в геометричното моделиране. Една рационална крива в равнината се дефинира по следния начин:

$$\mathbf{r}(u) = \begin{cases} x = \dfrac{x(u)}{w(u)} \\ y = \dfrac{y(u)}{w(u)} \end{cases},$$

където $x(u)$, $y(u)$, $w(u)$ са параметрични полиномиални криви, чиито геометрични коефициенти (или характеристични точки) са зададени в двумерни хомогенни координати. Тъй като преобразуването от хомогенни в нехомогенни координати се извършва като се разделят $x(u)$ и $y(u)$ на $w(u)$, всяка нерационална крива може да се представи като рационална с добавяне на $w(u) = 1$. Макар че се срещат много варианти на рационални криви (например *рационални Безие-криви* в системите EUCLID и UNISURF), тук ще разгледаме само рационалните B-сплайн криви. Най-много се използват т.нар. *рационални B-сплайн криви* върху неравномерно множество от възли - *NURBS-криви* (от Non–Uniform Rational B–Spline curves). NURBS-кривите бяха включени в стандарта за обмен на графична информация IGES още през 1983 год., а в системата PHIGS това е и единственият начин за моделиране на криви. В много съвременни графични дисплеи процедурите за визуализация на NURBS-криви са реализирани апаратно.

Една рационална B-сплайн крива се задава в нехомогенни координати с:

$$\mathbf{r}(u) = \frac{\sum_{i=0}^{n} w_i N_{m,i}(u)\mathbf{r}_i}{\sum_{k=0}^{n} w_k N_{m,k}(u)},$$

където $\mathbf{r}_i$ са характеристичните точки, $N_{m,i}(u)$ са нормираните базисни B-сплайн функции, а w_k са теглата за всяка от характеристичните точки. Това са всъщност последните координати в хомогенното представяне на тези точки. Да отбележим, че ако всички тегла са равни на 1, ще получим нерационална крива [6.20], защото знаменателят ще е също 1 (използвайки [6.21]).

Едно от предимствата на рационалните криви е, че те са инвариантни по отношение на перспективна трансформация в пространството. Тази особеност ще стане ясна при разглеждане на матричното представяне на перспективна проекция в хомогенни координати.

Друго предимство на използването на рационални криви е, че те включват всички възможни частично-полиномиални криви. В частност, квадратичните рационални криви интерполират точно конични сечения. Това означава, че с рационални криви е възможно точното представяне на окръжности, елипси, хиперболи и параболи, които са често използвани в компютърната графика и особено в геометричното моделиране. За целта при задаването на една рационална крива е необходимо да се определи не само положението на характеристичните ѝ точки, а също така и техните тегла (техните последни хомогенни координати).

По-долу е показан прост пример на използване на механизма на теглата на характеристичните точки в една квадратична NURBS-крива, определена върху възловия вектор [0 0 0 1 1 1]:

$$\mathbf{r}(u)=\frac{w_0N_{3,0}(u)\mathbf{r}_0+w_1N_{3,1}(u)\mathbf{r}_1+w_2N_{3,2}(u)\mathbf{r}_2}{w_0N_{3,0}(u)+w_1N_{3,1}(u)+w_2N_{3,2}(u)}$$

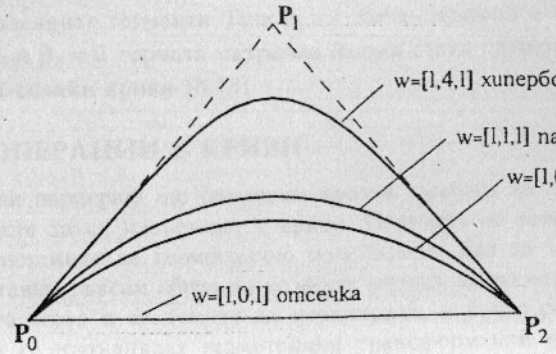
Ако $w_0=w_2=1$, уравнението на кривата ще добие вида:

$$\mathbf{r}(u)=\frac{N_{3,0}(u)\mathbf{r}_0+w_1N_{3,1}(u)\mathbf{r}_1+N_{3,2}(u)\mathbf{r}_2}{N_{3,0}(u)+w_1N_{3,1}(u)+N_{3,2}(u)}$$

Като не променяме положението на характеристичните точки, а само теглото на средната, за различни стойности на w_1 ще получим крива с различна форма, която се приближава до точката $\mathbf{P}_1$ толкова повече, колкото е по-голямо w_1:

- при $w_1=0$ $\mathbf{r}(u)$ е отсечка;
- при $0<w_1<1$ $\mathbf{r}(u)$ е част от елипса;
- при $w_1=1$ $\mathbf{r}(u)$ е част от парабола;
- при $w_1>1$ $\mathbf{r}(u)$ е част от хипербола;

Голямото значение на NURBS–кривите се определя най-вече от възможността чрез една форма да се представят разнообразни семейства от криви: конични сечения, криви на Безие, нерационални B-сплайн криви и др. В последните години бяха разработени добре математически методи за осъществяване на най-важните операции с тях: пресичане, сливане, смяна на параметризацията и т.н. Горното позволява значително опростяване на един контурен (или граничноопределен) геометричен модел, а именно - представяне на всички елементи в него като NURBS–криви. Това прави всички операции с елементите от модела еднотипни, което пък е основа за ефективна апаратна реализация.



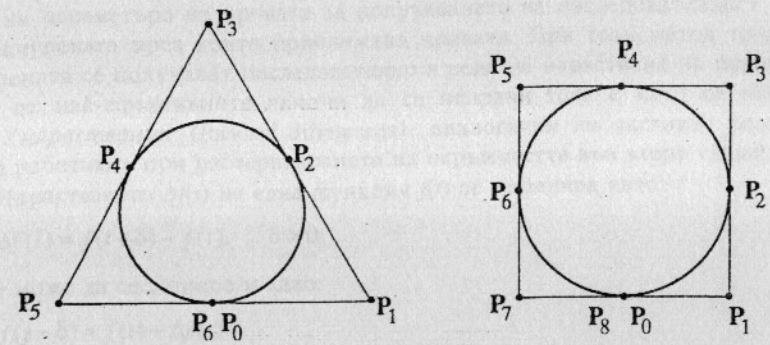
Фиг. 6-29

Тук ще покажем два от начините за точно представяне на окръжност като квадратична рационална B-сплайн крива върху неравномерно множество от възли. На фиг. 6-30а за характеристични точки са използвани върховете на многоъгълника $\mathbf{P}_0\mathbf{P}_1\mathbf{P}_2\mathbf{P}_3\mathbf{P}_4\mathbf{P}_5\mathbf{P}_6=\mathbf{P}_0$ с тегла и възлов вектор съответно:

$\mathbf{w}=[\,1\ \ 1/2\ \ 1\ \ 1/2\ \ 1\ \ 1/2\ \ 1\,]$
$\mathbf{k}=[\,0\ 0\ 0\ 1\ 1\ 2\ 2\ 3\ 3\ 3\,]$,

а на фиг. 6-30б характеристичните точки са $\mathbf{P}_0\mathbf{P}_1\mathbf{P}_2\mathbf{P}_3\mathbf{P}_4\mathbf{P}_5\mathbf{P}_6\mathbf{P}_7\mathbf{P}_8=\mathbf{P}_0$ със следните тегла и възли:

$\mathbf{w}=[\,1\ \sqrt{2}/2\ \ 1\ \sqrt{2}/2\ \ 1\ \sqrt{2}/2\ \ 1\,]$
$\mathbf{k}=[\,0\ 0\ 0\ 1\ 1\ 2\ 2\ 3\ 3\ 4\ 4\ 4\,]$.



Фиг. 6-30а,б

6.4 ДРУГИ СПЛАЙН ФОРМИ

В тази част ще представим две други сплайн форми, без да се занимаваме с тяхното извеждане. Надяваме се, че представените формули биха могли да послужат за решаването на някои практически задачи.

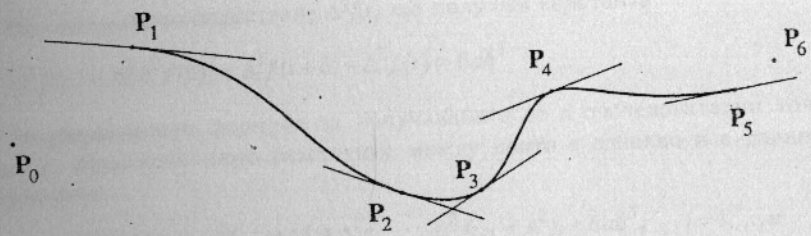
6.4.1 Сплайни на Катмул-Ром

Сплайните на Катмул-Ром (Catmull–Rom) са интерполиращи криви, минаващи през последователност от характеристични точки и имащи определена посока на допирателните в тези точки. Те притежават възможност за локална модификация, но не и свойството на изпъкналата обвивка.

Тези сплайни могат лесно да се зададат в матрична форма подобна на [6.18], която използвахме за PUNRBS–кривите:

$$\mathbf{r}(\overline{u}) = \begin{bmatrix} 1 & u-i & (u-i)^2 & (u-i)^3 \end{bmatrix} \cdot \begin{bmatrix} 0 & 2 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 2 & -4 & 4 & -1 \\ -1 & 3 & -3 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{r}_i \\ \mathbf{r}_{i+1} \\ \mathbf{r}_{i+2} \\ \mathbf{r}_{i+3} \end{bmatrix}, \qquad i = \lfloor u \rfloor$$

При зададени точки $\mathbf{P}_0$, $\mathbf{P}_1$,..., $\mathbf{P}_n$ с радиус-вектори съответно $\mathbf{r}_0$, $\mathbf{r}_1$,..., $\mathbf{r}_n$, сплайнът на Катмул-Ром минава през точките $\mathbf{P}_1$, $\mathbf{P}_2$,..., $\mathbf{P}_{n-1}$, а във всяка точка $\mathbf{P}_i$ допирателната има посоката на вектора $\mathbf{r}_{i+1} - \mathbf{r}_{i-1}$.



Фиг. 6-31

6.4.2 Бета-сплайн форми

Бета-сплайните са обобщение на нерационалните B-сплайн криви, при които се въвеждат два допълнителни параметъра. Тези параметри влияят глобално на кривата. Формата на тези криви е следната:

$$\mathbf{r}(\overline{\mathbf{U}}) = \overline{\mathbf{U}} \cdot \frac{1}{\delta} \begin{bmatrix} 2\beta_1^3 & \beta_2 + 4(\beta_1^2 + \beta_1) & 2 & 0 \\ -6\beta_1^3 & 6(\beta_1^3 + \beta_1) & 6\beta_1 & 0 \\ 6\beta_1^3 & -3(\beta_2 + 2\beta_1^3 + \beta_1^2) & 3(2\beta_1^2 + \beta_2) & 0 \\ -2\beta_1^3 & 2(\beta_2 + \beta_1^3 + \beta_1^2 + \beta_1) & 2(\beta_2 + \beta_1^2 + \beta_1 + 1) & 2 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{r}_i \\ \mathbf{r}_{i+1} \\ \mathbf{r}_{i+2} \\ \mathbf{r}_{i+3} \end{bmatrix},$$

където $\delta = \beta_2 + 2\beta_1^3 + 4\beta_1^2 + 4\beta_1 + 2$. Първият параметър β_1 се нарича *параметър за настройка*, а β_2 - *параметър на "изтегляне"*.

Параметърът β_1 управлява влиянието на положението на характеристичните точки върху допирателния вектор във възлите на кривата. При увеличаване на β_1 кривата се извива така, че допирателните вектори във възлите да се доближават повече до векторите между съответните характеристични точки. При увеличаване на параметъра β_2 сплайнът се доближава повече до отсечките, съединяващи характеристичните му точки. Необходимо е да се отбележи, че тези сплайни са G^2, но само C^0-непрекъснати в точките на съединяване на отделните сегменти. При $\beta_1 = 1$, бета-сплайнът е вече C^1-непрекъснат. При $\beta_1 = 1$ и $\beta_2 = 0$ горната матрична форма става идентична с тази на периодичните B-сплайн криви [6.18].

6.5 ОПЕРАЦИИ С КРИВИ

В този параграф ще направим кратък преглед на операциите, които се налага често да се извършват с криви. Повечето от тези операции са важна част от системите за геометрично моделиране. Без да навлизаме в детайли, ще представим съвсем сбито по-важните методи за тяхното осъществяване.

Много често в системите за моделиране, в които се работи с криви, се налага да се изпълняват геометрични трансформации над различни обекти. Голямото предимство на параметричното представяне е, че геометричните трансформации: транслация, ротация, мащабиране и симетрия могат да се изпълняват директно над последователността от характеристични точки (или в общия случай над геометричния вектор, с който е дефинирана кривата), което води и до съответната трансформация на точките от кривата. Поради тази причина няма да се спираме специално на този вид операции.

6.5.1 Визуализация на криви

Има най-общо казано два начина за визуализация на криви:

- чрез чертане на начупена през последователност от точки върху нея;
- чрез рекурсивното й подразделяне до получаване на сегмент, който се чертае просто.

В предишните примерни програми използвахме първия начин чрез вариране на параметъра на кривата за получаването на последователност от точки, начупената през които приближава кривата. При този метод точките от начупената се получават последователно в реда на нарастване на параметъра. Един от най-ефективните начини да се направи това е като се използват т.нар. *нараствания* (forward differences), аналогични на частните разлики, с които работихме при растеризирането на окръжността във втора глава.

Нарастването $\Delta f(t)$ на една функция $f(t)$ се дефинира като:

$$\Delta f(t) = f(t+\delta) - f(t), \qquad \delta > 0,$$

което може да се запише и като:

$$f(t+\delta) = f(t) + \Delta f(t).$$

Записано в итерационни термини, горното равенство изглежда така:

$$f_{n+1} = f_n + \Delta f_n. \qquad [6.22]$$

Ще изведем рекурентна зависимост за стойността на функцията. За целта е необходимо да изразим нарастването (а евентуално и неговото нараства-

не също рекурентно). Ще трябва да диференцираме по този начин докато получим нарастване, което е константа.

Ако $f(t)$ е полином от трета степен (каквито са и почти всички криви, които разгледахме дотук), $\Delta f(t)$ е следният полином от втора степен:

$$f(t) = at^3 + bt^2 + ct + d,$$
$$\Delta f(t) = a(t-\delta)^3 + b(t-\delta)^2 + c(t-\delta) + d(t-\delta) - at^3 + bt^2 + ct + d \qquad [6.23]$$
$$= 3at^2\delta + t(3a\delta^2 + 2b\delta) + a\delta^3 + \delta^2 + c\delta$$

Можем сега да потърсим нарастването $\Delta^2 f(t)$ на функцията $\Delta f(t)$:

$$\Delta^2 f(t) = \Delta(\Delta f(t)) = \Delta f(t+\delta) - \Delta f(t).$$

След заместване с [6.23] в горната формула ще получим полином от първа степен спрямо t:

$$\Delta^2 f(t) = 6a\delta^2 t + 6a\delta^3 + 2b\delta^2. \qquad [6.24]$$

За следващото нарастване $\Delta^3 f(t)$ ще получим константа:

$$\Delta^3 f(t) = \Delta\left(\Delta^2 f(t)\right) = \Delta^2 f(t+\delta) - \Delta^2 f(t) = 6a\delta^3$$

Итерационните формули за получаването на n последователни точки от кривата, параметричното разстояние между които е еднакво и е равно на δ, са следните:

$$f_{i+1} = f_i + \Delta f_i; \qquad \Delta f_{i+1} = \Delta f_i + \Delta^2 f_i; \qquad \Delta^2 f_{i+1} = \Delta^2 f_i + 6a\delta^3; \qquad i = 1,\ldots,n$$

Началните стойности за f_0, Δf_0, $\Delta^2 f_0$ се получават директно от [6.23], [6.24] за $t = 0$:

$$f_0 = d; \qquad \Delta f_0 = a\delta^3 + b\delta^3 + c\delta; \qquad \Delta^2 f_0 = 6a\delta^3 + 2b\delta^2.$$

Тези рекурентни зависимости са приложени за координатните функции:

$$x = x(u) = a_x u^3 + b_x u^2 + c_x u + d_x \qquad u = 0,\ldots,1$$
$$y = y(u) = a_y u^3 + b_y u^2 + c_y u + d_y$$

```
void DrawFwdDiffCurve(ax,bx,cx,dx,ay,by,cy,dy,n)
float ax,bx,cx,dx, ay,by,cy,dy; int n;
{int i; float x,y, dx,dy,d2x,d2y,d3x,d3y, step;
    step=1/n;
    InitializeDiff(&x,&dx,&d2x,&d3x,ax,bx,cx,dx,step);
    InitializeDiff(&y,&dy,&d2y,&d3y,ay,by,cy,dy,step);
    MoveTo(x,y);
    for (i=0; i<n; i++) {
        x+=dx; dx+=d2x; d2x+=d3x;
        y+=dy; dy+=d2y; d2y+=d3y;
        DrawTo(x,y);
    }
}
```



```
void InitializeDiff(t,dt,d2t,d3t,a,b,c,d,step)
float *t,*dt,*d2t,*d3t,a,b,c,d,step;
{   (*t)  =d;
    (*dt) =step*(c+step*(b+a*step));
    (*d2t)=2*step*step*(3*a*step+b);
    (*d3t)=6*a*step*step;
}
```

Вторият начин за визуализация е чрез рекурсивно делене на кривата на части дотогава, докато:

- дължината ѝ стане по-малка или равна на някаква предварително зададена стойност или
- полученият сегмент има кривина, по-малка от някаква стойност и може достатъчно добре да се приближи от отсечка.

При визуализация върху растерни дисплеи деленето може да продължи докато частта от кривата стане по-малка или равна на един пиксел. Този метод е удобен не само за визуализация, но и за представяне на кривата чрез начупена линия с управляема точност. Това се налага често при автоматично генериране на програми за машини с ЦПУ (цифрово-програмно управление) при изработване на детайли с предварително зададен допуск. Тъй като споменатите машини работят най-вече с отсечки, достатъчно точното представяне на кривата чрез именно такива примитиви може да бъде от важно практическо значение.

За някои видове криви оценката за кривина може да се извърши сравнително лесно. Например за сегмент от крива на Безие като оценка за кривината може да се вземе максималното разстояние от точките $\mathbf{P}_1$ и $\mathbf{P}_2$ до отсечката $\mathbf{P}_0\mathbf{P}_3$.

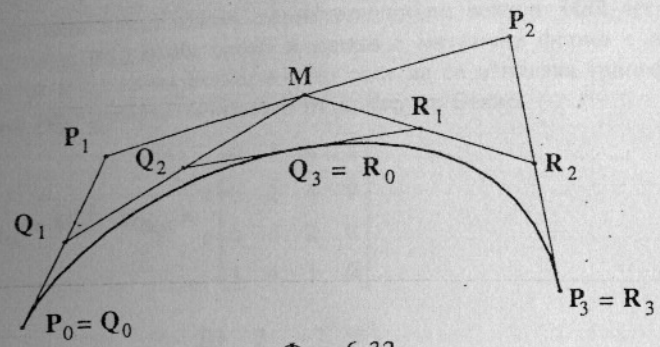
Разделянето на една крива на части може да се извърши по различни начини. В следващия параграф ще разгледаме някои от тях.

6.5.2 Подразделяне на криви

Разделянето на крива на няколко сегмента се налага много често в моделирането. Освен за визуализация чрез рекурсивно делене, това често се прави за да се увеличат степените на свобода на кривата *без* да се изменя нейната форма. Един начин за да се постигне това е да се повиши степента на кривата. Тъй като това в повечето случаи е нежелателно, тук ще се спрем на някои алтернативни методи.

Типичен пример за ефективно разделяне са кривите на Безие. Нека си припомним свойството на тези криви показано в 6.2.1 (фиг. 6-11). Да разгледаме един сегмент от кубична крива на Безие, определена с характеристичните точки $\mathbf{P}_0,\mathbf{P}_1,\mathbf{P}_2,\mathbf{P}_3$. Да разделим отсечките свързващи звената на характеристичната начупена и отсечките, свързващи средите им на по две равни части. Ще получим новите точки:

$$\mathbf{Q}_1 = (\mathbf{P}_0 + \mathbf{P}_1)/2, \qquad \mathbf{M} = (\mathbf{P}_1 + \mathbf{P}_2)/2, \qquad \mathbf{R}_2 = (\mathbf{P}_2 + \mathbf{P}_3)/2;$$
$$\mathbf{Q}_2 = (\mathbf{Q}_1 + \mathbf{M})/2, \qquad \mathbf{R}_1 = (\mathbf{M} + \mathbf{R}_2)/2;$$
$$\mathbf{Q}_3 = \mathbf{R}_0 = (\mathbf{Q}_3 + \mathbf{R}_1)/2$$



Фиг. 6-32

Получените два нови сегмента на Безие, определени с характеристичните начупени $\mathbf{Q}_0,\mathbf{Q}_1,\mathbf{Q}_2,\mathbf{Q}_3$ и $\mathbf{R}_0,\mathbf{R}_1,\mathbf{R}_2,\mathbf{R}_3$ ще съвпадат напълно с първоначалния. Всеки от тях обаче може да бъде модифициран независимо от другия чрез промяна на положението на характеристичните му точки.

В някои случаи се налага подразделяне в определено съотношение. Тогава потребителят трябва да зададе число t, а деленето на отсечките да се извърши в отношение $t:(1-t)$.

ПОДРАЗДЕЛЯНЕ НА В-СПЛАЙН КРИВИ. При В-сплайн кривите има два основни метода за повишаване на гъвкавостта. И двата имат за цел да вмъкнат нов възел във възловия вектор (и съответна нему нова характеристична точка) като разделят един параметричен интервал на две части. Първият метод е предложен от Коен-Лич-Ризенфелд (Cohen-Lyche-Riesenfeld) и е известен като алгоритъма "Осло", а другият принадлежи на Бьом (Böhm). Алгоритъмът "Осло" вмъква няколко възела едновременно, докато вторият прави това само за един възел и е донякъде частен случай на първия.

В основата на алгоритъма "Осло" стои фактът, че съществуват безброй характеристични начупени, които представят една и съща В-сплайн крива. Нека първоначалната крива има формата:

$$\mathbf{r}(t)=\sum_{i=0}^{n}\mathbf{r}_i N_{m,i+1}(t)$$

като възловият ѝ вектор е $\mathbf{k}_r=[t_0 \quad t_1 \quad \cdots \quad t_{n+m}]$.

Нека след вмъкването на нови възли кривата има новата форма:

$$\mathbf{q}(s)=\sum_{j=0}^{k}\mathbf{q}_j N_{m,j+1}(s)$$

като възловият ѝ вектор е $\mathbf{k}_q=[s_0 \quad s_1 \quad \cdots \quad s_{k+m}]$, при което $k>n$.

За да бъде изпълнено $\mathbf{r}(t)=\mathbf{q}(s)$, връзката между радиус-векторите на върховете на двете начупени в алгоритъма "Осло" се задава по следния начин:

$$\mathbf{q}_j=\sum_{i=0}^{n}\lambda_{i,j}^{m}\mathbf{r}_i \qquad j=0,1,\ldots,k. \qquad [6.25]$$

Коефициентите в уравненията [6.25] се пресмятат рекурсивно по формулите:

$$\lambda_{i,j}^{1}=\begin{cases}1 & t_i\le s_j<t_{i+1}\\ 0 & s_j\notin[t_i,t_{i+1})\end{cases}$$

$$\lambda_{i,j}^{m}=\frac{s_{j+m-1}-t_i}{t_{i+m-1}-t_i}\lambda_{i,j}^{m-1}+\frac{t_{i+m}-s_{j+m-1}}{t_{i+m}-t_{i+1}}\lambda_{i+1,j}^{m-1}$$

Включването на много върхове едновременно позволява да се запази равномерността (периодична или не) на първоначалния възлов вектор. Това може да стане, като всеки ненулев интервал се раздели на две равни части чрез прибавяне на средата му като нов възел. Когато първоначалният сплайн е неравномерен, новите възли могат да бъдат добавяни произволно.

Забележете, че този алгоритъм може да служи за преобразуване на една неравномерна В-сплайн крива в равномерна и дори периодична, което пък от своя страна може да се използва при преминаване от една форма към друга. На необходимостта от последното ще се спрем съвсем накратко в следващия параграф.

6.5.3 Преминаване от една форма към друга

Под преминаване от едно представяне в друго ще разбираме намиране на такъв характеристичен вектор и такава матрица, чрез умножение с която характеристичният вектор на дадена форма да се преобразува в търсения. Това се налага при:

- обмен на информация между различни системи за моделиране;
- предоставяне на потребителя на една система на възможност за работа с различни математически форми, докато вътрешното представяне, т.е. геометричният модел на тази система е само един;
- извършване на операции, които са по-удобни върху представяне, различно от това, с което се оперира. Типичен пример за това е подразделянето на криви на Безие.

Намирането на матричен израз на преминаването от една форма в друга е сравнително лесно за тези форми, които имат матрично представяне. Нека вземем за пример кривите на Безие и периодичните В-сплайн криви, зададени матрично съответно с [6.12] и [6.18]. Ако за простота разгледаме В-сплайн с характеристичен вектор $\mathbf{P}_0,\mathbf{P}_1,\mathbf{P}_2,\mathbf{P}_3$, при същия характеристичен вектор за една крива на Безие трансформационните матрици са задават с равенствата [6.26] и [6.27] съответно.

Интересен е случаят с *NUNRBS-кривите*, които нямат матрична форма. За тях се използва следният резултат: Една NUNRBS–крива с характеристичен вектор от четири точки върху възлов вектор [0 0 0 0 1 1 1 1] е крива на

Безие, определена върху същия характеристичен вектор. Най-лесният начин за преобразуване на такава крива в крива с матрична форма е първо да се преобразува тя в крива на Безие, а след това да се използва трансформационната матрица на преобразуването ѝ от крива на Безие.

$$M_{Bspl,Bez} = M_{Bez}^{-1} \cdot M_{Bspl} = \frac{1}{6}\begin{bmatrix} 0 & 1 & 4 & 1 \\ 0 & 2 & 4 & 0 \\ 0 & 4 & 2 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \qquad [6.26]$$

$$M_{Bez,Bspl} = M_{Bspl}^{-1} \cdot M_{Bez} = \begin{bmatrix} 0 & 2 & -7 & 6 \\ 0 & -1 & 2 & 0 \\ 0 & 2 & -1 & 0 \\ 6 & -7 & 2 & 0 \end{bmatrix} \qquad [6.27]$$

Задачи

6.1 Намерете параметричен сплайн, който интерполира зададено наредено множество от точки, първата и последната от които съвпадат. Съставете програма за визуализиране на получената затворена крива.

6.2 Напишете програма, която визуализира кривата на Оверхаузер, интерполираща зададено наредено множество от точки.

6.3 Кубичен сегмент на Безие е зададен с контролните точки: (0,0), (4,2), (1,3) и (2,1). Намерете наклона на допирателната към него в точката, в която параметърът $t=0.3$.

6.4 Намерете четири контролни точки, за да апроксимирате една окръжност с B-сплайн крива от втора степен върху равномерен възлов вектор. Каква е грешката на апроксимацията при $t=0.5$?

6.5 Напишете програма за визуализация на съставна крива на Безие от трета степен с неявно зададени краища на междинните сегменти, като на фиг. 6-13.

6.6 При използване на равномерен периодичен възлов вектор видяхме, че кубичният B-сплайн не интерполира началната и крайната си точки, ако те не са кратни. За един такъв сплайн, зададен с четири контролни точки, изразете началната и крайната му точки чрез контролните. Каква е разликата, ако B-сплайнът е от втора степен (трети ред)?

6.7 Напишете интерактивна програма, която осъществява локална модификация на B-сплайн чрез репозициониране на една от контролните му точки. Каква част от кривата трябва да пречертавате динамично за всяка нова позиция, получена от локатора?

6.8 Използвайте съставна крива от сегменти на Безие от втора степен, за да апроксимирате елипса, чиято голяма полуос е два пъти по-голяма от малката. Колко сегмента са ви необходими?

6.9 Изведете трансформацията, която преобразува четирите контролни точки на кубичен B-сплайн с равномерен възлов вектор (без повторения) в контролни точки на сегмент от крива на Безие, така че двете криви да съвпадат.

6.10 Предложете алгоритъм за намиране на пресечната точка (или точки) на два кубични сегмента на Безие.

6.11 Напишете програма за визуализация на рационална B-сплайн крива с неравномерен възлов вектор, така че да можете да променяте теглата на контролните точки интерактивно.

6.12 Напишете програма, която да разделя един кубичен сегмент на Безие на две части в точката, зададена със стойността на параметъра t.

6.13 Използвайте метода на Лайминг за да моделирате затворена крива, която се състои от гладко съединяващи се сегменти от конични сечения. Кои точки ще изберете за характеристични?

6.14 Ако е зададена B-сплайн крива с неравномерен възлов вектор и точка (x,y) в равнината, напишете програма, която намира съответната стойност на параметъра t. Не забравяйте за възможността да има много стойности y за дадено x, а също така и много стойности x за дадено y.

6.15 Какъв вид възлов вектор е най-подходящ за генериране на апроксимираща затворена B-сплайн крива? Как крайните характеристичните точки управляват нейната форма?

6.16 Напишете програма за визуализация на съставна крива от сегменти на Безие чрез рекурсивно делене на кривата на части при подходящо избран критерий за край на деленето. Използвайте например площта на четириъгълника, определен от характеристичните точки на сегмента.

6.17 Намерете аналитичен израз на първата производна на един сегмент на Безие. Начертайте кривата, която представя първата производна.

6.18 Напишете програма, която разделя един фрагмент от крива на Безие на два фрагмента в точка, зададена от потребителя.

6.19 Предложете метод за преобразуване на една неравномерна B-сплайн крива в периодичен равномерен сплайн.

6.20 Предложете структури от данни за съхраняване на съставни криви на Безие, сегментите на които да бъдат от произволна степен.

6.21 Предложете структури от данни за съхраняване на рационални B-сплайн криви.