Федеральное агентство по образованию
Озерский технологический институт (филиал)
ГОУ ВПО «Московский инженерно-физический институт
(государственный университет)»

Кафедра прикладной математики

Компьютерная графика

Учебное пособие
Часть 1. Вводный курс

Озерск, 2006

УДК 681.3.06
П 56
Пономарев В. В.
Компьютерная графика. Учебное пособие. Часть 1. Вводный курс.
Редакция 2. 09.02.2006.
Озерск: ОТИ МИФИ, 2006. — 122 с., ил.

Учебное пособие предназначено для изучения дисциплины «Компьютерная графика» студентами-программистами. В первой части пособия приведены сведения о свете и цвете, устройстве видеосистемы компьютера, способах получения графических изображений в операционной системе Windows. (Вторая часть пособия описывает преобразования и растровые алгоритмы.) Пособие снабжено большим количеством рисунков и примеров. В конце пособия приведен список литературы для самостоятельного изучения.

Содержание

Введение

Компьютерная графика — это феномен компьютерных технологий. Сегодня невозможно представить область практической деятельности человека, в которой компьютерная графика не применялась бы. Это всевозможные CAD (САПР) системы, предназначенные для визуального интерактивного проектирования, приложения для рисования, обработки фотографий, создания видеороликов, обработки результатов эксперимента, визуализации технологических процессов, картография, геоинформационные системы, наконец — видео, игры и «виртуальная реальность».

Для создания графического изображения используются самые изощренные алгоритмы и самые современные технические усовершенствования. Развитие компьютерных технологий в большей степени есть следствие все возрастающей потребности в графических возможностях компьютерных систем.

С обработкой графической информации на компьютере традиционно связывают три направления:

1) распознавание образов;

2) обработку изображений;

3) машинную графику.

Распознавание образов является одной из самых сложных задач. Изображение, полученное тем или иным способом, преобразуется в символьный, понятный вид, доступный для дальнейшего анализа. Целью распознавания является определение атрибутов изображения, позволяющих классифицировать его как принадлежащий к той или иной категории образов и к тому или иному конкретному образу в этой категории. Примером применения распознавания образов являются известные нам системы оптического распознавания текста.

Обработка изображений оперирует с изображениями на входе и на выходе. Целью является получение того или иного визуального эффекта, такого, как тонирование, цветоделение, удаление шумов, изменение контрастности или яркости и т. п. Дополнительно при обработке изображений применяется их комбинирование с получением нового изображения. Применяется в полиграфии, в кино, на телевидении, в современной живописи, при производстве игр и т. д.

Машинная, или *компьютерная графика* имеет дело с математическими представлениями графических изображений. Изображение, которое можно получить с помощью фотокамеры, не поддается (с небольшими затратами времени) описанию в виде формул и его нельзя синтезировать при помощи программы. Но изображение графика функции, гистограммы доходов, конструкции подъемного крана, а также упрощенного движущегося (мультипликационного) изображения можно достаточно просто описать в виде тех или иных алгоритмов.

В настоящем пособии в очень краткой форме описываются основные положения, связанные с получением изображений неизобразительной природы. Компьютерная графика слишком широкое понятие, чтобы его можно было осветить полностью в одной книге. При составлении пособия принимались во внимание важность конкретного вопроса и доступность (или недоступность) информации по нему. Целью пособия является изложение начальных сведений о формировании компьютерных изображений.

Свет и цвет

Видеть можно только те предметы, которые освещены светом или сами являются его источником. Приступая к изучению компьютерной графики, занимающейся получением изображения на экране монитора, принтере или другом устройстве отображения, прежде всего полезно вспомнить основные положения учения о свете и цвете.

Электромагнитные волны

Свет можно рассматривать как поток частиц и как электромагнитную волну. В самом широком толковании свет — это электромагнитные колебания любых частот, т. е. фотоны любых энергий. Электромагнитная волна представляет собой синусоидальные колебания взаимно перпендикулярных электрического и магнитного полей. На рис. 1 показано изменение электрического ***E*** и магнитного ***H*** полей во времени (пространстве).

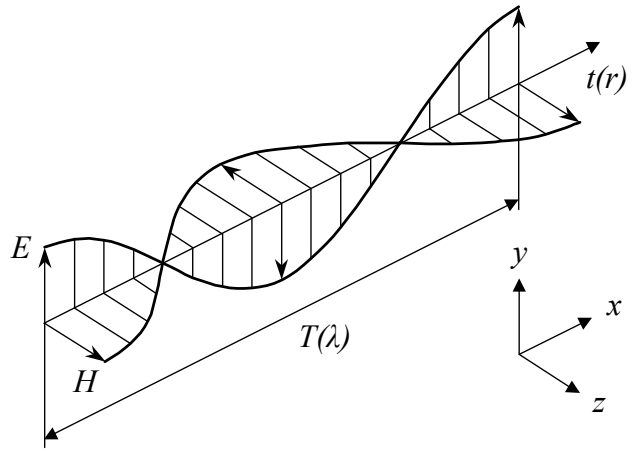


Рис. 1. Распространение электромагнитной волны во времени (пространстве)

Периодом ***T*** называется промежуток времени между двумя последовательными колебаниями, а длиной волны λ — расстояние между ними. Постоянной характеристикой электромагнитной волны является ее частота ***f***, т. е. число колебаний в секунду. Производной характеристикой волны является ее длина λ, которая зависит от скорости распространения ***v*** (1):

(1) $\lambda = v / f$.

Скорость распространения ***v*** зависит от среды, в которой волна распространяется. В вакууме $v = c = 3 \cdot 10^8$м. В веществе длина световой волны изменяется в соответствии с показателем преломления η. Если принять длину волны в вакууме равной λ_0, длина волны в веществе определяется из соотношения (2)

(2) $\lambda = \lambda_0 / \eta$.

Электромагнитный спектр

Электромагнитные волны в принципе могут иметь любую частоту от нуля до бесконечности. Классификация электромагнитных волн по частотам называется *спектром электромагнитных волн* (рис. 2).

Из всего диапазона существующих в природе электромагнитных волн лишь узенький их участок в пределах $\lambda \approx 380..780$ *нм* (*нм* — нанометр; *1 нм* = 10^{-9} *м*)

обладает способностью вызывать ощущение *света*. Будем называть этот участок электромагнитного спектра ***световым диапазоном***.

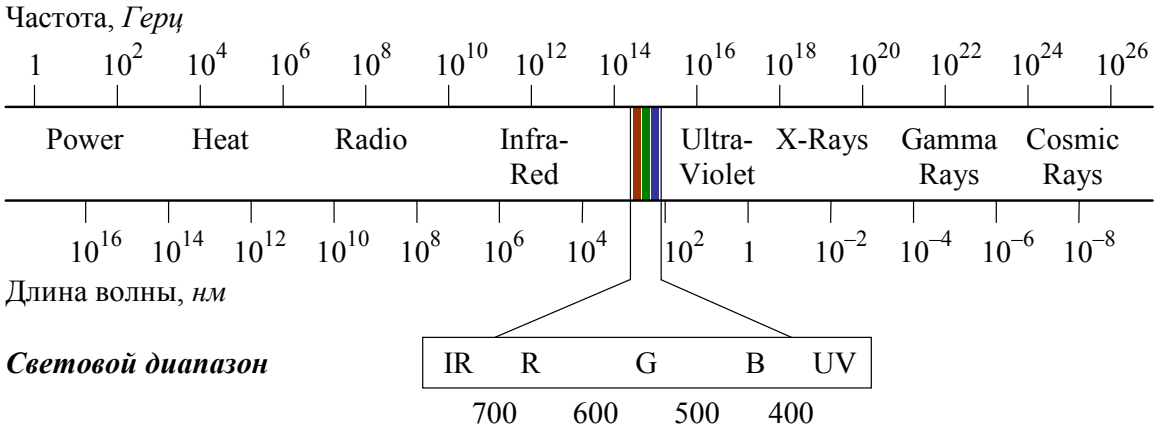


Рис. 2. Электромагнитный спектр

Волны различной длины вызывают ощущение света *различного цвета*. Шкала распределения диапазона световых волн между наиболее хорошо различимыми глазом ***спектральными*** (*содержащимися в солнечном свете*) цветами приведена на рис. 3.

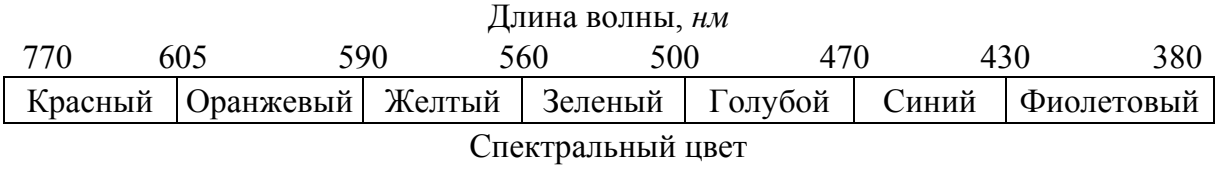


Рис. 3. Спектральные цвета

Спектральные цвета хорошо известны нам из шуточного мнемонического определения цветов радуги: «**К**аждый **О**хотник **Ж**елает **З**нать, **Г**де **С**идит **Ф**азан» (см. также цветную вкладку — главные цвета).

Зрение

Важной особенностью глаза человека является его способность *ощущать электромагнитные колебания*. Световые волны фокусируются при помощи хрусталика на *сетчатке*, покрывающей почти всю внутреннюю поверхность глазного яблока (рис. 4). Сетчатка, состоящая из *рецепторов — колбочек* (***cones***) и *палочек* (***rods***), преобразует световую энергию в химическую и электрическую, формируя *нервные импульсы*. Импульсы передаются в мозг по волокнам зрительных нервов раздельно для левого и правого глаза. Мозг формирует представление о трехмерном изображении, которое здесь же анализируется и вызывает ту или иную реакцию человека.

Изображение, которое мы видим, является *трехмерным*, *яркостным* и *цветным*. Способность зрения различать объем, яркость и цвет определяются разными механизмами.

Объемность предмета, то есть его протяженность по глубине, мы воспринимаем за счет того, что видим предмет с двух позиций (точек зрения), определяемых разнесенностью глазных яблок по ширине. Это дает возможность определить расстояние до различных точек изображения — *глубину точек*.

Яркость изображения формируется сигналами от *палочек*. Палочки сосредоточены в основном на периферии сетчатки. Свет в палочках поглощается пигментом *родопсином*, который является сложным белковым веществом.

Палочки обладают очень высокой светочувствительностью и обеспечивают зрение в условиях очень низкой освещенности. Но они не различают цвета и создают ахроматический (черно-белый) зрительный образ. Народная мудрость отметила эту особенность ночного зрения поговоркой «*ночью все кошки серы*».

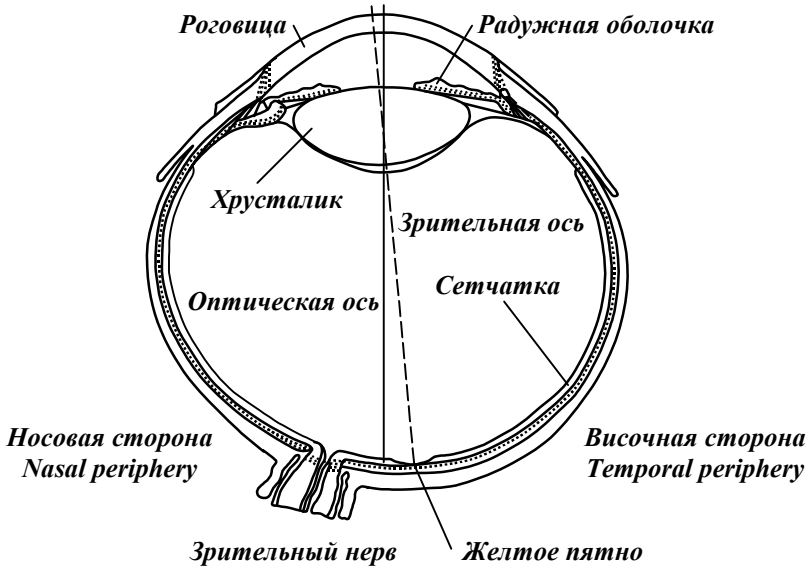


Рис. 4. Разрез правого глаза

Цвет предмета формируется сигналами, поступающими от колбочек, которые различают длину волны. Достоверно принцип их действия неизвестен, хотя их устройство изучено основательно. Не обнаружен пигмент или пигменты, которые поглощают световые волны определенной длины.

Колбочки сосредоточены в основном в центральной части сетчатки, напротив зрительной оси, а в желтом пятне, являющемся центром зрения, присутствуют только колбочки. По степени восприятия электромагнитных волн той или иной длины ***колбочки делятся на три вида***.

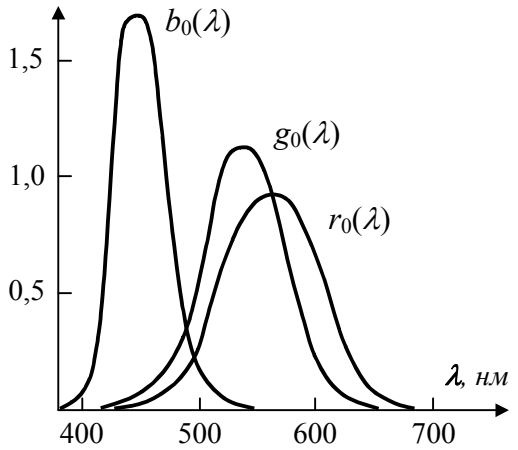


Рис. 5. Спектральная чувствительность трех рецепторов глаза

Часть колбочек реагирует на колебания низкочастотной (красной) части светового диапазона, часть — на колебания среднечастотной (желто-зеленой) части, а часть — на колебания высокочастотной (синей) части. Можно условно назвать

колбочки разных видов приемниками ***красного***, ***зеленого*** и ***синего*** цветов. Наиболее чувствительны «синие» колбочки, «зеленые» колбочки менее чувствительны, а наименьшей чувствительностью обладают колбочки «красного» цвета. Это иллюстрирует рис. 5. Здесь ***r(λ)***, ***g(λ)***, ***b(λ)*** — спектральная чувствительность «красных», «зеленых» и «синих» рецепторов. Цвета «красный», «зеленый» и «синий» называют ***первичными цветами***.

Восприятие цвета глазом интегрируется по длинам волн. Интегральная чувствительность глаза к волнам разной длины называется ***относительной спектральной эффективностью*** света или ***кривой видности*** **V(λ)**. Эта зависимость отображена на рис. 6.

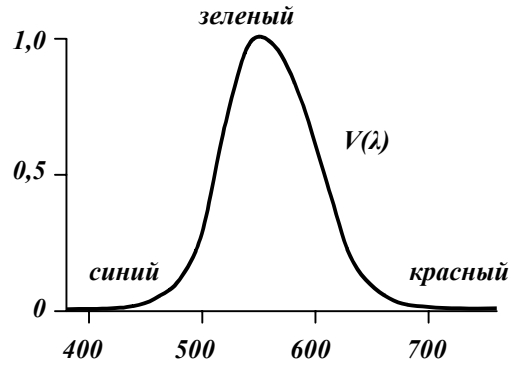


Рис. 6. Относительная спектральная эффективность (для дневного зрения)

Наибольшую эффективность имеют волны желто-зеленого цвета с длиной **555 нм**. Это означает, что при одной и той же мощности светового потока возбуждение, испытываемое глазом, сильнее для волн желто-зеленого цвета. Для ночного зрения кривая видности сдвинута влево примерно на **50 нм** (эффект *Пуркинье*).

Важной характеристикой зрения является способность различать мелкие детали, называемая ***разрешающей способностью***. Она характеризуется минимальным угловым размером предмета, при котором глаз еще отличает этот предмет от других. Разрешающая способность глаза составляет примерно 1 угловую минуту.

Кроме того, зрение обладает ***инерционностью***, когда изображение на сетчатке не исчезает сразу после того, как исчезло световое воздействие, а сохраняется некоторое время. Инерционность глаза составляет от 0,01 с (ощущение последействия как самого действия) до 0,35 с (полное отсутствие последействия).

Измерение света

Качественно и количественно свет характеризуют при помощи *светотехнических*, или *фотометрических* величин.

Количество вещества измеряют массой. Количество света тоже можно определять его массой. Но практически это неудобно, поэтому количество света измеряют его энергией в Джоулях или мощностью в Ваттах. Мощность света называют ***потоком излучения*** или ***лучистым потоком***.

Мощность света определяет его силу. В качестве эталона силы света положено излучение черного тела при температуре затвердевания платины. Сила света одного квадратного сантиметра черного тела при температуре *2042K* принята равной *60 свечам* (канделам — **кд**). От канделы образованы другие фотометри-

ческие единицы. Однако удобнее за основную единицу принять не силу света, а световой поток.

Световой поток Ф — это интенсивность, мощность лучистой энергии света, *видимая глазом энергия*. Точечный источник, излучающий равномерно во все стороны и имеющий силу света ***I***, создает в телесном угле **4π** световой поток ***Ф***, который вычисляется по формуле (3)

(3) $$\Phi = 4\pi \cdot I,$$

а в любом другом телесном угле ***dw*** (4)

(4) $$d\Phi = I \cdot dw.$$

Единица измерения светового потока — *люмен*. **1 лм** — это световой поток, который создает равномерный источник света силой в **1 кд** в телесном угле в **1 ср**. Размерность люмена **кд·ср**.

Связь между *лучистым* и *световым* потоками соответствует особенности зрения по-разному воспринимать электромагнитные волны разной длины (рис. 6). Из опыта мы знаем, что электрическая лампочка мощностью 100 Вт дает больше света, чем лампочка мощностью 40 Вт. Но оценивать силу света по мощности можно только для источников одного типа. Электроплитка с открытой спиралью тоже излучает свет. Однако каждый знает, что лампочка мощностью 40 Вт светит гораздо лучше, чем электроплитка мощностью 1000 Вт. Из-за низкой температуры плитки почти всю энергию она излучает в инфракрасном диапазоне волн, и лишь небольшую часть — в области видимой части спектра. ***Световой поток — это мощность излучения, оцениваемая по ее действию на глаз*** (или другой спектрально селективный приемник).

Для монохроматического (одноцветного) излучения в узком интервале длин волн ***dλ*** световой поток определяется по формуле (5)

(5) $$d\Phi = k_m \cdot V(\lambda) \cdot P(\lambda) \cdot d\lambda,$$

где **P(*λ*)** — спектральная плотность мощности излучения (6)

(6) $$P(\lambda) = \frac{dP}{d\lambda},$$

учитывающая спектральный состав света; k_m — максимальная спектральная эффективность, — световой поток, создаваемый одним Ваттом излучения с длиной волны, для которой **V(*λ*) = 1** (рис. 6). Экспериментально найдено, что k_m **= 683 лм/Вт**.

Для определения светового потока композитного излучения нужно взять интеграл от выражения (5) с пределами интегрирования в диапазоне видимого света или от нуля до бесконечности (7):

(7) $$\Phi = \int k_m \cdot V(\lambda) \cdot P(\lambda) \cdot d\lambda.$$

Для белого цвета с равномерным распределением энергии в видимой части электромагнитного спектра принимают **1 Вт = 220 лм**.

Источник света может иметь неравномерное излучение по разным направлениям. ***Сила света I*** — это пространственная плотность светового потока (8)

(8) $$I = \frac{d\Phi}{dw}.$$

где ***dФ*** — световой поток внутри телесного угла ***dw***.

Сила света измеряется в канделах и имеет размерность **лм/ср**.

Яркость источника света B — это интенсивность свечения его поверхности, непосредственно воспринимаемая глазом. Яркость характеризует силу света по отношению к площади ***S***, которая этот свет излучает (9):

$$B = \frac{I}{S \cdot \cos\gamma}. \quad (9)$$

Здесь γ — угол между нормалью к поверхности и направлением наблюдения. Единица измерения яркости — кандела на квадратный метр (**кд/м^2**).

Описанные величины характеризуют энергию источника света. Чтобы охарактеризовать действие света на окружающие предметы, вводится понятие освещенности.

Освещенность E — это плотность светового потока по освещаемой площади

$$E = \frac{d\Phi}{ds} \cos\gamma. \quad (10)$$

Здесь γ — это угол между нормалью к освещаемой поверхности и направлением светового потока. Единица измерения освещенности — люкс (**1 лк = 1 лм/м^2**). Освещенность экрана в кинотеатре, например, составляет приблизительно **200 лк**.

Если свет падает на поверхность **Δs** по нормали, **Δs = r^2·Δw**. Из (4) следует, что **Δ*Φ* = *I*·Δw**. Подставляя оба выражения в (10), получим (11)

$$E = \frac{I}{r^2} \cos\gamma. \quad (11)$$

Таким образом, освещенность поверхности, перпендикулярной к лучам света, обратно пропорциональна квадрату расстояния до источника света.

Оптические свойства предметов

Большинство окружающих нас предметов не является источниками света, а лишь отражает в той или иной степени падающий на них световой поток $\Phi_{пад}$. Благодаря этому они приобретают некоторую яркость и сами становятся источниками вторичного света (становятся видимыми). Например, белый экран, обладающий равномерным диффузным отражением с коэффициентом отражения ρ, при освещенности ***E*** приобретает яркость **L** (12):

$$L = \frac{E \cdot \rho}{\pi}. \quad (12)$$

Падающий световой поток не только отражается освещаемой поверхностью, но может частично или полностью поглощаться, а в случае прозрачных поверхностей, пропускаться через них. Сумма отраженного $\Phi_{отр}$, пропущенного $\Phi_{пр}$ и поглощенного $\Phi_{погл}$ световых потоков должна быть равна падающему световому потоку $\Phi_{пад}$ (13):

$$\Phi_{отр} + \Phi_{пр} + \Phi_{погл} = \Phi_{пад}. \quad (13)$$

Обозначая общее количество падающего на поверхность светового потока через единицу, можно записать (14)

$$\rho + \tau + \alpha = 1. \quad (14)$$

Здесь через ρ мы обозначили долю отраженного, через τ — долю пропущенного, в через α — долю поглощенного светового потока. Эти *интегральные коэффициенты* описывают отражающие, пропускающие и поглощающие характе-

ристики поверхностей. Селективность свойств поверхности к длине волны учитывается при помощи *спектральных коэффициентов*.

Интегральный коэффициент диффузного отражения определяется соотношением падающего и отраженного световых потоков (15):

$$\rho = \frac{\Phi_{отр}}{\Phi_{пад}}. \quad (15)$$

Для узкого диапазона волн $d\lambda$ отражаемый световой поток определяется по формуле (16)

$$d\Phi_{отр} = k_m \cdot \rho(\lambda) \cdot V(\lambda) \cdot P(\lambda) \cdot d\lambda, \quad (16)$$

где $\rho(\lambda)$ — ***спектральный коэффициент диффузного отражения***.

Для композитного излучения интегральный коэффициент отражения поверхности определяется по формуле (17)

$$\rho = \frac{\int \rho(\lambda) \cdot P(\lambda) \cdot V(\lambda) \cdot d\lambda}{\int P(\lambda) \cdot V(\lambda) \cdot d\lambda}. \quad (17)$$

Интегральный коэффициент пропускания τ определяется аналогичным образом как отношение пропущенного и падающего потоков (18):

$$\tau = \frac{\Phi_{пр}}{\Phi_{пад}}. \quad (18)$$

Для узкого диапазона волн $d\lambda$ пропускаемый световой поток определяется по формуле (19)

$$d\Phi_{пр} = k_m \cdot \tau(\lambda) \cdot V(\lambda) \cdot P(\lambda) \cdot d\lambda, \quad (19)$$

где $\tau(\lambda)$ — ***спектральный коэффициент пропускания***.

Для композитного излучения интегральный коэффициент пропускания определяется по формуле (20)

$$\tau = \frac{\int \tau(\lambda) \cdot P(\lambda) \cdot V(\lambda) \cdot d\lambda}{\int P(\lambda) \cdot V(\lambda) \cdot d\lambda}. \quad (20)$$

Интегральный коэффициент поглощения α можно найти из (14).

Если все спектральные составляющие падающего белого светового потока отражаются, то коэффициент рассеянного отражения от поверхности близок к единице и она имеет белый цвет. Так, например, для мела, снега $\rho \approx 0{,}95$. Если же поглощаются все световые лучи, коэффициент рассеянного отражения от поверхности близок к нулю и она выглядит черной.

Поверхности частично, но равномерно, отражающие и поглощающие все спектральные составляющие белого света, приобретают серые, т. е. в той или иной мере ослабленные белые цвета ($0 < \rho < 1$).

Если поверхность отражает отдельные составляющие спектра белого света, а все остальные поглощает, то *она приобретает цвет отраженного светового потока*. Отражение и пропускание светового потока может принимать одну из четырех форм: направленное (зеркальное), направленно-рассеянное, диффузное и смешанное (рис. 7, рис. 8).

Направленным отражением обладает непрозрачный материал с зеркальной поверхностью. Направленное отражение характеризуется неизменностью

структуры пучка лучей после отражения и равенством углов падения и отражения (рис. 7, а).

При ***направленно-рассеянном*** отражении (рис. 7, б) ось отраженного пучка направлена в соответствии с законом зеркального отражения, но телесный угол отраженного пучка увеличен за счет рассеяния света неоднородностями отражающей поверхности. Яркость поверхности максимальна в направлении отраженного луча и образует расплывчатое световое пятно.

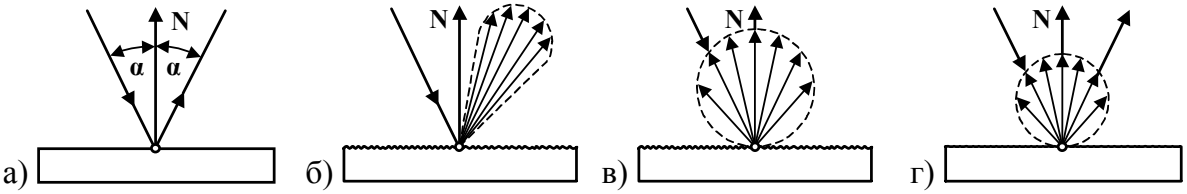


Рис. 7. Виды отражения

Если отраженный свет рассеивается во все стороны с одинаковой интенсивностью, отражение называется ***диффузным*** (рис. 7, в). Материалы, обладающие такими свойствами, называются *равномерными рассеивателями*. Если при этом не происходит поглощения, отражающий рассеиватель называется *совершенным*. Реальные предметы не обладают такими свойствами.

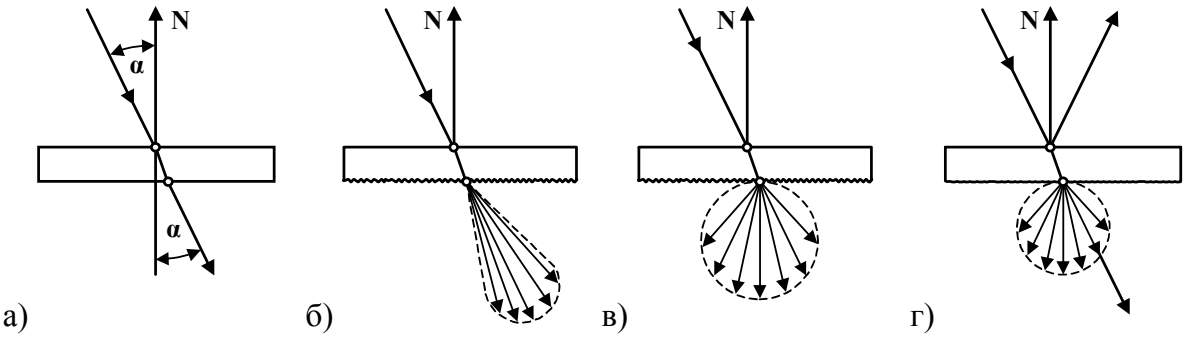


Рис. 8. Виды пропускания

Большинство реальных поверхностей обладают свойствами как зеркального, так и рассеянного отражения. При ***смешанном*** отражении (рис. 7, г) часть светового потока отражается зеркально, а оставшаяся часть рассеивается более или менее равномерно по всем направлениям. Поверхности, приближающиеся по своим свойствам к зеркальным, называют ***глянцевыми***, а более близкие к диффузным — ***матовыми***.

Направленное пропускание (рис. 8, а) характерно для прозрачных материалов (стекла и пластмассы), направленно-рассеянное пропускание (рис. 8, б) характерно для прозрачных материалов с одной или двумя рассеивающими поверхностями (матированные стекла и пластмассы). Диффузным и смешанным пропусканием (рис. 8, в, г) обладают материалы, имеющими оптическую неоднородность. Отражающие и пропускающие свойства в значительной степени также зависят от того, под каким углом к поверхности предмета падает световой поток.

При наблюдении предметов существенную роль играет диапазон яркостей в поле зрения, который принято называть ***контрастностью*** **К** и представлять в виде отношения яркости наиболее светлого элемента изображения $\boldsymbol{B}_{\max}$ к яркости темного элемента $\boldsymbol{B}_{\min}$ (21),

(21) $$\mathbf{K} = \frac{\boldsymbol{B}_{\max}}{\boldsymbol{B}_{\min}}.$$

$B_{\min}$

Обычно $\mathbf{K \leq 100}$, но иногда может достигать и нескольких тысяч.

Трехмерность цвета

Представим себе следующий опыт. Есть несколько карточек, которые отражают свет одинаково во всем спектральном диапазоне, но с разным коэффициентом диффузного поглощения ρ. Такими свойствами обладают карточки с различными оттенками серого цвета, от белого до черного (рис. 9).

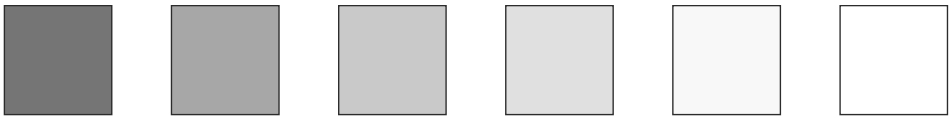


Рис. 9. Серые карточки

Попробуем разложить карточки в какой-нибудь разумной последовательности. Можно предположить, что карточки будут разложены по степени своей яркости (неважно, от белой к черной, или от черной к белой). Яркость — это скалярная величина, а глаз человека легко распознает яркость.

Перейдем ко второму опыту (см. цветную вкладку). Пусть карточки цветные, то есть обладающие разными коэффициентами спектрального отражения $\rho(\lambda)$. Разложить цветные карточки намного труднее. Однако сначала мы сможем разделить их на группы по принципу радуги — фиолетовые, синие, голубые, зеленые, желтые, оранжевые и красные. В каждой группе окажется несколько карточек, имеющих одинаковый цветовой тон.

Далее можно заметить, что карточки в группе различаются по степени насыщенности цвета. Например, есть зеленая карточка, а есть зеленоватая. Такую карточку стоит положить не в ряд разных цветов, а ниже зеленой. Таким образом цветные карточки займут плоскость.

Это еще не все. Останутся еще карточки, которые не отличаются от разложенных ни по цветовому тону, ни по насыщенности цвета. Они светлее или темнее, то есть более или менее яркие. Эти карточки можно положить поверх остальных.

Таким образом, для классификации цветов понадобилось расположить карточки в пространстве. Иначе говоря, цвет трехмерен, и координатами цвета являются ***цветовой тон***, чистота или ***насыщенность*** цвета, и ***яркость***. Это те характеристики цвета, которые мы ощущаем *непосредственно*.

Цветовой тон — это свойство светового потока, которое позволяет отличить его по цвету от других световых потоков. Цветовой тон характеризуется преобладающей, доминирующей в данном световом потоке длиной волны $\lambda_д$. Например, для светового потока красного спектрального цвета принято $\lambda_д$ = 700 нм, для зеленого — 546 нм, для синего — 436 нм. Пурпурные цвета (сиреневый, вишневый и т. п.), которые не являются спектральными, а образуются в результате смешения красного и синего цветов, характеризуются длинами волн их дополнительных цветов.

Глаз способен различать до 150–180 оттенков спектральных цветов (цветовых тонов) и около 40 *пурпурных цветовых тонов* (число различаемых цветовых тонов очень сильно варьируется в зависимости от тестирующего. Так, люди, большую часть жизни занимающиеся сравнением и анализом цветов, могут

различать до 800 и более, до нескольких тысяч, цветовых оттенков. Обычный человек наверняка может различить от 9 до 30 цветов).

Если на глаз одновременно воздействуют все спектральные цвета, имеющие примерно равные энергии, то создается ощущение *белого цвета*. Такое же ощущение может быть получено при воздействии на глаз только двух, но вполне определенных, цветов. Эти два цвета, создающие при смешении ощущение белого цвета, называют *дополнительными* (*complementary*). Для каждого данного цвета существует свой дополнительный цвет, например, для желтого дополнительным служит синий цвет, для оранжевого — голубой, для зеленого — пурпурный и т. д.

Насыщенность *или* ***чистота цвета*** — это степень свободы цвета от примеси белого света. Световые потоки одного и того же цветового тона, в зависимости от примеси белого света, могут иметь различную насыщенность. При этом цветовой тон не меняется, а создает лишь впечатление более блеклой окраски. Количественно насыщенность оценивается чистотой цвета **p**, которая устанавливает относительное содержание светового потока $\boldsymbol{\Phi_д}$ чистого (спектрального) цвета в световом потоке $\boldsymbol{\Phi_0}$ того же цветового тона (22):

$$\mathbf{p} = \frac{\boldsymbol{\Phi_д}}{\boldsymbol{\Phi_0}} = \frac{\boldsymbol{\Phi_д}}{(\boldsymbol{\Phi_д} + \boldsymbol{\Phi_б})}, \qquad (22)$$

где $\boldsymbol{\Phi_б}$ — световой поток белого цвета ($\boldsymbol{\Phi_б} = \boldsymbol{\Phi_0} - \boldsymbol{\Phi_д}$).

Для чистого спектрального цвета: $\boldsymbol{\Phi_б} = \mathbf{0}$ и $\mathbf{p} = \mathbf{1}$; для белого цвета: $\boldsymbol{\Phi_д} = \mathbf{0}$ и $\mathbf{p} = \mathbf{0}$. Следовательно, чем ближе значение чистоты цвета **p** к единице, тем больше насыщенность.

Цветовой тон и насыщенность определяют ***цветность*** светового потока. ***Яркость*** определяется яркостью источника света.

Так как яркость может принимать значения от нуля до бесконечности, вместо яркости для несамосветящихся объектов часто используют интегральный коэффициент диффузного отражения ρ.

Цветовой круг Ньютона

Первая система представления цветов была предложена Ньютоном в 1669 г.

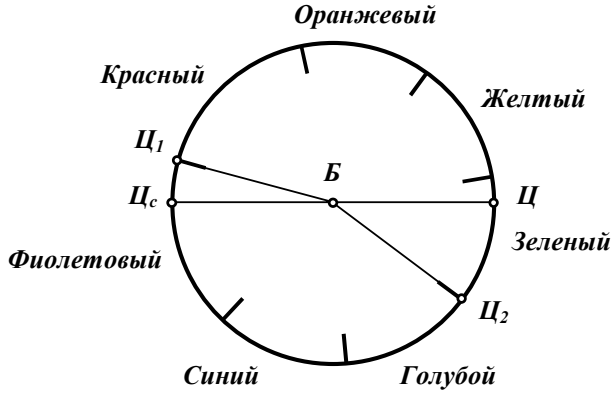


Рис. 10. Цветовой круг И. Ньютона

Всю гамму спектральных и пурпурных цветов (всего 7 цветов) он разместил по окружности (рис. 10, см. также цветную вкладку). В центре круга находится белый цвет ***Б***. Цвета, представленные точками на прямой, соединяющей точку спектрального цвета (на внешней окружности) ***Ц*** с центром, получаются сложе-

нием цветов ***Ц*** и ***Б*** (на рисунке это зеленые цвета). По мере перемещения точки на этой прямой к центру круга доля энергии белого цвета в смеси цветов возрастает (насыщенность цвета уменьшается).

Если продолжить прямую ***ЦБ*** до пересечения с противоположной стороной внешней окружности, получим точку $Ц_c$, цвет которой будет дополнительным к цвету ***Ц***. Для всякого цвета имеется дополнительный цвет, в смеси с которым (в определенной пропорции яркостей) данный цвет дает ахроматический (серый) световой поток.

Неспектральные цвета также имеют свои дополнительные цвета, расположенные на противоположной стороне прямой, проведенной из точки цвета через центр круга. На рис. 11 два цвета $Ц_3$ и $Ц_4$ являются взаимодополнительными: $Ц_3 + Ц_4 = Б$.

Если смешивать два спектральных цвета $Ц_5$ и $Ц_6$, то получаемый цвет смеси $Ц_7 = Ц_5 + Ц_6$ будет представлен точкой, которая находится между смешиваемыми цветами на прямой, их соединяющей. Например, при смешивании красного и зеленого цветов образуется желтый цвет, при смешивании зеленого и синего — голубой. Положение точки на прямой (цвет смеси) определяется яркостями складываемых цветов.

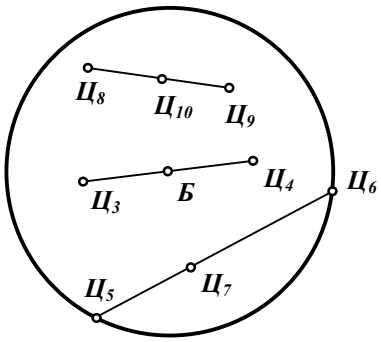


Рис. 11. Сложение цветов на цветовом круге Ньютона

Так же смешиваются и неспектральные цвета: $Ц_{10} = Ц_8 + Ц_9$[1].

Если на цветовом круге провести радиусы до точек $Ц_1$ и $Ц_2$, то в верхней части круга будут находиться теплые цвета (*красный, оранжевый, желтый и зеленый*), а в нижней — холодные (*голубой, синий, фиолетовый*).

Природа цвета

Цвет — это феномен, присущий зрительной системе человека. Цвета как такового, как физической сущности, в природе не существует. Ни свет, ни окружающие нас предметы не обладают цветом, они лишь излучают *свет определенного спектрального состава*. Цвет сложным образом формируется в мозге человека из возбуждений, поступающих от фоторецепторов глаза — колбочек. Механизм этого цветового восприятия до конца не изучен. В сетчатке глаза сосредоточено примерно 127 млн. рецепторов (120 млн. колбочек и 7 млн. палочек), которые передают информацию в мозг посредством примерно 1 млн. нервных волокон от каждого глаза. Следовательно, в сетчатке глаза происходит первичная обработка зрительной информации. Далее нервные волокна

[1] Для получения суммы цветов Ньютон предложил прикрепить к их местам на периферии нити с грузами, пропорциональными интенсивностям цветов. Суммарный цвет будет лежать в точке круга, к которой приложена равнодействующая сил.

перекрещиваются в хиазме и попадают в два зрительных центра мозга. Зрительные впечатления и даже просто свет оказывают воздействие на всю нервную систему человека. Подчеркивая тесную связь глаза с нервной системой, сетчатку нередко называют частью мозга, вынесенной на периферию.

Тот факт, что цвет не является физической сущностью, подтверждает существование людей с нарушениями цветового зрения. Эти нарушения проявляются в том, что колбочки одного, двух или всех трех видов у них не работают, что приводит к пропаданию части цветной составляющей картинки. Кроме этого, встречаются люди, у которых возбуждение рецепторов какой-нибудь группы ослаблено. Эти люди не признают общепринятых цветовых равенств, хотя различение цветов по спектру у них сохранено.

Цвет — это ощущение, вызываемое волнами разной длины. И хотя само ощущение имеет физическую природу (нервные импульсы суть химические и электрические процессы), цветовое восприятие — результат сложных преобразований в мозге.

Рис. 12. Смешение цветов в опыте Т. Юнга

Трехмерность цвета происходит из самого процесса возникновения цветовых ощущений, основанного на фоторецепторах трех видов. М. В. Ломоносов первым высказал мысль, что в глазе находятся три вещества, возбуждаемые тремя разными участками видимого света. Трехкомпонентная теория была развита Томасом Юнгом (1773—1829). Ньютон показал, что разные цвета можно получать смешиванием. Юнг ввел понятие трех основных цветов. Взяв три проекционных фонаря с красным, зеленым и синим светофильтрами, он направил их на белый экран так, что чтобы проекции кругов частично перекрывались. При наложении света от красного и синего фонарей получился пурпурный цвет, от красного и зеленого — желтый, от зеленого и синего — голубой, а в центре, при сложении всех трех цветов — белый (рис. 12, см. также цветную вкладку).

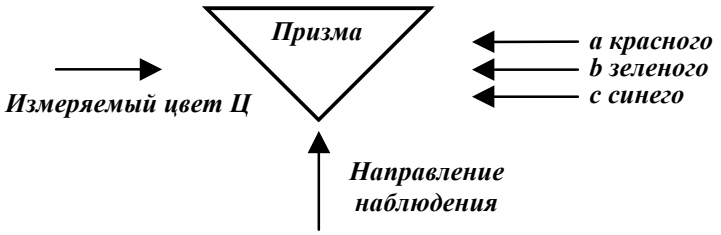


Рис. 13. Схема колориметра

Д. К. Максвелл (1831—1879) построил первый *колориметр*, в котором измеряемый цвет освещал одну половину поля зрения, а другая освещалась смесью излучений красного, зеленого и синего света (рис. 13). Изменяя интенсивность излучений, можно установить равенство цвета обоих полей прибора. После это-

го, используя интенсивности излучений красного ***a***, зеленого ***b*** и синего ***c*** цветов, можно написать уравнение для измеряемого цвета **Ц** (23):

(23) **Ц** = ***a*****К** + ***b*****З** + ***c*****С**.

Графически смешение цветов Максвелл изобразил в виде цветового треугольника (рис. 14, см. также цветную вкладку).

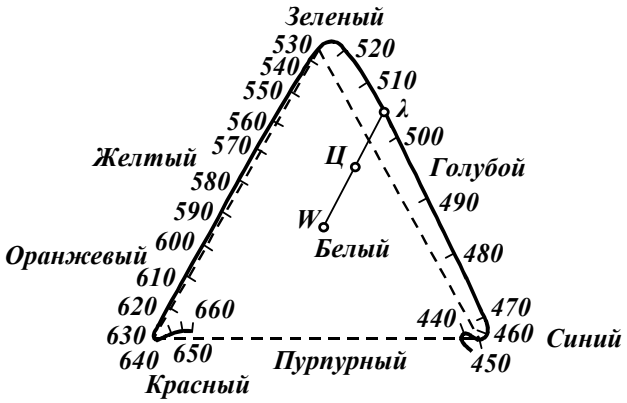


Рис. 14. Цветовой треугольник Максвелла

В вершинах треугольника находятся излучения $\lambda_к$=630 нм, $\lambda_з$=528 нм, $\lambda_с$=457 нм. Цвет, получающийся в результате сложения интенсивностей ***a***, ***b*** и ***c***, можно найти по принципу центра тяжести, как в круге Ньютона. При равенстве интенсивностей ***a*** = ***b*** = ***c*** получается белый цвет ***W*** в центре треугольника.

Спектральные цвета лежат на сплошной линии за пределами треугольника. Используя положительные количества излучений ***a***, ***b***, ***c***, можно получить только цвета, лежащие внутри треугольника или на его сторонах. Для получения спектральных цветов отдельные основные цвета вычитаются (подаются на призму с другой стороны). Если провести прямую из центра ***W*** через любую точку ***Ц*** до пересечения с кривой спектральных цветов, получим длину волны ***λ*** цвета ***Ц***. Чем дальше точка ***Ц*** от центра, тем больше чистота цвета ***Ц***. Яркость цвета определяется абсолютными значениями ***a***, ***b***, ***c***.

Колориметрия

Колориметрия — это наука об измерении цвета, базирующаяся на законах смешения цветов Х. Грассмана, который в 1853 г. занимался изучением цветового круга Ньютона. Сущность этих законов следующая.

Первый закон Грассмана. Для определения цвета требуется три величины. Такими величинами могут быть, например, цветовой тон, насыщенность и яркость.

Второй закон Грассмана. Цвет изменяется непрерывно при изменении количеств составляющих его цветов.

Третий закон Грассмана. Два одинаковых цвета, каким бы способом они не были получены, при смешении всегда дают один и тот же цвет.

Из этого закона следуют два вывода:

а) Цвет смеси нескольких излучений определяется цветами этих излучений, а не их спектральными составами (***метамеризм цвета***);

б) Если цвет $Ц_1$ согласован (создает равное визуальное ощущение) с цветом $Ц_2$ и цвет $Ц_3$ согласован с цветом $Ц_4$, то смесь цветов $Ц_1$ и $Ц_2$ даст цвет $Ц_1$

(можно также утверждать, что цвет смеси $\boldsymbol{Ц_2}$), а смесь $\boldsymbol{Ц_1}$ и $\boldsymbol{Ц_3}$ согласуется с цветом смеси $\boldsymbol{Ц_2}$ и $\boldsymbol{Ц_4}$.

Это можно записать *цветовыми тождествами*:

если $Ц_1 \equiv Ц_2$ и $Ц_3 \equiv Ц_4$, то

$Ц_1 + Ц_2 \equiv Ц_1$; $Ц_1 + Ц_2 \equiv Ц_2$; $Ц_3 + Ц_4 \equiv Ц_3$; $Ц_3 + Ц_4 \equiv Ц_4$;

$Ц_1 + Ц_3 \equiv Ц_2 + Ц_4$;

$k_1Ц_1 \equiv k_1Ц_2$; $k_2Ц_3 \equiv k_2Ц_4$,

где k_1 и k_2 — константы, если спектры излучений не изменяются при умножении на k_1 и k_2.

Четвертый закон Грассмана. При смешении цветов их количества складываются.

Обозначим через (R), (G), (B) количество (яркость) основных цветов, принятые за единицу. Обозначим количество этих единиц в цвете $Ц$ через R, G, B. Тогда цвета $Ц_1$ и $Ц_2$ можно представить цветовыми тождествами:

$Ц_1 \equiv R_1(\mathrm{R}) + G_1(\mathrm{G}) + B_1(\mathrm{B})$; $Ц_2 \equiv R_2(\mathrm{R}) + G_2(\mathrm{G}) + B_2(\mathrm{B})$,

а цвет их смеси

$Ц_1 + Ц_2 \equiv (R_1 + R_2)(\mathrm{R}) + (G_1 + G_2)(\mathrm{G}) + (B_1 + B_2)(\mathrm{B})$.

Цветовое пространство

На основании первого и второго законов смешения цвет можно представить точкой в трехмерном пространстве. Использование нереальных цветов позволяет получить взаимно однозначное соответствие цветов и точек этого пространства. Каждому цвету соответствует некоторая точка цветового пространства, а каждой точке — определенный цвет, реальный или нереальный.

Цветовое пространство является аффинным. В аффинном пространстве, в отличие от евклидова, нет понятий угла и длины, хотя понятие параллельности сохраняется. В аффинном пространстве можно сравнивать отрезки на одной прямой, или на параллельных прямых.

Для выполнения математических действий с цветами в цветовом пространстве нужно ввести координатную систему. Можно задать сколь угодно координатных систем. Началом координат в них выбирается точка $\boldsymbol{B}$, соответствующая черному цвету (отсутствие цвета). Координатные оси задаются единичными точками, в качестве которых выбирают три основных цвета, называемых ***цветовыми стимулами***. Масштабы координатных осей задаются косвенно при помощи четвертого, опорного для данной цветовой системы цвета, называемого ***исходным*** (рис. 15, точка $\boldsymbol{W}$). Это цвет, получающийся при сложении цветовых стимулов с единичными количествами (белый цвет). Координатные оси задаются произвольно, однако должна выполняться их линейная независимость, — ни один из цветовых стимулов не может быть выражен через два других.

После того, как исходный цвет определен, цветовое пространство упорядочивается. Для задания цвета используются тройка цветовых координат *(L, M, N)*. Исходный цвет имеет координаты $\boldsymbol{W}(1, 1, 1)$, а основные цвета (цветовые стимулы) — $\boldsymbol{L}(1, 0, 0)$, $\boldsymbol{M}(0, 1, 0)$ и $\boldsymbol{N}(0, 0, 1)$. Единичные основные цвета обозначают теми же буквами, что и координаты, заключенными в скобки. Выражение для произвольного цвета $\boldsymbol{Ц}$ записывается в виде

(24) $\qquad \boldsymbol{Ц} = \boldsymbol{L}_{Ц}(\mathrm{L}) + \boldsymbol{M}_{Ц}(\mathrm{M}) + \boldsymbol{N}_{Ц}(\mathrm{N})$.

Прямые, исходящие из начала координат, являются геометрическими местами цветов одной цветности, но разной чистоты. Чем дальше от центра расположена точка цвета на прямой, чем выше насыщенность (чистота) цвета.

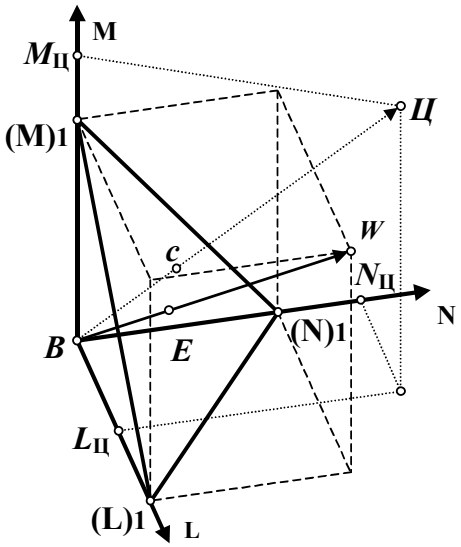


Рис. 15. Цветовое пространство

Количество света в колориметрии определяется суммой координат цветового вектора ***T*** (25), или яркостью ***Y*** (26):

(25) $$T = L_Ц + M_Ц + N_Ц.$$

Яркость ***Y*** является световой мерой количества света. Она связывает колориметрию с фотометрией, и определяется через координаты цвета и через яркостные коэффициенты L_L, L_M, L_N:

(26) $$Y = L_L \cdot L_Ц + L_M \cdot M_Ц + L_N \cdot N_Ц.$$

В системе ***RGB*** нормализованная яркость определяется по формуле (27).

(27) $$Y = 0{,}299 \cdot R + 0{,}587 \cdot G + 0{,}114 \cdot B.$$

Плоскость T=1 называется единичной плоскостью цветового пространства. Она проходит через точки основных цветов на координатных осях. Для единичной плоскости сумма координат равна 1 (28):

(28) $$L + M + N = 1.$$

Цветовой треугольник

Для колориметрических расчетов используют единичную плоскость, так как в большинстве случаев яркость не нужна. Представление единичной плоскости называют ***цветовым треугольником*** или ***графиком цветности***. Цвета, расположенные на единичной плоскости, имеют равную яркость (равное количество). Единичная плоскость является одним из геометрических мест всех возможных цветовых тонов.

На рис. 16 приведен цветовой график, в котором в качестве основных выбраны красный ***R***, зеленый ***G*** и синий ***B*** цвета. Оси координат пересекают единичную плоскость в вершинах равностороннего треугольника. Кривая, проходящая через монохроматические цвета и соответствующая 100% насыщенным спектральным цветам, называется ***локусом***.

Любой цвет ***c*** может быть представлен в виде суммы трех основных цветов, взятых в определенных количествах: $c = r\mathbf{R} + g\mathbf{G} + b\mathbf{B}$, где координаты ***r***, ***g***, ***b*** определяют относительные количества основных цветов, а **R**, **G**, **B** являются единичными количествами основных цветов.

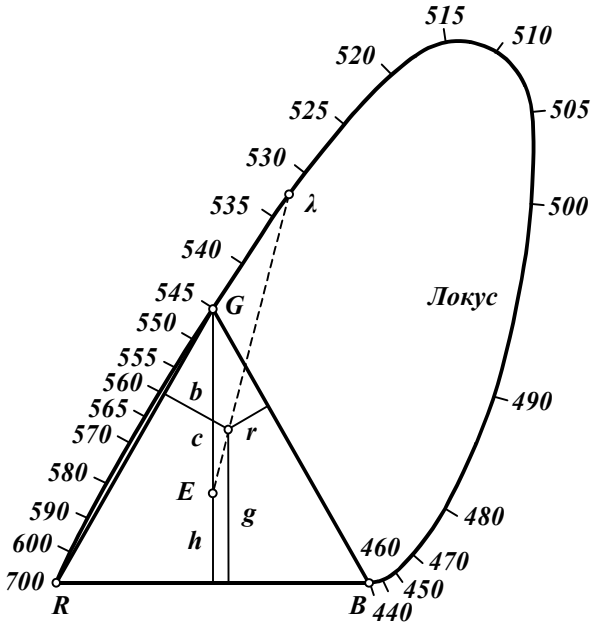


Рис. 16. Цветовой треугольник RGB внутри локуса

Координаты цвета рассчитываются как отношения отрезков ***r***, ***g***, ***b*** к высоте треугольника ***h***, которая принимается за единицу. Для определения доминирующей длины волны ***λ*** цвета ***c*** нужно провести прямую из точки ***E*** через точку ***c*** до пересечения с локусом. Относительное положение точки ***c*** на прямой ***Eλ*** определяет чистоту цвета. При этом точка ***λ*** соответствует чистоте, равной 1, а точка ***E*** — чистоте 0.

Каждой единице цвета соответствует своя яркость, определяемая ее *яркостным коэффициентом* L_X. Для получения относительной яркости цвета нужно сложить яркости составляющих основных цветов:

(29) $$L = L_R r + L_G g + L_B b,$$

где яркостные коэффициенты находятся в соотношении

(30) $$L_R : L_G : L_B = 1 : 4{,}5907 : 0{,}0601.$$

Яркостные коэффициенты показывают количества основных цветов, дающие при смешении белый (равно-энергетический) цвет.

Как следует из цветового графика, большая область цветов в системе **RGB** может быть задана, только если одна или две координаты являются отрицательными, поэтому система **RGB** имеет ***неполный цветовой охват***.

Международной комиссией по освещению (МКО) утверждены для использования и другие цветовые схемы, одной из которых является система **XYZ**. В этой системе основные цвета подобраны таким образом, чтобы получить полный цветовой охват. При этом сами цвета **X**, **Y**, **Z** не являются реальными, а соотношение яркостных коэффициентов выбрано $L_X : L_Y : L_Z = 0 : 1 : 0$ (яркость определяется только координатой Y).

Цилиндрическая цветовая система координат

На практике часто используются и другие системы задания цвета, например, цилиндрическая (рис. 17, см. также цветную вкладку). В ней цветность также образует плоскость, но цветовой тон задается точкой на окружности, соответствующей спектрально чистым цветам, как в цветовом круге Ньютона. Цветовой тон при этом может измеряться в градусах. Насыщенность или чистота цвета изменяется по радиусу от нуля в центре круга до единицы на окружности. Яркость определяется высотой точки над плоскостью цветности.

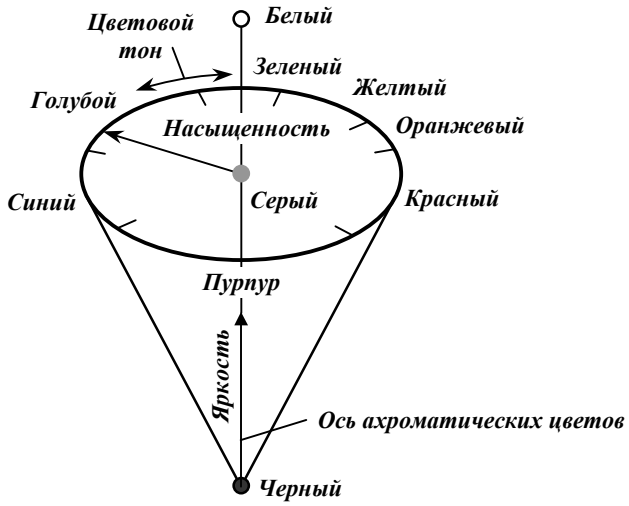


Рис. 17. Цилиндрическая цветовая система координат

Примером цилиндрической системы координат служит система ***HSB*** (***Hue*** — цветовой тон, ***Saturation*** — насыщенность, ***Brightness*** — яркость). Эта система задания цвета является одной из альтернативных в приложении ***Paint*** операционной системы ***Windows***. Существуют и другие цилиндрические системы.

Аддитивное смешение цветов

Аддитивное сложение (смешение) цветов происходит, когда источниками цвета являются излучатели света — проекторы, мониторы. Признаком аддитивного сложения цветов является наложение цвета на черное (на черный монитор, на черноту зала при проецировании на белый экран). Смешение называется аддитивным потому, что составляющие цвета складываются. Яркость результирующего цвета при этом повышается, так как яркость — это сумма цветовых координат. Результат сложения основных цветов приведен в таблице ниже.

Табл. 1. Сложение цветов при аддитивном смешении

Смешиваемые цвета	***Цвет смеси***
Красный + зеленый	Желтый
Красный + синий	Пурпурный
Синий + зеленый	Голубой
Красный + синий + зеленый	Белый

Аддитивное смешение цветов исследовалось Ньютоном. Законы Грассмана также описывают аддитивное смешение. Примером аддитивного сложения является система ***RGB***, используемая в телевидении и в компьютерных видеосистемах. Большинство цветовых систем являются аддитивными.

Способы аддитивного смешения цветов

До сих пор говоря о смешении, мы подразумевали ***одновременное смешение цветов***, когда световые потоки смешиваемых цветов наблюдатель видит одновременно. Возможно ***поочередное смешение цветов***, когда световые потоки смешиваемых цветов воздействуют на глаз наблюдателя поочередно, а не одновременно. При достаточно высокой частоте смены этих потоков, вследствие инерционности зрения, создается такое же впечатление от смеси, как и при одновременном смешивании.

Можно также получить ***пространственное смешение цветов***, когда наблюдаемая поверхность покрывается мелкими разноцветными пятнами или полосками, и с достаточно большого расстояния кажется не пестрой, а однородной вследствие недостаточной разрешающей способности глаза. Пространственное смешение цветов применяется в живописи (*пуантилизм*), текстильной промышленности, цветном телевидении.

Известно еще ***бинокулярное смешение цветов***, когда на один глаз наблюдателя воздействует световой поток одного цвета, а на другой — другого. В зрительном аппарате возникает ощущение цвета их смеси. Бинокулярное смешение применяется в стерео-цветном кино и телевидении.

Субтрактивное смешение цветов

Субтрактивное смешение цветов происходит, когда составляющие цвета накладываются на белое, например, при печати цветного изображения на бумаге. При субтрактивном смешении цвета вычитаются из белого. Очень часто из белого вычитаются первичные цвета — синий, зеленый и красный (например, в цветной фотографии). При цветной печати чаще вычитаются цвета, дополнительные к первичным.

Для вычитания цветов используют светофильтры, поглощающие один из спектральных цветов. Если из белого цвета вычесть, например, желтый (при помощи желтого светофильтра), то синий цвет (дополнительный к желтому, см. цветовой круг Ньютона) будет поглощен. Если на белую бумагу нанести слой желтой краски, она поглотит синие длины волн, останутся только красные и зеленые (рис. 18, а). Если теперь на слой желтой краски нанести слой голубой, она поглотит красные длины волн, являющийся дополнительным к голубому. В результате останется только зеленый цвет (рис. 18, б).

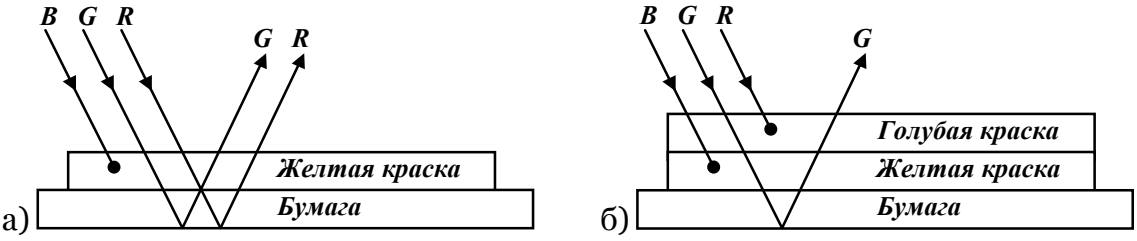


Рис. 18. Вычитание цветов из белого

Если на слой желтой и голубой красок нанести слой пурпурной (дополнительной к зеленому), все длины волн будут поглощены, получится черный цвет.

Таблица вычитания цветов из белого приведена ниже (см. также цветную вкладку, опыт Юнга):

Табл. 2. Вычитание цветов из белого

Цвет фильтра	*Задерживаемые цвета*	*Результирующий цвет*
Желтый	Синий	Желтый
Пурпурный	Зеленый	Пурпурный
Голубой	Красный	Голубой
Желтый + пурпурный	Синий + зеленый	Красный
Желтый + голубой	Синий + красный	Зеленый
Пурпурный + голубой	Зеленый + красный	Синий

Система CMY

Система ***CMY*** широко применяется в принтерах цветной печати. Она построена на субтрактивной цветовой модели, описанной выше. Для печати используются краски, имеющие дополнительные к первичным цвета — голубая (***C****yan*), пурпурная (***M****agenta*) и желтая (***Y****ellow*).

Существует также разновидность системы ***CMYK***, в которой добавлен еще один цвет — черный, идущий в названии под буквой ***K*** (от *Blac****k***). Использование черного цвета позволяет получить более качественную печать, так как из-за неточного воспроизведения цветов красок и неточности печатающей головки получить качественный черный цвет сложно.

Цветовая температура

Понятие *цветовая температура* используется для характеристики света по спектральному составу. Все нагретые тела излучают электромагнитный спектр. При низких температурах они испускают невидимое длинноволновое излучение. При повышении температуры тела начинают светиться сначала темно-красным, затем ярко-красным, желтым, белым и наконец, голубовато-белым светом (свечение электросварочной дуги). Таким образом, между температурой светящегося тела и цветностью излучения существует прямая связь. Она детально изучена для *абсолютно черного тела*.

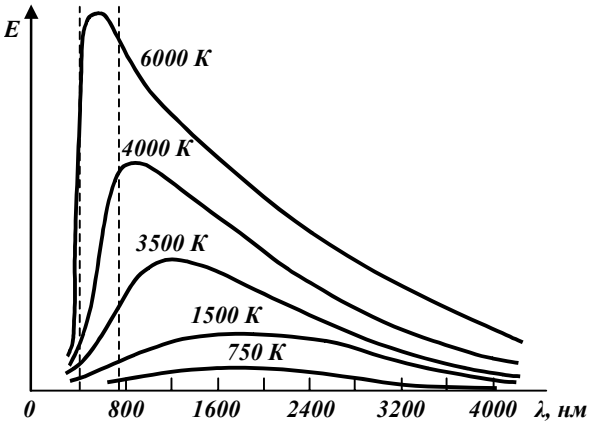


Рис. 19. Распределение энергии излучения абсолютно черного тела

Для каждого значения температуры абсолютно черного тела известен состав света, который оно излучает. Исходя из этого спектральный состав света характеризуют цветовой температурой — температурой абсолютно черного тела, при которой оно излучает свет того же спектрального состава, что и исследуемый.

Цветовая температура выражается в единицах абсолютной температуры — в градусах Кельвина (в Кельвинах). ***Ее значение характеризует распределение энергии световых излучений по спектру, а не температуру источника света***.

Для абсолютно черного тела это распределение показано на рис. 19. С увеличением температуры максимум энергии излучения сдвигается в сторону коротких волн. То есть, чем выше цветовая температура, тем больше в составе света голубого, синего и фиолетового цветов. И наоборот, чем ниже цветовая температура, тем больше в составе света длинноволновых составляющих — желтых, оранжевых и красных цветов.

К некоторым источникам света (лазерам, газосветным трубкам, светящимся краскам и организмам) понятие цветовой температуры неприменимо.

Табл. 3. Цветовая температура некоторых источников света

Источник света	***Цветовая температура, К***
Лампа накаливания	2700—3500
Лампа импульсная ксеноновая	6000
Лампа люминесцентная	
ЛТБ	2850
ЛХБ	4700
ЛД	6700
Свет солнца	
утром и вечером	4000—4500
в полдень без света неба	4700
плюс свет неба	5100—5500
Излучение небосвода без солнечного света	10 000—20 000

Цветовая температура некоторых источников света приведена в таблице выше.

Контрольные вопросы

1. Что называется цветом?
2. Чем определяется цвет предметов?
3. Что такое цветовой тон, насыщенность, яркость?
4. Что называется дополнительным цветом?
5. Какие системы измерения цвета вы знаете?
6. В чем разница между аддитивным и субтрактивным смешением цветов?
7. Что показывает кривая видности?
8. Как пользоваться цветовым кругом Ньютона?
9. Что показывает интегральный коэффициент диффузного отражения?
10. Что называется цветовой температурой?
11. Какие существуют виды отражения света?
12. Какие способы аддитивного смешения цветов вы знаете?
13. В чем заключается метамеризм цвета?
14. В чем заключается неполный цветовой охват системы RGB?

Видеосистема компьютера

Формирование изображения на экране монитора

Триады

Изображение на экране монитора с электронно-лучевой трубкой (кинескопом) формируется засветкой цветных экранных точек. Экранные точки, в свою очередь, состоят из ***триад*** — трех одноцветных точек, расположенных треугольником. Одноцветные точки покрывают всю рабочую поверхность монитора, и представляют собой капельки цветного люминофора. На рис. 20 слева (см. также цветную вкладку) приведен фрагмент поверхности монитора с большим увеличением. Такое изображение можно увидеть с помощью лупы.

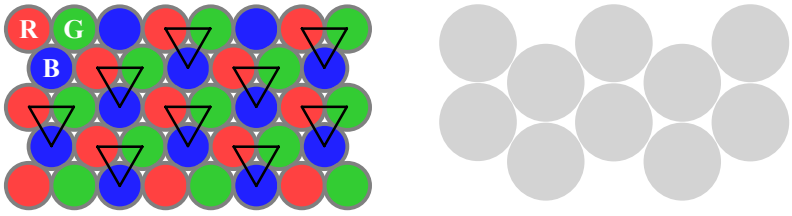


Рис. 20. Экранные точки-триады и эквивалентные цветные точки (зерна)

Одноцветные точки имеют *синий*, *зеленый* и *красный* цвета и расположены по поверхности экрана строго упорядоченно. Глаз не в состоянии различить отдельные одноцветные точки, их угловой размер с расстояния в 50 см равен всего 0,9 угловой минуты, (диаметр — примерно 0,13 мм). Происходит аддитивное пространственное смешение цветов, в результате которого как отдельные цветные точки воспринимаются триады, размер которых находится на уровне разрешения глаза (рис. 20, справа). Расстояние между триадами является одной из характеристик монитора, оно называется ***шагом точки*** (***dot pitch***) или ***зернистостью*** (***granularity***). Рис. 21 поясняет, как измеряется зернистость.

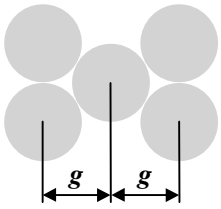


Рис. 21. Зернистость

Развертка

Засвечивание одноцветных точек происходит при их бомбардировке электронным лучом. Так как на экране имеются три типа одноцветных точек, используется три электронных луча. Каждый луч бомбардирует точки только своего цвета. Это достигается при помощи масок, располагаемых перед экраном.

В задней части кинескопа располагаются три электронных пушки, непрерывно испускающие узкие потоки электронов. На пути к экрану лучи проходят через магнитную отклоняющую систему, которая изменяет прямолинейное движение электронов лучей в центр экрана, и заставляет их «пробегать» экран по сложной зигзагообразной траектории, называемой ***растром***.

Рассмотрим, как формируется растр одного луча (рис. 22). Луч начинает движение в левой верхней точке экрана ***0***, и движется вправо, пробегая по своим одноцветным точкам по горизонтали. В результате этого движения одноцветные точки начинают светиться своим цветом, формируя след — ***линию растра***. Это движение по горизонтали происходит с равномерной скоростью и называется ***прямым ходом строки***.

Достигнув правого края изображения, луч гасится, и быстро перемещается к началу следующей линии растра. Это движение луча называется ***обратным ходом строки***. Далее процесс вычерчивания растровых линий повторяется до тех пор, пока луч не прочертит все линии растра и дойдет до точки ***1***.

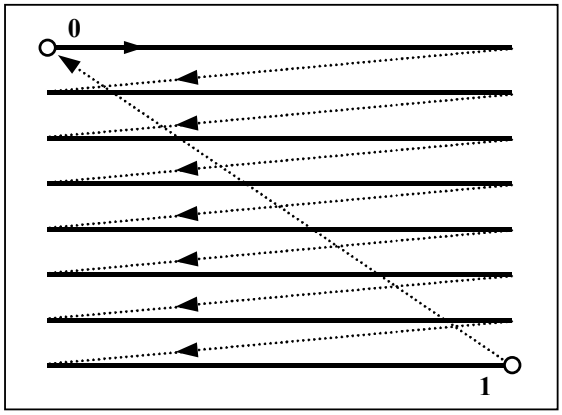


Рис. 22. Развертка

В точке 1 луч снова гасится, и быстро перемещается к началу растра. Это движение луча называется ***обратным ходом кадра***. Изображение, которое получается в результате развертки всех строк, называется ***кадром***.

*Существуют системы, в которых развертка вычерчивает не кадр, а полукадр. Целый кадр в этом случае формируется при помощи двух последовательно идущих полукадров, в первом из которых вычерчиваются нечетные строки развертки, а во втором — четные. Так изображение разворачивается, например, в телевизорах. Режим развертки называется чересстрочным (**interlaced**). В качественных мониторах развертка строчная (**non-interlaced, NI**).*

Так как люминофор, используемый для триад, обладает некоторым послесвечением, в тот момент, когда луч достигает конца растра, точки, которые были засвечены в начале развертки, все еще продолжают светиться.

Пиксели

В результате развертки на экране остаются линии растра. Если во время прямого хода луча изменять его плотность, интенсивность свечения (и послесвечения) его одноцветных точек также будет изменяться. В результате смешения цветов линия растра получит цветовую окраску, в точности соответствующую изменениям амплитуды плотности трех электронных лучей.

Рис. 23. Непрерывное изменение плотности электронного луча

Если плотность электронных лучей изменяется непрерывно, линия растра также непрерывно изменяет свой цвет. Такой способ получения изображения используется в телевидении (рис. 23).

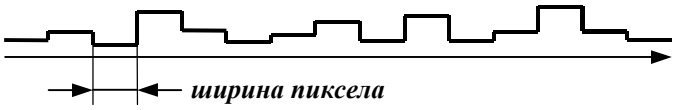


Рис. 24. Дискретное изменение плотности электронного луча

В видеосистеме компьютера изменение плотности лучей происходит дискретно, скачками. От одного скачка плотности до другого плотность луча остается неизменной. В результате вдоль линии растра на ней остаются прямоугольные области с одинаковым цветом — ***пиксели*** (рис. 24).

Формирование линии растра

Управление электронными лучами осуществляет видеоадаптер компьютерной системы. В точке ***0*** развертки (рис. 22) видеоадаптер вырабатывает сигнал начала прямого хода строки, посылая на монитор ***сигнал горизонтальной синхронизации H-Sync***. Монитор начинает разворачивать электронный луч по горизонтали. Одновременно видеоадаптер считывает из своей памяти, называемой ***видеопамятью***, значения цветных составляющих ***RGB*** первого пикселя и также посылает эти значения на монитор в виде трех сигналов цвета. Монитор начинает вычерчивать линию растра с заданным цветом. Отсчитав некоторое время, соответствующее ширине пикселя, видеоадаптер считывает из видеопамяти следующую ячейку, в которой записаны ***RGB*** составляющие цвета второго пикселя, и меняет сигналы цвета, посылаемые на монитор. На линии растра начинает разворачиваться второй пиксель, который имеет свой собственный цвет. Так продолжается до конца линии развертки. Таким образом, устанавливается соответствие между ячейками видеопамяти и пикселями в строке развертки (рис. 25).

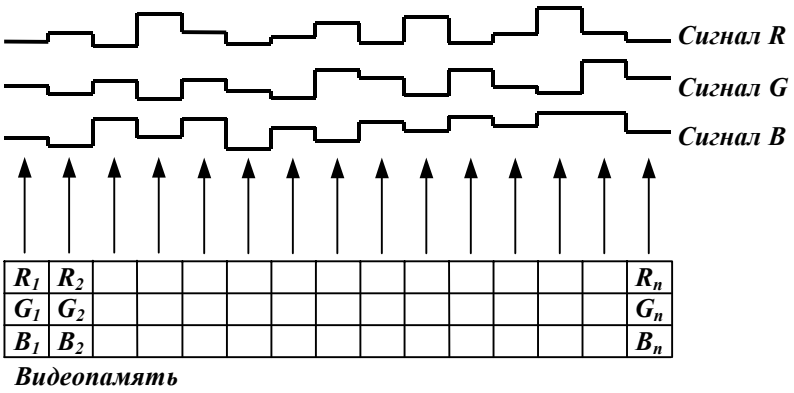


Рис. 25. Синхронизация между видеопамятью и пикселями первой линии растра

На время обратного хода строки видеоадаптер посылает сигнал *чернее черного*, чтобы обеспечить невидимость луча при изменениях яркости изображения пользователем. Видеопамять при этом не используется.

Перейдя к новой линии растра, видеоадаптер продолжает посылать сигналы цвета на монитор, считывая их *из последовательных ячеек видеопамяти*.

После развертки последней линии растра видеоадаптер вырабатывает сигнал начала нового кадра ***V-Sync*** (***сигнал вертикальной синхронизации***), по которому луч переходит в точку ***0***.

В прямоугольную область пикселя попадает некоторое количество одноцветных точек, которые формируют цвет пикселя. Высота пикселя определяется толщиной электронного луча, а ширина — видеоадаптером, который отсчитывает

время удержания постоянных сигналов цвета, соответствующих текущему пикселю. В разных режимах развертки монитор выводит разное количество строк, а видеоадаптер отсчитывает разное количество пикселей. Чем больше строк выводится во время развертки, тем больше пикселей формируется в строке, чтобы обеспечить квадратность пикселя. Однако, чем меньше пиксель, тем меньшее количество одноцветных точек попадает в его площадь. При завышенном количестве строк развертки на один пиксель может приходиться меньше триады, и тогда нарушается правильная цветопередача.

Например, на экране 19-ти дюймового монитора с зернистостью 0,2 и шириной рабочей области 352 мм расположено 2640 одноцветных точек, что соответствует 1760 триадам (зернам). При разрешении 1600×1200 на один пиксель по ширине приходится 1760/1600 = 1,1 зерна, или 2640/1600 = 1,65 одноцветных точки. Это предел для данного монитора. При разрешении 800×600 на один пиксель приходится уже 2,2 зерна и 3,3 одноцветных точки, что обеспечивает лучшую цветопередачу и четкость.

Экранная система координат

В результате развертки изображения на экране монитора остаются цветные прямоугольные (или похожие на прямоугольные) области с постоянным цветом — пиксели. Они образуют прямоугольную сетку, состоящую из ***m*** строк. В каждой строке разворачивается ***n*** пикселей. Таким образом, изображение разворачивается в сетку ***n*×*m***, в которой конкретный пиксель может быть указан номером строки ***y*** и позицией в ней ***x***.

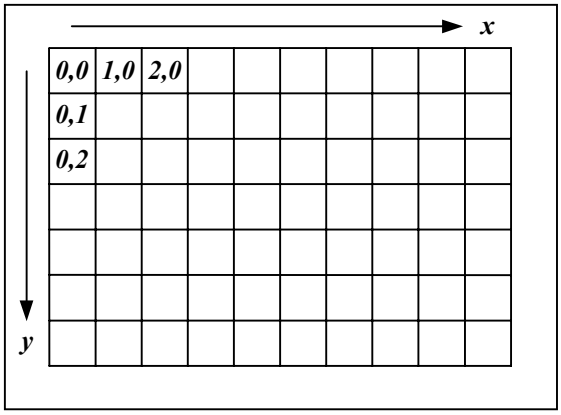


Рис. 26. Экранная система координат

Эти два числа являются координатами пикселя на экране. Так как во время развертки изображение считывается из последовательных ячеек видеопамяти, несложно установить соответствие между номером ячейки памяти и координатами пикселя. При этом начало экрана находится в левом верхнем углу экрана и соответствует началу видеопамяти.

Формирование изображения при помощи пикселей

Изображение формируется посредством засвечивания пикселей необходимыми цветами. Для этого в видеопамять адаптера записываются значения составляющих цвета для каждого пикселя. Конечно, программисты редко используют прямой доступ в видеопамять, так как это сопряжено с большой сложностью. Вместо этого операционная система или другой программный компонент предоставляют функции для выполнения элементарных графических операций,

таких, как вывод одного пикселя, рисование отрезка, прямоугольника, круга, дуги, закраска области и т. п.

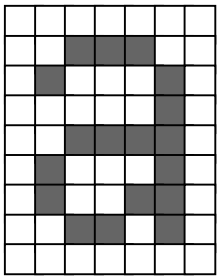


Рис. 27. Клеточная структура растрового изображения

В силу клеточной структуры экрана лучше всего изображаются вертикальные и горизонтальные линии. Формирование кривых линий затруднено (рис. 27).

Другие способы получения изображения

Кроме растровых, существуют еще ***векторные мониторы***. В них картинка формируется линиями, которые вырисовываются непосредственно электронным лучом. Так как движение электронного луча в них задается произвольным, требуемым образом, изображение кривых линий в таких мониторах не представляет трудности и отличается высоким качеством.

Другие типы мониторов

В последнее время широкое распространение получили плоские мониторы на жидких кристаллах. В них используется тот же принцип получения цветного изображения, что и в обычных мониторах с электронно-лучевой трубкой — пространственное смешение основных цветов ***RGB***. Пиксели этих мониторов образованы непосредственно излучающими элементами, и имеют квадратную форму, поделенную на три вертикальные цветные полоски (рис. 28).

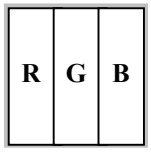


Рис. 28. Пиксель жидкокристаллического монитора

Видеоадаптер

Видеоадаптер является неотъемлемой частью настольной вычислительной системы. Его основное назначение — сформировать сигналы для управления электронными лучами (если используется электронно-лучевой монитор). Эти сигналы состоят из сигналов горизонтальной ***H-Sync*** и вертикальной ***V-Sync*** синхронизации и трех сигналов цвета ***RGB***.

Важной характеристикой изображения, формируемого адаптером, является ***разрешение***, которое указывается в виде произведения количества пикселей в строке развертки на количество самих строк. Например, разрешение *640×480* означает режим, при котором формируется *480* строк по *640* пикселей. Дополнительно к этому третьим сомножителем может указываться количество одновременно отображаемых цветов, например, *640×480×16*.

Большинство видеоадаптеров хранят изображение, которое необходимо вывести на экран монитора, в специальной памяти, называемой *видеопамятью* (другие адаптеры используют основную память). Видеопамять логически представляет собой последовательные ячейки по одной на каждый отображаемый пиксель. Каждая ячейка хранит значения цветовых составляющих ***RGB***, закодированные в той или иной форме.

Краткая история развития видеоадаптеров

Графический адаптер ***CGA*** (***Color Graphics Adapter*** — цветной графический адаптер) первой модели ***IBM PC*** (*1981*) мог выводить цветное изображение с разрешением всего *320×200×4* и черно-белое с разрешением *640×200×2*. Так как этого было явно недостаточно, вскоре появился новый адаптер ***EGA*** (***Enhanced Graphics Adapter*** — улучшенный графический адаптер, *1984*). Он имел разрешение в цветном режиме *640×350* при *16*-ти цветах, которые могли выбираться из палитры размером *64* цвета. Этот адаптер вместе с адаптерами ***Hercules*** компании ***Hercules Computer Technologies*** оказал значительное влияние на развитие графических систем компьютеров в начале их развития.

На смену адаптеру ***EGA*** в *1987* году пришел адаптер ***VGA*** (***Video Graphic Array*** — графический видеомассив) с разрешением *640×480*, ставший первым стандартом. При объеме видеопамяти *256* Кбайт адаптер ***VGA*** позволял одновременно выводить *16* цветов, а при разрешении *320×200 — 256* цветов из палитры в *256* Кбайт цветов. По сегодняшним меркам возможности этого адаптера более, чем скромные, однако производители видеокарт улучшали возможности адаптеров, построенных на базе стандарта ***VGA***. Появилось множество видеокарт, называемых ***SVGA*** (***Super VGA***) и ***UVGA*** (***Ultra VGA***). Разрешение этих адаптеров достигает *800×600* и *1024×768* при количестве цветов до *16* Мбайт. На самом деле режимы, предлагаемые видеокартами ***SVGA***, не являются стандартными, и определяются производителем, хотя разрешение *800×600* часто приписывают стандарту ***SVGA***.

Большое влияние на развитие стандартов устройств отображения информации оказала группа ***VESA*** (***Video Electronics Standards Association***). Стандарты ***VESA*** определяют разрешение устройств, способы построения, параметры сигналов, устройства соединения и др. Стандарт ***VBE*** (***VESA BIOS Extension***, *1998*), определяет расширение ***BIOS*** видеокарт для поддержки стандартов ***VESA***. Это расширение дает возможность использовать новые типы видеоадаптеров в любой среде, например, в *Turbo Pascal* под *MS-DOS*.

Видеорежим

Видеорежим — это номер, используемый видеоадаптером для идентификации важнейших характеристик, которые адаптер может выработать — разрешение и максимальное количество цветов. В таблице ниже приведены *режимы MS-DOS*, поддерживаемые всеми видеоадаптерами и мониторами.

Разрешение задает *частоты* смены строк и кадров, которые адаптер передает на монитор в виде *сигналов синхронизации* (сигналов развертки). Сигнал синхронизации ***H-Sinc*** заставляет монитор разворачивать новую строку, а сигнал синхронизации ***V-Sinc*** заставляет монитор разворачивать новый кадр (начинать новую строку с начала экрана).

Режим *640×480×16*, называемый ***Standard VGA***, является одним из важных. Операционная система Windows, начиная с версии *3.1*, может быть установлена

только на компьютер, оснащенный как минимум адаптером ***VGA***, поэтому наличие этого режима предполагается в любой конфигурации аппаратуры.

Табл. 4. Видеорежимы MS-DOS

Режим	***Разрешение***	***Цвета***	***Текст/Графика***	***CGA***	***EGA***	***VGA***
00h	40×25	16	Текст	+	+	+
01h	40×25	16	Текст	+	+	+
02h	80×25	16	Текст	+	+	+
03h	80×25	16	Текст	+	+	+
04h	320×200	4	Графика	+	+	+
05h	320×200	4	Графика	+	+	+
06h	640×200	2	Графика	+	+	+
07h	80×25	2	Текст		+	+
0Dh	320×200	16	Графика		+	+
0Eh	640×200	16	Графика		+	+
0Fh	640×350	2	Графика		+	+
10h	640×350	16	Графика		+	+
11h	640×480	2	Графика			+
12h	640×480	16	Графика			+
13h	320×200	256	Графика			+

В табл. 5 приведены некоторые режимы VESA, определенные в VBE 2.0.

Табл. 5. Видеорежимы Windows

Режим VESA	***Разрешение***	***Количество цветов***
100h	640×400	256
10Dh (10Eh, 10Fh)	320×200	32K (64K, 16.8M)
101h (110h, 111h, 112h)	640×480	256 (32K, 64K, 16.8M)
102h (103h, 113h, 114h, 115h)	800×600	16 (256, 32K, 64K, 16.8M)
104h (105h, 116h, 117h, 118h)	1024×768	16 (256, 32K, 64K, 16.8M)
106h (107h, 119h, 11Ah, 11Bh)	1280×1024	16 (256, 32K, 64K, 16.8M)

При установке операционной системы не делается никаких предположений о наличии более качественного видеоадаптера вплоть до завершающей стадии. При сбоях операционная система также выбирает режим *640×480×16*, чтобы защитить монитор от порчи недопустимыми сигналами развертки.

Видеопамять

Изображение, выводимое на экран, записывается в *видеопамять* видеоадаптера. Чтобы сформировать цветное изображение, на монитор подаются три цветовых сигнала ***R***, ***G*** и ***B***, числовые величины которых определяют степень засветки одноцветных точек монитора и результирующий цвет мониторного пикселя. Варьируя значениями ***RGB***, можно получить почти любой требуемый цвет, однако практические значения сигналов цвета ограничиваются из-за необходимости хранить эти значения в видеопамяти.

Так, в адаптере ***EGA*** для хранения всех составляющих цвета отводится *4* бита, что дает возможность записать только *16* различных значений. В табл. 6 приведен перечень цветов, вырабатываемых адаптером ***EGA*** в этом режиме. Буквой ***I*** обозначена интенсивность.

Количество бит, отводимое в видеопамяти для хранения цвета одного пикселя, называется ***глубиной цвета***. Существующие цветные режимы видеоадаптеров предусматривают глубину цвета в *4*, *8*, *15*, *16* и *24* бита, что позволяет отображать *16*, *256*, *32 768*, *65 536* или *16 777 216* цветов одновременно. Возможна и бóльшая глубина цвета.

Табл. 6. Цвета 16-ти цветного режима MS-DOS (цвета EGA)

Цвет	***I***	***R***	***G***	***B***	***Номер***
Black (черный)	0	0	0	0	0
Blue (синий)	0	0	0	1	1
Green (зеленый)	0	0	1	0	2
Cyan (сине-зеленый)	0	0	1	1	3
Red (красный)	0	1	0	0	4
Magenta (фиолетовый)	0	1	0	1	5
Brown (коричневый или грязно-желтый)	0	1	1	0	6
Light Gray (светло-серый)	0	1	1	1	7
Dark Gray (темно-серый)	1	0	0	0	8
Light Blue (ярко-синий)	1	0	0	1	9
Light Green (ярко-зеленый)	1	0	1	0	10
Light Cyan (бирюзовый)	1	0	1	1	11
Light Red (ярко-красный)	1	1	0	0	12
Light Magenta (лиловый)	1	1	0	1	13
Yellow (желтый)	1	1	1	0	14
White (белый)	1	1	1	1	15

Наличие определенного объема видеопамяти ограничивает сочетание глубины цвета и разрешения. Если, например, объем памяти видеоадаптера составляет 1 Мбайт, он может поддерживать глубину цвета *24* бита при разрешении *640×480*. При разрешении *800×600* максимальная глубина цвета может составлять 8 бит (*256* цветов) или *15* бит (*32768* цветов). При разрешении *1024×768* этот адаптер может отображать максимум *256* цветов (*1024×768×8 = 786432*).

Современные видеоадаптеры оснащаются видеопамятью объемом не менее *4* Мбайт, что дает возможность отображать глубину цвета в *24* бита при разрешении *1024×768* и получать, таким образом, картинку фотографического качества. Глубину цвета *24* бит называют также режимом ***True Color*** (истинный цвет). Режимы, в которых одновременно можно отобразить *32768* или *65536* цветов, получили название ***High Color*** (цвет высокого разрешения).

В графическом режиме *MS-DOS* видеопамять *проецируется* на основную память компьютера в сегмент ***A000h***, а в текстовом режиме — в сегмент ***B800h***. Это позволяет записывать изображение непосредственно в видеоадаптер обращением к нему, как к обычной памяти. В графическом режиме видеопамять имеет сложную многослойную структуру. Это имеет значение при программировании компьютерной графики в операционной среде *MS-DOS*, а также при разработке видео драйверов в операционной системе *Windows*. Описание такого программирования составляет предмет отдельной книги и здесь не рассматривается.

Видеопроцессор

Первоначально видеоадаптеры оснащались ***видеоконтроллером*** — микросхемой, которая вырабатывала видеосигналы ***RGB*** и сигналы синхронизации

для получения нужного количества строк и пикселей. Одной из наиболее популярных являлась микросхема **6845** фирмы ***Motorola***. В современных видеоадаптерах используется ***видеопроцессор***, основу которого составляет быстродействующий процессор, сравнимый по скорости работы с центральным процессором и работающий с памятью, объем которой сопоставим с объемом основной памяти.

Для ускорения обмена с видеопамятью в большинстве случаев используется шина ***AGP***, которая на частоте *66* МГц обеспечивает скорость обмена *528* Мбайт/с. Скорость обмена имеет важное значение для обеспечения мультимедийных возможностей компьютера, таких, как вывод видеоизображения. В последнее время появилась новая шина ***PCIExpress***.

Для обеспечения высокой скорости обмена в видеоадаптерах также используются специальные типы памяти, например, память типа ***VRAM***, обеспечивающая одновременное чтение и запись при помощи двух портов. Объем памяти видеоадаптера составляет *16*, *32* и более Мбайт. С одной стороны, большой объем памяти обеспечивает нужную глубину цвета при больших разрешениях, таких, как *1280×1024* и более. С другой стороны, видеопроцессор оснащается аппаратными средствами ускорения формирования графических элементов изображения, таких, как *линии* и *треугольники*, и выполняет основные операции с массивами бит видеопамяти, такие, как *очистка*, *копирование*, *сдвиг*, *логические операции* и др. Для обеспечения высокой скорости смены кадров видеоизображения видеопамять может хранить несколько кадров одновременно. В видеопамяти хранятся также *текстуры*, накладываемые на графические изображения.

Программирование видеоадаптера, оснащенного видеопроцессором, представляет собой сложную задачу. Для ее упрощения разработаны стандарты, называемые ***интерфейсами***. Одним из наиболее известных промышленных стандартов является интерфейс ***OpenGL***, предназначенный для визуализации трехмерных сцен. Другим важным стандартом является интерфейс ***DirectX***, разработанный фирмой ***Microsoft*** и предназначенный для операционной системы ***Windows***. Подсистемы ***Direct3D*** и ***DirectDraw*** этого интерфейса обеспечивают вывод трехмерных изображений и непосредственный доступ к видеопамяти.

Контрольные вопросы

1. Какое смешение цветов происходит в мониторе — аддитивное или субтрактивное, одновременное, пространственное, поочередное или бинокулярное?
2. Что называется триадой? Что такое зернистость монитора?
3. Что называется растром? Что такое развертка?
4. В чем суть синхронизации развертки с видеоданными?
5. Что называется видеорежимом? Что определяет видеорежим?
6. Что называется глубиной цвета? Что такое *HighColor* и *TrueColor*?
7. Что называется разрешением? Как оно измеряется?
8. Что такое пиксель? Из какого количества триад состоит пиксель?
9. В чем разница между видеоконтроллером и видеопроцессором?
10. Какие типы мониторов, кроме растровых, вы знаете?

Растровая графика

Растровые устройства

В компьютерной графике изображения выводятся в большинстве случаев на ***растровое устройство***, такое как монитор, принтер, плоттер. Растровое устройство представляет собой решетку, или сетку элементов изображения — точек, расположенных равномерно по всей поверхности устройства. Каждый элемент изображения адресуется парой чисел *(x, y)* — двумя декартовыми координатами, и может принимать один из заранее определенных цветов.

Растровый монитор представляет сетку пикселей, которые адресуются номером строки развертки и позицией в строке. Пиксели представляют из себя квадраты (или близкие к квадрату прямоугольники), которые могут принимать заданный цвет. Пиксели на растровом мониторе не перекрывают друг друга, вследствие чего между ними могут возникать нецветные промежутки.

Растровый черно-белый принтер или плоттер представляет сетку из круглых или близких к круглым точек, которые, в отличие от монитора, могут частично перекрывать друг друга.

Перекрытие точек позволяет сглаживать линии. На рис. 29 слева показано, как выглядит отрезок прямой линии при печати без перекрытия точек, а в центре — с перекрытием.

Рис. 29. Прямая линия на растровом принтере

Перекрытие точек может быть получено за счет того, что расстояние между точками меньше, чем их диаметр (рис. 29, б). В другом случае регулируется диаметр точек при неизменном шаге (рис. 29, в).

Адресация точек в принтерах производится декартовыми координатами *(x, y)*, которые задают положение точки по отношению к левому верхнему углу текущего листа. В некоторых принтерах адресация точек невозможна. В этом случае принтер принимает битовый массив, описывающий точки, и вставляет его в поток вывода.

Принтеры могут также принимать графическую информацию в описательной форме. Например, отрезок описывается координатами начальной и конечной точек, толщиной линии и способом ее заполнения, в том числе цветом и узором. В этом случае разложение изображения на точки выполняется процессором принтера.

Например, на матричном принтере без программирования вывода отрезок прямой будет выведен так, как показано на рис. 29 слева, а с дополнительным программированием можно получить изображение, показанное на этом же рисунке в центре.

Рис. 30. Отрезок прямой, напечатанный как растровая картинка

На лазерном принтере отрезок прямой можно вывести как битовую картинку, напечатав ее непосредственно из графического редактора типа *Paint* (рис. 30). Такой же отрезок можно напечатать с лучшим качеством, если нарисовать его в

редакторе *Microsoft Word* и напечатать его из этого приложения (рис. 31). *Word* посылает на лазерный принтер описательную информацию, либо пересчитывает изображение в точки сам, — это зависит от типа принтера и его настроек.

Рис. 31. Отрезок прямой, растрированный в принтере

Одной из задач программирования графики является согласование масштаба изображения с размерами устройства вывода. В одних случаях можно использовать оригинальные единицы измерения исходного изображения, например Пиксели или сантиметры, в других программист сам выполняет пересчет координат, по которым строится изображение.

Растрирование

Рисование на растровом устройстве заключается в вычислении точек, которые покрываются элементами графического изображения, и выводе этих точек на устройство. Процесс вычисления точек изображения называется ***растрированием*** или ***растеризацией*** (английский термин — ***rasterization***). Рис. 32 поясняет растрирование. Слева на этом рисунке показан контур исходного изображения, на которое наложена сетка, а справа — результат растрирования.

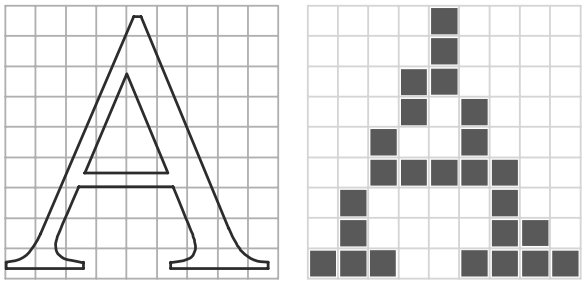


Рис. 32. Растрирование изображения

Решение о том, следует закрашивать точку, или нет, принимается на основе процентного соотношения между той частью контура, которая покрывает точку, и той, которая остается непокрытой. Если процент покрытия точки контуром превышает заданную величину, точка закрашивается.

При этом не всегда удастся получить правильное изображение. Некоторые точки в растрированном изображении буквы А на рисунке выше необходимо поставить для того, чтобы обеспечить ее узнаваемость и связность (неразрывность) линий.

Растрирование изображений является достаточно сложным преобразованием, поэтому в большинстве случаев оно выполняется системными функциями или непосредственно устройством вывода (например, принтером). Основным графическим примитивом, растрирование которого берет на себя операционная среда, является *отрезок прямой линии*. Любое графическое изображение можно построить при помощи только отрезков. На практике в распоряжении программиста есть более сложные примитивы, такие, как дуги, прямоугольники (квадраты), эллипсы (окружности) и другие.

Графические функции операционной среды выполняют также закраску областей одним цветом или плавным переходом одного цвета в другой, а также заполнение области двухцветным рисунком (узором) и др.

Растровые и векторные изображения

Все исходные графические изображения условно можно разделить на две группы — *точечные* (битовые) и *векторные*.

Точечное изображение (в просторечии *битман — bitmap*) представляет собой готовую растрированную картинку, описание которой представляет собой карту распределения цвета по пикселям. Примерами таких изображений являются файлы типа *.bmp* и символ экранного шрифта. Точечные изображения получают в основном при помощи графических редакторов, таких, как *Paint*, и сканированием.

Векторные изображения строятся по формулам или функциям, описывающим области закраски. Примерами векторных изображений являются отрезок прямой, задаваемый координатами своих концевых точек и толщиной линии, символ шрифта ***TrueType***, файл типа *.wmf* (*Windows metafile*). Векторные изображения создаются при помощи редакторов векторной графики, таких, как ***Corel Draw***, ***3D Max***, ***AutoCAD*** и др. Следует понимать, что векторное изображение — это способ записи, а не отображения. *Независимо от того, какой вид или формат имеет графическое изображение, при выводе на растровое устройство оно растеризуется.*

Точечные изображения используют в случаях, когда требуется сложное сочетание точек, описать которое математически не представляется возможным. Примером служит фотография. Остальные изображения получают комбинацией простейших графических примитивов, обеспечиваемых операционной средой. Пример — график функции, который можно построить при помощи отрезков прямых.

Не следует воспринимать исходное точечное изображение как готовый к использованию графический примитив. Чаще всего требуются дополнительные преобразования точечного изображения с тем, чтобы получить приемлемый результат. Примером является масштабирование точечного рисунка, полученного сканированием с высоким разрешением. Непосредственная печать или вывод на экран такого изображения во многих случаях может оказаться неприемлемой из-за слишком большого размера. Другой немаловажной проблемой является цветопередача.

Графические форматы

Графические форматы предназначены для хранения графических изображений. Так же, как и источники изображений, они делятся на растровые и векторные. В растровых форматах картинка описывается как совокупность цветных точек, а в векторных — как совокупность графических примитивов (функций).

Существует огромное количество графических форматов для хранения как растровых, так и векторных изображений. Если для растровых форматов информация о структуре файлов относительно широко известна и доступна, то для векторных форматов это не так. Практически каждый редактор векторного изображения имеет свой собственный формат для хранения, что усложняет обмен графической информацией между приложениями. Ниже в таблице приведены характеристики наиболее распространенных растровых графических форматов.

В операционной системе *Windows* используются также специальные растровые форматы для значков (***icon***) и указателей (***cursor***).

Графическая информация часто имеет большой объем, поэтому она может сжиматься с использованием различных алгоритмов. На объем информации оказывает влияние размер картинки и глубина цвета, используемая для записи изображения. Глубина цвета, в свою очередь, выбирается исходя из необходимого количества разных цветов. Так, при глубине цвета в *1* бит/пиксель изображение может иметь всего два цвета (например, черный и белый), а при глубине *24* бит/пиксель максимальное количество цветов *16* млн.

Табл. 7. Распространенные растровые графические форматы

Формат	***Глубина цвета, макс.***	***Число цветов, макс.***	***Размер, макс.***	***Методы сжатия***	***Кодирование нескольких изображений***
BMP	24	16M	64K×64K	RLE*	–
GIF	8	256	64K×64K	LZW	+
JPEG	24	16M	64K×64K	JPEG	–
PNG	48	256M	2Г×2Г	Deflation	–
TIFF	24	16M	4Г	LZW, RLE и др.*	+

* Сжатие выполняется факультативно

Далее коротко описываются наиболее распространенные форматы графических файлов. Подробную информацию можно найти в приложении.

Формат BMP

Формат *BMP* является внутренним форматом хранения изображений в операционной системе *Windows*, что обуславливает его широкое распространение.

На рис. 33 показана структура независимого от устройства графического растрового файла (***dib***-формат, ***device independent bitmap***). Все *BMP*-файлы должны храниться в ***dib***-формате, чтобы их можно было отображать на компьютерах с различной графической аппаратурой.

Раздел	***Размер, байт***
Заголовок файла	14
Информационный заголовок	40
Таблица цветов	переменный
Массив данных	переменный

Рис. 33. Структура *dib*-формата

Файл ***bmp*** содержит 4 раздела:

1) *Заголовок файла* содержит признак **BM**, размер файла и смещение к таблице бит (массиву данных).
2) *Информационный заголовок* описывает изображение. Он содержит размер массива данных, размер изображения, глубину цвета, признак сжатия и др.
3) *Таблица цветов* описывает цвета, используемые в изображении. Если глубина цвета равна *24*, таблица отсутствует.
4) *Массив данных* содержит цвета пикселей изображения в виде индексов таблицы цветов, или непосредственно цвета, если глубина цвета *24*.

Формат GIF

Формат *GIF* (***Graphics Interchange Format*** — формат обмена графическими данными), разработанного компанией ***CompuServe***, является широко распространенным графическим форматом, используемым в сети *Интернет*. Его особен-

ностями являются ограниченное количество цветов, возможность хранить несколько изображений и компактный размер. К каждому изображению может быть приписана дополнительная графическая и неграфическая информация.

Формат PNG

Формат *PNG* (***Portable Network Graphic*** — переносимый сетевой формат, произносится «пинг») был разработан для замены *GIF*, чтобы обойти юридические препятствия, стоящие на пути использования *GIF*-файлов. *PNG* унаследовал многие возможности *GIF* и, кроме того, он позволяет хранить изображения с истинными цветами. Формат *PNG* сжимает информацию растрового массива в соответствии с вариантом пользующегося высокой репутацией алгоритма сжатия *LZ77* (предшественника *LZW*), которым любой может пользоваться бесплатно.

Формат JPEG

Формат файла *JPEG* (***Joint Photographic Experts Group*** — объединенная экспертная группа по фотографии, произносится «джейпег») был разработан компанией ***C-Cube Microsystems*** как эффективный метод хранения изображений с большой глубиной цвета, например, получаемых при сканировании фотографий с многочисленными, едва уловимыми оттенками цвета. Главная особенность формата *JPEG* заключается в том, что в нем используется алгоритм сжатия с потерями информации. При этом приносится в жертву часть информации об изображении, чтобы достичь большего коэффициента сжатия.

Формат TIFF

Формат *TIFF* (***Tagged Image File Format***, формат файлов изображения, снабженных тегами) — один из самых сложных. Особенностью формата является описание характеристик изображения при помощи *тегов*. Одни теги описывают размер изображения, другие — таблицу цветов, третьи — массив данных. Теги могут добавляться по мере необходимости, что затрудняет расшифровку файла, снабженного нестандартными тегами.

Несмотря на свою сложность, формат *TIFF* является одним из лучших для передачи растровых массивов с одной платформы на другую благодаря своей универсальности, позволяющей кодировать в двоичном виде практически любое изображение без потери его визуальных или каких-либо иных атрибутов.

Формат WMF

Файл метафайла (*Windows metafile*) похож на музыкальную запись. Он содержит запись команд графического интерфейса *Windows GDI*. Для отображения файла эти команды передаются функции *PlayMetaFile*. Эта функция рисует графические примитивы на устройстве в соответствии с его возможностями.

Метафайлы обеспечивают независимые от устройства средства записи изображений. Они идеально подходят для хранения информации об изображении карт, диаграмм, чертежей и прочих изображений, имеющих четко выраженную *перекрывающуюся* структуру. Команды метафайла размещаются на диске, что позволяет экономить основную память.

Погрешности растрового изображения

Во многих случаях изображение получается при помощи отрезков прямых. Если говорить о качестве такого изображения, относительно правильно будут отображаться только горизонтальные и вертикальные отрезки, все остальные — приближенно. Хуже всего дело обстоит с отрезками, которые почти горизонтальны или почти вертикальны. На рис. 34 показан почти горизонтальный отрезок. Разница положения его концевых точек по высоте составляет всего 1 пиксель, хотя разница координат может быть большей. В результате наклонный отрезок выглядит как две горизонтальные линии.

Рис. 34. Ступенчатость отрезка прямой

Аналогичная ситуация складывается с изображением кривых линий, например, окружностей и дуг. На рис. 35 показано увеличенное в *4* раза изображение окружности диаметром 4 пикселя.

Рис. 35. Окружность малого диаметра

Не лучше обстоит дело и с толщиной линий. Растровый монитор способен отображать вертикальную или горизонтальную линию толщиной в целое число пикселей. Поэтому часто оказывается, что две линии, предположительно имеющие разную толщину, выглядят, как имеющие одинаковую толщину. Если же линия наклонная, понятие толщины линии расплывается, потому что нельзя определить размер в произвольном направлении точным числом пикселей.

Рис. 36. Ширина наклонной линии

На рис. 36 две линии толщиной *2* пикселя нарисованы горизонтально и под углом *45°*. Наклонная линия кажется уже горизонтальной, хотя представляется ровной. На самом деле наклонная линия имеет ступенчатую, змеевидную структуру, о чем свидетельствует ее увеличенное изображение. Толщина линии в любом ее месте составляет диагональ пикселя, т. е. *1,4*.

Проблемой растрового изображения является также непропорциональность размера пикселя по ширине и высоте, что приводит к изображению квадрата как прямоугольника, и к разной толщине горизонтальных и вертикальных линий. Размер экрана монитора, например, в большинстве случаев имеет соотношение сторон ширина:высота, равное *4:3*. Большинство режимов видеоадаптеров поддерживают это соотношение, но не все, например, использовавшийся ранее режим с разрешением *640×400*. Если соотношение сторон экрана и разрешение соответствуют друг другу, правильное отображение можно настроить органами регулировки размера изображения монитора.

Аналогичную проблему имеют и принтеры, так как разрешающая способность большинства принтеров различна в горизонтальном и вертикальном направлениях. Но в принтерах вследствие высокой общей разрешающей способности эта проблема не так заметна, за исключением матричных принтеров, разрешающая способность которых невелика.

Улучшение растрового изображения

Растровое изображение во многих случаях можно улучшить, используя плавный переход цвета. На рис. 37 слева приведены два растровых изображения окружности диаметром *13* с толщиной линии *2*. Левая окружность получена при помощи только черных пикселей стандартной процедурой редактора ***Paint***, правая улучшена добавлением двух градаций серого. Справа на рисунке показан улучшенный круг с *4*-х кратным увеличением.

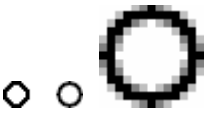


Рис. 37. Улучшение изображения окружности

Antialiasing

Ступенчатость геометрического растрового изображения в англоязычной литературе называют *aliasing*, а метод устранения ступенчатости — *antialiasing*. Ступенчатость устраняется окрашиванием угловых, выступающих точек геометрических фигур в более светлые тона, чем основная часть изображения.

Геометрическая фигура представляет собой некоторый геометрический контур, который закрашивается заданным цветом. В качестве примера рассмотрим рисование окружности диаметром *11* с толщиной линии *2*. Стандартный алгоритм закрашивает все пиксели, попавшие внутрь контура целиком и также пиксели, бóльшая часть площади которых также попадает внутрь геометрического контура (рис. 38).

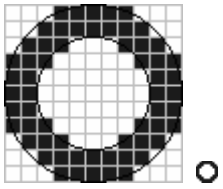


Рис. 38. Стандартный алгоритм рисования

Алгоритм, использующий устранение ступенчатости, закрашивает пиксели, попавшие в контур фигуры лишь частично, некоторым промежуточным цветом (рис. 39). Этот цвет рассчитывается, исходя из цвета фигуры и цвета фона, и может быть выбран в соответствии с той площадью пикселя, которая попадает в контур фигуры. Если фигура занимает p долю площади пикселя, промежуточный цвет можно рассчитать по формуле:

$C = Ц \cdot p + Ф \cdot (1 - p)$,

здесь Ц — цвет фигуры, а Ф — цвет фона. Например, при $Ц = (128,0,0)$, $Ф = (255,255,255)$, $p = 0.7$ и с учетом представления цвета в виде вектора RGB получим промежуточный цвет $C = (90,0,0) + (76,76,76) = (156,76,76)$.

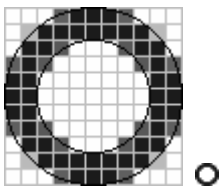


Рис. 39. Устранение ступенек

Цифровая фильтрация

Для устранения ступенек на уже нарисованном изображении используется ***цифровая фильтрация***. Цвет пикселя вычисляется как взвешенная сумма цветов пикселей в некоторой окрестности обрабатываемого пикселя. На окрестность пикселя накладывается ***маска***, представляющая собой весовые коэффициенты как самого пикселя, так и пикселей, которые его окружают. На рис. 40 маленький квадрат — это тестируемый пиксель, большой квадрат — окрестность размером *3×3*. Цифрами обозначены весовые коэффициенты пикселей. Цвет каждого пикселя окрестности умножается на весовой коэффициент, все полученные цвета складываются и делятся на некоторое число, нормирующее яркость, например, на сумму всех весовых коэффициентов (в примере равную *7*). Полученный цвет присваивается текущему пикселю.

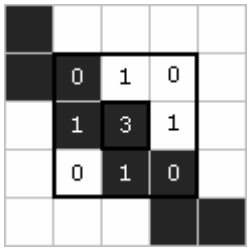


Рис. 40. Цифровой фильтр

На рис. 41 приведен пример применения указанного фильтра к окружности. Слева показана окружность в увеличенном и нормальном масштабе до фильтрации, в центре— после фильтрации.

Рис. 41. Применение цифрового фильтра

Избирательное применение фильтра дает лучшее сглаживание. На рис. 41, справа фильтр применялся только в том случае, если цвет тестируемого пикселя не совпадал с заданным цветом круга. В результате цвет самого круга не претерпел изменения.

Данный фильтр сглаживает переходы цвета. При помощи фильтров можно также повысить резкость, выделить контур и др. Размер маски может варьироваться в значительных пределах, от *2* до *9*.

Дизеринг

При выводе полноцветного изображения на устройство с ограниченным набором отображаемых цветов возникает ***проблема цветопередачи***. Примером может служить вывод цветной фотографии на черно-белый принтер. В полиграфии эта задача решается при помощи *растрирования*, когда изображение раскладывается на элементы — точки, штрихи. Если рассмотреть при помощи лупы фотографию в черно-белой газете, мы увидим множество черных точек разного диаметра (рис. 42, см. также рис. 44, г).

Для получения света и тени на черно-белом изображении используется тот же оптический эффект, что и при устранении ступенчатости — слияние мелких деталей за счет ограниченной разрешающей способности глаза человека. Цвет,

который нельзя отобразить, получается смешиванием имеющихся цветов в той или иной пропорции, в данном случае — изменением площади черных точек.

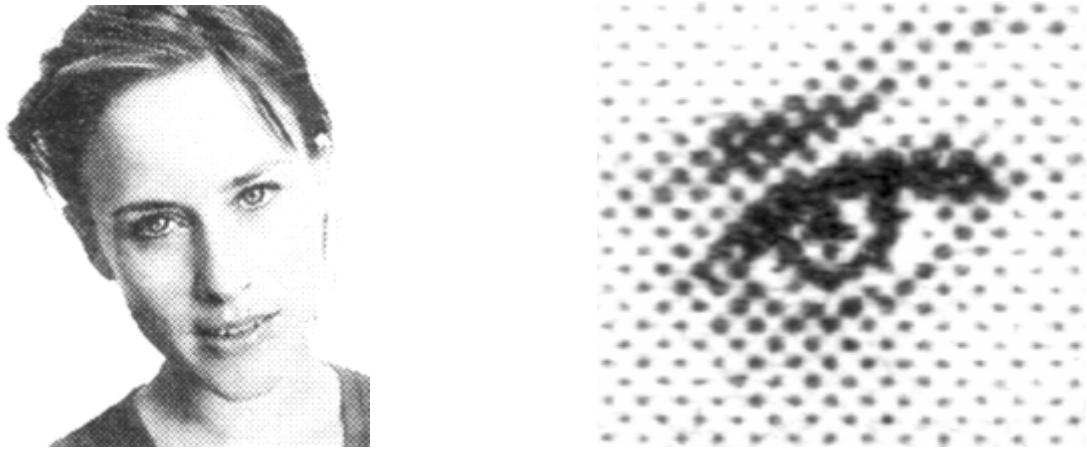


Рис. 42. Точечная структура напечатанной фотографии (Патрисия Аркетт)

Дизеринг (***dithering***, русский термин — ***псевдотонирование***) — это способ получения отсутствующего на устройстве цвета ***чередованием*** двух или более имеющихся в распоряжении цветов. Например, чтобы получить серый цвет на черно-белом устройстве, чередуются черные и белые точки. Общая яркость изображения при этом изменится за счет незакрашенных точек (которые будут черными или белыми в зависимости от устройства), создавая иллюзию нечерного (небелого) изображения.

В зависимости от желаемого результирующего цвета чередование может выполняться через точку, через две, через три и т. д. Так, при чередовании красного и зеленого цвета через точку при аддитивном смешении получим желтый, а при чередовании через две точки — оранжевый (одна зеленая, две красных) или салатный (две зеленых, одна красная).

Для выполнения псевдотонирования закрашиваемая поверхность разбивается на области размером ***m×n*** точек. Каждая такая область будет представлять собой один элемент изображения с заданным цветом — новый, увеличенный пиксель. Далее вычисляется процентное соотношение цветов, которые будут использоваться для чередования и количество точек одного и другого цветов. Остается только равномерно распределить точки одного и другого цвета в пределах области нового пикселя.

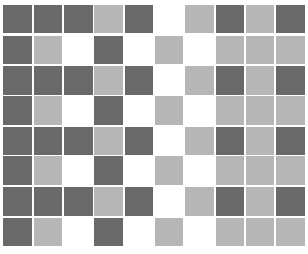


Рис. 43. Чередование красного (темный) и зеленого (светлый) цветов

В качестве примера на рис. 43 приведено чередование красного и зеленого цветов на квадрате размером *2×2*. На изображении пять вертикальных полос с распределением соотношения красный/зеленый *3/1*, *2/1*, *1/1*, *1/2* и *1/3*.

Очевидно, что существует множество вариантов размещения цветов, поэтому задача размещения цветных точек внутри нового пикселя нетривиальна.

Задача усложнится, если размер нового пикселя увеличить, а также если область закраски не прямоугольная, а произвольная.

Не всегда можно выполнить дизеринг чередованием цветов. Например, исходное изображение не содержит достаточно протяженных одноцветных областей, а состоит из отдельных разноцветных пикселей. Выполнить псевдотонирование в этом случае можно, закрашивая увеличенные пиксели взвешенной совокупной яркостью пикселей.

Часто вместо псевдотонирования выполняется замена одного цвета другим, наиболее подходящим. Результат такого псевдотонирования не всегда удовлетворителен. Однако в современных технологиях такое псевдотонирование широко используется при сканировании и в цифровой фотографии. Сканеры распознают цвет с глубиной *48* бит/пиксель, а цифровые камеры — с глубиной *30* или *36* бит/пиксель. Формат *BMP* для хранения таких цветных изображений не подходит, поэтому используются формат *JPEG* и формат *RAW* (необработанный), сходный с форматом *TIFF*. При выводе таких изображений на экран монитора или при сохранении в файл типа *BMP* псевдотонирование неизбежно.

Масштабирование растрового изображения

Проблема цветопередачи возникает во время вывода изображения на некоторое устройство при его одновременном масштабировании. Исходная картинка (источник) имеет определенные размеры и палитру используемых цветов (глубину цвета). Результирующая картинка (получатель, приемник) может иметь другой размер и другую глубину цвета.

Если предположить, что глубина цвета источника и получателя одинаковы, то задача отображения сводится к замене пикселей. Если получатель имеет меньший размер, несколько пикселей оригинала заменяются одним пикселем получателя. Цвет результирующего пикселя рассчитывается суммированием отдельно красной, зеленой и синей составляющей всех исходных пикселей и делением полученных величин на количество просуммированных пикселей.

На рис. 44, а) показан фрагмент картинки, уменьшаемой в *2* раза (увеличено). Каждые *4* пикселя источника, образующие квадрат, заменяются одним пикселем получателя. Составляющие цвета исходных четырех пикселей складываются и сумма делится на *4*. Новые составляющие цвета используются для записи одного пикселя получателя. Результат — полное сглаживание цвета, потеря смысла картинки (рис. 44, б). На рис. 44, в) и г) ситуация немного лучше — картинка получателя имеет структуру, отдаленно напоминающую оригинал.

Рис. 44. Пересчет цвета при уменьшении количества пикселей

Картинку можно сгладить, сделать расплывчатой, если учитывать большее число пикселей изображения источника. На рис. 45, б) показан результат уменьшения четырех пикселей в один. На рис. 45, в) число учитываемых пикселей не *4* а *9*, т. к. для расчета пикселя приемника учитывалась область размером *3×3*.

При увеличении изображения не в целое число раз ситуация осложняется. В целом изображение будет лучше, если учитывать большее число пикселей. При масштабировании в нецелое число раз количество учитываемых пикселей по-

лучается также нецелым. Его следует округлить до ближайшего большего целого. Так, если картинка масштабируется в *2,3* раза, следует учитывать область размером *3×3* или больше. Начало области при этом также округляется. Так, в случае уменьшения в *2,3* раза пиксель приемника *(2, 2)* должен учитываться областью, начинающейся с пикселя *(1, 1)*, так как $2/2{,}3 = 0{,}87 \approx 1$, а пиксель *(4, 4)* — начиная с пикселя *(2, 2)*, так как $4/2{,}3 = 1{,}74 \approx 2$. Нумерация пикселей здесь от единицы.

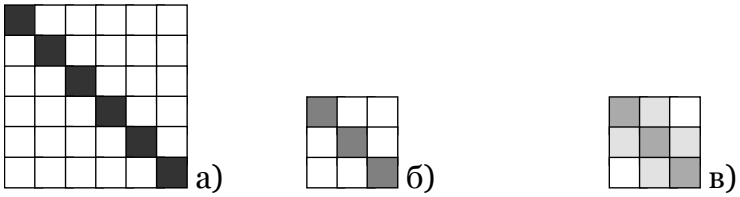


Рис. 45. Учет большего числа пикселей при уменьшении размера картинки

При увеличении изображения расчет результирующих пикселей проще. В этом случае требуется определить цвет дополнительных пикселей, появляющихся на изображении получателя. Самое простое решение — каждый пиксель источника продублировать ***n*** пикселями получателя, где ***n*** — коэффициент масштабирования. При этом изображение получателя будет обладать заметной клеточной структурой (рис. 46, б).

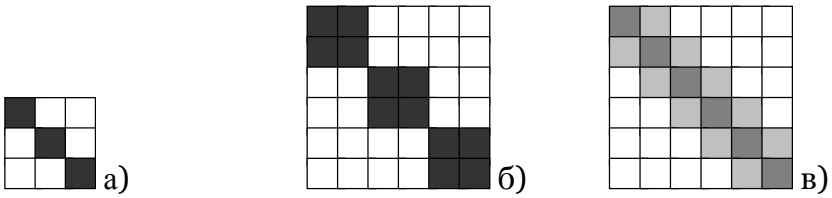


Рис. 46. Увеличение картинки в 2 раза

Для получения более качественного результата можно интерполировать цвет с использованием цветов пикселей, непосредственно прилегающих к увеличиваемому (рис. 46, в).

В заключение отметим, что результат масштабирования картинки в значительной степени зависит от ее структуры. Картинка с неупорядоченной структурой типа фотографии масштабируется лучше, чем картинка с упорядоченной структурой типа текста.

Масштабирование в целое число раз в англоязычной литературе обозначается словом ***zoom***, а масштабирование в нецелое число раз — ***transfocation***.

Качество растровой картинки

Разрешение и глубина цвета определяют качество изображения, которое может быть получено на экране растрового монитора или при печати. Глубина цвета имеет значение для правильной передачи полноцветных картинок, таких, например, как фотографии. Бóльшее количество оттенков цвета дает лучшее изображение даже при более низком разрешении. Высокое разрешение в этом случае способствует передаче мелких деталей.

Разрешающую способность точечных устройств принято измерять в ***точках на дюйм*** (***dpi*** — ***dots per inch***). Для монитора эта характеристика равна *60–120* ***dpi***. Для сравнения, матричный принтер имеет разрешение *72–144* ***dpi***, а лазер-

ный или струйный принтер — не менее *600* ***dpi***. Поэтому, например, один и тот же шрифт выглядит по-разному на экране и на печатном документе.

Одно и то же изображение, представленное с разной глубиной цвета, будет выглядеть совершенно по-разному. Для сравнения на рис. 47 (см. также цветную вкладку) представлено одно и то же изображение с разной глубиной цвета — *24* (слева), *8* (в центре) и *4* (справа). Разрешение всех изображений *200* ***dpi***.

Рис. 47. Влияние глубины цвета на качество картинки

При уменьшении глубины цвета изображение приобретает черты плакатности, так как все большая часть пикселей заменяется одним и тем же цветом.

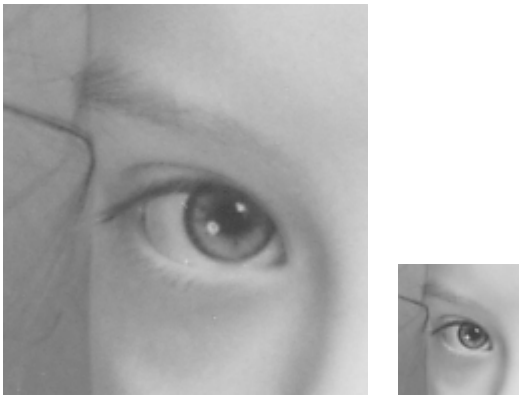


Рис. 48. Влияние разрешения на качество картинки

В то же время уменьшение разрешения изображения практически не ухудшает качества картинки (рис. 48, слева разрешение *300* ***dpi***, справа — *100* ***dpi***). При сканировании изображение уменьшается. При печати цветной фотографии с разным разрешением различие в качестве проявится только при увеличении изображения против исходного размера.

Контрольные вопросы

1. В чем разница между растровой и векторной графикой? Как эта разница влияет на качество вывода?
2. В чем заключаются недостатки растровой графики?
3. В чем заключается anti-aliasing?
4. В чем суть дизеринга?
5. Как влияют разрешение и глубина цвета на качество картинки?
6. Как выполняется сложение цветов?

Графика в Windows

В *Windows* в распоряжение программиста предоставляется высокое разрешение и глубина цвета монитора. Кроме того, система предлагает огромные возможности для графического вывода на самые разнообразные устройства.

Одним из основных понятий *Windows* является понятие «окна». Под окном понимается некоторая прямоугольная (в общем случае) область, внутри которой может располагаться любой графический вывод. При этом одни окна становятся окнами приложений, другие — элементами управления, третьи — рисунками. В принципе, рисовать в *Windows* можно непосредственно на экране, так как экран *Windows* также представляет собой окно. Но обычно для рисования используется одно из окон приложения.

Окно приложения любого типа поделено в *Windows* на две части. Одна часть называется ***системной***, она рисуется и обновляется операционной системой. В системную часть окна входят заголовок, системное меню и рамка. Вторая часть называется ***клиентской*** и она предназначена для создания внешнего вида приложения. Обычно клиентская часть заполняется всевозможными элементами управления, такими, как кнопки, списки, надписи и т. п. При необходимости графическая часть приложения также может быть нарисована внутри клиентской части какого-нибудь окна.

Прежде, чем начать рисование, нужно получить ***контекст устройства***. Это дескриптор, указывающий на область вывода и все, что с ней связано. Почти все графические функции *Windows* требуют контекст устройства в качестве обязательного параметра, потому что он указывает, ***где*** рисовать. После завершения рисования контекст следует ***освободить***.

Windows устроена так, что все графические объекты, которые будут рассмотрены далее, располагаются в так называемой ***системной памяти***. Перед использованием графического объекта его нужно создать. При этом он, в виде некоторой структуры, размещается в системной памяти, а пользователю возвращается дескриптор объекта для последующей работы. *Крайне важно освобождать объект после его использования, чтобы избежать истощения системной памяти.* Так, например, если создать всего лишь *1000* небольших картинок, размером со значок, системная память может закончиться. При этом начнут пропадать значки и меню приложений и в конце концов система сообщит, что ей «*существенно не хватает ресурсов*».

Для программирования графики в *Windows* несравненно больше возможностей, чем в *MS-DOS*. Обычно для графических программ используется среда программирования типа *Microsoft Visual C++* или аналогичная. Графические функции, определенные в *Windows*, становятся доступны программе на *Cи* сразу, так как заголовочные файлы *Windows* включаются в код автоматически. Лучшим описанием графических функций является справочная система *MSDN Library* (*Microsoft Developer Network*), раздел «*Platform SDK*».

Языки программирования, такие, как *Visual Basic*, скрывают от программиста многие детали, связанные с контекстом и графическими объектами. Использование графических средств, определенных в языках, однако, ограничивает скорость графического вывода, поэтому в некоторых случаях необходимо обращаться к системным функциям. Знание графических возможностей и особенностей вывода графики в *Windows* при этом крайне желательно.

Программирование тестов

Все приведенные в тексте учебного пособия примеры кода тестировались в среде программирования *Microsoft® Visual C++ 6.0*.

Тестирующее приложение создается в следующем порядке:

1. Открыть среду, выбрать в меню *File—New*, выбрать вкладку *Projects*, выбрать *Win32 Application*;
2. В поле *Location* ввести каталог, в котором будет располагаться каталог тестового приложения, а в поле *Project name* ввести название тестового проекта, например *KGTest*;
3. Щелкнуть кнопку *OK*;
4. Выбрать «*A typical "Hello World!"* application», щелкнуть кнопку *Finish*, щелкнуть кнопку *OK*.
5. Если не открыто, открыть окно *Workspace* кнопкой *Workspace*  , расположенной на панели инструментов. Окно *Workspace* располагается вертикально в левой части среды.
6. Выбрать в окне *Workspace* вкладку *FileView*. Раскрыть папку *Source Files*, дважды щелкнуть на название файла *KGTest.cpp*. Откроется окно модуля кода, которое содержит все необходимые для старта приложения функции.

В самом начале модуля следует добавить два заголовочных файла к имеющимся двум так, чтобы получилось:

```
#include "stdafx.h"
#include "resource.h"
#include <winuser.h>
#include <commdlg.h>
```

Ниже объявим прототипы функций для тестирующего кода:

```
void Draw(HWND);
void Metafile(HWND, LPCSTR, LPCSTR);
void Printer(HWND);
void Choose(HWND);
```

Переместимся в конец модуля (*Ctrl+End*) и добавим сами функции:

```
/* тестирование графических примитивов */
void Draw(HWND hWnd) {
}
/* тестирование метафайла */
void Metafile(HWND hWnd, LPCSTR path, LPCSTR desc) {
}
/* тестирование печати на принтер */
void Printer(HWND hWnd) {
}
/* тестирование диалога для выбора шрифта */
void Choose(HWND hWnd) {
}
```

Далее нужно модифицировать проект с тем, чтобы меню приложения отражало потребности тестирования. Перейдем на вкладку *ResourceView* в окне *Workspace*. Раскроем папку *KGTest resources*, раскроем папку *Menu*. Дважды щелкнем на значок ресурса с названием *IDC_KGTest*... Откроется меню приложения:

Щелкнем правой кнопкой на пустой пункт меню и выберем *Properties*. Появит-

ся диалог свойств. В поле *Caption* введем название пункта «*Draw*», выключим флажок *Pop-up*. Таким же образом добавим пункты «*Metafile*», «*Printer*», «*Choose*». Щелкнем на вкладку *FileView*, чтобы перейти в окно кода.

Выше по тексту найдем процедуру окна, имеющую название *WndProc*. Здесь происходят все основные действия. В нее поступают сообщения *Windows*, которые следует разобрать и что-то сделать в ответ.

Найдем разбор команд — оператор выбора `switch (wmId)`. Добавим следующий код (выделен волнистой линией):

```
case IDM_EXIT:
   DestroyWindow(hWnd);
   break;
case ID_DRAW:
   Draw(hWnd);
   break;
case ID_METAFILE:
   Metafile(hWnd, "C:\\TEST.emf", "KGTest\0TEST\0\0");
   break;
case ID_PRINTER:
   Printer(hWnd);
   break;
case ID_CHOOSE:
   Choose(hWnd);
   break;
default:
```

Найдем в процедуре *WndProc* разбор сообщения `WM_PAINT`. Здесь находится код, который рисует графический вывод в окно. Лишний код нужно удалить. Это место используется для того, чтобы сразу увидеть результат графического вывода, который к тому же не стирается. Код в этом месте ***должен всегда располагаться между функциями*** `BeginPaint` и `EndPaint`:

```
case WM_PAINT:
   hdc = BeginPaint(hWnd, &ps);
      /* место для графического вывода */
   EndPaint(hWnd, &ps);
   break;
```

Для тестирования перьев, кистей и других графических объектов код функции *Draw* должен содержать их объявления. Графические объекты в тексте пособия всегда фигурируют под одними и теми же именами — перья называются `hp` и `exhp`, кисти называются `hb` и `exhb`, шрифты называются `hf` и `exhf`. Префикс `ex` в названиях переменных обозначает «*бывший объект*». При необходимости тестирующий должен сам объявить, выбрать и удалить объект.

Тестирующий код функции *Draw* ***должен всегда располагаться между функциями*** `GetDC` и `ReleaseDC`:

```
void Draw(HWND hWnd) {
   HDC hdc = GetDC(hWnd);
      /* место для графического вывода */
   ReleaseDC(hWnd, hdc);
}
```

Контекст устройства

Как было сказано, прежде, чем рисовать, нужно получить контекст устройства. ***Контекст устройства*** — это структура, описывающая *графические объ-*

екты, связанные с ними *атрибуты*, а также *режимы*, которые используются для графического вывода *на конкретное устройство*.

Контекст устройства предназначен для обеспечения ***независимости графического вывода от устройства***. Такая независимость является важнейшей чертой Windows. Мы хотим нарисовать красивое полноцветное изображение и отобразить результат нашего творчества на каком-то устройстве. Для отображения используются разнообразные устройства — монитор, принтер, другие. Каждое из устройств характеризуется своими графическими возможностями. Например, часто используется черно-белый принтер.

Во время вывода происходит взаимодействие минимум двух программных компонентов — ядра графической системы *Windows*, `GDI.DLL`, и драйвера устройства. Ядро графической системы работает с независимыми от устройства структурами, которые позволяют определить изображение в максимально качественной форме. Драйвер, получая информацию от ядра, отображает графику так, как может. При этом происходит преобразование *логических* характеристик изображения в *физические*. ***Логические характеристики*** — те, которые мы желали бы получить. ***Физические характеристики*** — те, которые фактически доступны на данном устройстве. С помощью этих двух компонентов система обеспечивает возможно более точное соответствие между тем, что мы ожидаем, и тем, что реально получим.

Существует ***четыре типа контекстов*** — *контекст дисплея*, *контекст принтера*, *контекст памяти* и *информационный контекст*. Можно также рассматривать в качестве пятого *контекст метафайла*.

Контекст дисплея используется для вывода на экран. Его можно получить от текущего окна приложения при помощи функции

```
HDC GetDC(HWND hWnd);
```

Это наиболее важный контекст при создании графических приложений. Параметр `hWnd` — это дескриптор окна, в которое направляется вывод. В качестве окна используется одно из окон приложения или специально созданное окно.

Контекст принтера создается функцией

```
HDC CreateDC(драйвер, устройство, порт, параметры);
```

Параметр ***драйвер*** — имя драйвера, "`WINSPOOL`" или "`WINSPL16`". Параметр ***устройство*** записывается так, как оно отображается в диспетчере принтеров, например "`hp LaserJet 1015`". Параметр ***порт*** для *Win32* равен `NULL`, для *Win16* порт обычно "`LPT1:`". Дополнительные параметры можно не указывать и принять равными `NULL`.

Контекст памяти используется для рисования картинок в памяти (см. стр. 74). ***Информационный контекст*** используется для получения информации об устройстве и объектах (см. стр. 104).

После того, как контекст устройства получен, можно рисовать с получением вывода в контексте. Так, если получен контекст принтера, вывод будет получен на принтере, а если получен контекст памяти, вывод будет осуществляться на картинку в памяти.

После завершения графических операций контекст, полученный при помощи `GetDC`, нужно освободить при помощи функции

```
int ReleaseDC(HWND hWnd, HDC hDC);
```

Дополнительно см. раздел «*Управление графическим выводом*», стр. 80.

Контекст устройства устанавливает ***пять графических режимов***. Они определяют, как смешиваются цвета, как происходит преобразование логических единиц в физические и т. п. Эти режимы представлены в следующей таблице:

Режим	***Определяет***
Background	Смешивание фона картинок и текста с цветом окна
Drawing	Смешивание цвета перьев, кистей, картинок и текста с цветом окна
Mapping	Отображение из логической (мировой) системы на устройство
Polygon-fill	Использование растра кисти для закраски фигур
Stretching	Смешивание цветов картинки и окна при масштабировании

Режимы могут быть получены для анализа при помощи функций типа **`Get`**
`GetBkMode`, **`GetROP2`**, **`GetMapMode`**, **`GetPolyFillMode`**, **`GetStretchBltMode`**,
и изменены при помощи функций типа **`Set`**
`SetBkMode`, **`SetROP2`**, **`SetMapMode`**, **`SetPolyFillMode`**, **`SetStretchBltMode`**.
Дополнительно см. раздел «Графические режимы», стр. 55.

В контекст включаются следующие ***семь типов графических объектов***:
перья (***pen***) для рисования линий;
кисти (***brush***) для закраски;
палитры (***palette***) для определения допустимых цветов;
шрифты (***font***) для вывода текста (надписей);
пиксельные наборы (***bitmap***) для создания картинок;
пути (***path***) для создания сложных фигур;
области (***region***) для создания комбинированных фигур.

Получение контекста дает пользователю ***стандартные графические объекты***, или *объекты по умолчанию*. Для целей рисования пользователь создает необходимые новые графические объекты, заменяя ими имеющиеся стандартные. Так, по умолчанию в контексте имеется черное перо сплошной линии. Если для вывода требуется перо другого стиля, его нужно создать и заменить им стандартное.

Замена графического объекта в контексте называется ***выбором объекта***. Он выполняется при помощи функции

```
HGDIOBJ SelectObject(HDC hdc, HGDIOBJ hgdiobj);
```

Здесь **`hdc`** — контекст устройства, **`hgdiobj`** — дескриптор объекта, полученный при его создании. Функция не только выбирает объект в контекст, но и возвращает дескриптор текущего, ***ex***-объекта. Дополнительно см. раздел «*Выбор и изменение графических объектов*», стр. 81.

Примечание. После использования нового объекта старый нужно вернуть в контекст, чтобы не нарушить работу системы по обновлению изображения клиентской части.

После использования объекта его нужно удалить из контекста и из системной памяти. Первая операция выполняется при помощи **`SelectObject`** (при этом выбирается другой объект), а вторая — при помощи функции

```
BOOL DeleteObject(HGDIOBJ hObject);
```

Параметр **`hObject`** — идентификатор графического объекта.

Примечание. Некоторые объекты не подлежат удалению, например стандартные (фондовые, stock).

Цвет в Windows

В *Windows* цвет записывается как целое число, состоящее из *4*-х байт. Эта структура описывает системный или логический цвет. ***Логический цвет*** определяет относительные значения интенсивности основных цветов RGB, которые находятся в диапазоне ***0..255***. Если видеоадаптер поддерживает прозрачность цвета, логический цвет может также определять прозрачность *Alpha*. Распределение составляющих цвета по байтам и битам приведено на рис. 49.

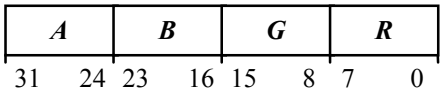


Рис. 49. Хранение составляющих цвета в 4-х байтной структуре Long

Системные цвета — номера от *0* до *28*, закрепленные за тем или иным элементом графического интерфейса *Windows*. Цвет, в виде составляющих RGB, хранится в системных таблицах, и настраивается пользователем.

Чтобы отличить системный цвет от логического, старший байт (***A***) устанавливается в ***80h***. Номер системного цвета при этом записывается в младший байт (***B***). Получить цвет элемента интерфейса можно при помощи функции

```
DWORD GetSysColor(элемент);
```

Параметр ***элемент*** — константа, которая выбирается из таблицы «Системные цвета *Windows*» на стр. 116.

Тип переменной для цвета выбирается один из следующих:

```
typedef DWORD COLORREF;
typedef DWORD *LPCOLORREF;
```

Задать значения этих типов можно при помощи макроса RGB:

```
COLORREF RGB(красный, зеленый, синий);
```

Здесь ***красный, зеленый*** и ***синий*** задают значения ***0..255*** составляющих цвета.

Пример задания цвета при создании черного, красного и розового перьев:

```
HPEN hp = CreatePen(PS_SOLID, 1, 0);
HPEN hp = CreatePen(PS_SOLID, 1, 255);
HPEN hp = CreatePen(PS_SOLID, 1, RGB(255, 32, 112));
```

Извлечь составляющие цвета можно при помощи макросов:

```
BYTE GetRValue(rgb_значение);
BYTE GetGValue(rgb_значение);
BYTE GetBValue(rgb_значение);
```

Аппроксимация и дизеринг цвета

Учитывая возможные значения составляющих цвета, глубина логического цвета равна *24*, а общее количество цветов — *16 777 216*. Несмотря на то, что современные цветные устройства поддерживают огромное количество цветов, всегда найдется такое, на котором можно отобразить лишь небольшое их количество. Например, черно-белый принтер может иметь *256* цветов (градаций серого).

Когда приложение запрашивает цвет пера или текста, отсутствующий в устройстве, система подбирает наиболее подходящий. Этот процесс называется ***аппроксимацией***. Чтобы определить, какой цвет устройства ближе всего к требуемому логическому, следует использовать функцию

```
COLORREF GetNearestColor(контекст, требуемый_цвет);
```

Эта функция возвращает ближайший по яркости цвет, имеющийся на устройстве (в данном контексте). Так, если запросить красный цвет в контексте черно-белого принтера, система вернет скорее всего черный.

Когда приложение создает сплошную кисть, система может выполнить ***дизеринг***, подбирая чередование имеющихся цветов по некоторому алгоритму. Результат дизеринга не всегда предсказуем и может оказаться неожиданным. Дизеринг выполняет драйвер устройства. Повлиять на алгоритм дизеринга невозможно, но приложение может выполнить дизеринг самостоятельно, создав кисть со штриховкой, которая использует цвета, имеющиеся в контексте устройства. Дополнительно см. раздел «*Палитры*», стр. 62.

Растровые операции (*ROPs*)

Есть два аспекта графического вывода — план (*foreground*) и фон (*background*).

План — это первичный, передний, главный цвет рисования, — цвет пера или кисти, цвет текста, цвет в пиксельном наборе (картинке).

Фон — вторичный, вспомогательный цвет, заполняющий промежутки, остающиеся после наложения плана — в разрывах прерывистых линий, между штрихами узора, между штрихами символов, внутри замкнутых областей.

Растровые операции (*raster operation — ROP*) описывают, как комбинируется цвет плана (источник) с цветом на поверхности вывода (приемник). Кроме цветов источника и приемника, в операции также могут принимать участие цвет кисти приемника и маска — вспомогательный черно-белый пиксельный набор.

Результирующий цвет каждого рисуемого пикселя формируется отдельно. В зависимости от количества учитываемых цветов различают двух, трех и четырехместные растровые операции, для обозначения которых используются ***коды растровых операций***, образующие группы **ROP2**, **ROP3** и **ROP4** (табл. 8).

Табл. 8. Группы растровых операций

Группа ROP	*Значение*	*Описание*
ROP2	1—16	Источник (перо или кисть) комбинируется с приемником. Всего 16 операций (табл. 9).
ROP3	0x00—0xFF	Кисть и пиксельный набор-источник комбинируются с приемником. Всего 256 операций (табл. 10).
ROP4	—	Если бит маски равен 1 — выполняется ROP3 для плана, если бит маски равен 0 — выполняется ROP3 для фона.

Растровые операции записываются в (обратной) польской нотации. Для обозначения цветов используют прописные буквы:
D — приемник (*Destination bitmap*),
P — перо или кисть (*Pen*, *Pattern*),
S — пиксельный набор-источник (*Source bitmap*).

В операциях **ROP2** участвуют D и P, в операциях **ROP3** участвует также S.

Для комбинирования цветов используются побитовые логические операции **AND**, **NOT**, **OR** и **XOR**. Они обозначаются строчными буквами a, n, o, x.

Таким образом, запись Dn означает инверсию приемника, запись DPo обозначает комбинацию приемника и пера (P or D), запись DPSoo — комбинацию всех цветов ((S or P) or D), а запись DSPDSanaxxn (**ROP3** операция **E9**, код **0x00E95CE6**) — операцию not ((((not (S and D)) and P) xor S) xor D).

Операции **ROP2** выполняются в функциях рисования пером или кистью.

Табл. 9. Растровые операции ROP2

Константа	*Значение*	*Операция*	*Результирующий цвет*
R2_BLACK	1	0	Цвет приемника 0 (черный)
R2_NOTMERGEPEN	2	DPon	NOT (источник OR приемник)
R2_MASKNOTPEN	3	DPna	(NOT источник) AND приемник
R2_NOTCOPYPEN	4	Pn	NOT источник
R2_MASKPENNOT	5	PDna	источник AND (NOT приемник)
R2_NOT	6	Dn	NOT приемник
R2_XORPEN	7	DPx	источник XOR приемник
R2_NOTMASKPEN	8	DPan	NOT (источник AND приемник)
R2_MASKPEN	9	DPa	источник AND приемник
R2_NOTXORPEN	10	DPxn	NOT (источник XOR приемник)
R2_NOP	11	D	приемник
R2_MERGENOTPEN	12	DPno	(NOT источник) OR приемник
R2_COPYPEN	13	P	источник
R2_MERGEPENNOT	14	PDno	источник OR (NOT приемник)
R2_MERGEPEN	15	DPo	источник OR приемник
R2_WHITE	16	1	Цвет приемника 1 (белый)

Операции **ROP3** выполняются в функциях вывода картинок **BitBlt, StretchBlt**.

Табл. 10. Поименованные растровые операции ROP3

Константа	*Значение*	*Операция*	*Результирующий цвет*
BLACKNESS	**0x00000042**	0	Цвет приемника 0 (черный)
NOTSRCERASE	**0x001100A6**	DSon	NOT (источник OR приемник)
NOTSRCCOPY	**0x00330008**	Sn	NOT источник
SRCERASE	**0x00440328**	SDna	(NOT приемник) AND источник
DSTINVERT	**0x00550009**	Dn	NOT приемник
PATINVERT	**0x005A0049**	DPx	кисть приемника XOR приемник
SRCINVERT	**0x00660046**	DSx	источник XOR приемник
SRCAND	**0x008800C6**	DSa	источник AND приемник
MERGEPAINT	**0x00BB0226**	DSno	(NOT источник) OR приемник
MERGECOPY	**0x00C000CA**	PSa	источник AND кисть приемника
SRCCOPY	**0x00CC0020**	S	источник
SRCPAINT	**0x00EE0086**	DSo	источник OR приемник
PATCOPY	**0x00F00021**	P	кисть приемника
PATPAINT	**0x00FB0A09**	DPSnoo	(NOT источник) OR кисть OR приемник
WHITENESS	**0x00FF0062**	1	Цвет приемника 1 (белый)

Операции **ROP4** выполняются функцией **MaskBlt**. Для формирования кода **ROP4** следует использовать макрос

DWORD MAKEROP4(*ROP3_план, ROP3_фон*);

В табл. 11 показано, как формируется код операции **ROP2**. Первые два столбца задают все возможные сочетания источника и приемника. Следующие 16 столбцов определяют все возможные исходы этих сочетаний при применении к ним одних и тех же логических операций, и образуют коды операций от 0 до 15. Читать код нужно в столбце операции снизу вверх. Так, для операции DPon код операции равен **0x01**, а для операции PDno код операции равен **0x0D**. К этому коду нужно прибавить единицу, чтобы получить значение константы операции.

Табл. 11. Формирование кодов операций ROP2

P	*D*	*0*	*DPon*	*DPna*	*Pn*	*…*	*P*	*PDno*	*DPo*	*1*
0	0	0	1	0	1		0	1	0	1
0	1	0	0	1	1		0	0	1	1
1	0	0	0	0	0		1	1	1	1
1	1	0	0	0	0		1	1	1	1

Аналогичная таблица из 16 строк может быть составлена и для операций **ROP3**. В этом случае код операции как есть записывается в третий байт значения константы операции.

Растровые операции избавляют программиста от необходимости выполнять преобразования самостоятельно. Множество растровых операций покрывает все возможные комбинации цветов, однако не все они имеют четко выраженный смысл, и призваны лишь для обеспечения универсальности.

На практике часто используется операция **R2_COPYPEN=13**, которая выполняет обычное рисование пером или кистью. Операция **R2_XORPEN=7** обладает свойством сохранять приемник в результирующем цвете. Одно и то же изображение, нарисованное дважды при помощи этой операции, возвращает исходное состояние приемника. Однако при этом результирующий цвет равен инверсии цвета пера. Поэтому имеет смысл применить операцию **R2_NOTXORPEN=10**, которая инвертирует результирующий цвет. Эти операции позволяют динамически рисовать изображения на незакрашенной (белой) поверхности.

Графические режимы

Графические режимы определяют, что получится на поверхности устройства вывода, исходя из цвета ее пикселей и цвета пера, кисти, картинки или текста, а также как логические единицы измерения, используемые для рисования, преобразуются в физические единицы устройства.

Background (фон)

Определяет, как заполняется фон перед выводом текста, кисти с узором или несплошного пера. Режим фона устанавливает функция

int SetBkMode(*контекст, режим*);

Параметр ***контекст*** — идентификатор контекста. Параметр ***режим*** выбирается из табл. 12. Возвращает предыдущий режим или *0* при ошибке. Режим влияет на вывод перьев, созданных при помощи функции **CreatePen**, но не влияет на вывод перьев, созданных при помощи функции **ExtCreatePen**. Режим по умолчанию **OPAQUE**.

Табл. 12. Режимы фона

Режим фона	*Значение*	*Описание*
TRANSPARENT	1	Фон не изменяется
OPAQUE	2	Фон заполняется текущим цветом фона

Для управления фоном используются функции:

BOOL PatBlt(*контекст, x, y, w, h, режим*);

Закрашивает область, которая начинается в точке ***x, y*** и имеет размер ***w×h***, текущей кистью контекста. Результирующий цвет определяет параметр ***режим***, который может принимать значения **BLACKNESS**, **DSTINVERT**, **PATINVERT**, **PATCOPY** и **WHITENESS** (табл. 10, стр. 54).

COLORREF SetBkColor(*контекст, цвет*);

Устанавливает новый цвет фона и возвращает текущий. Действует на последующий вывод промежутков в прерывистых линиях и фона символов текста.

Drawing (план)

Определяет цвет графического вывода. Цвет пера и кисти смешивается с цветом, который уже нарисован на поверхности вывода (например, на экране). Режим устанавливает функция

int SetROP2(*контекст, ROP2*);

Параметр ***ROP2*** выбирается из табл. 9. Возвращает предыдущий режим или *0* при ошибке. Режим по умолчанию — **R2_COPYPEN**.

Mapping (отображение)

Определяет единицы измерения в пространстве вывода и ориентацию координатных осей. Режим устанавливает функция

int SetMapMode(*контекст, режим*);

Параметр ***режим*** выбирается из табл. 13. Возвращает предыдущий режим или *0*.

Табл. 13. Режимы отображения

Режим	*Значение*	*Логическая единица отображается в*
MM_TEXT	1	физическую (пиксель). Ось Y направлена *вниз*.
MM_LOMETRIC	2	0,1 мм. Ось Y направлена вверх.
MM_HIMETRIC	3	0,01 мм. Ось Y направлена вверх.
MM_LOENGLISH	4	0,01 дюйма. Ось Y направлена вверх.
MM_HIENGLISH	5	0,001 дюйма. Ось Y направлена вверх.
MM_TWIPS	6	1/1440 дюйма (*twip*). Ось Y направлена вверх.
MM_ISOTROPIC	7	равномерные единицы вдоль осей X и Y
MM_ANISOTROPIC	8	произвольные единицы вдоль осей X и Y

Режим по умолчанию **MM_TEXT**. В режимах **MM_ISOTROPIC** и **MM_ANISOTROPIC** размеры единиц в пространстве вывода определяются на основе соотношения между размером *окна вывода* и размером *порта вывода*. В режиме **MM_ISOTROPIC** единицы вдоль осей X и Y одинаковы, что обеспечивает соотношение *1:1*.

Размеры окна вывода устанавливает функция

BOOL SetWindowExtEx(*контекст, ширина, высота, экс-размер*);

Параметры ***ширина*** и ***высота*** определяют размеры окна вывода в логических единицах. Параметр ***экс-размер*** — переменная типа **LPSIZE** (стр. 116), в которую возвращается текущий размер окна вывода. Может быть **NULL**.

Размеры порта вывода устанавливает функция

BOOL SetViewportExtEx(*контекст, ширина, высота, экс-размер*);

Параметры ***ширина*** и ***высота*** определяют размеры окна вывода в физических единицах устройства (пикселях). Параметр ***экс-размер*** имеет тот же смысл, что и для предыдущей функции.

Функция **SetWindowExtEx** должна вызываться первой.

Дополнительно см. раздел «*Преобразования*», стр. 102

Polygon-fill (закрашивание)

Определяет, как закрашиваются замкнутые области фигур.

Режим устанавливает функция

int SetPolyFillMode(*контекст, режим*);

Параметр ***режим*** выбирается из табл. 14. Возвращает предыдущий режим или *0*.

Табл. 14. Режимы закрашивания

Режим	***Значение***	***Закрашивается*** *каждая замкнутая область*
ALTERNATE	1	между четными и нечетными сторонами многоугольника на каждой растровой линии
WINDING	2	которую перо обошло нечетное число раз

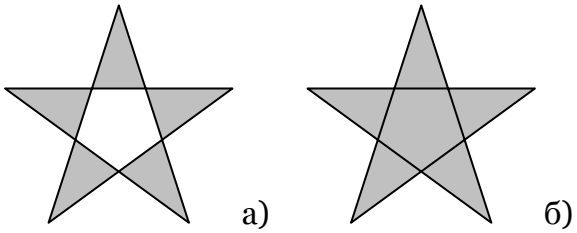


Рис. 50. Режимы закраски областей ALTERNATE (а) и WINDING (б)

Закрашивается точка, или нет, определяет число *winding*. Это число подсчитывает ребра, пересекаемые лучом, выпущенным из некоторой точки в направлении оси X. В режиме **ALTERNATE** считаются все ребра. Если число пересечений нечетно, точка закрашивается. В режиме **WINDING** ребра, направленные вдоль оси Y, увеличивают число, а ребра, направленные против оси Y, уменьшают его. Закрашивается точка, для которой число ненулевое.

Stretching (масштабирование)

Определяет, как растягиваются и сжимаются картинки функцией **StretchBlt**. Режим устанавливает функция

int SetStretchBltMode(*контекст, режим*);

Параметр ***режим*** выбирается из табл. 15. Возвращает предыдущий режим или *0*.

Табл. 15. Режимы масштабирования

Режим	***Значение***	***Описание***
BLACKONWHITE	1	Цвет экрана AND цвет исключаемого пикселя. Для ч/б картинки сохраняет черные пиксели в ущерб белым.
WHITEONBLACK	2	Цвет экрана OR цвет исключаемого пикселя. Для ч/б картинки сохраняет белые пиксели в ущерб черным.
COLORONCOLOR	3	Исключаемые пиксели удаляются.
HALFTONE	4	Средний цвет приемника аппроксимируется к среднему цвету источника.

Режимы **BLACKONWHITE** и **WHITEONBLACK** используются для черно-белых картинок. Режим **COLORONCOLOR** используется для сохранения цвета в цветных картинках. Режим **HALFTONE** используется для создания плавного перехода цвета, однако он медленнее. После установки режима **HALFTONE** следует вызвать функцию **SetBrushOrgEx** (стр. 61) для коррекции положения растра кисти. Дополнительно должна быть создана полутоновая палитра при помощи функции

HPALETTE CreateHalftonePalette(*контекст*);

Функция возвращает палитру, которую нужно выбрать в контекст устройства при помощи функции **SelectPalette**, и реализовать при помощи функции **RealizePalette**.

Дополнительно см. раздел «*Палитры*».

Графические объекты

Перья

Перо определяет, как рисуются прямые и кривые линии (линии как таковые), а также линии, являющиеся граничными, контурными при рисовании замкнутых фигур, таких, как эллипс или многоугольник.

Различают *косметические* (*cosmetic*) и *геометрические* (*geometric*) перья.

Косметические перья используются тогда, когда требуются линии фиксированной ширины. Например, приложение *AutoCAD* использует косметические перья для рисования скрытых, осевых и размерных линий, ширина которых не превышает *0,015..0,022* дюйма (*0,38..0,56* мм) независимо от масштаба изображения. Косметические линии рисуются в *3..10* раз быстрее геометрических.

Атрибуты косметического пера – это *стиль линии*, *ширина* и *цвет*.

Стиль линии определяет изображение линии по ее длине. Предусмотрено 9 стандартных стилей (табл. 17). Стиль «*нулевая*» скрывает линию, когда она рисуется, но не нужна. Стиль «*ширина внутри*» означает, что ширина линии располагается целиком во внутренней части замкнутого контура, такого, как прямоугольник. Не имеет смысла для незамкнутых контуров. Стиль «*пользовательская*» использует данные о длине штрихов и промежутков, задаваемые пользователем. Стиль «*через точку*» предназначен для пиксельного вывода. Рисует пиксели только с нечетными координатами экрана.

Ширина задается в единицах устройства вывода (пикселях).

Косметическое перо создается при помощи функций

HPEN CreatePen(*стиль, ширина, цвет*);

HPEN CreatePenIndirect(*логическое_перо*);

Параметр ***логическое_перо*** — указатель на структуру **LOGPEN** (стр. 116).

Функции возвращают ***дескриптор*** объекта или ***0***, если перо не создано. Концы и соединения косметического пера *всегда круглые*.

С помощью этих функций нельзя создать стиль **PS_ALTERNATE**. Следует использовать функцию **ExtCreatePen**. Пример создания косметического пера с пользовательским стилем линии:

Листинг 1. Пользовательский стиль линии

```
ULONG us[] = { 10, 1 };
LOGBRUSH lb = { BS_SOLID, 0, 0};
HPEN hp = ExtCreatePen(PS_COSMETIC | PS_USERSTYLE, 1, &lb, 2, us);
SelectObject(hdc, hp);
Rectangle(hdc, 0, 0, 120, 40);
```

Геометрические перья используются тогда, когда требуется масштабируемые линии с характерными концевыми точками и соединениями в углах. Как правило, это линии, ширина которых превышает один пиксель. Они используются, например, для рисования гистограмм.

Атрибуты геометрического пера — *стиль линии*, *ширина*, *цвет*, *узор* (*pattern*), *штриховка* (*hatch*), *тип концевых точек* и *тип соединений*.

Атрибуты *узор* и *штриховка* определяют узор кисти. Атрибут *штриховка* — это узор из прямых линий, а атрибут *узор* — произвольный.
Геометрическое перо создается функцией

HPEN ExtCreatePen(*стиль, ширина, кисть, размер_массива, массив*);

Параметр ***стиль*** — сумма констант: тип пера (табл. 16), стиль линии (табл. 17), тип концевых точек (табл. 18) и тип соединений в углах (табл. 19).

Для определения специфического стиля линии используются два последних параметра. Они действительны, если параметр ***стиль*** равен **PS_USERSTYLE**. См. также описание структуры **EXTLOGPEN** на стр. 116. Тип соединений предназначен только для *непрерывных линий* (прямоугольник, многоугольник, полилиния).

Параметр ***ширина*** задает ширину линии пера в *логических единицах*. О логических единицах см. раздел «*Преобразования*», стр. 102.

Параметр ***кисть*** — указатель на структуру **LOGBRUSH** (стр. 116), которая описывает логическую кисть, используемую для заливки линии пера.

Табл. 16. Типы перьев

Тип пера	*Значение*	*Описание*
PS_COSMETIC	0x0	косметическое перо
PS_GEOMETRIC	0x10000	геометрическое перо

Табл. 17 Стили линий

Стиль линии	*Значение*	*Описание*	*Вид*
PS_SOLID	0	Сплошная	———————
PS_DASH	1	Штриховая *	- - - - - - - - - -
PS_DOT	2	Пунктирная *	.....
PS_DASHDOT	3	Штрих-пунктирная *	- . - . - . - . - .
PS_DASHDOTDOT	4	Штрих-два пунктира *	- . . - . . - . .
PS_NULL	5	Нулевая (невидимая)	нет линии
PS_INSIDEFRAME	6	Ширина внутри объекта	
PS_USERSTYLE	7	Пользовательская	произвольный
PS_ALTERNATE	8	Через точку (*Windows XP*) *	не для печати

* Ширина косметического пера 1 пиксель для Windows 98.

Табл. 18. Типы концевых точек геометрических перьев

Концевая точка	*Значение*	*Описание*
PS_ENDCAP_ROUND	0x0	круглая (рис. 51, а)
PS_ENDCAP_SQUARE	0x100	прямоугольная (рис. 51, б, в)
PS_ENDCAP_FLAT	0x200	плоская (рис. 51, г, д)

Табл. 19. Типы соединений геометрических перьев

Соединение	*Значение*	*Описание*
PS_JOIN_ROUND	0x0	круглое (рис. 51, а)
PS_JOIN_BEVEL	0x1000	скошенное (рис. 51, б, в)
PS_JOIN_MITER	0x2000	острое (рис. 51, г, д)

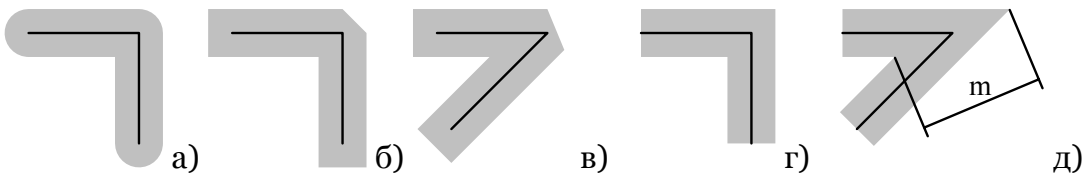


Рис. 51. Типы соединений геометрических перьев

Заострение соединения **PS_JOIN_MITER** ограничивается при помощи функции

```
BOOL SetMiterLimit(контекст, ограничение, экс_ограничение);
```

Здесь параметр ***ограничение*** — новое предельное значение заострения, параметр ***экс_ограничение*** — переменная типа ***float*** для возвращаемого старого значения. Ограничение задается как отношение длины заострения ***m*** к длине линии. По умолчанию предел равен 10,0.

Пример создания и использования геометрического пера показывает, как для этого документа был получен рис 51, б) в виде метафайла (листинг 2).

Листинг 2. Пример использования перьев

```
void Metafile(HWND hWnd, LPCSTR path, LPCSTR desc) {
   int W = 2000, H = 2000, C = 192;
   RECT rc = { 0, 0, W, H };
   hdc = CreateEnhMetaFile(NULL, "C:\\PS_JOIN_BEVEL.emf", &rc, NULL);
   SetMapMode(hdc, MM_LOMETRIC);
   LOGBRUSH lb = { BS_SOLID, RGB(C, C, C), 0 };
   HPEN hp = ExtCreatePen(PS_GEOMETRIC | PS_ENDCAP_SQUARE |
               PS_JOIN_BEVEL, 60, &lb, 0, 0);
   SelectObject(hdc, hp);
   POINT pt[] = { { 31, -31 }, { 169, -31 }, { 169, -169 } };
   Polyline(hdc, pt, 3);
   DeleteObject(hp);
   hp = CreatePen(PS_SOLID, 3, 0);
   SelectObject(hdc, hp);
   Polyline(hdc, pt, 3);
   DeleteObject(hp);
   DeleteEnhMetaFile(CloseEnhMetaFile(hdc));
}
```

Дополнительно см. «*Выбор и изменение графических объектов*», стр. 81.

Кисти

Кисти используются для закрашивания фигур и являются синонимами стиля (способа) заливки. ***Логическая кисть*** описывает идеальный *растр*, который используется для закрашивания. ***Физическая кисть*** – фактически используемая при рисовании на определенном устройстве. При создании кисти приложение получает дескриптор логической кисти. При выборе кисти в контекст устройства драйвер устройства создает физическую кисть, наиболее точно соответствующую логической.

Существует 4 типа логических кистей:

Стиль	***Описание***
Solid Brush	Произвольная одноцветная кисть с растром 8×8
Stock Brush	Стандартная кисть (7 одноцветных типов)
Hatch Brush	Кисть со стандартной штриховкой (6 типов)
Pattern Brush	Узор, задаваемый пользователем

Если кисть имеет узор, то результат заливки зависит от расположения фигуры в контексте устройства, потому что штриховка битового растра отсчитывается от начала (левой верхней точки) контекста. На рис 52 показано устройство, залитое кистью, растр которой обведен квадратом в левой верхней части. Два одинаковых прямоугольника закрашиваются этой кистью и имеют разный вид, так как расположены по-разному по отношению к начальной точке растра.

Чтобы уточнить положение растра кисти, используется функция

```
BOOL SetBrushOrgEx(контекст, x, y, экс_начало);
```

Параметры ***x*** и ***y*** задают новое положение растра кисти по отношению к левому верхнему углу контекста. Параметр ***экс_начало*** типа `LPPOINT` (стр. 115) принимает предыдущее положение начальной точки кисти.

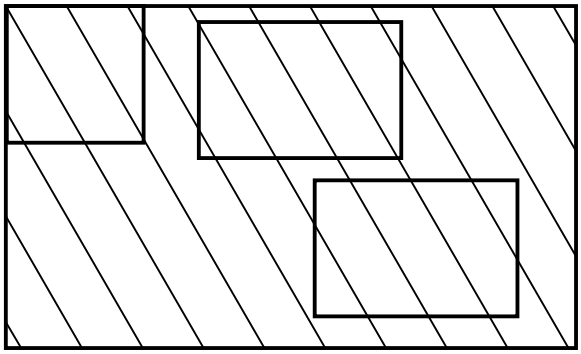


Рис. 52. Наложение штриховки на объект

Одноцветная кисть создается при помощи функции

```
HBRUSH CreateSolidBrush(цвет);
```

Пример:

```
HBRUSH hb = CreateSolidBrush(RGB(32, 64, 128));
```

Кисть системного цвета можно получить при помощи функции

```
HBRUSH GetSysColorBrush(цвет);
```

Параметр ***цвет*** выбирается из таблицы «Системные цвета Windows», стр. 115. Системную кисть *нельзя удалять* при помощи `DeleteObject`. Пример:

```
HBRUSH hb = GetSysColorBrush(COLOR_BTNFACE);
```

Стандартную кисть можно получить при помощи функции

```
HGDIOBJ GetStockObject(объект);
```

Параметр ***объект*** выбирается из табл. 21. Пример:

```
HBRUSH hb = (HBRUSH)GetStockObject(GRAY_BRUSH);
```

Табл. 20. Стандартные (*stock*) кисти Windows

Объект	***Значение***	***Описание***
WHITE_BRUSH	0	Белая кисть
LTGRAY_BRUSH	1	Светло-серая кисть
GRAY_BRUSH	2	Серая кисть
DKGRAY_BRUSH	3	Темно-серая кисть
BLACK_BRUSH	4	Черная кисть
HOLLOW_BRUSH=NULL_BRUSH	5	Пустая или нулевая кисть

Штриховая кисть создается функцией

```
HBRUSH CreateHatchBrush(штриховка, цвет_штриховки);
```

Параметр ***штриховка*** выбирается из табл. 22 below. Пример:

```
HBRUSH hb = CreateHatchBrush(HS_FDIAGONAL, RGB(32, 64, 128));
```

Кисть с узором создает функция

```
HBRUSH CreatePatternBrush(картинка);
```

Параметр ***картинка*** — это дескриптор картинки (пиксельного набора).

В примере создается кисть с узором на основе черно-белой картинки 8×8:

```
WORD bits[8] = {0xFF, 0xFD, 0xFB, 0x87, 0xB7, 0xB7, 0x87, 0xFF};
HBITMAP hbmp = CreateBitmap(8, 8, 1, 1, (VOID*)bits);
HBRUSH hb = CreatePatternBrush(hbmp);
```

Табл. 21. Стандартные штриховки

Штриховка	*Значение*	*Описание*
HS_HORIZONTAL	0	горизонталь
HS_VERTICAL	1	вертикаль
HS_FDIAGONAL	2	диагональ слева направо вниз (*forward diagonal*)
HS_BDIAGONAL	3	диагональ слева направо вверх (*backward diagonal*)
HS_CROSS	4	клетка
HS_DIAGCROSS	5	диагональная клетка

Вид стандартных штриховок приведен ниже.

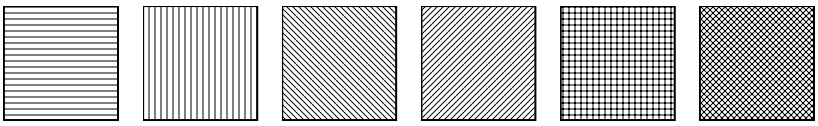


Рис. 53. Стандартные штриховки

Если картинка двухцветная, например, черно-белая, цвет ***0*** картинки заменяется на цвет текста, а цвет ***1*** — на цвет фона в данном контексте. О создании картинок см. раздел «*Пиксельные наборы*», стр. 79.

Неявно кисть создается при помощи функции

```
HBRUSH CreateBrushIndirect(логическая_кисть);
```

Параметр ***логическая_кисть*** — указатель на структуру `LOGBRUSH` (стр. 116).

Пример:

```
LOGBRUSH lb = { BS_HATCHED, RGB(32, 64, 128), HS_FDIAGONAL };
HBRUSH hb = CreateBrushIndirect(&lb);
```

Дополнительно см. «*Выбор и изменение графических объектов*», стр. 81.

Палитры

Палитра — это набор красок для рисования на конкретном устройстве. Она представляет собой массив значений RGB-составляющих цвета. Палитры используются в случае, если устройство позволяет генерировать произвольные цвета, но в конкретный момент времени предоставляет только некоторое подмножество. Для таких устройств создается ***системная палитра***, которая предлагает текущий набор цветов. Приложения не имеют прямого доступа к системной палитре. Вместо этого в контексте устройства создается ее копия — ***палитра по умолчанию*** (*default palette*). Эта палитра содержит цвета, доступные на устройстве сразу после получения контекста. Приложение использует палитру по умолчанию или создает собственную, уникальную ***логическую палитру***, которая выбирается в контекст устройства и реализуется.

Устройство поддерживает палитру, если вызов функции

```
int GetDeviceCaps(контекст, RASTERCAPS);
```

возвращает значение, в котором установлен бит `RC_PALETTE=0x100`.

Типичная палитра по умолчанию содержит 20 цветов, однако точное число зависит от конкретного устройства. Определить его можно при помощи вызова

```
int GetDeviceCaps(контекст, NUMCOLORS);
```

Если количество цветов превышает *256*, возвращается значение *–1*, которое означает, что цветов слишком много для перечисления.

Все цвета палитры устройства можно получить при помощи функции перечисления сплошных перьев, приведенной в приложении на стр. 122. Получить цвет или несколько цветов системной палитры можно при помощи функции

```
UINT GetSystemPaletteEntries(контекст, начало, количество, массив);
```

Параметр ***начало*** — индекс первого элемента, ***количество*** — число элементов, ***массив*** — указатель на массив элементов палитры (типа **PALETTEENTRY**, стр. 119).

Для рисования, закраски и вывода текста приложение использует цвета. Цвет обычно задается при помощи макроса RGB, например:

```
HPEN hp = CreatePen(PS_SOLID, 1, RGB(255, 32, 112));
```

Если запрашиваемый для пера или текста цвет в палитре отсутствует, выбирается ближайший в палитре (цвет аппроксимируется). Если приложение запрашивает цвет для сплошной кисти, при необходимости выполняется дизеринг.

Чтобы предотвратить аппроксимацию и дизеринг, приложение может указывать индекс цвета вместо значения. *Индекс цвета* — это номер, под которым цвет присутствует в палитре. Использовать следует не собственно индекс, а число, которое возвращает макрос **PALETTEINDEX**:

```
COLORREF PALETTEINDEX(индекс_цвета_в_палитре);
```

Однако при этом создается *зависимость от устройства*, так как используются только цвета устройства. Чтобы избежать зависимости от устройства, следует использовать ***родственные палитре цвета*** (*palette-relative*). Они создаются при помощи макроса **PALETTERGB**:

```
COLORREF PALETTERGB(красный, зеленый, синий);
```

При использовании родственных палитре цветов вместо дизеринга выполняется аппроксимация. Например, для того же пера цвет можно задать так:

```
HPEN hp = CreatePen(PS_SOLID, 1, PALETTERGB(255, 32, 112));
```

Особенность палитр проявляется при одновременном использовании устройства несколькими приложениями. Если перья и кисти для разных приложений могут быть разными, то цвета устройства — нет. Например, если одно приложение изменяет палитру экрана, окна других приложений также изменяют свой цвет. Это может привести к «мешанине» на экране. Поэтому во время реализации логической палитры предпочтение отдается *только одной*, и она может принадлежать только активному окну. Говорят, что эта палитра находится в режиме *Foreground*. Все остальные логические палитры находятся в режиме *Background* или переводятся в него автоматически независимо от установленного для них режима.

Разница между режимами заключается в порядке предоставления цветов. Палитре в режиме *Foreground* отдаются все свободные цвета в системной палитре. Свободные цвета те, которые не помечены как статические (неизменные). Палитра в режиме *Background* получает оставшиеся цвета в порядке очередности.

Приложение может уменьшить количество статических цветов до двух (белый и черный), если оно находится в режиме полного экрана и обладает фокусом. Для этого приложение вызывает функцию

```
UINT SetSystemPaletteUse(контекст, режим);
```

Параметр ***режим*** выбирается из табл. 23.

Табл. 22. Режимы использования системной палитры

Режим	*Значение*	*Описание*
SYSPAL_STATIC	1	Восстановить статические цвета
SYSPAL_NOSTATIC	2	Освободить статические цвета

Если приложение удаляет таким образом статические цвета, оно должно немедленно реализовать свою палитру, сохранить системные цвета, установить новые системные цвета с использованием только белого и черного цветов, и послать сообщение **WM_SYSCOLORCHANGE** об изменении системных цветов.

Если такое приложение теряет фокус или закрывается, оно должно вернуть статические цвета, реализовать логическую палитру, восстановить системные цвета, и послать сообщение об их изменении.

Для создания уникального набора цветов приложение конструирует логическую палитру при помощи функции

HPALETTE CreatePalette(*палитра*);

Параметр ***палитра*** является указателем на структуру **LOGPALETTE** (стр. 119). Элементы палитры следует располагать в порядке предпочтения — цвета, расположенные в начале, имеют больше шансов попасть в системную палитру. Количество цветов в логической палитре не должно превышать размер палитры устройства. Это число можно получить при помощи функции **GetDeviceCaps** с параметром **SIZEPALETTE**.

После создания логической палитры она выбирается в контекст функцией

HPALETTE SelectPalette(*контекст, палитра, режим*);

Параметр ***палитра*** — идентификатор палитры, параметр ***режим*** принимает значение **TRUE** для режима *Background*, и **FALSE** для *Foreground*.

После выбора палитры в контекст ее нужно ***реализовать*** функцией

UINT RealizePalette(*контекст*);

Во время реализации происходит замена цветов системной палитры цветами логической палитры в соответствии с ее режимом.

После создания палитры ее элементы можно изменить при помощи функции

UINT SetPaletteEntries(*палитра, начало, количество, массив*);

Параметр ***палитра*** — дескриптор палитры, другие параметры соответствуют вызову функции **GetSystemPaletteEntries** (стр. 63).

Как только палитра станет ненужной, ее необходимо удалить при помощи функции **DeleteObject**.

Пиксельные наборы

Пиксельный набор (*bitmap*, в дальнейшем просто *картинка*) — это массив пикселей, образующих прямоугольную растровую картинку. Используется для создания, отображения и сохранения изображений на диск.

Картинка содержит ***битовый массив, описывающий пиксели в прямоугольной области***. Битовый массив состоит из выровненных по границе слова или двойного слова строк, соответствующих растровым линиям устройства. В строке записываются последовательности бит, указывающие цвета пикселей. Для записи цвета одного пикселя используется *1*, *4*, *8*, *15*, *16*, *24* или *32* бита.

Если для записи цвета используется *1* бит, картинка черно-белая, бит *0* обозначает черный цвет (цвет с номером *0*), бит *1* — белый (цвет с номером *1*).

В качестве примера рассмотрим черно-белую картинку размером *8×8* (рис. 55). Для каждой линии растра справа приведено значение строки в битах, и соответствующее шестнадцатеричное число. Учитывая, что каждая линия растра должна иметь количество бит, кратное 16-ти (выравнивание по границе слова), битовый массив описывается числами:

0x00FF, 0x00FD, 0x00FB, 0x0087, 0x00B7, 0x00B7, 0x0087, 0x00FF.

Каждое число здесь представляет одну растровую строку и имеет размер **WORD**.

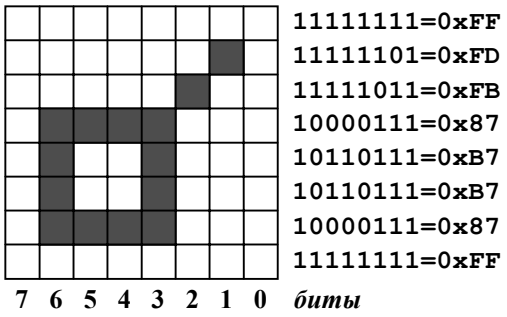


Рис. 54. Растр черно-белой картинки размером 8×8

Если картинка имеет глубину цвета *4*, для записи одного пикселя используется *4* бита, а для записи 8 пикселей в строке — *32* бита (**DWORD**). Четыре бита определяют номер, под которыми цвет пикселя записан в таблице (палитре), которая должна прилагаться к битовому массиву. Если принять, что черные пиксели картинки на рис. 55 — красные, то битовый массив будет иметь вид:

0xFFFFFFFF, 0xFFFFFF1F, 0xFFFFF1FF, 0xF1111FFF,
0xF1FF1FFF, 0xF1FF1FFF, 0xF1111FFF, 0xFFFFFFFF

Каждая шестнадцатеричная цифра здесь представляет один пиксель. Цифра **F** — номер белого цвета, а цифра **1** — номер красного цвета в *16*-цветной палитре.

Аналогичным образом формируется битовый массив для глубины цвета 8. На каждый пиксель в нем отводится один байт, а для описания самих цветов используется палитра из 256 элементов. Размер строки будет равен 64 битам.

Для описанных форматов требуется палитра, которая прилагается к битовому массиву. При глубине цвета 16, 24 и 32 палитра теряет смысл, так как индекс цвета по размеру сопоставим с самим цветом. Так, при глубине цвета 24 палитра должна описывать 16,8 млн. цветов размером 4 байта каждый. Индекс цвета в битовом массиве также имеет размер 4 байта. Поэтому при большой глубине цвета битовый массив содержит не индексы, а сами цвета в формате RGB.

В зависимости от способа задания цветов различают картинки, зависимые от устройства (*ddb — device-dependent bitmap*) и независимые от устройства (*dib — device-independent bitmap*).

Если используются цвета или индексы цветов конкретного устройства, получается устройство-зависимая, *ddb*-картинка. Ее достоинство — она выводится на данное устройство без потери качества, так как состоит только из имеющихся на устройстве цветов. Недостаток — на другое устройство картинка не выводится, или, в лучшем случае, цвета картинки могут быть сильно искажены.

Если цвета картинки описываются произвольными RGB-составляющими, получается независимая от устройства, или *dib*-картинка. Ее достоинство — она описывает изображение наиболее точным способом. При выводе на произвольное устройство цвета ее палитры аппроксимируются к цветам устройства так, чтобы обеспечить наилучшее качество.

Черно-белую *ddb-картинку* следует создавать при помощи функции

```
HBITMAP CreateBitmap(ширина, высота, 1, глубина_цвета, (LPVOID)массив);
```

Параметр ***массив*** представляет собой массив байт или слов, сформированных в растровые строки. Каждая строка должна быть выровнена по границе слова.

В следующем примере создается черно-белая картинка, показанная на рис. 55:

```
WORD bits[] = {0xFF, 0xFD, 0xFB, 0x87, 0xB7, 0xB7, 0x87, 0xFF};
HBITMAP hbmp = CreateBitmap(8, 8, 1, 1, (LPVOID)bits);
```

При помощи этой функции можно создать и цветную картинку, *но только совместимую с контекстом*, в котором она будет использоваться. Совместимость означает *одинаковый формат задания цвета*. Так, если картинка предназначена для экрана, нужно знать количество битовых плоскостей и глубину цвета экрана. Их можно определить при помощи вызовов:

```
int ps = GetDeviceCaps(hdc, PLANES);
int bp = GetDeviceCaps(hdc, BITSPIXEL);
```

Если, например, `ps=1` и `bp=32`, то картинку размером *2×2* с белыми и красными точками (рис. 56) можно создать следующим образом:

```
DWORD bits[] = {0x00FFFFFF, 0x00FF0000, 0x00FF0000, 0x00FFFFFF};
HBITMAP hbmp = CreateBitmap(2, 2, 1, 32, (LPVOID)bits);
```

```
0x00FFFFFF,0x00FF0000
0x00FF0000,0x00FFFFFF
```

Рис. 55. Растр цветной картинки 2×2

Цветную *ddb-картинку* следует создавать при помощи функции

```
HBITMAP CreateCompatibleBitmap(контекст, ширина, высота);
```

Контекст устройства определяет здесь возможные цвета картинки, поэтому она называется совместимой (*compatible*). Если ширина или высота равны *0*, картинка имеет размер *1×1*. В примере создается картинка размером *16×16*:

```
HBITMAP hbmp = CreateCompatibleBitmap(hdc, 16, 16);
```

В отличие от предыдущего примера, изображение этой картинки не может быть задано в виде массива бит. Вместо этого она выбирается в контекст памяти и изображение на ней рисуется функциями рисования графических примитивов (см. ниже раздел «*Контекст памяти*»).

Устройство-зависимая картинка описывается структурой `BITMAP` (стр. 119) и может быть создана также косвенно при помощи функции

```
HBITMAP CreateBitmapIndirect(картинка);
```

Здесь параметр ***картинка*** — указатель на структуру `BITMAP`.

Контекст памяти

После того, как *ddb*-картинка создана, ее следует выбрать в контекст памяти. ***Контекст памяти*** — это копия устройства в памяти. Он обладает всеми атрибутами копируемого устройства, но не выводит изображение. Контекст памяти называется ***совместимым***, так как он использует цветовую схему устройства, и создается при помощи функции

```
HDC CreateCompatibleDC(контекст);
```

Параметр ***контекст*** указывает на устройство, для которого создается контекст в памяти, например, контекст принтера или контекст окна.

После того, как картинка выбрана в контекст, ее можно ***скопировать*** в контекст любого устройства при помощи одной из основных функций — *Bit Block Transfer*:

BOOL BitBlt(*приемник, x, y, w, h, источник, x0, y0, ROP3*);

Здесь ***приемник*** — контекст приемника, ***источник*** — контекст источника. ***x0, y0*** — координаты начальной точки в источнике, ***x, y*** — координаты начальной точки в приемнике, ***w, h*** — ширина и высота в приемнике.

Параметр ***ROP3*** уточняет, как пиксели картинки будут сочетаться с пикселями устройства (табл. 10, стр. 54). Простейший вариант — просто скопировать пиксели (**SRCCOPY**). Дополнительно см. раздел «*Прозрачность*», стр. 75.

В качестве примера выведем на экран черно-белую картинку рис. 55:

Листинг 3. Пример копирования картинки на экран

```
WORD bits[] = {0xFF, 0xFD, 0xFB, 0x87, 0xB7, 0xB7, 0x87, 0xFF};
HBITMAP hbmp = CreateBitmap(8, 8, 1, 1, (LPVOID)bits);
HDC cdc = CreateCompatibleDC(hdc);
SelectObject(cdc, hbmp);
BitBlt(hdc, 0, 0, 8, 8, cdc, 0, 0, SRCCOPY);
DeleteDC(cdc);
DeleteObject(hbmp);
```

Скопировать картинку с поворотом можно при помощи функции

BOOL PlgBlt(*приемник, массив, источник, x0, y0, w0, h0, маска, xm, ym*);

Здесь также ***приемник*** — контекст приемника, ***источник*** — контекст источника. ***x0, y0*** — координаты начальной точки в источнике, ***w0, h0*** — ширина и высота в источнике. Параметр ***массив*** задает параллелограмм в приемнике, в который копируется картинка. Он является указателем на массив из трех вершин. Верхний левый угол прямоугольника источника становится вершиной A, верхний правый угол — вершиной B, нижний левый угол становится вершиной C. Четвертая вершина определяется системой из соотношения D = B + C – A.

Параметр ***маска*** — дескриптор черно-белой картинки, ***xm, ym*** — координаты начальной точки маски. Маскирующая картинка, если задана, используется для маскирования цветов картинки. Значение *1* в маске определяет, что цвет источника копируется в приемник, а значение *0* определяет, что цвет не изменяется.

Листинг 4. Пример копирования картинки с поворотом

```
HBITMAP hbmp = CreateCompatibleBitmap(hdc, 16, 16);
HDC cdc = CreateCompatibleDC(hdc);
SelectObject(cdc, hbmp);
hp = CreatePen(PS_SOLID | PS_INSIDEFRAME, 4, RGB(192, 0, 0));
SelectObject(cdc, hp);
Rectangle(cdc, 0, 0, 16, 16);
POINT pt[] = { {19, 21}, {30, 10}, {30, 32}};
PlgBlt(hdc, pt, cdc, 0, 0, 16, 16, NULL, 0, 0);
DeleteObject(hp);
DeleteDC(cdc);
DeleteObject(hbmp);
```

В примере листинг 4 создается контекст памяти для экрана, в него выбирается цветная картинка 16×16, на которой рисуется красный квадрат. Затем картинка копируется с поворотом. При рисовании картинки в памяти следует помнить, что поверхность картинки изначально черная, ее нужно закрашивать.

Для копирования и ***масштабирования*** картинки используется функция

BOOL StretchBlt(*приемник, x1, y1, w1, h1, источник, x0, y0, w0, h0, ROP3*) ;

Здесь параметры ***x0, y0, w0, h0*** описывают начальную точку и размеры исходной картинки, а параметры ***x1, y1, w1, h1*** — результирующей. Параметр ***ROP3*** имеет тот же смысл, что и для функции **BitBlt**.

Работа с картинками в памяти обычно значительно быстрее. Этот прием часто используется для создания анимации и носит название *двойной буферизации*.

Устройство-независимые картинки

Устройство-независимые картинки состоят из двух частей: *информационной*, описывающей характеристики изображения, и собственно *битового массива*, описывающего пиксели.

Информационная часть представляет собой структуру **BITMAPINFO**:

```
typedef struct tagBITMAPINFO {
   BITMAPINFOHEADER bmiHeader;
   RGBQUAD          bmiColors[1];
} BITMAPINFO, *PBITMAPINFO;
```

Она состоит из информационного заголовка и палитры из минимум двух элементов. Палитра отсутствует, если глубина цвета равна 24.

Информационный заголовок содержит размеры изображения, характеристики цветовой схемы (число битовых плоскостей и глубину цвета), признак алгоритма сжатия, разрешение устройства, для которого картинка была создана и др.

Существует ***4 типа информационных заголовков***.

Заголовок **BITMAPCOREHEADER** описывает *dib*-картинку системы OS/2:

```
typedef struct tagBITMAPCOREHEADER {
   DWORD  bcSize;          /* размер структуры (0x0C = 12) */
   WORD   bcWidth;         /* ширина картинки              */
   WORD   bcHeight;        /* высота                       */
   WORD   bcPlanes;        /* битовые плоскости            */
   WORD   bcBitCount;      /* глубина цвета, бит/пиксель   */
} BITMAPCOREHEADER, *PBITMAPCOREHEADER;
```

Глубина цвета составляет 1, 4, 8 или 24. Для описания картинки OS/2 используется структура **BITMAPCOREINFO**:

```
typedef struct _BITMAPCOREINFO {
   BITMAPCOREHEADER  bmciHeader;
   RGBTRIPLE         bmciColors[1];
} BITMAPCOREINFO, *PBITMAPCOREINFO
```

Палитра здесь описывается триплетами **RGBTRIPLE** (стр. 118).

Заголовок **BITMAPINFOHEADER** является расширением заголовка **BITMAPCOREINFO**. Его подробное описание приведено в приложении «*Формат BMP*», стр. 120.

Информационный заголовок **BITMAPV4HEADER** является расширением заголовка **BITMAPINFOHEADER**, а заголовок **BITMAPV5HEADER** является последующим расширением заголовка **BITMAPV4HEADER**. Их важное отличие заключается в *возможности задавать координаты основных цветов* RGB в системе XYZ. Здесь эти заголовки не рассматриваются, подробнее см. *MSDN Library*.

Сжатие производится для картинок с глубиной цвета 4 и 8, а также для картинок в формате JPEG и PNG, которые поддерживаются в некоторых системах для возможности прямой печати изображений. Алгоритмы сжатия BMP-картинок описаны в приложении, стр. 121. *Расшифровка сжатых изображений*

встроена в драйвер дисплея. Иначе говоря, не требуется предпринимать никаких дополнительных усилий для расшифровки. Алгоритм сжатия используется приложениями для записи картинки в файл с целью уменьшения его размера.

Если битовый массив сжат, он располагается непосредственно за информационной частью, а картинка называется ***упакованным dib-форматом***. В этом случае используется *единственный указатель на картинку*. Поле **biClrUsed** в структуре **BITMAPINFOHEADER** должно быть равно нулю либо содержать фактический размер палитры. В любом случае это число должно быть *четным* для того, чтобы битовый массив начинался с границы двойного слова.

Строки битового массива могут располагаться в прямом и обратном порядке. Если параметр *Height* в информационной структуре положителен, картинка записывается задом наперед (первая строка последней), а если отрицателен, то нормальным образом. При этом картинки второго типа *не могут быть сжаты*. Строки битового массива должны быть выровнены по границе двойного слова, а не слова, как в *ddb*-картинках. Это не касается сжатых изображений.

Устройство-независимые картинки изначально предназначены для описания формата хранения, а не обработки. В большинстве случаев создаются и используются *ddb*-картинки. Они обрабатываются быстрее.

Тем не менее, приложения используют *dib*-картинки в следующих двух контекстах: картинку, считанную из файла, нужно отобразить на конкретном устройстве, и наоборот, картинку, нарисованную на конкретном устройстве, нужно сохранить в файл. Кроме этого, *dib*-картинки используются в качестве промежуточного формата при преобразовании цветовой схемы *ddb*-картинок, а также при создании кистей.

Преобразование *ddb→dib* осуществляется функция **GetDIBits**. Она извлекает информацию из *совместимой* картинки и формирует параметры и битовый массив *dib*-картинки:

```
int GetDIBits(
   HDC hdc,            /* контекст                              */
   HBITMAP hbmp,       /* дескриптор ddb-картинки               */
   UINT uStartScan,    /* начальная строка развертки            */
   UINT cScanLines,    /* количество строк развертки            */
   LPVOID lpvBits,     /* буфер для битового массива            */
   LPBITMAPINFO lpbi, /* буфер для информации о dib-картинке */
   UINT uUsage         /* режим использования палитры           */
);
```

Здесь параметр ***контекст*** — контекст устройства, ***картинка*** — дескриптор совместимой с контекстом картинки. Она должна быть выбрана в контекст перед началом преобразования. ***Начальная строка развертки*** и ***количество строк*** задают область преобразования и используются при выводе картинки по частям с целью ускорения обработки. Параметр ***режим*** определяет, прилагается к *dib*-картинке палитра, или нет, и если прилагается, то что она содержит. Может принимать два значения (табл. 24).

Табл. 23. Режимы использования палитры

Режим	*Значение*	*Описание*
DIB_RGB_COLORS	0	Палитра содержит RGB-составляющие.
DIB_PAL_COLORS	1	Палитра содержит 16-ти битовые индексы логической палитры устройства.

Буфер для битового массива принимает картинку в формате, заданном буфером для информации о *dib*-картинке, который должен быть заполнен перед вызовом функции (анализируются первые 6 полей). ***Буфер для информации*** заполняется необходимыми значениями, если буфер для битового массива равен **NULL**. В любом случае поле **biSize** должно указывать размер структуры.

Палитра dib-картинки формируется в соответствии с требуемой цветовой схемой и режимом использования. В режиме **DIB_RGB_COLORS** элементы палитры имеют размер *32* бита, а в режиме **DIB_PAL_COLORS** — *16* бит. Для размещения палитры память выделяется непосредственно за структурой **lpbi**, учитывая, что эта структура уже содержит один или два элемента палитры (*32* бита).

Для глубины цвета *1*, *4* и *8* генерируется палитра из соответственно *2*, *16* или *256* элементов. Для глубины цвета *16* и *32* генерируется палитра из трех двойных слов, содержащих маски для выделения цветовых составляющих, и дополнительно требуется память для двух двойных слов. Для глубины цвета *24* палитра не нужна, но элемент палитры все равно присутствует.

Если формируется *упакованная dib*-картинка, то память под битовый массив должна быть выделена непосредственно за палитрой. Количество требуемой памяти указывает поле **biSizeImage** с учетом выравнивания строк развертки.

Преобразование *dib→ddb* осуществляет функция **SetDIBits**. Она формирует битовый массив *совместимой* картинки, которая должна быть создана и выбрана в контекст устройства перед преобразованием.

```
int SetDIBits(
   HDC hdc,                  /* контекст                    */
   HBITMAP hbmp,             /* дескриптор ddb-картинки     */
   UINT uStartScan,          /* начальная строка развертки  */
   UINT cScanLines,          /* количество строк развертки  */
   CONST VOID *lpvBits,      /* битовый массив              */
   CONST BITMAPINFO *lpbmi,  /* информация о dib-картинке   */
   UINT fuColorUse           /* режим использования палитры */
);
```

Здесь параметры имеют тот же смысл, что и для функции **GetDIBits**. Для ускорения обработки палитра должна содержать индексы логической палитры устройства. Получить эти индексы должно приложение при помощи функции **GetSystemPaletteEntries** (стр. 63).

Создать *совместимую* картинку из *dib*-картинки можно при помощи функции

```
HBITMAP CreateDIBitmap(
   HDC hdc,                        /* контекст                    */
   CONST BITMAPINFOHEADER *lpbmih, /* информация о ddb-картинке   */
   DWORD fdwInit,                  /* режим инициализации         */
   CONST VOID *lpbInit,            /* инициализирующий массив     */
   CONST BITMAPINFO *lpbmi,        /* информация о dib-картинке   */
   UINT fuUsage                    /* режим использования палитры */
);
```

Параметр ***контекст*** — это контекст устройства, для которого создается *ddb*-картинка. Размер картинки определяется параметром ***информация о ddb-картинке***. Если параметр ***режим инициализации*** равен **CBM_INIT=4**, картинка инициализируется битовым массивом **lpbInit**, иначе картинка не инициализируется (остается черной). Если картинка инициализируется, информация о битовом масси-

ве извлекается из параметра **lpbmi**. Использование функции в этом случае эквивалентно вызову функций **CreateCompatibleBitmap** и **SetDIBits**.

Для вывода *dib*-картинки на устройство используется функция

```
int SetDIBitsToDevice(
   HDC hdc,                /* контекст устройства                */
   int XDest,              /* x-координата нач. точки приемника */
   int YDest,              /* y-координата нач. точки приемника */
   DWORD dwWidth,          /* ширина картинки                    */
   DWORD dwHeight,         /* высота картинки                    */
   int XSrc,               /* x-координата нач. точки источника */
   int YSrc,               /* y-координата нач. точки источника */
   UINT uStartScan,        /* начальная строка развертки         */
   UINT cScanLines,        /* количество строк развертки         */
   CONST VOID *lpvBits,    /* битовый массив                     */
   CONST BITMAPINFO *lpbmi, /* информация о dib-картинке         */
   UINT fuColorUse         /* режим использования палитры        */
);
```

Большинство параметров этой функции имеют тот же смысл, что и для функции **GetDIBits**. Дополнительные параметры задают положение картинки на устройстве вывода и в источнике. Если картинка имеет большую площадь, вывод осуществляется частями, по несколько строк развертки за один раз.

Следующая функция выводит *dib*-картинку на устройство с одновременным изменением ее размера (сжатием или растяжением):

```
int StretchDIBits(
   HDC hdc,                  /* контекст устройства                */
   int XDest,                /* x-координата нач. точки приемника */
   int YDest,                /* y-координата нач. точки приемника */
   int nDestWidth,           /* ширина приемника картинки          */
   int nDestHeight,          /* высота приемника картинки          */
   int XSrc,                 /* x-координата нач. точки источника */
   int YSrc,                 /* y-координата нач. точки источника */
   int nSrcWidth,            /* ширина источника картинки          */
   int nSrcHeight,           /* высота источника картинки          */
   CONST VOID *lpBits,         /* битовый массив                   */
   CONST BITMAPINFO *lpBitsInfo, /* информация о dib-картинке      */
   UINT iUsage,                /* режим использования палитры    */
   DWORD dwRop                 /* код операции ROP3              */
);
```

Описанные две функции поддерживаются не всеми устройствами. Чтобы определить поддержку устройством, используется вызов

```
int GetDeviceCaps(контекст, RASTERCAPS);
```

Если возвращаемое значение содержит установленный бит **RC_DIBTODEV=0x200**, устройство поддерживает функцию **SetDIBitsToDevice**. Если возвращаемое значение содержит установленный бит **RC_STRETCHDIB=0x2000**, устройство поддерживает функцию **StretchDIBits**.

Как было сказано ранее, совместимая картинка не может быть отображена на устройстве с другой цветовой схемой. Для решения этой проблемы используется следующий механизм: *ddb*-картинка одной цветовой схемы преобразуется в *dib*-картинку, а затем снова в *ddb*-картинку, но с другой цветовой схемой. Первое преобразование выполняет функция **GetDIBits**, а второе — одна из функций **SetDIBits**, **SetDIBitsToDevice** или **StretchDIBits**.

В примере листинг 5 создается совместимая картинка размером *16×16*. Далее она преобразуется в *dib*-картинку с глубиной цвета *4*, записывается в файл BMP, преобразуется в *ddb*-картинку и выводится на экран.

Листинг 5. Пример работы с картинками

```
const int cx = 16, cy = 16, cquad = sizeof(RGBQUAD),
   cbfh = sizeof(BITMAPFILEHEADER),
   cbmi = sizeof(BITMAPINFO) + cquad * 15;
HDC hdc = GetDC(hWnd);
HBITMAP hbmp = CreateCompatibleBitmap(hdc, cx, cy);
HDC cdc = CreateCompatibleDC(hdc);
SelectObject(cdc, hbmp);
HPEN hp = CreatePen(PS_INSIDEFRAME, 4, RGB(255, 0, 0));
SelectObject(cdc, hp);
Rectangle(cdc, 0, 0, cx, cy);
BITMAPINFO *bmi;
bmi = (BITMAPINFO*)LocalAlloc(LPTR, cbmi);
bmi->bmiHeader.biSize = 40;
bmi->bmiHeader.biWidth = cx;
bmi->bmiHeader.biHeight = cy;
bmi->bmiHeader.biBitCount = 4;
bmi->bmiHeader.biPlanes = 1;
bmi->bmiHeader.biCompression = BI_RGB;
int bx = (cx + 1) >> 1;
int rx = bx % 8;
int cbs = (bx + (rx ? 8 - rx : 0)) * cy;
BYTE *dibs = (BYTE*)LocalAlloc(LPTR, cbs);
GetDIBits(cdc, hbmp, 0, cy, dibs, bmi, DIB_RGB_COLORS);
int offs = cbfh + cbmi;
BITMAPFILEHEADER bfh = { 'MB', offs + cbs, 0, 0, offs };
HANDLE bf = CreateFile("C:\\test.bmp", GENERIC_WRITE, 0, 0,
   CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
ULONG rd;
WriteFile(bf, &bfh, cbfh, &rd, 0); /* файловый заголовок   */
WriteFile(bf, bmi, cbmi, &rd, 0);  /* информация и палитра */
WriteFile(bf, dibs, cbs, &rd, 0);  /* битовый массив       */
CloseHandle(bf);
HBITMAP cbmp = CreateDIBitmap(cdc, &bmi->bmiHeader,
   CBM_INIT, dibs, bmi, DIB_RGB_COLORS);
SelectObject(cdc, cbmp);
BitBlt(hdc, 0, 0, cy, cx, cdc, 0, 0, SRCCOPY);
DeleteDC(cdc);
DeleteObject(hbmp);
DeleteObject(cbmp);
DeleteObject(hp);
DeleteObject(hb);
LocalFree(dibs);
LocalFree(bmi);
ReleaseDC(hWnd, hdc);
```

Чтение полученной в предыдущем примере картинки демонстрирует следующий пример (предполагается, что картинка записана с глубиной цвета *16*):

Листинг 6. Чтение картинки

```
hdc = GetDC(hWnd);
const int cquad = sizeof(RGBQUAD),
   cbfh = sizeof(BITMAPFILEHEADER),
   cbmi = sizeof(BITMAPINFO) + cquad * 15;
DWORD rd;
HANDLE bf = CreateFile("C:\\test.bmp", GENERIC_READ,
   0, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
BITMAPFILEHEADER bfh;
ReadFile(bf, &bfh, cbfh, &rd, 0); /* файловый заголовок */
BITMAPINFO *bmi;
bmi = (BITMAPINFO*)LocalAlloc(LPTR, cbmi);
ReadFile(bf, bmi, cbmi, &rd, 0);  /* информация и палитра */
int cx = bmi->bmiHeader.biWidth;
int cy = bmi->bmiHeader.biHeight;
int cbs = bmi->bmiHeader.biSizeImage;
BYTE *dibs = (BYTE*)LocalAlloc(LPTR, cbs);
ReadFile(bf, dibs, cbs, &rd, 0);  /* битовый массив */
CloseHandle(bf);
HDC cdc = CreateCompatibleDC(hdc);
HBITMAP cbmp = CreateDIBitmap(hdc, &bmi->bmiHeader,
   CBM_INIT, dibs, bmi, DIB_RGB_COLORS);
SelectObject(cdc, cbmp);
BitBlt(hdc, 0, 0, cx, cy, cdc, 0, 0, SRCCOPY);
DeleteDC(cdc);
DeleteObject(cbmp);
ReleaseDC(hWnd, hdc);
```

Функция `CreateDIBSection` создает *dib*-картинку, в которую можно записывать битовую информацию *непосредственно*:

```
HBITMAP CreateDIBSection(
   HDC hdc,                /* контекст                          */
   CONST BITMAPINFO *pbmi, /* информация о dib-картинке         */
   UINT iUsage,            /* режим использования палитры       */
   VOID **ppvBits,         /* переменная для битового массива */
   HANDLE hSection,        /* файл отображения                  */
   DWORD dwOffset          /* смещение к битовому массиву       */
);
```

Параметр ***контекст*** используется для инициализации цветов картинки из логической палитры при режиме использования `DIB_PAL_COLORS`. Параметр ***информация о dib-картинке*** определяет размер картинки и палитру. Параметр `ppvBits` — указатель на переменную, которая получит указатель на битовый массив. Он используется далее для записи битовых значений.

Параметр ***файл отображения*** — это дескриптор файла, проецируемого в память. Он создается при помощи функции `CreateFileMapping`. Параметр ***смещение к битовому массиву*** указывает расположение битового массива в файле отображения. Если файл отображения не используется, последние два параметра принимаются равными нулю, и картинка создается в памяти.

В примере листинг 7 создается картинка *16×16* с глубиной цвета 16:

Листинг 7. Создание dib-картинки в памяти

```
HDC hdc = GetDC(hWnd);
const int H = 128, C = 192, F = 255,
   cx = 16, cy = 16, cb = (cx * cy) >> 1;
BYTE pal[60] = {
   H,0,0,0,0,H,0,0,H,H,0,0,0,0,H,0,H,0,H,0,0,H,H,0,H,H,H,0,
   C,C,C,0,F,0,0,0,0,F,0,0,F,F,0,0,0,0,F,0,F,0,F,0,0,F,F,0,F,F,F,0
};
BITMAPINFO bmi = { {40,cx,-cy,1,4,0,0,0,0,0,0}, {0,0,0,0} };
BYTE *dibs = NULL;
HBITMAP cbmp = CreateDIBSection(hdc, &bmi, DIB_RGB_COLORS,
   (VOID**)&dibs, 0, 0);
for (int i = 0; i < cb; i++) dibs[i] = 0x9C;
HDC cdc = CreateCompatibleDC(hdc);
SelectObject(cdc, cbmp);
BitBlt(hdc, 0, 0, cx, cy, cdc, 0, 0, SRCCOPY);
DeleteDC(cdc);
DeleteObject(cbmp);
ReleaseDC(hWnd, hdc);
```

Параметры картинки заданы структурой `bmi`. Отрицательная высота указывает на прямой порядок расположения строк развертки. Глубина цвета равна *4*. Перед структурой `bmi` расположена палитра `pal`, состоящая из *15*-ти цветов (цвет номер *0* располагается в структуре `bmi`). На самом деле палитра расположена непосредственно *за* структурой `bmi`, так как локальные переменные располагаются в стеке. После получения указателя на битовый массив он используется для создания синих (цвет `0x9`) и красных (цвет `0xC`) вертикальных полос. Так как глубина цвета равна *4*, один байт битового массива описывает два пикселя.

В следующем примере используется файл отображения, который содержит битовый массив. Пропущенные части кода полностью соответствуют примеру, приведенному в листинге 5.

Листинг 8. Пример создания файла отображения

```
. . .
HANDLE bf = CreateFile("C:\\tmf.dat", GENERIC_READ |
   GENERIC_WRITE, 0, 0, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
HANDLE mo = CreateFileMapping(bf, NULL,
   PAGE_READWRITE, 0, cb, NULL);
HBITMAP cbmp = CreateDIBSection(hdc, &bmi, DIB_RGB_COLORS,
   (VOID**)&dibs, mo, 0);
. . .
CloseHandle(mo);
CloseHandle(bf);
```

Файл отображения не является файлом в формате BMP. Важно — параметр ***смещение к битовому массиву*** должен быть числом, кратным четырем.

С помощью файла отображения битовый массив можно записать в отображаемый файл (в примере «*tmf.dat*»). Записанный битовый массив можно использовать далее для инициализации *dib*-картинки. Кроме этого, файл отображения можно использовать для одновременного совместного доступа к картинке из нескольких приложений (потоков). Для этой цели файл отображения должен быть поименованным (последний параметр функции `CreateFileMapping`).

Прозрачность

Картинка с глубиной цвета *32* может определять степень прозрачности (*transparency*) пикселей. Старший байт тетрады **RGBQUAD** дает возможность задать значение прозрачности от нуля (полностью прозрачный пиксель) до *255* (полностью непрозрачный пиксель). Кроме того, прозрачность может быть установлена для картинки в целом.

Для вывода картинки с прозрачностью используется функция

```
BOOL AlphaBlend(
   HDC hdcDest,                 /* контекст приемника           */
   int nXOriginDest,            /* x-коорд. нач. точки приемника */
   int nYOriginDest,            /* y-коорд. нач. точки приемника */
   int nWidthDest,              /* ширина в приемнике           */
   int nHeightDest,             /* высота в приемнике           */
   HDC hdcSrc,                  /* контекст источника           */
   int nXOriginSrc,             /* x-коорд. нач. точки источника */
   int nYOriginSrc,             /* y-коорд. нач. точки источника */
   int nWidthSrc,               /* ширина в источнике           */
   int nHeightSrc,              /* высота в источнике           */
   BLENDFUNCTION blendFunction  /* функция                      */
);
```

Структура **BLENDFUNCTION** описывает, как вычисляется прозрачность:

```
typedef struct _BLENDFUNCTION {
   BYTE BlendOp;                /* операция смешивания           */
   BYTE BlendFlags;             /* параметры смешивания          */
   BYTE SourceConstantAlpha;    /* общий коэффициент прозрачности */
   BYTE AlphaFormat;            /* формат прозрачности           */
} BLENDFUNCTION, *PBLENDFUNCTION;
```

Поле *операция смешивания* может быть только **AC_SRC_OVER=0**. *Параметры смешивания* — всегда ноль. *Общий коэффициент прозрачности* управляет прозрачностью картинки в целом. Если он равен нулю, картинка полностью прозрачна. Если он равен *255*, прозрачность определяется прозрачностью только пикселя. Иначе общая прозрачность и прозрачность пикселя смешиваются. Параметр *формат прозрачности* может принимать значение **AC_SRC_ALPHA=0**, причем константа не определена. В проект, вызывающий функцию **AlphaBlend**, нужно включить библиотеку *MSIMG32.lib*, добавив ее в список *Project—Settings—Link—Object/Library modules*.

Если источник не содержит канала *Alpha*, смешение происходит по формуле

```
Dst.Red = Src.Red * SCA/255 + Dst.Red * (1 - SCA/255).
```

Формула приведена для одной составляющей цвета. **Src.Red** — составляющая цвета в источнике, **Dst.Red** — составляющая цвета в приемнике, **SCA** — общий коэффициент прозрачности, поле **SourceConstantAlpha**.

Если приемник имеет канал *Alpha*, смешение происходит по формуле

```
Dst.Alpha = Src.Alpha * SCA/255 + Dst.Alpha * (1 - SCA/255).
```

Если в источнике **SCA=255**, смешение происходит по формуле

```
Dst.Red = Src.Red + Dst.Red * (1 - Src.Alpha).
```

Здесь **Src.Alpha** — прозрачность пикселя.

Если при этом приемник имеет канал *Alpha*, смешение происходит по формуле

```
Dst.Alpha = Src.Alpha + Dst.Alpha * (1 - Src.Alpha).
```

Если **SCA<>255** и источник содержит канал *Alpha*, перед смешением выполняется умножение прозрачности пикселя на **SCA**. Результирующие формулы:

```
Src.Red = Src.Red * SCA/255
Dst.Red = Src.Red + (1 - Src.Alpha) * Dst.Red
Src.Alpha = Src.Alpha * SCA/255
Dst.Alpha = Src.Alpha + (1 - Src.Alpha) * Dst.Alpha
```

Если размер приемника не совпадает с размером источника, происходит масштабирование, при этом автоматически устанавливается режим **COLORONCOLOR**.

Если установлено преобразование, оно производится, за исключением отражения. Если цветовые форматы источника и приемника различаются, формат источника приводится к приемнику.

Еще одна функция имеет отношение к прозрачности. Это функция **TransparentBlt**, которая является расширенной версией функции **BitBlt**:

```
BOOL TransparentBlt(
   HDC hdcDest,        /* контекст приемника            */
   int nXOriginDest,   /* x-коорд. нач. точки приемника */
   int nYOriginDest,   /* y-коорд. нач. точки приемника */
   int nWidthDest,     /* ширина в приемнике            */
   int hHeightDest,    /* высота в приемнике            */
   HDC hdcSrc,         /* контекст источника            */
   int nXOriginSrc,    /* x-коорд. нач. точки источника */
   int nYOriginSrc,    /* y-коорд. нач. точки источника */
   int nWidthSrc,      /* ширина в источнике            */
   int nHeightSrc,     /* высота в источнике            */
   UINT crTransparent /* прозрачный цвет                */
);
```

Параметр **crTransparent** — цвет в формате RGB, который не переносится в приемник (делает соответствующий пиксель источника прозрачным).

Шрифты

Шрифт — это коллекция символов и знаков, имеющих одинаковый стиль написания. Три важнейших элемента стиля — это гарнитура (*typeface*), начертание (*style*) и размер (*size*).

Гарнитура определяет внешний вид символов разной шириной толстых и тонких основных штрихов, а также наличием или отсутствием засечек.

Засечка (*serif*) — небольшой поперечный штрих на конце основного штриха. Шрифт без засечек принято называть *sans-serif*.

Начертание определяет жирность (*weight*) и наклон (*slant*) символов.

Наклон символов определяется терминами *roman*, *oblique (наклонный)* и *italic (курсив)*. Табл. 25 описывает наклон символов.

Табл. 24. Наклон символов

Начертание	***Описание***
roman	Прямые, вертикальные символы.
oblique	Наклонные символы. Получаются *сдвигом* начертания *roman*.
italic	Истинно наклонный шрифт. Символы этого шрифта описаны как наклонные в источнике (файле) шрифта.

Жирность символов определяется терминами от *thin* (тонкий) до *heavy* (тяжелый) так, как представлено в табл. 26.

Табл. 25. Жирность символов

Константа	***Значение***	***Описание***
FW_DONTCARE	0	не имеет значения
FW_THIN	100	тонкий
FW_EXTRALIGHT, FW_ULTRALIGHT	200	очень легкий
FW_LIGHT	300	легкий
FW_NORMAL, FW_REGULAR	400	нормальный
FW_MEDIUM	500	средний
FW_SEMIBOLD, FW_DEMIBOLD	600	полужирный
FW_BOLD	700	жирный
FW_EXTRABOLD, FW_ULTRABOLD	800	очень жирный
FW_HEAVY, FW_BLACK	900	тяжелый, черный

Размер шрифта — это высота от верхней части буквы типа ***A*** до нижней части буквы типа ***g***. Для измерения используются пункты (*point*) — 1 пункт = 0,013837 дюйма = 0,351 мм. На практике используют соотношение 1 пункт = 1/72 = 0,01389 дюйма = 0,353 мм. Дополнительно см. рис. 62, стр. 118.

Логические характеристики шрифта описываются структурой **LOGFONT** (стр. 117). Она используется для косвенного создания шрифта при помощи функции **CreateFontIndirect**, а также для получения характеристик шрифта при поиске функциями типа **EnumFontFamilies** и **ChooseFont**. Физические характеристики шрифта описываются структурой **TEXTMETRIC** (стр. 117). Получить их можно при помощи функции

BOOL GetTextMetrics(*контекст, характеристики*);

Параметр ***характеристики*** — указатель на структуру **TEXTMETRIC**.

Важным параметром шрифта является расстояние между символами — *pitch*. Различают пропорциональные и моноширинные шрифты (табл. 27).

Табл. 26. Шаг шрифта

Константа	***Значение***	***Описание***
FIXED_PITCH	1	*Моноширинный*. Символы имеют одинаковую ширину.
VARIABLE_PITCH	2	*Пропорциональный*. Ширина зависит от символа.

Существует четыре ***технологии получения символов*** шрифтов: ***растровая*** (*raster*), ***векторная*** (*vector*), ***TrueType*** и ***OpenType***.

Соответственно есть растровые (экранные) и векторные шрифты, а также шрифты *TrueType* и *OpenType*. В растровых шрифтах ***глиф*** (образ символа) определяется пиксельным набором. В векторных шрифтах глиф задается концевыми точками своих штрихов. В шрифтах *TrueType* и *OpenType* глиф описывается командами рисования линий и кривых, а также набором подсказок (*hint*). Растровые шрифты зависимы от устройства, в то время как другие типы шрифтов позволяют легко масштабировать глифы на устройстве любого типа.

Семейством называется набор шрифтов с одинаковой шириной штрихов и наличием или отсутствием засечек. Существует 5 семейств (табл. 28) — *Roman* (типа *Times New Roman*), *Swiss* (типа *Arial*), *Modern* (типа *Courier New*), *Script* и *Decorative* (типа *Old English*). Семейство *Dontcare* предназначено для шрифта по умолчанию и используется, если информации о шрифте отсутствует или ес-

ли она не важна. Внутри семейств шрифты различаются размером и начертанием. Семейства используются при создании и выборе шрифтов, а также при извлечении информации о них.

Табл. 27. Семейства шрифтов

Константа	*Значение*	*Семейство*	*Описание*
FF_DONTCARE	0x00	Dontcare	Шрифт по умолчанию.
FF_ROMAN	0x10	Roman	Пропорциональный шрифт с засечками.
FF_SWISS	0x20	Swiss	Пропорциональный шрифт без засечек.
FF_MODERN	0x30	Modern	Моноширинный шрифт.
FF_SCRIPT	0x40	Script	Ручное написание.
FF_DECORATIVE	0x50	Decorative	Романтический стиль.

Использовать можно только шрифт, встроенный в устройство или установленный в системную таблицу шрифтов. ***Чтобы использовать шрифт*** для вывода текста, нужно взять стандартный шрифт из фонда или создать новый логический шрифт, и затем выбрать его в контекст устройства.

Выбор стандартного шрифта осуществляется обычным образом, при помощи функции **GetStockObject**. Следующий пример показывает, как из фонда выбирается *шрифт по умолчанию для пользователя*:

Листинг 9. Выбор стандартного шрифта

```
HFONT hf = (HFONT)GetStockObject(DEFAULT_GUI_FONT);
HFONT exhf = (HFONT)SelectObject(hdc, hf);
TextOut(hdc, 10, 10, "ABCDEF 012345 АБВГДЕ", 20);
SelectObject(hdc, exhf);
DeleteObject(hf);
```

Во многих случаях выбор шрифта предоставляется пользователю. В этом случае при помощи функции **ChooseFont** вызывается стандартный диалог для выбора шрифта. Характеристики шрифта, выбранного пользователем, возвращаются в виде структуры **LOGFONT** (листинг 10).

Листинг 10. Выбор шрифта при помощи стандартного диалога

```
void choose(HWND hWnd) {
   HDC hdc; CHOOSEFONT cf; static LOGFONT lf;
   static DWORD rgbCurrent;
   ZeroMemory(&cf, sizeof(CHOOSEFONT));
   cf.lStructSize = sizeof(CHOOSEFONT);
   cf.hwndOwner = hWnd;
   cf.lpLogFont = &lf;
   cf.rgbColors = rgbCurrent;
   cf.Flags = CF_SCREENFONTS | CF_EFFECTS;
   if (ChooseFont(&cf) == TRUE) {
      hdc = GetDC(hWnd);
      HFONT hf = CreateFontIndirect(cf.lpLogFont);
      SelectObject(hdc, hf);
      rgbCurrent= cf.rgbColors;
      SetTextColor(hdc, rgbCurrent);
      TextOut(hdc, 10, 10, "AbCdEf 012345 АБВГДЕ", 20);
      ReleaseDC(hWnd, hdc);
   }
}
```

Если есть уверенность, что требуемый шрифт в системе установлен, можно создать логический шрифт при помощи функции

```
HFONT CreateFont(
  int nHeight,               /* высота                        */
  int nWidth,                /* средняя ширина символа        */
  int nEscapement,           /* угол разворота строки         */
  int nOrientation,          /* угол поворота символов        */
  int fnWeight,              /* жирность                      */
  DWORD fdwItalic,           /* признак курсива               */
  DWORD fdwUnderline,        /* признак подчеркивания         */
  DWORD fdwStrikeOut,        /* признак перечеркивания        */
  DWORD fdwCharSet,          /* дескриптор набора символов    */
  DWORD fdwOutputPrecision,  /* точность вывода               */
  DWORD fdwClipPrecision,    /* точность отсечения            */
  DWORD fdwQuality,          /* качество                      */
  DWORD fdwPitchAndFamily,   /* шаг и семейство               */
  LPCTSTR lpszFace           /* название гарнитуры            */
);
```

Несмотря на обилие параметров, установить можно только три или четыре. Во-первых, параметр ***высота***, во-вторых — ***набор символов***. Для России набор символов имеет обозначение `RUSSIAN_CHARSET=204`. В-третьих, нужно либо указать требуемый тип шага и семейство (используя операцию `OR`), либо название шрифта. Следующий пример создает шрифт Arial высотой 16:

Листинг 11. Выбор установленного шрифта

```
hf = CreateFont(16,0,0,0,0,0,0,0,RUSSIAN_CHARSET,0,0,0,0,"ARIAL");
exhf = (HFONT)SelectObject(hdc, hf);
TextOut(hdc, 10, 10, "ABCDEF 012345 АБВГДЕ", 20);
SelectObject(hdc, exhf);
DeleteObject(hf);
```

Наконец, можно создать произвольный шрифт заданного семейства с необходимым типом шага, высотой и шириной символов:

Листинг 12. Создание произвольного шрифта

```
DWORD dwPIF = VARIABLE_PITCH | FF_ROMAN;
hf = CreateFont(16,5,0,0,0,0,0,0,RUSSIAN_CHARSET,0,0,0,dwPIF,"");
exhf = (HFONT)SelectObject(hdc, hf);
TextOut(hdc, 10, 10, "ABCDEF 012345 АБВГДЕ", 20);
SelectObject(hdc, exhf);
DeleteObject(hf);
```

Области и пути

Области (*region*) и пути (*path*) — это замкнутые фигуры, которые создаются комбинацией прямых, кривых, прямоугольников, многоугольников и эллипсов. Полученные фигуры используются для рисования, отсечения вывода и определения принадлежности точки. В контексте не создаются графические объекты этих типов по умолчанию.

Основное назначение областей — отсечение вывода и определение принадлежности точки, например, тестирование положения курсора. Основное назначение путей — создание сложных фигур для закраски.

Области и пути рассматриваются ниже (стр. 96 и стр. 98).

Информационный контекст

Информационный контекст используется для получения информации об устройстве и создается при помощи функции

```
HDC CreateIC(
   LPCTSTR lpszDriver,       /* имя драйвера              */
   LPCTSTR lpszDevice,       /* имя устройства            */
   LPCTSTR lpszOutput,       /* порт или имя файла        */
   CONST DEVMODE *lpdvmInit  /* данные для инициализации */
);
```

Для принтера используется имя драйвера **"WINSPOOL"** или **"WINSPL16"**, для дисплея — **"DISPLAY"**. Возможны и другие имена. Имя устройства записывается так, как оно указано в диспетчере соответствующих устройств. Для дисплея имя устройства принимается **NULL**. Параметр ***порт*** принимается равным **NULL**. Данные для инициализации также могут быть опущены.

После получения информационного контекста он используется для извлечения дескриптора текущего объекта при помощи функции

HGDIOBJ GetCurrentObject(*контекст, объект*);

Параметр ***объект*** выбирается из табл. 29.

Табл. 28. Константы типов графических объектов

Объект	*Значение*	*Описание*
OBJ_PEN	1	Текущее перо.
OBJ_BRUSH	2	Текущая кисть.
OBJ_PAL	5	Текущая палитра.
OBJ_FONT	6	Текущий шрифт.
OBJ_BITMAP	7	Текущий пиксельный набор.

Далее получить ***информацию об объекте*** можно при помощи функции

int GetObject(HGDIOBJ *объект*, int *размер_буфера*, LPVOID *буфер*);

Параметр ***объект*** — дескриптор графического объекта.

Получить информацию можно для объекта типа картинка, кисть, шрифт, палитра и перо. В качестве буфера используется одна из следующих структур:

Тип объекта	*Тип буфера*
HBITMAP	BITMAP
HBITMAP (объект создан CreateDIBSection)	DIBSECTION
HBRUSH	LOGBRUSH
HFONT	LOGFONT
HPALETTE	WORD (число элементов палитры)
HPEN	LOGPEN
HPEN (объект создан ExtCreatePen)	EXTLOGPEN

Функция **GetObject** возвращает размер буфера, если вызвать ее с первым параметром, равным **NULL**. Дополнительно об использовании функции **GetObject** см. раздел «*Выбор и изменение графических объектов*» ниже.

Информацию об устройстве возвращает также функция

int GetDeviceCaps(HDC hdc, int nIndex);

Параметр **nIndex** определяет требуемую характеристику. Список значений этого параметра см. в приложении «*Характеристики устройств*», стр. 114.

Выбор и изменение графических объектов

Как говорилось ранее, для нужд приложения создаются новые перья и кисти. Однако на практике часто необходимо изменить лишь цвет объекта. Есть две функции, предназначенные для этой цели:

```
COLORREF SetDCPenColor(HDC hdc, COLORREF crColor);
COLORREF SetDCBrushColor(HDC hdc, COLORREF crColor);
```

Обе функции возвращают цвет, который был установлен до вызова. При этом перо или кисть не обязательно должны быть выбраны в контекст (хотя использовать их можно только после выбора). Следует помнить, что речь идет только о фондовых (*stock*) объектах.

Следующий пример демонстрирует применение функции **SetDCPenColor**:

```
SelectObject(hdc, GetStockObject(DC_PEN));
SetDCPenColor(hdc, RGB(255, 0, 0));
Rectangle(hdc, 1, 1, 100, 100);
```

Аналогичным образом изменяется цвет фондовой кисти.

Следует также обратить внимание на способ выбора фондового объекта. Нет необходимости получать дескриптор объекта, так как стандартные объекты удалять не рекомендуется:

```
SelectObject(hdc, GetStockObject(DC_PEN));
```

Важно: указанные функции доступны только на платформе NT (Windows 2000 и выше). Для успешной компиляции нужно определить дополнительный символ **_WIN32_WINNT** в файле *StdAfx.h*, *перед* включением файла *windows.h*:

```
#define _WIN32_WINNT 0x0500
#include <windows.h>
```

Для получения требуемых перьев и кистей можно также использовать функцию **GetObject** и неявное создание объекта. В следующем примере эта функция возвращает сведения о текущем пере в структуру **LOGPEN**, далее изменяется стиль и косвенно создается новое перо:

```
LOGPEN lp;
GetObject(GetCurrentObject(hdc, OBJ_PEN), sizeof(lp), &lp);
lp.lopnStyle = PS_DASH;
HPEN hp = CreatePenIndirect(&lp);
SelectObject(hdc, hp);
Rectangle(hdc, 1, 1, 100, 100);
DeleteObject(hp);
```

Так же можно получить и другие объекты, например, шрифт:

```
LOGFONT lf;
GetObject(GetCurrentObject(hdc, OBJ_FONT), sizeof(lf), &lf);
strcpy(lf.lfFaceName, "Times New Roman");
lf.lfHeight = 72;
lf.lfWidth = 0;
lf.lfWeight = FW_HEAVY;
HFONT hf = CreateFontIndirect(&lf);
SelectObject(hdc, hf);
TextOut(hdc, 0, 0, "Hello!", 6);
DeleteObject(hf);
```

Здесь ширина символов обнуляется для сохранения пропорциональности.

Управление графическим выводом

Графический вывод отображается в конкретный контекст устройства, чтобы не происходило наложения графического вывода от разных приложений на общем экране. Чтобы изображение отображалось правильно, графический вывод располагается между двумя функциями, играющими роль скобок, ограничивающих описание процесса рисования.

Если используется контекст дисплея, роль таких скобок выполняют пара функций **BeginPaint** и **EndPaint** или **GetDC** и **ReleaseDC**. Первая пара используется при обработке события **WM_PAINT**:

```
PAINTSTRUCT ps;
HDC hdc = BeginPaint(hWnd, &ps);
/* графический вывод */
EndPaint(hWnd, &ps);
```

Функция **BeginPaint** подготавливает устройство к графическому выводу и заполняет структуру **PAINTSTRUCT**. Функция **EndPaint** завершает процесс рисования и должна вызываться обязательно, если первая функция была вызвана. Эта функция *освобождает контекст устройства*.

Во всех других случаях используется пара **GetDC** и **ReleaseDC**:

```
HDC hdc = GetDC(hWnd, &ps);
/* графический вывод */
ReleaseDC(hWnd, &ps);
```

Если используется контекст принтера, роль ограничивающих процесс рисования скобок, кроме **CreateDC** и **DeleteDC**, выполняют также функции **StartDoc** и **EndDoc**. В следующем примере эти функции используются внутри вызова функции **Escape**:

```
void printer(HWND hWnd) {
   HDC  hdc;
   hdc = CreateDC("WINSPOOL", "hp LaserJet 1015", 0, NULL);
   Escape(hdc, STARTDOC, 8, "Test-Doc", NULL);
   TextOut(hdc, 100, 100, "PRINTER TEST", 12);
   /* другой графический вывод */
   Escape(hdc, NEWFRAME, 0, NULL, NULL); /* конец страницы */
   Escape(hdc, ENDDOC, 0, NULL, NULL);
   DeleteDC(hdc);
}
```

Есть две разновидности контекстов дисплея: общий (*common*) и частный (*private*). Существующий классовый контекст устарел и здесь не рассматривается.

Общий контекст содержится в кэше системы. Он используется приложениями, которые рисуют сравнительно редко. Приложение получает контекст, выполняет графический вывод и освобождает контекст. Все изменения в контексте (в графических объектах и режимах) теряются. Если приложению нужно нарисовать что-то еще, снова нужно получить контекст, нарисовать и освободить контекст. Количество активных общих контекстов ограничено.

Частный контекст относится к окну приложения (к классу окна). После освобождения этот контекст сохраняет всю информацию. Частный контекст необходим приложениям, интенсивно использующим графический вывод, таким, как CAD (САПР) системы, графические редакторы и т. п.

Частный контекст создается во время инициализации окна. Поле **style** структуры, описывающей окно, должно устанавливать бит **CS_OWNDC**. Все окна, созданные на основе такого класса, будут обладать собственным контекстом. После получения частного контекста его можно использовать вплоть до завершения работы приложения (до закрытия окна данного класса). Для рисующих приложений имеет смысл создание специального окна для рисования с частным контекстом. В следующем примере приведена процедура создания такого окна:

Листинг 13. Создание окна для графического вывода

```
BOOL CreateGD(HINSTANCE hInstance, HWND hWnd) {
   WNDCLASSEX wcex;
   memset(&wcex, 0, sizeof(WNDCLASSEX));
   wcex.cbSize = sizeof(WNDCLASSEX);
   wcex.style = CS_OWNDC;       /* окно с частным контекстом */
   wcex.lpfnWndProc = (WNDPROC)GDProc; /* оконная процедура */
   wcex.hInstance = hInstance;
   wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
   wcex.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
   wcex.lpszClassName = "GDClass";
   RegisterClassEx(&wcex);
   hGDWnd = CreateWindow("GDClass", "", WS_CHILDWINDOW | WS_BORDER,
      0, 0, 400, 400, hWnd, NULL, hInstance, NULL);
   if (!hGDWnd) return FALSE;
   ShowWindow(hGDWnd, SW_SHOW);
   UpdateWindow(hGDWnd);
   hGDDC = GetDC(hGDWnd);       /* создание частного контекста */
   return TRUE;
}
```

Параметр **hWnd** указывает на окно приложения (родительское), в котором будет расположено окно графического вывода. Функция **CreateGD** заполняет структуру **wcex**, описывающую окно, регистрирует класс «*GDClass*» и создает окно с частным контекстом **hGDDC**. Окно располагается в клиентской части родительского окна в позиции *0,0* и имеет размер *400×400*. Переменные **hGDWnd** и **hGDDC** должны быть объявлены как глобальные. Для обработки сообщений окна графического вывода должна быть описана оконная процедура, аналогичная процедуре родительского окна, например:

```
LRESULT CALLBACK
GDProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam) {
   switch (msg) {
      case WM_PAINT:
         . . .
         break;
      default:
         return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;

}
```

Оконная процедура регистрируется во время создания окна для графического вывода. Для интерактивного рисования (мышью) она должна обрабатывать события курсора (мыши), такие, как нажатие, отпускание кнопок и перемещение. Дополнительно см. пример рисования мышью, стр. 102, и листинг 15.

Функция **CreateGD** вызывается в основной процедуре приложения **WinMain** после инициализации родительского окна:

```
CreateGD(hInstance, hMainWnd);
```

Здесь **hMainWnd** — глобальная переменная, дескриптор родительского окна. Она должна быть инициализирована в процедуре создания родительского окна:

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow) {
   hInst = hInstance;
   hMainWnd = CreateWindow(. . .);
   . . .
}
```

Функция **CreateGD** создает частный контекст **hGDDC**, который в дальнейшем используется для рисования *без вызова* функций **GetDC** и **ReleaseDC**.

Событие WM_PAINT

Приложения рисуют изображение в различные моменты времени.

Когда окно создается, изображение в клиентской части должно быть нарисовано в первый раз. При изменении размеров окон и при их относительном перемещении может быть открыта ранее скрытая часть окна, которую также нужно нарисовать. Моменты создания, уничтожения, перемещения и изменения размеров окон *отслеживает операционная система*, которая при необходимости и при первой возможности посылает оконной процедуре сообщение **WM_PAINT**. Это сообщение является сигналом приложению о необходимости обновления содержимого окна.

Во всех других случаях инициатором изменения графического содержимого окна является приложение. Например, приложению нужно показать содержимое открываемого файла. В этом случае приложение отмечает часть окна, требующую изменения изображения, как область обновления. Система *отслеживает изменение области обновления*, и при первой возможности посылает приложению сообщение **WM_PAINT**.

Система посылает сообщение **WM_PAINT** только тогда, когда очередь сообщений приложения пуста, предполагая, что графический вывод в окно имеет самый низкий приоритет по отношению к другим задачам. Это позволяет системе накапливать отдельные, возможно, небольшие изменения изображения, и обрабатывать их за один раз.

Получив сообщение **WM_PAINT**, приложение принимает решение о том, что и где следует нарисовать. В простейшем случае перерисовывается изображение всего окна, хотя такое решение может оказаться не всегда экономным. Обрабатывая сообщение **WM_PAINT**, приложение получает контекст устройства, используя функцию **BeginPaint**.

Иногда изображение необходимо вывести немедленно, не дожидаясь получения сообщения **WM_PAINT**. Например, пользователь рисует мышью, или выделяет часть информации. В этом случае изображение рисуется по мере необходимости с использованием контекста, получаемого при помощи функции **GetDC**. Чтобы динамически нарисованное изображение не исчезло при первом же событии **WM_PAINT**, должен быть предусмотрен какой-нибудь механизм отображения изменений графического содержания окна в обработке этого события. Разработка такого механизма составляет одну из важных и сложных задач графического приложения.

Область обновления

Вывод изображения в ответ на событие **WM_PAINT** является наилучшим решением во всех практических случаях. Событие **WM_PAINT** посылается системой, когда появляется *непустая область обновления*. ***Область обновления*** (*update region*) — это часть клиентской области окна, изображение которой требует перерисовки, так как текущее изображение в этой части «устарело», и не отображает изменений по ходу выполнения приложения.

Как было сказано ранее, для получения контекста в этом случае используется функция **BeginPaint**. Эта функция заполняет структуру **PAINTSTRUCT**:

```
typedef struct tagPAINTSTRUCT {
   HDC  hdc;              /* контекст                  */
   BOOL fErase;           /* признак перерисовки фона */
   RECT rcPaint;          /* область обновления        */
   BOOL fRestore;         /* зарезервировано системой */
   BOOL fIncUpdate;       /* зарезервировано системой */
   BYTE rgbReserved[32]; /* зарезервировано системой */
} PAINTSTRUCT, *PPAINTSTRUCT;
```

Поле **rcPaint** указывает на область, нуждающуюся в перерисовке. Поле **fErase** указывает, кто отвечает за обновление фона. Если это поле *ненулевое*, рисовать фон обязано приложение. Такое положение возникает в случае, если при создании окна не была указана кисть (не установлено поле **hbrBackground** структуры **WNDCLASSEX**).

Функция **BeginPaint** обнуляет область обновления. Если приложение использует для рисования частный контекст, или контекст, полученный при помощи функции **GetDC**, оно *должно самостоятельно обнулить область обновления*, в противном случае система будет непрерывно посылать сообщение **WM_PAINT**, и это проявится в виде мелькания в клиентской части окна.

Приложение не обязательно использует область обновления. В силу сложности вырисовывания отдельных частей графического изображения, или, наоборот, в силу его простоты, приложение может перерисовывать изображение во всей клиентской части, полагаясь на область отсечения. Система выводит изображение только в пределах ***области отсечения*** (*clip region*), которая является результатом пересечения области обновления и видимой области окна.

Приложение устанавливает биты **CS_HREDRAW** и **CS_VREDRAW** поля **style** структуры **WNDCLASSEX** для того, чтобы получать событие **WM_PAINT** при изменении соответственно горизонтального или вертикального размера окна. Заметим, что при уменьшении размеров окна событие генерируется дважды.

Для определения области рисования, равной всей клиентской части окна, приложение может использовать функцию

```
BOOL GetClientRect(HWND hWnd, LPRECT lpRect);
```

Параметр **lpRect** получает размер области.

Далее приложение рисует необходимое изображение с учетом новых размеров. В следующем примере синусоида вписывается во всю ширину и высоту клиентской части:

Листинг 14. Графический вывод по ширине и высоте окна

```
   RECT rc;
   int i; double x = 0, dx = 6.28 / 32;
```

```
   switch (message) {
      case WM_PAINT:
         hdc = BeginPaint(hWnd, &ps);
         GetClientRect(hWnd, &rc);
         SetMapMode(hdc, MM_ANISOTROPIC);
         SetWindowExtEx(hdc, 628, 200, NULL);
         SetViewportExtEx(hdc, rc.right, rc.bottom, NULL);
         MoveToEx(hdc, 0, 100, NULL);
         for (i = 0; i < 32; i++) {
            x += dx;
            LineTo(hdc, 100 * x, 100 + 100 * sin(x));
         }
         EndPaint(hWnd, &ps);
         break;
```

Если приложение может вычислить, как нарисовать часть изображения, оно может использовать область обновления для вывода только этой части. В следующем примере область обновления используется для копирования изображения из контекста памяти `hCDC`:

Листинг 15. Графический вывод в области обновления

```
LRESULT CALLBACK
GDProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {
   PAINTSTRUCT ps; HDC hdc;
   int L, T, W, H;
   switch (message) {
      case WM_PAINT:
         hdc = BeginPaint(hWnd, &ps);
         L = ps.rcPaint.left;
         T = ps.rcPaint.top;
         W = ps.rcPaint.right - L;
         H = ps.rcPaint.bottom - T;
         BitBlt(hGDDC, L, T, W, H, hCDC, L, T, SRCCOPY);
         EndPaint(hWnd, &ps);
         break;
      case WM_LBUTTONDOWN:
         pMouseDown = TRUE;
         X1 = LOWORD(lParam);
         Y1 = HIWORD(lParam);
         break;
      case WM_MOUSEMOVE:
         if (pMouseDown) DrawLine(LOWORD(lParam), HIWORD(lParam));
         break;
      case WM_LBUTTONUP:
         if (pMouseDown) DrawLine(LOWORD(lParam), HIWORD(lParam));
         pMouseDown = FALSE;
         break;
      default:
         return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Здесь вывод осуществляется в специальное окно для графического вывода, описанное выше. Контекст памяти **hCDC** создается в процедуре создания этого окна:

```
BOOL CreateGD(HINSTANCE hInstance, HWND hWnd) {
   . . .
   hGDDC = GetDC(hGDWnd);
   hCDC = CreateCompatibleDC(hGDDC);
   hCBMP = CreateCompatibleBitmap(hGDDC, 400, 400);
   SelectObject(hCDC, hCBMP);
   PatBlt(hCDC, 0, 0, 400, 400, PATCOPY);
   UpdateWindow(hGDWnd);
   return TRUE;
}
```

Рисование производится мышью в контекст памяти. Для рисования используются две глобальные переменные **X1** и **Y1**. Процедура рисования отрезка прямой, который создается при одном движении мыши, вычисляет прямоугольник, требующий обновления:

```
void DrawLine(int X2, int Y2) {
   RECT rc;
   MoveToEx(hCDC, X1, Y1, NULL);
   LineTo(hCDC, X2, Y2);
   rc.left = min(X1, X2);
   rc.top = min(Y1, Y2);
   rc.right = 1 + rc.left + abs(X1 - X2);
   rc.bottom = 1 + rc.top + abs(Y1 - Y2);
   InvalidateRect(hGDWnd, &rc, 0);
   X1 = X2;
   Y1 = Y2;
}
```

Функция **InvalidateRect** добавляет указанный прямоугольник к области обновления, которая *накапливается* и рисуется по мере возможности:

```
BOOL InvalidateRect(HWND hWnd, CONST RECT* lpRect, BOOL bErase);
```

Если параметр **lpRect** равен **NULL**, перерисовывается вся клиентская область. Параметр **bErase**, равный **TRUE**, вызывает очистку фона во время вызова функции **BeginPaint**.

Область обновления изменяет также функция

```
BOOL InvalidateRgn(HWND hWnd, HRGN hRgn, BOOL bErase);
```

Отличие этой функции от предыдущей заключается в том, что к области обновления добавляется прямоугольник, описанный вокруг области **hRgn**.

Противоположное действие оказывают следующие функции:

```
BOOL ValidateRect(HWND hWnd, CONST RECT *lpRect);
BOOL ValidateRgn(HWND hWnd, HRGN hRgn);
```

Они исключают прямоугольник или область из области обновления.

Вычислить область обновления можно при помощи функции

```
BOOL GetUpdateRect(HWND hWnd, LPRECT lpRect, BOOL bErase);
```

Параметр **bErase**, равный **TRUE**, вызывает очистку фона.

В следующем примере функция **GetUpdateRect** используется для извлечения области обновления, а функция **ValidateRect** обнуляет ее:

```
RECT rc;
int L, T, W, H;
switch (message) {
   case WM_PAINT:
      GetUpdateRect(hWnd, &rc, 0);
      L = rc.left;
      T = rc.top;
      W = rc.right - L;
      H = rc.bottom - T;
      BitBlt(hGDDC, L, T, W, H, hCDC, L, T, SRCCOPY);
      ValidateRect(hGDWnd, &rc);
      break;
```

Функции **BeginPaint** и **EndPaint** здесь не нужны.

Вместо области обновления приложение может также использовать *область отсечения*. Дополнительно см. раздел «*Отсечение вывода*», стр. 99.

Очистка фона

Во время выполнения функции **BeginPaint** система генерирует и посылает событие **WM_ERASEBKGND**, которое сигнализирует о необходимости обновить фон окна. Обычно обработку этого события передают системе:

```
switch (message) {
   case WM_PAINT:
   . . .
   default:
      return DefWindowProc(hWnd, message, wParam, lParam);
}
```

Система очищает фон, используя кисть, определенную полем **hbrBackground** структуры **WNDCLASSEX**. Если кисть недоступна (поле равно **NULL**), система устанавливает флаг **fErase** структуры **PAINTSTRUCT**, чтобы уведомить приложение.

Если приложение управляет очисткой окна, оно может использовать параметр **wParam** сообщения, которое содержит контекст устройства. После завершения очистки приложение должно вернуть *ненулевое значение*, чтобы исключить ошибочную установку флага **fErase**. В следующем примере создается кисть со стандартной штриховкой и используется для создания фона в обработке события **WM_ERASEBKGND**:

```
HDC hdc;
HBRUSH hb;
switch (message) {
   case WM_PAINT:
   . . .
   case WM_ERASEBKGND:
      hdc = (HDC)wParam;
      hb = CreateHatchBrush(HS_DIAGCROSS, 0);
      SelectObject(hdc, hb);
      GetClientRect(hWnd, &rc);
      PatBlt(hdc, 0, 0, rc.right, rc.bottom, PATCOPY);
      DeleteObject(hb);
      return 1L;
```

Синхронный графический вывод

Обычно при использовании события **WM_PAINT** возникает задержка между моментом возникновения области обновления и моментом генерации события и выводом изображения. Область обновления при этом может накапливаться последовательными вызовами функции **InvalidateRect**. Система генерирует событие **WM_PAINT** в те моменты времени, когда нет других задач.

Такой графический вывод является *асинхронным*. Если требуется получить немедленную реакцию системы на изменение графического содержимого окна, следует использовать *синхронный* вывод, когда событие **WM_PAINT** посылается непосредственно оконной процедуре, минуя очередь сообщений. Следующие две функции предназначены для синхронного вывода:

```
BOOL UpdateWindow(HWND hWnd);
BOOL RedrawWindow(
   HWND hWnd,
   CONST RECT *lprcUpdate,
   HRGN hrgnUpdate,
   UINT flags
);
```

Вторая функция предоставляет больше возможностей. Во-первых, она позволяет указать как прямоугольник, так и область в качестве области обновления. Если указана область, прямоугольник игнорируется. Во-вторых, с помощью второй функции можно и увеличить, и уменьшить область обновления, в зависимости от значения параметра **flags**, который выбирается из табл. 30.

Табл. 29. Режимы перерисовки окна

Константа	***Значение***	***Описание***
Увеличение области обновления		
RDW_INVALIDATE	**0x1**	Создает область обновления.
RDW_INTERNALPAINT	**0x2**	Генерирует внутреннее событие **WM_PAINT.**
RDW_ERASE	**0x4**	Генерирует событие **WM_ERASEBKGND.**
RDW_FRAME	**0x400**	Генерирует событие **WM_NCPAINT**.
Уменьшение области обновления		
RDW_VALIDATE	**0x8**	Обнуляет область обновления.
RDW_NOINTERNALPAINT	**0x10**	Подавляет внутренние события **WM_PAINT.**
RDW_NOERASE	**0x20**	Подавляет события **WM_ERASEBKGND.**
RDW_NOFRAME	**0x800**	Подавляет события **WM_NCPAINT**.
Время обновления		
RDW_UPDATENOW	**0x100**	**WM_PAINT** посылается немедленно.
RDW_ERASENOW	**0x200**	**WM_PAINT** посылается обычным образом.
Область действия		
RDW_NOCHILDREN	**0x40**	Исключая дочерние окна.
RDW_ALLCHILDREN	**0x80**	Включая дочерние окна.

Внутренние события **WM_PAINT** посылаются независимо от наличия области обновления. Флаги **RDW_ERASE** и **RDW_FRAME** действительны только в сочетании с флагом **RDW_INVALIDATE.** Аналогично, флаги **RDW_NOERASE** и **RDW_NOFRAME** действительны только в сочетании с флагом **RDW_VALIDATE.**

Графические примитивы

Графические примитивы — это графические элементы, из которых формируется изображение. К ним относятся простые примитивы, такие, как отрезок, кривая Безье, строка текста, а также составные, такие, как полилиния или путь. К графическим примитивам применяются графические режимы для смешения цветов, закрашивания, преобразования, а также некоторые графические функции, например, для закрашивания области.

Пиксель

COLORREF SetPixel(***контекст, x, y, цвет***);

Рисует пиксель с координатами ***x, y*** и возвращает цвет, который был установлен на самом деле. Он может отличаться от того, который указан для вывода, вследствие аппроксимации.

COLORREF GetPixel(***контекст, x, y***);

Возвращает цвет указанного пикселя. Если запрашивается пиксель за пределами текущей области отсечения, функция возвращает **CLR_INVALID=-1**.

Не все устройства поддерживают пиксельный вывод. Чтобы проверить, есть ли такая поддержка, следует использовать функцию **GetDeviceCaps** с параметром **TECHNOLOGY=2**. Результат должен соответствовать одной из следующих констант: **DT_RASDISPLAY=1**, **DT_RASPRINTER=2**.

Позиция

BOOL MoveToEx(***контекст, x, y, экс_точка***);

Текущая точка перемещается в ***x, y***. В переменную ***экс_точка***, имеющую тип **POINT**, возвращается текущая точка, которая существовала до вызова.

Прямые и кривые

Отрезок

BOOL LineTo(***контекст, x, y***);

Отрезок рисуется от текущей точки до точки ***x, y***. Если текущее перо геометрическое, используется текущая кисть. Текущая позиция обновляется в точку ***x, y***, при этом сама точка ***x, y*** никогда не рисуется.

Полилиния

BOOL Polyline(***контекст, точки, число_точек***);

Рисует последовательность соединенных отрезков, заданных массивом ***точки***. Массив имеет тип указателя на **POINT**. В отличие от множества последовательных отрезков, соединения сегментов прямых определяются стилем геометрического пера, см. табл. 19. Пример см. функцию **PolyBezier**.

Полилиния до

BOOL PolylineTo(***контекст, точки, число_точек***);

Полностью аналогична предыдущей функции, за исключением того, что линия рисуется от текущей точки до первой точки массива и далее по точкам массива. Текущая точка обновляется в последнюю точку массива.

Множество полилиний

BOOL PolyPolyline(*контекст, точки, числа_точек, число_полилиний*);

Рисует несколько полилиний, не связанных между собой. Массив ***точки*** одержит точки всех полилиний. ***Числа_точек*** — это массив целых чисел, задающих количества точек в полилиниях. В примере рисуется две полилинии, образующие два треугольника:

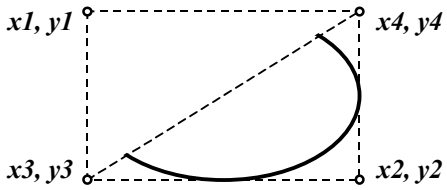
```
POINT pt[] = {{0,0},{0,8},{8,8},{0,0},{8,8},{16,8},{16,0},{8,8}};
ULONG sizes[] = { 4, 4 };
PolyPolyline(hdc, pt, sizes, 2);
```

Дуга

BOOL Arc(*контекст, x1, y1, x2, y2, x3, y3, x4, y4*);

Дуга рисуется в пределах прямоугольника, заданного своими диагональными точками ***x1, y1*** и ***x2, y2***. Начальная точка дуги задается пересечением с радиусом в точку ***x3, y3***. Конечная точка дуги задается пересечением с радиусом в точку ***x4, y4***. Дуга может быть эллиптической. Углы дуг отсчитываются от направления оси *X* против часовой стрелки. Текущая точка не обновляется.

```
hp = CreatePen(PS_DASH, 1, 0);
SelectObject(hdc, hp);
Rectangle(hdc, 0, 0, 105, 65);
MoveToEx(hdc, 0, 64, NULL);
LineTo(hdc, 104, 0);
DeleteObject(hp);
hp = CreatePen(PS_SOLID, 2, 0);
SelectObject(hdc, hp);
Arc(hdc, 0, 0, 105, 65, 0, 64, 104, 0);
DeleteObject(hp);
```



Отрезок и дуга

BOOL AngleArc(*контекст, x, y, радиус, угол_начала, центральный_угол*);

Параметры ***x*** и ***y*** — центр дуги. Отрезок рисуется от текущей точки до начальной точки дуги. Текущая точка не обновляется. В примере сначала устанавливается текущая точка в центр дуги, затем вызывается функция **AngleArc**, которая рисует дугу 90° радиусом 48:

```
MoveToEx(hdc, 0, 48, NULL);
AngleArc(hdc, 0, 48, 48, 0, 90);
```

Отрезок и дуга до

BOOL ArcTo(*контекст, x1, y1, x2, y2, x3, y3, x4, y4*);

Параметры функции аналогичны параметрам функции **Arc**, а действие — функции **AngleArc**. Отрезок рисуется от текущей точки до начальной точки дуги. Текущая точка обновляется в позицию конца дуги.

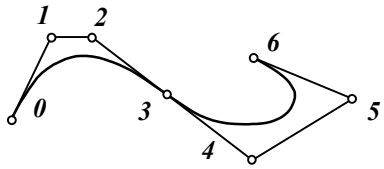
Кривая Безье

BOOL PolyBezier(*контекст, точки, число_точек*);

Рисует гладкую кривую, ограниченную многоугольником, образованным точками массива ***точки***. Кривая Безье состоит из сегментов. Каждый сегмент задается четырьмя точками (тремя прямыми). Две точки являются *конечными*, кривая проходит через них. Две промежуточные точки являются *контрольными*. Кривая никогда не проходит через них, эти точки задают кривизну. Семь

точек строят два сегмента, 10 точек строят три сегмента и т. д. Лишние точки отбрасываются. Точки задаются при помощи структуры **POINT**. На рисунке ниже показаны два сегмента кривой Безье вместе с задающим многоугольником (ломаной линией). Пример демонстрирует также функцию **Polyline**:

```
pt[0].x = 10;  pt[0].y = 60;
pt[1].x = 30;  pt[1].y = 20;
pt[2].x = 50;  pt[2].y = 20;
pt[3].x = 90;  pt[3].y = 50;
pt[4].x = 130; pt[4].y = 80;
pt[5].x = 180; pt[5].y = 50;
pt[6].x = 130; pt[6].y = 30;
PolyBezier(hdc, pt, 7);
Polyline(hdc, pt, 7);
```



Для обеспечения гладкости сопряжения сегментов точки 2, 3 и 4 следует располагать на одной прямой.

Кривая Безье до

BOOL PolyBezierTo(*контекст, точки, число_точек*);

Полностью аналогична предыдущей функции, за исключением того, что первый сегмент кривой рисуется от текущей точки до третьей точки (первые две точки массива ***точки*** являются контрольными). Текущая точка обновляется в конечную точку кривой.

Полилиния с кривыми Безье

BOOL PolyDraw(*контекст, точки, массив_типов_точек, число_точек*);

Рисует сложную линию, состоящую из отрезков прямых и сегментов кривых Безье. Массив ***точки*** задает точки, которые используются как точки перемещения, концевые точки отрезков и точки сегментов кривой Безье. Интерпретация точек задается при помощи массива ***массив_типов_точек***, значения которого определяется следующими константами:

Назначение	*Значение*	*Описание*
PT_CLOSEFIGURE	1	Замыкание
PT_LINETO	2	Конечная точка отрезка
PT_BEZIERTO	4	Точка многоугольника Безье
PT_MOVETO	6	Перемещение текущей точки

Точки многоугольника Безье всегда присутствуют тройками, при этом стартовой точка кривой является текущая точка, две точки являются контрольными, а третья — конечная.

Текущая точка обновляется каждым сегментом прямой или кривой, а также перемещением. Точка может иметь дополнительный признак замыкания из нее на начало линии. Этой точке, кроме константы, указывающей ее тип в массиве типов точек, дополнительно прибавляется константа **PT_CLOSEFIGURE**.

Закрашиваемые фигуры

Все описываемые далее фигуры закрашиваются текущей кистью контекста. На закрашивание влияет режим фона (см. раздел «*Background*», стр. 55).

Прямоугольник

BOOL Rectangle(*контекст, x1, y1, x2, y2*);

Задается диагональными точками ***x1, y1*** и ***x2, y2***. Закрашивается текущей кистью. Текущая точка не обновляется. Прямоугольник рисуется так, как показано на рисунке ниже — крайний правый столбец и крайняя нижняя строка не входят в прямоугольник (это выполняется не всегда). Разница координат вдоль оси дает размер вдоль этой оси.

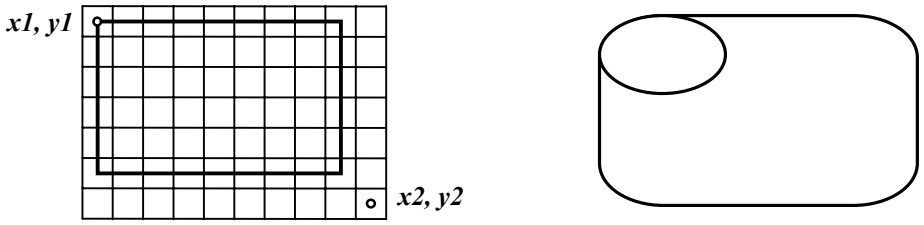


Рис. 56. Прямоугольник (слева) и скругленный прямоугольник (справа)

Скругленный прямоугольник

BOOL RoundRect(*контекст, x1, y1, x2, y2, cx, cy*);

Задается диагональными точками, как прямоугольник. Скругление углов прямоугольника осуществляется по форме эллипса, заданного своими осями ***cx*** и ***cy***. Текущая точка не обновляется. В примере дополнительно рисуется эллипс:

```
RoundRect(hdc, 1, 1, 161, 97, 64, 40);
Ellipse(hdc, 1, 1, 65, 41);
```

Многоугольник

BOOL Polygon(*контекст, точки, число_точек*);

Рисует отрезки прямых от одной точки до другой и замыкает последнюю заданную точку с первой. Количество точек должно быть не менее двух. Текущая точка не обновляется. В примере рисуется треугольник:

```
POINT pt[] = { {0,0},{0,80},{80,80} };
Polygon(hdc, pt, 3);
```

Множество многоугольников

BOOL PolyPolygon(*контекст, точки, числа_точек, число_многоугольников*);

Рисует несколько не связанных между собой многоугольников. Массив ***точки*** содержит точки всех многоугольников. ***Числа_точек*** — массив целых чисел, задающих размеры многоугольников. В примере рисуется два треугольника:

```
POINT pt[] = { {0,0},{0,80},{80,80},{80,80},{160,80},{160,0} };
int sizes[] = { 3, 3 };
int i = PolyPolygon(hdc, pt, sizes, 2);
```

Эллипс (круг)

BOOL Ellipse(*контекст, x1, y1, x2, y2*);

Эллипс рисуется в пределах прямоугольника заданного диагональными вершинами ***x1, y1*** и ***x2, y2***. Эллипс закрашивается текущей кистью контекста. Текущая точка не обновляется. Пример см. «*Скругленный прямоугольник*».

Сектор эллипса (круга)

BOOL Pie(*контекст, x1, y1, x2, y2, x3, y3, x4, y4*);

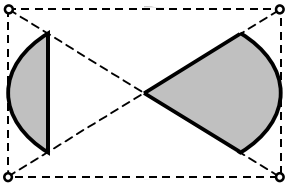
Полностью аналогична функции **Arc**, за исключением того, что концы дуги соединяются прямыми линиями с центром, и полученная замкнутая область закрашивается. Текущая точка не обновляется. Пример см. функцию **Chord**.

Сегмент эллипса (круга)

BOOL Chord(*контекст, x1, y1, x2, y2, x3, y3, x4, y4*);

Полностью аналогична функции **Arc**, за исключением того, что концы дуги соединяются прямой линией, и полученная замкнутая область закрашивается. Текущая точка не обновляется. Пример:

```
hp = CreatePen(PS_DASH, 1, 0);
SelectObject(hdc, hp);
Rectangle(hdc, 0, 0, 105, 65);
DeleteObject(hp);
hp = CreatePen(PS_SOLID, 2, 0);
SelectObject(hdc, hp);
hb = CreateSolidBrush(RGB(192, 192, 192));
SelectObject(hdc, hb);
Chord(hdc, 0, 0, 105, 65, 104, 64, 104, 0);
Pie(hdc, 0, 0, 105, 65, 0, 0, 0, 64);
SelectObject(hdc, exhb);
DeleteObject(hb);
DeleteObject(hp);
```



На рисунке слева **Chord**, справа **Pie**. Две диагональные линии для ориентира.

Графические функции

Рамка

int FrameRect(HDC hDC, CONST RECT *lprc, HBRUSH hbr);

Рисует контур прямоугольника указанной кистью.

Прямоугольник

int FillRect(HDC hDC, CONST RECT *lprc, HBRUSH hbr);

Закрашивает прямоугольную область указанной кистью. В качестве третьего параметра может быть указан системный цвет плюс *1*:

```
FillRect(hdc, &rc, (HBRUSH)(COLOR_WINDOW + 1));
```

Инверсия прямоугольника

BOOL InvertRect(HDC hDC, CONST RECT *lprc);

Инвертирует цвета в область указанного прямоугольника. Если область черно-белая, черный цвет заменяется белым и наоборот. Если область разноцветная, результат зависит от цветовой схемы, но двойное применение функции возвращает область в исходное состояние.

Направление дуги

int SetArcDirection(HDC hdc, int ArcDirection);

Устанавливает направление отсчета углов для дуг. Параметр **ArcDirection** может принимать значения **AD_COUNTERCLOCKWISE=1** (против часовой стрелки) или **AD_CLOCKWISE=2** (по часовой стрелке).

Текст

Прежде, чем выводить текст, в контекст устройства нужно выбрать подходящий шрифт (см. раздел «Шрифты», стр. 64). Далее нужно установить текущее выравнивание текста. Это можно сделать при помощи функции

UINT SetTextAlign(*контекст, выравнивание*);

Параметр ***выравнивание*** является суммой констант из табл. 31.

Табл. 30. Константы выравнивания текста

Выравнивание	*Значение*	*Положение точки выравнивания*
TA_BASELINE	24	на базовой линии
TA_BOTTOM	8	на нижней линии шрифта
TA_TOP	0	на верхней линии шрифта
TA_CENTER	6	на середине надписи
TA_LEFT	0	слева от надписи
TA_RIGHT	2	справа от надписи
TA_NOUPDATECP	0	текущая позиция не обновляется
TA_UPDATECP	1	текущая позиция обновляется

Должно быть заданы три константы — положение точки выравнивания по вертикали, по горизонтали, а также способ обновления текущей точки. Точка выравнивания — та, от которой растекается текст. Если она расположена на верхней линии шрифта, текст будет располагаться ниже, если на правой стороне выводимой надписи, то текст дополнительно будет растекаться влево от нее. Положение точки выравнивания получается на пересечении соответствующих линий (рис. 57). *По умолчанию* точка выравнивания находится слева и вверху, обновление текущей позиции не происходит.

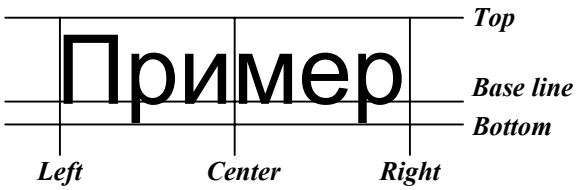


Рис. 57. Линии выравнивания текста

Цвет надписи задается при помощи функции

COLORREF SetTextColor(*контекст, цвет*);

Фон надписи устанавливается функцией

COLORREF SetBkColor(*контекст, цвет*);

Обе функции возвращают цвет, который был установлен до вызова.

Цвет фона выводится, если установлен режим фона **OPAQUE** (см. стр. 55).

Следующая функция ***выводит текст***

BOOL TextOut(*контекст, x, y, строка, длина_строки*);

Здесь ***x*** и ***y*** — позиция точки выравнивания (*абсолютное позиционирование*), ***строка*** — литерал или указатель на буфер текста (завершающий нулевой символ не требуется, так как число выводимых символов указывается прямо), ***длина_строки*** — количество символов в строке. Позиция текста игнорируется, если установлен режим обновления текущей точки. В этом случае текст выводится от текущей точки (*относительное позиционирование*).

В примере выводится нескольких строчек текста с использованием абсолютного и относительного позиционирования точки выравнивания:

Листинг 16. Вывод текста в разных режимах

```
hf = (HFONT)GetStockObject(ANSI_VAR_FONT);
exhf = (HFONT)SelectObject(hdc, hf);
TextOut(hdc, 100, 100,"Это пример вывода строки текста", 31);
SetTextAlign(hdc, TA_UPDATECP);
TextOut(hdc, 0, 0,"Это пример вывода строки текста", 10);
MoveToEx(hdc, 0, 30, (LPPOINT)NULL);
SetTextColor(hdc, RGB(128, 0, 0));
TextOut(hdc, 0, 0,"Это пример вывода строки текста", 17);
SetTextColor(hdc, RGB(255, 255, 0));
SetBkColor(hdc, RGB(0, 0, 128));
SetBkMode(hdc, OPAQUE);
SetTextCharacterExtra(hdc, 5);
TextOut(hdc, 0, 0,"Это пример вывода строки текста", 24);
SelectObject(hdc, exhf);
DeleteObject(hf);
```

Другие функции для вывода текста здесь не рассматриваются.

Области

Область (*region*) — это прямоугольник, многоугольник, эллипс или их комбинация. Области можно закрашивать, обводить рамкой, инвертировать, использовать для отсечения, тестировать на пересечение с прямоугольником и точкой.

Создание

Прямоугольная область создается следующими двумя функциями:

```
HRGN CreateRectRgn(x1, y1, x2, y2);
HRGN CreateRectRgnIndirect(прямоугольник);
```

Параметр ***прямоугольник*** — указатель на структуру **RECT** (стр. 115). Прямоугольная область рисуется точно так же, как и прямоугольник.

Прямоугольная область со скругленными углами создает функция

```
HRGN CreateRoundRectRgn(x1, y1, x2, y2, cx, cy);
```

Параметры ***cx*** и ***cy*** задают оси эллипса, который определяет скругления углов.

Эллиптическая область создается функцией

```
HRGN CreateEllipticRgn(x1, y1, x2, y2);
```

Здесь эллипс вписывается в заданный прямоугольник. Эллиптическую область можно также создать косвенно функцией

```
HRGN CreateEllipticRgnIndirect(прямоугольник);
```

Параметр ***прямоугольник*** — указатель на структуру **RECT** (стр. 115).

Многоугольная область создается функцией

```
HRGN CreatePolygonRgn(вершины, размер, режим_закраски);
```

Параметр ***вершины*** — вершины в виде массива структур **POINT**, ***размер*** — количество вершин. Режим закраски выбирается из табл. 14, стр. 57.

Область из нескольких многоугольников создает функция

```
HRGN CreatePolyPolygonRgn(вершины, размеры, количество, режим);
```

Здесь ***вершины*** — массив всех вершин, ***размеры*** — массив чисел `int`, содержащий количества вершин разных многоугольников, ***количество*** — число многоугольников. Многоугольники могут перекрывать друг друга.

Операции

Объединить две области можно при помощи функции

```
int CombineRgn(результат, область1, область2, режим);
```

Параметры ***результат, область1, область2*** — дескрипторы. Область ***результат*** должна существовать. Режимы объединения приведены в табл. 32, а результат объединения двух областей представлен на рис. 58. На рисунке предполагается, что прямоугольник — это первая область (***область1***).

Табл. 31. Режимы объединения областей

Режим	*Значение*	*Описание*
RGN_AND	1	Пересечение областей (рис. 58, а)
RGN_OR	2	Объединение областей (рис. 58, б)
RGN_XOR	3	Непересечение областей (рис. 58, в)
RGN_DIFF	4	Вычитание области 2 из области 1 (рис. 58, г)
RGN_COPY	5	Копирование области 1 (рис. 58, д)

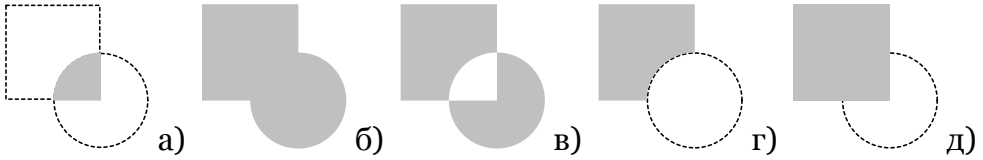


Рис. 58. Режимы объединения областей

Закрасить область можно функцией

```
BOOL FillRgn(контекст, область, кисть);
```

Здесь ***область*** — дескриптор области, ***кисть*** — дескриптор кисти.

Закрасить область текущей кистью контекста можно функцией

```
BOOL PaintRgn(контекст, область);
```

Инвертировать цвета области можно функцией

```
BOOL InvertRgn(контекст, область);
```

Рамку вокруг области рисует функция

```
BOOL FrameRgn(контекст, область, кисть, ширина, высота);
```

Здесь ***ширина*** и ***высота*** задают размер кисти.

Прямоугольник, описывающий область, вычисляет функция

```
int GetRgnBox(область, прямоугольник);
```

Здесь ***область*** — дескриптор области, а ***прямоугольник*** — указатель на структуру `RECT`. Результат функции — наименьший прямоугольник, который можно описать вокруг области.

Перемещает область на указанные смещения функция

```
int OffsetRgn(область, x_смещение, y_смещение);
```

Принадлежность точки области определяет функция

```
BOOL PtInRegion(область, x, y);
```

При помощи этой функции приложение тестирует положение курсора (мыши).

Наличие пересечения указанного прямоугольника с заданной областью определяет функция:

```
BOOL RectInRegion(область, прямоугольник);
```

Параметр ***прямоугольник*** — указатель на структуру **RECT**.

В примере используется объединение областей для того, чтобы получить изображение рис. 58, б).

Листинг 17. Объединение областей

```
void Metafile(HWND hWnd, LPCSTR path, LPCSTR desc) {
   int W = 1820, H = 1820;
   RECT rc = { 0, 0, W, H };
   hdc = CreateEnhMetaFile(NULL, "C:\\RGN_OR.emf", &rc, NULL);
   SetMapMode(hdc, MM_HIMETRIC);
   HRGN hr1 = CreateRectRgn(10, -10, 1210, -1210);
   HRGN hr2 = CreateEllipticRgn(610, -610, 1810, -1810);
   CombineRgn(hr1, hr1, hr2, RGN_OR);
   HBRUSH hb = CreateSolidBrush(RGB(192, 192, 192));
   SelectObject(hdc, hb);
   PaintRgn(hdc, hr1);
   DeleteObject(hb);
   CloseEnhMetaFile(hdc);
}
```

Пути

Путь (*path*) — это набор графических примитивов, определяющий их как единое целое. Приложения используют пути для создания сложных областей, состоящих из регулярных фигур (прямоугольник, эллипс), нерегулярных кривых (Безье), а также из произвольных отрезков и текста.

Создание

Процесс создания пути обрамляется «путевыми скобками» (*path bracket*) и состоит из трех частей:

1) функция **BeginPath**(***контекст***)

2) функции, рисующие графические примитивы

3) функция **EndPath**(***контекст***)

Здесь ***контекст*** — дескриптор контекста устройства.

Путь состоит из одной или нескольких замкнутых фигур. Замыкание отдельной фигуры должно выполняться при помощи функции

```
BOOL CloseFigure(контекст);
```

Замыкание следует выполнять даже в том случае, когда нарисованная фигура замкнута. Функция рисует отрезок от текущей точки до первой, которая задается обычно функцией **MoveToEx**. Затем функция соединяет отдельные отрезки с использованием текущего типа соединения (табл. 19). Если фигура была замкнута, функция замыкания рисует концы (табл. 18) вместо соединений.

Для рисования пути используют функции: *AngleArc**, *Arc**, *ArcTo**, *Chord**, *CloseFigure*, *Ellipse**, *ExtTextOut*, *LineTo*, *MoveToEx*, *Pie**, *PolyBezier*, *PolyBezierTo*, *PolyDraw**, *Polygon*, *Polyline*, *PolylineTo*, *PolyPolygon*, *PolyPolyline*, *Rectangle**, *RoundRect**, *TextOut*.

Звездочкой отмечены функции, недопустимые в *Windows 98, Me*.

Операции

После создания пути его можно ***закрасить*** кистью контекста:

```
BOOL FillPath(контекст);
```

При этом открытые фигуры закрываются. При закраске используется текущий графический режим *Polygon-fill* (см. раздел «Графические режимы»).

Можно ***нарисовать рамку*** замкнутого пути пером контекста:

```
BOOL StrokePath(контекст);
```

Закрасить путь и одновременно ***нарисовать рамку*** можно функцией

```
BOOL StrokeAndFillPath(контекст);
```

Преобразовать путь в область можно при помощи функции

```
BOOL PathToRegion(контекст);
```

Спрямить кривые в пути можно при помощи функции

```
BOOL FlattenPath(контекст);
```

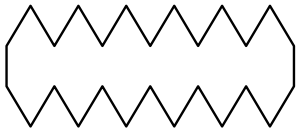
Каждая кривая при этом заменяется последовательностью отрезков прямых.

Пути, так же, как и области, используются для отсечения вывода.

Пример создания пути приведен в листинге 18.

Листинг 18. Пример создания пути

```
int h = 20, w = 12, i; const n = 6;
BeginPath(hdc);
MoveToEx(hdc, 0, h, NULL);
for (i = 0; i < n; i++) {
   LineTo(hdc, (2 * i + 1)*w, 0);
   LineTo(hdc, (2 * i + 2)*w, h);
}
LineTo(hdc, 2 * n * w, 2 * h);
for (i = n - 1; i >= 0; i--) {
   LineTo(hdc, (2 * i + 1) * w, 3 * h);
   LineTo(hdc, (2 * i) * w, 2 * h);
}
LineTo(hdc, 0, h);
CloseFigure(hdc);
EndPath(hdc);
StrokePath(hdc);
```



Отсечение вывода

Отсечение вывода (*clipping*) — это ограничение пространства вывода произвольной областью, которая задается при помощи объекта *область* или объекта *путь*. Отсечение на основе объекта *область* создает функция

```
int SelectClipRgn(контекст, область);
```

Параметр ***контекст*** — контекст устройства, ***область*** — дескриптор области.

Отсечение на основе пути создает функция

```
BOOL SelectClipPath(контекст, режим);
```

Параметр режим выбирается из табл. 32. Путь — первая область объединения.

В следующем примере путь, созданный при помощи листинг 18, используется для отсечения закраски поверхности кистью из вертикальных штрихов:

Листинг 19. Отсечение вывода при помощи пути

```
( то же, что и листинг 18 )
EndPath(hdc);
SelectClipPath(hdc, RGN_COPY);
HBRUSH hb = CreateHatchBrush(HS_VERTICAL, 0);
SelectObject(hdc, hb);
PatBlt(hdc, 0, 0, w * n * 2, h * 3, PATCOPY);
```

Следующая функция определяет новую область отсечения как пересечение текущего отсечения с прямоугольной областью, задаваемой диагональными точками ***x1, y1*** и ***x2, y2***:

```
int IntersectClipRect(контекст, x1, y1, x2, y3);
```

Следующая функция исключает прямоугольник из текущего отсечения:

```
int ExcludeClipRect(контекст, x1, y1, x2, y3);
```

Для смещения области отсечения на величину ***смещение_x*** по горизонтали и ***смещение_y*** по вертикали используется функция

```
int OffsetClipRgn(контекст, смещение_x, смещение_y);
```

Для определения принадлежности точки ***x, y*** области отсечения используется функция

```
BOOL PtVisible(контекст, x, y);
```

Для определения видимости любой части прямоугольника в области отсечения используется функция

```
BOOL RectVisible(контекст, прямоугольник);
```

Параметр ***прямоугольник*** — указатель на структуру **RECT**.

Если путь описывается текстом, то область отсечения на основе этого пути находится внутри контуров глифов.

Получить прямоугольник области отсечения можно при помощи функции

```
int GetClipBox(HDC hdc, LPRECT lprc);
```

Получить объект *область*, описывающий область отсечения, можно при помощи функции

```
int GetClipRgn(HDC hdc, HRGN hrgn);
```

Закрашивание поверхности

Монотонное закрашивание

Для закрашивания поверхности контекста используется функция

```
BOOL ExtFloodFill(контекст, x, y, цвет, режим);
```

Параметры ***x*** и ***y*** задают координаты точки, расположенной внутри области закрашивания, а параметр ***цвет*** вместе с параметром ***режим*** определяют границы закрашиваемой области.

Если параметр ***режим*** равен **FLOODFILLSURFACE=1** закрашиваются все пиксели, начиная от заданной точки, цвет которых ***равен*** указанному параметром ***цвет***.

Если параметр ***режим*** равен **FLOODFILLBORDER=0**, пиксели закрашиваются до тех пор, ***пока*** их цвет ***не равен*** цвету, заданному параметром ***цвет***.

Первый режим перекрашивает заданный цвет везде, где он доступен из заданной точки, а второй закрашивает до границы заданного цвета.

Градиентное закрашивание

Для закрашивания с плавным переходом цвета внутри прямоугольной или треугольной области используется функция

```
BOOL GradientFill(
   HDC hdc,              /* контекст                  */
   PTRIVERTEX pVertex,   /* массив вершин             */
   ULONG dwNumVertex,    /* количество вершин         */
   PVOID pMesh,          /* массив градиентов         */
   ULONG dwNumMesh,      /* размер массива градиентов */
   ULONG dwMode          /* режим закраски            */
);
```

Вершины задаются при помощи структуры **TRIVERTEX**, которая описывает координаты положения и цветовые составляющие:

```
typedef struct _TRIVERTEX {
   LONG        x;
   LONG        y;
   COLOR16     Red;
   COLOR16     Green;
   COLOR16     Blue;
   COLOR16     Alpha;
} TRIVERTEX, *PTRIVERTEX;
```

Цветовые составляющие описываются типом **COLOR16**:

```
typedef USHORT COLOR16;
```

Данный тип позволяет описывать цвет с более высоким разрешением, — диапазон значений цвета равен **0x0000÷0xFF00** (*0÷65280*).

Параметр ***массив градиентов*** описывает закрашиваемую поверхность в виде треугольников или прямоугольников.

Один треугольник массива описывается структурой **GRADIENT_TRIANGLE**:

```
typedef struct _GRADIENT_TRIANGLE {
   ULONG Vertex1;
   ULONG Vertex2;
   ULONG Vertex3;
} GRADIENT_TRIANGLE, *PGRADIENT_TRIANGLE;
```

Один прямоугольник массива описывается структурой **GRADIENT_RECT**:

```
typedef struct _GRADIENT_RECT {
   ULONG UpperLeft;
   ULONG LowerRight;
} GRADIENT_RECT, *PGRADIENT_RECT;
```

Поля структур — это номера точек в массиве вершин. Прямоугольник задается своими диагональными точками.

Режим закраски выбирается из табл. 33.

Табл. 32. Режимы градиентного закрашивания

Режим	*Значение*	*Описание*
GRADIENT_FILL_RECT_H	0	Градиент по горизонтали прямоугольника
GRADIENT_FILL_RECT_V	1	Градиент по вертикали прямоугольника
GRADIENT_FILL_TRIANGLE	2	Градиент по площади треугольника

При необходимости система выполняет дизеринг.

Пример показывает, как выполнить градиентное закрашивание треугольника:

Листинг 20. Градиентное закрашивание

```
TRIVERTEX V[3];
V[0].x =   0; V[0].y =   0;
V[0].Red = 0; V[0].Green = 0x0f00; V[0].Blue = 0x0f00;
V[1].x = 100; V[1].y =   0;
V[1].Red = 0; V[1].Green = 0x0000; V[1].Blue = 0xff00;
V[2].x =  50; V[2].y = 100;
V[2].Red = 0; V[2].Green = 0x0f00; V[2].Blue = 0x7f00;
GRADIENT_TRIANGLE T = { 0, 1, 2 };
GradientFill(hdc, V, 3, &T, 1, GRADIENT_FILL_TRIANGLE);
```

Рисование мышью

Чтобы рисовать мышью, нужно обрабатывать ее сообщения в процедуре окна *WndProc*. В событии нажатия левой кнопки мыши **WM_LBUTTONDOWN** нужно установить признак нажатия и запомнить текущие координаты. Далее каждое движение мышью будет вызывать событие **WM_MOUSEMOVE**, в котором проверяется установленный флаг, и если он установлен, то рисуется отрезок от запомненной ранее точки до текущего положения мыши. Когда кнопка мыши отпускается, возникает событие **WM_LBUTTONUP**, дорисовывается последний отрезок и снимается флаг.

Листинг 21. Рисование мышью

```
static BOOL pMouseDown = FALSE;
static POINT pred_point;
   . . .
   case WM_LBUTTONDOWN:
      pMouseDown = TRUE;
      pred_point.x = LOWORD(lParam);
      pred_point.y = HIWORD(lParam);
              break;
   case WM_MOUSEMOVE:
      if (pMouseDown) {
         MoveToEx(hGDDC, pred_point.x, pred_point.y, NULL);
         pred_point.x = LOWORD(lParam);
         pred_point.y = HIWORD(lParam);
         LineTo(hGDDC, pred_point.x, pred_point.y);
      }
      break;
   case WM_LBUTTONUP:
      if (pMouseDown) {
         MoveToEx(hGDDC, pred_point.x, pred_point.y, NULL);
         LineTo(hGDDC, LOWORD(lParam), HIWORD(lParam));
      }
      pMouseDown = FALSE;
      break;
```

В данном примере используется частный контекст устройства. Если используется общий контекст, его получают в событиях **WM_MOUSEMOVE** и **WM_LBUTTONUP** при помощи функции **GetDC** перед выполнением графических операций, и освобождают по завершении. Дополнительно см. листинг 15, стр. 86.

Рамки

Рамки (*rectangle*) предназначены для определения прямоугольных областей экрана. Они используются для многих целей, например, для задания области отсечения, области тестирования курсора, области обновления изображения, области вывода текста, скроллинга. Они полезны также для интерактивного получения от пользователя областей выделения (пользователь обводит на экране некоторую область, создается рамка, область выделяется).

Рамки *не производят никакого графического вывода*, они лишь определяют прямоугольник в виде структуры **RECT** (см. стр. 115) и позволяют выполнить простейшие действия с прямоугольниками.

Рамка ***создается*** при помощи функции

```
BOOL SetRect(прямоугольник, x1, y1, x2, y2);
```

Параметр ***прямоугольник*** — указатель на структуру **RECT**, которая заполняется значениями координат ***x1, y1*** и ***x2, y2***.

Определить рамку — ***пересечение*** двух рамок можно при помощи функции

```
BOOL IntersectRect(результат, рамка1, рамка2);
```

Все параметры — указатели на **RECT**.

Объединение двух рамок вычисляет функция

```
BOOL UnionRect(результат, рамка1, рамка2);
```

Все параметры — указатели на **RECT**. Объединенная рамка полностью включает в себя объединяемые рамки.

Обе функции часто используются для определения области обновления графического вывода.

Разность двух рамок вычисляет функция

```
BOOL SubtractRect(результат, рамка1, рамка2);
```

Все параметры — указатели на **RECT**. Вычитаемая рамка ***рамка2*** должна совпадать либо по координате ***x***, либо по координате ***y*** с рамкой ***рамка1***.

Определить ***принадлежности точки*** рамке можно при помощи функции

```
BOOL PtInRect(LPRECT прямоугольник, точка);
```

Здесь ***прямоугольник*** — указатель на **RECT**, ***точка*** — **POINT**.

Изменить размеры рамки на значения ***dx*** и ***dy*** можно при помощи функции

```
BOOL InflateRect(LPRECT прямоугольник, dx, dy);
```

Значения ***dx*** и ***dy*** — со знаком.

Переместить рамку на значения ***dx*** и ***dy*** можно при помощи функции

```
BOOL OffsetRect(LPRECT прямоугольник, dx, dy);
```

Следующая функция определяет, является рамка ***пустой*** или нет:

```
BOOL IsRectEmpty(LPRECT прямоугольник);
```

Определить ***равенство*** рамок можно при помощи функции

```
BOOL EqualRect(прямоугольник1, прямоугольник2);
```

Копию рамки можно получить при помощи функции

```
BOOL CopyRect(приемник, источник);
```

Оба параметра — указатели на **RECT**.

Преобразования

Некоторые приложения, такие, как CAD (САПР) системы, используют координатные пространства и преобразования для получения необходимого изображения. Координатное пространство — это плоскость, на которой задана декартова система координат. Положение точки на плоскости задается двумя координатами ***x*** и ***y***. Система Windows определяет следующие четыре типа координатных пространств (табл. 34).

Табл. 33. Координатные пространства Windows

Пространство	***Описание***
World мировое	Исходное пространство объекта. Его можно масштабировать, переносить, сдвигать, вращать и отражать. Размер пространства 2^32×2^32.
Page страничное	Вторичное или исходное пространство для преобразований. Устанавливает отображение логических единиц в физические. Размер пространства 2^32×2^32.
Device устройство	Следующее за страничным пространство, допускающее только перенос (коррекцию положения начальной точки). Размер пространства 2^27×2^27.
Physical device физическое	Результирующее пространство графических преобразований. Это клиентская часть окна, окно в целом, страница принтера и т. д. Размер пространства определяется устройством.

Физическое пространство недоступно для программирования, — им управляет система. Если не предпринимать никаких дополнительных действий по преобразованию координат, то координаты — это физические пиксели устройства:

```
MoveToEx(hdc, 0, 0, NULL);
LineTo(hdc, 80, 0);
```

Отрезок длиной 80 единиц будет занимать 80 пикселей на экране и 80 точек при печати. Возникает несоответствие, — размер экранного пикселя больше, чем точка принтера. Отрезки получаются разными.

Если рассчитать разрешение устройства в мм/пиксель, можно получить необходимую длину отрезка в физических пикселях, которая получится разной для разных устройств. Функция **`GetDeviceCaps`** с параметром **`LOGPIXELSX`** возвращает разрешение устройства в точках/дюйм (*dpi*) по горизонтали. Поделив дюйм на разрешение *dpi*, получаем горизонтальное разрешение устройства **`HRes`** в мм/пиксель. В качестве примера будем рассматривать горизонтальный отрезок длиной 8 см = 80 мм:

```
double HRes = 25.4 / GetDeviceCaps(hdc, LOGPIXELSX);
MoveToEx(hdc, 0, 0, NULL);
LineTo(hdc, 80 / HRes, 0);
```

На моем мониторе отрезок имеет длину 86 мм. Можно также рассчитать разрешение, поделив размер устройства в миллиметрах на размер в пикселях:

```
double HRes, HS;
HS = GetDeviceCaps(hdc, HORZSIZE);
HRes = HS / GetDeviceCaps(hdc, HORZRES);
MoveToEx(hdc, 0, 0, NULL);
LineTo(hdc, 80 / HRes, 0);
```

Размер отрезка на экране возрастет (на моем мониторе 93 мм), но он не будет зависеть от разрешения экрана, как в предыдущем случае.

Чтобы понять, почему отрезок на экране имеет большую длину, чем предполагалось, нужно произвести некоторые расчеты. Когда разрешение экрана рассчитывается через размер экрана в миллиметрах и пикселях, ошибка получается из-за того, что фактический размер экрана больше, чем возвращает функция GetDeviceCaps. Речь может идти только о конкретном мониторе. Так, для моего монитора, у которого ширина рабочей поверхности равна 365 мм (причем я могу ее регулировать в небольших пределах), функция возвращает 320 мм. Разница велика, но она постоянна при разных режимах монитора.

*Когда разрешение мм/пиксель рассчитывается исходя из разрешения dpi, ошибка тоже возникает, потому что размер логического пикселя — величина неопределенная. Для всех мониторов, независимо от размера и режима работы, система устанавливает разрешение в 96 dpi. Однако для моего монитора при указанной ширине в 365 мм и разрешении по горизонтали 1280 пикселей фактическое разрешение равно 1280*25,4/365 = 89 dpi. Причем эта величина зависит от текущего режима монитора — чем меньше разрешение в пикселях, тем меньше dpi и больше неточность воспроизведения длины. Нетрудно подсчитать, что разрешение 96 dpi соответствует режиму, при котором разрешение в пикселях составляет 1380.*

Каждый, кто работал в редакторе Word, знает, что при масштабе 100% фактический размер листа бумаги на экране больше, чем есть на самом деле. И чем меньше разрешение в пикселях, тем больше неточность. Соотношение 89/96 (dpi/dpi) дает 0,93, что соответствует 93%. Устанавливая масштаб 93% в редакторе Word я получаю на своем мониторе точное соответствие между размерами листа на экране и бумаге. С другой стороны, если установить разрешение монитора в 89 dpi, используя диалог свойств монитора, я получаю то же самое, и, кроме того, отрезок длиной 80 мм имеет длину 80 мм. Правда, при этом размеры шрифтов на экране уменьшаются соответствующим образом.

Несоответствие размеров на экране действительным, видимо, неизбежно, так как изображение на экране монитора кажется меньшим, чем есть на самом деле. Этому способствует и низкая разрешающая способность дисплея. В MSDN указывается, что размер логического пикселя может на 40% отличаться от действительного. Большие отклонения соответствуют меньшим разрешениям в пикселях.

Вместо того, чтобы пересчитывать размеры в пиксели, мы можем установить *устройство-независимую систему координат — страничное пространство*. Есть 6 предопределенных страничных пространств, приведенных в табл. 13, стр. 56. Страничное пространство устанавливает функция **SetMapMode**. Например, в режиме **MM_LOMETRIC** одна логическая единица равна 0,1 мм = 0,01 см. Тогда длина отрезка в логических единицах будет равна 800:

```
SetMapMode(hdc, MM_LOMETRIC);
MoveToEx(hdc, 0, 0, NULL);
LineTo(hdc, 800, 0);
```

Теперь мы имеем два вида координат — логические в страничном пространстве, и физические на устройстве. Пересчет логических координат в физические берет на себя система. Страничное пространство дает возможность задавать координаты в независимых единицах измерения. При необходимости можно получить физические координаты, если известны логические, при помощи функции

```
BOOL LPtoDP(HDC hdc, LPPOINT lpPoints, int nCount);
```

Здесь второй параметр — это указатель на массив логических координат, а третий — размер массива. Обратное преобразование, из физических координат в логические, выполняет функция

```
BOOL DPtoLP(HDC hdc, LPPOINT lpPoints, int nCount);
```

Эту функцию можно использовать, например, чтобы определить, каковы размер области в логических единицах:

```
RECT rc;
GetClientRect(hWnd, &rc);
SetMapMode(hdc, MM_HIMETRIC);
DPtoLP(hdc, (LPPOINT)&rc, 2);
```

Здесь прямоугольник `rc` после преобразования будет содержать размеры окна в сотых долях миллиметра.

Страничные пространства устанавливают не только соотношение между физическими и логическими пикселями, но и направление осей координат. Предопределенные страничные пространства, за исключением **`MM_TEXT`**, разворачивают ось *y* вверх, при этом начало координат находится в левом верхнем углу. Поэтому вертикальный отрезок в страничном пространстве рисуется *с отрицательными координатами*:

```
SetMapMode(hdc, MM_LOMETRIC);
MoveToEx(hdc, 0, 0, NULL);
LineTo(hdc, 0, -800);
```

Если предопределенные страничные пространства не подходят для целей приложения, имеется возможность установить произвольное пространство, в котором единицы измерения имеют необходимый размер.

Произвольное пространство может быть изотропным и анизотропным. В ***изотропном*** пространстве координатные оси имеют одинаковый масштаб, в ***анизотропном*** — разный. Изотропное пространство больше подходит для CAD (САПР) систем, а анизотропное — для отображения графиков функций.

Соотношение между физическими и логическими координатами в этих пространствах устанавливается при помощи двух функций. Первая функция устанавливает размер окна вывода в логических единицах:

```
BOOL SetWindowExtEx(контекст, ширина, высота, экс-размер);
```

Вторая функция устанавливает размер порта вывода в физических единицах:

```
BOOL SetViewportExtEx(контекст, ширина, высота, экс-размер);
```

В качестве примера рассмотрим изотропное пространство, в котором единицы измерения — миллиметры. Выводить изображение будем на экран. Сначала нужно рассчитать разрешение экрана в мм/пиксель, а затем определить размеры окна вывода в миллиметрах и размеры порта вывода в пикселях:

```
double HRes, HS, VRes, VS;
HS = GetDeviceCaps(hdc, HORZSIZE);
HRes = HS / GetDeviceCaps(hdc, HORZRES);
VS = GetDeviceCaps(hdc, VERTSIZE);
VRes = VS / GetDeviceCaps(hdc, VERTRES);
RECT rc;
GetClientRect(hWnd, &rc);
SetMapMode(hdc, MM_ISOTROPIC);
SetWindowExtEx(hdc, rc.right * HRes, rc.bottom * VRes, NULL);
SetViewportExtEx(hdc, rc.right, rc.bottom, NULL);
MoveToEx(hdc, 0, 0, NULL);
LineTo(hdc, 80, 0);
LineTo(hdc, 80, 80);
```

Здесь рисуется два отрезка длиной 80 мм, один горизонтальный, другой вертикальный. Пример анизотропного пространства см. листинг 14, стр. 85.

Следует иметь в виду, что как изотропное, так и анизотропное пространство устанавливаются посредством соотношений целых чисел. Это значит, что нельзя установить такое пространство, размер которого меньше единицы. Например, если для окна шириной 400 пикселей установить единицы измерения *метры*, то размер окна в логических единицах будет равен 0,01 м, и задать его не удастся. Кроме того, при больших единицах измерения начнет сказываться потеря точности из-за округления. Поэтому необходимо принимать меры для подбора таких единиц, которые сохранят необходимые точность и диапазон.

С этой точки зрения пространство **MM_HIMETRIC** является одним из лучших для метрической системы. Его точность (0,01 мм) превышает точность лазерного принтера с разрешением 600 *dpi* в четыре раза. Это позволяет с большой точностью задавать ширину линий для печати. Введя коэффициенты пересчета из сотых долей миллиметра в другие единицы, можно сравнительно просто решить задачу отображения самых разных изотропных пространств.

Кроме определения того или иного масштаба координатных осей, можно также установить ***положение центра*** системы координат. Следующая функция устанавливает, какая точка окна вывода будет начальной:

```
BOOL SetWindowOrgEx(HDC hdc, int X, int Y, LPPOINT lpPoint);
```

При этом устанавливаются координаты левой верхней точки на координатной плоскости, а не координаты центра системы. В следующем примере начало координат устанавливается на 10 логических единиц ниже верхней кромки окна и на 10 единиц правее:

```
   SetWindowOrgEx(hdc, -10, -10, NULL);
```

Аналогично может быть установлена точка начала для порта вывода:

```
BOOL SetViewportOrgEx(HDC hdc, int X, int Y, LPPOINT lpPoint);
```

Пространство мировых координат возникает, когда используются преобразования координат с целью поворота, переноса, растяжения, сдвига или отражения. Для преобразований используется матрица двумерных однородных координат размером *3×3*, в которой третий столбец (проекции) фиксирован. Дополнительно о преобразованиях см. учебное пособие «*Машинная графика*».

Матрица преобразования задается при помощи структуры **XFORM** (в комментариях приведены обозначения элементов матрицы согласно описанию матриц преобразования в учебном пособии «*Машинная графика*»):

```
typedef struct _XFORM {
   FLOAT eM11;     /* a */
   FLOAT eM12;     /* b */
   FLOAT eM21;     /* c */
   FLOAT eM22;     /* d */
   FLOAT eDx;      /* m */
   FLOAT eDy;      /* n */
} XFORM, *PXFORM;
```

Рис. 59. Матрица преобразования

Для выполнения преобразования нужно заполнить матрицу подходящими значениями мировых координат, и передать указатель на нее в вызов функции

```
BOOL SetWorldTransform(контекст, матрица);
```

Все последующие за вызовом этой функции графические примитивы будут подвергнуты преобразованию.

Преобразования возможны не на всех устройствах и только в *Windows* старше *Millennium*. В исходном состоянии графический вывод не допускает преобразования. Чтобы их разрешить, нужно использовать функцию

```
int SetGraphicsMode(контекст, режим);
```

Здесь ***контекст*** — контекст устройства, для которого устанавливаются преобразования, а ***режим*** — константа `GM_ADVANCED=2` для разрешения преобразований или `GM_COMPATIBLE=1` для запрещения. Вернуться в режим `GM_COMPATIBLE` после перехода в режим преобразований возможно, только если матрица преобразований *единичная*.

Матрица преобразования выполняет пересчет мировых координат в страничные. И мировые и страничные координаты являются логическими.

В примере выполняется поворот эллипса на 90° и одновременно его перенос в область рисования (при повороте эллипс выходит за границу окна).

Листинг 22. Преобразование эллипса

```
SetGraphicsMode(hdc, GM_ADVANCED);
XFORM xf = { 0, 1, -1, 0, 97, 0 };
SetWorldTransform(hdc, &xf);
Ellipse(hdc, 0, 0, 49, 97);
```

Несколько матриц преобразования могут быть преобразованы в одну при помощи функции

```
BOOL CombineTransform(результат, матрица1, матрица2);
```

Все параметры — указатели на `XFORM`. Результирующая матрица равна произведению матрицы 1 на матрицу 2. Изменить текущую матрицу преобразования можно при помощи функции

```
BOOL ModifyWorldTransform(контекст, матрица, режим);
```

Параметр ***матрица*** задает матрицу преобразования, которая будет комбинироваться с текущей матрицей в соответствии с режимом. Параметр ***режим*** выбирается из табл. 35.

Табл. 34. Режимы преобразований

Режим	*Значение*	*Описание*
MWT_IDENTITY	1	Результат — единичная матрица преобразования.
MWT_LEFTMULTIPLY	2	Текущая матрица — правая, заданная — левая.
MWT_RIGHTMULTIPLY	3	Текущая матрица — левая, заданная — правая.

Если задано единичное преобразование, заданная матрица игнорируется.

Метафайлы

Метафайл — это запись команд графического ядра GDI, которые «проигрываются» при помощи функции `PlayEnhMetaFile`. Различают метафайл *Windows*, использовавшийся на платформе *Win16*, и улучшенный метафайл (*EMF*), используемый на платформе *Win32*. Улучшенный метафайл (далее просто *метафайл*) состоит из массива записей, которые образуют следующие разделы: заголовок, необязательная таблица дескрипторов графических объектов, необязательная палитра, список команд для воспроизведения — мета-записи.

Первая запись — это заголовок, структура `ENHMETAHEADER`. Он содержит размер файла, размер картинки в единицах устройства, размер картинки в сотых долях

миллиметра, число мета-записей, число цветов в палитре, смещение к необязательной строке описания, разрешение устройства, создавшего метафайл в пикселях и миллиметрах, и некоторую дополнительную информацию.

Строка описания, если она есть, располагается после заголовка. Она предназначена для указания названия программы и названия файла. Строка описания должна заканчиваться двумя нулевыми символами, а одиночный нулевой символ разделяет отдельные строки описания.

Мета-записи идентифицируют функции, которые следует вызывать для вывода метафайла, и, если функции имеют параметры, то и сами эти параметры. Число различных мета-записей в современной Windows равно 120, они пронумерованы числами от 1 до 120, каждому номеру сопоставлена константа. Константы определены в файле *wingdi.h*. Например, функции **Ellipse** сопоставлена константа **EMR_ELLIPSE**, равная 42. Так как разные функции требуют разное число параметров, мета-записи имеют различный размер. Реализованы практически все графические функции.

При создании метафайла создается контекст метафайла, одновременно метафайл создается на диске, поэтому во время создания требуется знать допустимое имя файла. После создания контекста метафайла в него можно выводить любые графические примитивы, устанавливать режимы, выполнять преобразования, закраску, создавать перья и кисти, области и пути, палитру и строку описания. После завершения операций метафайл закрывается, при этом он записывается на диск и создается контекст этого файла, который можно использовать для «проигрывания» метафайла.

Метафайл создается при помощи функции

```
HDC CreateEnhMetaFile(контекст, путь, прямоугольник, строка_описания);
```

Здесь ***контекст*** — контекст устройства, для которого создается метафайл. Обычно это контекст окна или **NULL**, чтобы использовать контекст дисплея. Параметр ***путь*** указывает на файл метафайла. Если его принять равным **NULL**, метафайл будет создаваться только в памяти. Размер области метафайла ограничивается параметром ***прямоугольник***, который задается в *сотых долях миллиметра*. Если размер области метафайла задан **NULL**, она вычисляется автоматически по размеру изображения. Функция возвращает контекст метафайла.

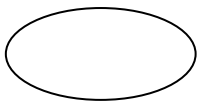
Контекст метафайла закрывает функция

```
HENHMETAFILE CloseEnhMetaFile(контекст_метафайла);
```

В качестве примера рассмотрим простейший метафайл, состоящий из одного эллипса (листинг 23). Размер области метафайла здесь вычисляется автоматически, так как она не задана.

Листинг 23. Пример простого метафайла

```
void Metafile(HWND hWnd, LPCSTR path, LPCSTR desc) {
   HDC hdc = CreateEnhMetaFile(NULL, path, NULL, NULL);
   Ellipse(hdc, 0, 0, 97, 49);
   CloseEnhMetaFile(hdc);
}
```



Результат работы функции — файл *C:\test.emf*, который вставлен в данный документ, чтобы показать результат (имя файла задано при вызове функции).

Особенность метафайлов (и не только метафайлов) — ***независимость от устройства***. Есть два подхода к получению точности изображения.

Первый заключается в использовании страничного пространства. Установив пространство **MM_HIMETRIC**, можно с точностью 0,01 мм описывать изображение, включая толщину линий. Результат вывода метафайла на устройство будет определяться только разрешающей способностью устройства. В качестве примера метафайл, рисующий эллипс размером 50×30 мм:

Листинг 24. Метафайл заданного размера

```
void Metafile(HWND hWnd, LPCSTR path, LPCSTR desc) {
   HDC hdc;
   int W = 5000, H = 3000;
   RECT rc = { 0, 0, W, H };
   hdc = CreateEnhMetaFile(NULL, path, &rc, NULL);
   SetMapMode(hdc, MM_HIMETRIC);
   HPEN hp = CreatePen(PS_SOLID | PS_INSIDEFRAME, 50, 0);
   SelectObject(hdc, hp);
   Ellipse(hdc, 0, 0, W, -H);
   CloseEnhMetaFile(hdc);
}
```

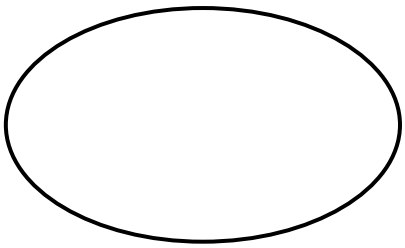


Рис. 60. Эллипс размером 50×30 мм (уменьшено в 1,4 раза)

Фактический размер картинки в *Word* равен 52,9×31,8 мм, но это особенность *Word*. *PictureManager Windows XP* показывает размер в пикселях 201×121, что соответствует размеру эллипса 50×30 мм (разрешение равно 0,25 мм/пиксель).

Второй подход заключается в использовании физических единиц. В этом случае размеры вычисляются делением на разрешение в мм/пиксель. Пример:

Листинг 25. Метафайл с областью в пикселях

```
void Metafile(HWND hWnd, LPCSTR path, LPCSTR desc) {
   HDC hdc; int HRes, VRes, int W = 5000, H = 3000;
   hdc = GetDC(hWnd);
   HRes = 100*GetDeviceCaps(hdc,HORZSIZE)/GetDeviceCaps(hdc,HORZRES);
   VRes = 100*GetDeviceCaps(hdc,VERTSIZE)/GetDeviceCaps(hdc,VERTRES);
   ReleaseDC(hWnd, hdc);
   RECT rc = { 0, 0, W, H };
   hdc = CreateEnhMetaFile(NULL, "C:\\E2.emf", &rc, NULL);
   HPEN hp = CreatePen(PS_SOLID | PS_INSIDEFRAME, 50/HRes, 0);
   SelectObject(hdc, hp);
   Ellipse(hdc, 0, 0, W/HRes, H/VRes);
   CloseEnhMetaFile(hdc);
}
```

Недостаток данного подхода заключается в округлении единиц до целых чисел. Например, толщина линии может принимать значения только 1, 2, 3..., что соответствует логическим размерам 25, 50, 75...

Контекст метафайла можно использовать для «*проигрывания*» метафайла на любом устройстве. Контекст метафайла возвращают функции **CloseEnhMetaFile** и **GetEnhMetaFile(*путь*)**. В следующем примере показано, как использовать контекст метафайла:

```
void Metafile(HWND hWnd, LPCSTR path, LPCSTR desc) {
   HDC hdc = CreateEnhMetaFile(NULL, path, NULL, NULL);
   Ellipse(hdc, 0, 0, 37, 25);
   HENHMETAFILE hemf = CloseEnhMetaFile(hdc);
   hdc = GetDC(hWnd);
   RECT rc = { 0, 0, 200, 300 };
   PlayEnhMetaFile(hdc, hemf, &rc);
   DeleteEnhMetaFile(hemf);
   ReleaseDC(hWnd, hdc);
}
```

Контекст метафайла удаляется функцией **DeleteEnhMetaFile**.

Преимуществом метафайла является высокая точность отображения. Однако при этом он проигрывает в скорости вывода пиксельному набору. Метафайл, являясь записью команд, программой, может содержать не лучшую их последовательность, а ошибки в метафайле могут привести к нарушению нормальной работы приложения. Еще один недостаток метафайла — не всегда точное определение границ рисунка. К тому же разные версии *Word* отображают одни и те же метафайлы по-разному.

Контрольные вопросы

1. Что описывает контекст устройства?
2. Какие объекты включаются в контекст устройства?
3. В чем разница между логическими и физическими характеристиками?
4. Что регулируют графические режимы контекста устройства?
6. Что такое стандартные (*stock*) объекты?
7. В чем назначение перьев? Какие виды, типы перьев вы знаете?
8. В чем назначение кистей? Какие виды кистей вы знаете?
9. Как кодируется цвет? Что такое палитра?
10. Как выполняется переход от логического цвета к физическому?
11. Назовите характеристики шрифта.
12. Что такое область и чем она отличается от пути?
13. Что называется пиксельным набором (*bitmap*) ?
14. Чем различаются *dib* и *ddb*-картинки?
15. Как устроен метафайл? В чем его преимущества и недостатки?

Литература

1. MSDN Library, October 2001.
2. Блиндер Е. М., Фурман С. Л. Телевидение: учебник для ПТУ. — М.: Радио и связь, 1984 —272 с., ил.
3. Киселев Ф. Я., Виленский Ю. Б. Физические и химические основы цветной фотографии: Справ. изд. — Л.: Химия, 1988. — 304 с., ил.
4. Кривошеев М. И., Кустарев А. К. Цветовые измерения. — М.: Энергоатомиздат, 1990. — 240 с.: ил.
5. Луизов А. В. Глаз и свет. Л.: Энергоатомиздат. Ленингр. отд-ние, 1983. — 144 с., ил.
6. Луизов А. В. Цвет и свет. — Л.: Энергоатомиздат. Ленингр. отд-ние, 1989. — 256 с.: ил.
7. Новаковский С. В. Цвет в цветном телевидении. — М.: Радио и связь, 1988. — 288 с.: ил.
8. В. Пономарев. Машинная графика. Учебное пособие. Озерск: ОТИ МИФИ, 2005. — 70 с.: ил.
9. В. Ю. Романов. Популярные форматы файлов для хранения графических изображений на IBM PC. — М.: Унитех, 1992. — 156 с.: ил.
10. Порев В. Н. Компьютерная графика. — СПб.: БХВ-Петербург, 2002. — 432 с.: ил.
11. Самойлов В. Ф., Хромой Б. П. Основы цветного телевидения. — М.: Радио и связь, 1982. (Массовая радиобиблиотека; Вып. 1047) — 160с., ил.
12. Том Сван. Форматы файлов Windows. Пер. с англ. —М.: БИНОМ, 1994 — 288 с.: ил.

Приложения

Системные цвета Windows

Цвет	*Знач.*	*Описание*
COLOR_SCROLLBAR	0	Линейка прокрутки.
COLOR_BACKGROUND, COLOR_DESKTOP	1	Рабочий стол.
COLOR_ACTIVECAPTION	2	Заголовок активного окна. Левый цвет в градиентной заливке.
COLOR_INACTIVECAPTION	3	Пассивный заголовок. Левый цвет в градиентной заливке.
COLOR_MENU	4	Меню.
COLOR_WINDOW	5	Фон окна.
COLOR_WINDOWFRAME	6	Рамка окна.
COLOR_MENUTEXT	7	Текст меню.
COLOR_WINDOWTEXT	8	Текст окна.
COLOR_CAPTIONTEXT	9	Текст надписи. Цвет стрелки прокрутки.
COLOR_ACTIVEBORDER	10	Граница активного окна.
COLOR_INACTIVEBORDER	11	Граница пассивного окна.
COLOR_APPWORKSPACE	12	Фон рабочей области окна MDI.
COLOR_HIGHLIGHT	13	Подсветка выбора.
COLOR_HIGHLIGHTTEXT	14	Текст подсветки выбора.
COLOR_BTNFACE, COLOR_3DFACE	15	Цвет кнопки и диалогового окна.
COLOR_BTNSHADOW, COLOR_3DSHADOW	16	Неосвещенное ребро элемента.
COLOR_GRAYTEXT	17	Серый (недоступный) текст.
COLOR_BTNTEXT	18	Текст кнопки.
COLOR_INACTIVECAPTIONTEXT	19	Текст пассивного заголовка.
COLOR_BTNHIGHLIGHT, COLOR_BTNHILIGHT, COLOR_3DHILIGHT, COLOR_3DHIGHLIGHT	20	Светлое освещенное ребро элемента.
COLOR_3DDKSHADOW	21	Тень (dark shadow).
COLOR_3DLIGHT	22	Освещенное ребро элемента.
COLOR_INFOTEXT	23	Цвет подсказки.
COLOR_INFOBK	24	Фон подсказки.
COLOR_HOTLIGHT	26	Подсветка активного элемента при включенной активации одним щелчком.
COLOR_GRADIENTACTIVECAPTION	27	Правый цвет в градиентной заливке активного окна[2].
COLOR_GRADIENTINACTIVECAPTION	28	Правый цвет в градиентной заливке пассивного окна[2].

Примечания

1. Цвета 0—20 определены в *Windows* версии ниже 4.00. Цвета 21—24 определены в версии 4.00. Цвета 26—28 определены в версии 5.00. Константы определены в **`winuser.h`**.
2. Для определения возможности градиентной заливки используется функция **`SystemParametersInfo`** с параметром **`SPI_GETGRADIENTCAPTIONS`**.

Стандартные объекты Windows

Стандартные объекты находятся в фонде (***stock***).

Объект	***Знач.***	***Описание***
WHITE_BRUSH	0	Белая кисть.
LTGRAY_BRUSH	1	Светло-серая кисть.
GRAY_BRUSH	2	Серая кисть.
DKGRAY_BRUSH	3	Темно-серая кисть.
BLACK_BRUSH	4	Черная кисть.
HOLLOW_BRUSH	5	Пустая кисть (эквив. NULL_BRUSH).
NULL_BRUSH	5	Нулевая кисть (эквив. HOLLOW_BRUSH).
WHITE_PEN	6	Белое перо.
BLACK_PEN	7	Черное перо.
OEM_FIXED_FONT	10	Зависимый от производителя (OEM) моноширинный шрифт
ANSI_FIXED_FONT	11	Моноширинный системный шрифт Windows.
ANSI_VAR_FONT	12	Пропорциональный системный шрифт Windows.
SYSTEM_FONT	13	Системный шрифт. Используется по умолчанию для меню, диалогов и текста. Windows 95/98 и NT: MS Sans Serif; 2000: Tahoma
DEVICE_DEFAULT_FONT	14	Windows NT/2000: Независимый от устройства шрифт.
DEFAULT_PALETTE	15	Палитра по умолчанию.
SYSTEM_FIXED_FONT	16	Моноширинный системный шрифт для совместимости с Windows версии ниже 3.0.
DEFAULT_GUI_FONT	17	Шрифт по умолчанию для меню и диалогов пользователя — MS Sans Serif.
DC_BRUSH	18	Windows 98, 2000: Сплошная цветная кисть[1].
DC_PEN	19	Windows 98, 2000: Сплошное цветное перо[2].

Примечания

1. Цвет по умолчанию белый. Его можно изменить при помощи функции **`SetDCBrushColor`**.
2. Цвет по умолчанию черный. Его можно изменить при помощи функции **`SetDCPenColor`**.
3. Константы определены в **`wingdi.h`**.

Характеристики устройств

Используются функцией **`GetDeviceCaps`**.

Константа	***Значение***	***Описание***
DRIVERVERSION	0	Версия драйвера.
TECHNOLOGY	2	Технология вывода изображения. *
HORZSIZE	4	Ширина в мм.
VERTSIZE	6	Высота в мм.
HORZRES	8	Ширина в пикселях.
VERTRES	10	Высота в пикселях.
BITSPIXEL	12	Количество бит, отводимое на пиксель.

Константа	*Значение*	*Описание*
PLANES	14	Количество битовых плоскостей.
NUMBRUSHES	16	Количество кистей.
NUMPENS	18	Количество перьев.
NUMFONTS	22	Количество шрифтов.
NUMCOLORS	24	Количество цветов. –1, если больше 256.
LINECAPS	26	Возможности по рисованию прямых. *
CURVECAPS	28	Возможности по рисованию кривых. *
POLYGONALCAPS	32	Возможности по выводу многоугольников. *
TEXTCAPS	34	Возможности по выводу текста. *
CLIPCAPS	36	Возможность отсечения вывода.
RASTERCAPS	38	Возможности по работе с картинками. *
ASPECTX	40	Относительная ширина пикселя.
ASPECTY	42	Относительная высота пикселя.
LOGPIXELSX, LOGPIXELSY	88	Число пикселей в логическом дюйме, dpi.
SIZEPALETTE	104	Размер системной палитры.
NUMRESERVED	106	Зарезервированные элементы палитры.
COLORRES	108	Текущая глубина цвета, бит/пиксель.
PHYSICALWIDTH	110	Число пикселей по ширине (принтер).
PHYSICALHEIGHT	111	Число пикселей по высоте (принтер).
PHYSICALOFFSETX	112	Левое поле в пикселях (принтер).
PHYSICALOFFSETY	113	Верхнее поле в пикселях (принтер).
VREFRESH	116	Частота кадровой развертки (дисплей).

Характеристики, отмеченные звездочкой, возвращаются в виде битовых полей, в которых отдельные биты описывают те или иные возможности устройства.

Растровые характеристики

Возвращаются функцией **GetDeviceCaps** с параметром **RASTERCAPS.**

Объект	*Значение*	*Описание*
RC_BITBLT	1	Поддерживает BitBlt.
RC_SCALING	4	Поддерживает масштабирование.
RC_PALETTE	0x0100	Поддерживает палитру.
RC_DIBTODEV	0x0200	Поддерживает SetDIBitsToDevice.
RC_STRETCHBLT	0x0800	Поддерживает StretchBlt.
RC_FLOODFILL	0x1000	Поддерживает закрашивание.
RC_STRETCHDIB	0x2000	Поддерживает StretchDIBits.
RC_DI_BITMAP	0x8000	Поддерживает SetDIBits и GetDIBits.

Основные структуры Windows

POINT

```
typedef struct tagPOINT {
   LONG x;
   LONG y;
} POINT, *PPOINT;
```

Описывает точку на плоскости.

RECT

```
typedef struct _RECT {
   LONG left;
   LONG top;
   LONG right;
   LONG bottom;
} RECT, *PRECT;
```

Описывает прямоугольную область. Задается координатами диагональных точек — левой верхней и правой нижней. Особенность — сама правая нижняя точка исключается (равно как и нижняя растровая линия и правый столбец).

SIZE

```
typedef struct tagSIZE {
   LONG cx;
   LONG cy;
} SIZE, *PSIZE;
```

Описывает размер объекта.

LOGPEN

```
typedef struct tagLOGPEN {
    UINT     lopnStyle;     /* табл. 17 на стр. 59 */
    POINT    lopnWidth;     /* ширина */
    COLORREF lopnColor;     /* цвет   */
} LOGPEN, *PLOGPEN;
```

Описывает косметическое логическое перо.
Используется функцией `CreatePenIndirect`. В поле `lopnWidth` поле ***x*** указывает на ширину, поле ***y*** не используется.

EXTLOGPEN

```
typedef struct tagEXTLOGPEN {
    DWORD     elpPenStyle;      /* табл. 17 на стр. 59 */
    DWORD     elpWidth;         /* ширина              */
    UINT      elpBrushStyle;    /* табл. 36            */
    COLORREF  elpColor;         /* цвет                */
    ULONG_PTR elpHatch;         /* табл. 22 на стр. 61 */
    DWORD     elpNumEntries;
    DWORD     elpStyleEntry[1];
} EXTLOGPEN, *PEXTLOGPEN;
```

Описывает геометрическое перо. Используется функцией `ExtCreatePen`.
Поле `elpStyleEntry` — указатель на массив, описывающий пользовательский стиль линии в формате: ***длина_штриха1, длина_промежутка1, длина_штриха2, длина_промежутка2...*** Поле `elpNumEntries` указывает количество элементов в массиве. Пример см. на стр. 58.

LOGBRUSH

```
typedef struct tagLOGBRUSH {
   UINT     lbStyle;          /* табл. 36 */
   COLORREF lbColor;
   LONG     lbHatch;
} LOGBRUSH, *PLOGBRUSH;
```

Описывает логическую кисть. Используется функцией *CreateBrushIndirect*.

Табл. 35. Стили кистей

Стиль	Значение	*Описание*
BS_SOLID	0	Сплошная кисть
BS_HOLLOW, BS_NULL	1	Нулевая (пустая) кисть
BS_HATCHED	2	Кисть со стандартной штриховкой
BS_PATTERN	3	Кисть с узором на основе картинки в памяти
BS_DIBPATTERN	5	Узор задает дескриптор картинки
BS_DIBPATTERNPT	6	Узор задает указатель на картинку

Поле **lbColor** задает цвет одноцветной кисти или цвет штриховки.

Поле **lbHatch** определяет узор. Для стиля **BS_SOLID** поле игнорируется. Для стиля **BS_HATCHED** поле **lbHatch** выбирается из табл. 22, стр. 61. Для стиля **BS_DIBPATTERN** поле **lbHatch** содержит дескриптор картинки, созданной при помощи **CreateBitmap** или **CreateCompatibleBitmap**.

Для стиля **BS_DIBPATTERN** (предназначен для Windows 98), поле **lbHatch** определяет упакованный пиксельный набор DIB (*Packed DIB*). Чтобы его получить, нужно зарезервировать место в памяти при помощи функции **LocalAlloc** с первым параметром **LMEM_MOVEABLE**. Далее в область памяти копируется пиксельный набор, состоящий из информационного заголовка и битового массива.

Для стиля **BS_DIBPATTERNPT** (предназначен для Windows XP), поле **lbHatch** определяет указатель на упакованный DIB. Указатель возвращает функция **LocalAlloc** с первым параметром **LMEM_FIXED**.

LOGFONT

```
typedef struct tagLOGFONT {
   LONG lfHeight;                  /* высота                      */
   LONG lfWidth;                   /* средняя ширина символа      */
   LONG lfEscapement;              /* угол разворота строки       */
   LONG lfOrientation;             /* угол поворота символов      */
   LONG lfWeight;                  /* жирность                    */
   BYTE lfItalic;                  /* признак курсива             */
   BYTE lfUnderline;               /* признак подчеркивания       */
   BYTE lfStrikeOut;               /* признак перечеркивания      */
   BYTE lfCharSet;                 /* дескриптор набора символов */
   BYTE lfOutPrecision;            /* точность вывода             */
   BYTE lfClipPrecision;           /* точность отсечения          */
   BYTE lfQuality;                 /* качество                    */
   BYTE lfPitchAndFamily;          /* шаг и семейство             */
   TCHAR lfFaceName[LF_FACESIZE]; /* название гарнитуры          */
} LOGFONT, *PLOGFONT;
```

Описывает логический шрифт. **LF_FACESIZE=32**. Поле **lfEscapement** задает угол выхода базовой линии относительно оси x. Поле **lfOrientation** задает угол поворота базовой линии символа относительно оси x. Оба угла задаются в десятых долях градуса.

TEXTMETRIC

```
typedef struct tagTEXTMETRIC {
   LONG tmHeight;
   LONG tmAscent;
   LONG tmDescent;
   LONG tmInternalLeading;
   LONG tmExternalLeading;
```

```
   LONG tmAveCharWidth;     /* средняя ширина символа      */
   LONG tmMaxCharWidth;     /* максимальная ширина символа */
   LONG tmWeight;           /* вес (жирность) символа      */
   LONG tmOverhang;         /* превышение длины            */
   LONG tmDigitizedAspectX; /* разрешение устройства       */
   LONG tmDigitizedAspectY; /* разрешение устройства       */
   TCHAR tmFirstChar;       /* первый символ шрифта        */
   TCHAR tmLastChar;        /* последний символ шрифта     */
   TCHAR tmDefaultChar;     /* символ по умолчанию         */
   TCHAR tmBreakChar;       /* символ пробела (разрыва)    */
   BYTE tmItalic;           /* признак курсива             */
   BYTE tmUnderlined;       /* признак подчеркивания       */
   BYTE tmStruckOut;        /* признак перечеркивания      */
   BYTE tmPitchAndFamily;   /* шаг и семейство             */
   BYTE tmCharSet;          /* набор символов              */
} TEXTMETRIC, *PTEXTMETRIC;
```

Описывает метрические характеристики физического шрифта. Все размеры задаются в логических единицах.

Рис. 61. Метрические характеристики шрифта

На рис. 62 высота шрифта обозначена через H, она соответствует параметру *Height*. Высота равна сумме параметров *Ascent* (*A*, превышение над базовой линией) и *Descent* (*D*, занижение под базовой линией). Параметр *Internal Leading* (*I*) — это место для надстрочных знаков. Параметр *External Leading* (*E*) — базовое междустрочное расстояние.

RGBQUAD

```
typedef struct tagRGBQUAD {
   BYTE rgbBlue;
   BYTE rgbGreen;
   BYTE rgbRed;
   BYTE rgbReserved;
} RGBQUAD;
```

Описывает цвет палитры в виде цветовых составляющих RGB и, возможно, *Alpha*.

RGBTRIPLE

```
typedef struct tagRGBTRIPLE {
   BYTE rgbtBlue;
   BYTE rgbtGreen;
   BYTE rgbtRed;
} RGBTRIPLE;
```

Описывает цвет палитры в виде цветовых составляющих RGB.

LOGPALETTE

```
typedef struct tagLOGPALETTE {
   WORD         palVersion;     /* версия ОС             */
   WORD         palNumEntries;  /* количество элементов */
   PALETTEENTRY palPalEntry[n]; /* массив элементов      */
} LOGPALETTE;
```

Описывает логическую палитру.

PALETTEENTRY

```
typedef struct tagPALETTEENTRY {
   BYTE peRed;    /* содержание красного цвета */
   BYTE peGreen;  /* содержание зеленого цвета */
   BYTE peBlue;   /* содержание синего цвета   */
   BYTE peFlags;  /* использование элемента    */
} PALETTEENTRY;
```

Описывает элемент палитры. Элемент **peFlags** определяет использование элемента палитры (табл. 37).

Табл. 36. Использование элемента палитры

Константа	*Значение*	*Описание*
NULL	0	Использовать цвет обычным образом
PC_RESERVED	1	Цвет используется для анимации (будет меняться).
PC_EXPLICIT	2	Младший байт цвета является индексом цвета в палитре устройства
PC_NOCOLLAPSE	4	Цвет следует включить в системную палитру. Если это невозможно, цвет аппроксимируется.

BITMAP

```
typedef struct tagBITMAP {
   LONG   bmType;        /* тип = 0                          */
   LONG   bmWidth;       /* ширина в пикселях, > 0           */
   LONG   bmHeight;      /* высота в пикселях, > 0           */
   LONG   bmWidthBytes;  /* ширина строки в байтах, кратная 2 */
   WORD   bmPlanes;      /* количество битовых плоскостей, 1  */
   WORD   bmBitsPixel;   /* глубина цвета, бит/пиксель        */
   LPVOID bmBits;        /* битовый массив                   */
} BITMAP, *PBITMAP;
```

Описывает устройство-зависимый (*ddb*) пиксельный набор (картинку).

Формат BMP файла

Файл BMP состоит из 4-х разделов:

Раздел	*Размер, байт*
`BITMAPFILEHEADER bmfh`	14
`BITMAPINFOHEADER bmih`	40
`RGBQUAD aColors[]`	переменный
`BYTE aBitmapBits[]`	переменный

Рис. 62. Структура *dib*-формата

Заголовок файла описывается структурой

```
typedef struct tagBITMAPFILEHEADER {
   UINT bfType;
   DWORD bfSize;
   UINT bfReserved1;
   UINT bfReserved2;
   DWORD bfOffBits;
} BITMAPFILEHEADER;
```

Рис. 63. Структура заголовка *dib*-формата

Поле `bfType` должно содержать два символа `BM` (`0x42` и `0x4D`). Поле `bfSize` определяет размер файла в байтах. Поля `bfReserved1` и `bfReserved2` должны быть равны `0`. Поле `bfOffBits` — смещение к началу битового массива.

Два следующих раздела файла описываются одной структурой

```
typedef struct tagBITMAPINFO {
   BITMAPINFOHEADER bmiHeader;
   RGBQUAD          bmiColors[1];
} BITMAPINFO, *PBITMAPINFO;
```

Заголовок информационной части описывается структурой

```
typedef struct tagBITMAPINFOHEADER {
   DWORD biSize;          // размер структуры (0x28=40)
   LONG  biWidth;         // ширина и высота картинки
   LONG  biHeight;
   WORD  biPlanes;        // всегда 1
   WORD  biBitCount;      // глубина цвета, бит/пиксель
   DWORD biCompression;   // сжатие (0, 1, 2 или 3)
   DWORD biSizeImage;     // размер картинки; 0, если нет сжатия
   LONG  biXPelsPerMeter; // желаемое разрешение по X, пиксель/метр
   LONG  biYPelsPerMeter; // желаемое разрешение по Y, пиксель/метр
   DWORD biClrUsed;       // используемое число цветов
   DWORD biClrImportant;  // число важных цветов
} BITMAPINFOHEADER, *PBITMAPINFOHEADER;
```

Рис. 64. Структура информационного заголовка *dib*-формата

Желаемое разрешение используется для выбора подходящего изображения для отображения на конкретном устройстве (а не наоборот). *Используемое число цветов* показывает, как используется палитра. Если это поле равно нулю, используются все цвета. Если поле не равно нулю и глубина цвета меньше 16, то поле указывает фактическое количество используемых цветов. Если глубина цвета больше или равна 16, поле содержит размер оптимизированной палитры.

Важные цвета должны располагаться в начале палитры. Они будут отображаться наиболее точно. Если поле равно нулю, все цвета считаются важными. Поле не имеет смысла, если глубина цвета равна 24.

Поле `biBitCount` выбирается из табл. 38.

Табл. 37. Глубина цвета dib-картинки

Значение	***Описание***
0	Глубину цвета задает JPEG или PNG формат картинки.
1	Монохромная картинка, палитра состоит из 2 цветов.
4	Палитра содержит 16 цветов, битовый массив — 4-х битовые индексы.
8	Палитра содержит 256 цветов, битовый массив — байтные индексы.
16	Число цветов в картинке максимум 65536.

Значение	*Описание*
	Если biCompression = BI_RGB, палитра отсутствует, каждое слово битового массива содержит значения RGB, по 5 бит на цвет (схема 5-5-5). Если biCompression = BI_BITFIELDS, палитра состоит из 3-х двойных слов — масок для выделения цвета из индекса в битовом массиве. Маска синего цвета **0x0000001F**, зеленого **0x000003E0**, красного **0x00007C00** для схемы 5-5-5 и **0x0000001F**, **0x000007E0**, **0x0000F800** для схемы 5-6-5. Битовый массив содержит цвета в виде слов по схеме 5-5-5 или 5-6-5.
24	Палитра отсутствует. Битовый массив содержит цвета в виде троек байт синей, зеленой и красной составляющих.
32	Число цветов в картинке максимум 16 777 216. Если biCompression = BI_RGB, палитра отсутствует, каждое двойное слово битового массива содержит значения RGB, по 1 байту на цвет. Если biCompression = BI_BITFIELDS, палитра состоит из 3-х двойных слов — масок для выделения цвета из индекса в битовом массиве. Маска синего цвета **0x000000FF**, зеленого **0x0000FF00**, красного **0x00FF0000**. Битовый массив содержит цвета в виде двойных слов.

Поле **biCompression** указывает на алгоритм сжатия и выбирается из табл. 39.

Табл. 38. Алгоритмы сжатия dib-картинки

Алгоритм	*Значение*	*Описание*
BI_RGB	0	Нет сжатия.
BI_RLE8	1	8-х разрядное сжатие 256-ти цветных картинок.
BI_RLE4	2	4-х разрядное сжатие 16-ти цветных картинок.
BI_BITFIELDS	3	Нет сжатия. Таблица цветов содержит маски.
BI_JPEG		Картинка JPEG.
BI_PNG		Картинка PNG.

Палитра или таблица цветов, если она есть, располагается за информационным заголовком и состоит из минимум двух элементов типа **RGBQUAD** (старший байт структуры при этом не используется) или типа **RGBTRIPPLE**.

Далее файл содержит битовый массив, описывающий пиксели. В зависимости от глубины цвета, он содержит либо индексы цветов в палитре, либо непосредственно значения RGB.

Массив состоит из строк, выровненных по границе двойного слова. Каждая строка описывает одну строку растра. Строки располагаются в обратном порядке, — первая сверху строка растра записывается последней и наоборот.

Данная структура файла не является единственно возможной. Признаком данной структуры является 14-й байт, равный 40, а также два символа BM.

Сжатие растровых изображений

Группы нескольких подряд идущих пикселей одного цвета записываются в виде числа ***q***, обозначающего количество одинаковых пикселей, и числа ***v***, равного значению цвета этих пикселей (точнее — индексу цвета).

Формат сжатия BI_RLE8

Битовый массив состоит из блоков. Блок состоит из минимум двух байтов. Если первый (младший) байт блока равен нулю, блок описывает команду (производится выход — *escape*):

	Байты команды	*Команда*
`00`	`00`	Закончить текущую строку развертки и начать новую.
`00`	`01`	Конец данных. Закончить распаковку.
`00`	`02 xx yy`	Дельта (вектор). Перейти в точку со смещением `xx,yy` от текущей позиции. Цвета пропущенных пикселей не определяются.
`00`	`nn k1,k2,..,knn`	Абсолютный режим. `2<nn<256` и показывает количество последующих индексов, записанных без сжатия.

Если первый байт блока не равен нулю, то это ***q***, а второй байт — это ***v***.

Формат сжатия BI_RLE4

Сжатие производится аналогично методу *BI_RLE8*. Сжатые последовательности из ***q*** пар 4-х разрядных индексов ***f*** и ***g*** записываются в виде двух байт. Первый байт равен ***q***, второй ***fg***, где ***f*** — старшая тетрада, ***g*** — младшая. Если последовательные пары индексы различаются, они записываются в абсолютном режиме, — байт `00`, байт с количеством последующих индексов в абсолютном режиме, сами индексы, упакованные парами в один байт, первый байт в старшей тетраде, следующий в младшей и т. д.
Команды выхода, описанные ранее, действуют так же.

Примеры кода

Перечисление цветов устройства

Следующая функция возвращает массив цветов палитры устройства. Нулевой элемент возвращаемого массива содержит количество цветов.

Листинг 26. Перечисление цветов сплошных перьев

```
COLORREF * GetDeviceColors(HDC hdc) {
   int NC; COLORREF *aColors;
   NC = GetDeviceCaps(hdc, NUMCOLORS); /* количество цветов */
   if (NC < 0) return;
   aColors = (COLORREF*)LocalAlloc(LPTR, sizeof(COLORREF) * (NC+1));
   aColors[0] = (LONG)NC;   /* количество цветов */
   EnumObjects(hdc, OBJ_PEN, (GOBJENUMPROC)EnumPen, (LPARAM)aColors);
   aColors[0] = (LONG)NC;   /* восстанавливаем количество цветов */
}
```

Дополнительная возвратно-вызываемая функция записывает отдельный цвет:

Листинг 27. Функция для записи цвета

```
int CALLBACK EnumPen(LPVOID lp, LPARAM lpp) {
   LPLOGPEN lopn = (LPLOGPEN)lp;
   COLORREF *aColors = (COLORREF*)lpp;
   int iColor;
   if (lopn->lopnStyle == PS_SOLID) {
      /* условие завершения */
      if ((iColor = (int)aColors[0]) <= 0) return 0;
      aColors[iColor] = lopn->lopnColor;
      aColors[0]--;
   }
   return 1;
}
```