МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

Лабораторная работа №3

по дисциплине: Компьютерная графика
тема: «Аффинные преобразования на плоскости»

Выполнил: ст. группы ПВ-233
Мовчан Антон Юрьевич

Проверили:
ст. пр. Осипов Олег Васильевич

Белгород 2025 г.

Лабораторная работа №3

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

Порядок выполнения работы

Порядок выполнения работы

1. Изучить приложенные на языке C++ программы для визуализации изображений.
2. Разработать алгоритм и составить программу для построения на экране изображения в соответствии с номером варианта.

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. Внутри функции растеризации треугольника реализовать прозрачность (как описано во второй лаб. Работе).
3. Нарисовать две различные русские буквы из своего имени, отчества. На оценку «отлично» рисовать буквы, которые содержат закруглѐнные линии, например, «Ю», «Э», «Я», «Р». Такие буквы, как «Н», «Т», «Г», «Е» , «Х» , «А» считаются простыми, так как состоят только из многоугольников.
4. Применить к буквам все виды матричных аффинных преобразований: поворот, перенос, масштабирование. На основе этих операций сделать любую анимацию на собственное усмотрение. Например, буквы, поворачиваясь, циклически уменьшаются, а затем, по достижении определѐнного размера, снова увеличиваются. Или, например, буквы разлетаются по спирали от центра экрана, уменьшаясь в размере.
5. Реализовать приведѐнные ниже эффекты текстурирования и «затопления» буквы. Поэкспериментировать и придумать свой эффект (например, блики, освещение, намокание, различные виды волн, светлячки, капли).

Main.cpp

```cpp
#include <SDL2/SDL.h>
#include <cmath>
#include <vector>
#include <algorithm>
#include <string>
#include <memory>
#include <array>

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

const int INITIAL_WIDTH = 800;
const int INITIAL_HEIGHT = 600;
const int MIN_RESOLUTION = 10;
```

```
const int MAX_RESOLUTION = 2000;

// Глобальные переменные для анимации и прозрачности
float global_angle = 0.0f;
float global_alpha = 0.8f;
const float rotation_speed = 1.0f;

// Переменные для эффекта лупы
bool magnifier_active = false;
float magnifier_x = 0.0f;
float magnifier_y = 0.0f;
const float magnifier_radius = 80.0f;
const float magnifier_zoom = 2.0f;

struct COLOR {
    Uint8 r, g, b, a;

    COLOR() : r(0), g(0), b(0), a(255) {}
    COLOR(Uint8 red, Uint8 green, Uint8 blue, Uint8 alpha = 255)
        : r(red), g(green), b(blue), a(alpha) {}
};

// Единый цвет для всех букв
const COLOR LETTER_COLOR = COLOR(200, 200, 33, 255);

class Matrix {
    float M[3][3];
public:
    Matrix() : M{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}} {}

    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}} {}

    Matrix operator * (const Matrix& A) const {
        Matrix R;
        for (int i = 0; i < 3; i++) {
            for (int 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;
    }

    static Matrix Rotation(float angle) {
        float rad = angle * M_PI / 180.0f;
        float cosA = cos(rad);
        float sinA = sin(rad);
        return Matrix(cosA, sinA, 0,
                      -sinA, cosA, 0,
                      0, 0, 1);
    }

    static Matrix Translation(float tx, float ty) {
        return Matrix(1, 0, 0,
                      0, 1, 0,
                      tx, ty, 1);
```

```
    }

    static Matrix Scaling(float sx, float sy) {
        return Matrix(sx, 0, 0,
                      0, sy, 0,
                      0, 0, 1);
    }

    static Matrix Reflection(bool reflectX, bool reflectY) {
        return Matrix(reflectX ? -1 : 1, 0, 0,
                      0, reflectY ? -1 : 1, 0,
                      0, 0, 1);
    }

    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);
        float py = (Y2 - Y1) / (y2 - y1);
        return Matrix(px, 0, 0,
                      0, -py, 0,
                      X1 - x1 * px, Y2 + y1 * py, 1);
    }

    const float* operator[](int index) const { return M[index]; }
    float* operator[](int index) { return M[index]; }
};

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

// Базовый класс для шейдеров
class BaseShader {
public:
    virtual ~BaseShader() = default;
    virtual COLOR getColor(float x, float y, float h0, float h1, float h2) = 0;
};

// Шейдер для эффекта "пульсации"
class PulseShader : public BaseShader {
    float time;
    COLOR baseColor;
    float alpha;
```

```
public:
    PulseShader(COLOR color, float t, float a) : baseColor(color), time(t), alpha(a) {}

    COLOR getColor(float x, float y, float h0, float h1, float h2) override {
        float pulse = sin(time * 3) * 0.3f + 0.7f;
        return COLOR(
            static_cast<Uint8>(baseColor.r * pulse),
            static_cast<Uint8>(baseColor.g * pulse),
            static_cast<Uint8>(baseColor.b * pulse),
            static_cast<Uint8>(baseColor.a * alpha)
        );
    }
};

// 1. Волны
class WaveShader : public BaseShader {
    float time;
    COLOR baseColor;
public:
    WaveShader(COLOR color, float t) : baseColor(color), time(t) {}

    COLOR getColor(float x, float y, float h0, float h1, float h2) override {
        float wave = 0.5f + 0.5f * sin(x * 0.2f + time * 3.0f);
        return COLOR(
            static_cast<Uint8>(baseColor.r * wave),
            static_cast<Uint8>(baseColor.g * wave),
            static_cast<Uint8>(baseColor.b * wave),
            255
        );
    }
};

// 2. Блики
class ShinyShader : public BaseShader {
    float time;
    COLOR baseColor;
public:
    ShinyShader(COLOR color, float t) : baseColor(color), time(t) {}

    COLOR getColor(float x, float y, float h0, float h1, float h2) override {
        float shine = 0.5f + 0.5f * cos((y + time * 50.0f) * 0.1f);
        return COLOR(
            std::min(255, static_cast<int>(baseColor.r + 255 * shine * 0.3f)),
            std::min(255, static_cast<int>(baseColor.g + 255 * shine * 0.3f)),
            std::min(255, static_cast<int>(baseColor.b + 255 * shine * 0.3f)),
            255
        );
    }
};

// 3. Капли
class DropShader : public BaseShader {
    float time;
    COLOR baseColor;
public:
    DropShader(COLOR color, float t) : baseColor(color), time(t) {}

    COLOR getColor(float x, float y, float h0, float h1, float h2) override {
        float drop = fmod(y + time * 30.0f + sin(x*0.5f)*20.0f, 30.0f);
        float alpha = (drop < 2.0f) ? 0.0f : 1.0f; // маленькая прозрачная капля
        return COLOR(
```

```
            baseColor.r,
            baseColor.g,
            baseColor.b,
            static_cast<Uint8>(255 * alpha)
        );
    }
};


class Frame {
    int width, height;
    std::vector<COLOR> pixels;

public:
    Frame(int w, int h) : width(w), height(h), pixels(w * h) {}

    void Resize(int w, int h) {
        width = w;
        height = h;
        pixels.resize(w * h);
    }

    void SetPixel(int x, int y, COLOR color) {
        if (x >= 0 && x < width && y >= 0 && y < height) {
            if (color.a == 255) {
                pixels[y * width + x] = color;
            } else {
                COLOR dest = pixels[y * width + x];
                float alpha = color.a / 255.0f;
                float inv_alpha = 1.0f - alpha;
                pixels[y * width + x] = {
                    static_cast<Uint8>(color.r * alpha + dest.r * inv_alpha),
                    static_cast<Uint8>(color.g * alpha + dest.g * inv_alpha),
                    static_cast<Uint8>(color.b * alpha + dest.b * inv_alpha),
                    255
                };
            }
        }
    }

    COLOR GetPixel(int x, int y) const {
        if (x >= 0 && x < width && y >= 0 && y < height) {
            return pixels[y * width + x];
        }
        return COLOR(0, 0, 0, 0);
    }

    void Clear(COLOR color) {
        std::fill(pixels.begin(), pixels.end(), color);
    }

    int GetWidth() const { return width; }
    int GetHeight() const { return height; }

    // Метод для применения эффекта лупы к кадру
    void ApplyMagnifier(float mx, float my, float radius, float zoom) {
        // Создаем временную копию кадра
        std::vector<COLOR> tempPixels = pixels;

        // Применяем эффект увеличения в круговой области
```

```
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            // Вычисляем расстояние от центра лупы
            float dx = x - mx;
            float dy = y - my;
            float distance = sqrt(dx * dx + dy * dy);

            // Если пиксель внутри круга лупы
            if (distance < radius) {
                // Вычисляем координаты в исходном изображении с учетом увеличения
                float srcX = mx + (x - mx) / zoom;
                float srcY = my + (y - my) / zoom;

                // Билинейная интерполяция для сглаживания
                int x1 = static_cast<int>(srcX);
                int y1 = static_cast<int>(srcY);
                int x2 = x1 + 1;
                int y2 = y1 + 1;

                float fx = srcX - x1;
                float fy = srcY - y1;

                // Получаем цвета соседних пикселей
                COLOR c11 = GetPixelFromCopy(tempPixels, x1, y1);
                COLOR c12 = GetPixelFromCopy(tempPixels, x1, y2);
                COLOR c21 = GetPixelFromCopy(tempPixels, x2, y1);
                COLOR c22 = GetPixelFromCopy(tempPixels, x2, y2);

                // Интерполируем цвет
                COLOR interpolated = BilinearInterpolate(c11, c12, c21, c22, fx, fy);

                // Устанавливаем увеличенный пиксель
                SetPixel(x, y, interpolated);
            }
        }
    }
}

class BarycentricInterpolator {
    float x0, y0, x1, y1, x2, y2, S;

public:
    BarycentricInterpolator(float _x0, float _y0, float _x1, float _y1,
                            float _x2, float _y2)
        : x0(_x0), y0(_y0), x1(_x1), y1(_y1), x2(_x2), y2(_y2),
          S((_y1 - _y2) * (_x0 - _x2) + (_x2 - _x1) * (_y0 - _y2)) {}

    void getWeights(float x, float y, float& h0, float& h1, float& h2) {
        h0 = ((y1 - y2) * (x - x2) + (x2 - x1) * (y - y2)) / S;
        h1 = ((y2 - y0) * (x - x2) + (x0 - x2) * (y - y2)) / S;
        h2 = 1.0f - h0 - h1;
    }
};

// Исправлено: метод DrawTriangle теперь публичный
template <class ShaderClass>
void DrawTriangle(float x0, float y0, float x1, float y1, float x2, float y2, ShaderClass&& shader) {
    BarycentricInterpolator interpolator(x0, y0, x1, y1, x2, y2);

    float minX = std::min({x0, x1, x2});
```

```
        float maxX = std::max({x0, x1, x2});
        float minY = std::min({y0, y1, y2});
        float maxY = std::max({y0, y1, y2});

        int startX = std::max(0, static_cast<int>(minX));
        int endX = std::min(width - 1, static_cast<int>(maxX));
        int startY = std::max(0, static_cast<int>(minY));
        int endY = std::min(height - 1, static_cast<int>(maxY));

        for (int y = startY; y <= endY; y++) {
            for (int x = startX; x <= endX; x++) {
                float h0, h1, h2;
                interpolator.getWeights(x + 0.5f, y + 0.5f, h0, h1, h2);

                if (h0 >= -1e-6f && h1 >= -1e-6f && h2 >= -1e-6f) {
                    COLOR color = shader.getColor(x + 0.5f, y + 0.5f, h0, h1, h2);
                    SetPixel(x, y, color);
                }
            }
        }
    }

private:
    COLOR GetPixelFromCopy(const std::vector<COLOR>& copy, int x, int y) const {
        if (x >= 0 && x < width && y >= 0 && y < height) {
            return copy[y * width + x];
        }
        return COLOR(0, 0, 0, 0);
    }

    COLOR BilinearInterpolate(COLOR c11, COLOR c12, COLOR c21, COLOR c22, float fx, float fy) {
        COLOR result;

        // Интерполяция по красному каналу
        float r1 = c11.r * (1 - fx) + c21.r * fx;
        float r2 = c12.r * (1 - fx) + c22.r * fx;
        result.r = static_cast<Uint8>(r1 * (1 - fy) + r2 * fy);

        // Интерполяция по зеленому каналу
        float g1 = c11.g * (1 - fx) + c21.g * fx;
        float g2 = c12.g * (1 - fx) + c22.g * fx;
        result.g = static_cast<Uint8>(g1 * (1 - fy) + r2 * fy);

        // Интерполяция по синему каналу
        float b1 = c11.b * (1 - fx) + c21.b * fx;
        float b2 = c12.b * (1 - fx) + c22.b * fx;
        result.b = static_cast<Uint8>(b1 * (1 - fy) + b2 * fy);

        // Интерполяция по альфа-каналу
        float a1 = c11.a * (1 - fx) + c21.a * fx;
        float a2 = c12.a * (1 - fx) + c22.a * fx;
        result.a = static_cast<Uint8>(a1 * (1 - fy) + a2 * fy);

        return result;
    }

    void DrawLine(int x1, int y1, int x2, int y2, COLOR color) {
        int dx = abs(x2 - x1);
        int dy = abs(y2 - y1);
        int sx = (x1 < x2) ? 1 : -1;
```

```
        int sy = (y1 < y2) ? 1 : -1;
        int err = dx - dy;

        while (true) {
            SetPixel(x1, y1, color);
            if (x1 == x2 && y1 == y2) break;

            int e2 = 2 * err;
            if (e2 > -dy) {
                err -= dy;
                x1 += sx;
            }
            if (e2 < dx) {
                err += dx;
                y1 += sy;
            }
        }
    }

    void FillCircle(int x0, int y0, int radius, COLOR color) {
        for (int y = -radius; y <= radius; y++) {
            for (int x = -radius; x <= radius; x++) {
                if (x*x + y*y <= radius*radius) {
                    SetPixel(x0 + x, y0 + y, color);
                }
            }
        }
    }

    void FillCircleRadial(int x0, int y0, int radius, COLOR centerColor, COLOR edgeColor) {
        for (int y = -radius; y <= radius; y++) {
            for (int x = -radius; x <= radius; x++) {
                if (x*x + y*y <= radius*radius) {
                    float distance = sqrt(x*x + y*y) / radius;
                    COLOR interpolated = InterpolateColor(centerColor, edgeColor, distance);
                    SetPixel(x0 + x, y0 + y, interpolated);
                }
            }
        }
    }

    COLOR InterpolateColor(COLOR c1, COLOR c2, float t) {
        return COLOR(
            static_cast<Uint8>(c1.r * (1-t) + c2.r * t),
            static_cast<Uint8>(c1.g * (1-t) + c2.g * t),
            static_cast<Uint8>(c1.b * (1-t) + c2.b * t),
            static_cast<Uint8>(c1.a * (1-t) + c2.a * t)
        );
    }
};

// Простой шейдер для отверстия
class HoleShader : public BaseShader {
    COLOR holeColor;
public:
    HoleShader(COLOR color) : holeColor(color) {}

    COLOR getColor(float x, float y, float h0, float h1, float h2) override {
        return holeColor;
    }
};
```

```
// Класс для буквы "Н"
class LetterN {
private:
    std::vector<Vector> vertices;
    std::vector<std::tuple<int, int, int>> triangles;
public:
    LetterN() {
        // Левая вертикальная ножка
        vertices = {
            Vector(0.0f, 0.0f), Vector(0.0f, 2.0f),
            Vector(0.3f, 2.0f), Vector(0.3f, 0.0f),

            // Правая вертикальная ножка
            Vector(0.7f, 0.0f), Vector(0.7f, 2.0f),
            Vector(1.0f, 2.0f), Vector(1.0f, 0.0f),

            // Горизонтальная перекладина
            Vector(0.3f, 1.0f), Vector(0.7f, 1.0f),
            Vector(0.7f, 1.2f), Vector(0.3f, 1.2f)
        };

        // Треугольники для левой ножки
        triangles.push_back({0,1,2});
        triangles.push_back({0,2,3});

        // Треугольники для правой ножки
        triangles.push_back({4,5,6});
        triangles.push_back({4,6,7});

        // Треугольники для горизонтальной перекладины
        triangles.push_back({8,9,10});
        triangles.push_back({8,10,11});
    }

    void Draw(Frame& frame, const Matrix& transform, float time, float alpha) {
        std::vector<Vector> transformedVertices;
        for (auto& v : vertices) transformedVertices.push_back(v * transform);

        // Рисуем все треугольники с эффектом пульсации
        for (size_t i = 0; i < triangles.size(); i++) {
            int i0,i1,i2; std::tie(i0,i1,i2) = triangles[i];
            Vector v0 = transformedVertices[i0];
            Vector v1 = transformedVertices[i1];
            Vector v2 = transformedVertices[i2];

            if (i < 2) {
                // Левая ножка - волны
                WaveShader shader(LETTER_COLOR, time);
                frame.DrawTriangle(v0.x, v0.y, v1.x, v1.y, v2.x, v2.y, shader);
            } else if (i < 4) {
                // Правая ножка - блики
                ShinyShader shader(LETTER_COLOR, time);
                frame.DrawTriangle(v0.x, v0.y, v1.x, v1.y, v2.x, v2.y, shader);
            } else {
                // Перекладина - капли
                DropShader shader(LETTER_COLOR, time);
                frame.DrawTriangle(v0.x, v0.y, v1.x, v1.y, v2.x, v2.y, shader);
            }
        }
    }
```

```
};


// Класс для буквы "T"
class LetterT {
private:
    std::vector<Vector> vertices;
    std::vector<std::tuple<int,int,int>> triangles;
public:
    LetterT() {
        vertices = {
            // Верхняя перекладина
            Vector(0.0f,1.8f), Vector(1.0f,1.8f), Vector(1.0f,2.0f), Vector(0.0f,2.0f),
            // Вертикальная ножка
            Vector(0.4f,0.0f), Vector(0.6f,0.0f), Vector(0.6f,1.8f), Vector(0.4f,1.8f)
        };

        triangles = {
            {0,1,2}, {0,2,3},   // верхняя перекладина
            {4,5,6}, {4,6,7}    // вертикальная ножка
        };
    }

    void Draw(Frame& frame, const Matrix& transform, float time, float alpha) {
        std::vector<Vector> transformedVertices;
        for (auto& v : vertices)
            transformedVertices.push_back(v * transform);

        for (size_t i = 0; i < triangles.size(); i++) {
            int i0, i1, i2;
            std::tie(i0, i1, i2) = triangles[i];
            Vector v0 = transformedVertices[i0];
            Vector v1 = transformedVertices[i1];
            Vector v2 = transformedVertices[i2];

            if (i < 2) {
                // Верхняя перекладина
                WaveShader shader(LETTER_COLOR, time);
                frame.DrawTriangle(v0.x, v0.y, v1.x, v1.y, v2.x, v2.y, shader);
            } else {
                // Вертикальная ножка
                ShinyShader shader(LETTER_COLOR, time);
                frame.DrawTriangle(v0.x, v0.y, v1.x, v1.y, v2.x, v2.y, shader);
            }
        }
    }
};

int main(int argc, char* argv[]) {
    if (SDL_Init(SDL_INIT_VIDEO) != 0) {
        SDL_Log("Не удалось инициализировать SDL: %s", SDL_GetError());
        return 1;
    }

    SDL_Window* window = SDL_CreateWindow("Лабораторная работа №3 - Аффинные преобразования и эффекты",
                                          SDL_WINDOWPOS_CENTERED,
                                          SDL_WINDOWPOS_CENTERED,
                                          INITIAL_WIDTH, INITIAL_HEIGHT,
                                          SDL_WINDOW_RESIZABLE);
    if (!window) {
        SDL_Log("Не удалось создать окно: %s", SDL_GetError());
```

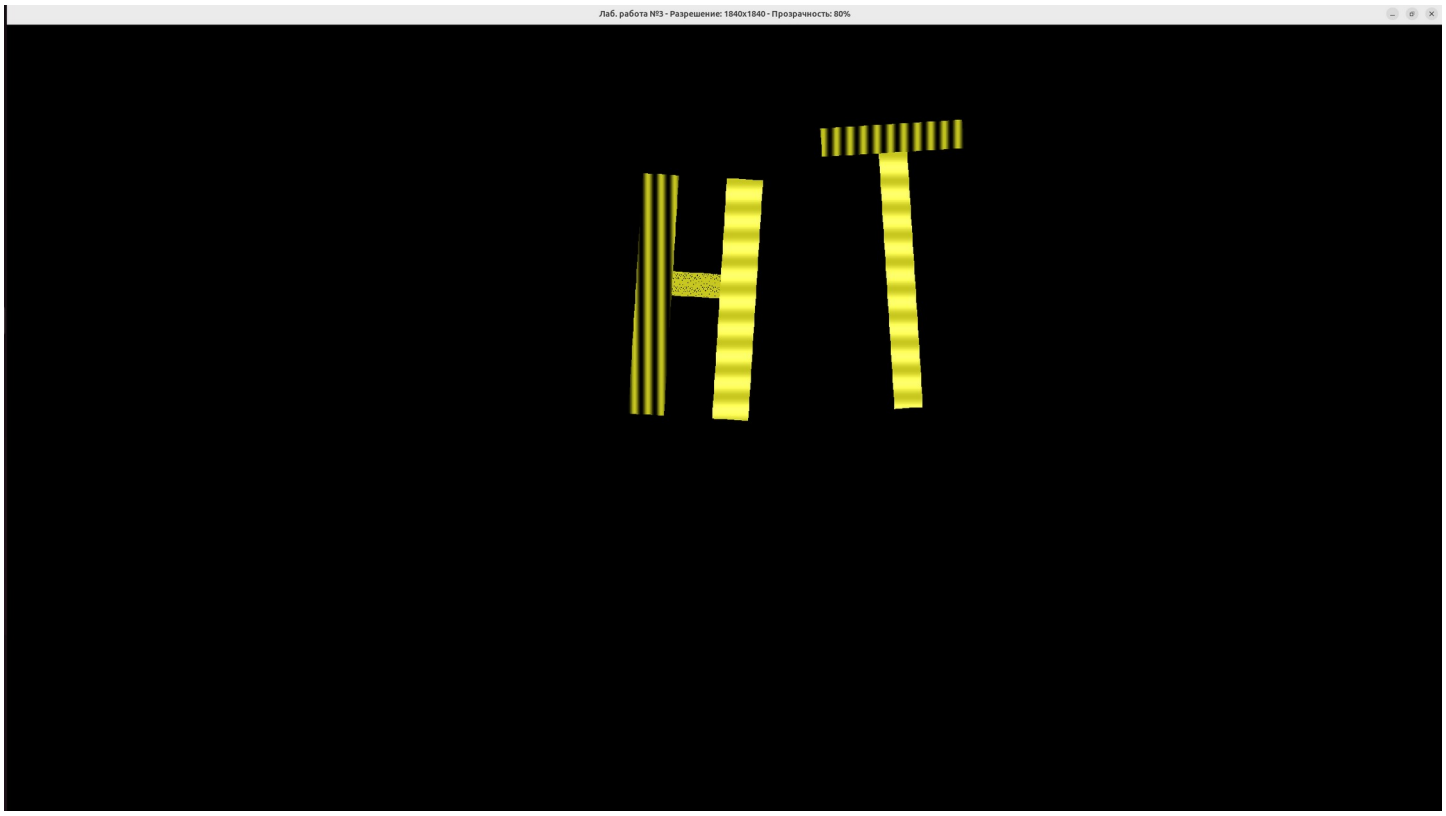
```
    SDL_Quit();
    return 1;
}

SDL_Renderer* renderer = SDL_CreateRenderer(window, -1,
                                            SDL_RENDERER_ACCELERATED |
                                            SDL_RENDERER_PRESENTVSYNC);
if (!renderer) {
    SDL_Log("Не удалось создать рендерер: %s", SDL_GetError());
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 1;
}

SDL_PixelFormat* format = SDL_AllocFormat(SDL_PIXELFORMAT_RGBA32);
if (!format) {
    SDL_Log("Не удалось получить формат пикселей: %s", SDL_GetError());
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 1;
}

int logicalWidth = 500;
int logicalHeight = 500;
Frame frame(logicalWidth, logicalHeight);

SDL_Texture* texture = SDL_CreateTexture(renderer,
                                         SDL_PIXELFORMAT_RGBA32,
                                         SDL_TEXTUREACCESS_STREAMING,
                                         logicalWidth, logicalHeight);
if (!texture) {
    SDL_Log("Не удалось создать текстуру: %s", SDL_GetError());
    SDL_FreeFormat(format);
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 1;
}

LetterN letterN;
LetterT letterT;

// Получаем текущий размер окна для правильного масштабирования координат
int windowWidth = INITIAL_WIDTH;
int windowHeight = INITIAL_HEIGHT;

auto updateWindowTitle = [window, &logicalWidth, &logicalHeight]() {
    std::string title = "Лаб. работа №3 - Разрешение: " +
                        std::to_string(logicalWidth) + "x" +
                        std::to_string(logicalHeight) +
                        " - Прозрачность: " + std::to_string(static_cast<int>(global_alpha * 100)) + "%";
    SDL_SetWindowTitle(window, title.c_str());
};

updateWindowTitle();

bool running = true;
SDL_Event event;
Uint32 startTime = SDL_GetTicks();
```

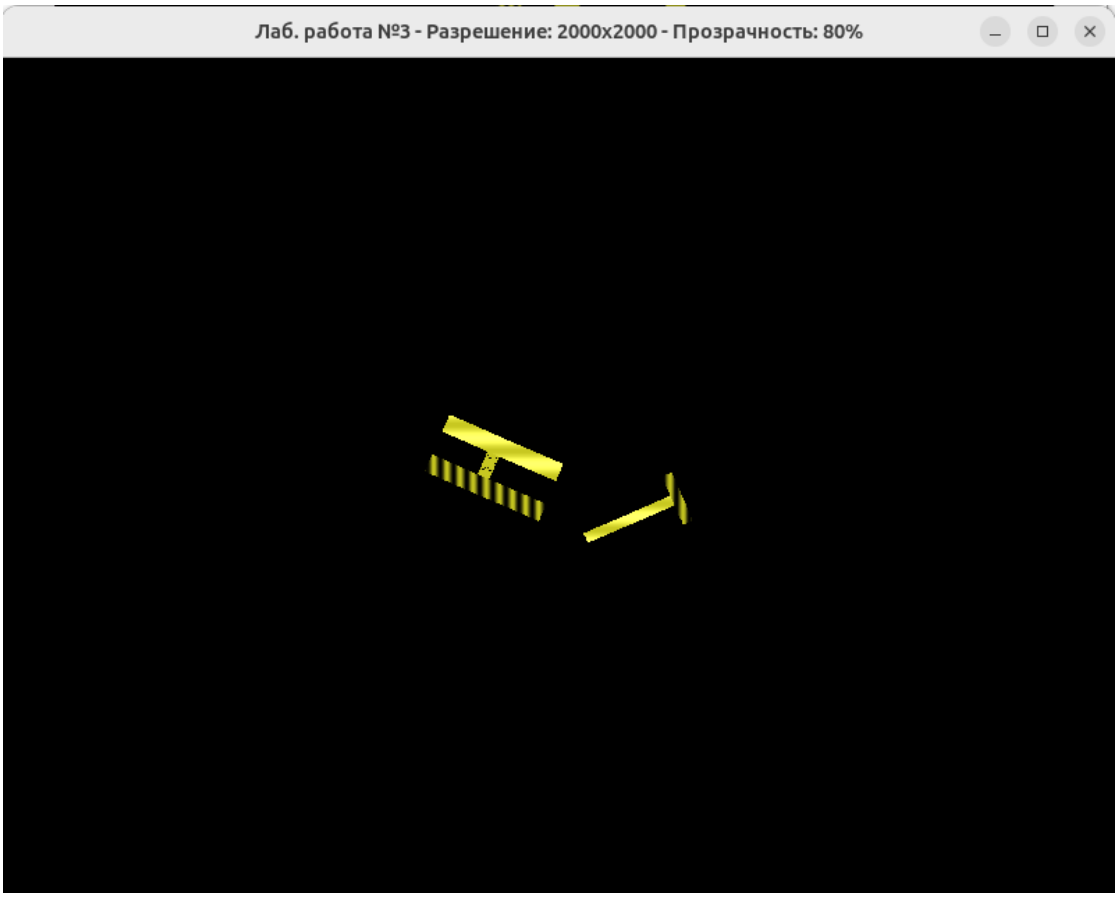
```
while (running) {
    Uint32 frameStart = SDL_GetTicks();
    float time = (frameStart - startTime) / 1000.0f;

    while (SDL_PollEvent(&event)) {
        if (event.type == SDL_QUIT) {
            running = false;
        } else if (event.type == SDL_WINDOWEVENT) {
            // Обновляем размеры окна при изменении
            if (event.window.event == SDL_WINDOWEVENT_RESIZED) {
                windowWidth = event.window.data1;
                windowHeight = event.window.data2;
            }
        } else if (event.type == SDL_KEYDOWN) {
            if (event.key.keysym.sym == SDLK_F2) {
                logicalWidth = std::max(MIN_RESOLUTION, logicalWidth - 10);
                logicalHeight = logicalWidth;
                frame.Resize(logicalWidth, logicalHeight);
                SDL_DestroyTexture(texture);
                texture = SDL_CreateTexture(renderer,
                                            SDL_PIXELFORMAT_RGBA32,
                                            SDL_TEXTUREACCESS_STREAMING,
                                            logicalWidth, logicalHeight);
                updateWindowTitle();
            } else if (event.key.keysym.sym == SDLK_F3) {
                logicalWidth = std::min(MAX_RESOLUTION, logicalWidth + 10);
                logicalHeight = logicalWidth;
                frame.Resize(logicalWidth, logicalHeight);
                SDL_DestroyTexture(texture);
                texture = SDL_CreateTexture(renderer,
                                            SDL_PIXELFORMAT_RGBA32,
                                            SDL_TEXTUREACCESS_STREAMING,
                                            logicalWidth, logicalHeight);
                updateWindowTitle();
            } else if (event.key.keysym.sym == SDLK_PLUS || event.key.keysym.sym == SDLK_KP_PLUS) {
                // Увеличение прозрачности
                global_alpha = std::min(global_alpha + 0.1f, 1.0f);
                updateWindowTitle();
            } else if (event.key.keysym.sym == SDLK_MINUS || event.key.keysym.sym == SDLK_KP_MINUS) {
                // Уменьшение прозрачности
                global_alpha = std::max(global_alpha - 0.1f, 0.1f);
                updateWindowTitle();
            }
        } else if (event.type == SDL_MOUSEBUTTONDOWN) {
            // Активируем лупу при клике мыши с правильным масштабированием координат
            if (event.button.button == SDL_BUTTON_LEFT) {
                magnifier_active = true;
                // Правильное преобразование координат с учетом текущего размера окна
                magnifier_x = (event.button.x * logicalWidth) / windowWidth;
                magnifier_y = (event.button.y * logicalHeight) / windowHeight;
            }
        } else if (event.type == SDL_MOUSEBUTTONUP) {
            // Деактивируем лупу при отпускании кнопки мыши
            if (event.button.button == SDL_BUTTON_LEFT) {
                magnifier_active = false;
            }
        } else if (event.type == SDL_MOUSEMOTION) {
            // Обновляем позицию лупы при движении мыши с правильным масштабированием
            if (magnifier_active) {
                magnifier_x = (event.motion.x * logicalWidth) / windowWidth;
```

```
            magnifier_y = (event.motion.y * logicalHeight) / windowHeight;
        }
    }
}

global_angle += rotation_speed;
if (global_angle > 360.0f) {
    global_angle -= 360.0f;
}

// Черный фон
frame.Clear(COLOR(0, 0, 0, 255));

// Матрица для преобразования мировых координат в экранные
Matrix WS = Matrix::WorldToScreen(50, 50, logicalWidth - 50, logicalHeight - 50,
                                  -2, -2, 2, 2);

// Анимация для буквы "H"
float scaleN = 0.5f + 0.3f * sin(time * 2.0f);
Matrix transformN = Matrix::Translation(-0.8f, 0.0f) *
                    Matrix::Rotation(time * 90.0f) *
                    Matrix::Scaling(scaleN, scaleN) *
                    WS;
letterN.Draw(frame, transformN, time, global_alpha);

// Анимация для буквы "T"
float scaleT = 0.5f + 0.3f * cos(time * 2.0f);
Matrix transformT = Matrix::Translation(0.8f, 0.0f) *
                    Matrix::Rotation(-time * 90.0f) *
                    Matrix::Scaling(scaleT, scaleT) *
                    WS;
letterT.Draw(frame, transformT, time, global_alpha);

// Применяем эффект лупы, если он активен
if (magnifier_active) {
    frame.ApplyMagnifier(magnifier_x, magnifier_y, magnifier_radius, magnifier_zoom);
}

// Обновляем текстуру
void* texturePixels;
int pitch;
SDL_LockTexture(texture, NULL, &texturePixels, &pitch);
Uint32* texPixels = static_cast<Uint32*>(texturePixels);

for (int y = 0; y < logicalHeight; y++) {
    for (int x = 0; x < logicalWidth; x++) {
        COLOR color = frame.GetPixel(x, y);
        texPixels[y * (pitch / sizeof(Uint32)) + x] =
            SDL_MapRGBA(format, color.r, color.g, color.b, color.a);
    }
}
SDL_UnlockTexture(texture);

// Очищаем рендерер и копируем текстуру с правильным масштабированием
SDL_RenderClear(renderer);

// Создаем прямоугольник назначения с правильными пропорциями
SDL_Rect dstRect;
int currentWidth, currentHeight;
SDL_GetWindowSize(window, &currentWidth, &currentHeight);
```

```
        // Сохраняем пропорции логического разрешения
        float aspectRatio = (float)logicalWidth / logicalHeight;
        float windowAspect = (float)currentWidth / currentHeight;

        if (windowAspect > aspectRatio) {
            // Окно шире - добавляем черные полосы по бокам
            int height = currentHeight;
            int width = (int)(height * aspectRatio);
            dstRect.x = (currentWidth - width) / 2;
            dstRect.y = 0;
            dstRect.w = width;
            dstRect.h = height;
        } else {
            // Окно выше - добавляем черные полосы сверху и снизу
            int width = currentWidth;
            int height = (int)(width / aspectRatio);
            dstRect.x = 0;
            dstRect.y = (currentHeight - height) / 2;
            dstRect.w = width;
            dstRect.h = height;
        }

        SDL_RenderCopy(renderer, texture, NULL, &dstRect);
        SDL_RenderPresent(renderer);

        Uint32 frameTime = SDL_GetTicks() - frameStart;
        if (frameTime < 33) {
            SDL_Delay(33 - frameTime);
        }
    }

    SDL_FreeFormat(format);
    SDL_DestroyTexture(texture);
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();

    return 0;
}
```

Результат работы программы:





Вывод: в ходе выполнения л.р были получены навыки выполнения аффинных преобразований на плоскости и создание графического приложения на языке C++ для создания простейшей анимации.