Proiect 2 - Labirint

Conceptul proiectului

Proiectul reprezintă un labirint 3D dintr-o perspectivă first-person. Proiectul include abstractizarea în clase a elementelor de baza din OpenGL, precum VAO, VBO, EBO, Shader, Texture sau Renderer.

Elemente implementate

- Cuaternioni

  Camera este implementata folosind cuaternioni. Pentru aducerea camerei înapoi în forma de matrice se folosește funcția `glm::mat4_cast`.

- Cubemap / Skybox

  Background-ul este implementat folosind Skybox. Pentru a păstra aparența distanței translatiile sunt ignorate.

- Texturi

  Blocurile din labirint sunt texturate folosind imaginea “graffiti.jpg”.

- Randare instanțiată

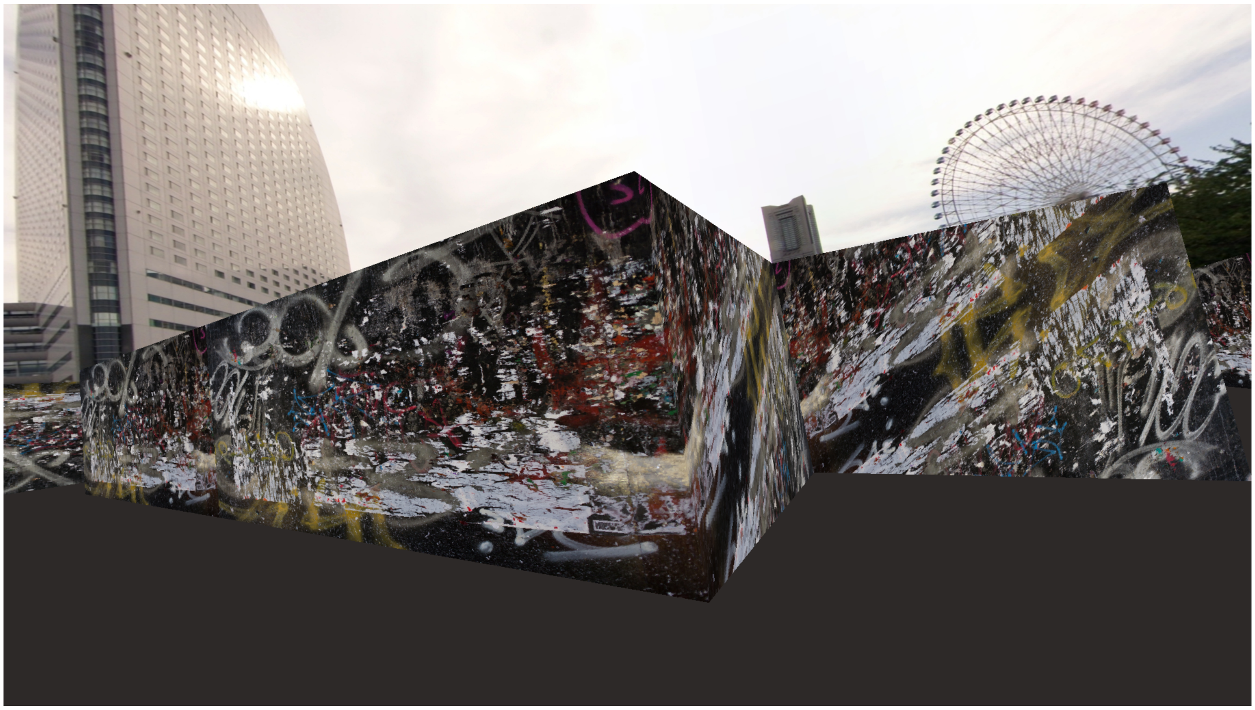
  Pentru a crea labirintul se folosește randarea instantiata. Un labirint este generat aleator în clasa Maze și este format din blocuri verticale, orizontale, cuburi și blocuri de tip corner.

- Reprezentare obiecte 3D
- Indexare

De ce este original?

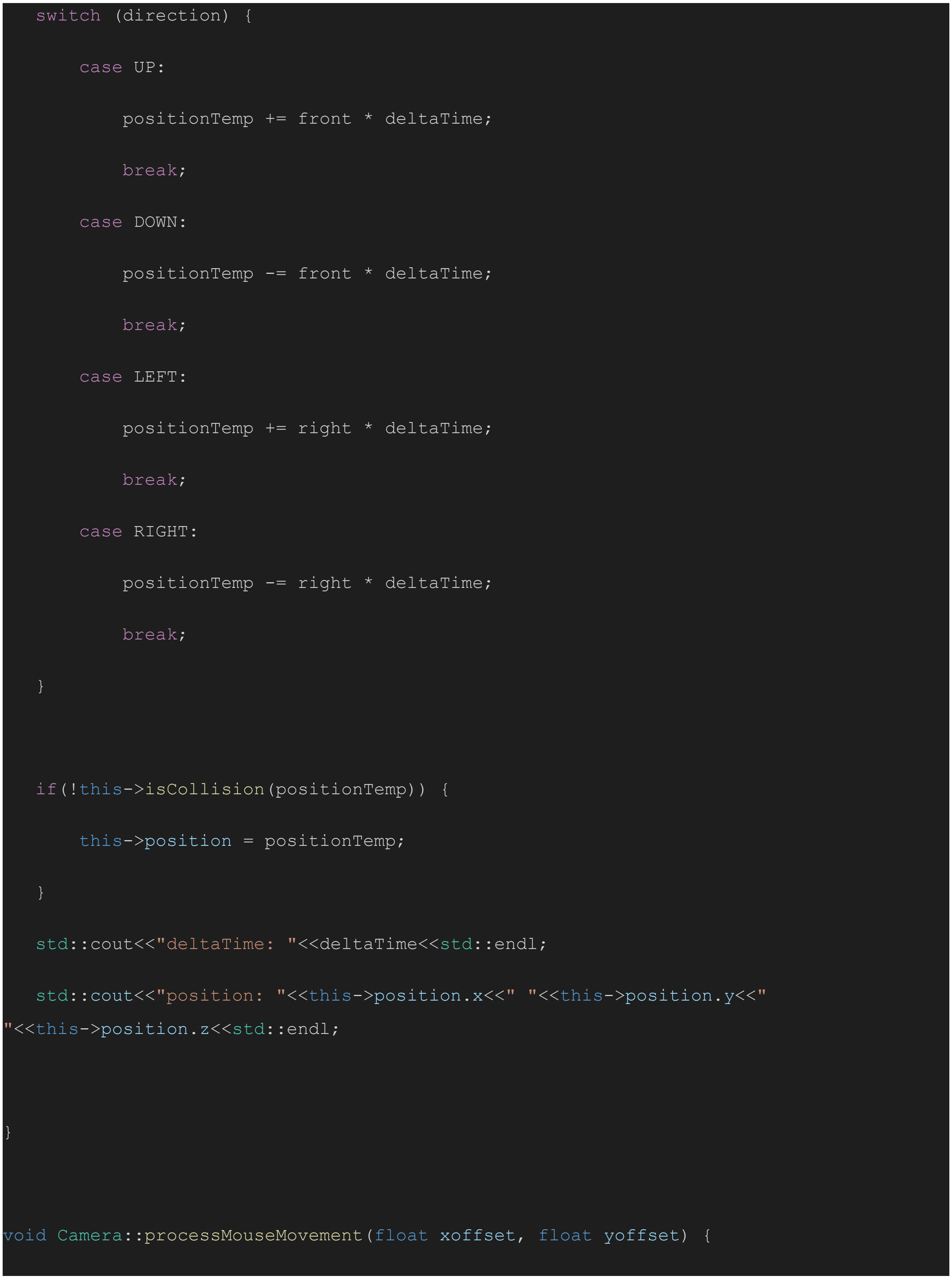
Originalitatea proiectului consta în abstractizarea în clase a elementelor de OpenGL. Astfel baza proiectului poate fi folosită pentru mai multe scene care folosesc elementele implementate. De asemenea, este implementata o perspectiva first person, care se poate roti 360 de grade. În plus, labirintul este generat aleator, scena fiind diferită de fiecare data cand proiectul este recompilat

Capturi de ecran



Cod

```
Camera::Camera(glm::vec3 position, glm::vec3 front, glm::vec3 worldUp) {

   this->position = position;

   this->front = front;

   this->worldUp = worldUp;

   this->lastX = 400.0f;

   this->lastY = 300.0f;

   this->yaw = -50.0f;

   this->pitch = 0.0f;

   this->orientation = glm::quat(0.0f, 0.0f, 0.0f, -1.0f);
   this->updateVectors();
}

Camera::~Camera() {

}

glm::mat4 Camera::getViewMatrix() {
   updateVectors();
   glm::quat conjugate = glm::conjugate(orientation);
   glm::mat4 rotate = glm::mat4_cast(conjugate);
   glm::mat4 translate = glm::translate(glm::mat4(1.0f), -this->position);
   return rotate * translate;
}

bool Camera::isCollision(glm::vec3 position) {
   int x = (int) position.x / 200;
   int z = (int) position.z / 200;

   int modX = abs((int)position.x % 200);
   int modZ = abs((int)position.z % 200);
   if(x < 0 || x >= 7 || z < 0 || z >= 7) {
       return false;
   }
   // std::cout<<"x: "<<x<<" z: "<<z<<" vertical: "<<verticalWalls[x][z]<<" horizontal: "<<horizontalWalls[x][z]<<std::endl;
   // std::cout<<"modX: "<<modX<<" modZ: "<<modZ<<std::endl;
   // bool insideVertical = (verticalWalls[x][z] == 1) && (modZ <= 30);
   // // std::cout<<"cond 1: "<<verticalWalls[x][z]<<" cond 2: "<< (modZ <= 30) <<std::endl;
   // bool insideHorizontal = (horizontalWalls[x][z] == 1) && (modX <= 30);
   // // std::cout<<"insideVertical: "<<insideVertical<<" insideHorizontal: "<<insideHorizontal<<std::endl;
   // if(insideHorizontal || insideVertical) {
   //     std::cout<<"collision"<<std::endl;
   //     // return true;
   // }
   return false;
}

void Camera::processKeyboard(direction direction, float deltaTime) {
   glm::quat qt = this->orientation * glm::quat(0.0f, 0.0f, 0.0f, -1.0f) * glm::conjugate(this->orientation);
   this->front = glm::vec3(qt.x, qt.y, qt.z);
   glm::vec3 right = glm::normalize(glm::cross(this->front, this->worldUp));
   right.y = 0.0f;
   front.y = 0.0f;

   glm::vec3 positionTemp = this->position;
```



```
    switch (direction) {
        case UP:
            positionTemp += front * deltaTime;
            break;
        case DOWN:
            positionTemp -= front * deltaTime;
            break;
        case LEFT:
            positionTemp += right * deltaTime;
            break;
        case RIGHT:
            positionTemp -= right * deltaTime;
            break;
    }

    if(!this->isCollision(positionTemp)) {
        this->position = positionTemp;
    }
    std::cout<<"deltaTime: "<<deltaTime<<std::endl;
    std::cout<<"position: "<<this->position.x<<" "<<this->position.y<<"
"<<this->position.z<<std::endl;

}

void Camera::processMouseMovement(float xoffset, float yoffset) {
    std :: cout << "xoffset: " << xoffset << " yoffset: " << yoffset << std::endl;
    if(xoffset <= 30.0)
        this->yaw += 1.0f;

    if(xoffset >= glutGet(GLUT_WINDOW_WIDTH) - 30.0)
        this->yaw -= 1.0f;

    float deltaX = xoffset - this->lastX;
    float deltaY = yoffset - this->lastY;
    this->lastX = xoffset;
    this->lastY = yoffset;

    this->yaw -= deltaX / 20.0f;
    this->pitch += deltaY / 20.0f;
    if(this->pitch > 15.0f)
        this->pitch = 15.0f;
    if(this->pitch < -15.0f)
        this->pitch = -15.0f;

    this->updateVectors();
}

void Camera::updateVectors(){
    glm::quat xQuat = glm::angleAxis(glm::radians(this->yaw), glm::vec3(0.0f, 1.0f,
0.0f));
```

```
   glm::quat yQuat = glm::angleAxis(glm::radians(this->pitch), glm::vec3(1.0f, 0.0f,
0.0f));
   this->orientation = glm::normalize(yQuat * xQuat);
}
```

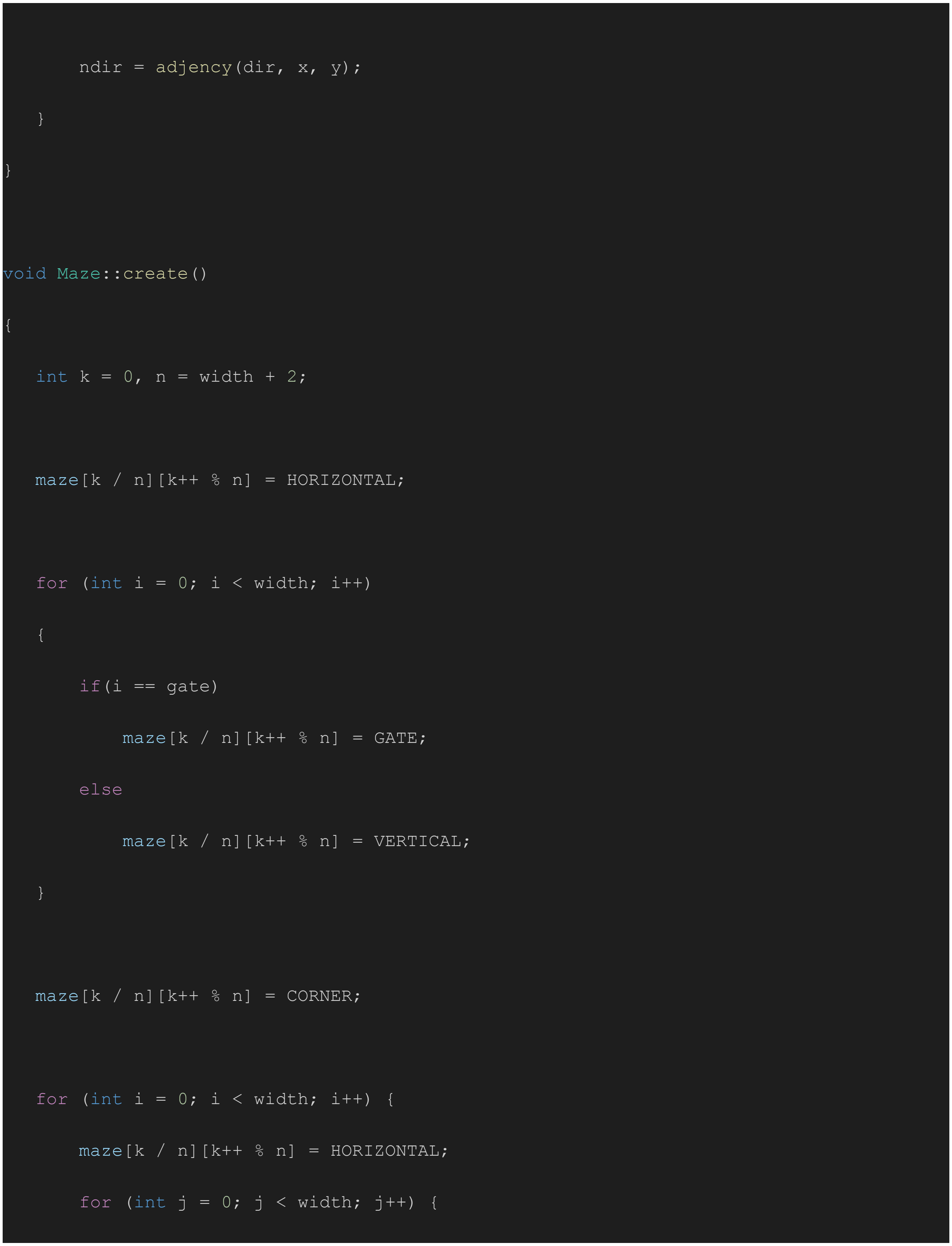
```
EBO::EBO(const GLuint *indices, GLuint count) : indices(indices), count(count)
{
   GLCall(glGenBuffers(1, &indicesBufferId));
   GLCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indicesBufferId));
   GLCall(glBufferData(GL_ELEMENT_ARRAY_BUFFER, count * sizeof(GLuint), indices,
GL_STATIC_DRAW));
}


EBO::~EBO(){
   GLCall(glDeleteBuffers(1, &indicesBufferId));
}
void EBO::bind() const{
   GLCall(glBindBuffer(GL_ARRAY_BUFFER, indicesBufferId));
}
void EBO::unbind() const{
   GLCall(glBindBuffer(GL_ARRAY_BUFFER, 0));
}
```

```cpp
#define ASSERT(x) if (!(x)) raise(SIGTRAP);
#define GLCall(x) GLClearError();\
   x;\
   ASSERT(GLLogCall(#x, __FILE__, __LINE__))

void GLClearError();

bool GLLogCall(const char* function, const char* file, int line);

void GLClearError()
{
   while (glGetError() != GL_NO_ERROR);
}

bool GLLogCall(const char* function, const char* file, int line)
{
   if (GLenum error = glGetError())
   {
       std::cout << "[OpenGL Error] (" << error << "): " << function << " " << file << ":" << line << std::endl;
       return false;
   }
   return true;
}
```

```
enum {

    North,

    East,

    South,

    West,

    NDir

};


Maze::Maze(int width)

{

    this->gate = rand() % width - 2;

    srand(time(NULL));

    this->width = width;

    visited = new int *[width];

    verticalWalls = new int *[width];

    horizontalWalls = new int *[width];

    maze = new WALL *[width + 2];

    for (int i = 0; i < width; i++)

    {

        visited[i] = new int[width];

        verticalWalls[i] = new int[width];

        horizontalWalls[i] = new int[width];

        for (int j = 0; j < width; j++)
        {
            visited[i][j] = 0;
        }
    }

    for (int i = 0; i < width + 2; i++)
    {
        maze[i] = new WALL[width + 2];
    }
}

Maze::~Maze()
{
    for (int i = 0; i < width; i++)
    {
        delete[] visited[i];
        delete[] verticalWalls[i];
        delete[] horizontalWalls[i];
    }
    for (int i = 0; i < width + 2; i++)
    {
        delete[] maze[i];
    }
    delete[] maze;
    delete[] visited;
```

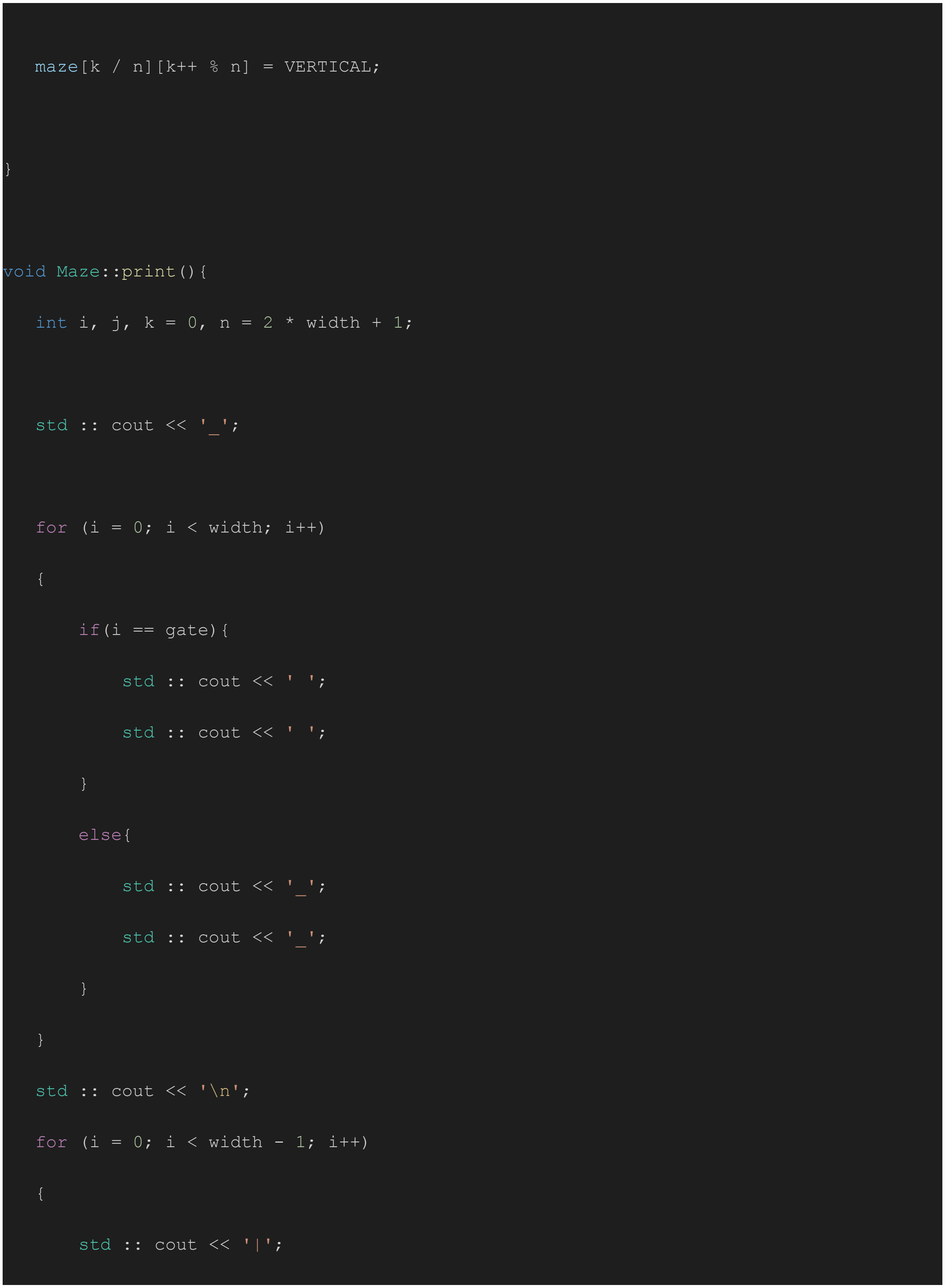
```
   delete[] verticalWalls;
   delete[] horizontalWalls;
}


int Maze::adjency(int dir[], int x, int y)
{
   int ndir = 0;

   if (y > 0 && visited[y - 1][x] == 0)
       dir[ndir++] = North;
   if (x < width - 1 && visited[y][x + 1] == 0)
       dir[ndir++] = East;
   if (y < width - 1 && visited[y + 1][x] == 0)
       dir[ndir++] = South;
   if (x > 0 && visited[y][x - 1] == 0)
       dir[ndir++] = West;

   return ndir;
}

void Maze::generate(int x, int y)
{
   int dir[NDir];
   int ndir;
```

```
visited[y][x] = 1;
ndir = adjency(dir, x, y);
while (ndir)
{
    int nextDir = rand() % ndir;

    switch (dir[nextDir])
    {
    case North:
        verticalWalls[y - 1][x] = 1;
        generate(x, y - 1);
        break;
    case East:
        horizontalWalls[y][x] = 1;
        generate(x + 1, y);
        break;
    case South:
        verticalWalls[y][x] = 1;
        generate(x, y + 1);
        break;
    case West:
        horizontalWalls[y][x - 1] = 1;
        generate(x - 1, y);
        break;
    }
```

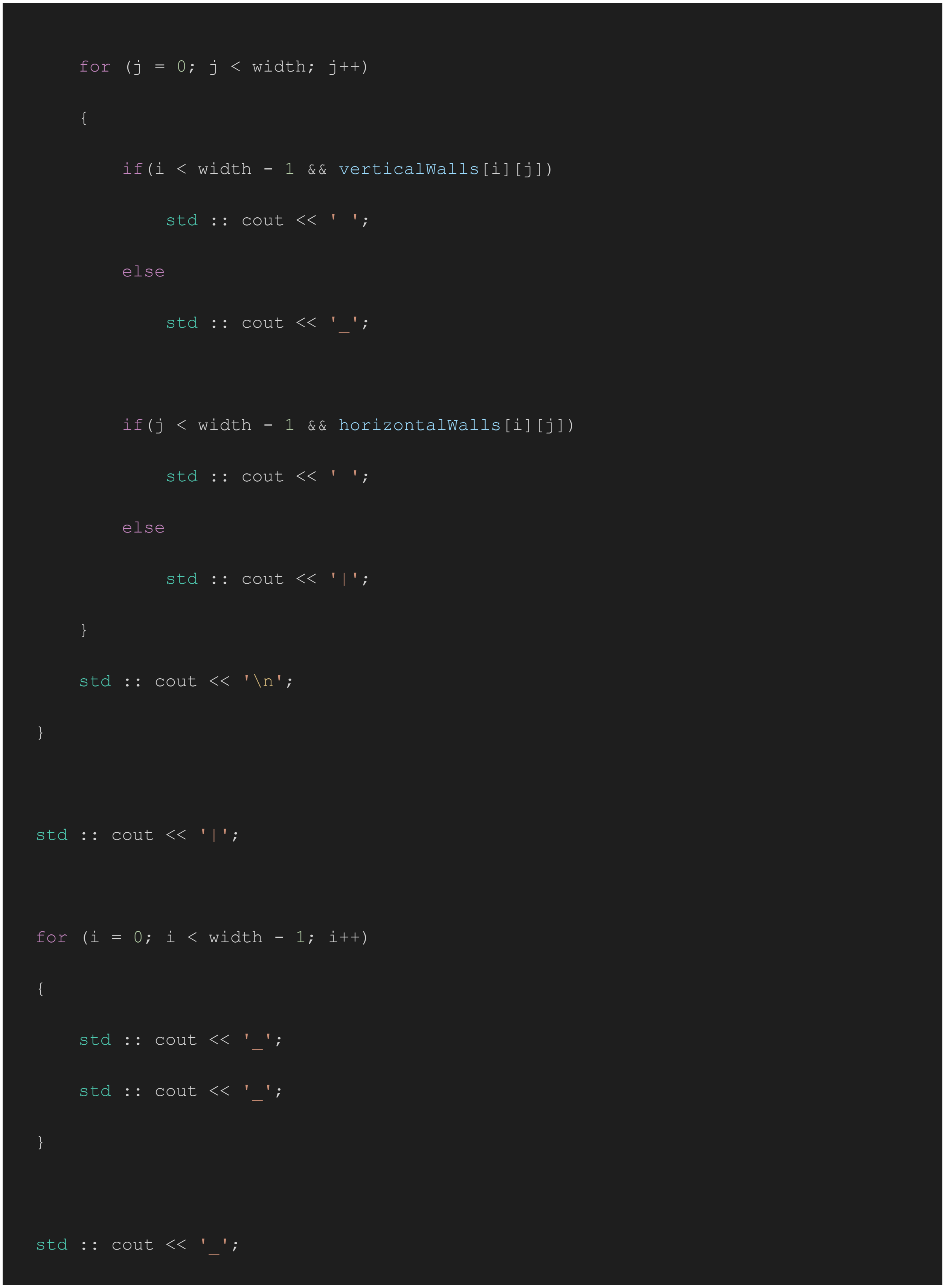


```
        ndir = adjency(dir, x, y);
    }
}

void Maze::create()
{
    int k = 0, n = width + 2;

    maze[k / n][k++ % n] = HORIZONTAL;

    for (int i = 0; i < width; i++)
    {
        if(i == gate)
            maze[k / n][k++ % n] = GATE;
        else
            maze[k / n][k++ % n] = VERTICAL;
    }

    maze[k / n][k++ % n] = CORNER;

    for (int i = 0; i < width; i++) {
        maze[k / n][k++ % n] = HORIZONTAL;
        for (int j = 0; j < width; j++) {
```

```
        std :: cout << "i = " << i << " j = " << j << " k = " << k << " val:" << maze[(k-1) / n][ (k-1) % n ]<<std :: endl;
        if (horizontalWalls[i][j] == 1) {
            maze[k / n][k++ % n] = HORIZONTAL;
            continue;
        }
        if (verticalWalls[i][j] == 1) {
            maze[k / n][k++ % n] = VERTICAL;
            continue;
        }
        if (verticalWalls[i][j] == 1 && horizontalWalls[i][j] == 1) {
            maze[k / n][k++ % n] = CORNER;
            continue;
        }
        maze[k / n][k++ % n] = GATE;
    }
    maze[k / n][k++ % n] = HORIZONTAL;
}

maze[k / n][k++ % n] = CUBE;

for (int i = 0; i < width; i++)
{
    maze[k / n][k++ % n] = VERTICAL;
}
```



```
    maze[k / n][k++ % n] = VERTICAL;

}

void Maze::print(){
    int i, j, k = 0, n = 2 * width + 1;

    std :: cout << '_';

    for (i = 0; i < width; i++)
    {
        if(i == gate){
            std :: cout << ' ';
            std :: cout << ' ';
        }
        else{
            std :: cout << '_';
            std :: cout << '_';
        }
    }
    std :: cout << '\n';
    for (i = 0; i < width - 1; i++)
    {
        std :: cout << '|';
```



```
    for (j = 0; j < width; j++)
    {
        if(i < width - 1 && verticalWalls[i][j])
            std :: cout << ' ';
        else
            std :: cout << '_';

        if(j < width - 1 && horizontalWalls[i][j])
            std :: cout << ' ';
        else
            std :: cout << '|';
    }
    std :: cout << '\n';
}

std :: cout << '|';

for (i = 0; i < width - 1; i++)
{
    std :: cout << '_';
    std :: cout << '_';
}

std :: cout << '_';
```

```
   std :: cout << '|';

   std :: cout << '\n';

}
```

```
Renderer::Renderer(GLfloat *vertices, GLuint *indices, int verticesSize, int
indicesSize)

{

   this->shader = new Shader("resource/shader.vert", "resource/shader.frag");

   this->vao = new VAO();

   this->vbo = new VBO(vertices, verticesSize);

   this->ebo = new EBO(indices, indicesSize);

   vao->addBufferVec4(*vbo, false);

}



Renderer::Renderer(VAO *vao, VBO *vbo, EBO *ebo, Shader *shader)

{

   this->vao = vao;

   this->vbo = vbo;

   this->ebo = ebo;

   this->shader = shader;

}



Renderer::~Renderer(){

   delete this->vao;

   delete this->vbo;
```

```
    delete this->ebo;
}


void Renderer::instance(int **map, int x, int y, int distance, glm::mat4 transform){

    this->instanceCount = 0;
    for(int i = 0; i < x; i++)
        for(int j = 0; j < y; j++)
            if(map[i][j])
                this->instanceCount++;

    glm::vec4 colors[instanceCount];
    glm::vec2 texture[instanceCount];
    glm::mat4 matModel[instanceCount];

    for (int n = 0; n < instanceCount; n++)
    {
        float a = float(n) / 4.0f;
        float b = float(n) / 5.0f;
        float c = float(n) / 6.0f;
        colors[n][0] = 0.35f + 0.30f * (sinf(a + 2.0f) + 1.0f);
        colors[n][1] = 0.25f + 0.25f * (sinf(b + 3.0f) + 1.0f);
        colors[n][2] = 0.25f + 0.35f * (sinf(c + 4.0f) + 1.0f);
        colors[n][3] = 1.0f;
    }
    int k = 0;
    texture[0] = glm::vec3(1.0f, 1.0f, 0.0f);
    texture[1] = glm::vec3(0.0f, 1.0f, 0.0f);
    texture[2] = glm::vec3(0.0f, 0.0f, 0.0f);
    texture[3] = glm::vec3(1.0f, 0.0f, 0.0f);
    texture[4] = glm::vec3(0.0f, 0.0f, 0.0f);
    texture[5] = glm::vec3(1.0f, 0.0f, 0.0f);
    texture[6] = glm::vec3(1.0f, 1.0f, 0.0f);
    texture[7] = glm::vec3(0.0f, 1.0f, 0.0f);

    for(int i = 0; i < x; i++)
        for(int j = 0; j < y; j++)
            if(map[i][j] == 1){
                matModel[k++] = glm::translate(glm::mat4(1.0f), glm::vec3(distance * i, 0.0f, distance * j)) * transform;
        }

    instanceVBO = new VBO(matModel, sizeof(matModel));
    colorVBO = new VBO(colors, sizeof(colors));
    textureVBO = new VBO(texture, sizeof(texture));
    vao->addBufferVec4(*colorVBO, true);
    vao->addBufferVec2(*textureVBO);
    vao->addBufferMat4(*instanceVBO);
}
```

```
void Renderer::draw(glm::mat4 viewMatrix, glm::mat4 projectionMatrix){
   vao->bind();
   vbo->bind();
   ebo->bind();
   int codCol = 0;
   shader->bind();
   shader->setInt("codCol", codCol);
   shader->setMat4("viewMatrix", viewMatrix);
   shader->setMat4("projectionMatrix", projectionMatrix);
   glDrawElements(GL_TRIANGLES, ebo->getCount(), GL_UNSIGNED_INT, 0);
   vao->unbind();
}

void Renderer::drawInstanced(glm::mat4 viewMatrix, glm::mat4 projectionMatrix, int codCol){
   vao->bind();
   vbo->bind();
   colorVBO->bind();
   instanceVBO->bind();
   ebo->bind();
   shader->bind();
   int texture = 0;
   shader->setInt("tex_Unit", texture);
   shader->setInt("codCol", codCol);
```

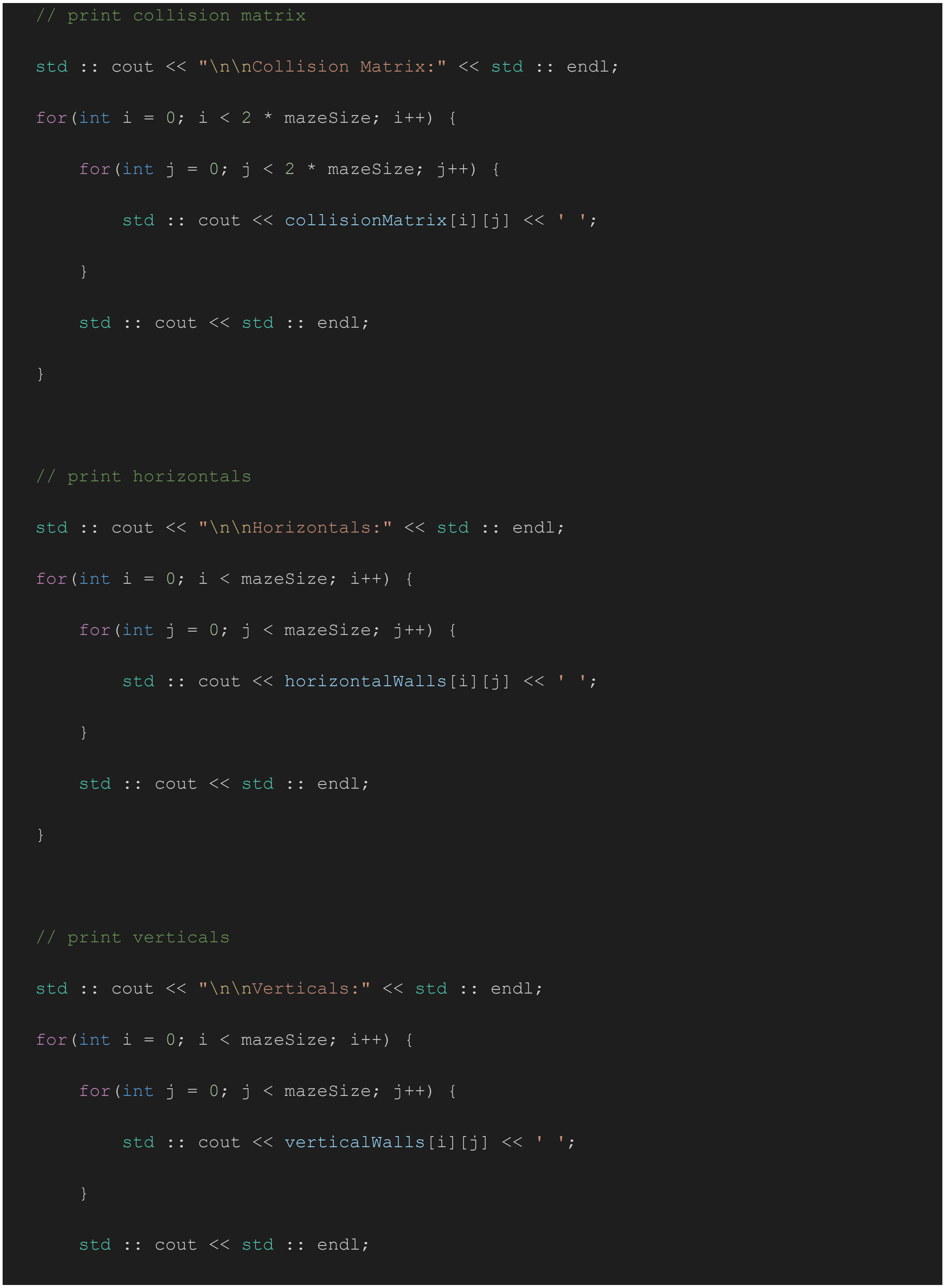
```
    shader->setMat4("viewMatrix", viewMatrix);
    shader->setMat4("projectionMatrix", projectionMatrix);
    glDrawElementsInstanced(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0, instanceCount);
}
```

```
Scene::Scene() { }
Scene::~Scene() { }

Scene* Scene::instance = nullptr;
Scene* Scene::getInstance() {
    if (instance == nullptr) {
        instance = new Scene();
    }
    return instance;
}

void Scene::renderWrapper() {
    Scene::getInstance()->render();
}

void Scene::normalKeyWrapper(unsigned char key, int x, int y) {
    Scene::getInstance()->processNormalKeys(key, x, y);
}

void Scene::specialKeyWrapper(int key, int x, int y) {
    Scene::getInstance()->processSpecialKeys(key, x, y);
}

void Scene::mouseMoveWrapper(int x, int y) {
    Scene::getInstance()->processMouseMovement(x, y);
}

void Scene::cleanupWrapper() {
    Scene::getInstance()->cleanup();
}

void Scene::start(int *argc, char **argv) {
    // set the display mode
    glutInitDisplayMode(GLUT_3_2_CORE_PROFILE | GLUT_SINGLE| GLUT_RGB | GLUT_DEPTH);
    // initialize glut
    glutInit(argc, argv);
    // create the window
    glutInitWindowSize(800, 600);
    glutCreateWindow("Labyrinth");
    glutFullScreen();
    // initialize the program
    this->init();
    // set the display function
    GLCall(glutDisplayFunc(renderWrapper));
    // set the reshape function
    GLCall(glutIdleFunc(renderWrapper));

    // set the keyboard function

    GLCall(glutKeyboardFunc(normalKeyWrapper));

    // set the special keyboard function

    GLCall(glutSpecialFunc(specialKeyWrapper));

    // set the mouse function

    GLCall(glutPassiveMotionFunc(mouseMoveWrapper));

    // set the menu function

    GLCall(glutCreateMenu(NULL));

    // set close function

    GLCall(glutWMCloseFunc(cleanupWrapper));

    // main loop

    GLCall(glutMainLoop());

}



// initialize the program

void Scene::init(void)

{

    GLCall(glutSetCursor(GLUT_CURSOR_NONE));

    const GLfloat PI = 3.141592;

    // set the background color white

    GLCall(glClearColor(1.0f, 1.0f, 1.0f, 0.0f));

    int mazeSize = 8, padding = 10, floorSize;

    maze = new Maze(mazeSize);

    maze->generate(0, 0);
```

```
   mazeSize += 2;
   maze->create();
   maze->print();
   WALL **mazeArray = maze->getMaze();
   int **corners, **horizontalWalls, **verticalWalls, **cubes, **floorArray, **collisionMatrix;

   floorArray = new int*[1];
   floorArray[0] = new int[1];
   floorArray[0][0] = 1;

   corners = new int*[mazeSize];
   horizontalWalls = new int*[mazeSize];
   verticalWalls = new int*[mazeSize];
   collisionMatrix = new int*[2 * mazeSize];
   cubes = new int*[mazeSize];
   for (int i = 0; i < 2 * mazeSize ; i++) {
       collisionMatrix[i] = new int[2 * mazeSize];
   }
   for (int i = 0; i < mazeSize; i++) {
       corners[i] = new int[mazeSize];
       horizontalWalls[i] = new int[mazeSize];
       verticalWalls[i] = new int[mazeSize];
       cubes[i] = new int[mazeSize];
   }
```

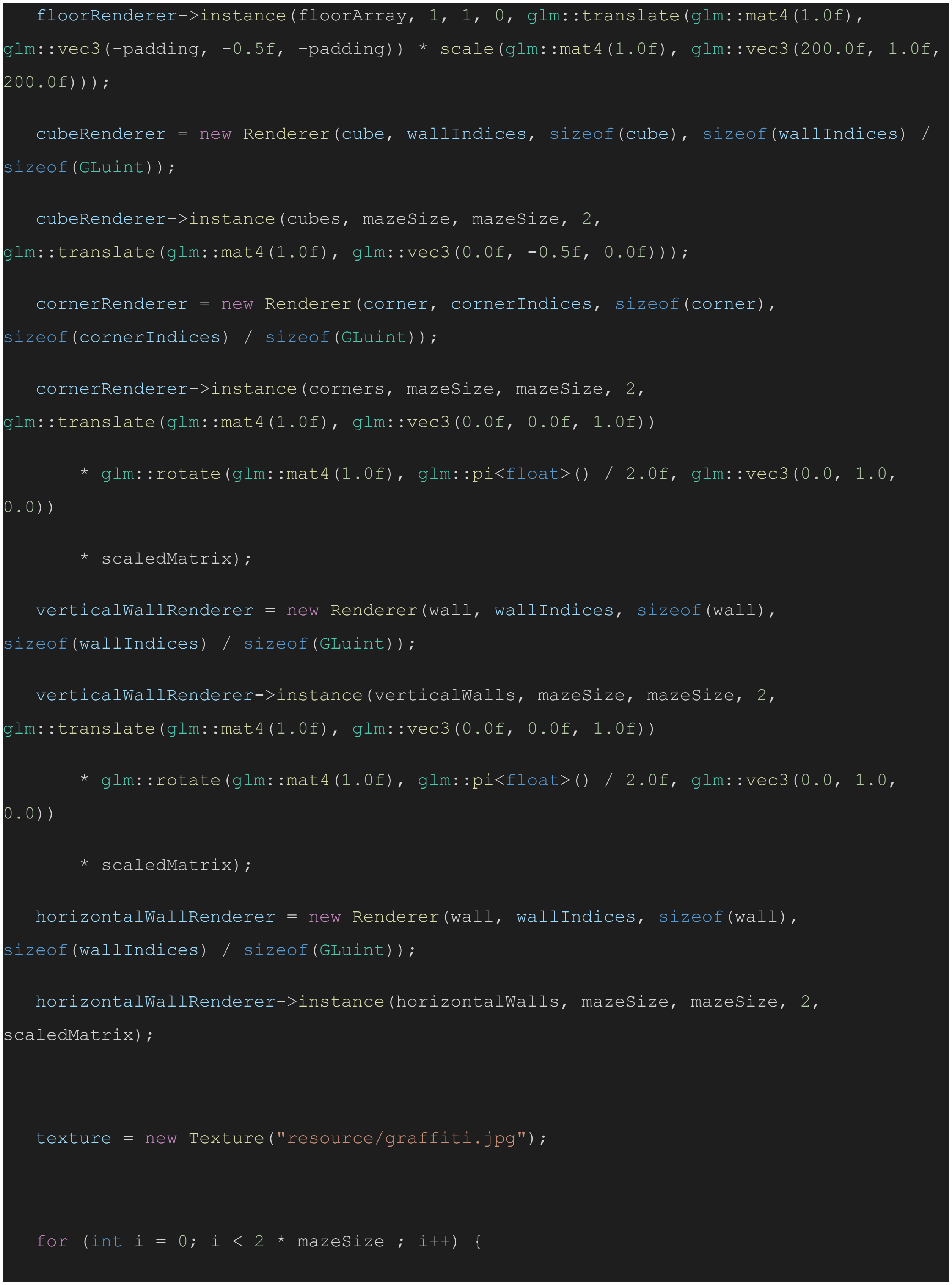
```
for (int i = 0; i < 2 * mazeSize ; i++)
    for (int j = 0; j < 2 * mazeSize ; j++)
        collisionMatrix[i][j] = 0;

for(int i = 0; i < mazeSize; i++) {
    for(int j = 0; j < mazeSize; j++) {
        corners[i][j] = 0;
        horizontalWalls[i][j] = 0;
        verticalWalls[i][j] = 0;
        cubes[i][j] = 0;
    }
}

for(int i = 0; i < mazeSize; i++) {
    for(int j = 0; j < mazeSize; j++) {
        std :: cout << mazeArray[i][j] << ' ';
        switch(mazeArray[i][j]){
        case HORIZONTAL:
            horizontalWalls[i][j] = 1;
            collisionMatrix[2 * i][2 * j] = 1;
            if(i != 0) {
                collisionMatrix[2 * i - 1][2 * j] = 1;
            }
            else
```

```
                collisionMatrix[2 * i][2 * j + 1] = 1;

            break;

        case VERTICAL:

            verticalWalls[i][j] = 1;

            collisionMatrix[2 * i][2 * j] = 1;

            collisionMatrix[2 * i][2 * j + 1] = 1;

        case CORNER:

            corners[i][j] = 1;

            collisionMatrix[2 * i][2 * j] = 1;

            collisionMatrix[2 * i + 1][2 * j] = 1;

            collisionMatrix[2 * i + 1][2 * j + 1] = 1;

            break;

        case CUBE:

            cubes[i][j] = 1;

            collisionMatrix[2 * i][2 * j] = 1;

            collisionMatrix[2 * i + 1][2 * j] = 1;

            collisionMatrix[2 * i + 1][2 * j + 1] = 1;

            break;

        default:

            break;

        }

    }

    std :: cout << std :: endl;

}
```



```
// print collision matrix
std :: cout << "\n\nCollision Matrix:" << std :: endl;
for(int i = 0; i < 2 * mazeSize; i++) {
    for(int j = 0; j < 2 * mazeSize; j++) {
        std :: cout << collisionMatrix[i][j] << ' ';
    }
    std :: cout << std :: endl;
}


// print horizontals
std :: cout << "\n\nHorizontals:" << std :: endl;
for(int i = 0; i < mazeSize; i++) {
    for(int j = 0; j < mazeSize; j++) {
        std :: cout << horizontalWalls[i][j] << ' ';
    }
    std :: cout << std :: endl;
}


// print verticals
std :: cout << "\n\nVerticals:" << std :: endl;
for(int i = 0; i < mazeSize; i++) {
    for(int j = 0; j < mazeSize; j++) {
        std :: cout << verticalWalls[i][j] << ' ';
    }
    std :: cout << std :: endl;
```

```
}


GLfloat wall[] =
{
    0.0f,  0.0f, 0.0f, 1.0f,
    1.0f,  0.0f, 0.0f, 1.0f,
    1.0f,  1.0f, 0.0f, 1.0f,
    0.0f,  1.0f, 0.0f, 1.0f,

    0.0f,  0.0f, 0.5f, 1.0f,
    1.0f,  0.0f, 0.5f, 1.0f,
    1.0f,  1.0f, 0.5f, 1.0f,
    0.0f,  1.0f, 0.5f, 1.0f,
};

GLfloat cube[] =
{
    0.0f,  0.0f, 0.0f, 1.0f,
    1.0f,  0.0f, 0.0f, 1.0f,
    1.0f,  1.0f, 0.0f, 1.0f,
    0.0f,  1.0f, 0.0f, 1.0f,

    0.0f,  0.0f, 1.0f, 1.0f,
    1.0f,  0.0f, 1.0f, 1.0f,
    1.0f,  1.0f, 1.0f, 1.0f,
    0.0f,  1.0f, 1.0f, 1.0f,
};

GLfloat corner[] =
{
    0.0f,  0.0f, 0.0f, 1.0f,
    1.0f,  0.0f, 0.0f, 1.0f,
    1.0f,  1.0f, 0.0f, 1.0f,
    0.0f,  1.0f, 0.0f, 1.0f,

    0.0f,  0.0f, 0.5f, 1.0f, // 0
    1.0f,  0.0f, 0.5f, 1.0f,
    1.0f,  1.0f, 0.5f, 1.0f,
    0.0f,  1.0f, 0.5f, 1.0f, // 3

    0.5f,  0.0f, 0.5f, 1.0f, // 1
    0.5f,  1.0f, 0.5f, 1.0f, // 2

    0.0f,  0.0f, 1.0f, 1.0f, // 4
    0.5f,  0.0f, 1.0f, 1.0f, // 5
    0.5f,  1.0f, 1.0f, 1.0f, // 6
    0.0f,  1.0f, 1.0f, 1.0f, // 7
};

GLfloat floor[] =
{
    0.0f,  -0.25f, 0.0f, 1.0f,
    1.0f,  -0.25f, 0.0f, 1.0f,
    1.0f,  0.0f, 0.0f, 1.0f,
    0.0f,  0.0f, 0.0f, 1.0f,

    0.0f,  -0.25f, 1.0f, 1.0f,
    1.0f,  -0.25f, 1.0f, 1.0f,
    1.0f,  0.0f, 1.0f, 1.0f,
    0.0f,  0.0f, 1.0f, 1.0f,
};

GLuint wallIndices[] =
{
    1, 2, 0,   0, 2, 3,
    2, 3, 6,   6, 3, 7,
    7, 3, 4,   4, 3, 0,
    4, 0, 5,   5, 0, 1,
    1, 2, 5,   5, 2, 6,
    5, 6, 4,   4, 6, 7,
};

// indicii pentru varfuri
GLuint cornerIndices[] =
{
       1, 2, 0,   0, 2, 3,
       2, 3, 6,   6, 3, 7,
       7, 3, 4,   4, 3, 0,
       4, 0, 5,   5, 0, 1,
       1, 2, 5,   5, 2, 6,
       5, 6, 4,   4, 6, 7,

       8, 9, 4,   4, 9, 7,
       9, 7, 12,   12, 7, 13,
       13, 7, 10,   10, 7, 4,
       10, 4, 11,   11, 4, 8,
       8, 9, 11,   11, 9, 12,
       11, 12, 10,   10, 12, 13,
   };



   skybox = new Skybox();
   camera = new Camera(glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(1.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
   projectionMatrix = glm::infinitePerspective(PI / 2.0f, (float) glutGet(GLUT_WINDOW_WIDTH) / glutGet(GLUT_WINDOW_HEIGHT), 0.01f);


   glm::mat4 scaledMatrix = glm::scale(glm::mat4(1.0f), glm::vec3(2.0f, 1.0f, 2.0f)) * glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, -0.5f, 0.0f));
   floorRenderer = new Renderer(floor, wallIndices, sizeof(floor), sizeof(wallIndices) / sizeof(GLuint));
```



```
   floorRenderer->instance(floorArray, 1, 1, 0, glm::translate(glm::mat4(1.0f),
glm::vec3(-padding, -0.5f, -padding)) * scale(glm::mat4(1.0f), glm::vec3(200.0f, 1.0f,
200.0f)));
   cubeRenderer = new Renderer(cube, wallIndices, sizeof(cube), sizeof(wallIndices) /
sizeof(GLuint));
   cubeRenderer->instance(cubes, mazeSize, mazeSize, 2,
glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, -0.5f, 0.0f)));
   cornerRenderer = new Renderer(corner, cornerIndices, sizeof(corner),
sizeof(cornerIndices) / sizeof(GLuint));
   cornerRenderer->instance(corners, mazeSize, mazeSize, 2,
glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, 1.0f))
       * glm::rotate(glm::mat4(1.0f), glm::pi<float>() / 2.0f, glm::vec3(0.0, 1.0,
0.0))
       * scaledMatrix);
   verticalWallRenderer = new Renderer(wall, wallIndices, sizeof(wall),
sizeof(wallIndices) / sizeof(GLuint));
   verticalWallRenderer->instance(verticalWalls, mazeSize, mazeSize, 2,
glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, 1.0f))
       * glm::rotate(glm::mat4(1.0f), glm::pi<float>() / 2.0f, glm::vec3(0.0, 1.0,
0.0))
       * scaledMatrix);
   horizontalWallRenderer = new Renderer(wall, wallIndices, sizeof(wall),
sizeof(wallIndices) / sizeof(GLuint));
   horizontalWallRenderer->instance(horizontalWalls, mazeSize, mazeSize, 2,
scaledMatrix);

   texture = new Texture("resource/graffiti.jpg");

   for (int i = 0; i < 2 * mazeSize ; i++) {
       delete collisionMatrix[i];
   }
   for (int i = 0; i < mazeSize; i++) {
       delete corners[i];
       delete horizontalWalls[i];
       delete verticalWalls[i];
       delete cubes[i];
   }
   delete floorArray[0];

   delete corners;
   delete horizontalWalls;
   delete verticalWalls;
   delete cubes;
   delete floorArray;
   delete collisionMatrix;
}

// render the program
void Scene::render(void)
{
   GLCall(glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT));
   viewMatrix = camera->getViewMatrix();
   skybox->draw(viewMatrix, projectionMatrix);
   texture->bind();
```

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

   floorRenderer->drawInstanced(viewMatrix, projectionMatrix, 1);
   cubeRenderer->drawInstanced(viewMatrix, projectionMatrix, 1);
   cornerRenderer->drawInstanced(viewMatrix, projectionMatrix, 1);
   verticalWallRenderer->drawInstanced(viewMatrix, projectionMatrix, 1);
   horizontalWallRenderer->drawInstanced(viewMatrix, projectionMatrix, 1);

   GLCall(glDisable(GL_BLEND));
   GLCall(glutSwapBuffers());
   GLCall(glFlush());

}



// process normal keys
void Scene::processNormalKeys(unsigned char key, int x, int y)
{
   float cameraSpeed = 0.1f;
   if(key == 'w')
   {
       camera->processKeyboard(UP, cameraSpeed);
   }
   else if(key == 's')
   {
      camera->processKeyboard(DOWN, cameraSpeed);
   }
   else if(key == 'a')
   {
      camera->processKeyboard(LEFT, cameraSpeed);
   }
   else if(key == 'd')
   {
      camera->processKeyboard(RIGHT, cameraSpeed);
   }
}

// process special keys
void Scene::processSpecialKeys(int key, int x, int y)
{
   float cameraSpeed = 0.1f;
   if(key == 27)
   {
       exit(0);
   }
   if(key == GLUT_KEY_UP)
   {
      camera->processKeyboard(UP, cameraSpeed);
   }
   else if(key == GLUT_KEY_DOWN)
   {
      camera->processKeyboard(DOWN, cameraSpeed);
   }
   else if(key == GLUT_KEY_LEFT)
   {
      camera->processKeyboard(LEFT, cameraSpeed);
   }
   else if(key == GLUT_KEY_RIGHT)
   {
      camera->processKeyboard(RIGHT, cameraSpeed);
   }
}

void Scene::processMouseMovement(int x, int y)
{
   camera->processMouseMovement(x, y);
}

// cleanup the program
void Scene::cleanup(void)
{
```

```cpp
   delete camera;
   delete verticalWallRenderer;
   delete horizontalWallRenderer;
   delete cornerRenderer;
   delete cubeRenderer;
   delete floorRenderer;
   delete texture;
   delete skybox;
}
```

```cpp
std::string Shader::readFile(const char *filePath) {
   std::string content;
   std::ifstream fileStream(filePath, std::ios::in);

   if(!fileStream.is_open()) {
       std::cerr << "Could not read file " << filePath << ". File does not exist." << std::endl;
       return "";
   }

   std::string line = "";
   while(!fileStream.eof()) {
       std::getline(fileStream, line);
       content.append(line + "\n");
   }
```

```
    fileStream.close();

    return content;

}




GLuint Shader::loadShaders(const char *vertex_path, const char *fragment_path) {

    GLCall(GLuint vertShader = glCreateShader(GL_VERTEX_SHADER));

    GLCall(GLuint fragShader = glCreateShader(GL_FRAGMENT_SHADER));


    // Read shaders


    std::string vertShaderStr = readFile(vertex_path);

    std::string fragShaderStr = readFile(fragment_path);

    const char *vertShaderSrc = vertShaderStr.c_str();

    const char *fragShaderSrc = fragShaderStr.c_str();


    GLint result = GL_FALSE;

    int logLength;


    // Compile vertex shader


    std::cout << "Compiling vertex shader." << std::endl;

    GLCall(glShaderSource(vertShader, 1, &vertShaderSrc, NULL));

    GLCall(glCompileShader(vertShader));
```

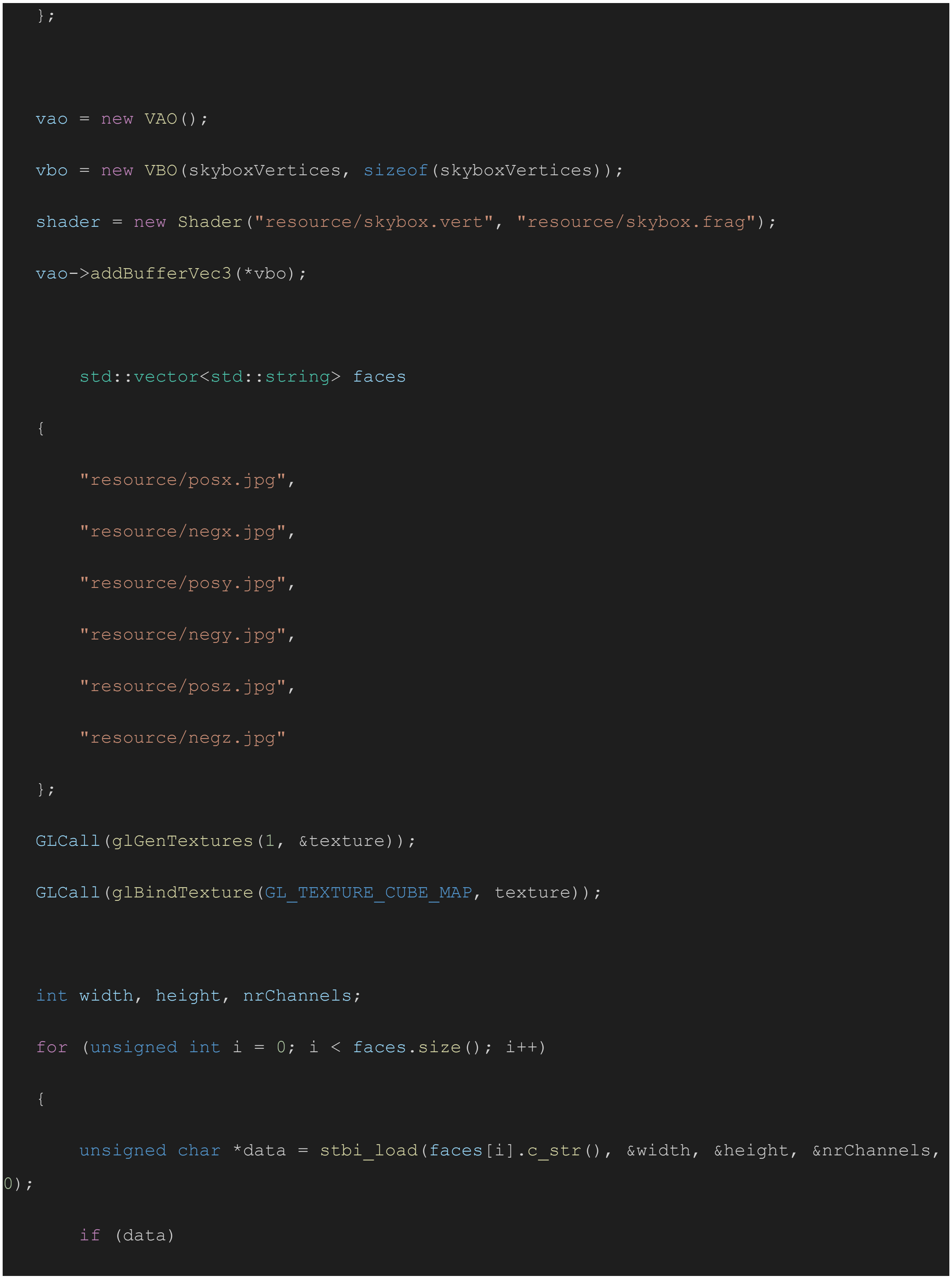
```
// Check vertex shader

GLCall(glGetShaderiv(vertShader, GL_COMPILE_STATUS, &result));
GLCall(glGetShaderiv(vertShader, GL_INFO_LOG_LENGTH, &logLength));
std::vector<char> vertShaderError((logLength > 1) ? logLength : 1);
GLCall(glGetShaderInfoLog(vertShader, logLength, NULL, &vertShaderError[0]));
std::cout << &vertShaderError[0] << std::endl;

// Compile fragment shader

std::cout << "Compiling fragment shader." << std::endl;
GLCall(glShaderSource(fragShader, 1, &fragShaderSrc, NULL));
GLCall(glCompileShader(fragShader));

// Check fragment shader

GLCall(glGetShaderiv(fragShader, GL_COMPILE_STATUS, &result));
GLCall(glGetShaderiv(fragShader, GL_INFO_LOG_LENGTH, &logLength));
std::vector<char> fragShaderError((logLength > 1) ? logLength : 1);
GLCall(glGetShaderInfoLog(fragShader, logLength, NULL, &fragShaderError[0]));
std::cout << &fragShaderError[0] << std::endl;

std::cout << "Linking program" << std::endl;
GLCall(GLuint program = glCreateProgram());
```

```
    GLCall(glAttachShader(program, vertShader));
    GLCall(glAttachShader(program, fragShader));
    GLCall(glLinkProgram(program));

    GLCall(glGetProgramiv(program, GL_LINK_STATUS, &result));
    GLCall(glGetProgramiv(program, GL_INFO_LOG_LENGTH, &logLength));
    std::vector<char> programError( (logLength > 1) ? logLength : 1 );
    GLCall(glGetProgramInfoLog(program, logLength, NULL, &programError[0]));
    std::cout << &programError[0] << std::endl;
    if(logLength > 0) {
        std::cout << "Vertex shader compilation failed." << std::endl;
        raise(SIGTRAP);
        exit(0);
    }

    GLCall(glDeleteShader(vertShader));
    GLCall(glDeleteShader(fragShader));
    this->program = program;

    return program;
}


Shader::Shader(const char *vertex_path, const char *fragment_path) : program(0) {
    program = loadShaders(vertex_path, fragment_path);
```

```
   GLCall(glUseProgram(program));
}


Shader::~Shader(){
   GLCall(glDeleteProgram(program));
}
void Shader::setInt(const char *name, int value) const {
   unsigned int location = glGetUniformLocation(program, name);
   GLCall(glUniform1i(location, value));
}
void Shader::setFloat(const char *name, float value) const {
   unsigned int location = glGetUniformLocation(program, name);
   GLCall(glUniform1f(location, value));
}
void Shader::setVec2(const char *name, glm::vec2 value) const {
   unsigned int location = glGetUniformLocation(program, name);
   GLCall(glUniform2fv(location, GL_FALSE, &value[0]));
}
void Shader::setMat4(const char *name, glm::mat4 value) const {
   unsigned int location = glGetUniformLocation(program, name);
   GLCall(glUniformMatrix4fv(location, 1, GL_FALSE, &value[0][0]));
}


void Shader::bind() const {
   GLCall(glUseProgram(program));
```

```
}


void Shader::unbind() const {
   GLCall(glUseProgram(0));
}
```

```
Skybox::Skybox() {
   float skyboxVertices[] = {
   // positions
   -1.0f,  1.0f, -1.0f,
   -1.0f, -1.0f, -1.0f,
    1.0f, -1.0f, -1.0f,
    1.0f, -1.0f, -1.0f,
    1.0f,  1.0f, -1.0f,
   -1.0f,  1.0f, -1.0f,

   -1.0f, -1.0f,  1.0f,
   -1.0f, -1.0f, -1.0f,
   -1.0f,  1.0f, -1.0f,
   -1.0f,  1.0f, -1.0f,
   -1.0f,  1.0f,  1.0f,
   -1.0f, -1.0f,  1.0f,

    1.0f, -1.0f, -1.0f,
    1.0f, -1.0f,  1.0f,
 1.0f,  1.0f,  1.0f,
 1.0f,  1.0f,  1.0f,
 1.0f,  1.0f, -1.0f,
 1.0f, -1.0f, -1.0f,

-1.0f, -1.0f,  1.0f,
-1.0f,  1.0f,  1.0f,
 1.0f,  1.0f,  1.0f,
 1.0f,  1.0f,  1.0f,
 1.0f, -1.0f,  1.0f,
-1.0f, -1.0f,  1.0f,

-1.0f,  1.0f, -1.0f,
 1.0f,  1.0f, -1.0f,
 1.0f,  1.0f,  1.0f,
 1.0f,  1.0f,  1.0f,
-1.0f,  1.0f,  1.0f,
-1.0f,  1.0f, -1.0f,

-1.0f, -1.0f, -1.0f,
-1.0f, -1.0f,  1.0f,
 1.0f, -1.0f, -1.0f,
 1.0f, -1.0f, -1.0f,
-1.0f, -1.0f,  1.0f,
 1.0f, -1.0f,  1.0f
```



```
    };


    vao = new VAO();
    vbo = new VBO(skyboxVertices, sizeof(skyboxVertices));
    shader = new Shader("resource/skybox.vert", "resource/skybox.frag");
    vao->addBufferVec3(*vbo);


        std::vector<std::string> faces
    {
        "resource/posx.jpg",
        "resource/negx.jpg",
        "resource/posy.jpg",
        "resource/negy.jpg",
        "resource/posz.jpg",
        "resource/negz.jpg"
    };
    GLCall(glGenTextures(1, &texture));
    GLCall(glBindTexture(GL_TEXTURE_CUBE_MAP, texture));


    int width, height, nrChannels;
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        unsigned char *data = stbi_load(faces[i].c_str(), &width, &height, &nrChannels,
0);
        if (data)
        {
            GLCall(glTexImage2D(
                GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
                0,
                GL_RGB,
                width,
                height,
                0,
                GL_RGB,
                GL_UNSIGNED_BYTE,
                data
            ));
            stbi_image_free(data);
        }
        else
        {
            std::cout << "Cubemap tex failed to load at path: " << faces[i] <<
std::endl;
            stbi_image_free(data);
        }
    }
    GLCall(glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR));
    GLCall(glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR));
    GLCall(glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE));
    GLCall(glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE));
    GLCall(glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE));
}

Skybox::~Skybox() {
    delete vao;
    delete shader;
}

void Skybox::draw(glm::mat4 viewMatrix, glm::mat4 projectionMatrix) {
    vao->bind();
    vbo->bind();
    shader->bind();
    GLCall(glDepthMask(GL_FALSE));
    GLCall(glBindTexture(GL_TEXTURE_CUBE_MAP, texture));
    GLCall(glGenerateMipmap(GL_TEXTURE_CUBE_MAP));
    shader->setMat4("projection", projectionMatrix);
    glm::mat4 view = glm::mat4(glm::mat3(viewMatrix));
    shader->setMat4("view", view);

    GLCall(glDrawArrays(GL_TRIANGLES, 0, 36));
    GLCall(glDepthMask(GL_TRUE));
}
```

```
Texture::Texture(const char *filePath)
{
    this->texture = loadTexture(filePath);
}

Texture::~Texture()
{
    GLCall(glDeleteTextures(1, &this->texture));
}

void Texture::bind() const
{
    GLCall(glActiveTexture(GL_TEXTURE0));
    GLCall(glBindTexture(GL_TEXTURE_2D, this->texture));
}

void Texture::unbind() const
{
    GLCall(glBindTexture(GL_TEXTURE_2D, 0));
}

unsigned int Texture::loadTexture(const char *filePath){
    int width, height, nrChannels;
    unsigned char *data = stbi_load(filePath, &width, &height, &nrChannels, 0);
    if (data == NULL){
        std::cout << "Failed to load texture" << std::endl;
        raise(SIGTRAP);
        return 0;
    }
    unsigned int texture;
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR));
    GLCall(glGenTextures(1, &texture));
    GLCall(glBindTexture(GL_TEXTURE_2D, texture));
    GLCall(glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data));
    GLCall(glGenerateMipmap(GL_TEXTURE_2D));
    GLCall(glBindTexture(GL_TEXTURE_2D, 0));
    stbi_image_free(data);

    return texture;
}
```

```
VAO::VAO(){
    GLCall(glGenVertexArrays(1, &vao));
    GLCall(glBindVertexArray(vao));
    atribCount = 0;
    size = 0;
}
VAO::~VAO(){
   GLCall(glDisableVertexAttribArray(1));
   GLCall(glDisableVertexAttribArray(0));

   GLCall(glBindBuffer(GL_ARRAY_BUFFER, 0));

   GLCall(glBindVertexArray(0));
   GLCall(glDeleteVertexArrays(1, &vao));
}
void VAO::bind(){
   GLCall(glBindVertexArray(vao));
}
void VAO::unbind(){
   GLCall(glBindVertexArray(0));
}
void VAO::addBufferVec2(VBO& vbo){
   bind();
   vbo.bind();
   GLCall(glEnableVertexAttribArray(atribCount));
   GLCall(glVertexAttribPointer(atribCount, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(GLfloat), 0));
   atribCount++;
}
void VAO::addBufferVec3(VBO& vbo){
   bind();
   vbo.bind();
   GLCall(glEnableVertexAttribArray(atribCount));
   GLCall(glVertexAttribPointer(atribCount, 3, GL_FLOAT, GL_FALSE, 3 *
sizeof(GLfloat), 0));
   atribCount++;
}
void VAO::addBufferVec4(VBO& vbo, bool withDivisor){
   vbo.bind();

   GLCall(glEnableVertexAttribArray(atribCount));
   GLCall(glVertexAttribPointer(atribCount, 4, GL_FLOAT, GL_FALSE, 4 *
sizeof(GLfloat), (GLvoid*)0));
   if (withDivisor){
       GLCall(glVertexAttribDivisor(atribCount, 1));
   }
   atribCount++;
}
void VAO::addBufferMat4(VBO& vbo){
   vbo.bind();

   for (int i = 0; i < 4; i++) // Pentru fiecare coloana
   {
       glEnableVertexAttribArray(atribCount + i);
       glVertexAttribPointer(atribCount + i,              // Location
           4, GL_FLOAT, GL_FALSE,                // vec4
           sizeof(glm::mat4),                    // Stride
           (void*)(sizeof(glm::vec4) * i));      // Start offset
       glVertexAttribDivisor(atribCount + i, 1);
   }

   atribCount += 4;
}
```

```
VBO::VBO(const void *vertices, unsigned int size) : vertices(vertices){
   GLCall(glGenBuffers(1, &verticesBufferId));
   GLCall(glBindBuffer(GL_ARRAY_BUFFER, verticesBufferId));
   GLCall(glBufferData(GL_ARRAY_BUFFER, size, vertices, GL_STATIC_DRAW));
}

VBO::~VBO(){
   GLCall(glDeleteBuffers(1, &verticesBufferId));
}
void VBO::bind() const{
   GLCall(glBindBuffer(GL_ARRAY_BUFFER, verticesBufferId));
}
void VBO::unbind() const{
   GLCall(glBindBuffer(GL_ARRAY_BUFFER, 0));
}
```

```
int main(int argc, char* argv[])

{

   Scene *scene = Scene::getInstance();

   scene->start(&argc, argv);

   delete scene;

   return 0;

}
```

```
#version 410



in vec4 ex_Color;

in vec2 tex_Coord;

out vec4 out_Color;



uniform int codCol;

uniform sampler2D tex_Unit;



void main(void)

{

 if(codCol == 0) {

   out_Color = ex_Color;

 }

 else {

   out_Color = texture(tex_Unit, tex_Coord);

 }
}
```

```
#version 410

layout (location = 0) in vec4 in_Position;
layout (location = 1) in vec4 in_Color;
layout (location = 2) in vec2 texCoord;
layout (location = 3) in mat4 modelMatrix;

out vec4 gl_Position;
out vec4 ex_Color;
out vec2 tex_Coord;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;
void main(void)
{
   gl_Position = projectionMatrix * viewMatrix * modelMatrix * in_Position;
   tex_Coord = vec2(texCoord.x, texCoord.y);
   ex_Color=in_Color;
}
```

```
#version 330 core
out vec4 FragColor;
```

```
in vec3 TexCoords;

uniform samplerCube skybox;

void main()
{
   FragColor = texture(skybox, TexCoords);
}
```

```
#version 330 core
layout (location = 0) in vec3 aPos;

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
   TexCoords = aPos;
   gl_Position = projection * view * vec4(aPos, 1.0);
}
```

```cmake
cmake_minimum_required(VERSION 3.22)
project(labyrinth VERSION 0.1.0)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

include_directories(${CMAKE_SOURCE_DIR}/include)
aux_source_directory(${CMAKE_SOURCE_DIR}/source LIBS)
add_library(lib STATIC ${LIBS})
message(STATUS "include: ${CMAKE_SOURCE_DIR}/include")

set(GLM_INCLUDE_DIRS /opt/homebrew/Cellar/glm/0.9.9.8/include)
message(STATUS "GLM found: ${GLM_INCLUDE_DIRS}")

find_package(GLUT REQUIRED)
message(STATUS "GLUT found: ${GLUT_INCLUDE_DIRS}")

find_package(OpenGL REQUIRED)
message(STATUS "OpenGL found: ${OPENGL_LIBRARIES}")

include_directories(${GLUT_INCLUDE_DIRS} ${GLM_INCLUDE_DIRS} ${OPENGL_INCLUDE_DIRS} ${CMAKE_SOURCE_DIR}/include)

add_executable(labyrinth main.cpp)
```

```
target_link_libraries(labyrinth ${OPENGL_LIBRARIES} ${GLUT_LIBRARY} ${GLM_LIBRARIES}
lib)


add_custom_target(CopyShaders ALL

   COMMAND ${CMAKE_COMMAND} -E copy_directory

   "${CMAKE_SOURCE_DIR}/resource" "${CMAKE_BINARY_DIR}/resource"

   COMMENT "Copy resource directory to build tree" VERBATIM)




set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wno-deprecated-declarations -std=c++17
-stdlib=libc++ -framework OpenGL -framework GLUT -I${GLUT_INCLUDE_DIRS}
-I${GLM_INCLUDE_DIRS} -I${OPENGL_INCLUDE_DIRS} -I${CMAKE_SOURCE_DIR}/include")
```