

Лабораторная работа №3

Аффинные преобразования на плоскости

Цель работы: получение навыков выполнения аффинных преобразований на плоскости и создание графического приложения на языке C++ для создания простейшей анимации.

Порядок выполнения работы

1. Разработать алгоритм и составить программу для построения на экране изображения в соответствии с номером варианта. В качестве исходных данных взять указанные в таблице №1.

Требования к программе

1. Разработать модуль для выполнения аффинных преобразований на плоскости с помощью матриц. В модуле должны быть реализованы перегруженные операции действия с матрицами (умножение), с векторами и матрицами (умножение вектора-строки на матрицу), конструкторы различных матриц (переноса, масштабирования, переноса, отражения).
2. Раскрасить (залить) примитивы (круги, многоугольники и др.) по собственному усмотрению.

Содержание отчёта

1. Название темы.
2. Цель работы.
3. Постановка задачи.
4. Вывод необходимых геометрических формул для построения изображения и расчёта цвета. Также указать, какие матрицы используются и в какой последовательности они умножаются для реализации анимации.
5. Текст программы для рисования фигур.
6. Результат работы программы (снимки экрана).
7. Выводы о проделанной работе.

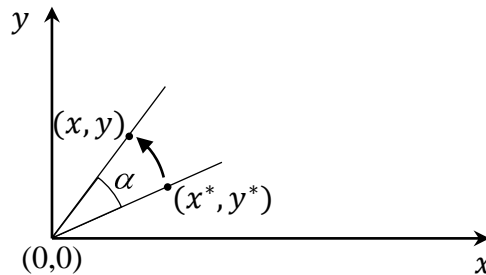
Теоретические сведения

Аффинными называют математические преобразования над геометрическим объектами, при которых у них сохраняются углы и отношения сторон.

Аффинные преобразования упрощают выполнение простейших геометрических операций, избавляя от необходимости выводить геометрические формулы для расчёта сложных движений объектов в пространстве.

В компьютерной графике рассматривают 4 аффинных преобразования:

1. **Поворот** точки (x, y) вокруг начала координат на угол α .



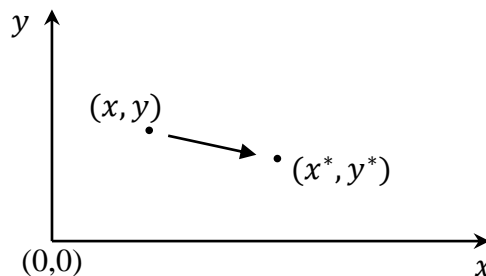
Новые координаты точки (x^*, y^*) , полученные в результате поворота, записываются в виде:

$$\begin{aligned}x^* &= x \cos \alpha - y \sin \alpha, \\y^* &= x \sin \alpha + y \cos \alpha;\end{aligned}$$

или, в матричной форме:

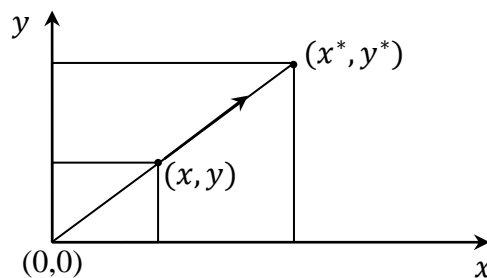
$$\begin{pmatrix} x^* \\ y^* \end{pmatrix} = \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

2. **Перенос** точки (x, y) вдоль вектора $(\Delta x, \Delta y)$.



$$\begin{aligned}x^* &= x + \Delta x, \\y^* &= y + \Delta y.\end{aligned}$$

3. **Масштабирование** относительно начала координат на величины $k_x, k_y > 0$.



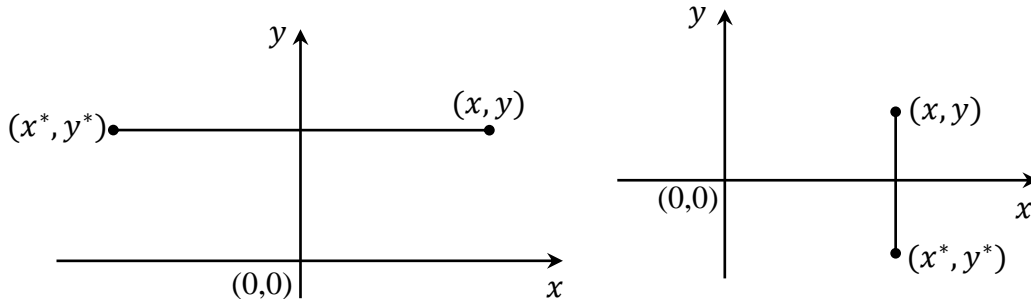
$$x^* = k_x x,$$

$$y^* = k_y y.$$

В матричной форме данное преобразование записывается в виде:

$$\begin{pmatrix} x^* \\ y^* \end{pmatrix} = \begin{pmatrix} k_x & 0 \\ 0 & k_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

4. **Отражение** точки (x, y) относительно оси абсцисс и/или оси ординат.



Данное преобразование равносильно изменению знака у одной (x или y) или двух (x и y) координат точки. Отражение относительно оси ординат:

$$\begin{aligned} x^* &= -x, \\ y^* &= y, \end{aligned} \quad \Leftrightarrow \quad \begin{pmatrix} x^* \\ y^* \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

Отражение относительно оси абсцисс:

$$\begin{aligned} x^* &= x, \\ y^* &= -y, \end{aligned} \quad \Leftrightarrow \quad \begin{pmatrix} x^* \\ y^* \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

Из приведённых выше преобразований можно заметить, что перенос пока не выражен в матричном виде. Это объясняется тем, что операцию переноса невозможно записать с использованием матрицы размера 2×2 . По этой причине переходят к матрицам размера 3×3 .

Однородными координатами точки (x, y) называют тройку чисел $x_1 : x_2 : k$, связанные с исходными декартовыми координатами следующими соотношениями:

$$x = \frac{x_1}{k}, y = \frac{x_2}{k}.$$

Использование однородных координат позволяет охватить матричными преобразованиями все 4 аффинных преобразования. Переход к однородным координатам также сокращает количество операций деления в геометрических преобразованиях, что выгодно снижает вычислительные затраты. Операция деления при этом заменяется операцией умножения, а последняя выполняется процессором гораздо быстрее.

Таким образом, операция переноса с использованием однородных координат запишется в виде:

$$\begin{pmatrix} x^* \\ y^* \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Перепишем остальные преобразования с использованием матриц размера 3×3 .

Поворот:

$$\begin{pmatrix} x^* \\ y^* \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Масштабирование:

$$\begin{pmatrix} x^* \\ y^* \\ 1 \end{pmatrix} = \begin{pmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Отражение относительно оси ординат:

$$\begin{pmatrix} x^* \\ y^* \\ 1 \end{pmatrix} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Отражение относительно оси абсцисс:

$$\begin{pmatrix} x^* \\ y^* \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Пример. Повернём точку $A(x, y)$ вокруг точки $O(x_0, y_0)$ на угол α . Для этого нужно выполнить 3 аффинных преобразования:

1. Перенести точки A и O вдоль вектора $(-x_0, -y_0)$ таким образом, чтобы точка O оказалась в начале координат:

$$\begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & -\Delta x \\ 0 & 1 & -\Delta y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Обозначим полученные координаты как $B(x_1, y_1)$.

2. Выполнить поворот точки B вокруг начала координат на угол α :

$$\begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix}.$$

3. Выполнить обратный перенос точки B вдоль вектора (x_0, y_0) :

$$\begin{pmatrix} x_3 \\ y_3 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix}.$$

В последнем выражении заменим однородные координаты $x_2: y_2: 1$ на предпоследнее выражение:

$$\begin{pmatrix} x_3 \\ y_3 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix}.$$

Аналогично заменим $x_1: y_1: 1$:

$$\begin{pmatrix} x_3 \\ y_3 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -\Delta x \\ 0 & 1 & -\Delta y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Произведение матриц можно посчитать заранее и обозначить как M , тогда:

$$\begin{pmatrix} x_3 \\ y_3 \\ 1 \end{pmatrix} = M \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Вид матрицы M зависит от угла α и координат точки $O(x_0, y_0)$. Изменяя эти параметры, можно поворачивать вершины фигур вокруг других точек на произвольный угол. К примеру, чтобы повернуть треугольник вокруг точки O , нужно три раза умножить матрицу M на однородные координаты трёх вершин треугольника. Заметим, что в приведённых преобразованиях матрицы умножаются в обратном порядке. Чтобы записывать матрицы аффинных преобразований в прямом порядке, нужно их транспонировать:

$$(x_3 \ y_3 \ 1) = (x \ y \ 1) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\Delta x & -\Delta y & 1 \end{pmatrix} \begin{pmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \Delta x & \Delta y & 1 \end{pmatrix}.$$

Оба способа записи эквиваленты, но второй способ, при котором матрицы транспонированы, более удобен при написании программ.

Системы координат

В задачах компьютерной графики часто приходится оперировать двумя системами координат: мировой системой координат и экранной системой координат.

Экранная система координат – система координат графического компонента (окна приложения). Данную систему координат мы уже рассматривали. Координатами точки в экранной системе координат является номер пикселя вдоль оси X и номер строки пикселей вдоль оси Y . Координаты X, Y могут меняться в следующих пределах:

$$0 \leq X_{\min} \leq X < X_{\max},$$

$$0 \leq Y_{\min} \leq Y < Y_{\max}.$$

Размеры экранной системы координат (X_{\max}, Y_{\max}) зависят от разрешения экрана и размеров окна. К примеру, при разрешении Full HD (1920×1080) $X_{\max} = 1920$, $Y_{\max} = 1080$. Начало экранной системы координат расположено в левом верхнем углу экрана (окна).

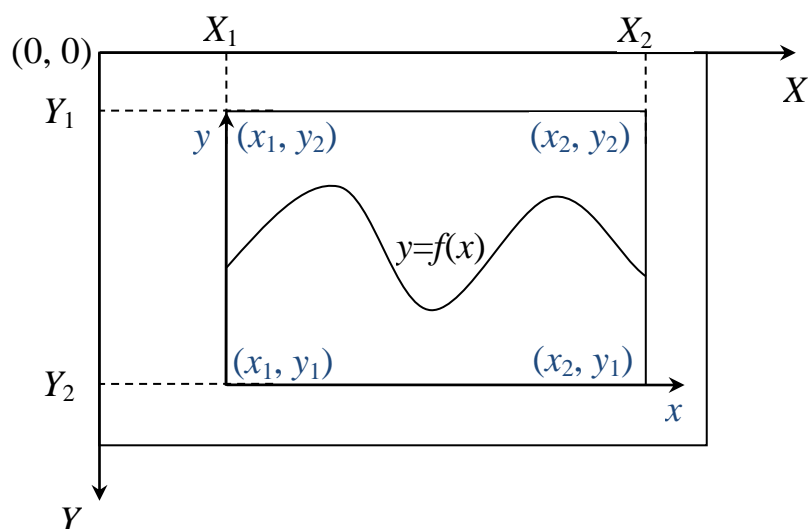


Рис. 1. Экранная (X, Y) и мировая (x, y) системы координат

Параметры экранной системы координат (максимальное число пикселей в строке X_{\max} и максимальное число строк пикселей Y_{\max}) зависят от размера окна.

Вторая система координат – так называемая *мировая* или математическая. Она представляет собой декартову систему (x, y) , определяемую программистом, и является независимой от конкретного графического устройства:

$$x_1 < x < x_2,$$

$$y_1 < y < y_2.$$

Параметры x_1, y_1, x_2, y_2 , которыми задаются диапазоны изменения x и y , определяют прямоугольную область в математическом двумерном пространстве. Эти параметры зависят только от конкретной задачи, а единицей измерения является реальная физическая величина (метр, секунда или др.).

Мировые координаты и координаты устройства связаны между собой простыми соотношениями:

$$X = (X_2 - X_1) \left(\frac{x - x_1}{x_2 - x_1} \right) + X_1,$$

$$Y = Y_2 - (Y_2 - Y_1) \left(\frac{y - y_1}{y_2 - y_1} \right).$$

Обозначим

$$p_x = \frac{X_2 - X_1}{x_2 - x_1}, \quad p_y = \frac{Y_2 - Y_1}{y_2 - y_1}.$$

Тогда преобразование координат запишется в виде:

$$\begin{aligned} X &= xp_x - x_1p_x + X_1, \\ Y &= -yp_y + Y_2 + y_1p_y. \end{aligned}$$

На основе данных выражений запишем *матрицу преобразования мировых координат в экранные*:

$$\begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} = \begin{pmatrix} p_x & 0 & X_1 - x_1p_x \\ 0 & -p_y & Y_2 + y_1p_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Экранная система координат неудобна тем, что её ось ординат непривычно направлена вниз, а координаты представляют собой целочисленные номера пикселей. Данная матрица существенно облегчает вычисления, позволяя отвлечься от экранной системы координат и оперировать только с мировыми координатами.

Для работы с матрицами нужно создать отдельный модуль, в котором будут реализованы операции конструирования матриц, умножения матриц, умножения вектора на матрицу и другие необходимые. Матрицами будем оперировать в транспонированном виде. В следующем классе `Matrix` реализовано умножение матриц, конструирование матрицы поворота и матрицы для линейного преобразования координат. Остальные методы необходимо реализовать самостоятельно.

```
class Matrix
{
    float M[3][3];

public:

    // По умолчанию матрица инициализируется как единичная
    Matrix() : M {
        { 1, 0, 0 },
        { 0, 1, 0 },
        { 0, 0, 1 } }
    {}

    // Конструктор, который инициализирует матрицу M поэлементно значениями аргументов
    Matrix( float A00, float A01, float A02,
            float A10, float A11, float A12,
            float A20, float A21, float A22) : M {
        { A00, A01, A02 },
        { A10, A11, A12 },
        { A20, A21, A22 } }
    {}

    // Умножение матриц
    Matrix operator * (const Matrix& A) const
    {
        Matrix R;
        char i, j;
        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                R.M[i][j] = M[i][0] * A.M[0][j] + M[i][1] * A.M[1][j] + M[i][2] *
A.M[2][j];
        return R;
    }

    // Матрица поворота вокруг начала координат на угол angle против часовой стрелки
```

```

// angle - угол поворота (в градусах) вокруг точки (0, 0)
static Matrix Rotation(float angle)
{
    const float PI = 3.14159274;
    angle = angle / 180 * PI;
    float cosA = cos(angle);
    float sinA = sin(angle);
    return Matrix(cosA, sinA, 0,
                  -sinA, cosA, 0,
                  0, 0, 1);
}

// Матрица для преобразования мировых координат в экранные координаты области (порта)
вывода
// (X1, Y1) - экранные координаты левого верхнего угла области (порта) вывода
// (X2, Y2) - экранные координаты правого нижнего угла области (порта) вывода
// (x1, y1) - мировые координаты левого нижнего угла области вывода
// (x2, y2) - мировые координаты правого верхнего угла области вывода
static Matrix WorldToScreen(float X1, float Y1, float X2, float Y2, float x1, float
y1, float x2, float y2)
{
    float px = (X2 - X1)/(x2 - x1), py = (Y2 - Y1)/(y2 - y1);
    return Matrix(
        px,          0,          0,
        0,          -py,         0,
        X1 - x1 * px, Y2 + y1 * py, 1);
}

friend class Vector;
};

class Vector
{
public:

    float x, y;

    Vector() : x(0), y(0)
    {
    }

    Vector(float _x, float _y) : x(_x), y(_y)
    {
    }

    Vector operator * (const Matrix &A)
    {
        Vector E;
        E.x = x*A.M[0][0] + y*A.M[1][0] + A.M[2][0];
        E.y = x*A.M[0][1] + y*A.M[1][1] + A.M[2][1];
        float h = x*A.M[0][2] + y*A.M[1][2] + A.M[2][2];
        E.x /= h;
        E.y /= h;
        return E;
    }
};

```

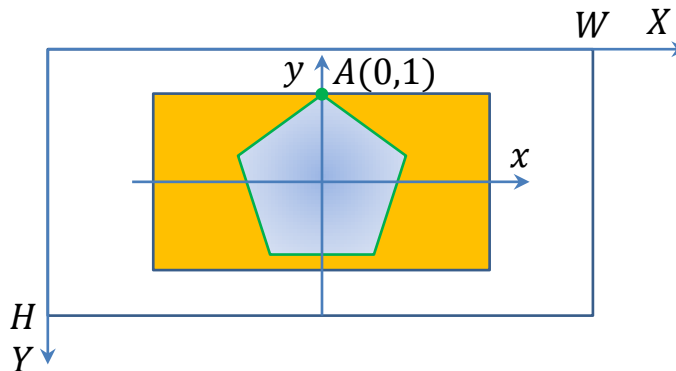
Задача 1. Нарисовать вращающийся правильный n -угольник.

Решение. Пусть центр многоугольника расположен в начале координат, а первая вершина имеет координаты $A(0,1)$. Значит, чтобы получить координаты всех вершин, нужно умножать однородные координаты точки A на матрицы поворота:

$$P_i = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \begin{pmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \alpha = \frac{2\pi i}{n} + \beta.$$

Параметр β задаёт дополнительный угол поворота для создания анимации вращения. Предполагается, что β постоянно изменяется по таймеру в другом потоке.

Отступим от края экрана и введём новую мировую систему координат (x, y) внутри окна следующим образом:



Построим матрицу преобразования мировых координат (x, y) в экранные координаты (X, Y) . Выделим квадрат со стороной $a = 7/8 \cdot \min(W, H)$ и сопоставим точке $\left(\frac{W}{2} + \frac{a}{2}, \frac{H}{2} + \frac{a}{2}\right)$ мировые координаты $(1, -1)$, а противоположной точке $\left(\frac{W}{2} - \frac{a}{2}, \frac{H}{2} - \frac{a}{2}\right)$ — координаты $(-1, 1)$. Тогда для построения матрицы WS для преобразования мировых координат в экранные используем следующие параметры:

$$\begin{aligned} X_1 &= \frac{W}{2} - \frac{a}{2}, & Y_1 &= \frac{H}{2} - \frac{a}{2}, \\ X_2 &= \frac{W}{2} + \frac{a}{2}, & Y_2 &= \frac{H}{2} + \frac{a}{2}, \\ x_1 &= -1, & y_1 &= -1, & x_2 &= 1, & y_2 &= 1. \end{aligned}$$

Для получения окончательных экранных координат достаточно умножить все координаты P_i на матрицу WS .

На языке C++ приведённые вычисления реализует следующая функция:

```
void Draw(Frame& frame)
{
    const int N = 7; // Количество углов многоугольника

    float W = frame.width, H = frame.height;

    // Размер рисунка возьмём меньше (7 / 8), чтобы он не касался границ экрана
    float a = 7.0 / 8.0 * ((W < H) ? W : H);
```



```

float beta = global_angle; // Угол поворота фигуры

// Инициализация исходных координат
Vector A(0, 1); // Координаты первой вершины многоугольника в мировой системе
координат

Vector O(0, 0); // Координаты центра многоугольника в мировой системе координат

// Матрица для преобразования мировых координат в экранные
// Область вывода определяем как квадрат со стороной a в центре экрана
Matrix WS = Matrix::WorldToScreen(W/2 - a/2, H/2 - a/2, W/2 + a/2, H/2 + a/2, -1,
-1, 1, 1);

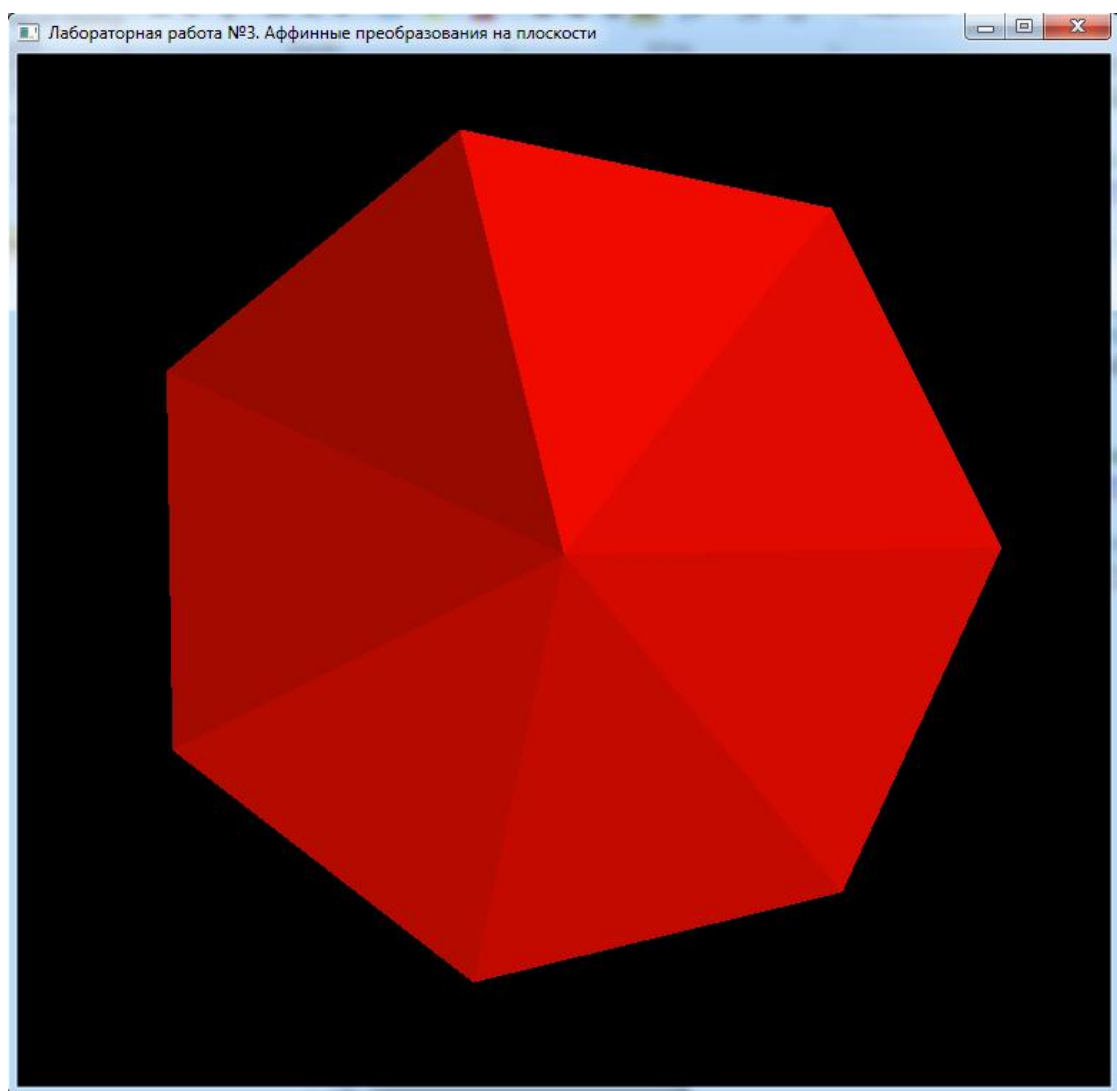
Vector V[N], C = O * WS; // Координаты вершин многоугольника и его центра в
экранной системе координат

// Вычисление координат вершин многоугольника
for (int i = 0; i < N; i++)
{
    // Добавляем к каждому углу дополнительный угол beta для создания анимации
    вращения

    // Умножаем матрицу поворота на матрицу WS
    Matrix M = Matrix::Rotation(360.0f / N * i + beta) * WS;
    V[i] = A * M; // Координаты i-ой вершины в экранной системе координат
}

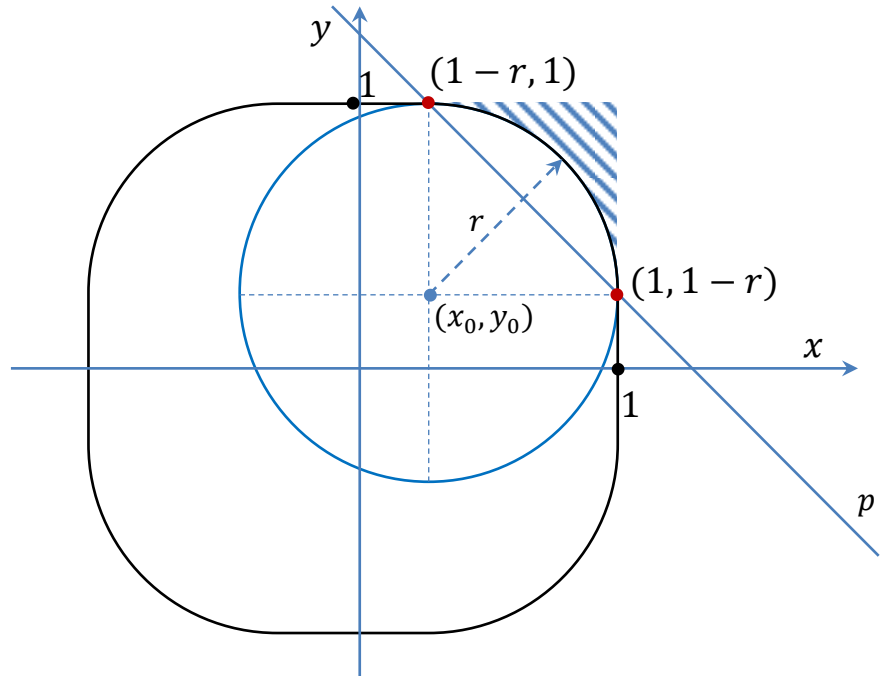
// Рисование многоугольника
for (int i = 0; i < N; i++)
{
    Vector T1 = V[i], T2 = V[(i + 1) % N];
    frame.Triangle(T1.x, T1.y, C.x, C.y, T2.x, T2.y, COLOR((15*i) % 100 + 150,
10, 0));
}
}

```



Задача 2. Нарисовать квадрат со скруглёнными краями.

Введём ещё одну дополнительную систему координат, связанную с самой фигурой, в которой будем её рисовать. Она не будет меняться при движении, вращении и масштабировании фигуры. Квадрат имеет сторону, равную 2 и расположен в начале координат. Радиус сопряжения равен r .



Чтобы скруглить края квадрата, нужно при его растеризации для каждого пикселя (x, y) , который расположен в углу квадрата, проверять, выходит ли он за пределы круга с радиусом r . Прямая p имеет уравнение

$$x + y = 2 - r.$$

Значит точка находится в заштрихованной области, если выполняется условие: $x + y > 2 - r$ и $(x - x_0)^2 + (y - y_0)^2 > r^2$. При выполнении этого условия мы не будем рисовать пиксель.

Таким образом мы закруглили правый верхний угол. Для того, чтобы закруглить остальные углы нужно выполнить отражение точек в первую четверть. Если какая-то из координат x или y отрицательная, то нужно просто поменять у неё знак.

Мы разобрались, как рисовать квадрат с закруглёнными углами в начале координат. Но как же его рисовать в другом месте? Как его можно повернуть или увеличить? В этом нам спешит на помощь барицентрическая интерполяция! Никто нам не запрещает вместо цвета интерполировать координаты вершин квадрата, т.е. те координаты, по которым он действительно будет нарисован.

Пусть вершины квадрата имеют экранные координаты A_0, A_1, A_2, A_3 . Рассмотрим треугольник $\Delta A_0 A_1 A_2$. Вычисляя барицентрические координаты $(\lambda_0, \lambda_1, \lambda_2)$ некоторого пикселя квадрата (x, y) при растеризации, интерполируем координаты (wx, wy) в системе координат квадрата. Пусть

точке A_0 соответствует вершина $W_0(-1, -1)$ в системе координат квадрата, $A_1 - W_1(1, -1)$, $A_2 - W_2(1, 1)$.

Тогда пикселю (x, y) соответствуют координаты точки (wx, wy) на квадрате:

$$\begin{aligned} wx &= \lambda_0 W_{0x} + \lambda_1 W_{1x} + \lambda_2 W_{2x}, \\ wy &= \lambda_0 W_{0y} + \lambda_1 W_{1y} + \lambda_2 W_{2y}. \end{aligned}$$

Для второй половины квадрата $\Delta A_2 A_3 A_0$ нужно рассмотреть тройку точек $W_0(1, 1)$, $W_1(-1, 1)$, $W_2(-1, -1)$.

Теперь вершины квадрата можно как угодно перемещать, поворачивать и масштабировать. Единственное условие аффинных преобразований – не должны меняться углы и соотношения сторон фигуры.

Следующая программа анимирует вращающийся квадрат со скруглёнными углами.

```
void Draw(Frame& frame)
{
    float W = frame.width, H = frame.height;

    // Размер рисунка возьмём меньше (7 / 8), чтобы он не касался границ экрана
    float a = 7.0 / 8.0 * ((W < H) ? W : H);

    float beta = global_angle; // Угол поворота фигуры

    // Матрица для преобразования мировых координат в экранные
    // Область вывода определяем как квадрат со стороной a в центре экрана
    Matrix WS = Matrix::WorldToScreen(W/2 - a/2, H/2 - a/2, W/2 + a/2, H/2 + a/2, -1,
-1, 1, 1);

    // Шейдерный класс для рисования треугольника, который будет половиной квадрата
    class TriangleShader
    {
    public:
        float x0, y0, x1, y1, x2, y2; // Координаты трёх вершин квадрата
        // Координаты трёх вершин квадрата в
        // экранной системе координат
        float wx0, wy0, wx1, wy1, wx2, wy2; // Координаты трёх вершин квадрата в
        // мировой системе координат
        float S; // Площадь треугольника
        COLOR C0, C1; // Цвет ободка и внутренней части

        TriangleShader(float _x0, float _y0, float _x1, float _y1, float _x2, float
_y2,
        float _wx0, float _wy0, float _wx1, float _wy1,
        float _wx2, float _wy2, COLOR _C0, COLOR _C1) :
            x0(_x0), y0(_y0), x1(_x1), y1(_y1), x2(_x2), y2(_y2),
            wx0(_wx0), wy0(_wy0), wx1(_wx1), wy1(_wy1), wx2(_wx2), wy2(_wy2),
            C0(_C0), C1(_C1),
            S((_y1 - _y2) * (_x0 - _x2) + (_x2 - _x1) * (_y0 - _y2))
        {
        }

        // Основная функция main рассчитывает цвет точки с координатами (x, y) для
        // квадрата, который расположен в начале координат, и
        // закругляет ему края
        // Центр квадрата имеет координаты (0, 0). Сторона квадрата равна 2
        COLOR main(float x, float y)
        {
            if (x < 0) x = -x; // Отражаем точку в первую четверть, чтобы было
            легче считать
```

```

        if (y < 0) y = -y;

        float r = 0.5; // Радиус сопряжения
        // Координаты центра сопряжения в первой четверти
        float x0 = 1 - r, y0 = 1 - r;
        // Если точка выходит за пределы области, возвращаем полностью
прозрачный пиксель
        if (x + y > 2 - r && (x - x0)*(x - x0) + (y - y0)*(y - y0) > r*r)
        {
            return COLOR(0, 0, 0, 0);
        }

        float d;
        // Сделаем так, чтобы цвет зависел от расстояния до границы фигуры
        // Если точка находится вблизи угла квадрата, то рассчитываем
расстояние от точки (x, y) до круга и делим его на радиус
        if (x > 1 - r && y > 1 - r)
        {
            d = (r - sqrt((x - x0) * (x - x0) + (y - y0) * (y - y0))) / r;
        }
        else // В противном случае находим ближайшую сторону квадрата
        {
            if (x > 1 - r || y > 1 - r)
                d = min(1 - y, 1 - x) / r;
            else d = 1;
        }
        return COLOR( C0.RED*d + C1.RED*(1 - d),
                     C0.GREEN * d + C1.GREEN * (1 - d),
                     C0.BLUE * d + C1.BLUE * (1 - d), 255 - (1
- d)*100);
    }

    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 = ((y0 - y1) * (x - x1) + (x1 - x0) * (y - y1)) / S;
        //float h2 = 1 - h0 - h1;

        // Если точка (x, y) находится вне треугольника
        if (h0 < -1E-6 || h1 < -1E-6 || h2 < -1E-6)
        {
            return COLOR(0, 0, 0); // Ошибка алгоритма растеризации, если
рисуется чёрный пиксель
        }

        // Интерполируем мировые координаты вершин
        float wx = h0 * wx0 + h1 * wx1 + h2 * wx2;
        float wy = h0 * wy0 + h1 * wy1 + h2 * wy2;

        return main(wx, wy);
    }
};

Vector A[] = { Vector(-0.75f, -0.75f), Vector(0.75f, -0.75f), Vector(0.75f,
0.75f), Vector(-0.75f, 0.75f) };

// Поворот квадрата
for (int i = 0; i < _countof(A); i++)
{
    A[i] = A[i] * (Matrix::Rotation(beta * 5) * WS);
}

```

```

    TriangleShader shader1(A[0].x, A[0].y, A[1].x, A[1].y, A[2].x, A[2].y, -1, -1, 1,
-1, 1, 1, COLOR(50, 255, 50), COLOR(255, 255, 255));
    frame.Triangle(A[0].x, A[0].y, A[1].x, A[1].y, A[2].x, A[2].y, shader1);

    TriangleShader shader2(A[2].x, A[2].y, A[3].x, A[3].y, A[0].x, A[0].y, 1, 1, -1,
1, -1, -1, COLOR(50, 255, 50), COLOR(255, 255, 255));
    frame.Triangle(A[2].x, A[2].y, A[3].x, A[3].y, A[0].x, A[0].y, shader2);

}

```

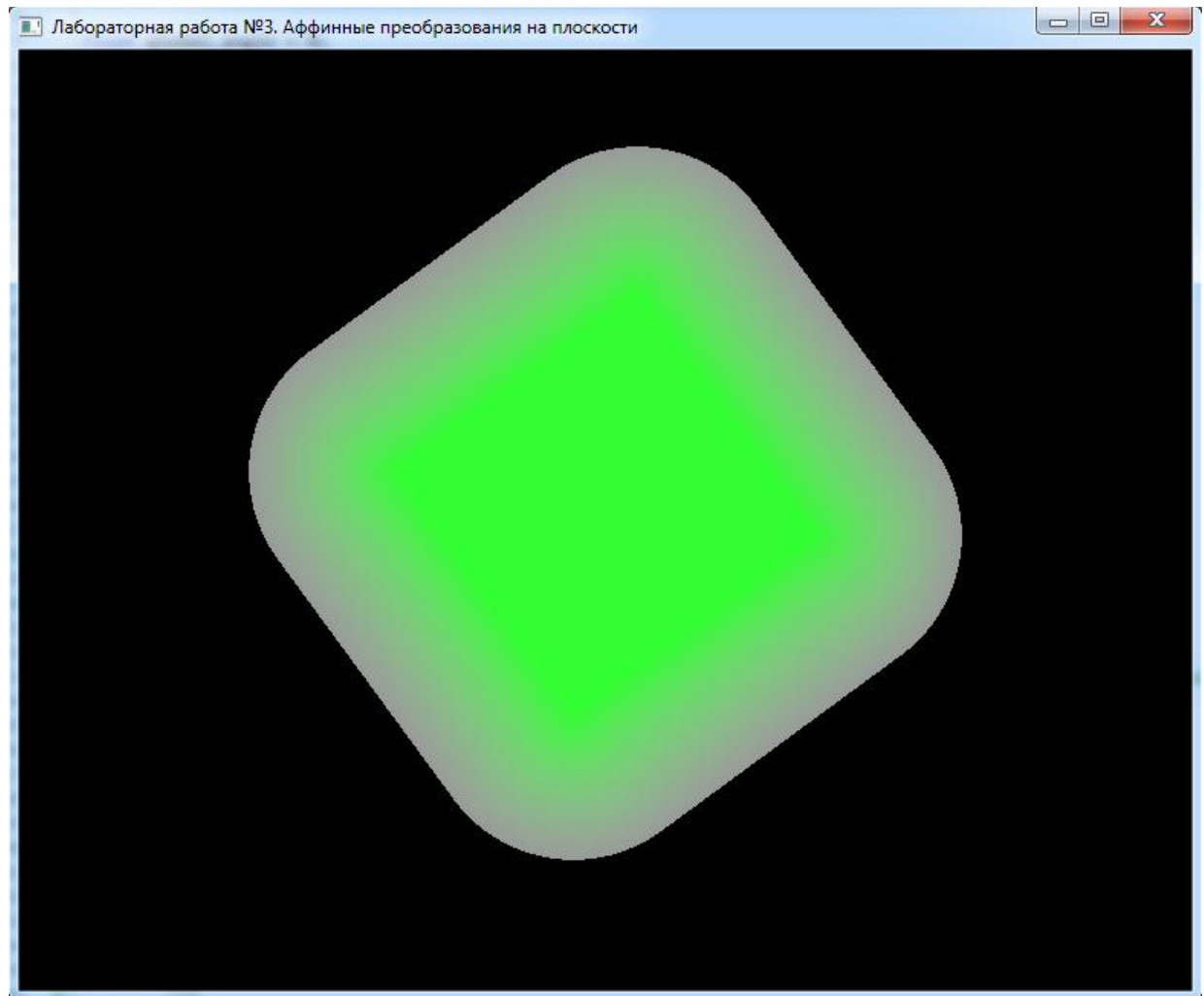
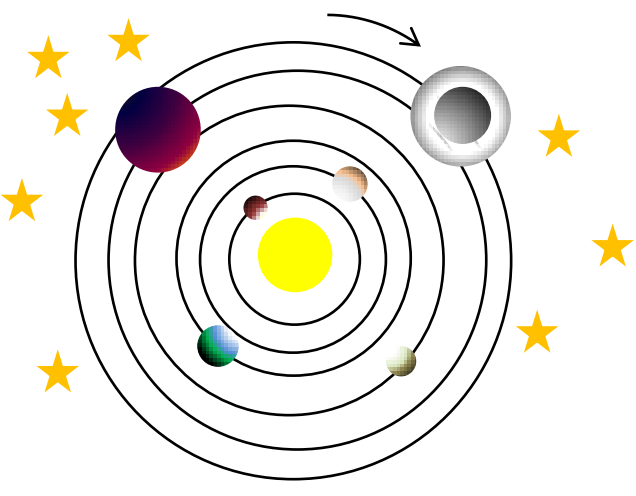
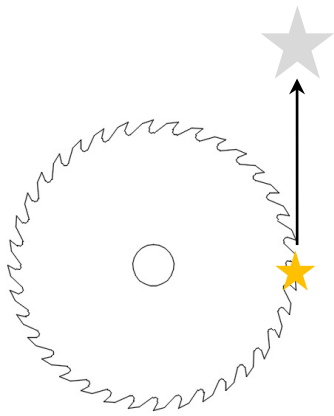
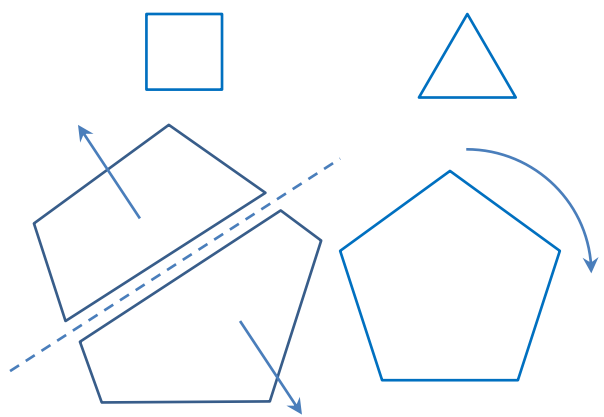
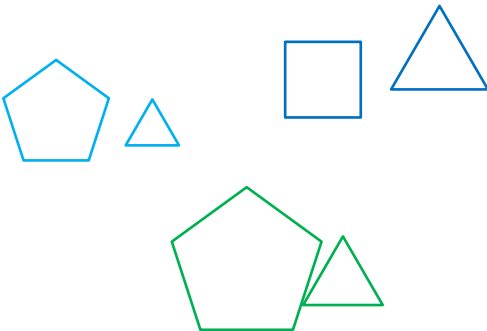

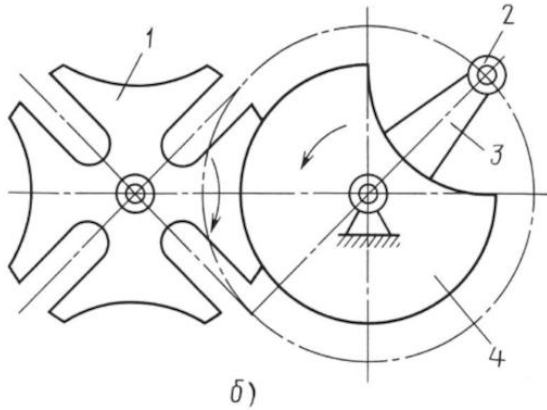
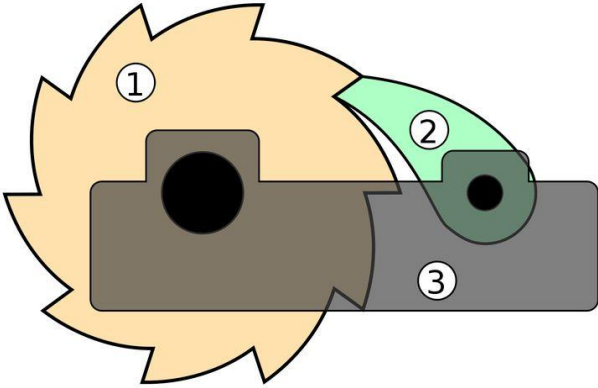
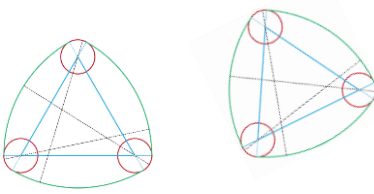
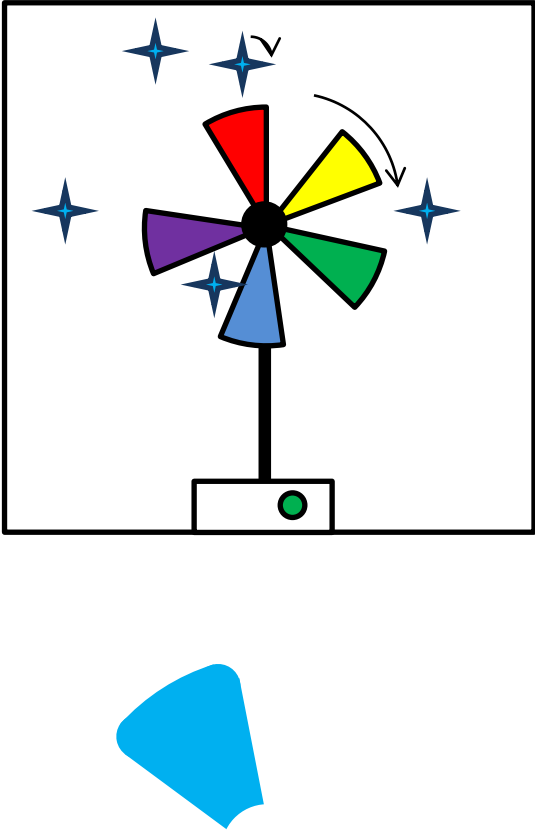
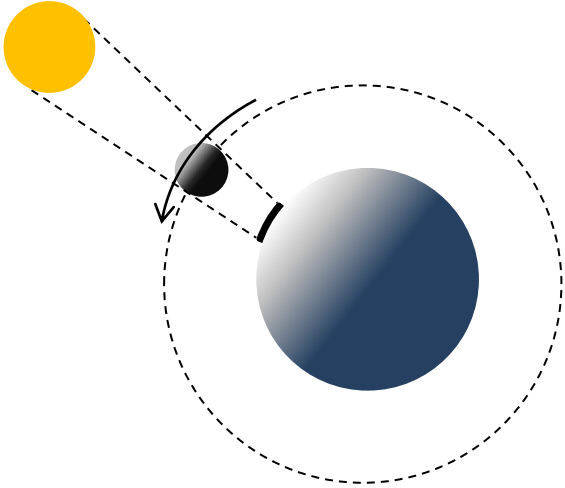


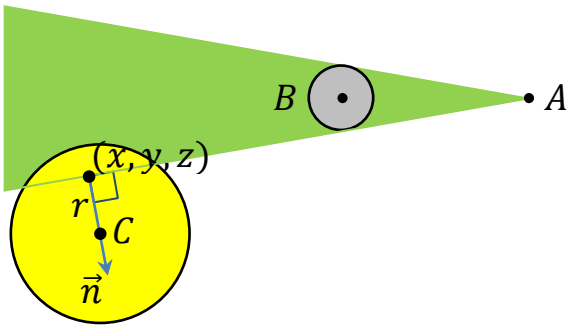
Таблица №1

Вариант	Рисунок	Исходные данные
1,11,21		Изобразить вращение планет солнечной системы вокруг солнца и анимацию: вращение и периодическое изменение размера (увеличение/ уменьшение) звезд, расположенных вокруг планет в случайных местах. Солнце освещает планеты только с ближайшей к нему стороны. Реализовать вращение планет с разной скоростью вокруг солнца. У Сатурна изобразить кольца. Периодически пролетают кометы, которые освещаются также с ближайшей к Солнцу стороны. Функция-шейдер рассчитывает цвет пикселей внутри кругов в зависимости от взаимного расположения планеты и солнца.
2,12,22		Изобразить вращение пилы, от которой отскакивают искры. Искры изобразить в виде вращающихся звёздочек, которые отскакивают преимущественно по центробежной траектории с небольшим разбросом ($\pm 5^\circ$). По мере удаления искр, они становятся серыми, увеличиваются в размере и теряют угловую скорость. Искры исчезают за пределами экрана. Они появляются у острого края пилы в случайных местах и имеют случайный красно-жёлтый оттенок. Чем быстрее вращается пила, тем больше появляется искр. Разработать функцию-шейдер, которая рисует один зубец пилы и применить его для рисования всех зубцов.
3,13,23		На экране в случайных местах непрерывно появляются равносторонние многоугольники. Постепенно скорость вращения каждого многоугольника относительно своего центра увеличивается и сам многоугольник тоже плавно увеличивается. При достижении определённой угловой скорости многоугольник случайным образом «трескается» пополам относительно центра. Два осколка разлетаются с равной скоростью перпендикулярно рассекающей прямой (как показано на рисунке). Осколки исчезают за пределами экрана. Цвет каждой точки на фигуре должен зависеть от расстояния до края фигуры.

Вариант	Рисунок	Исходные данные
4,14,24		<p>На экране в случайных местах непрерывно появляются очень маленькие (практически невидимые) вращающиеся равносторонние многоугольники. У каждой фигуры определено две фазы существования. В первой фазе фигура постепенно увеличивается, пока не коснется границы или вершины какой-либо другой фигуры. Во второй фазе, когда фигура коснулась другой фигуры (или границы), обе фигуры постепенно уменьшаются, пока не исчезнут. Чтобы фигуры не появлялись внутри друг друга, запретить при генерации появление новой фигуры внутри описанных вокруг каждой фигуры окружностей. Цвет каждой точки на фигуре должен зависеть от расстояния до края фигуры.</p>
5,15,25		<p>Изобразить на плоскости работу зубчатого механизма с тремя шестерёнками. Создать анимацию – шестерёнки вращаясь, передают движение друг другу. Шестерёнки изобразить с вырезами, как на рисунке (с использованием шейдера).</p>

Вариант	Рисунок	Исходные данные
6,16,26		<p>Изобразить на плоскости работу мальтийского механизма с 4 пазми. Программа должна нарисовать анимацию сверху, как показано на рисунке. Ведущее звено (4) вращает пальцем (2) мальтийский крест (1). Пример работы мальтийского механизма представлен в записи:</p> <p>https://vk.com/video8052051_456239052</p> <p>Шейдер должен рисовать $\frac{1}{4}$ или $\frac{1}{8}$ часть мальтийского механизма.</p>
7,17,27		<p>Изобразить на плоскости работу храпового механизма. Программа должна нарисовать анимацию сверху, как показано на рисунке. Храповик (1) вращается против часовой стрелки, при этом собачка (2) прижимается к нему и не даёт вращаться в обратную сторону.</p> <p>Пример работы храпового механизма представлен в записи:</p> <p>https://rutube.ru/video/0e592378963589a50022c3c4feeca521</p> <p>Шейдер должен рисовать один зубец.</p>
8,18,28		<p>Нарисовать несколько треугольников Рело с закруглёнными краями, хаотически движущихся в пределах экрана. Треугольники должны вращаться, плавно масштабироваться. Рассчитать цвет таким образом, чтобы он менялся плавно от края треугольника к центру.</p> <p>Описание треугольника Рело:</p> <p>https://ru.ruwiki.ru/wiki/%D0%A2%D1%80%D0%B5%D1%83%D0%B3%D0%BE%D0%BB%D1%8C%D0%BD%D0%B8%D0%BA_%D0%A0%D1%91%D0%BB%D0%BE</p>

Вариант	Рисунок	Исходные данные
9,19,29		<p>Реализовать вращение лопастей вентилятора и следующую анимацию: при нажатии на кнопку, вентилятор начинает работать и кнопка горит зеленым. В выключенном состоянии кнопка горит красным. В центре вентилятора появляются 4-конечные вращающиеся звезды, которые движутся по спирали в направлении движения лопастей с уменьшением первоначальной скорости. Звезды появляются в центре вентилятора в произвольный момент времени. Звезды вращаются вокруг своих центров, каждая с разной скоростью. Появление звезд случайно. При приближении к границе рисунка звезды растворяются. Лопасты вентилятора должны иметь закруглённые углы, как показано на рисунке слева.</p>
10,20,30		<p>Реализовать вращение луны вокруг планеты. Радиус орбиты луны периодически плавно изменяется. Солнце освещает только те половины луны и Земли, которые ближе к нему. Пусть при каждом обороте луна создаёт солнечное затмение. При движении луны тень, создаваемая ей на планете, отображается в виде пятна. Для расчёта тени от луны написать отдельный шейдер для растеризации круга (Земли). Он должен работать в трёхмерном пространстве следующим образом. Для каждой точки (пикселя) $P(x, y)$ круга Земли,</p>

Вариант	Рисунок	Исходные данные
	 <p>Чтобы определить, затмевает ли луна солнце в точке A, нужно построить каноническое уравнение конуса (зелёного), образованного луной, с вершиной в точке A. Зная уравнение конуса $F(x, y, z) = 0$, можно выразить расстояние r от точки $C(x_0, y_0, z_0)$ до поверхности конуса, а также определить, находится ли точка C внутри конуса (по знаку $F(x_0, y_0, z_0)$). Для этого нужно опустить нормаль \vec{n} из точки C к поверхности конуса в точку (x, y, z):</p> $n_x = \frac{\partial F}{\partial x}(x, y, z), \quad n_y = \frac{\partial F}{\partial y}(x, y, z),$ $n_z = \frac{\partial F}{\partial z}(x, y, z).$ <p>Далее нужно решить систему уравнений с 4 неизвестными x, y, z, λ:</p> $x + \lambda n_x = x_0, \quad y + \lambda n_y = y_0, \quad z + \lambda n_z = z_0,$ $F(x, y, z) = 0.$ <p>Окончательно определим расстояние от центра Солнца до поверхности конуса:</p> $r^2 = (x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = \lambda^2 \cdot \vec{n} ^2$ <p>или $r = \lambda \cdot \vec{n}$.</p> <p>Если r больше радиуса Солнца и точка C находится внутри конуса, то точка A не освещена. Чтобы было проще выполнять данные геометрические преобразования, лучше с использованием аффинных преобразований перенести вершину конуса в начало координат, а его ось повернуть вдоль оси абсцисс.</p>	<p>сначала определить её координаты в трёхмерном пространстве $A(x, y, z)$ на сфере. Для этого достаточно найти координату z из уравнения сферы $(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r$, при известных x, y, x_0, y_0, z_0, r. Затем в этом же трёхмерном пространстве определить, затмевает ли точку $A(x, y, z)$ луна. Если из этой точки видимый диск луны полностью закрывает диск Солнца, то точка A не освещается.</p>