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 первой лабораторной работе, необходимо заполнить цветом. Реализовать следующие способы заливки: барицентрическая, радиальная, секторная. Изучить пример программы lab_2_colored_square.vcxproj. Реализовать возможность выбора пользователем (например, клавишами) различных способов заливки для каждой геометрической фигуры.
3. Программа должна реагировать на выделение пользователем фигур, когда он выбирает их с использованием кнопок мыши. Фигура должна подсвечиваться другой текстурой или цветом, когда она выбрана после наведения (клика) на неё курсора мыши.
4. В программе должна быть предусмотрена возможность изменять прозрачность фигур. Фигуры должны быть нарисованы в порядке убывания площади, чтобы большие фигуры не закрывали маленькие.
5. Изображение должно масштабироваться строго по центру окна с радиусом 7/8 относительно размера окна (см. пример проекта lab_1_basics.vcxproj).
6. 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.
7. Если в задании указано, что требуется реализовать анимацию (например, вращение), то перерисовку изображения нужно выполнять по таймеру 30 раз в секунду.
8. Цвет примитивов выбрать по собственному усмотрению.

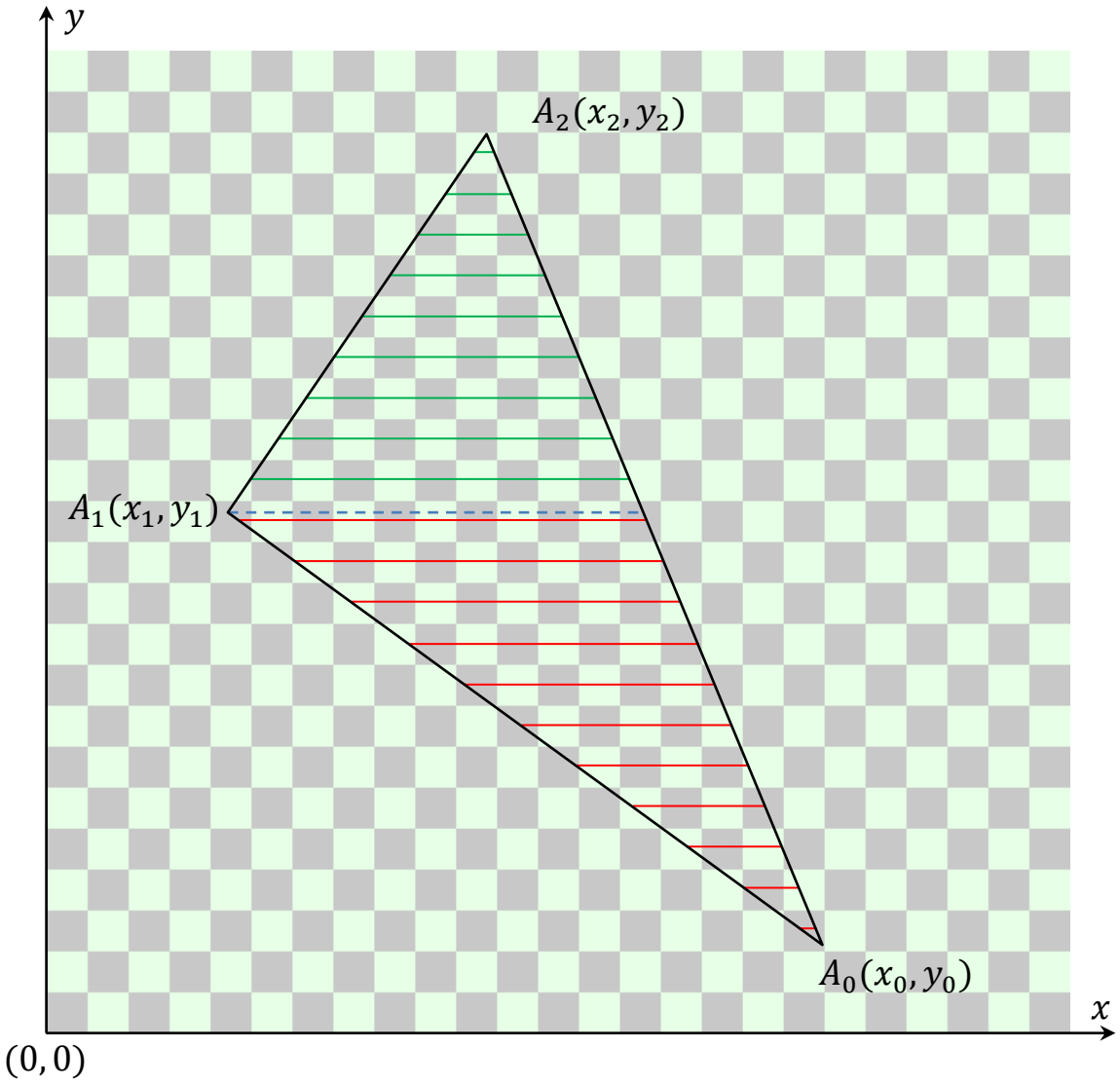
Содержание отчёта

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 $A_0(x_0, y_0)$, $A_1(x_1, y_1)$, $A_2(x_2, y_2)$, которые отсортированы таким образом, чтобы $y_0 < y_1 < y_2$ (рисунок). Треугольник разделяется прямой $y = y_1$ 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, около которых находятся точки пересечения горизонтальных линий со сторонами треугольника считаются *граничными*.



Координаты левого (X_0) и правого (X_1) граничных пикселей в строке пикселей с номером Y определяются так:

$$X_0 = \left\lfloor \frac{Y + 1/2 - y_0}{y_1 - y_0}(x_1 - x_0) + x_0 + \frac{1}{2} \right\rfloor,$$

$$X_1 = \left\lfloor \frac{Y + 1/2 - y_0}{y_2 - y_0}(x_2 - x_0) + x_0 + \frac{1}{2} \right\rfloor - 1.$$

Поскольку Y – целочисленный номер, то нужно прибавить к нему 1/2, чтобы линия проходила через центр пикселей (строки). Граничный пиксель большей частью своей площади должен находиться внутри треугольника, поэтому внутри скобки к x_0 добавляется 1/2. Скобками вида $\lfloor \cdot \rfloor$ обозначается округление в меньшую сторону.

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

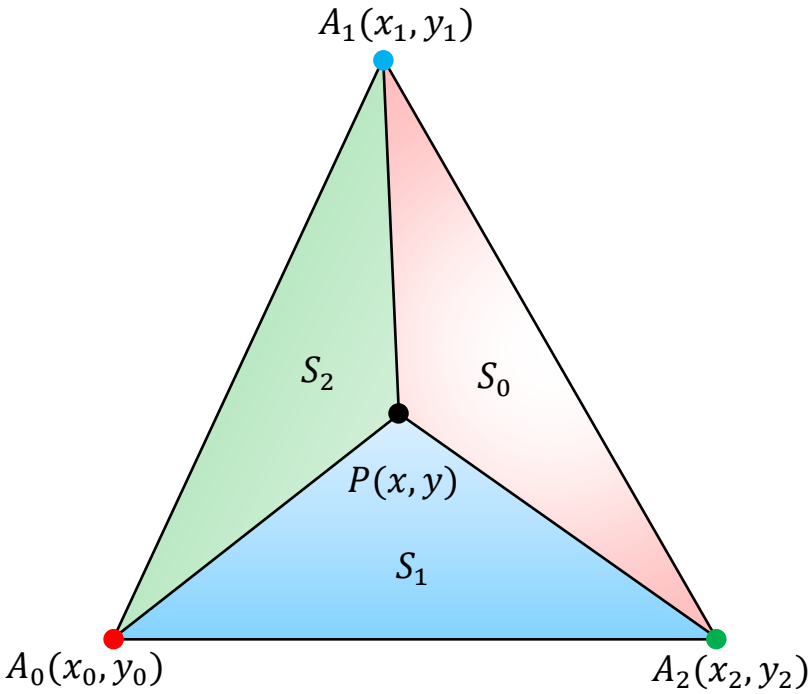
Текст программы для растровой закраски треугольника приведён в листинге.

```
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 + 0.5f);
		int X1 = (int) ((y - y0) / (y2 - y0) * (x2 - x0) + x0 + 0.5f);
```

```
            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 + 0.5f);
            int X1 = (int)((y - y0) / (y2 - y0) * (x2 - x0) + x0 + 0.5f);

            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 $\lfloor \cdot \rfloor$ 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 A_0 имеет цвет (r_0, g_0, b_0), A_1 – (r_1, g_1, b_1), A_2 – (r_2, g_2, b_2). Цвет точки $P(x, y)$ определяется из соотношения площадей образуемых ею треугольников S_0, S_1, S_2 согласно следующей пропорции:

$$r = \frac{S_0 r_0 + S_1 r_1 + S_2 r_2}{S}, \qquad g = \frac{S_0 g_0 + S_1 g_1 + S_2 g_2}{S},$$

$$b = \frac{S_0 b_0 + S_1 b_1 + S_2 b_2}{S}, \qquad S_0 = \begin{vmatrix} x & y & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix},$$

$$S_1 = \begin{vmatrix} x & y & 1 \\ x_0 & y_0 & 1 \\ x_2 & y_2 & 1 \end{vmatrix}, \qquad S_2 = \begin{vmatrix} x & y & 1 \\ x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \end{vmatrix}, \qquad S = \begin{vmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix}.$$

Обозначим $\lambda_0 = \frac{S_0}{S}$, $\lambda_1 = \frac{S_1}{S}$, $\lambda_2 = 1 - \lambda_0 - \lambda_1$. Произвольный параметр p, заданный в трёх вершинах треугольника значениями p_0, p_1, p_2 интерполируется по формуле:

$$p = \lambda_0 p_0 + \lambda_1 p_1 + \lambda_2 p_2.$$

$$\lambda_0 = \frac{(y_1 - y_2)(x - x_2) + (x_2 - x_1)(y - y_2)}{(y_1 - y_2)(x_0 - x_2) + (x_2 - x_1)(y_0 - y_2)},$$

$$\lambda_1 = \frac{(y_2 - y_0)(x - x_2) + (x_0 - x_2)(y - y_2)}{(y_1 - y_2)(x_0 - x_2) + (x_2 - x_1)(y_0 - y_2)},$$

$$\lambda_2 = \frac{(y_0 - y_1)(x - x_1) + (x_1 - x_0)(y - y_1)}{(y_1 - y_2)(x_0 - x_2) + (x_2 - x_1)(y_0 - y_2)}.$$

Рассмотренный способ интерполяции называется *барицентрической интерполяцией*. Если какое-то из значений $\lambda_0, \lambda_1, \lambda_2$ меньше нуля, то точка (x, y) находится вне треугольника. Это простое условие можно использовать для определения принадлежности точки треугольнику, при этом знаменатель (площадь S) в этих параметрах можно не вычислять, так как он всегда положительный.

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 (x, y).

```
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 getColor(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;
                return COLOR(r, g, b, a);
        }
};
```

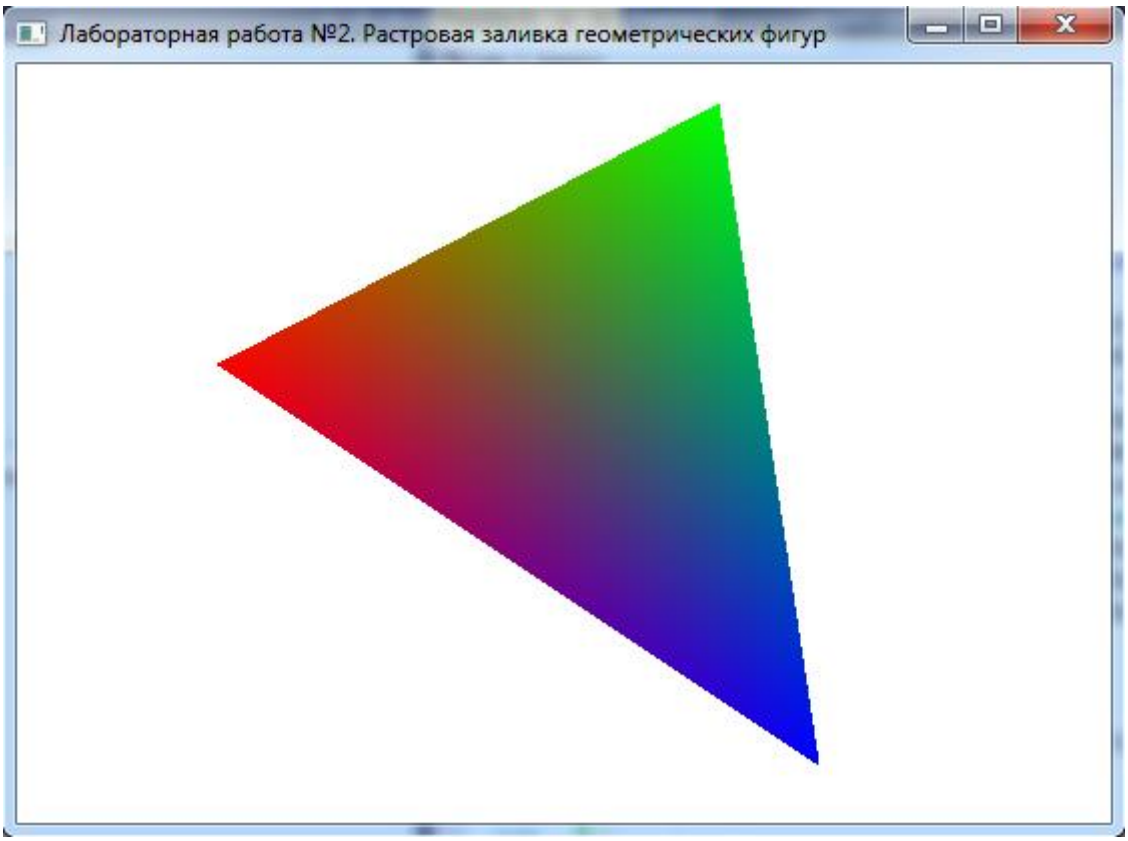
В функции Triangle достаточно изменить заголовок и внутренние циклы, добавив в них вызов функции getColor:

```
template <class ShaderClass>
    void Triangle(float x0, float y0, float x1, float y1, float x2, float y2,
ShaderClass & shader)
    //...
    for (int x = X0; x < X1; x++)
    {
      COLOR color = shader.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 и даже вычислительно трудные эффекты, например, освещение, текстурирование, преломление света, лучи «бога», артефакты, туман и многие другие.

Напишем функцию, которая выполняет радиальную заливку прямоугольника.

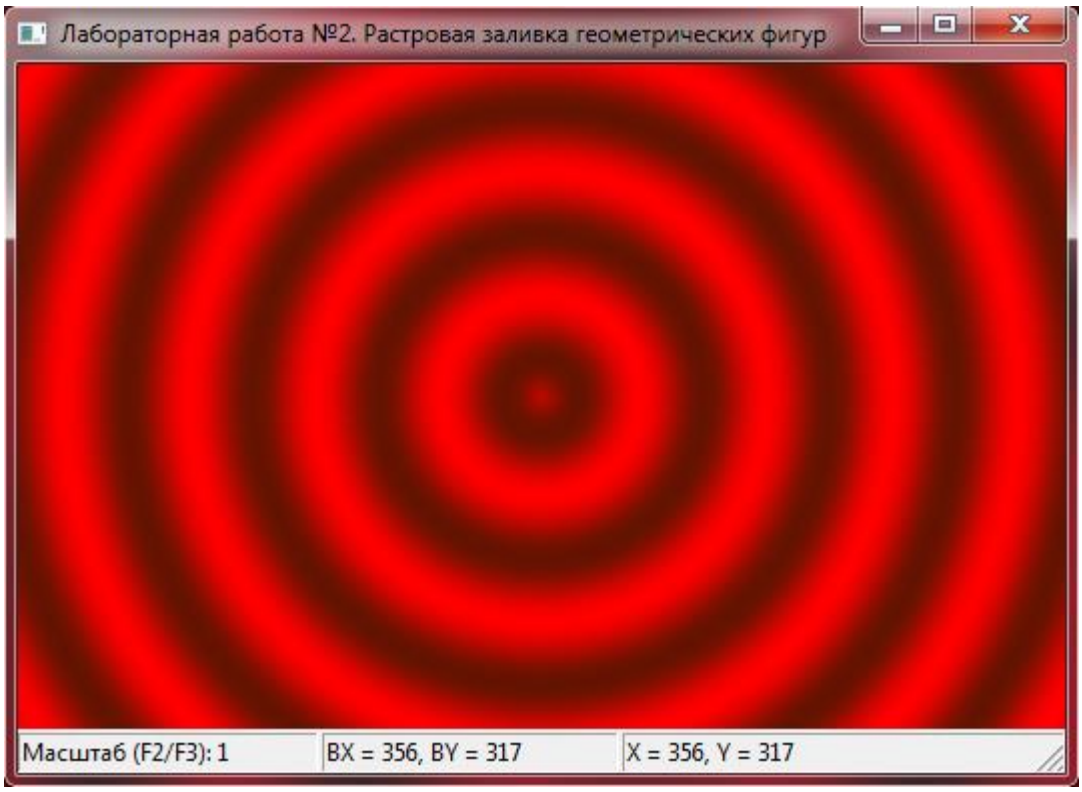
```
class RadialBrush
{
 float cx, cy; // Центр прямоугольника
 COLOR C0, C1; // Цвета радиальной заливки
 float angle;  // Начальный угол заливки

public:
 RadialBrush (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 getColor(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;
RadialBrush radialBrush(x0, y0, x1, y1,
        COLOR(255, 0, 0), COLOR(100, 20, 0), global_angle);
frame.Triangle(x0, y0, x0, y1, x1, y0, radialBrush);
frame.Triangle(x0, y1, x1, y0, x1, y1, radialBrush);
```

Результат закраски представлен на рисунке ниже.



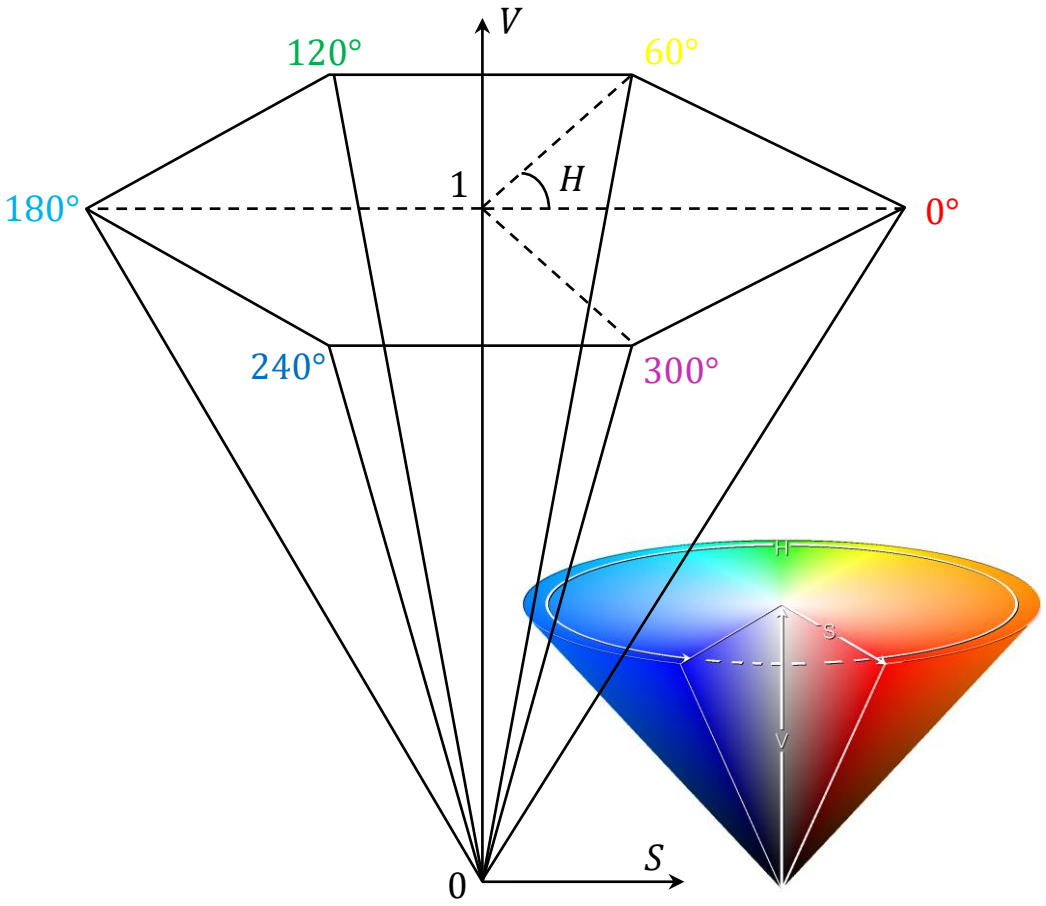
Растровая заливка круга

Пусть дана окружность с центром в точке (x_0, y_0) и радиусом r. Самый простой алгоритм растровой заливки круга можно описать следующим образом. Опишем вокруг окружности квадрат с противоположными вершинами $(x_0 - r, y_0 - r)$, $(x_0 + r, y_0 + r)$ и будем построчно его растеризовать. Для каждого пикселя (X, Y) квадрата нужно проверить условие – если его серединная точка $\left(X + \frac{1}{2}, Y + \frac{1}{2}\right)$ находится внутри заданного круга, то закрасить пиксель, иначе не закрашивать. Условие нахождения точки внутри круга рассмотрено в лаб. работе №1. Такое решение простое в реализации, но не самое оптимальное с точки зрения производительности. Для оптимального решения данной задачи необходимо

вычислять координаты граничных точки по алгоритму Брезенхейма, который был рассмотрен в лаб. работе №1.

Цветовая модель HSV

Цветовой формат RGB больше подходит для работы с цветопередающей аппаратурой. Но для человеческого глаза более характерно восприятие чистых тонов, поэтому в компьютерной анимации чаще используется цветовая модель HSV, которая позволяет оперировать интуитивно понятными терминами насыщенности, тона и яркости цвета. Аббревиатура HSV происходит от слов hue (тон), saturation (насыщенность), value (значение яркости). Множество допустимых цветов в модели HSV можно представить в виде перевёрнутой 6-гранной пирамиды, построенной в цилиндрической системе координат. Каждой вершине основания пирамиды поставлен в соответствие определённые угол и цвет. Красному цвету соответствует угол в 0°, жёлтому – 60°, зелёному – 120°, голубому – 180°, синему – 240°, малиновому – 300°. Наиболее ярким цветам соответствует основание пирамиды, лежащей на плоскости V=1. При движении вниз цвета становятся не такими яркими, т.е. более тёмными. Центр основания пирамиды соответствует белому цвету (S = 1, V = 1), вершина пирамиды (V = 0) – чёрному цвету. Таким образом на оси V располагаются серые тона. Тон H представляет собой угол поворота относительно вершины, соответствующей красному цвету. Величина S изменяется от нуля на оси OV до единицы на гранях пирамиды. Для преобразования цвета из формата HSV в RGB, нужно для точки, заданной в формате HSV, интерполировать цвета в нужном секторе пирамиды в системе координат RGB, которая образована цветами, заданными в вершинах пирамиды.

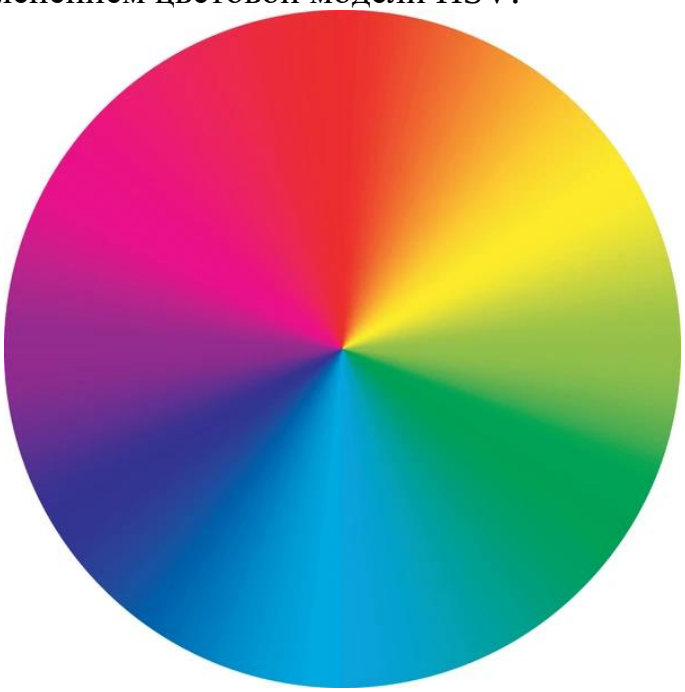


Текст программы для преобразования цвета из формата HSV в формат RGB представлен в листинге.

```
void ColorFromHSV(double hue, double saturation, double value, GLint *color)
{
    int hi = int(floor(hue / 60)) % 6;
    double f = hue / 60 - floor(hue / 60);

    value = value * 255;
    int v = (int)(value);
    int p = (int)(value * (1 - saturation));
    int q = (int)(value * (1 - f * saturation));
    int t = (int)(value * (1 - (1 - f) * saturation));

    if (hi == 0)
    {
      color[0] = v;
      color[1] = t;
      color[2] = p;
    }
    else if (hi == 1)
    {
      color[0] = q;
      color[1] = v;
```

```
    color[2] = p;
  }
  else if (hi == 2)
  {
    color[0] = p;
    color[1] = v;
    color[2] = t;
  }
  else if (hi == 3)
  {
    color[0] = p;
    color[1] = q;
    color[2] = v;
  }
  else if (hi == 4)
  {
    color[0] = t;
    color[1] = p;
    color[2] = v;
  }
  else
  {
    color[0] = v;
    color[1] = p;
    color[2] = q;
  }
}
```

На рисунке показан пример секторной заливки круга, которую можно изобразить с применением цветовой модели HSV.



Рисование полупрозрачных фигур

Рассмотрим далее алгоритм рисования полупрозрачных фигур, который основан на *альфа-смешивании*. Альфа-канал – величина прозрачности фигуры. Абсолютно прозрачной фигуре соответствует значение альфа-канала $\alpha = 0$, непрозрачной – $\alpha = 1$. Альфа-смешивание цветов заключается в том, что при рисовании фигуры с прозрачностью $0 \leq \alpha \leq 1$ и цветом (r, g, b) поверх другой фигуры, которая была отрисована ранее, необходимо обновить цвета фона (r', g', b') в буфере кадра по следующим простым формулам:

$$r = r\alpha + r'(1 - \alpha),$$
$$g = g\alpha + g'(1 - \alpha),$$
$$b = b\alpha + b'(1 - \alpha).$$

Следующий фрагмент программы на языке C++ выполнит обновление пикселя в буфере кадра по координатам (x, y).

```
// Для рисования полупрозрачных фигур будем использовать альфа-смешивание
if (color.ALPHA < 255)
{
  COLOR written = matrix[(int)y][x]; // Уже записанное в буфере кадра значение цвета,
т.е. цвет фона
  float a = color.ALPHA / 255.0f, b = 1 - a;
  color.RED =   color.RED   * a + written.RED   * b;
  color.GREEN = color.GREEN * a + written.GREEN * b;
  color.BLUE =  color.BLUE  * a + written.BLUE  * b;
}
SetPixel(x, y, color);
```