

Министерство образования и науки Российской Федерации

Калужский филиал
федерального государственного бюджетного образовательного
учреждения высшего образования
**«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»**
(КФ МГТУ им. Н.Э. Баумана)

Ю.С. Белов

ВВЕДЕНИЕ В OPENGL

Методические указания к лабораторной работе
по дисциплине «Компьютерная графика»

Калуга, 2018

УДК 004.62
ББК 32.972.5
Б435

Методические указания составлены в соответствии с учебным планом КФ МГТУ им. Н.Э.Баумана по направлению подготовки 09.03.04 «Программная инженерия» кафедры «Программного обеспечения ЭВМ, информационных технологий и прикладной математики».

Методические указания рассмотрены и одобрены:

- Кафедрой «Программного обеспечения ЭВМ, информационных технологий и прикладной математики» (ФН1-КФ) протокол № 7 от «21» февраля 2018 г.

И.о. зав. кафедрой ФН1-КФ  к.т.н., доцент Ю.Е. Гагарин

- Методической комиссией факультета ФНК протокол № 1 от «08» 02 2018 г.

Председатель методической комиссии факультета ФНК  к.х.н., доцент К.Л. Анфилов

- Методической комиссией КФ МГТУ им.Н.Э. Баумана протокол № 2 от «06» 03 2018 г.

Председатель методической комиссии КФ МГТУ им.Н.Э. Баумана  д.э.н., профессор О.Л. Перерва

Рецензент:  к.т.н., зав. кафедрой ЭИУ2-КФ И.В. Чухраев

Авторы  к.ф.-м.н., доцент кафедры ФН1-КФ Ю.С. Белов
 к.ф.-м.н., доцент кафедры ФН1-КФ С.А. Глебов

Аннотация

Методические указания по выполнению лабораторной работы по курсу «Компьютерная графика» содержат общие сведения о способе описания точек в трехмерном пространстве средствами OpenGL. В методических указаниях приводятся теоретические сведения о координатных пространствах и графических примитивах поддерживаемых OpenGL. Рассмотрен процесс помещения точек в конвейер обработки, отрисовки простейших геометрических форм и решения задач отсечения граней с целью оптимизации вычислений, а также приведен пример работы с фактурами.

Предназначены для студентов 2-го курса бакалавриата КФ МГТУ им. Н.Э.Баумана, обучающихся по направлению подготовки 09.03.04 «Программная инженерия».

© Калужский филиал МГТУ им. Н.Э. Баумана, 2018 г.
© Ю.С. Белов, С.А.Глебов, 2018 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	4
ЦЕЛЬ И ЗАДАЧИ РАБОТЫ, ТРЕБОВАНИЯ К РЕЗУЛЬТАТАМ ЕЕ ВЫПОЛНЕНИЯ	5
ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ	6
ЗАДАЧИ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ	25
ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ	60
ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ	60
ВАРИАНТЫ ЗАДАНИЙ	60
КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ	62
ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ	63
ОСНОВНАЯ ЛИТЕРАТУРА	64
ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА.....	65

ВВЕДЕНИЕ

Настоящие методические указания составлены в соответствии с программой проведения лабораторных работ по курсу «Компьютерная графика» на кафедре «Программное обеспечение ЭВМ, информационные технологии и прикладная математика» факультета фундаментальных наук Калужского филиала МГТУ им. Н.Э. Баумана.

Методические указания, ориентированные на студентов 2-го курса бакалавриата направления подготовки 09.03.04 «Программная инженерия», содержат краткую теоретическую часть, описывающую способы представления проекций средствами программного интерфейса OpenGL, поэтапные примеры создания матрицы цветов и её отображения с учетом системы координат, комментарии и пояснения по вышеназванным этапам, а также задание на лабораторную работу.

Методические указания составлены в расчете на начальное ознакомление студентов с основами работы с программным интерфейсом OpenGL. Для выполнения лабораторной работы студенту необходимо уметь ориентироваться в методах и типах данных OpenGL, уметь работать с библиотекой GLUT и системами проекций, создавать и редактировать графические примитивы, а также адаптировать их в абсолютных и относительных координатах.

Программный интерфейс OpenGL, кратко описанный в методических указаниях, может быть использован при создании моделей использующих конвейер трехмерной графики.

ЦЕЛЬ И ЗАДАЧИ РАБОТЫ, ТРЕБОВАНИЯ К РЕЗУЛЬТАТАМ ЕЕ ВЫПОЛНЕНИЯ

Целью выполнения лабораторной работы является формирование практических навыков по работе с проекционной матрицей средствами OpenGL, а также созданию простейших анимаций графических примитивов и их адаптации в абсолютных и относительных оконных координатах.

Основными задачами выполнения лабораторной работы являются: сформировать представление о методах и секторе решаемых OpenGL задач, изучить основные принципы работы OpenGL, представлять и понимать основные реализации OpenGL, знать типы данных OpenGL и специфику именования переменных, понимать основные принципы трехмерного программирования компьютерной графики, иметь представление о проекциях, уметь создавать типовой проект в различных средах разработки (Visual Studio), иметь представление о двойной буферизации.

Результатами работы являются:

- Выполненные преобразования проекций средствами OpenGL
- Реализованная согласно варианту анимация графических примитивов
- Соблюдение пропорций и размеров объектов
- Подготовленный отчет

ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ

Определение OpenGL

Строго OpenGL определяется как *программный интерфейс к графической аппаратуре*. По сути же, это очень мобильная и очень эффективная библиотека трехмерной графики и моделирования. Используя OpenGL, вы можете создавать элегантную и прекрасную трехмерную графику практически с таким же визуальным качеством, что дает программа построения хода лучей. Самое большое преимущество использования OpenGL заключается в том, что скорость обработки здесь на порядки выше, чем у программ построения хода лучей. Первоначально в OpenGL использовались алгоритмы, тщательно разработанные и оптимизированные компанией Silicon Graphics, Inc. (SGI), признанного мирового лидера в сфере компьютерной графики и анимации. Со временем OpenGL эволюционировал; свой опыт и интеллектуальную собственность в него вложили многие производители, разработав собственные высокопроизводительные реализации.

OpenGL — это не язык программирования, как С или С++. Строго говоря, "программ OpenGL" не существует, правильно говорить о программах, которые пишутся с учетом того, что в качестве одного из их программных интерфейсов приложения (Application Programming Interfaces — API) будет использован OpenGL. OpenGL предназначен для использования с аппаратным обеспечением компьютера, которое разработано и оптимизировано под отображение трехмерной графики и манипуляцию ею. Возможны также только программные, "общие" реализации OpenGL, и к этой категории относятся реализации, выполненные Microsoft. С помощью только программной реализации визуализацию невозможно выполнить так же быстро, как с помощью аппаратной, а некоторые нетривиальные спецэффекты вообще могут быть недоступны. В то же время применение программной реализации означает, что программа теоретически сможет запускаться на множестве разнообразных компьютерных систем, в которых могут отсутствовать графические карты 3D.

OpenGL используется для решения множества задач, возникающих в различных сферах — от архитектурных и инженерных приложений до программ моделирования, используемых для создания компьютерных монстров в фильмах со спецэффектами. С распространением аппаратного ускорения и быстродействующих микропроцессоров для персональных компьютеров трехмерная графика стала типичной составляющей пользовательских и коммерческих приложений, а не только игр и научных приложений, как было раньше.

Сейчас OpenGL является предпочтительным программным интерфейсом для широкого диапазона приложений и аппаратных платформ. Это поставило OpenGL в выгодную позицию с точки зрения преимуществ будущих изобретений в сфере компьютерной графики. С добавлением к OpenGL языка затенения OpenGL продемонстрировал свою приспособляемость к требованиям развивающегося конвейера программирования трехмерной графики. Наконец, OpenGL — это спецификация, продемонстрировавшая, что ее можно применять к широкому диапазону парадигм программирования. В создании игр для ПК с использованием OpenGL сейчас применяются языки от C/C++ до Java и Visual Basic, и даже такие новые языки, как C#. OpenGL прочно вошел в наш мир.

Принцип работы OpenGL

OpenGL — скорее процедурный, чем описательный графический программный интерфейс приложений. Вместо того чтобы описывать сцену и то, как она должна выглядеть, программист прописывает шаги, необходимые для получения определенного внешнего вида или эффекта. Эти "шаги" включают вызовы многих команд OpenGL. Команды, используются для рисования таких графических примитивов, как точки, линии и многоугольники в трех измерениях. Кроме того, OpenGL поддерживает освещение и затенение, наложение текстуры, смещение, прозрачность, анимацию и многие другие специальные эффекты и возможности.

OpenGL не включает никаких функций управления окнами, взаимодействия с пользователем или ввода в файл/вывода из файла. Каж-

дая среда хоста (такая, как Microsoft Windows) имеет собственные функции для этой цели и отвечает за реализацию средств передачи OpenGL управления рисованием в окне

Не существует такого понятия, как "файловый формат OpenGL" для моделей или виртуальных сред. Программисты создают среды, подходящие для собственных нужд, а затем аккуратно программируют их, используя низкоуровневые команды OpenGL.

Реализации OpenGL

На рис. 1. показано типичное место OpenGL и общей реализации в запущенном приложении. Типичная программа вызывает множество функций, часть которых создает программист, а другие предоставляются операционной системой или библиотекой времени выполнения языка программирования. Приложения Windows, ожидающие вывода на экран, обычно вызывают программный интерфейс Windows, именуемый *GDI* (Graphics Device Interface — интерфейс графических устройств). *GDI* содержит методы, позволяющие писать текст в окне, рисовать простые двухмерные линии и т.д.

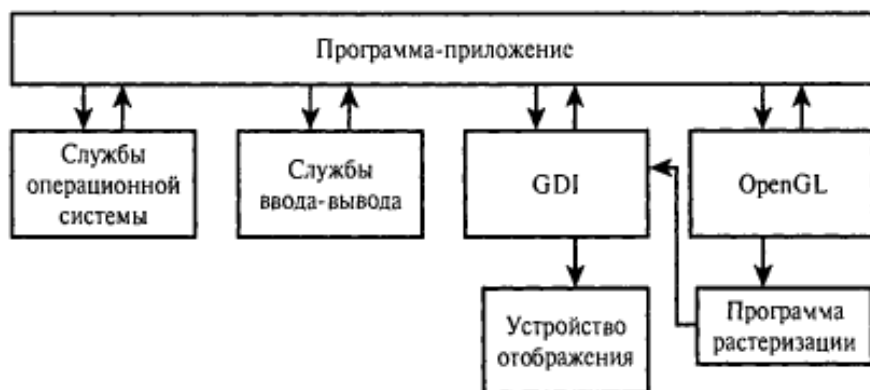


Рис.1. – Место OpenGL в типичной программе-приложении

Обычно производители графических карт поставляют драйвер аппаратного устройства, с которым интерфейсы *GDI* выводят изображение на монитор. Программная реализация OpenGL получает графиче-

ские запросы от приложений и строит (растеризует) цветное изображение трехмерной графики. Затем она передает это изображение GDI для отображения на монитор. В другой операционной системе описанный процесс по сути такой же, но вместо GDI применяются присущие этой операционной системе службы дисплея.

Аппаратные реализации

Аппаратная реализация OpenGL обычно имеет вид драйвера графической карты. На рис. 2 показана ее связь с приложением (подобно тому, как на рис. 1 иллюстрируется программная реализация). Обратите внимание на то, что вызовы программного интерфейса OpenGL передаются драйверу аппаратного устройства. Этот драйвер не передает свой выход GDI Windows с целью вывода на экран, драйвер сопрягается непосредственно с аппаратным обеспечением графического дисплея.



Рис. 2. – Место OpenGL с аппаратным ускорением в типичной программе – приложении

Аппаратная реализация часто называется *ускоренной реализацией*, поскольку трехмерная графика с аппаратной поддержкой обычно существенно превосходит по характеристикам только программные реализации. На рис. 2 не показано, что иногда часть функциональных возможностей OpenGL реализуется в программном обеспечении как

часть драйвера, и что другие особенности и функциональные возможности могут непосредственно передаваться аппаратному обеспечению. Данная идея приводит нас к следующей теме — конвейеру OpenGL.

Конвейер

Слово конвейер используется для описания процесса, который может захватывать несколько отдельных этапов или каскадов. На рис. 3 показана упрощенная версия конвейера OpenGL. Когда приложение инициирует вызов функции интерфейса OpenGL, команды помещаются в буфер команд. Со временем этот буфер заполняется командами, данными о вершинах, текстурах и т.д. Когда буфер заполняется (или программно или согласно структуре драйвера), команды и данные передаются на следующий каскад конвейера.

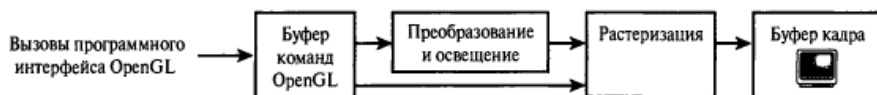


Рис. 3. — Упрощенная версия конвейера OpenGL

Данные о вершинах обычно преобразуются и изначально освещаются. Расчет освещения выполняется для того, чтобы указать, насколько яркими должны быть цвета в каждой вершине.

Когда этот этап завершен, данные подаются на каскад растеризации конвейера. На этом этапе по геометрическим, цветным и текстурным данным создается цветное изображение. Затем изображение помещается в *буфер кадров*. Буфер кадров — это память устройства графического вывода, посредством которой изображение отображается на экране.

На диаграмме конвейер OpenGL представлен упрощенно, но на данный момент этого достаточно, чтобы вы поняли концепцию визуализации трехмерной графики. На высоком уровне данное представление точное, но на низком уровне внутри каждого изображенного прямоугольника имеется множество других составляющих. Кроме того,

имеется несколько исключений, например стрелка на рисунке, указывающая, что некоторые команды вообще пропускают этап преобразования и освещения (например, это отображение на экране необработанных данных изображения).

Ранние аппаратные ускорители OpenGL были не более чем устройствами быстрой растеризации. Они ускоряли только участок растеризации конвейера. Центральный процессор выполнял преобразование и освещение в программной реализации этого фрагмента конвейера. В более мощных (т.е. дорогих) ускорителях преобразование и освещение встраивалось в графический ускоритель. В такой схеме выполнение большей части конвейера OpenGL возлагалось на аппаратное обеспечение, что гарантировало более высокую производительность. В настоящее время даже маломощное потребительское аппаратное обеспечение имеет каскад преобразования и освещения на аппаратном уровне. Суммарный эффект этой установки заключается в том, что теперь возможны модели с большей детализацией и создание более сложной графики в реальном времени на недорогом потребительском аппаратном обеспечении. Разработчики игр и приложений могут усиливать этот эффект, предлагая более детальные и визуально богатые окружения.

Большей частью OpenGL не является языком программирования; это программный интерфейс приложения (Application Programming Interface — API). Когда мы говорим "программа основана на OpenGL" или "приложение OpenGL", то подразумеваем, что она была написана на каком-то языке программирования (например, на C или C++) с вызовами, обращенными к одной или нескольким библиотекам OpenGL.

Библиотеки и заголовки

Хотя OpenGL является "стандартной" программной библиотекой, она имеет много реализаций Microsoft Windows поставляется с поддержкой OpenGL как средства программной визуализации. Это означает, что, когда программа, написанная с использованием OpenGL, вызывает функции OpenGL, реализация Microsoft выполняет функции трехмерной визуализации, и вы видите результаты в окне приложе-

ния. Реальная программная реализация Microsoft находится в динамически подключаемой библиотеке `opengl32.dll`, расположенной в системном каталоге Windows. На большинстве платформ библиотека OpenGL сопровождается библиотекой GLU (OpenGL Utility — набор программ OpenGL), которая в Windows располагается в файле `glu32.dll` также в системном каталоге. Эта библиотека является набором служебных функций, которые выполняют распространенные (но иногда сложные) задачи, например специальные матричные операции, или предоставляют поддержку распространенных типов кривых и поверхностей.

Прототипы всех функций, типов и макросов OpenGL содержатся (по договоренности) в файле заголовка `gl.h`. С этим файлом поставляются инструменты программирования Microsoft, а также большинство других сред программирования для Windows или других платформ (по крайней мере те, что исконно поддерживают OpenGL). Прототипы функций библиотеки GLU содержатся в файле `glu.h`. Оба указанных файла обычно расположены в специальной папке, прописанной в команде `include`. Например, в следующем коде представлен типичный исходный заголовок, включаемый в типичную программу Windows, в которой используется OpenGL.

```
#include<windows.h>
#include<gl/gl.h>
#include<gl/glu.h>
```

Типы данных OpenGL

Чтобы облегчить переносимость кода OpenGL с одной платформы на другую, в OpenGL определены собственные типы данных. Эти типы данных легко отобразить в нормальные типы данных C, с которыми вы можете работать. Однако различные компиляторы и среды имеют собственные правила, касающиеся размера и схем распределения памяти для различных переменных C. Используя определенные OpenGL типы переменных, вы отделяете код от подобных изменений.

В табл. 1 перечислены типы данных OpenGL, соответствующие им типы данных C в 32-битовых средах Windows (Win32) и подходящие суффиксы для литералов. Позже вы увидите, что эти суффиксы применяются во многих именах функций OpenGL.

Таблица 1. Типы переменных OpenGL и соответствующие типы данных C

Тип данных OpenGL	Внутреннее представление	Определение в форме типа C	Суффикс литералов C
GLbyte	8-битовое целое	signed char	b
GLshort	16-битовое целое	short	s
GLint, GLsizei	32-битовое целое	long	l
GLfloat, GLclampf	32-битовое с плавающей	float	f
	запятой		
GLdouble, GLclampd	64-битовое с плавающей	double	d
	запятой		
GLubyte, GLboolean	8-битовое целое без знака	unsigned char	ub
GLushort	16-битовое целое без знака	unsigned short	us
GLuint, GLenum,	32-битовое целое без знака	unsigned long	ul
GLbitfield			

Все типы данных начинаются с GL, что обозначает "OpenGL". После большинства из них указываются соответствующие типы данных C (`byte`, `short`, `int`, `float` и т.д.) Перед некоторыми имеется `u`, что указывает тип данных без знака. Например, `ubyte` обозначает тип `byte` без знака. В некоторых случаях приведено более описательное имя, например `size`, указывающее значение длины или глубины. Например, `GLsizei` — это переменная OpenGL, обозначающая параметр размера, который представляется целым числом. Обозначение `clamp` является подсказкой; предполагается, что значение будет ограничено согласно допустимому диапазону 0,0-1,0 ("зажато", `clamped`). Переменные типа `GLboolean` используются для обозначения условий `true` ("истина") и `false` ("ложь"), `GLenum` — для перечислимых переменных, а `GLbitfield` — для переменных, содержащих поля двоичных разрядов.

Указатели и массивы не имеют специальных обозначений. Массив из 10 переменных типа `GLshort` объявляется просто как `GLshort shorts[10];`

Массив из 10 указателей на переменные типа `GLdouble` определяется следующим образом `GLdouble *doubles[10];`

Несколько других типов указателей используются для сплайнов NURBS и поверхностей второго порядка.

Правила именования

Для большинства функций OpenGL используются правила именования, позволяющие сообщить, из какой библиотеки пришла функция, сколько она принимает аргументов и какого типа. Все функции имеют корень, представляющий соответствующую функции команду OpenGL. Например, `glColor3f` имеет корень `Color`. Префикс `gl` представляет библиотеку `gl`, а суффикс `3f` означает, что функция принимает три аргумента с плавающими запятыми. Все функции OpenGL имеют следующий формат.

<Префикс библиотеки><Команда основной библиотеки><Необязательный счетчик аргументов><Необязательный тип аргументов>



Рис. 4. – Разбор функции OpenGL

Элементы имени функции OpenGL иллюстрируются на рис. 4. Эта простая функция с суффиксом `3f` принимает три аргумента с плавающей запятой. Другие разновидности этой функции принимают три целых числа (`glColor3i`), три числа двойной точности (`glColor3d`) и т.д. Такое добавление числа и типа аргументов (см. табл.1) к концу функций OpenGL облегчает запоминание списка аргументов. Некоторые версии `glColor` принимают четыре аргумента, дополнительно задавая компонент альфа (прозрачность).

Многие компиляторы C/C++ для Windows предполагают, что любое литеральное значение с плавающей запятой относится к типу `double`, если с помощью суффикса явно не указано иное. Когда литералы используются для представления аргументов с плавающей запятой, а вы не указали, что эти аргументы относятся к типу `float`, а не `double`, компилятор выдаст предупреждение при обработке, обнаружив, что вы передаете величину двойной точности функции, которая по определению принимает только величины с плавающей запятой. Отметим, что это может привести к снижению точности. По мере того как растет программа OpenGL, число предупреждений быстро станет измеряться сотнями, и среди них будет трудно заметить синтаксические ошибки.

Кроме того, можно поддаться соблазну использовать функции, принимающие аргументы двойной точности с плавающей запятой, вместо того, чтобы возиться с указанием литералов как величин типа `float`. Однако во внутреннем представлении OpenGL использует именно величины с плавающей запятой, а использование всего, что отличается от функций обычной точности с плавающей запятой, снижает про-

изводительность, поскольку в процессе обработки OpenGL значения преобразуются в тип `float`. Кроме того, любая величина двойной точности требует вдвое больше памяти, чем величина обычной точности. Для программы, в которой "плавает" очень много чисел, такой удар по производительности может быть очень ощутимым!

Использование GLUT

Вначале был AUX — дополнительная библиотека OpenGL (OpenGL auxiliary library). Библиотека AUX была создана для содействия обучению и такому написанию программ OpenGL, чтобы программист не занимался деталями конкретной среды — будь то UNIX, Windows или что-либо еще. Используя AUX, вы не сможете написать "окончательный" код; скорее он подходит для подготовки фундамента для проверки ваших идей. Нехватка базовых функций GUI ограничивает использование библиотеки для создания полезных приложений.

Вскоре на смену AUX пришла библиотека GLUT, предназначенная для межплатформенных примеров программирования и демонстраций. GLUT — это сокращение от OpenGL Utility Toolkit (набор инструментов OpenGL, не путайте с GLU — библиотекой инструментов OpenGL). Марк Килгард (Mark Kilgard), работавший в SGI, написал GLUT как более искусную замену библиотеки AUX и включил в нее несколько элементов графического пользовательского интерфейса, по крайней мере для того, чтобы сделать программы-примеры более удобными в системе X Windows. Эта замена включала использование всплывающих меню, управление другими окнами и даже обеспечение поддержки джойстика. GLUT не является публичным достоянием, но она бесплатна и бесплатно ее распространение.

Трёхмерное пространство. Системы координат

Рассмотрим, как описываются объекты в трех измерениях. Прежде чем вы сможете задавать положение и размер объекта, потребуется система отсчета, относительно которой можно измерять положения тел. При рисовании линий или расстановке точек на простом экране компьютера вы задаете положение через строки и столбцы. Например, стандартный экран VGA имеет 640 позиций пикселей слева направо и 480 позиций

сверху вниз. Чтобы задать точку в центре экрана, вы указываете, что ее нужно поместить в позицию (320, 240) — т.е. на 320 пикселей от левого края экрана вправо и на 240 пикселей от верхнего края экрана вниз.

В OpenGL (да и практически в любом программном интерфейсе приложения трехмерной графики) при создании окна, в котором вы будете рисовать, необходимо задать систему координат, которую вы желаете использовать; следует также указать, как заданные координаты будут отображаться в физические пиксели экрана. Рассмотрим вначале, как это выглядит для двумерных изображений, а затем расширим сформулированные принципы на три измерения.

Двухмерные декартовы координаты

Для представления двумерных рисунков чаще всего используется декартова система координат. Декартовы координаты представляются координатой x — положением в горизонтальном направлении и y — положением в вертикальном направлении.

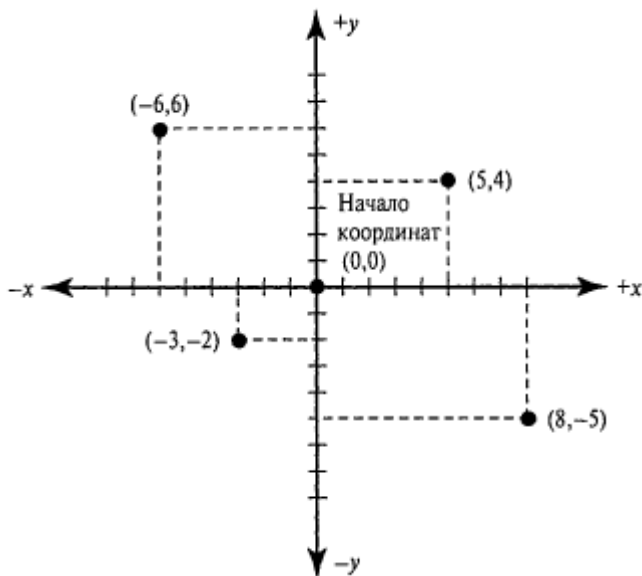


Рис. 5. — Декартова плоскость

Начало декартовой системы координат находится в точке $x = 0$, $y = 0$. Декартовы координаты записываются как пары значений в круглых скоб-

ках; вначале указывается координата x , затем координата y . Например, начало координат записывается как $(0,0)$. На рис. 5 представлена декартова система координат в двух измерениях. Снабженные метками линии x и y называются осями, и они могут следовать от минус до плюс бесконечности.

Оси x и y перпендикулярны и вместе определяют плоскость xy . В любой системе координат две оси, которые пересекаются под прямым углом, определяют плоскость. В системе, где существует всего две оси, естественным образом можно изобразить только одну плоскость.

Отсечение координат

Физически окно измеряется в пикселях. Прежде чем вы сможете начать изображать в окне точки, линии и формы, вы должны сообщить OpenGL, как переводить указанные пары значений в экранные координаты. Для этого вы задаете область декартова пространства, которую занимает окно; она называется *областью отсечения*.

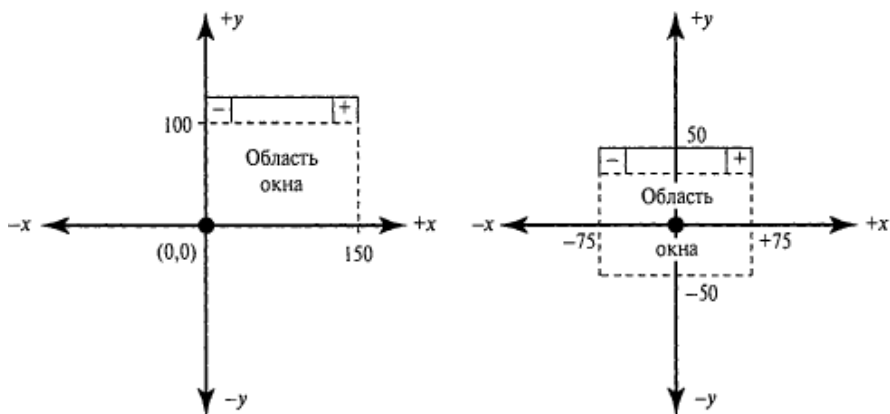


Рис. 6 – Две области отсечения

В двумерном пространстве область отсечения — это минимальное и максимальное значения x и y , которые принадлежат окну. Мы можем описать это и по-другому, задав положение начала системы координат относительно окна. Два распространенных выбора областей отсечения показаны на рис. 6.

В первом примере на рис. 6 (слева) координаты x в окне меняются слева направо от 0 до +150, а координаты y — снизу вверх от 0 до +100. Точка в центре экрана будет представлена как (75, 50). На втором рисунке показана область отсечения, координата x которой меняется слева направо от -75 до +75, а координата y — снизу вверх от - 50 до +50. В этом примере точка в центре экрана будет совпадать с началом координат — точкой (0,0). Кроме того, используя функции OpenGL (или обычные функции Windows для рисования с помощью GDI), можно перевернуть систему координат сверху вниз или справа налево. Отображению по умолчанию в окнах Windows соответствует положительное направление оси y сверху вниз. Хотя такая организация и удобна при рисовании сверху вниз, для рисования графики и графиков отображение по умолчанию непривычно.

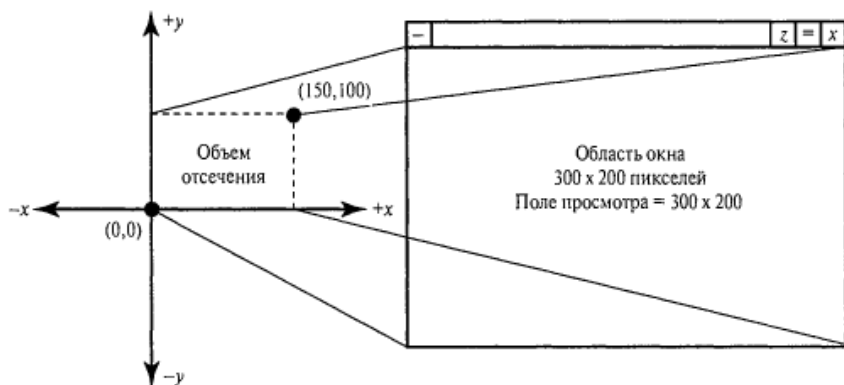


Рис. 7. — Поле просмотра, определенное как удвоенная область отсечения

Поля просмотра: отображение координат рисунка в координаты окна

Очень редко ширина и высота области отсечения точно совпадает с шириной и высотой окна в пикселях. Следовательно, нужно преобразовать систему координат: отобразить логические декартовы координаты в физические координаты пикселей экрана. Такое отображение задается с помощью *поля просмотра*. Поле просмотра — это область внутри клиен-

та окна, которая используется для рисования области отсечения. Поле просмотра просто отображает область отсечения в область окна. Обычно поле просмотра определяется как все окно, но это не является строго необходимым, например, вы можете указать, что рисовать разрешается только в нижней половине окна.

На рис 7 показано большое окно, насчитывающее 300×200 пикселей, с полем просмотра, определенным как вся область клиента. Если установить, что область отсечения этого окна простирается от 0 до 150 вдоль оси x и от 0 до 100 вдоль оси y , логические координаты будут отображены в большую систему экранных координат в окне наблюдения. Каждый элемент в логической системе координат будет удвоен в физической системе координат (пиксели) окна.

На рис. 8 показано поле просмотра, равное области отсечения. Окно наблюдения по-прежнему равно 300×200 пикселей, поэтому область наблюдения занимает левую нижнюю область окна.

С помощью поля просмотра можно сжимать или растягивать изображения внутри окна, а также отображать только часть области отсечения (в таком случае поле просмотра задается большим областью клиента окна).

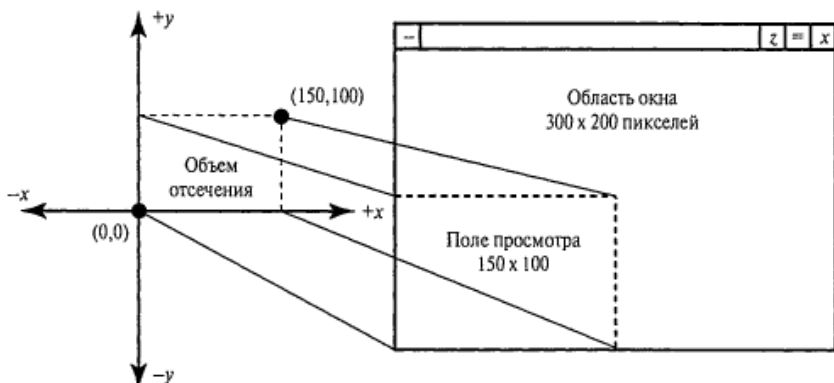


Рис. 8 – Поле просмотра, определенное с размерами области отсечения

Вершина — точка в пространстве

При рисовании двух- или трехмерного объекта вы составляете его из нескольких меньших форм, именуемых *примитивами*. Примитивы — это такие одно- или двухмерные объекты или поверхности, как точки, линии

и многоугольники (плоские формы с несколькими сторонами), собираемые в трехмерном пространстве для создания трехмерных объектов. Например, трехмерный куб состоит из шести двумерных квадратов, размещенных в разных плоскостях. Каждый угол квадрата (или любого примитива) называется *вершиной*. Этим вершинам сопоставляются координаты в трехмерном пространстве. Вершина — это всего лишь набор координат в двух- или трехмерном пространстве.

Трехмерные декартовы координаты

Расширим теперь двумерную систему координат в третье измерение, добавив компоненту глубины. На рис. 9 показана декартова система координат с новой осью z , перпендикулярной как оси x , так и оси y . Новая ось представляет линию, проведенную перпендикулярно от центра экрана к наблюдателю. Теперь можно задавать точку в трехмерном пространстве с помощью трех координат: x , y и z . На рис. 9 для конкретности показана точка $(-4, 4, 4)$.

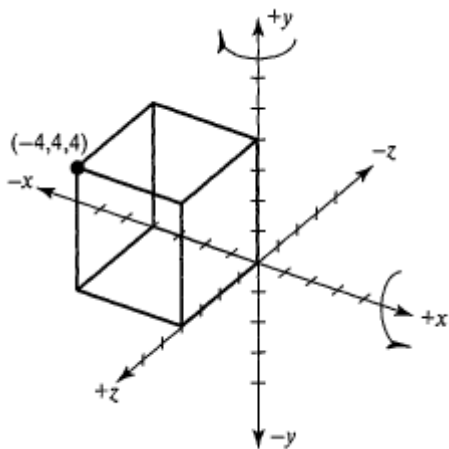


Рис. 9 – Декартовы координаты в трех измерениях

Проектирование: получение 2D из 3D

Первая концепция, которую вы действительно должны понять, называется проектированием. Трехмерные координаты, с помощью которых вы создаете геометрию, опускаются (или *проектируются*) на двумер-

ную плоскость (фон окна). Посмотрите на рис. 10, где контуры дома, находящегося на заднем плане, переносятся на плоскую поверхность стекла. Задавая проекцию, вы задаете *наблюдаемый объем*, который вы желаете отобразить в своем окне, и то, как он должен преобразовываться.

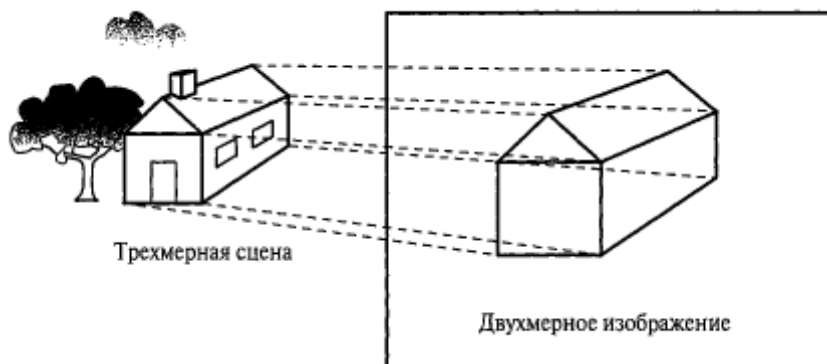


Рис. 10 – Трехмерное изображение, спроектированное на двухмерную поверхность

Ортографические проекции

В OpenGL мы будем работать с двумя основными типами проекций. Первый называется *ортографической* или *параллельной проекцией*. Для использования такой проекции вы задаете квадратный или прямоугольный наблюдаемый объем. Все, что находится вне этого объема, не изображается. Более того, все объекты, которые имеют одинаковые размеры, будут при показе иметь одинаковую величину вне зависимости от того, близко или далеко они расположены. Проекция такого типа (одна из них показана на рис. 11) чаще всего используются в архитектурном дизайне, автоматизированном проектировании или двухмерных графах. Довольно часто с помощью ортографической проекции добавляется текст или двухмерные рисунки поверх трехмерной графической сцены.

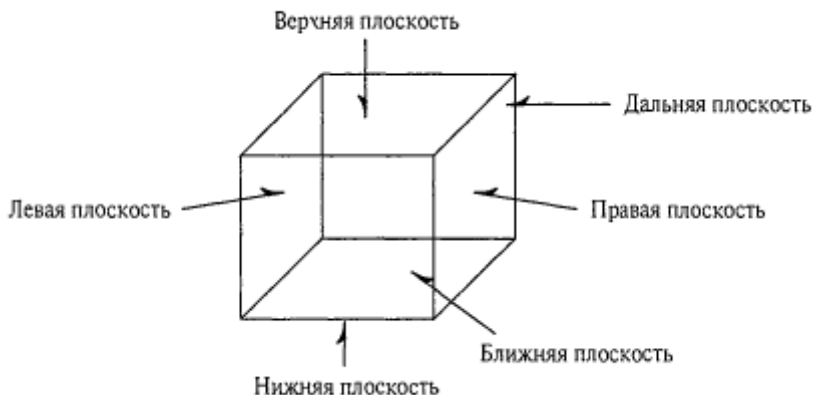


Рис. 11 – Объем отсечения ортографической проекции

Наблюдаемый объем в ортографической проекции задается через ближнюю, дальнюю, левую, правую, верхнюю и нижнюю отсекающие плоскости. Объекты и рисунки, которые вы помещаете внутрь этого наблюдаемого объема, проектируются (с учетом ориентации) на двухмерное изображение, которое вы видите на экране.

Перспективные проекции

Второй из наиболее распространенных проекций является перспективная. Данная проекция добавляет эффект уменьшения удаленных объектов. Наблюдаемый объем (рис. 12) напоминает пирамиду с обрезанной верхушкой. Данная форма называется усеченной пирамидой. Объекты, ближайшие к передней грани наблюдаемого объема, ближе к истинному размеру, а объекты, расположенные вблизи дальней грани сжимаются при проектировании на переднюю грань объема. Проекции подобного типа являются наиболее реалистичными в моделировании и трехмерной анимации.

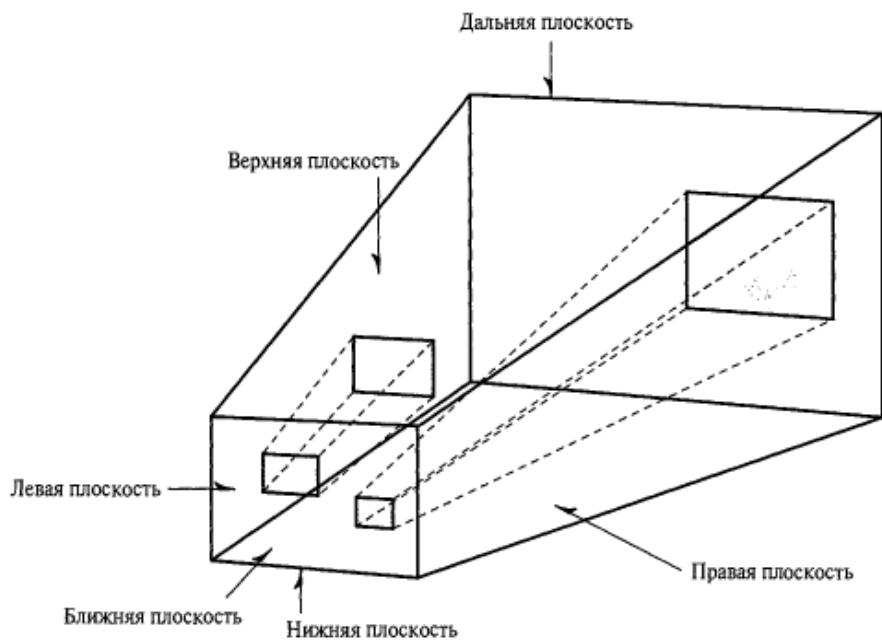


Рис. 12 – Объем отсечения перспективной проекции (усеченная пирамида)

ЗАДАЧИ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Создание типового проекта Microsoft Visual Studio

Для создания и настройки типичного проекта выполните следующие шаги.

Шаг 1: Создание проекта.

Запустите Microsoft Visual Studio 8.0. В меню выберите File->New->Project (Рис. 13)

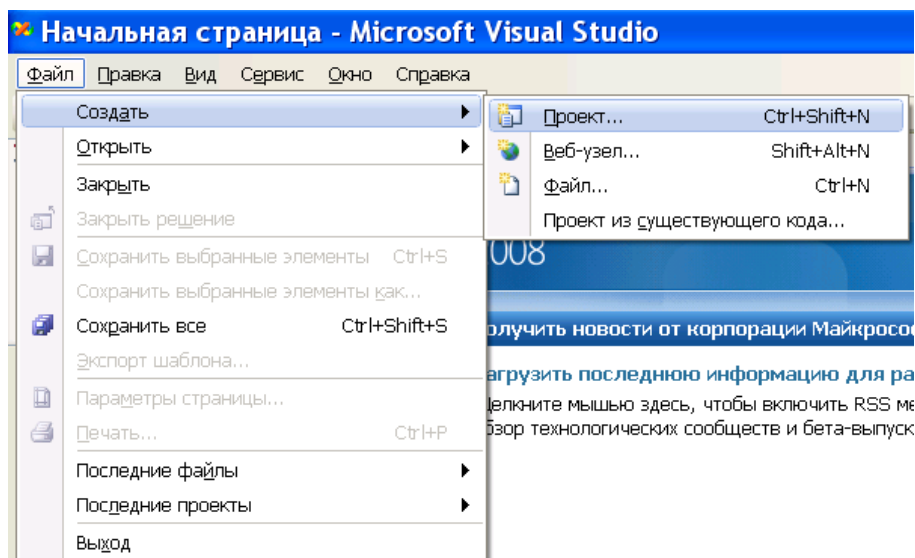


Рис. 13 – Создание проекта в Microsoft Visual Studio

В диалоговом окне выберите слева тип проекта Visual C++ → Win32. Справа в шаблонах выберите Win32 Console Application. Снизу определите расположение проекта (Location), имя проекта (Name) и имя решения (Solution Name). (Рис. 14) Обратите внимание, что каждое решение может содержать в себе неограниченное число проектов. Эти проекты можно добавить в проект позднее. Каждому проекту внутри решения предоставляется своя подпапка с именем проекта. Конечный путь к файлам проекта будет выглядеть так: Location\Solution Name\Name (Рис. 14)

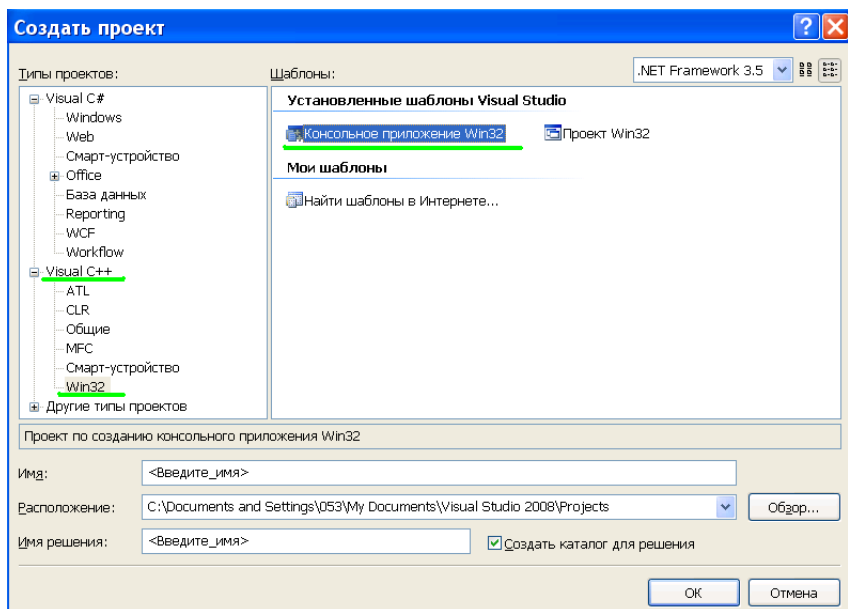


Рис. 14 – Основные характеристики проекта в Microsoft Visual Studio

В новом диалоговом окне перейдите на вкладку Application Settings, где поставьте галочку напротив Empty Project и нажмите Finish. (Рис. 15)

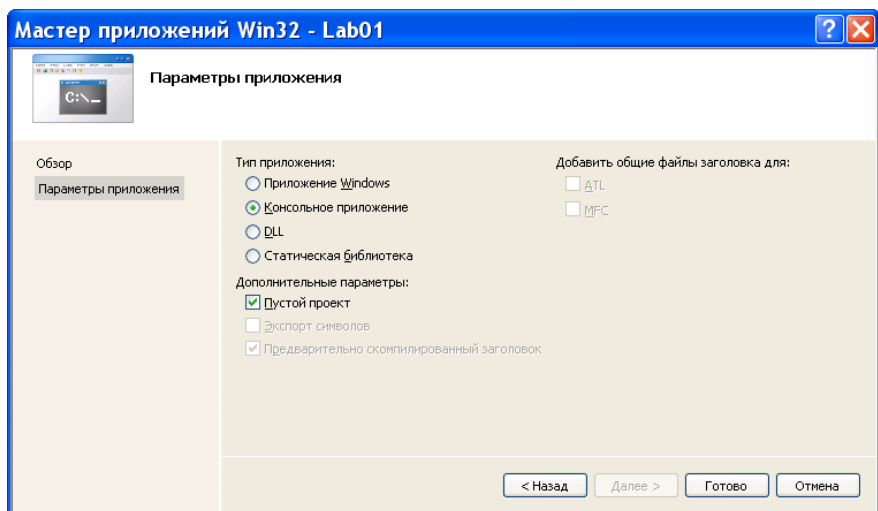


Рис. 15 – Определение типа проекта в Microsoft Visual Studio

Шаг 2: Подключение библиотек

В левой части откройте вкладку Solution Explorer. Здесь находится решение и в нем список всех проектов. Для того чтобы добавить новый проект в решение кликните правой кнопкой мыши на имени решения и в выпадающем меню выберите Add->New Project. (Рис. 16)

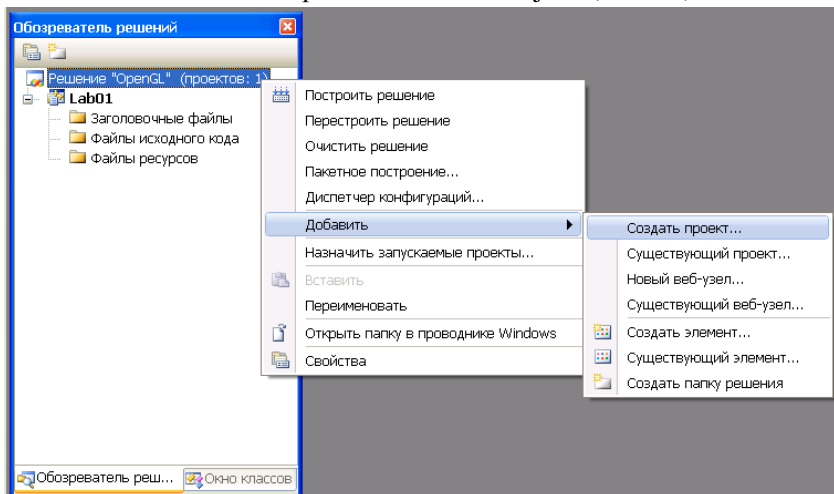


Рис. 16 – Основные характеристики проекта в Microsoft Visual Studio

Необходимо подключить к проекту библиотеки. Для этого для начала скопируйте из папки с заданием файлы `glew32.lib` и `glut32.lib` (именно `lib`, а не `dll`) в папку с проектом (там же должен располагаться файл с именем проекта и расширением `vcproj`). (Рис. 17)

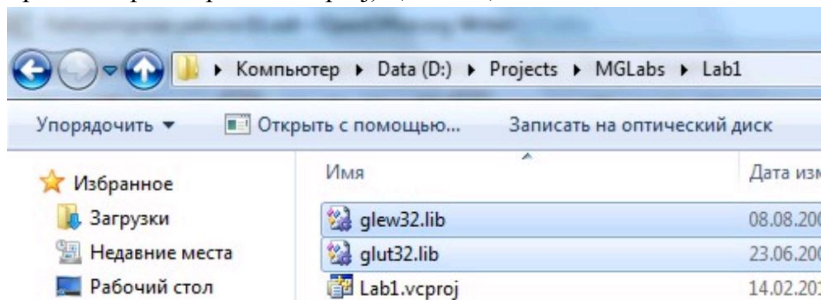


Рис. 17 – Подключение файлов библиотек

В Visual Studio необходимо открыть слева вкладку Solution Explorer. Здесь клик правой кнопкой по имени проекта (проекта, а не решения!) и выбор в самом низу пункта Properties. (Рис. 18)

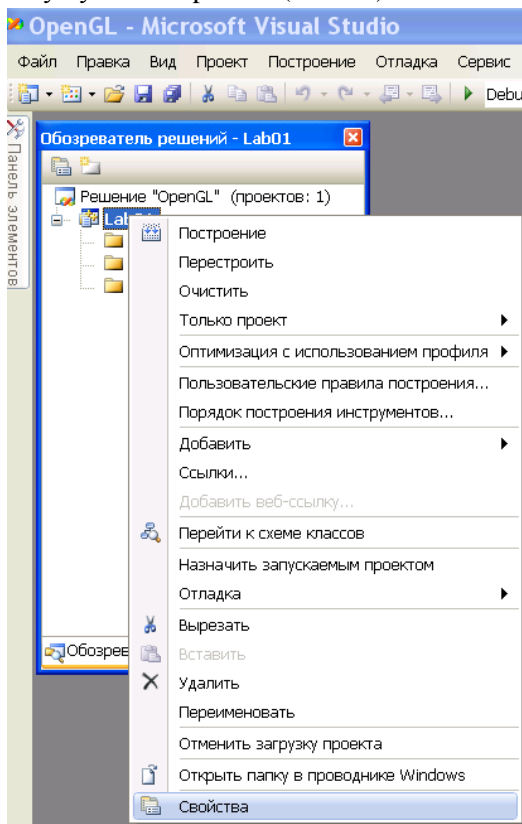


Рис. 18 – Настройка свойств проекта в Microsoft Visual Studio

Это диалоговое окно с настройками проекта. Здесь можно задать абсолютно все настройки, которые касаются конкретного проекта, начиная от настроек компилятора и компоновщика и кончая сервисными функциями по копированию файлов после сборки.

Сейчас надо настроить компоновщик, чтобы при сборке программы он присоединил к исполняемому файлу две скопированные библиотеки. Для этого слева выберите Configuration Properties → Linker → Input. (Рис. 19)

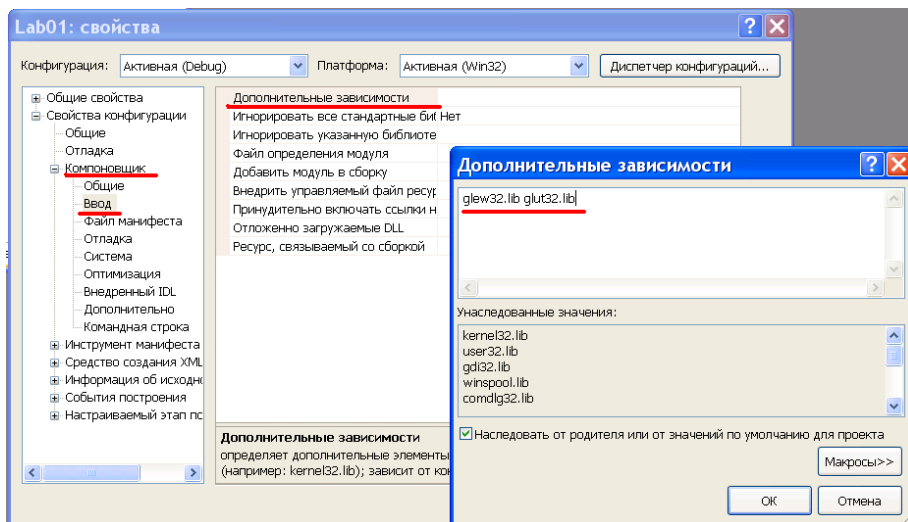


Рис. 19 – Подключение зависимостей

Теперь справа в строчке Additional Dependencies перечислите через пробел имена lib-файлов библиотек: `glew32.lib` `glut32.lib`. Нажмите ОК.

Порядок имен не имеет значения. Если в будущем понадобится подключить библиотеки к проекту — делайте это здесь же.

Шаг 3: Создание главного модуля и инициализация GLUT

Для начала необходимо скопировать заголовочные файлы из папки с заданием библиотек в папку с проектом (Рис. 20) (туда же, куда скопировали и lib-файлы).

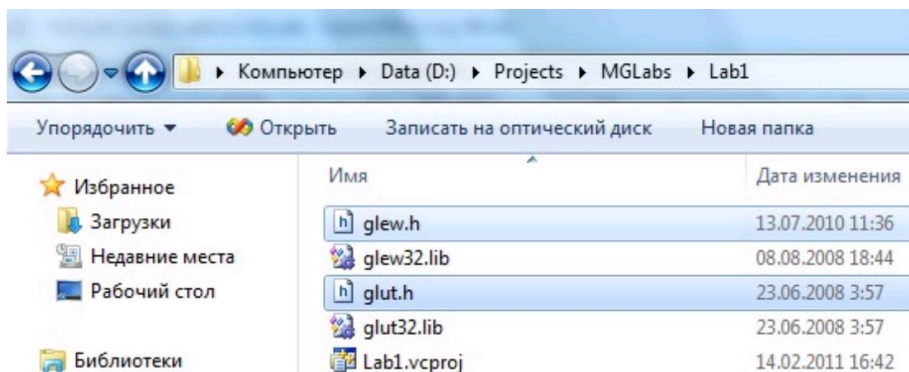


Рис. 20 – Добавление заголовочных файлов

Теперь необходимо создать главный модуль и подключить заголовочные файлы. Для этого кликните правой кнопкой мыши по имени проекта (проекта, а не решения!) и выберите Add->New Item. (Рис. 21)

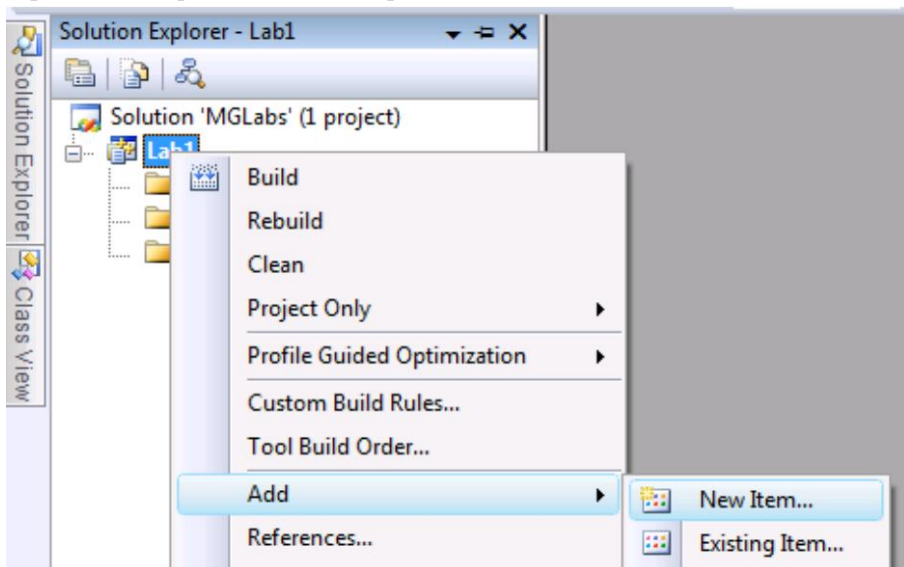


Рис. 21 – Добавление файлов в проект

Откроется новое окно, с помощью которого можно добавлять к проекту новые элементы. Далее необходимо выбрать категорию Visual C++ →

Code, справа в шаблонах C++ File (.cpp), а снизу указать имя файла, например main.cpp и нажать OK. (Рис. 22)

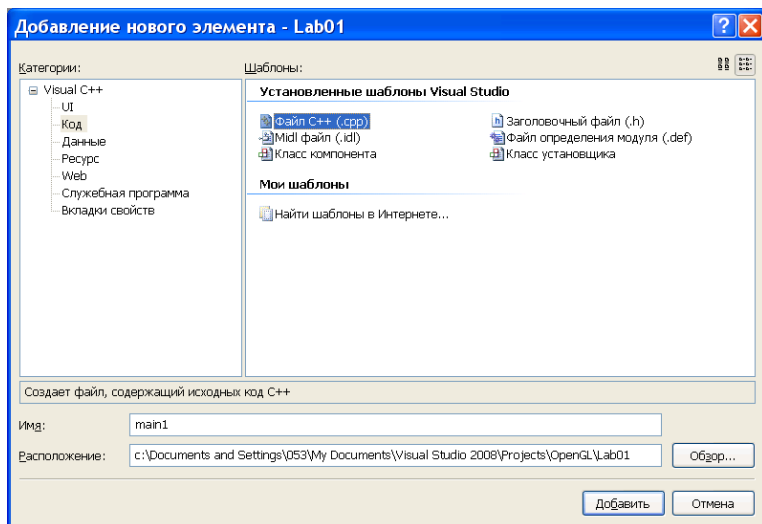


Рис. 22 – Создание файлов исходного кода

Созданный файл можно будет найти в Solution Explorer в проекте в папке Source Files. Открыть его можно будет двойным кликом.

Далее требуется создать шаблон программы, где будет инициализирован [GLUT](#), и создано окно по умолчанию. Вставьте в main.cpp код ниже.

Листинг 1 – Простейшее окно GLUT

```
// подключаем заголовочные файлы библиотек
#include "glew.h"
#include "glut.h"
// создаем обработчик отрисовки
void display()
{
    // переключаем буферы экрана
    glutSwapBuffers ();
}
```

```
// описываем главную функцию. рекомендуется использо-
вать
// именно такой формат ее заголовка
void main ( int argc, char* argv[] )
{

    // инициализируем экран в режиме RGBA и двойным
    буфером
    glutInitDisplayMode ( GLUT_RGBA|GLUT_DOUBLE );
    // создаем окно по умолчанию
    glutCreateWindow ( "Test Window");
    // устанавливаем обработчик перерисовки окна
    glutDisplayFunc ( display);
    // запускаем основной цикл окна
    glutMainLoop ();
}
```

Шаг 4: Сборка и запуск программы.

Итак, программа готова к запуску. Теперь надо собрать ее. Для этого перейдите в Solution Explorer и, кликнув правой кнопкой мыши, в выпадающем меню выберите Build.

Начнется процесс сборки программы а снизу в закладке Output начнет выводиться информация по компиляции и компоновке. В конце должно последовать сообщение о том, что сборка завершена успешно.

После того как проект собран, перейдите в папку с вашим решением (решением, а не проектом!), в ней находится вновь созданная папка Debug. Именно в ней был создан исполняемый файл. Его имя соответствует имени нашего проекта.

При попытке простого запуска операционная система выдаст сообщение об ошибке. Дело в том, что любая динамически подключаемая библиотека (коими являются GLEW и GLUT) состоит из трех частей — заголовочного файла .h, lib-файла и dll-файла.

Первые два нужны для компиляции и были скопированы нами в папку с самим проектом. Третий нужен уже при запуске программы. Требуется скопировать файлы glew32.lib glut32.lib из папки с заданием в папку с исполняемым файлом. (Рис. 23)






 glew32.dll	08.08.2008 18:44
 glut32.dll	23.06.2008 3:56
 Lab1.exe	15.02.2011 0:10
 Lab1.ilk	15.02.2011 0:10
 Lab1.pdb	15.02.2011 0:10

Рис. 23 – Окружение исполняемого файла

После запуска приложения на экране примерно следующее:

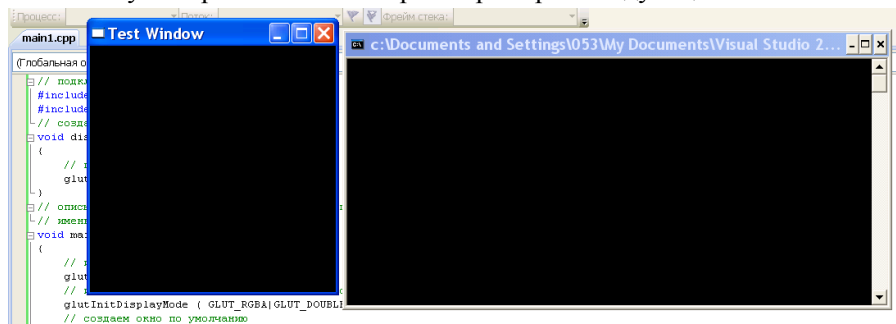


Рис. 24 – Окружение исполняемого файла

Здесь (Рис. 24) большое окно — это консольное приложение, а маленькое — окно GLUT.

Лучше понять библиотеку GLUT, поможет, пожалуй, самая короткая в мире программа OpenGL, которая была написана с использованием библиотеки GLUT. В листинге 2 представлена программа SIMPLE. Результат ее выполнения показан на рис. 25.

Листинг 2 – Программа SIMPLE

```
// подключаем заголовочные файлы библиотек
#include "glew.h"
#include "glut.h"
void RenderScene(void)
{
    // Окно очищается текущим цветом очистки
    glClear(GL_COLOR_BUFFER_BIT);
```

```

        // В буфер вводятся команды рисования
        glFlush();
    }
    // Устанавливается состояние визуализации
    void SetupRC(void)
    {
        glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
    }
    // Точка входа основной программы
    void main(void)
    {
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutCreateWindow("Simple");
        glutDisplayFunc(RenderScene);
        SetupRC();
        glutMainLoop();
    }

```

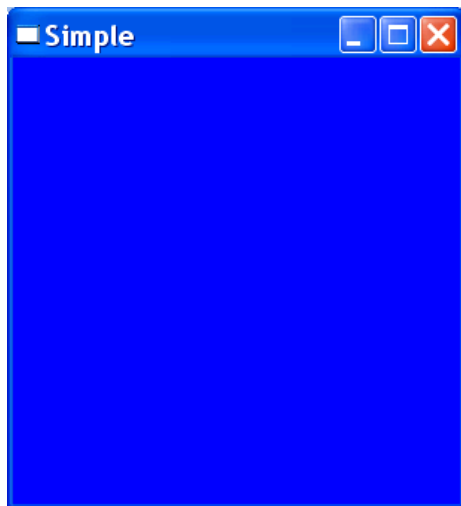


Рис. 25 – Результат выполнения программы SIMPLE

Программа SIMPLE делает немного. При запуске из командной строки (или среды разработки) она создает стандартное окно GUI с надписью Simple и синим фоном. Указанная простая программа содержит четыре

функции библиотеки GLUT (с префиксом `glut`) и три "настоящих" функции OpenGL ([с префиксом `gl`](#)). Рассмотрим программу построчно, а затем введем новые функции и существенно улучшим первый пример.

Анализ простейшей программы OpenGL

Заголовок. Листинг 2 содержит только один включаемый файл:

```
// подключаем заголовочные файлы библиотек
#include "glew.h"
#include "glut.h"
```

Тело. Переходим к точке входа всех программ C:

```
void main(void)
{
```

Программы C и C++ консольного режима работы всегда начинают выполнение с функции `main`.

Режим отображения: один буфер. Первая строка приведенного кода сообщает библиотеке GLUT, какой тип режима отображения использовать при создании окна:

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
```

Приведенные метки указывают задействовать окно с простой буферизацией (с одним буфером) (`GLUT_SINGLE`) и режим цвета `RGBA` (`GLUT_RGBA`). Простая буферизация означает, что все команды рисования выполняются в отображенном окне. Альтернативой является двойная буферизация, когда команды рисования выполняются в буфере вне экрана, а затем быстро отображаются в окне. Этот метод часто применяется для создания эффектов анимации. Режим цвета `RGBA` означает, цвета задаются указанием различных интенсивностей красного, зеленого и синего компонентов. Альтернативой является режим индексирования цвета, довольно распространенный и заключающийся в том, что вы задаете цвета с помощью указателей на палитру цветов.

Создание окна OpenGL. Следующий вызов к библиотеке GLUT создает окно на экране. С помощью приведенного ниже кода создается окно `Simple`.

```
glutCreateWindow("Simple");
```

Единственным аргументом `glutCreateWindow` является надпись в строке заголовка окна.

Отображение обратного вызова. Следующая строка кода, относящаяся к GLUT, имеет такой вид:

```
glutDisplayFunc(RenderScene);
```

В этой строке ранее определенная функция `RenderScene` устанавливается как функция обратного вызова дисплея. Это означает, что GLUT вызывает функцию, указывающую в то место, где нужно нарисовать окно. Такой вызов выполняется, например, при первом отображении окна, его изменении и разворачивании из пиктограммы. Именно в этом месте мы помещаем вызовы функций визуализации OpenGL.

Настройка контекста. Следующая строка не относится ни к GLUT, ни к OpenGL:

```
SetupRC();
```

В этой функции выполняется вся инициализация OpenGL, которую нужно выполнить перед визуализацией. Многие состояния OpenGL устанавливаются только один раз, и их не нужно обновлять при каждой визуализации кадра (экрана, заполненного графикой)

Последний вызов функции GLUT выполняется в конце программы.

```
glutMainLoop();
```

Эта функция запускает оболочку GLUT. Определив обратные вызовы для экрана дисплея и других функций, требуется освободить GLUT. Функция `glutMainLoop` не имеет обратного хода — после того, как ее вызвали, она выполняется до завершения программы; приложение должно вызывать ее только один раз. Эта функция обрабатывает все сообщения, связанные с операционной системой, нажатием клавиш и т.д., пока вы не завершите программу.

Другие функции графического вызова

Функция `SetupRC` содержит единственный вызов функции OpenGL.

```
glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
```

Эта функция устанавливает цвет, используемый для очистки окна. Прототип функции выглядит так:

```
void glClearColor(GLclampf red, GLclampf green,  
GLclampf blue, GLclampf alpha);
```

В большинстве реализаций OpenGL `GLclampf` определяется как величина типа `float`. В OpenGL единый цвет представляется как смесь красного, зеленого и синего компонентов. Каждый компонент может

представляться величиной, принадлежащей диапазону 0,0-1,0. Это похоже на спецификацию цветов в Windows с использованием макроса RGB для создания значения COLORREF. Различие заключается в том, что в Windows каждый компонент цвета в COLORREF принадлежит диапазону 0-255, что в сумме дает $256 \times 256 \times 256$ или более 16 миллионов цветов. В OpenGL значением каждого компонента могут быть приемлемые значения с плавающей запятой от 0 до 1, следовательно, число возможных цветов практически бесконечно. На практике цветовые возможности большинства устройств ограничены величиной 24 бит (16 миллионов цветов).

Таблица 2. Некоторые распространенные составные цвета

Составной цвет	Красный компонент	Зеленый компонент	Синий компонент
Черный	0.0	0.0	0.0
Красный	1.0	0.0	0.0
Зеленый	0.0	1.0	0.0
Желтый	1.0	1.0	0.0
Синий	0.0	0.0	1.0
Пурпурный	1.0	0.0	1.0
Голубой	0.0	1.0	1.0
Темно-серый	0.25	0.25	0.25
Светло-серый	0.75	0.75	0.75
Коричневый	0.60	0.40	0.12
Тыквенно-оранжевый	0.98	0.625	0.12
Пастельный розовый	0.98	0.04	0.7
Мягкий пурпурный	0.60	0.40	0.70
Белый	1.0	1.0	1.0

Отметим, что и Windows, и OpenGL, получая код цвета, во внутреннем представлении преобразовывают его в ближайшую точную величину, согласующуюся с возможностями доступной видеоаппаратуры.

Распространенные цвета и их коды приведены в табл. 2. Эти значения можно использовать в любой из функций OpenGL, связанных с цветом.

Последний аргумент функции `glClearColor` — это компонент альфа, который используется при смешении и создании таких специальных эффектов, как просвечивание. Просвечиванием называется способность объекта пропускать свет. Предположим, требуется создать кусок красного стекла, за которым находится источник синего света. Синий свет влияет на вид красного стекла (синий + красный = фиолетовый). Для генерации красного цвета полупрозрачного объекта можно использовать компонент альфа, при этом объект будет выглядеть как кусок цветного стекла; кроме того, будут видны объекты, находящиеся за ним. В создании эффектов такого типа участвуют не только значения альфа.

Очистка буфера цвета. Все, что было описано до этого момента, — это указали OpenGL использовать в качестве цвета очистки синий. В функции `RenderScene` требуется команда, выполняющая собственно очистку: `glClear(GL_COLOR_BUFFER_BIT);`

Функция `glClear` очищает определенный буфер или комбинацию буферов. Буфер — это область хранения информации об изображении. Красный, зеленый и синий компоненты рисунка обычно объединяются под общим названием *буфера цветов* или *буфера пикселей*.

Буфер цветов — это место, в котором хранится отображаемое изображение, и что очистка буфера с помощью команды `glClear` удаляет из окна последний отображенный рисунок.

Освобождение очереди. Последним идет завершающий вызов функции OpenGL.

```
glFlush();
```

В этой строке указывается выполнить все невыполненные команды OpenGL; у нас есть только одна такая команда — `glClear`.

Во внутреннем представлении OpenGL использует конвейер, последовательно обрабатывающий команды. Команды OpenGL часто выстраиваются в очередь, чтобы потом драйвер OpenGL обработал несколько "команд" одновременно. Такая схема увеличивает производительность, по-

сколькo связь с аппаратным обеспечением происходит медленно. В короткой программе, приведенной в листинге 1, функция `glFlush` просто сообщает OpenGL, что он должен обработать команды рисования, переданные на этот момент, и ожидать следующих команд.

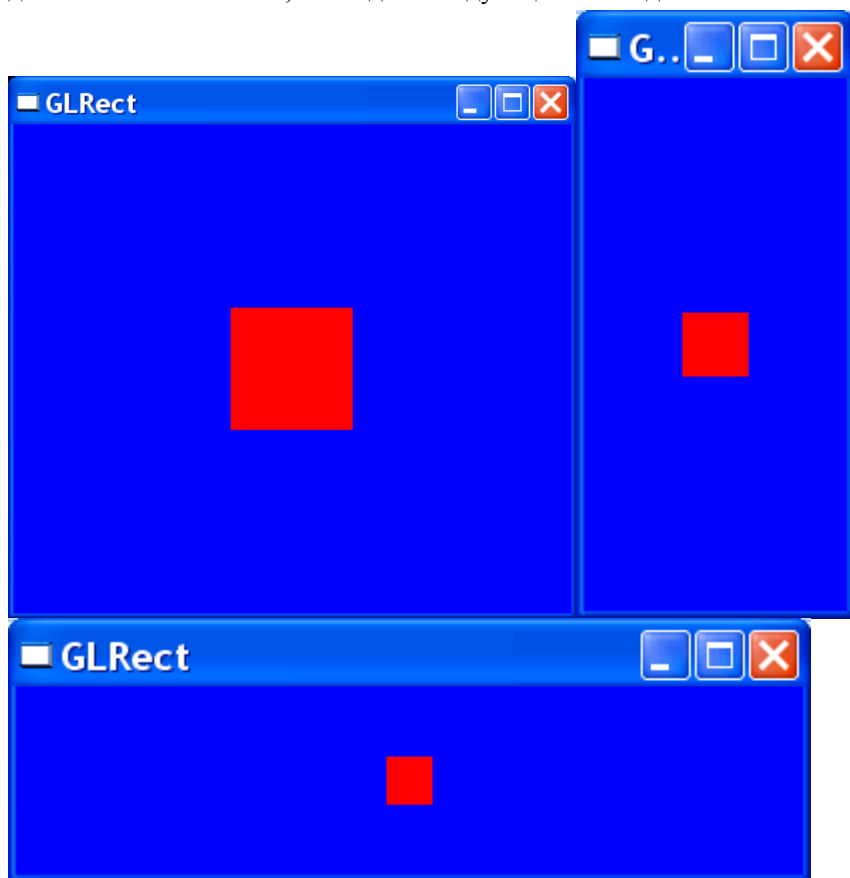


Рис. 26 – Результат выполнения программы GLRect

SIMPLE может не быть самой интересной программой OpenGL по сути, но она демонстрирует основы создания окна с помощью библиотеки GLUT, и на ее примере объясняется, как задавать цвет и очищать окно. Далее программа будет усложняться, за счет добавления в нее дополнительных функций библиотеки GLUT и OpenGL.

Рисование форм с помощью OpenGL. Программа SIMPLE создает пустое окно с синим фоном. Далее программа будет преобразована, чтобы на этом фоне был какой-нибудь рисунок. Кроме того, хотелось бы иметь возможность перемещать и изменять размеры окна, на что соответствующим образом реагирует код визуализации. Модифицированный вариант программы (GLRect) приведен в листинге 3, а результат ее выполнения изображен на рис. 26

Листинг 3 – Изображение централизованного прямоугольника с помощью OpenGL

```
// подключаем заголовочные файлы библиотек
#include "glew.h"
#include "glut.h"
void RenderScene(void) {
    // Очищаем окно, используя текущий цвет очистки
    glClear(GL_COLOR_BUFFER_BIT);
    // В качестве текущего цвета рисования задает
    красный //RGB
    glColor3f(1.0f, 0.0f, 0.0f);
    // Рисует прямоугольник, закрасенный текущим
    цветом
    glRectf(-25.0f, 25.0f, 25.0f, -25.0f);
    // Очищает очередь текущих команд
    glFlush();
}
////////// Задает состояние визуализации
void SetupRC(void)
{
    // Устанавливает в качестве цвета очистки синий
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
}
//////////Вызывается библиотекой GLUT при изменении размеров окна
void ChangeSize(GLsizei w, GLsizei h)
```



```

{
    GLfloat aspectRatio;
    // Предотвращает деление на ноль
    if(h == 0)
        h = 1;
    // Устанавливает поле просмотра с размерами ок-
на
    glViewport(0, 0, w, h);
    // Обновляет систему координат
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // С помощью плоскостей отсечения (левая, пра-
вая, нижняя,
    // верхняя, ближняя, дальняя) устанавливает
объем отсечения
    aspectRatio = (GLfloat)w / (GLfloat)h; if (w
<= h)
        glOrtho (-100.0, 100.0, -
100/aspectRatio, 100.0/aspectRatio, 1.0, -1.0);
    else
        glOrtho (-100.0 * aspectRatio, 100.0 *
aspectRatio, -100.0, 100.0, 1.0, -1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity(); }

//////////Точка входа основной программы
void main(void) {
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutCreateWindow("GLRect");
    glutDisplayFunc(RenderScene) ;
    glutReshapeFunc(ChangeSize);
    SetupRC();
    glutMainLoop();
}

```

Рисование прямоугольника. Ранее все программы очищали экран. Теперь в коде рисования присутствуют следующие строки:

```
// В качестве текущего цвета рисования задает красный
//RGB
glColor3f(1.0f, 0.0f, 0.0f);
```

В этих строках с помощью вызова `glColor3f` задается цвет, используемый для будущих операций рисования (цвет линий и заполнения). После этого функция `glRectf` отображает окрашенный прямоугольник.

Функция `glColor3f` выбирает цвет так же, как и `glClearColor`, но при этом не требуется указывать компонент прозрачности альфа (по умолчанию это значение равно 1.0 и соответствует непрозрачному объекту):

```
void glColor3f(GLfloat red, GLfloat green, GLfloat
blue);
```

Функция `glRectf` принимает аргументы с плавающей запятой, на что указывает суффикс `f`. Число аргументов в имени функции не используется, поскольку все разновидности `glRect` принимают четыре аргумента. Используемые в нашем примере аргументы функции `glRectf` представляют две пары координат— (x_1, y_1) и (x_2, y_2) :

```
void glRectf(GLfloat x1, GLfloat y1, GLfloat x2,
GLfloat y2);
```

Первая пара представляет левую верхнюю вершину прямоугольника, вторая — правую нижнюю.

Как OpenGL преобразует эти координаты в реальные точки окна? Это делается в функции обратного вызова `ChangeSize`. Функция задается как функция обратного вызова для любого изменения размера окна (когда оно растягивается, увеличивается до максимального размера и т.д.). Это задается так же, как устанавливается функция обратного вызова дисплея.

```
glutReshapeFunc(ChangeSize);
```

При каждом изменении размеров окна необходимо обновлять систему координат.

Масштабирование под размеры окна. Практически во всех средах с управлением окнами пользователь может в любой момент изменить размеры окна. Даже если вы пишете игру, которая всегда запускается в полноэкранном режиме, все равно считается, что окно меняет размер один раз — при создании. Когда это происходит, окно обычно отвечает перерисовыванием своего содержимого с учетом новых размеров. Иногда

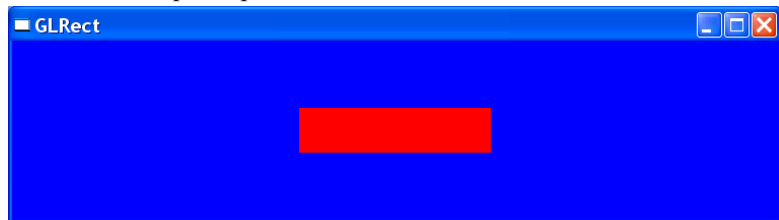
нужно просто вставить иллюстрацию в окно меньших размеров или отобразить весь рисунок в большем окне с исходными размерами. В нашем случае требуется масштабировать рисунок, чтобы он влезал в окно независимо от размеров окна или рисунка. Следовательно, очень маленькое окно будет иметь полное, но крошечное изображение, а в большем окне будет демонстрироваться похожий, но больший рисунок. Этот эффект можно наблюдать в большинстве программ рисования, когда вы растягиваете окно, не увеличивая изображение. Растягивание окна обычно не меняет размер расположенной в нем иллюстрации, но увеличение изображения приводит к ее росту.

Установка поля просмотра и отсекающего объема

Мы уже обсуждали, как поле просмотра и наблюдаемый объем влияют на диапазон координат и масштабирование двух- и трехмерных рисунков в двумерном окне на экране компьютера. Теперь мы исследуем установку поля просмотра и координат отсекающего объема в OpenGL.

Хотя наш рисунок — это двумерный плоский прямоугольник, в действительности мы рисуем в трехмерном координатном пространстве. Функция `glRectf` изображает прямоугольник на плоскости xy при $z = 0$. Наблюдение ведется вдоль положительного направления оси z , в результате при $z = 0$ вы видите квадрат.

При любом изменении размера окна нужно переопределить поле просмотра и отсекающий объем, учитывая размеры нового окна. В противном случае вы увидите эффект, подобный показанному на рис. 27, где отображение системы координат в экранные координаты остается постоянным для любых размеров окон.



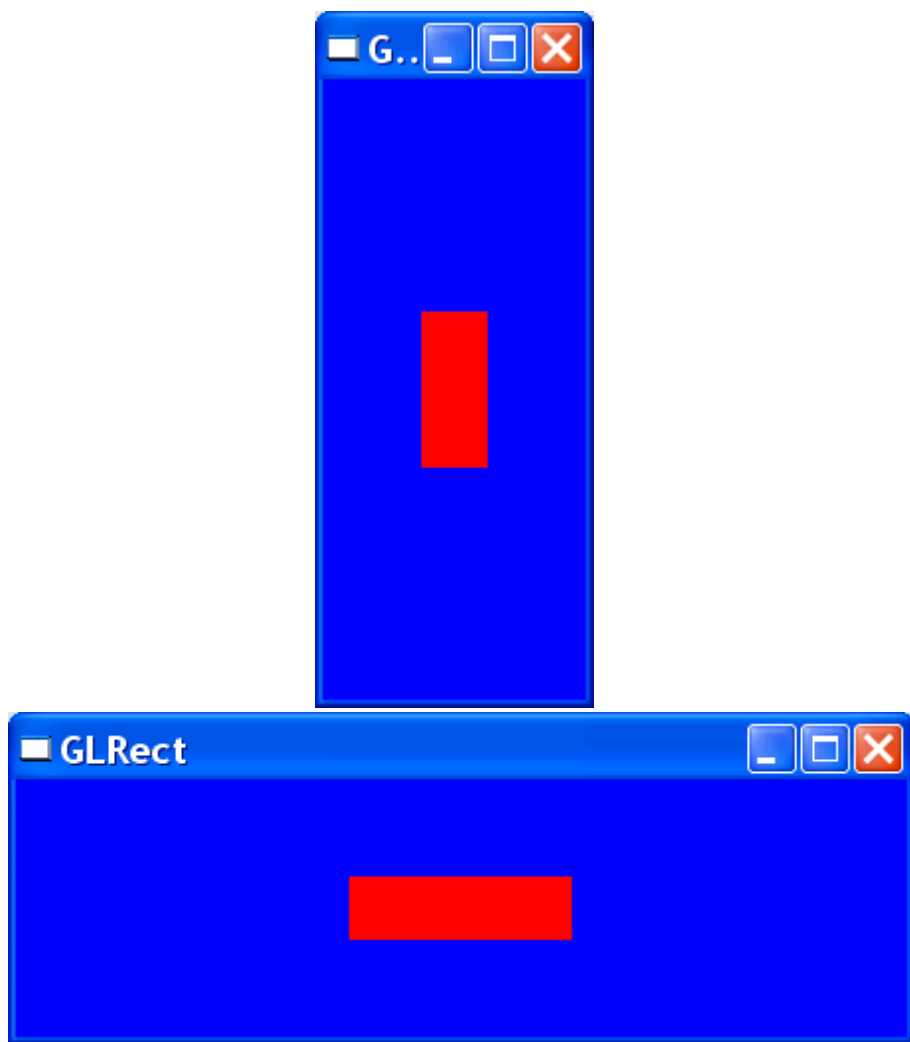


Рис. 27 – Эффекты изменения размера окна при фиксированной системе координат

Листинг 4 – Устойчивый к растяжению окна квадрат

```
// подключаем заголовочные файлы библиотек
#include "glew.h"
#include "glut.h"
```

```

void RenderScene(void) {
    // Очищаем окно, используя текущий цвет очистки
    glClear(GL_COLOR_BUFFER_BIT);
    // В качестве текущего цвета рисования задает
    красный //RGB
    glColor3f(1.0f, 0.0f, 0.0f);
    // Рисует прямоугольник, закрасенный текущим
    цветом
    glRectf(-25.0f, 25.0f, 25.0f, -25.0f);
    // Очищает очередь текущих команд
    glFlush();
}
////////// Задает состояние визуализации
void SetupRC(void)
{
    // Устанавливает в качестве цвета очистки синий
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
}
////////// Вызывается библиотекой GLUT при изменении размеров окна
void ChangeSize(GLsizei w, GLsizei h)
{
    GLfloat aspectRatio;
    // Предотвращает деление на ноль
    if(h == 0)
        h = 1;
    // Устанавливает поле просмотра с размерами окна
    glViewport(0, 0, w, h);
    // Обновляет систему координат
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // С помощью плоскостей отсечения (левая, правая,
    нижняя,

```

```

        // верхняя, ближняя, дальняя) устанавливает
объем отсечения
        glOrtho (-100.0, 100.0, -100, 100.0, 1.0,
-1.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
    }

//////////Точка входа основной программы
void main(void) {
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutCreateWindow("GLRect");
    glutDisplayFunc(RenderScene);
    glutReshapeFunc(ChangeSize);
    SetupRC();
    glutMainLoop();
}

```

Поскольку изменения размеров окна детектируется и обрабатывается по-разному в разных средах, библиотека GLUT предлагает функцию `glutReshapeFunc`, регистрирующую обратный вызов, который библиотека GLUT запускает при любом изменении размеров окна. Функция, которую вы передаете `glutReshapeFunc`, имеет такой прототип:

```
void ChangeSize(GLsizei w, GLsizei h);
```

В качестве описательного имени этой функции мы выбрали `ChangeSize` и будем его использовать в дальнейших примерах.

Функция `ChangeSize` принимает новую ширину и высоту после любого изменения размера окна. Используя две функции OpenGL `glViewport` и `glOrtho`, эту информацию можно применить для модификации отображения системы координат в действительные экранные координаты.

Определение поля просмотра

Чтобы понять, как получается определение поля просмотра, рассмотрим более внимательно функцию `ChangeSize`. Вначале она вызывает

`glViewport` с новой шириной и высотой. Функция `glViewport` определена следующим образом:

```
void glViewport(GLint x,  GLint y,  GLsizei width,
GLsizei height);
```

Параметры x и y задают левый нижний угол поля просмотра внутри окна, а параметры ширины и высоты определяют эти размеры в пикселях. Обычно x и y равны 0, но поля просмотра можно использовать для визуализации нескольких рисунков в различных областях окна. Поле просмотра определяет область внутри окна в реальных экранных [координатах](#), которые OpenGL может использовать для рисования (рис. 28). После этого текущий отсекающий объем отображается в новое поле просмотра. Если задать поле просмотра, меньшее окна, визуализация будет соответствующим образом масштабирована, как показано на рис. 29.

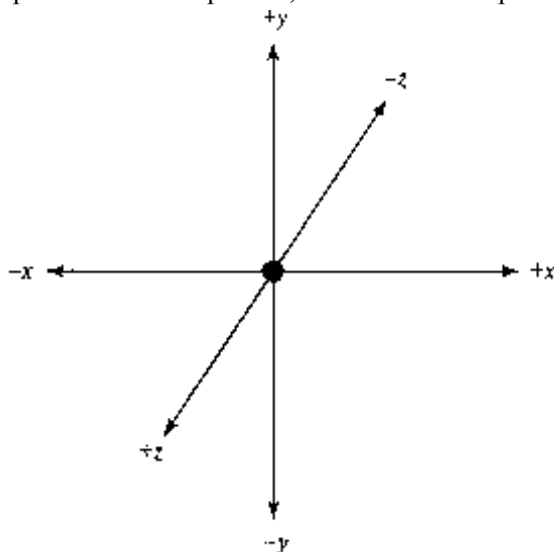


Рис. 28 – Декартово пространство окна



Рис. 29 – Отображение поля просмотра в окно

Определение отсеченного наблюдаемого объема

Последнее, что требуется от функции `ChangeSize`, — переопределить объем отсечения, чтобы характеристическое отношение по-прежнему соответствовало квадрату. Характеристическое отношение — это отношение числа пикселей вдоль единицы длины в вертикальном направлении к числу пикселей вдоль такой же единицы длины в горизонтальном направлении. Характеристическое отношение 1.0 определяет квадрат; 0.5 — задает, что на каждые два пикселя в горизонтальном направлении на единицу длины приходится один пиксель в вертикальном направлении на такую же единицу длины.

Если задается поле просмотра, не являющееся квадратом, но которое отображается в квадратный объем отсечения, изображение будет искажено. Например, поле просмотра, соответствующее размерам окна, но отображенное в квадратный отсеченный объем, приведет к тому, что изображения будут казаться высокими и тонкими в высоких и узких окнах и широкими и короткими в широких и коротких окнах. В подобном случае квадрат будет выглядеть квадратом только в окне квадратного размера.

В нашем примере для представления объема отсечения используется ортографическая проекция. Для создания этой проекции применяется команда OpenGL `glOrtho`:

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

В трехмерном декартовом пространстве значения `left` и `right` задают минимальную и максимальную координату точек, отображаемых вдоль оси `x`; `bottom` и `top` делают то же для оси `y`. Параметры `near` и `far` предназначены для оси `z`, обычно удалению от наблюдателя соответствуют отрицательные значения (рис. 28). Многие графические библиотеки используются в командах рисования координаты окна (пиксели).

Обратите внимание на два вызова функций сразу после кода с `glOrtho`.

```
// Обновляет систему координат
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

Наблюдаемый объем вы фактически определяете в проекционной матрице. Единственный вызов `glLoadIdentity` необходим, поскольку `glOrtho` в действительности не устанавливает объем отсечения, а моди-

фицирует существующий. Эта функция перемножает свои аргументы — матрицу, описывающую текущий объем отсечения, и матрицу, представляющую другой объем отсечения. Сейчас нужно знать только то, что прежде чем можно будет выполнять действия с матрицами, с помощью `glLoadIdentity` "обновляется" система координат. Без этого "обновления" каждый последующий вызов `glOrtho` может дать дальнейшее искажение целевого объема отсечения, который, возможно, даже не будет отображать прямоугольник.

Последние две строки кода, приведенные ниже, сообщают OpenGL, что все последующие преобразования повлияют на модель (то, что мы рисуем).

```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();
```

Как удержат квадрат квадратным?

За то, чтобы "квадрат" действительно был квадратным, отвечает следующий код:

```
// С помощью плоскостей отсечения (левая, правая,  
нижняя, верхняя,  
// ближняя, дальняя) устанавливает объем отсечения  
aspectRatio = (GLfloat)w / (GLfloat)h;  
if (w <= h)  
glOrtho (-100.0, 100.0, -100/aspectRatio,  
100.0/aspectRatio, 1.0, -1.0);  
else  
glOrtho (-100.0 * aspectRatio, 100.0 * aspectRatio, -  
100.0, 100.0, 1.0, -1.0);
```

Объем отсечения (видимое координатное пространство) модифицируется так, что левый край всегда проходит по линии $x = -100$, а правый простирается до 100, если ширина окна не больше его высоты. В таком случае горизонтальные размеры масштабируются согласно характеристическому значению окна. Подобным образом низ окна всегда проходит по линии $y = -100$, а верх простирается до 100, если высота окна не больше его ширины. В таком случае верхняя координата также масштабируется с коэффициентом, равным характеристическому отношению. Это позволяет поддерживать квадратную область 200 x 200 (с центром в точке 0,0)

вне зависимости от формы окна. Принцип действия таких установок показан на рис. 30.

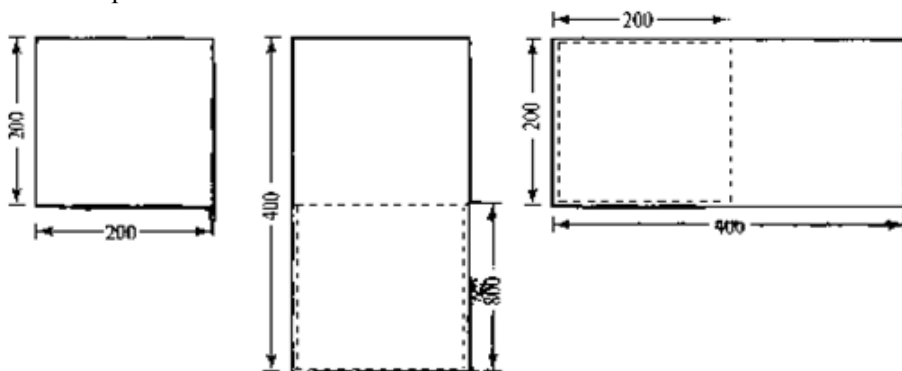


Рис. 30 – Область отсечения для трех различных окон

Анимация с помощью OpenGL и GLUT

До этого момента обсуждались основы применения библиотеки GLUT для создания окна и использования команд OpenGL для рисования. Часто нужно перемещать или поворачивать сцены, создавать анимационные эффекты. Возвращаясь к рассмотренному выше примеру с нарисованным квадратом, можно изменить его так, чтобы он рикошетом отскакивал от сторон окна. Можно создать цикл, который непрерывно меняет координаты объекта, перед вызовом функции `RenderScene`. В результате квадрат будет перемещаться в пределах окна.

Библиотека GLUT позволяет регистрировать функцию обратного вызова, которая облегчает установку простых анимированных последовательностей: `glutTimerFunc` принимает имя функции, которую нужно вызывать, и время ожидания до вызова функции.

```
void glutTimerFunc(unsigned int msecs, void  
(*func)(int value), int value);
```

В данном коде указывается, что GLUT должна ожидать msecs миллисекунд перед вызовом функции `func`. Параметру `value` можно передать определенное пользователем значение. Функция, вызываемая с помощью этого таймера, имеет следующий прототип:

```
void TimerFunction (int value);
```

В отличие от таймера Windows, эта функция срабатывает только один раз. Чтобы создать непрерывную анимацию, следует обновить таймер в соответствующей функции.

В программе GLRect можно заменить жестко запрограммированное положение прямоугольника переменными, а затем постоянно модифицировать эти переменные в функции-таймере. В результате будет казаться, что прямоугольник движется по окну. Рассмотрим пример анимации такого типа. В листинге 5 модифицирован листинг 4, чтобы квадрат отскакивал от внутренних границ окна. Нужно отслеживать положение и размер прямоугольника, а также учитывать любые изменения размера окна.

Листинг 5 - Анимированный прыгающий квадрат

```
// подключаем заголовочные файлы библиотек
#include "glew.h"
#include "glut.h"
// Исходное положение и размер прямоугольника
GLfloat x1 = 0.0f;  GLfloat y1 = 0.0f; GLfloat rsize
= 25;
// Величина шага в направлениях x и y (число пикселей,
// на которые на каждом шаге перемещается прямоуголь-
ник)
GLfloat xstep = 1.0f;  GLfloat ystep = 1.0f;
// Отслеживание изменений ширины и высоты окна
GLfloat windowWidth;  GLfloat windowHeight;
//Вызывается для рисования сцены
void RenderScene(void)
{
    // Очищаем окно,  используя текущий цвет очист-
ки
    glClear(GL_COLOR_BUFFER_BIT);
    // В качестве текущего цвета рисования задает
    красный //RGB
    glColor3f(1.0f, 0.0f, 0.0f);
```

```

        // Рисует прямоугольник, закрашенный текущим
цветом
        glRectf(x1, y1, x1 + rsize, y1- rsize);
        // Очищает очередь текущих команд и переключает
буферы
        glutSwapBuffers();
    }
    //Вызывается библиотекой GLUT в холостом состоянии
(окно не меняет
//размера и не перемещается)
void TimerFunction(int value)
{
    // Меняет направление на противоположное при
подходе
    // к левому или правому краю
    if(x1 > windowWidth-rsize || x1 < -
windowWidth)
        xstep = -xstep;
    // Меняет направление на противоположное при
подходе
    // к верхнему или нижнему краю
    if(y1 > windowHeight || y1 < -windowHeight +
rsize)
        ystep = -ystep;
    // Перемещает квадрат
    x1 += xstep; y1 += ystep;
    // Проверка границ. Если окно меньше прямо-
угольника,
    // который прыгает внутри, и прямоугольник об-
наруживает
    // себя вне нового объема отсечения
    if(x1 > (windowWidth-rsize + xstep))
        x1 = windowWidth-rsize-1; else if(x1 < -
(windowWidth + xstep)) x1 = - windowWidth -1; if(y1 >
(windowHeight + ystep))

```

```

        y1 = windowHeight-1; else if(y1 < -
(windowHeight - rsize + ystep)) y1 = -windowHeight +
rsize -1;
        // Перерисовывает сцену с новыми координатами
        glutPostRedisplay();
        glutTimerFunc(33,TimerFunction, 1);
    }
//Задаёт состояние визуализации
void SetupRC(void)
{
    // Устанавливает в качестве цвета очистки синий
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
}
//Вызывается библиотекой GLUT при изменении размеров
окна
void ChangeSize(GLsizei w,  GLsizei h)
{
    GLfloat aspectRatio;
    // Предотвращает деление на ноль
    if(h == 0) h = 1;
    // Устанавливает поле просмотра с размерами ок-
на
    glViewport(0, 0, w, h);
    // Обновляет систему координат
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    //С помощью плоскостей отсечения (левая, пра-
вая, нижняя,
    // верхняя, ближняя, дальняя) устанавливает
объём отсечения
    aspectRatio = (GLfloat)w / (GLfloat)h;
    if (w <= h)
    {
        windowWidth = 100;
        windowHeight = 100 / aspectRatio;
    }
}

```

```

        glOrtho (-100.0, 100.0, -
windowHeight, windowHeight, 1.0, -1.0);
    }
    else
    {
        windowHeight = 100 * aspectRatio;
windowHeight = 100;
        glOrtho (-windowWidth, windowHeight, -
100.0, 100.0, 1.0, -1.0);
    }
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
//Точка входа основной программы
void main(void)
{
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutCreateWindow("Bounce") ;
    glutDisplayFunc(RenderScene) ;
    glutReshapeFunc(ChangeSize) ;
    glutTimerFunc(33, TimerFunction, 1);
    SetupRC();
    glutMainLoop() ;
}

```

Двойная буферизация

Одной из наиболее важных особенностей любого графического пакета является поддержка *двойной буферизации*. Это позволяет выполнять код рисования, визуализируя при этом закадровый буфер. Затем с помощью команды замены рисунок мгновенно выводится на экран.

Двойная буферизация может служить двум целям. Первая: отображение сложных рисунков требует немалого времени, и возможно, вы не хотите видеть каждый этап построения изображения. С помощью двойной буферизации можно сформировать изображение и отобразить его только после завершения. Пользователь никогда не увидит частичного изобра-

жения — только после того, как иллюстрация будет готова целиком, она будет показана на экране.

Вторая цель двойной буферизации проявляется при анимации. Каждый кадр строится в закадровом буфере и, когда он готов, быстро переключается на экран. Отметим, что библиотека GLUT поддерживает окна с двойной буферизацией. Итак, обратите внимание на следующую строку в листинге 5:

```
glutInitDisplayMode\(GLUT\_DOUBLE | GLUT\_RGB\);
```

Мы изменили `GLUT_SINGLE` на `GLUT_DOUBLE`. В результате этой модификации весь код, относящийся к рисованию, визуализируется в закадровом буфере.

Далее мы также изменили конец функции `RenderScene`.

```
...  
// Очищает очередь текущих команд и переключает буфе-  
ры  
glutSwapBuffers();  
}
```

Функция [glFlush](#) уже не вызывается. Она уже не нужна, поскольку, выполняя замену буферов, мы неявно выполняем операцию очистки буфера.

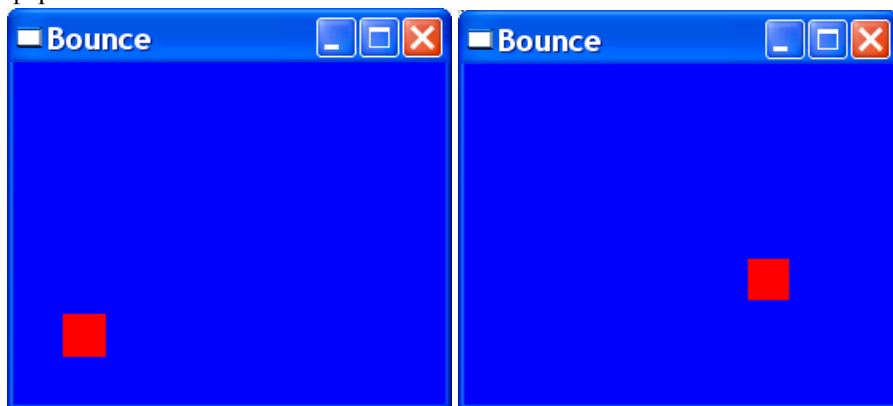


Рис. 31 – Следуй за прыгающим квадратом

Вследствие указанных изменений получаем анимированный прямоугольник, показанный на рис. 31. Функция `glutSwapBuffers` по-

прежнему выполняет операцию очистки буфера, даже если работает в режиме одного буфера. Просто восстановите вместо `GLUT_DOUBLE` значение `GLUT_SINGLE` в примере с прыгающим прямоугольником, чтобы посмотреть анимацию без двойной буферизации. Видно, что прямоугольник постоянно мигает и запинаяется — при единственном буфере анимация получается весьма некачественной.

Машина состояний OpenGL

Если дан геометрический объект, на его рисование может повлиять множество фактов. Освещен ли он? Каковы свойства света? Каковы свойства материала? Какую текстуру следует применить (или вообще никакой)? Список можно продолжать очень долго.

Такой набор переменных мы называем состоянием конвейера. Машина состояний (или конечный автомат) — это абстрактная модель набора переменных состояния, имеющих разные значения, включенные и выключенные и т.д. Задавать все переменные состояния, когда что-то рисуется в OpenGL, непрактично. Вместо этого в OpenGL реализована модель состояний (или конечный автомат), предназначенная для отслеживания всех переменных состояния OpenGL. Установленное значение состояния остается до тех пор, пока другая функция его не изменит. Многие состояния — это просто метки "включено" или "выключено". Например, освещение либо включено, либо выключено. Геометрия, нарисованная без освещения, рисуется без применения к набору цветов расчетов освещения. Любая геометрия, нарисованная после включения освещения, изображается согласно расчетам освещенности.

Чтобы включать и выключать переменные состояния подобного типа используется следующая функция OpenGL:

```
void glEnable(GLenum capability);
```

Для отключения переменной используется соответствующая функция:

```
void glDisable(GLenum capability);
```

Освещение, например, можно включить с помощью следующей команды:

```
glEnable(GL_LIGHTING);
```

Отключается освещения с помощью такой функции:

```
glDisable(GL_LIGHTING);
```

Если требуется проверить переменную состояния — активизирована она или отключена, — OpenGL предлагает следующий удобный механизм:

```
GLboolean glIsEnabled(GLenum capability);
```

Однако не все переменные состояний могут быть просто включенными или выключенными. Многие из функций OpenGL задают значения, "связанные" до момента изменения. Эти значения также можно в любой момент проверить. Существует целый набор функций запроса, позволяющих узнавать значения переменных булевого типа, целых, с плавающей запятой и двойной точности. Эти четыре функции имеют следующие следующие прототипы:

```
void glGetBooleanv(GLenum pname, GLboolean *params);  
void glGetDoublev(GLenum pname, GLdouble *params);  
void glGetFloatv(GLenum pname, GLfloat *params);  
void glGetIntegerv(GLenum pname, GLint *params);
```

Каждая функция возвращает одно значение или массив значений, хранящий результаты искомого запроса.

Запись и восстановление состояний

OpenGL также имеет удобный механизм для хранения диапазона значений состояния с возможностью их последующего восстановления. Здесь следует ввести понятие *стек* — удобной структуры данных, позволяющей *вталкивать* (записывать) значения в стек и впоследствии *выталкивать* (извлекать) их из стека. Элементы извлекаются из стека в порядке, противоположном тому, в каком они туда помещались.

Одну переменную состояния OpenGL или целый набор связанных значений переменных состояния можно поместить в стек атрибутов с помощью следующей команды:

```
void glPushAttrib(GLbitfield mask);
```

Приведенная ниже команда позволяет извлекать соответствующие значения.

```
void glPopAttrib(GLbitfield mask);
```

Обратите внимание на то, что аргумент этих функций — битовое поле. Это означает, что используется побитовая маска, позволяющая с помощью операции побитового ИЛИ (в C — используя оператор `|`) указывать несколько значений переменных состояния в одном вызове функции. На-

пример, при выполнении приведенной ниже команды, запишутся состояния освещения и текстуры.

```
glPushAttrib(GL_TEXTURE_BIT | GL_LIGHTING_BIT);
```

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Воспроизвести результаты, представленные в теоретическом обзоре, применить различные виды проекции, согласно варианту, полученному у преподавателя, изменить анимацию графического примитива.

ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ

Задание выполняется согласно варианту. По завершении готовится отчет.

- 1) Для [Листинга 1](#) знать все функции и уметь объяснить основные входные параметры
- 2) Для [Листинга 2](#) поменять цвет фона и название окна
- 3) Для [Листинга 3](#) и [Листинга 4](#) поменять цвет фона, цвет объекта, координаты объекта и количество объектов. Очень важно понимать работу функций `void glViewport(GLint x, GLint y, GLsizei width, GLsizei height)` и `void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)`. Продемонстрировать работу программы с не менее чем 5 разными вариантами параметров для каждой из этих функций. Объяснить влияние каждой функции на вывод графических объектов.
- 4) Для [Листинга 5](#) реализовать следующие изменения согласно варианту.

ВАРИАНТЫ ЗАДАНИЙ

- 1) выполнить вращение квадрата,
- 2) выполнить перемещение двух квадратов,
- 3) выполнить вращение прямоугольника,
- 4) выполнить перемещение двух прямоугольников с разными параметрами (цвет, размер),
- 5) выполнить пульсирующее масштабирование квадрата с центром в произвольной координате,

- 6) выполнить пульсирующее масштабирование прямоугольника с центром в середине окна,
- 7) выполнить пульсирующее масштабирование с вращением квадрата,
- 8) выполнить пульсирующее масштабирование с вращением прямоугольника,
- 9) выполнить пульсирующее масштабирование движущегося квадрата,
- 10) выполнить пульсирующее масштабирование движущегося прямоугольника.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Дайте определение OpenGL и GLUT.
2. Приведите классификацию видов проекций и опишите особенности их реализации в OpenGL.
3. Сформулируйте принципы именования методов OpenGL и поясните значения префиксов и суффиксов.
4. Объясните сущность процесса подключения библиотек OpenGL к проекту.
5. Охарактеризуйте, как описывается декартово пространство в приложении к окну.
6. Раскройте область применения цветовой модели использованной в работе.
7. Раскройте значение термина поле просмотра, и назовите функцию его определяющую.
8. Объясните принцип работы функции очистки буфера.
9. Раскройте основные цели отсечения наблюдаемого объема.
10. Опишите сущность и назначение двойной буферизации.
11. Изложите концепцию управления машиной состояний.
12. Сопоставьте типы данных, которые используются в OpenGL и в С-подобных языках.
13. Проведите классификацию видов координат, использовавшихся в работе.
14. Раскройте основные этапы действия упрощенной модели графического конвейера.
15. Перечислите основные области применения программного интерфейса OpenGL .

ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ

На выполнение лабораторной работы отводится 2 занятия (4 академических часа: 3 часа на выполнение и сдачу практического задания и 1 час на подготовку отчета).

Номер варианта студента назначается индивидуально преподавателем.

В отчете должны быть представлены:

- 1) Текст задания для лабораторной работы и номер варианта.
- 2) Листинги программ: Листинг 2 с комментариями и измененными параметрами; Листинг 4 или Листинг 3 с измененными параметрами (отдельно указать не менее 5 вариантов изменений каждой функции и продемонстрировать ключевые снимки экрана, не менее 4); Листинг 5 привести согласно варианта задания и продемонстрировать не менее 2-х снимков экрана.

Отчет по каждому новому заданию начинать с новой страницы. В выводах отразить затруднения при ее выполнении и достигнутые результаты.

Отчет на защиту предоставляется в печатном виде.

Отчет содержит: Структура отчета (на отдельном листе(-ах)): титульный лист, формулировка задания (вариант), этапы выполнения работы, результаты выполнения работы, выводы.

ОСНОВНАЯ ЛИТЕРАТУРА

1. Боресков А.В. Основы работы с технологией CUDA / А.В. Боресков, А.А. Харламов - Издательство "ДМК Пресс", 2010. - 232 с. - ISBN 978-5-94074-578-5; ЭБС «Лань». - URL: https://e.lanbook.com/book/1260#book_name (23.12.2017).
2. Васильев С.А. OpenGL. Компьютерная графика : учебное пособие / С.А. Васильев. — Электрон. текстовые данные. — Тамбов: Тамбовский государственный технический университет, ЭБС АСВ, 2012. — 81 с. — 2227-8397. — Режим доступа: <http://www.iprbookshop.ru/63931.html> — ЭБС «IPRbooks», по паролю
3. Вольф Д. OpenGL 4. Язык шейдеров. Книга рецептов/ Вольф Д. - Издательство "ДМК Пресс", 2015. - 368 с. - 978-5-97060-255-3; ЭБС «Лань». - URL: https://e.lanbook.com/book/73071#book_name (23.12.2017).
4. Гинсбург Д. OpenGL ES 3.0. Руководство разработчика/Д. Гинсбург, Б. Пурномо. - Издательство "ДМК Пресс", 2015. - 448 с. - ISBN 978-5-97060-256-0; ЭБС «Лань». - URL: https://e.lanbook.com/book/82816#book_name (29.12.2017).
5. Лихачев В.Н. Создание графических моделей с помощью Open Graphics Library / В.Н. Лихачев. — Электрон. текстовые данные. — М. : Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. — 201 с. — 2227-8397. — Режим доступа: <http://www.iprbookshop.ru/39567.html>
6. Забелин Л.Ю. Основы компьютерной графики и технологии трехмерного моделирования : учебное пособие/ Забелин Л.Ю., Конюкова О.Л., Диль О.В.— Новосибирск: Сибирский государственный университет телекоммуникаций и информатики, 2015.— 259 с.— Режим доступа: <http://www.iprbookshop.ru/54792>.— ЭБС «IPRbooks», по паролю
7. Папуловская Н.В. Математические основы программирования трехмерной графики : учебно-методическое пособие / Н.В. Папуловская. — Электрон. текстовые данные. — Екатеринбург: Уральский федеральный университет, 2016. — 112 с. — 978-5-7996-1942-8. — Режим доступа: <http://www.iprbookshop.ru/68345.html>
8. Перемитина, Т.О. Компьютерная графика : учебное пособие / Т.О. Перемитина ; Министерство образования и науки Российской Федера-

ции, Томский Государственный Университет Систем Управления и Радиоэлектроники (ТУСУР). - Томск : Эль Контент, 2012. - 144 с. : ил.,табл., схем. - ISBN 978-5-4332-0077-7 ; - URL: <http://biblioclub.ru/index.php?page=book&id=208688> (30.11.2017).

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

Электронные ресурсы:

1. <http://rdsn.org/article/opengl/ogltut2.xml> - Графическая библиотека OpenGL - примеры использования
2. <https://ru.wikipedia.org/wiki/OpenGL> - OpenGL — Википедия
3. http://www.opengl-master.ru/view_func.php - Справочник функций OpenGL
4. <https://sites.google.com/site/raznyeuropoinformatiki/home/opengl-s/vstroennyye-funkcii> - Функции для работы с буфером кадра OpenGL