Course Name: Computer Graphics Fundamentals

Project Title: Assignment 1

Name: Tangnur Bolatkhankyzy

Instructor: Kamila Batkuldinova

Date of Submission: 21.09.2025

Table of Contents

1. Introduction

This project explores modern 2D rendering with OpenGL 3.3 Core Profile using C++. It emphasizes the programmable pipeline (vertex/fragment shaders), vertex buffer objects (VBO), vertex array objects (VAO), and basic parametric geometry.
We implemented three deliverables:
• Task 1: A red triangle and a blue square (two separate programs).
• Task 2 (Part 1): A composite scene: gradient triangle, a circle and an ellipse, and concentric black/white squares.
• Task 2 (Part 2): A second scene: at least three different objects using creative color ranges (star, ring, hexagon).

2. Project Description

2.1 Objective

Build multiple OpenGL programs without deprecated functions, pass vertex/colour data through shader attributes (vPosition and vColor), and render scenes using only GL_TRIANGLES, GL_TRIANGLE_FAN, and GL_TRIANGLE_STRIP.

2.2 Problem Statement

Learn to implement basic 2D shapes and visual effects using the modern OpenGL pipeline, which is foundational for future 3D graphics and shading techniques.

2.3 Scope

Included: OpenGL 3.3 Core, GLFW (window/context), GLAD (function loader), C++17, Makefile, parametric primitives (circle/ellipse), colour interpolation, required primitive modes.
Excluded: Textures, advanced lighting (Phong/PBR), animation, complex 3D geometry.

3. Technical Implementation

3.1 Tools and Technologies Used

• Language: C++17
• API: OpenGL 3.3 Core Profile
• Libraries:
• GLFW for window/context creation and input
• GLAD as the OpenGL function loader
• Environment: VS Code + MSYS2 MinGW-w64 (g++)
• Build: Makefile with targets: red_triangle, blue_square, task2_scene, part2_scene

3.2 Development Process

1. Configure MSYS2 MinGW64 in VS Code and generate GLAD.
2. Write minimal shaders and pipeline for Task 1 programs.
3. Build Task 2 (Part 1) composite scene strictly meeting the spec (names vPosition/vColor,

required draw modes).

4. Design Part 2 (creative scene) with at least three different objects.
5. Debug: toolchain profile (use MINGW64), Makefile tabs, Windows linking, constant M_PI.

3.3 Algorithms and Techniques

This project consists of several OpenGL programs (Task 1, Task 2, Part 2). Each program generates vertex data in C++ and passes it through vertex/fragment shaders to render 2D shapes. Below, the main algorithms and techniques are described together with key code snippets.

Common Initialization (identical across all programs)

1) Headers and standard libraries

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <vector>
#include <cmath>
#include <cstdio>
#include <cstdlib>
```

These headers are included in every file.

- GLAD loads OpenGL functions.
- GLFW creates the window and manages the context.
- Standard libraries provide math functions, vectors, and input/output utilities.

2) Shader source definitions

```
// Vertex Shader
layout(location=0) in vec2 vPosition;
layout(location=1) in vec3 vColor;
out vec3 color;
void main(){
    color = vColor;
    gl_Position = vec4(vPosition, 0.0, 1.0);
}
// Fragment Shader
in vec3 color;
out vec4 FragColor;
void main(){
    FragColor = vec4(color, 1.0);
}
```

The same pair of shaders is used in all programs.

- vPosition stores the vertex coordinates.
- vColor stores the vertex color.
- The fragment shader outputs the interpolated color.

3) Window-resize callback and error check

```
static void framebuffer_size_callback(GLFWwindow*, int w, int h){
    glViewport(0,0,w,h);
}
static void check(bool ok, const char* msg){
    if(!ok){
        std::fprintf(stderr,"[!] %s\n", msg);
        std::exit(1);
    }
}
```

- framebuffer_size_callback adjusts the viewport when the window is resized.
- check is a utility for error handling.
- Both functions appear unchanged in every program.

4) Shader program builder

```
static GLuint makeProgram(const char* vs, const char* fs){
    GLuint v = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(v, 1, &vs, nullptr); glCompileShader(v);
    GLint ok=0; glGetShaderiv(v, GL_COMPILE_STATUS, &ok);
    if(!ok){ char log[1024]; glGetShaderInfoLog(v,1024,nullptr,log);
std::fprintf(stderr,"VS error:\n%s\n",log); std::exit(1); }

    GLuint f = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(f, 1, &fs, nullptr); glCompileShader(f);
    glGetShaderiv(f, GL_COMPILE_STATUS, &ok);
    if(!ok){ char log[1024]; glGetShaderInfoLog(f,1024,nullptr,log);
std::fprintf(stderr,"FS error:\n%s\n",log); std::exit(1); }

    GLuint p = glCreateProgram(); glAttachShader(p,v); glAttachShader(p,f);
glLinkProgram(p);
    glGetProgramiv(p, GL_LINK_STATUS, &ok);
    if(!ok){ char log[1024]; glGetProgramInfoLog(p,1024,nullptr,log);
std::fprintf(stderr,"Link error:\n%s\n",log); std::exit(1); }
    glDeleteShader(v); glDeleteShader(f);
    return p;
}
```

- A helper function (makeProgram) compiles the vertex and fragment shaders, checks for errors, and links them into a single program. It is the same in every file.

Task 1:

Common Initialization for Red Triangle and Blue Square

```
int main(){
    check(glfwInit(), "glfwInit failed");
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR,3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR,3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    GLFWwindow* win = glfwCreateWindow(500,500,"Task 1 – Blue Square",
nullptr, nullptr);
    check(win!=nullptr, "glfwCreateWindow failed");
    glfwMakeContextCurrent(win);
    glfwSetFramebufferSizeCallback(win, framebuffer_size_callback);

    check(gladLoadGLLoader((GLADloadproc)glfwGetProcAddress),
"gladLoadGLLoader failed");
    glViewport(0,0,500,500);

    GLuint prog = makeProgram(VS,FS);
```

- GLFW Initialization: sets OpenGL version to 3.3 Core profile.
- Window Creation: opens a 500×500 window.
- GLAD Initialization: loads OpenGL functions.
- Viewport Setup: ensures that rendering fills the window.
- Shader Program: compiles and links vertex and fragment shaders through makeProgram.

Common Render Loop:

```
while(!glfwWindowShouldClose(win)){
        glClearColor(0.05f,0.05f,0.05f,1.0f);
        glClear(GL_COLOR_BUFFER_BIT);
        glUseProgram(prog);
        glBindVertexArray(vao);
        glDrawArrays(GL_TRIANGLES, 0, 3);
        glfwSwapBuffers(win);
        glfwPollEvents();
    }
```

- Clears the window with a dark gray background.
- Activates the compiled shader program.
- Binds the VAO that stores vertex data.
- Executes glDrawArrays, where the primitive type (GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN) and number of vertices depend on the shape.
- Swaps buffers and processes input events.

The only differences between the two programs are in the vertex data (triangle vs. square) and the drawing command (GL_TRIANGLES vs. GL_TRIANGLE_STRIP).

Red triangle

Vertex Data:

```
float verts[] = {
   -0.5f, -0.4f, 1.f,0.f,0.f, // left, red
    0.5f, -0.4f, 1.f,0.f,0.f, // right, red
    0.0f, 0.5f, 1.f,0.f,0.f // top, red
};
```

- Defines 3 vertices. Each vertex stores:
- 2D position (x,y)
- RGB color (r,g,b)
- All vertices are pure red (1,0,0), ensuring the triangle is solid red.

VAO and VBO Setup:

```
GLuint vao,vbo;
glGenVertexArrays(1,&vao);
glGenBuffers(1,&vbo);
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(verts), verts, GL_STATIC_DRAW);
```

- Creates a Vertex Array Object (VAO) and a Vertex Buffer Object (VBO).
- Uploads the vertex data to GPU memory.
- VAO stores the configuration of how data is interpreted.

Vertex Attribute Pointers:

```
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 5*sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 5*sizeof(float),
(void*)(2*sizeof(float)));
glEnableVertexAttribArray(1);
```

- Attribute 0: vPosition → 2 floats (x,y).
- Attribute 1: vColor → 3 floats (r,g,b).
- The stride is 5*sizeof(float) (because each vertex has 5 floats total).
- Enables these attributes so the shader can access them.

Blue Square

Vertex Data:

```
float square[] = {
   -0.5f,-0.5f, 0,0,1,  // BL
    0.5f,-0.5f, 0,0,1,  // BR
   -0.5f, 0.5f, 0,0,1,  // TL
    0.5f, 0.5f, 0,0,1   // TR
};
```

- Defines 4 vertices.
- Each vertex contains (x, y) coordinates and an RGB color.
- All colors are (0, 0, 1), producing a solid blue output.
- The coordinates range from -0.5 to +0.5, creating a square centered at the origin.

VAO and VBO Setup:

```
GLuint vao,vbo;
glGenVertexArrays(1,&vao);
glGenBuffers(1,&vbo);

glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(square), square, GL_STATIC_DRAW);
```

- Creates a Vertex Array Object (VAO) and Vertex Buffer Object (VBO).
- Uploads the square vertex data to GPU memory.

Attribute Pointers:

```
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 5*sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 5*sizeof(float),
(void*)(2*sizeof(float)));
glEnableVertexAttribArray(1);
```

- Attribute 0 → vPosition (2 floats: x, y).
- Attribute 1 → vColor (3 floats: r, g, b).
- Ensures that the shader receives correct data for position and color.

Task 2 (Part 1):

Function: makeCircleFan

```
static std::vector<float> makeCircleFan(float cx, float cy, float r, float
yScale,
                                        int segments, float redPhase){
    std::vector<float> v; v.reserve((segments+2)*5);
    v.insert(v.end(), {cx, cy, 0.18f, 0.f, 0.f});
    const float TAU = 6.28318530718f;
    for(int i=0;i<=segments;++i){
        float a = (float)i/segments * TAU;
        float x = cx + r * std::cos(a);
        float y = cy + (r * yScale) * std::sin(a);
        float red = 0.35f + 0.65f * (0.5f*(1.0f+std::cos(a+redPhase)));
        v.insert(v.end(), {x, y, red, 0.f, 0.f});
    }
    return v;
}
```

- GL_TRIANGLE_FAN requires the first vertex to be the center of the fan. Therefore, the function inserts (cx, cy) as the initial vertex.
- The parametric circle equation is used with an additional yScale factor to turn the circle into an ellipse if needed.
- The loop runs from 0 to 2π in small increments (segments), generating evenly spaced points along the perimeter.
- The red gradient is calculated as a cosine-based function of angle: brighter at some positions and darker at others. This gives the right-hand circle in Task 2 a smooth shading effect.

Function: makeCircleFanSolid

```
static std::vector<float> makeCircleFanSolid(float cx, float cy, float r,
float yScale,
                                             int segments, float R, float G,
float B){
    std::vector<float> v; v.reserve((segments+2)*5);
    v.insert(v.end(), {cx, cy, R, G, B});
    const float TAU = 6.28318530718f;
    for(int i=0;i<=segments;++i){
        float a = (float)i/segments * TAU;
        float x = cx + r * std::cos(a);
        float y = cy + (r * yScale) * std::sin(a);
        v.insert(v.end(), {x, y, R, G, B});
    }
    return v;
}
```

- Generates a filled circle or ellipse using GL_TRIANGLE_FAN.

- The first vertex is the center.
- Subsequent vertices are computed by parametric equations.
- All vertices have the same color (R,G,B), producing a solid appearance.
- When yScale = 0.6, the circle is compressed vertically → an ellipse.

Function: makeSquareByAngleFilled

```
static std::vector<float> makeSquareByAngleFilled(float cx, float cy, float
radius, float gray){
    float c[4][2];
    const float start = 3.1415926535f/4.0f; // π/4
    for(int i=0;i<4;++i){
        float a = start + i*(3.1415926535f/2.0f);
        c[i][0] = cx + radius * std::cos(a);
        c[i][1] = cy + radius * std::sin(a);
    }
    std::vector<float> v = {
        c[2][0], c[2][1], gray,gray,gray, // BL
        c[3][0], c[3][1], gray,gray,gray, // BR
        c[1][0], c[1][1], gray,gray,gray, // TL
        c[0][0], c[0][1], gray,gray,gray  // TR
    };
    return v;
}
```

- Generates a filled square based on polar angles.
- Vertices are computed at π/4 and increments of π/2 → oriented at 45°.
- Reordered for correct GL_TRIANGLE_STRIP sequence (BL, BR, TL, TR).
- The grayscale value gray controls color (0.0 = black, 1.0 = white).
- Multiple squares with alternating black and white produce the "frame" effect.

Main Program Flow

1. Initialization:

Same as in Task 1: GLFW and GLAD setup, window creation (500×500), viewport configuration, and shader compilation with makeProgram.

2. Triangle (nose):

```
float tri[] = {
         0.00f,  0.65f,    1.f,0.f,0.f,
        -0.25f,  0.25f,    0.f,1.f,0.f,
         0.25f,  0.25f,    0.f,0.f,1.f
    };
```

- A simple RGB triangle.

- Each vertex has a different color (red, green, blue).
- Rendered with GL_TRIANGLES.
- Produces a gradient fill through color interpolation.

3. Ellipse and Circle (eyes):

```
auto leftEllipse  = makeCircleFanSolid(-0.55f, 0.60f, 0.17f, 0.60f, 96, 1.0f,
0.0f, 0.0f);
auto rightCircle  = makeCircleFan(      0.55f, 0.60f, 0.17f, 1.00f, 96,
1.2f);
```

- Left: a solid red ellipse (yScale=0.6).
- Right: a circle with shaded red perimeter (gradient by angle).
- Both rendered with GL_TRIANGLE_FAN.

4. Concentric Squares (mouth):

```
const float cx = 0.f, cy = -0.35f;
auto sq1_white = makeSquareByAngleFilled(cx, cy, 0.57f, 1.00f); // outer
white (slightly   thinner)
auto gap1_blk  = makeSquareByAngleFilled(cx, cy, 0.47f, 0.00f); // black gap
auto sq2_white = makeSquareByAngleFilled(cx, cy, 0.36f, 1.00f); // medium
white
auto gap2_blk  = makeSquareByAngleFilled(cx, cy, 0.26f, 0.00f); // black gap
auto sq3_white = makeSquareByAngleFilled(cx, cy, 0.165f,1.00f); // inner
white
auto inner_blk = makeSquareByAngleFilled(cx, cy, 0.095f,0.00f); // black
center
```

- Six squares stacked on top of each other.
- Alternating white and black values create a set of frames.
- Rendered with GL_TRIANGLE_STRIP (4 vertices each).

5. Upload to Buffers:

```
auto upload = [&](int i, const float* data, size_t floats){
        glBindVertexArray(vao[i]);
        glBindBuffer(GL_ARRAY_BUFFER, vbo[i]);
        glBufferData(GL_ARRAY_BUFFER, floats*sizeof(float), data,
GL_STATIC_DRAW);
        glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 5*sizeof(float),
(void*)0);
        glEnableVertexAttribArray(0); // vPosition
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 5*sizeof(float),
(void*)(2*sizeof(float)));
        glEnableVertexAttribArray(1); // vColor
    };
```

- A helper lambda function used to upload geometry data into GPU buffers.
- Simplifies VAO/VBO setup for multiple objects.
- Each object (triangle, ellipse, circle, and squares) is assigned a dedicated VAO/VBO.

6. Render Loop:

```
while(!glfwWindowShouldClose(win)){
        glClearColor(0.0f,0.0f,0.0f,1.0f);
        glClear(GL_COLOR_BUFFER_BIT);
        glUseProgram(prog);

        glBindVertexArray(vao[1]); glDrawArrays(GL_TRIANGLE_FAN, 0,
(GLsizei)(leftEllipse.size()/5));
        glBindVertexArray(vao[2]); glDrawArrays(GL_TRIANGLE_FAN, 0,
(GLsizei)(rightCircle.size()/5));
        glBindVertexArray(vao[0]); glDrawArrays(GL_TRIANGLES,    0, 3);

        for(int i=3;i<=8;++i){
            glBindVertexArray(vao[i]);
            glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
        }

        glfwSwapBuffers(win);
        glfwPollEvents();
    }
```

- Clears the screen black.
- Draws the ellipse and circle (eyes), then the triangle (nose).
- Iterates through the VAOs for the squares to draw the mouth.
- Produces the composite scene.

Why this approach was used:

- GL_TRIANGLE_FAN is optimal for circles and ellipses, because a center point and perimeter vertices form a closed disk.
- GL_TRIANGLE_STRIP is ideal for squares: only 4 vertices are required to generate 2 triangles.
- A combination of these primitives matches the task's requirement to use GL_TRIANGLES, GL_TRIANGLE_FAN, and GL_TRIANGLE_STRIP.
- The alternation of white and black squares produces the required frame effect.

Part 2: Creative Scene

Function: makeStar

```
static std::vector<float> makeStar(float cx,float cy,float rOut,float rIn){
    std::vector<float> v;
    v.insert(v.end(),{cx,cy,1,1,0}); // center yellow
    for(int i=0;i<=10;i++){
        float a=(M_PI/2)+(i*36.0f*M_PI/180.0f); // 36° per step
        float r=(i%2==0)?rOut:rIn;
        float x=cx+r*cos(a), y=cy+r*sin(a);
        v.insert(v.end(),{x,y,1,0.5f,0}); // orange
    }
    return v;
}
```

- Draws a five-point star using GL_TRIANGLE_FAN.
- Starts with a yellow center point.
- Alternates between outer radius (rOut) and inner radius (rIn) every 36°, producing 10 perimeter vertices.
- The star is rendered in orange shades.

Function: makeRing

```
static std::vector<float> makeRing(float cx,float cy,float rOut,float rIn,int
seg){
    std::vector<float> v;
    for(int i=0;i<=seg;i++){
        float a=i*2*M_PI/seg;
        float co=cos(a), si=sin(a);
        v.insert(v.end(),{cx+rOut*co,cy+rOut*si,0,1,1}); // outer cyan
        v.insert(v.end(),{cx+rIn*co, cy+rIn*si, 0,0.2f,0.5f}); // inner dark
    }
    return v;
}
```

- Generates a ring (annulus) using GL_TRIANGLE_STRIP.
- For each angle step, two vertices are created: one on the outer circle and one on the inner circle.
- When connected as a strip, these form a continuous band.
- Outer vertices are cyan, inner vertices are dark blue, producing a shaded ring effect.

Function: makeHexagon

```
static std::vector<float> makeHexagon(float cx,float cy,float r){
    std::vector<float> v;
    v.insert(v.end(),{cx,cy,0.8f,0,0.8f}); // center purple
    for(int i=0;i<=6;i++){
        float a=M_PI/6+i*M_PI/3; // start 30°, step 60°
        float x=cx+r*cos(a), y=cy+r*sin(a);
        v.insert(v.end(),{x,y,0.3f+0.1f*i,0.1f*i,1.0f-0.1f*i}); // gradient
    }
    return
```

- Creates a hexagon with GL_TRIANGLE_FAN.
- Starts at 30° and steps by 60° to compute six perimeter vertices.
- A purple center is connected to these points, forming six triangles.
- Vertex colors gradually change to produce a gradient around the hexagon.

Main Program Flow

1. Initialization:

GLFW and GLAD setup, window creation (500×500), viewport configuration, shader program compilation (makeProgram).

2. Shape Generation:

```
auto star=makeStar(-0.5f,0.5f,0.2f,0.08f);
auto ring=makeRing( 0.5f,0.5f,0.2f,0.14f,64);
auto hex =makeHexagon(0.0f,-0.3f,0.3f);
```

- Generates vertex data for three objects with different parameters.

3. Buffer Upload:

```
auto upload=[&](int i,const std::vector<float>&d){
    glBindVertexArray(vao[i]); glBindBuffer(GL_ARRAY_BUFFER,vbo[i]);

glBufferData(GL_ARRAY_BUFFER,d.size()*sizeof(float),d.data(),GL_STATIC_DRAW);
    glVertexAttribPointer(0,2,GL_FLOAT,GL_FALSE,5*sizeof(float),(void*)0);
    glEnableVertexAttribArray(0);

glVertexAttribPointer(1,3,GL_FLOAT,GL_FALSE,5*sizeof(float),(void*)(2*sizeof(
float)));
    glEnableVertexAttribArray(1);
};
```

- Reusable lambda function for VAO/VBO setup.
- Called separately for each object.

4. Render Loop:

```
while(!glfwWindowShouldClose(win)){
    glClearColor(0,0,0,1);
    glClear(GL_COLOR_BUFFER_BIT);
    glUseProgram(prog);
    glBindVertexArray(vao[0]); glDrawArrays(GL_TRIANGLE_FAN,0,star.size()/5);
    glBindVertexArray(vao[1]);
glDrawArrays(GL_TRIANGLE_STRIP,0,ring.size()/5);
    glBindVertexArray(vao[2]); glDrawArrays(GL_TRIANGLE_FAN,0,hex.size()/5);
    glfwSwapBuffers(win);
    glfwPollEvents();
}
```

- Clears the screen to black.
- Draws the star, the ring, and the hexagon in sequence.
- Uses different primitives: GL_TRIANGLE_FAN for the star and hexagon, GL_TRIANGLE_STRIP for the ring.

Why this approach was used:

- The star requires alternating radii, which are best expressed with a triangle fan centered at the origin.
- The ring is naturally expressed as a strip, since each outer vertex connects to an inner vertex.
- The hexagon is efficiently drawn as a triangle fan with one central vertex and six perimeter vertices.
- Each object demonstrates a different rendering primitive, as required by the assignment.

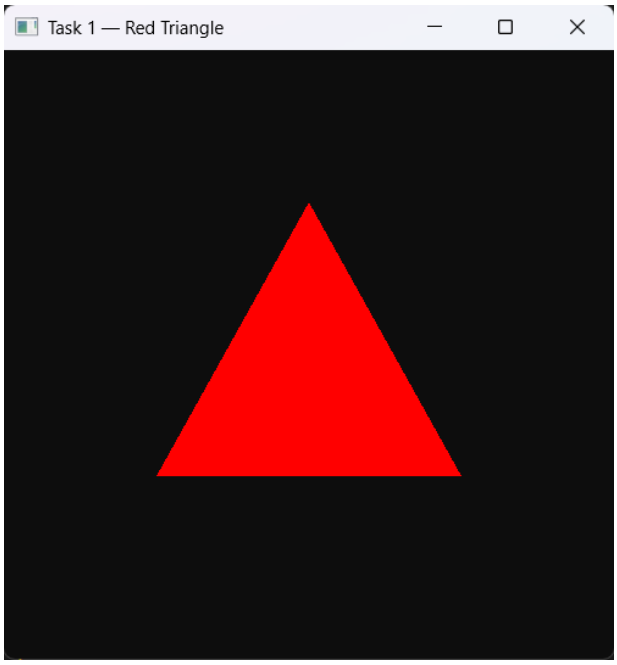
4. Graphics and Rendering

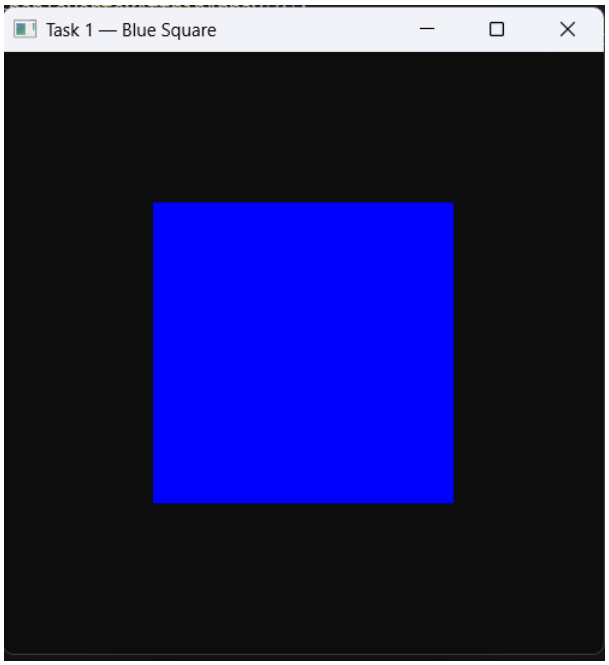
4.1 2D/3D Rendering Techniques

All shapes in this project were procedurally generated and rendered using fundamental OpenGL primitives. Specifically:

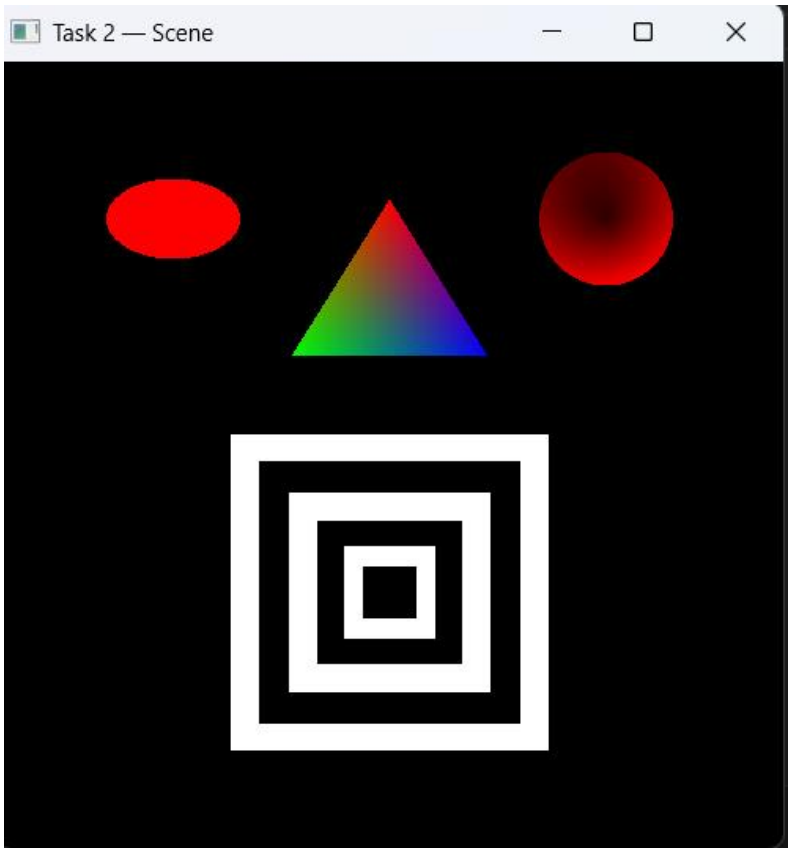
- glDrawArrays(GL_TRIANGLES, …) — used for the red triangle (Task 1) and the RGB triangle (Task 2).
- glDrawArrays(GL_TRIANGLE_FAN, …) — used for circles, ellipses, and hexagons. This primitive is ideal because the first vertex is the center and all other vertices form connected triangles radiating outward.
- glDrawArrays(GL_TRIANGLE_STRIP, …) — used for squares and rings, as it minimizes vertex duplication while still covering rectangular areas.

- Task 1: red triangle; blue square (two separate programs).





• Task 2 (Part 1): final composite scene (triangle + circle + ellipse + square frames).



•

Task 2 (Part 2): creative scene (star + ring + hexagon).

4.2 Lighting and Shading

This project does not use advanced physical lighting models such as Phong or Blinn-Phong. Instead, visual quality is achieved through vertex color interpolation

5. Testing and Evaluation

5.1 Testing Methodology

Testing was carried out by:

- Compiling with make clean && make to ensure reproducibility.
- Running each binary (./bin/red_triangle, ./bin/blue_square, ./bin/task2_scene, ./bin/part2_scene) to confirm the window opened at 500×500 pixels and rendered the expected shapes.
- Visual inspection of geometric correctness:
- Circle smoothness depended on segments count (96 chosen for quality).
- Ellipse confirmed with yScale = 0.6.
- Square frames alternated correctly in white/black pattern.
- Star, ring, and hexagon showed expected symmetry.

5.2 Performance Evaluation

Because the vertex counts are very low (tens or hundreds of vertices at most), the application runs at stable 60 FPS with vsync enabled. Memory usage is negligible, since only static buffers

are uploaded once with glBufferData. This makes the implementation highly efficient and responsive.

5.3 Issues and Debugging

Several issues arose during development:

- Linking errors with GLFW and GLAD in MSYS2: fixed by ensuring correct library flags in the Makefile (-lglfw3 -lopengl32 -lgdi32 -luser32 -lkernel32).
- Circle drawing errors when the first vertex was omitted: solved by explicitly adding the center point before perimeter points for GL_TRIANGLE_FAN.
- Gradient not appearing: corrected by computing red intensity as a cosine-based function of angle instead of a constant.
- Nested squares thickness: adjusted radii manually to visually match the assignment's reference image.
- Part 2 shapes (star, ring, hexagon) initially misaligned: resolved by carefully calculating angular increments (36°, 60°, etc.) and using cos/sin consistently.

6. Conclusion and Future Work

This project successfully implemented a series of OpenGL programs that render primitive 2D shapes, both individually (Task 1) and in composite/creative scenes (Task 2 and Part 2). The assignment demonstrates fundamental graphics programming skills:

- Correct use of vertex and fragment shaders.
- Procedural generation of geometry (circles, ellipses, stars, polygons).
- Application of color interpolation to simulate shading.
- Efficient use of OpenGL primitives (TRIANGLES, TRIANGLE_FAN, TRIANGLE_STRIP).

Future improvements could include:

- Adding transformations (rotation, scaling, translation) controlled by user input.
- Experimenting with texture mapping to enrich visual effects.
- Extending to 3D primitives (cubes, spheres) and introducing basic lighting models.
- Applying animation techniques (rotating shapes, pulsing colors).

7. References

1. LearnOpenGL. *Getting Started with OpenGL* — https://learnopengl.com/Getting-started/OpenGL
2. LearnOpenGL. *Hello Triangle* — https://learnopengl.com/Getting-started/Hello-Triangle
3. LearnOpenGL. *Shaders* — https://learnopengl.com/Getting-started/Shaders

8.Appendix

Task 1:

Red triangle full code:

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <cstdio>
#include <cstdlib>

static const char* VS = R"(#version 330 core
layout(location=0) in vec2 aPos;
layout(location=1) in vec3 aColor;
out vec3 vColor;
void main(){
    vColor = aColor;
    gl_Position = vec4(aPos, 0.0, 1.0);
})";

static const char* FS = R"(#version 330 core
in vec3 vColor;
out vec4 FragColor;
void main(){
    FragColor = vec4(vColor, 1.0);
})";

static void framebuffer_size_callback(GLFWwindow*, int w, int h){
glViewport(0,0,w,h); }
static void check(bool ok, const char* msg){ if(!ok){
std::fprintf(stderr,"[!] %s\n", msg); std::exit(1);} }

static GLuint makeProgram(const char* vs, const char* fs){
    GLuint v = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(v, 1, &vs, nullptr); glCompileShader(v);
    GLint ok=0; glGetShaderiv(v, GL_COMPILE_STATUS, &ok);
    if(!ok){ char log[1024]; glGetShaderInfoLog(v,1024,nullptr,log);
std::fprintf(stderr,"VS error: %s\n",log); std::exit(1); }

    GLuint f = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(f, 1, &fs, nullptr); glCompileShader(f);
    glGetShaderiv(f, GL_COMPILE_STATUS, &ok);
    if(!ok){ char log[1024]; glGetShaderInfoLog(f,1024,nullptr,log);
std::fprintf(stderr,"FS error: %s\n",log); std::exit(1); }

    GLuint p = glCreateProgram(); glAttachShader(p,v); glAttachShader(p,f);
glLinkProgram(p);
    glGetProgramiv(p, GL_LINK_STATUS, &ok);
    if(!ok){ char log[1024]; glGetProgramInfoLog(p,1024,nullptr,log);
std::fprintf(stderr,"Link error: %s\n",log); std::exit(1); }
    glDeleteShader(v); glDeleteShader(f);
    return p;
}

int main(){
    check(glfwInit(), "glfwInit failed");
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR,3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR,3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    GLFWwindow* win = glfwCreateWindow(500,500,"Task 1 — Red Triangle",
nullptr, nullptr);
```

```
    check(win!=nullptr, "glfwCreateWindow failed");
    glfwMakeContextCurrent(win);
    glfwSetFramebufferSizeCallback(win, framebuffer_size_callback);

    check(gladLoadGLLoader((GLADloadproc)glfwGetProcAddress),
"gladLoadGLLoader failed");
    glViewport(0,0,500,500);

    GLuint prog = makeProgram(VS,FS);

    // 3 vertices: position (x,y) + color (r,g,b)
    float verts[] = {
        //    x      y       r   g   b
           -0.5f, -0.4f,   1.f,0.f,0.f,
            0.5f, -0.4f,   1.f,0.f,0.f,
            0.0f,  0.5f,   1.f,0.f,0.f,
    };

    GLuint vao,vbo;
    glGenVertexArrays(1,&vao);
    glGenBuffers(1,&vbo);
    glBindVertexArray(vao);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, sizeof(verts), verts, GL_STATIC_DRAW);
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 5*sizeof(float),
(void*)0);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 5*sizeof(float),
(void*)(2*sizeof(float)));
    glEnableVertexAttribArray(1);

    while(!glfwWindowShouldClose(win)){
        glClearColor(0.05f,0.05f,0.05f,1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        glUseProgram(prog);
        glBindVertexArray(vao);
        glDrawArrays(GL_TRIANGLES, 0, 3);

        glfwSwapBuffers(win);
        glfwPollEvents();
    }
    glfwTerminate();
    return 0;
}
```

Blue square full code:

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <cstdio>
#include <cstdlib>

static const char* VS = R"(#version 330 core
layout(location=0) in vec2 aPos;
layout(location=1) in vec3 aColor;
out vec3 vColor;
void main(){
    vColor = aColor;
    gl_Position = vec4(aPos, 0.0, 1.0);
})";
```

```
static const char* FS = R"(#version 330 core
in vec3 vColor;
out vec4 FragColor;
void main(){
    FragColor = vec4(vColor, 1.0);
})";

static void framebuffer_size_callback(GLFWwindow*, int w, int h){
glViewport(0,0,w,h); }
static void check(bool ok, const char* msg){ if(!ok){
std::fprintf(stderr,"[!] %s\n", msg); std::exit(1);} }

static GLuint makeProgram(const char* vs, const char* fs){
    GLuint v = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(v, 1, &vs, nullptr); glCompileShader(v);
    GLint ok=0; glGetShaderiv(v, GL_COMPILE_STATUS, &ok);
    if(!ok){ char log[1024]; glGetShaderInfoLog(v,1024,nullptr,log);
std::fprintf(stderr,"VS error: %s\n",log); std::exit(1); }

    GLuint f = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(f, 1, &fs, nullptr); glCompileShader(f);
    glGetShaderiv(f, GL_COMPILE_STATUS, &ok);
    if(!ok){ char log[1024]; glGetShaderInfoLog(f,1024,nullptr,log);
std::fprintf(stderr,"FS error: %s\n",log); std::exit(1); }

    GLuint p = glCreateProgram(); glAttachShader(p,v); glAttachShader(p,f);
glLinkProgram(p);
    glGetProgramiv(p, GL_LINK_STATUS, &ok);
    if(!ok){ char log[1024]; glGetProgramInfoLog(p,1024,nullptr,log);
std::fprintf(stderr,"Link error: %s\n",log); std::exit(1); }
    glDeleteShader(v); glDeleteShader(f);
    return p;
}

int main(){
    check(glfwInit(), "glfwInit failed");
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR,3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR,3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    GLFWwindow* win = glfwCreateWindow(500,500,"Task 1 — Blue Square",
nullptr, nullptr);
    check(win!=nullptr, "glfwCreateWindow failed");
    glfwMakeContextCurrent(win);
    glfwSetFramebufferSizeCallback(win, framebuffer_size_callback);

    check(gladLoadGLLoader((GLADloadproc)glfwGetProcAddress),
"gladLoadGLLoader failed");
    glViewport(0,0,500,500);

    GLuint prog = makeProgram(VS,FS);

    // Vertex data : Blue square
float square[] = {
   -0.5f,-0.5f, 0,0,1,  // BL
    0.5f,-0.5f, 0,0,1,  // BR
   -0.5f, 0.5f, 0,0,1,  // TL
    0.5f, 0.5f, 0,0,1   // TR
};

// VAo/VBO setup
GLuint vao,vbo;
glGenVertexArrays(1,&vao);
glGenBuffers(1,&vbo);
```

```
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(square), square, GL_STATIC_DRAW);


//attributes: location 0 = vPosition (x,y), location 1= vColor (r,g,b)
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 5*sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 5*sizeof(float),
(void*)(2*sizeof(float)));
glEnableVertexAttribArray(1);


// Render loop
while(!glfwWindowShouldClose(win)){
    glClearColor(0.05f,0.05f,0.05f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glUseProgram(prog);
    glBindVertexArray(vao);
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

    glfwSwapBuffers(win);
    glfwPollEvents();
}


    glfwTerminate();
    return 0;
}
```

Task 2:

Part 1 full code:

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <vector>
#include <cmath>
#include <cstdio>
#include <cstdlib>

static const char* VS = R"(#version 330 core
layout(location=0) in vec2 vPosition;
layout(location=1) in vec3 vColor;
out vec3 color;
void main(){
    color = vColor;
    gl_Position = vec4(vPosition, 0.0, 1.0);
})";

static const char* FS = R"(#version 330 core
in vec3 color;
out vec4 FragColor;
void main(){
    FragColor = vec4(color, 1.0);
})";

static void framebuffer_size_callback(GLFWwindow*, int w, int h){
glViewport(0,0,w,h); }
```

```
static void check(bool ok, const char* msg){ if(!ok){
std::fprintf(stderr,"[!] %s\n", msg); std::exit(1);} }

static GLuint makeProgram(const char* vs, const char* fs){
    GLuint v = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(v, 1, &vs, nullptr); glCompileShader(v);
    GLint ok=0; glGetShaderiv(v, GL_COMPILE_STATUS, &ok);
    if(!ok){ char log[1024]; glGetShaderInfoLog(v,1024,nullptr,log);
std::fprintf(stderr,"VS error:\n%s\n",log); std::exit(1); }

    GLuint f = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(f, 1, &fs, nullptr); glCompileShader(f);
    glGetShaderiv(f, GL_COMPILE_STATUS, &ok);
    if(!ok){ char log[1024]; glGetShaderInfoLog(f,1024,nullptr,log);
std::fprintf(stderr,"FS error:\n%s\n",log); std::exit(1); }

    GLuint p = glCreateProgram(); glAttachShader(p,v); glAttachShader(p,f);
glLinkProgram(p);
    glGetProgramiv(p, GL_LINK_STATUS, &ok);
    if(!ok){ char log[1024]; glGetProgramInfoLog(p,1024,nullptr,log);
std::fprintf(stderr,"Link error:\n%s\n",log); std::exit(1); }
    glDeleteShader(v); glDeleteShader(f);
    return p;
}


static std::vector<float> makeCircleFan(float cx, float cy, float r, float
yScale,
                                        int segments, float redPhase){
    std::vector<float> v; v.reserve((segments+2)*5);
    v.insert(v.end(), {cx, cy, 0.18f, 0.f, 0.f});
    const float TAU = 6.28318530718f;
    for(int i=0;i<=segments;++i){
        float a = (float)i/segments * TAU;
        float x = cx + r * std::cos(a);
        float y = cy + (r * yScale) * std::sin(a);
        float red = 0.35f + 0.65f * (0.5f*(1.0f+std::cos(a+redPhase)));
        v.insert(v.end(), {x, y, red, 0.f, 0.f});
    }
    return v;
}


static std::vector<float> makeCircleFanSolid(float cx, float cy, float r,
float yScale,
                                             int segments, float R, float G,
float B){
    std::vector<float> v; v.reserve((segments+2)*5);
    v.insert(v.end(), {cx, cy, R, G, B});
    const float TAU = 6.28318530718f;
    for(int i=0;i<=segments;++i){
        float a = (float)i/segments * TAU;
        float x = cx + r * std::cos(a);
        float y = cy + (r * yScale) * std::sin(a);
        v.insert(v.end(), {x, y, R, G, B});
    }
    return v;
}

static std::vector<float> makeSquareByAngleFilled(float cx, float cy, float
radius, float gray){
    float c[4][2];
    const float start = 3.1415926535f/4.0f; // π/4
    for(int i=0;i<4;++i){
```

```
        float a = start + i*(3.1415926535f/2.0f);
        c[i][0] = cx + radius * std::cos(a);
        c[i][1] = cy + radius * std::sin(a);
    }

    std::vector<float> v = {
        c[2][0], c[2][1], gray,gray,gray, // BL
        c[3][0], c[3][1], gray,gray,gray, // BR
        c[1][0], c[1][1], gray,gray,gray, // TL
        c[0][0], c[0][1], gray,gray,gray  // TR
    };
    return v;
}

int main(){
    check(glfwInit(), "glfwInit failed");
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR,3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR,3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    GLFWwindow* win = glfwCreateWindow(500,500,"Task 2 — Scene", nullptr,
nullptr);
    check(win!=nullptr, "glfwCreateWindow failed");
    glfwMakeContextCurrent(win);
    glfwSetFramebufferSizeCallback(win, framebuffer_size_callback);
    check(gladLoadGLLoader((GLADloadproc)glfwGetProcAddress),
"gladLoadGLLoader failed");
    glViewport(0,0,500,500);

    GLuint prog = makeProgram(VS,FS);

    //RGB-triangle
    float tri[] = {
         0.00f,  0.65f,    1.f,0.f,0.f,
        -0.25f,  0.25f,    0.f,1.f,0.f,
         0.25f,  0.25f,    0.f,0.f,1.f
    };


    auto leftEllipse  = makeCircleFanSolid(-0.55f, 0.60f, 0.17f, 0.60f, 96,
1.0f, 0.0f, 0.0f);
    auto rightCircle  = makeCircleFan(      0.55f, 0.60f, 0.17f, 1.00f, 96,
1.2f);

    const float cx = 0.f, cy = -0.35f;
    auto sq1_white = makeSquareByAngleFilled(cx, cy, 0.57f, 1.00f); // outer
white (slightly thinner)
    auto gap1_blk  = makeSquareByAngleFilled(cx, cy, 0.47f, 0.00f); // black
gap
    auto sq2_white = makeSquareByAngleFilled(cx, cy, 0.36f, 1.00f); // medium
white
    auto gap2_blk  = makeSquareByAngleFilled(cx, cy, 0.26f, 0.00f); // black
gap
    auto sq3_white = makeSquareByAngleFilled(cx, cy, 0.165f,1.00f); // inner
white
    auto inner_blk = makeSquareByAngleFilled(cx, cy, 0.095f,0.00f); // black
center

    GLuint vao[9], vbo[9];
    glGenVertexArrays(9, vao);
    glGenBuffers(9, vbo);

    auto upload = [&](int i, const float* data, size_t floats){
        glBindVertexArray(vao[i]);
```

```
        glBindBuffer(GL_ARRAY_BUFFER, vbo[i]);
        glBufferData(GL_ARRAY_BUFFER, floats*sizeof(float), data,
GL_STATIC_DRAW);
        glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 5*sizeof(float),
(void*)0);
        glEnableVertexAttribArray(0); // vPosition
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 5*sizeof(float),
(void*)(2*sizeof(float)));
        glEnableVertexAttribArray(1); // vColor
    };

    upload(0, tri,             sizeof(tri)/sizeof(float));
    upload(1, leftEllipse.data(), leftEllipse.size());
    upload(2, rightCircle.data(), rightCircle.size());
    upload(3, sq1_white.data(),   sq1_white.size());
    upload(4, gap1_blk.data(),    gap1_blk.size());
    upload(5, sq2_white.data(),   sq2_white.size());
    upload(6, gap2_blk.data(),    gap2_blk.size());
    upload(7, sq3_white.data(),   sq3_white.size());
    upload(8, inner_blk.data(),   inner_blk.size());

    while(!glfwWindowShouldClose(win)){
        glClearColor(0.0f,0.0f,0.0f,1.0f);
        glClear(GL_COLOR_BUFFER_BIT);
        glUseProgram(prog);


        glBindVertexArray(vao[1]); glDrawArrays(GL_TRIANGLE_FAN, 0,
(GLsizei)(leftEllipse.size()/5));
        glBindVertexArray(vao[2]); glDrawArrays(GL_TRIANGLE_FAN, 0,
(GLsizei)(rightCircle.size()/5));
        glBindVertexArray(vao[0]); glDrawArrays(GL_TRIANGLES,    0, 3);


        for(int i=3;i<=8;++i){
            glBindVertexArray(vao[i]);
            glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
        }

        glfwSwapBuffers(win);
        glfwPollEvents();
    }

    glfwTerminate();
    return 0;
}
```

Part 2 full code:

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <vector>
#include <cmath>
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif
#include <cstdio>
#include <cstdlib>

static const char* VS = R"(#version 330 core
layout(location=0) in vec2 vPosition;
layout(location=1) in vec3 vColor;
```

```
out vec3 color;
void main(){
    color = vColor;
    gl_Position = vec4(vPosition, 0.0, 1.0);
})";

static const char* FS = R"(#version 330 core
in vec3 color;
out vec4 FragColor;
void main(){
    FragColor = vec4(color, 1.0);
})";

static void framebuffer_size_callback(GLFWwindow*, int w, int h){
glViewport(0,0,w,h); }
static void check(bool ok, const char* msg){ if(!ok){
std::fprintf(stderr,"[!] %s\n", msg); std::exit(1);} }

static GLuint makeProgram(const char* vs, const char* fs){
    GLuint v = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(v, 1, &vs, nullptr); glCompileShader(v);
    GLint ok=0; glGetShaderiv(v, GL_COMPILE_STATUS, &ok);
    if(!ok){ char log[1024]; glGetShaderInfoLog(v,1024,nullptr,log);
std::fprintf(stderr,"VS error:\n%s\n",log); std::exit(1); }

    GLuint f = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(f, 1, &fs, nullptr); glCompileShader(f);
    glGetShaderiv(f, GL_COMPILE_STATUS, &ok);
    if(!ok){ char log[1024]; glGetShaderInfoLog(f,1024,nullptr,log);
std::fprintf(stderr,"FS error:\n%s\n",log); std::exit(1); }

    GLuint p = glCreateProgram(); glAttachShader(p,v); glAttachShader(p,f);
glLinkProgram(p);
    glGetProgramiv(p, GL_LINK_STATUS, &ok);
    if(!ok){ char log[1024]; glGetProgramInfoLog(p,1024,nullptr,log);
std::fprintf(stderr,"Link error:\n%s\n",log); std::exit(1); }
    glDeleteShader(v); glDeleteShader(f);
    return p;
}


//shapes

// Star (5-point) as TRIANGLE_FAN
static std::vector<float> makeStar(float cx,float cy,float rOut,float rIn){
    std::vector<float> v;
    v.insert(v.end(),{cx,cy,1,1,0}); // center yellow
    for(int i=0;i<=10;i++){
        float a=(M_PI/2)+(i*36.0f*M_PI/180.0f); // 36° per step
        float r=(i%2==0)?rOut:rIn;
        float x=cx+r*cos(a), y=cy+r*sin(a);
        v.insert(v.end(),{x,y,1,0.5f,0}); // orange
    }
    return v;
}

// Ring as TRIANGLE_STRIP
static std::vector<float> makeRing(float cx,float cy,float rOut,float rIn,int
seg){
    std::vector<float> v;
    for(int i=0;i<=seg;i++){
        float a=i*2*M_PI/seg;
        float co=cos(a), si=sin(a);
        v.insert(v.end(),{cx+rOut*co,cy+rOut*si,0,1,1}); // outer cyan
```

```
        v.insert(v.end(),{cx+rIn*co, cy+rIn*si, 0,0.2f,0.5f}); // inner dark
    }
    return v;
}

// Hexagon as TRIANGLE_FAN
static std::vector<float> makeHexagon(float cx,float cy,float r){
    std::vector<float> v;
    v.insert(v.end(),{cx,cy,0.8f,0,0.8f}); // center purple
    for(int i=0;i<=6;i++){
        float a=M_PI/6+i*M_PI/3; // start 30°, step 60°
        float x=cx+r*cos(a), y=cy+r*sin(a);
        v.insert(v.end(),{x,y,0.3f+0.1f*i,0.1f*i,1.0f-0.1f*i}); // gradient
    }
    return v;
}

int main(){
    check(glfwInit(),"glfwInit failed");
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR,3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR,3);
    glfwWindowHint(GLFW_OPENGL_PROFILE,GLFW_OPENGL_CORE_PROFILE);
    GLFWwindow* win=glfwCreateWindow(500,500,"Part 2 Scene",nullptr,nullptr);
    check(win!=nullptr,"glfwCreateWindow failed");
    glfwMakeContextCurrent(win);
    glfwSetFramebufferSizeCallback(win,framebuffer_size_callback);

check(gladLoadGLLoader((GLADloadproc)glfwGetProcAddress),"gladLoadGLLoader
failed");

    GLuint prog=makeProgram(VS,FS);

    auto star=makeStar(-0.5f,0.5f,0.2f,0.08f);
    auto ring=makeRing(0.5f,0.5f,0.2f,0.14f,64);
    auto hex =makeHexagon(0.0f,-0.3f,0.3f);

    GLuint vao[3],vbo[3];
    glGenVertexArrays(3,vao); glGenBuffers(3,vbo);

    auto upload=[&](int i,const std::vector<float>&d){
        glBindVertexArray(vao[i]); glBindBuffer(GL_ARRAY_BUFFER,vbo[i]);

glBufferData(GL_ARRAY_BUFFER,d.size()*sizeof(float),d.data(),GL_STATIC_DRAW);

glVertexAttribPointer(0,2,GL_FLOAT,GL_FALSE,5*sizeof(float),(void*)0);
        glEnableVertexAttribArray(0);

glVertexAttribPointer(1,3,GL_FLOAT,GL_FALSE,5*sizeof(float),(void*)(2*sizeof(
float)));
        glEnableVertexAttribArray(1);
    };
    upload(0,star); upload(1,ring); upload(2,hex);
    while(!glfwWindowShouldClose(win)){
        glClearColor(0,0,0,1); glClear(GL_COLOR_BUFFER_BIT);
        glUseProgram(prog);
        glBindVertexArray(vao[0]);
glDrawArrays(GL_TRIANGLE_FAN,0,star.size()/5);
        glBindVertexArray(vao[1]);
glDrawArrays(GL_TRIANGLE_STRIP,0,ring.size()/5);
        glBindVertexArray(vao[2]);
glDrawArrays(GL_TRIANGLE_FAN,0,hex.size()/5);
        glfwSwapBuffers(win); glfwPollEvents();
    }
    glfwTerminate(); return 0;
```

```
}
```