Библиотеки OpenGL и GLU включены в состав Microsoft Visual Studio 2012, однако библиотеку GLUT необходимо подключить самостоятельно. Для этого нужно скачать её по адресу http://user.xmission.com/~nate/glut/glut-3.7.6-bin.zip. Архив содержит пять файлов, два из которых являются описательными, а три других нужно поместить в следующие папки:

- ✓ glut.h в папку C:\Program Files\Microsoft Visual Studio 11.0\VC\include
- ✓ glut32.lib в папку C:\Program Files\Microsoft Visual Studio 11.0\VC\lib
- ✓ glut32.dll в папку C:\Windows\System32 (или C:\Windows\SysWOW64 для 64-битных систем)

Примечание: Создайте новый проект в Visual Studio 2012, выбрав пункт меню Файл-> Создать-> Проект-> Пустой проект. Затем щёлкните правой кнопкой мыши по своему проекту в левой части окна и выберите Добавить-> Создать элемент-> Файл C++. Компилируйте и запускайте программы с помощью клавиш F7 и F5 соответственно.

ЛАБОРАТОРНАЯ РАБОТА №1.
ЗНАКОМСТВО С OPENGL (ГЕНЕРАЦИЯ ПРЯМЫХ)

Цель. Получить начальные сведения об OpenGL и реализовать алгоритмы построения прямых линий

Алгоритмы построения прямых линий

Декартово уравнение прямой линии имеет вид:

$$y=kx+b, \quad (1)$$

где k – тангенс угла наклона прямой (или, другими словами, линейный коэффициент), b – точка ее пересечения с осью y. Если известно, что два конца отрезка заданы как точки с координатами (x_1, y_1) и (x_2, y_2), то можно найти значения тангенса угла наклона по формуле:

$$k = (y_2 - y_1)/(x_2 - x_1). \quad (2)$$

Назначение генератора векторов – соединение двух точек изображения отрезком прямой.

Существует несколько алгоритмов построения прямых линий.

1. Симметричный цифровой дифференциальный анализатор (*DDA – Digital Differential Analyzer*).

В обычном ЦДА, используемом, как правило, в векторных устройствах, тем или иным образом определяется количество узлов N, используемых для аппроксимации отрезка. Затем за N циклов вычисляются координаты очередных узлов:

$$x_{i+1} = x_i + \frac{d_x}{N}; \quad (3)$$

$$y_{i+1} = y_i + \frac{d_y}{N}, \quad (4)$$

где $d_y = y_2 - y_1$ – приращение координат отрезка по оси y, а $d_x = x_2 - x_1$ – приращение координат отрезка по оси x.

Получаемые значения x_i, y_i преобразуются в целочисленные значения координаты очередного подсвечиваемого пикселя либо округлением, либо отбрасыванием дробной части.

Генератор векторов, использующий этот алгоритм, имеет тот недостаток, что точки могут прописываться дважды, что увеличивает время построения.

Кроме того, из-за независимого вычисления обеих координат нет предпочтительных направлений, и построенные отрезки кажутся не очень красивыми.

Для генерации отрезка из точки (x_1, y_1) в точку (x_2, y_2) в первом октанте ($p_x \geq p_y \geq 0$) алгоритм симметричного ЦДА имеет вид:

1. Вычислить приращения координат:
   ```
   dx=x2-x1;
   dy=y2-y1;
   ```
2. Задать количество узлов аппроксимации N
3. Занести начальную точку отрезка
   ```
   SetPixel (x1,y1);
   ```
4. Сгенерировать отрезок
   ```
   while (x1<x2)
   {
   x1:= x1+dx/N;
   y1:= y1+dy/N;
   SetPixel(x1,y1);
   }
   ```

Далее приведена программа, которая учитывает как случай ($p_x \geq p_y \geq 0$), так и случай ($p_y \geq p_x \geq 0$).

Листинг 1

```
#include <glut.h>  /*подключаем библиотеку GLUT*/
#include <math.h>  /*подключаем библиотеку для
математических операций*/

/*функция округления*/
inline int round(const float a)
{
  return int(a + 0.5);
}

/*определение параметров области построения*/
void init(void)
{
  glClearColor(1.0, 1.0, 1.0, 0.0);
  glMatrixMode(GL_PROJECTION);
  gluOrtho2D(0.0, 500.0, 0.0, 400.0);
  glPointSize(1);
}

/*функция, строящая точки*/
void setPixel(int x, int y)
```

```
{
  glBegin(GL_POINTS);
    glVertex2f(x, y);
  glEnd();
}

/*Симметричный ЦДА*/
void lineDDA(int x1, int y1, int x2, int y2)
{
  int dx=x2 - x1, dy=y2 - y1, steps, k;
  float xIn, yIn, x = x1, y = y1;
  steps = 300;
/*приводим переменные dx, dy и steps к типу floa*/t
  xIn = float(dx)/float(steps);
  yIn = float(dy)/float(steps);
  setPixel(round(x), round(y));
/*цикл от k до steps с шагом 1*/
  for(k = 0; k < steps; k++)
  {
    x += xIn; /*увеличение x на xIncrement*/
    y += yIn;
    setPixel(round(x), round(y));/*округляем значения
x и y, так как координаты пикселей могут иметь только
целочисленные значения*/
  }
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(0.0, 0.0, 1.0); /*задание цвета линии*/
  lineDDA(20, 20, 300, 390); /*задание x1,y1,x2,y2
соответственно*/
  glFlush();
}

/*основная функция программы*/
int main(int argc, char* argv[])
{
```

```
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
  glutInitWindowPosition(100, 100);
  glutInitWindowSize(500, 400);
  glutCreateWindow("DDA");
  init();
  glutDisplayFunc(display);
  glutMainLoop();
  return 0;
}
```

2. Несимметричный цифровой дифференциальный анализатор

В большинстве случаев гораздо лучше смотрятся вектора с единичным шагом по большей относительной координате (несимметричный ЦДА).

Для реализации этого алгоритма в первую очередь необходимо определить по какой оси наибольшее приращение, т.е. определить, что больше, *dx* или *dy*.

Пусть dx, $dy > 0$ и $dx > dy$. Координата по X увеличивается на 1 $(x_2-x_1)/dx$ раз, а координата Y тоже столько же раз, но на величину dy/dx.

Таким образом, количество узлов аппроксимации равно числу пикселей вдоль наибольшего приращения. Для построения отрезка необходимо:

1. Вычислить приращения координат:
```
dx= x2 - x1;
dy= y2 - y1;
```
2. Определить наибольшее приращение и сделать его равным шагу
   Если (dx) > (dy), то шаг равен dx, иначе шаг равен dy.
3. Вычислить приращения по x и по y
```
xin= dx /шаг;
yin= dy / шаг;
```
4. Занести начальную точку отрезка
```
PutPixel (x1, y1);
```
5. Сгенерировать отрезок:
```
while k<=шаг
{
x1=x1+ xin;
y1= y1 + yin;
PutPixel (x1, y1);
```

}

3. Алгоритм Брезенхема

Так как приращения координат, как правило, не являются целой степенью двойки, то в ЦДА-алгоритме, требуется выполнение деления, что не всегда желательно, особенно при аппаратной реализации.

Брезенхем предложил алгоритм, не требующий деления, но обеспечивающий минимизацию отклонения сгенерированного образа от истинного отрезка. Основная идея алгоритма состоит в том, что если угловой коэффициент прямой меньше 1/2, то естественно точку, следующую за точкой (0,0), поставить в позицию (1,0) (рис. 1.а), а если угловой коэффициент больше 1/2, то в позицию (1,1) (рис. 1.б). Для принятия решения куда заносить очередной пиксель вводится величина отклонения d точной позиции от середины между двумя возможными растровыми точками в направлении наименьшей относительной координаты. Знак d используется как критерий для выбора ближайшей растровой точки.

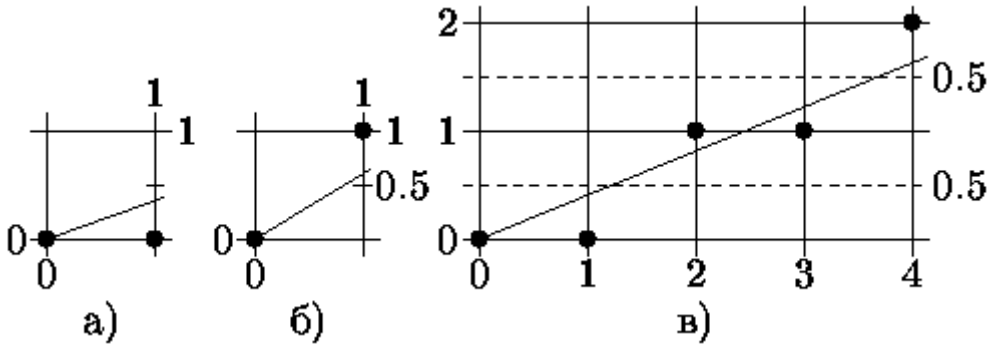


Рис. 1. Алгоритм Брезенхема для генерации отрезков

Если d<0, то точное *y*-значение округляется до последнего меньшего целочисленного значения *y*, т.е. *y*-координата не меняется по сравнению с предыдущей точкой. В противном случае *y* увеличивается на 1.

Для вычисления d без ограничения общности упрощающе положим, что рассматриваемый вектор начинается в точке (0,0) и проходит через точку (4, 1.5) (см. рис. 1.в), т.е. имеет положительный наклон меньший 1.

Из рис. 1.в видно, отклонение для первого шага $d_1 = \frac{d_y}{d_x} - \frac{1}{2} < 0$, поэтому для занесения пикселя выбирается точка (1,0).

Отклонение для второго шага вычисляется добавлением приращения *y*-координаты для следующей *x*-позиции (см. рис. 1.в):

$$d_2 = d_1 + \frac{d_y}{d_x} > 0,$$

поэтому для занесения пикселя выбирается точка (2,1). Так как отклонение считается от у-координаты, которая теперь увеличилась на 1, то из накопленного отклонения для вычисления последующих отклонений надо вычесть 1:

$$d_2 = d_2 - 1$$

Отклонение для третьего шага:

$$d_3 = d_2 + \frac{d_y}{d_x} < 0,$$

поэтому для занесения пикселя выбирается точка (3,1).

Таким образом, получается алгоритм, в котором используются только целые числа, сложение, вычитание и сдвиг:

1. Ввести координаты концов отрезка и поместить левый конец в точку с координатами (*x*, *y*).

```
X=x1;
Y=y1;
```

2. Поставить первую точку.

```
SetPixel(X,Y); /* Первая точка вектора*/
```

3. Вычислить:

```
dx=x2-x1;
dy=y2-y1;
2dy=2*dy;
k=2*(dy-dx);
d=2*dy-dx;
i=dx;
```

4. Выполнить построение остального отрезка

```
while(i=i-1>=0)
{
if (d>=0)
{
X=X+1;
Y=Y+1;
d=d+k;
}
Else
X=X+1;
d=d+2dy;
```

```
        SetPixel(X,Y); /* Очередная точка вектора*/
        }
```

Этот алгоритм пригоден для случая $0 \le dy \le dx$. Для других случаев алгоритм строится аналогичным образом.

Основы библиотеки OpenGL

Для тренировки возьмём простое приложение, строящее красный треугольник на чёрном фоне.

Листинг 2

```
#include <glut.h>
void init (void)
{
  glClearColor(0.0, 0.0, 0.0, 0.0);
  glMatrixMode(GL_PROJECTION);
  gluOrtho2D(0.0, 30.0, 0.0, 30.0);
}

/*ф-ция построения треугольника*/
void triangle(void)
{
/*построение треугольника по трём точкам*/
  glBegin(GL_TRIANGLES);
    glVertex2f(5.0, 5.0);
    glVertex2f(25.0, 5.0);
    glVertex2f(5.0, 25.0);
  glEnd();
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);/*очистка буфера
цвета*/
  glColor3f(1.0, 0.0, 0.0);/*выбор цвета*/
  triangle();
  glFlush();
}

int main(int argc, char* argv[])
{
  glutInit(&argc, argv);
```

```
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize(500, 500);
  glutInitWindowPosition(100, 100);
  glutCreateWindow("Example");
  init();
  glutDisplayFunc(display);
  glutMainLoop();
  return 0;
}
```

Обратите внимание: при изменении формы окна изображения рисунка будет искажаться. Для решения этой проблемы добавим функцию изменения формы окна.

Функция изменения формы окна

После создания рисунка часто возникает необходимость воспользоваться указателем мыши и переместить окно изображения в другое место на экране или изменить его размер. Изменение размеров окна изображения может привести к изменению его характеристического отношения и, следовательно, к искажению изначальной формы изображенных объектов.

Для компенсации изменения размеров окна изображения в библиотеке *GLUT* существует следующая процедура (относится к функциям двухмерного наблюдения)

```
glutReshapeFunc (reshape);
```

Эту функцию можно включить в процедуру main нашей программы, вместе с другими процедурами *GLUT*, и она будет активизироваться при каждом изменении размеров окна изображения. Аргументом этой функции *GLUT* является название процедуры, которой сообщаются новые значения ширины и высоты окна изображения. На основе новых размеров можно изменить параметры проектирования, а также выполнить любые другие операции, такие как изменение цвета окна изображения. Кроме того, новые значения высоты и ширины можно сохранить, чтобы использовать в других процедурах данной программы.

В качестве примера приведем программу, иллюстрирующую построение процедуры `reshape`. В функцию изменения формы входит команда `glLoadIdentity`, так что любые предыдущие значения проекционных параметров не влияют на образование новых проекций.

В примере, приведенном выше, изменим процедуру `main` (Листинг 3).

Листинг 3

```
int main(int argc, char* argv[])
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize(500, 500);
  glutInitWindowPosition(100, 100);
  glutCreateWindow("Example");
  init();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutMainLoop();
  return 0;
}
```

Далее необходимо добавить процедуру `reshape` (Листинг 4).

Листинг 4

```
void reshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if(w <= h)
     gluOrtho2D(0.0, 30.0, 0.0, 30.0 * (GLfloat) h /
(GLfloat) w);
  else
     gluOrtho2D(0.0, 30.0 * (GLfloat) w / (GLfloat) h,
0.0, 30.0);
}
```

Процедура `init` изменится (Листинг 5)

Листинг 5

```
void init (void)
{
  glClearColor(0.0, 0.0, 0.0, 0.0);
}
```

Обратите внимание: процедуру `reshape` необходимо вставить в такое место программы, чтобы она была доступна вызывающим процедурам (в данном случае процедуре `main`), т.е. выше по тексту. Например, можно расположить процедуру `reshape` после процедуры `display`.

Теперь никакие манипуляции с размером окна изображения не повлияют на характеристические отношения Вашего рисунка.

Рассмотрим более подробно функции, которые находятся в процедуре `reshape`.

Функции режима проектирования OpenGL

```
glMatrixMode(GL_PROJECTION);
```

Указанная команда присваивает проекционной матрице статус текущей (изначально текущей является единичная матрица). Впрочем, если предполагается отменить данную команду, например, чтобы получить другую проекцию сцены, инициализацию можно также проводить с помощью команды

```
glLoadIdentity(  );
```

Это гарантирует, что при каждом переходе в режим проектирования матрица становится единичной, так что новые параметры наблюдения не будут объединяться с предыдущими.

Функция отсекающего окна GLU (относится к функциям двухмерного наблюдения)

Чтобы определить двухмерное отсекающее окно, можно использовать функцию *OpenGL Utility*:

```
gluOrtho2D(xwmin,xwmax,ywmin,ywmax);
```

Координаты границ отсекающего окна указываются как числа с двойной точностью. Данная функция задает ортогональную проекцию для отображения сцены на экран. Для трехмерной сцены это означает, что объекты будут проектироваться вдоль параллельных линий, которые перпендикулярны двухмерному экрану *xy*. Однако в двухмерных приложениях объекты уже определены на плоскости *xy*. Следовательно, ортогональная проекция не влияет на нашу двухмерную сцену иным образом, кроме преобразования точек объекта в нормированные координаты. Тем не менее, ортогональную проекцию нужно задать, поскольку наша двухмерная сцена обрабатывается по полному трехмерному конвейеру наблюдения *OpenGL*. Фактически отсекающее окно можно задать с использованием трехмерной версии функции `gluOrtho2D` из корневой библиотеки *OpenGL*.

В процедурах отсечения *OpenGL* используются нормированные

координаты в диапазоне от -1 до 1. Функция `gluOrtho2D` задает трехмерную версию матрицы преобразования для отображения объектов в отсекающем окне в нормированные координаты. Объекты вне нормированного квадрата (и вне отсекающего окна) удаляются с отображаемой сцены.

Если в программе-приложении не задать отсекающее окно, будут использоваться координаты по умолчанию $(xw_{\min}, yw_{\min}) = (-1{,}0; -1{,}0)$ и $(xw_{\max}, yw_{\max}) = (1{,}0; 1{,}0)$. Следовательно, по умолчанию отсекающее окно — это нормированный квадрат с центром в начале координат и стороной 2.

Функция поля просмотра OpenGL (относится к функциям двухмерного наблюдения)

Параметры поля просмотра задаются с помощью следующей функции *OpenGL*.

```
glViewport(xvmin,yvmin,vpWidth,vpHeight);
```

Все значения параметров даются в целочисленных экранных координатах относительно окна на экране дисплея. Параметры *xvmin* и *yvmin* задают положение левого нижнего угла поля просмотра относительно левого нижнего угла окна на экране дисплея. Ширина и высота пикселя поля просмотра задаются параметрами *vpWidth* и *vpHeight*. Если в программе не вызывать функцию `glViewport`, по умолчанию размер и положение поля просмотра будут такими же, как у окна на экране дисплея.

После применения процедур отсечения, точки в нормированном квадрате переводятся в прямоугольник поля просмотра с использованием матрицы. Требуемые в данной матрице координаты правого верхнего угла поля просмотра вычисляются через ширину и высоту поля просмотра:

$$xv_{\max} = xv_{\min} + vpWidth,$$
$$yv_{\max} = yv_{\min} + vpHeight$$

Для окончательного преобразования в буфер регенерации в заданные положения загружаются цвета пикселей примитивов поля просмотра.

В *OpenGL* можно создавать несколько полей просмотра. Параметры текущего активного поля просмотра можно узнать, используя такую функцию запроса

```
glGetlntegerv(GL_VIEWPORT,vpArray);
```

Здесь *vpArray* - четырехэлементный массив с одним индексом. Данная функция `Get` возвращает параметры текущего поля просмотра в *vpArray* в порядке *xvmin*, *yvmin*, *vpWidth* и *vpHeight*. В интерактивных приложениях, например, данную функцию можно использовать для получения параметров поля просмотра, которое содержит курсор экрана.

Поскольку процедуры интерактивного ввода устройствами в программе *OpenGL* требуют сопряжения с системой окон, такой ввод обрабатывается с помощью процедур из *Utility Toolkit (GLUT).*

Функции GLUT (мышь)

Чтобы задать ("зарегистрировать") процедуру, вызываемую, когда указатель мыши находится в окне на экране дисплея и нажимается или отпускается клавиша мыши, используется следующая функция

```
glutMouseFunc(mouseFcn);
```

Эта функция обратного вызова процедуры `mouseFcn`, имеющей четыре аргумента:

```
void mouseFcn(GLint button,GLint action,GLint xMouse,GLint yMouse)
```

Параметру *button* присваивается значение символьной константы *GLUT*, обозначающей одну из трех кнопок мыши, а параметру *action* присваивается символьная константа, определяющая, какое действие мы хотим использовать, чтобы запустить событие активизации мыши. Допустимыми значениями *button* являются: *GLUT_LEFT_BUTTON*, *GLUT_MIDDLE_BUTTON* и *GLUT_RIGHT_BUTTON*. (Если есть только двухкнопочная мышь, то применяются только обозначения для левой и правой кнопок; при использовании однокнопочной мыши параметру *button* можно присвоить только значение *GLUT_LEFT_BUTTON*.) Действием, связанным с параметром, может быть *GLUT_DOWN* или *GLUT_UP* в зависимости от того, когда нужно инициировать действия - при нажатии или при отпускании клавиши. При вызове процедуры `mouseFcn` положение курсора мыши в окне на экране дисплея возвращается как координата (*xMouse, yMouse*). Это положение отсчитывается от верхнего левого угла окна на экране дисплея, так что *xMouse* — это расстояние (в пикселях) от левого края окна на экране дисплея, а *yMouse* — расстояние (в пикселях) вниз от вершины окна на экране дисплея.

Вернемся к нашему примеру:

Для удобства дальнейшей работы, изменим масштаб рисунка (Листинг 6)

Листинг 6

```
if(w <= h)
  gluOrtho2D(-10.0, 60.0, -10.0, 60.0 * (GLfloat) h / (GLfloat) w);
else
  gluOrtho2D(-10.0, 60.0 * (GLfloat) w / (GLfloat) h,
```

```
-10.0, 60.0);
```

Зададим координаты треугольника как глобальные переменные[1] (Листинг 7).

Листинг 7

```
GLfloat x1 = 5.0, y1 = 5.0, x2 = 25.0, y2 = 5.0, x3 =
5.0, y3 = 25.0;
```

Процедура `triangle` примет вид (Листинг 8)

Листинг 8

```
void triangle(void)
{
  glBegin(GL_TRIANGLES);
    glVertex2f(x1, y1);
    glVertex2f(x2, y2);
    glVertex2f(x3, y3);
  glEnd();
}
```

Добавим в процедуру main функцию обратного вызова процедуры `mouse` (Листинг 9)

Листинг 9

```
glutMouseFunc (mouse);
После этого процедура main примет вид:
int main(int argc, char* argv[])
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize(500, 500);
  glutInitWindowPosition(100, 100);
  glutCreateWindow("Example");
  init();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutMouseFunc (mouse);
  glutMainLoop();
```

[1] Вставьте эту строку после директив подключения заголовочных файлов

```
  return 0;
}
```

Создаем процедуру mouse, реагирующую на события мыши, можно расположить после reshape (Листинг 10).

Листинг 10

```
void mouse(GLint button, GLint action, GLint xMouse,
GLint yMouse)
{ /*в условиях равенство обозначается двумя знаками
«равно»
  if (button == GLUT_LEFT_BUTTON && action ==
GLUT_DOWN)
    { /*два знака «амперсанд» соответствуют
логическому И
        y1 = y1 + 2;
        y2 = y2 + 2;
        y3 = y3 + 2;
    }
  else
    if (button == GLUT_RIGHT_BUTTON)
    {
        y1 = y1 - 2;
        y2 = y2 - 2;
        y3 = y3 - 2;
    }
  glutPostRedisplay( );
}
```

Обратите внимание, что перемещение фигуры вверх происходит только при нажатии левой кнопки мыши, а перемещение фигуры вниз происходит и при нажатии, и при отпускании правой кнопки мыши.

Остановимся более подробно на функции glutPostRedisplay( );

Эта функция используется, чтобы указать, что содержимое текущего окна на экране нужно обновить.

Функции GLUT (клавиатура)

Для задания процедуры, вызываемой при нажатии клавиши, используется следующая функция.

```
glutKeyboardFunc(keyFcn);
```

Заданная процедура имеет три аргумента.

```
void keyFcn (GLubyte key, GLint xMouse, GLint yMouse)
```

Параметру *key* присваивается символьное значение или соответствующий код *ASCII*. Положение мыши в окне на экране дисплея возвращается как точка (*xMouse*, *yMouse*) относительно верхнего левого угла окна. При нажатии обозначенной клавиши на основе положения мыши можно инициировать некоторое действие, независимо от того, нажата ли еще какая-то кнопка мыши.

Для функциональных клавиш, клавиш управления курсором и других специальных клавиш можно использовать команду

```
glutSpecialFunc (specialKeyFcn);
```

Заданная процедура имеет три аргумента:

```
void specialKeyFcn (GLint specialKey, GLint xMouse, GLint yMouse)
```

Однако теперь параметру *specialKey* присваивается целочисленное значение символьной константы *GLUT*. Чтобы выбрать функциональную клавишу, используем одну из констант от *GLUT_KEY_F1* до *GLUT_KEY_F12*. Для клавиш управления курсором используем такие константы, как *GLUT_KEY_UP* и *GLUT__KEY_RIGHT*. Другие клавиши можно выделить, используя *GLUT_KEY__PAGE_DOWN*, *GLUT__KEY_HOME* и другие подобные константы для клавиш <Page Up>, <End> и <Insert>. Для использования клавиш <Backspace>, <Delete> и <Escape> нужна процедура `glutKeyboardFunc` и коды ASCII этих клавиш — 8, 127 и 27 соответственно.

Добавим в процедуру `main` функцию обратного вызова процедуры `keyboard` (Листинг 11).

Листинг 11

```
glutMouseFunc (keyboard);
```

После этого процедура `main` примет вид (Листинг 12).

Листинг 12

```
int main(int argc, char* argv[])
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize(500, 500);
  glutInitWindowPosition(100, 100);
  glutCreateWindow("Example");
  init();
```

```
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutMouseFunc (mouse);
  glutKeyboardFunc(keyboard);
  glutMainLoop();
  return 0;
}
```

Создаем процедуру `keyboard`, реагирующую на события клавиатуры, можно расположить после `mouse` (Листинг 13).

Листинг 13

```
void keyboard (GLubyte key, GLint xMouse, GLint
yMouse)
{ /*две вертикальных черты соответствуют логическому
ИЛИ
  if (key == 'r' || key == 'R')
    {
        x1 = x1 + 2;
        x2 = x2 + 2;
        x3 = x3 + 2;
    }
  else
    if (key == 'l' || key == 'L')
    {
        x1 = x1 - 2;
        x2 = x2 - 2;
        x3 = x3 - 2;
    }
  glutPostRedisplay( );
}
```

Предположим, нам необходимо нарисовать несколько раз один и тот же треугольник. Один из возможных способов – разделить окно на несколько частей и выводить изображение того же самого треугольника в разных частях окна. В *OpenGL* имеется специально предназначенная для этого функция: `glViewport` – функция расщепления окна.

Внесите изменения в процедуру `display` (Листинг 14)

Листинг 14

```
void display(void)
```

```
{
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(1.0, 0.0, 0.0);
  glViewport (0, 0, 250, 250);
  triangle();
  glColor3f(0.0, 0.0, 1.0);
  glViewport (250, 0, 250, 250);
  triangle();
  glColor3f(0.0, 1.0, 0.0);
  glViewport (250, 0, 250, 250);
  triangle();
  glColor3f(1.0, 1.0, 1.0);
  glViewport (250, 250, 250, 250);
  triangle();
  glFlush();
}
```

Самостоятельно:

1. На основе примера создайте линию согласно номеру варианта. Напишите функцию, рассчитывающую координаты конца отрезка, исходя из координат начала отрезка и тангенса наклона.

Варианты	Тангенс угла наклона
Вариант 1.	0.5
Вариант 2.	0.5
Вариант 3.	0.6
Вариант 4.	0.6
Вариант 5.	0.7
Вариант 6.	0.7
Вариант 7.	0.8
Вариант 8.	0.8
Вариант 9.	0.9
Вариант 10.	0.9

2. Измените процедуру lineDDA для несимметричного случая и постройте линию, согласно номеру варианта.

3. Измените процедуру lineDDA для алгоритма Брезенхема и постройте линию, согласно номеру варианта.

4. Создайте фигуру согласно номеру варианта, которая будет двигаться по диагонали, либо придумайте своё движение (по кругу, вращение и т.д.).

Варианты	Фигуры
1	Прямоугольник
2	Прямоугольник
3.	Квадрат
4.	Квадрат
5.	Ромб
6.	Ромб
7.	Трапеция
8.	Трапеция
9.	Дельтоид
10.	Дельтоид

Примечание: для четырёхугольных фигур необходимо использовать `GL_QUADS` вместо `GL_TRIANGLES`.