NightMower

Projekt gry 3D w języku C++ z wykorzystaniem OpenGL

Dokumentacja projektu

2EF-DI 2023/24, L01

Marcin Bator 173592

Bogdan Bosak 173597

Politechnika Rzeszowska,

Grafika Komputerowa 2023/24

Spis treści

1. Założenia projektu

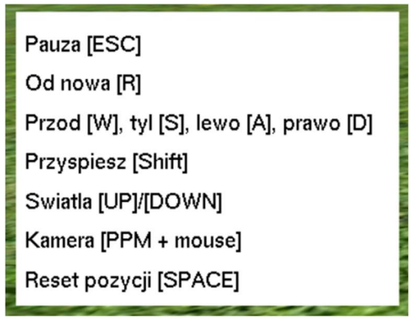
Celem projektu było stworzenie gry 3D na platformę Windows w języku C++ z wykorzystaniem OpenGL. W grze miały zostać zaimplementowane podstawowe zjawiska fizyczne, takie jak sterowanie pojazdem, kolizje z obiektami czy pęd poruszającego się ciała. Wymagane było także teksturowanie obiektów, poruszanie kamerą oraz dodanie prostej fabuły. Wszystkie założenia projektu wynikające z instrukcji do laboratorium zostały zrealizowane i poszerzone o dodatkowe funkcjonalności.

2. Opis projektu

Projekt to gra komputerowa o nazwie *NightMower* pozwalająca użytkownikowi na prowadzenie kosiarki do trawy. Celem jest zebranie wszystkich ziemniaków pojawiających się na trawie w losowych miejscach. Po zebraniu wszystkich zwiększa się poziom oraz losują się nowe ziemniaki. Za zebranie ziemniaka gracz dostaje 30 punktów. W zbieraniu przeszkadzają drzewa oraz mur pośrodku: za uderzenie w którąś z przeszkód lub w góry na krawędziach mapy gra zabiera 50 punktów. Najlepszy wynik jest zapisywany do pliku i wyświetlony na ekranie.

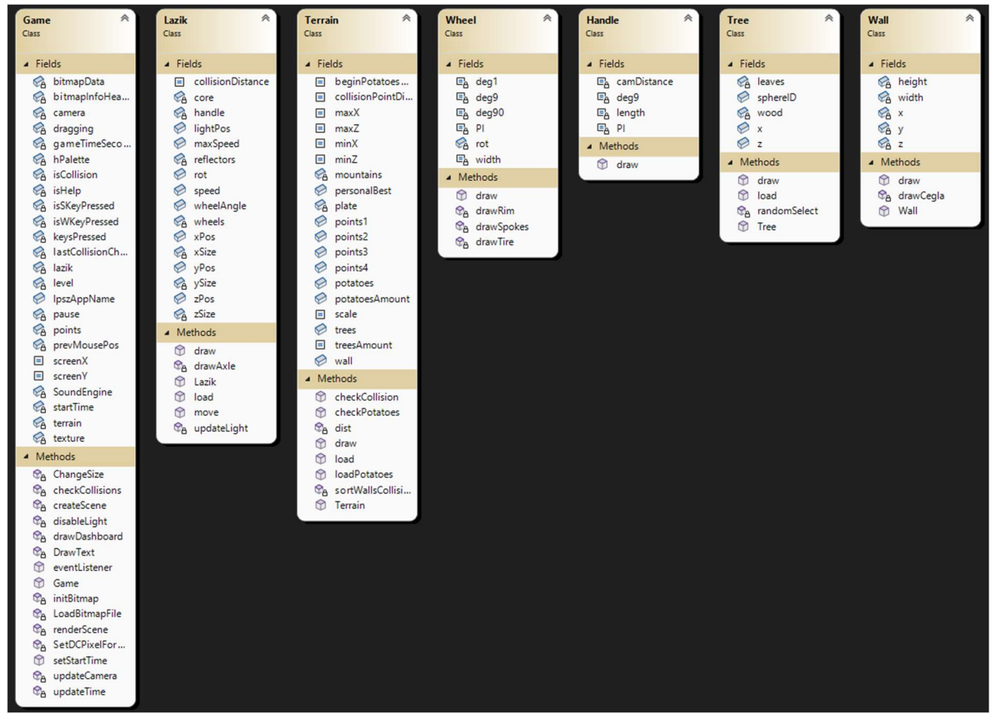
Rys. 2.1. – zrzut ekranu z gry

Gracz ma możliwość sterowania kosiarką za pomocą klawiszy. Może on też zwiększać jej prędkość, sterować wysokością świateł, zatrzymać lub zrestartować grę oraz wrócić do początkowej pozycji. Sterowanie kamerą jest możliwe w dwóch płaszczyznach: odległości od kosiarki (w określonych granicach) oraz azymutu.

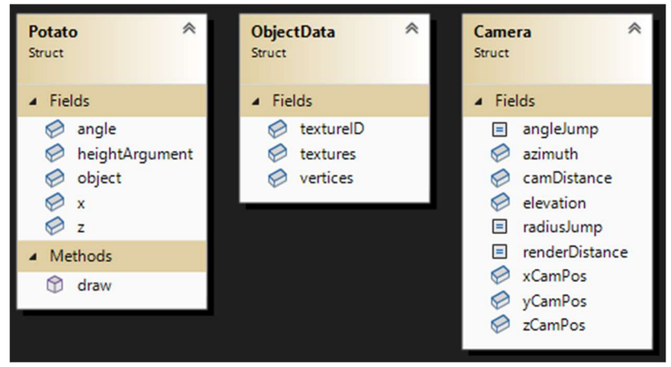


Rys. 2.2. – okno sterowania

Projekt został napisany w przeważającej części w paradygmacie obiektowym, na tyle na ile pozwalały nieco przestarzałe pryncypia OpenGL w wersji udostępnionej w projekcie *szescian*. Z tego powodu zdecydowano się na niezbyt rygorystyczne podejście co do zasad hermetyzacji, która i tak nie do końca może być zachowana biorąc pod uwagę nieobiektową konstrukcję biblioteki OpenGL. Luźne podejście do wyżej wymienionych zasad pozwoliło na przyspieszenie prac nad projektem i ułatwienie procesu debugowania. Na całość projektu składa się 7 klas oraz 3 struktur, których schemat widać poniżej:



Rys. 2.3. – schemat klas



Rys. 2.4. – schemat struktur

Szczegółowe omówienie każdej z klas wraz z zastosowaniem znajduje się w dalszej części dokumentacji.

3. Zasady gry

Kosiarka zawsze jest tworzona w punkcie 0,0 na poziomej osi współrzędnych. Granice mapy oraz mur są zawsze położone w tym samym miejscu, zaś położenie drzew jest losowane po uruchomieniu programu.

Położenie ziemniaków jest każdorazowo losowane na początku programu oraz po przejściu na kolejny poziom. Ich liczba początkowo wynosi 5 i jest zwiększana na każdym z poziomów. Liczba poziomów jest nieograniczona. Za zebranie ziemniaka gracz otrzymuje 30 punktów, zaś za kolizję z obiektem zabiera się mu 50 punktów. Najlepszy wynik jest zapisywany do pliku .txt.

4. Instrukcja gry

Podczas gry użytkownik ma wgląd w liczbę punktów, rekord punktów, czas rozgrywki oraz poziom. Po naciśnięciu przycisku H na klawiaturze ukazuje się okno z opisem wszystkich przycisków. Gracz może sterować kosiarką za pomocą przycisków W, A, S, D, a także zwiększyć jej prędkość maksymalną przytrzymując Shift.

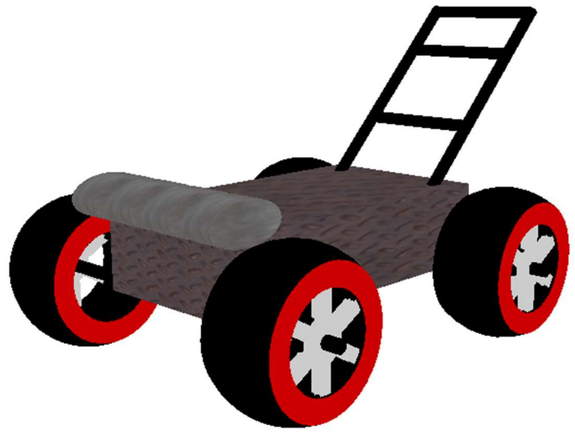
Sterowanie kamery odbywa się za pomocą myszy. Po przytrzymaniu jej prawego przycisku i poruszaniu zmienia się azymut kamery, a za pomocą suwaka można regulować odległość kamery od kosiarki. Wysokość świateł zmienia się za pomocą strzałek Up/Down. Gracz może zatrzymać rozgrywkę za pomocą przycisku ESC oraz zrestartować ją wciskając R. Pozycja łazika w razie zablokowania może być przywrócona na pole początkowe za pomocą spacji.

5. Szczegółowy opis etapów

5.1. Stworzenie obiektu kosiarki (laboratorium III)

5.1.1. Opis prac

Pierwszym etapem była zmiana implementacji programu *sześcian* napisanego w języku C na projekt obiektowy w języku C++, a także utworzenie obiektu kosiarki składającej się z wielu brył. Kosiarka zbudowana w projekcie może być podzielona na bryły napisane bezpośrednio w języku C++ oraz na bryły zaimportowane w formacie .obj z programu graficznego Blender. Do tych pierwszych zaliczają się: koła, felgi, osie kół oraz rączka kosiarki. Zaimportowane i oteksturowane zostały przednie czujniki/reflektory oraz główny człon łazika.



Rys. 5.1. – kosiarka

Każde koło kosiarki składa się z 8 brył elementarnych: opony, sześciu przęseł felgi oraz okrągłego elementu na środku felgi. Pojazd posiada 4 reflektory (spłaszczone walce), dwie osie na których zamocowane są koła, zaś jego rączka składa się z pięciu walców. Podsumowując, na bryłę kosiarki składa się 39 brył napisanych ręcznie w OpenGL oraz 5 brył zaimportowanych z pliku .obj – łącznie 44 bryły.

Kod użyty do importowania tekstur i obiektów został omówiony w części poświęconej budowie otoczenia (rozdział 5.2.).

5.1.2. Omówienie kodu

Za obsługę kosiarki odpowiedzialna jest klasa Lazik. Posiada ona dane dotyczące obiektu i tekstury, pozycji, rozmiarów pojazdu jak i jest odpowiedzialna za jego ruch oraz rysowanie świateł.

Lazik.h

lazik (Global Scope)

```h
#pragma once
#include "loader.h"
#include <Wheel.h>
#include <Handle.h>
#include <Camera.h>
#include <set>

#define GL_PI 3.14159265359

using namespace std;

class Lazik
{
    int xSize;
    int ySize;
    int zSize;
    Handle handle;
    ObjectData core;
    Wheel wheels[4];
    ObjectData reflectors[4];
    void updateLight();
    void drawAxle(int x, int y, int z);
public:
    GLfloat xPos = 0;
    GLfloat yPos = 0;
    GLfloat zPos = 0;
    GLfloat rot = 0;
    float speed = 0;
    float maxSpeed = 4;
    float wheelAngle = 0;
    double lightPos = GL_PI / 12 * 0.3;
    static const int collisionDistance = 25;
    Lazik(int xSize, int ySize, int zSize);
    void move(bool pause, bool& isWKeyPressed, bool& isSKeyPressed, Camera* camera, set<int>& keysPressed);
    void load();
    void draw();
};

```

List. 5.1. – klasa Lazik

Pierwsze trzy pola składowe określają rozmiary kosiarki w trzech płaszczyznach. Następnie znajduje się obiekt klasy Handle (rączka kosiarki – rozdział 5.1.2.3.) oraz 4-elementowa tablica obiektów typu Wheel (koło łazika – rozdział 5.1.2.2.). Pola typu ObjectData – *core* (główny element kosiarki) oraz tablica *reflectors* (reflektory).

Pola publiczne zaczynają się od określenia aktualnej pozycji kosiarki a także kąta jej obrotu, prędkości, prędkości maksymalnej, kąta nachylenia kół oraz aktualnej pozycji światła. Stała *collisionDistance* wyraża promień okręgu o środku w punkcie środkowym łazika jaki jest wykorzystywany do wykrywania kolizji.

Konstruktor ustawia podstawowe parametry obiektu, takie jak rozmiar i pozycję.

Metoda *load* jest dość podobna dla każdej z klas zawierającej obiekty z teksturami, dlatego że odpowiada ona za wywołanie metody *loadFile* z pliku *loader.cpp*, który zostanie omówiony w dalszej części. Z tego powodu zostanie ona zaprezentowana jedynie na przykładzie kosiarki.

```h
void Lazik::load()
{
    core = loadFile("core.obj", "floor.jpg");
    for (int i = 0; i < 4; i++) {
        reflectors[i] = loadFile("reflector.obj", "metal.jpg");
    }
}
```

List. 5.2. – funkcja ładowania obiektów dla kosiarki

Metoda rysowania kosiarki opiera się na ustawienie środka układu współrzędnych na miejsce przebywania łazika za pomocą funkcji *glPushMatrix*, ustawienie rotacji oraz narysowanie poszczególnych elementów pojazdu.

```
21  void Lazik::draw()
22  {
23      updateLight();
24      glPushMatrix();
25      glTranslatef(xPos, yPos, zPos);
26      glRotatef(rot, 0.0f, 1.0f, 0.0f);
27      glPolygonMode(GL_BACK, GL_LINE);
28      drawObj(&core, 0,  5, 0, xSize * 0.5, zSize, ySize * 0.78, 2, 2);
29      handle.draw(-this->xSize + 10, - 10 + 3 * this->ySize, 0, this->zSize);
30      drawAxle(4 + this->xSize / 2, 0, 0);
31      drawAxle(- 4 - this->xSize / 2, 0, 0);
32      wheels[0].draw(  4 + this->xSize / 2, 0,  - this->ySize - 12, 0, speed, wheelAngle);
33      wheels[1].draw(  4 + this->xSize / 2, 0,  this->ySize + 12, 1, speed, wheelAngle);
34      wheels[2].draw( - 4 - this->xSize / 2, 0,  - this->ySize - 12, 0, speed, 0);
35      wheels[3].draw( - 4 - this->xSize / 2, 0,  this->ySize + 12, 1, speed, 0);
36      for (int z = -2; z < 2; z++) {
37          drawObj(&reflectors[z + 2],  5 + xSize / 2,  3 + ySize, z * 8 + 4, xSize * 0.8, zSize * 0.8, ySize * 0.8, 2, 2);
38      }
39      glPopMatrix();
40  }
```

List. 5.3. – funkcja rysowania pojazdu

Jak wspomniano wyżej, do rysowania łazika użyto wielu walców w różnych wymiarach. Przykładem jest metoda *drawAxle* rysująca walec będący osią kół pojazdu. Po ustawieniu układu współrzędnych w odpowiednie miejsce, funkcja używa rysowania za pomocą trójkątów *GL_TRIANGLE_STRIP*.

```
153 void Lazik::drawAxle(int x, int y, int z)
154 {
155     GLfloat PI = 3.14159;
156     GLfloat deg9 = PI / 20;
157     GLfloat camDistance = 1;
158     GLfloat length = 15;
159
160     glPushMatrix();
161     glTranslatef(x, y, z);
162
163     glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
164     {
165         glColor3f(0, 0, 0);
166         for (int i = -length; i < length; i++) {
167             glBegin(GL_TRIANGLE_STRIP);
168             for (int j = 0; j < 41; j++) {
169                 GLfloat x = 2 * i;
170                 GLfloat y = camDistance * cos(deg9 * j);
171                 GLfloat z = camDistance * sin(deg9 * j);
172                 glVertex3d(z, y, x);
173                 x = 2 * (i + 1);
174                 glVertex3d(z, y, x);
175             }
176             glEnd();
177         }
178         for (int k = 0; k < 2; k++) {
179             int capDist = (k ? -2 : 2) * length;
180             glBegin(GL_TRIANGLE_FAN);
181             for (int j = 0; j < 41; j++) {
182                 GLfloat x = cos(deg9 * j);
183                 GLfloat y = sin(deg9 * j);
184                 GLfloat z = capDist;
185                 glVertex3d(x, y, z);
186             }
187             glEnd();
188         }
189     }
190     glPopMatrix();
191 }
```

List. 5.4. – funkcja rysowania osi kół pojazdu

Bardzo podobny do wyżej wymienionego sposób rysowania walców posiada klasa *Handle*, której jedyna metoda *draw* służy do narysowania pięciu walców w różnych rozmiarach i pod różnymi kątami, które składają się w całość jako uchwyt kosiarki.

Zdecydowanie najciekawszą i najbardziej zaawansowaną klasą wykonaną w pierwszym etapie budowania gry jest klasa *Wheel*. Ze względu na skomplikowaną strukturę bryły koła, narysowanie go zostało podzielone na kilka metod:

```
1   #pragma once
2   #ifdef _MSC_VER
3   #  pragma comment(lib, "opengl32.lib")
4   #  pragma comment(lib, "glu32.lib")
5   #endif
6   #include <windows.h>
7   #include <gl\gl.h>
8   #include <gl\glu.h>
9
10  class Wheel
11  {
12      int rot = 0;
13      const GLfloat PI = 3.14159;
14      const GLfloat deg90 = PI / 2;
15      const GLfloat deg9 = PI / 20;
16      const GLfloat deg1 = deg90 / 90;
17      const int width = 6;
18      void drawTire();
19      void drawSpokes();
20      void drawRim(bool flip, GLfloat someDegree);
21  public:
22      void draw(int xSize, int ySize, int zSize, bool mirror = 0, GLfloat rotation = 0, float angle=0);
23  };
24
```

List. 5.5. – klasa Wheel

Publiczna funkcja *draw* klasy umożliwia ustawienie strony po której ma znajdować się felga (*mirror)*, rotacji koła względem swojej osi (*rotation*), a także kąta wychylenia koła podczas skrętu (*angle*).

```
5   void Wheel::draw(int x, int y, int z, bool mirror, GLfloat rotation, float angle)
6   {
7       glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
8       {
9           glPushMatrix();
10          glTranslatef(x, y, z);
11          glRotatef(90, 0, 1.0, 0);
12          glRotatef(angle, 0, 1, 0);
13
14          drawTire();
15          rot += rotation * random(90, 180);
16          drawRim(mirror, rot);
17
18          glPopMatrix();
19      }
20  }
```

List. 5.6. – metoda draw

Aby uwidocznić ciągłość obracania się koła pojazdu, rotacja jest ustawiana jako losowa liczba z zakresu od 90 do 180 stopni pomnożona razy parametr *rotation*, przez który tak naprawdę przekazywana jest prędkość pojazdu będąca jednocześnie mnożnikiem szybkości obrotu koła. Dzięki temu szybkość obrotu koła rośnie wprost proporcjonalnie do szybkości pojazdu.

Najbardziej złożoną metodą jest metoda *drawTire*.

```
void Wheel::drawTire() {
    GLfloat bend = 10 * deg1;
    int stripes = width / 2;   // num of stripes in a half a tire
    int outRds = 12; // outer radius
    int innRds = 7;  // inner radius

    //-----outer part of a tire-----
    glColor3f(0, 0, 0);
    for (int i = -stripes; i < stripes; i++) {
        glBegin(GL_TRIANGLE_STRIP);
        GLfloat a1 = cos(bend * i);
        GLfloat a2 = cos(bend * (i + 1));
        for (int j = 0; j < 41; j++) {
            GLfloat b = deg9 * j;

            GLfloat x = 2 * i;
            GLfloat y = outRds * a1 * cos(b);
            GLfloat z = outRds * a1 * sin(b);
            glVertex3d(x, y, z);
            x = 2 * (i + 1);
            y = outRds * a2 * cos(b);
            z = outRds * a2 * sin(b);
            glVertex3d(x, y, z);
        }
        glEnd();
    }

    //-----inner part of a tire-----
    for (int i = -stripes; i < stripes; i++) {
        glBegin(GL_TRIANGLE_STRIP);
        for (int j = 0; j < 41; j++) {

            GLfloat x = 2 * i;
            GLfloat y = innRds * cos(deg9 * j);
            GLfloat z = innRds * sin(deg9 * j);
            glVertex3d(x, y, z);
            x = 2 * (i + 1);
            glVertex3d(x, y, z);
        }
        glEnd();
    }

    //-----side of a tire-----
    glColor3f(0.8, 0, 0);
    for (int k = 0;k < 2;k++) {
        glBegin(GL_TRIANGLE_STRIP);
        for (int i = 0; i < 41; i++) {
            GLfloat a = deg9 * i;

            GLfloat x = k ? width : -width;
            GLfloat y = innRds * sin(a);
            GLfloat z = innRds * cos(a);
            glVertex3d(x, y, z);
            y = outRds * sin(a) * cos(bend * stripes);
            z = outRds * cos(a) * cos(bend * stripes);
            glVertex3d(x, y, z);
        }
        glEnd();
```

List. 5.7. – metoda drawTire

Metoda również korzysta z funkcji rysowania za pomocą trójkątów *GL_TRIANGLE_STRIP*. W pierwszej jej części za pomocą odpowiedniego zmniejszania kąta powierzchni opony względem felgi podczas zbliżania się do jej środka określonego przez pomocniczą zmienną *bend* i

zastosowanie wartości trygonometrycznej na trzeciej osi według wartości *a1* i *a2* udało się uzyskać oponę bardzo zbliżoną do tej z bolidu Formuły 1. Ostatnia część kodu natomiast łączy zewnętrzną część opony z wewnętrzną w punktach określonych przez maksymalną wartość odchylenia.

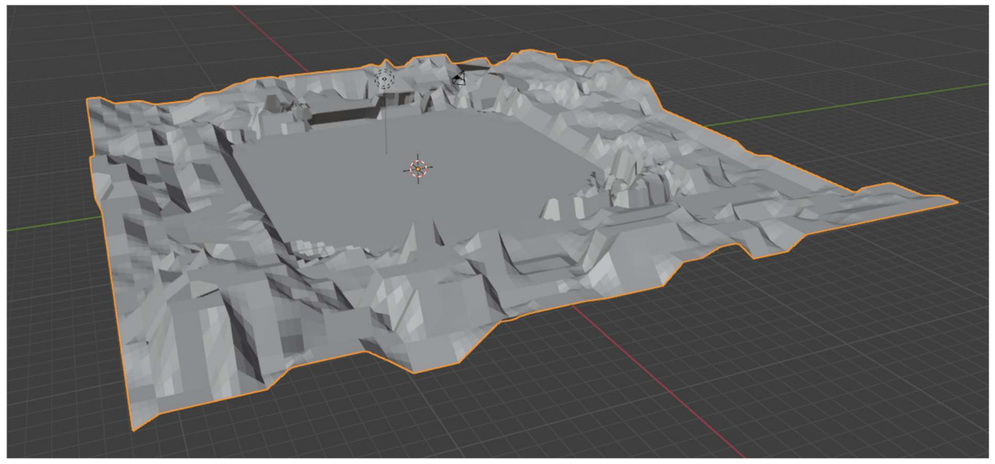
5.1.3. Podsumowanie

Prace wykonane w pierwszym etapie nie okazały się szczególnie łatwe ze względu na brak wiedzy na temat ładowania plików z formatu .obj do programu. Wszystkie bryły elementarne były pisane ręcznie za pomocą trójkątów. Elementy załadowane do łazika dodane zostały dopiero w późniejszym etapie, który skupił się w większości na opracowaniu pół-autorskiego sposobu importu plików .obj oraz .jpg.

5.2. Stworzenie otoczenia i opracowanie ładowania pliku .obj (laboratorium IV)

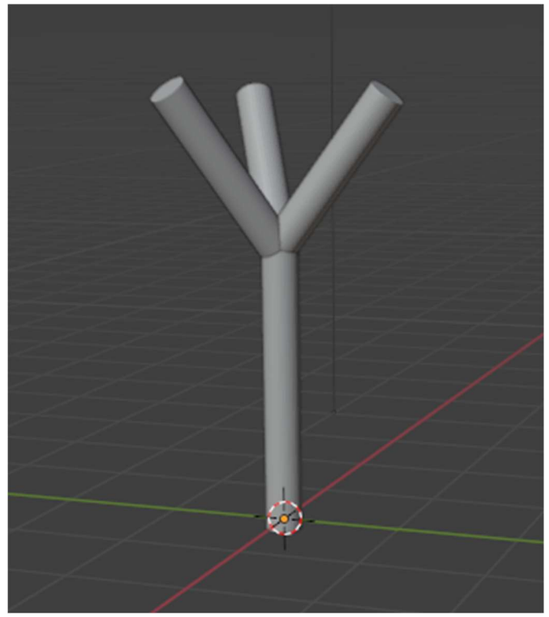
5.2.1. Opis prac

Na tym etapie założeniem było zaprojektowanie kształtu otoczenia w programie Blender, a następnie użycie go w grze. Zdecydowano się na użycie formatu .obj, który przechowuje informacje o współrzędnych wierzchołków trójkątów służących do rysowania obiektu.

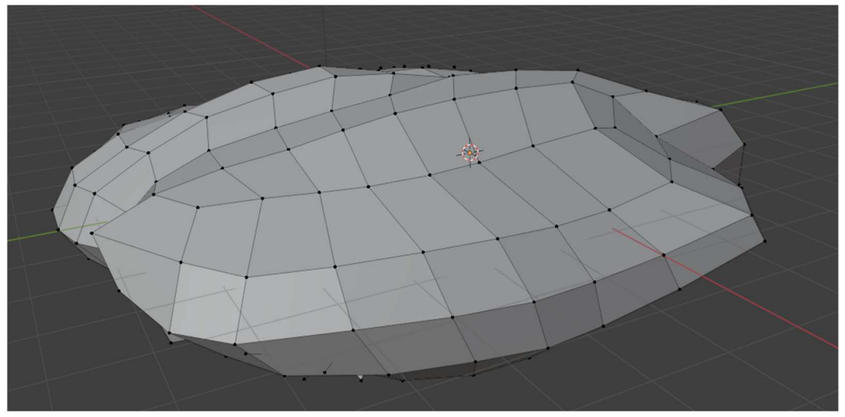


Rys. 5.2 – projekt otoczenia w programie Blender

Wykonanie otoczenia w programie graficznym polegało na podzieleniu obiektu na części za pomocą funkcji *subdivide*, a następnie ukształtowaniu ich za pomocą zmiany położenia zaznaczonych punktów. Następnie należało wyeksportować obiekt do formatu .obj. Podobnie postąpiono przy tworzeniu obiektów pnia oraz korony drzewa a także ziemniaka.



Rys. 5.3. – obiekt pnia drzewa



Rys. 5.4. – obiekt korony drzewa

5.2.2. Omówienie kodu

Po wyeksportowaniu obiektów, największym wyzwaniem okazało się opracowanie metody do ładowania ich do programu. Z pomocą przyszła biblioteka *TinyObjLoader*, która pozwala parsować zawartość pliku .obj do wektorów wierzchołków oraz tekstur. Link do biblioteki:

https://github.com/tinyobjloader/tinyobjloader

Bez wątpienia jednym z najbardziej wymagających etapów było opracowanie funkcji, która pozwoliłaby na załadowanie obiektu, a w kolejnym kroku tekstury do tego obiektu. Pierwotna implementacja była bardzo nieoptymalna, bowiem wywoływana podczas rysowania sceny w każdej klatce gry ładowała za każdym razem obiekt i teksturę z pliku, co powodowało ogromny

spadek wydajności programu. Dopiero jej zaktualizowanie przy okazji pracy nad fabułą gry przyniosło skutek w postaci znacznego wzrostu klatkażu. Aby było to możliwe, została wprowadzona struktura *ObjectData*, która przechowuje dane o załadowanych teksturze oraz obiekcie.

```
8   struct ObjectData {
9       std::vector<float> vertices;
10      std::vector<float> textures;
11      GLuint textureID;
12  };
13
```

List. 5.8. – struktura ObjectData

Dzięki opracowaniu tej struktury możliwe stało się załadowanie obiektu oraz tekstury tylko na początku gry (funkcja *loadFile*), a następnie rysowanie ich w każdej klatce (funkcja *drawObj*).

```
26  ObjectData loadFile(const std::string& filename, const std::string& textureName) {
27      ObjectData objData;
28      tinyobj::attrib_t attrib;
29      std::vector<tinyobj::shape_t> shapes;
30      std::vector<tinyobj::material_t> materials;
31      std::string err;
32      std::string objFilePath = "objects/" + filename;
33      bool ret = tinyobj::LoadObj(&attrib, &shapes, &materials, &err, objFilePath.c_str());
34      std::string textureFilePath = "textures/" + textureName;
35      objData.textureID = loadTexture(textureFilePath.c_str());
36      for (const auto& shape : shapes) {
37          for (const auto& index : shape.mesh.indices) {
38              objData.vertices.push_back(attrib.vertices[3 * index.vertex_index + 0]);
39              objData.vertices.push_back(attrib.vertices[3 * index.vertex_index + 1]);
40              objData.vertices.push_back(attrib.vertices[3 * index.vertex_index + 2]);
41
42              if (!attrib.texcoords.empty()) {
43                  objData.textures.push_back(attrib.texcoords[2 * index.texcoord_index + 0]);
44                  objData.textures.push_back(attrib.texcoords[2 * index.texcoord_index + 1]);
45              }
46          }
47      }
48
49      return objData;
50  }
```

List. 5.9. – funkcja loadFile

Funkcja przyjmuje parametry będące nazwą pliku .obj oraz pliku z teksturą. Następnie ładuje je do struktury z wykorzystaniem funkcjonalności biblioteki *TinyObjLoader*, na końcu zaś zwraca strukturę z odpowiednio ustawionymi polami składowymi.

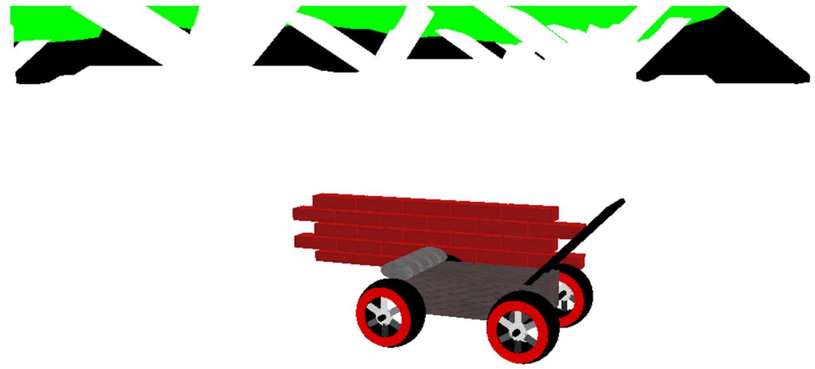
```
52  void drawObj(ObjectData* objectData, int x, int y, int z, int scaleX, int scaleY, int scaleZ, int repeatX, int repeatY, int r, int g, int b) {
53      glEnable(GL_TEXTURE_2D);
54      glBindTexture(GL_TEXTURE_2D, objectData->textureID);
55      glBegin(GL_TRIANGLES);
56      int textureIndex = 0;
57      glColor3f(r, g, b);
58      for (size_t i = 0; i < objectData->vertices.size(); i += 3) {
59          if (textureIndex < objectData->textures.size()) {
60              glTexCoord2f(repeatX * objectData->textures[textureIndex], repeatY * objectData->textures[textureIndex + 1]);
61              textureIndex += 2;
62          }
63          glVertex3f(scaleX * objectData->vertices[i] + x, scaleY * objectData->vertices[i + 1] + y, scaleZ * objectData->vertices[i + 2] + z);
64      }
65
66      glEnd();
67      glDisable(GL_TEXTURE_2D);
68  }
```

List. 5.10. – funkcja drawObj

Powyższa funkcja wywoływana jest w każdej klatce gry. Przyjmuje wskaźnik do struktury *ObjectData* z danymi obiektu, jego pozycję, rozmiar, ilość powtarzania tekstury w wymiarze X i Y,

a także wartości RGB w razie chęci nadania dodatkowo koloru obiektowi. Jest to tak naprawdę zwykłe ręczne rysowanie trójkątów o wierzchołkach załadowanych uprzednio z pliku .obj. Wykorzystanie metody *drawObj* do narysowania elementów łazika zostało przedstawione na listingu 5.2. Funkcja *loadTexture* użyta do ładowania tekstury została omówiona w kolejnym rozdziale.

Po załadowaniu obiektów przyszedł czas na ułożenie ich na planszy gry. Oprócz otoczenia i drzew użyty został także murek zaprojektowany ręcznie podczas II laboratorium.



Rys. 5.5. – efekt załadowania obiektów do gry

5.2.3. Podsumowanie

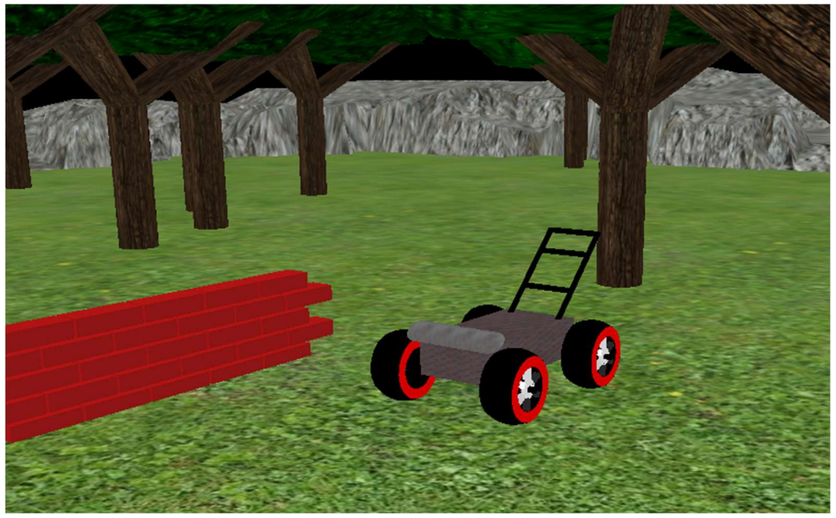
Jak wspomniano wyżej, największym osiągnięciem tego etapu było opracowanie metody, która pozwala na szybkie ładowanie pliku .obj do gry. Główną trudnością było odpowiednie użycie załadowanych wektorów wierzchołków oraz zachowanie zadanej skali.

5.3. Teksturowanie obiektów i sterowanie kamerą (laboratorium V)

5.3.1. Opis prac

Ten etap zakładał załadowanie tekstur z plików .jpg, tak aby pokryły one załadowane uprzednio obiekty. Do wczytania obrazów została wykorzystana biblioteka *STB_IMAGE*. Link do biblioteki:

https://github.com/nothings/stb



Rys. 5.6. – gra po załadowaniu tekstur

Kolejnym etapem była implementacja sterowania kamerą. Rozwiązanie pozwala na ruch kamery po orbicie dookoła kosiarki na pewnej wysokości za pomocą myszki po wciśnięciu jej prawego przycisku. Odległość kamery od kosiarki można regulować za pomocą scrolla.

5.3.2. Omówienie kodu

Ładowanie tekstur było tak naprawdę rozszerzeniem przedstawionej w poprzednim rozdziale metody ładowania obiektów z pliku. Ponieważ plik .obj posiada również informacje o położeniu tekstury, wystarczyło opracować sposób na załadowanie tekstury do zmiennej typu *Gluint* i użycie jej do pokrycia odczytanych wierzchołków.

```
GLuint loadTexture(const char* filename) {
    GLuint texture;
    glewInit();
    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;
}
```

List. 5.11. – funkcja loadTexture

Po ustawieniu odpowiednich parametrów tekstury, takich jak powtarzanie i zawijanie, kluczowym momentem metody jest użycie funkcji *stbi_load* służącej do załadowania pliku o podanej nazwie. Ważne było ustawienie ostatniego parametru na *STBI_rgb*, gdyż jest ono konieczne do prawidłowego wyświetlania kolorów.

Wywołanie *loadTexture* ma miejsce w funkcji *loadFile* przedstawionej na listingu 5.9. Użycie funkcji *glGenerateMipmap* wymusza wywołanie wyżej wymienionej metody dopiero po załadowaniu mipmapy podczas inicjalizacji gry – w przeciwnym razie zostanie zgłoszony wyjątek dostępu do stosu. Rysowanie załadowanych tekstur przedstawione jest na listingu 5.10.

Podstawą do wykonania ruchu kamery było odpowiednie ustawienie renderowania pola widzenia. Wykorzystano do tego funkcję *gluPerspective*, która umożliwia ustawienie FOV oraz odległości renderowania, a także modelu wyświetlania.

```
330  void Game::ChangeSize(GLsizei w, GLsizei h) {
331      if (h == 0) h = 1;
332      glViewport(0, 0, w, h);
333      glMatrixMode(GL_PROJECTION);
334      glLoadIdentity();
335      gluPerspective(45.0f, static_cast<GLfloat>(w) / static_cast<GLfloat>(h), 0.1f, camera.renderDistance);
336      glMatrixMode(GL_MODELVIEW);
337      glLoadIdentity();
338  }
```

List. 5.12. – ustawienie pola widzenia

Pozycja i cel kamery ustawiane są w każdej klatce gry w funkcji *updateCamera* za pomocą *gluLookAt*. Zmienne *camera.azimuth*, *camera.camDistance* oraz *camera.elevation* to składowe struktury *Camera* przechowujące informacje o kącie, odległości kamery od obiektu oraz jej wysokości nad mapą.

```
218  void Game::updateCamera() {
219      glLoadIdentity();
220      double a = camera.camDistance * cos(camera.elevation);
221      camera.xCamPos = lazik.xPos + a * cos(camera.azimuth);
222      camera.yCamPos = camera.camDistance * sin(camera.elevation);
223      camera.zCamPos = lazik.zPos + a * sin(camera.azimuth);
224      gluLookAt(camera.xCamPos, camera.yCamPos, camera.zCamPos, lazik.xPos, lazik.yPos, lazik.zPos, 0, 1, 0);
225  }
```

List. 5.13. – ustawienie pozycji kamery

Obliczenia nowych wartości *elevation* oraz *azimuth* mają miejsce w pętli gry w momencie wykrycia wejścia z myszki.

```
173      case WM_MOUSEMOVE:
174      {
175          if (dragging) {
176              POINTS currentMousePos = MAKEPOINTS(lParam);
177              int deltaX = currentMousePos.x - prevMousePos.x;
178              camera.azimuth += camera.angleJump * deltaX / 5;
179              prevMousePos = currentMousePos;
180          }
181          break;
182      }
```

List. 5.14. – obliczenie nowego azymutu

Azymut jest zwiększany o wartość *angleJump* po wykryciu ruchu myszką.

```
80      case WM_MOUSEWHEEL:
81      {
82          int delta = GET_WHEEL_DELTA_WPARAM(wParam);
83          if (delta > 0 && camera.camDistance > 150) {
84              camera.camDistance -= camera.radiusJump;
85          }
86          else if (delta < 0 && camera.camDistance < 1000) {
87              camera.camDistance += camera.radiusJump;
88          }
89          InvalidateRect(hWnd, NULL, FALSE);
90          break;
91      }
92      case WM_RBUTTONDOWN:
93      {
94          prevMousePos = MAKEPOINTS(lParam);
95          dragging = true;
96          break;
97      }
98      case WM_RBUTTONUP:
99      {
100         dragging = false;
101         break;
102     }
```

List. 5.15. – kod odpowiedzialny za obsługę prawego przycisku myszy i scrolla

Odległość kamery od łazika jest możliwa do zmiany w zakresie od 150 do 1000.

5.3.3. Podsumowanie

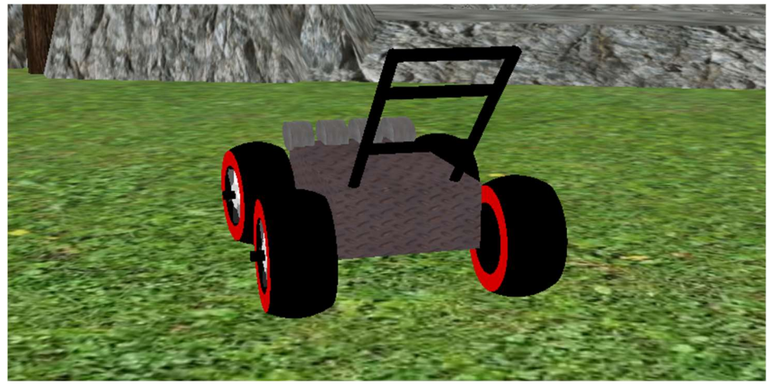
Etap okazał się dość skomplikowany mimo małej ilości kodu do napisania. Było to spowodowane ograniczoną dostępnością materiałów w internecie pozwalających na zgłębienie działania biblioteki *STB_IMAGE*. Okazało się że głównym powodem problemów było wywoływanie funkcji ładowania tekstury w nieodpowiednim miejscu, tzn. przed załadowaniem mipmapy przy inicjalizacji gry.

Implementacja ruchu kamery nie przyniosła żadnych większych problemów, gdyż opiera się ona w większości na opracowaniu odpowiednich obliczeń pozycji kamery.

5.4. Ruch pojazdu (laboratorium VI)

5.4.1. Opis prac

Etap zakładał stworzenie modelu jazdy łazikiem. Opracowany model zawiera mechanikę stopniowego przyśpieszenia i hamowania (pęd ciała), stopniowe skręcanie zależne od szybkości pojazdu a także animację kręcących się kół i skręcania przednich kół.



Rys. 5.7. – pojazd podczas skrętu

Sterowanie zostało udostępnione pod przyciskami W, A, S, D oraz Shift. Naciśnięcie W lub S powoduje ruch przód/tył, zaś A i D – w lewo i w prawo. Przytrzymanie przycisku Shift podczas jazdy powoduje zwiększenie prędkości maksymalnej łazika.

Kod umożliwia jednoczesne naciskanie przycisku jazdy przód/tył jak i skręcania, a także jazdę do tyłu.

5.4.2. Omówienie kodu

Model jazdy został zaimplementowany w funkcji *move* będącej metodą klasy łazik.

```cpp
void Lazik::move(bool pause, bool& isWKeyPressed, bool& isSKeyPressed, Camera* camera, set<int>& keysPressed)
{
    if (pause) {
        return;
    }
    const float acceleration = 0.3f;
    const float deceleration = 0.2f;
    wheelAngle = 0;
    if (isWKeyPressed || isSKeyPressed) {
        if (isWKeyPressed) {
            if (speed < maxSpeed) {
                speed += acceleration;
            }
            else speed = maxSpeed;
        }
        else if (isSKeyPressed && speed > -maxSpeed) {
            speed -= acceleration;
        }
        if (keysPressed.count('D') && !keysPressed.count('A')) {
            camera->azimuth += (speed / 3 * (GL_PI / 180));
            rot -= speed / 3;
            if (speed > 0) {
                wheelAngle = -speed * 2;
            }
            else {
                wheelAngle = speed * 2;
            }
        }
        else if (keysPressed.count('A') && !keysPressed.count('D')) {
            camera->azimuth -= (speed / 3 * (GL_PI / 180));
            rot += speed / 3;
            if (speed > 0) {
                wheelAngle = speed * 2;
            }
            else {
                wheelAngle = -speed * 2;
            }
        }
    }
    else {
        if (speed > 0) {
            speed -= deceleration;
        }
        else if (speed < 0) {
            speed += deceleration;
        }
        if (abs(speed) > 0.1) {

            if (keysPressed.count('D') && !keysPressed.count('A')) {
                camera->azimuth += (speed / 3 * (GL_PI / 180));
                rot -= speed / 3;
                if (speed > 0) {
                    if (speed > 0) {
                        wheelAngle = -speed * 2;
                    }
                    else {
                        wheelAngle = speed * 2;
                    }
                }
            }
            else if (keysPressed.count('A') && !keysPressed.count('D')) {
                camera->azimuth -= (speed / 3 * (GL_PI / 180));
                rot += speed / 3;
                if (speed > 0) {
                    wheelAngle = speed * 2;
                }
                else {
                    wheelAngle = -speed * 2;
                }
            }
        }
        else {
            speed = 0;
        }
    }
```

List. 5.16. – Funkcja move

Funkcja jest wywoływana w każdej klatce gry i przyjmuje m.in. zbiór aktualnie wciśniętych klawiszy oraz zmienne pomocnicze wskazujące na to czy jest wciśnięty przycisk jazdy do przodu lub do tyłu. Funkcję podzielić można zasadniczo na dwa fragmenty: jeden odpowiedzialny za ruch w momencie gdy użytkownik zwiększa lub zmniejsza prędkość, zaś drugi odpowiedzialny za ruch swobodny łazika.

W zależności od tego który klawisz kierunku jest wciśnięty, funkcja zwiększa lub zmniejsza kąt rotacji łazika. Zmiana kąta rośnie wprost proporcjonalnie do szybkości łazika. Podczas skrętu zmieniany jest również aktualny kąt nachylenia kół.

Wywołanie funkcji *move* ma miejsce nie tylko w momencie naciśnięcia

Obliczanie następnej pozycji łazika jest ściśle zintegrowane z mechanizmem wykrywania kolizji, więc mechanizm tego działania zostanie przedstawiony w następnym rozdziale.

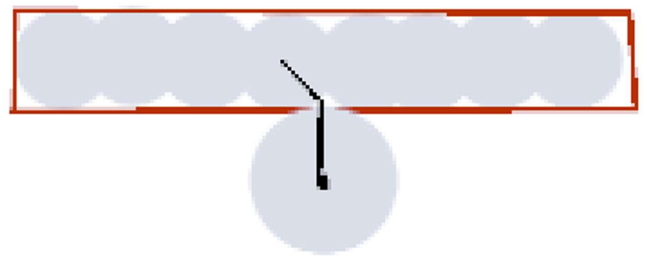
5.4.3. Podsumowanie

Mechanizm ruchu kosiarki nie przysporzył wielu trudności od strony programistycznej, ale wymagał logicznego myślenia i analizy poszczególnych sytuacji. Największym przełomem było wywoływanie funkcji *move* nie tylko przy naciśnięciu przycisków, ale w każdej klatce programu, dzięki czemu możliwe jest odtworzenie ruchu żądanego przez naciskanie wielu przycisków naraz.

5.5. Kolizje (laboratorium VII)

5.5.1. Opis prac

Wykrywanie kolizji zostało zaimplementowane przy pomocy modelu okręgu o środku w środku kosiarki oraz o promieniu o długości będącej kompromisem pomiędzy najdalej i najbliżej wysuniętym od środka kosiarki punktem. Również obszar wykrywania kolizji dla innych elementów reprezentują okręgi ustawione przy sobie z odpowiednim odstępem. Podczas projektowania kolizji posłużono się następującym modelem:



Rys. 5.8. – model kolizji ze ścianą

W przedstawionym modelu kolizja jest wykrywana, kiedy odległość między środkiem dużego koła (łazik) a jednego z małych kół (ściana) jest mniejsza od sumy promieni tych kół. Pozwala to na dość dokładne odwzorowanie kolizji przy założeniu, że kształt łazika w rzucie na oś XY jest zbliżony do kwadratu.

Aby zwiększyć wydajność, mapa została podzielona na 4 obszary wykrywania kolizji. Dzięki temu program nie musi za każdym razem sprawdzać odległości łazika od wszystkich punktów kolizyjnych, ale tylko tych, które znajdują się w obszarze, w jakim aktualnie znajduje się pojazd.

5.5.2. Omówienie kodu

Podczas tworzenia drzewa losowane są jego współrzędne (z pominięciem obszaru w promieniu 30 od miejsca utworzenia łazika). Następnie na ich podstawie jest ustawiana wartość *sphereID* oznaczająca numer strefy kolizji w jakiej znajduje się drzewo.

```
Tree::Tree()
{
    x = randomSelect(Terrain::minX, Terrain::maxX);
    z = randomSelect(Terrain::minZ, Terrain::maxZ);
    if (z < 0) {
        if (x > 0) {
            sphereID = 1;
        }
        else {
            sphereID = 2;
        }
    }
    else {
        if (x > 0) {
            sphereID = 3;
        }
        else {
            sphereID = 4;
        }
    }
}
```

List. 5.17. – konstruktor obiektu klasy Tree

Następnie, przy tworzeniu obiektu klasy Terrain drzewa zostają zapisane do jednego z czterech wektorów – po jednym na każdy obszar kolizji.

```
Terrain::Terrain() : wall(30, -10, 20 + 70, 5, 5)
{
    for (int i = 0; i < treesAmount; ++i) {
        Tree tree;
        trees.push_back(tree);
        switch (tree.sphereID) {
        case 1:
            points1.push_back({tree.x, tree.z});
            break;
        case 2:
            points2.push_back({tree.x, tree.z});
            break;
        case 3:
            points3.push_back({tree.x, tree.z});
            break;
        case 4:
            points4.push_back({tree.x, tree.z});
            break;
        }
    }
```

List. 5.18. – przydzielenie drzew do obszarów kolizji

W podobny sposób dodawane są punkty kolizji dla otoczenia oraz murku. Klasa terrain posiada także metodę *checkCollision*, która sprawdza odpowiednie punkty kolizji w stosunku do podanych jej w parametrze współrzędnych łazika.

```
bool Terrain::checkCollision(POINT coords) {
    int distance = Terrain::collisionPointDistance + Lazik::collisionDistance;
    std::vector<POINT>* pointsToCheck;
    if (coords.y < 0) {
        pointsToCheck = (coords.x > 0) ? &points1 : &points2;
    }
    else {
        pointsToCheck = (coords.x > 0) ? &points3 : &points4;
    }
    for (auto point : *pointsToCheck) {
        if (dist(point, coords) < distance) {
            return true;
        }
    }
    return false;
}
```

List. 5.19. – wykrywanie kolizji

Powyższa metoda jest wywoływana w każdej klatce podczas ruchu kosiarki w klasie *Game*.

```
void Game::checkCollisions()
{
    if (pause) {
        return;
    }
    GLfloat nextX = lazik.xPos + lazik.speed * sin(lazik.rot * (GL_PI / 180) + GL_PI / 2);
    GLfloat nextY = lazik.zPos + lazik.speed * cos(lazik.rot * (GL_PI / 180) + GL_PI / 2);
    POINT next{ nextX, nextY };
    auto currentTime = std::chrono::high_resolution_clock::now();
    if (!terrain.checkCollision(next)) {
        lazik.xPos = nextX;
        lazik.zPos = nextY;
    }
    else {
        if (abs(lazik.speed) > 2.2) {
            points -= 50;
            isCollision = true;
            lastCollisionCheckTime = currentTime;
            SoundEngine->play2D("audio/explosion.wav", false);
            lazik.speed = -lazik.speed / 2;
        }
        else {
            lazik.speed = 0;
        }
    }
    terrain.checkPotatoes(next, points, SoundEngine, level);
    auto elapsedTime = std::chrono::duration_cast<std::chrono::milliseconds>(currentTime - lastCollisionCheckTime).count();
    if (elapsedTime > 2000) {
        isCollision = false;
    }
}
```

List. 5.20. – wykrywanie kolizji w klasie Game

Metoda *checkCollisions* jest uzupełnieniem funkcji *move* omawianej w poprzednim rozdziale. Wyznacza ona następne koordynaty kosiarki, uwzględniając kąt jej obrotu oraz prędkość. Następnie sprawdza, czy została wykryta kolizja i jeśli tak, to wywoływany jest efekt odbicia pojazdu od przeszkody poprzez ustawienie prędkości na wartość odwrotną pomniejszoną dwukrotnie.

Aby zapobiec ciągłemu wykrywaniu kolizji, nawet wtedy gdy pojazd stoi stykając się jedynie z przeszkodą, sprawdzana jest aktualna prędkość pojazdu. Kolizja jest wykrywana jedynie, jeśli prędkość w momencie uderzenia była większa od 2,2.

5.5.3. Podsumowanie

Wykrywanie kolizji nie zostało wykonane w najbardziej dokładny sposób, jednak jest to metoda której głównym celem jest utrzymanie optymalności oraz wysokiej wydajności gry. Dla kształtu pojazdu działa ona bardzo dobrze i rzadkością jest sytuacja kiedy pojazd zostaje zablokowany w przeszkodzie – nie zdarza się ona prawie nigdy, jednak w pewnych bardzo specyficznych

sytuacjach jej wystąpienie jest możliwe. W takim wypadku użytkownik ma możliwość zresetowania pozycji pojazdu do 0,0 za pomocą klawisza spacji.

5.6. Oświetlenie i fabuła (laboratorium VIII)

5.6.1. Opis prac

Zasadniczo prace wykonane w tym etapie można podzielić na dwie części – wykonanie oświetlenia oraz zaimplementowanie prostej fabuły.

Oświetlenie w projekcie ma jedno źródło – reflektory pojazdu. Promienie padające z nich oświetlają otoczenie, które domyślnie jest przyciemnione by uzyskać efekt nocnej jazdy. Dodatkowo zostało dodane oświetlenie które rozjaśnia menu użytkownika.

Fabuła gry polega na zbieraniu losowo wygenerowanych ziemniaków i unikaniu kolizji z obiektami dookoła. Po zebraniu wszystkich ziemniaków zwiększa się poziom, losowanych jest więcej ziemniaków niż poprzednio, zaś jeśli pobity został najwyższy wynik zapisany w pliku .txt, to jest on zapisywany.

Wyświetlany jest także interfejs użytkownika, na którym gracz może sprawdzić stan gry oraz sterowanie.

Rys. 5.9. – oświetlona gra oraz interfejs użytkownika z włączonym sterowaniem

5.6.2. Omówienie kodu

Do ustawienia świateł pojazdu używana jest metoda *updateLight* z klasy *Lazik*. Metoda przygotowuje środowisko OpenGL do renderowania, aktywując oświetlenie, ustawiając parametry światła punktowego (GL_LIGHT0) i definiując pewne właściwości oświetlenia, takie jak kierunek, kąt odcięcia i składowe światła. Dodatkowo inicjalizuje ustawienia kolorów, materiałów i modelu cieniowania, przygotowując scenę do prawidłowego renderowania obrazu 3D.

```
129 void Lazik::updateLight()
130 {
131     glEnable(GL_COLOR_MATERIAL);
132     glEnable(GL_MULTISAMPLE);
133     glEnable(GL_DEPTH_TEST);
134     glEnable(GL_LIGHTING);
135     glEnable(GL_LIGHT0);
136     glNormal3f(0, 1, 0);
137     glColor3f(0.8f, 0.8f, 0.8f);
138     glClearColor(0.0, 0, 0.0, 1.0f);
139     glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
140     glShadeModel(GL_SMOOTH);
141     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
142     GLfloat lightDir[] = { xPos, yPos, zPos, 1.0f };
143     glLightfv(GL_LIGHT0, GL_POSITION, lightDir);
144     GLfloat spotCutoff = 40.0;
145     glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 20);
146     glLightfv(GL_LIGHT0, GL_SPOT_CUTOFF, &spotCutoff);
147     GLfloat spotDirection[] = {
148         static_cast<GLfloat>(sin((rot + 90) * GL_PI / 180) * cos(lightPos)),
149         static_cast<GLfloat>(sin(lightPos)),
150         static_cast<GLfloat>(cos((rot + 90) * GL_PI / 180) * cos(lightPos))
151     };
152     glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spotDirection);
153     GLfloat increasedAmbient[] = { 1.0f, 1.0f, 1.0f, 1.0f };
154     glLightfv(GL_LIGHT0, GL_AMBIENT, increasedAmbient);
155     GLfloat increasedDiffuse[] = { 1.2f, 1.2f, 1.2f, 1.0f };
156     glLightfv(GL_LIGHT0, GL_DIFFUSE, increasedDiffuse);
157     GLfloat increasedSpecular[] = { 1.0f, 1.0f, 1.0f, 3.0f };
158     glLightfv(GL_LIGHT0, GL_SPECULAR, increasedSpecular);
159 }
```

List. 5.21. – metoda updateLight

Do utworzenia fabuły zbierania ziemniaków został wykorzystany podobny mechanizm wykrywania kolizji jak ten znany z klasy *Tree*. W programie Blender został zaprojektowany model ziemniaka, zaś podczas tworzenia terenu i przy każdym następnym poziomie zostaje załadowany obiekt wraz z teksturą na losowo wybranych koordynatach.

Kolizje ziemniaków są sprawdzane za pomocą funkcji *checkPotatoes*. Po zebraniu ziemniaka gracz otrzymuje 30 punktów, zaś po uderzeniu w przeszkodę zabierane mu jest 50 punktów.

Ziemniaki mają własną fizykę, obracają się one wokół własnej osi, zaś ich wysokość pionową można opisać funkcją y=sin(x), gdzie x wzrasta o 0,1 w każdej klatce. Implementacja fizyki znajduje się w funkcji *draw* klasy *Terrain*.

```
63  void Terrain::draw()
64  {
65      glPushMatrix();
66      glPolygonMode(GL_BACK, GL_LINE);
67      drawObj(&plate, 0, -20, 0, scale, scale / 2, scale, 10, 10);
68      drawObj(&mountains, 0, -20, 0, scale, scale / 2, scale, 10, 10);
69      for (int i = 0; i < treesAmount; i++) {
70          trees[i].draw();
71      }
72      wall.draw();
73      glPopMatrix();
74      for (auto& potato : potatoes) {
75          glPushMatrix();
76          glPolygonMode(GL_BACK, GL_LINE);
77          glTranslatef(potato.x, 2 * sin(potato.heightArgument), potato.z);
78          glRotatef(potato.angle, 0, 1, 0);
79          potato.heightArgument += 0.1;
80          if (potato.heightArgument > 1000) potato.heightArgument = 0;
81          potato.angle += 1;
82          if (potato.angle > 360) potato.angle = 0;
83          potato.draw();
84          glPopMatrix();
85      }
86  }
```

List. 5.22. – fizyka ziemniaków

```
void Terrain::checkPotatoes(POINT coords, int& points, ISoundEngine* soundEngine, int& level) {
    std::vector<int> toRm;
    int counter = 0;
    for (auto& potato : potatoes) {
        POINT point{ potato.x, potato.z };
        if (dist(point, coords) < collisionPointDistance + Lazik::collisionDistance) {
            points += 30;
            toRm.push_back(counter);
            soundEngine->play2D("audio/collect.mp3", false);
            if (toRm.size() == 1 && potatoes.size() == 1) {
                level++;
                Terrain::potatoesAmount += 3;
                loadPotatoes();
                soundEngine->play2D("audio/level.mp3", false);
                if (points > personalBest) {
                    personalBest = points;
                    fstream out;
                    out.open("data.txt", ios::out);
                    out << personalBest;
                    out.close();
                }
            }
        }
        counter++;
    }
    for (auto i : toRm) {
        potatoes.erase(potatoes.begin() + i);
    }
}
```

List. 5.23. – wykrywanie zebrania ziemniaka

Jak widać w kodzie powyżej, w momencie zebrania ziemniaka bądź wejścia na wyższy poziom włączany jest dźwięk. Do obsługi plików audio została wykorzystana biblioteka IrrKlang. Link do biblioteki:

https://www.ambiera.com/irrklang/

W celu wyświetlania graczowi podstawowych parametrów takich jak poziom, liczba punktów, aktualny rekord czy czas rozgrywki oraz zapoznania go ze sterowaniem wyświetlany jest interfejs użytkownika w 2D. Do jego podświetlenia wykorzystana jest stała GL_LIGHT1.

```
222
223  void Game::drawDashboard() {
224      glEnable(GL_LIGHTING);
225      glEnable(GL_LIGHT1);
226      GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
227      GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
228      glLightfv(GL_LIGHT1, GL_POSITION, light_position);
229      glLightfv(GL_LIGHT1, GL_DIFFUSE, light_diffuse);
230
231      char collisionCountText[100];
232      char levelText[100];
233      char left[100];
234      char timeText[100];
235      char collText[100];
236      char pauseText[100];
237      char steeringText1[20];
238      char steeringText2[50];
239      char steeringText3[20];
240      char steeringText4[20];
241      char steeringText5[20];
242      char steeringText6[30];
243      char steeringText7[30];
244      char steeringText8[30];
245      char helpText[20];
246      char pbText[20];
247      sprintf(collisionCountText, "Punkty: %d", points);
248      sprintf(levelText, "Poziom %d", level);
249      sprintf(left, "Pozostalo %d ziemniakow", terrain.potatoes.size());
250      sprintf(timeText, "Czas gry: %d sekund", gameTimeSeconds);
251      sprintf(collText, "Kolizja! -50");
252      sprintf(steeringText1, "Pauza [ESC]");
253      sprintf(steeringText4, "Od nowa [R]");
254      sprintf(steeringText2, "Przod [W], tyl [S], lewo [A], prawo [D]");
255      sprintf(steeringText3, "Przyspiesz [Shift]");
256      sprintf(steeringText5, "Swiatla [UP]/[DOWN]");
257      sprintf(steeringText6, "Kamera [PPM + mouse]");
258      sprintf(steeringText8, "ZOOM [Scroll]");
259      sprintf(steeringText7, "Reset pozycji [SPACE]");
260      sprintf(helpText, "Sterowanie [H]");
261      sprintf(pbText, "Rekord: %d", terrain.personalBest);
262      DrawText(timeText, 20, 170);
263      DrawText(collisionCountText, 20, 20);
264      DrawText(levelText, 20, 70);
265      DrawText(left, 20, 120);
266      DrawText(helpText, 20, 920);
267      DrawText(pbText, 300, 920);
268      if (pause) {
269          sprintf(pauseText, "Pauza");
270          DrawText(pauseText, 700, 120);
271      }
272      if (isCollision) {
273          DrawText(collText, 700, 20);
274      }
275      if (isHelp) {
276          DrawText(steeringText1, 20, 580);
277          DrawText(steeringText4, 20, 540);
278          DrawText(steeringText2, 20, 500);
279          DrawText(steeringText3, 20, 460);
280          DrawText(steeringText5, 20, 420);
281          DrawText(steeringText6, 20, 380);
282          DrawText(steeringText8, 20, 340);
283          DrawText(steeringText7, 20, 300);
284      }
285      glDisable(GL_LIGHTING);
286      glDisable(GL_LIGHT1);
287  }
```

List. 5.24. – wyświetlanie interfejsu

5.6.3. Podsumowanie

Dopracowanie kosmetyczne gry przyniosło w skutkach zwiększenie wydajności oraz komfortu użytkowania. Fabuła jest ciekawa i zachęcająca do gry, ponieważ poziomy nigdy się nie kończą, a maksymalny wynik zebranych ziemniaków ograniczony jest jedynie cierpliwością gracza i pojemnością pamięci RAM, gdyż przy bardzo dużej liczbie ziemniaków wzrasta ilość punktów kolizji do sprawdzenia, a co za tym idzie – spada wydajność.

6. Podsumowanie

Projekt "NightMower" to trójwymiarowa gra stworzona w języku C++ z użyciem biblioteki OpenGL. W ramach projektu zrealizowano założenia obejmujące podstawowe zjawiska fizyczne, takie jak sterowanie pojazdem, kolizje z obiektami, oraz pęd poruszającego się ciała. Dodatkowo, w grze zaimplementowano teksturowanie obiektów, poruszanie kamerą, oraz dodano elementy prostej fabuły.

Głównym celem gry jest sterowanie kosiarką, zbieranie ziemniaków rozmieszczonych na trawie oraz unikanie przeszkód w postaci drzew i murów. Gracz zdobywa punkty za zebranie ziemniaków, ale traci je za kolizje z przeszkodami. Najlepszy wynik zapisywany jest do pliku, a gra oferuje także różne funkcje sterowania, takie jak zatrzymywanie, restartowanie, czy powrót do początkowej pozycji.

Projekt został podzielony na kilka etapów, zaczynając od stworzenia obiektu kosiarki, a następnie dodawania otoczenia za pomocą plików .obj z programu Blender. Teksturowanie obiektów oraz sterowanie kamerą zostały zaimplementowane w kolejnych etapach, umożliwiając pełniejsze doświadczenie wizualne dla gracza.

Model ruchu pojazdu obejmuje stopniowe przyśpieszanie, hamowanie, skręcanie zależne od szybkości pojazdu, oraz animację obracających się kół. Ruch pojazdu został zintegrowany z wykrywaniem kolizji, które zostały zamodelowane na podstawie okręgów reprezentujących poszczególne elementy gry.

Wszystkie etapy projektu zostały zrealizowane zgodnie z założeniami, a implementacja gry opiera się na paradygmacie obiektowym w języku C++. Projekt uwzględnia także optymalizacje, takie jak podział obszaru kolizji na mniejsze sektory, co poprawia wydajność gry.

W rezultacie, "NightMower" stanowi kompleksową grę 3D, która oferuje użytkownikowi ciekawe wyzwanie, kreatywną oprawę wizualną, oraz różnorodność w mechanice rozgrywki.

Projekt okazał się niełatwym orzechem do zgryzienia głównie od strony załadowania obiektów i tekstur do gry. Sama logika nie była tak trudna do zaimplementowania, jednak wymagania starszej wersji OpenGL stawiały trudności przez mniejszą dostępność informacji o niej w internecie.