src\for_print.cpp

```
// All imports

void framebuffer_size_callback(GLFWwindow* window, int width, int height);
void processInput(GLFWwindow* window);

// settings
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;

float translate_X = 0.0, translate_Y = 0.0, rotateAngle = 0.0, scale_X = 1.0, scale_Y = 1.0;
float translate_X_Cream = 0.0f, translate_Y_Cream = 0.0f, rotateAngle_Cream = 0.0f, scale_X_Cream = 1.0f, scale_Y_Cream = 1.0f;
float translate_X_Cone = 0.0f, translate_Y_Cone = 0.0f, rotateAngle_Cone = 0.0f, scale_X_Cone = 1.0f, scale_Y_Cone = 1.0f;

const char *vertexShaderSource = "#version 330 core\n"
    "layout (location = 0) in vec3 aPos;\n"
    "uniform mat4 transform;\n"
    "void main()\n"
    "{\n"
    "   gl_Position = transform * vec4(aPos, 1.0);\n"
    "}\0";

const char* fragmentShaderSource = "#version 330 core\n"
"out vec4 FragColor;\n"
"uniform vec3 color;\n"  // Define the color uniform
"void main()\n"
"{\n"
"   FragColor = vec4(color, 1.0f);\n"  // Use the color uniform here
"}\n\0";

float** readPoints(const std::string& filename, int& numLists, int*& vertexCounts);

int initGlfw(GLFWwindow*& window){
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    // glfw window creation
    window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "LabOne", NULL, NULL);
    if (window == NULL){ cout << "Failed to create GLFW window" << endl; glfwTerminate();
return -1; }

    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

    // glad: load all OpenGL function pointers
```

```
46      if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) { cout << "Failed to initialize
    GLAD" << endl; return -1; }
47      return 0;
48  }
49
50  int initVertexShader(unsigned int &vertexShader){
51      // vertex shader
52      glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
53      glCompileShader(vertexShader);
54      // check for shader compile errors
55      int success;
56      char infoLog[512];
57      glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
58      if (!success) {
59          glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
60          cout << "VERTEX::COMPILATION_FAILED\n" << infoLog << endl;
61          return -1;
62      }
63      return 0;
64  }
65
66  int initFragmentShader(unsigned int &fragmentShader){
67      int success;
68      char infoLog[512];
69
70      glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
71      glCompileShader(fragmentShader);
72      // check for shader compile errors
73      glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
74      if (!success){
75          glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
76          cout << "FRAGMENT::COMPILATION_FAILED\n" << infoLog << endl;
77          return -1;
78      }
79      return 0;
80  }
81
82  int initLinkShader(unsigned int &shaderProgram, unsigned int vertexShader, unsigned int
    fragmentShader){
83      glAttachShader(shaderProgram, vertexShader);
84      glAttachShader(shaderProgram, fragmentShader);
85      glLinkProgram(shaderProgram);
86
87      int success;
88      char infoLog[512];
89      // check for linking errors
90      glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
91      if (!success) {
92          glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
93          cout << "LINKING_FAILED\n" << infoLog << endl;
```

```
        return -1;
    }
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
    return 0;
}

void initBinding(unsigned int &VAO, unsigned int &VBO){
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
}

int main()
{
    GLFWwindow* window = nullptr;
    if( initGlfw(window) ) return -1;

    unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
    if( initVertexShader(vertexShader) ) return -1;

    unsigned int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    if( initFragmentShader(fragmentShader) ) return -1;

    // link shaders
    unsigned int shaderProgram = glCreateProgram();
    if( initLinkShader(shaderProgram, vertexShader, fragmentShader) ) return -1;

    int numLists;
    int* vertexCounts;
    float** vertices = readPoints(...);

    // Set up the VAO and VBO
    unsigned int VBO, VAO;
    initBinding(VAO, VBO);

    set<int> fillSet = {0,1,2,3,4};
    set<int> creamSet = {1,2,3,33};
    float colors[][3] = {
        {0.12, 0.12, 0.12}, // cone
        {40/255.0, 0/255.0, 189/255.0}, // blue
        {183/255.0, 199/255.0, 6/255.0}, // top yellow
        {26/255.0, 161/255.0, 5/255.0}, // green
        {219/255.0, 44/255.0, 173/255.0}, // pink
        {204/255.0, 112/255.0, 4/255.0} // muddy
    };

    // Render loop
    while (!glfwWindowShouldClose(window)){
```

```
144         processInput(window);
145 
146         // Render
147         glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
148         glClear(GL_COLOR_BUFFER_BIT);
149 
150         // Use the shader program
151         glm::mat4 translationMatrix, rotationMatrix, scaleMatrix, modelMatrix;
152         glm::mat4 identityMatrix = glm::mat4(1.0f);
153         translationMatrix = glm::translate(identityMatrix, glm::vec3(translate_X, translate_Y ,
    0.0f));
154         rotationMatrix = glm::rotate(identityMatrix, glm::radians(rotateAngle), glm::vec3(0.0f,
    0.0f, 1.0f));
155         scaleMatrix = glm::scale(identityMatrix, glm::vec3(scale_X, scale_Y, 1.0f));
156         modelMatrix = translationMatrix * rotationMatrix * scaleMatrix;
157 
158         glm::mat4 translationMatrixCream, rotationMatrixCream,
    scaleMatrixCream,modelMatrixCream;
159 
160         translationMatrixCream = glm::translate(modelMatrix, glm::vec3(translate_X_Cream,
    translate_Y_Cream , 0.0f));
161         rotationMatrixCream = glm::rotate(modelMatrix, glm::radians(rotateAngle_Cream),
    glm::vec3(0.0f, 0.0f, 1.0f));
162         scaleMatrixCream = glm::scale(modelMatrix, glm::vec3(scale_X_Cream, scale_Y_Cream,
    1.0f));
163         modelMatrixCream = translationMatrixCream * rotationMatrixCream * scaleMatrixCream;
164 
165         glm::mat4 translationMatrixCone, rotationMatrixCone, scaleMatrixCone, modelMatrixCone;
166 
167         translationMatrixCone = glm::translate(modelMatrix, glm::vec3(translate_X_Cone,
    translate_Y_Cone , 0.0f));
168         rotationMatrixCone = glm::rotate(modelMatrix, glm::radians(rotateAngle_Cone),
    glm::vec3(0.0f, 0.0f, 1.0f));
169         scaleMatrixCone = glm::scale(modelMatrix, glm::vec3(scale_X_Cone, scale_Y_Cone, 1.0f));
170         modelMatrixCone = translationMatrixCone * rotationMatrixCone * scaleMatrixCone;
171 
172         // get matrix's uniform location and set matrix
173         glUseProgram(shaderProgram);
174         unsigned int transformLoc = glGetUniformLocation(shaderProgram, "transform");
175 
176         glBindVertexArray(VAO);
177         int colorLocation = glGetUniformLocation(shaderProgram, "color");
178 
179         // Draw each sublist separately
180         for (int i = 0; i < numLists; ++i) {
181 
182             if(creamSet.find(i) != creamSet.end()){
183                 glUniformMatrix4fv(transformLoc, 1, GL_FALSE,
    glm::value_ptr(modelMatrixCream));
184             }
185             else{
```

```
                glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(modelMatrixCone));
            }

            int index = (i >= 4) ? 5 : i;

            glUniform3f(colorLocation, colors[index][0], colors[index][1], colors[index][2]);
            if(i == numLists-1){
                glUniform3f(colorLocation, colors[4][0], colors[4][1], colors[4][2]);
            }

            glBindBuffer(GL_ARRAY_BUFFER, VBO);
            glBufferData(GL_ARRAY_BUFFER, vertexCounts[i] * sizeof(float), vertices[i],
GL_STATIC_DRAW);
            glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
            glEnableVertexAttribArray(0);

            if(fillSet.find(i) != fillSet.end() || i >= 5)
                glDrawArrays(GL_TRIANGLE_FAN, 0, vertexCounts[i] / 3);
            else
                glDrawArrays(GL_LINE_STRIP, 0, vertexCounts[i] / 3);
        }

    }
    // clean up
    return 0;
}

std::unordered_map<int, bool> keyState;
bool isKeyPressedOnce(GLFWwindow* window, int key) {
    if (glfwGetKey(window, key) == GLFW_PRESS) {
        if (!keyState[key]) {
            keyState[key] = true;
            return true;
        }
    } else {
        keyState[key] = false;
    }
    return false;
}

void processInput(GLFWwindow* window)
{
    float unit = 0.05;
    float scaleFactor = 0.2f;
    float rotate_angle = 30;

    // for whole object
    if ( glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

```

```
    else if( isKeyPressedOnce(window, GLFW_KEY_W) == GLFW_PRESS ) // translate-x negative
        translate_X -= unit;
    else if( isKeyPressedOnce(window, GLFW_KEY_Q) == GLFW_PRESS ) // translate-x positive
        translate_X += unit;
    else if( isKeyPressedOnce(window, GLFW_KEY_E) == GLFW_PRESS ) // translate-y negative
        translate_Y -= unit;
    else if( isKeyPressedOnce(window, GLFW_KEY_R) == GLFW_PRESS ) // translate-y positive
        translate_Y += unit;
    else if( isKeyPressedOnce(window, GLFW_KEY_T) == GLFW_PRESS ){ // scale up
        scale_X += scaleFactor;
        scale_Y += scaleFactor;
    }
    else if( isKeyPressedOnce(window, GLFW_KEY_Y) == GLFW_PRESS ){ // scale down
        scale_X -= scaleFactor;
        scale_Y -= scaleFactor;
    }
    else if( isKeyPressedOnce(window, GLFW_KEY_U) == GLFW_PRESS ) // rotate clockwise
        rotateAngle += rotate_angle * 3.14/180;
    else if( isKeyPressedOnce(window, GLFW_KEY_I) == GLFW_PRESS ) // rotate counter-clockwise
        rotateAngle -= rotate_angle * 3.14/180;

    // similar for cream and cone
    // ...
}


void framebuffer_size_callback(GLFWwindow* window, int width, int height){
    glViewport(0, 0, width, height);
}

```