

Tractor Rush: Carrots Edition

DOKUMENTACJA

Paweł Buczek (173599) L01 EF-DI
Bartłomiej Krówka (173650) L01 EF-DI

Spis treści:

1. Założenia projektu.....	2
2. Opis projektu.....	2
3. Wykorzystane technologie.....	2
4. Zasady gry.....	2
5. Instrukcja gry.....	2
6. Etapy powstawania projektu.....	3
6.1. Zbudowanie obiektu pojazdu.....	3
6.1.1. Opis prac.....	3
6.1.2. Omówienie kodu.....	4
6.1.3. Podsumowanie.....	12
6.2. Stworzenie świata gry.....	12
6.2.1. Opis prac.....	12
6.2.2. Omówienie kodu.....	13
6.2.3. Podsumowanie.....	14
6.3. Teksturowanie obiektów i sterowanie kamerą.....	14
6.3.1. Opis prac.....	14
6.3.2. Omówienie kodu.....	15
6.3.3. Podsumowanie.....	17
6.4. Implementacja sterowania.....	17
6.4.1. Opis prac.....	17
6.4.2. Omówienie kodu.....	18
6.4.3. Podsumowanie.....	19
6.5. Implementacja kolizji.....	19
6.5.1. Opis prac.....	19
6.5.2. Omówienie kodu.....	20
6.5.3. Podsumowanie.....	21
6.6. Opracowanie fabuły gry.....	21
6.6.1. Opis prac.....	21
6.6.2. Omówienie kodu.....	22
6.6.3. Podsumowanie.....	25
7. Podsumowanie.....	25

1. Założenia projektu

Celem projektu jest stworzenie gry trójwymiarowej, zaimplementowanej w języku C++ przy użyciu biblioteki OpenGL. W ramach wymagań projektowych, przewiduje się konstrukcję sterowanego obiektu, kreację świata gry, teksturowanie wszystkich elementów 3D, implementację mechanizmów sterowania oraz symulację podstawowych zjawisk fizycznych. Dodatkowo, projekt zakłada opracowanie celu rozgrywki, który będzie motywować graczy do kontynuowania gry.

2. Opis projektu

Gra komputerowa umożliwia wcielenie się w rolę operatora zdalnie sterowanego pojazdu rolniczego, zadaniem którego jest zbieranie marchewek w trudno dostępnych lokalizacjach. Elementy świata gry są umieszczone w sposób statyczny, podczas gdy warzywa są generowane losowo, co gwarantuje niepowtarzalność każdej sesji. Każdy obiekt trójwymiarowy został odpowiednio oteksturowany. W ramach projektu zaimplementowano mechanizmy sterowania oraz symulację podstawowych zjawisk fizycznych, takich jak kolizje, pęd i siła odśrodkowa. Dzięki temu gra oferuje ciekawe doświadczenie związane z prowadzeniem pojazdu rolniczego.

3. Wykorzystane technologie

język programowania: ISO C++ 14

biblioteki: freeglut 3.0.0-1, glew-2.1.0, irrKlang-64bit-1.6.0, tinyobjloader v1.0.0, stb-image

IDE: Microsoft Visual Studio 2022 Community Edition

4. Zasady gry

Pojazd tworzony jest na początku układu współrzędnych. Zadaniem gracza jest odnalezienie i zebranie wszystkich 20 marchewek w jak najkrótszym czasie, jednocześnie unikając jakichkolwiek kolizji. Każde uderzenie skutkuje nałożeniem kary czasowej. Wyniki najlepszych graczy są rejestrowane i zapisywane w tablicy wyników.

5. Instrukcja gry

Sterowanie pojazdem rolniczym jest realizowane za pomocą standardowego zestawu klawiszy: W, A, S, D. Odległość kamery od obiektu reguluje się klawiszem Z. Gra wstrzymuje się po naciśnięciu klawisza P, natomiast sesję można zrestartować za pomocą klawisza BACKSPACE. Cała klawiszologia wyświetlana jest po naciśnięciu TAB. Interfejs użytkownika dostarcza informacji na temat aktualnego poziomu przybliżenia, liczby zebranych marchewek, aktualnego czasu gry oraz ewentualnej kary czasowej, a także informuje o aktualnej prędkości osiąganej przez pojazd.

6. Etapy powstawania projektu

6.1. Zbudowanie obiektu pojazdu

6.1.1. Opis prac

Pierwszym krokiem do zbudowania głównego obiektu gry komputerowej było dołączenie do projektu biblioteki freeglut, a następnie utworzenie brył z których składa się pojazd. Każdy z elementów zdefiniowany jest w osobnej klasie. Cały pojazd zawiera 31 brył.

Wykorzystane elementy:

- 6 kół
- 3 osie
- podwozie
- 4 światła
- 2 obramowania tablicy rejestracyjnej
- 2 tablice rejestracyjne
- 2 kierunkowskazy
- pokrywa silnika
- kabina
- 2 rury wydechowe
- 2 tłumiki
- 4 słupki
- dach



Rys. 6.1.1. - pojazd rolniczy

6.1.2. Omówienie kodu

```
class Robot
{
public:
    Robot();
    float x, y, z, rotationAngle, inclinationDegree;
    bool isA, isD;
    void setRotate(float isA, float isD);
    void draw();
};
```

Kod 6.1.1. - klasa Robot

Klasa Robot zawiera metody do rysowania pojazdu i ustawiania jego obrotu. Posiada również zmienne do przechowywania jego pozycji, kąta obrotu, stopnia nachylenia i stanów klawiszy 'A' i 'D'. Metoda draw() jest odpowiedzialna za rysowanie różnych części, w odpowiednim miejscu i orientacji.

```
Robot::Robot() {
    x = 0.0f;
    y = 0.0f;
    z = 0.5f;
    rotationAngle = 0.0f;
    isA = false;
    isD = false;
    inclinationDegree = 0.0f;
}
```

Kod 6.1.2. - konstruktor klasy Robot

Konstruktor klasy inicjalizuje wszystkie zmienne pola klasy na wartości początkowe. Wszystkie te inicjalizacje są wykonywane, aby umożliwić dalsze operacje, takie jak rysowanie i obracanie.

```
void Robot::setRotate(float isA, float isD)
{
    this->isA = isA;
    this->isD = isD;
}
```

Kod 6.1.3. - metoda setRotate

Metoda setRotate jest odpowiedzialna za przekazanie informacji o tym, który kierunkowskaz powinien się zaświecić. Przyjmuje dwa argumenty typu float: isA i isD, które reprezentują stan klawiszy 'A' i 'D'. W zależności od wartości tych argumentów, metoda setRotate aktualizuje odpowiednie zmienne członkowskie klasy.

```

void Robot::draw() {
    glPushMatrix();
    glTranslatef(x, y, z);
    glRotatef(rotationAngle, 0.0f, 0.0f, 1.0f);
    glRotatef(inclinationDegree, 1.0f, 0.0f, 0.0f);
    glTranslatef(-x, -y, -z);
    Korpus korpus(x, y, z); //wymiary: 3.0f, 1.0f, 0.4f
    korpus.draw();
    //KÓŁKA LEWE
    for (float i = 0; i < 3; i++) {
        Kolo kolo(x - 1.5f + i * 1.5f, y + 0.63f, z-0.2f); //wymiary 0.5f, 0.5f
        kolo.setTexture("textures/tire2.png");
        kolo.draw();
    }
    //KÓŁKA PRAWE
    for (float i = 0; i < 3; i++) {
        Kolo kolo(x - 1.5f + i * 1.5f, y - 0.63f, z-0.2f); //wymiary 0.5f, 0.5f
        kolo.setTexture("textures/tire.jpg");
        kolo.draw();
    }
    for (float i = 0; i < 3; i++) {
        Lacznik lacznik(x - 1.5f + i * 1.5f, y, z - 0.2f);
        lacznik.draw();
    }
    for (float i = 0; i < 2; i++) {
        Tablica tablica(x - 1.5f + i * 3.0f, y, z);
        tablica.draw();
    }
    Przod przod(x-0.3f, y-0.5f, z + 0.2f);
    przod.draw();
    Tyl tyl(x-0.9f, y, z+0.65f);
    tyl.draw();
    for (float i = 0.0; i < 2; i++) {
        Swiatlo swiatlo(x - 1.5f, y - 0.35f + 0.7 * i, z);
        swiatlo.draw();
    }
    for (float i = 0.0; i < 2; i++) {
        Swiatlo swiatlo(x + 1.5f, y - 0.35f + 0.7 * i, z);
        swiatlo.draw();
    }
    for (float i = 0.0; i < 2; i++) {
        Slupek slupek(x - 1.45f, y + 0.45f - 0.9 * i, z + 1.25f);
        slupek.draw();
    }
    for (float i = 0.0; i < 2; i++) {
        Slupek slupek(x - 0.35f, y + 0.45f - 0.9 * i, z + 1.25f);
        slupek.draw();
    }
    Dach dach(x - 0.9f, y, z + 1.5f);
    dach.draw();
    for (float i = 0.0; i < 2; i++) {
        Rura rura(x + 0.3f, y + 0.25f + 0.15 * i, z + 1.10f);
        rura.draw();
        Tlumik tlumik(x + 0.3f, y + 0.25f + 0.15 * i, z + 1.10f);
        tlumik.draw();
    }
    for (int i = 0.0; i < 2; i++) {
        if (i == 0) {
            Kierunkowskaz kierunkowskaz(x - 0.3f, y + 0.5f - i * 1.0, z + 0.55f, isA);
            kierunkowskaz.draw();
        }
        else {
            Kierunkowskaz kierunkowskaz(x - 0.3f, y + 0.5f - i * 1.0, z + 0.55f, isD);
            kierunkowskaz.draw();
        }
    }
    glPopMatrix();
}

```

Kod 6.1.4. - metoda draw

Metoda `draw` jest odpowiedzialna za rysowanie pojazdu w przestrzeni 3D. Na początku, metoda zapisuje aktualny stan macierzy za pomocą funkcji `glPushMatrix()`. Następnie przesuwa układ współrzędnych o wartości (`x`, `y`, `z`), co odpowiada pozycji pojazdu. Potem obraca układ współrzędnych o `rotationAngle` wokół osi `Z` i `inclinationDegree` wokół osi `X`, co odpowiada obrotowi i nachyleniu. Po tych transformacjach, metoda `draw()` rysuje różne części, takie jak koła, osie, podwozie itd., każda z nich reprezentowana przez osobną klasę z własną metodą `draw()`. Na koniec, metoda przywraca poprzedni stan macierzy za pomocą `glPopMatrix()`, co pozwala na rysowanie innych obiektów w ich własnych układach współrzędnych. W ten sposób, metoda pozwala na dynamiczne rysowanie pojazdu z uwzględnieniem jego pozycji, obrotu i nachylenia.

Komponenty pojazdu składają się głównie z prostopadłościanów i walców, które różnią się pod względem wymiarów oraz punktów, na których są renderowane. Wyjątkiem jest pokrywa silnika, która ma kształt graniastosłupa trójkątnego.

```
class Korpus
{
public:
    Korpus(float x, float y, float z);
    void draw();

private:
    float x, y, z;
};
```

Kod 6.1.5. - klasa Korpus

Klasa `Korpus` zawiera konstruktor, który przyjmuje trzy argumenty typu `float`: `x`, `y` i `z`, które reprezentują pozycję podwozia w przestrzeni 3D. Zawiera również metodę `draw()`, która jest używana do rysowania elementu. Zawiera trzy prywatne zmienne członkowskie: `x`, `y` i `z`, które przechowują współrzędne pozycji korpusu.

```

void Korpus::draw() {
    glPushMatrix();
    glTranslatef(x, y, z);

    float halfWidth = 3.0f / 2.0f;
    float halfHeight = 1.0f / 2.0f;
    float halfDepth = 0.4f / 2.0f;

    glColor3f(0.2f, 0.2f, 0.2f);
    glBegin(GL_TRIANGLES);

    // Przód
    glVertex3f(-halfWidth, halfHeight, halfDepth);
    glVertex3f(halfWidth, halfHeight, halfDepth);
    glVertex3f(halfWidth, -halfHeight, halfDepth);

    glVertex3f(-halfWidth, halfHeight, halfDepth);
    glVertex3f(halfWidth, -halfHeight, halfDepth);
    glVertex3f(-halfWidth, -halfHeight, halfDepth);

    // Tył
    glVertex3f(halfWidth, halfHeight, -halfDepth);
    glVertex3f(-halfWidth, halfHeight, -halfDepth);
    glVertex3f(-halfWidth, -halfHeight, -halfDepth);

    glVertex3f(halfWidth, halfHeight, -halfDepth);
    glVertex3f(-halfWidth, -halfHeight, -halfDepth);
    glVertex3f(halfWidth, -halfHeight, -halfDepth);

    // Góra
    glVertex3f(-halfWidth, halfHeight, halfDepth);
    glVertex3f(halfWidth, halfHeight, halfDepth);
    glVertex3f(halfWidth, halfHeight, -halfDepth);

    glVertex3f(-halfWidth, halfHeight, halfDepth);
    glVertex3f(halfWidth, halfHeight, -halfDepth);
    glVertex3f(-halfWidth, halfHeight, -halfDepth);

    // Dół
    glVertex3f(halfWidth, -halfHeight, halfDepth);
    glVertex3f(-halfWidth, -halfHeight, halfDepth);
    glVertex3f(-halfWidth, -halfHeight, -halfDepth);

    glVertex3f(halfWidth, -halfHeight, halfDepth);
    glVertex3f(-halfWidth, -halfHeight, -halfDepth);
    glVertex3f(halfWidth, -halfHeight, -halfDepth);

    // Lewa strona
    glVertex3f(-halfWidth, halfHeight, halfDepth);
    glVertex3f(-halfWidth, -halfHeight, halfDepth);
    glVertex3f(-halfWidth, -halfHeight, -halfDepth);

    glVertex3f(-halfWidth, halfHeight, halfDepth);
    glVertex3f(-halfWidth, -halfHeight, -halfDepth);
    glVertex3f(-halfWidth, halfHeight, -halfDepth);

    // Prawa strona
    glVertex3f(halfWidth, halfHeight, halfDepth);
    glVertex3f(halfWidth, -halfHeight, halfDepth);
    glVertex3f(halfWidth, -halfHeight, -halfDepth);

    glVertex3f(halfWidth, halfHeight, halfDepth);
    glVertex3f(halfWidth, -halfHeight, -halfDepth);
    glVertex3f(halfWidth, halfHeight, -halfDepth);

    glEnd();

    glPopMatrix();
}

```

Kod 6.1.6. - metoda draw rysująca prostopadłościan

Metoda jest odpowiedzialna za rysowanie podwozia. Na początku, zapisuje aktualny stan macierzy za pomocą `glPushMatrix()`. Następnie przesuwa układ współrzędnych o wartości (x , y , z), co odpowiada pozycji pojazdu. Potem oblicza połowę szerokości, wysokości i głębokości korpusu. Ustawia kolor rysowania na ciemnoszary za pomocą `glColor3f(0.2f, 0.2f, 0.2f)`. Rozpoczyna rysowanie trójkątów za pomocą `glBegin(GL_TRIANGLES)`. Następnie rysuje sześć ścian korpusu (przód, tył, góra, dół, lewa strona, prawa strona). Każda ściana jest rysowana jako dwa trójkąty, a każdy trójkąt jest definiowany przez trzy punkty (`glVertex3f`). Kończy rysowanie trójkątów za pomocą `glEnd()`. Na koniec, metoda przywraca poprzedni stan macierzy za pomocą `glPopMatrix()`. W ten sposób pozwala na dynamiczne rysowanie podwozia z uwzględnieniem pozycji pojazdu.

Każdy inny prostopadłościenny element obiektu rysowany jest na takiej samej zasadzie.

```
class Kolo {
public:
    Kolo(float x, float y, float z);
    void setTexture(const char* textureName);
    void draw();

private:
    float x, y, z;           // Położenie koła
    int sides;               // Liczba boków koła (ilość trójkątów)
    const float height = 0.2f;
    const float radius = 0.2f;
    const char* textureName = "";
};
```

Kod 6.1.6. - klasa Kolo

Klasa Kolo zawiera konstruktor, który przyjmuje trzy argumenty typu float: x , y i z , które reprezentują pozycję koła w przestrzeni 3D. Zawiera również metody `setTexture(const char* textureName)` i `draw()`, które są używane do ustawiania tekstury koła i rysowania koła. Posiada prywatne zmienne członkowskie do przechowywania współrzędnych pozycji koła, liczby boków koła, wysokości, promienia i nazwy tekstury koła.

```

void Kolo::draw() {
    glPushMatrix();
    glTranslatef(x, y, z);

    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, loadTexture(textureName));

    // Rysuj górną powierzchnię kola
    glBegin(GL_TRIANGLE_FAN);
    glTexCoord2f(0.5f, 0.5f);
    glVertex3f(0.0f, height / 2.0f, 0.0f);

    for (int i = 0; i <= sides; i++) {
        float angle = 2.0f * 3.14 * (float)i / (float)sides;
        glTexCoord2f(0.5f * cos(angle) + 0.5f, 0.5f * sin(angle) + 0.5f);
        glVertex3f(radius * cos(angle), height / 2.0f, radius * sin(angle));
    }
    glEnd();

    // Rysuj dolną powierzchnię kola
    glBegin(GL_TRIANGLE_FAN);
    glTexCoord2f(0.5f, 0.5f);
    glVertex3f(0.0f, -height / 2.0f, 0.0f);

    for (int i = 0; i <= sides; i++) {
        float angle = 2.0f * 3.14 * (float)i / (float)sides;
        glTexCoord2f(0.5f * cos(angle) + 0.5f, 0.5f * sin(angle) + 0.5f);
        glVertex3f(radius * cos(angle), -height / 2.0f, radius * sin(angle));
    }
    glEnd();

    // Rysuj boki kola za pomocą trójkątów
    glBegin(GL_TRIANGLES);
    for (int i = 0; i < sides; i++) {
        float angle1 = 2.0f * 3.14 * (float)i / (float)sides;
        float angle2 = 2.0f * 3.14 * (float)(i + 1) / (float)sides;

        // Górska trójkątna ściana
        glTexCoord2f(0.5f * cos(angle1) + 0.5f, 0.5f * sin(angle1) + 0.5f);
        glVertex3f(radius * cos(angle1), height / 2.0f, radius * sin(angle1));

        glTexCoord2f(0.5f * cos(angle2) + 0.5f, 0.5f * sin(angle2) + 0.5f);
        glVertex3f(radius * cos(angle2), height / 2.0f, radius * sin(angle2));

        glTexCoord2f(0.5f * cos(angle2) + 0.5f, 0.5f * sin(angle2) + 0.5f);
        glVertex3f(radius * cos(angle2), -height / 2.0f, radius * sin(angle2));

        // Dulska trójkątna ściana
        glTexCoord2f(0.5f * cos(angle2) + 0.5f, 0.5f * sin(angle2) + 0.5f);
        glVertex3f(radius * cos(angle2), -height / 2.0f, radius * sin(angle2));

        glTexCoord2f(0.5f * cos(angle1) + 0.5f, 0.5f * sin(angle1) + 0.5f);
        glVertex3f(radius * cos(angle1), -height / 2.0f, radius * sin(angle1));

        glTexCoord2f(0.5f * cos(angle1) + 0.5f, 0.5f * sin(angle1) + 0.5f);
        glVertex3f(radius * cos(angle1), height / 2.0f, radius * sin(angle1));
    }
    glEnd();

    glDisable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, 0);

    glPopMatrix();
}

```

Kod 6.1.7. - metoda draw rysująca walec

Metoda jest odpowiedzialna za rysowanie koła. Na początku, zapisuje aktualny stan macierzy za pomocą `glPushMatrix()`. Następnie przesuwa układ współrzędnych o wartości (x , y , z), co odpowiada pozycji koła. Rysuje górną i dolną powierzchnię koła za pomocą trójkątów. Każda powierzchnia jest rysowana jako wachlarz trójkątów (`GL_TRIANGLE_FAN`), gdzie każdy trójkąt dzieli wspólny wierzchołek w środku koła. Potem rysuje boki koła za pomocą trójkątów. Boki są rysowane jako dwa trójkąty, które razem tworzą prostokąt. Na koniec, przywraca poprzedni stan macierzy za pomocą `glPopMatrix()`.

Każdy inny element obiektu tworzony na bazie walca rysowany jest na takiej samej zasadzie.

```
class Przod
{
public:
    Przod(float x, float y, float z);
    void draw();
private:
    float x, y, z;
};
```

Kod 6.1.8. - klasa Przod

Klasa Przod zawiera konstruktor, który przyjmuje trzy argumenty typu `float`: x , y i z , które reprezentują pozycję pokrywy silnika w przestrzeni 3D. Także zawiera metodę `draw()`, która jest używana do rysowania przedniej części. Klasa zawiera prywatne zmienne członkowskie do przechowywania współrzędnych pozycji.

```

void Przod::draw() {
    float szerokosc = 1.8f;
    float wysokosc = 0.7f;
    float glebokosc = 1.0f;

    glPushMatrix();
    glTranslatef(x, y, z);

    glBegin(GL_TRIANGLE_STRIP);
    glColor3d(0.2, 0.4, 0.2);
    for (float stepX = 0.0f; stepX <= szerokosc; stepX += 0.02f)
    {
        glVertex3f(stepX, 0.0f, 0.0f);
        glVertex3f(stepX, glebokosc, 0.0f);
    }
    glEnd();

    glBegin(GL_TRIANGLE_STRIP);
    glColor3d(0.2, 0.4, 0.2);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(szerokosc, 0.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, wysokosc);
    glEnd();

    glBegin(GL_TRIANGLE_STRIP);
    glColor3d(0.2, 0.4, 0.2);
    glVertex3f(0.0f, glebokosc, 0.0f);
    glVertex3f(szerokosc, glebokosc, 0.0f);
    glVertex3f(0.0f, glebokosc, wysokosc);
    glEnd();

    glBegin(GL_TRIANGLE_STRIP);
    glColor3d(0.2, 0.4, 0.2);
    for (float stepY = 0.0f; stepY <= glebokosc; stepY += 0.02f)
    {
        glVertex3f(0.0f, stepY, 0.0f);
        glVertex3f(0.0f, stepY, wysokosc);
    }
    glEnd();

    glBegin(GL_TRIANGLE_STRIP);
    glColor3d(0.2, 0.4, 0.2);
    glVertex3f(szerokosc, 0.0f, 0.0f);
    glVertex3f(szerokosc, glebokosc, 0.0f);
    glVertex3f(0.0f, 0.0f, wysokosc);
    glEnd();

    glPopMatrix();
}

```

Kod 6.1.9. - metoda draw rysująca graniastosłup trójkątny

Metoda jest odpowiedzialna za rysowanie przedniej pokrywy silnika. Na początku, zapisuje aktualny stan macierzy. Następnie przesuwa układ współrzędnych o wartości (x , y , z).

Metoda rysuje różne części pokrywy za pomocą pasm trójkątów (`GL_TRIANGLE_STRIP`). Każde pasmo trójkątów jest rysowane jako seria wierzchołków (`glVertex3f`), które definiują trójkąty. Rysuje pięć różnych pasm trójkątów, które odpowiadają różnym ścianom przedniej części pojazdu: górnej, dolnej, lewej, prawej i tylnej. Na koniec, przywraca poprzedni stan macierzy.

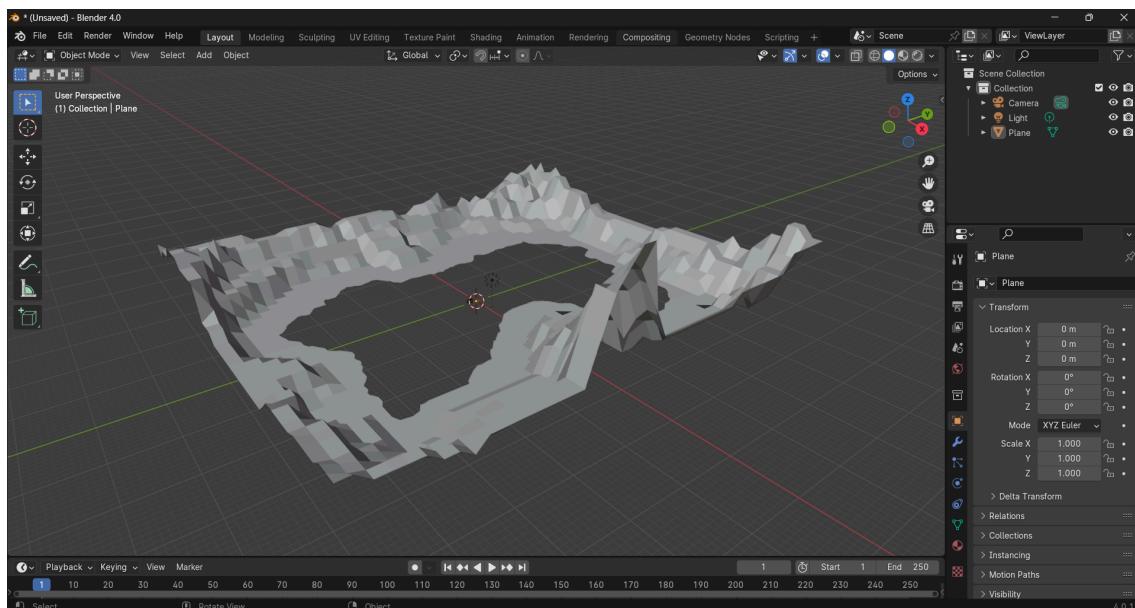
6.1.3. Podsumowanie

Bryły w OpenGL definiuje się jako połączone punkty w przestrzeni trójwymiarowej. Mogą one być następnie wykorzystane do konstruowania bardziej złożonych struktur, co umożliwia tworzenie rozbudowanych obiektów.

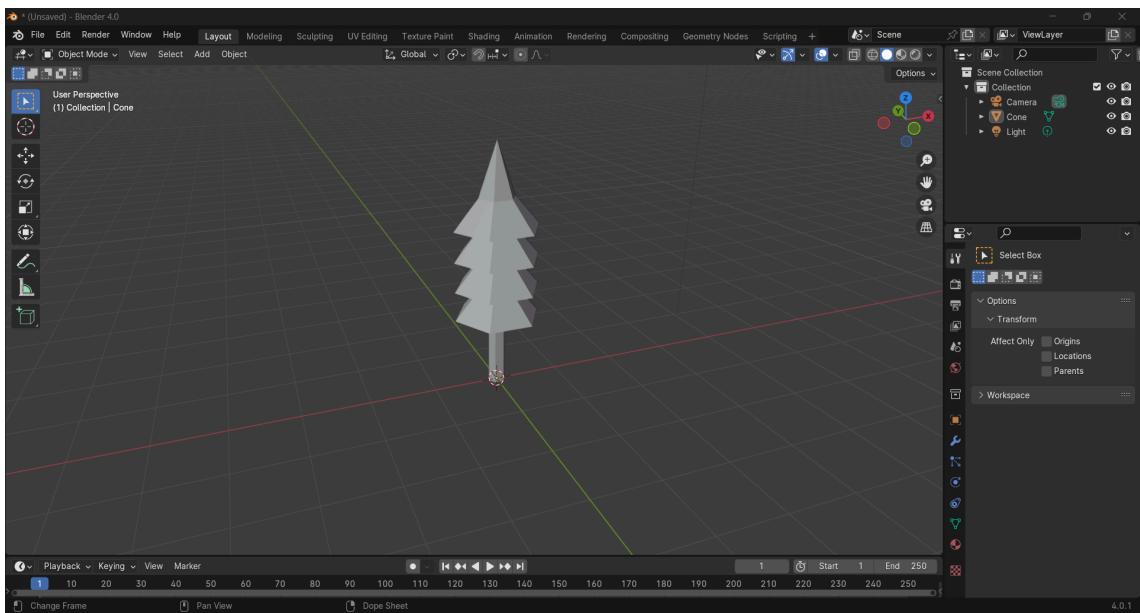
6.2. Stworzenie świata gry

6.2.1. Opis prac

W kolejnym etapie procesu tworzenia, do projektu dodano teren oraz drzewa, które wspólnie tworzą świat. Elementy te zostały zbudowane przy użyciu zaawansowanego programu graficznego Blender. Następnie obiekty te zainportowano za pomocą odpowiedniej funkcji napisanej w języku C++.



Rys. 6.2.1 - teren zaprojektowany w programie Blender 4.0



Rys. 6.2.2 - drzewo zaprojektowane w programie Blender 4.0

6.2.2. Omówienie kodu

```

std::pair<std::vector<float>, std::vector<float>> loadObjFile(const std::string& filename, GLuint textureName)
{
    std::vector<float> vertices;
    std::vector<float> textures;
    tinyobj::attrib_t attrib;
    std::vector<tinyobj::shape_t> shapes;
    std::vector<tinyobj::material_t> materials;
    std::string err;
    bool ret = tinyobj::LoadObj(&attrib, &shapes, &materials, &err, filename.c_str());
    GLuint textureID = textureName;

    for (const auto& shape : shapes) {
        for (const auto& index : shape.mesh.indices) {
            vertices.push_back(attrib.vertices[3 * index.vertex_index + 0]);
            vertices.push_back(attrib.vertices[3 * index.vertex_index + 1]);
            vertices.push_back(attrib.vertices[3 * index.vertex_index + 2]);

            if (!attrib.texcoords.empty()) {
                textures.push_back(attrib.texcoords[2 * index.texcoord_index + 0]);
                textures.push_back(attrib.texcoords[2 * index.texcoord_index + 1]);
            }
        }
    }
    return std::make_pair(vertices, textures);
}

```

Kod 6.2.1. - funkcja importująca model 3D do gry

Funkcja `loadObjFile` jest używana do wczytywania plików OBJ. Do jej odpowiedniego działania wymagane jest załączenie biblioteki `tinyobjloader`. Funkcja przyjmuje dwa argumenty: `filename`, który jest nazwą pliku OBJ do wczytania, i `textureName`, który jest identyfikatorem tekstuury. Na końcu zwraca parę wektorów: `vertices` i `textures`.

```

Void drawObjects(GLuint textureName, std::pair<std::vector<float>, std::vector<float>> vectors, GLfloat red, GLfloat green, GLfloat blue, float x, float y, float z, float scaleX, float scaleY, float scaleZ)
{
    GLuint textureID = textureName;
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, textureID);
    glBegin(GL_TRIANGLES);
    glColor3f(red, green, blue);
    int textureIndex = 0;
    for (size_t i = 0; i < vectors.first.size(); i += 3) {
        if (textureIndex < vectors.second.size()) {
            glTexCoord2f(vectors.second[textureIndex], vectors.second[textureIndex + 1]);
            textureIndex += 2;
        }
        glVertex3f(scaleX * vectors.first[i] + x, scaleY * vectors.first[i + 1] + y, scaleZ * vectors.first[i + 2] + z);
    }
    glEnd();
    glDisable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, 0);
}

```

Kod 6.2.2. - funkcja rysująca zimportowany model

Funkcja `drawObjects` rysuje zimportowane obiekty 3D w OpenGL. Przyjmuje identyfikator tekstury, wektory wierzchołków i tekstur, kolor RGB, położenie i skalę obiektu. Umożliwia ustawienie elementu w odpowiednim miejscu w przestrzeni.

Wizualizację świata gry po procesie importowania obiektów można zobaczyć na ilustracji oznaczonej jako *Rys. 6.3.1*, który znajduje się w kolejnym podrozdziale dokumentacji.

6.2.3. Podsumowanie

Integracja zewnętrznych obiektów do środowiska gry stworzonej w języku C++ jest skomplikowanym procesem. Obejmuje on szereg etapów, takich jak importowanie modeli 3D oraz innych zasobów, a następnie ich optymalizację i adaptację do specyfikacji gry. Wymaga to nie tylko biegłości w programowaniu C++, ale także znajomości specyficznych bibliotek i narzędzi używanych do manipulacji danymi graficznymi.

6.3. Teksturowanie obiektów i sterowanie kamerą

6.3.1. Opis prac

W tym etapie procesu zaimplementowano tekstury do wcześniej zimportowanych obiektów. Tekstury, zapisane w formacie .jpg, zostały nałożone na obiekty, aby poprawić wizualną jakość gry. Do realizacji tego zadania skorzystaliśmy z biblioteki STB_IMAGE. Dzięki jej elastyczności i wydajności, byliśmy w stanie skutecznie zintegrować tekstury z naszymi obiektami 3D, co znacznie wzbogaciło wizualne doświadczenie użytkownika.



Rys. 6.3.1. - świat gry po zimportowaniu i oteksturowaniu obiektów

Sterowanie kamerą jest realizowane tak, aby umożliwić dynamiczne śledzenie ruchu pojazdu. Zmiana pozycji jest obliczana w taki sposób, aby kamera zawsze utrzymywała pojazd w swoim polu widzenia, niezależnie od ruchu pojazdu. Współrzędne położenia są dynamicznie modyfikowane w odpowiedzi na interakcje użytkownika z klawiszami sterowania.

6.3.2. Omówienie kodu

```

GLuint loadTexture(const char* filename) {
    GLuint texture;
    glewInit();
    //glewExperimental = GL_TRUE;
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    int width, height, nrChannels;
    unsigned char* data = stbi_load(filename, &width, &height, &nrChannels, STBI_rgb);
    if (data) {
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);
    }
    stbi_image_free(data);
    return texture;
}

```

Kod 6.3.1. - wczytywanie tekstuury do projektu

Funkcja `loadTexture` jest używana do ładowania tekstur w OpenGL. Przyjmuje nazwę pliku jako argument, generuje teksturę, wiąże ją z określonym typem tekstury (2D), ustawia różne parametry tekstury, ładuje dane obrazu za pomocą biblioteki `stb_image`, a następnie, jeśli dane są poprawne, tworzy teksturę 2D i generuje mipmapę. Na koniec zwalnia załadowane dane obrazu i zwraca identyfikator tekstury.

Za nałożenie tekstury na obiekt odpowiada funkcja `loadObjFile` i `drawObjects`, opisane we wcześniejszych podrozdziałach.

```
void Game::loadAll()
{
    tekstyry[0] = loadTexture("textures/moutain1600.jpg");
    vectors.push_back(loadObjFile("obj/moutains3.obj", tekstyry[0]));

    tekstyry[1] = loadTexture("textures/grass2048.jpg");
    vectors.push_back(loadObjFile("obj/square.obj", tekstyry[1]));

    //TREES
    tekstyry[2] = loadTexture("textures/leaves.jpg");
    vectors.push_back(loadObjFile("obj/tree.obj", tekstyry[2]));

    tekstyry[3] = loadTexture("textures/leaves2.jpg");
    vectors.push_back(loadObjFile("obj/tree.obj", tekstyry[3]));

    tekstyry[4] = loadTexture("textures/leaves3.png");
    vectors.push_back(loadObjFile("obj/tree.obj", tekstyry[4]));

    //CARROTS
    tekstyry[5] = loadTexture("textures/orange.jpg");
    vectors.push_back(loadObjFile("obj/carrot_down.obj", tekstyry[5]));

    tekstyry[6] = loadTexture("textures/green.jpg");
    vectors.push_back(loadObjFile("obj/carrot_top.obj", tekstyry[6]));
}
```

Kod 6.3.2. - metoda importująca wszystkie modele 3D oraz tekstury

```
robot.x += speedRoverForward * cos(robot.rotationAngle * M_PI / 180.0f);
robot.y += speedRoverForward * sin(robot.rotationAngle * M_PI / 180.0f);
cameraPosY += speedRoverForward * sin(robot.rotationAngle * M_PI / 180.0f);
cameraPosX += speedRoverForward * cos(robot.rotationAngle * M_PI / 180.0f);
liczenie = 0;
```

Kod 6.3.3. - fragment wyliczający położenie pojazdu

Fragment kodu odpowiedzialny za aktualizację pozycji w grze. Współrzędne pojazdu oraz kamery są modyfikowane na podstawie prędkości i kąta obrotu. Dzięki temu, kamera zawsze podąża za obiektem.

```
gluLookAt(-10 + cameraPosX, cameraPosY, 7 + cameraPosZ, cameraPosX, cameraPosY, cameraPosZ, 0, 0, 1);
glRotatef(rotationAngle, 0.0f, 0.0f, 1.0f);
```

Kod 6.3.4.- fragment odpowiedzialny za położenie kamery

Kod manipuluje kamerą i orientacją sceny w OpenGL. `gluLookAt` ustawia pozycję kamery i kierunek, w którym patrzy. Pierwsze trzy argumenty określają pozycję kamery, następne trzy wskazują punkt, na który jest skierowana, a ostatnie trzy definiują kierunek “góry” kamery. `glRotatef` obraca scenę wokół osi Z o określony kąt.

6.3.3. Podsumowanie

Teksturowanie zaimportowanych obiektów 3D to skomplikowany proces, który wymaga zrozumienia i zastosowania specjalistycznych bibliotek programistycznych. Biblioteki, które są dodane do projektu jako zewnętrzne zależności, oferują narzędzia i funkcje niezbędne do manipulowania teksturami. Aby skutecznie wykorzystać te biblioteki, trzeba dokładnie zgłębić ich dokumentację i zrozumieć, jak działają poszczególne funkcje.

6.4. Implementacja sterowania

6.4.1. Opis prac

Do projektu zaimplementowano model jazdy, który symuluje dynamikę poruszania się pojazdu gąsienicowego. Model ten uwzględnia kluczowe zjawiska fizyczne, które wpływają na ruch pojazdu, takie jak przyśpieszenie, pęd i siła odśrodkowa. Sterowanie pojazdem jest intuicyjne i opiera się na standardowym zestawie przycisków W, A, S, D, co jest powszechnie stosowane w wielu grach. Przycisk ‘W’ służy do jazdy do przodu, ‘S’ do jazdy do tyłu, podczas gdy ‘A’ i ‘D’ służą do skręcania w lewo i prawo.



Rys. 6.4.1. - działanie siły odśrodkowej na pojazd

6.4.2. Omówienie kodu

```

if (isW) {
    tempX = robot.x + ((speedRoverForward + (max_speed_forward / 30)) * cos(robot.rotationAngle * M_PI / 180.0f));
    tempY = robot.y + ((speedRoverForward + (max_speed_forward / 30)) * sin(robot.rotationAngle * M_PI / 180.0f));
    collisions.setRobotX(tempX);
    collisions.setRobotY(tempY);
    collisions.setRobotZ(robot.z);
    radiansRA = robot.rotationAngle * M_PI / 180.0f;
    collisions.setRobotRA(radiansRA);
    if (!collisions.isCollisionDetected()) {
        if (speedRoverForward < max_speed_forward) {
            speedRoverForward += (max_speed_forward / 30);
        }
        robot.x += speedRoverForward * cos(robot.rotationAngle * M_PI / 180.0f);
        robot.y += speedRoverForward * sin(robot.rotationAngle * M_PI / 180.0f);
        cameraPosY += speedRoverForward * sin(robot.rotationAngle * M_PI / 180.0f);
        cameraPosX += speedRoverForward * cos(robot.rotationAngle * M_PI / 180.0f);
        liczenie = 0;
    }
    else {
        if (speedRoverForward != 0 || speedRoverBackward != 0) {
            if (liczenie == 0) {
                robot.inclinationDegree = 0.0f;
                speedRoverForward = -speedRoverForward / 2;
                speedRoverBackward = 0;
                addedTimeForCollision += 0.5;
                engineCollision->play2D("sounds/crash.mp3", false);
            }
            liczenie = 1;
        }
    }
}
}

```

Kod 6.4.1. - fragment implementacji sterowania pojazdu (jazda do przodu)

Jeśli przycisk ‘W’ jest naciśnięty (isW jest prawdziwe), kod oblicza tymczasowe współrzędne dla pojazdu, które reprezentują przewidywane położenie po uwzględnieniu aktualnej prędkości i kierunku. Obliczenia te uwzględniają również maksymalną prędkość do przodu.

```

if (isA) {
    tempX = robot.x - ((speedRoverBackward + (max_speed_backward / 30)) * cos(robot.rotationAngle * M_PI / 180.0f));
    tempY = robot.y - ((speedRoverBackward + (max_speed_backward / 30)) * sin(robot.rotationAngle * M_PI / 180.0f));
    collisions.setRobotX(tempX);
    collisions.setRobotY(tempY);
    collisions.setRobotZ(robot.z);
    if (!isS) {
        radiansRA = (robot.rotationAngle + turning_angle) * M_PI / 180.0f;
        if (speedRoverForward >= max_speed_forward && robot.inclinationDegree < 8.0f) {
            robot.inclinationDegree += 0.25f;
        }
        else if (robot.inclinationDegree > 0.0f) {
            robot.inclinationDegree -= 0.25f;
        }
    }
    else {
        radiansRA = (robot.rotationAngle - turning_angle) * M_PI / 180.0f;
    }
    collisions.setRobotRA(radiansRA);
    if (!collisions.isCollisionDetected()) {
        collisionA = false;
        robot.setRotate(isA, isD);
        if (!isS) {
            robot.rotationAngle += turning_angle;
        }
        else {
            robot.rotationAngle -= turning_angle;
        }
    }
    else {
        if (!collisionA) {
            robot.inclinationDegree = 0.0f;
            addedTimeForCollision += 0.5;
            engineCollision->play2D("sounds/crash.mp3", false);
            collisionA = true;
        }
    }
}
}

```

Kod 6.4.2. - fragment implementacji sterowania pojazdu (skręcanie w lewo)

Jeśli przycisk 'A' jest naciśnięty, kod oblicza tymczasowe współrzędne, które reprezentują przewidywane położenie obiektu po uwzględnieniu aktualnej prędkości i kierunku. Obliczenia te uwzględniają również maksymalną prędkość do tyłu. Jeśli przycisk 'S' nie jest naciśnięty, kąt obrotu jest zwiększany. Jeśli prędkość do przodu jest większa lub równa maksymalnej prędkości do przodu i kąt nachylenia jest mniejszy niż 8.0 stopni, kąt nachylenia jest zwiększany o 0.25 stopnia. W przeciwnym razie, jeśli kąt nachylenia pojazdu jest większy niż 0.0 stopni, kąt nachylenia jest zmniejszany o 0.25 stopnia.

6.4.3. Podsumowanie

Implementacja sterowania i symulacji zjawisk fizycznych w środowisku wirtualnym jest procesem, który w dużej mierze polega na manipulacji wartościami zmiennych i przeprowadzaniu odpowiednich obliczeń matematycznych. Zmienne te obejmują pozycję, prędkość, kierunek, rotację i wiele innych aspektów, które muszą być ciągle aktualizowane i dostosowywane w odpowiedzi na akcje użytkownika lub zdarzenia w grze.

6.5. Implementacja kolizji

6.5.1. Opis prac

Wykrywanie kolizji polega na ciągłym monitorowaniu i sprawdzaniu, czy punkty definiujące strukturę pojazdu nie znajdują się w obszarach kolizji otoczenia. Obszary kolizji dla elementów otoczenia są z góry zdefiniowane i reprezentują przestrzeń, w której inne obiekty nie mogą się znaleźć bez wywołania kolizji. Podczas rozgrywki, dla każdej klatki animacji, system sprawdza, czy jakiekolwiek z punktów pojazdu znajdują się w jednym z czterech obszarów kolizji elementu otoczenia. Jeśli tak, jest to interpretowane jako kolizja i podejmowane są odpowiednie działania - zatrzymanie pojazdu lub odbicie od obiektu (w zależności od prędkości pojazdu).



Rys. 6.5.1. - kolizja

6.5.2. Omówienie kodu

```
    treesNN.push_back({ -7, -16, -0.9 });
    treesNN.push_back({ -16, -26, -0.9 });

    linesPN.push_back({ {17.8, 4.5, -0.4}, {17.8, -1.15, -0.4} });
    linesPN.push_back({ {19.7, -1.15, -0.4}, {17.8, -1.15, -0.4} });
```

Kod 6.5.1. - definicja obszarów kolizji elementów otoczenia

Listy wypełnione są współrzędnymi elementów otoczenia (wektory `treesPN` itd.) oraz granic mapy (wektory `linesPN` itd.). Na wszystkich tych obiektach weryfikowane są kolizje.

```
bool Collisions::czyProstokatKolidujeZLinia(WspolrzedneRobot robot, float robotWidth, float robotHeight, Linia linia) {
    Wspolrzedne punkt1 = { robot.x - robotWidth / 2, robot.y - robotHeight / 2, -0.4 };
    Wspolrzedne punkt2 = { robot.x + robotWidth / 2, robot.y - robotHeight / 2, -0.4 };
    Wspolrzedne punkt3 = { robot.x - robotWidth / 2, robot.y + robotHeight / 2, -0.4 };
    Wspolrzedne punkt4 = { robot.x + robotWidth / 2, robot.y + robotHeight / 2, -0.4 };

    Linia krawedz1 = { punkt1, punkt2 };
    Linia krawedz2 = { punkt2, punkt4 };
    Linia krawedz3 = { punkt4, punkt3 };
    Linia krawedz4 = { punkt3, punkt1 };

    return czyLinieSiePrzecinaja(krawedz1, linia) ||
           czyLinieSiePrzecinaja(krawedz2, linia) ||
           czyLinieSiePrzecinaja(krawedz3, linia) ||
           czyLinieSiePrzecinaja(krawedz4, linia);
}
```

Kod 6.5.2. - funkcja wykrywająca kolizję z granicami mapy

Funkcja jest używana do sprawdzania, czy pojazd koliduje z granicą mapy (liniami rysowanymi wokół kształtu mapy). Jeżeli któraś z krawędzi pojazdu przecina się z daną linią, funkcja zwraca `true`, co oznacza, że koliduje z linią. W przeciwnym razie, funkcja zwraca `false`, co oznacza, że nie ma kolizji.

```
bool Collisions::checkingCollision(WspolrzedneRobot a, Wspolrzedne b) {
    // Oblicz współrzędne okręgu w układzie współrzędnych prostokąta
    float cosTheta = cos(-robot.ra);
    float sinTheta = sin(-robot.ra);
    float dx = b.x - a.x;
    float dy = b.y - a.y;
    float x = dx * cosTheta - dy * sinTheta;
    float y = dx * sinTheta + dy * cosTheta;

    // Znajdź najbliższy punkt na prostokącie do środka okręgu
    float najblizszyX = max(-robotWidth / 2, min(x, robotWidth / 2));
    float najblizszyY = max(-robotHeight / 2, min(y, robotHeight / 2));

    // Oblicz dystans między tym punktem a środkiem okręgu
    float dystansX = x - najblizszyX;
    float dystansY = y - najblizszyY;

    float promien = 0.15;

    if (b.z == -0.4) {
        promien = 0.15;
    }
    else if (b.z == -0.9) {
        promien = 0.25;
    }
    else if (b.z == 0.1) {
        promien = 0.1;
    }

    // Jeśli dystans jest mniejszy lub równy promieniowi okręgu, mamy kolizję
    return (dystansX * dystansX + dystansY * dystansY) <= (promien * promien);
}
```

Kod 6.5.3. - funkcja wykrywająca kolizję z elementami świata

Funkcja jest używana do sprawdzania, czy występuje kolizja między okręgiem (obszarem kolizji m.in. drzew) a pojazdem. Jeśli dystans jest mniejszy lub równy promieniowi okręgu, funkcja zwraca `true`, co oznacza, że występuje kolizja. W przeciwnym razie zwraca `false`, co oznacza brak kolizji.

6.5.3. Podsumowanie

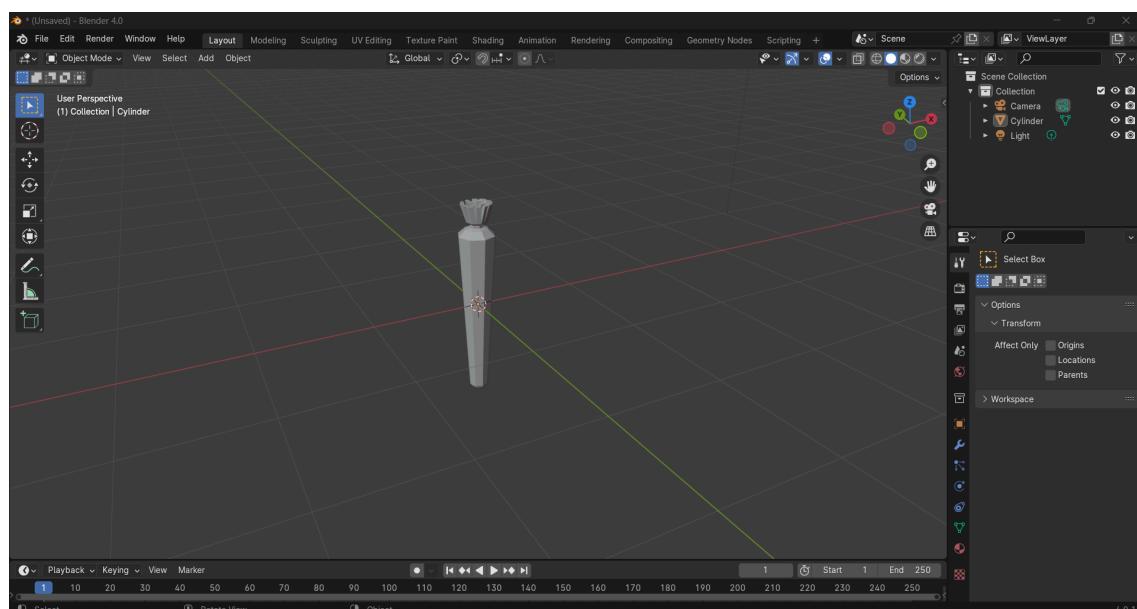
Wykrywanie kolizji to kluczowy element w grach komputerowych, który umożliwia interakcję między obiektami w środowisku wirtualnym. Elementy otoczenia, takie jak drzewa oraz granice mapy są reprezentowane przez zestawy punktów w przestrzeni 3D. System wykrywania kolizji sprawdza, czy współrzędne pojazdu pokrywają się z współrzędnymi któregokolwiek z tych elementów. Jeżeli tak, jest to interpretowane jako kolizja.

6.6. Opracowanie fabuły gry

6.6.1. Opis prac

Celem gry jest zebranie 20 marchewek, które są generowane w losowych miejscach na mapie. Gracz musi zebrać wszystkie marchewki w jak najkrótszym czasie, jednocześnie unikając kolizji z elementami otoczenia (każda kolizja nakłada karę czasową).

Aby zrealizować fabułę, do projektu został zainportowany trójwymiarowy model marchewki. Dodatkowo, zostały napisane specjalne funkcje, które losują położenie i wykrywają moment, w którym gracz zbiera marchewkę. Interfejs zaprojektowano tak, aby motywować gracza do ciągłego doskonalenia swoich umiejętności. Wyświetla on trzy najlepsze globalne wyniki (nie tylko z aktualnego uruchomienia gry), co stanowi dodatkową motywację do bicia własnych rekordów. Dzięki temu, istnieje możliwość śledzenia własnego postępu. Dodano także elementy audio, w tym muzykę i satysfakcjonujące dźwięki zbierania marchewek.



Rys. 6.6.1. - marchewka zaprojektowana w programie Blender 4.0



Rys. 6.6.2. - interfejs użytkownika

6.6.2. Omówienie kodu

```
void Game::randomCarrotsXYGenerate()
{
    std::random_device rd;
    std::mt19937 gen(rd());

    uniform_real_distribution<float> x_dist(-23.0, 28.0);
    uniform_real_distribution<float> y_dist(10.0, 36.0);

    do {
        float randX = x_dist(gen);
        float randY = y_dist(gen);
        randCarrotsXY.push_back({randX, randY});
        collisions.addCarrotsXY(randX, randY);
        licznik++;
    } while (licznik < 8);

    uniform_real_distribution<float> x_dist2(-20.0, 16.0);
    uniform_real_distribution<float> y_dist2(-6.0, 9.0);

    do {
        float randX2 = x_dist2(gen);
        float randY2 = y_dist2(gen);
        randCarrotsXY.push_back({randX2, randY2});
        collisions.addCarrotsXY(randX2, randY2);
        licznik2++;
    } while (licznik2 < 4);

    uniform_real_distribution<float> x_dist3(-21.0, 22.0);
    uniform_real_distribution<float> y_dist3(-33.0, -8.0);

    do {
        float randX3 = x_dist3(gen);
        float randY3 = y_dist3(gen);
        randCarrotsXY.push_back({randX3, randY3});
        collisions.addCarrotsXY(randX3, randY3);
        licznik3++;
    } while (licznik3 < 8);
}
```

Kod 6.6.1. - funkcja losująca położenie marchewek na mapie

Metoda jest odpowiedzialna za generowanie losowych pozycji marchewek. Mapa gry została podzielona na 3 obszary losowania, co zapewnia równomierne rozłożenie punktów, dzięki czemu marchewki są układane na całej mapie. Po wygenerowaniu każdej marchewki, metoda dodaje obszar kolizji dla niej do systemu wykrywania kolizji. Obszar ten jest używany do wykrywania, kiedy gracz zbiera marchewkę.

```
for (int i = 0; i < carrotsXY.size(); i++) {
    if (find(indexToDelete.begin(), indexToDelete.end(), i) == indexToDelete.end()) {
        if (checkingCollision(robot, carrotsXY[i])) {
            indexToDelete.push_back(i);
            punkty += 1;
            enginePoint->play2D("sounds/collect_point.mp3", false);
            return false;
        }
    }
}
```

Kod 6.6.2. - fragment kodu sprawdzający zebranie marchewki

Fragment kodu sprawdza, czy gracz zebrał marchewkę. Dla każdej marchewki na liście sprawdza, czy doszło do kolizji między nią a pojazdem. Jeśli tak, dodaje indeks marchewki do listy `indexToDelete`, zwiększa liczbę punktów o 1 i zwraca `false`.

```
void Game::DrawText(const char* text, GLfloat x, GLfloat y, GLfloat fontSize, int ID)
{
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D(0, 1000, 0, 1000);
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();

    glColor3f(0.0, 0.0, 0.0);
    glRasterPos2i(static_cast<int>(x + 5), static_cast<int>(y + 20));
    for (const char* c = text; *c != '\0'; ++c) {
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, *c);
    }

    ...
    glBegin(GL_QUADS);
    glVertex2i(static_cast<int>(x), static_cast<int>(y));
    glVertex2i(static_cast<int>(x + 200, static_cast<int>(y));
    glVertex2i(static_cast<int>(x + 200, static_cast<int>(y + 60));
    glVertex2i(static_cast<int>(x), static_cast<int>(y + 60));
    glEnd();

    glPopMatrix();
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
}
```

Kod 6.6.3. - fragment kodu odpowiedzialny za stworzenie pola tekstowego z wiadomością

Funkcja `DrawText` odpowiada za stworzenie na ekranie tła w odpowiednim kolorze, wraz z tekstem podanym jako parametr. Określa ona także miejsce, w którym dana wiadomość ma się pojawić, a także rozmiar czcionki, którą ma być wypisywana.

```

void Game::drawTextObjects()
{
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D(0, 1000, 0, 1000);
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();
    glColor3f(1.0, 1.0, 1.0);
    ...
    char gameTime[40];
    char collectedCarrots[30];

    ...
    sprintf_s(gameTime, "Czas gry: %0.2f s + %0.1f s (kolizje)", _time_span.count(), addedTimeForCollision)
    sprintf_s(collectedCarrots, "Zebrańo %d/20 marchewek", collisions.getCollectedCarrots());
    ...

    if (gameStarted) {
        DrawText(gameTime, 20, 80, 20, 1);
        DrawText(collectedCarrots, 20, 150, 20, 1);
        DrawText(robotSpeed, 920, 80, 20, 2);
        DrawText(top3Table, 20, 920, 20, 1);
        DrawText(top1, 20, 860, 20, 4);
        DrawText(top2, 20, 800, 20, 5);
        DrawText(top3, 20, 740, 20, 6);
        DrawText(controls, 800, 920, 20, 1);
        DrawText(zoomSize, 20, 220, 20, 1);
        if (tab) {
            DrawText(wsad, 800, 860, 20, 8);
            DrawText(zoom, 800, 800, 20, 9);
            DrawText(pauza, 800, 740, 20, 8);
            DrawText(backspace, 800, 680, 20, 9);
        }
        if (stop) {
            DrawText(stopped, 407, 530, 20, 7);
        }
    }
    else {
        DrawText(gameBeginning, 407, 530, 20, 7);
        if (lastResult > 0.00) {
            DrawText(showLastResult, 407, 450, 20, 3);
        }
    }
    ...
    glPopMatrix();
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
}

```

Kod 6.6.4. - fragment kodu odpowiadający za wyświetlenie konkretnych komunikatów na ekranie w odpowiednim momencie

Funkcja `drawTextObjects` ma za zadanie określić jakie wiadomości mają być dokładnie wyświetlane na ekranie (m.in. interfejs użytkownika) z użyciem funkcji `DrawText`. Ustala ona, które teksty mają pojawić się przed rozpoczęciem gry, które po, a które podczas pauzy.

```
ISoundEngine* engine = createIrrKlangDevice();

if (!engine) {
    return 0;
}

engine->play2D("sounds/farming.mp3", true);
```

Kod 6.6.5. - fragment kodu odpowiadający za odtwarzanie muzyki w tle trwania gry

W projekcie zainportowana jest biblioteka irrKlang, która odpowiada za odtwarzanie muzyki w tle gry, a także imitację dźwięków kolizji pojazdu z różnymi przeszkodami. Odpowiedni sygnał usłyszmy także zbierając kolejne marchewki.

6.6.3. Podsumowanie

Gra łączy w sobie elementy zręczności i strategii, oferując graczom wyzwanie polegające na zebraniu wszystkich marchewek w jak najkrótszym czasie, jednocześnie unikając kolizji z elementami otoczenia. Zaawansowany interfejs gry i system zapisywania wyników dodatkowo i satysfakcjonujące udźwiękowienie zwiększą zaangażowanie graczy, motywując ich do ciągłego doskonalenia swoich umiejętności.

7. Podsumowanie

“Tractor Rush: Carrots Edition” to gra 3D, która została zaprojektowana i zaimplementowana w języku C++ z wykorzystaniem biblioteki OpenGL. Projekt ten skupiał się na realizacji kluczowych mechanizmów fizycznych, takich jak sterowanie pojazdem, detekcja kolizji oraz dynamika ruchu. Dodatkowo, gra zawiera tekstury obiektów, możliwość manipulacji kamerą oraz elementy fabuły.

Celem gry jest sterowanie pojazdem rolniczym w celu zebrania marchewek umieszczonych w trudno dostępnych miejscach, jednocześnie unikając kolizji z elementami otoczenia w jak najkrótszym czasie. Każda kolizja skutkuje nałożeniem kary czasowej, a najlepsze wyniki są rejestrowane na tablicy wyników.

Projekt był realizowany etapami, począwszy od stworzenia modelu pojazdu, a następnie dodawania otoczenia za pomocą plików .obj z Blendera. Teksturowanie obiektów i sterowanie kamerą zostały dodane w późniejszych etapach.

Model ruchu pojazdu uwzględnia stopniowe przyspieszanie, hamowanie, skręcanie, a także takie podstawowe zjawiska fizyczne jak pęd i siła odśrodkowa. Mechanizm ruchu został zintegrowany z systemem detekcji kolizji.

W rezultacie powstała ciekawa gra zręcznościowa idealna na nudę.