Dokumentacja techniczna

Aleksandra Wnuk, Oleksii Sytnik, Paweł Adamczuk

Blok i jego reprezentacja

Generacja świata koło playera

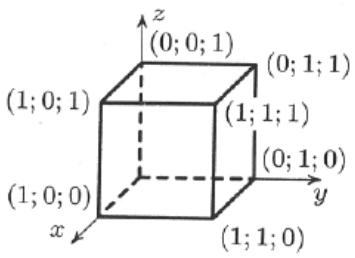
Generowanie terenu

Blok i jego reprezentacja

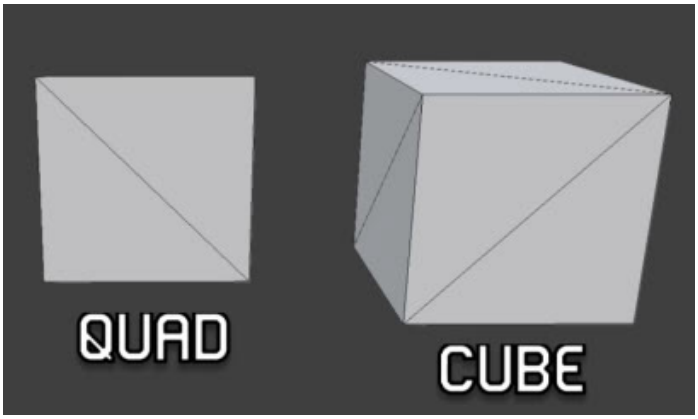
Opis: (Oleksii Sytnik) W tej sekcji będę tłumaczył jak jest zaimplementowany i reprezentowany blok

Kilka informacji które trzeba wiedzieć:

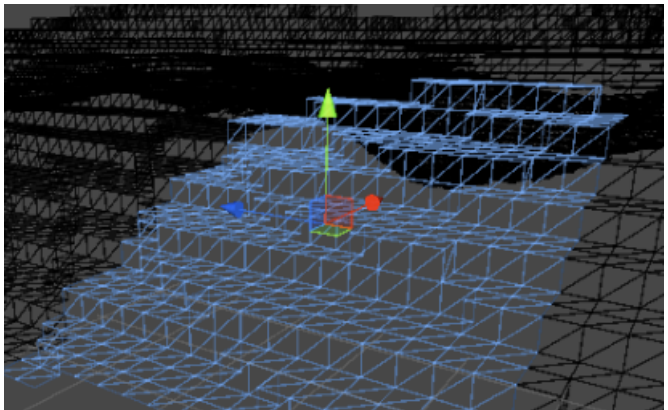
1) Block się składa z 8 wierzchołków które są reprezentowane w postaci wektorów (x, y, z). Każda strona składa się z czterech wierzchołków.



2) Silnik unity potrzebuje trójkątów, które składają się z 3 wierzchołków, aby poprawnie wyświetlić stronę. Więc do odrysowywania jednej strony potrzebujemy 2 trójkątów. Takie dwa trójkąty nazywają się QUAD. Łącząc 6 QUAD-ów dostajemy cały block.



3) Mając informacje jak zbudować jeden block możemy zbudować cały **widoczny** świat, który jest reprezentowany za pomocą MESH



Implementacja w kodzie

Do reprezentacji bloku Chunk (opisany później) żąda dostać dane o MESH konkretnego bloku poprzez wywołanie metody GetMeshData() w klasie BlockHelper.
Dodanie danych o MESH do świata:

Idea funkcji:

Odrysować ściany do bloku, żeby był on widoczny.Jeżeli blok nie ma sąsiadów w jakimś kierunku albo sąsiadem jest blok przezroczysty to jest czas na dodanie danych do MESH żeby ta strona była widoczna, jeżeli blok ma sąsiada to nie trzeba odrysowywać tej strony, bo nie jest ona widoczna, pozwala to na wielokrotne przyspieszenie gry i ulepszenie wydajności.

Działanie:

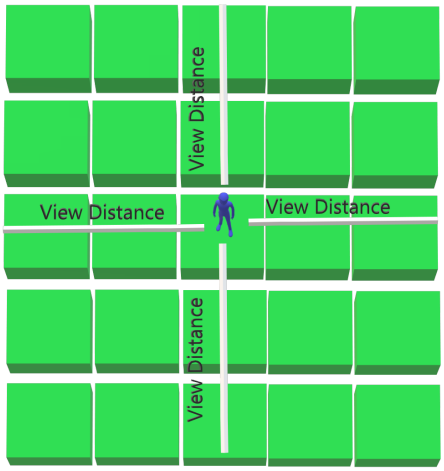
Jest uruchomiona pętla do sprawdzenia każdego kierunku od bloku, który jest podany jako parametr, gdy w tym kierunku możemy odrysować ścianę to wołamy funkcję GetFaceDataIn(). Ta funkcja ma za zadanie dostarczyć do MESH takie dane jak:

1) Wierzchołki. Jest to realizowane za pomocą funkcji GetFaceVertices(). Ta funkcja ma za zadanie dodać 4 wierzchołki w zależności od kierunku.
2) Trójkąty. Jest to realizowane za pomocą funkcji AddQuadTriangles() w klasie MeshData
3) Tekstury. Jest to realizowane za pomocą funkcji FaceUVs().

Generacja świata koło playera

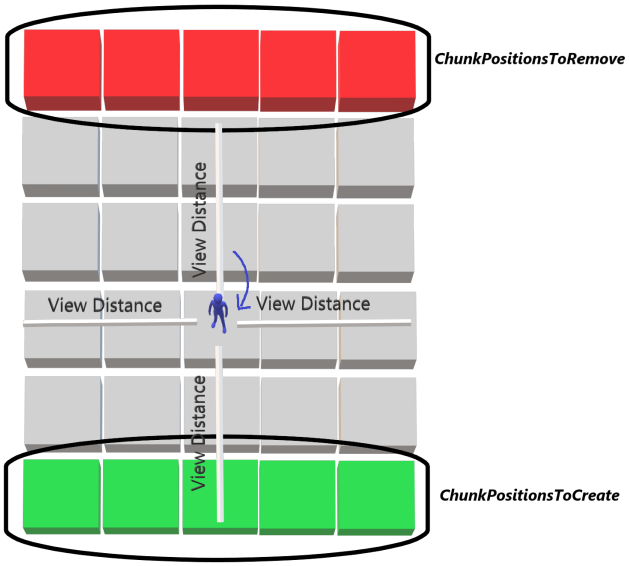
Opis: (Paweł Adamczuk) W tej sekcji będę tłumaczył jak została zmodyfikowana generacja świata przy uruchomieniu gry wraz z tworzeniem instancji postaci głównej na mapie.

Generacja świata: Po naciśnięciu przycisku “New Game” w menu, Unity otwiera nową scenę (główna scena gry) i tworzy gameObjecty podpięty pod sceną. Jednym z nich jest obiekt World zaimplementowany w klasie *World.cs*. Script korzysta z funkcji *Awake()* (od Unity) który zaczyna działanie po utworzeniu instancji gameObjectu World (a więc zaczyna się po zmianie sceny). Wpisuje on do struktury *WorldData* dane potrzebne do zapisania rozmiarów Chunków i ich pozycji na mapie używając do tego Słowników. Po utworzeniu struktury wywoływana jest funkcja *GenerateWorld()*.



Idea funkcji: Zanim przejdziemy do działania funkcji przedstawię ideę funkcji. *GenerateWorld()* nie służy tylko do generacji świata za pierwszym razem, ale także do update’owania świata po poruszeniu się Playera. Funkcja wywołuje także event *OnWorldCreated,* który ma pod sobie podpiętą funkcję *SpawnPlayer()* w *GameManager.cs* .

Działanie: Pierwszą rzeczą, którą wykonuje *GenerateWorld()* jest zawołanie funkcji *GetPositionsThatPlayersSees(pos),* która zwraca strukturę zawierającą listy wektorów z pozycjami do dodawania (na rysunku są to chunki na **zielono**) i do usuwania (są to chunki na **czerwono**). Przy pierwszym wywołaniu funkcja dodaje wszystkie bloki z otoczenia Playera do listy *ChunkPositionsToCreate* natomiast przy każdym następnym tworzy listę chunków w widoku i porównuje ze słownikiem, sprawdza czy jest jakiś chunk który był w widoku ale już jest poza nim lub czy jest jakiś chunk, którego nie ma w słowniku i trzeba go dodać. Korzystamy z różnych funkcji z klasy *WorldDataHelper.cs.* Jest to statyczna klasa z pomocniczymi funkcjami służącymi do wykonywania obliczeń. Jak mamy już strukturę wypełnianą listami wracamy do *GenerateWorld()* i iterujemy po list, po kolei usuwając każdy chunk w *ChunkPositionsToRemove* wraz z swoją data (korzystamy z



WorldDataHelper.cs) z słownika i z świata. Następnie dodajemy każdy chunk do dodawania, zawołując się do *TerrainGenerator.cs* jeśli jest to nowy chunk (zapisujemy dane nowego chunku, a potem tworzymy instancję chunku jako GameObject i rysujemy za pomocą klasę *MeshData.cs*). Dodajemy nowe dane Chunku do słownika razem z instancją chunku a pod samym końcem invokujemy *OnWorldCreated* który ma podpięty funkcję *SpawnPlayer()* w *GameManager.* To kończy działanie funkcji *GenerateWorld()*, czeka on na następnie zawołanie funkcji żeby ponownie wszystko zrobić.

Generowanie terenu

Opis: (Aleksandra Wnuk) W tej sekcji omówię w jaki sposób generowany jest teren w grze.

Użyte algorytmy:

Generacja świata opera się o użycie funkcji perlinNoise:

https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html

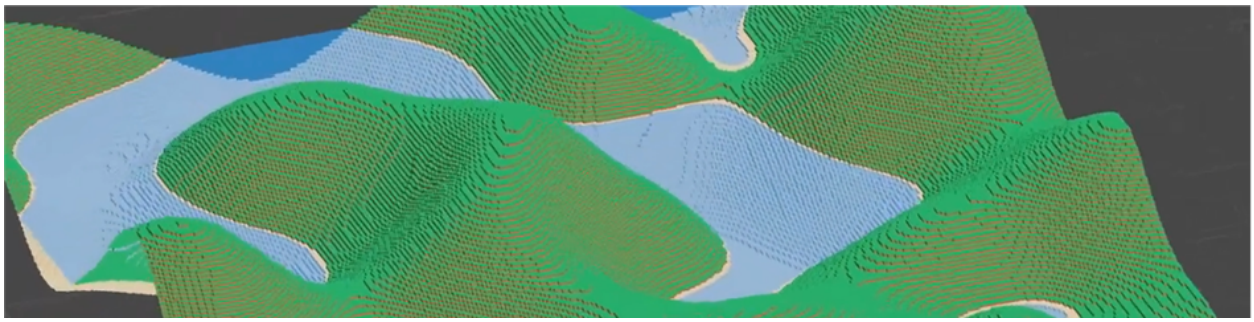
Do funkcji przekazywane są współrzędne szerokości i długości (x i y) punktu na mapie i na ich podstawie zwracana jest wartość z, czyli współrzędna wysokości.

Funkcja ta jest naturalnym wyborem do generowania terenu z kilku powodów:

- po pierwsze otrzymywane z jej pomocą wartości są zdeterminowane przez argumenty, dla konkretnych współrzędnych x i y otrzymamy zawsze takie samo z, dzięki temu nie musimy przechowywać w pamięci całej mapy, tylko możemy generować ją na bieżąco na podstawie długości i szerokości,

- po drugie zwracane przez perlinNoise wartości tworzą powierzchnie przypominającą fale, co budzi skojarzenia z naturalnie występującymi w naturze obrazami.

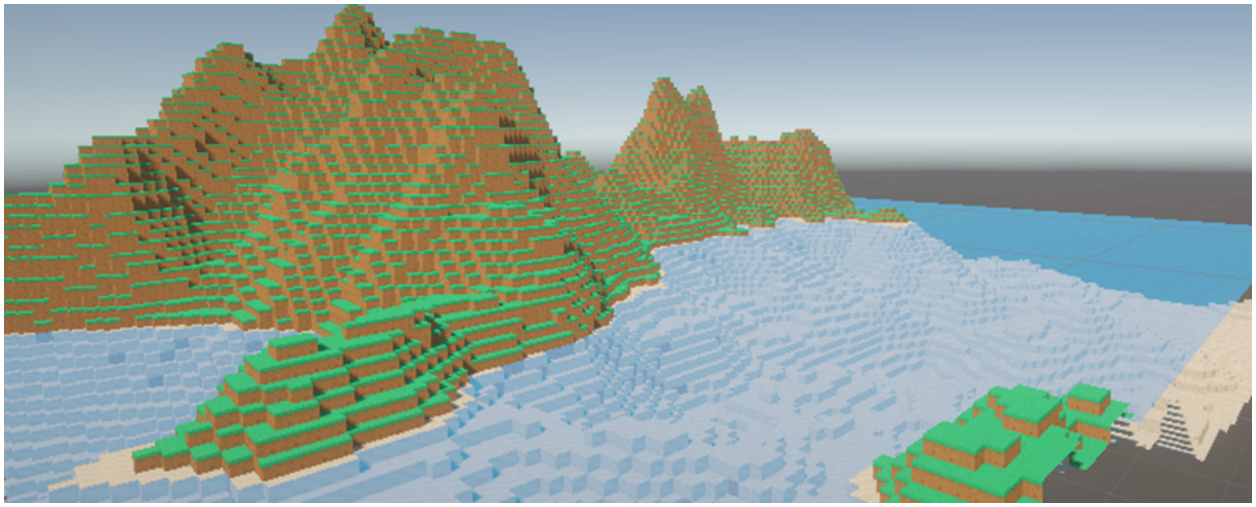
Niezmodyfikowana funkcja perlinNoise ma jednak taką wadę, że wygenerowany z jej pomocą teren jest bardzo monotonny i tym samym wygląda nienaturalnie.



Żeby urozmaicić generowany krajobraz zastosowano dwie modyfikacje:

1. zastosowano Octave Perlin Noise. Metoda polega na dodawaniu do siebie wartości z kilku różnych funkcji perlinNoise, gdzie przez różne funkcje perlinNoise rozmiemy takie, które różnią się amplitudami i częstotliwościami, dokładne wyjaśnienie pod linkiem https://adrianb.io/2014/08/09/perlinnoise.html. W efekcie tej modyfikacji uzyskany krajobraz jest znacznie bardziej zmienny, wspomniane wcześniej w niezmienionym perlinNoise fale są znacznie bardziej zróżnicowane

2. zastosowano domain warping. Metoda polega na dodaniu do funkcji perlinNoise tzw. offsetu, czyli przesunięcia na argumentach x i y. Przesunięcie to powoduje powstawanie na wygenerowanej mapie uskoków (przesunięcie może pochłonąć część wybrzuszenia) czy gwałtownych wzniesień (przesunięcie może pochłonąć wypłaszczenie wzniesieina). Gwałtowne zmiany uzyskane tą metodą dobrze odzwierciedlają występujące w naturze krajobrazy (góry, skarpy). Dokładne wyjaśnienie metody pod linkiem: http://www.nolithius.com/articles/world-generation/world-generation-techniques-domain-warping.



Implementacja:

Implementaja algorytmu do generowania świata umieszczona jest w klasie statycznej Noise.cs w metodzie *public static float OctavePerlin(float x, float y, NoiseSettings settings)*. Metoda ta zwraca współrzędną z (wysokość) na podstawie współrzędnych x i y (szerokość i długość), dodatkowo przyjmuje obiekt NoiseSettings, który zawiera parametry dla perlinNoise:

- noiseZoom – skalowanie w szerz funkcji perlinNoise (skalowanie argumentów), skutkuje zagęszczeniem (przerzedzeniem) liczby wzgórz i dolin
- octaves – liczba sumowań wewnątrz Octave Perlin Noise
- offset – przesunięcie dla domain warpingu
- worldOffset – przesunięcie generacji względem współrzędnej (0,0) świata (odpowiednik ziarna)
- Persistance - czynnik zmieniający wartość amplitudy dla kolejnych wywołań perlinNoise w OctaveNoise,
- redistributionModifier, exponent - argumenty metody *public static float Redistribution(float noise, NoiseSettings settings)*, metoda ta ma na celu uwydatnić lub zniwelować zmienność terenu (w zależności od przekazanych argumentów)