

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра автоматизированных систем управления

**МЕТОДИЧЕСКИЕ УКАЗАНИЯ
К ЛАБОРАТОРНЫМ РАБОТАМ**

по дисциплине

«КОМПЬЮТЕРНАЯ ГРАФИКА»

для студентов специальностей

09.03.02 Информационные системы и технологии,

09.03.01 Информатика и вычислительная техника

Рассмотрено на заседании кафедры
«Автоматизированные системы
управления»

Протокол № 6 от 19 января 2017 г.

Утверждено на заседании
учебно – издательского совета ДонНТУ
Протокол № 3 от 6 апреля 2017 г.

**Донецк
2017**

Методические указания к лабораторным работам по дисциплине «Компьютерная графика» для студентов специальностей 09.03.02 «Информационные системы и технологии» (ИС), 09.03.01 «Информатика и вычислительная техника» (АСУ)/ Составили: Васяева Т.А., Матях И.В. – Донецк: ДНТУ, 2017. – 91 с.

Методические указания содержат краткие теоретические сведения, методические рекомендации и задания для выполнения лабораторных работ по дисциплине «Компьютерная графика». Изложена методика выполнения каждой из лабораторных работ, требования к содержанию отчетов, список рекомендуемой литературы.

Утверждено методическими комиссиями специальностей 09.03.02 Информационные системы и технологии, 09.03.01 Информатика и вычислительная техника

Составители: к.т.н., доц. Васяева Т.А.
ас. Матях И.В.

Рецензент: к.т.н., доц., доц. каф. АСУ

М.В. Привалов

Рецензент: к.т.н., доц., доц. каф. АТ

В.В. Червинский

Ответственный за выпуск:

зав. каф. АСУ Привалов М.В.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	4
ЛАБОРАТОРНАЯ РАБОТА №1 - ПОСТРОЕНИЕ ГРАФИЧЕСКИХ ФИГУР НА ПЛОСКОСТИ	5
ЛАБОРАТОРНАЯ РАБОТА №2 - МИРОВЫЕ И ЭКРАННЫЕ КООРДИНАТЫ, НАСТРОЙКА ПОРТА ПРОСМОТРА	15
ЛАБОРАТОРНАЯ РАБОТА №3 - КООРДИНАТНЫЕ ПРЕОБРАЗОВАНИЯ НА ПЛОСКОСТИ	24
ЛАБОРАТОРНАЯ РАБОТА №4 - ПОСТРОЕНИЕ ГРАФИЧЕСКИХ ФИГУР В ПРОСТРАНСТВЕ, КООРДИНАТНЫЕ ПРЕОБРАЗОВАНИЯ В ПРОСТРАНСТВЕ	33
ЛАБОРАТОРНАЯ РАБОТА №5 - РАСТРОВЫЕ АЛГОРИТМЫ ПОСТРОЕНИЯ КОНТУРА ФИГУР	45
ЛАБОРАТОРНАЯ РАБОТА №6 - РАСТРОВЫЕ АЛГОРИТМЫ ЗАПОЛНЕНИЯ ФИГУР.....	61
ЛАБОРАТОРНАЯ РАБОТА №7 - АЛГОРИТМЫ ОТСЕЧЕНИЯ ФИГУР	67
ЛАБОРАТОРНАЯ РАБОТА №8 - ПОСТРОЕНИЕ РЕАЛИСТИЧНЫХ ИЗОБРАЖЕНИЙ.....	75
СПИСОК ЛИТЕРАТУРЫ	90

ВВЕДЕНИЕ

Компьютерная графика представляет собой одно из направлений развития систем обработки информации связанной с изображением объектов, которое возникло в связи с необходимостью широкого использования систем компьютерной графики в виртуальной реальности, в глобальной сети Internet и системах интерактивной графики. Понятие «компьютерная графика» очень часто трактуется по-разному. Компьютерная графика – это новая отрасль знаний, которая, с одной стороны, представляет комплекс аппаратных и программных средств, используемых для формирования, преобразования и выдачи информации в визуальной форме на средства отображения ЭВМ. С другой стороны, под компьютерной графикой понимают совокупность методов и приемов для преобразования при помощи ЭВМ данных в графическое представление. Системы компьютерной графики обеспечивают пользователю широкий набор услуг и позволяют создавать целый ряд различных способов диалога, типа человек – компьютер, позволяют создавать анимационные и реалистичные изображения и совершенствуют способы ввода-вывода информации.

Изучение данной дисциплины вносит необходимый вклад в достижение ожидаемых результатов в профессиональной части программы подготовки студентов специальностей «Информационные системы и технологии» и «Информатика и вычислительная техника».

Курс основан на изучении математических основ предметной области, значительное внимание уделяется программной реализации.

Основная задача курса – дать в доступной форме математический аппарат для создания изображений геометрических объектов на экране и их манипуляций, алгоритмы растровой графики; представление пространственных форм, включая алгоритмы удаления скрытых линий и поверхностей.

Лабораторная работа № 1

Тема: построение графических фигур на плоскости

Аппаратно-независимое программирование и OpenGL

Строго OpenGL определяется как программный интерфейс к графической аппаратуре. Этот интерфейс сделан в виде библиотеке функций (Open Graphics Library – OpenGL). Разработчик – фирма Silicon Gragphics. Таким образом, можно сказать что OpenGL очень мобильная и очень эффективная библиотека трехмерной графики и моделирования.

Сейчас OpenGL стал индустриальным стандартом, поддерживается многими операционными системами для разнообразных аппаратных платформ. Приятно, когда возможен одинаковый подход к написанию графических приложений, когда, например, одна и та же программа может быть скомпилирована и запущена в различных графических средах, когда есть гарантия, что на любом дисплее будет получено примерно одинаковое изображение. Такой подход и называется аппаратно-независимым графическим программированием. В OpenGL предлагается программное средство, при котором перенос графической программы требует только установки соответствующих библиотек OpenGL на новой машине, а само приложение не требует никаких изменений и вызывает из новой библиотеки те же самые функции с теми же параметрами. В результате получаются те же графические результаты. Метод создания графики в OpenGL был принят большим количеством компаний, и библиотеки OpenGL существуют для всех значимых графических сред.

Программирование управляемости событиями

Многие современные графические системы являются оконными и показывают на экране несколько перекрывающихся окон одновременно. Пользователь с помощью мыши может перемещать окна по экрану, а также изменять их размер. Общим свойством, которым обладает большая часть оконных

программ, является их управляемость событиями. Это означает, что программы реагируют на различные события, такие как щелчок мышью, нажатие на клавишу клавиатуры или изменение размеров окна на экране. Система автоматически устанавливает очередь событий, которая получает сообщения о том, что произошло некоторое событие, и обслуживает их по принципу «первым пришел — первым обслужен». Программист организует свою программу как набор функций обратного вызова (callback functions), которые выполняются, когда происходят события. Функция обратного вызова создается для каждого типа событий, которые могут произойти. Когда система удаляет событие из очереди, оно просто выполняет функцию обратного вызова, связанную с данным типом событий. Тем программистам, которые привыкли делать программы по принципу «сделай то, затем сделай это», потребуется некоторое переосмысление. Новая структура программ может быть передана словами «не делай ничего, пока не произойдет какое-нибудь событие, а затем сделай определенную вещь».

Функции обратного вызова:

- `glutDisplayFunc(myDisplay)`. Когда система решает, что какое-нибудь окно на экране подлежит перерисовке, она создает событие «redraw» (обновить). Это имеет место при первом открытии окна, а также когда окно вновь становится видимым, когда другое окно перестало его заслонять. В данном случае функция `myDisplay()` зарегистрирована в качестве функции обратного вызова для события обновления.

- `glutReshapeFunc(myReshape)`. Форма экранного окна может быть изменена (reshape) пользователем, обычно путем захвата мышью угла окна и смещения его с помощью мыши (простое перемещение окна не приводит к событию «изменение формы».) В данном случае функция `myReshape()` зарегистрирована с событием изменения формы окна. Как мы увидим дальше, функции `myReshape()` автоматически передаются аргументы, информирующие о новой ширине и высоте изменившего свою форму окна.

– `glutMouseFunc(myMouse)`. Когда нажимают или отпускают одну из кнопок мыши, то возникает событие «мышь» (`mouse`). В данном случае функция `myMouse()` зарегистрирована в качестве функции, которая будет вызываться всякий раз, когда произойдет событие «мышь». Функции `myMouse()` автоматически передаются аргументы, описывающие местоположение мыши, а также вид действия, вызываемого нажатием кнопки мыши.

– `glutKeyboardFunc(myKeyboard)`. Эта команда регистрирует функцию `myKeyboard()` для события, заключающегося в нажатии или отпускании какой-либо клавиши на клавиатуре. Функции `myKeyboard()` автоматически передаются аргументы, сообщающие, какая клавиша была нажата. Для удобства этой функции также передается информация о положении мыши в момент нажатия клавиши.

Если в какой-либо программе мышь не используется, то соответствующую функцию обратного вызова не нужно писать и регистрировать. В этом случае щелчок мыши не окажет на программу никакого действия. Это же верно для программ, не использующих клавиатуру.

Функция `glutMainLoop()`, которую необходимо использовать после регистрации функций обратного вызова, выполняется после того, как программа рисует начальную картину и входит в бесконечный цикл, находясь в котором она просто ждет наступления событий.

Первая задача при создании изображений заключается в открытии экранного окна для рисования. Эта работа является достаточно сложной и зависит от системы. Поскольку функции OpenGL являются аппаратно-независимыми, то в них не предусмотрено поддержки управления окнами определенных систем. Однако в OpenGL Utility Toolkit включены функции для открытия окна в любой используемой вами системе.

– `glutInit(&argc, argv)`. Эта функция инициализирует OpenGL Utility Toolkit. Ее аргументы для передачи информации о командной строке являются стандартными; не всегда есть необходимость их использовать.

– `glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB)`. Эта функция определяет, как должен быть инициализирован дисплей. Встроенные в нее константы `GLUT_SINGLE` и `GLUT_RGB` с оператором OR (или) между ними показывают, что следует выделить один дисплейный буфер и что цвета задаются с помощью сочетаний красного, зеленого и синего цветов. Естественно могут быть и другие аргументы. Например, для плавной анимации используют двойную буферизацию.

– `glutInitWindowSize(640,480)`. Эта функция устанавливает, что экранное окно при инициализации должно иметь 640 пикселей в ширину и 480 пикселей в высоту. Во время выполнения программы пользователь может изменять размер этого окна по своему желанию.

– `glutInitWindowPosition(100,150)`. Эта функция устанавливает, что верхний левый угол данного окна должен находиться в 100 пикселях от левого края и в 150 пикселях от верхнего края. Во время выполнения программы пользователь может перемещать это окно, куда пожелает.

– `glutCreateWindow("my first attempt")`. Эта функция фактически открывает и отображает экранное окно, помещая текст заголовка «my first attempt» (моя первая попытка) в строку заголовка (title bar) окна.

Рисование основных графических примитивов

Команды рисования помещаются в функцию обратного вызова, связанную с событием `redraw` (обновление), например, в функцию `myDisplay()`.

Для рисования объектов в OpenGL необходимо передать ему список вершин. Этот список возникает между двумя вызовами OpenGL-функций: `glBegin()` и `glEnd()`. Аргумент в `glBegin()` определяет, какой объект рисуется. Например, можно нарисовать три точки, нарисованные в окне шириной в 640 пикселей и высотой в 480 пикселей. Эти точки рисуются с помощью следующей последовательности команд:

```
glBegin(GL_POINTS);
glVertex2i(100,50);
```



```
glVertex2i(100,130);
glVertex2i(150,130);
glEnd();
```

GL_POINTS является встроенной константой OpenGL, для рисования точек. Для рисования других примитивов следует заменить GL_POINTS на другие константы:

GL_LINES используется для рисования отрезков, при этом необходимо задавать четное количество вершин между командами glBegin(GL_LINES) и glEnd().

GL_LINE_STRIP используется для рисования ломаной линии (совокупности отрезков прямых, соединенных своими концами). В OpenGL ломаная рисуется посредством задания вершин в нужном порядке, между командами glBegin(GL_LINE_STRIP) и glEnd().

GL_LINE_LOOP используется если нужно соединить последнюю точку с первой, чтобы ломаная линия превратилась в полигон. Полигоны, нарисованные с помощью GL_LINE_LOOP, нельзя заполнять цветом или узором.

GL_POLYGON используется для рисования закрашенных полигонов.

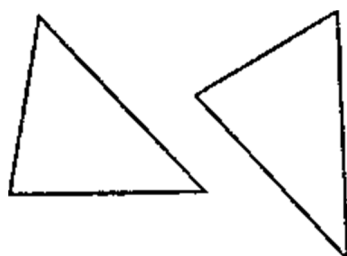
GL_TRIANGLES берет вершины из списка по три за один прием и рисует для каждой тройки отдельный треугольник.

GL_QUADS берет вершины по четыре за один прием и рисует для каждой четверки отдельный четырехугольник.

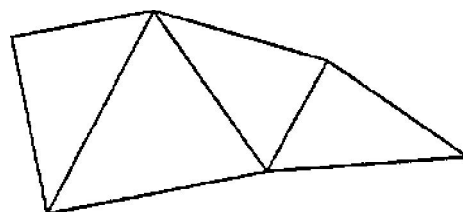
GL_TRIANGLE_STRIP (полоса из треугольников) рисует последовательность треугольников, опирающихся на тройки вершин: v_0, v_1, v_2 , затем v_2, v_1, v_3 , затем v_2, v_3, v_1 и т. д. (порядок следования такой, что все треугольники «кладутся поперек» в одном направлении, например, против часовой стрелки).

GL_TRIANGLE_FAN (веер из треугольников) рисует последовательность соединенных между собой треугольников, опирающихся на тройки вершин: v_0, v_1, v_2 затем v_0, v_2, v_3 , затем v_0, v_3, v_4 и т.д.

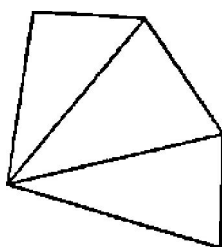
GL_QUAD_STRIP (полоса из четырехугольников) рисует последовательность четырехугольников, опирающихся на четверки вершин: вначале v_0, v_1, v_3, v_2 , затем v_2, v_3, v_5, v_4 , затем v_4, v_5, v_7, v_6 , и т. д. (порядок следования такой, что все четырехугольники «кладутся поперек» в одном направлении, например против часовой стрелки).



GL_TRIANGLES



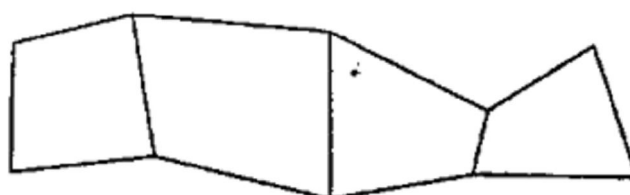
GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN



GL_QUADS



GL_QUAD_STRIP

Рисунок 1.1 - Рисование примитивов

Эти команды посылают информацию о каждой вершине в «графический конвейер» («graphics pipeline»), в котором она проходит ряд преобразований. 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, поскольку связь с аппаратным обеспечением происходит медленно. Функция *glFlush()* сообщая

ет OpenGL что нужно обрабатывать команды рисования, переданных на этот момент и ожидать следующих, поэтому ее следует вызывать после рисования объекта.

Пример законченной программы на OpenGL

```
// Пример рисования 3-х точек
#include <GL/glut.h> // подключение библиотек
#include <GL/gl.h> //в папке include (вашей про-
граммной среды) создать папку GL и поместить файлы
gl.h, glu.h, glut.h
// в папку lib (вашей программной среды) поместить
файлы opengl32.lib, glu32.lib, glut32.lib
// в папку system (вашей ОС) поместить файлы
opengl32.dll, glu32.dll, glut32.dll, glut.dll

void display() // функция обратного вызова для со-
бытия обновления окна
{
    glClear(GL_COLOR_BUFFER_BIT); // очистка окна цве-
том фона
    glBegin(GL_POINTS); // рисуем точки
    glVertex2i(100,50);
    glVertex2i(100,100); // координаты вершин
    glVertex2i(50,50);
    glEnd(); // прекращаем рисовать точки
    glFlush(); // отправляем весь вывод на дисплей
}
int main(int argc, char** argv)
{
```

```

    glutInit(&argc, argv);    // инициализирует OpenGL
Utility Toolkit

    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB); // иници-
ализирует дисплей (GLUT_SINGLE - один дисплейный бу-
фер, GLUT_RGB - задание цвета как сочетание красного,
зеленого и синего

    glutInitWindowSize(640,480); // задаем ширину и вы-
соту окна при его инициализации

    glutInitWindowPosition(100,150); // задаем коорди-
наты верхнего левого угла окна относительно верхнего
левого угла экрана

    glutCreateWindow("Primer"); // открываем и отобра-
жаем окно с указанным названием

    glClearColor(1.0, 1.0, 1.0, 0.0); // цвет фона
(красный, зеленый, синий, прозрачность)

    glColor3f(0.0,0.0,0.0);    // цвет рисования (крас-
ный, зеленый, синий)

    glPointSize(4.0);    // устанавливаем размер точки
(одна точка квадратик в 4 пиксела)

    glMatrixMode(GL_PROJECTION);    // установка
glLoadIdentity(); // простой системы координат
gluOrtho2D(0, 640.0, 0, 480.0);

    glutDisplayFunc(display); // регистрируем функцию
обновления окна

    glutMainLoop(); // входим в бесконечный главный
цикл

    return 0;
}

```

Правильные многоугольники

Многоугольник называется правильным, если он является простым, если все его стороны имеют равную длину и, если его смежные стороны образуют равные внутренние углы.

Простым называется многоугольник, если никакие две его стороны не пересекаются (только смежные стороны соприкасаются и только в их общей концевой точке).



Рисунок 1.2 - Примеры правильных n-угольников

Если число сторон n-угольника велико, то этот многоугольник по внешнему виду стремится к окружности. Вершины n-угольника лежат на так называемой порождающей окружности данного n-угольника и их расположение легко вычисляется:

$$x_i = R \cdot \cos(2\pi \cdot i / n), \text{ и } y_i = R \cdot \sin(2\pi \cdot i / n), \quad i = 0, \dots, n-1. \quad (1.1)$$

$$y_i = R \cdot \sin(2\pi \cdot i / n), \quad i = 0, \dots, n-1. \quad (1.2)$$

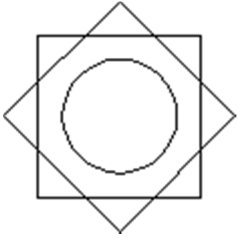

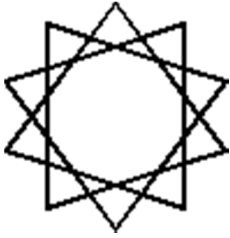

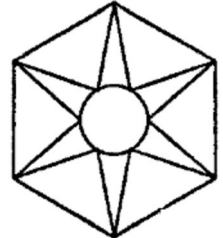
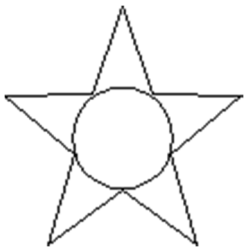


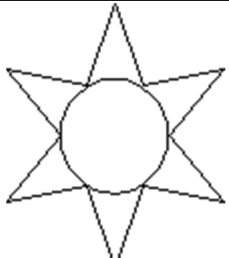
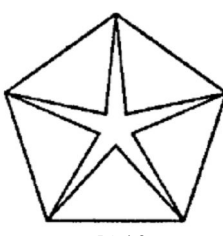
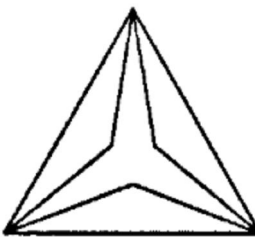

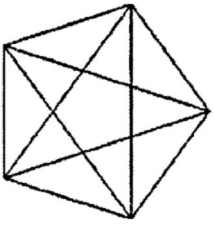
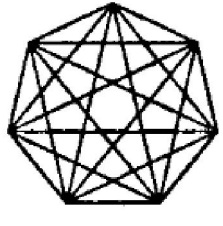
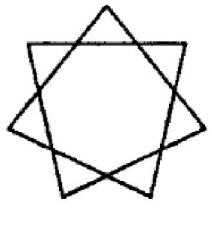
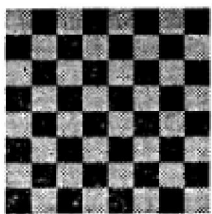



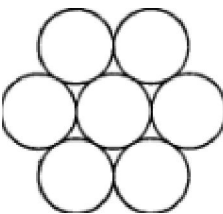
Задание

Написать и отладить программу на языке высокого уровня с использованием функциональных возможностей OpenGL для построения графической фигуры. Для этого:

1. Написать функцию для вывода на экран заданной про варианту графической фигуры (табл. 1.1). В функцию передавать минимально необходимый набор параметров (Н-р: координаты центра или другой точки фигуры и один или два радиуса, в зависимости от вида фигуры).

2. Нарисовать заданную по варианту фигуру (табл. 1.1) в центре экрана.

Таблица 1.1 – Варианты графических фигур

 №1	 №2	 №3	 №4	 №5
 №6	 №7	 №8	 №9	 №10
 №11	 №12	 №13	 №14	 №15
 №16	 №17	 №18	 №19	 №20

Содержание отчета

1. Титульный лист.
2. Задание.
3. Краткие теоретические сведения.
4. Текст программы.

При защите лабораторной работы тестирование программы **обязательно!**

Лабораторная работа № 2

Тема: мировые и экранные координаты, настройка порта просмотра

Мировые окна и порты просмотра в OpenGL

Предположим, необходимо исследовать природу некоторой математической функции, например функции sinc , хорошо известной в области обработки сигналов. Эта функция определяется формулой:

$$\text{sinc}(x) = \frac{\sin(x)}{x} \quad (2.1)$$

Необходимо знать, как эта функция изгибается и извивается по мере изменения x . Предположим, вы знаете, что при изменении x от $-\infty$ до ∞ функция $\text{sinc}(x)$ изменяется в пределах от -1 до 1 и что наиболее интересно поведение $\text{sinc}(x)$ при значениях x в окрестностях нуля. Поэтому вы хотите, чтобы график имел центром точку $(0, 0)$ и показывал функцию $\text{sinc}(x)$ для значений x , плотно расположенных в промежутке от -4,0 до 4,0. На рис. 2.1 показан график функции $\text{sinc}(x)$, созданный с помощью простой отображающей функции OpenGL (конечно, после того, как были заданы подходящие мировое окно и порт просмотра).

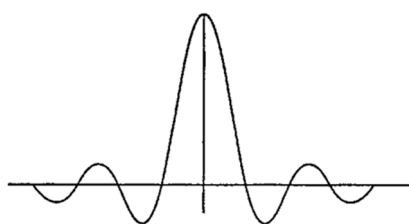


Рисунок 2.1 - График функции $\text{sinc}(x)$

Вот код этой программы:

```
void myDisplay(void)
{
    glBegin(GL_LINE_STRIP);
    for(GLfloat x = -4.0; x < 4.0; x += 0.1)
    {
```

```

        GLfloat y = sin(3.14159 * x) / (3.14159 * x);
        glVertex2f(x, y);
    }
    glEnd();
    glFlush();
}

```

Заметим, что в этом коде используется естественная система координат для решения данной задачи: задан маленький шаг по x в диапазоне от $-4,0$ до $4,0$. Главный вопрос заключается в том, каким образом следует масштабировать и сдвигать различные величины (x, y) , чтобы изображение должным образом размещалось в экранном окне.

Необходимые масштабирование и сдвиг осуществляются посредством установки мирового окна и порта просмотра и задания соответствующего преобразования между ними. Как мировое окно, так и порт просмотра являются выровненными прямоугольниками. У выровненного прямоугольника все стороны выровнены параллельно осям координат. Мировое окно выражается мировыми координатами. Порт просмотра является частью экранного окна. На рис. 2.2 приведен пример мирового окна и порта просмотра. Идея заключается в следующем: все, что находится внутри мирового окна, масштабируется и сдвигается так, чтобы оно правильно отображалось в порту просмотра; все остальное отсекается и не отображается.

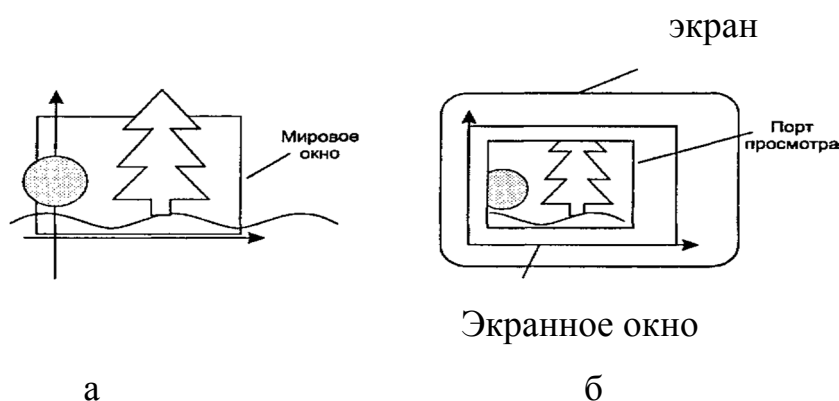


Рисунок 2.2 - Мировое окно (а) и порт просмотра (б)

В рамках OpenGL перевод из окна в порт просмотра очень прост, он автоматически осуществляет для каждой задаваемой вершины нужную последовательность преобразований с помощью команды `glVertex*()`. Кроме того, OpenGL автоматически отсекает части объекта, находящиеся за пределами мирового окна.

Для двумерного случая мировое окно устанавливается с помощью функции `gluOrtho2D()`, а порт просмотра — функцией `glViewport()`. Эти функции имеют следующие прототипы: функция `void gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top)`; устанавливает для окна левый нижний угол (`left, bottom`) и правый верхний угол (`right, top`), а функция `void glViewport(GLint x, GLint y, GLint width, GLint height)`; устанавливает для порта просмотра нижний левый угол (`x, y`) и верхний правый угол (`x + width, y + height`).

По умолчанию порт просмотра является полным экранным окном: если ширина и высота экранного окна равны соответственно `W` и `H`, то порт просмотра по умолчанию имеет левый нижний угол `(0, 0)` и верхний правый угол `(W, H)`.

Поскольку в OpenGL для настройки всех его преобразований используются матрицы, то до вызова функций `gluOrtho2D()` необходимо вызвать две «установочные» функции: `glMatrixMode(GL_PROJECTION)` и `glLoadIdentity()`. Таким образом, для настройки окна и порта просмотра из примера построения графика функции нам следует использовать следующий код:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-4.0, 2.0, 4.0, 1.0); // устанавливает
мировое окно
glViewport(0, 0, 640, 480); // устанавливает порт
просмотра
```

В дальнейшем каждая точка, отправленная в OpenGL посредством функции `glVertex2*(x, y)`, подвергается преобразованию согласно выше приведенным уравнениям, а края автоматически отсекаются границами окна.

Мы сделаем программы более читабельными, если поместим все настраивающие мировое окно команды внутрь функции `setWindowMir()`, а в функцию `setViewEkr()` все детали функции `glViewport()`, можно добавить и другие функции. Для простоты использования функции `setViewEkr()` ее параметры слегка перегруппированы, чтобы соответствовать параметрам `setWindowMir()`, так что теперь в обеих функциях они идут в таком порядке: `left, right, bottom, top` (левый, правый, нижний, верхний).

```
void setWindowMir (float left, float right, float
bottom, float top) // мировое окно
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(left, right, bottom, top); // мировое окно
}

void setViewEkr(int left, int right, int bottom,
int top ) // экранное окно
{
    glViewport(left, bottom, right-left, top-bottom);
// порт вывода
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glColor3f(0.0f, 0.0f, 0.0f);
    glPointSize(4.0);
}
```

Отметим, что для удобства мы используем для параметров функции `setWindowMir()` тип `float`. Параметры `left, right` и т. д. автоматически преобразуются к типу `GLDouble` при передаче их функции `gluOrtho2D()`, как указано в прототипе этой функции. Подобным же образом мы используем тип `int` для

параметров функции `setViewEk()`, зная, что тип аргументов функции `glViewport()` также будет правильно преобразован.

Вывод текста на экран

Для вывода текста на экран можно использовать следующие функции: `glRasterPos2f(x,y)` которая имеет следующий прототип: `void glRasterPos2f(GLfloat x, GLfloat y)` – устанавливает `x` и `y` текущей позицией растра, а для того чтобы изобразить текст в GLUT имеется команда: `void glutBitmapCharacter(void *font, int character)`, которая рисует так называемый Bitmap текст.

Для Bitmap определено большее число шрифтов (первый аргумент функции):

- GLUT_BITMAP_9_BY_15
- GLUT_BITMAP_8_BY_13
- GLUT_BITMAP_TIMES_ROMAN_10
- GLUT_BITMAP_TIMES_ROMAN_24
- GLUT_BITMAP_HELVETICA_10
- GLUT_BITMAP_HELVETICA_12
- GLUT_BITMAP_HELVETICA_18

Пример вывода на экран одиночного символа `Y` размером `8x13`:
`glutBitmapCharacter(GLUT_BITMAP_8_BY_13,'Y');`

Автоматическая установка окна и порта просмотра

Рассмотрим, как выбирать окно и порт просмотра, чтобы получить нужные изображения сцены. Как правило, не известно, где располагается интересующий объект или каковы его размеры в мировых координатах. Поэтому выбор окна удобно предоставить самому приложению. Обычно стремятся найти окно, в которое объект помещается целиком. Для этого необходимо найти обрамление данного объекта, называемое экстентом (extent). Экстент,

или ограничивающий прямоугольник (bounding box) объекта — это выровненный прямоугольник, в точности покрывающий данный объект.

Как вычислить экстент для заданного объекта? Если все конечные точки линий, из которых состоит объект, хранятся в массиве, то экстент можно определить, найдя экстремальные значения x и y в данном массиве. Например, левая сторона экстента равна минимуму из всех значений x . После того как экстент найден, окно можно положить равным ему. Если же объект задан процедурно, то не представляется возможным заранее определить его экстент. В этом случае можно прогнать рисуемую данный объект подпрограмму дважды, но по-разному:

1 прогон. Выполнить подпрограмму рисования, но фактически не рисовать, а только вычислять экстент. Затем установить окно.

2 прогон. Снова выполнить подпрограмму, уже с рисованием.

Предположим, вы хотите нарисовать фигуру наибольшего размера, которая без искажений поместится в экранном окне. Для того чтобы сделать это, необходимо задать порт просмотра с тем же форматным соотношением, какое имеет мировое окно. Обычно выбирают наибольший из портов просмотра, который поместится внутри экранного окна на дисплее.

Предположим, что форматное соотношение мирового окна известно и равно R , а экранное окно имеет ширину W и высоту H . Теперь следует рассмотреть отдельно два случая: мировое окно может иметь большее форматное соотношение, чем экранное окно ($R > W/H$), или, наоборот, меньшее форматное соотношение ($R < W/H$). Оба этих случая показаны на рис. 2.3. Мы рассмотрим их поочередно.

Случай а): $R > W/H$. Здесь мировое окно короче и толще, чем экранное окно, следовательно, порт просмотра с таким же форматным соотношением R целиком поместится по ширине экранного окна, однако сверху и/или внизу останется некоторое неиспользованное пространство. Поэтому наибольший порт просмотра будет иметь ширину W и высоту W/R и его можно задать с

помощью следующей команды (убедитесь, что этот порт просмотра действительно имеет форматное соотношение R): `setViewEkr(0, W, 0, W/R);`

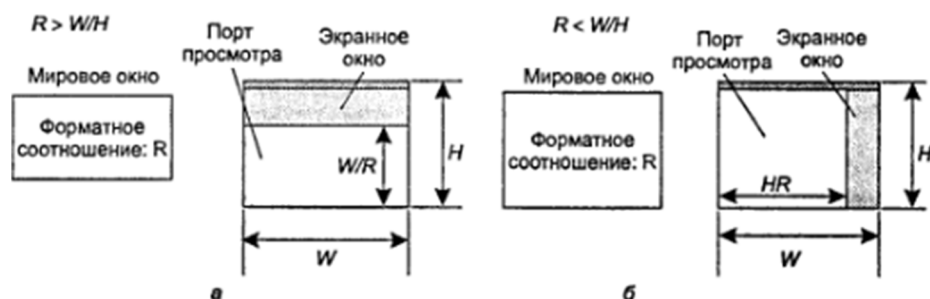


Рисунок 2.3 - Возможные форматные соотношения для мирового и экранного окон: а) $R > W/H$; б) $R < W/H$

Случай б): $R < W/H$. Здесь мировое окно выше и уже, чем экранное окно, следовательно, порт просмотра с таким же форматным соотношением R расположится сверху донизу экранного окна, однако справа и/или слева останется неиспользованное пространство. Поэтому наибольший порт просмотра будет иметь высоту H , а длину $H \cdot R$; это устанавливается следующей командой: `setViewEkr(0, H*R, 0, H);`

Изменение размеров экранного окна; событие **Resize**

В оконной системе пользователь может изменять размеры экранного окна динамически, обычно протаскиванием мышью одного из углов окна. Это действие генерирует событие `resize` (изменение размеров), на которое система может реагировать. Функция `glutReshapeFunc()` из инструментария OpenGL специфицирует функцию `myReshape`, вызываемую при возникновении данного события: `glutReshapeFunc(myReshape)`, задает функцию, вызываемую по событию `resize`. Кроме того, эта зарегистрированная функция вызывается, когда окно открывается впервые, и должна иметь следующий прототип: `void myReshape(GLsizei W, GLsizei H)`. При выполнении данной функции система автоматически передает в нее новую ширину и высоту того экранного окна, которое эта функция может затем использовать в своих вычислениях. (`GLsizei` является 32-битным целым.)

Что должна делать `myReshape()`? Если пользователь увеличивает экранное окно, можно использовать прежний порт просмотра, однако пользователь может захотеть увеличить порт просмотра, чтобы использовать преимущества увеличенного размера окна. Если же пользователь уменьшает свое экранное окно, пересекая при этом одну из границ порта просмотра, то он почти наверняка захочет пересчитать новый порт просмотра.

Обычно требуется найти новый порт просмотра так, чтобы он вписался в новое экранное окно и имел такое же форматное соотношение, что и мировое окно. Согласование форматных соотношений порта просмотра и мирового окна позволит избежать искажений в новом изображении.

Пример. Использование функции изменения формы (`reshape`) для задания наибольшего согласованного порта просмотра при обработке события `resize`.

```
Void myReshape(GLsizei W, GLsizei H)
{
    ...
    If (R > W/H)    // используем форматное соотношение
мирового окна - R
// R - можно объявить глобальной переменной или
// рассчитывать непосредственно в этой функции, пере-
дать // в функцию нельзя!
        setViewEkr(0, W, 0, W/R);
    else
        setViewEkr(0, H*R, 0, H);
}
```

Задание

Написать и отладить программу на языке высокого уровня с использованием функциональных возможностей OpenGL для построения графика функции. Для этого:

1. Нарисовать координатные оси с разметкой и подписями масштабных единиц.
2. Нарисовать заданный по варианту график функции (табл. 2.1) на всю область окна в соответствии с разметкой масштабных единиц.
3. Обеспечить масштабирование с сохранением первоначальных пропорций при изменении размеров окна (для четных вариантов рисовать по центру окна, для нечетных – в правом нижнем углу).

Содержание отчета

1. Титульный лист.
2. Задание.
3. Краткие теоретические сведения.
4. Текст программы.

Таблица 2.1 – Условия индивидуальных заданий

Вариант	Вид функции	Промежуток нахождения решения
1	$(1.85-x) \cdot \cos(3.5x-0.5)$	$x \in [-10, 10]$
2	$\cos(\exp(x)) / \sin(\ln(x))$	$x \in [2, 4]$
3	$\sin(x) / x^2$	$x \in [3.1, 20]$
4	$\sin(2x) / x^2$	$x \in [-20, -3.1]$
5	$\cos(2x) / x^2$	$x \in [-20, -2.3]$
6	$(x-1) \cos(3x-15)$	$x \in [-10, 10]$
7	$\ln(x) \cos(3x-15)$	$x \in [1, 10]$
8	$\cos(3x-15) \cdot x$	$x \in [-9.6, 9.1]$
9	$\sin(x) / (1 + \exp(-x))$	$x \in [0.5, 10]$
10	$\cos(x) / (1 + \exp(-x))$	$x \in [0.5, 10]$
11	$(\exp(x) - \exp(-x)) \cos(x) / (\exp(x) + \exp(-x))$	$x \in [-5, 5]$
12	$(\exp(-x) - \exp(x)) \cos(x) / (\exp(x) + \exp(-x))$	$x \in [-5, 5]$
13	$\cos(x-0.5) / \text{abs}(x)$	$x \in [-10, 0), (0, 10]$
14	$\cos(2x) / \text{abs}(x-2)$	$x \in [-10, 2), (2, 10]$

При защите лабораторной работы тестирование программы **обязательно!**

Лабораторная работа № 3

Тема: координатные преобразования на плоскости

Однородные координаты и матричное представление двумерных преобразований

Однородные координаты были введены в геометрии и впоследствии использованы в графике. С однородными координатами и преобразованиями над ними работают многие пакеты графических подпрограмм и некоторые дисплейные процессоры. В одних случаях эти координаты используются прикладной программой непосредственно при задании параметров для графического пакета, в других – применяются лишь внутри самого пакета и недоступны для программиста.

В однородных координатах точка $P(x, y)$ записывается как $P(W \cdot x, W \cdot y, W)$ для любого масштабного множителя $W \neq 0$. При этом если для точки задано ее представление в однородных координатах $P(X, Y, W)$, то можно найти ее двумерные декартовы координаты как $x = X/W$ и $y = Y/W$. Если W будет равно 1, операция деления не требуется.

Преимущество введения однородных координат проявляется при использовании матрицы преобразований общего вида порядка 3×3

$$\begin{bmatrix} a & b & p \\ c & d & q \\ m & n & s \end{bmatrix} \quad (3.1)$$

с помощью которой можно выполнять различные преобразования, такие как смещение, операции изменения масштаба и сдвига, обусловленные матричными элементами.

Уравнения переноса записываются в виде матрицы преобразования в однородных координатах следующим образом:

$$[x' \ y' \ 1] = [x \ y \ 1] \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Dx & Dy & 1 \end{bmatrix} \quad (3.2)$$

$$P' = P \cdot T(Dx, Dy), \quad (3.3)$$

$$T(Dx, Dy) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Dx & Dy & 1 \end{bmatrix}. \quad (3.4)$$

Уравнения масштабирования в матричной форме записываются в виде:

$$[x' \ y' \ 1] = [x \ y \ 1] \cdot \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}. \quad (3.5)$$

$$P' = P \cdot S(Sx, Sy). \quad (3.6)$$

$$S(Sx, Sy) = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad (3.7)$$

Уравнения поворота можно представить в виде:

$$[x' \ y' \ 1] = [x \ y \ 1] \cdot \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}. \quad (3.8)$$

$$P' = P \cdot R(\theta). \quad (3.9)$$

$$R(\theta) = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad (3.10)$$

Координатные преобразования в OpenGL

В целом ряде графических платформ, в том числе и в OpenGL, предусмотрен так называемый «графический конвейер» («graphics pipeline») или, иными словами, последовательность операций, применяемых ко всем точкам, которые «пропускаются» через такой конвейер. Рисунок создается путем обработки каждой точки. На рис. 3.1 показан упрощенно графический конвейер. При вызове функции `glVertex2d()` с аргументом V вершина V вначале преобразуется при помощи текущего преобразования (current transformation — CT) в точку Q , которая затем подвергается отображению «окно — порт просмотра», после чего превращается в точку S экранного окна.

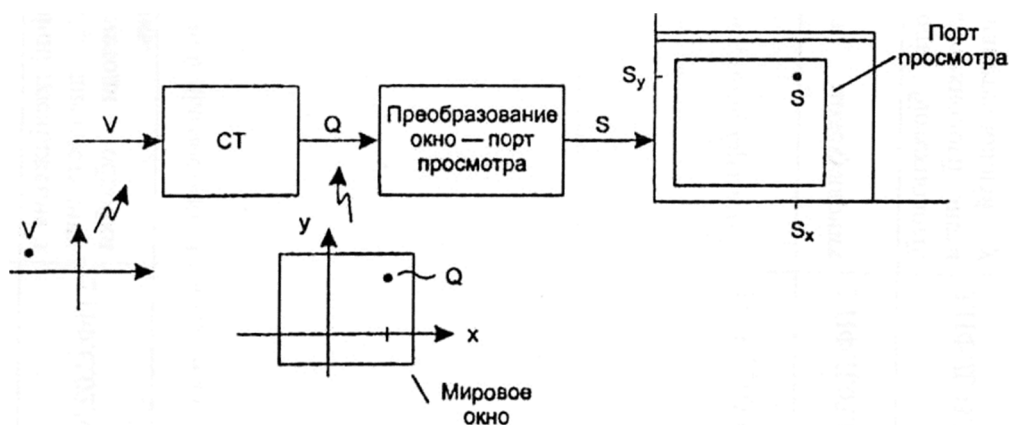


Рисунок 3.1 - Текущее преобразование, применяемое к вершинам

В OpenGL содержится так называемая матрица моделирования-вида (modelview matrix), на которую умножается каждая вершина, проходящая через графический конвейер. Чтобы выполнить координатные преобразования необходимо настроить данную матрицу на выполнение нужного преобразования.

OpenGL всегда работает в трех измерениях, поэтому его матрица моделирования-вида осуществляет 3D-преобразования. В данном случае мы работаем с матрицей моделирования-вида, ограничиваясь частным случаем двумерных преобразований. Основная идея заключается в том, что двумерное рисование производится в плоскости xu . Предполагается, что координата z равна нулю. Поэтому при преобразовании двумерных точек та часть основного трехмерного преобразования, которая воздействует на координату z , настраивается так, чтобы это воздействие полностью отсутствовало. Например, вращение вокруг начала координат в двух измерениях эквивалентно повороту вокруг оси z в трех измерениях. Далее, поскольку у трехмерного масштабирования имеется три масштабных множителя S_x , S_y , и S_z для масштабирования соответственно в направлениях x , y и z , мы задаем масштабный множитель $S_z = 1$.

Рассмотрим теперь, какие функции предусмотрены в OpenGL для моделирования и установки камеры. Ниже приводятся функции, обычно используемые для модификации матрицы моделирования-вида, которая внача-

ле должна быть сделана «текущей» посредством вызова функции `glMatrixMode(GL_MODELVIEW)`:

- `glScaled(sx, sy, sz)`; умножает справа текущую матрицу на матрицу, выполняющую масштабирование на величину sx по оси x , на sy по оси y и на sz по оси z . Результат помещается обратно в текущую матрицу.
- `glTranslated(dx, dy, dz)`; умножает справа текущую матрицу на матрицу, которая выполняет перемещение на dx по оси x , на dy по оси y и на dz по оси z . Результат помещается обратно в текущую матрицу.
- `glRotated (angle, ux, uy, uz)`; умножает справа текущую матрицу на матрицу, которая выполняет поворот на $angle$ градусов вокруг оси, которая проходит через начало координат и точку (ux, uy, uz) . Результат помещается обратно в текущую матрицу.

Основными подпрограммами для поддержки матрицы моделирования-вида являются `glRotated()`, `glScaled()` и `glTranslated()`. Они не устанавливают СТ прямо. Вместо этого в каждой из них матрица моделирования-вида СТ умножается справа на заданную матрицу, например M , после чего результат помещается обратно в СТ. Таким образом, в каждой из этих подпрограмм создается матрица M , необходимая для нового преобразования, и выполняется следующая операция:

$$СТ = СТ * M \quad (3.11)$$

Порядок следования здесь имеет значение: как мы уже видели ранее, применение преобразования $СТ * M$ к точке означает, что выполняется преобразование, задаваемое матрицей M , а затем преобразование, диктуемое прежним значением СТ. Если же рассуждать в терминах преобразования системы координат, то все это эквивалентно выполнению одного дополнительного преобразования текущей системы координат.

Ниже приведены подпрограммы OpenGL для преобразования в случае двух измерений:

- `glScaled(sx, sy, 1.0)`; умножает СТ справа на матрицу, выполняющую масштабирование на величину sx по оси x и на величину sy по оси y ;

результат помещается обратно в СТ. По оси z никакого масштабирования не производится.

- `glTranslated(dx, dy, 0);` умножает СТ справа на матрицу, выполняющую перемещение на `dx` по оси `x` и на `dy` по оси `y`; результат помещается обратно в СТ. По оси `z` никакого перемещения не производится.
- `glRotated (angle, 0, 0, 1);` умножает СТ справа на матрицу, выполняющую поворот на `angle` градусов вокруг оси `z` (она обозначена `(0,0,1)`); результат помещается обратно в СТ.

Поскольку эти подпрограммы только komponуют преобразование с СТ, то для начала процесса нам необходимо инициировать СТ для тождественного преобразования. Для этого в OpenGL содержится подпрограмма `glLoadIdentity()`. Поскольку все перечисленные функции могут быть настроены для работы с любой из поддерживаемых в OpenGL матриц, мы должны информировать OpenGL о том, какую матрицу мы изменяем. Это осуществляется посредством функции `glMatrixMode(GL_MODELVIEW)`.

Простое взаимодействие с помощью мыши и клавиатуры

Интерактивные графические приложения дают пользователю возможность управлять ходом программы с помощью естественных для человека движений: манипулированием и щелчками мышью, нажатием на различные клавиши клавиатуры. Положение мыши во время щелчка или нажатие конкретной клавиши фиксируются программой приложения и обрабатываются нужным образом.

Когда пользователь нажимает или отпускает кнопку мыши, перемещает мышь или нажимает на клавишу клавиатуры, происходит некоторое событие. С помощью инструментария OpenGL Utility Toolkit (CLUT) программист может связать с каждым из этих событий функцию обратного вызова. Это делается посредством следующих команд:

- `glutMouseFunc(myMouse)` — связывает `myMouse()` с событием, возникающим при нажатии или отпускании кнопки мыши;

– `glutKeyboardFunc(myKeyboard)` — связывает `KeyBoard()` с событием, возникающим при нажатии любой клавиши клавиатуры.

Взаимодействие с помощью мыши

Чтобы передать информацию, которая относится к мыши в приложение необходимо предусмотреть четыре параметра у функции обратного вызова `myMouse()`, так чтобы она имела следующий прототип:

```
void myMouse(int button, int state, int x, int y).
```

Когда происходит событие «мышь», система вызывает связанную с ним функцию и присваивает параметрам определенные значения. Параметр `button` должен принять одно из следующих значений: `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, `GLUT_RIGHTBUTTON` смысл этих значений очевиден (левая кнопка, средняя кнопка, правая кнопка). Параметр `state` должен быть равен `GLUTE_UP` или `GLUTE_DOWN` (вверх или вниз). Значения `x` и `y` сообщают о положении мыши в момент события. Необходимо помнить: величина `x` равна числу пикселей от левого края окна, а величина `y` равна числу пикселей вниз от верха окна.

Пример размещения точек с помощью мыши. Каждый раз, когда пользователь нажимает левую кнопку мыши, на экране рисуется точка в месте нахождения мыши. Если же пользователь нажимает правую кнопку, программа прекращает работу. Эту работу выполняет показанная ниже версия функции `myMouse()`. Отметим, что поскольку `y`-координата мыши равна числу пикселей от верха экранного окна, то необходимо рисовать точку с координатами не `(x, y)`, а `(x, screenHeight-y)`, где `screenHeight` — высота окна в пикселах. Для этого будем использовать функцию `glutGet(GLUT_SCREEN_HEIGHT)`, которая вернет текущую высоту окна (соответственно `glutGet(GLUT_SCREEN_WIDTH)`, вернет текущую ширину окна). И напишем вспомогательную функцию рисования точки.

```
void drawDot(GLint x, GLint y) // рисуем точку
{
```

```

    glBegin(GL_POINTS);
    glVertex2i(x,y);
    glEnd();
    glFlush();
}

void Mouse (int button, int state, int x, int y) //
функция обратного вызова для событий при нажатии кнопок
мышь
{ if (button==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
// если нажата левая кнопка мышь
{ draw-
Dot((GLint)x, (GLint)glutGet(GLUT_WINDOW_HEIGHT)-y); //
рисуем точки
} else
{ if (button==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
{exit(-1);}} // если нажата правая кнопка мышь то вы-
ход
}

```

Взаимодействие с помощью клавиатуры

Нажатие клавиши на клавиатуре вызывает событие клавиатуры (keyboard event). Функция обратного вызова myKeyboard() регистрируется с данным типом события посредством подпрограммы glutKeyboardFunc(myKeyboard), которая должна иметь следующий прототип:

```
void myKeyboard(unsigned char key, int x, int y);
```

Величина key определяется ASCII-кодом нажатой клавиши. Величины x и y сообщают позицию мышь в момент возникновения события (у измеряется числом пикселей вниз относительно верхней стороны окна.)

Большинство реализаций функции myKeyboard() состоит из длинного оператора switch с описанием case для каждой клавиши, представляющей интерес.

Пример размещения точек в месте нахождения мыши, по нажатию клавиши «р» на клавиатуре.

```
void Keyboard (unsigned char key, int x, int y) //
функция обратного вызова для событий при нажатии кнопок
клавиатуры; x, y текущие координаты мыши
    switch (key)
    {
        case 'p':                                     draw-
Dot((GLint)x, (GLint)glutGet(GLUT_WINDOW_HEIGHT)-y);
break;
        default : break;
    }
    }
```

Задание

I. Изменить программу для построения графика функции из лабораторной работы №2 следующим образом:

1. Обеспечить масштабирование графика путем нажатия клавиш на клавиатуре или движением мыши без изменения размеров окна.
2. Предусмотреть движение графика путем нажатия клавиш на клавиатуре или движением мыши без каких либо изменений окна.

II. Изменить программу для построения графической фигуры из лабораторной работы №1 следующим образом:

1. Обеспечить перемещение фигуры на некоторую величину. Предусмотреть перемещение по обеим осям, по каждой оси отдельно (оси рисовать не нужно!).
2. Обеспечить масштабирование фигуры на некоторую величину относительно центра фигуры. Предусмотреть масштабирование по обеим осям, по каждой оси по отдельности (оси рисовать не нужно!).

3. Обеспечить поворот на некоторый угол относительно центра фигуры. Предусмотреть поворот в обе стороны (оси рисовать не нужно!).

Содержание отчета

1. Титульный лист.
2. Задание.
3. Краткие теоретические сведения.
4. Текст программы.

При защите лабораторной работы тестирование программы **обязательно!**

Лабораторная работа № 4.

Тема: построение графических фигур в пространстве, координатные преобразования в пространстве

Установка камеры в OpenGL (для случая параллельной проекции)

Функция `glOrtho(left, right, bottom, top, near, far)`; устанавливает в качестве отображаемого объема параллелепипед, располагающийся от `left` до `right` по оси x , от `bottom` до `top` по оси y и от `-near` до `-far` по оси z . (Так как это определение работает в координатах наблюдателя, объектив камеры находится в начале координат и направлен вдоль отрицательной части оси z в сторону уменьшения значений z , то есть в сторону увеличения абсолютных величин). Эта функция создает матрицу и умножает на нее справа текущую матрицу.

Поясним знак «минус» перед величинами `near` и `far`. Так как по умолчанию камера расположена в начале координат и смотрит вдоль отрицательных значений оси z , использование для `near` значения 2 означает помещение ближней плоскости в $z = -2$, то есть на расстояние в две единицы перед глазом. Аналогично, использование 20 для `far` помещает дальнюю плоскость в 20 единицах перед глазом. Установка проекционной матрицы осуществляется с помощью следующего кода:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(left, right, bottom, top, near, far);
```

В OpenGL предлагается функция, которая упрощает установку основной камеры:

```
gluLookAt (eye_x, eye_y, eye_z, look_x, look_y,
look_z, up_x, up_y, up_z);
```

Эта функция создает матрицу просмотра и умножает на нее справа текущую матрицу. Функция принимает в качестве параметров: `eye` — положение наблюдателя (камеры) и точку, на которую направлен взгляд — `look`.

Кроме того, она принимает параметр *up* — приблизительное направление вверх. Поскольку, как правило, известно, где расположена интересующая часть сцены, обычно нетрудно выбрать соответствующие значения *eye* и *look* для обеспечения хорошего первого взгляда. При этом параметр *up* чаще всего принимается равным (0, 1, 0) для того, чтобы направление вверх было параллельно оси *y*.

Мы хотим, чтобы эта функция установила часть *V* матрицы моделирования-вида *VM*. Поэтому она запускается до того, как добавлены какие-либо преобразования моделирования, поскольку последующие преобразования будут умножаться на матрицу моделирования-вида справа. Поэтому для использования функции `gluLookAt()` необходимо применить такую последовательность команд:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eye_x, eye_y, eye_z, look_x, look_y,
look_z, up_x, up_y, up_z);
```

Камеры часто устанавливают так, чтобы они смотрели «вниз» на сцену с какой-либо ближней позиции. На рис. 4.1 показана камера, объектив которой расположен в точке *eye* = (4, 4, 4) и смотрит в начало координат, причем значение параметра *look* = (0,1,0). Направление вверх установлено параметром *up* = (0,1,0). Предположим, нам нужно, чтобы отображаемый объем имел ширину 6,4 и высоту 4,8 (при этом форматное соотношение равно 640/480), а также чтобы *near* = 1 и *far* = 50. В этом случае камеру можно установить посредством следующего кода:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-3.2, 3.2, -2.4, 2.4, 1, 50);
glMatrixMode(GL_MODELVIEW);
gluLookAt(4, 4, 4, 0, 1, 0, 0, 1, 0);
```

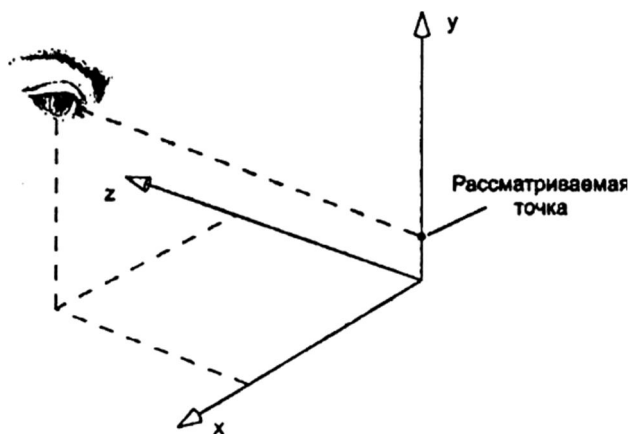


Рисунок 4.1. Установка камеры.

Установка камеры в OpenGL (для случая перспективной проекции)

Мы использовали камеру, которая осуществляет параллельные проекции. Ее отображаемый объем — это параллелепипед, ограниченный шестью стенками, включая ближнюю плоскость и дальнюю плоскость. Кроме этого, OpenGL поддерживает работу камеры, создающей перспективные виды трехмерных сцен. Во многих отношениях это похоже на камеру, использованную нами раньше, за исключением того, что ее отображаемый объем имеет другую форму.

На рис. 4.2 показан общий вид такой камеры. Она имеет глаз (eye), расположенный в некоторой точке пространства, а ее отображаемый объем (view volume) представляет собой часть четырехугольной пирамиды, вершина которой совпадает с глазом. Раствор (opening) этой пирамиды задается углом зрения. Перпендикулярно к оси пирамиды определены две плоскости: ближняя плоскость (near plane) и дальняя плоскость (far plane). Там, где эти две плоскости пересекают пирамиду, они образуют прямоугольные окна. Эти окна имеют определенное форматное соотношение (aspect ratio), которое может задаваться программно. OpenGL отсекает все те части сцены, которые располагаются вне отображаемого объема. Точки, лежащие внутри отображаемого объема, проецируются на плоскость просмотра (viewplane) в соот-

ветствующую точку P' , как показано на рис. 4.2, в. При использовании перспективной проекции точка P' задается как точка пересечения прямой, соединяющей глаз с точкой P , с плоскостью просмотра. Наконец, изображение, полученное в плоскости просмотра, преобразуется в порт просмотра, как показано на рис. 4.2, в, после чего оно становится видимым на устройстве отображения.

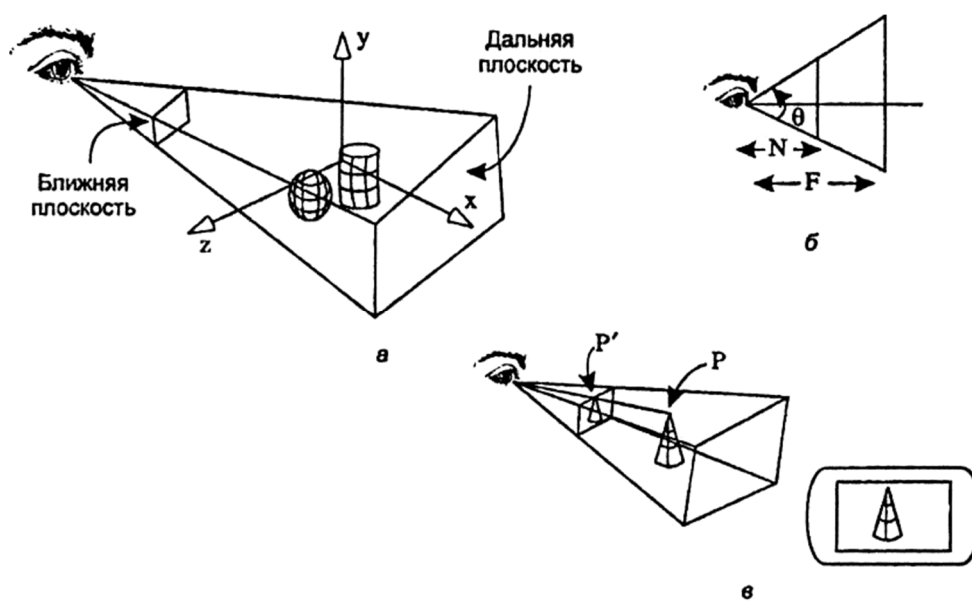


Рисунок 4.2 - Камера для создания перспективных видов сцены

На рис. 4.3 показана камера в исходной позиции, когда глаз находится в начале координат, а ось пирамиды выровнена вдоль оси z . Глаз смотрит в сторону отрицательных значений оси z .

В OpenGL предусмотрено два простых способа программной установки отображаемого объема. Вспомним, что форма отображаемого объема камеры записана в проекционной матрице (projection matrix), поступающей в графический конвейер. Эта проекционная матрица устанавливается с помощью функции `gluPerspective()` с четырьмя параметрами. Применяется такая последовательность:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(viewAngle, aspectRatio, N, F);
```

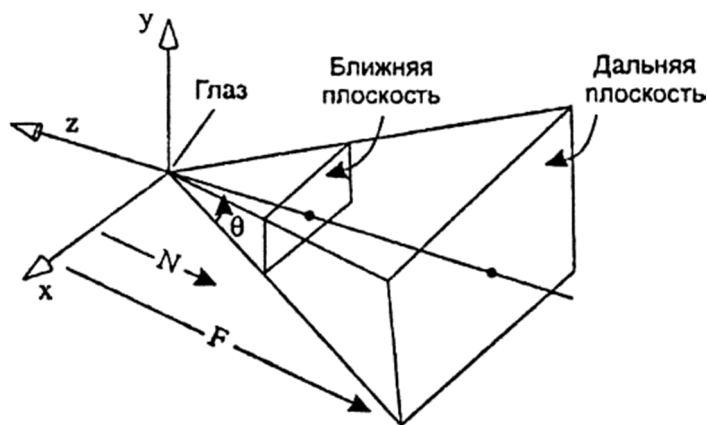


Рисунок 4.3 - Камера в исходной позиции

Параметр `viewAngle`, показанный на рисунке как угол θ , задается в градусах и определяет угол между верхней и нижней стенками пирамиды. Параметр `aspectRatio` устанавливает форматное соотношение любого окна, параллельного плоскости xu . Величина N — это расстояние от глаза до ближней плоскости, а F — расстояние от глаза до дальней плоскости. N и F должны быть положительными. Например, вызов `gluPerspective(60.0, 1.5, 0.3, 50.0)` устанавливает отображаемый объем с вертикальным раствором 60° и окном с форматным соотношением 1,5. Ближняя плоскость имеет координату $z = -0,3$, а для дальней плоскости $z = -50,0$.

`gluPerspective()` ограничена созданием только пирамид, симметричных вдоль линии обзора по x - и y -осям, но обычно это именно то, что и требуется.

Установим позицию и ориентацию камеры так, как это делалось для камеры с параллельной проекцией. (Все различие между камерами с параллельной и перспективной проекциями заключается в проекционной матрице, которая определяет форму отображаемого объема.) Проще всего вновь использовать функцию `gluLookAt()` в такой последовательности:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eye_x, eye_y, eye_z, look_x, look_y,
look_z, up_x, up_y, up_z);
```

Как и прежде, в результате выполнения этого кода камера передвинется так, что ее глаз расположится в точке `eye` и будет направлен в интересующую нас точку `look`. Направление «вверх» обычно задается вектором `up`, который чаще всего равен $(0,1,0)$.

Второй способ определения объема видимости в форме усеченной пирамиды `glFrustum()`, данная команда вычисляет матрицу, выполняющую перспективное проецирование, и умножает на нее текущую матрицу проекции (обычно единичную). `glFrustum()` не требует от вас указания симметричного объема видимости.

```
Void glFrustum (GLdouble left, GLdouble right, GLdouble
bottom, GLdouble top, GLdouble near, GLdouble
far);
```

Создает матрицу перспективного проецирования и умножает на нее текущую матрицу. Объем видимости задается параметрами (`left`, `bottom`, `-near`) и (`right`, `top`, `-near`) определяющими координаты (x , y , z) левого нижнего и правого верхнего углов ближней отсекающей плоскости; `near` и `far` задают дистанцию от точки наблюдения до ближней и дальней отсекающих плоскостей (они всегда должны быть положительными).

Рисование элементарных форм, поддерживаемых OpenGL

В инструментарии GLUT предусмотрено несколько готовых объектов, включая сферу, конус, тор, пять Платоновых тел, а также чайник. Каждый такой объект доступен в виде каркасной модели и в виде объемной модели с гранями, которые могут быть закрашены. Ниже приводится список, показывающий, какие функции используются для рисования некоторых из этих объектов:

- куб: `glutWireCube(GLdouble size)`; каждая сторона имеет длину `size`;
- сфера: `glutWireSphere(GLdouble radius, GLint nSlices, GLint nStacks)`;
- тор: `glutWireTorus(GLdouble inRad, GLdouble outRad, GLint nSlices, GLint nStacks)`;

- чайник: `glutWireTeapot(GLdouble size)`.

Существуют также функции `glutSolidCube()`, `glutSolidSphere()` и другие. Форма тора задается с помощью внутреннего радиуса `inRad` и внешнего радиуса `outRad`. Сфера и тор аппроксимируются полигональными гранями, поэтому путем изменения параметров `nSlices` и `nStacks` можно устанавливать, сколько граней будет использовано в аппроксимации. Параметр `nSlices` — это число «долек» вокруг оси z , а `nStacks` — число «ломтиков» вдоль оси z , как если бы данная форма была стопкой из множества `nStacks` дисков.

Для визуализации четырех из Платоновых тел (пятым является куб, который уже был представлен) используются следующие функции:

- тетраэдр: `glutWireTetrahedron()`;
- октаэдр: `glutWireOctahedron()`;
- додекаэдр: `glutWireDodecahedron()`;
- икосаэдр: `glutWireIcosahedron()`.

Все вышеприведенные формы имеют центр в начале координат. Кроме них, имеется еще конус:

`glutWireCone(GLdouble baseRad, GLdouble height, GLint nSlices, GLint nStacks);`

Оси конуса совпадают с осью z . Их основания находятся в плоскости $z = 0$ и простираются вдоль оси z до плоскости $z = \text{height}$. Радиусы конуса и усеченного конуса при $z = 0$ задаются переменной `baseRad`. Радиус усеченного конуса при $z = \text{height}$ равен `topRad`.

Так же есть формы библиотеки `glu`, но для их рисования необходимо выполнить дополнительные действия, например:

```
GLUQuadricObj * qobj =gluNewQuadric(); // создаем
квадратичный объект
gluQuadricDrawStyle(qobj, GLU_LINE); // устанавли-
ваем стиль каркасной модели
gluCylinder(qobj, baseRad, topRad, height, nSlices,
nStacks); // рисуем цилиндр.
```

```
gluDeleteQuadric(qObj); // Освобождается память,
которую занимал квадратичный объект
```

Настройка квадратичных состояний заключается в следующем:

```
GLUQuadricObj *pObj; // Создается и инициализиру-
ется
pObj = gluNewQuadric(); // квадратичная поверхность
...
// Задаются параметры визуализации квадратичных по-
верхностей
// Рисуются поверхности
...
gluDeleteQuadric(pObj); // Освобождается память,
которую занимала квадратичная поверхность
```

Обратите внимание на то, что создается указатель на тип данных `GLUQuadricObj`, а не на экземпляр самой структуры данных. Это объясняется тем, что функция `gluNewQuadric` не только выделяет память для структуры, но и инициализирует элементы структуры со значениями по умолчанию.

Для модификации состояния рисования объекта `GLUQuadricObj` и, соответственно, поверхностей, рисуемых с его помощью, предназначены четыре функции. Первая функция устанавливает стиль рисования квадратичной поверхности.

```
void gluQuadricDrawStyle(GLUquadricObj *obj, GLenum
drawStyle);
```

Первый параметр является указателем на объект второго порядка, а параметр `drawStyle` имеет одно из значений, перечисленных в табл. 4.1.

Следующая функция задает, будет ли геометрия квадратичной поверхности генерироваться с помощью нормалей.

```
void gluQuadricNormals(GLUquadricObj *pbj, GLenum
normals);
```

Квадратичные поверхности могут рисоваться без нормалей (`GLU_NONE`), с гладкими нормальями (`GLU_SMOOTH`) или с плоскими нор-

малями (GLU_FLAT). Основное отличие гладких нормалей от плоских заключается в том, что в первом случае задаются нормали для каждой вершины поверхности, что позволяет получить сглаженный внешний вид. При использовании плоских нормалей задается одна нормаль для всех вершин ячейки (треугольника) поверхности.

Таблица 4.1 - Стили рисования объектов второго порядка

Константа	Описание
GLU_FILL	Квадратичные объекты рисуются как сплошные
GLU_LINE	Квадратичные объекты рисуются как каркасные
GLU_POINT	Квадратичные объекты рисуются как набор точек-вершин
GLU_SILHOUETTE	Похоже на каркасное изображение, только смежные грани многоугольников не рисуются

Кроме того, можно задать, указывает нормаль наружу или внутрь от поверхности. Например, при рисовании окрашенной сферы естественно положить, что нормали направлены наружу. Если же вы рисуете внутреннюю часть сферы (например, часть куполообразного потолка), удобно, чтобы освещение и нормали относились к внутренней части сферы. Данный параметр устанавливается с помощью следующей функции.

```
void gluQuadricOrientation(GLUquadricObj *obj, GLenum orientation);
```

Значением параметра orientation может быть GLU_OUTSIDE либо GLU_INSIDE. По умолчанию квадратичные поверхности обходятся против часовой стрелки, и передние грани направлены наружу от поверхности. Значение термина «наружная поверхность» для сфер и цилиндров понятно, у дисков так называется сторона, направленная по положительному направлению оси z.

Вызвав указанную ниже функцию, можно также затребовать для поверхностей второго порядка генерацию текстурных координат:

```
void gluQuadricTexture (GLUquadricObj *obj, GLenum
textureCoords);
```

Параметр `textureCoords` может иметь значение `GL_TRUE` либо `GL_FALSE`. При генерации текстурных координат для квадратичных поверхностей текстуры равномерно оборачиваются вокруг сфер и цилиндров, при наложении текстуры на диск центр текстуры совмещается с центром диска, а края текстуры выравниваются по краям диска.

Рисование поверхностей второго порядка. Задав состояние объекта второго порядка, поверхность можно нарисовать, вызвав единственную функцию. Например, чтобы нарисовать сферу, вызывается такая функция:

```
void gluSphere(GLUquadricObj *obj, GLdouble radius,
GLint slices, GLint stacks);
```

Первый параметр `obj` является просто указателем на квадратичный объект, ранее установленный для желаемого состояния визуализации. Параметр `radius` является радиусом сферы, после него указывается число секторов и слоев. Сферы рисуются с помощью кольцеобразных полос треугольников (или квадратов, это зависит от используемой библиотеки GLU), уложенных штабелем снизу-вверх. Число ломтиков задает, сколько наборов треугольников (или квадратов) применяется для совершения полного оборота вокруг сферы. Описанное представление сферы похоже на условное разделение глобуса линиями широты и долготы.

Квадратичные сферы рисуются так, что положительное направление оси z указывает на вершину сферы. На рис. 4.4 для примера показана каркасная квадратичная сфера, нарисованная возле единичных осей.

Цилиндры также рисуются вдоль положительного направления оси z и состоят из набора полосок, уложенных штабелем. Ниже приведена функция изображения цилиндра:

```
void gluCylinder(GLUquadricObj *obj, GLdouble
baseRadius, GLdouble topRadius, GLdouble height, GLint
slices, GLint stacks);
```

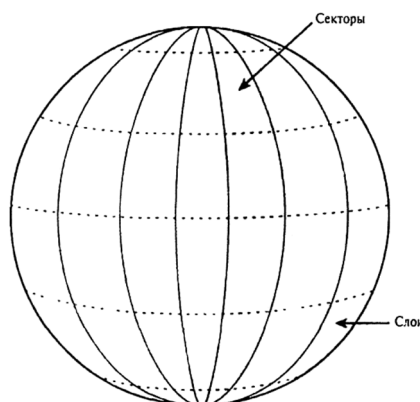


Рисунок 4.4 - Секторы и слои квадратичной сферы

С помощью этой функции можно задавать радиус основания (возле начала координат) и радиус вершины (по положительному направлению оси z). Параметр `height` просто определяет длину цилиндра, если параметр `topRadius` задать равным нулю, то в результате получается конус.

Последней поверхностью второго порядка является диск. Диски рисуются с помощью колец полос треугольников или квадратов, разделенных на несколько ломтиков. Для визуализации диска применяется такая функция:

```
void gluDisk(GLUquadricObj*obj, GLdouble innerRadius,
GLdouble outerRadius, GLint slices, GLint loops);
```

Чтобы нарисовать диск, задается внутренний и внешний радиус. Если внутренний радиус равен 0, получается сплошной диск. Ненулевой радиус определяет отверстие в шайбе. В обоих случаях диск рисуется в плоскости xy .

Задание

Написать и отладить программу на языке высокого уровня с использованием функциональных возможностей OpenGL для рисования трехмерных сцен и реализации координатных преобразований в пространстве. Для этого:

1. Нарисовать заданное по варианту тело (табл. 4.2, столбец 2) из библиотеки `glut`. Для четных вариантов использовать параллельную проекцию для нечетных – перспективную.

2. Нарисовать заданное по варианту тело (табл. 4.2, столбец 3) из библиотеки glu. Для четных вариантов использовать перспективную проекцию для нечетных – параллельную.
3. Выполнить:
 - а. перемещение тела;
 - б. масштабирование фигуры;
 - с. поворот на заданный угол относительно каждой из осей.

Таблица 4.2 - Варианты заданий

№ варианта	Тело (glut)	Тело (glu)
1	Куб	Цилиндр
2	Сфера	Диск
3	Тор	Сфера
4	Чайник	Цилиндр
5	Конус	Диск
6	Усеченный конус	Сфера
7	Тетраэдр	Цилиндр
8	Октаэдр	Диск
9	Додекаэдр	Сфера
10	Икосаэдр	Цилиндр
11	Сфера	Диск
12	Куб	Сфера
13	Тор	Цилиндр
14	Чайник	Диск
15	Конус	Сфера

Содержание отчета

1. Титульный лист.
2. Задание.
3. Краткие теоретические сведения.
4. Текст программы.

При защите лабораторной работы тестирование программы **обязательно!**

Лабораторная работа № 5

Тема: растровые алгоритмы построения контура фигур

Алгоритм Брезенхема рисования линии

Пусть необходимо построить отрезок начало которого имеет координаты (x_1, y_1) , а конец (x_2, y_2) . Обозначим $dx = (x_2 - x_1)$, $dy = (y_2 - y_1)$. Не нарушая общности, будем считать, что начало отрезка совпадает с началом координат, и прямая имеет вид $y = \frac{dy}{dx}x$, где $\frac{dy}{dx} \in [0, 1]$, т.е. $dy < dx$. Считаем, что начальная точка находится слева. Пусть на $(i-1)$ -м шаге текущей точкой отрезка является $P_{i-1} = (r, q)$. Необходимо сделать выбор следующей точки $S_i(r+1, q+1)$ или $T_i(r+1, q)$.

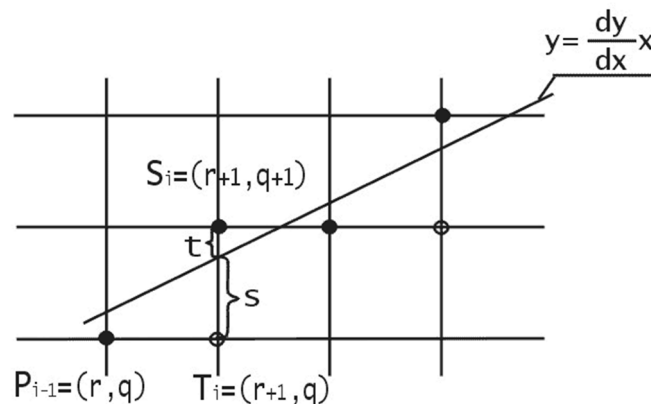


Рисунок 5.1 - Рисование отрезков прямых по методу Брезенхема

1. Рассчитаем $dx = (x_2 - x_1)$ и $dy = (y_2 - y_1)$.
2. $i = 1$, $d_i = 2dy - dx$, $y_i = y_1$, $x_i = x_1$.
3. Рисуем точку с координатами (x_i, y_i) .
4. Если $d_i < 0$, тогда $d_{i+1} = d_i + 2dy$ и $y_{i+1} = y_i$. Если $d_i \geq 0$, то $d_{i+1} = d_i + 2(dy - dx)$ и $y_{i+1} = y_i + 1$.
5. $x_{i+1} = x_i + 1$; $i = i + 1$.
6. Если $x_i < x_2$, то перейти на пункт 3, иначе – конец.

Если $dy > dx$, то необходимо будет использовать этот же алгоритм, но пошагово увеличивая y ($y_{i+1} = y_i + 1$) и на каждом шаге вычислять x ($x_i = x_i + 1$ или $x_i = x_i$), пока $y_i < y_2$.

Реализация этого алгоритма может выглядеть следующим образом:

```
void drawDot(GLint x, GLint y) // рисуем точку
{
    glBegin(GL_POINTS);
    glVertex2i(x,y);
    glEnd();
    glFlush();
}
```

```
void line(GLint x1, GLint y1, GLint x2, GLint y2)
// алгоритм Брезенхема для отрезка
{
    int dy=abs(y2-y1);
    int dx=abs(x2-x1);
    int tmp_x, tmp_y,x,y, flag;

    if (dx==0) { // вертикальная линия
        if (y2<y1) {tmp_y=y2;y2=y1; y1=tmp_y;}
        for (y=y1;y<=y2;y++)
            drawDot(x1,y);
        return;
    }

    if (dy==0) { // горизонтальная линия
        ...
    }
```

```

if (dy<dx) {                                     // наклонная линия
flag=0;
if (x2<x1)
{tmp_x=x2;    x2=x1;    x1=tmp_x;    tmp_y=y2;y2=y1;
y1=tmp_y;}
if (y2<y1){flag=1;}
int di;
di=2*dy-dx;
y=y1;
x=x1;
do {
    drawDot(x,y);
    if (di<0) {di=di+2*dy; x++;}
    else {di=di+2*(dy-dx); x++; if (flag==0) {y++;}
else {y--;}}
    } while (x<x2);
return;
    }

if (dy>dx){..... // наклонная линия
...
    }

if (dx==dy) // диагональ
{
    ...
}

}

```

Растровая развёртка окружности

Для упрощения алгоритма растровой развёртки стандартной окружности можно воспользоваться её симметрией относительно координатных осей и прямых $y = \pm x$; в случае, когда центр окружности не совпадает с началом координат, эти прямые необходимо сдвинуть параллельно так, чтобы они прошли через центр окружности. Тем самым достаточно построить растровое представление для 1/8 части окружности, а все оставшиеся точки получить симметрией (рис. 5.2).

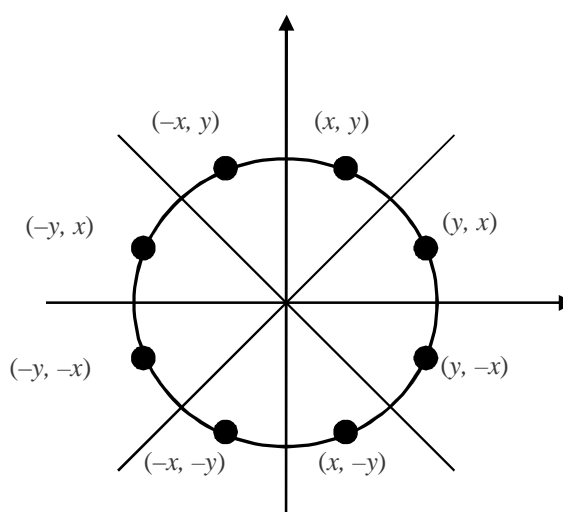


Рисунок 5.2 – Восьмисторонняя симметрия

Если точка (x, y) лежит на окружности, то легко вычислить семь точек, принадлежащих окружности, симметричных этой. То есть, имея функцию вычисления значения y по $x=0\dots R/\sqrt{2}$ для построения дуги от 0° до 45° , можно реализовать функцию, которая будет по координате одной точки рисовать сразу восемь точек, учитывая симметрию окружности.

```
void cir_pix(int x0, int y0, int x, int y) // 8
точек на окружности
{
    drawDot(x0+x, y0+y);
    drawDot(x0+y, y0+x);
```



```

drawDot (x0+x, y0-y) ;
drawDot (x0+y, y0-x) ;
drawDot (x0-x, y0-y) ;
drawDot (x0-y, y0-x) ;
drawDot (x0-x, y0+y) ;
drawDot (x0-y, y0+x) ;
}

```

Алгоритм Брезенхейма для окружности

Рассмотрим участок окружности $x \in [0, R/\sqrt{2}]$ (рис. 5.2). На каждом шаге алгоритм выбирает точку $P_i(x_i, y_i)$, которая является ближайшей к истинной окружности. Идея алгоритма заключается в выборе ближайшей точки при помощи управляющих переменных, значения которых можно вычислить в пошаговом режиме с использованием небольшого числа сложений, вычитаний и сдвигов.

Рассмотрим небольшой участок сетки пикселей, а также возможные способы (от А до Е) прохождения истинной окружности через сетку (рис. 5.3).

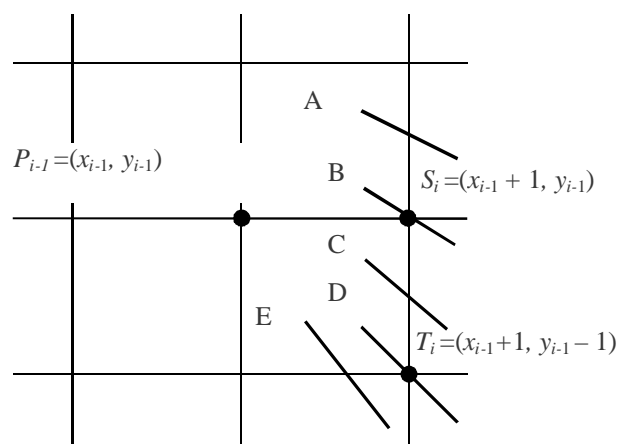


Рисунок 5.3 – Варианты прохождения окружности через растровую сетку

Предположим, что точка P_{i-1} была выбрана как ближайшая к окружности при $x = x_{i-1}$. Теперь найдем, какая из точек ($S_i(x_{i-1} + 1, y_{i-1})$ или $T_i(x_{i-1} + 1, y_{i-1} - 1)$) расположена ближе к окружности при $x = x_{i-1} + 1$.

Алгоритм формулируется следующим образом. Дано (x_0, y_0) – центр окружности, R – радиус.

1. Берем первую точку $i=1$, $x_i = 0$, $y_i = R$, рассчитываем $d_i = 3 - 2R$.
2. Вызов функции рисования 8-ми точек на окружности по координатам одной точки, учитывая смещение центра окружности от начала координат.
3. Если $d_i < 0$, то $d_{i+1} = d_i + 4x_i + 6$, $x_{i+1} = x_i + 1$, $y_{i+1} = y_i$.
4. Если $d_i \geq 0$, то $d_{i+1} = d_i + 4(x_{i-1} - y_{i-1}) + 10$, $x_{i+1} = x_i + 1$, $y_{i+1} = y_i - 1$.
5. $i = i + 1$.
6. Если $x_i > R/\sqrt{2}$, то перейти на пункт 2, иначе – конец.

Растровая развёртка эллипса

Простым методом растровой развертки эллипса является использование вычислений x и y по формулам:

$$x = R_x \cos \alpha, \quad (5.1)$$

$$y = R_y \sin \alpha \quad (5.2)$$

при пошаговом изменении угла α от 0° до 360° .

Можно также использовать симметрию эллипса. Существует модификация алгоритма Брезенхейма для эллипса.

Кривая Безье

Разработана математиком Пьером Безье. Кривые и поверхности Безье были использованы в 60-х годах компанией «Рено» для компьютерного проектирования формы кузовов автомобилей. В настоящее время они широко используются в компьютерной графике.

Кривые Безье описываются в параметрической форме:

$$x = P_x(t), \quad y = P_y(t). \quad (5.3)$$

Значение t выступает как параметр, которому соответствуют координаты отдельной точки линии. Параметрическая форма описания может быть удобнее для некоторых кривых, чем задание в виде функции $y = ?(x)$, поскольку функция $?(x)$ может быть намного сложнее, чем $P_x(t)$ и $P_y(t)$, кроме того, $?(x)$ может быть неоднозначной.

Многочлены Безье для P_x и P_y имеют такой вид:

$$P_x(t) = \sum_{i=0}^m C_m^i t^i (1-t)^{m-i} x_i, \quad P_y(t) = \sum_{i=0}^m C_m^i t^i (1-t)^{m-i} y_i, \quad (5.4)$$

где x_i и y_i — координаты точек-ориентиров P_i , а величины C_m^i — это известные из комбинаторики, так называемые сочетания (они также известны как коэффициенты бинома Ньютона):

$$C_m^i = \frac{m!}{i!(m-i)!}. \quad (5.5)$$

Значение m можно рассматривать и как степень полинома, и как значение, которое на единицу меньше количества точек-ориентиров.

Рассмотрим кривые Безье, классифицируя их по значениям m .

1. $m = 1$ (по двум точкам)

Кривая вырождается в отрезок прямой линии, которая определяется конечными точками P_0 и P_1 , как показано на рис. 5.4:

$$P(t) = (1-t)P_0 + tP_1 \quad (5.6)$$

2. $m = 2$ (по трем точкам), рис. 5.4:

$$P(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2. \quad (5.7)$$

3. $m = 3$ (по четырем точкам), кубическая (рис. 5.5), используется довольно часто, в особенности в сплайновых кривых:

$$P(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t) P_2 + t^3 P_3. \quad (5.8)$$

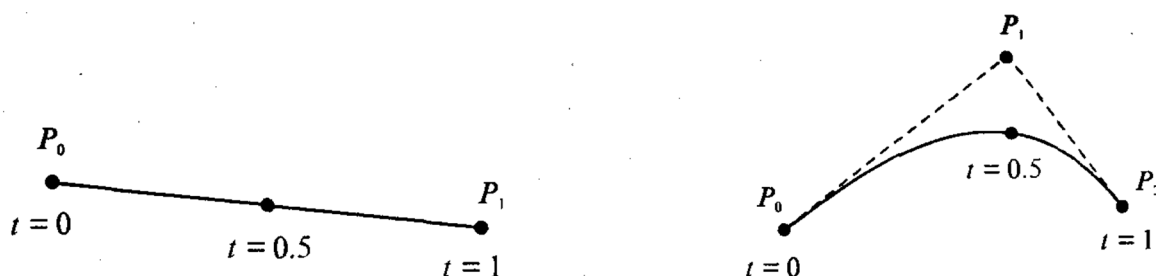


Рисунок 5.4 – Кривая Безье ($m=1$) и кривая Безье ($m=2$)

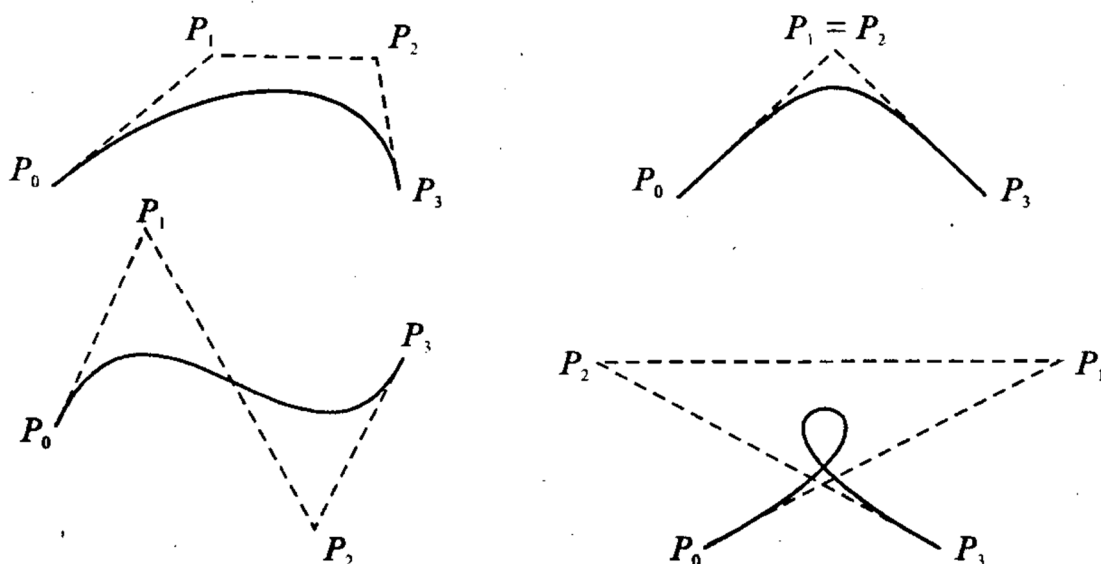


Рисунок 5.5 – Кубические кривые Безье ($m=3$)

Геометрический алгоритм для кривой Безье

Этот алгоритм позволяет вычислить координаты (x, y) точки кривой Безье по значению параметра t .

1. Каждая сторона контура многоугольника, который проходит по точкам-ориентирам, делится пропорционально значению t .
2. Точки деления соединяются отрезками прямых и образуют новый многоугольник. Количество узлов нового контура на единицу меньше, чем количество узлов предшествующего контура.
3. Стороны нового контура снова делятся пропорционально значению t . И так далее. Это продолжается до тех пор, пока не будет получена единственная точка деления. Эта точка и будет точкой кривой Безье (рис.5.6).

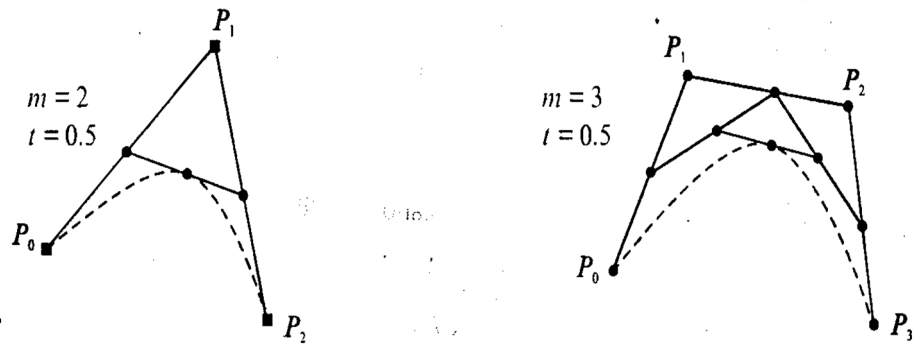


Рисунок 5.6 – Геометрический алгоритм для кривых Безье

Пример на языке C++:

```
int m=3; // степень полинома
int n=6; // количество точек на кривой
float t;
for (int l=0; l<=n; l++)
{
    t=(1.0/n)*l;
    for (int i=0; i<2; i++)
        for (int j=0; j<=m; j++)
            R[i][j]=(P[i][j]); // вспомогательный
//массив дублирует координаты точек - ориентиров.
    for (int i=0; i<2; i++) // координаты x и y
        for (int j=m; j>=0; j--)
            for (int k=0; k<j; k++)
                { R[i][k]+= t*(R[i][k+1]-R[i][k]);
                  T[i][l]=R[i][k] ;//координаты точек на
кривой
                }
}
```

Уравнения кривой Безье можно записать в матричном виде:

$$P(t) = [T][N][G] \quad (5.9)$$

где для $m = 3$:

$$P(t) = [T][N][G] = [t^3 \ t^2 \ t \ 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}. \quad (5.10)$$

Матрица N представляет собой коэффициенты при соответствующих t (Первый столбец при P_0 , второй – при P_1 , третий – при P_2 и четвертый – при P_3 . Первая строка при t^3 , вторая – при t^2 , третья – при t и четвертая – свободный член). P_0, P_1, P_2 и P_3 – вершины многоугольника Безье.

Аналогично для $m = 4$:

$$P(t) = [t_4 \ t_3 \ t_2 \ t \ 1] \begin{bmatrix} 1 & -4 & 6 & -4 & 1 \\ -4 & 12 & -12 & 4 & 0 \\ 6 & -12 & 6 & 0 & 0 \\ -4 & 4 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix} \quad (5.11)$$

Сплайн Эрмита

Одним из способов задания параметрического кубического сплайна является указание координат начальной и конечной точек, а также векторов касательных в них. Такой способ задания называется формой Эрмита. Обозначим концевые точки P_1 и P_4 , а касательные векторы в них R_1 и R_4 .

$$M_h G_{hx} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_{1x} \\ P_{4x} \\ R_{1x} \\ R_{4x} \end{bmatrix} \quad (5.12)$$

Здесь M_h – Эрмитова матрица, G_h – геометрический вектор Эрмита. в матричном виде $x(t)$: $x(t) = TM_h G_{hx}$. Аналогично для остальных координат: $y(t) = TM_h G_{hy}$, $z(t) = TM_h G_{hz}$.

Форму кривой, заданной в форме Эрмита, легко изменять если учитывать, что направление вектора касательной задает начальное направление, а модуль вектора касательной задает степень вытянутости кривой в направлении этого вектора, как показано на рис. 5.7.

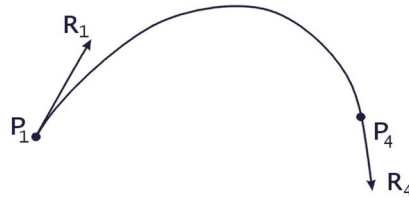


Рисунок 5.7 – Параметрический сплайн в форме Эрмита. Вытянутость кривой вправо обеспечивается тем, что $|R_1| > |R_4|$.

В-сплайны

Кривая, представленная в виде кубического В-сплайна, в общем случае может проходить через любые управляющие точки, однако она непрерывна и, кроме того, непрерывностью изменения обладают ее касательный вектор и кривизна (т. е. первая и вторая производные кривой непрерывны в конечных точках) в отличие от формы Безье, у которой в конечных точках непрерывны лишь первые производные (но которые проходят через управляющие точки). Таким образом, можно утверждать, что форма В-сплайнов «более гладкая», чем другие формы. В-сплайн описывается следующей формулой:

$$x(t) = TM_s G_{sx}, \quad (5.13)$$

где

$$M_s = \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}. \quad (5.14)$$

При аппроксимации управляющих точек P_1, P_2, \dots, P_n последовательно В-сплайнов необходимо применять между каждой парой соседних точек геометрические матрицы. Для аппроксимации в интервале, близком к точкам P_i и P_{i+1} , используется:

$$G_s^i = \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix}, \quad 2 \leq i \leq n-2. \quad (5.15)$$

Пример построения сплайна

```
// построение сплайна
...
float Dx[4][1]={ {300}, // точки ориентиры для
                 {250}, // кривой Безье/ В-сплайна
                 {250}, // (координаты x)
                 {300}};
float Dy[4][1]={ {200}, // ---//----
                 {250}, // (координаты y)
                 {150},
                 {200}};
for(float i=0; i<=1;i+=0.01)
    poit_b(i,&Dx[0][0],&Dy[0][0]); // ваша функция для
рисования одной точки на сплайне
...
```

Пример 1. Построение дерева (рис. 5.8)

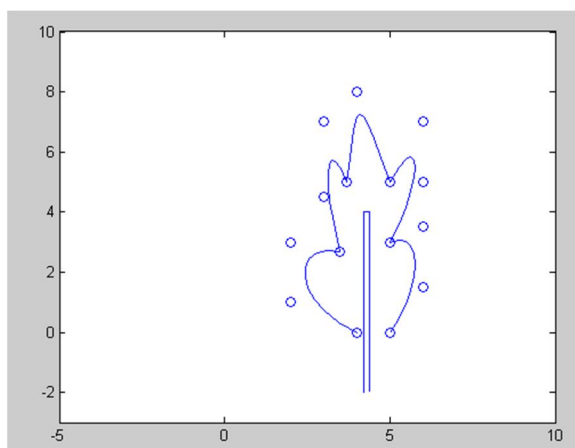


Рисунок 5.8 – Пример рисунка – дерево.

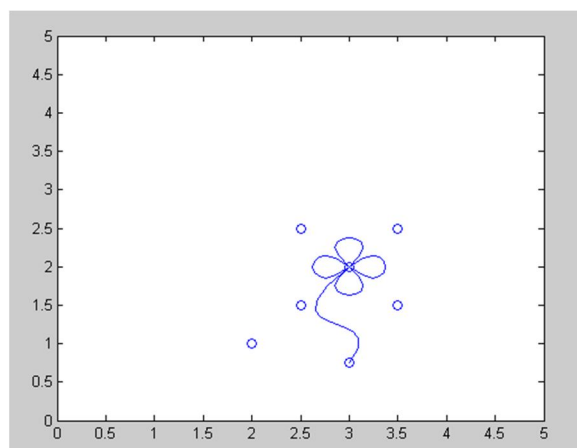


Рисунок 5.9 – Пример рисунка – цветок.

Точки – ориентиры для сегментов кривой Безье, сегменты нумеруются по часовой стрелке:

- 1) P= 4 0
 2 1
 2 3
 3.5 2.7
- 2) P= 3.5 2.7
 3 4.5
 3 7
 3.7 5
- 3) P= 3.7 5
 4 8
 4 8
 5 5
- 4) P= 5 5
 6 7
 6 5
 5 3
- 5) P= 5 3
 6 3.5
 6 1.5
 5 0

Пример 2. Построение цветка (рис. 5.9).

Точки – ориентиры для сегментов кривой Безье:

- 1) P= 3 2
 2.5 2.5
 2.5 1.5
 3 2
- 2) P= 3 2
 3.5 2.5
 3.5 1.5
 3 2
- 3) P= 3 2
 2.5 2.5
 3.5 2.5
 3 2
- 4) P= 3 2
 2.5 1.5
 3.5 1.5
 3 2

Стебель:

$$\begin{array}{rcl}
 5) \ P = & 3 & 0.75 \\
 & 3.5 & 1.5 \\
 & 2 & 1 \\
 & 3 & 2
 \end{array}$$

Для кривой Безье функция `void poit_b(float t, float *px, float *py)` может реализовывать один из трех алгоритмов: матричный, геометрический или параметрический. Для В-сплайнов – матричный алгоритм.

Задание

Написать и отладить программу на языке C/C++ для вывода контура рисунка по варианту (табл.5.1). Для этого:




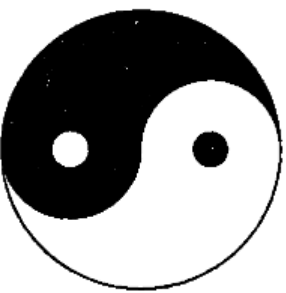



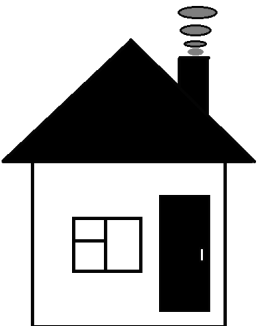


1. Написать необходимые функции для реализации растровой развертки графических примитивов:
 - отрезок (алгоритмом Брезенхема);
 - окружность (алгоритмом Брезенхема);
 - эллипс (можно параметрический);
 - сплайновые кривые (Безье и / или В-сплайны (сплайн Эрмита по желанию)).
2. Нарисовать заданный по варианту рисунок (**только контур**).


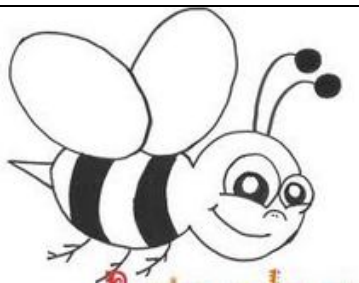



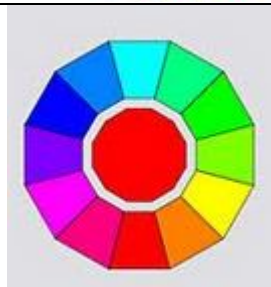
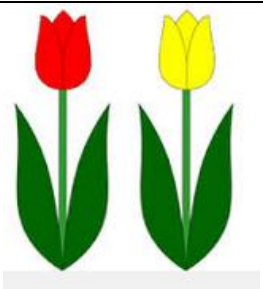

Содержание отчета

1. Титульный лист.
2. Задание.
3. Краткие теоретические сведения.
4. Текст программы.

При защите лабораторной работы тестирование программы **обязательно!**

Таблица 5.1 – Варианты индивидуальных заданий

№ вар.	Рисунок	№ вар.	Рисунок
1		2	
3		4	
5		6	
7		8	
9		10	

№ вар.	Рисунок	№ вар.	Рисунок
11		12	
13		14	
15		16	
17		18	

Лабораторная работа № 6

Тема: растровые алгоритмы заполнения фигур

Растровое заполнение круга и полигона

Основная идея – закрашивание фигуры отрезками прямых линий. Удобней использовать горизонтали. Алгоритм представляет собою цикл вдоль оси y , в ходе этого цикла выполняется поиск точек пересечения линии контура с соответствующими горизонталями.

Заполнение круга. Для заполнения круга можно использовать алгоритм вывода контура. В процессе выполнения этого алгоритма последовательно вычисляются координаты пикселей контура в границах одного октанта (например, алгоритмом Брезенхема). Для заполнения следует выводить горизонтали, которые соединяют пары точек на контуре, расположенные симметрично относительно оси y (рис. 6.1).

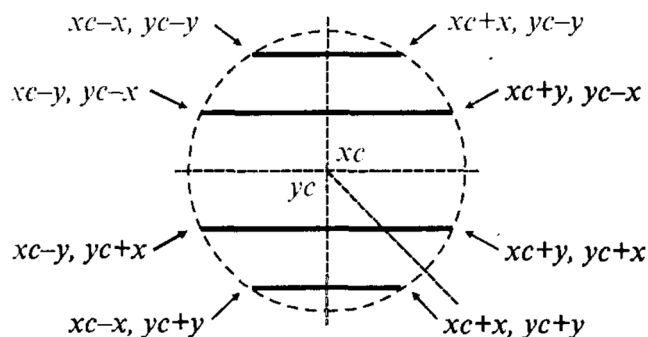


Рисунок 6.1. – Заполнение круга

Аналогично можно разработать алгоритм заполнения эллипса.

Заполнение полигонов. Контур полигона определяется вершинами, которые соединены отрезками прямых – ребрами (рис.6.2).

При нахождении точек пересечения горизонтали с контуром необходимо принимать во внимание особые точки. Если горизонталь имеет координату (y), совпадающую с координатой y_i вершины P_i тогда надлежит анализировать то, как горизонталь проходит через вершину. Если горизонталь при этом пересекает контур, как, например, в вершинах P_0 или P_4 , то в массив за-

писывается одна точка пересечения. Если горизонталь касается вершины контура (в этом случае вершина соответствует локальному минимуму или максимуму, как, например, в вершинах P_1 , P_2 , P_3 или P_5), тогда координата точки касания или не записывается, или записывается в массив два раза. Это является условием четного количества точек пересечения, хранящихся в массиве $\{x_j\}$.

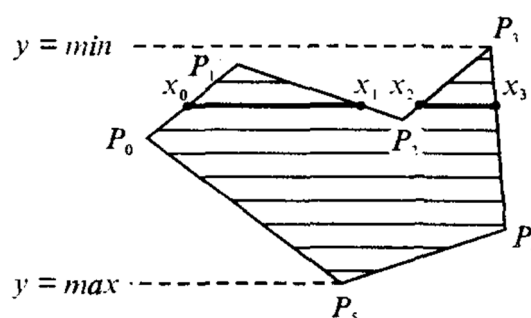


Рисунок 6.2 – Заполнение полигона

Процедура определения точек пересечения контура с горизонталью, учитывая анализ на локальный максимум, может быть достаточно сложной.

Рассмотрим, какие случаи могут возникнуть при делении многоугольника на области сканирующей строкой.

1. Простой случай. Например, на рис. 6.3 сканирующая строка $y = 4$ пересекает многоугольник при $x = 1$ и $x = 6$. Получается три области: $x < 1$; $1 \leq x \leq 6$; $x > 6$. Сканирующая строка $y = 6$ пересекает многоугольник при $x = 1$; $x = 2$; $x = 5$; $x = 6$. Получается пять областей: $x < 1$; $1 \leq x \leq 2$; $2 < x < 5$; $5 \leq x \leq 6$; $x > 6$. В этом случае x сортируется в порядке возрастания. Далее список иксов рассматривается попарно. Между парами точек пересечения закрашиваются все пиксели. Для $y = 4$ закрашиваются пиксели в интервале $(1, 6)$, для $y = 6$ закрашиваются пиксели в интервалах $(1, 2)$ и $(5, 6)$.

2. Сканирующая строка проходит через вершину (рис. 6.4). Например, по сканирующей строке $y = 3$ упорядоченный список x получится как $(2, 2, 4)$. Вершина многоугольника была учтена дважды, и поэтому закрашиваемый интервал получается неверным: $(2, 2)$. Следовательно, при пересечении вер-

шины сканирующей строкой она должна учитываться единожды. И список по x в приведенном примере будет (2, 4).

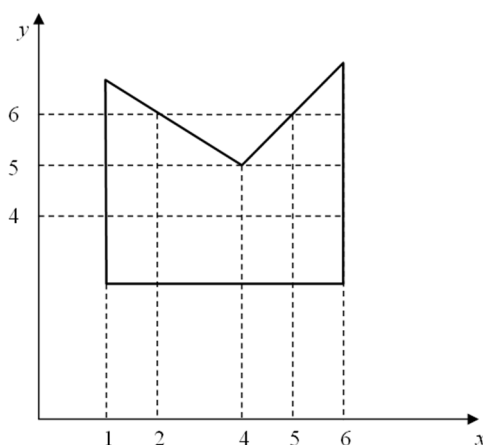


Рисунок 6.3 – Прохождение сканирующих строк по многоугольнику

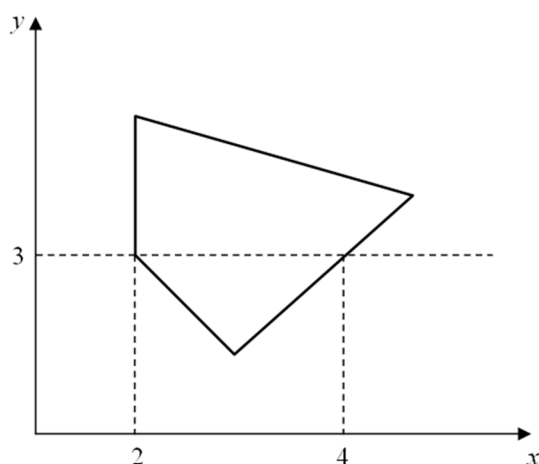


Рисунок 6.4. Прохождение сканирующей строки через вершину

3. Сканирующая строка проходит через локальный минимум или максимум (рис. 6.3 при $y = 5$). В этом случае учитываются все пересечения вершин сканирующей строкой. На рис. 6.4 при $y = 5$ формируется список x (1, 4, 4, 6). Закрашиваемые интервалы (1, 4) и (4, 6). Условие нахождения локального минимума или максимума определяется при рассмотрении концевых вершин для ребер, соединенных в вершине. Если у обоих концов координаты y больше, чем у вершины пересечения, то вершина – локальный минимум. Если меньше, то вершина пересечения – локальный максимум.

Алгоритмы закрашивания произвольных областей, заданных цветом границы

Простейший алгоритм закрашивания. Для такого алгоритма закрашивания нужно задавать начальную точку внутри контура с координатами x_0 , y_0 , от которой будет выполняться закрашка каждого соседнего пикселя рекурсивно.

Простейший рекурсивный алгоритм:

```
void PixelFill (int x, int y, unsigned char
*border_color, unsigned char *color) // рекурсивная за-
краска
{
    unsigned char p[4];
    int i,k=0,f=0;
    glColor4ub(*(color+0),*(color+1),*(color+2),*(color+3));
    glReadPixels(x, y, 1, 1, GL_RGBA, GL_UNSIGNED_BYTE,
    &p[0]);
    for(i=0;i<4;i++)
    {k++;
    if(p[i]!=*(border_color+i)) break;}
    for(i=0;i<4;i++)
    {f++;
    if(p[i]!=*(color+i)) break;}
    if((k<4)&&(f<4))
    { drawDot(x,y);
    PixelFill (x, y+1 , border_color, color);
    PixelFill (x+1, y , border_color, color);
    PixelFill (x, y-1 , border_color, color);
    PixelFill (x-1, y , border_color, color);
    }
}
```


Этот алгоритм является слишком неэффективным, так как для всякого уже отрисованного пикселя функция вызывается ещё 4 раза и, кроме того, данный алгоритм не пригоден для закрашивания контуров фигур площадью в тысячу и более пикселей, так как вложенные вызовы функций делаются для каждого пикселя, что приводит к переполнению стека в ходе выполнения программы.

Поэтому для решения задачи закрашки области предпочтительнее алгоритмы, способные обрабатывать сразу целые группы пикселей, т. е. использовать их «связность». Если данный пиксель принадлежит области, то, скорее всего, его ближайшие соседи также принадлежат данной области. Группой таких пикселей обычно выступает полоса, определяемая правым пикселем.

Алгоритм закрашивания линиями. Данный алгоритм получил широкое распространение в компьютерной графике. От простейшего алгоритма он отличается тем, что на каждом шаге закрашивания рисуется горизонтальная линия, которая размещается между пикселями контура. Алгоритм также рекурсивный, но поскольку вызов функции осуществляется для линии, а не для каждого отдельного пикселя, то количество вложенных вызовов уменьшается пропорционально длине линии. Это уменьшает нагрузку на стековую память компьютера и обеспечивает высокую скорость работы.

Задание

Написать и отладить программу на языке C/C++ для вывода и заполнения объекта вывода (табл. 5.1). Для этого:

1. Написать необходимые функции для реализации растровых алгоритмов:
 - закрашка окружности;
 - закрашка эллипса;
 - закрашка многоугольника;
 - закрашка произвольной области.

2. Нарисовать и закрасить объект по варианту согласно данным табл.
5.1. Не использовать рекурсивную закраску произвольной области для закраски круга, эллипса или полигона.

Содержание отчета

1. Титульный лист.
2. Задание.
3. Описание реализованных методов.
4. Текст программы.

При защите лабораторной работы тестирование программы **обязательно!**

Лабораторная работа № 7

Тема: алгоритмы отсечения фигур

Отсечение по полю вывода

Проблема отсечения изображения по некоторой границе, присутствует в большинстве задач компьютерной графики. Рассматривается задача применительно к отрезкам прямых и обобщается на многоугольники. Окружности и другие фигуры можно аппроксимировать отрезками (многоугольниками).

Алгоритм Козна-Сазерленда

Рассмотрим алгоритм Козна-Сазерленда для отсечения отрезков прямых. Этот алгоритм позволяет легко определять нахождение отрезка полностью внутри или полностью снаружи окна, и если так, то его можно рисовать или не рисовать, не заботясь об отсечении по границе окна.

Для работы алгоритма вся плоскость, в которой лежит окно, разбивается на девять подобластей или квадратов, как показано на рис. 7.1.

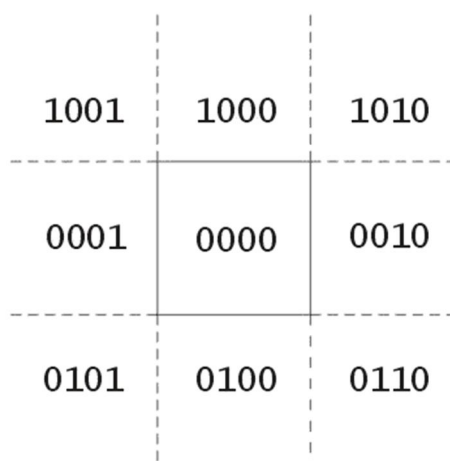


Рисунок 7.1 – Разбиение на подобласти в методе Козна-Сазерленда

Окну соответствует область обозначенная кодом 0000. Конечным точкам отрезка приписывается 4-битный код «вне/внутри», в зависимости от нахождения отрезка в соответствующей подобласти. Каждому биту присваивается значение 1 в соответствии со следующим правилом:

Бит 1 – точка находится выше окна;

Бит 2 – точка находится ниже окна;

Бит 3 – точка находится справа от окна;

Бит 4 – точка находится слева от окна;

Иначе биту присваивается нулевое значение. Значения этих битов для конечных точек отрезков легко определить по знакам соответствующих разностей (положительный результат – 0, отрицательный – 1): $(y_{\max} - y)$ – для 1-го бита, $(y - y_{\min})$ – для 2-го бита, $(x_{\max} - x)$ – для 3-го бита и $(x - x_{\min})$ – для 4-го бита.

Для каждого отрезка рассчитываются коды концов (K_1 , K_2) затем производится анализ:

- Если $K_1 \wedge K_2 \neq 0$, тогда отрезок лежит вне поля вывода – отрезок отбрасывается
- Если $K_1 = K_2 = 0$, тогда отрезок полностью лежит внутри поля вывода – отсечение не нужно, отрезок полностью прорисовывается
- Если $K_1 \wedge K_2 = 0$, отрезок может частично лежать внутри поля вывода – необходимо отсечение по полю вывода.

В этом случае применяется последовательное разделение отрезка, так что на каждом шаге конечная точка отрезка с ненулевым кодом «вне/внутри» заменяется на точку, лежащую на стороне окна или на прямой, содержащей сторону. При этом порядок перебора сторон окна не имеет значения.

Когда $K_1 \wedge K_2 = 0$ необходимо отсекал отрезок по границам поля вывода, отсечение происходит последовательно по всем сторонам рис.7.2.

На рис. 7.2 жирным выделено ребро, по которому происходит отсечение. Также надо отметить, что точки, лежащие на границе поля вывода, принадлежат полю вывода.

На каждом шаге отсечения вычисляются новые координаты одной точки, найдем формулы для вычисления новых координат.

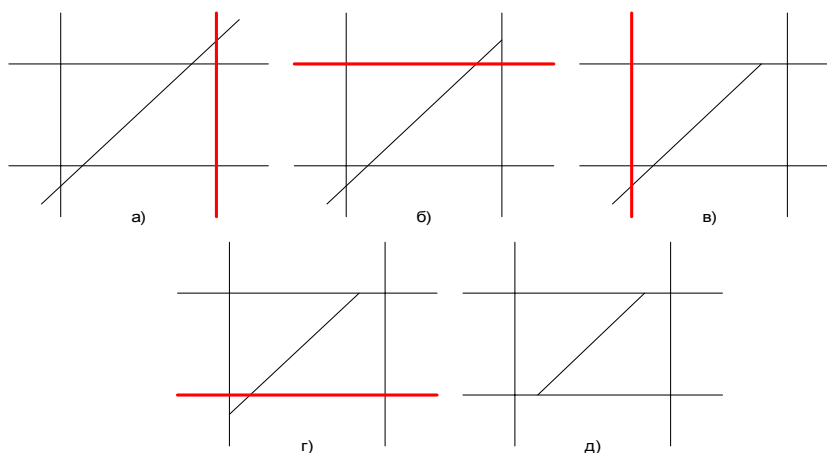


Рисунок 7.2 – «Отсечение отрезка по прямоугольной области»

Алгоритм Сазерленда-Ходгмана для отсечения многоугольников

Рассмотрим алгоритм Сазерленда-Ходгмана (Sutherland-Hodgman). В алгоритме используется стратегия «разделяй и властвуй», которая позволяет решение общей задачи свести к решению ряда простых и похожих подзадач. Примером такой подзадачи является отсечение многоугольника относительно одной отсекающей границы. Последовательное решение четырех таких задач позволяет провести отсечение относительно прямоугольной области (рис. 7.3).

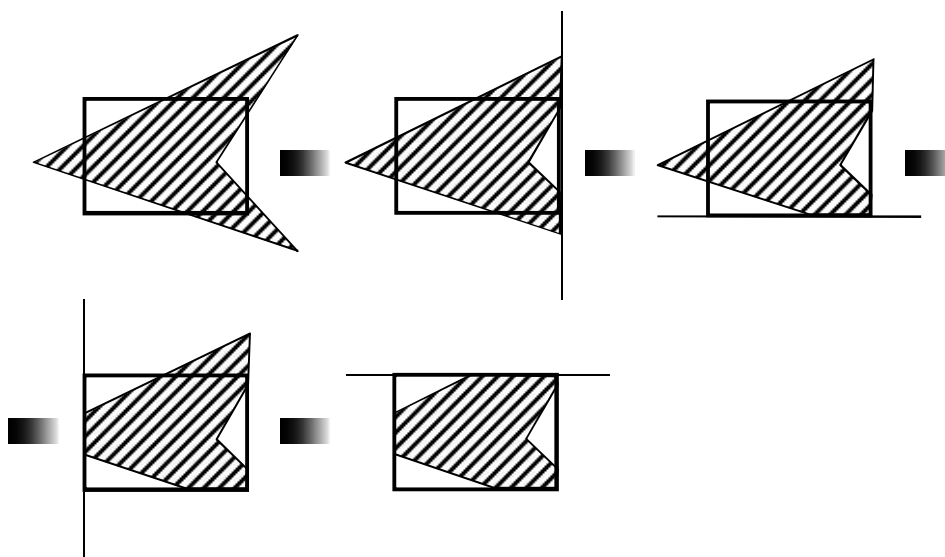


Рисунок 7.3 –Последовательное отсечение многоугольника

На вход алгоритма поступает последовательность вершин многоугольника V_1, V_2, \dots, V_n . Ребра многоугольника проходят от V_i к V_{i+1} , от V_n к V_1 . С

помощью алгоритма производится отсечение относительно ребра и выводится другая последовательность вершин, описывающая усеченный многоугольник.

Алгоритм «обходит» вокруг многоугольника от V_n к V_1 и обратно к V_n , проверяя на каждом шаге соотношение между последовательными вершинами и отсекающей границей. Необходимо проанализировать четыре случая (рис. 7.4).

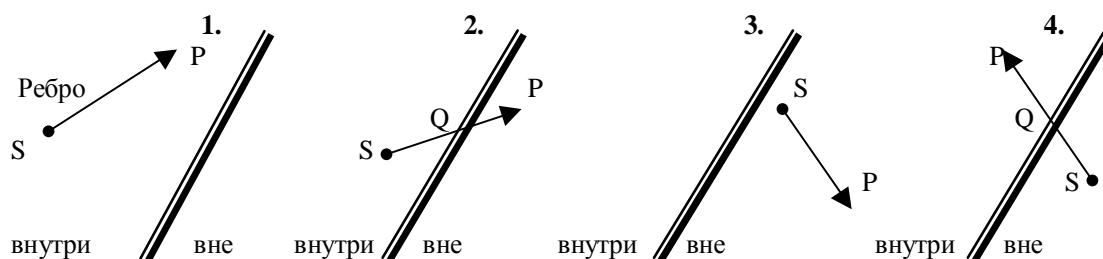


Рисунок 7.4 – Случаи, возникающие при отсечении многоугольников
Может также возникнуть вариант, когда ребро совпадает с границей.

В случае 1: Добавить в список рисуемых вершин P;

2: Добавить в список Q;

3: Ничего не добавляется;

4: Добавить в список P,Q;

Алгоритм Вейлера – Азертонна

Как обрабатываемый (объект), так и отсекающий (отсекатель) многоугольники описываются в алгоритме циклическими списками их вершин. Внешняя граница каждого из этих многоугольников обходится по часовой стрелке, а внутренние границы или отверстия – против часовой стрелки. Это условие означает, что при обходе вершин многоугольника в порядке их следования, в соответствующем списке внутренняя его область будет расположена справа от границы. Границы обрабатываемого (объекта) и отсекающего (отсекателя) многоугольников могут пересекаться или не пересекаться между собой. Если они пересекаются, то точки пересечения образуют пары. Одно пересечение из пары возникает, когда ребро обрабатываемого многоугольни-

ка входит внутрь отсекающего многоугольника, а другое – когда оно выходит оттуда. Основная идея заключается в том, что алгоритм начинается с точки пересечения входного типа, затем он прослеживает внешнюю границу по часовой стрелке до тех пор, пока не обнаруживается еще одно ее пересечение с отсекающим многоугольником. В точке последнего пересечения производится поворот направо и далее прослеживается внешняя граница отсекателя по часовой стрелке до тех пор, пока не обнаруживается ее пересечение с обрабатываемым многоугольником. И вновь, в точке последнего пересечения производится поворот направо и далее прослеживается граница обрабатываемого многоугольника. Этот процесс продолжается до тех пор, пока не встретится начальная вершина. Внутренние границы обрабатываемого многоугольника обходятся против часовой стрелки.

Пример. Отсечение с помощью алгоритма Вейлера — Азертон.

Случай простого многоугольника. Рассмотрим многоугольник, показанный на рис. 7.5, который отсекается по квадрату. На рисунке изображены также точки пересечения этих многоугольников, помеченные через I_t . Ниже приводятся списки вершин обрабатываемого и отсекающего многоугольников. В список входов занесены вершины I_2, I_4, I_6 и I_8 , а в список выходов — вершины I_1, I_3, I_5, I_7 .

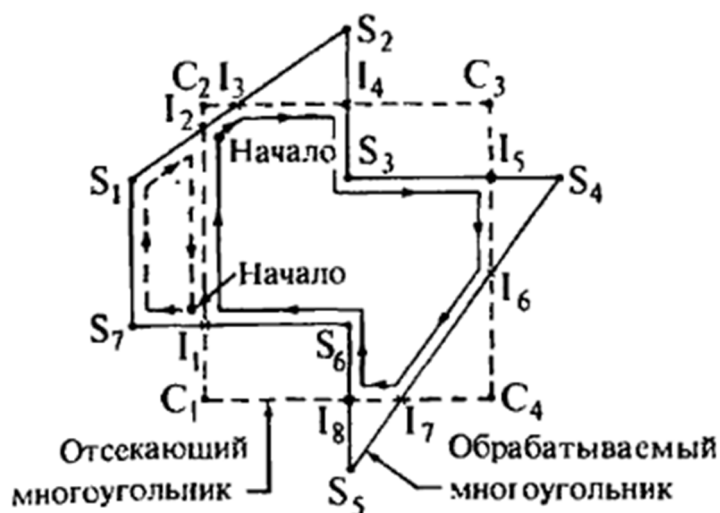


Рисунок 7.5 – Алгоритм Вейлера -Азертон

Для формирования внутреннего многоугольника возьмем первую точку пересечения из списка входов – I_2 . Описанный алгоритм дает результаты, показанные сплошной линией со стрелками на рис. 7.5, а также в таблице 7.1. Результирующий внутренний многоугольник таков:

$$I_2 I_3 I_4 S_3 I_5 I_6 I_7 I_8 S_6 I_1 I_2.$$

Таблица 7.1 - Пример отсечения

Вершины обрабатываемого многоугольника	Вершины отсекающего многоугольника	Вершины обрабатываемого многоугольника	Вершины отсекающего многоугольника
<p>Начало</p> <p>S_1</p> <p>I_2</p> <p>I_3</p> <p>S_2</p> <p>I_4</p> <p>S_3</p> <p>I_5</p> <p>S_4</p> <p>I_6</p> <p>I_7</p> <p>S_5</p> <p>I_8</p> <p>S_6</p> <p>I_1</p> <p>S_7</p> <p>S_1</p>	<p>C_1</p> <p>I_1</p> <p>I_2 Конец</p> <p>C_2</p> <p>I_3</p> <p>I_4</p> <p>C_3</p> <p>I_5</p> <p>I_6</p> <p>C_4</p> <p>I_7</p> <p>I_8</p> <p>C_1</p>	<p>S_1</p> <p>I_2</p> <p>I_3</p> <p>S_2</p> <p>I_4</p> <p>S_3</p> <p>I_5</p> <p>S_4</p> <p>I_6</p> <p>I_7</p> <p>S_5</p> <p>I_8</p> <p>S_6</p> <p>I_1 Начало</p> <p>S_7</p> <p>S_1</p>	<p>C_1</p> <p>I_1 Конец</p> <p>C_2</p> <p>I_3</p> <p>I_4</p> <p>C_3</p> <p>I_5</p> <p>I_6</p> <p>C_4</p> <p>I_7</p> <p>I_8</p> <p>C_1</p>
Внутренний многоугольник		Внешний многоугольник	

Если начинать с других точек пересечения из списка входов, т. е. с точек I_4 , I_6 и I_8 , то получится тот же самый результат.

Для формирования внешних многоугольников возьмем первую точку пересечения из списка выходов – I_1 . Описанный выше алгоритм дает результаты, показанные штриховой линией со стрелками на рис.7.5, и в таблице 7.1. Заметим, что вершины из списка вершин отсекаателя проходятся в обратном порядке от I_1 к I_2 . Получающиеся внешний многоугольник таков:

$$I_1 S_7 S_1 I_2 I_1.$$

Если же начинать построение с других точек пересечения – I_3 , I_5 или I_7 – из списка выходов, то получатся другие внешние многоугольники, соответственно:

$$I_3 S_2 I_4 I_3, I_5 S_4 I_6 I_5 \text{ или } I_7 S_5 I_8 I_7.$$

Задание

Написать и отладить программу на языке высокого уровня с использованием функциональных возможностей OpenGL для реализации отсечения фигур по полю вывода. Для этого:

1. Нарисовать поле вывода в центре экрана.
2. Нарисовать свою фигуру (табл.7.2) в произвольном месте. Координаты фигуры в таблице 7.2 задают относительные пропорции, а не абсолютные величины. Если одна часть фигуры внутри поля вывода, а другая – вне ее, то закрасить ту часть фигуры, которая внутри. Часть (фигура) вне поля вывода рисуется только контуром.
3. Обеспечить перемещение фигуры стрелками или другими клавишами (без изменения (движения) поля вывода) соблюдая правило: внутри поля вывода – закрашенная фигура, вне – только контур.

Таблица 7.2 - Варианты заданий

№	Фигура		Цвет контура	Цвет заливки
1	A(10,40), C(30,25), E(45,35).	B(15,15), D(50,18),	красный	синий
2	A(20,40), C(45,25), E(55,35).	B(25,15), D(60,18),	синий	красный
3	A(10,50), C(30,35), E(45,45).	B(15,25), D(50,28),	зеленый	белый
4	A(10,40), C(30,5), E(45,35).	B(15,15), D(50,18),	серый	желтый

№	Фигура	Цвет контура	Цвет заливки
5	A(5,40), B(10,15), C(15,10), D(45,18), E(40,35).	красный	белый
6	A(20,49), B(25,5), C(40,35), D(60,40), E(70,65).	желтый	зеленый
7	A(15,40), B(20,15), C(35,25), D(55,18), E(50,35).	белый	Красный
8	A(10,45), B(15,20), C(30,30), D(50,25), E(45,40).	синий	желтый
9	A(15,45), B(20,20), C(35,30), D(55,25), E(50,40).	серый	белый
10	A(12,40), B(18,15), C(35,25), D(54,18), E(47,35).	зеленый	синий

Содержание отчета

1. Титульный лист.
2. Задание.
3. Описание используемых методов.
4. Текст программы.

При защите лабораторной работы тестирование программы **обязательно!**

Лабораторная работа № 8

Тема: построение реалистичных изображений

Визуализация трехмерных объектов

Любой трехмерный объект может быть изображен по-разному и различными способами. Условно разделим способы визуализации по характеру изображений и по степени сложности соответствующих алгоритмов на такие уровни:

1. Каркасная визуализация;
2. Показ поверхностей в виде многогранников с плоскими гранями или сплайнов с удалением невидимых точек.

На рис. 8.1, *а* приведен типичный каркасный чертеж куба. Каркасный чертеж представляет трехмерный объект в виде изображения его ребер. Рис. 8.1, *а* можно интерпретировать двояко: как вид куба сверху слева или снизу справа. Удаление тех линий или поверхностей, которые невидимы с соответствующей точки зрения, позволяют избавиться от неоднозначности. Результаты показаны на рис. 8.1, *б* и 8.1, *в*.

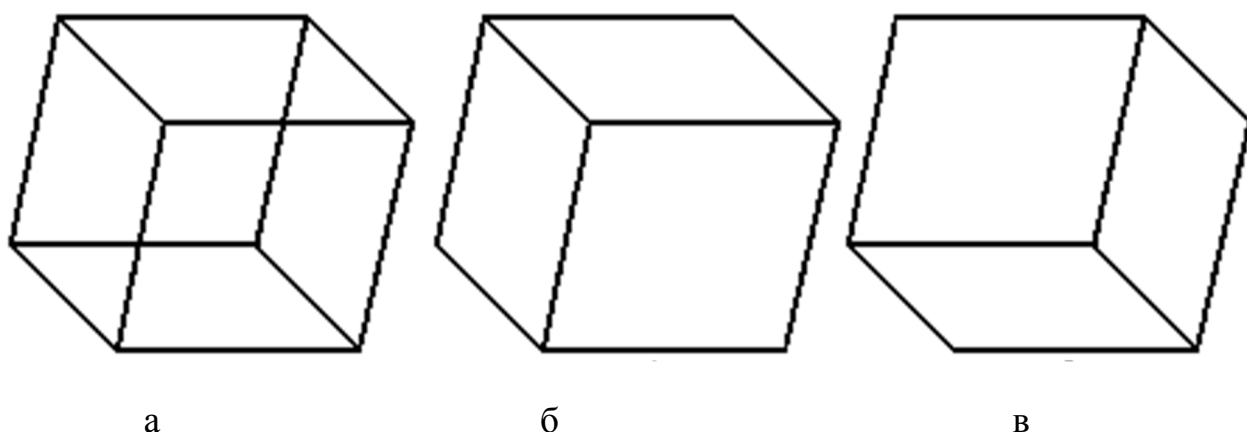


Рисунок 8.1 - Необходимость удаления невидимых линий

3. То же что и для второго уровня, плюс сложное закрашивание объектов для имитации отражения света, затенения, прозрачности, использование текстур.

Световая энергия, падающая на поверхность от источника света (рис.8.2а), может быть поглощена (рис. 8.2в), отражена (рис. 8.2б) и пропущена (рис. 8.2г). Количество поглощенной, отраженной и пропущенной энергии зависит от длины световой волны. При этом цвет поверхности объекта определяется поглощаемыми длинами волн.

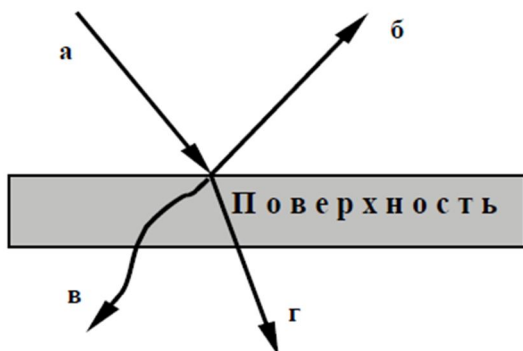


Рис. 8.2 - Падающий свет к поверхности

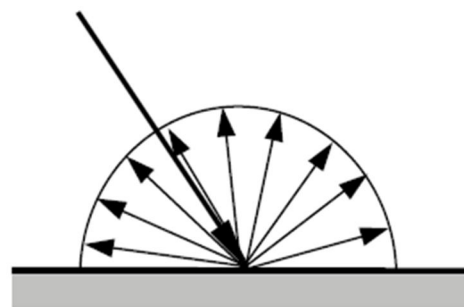


Рис. 8.2а - Диффузно отраженный свет

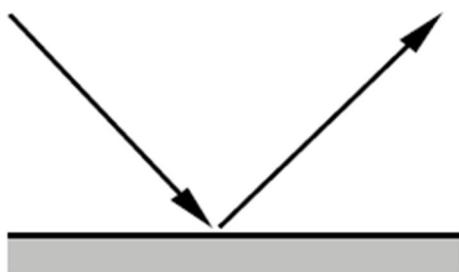


Рис. 8.2 б - Зеркальное отражение

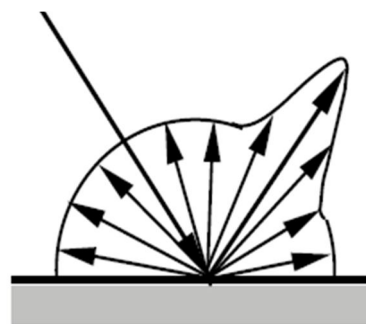


Рис. 8.2в - Поверхность, имеющая диффузную и зеркальную характеристики

Свойство отраженного света зависят от формы и направления источника света, а также от ориентации освещаемой поверхности и ее свойств. Свет, отраженный от объекта, может быть диффузным и зеркальным: диффузно отраженный свет рассеивается равномерно по всем направлениям (рис. 8.2а), зеркальное отражение происходит от внешней поверхности объекта (рис. 8.2б). На рисунке 8.2в показана поверхность, имеющая как диффузную, так и зеркальную характеристики.

Диффузное отражение и рассеянный свет

Матовые поверхности обладают свойством диффузного отражения, т. е. равномерного по всем направлениям рассеивания света. Матовой можно считать такую поверхность, размер шероховатостей которой уже настолько большой, что падающий луч рассеивается равномерно во все стороны. Такой тип отражения характерен, например, для гипса, песка, бумаги. Поэтому кажется, что поверхности имеют одинаковую яркость независимо от угла обзора. Для таких поверхностей справедлив закон косинусов Ламберта, устанавливающий соответствие между количеством отраженного света и косинусом угла θ между направлением \bar{L} на точечный источник света интенсивности I_p и нормалью \bar{N} к поверхности (рис. 8.3). При этом количество отраженного света не зависит от положения наблюдателя.

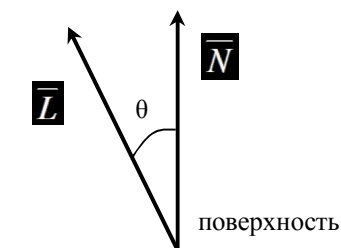


Рисунок 8.3 - Падающий свет и нормаль к поверхности

Освещенность рассеянным светом вычисляется по формуле

$$I_d = I_p k_d \cos \theta \quad (8.1)$$

I_d – интенсивность отраженного света. I_p – интенсивность источника света. Значение коэффициента диффузного отражения k_d является константой в диапазоне (0, 1) и зависит от материала. Если векторы \bar{L} и \bar{N} нормированы, то, используя скалярное произведение, формулу освещенности можно записать так:

$$I_d = I_p \cdot k_d \cdot (\bar{L} \cdot \bar{N}) \quad (8.2)$$

Предметы, освещенные одним точечным источником света, выглядят контрастными. Этот эффект аналогичен тому, который можно наблюдать, когда предмет, помещенный в темную комнату, виден при свете направленного

ной на него фотовспышки. В данной ситуации, в отличие, от большинства реальных визуальных сцен, отсутствует рассеянный свет, под которым здесь понимается свет постоянной яркости, созданный многочисленными отражениями от различных поверхностей. Такой свет практически всегда присутствует в реальной обстановке. Даже если предмет защищен от прямых лучей, исходящих от точечного источника света, он все равно будет виден из-за наличия рассеянного света. Учитывая это, формулу окраски можно записать так:

$$I_d = I_a \cdot k_a + I_p \cdot k_d \cdot (\bar{L} \cdot \bar{N}) \quad (8.3)$$

Рассеянный свет представлен членом I_a - интенсивность рассеянного света, и k_a коэффициента, который определяет количество рассеянного света, которое отражается от поверхности предмета.

Точечный источник света удобнее всего расположить в позиции, совпадающей с глазом наблюдателя. Тени в этом случае отсутствуют, а лучи света, падающие на поверхность, окажутся параллельными. Однако теперь, если две поверхности одного цвета параллельны друг другу и их изображения перекрываются, нормали к поверхностям совпадают и, следовательно, поверхности закрашиваются одинаково, различить их невозможно. Этот эффект можно устранить, если учесть, что энергия падающего света убывает пропорционально квадрату расстояния, которое свет проходит от источника до поверхности и обратно к глазу наблюдателя. Обозначая это расстояние за R , запишем:

$$I_d = I_a \cdot k_a + I_p \cdot k_d \cdot (\bar{L} \cdot \bar{N}) / R^2. \quad (8.4)$$

Однако данным правилом на практике трудно воспользоваться. Для параллельной проекции, когда источник света находится в бесконечности, расстояние R также становится бесконечным. Даже в случае центральной проекции величина $1/R^2$ может принимать значения в широком диапазоне, поскольку точка зрения часто оказывается достаточно близкой к предмету. В результате закрашка поверхностей, которые имеют одинаковые углы θ между

\bar{N} и \bar{L} , будет существенно различаться. Большей реалистичности можно достичь, если заменить R^2 на $r+k$, где k – некоторая константа, а r – расстояние от центра проекции до поверхности:

$$I_d = I_a \cdot k_a + I_p \cdot k_d \cdot (\bar{L} \cdot \bar{N}) / (r+k). \quad (8.5)$$

Для представления диффузного отражения от цветных поверхностей уравнения записываются отдельно для основных цветов модели СМУ (голубого, пурпурного и желтого). При этом константы отражения для этих цветов задаются тройкой чисел $(k_{dc} \ k_{dm} \ k_{dy})$. Эти цвета используются, поскольку отражение света является субтрактивным процессом. Поэтому интенсивность для цветного изображения описывается тремя уравнениями:

$$I_{dc} = I_{ac} \cdot k_{ac} + I_{pc} \cdot k_{dc} \cdot (\bar{L} \cdot \bar{N}) / (r+k) \quad \text{(для голубой компоненты);} \quad (8.6)$$

$$I_{dm} = I_{am} \cdot k_{am} + I_{pm} \cdot k_{dm} \cdot (\bar{L} \cdot \bar{N}) / (r+k) \quad \text{(для пурпурной компоненты);} \quad (8.7)$$

$$I_{dy} = I_{ay} \cdot k_{ay} + I_{py} \cdot k_{dy} \cdot (\bar{L} \cdot \bar{N}) / (r+k) \quad \text{(для желтой компоненты).} \quad (8.8)$$

Зеркальное отражение

Зеркальное отражение можно получить от любой блестящей поверхности. Осветите ярким светом яблоко – световой блик на яблоке возникает в результате зеркального отражения, а свет, отраженный от остальной части, появится в результате диффузного отражения. Отметим также, что в том месте, где находится световой блик, яблоко кажется не красным, а скорее белым, т. е. окрашенным в цвет падающего цвета.

Если мы изменим положение головы, то заметим, что световой блик тоже сместится. Это объясняется тем, что блестящие поверхности отражают свет неодинаково по всем направлениям. От идеального зеркала свет отражается только в том направлении, для которого углы падения и отражения совпадают. Это означает, что наблюдатель сможет увидеть зеркально отраженный свет только в том случае, если угол α (рис. 8.4.) равен нулю.

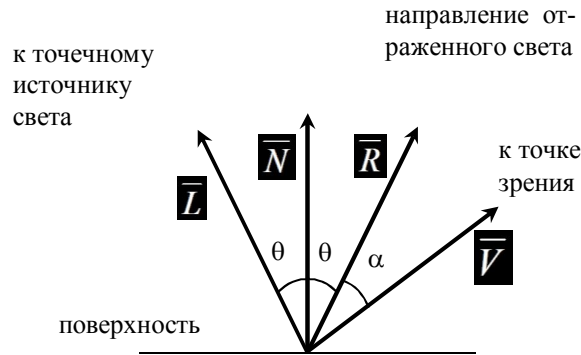


Рис. 8.4 - Зеркальное отражение

Для неидеальных отражающих поверхностей, таких, как яблоко, интенсивность отраженного света резко падает с ростом α . В модели предложенной Фонгом, быстрое убывание интенсивности описывается функцией $\cos^n \alpha$, где n обычно лежит в диапазоне 1–200, в зависимости от вида поверхности. Для идеального отражателя n бесконечно велико. В основе такой модели лежит эмпирическое наблюдение, а не фундаментальное понимание процесса зеркального отражения.

Количество падающего света, которое зеркально отражается в случае реальных материалов, зависит от угла падения θ . Обозначим зеркально отражаемую долю света через $W(\theta)$, тогда

$$I_d = I_a \cdot k_a + \frac{I_p}{r+k} \cdot [k_d \cdot \cos \theta + W(\theta) \cdot \cos^n \alpha]. \quad (8.9)$$

Если векторы направления отраженного света и направления к точке зрения \bar{R} и \bar{V} нормированы, то $\cos \alpha = (\bar{R} \cdot \bar{V})$. Часто в качестве $W(\theta)$ служит константа k_s , которая выбирается таким образом, чтобы получающиеся результаты были приемлемы с эстетической точки зрения. В этом случае уравнение с учетом зеркального отражения можно записать так:

$$I_d = I_a \cdot k_a + \frac{I_p}{r+k} \cdot [k_d \cdot (\bar{L} \cdot \bar{N}) + k_s \cdot (\bar{R} \cdot \bar{V})^n]. \quad (8.10)$$

I_d – интенсивность отраженного света. I_p – интенсивность источника света. I_a – интенсивность рассеянного света. k_d – коэффициент диффузного отражения. k_a коэффициент рассеянного света, k_s – коэффициент зеркального отражения. \bar{R} и \bar{V} – векторы направления отраженного света и направления к

точке зрения. \bar{L} и \bar{N} вектор направления на точечный источник света и нормаль к поверхности.

Для цветного изображения описываются три уравнения: для голубого, пурпурного и желтого цветов:

$$I_{dc} = I_{ac} \cdot k_{ac} + \frac{I_{pc}}{r+k} [k_{dc} \cdot (\bar{L} \cdot \bar{N}) + k \cdot (\bar{R} \cdot \bar{V})^n]; \quad (\text{для голубой компоненты}); \quad (8.11)$$

$$I_{dm} = I_{am} \cdot k_{am} + \frac{I_{pm}}{r+k} [k_{dm} \cdot (\bar{L} \cdot \bar{N}) + k_s \cdot (\bar{R} \cdot \bar{V})^n]; \quad (\text{для пурпурной компоненты}); \quad (8.12)$$

$$I_{dy} = I_{ay} \cdot k_{ay} + \frac{I_{py}}{r+k} [k_{dy} \cdot (\bar{L} \cdot \bar{N}) + k_s \cdot (\bar{R} \cdot \bar{V})^n]. \quad (\text{для желтой компоненты}). \quad (8.13)$$

Если источник света расположен в бесконечности, для заданного многоугольника произведение $(\bar{L} \cdot \bar{N})$ является константой, а $(\bar{R} \cdot \bar{V})$ меняет значение в многоугольнике.

Кроме эмпирической модели Фонга, для зеркального отражения разработана модель Торрэнса-Спэрроу, которая представляет собой теоретическую обоснованную модель отражающей поверхности. В этой модели предполагается, что поверхность является совокупностью микроскопических граней, каждая из которых – идеальный отражатель. Ориентация любой грани задается функцией распределения вероятностей Гаусса.

Создание источника света

Для установки источника света необходимо разместить на сцене один или несколько источников, настроить их свойства и включить их. В зависимости от реализации OpenGL на сцене могут присутствовать восемь и более источников света. Включить нулевой источник света можно командой:

```
glEnable(GL_LIGHT0);
```

Остальные включаются аналогичным способом, где вместо GL_LIGHT0 указывается GL_LIGHT_i.

Для управления свойствами источника света используются команды `glLight*`:

```
glLightf(GLenum light, GLenum pname, GLfloat param);
glLightfv(GLenum light, GLenum pname, const GLfloat
*param);
```

Команда `glLightf` используется для задания скалярных параметров, а `glLightfv` используется для задания векторных характеристик источников света.

Параметр `light` указывает OpenGL для какого источника света задаются параметры (`GL_LIGHT0`, `GL_LIGHT1` и т.д.).

Параметр `pname` задает параметр освещения. Допустимые параметры освещения описаны в таблице 8.1.

Таблица 8.1 – Допустимые параметры освещения

Имя параметра pname	Значение по умолчанию	Описание
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	цвет фонового излучения источника света
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0) или (0.0, 0.0, 0.0, 1.0)	цвет рассеянного излучения источника света (значение по умолчанию для <code>GL_LIGHT0</code> - белый, для остальных - черный)
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0) или (0.0, 0.0, 0.0, 1.0)	цвет зеркального излучения источника света (значение по умолчанию для <code>GL_LIGHT0</code> - белый, для остальных - черный)
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w) - направление источника направленного света. Первые три компоненты (x, y, z) задают вектор направления, а компонента w всегда равна нулю (иначе источник превратится в точечный).
GL_CONSTANT_ATTENUATION	1.0	постоянная k_{const} в функции затухания $f(d)$

Имя параметра pname	Значение по умолчанию	Описание
GL_LINEAR_ATTENUATION	0.0	коэффициент k_{linear} при линейном члене в функции затухания $f(d)$
GL_QUADRATIC_ATTENUATION	0.0	коэффициент $k_{quadratic}$ при квадрате расстояния в функции затухания $f(d)$
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0)	(x, y, z) - направление прожектора (ось ограничивающего конуса)
GL_SPOT_CUTOFF	180.0	угол между осью и стороной конуса (он же половина угла при вершине)
GL_SPOT_EXPONENT	0.0	экспонента убывания интенсивности

Рассмотрим на примере процесс создания источника света:

```
GLfloat ambientLight[] = {0.3f, 0.3f, 0.3f, 1.0f};
GLfloat diffuseLight[] = {0.7f, 0.7f, 0.7f, 1.0f};
// устанавливаем и активизируем источник света 0.
glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
glEnable(GL_LIGHT0); // включаем источник света
GL_LIGHT0.
```

В данном примере источник света `GL_LIGHT0` имеет рассеянный и диффузный компонент, интенсивности которых заданы в массивах `ambientLight[]` и `diffuseLight[]`. В результате мы получаем умеренно мощный белый источник света.

С помощью приведенного ниже кода, источник света размещается в пространстве:

```
GLfloat lightPos[] = {-50.0f, 50.0f, 100.0f, 1.0f};
// определяет положение источника света.

glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
```

```
glEnable(GL_LIGHTING); //включаем свет.
glEnable(GL_LIGHT0); // включаем конкретный источник.
```

Для отключения источника света необходимо использовать функцию `glDisable(GL_LIGHT0)`.

Закраска объектов, заданных полигональными сетками

Существует три основных способа закраски объектов, заданных полигональными сетками. В порядке возрастания сложности ими являются:

- однотонная закрашка (метод постоянного закрашивания грани);
- метод Гуро (основан на интерполяции значений интенсивности);
- метод Фонга (основан на интерполяции векторов нормали).

Однотонная закрашка полигональной сетки

При однотонной закрашке вычисляют один уровень интенсивности, который используется для закрашки всего многоугольника. При этом предполагается, что:

1. Источник света расположен в бесконечности, поэтому произведение $(\bar{L} \cdot \bar{N})$ постоянно на всей полигональной грани.
2. Наблюдатель находится в бесконечности, поэтому произведение $(\bar{N} \cdot \bar{V})$ постоянно на всей полигональной грани.
3. Многоугольник представляет реальную моделируемую поверхность, а не является аппроксимацией криволинейной поверхности. Если какое-либо из первых двух предположений оказывается неприемлемым, можно воспользоваться усредненными значениями \bar{L} и \bar{V} , вычисленными, например, в центре многоугольника.

Последнее предположение в большинстве случаев не выполняется, но оказывает существенно большее влияние на получаемое изображение, чем два других. Влияние состоит в том, что каждая из видимых полигональных

граней аппроксимированной поверхности хорошо отличима от других, поскольку интенсивность каждой из этих граней отличается от интенсивности соседних граней. Различие в окраске соседних граней хорошо заметно вследствие эффекта полос Маха.

В качестве модели освещенности обычно используются простейшие модели.

Метод Гуро

Метод закрашки, который основан на интерполяции интенсивности и известен как метод Гуро (по имени его разработчика), позволяет устранить дискретность изменения интенсивности. Процесс закрашки по методу Гуро осуществляется в четыре этапа:

1. Вычисляются нормали ко всем полигонам.
2. Определяются нормали в вершинах путем усреднения нормалей по всем полигональным граням, которым принадлежит вершина (рис 8.5).

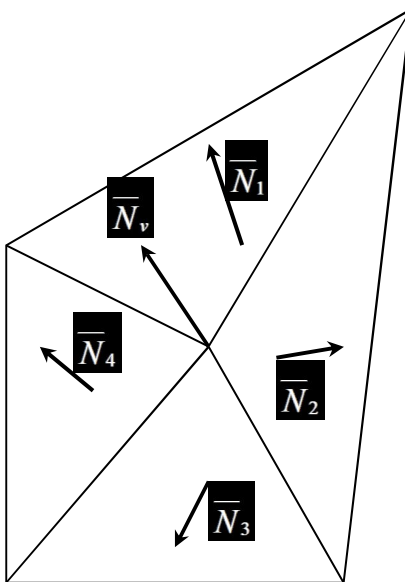


Рисунок 8.5 - Нормали к вершинам: $\bar{N}_v = (\bar{N}_1 + \bar{N}_2 + \bar{N}_3 + \bar{N}_4 + \bar{N}_v)/4$

3. Используя нормали в вершинах и применяя произвольный метод закрашки, вычисляются значения интенсивности в вершинах.

4. Каждый многоугольник закрашивается путем линейной интерполяции значений интенсивностей в вершинах сначала вдоль каждого ребра, а затем и между ребрами вдоль каждой сканирующей строки (рис. 8.6).

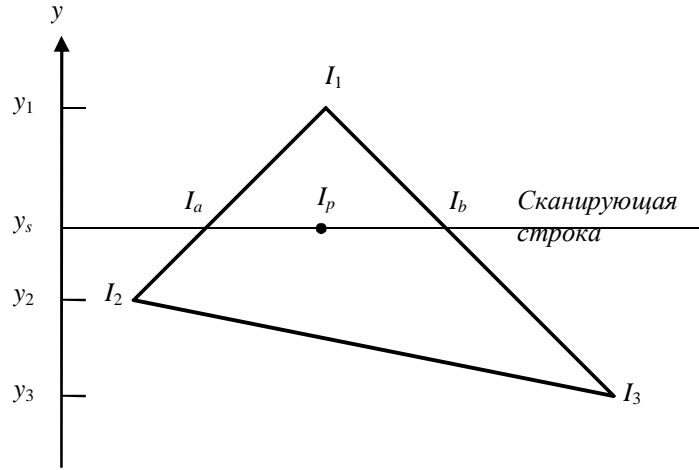


Рисунок. 8.6 - Интерполяция интенсивностей

Интерполяция вдоль ребер легко объединяется с алгоритмом удаления скрытых поверхностей, построенным на принципе построчного сканирования. Для всех ребер запоминается начальная интенсивность, а также изменение интенсивности при каждом единичном шаге по координате y . Заполнение видимого интервала на сканирующей строке производится путем интерполяции между значениями интенсивности на двух ребрах, ограничивающих интервал (рис 8.6).

$$I_a = I_1 \frac{y_s - y_2}{y_1 - y_2} + I_2 \frac{y_1 - y_s}{y_1 - y_2}; \quad (8.14)$$

$$I_b = I_1 \frac{y_s - y_3}{y_1 - y_3} + I_3 \frac{y_1 - y_s}{y_1 - y_3}; \quad (8.15)$$

$$I_p = I_a \frac{x_b - x_p}{x_b - x_a} + I_b \frac{x_p - x_a}{x_b - x_a}. \quad (8.16)$$

Для цветных объектов отдельно интерполируется каждая из компонент цвета.

Метод Фонга

В методе закраски, разработанном Фонгом, используется интерполяция вектора нормали \bar{N} к поверхности вдоль видимого интервала на сканирующей строке внутри многоугольника, а не интерполяция интенсивности. Интерполяция выполняется между начальной и конечной нормальными, которые сами тоже являются результатами интерполяции вдоль ребер многоугольника между нормальными в вершинах. Нормали в вершинах, в свою очередь, вычисляются так же, как в методе закраски, построенном на основе интерполяции интенсивности.

В каждом пикселе вдоль сканирующей строки новое значение интенсивности вычисляется с помощью любой модели закраски. Заметные улучшения по сравнению с интерполяцией интенсивности наблюдаются в случае использования модели с учетом зеркального отражения, так как при этом более точно воспроизводятся световые блики. Однако даже если зеркальное отражение не используется, интерполяция векторов нормали приводит к более качественным результатам, чем интерполяция интенсивности, поскольку аппроксимация нормали в этом случае осуществляется в каждой точке. При этом значительно возрастают вычислительные затраты.

Чтобы закрасить куски бикубической поверхности, для каждого пиксела, исходя из уравнений поверхности, вычисляется нормаль к поверхности. Этот процесс тоже достаточно дорогой. Затем с помощью любой модели закраски определяется значение интенсивности. Однако прежде чем применить метод закраски к плоским или бикубическим поверхностям, необходимо иметь информацию о том, какие источники света (если они имеются) в действительности освещают точку. Поэтому мы должны рассматривать также и тени.

Реализация тел в пространстве и стили рисования объектов в OpenGL (в том числе и выбор метода закраски) представлено в лабораторной работе 4.

Тени

Алгоритмы затенения в случае точечных источников света идентичны алгоритмам удаления скрытых поверхностей. В алгоритме удаления скрытых поверхностей определяются поверхности, которые можно увидеть из точки зрения, а в алгоритме затенения выделяются поверхности, которые можно «увидеть» из источника света. Поверхности, видимые как из точки зрения, так и из источника света, не лежат в тени. Те же поверхности, которые видны из точки зрения, но невидимы из источника света, находятся в тени. Эти рассуждения можно легко распространить на случай нескольких источников света. Отметим, однако, что, используя такой простой подход, нельзя смоделировать тени от распределенных источников света. При наличии таких источников потребуется вычислять как тени, так и полутени.

Задание

Написать и отладить программу на языке высокого уровня с использованием функциональных возможностей OpenGL для рисования трехмерных сцен и реализации координатных преобразований в пространстве. Для этого:

1. Нарисовать заданные по варианту тела (табл.4.2), применить и описать в отчете заданные по варианту модели закраски и освещения (табл. 8.2), по желанию наложить текстуру.
2. Выполнить:
 - рисование объекта;
 - рисование с удалением невидимых линий;
 - установить освещение.

Таблица 8.2 – Условия индивидуальных заданий

№ п/п	Источник света	Цвет объекта	Модель закраски объекта №1	Модель закраски объекта №2
1	Диффузное отражение	Красный	Каркасная	Поверхностная
2	Зеркальное отражение	Зеленый	Поверхностная	Каркасная
3	Диффузное отражение	Синий	Поверхностная	Каркасная

№ п/п	Источник света	Цвет объек- та	Модель закра- ки объекта №1	Модель закраски объекта №2
4	Зеркальное отражение	Белый	Каркасная	Поверхностная
5	Диффузное отражение	Черный	Поверхностная	Каркасная
6	Зеркальное отражение	Красный	Каркасная	Поверхностная
7	Диффузное отражение	Зеленый	Каркасная	Поверхностная
8	Зеркальное отражение	Синий	Поверхностная	Каркасная
9	Диффузное отражение	Белый	Поверхностная	Каркасная
10	Диффузное отражение	Черный	Каркасная	Поверхностная
11	Зеркальное отражение	Красный	Поверхностная	Каркасная
12	Диффузное отражение	Зеленый	Каркасная	Поверхностная
13	Зеркальное отражение	Синий	Поверхностная	Каркасная
14	Диффузное отражение	Белый	Каркасная	Поверхностная
15	Зеркальное отражение	Черный	Каркасная	Поверхностная
16	Диффузное отражение	Красный	Поверхностная	Каркасная
17	Зеркальное отражение	Зеленый	Каркасная	Поверхностная
18	Диффузное отражение	Синий	Поверхностная	Каркасная
19	Зеркальное отражение	Белый	Каркасная	Поверхностная
20	Диффузное отражение	Черный	Поверхностная	Каркасная

Содержание отчета

1. Титульный лист.
2. Задание.
3. Краткие теоретические сведения, обязательно описать используемые методы закрашки и освещения.
4. Текст программы.

При защите лабораторной работы тестирование программы **обязательно!**

СПИСОК ЛИТЕРАТУРЫ

1. Блинова Т.А. Компьютерная графика: учебник для вузов / Т. А. Блинова, В. Н. Порев ; Т.А. Блинова, В.Н. Порев; под ред. В.Н. Порева. - К. : Юниор; СПб. : Корона принт, 2006. - 520с. : ил. - (Учебное пособие). - ISBN 966-7323-48-X.
2. Роджерс Д.Ф. Алгоритмические основы машиной графики / Д. Ф. Роджерс; Д.Ф. Роджерс; пер. с англ.: С.А. Вичеса и др.; под ред.: Ю.М. Баяковского, В.А. Галактионова. - М. : Мир, 1989. - 503с. : ил. - Перевод изд.: Procedural elements for computer graphics/D. F. Rogers.
3. Дейтел, Х.М. Как программировать на С++ [Электронный ресурс] / Х. М. Дейтел, Дейтел П.Дж. ; Х.М. Дейтел, П.Дж. Дейтел ; пер. с англ. под ред. В.В. Тимофеева. - 5-е изд. - 19 Мб. - М. : Бином-Пресс, 2008. - 1 файл. - Перевод изд.: С++ How to Program/Н.М. Deitel, Р.Ј. Deitel.
4. OpenGL. Руководство по программированию / М. Ву [и др.] ; М.Ву, Т. Девис, Дж. Нейдер, Д. Шрайнер ; пер. с англ.: Е. Васильев, Е. Эрман. - 4-е изд. - СПб. : Питер, 2006. - 624с.: ил. - (Библиотека программиста). - Перевод изд.: OpenGL/ М. Woo, Т. Davis, J. Neider, D. Shreiner. - ISBN 5-94723-827-6.
5. Шрайнер Д. OpenGL. Официальный справочник : перевод с английского / Д. Шрайнер; Д. Шрайнер ; под ред. Д. Шрейнера. - СПб.: ДиаСофтЮП, 2002. - 512с. - Перевод изд.: OpenGL Reference Manual, Third Edition/ ed. D. Shreiner. - ISBN 5-93772-048-2.
6. Тихомиров, Ю.В. OpenGL. Программирование трехмерной графики / Ю. В. Тихомиров ; Ю.В. Тихомиров. - 2-е изд. - СПб. : БХВ-Петербург, 2002. - 304с.: ил. - (Мастер программ). - ISBN 5-94157-174-7.
7. Боресков, А.В. Компьютерная графика: первое знакомство / А. В. Боресков, Г. Е. Шикин, Г. Е. Шикина ; А.В. Боресков, Е.В. Шикин, Г.Е. Шикина ; под ред. Е.В. Шикина . - М. : Финансы и статистика, 1996. - 176с. : ил. - (Диалог с компьютером).
8. Гайван А.В. Компьютерная графика и численные методы / А. В. Гайван ; А.В. Гайван. - М. : ВА Принт, 1994. - 114с. : ил.

9. Залогова Л.А. Компьютерная графика : практикум / Л. А. Залогова ; Л.А. Залогова ; науч. ред. С.В. Русаков. - 2-е изд. - М. : БИНОМ. Лаборатория знаний, 2007. - 245с.: ил. - (Элективный курс. Информатика). - ISBN 978-5-94774-656-3.

10. Корриган Д. Компьютерная графика : секреты и решения / Д. Корриган ; пер.с англ. Д.А. Куликова. - М. : Энтроп, 1995. - 352с. : ил. - Перевод изд.: Computer graphics/J. Corrigan.

11. Коцюбинский А.О. Компьютерная графика : практическое пособие / А. О. Коцюбинский, С. В. Грошев ; А.О.Коцюбинский, С.В.Грошев. - М. : ТЕХНОЛОДЖИ-3000, 2001. - 752с. : ил. - ISBN 5-94472-011-8.

12. Дегтярев, В.М. Компьютерная геометрия и графика: учебник для вузов / В. М. Дегтярев ; В.М. Дегтярев. - М. : ИЦ "Академия", 2010. - 192с. - (Высшее профессиональное образование. Информатика и вычислительная техника). - ISBN 978-5-7695-5888-7.