Министерство образования и науки Российской Федерации

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

А.Г. ЗАДОРОЖНЫЙ, Д.В. ВАГИН,
Ю.И. КОШКИНА

ВВЕДЕНИЕ В ДВУМЕРНУЮ КОМПЬЮТЕРНУЮ ГРАФИКУ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ OpenGL

Утверждено
Редакционно-издательским советом университета
в качестве учебного пособия

НОВОСИБИРСК
2018

УДК 004.92(075.8)
З-156

Рецензенты:
канд. техн. наук, доцент *В.С. Карманов*
д-р техн. наук, профессор *М.Э. Рояк*

Работа подготовлена на кафедре прикладной математики НГТУ

Задорожный А.Г.

З-156 Введение в двумерную компьютерную графику с использованием библиотеки OpenGL: учебное пособие / А.Г. Задорожный, Д.В. Вагин, Ю.И. Кошкина. – Новосибирск: Изд-во НГТУ, 2018. – 103 с.

ISBN 978-5-7782-3601-1

В данном учебном пособии рассмотрены основные понятия и функции графической библиотеки OpenGL для работы с двумерной компьютерной графикой. Пособие содержит большое количество примеров на языке C++ и может быть рекомендовано как для самостоятельного изучения курсов «Компьютерная графика» и «Вычислительная геометрия», так и для подготовки к лабораторным и расчетно-графическим заданиям.

УДК 004.92(075.8)

ISBN 978-5-7782-3601-1

ВВЕДЕНИЕ В КОМПЬЮТЕРНУЮ ГРАФИКУ

В общем случае *компьютерная графика* – это область информатики, занимающаяся формированием изображений с помощью компьютера.

Компьютерная графика прошла путь от простейшего аппаратно-программного комплекса Sketchpad (первого векторного редактора, разработанного в 1963 г.), до современных систем, позволяющих создавать реалистические изображения, не уступающие фотографическим снимкам.

Сейчас компьютерная графика используется в самых различных областях, в зависимости от которых выделяют следующие виды:

- *научная* и *деловая графика* (визуализации процессов или результатов);
- *конструкторская графика* (работа с чертежами в системах автоматизации проектирования);
- *иллюстративная графика* (работа с произвольными изображениями в графических редакторах);
- *анимационная графика* (работа с видео и разработка компьютерных игр)

Можно выделить следующие основные направления и соответствующие задачи современной компьютерной графики:

1) *Изобразительная компьютерная графика*:
   - ✓ построение и преобразование модели объекта;
   - ✓ генерация и изменение изображения;
   - ✓ идентификация объекта и получение требуемой информации;

2) *Обработка и анализ изображений*:

- ✓ повышения качества изображения;
- ✓ оценка изображения (определение формы, позиции и других параметров требуемых объектов);
- ✓ распознавания образов (выделения и классификации свойств объектов);

3) *Персептивная компьютерная графика* (анализ сцен):

- ✓ исследованием абстрактных моделей графических объектов и взаимосвязей между ними;

4) *Когнитивная компьютерная графика* (только формирующееся новое направление для научных абстракций):

- ✓ визуализация тех знаний, для которых пока еще не существует символических описаний;
- ✓ поиск путей перехода от образа к формулировке гипотезы о механизмах и процессах, представленных этим образом;
- ✓ создание таких моделей представления знаний, в которых можно было бы однообразно представлять объекты, характерные и для логического, и для образного мышления.

ВИДЫ ИЗОБРАЖЕНИЙ

По способам задания изображений компьютерную графику можно разделить на двумерную (2D) и трехмерную (3D), растровую и векторную. Рассмотрим следующую классификацию:

- *растровая* (изображение представляется как набор пикселей);
- *воксельная* (изображение представляется как набор "трехмерных пикселей");
- *векторная* (изображение представляется как набор геометрических примитивов);
- *фрактальная* (изображение представляет собою самоподобную структуру, описываемую математическими уравнениями).

РАСТРОВАЯ ГРАФИКА

Растр – это матрица ячеек (*пикселей*), которые являются наименьшей единицей *растрового изображения*. Один пиксель (pixel, **pic**ture **cell**) может хранить информацию только об одном цвете, который и ассоциируется с данным пикселем.

Сам по себе пиксель не имеет размеров, поскольку является частью информационной модели изображения. В зависимости от расположения пикселей в пространстве различают квадратный, прямоугольный, гексагональный или иные типы растра.

Для устройств графического вывода (телевизор, принтер, и т.п.) пиксель является наименьшим физическим элементом матрицы изображения. Соответственно, *форма пикселя* определяется особенностями данного устройства: для принтера пиксели имеют круглую форму, а для жидкокристаллических дисплеев – квадратную.

Разрешение дисплея (экрана) определяется горизонтальным и вертикальным размерами выводимого изображения в пикселях, которое может быть без искажений выведено в полноэкранном режиме. Строго говоря, разрешение экрана служат лишь верхней оценкой, поскольку

часть пикселей может оказаться «битой» или неиспользуемой, соответственно, реальный вклад в изображения вносят только так называемые *эффективные пиксели* (effective pixel).

У большинства дисплеев пиксели физически состоят из триад (*субпикселей* красного, зеленого и синего цветов, расположенных рядом в определённой последовательности), которые можно разглядеть только на очень близком расстоянии.

Разрешающая способность характеризуется расстоянием между соседними пикселями, которое обычно измеряется количеством пикселей или точек на единицу длины (табл. 1).

Число бит, используемых для хранения цвета каждого пикселя, называется *глубиной* (*разрядностью*) *цвета*. При этом общее количество доступных цветов или оттенков (градаций) серого можно определить, если возвести число 2 в степень, равную количеству битов на пиксель.

Если изображение представляет собою градации серого (рис. 1) или другого цвета, тогда оно называется *полутоновым*. В частности, для 256 градаций серого достаточно одного байта на пиксель.

Самый простой тип растрового изображения – *монохромный (однобитовый*): пиксели могут принимать только два цвета (черный или белый). Если под цвет отводится 8 бит, то изображение является *восьмиразрядным* (*восьмибитовым*), а пиксели уже могут принимать 256 различных цветов. Цвета, описываемые 24 битами, часто называются *естественными цветами* (*true color*), а соответствующее изображение – *двадцатичетырехразрядным*.

В современных системах под цвет выделяется по 4 байта, из которых первые три отводятся под 24-разрядный цвет, а оставшийся либо не используется вовсе, либо используется как *альфа-канал* (alpha channel), отвечающий за прозрачность изображения в данном пикселе.

Рис. 1. Градации серого цвета

Таблица 1

Стандартные обозначения разрешающей способности

<table>
<tr><th colspan="2">Обозначение</th><th>Описание</th><th colspan="2">Примеры</th></tr>
<tr><td>dpi</td><td>dots
per inch</td><td>количество точек
на дюйм</td><td>принтер</td><td>1200×1200
dpi</td></tr>
<tr><td>ppi</td><td>pixels
per inch</td><td>количество пикселей
на дюйм</td><td>сканер</td><td>600
dpi</td></tr>
<tr><td>lpi</td><td>lines
per inch</td><td>количество линий
на дюйм</td><td colspan="2">графический планшет
(дигитайзер)</td></tr>
<tr><td>spi</td><td>samples
per inch</td><td>количество сэмплов
на дюйм,
плотность
дискретизации</td><td colspan="2">описание внутренних
процессов</td></tr>
<tr><td rowspan="3">pix
Mpix</td><td rowspan="3">разрешение
экрана
(матрицы)</td><td rowspan="3">количество пикселей
по горизонтали
и вертикали</td><td>Ultra HD
4K</td><td>3840×2160
8 Mpix</td></tr>
<tr><td>Full HD
1080p</td><td>1920×1080
2 Mpix</td></tr>
<tr><td>HD
720p</td><td>1280×720
1 Mpix</td></tr>
<tr><td rowspan="2">inch
см</td><td rowspan="2">диагональ
экрана</td><td rowspan="2">расстояние между
противоположными
углами экрана</td><td>21 дюйм</td><td>53 см</td></tr>
<tr><td>50 дюймов</td><td>127 см</td></tr>
<tr><td colspan="2" rowspan="2">соотношение
сторон,
пропорции
экрана</td><td rowspan="2">отношение числа
пикселей по горизон-
тали и вертикали</td><td>SDTV</td><td>4:3</td></tr>
<tr><td>HDTV</td><td>16:9</td></tr>
</table>

Система *TrueColor* сменила систему *HighColor* (*HiColor*) с ее 16-разрядной глубиною. В свою очередь, высокопрофессиональные графические системы используют уже систему цветов *DeepColor*, где глубина цвета достигает 48 бит. В частности, технологии работы с изображениями и видео *HDR* (High Dynamic Range) использует вещественные числа одинарной точности (интересно, что в таком случае возможны промежуточные яркости «белее белого» и даже «отрицательная яркость»).

Для экономии памяти можно отказаться от всего многообразия доступных цветов и ограничиться только малым (до 256) их набором – так называемой *палитрой цветов*. В этом случае на каждый пиксель теперь отводится всего по одному байту (для хранения номера цвета). Соответствующие изображения называются *индексированными*.

Поскольку обычные фотографии могут иметь тысячи цветов и их оттенков, то переход к фиксированной палитре приводит к существенной потере информации. Для компенсирования данной проблемы применяется технология *дизеринга*, создающая иллюзию глубины цвета при небольшом количестве доступных цветов. Идея метода заключается в том, чтобы получить отсутствующие цвета путем "перемешивания" имеющихся: например, для получения отсутствующего в палитре оранжевого цвета можно разместить рядом красные и желтые пиксели в шахматном порядке.

Растровое представление обычно используются для изображений с большим количеством деталей или оттенков. Для получения высококачественного изображения необходимо высокое разрешение, что приводит к большим затратам памяти. Другой недостаток – пустое место в старой позиции после перемещения части изображения. Последний недостаток – ухудшение качества при масштабировании. При уменьшении разрешения теряются мелкие детали и деформируются надписи. Увеличение же разрешения приводит либо к *ступенчатости/блочности* (новые пиксели получают цвет одного из соседних старых), либо к *размытию* (ухудшению резкости и яркости) изображения (новые пиксели получают цвета, интерполированные по соседним старым пикселям). При интерполяции в основном используются следующие уровни качества (по возрастанию сложности): «*Nearest Neighbor*» («Ближайший Сосед»), «*Bilinear*» («Билинейный») и «*Bicubic*» («Би-

кубический»). Первый вариант (как самый простой и быстрый) реализован во встроенном в Windows графическом редакторе *Paint*. Ставший уже эталонным (за счет своих способностей по обработке изображений) графический редактор *Photoshop* использует все уровни интерполяции, причем кубическая интерполяция у него представлена минимум в трех вариациях.

При выводе изображения на экран возникает такое понятие, как *кадровая частота* (Frame rate/frequency) – количество сменяемых кадров за единицу времени. Общепринятая единица измерения – кадры в секунду (*FPS*, frames per second). Для комфортного просмотра необходимо, чтобы FPS был больше 24 (желательно, 60), поскольку это и есть нижний физиологический предел заметности мерцания изображения.

Форматов хранения растровых изображений множество, но самым известным (и наиболее распространенным) является формат *BMP* (Bitmap Picture), разработанный компанией Microsoft.

Для хранения изображений фотографического качества давно стал классикой формат *JPEG* (Joint Photographic Experts Group), позволяющий сжимать изображения как с потерями информации, так и без (Lossless JPEG), при сохранении цветового многообразия.

При необходимости сохранения изображений (для сканирования или распознавании текста, в полиграфии) в различных цветовых пространствах (RBG, CMYK, LAB) и с различной глубиной цвета (8-64 бит) используется формат TIFF (Tagged Image File Format).

В цифровых камерах используется не имеющий четкой спецификации формат RAW, поступающий напрямую с матрицы фотокамеры.

Для передачи изображений по сетям в основном используются форматы *GIF* (Graphics Interchange Format) и *PNG* (Portable Network Graphics). Первый формат позволяет не только сжимать изображение, но и поддерживать анимацию и прозрачность (частично). Второй формат анимацию не поддерживает, зато корректно поддерживает прозрачность, практически неограниченное количество цветов (GIF ограничен всего 256 цветами), сжатие без потерь и прочее.

ВЕКТОРНАЯ ГРАФИКА

В отличие от растровой графики (где изображение представляет собою набор пикселей), *векторная графика* – это способ представления объектов и изображений на основе математического описания *примитивов* (элементарных геометрических объектов), таких как линии, многоугольники, сплайны, сферы и т.п. Соответственно, для вывода на растровые графические устройства векторные изображения должны быть предварительно *растеризованы* (преобразованы в растровый формат).

Например, для построения окружности (круга) в общем случае требуется задать следующие исходные данные:

- ✓ координаты центра окружности;
- ✓ величину радиуса;
- ✓ цвет заполнения;
- ✓ цвет и толщину контура;
- ✓ порядок плана (передний план, задний план).

Как можно видеть, описание окружности весьма компактно и практически не требует памяти для получения высококачественного изображения. Помимо этого, при изменении масштаба изображение остается корректным (за исключением вырожденных случаев), в частности, толщина контура меняться не будет.

Векторное представление позволяет корректно выполнять широкий спектр операций с объектом: поворот, перенос, масштабирование, вычисление различных проекций и разрезов и т.п. Потери при масштабировании могут возникнуть только при растеризации для сверхмалых разрешений типа 16x16 пикселей.

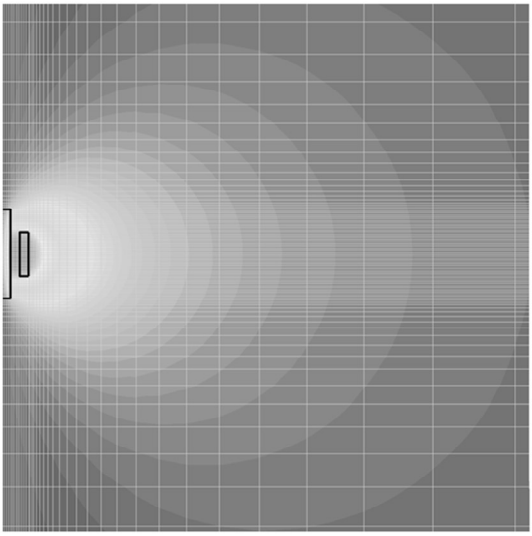
Векторная графика в основном используется в полиграфии и графических программных библиотеках. Кроме того, такие популярные компьютерные шрифты класса *TrueType*, как «Times New Roman» и «Arial», являются векторными, благодаря чему они одинаково выглядят как на экране, так и на бумаге.

На рис. 2 приведен пример сравнения результатов масштабирования для векторной и растровой графики на примере буквы «S» (шрифт «Times New Roman»). Векторное масштабирование было получено элементарно путем изменения размера шрифта. Растровый же вариант был получен путем шестикратного увеличения растровой картинки с буквой (нечеткость границы в этом варианте обуславливается самой особенностью шрифта, связанной со сглаживанием в целях улучшения читабельности текста). Видно, что при масштабировании качество векторного изображения не пострадало (в отличие от растрового).

Рис. 2. Пример шестикратного масштабирования буквы S

На рис. 3 приведено изображение, демонстрирующее распространение электромагнитного поля (рассчитанного в конечноэлементном программном комплексе «TELMA» на сетках с 5, 20 и 80 тысячами узлов), и пример сравнения размера файлов для хранения данного изображения в различных графических форматах (BMP, JPG и EMF). На Рис. 4 приведен результат масштабирования данного изображения для растрового (хранящего только пиксели) и векторного (хранящего цвета и конечноэлементную сетку) форматов. Как можно видеть, наличие большого количества элементов для векторного формата негативно сказывается на требованиях к памяти, зато при масштабировании качество не теряется.

В табл. 2 представлена сравнительная характеристика растровой и векторной графики.



Тип файла	Размер
BMP 1000*1000*24	3 Мб
JPG 1000*1000*24	0.1 Мб
EMF 5 тыс.узлов	5 Мб
EMF 20 тыс.узлов	10 Мб
EMF 80 тыс.узлов	30 Мб

Рис. 3. Сравнение эффективности графических форматов

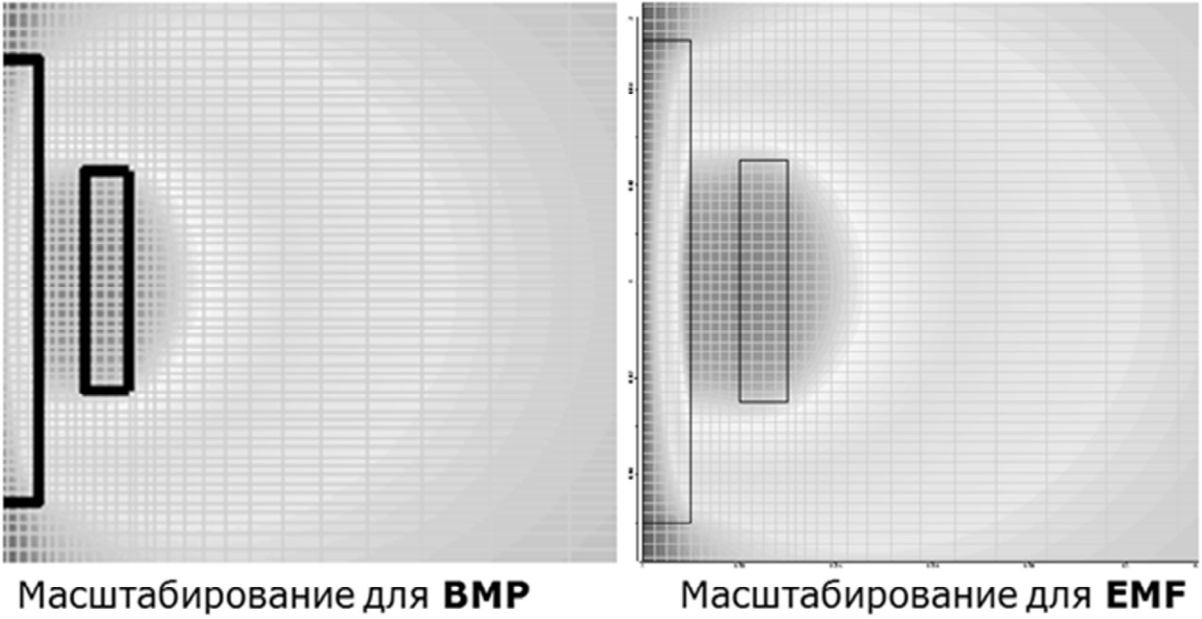


Рис. 4. Сравнение результатов масштабирования

Таблица 2

Сравнительная характеристика растровой и векторной графики

Критерий	Растровая графика	Векторная графика
объекты	пиксели	примитивы
сложность рисунка	любая	схематичная
распространенность	высокая	только для векторных графических систем
скорость обработки	высокая	низкая
размер файла	зависит только от разрешения	зависит только от количества объектов
масштабирование	с искажениями	без потерь

Форматов файлов векторной графики существует намного меньше, чем для растровой. Самыми популярными из них являются:

- *WMF / EMF* (Windows MetaFile / Enhanced Metafile),
- *SWF* (Small Web Format, формат платформы Adobe Flash),
- *CDR* (CorelDRaw, формат редактора CorelDraw),
- *AI* (формат редактора Adobe Illustrator).

В частности, формат SWF используется для флеш-анимации и может инкаспулировать даже аудио, в связи с чем приобрел такую популярность для web-дизайна.

ВОКСЕЛЬНАЯ ГРАФИКА

В общем случае растровая матрица может быть и трехмерной. В этом случае трехмерный (объемный) пиксель называется *вокселем* (voxel, **vo**lumetric pi**xel**), а графика, соответственно – *воксельной*. Изменяющийся во времени воксель получил название *доксель* (doxel, **d**ynamic v**oxel**). Таким образом, все воксельные модели состоят из кирпичиков-кубиков даже изнутри, а не только на поверхности.

Благодаря тому, что трехмерная матрица хранит значение вокселя для каждого единичного элемента объемного пространства, воксельные модели (Рис. 5) хорошо подходят для моделирования непрерывных сред, в отличии от полигональной модели, которая описывает только поверхность (внутри – "пустота").

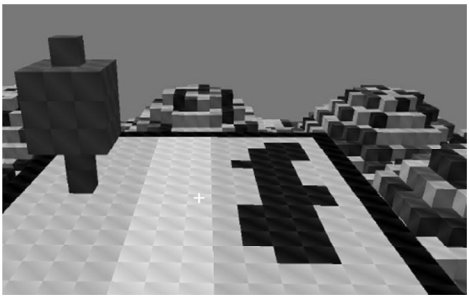


Рис. 5. Воксельный мир

Эта особенность полезна, например, в медицине. Такие медицинские комплексы, как компьютерный или магнитно-резонансный томографы, выдают послойную информацию при сканировании, из которой и формируется воксельная модель. Значением вокселей в этом случае может быть, например, величина прозрачности тела для рентгеновских лучей.

Также воксельная графика применяется и в ряде компьютерных играх – для моделирования *разрушаемого окружения* (множества объектов, которые могут быть частично или полностью разрушены) и генерации сложных ландшафтов, которые динамически деформируются и/или состоят из сложной системы туннелей и пещер. Воксельные графические движки использовались, например, в таких известных играх, как Delta Force, Crysis, Worms3D и Minecraft.

Серьезным недостаткам воксельной графики являются отсутствие аппаратной поддержки и чрезмерные требования к памяти. Проблемы с памятью вызваны необходимостью хранить трехмерный массив: например, для хранения небольшой модели размерностью

256×256×256 вокселей с восьмиразрядным цветом требуется целых 16Мб памяти.

Одним из вариантов решения проблемы с памятью является использование технологии *разреженного воксельного октодерева* (sparse voxel octree): корневой узел дерева является кубом, содержащим весь объект целиком, а каждый узел дерева имеет 8 кубов-потомков (либо не имеет их вовсе). Помимо значительной экономии памяти данная технология обладает следующими достоинствами: естественная генерация уровней детализации (аналога mipmap-карт или LOD-карт) и высокая скорость обработки в рейкастинге. К недостаткам же относится затратность перестроения октодерева при деформации объекта.

На Рис. 6 приведен пример построения воксельной модели снеговика при использовании технологии октодерева (изображение взято из https://ru.wikipedia.org/wiki/Воксел). Как можно видеть, внутри есть объединенные ячейки разного размера.

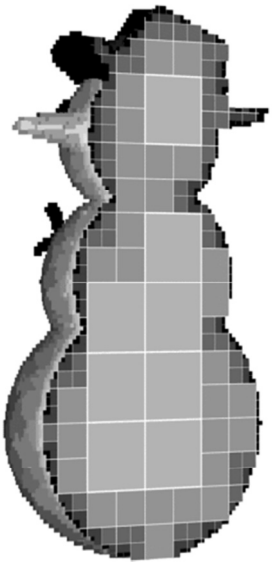


Рис. 6. Воксельная модель снеговика

ФРАКТАЛЬНАЯ ГРАФИКА

Фрактал – объект, в точности или приближенно совпадающий с частью себя самого. Поскольку более детальное описание элементов меньшего масштаба происходит по простому алгоритму, для описания подобного объекта достаточно нескольких математических уравнений. Так что для хранения фрактала достаточно запомнить формулу и соответствующие коэффициенты, изменяя которые можно получить большое разнообразие получаемых изображений.

В математике под фракталами понимают множества точек в евклидовом пространстве, имеющие дробную метрическую размерность (в смысле Минковского или Хаусдорфа), либо метрическую размерность, отличную от топологической. Первые примеры подобных множеств с необычными свойствами появились еще в XIX веке в результате изучения непрерывных недифференцируемых функций, а сам же термин был введен Бенуа Мандельбротом в 1975 г.

Оказалось, что в природе фракталы широко распространены: фрактальную структуру имеют такие объекты природы, как деревья, горные хребты, кристаллы и т.п. Классическим примером фракталов являются снежинки, самые первые фотографии которых были получены Уилсоном Бентли (https://ru.wikipedia.org/wiki/Бентли,_Уилсон) в конце XIX века (Рис. 7).

Рис. 7. Фотографии снежинок (Уилсон Бентли)

Помимо необходимости моделировать объекты природы, способность фрактальной графики получать изображения вычислительным путем часто используется для автоматической генерации необычных иллюстраций.

Обобщенную бесконечно рекурсивную процедуру для генерации фрактальных кривых на плоскости можно описать следующим образом: каждая часть заданной ломаной заменяется ломаной, подобной исходной, после чего процедура повторяется для вновь полученной ломаной.

Классическим примером фрактала является *кривая Коха*. Процесс ее построения выглядит следующим образом: отрезок разбивается на три равные части, средняя часть заменяется парой ребер равностороннего треугольника, и этот процесс повторяется для каждого из полученных отрезков (Рис. 8). Получившаяся кривая имеет бесконечную длину, нигде не дифференцируема и не спрямляема, не имеет самопересечений. Помимо этого, кривая имеет промежуточную хаусдорфову размерность, равную $\ln(4) / \ln(3) \approx 1.26$.

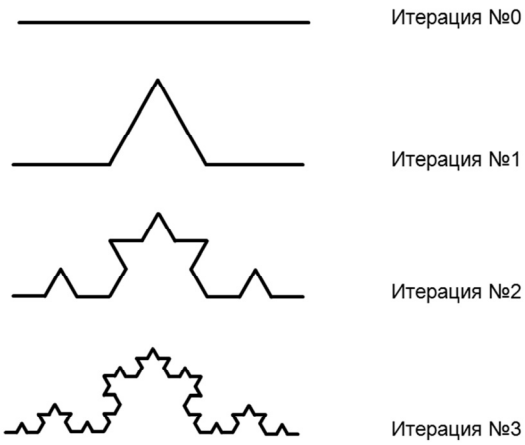


Рис. 8. Кривая Коха

Три кривых Коха, построенных в виде правильного треугольника, образуют замкнутую кривую, называемую *снежинкой Коха* (Рис. 9).

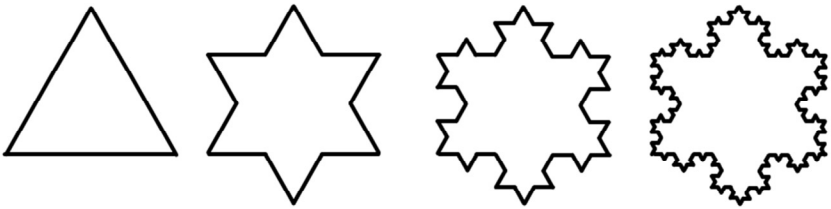


Рис. 9. Снежинка Коха

У кривой Коха есть множество модификаций. Например, если отрезки разбивать не на три части, а на четыпе, тогда можно получить *кривую Минковского* (Рис. 10) с размерность Хаусдорфа, равной $\ln(8)/\ln(4) = 1.5$.

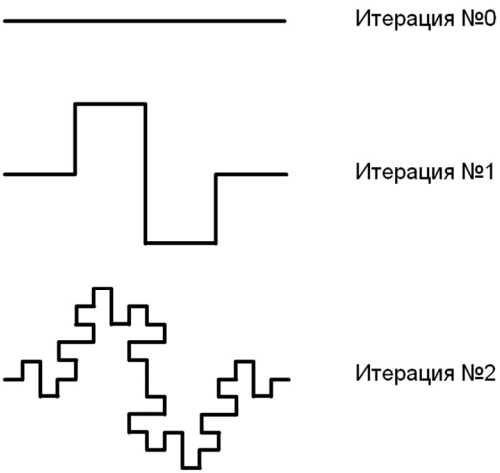


Рис. 10. Кривая Минковского

Если же вместо отрезка взять равносторонний треугольник и начать его рекурсивно дробить на четыре одинаковых равносторонних треугольника с отбрасыванием центрального, тогда можно получить *треугольник* (*салфетку*) *Серпинского* (Рис. 11).

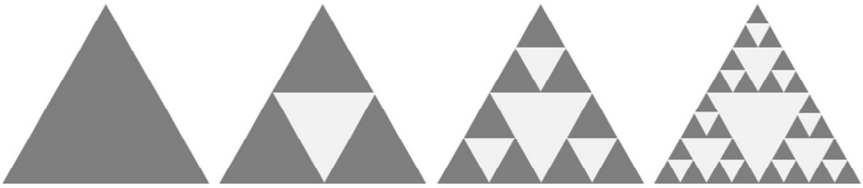


Рис. 11. Треугольник Серпинского

Еще одним известным (в т.ч. благодаря своим эффектным цветным визуализациям) фракталом является *множество Мандельброта* (Рис. 12) – множество таких точек c на комплексной плоскости, для которых рекуррентное соотношение $z_k = z_{k-1}^2 + c$ при $z_0 = 0$ задает ограниченную последовательность. Хотя фрагменты такого множества и не строго подобны исходному, но при многократном увеличении определенные части становятся все больше похожи друг на друга.

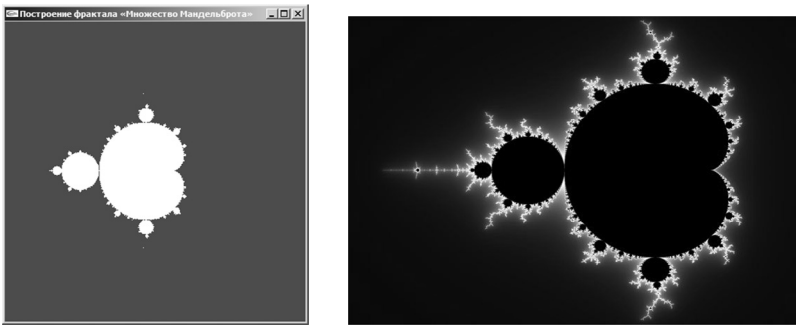


Рис. 12. Примеры множества Мандельброта

МОДЕЛИ ПРЕДСТАВЛЕНИЯ ЦВЕТА

С физической точки зрения, цвет – это набор определенных длин волн, отраженных от предметов (сами предметы не имеют цвета, мы всего лишь воспринимаем их такими). Например, белый лист бумаги выглядит белым именно потому, что отражает все видимые цвета, ничего не поглощая, в красном же свете он будет выглядеть красным. При белом освещении лист красной бумаги будет выглядеть красным (поглощаются все цвета, кроме красного), а при синем освещении – уже черным (так как синий свет красная бумага не отражает).

Обычно цвета в природе являются результатом смешивания базовых цветов. Поэтому для описания цвета вводится понятие *цветовой модели* как способа представления доступного количества цветов (*цветового пространства*) посредством их разложения на простые составляющие – *первичные цвета*, количество которых называют также *числом каналов* данной модели.

Соответственно, все остальные цвета из цветового пространства являются *составными*. В частности, при попарном смешивании первичных цветов образуются цвета второго порядка – *вторичные цвета*.

Ахроматические цвета – это оттенки серого (для черно-белого диапазона). Соответственно, *дополнительными цветами* называются пары цветов, которые при смешивании дают ахроматический, например, нейтральный серый получается при смешивании дополнительных цветов в равных пропорциях. Такие цвета используется, в частности, при настройке отображения цвета мониторами.

В цветовой модели (пространстве) каждому цвету можно поставить в соответствие строго определенную точку. Таким образом, цветовая модель – это простое геометрическое представление, основанное на системе координатных осей и принятого масштаба. В компьютерной графике используются различные формы представления (кодирования) цвета, примеры приведены в табл.3.

Таблица 3

Коды цветов в различных моделях

Цвет	#HEX	Модель RGB			Модель CMYK				Модель HSV			Модель LAB		
		R	G	B	C	M	Y	K	H	S	V	L	A	B
red	#FF0000	255	0	0	0	100	100	0	0	100	100	53	81	66
green	#00FF00	0	255	0	100	0	100	0	120	100	100	88	-86	83
blue	#0000FF	0	0	255	100	100	0	0	240	100	100	32	81	-109
cyan	#00FFFF	0	255	255	100	0	0	0	180	100	100	91	-48	-14
magenta	#FF00FF	255	0	255	0	100	0	0	300	100	100	60	99	-62
yellow	#FFFF00	255	255	0	0	0	100	0	60	100	100	97	-22	94
orange	#FFA500	255	165	0	0	35	100	0	39	100	100	72	25	80
khaki	#AA8530	170	133	48	30	45	80	5	42	72	67	58	1	52
violet	#8B00FF	139	0	255	45	100	0	0	273	100	100	43	83	-90
white	#FFFFFF	255	255	255	0	0	0	0	0	0	100	100	0	0
black	#000000	0	0	0	0	0	0	100	0	0	0	0	0	0
silver	#C0C0C0	192	192	192	0	0	0	25	0	0	75	78	-1	1

ЦВЕТОВОЙ КРУГ

Все многообразие доступных цветов часто представляют в виде *цветового круга* – способа представления цветов в условной форме для рассматриваемой цветовой модели. Секторы круга представляют определяемые цвета (как правило, с максимальной цветовой насыщенностью), размещенные в порядке условно близком к расположению в спектре видимого света. Противоположные цвета на этом круге обычно являются дополнительными, но при этом сами ахроматические цвета (особенно черный) часто в круге даже не представлены (для этого используют серую шкалу).

Цветовой круг в ряде эстетических концепциях делят на теплую и холодную части. К теплым цветам обычно относят цвета из красно-желтого диапазона, а к холодным – из сине-зеленого.

Первые цветовые круги появились с развитием теории цвета в трудах ученых и мистиков (например, Гете, Иттена, Ньютона) и использовались, в частности, в живописи для объяснения взаимодействия красителей.

Так по теории Гете цвета происходили от борьбы света и тьмы и сначала из этого столкновения появлялись цвета первого порядка: красный (**R**ed), желтый (**Y**ellow) и синий (**B**lue). Вторичными цветами здесь являются фиолетовый, зеленый и оранжевый. Эта цветовая модель **RYB** (Рис. 13) до сих пор используется художниками и дизайнерами при предварительном подборе цветовых сочетаний. В этой модели существует несколько основных схем сочетания цветов:

1) *Комплементарная* (энергичное сочетание цветов, которые расположены на противоположных сторонах);

2) *Триадная* (гармоничное сочетание трех-четырех цветов, лежащих на одинаковом расстоянии друг от друга);

3) *Аналогичная* (спокойное сочетание двух-пяти цветов, расположенных подряд).

В компьютерной же графике наиболее распространен восьмисекторный цветовой круг модели RGB (Рис. 14). Здесь к базовым цветам (красный, зеленый и синий) добавляют еще четыре "промежуточных" цвета – оранжевый, голубой, фиолетовый и пурпурный.

Рис. 13. Цветовой круг Иттена (модель RYB)

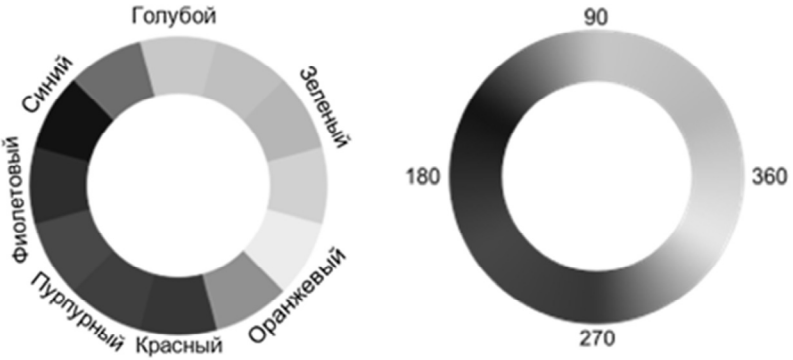


Рис. 14. Восьмисекторный цветовой круг модели RGB

АДДИТИВНАЯ МОДЕЛЬ

В аддитивной модели результирующий цвет получается путем *суммирования* лучей света разных цветов. В этой системе отсутствие всех цветов дает черный цвет, а присутствие всех – белый. Система аддитивных цветов работает с излучаемым светом, например от экрана телевизора или компьютера, т.е. является аппаратно-зависимой.

Самой распространенной аддитивной моделью является модель **RGB**. В этой трехканальной модели первичными цветами являются **R**ed (красный), **G**reen (зеленый) и **B**lue (синий).

Вторичными цветами в этой модели являются:

- «Red + Green» = «Yellow» (желтый),
- «Green + Blue» = «Cyan» (голубой, сине-зеленый),
- «Blue + Red» = «Magenta» (пурпурный, маджента).

Дополнительными цветами здесь являются:

- «Cyan + Red»,
- «Magenta + Green»,
- «Yellow + Blue».

При добавлении в модель *альфа-канала* (alpha channel), отвечающего за прозрачность, получается модель **RGBA**.

Популярность модели RGB привела к разработке единого стандарта sRGB, использующего гамма-коррекцию для устранения искажения яркости черно-белого. Применение данного стандарта означает, что любой откалиброванный в этой системе монитор не будет искажать цветовой баланс исходного изображения.

Обычно в графических приложениях под RGB-цвет выделяется по 4 байта: первые три используются под каналы красного, зеленого и синего, а четвертый либо не используется, либо отводится под прозрачность. Таким образом, размерность цветового пространства составляет более 16 миллионов цветов.

В этом случае, значения каналов кодируются в диапазоне [0, 256] либо масштабируется в интервал [0,1]. В web-дизайне значения каналов часто кодируются в шестнадцатеричной системе исчисления, в диапазоне [00,FF].

Таким образом, например, лазурный (cerulean) цвет может быть закодирован разными способами:

- (0, 123, 167),
- (0, 0.48, 0.65),
- #007BA7,
- **R**000**G**123**B**167.

Для задания цвета в WINDOWS предусмотрен стандартный четырехбайтовый тип `COLORREF`. Определить переменную данного типа можно с помощью встроенного макроса `RGB`:

```
COLORREF c1 = RGB( r, g, b ),
```

где `r`, `g` и `b` – значения интенсивностей цветовых каналов, меняющиеся в диапазоне от 0 до 255.

Но задать цвет можно и напрямую через шестнадцатеричное представление чисел:

```
COLORREF c2 = 0x00bbggrr,
```

где `rr`, `gg`, `bb` – значения интенсивностей цветовых каналов, меняющиеся в диапазоне от 0 до 0xFF.

На Рис. 15 приведены два классических варианта представления модели RGB – с помощью кругов и единичного куба.

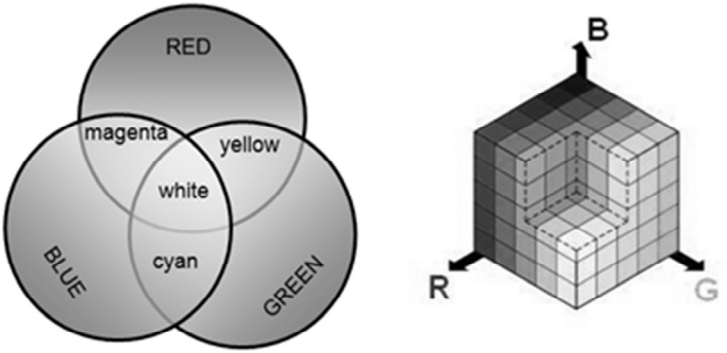


Рис. 15. Цветовая модель RGB

СУБТРАКТИВНАЯ МОДЕЛЬ

В субтрактивной модели происходит обратный процесс: результирующий цвет получается путем *вычитания* других цвета из общего луча света. В такой системе белый цвет соответствует отсутствию всех цветов, тогда как наличие их всех дает черный цвет. Система субтрактивных цветов работает с отраженным светом, например от листа бумаги.

Из субстрактивных моделей в основном используется трехканальная модель CMY и ее "уточнение" – четырехканальная модель CMYK.

В модели **CMY** основными цветами являются голубой (**C**yan), пурпурный (**M**agenta) и желтый (**Y**ellow), а вторичными, соответственно, – красный, зеленый и синий.

На практике (в силу неидеальности красителей и погрешностей при их смешивании) наложение всех трех основных цветов при печати на белой бумаге не позволяет получать качественный черный цвет. Для решения данной проблемы используется четвертая краска – черная (blac**K**), а модель преобразуется в **CMYK**. На Рис. 16 приведено графическое представление моделей CMY и CMYK.

В этой модели тот же лазурный цвет может быть закодирован как (100, 26, 0, 35) или **C**100**M**26**Y**00**K**35.

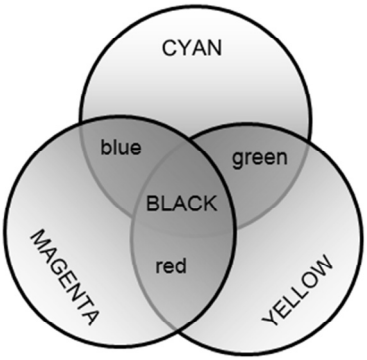


Рис. 16. Цветовая модель CMY (CMYK)

ПЕРЦЕПЦИОННАЯ МОДЕЛЬ

В отличие от моделей RGB и CMYK, более интуитивным способом описания цвета является представление его в виде цветового тона, насыщенности и освещенности (яркости или светлоты). Обычно значения этих параметров задается в диапазоне [0,100] или нормируется в интервал [0,1].

Цветовой тон (***H**ue*) ассоциируется в человеческом сознании с определенным цветом (типом красителя – красный, желтый и т.п.). В связи с тем, что все цвета расположены по кругу, то удобно диапазон изменений указывать в градусах [0-360°].

Насыщенность (***S**aturation*) характеризует степень (уровень) выраженности цветового тона (концентрации красителя), т.е. *чистоту цвета* (отсутствие примесей). Соответственно, чем больше этот параметр, тем более блеклым получается цвет.

Яркость (***B**rightness*) или *значение/сила* (***V**alue*) цвета показывает величину черного оттенка, добавленного к цвету, что делает его более темным.

Светлота (***L**ightness*) является субъективной яркостью участка изображения по отношению к яркости той части изображения, которая воспринимается человеком как белая.

На основании этих параметров разработаны цветовые модели **HSB** (**HSV**) и **HSL**. Хотя эти системы и больше соответствует природе цвета, но при выводе изображения на экран или принтер их все равно приходится преобразовывать в системы RGB и CMYK соответственно.

На Рис. 17 приведено классическое представление модели HSV в виде конуса. Здесь все одинаково насыщенные цвета (например, желтый и красный) располагаются на концентрических окружностях. При этом, чем ближе к центру круга, тем все более блеклыми получаются цвета (в самом центре любой цвет становится белым), а чем ниже – тем чернее (вплоть до черного). Работу с насыщенностью можно характеризовать как добавление доли белой краски, а с яркостью – как добавление черной.

Лазурный цвет в модели HSB может быть закодирован как (196, 100, 65) или **H**196**S**100**B**65, а в модели HSL – (196, 100, 33) или **H**196**S**100**L**33.

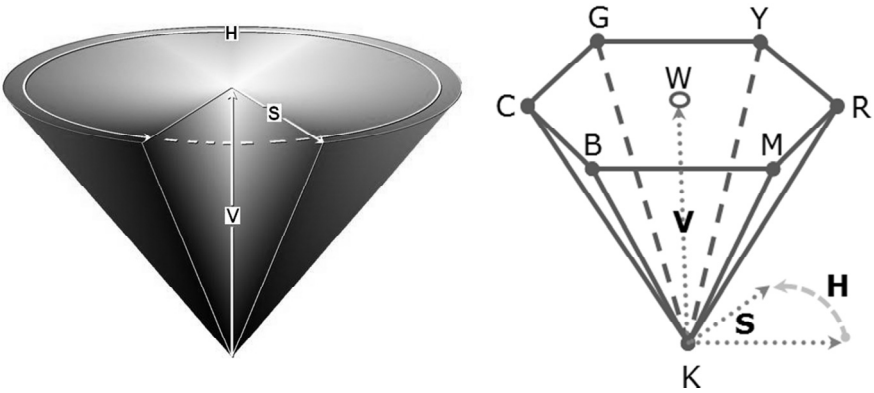


Рис. 17. Цветовая модель HSB (HSV)

Все рассмотренные модели являются *аппаратно-зависимыми* (в разных ситуациях одно и то же изображение выглядит неодинаково), поскольку ограничиваются возможностями аппаратного обеспечения (монитором, типографскими красками). Другая существенная проблема этих моделей – это *нелинейность* изменения цвета с точки зрения человеческого восприятия (одинаковое изменение значений координат цвета в разных областях цветового пространства не производит одинаковое ощущение изменения цвета).

Для преодоления указанных недостатков Международной комиссией по освещению (CIE) была разработана аппаратно-независимая модель **LAB**: канал **L** здесь все также отвечает за яркость, а за хроматические цвета отвечают каналы **A** (от зеленого до красного) и **B** (от синего до желтого). Лазурный цвет в данной модели может быть закодирован как (48, -13, -31). В настоящее время эта модель используется в качестве промежуточного цветового пространства для конвертации данных между другими цветовыми пространствами (например, из RGB-сканера в CMYK-принтера).

ПРОГРАММА PAINT

Существует множество программ для редактирования растровой графики, но самой простой и распространенной из них является Microsoft Paint – простой и удобный растровый графический редактор компании Microsoft, входящий в состав всех операционных систем Windows.

Помимо стандартных операций (выбор цвета, заливка/закраска, изменение размера, поворот/масштабирование) в программу встроен также набор кистей (brush) и графических примитивов (линии, многоугольники, овал). Кроме того, в программе предусмотрена возможность вставки текста.

Программа позволяет оперировать с графическими файлами форматов PNG, BMP, JPG, GIF и TIFF.

На Рис. 18 приведен скриншот программы в момент выбора цвета. Как можно видеть, программа позволяет задавать цвета непосредственно в моделях RGB и HSB в графическом и цифровом форматах.

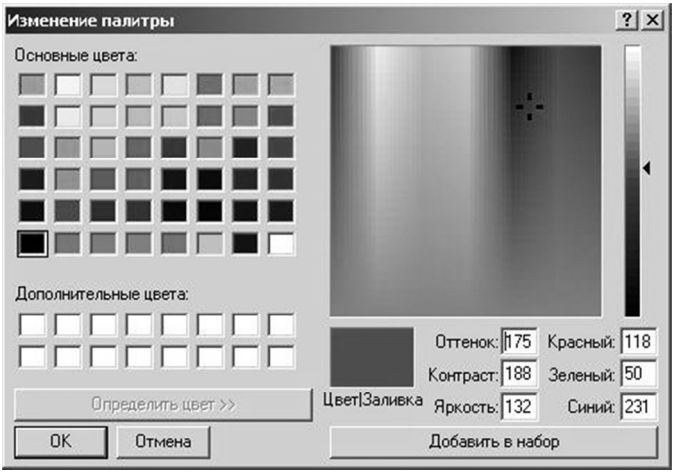


Рис. 18. Выбор цвета в программе Paint

БИБЛИОТЕКА STL

Библиотека стандартных шаблонов *STL* (Standard Template Library) – это набор согласованных обобщенных алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.

Библиотека предоставляет различные типобезопасные контейнеры для хранения коллекций связанных объектов одного типа. Контейнеры представляют собою шаблоны классов, которые содержат функции-члены для добавления/удаления элементов, доступа к элементу и т.п. Элементы библиотеки расположены в пространстве имен `std`.

Шаблон `vector` расположен в заголовочном файле `<vector>`. Контейнер расширяет возможности стандартного массива C++: позволяет динамически менять размер, вставлять и удалять элементы, контролировать выход за пределы массива и многое другое. Данные особенности очень полезны при работе со структурами данных, используемых в компьютерной графике. Например, можно создать контейнер фигур, каждая из которых содержит свой массив вершин и цветов, при этом появляется легкая возможность вставлять в произвольное место новые элементы, впрочем, как и изымать их из контейнера. Однако, когда операции вставки и удаления выполняются часто, тогда гораздо выгоднее использовать такие контейнеры, как *`set`* и *`deque`*.

Подробное описание библиотеки STL можно получить в MSDN или в Википедии:

https://msdn.microsoft.com/ru-ru/library/ct1as7hw.aspx
https://ru.wikipedia.org/wiki/Стандартная_библиотека_шаблонов

Описание контейнера `vector` вместе с примерами можно получить также в MSDN или на Википедии:

https://msdn.microsoft.com/ru-ru/library/9xd04bzs.aspx
https://ru.wikipedia.org/wiki/Vector_(C%2B%2B)

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Определите, к каким направлениям компьютерной графики относится обработка аэрокосмических снимков и чертежей, моделирование "машинного зрения" роботов.

2. Определите возможные геометрические формы субпикселей для ЖК-дисплея.

3. Определите количество доступных цветов для следующих глубин цвета: 2, 3, 4, 8, 16, 24, 32 и 64 бита.

4. Определите, в чем удобство того, что под цвет отводят по 4 байта, даже если из них реально используется только 3?

5. Определите тип базисных функций, использующихся для интерполяции при масштабировании растрового изображения.

6. Определите количество каналов и цветов, доступных в стандартной модели RGBA.

7. Определите значения каналов в моделях RGB, CMYK и HSV для следующих цветов: янтарный (amber, #FFBF00), кремовый (cream #FFFDD0), аквамариновый (aquamarine, #7FFFD4), золотой (gold, #FFD700), лайм (lime, #CCFF00), персиковый (peach, #FFE5B4).

8. Объясните, почему в современных телевизорах вводят дополнительные пиксели (например, желтого и белого цветов)?

9. Найдите длину периметра снежинки Коха.

10. Определите тип треугольника, появляющегося в результате дробления салфетки Серпинского.

ВВЕДЕНИЕ В БИБЛИОТЕКУ OpenGL

OpenGL (Open Graphics Library) является спецификацией, определяющей *API* (Application Programming Interface) для написания приложений в области двумерной и трехмерной графики. С точки зрения разработчика графического приложения OpenGL является прикладной программной библиотекой.

На основании этой спецификации производители создают уже свои *OpenGL-реализации* (библиотеки функций, соответствующих спецификации), призванные эффективно использовать возможности данного оборудования (в частности, если устройство не поддерживает какую-либо функцию, то эта функция обязана выполняться библиотекой программно). Как следствие, разработчикам графических приложений достаточно научиться грамотно использовать функции из спецификации, оставив их эффективную реализацию разработчикам конкретного аппаратного обеспечения. В результате, написанные с помощью OpenGL программы можно переносить практически на любые платформы, получая при этом практически одинаковый результат.

В частности, разработаны реализации OpenGL не только для таких популярных операционных систем, как Windows и Unix-подобных систем, но и для PlayStation 3 и Mac OS. На платформе Windows библиотека конкурирует с графической библиотекой Direct3D из пакета DirectX, разрабатываемого компанией Microsoft.

В связи с независимостью от языка программирования были разработаны различные варианты привязки (binding) функций OpenGL для различных языков: Java, Фортран 90, Python, Visual Basic, Pascal и т. д.

НАПРАВЛЕНИЯ РАЗВИТИЯ

Одной отличительной возможностью OpenGL является поддержка расширений. При появлении новой технологии разработчик может внедрить ее в драйвер видеокарты, не дожидаясь выхода новой версии OpenGL. Воспользоваться новыми функциями можно, если специальным образом запросить данное расширение (предварительно, конечно, проверив его поддержку видеокартой). Проверенные временем расширения включаются в следующую версию OpenGL консорциумом (наблюдательным советом) *ARB* (Architecture Review Board), в который входят представители таких компаний, как ATI, NVIDIA, Intel, IBM, Apple и многие другие.

Поначалу консорциум возглавляла компания Silicon Graphics (SGI), но после 2006 г. контроль за разработкой спецификаций OpenGL был передан промышленному консорциуму Khronos Group. Целью данного консорциума является выработка открытых стандартов интерфейсов программирования в области создания и воспроизведения динамической графики и звука на широком спектре платформ и устройств, с поддержкой аппаратного ускорения.

Первая спецификация была принята в 1992 г., а на следующий год появились и первые реализации. В 2003 г. вышла версия 1.4. В версии 2.0 (2004 г.) в библиотеку была добавлена поддержка языка шейдеров GLSL (OpenGL Shading Language). В 2009 г. с версии 3.1 часть компонент была объявлена устаревшей, так что доступ к ним стал возможен только через расширение `GL_ARB_compatibility`. Последняя (на сегодняшний момент) версия 4.6 вышла 31 июля 2017 г.

Одним из направлений дальнейшего развития является Vulkan («новое поколение OpenGL», «glNext»), цель которого – превзойти другие API (включая своего предшественника OpenGL) в части снижения накладных расходов, повышения степени прямого контроля над GPU и уменьшения нагрузки на CPU. Первая публичная версия вышла 6 февраля 2016 года вместе с экспериментальными драйверами для видеокарт AMD и NVIDIA.

Другим важным ответвлением является API OpenGL ES (OpenGL for Embedded Systems) – подмножество графического интерфейса OpenGL, разработанное специально для встраиваемых систем (мобильных телефонов, карманных компьютеров, игровых консолей). Первая версия вышла в 2003 и была основана на спецификации OpenGL 1.3. В августе 2012 вышла версия 3.0, которая базируется на OpenGL версий 3.3 и 4.2 и поддерживается в Android версии 4.3 и выше. А спустя три года вышла последняя на сегодняшний момент версия 3.2.

На основе OpenGL ES была разработана библиотека WebGL (Web-based Graphics Library). Эта библиотека позволяет создавать на языке программирования JavaScript интерактивную 3D-графику для современных версий веб-браузеров (таких, как Google Chrome 9.0 и выше). При этом, за счет использования низкоуровневых средств поддержки OpenGL, часть кода на WebGL может выполняться непосредственно на видеокартах. Фактически же, WebGL является контекстом элемента Canvas HTML, который обеспечивает API 3D графики без использования плагинов. Спецификация 1.0 была выпущена в 2011 на основе OpenGL ES 2.0. Стандарт WebGL 2.0, базирующийся на OpenGL ES 3.0, был принят в январе 2017 и реализован в браузерах Firefox 51, Chrome 56 и Opera 43.

Помимо этого, существуют также и такие открытые («неофициальные») реализации спецификации OpenGL, как библиотека Mesa 3D (http://mesa3d.org). Данная библиотека совместима с OpenGL на уровне кода и поддерживает не только программную эмуляцию, но и аппаратное ускорение графики (при наличии соответствующих драйверов).

ОПИСАНИЕ БИБЛИОТЕКИ

Библиотека OpenGL была разработана как обобщенный аппаратно-независимый интерфейс к графической аппаратуре. По этой причине библиотека не включает в себя функции для работы с окнами и устройствами ввода, средства задания высокоуровневых описаний сложных моделей (чайник, сфера, тор, диск и т.п.) – для этих целей используются такие надстройки над OpenGL, как GLU (OpenGL Utility Library), GLUT (OpenGL Utility Toolkit) и ряд других.

На официальном сайте OpenGL (https://www.opengl.org) можно найти различную документацию к библиотеке и скачать кроссплатформенную библиотеку GLUT для создания простого графического приложения, использующего OpenGL.

Работа с библиотекой построена по принципу «клиент-сервер»: приложение вызывает команды, а библиотека занимается их обработкой. В общем случае, OpenGL можно сравнить с конечным автоматом, состояние которого определяется множеством специальных констант и меняется посредством соответствующих команд, в результате каждый передаваемый объект проходит обработку в соответствии с этим текущим состоянием. Как следствие, код программы графического приложения получается весьма простым, понятным, удобным.

Большинство реализаций библиотеки имеет определенную последовательность стадий обработки, которая называется *конвейером визуализации* OpenGL (OpenGL rendering pipeline) и схематично представлена на Рис. 19.

На нулевом этапе работы этого конвейера необходимо аппроксимировать исходные объекты примитивами, доступными библиотеке.

Затем полученный массив данных подается на вход обработчику. На этом этапе из введенных вершин собираются и преобразовываются примитивы (например, отбрасываются не попадающие в поле зрения после поворота), вычисляются нормали (для учета освещения), текстурные координаты, проекции и т. п.

После чего изображение растеризуется (с учетом текстур) в буфер кадра, разбиваясь на *фрагменты*, которые и становятся пикселями, если пройдут ряд тестов (например, тесты на глубину и трафарет).

На предпоследнем этапе возможны дополнительные операции над пикселями, такие как "затуманивание", копирование из одной части буфера в другую или сохранение их в текстуру.

И, наконец, на последнем этапе содержимое буфера кадра отправляется для вывода на экран.

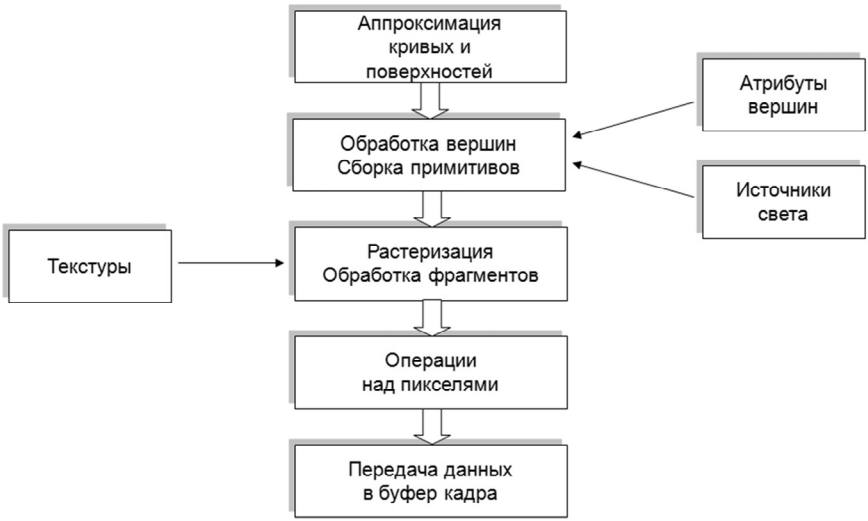


Рис. 19. Схематичное представление конвейера визуализации

КОМАНДЫ БИБЛИОТЕКИ

Все команды (процедуры и функции) OpenGL можно разбить на несколько групп:

- работа с примитивами,
- задание (настройка) текстур и освещения,
- геометрические (матричные) преобразования,
- работа с буферами и пикселями,
- работа с режимами состояния библиотеки.

В частности, к группе команд работы с примитивами относятся такие команды, как задание атрибутов вершин и установка режимов отрисовки примитива.

В данном учебном пособии будут рассмотрены только те команды, которые часто используются при работе с двумерной графикой, в частности, не будет затронута работа с освещением. Также не будут рассмотрена и работа с шейдерами.

СТРУКТУРА КОМАНД И ТИПЫ ДАННЫХ

Все команды библиотеки OpenGL начинаются с префикса «`gl`», а все константы – с префикса «`GL_`». Аналогично для библиотеки GLU соответствующие префиксы команд и констант обозначаются как «`glu`» и «`GLU_`», для библиотеки GLUT – «`glut`» и «`GLUT_`», для библиотеки `GLAUX` – «`aux`» и «`AUX_`» и т. д.

Кроме того, в имена команд входят суффиксы, несущие информацию о числе (1,2,3 или 4) и типе передаваемых параметров (аргументов). Разбор структуры имени команды для задания белого цвета `glColor3ub(255,255,255)` приведен в табл.4.

В табл.5 приводится ряд вводимых библиотекой типов данных, аналогичные им стандартные типы языка C++ и соответствующие суффиксы. В частности, суффикс «`-v`» указывает на то, что список аргументов представляет собой массив элементов заданного типа и передается одним параметром (указателем).

Таблица 4

Команда glColor3ub (255, 0, 255)

Часть имени	Описание
gl	Имя библиотеки, где описана функция (OpenGL)
Color	Имя самой функции задания цвета
3	Количество передаваемых аргументов
ub	Тип передаваемых аргументов
255,255,255	Список аргументов функции (RGB-компоненты)

Таблица 5

Основные типы данных

Суффикс	Описание типа		Тип в OpenGL	Тип в C++
b	целый	1 байт	GLbyte	signed char
ub			GLubyte	unsigned char
s		2 байта	Glshort	signed short
us			Glushort	unsigned short
i		4 байта	Glint	signed int
ui			Gluint	unsigned int
f	вещественный	4 байта	GLfloat	float
d		8 байт	GLdouble	double
	перечислимый		GLenum	unsigned int
	бинарный		GLboolean	unsigned char

ИНФОРМАЦИЯ О ТЕКУЩЕМ СОСТОЯНИИ

Получить значения параметров состояния библиотеки (как конечного автомата) можно с помощью команды `glGet`:

```
void glGetBooleanv ( GLenum, GLboolean* )

void glGetDoublev ( GLenum, GLdouble* )

void glGetFloatv ( GLenum, GLfloat* )

void glGetIntegerv ( GLenum, GLint* )

GLubyte* glGetString (GLenum )

GLenum glGetError ( void )
```

Например, для получения значения текущего цвета можно вызвать команду `glGetFloatv(GL_CURRENT_COLOR, rgba)`, где второй параметр – это массив из четырех элементов (режим RGBA).

Команда `glGetString(GL_EXTENSIONS)` возвращает текстовую строку (`GLubyte*`) с перечислением (через пробел) доступных расширений, полный список которых можно найти на сайте разработчика (https://www.opengl.org/registry/). Запуск этой же команды с параметром `GL_VERSION` возвращает строку с номерами версий библиотеки и видеодрайвера, `GL_VENDOR` – с информацией о производителе видеокарты, а `GL_RENDERER` - с информацией о самой видеокарте.

Помимо этого, можно получить и статус ошибки для ряда операций функцией `glGetError()`, которая возвращает значение `GL_NO_ERROR` при отсутствии зарегистрированных ошибок. На проблемы со стеком указывают значения `GL_STACK_OVERFLOW` и `GL_STACK_UNDERFLOW`, на выполнение недопустимой операции – `GL_INVALID_OPERATION`, на нехватку памяти – `GL_OUT_OF_MEMORY` и т.д.

ИЗМЕНЕНИЕ РЕЖИМОВ РАБОТЫ

В целях оптимизации такие затратные режимы работы библиотеки, как, например, наложение текстур или учет освещения, по умолчанию выключены.

Включение и отключение различных режимов библиотеки OpenGL осуществляется командами `glEnable(*) и glDisable(*)`, где в качестве параметра используется константа-идентификатор режима:

```
void glEnable (GLenum)
void glDisable (GLenum)
```

Например, выполнение команды `glEnable(GL_POINT_SMOOTH)` включает режим сглаживания вершин, который продолжается до тех пор, пока не будет вызвана команда `glDisable(GL_POINT_SMOOTH)`. Выбор качества скругления точки регулируется командой `glHint(GL_POINT_SMOOTH_HINT, GL_NICEST)`, которая может регулировать выполнение и некоторых других команд.

В случае неправильного указания константы-идентификатора будет указан код ошибки `GL_INVALID_ENUM`, а при вызове внутри операторных скобок – `GL_INVALID_OPERATION`. Код ошибки можно проверить командой `glGetError`.

Проверка режима включенности осуществляется командой `glIsEnabled`:

```
GLboolean glIsEnabled(GLenum)
```

Например, можно включить режимы прозрачности (`GL_ALPHA_TEST`), смешивание цветов (`GL_COLOR_LOGIC_OP и GL_BLEND`), сглаживания примитивов (`GL_POLYGON_SMOOTH, GL_LINE_SMOOTH, GL_POINT_SMOOTH`) и т.д.

Часть из этих режимов будет рассмотрена далее.

Более подробный список режимов можно получить, например, по адресу:

msdn.microsoft.com/en-us/library/windows/desktop/dd318845(v=vs.85).aspx

КОМАНДЫ ЗАДАНИЯ ЦВЕТА

Для задания цвета используется команда `glColor.` В табл.6 приведены варианты этой команды для различных типов данных (с указанием возможных значений для каждого из них) и режимов RGB/RGBA. Чаще всего, конечно, используются только четыре варианта:

```
void glColor3ub  (GLubyte R, GLubyte G, GLubyte B)
void glColor3ubv (GLubyte *RGB)
void glColor3f  (GLfloat R, GLfloat G, GLfloat B)
void glColor3fv (GLfloat *RGB)
```

Таблица 6

Варианты команды задания цвета

Тип данных	Режимы		Диапазон изменений
	RGB	RGBA	
Glbyte	glColor3b glColor3bv	glColor4b glColor4bv	[-128,127]
Glubyte	glColor3ub glColor3ubv	glColor4ub glColor4ubv	[0,255]
Glshort	glColor3s glColor3sv	glColor4s glColor4sv	$[-2^{15}, 2^{15}-1]$
Glushort	glColor3us glColor3usv	glColor4us glColor4usv	$[0, 2^{16}-1]$
Glint	glColor3i glColor3iv	glColor4i glColor4iv	$[-2^{31}, 2^{31}-1]$
Gluint	glColor3ui glColor3uiv	glColor4ui glColor4uiv	$[0, 2^{32}-1]$
Glfloat	glColor3f glColor3fv	glColor4f glColor4fv	[0,1]
Gldouble	glColor3d glColor3dv	glColor4d glColor4dv	[0,1]

КОМАНДЫ ЗАДАНИЯ ТОЧЕК

Ключевой (атомарный) объект – это *вершина* (точка, узел), все примитивы задаются путем перечисления своих вершин. К *атрибутам вершины* можно отнести геометрические и текстурные координаты вершин, цвет, нормаль.

В соответствии с различными типами данных и размерностью пространства, у команды задания вершины `glVertex` есть целых 24 варианта (табл. 7). При этом, вне зависимости от того, задаются вершины в 2D или в 3D, они все равно неявно библиотекой будут преобразованы в 4D для дальнейших расчетов.

При задании двумерных объектов чаще всего используются следующие варианты команды `glVertex`:

```
void glVertex2i (GLint x, GLint y)
void glVertex2iv(GLint *xy)
void glVertex2f (GLfloat x, GLfloat y)
void glVertex2fv(GLfloat *xy)
```

Таблица 7

Варианты команды задания точки

Размерность	Тип данных			
	GLshort	*GLint*	*GLfloat*	*GLdouble*
2D	glVertex2s glVertex2sv	glVertex2i glVertex2iv	glVertex2f glVertex2fv	glVertex2d glVertex2dv
3D	glVertex3s glVertex3sv	glVertex3i glVertex3iv	glVertex3f glVertex3fv	glVertex3d glVertex3dv
4D	glVertex4s glVertex4sv	glVertex4i glVertex4iv	glVertex4f glVertex4fv	glVertex4d glVertex4dv

ГЕОМЕТРИЧЕСКИЕ ПРИМИТИВЫ

В OpenGL примитивами являются такие геометрические фигуры, как точки, линии, треугольники, четырехугольники и полигоны (многоугольники). Примитивы задаются путем указания его вершин командой `glVertex` внутри операторных скобок `glBegin/glEnd`. Тип примитива задается параметром процедуры `glBegin(*)` и может принимать значения, представленные в табл. 8 (соответствующие примеры с нумерацией вершин приведены на Рис. 20). В поздних версиях библиотеки список доступных примитивов изменился, на Рис. 21 приведен соответствующий скриншот с официального сайта.

Внутри операторных скобок можно указывать только команды задания геометрических и текстурных координат, цвета, нормалей. Остальные команды будут в большинстве случаев проигнорированы.

При задании примитива автоматически применяются текущие параметры состояния библиотеки. Например, после команды установки цвета все задаваемые вершины примитива автоматически принимают значение этого цвет до тех пор, пока новая команда не сменит текущий цвет.

В случае наложения нескольких примитивов друг на друга, в большинстве случаев отображается тот, кто был задан последним.

На Рис. 22 приведен пример задания черного треугольника. Команда `glClear(GL_COLOR_BUFFER_BIT)` очищает буфер цвета белым цветом, установленным командой `glClearColor(1,1,1,1)`. Следующей командой `glColor3ub(0,0,0)` задается черный цвет, которым и закрасится весь треугольник. Дальше внутри операторных скобок (с параметром `GL_TRIANGLES`) перечисляются 3 пары вершин командой `glVertex2i`. Процедура подготовки буфера кадра завершается командой `glFinish()`, которая и отправляет содержимое буфера на экран.

Таблица 8

Типы примитивов

Параметр	Описание
GL_POINTS	Каждая *вершина* определяет отдельную *точку*
GL_LINES	Каждая отдельная *пара вершин* определяет отдельный *отрезок*
GL_LINE_STRIP	Вершины определяют *незамкнутую ломаную*
GL_LINE_LOOP	Вершины определяют *замкнутую ломаную*
GL_TRIANGLES	Каждая *отдельная тройка вершин* определяет *треугольник*
GL_TRIANGLE_STRIP	Каждая *следующая вершина вместе с двумя предыдущими* определяет *треугольник*
GL_TRIANGLE_FAN	*Первая вершина и каждая следующая пара* определяет *треугольник*
GL_QUADS	Каждая *отдельная четверка вершин* определяет *четырехугольник*
GL_QUAD_STRIP	Каждая *следующая пара вершин* вместе с предыдущей парой определяет *четырехугольник*
GL_POLYGON	Вершины определяют выпуклый *многоугольник*

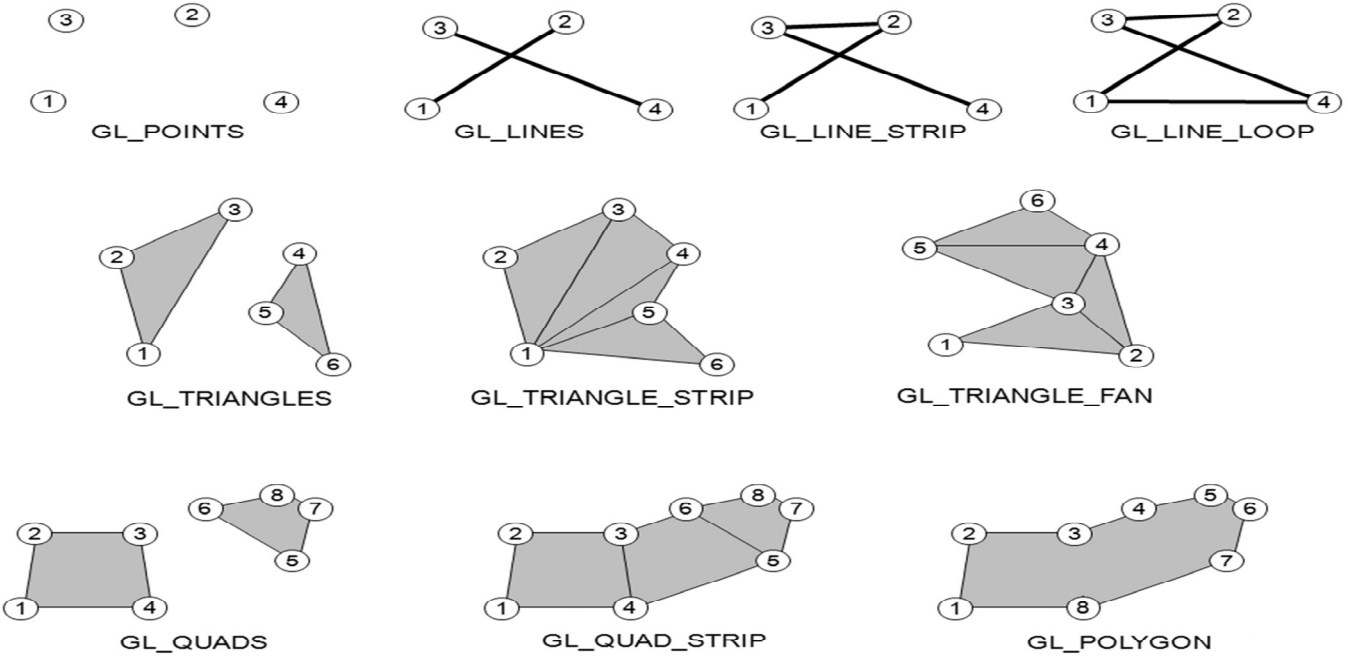


Рис. 20. Примитивы OpenGL 1.4

Primitive type	GL_FIRST_VERTEX_CONVENTION	GL_LAST_VERTEX_CONVENTION
GL_POINTS	i	i
GL_LINES	$2i - 1$	$2i$
GL_LINE_LOOP	i	$i + 1$, if $i <$ the number of vertices. 1 if i is equal to the number of vertices.
GL_LINE_STRIP	i	$i + 1$
GL_TRIANGLES	$3i - 2$	$3i$
GL_TRIANGLE_STRIP	i	$i + 2$
GL_TRIANGLE_FAN	$i + 1$	$i + 2$
GL_LINES_ADJACENCY	$4i - 2$	$4i - 1$
GL_LINE_STRIP_ADJACENCY	$i + 1$	$i + 2$
GL_TRIANGLES_ADJACENCY	$6i - 5$	$6i - 1$
GL_TRIANGLE_STRIP_ADJACENCY	$2i - 1$	$2i + 3$

Рис. 21. Примитивы OpenGL 4.6


```
glClearColor(1, 1, 1, 1);
glClear(GL_COLOR_BUFFER_BIT);

glColor3ub(0,0,0);
glBegin(GL_TRIANGLES);
  glVertex2i( 50, 50);
  glVertex2i(100,150);
  glVertex2i(150,100);
glEnd();

glFinish();
```

Рис. 22. Пример задания черного треугольника

При задании нескольких примитивов одного типа их вершины в целях оптимизации лучше перечислять в пределах одной пары `glBegin/glEnd`.

Пример кода для задания разноцветного двумерного треугольника с использованием различных типов данных:

```
GLubyte blue[3] = {0, 0, 255};
float coord[] = {2, 1};
glColor3f (1.0, 0., 0);

glBegin (GL_TRIANGLES);
  glVertex2f (0.0, 0.0);
  glColor3ub (0, 255, 0); glVertex2i (1, 2);
  glColor3ubv (blue);     glVertex2fv (coord);
glEnd();
```

ПРИМИТИВ «ТОЧКИ»

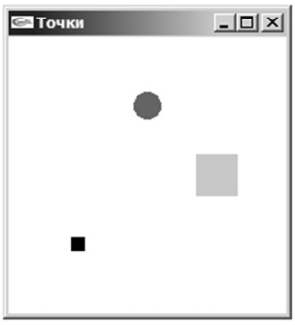
Точки задаются командой `glVertex2` внутри операторных скобок с параметром `GL_POINTS`, например, точки, хранящиеся в векторе *Points* пар (*x,y*), можно выдать так:

```
glBegin (GL_POINTS);
for (int i = 0; i < Points.size(); i++)
  glVertex2f ( Points[i].x, Points[i].y );
glEnd();
```

При задании вершин можно указать размер точки в пикселях командой `glPointSize`, которая должна быть указана до операторных скобок. Максимальный размер точки определяется реализацией OpenGL.

Также можно включить режим сглаживания (скругления) точки командой `glEnable(GL_POINT_SMOOTH)`. Соответственно, выключение данного режима выполняется "обратной" командой `glDisable(GL_POINT_SMOOTH)`.

Результат работы этих команд для точек разного цвета представлен на Рис. 23.




```
glColor3ub(100, 100, 100);
glPointSize(20);
glEnable(GL_POINT_SMOOTH);
glBegin(GL_POINTS); glVertex2i(100,150); glEnd();
glDisable(GL_POINT_SMOOTH);

glColor3ub(200, 200, 200);
glPointSize(30);
glBegin(GL_POINTS); glVertex2i(150,100); glEnd();

glColor3ub(0, 0, 0);
glPointSize(10);
glBegin(GL_POINTS); glVertex2i( 50 ,50); glEnd();
```

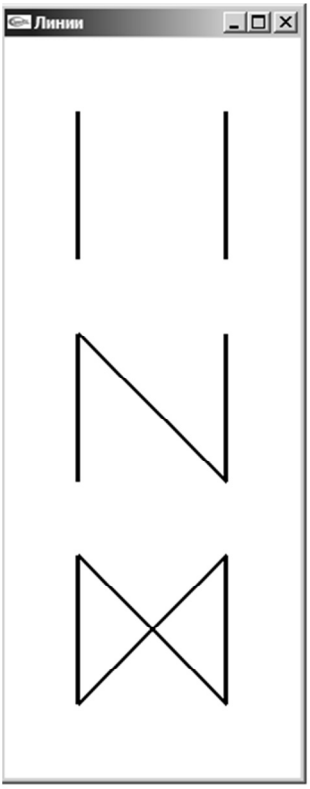
Рис. 23. Режимы задания точек

ПРИМИТИВ «ЛИНИИ»

У примитивов этого типа существует три варианта:

- «отдельные линии» или «отрезки» (`GL_LINES`);
- «незамкнутая ломаная» (`GL_LINE_STRIP`);
- «замкнутая ломаная» (`GL_LINE_LOOP`).

На Рис. 24 представлены примеры задания некоторой ломаной путем перечисления четырех узлов. Как можно видеть, при одном и том же коде внутри операторных скобок результат получается разным.




```
glBegin(GL_LINES);
  glVertex2i( 50, 350);
  glVertex2i( 50, 450);
  glVertex2i(150, 350);
  glVertex2i(150, 450);
glEnd();
```

```
glBegin(GL_LINE_STRIP);
  glVertex2i( 50, 200);
  glVertex2i( 50, 300);
  glVertex2i(150, 200);
  glVertex2i(150, 300);
glEnd();
```

```
glBegin(GL_LINE_LOOP);
  glVertex2i( 50,  50);
  glVertex2i( 50, 150);
  glVertex2i(150,  50);
  glVertex2i(150, 150);
glEnd();
```

Рис. 24. Задание ломаных разными примитивами

Очевидно, что ломаная типа «бабочка» получается автоматически при использовании примитива «замкнутая ломаная». Использование же примитива «незамкнутая ломаная» в этом случае приводит к необходимости дублирования (в конце списка перечисления) первой вершины, а для примитива «отрезки» необходимо продублировать все внутренние отрезки:

```
glBegin (GL_LINE_STRIP);
  glVertex2i ( 50,  50 );
  glVertex2i ( 50, 150 );
  glVertex2i (150,  50 );
  glVertex2i (150,  50 );
  glVertex2i (150, 150 );
  glVertex2i ( 50,  50 );
glEnd();
```

```
glBegin (GL_LINES);
  glVertex2i ( 50,  50 );  glVertex2i ( 50, 150 );
  glVertex2i ( 50, 150 );  glVertex2i (150,  50 );
  glVertex2i (150,  50 );  glVertex2i (150, 150 );
  glVertex2i (150, 150 );  glVertex2i ( 50,  50 );
glEnd();
```

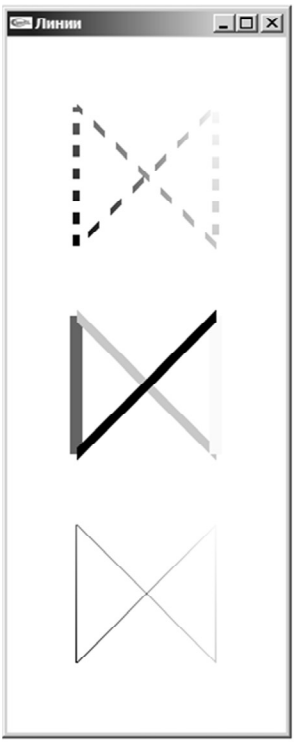
При задании ломаной каждой ее вершине можно указать свой цвет, который будет линейно интерполирован в каждом своем сегменте (отрезке). Команда `glShadeModel(GL_FLAT)` отменяет этот режим, в этом случае каждый сегмент закрашивается цветом своей второй вершины. Возвращение исходного режима осуществляется той же командой, но с параметром `GL_SMOOTH`.

Также можно указать и толщину линий с помощью команды `glLineWidth(*)`, в качестве параметра которой выступает новое значение толщины (по умолчанию она равна 1).

В некоторых случаях может быть полезным использование не сплошной линии для отрисовки отрезков ломаной, а по некоторому шаблону (маске, паттерну), например, пунктирной линией. Данный режим включается командой `glEnable(GL_LINE_STIPPLE)`, а сама мас-

ка задается командой `glLineStipple(GLint,GLushort)`, второй параметр которой задает сам шаблон, а первый – количество повторов битов шаблона.

Результат задания ломаной с вышеуказанными модификаторами представлен на Рис. 25.




```
glLineWidth(5);
glLineStipple(1,255);
glEnable(GL_LINE_STIPPLE);
glBegin(GL_LINE_LOOP);
  glColor3ub(  0,   0,   0); glVertex2i( 50, 350);
  glColor3ub(100, 100, 100); glVertex2i( 50, 450);
  glColor3ub(200, 200, 200); glVertex2i(150, 350);
  glColor3ub(250, 250, 250); glVertex2i(150, 450);
glEnd();
glDisable(GL_LINE_STIPPLE);

glShadeModel(GL_FLAT);
glLineWidth(9);
glBegin(GL_LINE_LOOP);
  glColor3ub(  0,   0,   0); glVertex2i( 50, 200);
  glColor3ub(100, 100, 100); glVertex2i( 50, 300);
  glColor3ub(200, 200, 200); glVertex2i(150, 200);
  glColor3ub(250, 250, 250); glVertex2i(150, 300);
glEnd();
glShadeModel(GL_SMOOTH);

glLineWidth(1);
glBegin(GL_LINE_LOOP);
  glColor3ub(  0,   0,   0); glVertex2i( 50,  50);
  glColor3ub(100, 100, 100); glVertex2i( 50, 150);
  glColor3ub(200, 200, 200); glVertex2i(150,  50);
  glColor3ub(250, 250, 250); glVertex2i(150, 150);
glEnd();
```

Рис. 25. Режимы задания ломаной

ПРИМИТИВ «ТРЕУГОЛЬНИКИ»

У примитивов данного типа также существует три варианта:

- «отдельные треугольники» (`GL_TRIANGLES`),
- «треугольники с общей вершиной» (`GL_TRIANGLE_FAN`),
- «треугольники с общими гранями» (`GL_TRIANGLE_STRIP`).

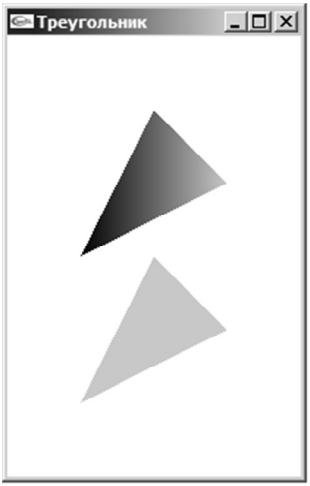
Как и для примитива «линии», команда `glShadeModel` также переключает режим интерполяции цвета (Рис. 26).

Для наложения маски (размером 32x32 бита) на треугольник используется команда `glPolygonStipple(*)`, в качестве параметра которой и указывается эта маска. Включение режима использования маски выполняется командой `glEnable(GL_POLYGON_STIPPLE)`. На Рис. 27 приведен пример генерации маски и результат ее влияния на отображение треугольника.

Другой вариант изменения режима отрисовки треугольника – это использование команды `glPolygonMode(GLenum,GLenum)`, в качестве параметров которой передается тип граней многоугольника (лицевые, обратные или обе) и режим отображения (точками, линиями или полная закраска). Результат применения команды представлен на Рис. 28.

В режиме «контур» треугольник отрисовывается только своими ребрами. При необходимости можно поставить запрет на отрисовку тех ребер, которые не должны оказаться видны. У каждой вершины есть булевский флаг, показывающий, должно ли отображаться ребро, выходящее из этой вершины. Данный флаг переключается командой `glEdgeFlag(*)` с параметром `GL_TRUE` (по умолчанию) или `GL_FALSE`. Например, если пара треугольников образует квадрат, то диагональное ребро, скорее всего, не должно быть видно (Рис. 29). Поскольку режим «контур» эквивалентен примитиву «замкнутая ломаная», то к нему применимы и соответствующие режимы отображения, то же самое касается и режима «по точкам».

Если при задании примитива число указанных вершин оказалось недостаточным, тогда последний примитив не отображается. Например, при задании только двух вершин на экране ничего не будет нарисовано, ибо нужно задать минимум 3 вершины.

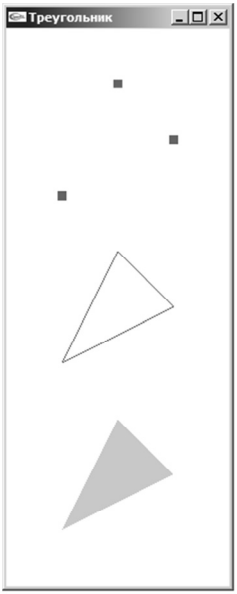



```
glShadeModel(GL_FLAT);
glBegin(GL_TRIANGLES);
glColor3ub(  0,   0,   0); glVertex2i( 50, 50);
glColor3ub(100, 100, 100); glVertex2i(100,150);
glColor3ub(200, 200, 200); glVertex2i(150,100);
glEnd();

glShadeModel(GL_SMOOTH);
glBegin(GL_TRIANGLES);
glColor3ub(  0,   0,   0); glVertex2i( 50,150);
glColor3ub(100, 100, 100); glVertex2i(100,250);
glColor3ub(200, 200, 200); glVertex2i(150,200);
glEnd();
```

Рис. 26. Режимы интерполяции цвета

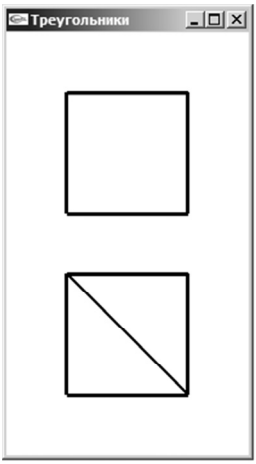

```
GLubyte maska[32][32];
for(int i=0;i<32;i++) for(int j=0;j<32;j++) maska[i][j] = (i-j)%2;

glPolygonStipple(&maska[0][0]);

glEnable(GL_POLYGON_STIPPLE);
glBegin(GL_TRIANGLES);
  glColor3ub(  0,   0,   0); glVertex2i( 50,150);
  glColor3ub(100, 100, 100); glVertex2i(100,250);
  glColor3ub(200, 200, 200); glVertex2i(150,200);
glEnd();
glDisable(GL_POLYGON_STIPPLE);

glBegin(GL_TRIANGLES);
  glColor3ub(  0,   0,   0); glVertex2i( 50, 50);
  glColor3ub(100, 100, 100); glVertex2i(100,150);
  glColor3ub(200, 200, 200); glVertex2i(150,100);
glEnd();
```

Рис. 27. Использование маски для закраски треугольника




```
glColor3ub(100, 100, 100);
glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
glBegin(GL_TRIANGLES);
glVertex2i( 50,350);
glVertex2i(100,450);
glVertex2i(150,400);
glEnd();

glColor3ub(100, 100, 100);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_TRIANGLES);
glVertex2i( 50,200);
glVertex2i(100,300);
glVertex2i(150,250);
glEnd();

glColor3ub(200, 200, 200);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glBegin(GL_TRIANGLES);
glVertex2i( 50 ,50);
glVertex2i(100,150);
glVertex2i(150,100);
glEnd();
```

Рис. 28. Режимы отображения примитива




```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_TRIANGLES);
  glVertex2i( 50,200); glEdgeFlag(GL_FALSE);
  glVertex2i( 50,300); glEdgeFlag(GL_TRUE);
  glVertex2i(150,200); glEdgeFlag(GL_FALSE);
  glVertex2i( 50,300); glEdgeFlag(GL_TRUE);
  glVertex2i(150,200);
  glVertex2i(150,300);
glEnd();

glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_TRIANGLES);
  glVertex2i( 50, 50);
  glVertex2i( 50,150);
  glVertex2i(150, 50);
  glVertex2i( 50,150);
  glVertex2i(150, 50);
  glVertex2i(150,150);
glEnd();
```

Рис. 29. Сокрытие "ненужных" ребер

ПРИМИТИВ «МНОГОУГОЛЬНИКИ»

К данным примитивам относятся:

- «отдельные четырехугольники» (`GL_QUADS`),
- «четырехугольники с общими гранями» (`GL_QUAD_STRIP`),
- «полигон» (`GL_POLYGON`).

Предполагается, что данные геометрические объекты являются простыми выпуклыми многоугольниками, вершины которых перечисляются по или против часовой стрелки. В противном же случае возможны некорректности при отображении, как показано на Рис. 30. Собственно, неоднозначность обработки многоугольников и привела к тому, что объекты такого типа были исключены из поздних версий таких библиотек, как OpenGL и DirectX.


```
glBegin(GL_QUADS);
  glColor3ub(  0,   0,   0); glVertex2i( 50, 350);
  glColor3ub(250, 250, 250); glVertex2i(150, 450);
  glColor3ub(200, 200, 200); glVertex2i(150, 350);
  glColor3ub(100, 100, 100); glVertex2i( 50, 450);
glEnd();
```

```
glBegin(GL_QUADS);
  glColor3ub(  0,   0,   0); glVertex2i( 50, 200);
  glColor3ub(100, 100, 100); glVertex2i( 50, 300);
  glColor3ub(200, 200, 200); glVertex2i(150, 200);
  glColor3ub(250, 250, 250); glVertex2i(150, 300);
glEnd();
```

```
glBegin(GL_QUADS);                              ;
  glColor3ub(  0,   0,   0); glVertex2i( 50,  50);
  glColor3ub(100, 100, 100); glVertex2i( 50, 150);
  glColor3ub(250, 250, 250); glVertex2i(150, 150);
  glColor3ub(200, 200, 200); glVertex2i(150,  50).
glEnd();
```

Рис. 30. Влияние порядка перечисления узлов

При использовании связных четырехугольников (с общей границей) для построения первого четырехугольника требуется 4 вершины, а для второго, третьего и так далее – всего по 2 вершины. При этом порядок обхода вершин не совпадает с порядком их задания (Рис. 31): 1-2-4-3 и 3-4-6-5. В результате, направление обхода вершин всех связных четырехугольников совпадает с направлением обхода первого четырехугольника.

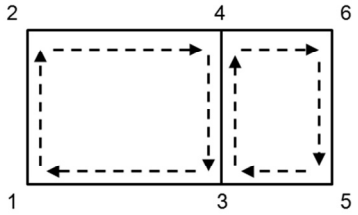
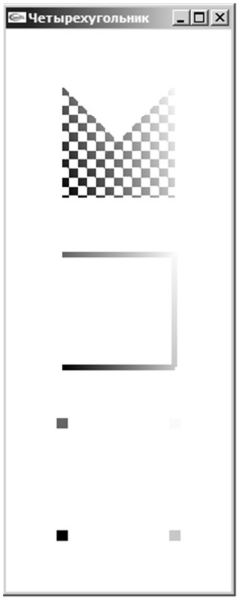


Рис. 31. Порядок обхода и нумерация

Фактически, многоугольники являются набором треугольников с общими вершинами. Соответственно, для них применяются те же самые правила, что и для треугольников (Рис. 32).




```
glEnable(GL_POLYGON_STIPPLE);
glBegin(GL_QUADS);
  glColor3ub(  0,   0,   0); glVertex2i( 50, 350);
  glColor3ub(100, 100, 100); glVertex2i( 50, 450);
  glColor3ub(200, 200, 200); glVertex2i(150, 350);
  glColor3ub(250, 250, 250); glVertex2i(150, 450);
glEnd();
glDisable(GL_POLYGON_STIPPLE);

glLineWidth(5);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_QUADS);
  glEdgeFlag (GL_FALSE);
  glColor3ub(  0,   0,   0); glVertex2i( 50, 200);
  glEdgeFlag (GL_TRUE);
  glColor3ub(100, 100, 100); glVertex2i( 50, 300);
  glColor3ub(250, 250, 250); glVertex2i(150, 300);
  glColor3ub(200, 200, 200); glVertex2i(150, 200);
glEnd();

glPointSize(10);
glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
glBegin(GL_QUADS);
  glColor3ub(  0,   0,   0); glVertex2i( 50,  50);
  glColor3ub(100, 100, 100); glVertex2i( 50, 150);
  glColor3ub(250, 250, 250); glVertex2i(150, 150);
  glColor3ub(200, 200, 200); glVertex2i(150,  50);
glEnd();
```

Рис. 32. Режимы отображения четырехугольника

СПИСКИ ВЕРШИН

При задании объектов, состоящих из большого количества вершин, вызов команды `glVertex` (вместе с сопутствующими ей командами типа `glColor`) становится уже накладным с точки зрения вычислительных ресурсов.

В этом случае рекомендуется использовать команды для работы со списком (массивом) вершин и цветов. Для этого необходимо:

1) объявить и инициализировать данные массивы;
2) передать эти массивы библиотеке;
3) включить соответствующие режимы рисования;
4) вызвать команду отрисовки массива;
5) отключить режим рисования.

В двумерном случае элементом массива вершин является пара чисел, являющихся (x,y)-координатами соответствующей вершины. Аналогично, элементом массива цветов является тройка чисел (для режима RGB), определяющая цвет соответствующей вершины.

Передача массива вершин и цветов осуществляется командами `glVertexPointer` и `glColorPointer` соответственно. В качестве параметров этих команд нужно задать число элементов, тип данных, сдвиг и сам массив элементов. Если вершина и ее цвет хранятся в соседних позициях общего массива, тогда в качестве сдвига нужно задавать не 0, а размерность элемента цвета.

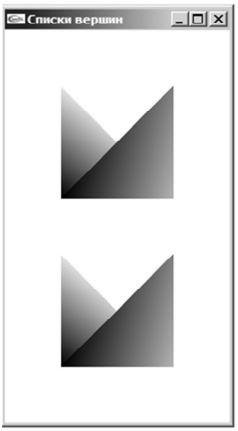
Включение данного режима рисования выполняется командой `glEnableClientState` с параметром `GL_VERTEX_ARRAY` для вершин или `GL_COLOR_ARRAY` для цвета. Выключение режима рисования выполняется командой `glDisableClientState` с теми же самыми параметрами.

Команда `glDrawArrays` выполняет отрисовку указанного массива вершин с учетом массива цветов. В качестве параметров она требует тип примитива, начальную позицию в массивах и количество отрисовываемых вершин. Как можно видеть, данная команда позволяет не отрисовывать начальные или последние вершины.

Пример задания массива pVer красных отрезков:

```
glEnableClientState(GL_VERTEX_ARRAY);

glVertexPointer ( 2, GL_FLOAT, 0, pVer);
glColor3ub ( 255, 0, 0);
glDrawArrays(GL_LINES,0,3);

glDisableClientState(GL_VERTEX_ARRAY);
```

На Рис. 33 представлен результат сравнения задания списка вершин с операторными скобками. Для общности, оба варианта работают с одним и тем же способом объявления массивов вершин и цветов.




```
GLfloat Vertex1[4][2] = { 50, 50, 50,150, 150, 50, 150,150};
GLfloat Vertex2[4][2] = { 50,200, 50,300, 150,200, 150,300};
GLfloat Colors[4][3] = { 0,0,0, 0.9,0.9,0.9, 0.7,0.7,0.7, 0.5,0.5,0.5 };

glBegin(GL_QUADS);
  glColor3fv(Colors[0]); glVertex2fv(Vertex1[0]);
  glColor3fv(Colors[1]); glVertex2fv(Vertex1[1]);
  glColor3fv(Colors[2]); glVertex2fv(Vertex1[2]);
  glColor3fv(Colors[3]); glVertex2fv(Vertex1[3]);
glEnd();

glVertexPointer(2, GL_FLOAT, 0, Vertex2);
glColorPointer(3, GL_FLOAT, 0, Colors);

glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);

glDrawArrays(GL_POLYGON, 0, 4);

glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
```

Рис. 33. Отрисовка четырехугольника через список вершин

ПРИМЕР РИСОВАНИЯ ЭЛЛИПСА

Каноническое уравнение эллипса в декартовой системе координат имеет вид:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1,$$

где a – большая полуось, b – малая полуось (при $a = b$ эллипс вырождается в окружность).

В параметрическом виде это уравнение можно представить в виде системы:

$$\begin{cases} x = a \sin t \\ y = b \cos t' \end{cases}$$

где параметр $t \in [0,2\pi]$ – это угол.

Поскольку примитива «эллипс» не существует, то возникает проблемы выбора подходящего примитива для отрисовки и уровня дискретизации (количества вершин фигуры).

Рассмотрим процесс отрисовки на примере построения эллипса с полуосями a и b равными 200 и 100 соответственно, при этом размер точек и линий был установлен в 5 пикселей, а уровень дискретизации `split` каждый раз подбирался экспериментально.

Для удобства разбора особенностей использования различных примитивов была написана функция set_ver, устанавливающая одну вершину через параметры эллипса и угол.

При отрисовки рассмотрим особенности использования следующих примитивов:

- «точки»,
- «отрезки»,
- «замкнутая ломаная»,
- «отдельные треугольники»,
- «треугольники с общей вершиной»,
- «полигон».

ФУНКЦИЯ ОТРИСОВКИ ВЕРШИНЫ ЭЛЛИПСА

Для начала рассмотрим простой код отрисовки одной вершины фигуры «эллипс» (параметр t является углом, задаваемом в градусах):

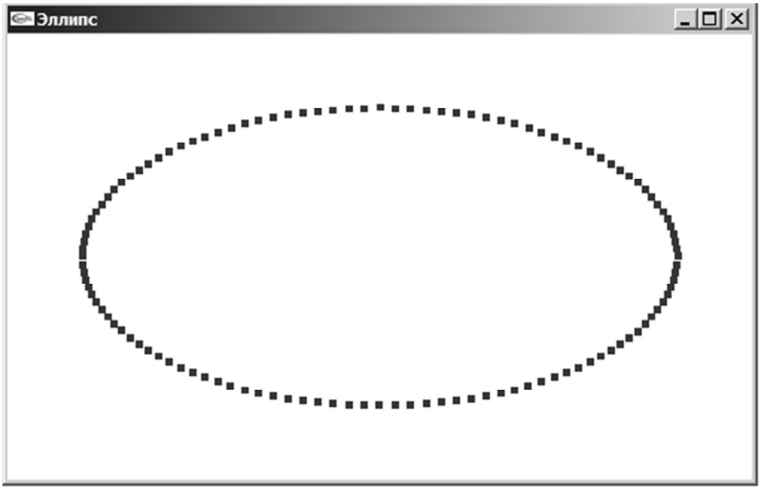
```
void set_ver ( float t )
{
   const float pi = 3.141592;      // число ПИ
   const float rad = t * pi / 180; // перевод в радианы
   const float a = 200;            // большая полуось
   const float b = 100;            // малая полуось

   float x = a * cos(rad);
   float y = b * sin(rad);
   glVertex2f ( x, y );
}
```

ИСПОЛЬЗОВАНИЕ ПРИМИТИВА «ТОЧКИ»

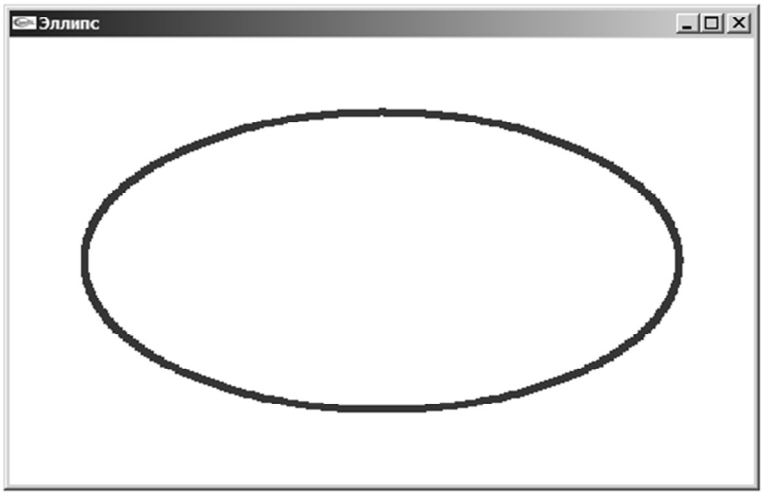
Код отрисовки эллипса через примитив «точки» будет выглядеть следующим образом:

```
   void ellipse()
   {
          float split = 300;
          glPointSize (5);
          glColor3f (1, 0, 0);

          glBegin ( GL_POINTS );
          float dfi = 360. / split;
          for (float fi = 0; fi <= 360; fi += dfi)
                set_ver ( fi );
          glEnd();
   }
```

На Рис. 34 представлен результат для двух уровней дискретизации (100 и 300).




```
float split = 100;
```




```
float split = 300;
```

Рис. 34. Эллипс через примитив «точки»

ИСПОЛЬЗОВАНИЕ ПРИМИТИВА «ЛИНИИ»

Код отрисовки эллипса через примитив «замкнутая ломаная» будет выглядеть следующим образом:

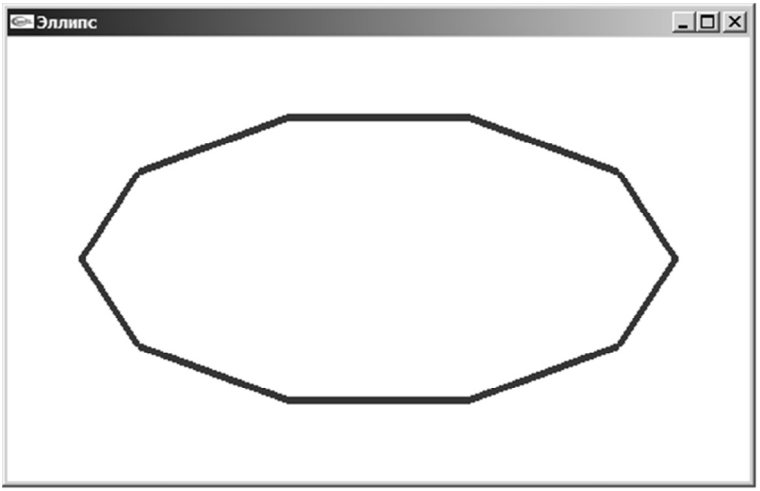
```
void ellipse()
{
	float split = 30;

	glBegin(GL_LINE_LOOP);

	float dfi = 360. / split;
	for(float fi = 0; fi < 360; fi += dfi )
		set_ver ( fi );

	glEnd();
}
```

Результат работы этой функции с двумя уровнями дискретизации представлен на Рис. 35.

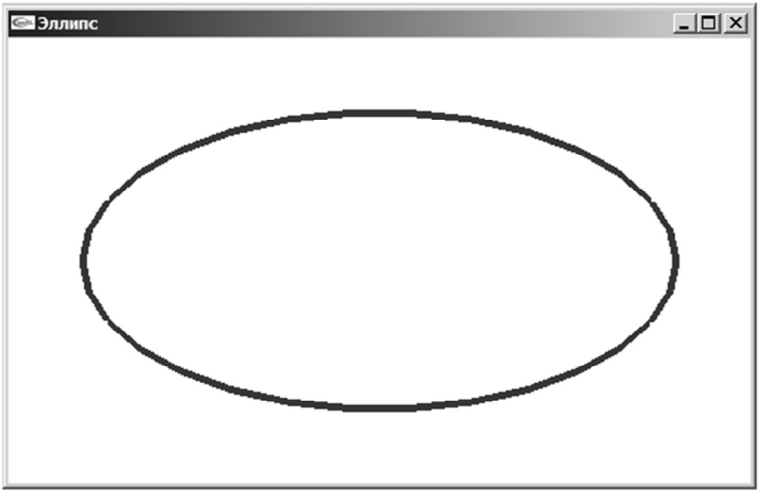
В случае использования примитива «незамкнутая ломаная», как уже говорилось ранее, в этом коде нужно будет, например, после цикла продублировать задание первой вершины.

При использовании же примитива «отрезки» граница эллипса станет разрывной ("пропадет" половина ребер). В этом случае необходимо удвоить число вызовов функции `set_ver`, сдублировав нужные вершины:

```
for(float fi = 0; fi < 360; fi += dfi )
{
	set_ver(fi);
	set_ver(fi+dfi);
}
```




```
float split = 10;
```




```
float split = 30;
```

Рис. 35. Эллипс через примитив «линии»

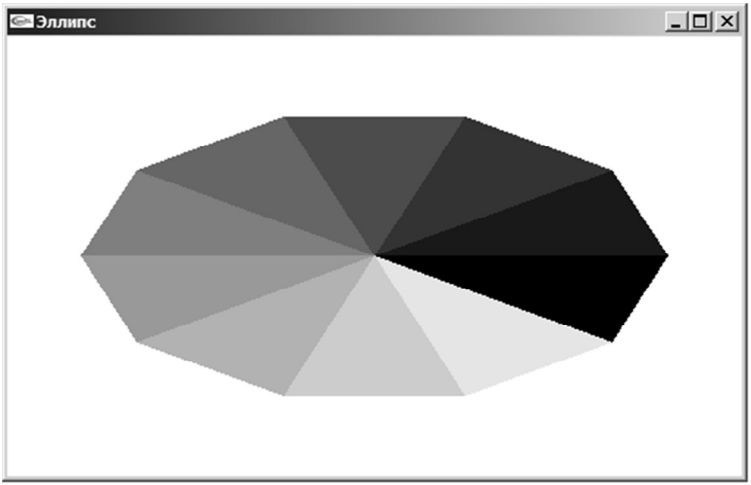
ИСПОЛЬЗОВАНИЕ ПРИМИТИВА «ТРЕУГОЛЬНИКИ»

При необходимости нарисовать закрашенный эллипс возникает необходимость в использовании треугольников. В случае приходится задавать по 3 вершины, одна из которых всегда расположена в точке (0,0). На Рис. 36 приведен пример отрисовки закрашенного эллипса с помощью десяти независимых треугольников, которые для наглядности раскрашены разными оттенками серого.

Для получения качественного эллипса в этом масштабе необходимо утроить уровень дискретизации: с 10 до 30. Использование же примитива «треугольники с общей вершиной» позволяет сократить число вершин в 3 раза. Соответствующий код имеет вид:

```
void ellipse()
{
	float split = 30;

	glBegin(GL_TRIANGLE_FAN);

	glColor3f(1, 1, 1);
	glVertex2f(0, 0);

	glColor3f(0, 0, 0);

	float dfi = 360. / split;
	for (float fi = 0; fi <= 360; fi += dfi)
		set_ver ( fi );

	glEnd();
}
```

На Рис. 37 представлен результат отрисовки эллипса через данный примитив в режимах «каркас» и «с заливкой». Для наглядности в этом случае цвет центральной точки был задан белым, а цвет точек периметра – черным.




```
{
    float split = 10;

    glColor3f(1, 0, 0);

    glBegin(GL_TRIANGLES);

    float dfi = 360. / split;
    for (float fi = 0; fi < 360; fi += dfi)
    {
        float col = fi / 360.;
        glColor3f(col, col, col);
        glVertex2f(0, 0);
        set_ver (fi);
        set_ver (fi - dfi);
    }

    glEnd();
}
```

Рис. 36. Эллипс через примитив «треугольники»

,

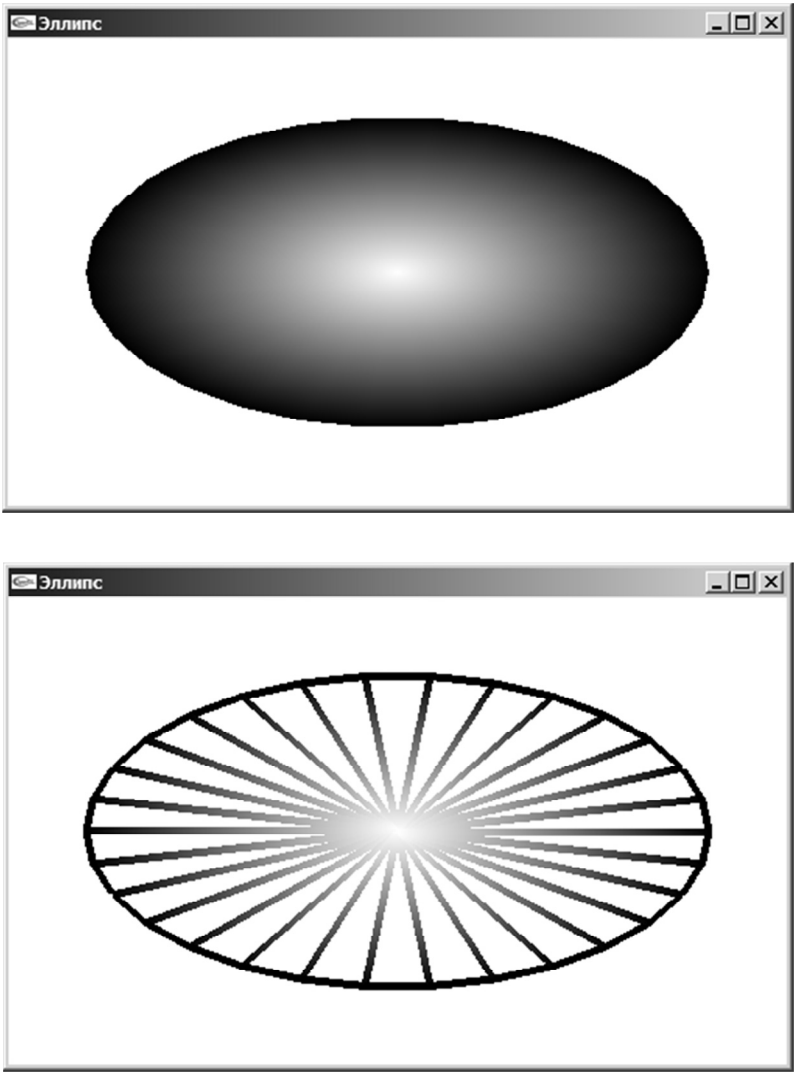


Рис. 37. Эллипс через примитив «треугольники с общей вершиной»

ИСПОЛЬЗОВАНИЕ ПРИМИТИВА «ПОЛИГОН»

На Рис. 38 представлен результат отрисовки эллипса через данный примитив (для наглядности цвет вершин рассчитывался как градиент серого), а ниже приведен соответствующий код:

```
void ellipse()
{
	float split = 30;

	glBegin(GL_POLYGON);

	float dfi = 360. / split;
	for (float fi = 0; fi <= 360; fi += dfi)
	{
		float col = fi / 360; glColor3f(col, col, col);
		set_ver(fi);
	}

	glEnd();
}
```

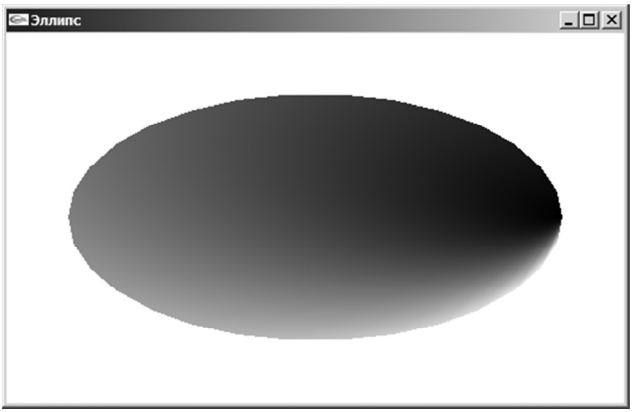


Рис. 38. Эллипс через примитив «полигон»

ГЕОМЕТРИЧЕСКИЕ ПРЕОБРАЗОВАНИЯ

Очень часто бывает необходимость выполнить над примитивами такие стандартные геометрические операции, как сдвиг, масштабирование и поворот. Эти операции можно, конечно, выполнять вручную для каждой вершины, но гораздо эффективнее использовать команды OpenGL, нужным образом меняющие текущую систему координат, в которой и задаются все последующие вершины. Соответственно, данные операции должны применяться ДО операторных скобок.

Фактически, эти преобразования являются модельно-видовыми, устанавливающими матрицу, на которую будут умножаться все вершины, задаваемые после выполнения данных команд.

Вначале матрица модельно-видовых преобразований является единичной. И, например, последовательный сдвиг сначала на 10, а потом на 20 приводит к тому, что результирующая матрица будет сразу осуществлять сдвиг сразу на 30, а начало системы координат сдвинется на 30. Сброс системы координат в исходное (единичное) состояние осуществляется командой `glLoadIdentity()`.

При выполнении данных команд нужно помнить, что, например, последовательность «сдвиг и поворот» не приведет к тому же результату, что и «поворот и сдвиг», поскольку во втором случае сдвиг осуществляется по уже повернутой оси, а не исходной.

В частности, если задать точку (1,0), затем выполнить сдвиг вправо на 2 и задать точку (3,0), то в результате получим две точки с координатами (1,0) и (5,0). Это произошло потому, что вследствие сдвига центр системы координат сместился на 2, соответственно, вторая точка задается уже в обновленной системе координат.

Однако если перед заданием второй точки выполнить команду `glLoadIdentity`, тогда система координат вернется в исходное состояние, и в результате получим точки с координатами (1,0) и (3,0).

При необходимости, текущую систему координат можно сначала запомнить командой `glPushMatrix()`, задать нужные вершины, а потом уже восстановить командой `glPopMatrix()`.

ОПЕРАЦИЯ «СДВИГ»

Сдвиг (перемещение) – это перенос вершин вдоль вектора (x,y,z). В двумерном случае сдвиг по OZ бессмысленен, поэтому компоненту z имеет смысл занулить.

Операция сдвига выполняется командой `glTranslate`:

```
void glTranslatef ( GLfloat  x, GLfloat  y, GLfloat  z )
void glTranslated ( GLdouble x, GLdouble y, GLdouble z )
```

На Рис. 39 приведен пример выполнения команды `glTranslatef(100, 50, 0)`. Выполнение данной команды означает, что примитив будет сдвинут по оси OX на 100, а по оси OY – на 50.

Рис. 39. Операция сдвига

ОПЕРАЦИЯ «МАСШТАБИРОВАНИЕ»

Масштабирование (растяжение или сжатие) выполняется вдоль трех осей OX, OY и OZ командой `glScale`:

```
void glScalef ( GLfloat  x, GLfloat  y, GLfloat  z )
void glScaled ( GLdouble x, GLdouble y, GLdouble z )
```

На Рис. 40 приведен пример выполнения команды масштабирования `glScalef(2, 0.25, 1)`. Поскольку случай двумерный, то не будет отличия от выполнения команды `glScalef(2, 0.5, 0)`. Выполнение данной команды означает, что примитив будет растянут по оси OX в 2 раза, а по оси OY – сужен в 4 раза.

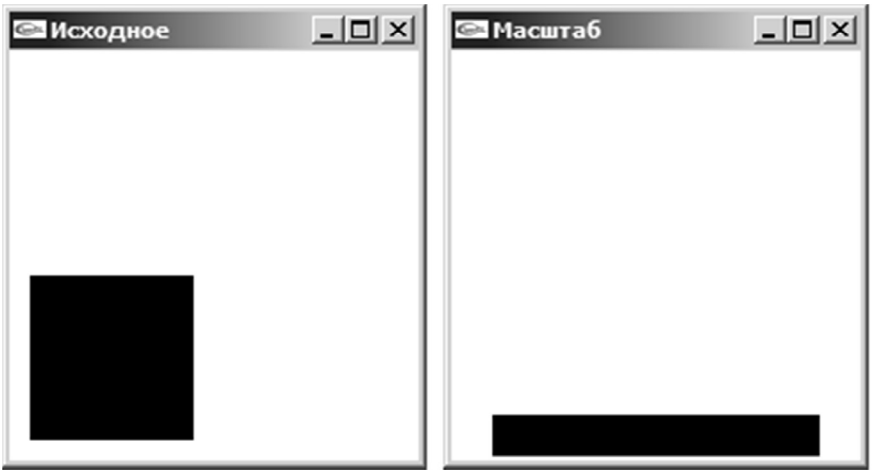


Рис. 40. Операция масштабирования

ОПЕРАЦИЯ «ПОВОРОТ»

Поворот (вращение) выполняется на угол `angle` вокруг вектора (x,y,z) командой `glRotate:`

```
void glRotatef ( GLfloat angle,
                 GLfloat  x, GLfloat  y, GLfloat  z )
void glRotated ( GLdouble angle,
                 GLdouble x, GLdouble y, GLdouble z )
```

В двумерном случае поворот вокруг осей OX и OY смысла не имеет (соответствующие компоненты зануляются), так что остается только поворот вокруг OZ.

Пример выполнения команды `glRotatef (15, 0, 0, 1)` приведен на Рис. 41. Выполнение данной команды означает, что примитив будет повернут по оси OZ на 15°.

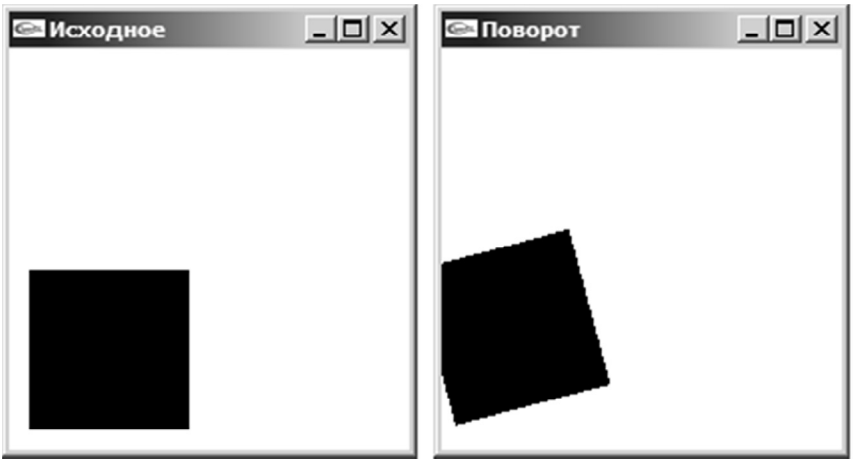


Рис. 41. Операция поворота

ВЫВОД НА ЭКРАН

Процесс обработки вершин в графическом конвейере завершается этапом отсечения тех примитивов, чьи вершины не попали в область видимости наблюдателя (куб видимости).

Далее оставшиеся примитивы проходят этап растеризации и ряд дополнительных тестов и сохраняются в виде пикселей в буфере кадра.

Таким образом, для вывода буфера кадра на экран необходимо сначала задать куб видимости и порт просмотра.

ПОРТ ПРОСМОТРА

Область (окно, порт) просмотра (view port) – это часть окна для вывода результирующего изображения. Порт представляет собой прямоугольник, размеры и позиция которого задаются командой `glViewPor`:

```
void glViewPort(GLint x0, GLint y0,
                GLint width, GLint height)
```

Первые два параметра этой команды – это координата левой нижней точки области вывода, а вторая пара – это ширина и высота области вывода.

Соответственно, преобразование реальных координат (X, Y) к оконным (x, y) выполняется по следующим формулам:

$$x = x_0 + (X + 1)\frac{\text{width}}{2}$$

$$y = y_0 + (Y + 1)\frac{\text{height}}{2}$$

БУФЕР КАДРА

В большинстве случаев, *буфер кадра* (frame buffer) – это область видеопамяти, временно содержащий различные данные, необходимые для вывода изображения на экран.

В простейшем случае буфер кадра состоит только из одного буфера - буфера цвета. *Буфер цвета* (color buffer) – это области видеопамяти для временного хранения данных о пикселях, требуемых для отображения одного кадра (полного изображения) на экране монитора, соответственно, емкость буфера определяется глубиной цвета и количеством пикселей.

При *одиночной буферизации* экран имеет только один буфер, в который происходит как запись, так и чтение данных. При слишком быстром изменении изображения на экране могут появиться связанные с этим артефакты.

В случае *двойной буферизации* экран имеет уже два цветовых буфера – передний и задний. Из переднего буфера происходит вывод на экран, а в это время в заднем буфере параллельно подготавливается новое изображение. При поступлении запроса на обмен содержимое заднего и переднего буферов логически меняются местами. В результате обеспечивается гладкость анимации.

Подготовка буфера кадра начинается с его очистки командой

```
glClear(GL_COLOR_BUFFER_BIT).
```

Очистка буферов производится цветом, предварительно установленным командой `glClearColor`, параметрами которой является 4 вещественные компоненты цвета модели RGBA. Например, команда установки белого цвета выглядит следующим образом:

```
glClearColor(1,1,1,1).
```

После задания всех объектов для визуализации, процедура подготовки буфера кадра завершается командой `glFinish()`, которая и отправляет содержимое буфера на экран. Для досрочного отображения буфера на экране можно использовать команду `glFlush()`.

КУБ ВИДИМОСТИ

Объем отсечения (*куб видимости*) – это область, доступная для наблюдения. Т.е. объекты, чьи вершины не попадают в куб видимости, не будут отображаться на экране. Обычно куб видимости задается ортографической (параллельной) или перспективной проекцией в левосторонней системе координат. В результате генерируется матрица, на которую с этого момента умножаются все вершины.

Поскольку графика двумерная, тогда в качестве проекции имеет смысл использовать только ортографическую проекцию.

В общем случае ортографическая проекция задается командой `glOrtho`, однако в двумерном случае имеет смысл использовать команду `gluOrtho2D`:

```
void glOrtho(GLdouble left, GLdouble right,
             GLdouble bottom, GLdouble top,
             GLdouble near, GLdouble far)

void gluOrtho2D(GLdouble left, GLdouble right,
                GLdouble bottom, GLdouble top)
```

Параметры первой команды задают точки (`left, bottom, -near`) и (`right, top, -near`), которые отвечают левому нижнему и правому верхнему углам окна вывода. Параметры `near` и `far` задают расстояние до ближней и дальней плоскостей отсечения по дальности от точки (0,0,0) и могут быть отрицательными.

Во второй команде, в отличие от первой, значения `near` и `far` устанавливаются равными -1 и 1 соответственно.

Запуск команды `gluOrtho2D(0, width, 0, height)` означает, что центр системы координат будет установлен в левый нижний угол области просмотра. Таким образом, отображаться будут только те объекты, которые находятся в первой четверти. Соответственно, для того, чтобы центр системы координат был по центру окна, нужно выполнить команду `gluOrtho2D(-width/2, width/2, -height/2, height/2)`.

ДОПОЛНИТЕЛЬНЫЕ ГРАФИЧЕСКИЕ БИБЛИОТЕКИ

В связи с ограничениями, накладываемые стандартом OpenGL, различными разработчиками были созданы дополнительные библиотеки для расширения возможностей OpenGL и облегчения создания соответствующих графических приложений.

БИБЛИОТЕКА GLU

Библиотека **GLU** (OpenGL Utility Library) фактически является стандартной надстройкой над OpenGL и обычно поставляется вместе с ней. Библиотека состоит из ряда функций, использующих библиотеку OpenGL, но предоставляющих больше удобства для пользователя. Поскольку эти функции реализуются через стандартные функции OpenGL, то было принято решение выделить их в отдельную библиотеку.

В библиотеке реализованы следующие функции:

- дополнительные средства работы с матрицами и камерой,
- средства задания и обработки текстур,
- генерация сплайновых кривых и поверхностей (B-сплайн, Безье и NURBS),
- задание дополнительных графических примитивов (сфера, конус, диск и т. п.).

Все функции и константы данной библиотеки начинаются с префиксов "glu" и "GLU" соответственно.

БИБЛИОТЕКИ ДЛЯ КОНСОЛЬНОГО ПРИЛОЖЕНИЯ

Библиотека **GLUT** (OpenGL Utility Toolkit) – это библиотека утилит для создания простых кроссплатформенных консольных графических приложений, использующих OpenGL. В библиотеку входит ряд функция по организации операций ввода-вывода при работе с операционной системой: создание и управление окнами приложения, реакция на сообщения от клавиатуры и мыши, организация всплывающего (popup) меню и т.п. Дополнительно в библиотеку входят функции для задания высокоуровневых описаний таких сложных моделей, как чайник и сфера. Все функции библиотеки имеют префикс "glut", а константы – "GLUT". Скачать последнюю версию библиотеки можно по адресу: https://www.opengl.org/resources/libraries/glut/.

В свою очередь, у библиотеки GLUT есть своя надстройка – библиотека элементов интерфейса пользователя **GLUI** (OpenGL User Interface Library). Она предоставляет такие виджеты, как «radio button», «button», «check box» и т.п. Скачать библиотеку можно по адресу http://glui.sourceforge.net.

Поскольку библиотека GLUT перестала развиваться, у нее появилась открытая альтернатива – библиотека **FreeGLUT**, которая практически полностью совместима с GLUT. FreeGLUT даже включен в некоторые дистрибутивы Linux вместо GLUT. Скачать библиотеку можно по адресу http://freeglut.sourceforge.net.

Альтернативой библиотеки GLUT является библиотека **GLAUX** (Auxiliary Library), являющейся разработкой компании Microsoft. В отличие от GLUT, эта библиотека предоставляет возможности для работы с изображениями. Все функции библиотеки имеют префикс "aux", а константы – "AUX". К сожалению, разработка библиотеки была давно остановлена.

Библиотека **GLFW** (http://www.glfw.org) – это еще одна популярная открытая кроссплатформенная библиотека для создания и открытия окон, создания OpenGL-контекста и управления вводом.

БИБЛИОТЕКИ ДЛЯ ПРИЛОЖЕНИЯ В СРЕДЕ .NET

Для использования OpenGL под C# необходимо использовать дополнительные библиотеки («обертки»).

TaoFramework (https://sourceforge.net/projects/taoframework) – это свободно-распространяемая библиотека с открытым исходным кодом, предоставляющая разработчикам .NET и Mono доступ к возможностям популярных графических (OpenGL, GLFW, FreeGLUT, DevIL) и мультимедийных (SDL, OpenAL, FFmpeg) библиотек.

В настоящий момент разработка библиотеки Tao Framework переросла в проект **OpenTK** (https://sourceforge.net/projects/opentk). Последняя на данный момент версия 1.1.4 вышла 21 июля 2014 г. и поддерживает OpenGL версии 4.4. Библиотека также доступна под Linux, Mac OS X, *BSD, SteamOS, Android and iOS и может быть интегрирована в различные популярные графические интерфейсы пользователя (GUI): Windows.Forms, WPF, Qt и т.п.

Другой популярной современной «оберткой» над OpenGL является библиотека **SharpGL** (https://github.com/dwmkerr/sharpgl). Последняя на данный момент версия 2.4 вышла 30 января 2015 г. и поддерживает OpenGL версии 4.3.

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Внесите изменения в программу рисования эллипса так, чтобы рисовалась окружность радиусом 300.

2. Добавьте код для автоматического определения нужного уровня дискретизации.

3. Измените код так, чтобы центр системы координат был в центре окна и в левом нижнем углу, но при этом окружность не сдвигалась (оставалась в центре окна)

4. Используйте для отрисовки окружности все варианты примитива «линии».

5. Внесите изменения в программу рисования эллипса так, чтобы рисовалась закрашенная окружность (круг) с использованием всех вариантов примитивов «треугольники» и «многоугольники».

6. Используйте режимы «каркас» и «с заливкой».

7. Для примитива «полигон» используйте случайный порядок задания вершин и объясните полученный эффект.

8. Перепишите код с использованием списков вершин и цветов.

9. Сравните влияние на получаемое изображение типов `float` и `int` для задания вершин окружности.

10. Примените к окружности и кругу операции сдвига, поворота и вращения.

СОЗДАНИЕ КОНСОЛЬНОГО ПРИЛОЖЕНИЯ

Разработка демонстрационного приложения будет осуществляться с помощью библиотеки GLUT. Выбор библиотеки обусловлен простотой подключения (работы) и кроссплатформенностью.

Все функции библиотеки можно разделить на следующие классы:

- инициализация (установка режимов работы),
- функции обратного вызова,
- работа с окнами,
- работа с меню,
- дополнительные примитивы.

Минимальная программа, которая создает окно, в котором что-то отрисовывает, состоит из следующих элементов:

- инициализация GLUT,
- установка параметров окна,
- создание окна,
- задание функции отрисовки,
- задание функции изменения размеров окна,
- запуск главного цикла GLUT.

По умолчанию, у консольного приложения создается два окна: консольное и графическое. В большинстве случаем консольное окно бесполезно, и его отключить можно путем задания следующей директивы препроцессора (изменение настроек проекта):

```
#pragma comment(linker,"/SUBSYSTEM:windows/ENTRY:mainCRTStartup")
```

ИНИЦИАЛИЗАЦИЯ ПРИЛОЖЕНИЯ

Инициализация приложения начинается с вызова команды инициализации библиотеки – `glutInit(argc,argv)`. В качестве параметров ей передаются параметры запуска приложения, из которых она выбирает свои параметры.

Следующий этап инициализации – инициализация буфера кадра функцией `glutInitDisplayMode(*)`. В качестве единственного параметра выступают перечисленные через побитовую логическую операцию «ИЛИ» константы-идентификаторы режимов: одинарная (`GLUT_SINGLE`) или двойная (`GLUT_DOUBLE`) буферизация, режим RGB (`GLUT_RGB`) или RGBA (`GLUT_RGBA`) и т.п.

При использовании режима двойной буферизации переключение буферов выполняется командой `glutSwapBuffers()`, которая заменяет собою команду `glFinish()`.

При желании, вместо режима RGB можно включить режим палитры цвета параметром `GLUT_INDEX`, соответственно, загрузка цветов палитры выполняется командой `glutSetColor`.

Для создания окна с заголовком используется функция `glutCreateWindow(*)`, единственным параметром которой является текстовая строка с заголовком главного окна. Функция возвращает номер (идентификатор) окна.

Регистрация функции, отвечающей за отрисовку (обновления окна), осуществляется командой `glutDisplayFunc(*)`, единственным параметром которой является имя функции отрисовки. В том случае, когда необходим досрочный вызов функции обновления окна, рекомендуется использовать команду `glutPostRedisplay()`. Также возможна регистрация и других функций, отвечающих, например, за реакцию на нажатие клавиш или изменения размеров окна.

Контроль всех событий и автоматический вызов соответствующих функций осуществляется командой `glutMainLoop()`.

ФУНКЦИИ ОБРАТНОГО ВЫЗОВА

Функции *обратного вызова* (*callback-функции*) – это функции, зарегистрированные для реакции на определенные события (нажатие клавиш, изменения размера окна и т.п.). Список часто используемых функций приведен в табл. 9, а функциональных клавиш – в табл. 10.

Таблица 9

Основные команды регистрации функций обратного вызова

Функция	Описание реакции	Параметры функции
`glutDisplayFunc`	Отрисовка окна	
`glutIdleFunc`	Функция простоя	
`glutReshapeFunc`	Изменение размеров окна	новая ширина новая высота
`glutKeyboardFunc`	нажатие обычных (ASCII) клавиш	нажатая клавиша, координаты мыши
`glutSpecialFunc`	нажатие функциональных клавиш	нажатая клавиша, координаты мыши
`glutMouseFunc`	нажатие кнопок мыши	нажатая клавиша, статус нажатия, координаты мыши
`glutMotionFunc`	движение мыши с зажатой кнопкой	координаты мыши
`glutPassiveMotionFunc`	движение мыши без зажатия кнопки	координаты мыши

Для мыши определены нажатия левой (GLUT_LEFT_BUTTON), правой (GLUT_RIGHT_BUTTON) и центральной (GLUT_MIDDLE_BUTTON) кнопок. Также можно проверить и факт нажатия (GLUT_UP) или отпускания (GLUT_DOWN) кнопки.

Для определения того, какая спецклавиша-модификатор («CTRL», «ALT» или «SHIFT») была нажата, используется функция glutGetModifiers(), которая должна вызываться только внутри процедуры обработки сообщений от клавиатуры или мыши. Возвращаемое значение этой функции является либо одним из трех следующих констант (или их комбинаций): GLUT_ACTIVE_SHIFT (для клавиши «Shift»), GLUT_ACTIVE_CTRL (для клавиши «Ctrl») и GLUT_ACTIVE_ALT (для клавиши «Alt»).

При необходимости изменения позиции курсора можно использовать функцию SetCursorPos(*) из WinAPI.

Таблица 10

Список функциональных клавиш

Константа-идентификатор	Функциональная клавиша
GLUT_KEY_F1 – GLUT_KEY_F12	F1-F12
GLUT_KEY_LEFT GLUT_KEY_RIGHT GLUT_KEY_UP GLUT_KEY_DOWN	«стрелка влево» «стрелка влево» «стрелка влево» «стрелка влево»
GLUT_KEY_PAGE_UP GLUT_KEY_PAGE_DOWN	«Page UP» «Page DOWN»
GLUT_KEY_HOME GLUT_KEY_END GLUT_KEY_INSERT	«HOME» «END» «INSERT»

ФУНКЦИИ РАБОТА С ОКНАМИ

Функция `glutCreateWindow` при успешном завершении возвращает дескриптор созданного окна (с OpenGL-контекстом). С помощью этого дескриптора, можно сделать окно текущим (активным), после чего к этому окну уже можно применять такие операции, как смена позицию, изменение заголовка и т. п.

Перед созданием окна можно установить его начальный размер командой `glutInitWindowSize` и начальную позицию командой `glutInitWindowPosition`.

Библиотека GLUT позволяет также работать и с подокнами, которые делят родительское окно на области со своим собственным контекстом OpenGL (включая и обратные вызовы). Одним из возможных применений данного подхода является предоставление нескольких видов или проекций одной и той же сцены одновременно. Подокно создается командой `glutCreateSubWindow`. Все функции обратного вызова (за исключением `glutIdleFunc`), определенные после создания подокна, являются зарегистрированными только для данного подокна.

В табл.11 приведены основные функции для работы с окнами и подокнами. Дополнительно можно изменить тип курсора или перейти в игровой режим. На Рис. 42 приведен пример результата создания двух подокон, причем в подокнах (как и в родительском окне) вызывается одна и та же функция отрисовки треугольника.

Тип курсора меняется командой `glutSetCursor`, в качестве параметра которой выступает идентификатор типа курсора, например:

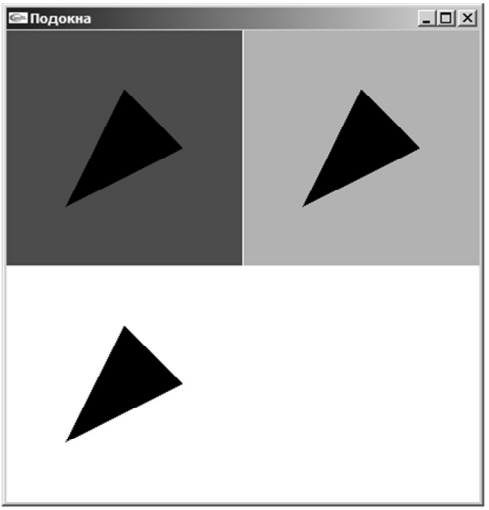
- `GLUT_CURSOR_UP_ARROW`,
- `GLUT_CURSOR_INFO`,
- `GLUT_CURSOR_NONE`,
- `GLUT_CURSOR_WAIT`.

Для перехода в игровой режим необходимо сначала установить параметры экрана командой `glutGameModeString`. Собственно, сам же вход в этот режим выполняется командой `glutEnterGameMode()`, а выход – `glutLeaveGameMode()`.

Таблица 11

Основные функции работы с окнами

Функция	Описание
`glutCreateWindow`	создание окна
`glutInitWindowSize`	задание начального размера окна
`glutInitWindowPosition`	задание начальной позиции окна
`glutGetWindow`	получение дескриптора активного окна
`glutSetWindow`	установление окна в активное состояние
`glutSetWindowTitle`	изменение заголовка окна
`glutPositionWindow`	установление новых координат окна
`glutReshapeWindow`	установление новых размеров окна
`glutIconifyWindow`	сворачивание окна в иконку
`glutHideWindow`	сокрытие окна
`glutShowWindow`	показывание окна
`glutDestroyWindow`	закрытие окна
`glutFullScreen`	переход в полноэкранный режим
`glutCreateSubWindow`	создание подокна




```
void cls(float r, float g, float b) { glClearColor(r,g,b,1); glClear(GL_COLOR_BUFFER_BIT); }
void draw()
{
 glColor3ub(0,0,0);
 glBegin(GL_TRIANGLES); glVertex2i(50,50); glVertex2i(100,150); glVertex2i(150,100);
 glEnd();
}
void Display(void) { cls(1,1,1); draw(); glFinish(); }
void Display2() { cls(0.3, 0.3, 0.3); draw();  glFinish(); }
void Display3() { cls(0.7, 0.7, 0.7); draw();  glFinish(); }
void main(int argc, char *argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB);
    glutInitWindowSize(401, 400);
    int parent = glutCreateWindow("Подокна");
    glutDisplayFunc(Display);
    glutReshapeFunc(Reshape);
    int subWindowR = glutCreateSubWindow(parent,   0,0, 200,200);
    glutDisplayFunc(Display2);
    glutReshapeFunc(Reshape);
    int subWindowG = glutCreateSubWindow(parent, 201,0, 200,200);
    glutDisplayFunc(Display3);
    glutReshapeFunc(Reshape);
    glutMainLoop();
}
```

Рис. 42. Создание подокон.

ФУНКЦИИ РАБОТЫ С МЕНЮ

Команда `glutCreateMenu(*)` создает новое всплывающее меню и возвращает его дескриптор. В качестве параметра этой команды выступает функция обратного вызова, реагирующая на выбор пункта меню. Значение, переданное функции обратного вызова, является номером выбранного пункта меню. Все пункты меню должны иметь сквозную нумерацию, общее количество пунктов меню возвращает функция `glutGet(GLUT_MENU_NUM_ITEMS)`. В табл. 12 приведены основные функции работы с меню, а на Рис. 43 – пример создания меню.

Таблица 12

Основные функции работы с меню

Функция	Описание функции
`glutAddMenuEntry`	добавление пункта в меню
`glutAddSubMenu`	добавление подменю в меню
`glutSetMenu`	выбор текущего меню
`glutGetMenu`	получение номера текущего меню
`glutRemoveMenuItem`	удаление элемента меню
`glutDestroyMenu`	удаление меню
`glutChangeToSubMenu`	изменение подменю в меню
`glutChangeToMenuEntry`	изменение пункта меню
`glutAttachMenu`	"привязывание" меню к кнопке мыши
`glutDetachMenu`	"отвязывание" меню от кнопки мыши


```
enum keys4menu { Cap1,Cap2,Cap3, Min } KeysMenu;

void Menu(int pos)
{
    int key = (keys)pos;
    switch( key )
    {
        case Cap1: glutSetWindowTitle("Заголовок 1"); break;
        case Cap2: glutSetWindowTitle("Заголовок 2"); break;
        case Cap3: glutSetWindowTitle("Заголовок 3"); break;
        case Min: glutIconifyWindow(); break;
    }
    glutPostRedisplay();
}

void CreateMenu()
{
    int menu1 = glutCreateMenu(Menu);
    glutAddMenuEntry("Заголовок 1", Cap1);
    glutAddMenuEntry("Заголовок 2", Cap2);
    glutAddMenuEntry("Заголовок 3", Cap3);

    int menu = glutCreateMenu(Menu);
    glutAddSubMenu("Смена заголовка окна", menu1);
    glutAddMenuEntry("Свернуть окно", Min);

    glutAttachMenu(GLUT_RIGHT_BUTTON);
}
```

Рис. 43. Создание меню

ПРИМЕРЫ СОЗДАНИЯ КОНСОЛЬНОГО ПРИЛОЖЕНИЯ

Сначала будет рассмотрено простейшее консольное приложение, а потом приведены варианты более сложных приложений: программы отрисовки набора точек и генерации фрактала.

ПРОСТЕЙШЕЕ КОНСОЛЬНОЕ ПРИЛОЖЕНИЕ

Данное приложение только создает окно (позиция и размер – по умолчанию) и закрашивает его белым цветом.

Соответственно, приложение состоит из следующих элементов:

- подключение библиотек GLUT и OpenGL,
- функция вывода на экран,
- функция изменения размеров окна с заданием куба видимости и порта просмотра,
- головная программа с инициализацией библиотеки GLUT.

Весь код данного приложения:

```
/* подключение библиотек GLUT и OpenGL */
#include <stdlib.h>
#include "glut.h"


/* Пустая функция отрисовки */
void Render()
{
}
```

```
/* Функция вывода на экран */
void Display(void)
{
    glClearColor(1,1,1,1);
    glClear(GL_COLOR_BUFFER_BIT);

       Render();

    glFinish();
}

/* Функция изменения размеров окна */
void Reshape(GLint w, GLint h)
{
       glViewport(0, 0, w, h);
       glMatrixMode(GL_PROJECTION);
       glLoadIdentity();
       gluOrtho2D(0, w, 0, h);
       glMatrixMode(GL_MODELVIEW);
       glLoadIdentity();
}


/* Головная программа */
void main(int argc, char *argv[])
{
       glutInit(&argc, argv);
       glutInitDisplayMode(GLUT_RGB);

       glutCreateWindow("Простейшее приложение");
       glutDisplayFunc(Display);
       glutReshapeFunc(Reshape);

       glutMainLoop();
}
```

ПРОГРАММА «ФРАКТАЛ Т-КВАДРАТ»

Построение фрактала начинается с построения "единичного" белого квадрата.

На первом шаге в центре исходного квадрата стоится черный квадрат со стороной 1/2, на вершинах которого в свою очередь строятся новые квадраты со стороной 1/4 и центрами в вершинах. Процесс рекурсивно повторяется для каждого из этих четырех квадратов.

Теоретически, процесс построения бесконечен, но его можно ограничить максимальным количеством итераций (которое можно даже аналитически прикинуть) или достижением стороны квадрата некоего минимального уровня (например, размера в 1 пиксель).

Собственно, само консольное приложение в этом случае получается минимально возможным, поскольку не требуются "навороты" в виде реакции на клавиатуру, создания меню и т. п. Единственное, имеет смысл добавить команды изменения размеров и заголовка окна – для примера их вызов будет осуществляться не инициализирующими функциями `glutCreateWindow` и `glutInitWindowSize`, а позже, в пользовательской функции `Render`.

В функции `Render` происходит изменение заголовка окна, установление размеров окна в 800x800 пикселей и вызов функции построения фрактала для квадрата радиусом 400 пикселей.

За саму отрисовку фрактала отвечает функция `FractalT`, в которой происходит инициализация параметров фрактала и запуск рекурсивной части генерации фрактала.

На Рис. 44 приведен процесс построения фрактала для разрешения в 200 пикселей, потребовалось 6 рекурсивных итераций и около 5 тысяч квадратов. Для разрешения же в 2000 пикселей необходимо уже 10 итераций и примерно 350 тысяч квадратов, при этом время выполнения – порядка 100 мс.

```
/* функция отрисовки */
void Render()
{
 int size = 400;
 glutSetWindowTitle ( "Построение фрактала «Т-квадрат»" );
 glutReshapeWindow(2*size, 2*size);
```

```
 FractalT(size);
}
/* подготовка для запуска процесса генерации */
void FractalT(int size)
{
 int center_x = size;
 int center_y = size;
 int max_iter = 10;

 glColor3ub( 0, 0, 0);
 draw_fractal_t ( center_x, center_y, len, max_iter );
}

/* рекурсивное построение фрактала */
void draw_fractal_t( int cx, int cy, int width, int iter)
{
    if(iter < 1) return;

    int radius = width / 2;
    if( radius < 1 ) return;

    int x0 = cx - radius;
    int x1 = cx + radius;
    int y0 = cy - radius;
    int y1 = cy + radius;

    glBegin(GL_QUADS);
      glVertex2i ( x0, y0 );
      glVertex2i ( x0, y1 );
      glVertex2i ( x1, y1 );
      glVertex2i ( x1, y0 );
    glEnd();

    draw_fractal_t ( x0, y0, radius, iter-1);
    draw_fractal_t ( x0, y1, radius, iter-1);
    draw_fractal_t ( x1, y1, radius, iter-1);
    draw_fractal_t ( x1, y0, radius, iter-1);
}
```

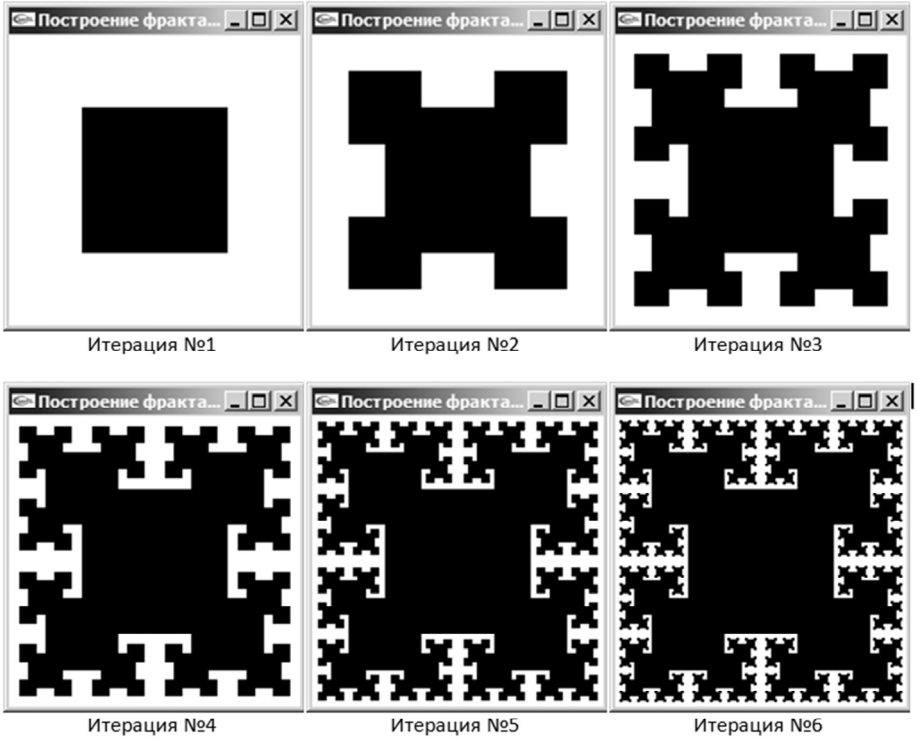


Рис. 44. Построение фрактала «Т-квадрат»

ПРОГРАММА «МНОЖЕСТВО ТОЧЕК»

В данной программе отрисовывается множество точек, на Рис. 45 приведен скриншот работы программы.

По левому клику точка добавляется в позицию курсора, по правому – вызывается всплывающее меню, а по среднему – удаляется последняя точка. Поскольку точки добавляются и удаляются динамически, то для их хранения был использован контейнер `vector` из библиотеки `STL`.

Программа также позволяет сдвигать сразу все точки по вертикали и горизонтали, менять покомпонентно их размер и цвет (текущие значения RGB-компонент отображаются в заголовке окна).

Команды клавиатуры продублированы простым всплывающим меню.

Хотя приложение и простое, но код получается громоздким из-за реализации функций работы с мышью, клавиатурой и меню. В связи с этим листинг программы вынесен в Приложение.

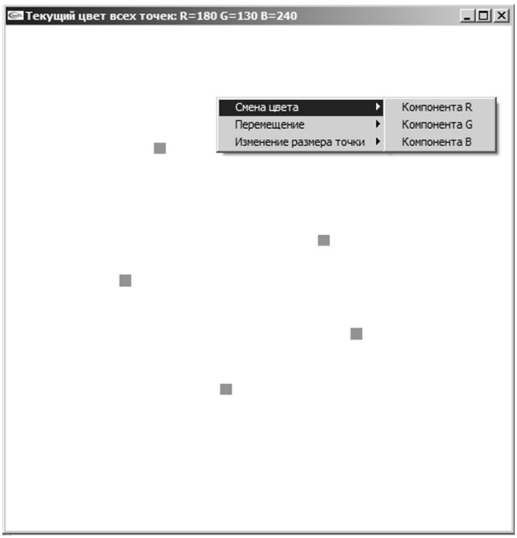


Рис. 45. Пример работы программы «Множество точек»

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Перепишите программу «Множество точек» так, чтобы по введенным точкам отрисовывался примитив, выбор которого осуществляется через меню.

2. Поправьте программу так, чтобы можно было изменить цвет и размер любой точки, а то только всех разом.

3. Поправьте программу так, чтобы можно было перетаскивать и удалять любые точки с помощью мыши.

4. Поправьте программу так, чтобы можно было применять сдвиг, поворот и масштабирование всех точек.

5. Поправьте программу так, чтобы можно было нарисовать несколько примитивов типа «полигон».

6. Поправьте программу так, чтобы в одном окне было 2 подокна, в каждом из которых введенные точки соединялись разными примитивами.

7. Поправьте программу так, чтобы в каждом окне введенные точки соединялись разными примитивами.

8. Перепишите программу «Фрактал Т-квадрат» так, чтобы она запускалась в полноэкранном и игровом режиме с автоматическим определением размеров окна.

9. Добавьте в программу «Фрактал Т-квадрат» цветной режим и выдачу количества итераций и затрачиваемого времени.

10. Получите формулу, позволяющую определить необходимое число итераций для построения фрактала «Т-квадрат» в зависимости от разрешения окна.

ПРИЛОЖЕНИЕ.
ЛИСТИНГ ПРОГРАММЫ «МНОЖЕСТВО ТОЧЕК»

```
#include "glut.h"
#include <vector>
using namespace std;

GLint Width = 512, Height = 512;
GLubyte ColorR = 0, ColorG = 0, ColorB = 0;
GLubyte PointSize = 5;

enum keys { Empty, KeyR,KeyG,KeyB, KeyW,KeyA,KeyS,KeyD, KeyU,KeyI };

struct type_point
{
	GLint x, y;
	type_point(GLint _x, GLint _y) { x = _x; y = _y; }
};
vector <type_point> Points;

/* Функции вывода на экран */
void Render (void)
{
	glColor3ub(ColorR, ColorG, ColorB);
	glPointSize(PointSize);
	glBegin(GL_POINTS);
	for (int i = 0; i<Points.size(); i++)
		glVertex2i(Points[i].x, Points[i].y);
	glEnd();
}

void Display(void)
{
    glClearColor(1,1,1,1);
    glClear(GL_COLOR_BUFFER_BIT);
	Render();
    glFinish();
}
```

```
/* Функция изменения размеров окна */
void Reshape(GLint w, GLint h)
{
	Width = w;	Height = h;
	glViewport(0, 0, w, h);

	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();

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

	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
}


/* Функция обработки сообщений от клавиатуры */
void Keyboard(unsigned char key, int x, int y)
{
	int i, n = Points.size();

	/* Изменение RGB-компонент цвета точек */
	if (key == 'r') ColorR += 5;
	if (key == 'g') ColorG += 5;
	if (key == 'b') ColorB += 5;

	/* Изменение XY-кординат точек */
if (key == 'w') for (i = 0; i<n; i++) Points[i].y += 5;
if (key == 's') for (i = 0; i<n; i++) Points[i].y -= 5;
if (key == 'a') for (i = 0; i<n; i++) Points[i].x -= 5;
if (key == 'd') for (i = 0; i<n; i++) Points[i].x += 5;

	/* Изменение размера точек */
	if (key == 'u') PointSize++;
	if (key == 'i') PointSize--;

	glutPostRedisplay();

	char v[50]; sprintf(v, "Текущий цвет всех точек: R=%.3d
G=%.3d B=%.3d", ColorR, ColorG, ColorB);
	glutSetWindowTitle(v);
}

/* Функция создания всплывающего меню */
void Menu(int pos)
```

```
{
     int key = (keys)pos;

     switch (key)
     {
          case KeyR: Keyboard('r', 0, 0); break;
          case KeyG: Keyboard('g', 0, 0); break;
          case KeyB: Keyboard('b', 0, 0); break;
          case KeyW: Keyboard('w', 0, 0); break;
          case KeyS: Keyboard('s', 0, 0); break;
          case KeyA: Keyboard('a', 0, 0); break;
          case KeyD: Keyboard('d', 0, 0); break;
          case KeyU: Keyboard('u', 0, 0); break;
          case KeyI: Keyboard('i', 0, 0); break;

      default:
           int menu_color = glutCreateMenu(Menu);
           glutAddMenuEntry("Компонента R", KeyR);
           glutAddMenuEntry("Компонента G", KeyG);
           glutAddMenuEntry("Компонента B", KeyB);

           int menu_move = glutCreateMenu(Menu);
           glutAddMenuEntry("Вверх",  KeyW);
           glutAddMenuEntry("Вниз",   KeyS);
           glutAddMenuEntry("Влево",  KeyA);
           glutAddMenuEntry("Вправо", KeyD);

           int menu_size = glutCreateMenu(Menu);
           glutAddMenuEntry("Увеличить", KeyU);
           glutAddMenuEntry("Уменьшить", KeyI);

           int menu = glutCreateMenu(Menu);
           glutAddSubMenu("Смена цвета", menu_color);
           glutAddSubMenu("Перемещение", menu_move);
           glutAddSubMenu("Изменение размера точки", menu_size);

        glutAttachMenu(GLUT_RIGHT_BUTTON);
        Keyboard(Empty,0,0);
     }
}

/* Функция обработки сообщения от мыши */
void Mouse(int button, int state, int x, int y)
```

```
{
	/* клавиша была нажата, но не отпущена */
	if (state != GLUT_DOWN) return;

	/* новая точка по левому клику */
	if (button == GLUT_LEFT_BUTTON)
	{
		type_point p(x, Height - y);
		Points.push_back(p);
	}
	/* удаление последней точки по центральному клику */
	if (button == GLUT_MIDDLE_BUTTON)
	{
		Points.pop_back();
	}

	glutPostRedisplay();
}

/* Головная программа */
void main(int argc, char *argv[])
{
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_RGB);
	glutInitWindowSize(Width, Height);
	glutCreateWindow("Текущий цвет всех точек:");
	Menu(Empty);
	glutDisplayFunc(Display);
	glutReshapeFunc(Reshape);
	glutKeyboardFunc(Keyboard);
	glutMouseFunc(Mouse);

	glutMainLoop();
}
```

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Залогова Л.А.* Компьютерная графика / Л. Залогова. – М., 2005. – 319 с.

2. *Дегтярев В.М.* Инженерная и компьютерная графика : учебник / В.М. Дегтярев, В.П. Затыльникова. – М., 2010. – 238 с.

3. *Порев В.Н.* Компьютерная графика / В. Порев. – СПб., 2005. – 428 с.

4. *Евченко А.И.* OpenGL и DirectX: программирование графики / А.И. Евченко. – СПб., 2006. – 349 с.

5. *Пономаренко С.И.* Пиксел и вектор. Принципы цифровой графики. – СПб., 2002. – 477 с.

6. *Петров М.Н.* Компьютерная графика: учеб. пособие / М.Н. Петров, В.П. Молочков. - СПб., 2002. – 735 с.

7. *Глушаков С.В.* Компьютерная графика: учебный курс / С.В. Глушаков, Г.А. Кнабе. – Харьков, 2001. – 500 с.

8. Гринько М.Е. Компьютерная графика: учебное пособие / М.Е. Гринько [и др.]. – Новосибирск: Изд-во НГТУ, 2009. – 286 с. – Режим доступа: http://elibrary.nstu.ru/source?bib_id=vtls000111616.

9. *Коцюбинский А.О.* Компьютерная графика: практич. пособие. – М., 2001. – 750 с.

СОДЕРЖАНИЕ

Задорожный Александр Геннадьевич
Вагин Денис Владимирович
Кошкина Юлия Игоревна

ВВЕДЕНИЕ В ДВУМЕРНУЮ КОМПЬЮТЕРНУЮ ГРАФИКУ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ OpenGL

Учебное пособие

В авторской редакции

Выпускающий редактор *И.П. Брованова*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *Н.В. Гаврилова*

Налоговая льгота – Общероссийский классификатор продукции
Издание соответствует коду 95 3000 ОК 005-93 (ОКП)

Подписано в печать 26.06.2018. Формат 60 × 84 1/16. Бумага офсетная
Тираж 150 экз. Уч.-изд. л. 6,04. Печ. л. 6,5. Изд. 400/17. Заказ № 946
Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630073, г. Новосибирск, пр. К. Маркса, 20