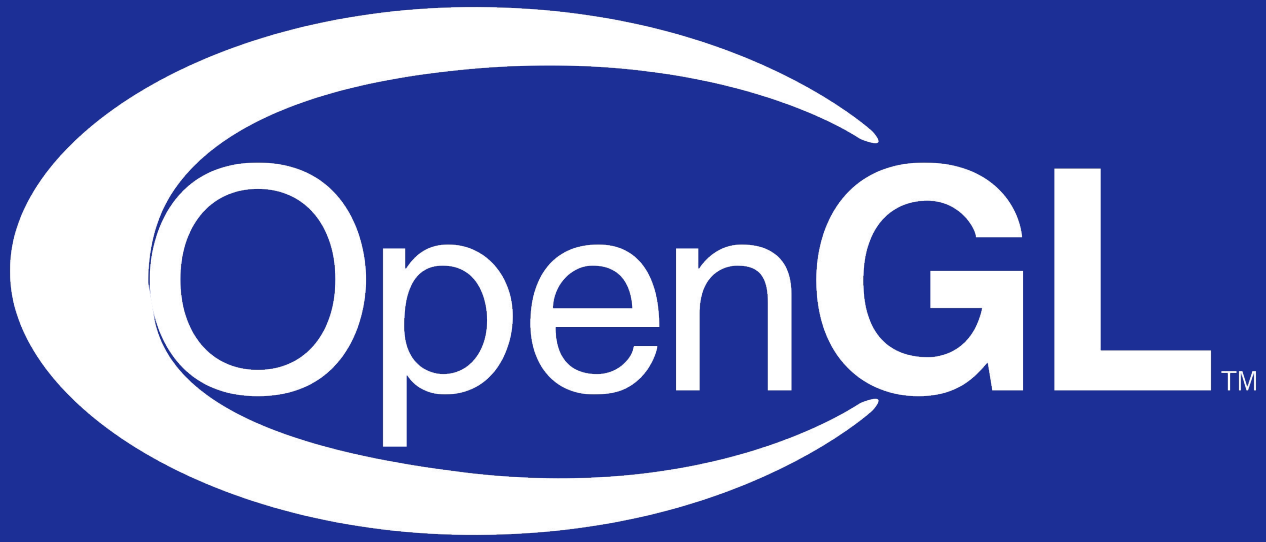
Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Gry Komputerowe
Laboratorium 3

Organizacja obiektów sceny
Kolizje obiektów



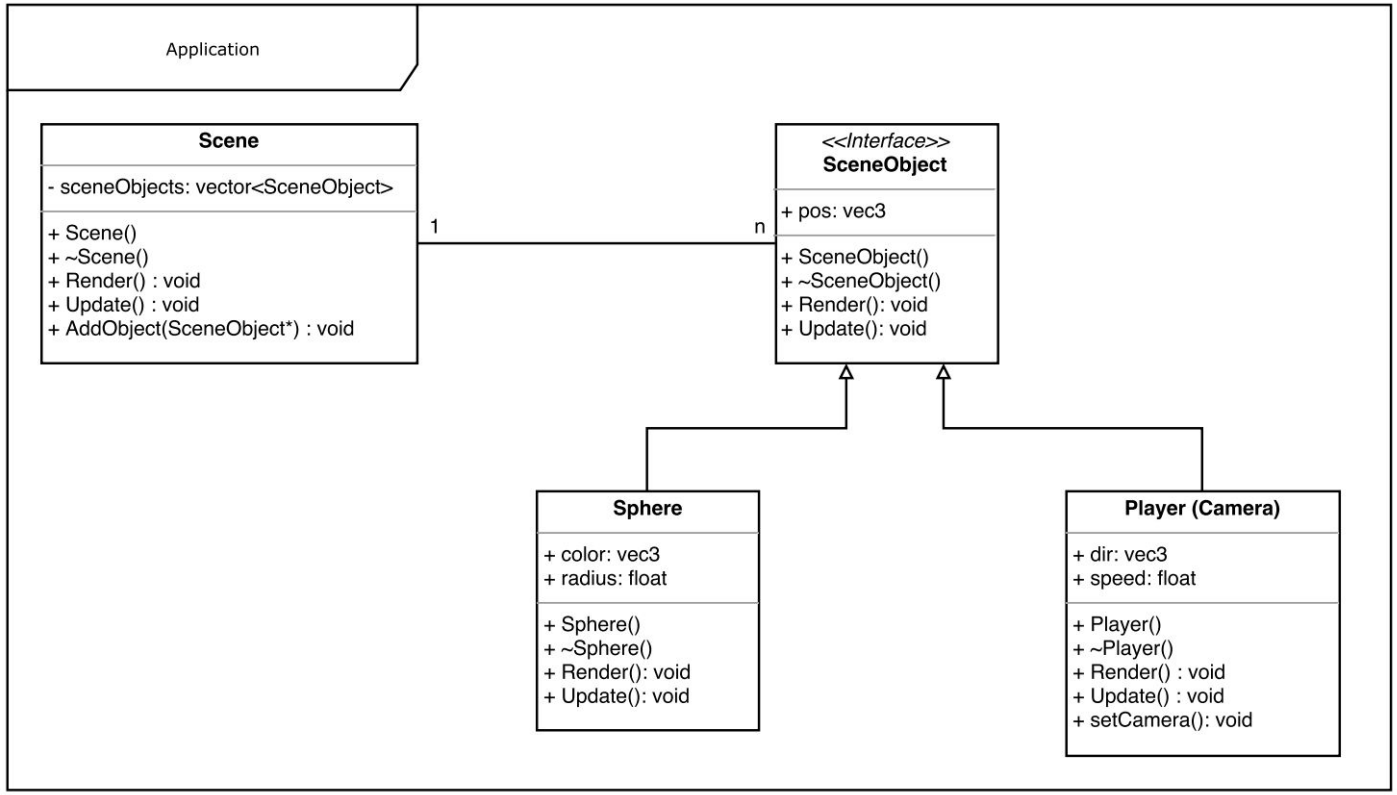
Wydział
Informatyki

mgr inż. Michał Chwesiuk

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Diagram Klas

Klasa GameObject
Klasa GLWidget
Rendering obiektów
Aktualizacja obiektów
Klasa Cube
Kolizje
Pocisk



Wydział
Informatyki

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Klasa GameObject

Klasa GameObject
Klasa GLWidget
Rendering obiektów
Aktualizacja obiektów
Klasa Cube
Kolizje
Pocisk

- Stwórz nową klasę GameObject. Klasa ta będzie klasą abstrakcyjną, z niej będą dziedziczyć wszystkie obiekty sceny zawarte w grze.
- Klasa powinna mieć taką postać (**gameobject.h**) :

```
#ifndef GAMEOBJECT_H
#define GAMEOBJECT_H

#include <QVector3D>

class GLWidget;

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);

   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;

};

#endif // GAMEOBJECT_H
```

Wydział
Informatyki

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Klasa Player

Klasa GameObject
Klasa GLWidget
Rendering obiektów
Aktualizacja obiektów
Klasa Cube
Kolizje
Pocisk

- Rozszerz klasę Player w pliku **player.h** z poprzednich zajęć o dziedziczenie po klasie *GameObject*, a także o polę *m_mesh* i metody dziedziczone z klasy *GameObject*.

```
#include "gameobject.h"
#include "cmesh.h"

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

    QVector3D direction;
    float speed;

    void init();
    void render(GLWidget* glwidget);
    void update();

    CMesh m_mesh;
};
```

- Zaimplementuj do klasy *Player* funkcje *init()*, *render()* i *update()* w pliku **player.cpp**.

```
void Player::init()
{
        m_mesh.generateMeshFromObjFile("resources/bunny.obj");
        scale = QVector3D(0.1f, 0.1f, 0.1f);
        m_radius = 0.1f;

        m_name = "Player";
}
void Player::render(GLWidget* glwidget)
{
        m_mesh.render(glwidget);
}
void Player::update()
{
        rotation.setY(90-atan2(direction.z(), direction.x()) * 180 / 3.14f);
}
```

Wydział
Informatyki

Klasa GLWidget

Klasa GameObject

Klasa GLWidget

Rendering obiektów

Aktualizacja obiektów

Klasa Cube

Kolizje

Pocisk

- Aby przystosować aplikację do nowych klas trzeba zmodernizować najważniejszą funkcję w programie, czyli **GLWidget**.
- Na początek należy **usunąć** kilka sygnałów, metod i pól z klasy *GLWidget* w pliku **glwidget.h**.

```
void setXRotation(float angle);
void setYRotation(float angle);
void setZRotation(float angle);

void xRotationChanged(float angle);
void yRotationChanged(float angle);
void zRotationChanged(float angle);

void qNormalizeAngle(float &angle);


float m_camXRot = 15;
float m_camYRot = 330;
float m_camZRot = 0;

QVector3D m_robotPosition;
float robotArmAngle = 0;
```

- Następnie należy **usunąć wszystkie użycia usuniętych składowych klasy**.
  - Najlepsza metoda to próba kompilacji, a następnie przejście do błędów i usunięcie linijek kodu powodujących błąd :>
  - W niektórych przypadkach należy usunąć kilka linijek, w niektórych całe funkcje.

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Klasa GameObject

Klasa GLWidget

Rendering obiektów

Aktualizacja obiektów

Klasa Cube

Kolizje

Pocisk

Klasa GLWidget
Wektor obiektów gry

- Wygodnym sposobem jest przechowywanie wszystkich obiektów gry (tj. gracz, przeciwnicy, pociski) w jednym wektorze, po którym można iterować.
- W klasie *GLWidget* w pliku **glwidget.h** należy dodać wektor z biblioteki standardowej C++ zawieracjących wskaźniki do obiektów typu *gameObject* (należy do pliku dopisać **#include <vector>**)

```
std::vector<GameObject*> m_gameObjects;
```

- Dodajmy także funkcję *addObject()*, która będzie umieszczać nowe obiekty w tym kontenerze przy jednoczesnym inicjowaniem jego.

```
void addObject(GameObject* obj);
```

- Zaimplementujemy także funkcję *addObject()* wewnątrz pliku **glwidget.cpp**.

```
void GLWidget::addObject(GameObject* obj)
{
        obj->init();
        m_gameObjects.push_back(obj);
}
```

- Mając zaimplementowany kontener na obiekty gry, a także funkcję dodającą do niego nowe obiekty, możemy dodać pierwszy obiekt, czyli gracza. W funkcji *GLWidget::initializeGL()* w pliku **glwidget.cpp** należy dodać odpowiedni kod.

```
addObject(&m_player);
```

Wydział
Informatyki

Klasa GLWidget
Renderowanie obiektów gry

Klasa GameObject
Klasa GLWidget
Rendering obiektów
Aktualizacja obiektów
Klasa Cube
Kolizje
Pocisk

- Trzymając obiekty gry w kontenerze, nie potrzebujemy już rysować obiektów oddzielnie.
- Należy **usunąć** z klasy *GLwidget* w pliku **glwidget.h** przechowywane pojedyncze siatki, a także wszystkie użycia tych siatek.

```
QMap<QString, CMesh*> m_meshes;
```

  - Można użyć podobnego sposobu z próbą kompilacji i usunięcia problematycznych linijek kodu.
- Funkcja renderująca scenę powinna rysować wszystkie obiekty z kontenera nie uwzględniając pojedynczych siatek.
- Funkcja *GLwidget::paintGL()* w pliku **glwidget.cpp** musi być zmodyfikowana w taki sposób, by iterowała po wektorze *m_gameObjects* i renderowała, ustawiając wcześniej macierz modelu i przekazując parametry materiału obiektu do programu cieniującego. **Kod jest zawarty na kolejnym slajdzie**.

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Klasa GLWidget
Renderowanie obiektów gry

Klasa GameObject

Klasa GLWidget

Rendering obiektów

Aktualizacja obiektów

Klasa Cube

Kolizje

Pocisk

```
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, QVector3D(0.0f, 0.0f, 15.0f));
   m_program->setUniformValue(m_lightLoc.ambient, QVector3D(0.1f, 0.1f, 0.1f));
   m_program->setUniformValue(m_lightLoc.diffuse, QVector3D(0.9f, 0.9f, 0.9f));

   m_world.setToIdentity();
   m_camera.setToIdentity();

   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++) {
       GameObject* obj = m_gameObjects[i];

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

       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;
   if(deltaTime >= (1.0f / FPS)) {
      updateGL();
      lastUpdateTime = timerTime;
   }
   update();
}
```

Wydział Informatyki

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Klasa GLWidget
Aktualizacja obiektów gry

Klasa GameObject
Klasa GLWidget
Rendering obiektów
Aktualizacja obiektów
Klasa Cube
Kolizje
Pocisk

- Oprócz renderowania obiektów w grze, trzeba je także aktualizować.
- Poprzednio zaktualizowaliśmy świat w funkcji tylko po wykryciu wejścia z klawiatury. Elementy wirtualnych światów często żyją także bez reakcji ze strony gracza, dlatego trzeba rozbudować nasz silnik o taką możliwość.
- W funkcji *GLWidget::updateGL()* w pliku **glwidget.cpp** należy przeiterować po obiektach w kontenerze *m_gameObjects* i uruchomić dla każdego z nich metodę *Update()*.

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

        obj->update();
}
```

- Na razie w metodzie *update()* nic się nie wykonuję, ale mamy już możliwość, aby elementy świata poruszały się samoczynnie.

Wydział
Informatyki

Przykład dodania nowego obiektu Klasa Cube

Klasa GameObject
Klasa GLWidget
Rendering obiektów
Aktualizacja obiektów
Klasa Cube
Kolizje
Pocisk

- W przypadku potrzeby dodania nowego rodzaju obiektu do gry (tj. przeciwnik) należy stworzyć klasę, która dziedziczy po klasie *GLwidget*.
- **Stwórz klasę Cube** i następnie rozbuduj tą klasę o metody abstrakcyjne klasy *GLwidget* (*init()*, *render()* i *update()*), oraz dodatkowo siatkę *m_mesh* w **cube.h**.

```
#include "gameobject.h"
#include "cmesh.h"

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

   void init();
   void render(GLWidget* glwidget);
   void update();

   CMesh m_mesh;
};
```

- Zaimplementuj ciała tych metod w **cube.cpp**.

```
Cube::Cube()
{
        m_name = "cube";
}


void Cube::init()
{
        m_mesh.generateCube(1.0f, 1.0f, 1.0f);
}
```

```
void Cube::render(GLWidget* glwidget)
{
        m_mesh.render(glwidget);
}


void Cube::update()
{

}
```

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Przykład dodania nowego obiektu Klasa Cube

Klasa GameObject
Klasa GLWidget
Rendering obiektów
Aktualizacja obiektów
Klasa Cube
Kolizje
Pocisk

- Dodajmy teraz kilka Cube'ów do sceny.
- W funkcji *GLWidget::initializeGL()* w **glwidget.cpp** użyj poniższego kodu (należy dodać **#include "cube.h"**).

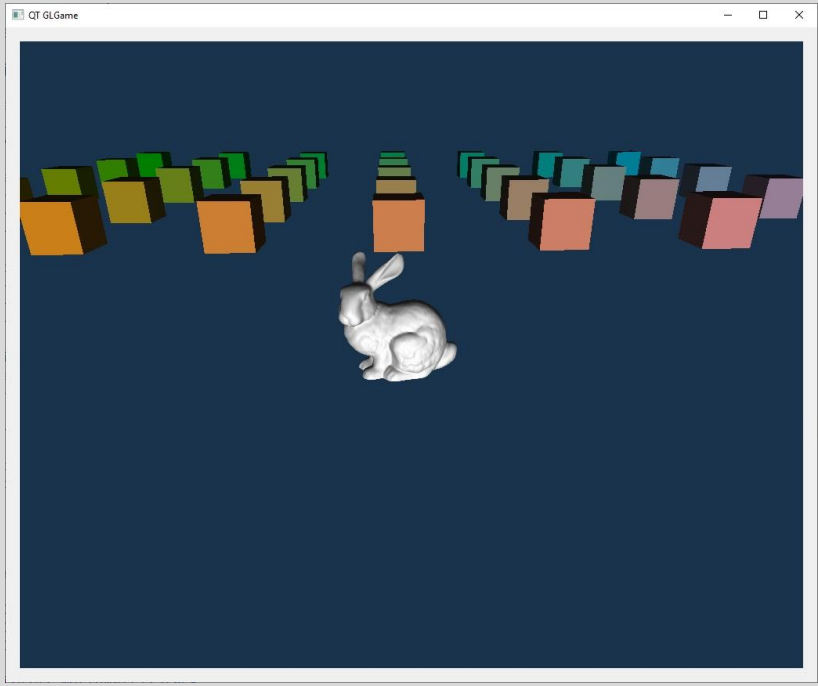
```
for(int i = 0 ; i < 5 ; i++)
  {
    for(int j = 0 ; j < 7 ; j++)
    {
      // Wskaźnik do Cube'a z tworzeniem obiektu.
      // MUSI BYĆ WSKAŹNIK !!!
      Cube* cube = new Cube();

      // Ustawienie pozycji Cube'a
      cube->position.setX(j * 1 - 3);
      cube->position.setY(0);
      cube->position.setZ(i * 1 - 6);

      // Kolor Cube'a. Zwykły gradient.
      cube->material_color.setX(i * 0.2f);
      cube->material_color.setY(0.5f);
      cube->material_color.setZ(j * 0.1f);

      // Wielkość Cube'a.
      // Ustawienie w jednej linijce
      // zamiast osobno dla X, Y i Z.
      cube->scale = QVector3D(0.3f, 0.3f, 0.3f);

      // Dodanie obiektu do sceny.
      addObject(cube);
    }
  }
```

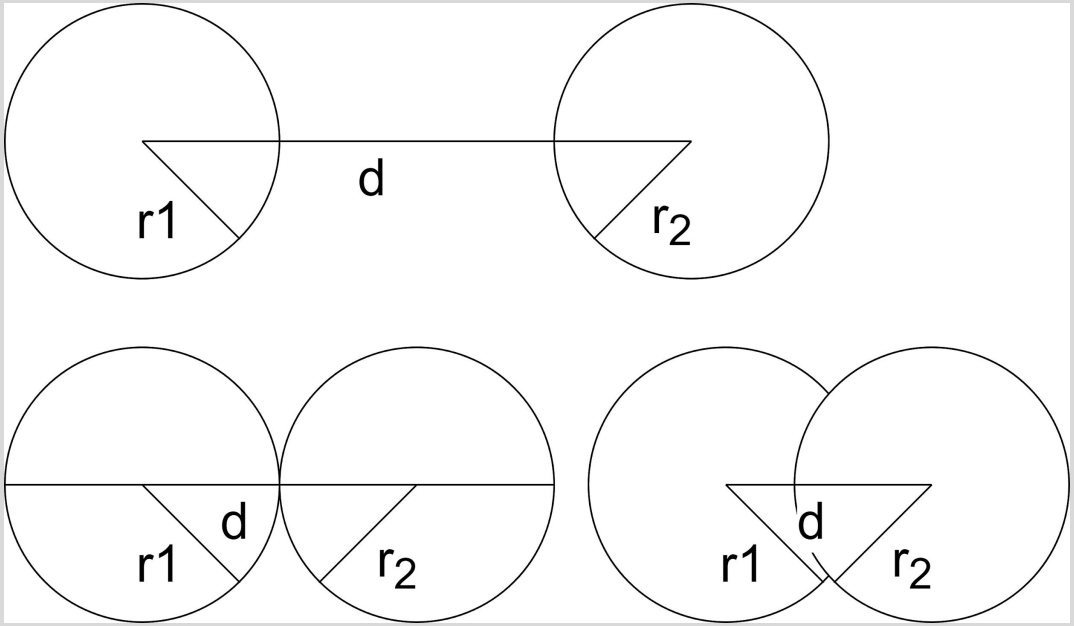


Wydział
Informatyki

Kolizje

Klasa GameObject
Klasa GLWidget
Rendering obiektów
Aktualizacja obiektów
Klasa Cube
Kolizje
Pocisk

- Wprowadzone zmiany w strukturze projektu pozwalają na implementację prostego systemu wykrywania kolizji między obiektami sceny.
- System ten będzie się opierał na tym, że każdy obiekt otaczamy niewidzialną kulą zwaną **collider'em** o pozycji równej pozycji obiektu oraz ustalonemu promieniowi.
- W sytuacji wykrycia kolizji dwóch colliderów, odpowiednio reagujemy, np. odpychamy obiekty od siebie.



Kolizje

Klasa GameObject
Klasa GLWidget
Rendering obiektów
Aktualizacja obiektów
Klasa Cube
Kolizje
Pocisk

- Aby wykryć kolizję, **należy obliczyć dla każdej pary kul odległość od ich środków i porównać z sumą ich promieni**. Jeśli na suma jest mniejsza, wykryto kolizję.
- Reakcja na kolizję może się różnić w zależności od rodzajów obiektów.
- **Nie należy porównywać obiektów samych ze sobą!**
- Pętle iterującą po obiektach w *GLWidget::updateGL()* w **glwidget.cpp** można rozbudować o wykrywanie kolizji w następujący sposób.

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

    // Porównujemy każdy obiekt z każdym
    for(int j = 0 ; j < m_gameObjects.size() ; j++)
    {
      if(i == j) // Nie porównujemy obiektów samych ze sobą
        continue;

      GameObject* obj2 = m_gameObjects[j];

      // Liczymy wektor od pozycji jednego obiektu do drugiego
      QVector3D v = obj->position - obj2->position;
      // Długość tego wektora to odległość między środkami obiektów
      float d = v.length();

      // Porównujemy z sumą promieni
      if(d < (obj->m_radius + obj2->m_radius))
      {
        // Reakcja na kolizję!
      }
    }

    obj->update();
  }
```

Fizyka w grach
Koncepcja energii

Klasa GameObject
Klasa GLWidget
Rendering obiektów
Aktualizacja obiektów
Klasa Cube
Kolizje
Pocisk

- Możemy stworzyć założenie, że każdy z obiektów trzyma w sobie pewną **energię**, która jest przechowywana w obiekcie jako vector trzy elementowy. Tą energię obiekt w pewien sposób wykorzystuje, na przykład do poruszania się. W sytuacji zderzenia się dwóch obiektów, energia ta jest rozdzielana na dwa obiekty.

- **Zaprezentowane rozwiązanie nie jest zgodne z prawami fizycznymi obowiązujących na świecie, a są dużym uproszczeniem użytym do implementacji w grze.**

- Do klasy *GameObject* w pliku **gameObject.h** dodajmy wektor zawierają tę energię.

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

- Obiekty te będą używać tej energii po przemieszczania się, wytrącając ją. W funkcjach *Player::update()* (**player.cpp**) oraz *Cube::update()* (**cube.cpp**) dodaj następujące linijki kodu.

```
// wykorzystanie energii.
position = position + energy;

// wytracanie energii.
energy = energy / 1.2f;
```

- Należy także dodać do miejsca gdzie dodawaliśmy Cube'y do sceny obliczenie ich promienia (*GLWidget::initializeGL()*, **glwidget.cpp**).

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

Fizyka w grach
Koncepcja energii

Klasa GameObject
Klasa GLWidget
Rendering obiektów
Aktualizacja obiektów
Klasa Cube
Kolizje
Pocisk

- Wykonajmy zmianę w sterowaniu gracza. Otóż teraz, zamiast zmieniać jego pozycję za pomocą klawiatury, dodajmy do wektora energi wartość zależną od obranego kierunku. Zamień **position** na **energy** w *GLWidget::updateGL()* (**glwidget.cpp**).

```
if(m_keyState[Qt::Key_W])
{
	m_player.energy.setX(m_player.energy.x() + m_player.direction.x() * m_player.speed);
	m_player.energy.setZ(m_player.energy.z() + m_player.direction.z() * m_player.speed);
}
if(m_keyState[Qt::Key_S])
{
		m_player.energy.setX(m_player.energy.x() - m_player.direction.x() * m_player.speed);
		m_player.energy.setZ(m_player.energy.z() - m_player.direction.z() * m_player.speed);
}
if(m_keyState[Qt::Key_A])
{
		m_player.energy.setX(m_player.energy.x() + m_player.direction.z() * m_player.speed);
		m_player.energy.setZ(m_player.energy.z() - m_player.direction.x() * m_player.speed);
}
if(m_keyState[Qt::Key_D])
{
		m_player.energy.setX(m_player.energy.x() - m_player.direction.z() * m_player.speed);
		m_player.energy.setZ(m_player.energy.z() + m_player.direction.x() * m_player.speed);
}
```

- Następnie w poniższy sposób wykonaj reakcję na kolizję także w *GLWidget::updateGL()* (**glwidget.cpp**) wewnątrz po porównaniu dystansuz sumą promieni.

```
// Reakcja na kolizję!
v.normalize();
float energySum = obj->energy.length() + obj2->energy.length();
obj->energy = v * energySum / 2;
obj2->energy = -v * energySum / 2;
```

Klasa GameObject
Klasa GLWidget
Rendering obiektów
Aktualizacja obiektów
Klasa Cube
Kolizje
Pocisk

Przykłady
Pociski

- Mając już zaimplementowany prosty system kolizji, możemy zaimplementować coś na kształt pocisku, który będzie tworzyć po wciśnięciu przycisku, i który będzie z czasem znikał.

- Stwórz nową klasę **Bullet** i wypełnij ciało tej klasy w pliku **bullet.h** tak samo jak w przypadku klasy *Cube* na **slajdzie 10**.

- Wypełnij ciała tych funkcji w **bullet.cpp** w następujący sposób.

```
Bullet::Bullet()
{

}

void Bullet::init()
{
    m_mesh.generateSphere(0.5f, 24);
}

void Bullet::render(GLWidget* glwidget)
{
    m_name = "bullet";
    m_mesh.render(glwidget);
}

void Bullet::update()
{
    position = position + energy * 0.3f;
    energy = energy / 1.1f;

    float energySum = energy.length();
    m_radius = energySum * 0.3f;
    scale = QVector3D(m_radius, m_radius, m_radius);

}
```

Przykłady
Tworzenie pocisków

Klasa GameObject
Klasa GLWidget
Rendering obiektów
Aktualizacja obiektów
Klasa Cube
Kolizje
Pocisk

- Teraz należy zaimplementować tworzenie obiektu *Bullet* po wciśnięciu przycisku.
- W funkcji *GLWidget::keyPressEvent* w **glwidget.cpp** dodaj następujący kod.

```
if (e->key() == Qt::Key_Escape)
        exit(0);
else if (e->key() == Qt::Key_Space)
{
        Bullet* bullet = new Bullet();

        bullet->position = m_player.position + m_player.direction * 0.7f;
        bullet->position.setY(0);
        bullet->scale = QVector3D(0.5f, 0.5f, 0.5f);
        bullet->m_radius = 0.5f;
        bullet->energy = 3 * m_player.direction;
        bullet->energy.setY(0);

        addObject(bullet);
}
else
        QWidget::keyPressEvent(e);
```

Klasa GameObject
Klasa GLWidget
Rendering obiektów
Aktualizacja obiektów
Klasa Cube
Kolizje
Pocisk

Przykłady
Usuwanie pocisków

- Do programu można dodać możliwość usuwania obiektów.
- W klasie *GameObject* w **gameobject.h** dodajmy nowe pole.

```
bool isAlive = true;
```

- Na końcu funkcji *GLWidget::updateGL()* w **glwidget.cpp** dodajmy dodatkową pętlę sprawdzającą, czy jest jakiś “martwy” obiekt i usunie go.

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

    if(obj->isAlive == false)
    {
      m_gameObjects.erase(m_gameObjects.begin() + i);
      delete obj;
    }
    else
      i++;
  }
```

- Usuńmy pocisk, gdy ma on zbyt małą energię. W funkcji *Bullet::update()* w **bullet.cpp** wstaw następujący kod.

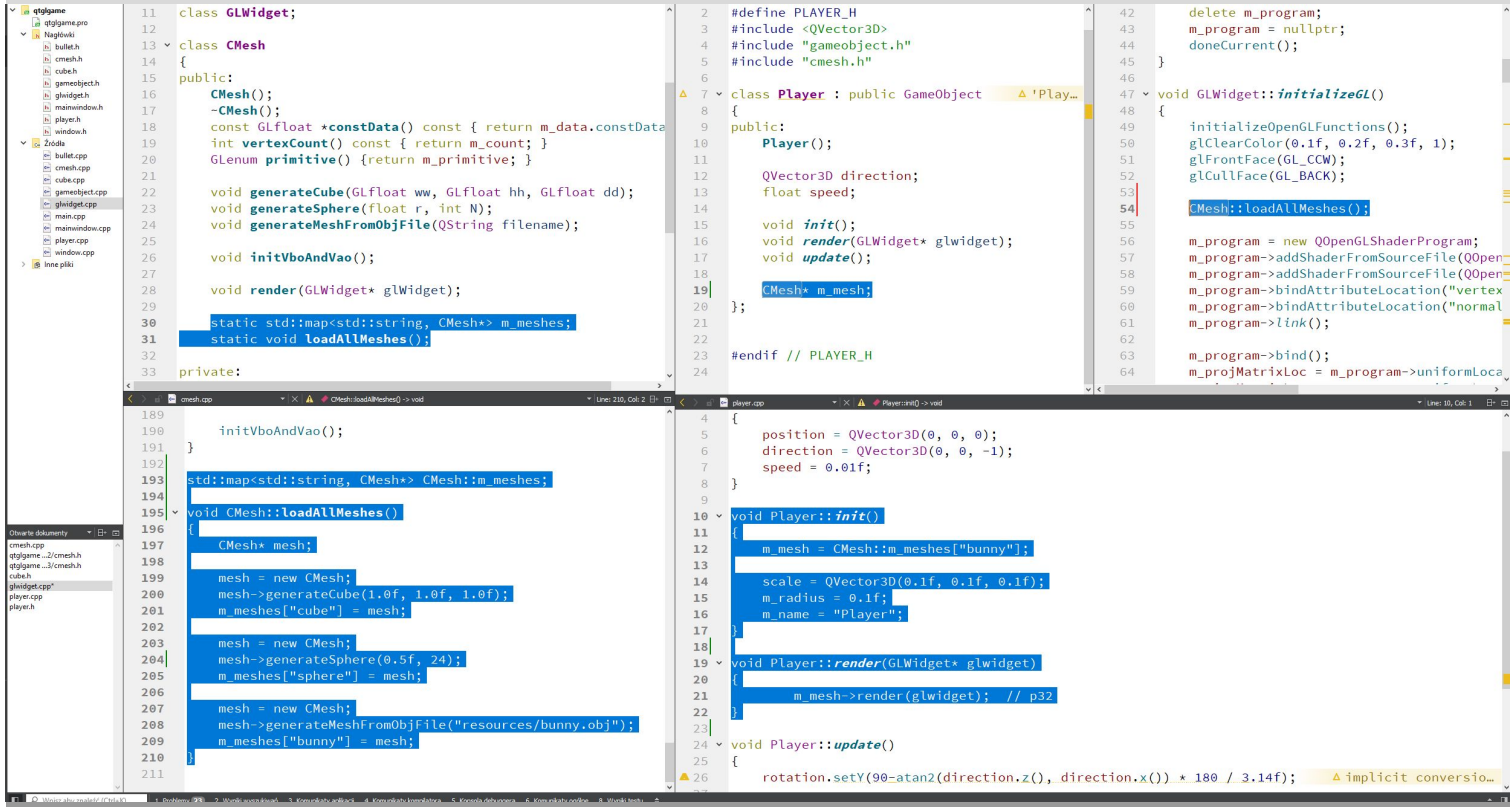
```
if(energySum < 0.1f)
    isAlive = false;
```

Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Mesh Management

- Należy dodać lepszą obsługę siatek w projekcie.
- Na poniższym zrzucie jest zaimplementowany prosty Mesh Manager.
- W przykładzie jest przykład jak użyć go w klasie Player. **Należy użyć go także w klasach Bullet i Cube**.

Klasa GameObject
Klasa GLWidget
Rendering obiektów
Aktualizacja obiektów
Klasa Cube
Kolizje
Pocisk



```
class GLWidget;

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:
```

```
#define PLAYER_H
#include <QVector3D>
#include "gameobject.h"
#include "cmesh.h"

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

    QVector3D direction;
    float speed;

    void init();
    void render(GLWidget* glwidget);
    void update();

    CMesh* m_mesh;
};


#endif // PLAYER_H
```

```
    delete m_program;
    m_program = nullptr;
    doneCurrent();
}

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(QOpen
    m_program->addShaderFromSourceFile(QOpen
    m_program->bindAttributeLocation("vertex
    m_program->bindAttributeLocation("normal
    m_program->link();

    m_program->bind();
    m_projMatrixLoc = m_program->uniformLoca
```

```
    initVboAndVao();
}

std::map<std::string, CMesh*> CMesh::m_meshes;

void CMesh::loadAllMeshes()
{
    CMesh* mesh;

    mesh = new CMesh;
    mesh->generateCube(1.0f, 1.0f, 1.0f);
    m_meshes["cube"] = mesh;

    mesh = new CMesh;
    mesh->generateSphere(0.5f, 24);
    m_meshes["sphere"] = mesh;

    mesh = new CMesh;
    mesh->generateMeshFromObjFile("resources/bunny.obj");
    m_meshes["bunny"] = mesh;
}
```

```
{
    position = QVector3D(0, 0, 0);
    direction = QVector3D(0, 0, -1);
    speed = 0.01f;
}

void Player::init()
{
    m_mesh = CMesh::m_meshes["bunny"];

    scale = QVector3D(0.1f, 0.1f, 0.1f);
    m_radius = 0.1f;
    m_name = "Player";
}
void Player::render(GLWidget* glwidget)
{
        m_mesh->render(glwidget);  // p32
}

void Player::update()
{
    rotation.setY(90-atan2(direction.z(), direction.x()) * 180 / 3.14f);
```

Wydział Informatyki

Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Typ obiektów w kolizji

- Aby sprawić, by gracz nie odpychał się od pocisków, trzeba sprawdzić nazwę obiektów.
- Można rozbudować sprawdzane kombinację o inne rodzaje obiektów.

Klasa GameObject

Klasa GLWidget

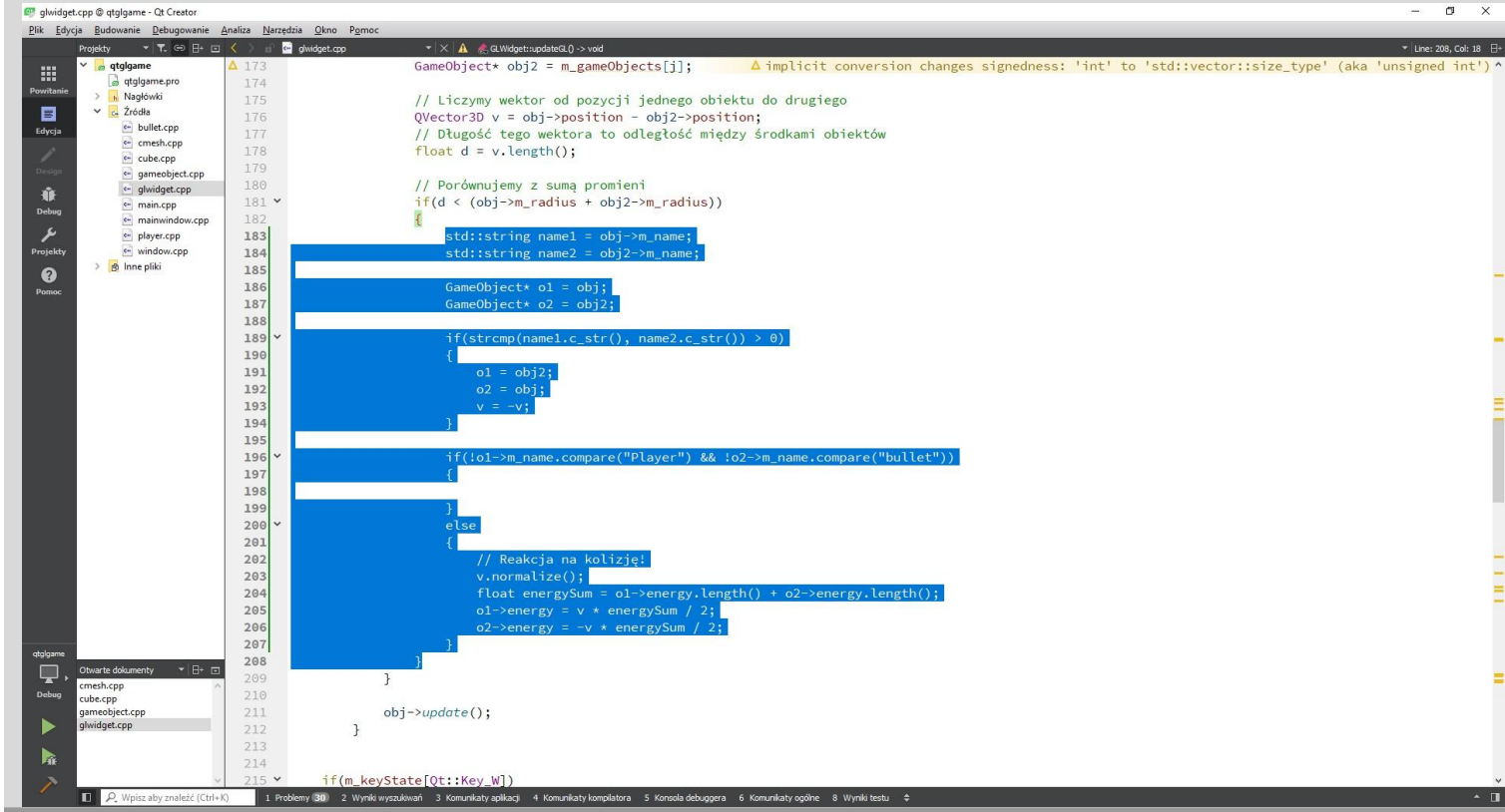
Rendering obiektów

Aktualizacja obiektów

Klasa Cube

Kolizje

Pocisk



Wydział Informatyki