

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, 26 maja 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 Jakie modele się znajdują	10
4.1 Wahadło Podwójne	10
4.1.1 Wahadło podwójne - kod	11
4.1.2 Wahadło podwójne - UE4	13
4.1.3 Wahadło podwójne - Wygląd w programie	14
4.2 Gra w życie	15
4.2.1 Gra w życie - kod	15
4.2.2 Gra w życie - UE4	17
4.2.3 Gra w życie - Wygląd w programie	18
4.3 Efekt Motyla	19
4.3.1 Efekt Motyla - kod	20
4.3.2 Efekt Motyla - UE4	20
4.3.3 Efekt Motyla - Wygląd w programie	21
4.4 Model Agentowy	22
4.4.1 Model Agentowy - kod	22
4.4.2 Model Agentowy - UE4	25
4.5 Boids	25
4.5.1 Boids - kod	25
5 Realizacja projektu	25
6 Wnioski	25

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

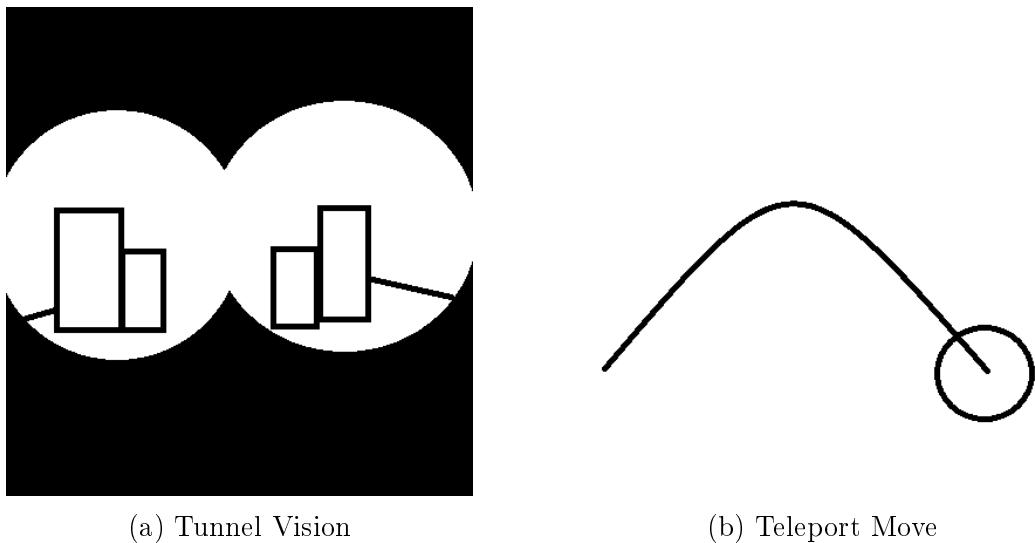
1.2 Cel i zakres pracy

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.

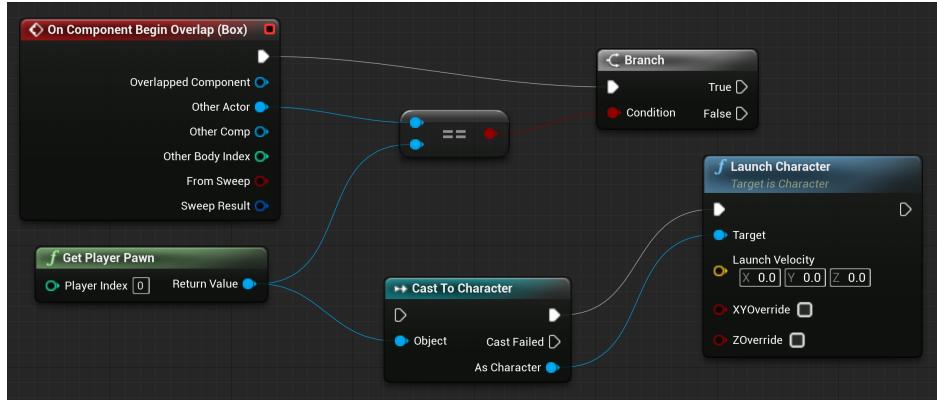
2.3 Część pokazowa modeli programu programu

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++.



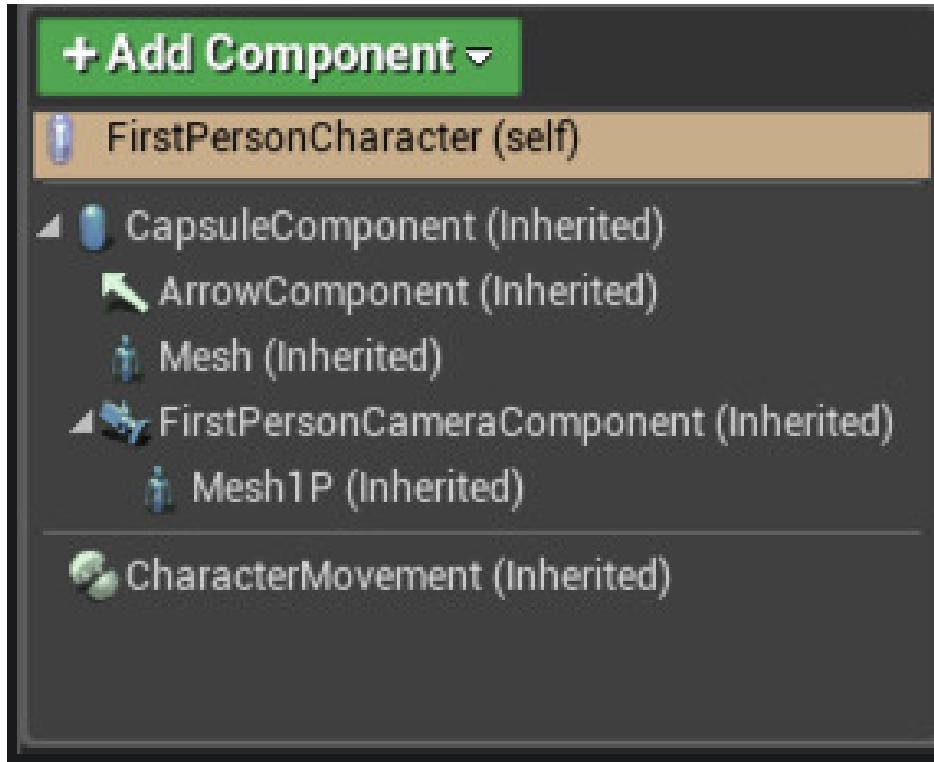
Rysunek 4: Przykładowy Blueprint w UE4

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

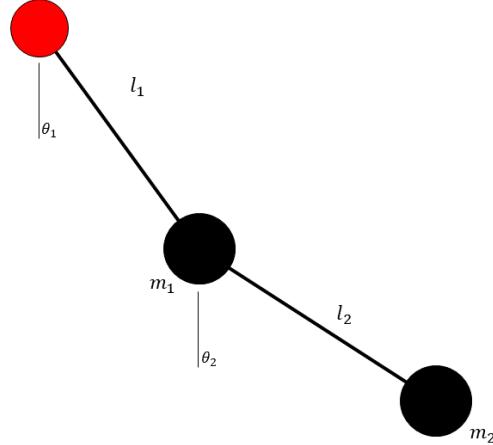
C++ pod względem działania funkcji jest szybszy od funkcji BP, więc z doświadczenia zalecam pisania głównie w nim i dopiero w mniejszym stopniu tworzyć jakieś małe funkcje pomocnicze w BP.

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 Jakie modele się znajdują

4.1 Wahadło Podwójne



Rysunek 6: θ 1-2 kąty swobodne, m_1 m_2 masy obiektów, l_1 l_2 długości pręta

Pierwszy model który umieściłem jest model wahadła podwójnego. Wahadło podwójne, jest wahadłem które ma przymocowane drugie wahadło na jego końcu. [6] W wahadle tym punkty masy są na końcu każdego pręta o danej długości. Wahadła obracają się swobodnie w płaszczyźnie pionowej. Energię potencjalną wahadła podwójnego oblicza się jako odniesienie uniesionego wahadła do stanu zerowej energii potencjalnej w położeniu równowagi, Dlatego równanie ma następującą postać:

$$V = m_1 g l_1 (1 - \cos \theta_1) + m_2 g [l_1 (1 - \cos \theta_1) + l_2 (1 - \cos \theta_2)] \quad (1)$$

W celu sformułowania równania energii kinetycznej wykorzystamy wyprowadzenie współrzędnych x i y układu, które przedstawiono poniżej:

$$\begin{aligned} x_2 &= l_1 \sin \theta_1 + l_2 \sin \theta_2 \\ y_2 &= l_1 \cos \theta_1 - l_2 \cos \theta_2 \end{aligned} \quad (2)$$

Po wyprowadzeniu i zsumowaniu wyrażeń możemy określić ostateczną postać energii kinetycznej:

$$K = \frac{1}{2} m_1 l_1^2 \dot{\theta}_1^2 + \frac{1}{2} m_2 l_1^2 \dot{\theta}_1^2 + \frac{1}{2} m_2 l_2^2 \dot{\theta}_2^2 + m_2 l_1 l_2 \cos(\theta_1 - \theta_2) \dot{\theta}_1 \dot{\theta}_2 \quad (3)$$

Najprostsze równanie Lagrange ma postać:

$$\frac{d}{dt} \left(\frac{\partial L}{d\dot{\theta}_i} \right) - \left(\frac{\partial L}{d\theta_i} \right) = 0 \quad (4)$$

Korzystając z równań Lagrange i dokonując odpowiednich podstawień, możemy sformułować sprzążone równanie ruchu dla podanego układu:

$$\begin{aligned}\ddot{\theta}_1 &= -\frac{l_2}{\mu l_1} \ddot{\theta}_2 \cos(\theta_1 - \theta_2) - \frac{l_2}{\mu l_1} \dot{\theta}_2^2 \sin(\theta_1 - \theta_2) - \frac{g}{l_1} \sin \theta_1 \\ \ddot{\theta}_2 &= -\frac{l_1}{l_2} \ddot{\theta}_1 \cos(\theta_1 - \theta_2) - \frac{l_1}{l_2} \dot{\theta}_1^2 \sin(\theta_1 - \theta_2) - \frac{g}{l_2} \sin \theta_2\end{aligned}\quad (5)$$

gdzie:

$$\mu = 1 + (m_1 + m_2) \quad (6)$$

Aby zdekomponować pojęcia przyspieszenia kątowego, podstawiamy je, wstawiając drugie równanie do pierwszego i odwrotnie, otrzymujemy:

$$\begin{aligned}\ddot{\theta}_1 &= \frac{g(\sin \theta_2 \cos(\theta_1 - \theta_2) - \mu \sin \theta_1) - (l_2 \dot{\theta}_2^2 + l_1 \dot{\theta}_1^2 \cos(\theta_1 - \theta_2)) \sin(\theta_1 - \theta_2)}{l_1(\mu - \cos^2(\theta_1 - \theta_2))} \\ \ddot{\theta}_1 &= \frac{g\mu(\sin \theta_1 \cos(\theta_1 - \theta_2) - \sin \theta_2) - (\mu l_1 \dot{\theta}_1^2 + l_2 \dot{\theta}_2^2 \cos(\theta_1 - \theta_2)) \sin(\theta_1 - \theta_2)}{l_2(\mu - \cos^2(\theta_1 - \theta_2))}\end{aligned}\quad (7)$$

nowych zmiennych w_1 i w_2 pozwala na sformułowanie czterech równań co ułatwia rozwiązywanie numeryczne:

$$\begin{aligned}\dot{\theta}_1 &= \omega_1 \\ \omega_1 &= \frac{g(\sin \theta_2 \cos(\theta_1 - \theta_2) - \mu \sin \theta_1) - (l_2 \dot{\theta}_2^2 + l_1 \dot{\theta}_1^2 \cos(\theta_1 - \theta_2)) \sin(\theta_1 - \theta_2)}{l_1(\mu - \cos^2(\theta_1 - \theta_2))} \\ \dot{\theta}_1 &= \omega_2 \\ \omega_2 &= \frac{g\mu(\sin \theta_1 \cos(\theta_1 - \theta_2) - \sin \theta_2) - (\mu l_1 \dot{\theta}_1^2 + l_2 \dot{\theta}_2^2 \cos(\theta_1 - \theta_2)) \sin(\theta_1 - \theta_2)}{l_2(\mu - \cos^2(\theta_1 - \theta_2))}\end{aligned}\quad (8)$$

4.1.1 Wahadło podwójne - kod

W C++ kod wahadła składa się z trzech klas: Pendulum, PendulumSpawn, PendulumControl

W klasie Pendulum, wahadło podwójne jest stworzone z Spline, na podstawie danych pozycji X i Y jest aktualizowany co klatkę. Na podstawie tych równań (8) stworzymy metodę która obliczą theta wahadła do obliczenia ich obecnej pozycji w programie:

```
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)));
theta1bis =
```

```

9   (g * u * (sin(theta0) * cos(theta0 - theta1) - sin(theta1)) + (u *
10    length0 * pow(theta0prim, 2)
11    + length1 * pow(theta1prim, 2) * cos(theta0 - theta1)) * sin(theta0
12    - theta1))
13   / (length1 * (u - pow(cos(theta0 - theta1), 2)));
14
15 theta0prim = theta0prim + theta0bis * dt;
16 theta1prim = theta1prim + theta1bis * dt;
17 theta0 = theta0 + theta0prim * dt;
18 theta1 = theta1 + theta1prim * dt;
19 }
```

Listing 1: Obliczanie wartości theta

Następnie wyliczone wartości są wykorzystywane do obliczania pozycji wahadła w metodzie computePosition i aktualizowana jest w nim pozycja Spline:

```

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,
25                                         ESplinePointType::Constant), false);
26        index++;
27    }
28
29    FirstColumn->UpdateSpline();
30
31    int32 SegmentNum = Path.Num() - 1;
32    for (int32 i = 0; i < SegmentNum; ++i)
33    {
34        if (ColumnsPathMeshPool.Num() <= i)
35        {
36            USplineMeshComponent* SplineMesh = NewObject<USplineMeshComponent>(
37                this);
38            SplineMesh->SetMobility(EComponentMobility::Movable);
```

```

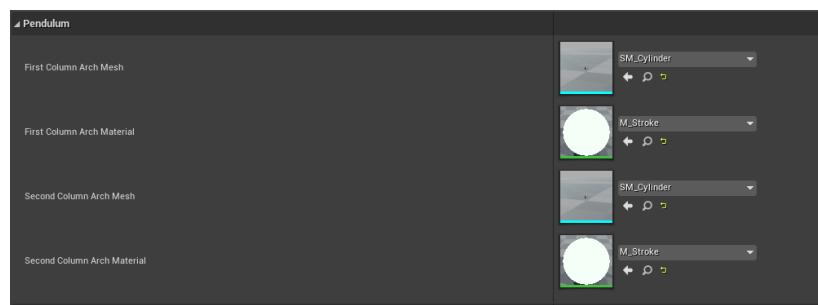
37     SplineMesh->AttachToComponent(FirstColumn ,
38     FAttachmentTransformRules::KeepRelativeTransform) ;
39     SplineMesh->SetStaticMesh(FirstColumnArchMesh) ;
40     SplineMesh->Set Material(0 , FirstColumnArchMaterial) ;
41     SplineMesh->RegisterComponent() ;
42
43     ColumnsPathMeshPool . Add(SplineMesh) ;
44 }
45
46 USplineMeshComponent* SplineMesh = ColumnsPathMeshPool[ i ] ;
47 SplineMesh->Set Visibility( true ) ;
48
49 FVector StarPos , StartTangent , EndPos , EndTangent ;
50 FVector Tangent = { 0 , 0 , 0 } ;
51 FirstColumn->GetLocalLocationAndTangentAtSplinePoint(i , StarPos ,
52 StartTangent) ;
53 FirstColumn->GetLocalLocationAndTangentAtSplinePoint(i + 1 , EndPos ,
54 EndTangent) ;
55 SplineMesh->SetStartAndEnd(StarPos , Tangent , EndPos , Tangent) ;
56 }
57 }
```

Listing 2: Aktualizacja pozycji wahadła

Klasa PendulumSpawn jest odpowiedzialna za dodawanie i usuwanie wahadeł oraz jest odpowiedzialna za uruchamianie i resetowanie wszystkich aktualnych wahadeł na scenie. Klasa ma też wskaźnik na klasę PendulumControl która ta jest odpowiedzialna za stworzenie UI dzięki któremu użytkownik może obsługiwać wahadła.

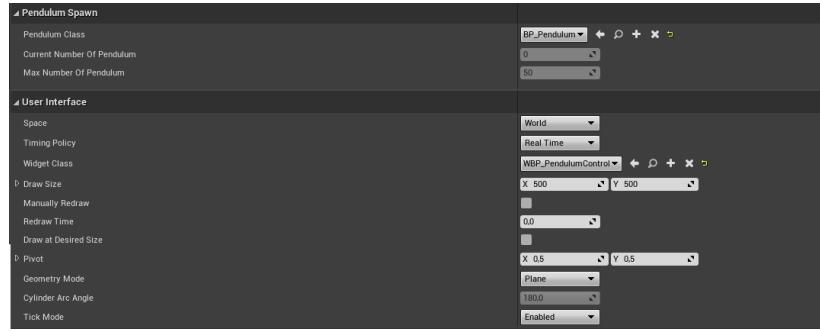
4.1.2 Wahadło podwójne - UE4

Na podstawie powyższych klas zostały stworzone klasy Blueprintowe które można potem umieścić w na poziomie w programie. BP_Pendulum ma tylko informację o tym jak Spline ma wyglądać w świecie gry:



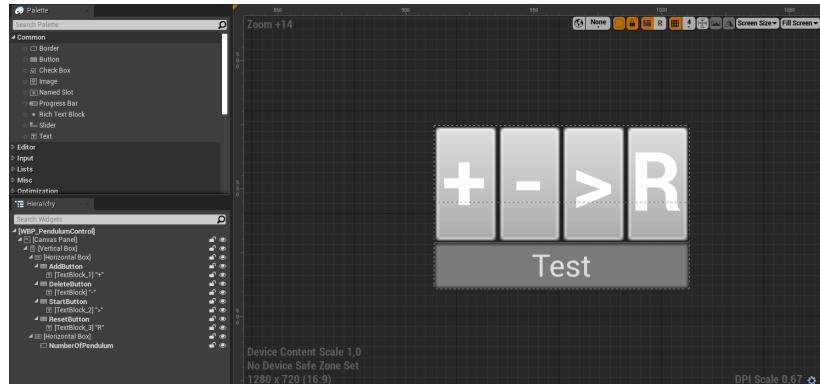
Rysunek 7: Ustawiony mesh i tekstura dla wahadła

BP_PendulumSpawn jest aktorem który zostaje umieszczony na poziomie, trzeba tylko ustawić w nim informację o wahadle które chcemy stworzyć na poziomie oraz jaki widget chcemy umieścić by można za jego pomocą sterować wahadłami.



Rysunek 8: Ustawione wahadło do stworzenia i widget

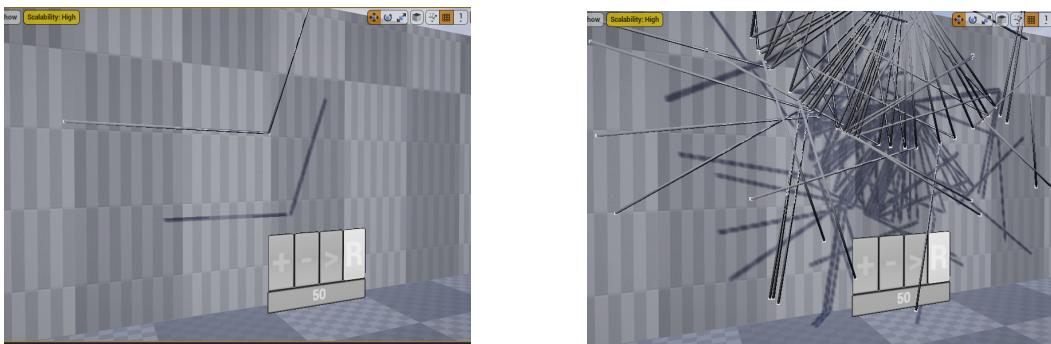
Klasa WBP_PendulumControl jest odpowiedzialna kontakt między użytkownikiem a programem, klasa ta jest widgetem dzięki któremu poprzez najechanie kontrolerem ruchowym na odpowiednie opcje możemy dodać lub usunąć wahadło oraz włączyć lub zresetować wszystkie wahadła podpięte pod dany spawner



Rysunek 9: Wygląd widgetu w programie

4.1.3 Wahadło podwójne - Wygląd w programie

W programie wahadło wygląda następująco:



(a) Wahadła na początku symulacji

(b) Wahadła na końcu symulacji

Rysunek 10: Działanie programu

4.2 Gra w życie

Gra w życie została wymyślona w 1970 przez Johna Conwaya zainspirowany pracami Stanisława Ulmana. Gra w życie jest automatem komórkowym, czyli jest to system składający się z pojedynczych komórek działający według określonych reguł.[7] W przypadku gry w życie, dana komórka, ma ośmiu sąsiadów przylegających do niej bo-kami i rogami w jednej chwili może być żywą lub martwą, a to czy żyje określają 4 zasady:

- 1.Żywa komórka, która ma mniej niż dwóch sąsiadów, w kolejnym kroku umiera.
- 2.Żywa komórka, która ma więcej niż trzech sąsiadów, umiera.
- 3.Żywa komórka, która ma dwóch lub trzech sąsiadów – przeżywa.
- 4.Martwa komórka, która ma trzech żywych sąsiadów – ożywa.

Gra w życiu nie jest do końca grą, rola "gracza" ogranicza się ustawienia stanu początkowego planszy, poprzez ożywianie lub uśmiercanie danej komórki.[8]

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. Sama w sobie ma informację o swoim położeniu X i Y na siatce 2D, o swoim obecnym stanie życia oraz o stanie życia w następnej klatce trwania programu. Kiedy użytkownik najedzie na daną komórkę i kliknie spust w metodzie Clicked zmieni stan początkowy komórki z martwą na żywą i vice 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

W metodzie Update która jest wykorzystywana przez GridActora podobnie jak w metodzie Clicked na podstawie zmiennej AliveNext, pozostawia przy życiu lub uśmierca daną komórkę

```
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    }
12 }
```

```

11     Alive = false;
12 }
13 }
```

Listing 4: Aktualizacja stanu danej komórki

Klasa ma też Gettery i Settery odpowiedzialne, za branie informacji o obecnym stanie komórki jak i o jej stanie w następnej klatce

```

1 bool GetAlive() const
2 {
3     return Alive;
4 }
5 void SetAlive(const bool IsAlive)
6 {
7     this->Alive = IsAlive;
8 }
9 bool GetAliveNext() const
10 {
11     return AliveNext;
12 }
13 void SetAliveNext(const bool IsAliveNext)
14 {
15     this->AliveNext = IsAliveNext;
16 }
```

Listing 5: Aktualizacja stanu danej komórki

Klasa GridActor jest odpowiedzialna za tworzenie siatki 2D z klasy CellActor o podanej wielkości. W samej klasie dzieją się też 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     }
18 }
```

Listing 6: 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 == 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    }
}

```

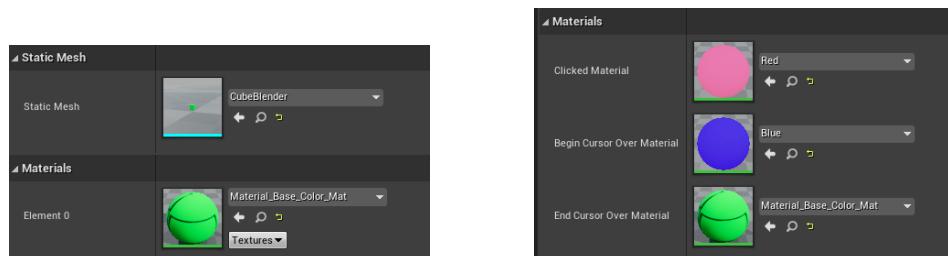
Listing 7: Aktualizacja stanu komórki w następnej klatce

Stworzona została jeszcze wersja 3D GridActora którego jedyną różnicą jest to, że na została dodana też głębokość.

Ostatnią klasą jest GameOfLifeControll która jest odpowiedzialna za stworzenie UI dzięki któremu użytkownik może włączyć grę w życie, zmieniać szybkość działania symulacji oraz resetowania jej do stanu początkowego.

4.2.2 Gra w życie - UE4

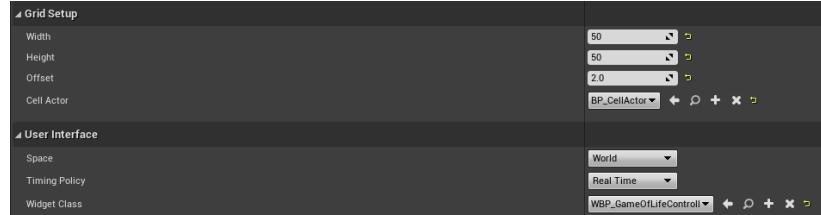
Na podstawie powyższych klas zostały stworzone klasy Blueprintowe które można potem umieścić w na poziomie w programie. W BP_CellActor ustawiamy jak dana komórka ma wyglądać w świecie gry i jak się zachować kiedy najedziemy na nią kontrolerem ruchowym:



(a) Ustawiony mesh w programie dla komórki
(b) Ustawiony wygląd w programie podczas akcji

Rysunek 11: Ustawienia dla CellActora w UE4

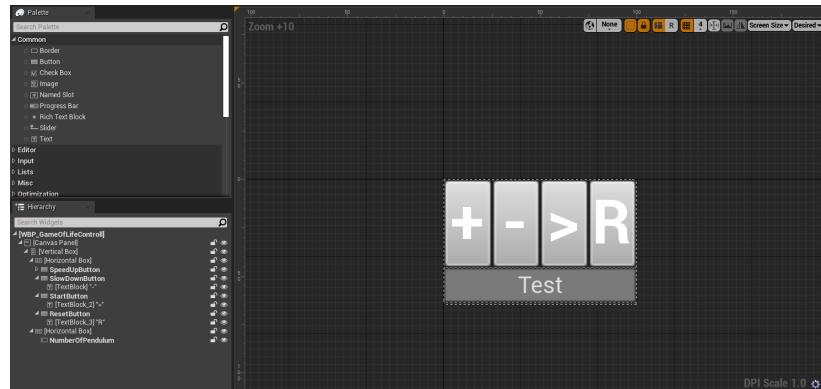
BP_GridActor2D jest klasą Blueprint która znajdzie się na poziomie gry. W klasie ustawiamy na jaką szerokość i wysokość ma zostać stworzona siatka składająca się z BP_CellActor oraz dodajemy widget, dzięki któremu możemy sterować symulacją:



Rysunek 12: Ustawienia BP_GridActor2D

Na podobnej zasadzie działa BP_GridActor3D w którym dochodzi jeszcze głębokość na którą możemy ustawić siatkę.

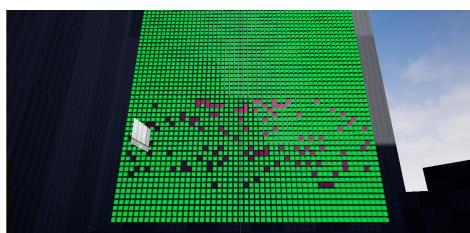
Klasa WBP_GameOfLifeControl jest odpowiedzialna kontakt między użytkownikiem a programem, klasa ta jest widgetem dzięki któremu poprzez najechanie kontrolerem ruchowym na odpowiednie opcje możemy włączyć symulację, przyśpieszyć lub ją spowolnić oraz ją zresetować do stanu początkowego:



Rysunek 13: Wygląd widgetu w programie

4.2.3 Gra w życie - Wygląd w programie

W programie Gra w życie wygląda następująco:



(a) Gra w życie podczas ustawia stanu początkowego



(b) Gra w życie w trakcie trwania symulacji

Rysunek 14: Działanie programu

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. 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. Okazało się jednak, że wyniki wyszły na odwrót. Po sprawdzeniu danych zauważał, 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[9].



Rysunek 15: Edward Norton Lorenz 1917-2008

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

$$\begin{aligned}\dot{x} &= \sigma(y - x) \\ \dot{y} &= x(\rho - z) - y \\ \dot{z} &= xy - \beta z\end{aligned}\tag{9}$$

gdzie:
sigma - stała Prandtla,

rho - stała Rayleigha,
 beta - obszar obejmujący równania
 $\sigma, \rho, \beta > 0$,
 Jednakże zwykle podaje się:
 $\sigma = 10$, $\rho = 8/3$, β jest zmienne.

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[10]

```

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 8: Metoda Tick()

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

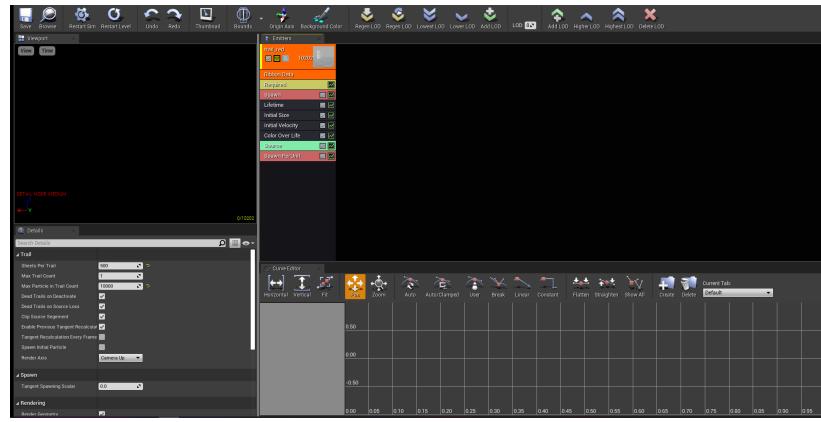
```

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 9: Tworzenie nowych atraktorów

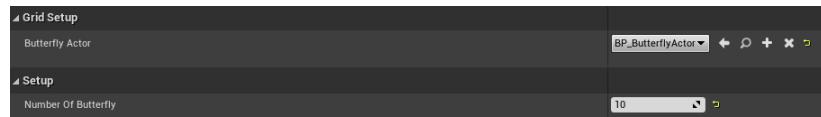
4.3.2 Efekt Motyla - UE4

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 16: Wygląd menu do tworzenia efektów cząsteczkowych w UE4

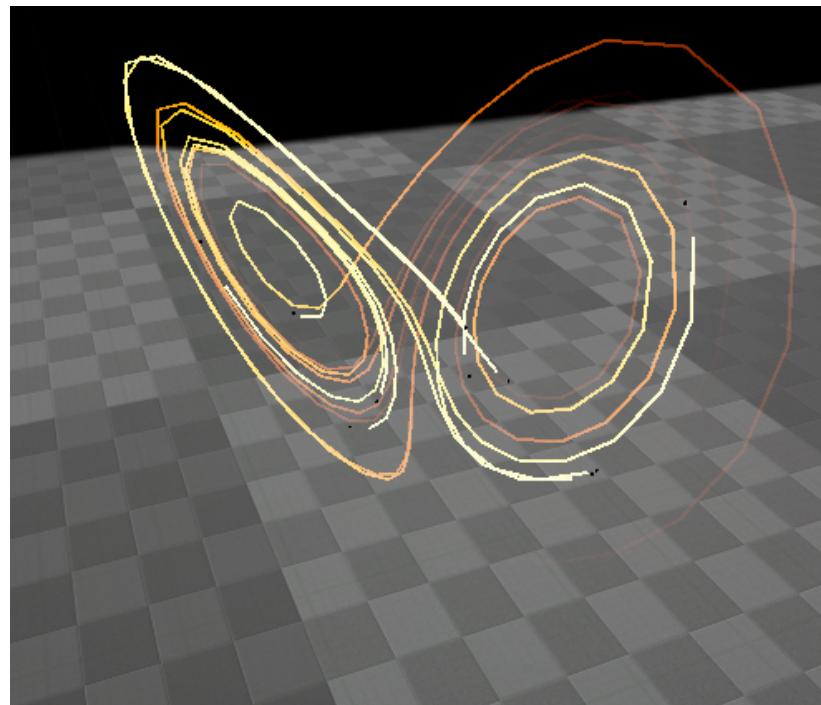
W BP_ButterflySpawner ustawiamy jakiego aktora chcemy ustawić na scenie, oraz ile motyli chcemy stworzyć.



Rysunek 17: Wygląd menu do tworzenia efektów cząsteczkowych w UE4

4.3.3 Efekt Motyla - Wygląd w programie

W programie Efekt Motyla wygląda następująco:



Rysunek 18: Wygląd motyli w programie

4.4 Model Agentowy

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.[11]

Ten ostatni wykorzystałem do pokazania działania agentów w UE4. 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 Model Agentowy - 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 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 10: 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 11: 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 króliki 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 SpawnerActorRef = GetWorld()->SpawnActor<ARabbitAgent>(
12             RabbitActor, Loc, GetActorRotation());
13         if (SpawnerActorRef)
14         {
15             SpawnerActorRef->hp = RABBIT_MAX_HUNGRY_HP_LEVEL;
16             SpawnerActorRef->SetUpAgent(true);
17         }
18     }
19 }
```

Listing 12: 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.

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

4.4.2 Model Agentowy - UE4

4.5 Boids

4.5.1 Boids - kod

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] IDC. Choroba Symulatorowa.
<https://vрpolska.eu/skad-sie-bierze-choroba-symulatorowa/>. [].
- [6] IDC. Wahadło podwójne.
https://web.archive.org/web/*/http://www.team.kdm.p.lodz.pl/master/Jankowski.pdf/. [].
- [7] IDC. Gra w życie .
<https://www.sztucznainteligencja.org.pl/sztuczna-inteligencja-gra-w-zycie/>. [].
- [8] IDC. Gra w życie .
https://en.wikipedia.org/wiki/Conway's_Game_of_Life/. [].
- [9] Paul Halpern. O motylach i burzach.
https://web.archive.org/web/20100909161235/http://czytelnia.onet.pl/0,1161316,do_czytania.html/. [].
- [10] Maciej Matyka. Jak narysować Motyle Lorenza .
[https://www.youtube.com/watch?v=XZ5QKKxHTXQ/](https://www.youtube.com/watch?v=XZ5QKKxHTXQ). [].
- [11] Bassel Karami. Intro to Agent Based Modeling.
<https://towardsdatascience.com/intro-to-agent-based-modeling-3eea6a070b72>. [].

kolor blue: rozpisać

kolor red: edytować i może dodać

kolor green: wymyślić co dodać