

Лабораторная работа №1

Растровые алгоритмы

Цель работы: изучение алгоритмов Брезенхейма растеризации графических примитивов: отрезков, окружностей.

Порядок выполнения работы

1. Изучить целочисленные алгоритмы Брезенхейма для растеризации окружности и линии.
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.

Требования к программе

1. Программа должна быть написана на языках Си или C++.
2. Алгоритмы растеризации окружности и линии должны быть максимально оптимизированы и реализованы как целочисленные (алгоритмы Брезенхейма).
3. Изображение должно масштабироваться строго по центру окна с радиусом $7/8$ относительно размера окна (см. пример проекта lab_1_basics.vcxproj).
4. Пользователь 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 Брезенхейма для рисования отрезка и окружности.
6. Текст программы для рисования основных фигур.
7. Результат работы программы (снимки экрана).
8. Вывод о проделанной работе.

Теоретические сведения

Компьютерная графика – наука, которая изучает алгоритмы преобразования объектов в их изображения. Изображение представляет собой матрицу, в которой каждая ячейка представляет собой цветной пиксель. Такой способ хранения изображения естественным образом следует из конструктивных особенностей LCD-мониторов. Каждый пиксель матрицы монитора состоит из трёх светодиодов: красного, зелёного, синего (Рис. 1).

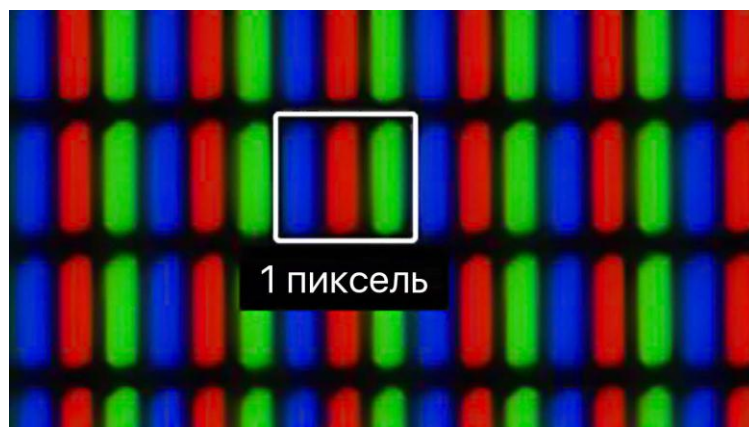


Рисунок 1 – Увеличенное изображение пикселей монитора

В свою очередь каждый светодиод может гореть с разной яркостью, что позволяет им всем вместе формировать всю палитру цветовых оттенков.

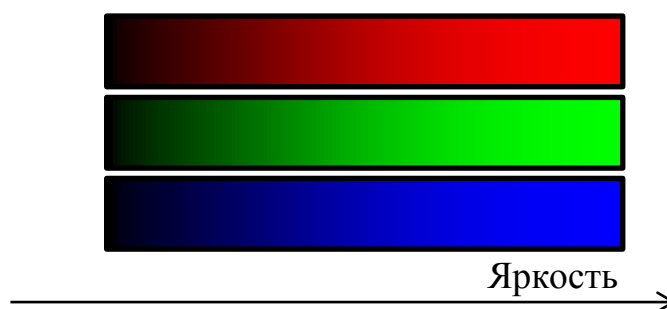


Рисунок 2 – Оттенки основных цветов

Если горит только один светодиод, то пиксель воспринимается как красный, зелёный или синий определённой насыщенности (Рис. 2). Если не горит ни один светодиод, то пиксель имеет чёрный цвет, если все три светодиода загораются максимальной яркостью – то белый. Формирование всех остальных цветов, который воспринимает человеческий глаз, происходит путём смешения этих трёх цветов разной насыщенности. Описанный формат представления цвета называется RGB (от англ. *red, green, blue*).

Если пиксели расположены на экране монитора в виде матрицы, то самым естественным способом хранения изображений в оперативной памяти является двумерный массив структур, в каждой из которых будет храниться цвет пикселя.

Определим структуру данных COLOR для хранения цвета:

```
typedef struct tagCOLOR
{
    unsigned char RED;           // Компонента красного цвета
    unsigned char GREEN;        // Компонента зелёного цвета
    unsigned char BLUE;         // Компонента синего цвета
    unsigned char ALPHA;        // Прозрачность (альфа-канал)
} COLOR;
```

В задачах компьютерной графики часто возникает задача рисования полупрозрачных объектов, поэтому для этого предусматривают отдельную переменную, в которую записывают величину прозрачности. Её принято называть альфа-каналом. Если альфа-канал имеет нулевое значение, то изображение прозрачное, при 128 – полупрозрачное, при 255 – абсолютно непрозрачное. В нашей структуре данных COLOR все цветовые поля имеют тип unsigned char, значит они могут принимать значения от нуля (минимальная интенсивность цвета) до 255 (максимальная интенсивность цвета).

Специально отведённая область памяти, в которой хранится информация о пикселях изображения, называется *буфером кадра*. Т.е. изображение – это то, что видит пользователь на экране монитора или бумаге, а буфер кадра – соответствующая данному изображению матрица пикселей типа COLOR, которая располагается в памяти ЭВМ. В задачах компьютерной графики принято сначала рисовать изображение внутри памяти, а уже потом выводить его на экран. Если «рисовать» изображение сразу на экране монитора, то пользователь наблюдает неприятные эффекты мерцания пикселей.

Обозначим ширину изображения как WIDTH, высоту – как HEIGHT. Определим структуру данных Frame для хранения буфера кадра и создадим в ней два метода для получения (GetPixel) и задания (SetPixel) цвета пикселя.

```
const int WIDTH = 500;           // Ширина изображения
const int HEIGHT = 500;         // Высота изображения

// Буфер кадра
class Frame
{
    // Матрица пикселей
    COLOR matrix[HEIGHT][WIDTH];

public:
    // Задаёт цвет color пикселю с координатами (x, y)
    void SetPixel(int x, int y, COLOR color)
    {
        matrix[y][x] = color;
    }

    // Возвращает цвет пикселя с координатами (x, y)
    COLOR GetPixel(int x, int y)
    {
        return matrix[y][x];
    }
}
```

Такой класс несовершенен ввиду того, что матрица `matrix` должна быть динамической, т.е. иметь произвольный размер, потому что пользователь имеет привычку менять размер изображений, окошек, разворачивать приложение на весь экран, запускать программу на разных дисплеях. Поэтому, будет очень печально, если программа будет поддерживать только фиксированный размер изображения. Динамической предлагается сделать эту матрицу читателю самостоятельно.

Далее необходимо реализовать целочисленные методы для рисования простейших геометрических примитивов – кругов и отрезков. Процесс закрашивания пикселей в буфере кадра, которые располагаются на линии отрезка или круга называется *растеризацией*. Определим методы для рисования отрезка, соединяющего пиксели с координатами (x_1, y_1) , (x_2, y_2) . Запишем каноническое уравнение прямой, проходящей через соответствующие данным пикселям точки:

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1}.$$

Обозначим $\Delta x = x_2 - x_1$, $\Delta y = y_2 - y_1$ и перепишем уравнение прямой таким образом, чтобы оно не содержало операции деления:

$$(x - x_1)\Delta y - (y - y_1)\Delta x = 0.$$

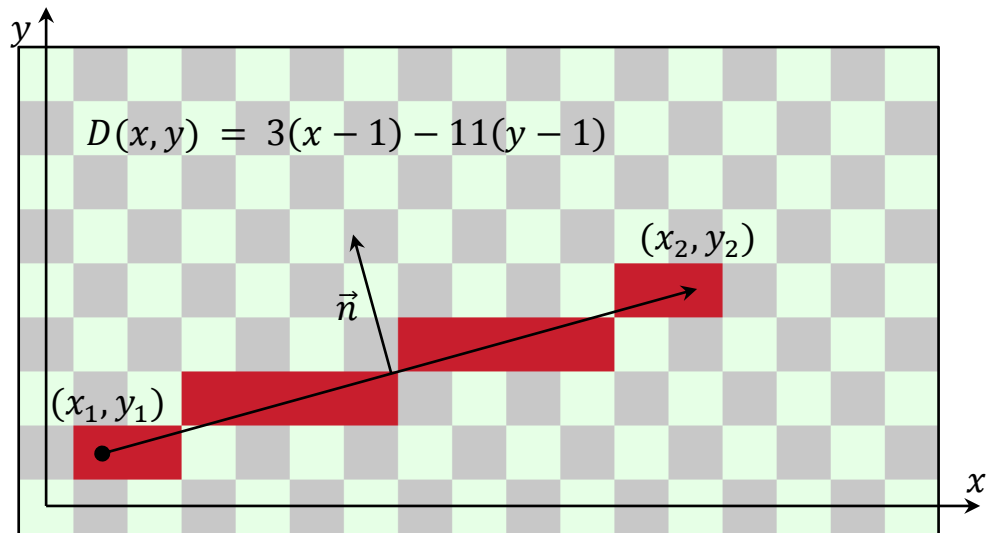
Рассмотрим функцию

$$D(x, y) = (x - x_1)\Delta y - (y - y_1)\Delta x.$$

Функция $\frac{D(x, y)}{\sqrt{(\Delta x)^2 + (\Delta y)^2}}$ выражает расстояние от точки (x, y) до прямой.

Можно также заметить, что знак функции $D(x, y)$ зависит от того, с какой стороны от прямой находится точка (x, y) . Если она находится в той полуплоскости, в которую направлен вектор нормали к прямой $\vec{n} = (\Delta y, -\Delta x)$, то $D(x, y) < 0$; если в другой полуплоскости, то $D(x, y) > 0$.

Идея Брезенхейма в растеризации отрезка заключается в том, что на каждой итерации одна из компонент (x или y) пикселей обязательно изменятся на ± 1 , а вторая выбирается таким образом, чтобы значение $|D(x, y)|$ было наименьшим. Рассмотрим случай (рисунок), когда точка (x_2, y_2) находится правее и выше точки (x_1, y_1) и $\Delta x > \Delta y$. Самым первым закрашивается пиксель с координатами (x_1, y_1) . Очевидно, что на каждой следующей итерации нужно обязательно сдвигаться на один пиксель вправо по оси абсцисс. Но как определить, когда нужно сдвигаться по оси ординат, т.е. вверх? Какой пиксель выбрать для закраски: (x, y) или $(x, y + 1)$? Для этого используем функцию $D(x, y)$. Можно проверить на рисунке, что $D(x, y + 1)$ всегда меньше нуля. Если $D(x, y) \geq -D(x, y + 1)$, значит точка $(x, y + 1)$ ближе к прямой и её нужно закрасить. Иначе закрашиваем точку (x, y) .



Фрагмент программы (неоптимизированный), который растеризует отрезок, представленный на рисунке выше:

```
int x1 = 1, y1 = 1, x2 = 12, y2 = 4;
int dy = y2 - y1, dx = x2 - x1, y = y1;
for (int x = x1; x <= x2; x++)
{
    int D1 = (x - x1) * dy - (y - y1) * dx;
    int D2 = (x - x1) * dy - (y + 1 - y1) * dx;
    if (D1 < -D2)
    {
        SetPixel(x, y, color);
    }
    else
    {
        y++;
        SetPixel(x, y, color);
    }
}
```

Чтобы закрашивать отрезок вправо вниз при $|\Delta x| > |\Delta y|$, достаточно внести следующее изменение: при $D1 < -D2$ уменьшать переменную y на единицу.

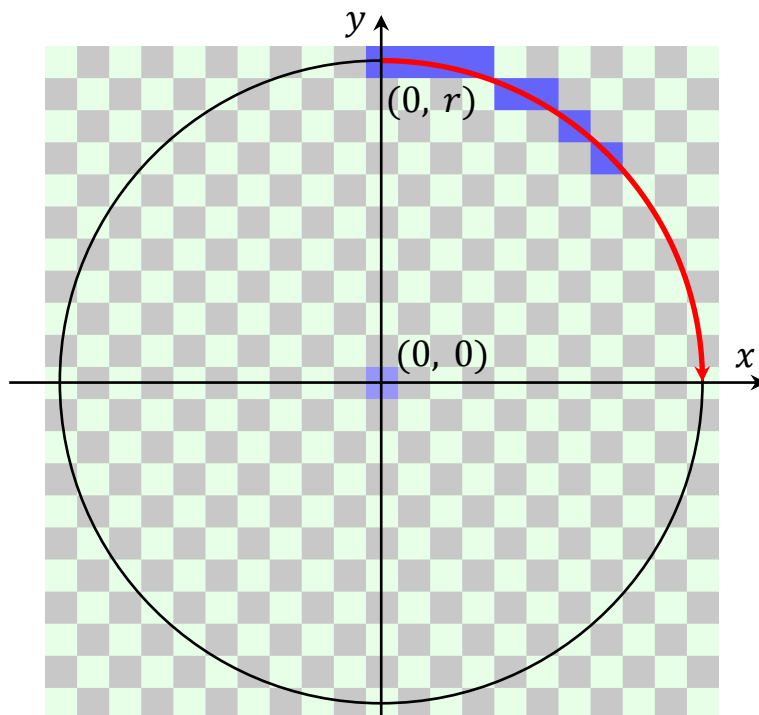
```
int x1 = 1, y1 = 4, x2 = 12, y2 = 1;
int dy = y2 - y1, dx = x2 - x1, y = y1;
for (int x = x1; x <= x2; x++)
{
    int D1 = (x - x1) * dy - (y - y1) * dx;
    int D2 = (x - x1) * dy - (y + 1 - y1) * dx;
    if (D1 < -D2)
    {
        y--;
        SetPixel(x, y, color);
    }
    else
    {
        SetPixel(x, y, color);
    }
}
```

Для написания полноценной функции, которая растеризует отрезок во всех направлениях, нужно ещё рассмотреть случай, когда $|\Delta y| > |\Delta x|$. Для этого достаточно поменять оси x и y местами. Обязательно требуется оптимизировать вычисление значения $D1 + D2$. Для этого следует учесть, что если x увеличивается на единицу, то сумма $D1 + D2$ увеличивается на $2dy$, а если y увеличивается на единицу, то значение $D1 + D2$ уменьшается на $2dx$. Чтобы значения x , y во внешнем цикле всегда увеличивались, нужно при необходимости поменять концы отрезка местами.

Для рисования окружности сначала опишем метод растеризации одного октанта (1/8 части) окружности радиусом r с центром в начале координат. Введём функцию, которая определяет, находится ли точка (x, y) внутри окружности:

$$D(x, y) = x^2 + y^2 - r^2.$$

Если $D(x, y) < 0$, то точка (x, y) располагается внутри окружности; если $D(x, y) > 0$, то точка (x, y) находится вне круга; если $D(x, y) = 0$, то точка (x, y) лежит на линии круга. Будем растеризовывать окружность с верхней точки вправо вниз, как показано на **рисунке**, пока $x < y$. Первым закрашивается пиксель с координатами $(0, r)$. На каждой итерации значение x будет обязательно увеличиваться на единицу. Поскольку линия направлена вниз, то нужно выбирать для закрашки из двух пикселей (x, y) и $(x, y - 1)$ тот, который ближе к линии окружности. Значение $D(x, y - 1)$ всегда отрицательное. Значит, если $D(x, y) > -D(x, y - 1)$, то закрашиваем пиксель $(x, y - 1)$, иначе закрашиваем пиксель (x, y) .



Таким образом можно нарисовать один октант окружности с центром в начале координат. Чтобы нарисовать окружность с произвольным центром (x_0, y_0) , достаточно сдвинуть все точки в направлении вектора (x_0, y_0) и

отобразить их на оставшиеся октанты. Соответствующий текст программы приведён в **листинге**.

```
void Circle(int x0, int y0, int radius, COLOR color)
{
    int x = 0, y = radius;
    while(x < y)
    {
        // Определяем, какая точка (пиксель): (x, y) или (x, y - 1) ближе к
линии окружности
        int D1 = x * x + y * y - radius * radius;
        int D2 = x * x + (y - 1) * (y - 1) - radius * radius;

        // Если ближе точка (x, y - 1), то смещаемся к ней
        if (D1 > -D2)
            y--;

        // Перенос и отражение вычисленных координат на все октанты
окружности
        SetPixel(x0 + x, y0 + y, color);
        SetPixel(x0 + x, y0 - y, color);
        SetPixel(x0 + y, y0 + x, color);
        SetPixel(x0 + y, y0 - x, color);
        SetPixel(x0 - x, y0 + y, color);
        SetPixel(x0 - x, y0 - y, color);
        SetPixel(x0 - y, y0 + x, color);
        SetPixel(x0 - y, y0 - x, color);
        x++;
    }
}
```

Данный фрагмент программы требуется оптимизировать. В начале цикла нужно инициализировать значение $D1+D2$ в отдельной переменной и приращивать его на каждой итерации, учитывая, что переменная y увеличивается на ноль или единицу, а переменная x всегда на $+1$.

После заполнения буфера кадра необходимо каким-либо образом вывести изображение на экран. Это можно делать различными способами в зависимости от того, какая используется библиотека для работы с графикой и в какой операционной системе работает программа. К примеру, можно использовать кроссплатформенную библиотеку `wxWidgets`, доступную в среде `Code::Blocks`.

В ОС Windows можно использовать WinAPI для масштабирования и вывода картинки на экран следующим образом (**листинг**). Подробно см. пример проекта `lab_1_basics.vcxproj`.

```
// Системная структура для хранения цвета пикселя
// Буфер кадра, который будет передаваться операционной системе, должен состоять из
массива этих структур
// Она не совпадает с порядком следования цветов в формате RGB
typedef struct tagRGBPIXEL
{
    unsigned char BLUE;        // Компонента синего цвета
    unsigned char GREEN;       // Компонента зелёного цвета
    unsigned char RED;          // Компонента красного цвета
    unsigned char ALPHA;       // Прозрачность
} RGBPIXEL;
```

```

    // Выделение памяти для второго буфера, который будет передаваться функции
    CreateBitmap для создания картинки
    RGBPIXEL* bitmap = (RGBPIXEL*) HeapAlloc(GetProcessHeap(), 0, width * height *
sizeof(RGBPIXEL));

    // Копирование массива пикселей в соответствии с системным форматом пикселя и
масштабирование картинки
    // W и H - ширина и высота изображения в буфере кадра
    // ratio - коэффициент масштабирования пикселей
    for (int y = 0; y < H * ratio; y++)
        for (int x = 0; x < W * ratio; x++)
        {
            RGBPIXEL* pixel = bitmap + y * width + x;
            COLOR color = frame.GetPixel(x / ratio, y / ratio);
            pixel->RED = color.RED;
            pixel->GREEN = color.GREEN;
            pixel->BLUE = color.BLUE;
            pixel->ALPHA = color.ALPHA;
        }

    // Получить дескриптор на новое растровое изображение
    HBITMAP hBitMap = CreateBitmap(width, height, 1, sizeof(RGBPIXEL) * 8, bitmap);

```


Пример выполнения работы

Задача. Реализовать вращение квадрата относительно центра экрана. Описать вокруг квадрата окружность.

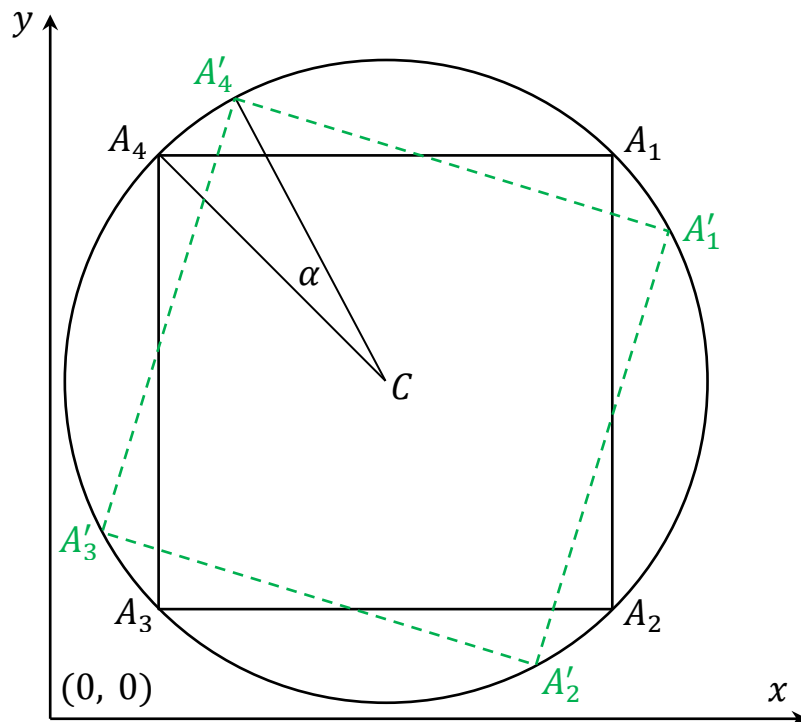
Решение. Пусть W – ширина экрана, H – высота экрана. Диаметр описанной вокруг квадрата окружности примем равным $7/8 \min(W, H)$. Тогда сторона квадрата равна $a = 7/8 \min(W, H) / \sqrt{2}$. Центр квадрата будет расположен по центру экрана, который имеет координаты $C(W/2, H/2)$. Обозначим α – угол поворота квадрата вокруг своего центра. Изначально вершины квадрата (рисунок) будут иметь координаты:

$$A_1(C_x + a/2, C_y + a/2),$$

$$A_2(C_x + a/2, C_y - a/2),$$

$$A_3(C_x - a/2, C_y - a/2),$$

$$A_4(C_x - a/2, C_y + a/2).$$



Поворот произвольной точки (x, y) вокруг начала координат на угол α соответствует следующему преобразованию:

$$x' = x \cos(\alpha) - y \sin(\alpha),$$

$$y' = x \sin(\alpha) + y \cos(\alpha).$$

Тогда, чтобы повернуть точку (x, y) вокруг произвольного центра C , нужно выполнить преобразование:

$$x' = (x - C_x) \cos(\alpha) - (y - C_y) \sin(\alpha) + C_x,$$

$$y' = (x - C_x) \sin(\alpha) + (y - C_y) \cos(\alpha) + C_y.$$

Подставим координаты точки $A_1(x_1, y_1)$ в предыдущее выражение:

$$x'_1 = a \cos(\alpha) - a \sin(\alpha) + C_x,$$

$$y'_1 = a \sin(\alpha) + a \cos(\alpha) + C_y.$$

Аналогичным образом получим координаты остальных вершин повёрнутого квадрата.

Для рисования квадрата и окружности с использованием приведённых формул ниже написан текст программы на языке C++.

```
int W = frame.width, H = frame.height;
// Масштаб рисунка возьмём меньше (7 / 8), чтобы он не касался границ экрана
float a = 7.0f / 8 * ((W < H) ? W - 1 : H - 1) / sqrt(2);
if (a < 1) return; // Если окно очень маленькое, то ничего не рисуем
float angle = global_angle; // Угол поворота
a = a / 2;

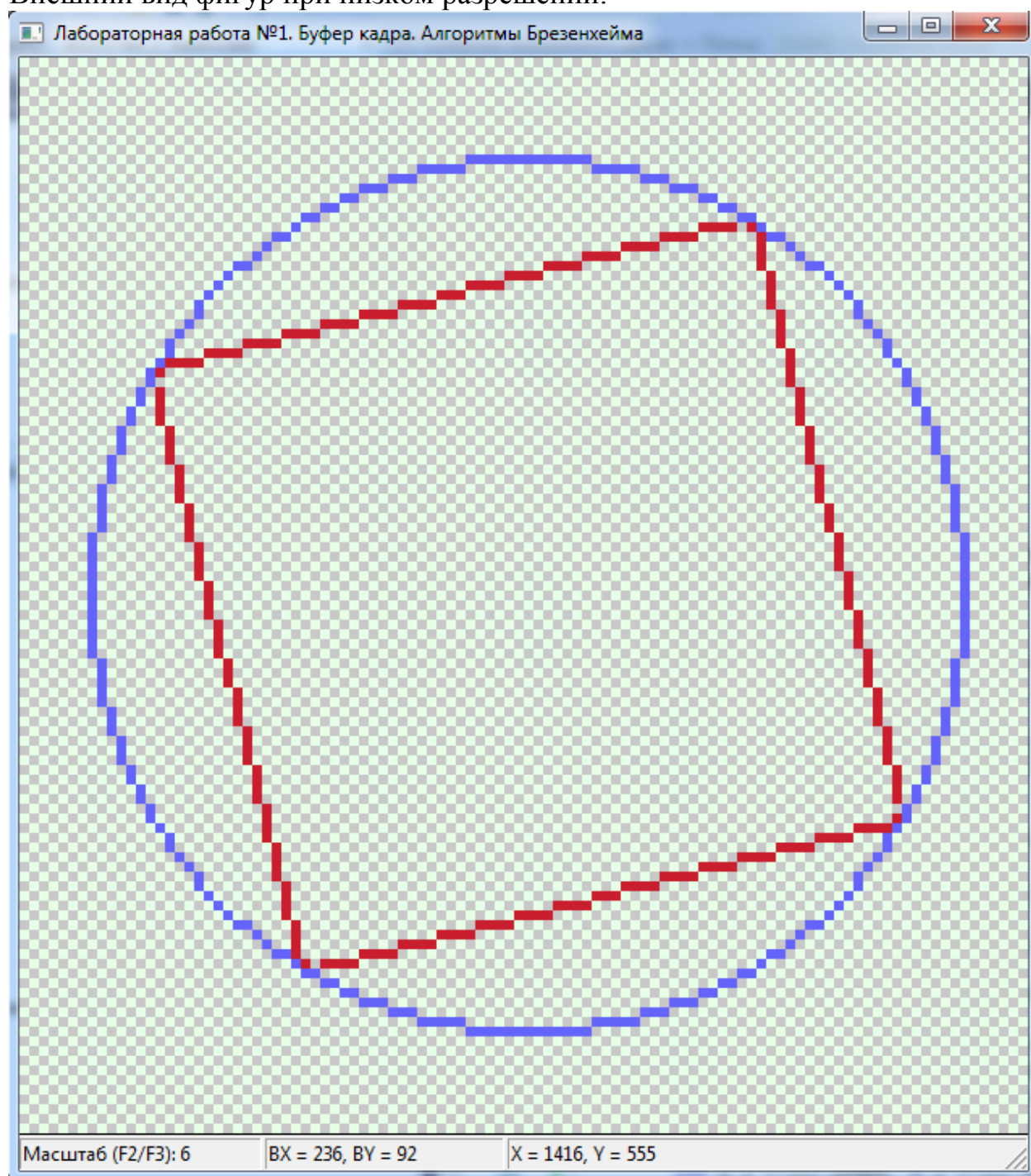
// Инициализируем исходные координаты центра и вершин квадрата
struct
{
    float x;
    float y;
} C = {W / 2, H / 2}, A[4] = { { C.x + a, C.y + a}, {C.x + a, C.y - a}, {C.x - a,
C.y - a}, {C.x - a, C.y + a} };

// Поворачиваем все вершины квадрата вокруг точки C на угол angle
for (int i = 0; i < 4; i++)
{
    float xi = A[i].x, yi = A[i].y;
    A[i].x = (xi - C.x) * cos(angle) - (yi - C.y) * sin(angle) + C.x;
    A[i].y = (xi - C.x) * sin(angle) + (yi - C.y) * cos(angle) + C.y;
}

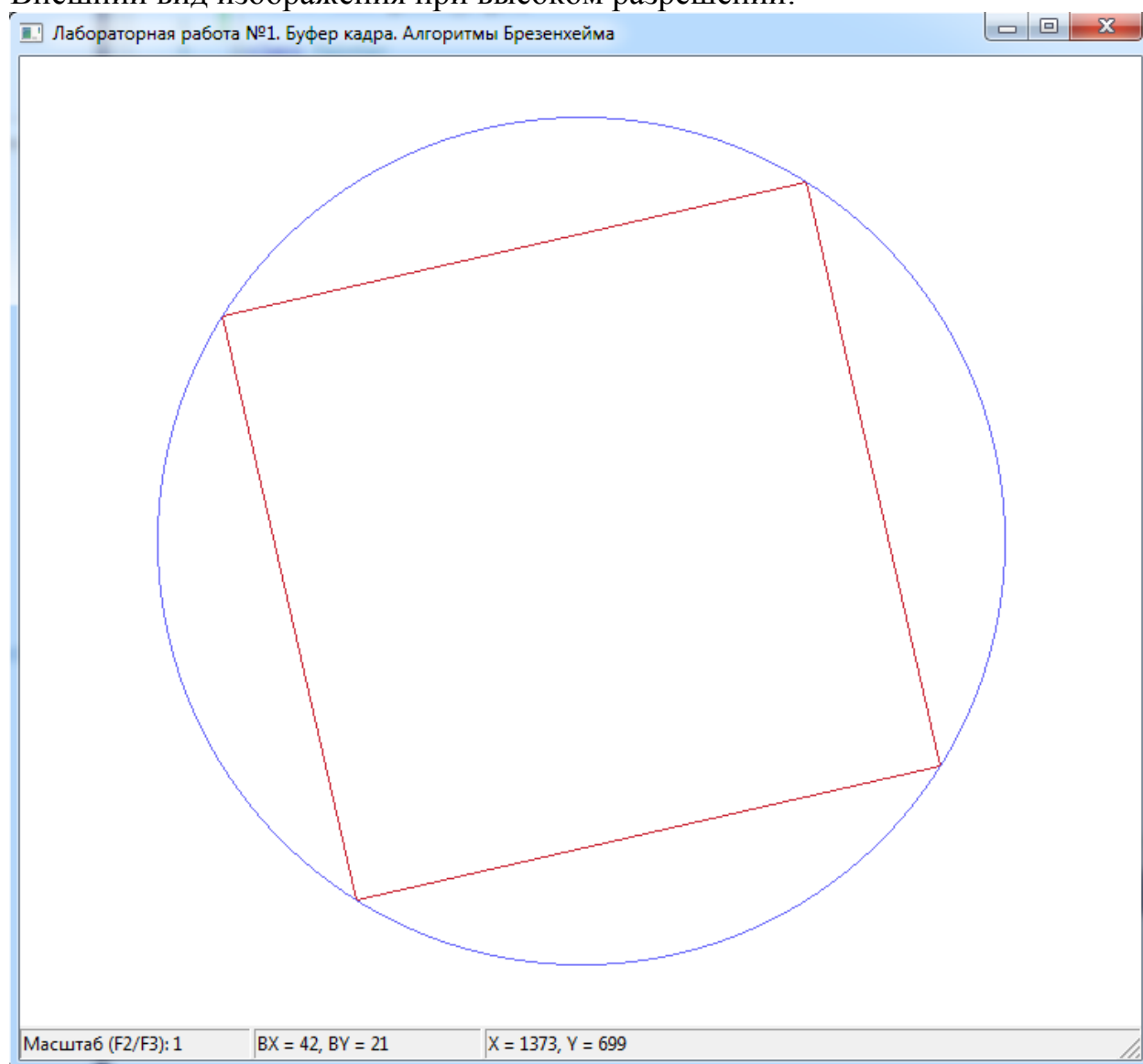
// Рисуем стороны квадрата
for (int i = 0; i < 4; i++)
{
    int i2 = (i + 1) % 4;
    frame.DrawLine( // Добавляем везде 0.5f, чтобы вещественные числа правильно
округлялись при преобразовании к целому типу
        int(A[i].x + 0.5f),
        int(A[i].y + 0.5f),
        int(A[i2].x + 0.5f),
        int(A[i2].y + 0.5f), COLOR(200, 30, 45));
}

// Рисуем описанную окружность
frame.Circle((int)C.x, (int)C.y, int(a*sqrt(2) + 0.5f), COLOR(100, 100, 250));
```

Внешний вид фигур при низком разрешении:

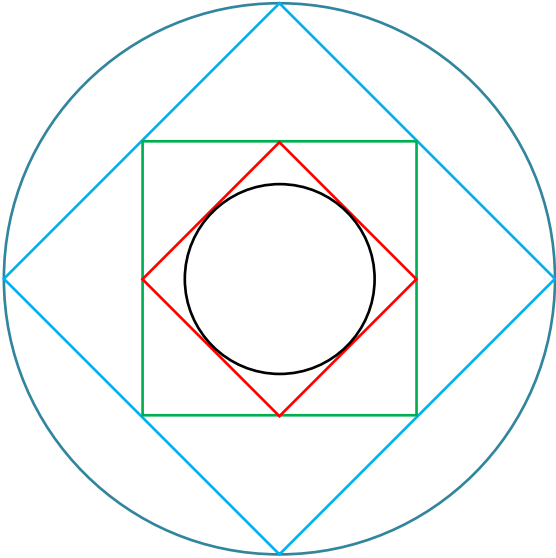
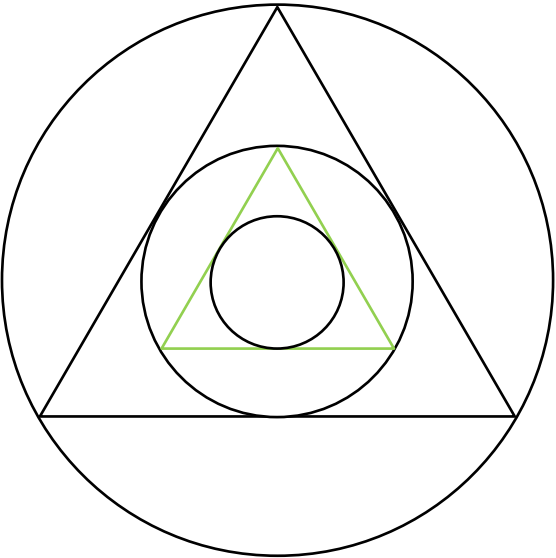


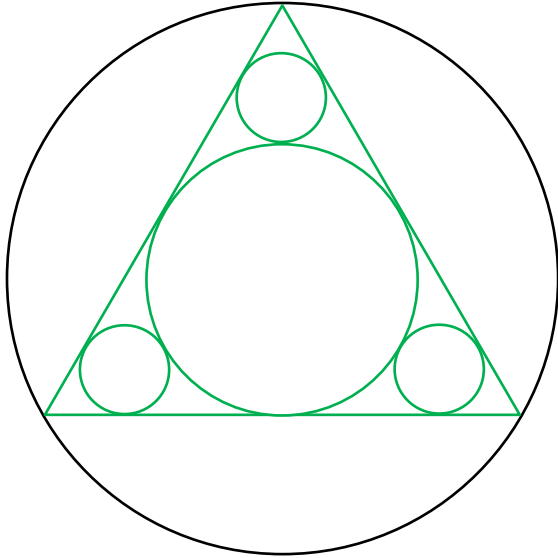
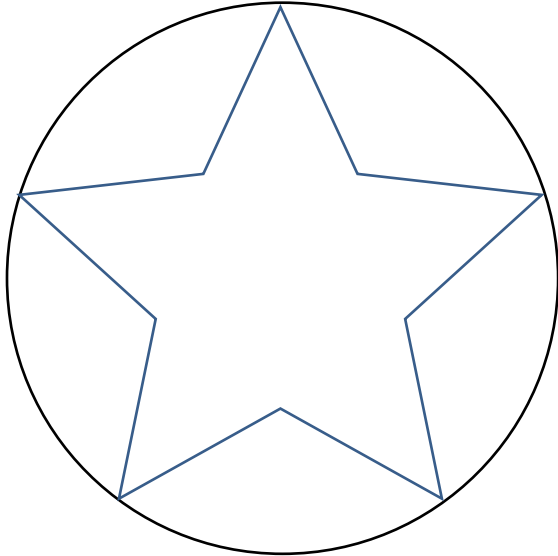
Внешний вид изображения при высоком разрешении:

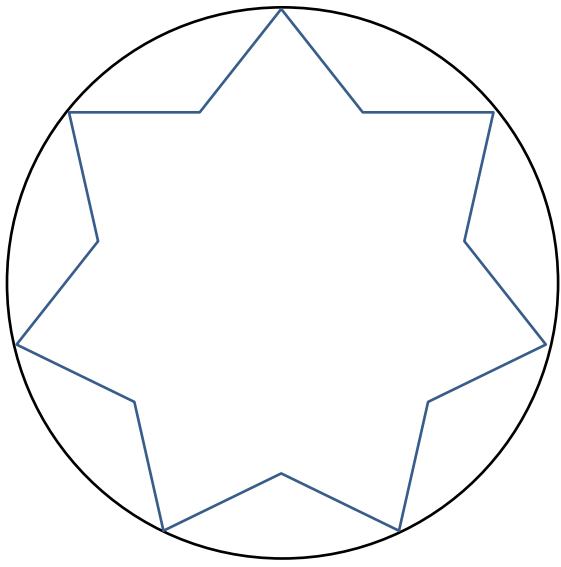
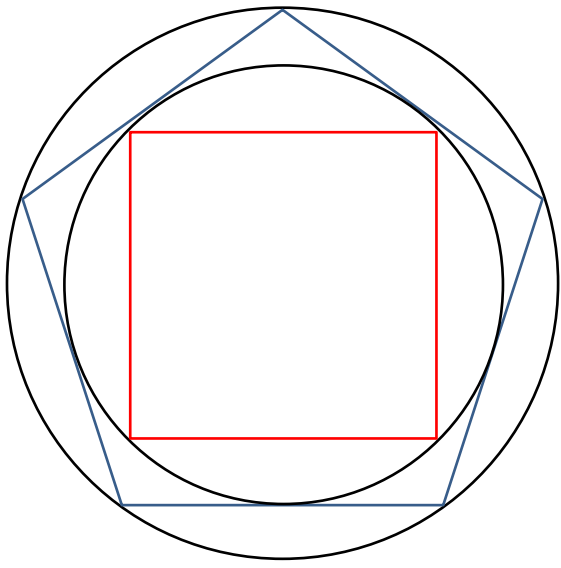


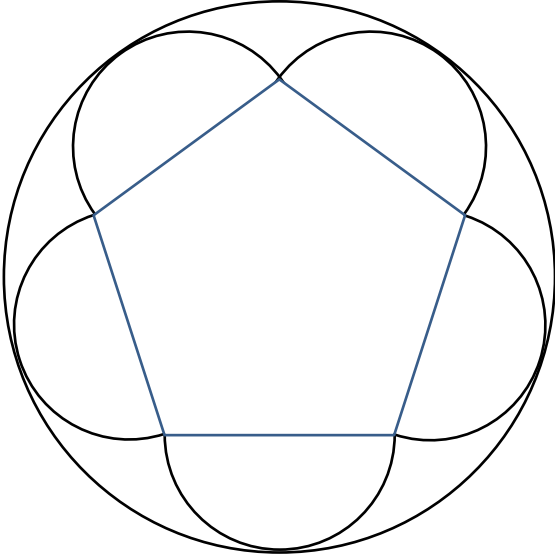
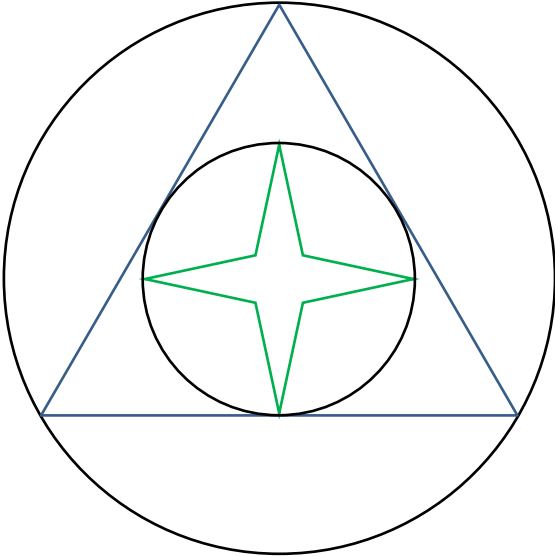
Варианты заданий

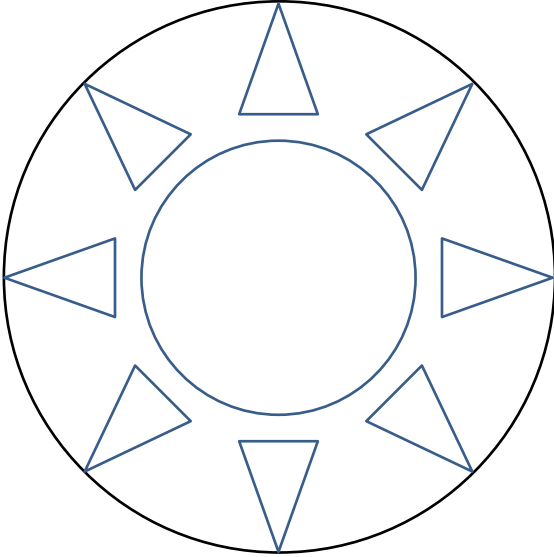
Таблица №1

Вариант	Рисунок	Исходные данные
1,10,19		Реализовать вращение красного квадрата против часовой стрелки.
2,11,20		Реализовать вращение внутреннего зелёного треугольника против часовой стрелки.

3,12,21		<p>Реализовать вращение внутреннего зелёного треугольника и трёх кругов по часовой стрелке.</p>
4,13,22		<p>Реализовать вращение звезды часовой стрелке.</p>

5,14,23		<p>Реализовать вращение 7-конечной звезды против часовой стрелки.</p>
6,15,24		<p>Реализовать вращение красного квадрата против часовой стрелки.</p>

7,16,25		<p>Каждая сторона пятиугольника делит круг на две равные части.</p> <p>Для отсечения точек во внутренней половине окружности использовать уравнение прямой, проходящей через соседние стороны пятиугольника</p>
8,17,26		<p>Реализовать вращение 4-конечной звезды против часовой стрелки.</p>

<p>9,18,27</p>		<p>Реализовать вращение треугольников вокруг центрального круга.</p>
----------------	--	--