Making Tetris with OpenGL

Presented by:

- Alicia Sudlerd

28.04.2023 - Computer Graphics Seminar (MTAT.03.305) - University of Tartu

Agenda

Goal:

1. Tetris making journey

2. Rendering and shader pipeline

Categories:

- OpenGL Installation
- Shader Pipeline
- Screen Projection
- Buffer(s)
- Rendering
- Game Logic

Reflections

- Lesson Learned
- What NOT to do

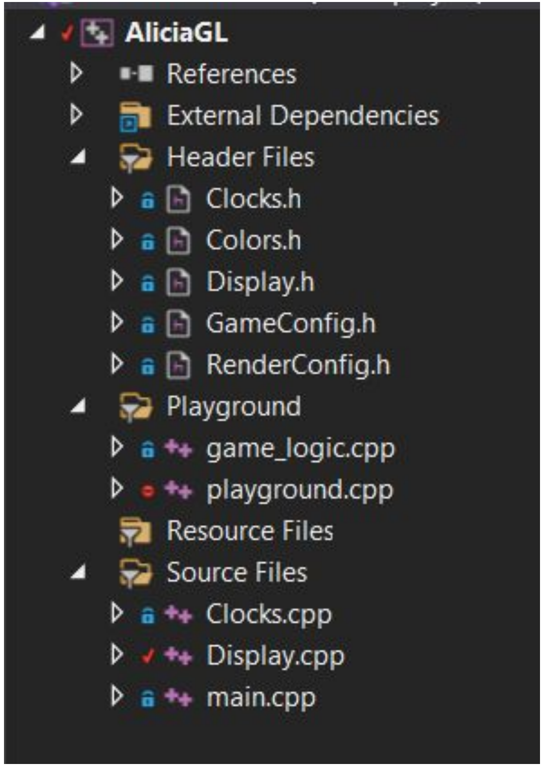
Prerequisites

OpenGL (3.3.8) Installation

- GLFW
- GLEW
- GLM
- include these libraries in the project's include and library paths. (manually)

DEMO

Project Structure



```
AliciaGL
    References
    External Dependencies
    Header Files
        Clocks.h
        Colors.h
        Display.h
        GameConfig.h
        RenderConfig.h
    Playground
        game_logic.cpp
        playground.cpp
    Resource Files
    Source Files
        Clocks.cpp
        Display.cpp
        main.cpp
```

- Main
- RenderConfig
- GameConfig
- Color
- Clock
- Display

Prerequisites

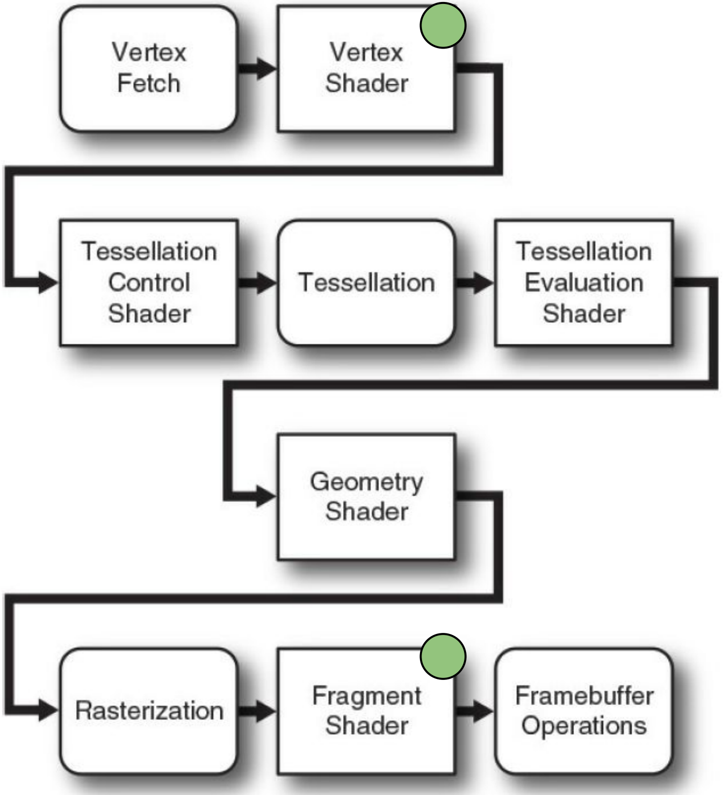
OpenGL (3.3.8) Installation

- GLFW
- GLEW
- GLM
- include these libraries in the project's include and library paths. (manually)

SHADER(S)

OpenGL Shader Pipeline

Source of a diagram: OpenGL SuperBibles CP1



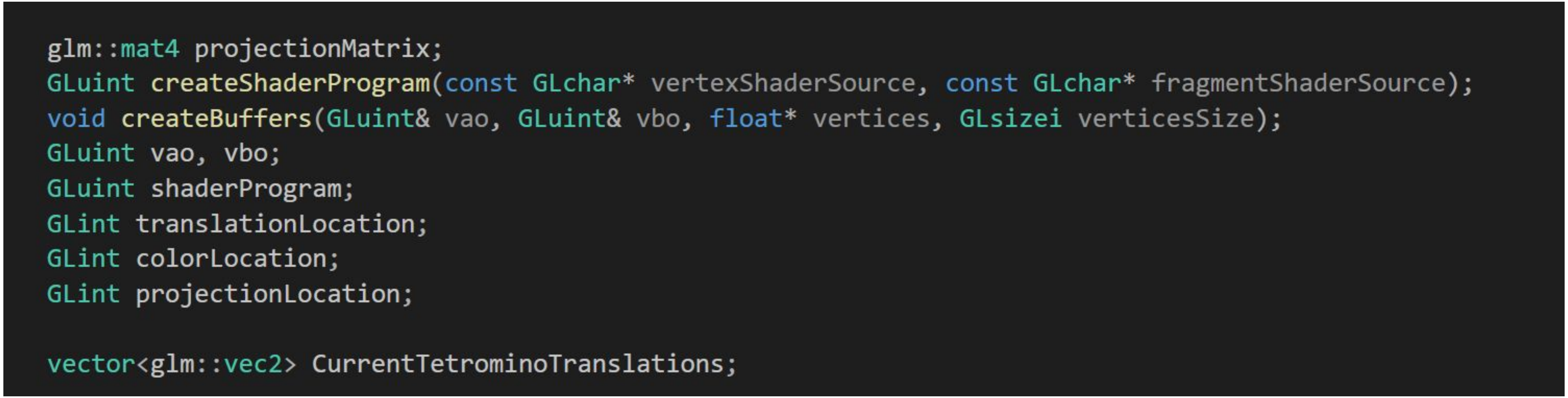
Vertex Shader

- Process vertices of primitives
- Taking care of translation, rotation, scaling, …
- We use uTranslation & aPos for translation and projectionMatrix to set the screen resolution

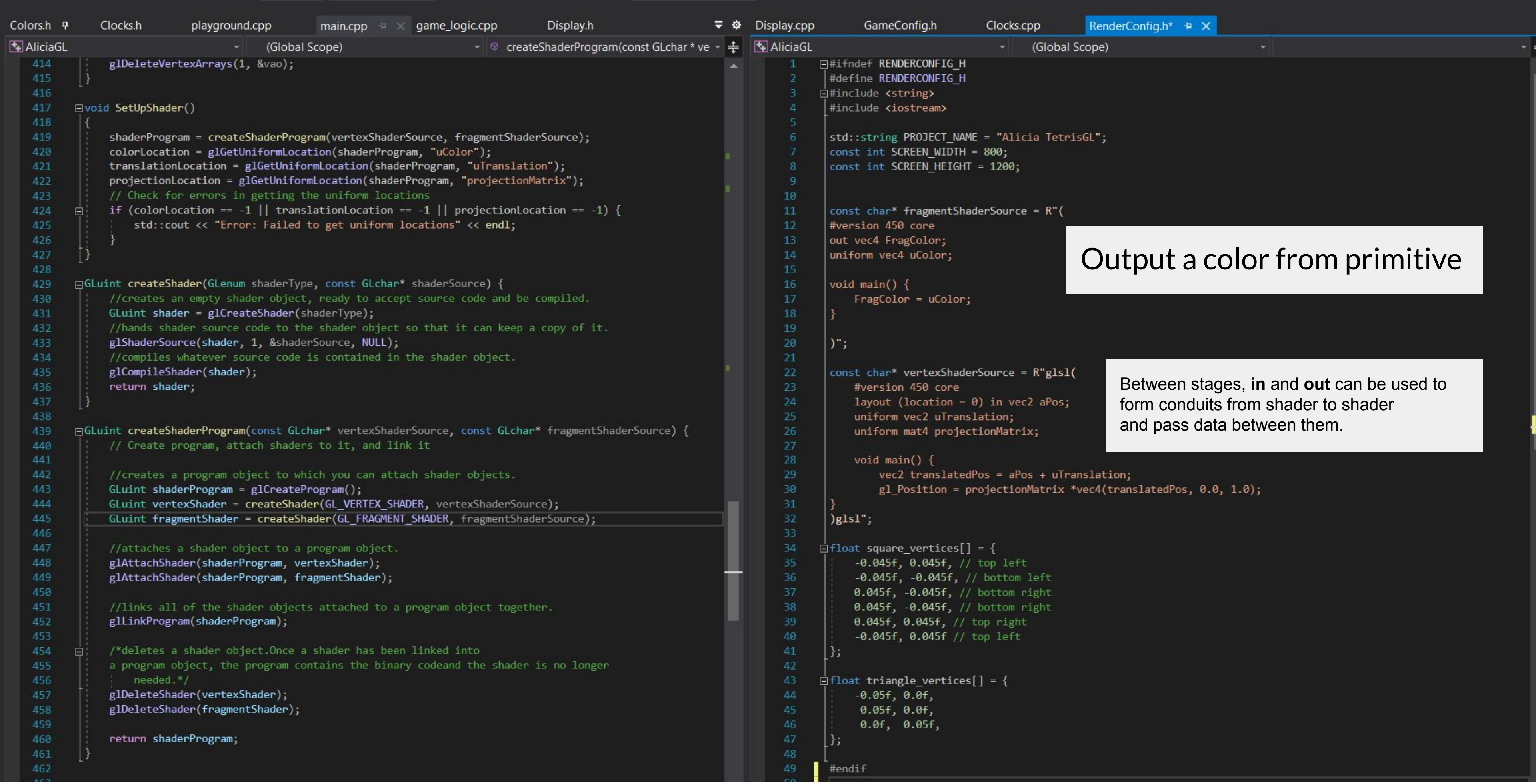
Fragment Shader

- Process colors and properties of each pixel
- In this project, we use a uniform variable uColor, which is used to set the color of the primitive.

Core variables



```
glm::mat4 projectionMatrix;
GLuint createShaderProgram(const GLchar* vertexShaderSource, const GLchar* fragmentShaderSource);
void createBuffers(GLuint& vao, GLuint& vbo, float* vertices, GLsizei verticesSize);
GLuint vao, vbo;
GLuint shaderProgram;
GLint translationLocation;
GLint colorLocation;
GLint projectionLocation;

vector<glm::vec2> CurrentTetrominoTranslations;
```



```cpp
    glDeleteVertexArrays(1, &vao);
}

void SetUpShader()
{
    shaderProgram = createShaderProgram(vertexShaderSource, fragmentShaderSource);
    colorLocation = glGetUniformLocation(shaderProgram, "uColor");
    translationLocation = glGetUniformLocation(shaderProgram, "uTranslation");
    projectionLocation = glGetUniformLocation(shaderProgram, "projectionMatrix");
    // Check for errors in getting the uniform locations
    if (colorLocation == -1 || translationLocation == -1 || projectionLocation == -1) {
        std::cout << "Error: Failed to get uniform locations" << endl;
    }
}

GLuint createShader(GLenum shaderType, const GLchar* shaderSource) {
    //creates an empty shader object, ready to accept source code and be compiled.
    GLuint shader = glCreateShader(shaderType);
    //hands shader source code to the shader object so that it can keep a copy of it.
    glShaderSource(shader, 1, &shaderSource, NULL);
    //compiles whatever source code is contained in the shader object.
    glCompileShader(shader);
    return shader;
}

GLuint createShaderProgram(const GLchar* vertexShaderSource, const GLchar* fragmentShaderSource) {
    // Create program, attach shaders to it, and link it

    //creates a program object to which you can attach shader objects.
    GLuint shaderProgram = glCreateProgram();
    GLuint vertexShader = createShader(GL_VERTEX_SHADER, vertexShaderSource);
    GLuint fragmentShader = createShader(GL_FRAGMENT_SHADER, fragmentShaderSource);

    //attaches a shader object to a program object.
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);

    //links all of the shader objects attached to a program object together.
    glLinkProgram(shaderProgram);

    /*deletes a shader object.Once a shader has been linked into
    a program object, the program contains the binary codeand the shader is no longer
        needed.*/
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);

    return shaderProgram;
}
```

```cpp
#ifndef RENDERCONFIG_H
#define RENDERCONFIG_H
#include <string>
#include <iostream>

std::string PROJECT_NAME = "Alicia TetrisGL";
const int SCREEN_WIDTH = 800;
const int SCREEN_HEIGHT = 1200;


const char* fragmentShaderSource = R"(
#version 450 core
out vec4 FragColor;
uniform vec4 uColor;

void main() {
    FragColor = uColor;
}

)";

const char* vertexShaderSource = R"glsl(
    #version 450 core
    layout (location = 0) in vec2 aPos;
    uniform vec2 uTranslation;
    uniform mat4 projectionMatrix;

    void main() {
        vec2 translatedPos = aPos + uTranslation;
        gl_Position = projectionMatrix *vec4(translatedPos, 0.0, 1.0);
}
)glsl";

float square_vertices[] = {
    -0.045f, 0.045f, // top left
    -0.045f, -0.045f, // bottom left
    0.045f, -0.045f, // bottom right
    0.045f, -0.045f, // bottom right
    0.045f, 0.045f, // top right
    -0.045f, 0.045f // top left
};

float triangle_vertices[] = {
    -0.05f, 0.0f,
     0.05f, 0.0f,
     0.0f,  0.05f,
};

#endif
```

Output a color from primitive

Between stages, **in** and **out** can be used to form conduits from shader to shader and pass data between them.

Source of descriptions: OpenGL SuperBibles

RENDERING

Rendering Setup

Projection:

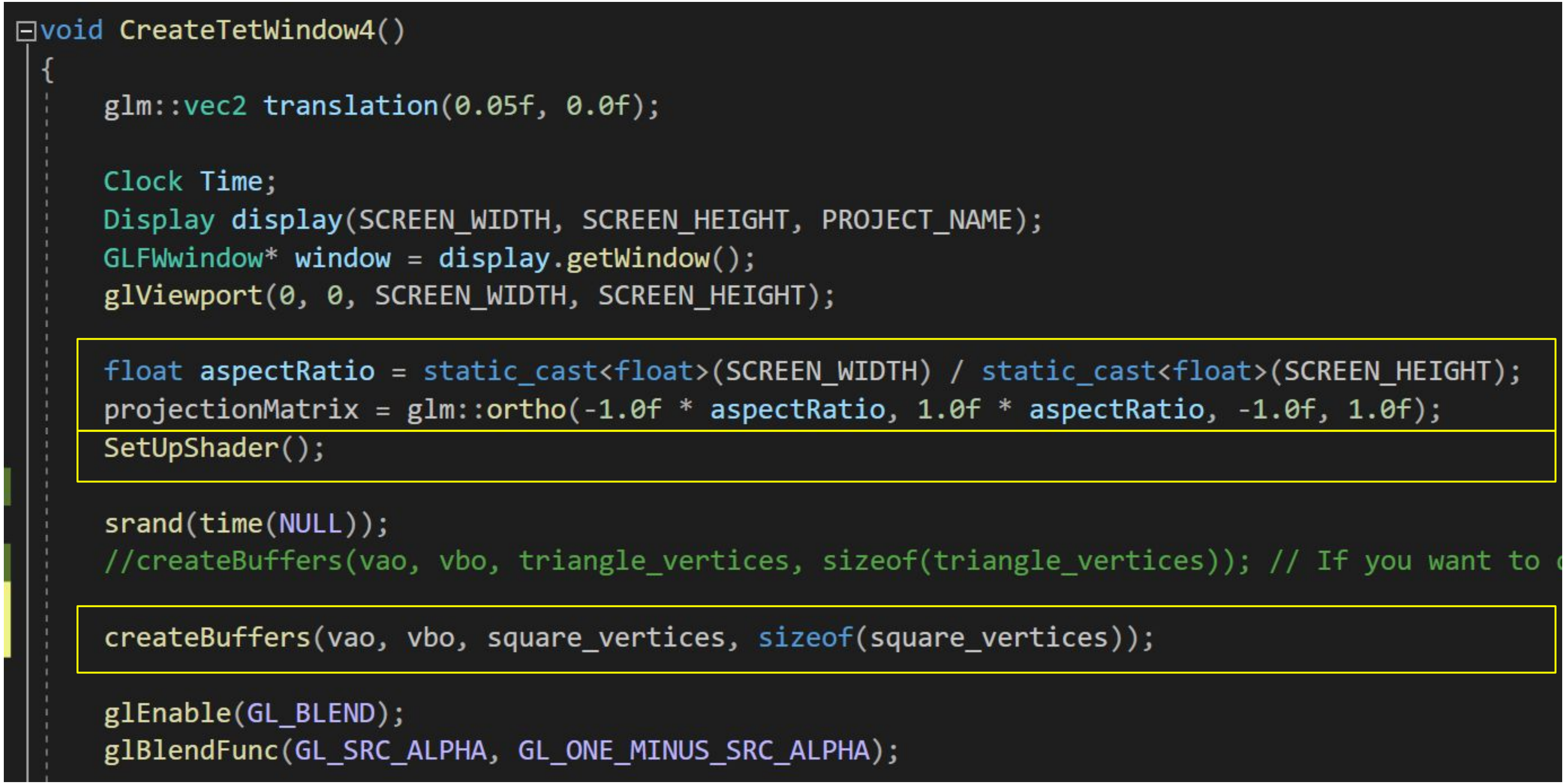
1. Orthographic

Buffers:

- Square
- Triangle(but not used)

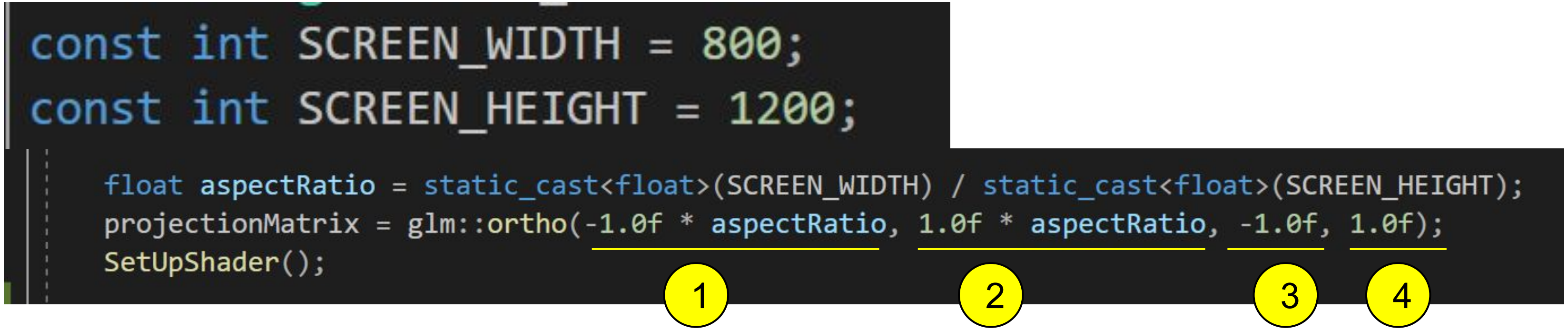
Colors

- Lesson Learned
- What NOT to do



```
void CreateTetWindow4()
{
    glm::vec2 translation(0.05f, 0.0f);

    Clock Time;
    Display display(SCREEN_WIDTH, SCREEN_HEIGHT, PROJECT_NAME);
    GLFWwindow* window = display.getWindow();
    glViewport(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT);

    float aspectRatio = static_cast<float>(SCREEN_WIDTH) / static_cast<float>(SCREEN_HEIGHT);
    projectionMatrix = glm::ortho(-1.0f * aspectRatio, 1.0f * aspectRatio, -1.0f, 1.0f);
    SetUpShader();

    srand(time(NULL));
    //createBuffers(vao, vbo, triangle_vertices, sizeof(triangle_vertices)); // If you want to c

    createBuffers(vao, vbo, square_vertices, sizeof(square_vertices));

    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

glm::ortho()



```
const int SCREEN_WIDTH = 800;
const int SCREEN_HEIGHT = 1200;
```

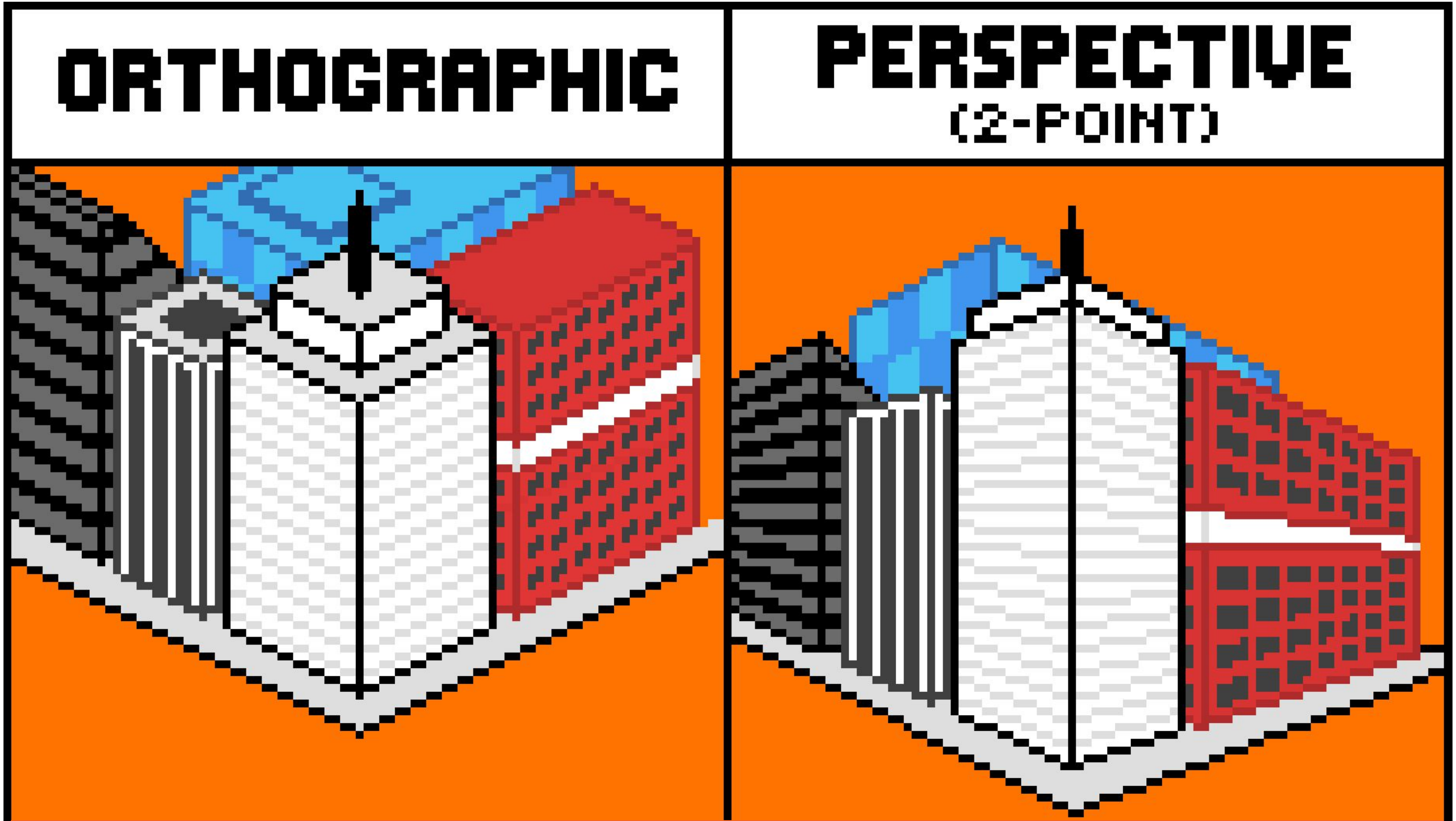
```
    float aspectRatio = static_cast<float>(SCREEN_WIDTH) / static_cast<float>(SCREEN_HEIGHT);
    projectionMatrix = glm::ortho(-1.0f * aspectRatio, 1.0f * aspectRatio, -1.0f, 1.0f);
    SetUpShader();
```

1 The left clipping plane of the projection

2 The right clipping plane of the projection

3 The bottom clipping plane of the projection

4 The top clipping plane of the projection



Source: https://grid-paint.com/images/details/5353898643554304

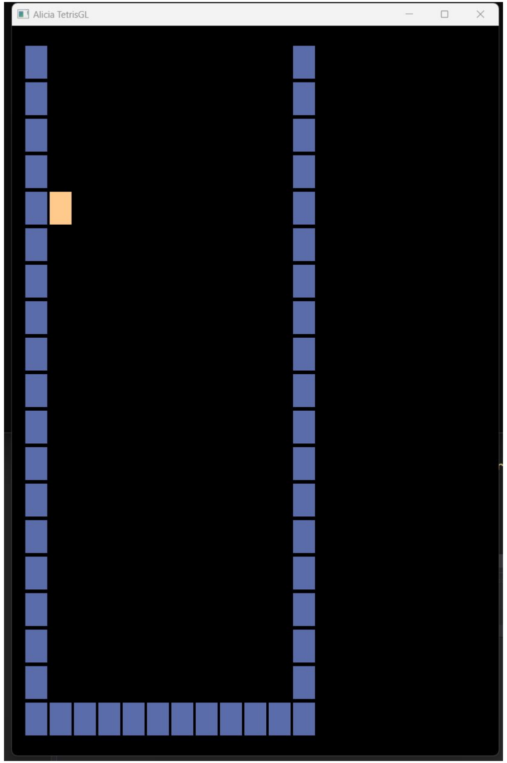
What would happen without setting the projection?

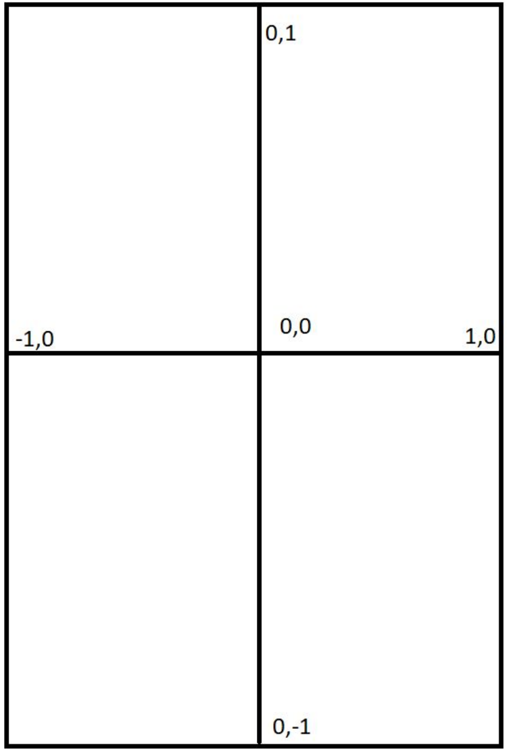
The coordinate system will be stretched.
And every object on the screen also gets stretched…

It can be roughly solved by

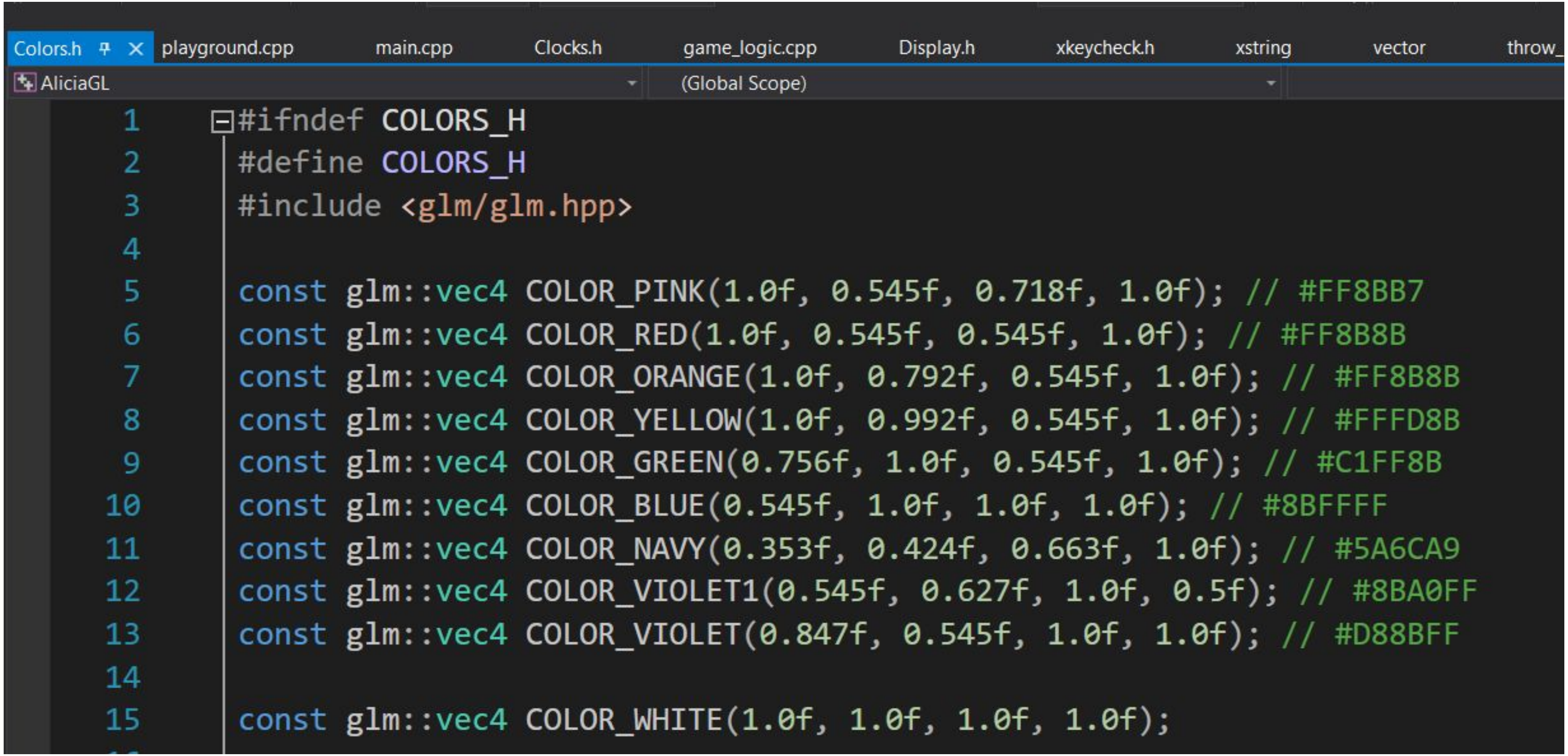
```
glViewport(0, 0, SCREEN_WIDTH, SCREEN_WIDTH);
```

(make the screen width = screen height)





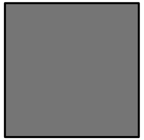
Colors



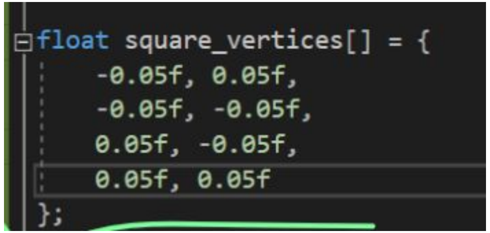
```cpp
#ifndef COLORS_H
#define COLORS_H
#include <glm/glm.hpp>

const glm::vec4 COLOR_PINK(1.0f, 0.545f, 0.718f, 1.0f); // #FF8BB7
const glm::vec4 COLOR_RED(1.0f, 0.545f, 0.545f, 1.0f); // #FF8B8B
const glm::vec4 COLOR_ORANGE(1.0f, 0.792f, 0.545f, 1.0f); // #FF8B8B
const glm::vec4 COLOR_YELLOW(1.0f, 0.992f, 0.545f, 1.0f); // #FFFD8B
const glm::vec4 COLOR_GREEN(0.756f, 1.0f, 0.545f, 1.0f); // #C1FF8B
const glm::vec4 COLOR_BLUE(0.545f, 1.0f, 1.0f, 1.0f); // #8BFFFF
const glm::vec4 COLOR_NAVY(0.353f, 0.424f, 0.663f, 1.0f); // #5A6CA9
const glm::vec4 COLOR_VIOLET1(0.545f, 0.627f, 1.0f, 0.5f); // #8BA0FF
const glm::vec4 COLOR_VIOLET(0.847f, 0.545f, 1.0f, 1.0f); // #D88BFF

const glm::vec4 COLOR_WHITE(1.0f, 1.0f, 1.0f, 1.0f);
```

glDrawArrays(<GL_x>, 0, n);

- GL_QUADS

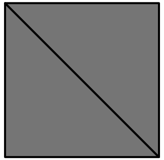


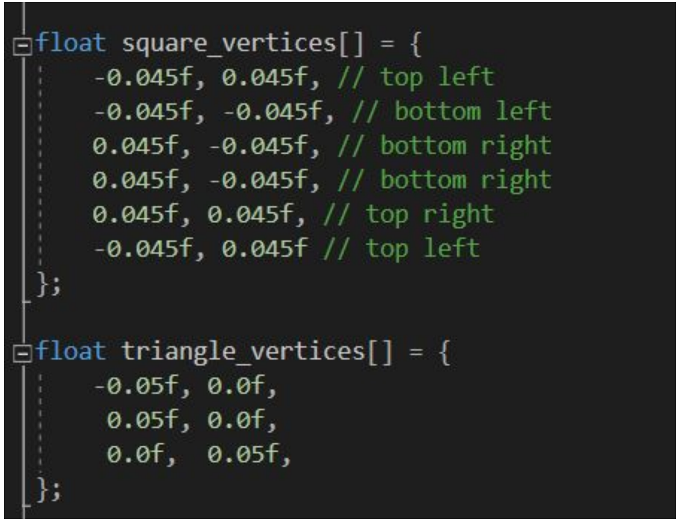
But obsolete…



```
float square_vertices[] = {
    -0.05f, 0.05f,
    -0.05f, -0.05f,
    0.05f, -0.05f,
    0.05f, 0.05f
};
```

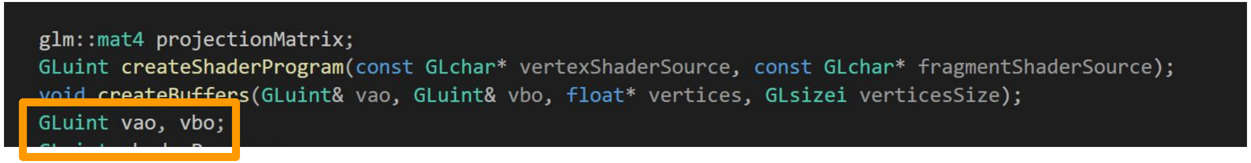
- GL_TRIANGLES



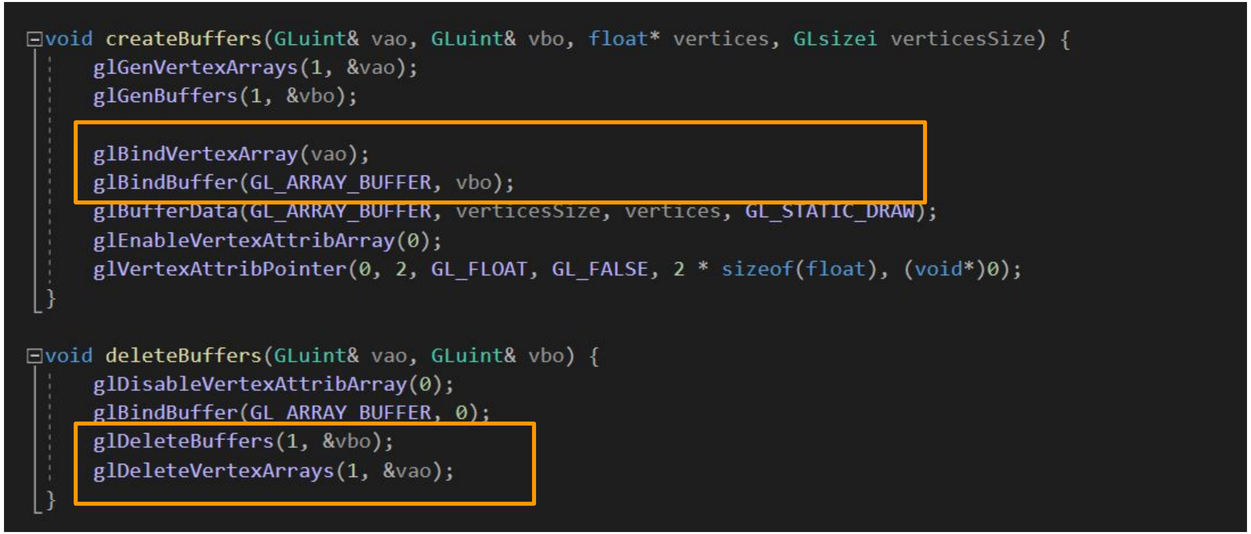


```
float square_vertices[] = {
    -0.045f, 0.045f, // top left
    -0.045f, -0.045f, // bottom left
    0.045f, -0.045f, // bottom right
    0.045f, -0.045f, // bottom right
    0.045f, 0.045f, // top right
    -0.045f, 0.045f // top left
};

float triangle_vertices[] = {
    -0.05f, 0.0f,
     0.05f, 0.0f,
     0.0f,  0.05f,
};
```

Create Buffer(s)



```
glm::mat4 projectionMatrix;
GLuint createShaderProgram(const GLchar* vertexShaderSource, const GLchar* fragmentShaderSource);
void createBuffers(GLuint& vao, GLuint& vbo, float* vertices, GLsizei verticesSize);
GLuint vao, vbo;
```



```
void createBuffers(GLuint& vao, GLuint& vbo, float* vertices, GLsizei verticesSize) {
    glGenVertexArrays(1, &vao);
    glGenBuffers(1, &vbo);

    glBindVertexArray(vao);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, verticesSize, vertices, GL_STATIC_DRAW);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)0);
}

void deleteBuffers(GLuint& vao, GLuint& vbo) {
    glDisableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glDeleteBuffers(1, &vbo);
    glDeleteVertexArrays(1, &vao);
}
```

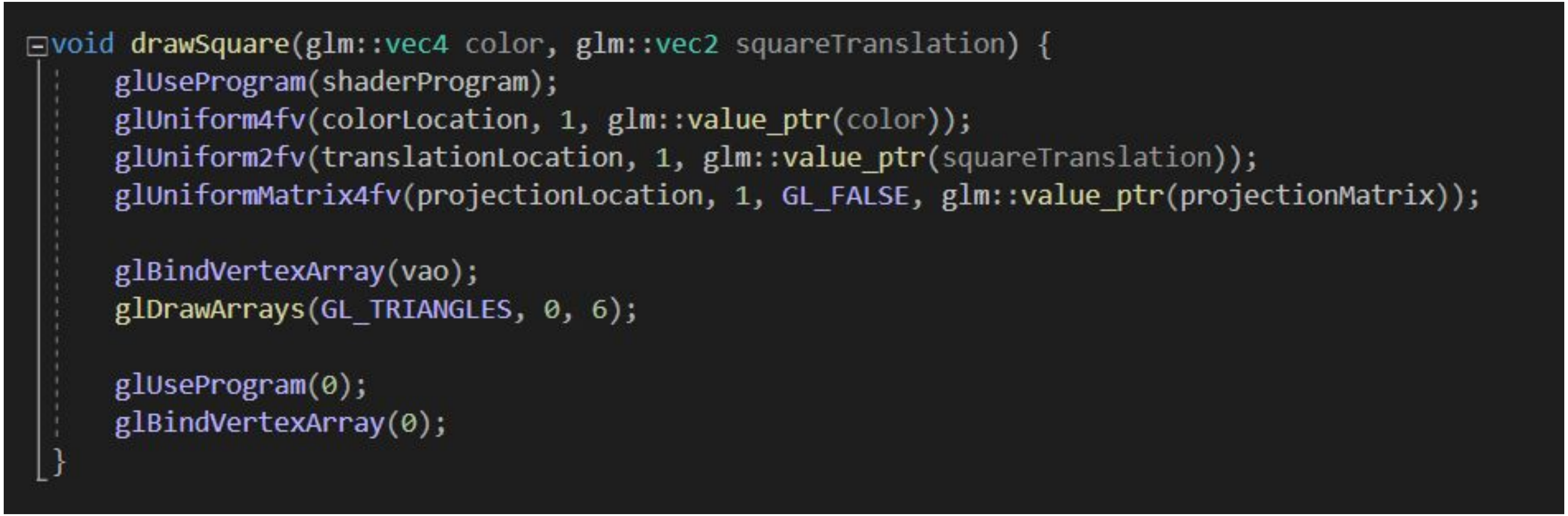
Create Buffer(s)

- **vao (Vertex Array Object)**
  - stores the information about how the data is arranged
  - (e.g. positions, colors, texture coordinates)

- **vbo (Vertex Buffer Object)**
  - stores the actual data itself.
  - In this project, we use it to store 6 vertices of the square.

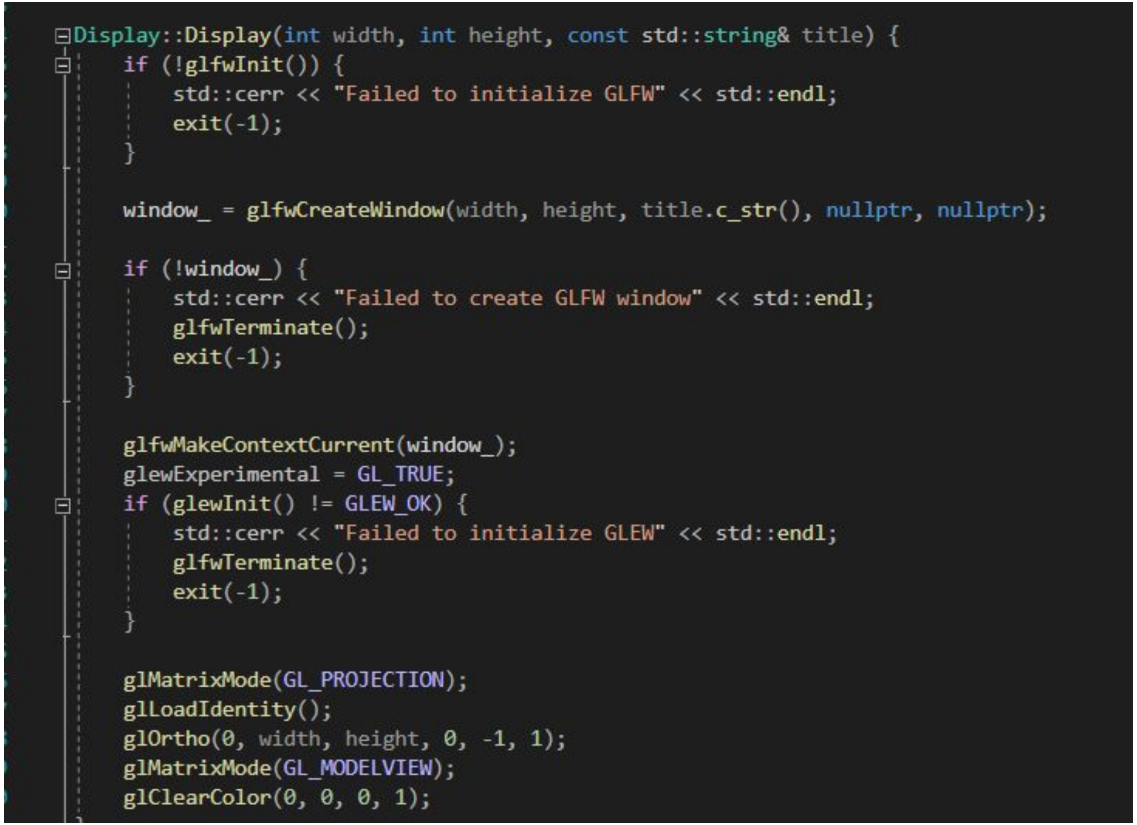
VAO and VBO are both used to store and manage vertex data in OpenGL.

DrawSquare()



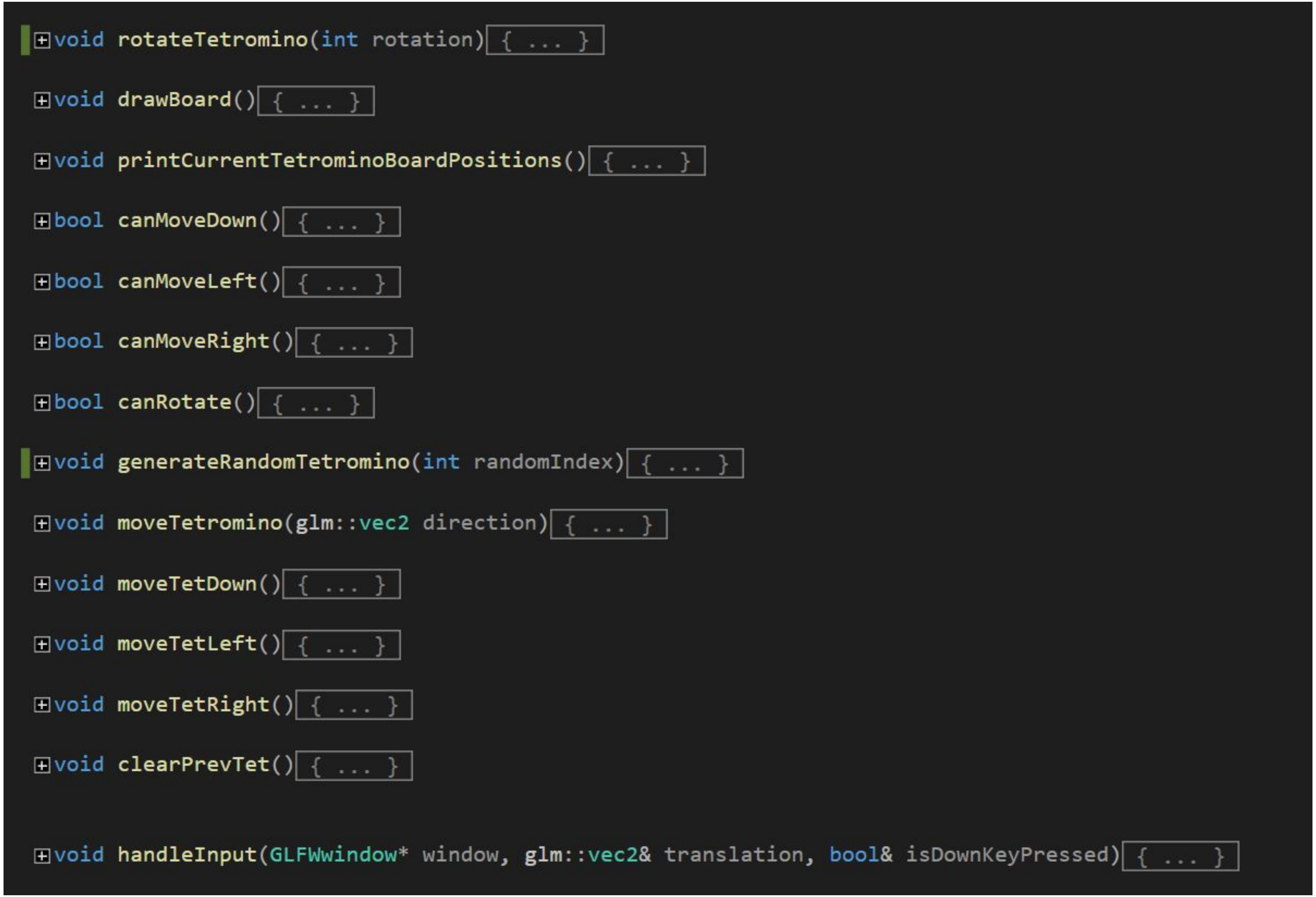
```
void drawSquare(glm::vec4 color, glm::vec2 squareTranslation) {
    glUseProgram(shaderProgram);
    glUniform4fv(colorLocation, 1, glm::value_ptr(color));
    glUniform2fv(translationLocation, 1, glm::value_ptr(squareTranslation));
    glUniformMatrix4fv(projectionLocation, 1, GL_FALSE, glm::value_ptr(projectionMatrix));

    glBindVertexArray(vao);
    glDrawArrays(GL_TRIANGLES, 0, 6);

    glUseProgram(0);
    glBindVertexArray(0);
}
```

Display

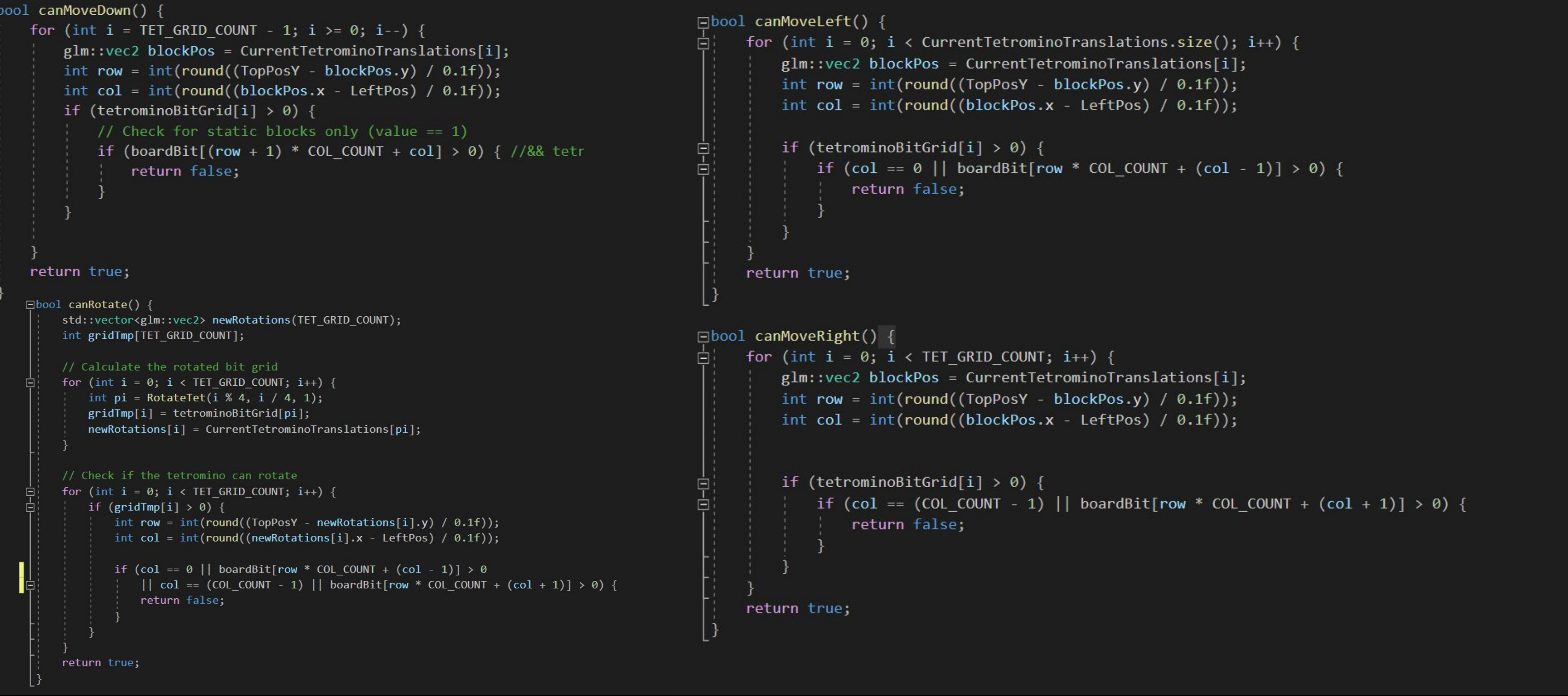


```
Display::Display(int width, int height, const std::string& title) {
    if (!glfwInit()) {
        std::cerr << "Failed to initialize GLFW" << std::endl;
        exit(-1);
    }

    window_ = glfwCreateWindow(width, height, title.c_str(), nullptr, nullptr);

    if (!window_) {
        std::cerr << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        exit(-1);
    }

    glfwMakeContextCurrent(window_);
    glewExperimental = GL_TRUE;
    if (glewInit() != GLEW_OK) {
        std::cerr << "Failed to initialize GLEW" << std::endl;
        glfwTerminate();
        exit(-1);
    }

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, width, height, 0, -1, 1);
    glMatrixMode(GL_MODELVIEW);
    glClearColor(0, 0, 0, 1);
```

GAME LOGIC



```cpp
void rotateTetromino(int rotation){ ... }

void drawBoard(){ ... }

void printCurrentTetrominoBoardPositions(){ ... }

bool canMoveDown(){ ... }

bool canMoveLeft(){ ... }

bool canMoveRight(){ ... }

bool canRotate(){ ... }

void generateRandomTetromino(int randomIndex){ ... }

void moveTetromino(glm::vec2 direction){ ... }

void moveTetDown(){ ... }

void moveTetLeft(){ ... }

void moveTetRight(){ ... }

void clearPrevTet(){ ... }


void handleInput(GLFWwindow* window, glm::vec2& translation, bool& isDownKeyPressed){ ... }
```



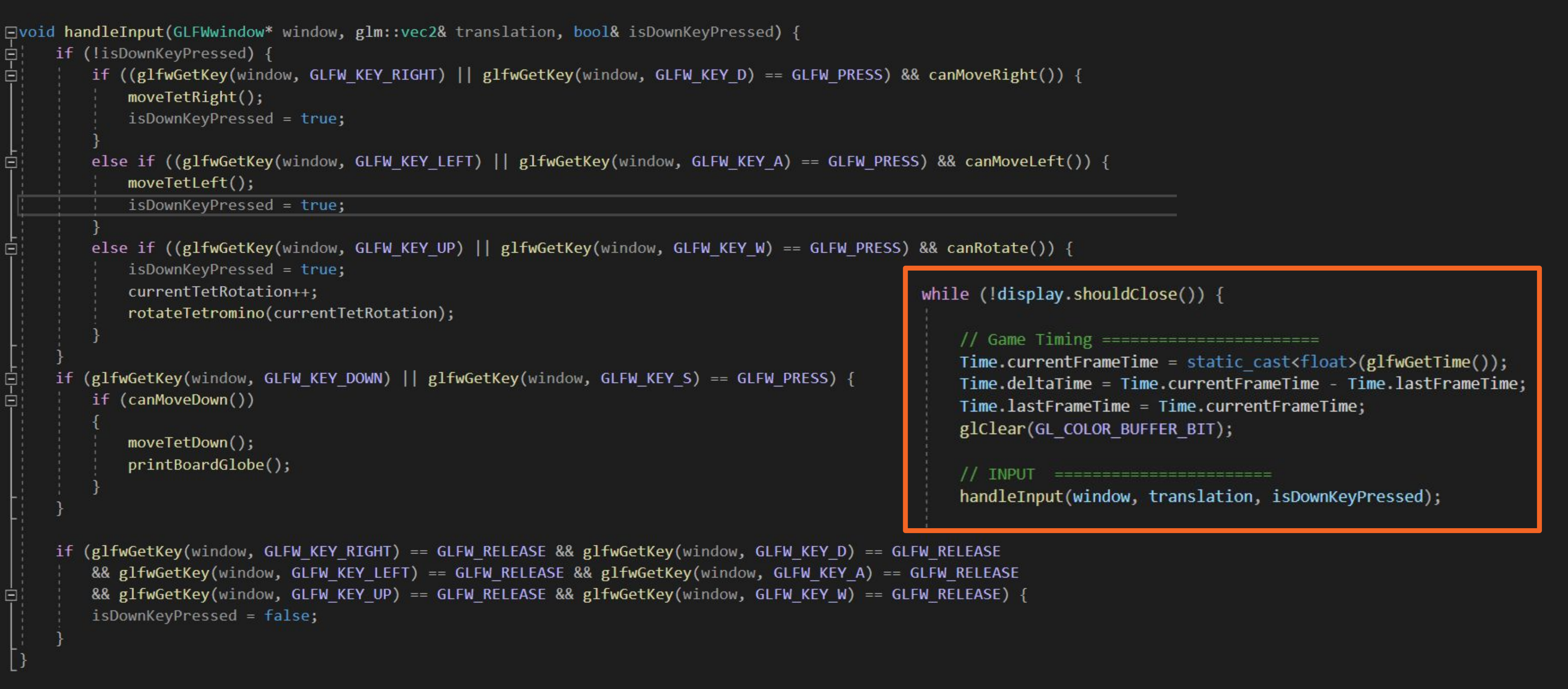
```
ool canMoveDown() {
    for (int i = TET_GRID_COUNT - 1; i >= 0; i--) {
        glm::vec2 blockPos = CurrentTetrominoTranslations[i];
        int row = int(round((TopPosY - blockPos.y) / 0.1f));
        int col = int(round((blockPos.x - LeftPos) / 0.1f));
        if (tetrominoBitGrid[i] > 0) {
            // Check for static blocks only (value == 1)
            if (boardBit[(row + 1) * COL_COUNT + col] > 0) { //&& tetr
                return false;
            }
        }

    }
    return true;
}
```

```
bool canRotate() {
    std::vector<glm::vec2> newRotations(TET_GRID_COUNT);
    int gridTmp[TET_GRID_COUNT];

    // Calculate the rotated bit grid
    for (int i = 0; i < TET_GRID_COUNT; i++) {
        int pi = RotateTet(i % 4, i / 4, 1);
        gridTmp[i] = tetrominoBitGrid[pi];
        newRotations[i] = CurrentTetrominoTranslations[pi];
    }

    // Check if the tetromino can rotate
    for (int i = 0; i < TET_GRID_COUNT; i++) {
        if (gridTmp[i] > 0) {
            int row = int(round((TopPosY - newRotations[i].y) / 0.1f));
            int col = int(round((newRotations[i].x - LeftPos) / 0.1f));

            if (col == 0 || boardBit[row * COL_COUNT + (col - 1)] > 0
                || col == (COL_COUNT - 1) || boardBit[row * COL_COUNT + (col + 1)] > 0) {
                return false;
            }
        }
    }
    return true;
}
```

```
bool canMoveLeft() {
    for (int i = 0; i < CurrentTetrominoTranslations.size(); i++) {
        glm::vec2 blockPos = CurrentTetrominoTranslations[i];
        int row = int(round((TopPosY - blockPos.y) / 0.1f));
        int col = int(round((blockPos.x - LeftPos) / 0.1f));

        if (tetrominoBitGrid[i] > 0) {
            if (col == 0 || boardBit[row * COL_COUNT + (col - 1)] > 0) {
                return false;
            }
        }
    }
    return true;
}
```

```
bool canMoveRight() {
    for (int i = 0; i < TET_GRID_COUNT; i++) {
        glm::vec2 blockPos = CurrentTetrominoTranslations[i];
        int row = int(round((TopPosY - blockPos.y) / 0.1f));
        int col = int(round((blockPos.x - LeftPos) / 0.1f));


        if (tetrominoBitGrid[i] > 0) {
            if (col == (COL_COUNT - 1) || boardBit[row * COL_COUNT + (col + 1)] > 0) {
                return false;
            }
        }
    }
    return true;
}
```

Check Tetromino Valid Move

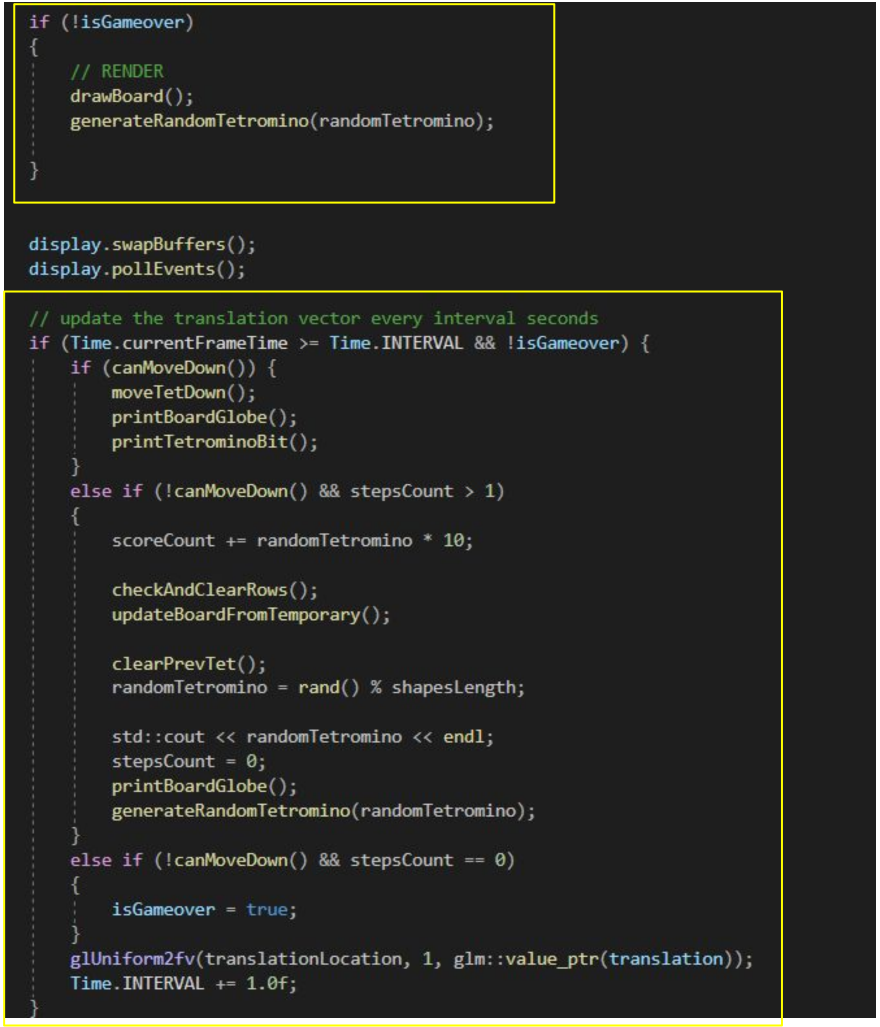


```
void handleInput(GLFWwindow* window, glm::vec2& translation, bool& isDownKeyPressed) {
    if (!isDownKeyPressed) {
        if ((glfwGetKey(window, GLFW_KEY_RIGHT) || glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS) && canMoveRight()) {
            moveTetRight();
            isDownKeyPressed = true;
        }
        else if ((glfwGetKey(window, GLFW_KEY_LEFT) || glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS) && canMoveLeft()) {
            moveTetLeft();
            isDownKeyPressed = true;
        }
        else if ((glfwGetKey(window, GLFW_KEY_UP) || glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) && canRotate()) {
            isDownKeyPressed = true;
            currentTetRotation++;
            rotateTetromino(currentTetRotation);
        }
    }
    if (glfwGetKey(window, GLFW_KEY_DOWN) || glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS) {
        if (canMoveDown())
        {
            moveTetDown();
            printBoardGlobe();
        }
    }

    if (glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_RELEASE && glfwGetKey(window, GLFW_KEY_D) == GLFW_RELEASE
        && glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_RELEASE && glfwGetKey(window, GLFW_KEY_A) == GLFW_RELEASE
        && glfwGetKey(window, GLFW_KEY_UP) == GLFW_RELEASE && glfwGetKey(window, GLFW_KEY_W) == GLFW_RELEASE) {
        isDownKeyPressed = false;
    }
}
```

```
while (!display.shouldClose()) {

    // Game Timing =======================
    Time.currentFrameTime = static_cast<float>(glfwGetTime());
    Time.deltaTime = Time.currentFrameTime - Time.lastFrameTime;
    Time.lastFrameTime = Time.currentFrameTime;
    glClear(GL_COLOR_BUFFER_BIT);

    // INPUT  =======================
    handleInput(window, translation, isDownKeyPressed);
```

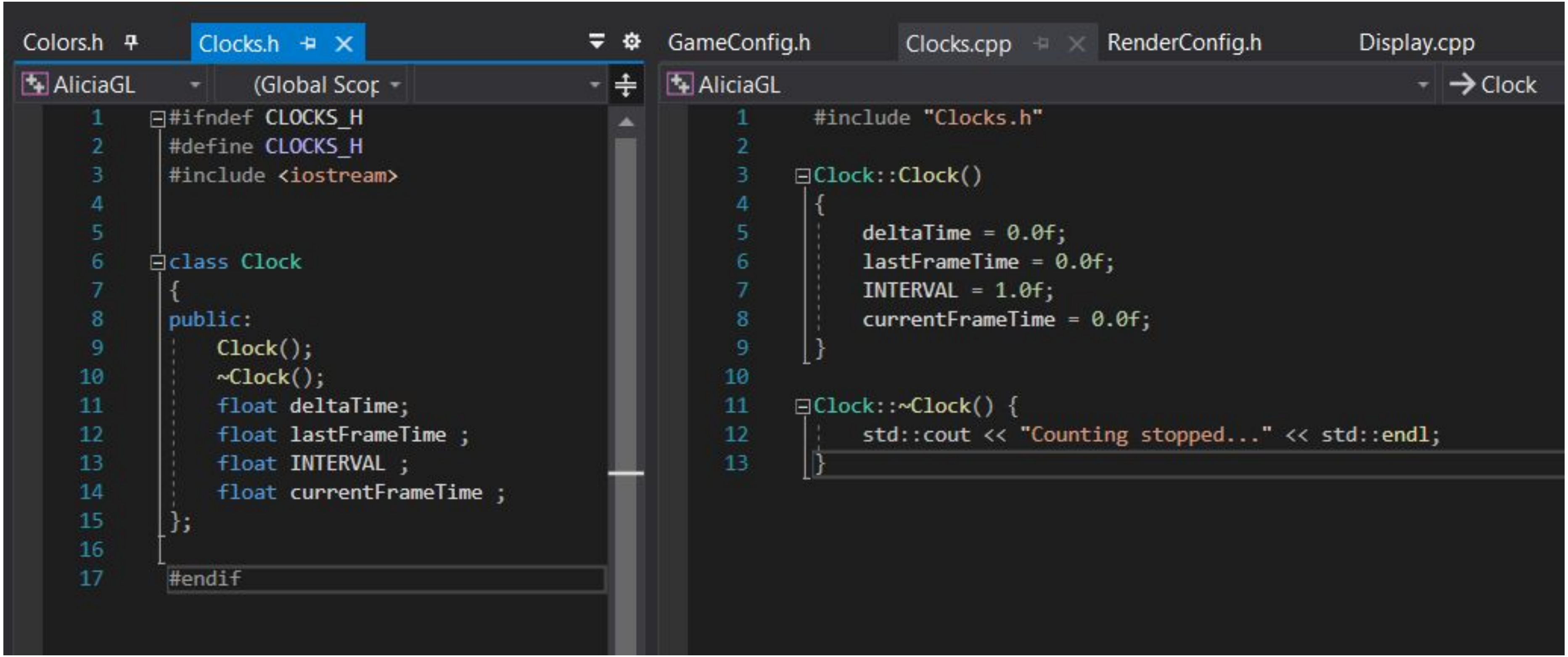
Input Handler

GAME LOOP



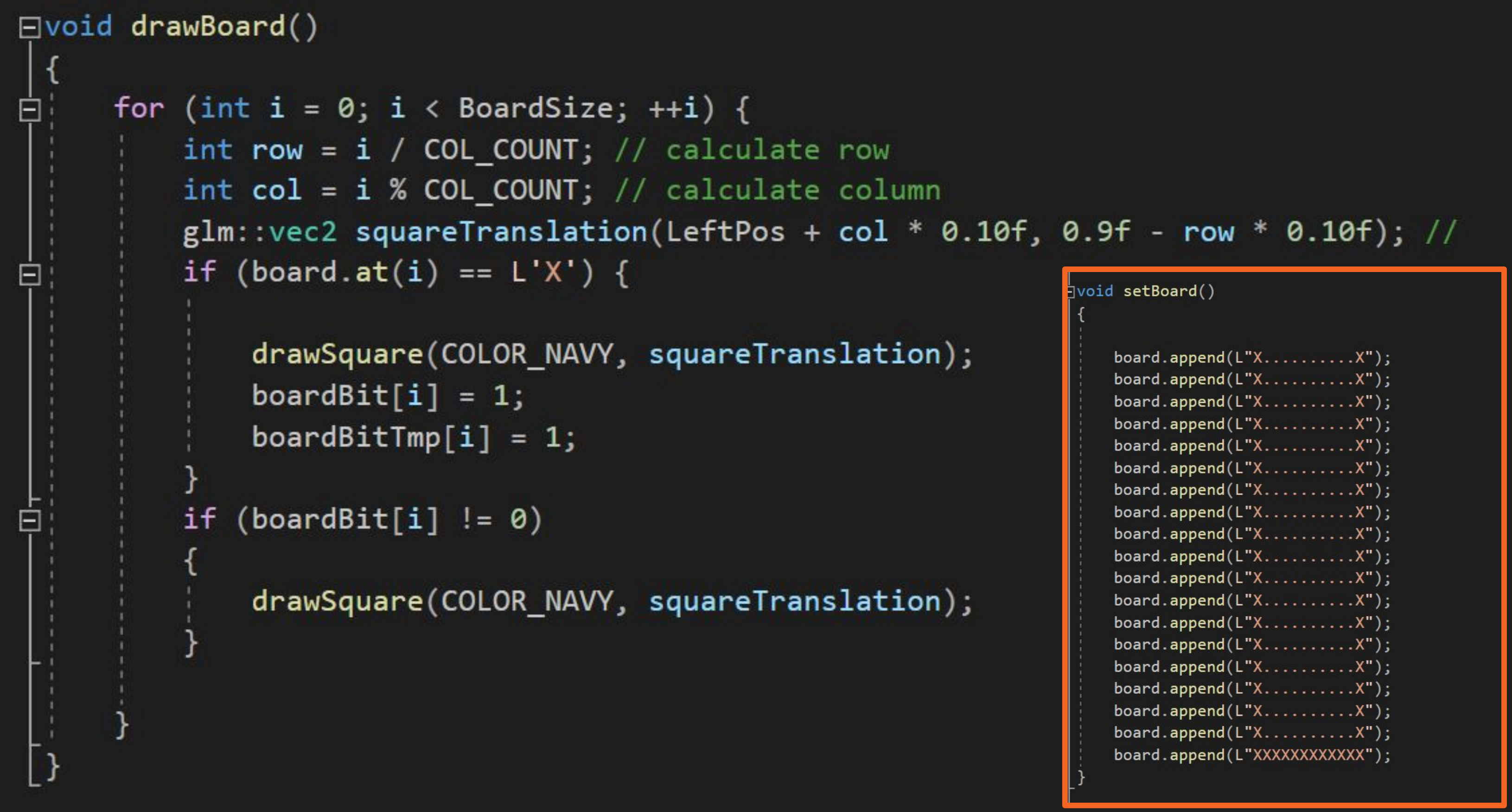
```
if (!isGameover)
{
    // RENDER
    drawBoard();
    generateRandomTetromino(randomTetromino);

}

display.swapBuffers();
display.pollEvents();

// update the translation vector every interval seconds
if (Time.currentFrameTime >= Time.INTERVAL && !isGameover) {
    if (canMoveDown()) {
        moveTetDown();
        printBoardGlobe();
        printTetrominoBit();
    }
    else if (!canMoveDown() && stepsCount > 1)
    {
        scoreCount += randomTetromino * 10;

        checkAndClearRows();
        updateBoardFromTemporary();

        clearPrevTet();
        randomTetromino = rand() % shapesLength;

        std::cout << randomTetromino << endl;
        stepsCount = 0;
        printBoardGlobe();
        generateRandomTetromino(randomTetromino);
    }
    else if (!canMoveDown() && stepsCount == 0)
    {
        isGameover = true;
    }
    glUniform2fv(translationLocation, 1, glm::value_ptr(translation));
    Time.INTERVAL += 1.0f;
}
```

Clock



```cpp
#ifndef CLOCKS_H
#define CLOCKS_H
#include <iostream>


class Clock
{
public:
    Clock();
    ~Clock();
    float deltaTime;
    float lastFrameTime ;
    float INTERVAL ;
    float currentFrameTime ;
};

#endif
```

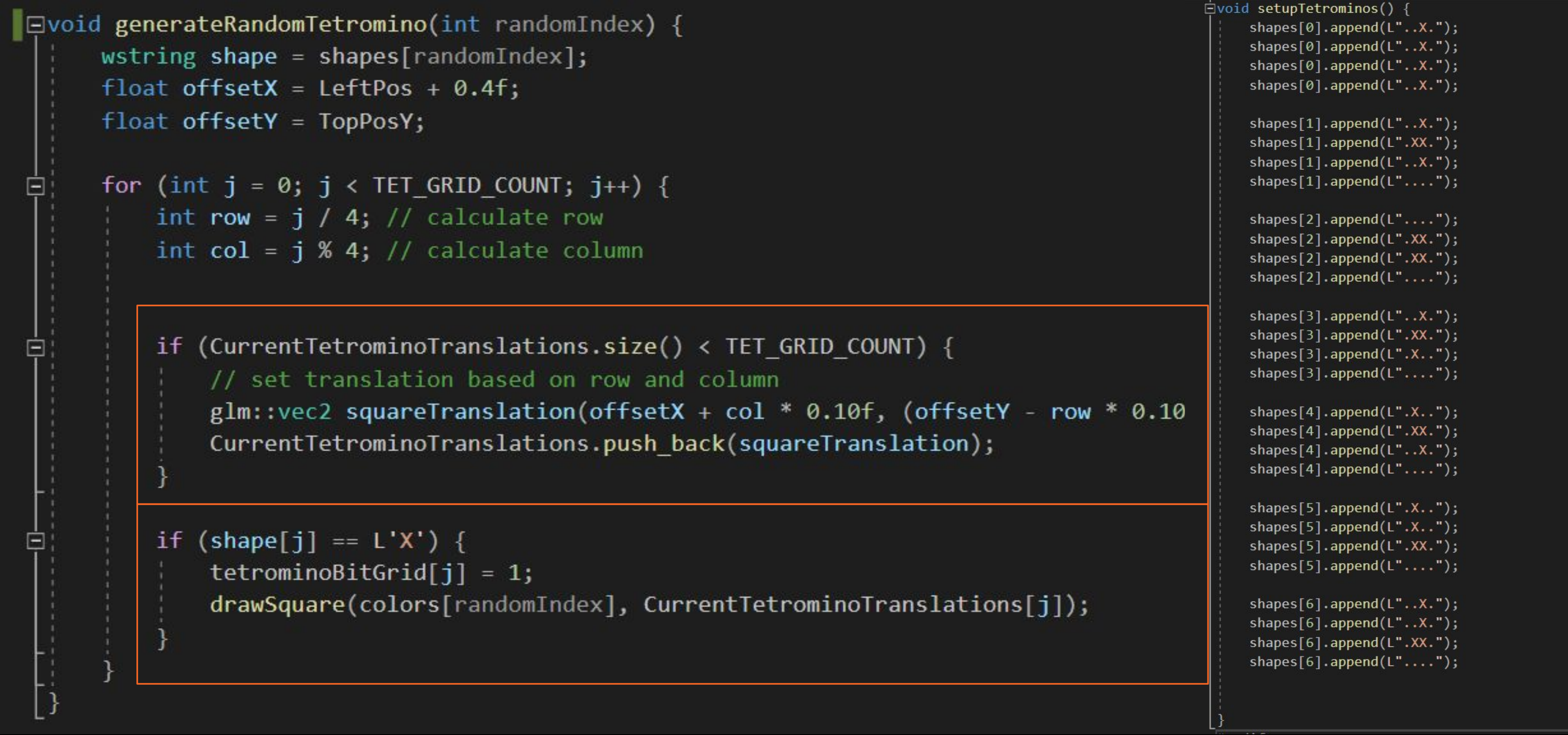
```cpp
#include "Clocks.h"

Clock::Clock()
{
    deltaTime = 0.0f;
    lastFrameTime = 0.0f;
    INTERVAL = 1.0f;
    currentFrameTime = 0.0f;
}

Clock::~Clock() {
    std::cout << "Counting stopped..." << std::endl;
}
```



```cpp
void drawBoard()
{
    for (int i = 0; i < BoardSize; ++i) {
        int row = i / COL_COUNT; // calculate row
        int col = i % COL_COUNT; // calculate column
        glm::vec2 squareTranslation(LeftPos + col * 0.10f, 0.9f - row * 0.10f); //
        if (board.at(i) == L'X') {

            drawSquare(COLOR_NAVY, squareTranslation);
            boardBit[i] = 1;
            boardBitTmp[i] = 1;
        }
        if (boardBit[i] != 0)
        {
            drawSquare(COLOR_NAVY, squareTranslation);
        }

    }
}
```

```cpp
void setBoard()
{

    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"X..........X");
    board.append(L"XXXXXXXXXXXX");
}
```

Draw Board



```cpp
void generateRandomTetromino(int randomIndex) {
    wstring shape = shapes[randomIndex];
    float offsetX = LeftPos + 0.4f;
    float offsetY = TopPosY;

    for (int j = 0; j < TET_GRID_COUNT; j++) {
        int row = j / 4; // calculate row
        int col = j % 4; // calculate column

        if (CurrentTetrominoTranslations.size() < TET_GRID_COUNT) {
            // set translation based on row and column
            glm::vec2 squareTranslation(offsetX + col * 0.10f, (offsetY - row * 0.10
            CurrentTetrominoTranslations.push_back(squareTranslation);
        }

        if (shape[j] == L'X') {
            tetrominoBitGrid[j] = 1;
            drawSquare(colors[randomIndex], CurrentTetrominoTranslations[j]);
        }
    }
}
```

```cpp
void setupTetrominos() {
    shapes[0].append(L"..X.");
    shapes[0].append(L"..X.");
    shapes[0].append(L"..X.");
    shapes[0].append(L"..X.");

    shapes[1].append(L"..X.");
    shapes[1].append(L".XX.");
    shapes[1].append(L"..X.");
    shapes[1].append(L"....");

    shapes[2].append(L"....");
    shapes[2].append(L".XX.");
    shapes[2].append(L".XX.");
    shapes[2].append(L"....");

    shapes[3].append(L"..X.");
    shapes[3].append(L".XX.");
    shapes[3].append(L".X..");
    shapes[3].append(L"....");

    shapes[4].append(L".X..");
    shapes[4].append(L".XX.");
    shapes[4].append(L"..X.");
    shapes[4].append(L"....");

    shapes[5].append(L".X..");
    shapes[5].append(L".X..");
    shapes[5].append(L".XX.");
    shapes[5].append(L"....");

    shapes[6].append(L"..X.");
    shapes[6].append(L"..X.");
    shapes[6].append(L".XX.");
    shapes[6].append(L"....");


}
```

Spawn(draw) Random Tetromino

```
0500
0550
0050
0000
```

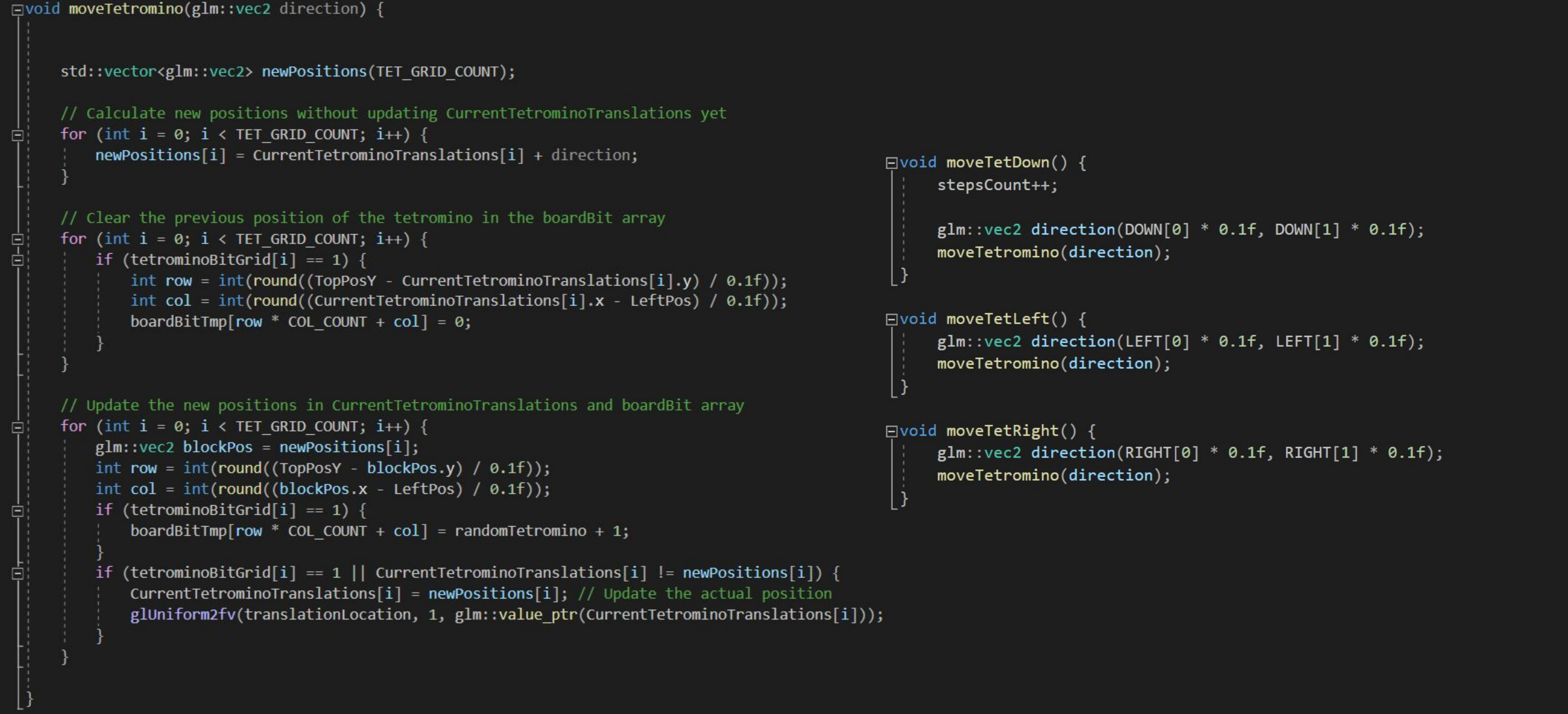
```
0010
0010
0010
0010
```

```
0020
0220
0020
0000
```

```
0000
0660
0060
0060
```

```
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
111111111111
```

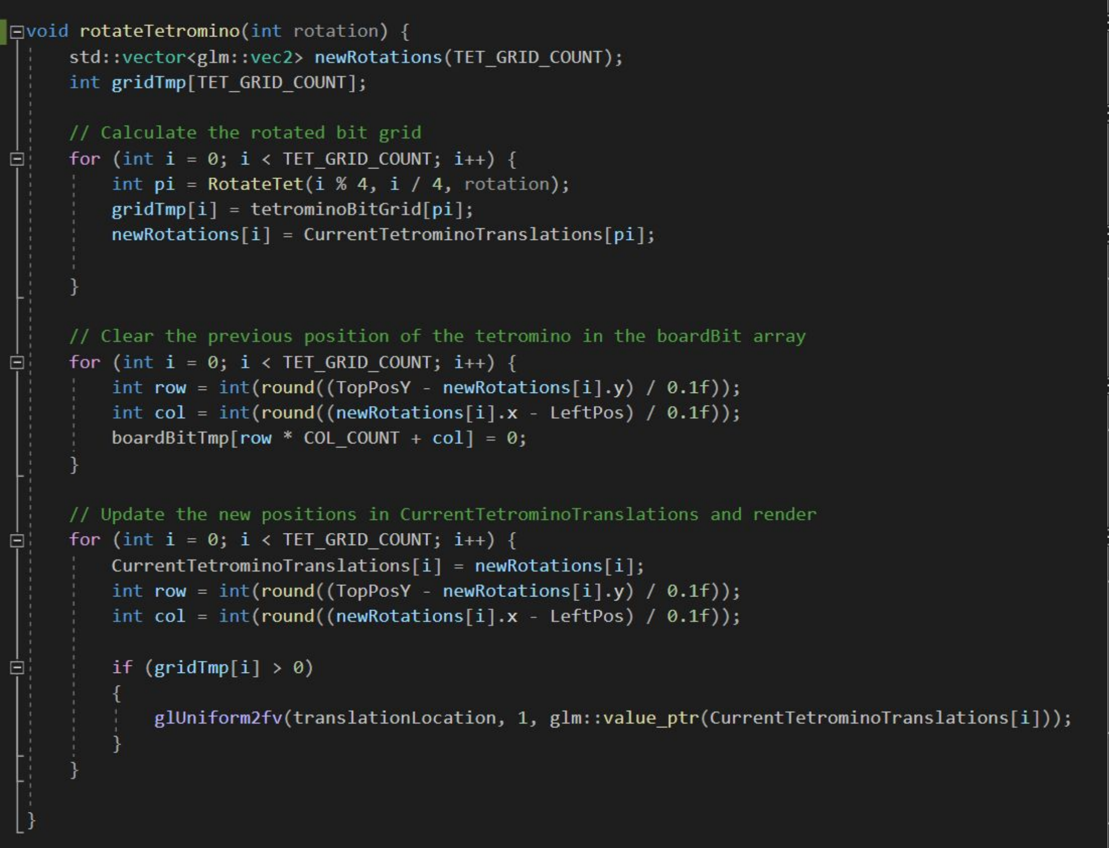
```
100000000001
100000000001
100000000001
100000000001
100000200001
100002200001
100000200001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
100000000001
111111111111
```



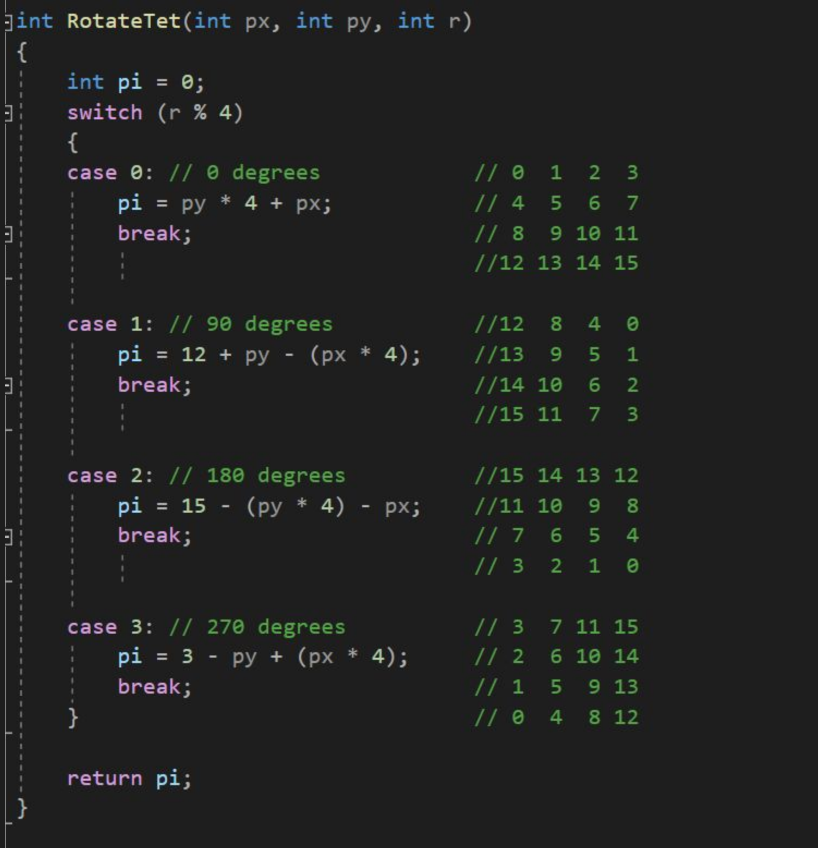
```cpp
void moveTetromino(glm::vec2 direction) {


    std::vector<glm::vec2> newPositions(TET_GRID_COUNT);

    // Calculate new positions without updating CurrentTetrominoTranslations yet
    for (int i = 0; i < TET_GRID_COUNT; i++) {
        newPositions[i] = CurrentTetrominoTranslations[i] + direction;
    }

    // Clear the previous position of the tetromino in the boardBit array
    for (int i = 0; i < TET_GRID_COUNT; i++) {
        if (tetrominoBitGrid[i] == 1) {
            int row = int(round((TopPosY - CurrentTetrominoTranslations[i].y) / 0.1f));
            int col = int(round((CurrentTetrominoTranslations[i].x - LeftPos) / 0.1f));
            boardBitTmp[row * COL_COUNT + col] = 0;
        }
    }

    // Update the new positions in CurrentTetrominoTranslations and boardBit array
    for (int i = 0; i < TET_GRID_COUNT; i++) {
        glm::vec2 blockPos = newPositions[i];
        int row = int(round((TopPosY - blockPos.y) / 0.1f));
        int col = int(round((blockPos.x - LeftPos) / 0.1f));
        if (tetrominoBitGrid[i] == 1) {
            boardBitTmp[row * COL_COUNT + col] = randomTetromino + 1;
        }
        if (tetrominoBitGrid[i] == 1 || CurrentTetrominoTranslations[i] != newPositions[i]) {
            CurrentTetrominoTranslations[i] = newPositions[i]; // Update the actual position
            glUniform2fv(translationLocation, 1, glm::value_ptr(CurrentTetrominoTranslations[i]));
        }
    }

}
```

```cpp
void moveTetDown() {
    stepsCount++;

    glm::vec2 direction(DOWN[0] * 0.1f, DOWN[1] * 0.1f);
    moveTetromino(direction);
}

void moveTetLeft() {
    glm::vec2 direction(LEFT[0] * 0.1f, LEFT[1] * 0.1f);
    moveTetromino(direction);
}

void moveTetRight() {
    glm::vec2 direction(RIGHT[0] * 0.1f, RIGHT[1] * 0.1f);
    moveTetromino(direction);
}
```

Translation(s)

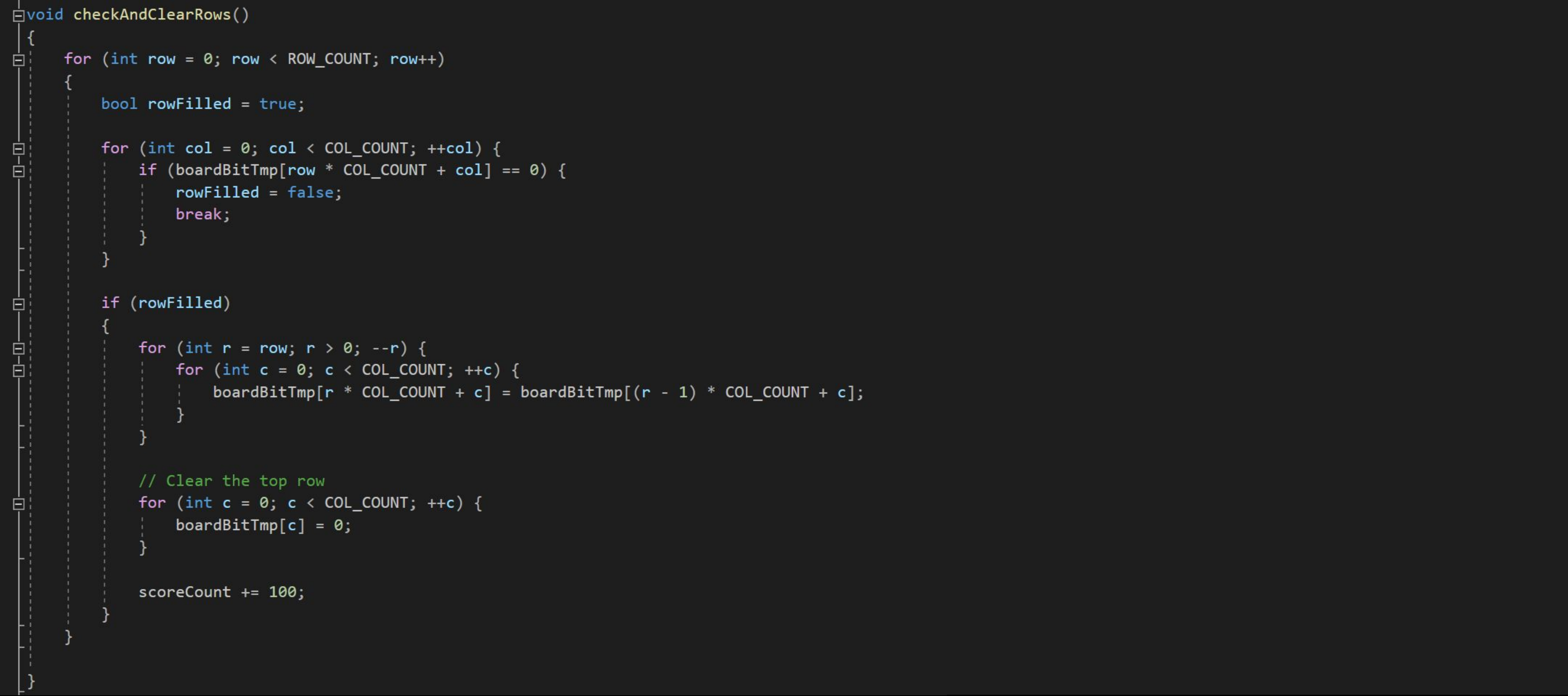


```cpp
void rotateTetromino(int rotation) {
    std::vector<glm::vec2> newRotations(TET_GRID_COUNT);
    int gridTmp[TET_GRID_COUNT];

    // Calculate the rotated bit grid
    for (int i = 0; i < TET_GRID_COUNT; i++) {
        int pi = RotateTet(i % 4, i / 4, rotation);
        gridTmp[i] = tetrominoBitGrid[pi];
        newRotations[i] = CurrentTetrominoTranslations[pi];

    }

    // Clear the previous position of the tetromino in the boardBit array
    for (int i = 0; i < TET_GRID_COUNT; i++) {
        int row = int(round((TopPosY - newRotations[i].y) / 0.1f));
        int col = int(round((newRotations[i].x - LeftPos) / 0.1f));
        boardBitTmp[row * COL_COUNT + col] = 0;
    }

    // Update the new positions in CurrentTetrominoTranslations and render
    for (int i = 0; i < TET_GRID_COUNT; i++) {
        CurrentTetrominoTranslations[i] = newRotations[i];
        int row = int(round((TopPosY - newRotations[i].y) / 0.1f));
        int col = int(round((newRotations[i].x - LeftPos) / 0.1f));

        if (gridTmp[i] > 0)
        {
            glUniform2fv(translationLocation, 1, glm::value_ptr(CurrentTetrominoTranslations[i]));
        }
    }

}
```



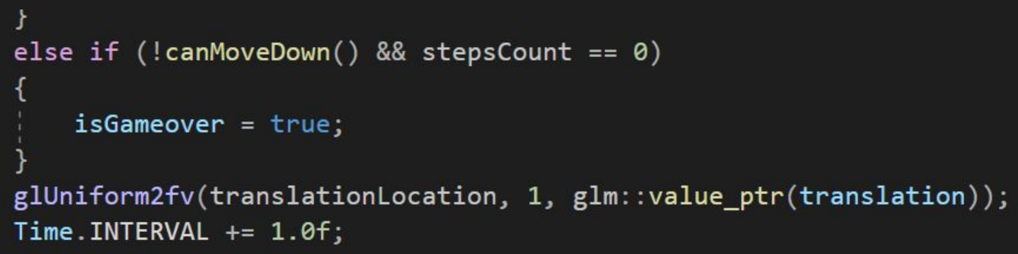
```cpp
int RotateTet(int px, int py, int r)
{
    int pi = 0;
    switch (r % 4)
    {
    case 0: // 0 degrees                // 0  1  2  3
        pi = py * 4 + px;               // 4  5  6  7
        break;                          // 8  9 10 11
                                        //12 13 14 15

    case 1: // 90 degrees               //12  8  4  0
        pi = 12 + py - (px * 4);        //13  9  5  1
        break;                          //14 10  6  2
                                        //15 11  7  3

    case 2: // 180 degrees              //15 14 13 12
        pi = 15 - (py * 4) - px;        //11 10  9  8
        break;                          // 7  6  5  4
                                        // 3  2  1  0

    case 3: // 270 degrees              // 3  7 11 15
        pi = 3 - py + (px * 4);         // 2  6 10 14
        break;                          // 1  5  9 13
    }                                   // 0  4  8 12

    return pi;
}
```

Rotation



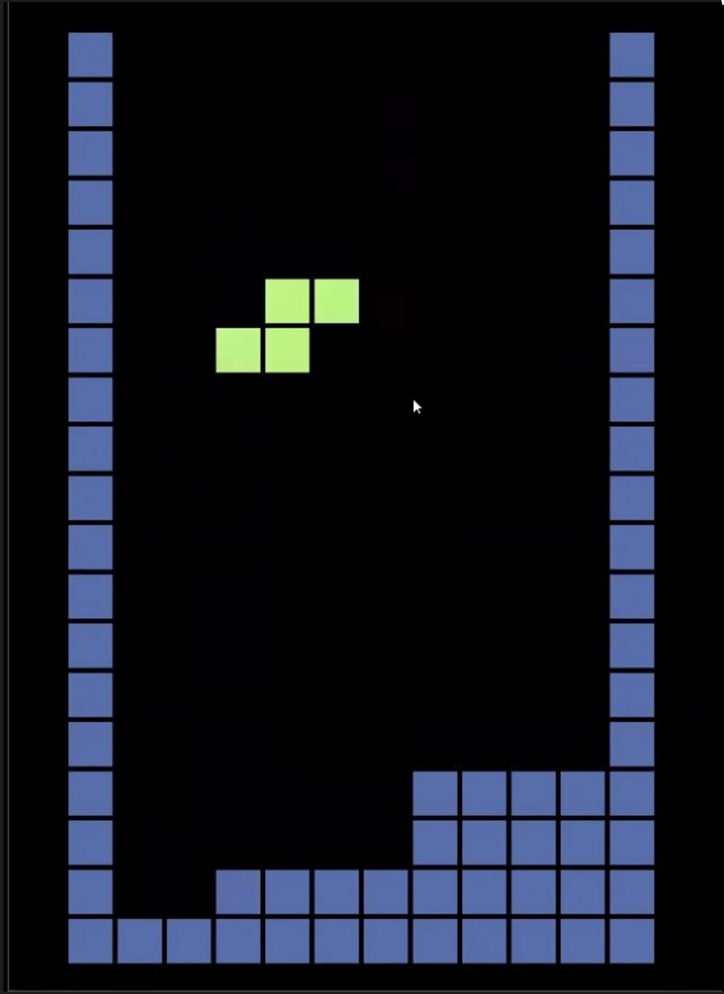
```
void checkAndClearRows()
{
    for (int row = 0; row < ROW_COUNT; row++)
    {
        bool rowFilled = true;

        for (int col = 0; col < COL_COUNT; ++col) {
            if (boardBitTmp[row * COL_COUNT + col] == 0) {
                rowFilled = false;
                break;
            }
        }

        if (rowFilled)
        {
            for (int r = row; r > 0; --r) {
                for (int c = 0; c < COL_COUNT; ++c) {
                    boardBitTmp[r * COL_COUNT + c] = boardBitTmp[(r - 1) * COL_COUNT + c];
                }
            }

            // Clear the top row
            for (int c = 0; c < COL_COUNT; ++c) {
                boardBitTmp[c] = 0;
            }

            scoreCount += 100;
        }
    }

}
```

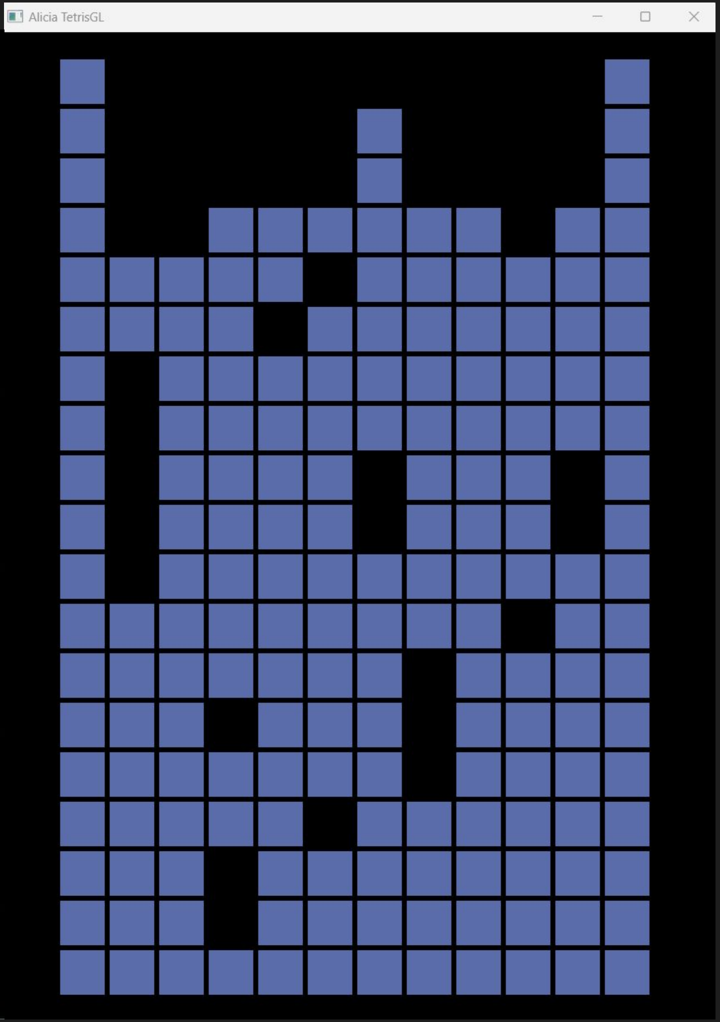
Clear Rows and Scoring



```
}
else if (!canMoveDown() && stepsCount == 0)
{
    isGameover = true;
}
glUniform2fv(translationLocation, 1, glm::value_ptr(translation));
Time.INTERVAL += 1.0f;
```

Check Gameover

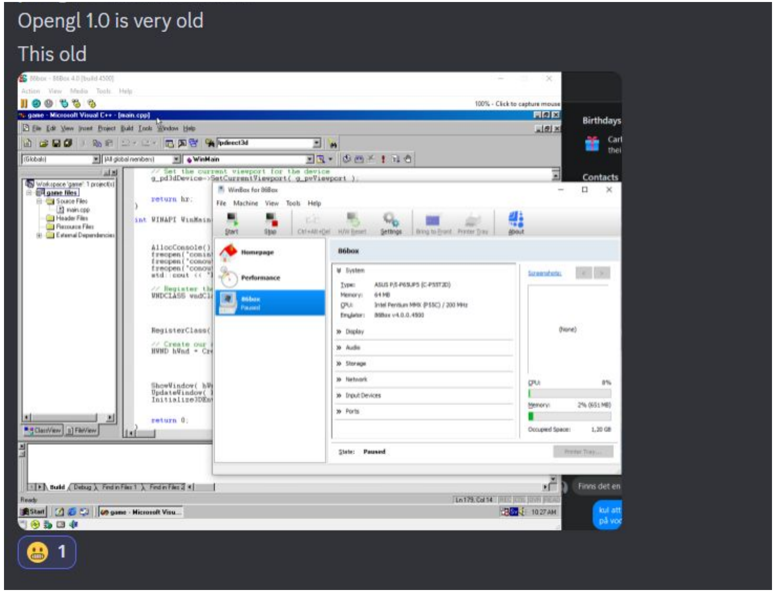




Alicia TetrisGL

Reflection

OpenGL1.0



Opengl 1.0 is very old

This old

86box

Type: ASUS P/I-P65UP5 (C-P55T2D)
Memory: 64 MB
CPU: Intel Pentium MMX (P55C) / 200 MHz
Emulator: 86Box v4.0.0.4500

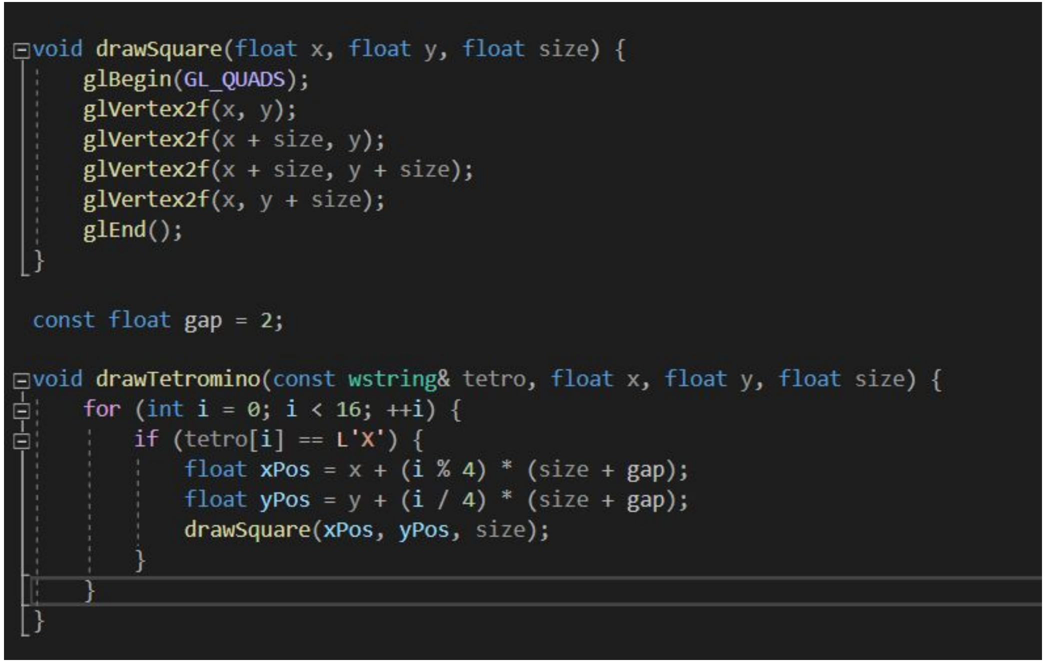
CPU: 0%
Memory: 2% (651 MB)
Occupied Space: 1,20 GB

State: Paused

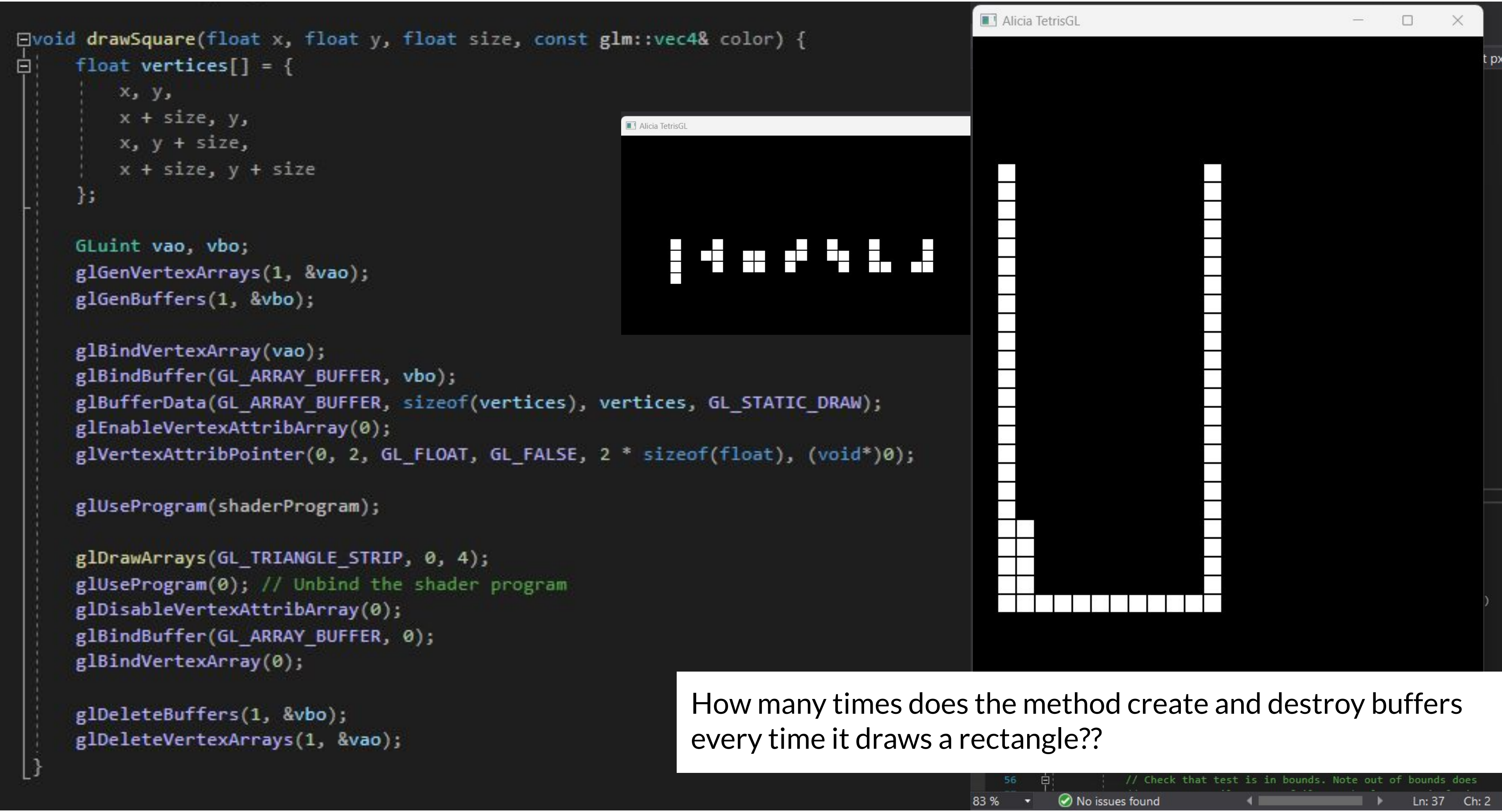
Finns det en

kul att på vo

😬 1



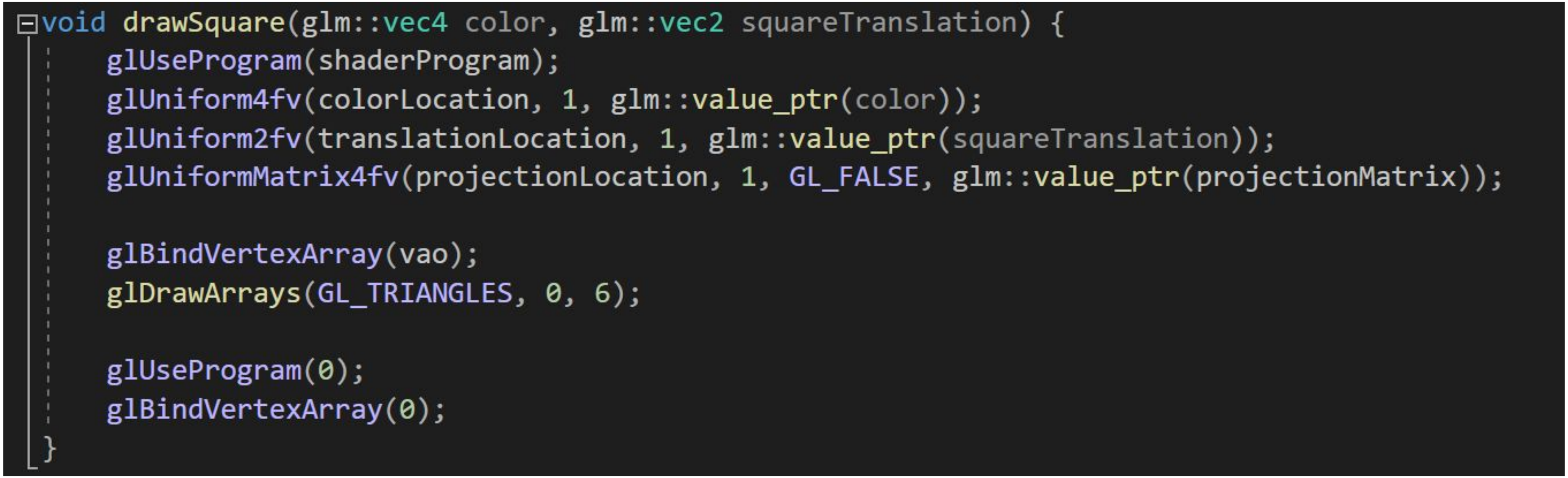
```cpp
void drawSquare(float x, float y, float size) {
    glBegin(GL_QUADS);
    glVertex2f(x, y);
    glVertex2f(x + size, y);
    glVertex2f(x + size, y + size);
    glVertex2f(x, y + size);
    glEnd();
}

const float gap = 2;

void drawTetromino(const wstring& tetro, float x, float y, float size) {
    for (int i = 0; i < 16; ++i) {
        if (tetro[i] == L'X') {
            float xPos = x + (i % 4) * (size + gap);
            float yPos = y + (i / 4) * (size + gap);
            drawSquare(xPos, yPos, size);
        }
    }
}
```



```
void drawSquare(float x, float y, float size, const glm::vec4& color) {
    float vertices[] = {
        x, y,
        x + size, y,
        x, y + size,
        x + size, y + size
    };

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

    glBindVertexArray(vao);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)0);

    glUseProgram(shaderProgram);

    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    glUseProgram(0); // Unbind the shader program
    glDisableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);

    glDeleteBuffers(1, &vbo);
    glDeleteVertexArrays(1, &vao);
}
```

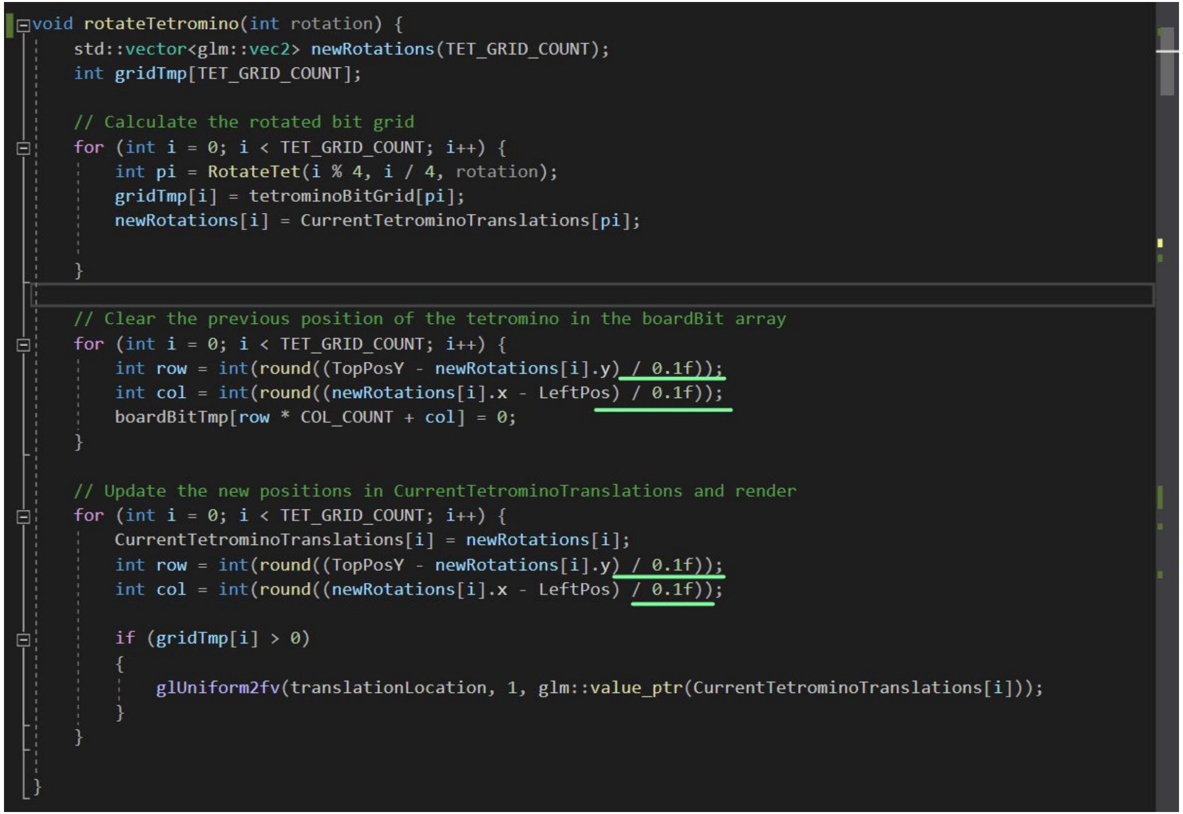
How many times does the method create and destroy buffers every time it draws a rectangle??

What it really needs…



```
void drawSquare(glm::vec4 color, glm::vec2 squareTranslation) {
    glUseProgram(shaderProgram);
    glUniform4fv(colorLocation, 1, glm::value_ptr(color));
    glUniform2fv(translationLocation, 1, glm::value_ptr(squareTranslation));
    glUniformMatrix4fv(projectionLocation, 1, GL_FALSE, glm::value_ptr(projectionMatrix));

    glBindVertexArray(vao);
    glDrawArrays(GL_TRIANGLES, 0, 6);

    glUseProgram(0);
    glBindVertexArray(0);
}
```

Classic Tetris doesn't Use float…



```
void rotateTetromino(int rotation) {
    std::vector<glm::vec2> newRotations(TET_GRID_COUNT);
    int gridTmp[TET_GRID_COUNT];

    // Calculate the rotated bit grid
    for (int i = 0; i < TET_GRID_COUNT; i++) {
        int pi = RotateTet(i % 4, i / 4, rotation);
        gridTmp[i] = tetrominoBitGrid[pi];
        newRotations[i] = CurrentTetrominoTranslations[pi];

    }

    // Clear the previous position of the tetromino in the boardBit array
    for (int i = 0; i < TET_GRID_COUNT; i++) {
        int row = int(round((TopPosY - newRotations[i].y) / 0.1f));
        int col = int(round((newRotations[i].x - LeftPos) / 0.1f));
        boardBitTmp[row * COL_COUNT + col] = 0;
    }

    // Update the new positions in CurrentTetrominoTranslations and render
    for (int i = 0; i < TET_GRID_COUNT; i++) {
        CurrentTetrominoTranslations[i] = newRotations[i];
        int row = int(round((TopPosY - newRotations[i].y) / 0.1f));
        int col = int(round((newRotations[i].x - LeftPos) / 0.1f));

        if (gridTmp[i] > 0)
        {
            glUniform2fv(translationLocation, 1, glm::value_ptr(CurrentTetrominoTranslations[i]));
        }
    }

}
```

Summary

“Setting up OpenGL shaders and buffers are daunting than applying game logic(s)…” - Alicia Sudlerd

Lesson Learned:

- Researched more about the game origin.
- Beware of OpenGL version

Future Plans

- We can make any 8-bit games using this engine (eg. Snake or Pong)

References

- OpenGL SuperBible
- SFML C++: TETRIS
    - https://www.youtube.com/watch?v=zH_omFPqMO4&t=12s
- Tetris - Programming from Scratch (game logic)
    - https://github.com/OneLoneCoder/Javidx9/blob/master/SimplyCode/OneLoneCoder_Tetris.cpp

Q&A