Sprawozdanie

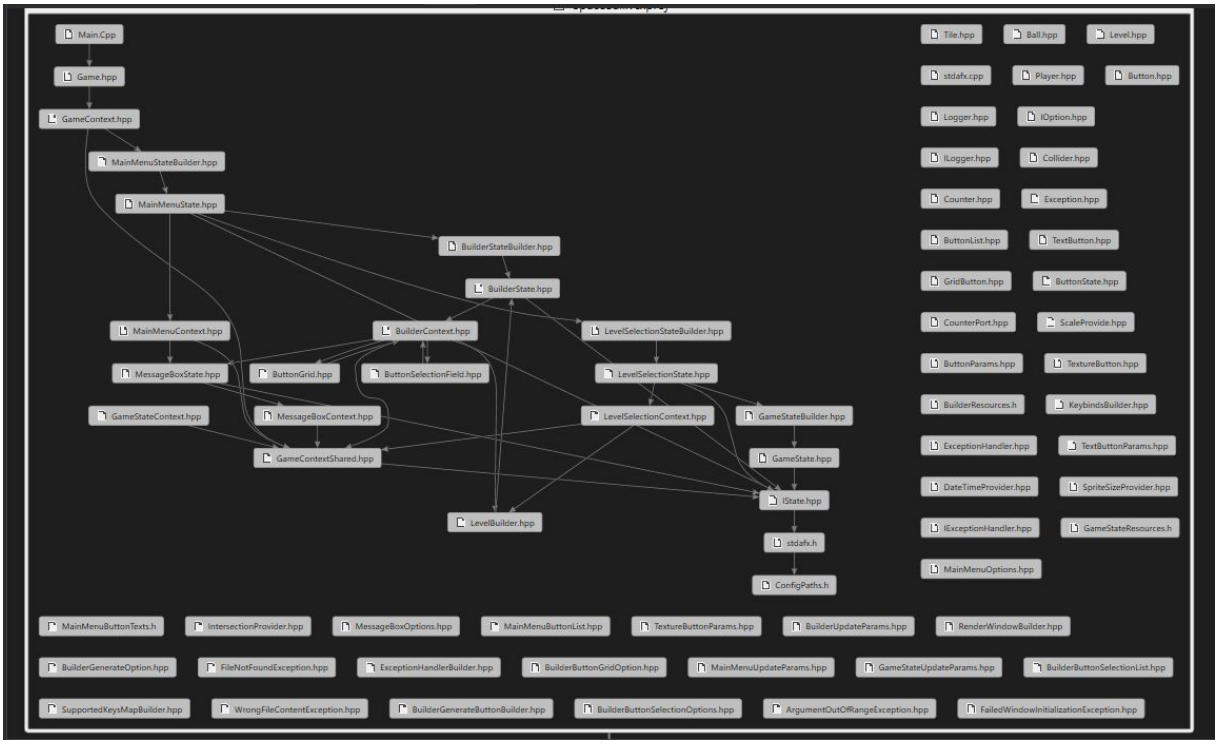
Projekt Space-Ball

Temat projektu	Gra typu arkanoid.
Autor	Aleksander Kijowski
Kierunek studiów	Informatyka
Rodzaj studiów	SSI
Semestr	IV
Prowadzący	Łukasz Dyga
GitHub	https://github.com/AleksanderKijowski/Space-Ball

Założenia projektu:

W ramach projektu powinna zostać zrealizowana w pełni funkcjonalna gra typu arkanoid napisana przy użyciu języka C++ i biblioteki graficznej SFML. Dodatkowo aplikacja powinna udostępniać narzędzie do własnoręcznego tworzenia poziomów .

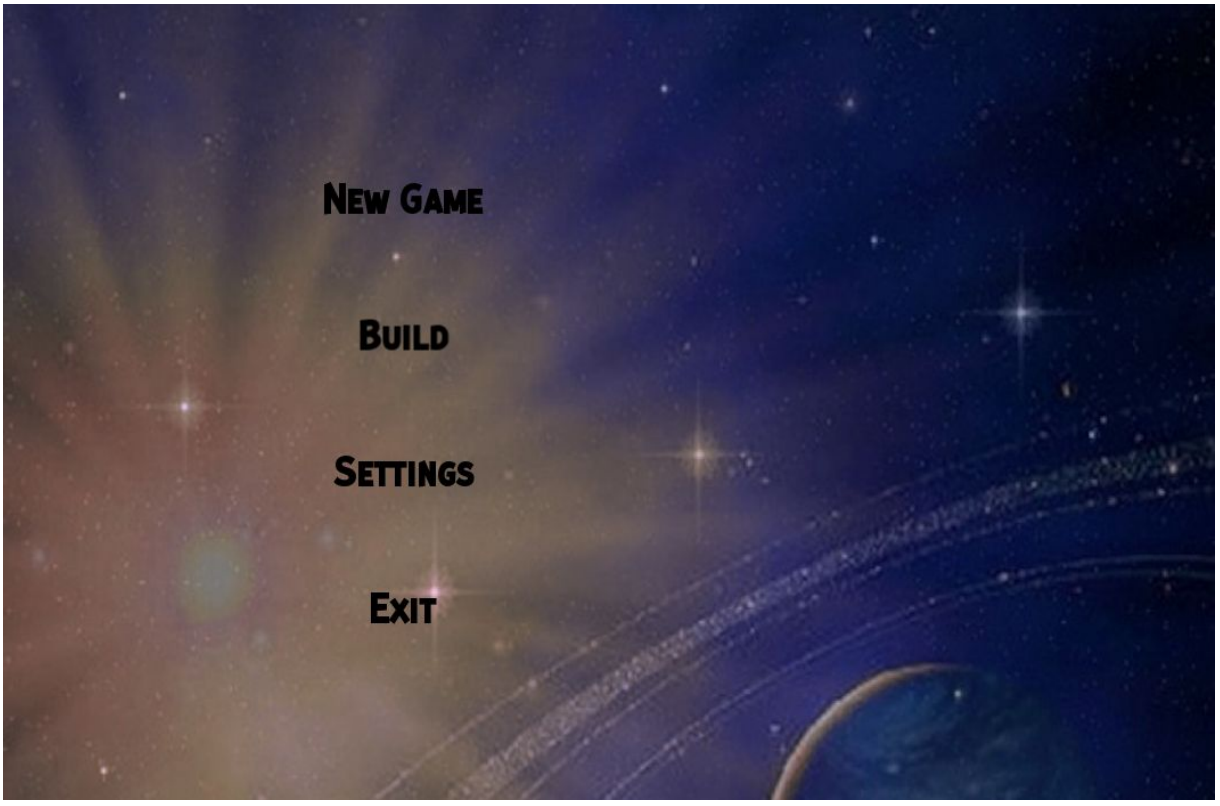
Diagram klas występujących w programie:



Rys. 1.1 Diagram klas.

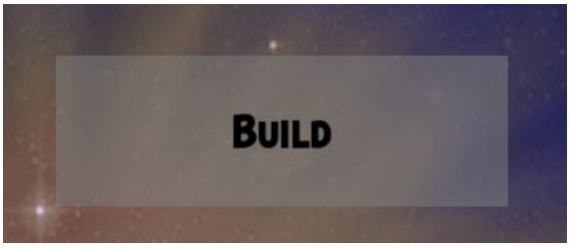
Korzystanie z aplikacji:

Aby skorzystać z aplikacji należy uruchomić plik .exe. Po jego uruchomieniu pokaże nam się widok menu głównego aplikacji. Wygląda ono w następujący sposób:



Rys. 1.2 Menu główne aplikacji.

Wybranie dowolnej opcji menu odbywa się poprzez naciśnięcie lewym klawiszem myszki po wcześniejszym najechaniu kursorem na odpowiedni przycisk (aplikacja sygnalizuję taki stan poprzez podświetlenie klawisza. Zostało to zaprezentowane poniżej:



Rys. 1.3 Prezentacja podświetlania klawiszy.

Aplikacja nie posiada predefiniowanych poziomów, więc aby pierwszy raz zagrać w grę najpierw należy zbudować poziom. Proces ten rozpoczynamy od wybrania pozycji „Build” z menu głównego.

Pokaże nam się wtedy następujący widok:

Rys. 1.4 Prezentacja selekcji bloku..

Należy poprzez kliknięcie wybrać któryś z klocków po lewej stronie.

Rys. 1.5 Prezentacja tworzenia własnego poziomu.

Teraz możemy tworzyć poziom, wybrany przez nas klocek stawiamy na odpowiedniej pozycji poprzez naciśnięcie lewego przycisku myszy na odpowiedniej pozycji w kratce po prawej stronie. Możemy usuwać postawione klocki za pomocą prawego klawisza myszy. Dodatkowo w każdej chwili jest możliwa zmiana wybranego klocka poprzez wybranie innego klocka z listy. Po zakończeniu tworzenia poziomu naciskamy klawisz „Generate”. Pokażę, nam się odpowiedni widok informujący o utworzeniu nowego poziomu i informujący o nadanej mu automatycznie nazwie. Przedstawiono go poniżej:

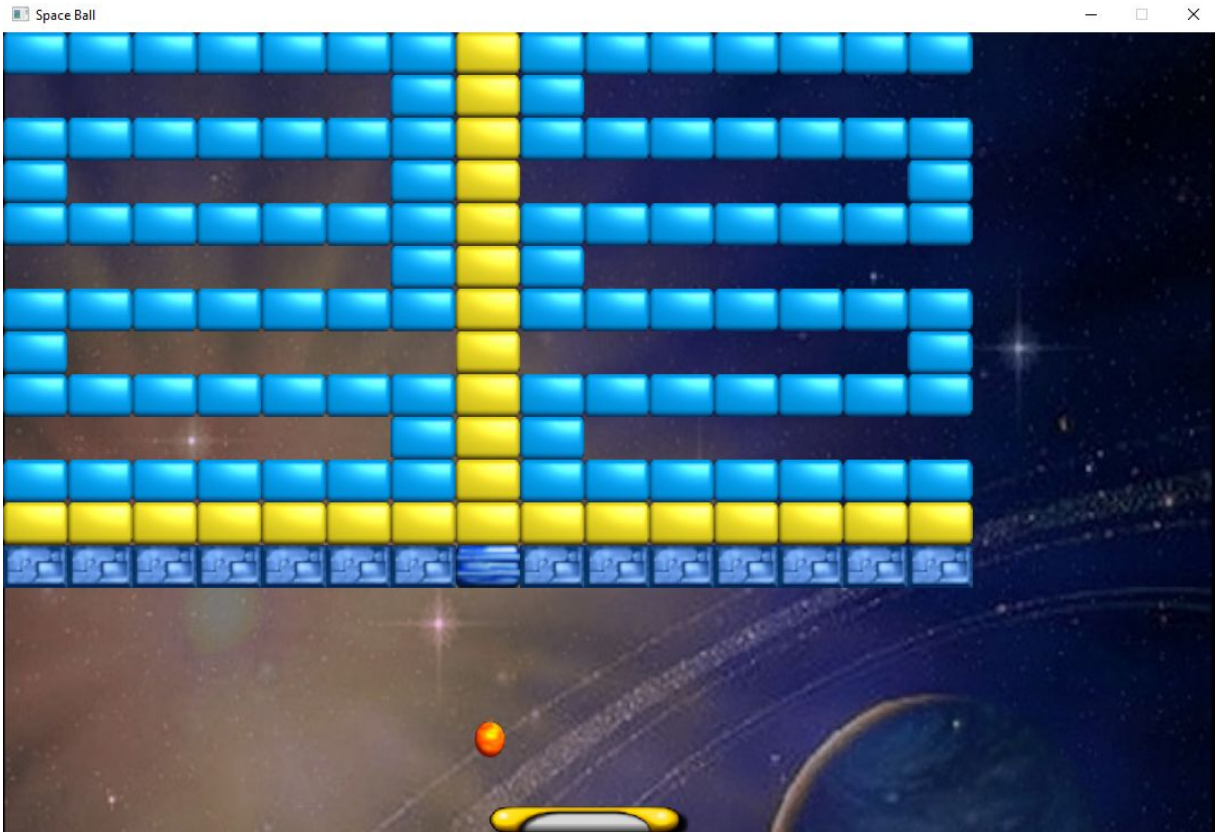
Rys. 1.6 Prezentacja okna potwierdzającego.

Potwierdzenia dokonujemy naciskając klawisz „OK”. Powracamy do menu głównego.

Teraz możemy rozpocząć nową grę. Naciskamy klawisz „New Game” i przechodzimy do widoku odpowiadającego za wybór poziomu.

Rys. 1.7 Prezentacja okna wyboru poziomu..

Wybieramy poziom w który chcemy zagrać korzystając z przycisków „-” i „+”. Potwierdzamy naciskając klawisz „Ok”. Następuje przekierowanie do gry:

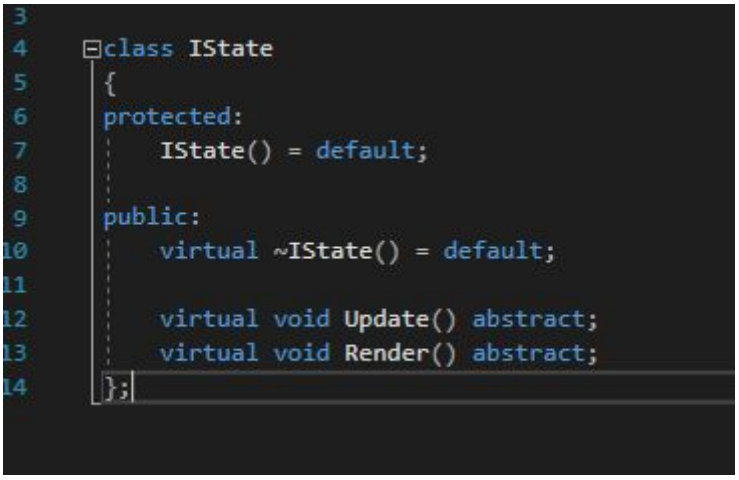


Rys. 1.8 Prezentacja gry.

Grę rozpoczynamy poprzez naciśnięcie klawisza „spacja”. Sterowanie graczem odbywa się za pomocą strzałek. Po zbiciu wszystkich klocków lub trzykrotnym upadku piłeczki pokaże nam się okno informujące odpowiednio o sukcesie/porażce.

Mechanizmy i kod wykorzystany przy tworzeniu gry:

Każda klasa widoku w grze implemetuję interfejs IState. Umożliwiło to stworzenie struktury widoków w której możemy wywołać metody Update i Render bez wiedzy który dokładnie widok zapewni ich implemetację (polimorfizm). Zdecydowałem się na trzymanie widoków w strukturze stosu, ponieważ zapewnia ona łatwe możliwości przechodzenia do poprzedniego, niezmienionego widoku.



```
3
4   class IState
5   {
6   protected:
7       IState() = default;
8
9   public:
10      virtual ~IState() = default;
11
12      virtual void Update() abstract;
13      virtual void Render() abstract;
14  };
```

Rys. 2.1 Klasa IState.

Konkretne widoki korzystają z konkretnych mechanizmów współdzielenia informacji, pozyskiwania danych itd. W aplikacji został również zaimplementowany system obsługi błędów. Każdy przewidziany wyjątek w aplikacji dziedziczy po klasie bazowej Exception.

```
#pragma once
#include "stdafx.h"

class Exception : public std::exception
{
private:
	string _message;

public:
	Exception(string message)
	{
		_message = message;
	}

	virtual string ToString()
	{
		return _message;
	}

	virtual List<string> GetLogMessage()
	{
		var result = List<string>();
		result.push_back(ToString());

		return result;
	}
};
```

Rys. 2.2 Klasa Exception.

Zapewniło to możliwość stworzenia następującej klasy odpowiedzialnej za obsługę wszystkich wyjątków w aplikacji.

ptionHandler.hpp

```cpp
#pragma once
#include "stdafx.h"
#include "IExceptionHandler.hpp"
#include "Logger.hpp"
#include "Exception.hpp"

class ExceptionHandler : public IExceptionHandler
{
public:
    ExceptionHandler()
        : IExceptionHandler(new Logger())
    {
    };

    void HandleException(Exception* exception) override
    {
        if (exception != nullptr)
        {
            _logger->Log(exception->GetLogMessage());
            delete exception;
            std::cerr << "Exception occured. See logs to for details." << std::endl;
        }
    }

    bool IsInitialized() const
    {
        return _logger->GetIsInitialized();
    }
};
```

Rys. 2.3 Klasa ExceptionHandler.

Powstał również system logowania wyjątków, który zapisuje wiadomość wyjątku do pliku tekstowego o nazwie odpowiadającej dacie.

```cpp
class Logger : public ILogger
{
public:
      Logger()
      {
            _isInitialized = true;
            Initialize();
      }

      ~Logger() = default;

      virtual void Log(List<string> text)
      {
            var path = ExceptionSavingDirectoryPath + "/" +
DateTime().Now().ToString() + ".txt";
            var file = std::ofstream(path, std::ios::out);

            if (file.is_open())
            {
                  for (var line : text)
                  {
                        file << line << std::endl;
                  }

                  file.close();
```

Rys. 2.4 Klasa Logger..

Udostępnianie danych pomiędzy komponentami zachodzi dzięki mechanizmowi portów. Skrótowy sposób działania przedstawię na przykładzie:

Widok menu główne posiada przycisk Exit który po naciśnięciu powinien sprawić, że wyjdziemy z aplikacji. Menu główne przy tworzeniu przycisku Exit udostępnia mu wskaźnik do portu na którym będzie się odbywała komunikacja. Menu główne sprawdza w każdym ticku aplikacji czy wartość portu nie uległa zmianie, jeżeli uległa (przycisk Exit został naciśnięty) to odpowienio modyfikuję własne działanie. Ponieważ marnotrawstwem byłoby używanie indywidualnych portów dla każdego z przycisków a z drugiej strony zachodzi potrzeba ich odróżnienia, initializator oprócz przekazania portu, przekazuję również wartość która powinna zostać ustawiona po aktywacji przycisku. Reprezentuję to klasa OptionValueObject.

```
class OptionValueObject
{
private:
    std::shared_ptr<IOption> Option;
    int Value;

public:
    OptionValueObject(std::shared_ptr<IOption> option, int value)
    {
        Option = option;
        Value = value;
    }

    void SetValue()
    {
        Option->SetValue(Value);
    }
};
```

Rys. 2.5 Klasa OptionValueObject..

Mechanizm wykrywania kolizji znajduję się w klasie Collider. Jest to dość obszerna klasa odpowiadająca za wykrywanie kolizji:

- piłka / klocek
- piłka / gracz
-

W przypadku wystąpienia kolizji z graczem bierze się również pod uwagę odległość od środka gracza w jakiej wystąpiła kolizja. Wyliczony współczynnik jest proporcjonalny do dodatkowej zmiany kąta odbicia:

```
	}

	float CalculateOffset(std::shared_ptr<Player> player,
std::shared_ptr<Ball> ball)
	{
		var circle = ball->GetHitbox();
		var radius = circle.getRadius();
		var rectangle = player->GetHitbox();

		var center = rectangle.getPosition().x +
rectangle.getSize().x / 2;

		return 50 * (circle.getPosition().x - center) /
rectangle.getSize().x / 2;
	}
```

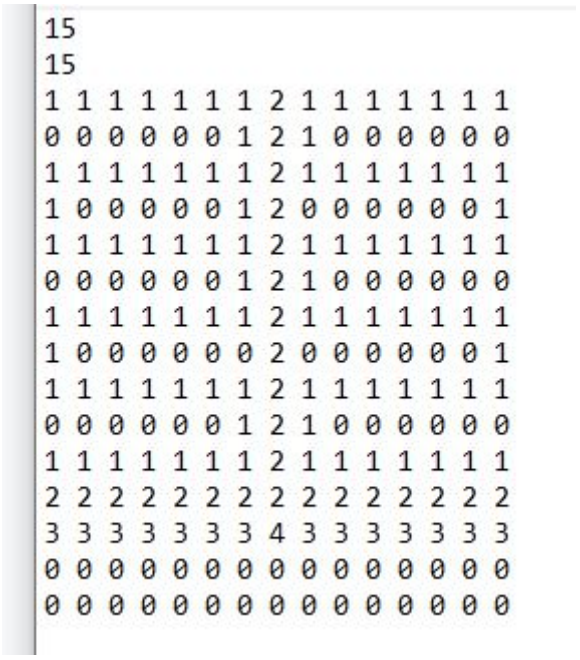
Rys. 2.7 Algorytm kalkulacji dodatkowego odchylenia kąta odbicia..

Algorytm wykrywania kolizji prezentuję się następująco:

```
private:
	bool intersects(const sf::CircleShape& c, const
sf::RectangleShape& r) {
		sf::FloatRect fr = r.getGlobalBounds();
		sf::Vector2f topLeft(fr.left, fr.top);
		sf::Vector2f topRight(fr.left + fr.width, fr.top);
		sf::Vector2f botLeft(fr.left, fr.top + fr.height);
		sf::Vector2f botRight(fr.left + fr.width, fr.top +
fr.height);

		return contains(c, topLeft) ||
			contains(c, topRight) ||
			contains(c, botLeft) ||
			contains(c, botRight);
	}

	bool contains(const sf::CircleShape& c, const
sf::Vector2f& p) {
		sf::Vector2f center = c.getPosition();
		float a = (p.x - center.x);
		float b = (p.y - center.y);
		a *= a;
		b *= b;
		float r = c.getRadius() * c.getRadius();

		return ((a + b) < r);
	}
};
```

Mechanizm zapisu i odczytu poziomu z pliku.

Najłatwiej jest go zrozumieć patrząc na plik przechowujący przykładowy poziom.



Rys. 2.9 Przykładowy plik tekstowy.

Teraz wszystko powinno stać się już jasne. Dwa pierwsze wiersze reprezentują rozmiar poziomu, gwarantuję to poprawne jego wczytanie. Kolejne liczby reprezentują id tekstur/ klocków których należy użyć przy ładowaniu poziomu.

Pozostałe algorytmy użyte w programie nie wymagają dodatkowego omówienia ze względu na ich prosty i czytelny sposób implementacji.