

**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

1.2. Założenia projektu

2. Wybrane narzędzia

2.1. Technologia – język C#

2.2. Środowisko programistyczne – Microsoft Visual Studio 2017

2.3. System kontroli wersji – GIT

2.4. Silnik Unity

3. Podział i omówienie prac

3.1 Piotr Zuber

3.1.1. Algorytm proceduralnego generowania mapy

3.1.2. Algorytm znajdowania ścieżki

3.2 Krzysztof Szwej

3.3 Konrad Sladkowski

3.4 Bartosz Czech

3. Dokumentacja użytkownika

a) O programie

b) Wymagania techniczne

c) Instalacja

d) Korzystanie z programu

4. Dokumentacja techniczna

a) Podział programu

b) Działanie programu

c) Algorytmy i ważniejsze metody

d) Zastosowane narzędzia

5. Podsumowanie

5.1. Wnioski

5.2. Spis źródeł

**1.Wstęp**

**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 //todo**

- Rozgrywka jest typu *tower defence*

- Mapy są generowane proceduralnie

- W grze występuje różnorodność przeciwników oraz wież

- Świat gry jest utrzymany w klimacie *sci-fi*

**2.Wybrane narzędzia**

**2.1.** **Technologia – język C#**

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

**2.2. 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.

**2.3. 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.

**2.4. 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 tutoriali czy dobrze napisanej dokumentacji. Członkowie sekcji chcieli również nauczyć się obsługi tego środowiska.

**3. Podział i omówienie prac**

**3.1. Piotr Zuber**

**3.1.1. Algorytm proceduralnego generowania mapy**

Stworzony algorytm generuje teren 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 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);

}

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;

}

**3.1.2. 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 hurystyką, dzięki czemu skrócony zostaje czas generowania ścieżki na mapie, na której poruszają się potwory. Poniżej został zamieszczony algorytm, wykorzystany w grze:

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;

}

**3.2 Krzysztof Szwej**

**3.2.1 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);

}

}

}

}

}

**3.2.2 Obsługa kamery**

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

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);

}

}

**3.3. Konrad Sladkowski**

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

Agenci, pociski oraz wieże, każdy z tych elementow posiada własną klasę dziedziczącą po klasie MonoBehaviour, zawierajacą metody oraz pola pozwalające na odwzorowanie tego elementu w grze. Dziedziczenie po MonoBehaviour umożliwia poprawne przyłączenie klasy jako komponentu lub skryptu do obieków „GameObject“ będących podstawowymi obiektami silnika „Unity“. Klasy te posiadają jedynie 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 generycznej 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 kolonowanie 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 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. Obsługa namierzania agnetów przez wieże również jest tutaj zaimplementowana.

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.

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

**WaveController.CS**

Klasa WaveController ma za zadanie przygotowanie przeciwników. Tworzy ona (w zależności od wybranego trybu gry) listę losowych, lub listę fal predefiniowanych (kampania). 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.

Ustawienie trybu gry w klasie obywa się poprzez użycie konstruktora, który przyjmuje argument bool randomWaveStyle, który domyślnie przyjmuje wartość false.

**5.Podsumowanie**

**5.1. Wnioski**

**5.2. 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/>