

**WYDZIAŁ  
ELEKTROTECHNIKI  
I INFORMATYKI  
POLITECHNIKI RZESZOWSKIEJ**

**Kacper Rudź**

Gra platformowa 3D

**Projekt inżynierski**

Opiekun projektu:  
dr inż. Sławomir Samolej prof. PRz

Rzeszów, 2023



# Spis treści

<b>1. Cel projektu . . . . .</b>	<b>5</b>
<b>2. Wstęp teoretyczny . . . . .</b>	<b>7</b>
2.1. Silnik . . . . .	7
2.1.1. Najbardziej znane silniki graficzne . . . . .	7
2.1.2. Kolejność działań w silniku Unity . . . . .	8
2.2. Grafika . . . . .	12
2.3. Projekt poziomów . . . . .	12
<b>3. Wprowadzenie do programu Blender . . . . .</b>	<b>14</b>
3.1. Blender – pola robocze i tryby . . . . .	14
3.2. Kości i szkielety – podstawy animacji szkieletowych . . . . .	15
3.2.1. Modyfikator Inverse Kinematics . . . . .	18
3.2.2. Modyfikator Copy Location i Copy Transforms . . . . .	19
3.3. Weight Paint – deformacja siatki przez szkielet . . . . .	22
3.4. Kluczowanie klatek – animowanie w programie Blender . . . . .	24
<b>4. Przygotowanie modeli, animacji i tekstur . . . . .</b>	<b>28</b>
4.1. Ciernie . . . . .	28
4.1.1. Szkielec . . . . .	28
4.1.2. Weight Paint . . . . .	30
4.2. Postać główna . . . . .	32
4.2.1. Szkielec animacji . . . . .	32
4.2.2. Model i Tekstury . . . . .	37
4.2.3. Animacje . . . . .	40
4.3. Eksportowanie modeli z animacjami . . . . .	47
<b>5. Skryptowanie i obróbka assetów w Unity . . . . .</b>	<b>50</b>
5.1. Input manager – ustawienie managera wejścia . . . . .	50
5.2. Prefab gracza . . . . .	52
5.3. Animator postaci głównej . . . . .	55
5.3.1. ‘Base Layer’ animacje ciała . . . . .	59
5.3.2. ‘Tail Layer’ – stereoawnie ogonem . . . . .	63
5.4. Skrypty postaci głównej . . . . .	66
5.5. Połączenie skryptu i animacji . . . . .	86

6. Efekt Pracy . . . . .	89
7. Podsumowanie i wnioski końcowe . . . . .	92
Załączniki . . . . .	93
Literatura . . . . .	93



## 1. Cel projektu

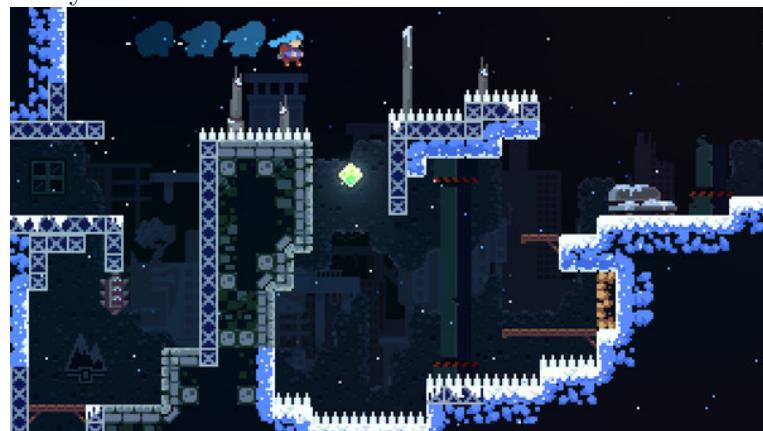
Tematem projektu inżynierskiego jest stworzenie gry platformowej 3D z zastosowaniem silnika do tworzenia gier Unity. Modele i animacje będą stworzone w programie Blender [1], a tekstury w programie QuixelMixer [2]. Rozwiązania w projekcie będą inspirowane rozwiązaniami z innych gier platformowych, takich jak seria gier Crash Bandicoot [3] (rysunek1.1), Celeste [4] (rysunek1.2) oraz Ori And The Blind Forest [5] (rysunek1.3). Główny nacisk będzie położony na skrypty w rozgrywce – najważniejszą częścią gier platformowych jest poruszanie się postacią oraz projekt poziomu. Rozgrywka będzie stworzona dla jednej osoby, a jej celem będzie przejście do końca poziomu. Na drodze do celu gracz będzie musiał zmagać się z ruchomymi platformami i pułapkami – ciernie zabierające zdrowie bohatera. Gdy zdrowie postaci spadnie do zera gracz zostaje cofnięty do ostatniego odwiedzonego punktu kontrolnego. Zdrowie postaci będzie się powoli regenerowało z czasem. Kamera będzie skierowana od boku postaci oddalona o pewną odległość. Gracz do dyspozycji będzie miał umiejętności związane z wspinaniem się po ścianach i skoki w powietrzu. W powietrzu gracz będzie miał ograniczoną kontrolę nad postacią – będzie można poruszać się w powietrzu jednak nie będzie to tak płynne jak poruszanie się po podłożu. Wspinanie po ścianach będzie ograniczała wytrzymałość. Wytrzymałość będzie się regenerować, gdy postać będzie stała na ziemi lub platformie. Liczbę skoków w powietrzu gracz będzie mógł zwiększyć poprzez dotarcie do odpowiednich punktów w grze.

Modele będą zrobione w stylu low/medium poly – szczegółowość modeli będzie niska. W tym stylu mocno widać krawędzie między ścianami. Modele będą posiadały proste tekstury. Animacje postaci będą proste, niezbyt skomplikowane. Tekstury należy ‘wypalić’ – zrobić mapy normalnych, kolorów itp.

Poruszanie postaci zostanie zaprojektowane tak, aby było wymagające, ale też intuicyjne i łatwe do zapamiętania – obsługa klawiszy będzie podobna jak w innych grach platformowych. Ważną częścią pisania skryptów jest optymalizacja, ze względu na to, że w dzisiejszych czasach 60 FPS(z ang. Frames Per Second – Klatki na sekundę) jest wymagane przez użytkowników. Dla programisty to oznacza, że obliczenia graficzne muszą zostać wykonane w czasie 1/60 sekundy. Obliczenia fizyczne muszą być wykonywane z taką samą częstotliwością lub szybciej od obliczeń graficznych. Dzięki temu postać i inne poruszające się obiekty będą poruszały się płynnie.



Rysunek 1.1: Crash Bandicoot 4: It's About Time



Rysunek 1.2: Celeste



Rysunek 1.3: Ori and the Blind Forest

## **2. Wstęp teoretyczny**

Gry są to programy, które służą głównie rozrywce. Gry komputerowe dzieli się na typy – różniące się rozgrywką. Te programy można podzielić na kilka części składowych:

- 1) skrypty,
- 2) grafika,
- 3) udźwiękowienie,
- 4) projekt poziomu,
- 5) projekt rozgrywki.

Waga każdego elementu zależy od gatunku gry – przykładowo w grach strategicznych ważną częścią są skrypty sztucznej inteligencji, z którą mierzy się gracz, a w grach rzeczywistościowych ważnym elementem jest projekt poziomu. Produkcja gier jest skomplikowanym procesem, i wymaga dużo czasu. W celu ułatwienia oraz przyspieszenia produkcji gier korzysta się z silników, które odpowiadają za symulację fizyki oraz rendering elementów graficznych (wygenerowanie i wyświetlenie obrazu gry).

### **2.1. Silnik**

#### **2.1.1. Najbardziej znane silniki graficzne**

Istnieje wiele silników do tworzenia gier, jednak silnikiem najprostszym i jednym z najbardziej rozpoznawalnych jest Unity[6] – do skryptów używa języka C#. Innym silnikiem, który dominuje na rynku jest Unreal Engine[7] – do skryptów używa języka C++. Oba silniki posiadają swoje wady i zalety. Zaletami Unity są prostota, skalowalność i szybkość powstawania projektów, a do wad należy zaliczyć fakt, że rozbudowane gry często działają wolno. Zaletami Unreal Engine są szybkość stworzonej już gry, elastyczność (silnik pozwala na bardzo dużo i twórców ogranicza głównie wyobraźnia oraz moc obliczeniowa komputera) oraz ilość przygotowanych elementów przez twórców, a do wad zalicza się wymagania edytora oraz czas potrzebny na zrobienie małego projektu. Oba silniki posiadają interfejsy graficzne do rozstawiania, przesuwania, skalowania, obracania obiektów. Oba silniki posiadają symulator fizyki, który jest odpowiedzialny za obliczenia związane z fizyką. Unity oraz Unreal Engine posiadają złożone systemy

oświetlenia – obiekty, które emitują światło można dowolnie modyfikować (położenie, kierunek, intensywność, kolor itp. )[8].

Każdy z silników jest zbudowany w inny sposób, ale zasada działania każdego z nich jest podobna.

### **2.1.2. Kolejność działań w silniku Unity**

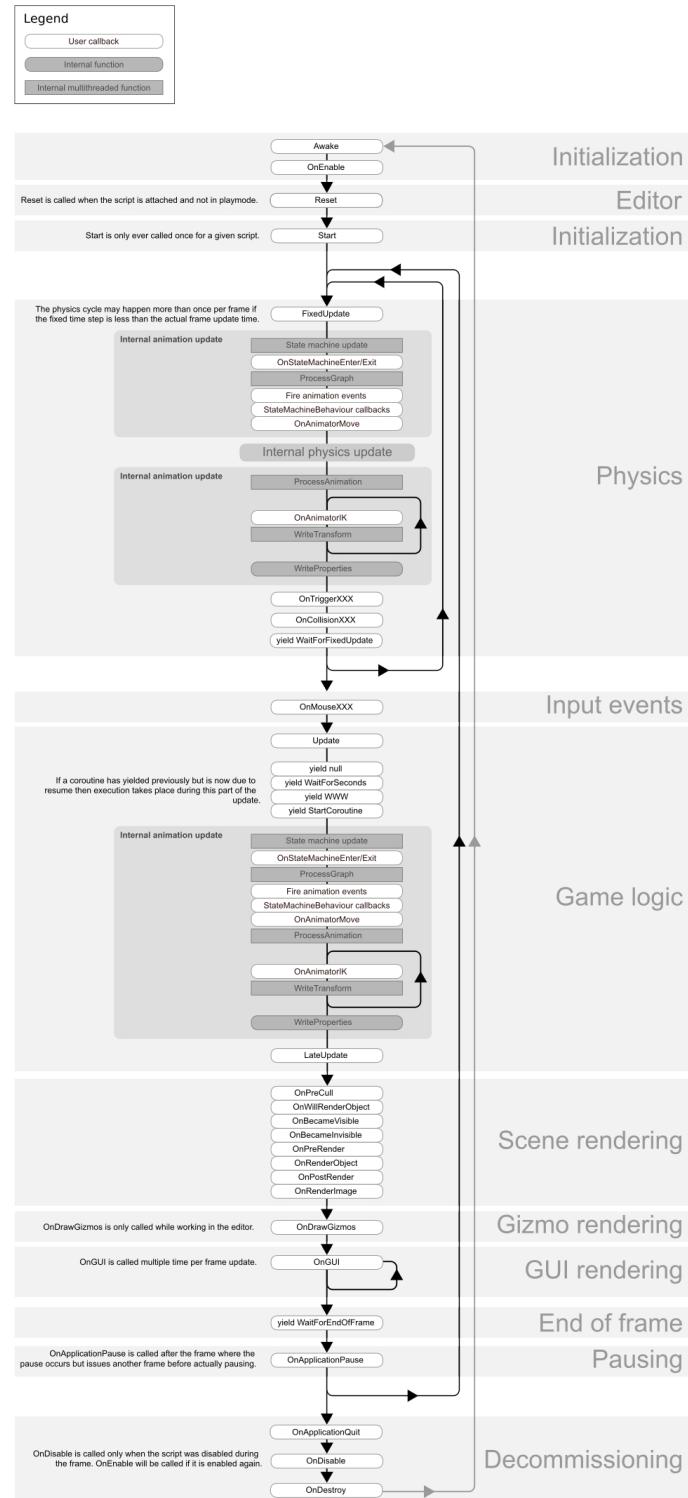
W silnikach graficznych, aby uniknąć nieporozumień i ułatwić pracę programistom tworzy się tzw. Flow Chart, wykresy, które obrazują zachowanie się silnika. Określają one kolejność wywoływanego metod klasy dziedziczącej z obiektu bazowego. W Unity obiektem bazowym każdej klasy skryptu w obiektach jest MonoBehaviour. Na rysunku 2.4 znajduje się podział kolejność wywoływanego metod – zaczynając od góry. Każdy blok odpowiada metodzie w klasie bazowej MonoBehavior, z której dziedziczy każda klasa przypięta do obiektu w silniku Unity. Każdą z tych metod można nadpisać. Każdy obiekt ma pewien czas życia – czas, w którym obiekt jest obecny w grze. Bloki białe – metody, które można nadpisać Broki szare – skrypty silnika Unity, tych metod nie można nadpisać. Podczas życia obiektu zachodzi wiele obliczeń, które można podzielić na kilka etapów. W każdym etapie znajdują się metody do wpływu na program. Metody inicjalizacji obiektu[9]:

- void Awake() – metoda wywoływana, gdy obiekt GameObject został zainicjalizowany, gdy scena zawierająca obiekt została załadowana, obiekt wcześniej nieaktywny został aktywowany lub obiekt został stworzony używając metody Object.Instantiate. Metoda jest wywoływana tylko raz w trakcie życia instancji skryptu
- void OnEnable() – metoda zostaje wywoływana, gdy skrypt jest aktywowany – wywołanie metody – SetActive(true), gdy skrypt poprzednio był nieaktywny.

Pętla Fizyczna – obliczenia związane z fizyką, mogą być wywoływane kilkukrotnie w ciągu klatki:

- void FixedUpdate() – metoda pętli fizycznej, może być wywoływana więcej razy niż metoda Update. W tej metodzie powinny się znaleźć metody, które odpowiadają za fizykę gry (nie muszą się tutaj znajdować).
- void OnStateMachineExit/Enter(Animator animator, int stateMachinePathHash) – za argumenty przyjmują component animatora oraz numer maszyny stanu w

animatorze. Metody działają na tej podobnej zasadzie – jedna jest wywoływana na zakończenie animacji, a druga podczas włączenia następnej animacji



Rysunek 2.4: Unity flowchart

- void FireAnimationEvent(void/float/string/int/object) – metoda wywoływana w pewnym miejscu animacji, oznaczonym w edytorze animacji silnika Unity – zakładka ‘Animation’ w inspektorze silnika.
- StateMachineBehaviour callbacks – wywoływanie metod klasy StateMachine-Behaviour. Klasa ta jest używana do skryptów używanych w trakcie działania pewnych animacji.
- void OnAnimatorIK(int) – wywoływana przed aktualizacją wewnętrznego systemu IK (inverse kinematics).
- metody OnTriggerXXX i OnCollisionXXX – metody wywoływanie podczas detekcji kolizji – są 3 rodzaje metod Enter (kolizja się rozpoczęła), Stay (kolizja trwa) i Exit (zakończenie kolizji). Są 2 typy kolizji w silniku Unity, kolizje typu collider-collider i typu collider-trigger. Kolizje collider-collider będą wywoływać metody OnCollisionXXX oraz powodują że obiekty nie ‘wchodzą’ w siebie. Kolizje typu collider-trigger wywołują metody typu OnTriggerXXX oraz powodują że jeden obiekt przechodzi przez drugi (collider typu trigger jest traktowany jak powietrze w kolizjach). Kolizje typu trigger-trigger nie występują.
- yield return – sygnalizuje, że dalsza część programu zostanie wykonana później. Nie jest to metoda, a korutyna (Coroutine).
- yield return WaitForFixeUpdate – kontynuacja wykonywania metod po metodzie FixedUpdate

Input Events (zdarzenia wejścia) – w tym miejscu wywoływane są zdarzenia urządzeń wejścia. Graf może się różnić, gdy jest używana paczka ‘Input system’, która obsługuje te zdarzenia. Logika Gry (Game Logic):

- void Update() – metoda wywoływana raz na klatkę. Powinna zawierać skrypty do zarządzania GUI, animacjami oraz elementy logiki gry.
- yield return null – kontynuacja skryptu w następnej klatce
- yield return WaitForSeconds – kontynuacja skryptu po podanej liczbie sekund
- yield return WWW – kontynuacja skryptu po ściągnięciu obiektu z strony www

- `yield return StartCoroutine` – kontynuacja skryptu po zakończeniu metody
- `void LateUpdate()` – wywoływana pod koniec logiki gry. Najczęściej używana do ustawiania kamery.

Scene rendering:

- `void OnPreCull()` – metoda wywoływana przed obcięciem sceny do pola widzenia kamery.
- `void OnWillRenderObject()` – metoda wywoływana dla każdej kamery jeżeli obiekt jest widoczny (a nie jest elementem UI).
- `void OnBecameVisible()` – metoda wywoływana, gdy obiekt jest widoczny przez jakąkolwiek kamerę.
- `void OnBecameInvisible()` – metoda wywoływana, gdy obiekt nie jest widoczny w żadnej z kamer.
- `void OnPreRender()` – metoda jest wywoływana przed renderowaniem sceny.
- `void OnRenderObject()` – metoda jest wywoływana w trakcie renderowania sceny.
- `void OnPostRender()` – metoda jest wywoywana po wyrenderowaniu sceny.
- `void OnRenderImage(RenderTexture, RenderTexture)` – pozwala na modyfikacje końcowego obrazu kamery. Najczęściej używana do efektów tzw. post processing'u.

Gizmo rendering – jest to renderowanie elementów do debugowania. Metoda `void OnDrawGizmos()` pozwala na nałożenie tych elementów. Będą one nałożone na resztę obiektów. GUI rendering – render elementów graficznego interfejsu użytkownika. Posiada tylko metodę `void OnGUI()`. Rekomendowane jest używanie tej metody do modyfikacji elementów UI i obsługi zdarzeń tych elementów (np. przycisków). Jest wywoływana wielokrotnie w ciągu klatki. Etap End Of frame kontynuuje program w miejscu wywołania `yield return WaitForEndOfFrame`. Decommissioning – etap w którym kończy się życie obiektu. Metody `void OnApplicationQuit()` – wywoływana podczas wyjścia z programu – `void OnDissable()` – zablokowanie wykonywania skryptu, wywoływana w

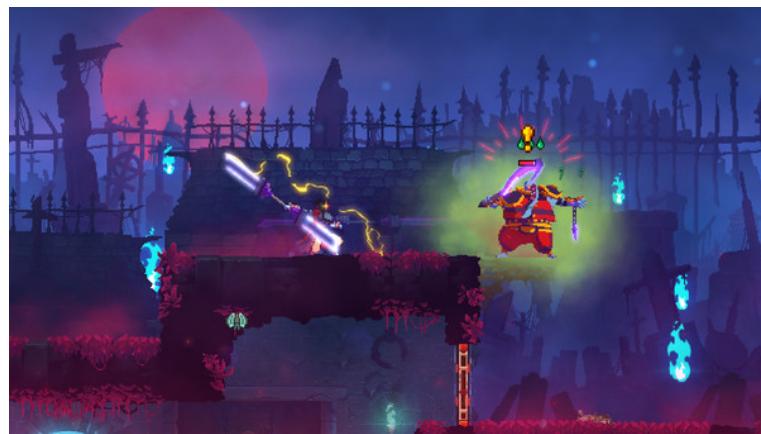
klatce, w której został on zablokowany – void `OnDestroy()` – metoda do zniszczenia obiektu, w tym miejscu powinny zostać zwolnione zasoby.

## 2.2. Grafika

W dzisiejszych czasach do modelowania oraz tworzenia animacji używa się wyspecjalizowanych do tego programów. Najczęściej używanym programem jest Blender[1] – darmowy program związany z trójwymiarową grafiką komputerową. Jest programem najczęściej wybieranym ze względu na zaawansowanie oraz fakt, że wszystkie narzędzia są w jednym programie. Jest to bardzo pomocne podczas tworzenia modeli i animacji. Tekstury są to różne mapy, które nakładane są na model – najczęściej używane to mapy kolorów, normalnych i oświetlenia (ambient, diffuse, specular). Tekstury można generować proceduralnie lub tworzyć ręcznie za pomocą programów graficznych. Tekstury generowane graficznie tworzone są szybciej jednak wymagają wypalania – proces, który zapisuje teksturę wygenerowaną do pliku – co wymaga dużo czasu. Największą zaletą generacji tekstur jest efekt nakładania się map, dzięki któremu mapy wyglądają naturalnie (najczęściej rozjeżdżają się mapy normalnych i oświetlenia). Do tworzenia tekstur można użyć programów do tego dedykowanych (np. Quixel Mixer), które wspierają np. warstwy.

## 2.3. Projekt poziomów

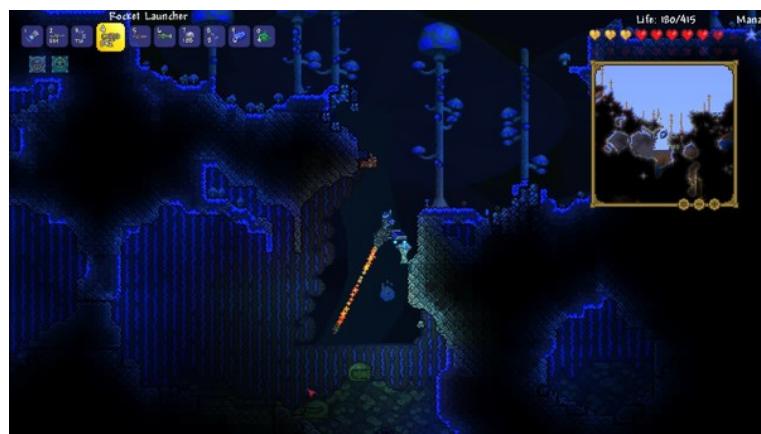
Poziomy tworzone są proceduralnie lub ręcznie. Proceduralna generacja poziomów dotyczy tylko określonych gier lub terenu. Pełna generacja poziomów zwykle dotyczy określonych gatunków gier, gdzie twórcy zakładają przechodzenie gry wielokrotnie np. Deadcells (2.5) lub bardzo ważne elementem jest proceduralna generacja poziomów np. Minecraft (2.6), Terraria (2.7). Tworzenie ręczne poziomów często wykorzystuje generację terenu (najczęściej obejmuje góry, jaskinie, polany). W innym wypadku pozostaje ręczne tworzenie terenu – ten proces jest bardzo czasochłonny. Na przygotowaną mapę terenów nakładane są modele drzew, krzewów, budynków itp. Narzędzia do tworzenia poziomów (generowanych i tworzonych ręcznie) są dostępne w silnikach lub zostały przygotowane dodatki lub wtyczki do tego celu.



Rysunek 2.5: Dead Cells



Rysunek 2.6: Minecraft



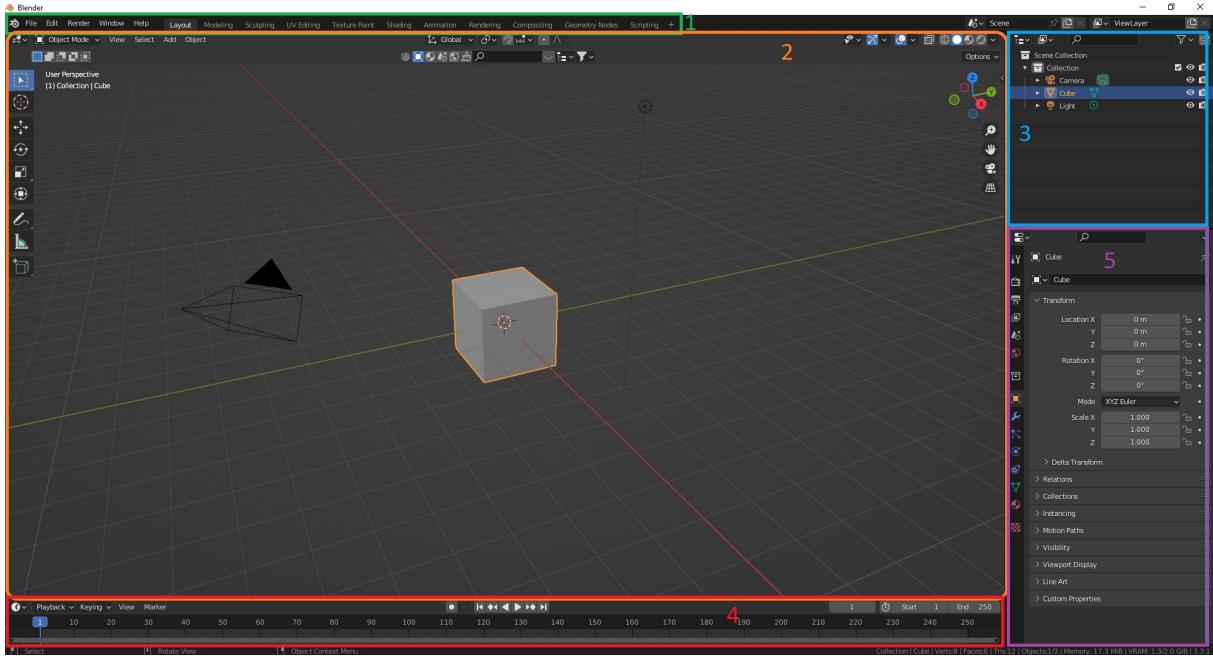
Rysunek 2.7: Terraria

### **3. Wprowadzenie do programu Blender**

#### **3.1. Blender – pola robocze i tryby**

Blender jest programem, który jest bardzo elastyczny. To stwierdzenie dotyczy też modyfikacji przestrzeni roboczej. Domyślnie w wersji programu v3.3 posiada 11 układów stworzonych przez twórców — Layout, Modeling, Sculpting, UV Editing, Texture Paint, Shading, Animation, Rendering, Compositing, Geometry Nodes, Scripting (pole nr. 1 na rysunku 3.8)– każdy z nich jest stosowany do innych celów. Możliwe jest też tworzenie własnych układów co zostało opisane w dokumentacji programu [10]. Znajdują się one w pasku u góry okna programu (na rysunku 3.8 nr. 1). Na rysunku pola 2, 3, 4 i 5 są obszarami roboczymi, które można zmieniać. [11] Rysunek 3.8 nr. 2 domyślnie przedstawia widok na scenę. Zawartość każdego pola roboczego można zmienić klikając w ikonę w lewym górnym rogu pola roboczego – zawarte są tutaj różnego rodzaju pola robocze do np. edytowania shaderów, podglądu sceny czy edytory animacji. Obok znajduje się tryb danego pola roboczego – w nie każdym polu roboczym taki się znajduje, ponieważ niektóre pola robocze nie mają innych trybów – który pozwala na jego zmianę. Domyślną zawartością pola nr. 3 — ‘Outliner’ — jest lista obiektów znajdujących się na scenie. Obiekty można grupować w kolekcje. W niektórych przypadkach obiekty mogą zawierać w sobie inne obiekty. Obszar nr. 4 standardowo jest obszarem kontroli czasu – nazywa się ‘Timeline’ – używanym głównie przy tworzeniu animacji. W obszarze nr. 5, który nazywa się ‘Properties’ domyślnie znajdują się właściwości obiektu, sceny itp.

Niektóre tryby są dostępne po zaznaczeniu odpowiedniego obiektu – np. ‘Pose mode’, za pomocą którego tworzy się animacje jest dostępny tylko w wypadku zaznaczenia obiektu typu ‘Armature’. Do zrobienia modeli najczęściej używa się trybów ‘Edit mode’ i ‘Sculpt mode’. Pierwszy z nich pozwala na tworzenie prostych siatek za pomocą narzędzi do skalowania, rotacji, przesuwania i tworzenia nowych ścian, krawędzi i/lub wierzchołków. ‘Sculpt mode’ jest trybem, który można porównać do tworzenia glinianej rzeźby – narzędzia pozwalają na wycinanie, spłaszczenie płaszczyzn, rozciąganie itp. Posiada nawet pędzel symulujący tkaniny. Tworzenie modeli wymaga sporo czasu i doświadczenia w używaniu tych dwóch trybów.



Rysunek 3.8: Blender podział przestrzeni roboczej

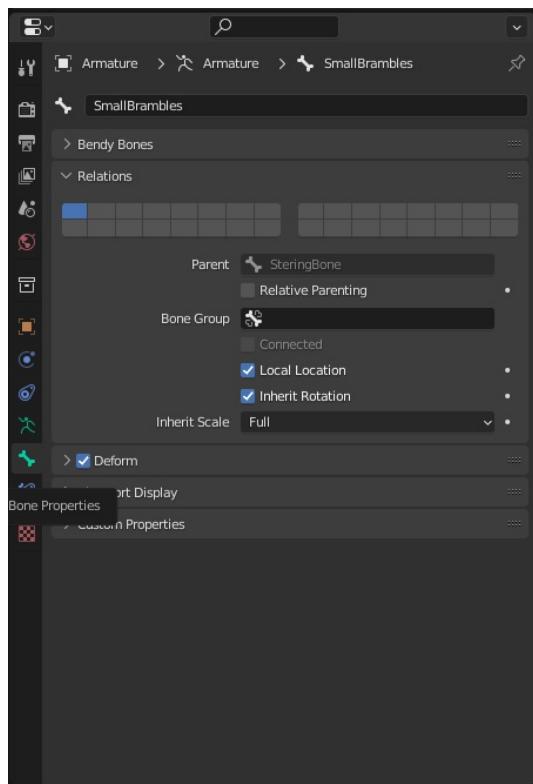
### 3.2. Kości i szkielety – podstawy animacji szkieletowych

Szkielet – w programie Blender obiekt szkieletu nazywa się ‘Armature’ – jest zbiorem kości, które są ze sobą w połączone w relacji rodzic – dziecko. Relacja ta polega na tym, że kość rodzica wpływa na dzieci. Każda kość może, ale nie musi mieć rodzica – tak samo jest w przypadku dzieci. Kości mają początek – nazwany ‘Head’ – i koniec – nazwany ‘Tail’. Na kości mogą działać modyfikatory – specjalne operacje z dziedziny animacji, które ułatwiają animowanie. Kości jedynie wpływają na siatkę – obiekt typu ‘Mesh’ – deformując ją. Kości nie są wyświetlane i służą tylko do deformacji. Jest to możliwe poprzez nakładanie wag – współczynnika – z jaką dana kość wpływa na dany punkt w siatce. Edytowanie wag jest możliwe w trybie ‘Weight Paint’ – należy zaznaczyć wybrać obiekty ‘Armature’ i ‘Mesh’ . Jeden obiekt typu ‘Armature’ może deformować wiele obiektów typu ‘Mesh’.

Do stworzenia szkieletu używa się ‘Edit mode’, gdy zaznaczony jest obiekt typu ‘Armature’. W tym trybie można tworzyć nowe kości, zmieniać ich rozmiar, relacje - określanie czyje i jakie transformacje ma dziedziczyć — czy określać, czy dana kość deformuje siatkę. Trybem używanym do robienia animacji jest wcześniej wspomniany tryb ‘Pose mode’. W tym trybie wcześniej stworzony szkielet działa według określonych zasad:

- jeżeli kość ma rodzica i jest ‘connected’ to wtedy nie może się od tego rodzica odczepić – początek kości będzie pyrzczepiony dokońca kości rodzica
- jeżeli kość ma odznaczoną opcję ‘inherit Rotation’ to rotacja rodzica nie rotuje tej kości, ale ją przesuwa
- modyfikatory mogą działać niezależnie od zaznaczenia pól ‘connected’ i ‘inherit rotation’

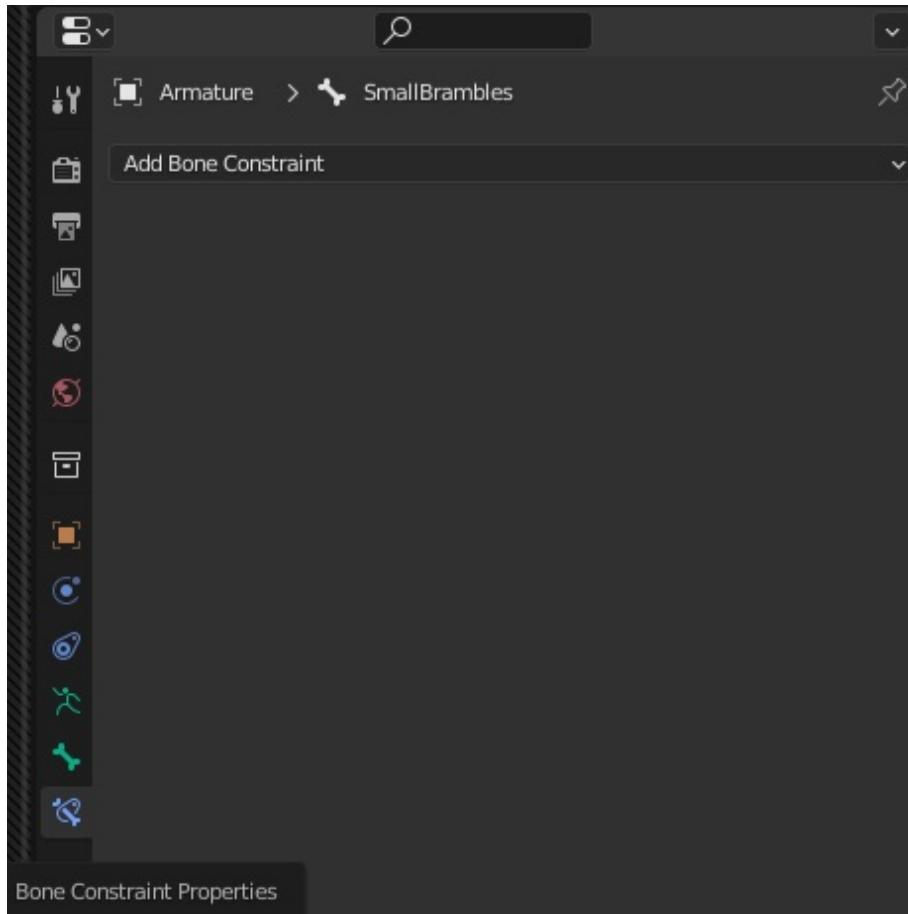
Położenie pól ‘connected’ i ‘inherit Rotation’ pokazano na rysunku 3.9 znajduje się ono w polu roboczym ‘Properties’, zakładka ‘Relations’ w zakładce ‘Bone Properties’, tutaj znajduje się również informacja, która kość jest rodzicem i opcja do zmiany czy dana kość deformuje siatkę – pole ‘Deform’. Zakładka ‘Bone Properties’ jest dostępna po wybraniu obiektu typu ‘Aramature’ – w trybie ‘Object Mode’ też jest dostępna, ale nie działa poprawnie, a poprawne działanie jest w trybach ‘Edit Mode’ i ‘Pose Mode’.



Rysunek 3.9: Miejsce zmiany właściwości Kości

Rysunek 3.10 pokazuje położenie zakładki zawierającej modyfikatory kości -- zakładka nazywa się ‘Bone Constraints Properties’ i jest dostępna tylko w trybie ‘Pose

mode'. Narzędzia tutaj zawarte bardzo ułatwiają tworzenie animacji szkieletowych – jest ich 28 w 4 różnych kategoriach pokazanych na rysunku 3.11.



Rysunek 3.10: Miejsce zmiany właściwości Kości

Motion Tracking	Transform	Tracking	Relationship
Camera Solver	Copy Location	Clamp To	Action
Follow Track	Copy Rotation	Damped Track	Armature
Object Solver	Copy Scale	Inverse Kinematics	Child Of
	Copy Transforms	Locked Track	Floor
	Limit Distance	Spline IK	Follow Path
	Limit Location	Stretch To	Pivot
	Limit Rotation	Track To	Shrinkwrap
	Limit Scale		
	Maintain Volume		
	Transformation		
	Transform Cache		

Rysunek 3.11: Lista modyfikatorów dostępnych w programie

W projekcie zostały użyte następujące modyfikatory kości:

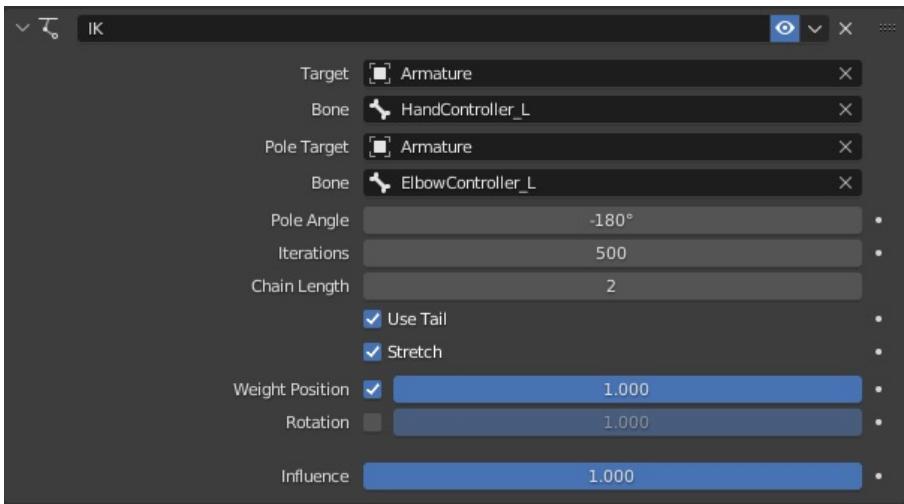
- Inverse Kinematics
- Copy Location
- Copy Transforms

### **3.2.1. Modyfikator Inverse Kinematics**

Modyfikator Inverse Kinematics (rysunek 3.12) wpływa na kością posiadającą ten modyfikator oraz rodziców – w zależności ile zostało ustawione w parametrze ‘Chain Length’. Efektem jest umieszczenie końca kości w pewnej pozycji – początku innjej kości. Kości, na które wpływa modyfikator są nazywane łańcuchem. Za parametry przyjmuje:

- Target – jest to obiekt, który będzie brany jako punkt, w którym koniec kości ma się znaleźć. W przypadku wybrania obiektu typu ‘Armature’ należy wybrać również kości tego obiektu.
- Bone – jest to kość w obiekcie ‘Armature’.
- Pole Target – obiekt, który określa, w którym kierunku ma zostać wygięty dany łańcuch. Można powiedzieć, że ten obiekt odpowiada za rotację łańcucha w odpowiednim kierunku.
- Pole Angle – stosowane do korygowania wygięcia łańcucha
- Iterations – określa ile maksymalnie iteracji obliczeń ma się odbyć w celu obliczenia transformacji kości
- UseTail – pole określa czy początek kości może być ostatnim elementem łańcucha
- Stretch – pozwala/zabrania na rozciąganie kości
- Rotation – stopień w jakim brana jest pod uwagę rotacja obiektu ‘Target’
- Influence – stopień wpływu modyfikatora na kości w łańcuchu

Ten modyfikator przydaje się podczas animacji kości, które poruszają się w jednej płaszczyźnie lub każda z kości może rotować się w jednej płaszczyźnie – np. nogi, ręka lub palce. Przy bardziej skomplikowanych odcinkach – np. kręgosłup – modyfikator IK nie sprawdza się.



Rysunek 3.12: Modyfikator kości Inverse Kinematics

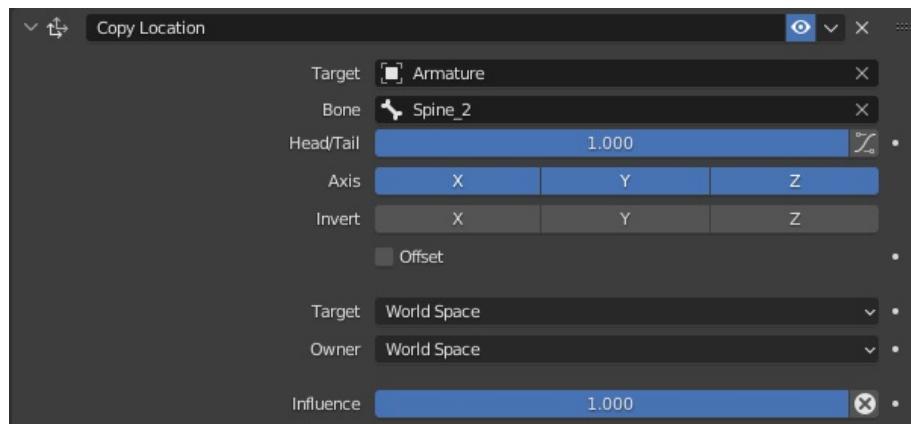
### 3.2.2. Modyfikator Copy Location i Copy Transforms

Copy Location (rysunek 3.13) jest modyfikatorem odpowiadającym za m.in. skopiowanie pozycji innej kości w określonym układzie odniesienia – wpływa na kość posiadającą ten modyfikator.

Modyfikator posiada następujące pola (rysunek 3.13):

- Target – obiekt, od którego kopiwana jest położenie. W przypadku wybrania obiektu typu ‘Armature’ pojawia się pole ‘Bone’ – kość wybranego szkieletu
- Head/Tail – określa czy kopiwana ma być kopiwana od początku czy końca wybranej kości
- Axis – określa czy wszystkie współrzędne mają być kopowane.
- Invert – pozwala wybrać, które współrzędne mają być odwrócone – jeżeli zaznaczone wtedy dana współrzędna jest mnożona przez -1
- Offset – jeżeli opcja będzie zaznaczona to kopowane współrzędne będą traktowane jako przesunięcie.
- target – jest to układ odniesienia, z którego kopowane będą współrzędne. Dostępne są następujące układy odniesienia [13]:
  - World Space (przestrzeń globalna) – punktem odniesienia jest punkt (0,0,0).

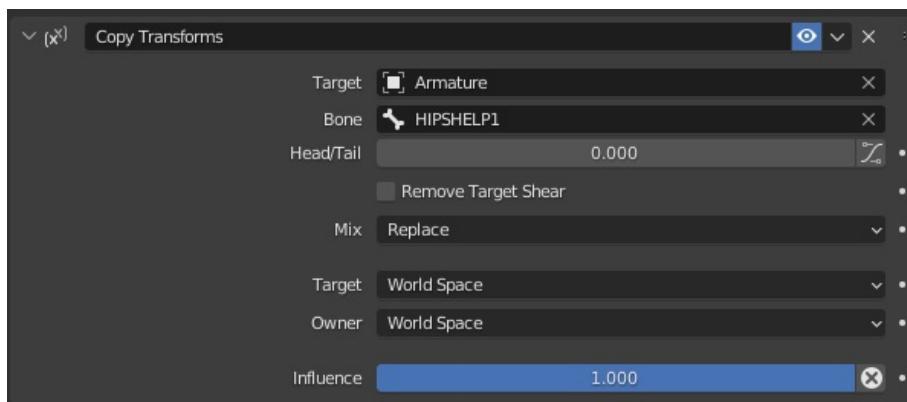
- Local Space (przestrzeń lokalna) – brane są transformacje względem obiektu rodzica.
- Local with Parent (lokalna z rodzicem) tylko dla kości. Transformacje są oceniane względem położenia i orientacji w pozycji spoczynkowej.
- Pose Space (przestrzeń pozy) tylko dla kości. Działa podobnie do globalnego układu odniesienia tyle że punktem odniesienia jest transformacja głównego obiektu ‘Armature’.
- Custom Space (przestrzeń niestandardowa) – transformacje są oceniane względem bieżącej pozycji i orientacji wybranego obiektu.
- Local Space (Owner Orientation) tylko dla kości ‘Target’. Ta przestrzeń pozwala na taki sam ruch kości która kopiuje transformacje pod warunkiem, że kości rodzice są w pozycji spoczynkowej.
- Owner pole podobnie jak target określa do jakiej przestrzeni będą kopiowane współrzędne przemieszczenia.



Rysunek 3.13: Modyfikator kości Copy Location

Modyfikator Copy Transforms jest rozwinięciem modyfikatora Copy Location – poza kopiowaniem lokacji kopiuje jeszcze skalę i rotację. Na rysunku 3.14 widać, że ten modyfikator jest obcięty z możliwości wybrania kopiowanych osi. Zamiast tego posiada pole ‘Mix’, w którym można określić w jaki sposób w jaki sposób transformacje kopiowane ingerują w transformacje kości-właściciela. W tym polu można ustawić następujące wartości:

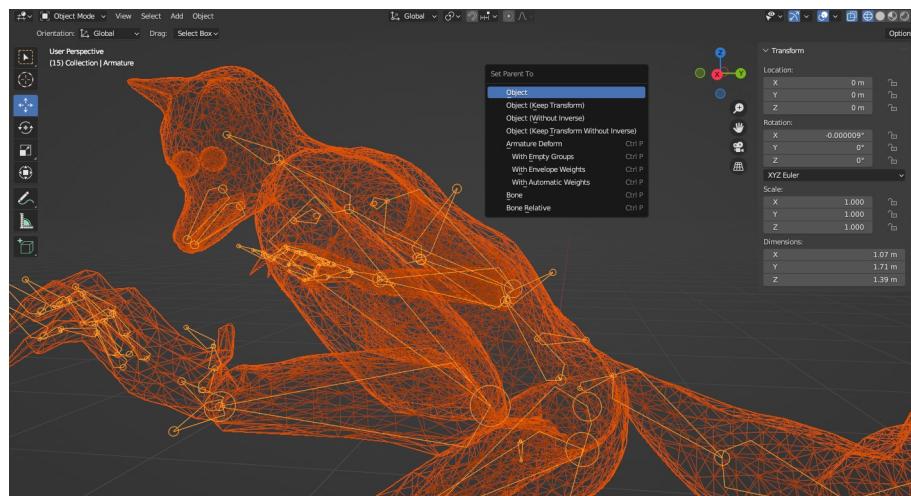
- Replace – podmienia transformacje na transformacje obiektu w polu ‘Target’ lub wybranej kości.
- Before/After Original (Full) – Nowa transformacja jest dodawana przed (Before) lub po (After) macierzy właściciela. Full oznacza, że skala jest w pełni dziedziczona – jest wyliczona z wszystkich rodziców obiektu, z której kopowane są transformacje.
- Before/After Original (Aligned) – Nowa transformacja jest dodawana przed (Before) lub po (After) macierzy właściciela. Aligned oznacza, że kość jest skalowana o całkowitą skalę obiektu ‘Target’ lub wybranej kości.
- Before/After Original (Split Channels) – w tym trybie rotacja, skala i lokacja są rozdzielane na osobne kanały. W tym przypadku kanały lokacji są dodawana oddziennie od kanałów rotacji i skali, dzięki czemu skala i rotacja nie mają wpływu na lokacje.



Rysunek 3.14: Modyfikator kości Copy Transforms

### 3.3. Weight Paint – deformacja siatki przez szkielet

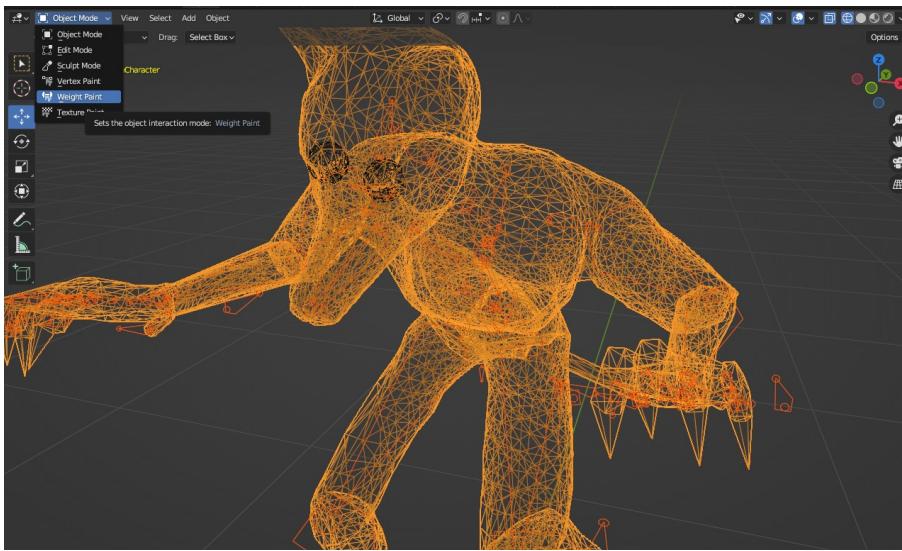
‘Weight Painting’ jest procesem polegającym na przypisaniu wagi kości wierzchołkom. W ten sposób siatka zmienia się odpowiednio według zmian transformacji kości. W celu zainicjalizowania ‘połączenia’ siatki i szkieletem należy wybrać obiekt siatki – ‘mesh’- i obiekt szkieletu – ‘Armature’ – jak na rysunku 3.15 i następnie wcisnąć klawisze ‘CTRL’ i ‘P’ w tym samym czasie, co otworzy menu jak na rysunku, a opcjami odpowiedzialnymi za związanie ze sobą tych obiektów są 3 poniżej opcji ‘Armature Deform’. Wynikiem ‘With Empty Groups’ sprawi, że każda kość nie będzie miała wag. ‘With Automatic Weights’ sprawi, że wagi będą tworzone automatycznie na podstawie odległości – wagi często są błędnie wyliczane. Ostatnia opcja ‘With Envelope Weights’ wykorzystuje ‘Bone Envelope’ – strefę wpływów kości. Problemem tej metody jest problematyczne ustawianie ‘Stref wpływów’.



Rysunek 3.15: Wstępne wyliczenie wag kości

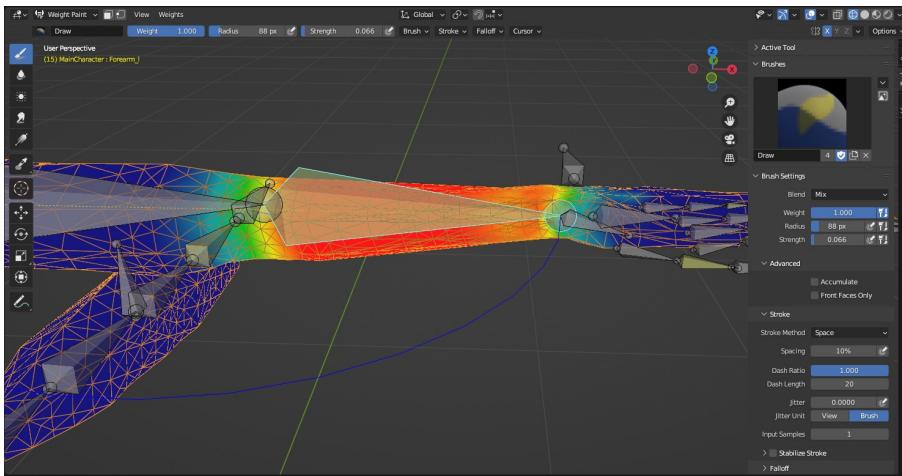
Szkielet może deformować wiele siatek w tym samym czasie. Dlatego w programie Blender wybór siatek odbywa się w trybie obiektowym. Na rysunku 3.16 znajduje się przykładowy wybór siatki do ‘Weigt Painting’u’.

Poziom wpływu danej kości na obszarze jest wyrażony w postaci kolorów. Kolor niebieski reprezentuje brak wpływu kości na siatkę na danym obszarze, kolorem czerwonym oznaczono wagę 1 dla danej kości, a kolorem różowym oznaczono kości, które nie mają stworzonej mapy do deformacji. Na rysunku 3.17 pokazano jak wygląda mapa wag dla kości. Kości można zmieniać poprzez kliknięcie lewym przyciskiem myszy kości



Rysunek 3.16: Wstępne wyliczenie wag kości

z wciśniętym klawiszem ‘CTRL’. Do malowania wag są dostępne różne pędzle. Zostały one opisane bardzo dobrze w filmie [14] – narzędzia pokazane są na starszej wersji programu, ale sam film jest nadal aktualny.



Rysunek 3.17: Wstępne wyliczenie wag kości dla kości przedramienia

### **3.4. Kluczowanie klatek – animowanie w programie Blender**

Animacja jest to – ogólnie mówiąc – ruch pewnego obiektu na obrazie. W grafice 2D zwykle odbywa się poprzez zmianę obrazów. W grafice 3D jest to bardziej skomplikowane, ponieważ polega na zmianie stanu obiektu w czasie. W grach komputerowych animacje nie wymagają takiej jakości jak w przypadku filmów. Jakość animacji w danym segmencie gry często jest zależna od dynamiki segmentu – jeżeli gra jest szybka, gracz nie będzie zwracał dużej uwagi na animacje. Tworzenie animacji jest również czasochłonne, a im bardziej dokładna animacja tym bardziej czasochłonne są nanoszone ewentualne poprawki. Dlatego do tworzenia animacji używa się technologii ‘Motion Capture’ – technologia pozwalająca na przechwytywanie ruchu aktorów. Problemem tego rozwiązania jest jego koszt – sprzęt jest drogi, a dane trzeba przetworzyć co też wymaga czasu. Kontrą do tego rozwiązania jest własnoręczne tworzenie animacji w programach graficznych.

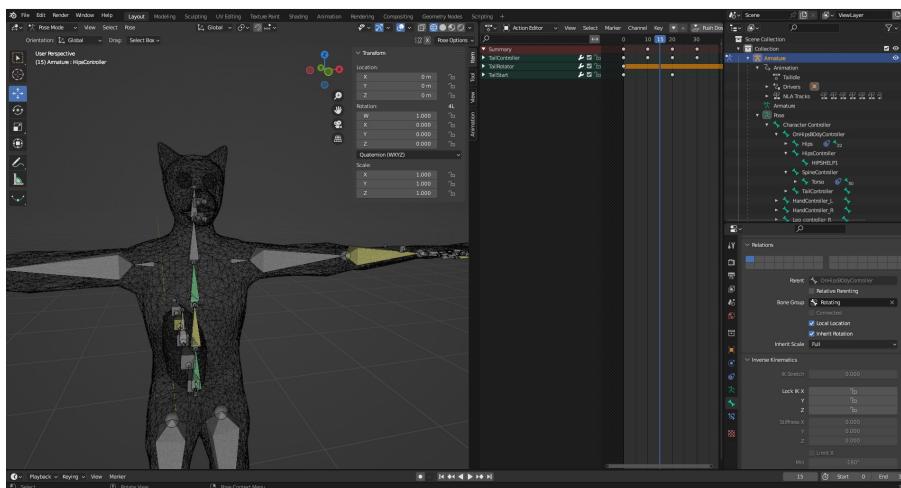
Ręczne tworzenie animacji wymaga kluczowania klatek. Jest to proces, w którym zapisuje się transformacje i inne właściwości. W tym celu Blender udostępnia wiele narzędzi pozwalających na animowanie. Na rysunku 3.18 przedstawiono zestaw pól roboczych używanych do animacji. Składają się na nie:

- Pierwszym polem roboczym – najbardziej po lewej – jest ‘3D Viewport’.
- Na środku znajduje ‘Action Editor’ z pola roboczego ‘Dope Sheet’.
- Po lewej stronie okna znajdują się właściwości i zawartość sceny.
- Na dole jest ’Timeline’

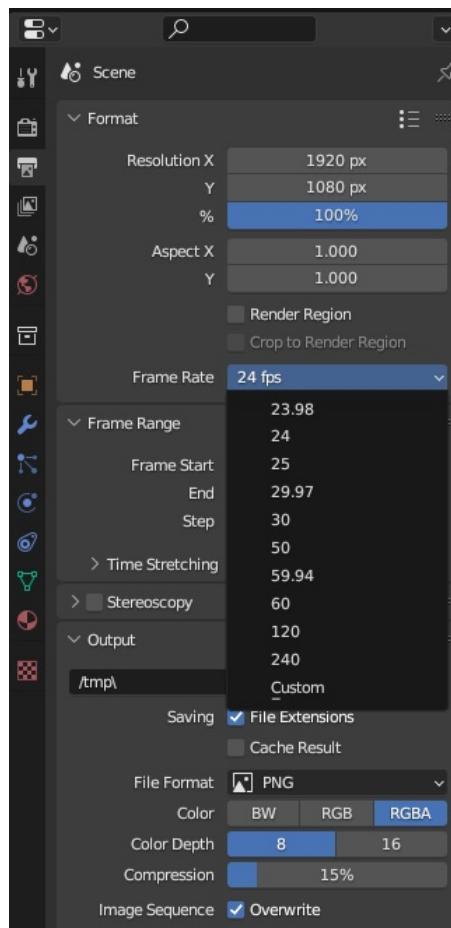
Taki zestaw pozwala na stworzenie zestawu animacji bez potrzeby przełączania się między polami roboczymi. Animacje zostają zapisane w jako tzw. Akcje dzięki czemu po wyeksportowaniu wszystkie animacje i model znajdują się w jednym pliku.

Domyślnie 24 klatki oznaczają jedną sekundę animacji. Można to zmienić w polu roboczym ‘Properties’, w zakładce ‘Output Properties’ zakładka ‘Format’ pole Frame Rate (pokazano na rysunku 3.19). Warto zaznaczyć, że im większy ‘Frame Rate’ tym większy będzie wyeksportowany plik, ale animacje będą bardziej szczegółowe.

Przed rozpoczęciem animowania należy ustawić ile klatek ma trwać dana animacja – domyślnie klatką początkową jest klatka 0, a końcową ostatnia zapisana klatka.



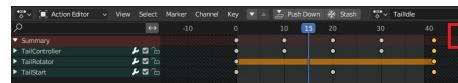
Rysunek 3.18: Zestaw pól roboczych używanych do animacji



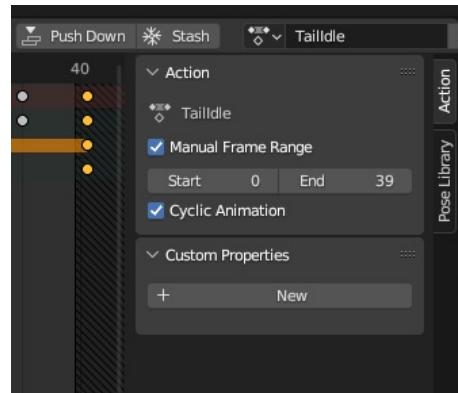
Rysunek 3.19: Zmiana Frame Rate

Na obranie 3.20 można zobaczyć strzałkę, która otwiera menu, które pozwala na zmianę początku i końca animacji pokazanego na rysunku 3.21. Poza tym można określić czy

dana animacja jest zapętlona – pole "Cyclic Animation".



Rysunek 3.20: Strzałka prowadząca do menu pozwalającego na zmianę klatek początku i końca animacji



Rysunek 3.21: Menu zmiany początku i końca animacji

Zmiana klatki odbywa się poprzez kliknięcie na liczbę reprezentującą klatkę na linii czasu (rysunki 3.22 i 3.23 pola 1)



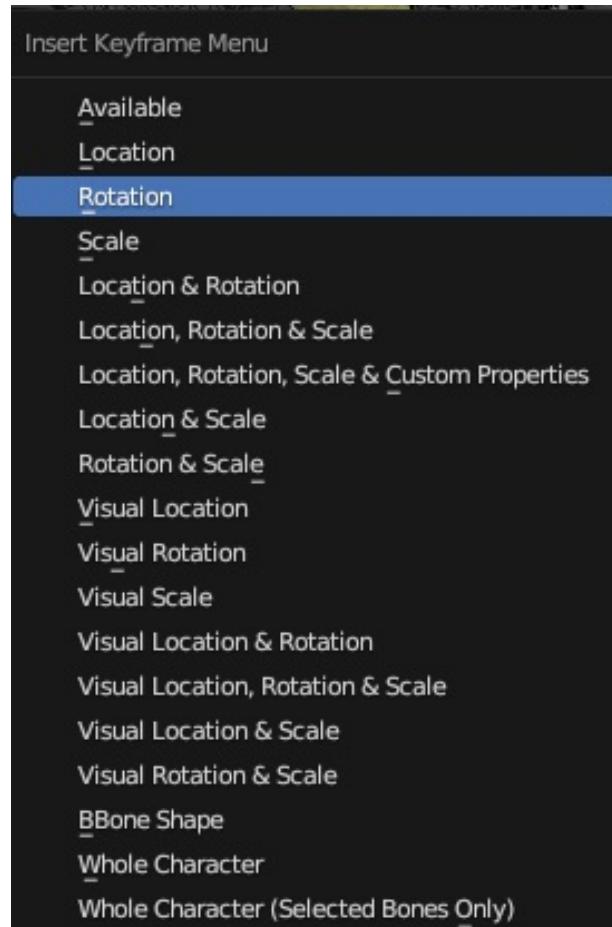
Rysunek 3.22: Linia czasu w polu roboczym Dope Sheet w trybie ActionEditor



Rysunek 3.23: Linia czasu w polu roboczym TimeLine

Kluczowanie klatek odbywa się poprzez zaznaczenie kości – jednej lub kilku – wciśnięcie klawiska 'I'. Wtedy pojawi się menu (rysunek 3.24), w którym można wybrać jakie wartości będą zapisywane w klatce animacji. Warto wspomnieć o różnicach

między poszczególnymi opcjami kluczowania klatki. ‘Visual Location’ zapisze lokację kości jaką widać na podglądzie. Opcja ta jest używana głównie w przypadku używania modyfikatorów. Zwykle ‘Location’ odnosi się do zmiany pozycji przez użytkownika. ‘Custom Properties’ mówi o innych właściwościach stworzonych przez użytkownika. ‘BBone Shape’ jest używany w przypadku specjalnych kości nazywanych ‘Bendy Bones’. ‘Whole Character’ kluczuje wszystkie właściwości wszystkich kości, jest to rzadko używane.



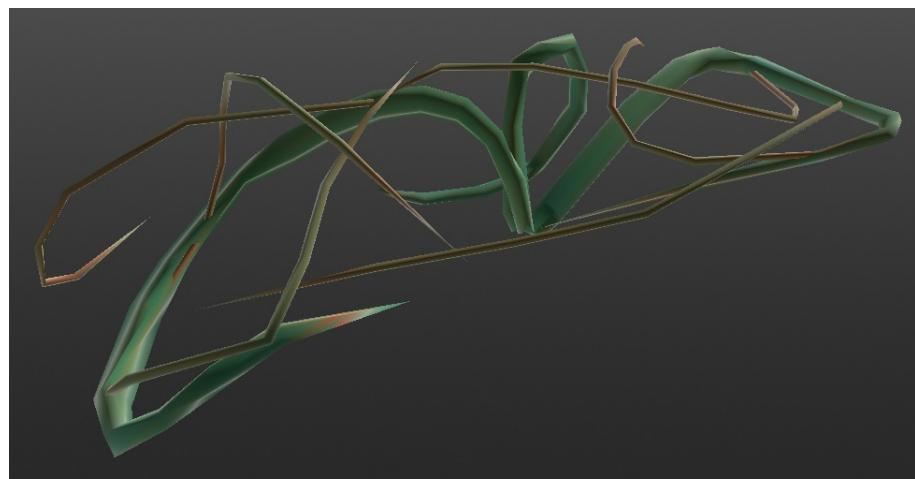
Rysunek 3.24: Menu wyboru zapisanych informacji do klatki animacji

## 4. Przygotowanie modeli, animacji i tekstur

Do projektu zostały przygotowane 2 modele z animacjami i teksturami – ciernie i postać główna.

### 4.1. Ciernie

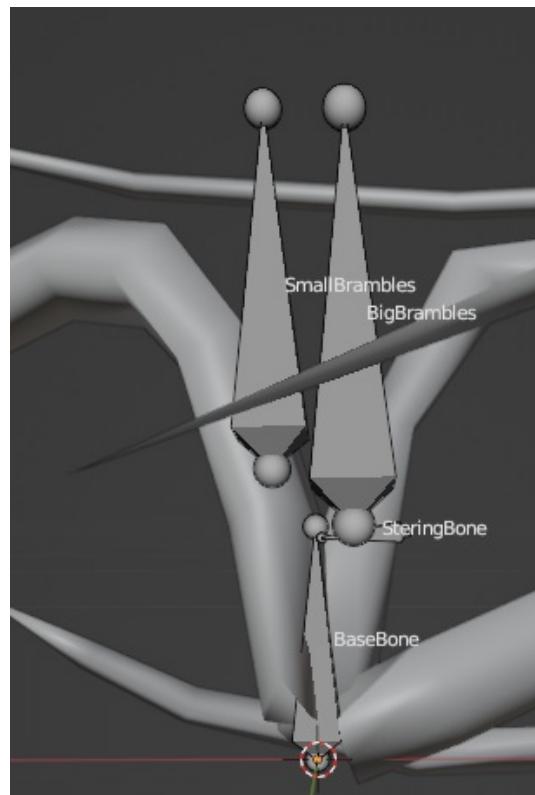
Model cierni został wykonany w stylu Low Polly – bardzo mała ilość detali, model może być kanciasty. Model został podzielony na 2 części – grubsza i chudsza część. Każda z tych części jest sterowana osobną kością. Zostały stworzone 3 animacje cierni – ‘FastFloating’, ‘SlowFloating’ i ‘Touched’. ‘FastFloating’ i ‘SlowFloating’ są animacjami cyklicznymi i mają symulować wpływ wiatru na ciernie. Animacja ‘Touched’ jest zachowaniem się cierni na wypadek dotknięcia ich przez postać główną.



Rysunek 4.25: Model cierni z nałożoną tekstrurą

#### 4.1.1. Szkielet

Szkielet jest bardzo prosty i składa się z 4 kości. Podstawową kośćią, która służy do przemieszczania i rotowania całym modelem jest ‘BasicBone’. Kości ‘SmallBrambles’ i ‘BigBrambles’ służą do poruszania odpowiednio cieńską i grubszą częścią. Kość ‘SteringBone’ służy do poruszania kośćmi ‘SmallBrambles’ i ‘BigBrambles’



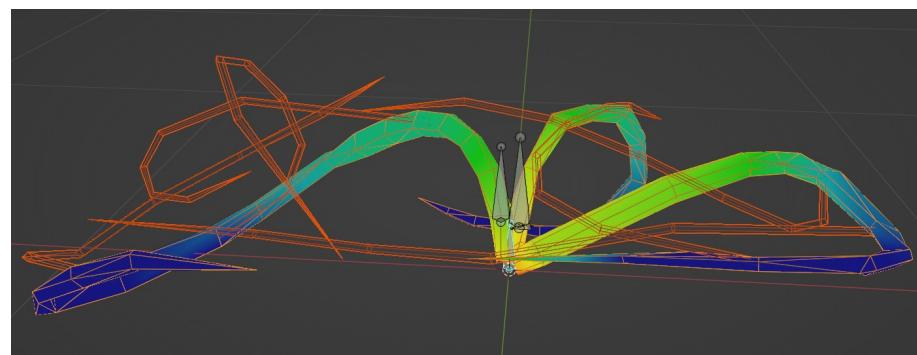
Rysunek 4.26: Położenie kości szkieletu cierni



Rysunek 4.27: Hierarchia kości dla skieletu cierni

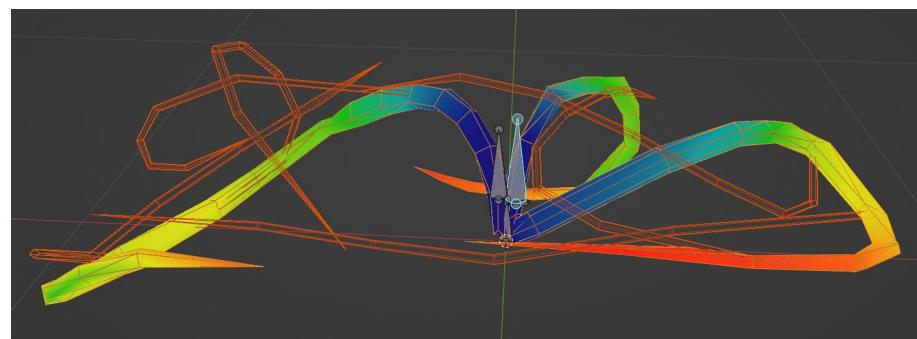
#### 4.1.2. Weight Paint

Kość ‘BaseBone’ wpływa tylko na grubszą część modelu (rysunek 4.28). W ten sposób ‘Korzeń’ modelu jest nieruchomy.



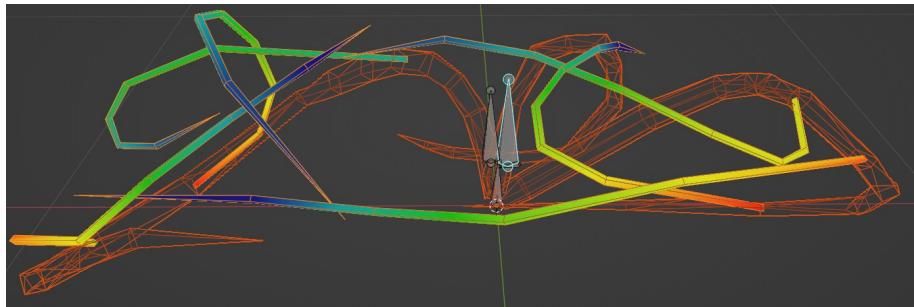
Rysunek 4.28: Wpływ kości ‘BaseBone’

Kość ‘BigBrambles’ wpływa na 2 części modelu. Na rysunku 4.29 przedstawiono mapę wag tej kości na modelu grubszych cierni. Rysunek 4.30 przedstawia wpływ tej kości na cieńsza część modelu. Takie rozwiążanie pozwoliło na połączenie ze sobą tych dwóch części. Częstym problemem w animacjach jest rozłączanie się modeli podzielonych na części.

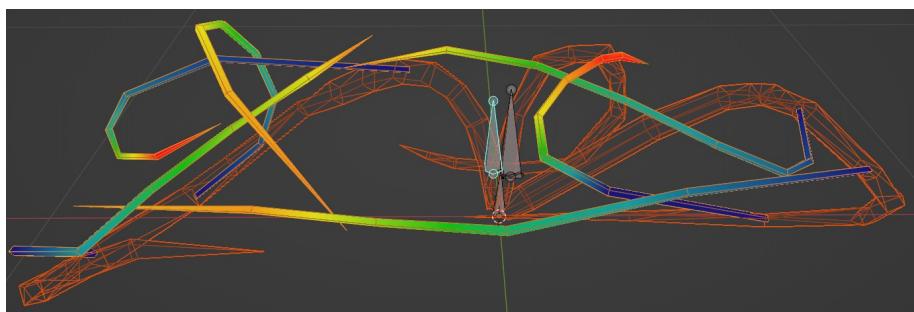


Rysunek 4.29: Wpływ kości ‘BigBrambles’ na grubszą część modelu

Kość ‘SmallBrambles’ odpowiada za deformacje cieńszych gałęzi modelu. Pokazano to na rysunku 4.31.



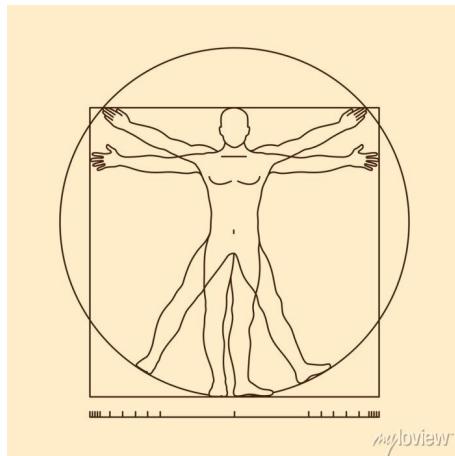
Rysunek 4.30: Wpływ kości 'BigBrambles' na cieńszą część modelu



Rysunek 4.31: Wpływ kości 'SmallBrambles' model

## 4.2. Postać główna

Pozą spoczynkową – w pozie spoczynkowej kości nie są transformowane – jest tzw. ‘T pose’ – postawa postaci przypomina literę T. Model postaci jest skierowany twarzą wzdłuż osi Y. Powodem tego jest fakt, że edycja kości w programie może działać lustrzanie, ale tylko w wypadku, gdy są one odbite lustrzanie wg. osi Y. Jako wzór do zachowania pewnych proporcji – np. stosunek rąk do tułowia – został użyty obraz człowieka witruwiańskiego (rysunek 4.32). Szkielet został lekko zniekształcony – zostały wydłużone kończyny – tak, aby kształtem wyglądał jak wilkołak. Proces używania obrazu jako referencji został pokazany w filmie [15]. Krokiem pierwszym stworzenia postaci głównej było zrobienie szkieletu, a następnie zrobienie modelu. W czasie tworzenia animacji szkielet zmienił się kilka razy. Powody zmian zostały zawarte w dziale o szkielecie postaci głównej.



Rysunek 4.32: Szkic człowieka witruwiańskiego

### 4.2.1. Szkielet animacji

Kości z końcówką L mają swój lustrzany odpowiednik R – gdy zostanie stworzona połowa szkieletu, to można ją symetrycznie odbić wzdłuż osi Y, a nazewnictwo jest wymagane, bo w przeciwnym razie kości nie są lustrzanie odbijane. Kości zostały podzielone na 2 grupy: kości deformujące i kości kontrolujące. Kości kontrolujące kontrolują za pomocą modyfikatorów kości kilka kości na raz (lub jedną w bardzo specyficzny sposób). Modyfikator IK (Inverse Kinematic) został użyty do kontroli palców, rąk i nóg. Inverse Kinematic jest skomplikowanym modyfikatorem i używanie go do animacji wielu kości, które mogą być rotowane wokół wielu osi może doprowadzić do niechcianych

ruchów. W przypadku kręgosłupa zamiast 24 kręgów zastosowano 5 kości – kość deformująca głowę (Head), kark (Neck), klatkę piersiową (Torso), biodra(Hips) i jedna kość między biodrami a klatką piersiową (Spine\_2) – co zostanie bardziej szczegółowo omówione w dalszej części pracy. Kość ‘CharacterConteroller’ została użyta do transformacji wszystkich kości, a kość ‘OnHipsBODYController’ do transformacji bez kości ‘HandCOnroller’ i ‘LegController’. Jest to spowodowane potrzebą odseparowania kończyn – nóg i rąk – od reszty ciała. Hierarchia kości szkieletu została przedstawiona na rysunkach

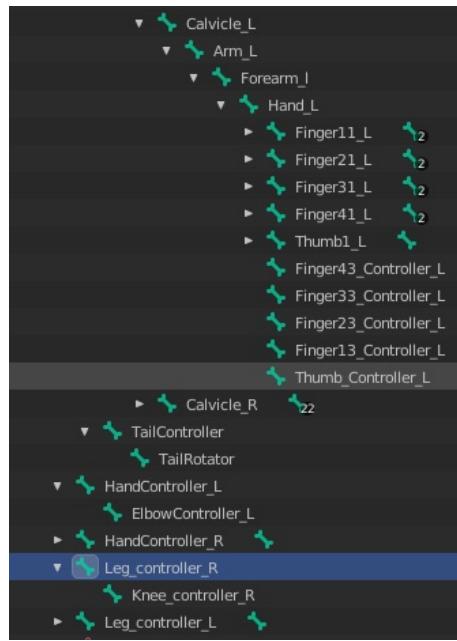
4.33

i

4.34.



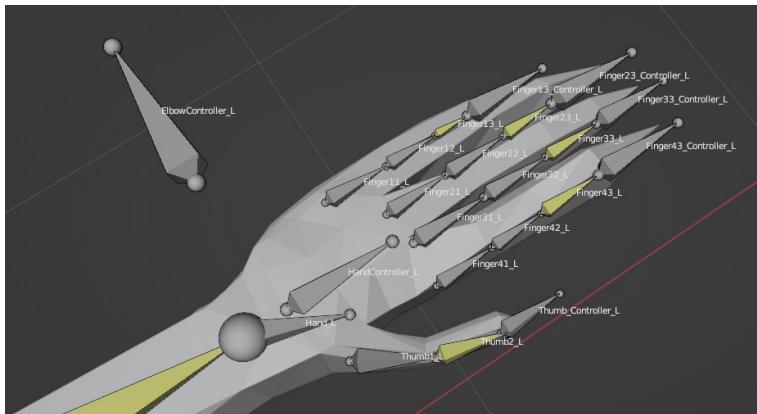
Rysunek 4.33: Hierarchia kości część 1



Rysunek 4.34: Hierarchia kości część 2

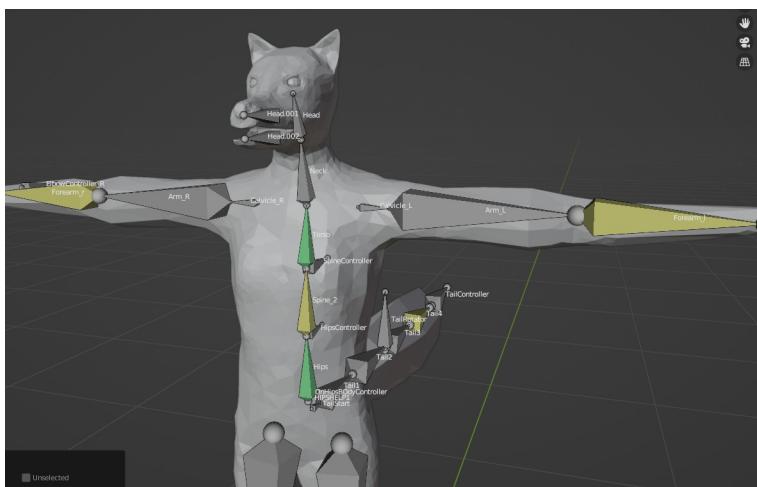
Żółte kości na rysunkach 4.35, 4.36 i 4.37 posiadają modyfikator IK, a na zielono posiadające modyfikator ‘CopyTransform’ lub ‘CopyLocation’. Kości, które są modyfikowane przez modyfikator IK z kościami ‘Pole Target’ nie mogą być w jednej osi. Większość kości otrzymała maksymalny kąt zgięcia w danej osi pod wpływem modyfikatora IK, co pozwoliło na np. bardziej realistyczne zginanie się kości. Opcja ta jest dostępna w polu roboczym ‘Properties’ -> ‘Bone Properties’ -> zakładka ‘Inverse Kinematics’ -> opcje ‘Limit X’, ‘Limit Y’ i ‘Limit Z’. Było to bardzo użyteczne podczas np. zginania ręki w łokciu – ręka w łokciu może się zginać tylko w jednym kierunku, a rotacja ramienia odbywa się w chrząstce między kością ramieniową, a obojczykiem. Na zasadzie obserwacji i metody prób i błędów udało się ustalić maksymalne kąty zgięcia w danym miejscu. Nie działa to perfekcyjnie, ale bardzo ułatwia pracę podczas tworzenia animacji. Blender posiada błąd, który powoduje, że modyfikator nie działa w takim wypadku poprawnie. Do obracania całej dłoni służy kość ‘Hand’, która ją też deformuje. Każdy z palców dłoni – poza kciukiem, który składa się z 2 kości – składa się z 3 kości deformujących sterowanych jedną kościami – np. ‘Finger33\_Controller\_R’ – za pomocą modyfikatora IK. Kontrolery palców za rodzica mają kość nadgarstka. Sterowaniem całą ręką – 2 kości, na kość obojczyka ‘calvicle’ nie ma wpływu - zajmuje się kość ‘HandController’ za pomocą modyfikatora IK przypiętego do kości ‘Forearm’. Rodzicem kości ‘HandController’ jest kość główna ‘Character Controller’. Jako kość rotacyjna ustawiona jest

kość ‘ElbowController’, której rodzicem jest kość ‘HandController’.



Rysunek 4.35: Kości Ręki

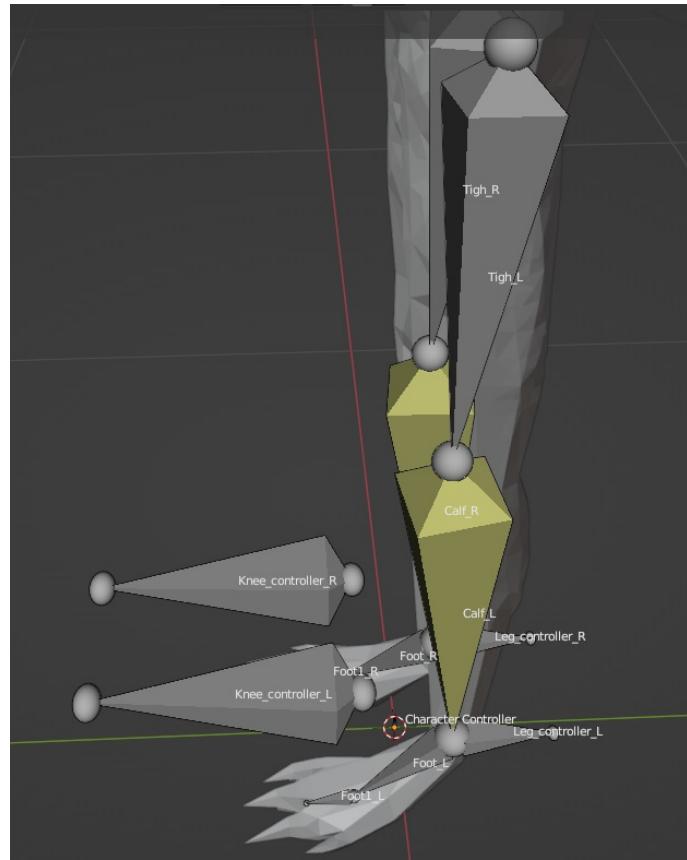
Kości kręgosłupa są sterowane inaczej niż kości nóg czy rąk. Kręgosłup jest częścią ciała, która jest najbardziej giętka i dynamiczna. W porównaniu do kończyn dolnych i górnych wymaga większej kontroli. Zastosowanie sterowania za pomocą jedynczego modyfikatora IK powodowało problemy – lekkie poruszanie powodowało bardzo duże zmiany w transformacjach kości. Dlatego na podstawie filmu [16] został zaimplementowany układ sterowania tułowiem. Nowy układ pozwala na swobodne sterowanie odpowiednimi częściami tułowia. Do sterowania deformacją w okolicach klatki piersiowej i bioder służą kości ‘SpineController’ i ‘HipsController’ (rysunek 4.36).



Rysunek 4.36: Kości górnej części ciała

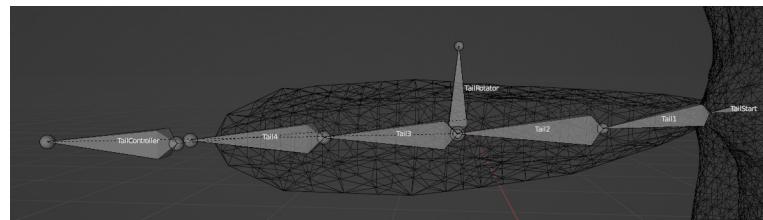
Sterowaniem nogą są używane 2 kości – kość kontrolera ‘Leg\_controller’ i kość służąca do rotacji ‘Knee\_controller’ – które sterują dwoma kośćmi – ‘Tigh’ i ‘Calf’.

Stopa składa się z dwóch kości i każda animowana jest osobno oraz kości palców, które nie są używane do animacji.



Rysunek 4.37: Kości dolnej części ciała

Ogon składa się z 7 kości – 5 kości deformujących, kości sterującej ‘TailController’ i kości rotującej ‘TailRotator’. ‘TailController’ steruje 4 z 5 kości deformujących – kości ‘Tail1-4’, gdzie ‘Tail4’ jest kośćią końcową. Kość ‘TailStart’ posłużyła do poprawnego ułożenia ogona względem ciała – np. podczas animacji biegu tułów i ogon powinny znajdować się w jednej płaszczyźnie.

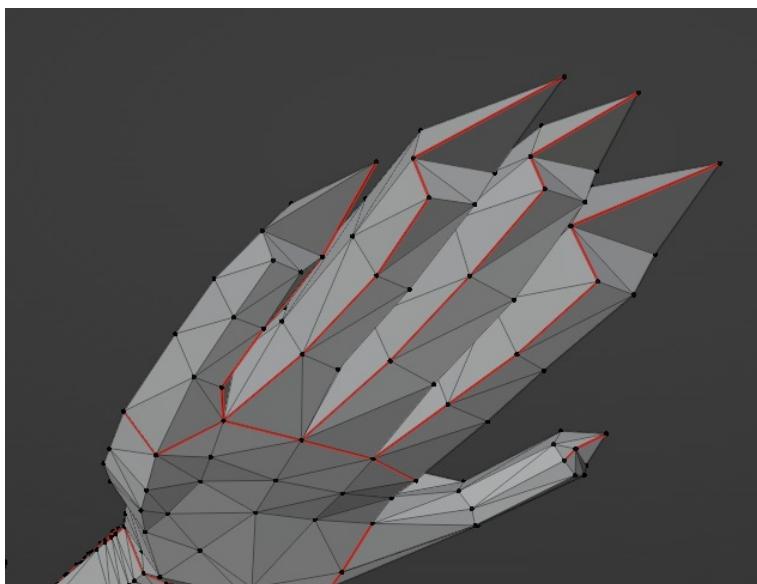


Rysunek 4.38: Kości ogona postaci

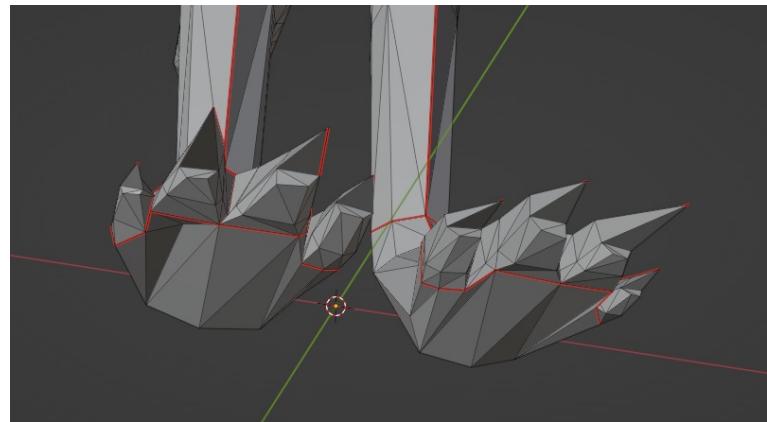
#### 4.2.2. Model i Tekstury

Model postaci (wraz z teksturami na rysunku 4.42 i rysunku 4.43) został przygotowany w stylu medium poly - określenie na modele graficzne posiadające od 1000 do 15000 ścian, cechuje się ilością detali. Model miejscami jest kanciasty, a częściami ciała, na których to najbardziej widać są: dlonie (rysunek 4.39), stopy (rysunek 4.40) i ogon 4.41. W grze dlonie i stopy są mało widoczne, przez co nie jest to widoczne. Ogon mimo posiadania wielu punktów nie został dobrze ‘wyrzeźbiony’. Ogon postaci wymagał bardziej skomplikowanej siatki ze względu na animacje.

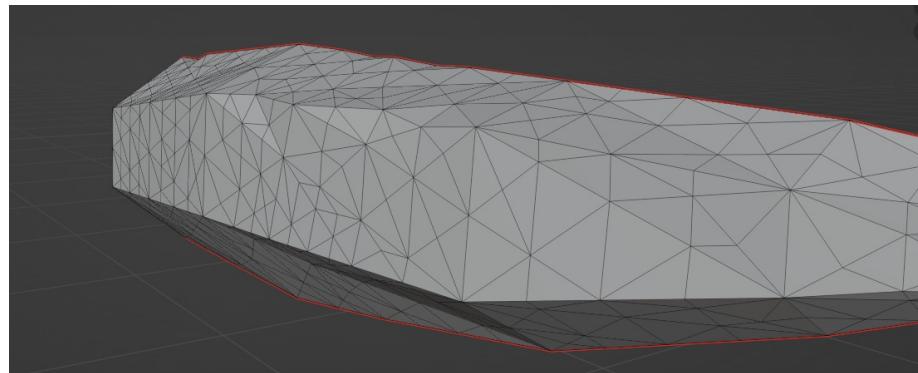
Przed przystąpieniem do robienia tekstur należy zrobić ‘UV maping’. Tekstury są dwu-wymiarowymi mapami nałożonymi na trójwymiarowy obiekt. Proces nakładania tekstur można porównać do procesu oklejania papierem obiektu. W niektórych przypadkach obiekt można okleić bez cięcia papieru. W większości przypadków papier trzeba pociąć wiele razy. Blender do tego celu posiada zestaw pól roboczych – ‘UV Editing’. Na rysunkach 4.39, 4.40 i 4.41 na czerwono oznaczono cięcia, które będą zachodzić podczas generowania mapy UV. Proces ten bardzo dobrze opisany – wraz z rzecząmi, na które należy uważać – został w filmach [18] i [17].



Rysunek 4.39: Ręka postaci głównej



Rysunek 4.40: Stopa postaci głównej



Rysunek 4.41: Ogon postaci głównej

Tekstury zostały wykonane w programie Quixel Mixer (rysunek 4.42). Tekstury zostały wykonane w programie Quixel Mixer. Program działa podobnie do programu Gimp – tworzenie grafik za pomocą warstw, ale dla modeli 3D. Program przyjmuje model stworzony za pomocą innych programów np. Blender lub Maya w formacie FBX. Największym problemem programu jest jego prędkość działania – wymaga on mocnej karty graficznej.



Rysunek 4.42: Oteksturowana postać – widok od przodu

Po lewej stronie rysunku 4.43 znajdują się warstwy. Ikona folderu oznacza grupę. Każda warstwa, która znajduje się w grupie, maluje tylko obszar dozwolony przez maskę w grupie. Maska może być wygenerowana – za pomocą szumów – lub malowana ręcznie. Każda warstwa może posiadać: poziom przezroczystości – ten zawsze musi być zdefiniowany – kolor, metalowość, chropowatość i kilka innych. Każdy z tych atrybutów może przyjmować wartość od 0 do 1. Materiałem może być też sam szum, który może wpływać na materiały poniżej niego. Materiał lub grupa, która jest najwyżej będzie najbardziej na wierzchu.



Rysunek 4.43: Postać główna w QuixelMixer

#### 4.2.3. Animacje

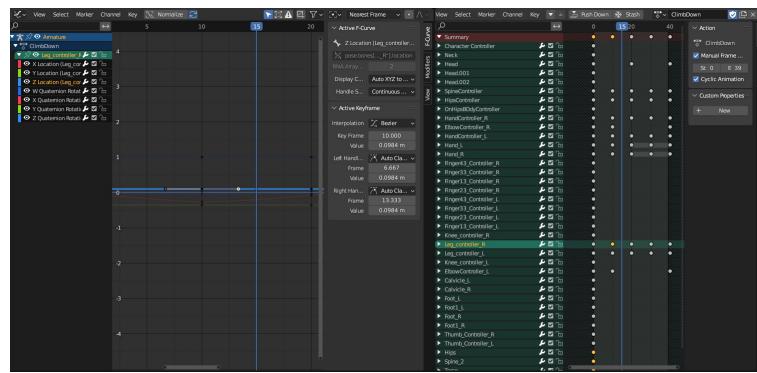
Animowanie są tematem złożonym, którego nie da się opisać w prosty sposób, a samo tworzenie animacji jest zależne od wielu czynników. Pierwszym krokiem przed rozpoczęciem animowania powinna być obserwacja ruchów zwierząt lub ludzi, którzy wykonują podobne ruchy, jakie mają znaleźć się w animacji. W dzisiejszych czasach, gdzie internet jest ogólnodostępny jest to proces jedynie czasochłonny. Obserwowany ruch następnie należy rozbić na etapy, a w każdym etapie rozbić ruch na części ciała – np. nogi, ręce tułów, głowa. Przykładowo chód ludzi – bez brania pod uwagę momentu startu i momentu stopu – ruch nóg można podzielić na 7 etapów – w uproszczeniu – gdzie jeden etap się powtarza:

- rozkrok dla uproszczenia lewa nogą jest z tyłu, biodra znajdują się w najniższym możliwym punkcie (podczas chodu), lewa ręka jest wysunięta do przodu, prawa ręka jest wysunięta do tyłu
- podniesienie nogi lewej, przeniesienie punktu ciężkości ciała na prawą nogę, co powoduje lekkie przechylenie w prawo
- nogą lewą wyprzedza nogę prawą, biodra znajdują się w najwyższym możliwym punkcie (podczas chodu), maksymalnie przechylenie ciała w prawo
- postawienie nogi lewej, nogi ponownie znajdują się w rozkroku prawa ręka wysunięta do przodu, prawa ręka wysunięta w tył, biodra znajdują się w najniższym możliwym punkcie (podczas chodu)

- podniesienie nogi prawej, przeniesienie punktu ciężkości ciała na lewą nogę, co powoduje lekkie przechylenie w lewo
- noga prawa wyprzedza nogę lewą, biodra znajdują się w najwyższym możliwym puncie, maksymalnie przychylanie ciała w prawo
- rozkrok dla uproszczenia lewa nogą jest z tyłu, biodra znajdują się najniżej, lewa ręka jest wysunięta do przodu, prawa ręka jest wysunięta do tyłu – powrót do pozycji startowej

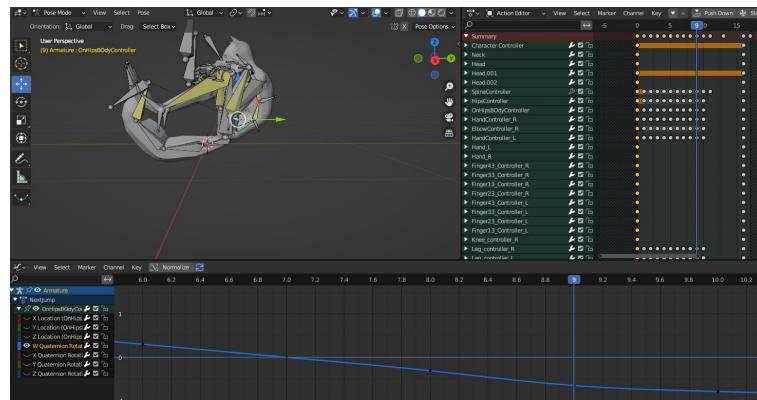
Dla bardziej zaawansowanych animacji taka rozpiska będzie bardziej obszerna – będzie zawierać informacje o np. napięciach mięśni. Niestety dla animacji twarzy taka rozpiska jest ciężka do zrobienia i zwykle twórcy te animacje robią metodą ‘motion capture’ lub ‘na czuja’. Każdy z tych etapów będzie kluczowaną klatką. W trakcie animowania należy zwracać uwagę na synchronizację między częściami ciała. Trzeba zwrócić uwagę czy dana animacja ma być zapętlona czy nie. W przypadku zapętlonych animacji ostatnią klatką animacji – dlatego klatki stop i start powinny być ustawiane ręcznie – nie może być pozycja startowa, ponieważ dojedzie do podwojenia klatki – postać nienaturalnie zatrzyma się.

Ostatnim etapem powinno być ustawienie czasu trwania poszczególnych etapów. Dla wyżej podanego przykładu, czas podniesienia między postawieniem nogi prawej, a podniesieniem nogi lewej powinien być bardzo krótki. Czas między podniesieniem, a postawieniem nogi powinien być najdłuższą częścią animacji – ok. 45% czasu animacji dla jednej nogi, czas liczony od początku do końca animacji. W programach graficznych jest możliwa edycja przebiegu animacji określonych kości między kluczowanymi klatkami – pole robocze ‘Graph Editor’ (rysunek 4.44). W edytorze grafów można bardziej precyjnie ustawić wartości zmieniane w danej chwili. Jest to szczególnie przydatne w trakcie obracania obiektów. Kwantony są systemem reprezentacji obrotu wokół pewnej osi i ten system jest bardziej rekompensuje wady zastosowania rotacji przez macierze rotacji. Normalizacja kwaternionów powoduje przejście zmiennej w przestrzeni w – odpowiedzialnej za kąt obrotu – z -1 do 1. W obliczeniach matematycznych nie robi to wielkiej różnicy, ale w przypadku animacji, gdy obiekt kręci się wokół osi może doprowadzić do obrotu w inną stronę. Dlatego przy pomocy edytora grafów można zmienić tę wartość tak, aby została przy wartości -1.



Rysunek 4.44: Edytor grafów w programie Blender

W animacji o nazwie ‘NextJump’ postać robi przewrót w przód w powietrzu (rysunek 4.45). Problemem okazało się znormalizowanie kwaternionu kości ‘OnHipsBODYController’ i postać zamiast przejść do pozycji wyjściowej animacji zrobiła obrót w przeciwną stronę. Najlepszym rozwiązaniem okazała się edycja przebiegu wartości ‘W’ rotacji – na dolnej części rysunku 4.45 przedstawiono poprawioną wersję przebiegu zmiennej.



Rysunek 4.45: Animacja ‘NextJump’ z edytorem grafów

Poza tym postać główna posiada 18 innych animacji:

- Animacja ‘Idle’ – postać czeka (pokazano na rysunku 4.46)
- Animacja chodzenia (rysunek 4.49) i biegu (rysunek 4.50)
- Animacje ogona – animacja spoczynku ogona i 4 animacje związane z ruchem postaci
- Animacja skoku

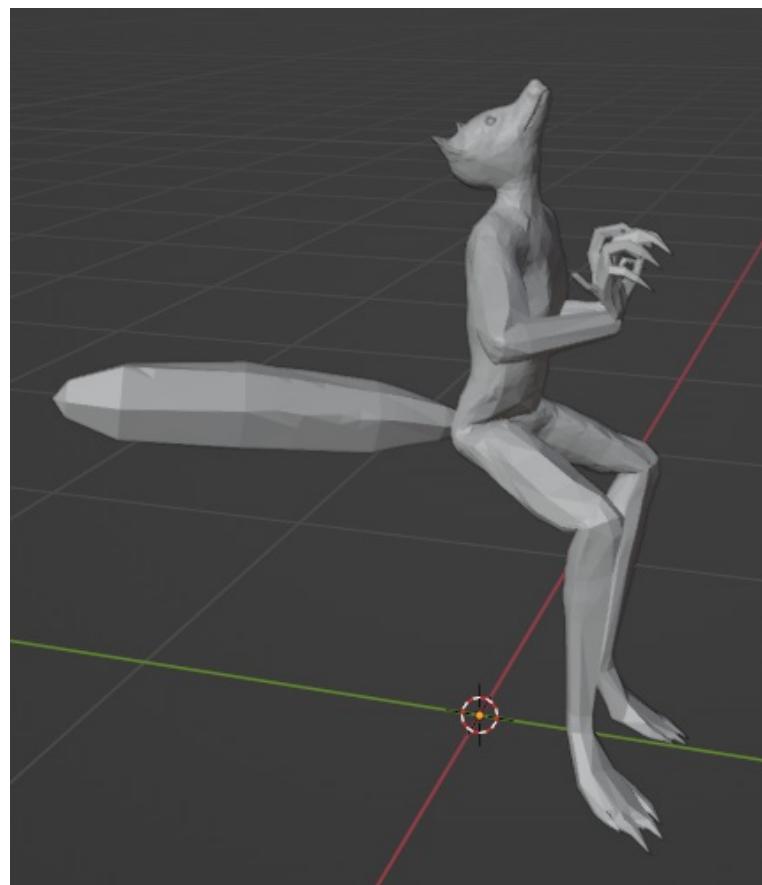
- Animacje wspinania (rysunek 4.47) i podciągnięcia na krawędzi (rysunek 4.48)
- Animacja spadania
- Animacja śmierci postaci (rysunek 4.51)



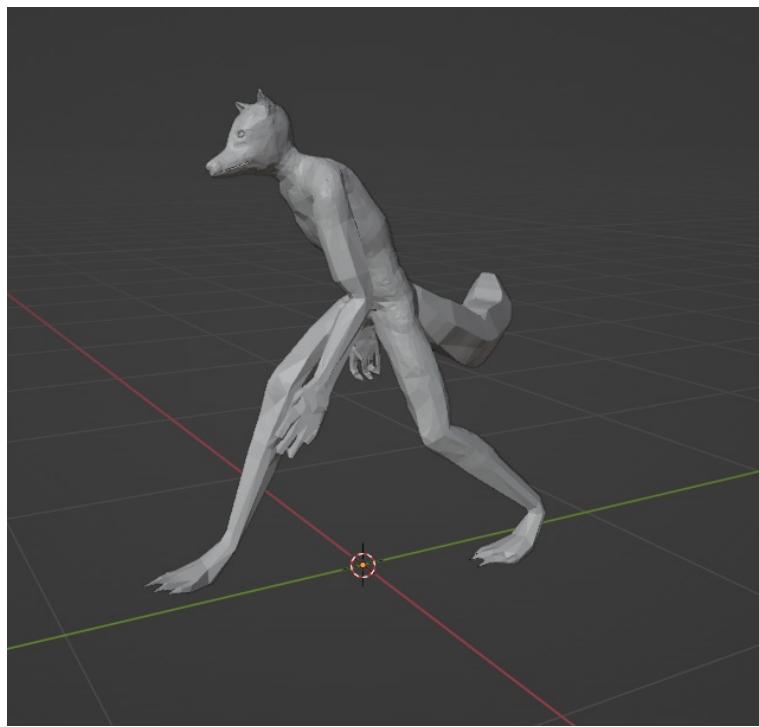
Rysunek 4.46: Animacja 'Idle' – postać oczekuje stojąc na ziemi



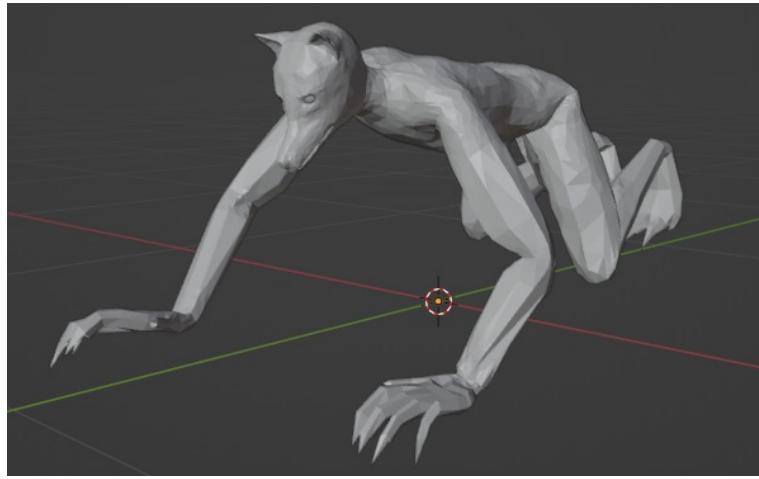
Rysunek 4.47: Animacja 'ClimbWait' – oczekiwanie podczas wspinania



Rysunek 4.48: Animacja 'ClimbOnEdge' – animacja podcięgnięcia na krawędzi



Rysunek 4.49: Animacja 'MoveLoop' – animacja chodu postaci



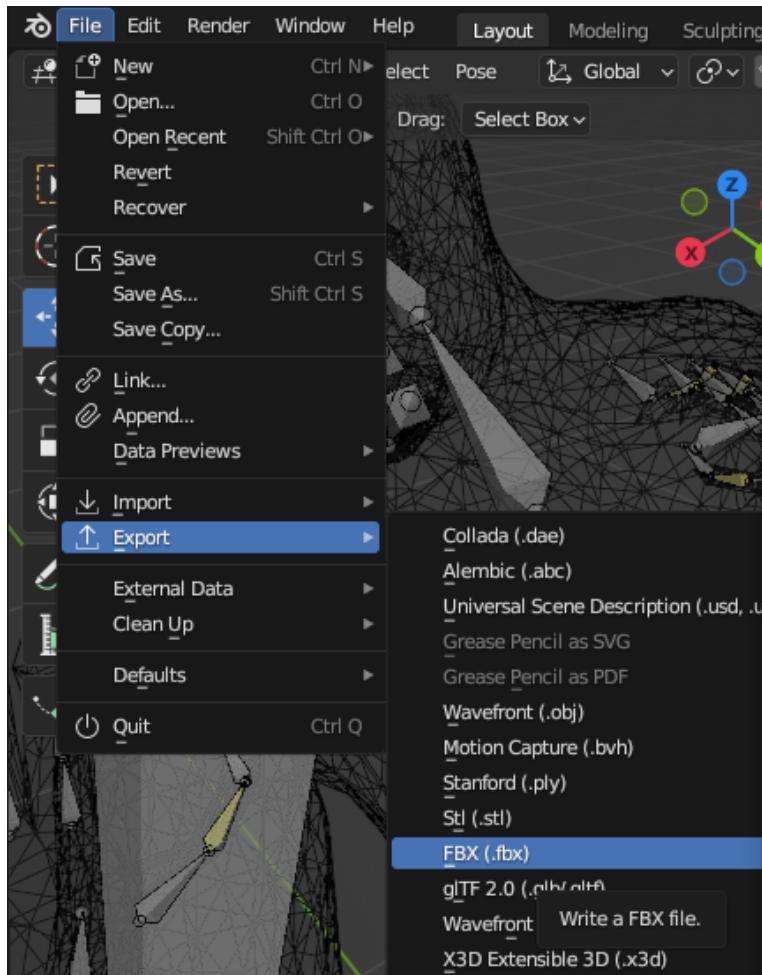
Rysunek 4.50: Animacja 'Spring' – animacja biegu postaci



Rysunek 4.51: Animacja 'Death' – animacja śmierci postaci

### 4.3. Eksportowanie modeli z animacjami

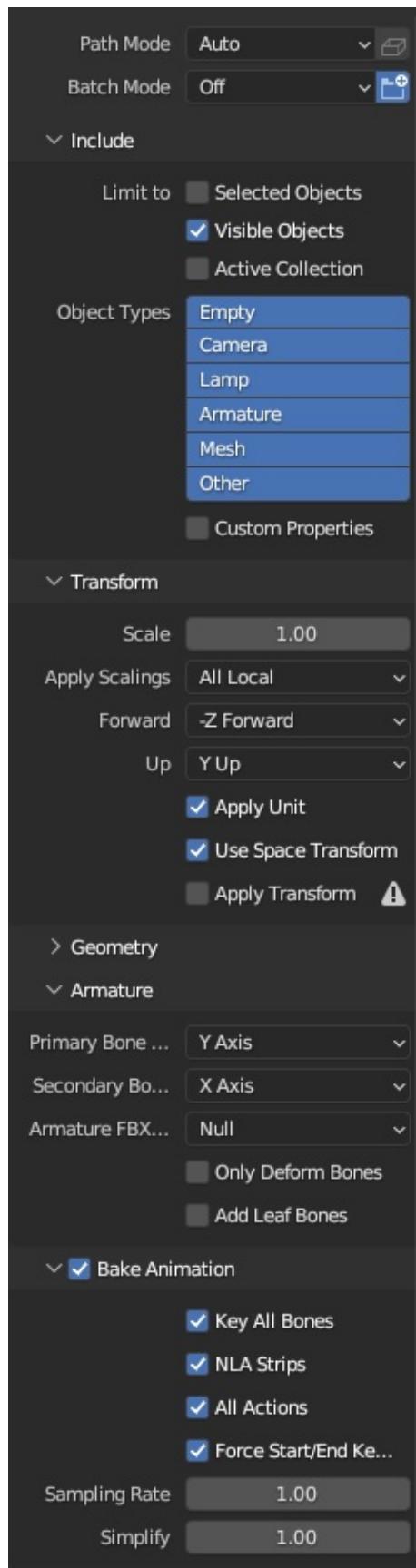
Pliki .blend mogą być używane bezpośrednio przez silnik unity, ale nie działa to poprawnie z animacjami. W tym celu należy ręcznie wyeksportować model z animacjami do pliku FBX. W tym celu należy przejść do zakładki ‘File’->‘Export’->‘FBX’ (rysunek 4.52).



Rysunek 4.52: Eksportowanie pliku FBX

Na rysunku 4.53 zostały pokazane zalecane opcje eksportu modelu z animacjami. W zakładce ‘Include’ należy zaznaczyć ‘Visible Objects’ przez co zostaną wyeksportowane obiekty, które są widoczne – obiekty można ukryć za pomocą klawisza ‘H’, a odkryć wszystkie obiekty kombinacją ‘Alt’ i ‘H’. Zakładka ‘Transform’ zawiera informacje jak obiekty będą zapisany m.in. jednostka, skala i informacje o kierunku osi – która osi wskazuje góre i przód, dla programu Blender będą to odpowiednio Z i Y. Należy zaznaczyć pola ‘Apply Unit’ i ‘Use Space Transform’. Pole ‘Apply Transform’ musi być

odznaczone. W zakładce ‘Armature’ odznaczyć pole ‘Add Leaf Bones’. Zaznaczenie tego pola spowoduje dodanie dodatkowych kości każdej kości, która nie ma potomka. Przez to działanie plik FBX stanie się dużo większy. Pole ‘Only Deform Bones’ jeżeli jest zaznaczone spowoduje usunięcie kości, które nie deformują. Zakładka ‘Bake Animation’ jest tutaj najważniejsza. Pole przy tej zakładce powinno być zaznaczone. Wypalanie animacji polega na zapisaniu właściwości w każdej klatce jeżeli zachodzi jakaś zmiana. ‘Key All Bones’ polega na zapisaniu transformacji każdej kości w klatce startowej animacji. ‘NLA Strips’ w przypadku tego projektu może być odznaczone – to pole odnosi się do narzędzia nieużywanego w tym projekcie. ‘All Actions’ musi być zaznaczone co pozwoli na zapisanie wszystkich animacji w jednym pliku. ‘Force Start/End Keying’ wymusza zapis klatki początkowej i końcowej, które zawierają właściwości zmieniane w danej animacji.



Rysunek 4.53: Opcje eksportowania

## **5. Skryptowanie i obróbka assetów w Unity**

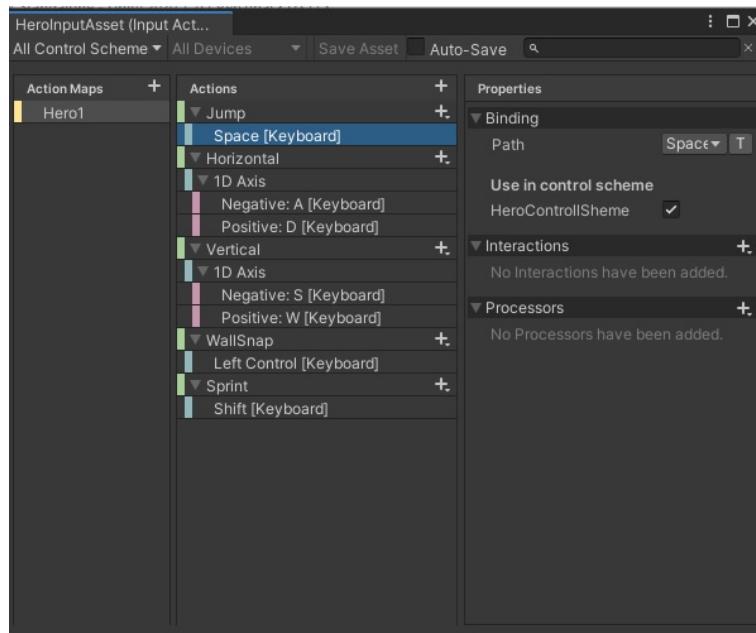
Do projektu została zainstalowana paczka ‘Input system’ za pomocą managera pakietów silnika Unity – ‘Window’-> ‘Packet Manager’. ‘Input system’ jest paczką, która unowocześnia odczytywanie sygnałów wejściowych – z klawiatury, padów, myszy itp. Zostały zainstalowane paczki dla środowiska Visula Studio Code o nazwie ‘Visual Studio Code Editor’. Skrypty były pisane z myślą o dalszym rozwoju.

### **5.1. Input manager – ustawienie managera wejścia**

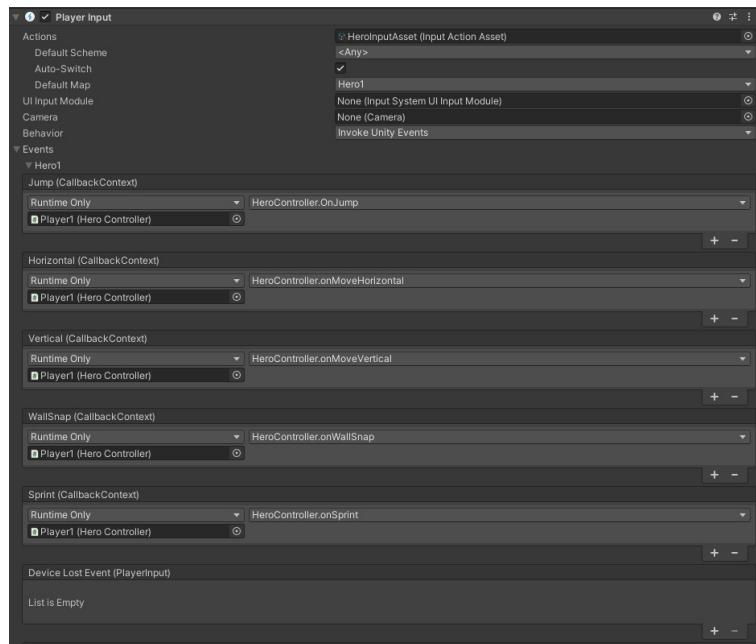
‘Input Manager’ jest bardzo przydatną rzeczą podczas tworzenia gier wieloplatformowych lub gier wymagających różnych kontrolerów. Przyciski i inne sygnały wejścia są przypisywane do akcji. Do jednej akcji może być przypisane wiele przycisków, jeden klawisz może być przypisany do wielu akcji. Wywołane akcje wywołują metody, które trzeba wcześniej przypisać w obiekcie. Jest też możliwość wczytywania stanu danej akcji w dowolnym momencie kodu – wymagany jest do tego specjalny obiekt.

Tworzenie akcji jest możliwe w trybie graficznym – jest opcja tworzenia ich za pomocą kodu. W tym celu trzeba najpierw stworzyć asset o nazwie ‘Input Action Asset’ – prawym przyciskiem myszy na miejsce, gdzie ma znajdować się dany Asset -> ‘Create’ -> ‘Input Actions’. Zostanie utworzona nowa mapa akcji. Po dwukrotnym kliknięciu na ten asset pokaże się okno pokazane na rysunku 5.54. Stworzenie i omówienie akcji zostało omówione w filmie [19].

Na rysunku 5.55 został pokazany skonfigurowany już komponent przypisany do obiektu gracza. Każda metoda została wzięta z jednego skryptu ale mogą być przypisane metody z różnych obiektów. Ważne, aby te metody miały za argument ‘InputAction.CallbackContext x’. Nazewnictwo metod jest dowolne.



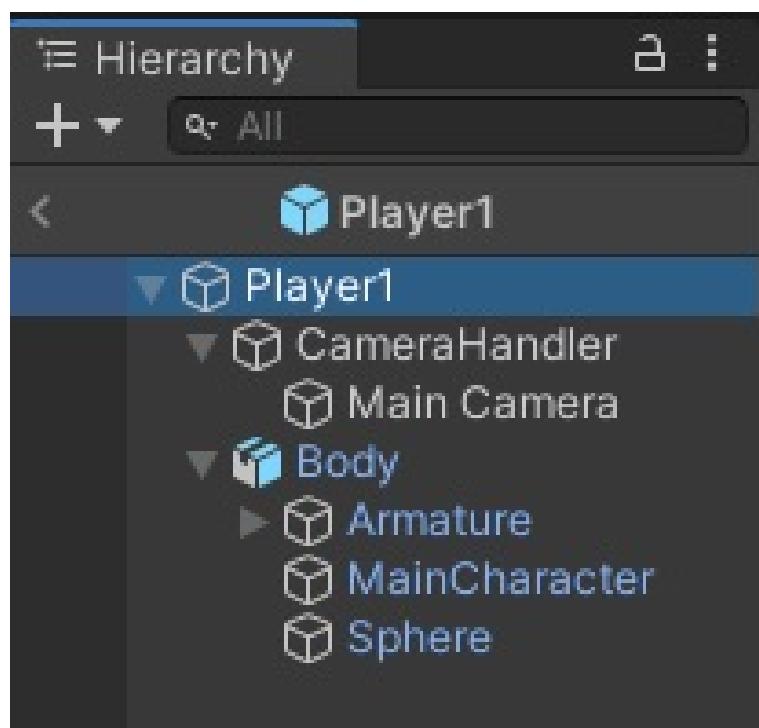
Rysunek 5.54: Mapa akcji stworzona w projekcie



Rysunek 5.55: Mapa akcji skonfigurowana w obiekcie gracza

## 5.2. Prefab gracza

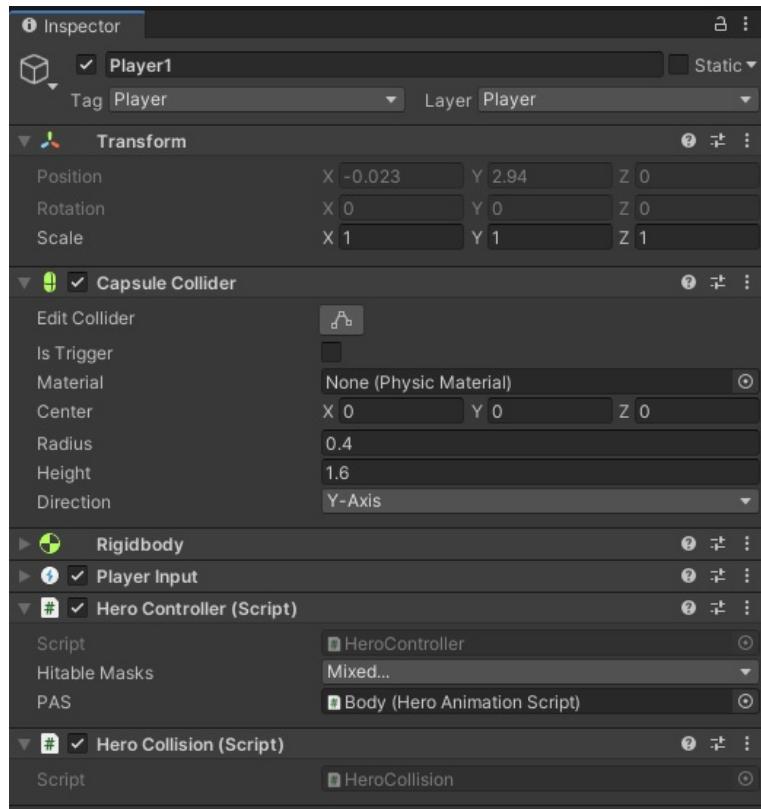
Prefab o nazwie ‘Player1’ jest postacią grywalną (hierarchia znajduje się na rysunku 5.56). W obiekcie zostały zawarte obiekty ciała – Body – oraz kamery – CameraHandler zawierający kamerę ‘Main Camera’. Obiekt ‘Body’ składa się z szkieletu i modelu, które zostały wcześniej zrobione w programie Blender. Obiekt kamery ‘Main Camera’ jest potomkiem obiektu ‘CameraHandler’. To rozwiązanie pozwala w łatwy sposób ustawić kamerę względem postaci.



Rysunek 5.56: Hierarchia obiektów w prefab’ie gracza

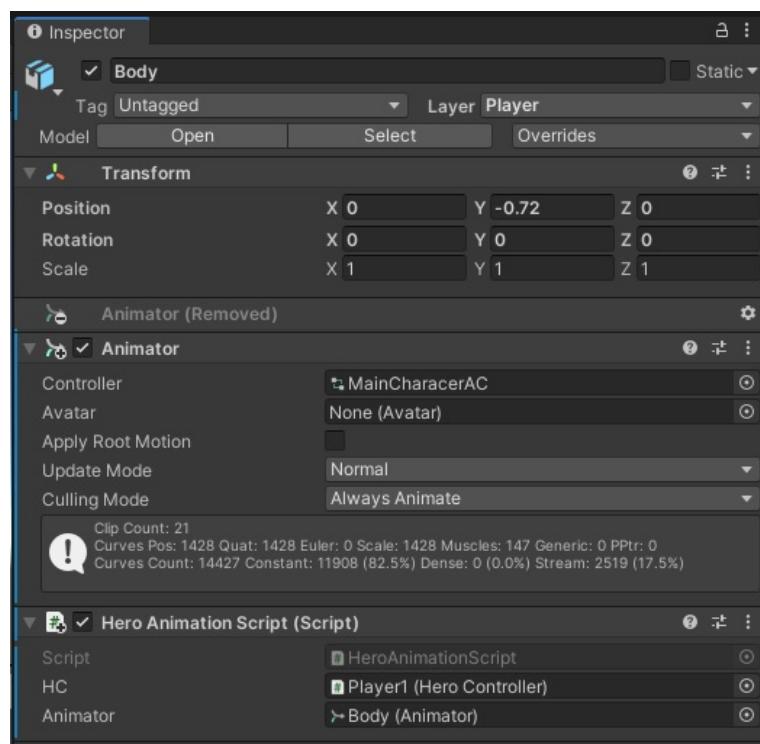
W komponentach gracza na rysunku 5.57 jest złożony z kilku komponentów. Pierwszym –poza wbudowanym, w każdy obiekt sceny ‘Transform’- komponentem jest ‘CapsuleCollider’, który odpowiada za kolizje z innymi obiektami zawierającymi component typu Collider. Wybór komponentu kolidującego jest decyzją zależną m.in. do czego obiekt będzie służył i jak będzie się poruszał. ‘CapsuleCollider’ jest bardzo dobrze przystosowany do poruszania się postaci. Następnym komponentem jest ‘RigidBody’, który odpowiada za symulowanie fizyki. Bez niego obiekt nie będzie podlegał grawitacji, siłom tarcia – jeżeli odpowiednio obiekty zostały przygotowane – i innym efektom fizycznym. W komponencie zostały zablokowane rotacje wokół osi, a detekcja kolizji

została ustawiona na ciągłą. Następnym komponentem jest opisany wcześniej (rysunek 5.55) komponent wejść. Ostatnimi komponentami są skrypty, jeden jest głównym skryptem gracza, a drugi jest skryptem obsługującym kolizje. Do skryptu ‘HeroController’ wprowadzana jest maska z jaką kolidują pewne elementy skryptu oraz referencja do skryptu obsługującego animacje postaci.



Rysunek 5.57: Komponenty głównego obiektu gracza

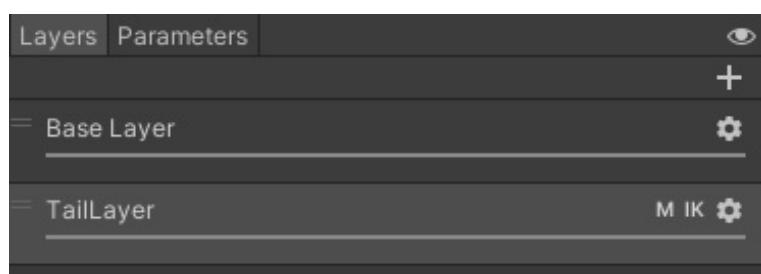
Na rysunku 5.58 zostały przedstawione 2 komponenty dołączone do obiektu ‘Body’. Pierwszym komponentem jest ‘Animator’, który odpowiada za przełączanie animacji. Przełączanie animacji odbywa się poprzez zmianę odpowiednich wartości przez skrypt obiektu – nie zawsze animacje są przełączane przez ten skrypt. Skrypt ‘Hero Animation Script’ zawiera zestaw metod uruchamianych w odpowiednich momentach animacji lub innych skryptów.



Rysunek 5.58: Komponenty obiektu ‘Body’ w prefab’ie gracza

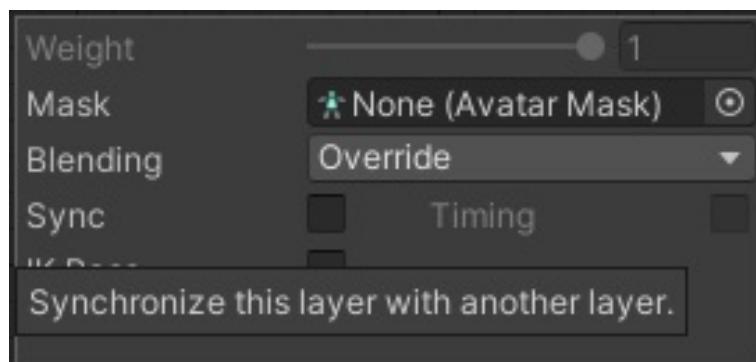
### 5.3. Animator postaci głównej

Animator jest komponentem zawierającym animacje oraz przejścia między animacjami. Nie ma standardowych animatorów, dlatego należy takowy stworzyć – w edytorze Unity prawy przycisk myszy na pole, gdzie znajdują się pliki -> ‘Create’ -> ‘Animator Controller’. Animator składa się z warstw (na rysunku 5.59 znajduje się warstwa podstawowa i dodatkowa warstwa) – domyślnie będzie ‘Base Layer’, pozwalają na włączanie kilku animacji w tym samym czasie dla różnych części ciała – parametrów (rysunek 5.61) – wartości, których zmiana jest sprawdzana w trakcie zmiany animacji – oraz stanów 5.63 — nazywanych też węzłami, reprezentacją animacji.



Rysunek 5.59: Warstwy w animatorze postaci głównej

W silniku Unity warstwy pozwalają na włączenie kilku różnych animacji kości w tym samym czasie. Na rysunku 5.60 pokazano ustawienia warstwy ‘Base Layer’.

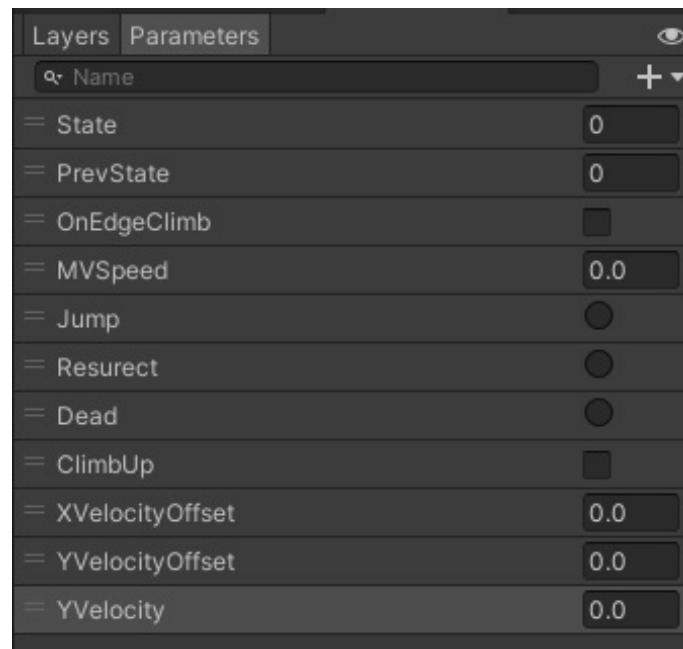


Rysunek 5.60: ustawienia warstwy ‘Base Layer’

Parametry mogą przyjąć 4 typy wartości: Float, Int, Bool i Trigger. Parametry zmieniane są w skryptach. Float, Int i Bool są typami występującymi w językach programowania i działają tak samo. Trigger swoim działaniem przypomina typ Bool. Jedyną różnicą jest to, jak zmieniana jest wartość. Trigger raz przełączony na ‘True’

w kodzie, może być przełączony na ‘False’ dopiero w przejściu, które wykorzystuje dany Trigger. Warto wspomnieć, że zmienne typu trigger powinny być używane tylko w przypadku, gdy dana animacja ma być włączona w wyniku wcisnięcia klawisza lub podczas akcji, która będzie rzadko występowała np. animacja śmierci postaci. Zmienna typu trigger zmieniana podczas kolizji może powodować włączanie tej samej animacji kilka razy pod rząd.

Na rysunku 5.61 są parametry używane w animatorze postaci sterowanej przez gracza. Parametr ‘State’ jest typu Int i określa stan w jakim postać się znajduje. Wartość 0 oznacza, że graczowi skończyło się zdrowie. Stany 1,2,3 i 4 oznaczają poruszanie się postaci. 1 i 2 to odpowiednio chód i bieg. Stan 3 oznacza, że postać spada lub skacze, a stan 4 oznacza, że postać wspina się po ścianie. PrevState określa jaki był stan postaci w poprzedniej klatce. ‘OnEdgeClimb’ jest typu Bool i oznacza, że została włączona sekwencja podciągnięcia się na krawędzi. MVSpeed jest długością wektora prędkości postaci w osiach X i Z. Włączenie Trigger'a ‘Jump’ wymusi na postaci skok. Triggery ‘Dead’ i ‘Resurect’ są przełączane odpowiednio podczas śmierci postaci oraz podczas odrodzenia się postaci w punkcie kontrolnym. ‘Resurect’ jest tylko do zresetowania animacji – włączenie domyślnej animacji po odrodzeniu. Parametry XVelocityOffset, YVelocityOffset i YVelocity są używane do sterowania animacjami ogona postaci.



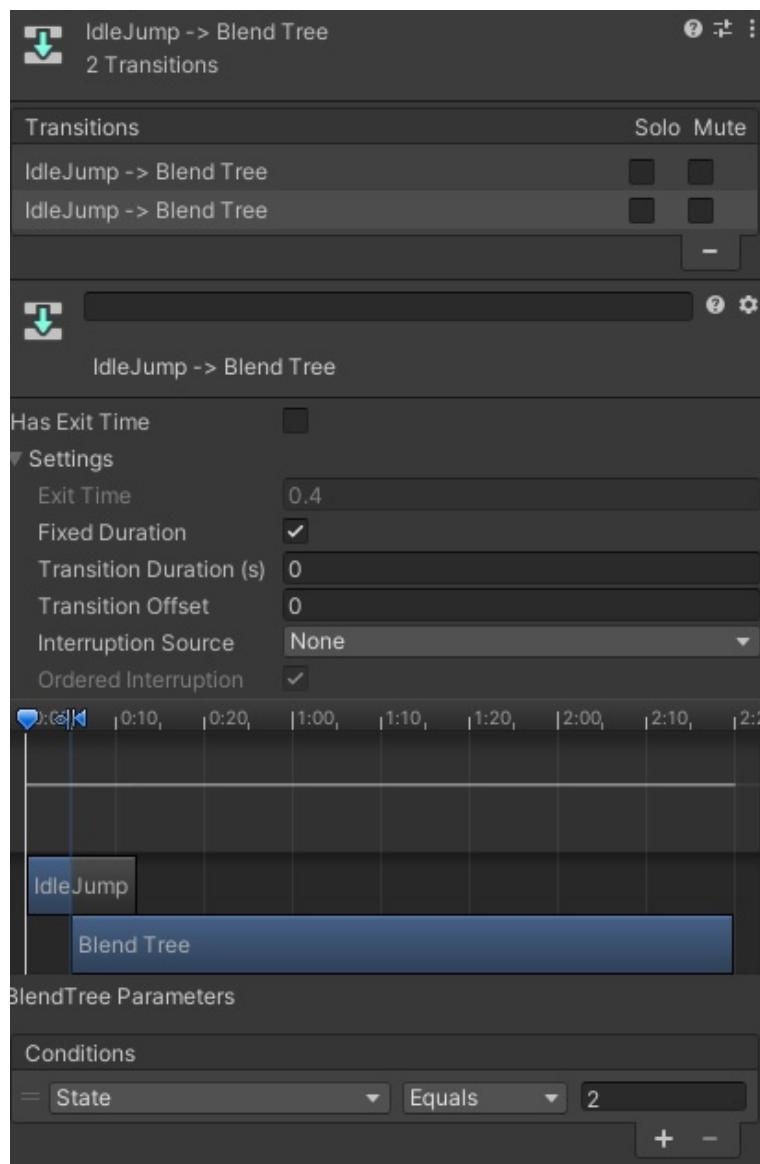
Rysunek 5.61: Parametry w animatorze postaci głównej

W każdej warstwie mogą się znajdować ‘Sub-StateMachine’, które mogą zawierać w sobie wiele węzłów. Są to pojemniki, do których można włożyć wiele animacji, które mają ze sobą cechy wspólne np. animacja biegu i czekania lub animacja spadania i skoku w powietrzu. Przejścia między węzłami mogą zachodzić, gdy zostanie spełniony warunek lub podczas zakończenia animacji, która nie jest zapętlona. Domyslnie w każdej warstwie i ‘Sub-StateMachine’ znajdują się 3 węzły podstawowe:

- Entry służy do oznaczenia animacji startowej. Pomarańczowa linia mówi, która animacja będzie włączana jako pierwsza podczas wejścia w daną warstwę lub pojemnik. Można to zmienić klikając prawym przyciskiem myszy na daną animację i ‘Set as Layer Default State’.
- Exit po wejściu w ten stan następuje wyjście z danej ‘Sub-StateMachine’. Jeżeli następuje w warstwie to przechodzi do węzła Entry.
- Any State – każde przejście od tego węzła będzie sprawdzane co klatkę. Jeżeli warunek zostanie spełniony to zostanie włączona animacja, do której prowadzi przejście niezależnie od tego jaka była grana poprzednia animacja.

Dodatkowo można stworzyć specjalny węzeł, który pozwala na częściowe włączenie kilku animacji – Blentree. Bardzo użyteczny węzeł podczas animacji, które są zależne od np. prędkości postaci.

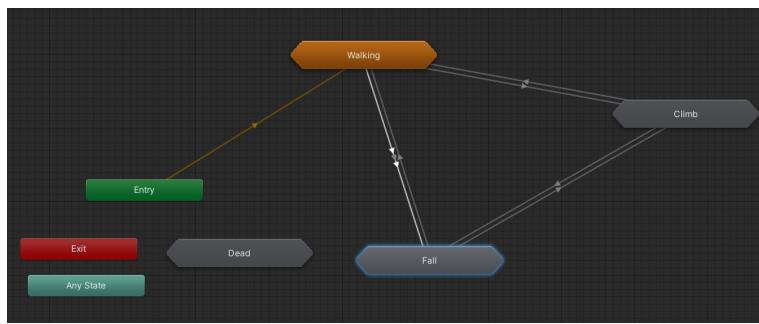
Na rysunku 5.62 przedstawiono przejście między węzłami zawierającymi animacje ‘IdleJump’ i ‘BlendTree’. Najważniejszym punktem konfiguracji przejścia jest pole ‘Has Exit Time’. Jeżeli pole jest zaznaczone przejście między animacjami może nastąpić po zakończeniu animacji oraz może trwać jakiś czas. To pole musi być odznaczone podczas animacji, które muszą być zapętlone, w innym wypadku animacja będzie przechodzić w wyznaczonym czasie animacji – wartość w polu ‘Exit Time’, która jest znormalizowaną wartością. Gdy ‘Has Exit Time’ jest odznaczone to przejście musi posiadać warunek przejścia. Pole ‘Transition Duration’ określa jak długo będzie przejście następować. Miejsce, od którego będzie grana nowa animacja jest określana w polu ‘Transition Offset’.



Rysunek 5.62: Przykładowe przejście między węzłami ‘IdleJump’ i ‘BlendTree’ w pojemniku ‘Walking’

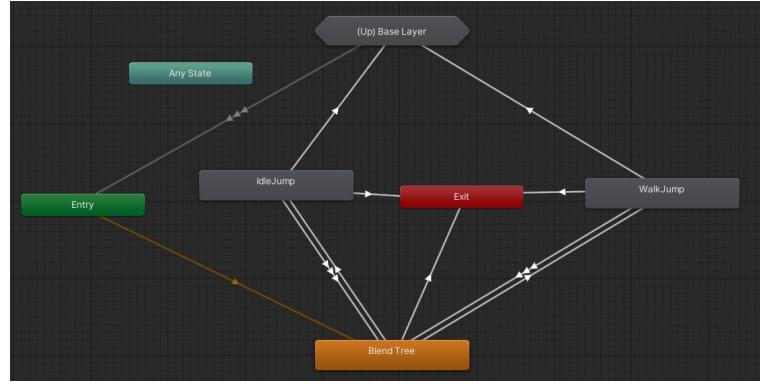
### 5.3.1. ‘Base Layer’ animacje ciała

Na rysunku 5.63 przedstawiono graf przejść między poszczególnymi ‘Sub-StateMachine’. Pojedyncze szare strzałki oznaczają, że przejścia następują między pojemnikami – tak zostały oznaczone przez edytor Unity. Te przejścia zachodzą podczas zmiany parametru ‘State’. Białe strzałki oznaczają przejścia między animacjami w ‘Sub-StateMachine’. Trzy strzałki oznaczają, że występuje kilka przejść między stanami lub pojemnikami. Na rysunku 5.63 są to szczególne przypadki, które często spowodowane były dziwnym zachowaniem animacji lub zapętleniem animacji. Startowym ‘Sub-StateMachine’ jest stan poruszania się po ziemi.

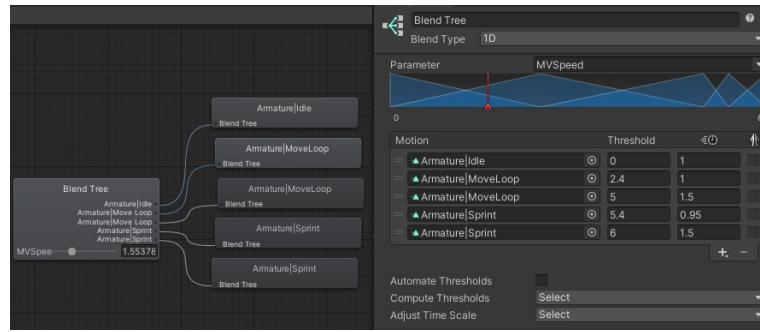


Rysunek 5.63: Węzły i przejścia w warstwie ‘Base Layer’

W pojemniku ‘Walking’ (rysunek 5.64) znajdują się 3 stany animacji – skok w ruchu, skok w spoczynku i ‘Blend Tree’. Animacja stworzona w ‘Blend Tree’ na rysunku 5.65 jest zależna od parametru MVSpeed. Wyjściowa animacja jest zależna od progów ‘Treshold’ oraz wartości danego parametru. Wartość ‘Treshold’ została wyznaczona metodą prób i błędów. Warunkiem przejścia z węzła ‘Blend Tree’ do węzła ‘IdleJump’ jest prędkość mniejsza niż 0.1 i trigger ‘Jump’ ustawiony na 1. Podobnie jest w przypadku przejścia do WalkJump, tylko że w tym przypadku prędkość ruchu postaci musi być większa lub równa 0.1. Warunkiem przejścia z tych węzłów do wyjścia z kontenera ‘(Up) BaseLayer’ prowadzi do węzła ‘NextJump’, co odpowiada wykonaniu animacji skoku w powietrzu. Przejście z węzła ‘BlendTree’ do węzła ‘Exit’ jest spowodowane zmianą wartości parametru ‘State’ na 0 lub liczbę większą niż 2.



Rysunek 5.64: Węzły i przejścia w pojemniku ‘Walking’



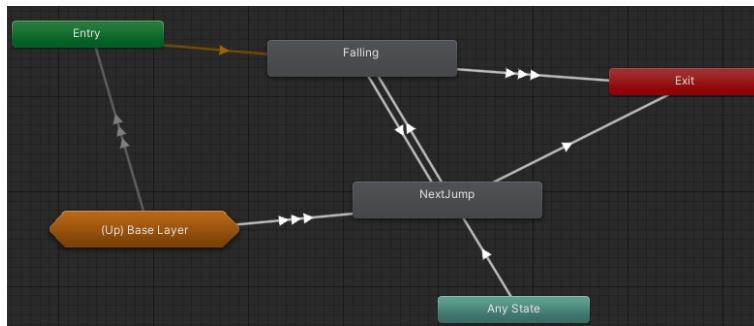
Rysunek 5.65: Stan ‘Blend Tree’ w pojemniku ‘Walking’

Zawartość kontenera ‘Fall’ znajduje się na rysunku 5.66. Znajdują się tutaj 2 węzły: ‘Falling’ – odpowiedzialny za animacje spadania – oraz ‘NextJump’ – skok w powietrzu. Domyślnym węzłem tego kontenera jest ‘Falling’. Innym przejściem prowadzącym do tego węzła jest przejście z ‘NextJump’ pod warunkiem, że parametr ‘State’ jest równy 3. Do węzła ‘NextJump’ prowadzą 4 przejścia:

- Z węzła ‘AnyState’ pod warunkiem, że ‘State’ jest równe 3, co oznacza, że postać spada, poprzedni stan był równy 4 oraz został aktywowany trigger ‘Jump’. To przejście było wymagane w celu wyeliminowania błędu animacji po skoku z ściany.
- 2 wcześniej opisane przejścia z animacji ‘IdleJump’ i ‘WalkJump’
- Z węzła ‘Falling’ pod warunkiem aktywacji triggera ‘Jump’

Przejścia do węzła ‘Exit’ są 3. Dwa z nich wychodzą z węzła ‘Falling’ – jedno pod warunkiem, że obecny stan nie jest równy 3 i drugie jest pozostałością z testów. Jedno

wychodzi z węzła ‘NextJump’ z warunkiem, że ‘State’ nie jest równy 3.

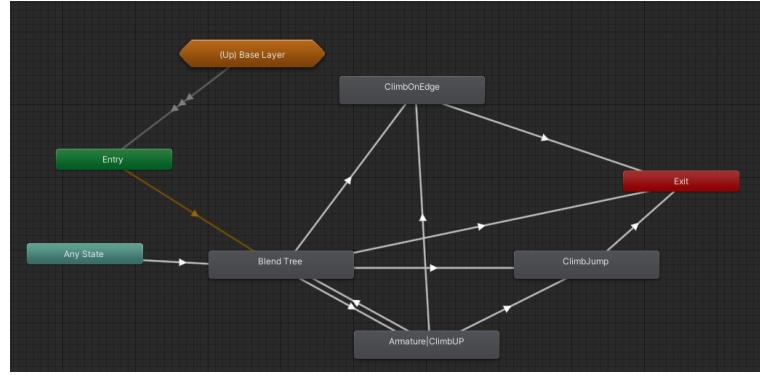


Rysunek 5.66: Węzły i przejścia w pojemniku ‘Falling’

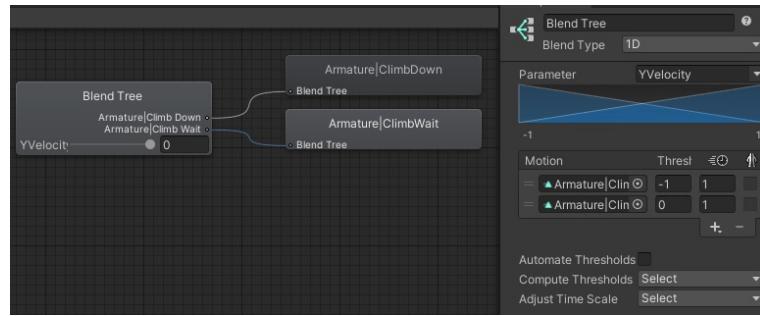
Na rysunku 5.67 przedstawiono kontener z animacjami dotyczącymi wspinania się postaci. Węzłem domyślnym jest ‘Blend Tree’, a jego zawartość jest pokazana na rysunku 5.68. Do tego węzła prowadzą przejścia z węzłów ‘Any State’ – pod warunkiem, że aktualny stan ma wartość 4, a poprzedni wartość różną od 4 – oraz ‘Armature|ClimbUP’ – pod warunkiem, że wartość trigger'a ‘Jump’ była ustawiona na True. Do węzła ‘ClimbOnEdge’ prowadzą 2 przejścia z węzłów ‘BlendTree’ i ‘Armature|ClimbUP’ oraz posiadają ten sam warunek przejścia – parametr ‘OnEdgeClimb’ ustawiony na true. Do węzła ‘ClimbJump’ prowadzą przejścia z węzłów ‘BlendTree’ i ‘Armature|ClimbUP’. Warunkiem przejścia jest ustawienie trigger'a ‘Jump’ na true.

Węzeł ‘Armature|ClimbUP’ został oddzielony z węzła ‘BlendTree’. Powodem tego rozwiązania był problem związany z prędkością postaci podczas wspinaczki w górę – podczas wchodzenia w górę w animacji można zauważać 2 cykle: podciągnięcie się postaci oraz przestawienie kończyn. Podczas podciągania się prędkość postaci w osi Y powinna wynosić pewną dodatnią wartość, a podczas przestawiania kończyn ta prędkość będzie wynosić około 0. Przy użyciu węzła typu ‘Blend Tree’ nie można ustawić dwóch animacji z wartością 0. Przejście z węzła ‘BlendTree’ do węzła ‘Armature|ClimbUP’ zachodzi tylko w wypadku zmiany flagi ‘ClimbUP’.

Wejście do węzła ‘Exit’ następuje, gdy: wykonywania animacji z węzła ‘ClimbOnEdge’ lub ‘BlendTree’ został zmieniony parametr ‘State’, skończyła się animacja ‘ClimbJump’.

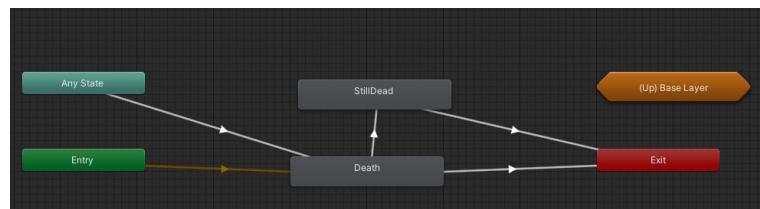


Rysunek 5.67: Węzły i przejścia w pojemniku ‘Climbing’



Rysunek 5.68: Stan ‘Blend Tree’ w pojemniku ‘Climbing’

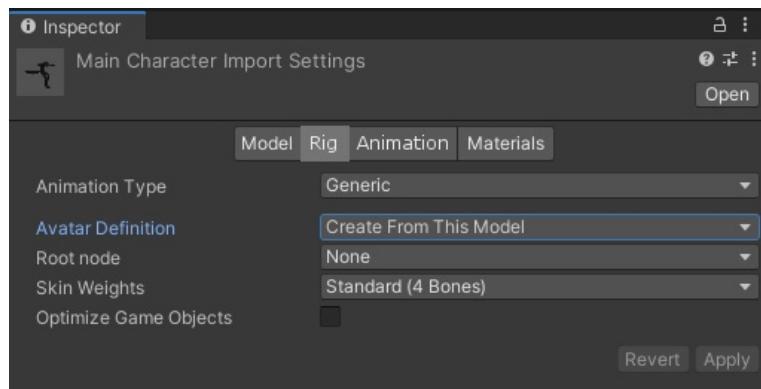
Na rysunku 5.69 została przedstawiona zawartość kontenera ‘Dead’. Jedyną możliwą metodą wejścia do tego pojemnika jest przestawienie parametru ‘State’ na 0, co następuje podczas spadku zdrowia postaci do 0. Ważne jest wykorzystanie tutaj węzła ‘Any State’, co pozwoliło na wymuszenie tej zmiany nieważne od aktualnej animacji. Wejście do węzła ‘Exit’ następuje tylko podczas przestawienia trigger'a ‘Resurect’ na True. Przejście z węzła ‘Death’ do ‘StillDead’ następuje po zakończeniu animacji ‘Death’.



Rysunek 5.69: Węzły i przejścia w pojemniku ‘Dead’

### 5.3.2. ‘Tail Layer’ – steroawnie ogonem

Warstwy w animatorze nie zostały nazwane warstwami bez przyczyny. Miało to na celu skojarzenie z warstwami w takich programach jak Photoshop czy Gimp. Działają one w podobny sposób. Każda warstwa może posiadać maskę, która określa na jakie kości dana maska wpływa – domyślnie warstwa wpływa na każdą kość w szkieletie. Są możliwe dwa rodzaje wpływu: ‘Additive’, w którym transformacje przemnożone przez wagę są dodawane oraz ‘Override’, gdzie transformacje są zastępowane. Jednak szkielety postaci zwykle się różnią między sobą, dlatego Unity pozwala na stworzenie Avatar'a – maski dla szkieletu. Na rysunku 5.70 przedstawiono zimportowanie szkieletu, dzięki czemu będzie go można użyć jako maskę – wybranie takich opcji pozwoliło na stworzenie Avatar'u dla postaci głównej w tym projekcie.

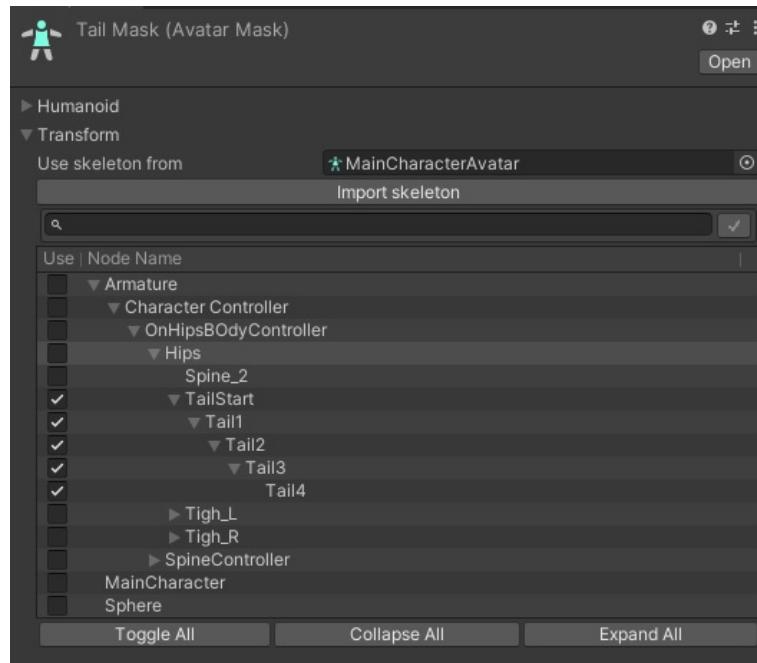


Rysunek 5.70: Ustawienie szkieletu jako dostępnego ‘Avatar'a’

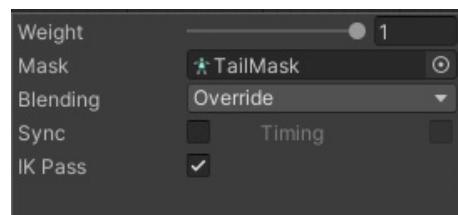
Dzięki tej operacji szkielet będzie widoczny w inspektorze, po wybraniu Avatar'a z plików – pusty Avatar można stworzyć klikając prawym przyciskiem myszy w przeglądarce plików edytora unity -> ‘Create’ -> ‘Avatar Mask’. Na rysunku 5.71 została pokazana skonfigurowana maska dla asset'u gracza. Zostały wybrane kości, które odpowiadają ogonowi postaci.

Rysunek 5.72 przedstawia ustawienia warstwy ‘TailLayer’ animatora. W polu ‘Mask’ znajduje się Avatar, który został pokazany na rysunku 5.71. ‘Blending’ został ustawiony na nadpisywanie transformacji kości.

Rysunek 5.73 zostały pokazane węzły i przejścia. Węzeł ‘New State’ nie zawiera animacji, ponieważ temu żadna animacja nie jest grana i kontrola jest oddawana warstwie ‘Base Layer’. Wejście do tego węzła następuje, gdy trigger ‘Dead’ jest aktywowany na wartość True. Przejście do węzła ‘Blend Tree’ następuje po zmianie wartości

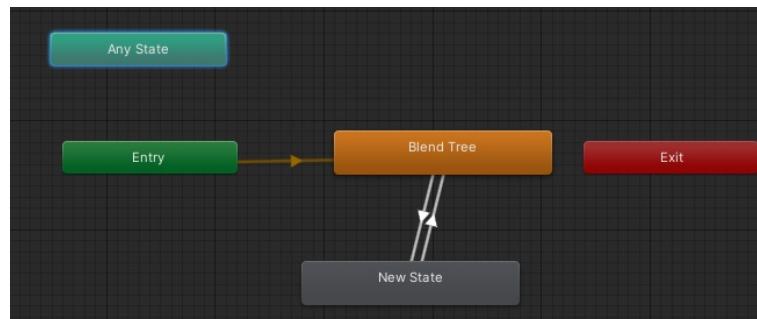


Rysunek 5.71: Ustawienie maski dla warstwy ‘TailLayer’



Rysunek 5.72: Ustawienia warstwy ‘TailLayer’

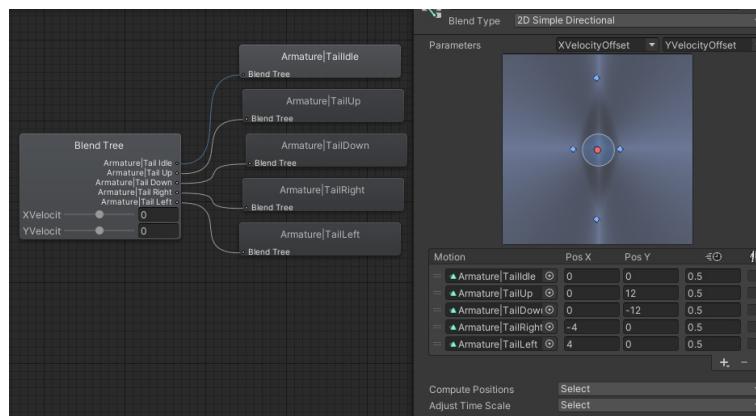
parametru ‘State’ na wartość większą niż 0.



Rysunek 5.73: Węzły i przejścia w warstwie ‘TailLayer’

Na rysunku 5.74 widać wnętrze węzła ‘Blend Tree’. Jest to drzewo dwuwymiarowe zależne od parametrów XvelocityOffset i YVelocityOffset. W tym drzewie ‘mie-

szane' sa animacje ogona.



Rysunek 5.74: Węzeł 'Blend Tree' w warstwie 'TailLayer'

## 5.4. Skrypty postaci głównej

Skryp głównej postaci został rozdzielony na kilka klas. Klasą główną sterującą wszystkimi klasami jest klasa ‘HeroController’. W tej klasie znajdują się obiekty do sterowania statystykami ‘HeroStatManager’ i obiekty odpowiadające trybom poruszania postacią – np. wspinaczkę po ścianach.

‘HeroStatManager’ jest klasą sterującą statystykami i stanami postaci (listing 2). Obliczenia przeprowadzane w obiekcie tej klasy są bardzo podstawowe, dlatego nie dziedziczy ona z klasy MonoBehaviour.

```
1  public class HeroStatManager
2  {
3      public enum playerMoveStateTAB
4      {
5          Dead,
6          Walk,
7          Sprint,
8          Fall,
9          Climb
10     };
11     //obecny stan postaci
12     private int playerMoveState = 0;
13     //obecne poziomy HP, SP
14     public float Health_Points, Stamina_Points;
15     public float jumpStrength, acceleration, AirControll,
staminaDCD;
16
17     static float staminaDepletionCD = 1.0f;
18     public int jumpsCounter;
19     //czas w jakim postac nie moze otrzymac ponownie obrazem
20     public float invincibilityTime;
21     //lista zawierajaca predkosci ruchu postaci
22     public List<float> mvSpeeds;
23     //obiekty zawierajace podstawowe statystyki i
statysstyki zebrane na mapie
24     [HideInInspector] private Stats Base_stats, UPD_Stats;
25     //flagi opisujace czy postac stoi na ziemi, czy biegnie,
czy sie wspina i czy zyje
26     public bool isGrounded = false, isSprinting = false,
isClimbing = false, isdead = false;
27     // Start is called before the first frame update
28     public HeroStatManager()
29     {
30         Base_stats = new Stats();
31         UPD_Stats = new Stats();
32         //inicjalizacja statystyk
33         InitBaseStats();
34         //poziom kontroli w powietrzu
35         AirControll = 0.75f;
36         //poczatkowe wartosci punkow zdrowia i punkow
wytrzymalosci
37         Health_Points = 2.0f;
```

```

38         Stamina_Points = 2.0f;
39         //zdefiniowanie predkosci ruchow w poszczegolnych
stanach
40         mvSpeeds = new List<float>();
41         mvSpeeds.Add(0.0f);
42         mvSpeeds.Add(4.0f);
43         mvSpeeds.Add(6.0f);
44         mvSpeeds.Add(4.0f);
45         mvSpeeds.Add(4.0f);
46         acceleration = 16.0f;
47         // ustawienie licznika skokow
48         jumpsCounter = (int)Base_stats.stats_values[(int)
Stats.StatisticsCODE.maxJump]+(int)UPD_Stats.stats_values[(int)
Stats.StatisticsCODE.maxJump];
49         //ustawienie sily skokow
50         jumpStrength = mvSpeeds[1]*1.5f;
51         staminaDCD = staminaDepletionCD;
52         invincibilityTime = 0.3f;
53     }
54
55     //metoda zwraca czy postac zyje
56     public bool Update(bool isMoving)
57     {
58         //sprawdzanie czy postac zyje
59         if(!isdead){
60             StaminaManage(isMoving);
61             //zmiana stanu postaci
62             changeState(isMoving);
63             return HealthManage();
64         }
65         return false;
66     }
67
68     //metoda odpowiedzialana za zarzadznie wytrzymaloscia
postaci
69     void StaminaManage (bool isMoving)
70     {
71         //ustawienie regeneracji wytrzymalosci -- jest to
kombinacja statystyk stalych i zebranych
72         float StaminaReg = Base_stats.stats_values[(int)
Stats.StatisticsCODE.SP_regen]+UPD_Stats.stats_values[(int)
Stats.StatisticsCODE.SP_regen];
73         //jeżeli postac biegnie, wspina sie lub jest w
powietrzu to wytrzymalosc sie nie regeneruje
74         if (isMoving && ((playerMoveState==(int)
playerMoveStateTAB.Sprint && isGrounded) || playerMoveState
==(int)playerMoveStateTAB.Climb ))
75         {
76             Stamina_Points -= (StaminaReg)*0.2f * Time.
deltaTime;
77
78         }
79         else
80             Stamina_Points += System.Convert.ToSingle(
isGrounded) * StaminaReg * Time.deltaTime;

```

```

82         //jezeli wytrzymalosc spradnie do 0 to postac
83         przestaje biec lub trzymac sie sciany
84         if (Stamina_Points <= 0.0f)
85         {
86             Stamina_Points = 0;
87             isSprinting = false;
88             isClimbing = false;
89             staminaDCD = staminaDepletionCD;
90
91         }
92         // zaokraglenie wytrzymalosci, zeby nie byla ponad
93         maksymalna wartosc
94         if (Stamina_Points > Base_stats.stats_values[(int)
95         Stats.StatisticsCODE.Max_SP] + UPD_Stats.stats_values[(int)
96         Stats.StatisticsCODE.Max_SP])
97         {
98             Stamina_Points = Base_stats.stats_values[(int)
99             Stats.StatisticsCODE.Max_SP]+ UPD_Stats.stats_values[(int)
100            Stats.StatisticsCODE.Max_SP];
101        }
102        //flaga zeby postac nie mogla caly czas zmieniac
103        stanow np. biegu
104        if(staminaDCD<=0.0f)
105        {
106            staminaDCD = 0.0f;
107        }
108        else
109            staminaDCD -=Time.deltaTime;
110    }
111    //metoda do zarzadzania zdrowiem, jezeli zdrowie spadnie
112    do 0 postac umiera
113    private bool HealthManage()
114    {
115
116        if(Health_Points<0.0f)
117        {
118            isdead = true;
119            playerMoveState = (int)playerMoveStateTAB.Dead;
120            changeState(false);
121            return true;
122        }
123        //regeneracja zdrowia
124        Health_Points += (Base_stats.stats_values[(int)Stats.
125        .StatisticsCODE.HP_regen]+UPD_Stats.stats_values[(int)Stats.
126        StatisticsCODE.HP_regen]) * Time.deltaTime;
127        if (Health_Points > Base_stats.stats_values[(int)
128        Stats.StatisticsCODE.Max_HP] + UPD_Stats.stats_values[(int)
129        Stats.StatisticsCODE.Max_HP])
130        {
131            Health_Points = Base_stats.stats_values[(int)
132            Stats.StatisticsCODE.Max_HP]+ UPD_Stats.stats_values[(int)
133            Stats.StatisticsCODE.Max_HP];
134        }
135        return false;
136    }
137    public void Heal(float heal)

```

```

124    {
125        Health_Points += heal;
126    }
127    public void StaminaHeal(float sHeal)
128    {
129        Stamina_Points +=sHeal;
130    }
131    //metoda zadania obrazem
132    public void applyDamage(float damage){
133        Health_Points -= damage;
134    }
135    //metoda zarzadzania stanami, do playerMoveState
136    zostaje zapisana liczba odpowiadajaca odpowiedniemu stanowi
137    poruszania
138    public void changeState(bool isMoving)
139    {
140        //if blokujacy przechodzenie dalej przez warunki
141        if(playerMoveState==(int)playerMoveStateTAB.Dead)
142        {
143        }
144        else if(isClimbing)
145        {
146            playerMoveState = (int)playerMoveStateTAB.Climb;
147        }
148        else if(!isGrounded)
149        {
150            playerMoveState = (int)playerMoveStateTAB.Fall;
151        }
152        else if(isSprinting && !isClimbing && staminaDCD
153        <=0.0f)
154        {
155            playerMoveState = (int)playerMoveStateTAB.Sprint;
156        }
157        else if( isMoving && (!isSprinting || staminaDCD>0.0
158        f))
159        {
160            isSprinting = false;
161            playerMoveState = (int)playerMoveStateTAB.Walk;
162        }
163    }
164    //odrodzenie postaci
165    public void resurrect()
166    {
167        playerMoveState = (int)playerMoveStateTAB.Walk;
168        isClimbing = false;
169        isSprinting = false;
170        isdead = false;
171        isGrounded =false;
172        Health_Points = Base_stats.stats_values[(int)Stats.
173        StatisticsCODE.Max_HP]+UPD_Stats.stats_values[(int)Stats.
174        StatisticsCODE.Max_HP];
175    }
176    //metoda do ulepszenia statystyk

```

```

173     public void upgradeStats(ref Stats Upgrade){
174         for(int i=0; i<sizeof(Stats.StatisticsCODE)+1; i++)
175         {
176             UPD_Stats.stats_values[i] += Upgrade.
177             stats_values[i];
178         }
179     }
180     public int getState()
181     {
182         return playerMoveState;
183     }
184     public float getMvSpeed()
185     {
186         return mvSpeeds[playerMoveState];
187     }
188     public float getInvincibilityTime()
189     {
190         return invincibilityTime;
191     }
192     public int getMaxJumps(){
193         return (int)Base_stats.stats_values[(int)Stats.
194         StatisticsCODE.maxJump]+(int)UPD_Stats.stats_values[(int)
195         Stats.StatisticsCODE.maxJump];
196     }
197     public float getMaxHP(){
198         return Base_stats.stats_values[(int)Stats.
199         StatisticsCODE.Max_HP]+UPD_Stats.stats_values[(int)Stats.
200         StatisticsCODE.Max_HP];
201     }
202     public float getMaxSP(){
203         return Base_stats.stats_values[(int)Stats.
204         StatisticsCODE.Max_SP]+UPD_Stats.stats_values[(int)Stats.
205         StatisticsCODE.Max_SP];
206     }
207     //metoda ustawienia podstawowych statystyk
208     private void InitBaseStats(){
209         Base_stats.stats_values[(int)Stats.StatisticsCODE.
210         Max_HP] = 20.0f;
211         Base_stats.stats_values[(int)Stats.StatisticsCODE.
212         Max_SP] = 20.0f;
213         Base_stats.stats_values[(int)Stats.StatisticsCODE.
214         HP_regen] = 1.0f;
215         Base_stats.stats_values[(int)Stats.StatisticsCODE.
216         SP_regen] = 5.0f;
217         Base_stats.stats_values[(int)Stats.StatisticsCODE.
218         maxJump] = 2.0f;
219     }
220 }
```

Listing 1: Skyrpt HeroStatManager

Klasa Stats jest bardzo prosta klasą, która zawiera pole typu enum - StatisticsCODE – i pole listy, której indeksy są powiązane z polem StatisticsCODE. W klasie

jest konstruktor przyjmujący listę, która definiuje pola listy. [SerializeField] jest dla kompilatora Unity i oznacza że dane pole ma być widoczne w inspektorze silnika. [System.Serializable] jest wymagane, gdy dana obiekt stworzony na podstawie klasy, jest polem innej klasy i ma być widoczny w inspektorze.

```
1 [System.Serializable]
2 public class Stats
3 {
4     public enum StatisticsCODE
5     {
6         Max_HP,
7         HP_regen,
8         Max_SP,
9         SP_regen,
10        maxJump,
11    };
12    [SerializeField] public float[] stats_values;
13    public Stats(){
14        //sizeof enum return biggest number in enum not actual
15        //size, highest number of enum is 1 less than size
16        stats_values = new float[sizeof(StatisticsCODE)+1];
17        for(int i =0; i<sizeof(StatisticsCODE); i++)
18        {
19            stats_values[i] = 0;
20        }
21        //metoda przyjmuje kod z StatisticsCODE i wartosc
22        //zwiększenia danej statystyki
23        void increaseStatsByIndex(int StatCode, int Value){
24            stats_values[StatCode] += Value;
25        }
26    }
}
```

Listing 2: Skyrpt HeroStatManager

Klasa ‘HeroController’(listing 3) jest połączona bezpośrednio do obiektu postaci. W metodzie ‘Start()’ są inicjowane wartości początkowe i inicjowane inne obiekty. W linii 51 pobrana jest referencja do transformacji obiektu, w którym znajduje się model. Pole o nazwie ‘moveSet’ jest listą zawierającą zestaw ruchów dla określonego stanu postaci. Klasa ‘WalkMoveSet’ jest klasą, która odpowiada za poruszanie się postacią podczas chodzenia, biegania i spadania – została opisana 4. Referencja obiektu utworzona z tej klasy została zapisana w 3 polach listy. Klasa ‘ClimbMoveset’ jest zestawem ruchów podczas wspinania postaci. Klasa ‘ClimbMoveset’ dziedziczy z klasy ‘WalkMoveSet’. Do pola ‘animator’ zapisany zostaje komponent ‘Animator’, który steruje animacjami. ‘playerStatus’ dostaje referencje do skryptu sterującego GUI – paski pokazujące stan zdrowia i wytrzymałości. Metoda ‘Update()’ jest metodą wywoły-

waną przez silnik co klatkę. Metoda ‘HeroTerrainColidingHandler()’ odpowiadająca za detekcję czy postać stoi i czy postać może się dalej wspinać. Ta metoda musi być tutaj wywoływana ponieważ podczas wywoływania w metodzie ‘FIxedUpdate()’ powodowała bardzo dziwne zachowania animacji. Metoda ‘Update’ obiektu ‘HeroStat’, która odpowiedzialna jest za zmianę stanów postaci. Jeżeli stanem postaci będzie przegrana, zostanie przełączony Trigger w animatorze postaci i zostanie włączona animacja śmierci postaci. Na końcu tej metody są ustawiane poziomy zdrowia i wytrzymałości w interfejsie graficznym.

Metoda ‘repsawn’ przyjmuje referencje do punktu kontrolnego. Pozycja bohatera jest zmieniana na pozycję punktu kontrolnego, następnie w obiekcie kontrolera statystyk jest wywoływana metoda ‘resutrect’. Ustawienie trigger'a ‘Resurect’ odpowiada zresetowaniu kontrolera animacji – po śmierci jest on zablokowany aż do zmiany tego pola. Na koniec odblokowana jest możliwość poruszania się postaci. W metodzie Fixe-dUpdate wywoływanej przed pętlą fizyczną jest metoda ‘movementController’. Metoda ‘movementController’ odpowiada za przemieszczanie postaci – sterowanie odbywa się poprzez dodawanie odpowiedniej prędkości w danym kierunku – oraz rotację modelu w kierunku ruchu. Linia 110 sprawdza czy postać znajduje się w poprawnym stanie – stan o numerze -1 mówi o tym, że zdrowie się skończyło. Rozdzielenie pewnych obliczeń w tym miejscu było potrzebne – rotowanie postaci nie może odbywać się w obiektach dotyczących ruchu, ponieważ należałoby przekazać wiele referencji do tych obiektów, a w założeniu były one małe i nie posiadały mało pól. Linie od 120 do 124 odpowiadają za nakierowanie modelu w pewnym kierunku. Rotacja modelu nie jest natychmiastowa, dzięki czemu poruszanie postaci jest bardziej naturalne.

Metoda ‘HeroTerrianColidingHandler()’ ma 2 zasadnicze zadania do wykonania: sprawdzić, czy postać dotyka podłogi i wywołanie metody, która sprawdzi czy postać nadal spełnia warunki do ‘przyklejenia się’ do ściany. Linie 139-149 odpowiadają za sprawdzenie, czy postać dotyka podłożą oraz zmienienie odpowiednich flag i warości. Metoda Raycast klasy Physics ‘wystrzeliwuje’ promień i jeżeli promień dotknie obiektu typu ‘collider’ to metoda zwróci wartość ‘True’. Pierwszym argumentem metody jest położenie – w tym wypadku jest to położenie środka collider'a gracza. Drugim argumentem jest kierunek wystrzelenia promienia. Za trzeci argument metoda przyjmuje obiekt typu RaycastHit – zapisuje w nim jaki obiekt został uderzony, kąt uderzenia itp. Do metody trzeba też podać długość promienia, w tym przypadku długość promienia

wynosi ponad połowę wysokości collider'a – przy podaniu dokładnie połowy może dojść do tego, że warunek nigdy nie będzie spełniony. Ostatnim argumentem jest maska – lista warstw kolizji, z którą promień może kolidować. Dodatkowo sprawdzane jest czy nachylenie podłoża nie jest zbyt duże. Jeżeli warunki zostały spełnione – postać dotyka nie stromego podłoża – to zostaje zapisany modyfikator podłoża – zmniejszenie przyśpieszenia, dzięki czemu postać ma pewną trudność podczas chodzenia po bardziej stromych zboczach – zostaje zresetowany licznik skoków oraz zostaje ustawiona flaga ‘isGrounded’. W liniach 151-158 dochodzi do sprawdzenia czy postać wspina się, następnie sprawdzane jest czy postać nadal może się wspinać. Jeżeli nie może to następuje wywołana funkcja wyjścia z stanu wspinania.

Metoda ‘OnJump’ jest metodą wywoływaną przez kontroler wejścia. Argumentem metody jest obiekt CallbackContext, który zawiera informacje czy klawisz jest wcisnięty, czy puszczyony, wartość wcisniętego klawisza itp. W tej metodzie są wywoływane metody aktualnego zestawu ruchów, gdy klawisz odpowiedzialny za akcje zostanie wcisnięty. Metody ‘onMoveHorizontal’ i ‘onMoveVertical’ działają na tej samej zasadzie. Są wywoływane podczas wcisnięcia odpowiedniego klawisza – lub jego puszczenia. W metodach do pola ‘inputMoveVector’ są zapisywane wartości – są to wartości od -1 do 1. Metoda onSprint zmienia flagę ‘isSprinting’ na true, gdy klawisz sprintu jest wcisnięty i false, gdy ten klawisz jest puszczyony. Metoda ‘onWallSnap’ jest wywoływana podczas wywołania akcji ‘WallSnap’. Jej działanie jest zależne od aktualnego stanu postaci – jeżeli się wspina to wspinanie jest przerywane, w innym wypadku bada czy jest możliwość ‘przyklejenia’ do ściany.

Pozostałe metody klasy są typowymi metodami get i set.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.InputSystem;
5
6  /*
7   * This class is for managing movement, collisions and keyinput
8   *
9   * FixedUpdate is mainly used for movement and collision related
10  * calculations
11  */
12 public class HeroController : MonoBehaviour
13 {
14
15     //private
16     private RespawnController RC;
```

```

16     private Rigidbody rb;
17     private CapsuleCollider collide;
18     //maska, ktorą określa jakie obiekty mogą wchodzić w
19     kolizje z postacią
20     [SerializeField] private LayerMask hitableMasks;
21     //pole dla kontrolera wejścia
22     private PlayerInput input;
23     private RaycastHit hit;
24     private HeroStatManager HeroStat;
25     private List<WalkMoveSet> moveSet;
26
27     private Animator animator;
28     private PlayerStatus playerStatus;
29     private bool canMove;
30     private float groundMultiplayer = 0.0f;
31     private Transform BodyTransform;
32     [SerializeField] HeroAnimationScript PAS;
33     //public
34     private Vector3
35     /*
36      movingPlane -- określa os (x,y,z) w jakiej postaci może
37      się poruszać
38     */
39     movingPlane = new Vector3(1, 1, 0),
40     lookDirection = new Vector3(1, 0, 1),
41     lastInputVector = new Vector3(1,0,1),
42     //vector of combined inputs (ad, jump, ws)
43     inputMoveVector = new Vector3(0, 0, 0);
44
45     private int moveSetState;
46
47     // Start is called before the first frame update
48     void Start()
49     {
50         moveSetState = 0;
51         rb = GetComponent<Rigidbody>();
52         collide = GetComponent<CapsuleCollider>();
53         input = GetComponent<PlayerInput>();
54         HeroStat = new HeroStatManager();
55         BodyTransform = this.transform.Find("Body").
56         transform;
56         rb.velocity = Vector3.zero;
57         moveSet = new List<WalkMoveSet>();
58         moveSet.Add(new WalkMoveSet());
59         moveSet.Add(moveSet[0]);
60         moveSet.Add(moveSet[0]);
61         moveSet.Add(new ClimbMoveset(hitableMasks, collide.
62         radius, collide.height));
63
64         rb.mass = 50.0f;
65
66         canMove = true;
67         HeroStat.resurect();
68         //animations
69         animator = this.transform.Find("Body").GetComponent<
70         Animator>();

```

```

67         ((ClimbMoveset)moveSet[(int)HeroStatManager.
playerMoveStateTAB.Climb-1]).setAnimator(ref animator);
68         //gui
69         playerStatus = GameObject.Find("Gui").GetComponent<
PlayerStatus>();
70         playerStatus.Initiate(HeroStat.getMaxHP(), HeroStat.
getMaxSP());
71         GetComponent<HeroCollision>().setHeroStat(ref
HeroStat);
72
73     }
74
75
76     // Update is called once per frame -- better use for
graphical effects
77     void Update()
78     {
79         HeroTerrainColidingHandler();
80         moveSetState = HeroStat.getState() -- 1;
81
82         if(HeroStat.Update(rb.velocity.x * rb.velocity.x +
rb.velocity.y * rb.velocity.y > 0.1f))
83         {
84             animator.SetTrigger("Dead");
85         }
86         playerStatus.SetCurrentLevels(HeroStat.Health_Points
, HeroStat.Stamina_Points);
87
88     }
89     public void respawn(ref GameObject RespawnPoint){
90         this.transform.position = RespawnPoint.transform.
position;
91         HeroStat.resurect();
92         animator.SetTrigger("Resurect");
93         canMove = true;
94
95     }
96
97
98     //Fixed update is called once in constant amount of time
-- better use for phisic related things
99     private void FixedUpdate()
100    {
101        movementController();
102        //Debug.Log(lookDirection);
103
104    }
105
106
107     //Controller of player forces, , Velocity
108     private void movementController()
109    {
110        if ((int)HeroStat.getState() > 0)
111        {
112            if(moveSetState!= (int)HeroStatManager.
playerMoveStateTAB.Climb-1)

```

```

113         {
114             canMove = true;
115             if (inputMoveVector.sqrMagnitude > 0)
116             {
117                 lastInputVector = this.transform.
118                 rotation * Vector3.Scale(inputMoveVector, movingPlane);
119             }
120             Quaternion tmp = Quaternion.FromToRotation(
121                 lastInputVector, new Vector3(0.0f, 0.0f, -1.0f));
122             BodyTransform.rotation = Quaternion.
123             RotateTowards(BodyTransform.rotation, tmp, 10.0f);
124             lookDirection = BodyTransform.rotation*(new
125                 Vector3(0.0f, 0.0f, 1.0f));
126             lookDirection.Normalize();
127         }
128         moveSet[moveSetState].movementController(
129             groundMultiplayer, inputMoveVector, this.transform.rotation);
130     }
131 }
132
133
134
135     //method for detecting ground and other colision related
136     things
137     private void HeroTerrainColidingHandler()
138     {
139         //if collider colides with ground
140         if (Physics.Raycast(rb.position+collide.center,
141             Vector3.down, out hit, collide.height * 0.55f, hitableMasks)
142             && hit.normal.y > 0.60f)
143         {
144
145             groundMultiplayer = hit.normal.y;
146             HeroStat.jumpsCounter = HeroStat.getMaxJumps()
147             -- 1;
148             HeroStat.isGrounded = true;
149         }
150         else
151         {
152             HeroStat.isGrounded = false;
153         }
154         //if is climbing
155         if (moveSetState == (int)HeroStatManager.
156             playerMoveStateTAB.Climb-1)
157         {
158             if (((ClimbMoveset)moveSet[(int)HeroStatManager.
159                 .playerMoveStateTAB.Climb-1]).canClimb(lookDirection,
160                 hitableMasks))
161             {
162                 HeroStat.isClimbing = false;
163                 ((ClimbMoveset)moveSet[(int)HeroStatManager.

```

```

157     playerMoveStateTAB.Climb - 1]).ExitState();
158     }
159   }
160 
161   //methods called when certain key is pressed
162   public void OnJump(InputAction.CallbackContext context)
163   {
164     if (canMove)
165     {
166 
167       if (context.started)
168       {
169         if (moveSet[moveSetState].onJump(context))
170           PAS.aChangeJumpFlag();
171       }
172     }
173   }
174 
175 
176   public void onMoveHorizontal(InputAction.CallbackContext
177 context)
178   {
179 
180     inputMoveVector.x = context.ReadValue<float>();
181   }
182 
183 
184   public void onMoveVertical(InputAction.CallbackContext
185 context)
186   {
187 
188     inputMoveVector.z = context.ReadValue<float>();
189   }
190 
191   public void onSprint(InputAction.CallbackContext context
192 )
193   {
194     if (context.started)
195     {
196       HeroStat.isSprinting = true;
197     }
198     if (context.canceled)
199     {
200       HeroStat.isSprinting = false;
201     }
202   }
203 
204 
205   public void onWallSnap(InputAction.CallbackContext
206 context)
207   {
208     if (context.started)

```

```

208         {
209             if (moveSetState == (int)HeroStatManager.
playerMoveStateTAB.Climb -- 1)
210             {
211                 ((ClimbMoveset)moveSet[moveSetState]).ExitState();
212                 HeroStat.isClimbing = false;
213                 HeroStat.changeState(rb.velocity.x * rb.
velocity.x + rb.velocity.y * rb.velocity.y > 0.1f);
214             }
215             else if(((ClimbMoveset)moveSet[(int)
HeroStatManager.playerMoveStateTAB.Climb-1]).canClimb(
lookDirection, hitableMasks))
216             {
217                 HeroStat.jumpsCounter = HeroStat.getMaxJumps
();
218                 HeroStat.isClimbing = true;
219                 HeroStat.changeState(rb.velocity.x * rb.
velocity.x + rb.velocity.y * rb.velocity.y > 0.1f);
220             }
221         }
222     }
223 }
224
225 public HeroStatManager GetHeroStat()
226 {
227     return HeroStat;
228 }
229 public RaycastHit GetHit()
230 {
231     return hit;
232 }
233 public void upgradeStats(ref Stats Upgrade){
234     HeroStat.upgradeStats(ref Upgrade);
235 }
236 public ClimbMoveset GetClimbMoveset(){
237     return (ClimbMoveset)moveSet[(int)HeroStatManager.
playerMoveStateTAB.Climb-1];
238 }
239 public ref HeroStatManager GetHeroStatManager()
240 {
241     return ref HeroStat;
242 }
243 public Vector3 getVelocityVector()
244 {
245     return rb.velocity;
246 }
247 public Vector3 getLookDirection()
248 {
249     return lookDirection;
250 }
251 public void setRespawnController(ref RespawnController
rc){RC = rc;}
252 }
```

Listing 3: Skyrpt HeroController – HeroController

Zamysłem stworzenia skryptu 4 było oddzielenie od siebie różnych akcji postaci i zawarcie ich do jednej klasy. Efektem końcowym jest kod, który może się bardzo rozrastać w przyszłości bez konieczności wprowadzania dużych zmian w klasie ‘HeroController’. ‘WalkMoveSet’ dziedziczy z klasy ‘MonoBehaviour’, ponieważ używa kilku funkcjonalności tej klasy ale nie jest komponentem obiektu. W polach tej klasy zawarto komponent ‘Rigidbody’ postaci, obiekt zarządzający statystykami postaci z klasy ‘HeroStatManager’, obiekt zarządzający sygnałami wejścia stworzony z klasy ‘PlayerInput’ oraz wektor 3D ‘movingPlane’, który określa jak przetwarzany jest wektor wejść z klawiatury – dla klawiszy ‘WASD’. W klasie znajdują się 2 metody wirtualne – ‘movementController’ i ‘onJump’.

Metoda ‘movementController’ odpowiada za wprawienie postaci w ruch w zależności od wcisniętych klawiszy. W liniach 23 -25 odbywa się ograniczenia długości wektora prędkości postaci w osiach X i Z do wartości prędkości maksymalnej. Linie 27 do 33 odpowiadają za wyhamowanie postaci, gdy żaden z klawiszy ‘WASD’ nie jest wcisnięty. W liniach 36-54 jest część skryptu odpowiedzialna za przyśpieszenie postaci w zależności od tego czy postać znajduje się w powietrzu, czy dotyka ziemi. Odpowiada za to linia 41, gdzie zamiast użycia formuły ‘if else’ zastosowano dodawanie. Miało to na celu przyśpieszenie działania kodu w tym miejscu. Linie 46-54 odpowiadają za zaokrąglenie prędkości spadania postaci do 100 jednostek na sekundę. ‘movementController’ jest wywoływana, przez metodę o takiej samej nazwie w klasie ‘HeroController’, która z kolei jest wywoywana w metodzie ‘FixedUpdate’.

Metoda ‘onJump’ jest wywoywana, gdy klawisz skoku został wcisnięty. W linii 59 jest sprawdzany warunek czy postać może skoczyć. Następnie wektor prędkości w osi Y jest zerowany, i zostaje do niego wprowadzona nowa wartość za pomocą metody ‘AddForce’ obiektu ‘rb’. W linii 63 następuje dekrementacja licznika skoków. Metoda zwraca wartość true, gdy postać wykona skok lub wartość false, gdy skok nie nastąpi.

```

1  public class WalkMoveSet : MonoBehaviour
2  {
3      protected Vector3 movingPlane = new Vector3(1, 1, 0);
4      /*
5          movingPlane -- określa czy postać może poruszać się
6          lewo prawo, przed tyl, gora dol
7          lookDirection -- saved last input vector > 0
8          */
9
10     protected Rigidbody rb;
11     protected PlayerInput input;

```

```

12         protected HeroStatManager HeroStat;
13
14     public WalkMoveSet()
15     {
16         rb = GameObject.Find("Player1").GetComponent<
Rigidbody>();
17
18         input = GameObject.Find("Player1").GetComponent<
PlayerInput>();
19         HeroStat = GameObject.Find("Player1").GetComponent<
HeroController>().GetHeroStat();
20     }
21
22     public virtual void movementController(float
groundMultiplayer, Vector3 inputMoveVector, Quaternion rot)
23     {
24         Vector3 tmp;
25         float speed = HeroStat.getMvSpeed();
26         tmp = Vector3.ClampMagnitude(new Vector3(rb.velocity
.x, 0.0f, rb.velocity.z), speed * groundMultiplayer);
27         //clamp moving speed
28         if (inputMoveVector.sqrMagnitude <= 0.001f)
29         {
30             if (tmp.sqrMagnitude >= 0.01)
31             {
32                 rb.AddForce(-tmp.normalized * HeroStat.
acceleration * 0.8f, ForceMode.Acceleration);
33             }
34         else
35         {
36             //tmp = this.transform.rotation * Vector3.Scale(
inputMoveVector, movingPlane).normalized * speed *
groundMultiplayer;
37
38             rb.velocity = new Vector3(tmp.x, rb.velocity.y,
tmp.z);
39
40             tmp = rot * Vector3.Scale(inputMoveVector,
movingPlane).normalized * HeroStat.acceleration
41             * (((System.Convert.ToInt32(HeroStat.isGrounded)
+ 1) % 2 * HeroStat.AirControll + (System.Convert.ToInt32(
HeroStat.isGrounded))));;
42             rb.AddForce(tmp, ForceMode.Acceleration);
43
44         }
45         //clamp fallling speed
46         if (rb.velocity.y * rb.velocity.y > 10000.0f)
47         {
48             tmp = Vector3.zero;
49             tmp.y = rb.velocity.y;
50             tmp = Vector3.ClampMagnitude(tmp, 100);
51             tmp.x = rb.velocity.x;
52             tmp.z = rb.velocity.z;
53             rb.velocity = tmp;
54         }
55     }

```

```

56     public virtual bool onJump(InputAction.CallbackContext
57     context)
58     {
59         Vector3 tmp = Vector3.zero;
60         if ((HeroStat.jumpsCounter > 0 || HeroStat.
61         isGrounded))
62         {
63             rb.velocity = new Vector3(rb.velocity.x, 0.0f,
64             rb.velocity.z);
65             rb.AddForce(0.0f, Mathf.Sqrt(-HeroStat.
66             jumpStrength * Physics.gravity.y), 0.0f, ForceMode.
67             VelocityChange);
68             HeroStat.jumpsCounter -= 1;
69             return true;
70         }
71         return false;
72     }
73 }

```

Listing 4: Skrypt ‘WalkMoveSet’

Klasą dziedziczącą z klasy ‘WalkMoveSet’ jest klasa ‘ClimbMovesset’ (listing 5). Klasa odpowiada za przetwarzanie sygnałów wejścia, gdy postać wspina się oraz posiada metodę sprawdzającą, czy postać może się wspinać. W polach klasy znajdują się:

- obiekt ‘Hit’, który zawiera informacje o kolizji
- wysokość oraz szerokość, obiektu collider'a postaci
- ‘lookDirection’ – wektor, który określa kierunek, w którym jest skierowana postać
- zestaw flag

Metoda ‘canClimb’ określa czy postać spełnia warunki do wspinania. Metoda zwraca wartość true jeżeli warunki zostały spełnione. Sprawdzanie flagi ‘isJumping’ ma na celu sprawdzenie, czy gracz nie wydał polecenia skoku. Linie 45, 46, 47 działają w podobny sposób. Różnica polega na sprawdzaniu innych wysokości – na wysokości oczu, w połowie obiektu ‘collider’ postaci oraz na poziomie kolana postaci. Jeżeli obiekt, który znajduje się w warstwie kolizji określonej w masce został trafiony przez promień z metody ‘Raycast’, to zostaje zwrócona wartość true. Pole ‘iWantClimb’ i ‘wantEdgeJump’ przechowują wyniki tych operacji. W linii 48 skryptu zostaje sprawdzany warunek, czy postać może wykonać spięcie się po krawędzi. Następnie zmieniona

jest odpowiednia flaga i zostaje włączona odpowiednia animacja. Metoda ‘Play‘ obiektu ‘anim‘ przyjmuje, nazwę animacji – kontener, w którym się znajduje oraz nazwa węzła w animatorze – oraz identyfikator warstwy animatora. Linia 52 odpowiada za ‘odmrożenie‘ – obliczenia związane z fizyką nie zmieniają pozycji lub rotacji postaci – pozycji osi Y – constraints jest zmienną typu int, gdzie na każdym biecie zapisane jest inna oś rotacji lub pozycji.

Nadpisana metoda ‘movementController’ została podzielona na 3 segmenty ‘if else‘. Pierwszy segment – linie 60–67 – opowiadają za wspinaczkę do góry. Pierwszą częścią segmentu jest ‘odmrożenie‘ osi Y oraz zresetowanie liczby skoków. Następnie zostaje zmieniona prędkość postaci w zależności od stanu flagi ‘goingUP‘. Na końcu zostaje podniesiona flaga ‘climbUp‘. Drugim segmentem jest wspinanie się w dół. Ostatnim segmentem jest ‘spoczynek na ścianie‘. Postać stoi w miejscu i się nie porusza. Pole ‘constraints‘ w obiekcie ‘rb‘ zostało wykorzystane do zamrożenia postaci w miejscu. W trakcie sprowadzania prędkości postaci w osi Y do 0 powodowało minimalny spadek postaci, a było to spowodowane faktem, że skrypty w metodzie ‘FixedUpdate‘ uruchamiane są przed obliczeniami fizycznymi silnika. Ostatnia linia metody zmienia parametr animatora ‘ClimbUp‘.

Metoda ‘onJump‘ jest dziedziczona z klasy ‘WalkingMoveSet‘ więc została nadpisana. Moment skoku postaci w założeniu miał następować w pewnym miejscu animacji. W tym celu musiała się pojawić odpowiednia flaga ‘isJumping‘. Było to spowodowane tym, że postać zostawała w miejscu – pole ‘Constaraints‘ obiektu ‘rb‘ były modyfikowane. Pole ‘LookDirection‘ zostaje negowane, ponieważ postać została by roztowana 2 razy – raz przez animacji i drugi raz przez skrypt.

Po pewnym czasie po uruchomieniu metody ‘onJump‘ zostaje uruchomiona metoda ‘jump‘, która sprawia, że postać odskakuje od krawędzi. Zmienna ‘tmp‘ w linii 27 zawiera wektor prostopadły do ściany, do której ‘przykleiła się‘ postać. Długość tego wektora jest równa 1, ale do wartości w osi Y zostaje dodane 1, a następnie wszystko jest ponownie normalizowane. Ma to na celu sprawienie, że postać będzie wyskakiwać trochę do góry. W liniach 30–33 są zmieniane odpowiednie flagi i liczniki, oraz zostaje zmieniony stan postaci – postać od tej pory nie wspina się.

Wspięcie się na krawędzi zostało podzielone na 2 etapy – wyskoczenie do góry i wyskoczenie w przód. Metody ‘onEdgeUp‘ i ‘onEdgeForward‘ dopowiadają za zmianę odpowiednich flag i wartości – wraz z prędkością postaci w odpowiednim kierunku.

Metoda ‘ExitState‘ jest odpowiedzialna za ustawnie odpowiednich flag do wartości domyślnych – wprowadzenie tej metody pozwoliło na zmniejszenie błędów w grze.

```
1  public class ClimbMoveset : WalkMoveSet
2  {
3      private RaycastHit hit;
4      private float capsWidth, capsheight;
5      private Vector3 lookDirection;
6      private bool isJumping, goingUp, iWantClimb,
wantEdgeJump, edgeClimb;
7      Animator anim;
8
9      public ClimbMoveset(int hitmask, float capwidth, float
capHeig)
10     {
11         movingPlane = new Vector3(0, 1, 1);
12         capsWidth = capwidth;
13         capsheight = capHeig;
14
15     }
16     // Start is called before the first frame update
17     void Start()
18     {
19         isJumping = false;
20         iWantClimb = false;
21     }
22     public void jump()
23     {
24         rb.constraints = (rb.constraints |
RigidbodyConstraints.FreezePositionY)^RigidbodyConstraints.
FreezePositionY;
25         Vector3 tmp = Vector3.zero;
26         tmp = new Vector3(hit.normal.x, hit.normal.y + 1,
hit.normal.z).normalized;
27         Debug.Log(tmp);
28         rb.AddForce(Mathf.Sqrt(-HeroStat.jumpStrength *
Physics.gravity.y) * tmp, ForceMode.VelocityChange);
29         HeroStat.jumpsCounter -= 1;
30         HeroStat.isClimbing = false;
31         isJumping = false;
32         HeroStat.changeState(true);
33         ExitState();
34     }
35     public bool canClimb(Vector3 lookDir, int hitableMasks)
36     {
37         if (!isJumping)
38         {
39             lookDirection = lookDir;
40         }
41         else {
42             return true;
43         }
44         iWantClimb = (Physics.Raycast(rb.position + new
Vector3(0.0f, capsheight*0.8f, 0.0f), lookDirection, out hit,
capsWidth * 1.1f, hitableMasks));
45     }
46 }
```

```

46         wantEdgeJump = !iWantClimb && (Physics.Raycast(rb.
position, lookDirection, out hit, capsWidth * 1.1f,
hitableMasks) ||
47             (Physics.Raycast(rb.position + new Vector3(0.0f, -
capsheight*0.8f, 0.0f), lookDirection, out hit, capsWidth * 1.1f, hitableMasks)));
48             if(wantEdgeJump && !edgeClimb && input.actions["
Vertical"].ReadValue<float>()>0)
49             {
50                 edgeClimb = true;
51                 anim.Play("Climb.ClimbOnEdge",anim.GetLayerIndex
("Base Layer"));
52                 rb.constraints = (rb.constraints |
RigidbodyConstraints.FreezePositionY)^RigidbodyConstraints.
FreezePositionY;
53             }
54             return iWantClimb || wantEdgeJump;
55         }
56         public override void movementController(float
groundMultiplayer, Vector3 inputMoveVector, Quaternion rot)
57         {
58             bool climbUp = false;
59             //if climbing upwards
60             if(input.actions["Vertical"].ReadValue<float>() > 0
&& !isJumping && !edgeClimb)
61             {
62                 //make sure that rb unfreezes
63                 rb.constraints = (rb.constraints |
RigidbodyConstraints.FreezePositionY)^RigidbodyConstraints.
FreezePositionY;
64                 HeroStat.jumpsCounter = HeroStat.getMaxJumps();
65                 rb.velocity = new Vector3(0.0f, HeroStat.
getMvSpeed() * System.Convert.ToSingle(goingUp), 0.0f);
66                 climbUp = true;
67             }
68             //if climbing downwards
69             else if(input.actions["Vertical"].ReadValue<float>()
< 0 && !isJumping)
70             {
71                 //make sure that rb unfreezes
72                 rb.constraints = (rb.constraints |
RigidbodyConstraints.FreezePositionY)^RigidbodyConstraints.
FreezePositionY;
73                 rb.velocity = new Vector3(0.0f, HeroStat.
getMvSpeed() * input.actions["Vertical"].ReadValue<float>(),
0.0f);
74             }
75             //if staing in position
76             else if(!isJumping && iWantClimb)
77             {
78                 //make sure that rb freezes
79                 rb.constraints = (rb.constraints |
RigidbodyConstraints.FreezePositionY);
80             }
81             anim.SetBool("ClimbUp", climbUp);
82         }

```

```

83
84     }
85     public override bool onJump(InputAction.CallbackContext
86 context)
87     {
88         if (HeroStat.jumpsCounter > 0)
89         {
90             anim.Play("Climb.ClimbJump", anim.GetLayerIndex(
91 "Base Layer"));
92             lookDirection = -lookDirection;
93             rb.constraints = rb.constraints | 
RigidbodyConstraints.FreezePositionY;
94             isJumping = true;
95             iWantClimb = false;
96         }
97         return isJumping;
98     }
99
100    public void setAnimator(ref Animator animat)
101    {
102        anim = animat;
103    }
104    public void setgoingUp(bool goup){
105        goingUp = goup;
106    }
107    public void onEdgeUp(){
108        edgeClimb = false;
109        rb.AddForce(Vector3.up*5.0f, ForceMode.
VelocityChange);
110    }
111    public void onEdgeForward(){
112        rb.AddForce(lookDirection.normalized*2.0f, ForceMode
.VelocityChange);
113        edgeClimb = false;
114        ExitState();
115    }
116    public void ExitState(){
117        rb.constraints = (rb.constraints |
RigidbodyConstraints.FreezePositionY)^RigidbodyConstraints.
FreezePositionY;
118        anim.SetBool("ClimbUp", false);
119        edgeClimb = false;
120    }
121    public void EnterState(){ }

```

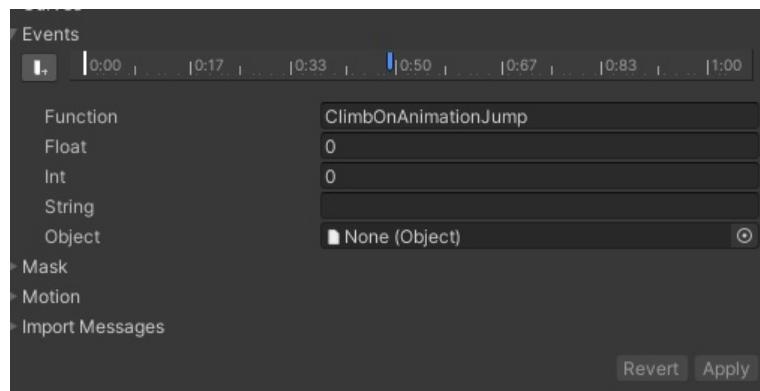
Listing 5: Skyrpt ‘ClimbMoveset’

## 5.5. Połączenie skryptu i animacji

System wydarzeń w animacjach został stworzony w celu ułatwienia synchronizacji włączania dźwięku chodzenia z stawianiem stopy na podłożu. Ten system można jednak wykorzystać na kilka innych sposobów. W tym projekcie został użyty do synchronizacji skryptów z animacjami.

W celu ustalenia wydarzenia animacji assetu importowanego w formacie fbx należy wejść w ustawienia importu – klikając dwukrotnie lewym przyciskiem myszki na dany asset – następnie przejść w zakładkę ‘Animations’. Tam znajduje się zakładka ‘Events’. Metody będą wywoływanie z obiektu dołączonego jako komponent do obiektu stworzonego z assetu. Nazwa obiektu nie jest ważna. Nazwy metod w edytorze wydarzeń animacji i w klasie, z której tworzony jest obiekt muszą się zgadzać.

Na rysunku 5.75 zostało pokazane jedno zdarzenie. Jest to zdarzenie, które powoduje u postaci odskok od ściany. Wywoływana funkcja nie przyjmuje argumentów, dlatego wszystkie pola – poza function – zostały zostawione z wartościami domyślnymi.

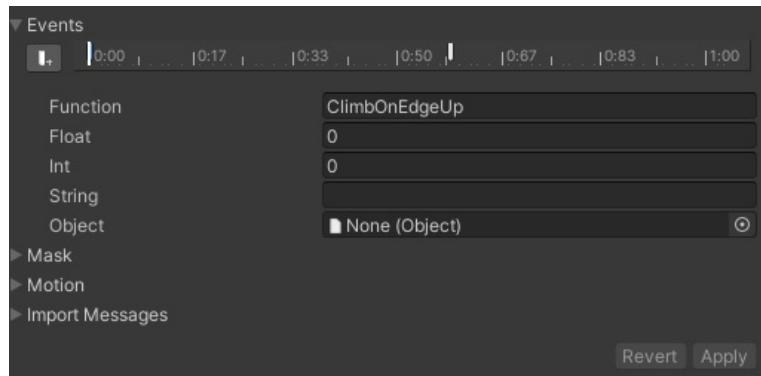


Rysunek 5.75: Implementacja wydarzenia w animacji odskoku od ściany

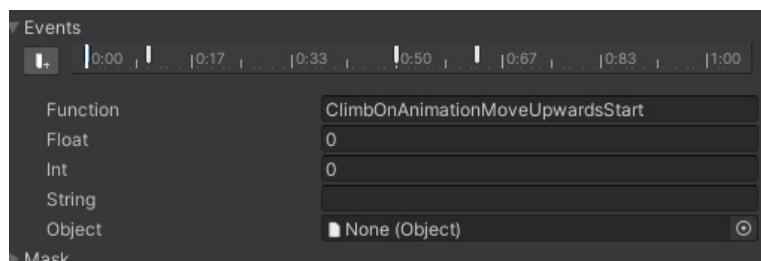
Na rysunku 5.76 zostały zaimplementowane 2 wydarzenia: wspięcia się do góry – ‘ClimbOnEdgeUp’ – oraz wyrzucenia do przodu - ‘ClimbOnEdgeForward’.

Na rysunku 5.77 przedstawiono wydarzenia w animacji wspinania się postaci do góry ‘ClimbUP’. Są 2 metody, które wywoływanie są na przemian: jedna powoduje, że wspina się – ‘ClimbOnAnimationMoveUpwardsStart’ – a druga powoduje, że postać zatrzymuje się - ‘ClimbOnAnimationMoveUpwardsStop’.

Skrypt 6 łączy w sobie 2 funkcjonalności – aktualizację parametrów animatora i zawarcie metod wywoływanych w wydarzeniach. Aktualizacją parametrów zajmuje się



Rysunek 5.76: Implementacja wydarzeń w animacji wsipięcia się po krawędzi postaci



Rysunek 5.77: Implementacja wyadrzeń podczas wspinania się postaci do góry

metoda Update wywoływana co klatkę.

```

1  public class HeroAnimationScript : MonoBehaviour
2 {
3     [SerializeField] HeroController HC;
4     [SerializeField] Animator animator;
5     Vector2 mvVeloffset;
6     //basic MonoBehaviour
7     void Start(){
8         mvVeloffset = new Vector2();
9     }
10    void Update(){
11        updateAnimationParameters();
12    }
13    //methods called on animation event
14    void ClimbOnAnimationMoveUpwardsStart(){
15        HC.GetClimbMoveset().setgoingUp(true);
16    }
17    void ClimbOnAnimationMoveUpwardsStop(){
18        HC.GetClimbMoveset().setgoingUp(false);
19    }
20    void ClimbOnAnimationJump(){
21        HC.GetClimbMoveset().jump();
22    }
23    void ClimbOnEdgeUp(){
24        HC.GetClimbMoveset().onEdgeUp();
25    }
26 }
```

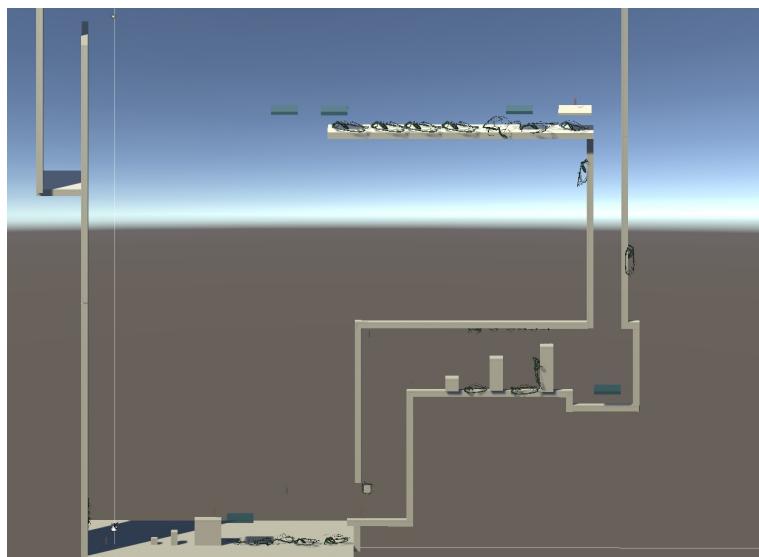
```

27     void ClimbOnEdgeForward(){
28         HC.GetClimbMoveset().onEdgeForward();
29     }
30     //method for setting animation parameters
31     private void updateAnimationParameters(){
32         aChangeState();
33         aChangeMVSpeed();
34     }
35     public void aChangeJumpFlag()
36     {
37         animator.SetTrigger("Jump");
38     }
39
40     private void aChangeState()
41     {
42         animator.SetInteger("PrevState", animator.GetInteger("State"));
43         animator.SetInteger("State", HC.GetHeroStat().getState());
44     }
45     private void aChangeMVSpeed()
46     {
47         Vector3 tmp = HC.getVelocityVector();
48         animator.SetFloat("MVSpeed", Mathf.Sqrt(tmp.x * tmp.x +
49             tmp.z * tmp.z));
50         animator.SetFloat("YVelocity", tmp.y);
51         tmp.Normalize();
52         Vector2 tmp2 = new Vector2((HC.getLookDirection().
53             normalized.x - tmp.x)*System.Convert.ToSingle(tmp.x>0.01f &&
54             tmp.x<-0.01f,
55             (HC.getLookDirection().normalized.z-tmp.z -1))*System.
56             Convert.ToSingle(tmp.z>0.01f && tmp.z<-0.01f);
57         //obliczenia opoznionej predkosci
58         mvVeloffset.x -= (mvVeloffset.x/5 + tmp2.SqrMagnitude())/
59         2;
60         mvVeloffset.y -= (mvVeloffset.y/8 + tmp.y)/4;
61         animator.SetFloat("XVelocityOffset", mvVeloffset.x);
62         animator.SetFloat("YVelocityOffset", mvVeloffset.y);
63     }
64 }
```

Listing 6: Skyrpt ‘HeroAnimationScript’

## 6. Efekt Pracy

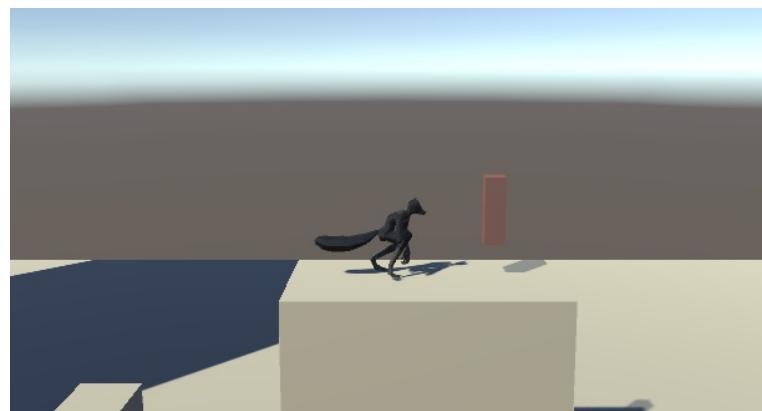
Na rysunku 6.78 przedstawiono mapę stworzoną do gry. Na rysunku 6.79 znajduje się postać sterowana przez gracza.



Rysunek 6.78: Mapa gry

Ruchem postaci steruje się za pomocą klawiszy ‘WASD’, klawisz spacji odpowiada za skok, klawisz lewy ‘Shift’ powoduje, że postać zaczyna biec. Jeżeli postać znajduje się blisko ściany i zostanie wciśnięty klawisz ‘lewy CTRL’ to postać rozpoczyna wspinaczkę jak na rysunku 6.80. Na rysunku 6.79 znajduje się postać sterowana przez gracza. Naprzeciw niej znajduje się punkt kontrolny, do którego postać zostaje przenoszona, gdy zdrowie postaci spadnie do 0. W lewym górnym rogu znajdują się paski wskazujące na ilość zdrowia – czerwony – oraz wytrzymałości – zielony.

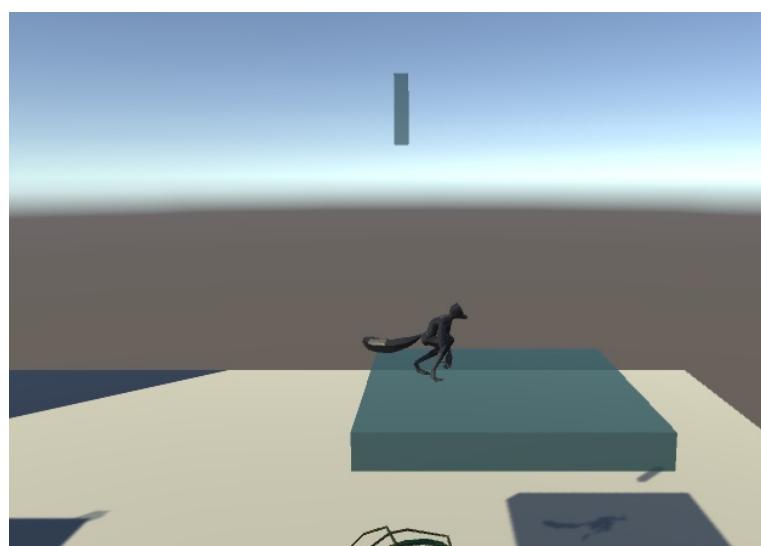
Na rysunku 6.81 można zobaczyć postać jadącą na platformie. Niebieski poł przezroczysty słupek spowoduje zwiększenie statystyk.



Rysunek 6.79: Postać stoi naprzeciw punktu kontrolnego



Rysunek 6.80: Postać wspina się w górę



Rysunek 6.81: Platforma i punkt zwiększenia statystyk

## 7. Podsumowanie i wnioski końcowe

Udało się zrealizować cel projektu polegający na napisaniu platformowej gry 3D. Zaimplementowano wszystkie funkcjonalności. W projekcie został stworzony poziom, który gracz może przemierzyć. Zaimplementowano prosty interfejs graficzny pokazujący stan zdrowia i wytrzymałość bohatera. Modele, tekstury i animacje zostały stworzone i zaimplementowane oraz działają prawidłowo. Skrypty w trakcie testów ręcznych nie wykazały błędów. Utworzono prosty system statystyk postaci, który działa prawidłowo. Za wkład własny w realizację projektu Autor uważa:

- Opracowanie
- Stworzenie modelu, animacji i tekstur postaci głównej
- Stworzenie modelu, animacji i tekstur otoczenia
- Przygotowanie prostej mapy
- Stworzenie skryptów do zarządzania statystykami gracza, sterowania postacią główną,
- Przygotowanie prostego interfejsu graficznego pokazującego stan zdrowia i wytrzymałości postaci
- Połączenie animacji i skryptów

Skrypty gry były pisane o możliwości dalszego rozwoju, możliwe jest rozwinięcie wachlarza umiejętności postaci. Dla modelu postaci można przeprowadzić proces retopologizacji – proces upraszczania siatki, który powoduje, że siatka jest bardziej ‘czysta’ – przez co animacje będą wyglądać lepiej. W celu urozmaicenia rozgrywki można dodać inne elementy przeszkadzające w trakcie gry. Przykładem mogą być rozmieszczone w różnych miejscach lasery lub wieżyczki. Inną opcją jest dodanie sterowanych przez sztuczną inteligencję przeciwników podążających za graczem. GUI można rozwiniąć o menu główne, menu ustawień oraz interfejs wyświetlający wszystkie statystyki postaci. Jest możliwe również dodanie kilku map.

## Załączniki

- 1) Kod źródłowy
- 2) Model opracowanej postaci
- 3) Modele elementów otoczenia

## Literatura

- [1] <https://www.blender.org/> Dostęp 17.01.2023
- [2] <https://quixel.com/mixer> Dostęp 17.01.2023
- [3] [https://store.steampowered.com/app/1378990/Crash\\_Bandicoot\\_4\\_Its\\_About\\_Time/](https://store.steampowered.com/app/1378990/Crash_Bandicoot_4_Its_About_Time/) Dostęp 17.01.2023
- [4] <https://store.steampowered.com/app/504230/Celeste/> Dostęp 17.01.2023
- [5] [https://store.steampowered.com/app/261570/Ori\\_and\\_the\\_Blind\\_Forest/?l=polish](https://store.steampowered.com/app/261570/Ori_and_the_Blind_Forest/?l=polish) Dostęp 17.01.2023
- [6] <https://unity.com/> Dostęp 17.01.2023
- [7] <https://www.unrealengine.com/en-US/unreal-engine-5> Dostęp 17.01.2023
- [8] <https://www.incredibuild.com/blog/unity-vs-unreal-what-kind-of-game-dev-are-you> Dostęp 17.01.2023
- [9] <https://docs.unity3d.com/Manual/ExecutionOrder.html> Dostęp 17.01.2023
- [10] [https://docs.blender.org/manual/en/latest/interface/window\\_system/workspaces.html](https://docs.blender.org/manual/en/latest/interface/window_system/workspaces.html) Dostęp 17.01.2023
- [11] <https://www.youtube.com/watch?v=jstOXABEj24> Dostęp 17.01.2023
- [12] [https://docs.blender.org/manual/en/latest/animation/constraints/tracking\\_ik\\_solver.html](https://docs.blender.org/manual/en/latest/animation/constraints/tracking_ik_solver.html) Dostęp 17.01.2023
- [13] <https://docs.blender.org/manual/en/latest/animation/constraints/interface/common.html#rigging-constraints-interface-common-space> Dostęp 17.01.2023

[14] <https://www.youtube.com/watch?v=za7BDv0-QsQ> Dostęp 17.01.2023

[15] <https://www.youtube.com/watch?v=j0yyjfzpKE8> Dostęp 17.01.2023

[16] [https://www.youtube.com/watch?v=Erqgl\\_PQyrk](https://www.youtube.com/watch?v=Erqgl_PQyrk) Dostęp 17.01.2023

[17] <https://www.youtube.com/watch?v=Y7M-B6xnaEM> Dostęp 17.01.2023

[18] [https://www.youtube.com/watch?v=scPSP\\_U858k](https://www.youtube.com/watch?v=scPSP_U858k) Dostęp 17.01.2023

[19] [https://www.youtube.com/watch?v=Yjee\\_e4fICc](https://www.youtube.com/watch?v=Yjee_e4fICc) Dostęp 17.01.2023

POLITECHNIKA RZESZOWSKA im. I. Łukasiewicza  
Wydział Elektrotechniki i Informatyki

Rzeszów, 2023

**STRESZCZENIE PRACY DYPLOMOWEJ INŻYNIERSKIEJ**  
**GRA PLATFORMOWA 3D**

Autor: Kacper Rudź, nr albumu: EF-164045

Opiekun: dr inż. Sławomir Samolej prof. PRz

Słowa kluczowe: Gra 3D Unity, Animacje

W pracy zawarto proces tworzenia trójwymiarowej gry platformowej. Opisano proces przygotowania szkieletu i animacji postaci humanoidalnej. W pracy zawarto problemy, na jakie natrafił twórca podczas tworzenia animacji. Pokazano przygotowanie skryptów sterowania postacią.

RZESZOW UNIVERSITY OF TECHNOLOGY  
Faculty of Electrical and Computer Engineering

Rzeszow, 2023

**BSC THESIS ABSTRACT**  
**3D PLATFROM GAME**

Author: Kacper Rudź, nr albumu: EF-164045

Supervisor: prof. PRz Sławomir Samolej PhD

Key words: 3D Unity Game, Animations

The thesis contains creation proces of 3D platformer game. The proces of preparing the skeleton and animation of humanoid character has been described. Thesis contains the problems encountered by the creator while creating the animansions. The preparation of character control scripts was shown.