[1. Цели и задачи компьютерной графики](#_Toc294915678)

[2. Растровая и векторная визуализация изображений. Основные характеристики растровых изображений](#_Toc294915679)

[3. Аддитивная цветовая модель RGB. Кодирование цвета. Палитра](#_Toc294915680)

[4. Графические форматы. Работа с растровыми изображениями (форматы DDB, DIB, bmp-файлы)](#_Toc294915681)

[5. Понятие об инкрементных алгоритмах растеризации. Растровое представление отрезка. Вывод и реализация алгоритма](#_Toc294915682)

[6. Понятие об инкрементных алгоритмах растеризации. Построение окружности методом средней точки. Вывод и реализация алгоритма](#_Toc294915683)

[7. Стиль линии (перо). Алгоритм вывода линии](#_Toc294915684)

[8. Стиль заполнения, кисть, текстура](#_Toc294915685)

[9. Геометрические основы компьютерной графики. Системы координат (декартова, полярная, сферическая) и векторы. Скалярное и векторное произведение векторов](#_Toc294915686)

[10. Геометрические основы компьютерной графики. Уравнение прямой, проходящей через две точки (на плоскости). Параметрическое уравнение прямой. Уравнение отрезка. Задача о взаимном расположении двух отрезков](#_Toc294915687)

[11. Построение кривых. Интерполяционный полином Лагранжа. Достоинства и недостатки. Реализация](#_Toc294915688)

[12. Построение кривых. Интерполяционный кубический сплайн. Реализация](#_Toc294915689)

[13. Построение кривых. Геометрические сплайны. Кривая Безье. Геометрический алгоритм построения кривой Безье. Реализация](#_Toc294915690)

[14. Мировые и экранные координаты. Алгоритм пересчёта мировых 2D-координат в оконные. Вывод и реализация](#_Toc294915691)

[15. Физическая и логическая системы координат. Режимы отображения (без настройки параметров). Алгоритм преобразования координат в GDI и его связь с алгоритмом пересчёта мировых координат в оконные](#_Toc294915692)

[16. Режимы отображения и настройка их параметров (функции класса CDC MFC). Функция *SetMyMode*, назначение, параметры и реализация](#_Toc294915693)

[17. Аффинные преобразования на плоскости. Преобразование системы координат: смещение, растяжение-сжатие, поворот](#_Toc294915694)

[18. Аффинные преобразования на плоскости. Преобразование объектов: смещение, растяжение-сжатие, поворот](#_Toc294915695)

[19. Аффинные преобразования в пространстве. Преобразование системы координат: смещение, растяжение-сжатие, повороты вокруг осей координат](#_Toc294915696)

[20. Аффинные преобразования в пространстве. Преобразования объектов: смещение, растяжение-сжатие, повороты вокруг осей координат](#_Toc294915697)

[21. Основные типы проекций. Видовая система координат (вывод матрицы преобразования)](#_Toc294915698)

[22. Перспективные преобразования (вывод матрицы преобразования). Схема пересчёта координат при переходе от мировых 3D-координат к экранным](#_Toc294915699)

[23. Модели описания поверхностей](#_Toc294915700)

[24. Каркасная визуализация трёхмерных изображений. Принцип удаления невидимых граней для выпуклого многогранника. Реализация](#_Toc294915701)

[25. Изображение поверхности z = f(x,y). Метод сортировки граней по глубине (алгоритм художника). Реализация](#_Toc294915702)

[26. Закрашивание поверхностей. Модели отражения света](#_Toc294915703)

[27. Закрашивание поверхностей. Вычисление нормалей и углов отражения. Реализация](#_Toc294915704)

[28. Метод закрашивания Гуро](#_Toc294915705)

[29. Графическая библиотека OpenGL. Особенности, основные возможности, архитектура, синтаксис команд](#_Toc294915706)

[30. Графическая библиотека OpenGL. Структура GLUT-приложения. Пример](#_Toc294915707)

[31. Графическая библиотека OpenGL. Рисование графических примитивов. Дисплейные списки. Удаление нелицевых граней. Вывод текста. Примеры](#_Toc294915708)

[32. Графическая библиотека OpenGL. Системы координат. Модельно-видовые преобразования. Примеры](#_Toc294915709)

[33. Графическая библиотека OpenGL. Проекции. Область вывода. Создание анимации. Примеры](#_Toc294915710)

[34. Графическая библиотека OpenGL. Материалы и освещение. Примеры](#_Toc294915711)

[35. Графическая библиотека OpenGL. Структура приложения с использованием библиотеки MFC](#_Toc294915712)

## 1. Цели и задачи компьютерной графики

Понятие "компьютерная графика" объединяет довольно широкий круг опе­раций по обработке графической информации с помощью компьютера. Причем наблюдается явная тенденция "компьютеризации" изображений, циркулирующих в обществе. Стали обыденностью термины "цифровое фото" и "видео". В виртуальных буднях грядущего компьютерной графике отводится огромная роль. Это свя­зано с тем, что, по мнению ученых, исследующих проблемы мозга, зритель­ная система в иерархии мозговых структур человека занимает особое место. С восприятием и обработкой визуальной информации непосредственно свя­зано примерно 20% мозга человека. Благодаря зрению мы получаем по раз­ным оценкам от 70 до 90% сведений об окружающем мире. Следовательно, образный мир компьютерной графики является одним из глубинных прояв­лений человеческой природы.

В компьютерной графике можно выделить несколько основных на­правлений:

* Визуализация научных данных

Большинство современных математических программных пакетов (Maple, MatLab) имеют средства для отображения гра­фиков, поверхностей и трехмерных тел, построенных на основе каких-либо расчетов. Кроме того, графическая информация может активно использоваться в самом процессе вычислений. Визуализация по­зволяет представить большой объем данных в удобной для анализа форме и широко используется при обработке результатов различных измерений и вычислений.

* Геометрическое проектирование и моделирование

Это направление компьютерной графики связано с решением задач начертательной гео­метрии – построением чертежей, эскизов, объемных изображений с по­мощью программных систем, получивших название CAD-системы (от английского Computer-Aided Design), например AutoCAD. Существует большое количество специализированных CAD-систем в ма­шиностроении, архитектуре и т. д.

* Распознавание образов

Способность распознавать абстрактные образы считают одним из важнейших факторов, определившим развитие мыслительных способностей человека, выделив его из животного мира. Задача распознавания и классификации графической информации явля­ется одной из ключевых и при создании искусственного интеллекта. Уже в наши дни компьютеры распознают образы повсеместно (системы идентификации футбольных хулиганов у входа на стадион; анализ аэро- и космических фотоснимков; системы сортировки, наведения и т. д.). Возможно, самый известный пример распознавания образов – сканиро­вание и перевод "фотографии'' текста в набор отдельных символов, фор­мирующих слова. Такую операцию позволяет выполнить программное обеспечение многих современных сканеров. Кроме того, сущест­вуют специализированные программы распознавания текста, например FineReader.

* Изобразительное искусство

К этому направлению можно отнести разно­образную графическую рекламу: от текстовых транспарантов и фирмен­ных знаков до компьютерных видеофильмов, обработку фотографий, соз­дание рисунков, мультипликацию и т. д. В качестве примера популярных программ из этой области компьютерной графики можно назвать Adobe Photoshop (обработка растровых изображений), CorelDRAW (создание векторной графики), 3DS Мах (трехмерное моделирование).

* Виртуальная реальность

Реальность, даже виртуальная, подразумевает воздействия на всю совокупность органов чувств человека, в первую оче­редь на его зрение. К компьютерной графике можно отнести задачи мо­делирования внешнего мира в различных приложениях: от компьютер­ных игр до тренажеров.

* Цифровое видео

Все более широкое распространение получают анимированные изображения, записанные в цифровом формате. Это прежде всего фильмы, передаваемые через компьютерные сети, а также видео­диски, цифровое, кабельное и спутниковое те­левидение.

Приведенная классификация сфер применения компьютерной графики яв­ляется во многом условной. Возможно, найдутся задачи, которые нельзя отнести ни к одному из обозначенных направлений.

## 2. Растровая и векторная визуализация изображений. Основные характеристики растровых изображений

Наиболее известны два способа визуализации: растровый и векторный. Первый способ ассоциируется с такими графическими устройствами, как дисплей, телевизор, принтер. Второй используется в векторных дисплеях, плоттерах.

Наиболее удобно, когда способ описания графического изображения соот­ветствует способу визуализации. Иначе нужна конвертация. Например, изо­бражение может храниться в растровом виде, а его необходимо вывести (ви­зуализировать) на векторном устройстве. Для этого нужна предварительная векторизация – преобразование из растрового в векторное описание. Или наоборот, описание изображения может быть в векторном виде, а нужно ви­зуализировать на растровом устройстве – необходима растеризация.

Растровая визуализация основывается на представлении изображения на эк­ране или бумаге в виде совокупности (массива) отдельных точек (пикселов). Вместе пикселы образуют растр.

Для обозначения массива пикселов часто используется термин bitmap (битовая карта). В bitmap каждому пикселу отводится определенное число битов (одинаковое для всех пикселов изображения). Это число назы­вается битовой глубиной пиксела или цветовой глубиной изображения, т. к. от количества битов, отводимых на один пиксел, зависит количество цветов изображения.

Векторная визуализация основывается на формировании изображения на эк­ране или бумаге рисованием линий (векторов) – прямых или кривых. Сово­купность типов линий (графических примитивов), которые используются как базовые для векторной визуализации, зависит от определенного устройства. Типичная последовательность действий при векторной визуализации для плоттера или векторного дисплея такова: переместить перо в начальную точ­ку (для дисплея – отклонить пучок электронов); опустить перо (увеличить яркость луча); переместить перо в конечную точку; поднять перо (уменьшить яркость луча).

Качество векторной визуализации для векторных устройств обуславливается точностью вывода и номенклатурой базовых графических примитивов – ли­ний, дуг, кругов, эллипсов и других.

Доминирующим сейчас является растровый способ визуализации. Это обу­словлено большей распространенностью растровых дисплеев и принтеров. Недостаток растровых устройств – дискретность изображения. Недостатки векторных устройств – проблемы при сплошном заполнении фигур, мень­шее количество цветов, меньшая скорость (в сравнении с растровыми уст­ройствами).

Растровые изображения и их основные характеристики

Растр – это матрица (массив) ячеек (пикселов). Каждый пиксел может иметь свой цвет. Совокупность пикселов различного цвета образует изображение. В за­висимости от расположения пикселов в пространстве различают квадратный, прямоугольный, гексагональный или иные типы растра. Для описания распо­ложения пикселов используют разнообразные системы координат. Общим для всех таких систем является то, что координаты пикселов образуют дис­кретный ряд значений (необязательно целые числа). Часто используется сис­тема целых координат – номеров пикселов с (0,0) в левом верхнем уголку. Такую систему мы будем использовать и в дальнейшем, ибо она удобна для рассмотрения алгоритмов графического вывода.

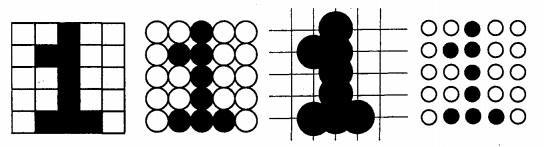
Геометрические характеристики растра

Разрешающая способность. Она характеризует расстояние между соседни­ми пикселами (см. рис. ниже). Разрешающую способность измеряют количеством пикселов на единицу длины. Наиболее популярной единицей измерения яв­ляется dpi (dots per inch) – количество пикселов в одном дюйме длины (2.54 см). Не следует отождествлять шаг с размерами пикселов – размер пикселов может быть равен шагу, а может быть как меньше, так и больше, чем шаг.



Размер растра обычно измеряется количеством пикселов по горизонтали и вертикали. Можно сказать, что для компьютерной графики зачастую наибо­лее удобен растр с одинаковым шагом для обеих осей, то есть dpiX = dpiY. Это удобно для многих алгоритмов вывода графических объектов.

Форма пикселов растра определяется особенностями устройства графиче­ского вывода (см. рис. ниже). Например, пикселы могут иметь форму прямоуголь­ника или квадрата, которые по размерам равны шагу растра (дисплей на жидких кристаллах); пикселы круглой формы, которые по размерам могут и не равняться шагу растра (принтеры).



Количество цветов (глубина цвета) – также одна из важнейших характери­стик растра. Количество цветов является важной характеристикой для любо­го изображения, а не только растрового. Согласно психофизиологическим исследованиям глаз человека способен различать 350000 цветов.

Классификация изображений:

* Двухцветные (бинарные) – 1 бит на пиксел. Среди двухцветных чаще всего встречаются черно-белые изображения.
* Полутоновые – градации серого или иного цвета. Например, 256 града­ций (1 байт на пиксел).
* Цветные изображения. От 2 бит на пиксел и выше. Глубина цвета 16 бит на пиксел (65 536 цветов) получила название High Color, 24 бит на пиксел (16,7 млн цветов) – True Color. В компьютерных графических системах используют и большую глубину цвета – 32, 48 и более бит на пиксел.

## 3. Аддитивная цветовая модель RGB. Кодирование цвета. Палитра

Для того чтобы компьютер имел возможность работать с цветными изобра­жениями, необходимо представлять цвета в виде чисел – кодировать цвет. Способ кодирования зависит от цветовой модели и формата числовых дан­ных в компьютере.

Чтобы оцифровать цвет, его необходимо измерить. Наука, которая изучает цвет и его измерения, называется колориметрией. Она описывает общие закономерности цветового восприятия света челове­ком. Одними из основных законов колориметрии являются законы смешивания цветов. Эти законы в наиболее полном виде были сформулированы в 1855 году немецким математиком Германом Грассманом:

* закон трехмерности – любой цвет может быть представлен ком­бинацией трех основных цветов;
* закон непрерывности – к любому цвету можно подобрать беско­нечно близкий;
* закон аддитивности – цвет смеси зависит только от цвета состав­ляющих.

Первый закон означает, что для любого заданного цвета (Color) можно записать такое цве­товое уравнение, выражающее линейную зависимость цветов:

Color = k1Color1 + k2Color2 + k3Color3,

где Color1, Color2, Color3 – некоторые базисные, линейно независимые цвета, коэффи­циенты k1 , k2 , k3 указывают количество соответствующего смешиваемого цвета. Линейная независимость цветов означает, что ни один из них не может быть выражен взвешенной суммой (линейной комбинацией) двух других.

Цветовая модель RGB

За основные три цвета приняты красный (Red), зеленый (Green), синий (Blue). В модели RGB любой цвет (Color) получается в результате сло­жения основных цветов.

Для модели RGB каждая из компонент может представляться числами, огра­ниченными некоторым диапазоном – например, дробными числами от 0 до 1 либо целыми числами от 0 до некоторого максимального значения. В настоящее время достаточно распространенным является формат True Color, в котором каждая компонента представлена в виде байта, что дает 256 градаций для каждой компоненты: R=0...255, G = 0...255, В = 0...255. Ко­личество цветов составляет 256\*256\*256 = 16.7 млн (224).

Такой способ кодирования цветов можно назвать компонентным. В компьютере коды изображений True Color представляются в виде троек байтов либо упаковываются в длинное целое (четырехбайтное) – 32 бита (так, на­пример, сделано в API Windows):

C = 00000000 bbbbbbbb gggggggg rrrrrrrr.

Цветовая модель RGB применяется для создания графических образов в устройствах, излучающих свет, - мониторах, телевизорах.

Цветовая модель CMYK. В полиграфических системах напечатанный на бумаге графический объект сам не излучает световых волн. Изобра­жение формируется на основе отраженной волны от окрашенных поверхностей. Окрашенные поверхности, на которые падает белый свет (т.е. сумма всех цветов), должны поглотить (т.е. вычесть) все составляющие цвета, кроме того, цвет которой мы видим. Цвет по­верхности можно получить красителями, которые поглощают, а не излучают. Например, если мы видим зеленое дерево, то это означа­ет, что из падающего белого цвета, т.е. суммы красного, зеленого, синего, поглощены красный и синий, а зеленый отражен. Цвета кра­сителей должны быть дополняющими:

* голубой (Cyan = В + G), дополняющий красного;
* пурпурный (Magenta = R + В), дополняющий зеленого;
* желтый (Yellow = R + G), дополняющий синего.

Но так как цветные красители по отражающим свойствам не одинаковы, то для повышения контрастности применяется еще чер­ный (Black). Модель CMYK названа по первым буквам слов Cyan, Magenta, Yellow и последней букве слова Black. Так как цвета вычи­таются, модель называется субстрактивной.

При работе с изображениями в системах компьютерной графики часто при­ходится искать компромисс между качеством изображения (требуется как можно больше цветов) и ресурсами, необходимыми для хранения и воспро­изведения изображения, исчисляемыми, например, объемом памяти (надо уменьшать количество бит на пиксел).

Кроме того, некоторое изображение само по себе может использовать огра­ниченное количество цветов. Например, для черчения может быть достаточ­но двух цветов, для человеческого лица важны оттенки розового, желтого, пурпурного, красного, зеленого; а для неба – оттенки голубого и серого. В этих случаях использование полноцветного кодирования цвета является избыточным.

При ограничении количества цветов используют палитру, представляющую набор цветов, важных для данного изображения. Палитру можно восприни­мать как таблицу цветов. Палитра устанавливает взаимосвязь между кодом цвета и его компонентами в выбранной цветовой модели.

## 4. Графические форматы. Работа с растровыми изображениями (форматы DDB, DIB, bmp-файлы)

Графическим форматом называют порядок (структуру), согласно которому данные, описывающие изображение, записаны в файле.

Типы форматов графических файлов определяются способом хранения и типом графических данных. Наиболее широко используются растровый, векторный и метафайловый форматы.

* Векторный формат наиболее удобен для хранения изображений, которые можно разложить на простые геометрические фигуры (например, чертежи или текст). Векторные файлы содержат математические описания элементов изображения. Наиболее распространенные векторные форматы: AutoCAD DXF и Microsoft SYLK.
* Растровый формат используется для хранения растровых данных. Файлы такого типа особенно хорошо подходят для хранения изображений реаль­ного мира, например оцифрованных фотографий. Растровые файлы содер­жат битовую карту изображения и се спецификацию. Наиболее распростра­ненные растровые форматы: BMP, TIFF, GIF, PCX, JPEG.
* Метафайловый формат позволяет хранить в одном файле и векторные, и растровые данные. Примером такого формата являются файлы CorelDRAW - CDR.

Кроме того, существуют файловые форматы для хранения мультипликации (видеоинформации), мультимедиа-форматы (одновременно хранят звуко­вую, видео- и графическую информацию), гипертекстовые (позволяют хра­нить не только текст, но и связи-переходы внутри него) и гипермедиа (ги­пертекст плюс графическая и видеоинформация) форматы, форматы трехмерных сцен, форматы шрифтов и т. д.

Формат файлов для хранения растровых изображений

К настоящему времени известно много форматов файлов для растровых изо­бражений. Здесь мы рассмотрим один из самых популярных форматов, кото­рый обязан своей распространенностью операционной системе Windows – формат BMP.

Общая структура ВМР – файла такова:



|  |  |
| --- | --- |
| BITIMAPFILEHEADER | 14 байт |
| BITMAPINFOHEADER | 40 байт |
| Палитра | Размер зависит от количества цветов |
| Битовый массив растрового изображения | Число байт определяется размерами растра и количеством бит на пиксел |

Заголовок файла BMP называется BITMAPFILEHEADER. В нем помещает­ся общее описание файла:

* код формата - символы "ВМ";
* общий размер файла в байтах;
* адрес битового массива в данном файле.

Далее в файле следует еще один заголовок – BITMAPINFOHEADER, в котором хранится описание размеров растра и цветового формата пикселов:

* размер заголовка;
* ширина растра в пикселах;
* высота растра в пикселах;
* битовая глубина пиксела (бит/пиксел);
* размер в байтах битового массива растра.

Затем в файле помещается палитра в виде записей RGBQUAD. Каждая запись содержит значения каждой из цветовых компонент модели RGB в виде числа 0…255. Количество записей RGBQUAD равно количеству используемых цветов.

Па­литра отсутствует, если число бит на пиксел равно 24. Также палитра не нужна и для некоторых цветовых форматов 16 и 32 бит на пиксел.

После палитры (если она есть) в файле BMP записывается растр в виде бито­вого (а точнее, байтового массива). В битовом массиве последовательно за­писываются байты строк растра. Количество байт в строке должно быть кратно четырем, поэтому если количество пикселов по горизонтали не соот­ветствует такому условию, то справа в каждую строку дописывается некото­рое число битов (выравнивание строк на границу двойного слова).

Форматы DDB и DIB

В операционной системе Windows используются два формата битовых изображений – аппаратно-зависимый DDB (device-dependent bitmap) и аппаратно-независимый DIB (device-independent bitmap).

Согласно определению, данному в документации к SDK, битовое изображение DDB есть набор бит в оперативной памяти, который может быть отображен на устройстве вывода(например, выведен на экран видеомонитора или распечатан на принтере). Внутренняя структура изображения DDB жестко привязана к аппаратным особенностям устройства вывода. Поэтому представление изображения DDB в оперативной памяти полностью зависит от устройства вывода.

Иногда битовые изображения называют растровыми изображениями, подчеркивая тот факт, что его можно рассматривать как совокупность строк растра (горизонтальных линий развертки).

Если бы в Windows можно было работать только с изображениями DDB, было бы необходимо иметь отдельные наборы изображений для каждого типа видеоконтроллера и каждого видеорежима, что, очевидно, крайне неудобно.

Аппаратно-независимое битовое изображение DIB содержит описание цвета пикселов изображения, которое не зависит от особенностей устройства отображения. Операционная система Windows после соответствующего преобразования может отобразить такое изображение на любом устройстве вывода. Несмотря на некоторое замедление процесса вывода по сравнению с выводом изображений DDB, универсальность изображений DIB делает их весьма привлекательными для хранения изображений.

Битовые изображения в формате DDB

Битовые изображения в формате DDB являются аппаратно-зависимыми. Поэтому структура изображения в оперативной памяти зависит от особенностей аппаратуры.

Как правило, изображения DDB либо загружаются из ресурсов приложения, либо создаются непосредственно в оперативной памяти. Для вывода изображений DDB на экран используются такие функции, как BitBlt и StretchBlt.

BOOL WINAPI BitBlt(

HDC hdcDest, // контекст для рисования

int nXDest, // x-координата верхнего левого угла

// области рисования

int nYDest, // y-координата верхнего левого угла

// области рисования

int nWidth, // ширина изображения

int nHeight, // высота изображения

HDC hdcSrc, // идентификатор исходного контекста

int nXSrc, // x-координата верхнего левого угла

// исходной области

int nYSrc, // y-координата верхнего левого угла

// исходной области

DWORD dwRop) // код растровой операции

Функция копирует битовое изображение из исходного контекста hdcSrc в контекст отображения hdcDest. Возвращаемое значение равно TRUE при успешном завершении или FALSE при ошибке.

В качестве кода растровой операции чаще всего используется константа SRCCOPY. При этом цвет пикселов копируемого изображения полностью замещает цвет соответствующих пикселов контекста отображения. В этом случае цвет кисти, выбранной в контекст отображения, не имеет значения, так как ни цвет кисти, ни цвет фона не влияют на цвет нарисованного изображения.

Для рисования битовых изображений можно использовать вместо функции BitBlt функцию StretchBlt , с помощью которой можно выполнить масштабирование (сжатие или растяжение) битовых изображений:

BOOL WINAPI StretchBlt(

HDC hdcDest, // контекст для рисования

int nXDest, // x-координата верхнего левого угла

// области рисования

int nYDest, // y-координата верхнего левого угла

// области рисования

int nWidthDest, // новая ширина изображения

int nHeightDest, // новая высота изображения

HDC hdcSrc, // идентификатор исходного контекста

int nXSrc, // x-координата верхнего левого угла

// исходной области

int nYSrc, // y-координата верхнего левого угла

// исходной области

int nWidthSrc, // ширина исходного изображения

int nHeightSrc, // высота исходного изображения

DWORD dwRop) // код растровой операции

Параметры этой функции аналогичны параметрам функции BitBlt, за исключением того, что ширина и высота исходного и полученного изображения должна определяться отдельно. Размеры исходного изображения (логические) задаются параметрами nWidthSrc и nHeightSrc, размеры нарисованного изображения задаются параметрами nWidthDest и nHeightDest.

Возвращаемое значение равно TRUE при успешном завершении или FALSE при ошибке.

Битовые изображения в формате DIB

Изображения DIB, в отличие от изображений DDB, являются аппаратно-независимыми, поэтому без дополнительного преобразования их нельзя отображать на экране с помощью функций BitBlt и StretchBlt. В операционной системе Windows битовые изображения хранятся в файлах с расширением имени bmp, при этом используется аппаратно-независимый формат DIB.

Файл, содержащий битовое изображение, начинается со структуры BITMAPFILEHEADER. Эта структура описывает тип файла и его размер, а также смещение области битов изображения.

typedef struct tagBITMAPFILEHEADER {

WORD bfType; // Признак BMP – файла, 42 4D (коды букв ВМ)

DWORD bfSize; // Размер файла

WORD bfReserved1; // Первое резервное поле - всегда ноль.

WORD bfReserved2; // Второе резервное поле - тоже ноль

DWORD bfOffBits; // Смещение от начала файла до первого

// байта графических данных

} BITMAPFILEHEADER, \*PBITMAPFILEHEADER;

Сразу после структуры BITMAPFILEHEADER в файле следует структура BITMAPINFO, которая содержит описание изображения и таблицу цветов.

typedef struct tagBITMAPINFO {

BITMAPINFOHEADER bmiHeader;

RGBQUAD bmiColors[1];

} BITMAPINFO, \*PBITMAPINFO;

Описание изображения (размеры изображения, метод компрессии, размер таблицы цветов и т. д.) находится в структуре BITMAPINFOHEADER.

typedef struct tagBITMAPINFOHEADER{

DWORD biSize; // Размер BITMAPINFOHEADER в байтах.

LONG biWidth; // Ширина картинки в пикселах

LONG biHeight; // Высота картинки в пикселах

WORD biPlanes; // Количество битовых плоскостей (=1, см. MSDN)

WORD biBitCount; // Количество бит на пиксел (определяет

// Mаксимальное число цветов в bitmap)

DWORD biCompression; // Cпособ сжатия (0 - данные не сжимаются).

DWORD biSizeImage; // Размер изображения в байтах (без заголовков)

LONG biXPelsPerMeter; // Число пикселов на метр по горизонтали

LONG biYPelsPerMeter; // Число пикселов на метр по вертикали

DWORD biClrUsed; // Количество элементов палитры, хранящихся

// после заголовка

DWORD biClrImportant; // Количество разных цветов, действительно

// используемых в рисунке.

} BITMAPINFOHEADER, \*PBITMAPINFOHEADER;

В некоторых случаях (не всегда) в файле может присутствовать таблица цветов (как массив структур RGBQUAD), присутствующих в изображении.

typedef struct tagRGBQUAD

{

BYTE rgbBlue;

BYTE rgbGreen;

BYTE rgbRed;

BYTE rgbReserved;

} RGBQUAD;

Биты изображения обычно располагаются сразу после таблицы цветов. Точное значение смещения битов изображения находится в структуре BITMAPFILEHEADER.

Загрузка данных из BMP файла

Чтобы изображение загрузить с диска в оперативную память и получить дескриптор изображения (типа HBITMAP), предназначены функции LoadBitmap() и LoadImage().

HBITMAP Bit = (HBITMAP)LoadImage(NULL, char\* Filename, IMAGE\_BITMAP, 0, 0, LR\_LOADFROMFILE | LR\_CREATEDIBSECTION)

Значение NULL первого параметра указывает, что картинка вводится из файла. При получении изображения из ресурса в первом параметре указывается дескриптор приложения. Во втором параметре задается имя файла или идентификатор ресурса. Третий параметр задает тип изображения и может принимать значения IMAGE\_BITMAP, IMAGE\_CURSOR, IMAGEICON. Четвертый и пятый параметры, которые указывают размеры изображения, при вводе битовой карты задаются нулями, так как эта информация имеется в заголовке BMP-файла. Шестой параметр содержит сочетание флагов. В частности, флаг LR\_LOADFROMFILE указывает, что изображение вводится из файла. При задании флага LR\_CREATEDIBSECTION создается аппаратно-независимое, а при его отсутствии – аппаратно-зависимое растровое изображение. Ниже приводится код функции для загрузки битовой карты, которая получает путь к файлу FileName и тип растра Туре. Нулевое значение будет означать, что используется тип DIB, а единичное значение будет применяться для типа DDB.

HBITMAP LoadPict(char \*FileName, int Type)

{

if(Type)

return LoadImage(NULL, FileName,

IMAGE\_BITMAP, 0, 0, LR\_LOADFROMFILE);

else return Loadlmage(NULL, FileName, IMAGE\_BITMAP, 0, 0,

LR\_LOADFROMFILE|LR\_CREATEDIBSECTION);

}

Функция LoadImage() позволяет загружать из ресурса битовую карту, курсор или пиктограмму, но для этого более удобны простые специализированные функции LoadBitmap(), LoadCursor(), LoadIcon().

## 5. Понятие об инкрементных алгоритмах растеризации. Растровое представление отрезка. Вывод и реализация алгоритма

Алгоритмы компьютерной графики можно разделить на два уровня: нижний и верхний. Группа алгоритмов нижнего уровня предназначена для реализации графических примитивов (линий, окружностей, заполнений и т.п.). Эти алгоритмы или подобные им воспроизведены в графических библиотеках языков высокого уровня или реализованы аппаратно в графических процессорах рабочих станций.

Среди алгоритмов нижнего уровня можно выделить следующие группы:

* I группа – используют простые математические методы и отличаются простотой реализации. Как правило, такие алгоритмы не являются наилучшими по объему выполняемых вычислений или требуемым ресурсам памяти.
* II группа – используют более сложные математические методы по сравнению с алгоритмами I группы (но часто и эвристические) и отличаются большей эффективностью.
* III группа – алгоритмы, которые могут быть без больших затруднений реализованы аппаратно (допускающие распараллеливание, рекурсивные, реализуемые в простейших командах). В эту группу могут попасть и алгоритмы, представленные в первых двух группах.

К алгоритмам верхнего уровня относятся в первую очередь алгоритмы удаления невидимых линий и поверхностей. Задача удаления невидимых линий и поверхностей продолжает оставаться центральной в машинной графике. От эффективности алгоритмов, позволяющих решить эту задачу, зависят качество и скорость построения трехмерного изображения.

К задаче удаления невидимых линий и поверхностей примыкает задача построения (закрашивания) полутоновых (реалистических) изображений, т.е. учета явлений, связанных с количеством и характером источников света, учета свойств поверхности тела (прозрачность, преломление, отражение света).

Однако при этом не следует забывать, что вывод объектов в алгоритмах верхнего уровня обеспечивается примитивами, реализующими алгоритмы нижнего уровня, поэтому нельзя игнорировать проблему выбора и разработки эффективных алгоритмов нижнего уровня.

Для разных областей применения компьютерной графики на первый план могут выдвигаться разные свойства алгоритмов. Для научной графики большое значение имеет универсальность алгоритма, быстродействие может отходить на второй план. Для систем моделирования, воспроизводящих движущиеся объекты, быстродействие становится главным критерием, поскольку требуется генерировать изображение практически в реальном масштабе времени.

Особенности растровой графики связаны с тем, что обычные изображения, с которыми сталкивается человек в своей деятельности (чертежи, графики, карты, художественные картины и т.п.), реализованы на плоскости, состоящей из бесконечного набора точек. Экран же растрового дисплея представляется матрицей дискретных элементов, имеющих конкретные физические размеры. При этом число их существенно ограничено. Поэтому нельзя провести точную линию из одной точки в другую, а можно выполнить только аппроксимацию этой линии с отображением ее на дискретной матрице (плоскости). Такую плоскость также называют целочисленной решеткой, растровой плоскостью или растром. Эта решетка представляется квадратной сеткой с шагом 1. Отображение любого объекта на целочисленную решетку называется разложением его в растр или просто растровым представлением.

Базовые растровые алгоритмы

Сформировать растровое изображение можно по-разному. Для того чтобы создать изображение на растровом дисплее, можно просто скопировать гото­вый растр в видеопамять. Этот растр может быть получен, например, с по­мощью сканера или цифрового фотоаппарата. А можно создавать изобра­жение объекта путем последовательного рисования отдельных простых эле­ментов.

Простые элементы, из которых складываются сложные объекты, называют графическими примитивами. Их можно встретить повсюду. Например, для построения изображения используется некоторый набор примитивов, ко­торые поддерживаются определенными графическими устройствами вывода. Графические примитивы также можно применять для описания пространст­венных объектов в базе данных компьютерной системы – модели объектов. Могут использоваться различные множества примитивов для модели и для алгоритмов отображения. Удобно, когда эти множества совпадают, тогда уп­рощается процесс отображения.

Простейшим и, вместе с тем, наиболее универсальным растровым графиче­ским примитивом является пиксел. Любое растровое изображение можно на­рисовать по пикселам, но это сложно и долго. Необходимы больше сложные элементы, для которых рисуются сразу несколько пикселов.

Инкрементные алгоритмы растеризации

Брезенхэм предложил подход, позволяющий разрабатывать так назы­ваемые инкрементные алгоритмы растеризации. Основной целью для разра­ботки таких алгоритмов было построение циклов вычисления координат на основе только целочисленных операций сложения/вычитания без использо­вания умножения и деления. Инкрементные алгоритмы выполняются как последовательное вычисление координат соседних пикселов путем добавления приращений координат.

Разработаны инкрементные алгоритмы для вывода отрезка, окружности, эллипса.

Алгоритм вывода прямой линии

dH

dL

xi-1

xi

Yi-1

Yi

Ai-1

Ai

Y

Пусть на растре заданы две точки с целочисленными координатами рисунка выше.

 и 

Запишем уравнение прямой, проходящей через эти точки



или

 (1)

где

д, , , 

Будем рассматривать случай, когда  и . Пусть точка  уже поставлена. Необходимо принять решение о том, какие координаты должна иметь точка:  или 

Запишем выражение для - координаты математической прямой для :



Вычислим разности









Тогда



Выполним замену в 

или



Полагая 

Получим  (2)

Где  (3)

 (4)

Так как , то



Вычитая (2) из (4), получим



но , тогда



При этом



Поэтому



Вычислим начальное значение 

Из (2) получаем

 (5)

Значение  соответствует точке 

Точка  по условию удовлетворяет уравнению прямой (1)



Отсюда

Подставляя полученное значение  в выражение для , получим





Подставляя полученное выражение для  в выражение (5) для , получим



## 6. Понятие об инкрементных алгоритмах растеризации. Построение окружности методом средней точки. Вывод и реализация алгоритма

Брезенхэм предложил подход, позволяющий разрабатывать так назы­ваемые инкрементные алгоритмы растеризации. Основной целью для разра­ботки таких алгоритмов было построение циклов вычисления координат на основе только целочисленных операций сложения/вычитания без использо­вания умножения и деления. Инкрементные алгоритмы выполняются как последовательное вычисление координат соседних пикселов путем добавления приращений координат.

Разработаны инкрементные алгоритмы для вывода отрезка, окружности, эллипса.

Алгоритм построения окружности методом средней точки

Рассмотрим уравнение окружности радиуса с центром в начале координат

 (1)

Введем в рассмотрение функцию окружности  (2)

Любая точка , которая лежит на окружности удовлетворяет урав­нению 

Если точка находится внутри круга, то функция окружности будет иметь отрицательное значение. Если точка лежит за пределами круга, значение функции окружности будет положительным. Подытоживая, можно сказать, что от­носительное положение любой точки с координатами  определяется проверкой знака функции окружности:

 (3)

Проверка выполняется на каждом этапе выборки для средних положений между пикселями вблизи заданной окружности. Таким образом, функция окружности – это параметр принятия решения в алгоритме средней точки, и для этой функции можно установить операции приращения, как это было сделано для алгоритма построения прямой линии.



Xi

Yi

Xi+1

Yi-1

Yi-½

На рис. показана средняя точка между двумя возможными пикселями в точке выборки. Предположим, что мы только что поставили точку в пикселе с координатами  Теперь нужно определить, какой из двух пикселей ближе к заданной окружности – пиксель с координатами  или пиксель с координатами  Параметром принятия решения будет функция окружности (3), которая рассчитывается для средней точки между этими двумя пикселями:



В случае, когда , полагаем . Тогда









При этом



Тогда

, 

, 

В случае, когда , делаем шаг вниз, полагая . Тогда











Где





Для рассматриваемого случая, когда , 



При этом



Тогда





Таким образом

, 

, 

Для случая, когда , 



И выражение для принимает вид



При этом

, 

, 

Таким образом,

если 



, 

, 

, 

если 



, 

, 

, 

Для удобства заменим в выражениях для  и   на , тогда

если 



, 

, 

, 

если 



, 

, 

, 

Начальные значения параметров  и  определим из условия  и 

Тогда 

Поскольку радиус и приращение,  целые числа, то значение можно округлить и принять









Алгоритм поэтапно:

* Установить начальные значение для 
* Поставить начальные точки
* Выполнять в цикле пока 
* если 



, 

, 

, 

* если 



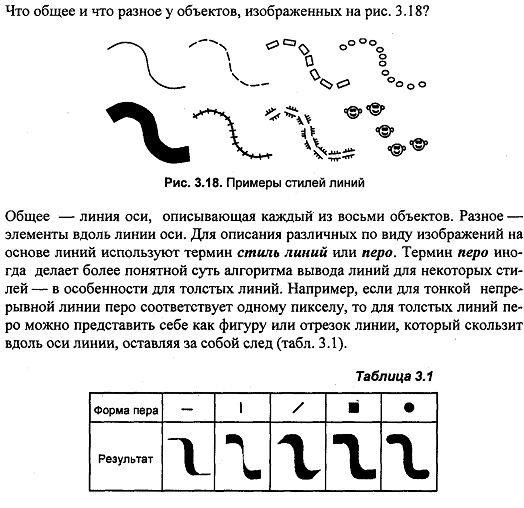
, 

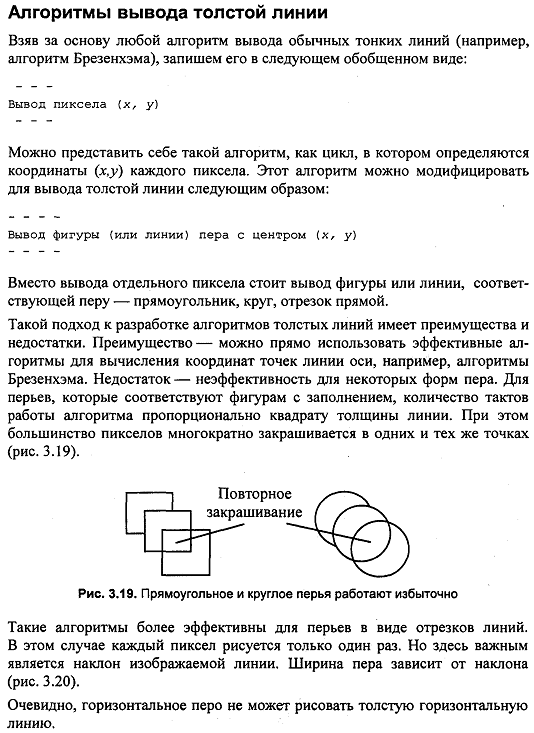
, 

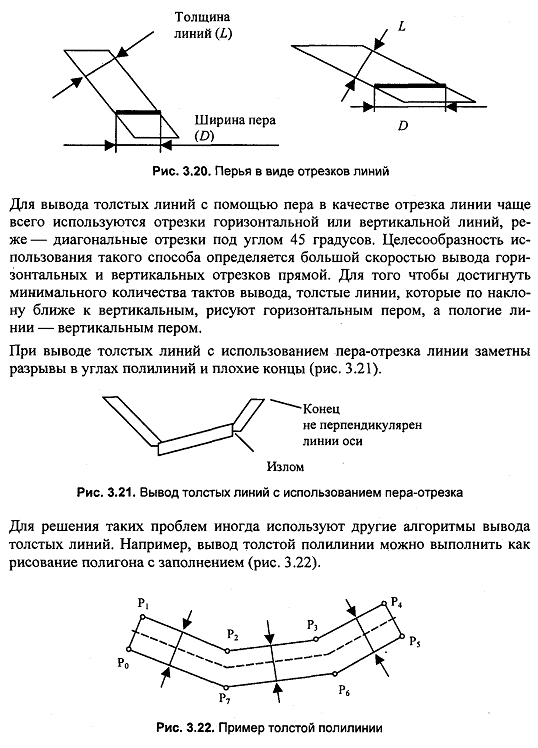
, 

* Поставить точки, соответствующие текущему шагу

## 7. Стиль линии (перо). Алгоритм вывода линии







## 8. Стиль заполнения, кисть, текстура

Для обозначения стилей заполнения, отличных от сплошного стиля, используют такие понятия, как кисть и текстура. Текстура – это стиль заполнения, закрашивание, которое имитирует сложную рельефную объемную поверхность, выполненную из какого-то материала.

Для описания алгоритмов заполнения фигур с определенным стилем исполь­зуем следующий алгоритм рисования линий:

* вывод пиксела заполнения цвета C с координатами (х, у)

Например, в алгоритме вывода полигонов пикселы заполнения рисуются в теле цикла горизонталей, а все другие операции предназначены для подсчета координат (х, у) этих пикселов. Сплошное заполнение означает, что цвет (С) всех пикселов одинаков, то есть C = const. Преобразуем ал­горитм заполнения следующим образом:

* С = f (х, у)
* вывод пиксела заполнения (х, у) цветом C.

Функция f(x, y) будет определять стиль заполнения. Аргументами функции цвета являются координаты текущего пиксела заполнения. Однако в отдельных случаях эти аргументы не нужны. Например, если цвет C вычисляется как случайное значение в определённых границах: С = random(), то можно создать иллюзию шершавой матовой поверхности (рис. 3.23).



Другой стиль заполнения – штриховой. Для него функцию цвета также можно записать в аналитической форме:



где *S* – период, а *T* – толщина штрихов, *Cш* – цвет штрихов, *Cф* – цвет фона.

Зачастую при использовании кистей и текстур используется копирование не­больших растровых изображений. Такой алгоритм заполнения можно опи­сать вышеупомянутой общей схемой, если строку С = f (х, у) заменить двумя другими строками:

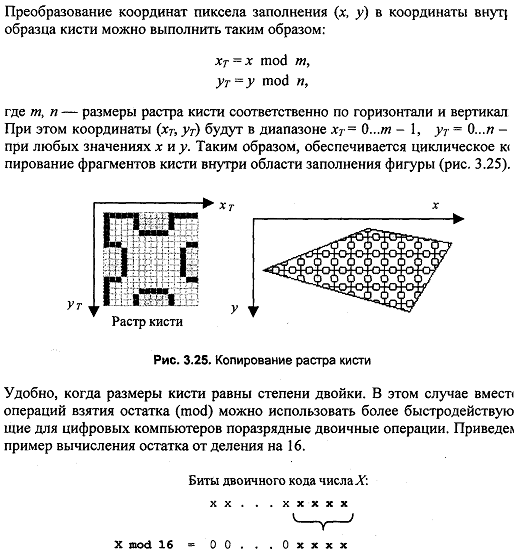
* координаты пиксела заполнения (х, у) преобразуем в растровые координаты образца кисти (хт / yт*)*
* по координатам (хт / yт*)* определяем цвет (С) пиксела в образце кисти
* вывод пиксела заполнения цвета С c координатами (х, у)

Преобразование координат пиксела заполнения (х, у) в координаты внутри образца кисти можно выполнить таким образом:

хт *= х* mod m,

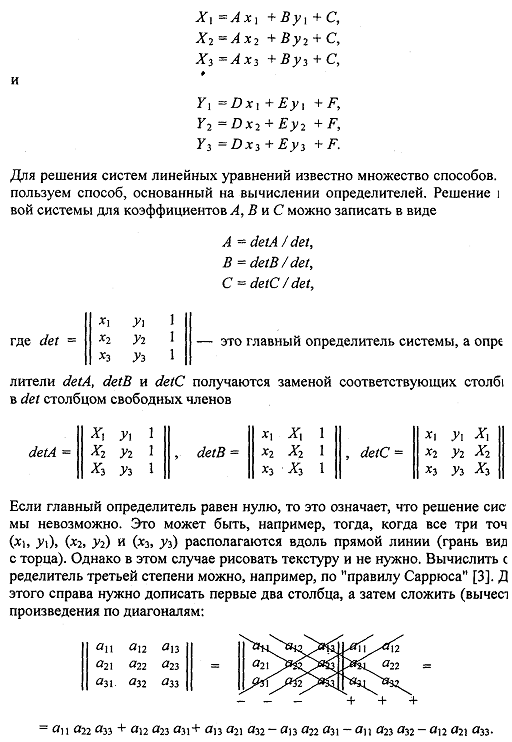
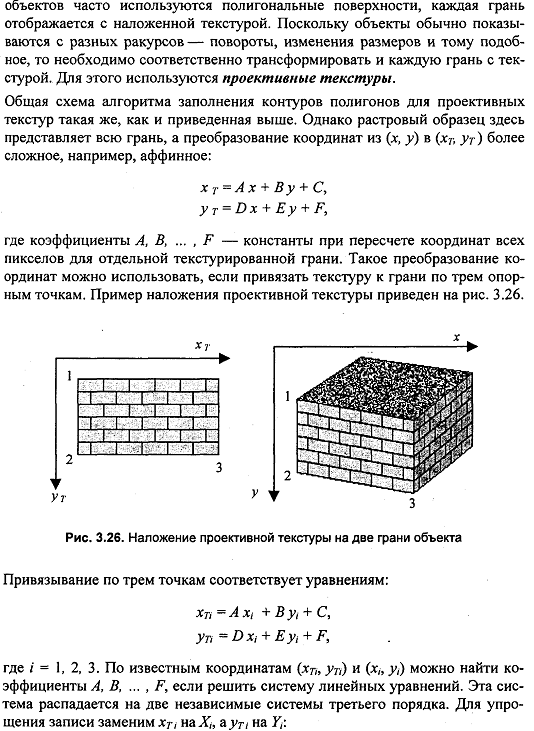
yт *= y* mod n*,*

где m, n – размеры растра кисти соответственно по горизонтали и вертикали. При этом координаты (хт, yт*)* будут в диапазоне хт *=* 0...m - 1, yт *=* 0...n –при любых значениях х и у.Таким образом, обеспечивается циклическое копирование фрагментов кисти внутри области заполнения фигуры.



Проективные текстуры

Поскольку объекты часто показываются с разных ракурсов – повороты, изменения размеров и т. п., то необходимо трансформировать соответственно и каждую грань с текстурой. Для этого используются проективные текстуры.

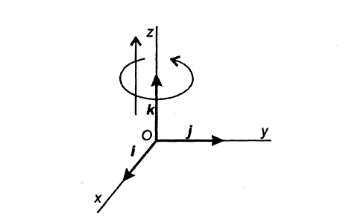


## 9. Геометрические основы компьютерной графики. Системы координат (декартова, полярная, сферическая) и векторы. Скалярное и векторное произведение векторов

Системы координат и векторы

В прямоугольной системе координат  направление осей задается тройкой перпендикулярных единичных векторов.

Система координат называется правой, если при повороте от вектора  к век­тору  на 90° направление вектора  совпадает с поступательным движением винта с правой резьбой (рис. ниже). Начальная точка векторов обозначается буквой О.















Пусть заданы два вектора и 

 и 

Тогда





При этом справедливы равенства



Скалярное произведение векторов

,

где  - угол между векторами и 

В координатной форме



Или

,

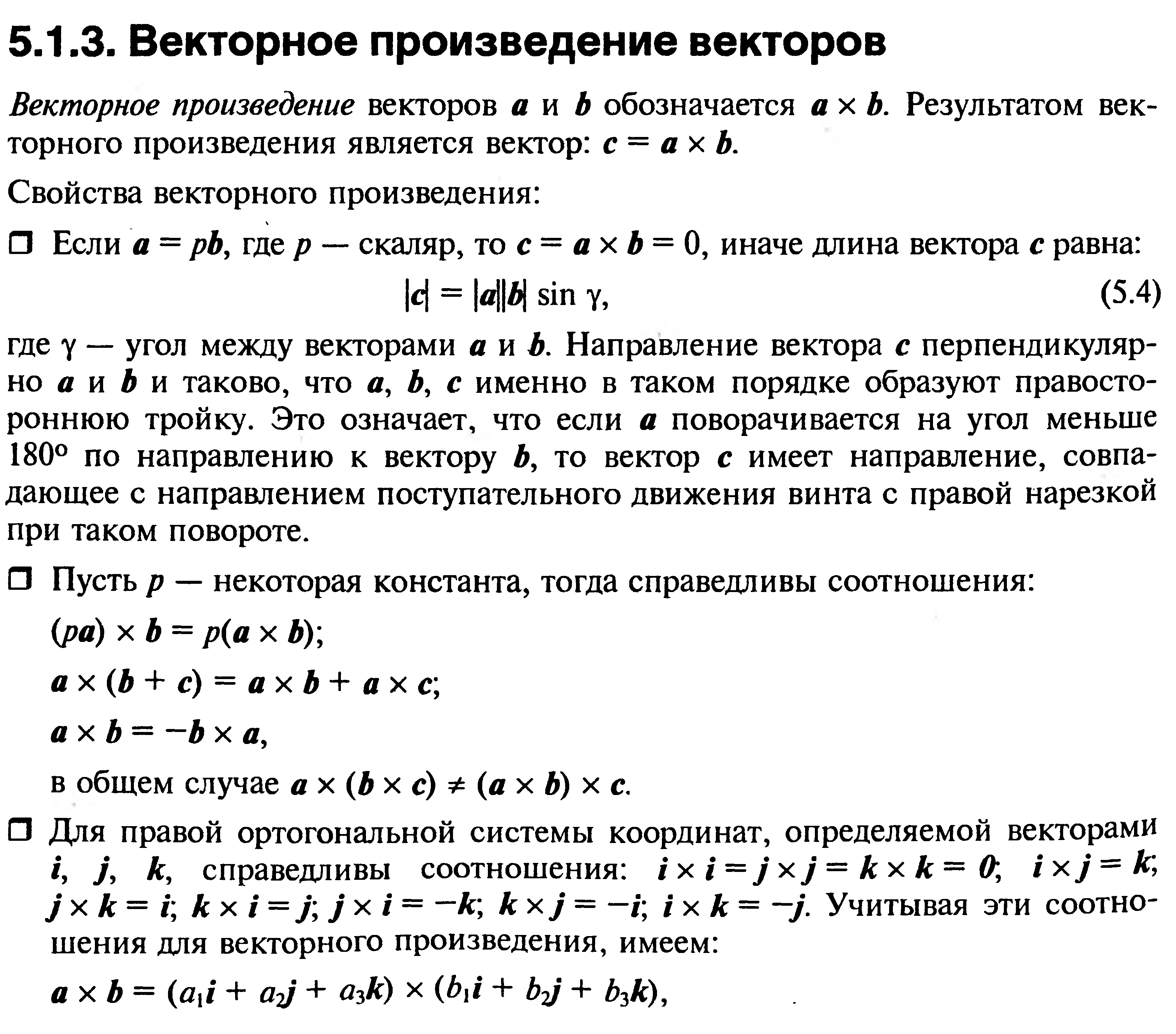
если вектора и  представлены в форме ().

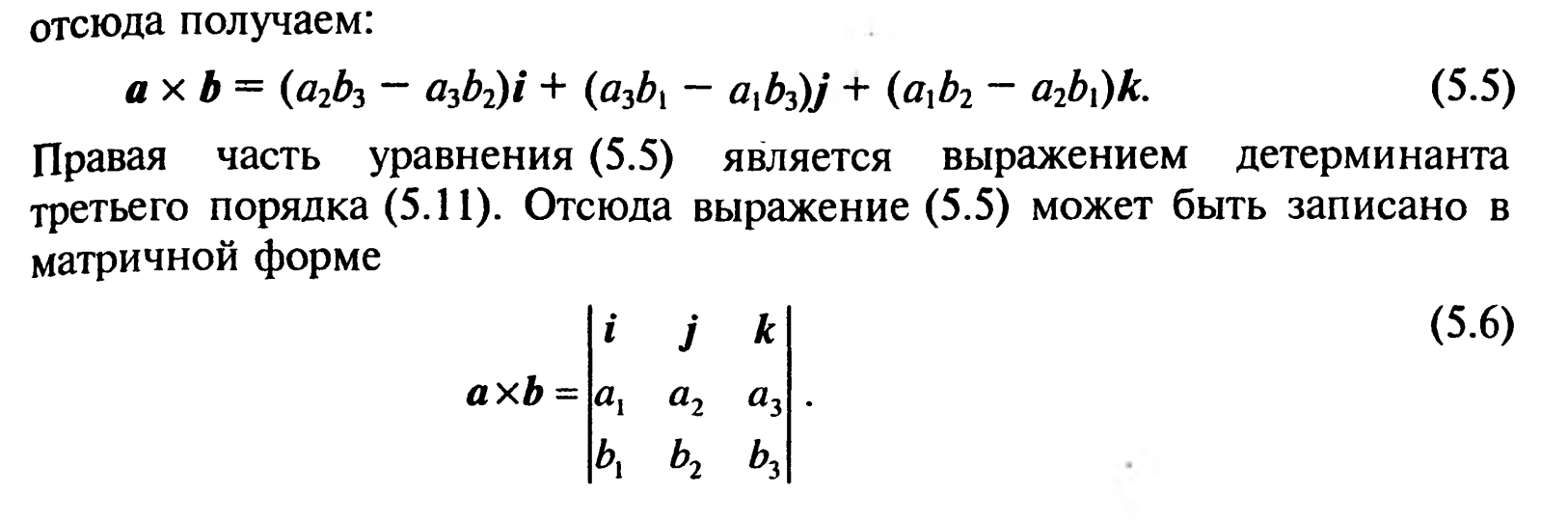


Если , то угол между векторами и является острым , при – вектора и  ортогональны, а если , то угол между векторами является тупым .

Отметим некоторые свойства скалярного произведения.

Пусть ,,- некоторые числа. Тогда справедливы равенства:





Полярная система координат

|  |  |
| --- | --- |
|  |  |

Сферическая система координат

|  |  |
| --- | --- |
|  |  |

## 10. Геометрические основы компьютерной графики. Уравнение прямой, проходящей через две точки (на плоскости). Параметрическое уравнение прямой. Уравнение отрезка. Задача о взаимном расположении двух отрезков

Уравнение прямой, проходящей через две точки

Пусть в декартовой системе координат (ДСК) заданы две точки

 и 

Тогда уравнение прямой, проходящей через эти точки имеет вид:



Параметрическое уравнение прямой, проходящей через две точки

Положим



Отсюда



Пусть



Тогда можно записать



или



Перепишем уравнение в виде



При  получаем, что . При  получаем, что 

При  точка  принадлежит отрезку 

Таким образом при , уравнение (\_) можно считать уравнением отрезка .

Задача о взаимном расположении двух отрезков

Условие задачи. На плоскости заданы два отрезка прямой:  и 

, 

, 

Определить: пересекаются или нет отрезки  и 

Запишем уравнения прямых  и , определяемых отрезкамии  соответственно в параметрической форме





Предположим, что прямые пересекаются и  точка из пересечения

В этой точке справедливо равенство



Или



Отсюда







Определитель системы



Если , то система уравнений (\_) решения не имеет: отрезки параллельны.

Если , то система уравнений (\_) имеет единственное решение  и .

При этом если  и , то отрезки пересекаются, иначе пересекаются их продолжения, или один отрезок пересекается с продолжением другого.





## 11. Построение кривых. Интерполяционный полином Лагранжа. Достоинства и недостатки. Реализация

Постановка задачи интерполирования

Пусть некоторая функция  задана таблицей своих значений, где 

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | … |  | – *n+*1 значение |
|  |  |  |  | … |  |

Требуется найти полином

 (1.1)

степени *n*, принимающий в заданных точках , те же значения, что и функция , т. е. такой, что

. (1.2)

Геометрически это означает, что нужно найти кривую , которая проходит через заданную систему точек на плоскости. (рис. 1.1).

Поскольку степень полинома известна, то поиск полинома сводится к нахождению набора его коэффициентов .

Рис. 1.1

*Y*

*X*









**…**





**…**











Сформулированная задача называется задачей интерполяции. Функция  называется интерполируемой функцией. Полином  называется интерполяционным полиномом. Формулы для построения полинома  называются интерполяционными формулами. Точки  называются узлами интерполяции.

Полученный полином  обычно используют в случае приближенного вычисления значений функции  для значений аргумента , отличных от узлов интерполяции. Процесс вычисления значений функции  в точках, отличных от узлов интерполяции, называется интерполированием функции .

При этом различают интерполирование в узком смысле, если , и экстраполирование, когда .

Замена функции  ее интерполяционным полиномом  может потребоваться не только тогда, когда известна лишь таблица ее значений (например, полученная в результате эксперимента), но и в том случае, когда аналитическое выражение для функции  известно, однако является слишком сложным и неудобным для дальнейших математических преобразований (интегрирования, дифференцирования).

Определение коэффициентов интерполяционного полинома

Воспользовавшись выражением (1.1) для искомого полинома  и условием (1.2), получим систему линейных уравнений относительно неизвестных коэффициентов 

 (1.3)

Систему уравнений (1.3) представим в матричном виде, введя соответствующие обозначения.

 (1.4)

Система уравнений (1.3) приобретает вид

 (1.5)

Откуда  (1.6)

Отметим, что задача интерполяции в описанной выше постановке имеет единственное решение. Это означает, что не существует двух различных наборов коэффициентов , удовлетворяющих условию (1.2). Это следует из того, что искомые коэффициенты полинома определяются как решение системы линейных уравнений, имеющей единственное решение.

Интерполяционный полином Лагранжа

Французский математик Лагранж предложил способ построения интерполяционного полинома без предварительного вычисления коэффициентов , т. е. без решения системы уравнений (1.3).

Будем искать интерполяционный полином, который в данном случае обозначим через , в виде

(1.7)

Неизвестные коэффициенты  определим из условия

.

Последовательно полагая , получим

.



…………………….







…………………………..







Подставляя найденные значения коэффициентов  в выражение (1.7) для многочлена , получим

 (1.8)

Полученный таким образом полином называется интерполяционным полиномом Лагранжа.

Следует отметить, что полином Лагранжа  просто представляет собой другую форму записи рассмотренного ранее полинома , что следует из единственности решения задачи интерполяции.

Выражение для полинома Лагранжа  может быть легко преобразовано к виду  путем группировки коэффициентов при соответствующих степенях аргумента .

Пример 1.



|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

.

Обозначая  и , получаем

.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 | 2 | 3 |
|  | 2 | 8 | 6 |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |





Достоинства полинома Лагранжа:

* график интерполяционного многочлена Лагранжа проходит через каждую точку массива;
* конструируемая функция легко описывается (число подлежащих определению коэффициентов интерполяционного многочлена Лагранжа на сетке *m* равно *т+* 1);
* построенная функция имеет непрерывные производные любого порядка;
* заданным массивом интерполяционный многочлен определен од­нозначно.

Недостатки полинома Лагранжа:

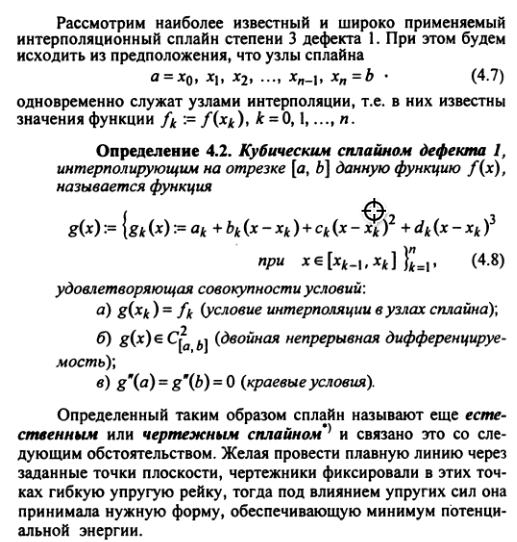
* степень интерполяционного многочлена Лагранжа зависит от числа узлов сетки, и чем больше это число, тем выше степень интерполяционного многочлена и, значит, тем больше требуется вычислений;
* изменение хотя бы одной точки в массиве требует полного пере­счета коэффициентов интерполяционного многочлена Лагранжа.

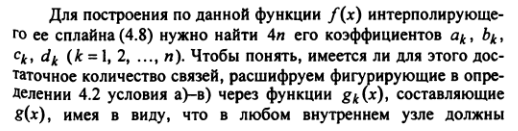
## 12. Построение кривых. Интерполяционный кубический сплайн. Реализация

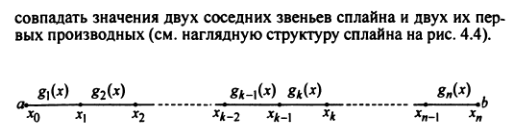
Пусть на отрезке задана упорядоченная система несовпадающих точек  ().

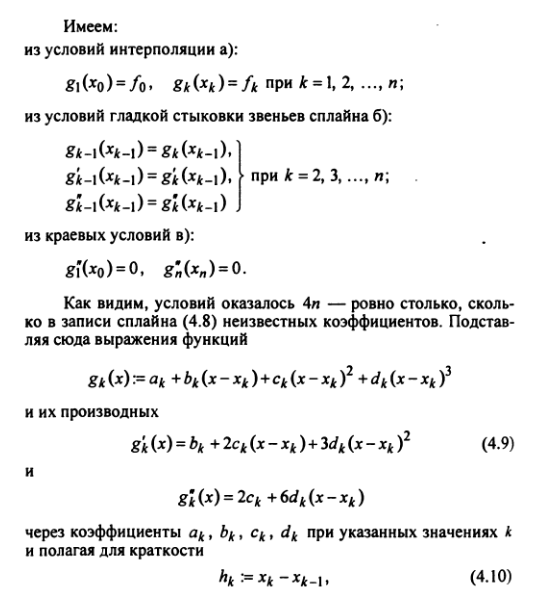
Определение. Сплайном ****называется определенная на  функция, принадлежащая классу  раз непрерывно дифференцируемых функций, такая, что на каждом промежутке – это многочлен -й степени. Разность между степенью сплайна  и показателем его гладкости  называется дефектомсплайна.

Если сплайн ****строится по некоторой функции так, чтобы выполнялись условия то такой сплайн называется интерполяционным сплайномдля функции ; при этом узлы сплайна,вообще говоря, могут не совпадать с узлами интерполяции .



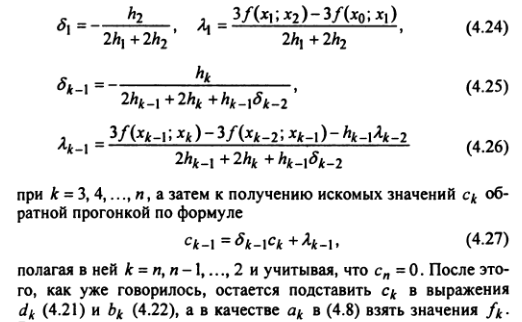






получим систему уравнений для определения неизвестных коэффициентов.

Полученная система решается методом прогонки.





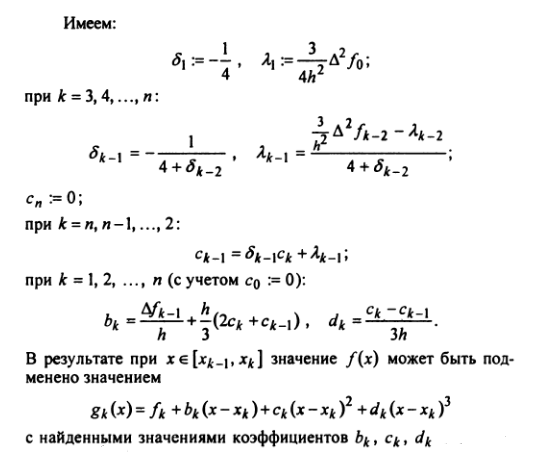


Где



Рассмотрим случай равноотстоящих узлов

Тогда  и расчетные формулы для коэффициентов полинома принимают вид:



Здесь  и  – конечные разности второго порядка.

Понятие о конечных разностях

Пусть 

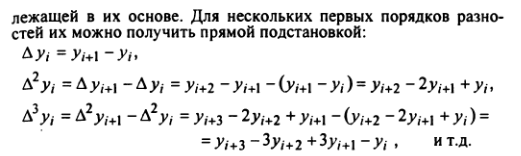
Конечные разности первого порядка определяются как



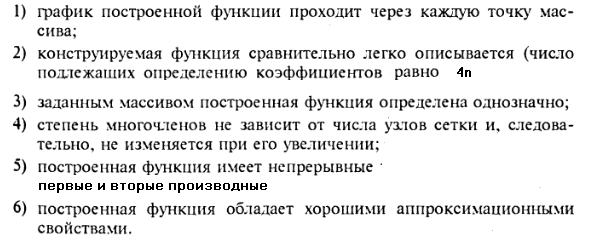
Конечные разности второго порядка определяются как







Достоинства кубической сплайн-интерполяции:



## 13. Построение кривых. Геометрические сплайны. Кривая Безье. Геометрический алгоритм построения кривой Безье. Реализация

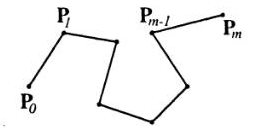
Во многих задачах требование того, чтобы конструируемая кривая однозначно проектировалась соответственно на прямую, является слишком жестким. Расширяя допустимые классы кривых, естественно обратиться и к более общему способу описания их частичных фрагментов. В качестве нового способа задания кривых удобно использовать парамет­рический способ.

Формулировка задачи: по заданному множеству вершин



с учетом их нумерации построить гладкую кривую, которая, плавно изменяясь, последовательно проходила бы вблизи этих вершин и удовлетворяла некоторым дополнительным условиям. Эти условий могут иметь различный характер. Например, можно потребовать, что­бы искомая кривая проходила через все заданные вершины или, про­ходя через заданные вершины, касалась заданных направлений, явля­лась замкнутой или имела заданную регулярность и т. п.

При отыскании подходящего решения задачи приближения важ­ную роль играет ломаная, звенья ко­торой соединяют соседние вершины заданного набора. Эту ломаную на­зывают контрольной или опорной, а ее вершины – контрольными или опорными.



Во многих случаях она довольно точно показывает, как будет проходить искомая кривая, что особенно полезно при решении задачи сглаживания. Каж­дая вершина заданного массива является либо внутреннейлибо гранич­ной(концевой). В массиве Р вершины Р1 …, Pm-1 внутренние, а вершины Р0 и Рm - граничные (концевые).

Никаких ограничений на множество вершин не накладывается - они могут быть заданы как на плоскости, так и в пространстве, их взаимное расположение может быть совершенно произвольным, некоторые из вершин могут совпадать и т. д. Поэтому описание нуж­ной кривой ищут в следующем виде:

 (1)

где  - некоторые функциональные коэффициенты, подлежащие определению.

Если количество вершин в заданном множестве Р достаточно ве­лико, то найти универсальные функциональные коэффициенты, как правило, довольно затруднительно. Если универсальные коэффициенты  все же найдены, то часто оказывается, что они наряду с нужными свойствами обладают и такими, которые не всегда удовлет­ворительно согласуются с ожидаемым поведением соответствующей кривой (например, кривая, описываемая уравнением (1) с этими коэффициентами, может осциллировать или заметно отклоняться от заданного множества).

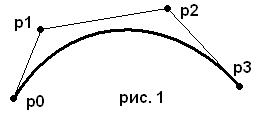
Для успешного решения поставленной задачи приближения, весьма удобно привлечь кривые, составленные из элементарных фраг­ментов. В случае, когда эти элементарные фрагменты строятся по еди­ной сравнительно простой схеме, такие составные кривые принято называть сплайновыми кривыми.

Параметрические уравнения каждого элементарного фрагмента ищутся в виде (1) с той лишь разницей, что всякий раз привлекается только часть заданных вершин множества Р. Для описания элементарных кривых и вычисления их геометри­ческих характеристик (информация о которых необходима при состы­ковке) в качестве функциональных коэффициентов обычно исполь­зуются многочлены невысоких степеней, 2-й или 3-й, в первую очередь потому, что они сравнительно просто вычисляются. Наибольшее распространение получили методы конструирования составных кривых, в которых используются кубические многочлены.

Кривая Безье

Кривая Безье задается несколькими точками в N-мерном пространстве (классический вариант, описываемый в литературе – 4 точки, но мы, для удобства, помимо этого случая будем использовать вырожденные случаи из 3-х точек, а так же пользоваться введением дополнительных вершин), но нам так много не нужно, поэтому ограничимся тремя измерениями (хотя все приведенные алгоритмы будут справедливы и для векторов другой мерности).

Для начала рассмотрим самый простой случай (его иллюстрация этого случая приведена на рис. 1) – когда у нас 4 вершины.



Здесь p0 и p3 – начальная и конечная точки соответственно, а p1 и p2 – промежуточные точки. Разности же вершин p1-p0 и p2-p3 определяют крутизну кривой в этих точках. Сама же кривая задается следующим уравнением:

C(s) = p0 \* ( 1 – s )3 + 3 \* p1 \* s \* ( 1 – s )2 + 3 \* p2 \* s2 \* ( 1 – s ) + p3 \* s3

Несложно показать, что

C( 0 ) = p0

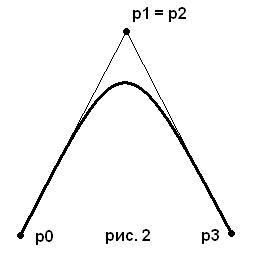
C( 1 ) = p3

C’( 0 ) = 3 \* ( p1 - p0 )

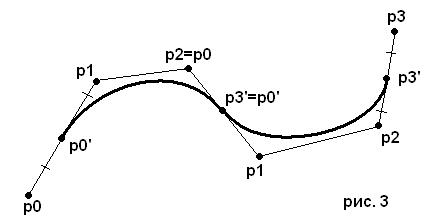
C’( 1 ) = 3 \* ( p3 – p2 )

Собственно как я и говорил – кривая проходит через конечные точки + имеется информация о крутизне кривой.

Но можно построить кривую Безье и по трем точкам. В этом случае p1 и p2 будут одинаковые. Правда в этом случае мы лишимся возможности перекручивать нашу кривую, плюс сама кривая будет не такая гладкая, как если бы она строилась по четырём точкам (рис. 2).



Наверняка с кривыми из 4 вершин Вы быстро наиграетесь и захотите задавать более длинные и изощренные кривые с петлями, изломами… Из 4 точек их уже не составить, поэтому придется стыковать их. Но как это осуществить? Если просто сделаем так, чтобы последняя вершина первой кривой совпадала с первой вершиной второй кривой, то мы получим излом. Это связано с тем, что производная меняется скачком – при подходе к концу кривой она у нас выгнута в одну сторону, а при проходе точки стыка она у нас мгновенно выгибается в другую сторону. Что бы этого не произошло, нам придется контролировать производную в точках стыка кусков кривых Безье. Вручную это делать как-то тоскливо, а автоматически не получиться, поэтому придется исхитриться. Собственно ничего сложного мы делать не будем – все детали Вы можете увидеть на рис. 3.



## 14. Мировые и экранные координаты. Алгоритм пересчёта мировых 2D-координат в оконные. Вывод и реализация

Мировые – система координат, которая вводится нужным образом в реальном мире и объекты описываются в данной системе координат**.**

Экранные – координаты на экране, всегда двумерные.













0









P



0























Введем обозначения

|  |  |
| --- | --- |
|  | – ширина области отображения в оконных координатах |
|  | – ширина области отображения в мировых координатах |
|  | – высота области отображения в оконных координатах |
|  | – высота области отображения в мировых координатах |



Или



, 



или



где

, , 

Перепишем пару рассматриваемых равенств в виде





,

, , 

## 15. Физическая и логическая системы координат. Режимы отображения (без настройки параметров). Алгоритм преобразования координат в GDI и его связь с алгоритмом пересчёта мировых координат в оконные

Физические координаты, как это следует из названия, имеют непосредственное отношение к физическому устройству вывода. В качестве единицы измерения длины в системе физических координат всегда используется пиксел. Если устройством вывода является экран монитора, физические координаты обычно называют экранными координатами.

Логические координаты передаются функциям GDI, выполняющим рисование фигур или вывод текста. Используемые единицы измерения зависят от режима отображения.

При отображении GDI преобразует логические координаты в физические. Способ преобразования зависит от режима отображения и других атрибутов контекста отображения, таких как расположение начала системы координат для окна, расположение начала системы физических координат, масштаб осей для окна и масштаб осей физических координат.

Физическая система координат

На рис. показана физическая система координат для экрана видеомонитора.

(0,0)

X

X

(x,y)

Начало этой системы координат располагается в левом верхнем углу экрана. Ось X направлена слева направо, ось Y - сверху вниз. В качестве единицы длины в данной системе координат используется пиксел.

Логическая система координат

Приложения Windows могут использовать одну из нескольких логических координат, устанавливая соответствующий режим отображения в контексте отображения. При этом, как мы уже говорили, можно использовать любое направление координатных осей и любое расположение начала координат. Например, возможна система координат, в которой задаются положительные и отрицательные координаты по любой оси.

Y

Y

(x,y)

Для установки режима отображения, непосредственно определяющего направление осей и размер логической единицы системы координат, используется функция SetMapMode:

int SetMapMode(HDC hdc, int nMapMode);

Параметр nMapMode может принимать одно из следующих значений.

|  |  |  |  |
| --- | --- | --- | --- |
| ***Режим отображения*** | ***Направление оси X*** | ***Направление***  ***оси Y*** | ***Размер одной логической единицы*** |
| **MM\_TEXT** | Вправо | Вниз | 1 пиксел |
| **MM\_LOMETRIC** | Вправо | Вверх | 0,1 мм |
| **MM\_HIMETRIC** | Вправо | Вверх | 0,01 мм |
| **MM\_LOENGLISH** | Вправо | Вверх | 0,01 дюйм |
| **MM\_HIENGLISH** | Вправо | Вверх | 0,001 дюйм |
| **MM\_TWIPS** | Вправо | Вверх | 1/1440 дюйма |
| **MM\_ISOTROPIC** | Можно выбирать | Можно выбирать | Произвольный, одинаковый для осей X и Y |
| **MM\_ANISOTROPIC** | Можно выбирать | Можно выбирать | Произвольный, может быть разный для осей X и Y |

С помощью функции GetMapMode приложение может в любой момент времени определить номер режима отображения, выбранный в контекст отображения hdc:

int GetMapMode(HDC hdc);

Преобразование координат

Приложение, вызывая для рисования функции GDI, указывает логические координаты. Перед выводом GDI преобразует их в физические с использованием следующих формул:





Org – Origin – начало координат

В этих формулах используются следующие обозначения:

xWindow, yWindow – логические координаты по оси X и Y соответственно.

xViewport , yViewport – физические (экранные) координаты по оси X и Y соответственно.

Таким образом, логические координаты (xWindow,yWindow) преобразуются в физические координаты (xViewport,yViewport)**.**

xViewOrg, yViewOrg – определяют расположение начала физической системы координат, по умолчанию xViewOrg = 0 и yViewOrg = 0

Приложение может сместить начало физической системы координат, изменив значения переменных xViewOrg и yViewOrg.

xWinOrg, yWinOrg – определяют расположение начала логической системы координат, по умолчанию xWinOrg = 0 и yWinOrg = 0

Приложение может сместить начало логической системы координат, изменив значения переменных xWinOrg и yWinOrg.

xViewExt, xWinExt – задают масштаб, который используется в процессе преобразования координат по оси x.

yViewExt, yWinExt – задают масштаб, который используется в процессе преобразования координат по оси y.

Эти масштабы зависят от установленного режима отображения. Приложения могут изменить его только в режимах MM\_ISOTROPICи MM\_ANISOTROPIC , для остальных режимов отображения используются фиксированные значения.

## 16. Режимы отображения и настройка их параметров (функции класса CDC MFC). Функция *SetMyMode*, назначение, параметры и реализация

Для установки режима отображения, непосредственно определяющего направление осей и размер логической единицы системы координат, используется функция SetMapMode:

int SetMapMode(HDC hdc, int nMapMode);

Параметр nMapMode может принимать одно из следующих значений.

|  |  |  |  |
| --- | --- | --- | --- |
| **Режим отображения** | **Направление оси X** | **Направление**  **оси Y** | **Размер одной логической единицы** |
| MM\_TEXT | Вправо | Вниз | 1 пиксел |
| MM\_LOMETRIC | Вправо | Вверх | 0,1 мм |
| MM\_HIMETRIC | Вправо | Вверх | 0,01 мм |
| MM\_LOENGLISH | Вправо | Вверх | 0,01 дюйм |
| MM\_HIENGLISH | Вправо | Вверх | 0,001 дюйм |
| MM\_TWIPS | Вправо | Вверх | 1/1440 дюйма |
| MM\_ISOTROPIC | Можно выбирать | Можно выбирать | Произвольный, одинаковый для осей X и Y |
| MM\_ANISOTROPIC | Можно выбирать | Можно выбирать | Произвольный, может быть разный для осей X и Y |

Режим MM\_TEXT

Режим отображения MM\_TEXT устанавливается в контексте отображения по умолчанию. Для этого режима формулы преобразования координат упрощаются:

xViewport = (xWindow - xWinOrg) + xViewOrg

yViewport = (yWindow - yWinOrg) + yViewOrg

Так как по умолчанию xViewOrg=0, yViewOrg=0, xWinOrg=0, yWinOrg=0, то

xViewport = xWindow

yViewport = yWindow

Соответствующая система координат представлена на рис. ниже (начало системы координат расположено точно в левом верхнем углу внутренней области окна, рисунок иллюстрирует только направление координатных осей).

Y

X

(x,y)

Так как в формуле преобразования не присутствуют переменные xViewExt, yViewExt, xWinExt и yWinExt, в данном режиме преобразования невозможно изменить масштаб осей координат. Поэтому логическая единица длины в режиме отображения MM\_TEXT равна физической, т. е. одному пикселу.

Тем не менее, приложение может изменить смещение физической или логической системы координат, изменив, соответственно, значение пар переменных (xViewOrg, yViewOrg) и (xWinOrg,yWinOrg). Для установки смещения можно использовать функции SetViewportOrgEx и SetWindowOrgEx.

BOOL SetViewportOrgEx(

HDC hdc, // контекст отображения

int nXOrigin, // новое значение для xWinOrg

int nYOrigin, // новое значение для yWinOrg

LPPOINTlpPoint // указатель на структуру POINT

);

**virtual CPoint SetViewportOrg(** **int** *x*, **int** *y* **); //** Метод класса CDC MFC

**CPoint SetViewportOrg(POINT** *point* **); //** Метод класса CDC MFC

**CPoint SetViewportOrg(CPoint** *point* **); //** POINT или CPoint

BOOL SetWindowOrgEx(

HDC hdc, // контекст отображения

int nXOrigin, // новое значение для xWinOrg

int nYOrigin, // новое значение для yWinOrg

LPPOINTlpPoint // указатель на структуру POINT

);

**CPoint SetWindowOrg(int** *x*, int *y* **); //** Метод класса CDC MFC

**CPoint SetWindowOrg( POINT** *point* **); //** Метод класса CDC MFC

**CPoint SetWindowOrg( CPoint** *point* **); //** POINT или CPoint

В структуру, адрес которой передается через параметр lpPoint, записываются старые координаты начала системы координат. Если они не нужны, можно положить lpPoint=NULL.

Обе функции возвращают TRUE в случае успеха и FALSE при возникновении ошибки.

В любой момент времени можно определить расположение начала физических или логических координат, если воспользоваться функциями GetViewportOrgEx и GetWindowOrgEx.

Метрические режимы отображения

Режим MM\_LOMETRIC, наряду с режимами MM\_HIMETRIC, MM\_LOENGLISH, MM\_HIENGLISH и MM\_TWIPS, относится к метрическим режимам. Эти режимы отображения позволяют использовать привычные единицы измерения, такие как миллиметры и дюймы.

В метрических режимах отображения используются полные формулы преобразования координат, приведенные выше в разделе "Преобразование координат". В этих формулах приложение может изменять переменные, определяющие смещение начала физической или логической системы координат xViewOrg, yViewOrg, xWinOrg и yWinOrg.

Приложение не может изменить значения переменных xViewExt, yViewExt, xWinExt и yWinExt, от которых зависит масштаб по осям координат. Отношения xViewExt/xWinExt и yViewExt/yWinExt имеют фиксированное значение для каждого из метрических режимов отображения.

Заметим, что для этих режимов отношение yViewExt/yWinExt имеет отрицательный знак, в результате чего ось Y оказывается направленной снизу вверх.

-------------------------------------------------------------------------------------------------------------------

Обратим внимание на одно важное обстоятельство, связанное с использованием метрических режимов отображения.

Сразу после переключения в метрический режим отображения система координат примет достаточно странный вид (рис. ниже).

Y(-)

X(+)

(x,y)

(0,0)

Ось X, как и следовало ожидать, окажется направленной слева направо, а ось Y - снизу вверх. Точка с координатами (0,0) будет находиться в верхнем левом углу экрана, поэтому для того чтобы нарисовать что-нибудь в такой системе координат, вам придется для y-координаты графических объектов использовать отрицательные числа. Для того чтобы система координат приняла более удобный вид, можно переместить начало физических координат в нижний левый угол окна или в центр окна.

Прежде, чем выполнять перемещение начала координат, следует определить размеры внутренней области окна. Это можно сделать при обработке сообщения WM\_SIZE:

---------------------------------- Для MFC -----------------------------------------------------

int dxClient, dyClient; // Объявлены в классе CChildView

void CMainFrame::OnSize(UINT nType, int cx, int cy)

{

CFrameWnd::OnSize(nType, cx, cy);

m\_wndView.dxClient=cx; // m\_wndView - переменная типа CChildView

m\_wndView.dyClient=cy;

}

---------------------- Для API --------------------------------------------------------------------

static short cxClient, cyClient;

....

case WM\_SIZE:

{

cxClient = LOWORD(lParam);

cyClient = HIWORD(lParam);

....

return 0;

}

-------------------------------------------------------------------------------------------------------

Для того чтобы расположить начало координат в левом нижнем углу окна, следует вызвать функцию SetViewportOrgEx, передав ей новые координаты начала физической системы координат (0,cyClient):

SetViewportOrgEx(hdc, 0, cyClient, NULL);

Полученная в результате система координат показана на рис.:

Y(+)

X(+)

(0,0)

Аналогичным образом можно расположить начало системы координат в середине окна (рис.6), обеспечив возможность использования положительных и отрицательных координат вдоль оси X и Y:

SetViewportOrgEx(hdc, cxClient/2, cyClient/2,NULL);

Y(+)

X(+)

(0,0)

Пример

void CChildView::OnPaint()

{

CPaintDC dc(this); // device context for painting

if(Index==1)

{

dc.SetMapMode(MM\_LOMETRIC); // 0,1 мм

dc.SetViewportOrg(0,dyClient);

//SetViewportOrgEx(hdc, 0, dyClient, NULL); //API

dc.MoveTo(0,0);

dc.LineTo(1000,1000);

}

if(Index==2)

{

dc.SetMapMode(MM\_LOMETRIC); // 0,1 мм

dc.SetViewportOrg(dxClient/2,dyClient/2);

//SetViewportOrgEx(hdc, , dyClient/2,NULL); //API

dc.MoveTo(-500,0);

dc.LineTo(500,0); //Ось X

dc.MoveTo(0,-500);

dc.LineTo(0,500); //Ось Y

dc.MoveTo(0,0);

dc.LineTo(200,200);

}

}

Режимы MM\_ISOTROPIC и MM\_ANISOTROPIC

Режимы отображения MM\_ISOTROPIC (изотропный) и MM\_ANISOTROPIC (анизотропный) допускают изменение направления осей X и Y, а также изменение масштаба осей координат. В изотропном режиме отображения MM\_ISOTROPIC масштаб вдоль осей X и Y всегда одинаковый (т. е. для обоих осей используются одинаковые логические единицы длины). Анизотропный режим MM\_ANISOTROPIC предполагает использование разных масштабов для разных осей (хотя можно использовать и одинаковые масштабы).

Для изменения ориентации и масштаба осей предназначены функции SetViewportExtEx и SetWindowExtEx.

Функция SetWindowExtEx устанавливает для формулы преобразования координат значения переменных xWinExt и yWinExt:

BOOL SetViewportExtEx(

HDC hdc, // идентификатор контекста отображения

int nXExtent, // значение для xViewExt

int nYExtent, // значение для yViewExt

LPSIZE lpSize // указатель на структуру SIZE

);

**virtual CSize SetViewportExt(int** *cx*, **int** *cy* **); //** Метод класса CDC MFC

**CSize SetViewportExt( SIZE** *size* **); //** Метод класса CDC MFC

Функция SetViewportExtEx должна использоваться после функции SetWindowExtEx. Она устанавливает для формулы преобразования координат значения переменных xViewExt и yViewExt:

BOOL SetWindowExtEx(

HDC hdc, // идентификатор контекста отображения

int nXExtent, // значение для xWinExt

int nYExtent, // значение для yWinExt

LPSIZE lpSize // указатель на структуру SIZE

);

**virtual CSize SetWindowExt( int** *cx***, int** *cy* **); //** Метод класса CDC MFC

**CSize SetWindowExt(SIZE** *size* **); //** Метод класса CDC MFC

Старые значения переменных, определяющих масштаб преобразования, записываются в структуру SIZE, указатель на которую передается через параметр lpSize.

typedef struct tagSIZE {

LONG cx;

LONG cy;

} SIZE, \*PSIZE;

Если они не нужны, можно положить lpSise=**NULL**

Изотропный режим отображения удобно использовать в тех случаях, когда надо сохранить установленное отношение масштабов осей X и Y при любом изменении размеров окна, в которое выводится изображение.

Анизотропный режим удобен в тех случаях, когда изображение должно занимать всю внутреннюю поверхность окна при любом изменении размеров окна. Соотношение масштабов при этом не сохраняется.





















, 

, 

, 

, 

|  |  |
| --- | --- |
| void SetMyMode(HDC hdc, RECT RS,  RECT RW) // API  // hdc - контекст устройства  // RS - область в мировых координатах  // RW - область в оконных координатах  {  int dsx=RS.right-RS.left;  int dsy=RS.top-RS.bottom;  int xsL=RS.left;  int ysL=RS.bottom;  int dwx=RW.right-RW.left;  int dwy=RW.bottom-RW.top;  int xwL=RW.left;  int ywH=RW.bottom;  SetMapMode(hdc,MM\_ANISOTROPIC);  SetWindowExtEx(hdc,dsx, dsy, NULL);  SetViewportExtEx(hdc, dwx,-dwy, NULL);  SetWindowOrgEx (hdc, xsL, ysL, NULL);  SetViewportOrgEx(hdc,xwL, ywH, NULL);  } | void SetMyMode(CDC& dc, CRect& RS,  CRect& RW) // MFC  // dc - ссылка на класс CDC MFC  // RS - область в мировых координатах  // RW - область в оконных координатах  {  int dsx=RS.right-RS.left;  int dsy=RS.top-RS.bottom;  int xsL=RS.left;  int ysL=RS.bottom;  int dwx=RW.right-RW.left;  int dwy=RW.bottom-RW.top;  int xwL=RW.left;  int ywH=RW.bottom;  dc.SetMapMode(MM\_ANISOTROPIC);  dc.SetWindowExt(dsx,dsy);  dc.SetViewportExt(dwx,-dwy);  dc.SetWindowOrg(xsL,ysL);  dc.SetViewportOrg(xwL,ywH);  } |

## 17. Аффинные преобразования на плоскости. Преобразование системы координат: смещение, растяжение-сжатие, поворот

Пусть на плоскости задана система координат (СК) и точка  принадлежащая некоторому объекту. Система координат трансформируется в СК  путем ряда последовательных смещений и поворотов относительно своего исходного состояния. При этом точка (объект) остается неподвижной. Необходимо определить координаты точки (объекта) в системе координат  (рис. 1а, рис 1б).

|  |  |
| --- | --- |
|  |  |
| Рис. 1а | Рис. 1б |

В общем случае преобразование координат мочки при переходе от системы координат к системе координат определяется системой линейных уравнений:

 ,

где 

Выражение (1) представляет собой аффинное преобразование координат при переходе от системы координат к системе координат .

Обратный переход от СК к СК определяется как

,

Аффинное преобразование (1) удобно представить в матричном виде:



В компьютерной графике принято использовать однородные координаты, которые вводятся следующим образом. Точке  ставится в соответствие точка , а точке  точка .

Тогда переход от системы координат к системе координат в матричном виде можно записать как



или

,

где

, , 

Обратное преобразование

,

где



Параллельный сдвиг системы координат

,



Или



|  |  |
| --- | --- |
|  |  |
| Рис. 2а | Рис. 2б |

Здесь



Обратное преобразование

,

,

где



Растяжение – сжатие системы координат





Или

,

где



Обратное преобразование

,



Или

,

где

,

Поворот системы координат

|  |  |
| --- | --- |
|  |  |
| Рис. 3а | Рис. 3б |

Координаты точки  в системе координат :



В системе координат :



С учетом (23), получаем:



Или



,

где



Обратное преобразование

Обратное преобразование соответствует повороту системы координат

 на угол 



Или



,

где



## 18. Аффинные преобразования на плоскости. Преобразование объектов: смещение, растяжение-сжатие, поворот

Под преобразованием объектов будем понимать изменение координат точек, принадлежащих этому объекту при изменении его положения в некоторой системе координат.

Пусть в системе координат  некоторая точка перемещается из положения  в положение (рис. 4а, 4б) или обратно из положения  в положение .

|  |  |
| --- | --- |
|  |  |
| Рис. 4а | Рис. 4б |

Рассмотрим частные случаи аффинных преобразований объектов.

Сдвиг объекта

|  |  |
| --- | --- |
|  |  |
| Рис. 5 | Рис. 6 |

Из рис. 5 получаем

,



Или



Здесь



Обратное преобразование

Обратное преобразование позволяет вычислить старые координаты объекта  по новым 

,

,

где



Растяжение – сжатие объекта





Или

,

где



Обратное преобразование

,



Или

,

где

,

Поворот объекта вокруг центра координат

Координаты точки  в системе координат (рис. 6):



Координаты точки  в системе координат :



С учетом (46), получаем:



Или



,

где



Обратное преобразование

Обратное преобразование можно получить, если представить себе поворот объекта с координатами  на угол .



Или



,

где



Связь преобразований объектов с преобразованиями координат

Преобразование объектов и преобразование систем координат тесно связаны между собой. Движение объектов можно рассматривать как движение в обратном направлении соответствующей системы координат.

Такая относительность движения дает дополнительные возможности для моделирования и визуализации различных объектов.

Запишем соответствующие соотношения, основываясь на сравнении выражений (11) и (35) для сдвига

**,**

выражений (18) и (42) для растяжения – сжатия



выражений (27) и (51) для поворота



**CMatrix CreateTranslate2D(double dx, double dy);**

// Формирует матрицу для преобразования координат ОБЪЕКТА при

// его смещении на dx по оси X и на dy по оси Y в фиксированной системе

// координат

// --- ИЛИ ---

// Формирует матрицу для преобразования координат объекта при

// смещении начала системы координат на -dx оси X и на -dy по оси Y при

// фиксированном положении объекта

**CMatrix CreateRotate2D(double fi);**

// Формирует матрицу для преобразования координат ОБЪЕКТА при его повороте

// на угол fi (при fi>0 против часовой стрелки)в фиксированной системе координат

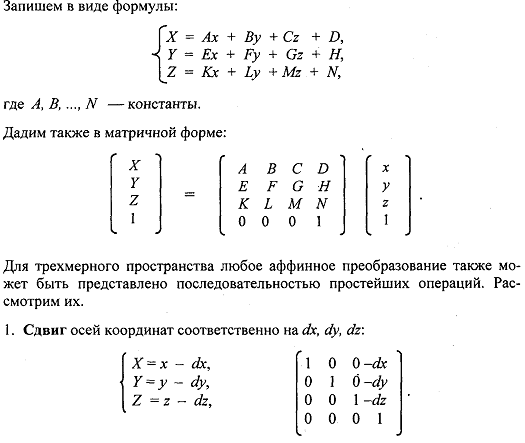
// --- ИЛИ ---

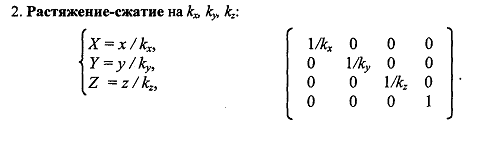
// Формирует матрицу для преобразования координат объекта при повороте начала

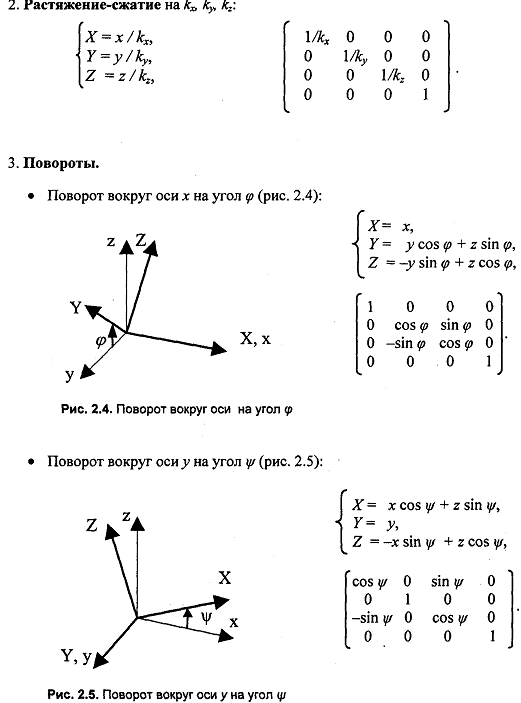
// системы координат на угол -fi при фиксированном положении объекта

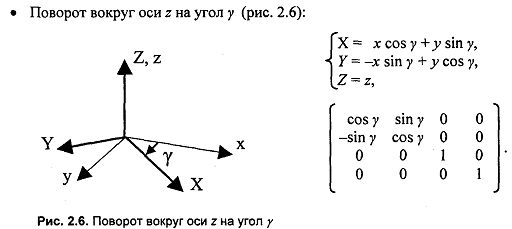
// fi - угол ***в градусах***

## 19. Аффинные преобразования в пространстве. Преобразование системы координат: смещение, растяжение-сжатие, повороты вокруг осей координат

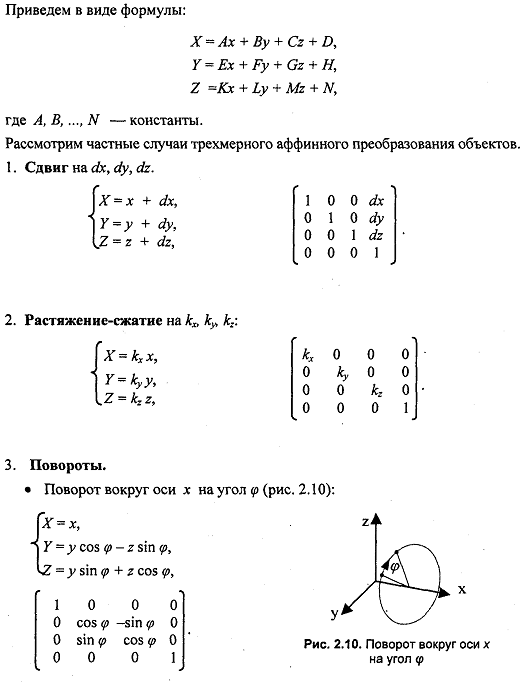


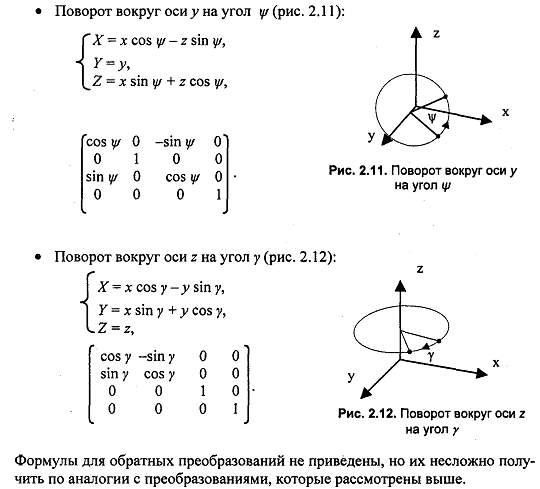






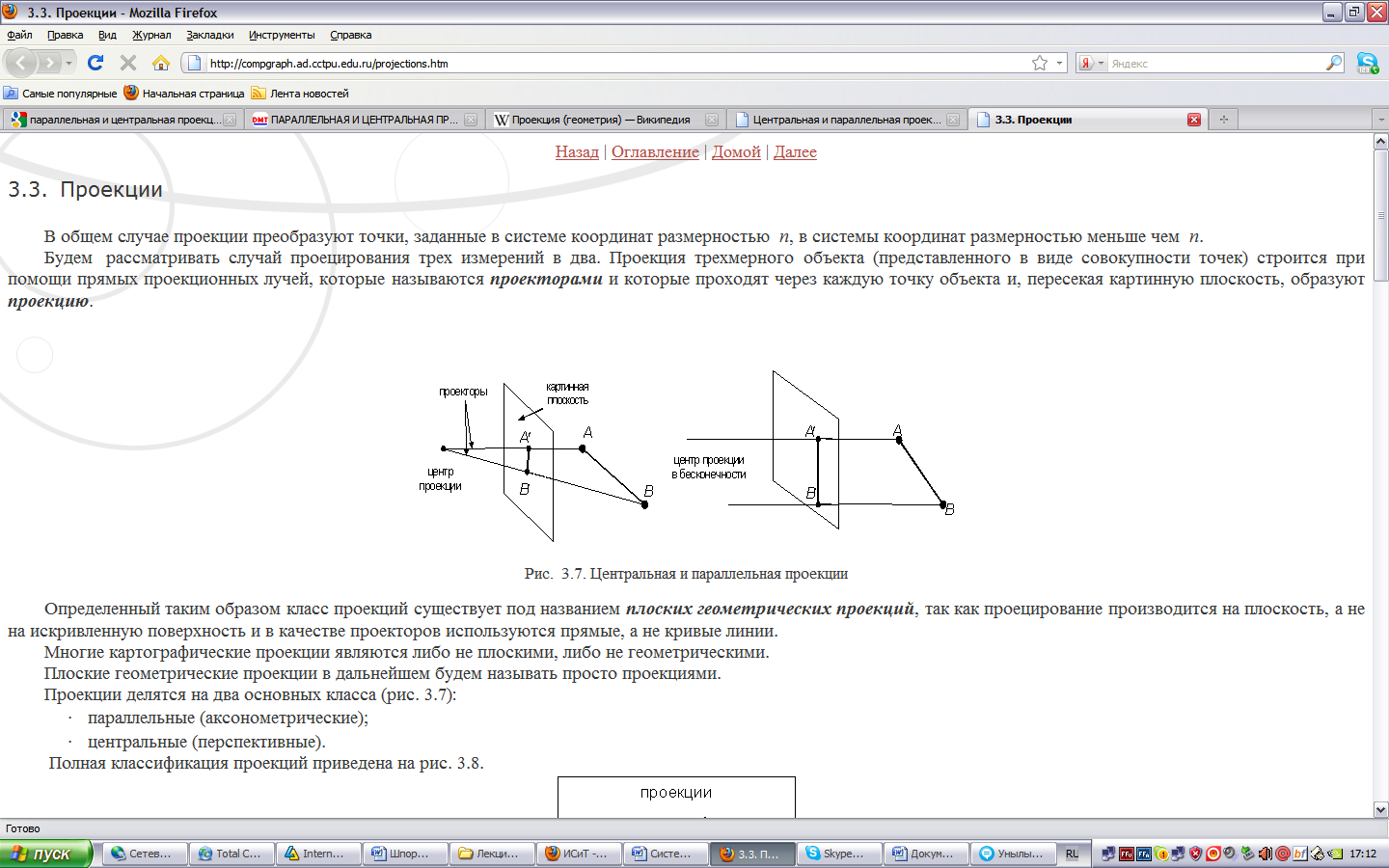
## 20. Аффинные преобразования в пространстве. Преобразования объектов: смещение, растяжение-сжатие, повороты вокруг осей координат



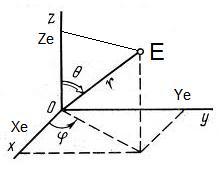


## 21. Основные типы проекций. Видовая система координат (вывод матрицы преобразования)

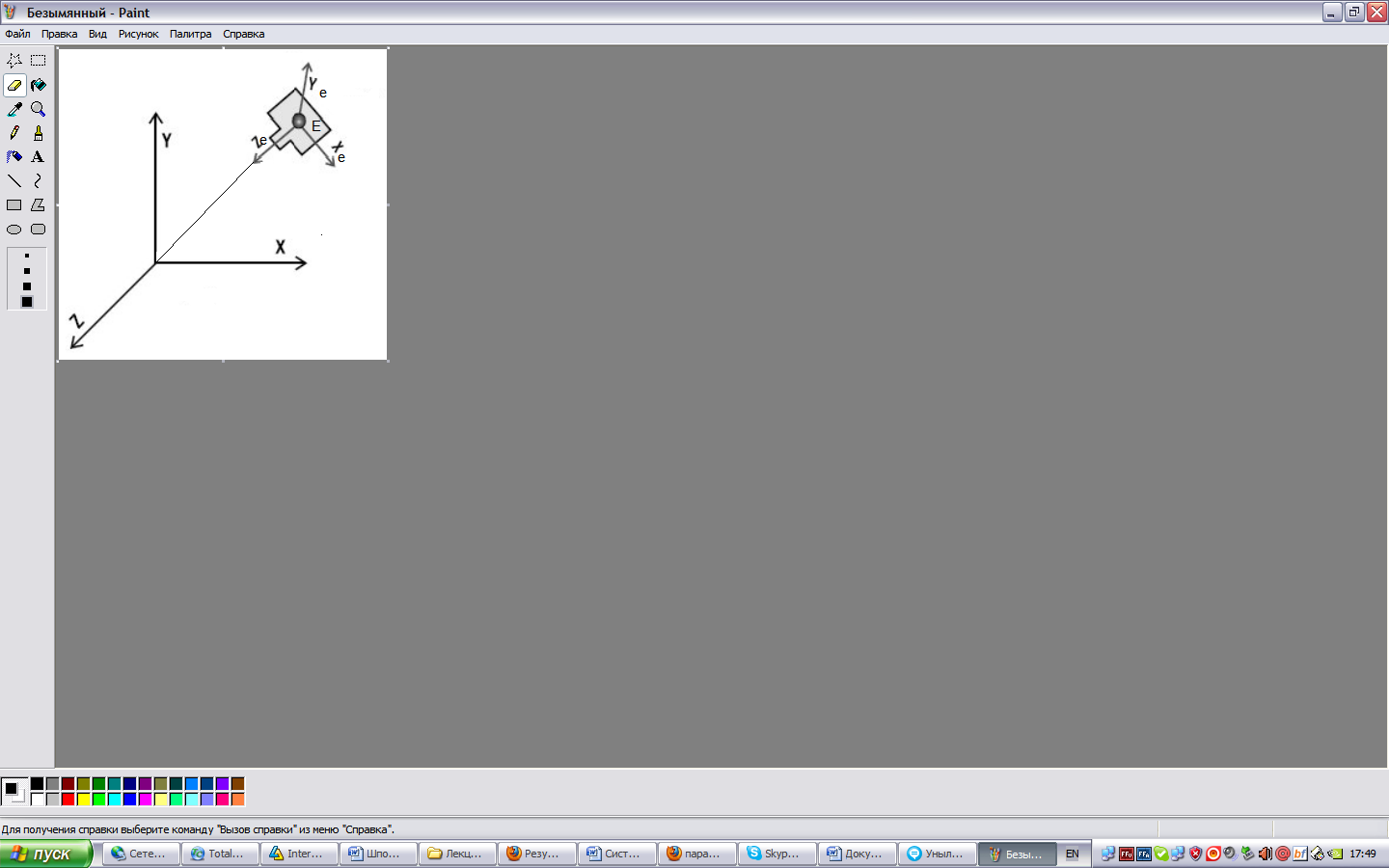
Картинная плоскость-плоскость, на которой выполняется проекция объекта. В компьютерной графике наиболее распространены центральные (рис. слева) и параллельные (рис. справа) проекции.

****

Видовая система координат

Для изображения объекта на экране необходимо преобразовать его координаты в некую другую СК, которая связана с точкой наблюдателя (положением камеры). Эта СК называется видовой. Мировая система координат традиционно является правосторонней, а видовая систем координат левосторонней.

Рассмотрим правую систему мировых координат. С помощью сферических координат зададим положение точки наблюдателя.

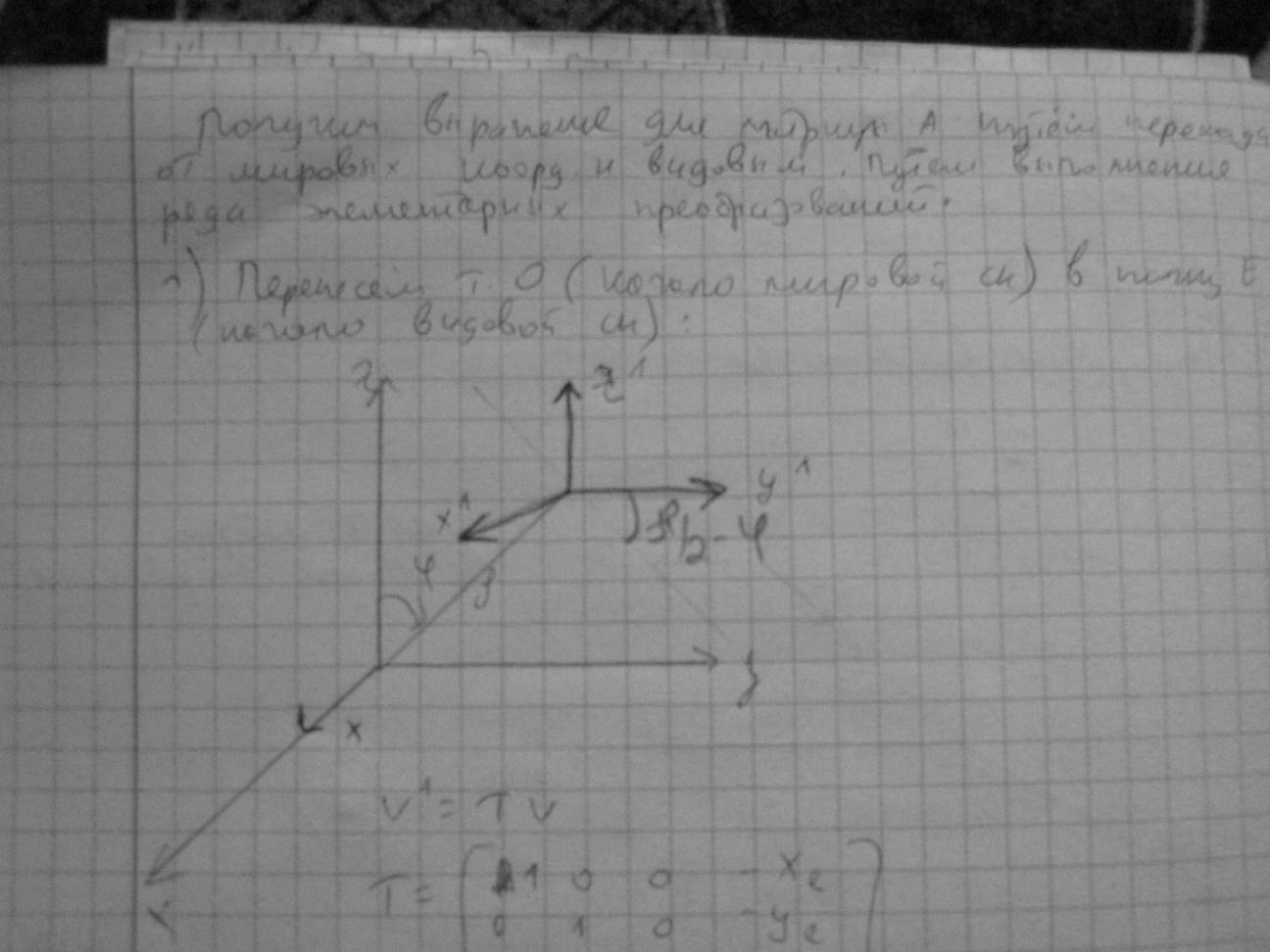
Изобразим видовую СК:

Ось Z смотрит в начало мировой СК. V(видовая)=A\*V(мировая)

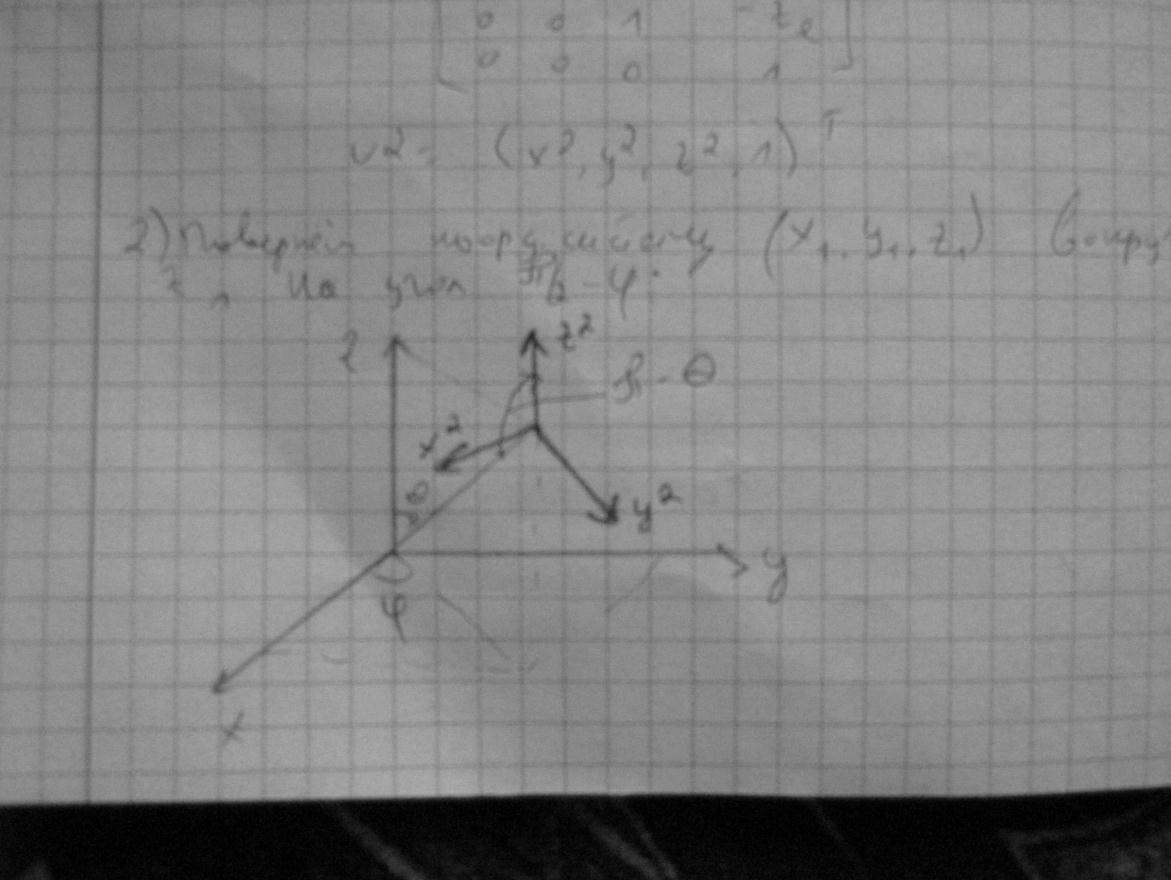
Получим выражение для матрицы А путем перехода от мировых координат к видовым. Путем выполнения ряда элементарных преобразований.

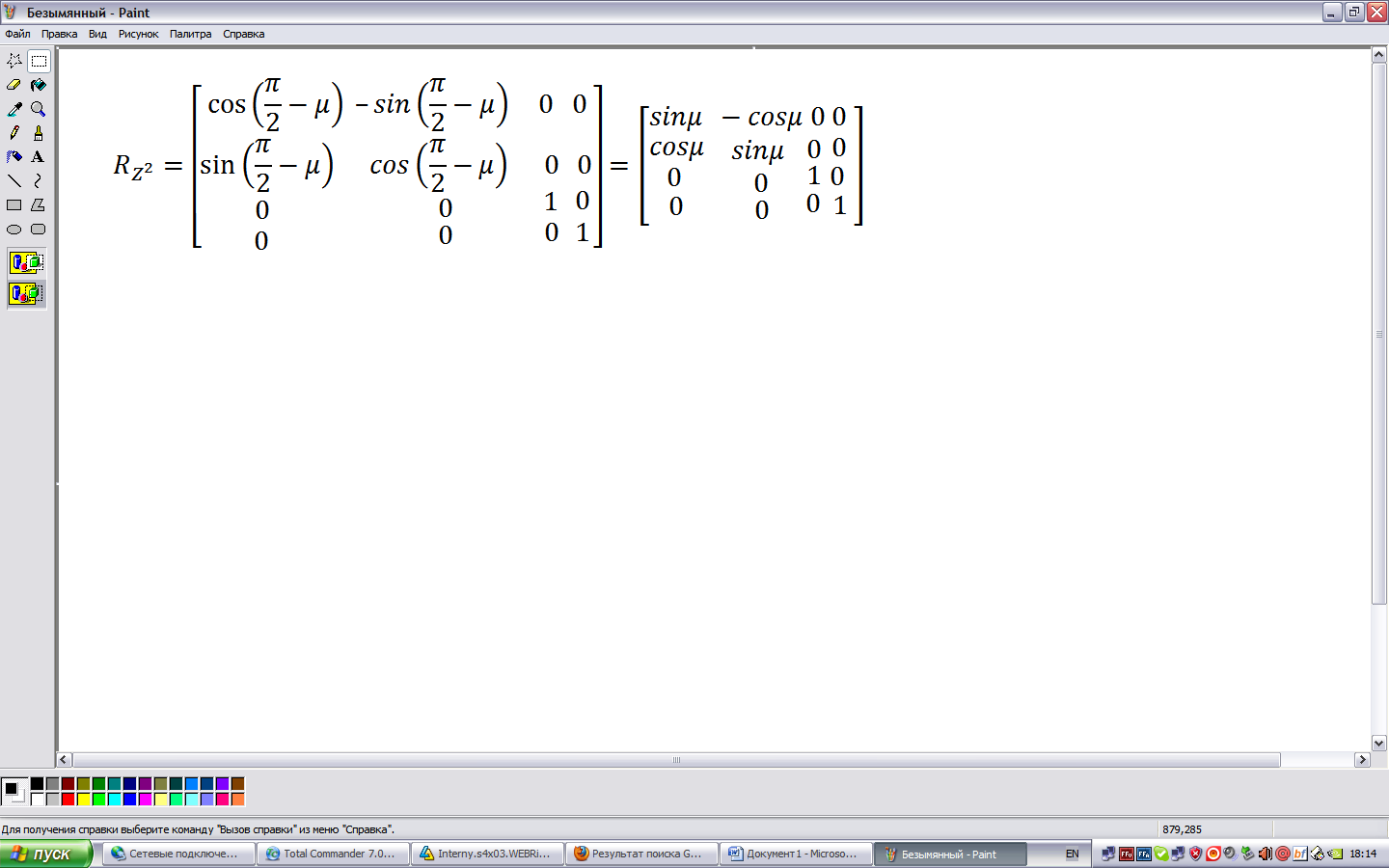
Перенесем точку О (начало мировой СК) в точку Е (начало видовой СК)

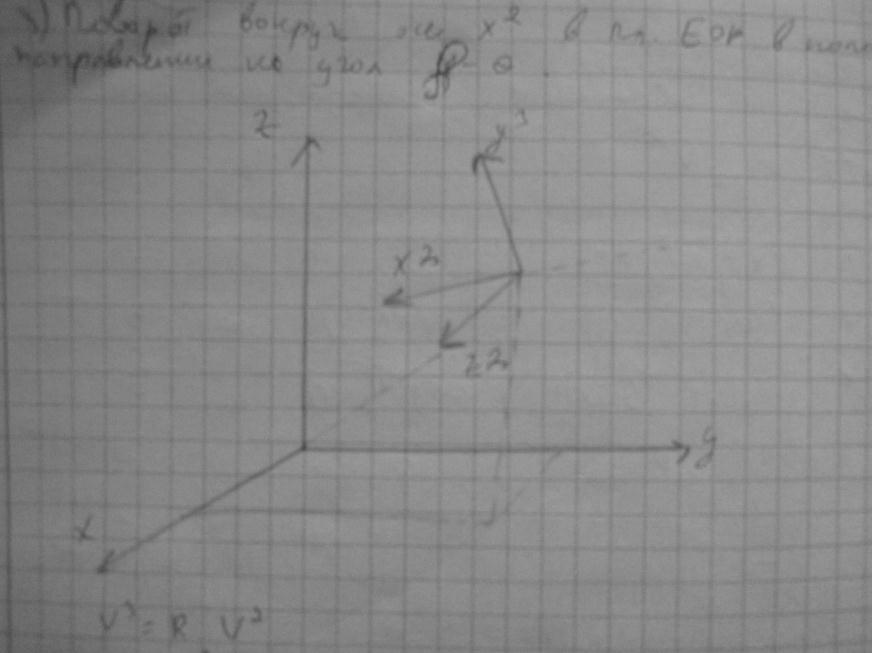
V^1=TV

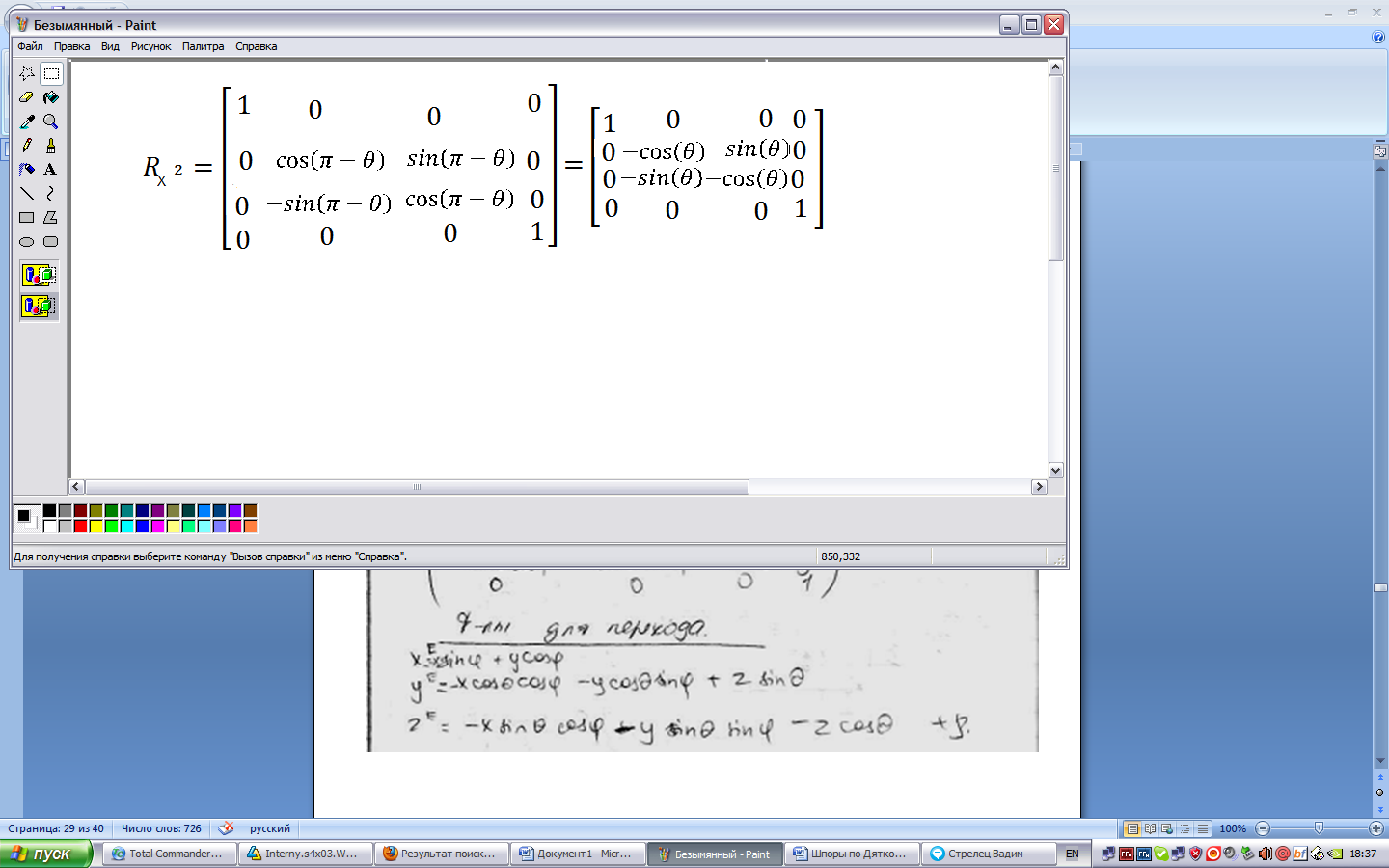


Повернем координатную систему (X1,Y1,Z1) вокруг оси Z1 на угол π/2-µ. Поворот выполняем в отрицательном направлении (против часовой стрелки) => матрица данного преобразования будет совпадать с преобразованием точки.

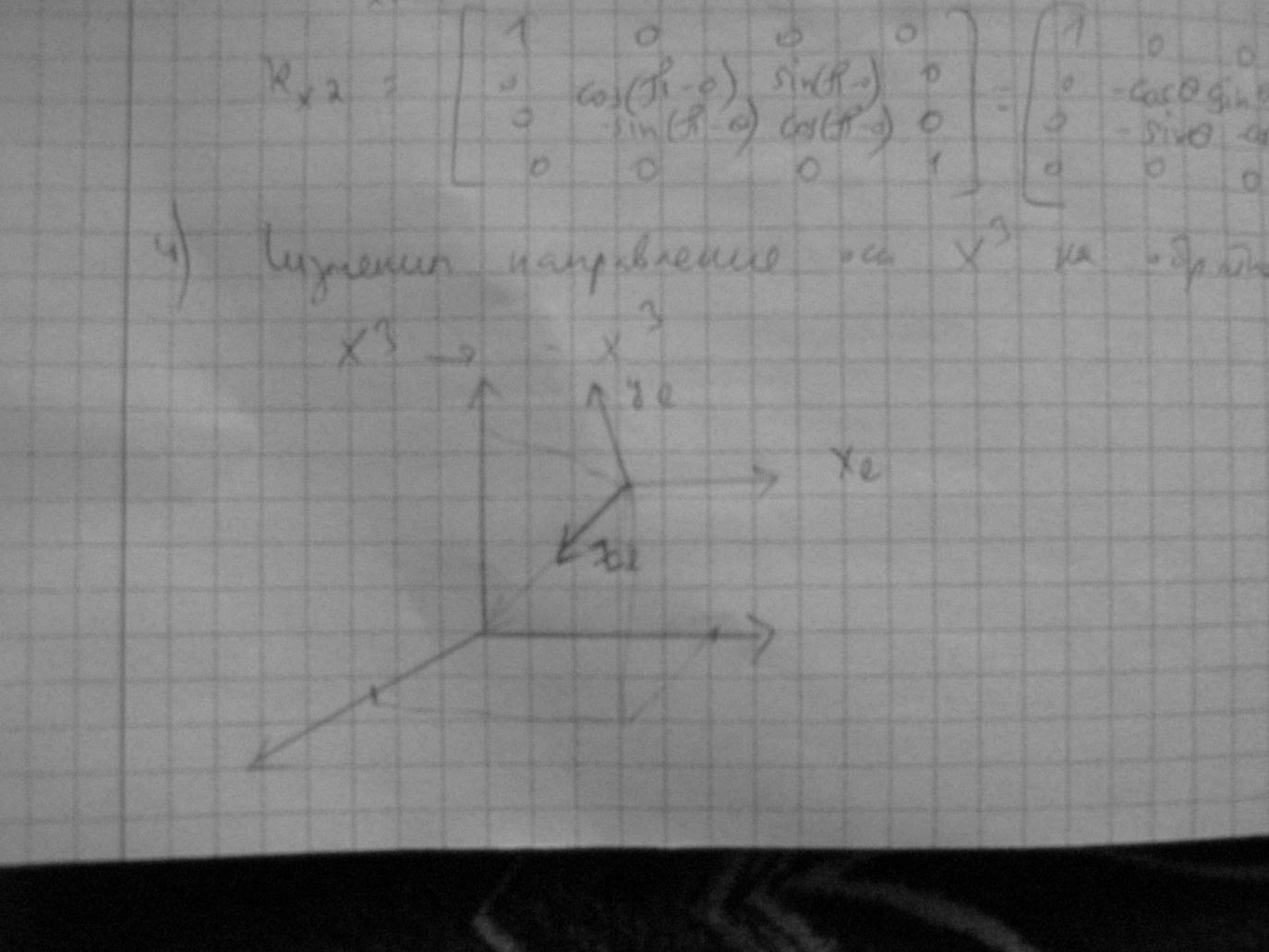
 ;

**

Поворот вокруг оси в положительном направлении на угол .

**

Изменим направление оси на обратное (-).



Вместо V3 подставим V2. Вместо V2 подставим V1

; ;

С учетом матрицы A:

Видовые координаты объекта представляют собой аксонометрическую проекцию этого объекта на координатную плоскость . Координаты можно непосредственно использовать для формирования изображения на экране.

## 22. Перспективные преобразования (вывод матрицы преобразования). Схема пересчёта координат при переходе от мировых 3D-координат к экранным























Картинная плоскость – плоскость проецирования в видовой системе координат.

М(xe, ye) – точка на плоскости проектирования P.

– координаты точки схода лучей.

При такой расстановке координат

– z координата объекта в видовой системе координат.

M’(xe’,ye’) – изображение точки M в картинной плоскости без учета перспективы.

Из подобия треугольников и получаем пропорцию:

*;* таккак *, ;*

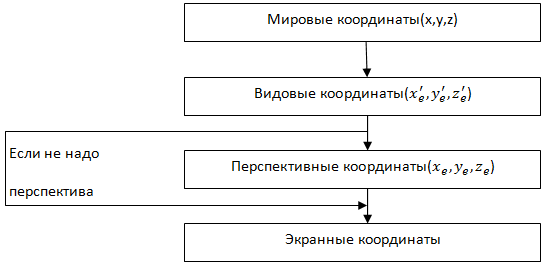
Аналогично их подобия треугольников и

Обозначим , тогда:

; ;

;

Т.е. преобразование координат при переходе от мировых к видовым включает следующие этапы:



## 23. Модели описания поверхностей

Аналитическая модель

Аналитической моделью будем называть описание поверхности математиче­скими формулами. В КГ можно использовать много разновидностей такого описания. Например, в виде функции двух аргументов *z =F(х, у).* Можно использовать уравнение *F (х, у, z)* = 0.

Зачастую используется параметрическая форма описания поверхности. За­пишем формулы для трехмерной декартовой системы координат *(х, у, z):*

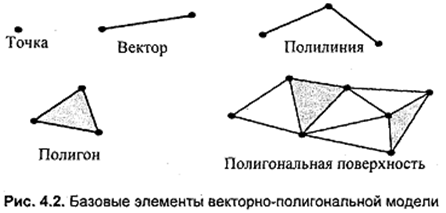


где s и t — параметры, которые изменяются в определенном диапазоне, функции *Fx*, *Fy* и *Fz* будут определять форму поверхности.

Преимущества параметрического описания — легко описывать поверхности, которые отвечают неоднозначным функциям, замкнутые поверхности.

Векторная полигональная модель

Для описания пространственных объектов здесь используются такие элементы: вершины; отрезки прямых (векторы); полигоны, полигональные поверхности(рис. 4.2).

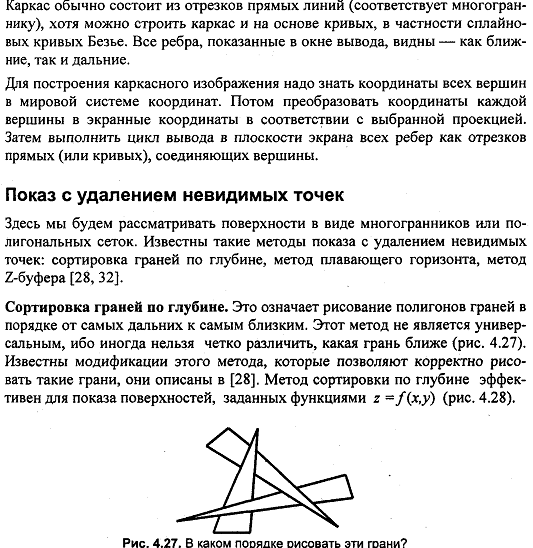


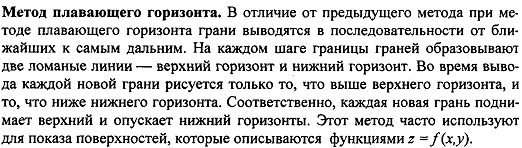
Элемент "вершина" (vertex) – главный элемент описания, все другие являются производными. При использовании трехмерной декартовой систем координаты вершин определяются как(xi, yi, zi).Каждый объект однозначно определяется координатами собственных вершин.

Вершина может моделировать отдельный точечный объект, размер которой не имеет значения, а также может использоваться в качестве конечных точек для линейных объектов и полигонов. Двумя вершинами задается вектор. Несколько векторов составляют полилинию. Полилиния может моделировать отдельный линейный объект, толщина которого не учитывается, а также может представлять контур полигона. Полигон моделирует площадный объект. Один полигон может описывать плоскую грань объемного объекта. Несколь­ко граней составляют объемный объект в виде полигональной поверхно­сти – многогранник или незамкнутую поверхность (в литературе часто употребляется название "полигональная сетка").

Векторную полигональную модель можно считать наиболее распространен­ной в современных системах трехмерной КГ.

## 24. Каркасная визуализация трёхмерных изображений. Принцип удаления невидимых граней для выпуклого многогранника. Реализация

****

****

## 25. Изображение поверхности z = f(x,y). Метод сортировки граней по глубине (алгоритм художника). Реализация

Поверхности определяется как множество точек, координаты которых удовлетворяют определённому виду уравнений:

F(x,\,y,\,z)=0\qquad (1)

Если функция F(x,\,y,\,z) непрерывна в некоторой точке и имеет в ней непрерывные частные производные, по крайней мере одна из которых не обращается в нуль, то в окрестности этой точки поверхность, заданная уравнением (1), будет правильной поверхностью.

Помимо указанного выше неявного способа задания поверхность может быть определена явно, если одну из переменных, например z, можно выразить через остальные:

z=f(x,y)\qquad (1')

Также существует параметрический способ задания.

Алгоритм художника

Алгоритм сортировки по глубине – поверхности объектов сортируются по удаленности от наблюдателя и заполняются соответствующими цветами, начиная с самой дальней.

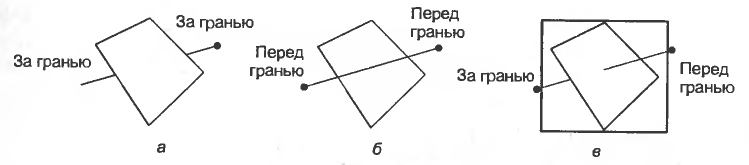
Алгоритм сортировки по глубине используется для удаления невидимых поверхностей. При применении алгоритма невидимых граней к множеству объектов удалено будет лишь около 50% невидимых линий. Нужен алгоритм, который удалял бы все невидимые линии независимо от количества объектов, их выпуклости и наличия криволинейных поверхностей.

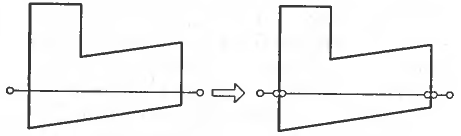
Один из таких алгоритмов действует следующим образом. Для каждого ребра (ребром объекта называется кривая пересечения соседних поверхностей, ограничивающих внутренний объем объекта) каждого объекта производится проверка, не закрыто ли оно гранями (гранями называются поверхности, ограничивающие объем объекта; площадь любой грани конечна, потому что все грани ограничиваются ребрами) каких-либо объектов. Закрытые части ребер последовательно исключаются до тех пор, пока не останется непроверенных поверхностей. Оставшиеся части ребер выводятся на экран. Реализация алгоритма включает несколько этапов.

1. Поверхности, направленные к наблюдателю, выделяются из всех остальных при помощи алгоритма невидимых граней. Выделенные поверхности сохраняются в массив FACE-TABLE. Грани, направленные от наблюдателя, учитывать не требуется, поскольку они сами по себе скрыты, а потому не скрывают ребра других граней. Для каждой грани сохраняется максимальное и минимальное значение Zv. Криволинейные поверхности разделяются по силуэтным линиям (как в алгоритме невидимых граней), а видимые части этих поверхностей также сохраняются в массиве FACE-TABLE. Вместе с плоскими гранями.
2. Ребра граней из массива FACE-TABLE выделяются из всех прочих ребер и собираются в отдельный список. Ребра других граней, не входящих в FACE-TABLE, можно не рассматривать, поскольку они невидимы. Затем для каждого ребра из списка производится проверка, не закрывается ли это ребро гранью из FACE-TABLE.
3. Скрытие ребра гранью можно обнаружить, сравнивая диапазоны значений Zv ребра и грани. Возможны три случая (рис. 45). В случае рис. 45 (а), все значения Zv ребра меньше минимального значения Zv грани, то есть, грань находится перед ребром. В случае рис. 45 (б) значения Zv ребра больше максимального значения Zv грани, то есть грань находится за ребром. В случае рис. 45 (в) диапазоны значений Zv грани и ребра перекрываются, то есть часть ребра находится за гранью, а другая часть – перед ней. Если ребро находится перед проверяемой гранью, из массива FACE-TABLE выбирается следующая грань и ребро сравнивается уже с ней. Если ребро оказывается за гранью, или проходит ее насквозь, приходится выполнять дополнительное действие.

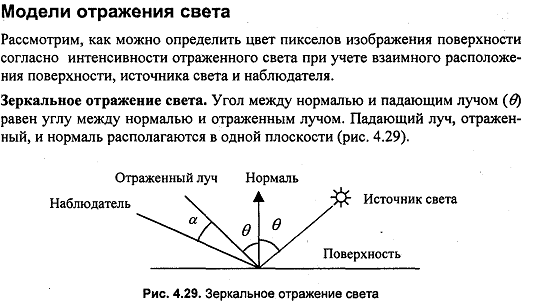
Ребро и грань проецируются на экран, после чего производится проверка перекрытия проекций. Если перекрытия нет, из этого следует, что ребро не закрывает проверяемую грань. Из массива FACE-TABLE выбирается следующая грань и проверяется согласно пункту 3. Если проекции перекрываются, ребро делится на две части по той точке, где она проходит сквозь проверяемую грань (рис. 46). Закрытая часть ребра отбрасывается, а видимые части добавляются в список. Затем пункт 3 повторяется для новых элементов списка. Исходное ребро удаляется из списка.

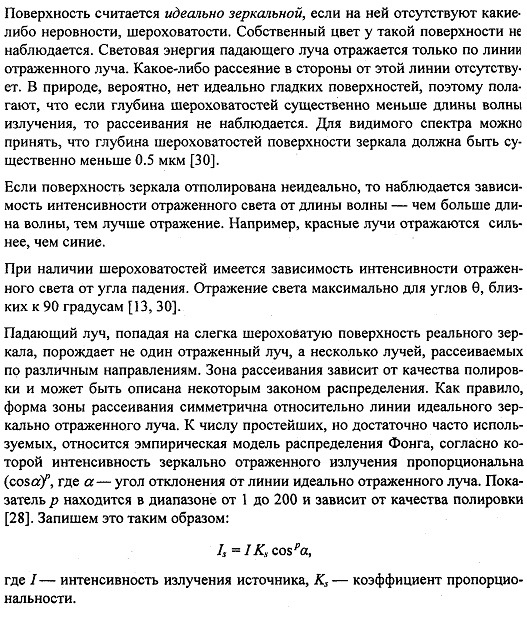
Ребра, прошедшие проверку со всеми гранями из FACE-TABLE, считаются видимыми и выводятся на экран.



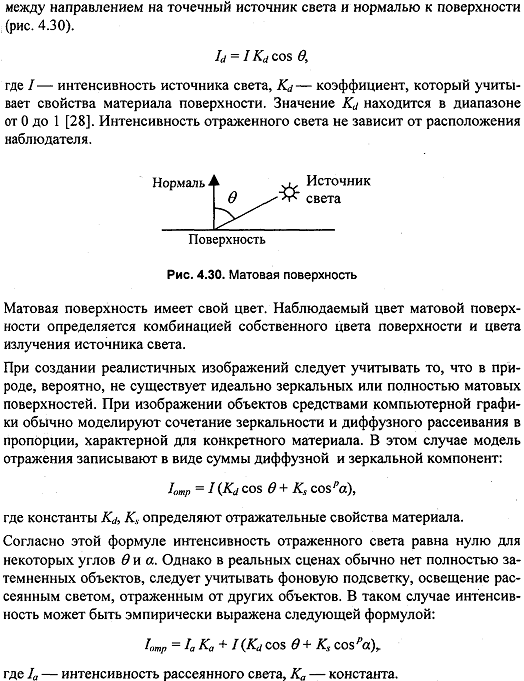


## 26. Закрашивание поверхностей. Модели отражения света

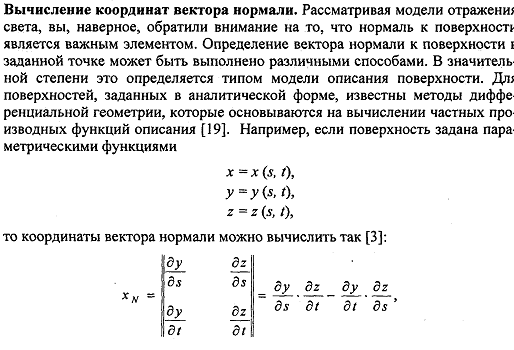
****

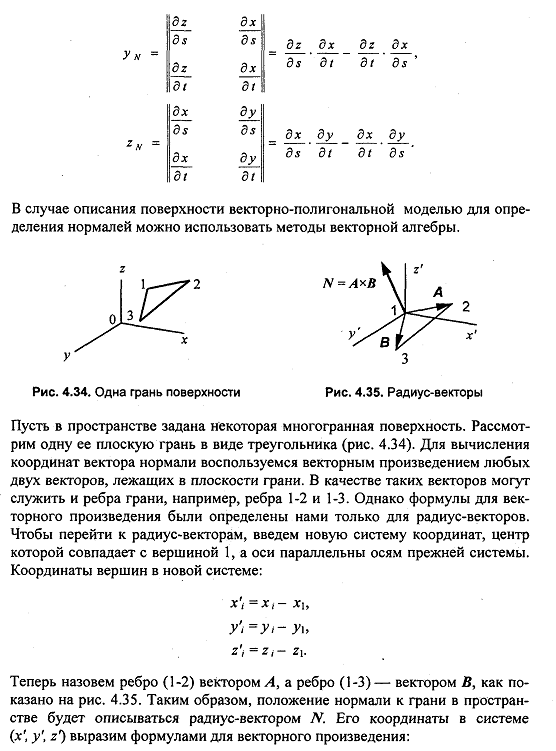
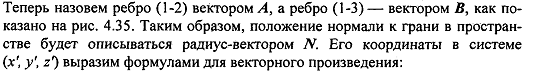
****

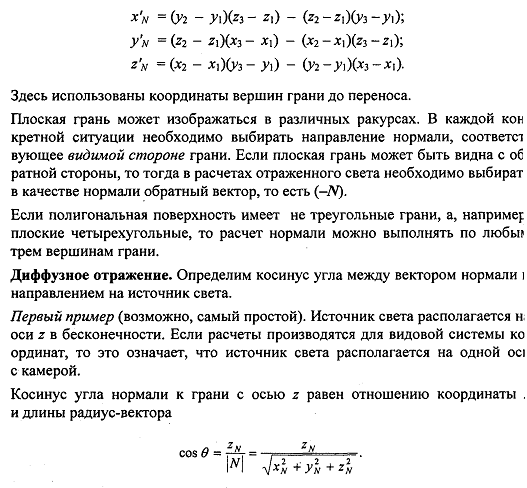
****

****

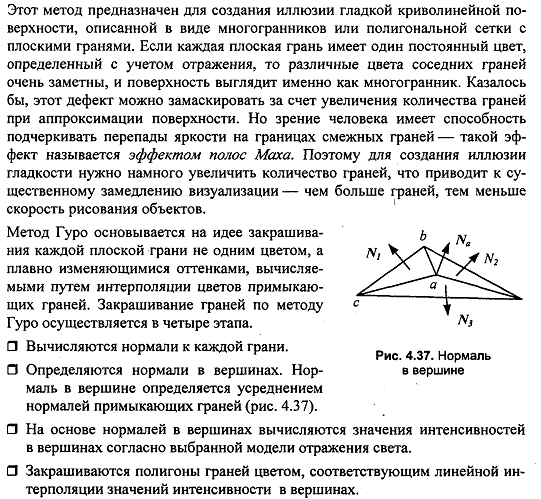
## 27. Закрашивание поверхностей. Вычисление нормалей и углов отражения. Реализация

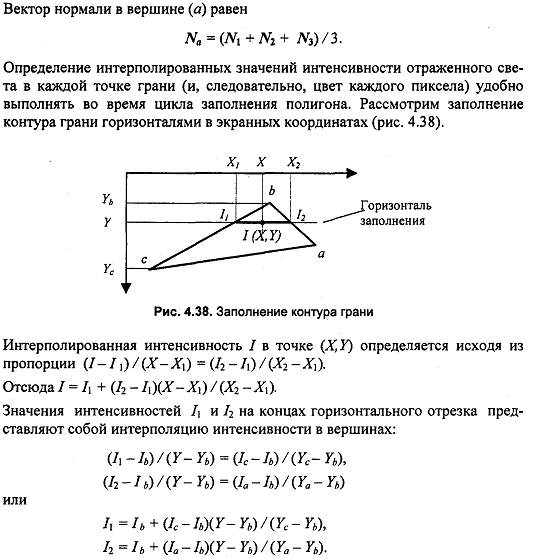
****

****  




## 28. Метод закрашивания Гуро

****

****

## 29. Графическая библиотека OpenGL. Особенности, основные возможности, архитектура, синтаксис команд

OpenGL является одним из самых популярных прикладных программных интерфейсов (API – Application Programming Interface) для разработки приложений в области двумерной и трехмерной графики. Стандарт OpenGL (Open Graphics Library – открытая графическая библиотека) был разработан и утвержден в 1992 году ведущими фирмами в области разработки программного обеспечения как эффективный аппаратно-независимый интерфейс, пригодный для реализации на различных платформах. Основой стандарта стала библиотека IRIS GL, разработанная фирмой Silicon Graphics Inc. Библиотека насчитывает около 120 различных команд, которые программист использует для задания объектов и операций, необходимых для написания интерактивных графических приложений. На сегодняшний день графическая система OpenGL поддерживается большинством производителей аппаратных и программных платформ. Эта система доступна тем, кто работает в среде Windows, пользователям компьютеров Apple. Характерными особенностями OpenGL, которые обеспечили распространение и развитие этого графического стандарта, являются:

* Стабильность. Дополнения и изменения в стандарте реализуются таким образом, чтобы сохранить совместимость с разработанным ранее программным обеспечением.
* Надежность и переносимость. Приложения, использующие OpenGL, гарантируют одинаковый визуальный результат вне зависимости от типа используемой операционной системы и организации отображения информации. Кроме того, эти приложения могут выполняться как на персональных компьютерах, так и на рабочих станциях и суперкомпьютерах.
* Легкость применения. Стандарт OpenGL имеет продуманную структуру и интуитивно понятный интерфейс, что позволяет с меньшими затратами создавать эффективные приложения, содержащие меньше строк кода, чем с использованием других графических библиотек. Необходимые функции для обеспечения совместимости с различным оборудованием реализованы на уровне библиотеки и значительно упрощают разработку приложений.

Основные возможности

Описать возможности OpenGL можно через функции его библиотеки. Все функции можно разделить на пять категорий:

1. Функции описания примитивов определяют объекты нижнего уровня иерархии (примитивы), которые способна отображать графическая подсистема. В OpenGL в качестве примитивов выступают точки, линии, многоугольники и т.д.
2. Функции описания источников света служат для описания положения и параметров источников света, расположенных в трехмерной сцене.
3. Функции задания атрибутов. С помощью задания атрибутов программист определяет, как будут выглядеть на экране отображаемые объекты. Другими словами, если с помощью примитивов определяется, что появится на экране, то атрибуты определяют способ вывода на экран. В качестве атрибутов OpenGL позволяет задавать цвет, характеристики материала, текстуры, параметры освещения.
4. Функции визуализации позволяет задать положение наблюдателя в виртуальном пространстве, параметры объектива камеры. Зная эти параметры, система сможет не только правильно построить изображение, но и отсечь объекты, оказавшиеся вне поля зрения.
5. Функции геометрических преобразований позволяют программисту выполнять различные преобразования объектов – поворот, перенос, масштабирование. При этом OpenGL может выполнять дополнительные операции, такие как использование сплайнов для построения линий и поверхностей, удаление невидимых фрагментов изображениями на уровне пикселей и т.д.

Интерфейс OpenGL

OpenGL состоит из набора библиотек. Все базовые функции хранятся в основной библиотеке, для обозначения которой используется аббревиатура GL. Помимо основной, OpenGL включает в себя несколько дополнительных библиотек.

Первая из них – библиотека утилит GL (GLU – GL Utility). Все функции этой библиотеки определены через базовые функции GL. В состав GLU вошла реализация более сложных функций, таких как набор популярных геометрических примитивов (куб, шар, цилиндр, диск), функции построения сплайнов, реализация дополнительных операций над матрицами и т.п.

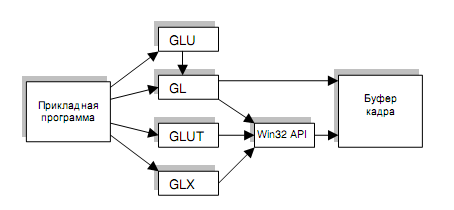


Рис. 1 Организация библиотеки OpenGL

OpenGL не включает в себя никаких специальных команд для работы с окнами или ввода информации от пользователя. Поэтому были созданы специальные переносимые библиотеки для обеспечения часто используемых функций взаимодействия с пользователем и для отображения информации с помощью оконной подсистемы. Наиболее популярной является библиотека GLUT (GL Utility Toolkit). Формально GLUT не входит в OpenGL, но фактически включается почти во все его дистрибутивы и имеет реализации для различных платформ. GLUT предоставляет только минимально необходимый набор функций для создания OpenGL-приложения. Существует также функционально аналогичная GLUT библиотека GLX. Однако она менее популярна.

Кроме того, функции, специфичные для конкретной оконной подсистемы, обычно входят в ее прикладной программный интерфейс. Так, функции, поддерживающие выполнение OpenGL, есть в составе Win32 API и Window. На рис. 1 схематически представлена организация системы библиотек в версии, работающей под управлением ОС Windows. Аналогичная организация используется и в других версиях OpenGL.

Архитектура OpenGL

Функции OpenGL реализованы в модели клиент-сервер. Приложение выступает в роли клиента – оно вырабатывает команды, а сервер OpenGL интерпретирует и выполняет их. Сам сервер может находиться как на том же компьютере, на котором находится клиент (например, в виде динамически загружаемой библиотеки – DLL), так и на другом (при этом может быть использован специальный протокол передачи данных между машинами).

GL обрабатывает и рисует в буфере кадра графические примитивы с учетом некоторого числа выбранных режимов. Каждый примитив – это точка, отрезок, многоугольник и т.д. Каждый режим может быть изменен независимо от других. Определение примитивов, выбор режимов и другие операции описываются с помощью команд в форме вызовов функций прикладной библиотеки.

Примитивы определяются набором из одной или более вершин (vertex). Вершина определяет точку, конец отрезка или угол многоугольника. С каждой вершиной ассоциируются некоторые данные (координаты, цвет, нормаль, текстурные координаты и т.д.), называемые атрибутами. В подавляющем большинстве случаев каждая вершина обрабатывается независимо от других.

С точки зрения архитектуры графическая система OpenGL является конвейером (рис. 2), состоящим из нескольких последовательных этапов обработки графических данных.

Команды OpenGL всегда обрабатываются в том порядке, в котором они поступают, хотя могут происходить задержки перед тем, как проявится эффект от их выполнения. В большинстве случаев OpenGL предоставляет непосредственный интерфейс, т.е. определение объекта вызывает его визуализацию в буфере кадра.

С точки зрения разработчиков, OpenGL – это набор команд, которые управляют использованием графической аппаратуры. Если аппаратура состоит только из адресуемого буфера кадра, тогда OpenGL должен быть реализован полностью с использованием ресурсов центрального процессора. Обычно графическая аппаратура предоставляет различные уровни ускорения: от аппаратной реализации вывода линий и многоугольников до изощренных графических процессоров с поддержкой различных операций над геометрическими данными.

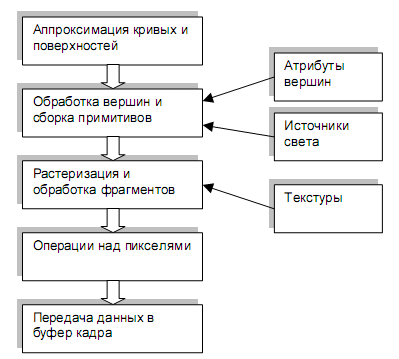


Рис. 2. Функционирование конвейера OpenGL

OpenGL является прослойкой между аппаратурой и пользовательским уровнем, что позволяет предоставлять единый интерфейс на разных платформах, используя возможности аппаратной поддержки.

Кроме того, OpenGL можно рассматривать как конечный автомат, состояние которого определяется множеством значений специальных переменных и значениями текущей нормали, цвета, координат текстуры и других атрибутов и признаков. Вся эта информация будет использована при поступлении в графическую систему координат вершины для построения фигуры, в которую она входит. Смена состояний происходит с помощью команд, которые оформляются как вызовы функций.

Синтаксис команд

Определения команд GL находятся в файле gl.h, для включения которого нужно написать

#include <gl/gl.h>

Для работы с библиотекой GLU нужно аналогично включить файл glu.h. Версии этих библиотек, как правило, включаются в дистрибутивы систем программирования, например Microsoft Visual Studio , Borland Delphi.

В отличие от стандартных библиотек, пакет GLUT нужно инсталлировать и подключать отдельно.

Все команды (процедуры и функции) библиотеки GL начинаются с префикса gl, все константы – с префикса GL\_. Соответствующие команды и константы библиотек GLU и GLUT аналогично имеют префиксы glu (GLU\_) и glut (GLUT\_)

Кроме того, в имена команд входят суффиксы, несущие информацию о числе и типе передаваемых параметров. В OpenGL полное имя команды имеет вид:

type LibNameCommandName[1 2 3 4][b s i f d ub us ui][v]

(type1 arg1,…,typeN argN)

type – тип возвращаемого функцией значения;

LibNameCommandName – имя команды, которое состоит из двух частей: LibNameиCommandName;

LibName – имя библиотеки, в которой описана эта функция; для базовых функций OpenGL, функций из библиотек GL, GLU, GLUT, GLAUX это gl, glu, glut, aux соответственно.

CommandName **–** имя команды (процедуры или функции).

[1 2 3 4] **–** число аргументов команды.

[b s i f d ub us ui] **–** тип аргумента.

Соответствие типов OpenGL и языка C(C++) приведено в таблице

|  |  |  |  |
| --- | --- | --- | --- |
| **Символ** | **Описание** | **Тип OpenGL** | **Тип C( C++)** |
| b | 8 – битовое целое | GLbyte | signed char |
| s | 16 – битовое целое | GLshort | short |
| i | 32 – битовое целое | GLint  Gsizei | long |
| f | 32 – битовое число с плавающей точкой | GLfloat  GLclampf | float |
| d | 64 – битовое число с плавающей точкой | GLdouble  GLclampf | double |
| ub | 8 – битовое беззнаковое целое | GLubyte  GLboolean | unsigned char |
| us | 16 – битовое беззнаковое целое | GLushort | unsigned short |
| ui | 32 – битовое беззнаковое целое | GLuint  GLenum  GLbitfield | unsigned long |
|  |  | GLvoid | void |

h [v] – наличие этого символа показывает, что в качестве параметров функции используется указатель на массив значений. Символы в квадратных скобках в некоторых названиях не используются. Например, команда glVertex2i(), описанная в библиотеке GL, использует в качестве параметров два целых числа, а команда glColor3fv() использует в качестве параметра указатель на массив из трех вещественных чисел. Использования нескольких вариантов каждой команды можно частично избежать, применяя перегрузку функций языка C++. Но интерфейс OpenGL не рассчитан на конкретный язык программирования, и, следовательно, должен быть максимально универсален.

## 30. Графическая библиотека OpenGL. Структура GLUT-приложения. Пример

Далее будем рассматривать построение консольного приложения при помощи библиотеки GLUT. Эта библиотека обеспечивает единый интерфейс для работы с окнами вне зависимости от платформы, поэтому описываемая ниже структура приложения остается неизменной для операционных систем Windows, Linux и других.

Функции GLUT могут быть классифицированы на несколько групп по своему назначению:

* Инициализация.
* Начало обработки событий.
* Управление окнами.
* Управление меню.
* Регистрация функций с обратным вызовом.
* Управление индексированной палитрой цветов.
* Отображение шрифтов.
* Отображение дополнительных геометрических фигур (тор, конус и др.).

Инициализация проводится с помощью функции:

void glutInit (int \*argcp, char \*\*argv),

где переменная argcp – указатель на стандартную переменную argc описываемую в функции main(), а argv – указатель на параметры, передаваемые программе при запуске, который описывается там же. Эта функция проводит необходимые начальные действия для построения окна приложения, и только несколько функций GLUT могут быть вызваны до нее. К ним относятся:

void glutInitWindowPosition (int x, int y)

– служит для установки положения окна OpenGL на экране, здесь x есть X -координата левого верхнего угла окна, y есть Y – координата левого верхнего угла окна. Координаты x и y определяются в пикселах.

void glutInitWindowSize (int width, int height),

– служит для установки размеров окна на экране, здесь width – ширина окна, height – высота окна. Значения width и height определяются в пикселах.

Если вызов этих функций опустить, то система присвоит параметрам окна значения по умолчанию.

void glutInitDisplayMode (unsigned int mode)

– задает режим отображения, где mode – определяет различные режимы отображения информации, которые могут совместно задаваться с использованием операции побитового «ИЛИ» ( «|» ) :

|  |  |
| --- | --- |
| **Значение** | **Комментарий** |
| GLUT\_RGBA | Выбор режима RGBA |
| GLUT\_RGB | То же, что и GLUT\_RGBA |
| GLUT\_INDEX | Палитровый видеорежим |
| GLUT\_SINGLE | Использование одинарного видеобуфера |
| GLUT\_DOUBLE | Использование двойной буферизации, применяется для создания анимации |
| GLUT\_ACCUM | Создавать аккумулирующий буфер |
| GLUT\_ALPHA | Создавать окно с α - каналом |
| GLUT\_DEPTH | Создавать окно с буфером глубины |
| GLUT\_SPENCIL | Создавать окно с буфером трафарета |

После инициализации можно создать окно, в которое будет происхо­дить вывод при помощи функции

int glutCreateWindow ( char \* name ),

где параметр name задает заголовок окна. Функция возвращает целочис­ленный идентификатор окна.

Функции библиотеки GLUT реализуют так называемый событийно-управляемый механизм. Это означает, что есть некоторый внутренний цикл, который запускается после соответствующей инициализации и обрабатывает одно за другим все события, объявленные во время инициализации. К событиям относятся: щелчок мыши, закрытие окна, изменение свойств окна, передвижение курсора, нажатие клавиши и "пустое" (idle) событие, когда ничего не происходит. Для проведения периодической проверки совершения того или иного события надо зарегистрировать функцию, которая будет его обрабатывать. Для этого используются следующие функции:

void glutDisplayFunc (void (\*func) (void))

– служит для установки обработчика события, связанного с необходимостью перерисовки содержимого окна. Параметр func задает функцию рисования для окна приложения, которая вызывается при необходимости создания или восстановления изображения.

void glutPostRedisplay (void)

– служит для явного указания, что окно надо обновить.

void glutReshapeFunc (void (\*func) (int width, int height))

– служит для установки обработчика события, связанного с изменением размера окна. Параметр func задает функцию которая будет вызываться при изменении размеров окна перед вызовом функции отрисовки содержимого окна. Параметры width и height определяют новые значения ширины и высоты окна в пикселах.

void glut KeyboardFunc(void (\*func)(unsigned char key, int x, int y))

– служит для установки обработчика сообщений от клавиатуры. Параметр func задает функцию которая будет вызываться в ответ на сообщения клавиатуры, параметр key содержит ASCII – код символа, а параметры x и y – координаты курсора мыши в пикселах по отношению к верхнему левому углу окна.

Обработчик клавиш, не генерирующих ASCII – кода (такие как <Shift>), обрабатываются с помощью функции

void glutSpecialFunc(void(\*func)(int key, int x, int y)),

где параметр key несет в себе информацию о нажатой клавише и может принимать одно из значений, указанных в таблице.

|  |  |
| --- | --- |
| **Значение *key*** | **Клавиша** |
| GLUT\_KEY\_F1 | Клавиша <F1> |
| GLUT\_KEY\_F2 | Клавиша <F2> |
| GLUT\_KEY\_F3 | Клавиша <F3> |
| GLUT\_KEY\_F4 | Клавиша <F4> |
| GLUT\_KEY\_F5 | Клавиша <F5> |
| GLUT\_KEY\_F6 | Клавиша <F6> |
| GLUT\_KEY\_F7 | Клавиша <F7> |
| GLUT\_KEY\_F8 | Клавиша <F8> |
| GLUT\_KEY\_F9 | Клавиша <F9> |
| GLUT\_KEY\_F10 | Клавиша <F10> |
| GLUT\_KEY\_F11 | Клавиша <F11> |
| GLUT\_KEY\_F12 | Клавиша <F12> |
| GLUT\_KEY\_LEFT | Клавиша <←> |
| GLUT\_KEY\_RIGHT | Клавиша <→> |
| GLUT\_KEY\_UP | Клавиша <↑> |
| GLUT\_KEY\_DOWN | Клавиша <↓> |
| GLUT\_KEY\_PAGE\_UP | Клавиша <Page UP> |
| GLUT\_KEY\_PAGE\_DOWN | Клавиша <Page Down> |
| GLUT\_KEY\_HOME | Клавиша <Home> |
| GLUT\_KEY\_END | Клавиша <End> |
| GLUT\_KEY\_INSERT | Клавиша <Insert> |

void glutMouseFunc (void (\*func) (int button, int state, int x, int y))

– служит для установки обработчика сообщений от мыши. Параметр func задает функцию которая будет вызываться в ответ на сообщения от мыши. Параметр button принимает одно из значений GLUT\_LEFT\_BUTTON, GLUT\_MIDDLE\_BUTTON, GLUT\_RIGHT\_BUTTON и несет в себе информацию о том, какая клавиша мыши была нажата (или отпущена). Параметр state принимает одно из значений GLUT\_DOWN (клавиша нажата) или GLUT\_UP (клавиша отпущена). Параметры x и y содержат координаты курсора мыши.

Вызов обработчика, установленного glutMouseFunc, происходит лишь при нажатии или отпускании клавиши мыши. Если необходимо установить обработчик на перемещение мыши, то для этого следует воспользоваться функциями

void glutMotionFunc (void (\*func)(int x, int y)),

void glutPassiveMotionFunc (void (\*func)(int x, int y))

Первая из них устанавливает обработчик, который будет вызываться при перемещении мыши при нажатой клавише (любой из имеющихся) мыши, а вторая – при передвижении мыши в том случае, когда ни одна из клавиш мыши не нажата.

int glutCreateMenu(void (\*func)(int ItemNamber)

– служит для создания меню. В качестве аргумента используется имя функции, которая будет вызываться при выборе пунктов меню. Параметр ItemNamber этой функции определяет пункт меню, который будет выбран при ее вызове.

glutAddMenuEntry(char\* ItemName, int Namber)

– служит для добавления пунктов в уже созданное меню. Параметр ItemName определяет название пункта меню, а параметр Namber – идентификатор пункта, который через параметр ItemNamber передается функции, вызываемой при выборе этого пункта меню.

glutAttachMenu(int button)

– задает кнопку мыши, которая будет использоваться для активизации меню. Параметр button принимает одно из значений GLUT\_LEFT\_BUTTON, GLUT\_MIDDLE\_BUTTON, GLUT\_RIGHT\_BUTTON.

Ниже приводится последовательность команд для создания меню, добавления в него пунктов и активизация с помощью правой клавиши мыши.

glutCreateMenu(NaneFunction);

glutAddMenuEntry("Первый пункт меню",1);

glutAddMenuEntry("Второй пункт меню",2);

glutAttachMenu(GLUT\_RIGHT\_BUTTON);

В приведенном примере параметр NaneFunction – имя функции котрая будет вызвана при выборе пункта меню.

void glutIdleFunc (void (\*func) (void))

– служит для установки обработчика, который будет постоянно вызываться в «фоновом» режиме. Наиболее часто этот обработчик используется для проведения некоторых вспомогательных действий, которые можно осуществить в моменты времени, когда приложение «ничего не делает».

void glutMainLoop (void)

– служит для запуска бесконечного цикла обработки сообщений. Функция вызывается в конце любой программы, использующей GLUT.

void glutSwapBuffers()

– при использовании двойной буферизации служит для смены буферов местами (буфера, содержимое которого видно сейчас на экране, и буфера в который производится вывод)

int glutGet(GLEnum state)

– служит для получения информации о внутренних переменных glut, параметр state может принимать значения, указанные в таблице.

|  |  |
| --- | --- |
| **Значение *state*** | **Комментарий** |
| GLUT\_WINDOW\_X | – для возврата x- координаты окна |
| GLUT\_WINDOW\_Y | – для возврата y- координаты окна |
| GLUT\_WINDOW\_WIDTH | – для возврата ширины окна |
| GLUT\_WINDOW\_HEIGHT | – для возврата высоты окна |
| GLUT\_ELAPSED\_TIME | – для возврата времени в миллисекундах с момента инициализации GLUT |

Библиотека OpcnGL реализована по модели клиент-сервер, приложение выступает в роли клиента – вырабатывает команды, a OpenGL (сервер) ин­терпретирует и обрабатывает их. При этом сервер может располагаться как на том же компьютере, что и клиент, так и на другом Сервер может под­держивать несколько контекстов OpenGL, каждый из которых инкапсулиру­ет состояние OpenGL. Клиент может подключиться к любому из них. По этой причине желательно гарантировать, что выполнение всех вызванных команд завершится либо за определенное, либо за конечное время. Для обеспечения этого в OpenGL реализованы две команды.

Первая из них

void glFinish()

блокирует дальнейшее выполнение программы, пока не будут завершены все вызванные перед ней команды OpenGL, включая изменения состояния и содержимого буфера кадра.

Вторая команда

void gIFlush()

служит для освобождения буферов OpenGL.

Различные реализации команд OpenGL для работы с буферами размешают­ся в различных местах, включая сетевые буферы и графические акселерато­ры. Команда gIFIush освобождает все эти буферы, заставляя все вызванные команды выполниться так быстро, как это требуется фактической реализа­цией воспроизведения. И хотя выполнение не может быть закончено за не­который заданный отрезок времени, оно завершается за конечный период.

Поскольку любая программа OpenGL может выполняться в сети или на графическом акселераторе, который накапливает команды, необходимо убедится, что функция gIFIush вызвана во всех программах, которые требуют, чтобы все их предварительно вызванные команды были закончены. Например, необходимо вызвать эту команду перед ожиданием пользовательского ввода, который зависит от сформированного образа.

Команда gIFIush, в отличие от glFinish, не ожидает завершения выполнения всех предварительно вызванных команд.

## 31. Графическая библиотека OpenGL. Рисование графических примитивов. Дисплейные списки. Удаление нелицевых граней. Вывод текста. Примеры

Как правило, задачей программы, использующей OpenGL, является обработка трехмерной сцены и интерактивное отображение в буфере кадра. Сцена состоит из набора трехмерных объектов, источников света и виртуальной камеры, определяющей текущее положение наблюдателя. Обычно приложение OpenGL в бесконечном цикле вызывает функцию обновления изображения в окне. В этой функции и сосредоточены вызовы основных команд OpenGL. Если используется библиотека GLUT, то это будет функция с обратным вызовом, зарегистрированная с помощью вызова **glutDisplayFunc()**. GLUT вызывает эту функцию, когда операционная система информирует приложение о том, что содержимое окна необходимо перерисовать (например, если окно было перекрыто другим). Создаваемое изображение может быть как статичным, так и анимированным, т.е. зависеть от каких-либо параметров, изменяющихся со временем. В этом случае лучше вызывать функцию обновления самостоятельно. Например, с помощью команды **glutPostRedisplay()**. Типичная функция обновления изображения, как правило, состоит из трех шагов:

1. очистка буферов OpenGL;
2. установка положения наблюдателя;
3. преобразование и рисование геометрических объектов.

OpenGL содержит внутри себя несколько различных буферов. Среди них ***фрейм-буфер*** (куда производится построение изображения), ***z-буфер***, служащий для удаления невидимых поверхностей, ***буфер трафарета*** и ***аккумулирующий буфер*** (рис. 3)



Рис. 3

Для очистки окна (экрана, внутренних буферов) служит команда

**void glClear(GLbitfield buf)**,

которая очищает буферы, заданные переменной **buf**. Параметр **buf**  определяет комбинацию констант, соответствующую буферам, которые нужно очистить

|  |  |
| --- | --- |
| **Значение переменной buf** | **Комментарий** |
| GL\_COLOR\_BUFFER\_BIT | Очистить буфер изображения (фрейм - буфер) |
| GL\_DEPTH\_BUFFER\_BIT | Очистить z - буфер |
| GL\_ACCUM\_BUFFER\_BIT | Очистить аккумулирующий буфер |
| GL\_STENSIL\_BUFFER\_BIT | Очистить буфер трафарета |

Типичная программа вызывает команду

**glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)**

для очистки буферов цвета и глубины.

При этом цвет, которым очищается буфер изображения, задается командой

**void glClearColor ( GLclampf red, GLclampf green, GLclampf blue,**

**GLclampf alpha )**

Команда glClearColor устанавливает цвет, которым будет заполнен буфер кадра. Первые три параметра команды задают R,G и B компоненты цвета и должны принадлежать отрезку [0,1]. Четвертый параметр задает так называемую альфа - компоненту. Как правило, он равен 1. По умолчанию цвет – черный (0,0,0,1).

Установка положения наблюдателя и преобразования трехмерных объектов (поворот, сдвиг и т.д.) контролируются с помощью задания ***матриц преобразования***. ***Преобразования*** объектов и настройка ***положения виртуальной камеры*** рассмотрены ниже.

Рассмотрим вопрос о том, как передать в OpenGL описания объектов, находящихся в сцене. Каждый объект является набором примитивов OpenGL.

**Вершины и примитивы**

Вершина является атомарным графическим примитивом OpenGL и определяет точку, конец отрезка, угол многоугольника и т.д. Все остальные примитивы формируются с помощью задания вершин, входящих в данный примитив. Например, отрезок определяется двумя вершинами, являющимися концами отрезка. С каждой вершиной ассоциируются ее атрибуты. В число основных атрибутов входят ***положение*** вершины в пространстве, ***цвет*** вершины и ***вектор*** нормали.

***Положение вершины в пространстве***

Положение вершины определяются заданием ее координат в двух-, трех-, или четырехмерном пространстве (однородные координаты). Это реализуется с помощью нескольких вариантов команды **glVertex\***:

**void glVertex[2 3 4][s i f d] (type x, …)**

**void glVertex[2 3 4][s i f d]v (type \*v)**

Каждая команда задает четыре координаты вершины: **x**, **y**, **z**, **w**. Команда **glVertex2\*** получает значения **x** и **y**. Координата **z** в таком случае устанавливается по умолчанию равной 0, координата w – равной 1. **Vertex3\*** получает координаты **x**, **y**, **z** и заносит в координату w значение 1. **Vertex4\*** позволяет задать все четыре координаты. Например

glVertex2s(1,2)

glvertex3f(2.3, 1.5, 0,2)

GLdouble vect[]={1.0, 2.0, 3.0, 4.0}

glVertex4dv (vect)

Для ассоциации с вершинами цветов, нормалей и текстурных координат используются текущие значения соответствующих данных, что отвечает организации OpenGL как конечного автомата. Эти значения могут быть изменены в любой момент с помощью вызова соответствующих команд.

***Цвет вершины***

Для задания текущего цвета вершины используются команды :

**void glColor[3 4][b s i f d ub us ui] (type r, type g, type b, type a)**

**void glColor[3 4][b s i f d ub us ui]v (const type \*v**)

Первые три параметра задают R, G, B компоненты цвета, а последний параметр определяет коэффициент непрозрачности (так называемая альфа-компонента). Если в названии команды указан тип **«f»** (float), то значения всех параметров должны принадлежать отрезку [0,1], при этом по умолчанию значение альфа-компоненты устанавливается равным 1.0, что соответствует полной непрозрачности. Тип «**ub**» (unsigned byte) подразумевает, что значения должны лежать в отрезке [0,255].

Вершинам можно назначать различные цвета, и, если включен соответствующий режим, то будет проводиться линейная интерполяция цветов по поверхности примитива.

Для управления режимом интерполяции используется команда

**void glShadeModel (GLenum mode)**

вызов которой с параметром **GL\_SMOOTH** включает интерполяцию (установка по умолчанию), а с **GL\_FLAT** – отключает.

***Нормаль***

Вектор нормали ***применяется для расчетов освещения и затенения***. Нормаль в OpenGL является ***атрибутом вершины***. В общем случае, когда речь идет о грани трехмерного объекта, вершины одной грани имеют разные нормали, так как каждая вершина принадлежит нескольким граням, лежащим в разных плоскостях. Тогда нормаль к вершине можно определить как сумму нормалей к граням, содержащим рассматриваемую вершину,  или как среднюю нормаль , где  – вектор нормали к -й грани, – число граней, образующих вершину. Для стандартных объектов OpenGL рассчитывает нормали автоматически. При отображении объектов, создаваемых пользователем, нормали должны быть ***рассчитаны самостоятельно***.

После определения координат нормали она должна быть задана командами

**void glNormal3[b s i f d] (GLtype nx,GL type ny, GLtype nz )**,

**void glNormal3[b s i f d]v (GLtype \*v )**,

где **nx**, **ny**, **nz** – координаты новой текущей нормали, **v** – массив координат вектора нормали.

Нормали задаются к каждой выводимой вершине.

Для правильного расчета освещения необходимо, чтобы вектор нормали имел ***единичную длину***. Командой **glEnable(GL\_NORMALIZE)** можно включить специальный режим, при котором задаваемые нормали будут нормироваться автоматически. По умолчанию нормализация не выполняется.

Режим автоматической нормализации должен быть ***включен***, если приложение использует модельные ***преобразования растяжения/сжатия***, так как в этом случае длина нормалей изменяется при умножении на модельно-видовую матрицу.

Однако применение этого режима уменьшает скорость работы механизма визуализации OpenGL, так как нормализация векторов имеет заметную вычислительную сложность (взятие квадратного корня и т.п.). Поэтому лучше сразу задавать ***единичные нормали***.

Отметим, что команды

**void glEnable (GLenum mode)**,

**void glDisable (GLenum mode)**,

где параметр **mode** определяет вид режима, производят включение и отключение того или иного режима работы конвейера OpenGL. Эти команды применяются достаточно часто, и их возможные параметры будут рассматриваться в каждом конкретном случае.

**Операторные скобки glBegin / glEnd**

Выше было рассмотрено задание атрибутов одной вершины. Однако чтобы задать атрибуты графического примитива, одних координат вершин недостаточно. Эти вершины надо объединить в одно целое, определив необходимые свойства. Для этого в OpenGL используются так называемые операторные скобки, являющиеся вызовами специальных команд OpenGL Определение примитива или последовательности примитивов происходит между вызовами команд

**void glBegin (GLenum mode)**

**void glEnd ()**

Параметр **mode** определяет тип примитива, который задается внутри и может принимать следующие значения (рис. 4):

|  |  |
| --- | --- |
| **Значение mode** | **Комментарий** |
| **GL\_POINTS** | Каждая вершина задает координаты некоторой точки. |
| **GL\_LINES** | Каждая отдельная пара вершин определяет отрезок; если задано нечетное число вершин, то последняя вершина игнорируется **v0v1, v2v3…** |
| **GL\_LINE\_STRIP** | Каждая следующая вершина задает отрезок вместе с предыдущей – незамкнутая ломаная **v0v1v2v3…vn** |
| **GL\_LINE\_LOOP** | Замкнутая ломаная **v0v1v2v3…vnv0** |
| **GL\_TRIANGLES** | Каждые отдельные три вершины определяют треугольник. Если задано не кратное трем число вершин, то последние вершины игнорируются **v0 v1v2,**  **v3v4v5...** |
| **GL\_TRIANGLE\_STRIP** | Каждая следующая вершина задает треугольник вместе с двумя предыдущими – связанная полоса треугольников **v0v1v2, v2v1v3, v2v3v4…** |
| **GL\_TRIANGLE\_FAN** | Треугольники задаются первой вершиной и каждой следующей парой вершин (пары не пересекаются) – веер треугольников **v0v1v2, v0v2v3, v0v3v4…** |
| **GL\_QUADS** | Каждая отдельная четверка вершин определяет четырехугольник; если задано не кратное четырем число вершин, то последние вершины игнорируются **v0v1v2v3, v4v5v6v7, …** |
| **GL\_QUAD\_STRIP** | Полоса четырехугольников **v0v1v3v2, v2v3v5v4 … .** |
| **GL\_POLYGON** | Последовательно задаются вершины выпуклого многоугольника. |

Например, чтобы нарисовать треугольник с разными цветами в вершинах, достаточно написать:

GLfloat BlueCol[3] = {0,0,1};

**glBegin(GL\_TRIANGLES)**;

glColor3f(1.0, 0.0, 0.0); // Красный

glVertex3f(0.0, 0.0, 0.0);

glColor3ub(0,255,0); // Зеленый

glVertex3f(1.0, 0.0, 0.0);

glColor3fv(BlueCol); // Синий

glVertex3f(1.0, 1.0, 0.0);

**glEnd();**

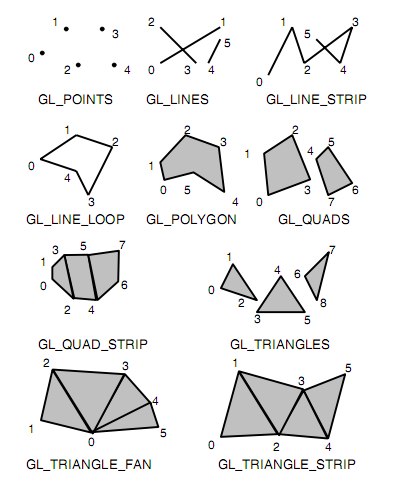


Рис. 4

Как правило, разные типы примитивов имеют различную скорость визуализации на разных платформах. Для увеличения производительности предпочтительнее использовать примитивы, требующие меньшее количество информации для передачи на сервер, такие как **GL\_TRIANGLE\_STRIP**, **GL\_QUAD\_STRIP**, **GL\_TRIAGLE\_FAN**. Кроме задания самих многоугольников, можно определить метод их отображения на экране. Для этого необходимо определить понятие ***лицевых***  и ***обратных граней***.

Под ***гранью*** (многоугольником) в OpenGL подразумевается замкнутый выпуклый многоугольник с несамопересекающейся границей.

***По умолчанию*** лицевой считается та сторона, вершины которой обходятся ***против***  часовой стрелки. Направление обхода вершин лицевых граней можно изменить вызовом команды

**void glFrontFace (GLenum mode)**

со значением параметра **mode** равным **GL\_CW (clockwise)**, а вернуть значение по умолчанию можно, указав **GL\_CCW (counter-clockwise)**. Чтобы изменить метод отображения многоугольника используется команда

**void glPolygonMode (GLenum face, Glenum mode)**

Параметр **mode** определяет, ***как*** будут отображаться многоугольники, а параметр **face** устанавливает ***тип*** многоугольников, к которым будет применяться эта команда и может принимать следующие значения:

**GL\_FRONT** – для лицевых граней,

**GL\_BACK** – для обратных граней,

**GL\_FRONT\_AND\_BACK** – для всех граней

Параметр **mode** может быть равен:

**GL\_POINT –** отображение только вершин многоугольников.

**GL\_LINE –** многоугольники будут представляться набором отрезков.

**GL\_FILL –** многоугольники будут закрашиваться текущим цветом с учетом освещения, и ***этот режим установлен по умолчанию***.

Также можно указывать, какой тип граней отображать на экране. Для этого сначала надо установить соответствующий режим вызовом команды

**glEnable (GL\_CULL\_FACE)**,

а затем выбрать тип отображаемых граней с помощью команды

**void glCullFace (GLenum mode)**

Вызов с параметром **GL\_FRONT** приводит к удалению из изображения всех лицевых граней, а с параметром **GL\_BACK** – обратных (установка по умолчанию).

Кроме рассмотренных стандартных примитивов в библиотеках GLU и GLUT описаны более сложные фигуры, такие как сфера, цилиндр, диск (в GLU) и сфера, куб, конус, тор, тетраэдр, додекаэдр, икосаэдр, октаэдр и чайник (в GLUT). Автоматическое наложение текстуры предусмотрено только для фигур из библиотеки GLU.

При изображении объектов в OpenGL можно изменять размер точек и толщину линий. Для этой цели служат следующие команды.

**void glPointSize(GLfloat size)** –

устанавливает ***диаметр (size) в пикселях*** точек, изображаемых с помощью примитива **GL\_POINTS**. По умолчанию значение параметра size равно 1.0. В любой реализации OpenGL гарантирована поддержка размера точки равного 1.0. В типичных реализациях Microsoft Windows поддерживается размер точки от 0.5 до 10 с шагом 0.125.

**void glLineWidth(GLfloat size)** –

устанавливает ***ширину (size) в пикселях*** линий, изображаемых с помощью примитивов **GL\_LINES**, **GL\_LINE\_STRIP** или **GL\_LINE\_LOOP**. По умолчанию значение параметра size равно 1.0. В любой реализации OpenGL гарантирована поддержка размера точки равного 1.0. В типичных реализациях Microsoft Windows поддерживается размер точки от 0.5 до 10 с шагом 0.125.

**Дисплейные списки**

В традиционных языках программирования существуют функции и процедуры – т. е. можно выделить определенный набор команд, запом­нить его в некотором одном определенном месте и вызывать каждый раз, когда возникает потребность в соответствующей последовательности ко­манд. Подобная возможность существует и в OpenGL – набор команд OpcnGL можно запомнить в так называемый дисплейный список (***display list***), при этом все команды и данные переводятся в некоторое внутреннее представление, наиболее удобное для данной реализации OpenGL, и затем вызвать при помощи всего одной команды.

Каждому дисплейному списку соответствует некоторое целое число, идентифицирующее этот список.

Для создания нового дисплейного списка необходимо поместить все команды, которые должны в него войти, между следующими операторными скобками:

**void glNewList (GLuint list, GLenum mode)**,

**void glEndList ()**,

где параметр **list** задает номер списка, а параметр **mode** определяет режим обработки команд, входящих в список и может принимать следующие значения:

**GL\_COMPILE** – команды записываются в список без выполнения, **GL\_COMPILE\_AND\_EXECUTE –** команды сначала выполняются, а затем записываются в список.

Следует иметь в виду, что если в качестве параметра **list** списка используется уже ***существующий*** ***номер***, то новый список ***не создается***, а существующий список ***заменяется на новый***.

Узнать, занят ли данный номер **list** каким-либо дисплейным списком, мож­но при помощи функции

**GLboolean gllsList ( GLuint list )**

Зарезервировать **range** свободных подряд идущих номеров для иденти­фикации дисплейных списков можно при помощи функции

**GLuint glGenLists ( GLsizei range )**

Эта функция возвращает первый свободный номер из блока зарезерви­рованных номеров данной командой***.***

После того, как список создан, его можно вызвать командой

**void glCallList (GLuint list)**

указав в параметре **list** идентификатор нужного списка. Для вызова сразу нескольких списков, можно воспользоваться командой

**void glCallLists (GLsizei n, GLenum type, const GLvoid \*lists)**

вызывающей **n** списков с идентификаторами из массива **lists**, тип элементов которого указывается в параметре **type**. Это могут быть типы **GL\_BYTE**, **GL\_UNSIGNED\_BYTE**, **GL\_SHORT**, **GL\_INT**, **GL\_UNSIGNED\_INT** и некоторые другие. Для удаления списков используется команда

**void glDeleteLists (GLint list, GLsizei range)**

которая удаляет списки с идентификаторами **ID** из диапазона

**list ≤ ID ≤ list+range-1**.

Пример:

**glNewList(1, GL\_COMPILE);**

glBegin(GL\_TRIANGLES);

glVertex3f(1.0f, 1.0f, 1.0f);

glVertex3f(10.0f, 1.0f, 1.0f);

glVertex3f(10.0f, 10.0f, 1.0f);

glEnd();

**glEndList()**

…

glCallList(1);

**Замечание**. Дисплейные списки ***нельзя изменять***. Список можно уничтожить или создать заново. В дисплейном списке команды запоми­наются вместе со своими аргументами на момент передачи, так что в сле­дующем примере последний оператор присваивания дисплейный список не изменяет**.**

GLfloat color [] ={0.0, 0.0, 0.0 };

glNewList ( 1, GL.COMPILE );

glColor3fv ( color );

glEndList ();

color [2] = 1.0; // Не изменяет дисплейный список.

Не все команды OpenGL могут быть записаны в дисплейный список. К ним относятяся такие команды как **glDeleteLists ()**, **glFinish()**, **glGenLists ()** и некоторые другие.

Дисплейные списки в оптимальном, скомпилированном виде хранятся в памяти сервера, что позволяет рисовать примитивы в такой форме максимально быстро. В то же время большие объемы данных занимают много памяти, что влечет, в свою очередь, падение производительности. Такие большие объемы (больше нескольких десятков тысяч примитивов) лучше рисовать с помощью массивов вершин.

**Массивы вершин**

Если вершин много, то чтобы не вызывать для каждой команду glVertex\*(), удобно объединять вершины в массивы, используя команду

**void glVertexPointer (GLint size, GLenum type, GLsizei stride, void\* ptr)**,

которая определяет способ хранения и координаты вершин. При этом **size**  определяет число координат вершины (может быть равен 2, 3, 4), **type** определяет тип данных (может быть равен **GL\_SHORT, GL\_INT, GL\_FLOAT, GL\_DOUBLE**). Иногда удобно хранить в одном массиве другие атрибуты вершины, тогда параметр **stride** задает смещение от координат одной вершины до координат следующей; если **stride =0**, это значит, что координаты расположены последовательно. В параметре **ptr** указывается адрес, где находятся данные.

Анологично можно определить ***массив*** ***нормалей***, ***цветов*** и некоторых других атрибутов вершины, используя команды

**void glNormalPointer ( GLenum type, GLsizei stride, void \*pointer )**,

**void glColorPointer ( GLint size, GLenum type, GLsizei stride, void \*pointer )**

Для того, чтобы эти массивы можно было использовать в дальнейшем, надо вызвать команду

**void glEnableClientState (GLenum array)**,

где параметр **array** может принимать значения **GL\_VERTEX\_ARRAY**, **GL\_NORMAL\_ARRAY** и **GL\_COLOR\_ARRAY** соответственно. После окончания работы с массивом желательно вызвать команду

**void glDisableClientState (GLenum array)**

с соответствующим значением параметра **array**.

Для отображения содержимого массивов используется команда

**void glArrayElement (GLint index)**,

которая передает OpenGL атрибуты вершины, используя элементы массива с номером **index**. Это аналогично последовательному применению команд вида **glColor\*(…)**, **glNormal\*(…)**, **glVertex\*(…)** c соответствующими параметрами. Однако вместо нее обычно вызывается команда

**void glDrawArrays (GLenum mode, GLint first, GLsizei count)**,

рисующая **count** примитивов, определяемых параметром **mode**, используя элементы из массивов с индексами от ***first*** до ***first+count-1***. Это эквивалентно вызову последовательности команд ***glArrayElement(***) с соответствующими индексами.

В случае, если одна вершина входит в несколько примитивов, то вместо дублирования ее координат в массиве удобно использовать ее индекс. Для этого надо вызвать команду

**void glDrawElements (GLenum mode, GLsizei count, GLenum type,**

**void \*indices)**

где ***indices*** – это массив номеров вершин, которые надо использовать для построения примитивов, **type** определяет тип элементов этого массива: **GL\_UNSIGNED\_BYTE**, **GL\_UNSIGNED\_SHORT**, **GL\_UNSIGNED\_INT**, а **count** задает их количество.

Важно отметить, что использование массивов вершин позволяет ***оптимизировать***  передачу данных на сервер OpenGL, и, как следствие, ***повысить скорость рисования трехмерной сцены***. Такой метод определения примитивов является одним из самых быстрых и хорошо подходит для визуализации больших объемов данных.

Помимо рассмотренных примитивов, определяемых внутри командных ско­бок, в OpenGL предусмотрены команды

**void gIRect[d f i s](GLtype x1, GLtype y1, GLtype x2, GLtype y2)**

и

**void gIRect[d f i s] v (GLtype\* v1, GLtype\* v2),**

которые позволяют достаточно эффективно задавать прямоугольник в плос­кости **z = 0**, определяя два его ***противоположных*** угла.

Параметры **xl** и **yl** задают координаты ***одной из вершин*** прямоугольника, а **х2** и **у2** – координаты его ***противоположной вершины***. Параметры **vl** и **v2** определяют указатели на соответствующие двухточечные массивы вершин.

Если прямоугольник изображается с помощью функции **glRect**, то стороны многоугольника прокладываются между вершинами в следующем порядке: (**x1, yl)**, **(х2, у1)**, **(х2, у2)**, **(xl, у2)**, а затем обратно к первой вершине.

Их действие анало­гично, например, следующей последовательности команд:

**glBegin(GL\_POLYGON) ;**

**glVertex2\*(xl, yl); // \* заменяется соответствующим символом,**

**glVertex2\*(х2, yl); // зависящим от типа используемых**

**glVertex2\*(х2, у2); //данных**

**glVertex2\*(xl,** *у2);*

**glEnd() ;**

Вывод текста

Существуют два типа шрифтов – ***штриховой (эскизный)*** и ***растровый***. Штриховой шрифт формируется по тому же принципу, что и прочие графические примитивы. Начертание символа определяется вершинами соответствующих прямолинейных и криволинейных отрезков. Если символ формируется из замкнутых контуров, то его внутреннюю область можно залить. Преимущество использования штриховых шрифтов заключается в том, что с ними можно обращаться в графической системе точно так же, как с любыми другими графическими объектами. Несомненным достоинством штриховых шрифтов является и сохранение начертания во всех деталях при выполнении геометрических преобразований масштабирования или повороте. Поэтому определение символов набора выполняется только для одного, базового, размера, а все прочие получаются после выполнения стандартных геометрических преобразований – масштабирования и поворота. Однако визуализируются штриховые шрифты медленнее, чем растровые.

Растровый шрифт определяется значительно проще, а отображается быстрее (рис. 4.1). Символы шрифта определяются на прямоугольной области и представляют собой блоки битов. Каждый блок задает определенный символ в виде образа из нулей и единиц, которые соответствуют засвеченным и незасвеченным точкам растра. При отображении определенный таким способом символ помешается в буфер кадра с помощью операции побитового переноса, которая выполняется очень быстро.

Увеличить размер символов растрового шрифта можно только дублированием пикселей, что при большом увеличении приводит к формированию символов довольно "грубой" формы. Выполнять какие-либо геометрические преобразования растрового шрифта бессмысленно. Кроме того, поскольку растровые шрифты, как правило, хранятся в постоянной памяти, они не переносятся с компьютера на компьютер.

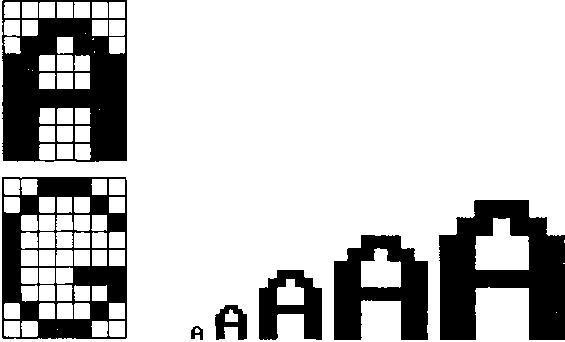


Рис. 4.1 Растровыйшрифт

Символы как штриховых, так и растровых шрифтов формируются из других примитивов, а потому ***в составе основной библиотеки OpenGL нет специального примитива для формирования текста***. Но в составе дополнительной библиотеки GLUT имеется несколько наборов символов (как штриховых, так и растровых), которые определены программно, следовательно, являются переносимыми.

Растровый символ GLUT можно получить с помощью функции

**glutBitmapCharacter (font, character)**

Параметр **font** определяет ***начертание*** шрифта и может принимать значение одной из символьных констант GLUT:

* **GLUT\_BITMAP\_8\_BY\_13;**
* **GLUT\_BITMAP\_9\_BY\_15;**
* **GLUT\_BITMAP\_TIMES\_ROMAN\_10;**
* **GLUT\_BITMAP\_HELVETICA\_12;**
* **GLUT\_BITMAP\_HELVETICA\_18.**

Параметру ***character*** при­сваивается либо ***код ASCII***, либо ***отдельный символ***, который мы хотим изобра­зить. Таким образом, чтобы изобразить прописную букву "А", можно восполь­зоваться либо значением 65 кода ASCII, либо обозначением 'А'.

Каждый символ, созданный с помощью функции **glutBitmapCharacter**, изображается так, что начало координат (нижний левый угол) битового массива на­ходится в ***текущем*** растровом положении. После того как битовый массив символа загружается в буфер регенерации, к координате ***х*** текущего растрового положения добавляется смещение, равное ширине символа.

Изменение значения текущей позиции растра выполняется с помощью разных модификаций функции **glRasterPos\*** (например, **glRasterPos2i(GLint x, GLint y)**, **glRasterPos3i(GLint x, GLint y, z GLint)**).

При установке растрового положения текущий ***цвет*** растровых изображений устанавливается равным текущему цвету. В качестве примера изобразим известную тек­стовую строку «Hellow Word!!!».

MyText[]=”Hellow Word!!!”;

glColor3ub(255,0,0); // Цвет текста

glRasterPos2i(-10,3); // Координаты начала вывода

len=strnlen(MyText2,100);

for(inti=0;i<len;i++) glutBitmapCharacter(GLUT\_BITMAP\_9\_BY\_15, MyText[i]);

Эскизный символ изображается путем вызова следующей функции.

**glutStrokeCharacter (font, character)**

Для этой функции параметру **font** можно присвоить значение **GLUT\_STROKE\_ ROMAN**, чтобы изобразить ***пропорциональный*** шрифт, или **GLUT\_STROKE\_MONO\_ ROMAN**, чтобы представить ***моноширинный*** шрифт. Размер и положение этих символов можно контролировать с помощью операций афинных преобразований, которые вызываются перед выполнением процедуры **glutStrokeCharacter**.

Работа с z–буфером

Как уже было сказано, OpenGL поддерживает использование аппарат­ного буфера для удаления невидимых поверхностей. Каждое поступающее значение глубины фрагмента сравнивается с имеющимся в буфере глубины и выводится на экран (или нет) в зависимости от результа­тов выполнения этого теста. Более того, результат этого теста влияет на содержимое буфера трафарета, что в совокупности представляет поразительно мощное средство создания изо­бражений практически любой степени сложности.

Исходно тест глубины заблокирован и может быть разрешен при помощи следующей команды:

**glEnable (GL\_DEPTH\_TEST)**

Выключить эту проверку можно командой

**glDisable (GL\_DEPTH\_TEST)**

Функция сравнения, ис­пользуемая в тесте, задается командой

**void glDepthFunc(GLenum func)**

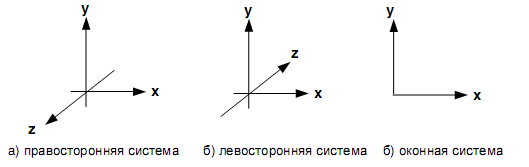
Вызов этой команды позволяет определить ту функцию, которая будет ис­пользоваться для сравнения каждого поступающего **z**-значения с тем, кото­рое хранится в буфере глубины. Функцию, используемую для сравнения глубин, определяет параметр **func**, который может принимать одно из сле­дующих значений:

|  |  |
| --- | --- |
| Константа | Тест завершается положительно... |
| GL\_NEVER | Никогда |
| GL\_LESS | Если поступающее значение меньше, чем хранящееся в буфере |
| GL\_EQUAL | Если поступающее z-значение равно хранящемуся в буфере |
| GL\_LEQUAL | Если поступающее z-значение меньше или равно, чем хранящееся в буфере |
| GL\_GREATER | Если поступающее значение больше, чем хранящееся в буфере |
| GL\_NOTEQUAL | Если поступающее значение не равно хранящемуся в буфере |
| GL\_GEQUAL | Если поступающее значение больше или равно хранящемуся в буфере |
| GL\_ALWAYS | Всегда |

***По умолчанию*** используется константа **GL\_LESS**. Если буфера глубины нет, то считается, что тест всегда завершается положительно.

## 32. Графическая библиотека OpenGL. Системы координат. Модельно-видовые преобразования. Примеры

В OpenGL используются как основные три системы координат: ***левосторонняя, правосторонняя и оконная***. Первые две системы являются трехмерными и отличаются друг от друга направлением оси **Z**: в правосторонней она направлена на наблюдателя, в левосторонней – в глубину экрана. Ось **X** направлена вправо относительно наблюдателя, ось **Y** – вверх.

Рис. 4

Рассматривая какой-либо трехмерный объект, мы всегда определяем его по­ложение и размеры относительно некоторой привычной, и удобной в на­стоящий момент системы координат, связанной с реальным миром. Такая исходная система координат в компьютерной графике является **правосто­ронней** и называется **мировой системой координат (МСК).** В ней над объектами выполняются заданные действия, например аффинные преобразования. Для того чтобы можно было изобразить объект на экране, его необходимо предварительно перевес­ти (или преобразовать) в другую систему координат, которая связана с точ­кой наблюдения и носит название **видовой системы координат или системой координат наблюдателя (СКН).** Эта система координат является левосторонней. В ней рассчитываются цвета объектов и выполняются заданные отсечения. Далее, координаты **СКН** переводятся в ***нормированные (нормализованные)*** координаты **(НК)**, которые лежат в диапазоне **[ -1, 1 ]**. И, наконец, любое трехмерное изображение мы всегда рисуем на двумерном экране, который имеет свою **экранную (оконную) систему координат (ОСК)**. Взаимное расположение мировой (XYZ) и видовой (XeYeZe ) систем координата также экранная система координат (xy) показаны на рис. 5 и рис. 6 соответственно.

|  |  |
| --- | --- |
| E  0  Ze  Xe  Ye  Y  X  Z | 0  Ye  x  0  Xe  y  Ze |
| Рис. 5 | Рис. 6 |

Таким образом, основная задача, которую необходимо решить, заключается в том, что объекты, описанные в мировых координатах, необходимо изобра­зить на плоской области вывода экрана. То есть необходимо проделать неко­торую последовательность преобразований и осуществить проекцию, перево­дящую трехмерные объекты на двумерную проекционную плоскость. Эту по­следовательность действий удобно выполнять в несколько этапов, как это и реализовано в OpenGL (рис.7)

Преобразования в

МСК

МСК

(правосторонняя)

СКН

(левосторонняя)

НК

Преобразования

проецирования

Преобразования области

вывода

Оконные координаты

Рис. 7

Координаты отображаемых объектов (примитивов) задаются в МСК. В ней же под воздействием ***видовой матрицы*** выполняются заданные действия, например аффинные преобразования. В результате воздействия видовой матрицы координаты вершин объектов пересчитываются в СКН. В ней рассчитываются цвета объектов и выполняются заданные отсечения. Далее под воздействием ***матрицы проецирования*** координаты СКН пересчитываются в ***нормированные*** координаты. Диапазон их изменения – **[-1.0, 1.0].** Непосредственно перед выводом в окно нормированные координаты преобразуются в оконные.

**Матрицы**

В OpenGL предусмотрены три типа матриц – ***видовая, проекций и текстуры***, все они имеют размер (4x4).

Для того чтобы с какой-либо матрицей можно было работать, ее необходимо сделать текущей, для чего предусмотрена команда

**void glMatrixMode(GLenum mode)**

Параметр **mode** определяет, с каким набором матриц будут выполняться по­следующие операции, и может принимать одно из трех значений:

|  |  |
| --- | --- |
| **GL\_MODELVIEW** | Последовательность операций над матрицами применяется к видовой матрице |
| **GL\_PROJECTION** | Последовательность операций над матрицами применяется к матрице проекции |
| **GL\_TEXTURE** | Последовательность операций над матрицами применяется к матрице текстуры |

После того как установлена текущая матрица, необходимо определить ее элементы. Для этой цели можно использовать следующие команды

**void glLoadMatrix[f d](GLtype\* m)**

Параметр **m** определяет указатель на матрицу 4x4, хранящуюся по порядку расположения столбцов как 16 последовательных вещественных значений одинарной (f) или двойной (d) точности.



Эта команда заменяет текущую матрицу на ту, которая определена в **m** Те­кущей является одна из матриц – ***видовая, проекций*** или ***текстуры***, в зави­симости от текущего режима.

Команда

**void gILoadIdentity()**

заменяет текущую матрицу на единичную (матрицу идентичности).



Семантически эта команда эквивалентна **glLoadMatrix** с параметром, определяю­щим единичную матрицу, но выполняется более эффективно.

Часто бывает необходимо сохранить содержимое текущей матрицы для дальнейшего использования, для чего применяются команды

**void glPushMatrix (void)**,

**void glPopMatrix (void)**

Они записывают и восстанавливают текущую матрицу из стека, причем для каждого типа матриц стек свой. Для модельно-видовых матриц его глубина равна как минимум 32, для остальных – как минимум 2.

При выполнении преобразований объектов, таких как ***перенос***, ***вращение***, ***масштабирование***, могутпонадобиться соответствующие операции над мат­рицами. OpenGL предоставляет набор команд для решения этой задачи. Фундаментальной является команда перемножения матриц

**void gIMultMatrix[f d](GLtype\* m)**,

где параметр **m** определяет указатель на матрицу размером 4x4, хранящуюся по порядку расположения столбцов. Эта команда умножает матрицу, заданную параметром **m** ***слева*** на текущую. Если обозначить текущую матрицу за **М**, передаваемую матрицу за **T**, то в результате выполнения команды glMultMatrix текущей становится матрица **M∙T** (T← M∙T). Однако обычно для изменения матрицы того или иного типа удобно использовать ***специальные команды***, которые по значениям своих параметров создают нужную матрицу и умножают ее на текущую.

Однако к использованию этой следует прибегать только в самом крайнем случае, т. к. для перемножения требуются значительные вычислительные затраты.

В целом, для отображения трехмерных объектов сцены в окно приложения используется последовательность, показанная на рис. 8

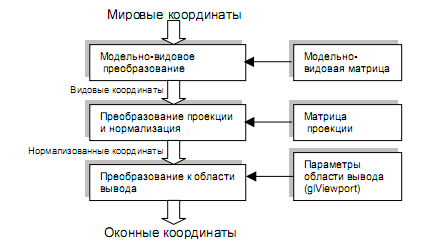


Рис. 8

Таким образом, все преобразования объектов и камеры в OpenGL производятся с помощью ***умножения векторов координат на матрицы***. Причем умножение происходит ***на текущую матрицу в момент определения координаты командой glVertex\* и некоторыми другими***.

**Модельно-видовые преобразования**

К модельно-видовым преобразованиям будем относить перенос, поворот и изменение масштаба вдоль координатных осей. Для проведения этих операций достаточно умножить на соответствующую матрицу каждую вершину объекта и получить измененные координаты этой вершины:



где M – матрица модельно-видового преобразования. Перспективное преобразование и проектирование производится аналогично. Сама матрица может быть ***создана*** с помощью следующих команд:

**void glTranslate[f d] (GLtype x, GLtype y, GLtype z)**,

**void glRotate[f d] (GLtype angle, GLtype x, GLtype y, GLtype z)** ,

**void glScale[f d] (GLtype x, GLtype y, GLtype z)**.

Где

**glTranlsate\*()** – производит ***перенос объекта***, ***прибавляя*** к координатам его вершин значения своих параметров.

**glRotate\*()** – производит ***поворот объекта против часовой стрелки*** на угол ***angle*** (***измеряется в градусах***) **вокруг вектора (x, y, z)**.

**glScale\*()** – производит ***масштабирование*** объекта (***сжатие или растяжение***) ***вдоль вектора (x, y ,z)***, умножая соответствующие координаты его вершин на значения своих параметров.

Все эти преобразования ***изменяют текущую матрицу***, а поэтому применяются к примитивам, ***которые определяются позже***. В случае, если надо, например, ***повернуть*** один объект сцены, а другой ***оставить неподвижным***, удобно сначала сохранить текущую видовую матрицу в стеке командой **glPushMatrix()**, затем вызвать **glRotate()** с нужными параметрами, описать примитивы, из которых состоит этот объект, а затем восстановить текущую матрицу командой **glPopMatrix().**

Кроме изменения положения самого объекта, часто бывает необходимо изменить положение наблюдателя, что также приводит к изменению модельно-видовой матрицы. Это можно сделать с помощью команды

**void gluLookAt (GLdouble ex, GLdouble ey, GLdouble ez,**

**GLdouble center\_x, GLdouble center\_y, GLdouble center\_z,**

**GLdouble up\_x, GLdouble up\_y, GLdouble up\_z)**

где точка **(ex, ey, ez)** определяет точку наблюдения в МСК,

точка **(center\_x, center\_y, center\_z)** задает центр сцены, который будет проектироваться в центр области вывода, а вектор **(up\_x, up\_y, up\_z)** задает направление вектора, вдоль которого в ***видовом порте*** (см. ниже) ориентируется ось **Y**, определяя поворот камеры. Причем вектор **(up\_x, up\_y, up\_z)** не должен быть параллелен линии, проведенной от точки наблюдения **(ex, ey, ez)** до наблюдаемой точки **(center\_x, center\_y, center\_z)** .

Если, например, камеру не надо поворачивать, то задается значение **(0,1,0)**, а со значением **(0,-1,0)** сцена будет перевернута.

Строго говоря, эта ***команда совершает перенос и поворот объектов сцены***, но в таком виде задавать параметры бывает удобнее. Следует отметить, что ***вызывать команду* gluLookAt()** имеет смысл перед определением преобразований объектов, когда модельно-видовая матрица равна ***единичной***.

**Замечание** . В общем случае матричные преобразования в OpenGL нужно записывать в ***обратном*** порядке. Например, необходимо сначала ***повернуть*** объект, а затем ***передвинуть*** его, то сначала надо вызвать команду **glTranslate()**, а только потом – **glRotate()**. После этого можно определить сам объект.

## 33. Графическая библиотека OpenGL. Проекции. Область вывода. Создание анимации. Примеры

Несоответствие между пространственными объектами и плоскими изображениями устраняется путем введения ***проекций***, которые отображают объекты на двумерной проекционной картинной плоскости. В зависимости от расстояния между наблюдателем и картинной плоскостью различают два основных класса проекций: ***параллельные*** и ***центральные (перспективные)***. Если это расстояние бесконечно, то проекция будет праллельной, а если конечно, то центральной. При описании центральной проекции необходимо явно задавать ***центр проекции***, а для определения параллельной проекции необходимо определить напрвление проецирования.

OpenGL поддерживает два вида прекций – ***ортографическую (параллельную)*** и ***перспективную*** проекции.

В OpenGL существуют стандартные команды для задания ортографической (параллельной) и перспективной проекций, выполнение которых ***устанавливает в системе координат наблюдателя область видимости графических данных***. Первый тип проекции может быть задан командами

**void glOrtho (GLdouble x\_left, GLdouble x\_right, GLdouble y\_bottom, GLdouble y\_top,**

**GLdouble z\_near, GLdouble z\_far)**

**void gluOrtho2D (GLdouble x\_left, GLdouble x\_right, GLdouble y\_bottom,**

**GLdouble y\_top)**

Первая команда создает матрицу проекции ***в усеченный объем видимости (параллелепипед видимости)*** в левосторонней системе координат. Параметры команды задают точки (**x\_left, y\_bottom, z\_near**) и (**x\_right, y\_top, z\_far**), которые отвечают левому нижнему и правому верхнему углам окна вывода. Параметры **z\_near** и **z\_far** задают расстояние до ближней и дальней плоскостей отсечения по удалению от точки **(0,0,0)** и могут быть ***отрицательными***.

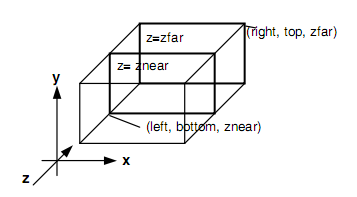


Рис. 9

Во второй команде, в отличие от первой, значения **znear** и **zfar** устанавливаются равными **–1** и **1** соответственно. Это удобно, если OpenGL используется для рисования двумерных объектов. В этом случае положение вершин можно задавать, используя команды **glVertex2\*().**

После преобразования ортографического проецирования координаты любых точек внутри преобразованной трехмерной области видимости находятся в диапазоне от **-1** до **1**. Таким образом, преобразованная область видимости представляет собой ***куб со стороной равной двум***. Напомним, что координаты, полученные после преобразования называются ***нормированными (нормализованными)***.

При использовании ***перспективной проекции*** в качестве области видимости необходимо определить не параллелепипед, а ***усеченную пирамиду*** (рис.10). Для этого в OpenGL реализованы две команды.

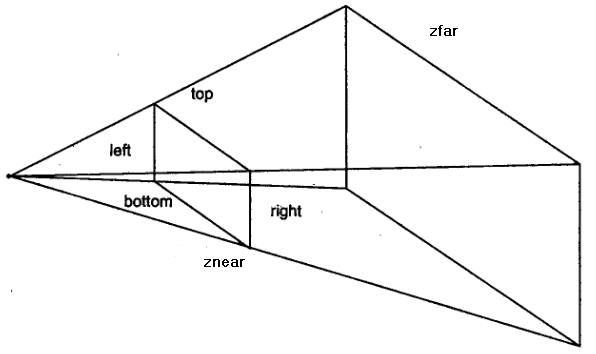


Рис. 10

Первая комманда

**void glFrustrum(GLdouble left, GLdouble right, GLdouble top,**

**GLdouble bottom, GLdouble znear, GLdouble zfar)**

Эта комманда создает матрицу перспективы. Параметры **left** и **right** определяют координаты ***левой*** и ***правой вертикальных***, а **bottom** и **top** – ***нижней*** и ***верхней горизонтальных*** плоскостей отсечения. Параметры **znear** и **zfar** – расстояния до ***ближней*** и ***дальней*** плоскостей отсечения по глубине, причем ***оба значения*** должны быть ***положительными***. Считается, что ***наблюдатель располагается*** в точке с координатами **(0, 0, 0)**.

Вторая комманда

**void gluPerspective (GLdouble angley, GLdouble aspect,**

**GLdouble znear, GLdouble zfar)**

которая также задает ***усеченную пирамиду видимости*** в левосторонней (видовой) системе координат (рис. 11).

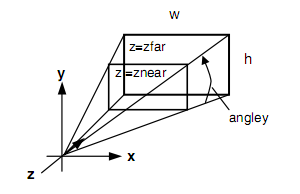


Рис. 11

Параметр **angley** определяет ***угол видимости в градусах по оси Y***, **tg(angley/2) = top/near,** и должен находиться в диапазоне от 0 до 180. ***Угол видимости вдоль оси X*** задается параметром **aspect**, который обычно задается как ***отношение*** ширины ***(width)*** и высоты ***(height)*** сторон области вывода (как правило, размеров окна). Величина отношения **aspect** должна соответствовать заданному видовому порту (см. ниже). Например, если **aspect=2,** то отношение ***width/ height*** (x/y) также следует задать равным двум.

Параметры **zfar** и **znear** задают расстояние от наблюдателя до плоскостей отсечения по глубине и должны быть ***положительными***. Действие этой команды эквивалентно вызову **glFrustum** при **left = -right, bottom = -top, tg(angley/2) = top/near, aspect = right/top**.

Чем больше отношение **zfar/znear**, тем хуже в буфере глубины будут различаться расположенные рядом поверхности, так как по умолчанию в него будет записываться «сжатая» глубина в диапазоне от 0 до 1 (см. далее).

Также как и для случая ортографической проекции, усеченный конус видимости отображается на куб ***со стороной равной двум***.

Для задания матрицы проекций необходимо ***включить режим работы*** с нужной матрицей командой **glMatrixMode(GL\_PROJECTION)** и сбросить текущую, вызвав **glLoadIdentity()**.

Пример:

//ортографическая проекция

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

glOrtho(0, w, 0, h, -1.0, 1.0);

**Область вывода**

***Область вывода*** представляет собой прямоугольник в оконной системе координат, размеры которого задаются командой:

**void glViewPort (GLint x, GLint y, GLint width, GLint height)**

Значения всех параметров задаются в ***пикселах***. Параметры **x** и **y** определяют координаты ***левого нижнего угла*** прямоугольника ***вывода в оконной системе координат*** (видовой порт)и ***по умолчанию принимаются равными нулю***. Размеры оконной системы координат определяются текущими размерами окна приложения, точка **(0,0)** находится в ***левом нижнем углу*** окна.

**РИСУНОК СИСТЕМ КООРДИНАТ–?**

Команда **glViewPort** задает аффинные преобразования области вывода, переводящие нормированные координаты **(xn, yn)** в оконные координаты **(xw, yw)**

Преобразование выполняется по формулам:

, ,

где  – целая часть числа.

Используя параметры команды **glViewPort()**, OpenGL вычисляет оконные координаты ***центра области вывода***  по формулам

, 

При этом точка с координатами **(0, 0, 0)** располагается в ***центре*** области вывода.

Пусть

, .

Тогда оконные координаты вершин можно представить в виде

, 

Команда **glViewPort()** обычно используется в функции, зарегистрированной с помощью команды **glutReshapeFunc()**, которая вызывается, если пользователь изменяет размеры окна приложения.

**СОЗДАНИЕ АНИМАЦИИ В OpenGL**

Для создания анимации в OpenGL необходимо выпонить следующие действия.

Установить режим двойной буферизации

glutInitDisplayMode(GLUT\_RGB|**GLUT\_DOUBLE**)

Зарегистрировать и создать функцию обратного вызова (например функцию с именем IdleFunction), которая вызывается системой когда нет других событий.

Внутри функции IdleFunction описываются действия, которые влияют на вид изображения после перерисовки окна. Например, изменяется угол поворота объекта.

void IdleFunction( )

{

// Необходимые действия

…………………………..

glutPostRedisplay( ); // Обновить окно

}

В функции рисования OnDraw( ) использовать команду переключения буферов – glutSwapBuffers().

Для управления процессом запуска и остановки анимации можно использовать команды:

* glutIdleFunc(IdleFunction) – разрешает выполнение функции IdleFunction;
* glutIdleFunc(NULL) – блокирует выполнение функции IdleFunction.

## 34. Графическая библиотека OpenGL. Материалы и освещение. Примеры

Для создания реалистичных изображений необходимо определить как свойства самого объекта, так и свойства среды, в которой он находится. Первая группа свойств включает в себя параметры материала, из которого сделан объект, способы нанесения текстуры на его поверхность, степень прозрачности объекта. Ко второй группе можно отнести количество и свойства источников света, уровень прозрачности среды, а также модель освещения. Все эти свойства можно задавать, вызывая соответствующие команды OpenGL.

**Модель освещения**

В OpenGL используется модель освещения, в соответствии с которой цвет точки определяется несколькими факторами: ***свойствами материала и текстуры***, ***величиной нормали в этой точке***, а также ***положением источника света и наблюдателя***. Для корректного расчета освещенности в точке надо использовать единичные нормали, однако команды типа **glScale\*()**, могут изменять длину нормалей. Чтобы это учитывать, необходимо использовать режим нормализации векторов нормалей, который включается вызовом команды **glEnable(GL\_NORMALIZE)**.

Для задания глобальных параметров освещения используются команды

**void glLightModel[i f] (GLenum pname, GLenum param)**,

**void glLightModel[i f]v (GLenum pname, const GLtype \*params)**.

Аргумент **pname** определяет, какой параметр модели **param** освещения будет настраиваться. Возможные значения аргументов **pname** и **param** приведены в таблице.

|  |  |
| --- | --- |
| Значение **pname** | Значение **param** |
| **GL\_LIGHT\_MODEL\_LOCAL\_VIEWER** | Параметр **param** должен быть ***булевым*** и задает положение наблюдателя. Если он равен **GL\_FALSE**, то направление обзора считается параллельным оси **Z** вне зависимости от положения в видовыx координатах. Если же он равен **GL\_TRUE**, то наблюдатель находится в начале видовой системы координат. Это может улучшить качество освещения, но усложняет его расчет. Значение ***по умолчанию***: **GL\_FALSE.** |
| **GL\_LIGHT\_MODEL\_TWO\_SIDE** | Параметр **param** должен быть ***булевым*** и управляет режимом расчета освещенности, как для лицевых, так и для обратных граней. Если он равен **GL\_FALSE**, то освещенность рассчитывается только для лицевых граней. Если же он равен **GL\_TRUE**, расчет проводится и для обратных граней. Значение ***по умолчанию***: **GL\_FALSE**. |
| **GL\_LIGHT\_MODEL\_AMBIENT** | Параметр **params** должен содержать четыре целых или вещественных числа, которые определяют ***цвет фонового освещения*** даже в случае ***отсутствия*** определенных источников света. Значение ***по умолчанию:*** **(0.2, 0.2, 0.2,1.0)**. |

Спецификация материалов

Для задания параметров текущего материала используются команды

**void glMaterial[i f] (GLenum face, GLenum pname, GLtype param),**

**void glMaterial[i f]v (GLenum face, GLenum pname, GLtype \*params)**

С их помощью можно определить ***рассеянный***, ***диффузный*** и ***зеркальный***  цвета материала, а также степень зеркального отражения и интенсивность излучения света, если объект должен светиться. Какой именно параметр будет определяться значением **param**, зависит от значения **pname**.

Возможные значения аргументов **pname** и **param** приведены в таблице.

|  |  |
| --- | --- |
| Значение **pname** | Значение **param** |
| **GL\_AMBIENT** | Параметр **params**  должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют ***рассеянный цвет*** материала (цвет материала в тени). Значение ***по умолчанию***: **(0.2, 0.2, 0.2, 1.0)** |
| **GL\_DIFFUSE** | Параметр **params** должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют ***диффузный цвет*** материала. Значение ***по умолчанию:*** **(0.8, 0.8, 0.8, 1.0)**. |
| **GL\_SPECULAR** | Параметр **params** должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют ***зеркальный цвет*** материала. Значение ***по умолчанию:* (0.0, 0.0, 0.0, 1.0)**. |
| **GL\_SHININESS** | Параметр **params** должен содержать одно целое или вещественное значение в диапазоне от **0** до **128**, которое определяет ***степень зеркального отражения*** материала. Значение ***по умолчанию:*** **0**. |
| **GL\_EMISSION** | Параметр **params**  должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют ***интенсивность излучаемого света***  материала. Значение ***по умолчанию:*** **(0.0, 0.0, 0.0, 1.0)**. |
| **GL\_AMBIENT\_AND\_DIFFUSE** | Этоэквивалентно двум вызовам команды glMaterial\*() со значением **pname** **GL\_AMBIENT** и **GL\_DIFFUSE** и одинаковыми значениями **params**. |

Из таблицы следует, что вызов команды **glMaterial[i f]()** возможен только для установки степени зеркального отражения материала ***(shininess)***. Команда **glMaterial[i f]v()** используется для задания остальных параметров.

Параметр **face** определяет тип граней, для которых задается этот материал и может принимать значения: **GL\_FRONT**, **GL\_BACK** или **GL\_FRONT\_AND\_BACK**.

Если в сцене материалы объектов различаются лишь одним параметром, рекомендуется сначала установить нужный режим, вызвав **glEnable()** c параметром **GL\_COLOR\_MATERIAL**, а затем использовать команду

**void glColorMaterial (GLenum face, GLenum pname)**,

где параметр **face** имеет аналогичный смысл, а параметр **pname** может принимать все перечисленные в таблице значения. После этого значения выбранного с помощью **pname** свойства материала для конкретного объекта (или вершины) устанавливаются вызовом команды **glColor\*()**, что позволяет избежать вызовов более ресурсоемкой команды **glMaterial\*()** и повышает эффективность программы.

Пример определения свойств материала:

float mat\_dif[]={0.8,0.8,0.8};

float mat\_amb[] = {0.2, 0.2, 0.2};

float mat\_spec[] = {0.6, 0.6, 0.6};

float shininess = 0.7 \* 128;

…

glMaterialfv (GL\_FRONT\_AND\_BACK,GL\_AMBIENT, mat\_amb);

glMaterialfv (GL\_FRONT\_AND\_BACK,GL\_DIFFUSE, mat\_dif);

glMaterialfv (GL\_FRONT\_AND\_BACK,GL\_SPECULAR, mat\_spec);

glMaterialf (GL\_FRONT,GL\_SHININESS, shininess);

**Описание источников света**

Определение свойств материала объекта имеет смысл, только если в сцене есть источники света. Иначе все объекты будут черными (или, строго говоря, иметь цвет, равный рассеянному цвету материала, умноженному на интенсивность глобального фонового освещения, см. команду **glLightModel**). Добавить в сцену источник света можно с помощью команд

**void glLight[i f] (GLenum light, GLenum pname, GLfloat param)**,

**void glLight[i f]v (GLenum light, GLenum pname, GLfloat \*params)**

Параметр **light** однозначно определяет источник света. Он выбирается из набора специальных символических имен вида **GL\_LIGHTi** , где **i** должно лежать в диапазоне от **0** до константы **GL\_MAX\_LIGHT**, которая ***обычно не превосходит восьми***.

Параметры **pname** и **params** имеют смысл, аналогичный команде **glMaterial**

Возможные взаимные значения аргументов **pname** и **param** приведены в таблице.

|  |  |
| --- | --- |
| Значение **pname** | Значение **param** |
| **GL\_SPOT\_EXPONENT** | Параметр **param** должен содержать целое или вещественное число от **0** до **128**, задающеераспределение интенсивности света. Этот параметр описывает уровень сфокусированности источника света. Значение ***по умолчанию***: **0** (рассеянный свет). |
| **GL\_SPOT\_CUTOFF** | Параметр **param** должен содержать целое или вещественное число между **0** и **90** или равное **180**, которое определяет максимальный угол разброса света. Значение этого параметра есть половина угла в вершине конусовидного светового потока, создаваемого источником. Значение ***по умолчанию***: **180** (рассеянный свет). |
| **GL\_AMBIENT** | Параметр **params** должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют цвет фонового освещения. Значение ***по умолчанию***: **(0.0, 0.0, 0.0, 1.0).** |
| **GL\_DIFFUSE** | Параметр **params** должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют цвет диффузного  освещения. Значение ***по умолчанию***: **(1.0, 1.0, 1.0, 1.0)**  для **GL\_LIGHT0**  и **(0.0, 0.0, 0.0, 1.0)** для остальных. |
| **GL\_SPECULAR** | Параметр **params** должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют цвет зеркального отражения. Значение ***по умолчанию:*** **(1.0, 1.0, 1.0, 1.0)**  для **GL\_LIGHT0** и **(0.0, 0.0, 0.0, 1.0)** для остальных. |
| **GL\_POSITION** | Параметр **params**  должен содержать четыре целых или вещественных числа, которые определяют положение источника света. Если значение компоненты **w = 0.0**, то источник считается бесконечно удаленным и при расчете освещенности учитывается только ***направление*** ***на точку*  (x, y, z)**, в противном случае считается, что источник расположен ***в точке* (x, y, z, w)**. В первом случае ослабления света при удалении от источника непроисходит, т.е. источник считается бесконечно удаленным. Значение ***по умолчанию:*** **(0.0, 0.0, 1.0, 0.0)**. |
| **GL\_SPOT\_DIRECTION** | Параметр **params** должен содержать четыре целых или вещественных числа, которые определяют направление света. Значение ***по умолчанию***: (**0.0, 0.0, -1.0, 1.0)**. Эта характеристика источника имеет смысл, если значение **GL\_SPOT\_CUTOFF** ***отлично*** от 180 (которое, задано ***по умолчанию***). |
| **GL\_CONSTANT\_ATTENUATION, GL\_LINEAR\_ATTENUATION, GL\_QUADRATIC\_ATTENUATION** | Параметр **param** задает значение одного из трех приведенных коэффициентов, определяющих ослабление интенсивности света при удалении от источника. Допускаются только неотрицательные значения. Если источник не является направленным (см. **GL\_POSITION**), то ослабление обратно пропорционально сумме: attconstant+attlineard+ attquadratic d2,  где d – расстояние между источником света и освещаемой им вершиной, attconstant, attlinear и attquadratic равны параметрам, заданным с помощью констант  **GL\_CONSTANT\_ATTENUATION**, **GL\_LINEAR\_ATTENUATION** и **GL\_QUADRATIC\_ATTENUATION** соответственно. ***По умолчанию*** эти параметры задаются тройкой (1, 0, 0), и фактически ослабления не происходит. |

При ***изменении положения источника света*** следует учитывать следующий факт: в OpenGL источники света являются объектами, во многом такими же, как многоугольники и точки. На них распространяется основное правило обработки координат в OpenGL – параметры, описывающее положение в пространстве, преобразуются текущей модельно-видовой матрицей в момент формирования объекта, т.е. в момент вызова соответствующих команд OpenGL. Таким образом, формируя источник света одновременно с объектом сцены или камерой, ***его можно привязать к этому объекту***. Или, наоборот, сформировать ***стационарный*** источник света, который будет ***оставаться на месте***, пока другие объекты перемещаются.

Если ***тип*** и ***положение*** источника света не задаются, параметр **GL\_POSITION** принимает значения по умолчанию (**0.0, 0.0, 1.0, 0.0**), что указывает на удаленный источник, лучи света от которого распространяются по отрицательному направлению оси **z**.

Положение источника света включается в описание сцены и вместе с положе­ниями объектов преобразовывается в координаты наблюдения с помощью матриц геометричсского преобразования и преобразования точки наблюдения. Следователь­но, если требуется ***зафиксировать*** источник света относительно объектов на сцене, его положение задается ***после*** указания в программе геометрических преобразований и преобразований точки наблюдения. Однако, если требуется, чтобы источник света двигался при изменении точки наблюдения, его положение задается перед специ­фикацией преобразования точки наблюдения. Кроме того, если необходимо, чтобы источник света двигался вокруг стационарной сцены, к его положению можно при­менить преобразование трансляции или вращения.

Для использования освещения сначала надо установить соответствующий режим вызовом команды **glEnable(GL\_LIGHTNING)**, а затем включить нужный источник командой **glEnable(GL\_LIGHTi)**. Еще раз обратим внимание на то, что при ***выключенном*** освещении цвет вершины равен текущему цвету, который задается командами **glColor\*()**. При ***включенном*** освещении цвет вершины вычисляется исходя из информации о материале, нормалях и источниках света. При выключении освещения визуализация происходит быстрее, однако в таком случае приложение должно само рассчитывать цвета вершин.

## 35. Графическая библиотека OpenGL. Структура приложения с использованием библиотеки MFC

Если OpenGL используется в MFC, то необходимо заполнить структуру PIXELFORMATDESCRIPTOR и вызвать SetPixelFormat(). В структуре необходимо указать PFD\_SUPPORT\_OPENGL | PFD\_DOUBLEBUFFER (формат с поддержкой OpenGL и двойной буферизацией).

int CChildView::OnCreate(LPCREATESTRUCT lpCreateStruct)

{

if (CWnd::OnCreate(lpCreateStruct) == -1)

return -1;

PIXELFORMATDESCRIPTOR pfd = // Описатель формата

{

sizeof(PIXELFORMATDESCRIPTOR),// Размер структуры

1, // Номер версии

PFD\_DRAW\_TO\_WINDOW | // Поддержка GDI

PFD\_SUPPORT\_OPENGL | // Поддержка OpenGL

PFD\_DOUBLEBUFFER, // Двойная буферизация

PFD\_TYPE\_RGBA, // Формат RGBA, не палитра

24, // Количество плоскостей

// в каждом буфере цвета

24, 0, // Для компоненты Red

24, 0, // Для компоненты Green

24, 0, // Для компоненты Blue

24, 0, // Для компоненты Alpha

0, // Количество плоскостей буфера Accumulation

0, // То же для компоненты Red

0, // для компоненты Green

0, // для компоненты Blue

0, // для компоненты Alpha

32, // Глубина Z-буфера

0, // Глубина буфера Stencil

0, // Глубина буфера Auxiliary

0, // Теперь игнорируется

0, // Количество плоскостей

0, // Теперь игнорируется

0, // Цвет прозрачной маски

0 // Теперь игнорируется

};

if ( (pixelformat = ChoosePixelFormat(m\_pDC->GetSafeHdc(), &pfd)) == 0 )

{

MessageBox("ChoosePixelFormat failed");

return FALSE;

}

if (SetPixelFormat(m\_pDC->GetSafeHdc(), pixelformat, &pfd) == FALSE)

{

MessageBox("SetPixelFormat failed");

return FALSE;

}

}