

**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 3](#_Toc484783417)

[1.1. Cel projektu 3](#_Toc484783418)

[1.2. Założenia projektu 3](#_Toc484783419)

[2. Wybrane narzędzia 4](#_Toc484783420)

[2.1. Technologia – język C# 4](#_Toc484783421)

[2.2. Środowisko programistyczne – Microsoft Visual Studio 2017 4](#_Toc484783422)

[2.3. System kontroli wersji – GIT 4](#_Toc484783423)

[2.4. Silnik Unity 4](#_Toc484783424)

[3. Podział i omówienie prac 4](#_Toc484783420)

[3.1 Piotr Zuber 4](#_Toc484783421)

[3.1.1. Algorytm proceduralnego generowania mapy 4](#_Toc484783422)

[3.1.2. Algorytm znajdowania ścieżki 4](#_Toc484783422)

[3.2 Krzysztof Szwej 4](#_Toc484783422)

[3.3 Konrad Sladkowski 4](#_Toc484783423)

[3.4 Bartosz Czech 4](#_Toc484783424)

[3. Dokumentacja użytkownika 4](#_Toc484783420)

[a) O programie 4](#_Toc484783421)

[b) Wymagania techniczne 4](#_Toc484783422)

[c) Instalacja 4](#_Toc484783423)

[d) Korzystanie z programu 4](#_Toc484783424)

[4. Dokumentacja techniczna 4](#_Toc484783425)

[a) Podział programu 4](#_Toc484783426)

[b) Działanie programu 5](#_Toc484783427)

[c) Algorytmy i ważniejsze metody 5](#_Toc484783428)

[d) Zastosowane narzędzia 5](#_Toc484783429)

[5. Podsumowanie 5](#_Toc484783430)

[a) Harmonogram oraz podział prac 5](#_Toc484783431)

[b) Wnioski 6](#_Toc484783432)

[c) Spis literatury 6](#_Toc484783433)

**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;

}