

Uniwersytet Wrocławski
Wydział Fizyki i Astronomii

Marcin Pietrzak

Galeria modeli komputerowych

Computer models gallery

Praca inżynierska na kierunku
Informatyka Stosowana i Systemy Pomiarowe

Opiekun
dr hab. Maciej Matyka, prof. UWr

Wrocław, 25 listopada 2022

Spis treści

1 Wstęp	5
1.1 Wprowadzenie	5
1.2 Cel i zakres pracy	6
2 Warstwa Użytkowa	8
2.1 Wygląd i Obsługa programu	8
2.2 Wygląd Galerii w Programie	9
3 Warstwa Programistyczna	10
3.1 Unreal Engine 4	10
3.2 Język C++	10
3.3 Oculus Quest	11
4 Modele wykorzystane w programie	12
4.1 Wahadło Podwójne	12
4.1.1 Wahadło podwójne - kod	12
4.1.2 Wahadło podwójne - UE4	15
4.1.3 Wahadło podwójne - Wygląd symulacji w projekcie	16
4.2 Gra w życie	17
4.2.1 Gra w życie - kod	18
4.2.2 Gra w życie - UE4	19
4.2.3 Gra w życie - Wygląd symulacji w projekcie	21
4.3 Efekt Motyla	22
4.3.1 Efekt Motyla - kod	23
4.3.2 Efekt Motyla - UE4	23
4.3.3 Efekt Motyla - Wygląd symulacji w projekcie	25
4.4 Modele Agentowe	26
4.4.1 Modele Agentowe - kod	26
4.4.2 Modele Agentowe - UE4	29
4.4.3 Modele Agentowe - Wygląd symulacji w projekcie	31
4.5 Boids	33
4.5.1 Boids - kod	33
4.5.2 Boids - UE4	35
4.5.3 Boids - Wygląd symulacji w projekcie	37
5 Nie realizacja, a co było jeszcze robione - tytuł do wymyślenia	38
6 Wnioski	42

Streszczenie

 Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Abstract

 Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

1 Wstęp

1.1 Wprowadzenie

W ciągu ostatnich 10 lat rynek gogli VR zaczął się budzić na nowo dzięki jednemu z ojców założycieli firmy Oculus, Palmer Luckey. Wiek 16 lat zaczął budować swoje pierwsze gogle VR, a kilka lat później założył firmę Oculus i dzięki udanej zbiórce na Kickstarterze jego firma miała fundusze stworzyć swoje pierwsze sklepowe gogle. Po drodze firma wypuściła kilka prototypów dla deweloperów. Sama firma została przejęta przez Facebooka, dzisiaj znana jako Meta, za 2 miliardy dolarów w 2014 roku. Ich pierwsze komercyjne gogle, Oculus Rift, zostały wydane w 2016 roku i kosztowały na start 599 USD. W 2017 roku z firmą pożegnał się Palmer Luckey, a obecnie zajmuje się firmą tworzącą technologię obronne. Natomiast kolejnym komercyjnym sprzętem Oculusa był Oculus Go wydanym w 2018 roku jako sprzęt all-in-one, czyli nie wymagały zewnętrznych urządzeń czy przewodów do działania. Gogle dzięki braku potrzeby podłączania komputera czy smartfonu, gogle okazały się ogromną innowacją na rynku sprzętu VR. Koszt gogli zaczynał się od 199 USD w dniu premiery. W kolejnym roku zostały wydane dwa nowe rodzaje gogli. Pierwszym był następca Oculus Rifta, czyli Oculus Rift S kosztującym 400 USD. Podobnie jak jego poprzednik były stworzone z myślą o PCVR. Recenzentom sprzęt też nie przypadł do gustu, zarzucając mu zbyt małą ilość innowacji od poprzedniej wersji. Inaczej było z drugim produktem, który wypuścili w podobnym czasie czyli Oculus Quest. Cena premierowa zaczynała się od 399 USD. Quest podobnie jak jego poprzednik nie wymagał żadnych dodatkowych urządzeń do działania. Dzięki dość niskiej cenie, jak za możliwości które oferowały gogle, okazały się one ogromnym sukcesem dla Oculusa na tyle ogromnym, że zaczęły one powoli wypychać swojego brata Rift S z rynku, a gwoździem do trumny dla Rift S było ogłoszenie przez Oculusa, że można będzie Questa podłączyć do PC i korzystać z nich jak Rift S. Sam Oculus od tego momentu zaczął tworzyć tylko gogle autonomiczne.



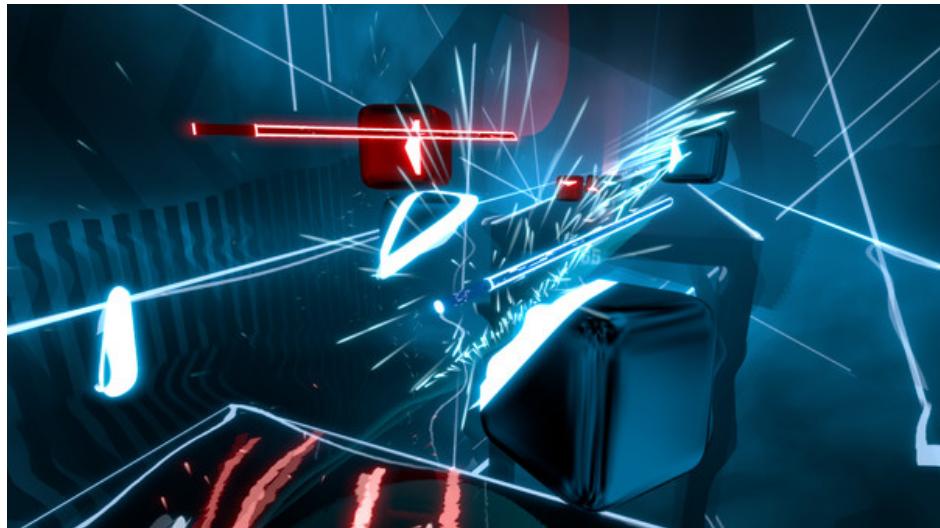
(a) Oculus Quest 1

(b) Meta Quest 2

Rysunek 1: Gogle VR od Reality Labs

W kolejnym roku, tj. 2020 wypuścili kolejne gogle, czyli Oculus Quest 2 których koszt zaczynał się od 299 USD. Dzięki jeszcze niższej cenie i lepszej mocy w stosunku do poprzednika stały się najpopularniejszymi goglami na rynku. W ciągu ostatnich dwóch lat Oculus zmienił nazwę na Reality Labs, Facebook na Meta, a Oculus Quest 2 na Meta Quest 2. Obecnie Meta jest firmą próbującą stworzyć swój pierwszy metavers. W ostatnich tygodniach wypuścili też gogle z myślą o zastosowaniach biznesowym i dla

metaversum, Meta Quest Pro, kosztujące 1500\$. Oczywiście same gogle Meta nie są jedyne na rynku. Podczas boomu na rynku VR w 2016 zostały wydane też inne gogle konkurencyjnych firm. Do największych należą HTC Vive, gogle wydane przez HTC we współpracy z firmą Valve, jako główny konkurent Oculus Rifta, bo podobnie jak Rift też były tworzone z myślą o rynku PC. Trochę inną drogo poszło Sony tworząc PlayStation VR, były to gogle stworzone z myślą o konsoli PlayStation 4. Natomiast w latach 2016-2022 powstała co najmniej setka nowych gogli VR różnych firm[1], z czego największą popularnością są gogle od Mety[2], w 2021 roku zostało sprzedanych 11 milionów sztuk urządzeń. Z czego 78% były to Meta Quest 2[1].



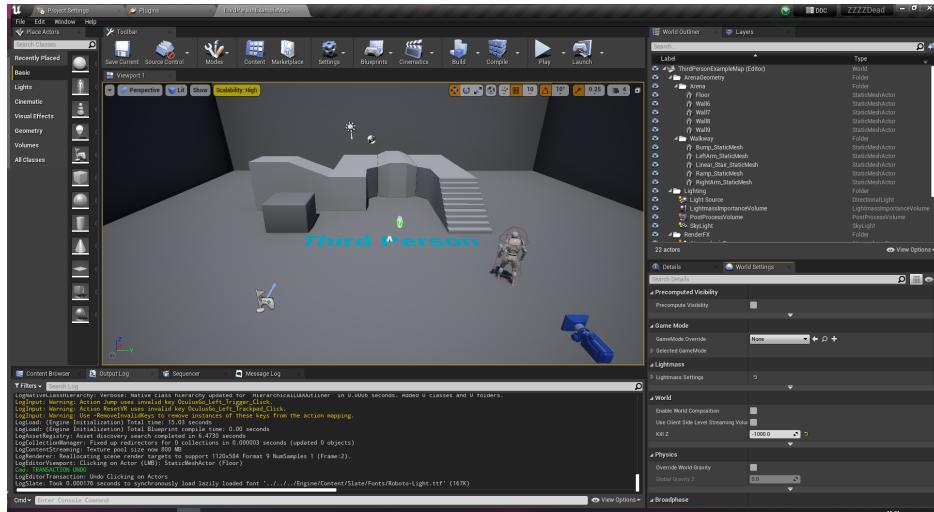
Rysunek 2: Przykład gry stworzonej z myślą o VR: "Beat Saber"

Rynek aplikacji z myślą o VR też się rozwinął w ciągu ostatnich lat. Firmy produkujące silniki do gier takie, jak Epic czy Unity dodały do swoich silników wsparcie VR. Dzięki wparciu od dużych firm rozpoczął się wysyp aplikacji na gogle VR. W oficjalnym sklepie Oculus jest obecnie dostępnych ponad 400 aplikacji[3], na Steam jest ich już ponad 4000[4]. Oczywiście sam sprzęt nie służy też do rozrywki. Może być też wykorzystywany np. przez wojsko do tworzenia symulacji, czy przez artystów do tworzenia dzieł sztuki lub galerii modelów komputerowych.

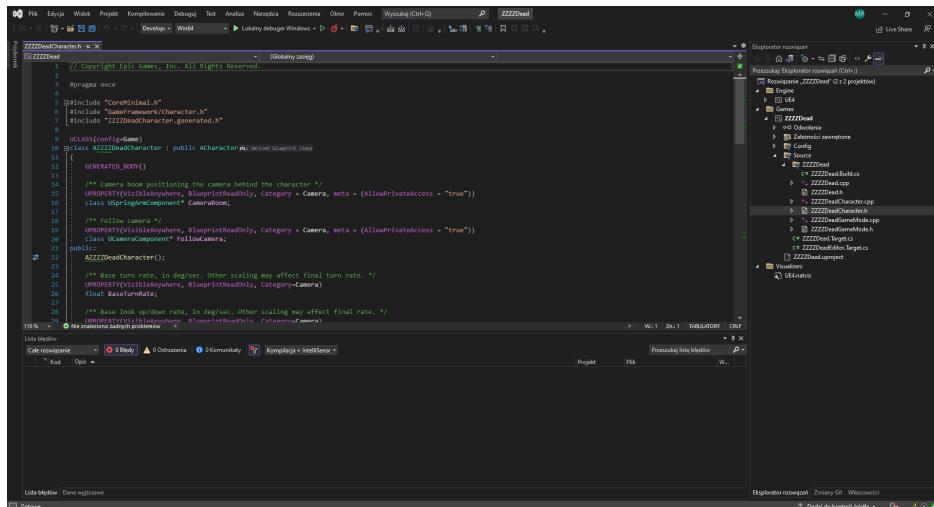
1.2 Cel i zakres pracy

Głównym założeniem jest stworzenie aplikacji VR która przedstawi kilka wybranych modeli komputerowych, dodatkowo będą znajdować się tam informacje o modelu w formie galerii w której znajdować się będzie historia danego modelu i ciekawostki z nim związane. Użyte gogle VR do pomocy przy tworzeniu aplikacji to Oculus Quest, aplikacja została natomiast napisana w silniku Unreal Engine oraz w środowisku programistycznym Visual Studio 2022. Silnik UE4 posiada wsparcia dla gogli VR, a także można w nim pisać klasy i skrypty w języku C++ oraz przy wykorzystaniu Blueprintów, jest to wizualny język skryptowy stworzony z myślą o ludziach, którzy nie mieli styczności z językiem programowania. Np. dla game designerów którzy mogą zrobić proste skrypty, a następnie programista może je przerobić na skrypty C++. Dodat-

kowo po stronie C++ można stworzyć bazowy system, który może zostać rozszerzony w Blueprintach o funkcje do których dostęp jest łatwiejszy w Blueprintach. Celem niniejszej pracy, jest pokazanie że przy dzisiejszej technologii i wiedzy można stworzyć program w mało wymagający sposób, który nie byłby możliwy do zrealizowania jeszcze 10 lat temu.



Rysunek 3: Podstawowy wygląd programu Unreal Engine 4

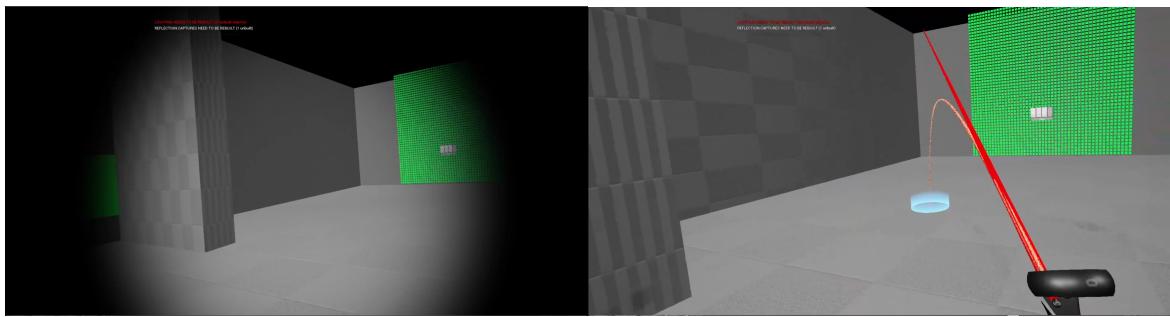


Rysunek 4: Podstawowy wygląd programu Visual Studio 2022

2 Warstwa Użytkowa

2.1 Wygląd i Obsługa programu

Aby uruchomić program potrzebną są gogle VR i kontrolery ruchowe, które są wspierane. W programie wspierane są gogle Oculus Quest i kontrolery Oculus Touch. Program podzielony jest na pokoje, gdzie każdy pokój różni się umieszczonym w nim modelem i możliwymi interakcjami z modelem. Po poziomach można poruszać się na dwa sposoby. Pierwszy sposób wymaga użycia lewego drążka. Dzięki odpowiedniemu wychyleniu drążka można poruszać się w odpowiednie miejsce po pokoju. Jednak przez taki sposób poruszania się niektóre osoby mogą nabawić się choroby symulatorowej[5], aby zmniejszyć szansę jej wystąpienia został zastosowany efekt zwany widzeniem tunelowym. Polega on na zmniejszeniu wielkości widocznego obrazu kiedy poruszamy się po planszy. Im mamy większą prędkość tym efekt jest mocniejszy. Drugi sposób poruszania się polega na teleportacji w inne miejsce na planszy. Po kliknięciu przycisku B na prawym kontrolerze pojawi się kołko informujące nas, że po zwolnieniu przycisku teleportujemy się tam. Ten sposób poruszania się jest bardziej wygodny dla osób z chorobą symulatorową.



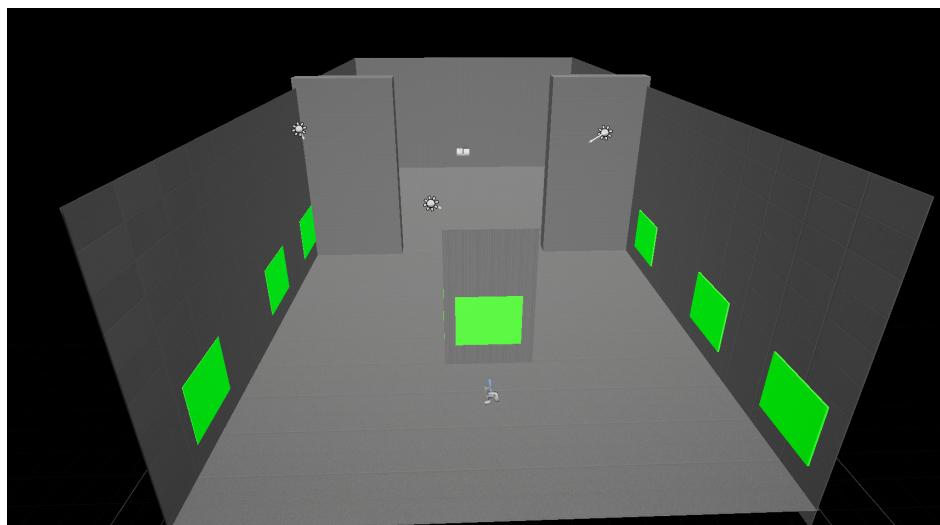
(a) Wizja tunelowa

(b) Ruch teleportacyjny

Rysunek 5: Różny rodzaj poruszania się w programie

2.2 Wygląd Galerii w Programie

Każdy pokój został podzielony na dwie części. W części pierwszej znajdują się informacje o modelu jaki się znajduje w pokoju. Swoim wyglądem przypomina to galerię obrazów, do których można podejść i poczytać ciekawostki o modelu. W drugiej części znajdują się model komputerowy. W zależności od modelu możemy z nim wchodzić w interakcje, aby zobaczyć jak model działa w ruchu.

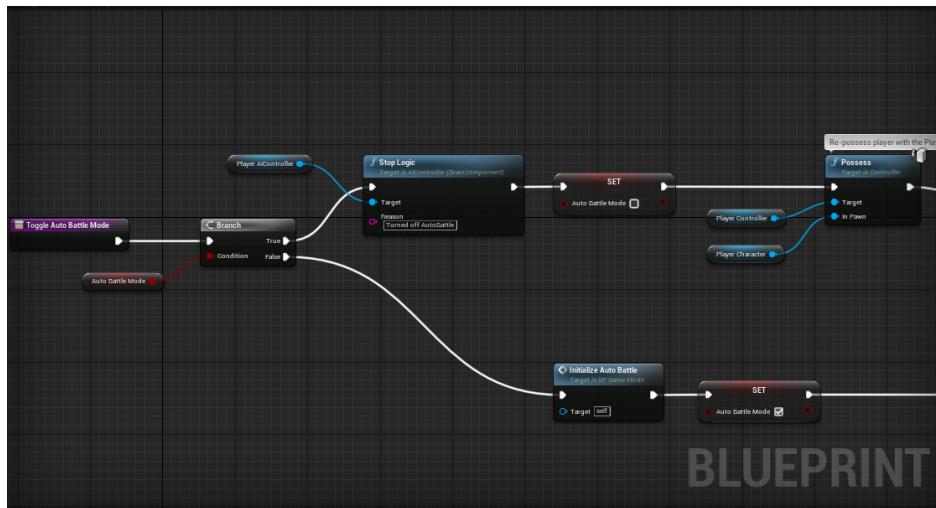


Rysunek 6: Wygląd pokoju galerii w programie

3 Warstwa Programistyczna

3.1 Unreal Engine 4

Program został napisany na silniku Unreal Engine 4. Został on wybrany z powodu wsparcia gogli VR, dużej społeczności, otwartego kodu źródłowego oraz został napisany C++. Dzięki wsparci gogli VR nie trzeba tracić czasu by napisać odpowiednie narzędzie do ich obsługi. Duża społeczność oznacza, że w internecie można znaleźć wiele pomocnych materiałów przy tworzeniu aplikacji. Otwartość kodu oznacza, że można bez większych problemów wprowadzić własne modyfikacje, jeśli są potrzebne. Jedną z najważniejszych funkcji UE4 jest Unreal Editor, czyli graficzny edytor silnika w którym bez znajomości programowania można stworzyć własny program. Kolejną ważną funkcją UE4 jest wbudowany język skryptowy zwany Blueprint. Jest on oparty na koncepcji wykorzystywania interfejsu opartego na węzłach, dzięki którym możemy tworzyć elementy programu bezpośrednio w Unreal Editor bez wchodzenia do kodu źródłowego C++. Jest on używany do definiowania klas lub obiektów metodą programowania obiektowego.



Rysunek 7: Przykładowa funkcja napisana w BP

Unreal Editor pozwala też na tworzenie poziomów w prosty i intuicyjny sposób, poprzez przeciąganie interesującego nas obiektu bezpośrednio na ekran a następnie zmiany jego obrotu czy skali. Edytor pozwala również na uruchomienie obecnie włączonego poziomu, by zobaczyć czy wszystkie funkcje zostały prawidłowo napisane i nie ma żadnych z nimi problemów. Jeśli pojawi się jakiś problem BP pozwala na debugowanie funkcji które zostały w nim napisane.

3.2 Język C++

Innym sposobem na tworzenie nowej funkcjonalności dla tworzzonego programu w Unreal Engine jest napisanie jej w C++. Klasy C++ mogą być używane jako klasy bazowe dla klas Blueprintowych, gdzie podstawy klasy tworzymy w C++, a następnie rozszerzamy jej możliwość w BP. C++ pozwala na tworzenie metod i zmiennych, które

następnie mogą być wykorzystane w BP, dzięki odpowienim specyfikatorom. Dla metod używany jest UFUNCTION z odpowiednimi flagami w zależności od tego, jak dana metoda ma się zachować w BP. Dzięki temu można na przykład w BP odwołać się do metody stworzonej w C++ i korzystać z niej jakby była stworzona w BP. Ma to swoje zalety z czego największą jest to, że metody w C++ są szybsze niż ich odpowiedniki w BP. Podobnie jak dla metod zmienne też posiadają swój specyfikator UPROPERTY, dzięki któremu zmienne stworzone w C++ możemy użyć w BP.

```
 2
 3 #pragma once
 4
 5 #include "CoreMinimal.h"
 6 #include "GameFramework/Actor.h"
 7 #include "BoidTarget.generated.h"
 8
 9 UCLASS()
10 class PRAYVR_API ABoidTarget : public AActor
11 {
12     GENERATED_BODY()
13
14 public:
15     // Sets default values for this actor's properties
16     ABoidTarget();
17
18 protected:
19     // Called when the game starts or when spawned
20     virtual void BeginPlay() override;
21
22     UPROPERTY(VisibleAnywhere, BlueprintReadOnly)
23         class UStaticMeshComponent* Mesh;
24 public:
25     // Called every frame
26     virtual void Tick(float DeltaTime) override;
27
28 };
29
```

Rysunek 8: Przykład kodu C++

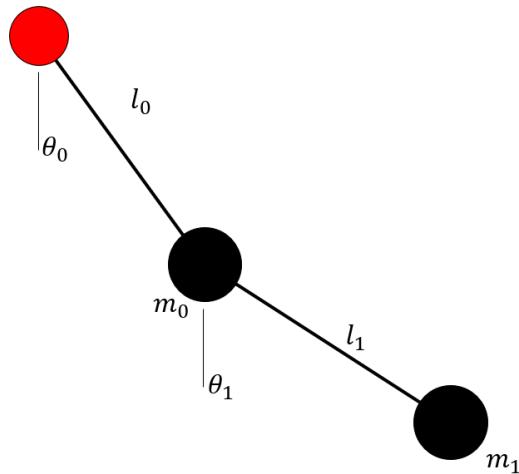
3.3 Oculus Quest

Gogle VR które były wykorzystywane do testowania projektu to Oculus Quest. Gogle posiadają dwa ekranы OLED o rozdzielczości 1440x1600 na każdy ekran. Częstotliwość odświeżania ekranów wynosi 72 Hz. Do obsługi gogli wykorzystywane są dwa kontrolery Oculus Touch drugiej generacji. Dzięki swojej autonomiczności nie potrzeba podłączać gogli bezpośrednio do komputera lecz można wykorzystać lokalną sieć WiFi i oficjalne oprogramowanie firmy Reality Labs. Natomiast dzięki oficjalnemu wsparciu w Unreal Engine można od razu po stworzeniu nowego projektu, zacząć pisać go z myślą o goglach VR.

4 Modele wykorzystane w programie

4.1 Wahadło Podwójne

Pierwszym modelem, który zamieściłem w swoim projekcie jest model wahadła podwójnego. Wahadło podwójne składa się z dwóch wahadeł prostych, tzn. pierwsze wahadło o masie m_0 zostało przymocowane za pomocą pręta o długości l_0 do stałego punktu obrotu, gdzie θ_0 jest przesunięciem kątowym wahadła od położenia równowagi. Drugie wahadło o masie m_1 zostało przymocowane za pomocą pręta o długości l_1 do pierwszego wahadła z własnym kątem swobody θ_1 . Z powodu występowania dwóch stopni swobody θ_0 i θ_1 , a przez to ma bardziej złożony ruch, wahadło jest podatne na chaotyczny ruch i jest bardzo wrażliwy na warunki początkowe [6].



Rysunek 9: Wahadło podwójne

4.1.1 Wahadło podwójne - kod

Kod C++ przeznaczony do tworzenia i zarządzania wahadłami składa się z trzech klas: Pendulum, PendulumSpawn, PendulumControl

Aby obliczyć wartość θ_0 i θ_1 potrzebne są najpierw prędkości kątowe $\dot{\theta}_0$ i $\dot{\theta}_1$ oraz ich przyśpieszenie kątowe $\ddot{\theta}_0$ i $\ddot{\theta}_1$. Aby obliczyć zmianie kąta w czasie trzeba rozwiązać układ równań różniczkowych. Najpierw robimy do dla kątów θ_0 i θ_1 , gdzie pochodna kąta od czasu jest równa $\dot{\theta}_0$ i $\dot{\theta}_1$. Zapisujemy równanie na pochodną prędkości od czasu dla $\dot{\theta}_0$ i $\dot{\theta}_1$ co daje nam przyśpieszenie kątowe $\ddot{\theta}_0$ i $\ddot{\theta}_1$.

$$\begin{aligned} \frac{\partial \theta_0}{\partial t} &= \dot{\theta}_0 \\ \frac{\partial \theta_1}{\partial t} &= \dot{\theta}_1 \\ \frac{\partial \dot{\theta}_0}{\partial t} &= \ddot{\theta}_0 \\ \frac{\partial \dot{\theta}_1}{\partial t} &= \ddot{\theta}_1 \end{aligned} \tag{1}$$

Równania [1] możemy zapisać jako równanie różnicowe, tzn. dokonujemy przekształcenia, w którym pochodną po czasie zastępujemy prostym operatorem liniowym:

$$\begin{aligned}\theta_0 &= \theta_0 + \dot{\theta}_0 * dt; \\ \theta_1 &= \theta_1 + \dot{\theta}_1 * dt; \\ \dot{\theta}_0 &= \dot{\theta}_0 + \ddot{\theta}_0 * dt; \\ \dot{\theta}_1 &= \dot{\theta}_1 + \ddot{\theta}_1 * dt;\end{aligned}\quad (2)$$

gdzie :

dt – krok czasowy

Jest to realizacja metody Eulera. Brakuje nam już tylko $\ddot{\theta}_0$ i $\ddot{\theta}_1$, które można wyrowadzić ze wzoru [7]:

$$\begin{aligned}\ddot{\theta}_0 &= \frac{g(\sin \theta_1 \cos(\theta_0 - \theta_1) - \mu \sin \theta_0) - (l_1 \dot{\theta}_1^2 + l_0 \dot{\theta}_0^2 \cos(\theta_0 - \theta_1)) \sin(\theta_0 - \theta_1)}{l_0(\mu - \cos^2(\theta_0 - \theta_1))} \\ \ddot{\theta}_1 &= \frac{g\mu(\sin \theta_0 \cos(\theta_0 - \theta_1) - \sin \theta_1) - (\mu l_0 \dot{\theta}_0^2 + l_1 \dot{\theta}_1^2 \cos(\theta_0 - \theta_1)) \sin(\theta_0 - \theta_1)}{l_1(\mu - \cos^2(\theta_0 - \theta_1))}\end{aligned}\quad (3)$$

gdzie :

$\mu = 1 + (m_0 + m_1)$

Tak wyżej utworzone równania możemy wykorzystać w kodzie do obliczenia wartości θ_0 i θ_1 w danym kroku czasowym.

```

1 void APendulum::computeAnglesEuler( float dt )
2 {
3     double u = 1 + mass0 + mass1;
4     theta0bis =
5         (g * (sin(theta1) * cos(theta0 - theta1) - u * sin(theta0)))
6         - (length1 * pow(theta1prim, 2) + length0 * pow(theta0prim, 2) *
7             cos(theta0 - theta1)) * sin(theta0 - theta1))
8         / (length0 * (u - pow(cos(theta0 - theta1), 2)));
9     theta1bis =
10        (g * u * (sin(theta0) * cos(theta0 - theta1) - sin(theta1)) + (u *
11            length0 * pow(theta0prim, 2)
12            + length1 * pow(theta1prim, 2) * cos(theta0 - theta1)) * sin(theta0
13            - theta1))
14        / (length1 * (u - pow(cos(theta0 - theta1), 2)));
15
16     theta0prim = theta0prim + theta0bis * dt;
17     theta1prim = theta1prim + theta1bis * dt;
18
19     theta0 = theta0 + theta0prim * dt;
20     theta1 = theta1 + theta1prim * dt;
}
```

Listing 1: Obliczanie wartości θ_0 i θ_1

Tak wyliczone wartości θ_0 i θ_1 są wykorzystane do obliczenia położen wahadeł w kolejnej funkcji. Pręta wahadeł są stworzone ze "Spline Mesh Component", jest to

zdeformowany Static Mesh w którym podajemy mu obecnie wyliczone współrzędne wahadła, dzięki temu uzyskujemy odpowiednio wyglądające wahadło.

```

51     StartTangent);
52     FirstColumn->GetLocalLocationAndTangentAtSplinePoint(i + 1, EndPos,
53     EndTangent);
54     SplineMesh->SetStartAndEnd(StarPos, Tangent, EndPos, Tangent);
55 }

```

Listing 2: Aktualizacja pozycji wahadła

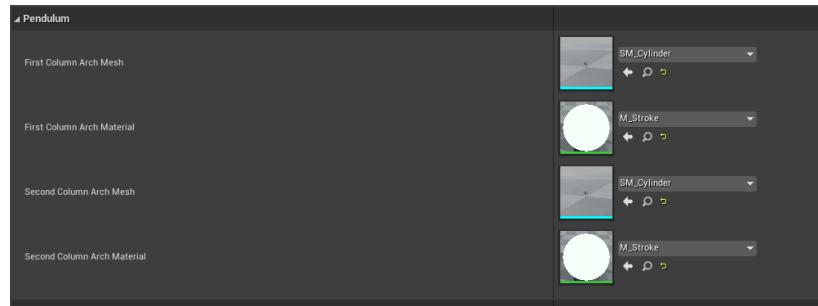
Klasa PendulumSpawn jest natomiast odpowiedzialna za dodawanie, usuwanie wahadł. Klasa jest też odpowiedzialna za uruchomianie i resetowanie wszystkich wahadł które zostały przez dany PendulumSpawn stworzone.

Ostatnią klasą jest klasa PendulumControl, w której są zawarte metody i zmienne potrzebne do stworzenia interfejsu użytkownika, odpowiedzialnego za kontakt między użytkownikiem programu, a PendulumSpawn.

4.1.2 Wahadło podwójne - UE4

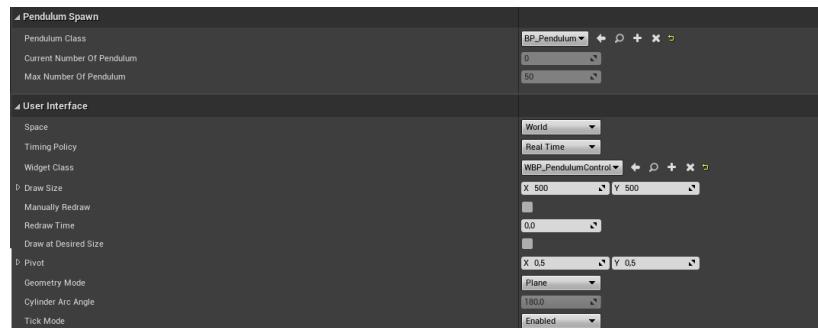
Na podstawie klas C++ z poprzedniego podrozdziału stworzyłem klasy blueprintowe, które to umieszczam na poziomie w edytorze lub są wykorzystane do stworzenia interfejsu użytkownika.

W klasie BP_Pendulum w blueprintie do najważniejszej rzeczy którą można zrobić, jest możliwość zmiany siatki 3D i materiału z którego jest stworzone wahadło:



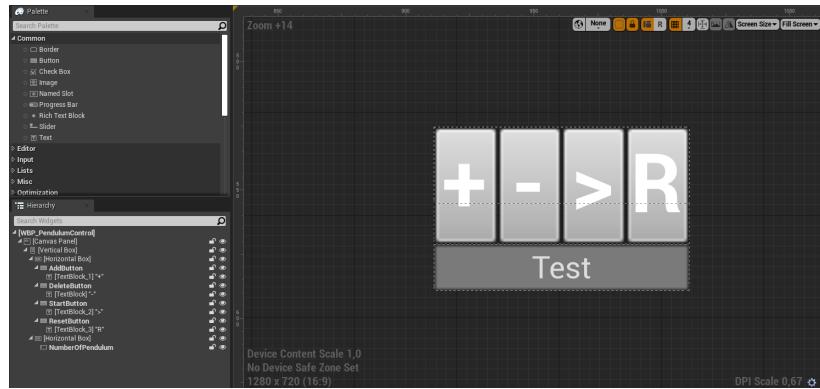
Rysunek 10: Ustawiony mesh i tekstura dla wahadła

W klasie BP_PendulumSpawn ustawiamy tylko jakie ma tworzyć wahadła oraz jaki widget ma być umieszczony do obsługi wahadłów. Po ustaleniu wszystkich potrzebnych rzeczy klasę można umieścić na poziomie:



Rysunek 11: Ustawione wahadło do stworzenia i widget

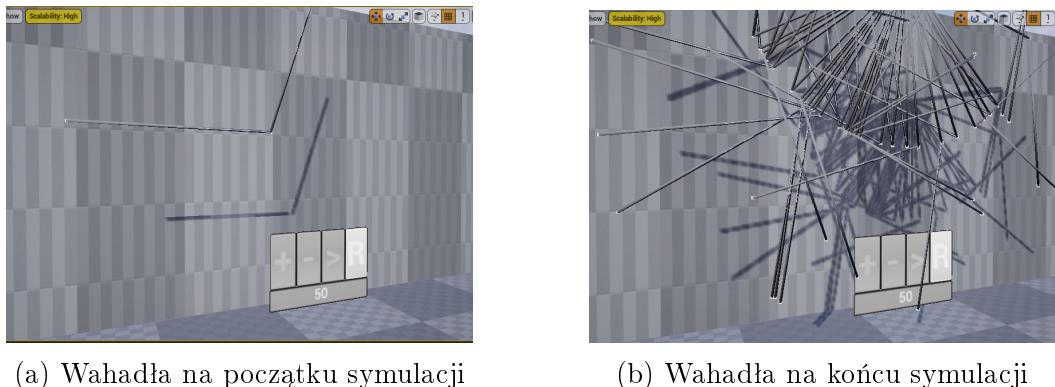
Klasa WBP_PendulumControl jest klasą typu widget, czyli klasą interfejsu użytkownika. W klasie tej trzeba stworzyć odpowiednie jej elementy na podstawie bazowej klasy C++, aby móc obsługiwać wahadła stworzone przez BP_PendulumSpawn



Rysunek 12: Wygląd widgetu w programie

4.1.3 Wahadło podwójne – Wygląd symulacji w projekcie

W projekcie po najechaniu wskaźnikiem z kontrolera ruchowego na panel użytkownika, możemy dodać, usunąć, uruchomić lub zresetować wahadła stworzone przez spawner. Jak pokazano na rysunku 13a, na początku symulacji wahadła są stosunkowo blisko siebie. Lecz w trakcie jej działania w pewnym momencie wahadła zaczną się zachowywać w sposób chaotyczny, tak jak to widać na rysunku 13b.



Rysunek 13: Działanie programu

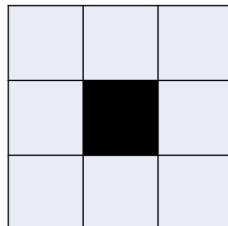
4.2 Gra w życie

Gra w życie, jest automatem komórkowym stworzonym przez Johna Conwaya. Jest to sieć komórek, symulowanych w pamięci komputera, w którym w danej chwili każda z komórek przyjmuje określony stan, według wcześniejszych ustanowionych reguł [8].

Conway zainteresował się problemem przedstawionym w latach 40 XX wieku przez matematyka Johna von Neumanna, który to próbował znaleźć hipotetyczną maszynę, która mogłaby budować kopie samej siebie. Neumannowi udało się, gdy znalazł matematyczny model takiej maszyny z bardzo skomplikowanymi regułami na prostokątnej siatce. Gra w życie powstała jako udana próba Conwaya uproszczenia idei von Neumanna [9]. Gra w życie Conwaya pierwotnie ukazała się w październiku 1970 roku w magazynie "Scientific American" w kolumnie "Mathematical Games" Martina Gardnera, pod tytułem: "The fantastic combinations of John Conway's new solitaire game 'life'" [10], gdzie z miejsca zdobyła ogromną popularność.

Sama Gra w życie nie jest do końca grą. Conway nazywał ją "gra bez graczy", czyli gra w której przebieg jest niezależny od gracza, a jego rola sprowadza się w tym wypadku tylko do utworzenia stanu początkowego, czyli pierwszej generacji komórek [11].

Gra w pierwotnym założeniu rozgrywa się na 2 wymiarowej siatce składającej się z komórek. Każda komórka w danej turze (generacji) może być albo żywa, albo martwa. To czy pojedyncza komórka w kolejnej turze będzie żyć lub nie, zależy od stanu jej 8 sąsiadów i ustanowionych początkowych reguł.



Rysunek 14: Pojedyncza komórka i 8 sąsiadów

Gra w życie Conwaya w pierwotnych założeniach składa się z 4 zasad [9]:

- Każda żywa komórka z mniej niż dwoma żywymi sąsiadami umiera
- Każda żywa komórka mająca więcej niż trzech żywych sąsiadów umiera
- Każda żywa komórka z dwoma lub trzema żywymi sąsiadami żyje, niezmieniona, do następnego pokolenia.
- Każda martwa komórka z dokładnie trzema żywymi sąsiadami ożywa.

Na podstawie kodu [12] zrobiłem wersję Gry w Życie, która działa z kontrolerami ruchów VR.

4.2.1 Gra w życie - kod

W C++ kod gry w życie składa się z trzech klas: CellActor, GridActor, GameOfLifeControl.

CellActor jest klasą w której znajdują się informacje o jednej komórce znajdującej się w siatce. W klasie znajdują się informacje o położeniu na dwuwymiarowej siatce (X i Y), również informację o stanie życia w obecnej generacji oraz o stanie życia w kolejnej generacji. Klasa posiada metodę Clicked, dzięki której jak użytkownik wejdzie w interakcję z komórką zmienia jej status początkowy z martwą na żywą i visa versa.

```
1 void ACellActor::Clicked()
2 {
3     if (Alive) {
4         StaticMeshComponent->SetMaterial(0, BeginCursorOverMaterial);
5         Alive = false;
6     }
7     else {
8         StaticMeshComponent->SetMaterial(0, ClickedMaterial);
9         Alive = true;
10    }
11 }
```

Listing 3: Aktualizacja stanu danej komórki przez użytkownika

Metoda Update, która jest wykorzystywana przez GridActora, podobnie jak metoda Clicked zmienia stan życia komórki, lecz w tym wypadku wykorzystuję zmienną AliveNext.

```
1 void ACellActor::Update()
2 {
3     if (AliveNext) {
4         StaticMeshComponent->SetMaterial(0, ClickedMaterial);
5         Alive = true;
6         SetActorHiddenInGame(false);
7     }
8     else {
9         SetActorHiddenInGame(true);
10        StaticMeshComponent->SetMaterial(0, EndCursorOverMaterial);
11        Alive = false;
12    }
13 }
```

Listing 4: Aktualizacja stanu danej komórki

Klasa GridActor jest odpowiedzialna za tworzenie dwuwymiarowej siatki składającej się z CellActorów i o podanej wysokości oraz długości. W klasie odbywają się wszystkie obliczenia związane z działaniem gry w życie od obliczania obecnie żyjących sąsiadów danej komórki w metodzie CountAliveNeighbors(const int i, const int j)

```
1 int AGridActor::CountAliveNeighbors(const int i, const int j)
2 {
3     int NumAliveNeighbors = 0;
4     for (int k = -1; k <= 1; k++) {
5         for (int l = -1; l <= 1; l++) {
6             if (!(l == 0 && k == 0)) {
7                 const int effective_i = i + k;
8                 const int effective_j = j + l;
```

```

9      if ((effective_i >= 0 && effective_i < Height) && (effective_j >=
10     0 && effective_j < Width)) {
11         if (CellActors[effective_j + effective_i * Width]->GetAlive())
12     {
13         NumAliveNeighbors++;
14     }
15 }
16 }
17 return NumAliveNeighbors;
18 }
```

Listing 5: Zliczanie żyjących sąsiadów danej komórki

Następnie na podstawie obecnie żyjących sąsiadów i reguł jakie zostały ustalone w metodzie UpdateAliveNext(const int Index, const int NumAliveNeighbors) ustawiamy czy dana komórka w następnej klatce będzie żywa lub martwa.

```

1 void AGridActor::UpdateAliveNext( const int Index , const int
2 NumAliveNeighbors)
3 {
4     const bool IsAlive = CellActors[Index]->GetAlive();
5     if (IsAlive && (NumAliveNeighbors < 2))
6     {
7         CellActors[Index]->SetAliveNext(false);
8     }
9     else if (IsAlive && ((NumAliveNeighbors == 2) || (NumAliveNeighbors ==
10    3)))
11    {
12        CellActors[Index]->SetAliveNext(true);
13    }
14    else if (IsAlive && (NumAliveNeighbors > 3))
15    {
16        CellActors[Index]->SetAliveNext(false);
17    }
18    else if (!IsAlive && (NumAliveNeighbors == 3))
19    {
20        CellActors[Index]->SetAliveNext(true);
21    }
22    else
23    {
24        CellActors[Index]->SetAliveNext(CellActors[Index]->GetAlive());
25    }
26 }
```

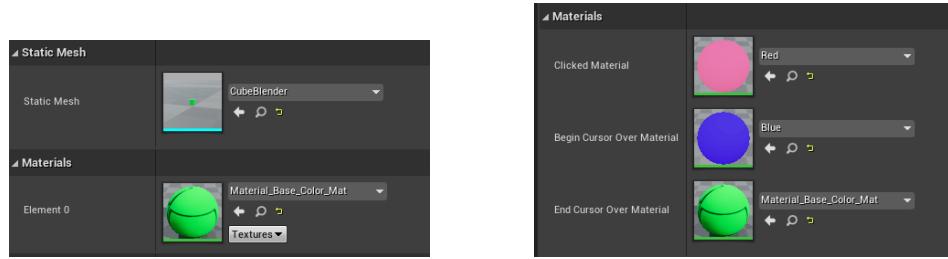
Listing 6: Aktualizacja stanu komórki w następnej klatce

Ostatnią klasą jest GameOfLifeControll która jest odpowiedzialna za stworzenia interfejsu użytkownika, dzięki któremu użytkownik może włączyć grę w życie, zmieniać szybkość działania gry oraz resetować ją do stanu początkowego.

4.2.2 Gra w życie - UE4

Na podstawie powyższych klas zostały stworzone klasy Blueprintowe które można potem umieścić w na poziomie w programie. W BP_CellActor ustawiamy jak dana komórka ma wyglądać w świecie gry i jak się zachować kiedy najedziemy na nią

kontrolerem ruchowym:



(a) Ustawiony mesh w programie dla ko-
mórki (b) Ustawiony wygląd w programie pod-
czas akcji

Rysunek 15: Ustawienia dla CellActora w UE4

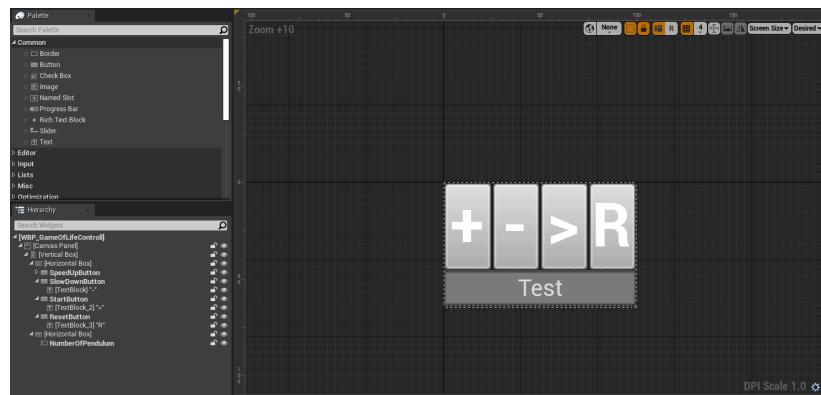
BP_GridActor2D jest klasą Blueprint która znajdzie się na poziomie gry. W klasie ustawiamy na jaką szerokość i wysokość ma zostać stworzona siatka składająca się z **BP_CellActor** oraz dodajemy widget, dzięki któremu możemy sterować symulacją:



Rysunek 16: Ustawienia BP_GridActor2D

Na podobnej zasadzie działa **BP_GridActor3D** w którym dochodzi jeszcze głębo-kość na którą możemy ustawić siatkę.

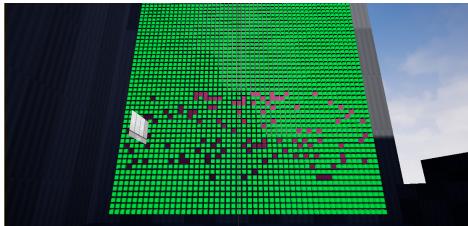
Klasa **WBP_GameOfLifeControll** jest odpowiedzialna kontakt między użytkownikiem a programem, klasa ta jest widgetem dzięki któremu poprzez najechanie kontrolerem ruchowym na odpowiednie opcje możemy włączyć symulację, przyśpieszyć lub ją spowolnić oraz ją zresetować do stanu początkowego:



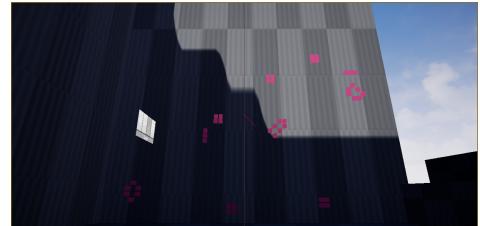
Rysunek 17: Wygląd widgetu w programie

4.2.3 Gra w życie - Wygląd symulacji w projekcie

W programie za pomocą kontrolerów możemy ustawić stan początkowy każdej komórki w poprzez najechanie na nią wskaźnikiem wystającym z kontrolerów.

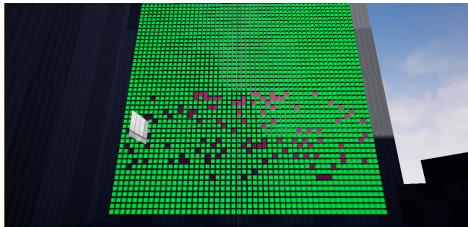


(a) Ożywianie komórek z pomocą kontrolera

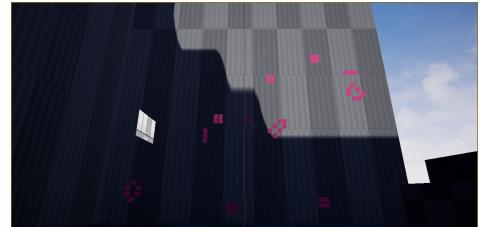


(b) Uśmiercanie komórek z pomocą kontrolera

Po lewej stronie od tablicy z komórkami znajduję się widżet, na którym możemy włączyć grę, przyśpieszyć jej działanie lub spowolnić, albo zresetować komórki do stanu początkowego.



(a) Wygląd tablicy z komórkami i widżetu



(b) Wygląd tablicy w trakcie uruchomienia gry

4.3 Efekt Motyla

"Dowolny układ fizyczny, który zachowuje się nieokresowo, jest nieprzewidywalny." Są to słowa Edwarda Lorenza, meteorologa który jako pierwszy odkrył, że nie można zrobić dobrej prognozy pogody na dłużej niż kilka dni do przodu. W 1960 roku pracował on nad programem komputerowym który miał prognozować pogodę, na podstawie zbioru równań określających zależności między prędkością wiatru, temperaturą, ciśnieniem i wilgotnością [13]. Gdy Lorenz testował swój program po wprowadzeniu danych i po wydrukowaniu wyniku w formie wykresu. Rozkład maksimów i minimów na wykresie wyglądał tak, jak się tego spodziewał Edward. Postanowił jednak ponownie zbadać wyniki, dlatego uruchomił program ponownie, wprowadzając jak myślał takie same wyniki [13]. Okazało się jednak, że wyniki wyszły na odwrót. Po sprawdzeniu danych zauważył, że podał je w postaci przybliżonej z mniejszą liczbą cyfr po przecinku. Było to dla niego tak ciekawe, że spróbował to samo z innymi zbiorami danych i zaobserwował identyczne zjawisko. Lorenz w ten sposób odkrył "efekt motyla", gdzie dla pewnych układów deterministycznych nawet minimalne zmiany wartości danych początkowych zostają bardzo szybko wzmacnione i powodują ogromne zmiany w ewolucji układu [13].



Rysunek 20: Burza

Obecnie układ Lorenza jest bardziej znany jako układ 3 nieliniowych równań różniczkowych[14]:

$$\begin{aligned}\dot{x} &= \sigma(y - x), \\ \dot{y} &= x(\rho - z) - y, \\ \dot{z} &= xy - \beta z,\end{aligned}\tag{4}$$

gdzie:

σ - stała Prandtla,

ρ - stała Rayleigha,

β - obszar obejmujący równania

$\sigma, \rho, \beta > 0$,

Jednakże zwykle podaje się[14]:

$$\begin{aligned}\sigma &= 10 \\ \rho &= \frac{8}{3} \\ \beta &- zmienna\end{aligned}\tag{5}$$

4.3.1 Efekt Motyla - kod

Kod C++ składa się z dwóch klas ButterflyActor i ButterflySpawner. W ButterflyActor najważniejszą metodą jest Tick(float DeltaTime) w której poprzez rozważanie wcześniej pokazanych równań różniczkowych po przecałkowaniu. Następnie odpowiednio ustawiamy te równania dla pozycji X, Y i Z danego aktora [15]

```

1 void AButterflyActor::Tick(float DeltaTime)
2 {
3     Super::Tick(DeltaTime);
4     DeltaTime = ButterflyChange;
5     auto position = GetActorLocation();
6
7     position.X = (position.X + sigma * (position.Y - position.X) *
8         DeltaTime);
9     position.Y = (position.Y + (-position.X * position.Z + rho * position.X
10        - position.Y) * DeltaTime);
11    position.Z = (position.Z + (position.X * position.Y - beta * position.Z
12        ) * DeltaTime);
13
14    SetActorLocation(position);
15 }
```

Listing 7: Metoda Tick()

Jedyną rolą ButterflySpawner jest utworzenie tyle ButterflyActor ile zostanie mu zadane na początku, z drobną zmianą odległości dla każdego kolejnego aktora.

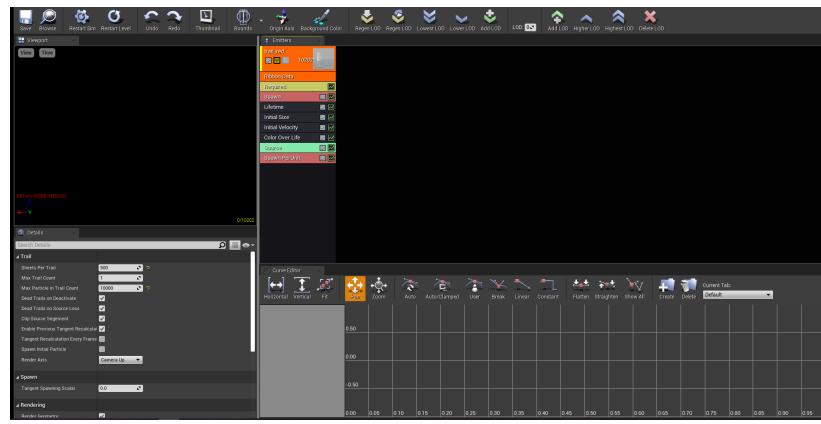
```

1 for (size_t i = 0; i < numberOfButterfly; i++)
2 {
3     const FVector Loc(Origin.X + i * 15, Origin.Y, Origin.Z);
4     AButterflyActor* const SpawnedActorRef = GetWorld()>SpawnActor<
5         AButterflyActor>(ButterflyActor, Loc, GetActorRotation());
6     ButterflyActors.Add(SpawnedActorRef);
```

Listing 8: Tworzenie nowych atraktorów

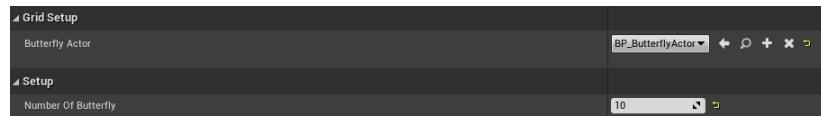
4.3.2 Efekt Motyla - UE4

Na podstawie powyższych klas zostały stworzone klasy Blueprintowe które można potem umieścić w na poziomie w programie. W BP_ButterflyActor dla którego stworzyłem efekt cząsteczkowy który zostawia ścieżką jaką poruszał się dany aktor.



Rysunek 21: Wygląd menu do tworzenia efektów cząsteczkowych w UE4

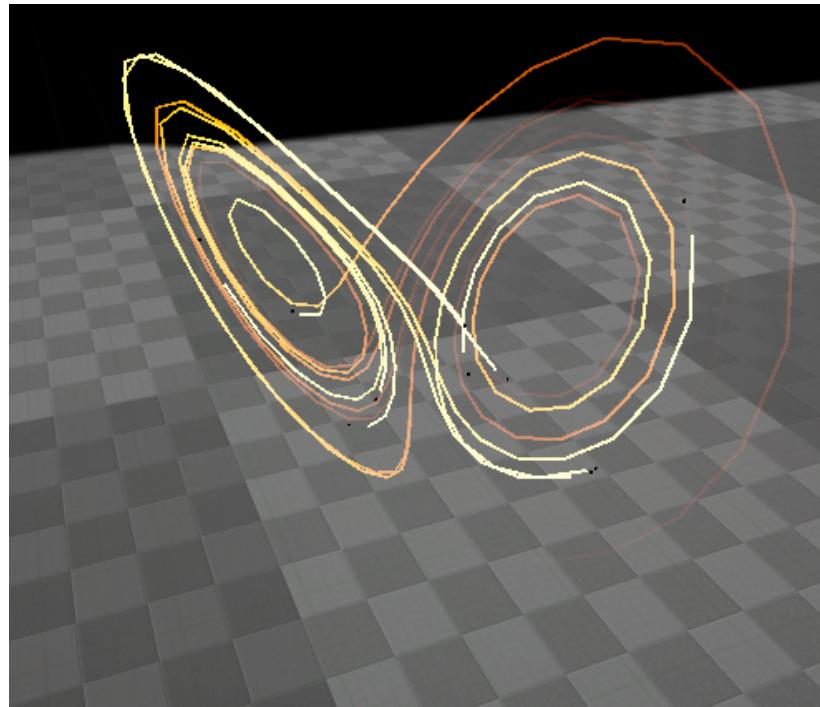
W BP _ButterflySpawner ustawiamy jakiego aktora chcemy ustawić na scenie, oraz ile motyli chcemy stworzyć.



Rysunek 22: Wygląd menu do tworzenia efektów cząsteczkowych w UE4

4.3.3 Efekt Motyla - Wygląd symulacji w projekcie

W programie użytkownik może obserwować efekt motyla z podstawowymi parametrami dla atraktorów[5], łącznie ze ścieżką, po jakiej się poruszają się atraktorzy:



Rysunek 23: Wygląd motyli w programie

4.4 Modele Agentowe

Model agentowy jest typem modelu gdzie analizujemy wpływ agenta na dane środowisko i vice versa. Agentem może być np. komórka, człowiek, zwierzę itd. W modelu agentowym ustawiamy zachowanie poszczególnego lub grup agentów i obserwujemy jak wpływają na resztę w danym środowisku testowym. Główną ideą jest sprawdzenie w jaki sposób czynniki w mikro skali wpływają na czynniki w środowisku makro. Dzięki temu możemy wykorzystać tak zdobytą wiedzę do różnych celów np. Sprawdzenie roznoszenia się epidemii, badać zachowania konsumenckie u ludzi czy tworzyć proste środowiska by zobaczyć jak populacja danych agentów zmieniała się w czasie [16].

W Unrealu Engine 4 stworzyłem prosty model środowiska, gdzie mamy 3 rodzaje agentów: Rośliny, Króliki i Lisy. Celem królików jest rozmnożenie się poprzez znalezienie najbliższego królika do tego zdolnego. Jeśli trakcie trwania symulacji zdrowie królika spadnie poniżej danego poziomu, ponieważ w trakcie trwania symulacji każdy agent nieustannie traci zdrowie lub w trakcie rozmnażania, jego celem wtedy jest znalezienie najbliższej rośliny i skonsumowanie jej. Po zjedzeniu rośliny może ponownie szukać partnera do rozmnażania. Celem lisa podobnie jak królika jest znalezienie partnera do reprodukcji. Podobnie jak królik w wyniku rozmnażania lub czasu traci zdrowie, wtedy jego celem jest znalezienie najbliższego królika i zjedzenie go. Aby każdy miał jakieś szansę na przeżycie wartości prędkości danej grupy agentów różnią się. W obecnym modelu króliki są szybsze od lisów, ale mają też mniej od nich mniej zdrowia. Króliki mają też wyższy licznik reprodukcji. Model też jest bardzo prosty, dlatego np. króliki nie uciekają od lisów.

4.4.1 Modele Agentowe - kod

Kod C++ składa się z następujących klas: AgentBase, PlantAgent, RabbitAgent, WolfAgent, AgentSpawner, AgentSpawnBox, AgentTable i AgentControl. AgentBase jest klasą bazową dla kolejnych trzech rodzajów agentów. W jego skład wchodzi informacja jaki mesh ma dany agent, prosty system kolizji wykorzystywany do interakcji z innymi agentami i użytkownikiem, stan zdrowia oraz informacja o spawnerze na mapie. Każdy agent ma odpowiednio własny zakres zachowań, takich jak: początkowy stan zdrowia czy prędkość. Każdy agent ma też funkcję które odziedziczyli po klasie bazowej, odpowiedzialne za inne zachowania. Do najważniejszych należą funkcja Move() w której dzieje się ruch danego agenta. Przykładowa w RabbitAgent, w zależności od jego stanu zdrowia albo szuka pożywienia albo partnera do kopulacji w pewnej odległości od niego by następnie się do niego zbliżać z zadaną mu prędkością. Jeśli jego zdrowie jest równe 0 dany agent się niszczy.

```
1 void ARabbitAgent :: Move()
2 {
3     Super :: Move();
4
5     float ConstantZ = GetActorLocation().Z;
6
7     if (hp <= RABBIT_MAX_HUNGRY_HP_LEVEL) {
8         TArray<AActor*> Plants;
9
10        UGameplayStatics :: GetAllActorsOfClass(GetWorld(), APlantAgent :: StaticClass(), Plants);
11        if (Plants.Num() > 0) {
```

```

12     size_t atractorIndex = 0;
13     auto atractor = Cast<APlantAgent>(Plants[atractorIndex])->
14     GetActorLocation() - GetActorLocation();
15     float atractortDist = atractor.Size();
16     for (int j = 1; j < Plants.Num(); j++) {
17         auto* plant = Cast<APlantAgent>(Plants[j]);
18         FVector vec = plant->GetActorLocation() - GetActorLocation();
19         float dist = vec.Size();
20         if (dist < atractortDist) {
21             atractorIndex = j;
22             atractor = vec;
23             atractortDist = dist;
24         }
25     }
26     SetActorLocation(GetActorLocation() + atractor / atractortDist *
27 RABBIT_VELOCITY);
28
29     atractorPlant = Cast<APlantAgent>(Plants[atractorIndex]);
30 }
31 else {
32     TArray<AActor*> Rabbits;
33
34     UGameplayStatics::GetAllActorsOfClass(GetWorld(), ARabbitAgent::
35     StaticClass(), Rabbits);
36
37     int i = 0;
38     for (int j = 0; j < Rabbits.Num(); j++) {
39
40         if (Cast<ARabbitAgent>(Rabbits[j]) == this)
41         {
42             i = j;
43             break;
44         }
45
46     size_t atractorIndex = 0;
47     auto atractor = GetActorLocation();
48     float atractortDist = 1000;
49     for (int j = 1; j < Rabbits.Num(); j++) {
50         auto partner = Cast<ARabbitAgent>(Rabbits[j]);
51         auto vec = partner->GetActorLocation() - GetActorLocation();
52         float dist = vec.Size();
53         if (dist < atractortDist && partner->hp >
54 RABBIT_MAX_HUNGRY_HP_LEVEL && j != i) {
55             atractorIndex = j;
56             atractor = vec;
57             atractortDist = dist;
58         }
59     }
60     auto partner = Cast<ARabbitAgent>(Rabbits[atractorIndex]);
61     if (this != partner) {
62         SetActorLocation(GetActorLocation() + atractor / atractortDist *
63 RABBIT_VELOCITY);
64
65     atractorRabbit = partner;

```

```

64
65     }
66 }
67
68 SetActorLocation(FVector(GetActorLocation().X, GetActorLocation().Y,
69                         ConstantZ));
70
71 if (hp != 0) {
72     hp--;
73 } else {
74     OnDestroy();
75 }
76 }
```

Listing 9: Metoda Move()

Kolejną ważną metodą w każdym agencie jest OnOverlapBegin(), jest to metoda która poprzez napisanie jej w taki sposób ma możliwość generowanie eventów kiedy jakiś inny obiekt wejdzie w obszar jego interakcji. Dla RabbitAgent, w zależności od aktora i obecnych potrzeb ma inne zastosowania. Jeśli królik obecnie poszukuje rośliny i wejdzie z nią w obszar interakcji, roślinę usuwa, a sam zyskuje zdrowie. Jeśli obecnie poszukuję partnera i wejdzie w jego obszar interakcji, dany agent jak i jego partner tracą zdrowie do danego poziomu, a następnie tworzą nowych agentów.

```

1 void ARabbitAgent::OnOverlapBegin(UPrimitiveComponent* OverlappedComp,
2                                   AActor* OtherActor,
3                                   UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep,
4                                   const FHitResult& SweepResult)
5 {
6     Super::OnOverlapBegin(OverlappedComp, OtherActor, OtherComp,
7                           OtherBodyIndex, bFromSweep, SweepResult);
8
9     if (Cast<APlantAgent>(OtherActor) == atractorPlant && atractorPlant)
10    {
11        atractorPlant->OnDestroy();
12        hp = RABBIT_MAX_HP;
13        atractorPlant = nullptr;
14    }
15    else if (Cast<ARabbitAgent>(OtherActor) == atractorRabbit &&
16              atractorRabbit)
17    {
18        hp = RABBIT_MAX_HUNGRY_HP_LEVEL;
19        atractorRabbit->hp = RABBIT_MAX_HUNGRY_HP_LEVEL;
20        for (int j = 0; j < RABBIT_REPRODUCE_COUNT; j++) {
21            const FVector Loc(GetActorLocation().X + FMath::RandRange(-50, 50),
22                             GetActorLocation().Y + FMath::RandRange(-50, 50), GetActorLocation().
23                             Z);
24            auto const SpawnedActorRef = GetWorld()->SpawnActor<ARabbitAgent>(
25                RabbitActor, Loc, GetActorRotation());
26            if (SpawnedActorRef)
27            {
28                SpawnedActorRef->hp = RABBIT_MAX_HUNGRY_HP_LEVEL;
29                SpawnedActorRef->SetUpAgent(true);
30            }
31        }
32    }
33 }
```

```

26     attractorRabbit = nullptr;
27 }
28 }
```

Listing 10: Interakcja agenta z innymi agentami

Kolejną klasą jest AgentSpawner którego głównym zadaniem jest kontrola populacji, aby nie przekroczyła zadanej ilości oraz by w razie potrzeby dodał nowych agentów danego rodzaju, jeśli ci się skończą na planszy. Na przykład kiedy na planszy nie ma już AgentRabbit, spawner generuje nowe królików w danej ilości i umieszcza je w losowych miejscach na planszy

```

1 if (Rabbits.Num() == 0)
2 {
3     const FVector Origin = GetActorLocation();
4     TArray<UStaticMeshComponent*> Components;
5     RabbitActor.GetDefaultObject()->GetComponents<UStaticMeshComponent>(
6         Components);
7     ensure(Components.Num() > 0);
8     for (int i = 0; i < RABBIT_COUNT; i++) {
9         const FVector Loc(Origin.X + FMath::RandRange(-50, 50), Origin.Y +
10             FMath::RandRange(-50, 50), Origin.Z);
11         auto const SpawnedActorRef = GetWorld()->SpawnActor<ARabbitAgent>(
12             RabbitActor, Loc, GetActorRotation());
13         if (SpawnedActorRef)
14         {
15             SpawnedActorRef->hp = RABBIT_MAX_HUNGRY_HP_LEVEL;
16             SpawnedActorRef->SetUpAgent(true);
17         }
18     }
19 }
```

Listing 11: Dodawanie nowych agentów metodzie Tick(float DeltaTime)

Drugim zadaniem jest też włączanie i wyłączanie wszystkich agentów na planszy.

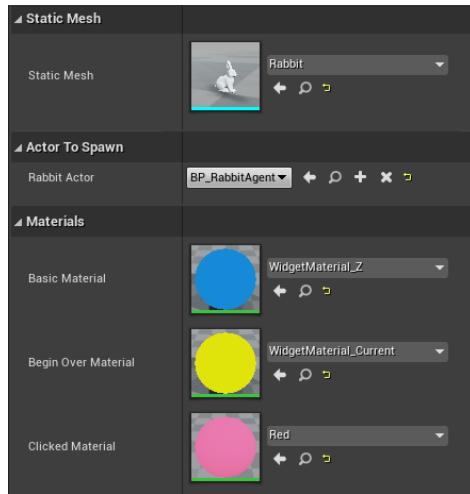
Kolejną klasą jest AgentSpawnBox. Podobnie jak AgentSpawner tworzy nowych agentów, ale z tą różnicą, że do pomocy potrzebuję gracza. W klasie tej nowy agent danego rodzaju jest tworzony, kiedy gracz wejdzie w interakcję z tym obiektem za pomocą kontrolera ruchowego. Dzięki temu, gracz może "wyciągnąć" nowego agenta i postawić go na planszy w takim miejscu jakim chce.

Kolejną klasą jest AgentTable. Ustawia on jedynie tych agentów których gracz postawi na planszy.

Ostatnia klasą jest AgentControl, który odpowiada za tworzenie UI i komunikację pomiędzy graczem, a klasą odpowiedzialną za działanie agentów, czyli AgentSpawner.

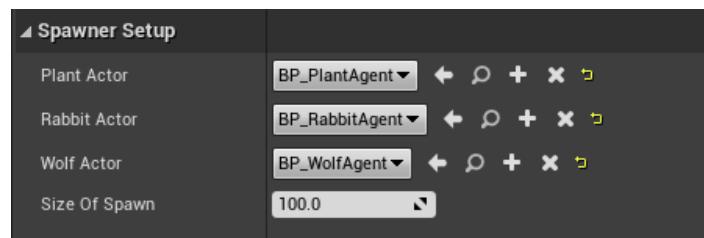
4.4.2 Modele Agentowe - UE4

Na podstawie kodów C++ stworzyłem w UE4 odpowiednie klasy Blueprintowe, które są dziećmi klas z kodu źródłowego. Klasy BP_RabbitAgent, BP_PlantAgent i BP_WolfAgent Ustawia się w taki sam sposób, poprzez wybranie odpowiedniego mesha jakiego chcemy dla danego agenta i wybrać jakiego actora ma stworzyć w trakcie etapu reprodukcji, z wyłączeniem PlantAgent, który powstaje tylko w AgentSpawner. Tutaj też wybieramy, jaki kolor ma mieć actor, kiedy będziemy w nim wchodzić w interakcje na etapie ręcznego tworzenia planszy.



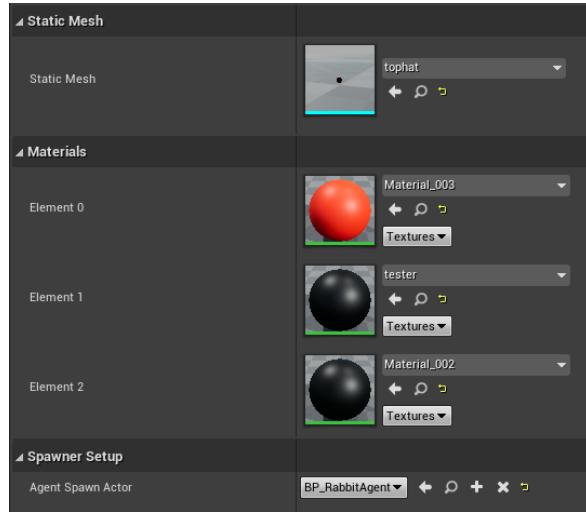
Rysunek 24: Ustawienia Agentów w UE4

Kolejną klasą jest BP_Spawner, która zostanie umieszczona na poziomie. W niej ustawiamy jakich dokładnie agentów chcemy ustawić, których stworzyliśmy w BluePrintach oraz jak duży ma być obszar tworzenia nowych agentów przez spawner.



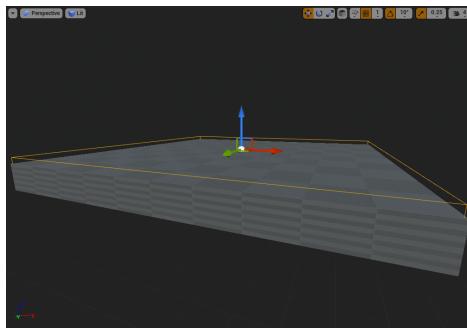
Rysunek 25: Ustawienia Spawnera w UE4

Kolejną klasą jest BP_AgentSpawnBox. Podobnie jak BP_Spawner tworzy nowych agentów, z tą różnicą, że w nim ustawiamy tylko jeden rodzaj agenta do stworzenia. Po ustawieniu agenta za pomocą kontrolerów ruchowych, możemy go postawić na planszy.

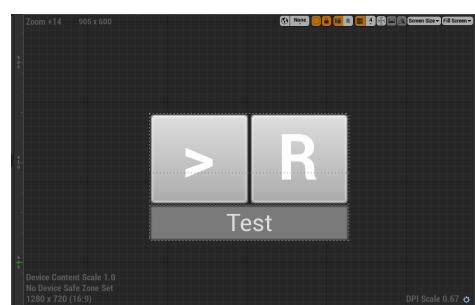


Rysunek 26: Ustawienia SpawnBox w UE4

Ostatnie dwie klasy to BP_AgentTable i BP_AgentControl. Pierwsza w nim jest planszą w której ustwiamy jak plansza ma wyglądać oraz miejsce gdzie możemy położyć i zabrać agentów. Drugą klasą jest widgetem i tak jak w poprzednich modelach, poprzez najechanie kontrolerem na odpowiednią opcję symulacja startuje, albo się resetuje.



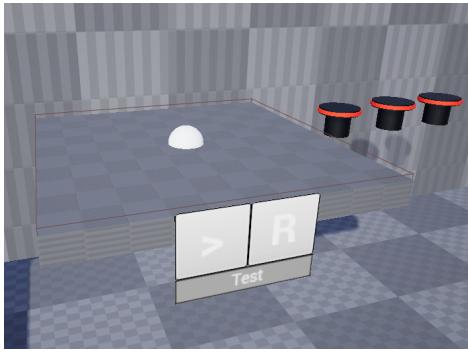
(a) Wygląd planszy w UE4



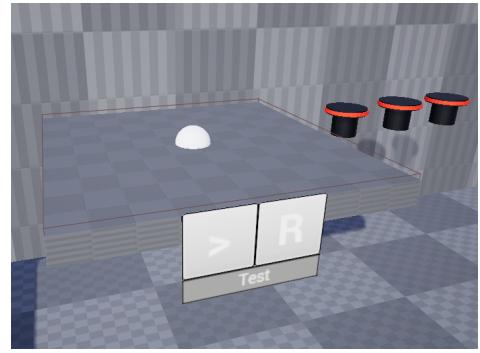
(b) Wygląd widgetu w UE4

4.4.3 Modele Agentowe - Wygląd symulacji w projekcie

W programie za pomocą kontrolerów możemy wyciągać agentów z SpawnBoxów i kłaść ich następnie na planszy. Każdy rodzaj agenta ma swój własny SpawnBox różniący się wyglądem.

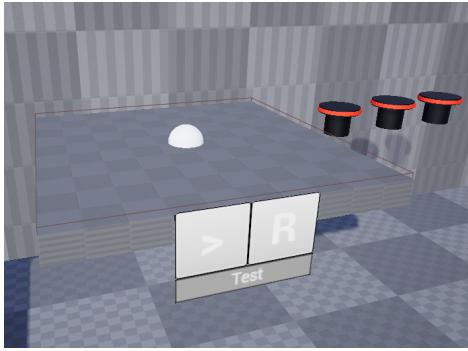


(a) SpawnBoxy w programie

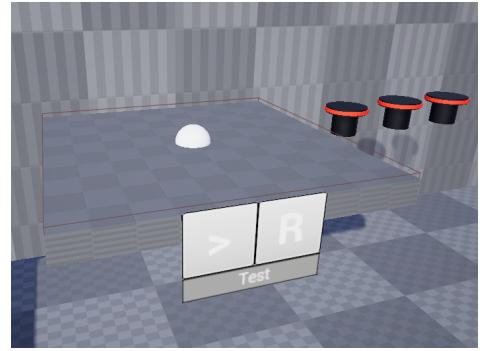


(b) Wyciąganie agenta z SpawnBoxu

Każdego agenta z SpawnBoxu możemy położyć w dowolnym miejscu na planszy. Kiedy agent zmieni kolor na żółty, oznacza to że możemy go spokojnie postawić i że nie zniknie. Agenta z planszy możemy też przełożyć w inne miejsce lub go usunąć poprzez wyciągnięcie go z planszy, a następnie poprzez puszczenie trzymanego agenta poza planszą.

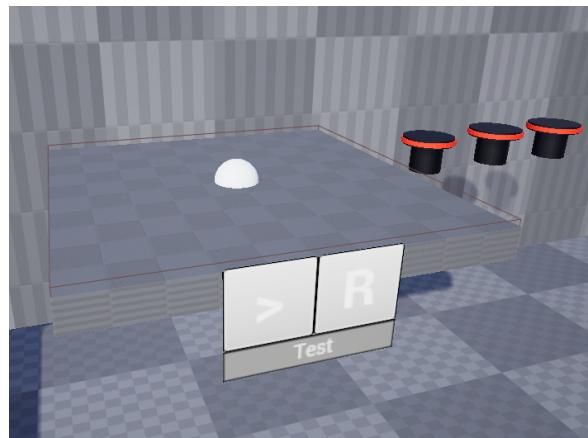


(a) Kładzenie agenta na planszy



(b) Plansza z agentami

Symulację uruchamiamy poprzez najechanie kontrolerem na przycisk start który znajduje się na widgecie przed planszą. Symulacja będzie trwać dopóki nie wyłączymy jej drugim przyciskiem. Podczas symulacji agenci będą się poruszać zgodnie z zachowaniami, jakie im zostały stworzone w kodzie.

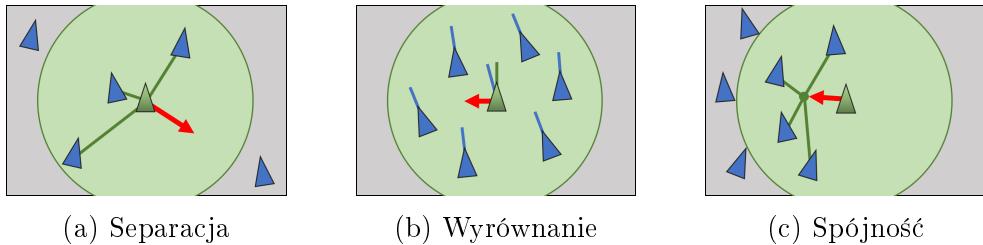


Rysunek 30: Wygląd symulacji w trakcie jej działania

4.5 Boids

W roku 1986 w ciągu dwóch miesięcy, Craig Reynolds pracujący wtedy w firmie "Symbolics" stworzył model komputerowy skoordynowanego ruchu zwierząt takich, jak ławica ryb czy stado ptaków. Model ten był oparty na trójwymiarowej geometrii, którą zwykle używa się animacji komputerowej. Stworzenia wchodzące w skład modelu nazwały "boid", które w dialekcie Nowo Jorskim oznaczają również ptaki, a samą nazwę podobno zainspirował się z filmu "The Producers" Mela Brooksa [17]. Sam podstawowy model stada opiera się na 3 prostych zasadach [18]:

- Separacja(a): Odpowiedzialne za unikanie kolizji pobliskich członków stada poprzez unikanie nagromadzenia ich w pobliżu. Czerwona strzałka oznacza kierunek steru, zielone linie odległości do pobliskich boidów które są w zasięgu danego boida
- Wyrównanie(b): Odpowiedzialne aby przemieszanie było uśrednione do innych pobliskich członków stada. Czerwona strzałka oznacza kierunek steru, zielone linia oznacza obecny kierunek, a niebieskie żądany kierunek ruchu
- Spójność(c): Odpowiedzialne za utrzymanie bliskości członków stada, poprzez poruszanie się obiektu w kierunku średniej pozycji pobliskich członków. Czerwona strzałka oznacza kierunek steru, zielone linia oznacza obecny kierunek, a zielone odległości od środka masy pobliskich boidów



Każdy z boidów reaguje, tylko na inne boidy w jego bliskim otoczeniu od niego, a każdy boid z poza tego otoczenia jest ignorowany. W początkowych eksperymentach, model był trochę bardziej rozbudowany, dzięki czemu boidy, mogły omijać obiekty i poszukiwać celu [19]. Na podstawie kodu [20] zrobiłem prostą wersję która działa z kontrolerami ruchów VR do możliwości sterowania ruchami stada, stado porusza się za obiektem.

4.5.1 Boids - kod

Kod C++ składa się z następujących klas: BoidTarget, BoidManager, BoidSpawner Boid, i BoidsMenu.

BoidTarget jest z nich najmniej rozbudowany, ponieważ jest tylko aktor z ustawnionym meshem i materiałem. Jego jedynym celem jest ustawnienie miejsca do którego mają zmierzać boidy, kiedy będziemy tego chcieli.

BoidManager jest klasą odpowiedzialną za zarządzaniem wszystkimi boidami na poziomie. W każdej klatce dla każdego boida odpalanie jego obliczanie rotacji oraz ją aktualizuję, tak samo robi z pozycją każdego boida.

```

1 void ABoidManager::Tick(float DeltaTime) {
2     Super::Tick(DeltaTime);
3
4     if (ManagedBoids.Num() != 0) {
5         for (ABoid* Boid : ManagedBoids) {
6             Boid->CalculateBoidRotation();
7             Boid->UpdateBoidRotation(DeltaTime);
8             Boid->CalculateBoidPosition(DeltaTime);
9             Boid->UpdateBoidPosition();
10        }
11    }
12 }
```

Listing 12: Metoda Tick()

W BoidManager są też informacje o tym, czy boidy mają udać się w stronę wyznaczonego celu. Znajduję się również informację z jaką siłą mają działać trzy zasady dotyczące boidów. Siłę działania zasad oraz podążanie za celem ustaviamy potem w widgecie, który jest dzieckiem klasy BoidsMenu.

BoidSpawner jest odpowiedzialny za tworzenie nowych boidów, w takiej ilości jaką mu zadamy oraz w danej mu odległości.

W klasie Boid odbywają się wszystkie obliczenia dotyczące 3 zasad. Dla każdej zasady pomiędzy obecnym Boidem a danymi w pobliżu, obliczamy wektor separacji, spójności i wyrównania. Po obliczeniu wektorów są dzielone przez liczbę pobliskich boidów i normalizowane.

```

1 void ABoid::CalculateSeparation(FVector& Separation, ABoid* Boid) {
2     FVector Sub = GetActorLocation() - Boid->GetActorLocation();
3     Separation += Sub * Sub.GetSafeNormal().Size();
4 }
5
6 void ABoid::CalculateAlignment(FVector& Alignment, ABoid* Boid) {
7     Alignment += Boid->GetActorForwardVector();
8 }
9
10 void ABoid::CalculateCohesion(FVector& Cohesion, ABoid* Boid) {
11     Cohesion += Boid->GetActorLocation();
12 }
```

Listing 13: Metody obliczające odpowiednie wektory

Jeśli śledzimy cel, to dodajemy też znormalizowany wektor pomiędzy pozycją celu i boida. Na koniec każdy wektor, po wcześniejszym pomnożeniu przez ich siłę z BoidManger, dodajemy do wektora interpolacji i mnożymy przez daną prędkość obrotu oraz normalizujemy, aby stworzyć rotator, który będzie kolejną rotacją boida.

```

1 void ABoid::CalculateBoidRotation() {
2     TArray<ABoid*> CloseBoids = CalculateClosestBoids(
3         AmountOfBoidsToObserve);
4     FVector InterpolatedForwardVector = FVector::ZeroVector;
5     FVector AlignmentVector = FVector::ZeroVector;
6     FVector CohesionVector = FVector::ZeroVector;
7     FVector SeparationVector = FVector::ZeroVector;
8     FVector TargetVector = FVector::ZeroVector;
9
10    if (CloseBoids.Num() != 0) {
11        for (int index = 0; index < CloseBoids.Num(); index++) {
```

```

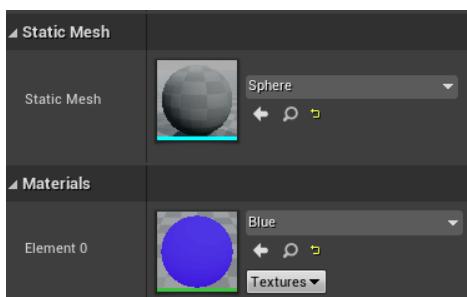
11 ABoid* Boid = CloseBoids[index];
12 CalculateAlignment(AlignmentVector, Boid);
13 CalculateCohesion(CohesionVector, Boid);
14 CalculateSeparation(SeparationVector, Boid);
15 }
16 AlignmentVector /= CloseBoids.Num();
17 CohesionVector /= CloseBoids.Num();
18 SeparationVector /= CloseBoids.Num();
19
20 AlignmentVector.Normalize();
21 CohesionVector.Normalize();
22 SeparationVector.Normalize();
23 }
24
25 if (Manager->IsBoidsFollowTarget()) {
26     TargetVector = CalculateTarget();
27 }
28
29 InterpolatedForwardVector += AlignmentVector * Manager->
30     GetAlignmentWeight();
31 InterpolatedForwardVector += CohesionVector * Manager->
32     GetCohesionWeight();
33 InterpolatedForwardVector += SeparationVector * Manager->
34     GetSeparationWeight();
35 InterpolatedForwardVector += TargetVector * Manager->GetTargetWeight();
36 InterpolatedForwardVector *= TurnSpeed;
37 InterpolatedForwardVector.Normalize();
38 NextBoidRotation = UKismetMathLibrary::MakeRotFromX(
39     InterpolatedForwardVector);
40 }
```

Listing 14: Obliczanie rotacji boida

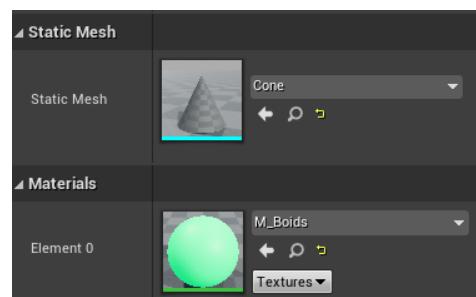
4.5.2 Boids - UE4

Na podstawie kodów C++ stworzyłem w UE4 odpowiednie klasy Blueprintowe, które są dziećmi klas z kodu źródłowego.

W BP_BoidTarget ustawiamy tylko mesh i materiał, a następnie umieszczamy ją na poziomie. Podobnie z BP_Boid w którym też ustawiamy tylko mesh i materiał, ponieważ cała logika została stworzona w C++.



(a) Ustawienia BoidTarget w UE4



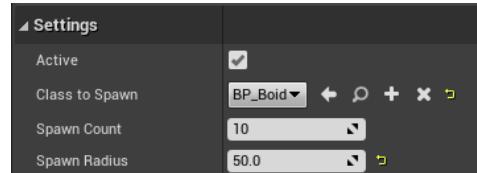
(b) Ustawienia Boid w UE4

BP_BoidManager i BP_BoidSpawner też nie mają rozbudowanych opcji. W pierwszym nic nie zmieniamy, ponieważ jego ustawienia będę wykorzystywane przez w wid-

get. W drugim notomiaż ustawiamy, czy ma być aktywny, jakie boidy ma robić oraz ich ilość i wielkość obszaru w którym mają być tworzone od niego.

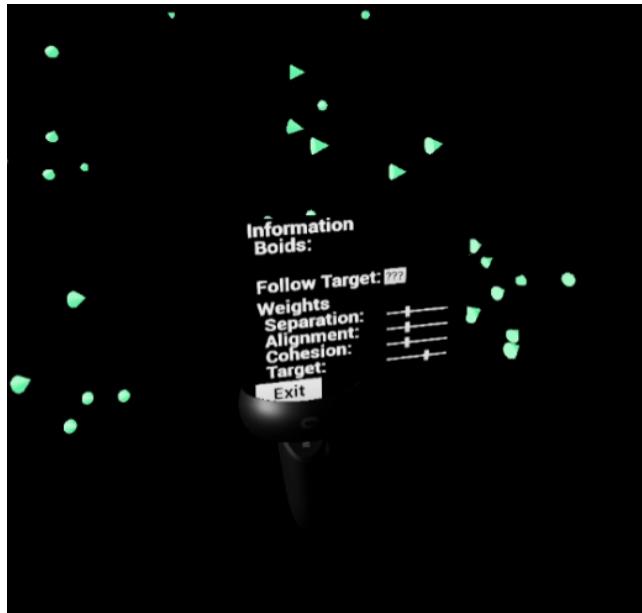


(a) Ustawienia BoidManager w UE4



(b) Ustawienia BoidSpawner w UE4

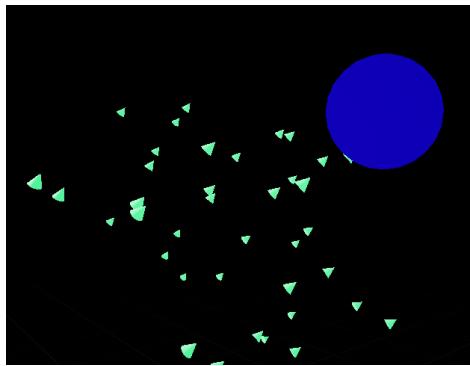
WBP_BoidHud tak jak wszystkie widgety w poprzednich symulacjach, po najechaniu kontrolerem ruchowym, możemy wybrać to czy boidy mają podążać za celem, czy się będą rozpraszać po okolicy. Możemy też wybrać z jaką siłą ma być wykonywana jedna z trzech zasad rządząca boidami, oraz jak blisko ma się zbliżyć celu.



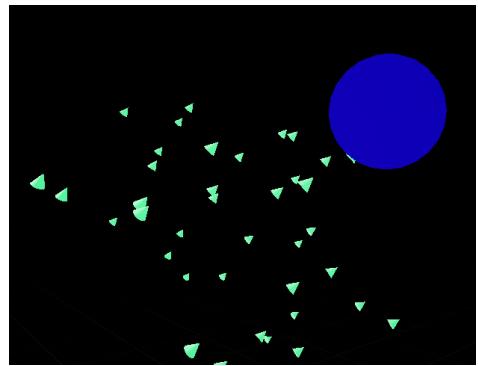
Rysunek 34: Wygląd widgetu do sterowania zachowaniem boidów

4.5.3 Boids - Wygląd symulacji w projekcie

W programie nad lewym kontrolerze mamy widget którym sterujemy założeniem boidów. Za pomocą prawego kontrolera możemy ustawić opcje np. ustawić czy boidy mają podążać za calem oraz możemy wrócić do galerii.

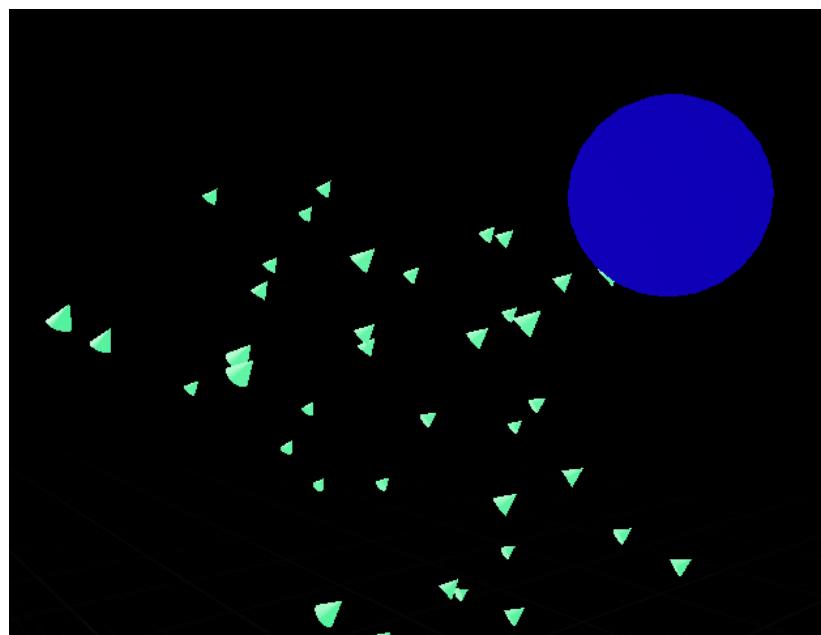


(a) Boidy zmierzają w kierunku celu



(b) Boidy się rozpraszają

Po kliknięciu na prawym kontrolerze spustu możemy zmienić położenie celu do którego mogą zmierzają boidy.



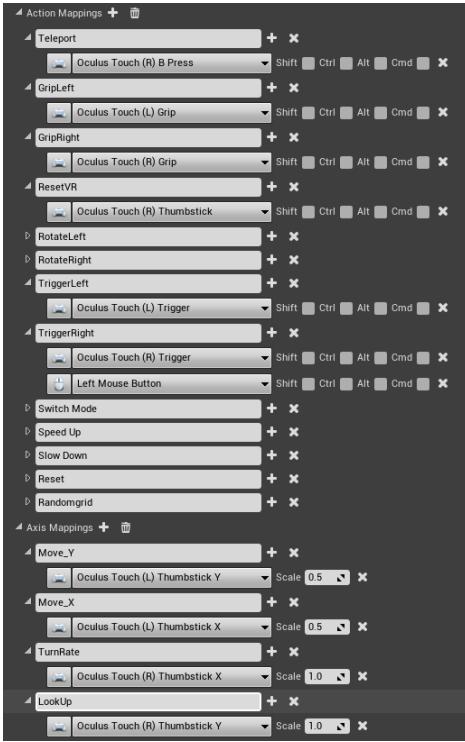
Rysunek 36: Przesuwanie celu w programie

5 Nie realizacja, a co było jeszcze robione - tytuł do wymyślenia

Jedną z pierwszych rzeczy jaką zrobiłem w grze była klasa odpowiedzialna za poruszanie się postaci. Klasa AVRCharacter jest dzieckiem klasy ACharacter. Klasa jest "pionkiem", którym może sterować gracz. Jest fizyczną reprezentacją gracza na poziomie z wbudowaną informacją o meshu, kolizji i logice ruchu. W klasie tej znajdują się wszystkie informacje potrzebne postaci w tym informacja o klasie kontrolerów ruchu, które w zależności od mapy na którym są modele, różnią się miedzy sobą. Metoda SetupPlayerInputComponent, która jest metodą wirtualną, jest wywołana w celu powiązania metod klasy z przyciskami, które zdefiniowaliśmy w ustawieniach projektu.

```
1 void AVRCharacter::SetupPlayerInputComponent(UInputComponent*  
      PlayerInputComponent)  
2 {  
3     Super::SetupPlayerInputComponent(PlayerInputComponent);  
4  
5     PlayerInputComponent->BindAxis(TEXT("Move_Y")) , this , &AVRCharacter::  
      MoveForward);  
6     PlayerInputComponent->BindAxis(TEXT("Move_X")) , this , &AVRCharacter::  
      MoveRight);  
7     PlayerInputComponent->BindAxis(TEXT("TurnRate")) , this , &AVRCharacter::  
      TurnAtRate);  
8     PlayerInputComponent->BindAxis(TEXT("LookUp")) , this , &AVRCharacter::  
      LookUpAtRate);  
9     PlayerInputComponent->BindAction(TEXT("Teleport")) , IE_Pressed , this , &  
      AVRCharacter::SetupTeleport);  
10    PlayerInputComponent->BindAction(TEXT("Teleport")) , IE_Released , this , &  
      AVRCharacter::BeginTeleport);  
11    PlayerInputComponent->BindAction(TEXT("GripLeft")) , IE_Pressed , this , &  
      AVRCharacter::GripLeft);  
12    PlayerInputComponent->BindAction(TEXT("GripLeft")) , IE_Released , this , &  
      AVRCharacter::ReleaseLeft);  
13    PlayerInputComponent->BindAction(TEXT("GripRight")) , IE_Pressed , this , &  
      AVRCharacter::GripRight);  
14    PlayerInputComponent->BindAction(TEXT("GripRight")) , IE_Released , this , &  
      AVRCharacter::ReleaseRight);  
15    PlayerInputComponent->BindAction(TEXT("TriggerRight")) , EInputEvent::  
      IE_Pressed , this , &AVRCharacter::RightTriggerPressed);  
16    PlayerInputComponent->BindAction(TEXT("TriggerRight")) , EInputEvent::  
      IE_Released , this , &AVRCharacter::RightTriggerReleased);  
17    PlayerInputComponent->BindAction(TEXT("TriggerLeft")) , EInputEvent::  
      IE_Pressed , this , &AVRCharacter::LeftTriggerPressed);  
18    PlayerInputComponent->BindAction(TEXT("TriggerLeft")) , EInputEvent::  
      IE_Released , this , &AVRCharacter::LeftTriggerPressed);  
19  
20    PlayerInputComponent->BindAction(TEXT("ResetVR")) , IE_Pressed , this , &  
      AVRCharacter::ResetVR);  
21    PlayerInputComponent->BindAction<FFooDelegate>(TEXT("RotateLeft")) ,  
      IE_Pressed , this , &AVRCharacter::Rotation,-30.f);  
22    PlayerInputComponent->BindAction<FFooDelegate>(TEXT("RotateRight")) ,  
      IE_Pressed , this , &AVRCharacter::Rotation,30.f);  
23 }
```

Listing 15: Przypisanie metod do przycisków



Rysunek 37: Przypisanie przycisków w projekcie

W projekcie można poruszać się na dwa sposoby. Pierwszy z nich polega na używaniu analoga na lewym kontrolerze do płynnego poruszania się po mapie w zależności od wychylenia analogu przy pomocy kciuka. Drugi sposób jest natomiast o wiele bardziej przyjemny do poruszania się dla osób mających chorobę symulatorową. A mianowicie teleportację do miejsca gdzie obecnie wskazuję znacznik celu. Znacznik aktywujemy poprzez kliknięcie przycisku na prawym kontrolerze. Do znalezienia miejsca w które się teleportowaliśmy użyłem kilku struktur silnikowych. FPredictProjectilePathParams do którego wstawiamy parametry wejściowe, są to:

- Promień naszej teleportacji używany podczas śledzenia kolizji
- Lokalizacja początku śladu teleportacji, jako punkt startowy przyjęty został prawy kontroler
- Początkowa prędkość startowa na początku śladu teleportacji, jako prędkości początkowa została przyjęty kierunek, który znajduje się na wprost kontrolera
- Maksymalny czas symulacji pocisku
- Kanał śledzenia używany do śledzenia kolizji

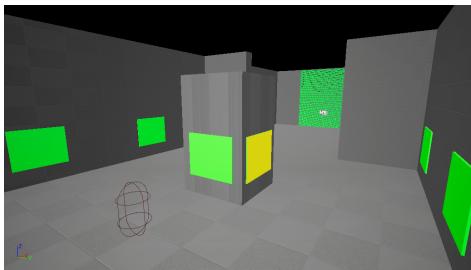
Kolejną strukturą jest FPredictProjectilePathResult, jest to kontener na wynik śledzenia ścieżki teleportacji, który dostajemy przy użyciu funkcji silnikowej PredictProjectilePath. Z FPredictProjectilePathResult możemy wyciągnąć informacje o tym, jak wygląda cała przewidziana ścieżka dla naszej teleportacji i miejsce w które uderzy. Dodatkowo sprawdzana jest jeszcze czy nasz punkt trafi w miejsce gdzie możemy spokojnie się teleportować poprzez sprawdzenie czy w miejscu teleportacji jest

navmesh. Jeśli tak, to poпусzczeniu przycisku na kontrolerze zostaniemy teleportowani w miejsce gdzie wskazaliśmy.

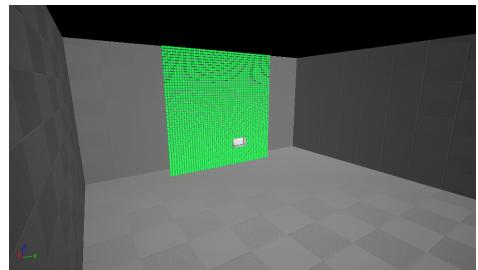
```
1 bool AVRCharacter::FindTeleportDestination(TArray< FVector>& OutPath,
2     FVector& OutLocation)
3 {
4     if (!RightController)
5     {
6         return false;
7     }
8     FVector Start = RightController->GetActorLocation();
9     FVector Look = RightController->GetActorForwardVector();
10
11     FPredictProjectilePathParams Params(TeleportProjectileRadius, Start,
12         Look * TeleportProjectileSpeed,
13         TeleportSimulationTime, ECC_Visibility, this);
14
15     FPredictProjectilePathResult Result;
16
17     bool bHit = UGameplayStatics::PredictProjectilePath(this, Params,
18             Result);
19
20     if (!bHit)
21     {
22         return false;
23     }
24
25     for (auto& PointData : Result.PathData)
26     {
27         OutPath.Add(PointData.Location);
28     }
29
30     FNavLocation NavLocation;
31
32     if (!UNavigationSystemV1::GetCurrent(GetWorld())->
33         ProjectPointToNavigation(Result.HitResult.Location, NavLocation,
34         TeleportProjectionExtent))
35     {
36         return false;
37     }
38
39     OutLocation = NavLocation.Location;
40
41     return true;
42 }
```

Listing 16: Metoda do znajdowania miejsca gdzie będziemy się teleportować

Ostatnie co zostało stworzone do projektu to pokoje (mapy) na których zostały umieszczone modele. Każdy pokój został podzielony na dwa segmenty. W pierwszym segmencie użytkownik może przeczytać informację o modelu na którego poziomie się znajduję. W drugim segmencie może zobaczyć jak dany model działa oraz uruchomić dany model.



(a) Pierwszy segment pokoju



(b) Drugi segment pokoju

Elementy pokoju zostały stworzone w programie do grafiki 3D Blender3D. Jest to darmowy program z którego w łatwy sposób można wrzucać stworzone tam meshy do UE4. Natomiast ściany, podłoga i sufit zostały stworzone przy pomocy meshy znajdujących się już w silniku UE4.

Obsługa kontrolerów została stworzona przy użyciu zasad dziedziczenia z klasy bazowej i metod wirtualnych. Każdy kontroler dziedziczy z klasy bazowej metody odpowiedzialne za zachowanie spustów, grypa i drążka. W zależności co było potrzebne do osiągnięcia w modelu zachowania poszczególnych funkcji się różnią, a dzięki zasadą dziedziczenia, mogę je wywołać w tych samych miejscach w kodzie, podmieniając tylko klasy kontrolerów.

Tutaj napisz informację z funkcji do teleportacji. Potem dodaj informację o kontrolerach ruchu, a dokładniej ich klasach. Na końcu napisz krótko o poziomach z jednym poziomem który został storzony. co się znajduje i jak go zrobiłeś. Na koncu przere daguj i wyślij matyce do poprawy. Potem popraw pierwsze trzy rozdziały i w 4 zrób poprawki których potrzebują

6 Wnioski

Podczas pracy nad projektem, jako osoba go robiąca doszczętnie kilku wniosków na bazie mojego doświadczenia z tego wynikająca:

- Najlepsza cecha blueprintów: pozwala nam zacząć kodować logikę gier wideo bez żadnej wiedzy programistycznej. Najgorsza cecha blueprintów: pozwala nam zacząć kodować logikę gier wideo bez żadnej wiedzy programistycznej.
- Dobrze napisana dokumentacja do programu to rzadkość, a UE4 niestety takowej dobrze napisanej nie posiada.
- Do projektowania grafiki 3D trzeba mieć odpowiednie umiejętności i czas by stworzyć coś ładnego, a zarazem optymalnego dla VR.
- Wybrany temat nauczył mnie tworzenia ciekawych modeli, które w przyszłości można rozbudować o nowe funkcje lub dodać kolejne modele do galerii.
- Tworzenie projektu z myślą o VR uczy prób pisania optymalnego kodu modeli, aby te podczas ich oglądania nie powodowały choroby symulatorowej spowodowanej niską ilością klatek na sekundę.

Literatura

- [1] VRcompare. VR Devices.
<https://vr-compare.com>.
- [2] Steam. Ankieta używanego sprzętu na steam.
<https://store.steampowered.com/hwsurvey>.
- [3] Ben Lang. Quest App Lab Soars to 1,440 Apps, More Than Tripling Those on the Main Store.
<https://www.roaddtovr.com/oculus-quest-app-lab-100-apps-best-rated-most-popular/>
- [4] Valve. Steam VR Game List.
<https://store.steampowered.com/vr>.
- [5] Michał Brzeżański. Choroba Symulatorowa.
<https://vrpolska.eu/skad-sie-bierze-choroba-symulatorowa/>.
- [6] James A. Blackburn Gregory L. Baker. The Pendulum A Case Study in Physics.
https://www.academia.edu/34415707/The_pendulum_A_case_study_in_physics_pdf.pdf.
- [7] Krzysztof Jankowski. Dynamics of double pendulum with parametric vertical excitation.
https://web.archive.org/web/*/http://www.team.kdm.p.lodz.pl/master/Jankowski.pdf/.
- [8] Encyklopedia PWN. Automat Komórkowy.
<https://encyklopedia.pwn.pl/haslo/;3872571>.
- [9] Conway's Game of Life.
https://conwaylife.com/wiki/Conway's_Game_of_Life.
- [10] Martin Gardner. The fantastic combinations of John Conway's new solitaire game "life".
<https://web.stanford.edu/class/sts145/Library/life.pdf>.
- [11] Staffan Björk i Jesper Juul. Zero-Player Games.
<https://www.jesperjuul.net/text/zeroplayergames/>.
- [12] sweyns. GameOfLife.
<https://github.com/sweyns/GameOfLife>.
- [13] Paul Halpern. O motylach i burzach.
https://web.archive.org/web/20100909161235/http://czytelnia.onet.pl/0,1161316,do_czytania.html/.
- [14] Jarosław Demkowski. Efekt motyla i dziwne atraktory.
<https://silo.tips/download/efekt-motyla-i-dziwne-atraktory>.
- [15] Maciej Matyka. Jak narysować Motyle Lorenza .
<https://www.youtube.com/watch?v=XZ5QKKxHTXQ/>.

- [16] Bassel Karami. Intro to Agent Based Modeling.
[https://towardsdatascience.com/intro-to-agent-based-modeling-3eea6a070b72.](https://towardsdatascience.com/intro-to-agent-based-modeling-3eea6a070b72)
- [17] DonHopkins. Boids Name.
[https://news.ycombinator.com/item?id=22710201.](https://news.ycombinator.com/item?id=22710201)
- [18] Wayne Carlson. Flocking Systems.
<https://ohiostate.pressbooks.pub/graphicshistory/chapter/19-2-flocking-systems/>.
- [19] Craig Reynolds. Boids.
<https://www.red3d.com/cwr/boids/>.
- [20] Darman1136. UE4Boids.
<https://github.com/Darman1136/UE4Boids>.

kolor blue: rozpisać

kolor red: edytować i może dodać

kolor green: wymyślić co dodać kolor purple: własne notatki z cyklu wszystko i nic