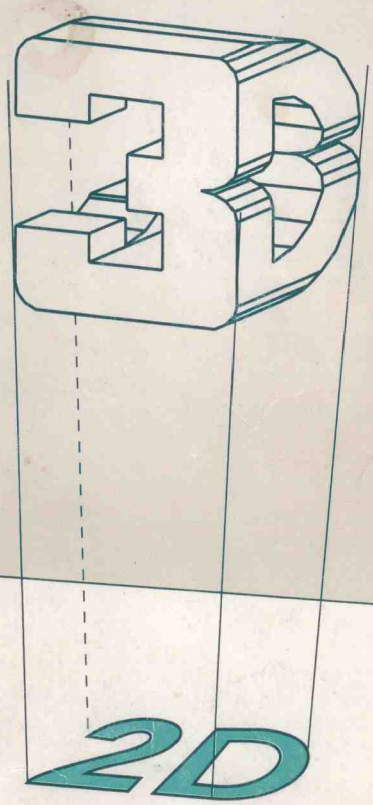
# КОМПЮТЪРНА ГРАФИКА



И

# ГЕОМЕТРИЧНО МОДЕЛИРАНЕ

ЧАСТ I В РАВНИНАТА




ЕВГЕНИЙ ЛУКИПУДИС

# 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.

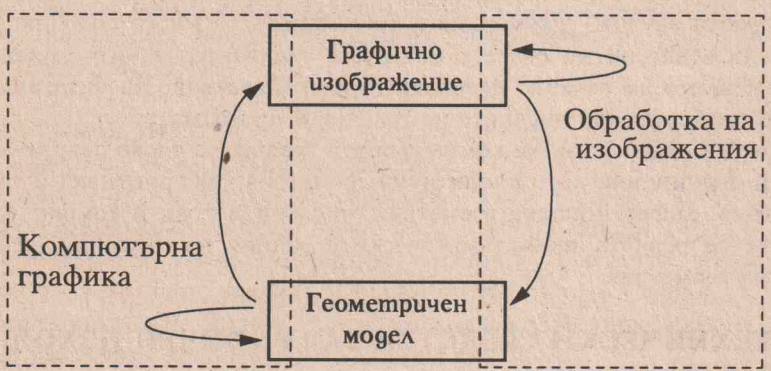
С масовото навлизане на сравнително евтините персонални компютри, снабдени с растерни дисплеи и устройства за графичен вход в началото на 80-те години разработването на приложни графични програми минава на друг етап. Графичното взаимодействие става предпочитан и сравнително лесно осъществим начин за 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.

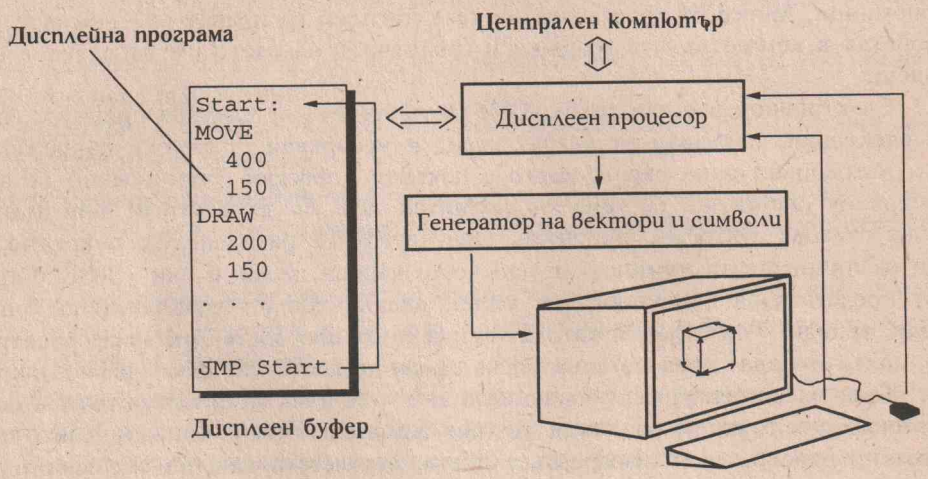
Не е нужно да се спираме по-подробно върху детайлите на горния процес. Това, което е важно за нас е, че светлинният импулс на луминофора затихва експоненциално с времето, така че образът се нуждае от непрекъснато прерисуване дори и тогава, когато е статичен. При честота на прерисуване типично 60Hz вече 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.



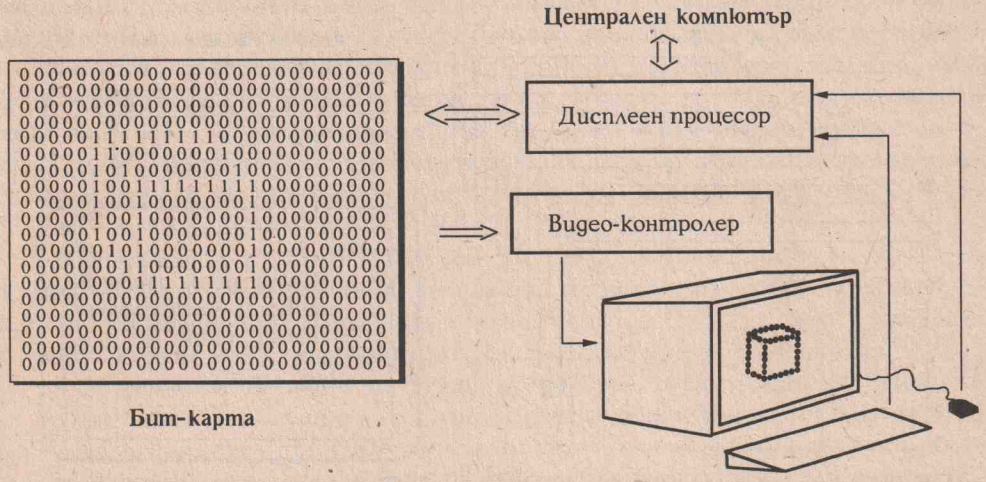

Фиг. 1-2

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 устрой-

ство. 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

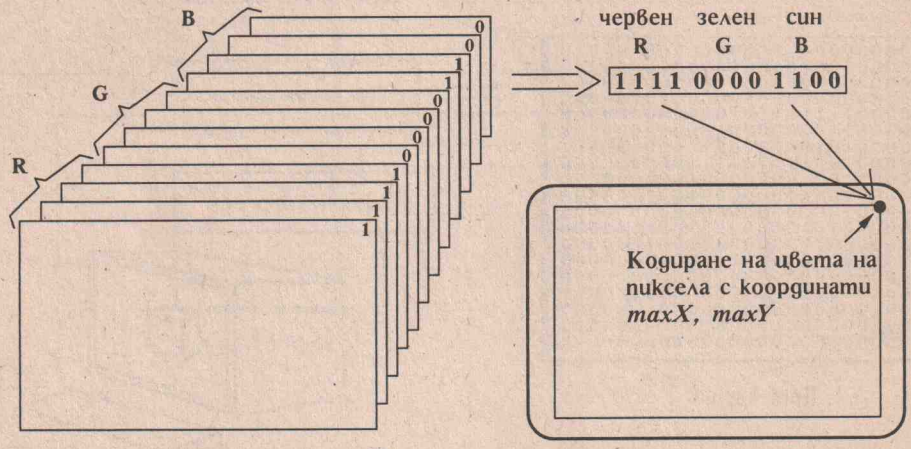
При черно-белите дисплеи дисплейният буфер се заменя от правоъгълна матрица (растер), която представя цялата област на екрана във вид на редове и стълбове от нули и единици, като всяка нула отбелязва точка, която не е осветена, а всяка единица - светеща точка. Тази матрица често се нарича *бит-карта*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




Пикселна карта с дълбочина 12 Екран на растерния дисплей

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 отчи-

тане 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

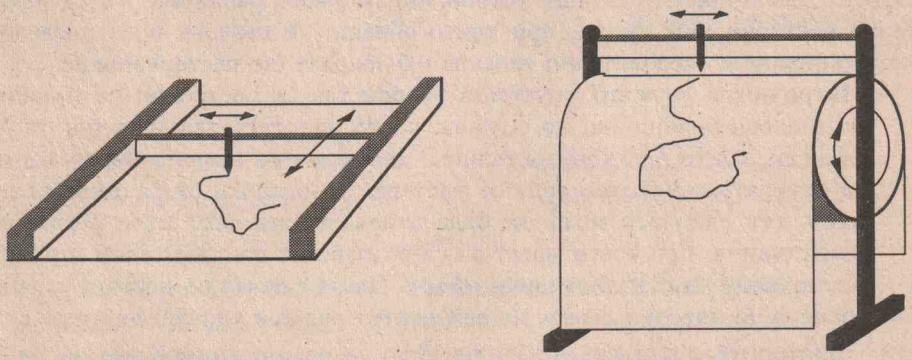
Тук ще направим съвсем кратък преглед на устройствата, които се използват за постоянен графичен изход, наричани с общото име *устройства за изход върху постоянен носител* (hardcopy devices). Те са необходими при окончателното визуализиране на резултата от работата на една приложна графична програма - 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.

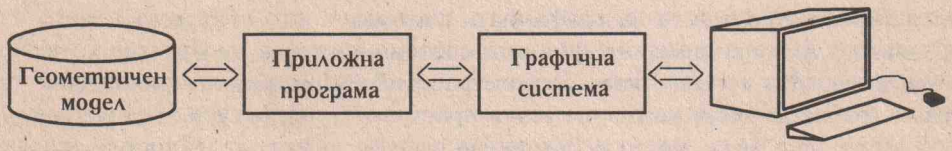
Използват се най-често два вида плотери (фиг. 1-5):

- **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 програмисткия 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).

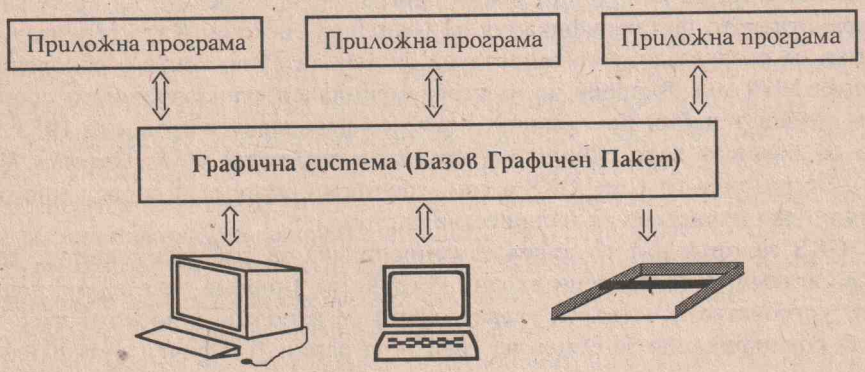
В някои приложения не се налага много често намеса на потребителя, а в други приложните програми са организирани като набор от програмни модули, всеки от които се активира и извършва определена дейност след интерактивно избиране на *команда* от потребителя. По-голяма част от интерактивната работа протича 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 пред-

лага на приложния програмист една графична йерархична база от данни. Всяка съхранена структура може да съдържа и нерафични 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 нерафична 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.

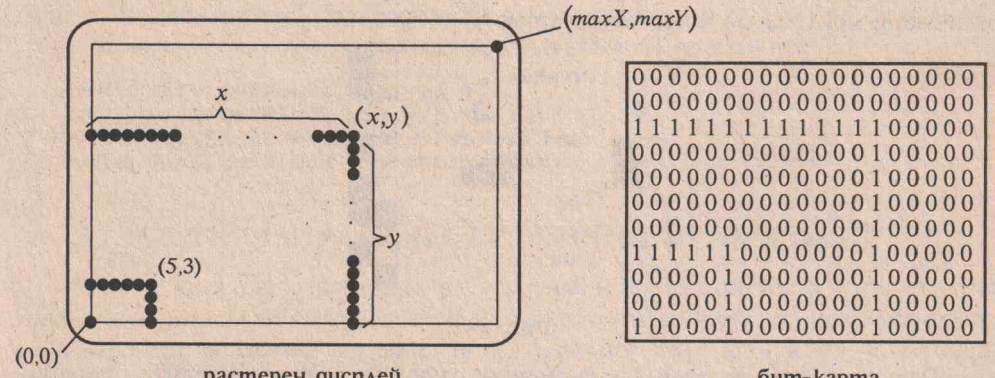
Приложенията могат да се класифицират и по вида на тяхната интерактивност - нуждата им от взаимодействие с потребителя. Тук възможностите са много, вариращи от приложни програми, в които потребителят задава поредица от данни и след това получава визуалния образ на резултата до т.нар. *интерактивно проектиране*, 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.

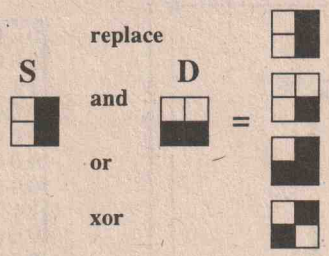
От гледна точка на програмиста обслужването на едно растерно устройство е сравнително елементарно. Пикселите се адресират с техните декартови координати и във всеки пиксел може да се запише дадена стойност, както и да се прочете тази, която вече е записана там. Записването на нова стойност в бит-картата 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** ← **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".

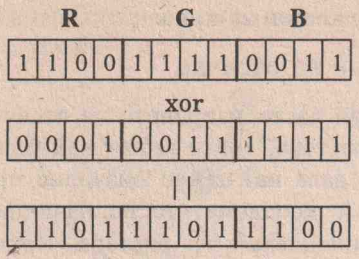
В огромна част от случаите записването в бит-картата се извършва с операцията **replace**. Затова и в повечето растерни системи се използва *текущ режим на записване*, вместо изрично да се задава операцията, с която всеки пиксел се модифицира. За растерни дисплеи, пикселите на които могат да имат само 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 коя-

то 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: **D** ≡ **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.

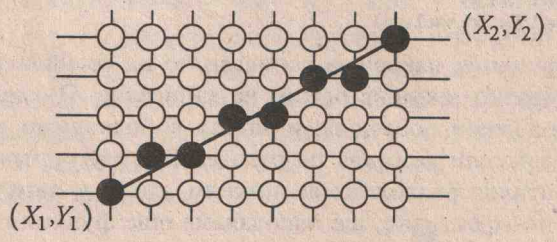
### 2.1.1 Растеризиране на отсечка

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:

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

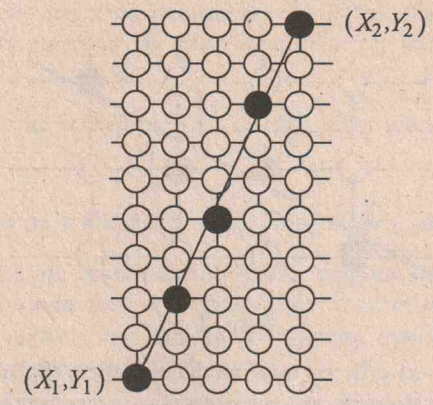



Фиг. 2-3

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

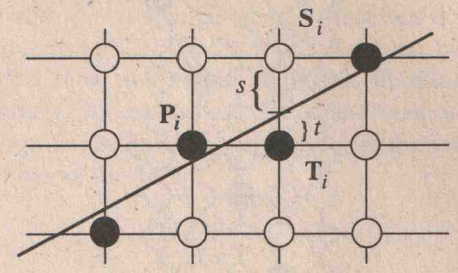
Следната функция ще осъществи растеризирането по описаната схема:

```
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;
    }
}
```

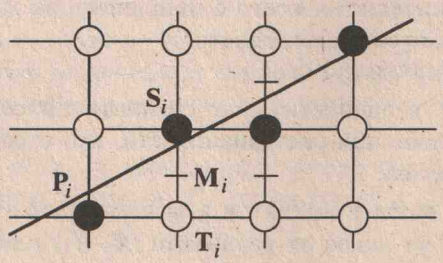
**АЛГОРИТЪМ НА СРЕДНАТА ТОЧКА.** Ван Аакен (Van Aken) разработва един интересен подход към растеризирането - т.нар. *принцип на средната точка* - използването на който позволява създаването на алгоритми за целочислено растеризиране не само на отсечки и окръжности, но и на конични сечения. Прилагането 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:

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

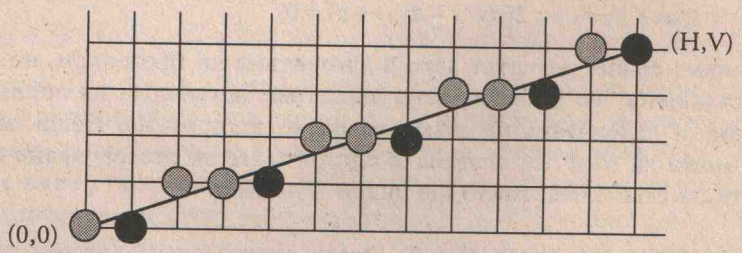
Нека да представим дробите в горната система от неравенства като цяла

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 $\theta$ в параметричното уравнение:

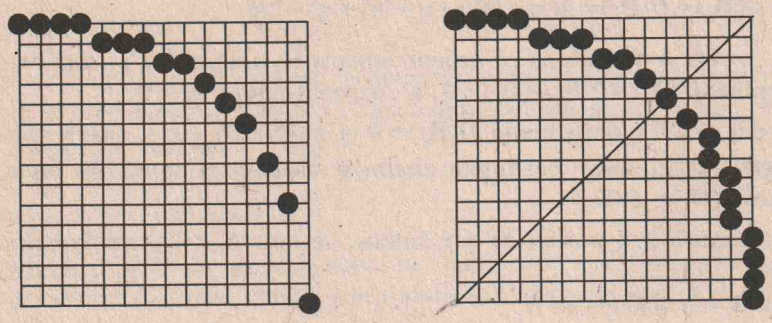
$$\begin{cases} x = X_C + R \cdot \cos\theta \\ y = Y_C + R \cdot \sin\theta \end{cases}$$

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:

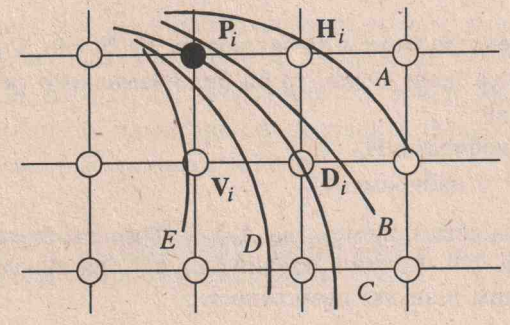
$$x = X_C + R \cdot \frac{1-t^2}{1+t^2} \qquad y = Y_C + R \cdot \frac{2t}{1+t^2}$$

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:

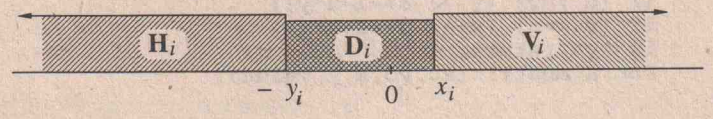
- Ако $\Delta_i < 0$ и $\Delta_i \le -y_i$ избираме точката $\mathbf{H}_i$;
- Ако $\Delta_i > 0$ и $\Delta_i > x_i$ избираме точката $\mathbf{V}_i$;
- Във всички останали случаи ще избираме $\mathbf{D}_i$

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 $\Delta_i$:




Фиг. 2-10

За оценката $\Delta_i$ 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 $\Delta_i$ ще бъде $\delta_i$, 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:

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

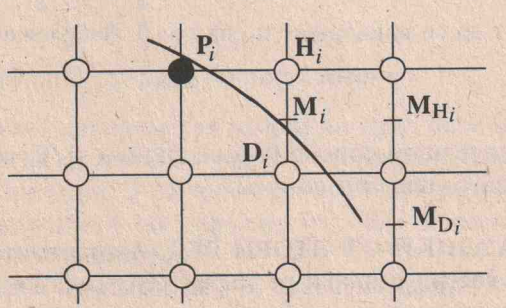
имайки пред вид, че началната точка е $(0,R)$.

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 $d_i^D$ нарастването 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:

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

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

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

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:

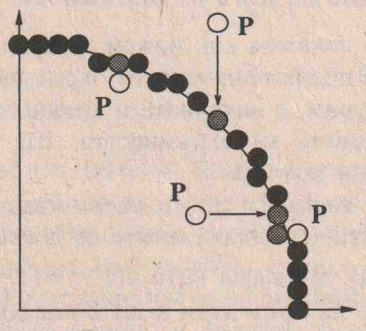
$$|e(x+1,y)| < |e(x,y+1)|$$

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].

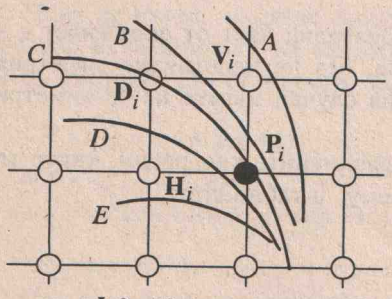
Тези нараствания съответстват на стъпка или само по $x$, или само по $y$, или по диагонал. Необходимо е да отбележим, че ролите на $x$ и $y$ в нашите разсъждения ще са разменени, защото растеризирането ще извършваме по посока на нарастването на ъгъла, което в I квадрант съвпада с нарастването на $y$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

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

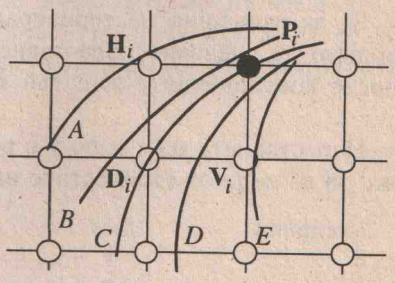
Тъй като $\Delta_i = D(\mathbf{D}_i) = D(\mathbf{P}_i) + ddx + ddy$, неравенството $\delta_i < 0$ ще се преобразува в:

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

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

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

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

Фиг. 2-13

Да разгледаме дефинициите на оценките и сравненията за първите два квадранта. Вижда се, че дефинициите на $\delta_i$ и $\varepsilon_i$ 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:

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

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

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

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.

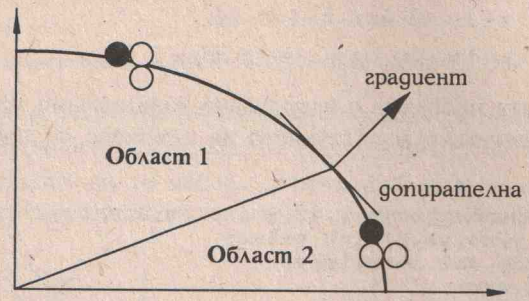
```
void DrawBresenhamArc(x,y,xc,yc,xe,ye,R,value)
int xc,yc,x,y,xe,ye,R; int value;
{int d,ddx,ddy,dd2,incx,incy;
    x-=xc; y-=yc;
    xe-=xc; ye-=yc;
    PutPointOnCentralCircle(&x,&y,R);
    PutPointOnCentralCircle(&xe,&ye,R);
     /* определяне на нарастванията в началния квадрант */
    incx=y?-sign(y):-sign(x);
    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

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

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.

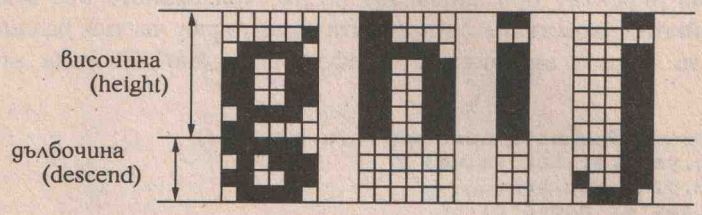
```
void DrawMidpointEllipse(xc,yc,a,b,value)
int xc,yc,a,b; int value;
{int x,y,d, asq,bsq;
    asq=a*a; bsq=b*b;
    x=0; y=b; d=4*bsq-asq*(1-4*b);
    while (asq*y>bsq*x) {                          /* Област 1 */
          FourSymmetric(xc,yc,x,y,value);
          if (d>=0) {
              d+=4*bsq*(2*x+3)+asq*(-2*y+2);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

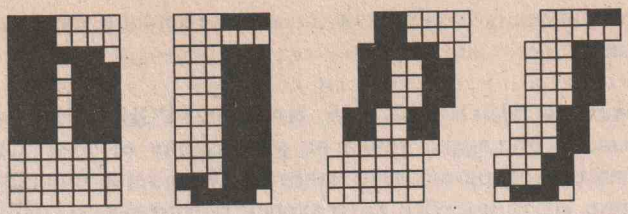
Да разгледаме един от тези методи:

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.

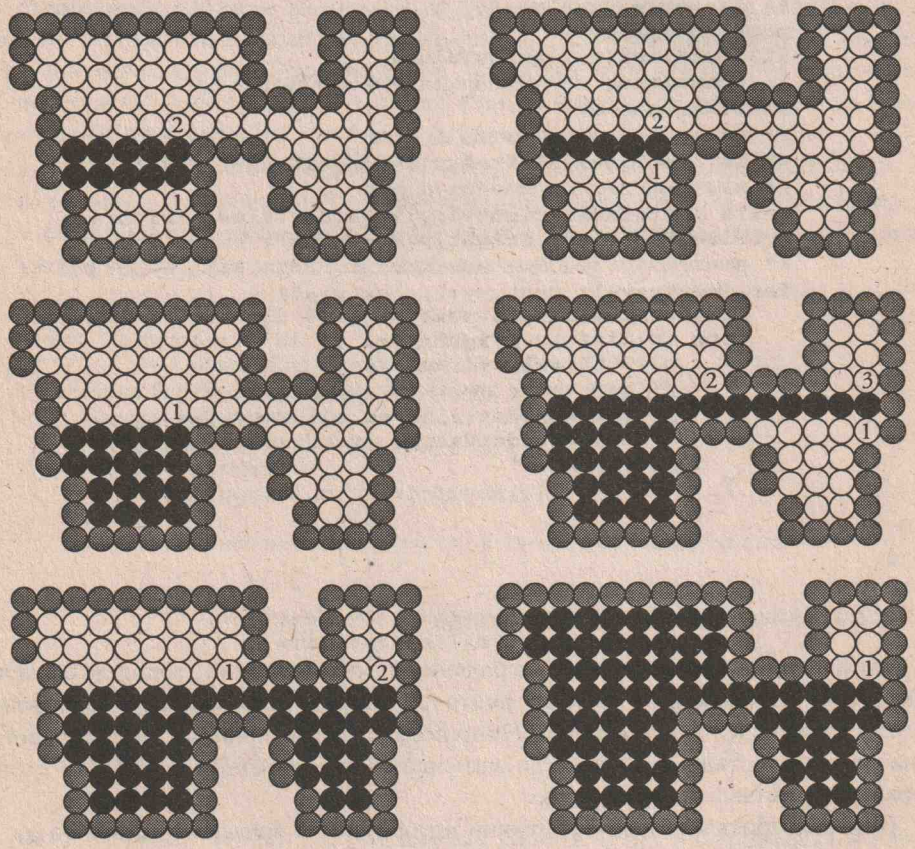
```
void SimpleBoundaryFill_4(x,y,NewValue,BorderValue)
int x,y,NewValue,BorderValue;
{int value;
    value=GetPixel(x,y);
    if (value!=NewValue && value!=BorderValue) {
       PutPixel(x,y,NewValue);
       SimpleBoundaryFill_4(x-1,y,NewValue,BorderValue);
       SimpleBoundaryFill_4(x+1,y,NewValue,BorderValue);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

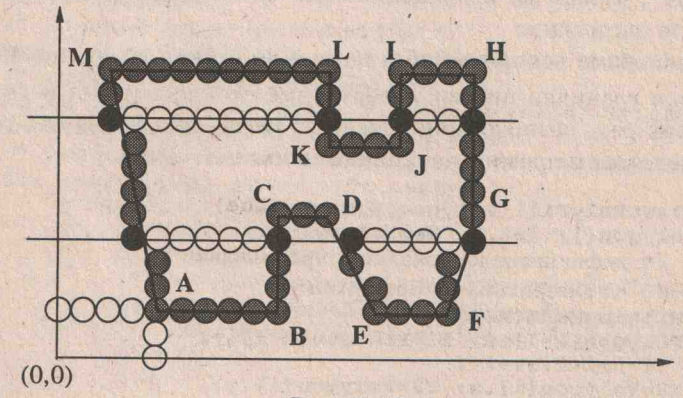
```
void InvertPolyFill(Polygon,N,NewValue)
Point Polygon[]; int N, NewValue;
{. . .  /* дефиниране на локалните променливи */
        /* от алгоритъма на Брезенхам  */
    SetWritingMode(XOR);
    X1=Polygon[N-1].x; Y1=Polygon[N-1].y;
    for (i=0;i<N;i++) {
       X2=Polygon[i].x; Y2=Polygon[i].y;
       . . .
       /* 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);

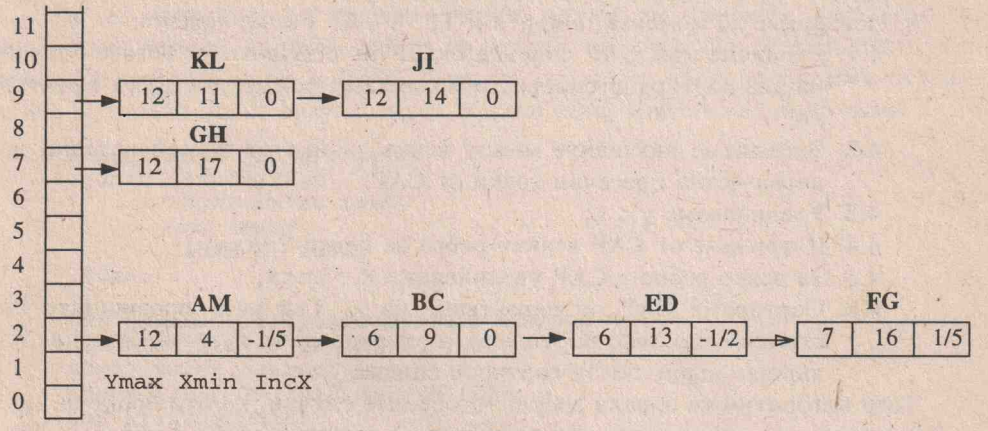
`Xmin`: $x$-координата на по-ниския край на реброто (това е $x$-координатата на този край, чиято $y$-координата е `Ymin`);

`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

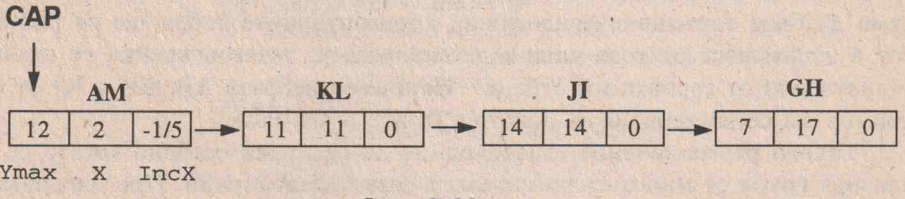
Алгоритъмът на сканиращия ред използва тази таблица, за да поддържа на всяка стъпка (за всеки текущо обработван ред от растера) един списък от всички ребра, които имат пресечни точки с този ред. Този списък ще наричаме *списък на активните ребра* - САР. Всеки елемент от него характеризира по едно ребро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

Ето и самия алгоритъм:

1. Построяваме ТР за многоъгълника;
2. Инициализираме САР като празен списък;
3. Даваме на `y` стойността на най-малкото `y` в ТР;
4. Повтаряме до момента, в който и ТР, и САР станат празни:
   - 4.1. Добавяме към САР списъка от ТР 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 САР;
   - 4.3. Увеличаваме `y` с `1`;
   - 4.4. Изтриваме от САР всички ребра за които `Ymax=y`;
   - 4.5. За всяко ребро в САР увеличаваме `X` с `IncX`;
   - 4.6. Сортираме САР 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 САР, 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 САР 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 сѐ 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.

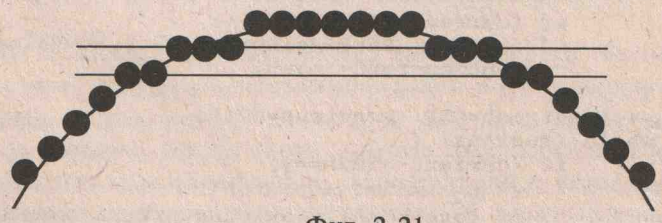
Тъй като IncX е рационално число, алгоритъмът може да се преобразува в целочислен, като се съхраняват отделно числителят и знаменателят на това число. Закръгляването надолу тогава извършваме, като за всяка пресечна точка X пазим и числителя на дробната ѝ част. На всяка стъпка ще прибавяме към него числителя 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.

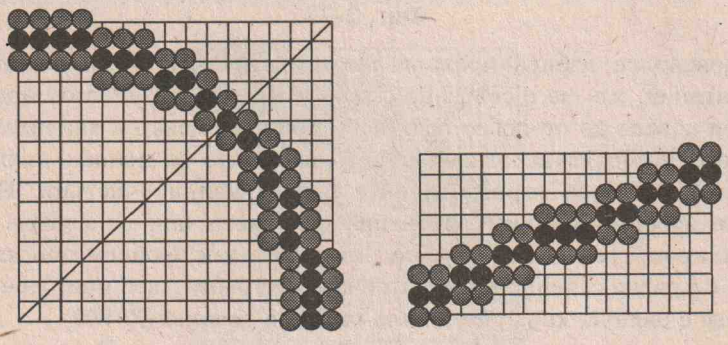
Възможностите на растерните дисплеи за визуализация на чертежи съвсем не отговарят на възможностите, които един чертожник има върху простия бял лист. Това обаче не пречи да се разработват средства за имитиране на най-важните графични атрибути от инженерните чертежи, така че създаваните визуални образи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



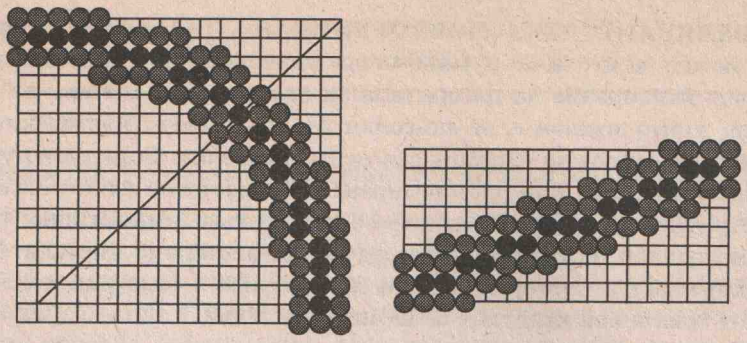
Фиг. 2-22

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:

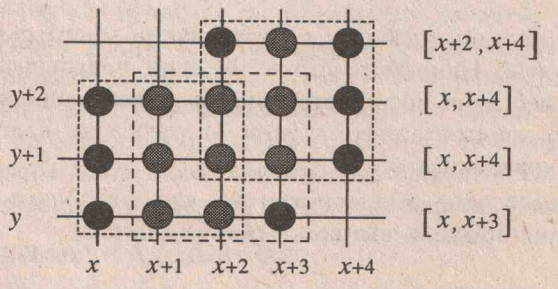
- Какво е най-подходящото сечение на перото?
- Трябва ли сечението да се ориентира спрямо примитива, който се удебелява?



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.

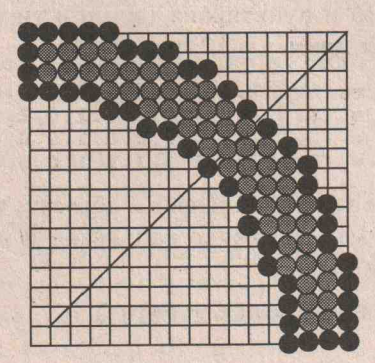



Фиг. 2-24

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.

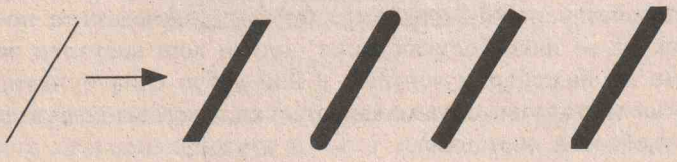
**УДЕБЕЛЯВАНЕ ЧРЕЗ ЗАПЪЛВАНЕ.** Този метод дава най-добри визуални резултати и може да се реализира чрез създаване на област, която впоследствие се запълва по някой от разгледаните в предната част методи. Контурът на областта се получава при отместване на примитива по посоката на нормалата му на $W/2$ 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.

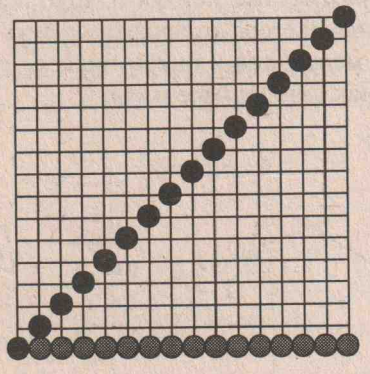
За разлика от образеца при запълване на области, този се привързва към обекта - почти винаги към неговата начална точка, а не към екрана. Единственото изключение са начупените линии, при които образецът трябва се привързва към началото на начупената, а не към началото на всяка нейна отсечка.

## 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 пикселна 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 $\sqrt{2}$.

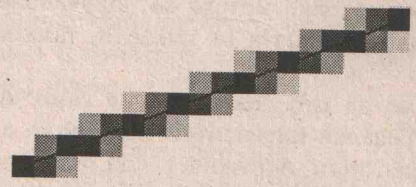


Фиг. 2-27

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?

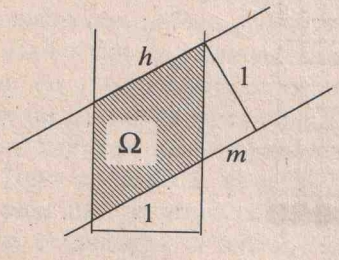
Възможни са няколко различни отговора на тези въпроси. На фигурите досега изобразявахме пикселите като кръгове, за да може да ги отличаваме от координатната мрежа. По-правилно е да третираме всеки пиксел като квадрат със страна 1, с чиито координати $(x,y)$ ще отбелязваме центъра му. Това означава, че 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 $d = F(\mathbf{M})$ 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 пря-

ко свързано със стойността на оценката:

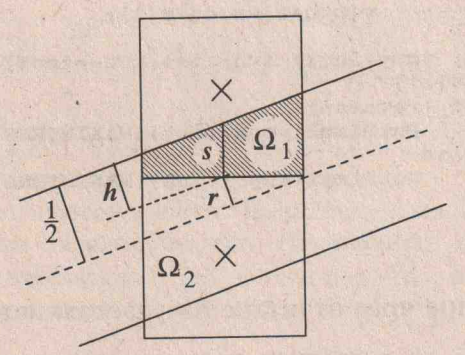
$$r = \frac{F\left(x_i + 1, y_i + \frac{1}{2}\right)}{2\sqrt{dx^2 + dy^2}} = \frac{d}{2\sqrt{dx^2 + dy^2}}.$$

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 $\Omega_1$ и $\Omega_2$.

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.

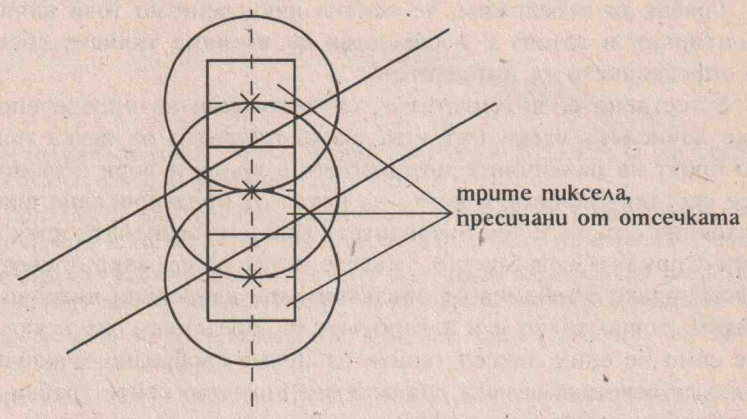
Накрая ще отбележим, че тази сравнително нетривиална апроксимация има смисъл само ако броят на различните интензитети, които можем да зададем на един пиксел е сравнително голям. Това далеч не е така при конвенционалните растерни дисплеи. Ако работим с растерен дисплей, в който за всеки пиксел са отделени 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:

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

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

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

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 постараете 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.

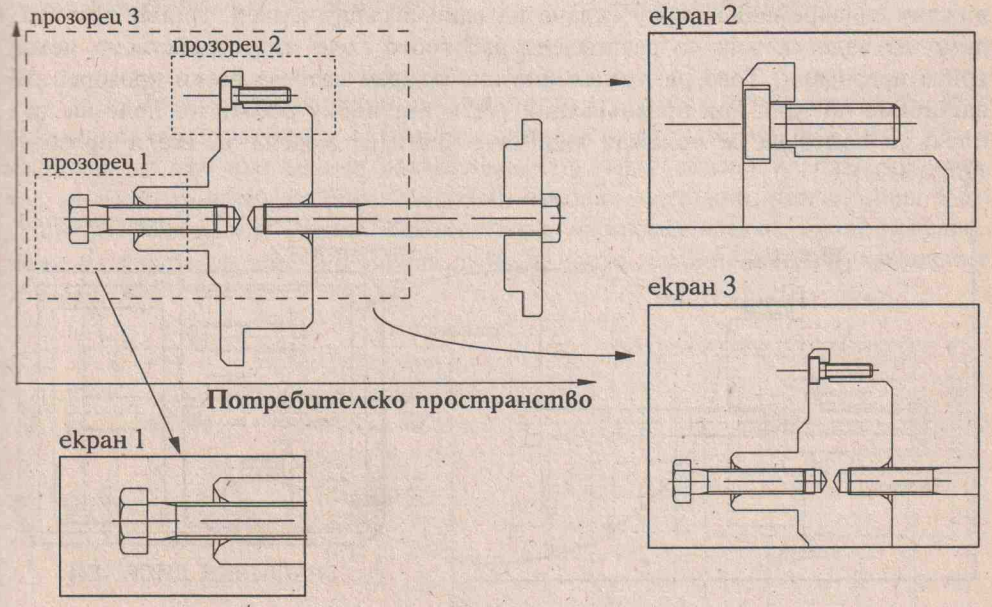
От друга страна, алгоритмите, които разгледахме в предишната глава извършват растеризиране в координатното пространство на графичното устройство. То се дефинира от координатна система, която също е декартова, но е целочислена и началото ѝ е най-често в началото на *работното поле* на устройството. Под *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

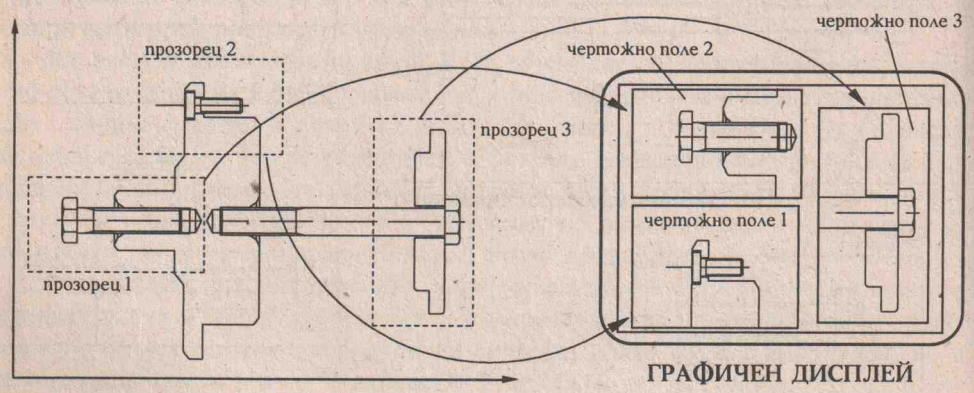
На фиг. 3-1 е показан геометричният модел на един машинен детайл. В него са дефинирани няколко различни потребителски прозореца, които са напълно независими един от друг. Можем да си представим, че всеки от тях е изобразен върху целия екран на три различни графични дисплея, давайки възможност на различни 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).

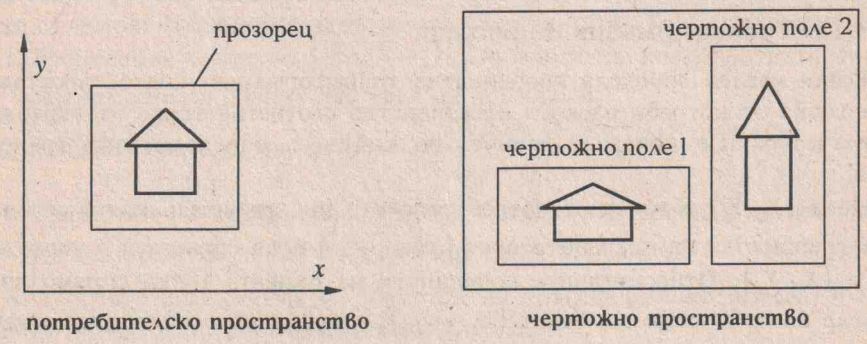



Фиг. 3-2

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

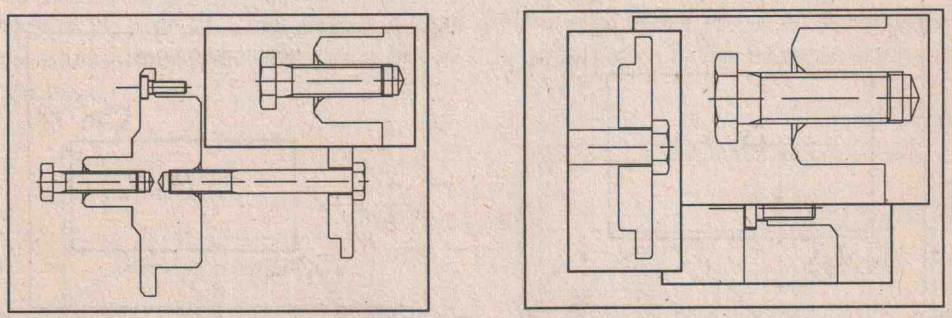
на чертожното поле естествено не бива да превишават размерите на работното поле на самото устройство. В частност, чертожното поле може да съвпада с целия екран на един графичен дисплей.

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.

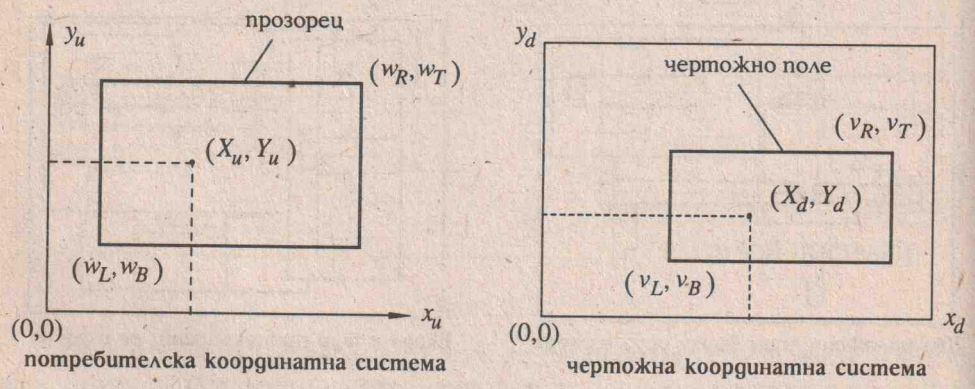
На рисунката отдясно три различни прозореца са визуализирани в три припокриващи се чертожни полета. Да отбележим, че в общия случай, когато отношението на ширина към височина на правоъгълниците на прозореца и чертожното поле са различни, визуализираните обекти ще бъдат мащабирани различно по $x$ и по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

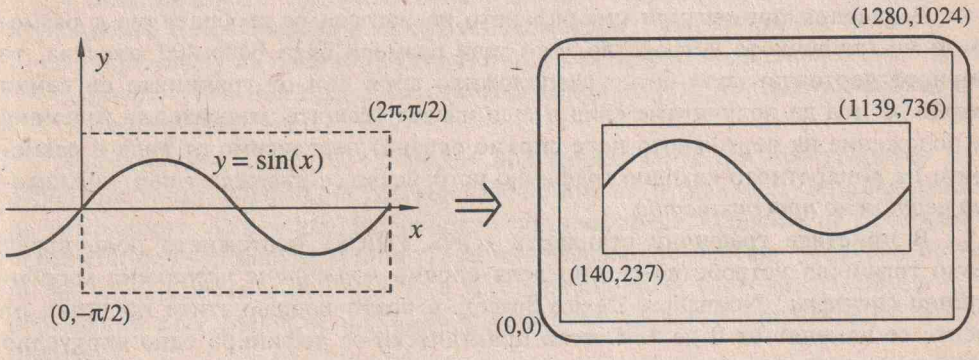
1. Транслация с вектор $(-w_L, -w_B)$: Началото на координатната система се премества в долния ляв ъгъл на прозореца;
2. Мащабиране с коефициенти $S_x = \frac{(w_R - w_L)}{(v_R - v_L)}$ и $S_y = \frac{(w_T - w_B)}{(v_T - v_B)}$;
3. Транслация с вектор $(v_L, v_B)$: Преместване на началото на координатната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

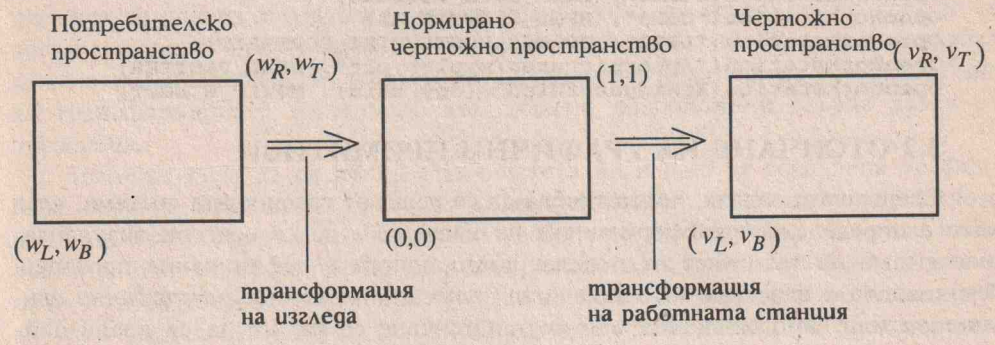
### 3.1.4 Нормирано чертожно пространство

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)
```

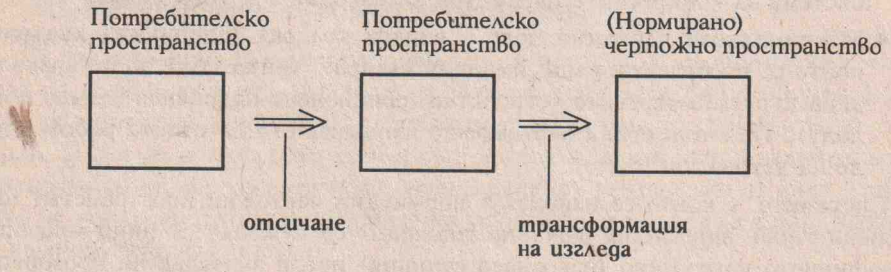
Това предизвиква генерирането на подходяща трансформация на работната станция и избора на съответния драйвер за обслужването на графичния изход. По-долу са показани няколко възможности за инициализация на графичното устройство: екран; плотер; печатащо устройство; файл за плотер и *прозорец на процес*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



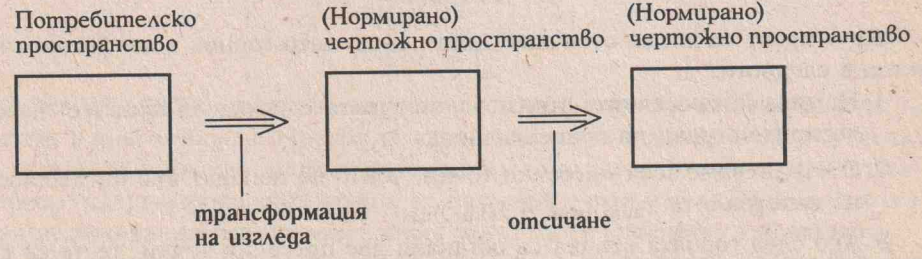

Фиг. 3-8

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, отсича-

нето 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.



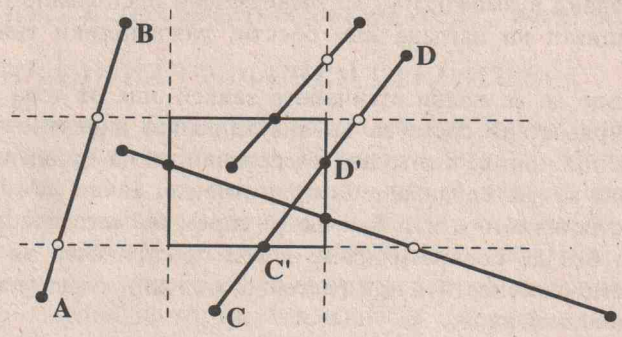

Фиг. 3-9

### 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 краищата 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:

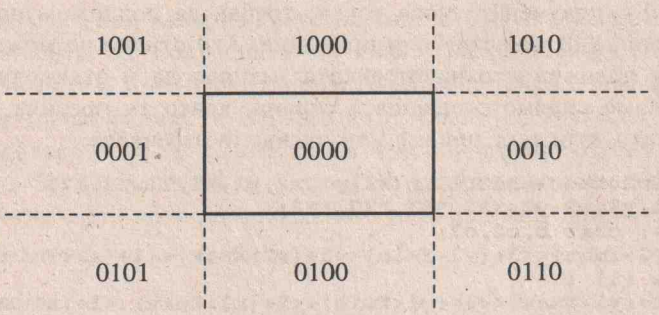
Правите, на които лежи всяка от страните на правоъгълника разделят равнината на 9 области, всяка от които може да се характеризира еднозначно от едно 4-битово число. Всеки бит от това число съответства на положението на областта спрямо всяка от четирите прави:

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.

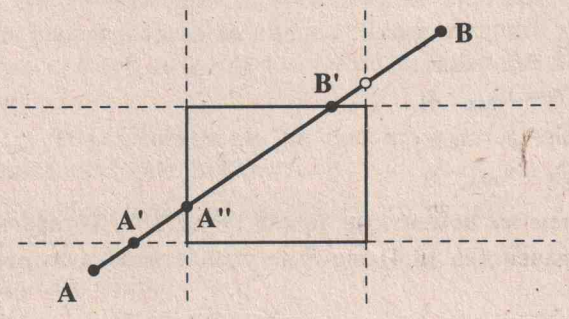
След като намерим пресечната точка с избраната страна и заменим съответния й край с получената нова точка, трябва да подложим новата отсечка отново на теста за отхвърляне или приемане. Ако отново не можем тривиално да определим положението на отсечката (изцяло да я отхвърлим или приемем), я отсичаме 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:

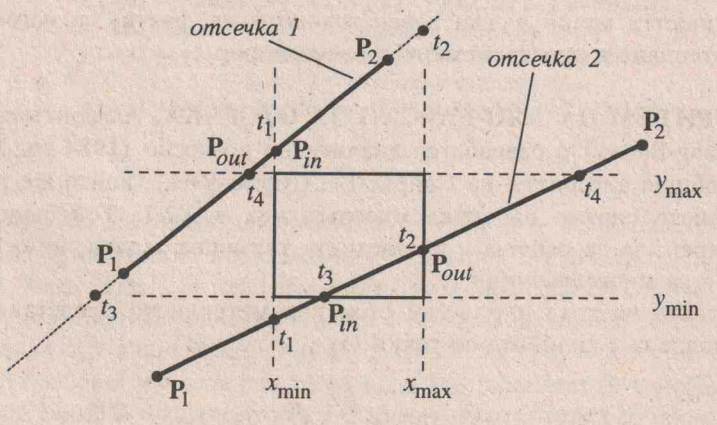
$$R_i \cdot t \le Q_i, \qquad i = 1,2,3,4 \qquad [3.4]$$

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).

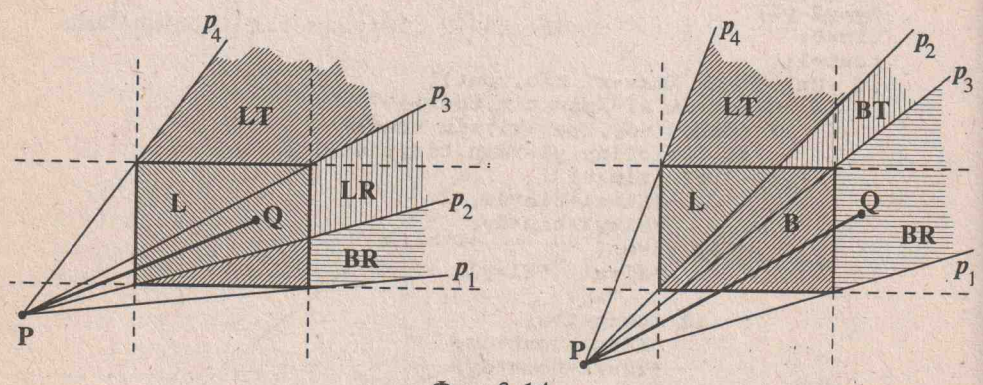
```
int LiangBarsky2DClip(x1,y1,x2,y2,X1,Y1,X2,Y2)
float x1,y1,x2,y2
float *X1,*Y1,*X2,*Y2;
{float dx,dy,tin,tout;
   dx=x2-x1;
   dy=y2-y1;
   tin=0;
   tout=1;
   if (CalcT(-dx,Xmax-x1,tin,tout))
      if (CalcT(dx,x1-Xmin,tin,tout))
         if (CalcT(-dy,Ymax-y1,tin,tout))
            if (CalcT(dy,y1-Ymin,tin,tout))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

дима. 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:

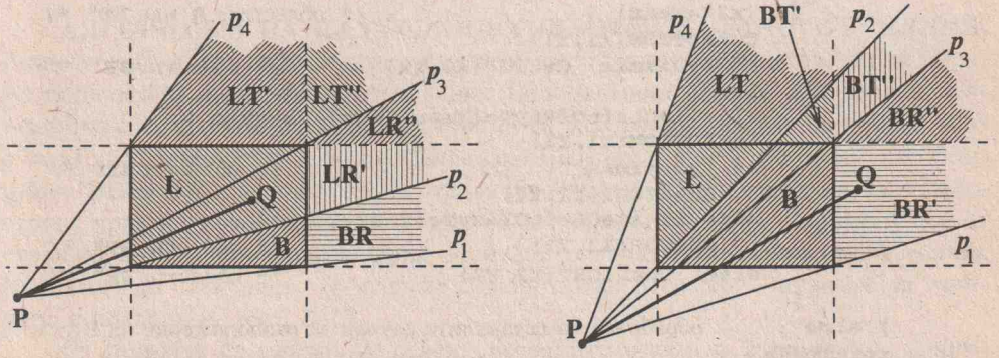
**L** - отсечката пресича лявата страна;
**LR** - отсечката пресича лявата и дясната страни;
**LT** - отсечката пресича лявата и горната страни;
**B** - отсечката пресича долната страна;
**BR** - отсечката пресича долната и дясната страни;
**BT** - отсечката пресича долната и горната страни;

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:

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

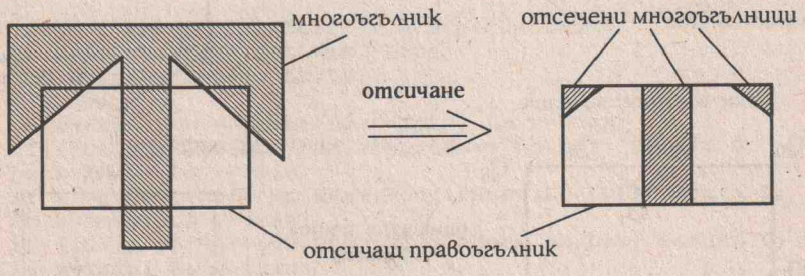
```
#define isQLeftOfRayTo(x,y)  dy*(x-x1)>dx*(y-y1)

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).



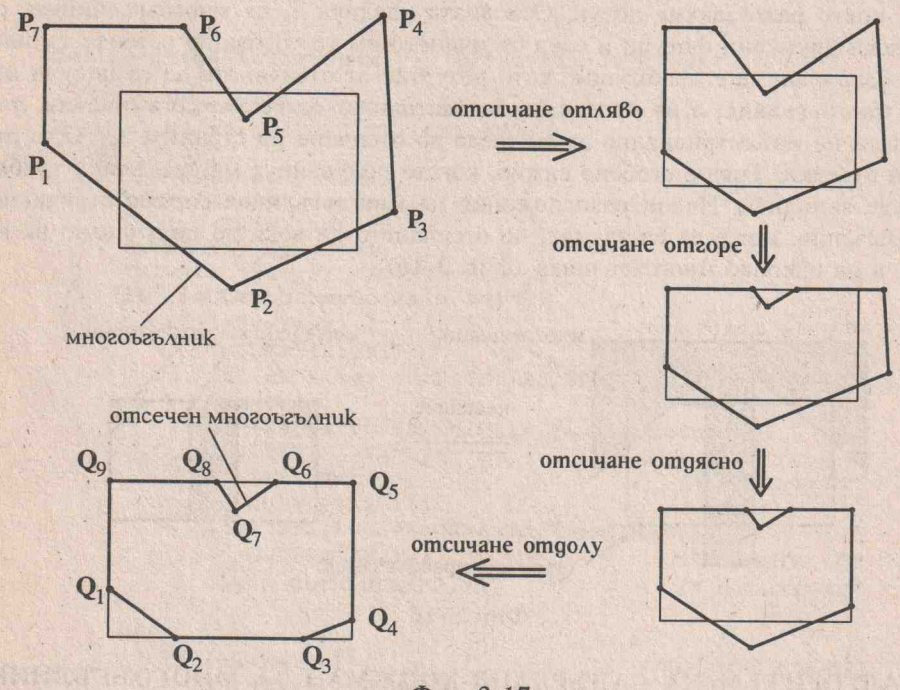

Фиг. 3-16

**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.



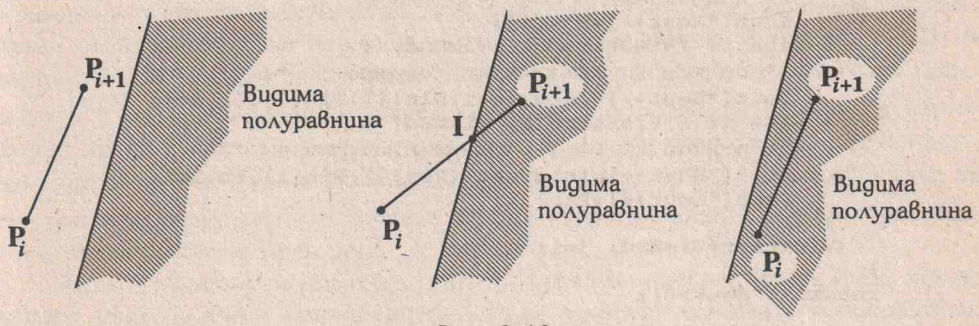

Фиг. 3-17

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 права-

та 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

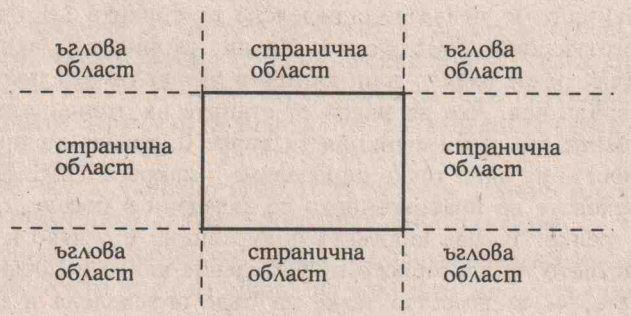
```
int SutherlandHodgman2DClip(Pin,Nin,Pout,Nout)
Point Pin[],Pout[]; int Nin,*Nout;
{Point Ptmp[MAXPNTS], Lb,Le;
 int Nnew;
   Lb.x=Xmin; Lb.y=Ymin; Le.x=Xmin; Le.y=Ymax;
   if (!ClipByLine(Pin,Nin,Ptmp,&Nnew,Lb,Le)) return 0;
   Le.x=Xmax; Le.y=Ymin;
   if (!ClipByLine(Ptmp,Nnew,Pout,&Nnew,Lb,Le))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

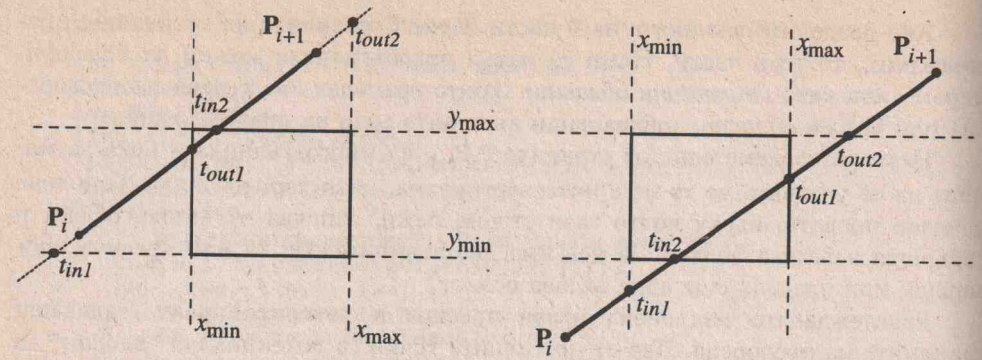
1. $t_{in2} \le t_{out1}$ (правата ѝ пресича прозореца) и
2. двата интервала $[t_{in2}, t_{out1}]$ и $[0,1]$ имат обща част. Това може да се запише като условието:

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

   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:

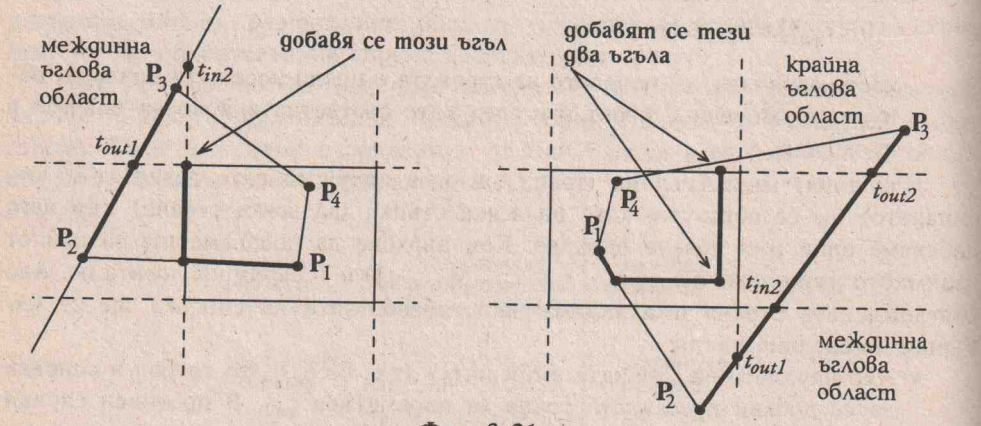
- Ако началото на страната е преди $t_{in2}$ (т.е. $0 < t_{in2}$), то трябва в списъка да се добави пресечната точка за параметъра $t_{in2}$. В противен случай началото ѝ е вътрешно за прозореца и е било добавено на предишната итерация (като крайна точка на предната страна).
- Ако краят на страната е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

### 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 символи 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 $\mathbf{Q}(t)$:

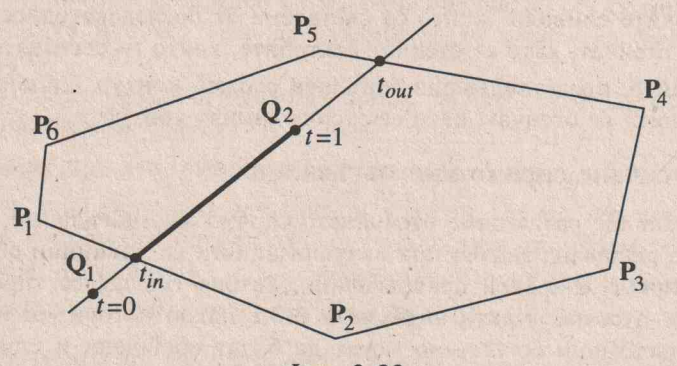
$$\mathbf{T} = \mathbf{Q}(t) - \mathbf{P}_i$$

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 $\pi$ (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:

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

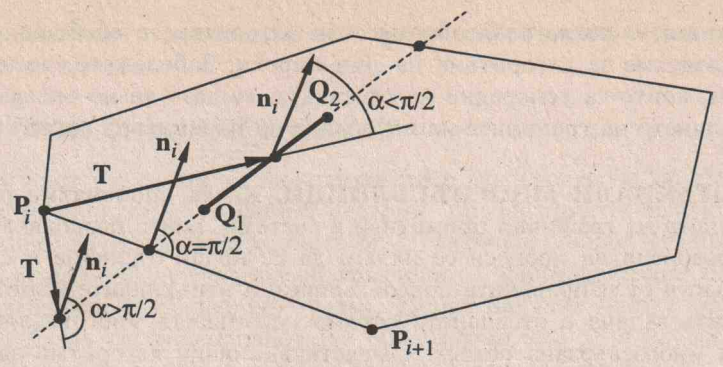
в които след заместване на $\mathbf{Q}(t)$ от [3.4] ще получим:

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

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 $\mathbf{D}=[\mathbf{Q}_2-\mathbf{Q}_1]$ 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 $t_{in}, t_{out}$:

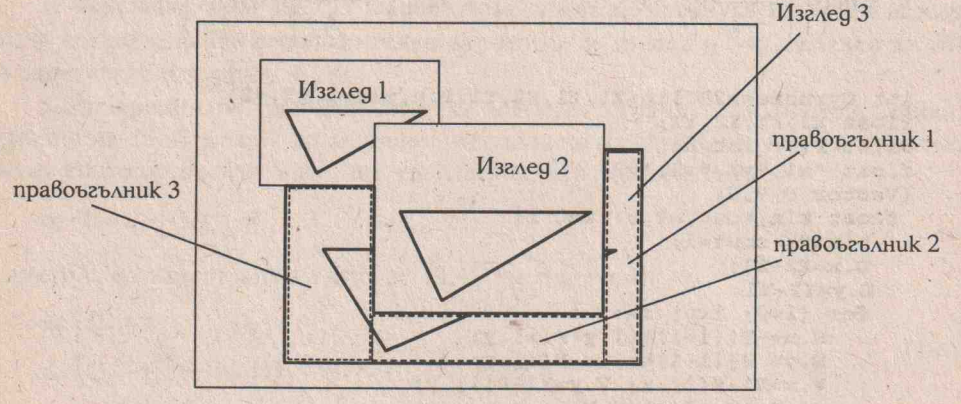
$$t_{in}=\max\left(\left\{-\frac{\mathbf{n}_i\cdot\mathbf{V}_i}{s_i}\middle| s_i>0\right\},0\right), \qquad t_{out}=\min\left(\left\{-\frac{\mathbf{n}_i\cdot\mathbf{V}_i}{s_i}\middle| s_i<0\right\},1\right).$$

```
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 подробно 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

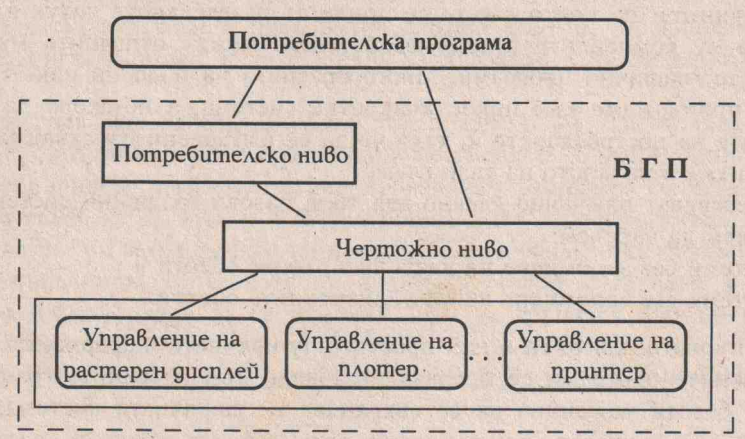
Приложният програмист има достъп до базовия графичен пакет чрез набор от функции, към които той може да прави обръщения от своята графична програма. Те са организирани в няколко основни групи. Едната включва функции, които осигуряват чертането в потребителска координатна система и установяването на параметри 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 ра-

ботната 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 $2\pi$ 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 представеният 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 работ-

ната станция и изгледа (в някои системи и за всички дефинирани изгледи);

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

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:

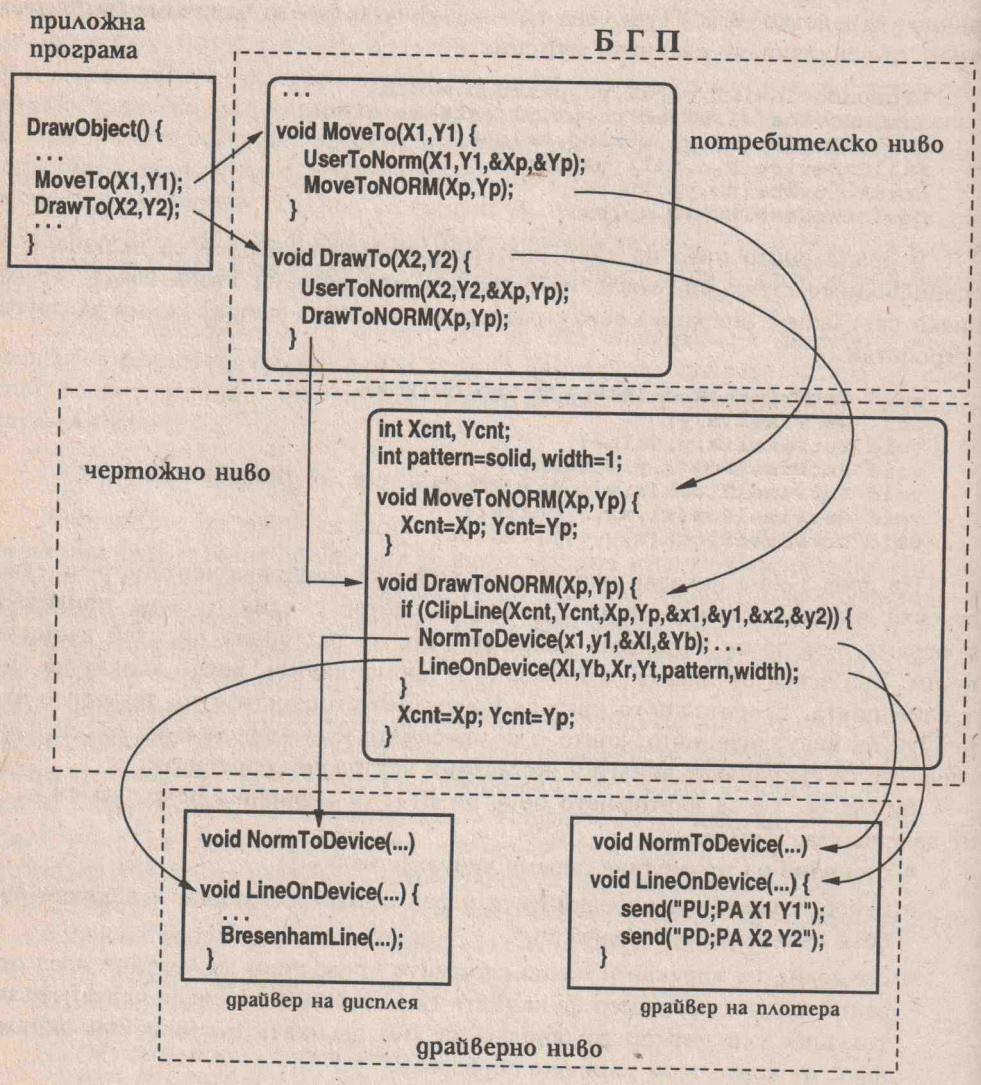
```
void SetWritingMode(REPLACE|AND|OR|XOR)
int  GetPixel(x,y)
void PutPixel(x,y,value)
void PutPixelRow(x,y,w,value)
void PutPixelBlock(x,y,w,h,buffer)
void GetPixelRow(x1,x2,y,buffer)
void GetPixelBlock(x,y,w,h,buffer)
```

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 показахме 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

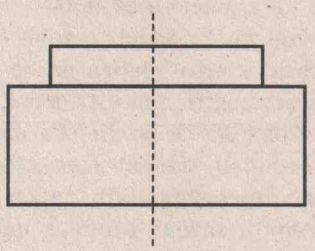
спрем по-късно, тъй като в нея освен частта за визуализация е реализирана и една *система за геометрично моделиране*.

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)
```

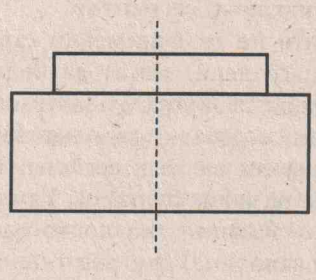
  - функция, която установява мястото, откъдето да започне добавянето или заместването с обръщения към нови функции (нови елементи):

```
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:

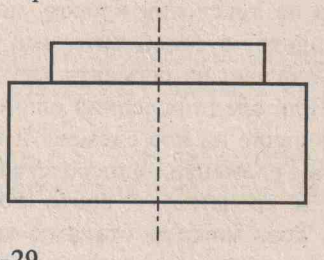
```
OpenSegment(1)
   SetLineType(LT_SOLID)
   SetLineWidth(1.5)
   SetColour(1)
   Rectangle(-50,-25,50,25)
   Rectangle(-40,25,40,35)
•  SetLabel(my_label)
   SetLineType(LT_DASHED)
   SetLineWidth(1)
   Line(0,-28,0,45)
CloseSegment()
```



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 графичния 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 нарича-ме *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 разглеждаме различните физически устройства, разделени в класове съобразно тяхната принадлежност към най-близкия абстрактен клас. Както ще видим по-късно, много рядко една конфигурация съпоставя 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.

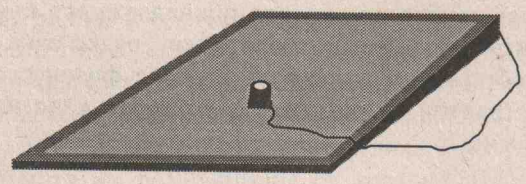
Таблетът е *абсолютно* позициониращо устройство, тъй като предаваните от него координати са тези на собствената му координатна система. Началото на тази координатна система има фиксирано разположение върху повърхността му (долния ляв ъгъл на работното му поле), а осите ѝ са успоредни на правоъгълника 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 позициониращия 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.

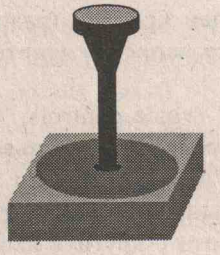
Мишките се отличават по броя на бутоните, като тези с 3 бутона са най-широко използвани, поради факта, че с три бутона могат да се зададат $2^3 - 1 = 7$ различни комбинации.

**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 по-

следното 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 -

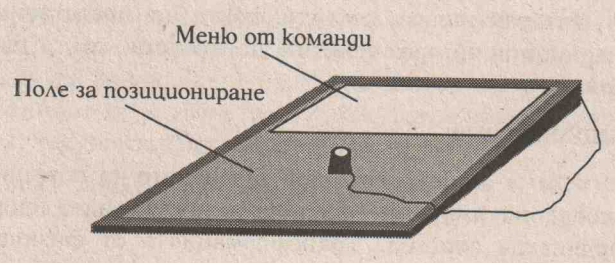
най-много 4096 × 4096. От друга страна първата възможност би усложнила *обратната връзка*, необходимостта от която ще покажем малко по-късно. Удобно решение е да се използва преобразуване до нормирани координати. По този начин, обратната връзка може да се извършва без загуба на точност чрез функциите 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:

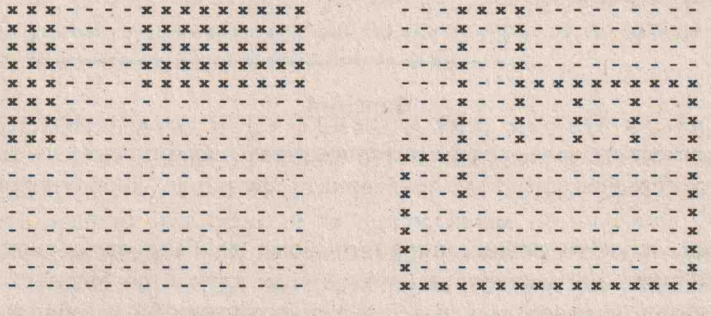
- преди да се прерисува старото положение на локатора и да се изрисува той на новото трябва да се провери дали новата позиция на локатора не е точно същата като старата. Ако това не се прави, показалецът започва да "мига" щом се остави локаторът неподвижен.
- при завършване на графичния вход е възможно 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 по-

лучена 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
    }
};
```



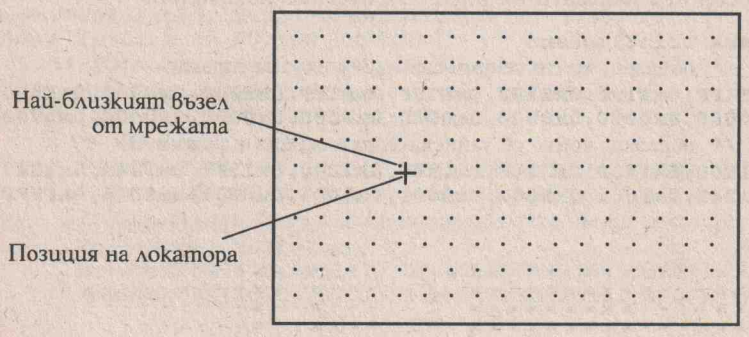
**КООРДИНАТНА МРЕЖА.** Полезно средство за позициониране е използването на координатна мрежа. Вместо текущите координати на локатора, графичната система предава на приложната програма най-близко разположения възел от тази мрежа чрез подходящо закръгляване на тези координати. Някои приложни програми дават 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 цен-

тър 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*.

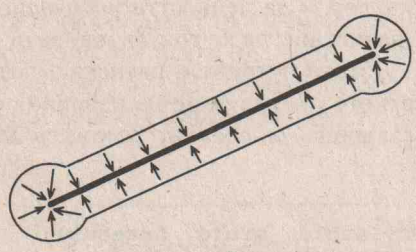
При изготвянето на всеки чертеж много често се налага една отсечка да има за крайна точка някой от върховете на фигура в създадения до момента чертеж. Ако координатите на този връх не са съхранени в някакви променливи и въведената точка не е възел от координатната мрежа, то нейното повторно интерактивно 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

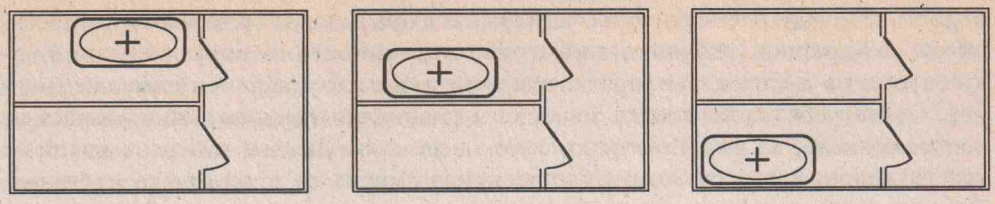
си изясним как се представят равнинните криви в един геометричен модел, на което е посветена част от шеста глава.

**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.



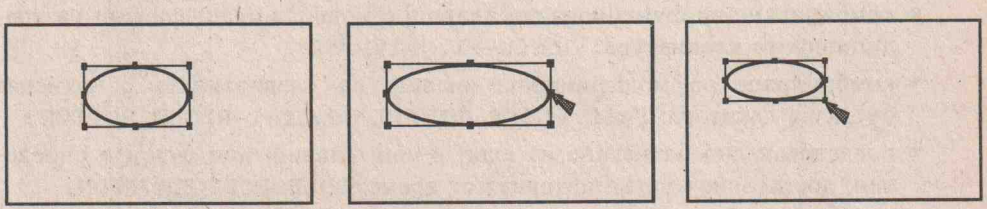
Фиг. 4-6

**2. *"Динамично въртене"* и *"динамично мащабиране"*:** Тези два примера на динамично манипулиране се реализират по същия начин като *влаченето* 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:

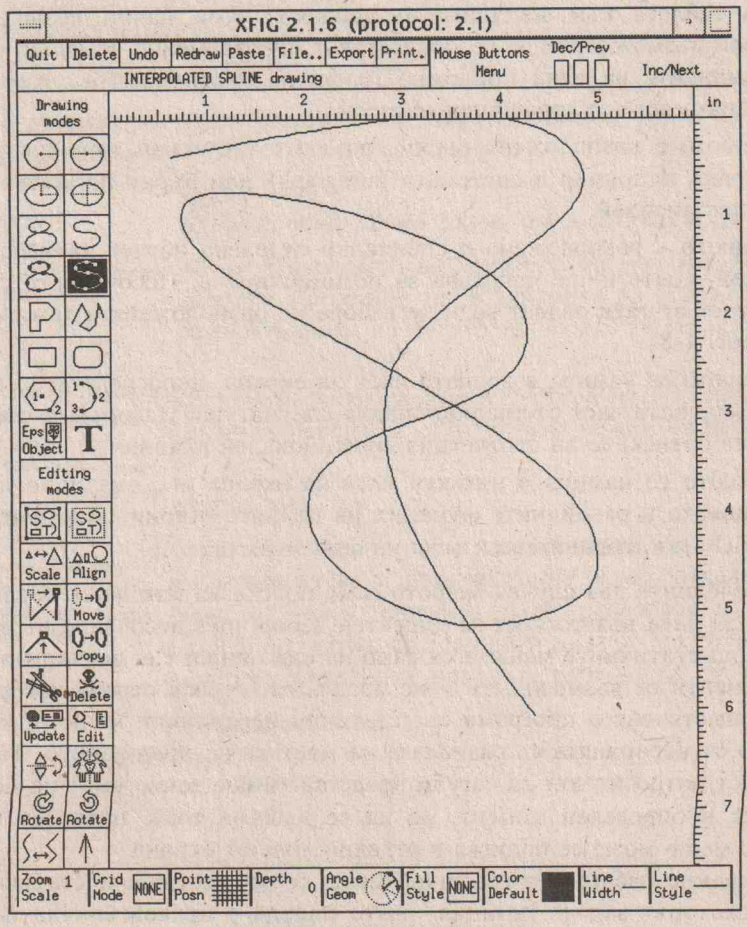
- комбиниране на функционален клавиш с модифициращ клавиш на традиционната клавиатура: `Shift-F1, Alt-PF2`;
- комбиниране на модифициращ клавиш на клавиатурата с натиснат бутон на локатора: `Shift-LEFT_BUTTON, Control-RIGHT_BUTTON;`
- последователно натискане на един и същ клавиш или бутон в определен, 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 на 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 приложна-

та 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";

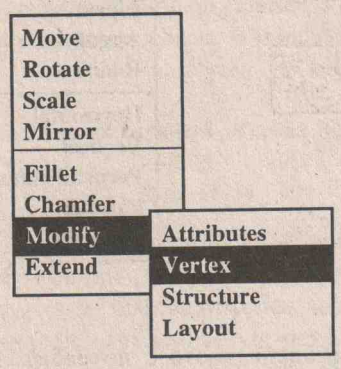
```
SetWritingMode(XOR);
FillRectangleDevice(X, Y+i*hItem ,X+w, Y+(i+1)*hItem);
```

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 структуриране 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.

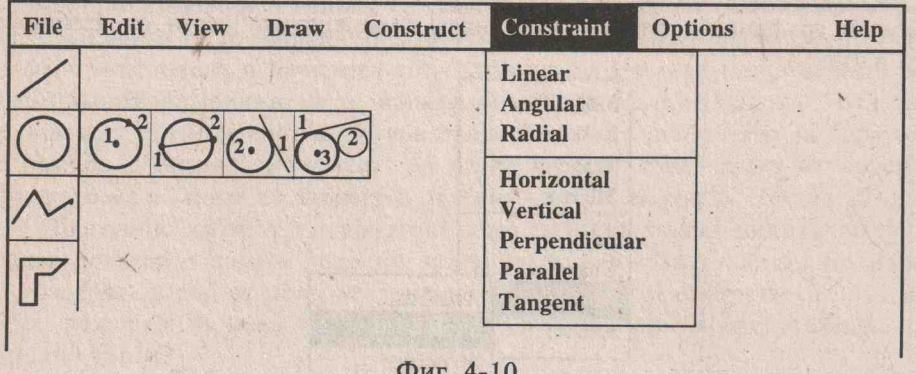



Фиг. 4-9

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 ня-

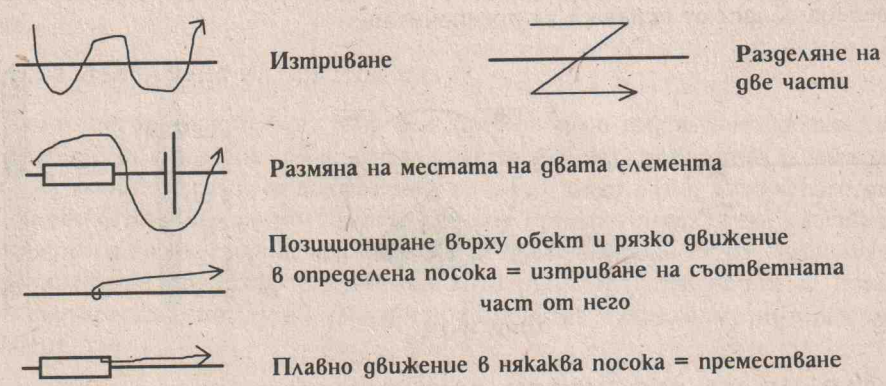
кои от падащите менюта са на две нива, което често затруднява и забавя избора. При твърде сложна система от менюта е желателно потребителят да може сам да ги аранжира или да може да избира между работа с пълно меню и кратко меню (от най-нужните и често използвани възможности).

**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

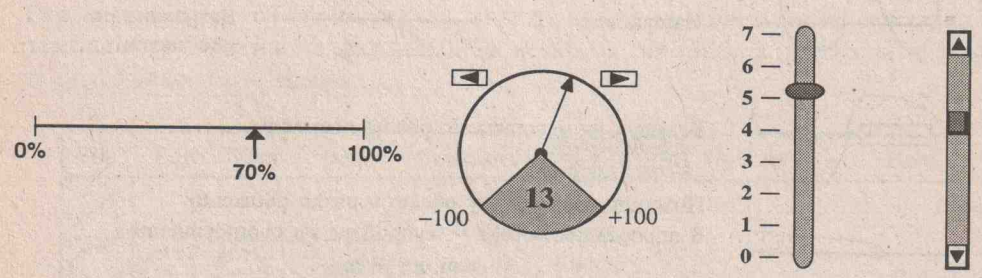
бъдат ограничен брой и да съответстват на естествени движения, характерни за конкретната приложна дейност. Например удобно е да се използват във вид на движения знаците, които употребяват коректорите на текст.

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 *компютрите-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.

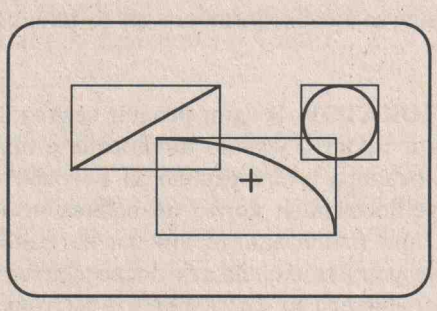
Няколкото начина за осъществяването на посочването на ниво графични примитиви се различават по реализирането на третата и четвъртата стъпки в горната последователност. Търсенето може да бъде:

- *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 най-късото разстояние между въведената точка и характеристичните точки. 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 **xor** 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. Напри-

мер потребителят може да зададе, че ще извършва идентификация само измежду 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

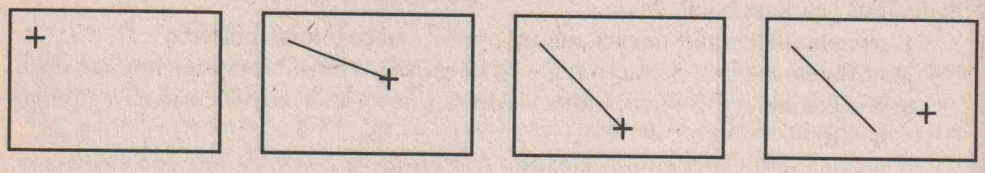
В тази част ще се спрем на някои интерактивни похвати, които обслужват задаването на поредица от точки. Тази дейност в някои системи се осъществява от отделно абстрактно устройство, а в други - от самите приложни програми. Типичен пример за тази дейност е въвеждането на върховете на многоъгълник.

**РАЗТЕГАТЕЛЕН ПРИМИТИВ.** При задаване на една отсечка обратната връзка за позиционирането на втория 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;

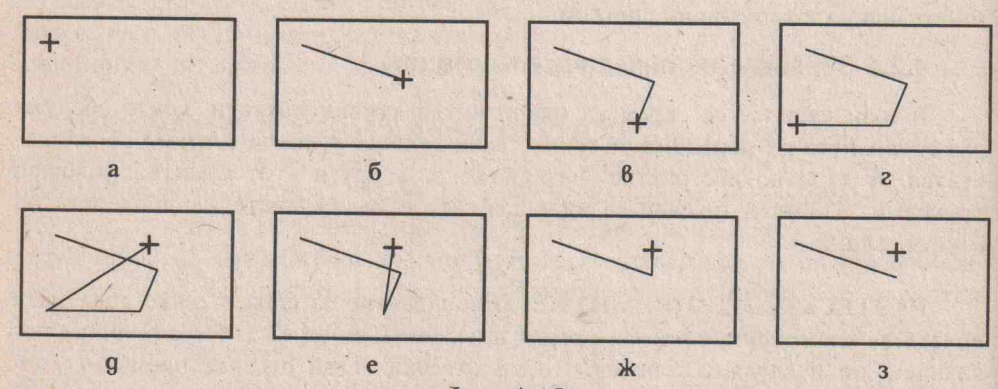
3.След освобождаване на бутона отсечката се фиксира и позиционирането на втората точка завършва (фиг. 4-14г).




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)

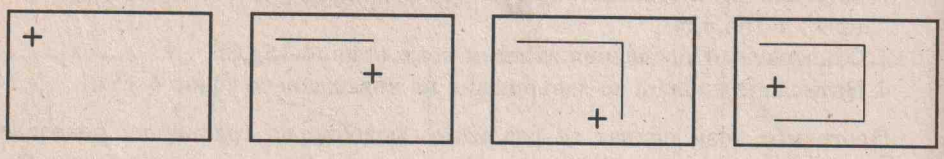
Програмно този похват се реализира подобно на графичния показалец. Ще покажем фрагмент, с които се осъществява *разтегателна отсечка*. След завършване на въвеждането последната визуализирана отсечка съответства на положението на локатора, в което бутонът е бил отпуснат.

```
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);
      }
   }
}
```