**Лабораторная работа №2**

**Растровая заливка геометрических фигур**

Цель работы: изучение алгоритмов растровой заливки основных геометрических фигур: кругов, многоугольников.

Порядок выполнения работы

1. Изучить растровые алгоритмы заливки геометрических фигур.
2. Разработать алгоритм и составить программу для построения на экране изображения в соответствии с номером варианта (по журналу старосты). В качестве исходных данных взять указанные в таблице №1 лаб. работы №1.

Требования к программе

1. Программа должна быть написана на языках Си или C++.
2. Фигуры, нарисованные в предыдущей лабораторной работе, необходимо заполнить цветом. Реализовать следующие способы заливки: барицентрическая, радиальная, секторная. Реализовать возможность выбора пользователем (например клавишами) различных способов заливки для каждой геометрической фигуры.
3. Изображение должно масштабироваться строго по центру окна с радиусом 7/8 относительно размера окна (см. пример проекта lab\_1\_basics.vcxproj).
4. Пользовать должен иметь возможность менять размер окна и изменять разрешение пикселей. См. пример проекта lab\_1\_basics.vcxproj, в котором разрешение изменяется клавишами F2/F3.
5. Если в задании указано, что требуется реализовать анимацию (например, вращение), то перерисовку изображения нужно выполнять по таймеру 30 раз в секунду.
6. Цвет примитивов выбрать по собственному усмотрению.

Содержание отчёта

1. Название темы.
2. Цель работы.
3. Постановка задачи.
4. Вывод необходимых геометрических формул для построения изображения.
5. Реализации алгоритмов заливки геометрических фигур.
6. Текст программы для рисования основных фигур.
7. Результат работы программы (снимки экрана).
8. Вывод о проделанной работе.

**Теоретические сведения**

Рассмотрим алгоритм растровой заливки треугольника. Пусть дан треугольник с вершинами , , , которые отсортированы таким образом, чтобы (рисунок). Треугольник разделяется прямой на два треугольника. Для начала необходимо провести через середину каждой строки горизонтальную линию и определить, где она пересекает стороны треугольника. На рисунке данные линии показаны красным цветом (для нижнего треугольника) и зелёным (для верхнего треугольника). Пиксели, внутри которых находятся точки пересечения горизонтальных линий со сторонами треугольника считаются *граничными*.



Координаты левого и правого граничных пикселей в строке пикселей с номером определяются так:

Поскольку – целочисленный номер, то нужно прибавить 1/2, чтобы линия проходила через центр пикселей (строки). Скобками вида обозначается округление в меньшую сторону.

После нахождения граничных пикселей нужно просто заполнить в буфере кадра пиксели между ними.

Текст программы для растровой закраски треугольника приведён в листинге.

void Triangle(float x0, float y0, float x1, float y1, float x2, float y2, COLOR color)

{

// Отсортируем точки таким образом, чтобы выполнилось условие: y0 < y1 < y2

if (y1 < y0)

{

swap(y1, y0);

swap(x1, x0);

}

if (y2 < y1)

{

swap(y2, y1);

swap(x2, x1);

}

if (y1 < y0)

{

swap(y1, y0);

swap(x1, x0);

}

// Определяем номера строк пикселей, в которых располагаются точки треугольника

int Y0 = (int) (y0 + 0.5f);

int Y1 = (int) (y1 + 0.5f);

int Y2 = (int) (y2 + 0.5f);

// Отсечение невидимой части треугольника

if (Y0 < 0) Y0 = 0;

else if (Y0 >= height) Y0 = height;

if (Y1 < 0) Y1 = 0;

else if (Y1 >= height) Y1 = height;

if (Y2 < 0) Y2 = 0;

else if (Y2 >= height) Y2 = height;

// Рисование верхней части треугольника

for (float y = Y0 + 0.5f; y < Y1; y++)

{

int X0 = (int) ((y - y0) / (y1 - y0) \* (x1 - x0) + x0);

int X1 = (int) ((y - y0) / (y2 - y0) \* (x2 - x0) + x0);

if (X0 > X1) swap(X0, X1);

if (X0 < 0) X0 = 0;

if (X1 >= width) X1 = width;

for (int x = X0; x < X1; x++)

{

// f(x + 0.5, y)

SetPixel(x, y, color);

}

}

// Рисование нижней части треугольника

for (float y = Y1 + 0.5f; y < Y2; y++)

{

int X0 = (int)((y - y1) / (y2 - y1) \* (x2 - x1) + x1);

int X1 = (int)((y - y0) / (y2 - y0) \* (x2 - x0) + x0);

if (X0 > X1) swap(X0, X1);

if (X0 < 0) X0 = 0;

if (X1 >= width) X1 = width;

for (int x = X0; x < X1; x++)

{

// f(x + 0.5, y)

SetPixel(x, y, color);

}

}

}

Данную подпрограмму необходимо оптимизировать, исключив повторяющиеся вычисления внутри циклов. Достаточно заметить, что неокруглённые значения внутри скобок всегда увеличиваются на одно и тоже значение.

Рассмотренная подпрограмма закрашивает треугольник одним цветом. Таким образом треугольник приобретает однотонную заливку. Рассмотрим простейший алгоритм создания градиентной заливки. Сопоставим каждой вершине треугольника свой цвет. На самом деле сопоставляемый параметр может быть любым, который может изменяться линейно по поверхности фигуры – удалённость, вектор нормали, координата текстуры и др.

Пусть точка имеет цвет , – , – . Цвет точки определяется из соотношения площадей образуемых ею треугольников , , согласно следующей пропорции:

Обозначим , , . Произвольный параметр , заданный в трёх вершинах треугольника значениями , , интерполируется по формуле:

Рассмотренный способ интерполяции называется *барицентрической интерполяцией*.

Реализуем расчёт барицентрической интерполяции по формулам выше в виде отдельного шаблонного класса BarycentricInterpolator. Пусть данный класс при конструировании принимает цвета и координаты вершин треугольника, а единственный метод getColor(x, y) возвращает цвет точки с координатами .

class BarycentricInterpolator

{

float x0, y0, x1, y1, x2, y2, S;

COLOR C0, C1, C2;

public:

BarycentricInterpolator(float \_x0, float \_y0, float \_x1, float \_y1, float \_x2, float \_y2, COLOR A0, COLOR A1, COLOR A2) :

x0(\_x0), y0(\_y0), x1(\_x1), y1(\_y1), x2(\_x2), y2(\_y2),

S((\_y1 - \_y2)\*(\_x0 - \_x2) + (\_x2 - \_x1)\*(\_y0 - \_y2)), C0(A0), C1(A1), C2(A2)

{

}

COLOR color(float x, float y)

{

// Барицентрическая интерполяция

float h0 = ((y1 - y2) \* (x - x2) + (x2 - x1) \* (y - y2)) / S;

float h1 = ((y2 - y0) \* (x - x2) + (x0 - x2) \* (y - y2)) / S;

float h2 = 1 - h0 - h1;

float r = h0 \* C0.RED + h1 \* C1.RED + h2 \* C2.RED;

float g = h0 \* C0.GREEN + h1 \* C1.GREEN + h2 \* C2.GREEN;

float b = h0 \* C0.BLUE + h1 \* C1.BLUE + h2 \* C2.BLUE;

float a = h0 \* C0.ALPHA + h1 \* C1.ALPHA + h2 \* C2.ALPHA; // Из-за погрешности аппроксимации треугольника учитываем, что центр закрашиваемого пикселя может находится вне треугольника.

// По этой причине значения r, g, b могут выйти за пределы диапазона [0, 255].

if (r >= 256) r = 255;

if (g >= 256) g = 255;

if (b >= 256) b = 255;

if (r < 0) r = 0;

if (g < 0) g = 0;

if (b < 0) b = 0;

return COLOR(r, g, b, a);

}

};

В функции Triangle достаточно изменить заголовок и внутренние циклы, добавив в них вызов функции getColor:  
template <class InterpolatorClass>

void Triangle(float x0, float y0, float x1, float y1, float x2, float y2, InterpolatorClass& Interpolator)

//...

for (int x = X0; x <= X1; x++)

{

COLOR color = Interpolator.getColor(x + 0.5f, y);

SetPixel(x, y, color);

}

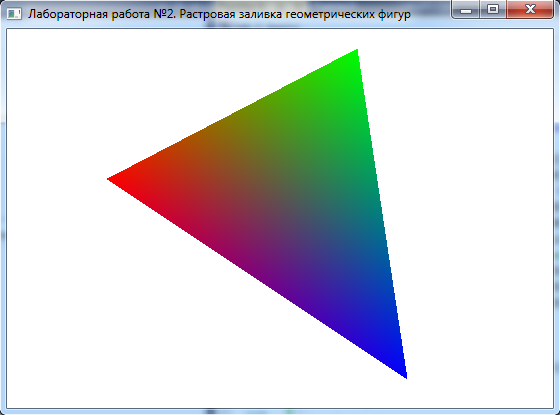
Вызов функции Triangle осуществляется следующим образом:

float x0 = 300, y0 = 20, x1 = 50, y1 = 150, x2 = 350, y2 = 350;

BarycentricInterpolator interpolator(x0, y0, x1, y1, x2, y2, COLOR(0, 255, 0), COLOR(255, 0, 0), COLOR(0, 0, 255));

frame.Triangle(x0, y0, x1, y1, x2, y2, interpolator);

Результат вызова представлен на рисунке.



Функции, подобные getColor называют функциями-шейдерами. Это название происходит от английского слова «shade», которое переводится как «тень». Изначально данное название означало графический эффект затенения, для расчёта которого разрабатывались специальные функции. Поэтому такие функции стали называть «шейдерами». В настоящее время шейдерами уже называют универсальные функции, которые рассчитывают разнообразные эффекты, например, освещение, текстурирование, преломление света, лучи, артефакты, туман и различные другие.

Напишем функцию, которая выполняет радиальную заливку прямоугольника.

class RadialInterpolator

{

float cx, cy; // Центр прямоугольника

COLOR C0, C1; // Цвета радиальной заливки

float angle; // Начальный угол заливки

public:

RadialInterpolator(float \_x0, float \_y0, float \_x1, float \_y1, COLOR A0, COLOR A1, float \_angle) :

cx((\_x0 + \_x1) / 2.0f), cy((\_y0 + \_y1)/ 2.0f),

C0(A0), C1(A1), angle(\_angle)

{

}

COLOR color(float x, float y)

{

double dx = (double)x - cx, dy = (double)y - cy;

double radius = sqrt(dx\*dx + dy\*dy);

float h0 = (sin(radius / 10 + angle) + 1.0f) / 2;

float h1 = 1 - h0;

float r = h0 \* C0.RED + h1 \* C1.RED;

float g = h0 \* C0.GREEN + h1 \* C1.GREEN;

float b = h0 \* C0.BLUE + h1 \* C1.BLUE;

return COLOR(r, g, b);

}

};

Создадим прямоугольник, который закрывает весь экран и применим к нему написанную функцию:

float x0 = 0, y0 = 0, x1 = frame.width, y1 = frame.height;

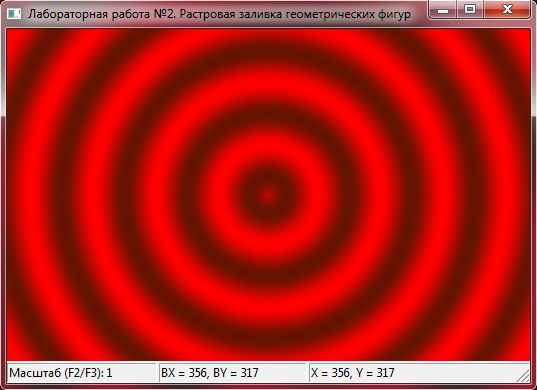
RadialInterpolator radialInterpolator(x0, y0, x1, y1,

COLOR(255, 0, 0), COLOR(100, 20, 0), global\_angle);

frame.Triangle(x0, y0, x0, y1, x1, y0, radialInterpolator);

frame.Triangle(x0, y1, x1, y0, x1, y1, radialInterpolator);

Результат закраски представлен на рисунке ниже.



На рисунке ниже показан пример секторной заливки.

