P O L I T E C H N I K A Ś W I Ę T O K R Z Y S K A
Wydział Elektrotechniki, Automatyki i Informatyki

Projekt: Podstawy grafiki komputerowej 2		
Temat projektu: **City Builder**		
I termin		
Informatyka - II rok	Wykonali: **Dunytskyi Daniil** **Kozieł Karol** **Orzeł Arkadiusz**	Grupa: **2ID14A**
Rok akademicki - 2023/2024		Semestr 4

Opis zastosowanych technologii:

Użyty język programowania: C++.

Biblioteki: OpenGL, GLFW, GLM i ImGui.

IDE: Microsoft Visual Studio 2022.

OS: Windows 10.

Kompilator: MSVC.

Opis implementacji z fragmentami kodu źródłowego:

Stworzyliśmy grę, w której użytkownik może kontrolować mapę i umieszczać na niej obiekty. Teraz pokażemy implementację tego, co zrobiliśmy. Zacznijmy od struktury. Nasza gra 3D ma dobrze zorganizowaną strukturę. Kod został rozproszony na wiele plików celem zachowania przejrzystości.

Ogólna struktura gry:

```
Engine
│
├── shaders
│       ├── test
│               ├── fragment.glsl
│               ├── vertex.glsl
│               ├── (...)                    // Inne shadery
│
├── source
│       ├── Engine
│               ├── main.cpp
│               ├── Engine.h
│               ├── Engine.cpp
│               ├── (...)                    // Inne pliki z kodem źródłowym silnika
│
├── textures
│       ├── (...)                            // Inne pliki tekstur, jeżeli istnieją
│
└── (...)                                    // Inne foldery i pliki
```

Przede wszystkim mamy folder **shaders**, który zawiera pliki glsl używane w programowaniu graficznym przy użyciu OpenGL. Kod tych plików nie jest skomplikowany, ale bardzo ważny dla naszej gry. Używamy ich do oświetlenia, kamery, obiektów w grze itp. Nie zalecamy wprowadzania tam żadnych zmian (tylko w ostateczności).

Folder **source** zawiera kod źródłowy gry, w tym plik *main.cpp*. Kluczowy plik *main.cpp* zawiera główną funkcję, inicjującą silnik gry.

Struktura main.cpp:

```
#include ...

#define WINDOW_WIDTH 1360
#define WINDOW_HEIGHT 768
#define WINDOW_TITLE "Game 3D"

int main()
{
	using namespace eng;

	Engine engine(WINDOW_WIDTH, WINDOW_HEIGHT, WINDOW_TITLE);

	// Debugowanie
	// (...)

	// Część właściwa.
	engine.Run();
}
```

Jak widzimy, *main.cpp* jest odpowiedzialny za uruchomienie silnika, ale może również ustawiać ważne parametry. Na przykład, możemy zmienić wartości stałych WINDOW_WIDTH, WINDOW_HEIGHT, WINDOW_TITLE. Oznaczają one odpowiednio szerokość i wysokość okna oraz jego nazwę.

Zaleca się, aby nie wprowadzać żadnych zmian w pliku *main.cpp*, ponieważ powinien on być odpowiedzialny tylko za uruchomienie silnika naszej gry. Z kolei plik *Engine.cpp* zawiera główną pętlę, która obsługuje zdarzenia, współpracuje z GUI, aktualizuje parametry kamery i rysuje obiekty.

Struktura Engine.cpp:

```
#include "engine.h"

namespace eng
{
	void Engine::Run()
	{
		std::cout << "OpenGL: " << glGetString(GL_VERSION) << std::endl;

		// ------------------------------------------
		//          Domyślne shadery.
		// ------------------------------------------
		// (...)

		// ------------------------------------------
		//          Tworzenie obiektów gry
		// ------------------------------------------
		// (...)

		Renderer* renderer = Renderer::Get();

		// Pętla główna.
		while (!m_Window.ShouldClose())
		{
			// ------------------------------------------
			//          Logika
			// ------------------------------------------
			// (...)
```

```
                    // ------------------------------------------
                    //          Rysowanie
                    // ------------------------------------------
                    // (...)

                    m_Window.SwapBuffers();
              }
        }

        //Inne metody
}
```

Tak wygląda główna pętla silnika gry. Tutaj interesują nas cztery główne rzeczy - shadery, tworzenie obiektów, logika i rysowanie. Pierwsze trzy omówimy później, ale teraz warto pokazać przykład renderowania.

```
// ------------------------------------------
//          Rysowanie
// ------------------------------------------
renderer->Clear(ENG_CLEAR_COLOR * 1.0f);

//Wywoływanie obiektów poprzez UpdatableVector.
for (const auto& object : UpdatableVector)
      object->Draw();

//Bezposrednie wywoływanie obiektów.
if (isHoveredShop)
      cell1.Draw();

// (...)
```

Ten przykład pokazuje renderowanie z wywołaniem wszystkich obiektów przechowywanych w *UpdatableVecto*r i bezpośrednie wywołanie bez tego wektora.

Wiadomo, że aby coś narysować, potrzebne są obiekty, wiec deweloper może tworzyć obiekty bezpośrednio w klasie Engine.

```
// ------------------------------------------
//          Tworzenie obiektów gry
// ------------------------------------------

std::vector<BaseObject*> UpdatableVector;

glm::mat4x3 positions;
positions[0] = glm::vec3(-1.0f, 0.0f, 1.0f);
positions[1] = glm::vec3(-1.0f, 0.0f, -1.0f);
positions[2] = glm::vec3(1.0f, 0.0f, -1.0f);
positions[3] = glm::vec3(1.0f, 0.0f, 1.0f);

Map* map = new Map(positions, shaderProgram);
UpdatableVector.push_back(map);
```

Istnieje również folder **textures** zawierający wszystkie tekstury potrzebne programiście. Sposób korzystania z tekstur zostanie opisany poniżej.

Przejdźmy teraz do szczegółów dotyczących aspektów silnika i sposobu korzystania z nich:

- **Obsługa klawiatury i myszy:**

Obsługa klawiatury i myszy może działać równie dobrze w funkcji lub bezpośrednio w silniku gry. Przyklad korzystania:

```
// ------------------------------------------
//           Logika
// ------------------------------------------
if (glfwGetKey(m_Window.GetWindow(), GLFW_KEY_UP) == GLFW_PRESS)
{
      //(...)
}

m_Camera.Inputs(m_Window.GetWindow());
```

Tutaj funkcja inputs() zawiera obsluge klawiatury i myszy.

- **Obsługa kamery:**

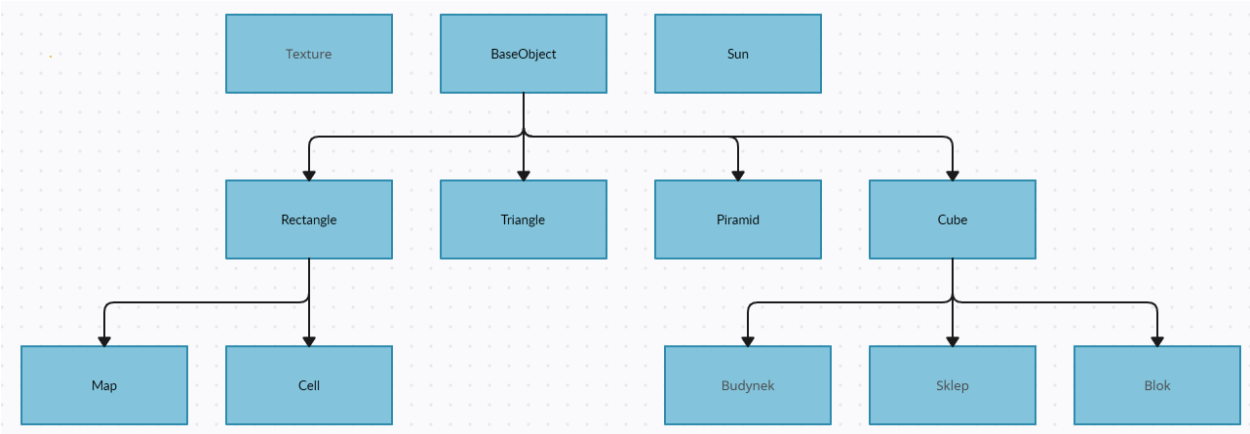
Przykład konfiguracji kamery w naszej grze:

```
void Engine::Run()
{
      // ------------------------------------------
      //           Domyślne shadery.
      // ------------------------------------------
      // (...)

      // Pętla główna.
      while (!m_Window.ShouldClose())
      {
            // ------------------------------------------
            //           Logika
            // ------------------------------------------
            m_Window.HandleEvents();

            m_Camera.updateMatrix(45.0f, 0.1f, 100.0f);

            shaderProgram.Activate();
            m_Camera.Matrix(shaderProgram, "camMatrix");

            // (...)

            lightShader.Activate();
            m_Camera.Matrix(lightShader, "camMatrix");

            m_Window.SwapBuffers();
            }
}
```

Widzimy, że kamera powinna działać tak samo zarówno dla pierwszego, jak i drugiego shadera.

- **Hierarchia klas:**

Opracowaną hierarchię klas mamy w pliku *GameObjects.h*. Hierarchia obiktów gry (budynków) jest tam bardzo dobrze opracowana.



Hierarchia klas (kod):

```
#include ...

namespace eng
{
	class Texture
	{
	public:
		//Konstruktory, pola i metody.

	};

	class BaseObject
	{
	public:
		//Konstruktory, pola i metody.

	};

	class Triangle : public BaseObject
	{
	public:
		//Konstruktory, pola i metody.
	};

	class Rectangle : public BaseObject
	{
	public:
		//Konstruktory, pola i metody.
	};

	class Map : public Rectangle
	{
	public:
		//Konstruktory, pola i metody.
	};

	class Cell : public Rectangle
	{                                       //Klasa dziedzicząca po klasie Rectangle
	public:
		//Konstruktory, pola i metody.
	 };

	 // Możliwe klasy, np:

	class Cube: public BaseObject
	{
	public:
		//Konstruktory, pola i metody.
```

```
	};

	class Budynek: public Cube
	{                                        //Klasa dziedzicząca po klasie Cube
	public:
		//Konstruktory, pola i metody.

	};

	// Inne klasy...

}
```

- **Oświetlenie:**

Oświetlenie znajduje się w shaderach. Przykład użycia oświetlenia w grze:

```
void Engine::Run()
{
	// ---------------------------------------------
	//           Domyślne shadery.
	// ---------------------------------------------
	Shader shaderProgram("shaders/test/vertex.glsl","shaders/test/fragment.glsl");
	Shader lightShader("shaders/lightvertex.glsl","shaders/ligthfragment.glsl");

	lightShader.Activate();
	glUniformMatrix4fv(glGetUniformLocation(lightShader.ID, "model"), 1, GL_FALSE,
glm::value_ptr(lightModel));
	glUniform4f(glGetUniformLocation(lightShader.ID, "lightColor"), lightColor.x,
lightColor.y, lightColor.z, lightColor.w);
	shaderProgram.Activate();
	glUniformMatrix4fv(glGetUniformLocation(shaderProgram.ID, "model"), 1,
GL_FALSE, glm::value_ptr(pyramidModel));
	glUniform4f(glGetUniformLocation(shaderProgram.ID, "lightColor"), lightColor.x,
lightColor.y, lightColor.z, lightColor.w);
	glUniform3f(glGetUniformLocation(shaderProgram.ID, "lightPos"), lightPos.x,
lightPos.y, lightPos.z);


	// Pętla główna.
	while (!m_Window.ShouldClose())
	{
		// ---------------------------------------------
		//           Logika
		// ---------------------------------------------
		m_Window.HandleEvents();

		m_Camera.updateMatrix(45.0f, 0.1f, 100.0f);

		shaderProgram.Activate();
		m_Camera.Matrix(shaderProgram, "camMatrix");

		// (...)

		lightShader.Activate();
		m_Camera.Matrix(lightShader, "camMatrix");

		m_Window.SwapBuffers();
		}
}
```

Ten kod deklaruje obiekt klasy Shader. Zarówno dla zwykłego shadera, jak i dla shadera oswietlenia. Widzimy, że musimy najpierw aktywować zwykły shader, aby coś z nim zrobić, a dopiero potem shader oświetlenia. W taki sposob nic sie nie zepsuje.

Deweloper może również przejść do pliku *fragment.glsl* i spróbować zmienić oświetlenie, ale to już zostało zrobione.

fragment.glsl:

```
#version 330 core

// (...)

vec4 pointLight()
{
	// (...)
}

vec4 directtLight()
{
	// (...)
}


vec4 spotLight()
{
	// (...)
}

void main()
{
	FragColor = pointLight(); // Tutaj deweloper może zmienić funkcję
}
```

Wystarczy zmienić FragColor na funkcję inną niż sugerowane.

- **Teksturowanie obiektów:**

Ważnym aspektem gry są tekstury, które dotyczą obiektów 3D. Nasze “budynki” to po prostu sześciany pokryte teksturami z każdej strony.

Przykład użycia tekstur:

```
Map::Map(const glm::mat4x3& positions, Shader& shader) : Rectangle(positions),
	planksTex("textures/grs.png", GL_TEXTURE_2D, 0, GL_RGBA, GL_UNSIGNED_BYTE),
	planksSpec("textures/planksSpec.png", GL_TEXTURE_2D, 1, GL_RED,
GL_UNSIGNED_BYTE)
{
	planksTex.texUnit(shader, "tex0", 0);
	planksSpec.texUnit(shader, "tex1", 1);
}

void Map::Draw()
{
	planksTex.Bind();
	planksSpec.Bind();

	m_VAO.Bind();
	glDrawElements(GL_TRIANGLES, m_EBO.GetCount(), GL_UNSIGNED_INT, 0);
	m_VAO.Unbind();

	planksTex.Unbind();
	planksSpec.Unbind();
```

```
}
```

Tutaj planksTex i planksSpec są teksturami dla mapy. Możemy dołączyć je do shaderów za pomocą *texUnit()* i aktywować je za pomocą *Bind()*.

- **Interfejs graficzny:**

Aby zaimplementować komunikację z użytkownikiem, stworzyliśmy graficzny interfejs użytkownika, który wykorzystuje bibliotekę ImGui.

Przykład użycia:

```
void Engine::Run()
{
	IMGUI_CHECKVERSION();
	ImGui::CreateContext();
	ImGuiIO& io = ImGui::GetIO(); (void)io;
	ImGui::StyleColorsDark();
	ImGui_ImplGlfw_InitForOpenGL(m_Window.GetWindow(), true);
	ImGui_ImplOpenGL3_Init("#version 330");
	// Pętla główna.
	while (!m_Window.ShouldClose())
	{
		// ---------------------------------------------
		//           Rysowanie
		// ---------------------------------------------
		renderer->Clear(ENG_CLEAR_COLOR * 1.0f);

		ImGui_ImplOpenGL3_NewFrame();
		ImGui_ImplGlfw_NewFrame();
		ImGui::NewFrame();

		ImGui::Begin("Statistics");
		ImGui::Text("Your balance: %d", balance);
		ImGui::End();

		ImGui::Render();
		ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());

	}
	ImGui_ImplOpenGL3_Shutdown();
	ImGui_ImplGlfw_Shutdown();
	ImGui::DestroyContext();
}
```

Ten kod tworzy małe okno przy użyciu ImGui.

Instrukcja obsługi “dla dewelopera”:

Nie używamy żadnych narzędzi, naszym IDE jest Microsoft Visual Studio 2022, a kompilacja przebiega sprawnie. Istnieją jednak błędy, których należy być świadomym. Właśnie dlatego mamy obsługę błędów, o której deweloper powinien wiedzieć. W naszym projekcie to wyglada nastepujaca:

1. Najbardziej podstawową rzeczą jest obsługa błędów OpenGL i zrobiliśmy to sami w pliku main.cpp:

```
#include ...
```

```
int main()
{
	using namespace eng;

	Engine engine(WINDOW_WIDTH, WINDOW_HEIGHT, WINDOW_TITLE);

	// Debugowanie
	glDebugMessageControl(GL_DEBUG_SOURCE_API, GL_DEBUG_TYPE_OTHER,
GL_DEBUG_SEVERITY_NOTIFICATION, 0, nullptr, GL_FALSE);
	glEnable(GL_DEBUG_OUTPUT);
	glDebugMessageCallback(MessageCallback, 0);

	// Część właściwa.
	engine.Run();
}
```

Takie debugowanie pozwala deweloperowi zrozumieć 90% błędów w projekcie.

2. Mamy gotowe funkcje do sprawdzania błędów kompilacji. Na przykład, w klasie shadera znajduje się funkcja:

```
void compileErrors(unsigned int shader, const char* type);
```

Która sprawdza, czy shadery są poprawnie zainstalowane, a jeśli nie, wyświetla odpowiedni komunikat.

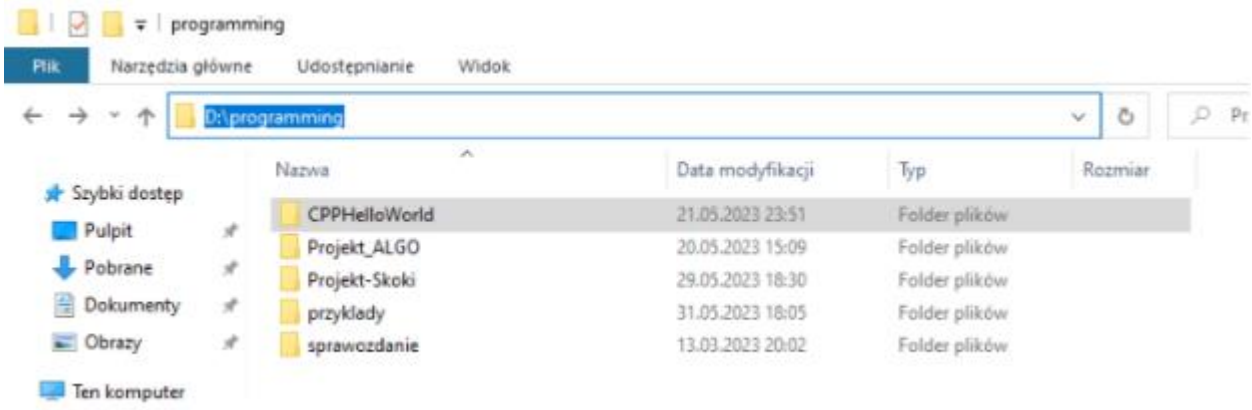
3. Sprawdzanie działających funkcji (wyjątki). Tutaj chodzi bardziej o funkcji, które obejmują obsługę błędów. Na przykład, konstruktor tekstur zawiera try-catch:

```
Texture::Texture(const char* image, GLenum texType, GLuint slot, GLenum format,
GLenum pixelType) {
	type = texType;
	int widthImg, heightImg, numColCh;
	unsigned char* bytes = nullptr;

	try
	{
		//(...)
	}
	catch (const std::runtime_error& e) {
		//(...)

		throw;
	}
}
```

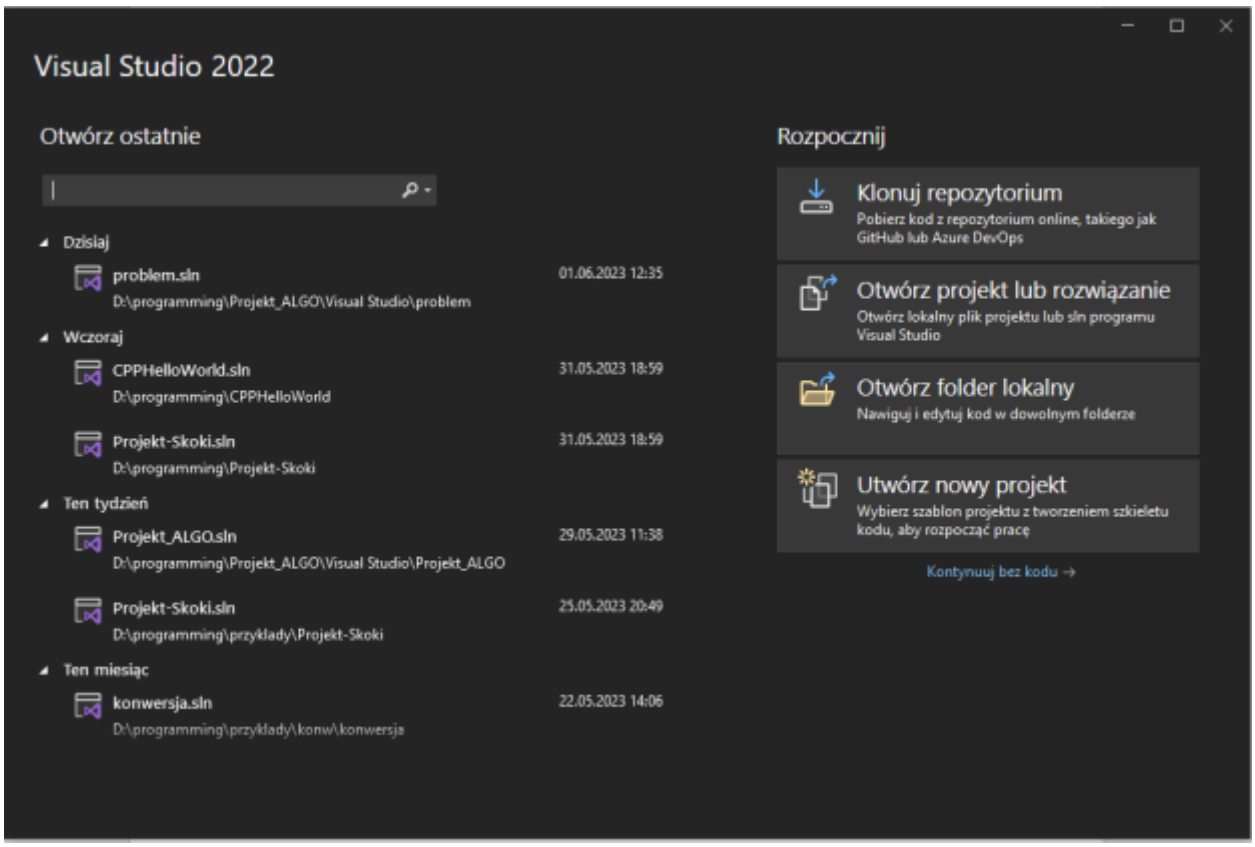
Oczywiście deweloper może stworzyć jeszcze więcej takich funkcji, ale musi mieć pewność, że nie złamie to optymalizacji kodu.

Instrukcja obsługi:

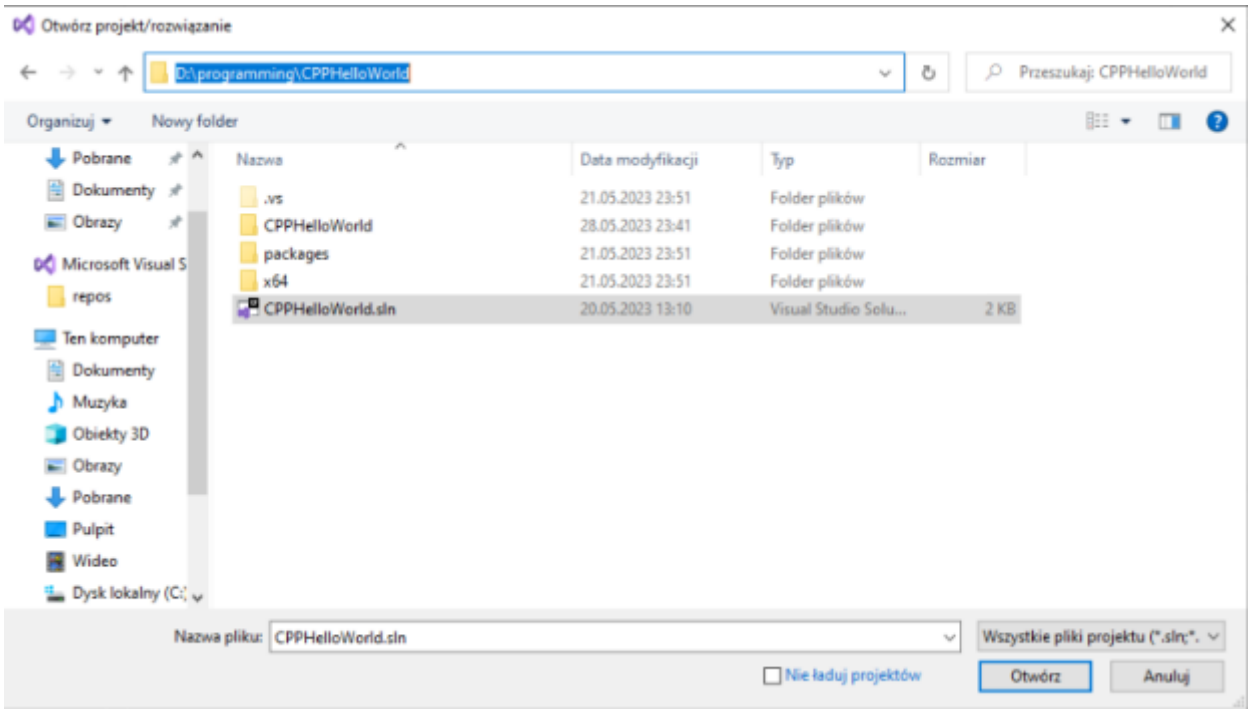
Po pobraniu projektu otrzymamy plik zip, który należy otworzyć i rozpakować na dysk.



Następnie otwieramy Microsoft Visual Studio, wybieramy zakładkę "Otwórz projekt lub rozwiązanie".

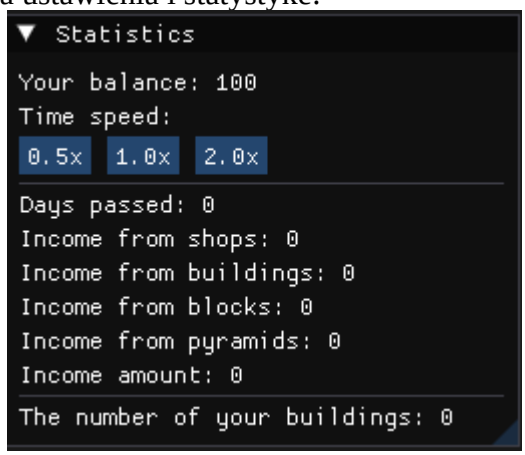


Otwieramy folder z projektem i szukamy pliku *.sln* jak na poniższym obrazku.

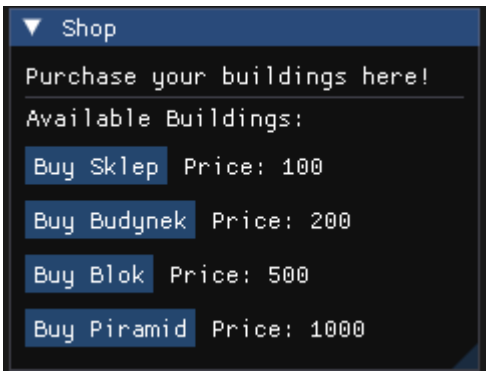


Następnie nasz projekt otworzy się w Visual Studio i jeśli wszystko pójdzie dobrze, zostanie pomyślnie uruchomiony.

Teraz widzimy naszą grę i nie wiemy, od czego zacząć, ponieważ jest tylko mapa i dwa okna. Pierwsze okno odpowiada za ustawienia i statystyke:

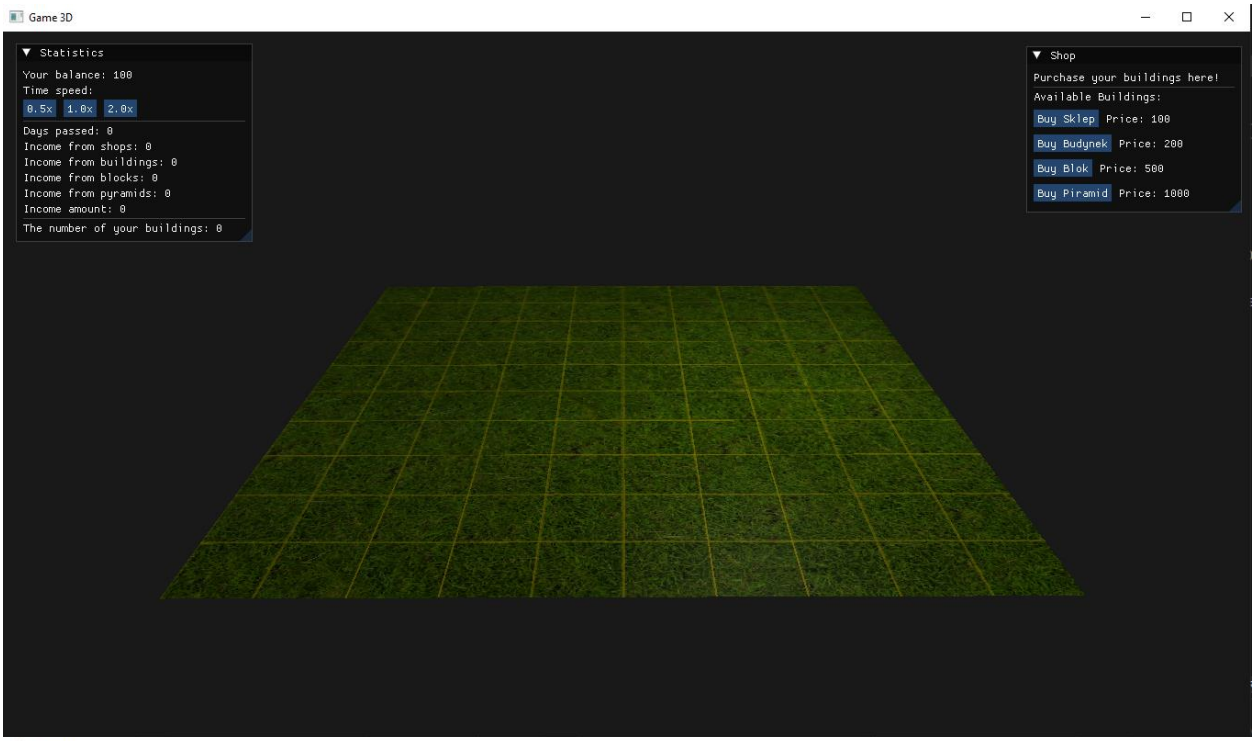


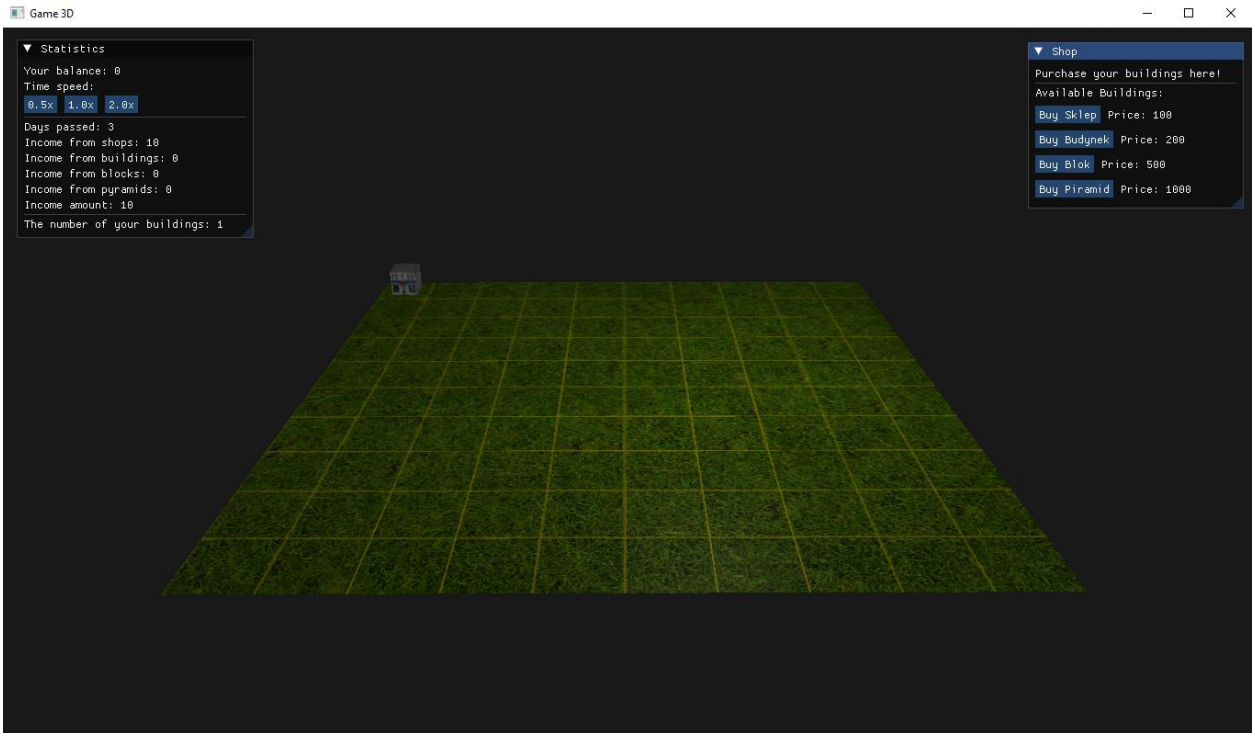
Tutaj możemy wybrać prędkość i zobaczyć, ile dochodu uzyskamy z budynków. Z drugiej strony, w drugim oknie:

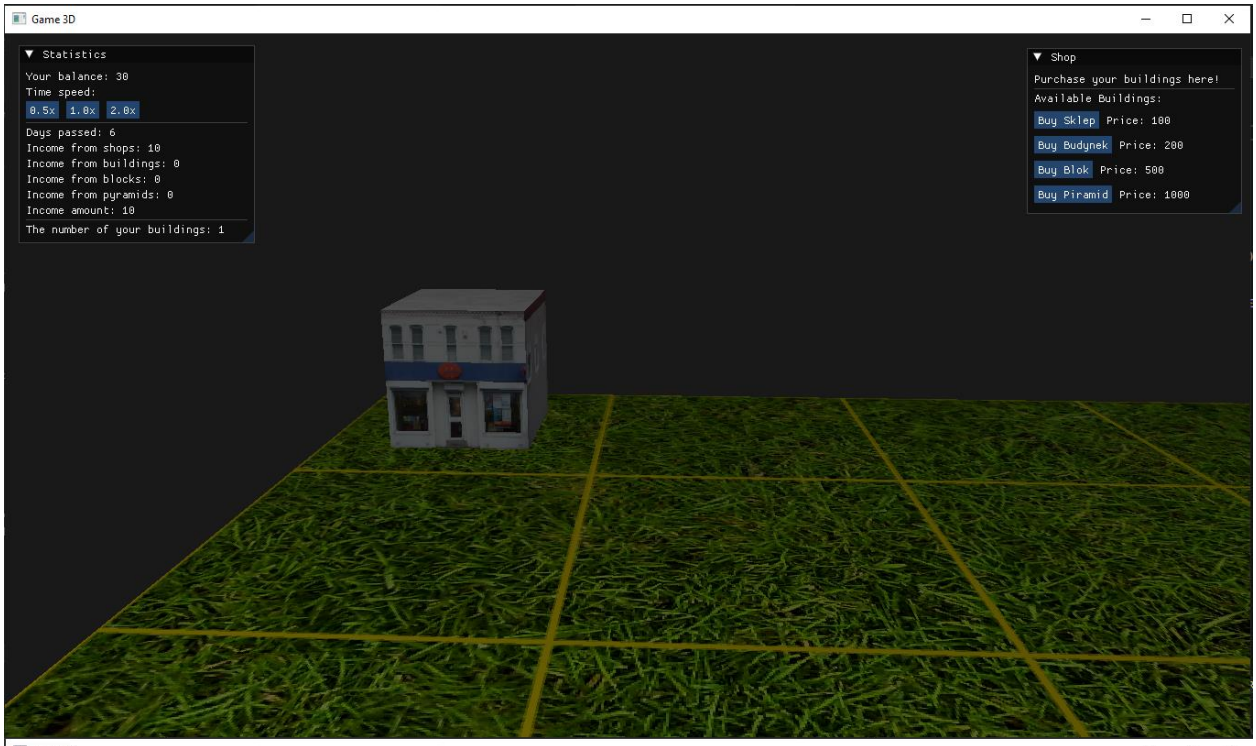


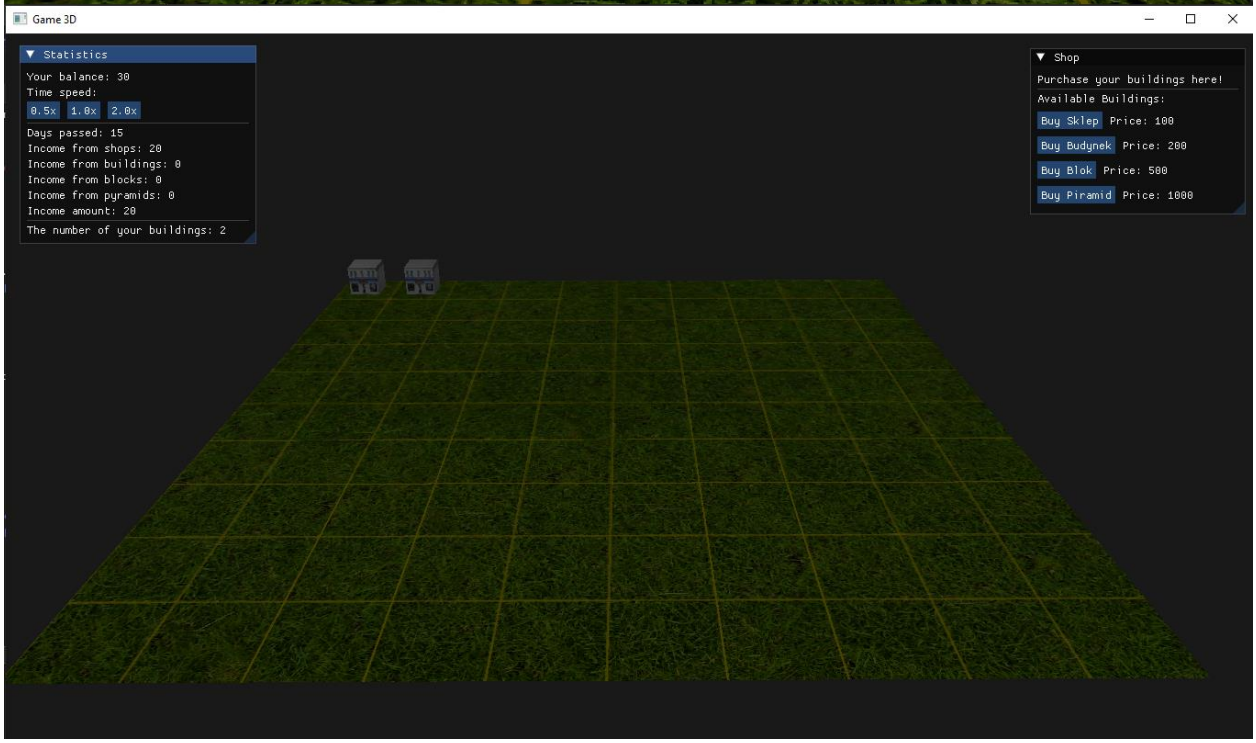
Widzimy tak zwany sklep budowlany. Tutaj możemy kupić wszystkie dostępne domy, a nawet piramidy.

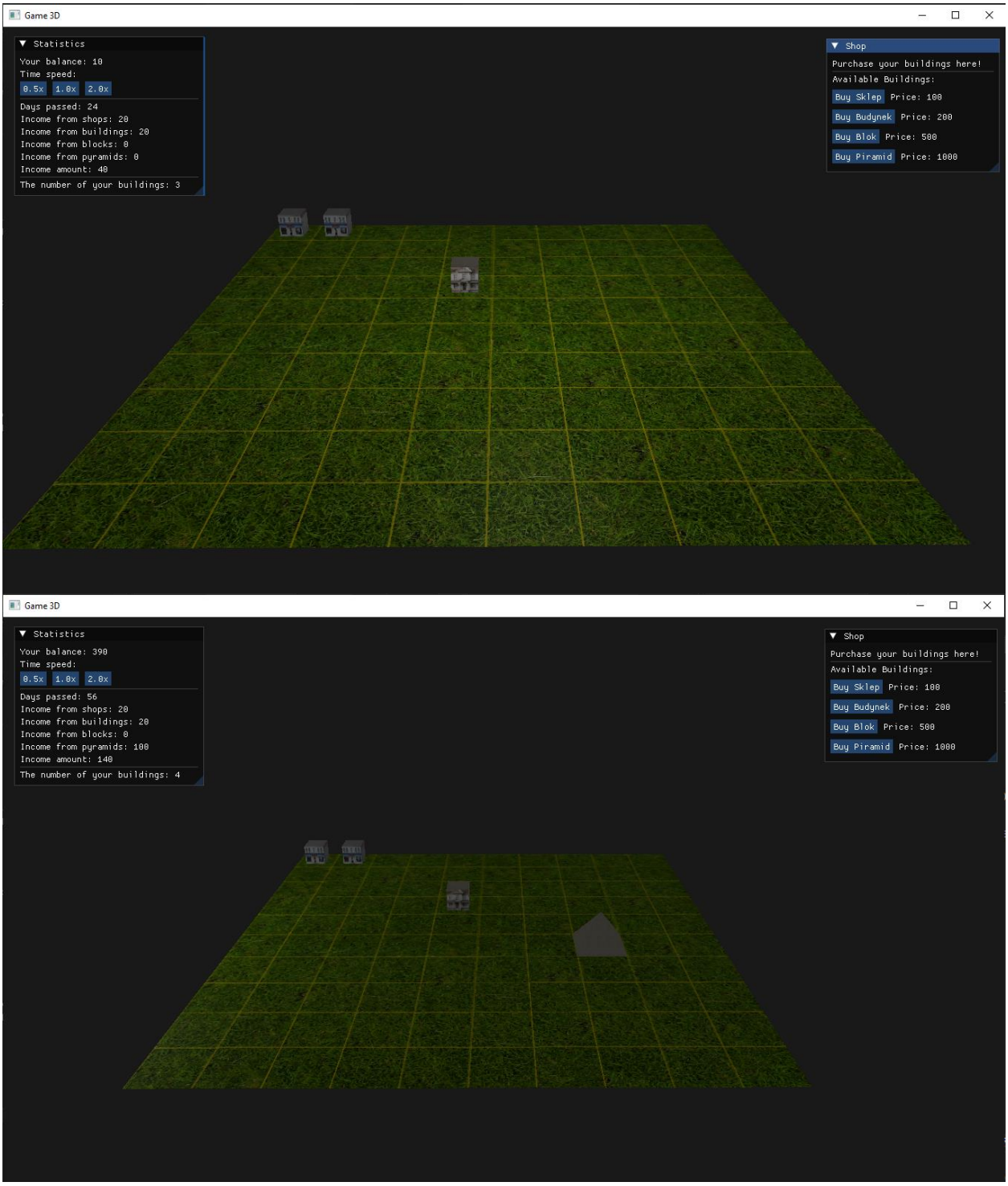
Przykłady działania ze zrzutami ekranu:

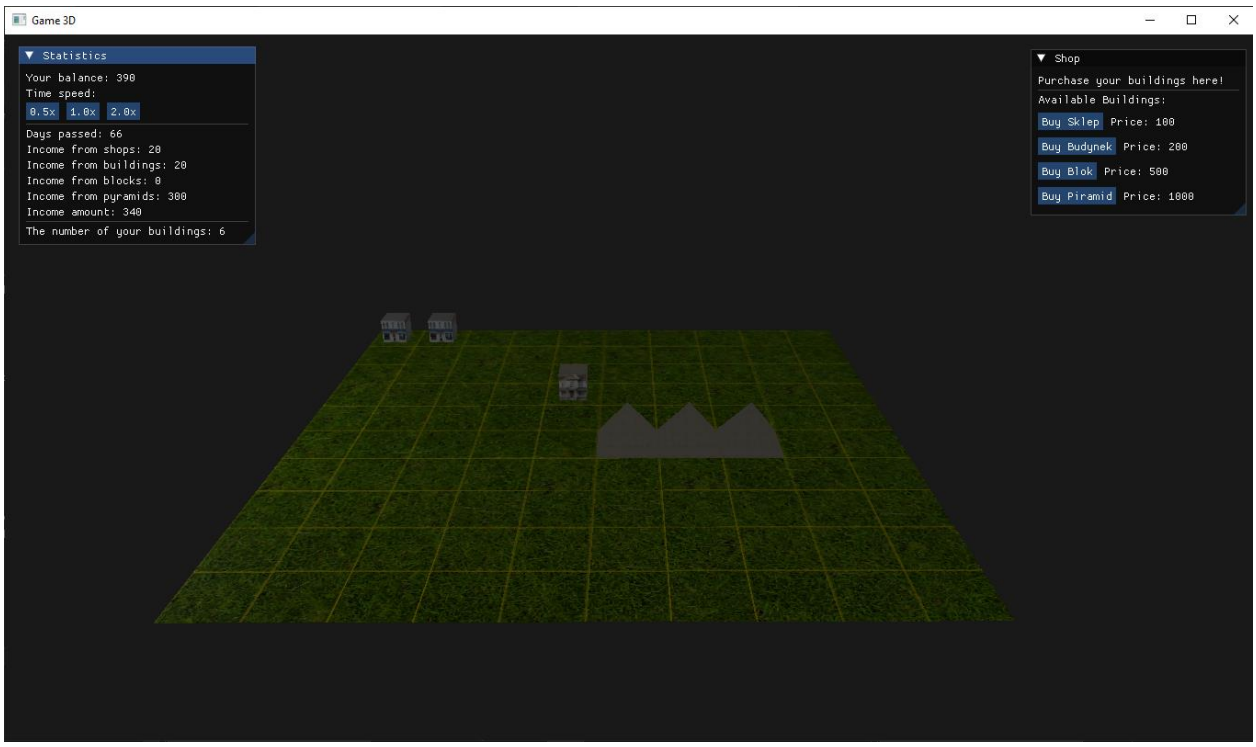





Game 3D
Statistics
Your balance: 30
Time speed:
0.5x 1.0x 2.0x
Days passed: 6
Income from shops: 10
Income from buildings: 0
Income from blocks: 0
Income from pyramids: 0
Income amount: 10
The number of your buildings: 1
Shop
Purchase your buildings here!
Available Buildings:
Buy Sklep Price: 100
Buy Budynek Price: 200
Buy Blok Price: 500
Buy Piramid Price: 1000


Game 3D
Statistics
Your balance: 30
Time speed:
0.5x 1.0x 2.0x
Days passed: 15
Income from shops: 20
Income from buildings: 0
Income from blocks: 0
Income from pyramids: 0
Income amount: 20
The number of your buildings: 2
Shop
Purchase your buildings here!
Available Buildings:
Buy Sklep Price: 100
Buy Budynek Price: 200
Buy Blok Price: 500
Buy Piramid Price: 1000


Game 3D
Statistics
Your balance: 10
Time speed:
0.5x 1.0x 2.0x
Days passed: 24
Income from shops: 20
Income from buildings: 20
Income from blocks: 0
Income from pyramids: 0
Income amount: 40
The number of your buildings: 3
Shop
Purchase your buildings here!
Available Buildings:
Buy Sklep Price: 100
Buy Budynek Price: 200
Buy Blok Price: 500
Buy Piramid Price: 1000
Game 3D
Statistics
Your balance: 390
Time speed:
0.5x 1.0x 2.0x
Days passed: 56
Income from shops: 20
Income from buildings: 20
Income from blocks: 0
Income from pyramids: 100
Income amount: 140
The number of your buildings: 4
Shop
Purchase your buildings here!
Available Buildings:
Buy Sklep Price: 100
Buy Budynek Price: 200
Buy Blok Price: 500
Buy Piramid Price: 1000



Podsumowanie:

W ramach naszego projektu udało nam się zrobić wszystko (zgodnie z wymaganiami), ale są też możliwe kierunki rozwoju projektu. Na przykład:

- Dodanie silnika dźwiękowego.
- Ulepszenie interfejsu(menu).
- Optymalizacja programu.

Takich zmian w projekcie można by wprowadzić znacznie więcej, ale główny cel gry (tworzenie swojego miasta) został osiągnięty.