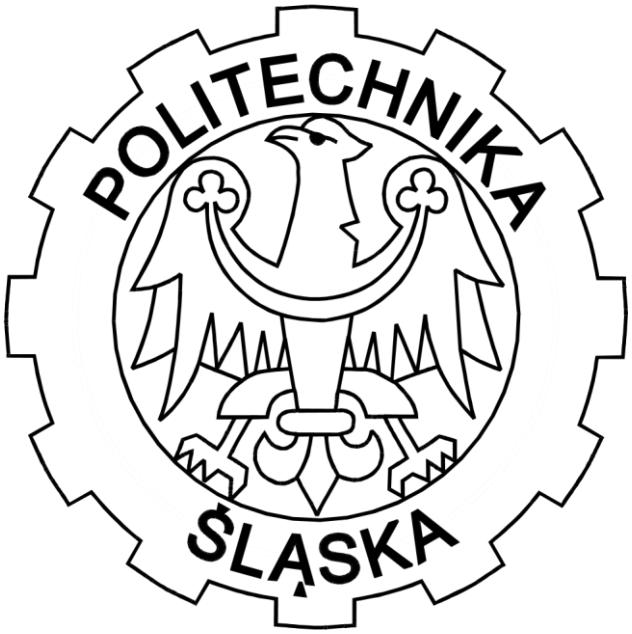


Grafika Ruchoma

Fake Tower Defence

Rok akademicki 2018/2019

Blok A

Sekcja II:

- Piotr Zuber
- Bartosz Czech
- Konrad Sladkowski
- Krzysztof Szwej

Spis treści

1.Wprowadzenie

1.1. Cel projektu

Projekt “Fake Tower Defence” ma na celu stworzenie gry na komputery stacjonarne działające pod systemem Windows. Jednym z głównych celów projektu jest stworzenie gry łatwej w rozbudowie, aby w przyszłości można było rozwijać projekt.

1.2. Założenia projektu

- Rozgrywka jest typu *tower defence*

- Mapy są generowane proceduralnie

- W grze występuje różnorodność przeciwników oraz wież

- Świat gry jest utrzymany w klimacie *science fiction*

1.3. Podział prac

Piotr Zuber – opracowanie i implementacja algorytmów proceduralnego generowania mapy oraz znajdowania ścieżki.

Konrad Sladkowski – implementacja logiki gry

Bartosz Czech – tworzenie tekstur, modeli 3D, animacji, efektów oraz interfejsu użytkownika

Krzysztof Szwej – implementacja kamery, jej obsługi oraz rysowanie mapy na podstawie tablicy

W trakcie trwania projektu, każdy z nas pomagał innym przy swojej pracy. W związku z tym, każdy miał częściowy wkład w pracę reszty sekcji.

1.4. Wymagania funkcjonalne

- Możliwość zapauzowania gry
- 4 typy wież
- 3 typy przeciwników
- 4 typy modeli przeszkód
- System gospodarowania punktami waluty przez gracza
- System proceduralnego generowania mapy i ścieżki dla przeciwników
- Autonomiczne poruszanie się przeciwników

1.5. Wymagania pozafunkcjonalne

- Płynność rozgrywki
- Trójwymiarowa grafika
- Intuicyjny interfejs

2. Dokumentacja użytkownika

2.1. O programie

Aplikacja *Fake Tower Defence* jest grą komputerową przeznaczoną na system Windows osadzoną w klimatach *science fiction*. Gra jest typu *tower* defence. Rozgrywka polega na uniemożliwieniu dotarcia przeciwnikom na koniec mapy za pomocą rozstawiania wież obronnych. Dzięki proceduralnemu generowaniu map, rozgrywka za każdym razem dostarcza innych doświadczeń. Wszystkie elementy są generowane w *3D,* co pozwala na uzyskanie większej immersji oraz głębsze doświadczanie rozgrywki.

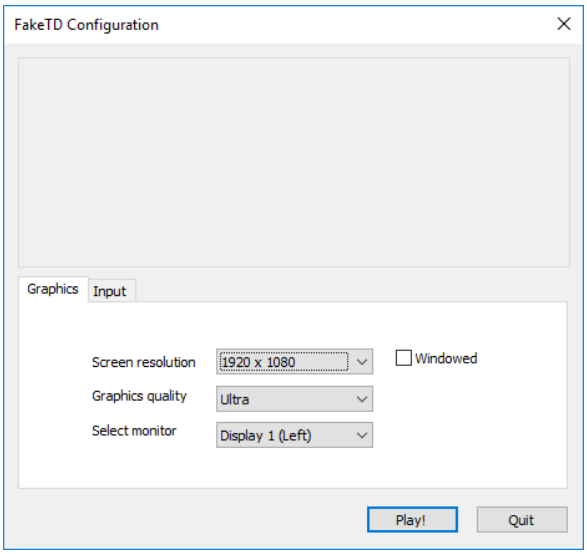
2.2. Wymagania techniczne

Zalecane wymagania sprzętowe:

- Windows 10
- 4 GM RAM
- Procesor o częstotliwości 2 Ghz
- Zintegrowana karta graficzna 1 GB
- 50 MB wolnego miejsca na dysku
- Klawiatura i mysz

2.3. Instalacja i uruchomienie

Instalacja polega na wypakowaniu dostarczonej paczki w dowolnym miejscu na dysku. Aby uruchomić grę należy dwukrotnie kliknąć ikonę FakeTD.exe i gra zostanie uruchomiona. Przy poprawnym uruchomieniu gry użytkownik powinien zobaczyć menu konfiguracyjne.



Rys.1. Menu konfiguracyjne

Aby rozpocząć grę należy wybrać odpowiednią rozdzielczość (zalecana to 1920x1018), jakość grafiki oraz monitor w wypadku, gdy użytkownik posiada więcej niż jeden. Aby uruchomić grę należy nacisnąć przycisk *Play!*. Po jego naciśnięciu użytkownik zobaczy następujące okno:

Rys.2. Menu gry

Aby przejść do rozgrywki należy nacisnąć przycisk *Play*.

2.4. Rozgrywka

Świat, który znamy został zaatakowany przez kosmitów. Aby nie dopuścić do inwazji, wcielamy się w kapitana Rakietę, który musi bronić przejścia do naszego świata. Gracz rozstawia na mapie wieże, które eliminują kolejne fale wrogów. Wraz z każdą falą rośnie poziom trudności. Fale robią się coraz większe a wrogowie zyskują więcej życia. Jedocześnie zestrzeliwując każdego z wrogów, zyskujemy fundusze, które można wydać na coraz lepsze wieże.

Nasze zmagania będą utrudnione przez różnorodne przeszkody umieszczone na mapie, które uniemożliwią stawianie wież obronnych na ich miejscu.

Każdy wróg, który przedostanie się przez naszą obronę, zabiera nam część punktów. Jeśli ich liczba spadnie do zera, przegramy a nasza ojczysta planeta pogrąży się w chaosie kosmicznej inwazji.

2.5. Sterowanie

Sterowanie kamerą:

W – góra

S – dół

A – lewo

D – prawo

Stawianie wież:

LPM – wybór wieży / stawianie wieży

Pauzowanie gry:

Esc – przeniesienie do menu pauzy

2.6. Instrukcja obsługi

Po wybraniu przycisku *„Play”* w głównym menu gry rozpocznie się rozgrywka.

Rys.3. Widok rozgrywki

1. Przyciski pozwalające na wybranie wieży.
2. Przycisk podświetlony na czerwono oznacza brak funduszy na daną wieżę.
3. W prawym górnym rogu przedstawiono informacje na temat numeru fali, zarobionych funduszy oraz punktów zdrowia.
4. Puste pole, na którym może zostać wybudowana wieża.
5. Podstawowa wieża, która została wybrana z menu kontekstowego i wybudowana na pustym polu.
6. Obiekt uniemożliwiający wybudowanie wieży.
7. Przeciwnik, który porusza się po wyznaczonej ścieżce.
8. Ścieżka, wzdłuż której należy budować wieże.
9. Przycisk pauzy, zatrzymujący rozgrywkę.

Rys.4. Widok pauzy

W momencie zatrzymania rozgrywki poprzez naciśnięcie klawisza **Esc** bądź naciśnięciu przycisku *„pause”* pojawi się następujące dwa przyciski – **Resume** oraz **Quit**. Pierwszy pozwoli na powrót do zatrzymanej wcześniej rozgrywki, natomiast drugi na wyjście z gry.

2.7. Obiekty w grze

2.7.1 Przeciwnicy

a) Czort - Podstawowy przeciwnik:

Najczęściej występujący w fali typ przeciwnika. Kosmiczne mięso armatnie. Proste do zniszczenia nawet najbardziej prymitywną bronią.

Statystyki:

-Punkty życia: 100

-Prędkość: 3

-Dochód za eliminację: 4

-Obrażenia zadawane bazie: 5

b) Szybki żbik – szybki przeciwnik:

Wyjątkowo szybki i zdradziecki najeźdźca z kosmosu. Pozornie łatwy do zabicia, może niespodziewanie przemknąć między wieżami i ominąć linię obrony.

Statystyki:

-Punkty życia: 40

-Prędkość: 8

-Dochód za eliminację: 2.5

-Obrażenia zadawane bazie: 2.5

c) Twardziel – wytrzymały przeciwnik:

Wielki i groźny. Twardy i wytrzymały. Największy postrach drogi mlecznej od czasu Huna Atylli. Potrzeba będzie niebywałych nakładów amunicji, żeby go powalić.

Statystyki:

-Punkty życia: 160

-Prędkość: 2.5

-Dochód za eliminację: 6

-Obrażenia zadawane bazie: 10

2.7.2 Wieże

a) Fuszerka – wieża podstawowa

Najtańsza, najbardziej pospolita. Powszechnie budowana na peryferiach kosmosu przez astro-chłopów w celu obrony przed pomniejszymi obcymi. Z całą pewnością nada się do zniszczenia pierwszych fal zastępów wroga.

Statystyki:

-Obrażenia: 50

-Prędkość przeładowania: 7

-Zasięg: 3

b) Hermes – wieża szybka

Zaprojektowanie i skonstruowanie kartaczownicy w 1861 roku przez Richarda Gatlinga miało niewątpliwy wpływ na przebieg wielu konfliktów w historii ludzkości. Teraz kosmici poczują jego moc. Wieża miota pociski w niewielkich odstępach czasu, niestety kosztem mniejszych obrażeń.

Statystyki:

-Obrażenia: 7

-Prędkość przeładowania: 20

-Zasięg: 4

c) Młot Thora – wieża ciężka

Duchowy spadkobierca Kanone 5 Kruppa. Działa w myśl zasady, „jeśli rozwiązania siłowe nie przynoszą skutku – prawdopodobnie używasz zbyt mało siły”. Teraz używasz jej zbyt dużo.

Statystyki:

-Obrażenia: 100

-Prędkość przeładowania: 2

-Zasięg: 5

d) Długa lufa – wieża dalekiego zasięgu

Wieża dalekiego zasięgu. Po opracowaniu technologii plazmowej, działa orbitalne starego typu zaczęto powszechnie przerabiać na wieżyczki obronne. W praktyce zasięg pocisku ograniczony jest pierwszą przeszkodą, na jaką natrafi.

Statystyki:

-Obrażenia: 60

-Prędkość przeładowania: 1.5

-Zasięg: 10

3. Dokumentacja techniczna

3.1. Wybrane narzędzia

3.1.1. Technologia – język C#

Język C# został dobrany ze względu największą styczność z nim w porównaniu do pozostałych poznanych do tej pory języków programowania. Wspomniany język jest silnie zorientowany obiektowo oraz pozwala niewielkim kosztem wydajnościowym na szybkie tworzenie aplikacji. Jest on również kompatybilny z wybranym przez nas silnikiem gry.

3.1.2. Środowisko programistyczne – Microsoft Visual Studio 2017

Ze środowiskiem Visual Studio sekcja miała przyjemność pracować od początku nauki programowania. Ponadto oferuje wsparcie dla Unity oraz GITa.

3.1.3. System kontroli wersji – GIT

System kontroli wersji znajduje swoje zastosowanie w każdym komercyjnym projekcie programistycznym. Pozwala na równoległą pracę wielu programistów, zapewnia najnowszej wersji aplikacji, śledzenie zmian kodu, integrację zmian dokonywanych przez programistów oraz wspomaga rozwiązywanie konfliktów, które mogą wystąpić na skutek pracy wielu programistów nad jednym plikiem.
Ze względu na duże doświadczenie z system kontroli wersji GIT, właśnie ten system został wybrany. Wykorzystano repozytorium oferowane przez serwis GitHub, oferujący bezpłatne publiczne repozytoria.

3.1.4. Silnik Unity

Unity jest zintegrowanym środowiskiem do tworzenia gier komputerowych oraz innych aplikacji interaktywnych w 2D I 3D. Wybrano to środowisko ze względu na dużą ilość dostępnych materiałów do nauki w formie poradników czy dobrze napisanej dokumentacji. Członkowie sekcji chcieli również nauczyć się obsługi tego środowiska.

3.1.5. Oprogramowanie do modelowania – Blender

Blender jest wolnym i otwartym oprogramowaniem do modelowania i mapowania obrazów oraz animacji trójwymiarowych o niekonwencjonalnym interfejsie użytkownika. Oprogramowanie zostało wybrane ze względu na jego możliwości oraz darmowy dostęp.

3.1.6. Oprogramowanie do obróbki graficznej – Paint i Gimp

Oba narzędzia pozwalają na tworzenie grafik, tekstur oraz do ich edycji. Zostały one wybrane, tak jak w przypadku programu Blender ze względów funkcjonalnych oraz by ograniczyć koszty produkcji.

3.2. Podział aplikacji

3.2.1. Algorytm proceduralnego generowania mapy

Stworzony algorytm generuje nierówności terenu na podstawie szumu Perlina. Szum został stworzony w roku 1983 przez Perlina, jako rezultat frustracji spowodowanej „maszyno-podobnym" wyglądem ówczesnej grafik komputerowej oraz w czasie jego prac nad filmem Tron. Wyniki swojej pracy Perlin opublikował w 1985 roku. W 1997 roku autor algorytmu otrzymał za swoją pracę Oscara w kategorii technicznej.

Szum jest generowany w klasie, która posiada metodę GetNoise. Została ona zaimplementowana w sposób, pokazany poniżej:

```
public int GetNoise(int x, int y, int range)
    {
        int chunkSize = 16;

        float noise = 0;

        range /= 2;

        while (chunkSize > 0)
        {
            int index_x = x / chunkSize;
            int index_y = y / chunkSize;

            float t_x = (x % chunkSize) / (chunkSize * 1f);
            float t_y = (y % chunkSize) / (chunkSize * 1f);

            float r_00 = random(index_x, index_y, range);
            float r_01 = random(index_x, index_y + 1, range);
            float r_10 = random(index_x + 1, index_y, range);
```

```
            float r_11 = random(index_x + 1, index_y + 1, range);

            float r_0 = lerp(r_00, r_01, t_y);
            float r_1 = lerp(r_10, r_11, t_y);

            noise += lerp(r_0, r_1, t_x);

            chunkSize /= 2;
            range /= 2;

            range = Mathf.Max(1, range);
        }

        return (int)Mathf.Round(noise);
    }
```

Generowanie mapy jest zrealizowane za pomocą algorytmu zamieszczonego poniżej. Przyjmuje on prawdopodobieństwo wystąpienia przeszkód, wektor startowy oraz końcowy ścieżki. Na podstawie szumu generuje wysokość macierzy w danym miejscu oraz ustawia, jeśli wylosowano przeszkodę, ustawia ją na czubku. Następnie szuka ścieżki za pomocą algorytmu A* i jeśli nie uda mu się tego zrobić, ponawia generowanie mapy. Po udanym wyznaczeniu ścieżki, w zależności od flagi *TrimTheMap*, boki mapy są przycinane.

```
private Terrain[,] GenerateMapMatrix(float obstaclePropabilityPercent, out Vector2 startPosition, out
Vector2 endPosition)
    {
        Terrain[,] map = new Terrain[_maxX, _maxZ];
        bool needToRegenerate = false;
        do
        {
            needToRegenerate = false;
            for (int i = _minX; i < _maxX; i++)

                for (int k = _minZ; k < _maxZ; k++)
                {
                        int columnHeight = 2 + noise.GetNoise(i - _minX, k - _minZ, _maxY - _minY -
                        2);
                    var terrainType = RandomTerrainType(obstaclePropabilityPercent);
                    if (terrainType != TerrainType.Normal)
                    {
                        map[i, k] = new Terrain(terrainType, columnHeight + 1);
                    }
                    else
                    {
                        map[i, k] = new Terrain(terrainType, columnHeight);
                    }
                }
            }

            Vector2 start, end;
            findStartAndEndOfMap(map, out start, out end);

            startPosition = start;
            endPosition = end;
            var pathFinder = new Pathfinder(map, _maxX, _maxZ);
            try
            {
                map = pathFinder.SearchPath(start, end);
                pathTilesList = new List<Vector2>(pathFinder.pathTiles);
            }
            catch(NoWayException)
            {
                needToRegenerate = true;
            }
        } while (needToRegenerate);
```

```
        if (TrimTheMap)
        {
            map = trimTheMap(map);
        }
        return map;
    }
```

3.2.2. Algorytm znajdowania ścieżki

Wyszukiwanie ścieżki zostało zrealizowane za pomocą algorytmu A*. Jest to algorytm heurystyczny, który dzięki temu, że jest zupełny, zawsze znajduje ścieżkę o ile taka istnieje. Jest to jednocześnie najkrótsza możliwa ścieżka. Został on wybrany, dlatego że podczas działania przeszukuje mniej węzłów niż inne algorytmy z taką samą heurystyką, dzięki czemu skrócony zostaje czas generowania ścieżki na mapie, na której poruszają się potwory. Poniżej został zamieszczony algorytm, wykorzystany w grze. Jest on napisany w języku angielskim, który był wykorzystywany podczas programowania projektu.

```
public Terrain[,] SearchPath(Vector2 startTile, Vector2 endTile)
    {
        this.startTile = startTile;
        this.endTile = endTile;

        //Reset all the values
        for (int i = 0; i < gridWidth; i++)
        {
            for (int j = 0; j < gridHeight; j++)
            {
                grid[i, j].cost = 0;
                grid[i, j].heuristic = 0;
            }
        }

        #region Path validation
        bool canSearch = true;

        if (grid[(int)startTile.x, (int)startTile.y].TerrainType!=TerrainType.Normal)
        {
            throw new WrongStartPointException();
        }
        if (grid[(int)endTile.x, (int)endTile.y].TerrainType != TerrainType.Normal)
        {
            throw new WrongEndPointException();
        }
        #endregion
        bool canFindWay = false;
        //Start the A* algorithm

        //add the starting tile to the open list
         openList.Add(startTile);
        currentTile = new Vector2(-1, -1);

        //while Openlist is not empty
        while (openList.Count != 0)
        {
            //current node = node from open list with the lowest cost
            currentTile = GetTileWithLowestTotal(openList);

            //If the currentTile is the endtile, then we can stop searching
            if (currentTile.x == endTile.x && currentTile.y == endTile.y)
            {
                canFindWay = true;
                break;
            }
            else
            {
                        //move the current tile to the closed list and remove it from the open list
```

```
                openList.Remove(currentTile);
                closedList.Add(currentTile);

                //Get all the adjacent Tiles
                List<Vector2> adjacentTiles = GetAdjacentTiles(currentTile);

                foreach (Vector2 adjacentTile in adjacentTiles)
                {
                    //adjacent tile can not be in the open list
                    if (!openList.Contains(adjacentTile))
                    {
                        //adjacent tile can not be in the closed list
                        if (!closedList.Contains(adjacentTile))
                        {
                            //move it to the open list and calculate cost
                            openList.Add(adjacentTile);

                                 Terrain tile = grid[(int)adjacentTile.x, (int)adjacentTile.y];

                            //Calculate the cost
                                tile.cost = grid[(int)currentTile.x, (int)currentTile.y].cost + 1;

                            //Calculate the manhattan distance
                            tile.heuristic = ManhattanDistance(adjacentTile);

                            //calculate the total amount
                            tile.total = tile.cost + tile.heuristic;
                        }
                    }
                }
            }
        }
         if (!canFindWay)
        {
            throw new NoWayException("Need to regenerate map");
        }

        //Show the path
        ShowPath();
        grid[(int)startTile.x, (int)startTile.y].Height = 1;
        grid[(int)startTile.x, (int)startTile.y].TerrainType = TerrainType.Path;

        grid[(int)endTile.x, (int)endTile.y].Height = 1;
        grid[(int)endTile.x, (int)endTile.y].TerrainType = TerrainType.Path;
        return grid;
    }
```

W skrócie można opisać działanie tego algorytmu w pseudokodzie:

Dodaj startowy węzeł do otwartej listy;

Dopóki otwarta lista nie jest pusta

{

Aktualny węzeł = węzeł z otwartej listy z najniższym kosztem;

Jeśli aktualny węzeł jest celem

Ścieżka zakończona;

w przeciwnym wypadku

Przenieś aktualny węzeł do zamkniętej listy

Dla każdego sąsiadującego węzła:

Jeśli leży na polu i nie jest przeszkodą i nie ma go na otwartej ani zamkniętej liście:

Przenieś go do otwartej listy i policz koszt;

}

3.2.3. Rysowanie mapy na podstawie macierzy

Mapa wygenerowana uprzednio ma postać tablicy trzywymiarowej. Za wyrysowanie jej na ekranie odpowiada funkcja `GenerateMapFromMatrix(Terrain[,] map).` Funkcja iteruje po kolejnych polach tablicy rysując kolumny sześcianów terenu.

```
private void GenerateMapFromMatrix(Terrain[,] map)
    {
        float width = dirtPrefab.transform.lossyScale.x;
        float height = dirtPrefab.transform.lossyScale.y;
        float depth = dirtPrefab.transform.lossyScale.z;

        for (int i = _minX; i < _maxX; i++)
        {//columns (x values)
            for (int k = _minZ; k < _maxZ; k++)
            {
                int columnHeight = map[i, k].Height;
                for (int j = _minY; j < _minY + columnHeight; j++)
                {//rows (y values)
                    if (j == columnHeight - 1) //item on top
                    {
                        switch (map[i, k].TerrainType)
                        {
                            case TerrainType.TrashBin:
                                {
                                    GameObject block = trashBinPrefab;
                                    Instantiate(block, new Vector3(i * width, j * height, k * depth),
Quaternion.identity);
                                }
                                break;
                            case TerrainType.BlackHole:
                                {
                                    GameObject block = blackHolePrefab;
                                    Instantiate(block, new Vector3(i * width, j * height, k * depth),
Quaternion.identity);
                                }
                                break;
                            case TerrainType.DestroyedTurret:
                                {
                                    GameObject block = destroyedTurretprefab;
                                    Instantiate(block, new Vector3(i * width, j * height, k * depth),
Quaternion.identity);
                                }
                                break;
                            case TerrainType.Cristal:
                                {
                                    GameObject block = cristalPrefab;
                                    Instantiate(block, new Vector3(i * width, j * height, k * depth),
Quaternion.identity);
                                }
                                break;
                            case TerrainType.Path:
                                {
                                    GameObject block = pathPrefab;
                                    Instantiate(block, new Vector3(i * width, j * height, k * depth),
Quaternion.identity);
                                }
                                break;
                            default:
                                {
                                    GameObject block = normalTerrainPrefab;
                                    Instantiate(block, new Vector3(i * width, j * height, k * depth),
Quaternion.identity);
                                }
                                break;
                        }
                    }
```

```
                    else
                    {
                        GameObject block = normalTerrainPrefab;
                        Instantiate(block, new Vector3(i * width, j * height, k * depth),
Quaternion.identity);
                    }

                }
            }
        }
    }
```

3.2.4. Obsługa kamery

W grze została zaimplementowana kamera z rzutu izometrycznego, sterowana przyciskami W, S, A, D. Możliwość przesuwania kamery jest ograniczona poprzez wielkość mapy.

Głównym problemem, który napotkaliśmy podczas tworzenia kamery było niewłaściwe ustawienie kamery względem osi X, Y, Z. Problem ten udało się rozwiązać dzięki umieszczeniu kamery w zewnętrznym obiekcie, którego osie były równoległe do osi sceny.

```
void Update()
    {
        if (Input.GetKeyDown(KeyCode.W))
        {
            dir = UP;
        }
        else if (Input.GetKeyDown(KeyCode.S))
        {
            dir = DOWN;
        }
        else if (Input.GetKeyDown(KeyCode.A))
        {
            dir = LEFT;

        }
        else if (Input.GetKeyDown(KeyCode.D))
        {
            dir = RIGHT;
        }
        else if (Input.GetKeyUp(KeyCode.W) | Input.GetKeyUp(KeyCode.S) | Input.GetKeyUp(KeyCode.A) |
Input.GetKeyUp(KeyCode.D))
        {
            dir = 0;
        }

        if (Input.GetKeyDown(KeyCode.Q))
        {
            rot = RLEFT;
        }
        else if (Input.GetKeyDown(KeyCode.E))
        {
            rot = RRIGHT;
        }
        else if (Input.GetKeyUp(KeyCode.Q) | Input.GetKeyUp(KeyCode.E))
        {
            rot = 0;
        }

        if (dir == UP)
        {
            if (Target.transform.position.z < 0 )
                Target.transform.Translate(0, 0, 10 * Time.deltaTime);
        }
        else if (dir == DOWN)
        {
            if (Target.transform.position.z > -gene.rows)
                Target.transform.Translate(0, 0, -10 * Time.deltaTime);
        }
```

```
        else if (dir == LEFT)
        {
            if (Target.transform.position.x < 0 )
                return;
            Target.transform.Translate(-10 * Time.deltaTime, 0, 0);
        }
        else if (dir == RIGHT)
        {
            if (Target.transform.position.x > gene.cols )
                return;
            Target.transform.Translate(10 * Time.deltaTime, 0, 0);
        }
    }
```

3.2.5. Logika gry

Logikę gry, przez co rozumie się elementy takie jak: strzelnie i obracanie wież, poruszanie się przeciwników (zwanych potocznie tutaj „agentami“), wywoływanie odpowiednich reakcji na klikanie przycisków, w znacznej mierze spaja klasa „GameLogic“. W niej zawarto odpowiednie listy obiektów przechowujące informacje o liczbie, położeniu, zachowaniu oraz modelach poszczególnych wież i agentów.

Agenci, pociski oraz wieże - każdy z tych elementów posiada własną klasę dziedziczącą po klasie MonoBehaviour, zawierającą metody, oraz pola pozwalające na odwzorowanie tego elementu w grze. Dziedziczenie po MonoBehaviour umożliwia poprawne przyłączenie klasy jako komponentu lub skryptu do obiektów „GameObject“ będących podstawowymi obiektami silnika „Unity“. Klasy te posiadają prywatne konstruktory domyślne, ze względu na zastosowany mechanizm w silniku „Unity“, który nie pozwala na bezpośrednie używanie słów kluczowych „new“. Wymusza natomiast dodawanie takowych elementów poprzez zastosowanie metody ‘AddComponent<Typ_obiektu>()’. Wobec tego klasy posiadają zaimplementowane metody typu void ‘Initalize() ‘, które pełnią rolę swoistych konstruktorów.

Tworzenie wież, agentów oraz pocisków odbywa się poprzez klonowanie odpowiednich modeli, które są przekazywana w wymienionej wyżej metodzie Initalize. Za pomocą enumeracji, określane są parametry każdego obiektu, tj. czy agent będzie „tankiem”, czyli idącym wolno i posiadającym dużą ilość zdrowia przeciwnikiem czy „normalnym” posiadającym bardziej zbalansowane parametry.

Klasa **GameLogic** realizuje również zadania związane z wyświetlaniem użytkownikowi komunikatów w grze, takich jak: liczba punktów zdrowia bazy, ilosc posiadanej waluty służącej do budowania wież,numerze akutalnej fali agentów, czy też komunikatów o przegranej lub wygranej. Pondadto tworzy ‘fale’ agentów. Oznacza to,że są tutaj wywoływane metody tworzące nowych agentów oraz obsługujące ich ruch w kierunku końca scieżki.

Obsługę namierzania agentów zrealizowano również w tej klasie w następujący sposób: wieża zapisuje referencje do konkretnego agenta, ktory wejdzie w jej zasięg. Jeśli agentów będzie więcej, wybór celu odbędzie się w sposób losowy. Kiedy przeciwnik wyjdzie z pola rażenia, referencja jest odrzucana i pobierana nowa. Wieże posiadają swój koszt, każdy według typu. Walutę użytkownik może zdobyć poprzez niszczenie przeciwników, którzy z kolei zapewniają graczowi środki w zależności od typu przeciwnika.

3.2.6. Tworzenie wież

K**lasa Tower** oprócz podstawowych wymienionych wcześniej, obsługuje obrót wierz oraz opóźnienie w wystrzale pocisków.

TowerBuilder.to klasa pomagająca obsługiwać dodawanie nowych wież. Sprawdza czy kostka, na której użytkownik będzie chciał postawić wierzę jest wolna oraz odpowiedniego typu. Nie jest możliwe umieszczenie dwóch wierz na tym samym polu. Zawarte tutaj także obsługę menu wyboru wierzy. Wydzielenie tej klasy pomogło przy obsłudze kliknięć w przyciski wyboru typu konstrukcji.

Pociski posiadają własna klasę, która jak wcześniej dopasowuje typ pocisku do odpowiedniej enumeracji zawartej w metodzie swojej metodzie Initalize. Zawarto tutaj także informacje o szybkości lotu pocisku do przeciwnika.

3.2.7. Fale przeciwników

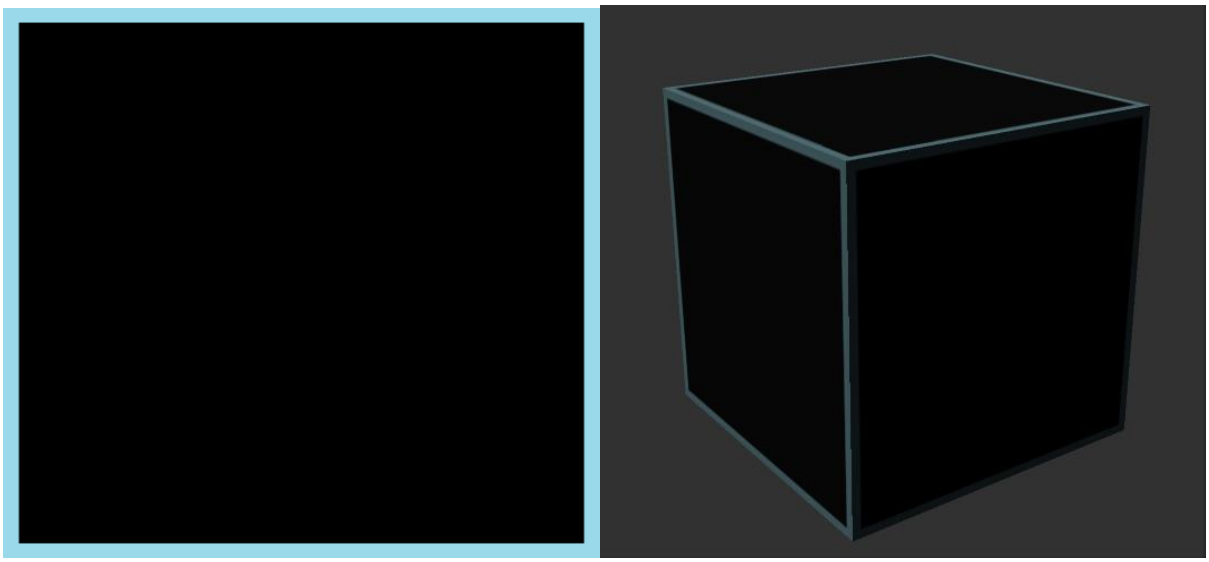
Klasa WaveController ma za zadanie przygotowanie przeciwników. Tworzy ona listę losowych fal przeciwników. Każda fala jest listą kolejnych agentów, którzy będą generowani w fali.

Jednym z problemów związanych z tworzeniem fali przeciwników była różniąca się prędkość poszczególnych typów agentów. Najprostszym rozwiązaniem okazało się być sortowanie listy agentów w fali według ich prędkości.

Losowy tryb gry generuje przypadkowe fale przeciwników z puli predefiniowanych typów. Nie jest ograniczone ich ilość, lecz sumaryczna ilość punktów życia przewidziana na falę. Umożliwia to większą różnorodność fal. Sumaryczna ilość punktów życia zwiększa się z fali na falę.

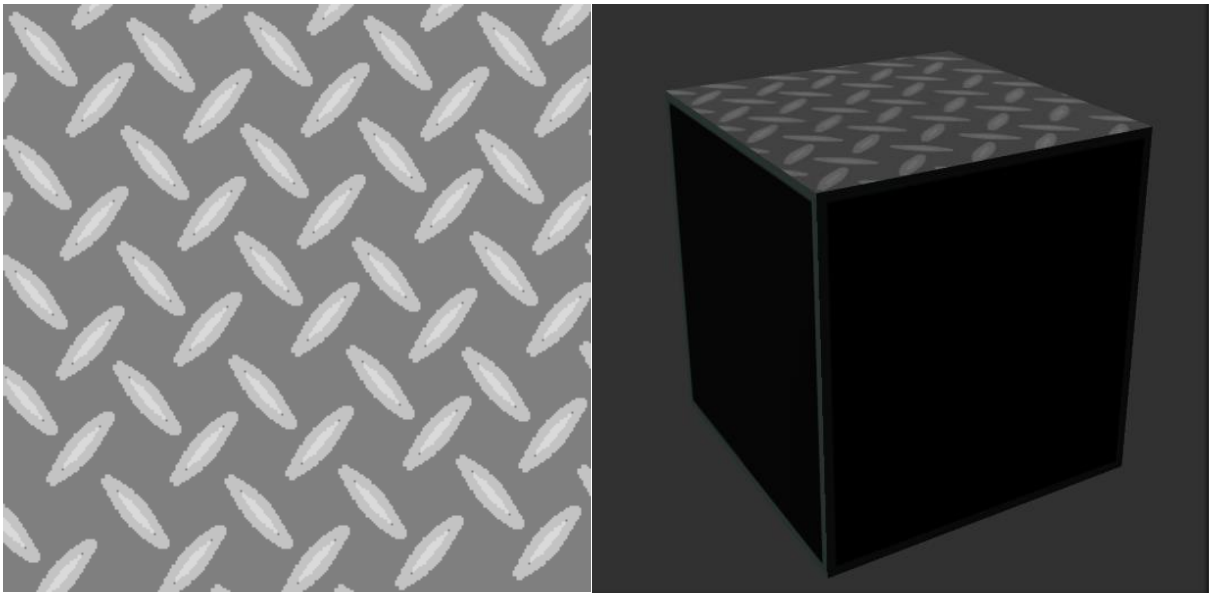
3.2.8. Modele graficzne

Mapa zbudowana została z następujących bloków. Sześciany zostały wygenerowane w programie Unity i zmapowane przy pomocy plików graficznych PNG, stworzonych w programie Paint. Poniższe bloki prezentują aktywne miejsca, na których można stawiać obiekty(wieże).



Rys.5. Tekstura oraz model podstawowego bloku

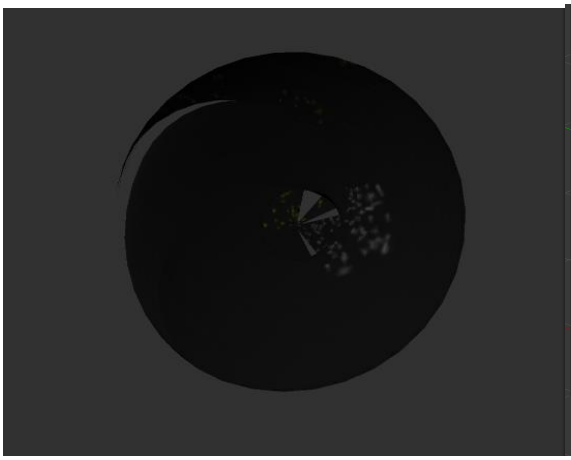
Poniższe bloki prezentują ścieżkę, po której poruszają się obiekty(przeciwnicy).



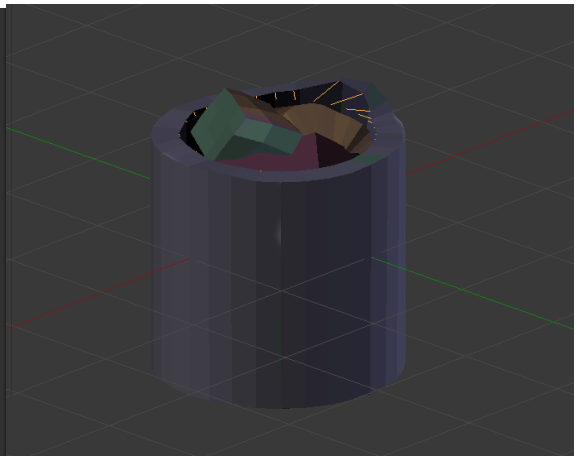
Rys.6. Tekstura oraz model ścieżki

Na blokach aktywnych można stawiać obiekty, w miejscach gdzie nie znajdują się statyczne modele. Statyczne modele są to cztery rodzaje obiektów, które zostały zaprojektowane w programie Blender i zmapowane z teksturami, ale nie posiadają animacji.

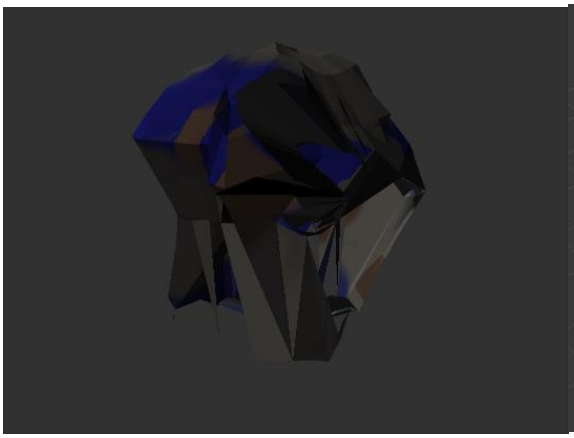
Poniżej prezentacja obiektów statycznych.



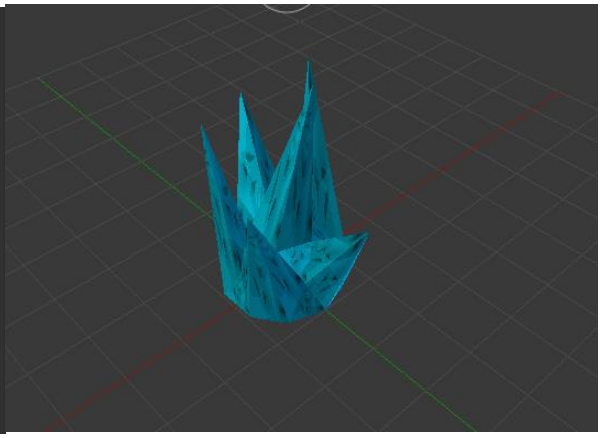
Rys.6. Model czarnej dziury



Rys.7. Model śmietnika

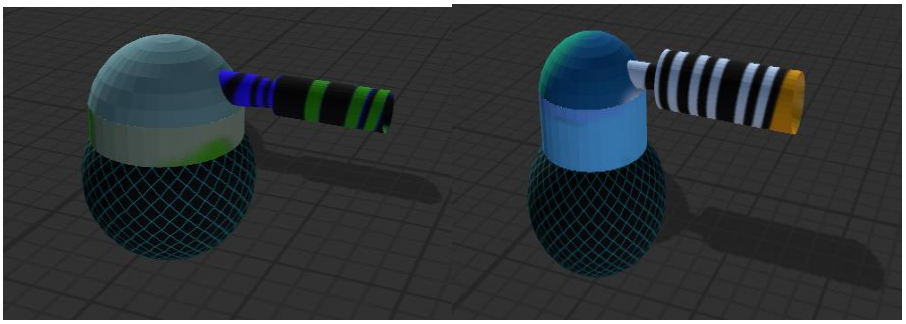


Rys.8. Model zniszczonej wieży



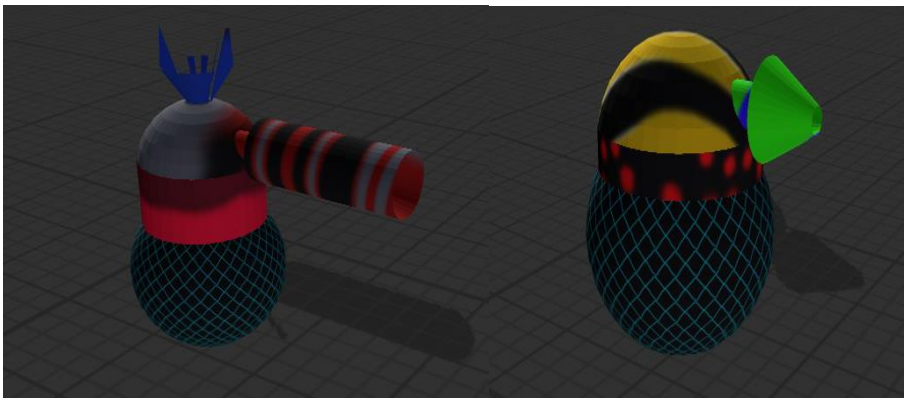
Rys.9. Model kryształów

W grze pojawiają się również modele dynamiczne, które zostały wykonane w programie Blender, ale różnią się tym od poprzednich, że posiadają również animacje. Poniższe obiekty prezentują modele wież. Każda z wież posiada animację wystrzału.



Rys.10. Model podstawowej wieży

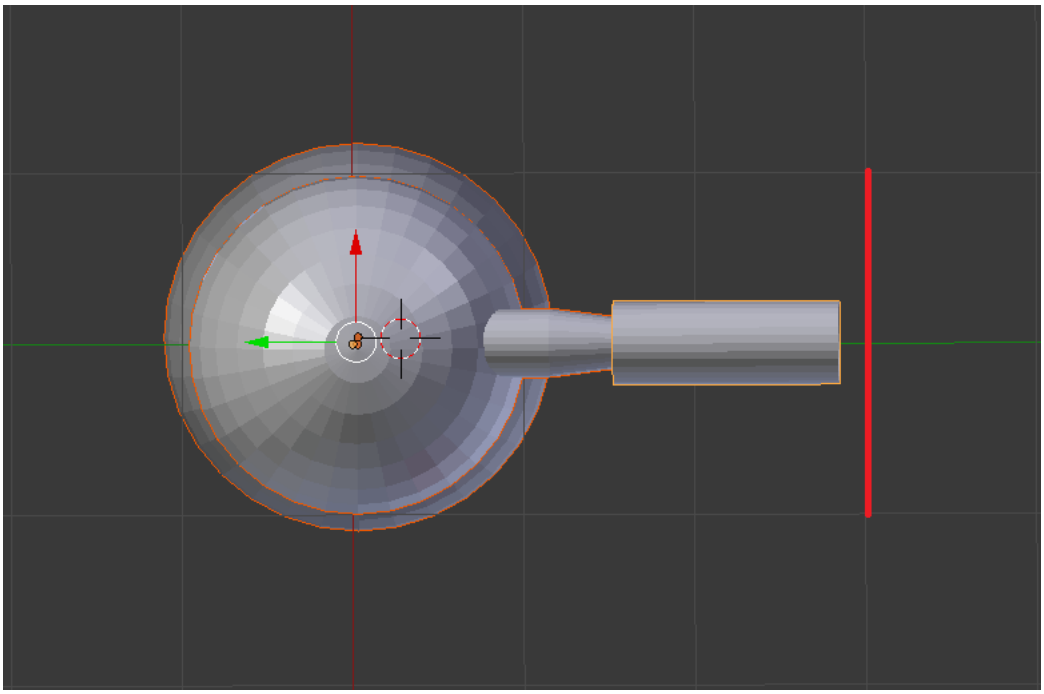
Rys.11. Model szybkiej wieży



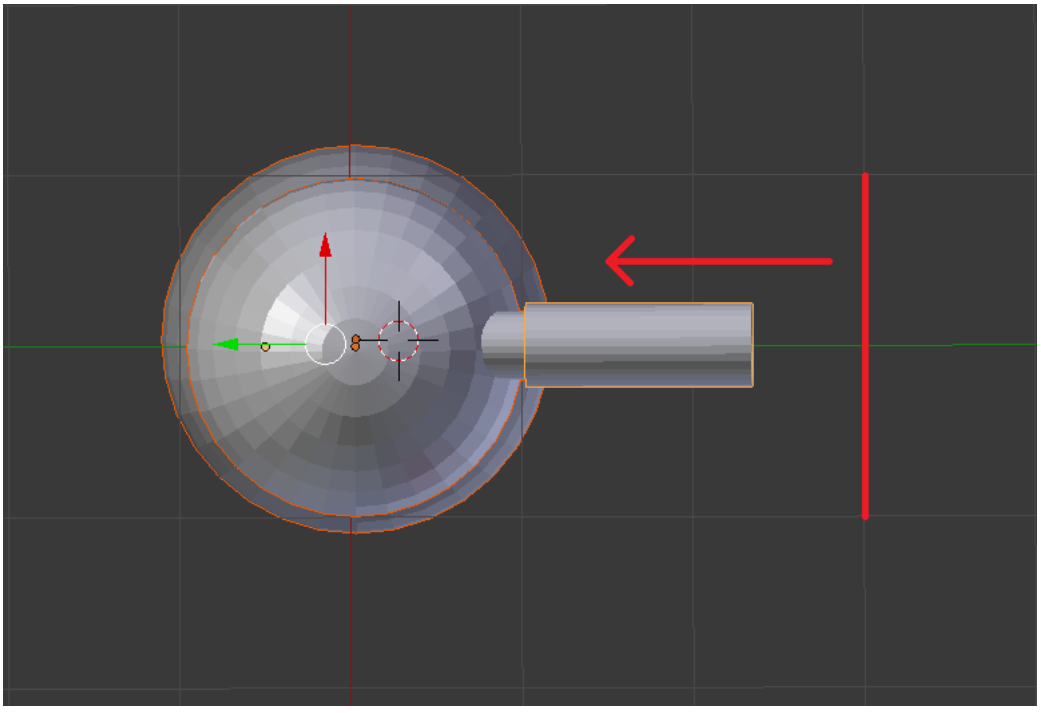
Rys.12. Model wieży snajperskiej

Rys.13. Model ciężkiej wieży

Animacje działa następująco:

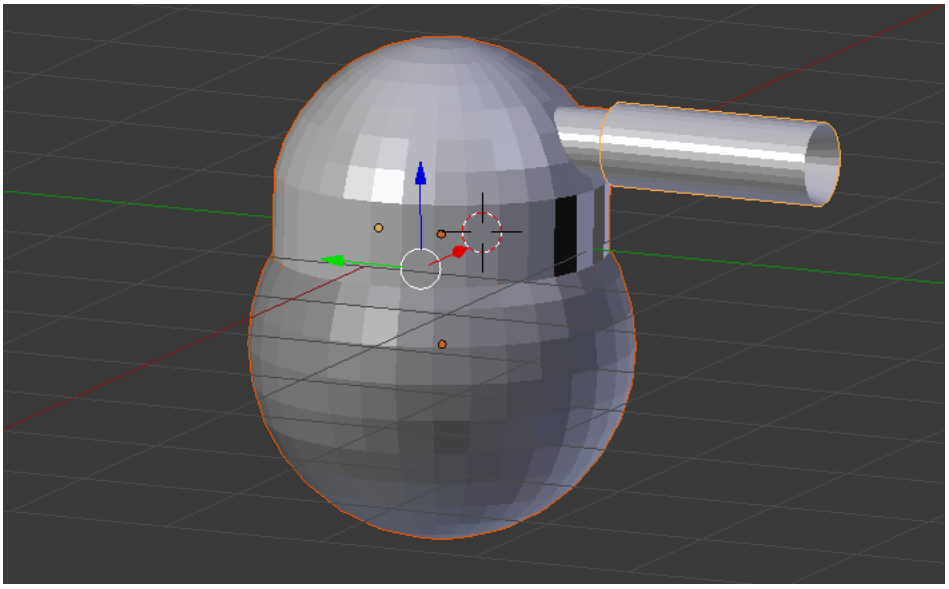


Rys.14. Model wieży z lufą w początkowej fazie animacji



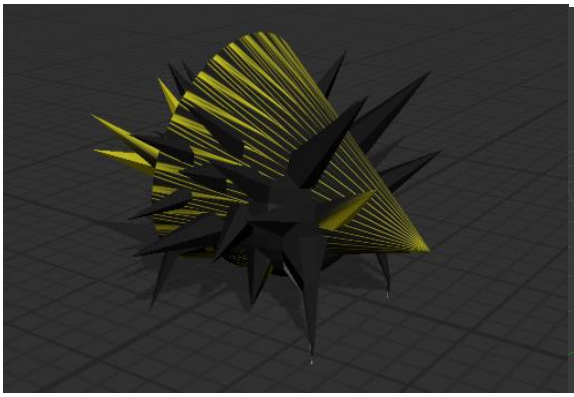
Rys.15. Model wieży po przesunięciu lufy

Lufa wieży podczas wystrzału się cofa, by następnie powrócić do swojej pierwotnej postaci. By wieża mogła oddawać strzał jej model został podzielony na dwie części – lufę oraz podstawę. Lufa zmienia pozycję w czasie, by otrzymać efekt poruszania.

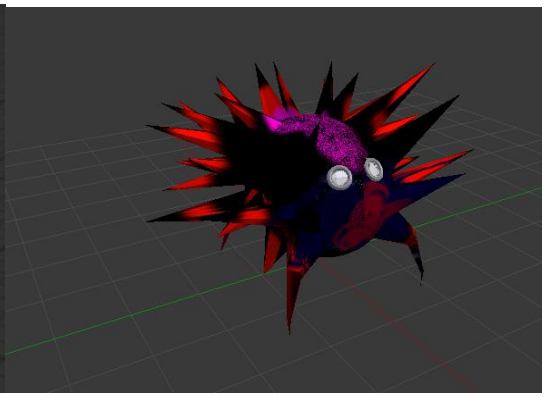


Rys.16. Reprezentacja podziału modelu na bryły

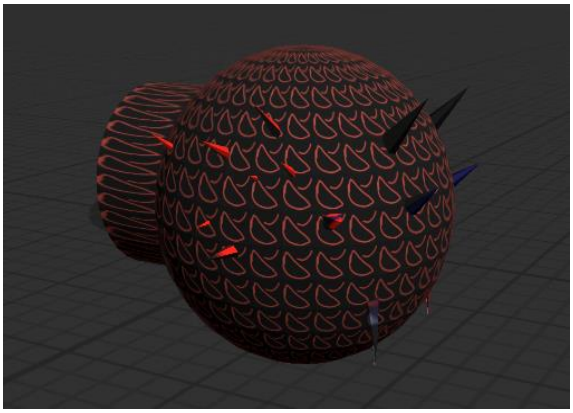
W grze występują też przeciwnicy. Poniżej zaprezentowane modele.



Rys.17. Model super przeciwnik



Rys.18. Model szybkiego przeciwnika



Rys.19. Model opancerzonego przeciwnika

Napotkane problemy podczas tworzenia modeli:

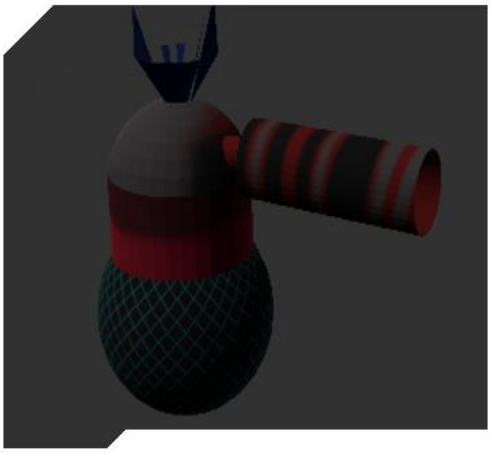
Dostosowywanie modeli do wciąż zminiającego się kodu może powodować wymuszenie tworzenia większej ilości modeli, które nie zostają wykorzystane w aplikacji. Najlepszym rozwiązaniem tego problemu jest tworzenie modeli bardziej uniwersalnych np. bez sprecywanego frontu.

Animowanie modeli przy pomocy kości jest dość skomplikowane i problematyczne, szczególnie gdy modele nie są humanoidalne. By tego uniknąć został zastosowany podział brył na części. Pozwala to na lepszą kontrolę nad modelami, ale metoda ta jest bardziej czasochłonna przy bardziej złożonych modelach, ponieważ trzeba dbać o odpowiednie łączenie części.

Blender oraz Unity posiadają różnie ułożone osie odniesienia(Oś Y oraz Z są zamienione). Rozwiązaniem tego problemu jest zwykły obrót bryły względem osi X. Może to jednak powodować zmianę wartości początkowej osi X w programie Unity z 0 na -90. Żeby tego uniknąć należy w Blenderze przejść do trybu edycji, następnie użyć skrótów klawiszowych R(rotacja), X(obrót względem osi X), -90(ustawienie wartości kąta obrotu), Enter(Akceptacja zmiany), CTRL+A(otworzenie menu zastosowania zmian), zaznaczyć Rotation i przejść do trybu obiektowego. Tak zapisany projekt pozwoli nam na obrócenie bryły bez ingerencji w pozycje startowe w programie Unity. Po przeniesieniu brył wszystkie wartości osi będą ustawione na 0.

3.2.9. Interfejs użytkownika

Interfejs użytkownika został wykonany w dwóch programach – Paint oraz Gimp. Animacje, efekty oraz funkcjonalność poszczególnych elementów zostały wykonane na silniku Unity.



Rys.20. Grafika przycisku

4.Podsumowanie

4.1. Wnioski

Podczas realizacji projektu, członkowie sekcji utwierdzili się w przeświadczeniu, iż programowanie obiektowe dobrze nadaje się do tworzenia gier komputerowych. Dzięki niemu łatwiejszy był podział zadań oraz odpowiedzialności.

Git, jako narzędzie pozwoliło na znaczne zrównoleglenie prac, dzięki czemu mogliśmy szybciej dostarczyć działającą aplikację. Nie obyło się niestety bez problemów i konfliktów przy mergowaniu, ale wspólnymi siłami udało nam się je rozwiązać. Również Skype pozwoliło nam lepiej się organizować i komunikować. Wspomagał on nasze wspólne programowanie, kiedy siedzieliśmy wszyscy nad jakimś problemem a jedna osoba udostępniała widok ekranu.

Silnik Unity miał w założeniach ułatwić nam pracę jednak brak znajomości jego obsługi przysporzył nam wielu problemów. Wiele czasu zajęła nam nauka jego obsługi. Również czasami było ciężko znaleźć rozwiązania do napotkanych przez nas problemów. Jednak społeczność zgromadzona wokół środowiska Unity jest spora i chętna do pomocy na wielu forach.

Przyjęto i zmodyfikowano model przyrostowy zarządzania projektem, dzięki któremu wyznaczaliśmy funkcjonalności, które w kolejnych przyrostach implementowaliśmy i testowaliśmy. Model ten wspomógł działanie naszej sekcji.

Podsumowując, mimo napotkanych trudności, sekcji udało się ukończyć projekt na czas oraz spełnić wszystkie jego założenia. Mimo braku znajomości Unity i niektórych narzędzi przez część sekcji, dzięki wzajemnemu wsparciu dostarczono projekt na czas.

4.2. Spis źródeł

[1]. https://unity3d.com/learn/tutorials/

[2]. https://www.youtube.com/watch?v=beuoNuK2tbk&list=PLPV2KyIb3jR4u5jX8za5iU1cqnQPmbzG0

[3]. https://www.youtube.com/user/quill18creates/

[4]. https://docs.unity3d.com/Manual/