

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 sierpnia 2022

Spis treści

1 Wstęp	5
1.1 Wprowadzenie	5
1.2 Cel i zakres pracy	6
2 Warstwa Użytkowa	7
2.1 Wygląd i Obsługa programu	7
2.2 Cześć Galerii Programu	7
2.3 Cześć pokazowa modeli programu programu	7
3 Warstwa Programistyczna	7
3.1 Unreal Engine 4	7
3.2 Język C++	8
3.3 Oculus Quest	9
4 Modele wykorzystane w programie	10
4.1 Wahadło Podwójne	10
4.1.1 Wahadło podwójne - kod	10
4.1.2 Wahadło podwójne - UE4	13
4.1.3 Wahadło podwójne - Wygląd symulacji w projekcie	14
4.2 Gra w życie	15
4.2.1 Gra w życie - kod	16
4.2.2 Gra w życie - UE4	17
4.2.3 Gra w życie - Wygląd symulacji w projekcie	19
4.3 Efekt Motyla	20
4.3.1 Efekt Motyla - kod	21
4.3.2 Efekt Motyla - UE4	21
4.3.3 Efekt Motyla - Wygląd symulacji w projekcie	23
4.4 Modele Agentowe	24
4.4.1 Modele Agentowe - kod	24
4.4.2 Modele Agentowe - UE4	27
4.4.3 Modele Agentowe - Wygląd symulacji w projekcie	29
4.5 Boids	31
4.5.1 Boids - kod	31
4.5.2 Boids - UE4	33
4.5.3 Boids - Wygląd symulacji w projekcie	35
5 Realizacja projektu	36
6 Wnioski	37

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 pewnych aspektach życia człowiek zastanawia się nad paroma rzecząmi, czy nie jesteśmy sami w kosmosie, kiedy nastąpi koniec, czy VR umarł, czy nie jesteśmy programem komputerowym. Pewnie nie poznamy odpowiedzi na te wszystkie pytania, jeszcze przez jakiś czas, ale dzisiaj jedno jest pewne. VR na pewno jeszcze nie umarł i ma się całkiem dobrze. W ciągu ostatnich kilku lat rynek gogli VR zaczął na nowo się rozwijać, powstało wiele gogli, a do najpopularniejszych z nich należą: PlayStation VR, Valve Index, HTC Vive Pro [1]. Każde z tych gogli ma jednak wady, a do najważniejszych należy to, że nie są to sprzęty typu plug and play. Trzeba się nie tylko męczyć z splątaniną przewodów, ale także gogle jak np. Valve Index wymagają stacji, dzięki którym gogle wiedzą gdzie znajdujesz się przestrzeni 3D. Kolejnym problemem jest oczywiście cena samego sprzętu który w większości przekracza ponad 3000 PLN za całość. Jedynie co się z tego zestawu różni to PlayStation VR, same są jednak przestarzałe, a Sony zapowiedziało ich następcę którego premiera nastąpi pod koniec 2022 roku [2]. Same gogle Sony nie rozwiązały dla mnie największego problemu, czyli obowiązek podłączenia przewodem, jednakże ten problem na szczęście rozwiązała już inna firma, mowa oczywiście o Oculus znaną obecnie jako Reality Labs, jedna z podfirm Facebooka obecnie znana jako Meta. Firma zaczęła sprzedawać w 2019 gogle Oculus Quest, które były rewolucyjne z jednego ważnego powodu, były autonomiczne, tzn. nie potrzebowały komputera do obsługi gogli, ponieważ wystarczą do tego same gogle z kontrolerami ruchowymi. Same gogle nie potrzebowały też stacji do określania położenia gogli, gdyż same w sobie mają diody podczerwone które do tego służą. Oculus Quest okazał się dość rewolucyjnym sprzętem wartym ok. 2000 pln za wersję podstawową, a rok później Oculus wypuścił następcę za którego zapłaciliśmy jeszcze mniej czyli ok. 1500 pln w wersji podstawowej. Nie odbyło się to bez kompromisów takich jak: Brak płynnej zmiany rozstawu soczewek dla oczu, gorszej jakości pasek na głowę, brak magnetycznego zabezpieczenia pojemnika na baterię. To nie znaczy oczywiście, że gogle były gorsze a do najważniejszych należą: Zwiększoną roździełość obrazu dla jednego oka, zmiana procesora na wydajniejszy, wydłużony czas pracy kontrolerów na jednej baterii [3].



(a) Oculus Quest 1



(b) Oculus Quest 2

Rysunek 1: Gogle VR od Oculusa

Quest 2 wśród gogli VR okazał się dużym sukcesem, w roku 2021 sprzedanych

zostało 11,2 miliona sztuk urządzeń z czego 78% to były Oculus Quest 2 [4]. Rynek aplikacji też się rozwinął w ciągu ostatnich lat. Na samego Oculusa Questa w oficjalnym sklepie jest obecnie dostępnych ponad 300 aplikacji, na Steam jest ich już ponad 2000. Oczywiście wśród nich nie znajdują się same gry, ponieważ gogle mogą być wykorzystane też np. do zaprezentowania ciała człowieka, być platformą do rysowania obrazów, lub sprzętem do relaksu czy oglądania filmów. Ja chciałem spróbować swoich sił w stworzeniu małego projektu, który mógłby być wykorzystany do pokazywania ciekawego można robić w komputerze, czyli galerii modelów komputerowych.



Rysunek 2: OpenBrush

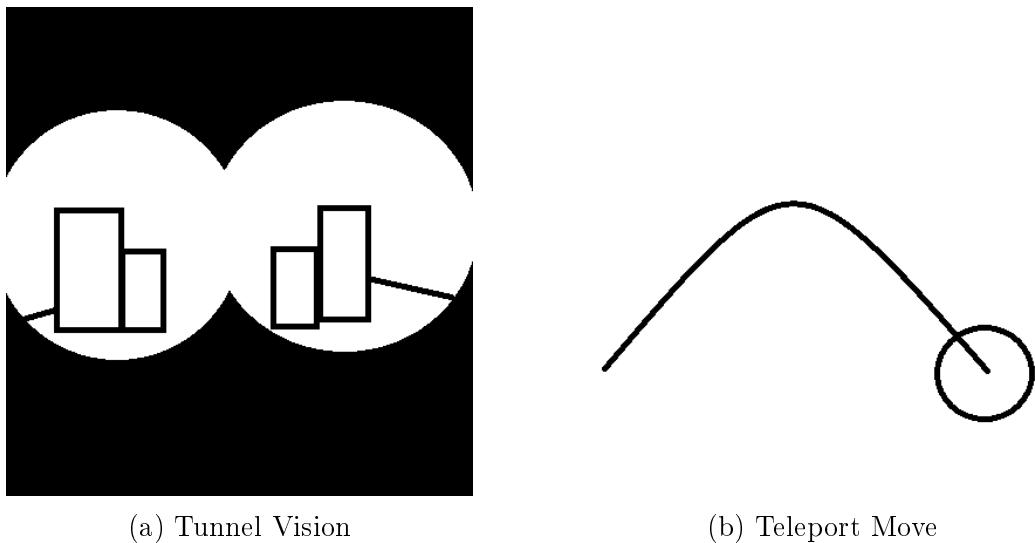
1.2 Cel i zakres pracy

Główym założeniem jest stworzenie aplikacji VR która przedstawi kilka wybranych przez mnie modeli komputerowych w przystępny sposób, łącznie z częścią galerii w której znajdować się będzie historia danego modelu i ciekawostki z nim związane. Google VR których używałem do testów to Oculus Quest pierwszej generacji, natomiast aplikację napisałem w Unreal Engine 4, ponieważ moge w tym silniku pisać w języku C++, który to najbardziej z języków programowania znam, oraz w blueprintach czyli Unrealowym wizualnym języku skryptowym, który jest przystępny dla nowych użytkowników. Silnik posiada pełne wsparcie dla gogli VR czy to wersji autonomicznej czy wersji PCVR. Sam aplikację pisałem z myślą o PCVR, ponieważ nie musiałem się aż tak obawiać o ograniczenia które stawia sprzęt w wersji androidowej np. brak wsparcia dla Unrealowych postprocesów obrazu. Projekt ten pokazuję, że w dzisiejszych czasach dzięki dostępnym narzędziom typu UE4 i gogle VR, człowiek jest w stanie stworzyć program w mało wymagający sposób który nie byłby możliwy do zrealizowania jeszcze 10 lat temu.

2 Warstwa Użytkowa

2.1 Wygląd i Obsługa programu

Program do uruchomienia wymaga gogli VR np. Oculus Quest i kontrolerów ruchowych. Można poruszać się po planszy na dwa sposoby. Pierwszy polega na użyciu lewego thumbsticca dzięki czemu można poruszać się płynnie po planszy, podczas poruszania się po planszy zmniejsza się obraz aby osoby które mają chorobę symulatorową[5] lepiej znosiły takie poruszanie się po poziomie. Drugi natomiast polega na teleportacji, po naciśnięciu przycisku B na prawy kontrolerze i wybraniu miejsca planszy gdzie chcemy się teleportować, jest to sposób poruszania się bardziej przyjazny dla osób z chorobą symulatorową.



(a) Tunnel Vision

(b) Teleport Move

Rysunek 3: Różny rodzaj poruszania się w programie

2.2 Część Galerii Programu

W tej części można poczytać i dowiedzieć się trochę o danym modelu. Swoim wyglądem przypomina to galerię obrazów, gdzie osoba podchodzi do danej ciekawostki i może ją przeczytać.

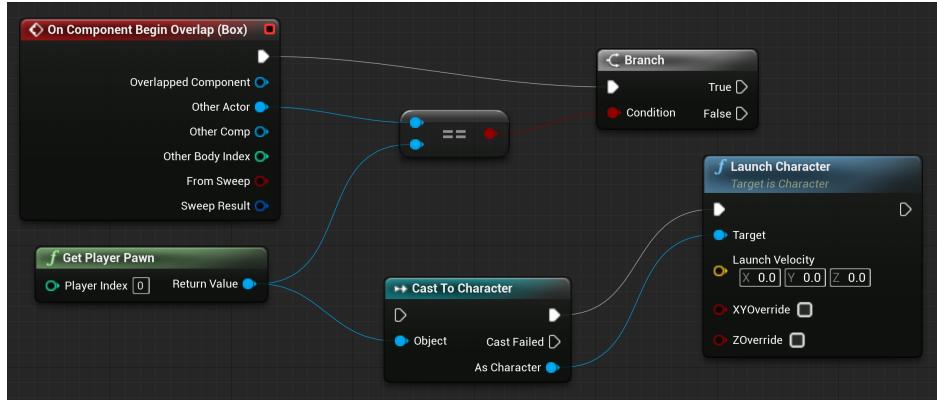
2.3 Część pokazowa modeli programu programu

Ta część galerii ma pokazać użytkownikowi programu w jaki sposób wygląda dany model w działaniu. Modele obsługuję się za pomocą kontrolerów ruchowych. W zależności od modelu interakcja wygląda trochę inaczej, szczegółowo każda zostanie omówiona w sekcji dotyczącej danego modelu.

3 Warstwa Programistyczna

3.1 Unreal Engine 4

Program był pisany w silniku Unreal Engine 4, z powodu że korzystam z tego silnika na codzień w swojej pracy. Sam silnik wspiera też bez większym problemów gogle VR, dzięki pluginowi stworzonemu przez twórców silnika. UE4 jest silnikiem wszechstronnym, czyli nie musi być wykorzystywany tylko do tworzenia gier. Sam silnik jest dość popularny wśród ludzi co oznacza, że w i nie tylko można znaleźć ogrom materiałów do pomocy przy projekcie. UE4 posiada wbudowany język skryptowy zwany Blueprint, opiera się głównie bloczkach które łączą się miedzy sobą. Głównym celem BP jest chęć zdobycia widowni wśród ludzi którzy na codzień nie programują i wolą patrzeć na coś milszego dla oka. Nody są podobne do tych wykorzystywanych w blenderze. Projekt w UE4 nie musi się opierać tylko na nodach, można też większość rzeczy pisać w C++.



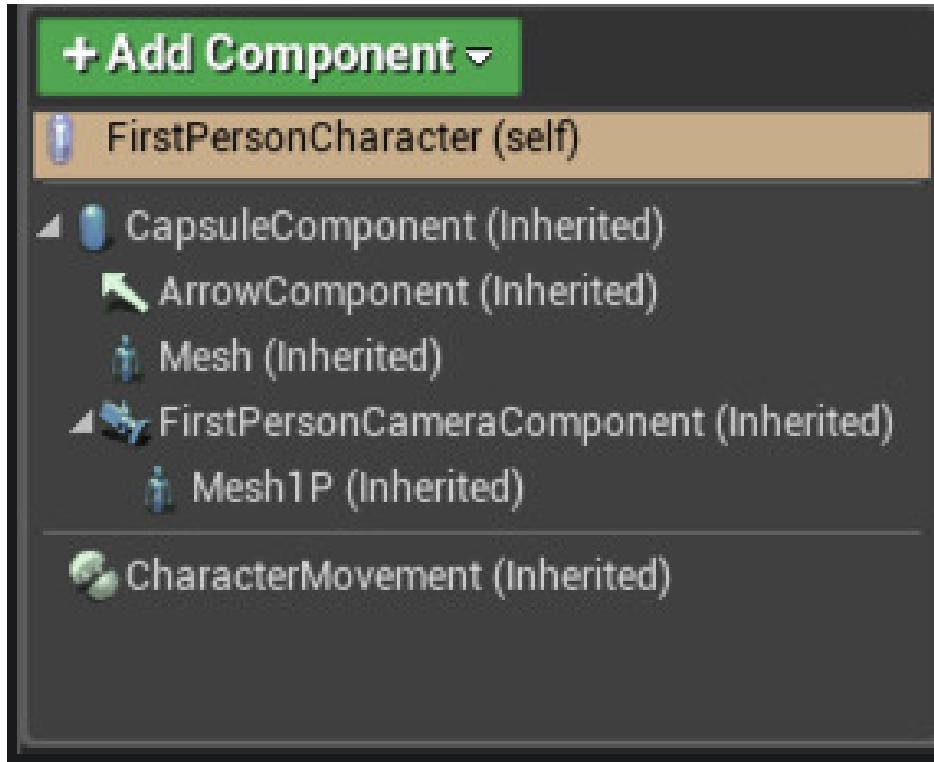
Rysunek 4: Przykładowy Blueprint w UE4

Sam w silnik pozwala też na tworzenie poziomów w prosty i intuicyjny sposób, poprzez przeciąganie interesującego nas obiektu bezpośrednio na ekran a następnie, na jego obracanie czy nachylenie. W edytorze możemy też uruchomić interesujący na poziom i zobaczyć czy wszystko działa tak jak należy. Możemy też edytować materiał dla meshy na poziomie w podobny sposób jak tworzymy klasy aktorów, czyli poprzez nody.

3.2 Język C++

UE4 pozwala też pisać klasy i funkcję w języku C++, a nie tylko w BP. Pisanie rzeczy C++ w UE4 nie różni się wiele od pisania w czystym C++, zaleca się oczywiście pisania rzeczy odwołujących się do biblioteki unreala, co nie znaczy, że nie można niektóre pisać czysto w C++. Można oczywiście daną zmienną w klasie trzymać tylko w kodzie źródłowym, jeśli chcemy się odwołać do edytora UE4 i tam też coś robić musimy użyć specyfikator UPROPERTY przed zmienną i UFUNCTION przed funkcją. Specyfikatory mają też dostępne opcje dzięki którym możemy np. edytować daną zmienną w edytorze lub podczas działania aplikacji. Najważniejszymi funkcjami które dziedziczymi po klasach z biblioteki UE4 to Begin Play, Tick i EndPlay. BeginPlay uruchamia się kiedy dany aktor zostaje wywołany w czasie gry podczas spawnu. Tick jest wywoływany w każdym ticku życia aktora na poziomie. EndPlay jest wywoływany kiedy aktor kończy swój żywot. Danego aktora możemy zrobić czysto w C++ i tak samo go spawnować na levelu, ale lepiej jest stworzyć klasę dziedziczącą po nim w edytorze, staje się wtedy ona klasą BP którą możemy rozbudować o nowe funkcje. Jeśli

w klasie C++ stworzymy odwołanie do actor componenta, specjalna funkcje (rozpisać co to), w klasie bp będzie widoczna jako odziedziona po klasie C++.



Rysunek 5: W nawiasie informacja, że odziedziczone z po klasie bazowej

C++ pod względem działania funkcji jest szybszy od funkcji BP, więc z doświadczenia zalecam pisania głównie w nim i dopiero w mniejszym stopniu tworzyć jakieś małe funkcje pomocnicze w BP.

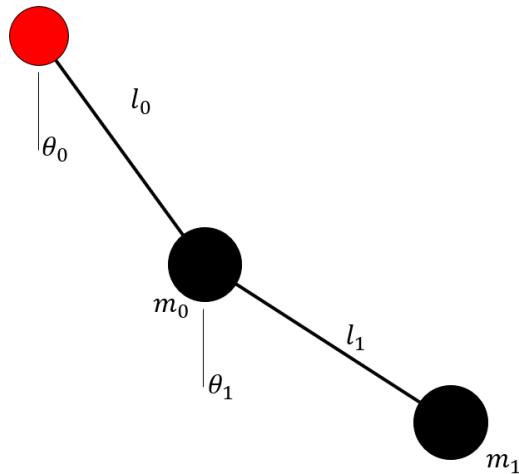
3.3 Oculus Quest

Gogle VR z których korzystam to Oculus Quest 1. Aby komunikować się z nimi korzystam z obrogramowania storzonym przez firmę Oculus, dzięki niemu mogę obsługiwać gogle na komputerze przez sieć WiFi, a nie przez podłączenie gogli po kablu do komputera. Gogle te nie potrzebują do pracy komputera, ponieważ są w pełni autonomiczne, cała technologia skrywa się w samych goglach, dzięki czemu nie są potrzebne stacje bazowe. Aby tworzyć programy na goglach nie trzeba się specjalnie męczyć, trzeba oczywiście stworzyć sterowania i komunikację ze światem wirtualnym przystosowanym specjalnie pod sterowanie ruchowe, oraz dostosowanie grafiki, aby ilość FPS nie spadała poniżej 72 i utrzymywała je stale na tej wartości, dzięki temu nie ma większych problemów z chorobą symulatorową dla większości osób

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 6: 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.

```

1 void APendulum::computePosition()
2 {
3     x0 = px + length0 * FMath::Sin(theta0);
4     y0 = py + length0 * FMath::Cos(theta0);
5
6     x1 = x0 + length1 * FMath::Sin(theta1);
7     y1 = y0 + length1 * FMath::Cos(theta1);
8
9     auto Frector = GetActorLocation();
10    FVector StarPos1 = { Frector.X, px, py, };
11    FVector EndPos1 = { Frector.X, x0, y0, };
12    FVector EndPos2 = { Frector.X, x1, y1, };
13
14    TArray< FVector> Path;
15    Path.Add(StarPos1);
16    Path.Add(EndPos1);
17    Path.Add(EndPos2);
18
19    FirstColumn->ClearSplinePoints(false);
20    int32 index = 0;
21    for (auto& Point : Path)
22    {
23        FVector LocalPosition = FirstColumn->GetComponentTransform().InverseTransformPosition(Point);
24        FirstColumn->AddPoint(FSplinePoint(index, LocalPosition, ESplinePointType::Constant), false);
25        index++;
26    }
27
28    FirstColumn->UpdateSpline();
29
30    int32 SegmentNum = Path.Num() - 1;
31    for (int32 i = 0; i < SegmentNum; ++i)
32    {
33        if (ColumnsPathMeshPool.Num() <= i)
34        {
35            USplineMeshComponent* SplineMesh = NewObject<USplineMeshComponent>(this);
36            SplineMesh->SetMobility(EComponentMobility::Movable);
37            SplineMesh->AttachToComponent(FirstColumn,
38                FAttachmentTransformRules::KeepRelativeTransform);
39            SplineMesh->SetStaticMesh(FirstColumnArchMesh);
40            SplineMesh->SetMaterial(0, FirstColumnArchMaterial);
41            SplineMesh->RegisterComponent();
42
43            ColumnsPathMeshPool.Add(SplineMesh);
44        }
45
46        USplineMeshComponent* SplineMesh = ColumnsPathMeshPool[i];
47        SplineMesh->SetVisibility(true);
48
49        FVector StarPos, StartTangent, EndPos, EndTangent;
50        FVector Tangent = { 0,0,0 };
51        FirstColumn->GetLocalLocationAndTangentAtSplinePoint(i, StarPos,

```

```

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

```

Listing 2: Aktualizacja pozycji wahadła

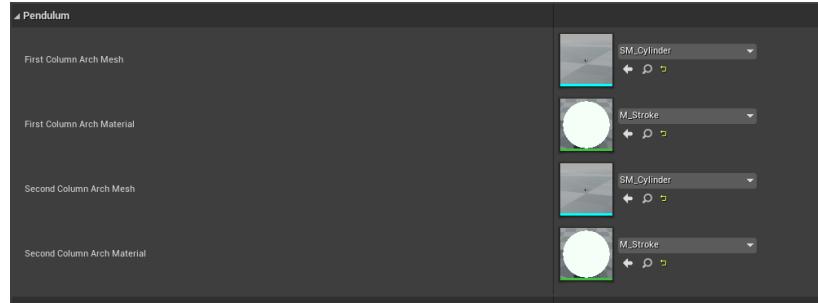
Klasa PendulumSpawn jest natomiast odpowiedzialna za dodawanie, usuwanie wahadł. Klasa jest też odpowiedzialna za uruchomianie i resetowanie wszystkich wahadł które zostały przez dany PendulumSpawn stworzone.

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

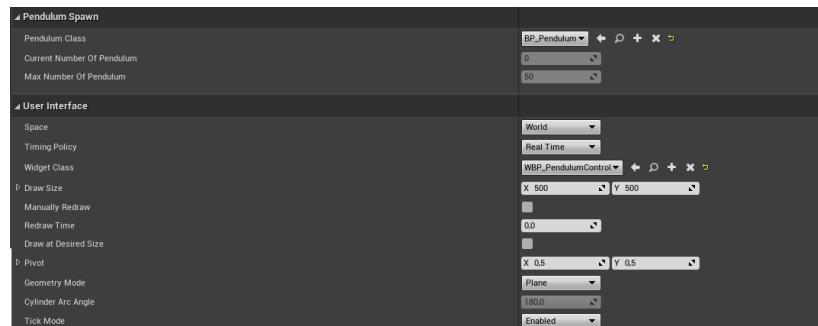
Na podstawie klas C++ z poprzedniego podrozdziału stworzyłem klasy blueprintowe, które to umieszczam na poziomie w edytorze lub są wykorzystane do stworzenia interfejsu użytkownika.

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



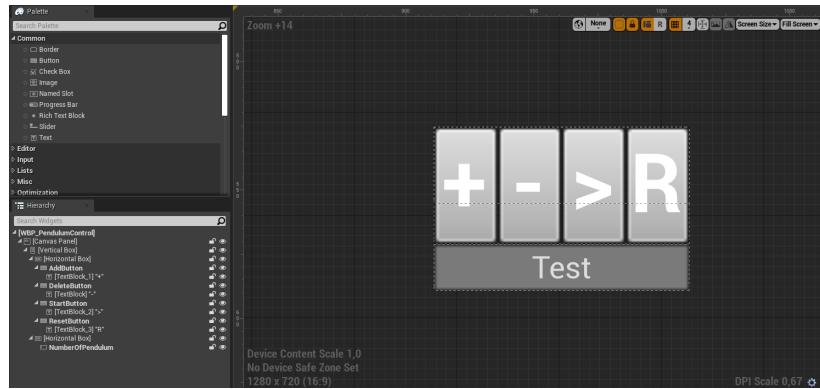
Rysunek 7: Ustawiony mesh i tekstura dla wahadła

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 8: Ustawione wahadło do stworzenia i widget

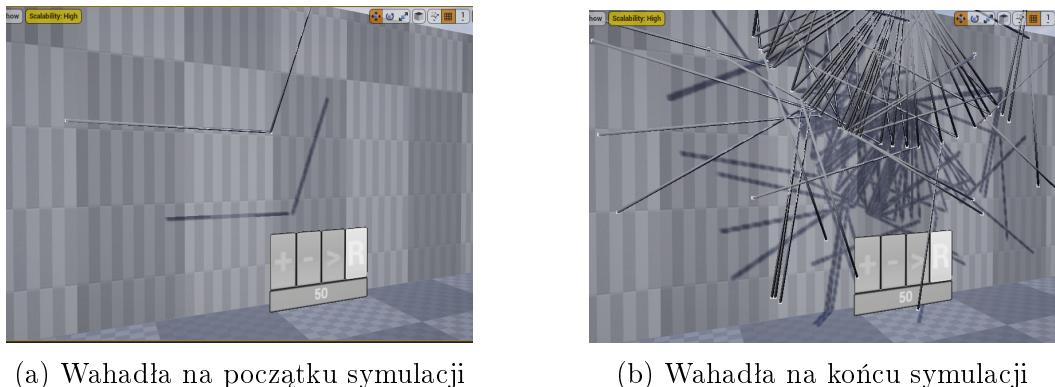
Klasa WBP_PendulumControl jest klasą typu widget, czyli klasą interfejsu użytkownika. W klasie tej trzeba stworzyć odpowiednie jej elementy na podstawie bazowej klasy C++, aby móc obsługiwać wahadła stworzone przez BP_PendulumSpawn



Rysunek 9: 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 10a, 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 10b.



Rysunek 10: Działanie programu

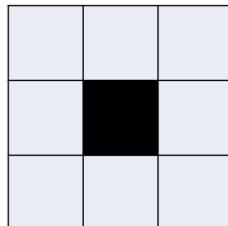
4.2 Gra w życie

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

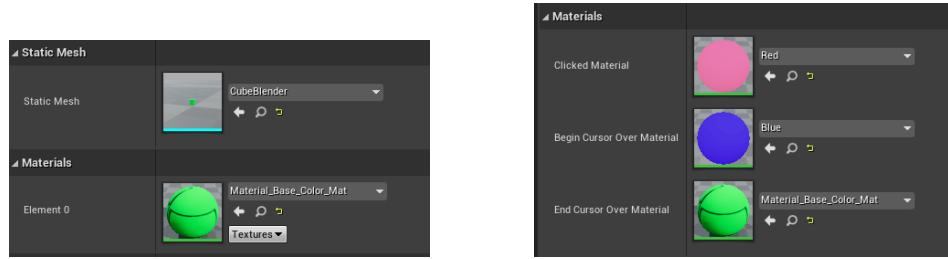
Listing 6: Aktualizacja stanu komórki w następnej klatce

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

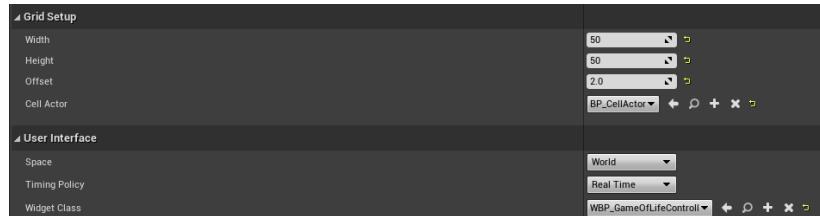
kontrolerem ruchowym:



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

Rysunek 12: Ustawienia dla CellActora w UE4

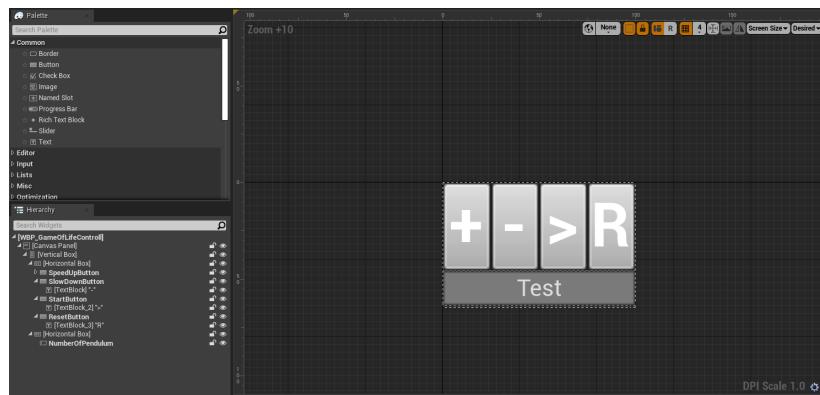
BP_GridActor2D jest klasą BluePrint która znajdzie się na poziomie gry. W klasie ustawiamy na jaką szerokość i wysokość ma zostać stworzona siatka składająca się z **BP_CellActor** oraz dodajemy widget, dzięki któremu możemy sterować symulacją:



Rysunek 13: Ustawienia BP_GridActor2D

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

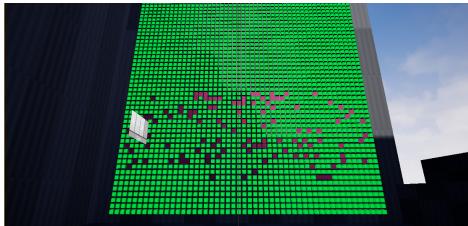
Klasa **WBP_GameOfLifeControll** jest odpowiedzialna kontakt między użytkowni-
kiem a programem, klasa ta jest widgetem dzięki któremu poprzez najechanie kontro-
lerem ruchowym na odpowiednie opcje możemy włączyć symulację, przyśpieszyć lub
ją spowolnić oraz ją zresetować do stanu początkowego:



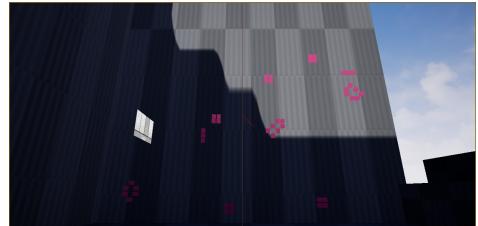
Rysunek 14: Wygląd widgetu w programie

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.

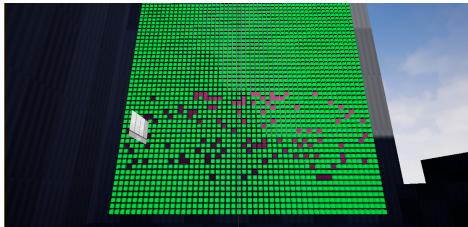


(a) Ożywianie komórek z pomocą kontrolera

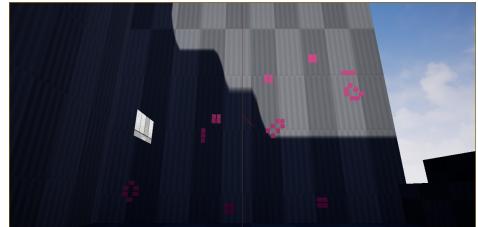


(b) Uśmiercanie komórek z pomocą kontrolera

Po lewej stronie od tablicy z komórkami znajduję się widżet, na którym możemy wyłączyć 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 17: 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.

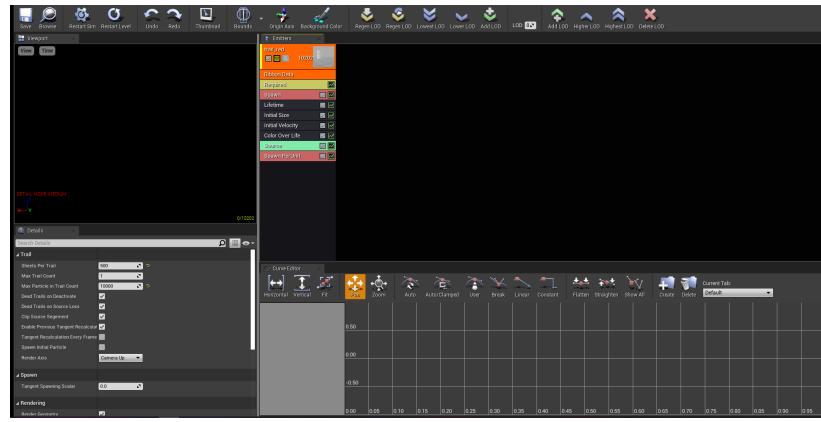
```

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 18: Wygląd menu do tworzenia efektów cząsteczkowych w UE4

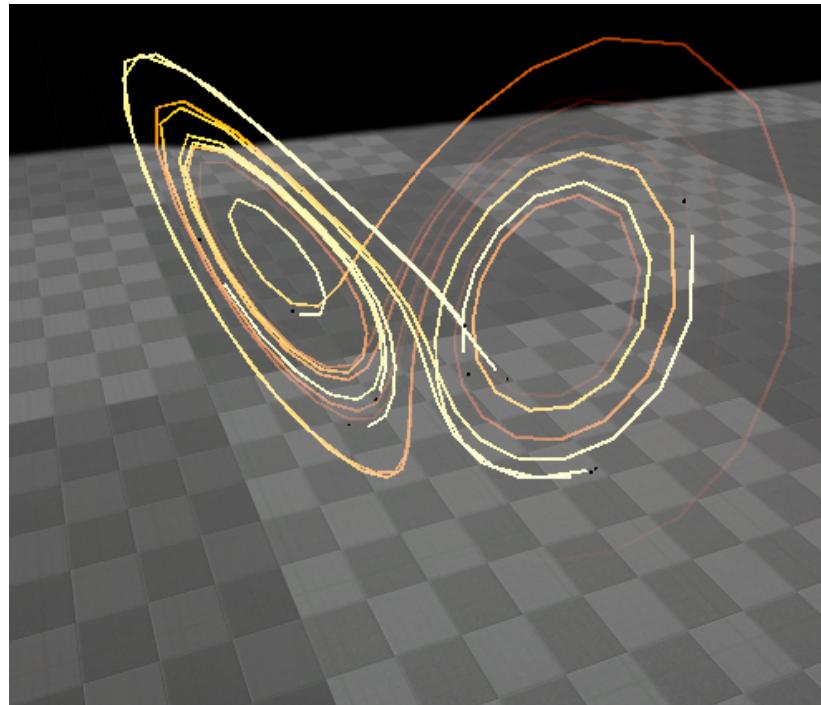
W BP _ButterflySpawner ustawiamy jakiego aktora chcemy ustawić na scenie, oraz ile motyli chcemy stworzyć.



Rysunek 19: 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 20: 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.

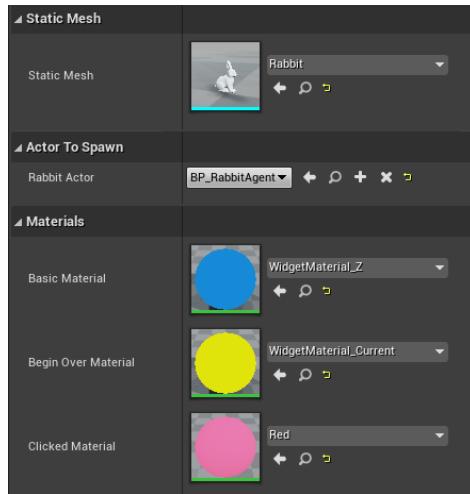
Kolejną klasą jest AgentSpawnBox. Podobnie jak AgentSpawner tworzy nowych agentów, ale z tą różnicą, że do pomocy potrzebuję gracza. W klasie tej nowy agent danego rodzaju jest tworzony, kiedy gracz wejdzie w interakcję z tym obiektem za pomocą kontrolera ruchowego. Dzięki temu, gracz może "wyciągnąć"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

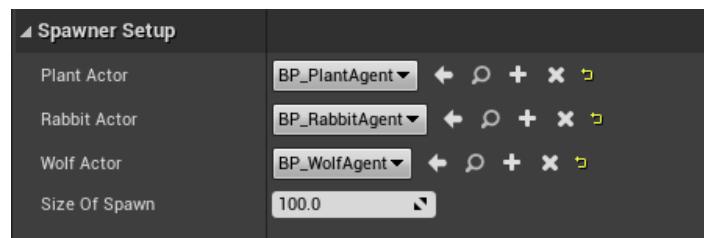
4.4.2 Modele Agentowe - UE4

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.



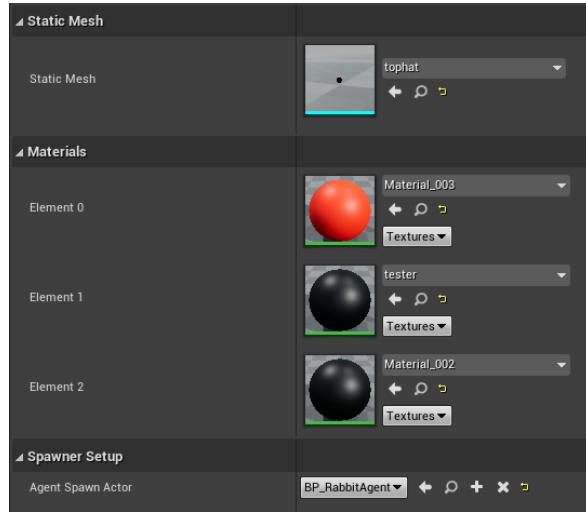
Rysunek 21: Ustawienia Agentów w UE4

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.



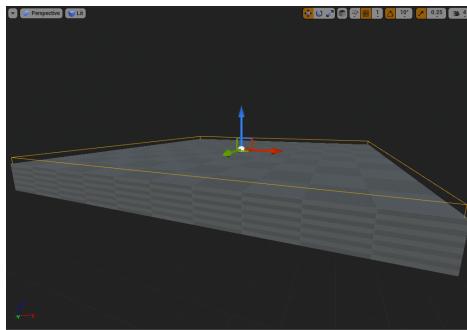
Rysunek 22: Ustawienia Spawnera w UE4

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.

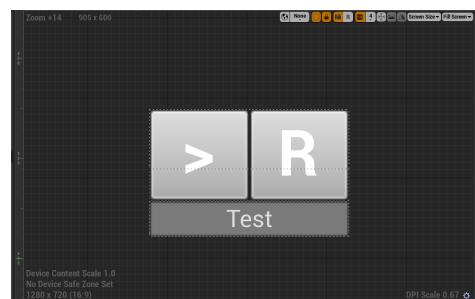


Rysunek 23: Ustawienia SpawnBox w UE4

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.



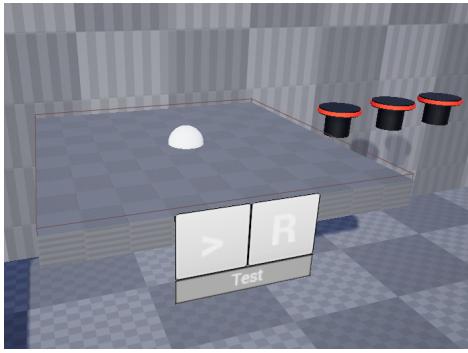
(a) Wygląd planszy w UE4



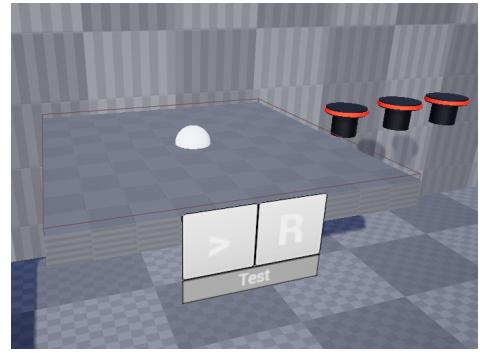
(b) Wygląd widgetu w UE4

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.

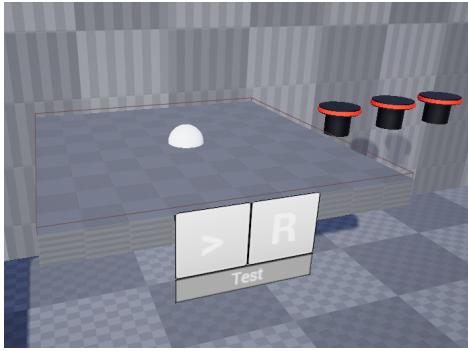


(a) SpawnBoxy w programie

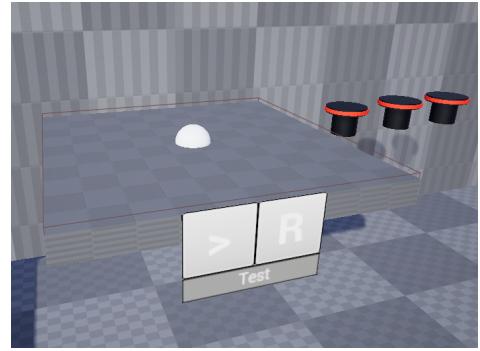


(b) Wyciąganie agenta z SpawnBoxu

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.

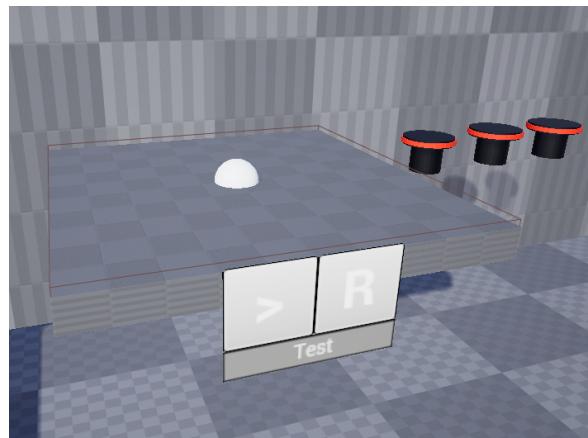


(a) Kładzenie agenta na planszy



(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.

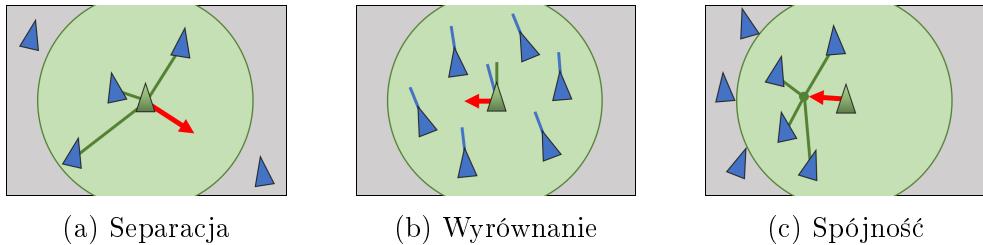


Rysunek 27: Wygląd symulacji w trakcie jej działania

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++) {
```

```

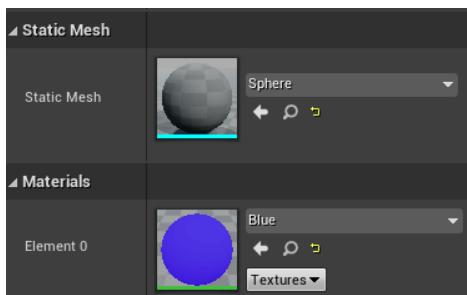
11 ABoid* Boid = CloseBoids[index];
12 CalculateAlignment(AlignmentVector, Boid);
13 CalculateCohesion(CohesionVector, Boid);
14 CalculateSeparation(SeparationVector, Boid);
15 }
16 AlignmentVector /= CloseBoids.Num();
17 CohesionVector /= CloseBoids.Num();
18 SeparationVector /= CloseBoids.Num();
19
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

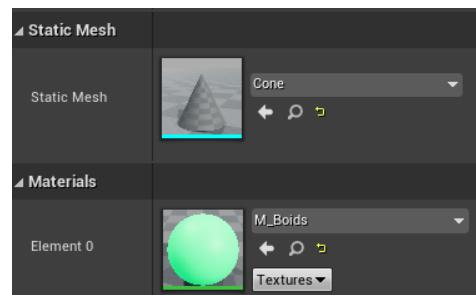
4.5.2 Boids - UE4

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



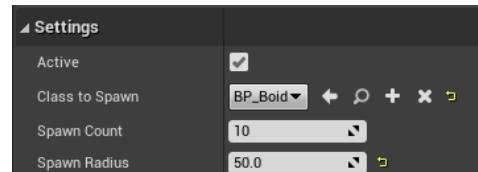
(b) Ustawienia Boid w UE4

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.

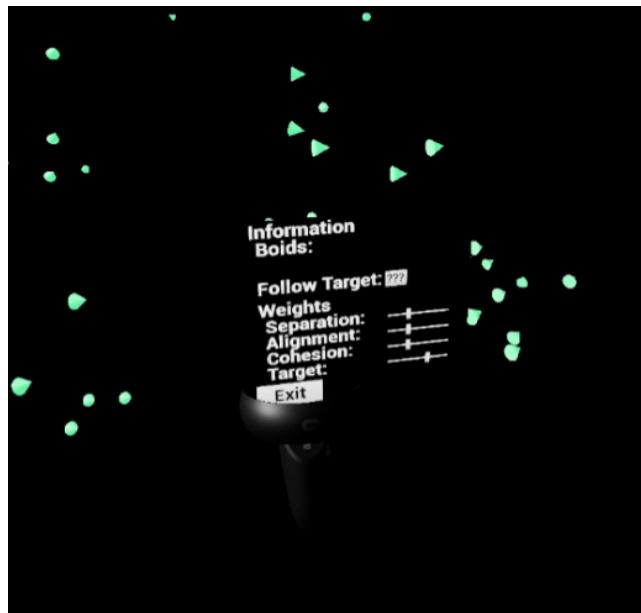


(a) Ustawienia BoidManager w UE4



(b) Ustawienia BoidSpawner w UE4

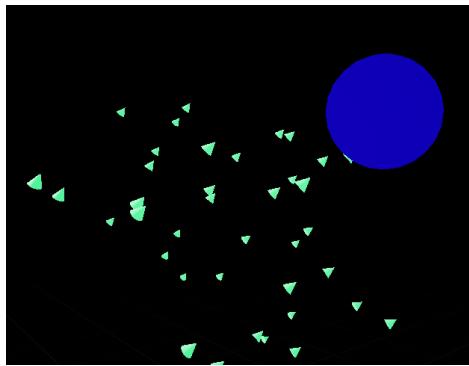
WBP_BoidHud tak jak wszystkie widgety w poprzednich symulacjach, po najechaniu kontrolerem ruchowym, możemy wybrać to czy boidy mają podążać za celem, czy się będą rozpraszać po okolicy. Możemy też wybrać z jaką siłą ma być wykonywana jedna z trzech zasad rządząca boidami, oraz jak blisko ma się zbliżyć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



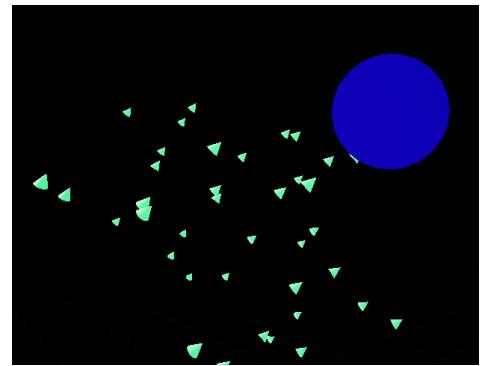
Rysunek 31: Wygląd widgetu do sterowania zachowaniem boidów

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.

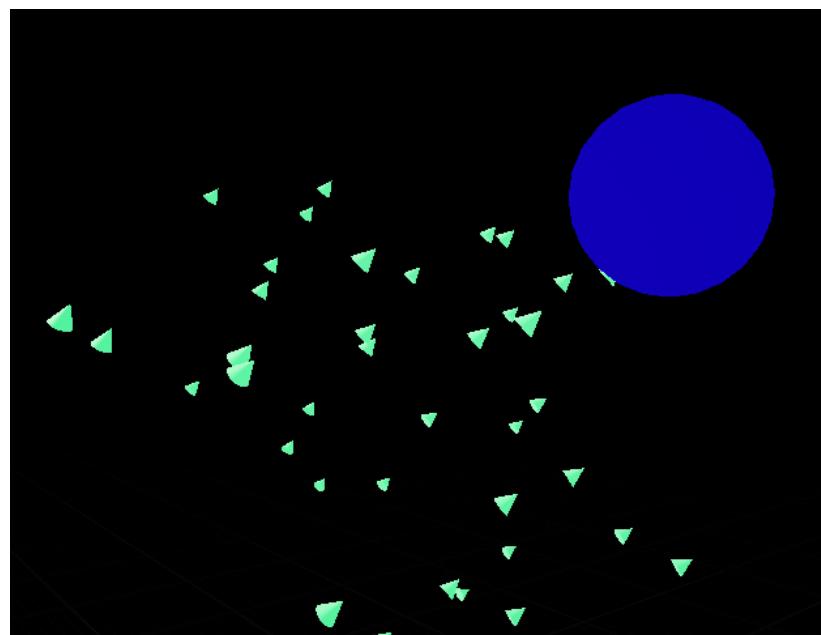


(a) Boidy zmierzają w kierunku celu



(b) Boidy się rozpraszają

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



Rysunek 33: Przesuwanie celu w programie

5 Realizacja projektu

6 Wnioski

Literatura

- [1] Steam. Ankieta używanego sprzętu na steam.
<https://store.steampowered.com/hwsurvey>.
- [2] Sony. Zapowiedź PlayStation VR2.
<https://www.playstation.com/pl-pl/ps-vr2/>.
- [3] VRcompare. Compare Oculus Quest and Oculus Quest 2.
<https://vr-compare.com/compare?h1=pDTZ02PkT&h2=GeZ01ojF8>.
- [4] IDC. AR/VR Headset Shipments Grew Dramatically in 2021.
<https://www.idc.com/getdoc.jsp?containerId=prUS48969722>.
- [5] Michał Brzeżański. Choroba Symulatorowa.
<https://vрpolska.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 atraktorysty.
<https://silo.tips/download/efekt-motyla-i-dziwne-atraktorysty>.
- [15] Maciej Matyka. Jak narysować Motyle Lorenza .
[https://www.youtube.com/watch?v=XZ5QKKxHTXQ/](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