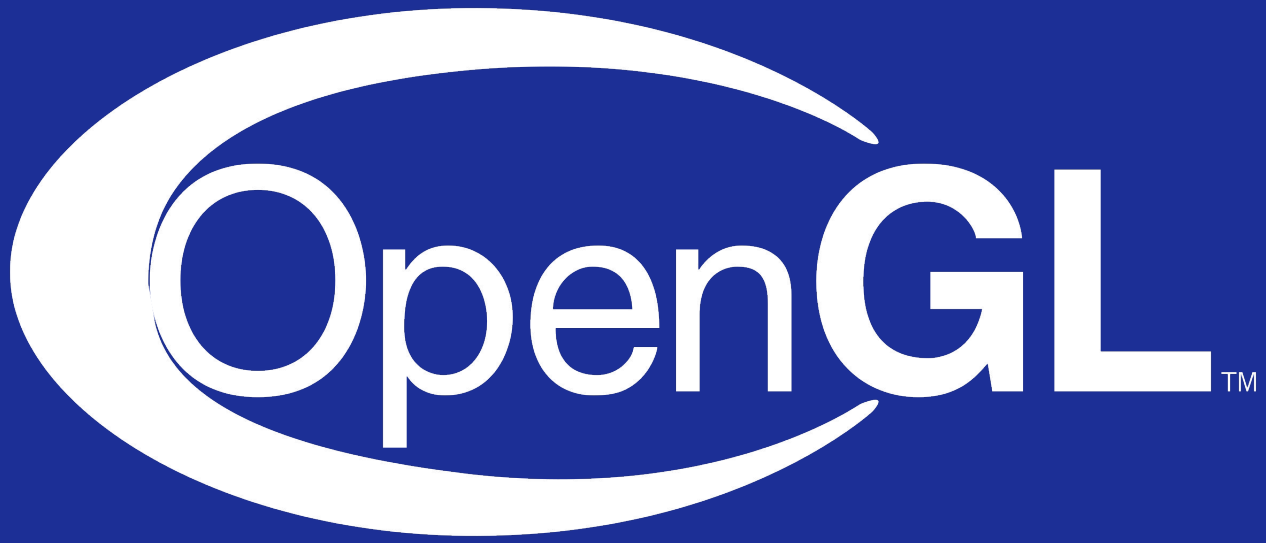
Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Gry Komputerowe
Laboratorium 4

Teksturowanie
Kolizje obiektów z otoczeniem



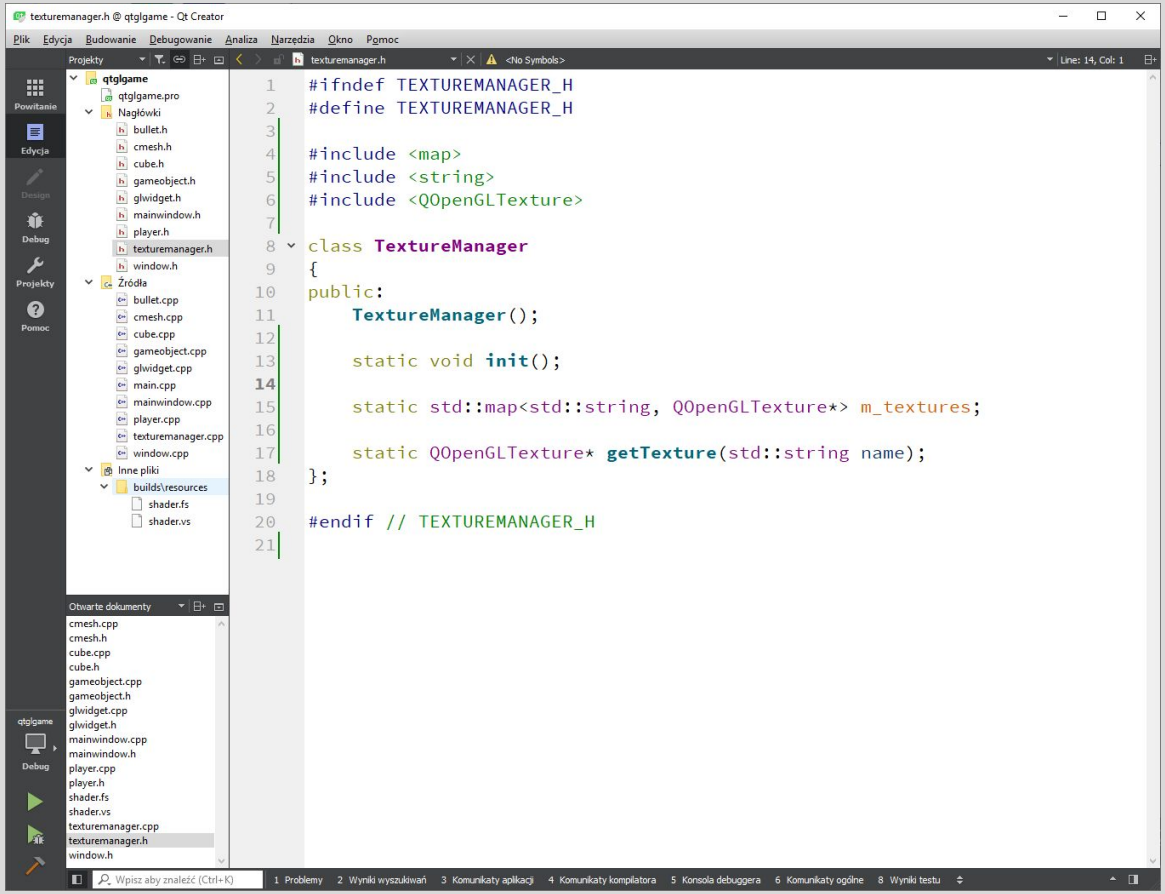
Wydział
Informatyki

mgr inż. Michał Chwesiuk

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Klasa Texture Manager

- Stwórzmy najpierw klasę **TextureManager**, która będzie obsługiwała tekstury w projekcie.

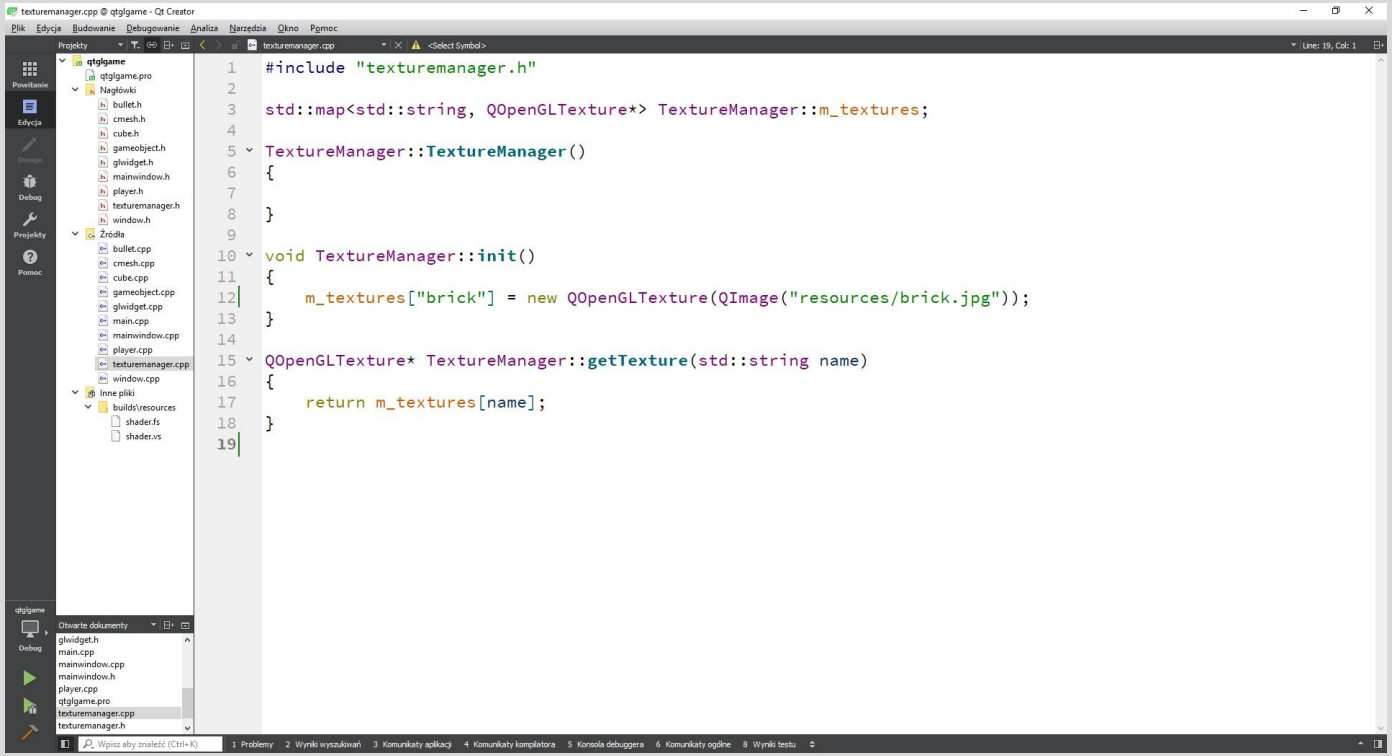


Wydział
Informatyki

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Klasa Texture Manager

- Stwórzmy najpierw klasę **TextureManager**, która będzie obsługiwała tekstury w projekcie.
- Należy dodać plik **brick.jpg** do folderu resource.
  - Plik może zostać znaleziony w internecie, nazwa może być dowolna, musi być ona jednak zgodna z tym w kodzie.



Wydział
Informatyki

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Klasa Texture Manager

- Użyjmy metody **TextureManager::init()** w celu wczytania tekstur do naszej gry wewnątrz funkcji **GLWidget::InitializeGL()** w **glwidget.cpp**.
- Do **glwidget.cpp** trzeba dodać **#include "TextureManager.h"**.

```
void GLWidget::initializeGL()
{
    initializeOpenGLFunctions();
    glClearColor(0.1f, 0.2f, 0.3f, 1);
    glFrontFace(GL_CCW);
    glCullFace(GL_BACK);

    CMesh::loadAllMeshes();

    m_program = new QOpenGLShaderProgram;
    m_program->addShaderFromSourceFile(QOpenGLShader::Vertex, "resources/shader.vs");
    m_program->addShaderFromSourceFile(QOpenGLShader::Fragment, "resources/shader.fs");
    m_program->bindAttributeLocation("vertex", 0);
    m_program->bindAttributeLocation("normal", 1);
    m_program->link();

    m_program->bind();
    m_projMatrixLoc = m_program->uniformLocation("projMatrix");
    m_viewMatrixLoc = m_program->uniformLocation("viewMatrix");
    m_modelMatrixLoc = m_program->uniformLocation("modelMatrix");
    m_modelColorLoc = m_program->uniformLocation("modelColor");
    m_hasTextureLoc = m_program->uniformLocation("hasTexture");
    m_lightLoc.position = m_program->uniformLocation("light.position");
    m_lightLoc.ambient = m_program->uniformLocation("light.ambient");
    m_lightLoc.diffuse = m_program->uniformLocation("light.diffuse");

    m_program->release();

    lastUpdateTime = 0;
    timer.start();

    TextureManager::init();

    addObject(&m_player);
```

Wydział
Informatyki

Teksturowanie - Shadery

- Pierwszym etapem implementacji teksturowania jest rozbudowa shaderów o wczytywanie **współrzędnych tekstury**, użycie **samplera 2D** i jego użycie.

- **shader.vs**

```
attribute vec4 vertex;
attribute vec3 normal;
attribute vec2 uvCoord;
uniform mat4 projMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelMatrix;
uniform highp int hasTexture;
uniform sampler2D texture;
varying highp vec3 fragNormal;
varying highp vec3 vertexWorldSpace;
varying highp vec2 fragUV;

struct Light {
   highp vec3 position;
   highp vec3 ambient;
   highp vec3 diffuse;
};
uniform Light light;

void main() {
   fragNormal = (modelMatrix * vec4(normal, 0)).xyz;
   fragUV = uvCoord;
   vertexWorldSpace = (modelMatrix * vertex).xyz;
   gl_Position = projMatrix * viewMatrix * modelMatrix * vertex;
}
```

Teksturowanie - Shadery

- **shader.fs**

```
uniform highp vec3 modelColor;
uniform highp int hasTexture;
uniform sampler2D texture;
varying highp vec3 fragNormal;
varying highp vec3 vertexWorldSpace;
varying highp vec2 fragUV;

struct Light {
   highp vec3 position;
   highp vec3 ambient;
   highp vec3 diffuse;
};
uniform Light light;

void main() {
   highp vec3 N = normalize(fragNormal);
   highp vec3 L = normalize(light.position - vertexWorldSpace);
   highp float cosNL = dot(N, L);
   cosNL = clamp(cosNL, 0.0, 1.0);
   highp vec3 colorAmb = modelColor * light.ambient;
   highp vec3 colorDif = modelColor * light.diffuse * cosNL;
   highp vec3 colorFull = clamp(colorAmb + colorDif, 0.0, 1.0);

   highp vec3 tex = texture2D(texture, fragUV).xyz;

   if(hasTexture == 1)
      gl_FragColor = vec4(colorFull * tex, 1.0);
   else
   {
      gl_FragColor = vec4(colorFull, 1.0);
   }
}
```

Teksturowanie - shadery

- W klasie **GLWidget** w **glwidget.h** trzeba dodać zmienną **m_hasTextureLoc** typu **int**, która będzie przechowywać lokację zmiennej **hasTexture** w shaderze.
- Trzeba także przypisać wartość tej zmiennej w **GLWidget::initializeGL()** w **glwidget.cpp**.

```
class GLWidget : public QOpenGLWidget, protected QOpenGLF
{
    Q_OBJECT

public:
    GLWidget(QWidget *parent = nullptr);
    ~GLWidget();

    QSize sizeHint() const override;

    friend CMesh;

public slots:
    void cleanup();

protected:
    void initializeGL() override;
    void paintGL() override;
    void resizeGL(int width, int height) override;
    void mousePressEvent(QMouseEvent *event) override;
    void mouseMoveEvent(QMouseEvent *event) override;
    void keyPressEvent(QKeyEvent *event) override;
    void keyReleaseEvent(QKeyEvent *event) override;

    void setTransforms(void);
    void updateGL();

private:

    struct LightLocStruct
    {
        int position;
        int ambient;
        int diffuse;
    };

    QPoint m_lastPos;
    QOpenGLShaderProgram *m_program;
    int m_projMatrixLoc;
    int m_viewMatrixLoc;
    int m_modelMatrixLoc;
    int m_modelColorLoc;
    int m_hasTextureLoc;
    LightLocStruct m_lightLoc;

    QMatrix4x4 m_proj;
    QMatrix4x4 m_camera;
```

```
void GLWidget::initializeGL()
{
    initializeOpenGLFunctions();
    glClearColor(0.1f, 0.2f, 0.3f, 1);
    glFrontFace(GL_CCW);
    glCullFace(GL_BACK);

    CMesh::loadAllMeshes();

    m_program = new QOpenGLShaderProgram;
    m_program->addShaderFromSourceFile(QOpenGLShader::Vertex, "resources/shader.vs");
    m_program->addShaderFromSourceFile(QOpenGLShader::Fragment, "resources/shader.fs");
    m_program->bindAttributeLocation("vertex", 0);
    m_program->bindAttributeLocation("normal", 1);
    m_program->link();

    m_program->bind();
    m_projMatrixLoc = m_program->uniformLocation("projMatrix");
    m_viewMatrixLoc = m_program->uniformLocation("viewMatrix");
    m_modelMatrixLoc = m_program->uniformLocation("modelMatrix");
    m_modelColorLoc = m_program->uniformLocation("modelColor");
    m_hasTextureLoc = m_program->uniformLocation("hasTexture");
    m_lightLoc.position = m_program->uniformLocation("light.position");
    m_lightLoc.ambient = m_program->uniformLocation("light.ambient");
    m_lightLoc.diffuse = m_program->uniformLocation("light.diffuse");

    m_program->release();

    lastUpdateTime = 0;
    timer.start();

    TextureManager::init();

    addObject(&m_player);
```

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Teksturowanie - Klasa GameObject

- Dodajemy do klasy GameObject w **gameobject.h** pole typu **QOpenGL_Texture*** i ustawiamy je na **nullptr**.
- Należy dodać **#include <QOpenGLTexture>** do **gameobject.h**.

gameobject.h @ qtglgame - Qt Creator

```
#ifndef GAMEOBJECT_H
#define GAMEOBJECT_H

#include <QVector3D>
#include <texturemanager.h>

class GLWidget;

class GameObject          △ 'GameObject' has virtual functions but non-virtual destructor
{
public:
    GameObject();

    QVector3D position = QVector3D(0.0f, 0.0f, 0.0f);
    QVector3D rotation = QVector3D(0.0f, 0.0f, 0.0f);
    QVector3D scale = QVector3D(1.0f, 1.0f, 1.0f);

    float m_radius = 1.0f;

    QVector3D material_color = QVector3D(1.0f, 1.0f, 1.0f);

    std::string m_name;

    virtual void init() = 0;
    virtual void render(GLWidget* glwidget) = 0;
    virtual void update() = 0;

    QVector3D energy = QVector3D(0.0f, 0.0f, 0.0f);

    bool isAlive = true;

    QOpenGLTexture* m_texture = nullptr;
};

#endif // GAMEOBJECT_H
```

Wydział
Informatyki

Teksturowanie - Klasa CMesh

- Należy rozwinąć klasę **CMesh** aby przechowywała obok pozycji wierzchołków i ich normalnych **współrzędne tekstur.**
- Do funkcji **CMesh::add()** trzeba dodać dodatkowy parametr **const QVector2D &uv** i dodać jego współrzędne do wektora **m_data**.

```
class CMesh
{
public:
    CMesh();
    ~CMesh();
    const GLfloat *constData() const { return m_data.constData(); }
    int vertexCount() const { return m_count; }
    GLenum primitive() {return m_primitive; }

    void generateCube(GLfloat ww, GLfloat hh, GLfloat dd);
    void generateSphere(float r, int N);
    void generateMeshFromObjFile(QString filename);

    void initVboAndVao();

    void render(GLWidget* glWidget);

    static std::map<std::string, CMesh*> m_meshes;
    static void loadAllMeshes();

private:
    void add(const QVector3D &v, const QVector3D &n, const QVector2D &uv);
```

```
void CMesh::add(const QVector3D &v, const QVector3D &n, const QVector2D &uv)
{
    m_data.append(v.x());
    m_data.append(v.y());
    m_data.append(v.z());
    m_data.append(n.normalized().x());
    m_data.append(n.normalized().y());
    m_data.append(n.normalized().z());
    m_data.append(uv.x());
    m_data.append(uv.y());
    m_count++;
}
```

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Teksturowanie - Aktualizacja VAO

- Należy zaktualizować funkcję **CMesh::initVboAndVao()**, aby rozszerzyła przekazywane atrybuty w **VAO** o współrzędne tekstur, a także włączyła ich przekazywanie.

- **Należy zwrócić uwagę o zmianę niektórych parametrów!**

qtglgame po lab 4 - tekstury/cmesh.cpp @ qtglgame - Qt Creator

```
32      m_data.append(uv.x());
33      m_data.append(uv.y());
34      m_count++;
35  }
36
37  void CMesh::initVboAndVao()
38  {
39      QOpenGLFunctions *f = QOpenGLContext::currentContext()->functions();
40      int dataSize = m_data.size() * int(sizeof(GLfloat));
41
42      m_vao.create(); // creates vertex array object
43      m_vao_binder = new QOpenGLVertexArrayObject::Binder(&m_vao); // binds vertex array object
44      m_vbo.create(); // creates vertex buffer object
45      m_vbo.bind(); // binds vertex buffer object
46      m_vbo.allocate(constData(), dataSize); // copies mesh data to vertex buffer object
47
48      f->glEnableVertexAttribArray(0);
49      f->glEnableVertexAttribArray(1);
50      f->glEnableVertexAttribArray(2);
51      f->glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), nullptr);
52      f->glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), reinterpret_cast<void *>(3 * sizeof(GLfloat)));
53      f->glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), reinterpret_cast<void *>(6 * sizeof(GLfloat)));
54  }
55
56  void CMesh::render(GLWidget* glWidget)
57  {
58      m_vao_binder->rebind();
59      glWidget->glDrawArrays(m_primitive, 0, vertexCount());
60  }
61
62  void CMesh::quad3(GLfloat x1, GLfloat y1, GLfloat z1,
63                    GLfloat x2, GLfloat y2, GLfloat z2,
64                    GLfloat x3, GLfloat y3, GLfloat z3,
65                    GLfloat x4, GLfloat y4, GLfloat z4)
66  {
67      QVector3D n = QVector3D::normal(QVector3D(x4 - x1, y4 - y1, z4 - z1), QVector3D(x2 - x1, y2 - y1, z2 - z1));
68
69      add(QVector3D(x1, y1, z1), n, QVector2D(0, 1));
70      add(QVector3D(x4, y4, z4), n, QVector2D(1, 1));
```

Wydział
Informatyki

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Teksturowanie
Mapowanie sześcienne i sferyczne

- Należy zaktualizować funkcję **CMesh::quad3()** o dodanie współrzędnych tekstury.

```
void CMesh::quad3(GLfloat x1, GLfloat y1, GLfloat z1,
                  GLfloat x2, GLfloat y2, GLfloat z2,
                  GLfloat x3, GLfloat y3, GLfloat z3,
                  GLfloat x4, GLfloat y4, GLfloat z4)
{
    QVector3D n = QVector3D::normal(QVector3D(x4 - x1, y4 - y1, z4 - z1), QVector3D(x2 - x1, y2 - y1, z2 - z1));

    add(QVector3D(x1, y1, z1), n, QVector2D(0, 1));
    add(QVector3D(x4, y4, z4), n, QVector2D(1, 1));
    add(QVector3D(x2, y2, z2), n, QVector2D(0, 0));

    add(QVector3D(x3, y3, z3), n, QVector2D(1, 0));
    add(QVector3D(x2, y2, z2), n, QVector2D(0, 0));
    add(QVector3D(x4, y4, z4), n, QVector2D(1, 1));
}
```

- Trzeba także zaktualizować funkcję **CMesh::generateSphere()** także o współrzędne tekstury (**mapowanie sferyczne**).

```
void CMesh::generateSphere(float r, int N)
{
    for(int j = 0 ; j <= N ; j++)
    {
        for(int i = 0 ; i < N ; i++)
        {
            float theta1 = 3.14f * float(j) / float(N);
            float theta2 = 3.14f * float(j-1) / float(N);
            float phi = 3.14f * 2 * float(i) / float(N);

            float nx1 = sin(theta1) * cos(phi);
            float nx2 = sin(theta2) * cos(phi);
            float nz1 = sin(theta1) * sin(phi);
            float nz2 = sin(theta2) * sin(phi);
            float ny1 = cos(theta1);
            float ny2 = cos(theta2);

            float x1 = r * nx1;
            float x2 = r * nx2;
            float y1 = r * ny1;
            float y2 = r * ny2;
            float z1 = r * nz1;
            float z2 = r * nz2;

            float u1 = 0.5 + atan2(nz1, nx1) / (2 * M_PI);    ⚠ implicit conversion loses
            float v1 = 0.5 + asin(ny1) / M_PI;    ⚠ implicit conversion loses floating-po

            float u2 = 0.5 + atan2(nz2, nx2) / (2 * M_PI);    ⚠ implicit conversion loses
            float v2 = 0.5 + asin(ny2) / M_PI;    ⚠ implicit conversion loses floating-po

            add(QVector3D(x1, y1, z1), QVector3D(nx1, ny1, nz1), QVector2D(u1, v1));
            add(QVector3D(x2, y2, z2), QVector3D(nx2, ny2, nz2), QVector2D(u2, v2));
        }
    }

    m_primitive = GL_TRIANGLE_STRIP;

    initVboAndVao();
}
```

Wydział
Informatyki

Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Teksturowanie
Współrzędne tekstur modelu 3D

- Przy wczytywaniu modeli 3D w CMesh::generateMeshFromObjFile () do funkcji **CMesh::Add()** trzeba przekazać także wczytane współrzędne tekstury.

```
            } else if (token == QStringLiteral("f")) {
                while (!lineStream.atEnd()) {
                    QString faceString;
                    lineStream >> faceString;

                    QVector3D v;
                    QVector3D n;
                    QVector2D uv;

                    QStringList indices = faceString.split(QChar::fromLatin1('/'));

                    v = vertices.at(indices.at(0).toInt() - 1);

                    if(hasTexCoords)
                        uv = texCoords.at(indices.at(1).toInt() - 1);

                    if(hasNormals)
                        n = normals.at(indices.at(2).toInt() - 1);

                    add(v, n, uv);
                }
            }
        }
    }

    m_primitive = GL_TRIANGLES;

    initVboAndVao();
}
```

```
void CMesh::generateMeshFromObjFile(QString filename)
{
    QFile file(filename);
    file.open(QFile::OpenMode::enum_type::ReadOnly);

    std::cout << "Loading " << filename.toStdString() << " - " << (file.isOpen() ? "Found!" : "Not Found!") << std::endl;

    QVector<QVector3D> vertices;
    QVector<QVector3D> normals;
    QVector<QVector2D> texCoords;

    bool hasNormals = false;
    bool hasTexCoords = false;

    QTextStream stream(&file);

    while (!stream.atEnd()) {
        QString line = stream.readLine();
        line = line.simplified();

        if (line.length() > 0 && line.at(0) != QChar::fromLatin1('#')) {
            QTextStream lineStream(&line, QIODevice::ReadOnly);
            QString token;
            lineStream >> token;

            if (token == QStringLiteral("v")) {
                float x, y, z;
                lineStream >> x >> y >> z;
                vertices.append(QVector3D( x, y, z ));
            } else if (token == QStringLiteral("vt")) {
                float s,t;
                lineStream >> s >> t;
                texCoords.append(QVector2D(s, t));
                hasTexCoords = true;
            } else if (token == QStringLiteral("vn")) {
                float x, y, z;
                lineStream >> x >> y >> z;
                normals.append(QVector3D( x, y, z ));
                hasNormals = true;
            } else if (token == QStringLiteral("f")) {
                while (!lineStream.atEnd()) {
                    QString faceString;
                    lineStream >> faceString;

                    QVector3D v;
                    QVector3D n;
                    QVector2D uv;

                    QStringList indices = faceString.split(QChar::fromLatin1('/'));

                    v = vertices.at(indices.at(0).toInt() - 1);

                    if(hasTexCoords)
                        uv = texCoords.at(indices.at(1).toInt() - 1);

                    if(hasNormals)
                        n = normals.at(indices.at(2).toInt() - 1);

                    add(v, n, uv);
                }
            }
        }
    }

    m_primitive = GL_TRIANGLES;

    initVboAndVao();
}
```

Wydział Informatyki

Teksturowanie
Rendering

- Przy renderingu obiektów należy sprawdzić, czy obiekt ma teksturę. Jeśli ją posiada, należy włączyć teksturowanie i ustawić odpowiednią teksturę.
- W funkcji **GLWidget::paintGL()** do pętli renderującej obiekty należy dodać sprawdzenie czy obiekt ma teksturę i opcjonalne włączenie jej.

```
m_camera.lookAt(
m_player.position - m_camDistance * m_player.direction,
m_player.position,
QVector3D(0, 1, 0) );

for(int i = 0 ; i < m_gameObjects.size() ; i++)        ⚠ comparison of int
{
    GameObject* obj = m_gameObjects[i];        ⚠ implicit conversion change

    m_program->setUniformValue(m_modelColorLoc, obj->material_color);

    if(obj->m_texture != nullptr)
    {
        m_program->setUniformValue(m_hasTextureLoc, 1);
        obj->m_texture->bind();
    }
    else
    {
        m_program->setUniformValue(m_hasTextureLoc, 0);
    }

    worldMatrixStack.push(m_world);
        m_world.translate(obj->position);
        m_world.rotate(obj->rotation.x(), 1, 0, 0);
        m_world.rotate(obj->rotation.y(), 0, 1, 0);
        m_world.rotate(obj->rotation.z(), 0, 0, 1);
        m_world.scale(obj->scale);
        setTransforms();
        obj->render(this);
    m_world = worldMatrixStack.pop();
}

m_program->release();

float timerTime = timer.elapsed() * 0.001f;
float deltaTime = timerTime - lastUpdateTime;
```

Teksturowanie
Dodanie tekstury do obiektu

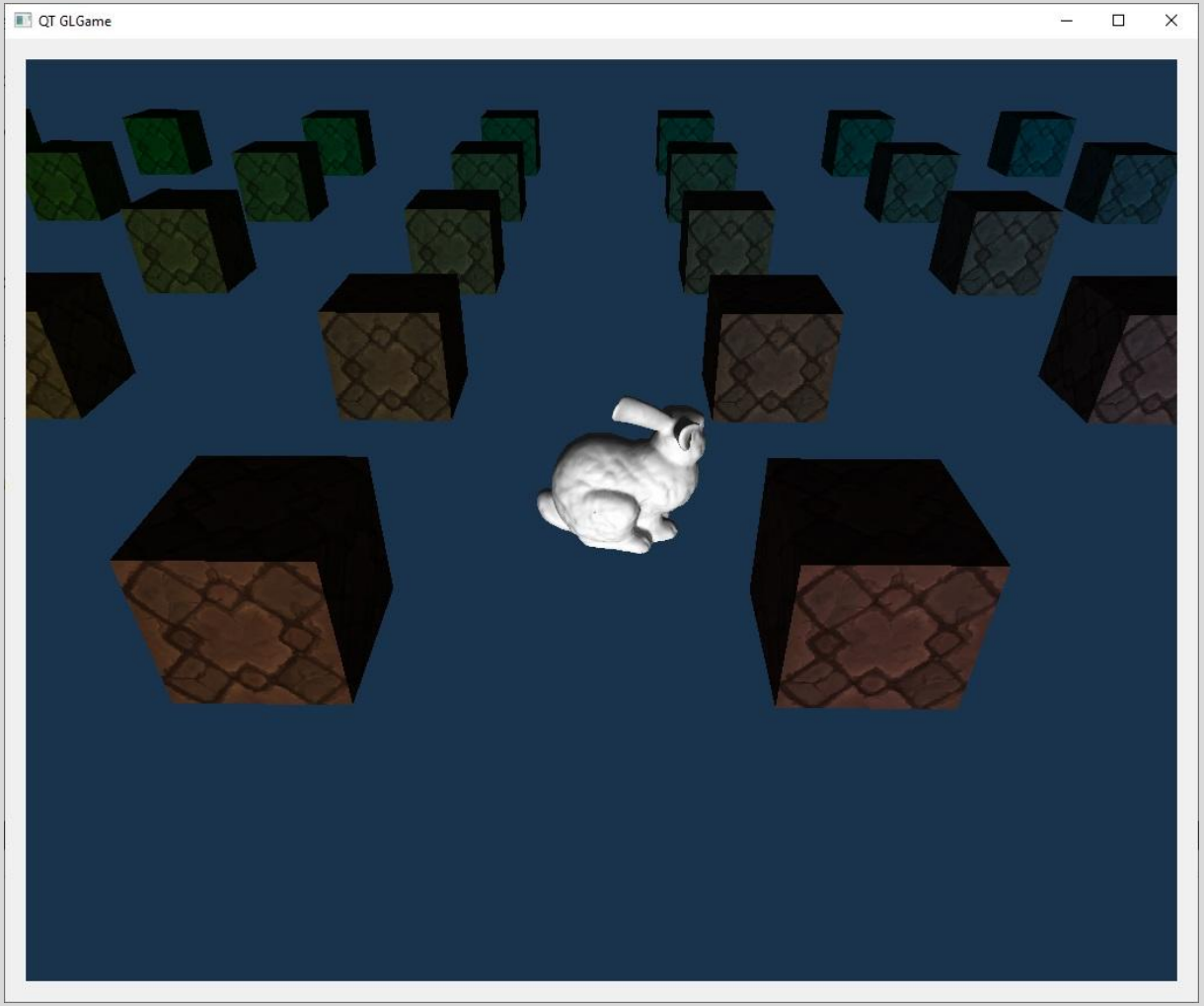
- Mając zaimplementowany w przedstawiony sposób menedżer teksturowania, aby dodać teksturę do obiektu należy do pola **m_teksture** przypisać wartość zwróconą z funkcji **TextureManager::getTexture()**.

- Przykład Cube :

```
for(int i = 0 ; i < 5 ; i++)
    {
        for(int j = 0 ; j < 7 ; j++)
        {
            Cube* cube = new Cube();

            cube->position.setX(j * 1 - 3);
            cube->position.setY(0);
            cube->position.setZ(i * 1 - 6);

            cube->material_color.setX(i * 0.2f);
            cube->material_color.setY(0.5f);
            cube->material_color.setZ(j * 0.1f);
            cube->scale = QVector3D(0.3f, 0.3f, 0.3f);

            cube->m_radius = 0.5 * sqrt(3 * cube->scale.x() * cube->scale.x());

            cube->m_texture = TextureManager::getTexture("brick");

            addObject(cube);
        }
    }
```

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Teksturowanie
Efekt końcowy



Wydział
Informatyki

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Kolizje - struktura Triangle

- Aby zrobić kolizję obiektów ze środowiskiem, potrzeba będzie struktura przechowująca atomiczny element sceny, **struct Triangle**. Całość będziemy przechowywać w **klasie GLWidget** (**glwidget.h**).

- Oprócz tego w pliku **glwidget.h** dodajmy **wektor** trójkątów, **siatkę** (VAO), funkcję **dodającą** nowy trójkąt, a także funkcję **inicjalizującą** trójkąty.

```
    int m_modelMatrixLoc;
    int m_modelColorLoc;
    int m_hasTextureLoc;
    LightLocStruct m_lightLoc;

    QMatrix4x4 m_proj;
    QMatrix4x4 m_camera;
    QMatrix4x4 m_world;

    bool m_keyState[256]={0};

    float m_camDistance = 1.5f;

    QElapsedTimer timer;
    float lastUpdateTime;

    Player m_player;

    std::vector<GameObject*> m_gameObjects;

    void addObject(GameObject* obj);

    float FPS = 60;

    struct Triangle
    {
        QVector3D v1, v2, v3;
        QOpenGLTexture* texture;
        QVector3D n;
        float A, B, C, D;
        int groupSize;
    };

    std::vector<Triangle> collisionTriangles;
    CMesh collisionTrianglesMesh;
    void initCollisionTriangles();
    void addTriangleCollider(QVector3D v1, QVector3D v2, QVector3D v3, int groupSize = 1,
                             QVector2D uv1 = QVector2D(0, 0), QVector2D uv2 = QVector2D(0, 0),
                             QVector2D uv3 = QVector2D(0, 0), QOpenGLTexture* texture = nullptr);
};

#endif
```

Wydział
Informatyki

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Kolizje - Inicjalizacja oraz dodawanie

- Należy zaimplementować funkcje **GLWidget::initCollisionTriangles()** oraz **GLWidget::addTriangleCollider()** w pliku **glwidget.h**.
- Należy zwrócić uwagę na komentarz gdzie wstawić nowy kolider. Można wstawić dwa przykładowe kolidery (podłoga).
- W klasie **CMesh** będzie potrzebne przeniesienie kilku pól do **public**.

```
void GLWidget::initCollisionTriangles()
{

    // TUTAJ DODAWAĆ NOWE KOLIDERY

    collisionTrianglesMesh.m_primitive = GL_TRIANGLES;
    collisionTrianglesMesh.initVboAndVao();
}

void GLWidget::addTriangleCollider(QVector3D v1, QVector3D v2, QVector3D v3, int groupSize,
                                   QVector2D uv1, QVector2D uv2,
                                   QVector2D uv3, QOpenGLTexture* texture)
{
    Triangle t;

    t.v1 = v1;
    t.v2 = v2;
    t.v3 = v3;
    t.texture = texture;
    t.groupSize = groupSize;

    t.n = QVector3D::crossProduct(v1 - v3, v2 - v1).normalized();

    t.A = t.n.x();
    t.B = t.n.y();
    t.C = t.n.z();
    t.D = -(t.A * v1.x() + t.B * v1.y() + t.C*v1.z());

    collisionTriangles.push_back(t);

    collisionTrianglesMesh.add(t.v1, t.n, uv1);
    collisionTrianglesMesh.add(t.v2, t.n, uv2);
    collisionTrianglesMesh.add(t.v3, t.n, uv3);
}
```

```
addTriangleCollider(
            QVector3D(25, 0, -25),
            QVector3D(-25, 0, -25),
            QVector3D(25, 0, 25),
            1,
            QVector2D(1, 1),
            QVector2D(0, 1),
            QVector2D(1, 0),
            TextureManager::getTexture("grass"));

addTriangleCollider(
            QVector3D(-25, 0, -25),
            QVector3D(-25, 0, 25),
            QVector3D(25, 0, 25),
            1,
            QVector2D(0, 1),
            QVector2D(0, 0),
            QVector2D(1, 0),
            TextureManager::getTexture("grass"));
```

Wydział
Informatyki

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Kolizje - Inicjalizacja oraz dodawanie

- Należy uruchomić funkcję **GLWidget::addTriangleCollider()** w **GLWidget::initializeGL()** w pliku **glwidget.cpp**.

```
void GLWidget::initializeGL()
{
    initializeOpenGLFunctions();
    glClearColor(0.1f, 0.2f, 0.3f, 1);
    glFrontFace(GL_CCW);
    glCullFace(GL_BACK);

    CMesh::loadAllMeshes();

    m_program = new QOpenGLShaderProgram;
    m_program->addShaderFromSourceFile(QOpenGLShader::Vertex, "resources/shader.vs");
    m_program->addShaderFromSourceFile(QOpenGLShader::Fragment, "resources/shader.fs");
    m_program->bindAttributeLocation("vertex", 0);
    m_program->bindAttributeLocation("normal", 1);
    m_program->link();

    m_program->bind();
    m_projMatrixLoc = m_program->uniformLocation("projMatrix");
    m_viewMatrixLoc = m_program->uniformLocation("viewMatrix");
    m_modelMatrixLoc = m_program->uniformLocation("modelMatrix");
    m_modelColorLoc = m_program->uniformLocation("modelColor");
    m_hasTextureLoc = m_program->uniformLocation("hasTexture");
    m_lightLoc.position = m_program->uniformLocation("light.position");
    m_lightLoc.ambient = m_program->uniformLocation("light.ambient");
    m_lightLoc.diffuse = m_program->uniformLocation("light.diffuse");

    m_program->release();

    lastUpdateTime = 0;
    timer.start();

    TextureManager::init();
    initCollisionTriangles();

    addObject(&m_player);

    for(int i = 0 ; i < 5 ; i++)
        {
```

Wydział
Informatyki

Kolizje - Poprzednia pozycja obiektów

- Do policzenia kolizji obiektów z trójkątami będzie potrzebna **poprzednia pozycja obiektów**. W tym celu należy dodać do klasy **GameObject** pola .
- Do pętli wykrywającej kolizje obiektów ze sobą dodajmy linijkę kodu przechowującą poprzednią pozycję obiektów, przed aktualizacją.
- Przy okazji możemy dodać grawitację, oraz lekko poprawić obsługę kolizji obiektów (odpychanie ich bez korzystania z energii).

```
class GameObject
{
public:
    GameObject();

    QVector3D position = QVector3D(0.0f, 0.0f, 0.0f);
    QVector3D rotation = QVector3D(0.0f, 0.0f, 0.0f);
    QVector3D scale = QVector3D(1.0f, 1.0f, 1.0f);

    QVector3D previousPosition = QVector3D(0.0f, 0.0f, 0.0f);

    float m_radius = 1.0f;

    QVector3D material_color = QVector3D(1.0f, 1.0f, 1.0f);

    std::string m_name;

    virtual void init() = 0;
    virtual void render(GLWidget* glwidget) = 0;
    virtual void update() = 0;

    QVector3D energy = QVector3D(0.0f, 0.0f, 0.0f);

    bool isAlive = true;

    QOpenGLTexture* m_texture = nullptr;
};

#endif // GAMEOBJECT_H
```

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Kolizje - Poprzednia pozycja obiektów


```
void GLWidget::updateGL()
{
    for(int i = 0 ; i < m_gameObjects.size() ; i++)
        {
            GameObject* obj = m_gameObjects[i];

            obj->previousPosition = obj->position; // Poprzednia pozycja

            for(int j = 0 ; j < m_gameObjects.size() ; j++)
            {
                if(i == j)
                    continue;

                GameObject* obj2 = m_gameObjects[j];

                QVector3D v = obj->position - obj2->position;
                float d = v.length();

                // Porównujemy z sumą promieni
                if(d < (obj->m_radius + obj2->m_radius))
                {
                    std::string name1 = obj->m_name;
                    std::string name2 = obj2->m_name;

                    GameObject* o1 = obj;
                    GameObject* o2 = obj2;

                    if(strcmp(name1.c_str(), name2.c_str()) > 0)
                    {
                        o1 = obj2;
                        o2 = obj;
                        v = -v;
                    }

                    if(!o1->m_name.compare("Player") && !o2->m_name.compare("bullet"))
                    {

                    }
                    else if(!o1->m_name.compare("bullet") && !o2->m_name.compare("cube"))
                    {
                        o1->isAlive = false;
                        o2->isAlive = false;
                    }
                    else
                    {
                        o1->position = o1->position + v * (d/2); // Poprawa
                        o2->position = o2->position - v * (d/2); // kolizji
                        v.normalize();
                        float energySum = o1->energy.length() + o2->energy.length();
                        o1->energy = v * energySum / 2;
                        o2->energy = -v * energySum / 2;
                    }
                }
            }

            obj->energy.setY(obj->energy.y() - 0.02f); // Grawitacja

            obj->update();
        }
```

Wydział
Informatyki

Kolizje - Wykrywanie

- Do funkcji **GLWidget::updateGL()** w pliku **glwidget.cpp** należy dodać wyszukiwanie kolizji z trójkątami.

```
for (unsigned int i = 0; i < m_gameObjects.size(); i++)
{
    GameObject* obj = m_gameObjects[i];

    for (int j = 0; j < collisionTriangles.size(); j++)
    {
        Triangle tr = collisionTriangles[j];

        float currDist = tr.A * obj->position.x() + tr.B * obj->position.y() + tr.C * obj->position.z() + tr.D;
        float prevDist = tr.A * obj->previousPosition.x() + tr.B * obj->previousPosition.y() + tr.C * obj->previousPosition.z() + tr.D;

        if ((currDist * prevDist < 0) || abs(currDist) < obj->m_radius)
        {
            // Rzut pozycji obiektu na plaszczyzne
            QVector3D p = obj->position - tr.n * currDist;

            // Przesuniecie punktu do srodka trojkata o dlugosc promienia kolidera
            QVector3D r = (tr.v1 + tr.v2 + tr.v3) * (1.0f / 3.0f) - p;
            r = r.normalized();
            p = p + r * obj->m_radius;

            // Obliczenie v, w, u - wspolrzednych barycentrycznych
            QVector3D v0 = tr.v2 - tr.v1, v1 = tr.v3 - tr.v1, v2 = p - tr.v1;
            float d00 = QVector3D::dotProduct(v0, v0);
            float d01 = QVector3D::dotProduct(v0, v1);
            float d11 = QVector3D::dotProduct(v1, v1);
            float d20 = QVector3D::dotProduct(v2, v0);
            float d21 = QVector3D::dotProduct(v2, v1);
            float denom = d00 * d11 - d01 * d01;

            float v = (d11 * d20 - d01 * d21) / denom;
            float w = (d00 * d21 - d01 * d20) / denom;
            float u = 1.0f - v - w;

            // Sprawdzenie czy punkt jest w srodku trojkata
            if (v >= 0 && w >= 0 && (v + w) <= 1)
            {
                float d = obj->m_radius - currDist;

                obj->position = obj->position + tr.n * d;

                obj->energy = obj->energy - tr.n * QVector3D::dotProduct(tr.n, obj->energy) * 2;
            }
        }
    }
}
```

Kolizje - Renderowanie

- Przed renderowaniem trzeba zaimplementować renderowanie fragmentów geometrii siatki.
- W klasie **CMesh** należy przeciążyć funkcję **render()**, tak by renderowała wskazany fragment (**cmesh.h**, **cmesh.cpp**).

```
class CMesh
{
public:
    CMesh();
    ~CMesh();
    const GLfloat *constData() const { return m_data.constData(); }
    int vertexCount() const { return m_count; }
    GLenum primitive() {return m_primitive; }

    void generateCube(GLfloat ww, GLfloat hh, GLfloat dd);
    void generateSphere(float r, int N);
    void generateMeshFromObjFile(QString filename);

    void initVboAndVao();

    void render(GLWidget* glWidget);
    void render(GLWidget* glWidget, int offset, int count);

    static std::map<std::string, CMesh*> m_meshes;
    static void loadAllMeshes();

    void add(const QVector3D &v, const QVector3D &n, const QVector2D &uv);
    GLenum m_primitive;

private:


    void quad3(GLfloat x1, GLfloat y1, GLfloat z1,
               GLfloat x2, GLfloat y2, GLfloat z2,
               GLfloat x3, GLfloat y3, GLfloat z3,
               GLfloat x4, GLfloat y4, GLfloat z4);

    QVector<GLfloat> m_data;
    int m_count;

    QOpenGLVertexArrayObject m_vao;
    QOpenGLBuffer m_vbo;
    QOpenGLVertexArrayObject::Binder* m_vao_binder;
};
```

```
void CMesh::render(GLWidget* glWidget, int offset, int count)
{
    m_vao_binder->rebind();
    glWidget->glDrawArrays(m_primitive, offset, count);
}
```

Kolizje - Renderowanie

- Do funkcji **GLWidget::paintGL()** w pliku **glwidget.cpp** należy dodać renderowanie z trójkątami.

```
        m_world.scale(obj->scale);
        setTransforms();
        obj->render(this);
    m_world = worldMatrixStack.pop();
}

for(int i = 0 ; i < collisionTriangles.size()  ; /*nic*/) {

    Triangle triangle = collisionTriangles[i];

    m_program->setUniformValue(m_modelColorLoc, QVector3D(1, 1, 1));

    if(triangle.texture != nullptr)
    {
        m_program->setUniformValue(m_hasTextureLoc, 1);
        triangle.texture->bind();
    }
    else
    {
        m_program->setUniformValue(m_hasTextureLoc, 0);
    }

    worldMatrixStack.push(m_world);
        m_world.translate(QVector3D(0, 0, 0));
        m_world.rotate(0, 1, 0, 0);
        m_world.rotate(0, 0, 1, 0);
        m_world.rotate(0, 0, 0, 1);
        m_world.scale(QVector3D(1 ,1 ,1));
        setTransforms();
        collisionTrianglesMesh.render(this, i * 3, triangle.groupSize * 3);
    m_world = worldMatrixStack.pop();

    i += triangle.groupSize;
}

m_program->release();

float timerTime = timer.elapsed() * 0.001f;
float deltaTime = timerTime - lastUpdateTime;
if(deltaTime >= (1.0f / FPS)) {
```

Kolizje - Przykładowa scena

- Poniższy kod można wstawić do **GLWidget::initCollisionTriangles()** w celu uzyskania przykładowej sceny.

```
addTriangleCollider(QVector3D(30, 0, -30), QVector3D(-30, 0, -30), QVector3D(30, 0, 30), 1, QVector2D(1, 1), QVector2D(0, 1), QVector2D(1, 0), TextureManager::getTexture("grass"));
addTriangleCollider(QVector3D(-30, 0, -30), QVector3D(-30, 0, 30), QVector3D(30, 0, 30), 1, QVector2D(0, 1), QVector2D(0, 0), QVector2D(1, 0), TextureManager::getTexture("grass"));

addTriangleCollider(QVector3D(-15, 0, 15), QVector3D(-15, 15, 15), QVector3D(15, 15, 15), 1, QVector2D(1, 0), QVector2D(1, 1), QVector2D(0, 1), TextureManager::getTexture("brick"));
addTriangleCollider(QVector3D(15, 15, 15), QVector3D(15, 0, 15), QVector3D(-15, 0, 15), 1, QVector2D(0, 1), QVector2D(0, 0), QVector2D(1, 0), TextureManager::getTexture("brick"));
addTriangleCollider(QVector3D(-15, 15, 15), QVector3D(-15, 0, 15), QVector3D(-15, 15, 20), 1, QVector2D(0, 1), QVector2D(0, 0), QVector2D(1, 1), TextureManager::getTexture("brick"));
addTriangleCollider(QVector3D(-15, 0, 15), QVector3D(-15, 0, 20), QVector3D(-15, 15, 20), 1, QVector2D(0, 0), QVector2D(1, 0), QVector2D(1, 1), TextureManager::getTexture("brick"));
addTriangleCollider(QVector3D(-35, 0, 20), QVector3D(-35, 15, 20), QVector3D(-15, 15, 20), 1, QVector2D(1, 0), QVector2D(1, 1), QVector2D(0, 1), TextureManager::getTexture("brick"));
addTriangleCollider(QVector3D(-15, 15, 20), QVector3D(-15, 0, 20), QVector3D(-35, 0, 20), 1, QVector2D(0, 1), QVector2D(0, 0), QVector2D(1, 0), TextureManager::getTexture("brick"));
addTriangleCollider(QVector3D(-35, 15, 20), QVector3D(-35, 0, 20), QVector3D(-35, 15, -30), 1, QVector2D(0, 1), QVector2D(0, 0), QVector2D(1, 1), TextureManager::getTexture("brick"));
addTriangleCollider(QVector3D(-35, 0, 20), QVector3D(-35, 0, -30), QVector3D(-35, 15, -30), 1, QVector2D(0, 0), QVector2D(1, 0), QVector2D(1, 1), TextureManager::getTexture("brick"));
addTriangleCollider(QVector3D(25, 0, -30), QVector3D(25, 15, -30), QVector3D(-35, 15, -30), 1, QVector2D(1, 0), QVector2D(1, 1), QVector2D(0, 1), TextureManager::getTexture("brick"));
addTriangleCollider(QVector3D(-35, 15, -30), QVector3D(-35, 0, -30), QVector3D(25, 0, -30), 1, QVector2D(0, 1), QVector2D(0, 0), QVector2D(1, 0), TextureManager::getTexture("brick"));
addTriangleCollider(QVector3D(25, 15, -30), QVector3D(25, 0, -30), QVector3D(25, 15, 25), 1, QVector2D(0, 1), QVector2D(0, 0), QVector2D(1, 1), TextureManager::getTexture("brick"));
addTriangleCollider(QVector3D(25, 0, -30), QVector3D(25, 0, 25), QVector3D(25, 15, 25), 1, QVector2D(0, 0), QVector2D(1, 0), QVector2D(1, 1), TextureManager::getTexture("brick"));
addTriangleCollider(QVector3D(25, 15, 25), QVector3D(25, 0, 25), QVector3D(15, 15, 15), 1, QVector2D(0, 1), QVector2D(0, 0), QVector2D(1, 1), TextureManager::getTexture("brick"));
addTriangleCollider(QVector3D(25, 0, 25), QVector3D(15, 0, 15), QVector3D(15, 15, 15), 1, QVector2D(0, 0), QVector2D(1, 0), QVector2D(1, 1), TextureManager::getTexture("brick"));

addTriangleCollider(QVector3D(5, 3, -21), QVector3D(-5, 3, -21), QVector3D(5, 3, -13), 1, QVector2D(1, 1), QVector2D(0, 1), QVector2D(1, 0), TextureManager::getTexture("wood"));
addTriangleCollider(QVector3D(-5, 3, -21), QVector3D(-5, 3, -13), QVector3D(5, 3, -13), 1, QVector2D(0, 1), QVector2D(0, 0), QVector2D(1, 0), TextureManager::getTexture("wood"));
addTriangleCollider(QVector3D(-5, 3, -21), QVector3D(-10, 0, -21), QVector3D(-5, 3, -13), 1, QVector2D(1, 1), QVector2D(0, 1), QVector2D(1, 0), TextureManager::getTexture("wood"));
addTriangleCollider(QVector3D(-10, 0, -21), QVector3D(-10, 0, -13), QVector3D(-5, 3, -13), 1, QVector2D(0, 1), QVector2D(0, 0), QVector2D(1, 0), TextureManager::getTexture("wood"));
addTriangleCollider(QVector3D(10, 0, -21), QVector3D(5, 3, -21), QVector3D(10, 0, -13), 1, QVector2D(1, 1), QVector2D(0, 1), QVector2D(1, 0), TextureManager::getTexture("wood"));
addTriangleCollider(QVector3D(5, 3, -21), QVector3D(5, 3, -13), QVector3D(10, 0, -13), 1, QVector2D(0, 1), QVector2D(0, 0), QVector2D(1, 0), TextureManager::getTexture("wood"));
addTriangleCollider(QVector3D(5, 3, -13), QVector3D(-5, 3, -13), QVector3D(5, 0, -13), 1, QVector2D(1, 1), QVector2D(0, 1), QVector2D(1, 0), TextureManager::getTexture("wood"));
addTriangleCollider(QVector3D(-5, 3, -13), QVector3D(-5, 0, -13), QVector3D(5, 0, -13), 1, QVector2D(0, 1), QVector2D(0, 0), QVector2D(1, 0), TextureManager::getTexture("wood"));
addTriangleCollider(QVector3D(-5, 3, -13), QVector3D(-10, 0, -13), QVector3D(-5, 0, -13), 1, QVector2D(1, 1), QVector2D(0, 0), QVector2D(1, 0), TextureManager::getTexture("wood"));
addTriangleCollider(QVector3D(5, 3, -13), QVector3D(5, 0, -13), QVector3D(10, 0, -13), 1, QVector2D(0, 1), QVector2D(0, 0), QVector2D(1, 0), TextureManager::getTexture("wood"));
addTriangleCollider(QVector3D(-5, 3, -21), QVector3D(5, 3, -21), QVector3D(-5, 0, -21), 1, QVector2D(1, 1), QVector2D(0, 1), QVector2D(1, 0), TextureManager::getTexture("wood"));
addTriangleCollider(QVector3D(5, 3, -21), QVector3D(5, 0, -21), QVector3D(-5, 0, -21), 1, QVector2D(0, 1), QVector2D(0, 0), QVector2D(1, 0), TextureManager::getTexture("wood"));
addTriangleCollider(QVector3D(-5, 3, -21), QVector3D(-5, 0, -21), QVector3D(-10, 0, -21), 1, QVector2D(1, 1), QVector2D(1, 0), QVector2D(0, 0), TextureManager::getTexture("wood"));
addTriangleCollider(QVector3D(5, 3, -21), QVector3D(10, 0, -21), QVector3D(5, 0, -21), 1, QVector2D(0, 1), QVector2D(1, 0), QVector2D(0, 0), TextureManager::getTexture("wood"));
```

- Kod pod adresem :
  - http://mchwesiuk.pl/wp-content/uploads/2019/04/qtglgame-sample-scene.txt

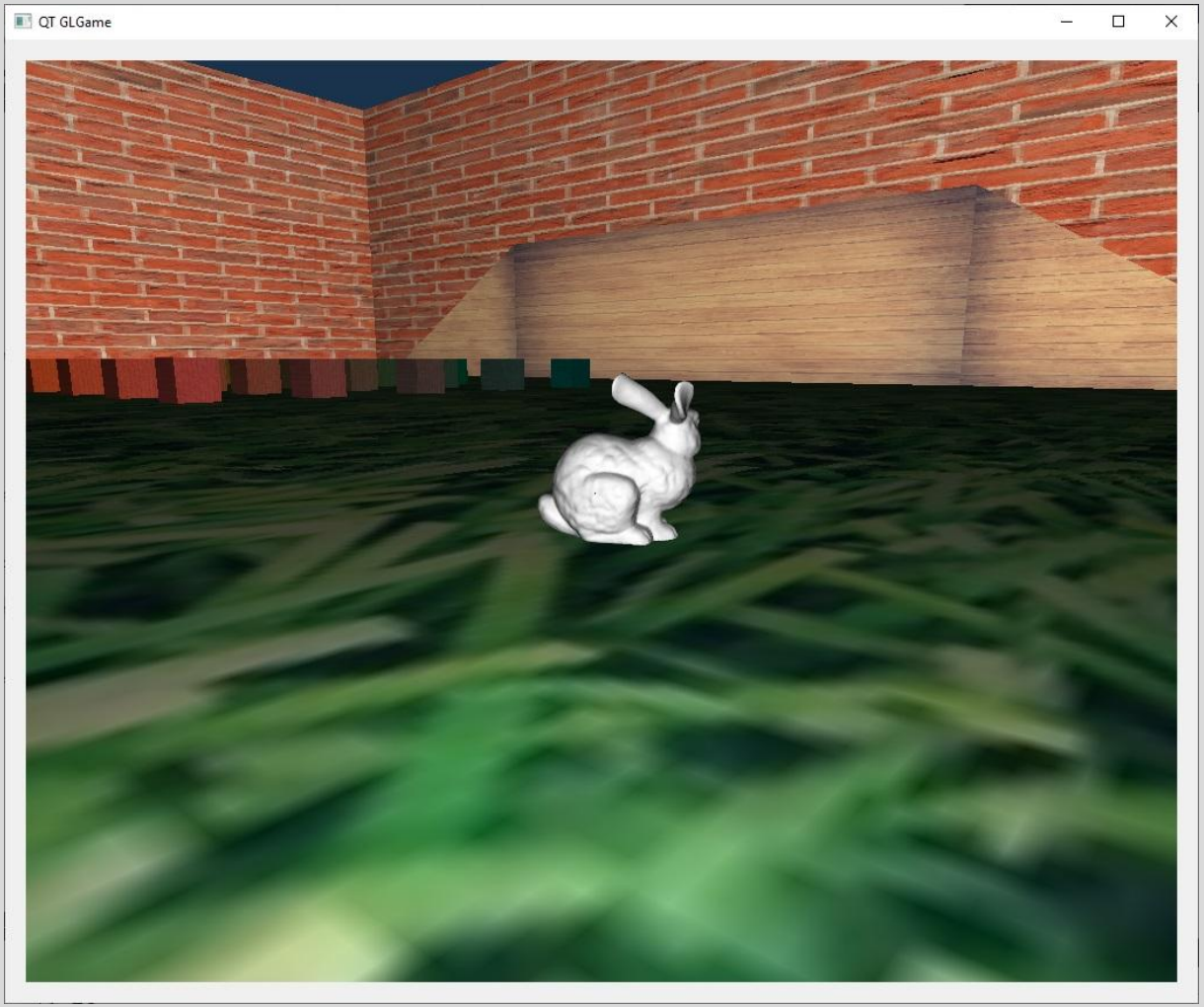
- Należy dodać pliki **grass.jpg** i **wood.jpg** do folderu resources, a także wczytać je w funkcji **TextureManager::init()**

- Można poprawić **oświetlenie ustawiając źródło światła za pozycję gracza** (**GLWidget::paintGL()**, **glwidget.cpp**).

```
void GLWidget::paintGL() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);

    QStack<QMatrix4x4> worldMatrixStack;

    m_program->bind();

    m_program->setUniformValue(m_lightLoc.position, m_player.position - m_player.direction);
```

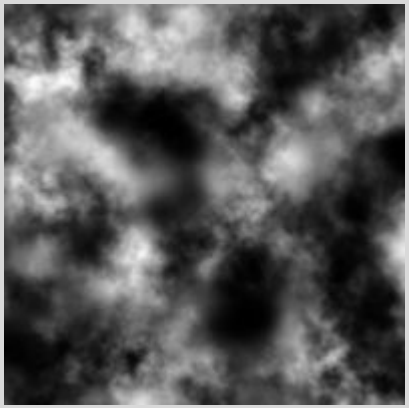
Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Kolizje - Efekt końcowy



Wydział
Informatyki

Mapa wysokości



- Można wykorzystać **mapę wysokości** żeby wygenerować teren.
- Należy znaleźć taką mapę na internecie (lub stworzyć samemu) i zapisać do folderu *resource*.

Mapa wysokości - implementacja

- Należy dodać deklarację metody **loadHeightMap()** do klasy **GLWidget** w pliku **glwidget.h**, oraz jej definicję w **glwidget.cpp**, oraz użyć tej metody wewnątrz **GLWidget::initCollisionTriangles()** (**glwidget.cpp**).

- Deklaracja **GLWidget::loadHeightMap()** (**glwidget.h**) :

```
    void addTriangleCollider(QVector3D v1, QVector3D v2, QVector3D v3, int groupSize = 1,
                             QVector2D uv1 = QVector2D(0, 0), QVector2D uv2 = QVector2D(0, 0),
                             QVector2D uv3 = QVector2D(0, 0), QOpenGLTexture* texture = nullptr);

    void loadHeightMap(QString filepath, QVector3D center, QVector3D size,
                       QVector2D maxUV = QVector2D(0, 0), QOpenGLTexture* texture = nullptr);
};

#endif
```

- Użycie **GLWidget::loadHeightMap()** (**glwidget.cpp**) :

```
void GLWidget::initCollisionTriangles()
{
    loadHeightMap("resources/heightmap.png", QVector3D(0.0f, -5.0f, 0.0f), QVector3D(200.0f, 20.0f, 200.0f),
                  QVector2D(10, 10), TextureManager::getTexture("grass"));

    collisionTrianglesMesh.m_primitive = GL_TRIANGLES;
    collisionTrianglesMesh.initVboAndVao();
}
```

- Implementacja **GLWidget::loadHeightMap()** na kolejnym slajdzie.

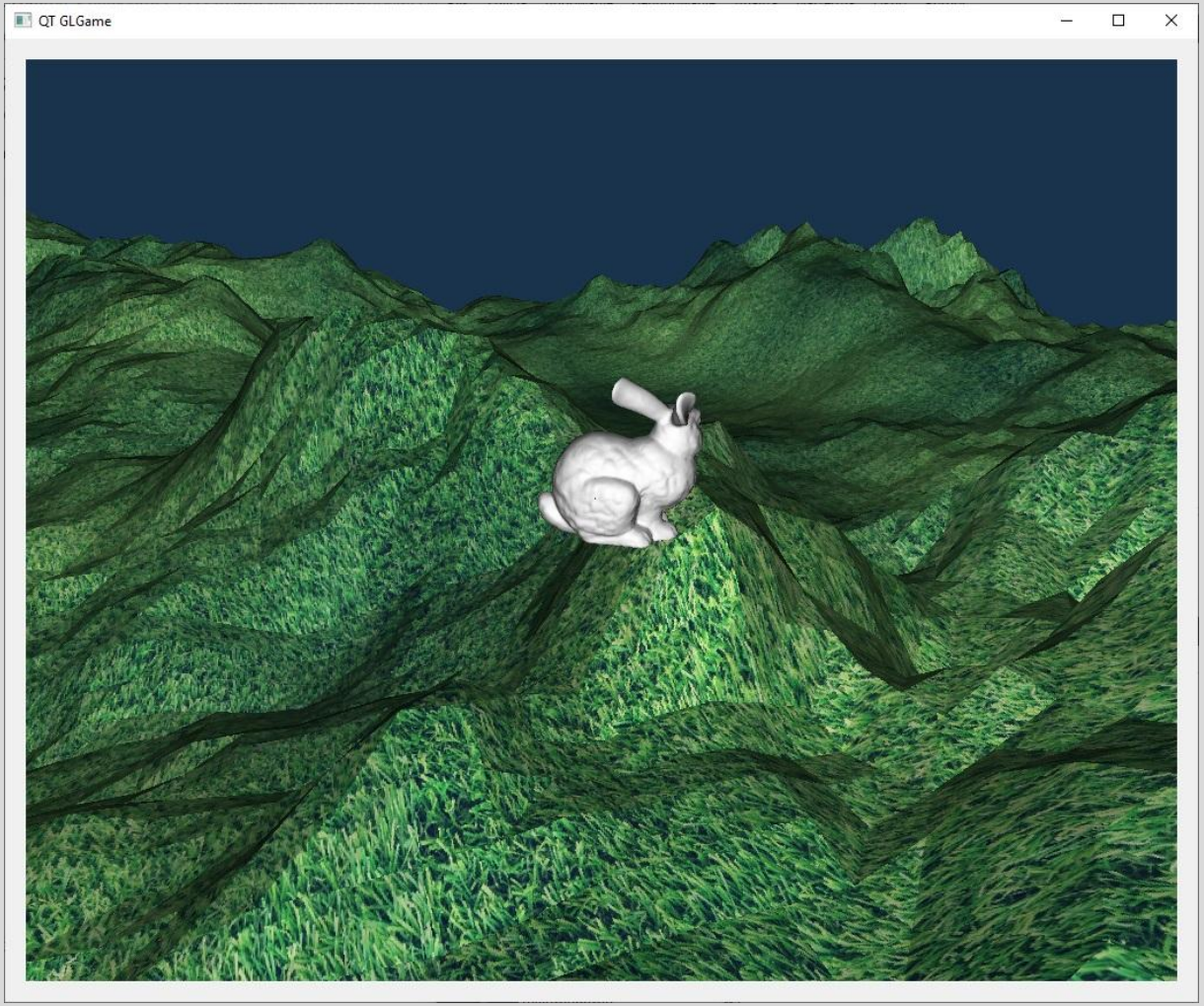
Mapa wysokości - implementacja

- Implementacja **GLWidget::loadHeightMap()** (**glwidget.cpp**) :

```
void GLWidget::loadHeightMap(QString filepath, QVector3D center, QVector3D size, QVector2D maxUV, QOpenGLTexture* texture)
{
    QImage img;
    img.load(filepath);

    float boxSizeX = size.x()/img.width();
    float boxSizeZ = size.z()/img.height();

    float offsetX = -boxSizeX * img.width()/2 + center.x();
    float offsetZ = -boxSizeZ * img.height()/2;

    float minY = center.y() - size.y() / 2;
    float maxY = center.y() + size.y() / 2;

    int groupSize = (img.width() - 1) * (img.height() - 1) * 2;

    for(int i = 0 ; i < img.height() - 1 ; i++)
    {
        for(int j = 0 ; j < img.width() - 1 ; j++)
        {
            float X1 = offsetX + boxSizeX * i;
            float X2 = offsetX + boxSizeX * (i+1);
            float Z1 = offsetZ + boxSizeZ * j;
            float Z2 = offsetZ + boxSizeZ * (j+1);

            float Y00 = QColor(img.pixel(j, i)).red() / 255.0f * (maxY - minY) + minY;
            float Y01 = QColor(img.pixel(j+1, i)).red() / 255.0f * (maxY - minY) + minY;
            float Y10 = QColor(img.pixel(j, i+1)).red() / 255.0f * (maxY - minY) + minY;
            float Y11 = QColor(img.pixel(j+1, i+1)).red() / 255.0f * (maxY - minY) + minY;

            QVector2D texCoord00(maxUV.x() * i / img.width(), maxUV.y() * j / img.height());
            QVector2D texCoord01(maxUV.x() * (i+1) / img.width(), maxUV.y() * j / img.height());
            QVector2D texCoord10(maxUV.x() * i / img.width(), maxUV.y() * (j+1) / img.height());
            QVector2D texCoord11(maxUV.x() * (i+1) / img.width(), maxUV.y() * (j+1) / img.height());

            addTriangleCollider(QVector3D(X1, Y00, Z1), QVector3D(X1, Y01, Z2), QVector3D(X2, Y11, Z2), groupSize,
                                texCoord00, texCoord10, texCoord11, texture);

            addTriangleCollider(QVector3D(X2, Y11, Z2), QVector3D(X2, Y10, Z1), QVector3D(X1, Y00, Z1), groupSize,
                                texCoord11, texCoord01, texCoord00, texture);
        }
    }
}
```

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Mapa wysokości - efekt końcowy



Wydział
Informatyki