Katolicki Uniwersytet Lubelski Jana Pawła II
Wydział Nauk Ścisłych i Nauk o Zdrowiu
Instytut Matematyki, Informatyki i Architektury Krajobrazu

Informatyka, studia stacjonarne I stopnia

Bartosz Zborowski
Nr albumu 150830

Trójwymiarowa gra wyścigowa wykonana w Blenderze i Unity

Praca licencjacka
napisana na seminarium **Zastosowania grafiki komputerowej**
pod kierunkiem
dr. Armena Grigoryana

Lublin 2022

Spis treści

Wstęp ... 1

Rozdział I: Zasoby graficzne wykonane w Blenderze ... 7

1.1 Układ torów ... 7

1.2 Barierki ... 10

1.3 Drzewo ... 11

1.4 Znaki ... 15

1.5 Samochód ... 19

Rozdział II: Implementacja logiki gry ... 24

2.1 Skrypt *CarBehaviour.cs* ... 24

2.2 Skrypt *CameraScript.cs* ... 27

2.3 Skrypty odpowiedzialne za okrążenia ... 29

2.4 Zarządzanie przyciskami i stanem gry ... 36

Rozdział III: Sceny gry ... 42

3.1 Menu główne, menu z instrukcją oraz menu wyboru toru ... 43

3.2 Pierwszy tor ... 48

3.3 Drugi tor ... 56

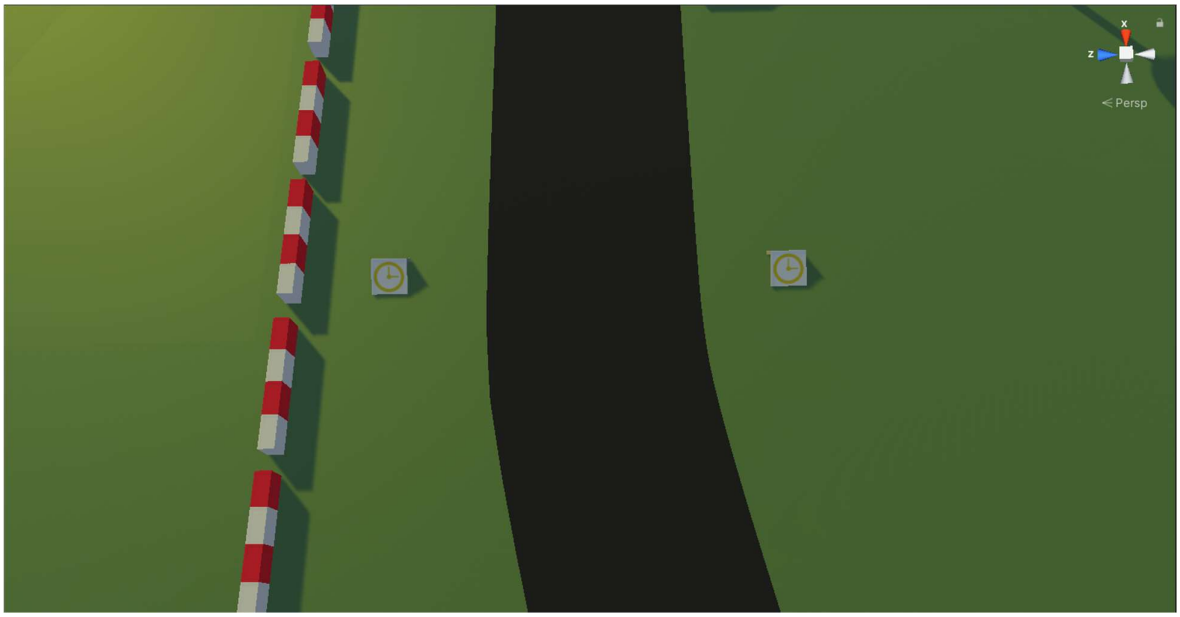
Podsumowanie ... 58

Bibliografia ... 59

Wstęp

Tematem niniejszej pracy jest przedstawienie zagadnień dotyczących tworzenia gry wyścigowej 3D z wykorzystaniem silnika gier Unity oraz oprogramowania trójwymiarowej grafiki komputerowej Blender. W tym celu została utworzona gra o nazwie BZRG2022 za pomocą Unity w wersji 2020.3.24f1. Poruszymy zagadnienia kreowania skryptów w języku programowania zorientowanego obiektowo C#. Służą one do implementacji logiki gry. Jako zintegrowane środowisko programistyczne wybraliśmy Microsoft Visual Studio w wersji 17.0.4. Wszystkie listingi zostały wykonane za pomocą tego edytora. Ponadto w pracy wyjaśnimy sposoby tworzenia modeli dla w/w projektu za pomocą oprogramowania Blender (wersja 3.1.2) oraz tekstur za pomocą programu GIMP (wersja 2.10.12). Wszystkie zasoby graficzne w/w gry, które zostały omówione w pierwszym rozdziale, zostały utworzone przez autora niniejszej pracy. Naszym głównym źródłem do Unity były pozycje [1] i [2], a do programu Blender – pozycja [3].

Celem gry wyścigowej BZRG2022 jest ustanowienie najkrótszego czasu na wybranym torze w ciągu trzech okrążeń samochodem kierowanym przez gracza. Za każdym razem, gdy gracz rozpoczyna wyścig od nowa, statystyki dotyczące najszybszego okrążenia są resetowane. Aby utrudnić walkę o najkrótszy czas, model prowadzenia samochodu został zaprogramowany na styl jazdy w kontrolowanym poślizgu, czyli gracz pokonuje zakręty „driftując". Pojazd podczas skręcania zaczyna jechać bokiem, więc należy tak obrócić i kontrolować samochód, aby zmieścił się w zakręcie. Dodatkowym utrudnieniem są ostre i czasem wąskie zakręty na torze, które są ograniczone barierkami. Jeżeli gracz zahaczy o barierkę samochodem, to nie dość, że spowolni, to jeszcze może stracić panowanie nad nim. Sterowanie pojazdem odbywa się przy wykorzystaniu samej myszki. Lewy przycisk myszy operuje gazem, prawy przycisk myszy operuje hamulcem. Kursorem myszy gracz wskazuje kierunek jazdy. Oczywiście na postoju samochód nie obraca się, więc im szybciej gracz jedzie tym szybciej jest w stanie skręcić nim. Aby okrążenie zostało prawidłowo zaliczone, trzeba przejechać przez punkt kontrolny znajdujący się mniej więcej w połowie każdego toru. Punkt kontrolny jest oznaczony znakami z symbolem zegara na białym tle (patrz *Rysunek 1*). Gracz musi między tymi znakami przejechać. Dopiero wtedy punkt kontrolny jest zaliczony. Po przekroczeniu linii mety okrążenie zostanie zakończone/zaliczone. Jeżeli gracz ominie punkt kontrolny (zachowanie nieprawidłowe), to po przekroczeniu linii mety nadal będzie liczony czas niezaliczonego okrążenia.



Rysunek 1. Punkt kontrolny

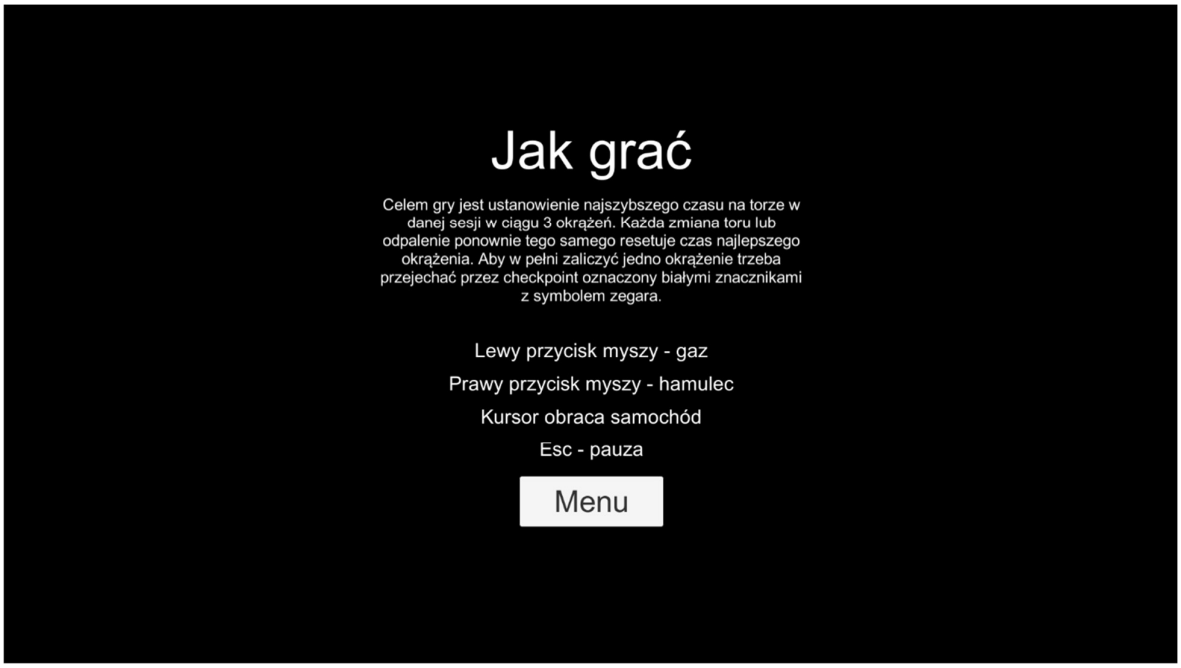
Źródło: opracowanie własne wygenerowane za pomocą Unity.

Poniżej widzimy menu główne gry (patrz *Rysunek 2*), które wita nas tuż po uruchomieniu programu. Możemy tutaj wybrać opcje „Graj”, „Jak grać” lub „Wyjdź”. W menu poruszamy się myszką.

Rysunek 2. Menu główne gry

Źródło: opracowanie własne wygenerowane za pomocą Unity.

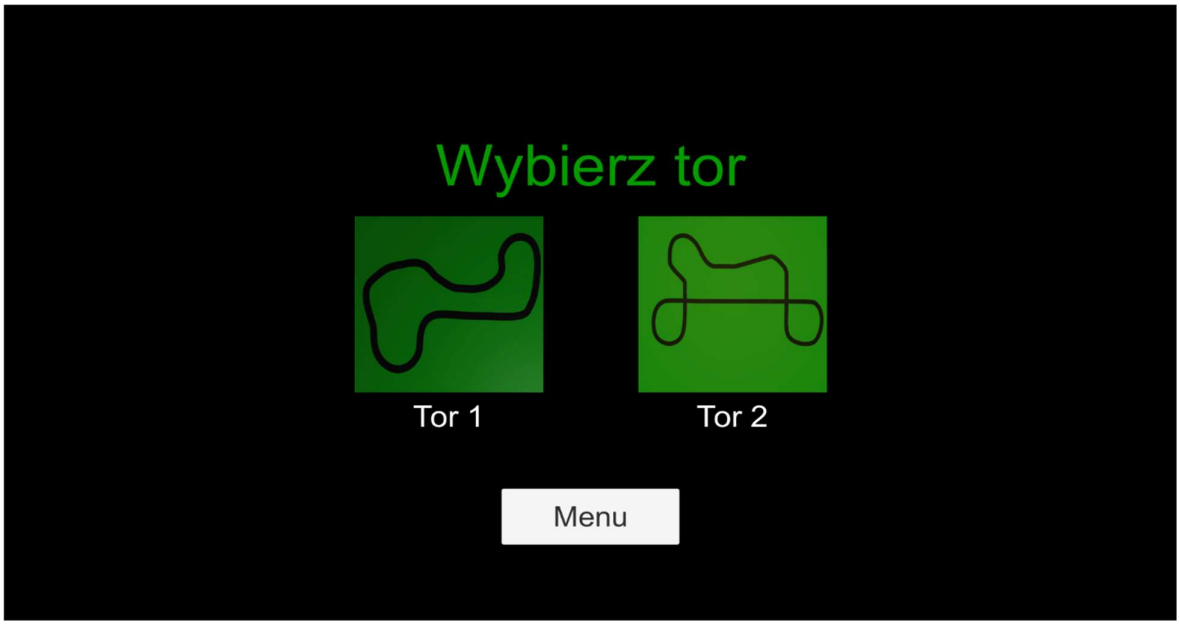
Wybierając opcję „Wyjdź” wychodzimy z gry. Po wybraniu „Jak grać” możemy przeczytać krótką instrukcję sterowania pojazdem oraz jaki jest cel gry (patrz *Rysunek 3*). Przycisk „Menu” pozwala wrócić do menu głównego.



Rysunek 3. Menu z opisem celu gry i sterowania

Źródło: opracowanie własne wygenerowane za pomocą Unity.

Wybierając w menu głównym przycisk „Graj” przenosimy się do menu wyboru toru (patrz *Rysunek 4*). Do wyboru są dwa tory: „Tor 1” i „Tor 2”. Układ każdego z nich jest przedstawiony na obrazku. Na dole okna znajduje się przycisk przenoszący nas do menu głównego. Klikając w obrazek lub nazwę toru, jesteśmy przenoszeni na wybraną trasę.



Rysunek 4. Menu z wyborem toru

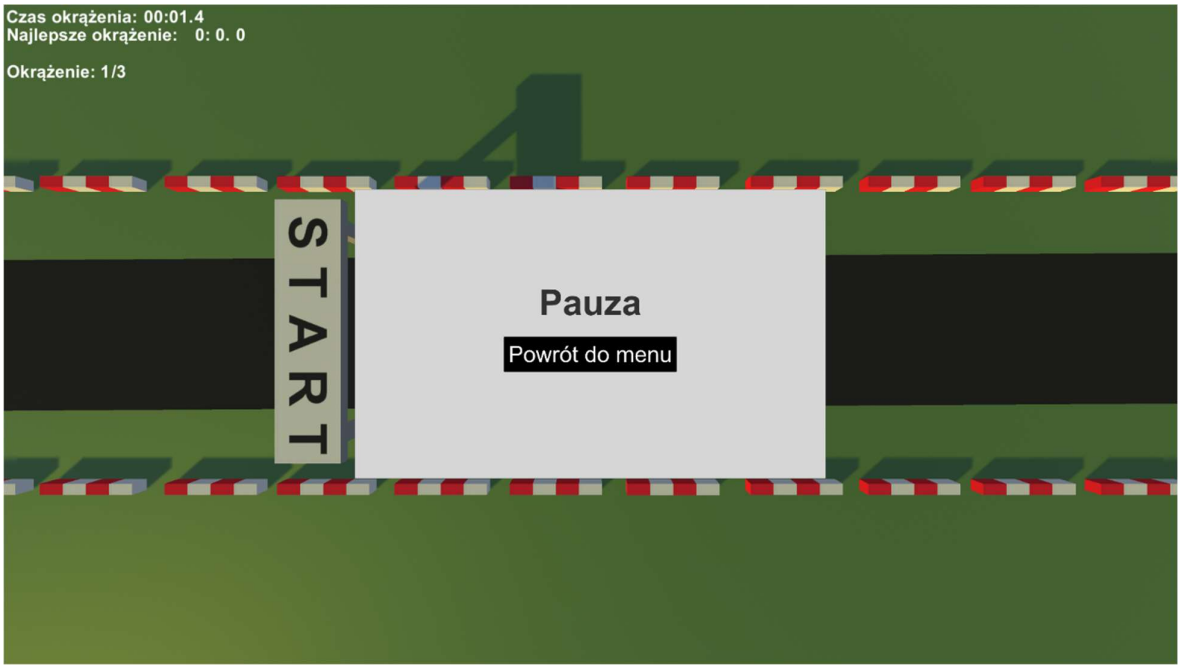
Źródło: opracowanie własne wygenerowane za pomocą Unity.

Po załadowaniu sceny z wybranym torem (patrz *Rysunek 5*) ukazuje nam się układ HUD (ang. Head-Up Display) zawierający czasy bieżącego oraz najlepszego okrążenia umiejscowione po lewej górnej stronie ekranu. Pod nimi jest jeszcze licznik zaliczonych okrążeń wraz z liczbą pokazującą ile okrążeń ma dana sesja. Aby zacząć wyścig trzeba kliknąć przycisk „START”. Po kliknięciu przycisku czas okrążenia zaczyna się naliczać.

Rysunek 5. Scena gry po wybraniu toru

Źródło: opracowanie własne wygenerowane za pomocą Unity.

Jeżeli mamy taką potrzebę możemy wstrzymać grę wciskając przycisk Esc na klawiaturze. Pojawi nam się menu pauzy pozwalające wrócić nam do menu głównego poprzez wciśnięcie przycisku „Powrót do menu” (patrz *Rysunek 6*). Wywołanie menu pauzy stopuje czas płynący w grze. Możemy wyłączyć pauzę i wrócić do gry ponownie wciskając klawisz Esc.



Rysunek 6. Menu pauzy podczas wyścigu

Źródło: opracowanie własne wygenerowane za pomocą Unity.

W pierwszym rozdziale pracy omawiamy sposoby tworzenia zasobów graficznych naszej gry. Drugi rozdział poświęcony jest analizie skryptów, które implementują logikę gry. W trzecim rozdziale podajemy opis organizacji naszego projektu w Unity. Praca została zakończona podsumowaniem i bibliografią.

Do uruchomienia i testowania gry BZRG2022 zostały wykorzystane komputery o poniższych konfiguracjach.

Komputer 1:

- Procesor: AMD Ryzen 5 2600
- Karta graficzna: Nvidia GeForce GTX 1060 6GB
- Pamięć RAM: 2x8GB DDR4 3000MHz CL16
- System: Windows 10 Pro x64 wersja 21H2

Komputer 2:

- Procesor: AMD Ryzen 5 5600X
- Karta graficzna: Nvidia GeForce RTX 3070 Ti 8GB
- Pamięć RAM: 2x8GB DDR4 3600MHz CL18
- System: Windows 10 Pro x64 wersja 21H2

Laptop:

- Procesor: AMD Ryzen 5 2500U
- Karta graficzna: AMD Radeon Vega 8
- Pamięć RAM: 2x4GB DDR4 2400MHz CL17
- System: Windows 10 Home x64 wersja 20H2

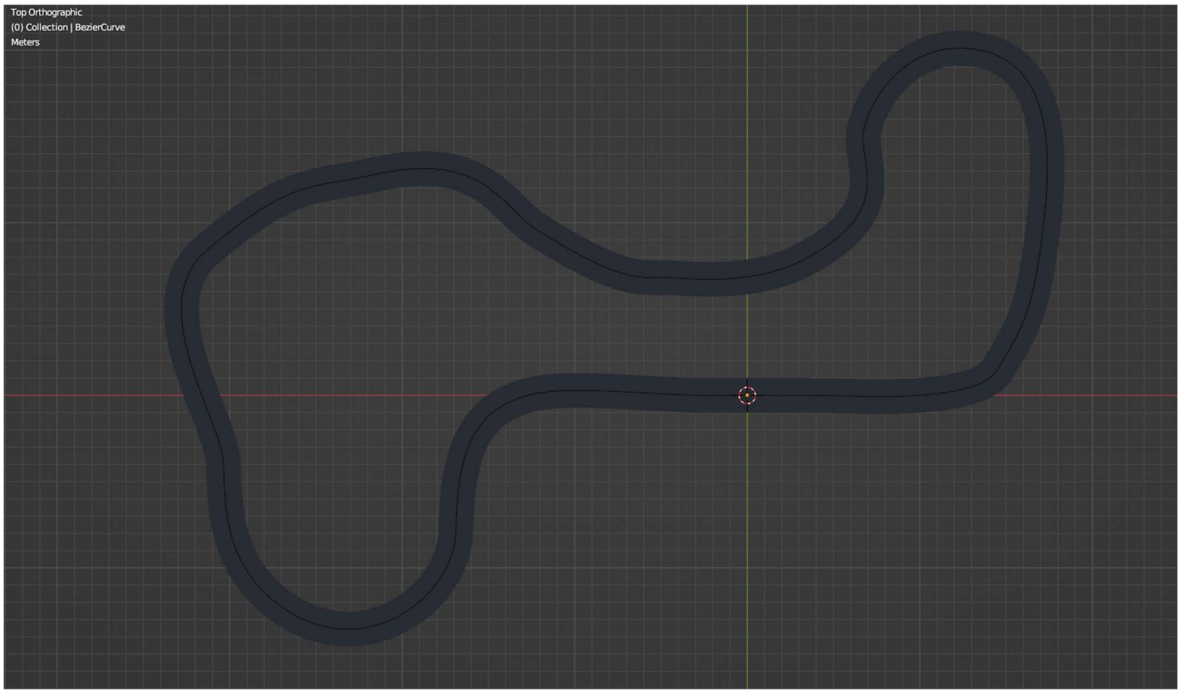
Płyta CD załączona do niniejszej pracy licencjackiej jest jej integralną częścią. Zawiera ona plik *Instrukcja.pdf* z instrukcją obsługi tejże płyty, plik *BZRG2022_build.zip* zawierający build gry, projekt *BZRG2022.zip* wykonany w Unity, oraz pliki *Blender_modele_fbx.zip* i *Zasoby_gimp.zip* zawierające zasoby własnego opracowania autora odpowiednio modeli 3D w formacie FBX i tekstur. Aby uruchomić grę należy wypakować plik *BZRG2022_build.zip* i uruchomić plik *BZRG2022.exe* znajdujący się w folderze *BZRG2022_build*.

Rozdział I: Zasoby graficzne wykonane w Blenderze

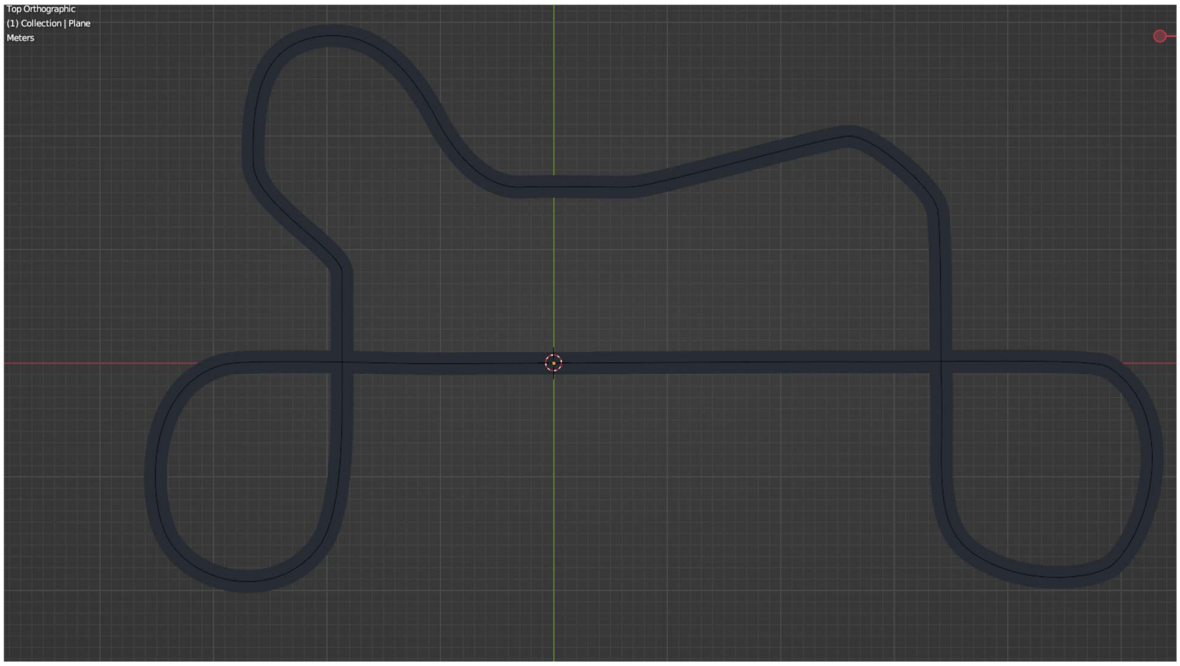
W niniejszym rozdziale omówimy zasoby graficzne naszego projektu BZRG2022. Zostały one wykonane przez autora niniejszej pracy.

1.1 Układ torów

Podstawą każdej gry wyścigowej, oprócz modelu jazdy, jest projekt tras. W naszym przypadku zostały stworzone dwa różne tory wyścigowe z wykorzystaniem programu Blender. Każdy tor został utworzony podobnymi metodami. Jedyne czym tory te się różnią to ich kształt (patrz *Rysunek 7* i *Rysunek 8*).



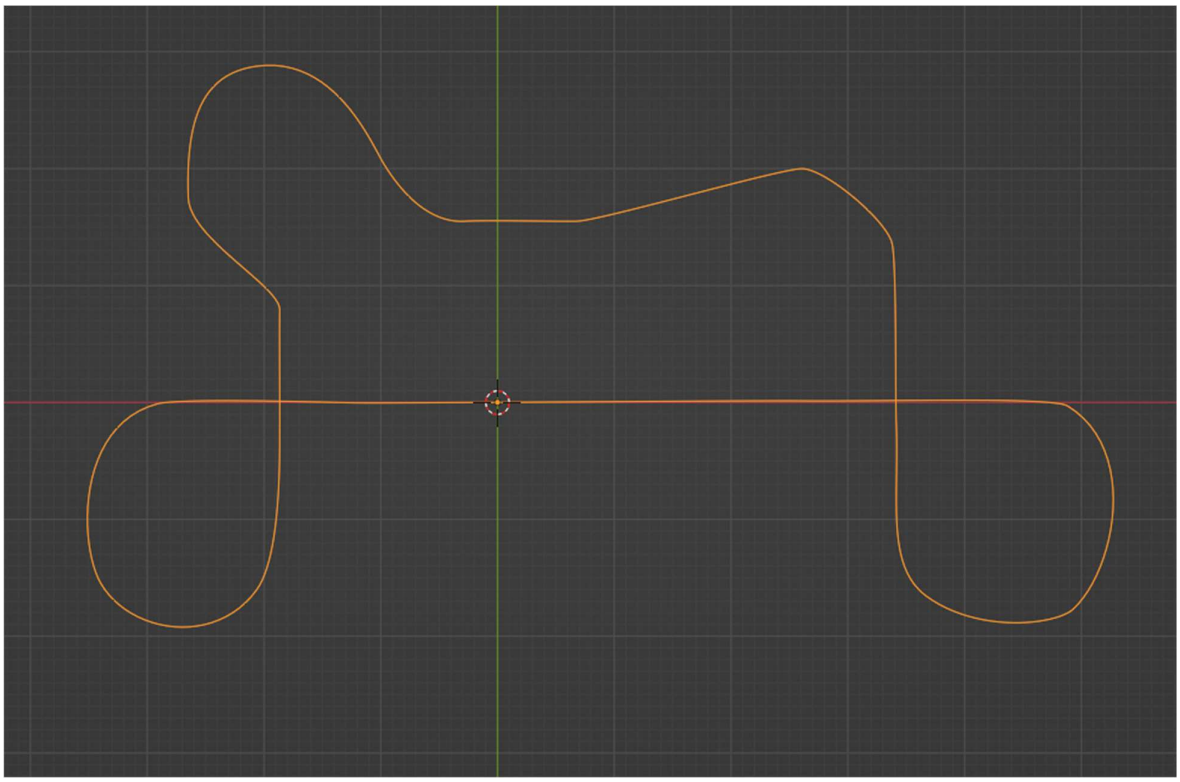
Rysunek 7. Układ pierwszego toru
Źródło: opracowanie własne wygenerowane za pomocą Blender.



Rysunek 8. Układ drugiego toru

Źródło: opracowanie własne wygenerowane za pomocą Blender.

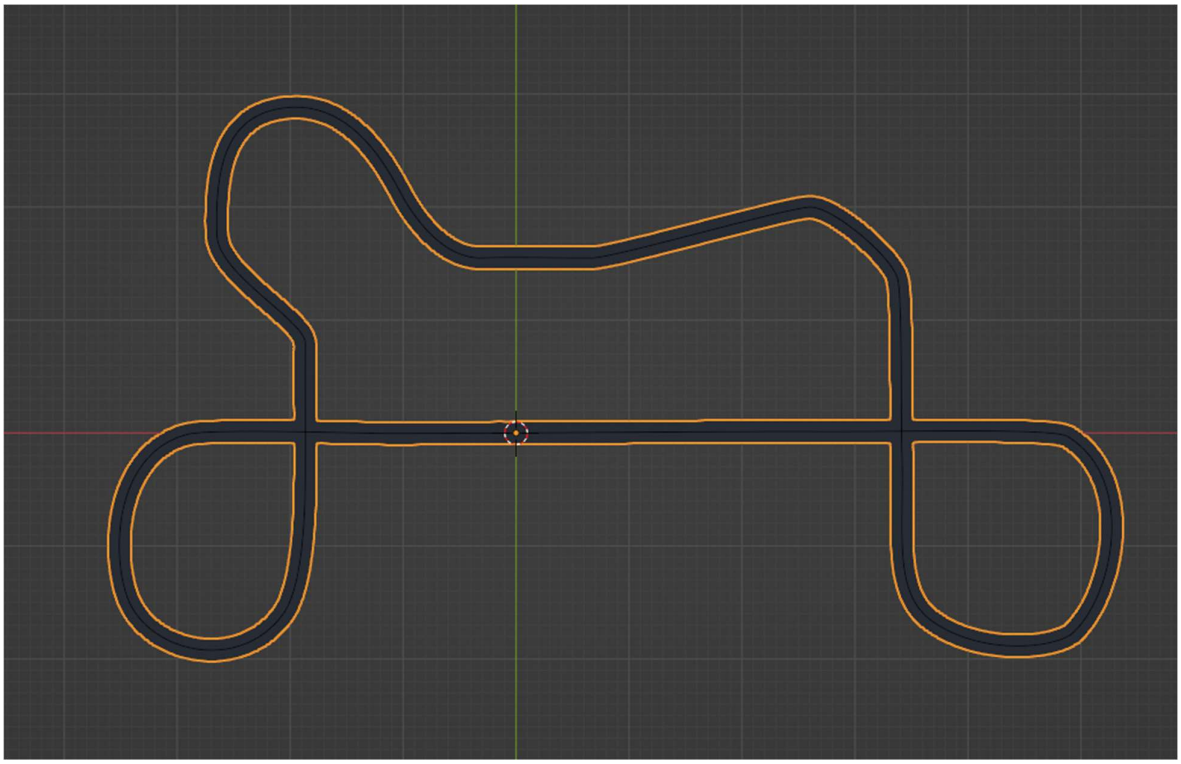
Na początku została dodana krzywa sklejana o segmentach będących krzywą *Bezier*, która tworzy kształt trasy według zamysłu autora (patrz *Rysunek 9*).



Rysunek 9. Krzywa Bezier tworząca układ toru wyścigowego

Źródło: opracowanie własne wygenerowane za pomocą Blender.

Następnie został dodany na scenie obiekt *Plane* typu *mesh* (ang. siatka) do którego zostały przypisane modyfikatory o nazwie *Array* i *Curve*. Modyfikator *Array* tworzy szereg obiektów złożony z *Plane*. Modyfikator *Curve* mówi szeregowi *Array*, że ma on ułożyć się wzdłuż krzywej *Bezier* (patrz *Rysunek 10*).

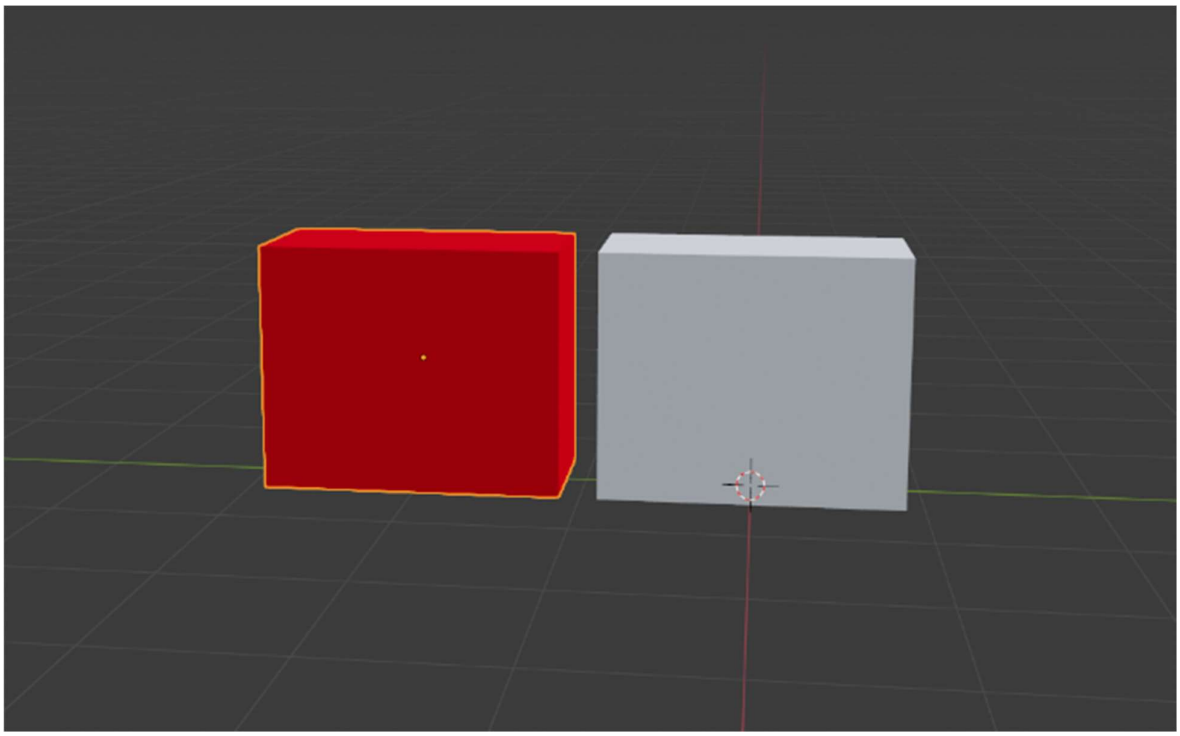


Rysunek 10. Wygląd toru po dodaniu do Plane modyfikatorów Array i Curve
Źródło: opracowanie własne wygenerowane za pomocą Blender.

Aby móc wykorzystać te układy torów w silniku Unity, musieliśmy wyeksportować je w formacie FBX. W tym celu należy przejść do *File –> Export* w programie Blender, a następnie wybrać odpowiedni format pliku (czyli FBX) oraz odpowiednie opcje.

1.2 Barierki

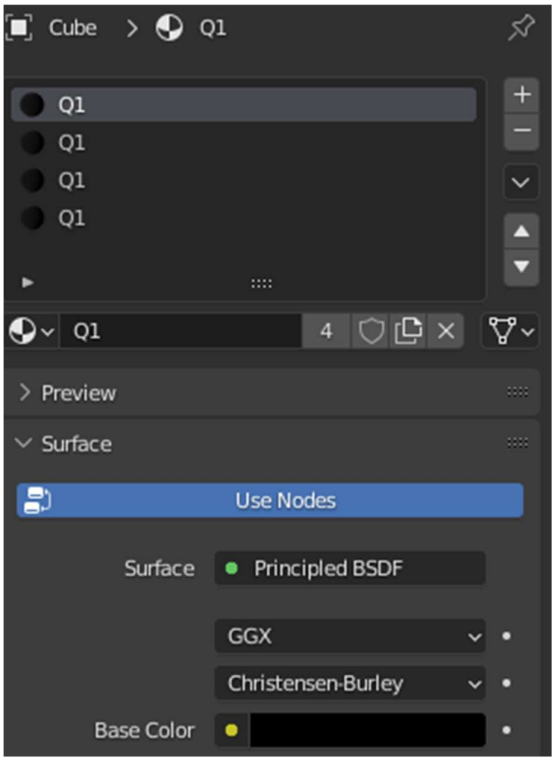
Do stworzenia barierki został wykorzystany obiekt *Cube* typu *mesh*. Po dopasowaniu jego rozmiarów został on zduplikowany. Tym sposobem pokolorowaliśmy jeden *Cube* na czerwono a drugi na biało (patrz *Rysunek 11*).



Rysunek 11. Zduplikowany czerwony Cube oraz bazowy biały Cube
Źródło: opracowanie własne wygenerowane za pomocą Blender.

Następnie przesunęliśmy jeden *Cube* do drugiego, tak aby lekko przenikały przez siebie. Zaznaczyliśmy oba obiekty trzymając klawisz Shift i połączyliśmy je w jeden kombinacją Ctrl+J. Następnie zduplikowaliśmy nowo powstały *mesh* i przesunęliśmy go tak, aby stykał się z duplikatem. Ponownie połączyliśmy je w całość poprzednio użytą kombinacją klawiszy. W tym momencie zostało skończone tworzenie modelu, który wyznacza granice toru

Czarna barierka od barierki biało-czerwonej odróżnia się tylko kolorem. Zatem użyliśmy ten sam model barierki biało-czerwonej zmieniając tylko jej kolor (patrz *Rysunek 12*).

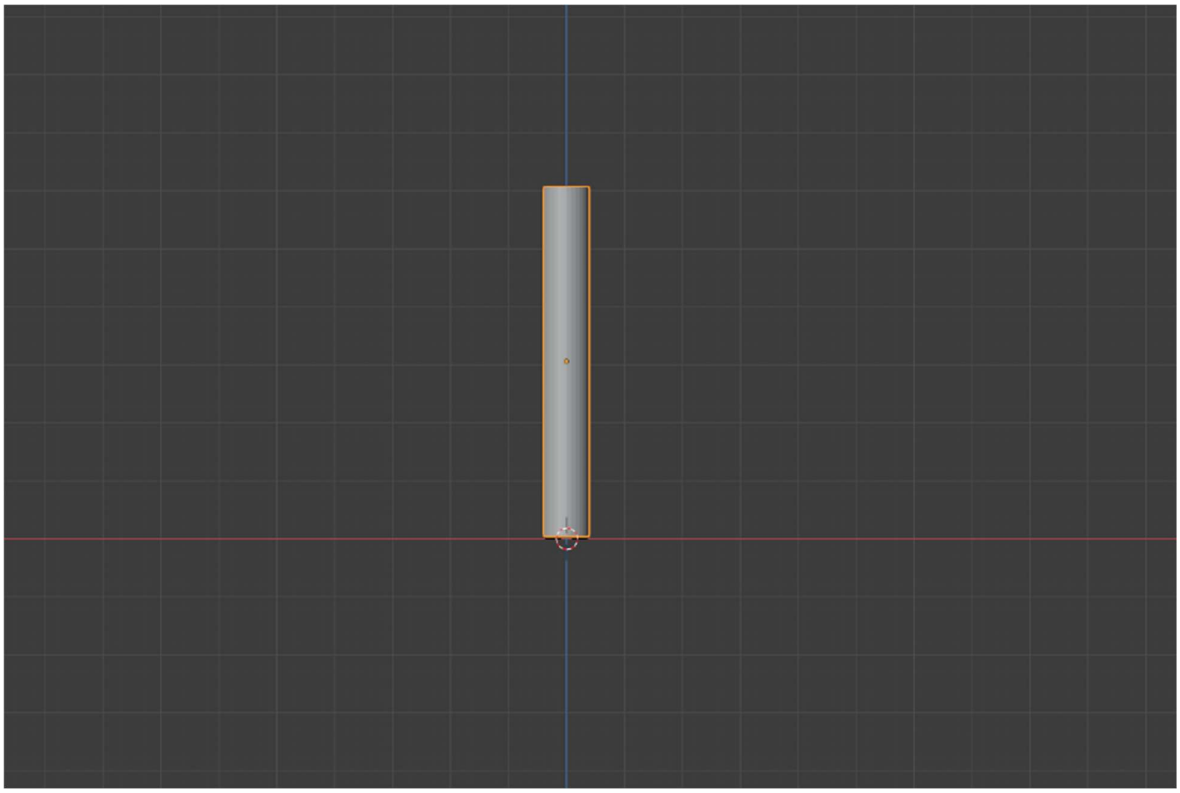


Rysunek 12. Zmiana koloru Cube'ów na czarny

Źródło: opracowanie własne wygenerowane za pomocą Blender.

1.3 Drzewo

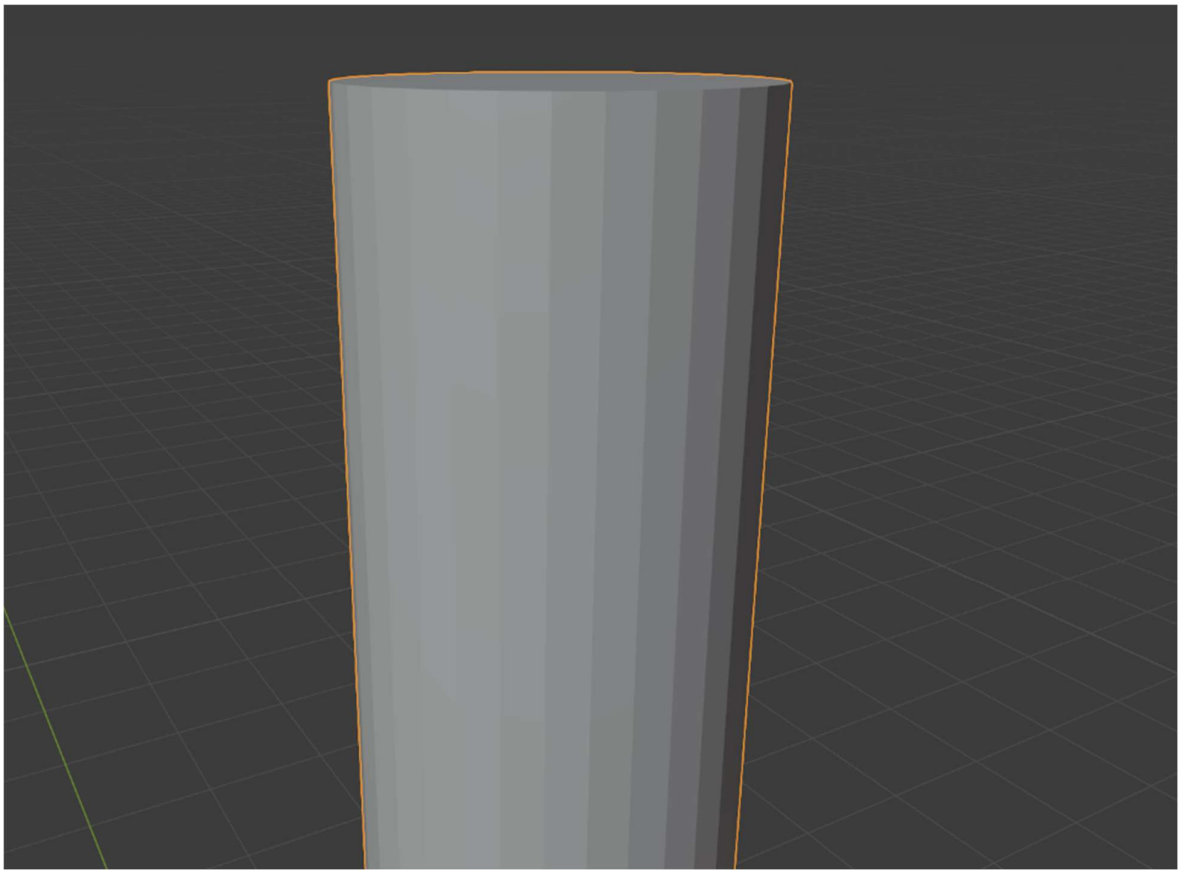
W celu stworzenia pnia drzewa dodaliśmy obiekt *Cylinder* typu *mesh*. Następnie zostały zmienione jego rozmiary, tj. zwiększono jego wysokość po osi Z oraz zmniejszono jego grubość, aby zachować odpowiednie proporcje (patrz *Rysunek 13*).



Rysunek 13. Pień drzewa stworzony z mesh'a Cylinder

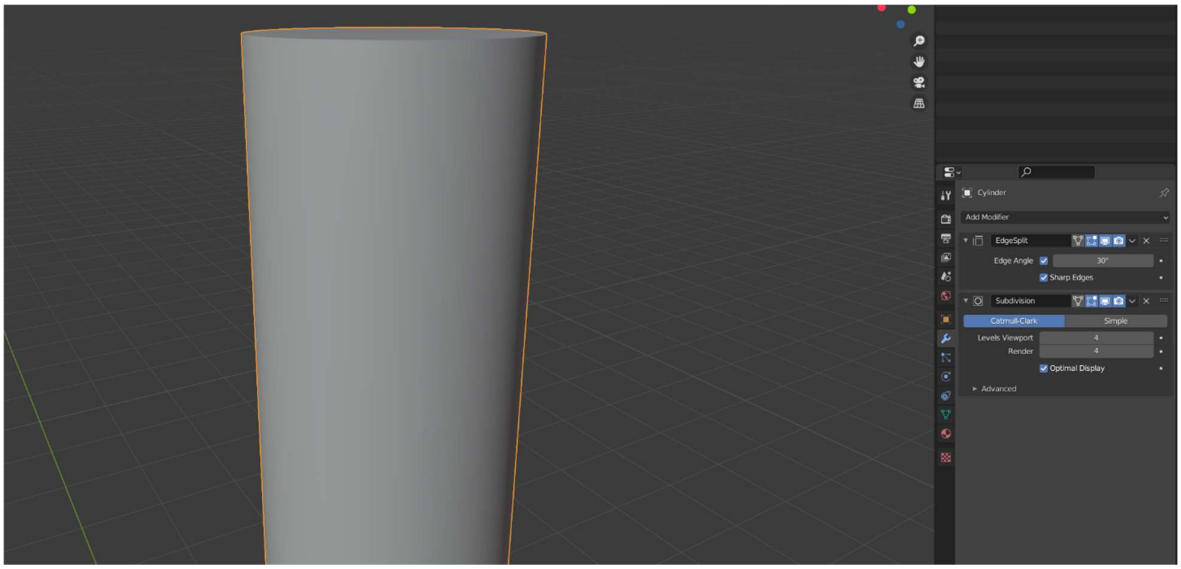
Źródło: opracowanie własne wygenerowane za pomocą Blender.

Pierwotnie *Cylinder* miał widoczne ostre krawędzie wokół zewnętrznej bocznej ścianki (patrz *Rysunek 14*). Aby wygładzić ją dodaliśmy modyfikatory *EdgeSplit* oraz *Subdivision* oraz zmieniliśmy w nich parametry tak, aby wygładzić krawędzie. Ostatecznie otrzymaliśmy wygładzony *Cylinder* (patrz *Rysunek 15*).



Rysunek 14. Cylinder przed wygładzeniem ścianki bocznej

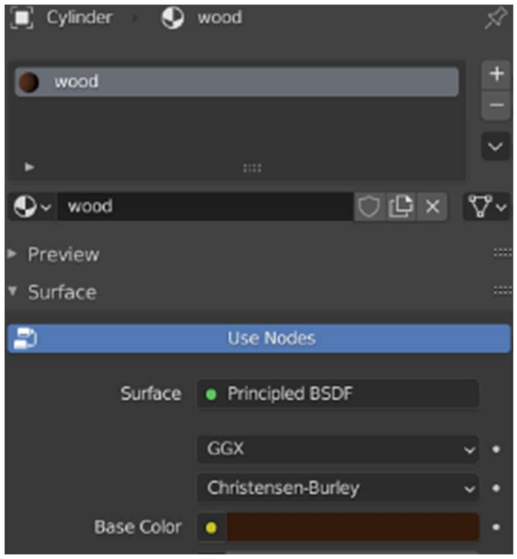
Źródło: opracowanie własne wygenerowane za pomocą Blender.



Rysunek 15. Cylinder po wygładzeniu oraz parametry dla modyfikatorów

Źródło: opracowanie własne wygenerowane za pomocą Blender.

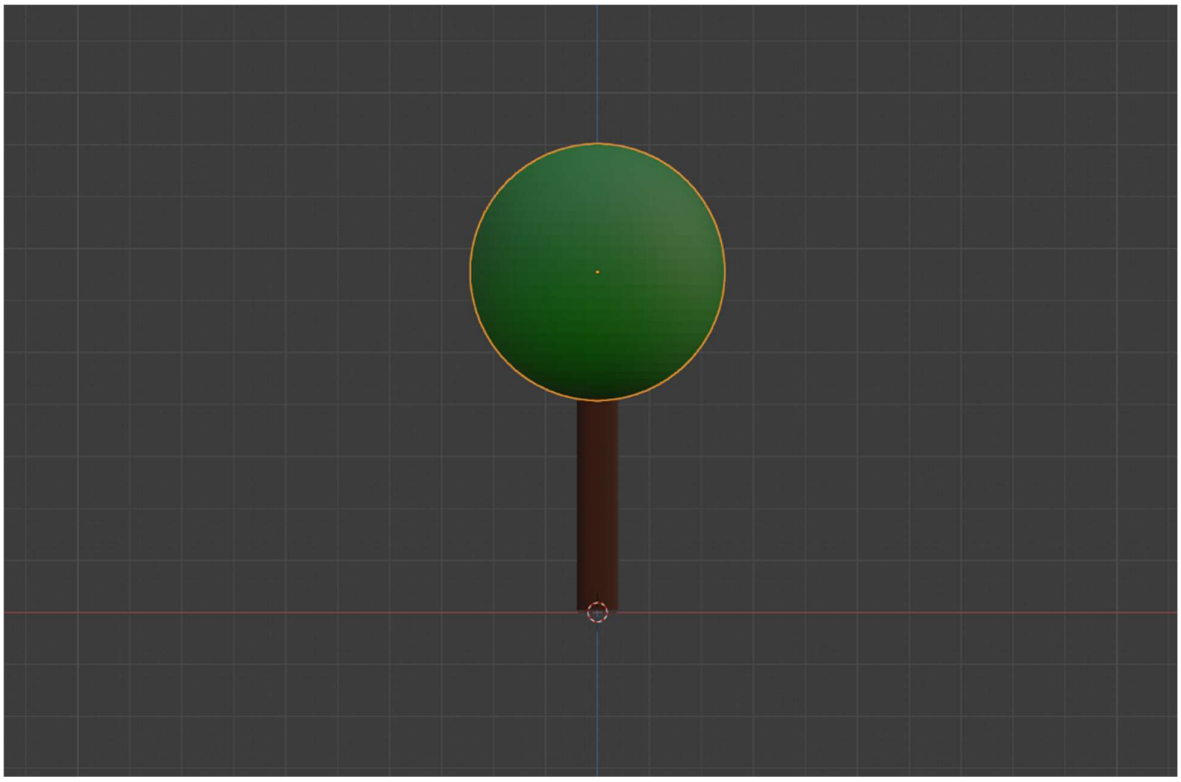
Po wykonaniu poprzednich czynności w zakładce *Material Properties* dodajemy materiał, którym zmienimy kolor pnia na brązowy (patrz *Rysunek 16*). Teraz zmieniamy kolor na brązowy.



Rysunek 16. Zmiana koloru pnia drzewa na brązowy

Źródło: opracowanie własne wygenerowane za pomocą Blender.

Do stworzenia korony drzewa wykorzystaliśmy obiekt typu *mesh* o nazwie *Sphere*. Dodaliśmy do niej modyfikator *Subdivision Surface*, aby lekko ją wygładzić. Zmieniliśmy kolor na zielony podobnie jak w przypadku pnia drzewa. Następnie ustawiliśmy pozycję *Sphere* tak, aby była na górze *mesh'a Cylinder* oraz zwiększyliśmy ją aby zachować odpowiednie proporcje (patrz *Rysunek 17*).



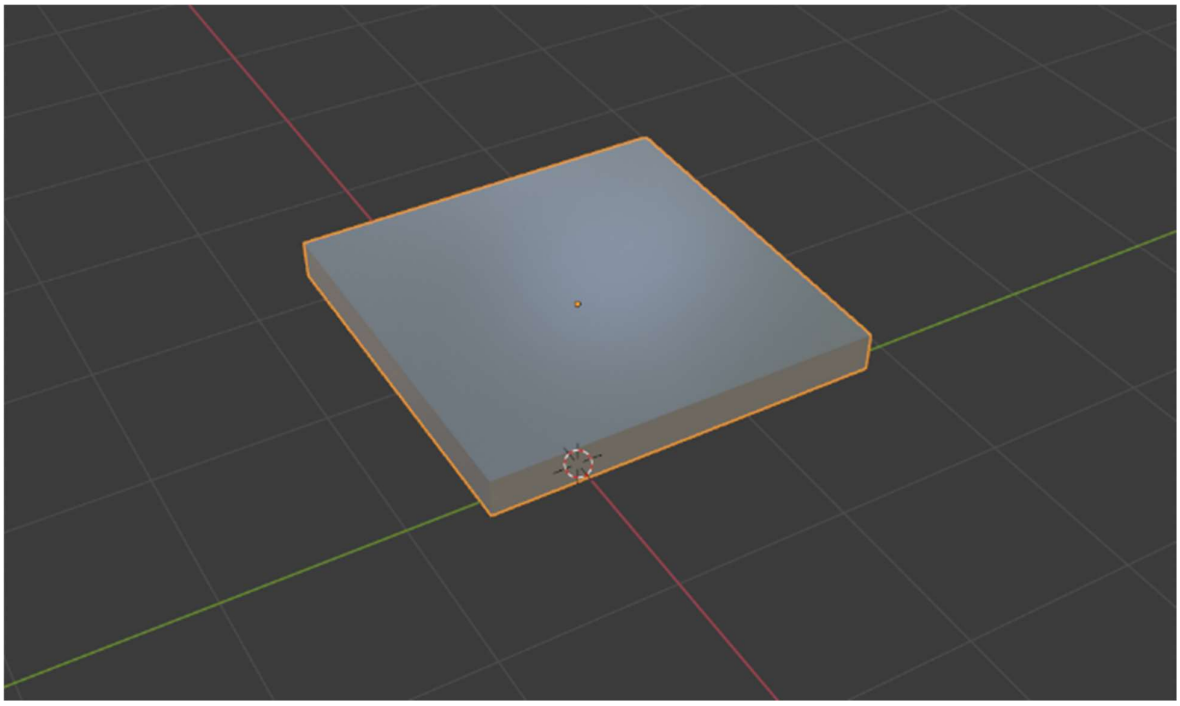
Rysunek 17. Wygląd drzewa po dostosowaniu wielkości Sphere
Źródło: opracowanie własne wygenerowane za pomocą Blender.

Aby móc wykorzystać model drzewa w silniku Unity został on wyeksportowany w formacie FBX.

1.4 Znaki

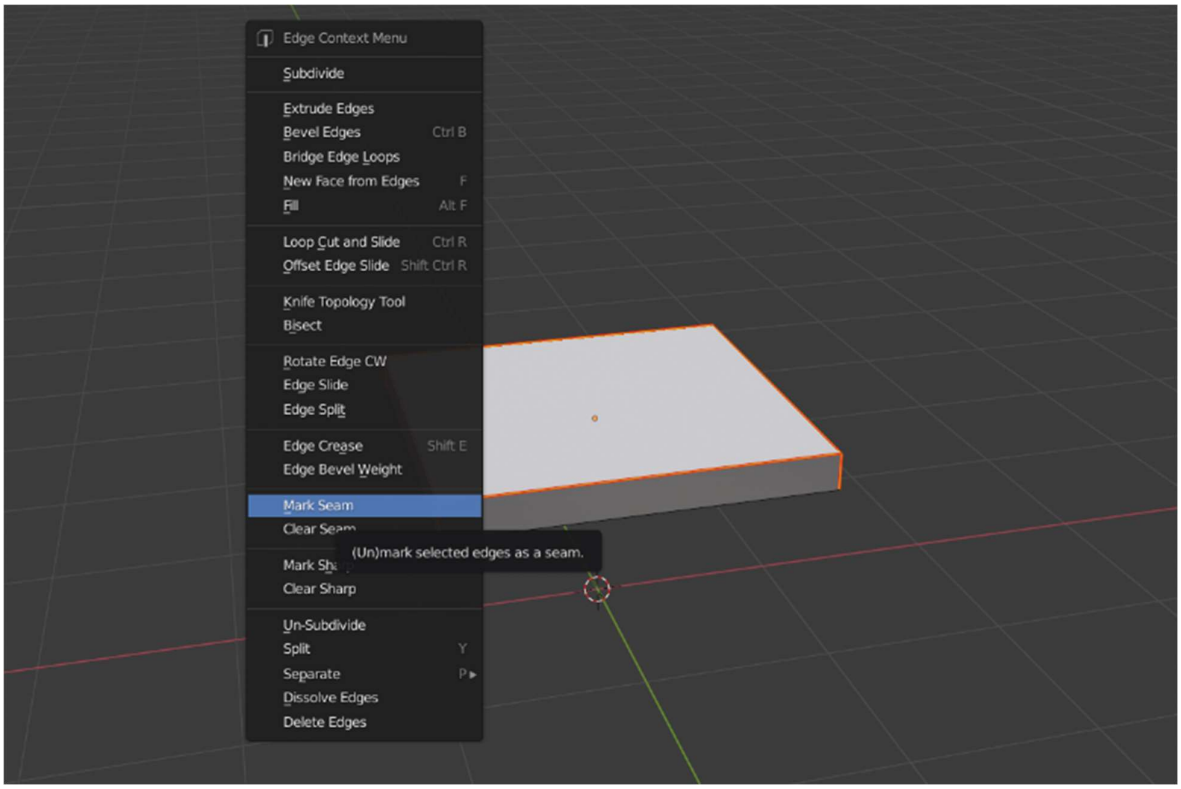
Do oznaczenia na torze linii startu, punktu kontrolnego i zakrętów zostały wykorzystane znaki, które składają się z tekstur stworzonych w programie GIMP oraz modeli w Blenderze. Tekstury dla znaków punktu kontrolnego i zakrętów zostały utworzone tą samą techniką. Mają wielkość 400x400 pikseli. Znak wskazujący linię startu ma na górze teksturę wielkości 800x200 pikseli (patrz *Rysunek 21*). W obu przypadkach tekstury zostały wyeksportowane w formacie PNG. Tekstury te znajdują się w archiwum *Zasoby_gimp.zip*.

Do stworzenia modelu 3D znaku, wykorzystanego potem jako checkpoint lub oznaczenie zakrętu, w Blenderze dodaliśmy na scenę obiekt *Cube* typu *mesh*, po czym zmieniliśmy jego wielkość względem osi Z i nadaliśmy mu kolor (patrz *Rysunek 18*).



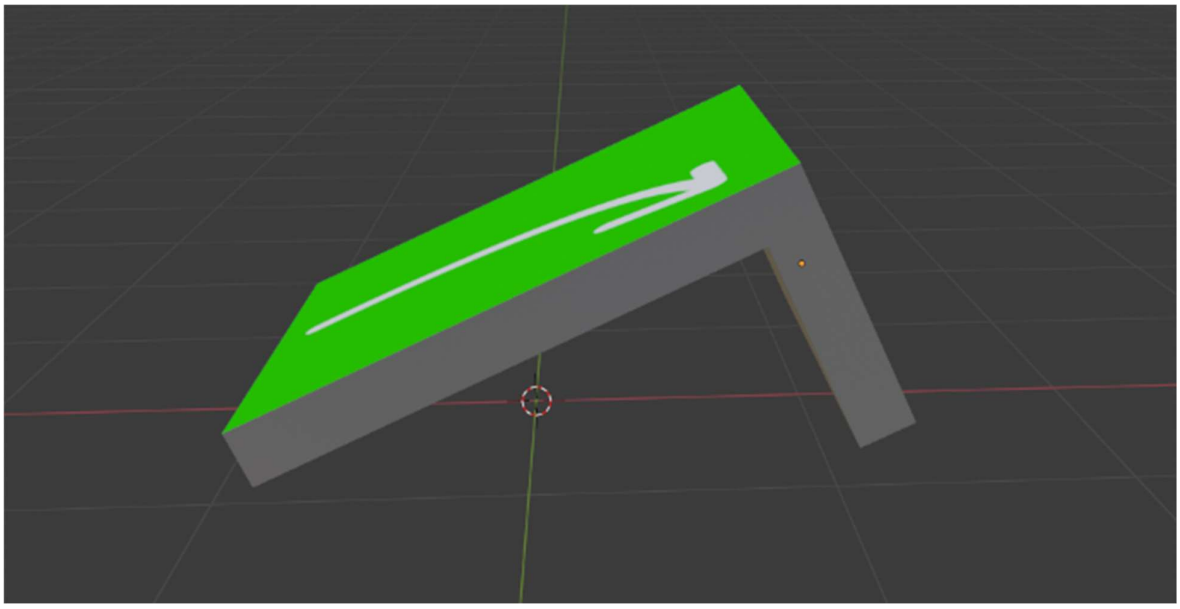
Rysunek 18. Przednia część znaku po przeskalowaniu i zmianie koloru
Źródło: opracowanie własne wygenerowane za pomocą Blender.

Następnie, aby móc poprawnie nałożyć teksturę na front tego *mesh'a*, zaznaczyliśmy górną ściankę, w zakładce *UV* wybraliśmy opcję *Unwrap* oraz dodaliśmy w zakładce *Material Properties* nowy slot przypisany do tej ścianki. Następnie zaznaczyliśmy potrzebne nam krawędzie i w zakładce *Edge* wybraliśmy *Mark Seam* (patrz *Rysunek 19*).



Rysunek 19. Zaznaczenie odpowiednich krawędzi i wybranie opcji „Mark Seam"
Źródło: opracowanie własne wygenerowane za pomocą Blender.

Dla utworzonego wcześniej w materiałach slotu zamiast koloru wybraliśmy obrazek i wskazaliśmy na wybraną teksturę. Następnie w *UV Editing* rozciągnęliśmy kwadrat na cały obrazek, aby cała jego powierzchnia była widoczna na górnej ściance modelu. Następnie, aby zrobić podpórkę pod znak, dodaliśmy kolejny *mesh Cube* i zmieniliśmy jego wielkość oraz kolor. Ustawiliśmy go z tyłu naszego wcześniejszego *mesh'a* (patrz *Rysunek 20*) i scaliliśmy opcją *Join*.



Rysunek 20. Projekt znaku wskazującego prawy lekki zakręt

Źródło: opracowanie własne wygenerowane za pomocą Blender.

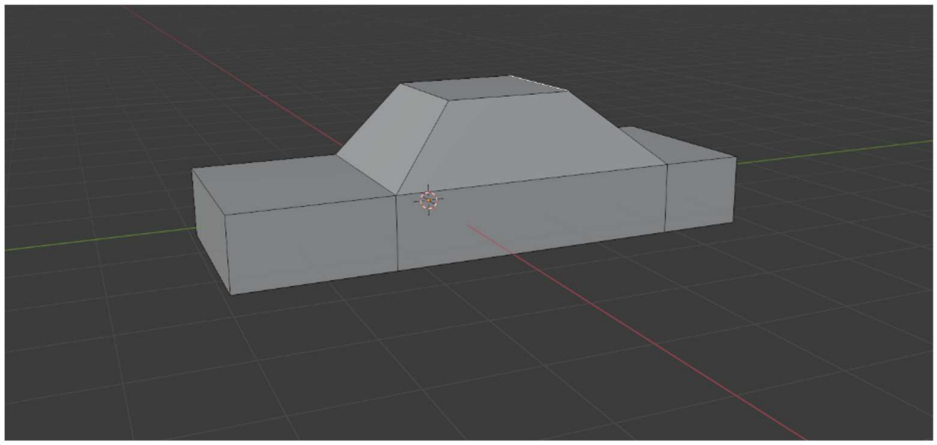
Aby nie powtarzać od nowa wcześniej opisanych czynności dla każdego znaku zakrętu i punktu kontrolnego podmieniliśmy tylko obrazy w slocie materiału i wyeksportowaliśmy pliki o różnych nazwach w formacie FBX.

Znak wskazujący linię startu został oparty o całkowicie inny projekt 3D, bo musi pod nim zmieścić się cały samochód. Model znaku start składa się z dwóch takich samych *mesh'ów* będących nogami znaku oraz trzeciego *mesh'a*, który na nich leży. Na górną ściankę ostatniego *mesh'a* nałożona została tekstura z napisem „START" w podobny sposób co w znakach wskazujących zakręt (patrz *Rysunek 21*). Ponownie wszystkie obiekty składające się na ten znak zostały scalone i wyeksportowane w formacie FBX.

Rysunek 21. Projekt znaku wskazującego linię startu
Źródło: opracowanie własne wygenerowane za pomocą Blender.

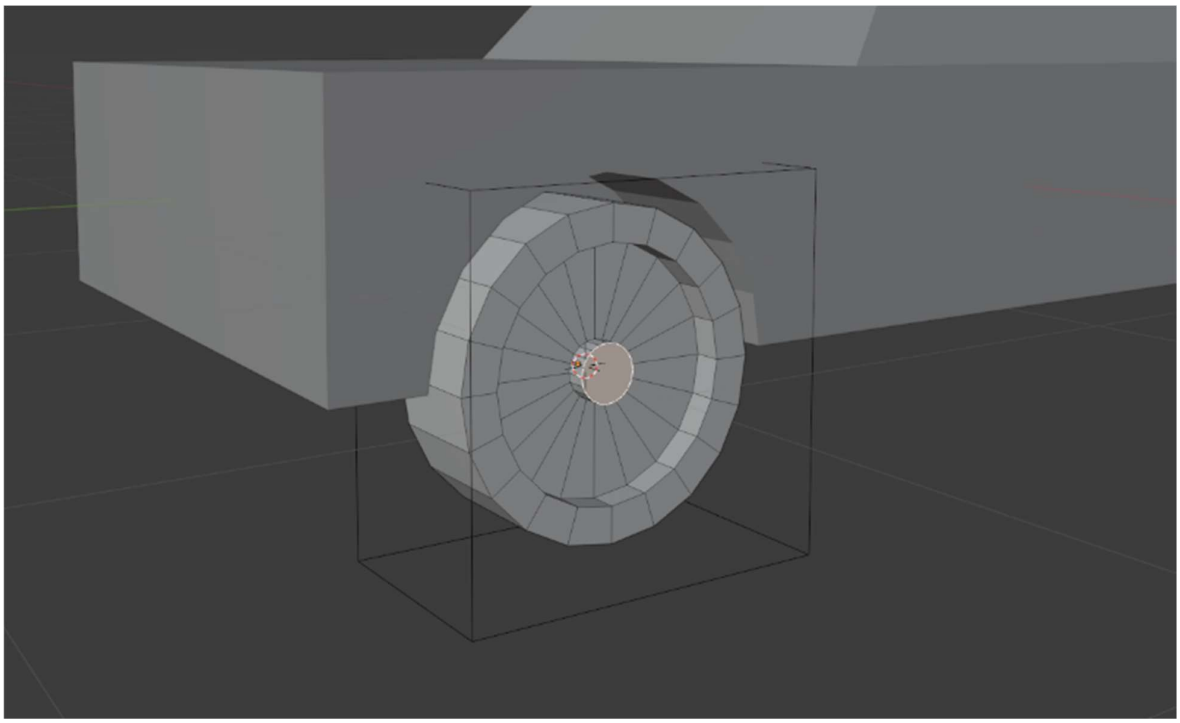
1.5 Samochód

Na początku podzieliliśmy wcześniej przeskalowany obiekt *Cube* na dwie części za pomocą operacji *Cut*, po czym zrobiliśmy *Extrude* na górnej oraz bocznej ściance większej części, aby nadać modelowi kształt samochodu. Ponownie zaznaczyliśmy górną ściankę i zmniejszyliśmy ją (patrz *Rysunek 22*).



Rysunek 22. Karoseria pojazdu po podzieleniu i przeskalowaniu Cube
Źródło: opracowanie własne wygenerowane za pomocą Blender.

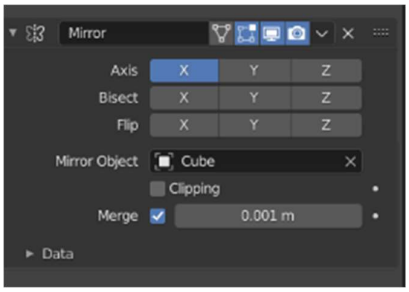
Dodaliśmy na scenę obiekt *Cylinder* typu *mesh* i zmniejszyliśmy go, po czym przesunęliśmy w miejsce, gdzie powinno znajdować się przednie koło. Zaznaczyliśmy najpierw *Cylinder*, potem trzymając klawisz Shift, karoserię i w menu *Object* wybraliśmy zakładkę *Bool Tool*, gdzie została wykonana operacja *Brush Boolean Difference*. Tym sposobem otrzymaliśmy w karoserii wycięte nadkole. Następnie dodaliśmy na scenę kolejny *Cylinder*, zmniejszyliśmy go i umieściliśmy w wyciętym nadkolu. Zaznaczyliśmy zewnętrzną ściankę koła i wykonaliśmy operację *Inset*, aby utworzyć oddzielną mniejszą ściankę na felgę. Następnie na ściance felgi zrobiliśmy *Extrude* do wnętrza koła. Jeszcze raz wykonaliśmy *Inset*, żeby utworzyć środek felgi, który następnie został wyciągnięty lekko na zewnątrz poleceniem *Extrude* (patrz *Rysunek 23*).



Rysunek 23. Wygląd modelu koła

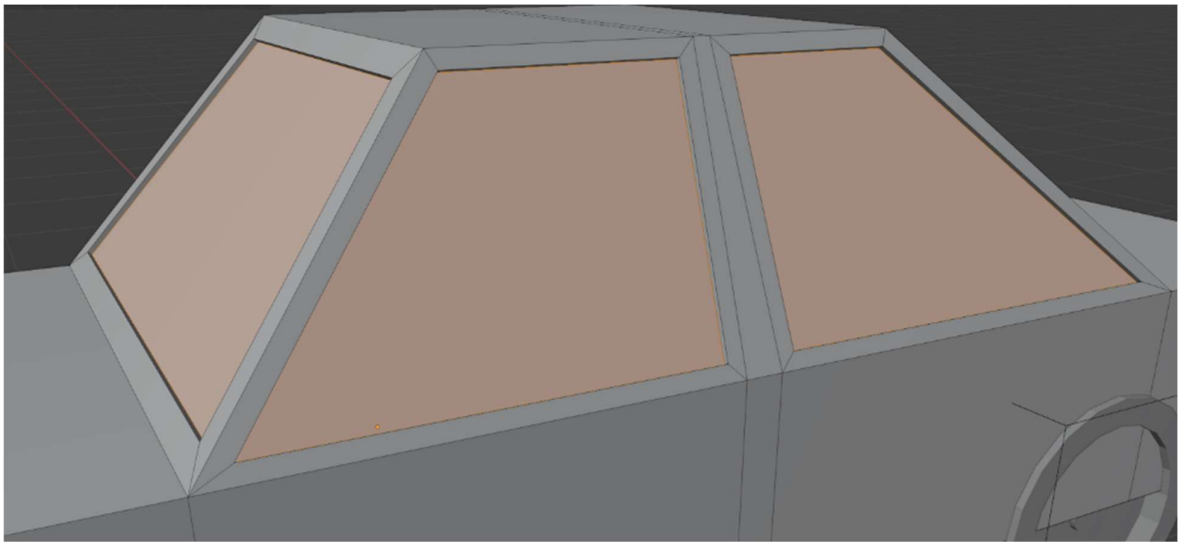
Źródło: opracowanie własne wygenerowane za pomocą Blender.

Aby otrzymać takie samo koło i nadkole po drugiej stronie samochodu, musieliśmy dodać odbicie lustrzane dla wycięcia i modelu koła. Dla obiektów koła i wycięcia w nadwoziu dodaliśmy modyfikator *Mirror* zaznaczając w polu *Mirror Object* karoserię (patrz *Rysunek 24*).



Rysunek 24. Modyfikator „Mirror" dodający koło i nadkole po drugiej stronie nadwozia
Źródło: opracowanie własne wygenerowane za pomocą Blender.

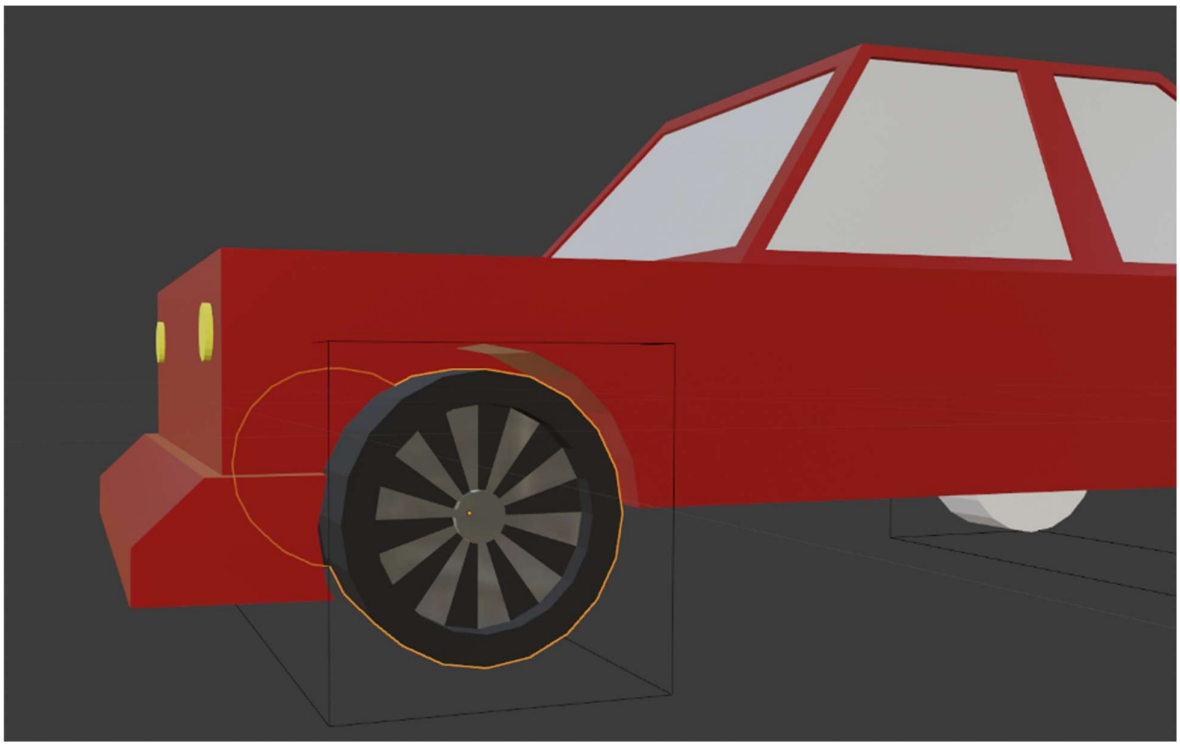
Aby zrobić takie same koła i nadkola z tyłu nadwozia, zaznaczyliśmy koło i obiekt wycinający nadkole, zduplikowaliśmy je po czym przesunęliśmy na tylną część karoserii. Wycięcia z tyłu karoserii nie zostały stworzone, więc wybraliśmy obiekt odpowiedzialny za nadkole oraz trzymając klawisz Shift zaznaczyliśmy dodatkowo nadwozie i tak jak wcześniej wykonaliśmy operację *Brush Boolean Difference*. Wykonaliśmy działanie *Apply Scale* dla nadwozia, żeby ówczesne wymiary stały się podstawowymi, co nie będzie powodować problemów przy tworzeniu szyb i innych elementów pojazdu. Do stworzenia szyb w samochodzie musieliśmy podzielić model ponownie za pomocą *Cut* w taki sposób, aby na środku wyciągniętego wcześniej dachu otrzymać słupki po bokach. Zaznaczyliśmy wszystkie ścianki odpowiedzialne za szyby i zmniejszyliśmy je operacją *Inset*. To samo zrobiliśmy dla ścianek odpowiedzialnych za przednią i tylną szybę. Zaznaczyliśmy wszystkie ścianki szyb, nacisnęliśmy kombinację Shift+E, wybraliśmy opcję *Extrude Faces Along Normals* i przysunęliśmy szyby do wewnątrz (patrz *Rysunek 25*).



Rysunek 25. Szyby stworzone operacjami „Cut", „Inset" i „Extrude Faces Along Normals"
Źródło: opracowanie własne wygenerowane za pomocą Blender.

Model lampy przedniej został stworzony poprzez dodanie na scenę obiekt Cylinde typu *mesh*, przeskalowanie go i przesunięcie na przód samochodu. Dodaliśmy do obiektu lampy modyfikator *Mirror* i otrzymaliśmy taką samą lampę przednią po drugiej stronie. Podobnie wykonaliśmy lampy tylne, ale zamiast *mesh'a Cylinder* wybraliśmy *Cube*, żeby z tyłu stworzyć lampy na kształt prostopadłościanu. Aby stworzyć przedni zderzak, podzieliliśmy model nadwozia w taki sposób, aby otrzymać małe części pionowe i poziome na przodzie modelu. Następnie zaznaczyliśmy nowopowstałe ścianki po bokach oraz z przodu i wykonaliśmy *Extrude*, a następnie skalowanie.

W tym momencie pokolorowaliśmy wyżej utworzone obiekty. Dla każdego z nich w zakładce *Material Properties* dodaliśmy nowy slot na kolor i zmieniliśmy go. W przypadku nadwozia, aby otrzymać białe szyby, dodaliśmy kolejny slot na biały kolor, zaznaczyliśmy ścianki szyb i przypisaliśmy im kolor przyciskiem *Assign*. Modele kół pokolorowaliśmy na czarno, ale żeby zmienić felgi na srebrne dodaliśmy slot dla koloru srebrnego i zaznaczyliśmy ramiona felg w modelu koła tak jak w przypadku szyb w nadwoziu i przypisaliśmy je do odpowiedniego koloru. To samo zrobiliśmy z tylnym kołem. Jako że koła mają modyfikator *Mirror*, to te po prawej stronie wyglądają tak samo.



Rysunek 26. Wygląd kół i przodu pojazdu po zmianie kolorów
Źródło: opracowanie własne wygenerowane za pomocą Blender.

Ostatnim elementem, który został stworzony jest spoiler z tyłu. Dodaliśmy na scenę obiekt *Cube* typu *mesh*, po czym przeskalowaliśmy go do małych rozmiarów i ustawiliśmy z lewej strony

bagażnika. Dodaliśmy mu modyfikator *Mirror*, aby otrzymać taką samą nóżkę spoilera po drugiej stronie. Ponownie dodaliśmy mesh *Cube* i przeskalowaliśmy go w taki sposób, aby wyglądał jak płat spoilera, po czym przesunęliśmy go na górę nóżek i zmieniliśmy kąt względem osi X. Następnie zmieniliśmy kolor podstawy spoilera na czarny, a płata na żółty. Zaznaczyliśmy wszystkie obiekty na scenie i w menu *Object* wybraliśmy *Join* co złożyło wszystkie elementy pojazdu w jeden obiekt. Taki oto model pojazdu (patrz *Rysunek 27*) wyeksportowaliśmy w formacie FBX.

Rysunek 27. Ostateczny wygląd modelu samochodu

Źródło: opracowanie własne wygenerowane za pomocą Blender.

Rozdział II: Implementacja logiki gry

W tym rozdziale zostaną omówione wszystkie skrypty potrzebne do zaprogramowania działania gry. Innymi słowy, są to skrypty stanowiące implementację logiki gry.

2.1 Skrypt *CarBehaviour.cs*

Plik *CarBehaviour.cs* to skrypt odpowiedzialny za ruch samochodu. W tym pliku została zdefiniowana klasa *CarBehaviour* (patrz *Listing 1*).

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class CarBehaviour : MonoBehaviour
6   {
7       Quaternion targetRotation;
8
9       [SerializeField] public float turnSpeed = 120;
10      [SerializeField] public float accel = 2000;
11
12      Rigidbody _rigidBody;
13
14      private void Start()...
18
19      void Update()...
24
25      private void SetRotationPoint()...
47
48      private void FixedUpdate()...
59  }
60
```

Listing 1. Klasa CarBehaviour

Źródło: opracowanie własne autora.

Jak widzimy zostały zdefiniowane pola *turnSpeed* oraz *accel* typu *float*, których wartości, dzięki atrybutowi *[SerializeField]*, możemy zmieniać w silniku gry Unity bez potrzeby przełączania się do edytora kodu. Zdeklarowane zostały również pole *targetRotation* jako obiekt klasy *Quaternion* (patrz [5]) oraz pole *_rigidBody* klasy *Rigidbody* (patrz [6]). Dodatkowo zostały zaimplementowane metody *Start()*, *Update()*, *SetRotationPoint()* i *FixedUpdate()* (patrz [4]). Wszystkie te metody należą do klasy nadrzędnej/rodzica *MonoBehaviour* (patrz [4]) nie wliczając w to *SetRotationPoint()*, która jest metodą stworzoną przez autora.

Funkcja *Start()* wykonuje się tylko na początku, gdy załaduje się scena gry. Jej implementację przedstawia *Listing 2*. W tym przypadku do obiektu *_rigidBody* przypisywany jest

komponent *RigidBody* dodany do *GameObject* o nazwie *driftCarV2* (patrz [20]). Obiekt *driftCarV2* reprezentuje właśnie nasz samochód.

```
private void Start()
{
    _rigidBody = GetComponent<Rigidbody>();
}
```

Listing 2. Metoda Start() klasy CarBehaviour
Źródło: opracowanie własne autora.

Teraz omówimy implementację metody *SetRotationPoint()* (patrz *Listing 3*).

```
private void SetRotationPoint()
{
    //tworzenie promienia od origin kamery do kursora myszki
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    //tworzenie płaszczyzny
    Plane plane = new Plane(Vector3.up, Vector3.zero);

    float distance;


    if(plane.Raycast(ray, out distance))
    {
        //punkt przeciecia promiena z płaszczyzną
        Vector3 target = ray.GetPoint(distance);
        //wektor, w kierunku którego chcemy obrocic pojazd
        Vector3 direction = target - transform.position;
        //argument wektora direction na plaszczyźnie Oxz od razu
        //zamieniony z radianow na stopnie
        float rotationAngle = Mathf.Atan2(direction.x, direction.z) * Mathf.Rad2Deg;
        //utworzenie z rotationAngle kata obrotu wzgledem osi y
        targetRotation = Quaternion.Euler(0, rotationAngle, 0);
    }
}
```

Listing 3. Metoda SetRotationPoint() klasy CarBehaviour
Źródło: opracowanie własne autora.

Na początku określenia tej metody mamy definicje obiektu *ray* struktury *Ray* (patrz [7]), *plane* struktury *Plane* (patrz [8]) oraz *distance* typu *float*. Obiekt *ray* dzięki metodzie *ScreenPointToRay(Input. mousePosition)* przechowuje stworzony promień mający początek w pozycji kamery i skierowany w kierunku aktualnej pozycji kursora myszki (patrz [9]). Obiekt *plane* przedstawia płaszczyznę przechodzącą przez punkt (0,0,0) i o wektorze normalnym (0,1,0) (patrz [10]). Jest ona równoległa do podłoża mapy. Instrukcja warunkowa *if(plane.Raycast(ray, out distance))* wykona się tylko wtedy, gdy *plane* nie będzie równoległy do promienia reprezentowanego obiektem *ray*. Zmienna *distance*, którą zwraca funkcja *plane.Raycast()*, jest

długością odcinka zawartego w promieniu ray od położenia kamery do punkt przecięcia promienia *ray* z płaszczyzną *plane*. W bloku kodu tej instrukcji warunkowej tworzone są zmienne *target* i *direction* typu *Vector3* oraz *rotationAngle* typu *float*. Zmienna *target* przechowuje punkt przecięcia promienia *ray* z płaszczyzną *plane* jako wynik wykonania metody *GetPoint(distance)* (patrz [11]). Wektor *direction* wskazuje kierunek, w którym chcemy obrócić pojazd, a w zmiennej *rotationAngle* przechowuje kąt między osią X a osią Z lokalnego układu współrzędnych samochodu. Tu wykorzystujemy metodę *Mathf.Atan2()* (patrz [12]). Kąt jest od razu zamieniany z radianów na stopnie poprzez pomnożenie przez *Mathf.Rad2Deg*. Pole *targetRotation* klasy *CarBehaviour* (patrz *Listing 1*) dzięki wywołaniu *Quaternion.Euler(0, rotationAngle, 0)*, otrzymuje kąt obrotu względem osi Y o kąt pobrany ze zmiennej *rotationAngle* (patrz [13]).

Przejdźmy do omówienia metody *Update()* (patrz *Listing 4*).

```
void Update()
{
    SetRotationPoint();

}
```

Listing 4. Metoda Update() klasy CarBehaviour

Źródło: opracowanie własne autora.

Metoda *Update()* jest wykonywana co każdą klatkę w grze. W tym przypadku co każdą klatkę jest w niej wywoływana wyżej opisana metoda *SetRotationPoint()*, patrz *Listing 3*.

Przejdźmy do metody *FixedUpdate()*, której definicję widzimy w *Listingu 5*.

```
private void FixedUpdate()
{
    //zmienna pobierajaca predkosc pojazdu podzielona przez 1000
    //aby wykorzystana przy kalkulacji predkosci
    //obrotu za szybko nie przekrecala samochodu
    float speed = _rigidBody.velocity.magnitude / 1000;
    //if LPM->1, else if PPM->-1, else 0 * przyspieszenie * update silnika gry
    float accelInput = accel *
        (Input.GetMouseButton(0) ? 1 : Input.GetMouseButton(1) ? -1 : 0)
        *Time.fixedDeltaTime;
    //przesuniecie pojazdu do przodu
    _rigidBody.AddRelativeForce(Vector3.forward * accelInput);
    //Lerp czyli obrocenie pojazdu od jego wlasnego kata do kata
    //targetRotation o predkosci
    //turnSpeed * update silnika * Clamp od -1 do 1 na zmiennej speed
    //zeby samochod obracal sie tylko gdy jedzie
    transform.rotation = Quaternion.Lerp(transform.rotation, targetRotation,
        turnSpeed * Mathf.Clamp(speed, -1, 1) * Time.fixedDeltaTime);
}
```

Listing 5. Metoda FixedUpdate() klasy CarBehaviour

Źródło: opracowanie własne autora.

Metoda *FixedUpdate()* wykonuje się co każdy wewnętrzny update silnika fizyki Unity, standardowo 50 razy w ciągu sekundy. Dzięki tej funkcji możemy napisać kod, który będzie wpływał na fizyczne właściwości pojazdu i nie będzie to powodować problemów z jego zachowaniem. W ciele tej funkcji tworzone są zmienne *speed* i *accelInput* typu *float*. Zmienna *speed* została stworzona po to, by wykorzystać ją przy zdefiniowaniu prędkości skrętu samochodu. Przypisana jest do niej długość wektora przyspieszenia podzielona przez 1000, aby zmniejszyć prędkość obrotu (patrz [14]). Zmienna *accelInput* będzie wykorzystywana do nadania przyspieszenia naszemu samochodowi. Składa się ona z przyspieszenia *accel* pomnożonego przez skróconą instrukcję warunkową, która w przypadku wciśnięcia lewego przycisku myszy nadaje przyspieszenie dodatnie, a przy wciśnięciu prawego przycisku myszy nadaje przyspieszenie ujemne. Jeżeli nic nie jest wciśnięte, to przyspieszenie jest pomnożone przez 0, co z automatu je zeruje. Wszystkie polecenia myszy są odczytywane dzięki metodzie *Input.GetMouseButton()*. Liczba 0 jako argument tej metody reprezentuje lewy przycisk myszy, a 1 reprezentuje prawy przycisk myszy (patrz [15]). Na koniec wynik jest pomnożony przez *Time.fixedDeltaTime* co wymusza odświeżanie tego przyspieszenia co każdą aktualizację fizyki silnika (patrz [16]). Wywołanie *_rigidBody.AddRelativeForce(Vector3.forward * accelInput)* nadaje naszemu samochodowi przyspieszenie wzdłuż wektora (0,0,1) pomnożonego przez wcześniej wyliczone przyspieszenie. Wartość pola *transform.rotation* powoduje zmianę kąta obrotu naszego samochodu (patrz [17]). Wywołanie *Quaterinon.Lerp(transform.rotation, targetRotation, turnSpeed * Mathf.Clamp(speed, -1, 1) * Time.fixedDeltaTime)* interpoluje obrót samochodu od kąta, który w danym momencie przybiera, do kąta *targetRotation* (patrz [18]). Prędkość interpolacji jest wyliczona poprzez pomnożenie *turnSpeed* przez *speed* ograniczą do wartości pomiędzy -1 i 1 za pomocą metody *Clamp* klasy *Mathf* (patrz [19]). Dodatkowo, podobnie jak wyżej, aktualizujemy wartość prędkości interpolacji co każdą aktualizację fizyki silnika poprzez pomnożenie jej przez *Time.fixedDeltaTime*.

2.2 Skrypt *CameraScript.cs*

Plik *CameraScript.cs* (patrz *Listing 6*) jest odpowiedzialny za zachowanie kamery podczas jazdy. Została w nim określona klasa *CameraScript* odziedziczona od klasy *MonoBehaviour*.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CameraScript : MonoBehaviour
{

    [SerializeField] Transform observable;
    [SerializeField] float aheadSpeed;
    [SerializeField] float followDamp;
    [SerializeField] float camHeight;

    Rigidbody _observableRigidBody;

    void Start()...


    void Update()...
}
```

Listing 6. Klasa CameraScript

Źródło: opracowanie własne autora.

Klasa *CameraScript* posiada metody *Start() oraz Update()*. Zawiera też pola *aheadSpeed*, *followDamp*, *camHeight* typu *float,* pole *observable,* które jest obiektem klasy *Transform,* oraz *_observableRigidBody* klasy *RigidBody*. Cztery pierwsze pola możemy zmieniać (eksperymentować z nimi) w silniku gry dzięki atrybutowi *[SerializeField]*. W szczególności, jako *observable* ustawiliśmy bieżącą właściwość *Transform* naszego samochodu (przypomnijmy jest to *GameObject* o nazwie *driftCarV2*), patrz *Rysunek 45*. Przejdźmy do implementacji metody *Start()* (patrz *Listing 7)*.

```
void Start()
{
    _observableRigidBody = observable.GetComponent<Rigidbody>();
}
```

Listing 7. Metoda Start() klasy CameraScript

Źródło: opracowanie własne autora.

Metoda *Start()* dzięki wywołaniu *observable.GetComponent<Rigidbody>()* ładuje do obiektu *_observableRigidBody* komponent *Rigidbody* obiektu *observable*, do którego – jak wcześniej zostało powiedziane – jest przypisany nasz samochód. Omówmy jeszcze ostatnią metodę w klasie *CameraScript* (patrz *Listing 8*), czyli metodę *Update()*.

```
void Update()
{
    //Jeżeli nie ma nic do obserwowania to nic nie rób
    if (observable == null)
        return;
    //nowa pozycja kamery podniesiona o wektor (0,1,0) *
    //camHeight oraz zmiana jej pozycji przy pomocy predkosci
    //obiektu * aheadSpeed
    Vector3 targetPosition = observable.position + Vector3.up * camHeight +
        _observableRigidBody.velocity * aheadSpeed;
    //zmiana pozycji kamery z poprzedniej na nowa targetPosition
    //co kazda klatke razy followDamp
    transform.position = Vector3.Lerp(transform.position, targetPosition,
        followDamp * Time.deltaTime);
}
```

Listing 8. Metoda Update() klasy CameraScript

Źródło: opracowanie własne autora.

Wszystko co jest zawarte w ciele metody *Update()* jest odpowiedzialne za przesuwanie kamery co każdą klatkę w grze. Instrukcja warunkowa kończy działanie funkcji, gdy do obiektu *observable* nie ma przypisanego samochodu, za którym kamera powinna podążać. Obiekt *targetPositon* struktury *Vector3* otrzymuje wartość, która będzie potem wykorzystana przy zmianie pozycji kamery w dalszej części kodu. Wartość ta jest wyliczana poprzez pobranie aktualnej pozycji samochodu i dodanie do niej wektora (0,1,0) pomnożonego przez zmienną *camHeight* odpowiedzialną za zmianę wysokości kamery (patrz [10]). Następnie zwiększamy wartość prędkości naszej kamery pomnożoną przez zmienną/współczynnik *aheadSpeed.* W ten sposób otrzymujemy wartość prędkości kamery lekko wyprzedzającą nasz samochód w kierunku jazdy, aby można było zobaczyć co znajduje się przed naszym pojazdem. Wywołanie *transform.positon = Vector3.Lerp(transform.position, targetPosition, followDamp * Time.deltaTime)* odpowiedzialne jest za zmianę pozycji kamery poprzez interpolację między dwoma pozycjami kamery: aktualnym i nowo wyliczoną w *targetPosition.* Zmienna *followDamp* odpowiada za wyhamowanie kamery, żeby nie wyprzedzała za szybko samochodu (patrz [21]). Bez takiej implementacji prędkości interpolacji pozycji kamery samochód znikałby poza nią, bo ta by była szybsza niż pojazd. Na koniec mnożymy tą prędkość przez *Time.deltaTime*, aby pozycja kamery zmieniała się co każdą klatkę (patrz [22]).

2.3 Skrypty odpowiedzialne za okrążenia

Zadaniem plików *HalfLapTriggerScript.cs* (patrz *Listing 9*), *LapTimeScript.cs*, *LapCompleteScript.cs* oraz *LoadLapTime.cs* jest zarządzanie czasem okrążeń, ilością pokonanych okrążeń oraz wyświetlaniem tych statystyk na ekranie gry. Wszystkie te pliki zostały stworzone

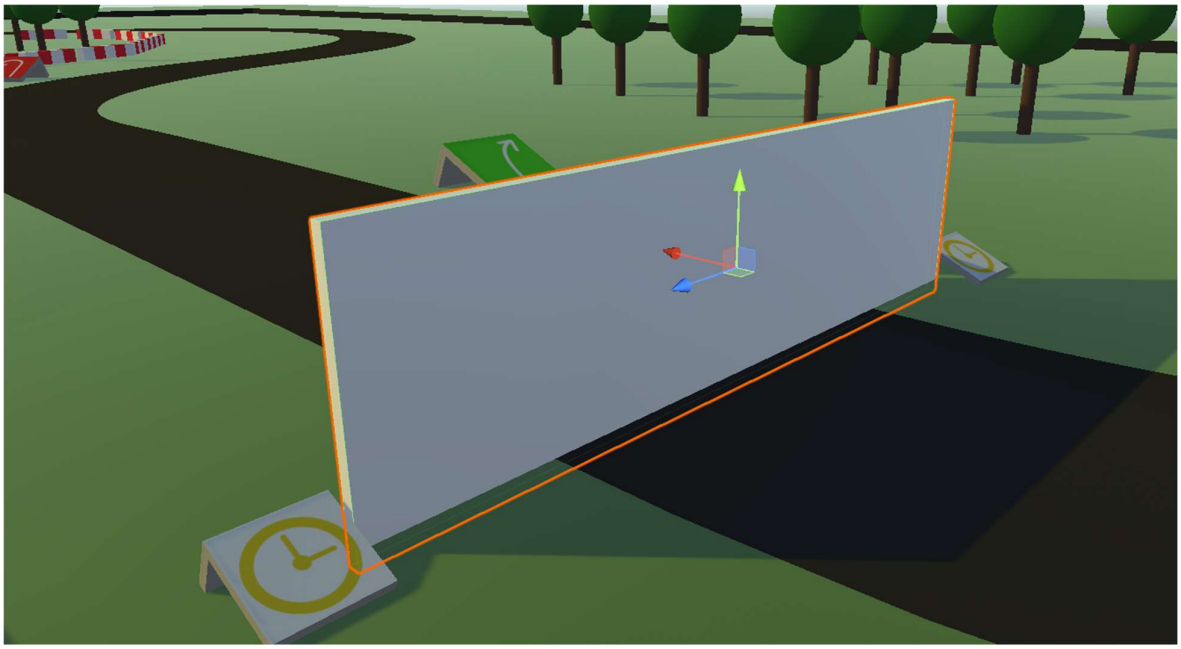
na podstawie [23, 24, 25, 26, 27]. Autor niniejszej pracy przeanalizował skrypty rozważane w tych źródłach i zaimplementował odpowiednie skrypty zgodnie z założeniami gry BZRG2022.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class HalfLapTriggerScript : MonoBehaviour
{
    public GameObject CompleteLapTrigger;
    public GameObject HalfLapTrigger;

    void OnTriggerEnter()
    {
        CompleteLapTrigger.SetActive(true);
        HalfLapTrigger.SetActive(false);
    }
}
```

Listing 9. Klasa HalfLapTriggerScript

Źródło: opracowanie autora na podstawie [23] z własnymi poprawkami.

Klasa *HalfLapTriggerScript* posiada pola *CompleteLapTrigger* oraz *HalfLapTrigger*, które są obiektami klasy *GameObject*. Metoda *OnTriggerEnter()* odpowiedzialna jest za przełączanie aktywności dla podpiętych pod zmienne obiektów na scenie. Na scenie dodajemy *GameObject* (przeskalowany sześcian, patrz *Rysunek 28*) w miejscu punktu kontrolnego oraz linii startu. Oczywiście, w przeciwieństwie do tego co widać poniżej na *Rysunku 28*, renderowanie siatki dla tych obiektów jest wyłączone, bo nie powinny one być widoczne.



Rysunek 28. GameObject Half Lap Trigger, który sprawdza zaliczenie punktu kontrolnego
Źródło: opracowanie własne wygenerowane za pomocą Unity.

Dla obiektu *HalfLapTrigger* podpinamy *GameObject* znajdujący się na linii punktu kontrolnego. Dla obiektu *CompleteLapTrigger* wskazujemy obiekt *Sart Trigger* (zob. *Rysunek 40*), który znajduje się na linii startu. Gdy dotkniemy naszym samochodem obiekt w punkcie kontrolnym, to zgodnie z *HalfLapTrigger.SetActive(false)*, wyłączamy go. W tym samym momencie wywołanie *CompleteLapTrigger.SetActive(true)* aktywuje obiekt na linii startu.

Przejdźmy do omówienia implementacji klasy *LapCompleteScript* (patrz *Listing 10*) zdefiniowanej w pliku *LapCompleteScript.cs*.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class LapCompleteScript : MonoBehaviour
{
    public GameObject LapCompleteTrigger;
    public GameObject HalfLapTrigger;

    public GameObject MinDisplay;
    public GameObject SecDisplay;
    public GameObject MilisecDisplay;

    public GameObject LapCounter;
    public static int LapsDone = 1;
    public static int LapsDoneFinish = 1;

    public float RawTime;



    void OnTriggerEnter()
    {


        RawTime = PlayerPrefs.GetFloat("RawTime");

        if(LapsDone <= 1)...
        if (LapsDone < 3)...

        LapsDoneFinish += 1;

        if (LapTimeScript.RawTime <= RawTime)...

        PlayerPrefs.SetInt("MinSave", LapTimeScript.MinCount);
        PlayerPrefs.SetInt("SecSave", LapTimeScript.SecCount);
        PlayerPrefs.SetFloat("MilisecSave", LapTimeScript.MilisecCount);
        PlayerPrefs.SetFloat("RawTime", LapTimeScript.RawTime);

        LapTimeScript.MinCount = 0;
        LapTimeScript.SecCount = 0;
        LapTimeScript.MilisecCount = 0;
        LapTimeScript.RawTime = 0;
        LapCounter.GetComponent<Text>().text = "" + LapsDone;

        HalfLapTrigger.SetActive(true);
        LapCompleteTrigger.SetActive(false);
    }

}
```

Listing 10. Klasa LapCompleteScript

Źródło: opracowanie autora na podstawie [26 i 27] z własnymi poprawkami.

Metoda *OnTriggrEnter()* klasy *LapCompleteScript* przełącza aktywność obiektów w punkcie kontrolnym i na linii startu w odpowiedniej kolejności. Poza tym klasa ta odpowiedzialna jest za wyświetlanie czasu najlepszego okrążenia oraz liczby okrążeń na ekranie gry. Aby móc zarządzać HUD'em (z ang. heads-up display), musimy w nagłówku dodać *using UnityEngine.UI*. W tej klasie mamy zadeklarowane obiekty *LapCompleteTrigger* i *HalfLapTrigger* klasy *GameObject*, które działają na takiej samej zasadzie jak w poprzednim skrypcie. Mamy też zadeklarowane obiekty *MinDisplay*, *SecDisplay*, *MilisecDisplay* i *LapCounter* klasy *GameObject*, do których przypiszemy odpowiednio na HUD obiekty, które będą wykorzystane do wyświetlenia odpowiednio minut, sekund, milisekund i ilości okrążeń. Następnie zostały zadeklarowane zmienne *LapsDone* i *LapsDoneFinish* typu *int* oraz zmienna *RawTime* typu *float*. Dalej znajduje się implementacja metody *OnTriggerEnter()* (zob. *Listing 10*). Na początku do zmiennej *RawTime* przypisujemy wartość „*RawTime*" wyciągniętą z rejestru *PlayerPrefs* (patrz [28]). Klasa *PlayerPrefs* została określona w silniku Unity. Jej zadaniem jest przechowywanie atrybutów gry w rejestrze systemu operacyjnego (patrz metody *SetInt()* oraz *SetFloat()* w Listingu 10). Zatem jej działanie jest ściśle związanie z systemem operacyjnym, patrz [28]. Instrukcja warunkowa *if(LapTimeScript.RawTime <= RawTime)* widoczna na *Listingu 10* sprawdza czy czas aktualnego okrążenia zapamiętywany w polu statycznym *RawTime* klasy *LapTimeScript* (patrz *Listing 11*) jest mniejszy od wartości zmiennej *RawTime* uzyskanej z rejestru systemu operacyjnego. Jeżeli tak, to mamy nowy rekord okrążenia i musimy go wyświetlić. Ciało tej instrukcji warunkowej zajmuje się formatowaniem wyświetlania tego czasu.

Omówmy teraz klasę *LapTimeScript* (patrz *Listing 11*) określoną w pliku *LapTimeScript.cs*. Klasa ta zajmuje się zliczaniem czasu aktualnego okrążenia.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class LapTimeScript : MonoBehaviour
{
    public static int MinCount;
    public static int SecCount;
    public static float MilisecCount;
    public static string MilisecDisplay;
    public GameObject MinBox;
    public GameObject SecBox;
    public GameObject MilisecBox;

    public static float RawTime;

    void Update()
    {
        MilisecCount += Time.deltaTime * 10;
        RawTime += Time.deltaTime;
        MilisecDisplay = MilisecCount.ToString("F0");
        MilisecBox.GetComponent<Text>().text = "" + MilisecDisplay;

        if(MilisecCount >= 10)...

        if(SecCount <= 9)...
        else...

        if(SecCount >= 60)...

        if(MinCount <= 9)...
        else...
    }
}
```

Listing 11. Klasa LapTimeScript

Źródło: opracowanie autora na podstawie [24] z własnymi poprawkami.

W tej klasie ponownie będziemy zarządzać tekstem wyświetlanym na HUD, więc musimy dodać w nagłówku *using UnityEngine.UI*. Na początku znajduje się deklaracja czterech pól odpowiedzialnych za przechowywanie wartości odpowiednio minut, sekund i dwa pola potrzebne do wyświetlenia milisekund. Dodatkowo mamy również deklarację obiektów typu *GameObject* potrzebnych do przypisania im pól, w których będzie wyświetlać się czas, oraz *RawTime* typu *float*. Dopisek *public static* pozwala na dostęp do pól poza skryptem, w którym zostały zadeklarowane, co pozwala na wykorzystanie ich na przykład w pliku *LapCompleteScript.cs*. W metodzie *Update()* do *MilisecCount* przypisujemy kod, który będzie zliczał dziesiętne części sekund, które upłynęły od rozpoczęcia okrążenia. W zmiennej *RawTime* zapamiętywany jest czas jaki minął od rozpoczęcia okrążenia do aktualnej chwili. Jest on potem wykorzystywany w *LapCompleteScript* do sprawdzenia, czy na aktualnie ukończonym okrążeniu pobiliśmy

poprzedni rekord. Wywołanie *MilisecCount.ToString(„F0”)* nadaje zmiennej *MilisecDisplay* wartość pobraną z *MilisecCount* zamienioną na typ *string* z zaokrągleniem liczby w dół. Następnie do obiektu odpowiedzialnego za wyświetlanie milisekund na ekranie przypisujemy wartość w postaci *string'a* ze zmiennej *MilisecDisplay*. Wszystkie kolejne instrukcje warunkowe (patrz *Listing 11*) odpowiedzialne są za formatowanie wyświetlania czasu na ekranie gry.

Teraz omówimy klasę *LoadLapTime* (patrz *Listing 12*) określoną w pliku *LoadLapTime.cs*. Klasa ta została stworzona po to, aby przy załadowaniu wyścigu usunęła wszystkie wpisy w *PlayerStats* (znajdujące się w rejestrze systemu operacyjnego), wyświetlała zera w miejscu czasu okrążenia oraz odpowiednią liczbę okrążeń do pokonania.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class LoadLapTime : MonoBehaviour
{
    public int MinCount;
    public int SecCount;
    public float MilisecCount;
    public GameObject MinDisp;
    public GameObject SecDisp;
    public GameObject MilisecDisp;
    public GameObject LapCounter;

    void Start()...

}
```

Listing 12. Klasa LoadLapTime

Źródło: opracowanie autora na podstawie [25] z własnymi poprawkami.

Na początku znajduje się deklaracja pól potrzebnych do wyświetlenia informacji o wyścigu oraz obiektów klasy *GameObject*, którym przypiszemy obiekty odpowiedzialne za wyświetlanie czasu i okrążenia. Metoda *Start()* odpowiedzialna jest za zresetowanie wpisów w rejestrze *PlayerPrefs* oraz wyświetlenie początkowego stanu pokonanych okrążeń i najlepszego czasu.

2.4 Zarządzanie przyciskami i stanem gry

Do zarządzania stanem gry i przyciskami zostało stworzone pięć plików: *ButtonsActions.cs*, *FinishScript.cs*, *GameStateReset.cs*, *PauseScript.cs* oraz *StartScript.cs*. Zacznijmy od omówienia *GameStateReset.cs*, który resetuje globalny stan gry (patrz *Listing 13*).

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameStateReset : MonoBehaviour
{
    void Start()
    {
        LapCompleteScript.LapsDone = 1;
        LapCompleteScript.LapsDoneFinish = 1;
        Time.timeScale = 0.0f;
        LapTimeScript.MinCount = 0;
        LapTimeScript.SecCount = 0;
        LapTimeScript.MilisecCount = 0;
        LapTimeScript.RawTime = 0;
        PauseScript.GamePaused = false;
        FinishScript.GameFinished = false;

    }
}
```

Listing 13. Klasa GameStateReset

Źródło: opracowanie własne autora.

Kod z *Listingu 13* wykonuje się za każdym razem, gdy wchodzimy do menu głównego gry (zob. również *Rysunki 32* i *36*). Niweluje to problem dalszego naliczania czasu i przechowywania wcześniej ukończonej liczby okrążeń po wyjściu do menu głównego. *LapCompleteScript.LapsDone* i *LapCompleteScript.LapsDoneFinish* ustawia zmienne z pliku *LapCompleteScript.cs* na wartość „1”, aby przy odpaleniu wyścigu od nowa zaczynać odliczanie okrążeń od początku. Przypisanie *Time.timeScale = 0.0f* zatrzymuje upływ czasu w całej grze, dzięki czemu przy wczytaniu toru nie będzie się naliczał czas okrążenia (zob. [29]). Następnie zerowane są pola statyczne klasy *LapTimeScript* odpowiedzialne za mierzenie czasu. *PauseScript.GamePaused = false* i *FinishScript.GameFinished = false* mówią, że gra nie jest ani wstrzymana, ani zakończona. Bez tego występowałby problem z poprawnym wyświetlaniem okna z przyciskiem „START” oraz panelu końca wyścigu.

Teraz opiszemy implementację obsługi działania przycisków UI naszej gry. Za ich poprawne działanie odpowiedzialna jest klasa *ButtonsActions* określona w pliku *ButtonsAction.cs*

(zob. *Listing 14*). Plik ten został przypisany do komponentu *Buttons Actions* obiektu *Buttons* (zob. *Rysunek 34*).

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class ButtonsActions : MonoBehaviour

{
    public GameObject StartUI;
    public void PlayGame()
    {
        SceneManager.LoadScene(1);
    }
    public void HowToPlay()
    {
        SceneManager.LoadScene(2);
    }

    public void MainMenu()
    {
        SceneManager.LoadScene(0);
    }

    public void Track1()
    {
        SceneManager.LoadScene(3);
    }

    public void Track2()
    {
        SceneManager.LoadScene(4);
    }

    public void Exit()
    {
        Application.Quit();
    }

    public void StartRace()
    {
        Time.timeScale = 1.0f;
        PauseScript.GamePaused = false;
        StartUI.SetActive(false);
    }
}
```

Listing 14. Klasa ButtonsAction

Źródło: opracowanie własne autora.

W tym pliku zostało zdefiniowane działanie wszystkich przycisków odpowiedzialnych za poruszanie się po każdym menu w grze, rozpoczynanie i kończenie wyścigu. Aby móc przełączać się po scenach gry i wykorzystać do tego HUD, muszą w nagłówku zostać umieszczone wiersze *using UnityEngine.UI* i *using UnityEngine.SceneManagement* (patrz [30]). Na początku zadeklarowany został obiekt *StartUI* klasy *GameObject*, do którego będziemy przypisywać przyciski w silniku gry. Każda metoda klasy *ButtonsAction* wykonuje się tylko w momencie kliknięcia na przypisany pod nią przycisk. Każda scena ma inaczej przypisane przyciski, o czym wspomnimy w rozdziale trzecim. W kodzie odwołaliśmy się do indeksów scen przyporządkowanych przez silnik Unity: *File –> Build Settings*. Metoda *PlayGame()* przełącza nas na scenę wyboru toru. *HowToPlay()* przekierowuje gracza do ekranu z instrukcjami jak grać. Działanie *MainMenu()* skutkuje przełączeniem się na scenę menu głównego. Metody *Track1()* i *Track2()* pozwalają na włączenie pierwszego albo drugiego toru z poziomu menu wyboru trasy. Przycisk przypisany do funkcji *Exit()* wyłącza grę. Po wciśnięciu przycisku z przypisaną do niego metodą *StartRace()* wznawiany jest upływ czasu, który został wcześniej zatrzymany w menu głównym lub przez inny skrypt. Następnie funkcja mówi obiektowi *GamePaused* klasy *PauseScript*, że gra nie jest spauzowana. Dodatkowo *StartUI.SetActive(false)* ukrywa panel pojawiający się tuż po wczytaniu toru (patrz *Rysunek 5*).

Przejdźmy do omówienia klasy *StartScript* określonej w pliku *StartScript.cs* (zob. *Listing 15*).

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class StartScript : MonoBehaviour
{
    public GameObject StartUI;
    private void Start()
    {
        Time.timeScale = 0.0f;
        PauseScript.GamePaused = true;
        StartUI.SetActive(true);
    }
}
```

Listing 15. Klasa StartScript

Źródło: opracowanie własne autora.

Klasa ta jest odpowiedzialna za wstrzymanie gry tuż po jej wczytaniu. Do obiektu *StartUI* przypisujemy panel, który wyświetla się na początku wyścigu (patrz *Rysunek 5*). Następnie w metodzie *Start()* czas jest zatrzymywany, zmienna logiczna odpowiedzialna za pauzę ustawiana na wartość *true*, a okno startowe wyścigu jest wywoływane.

Klasa *PauseScript* (patrz *Listing 16*) odnosi się do zaimplementowania menu pauzy.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class PauseScript : MonoBehaviour
{
    public static bool GamePaused = false;
    public GameObject PauseUI;
    void Resume()
    {
        PauseUI.SetActive(false);
        Time.timeScale = 1.0f;
        GamePaused = false;
    }

    void Pause()
    {
        PauseUI.SetActive(true);
        Time.timeScale = 0.0f;
        GamePaused = true;
    }
    void Update()
    {

        if (Input.GetKeyDown(KeyCode.Escape) && FinishScript.GameFinished == false)
        {
            if (GamePaused)
            {
                Resume();
            }
            else
            {
                Pause();
            }
        }
    }
}
```

Listing 16. Klasa PauseScript

Źródło: opracowanie własne autora.

Na początku zdefiniowane zostało pole statyczne *GamePaused* (z przypisaną wartością *false*), które jest globalnie dostępne, oraz zostało zadeklarowane pole *PauseUI*, do którego przypisane zostało okno pauzy w silniku Unity. Funkcja *Resume()* ustawia panel pauzy na nieaktywny, przywraca upływ czasu oraz ustawia wartość zmiennej *GamePaused* na *false*. W ciele metody

Pause() zostały podane instrukcje przeciwnie do tych zawartych w ciele metody *Resume()*. Czyli wyświetlany jest panel pauzy, zatrzymywany upływ czasu oraz zmienna *GamePaused* dostaje informację, że gra jest zatrzymana. Aby sprawdzić, kiedy ma być wyświetlane lub chowane menu pauzy, w metodzie *Update()* wprowadzamy instrukcję warunkową *if(GamePaused)*. Jeżeli wykryto wciśnięcie klawisza *Esc* oraz gra nie jest skończona, to przechodzimy do wykonania tej instrukcji (patrz [31]). Następnie sprawdzana jest zmienna przechowująca informacje o wstrzymaniu gry. Jeżeli gra jest wstrzymana, to wznawiamy ją wykonując metodę *Resume()*. W przeciwnym wypadku pauzujemy grę poprzez wywołanie metody *Pause()*.

Klasa *FinishScript* z *Listingu 17* mówi co ma się stać po przejechaniu wymaganej liczby okrążeń.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class FinishScript : MonoBehaviour
{
    [SerializeField] int NumberOfLaps;
    [SerializeField] GameObject FinishPanel;
    public static bool GameFinished = false;

    void Update()
    {
        if(LapCompleteScript.LapsDoneFinish >= NumberOfLaps+1)
        {
            FinishPanel.SetActive(true);
            Time.timeScale = 0.0f;
            GameFinished = true;
            return;
        }
    }
}
```

Listing 17. Klasa FinishScript

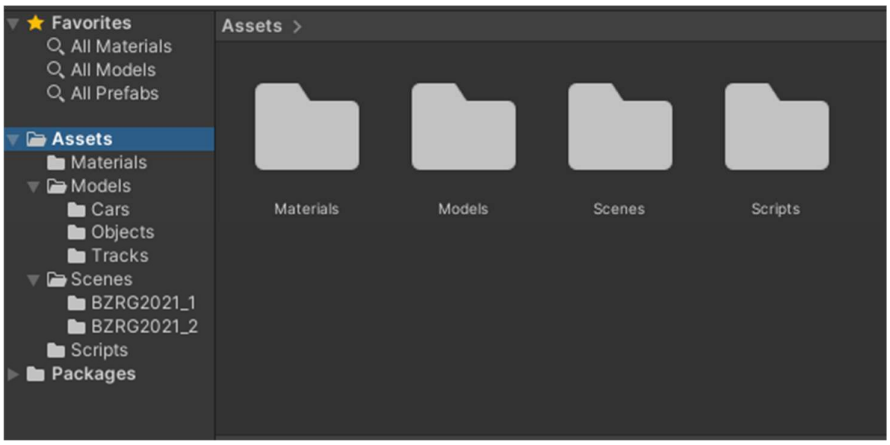
Źródło: opracowanie własne autora.

Na początku deklarowane zostały pole *NumbersOfLaps* typu *int* i obiekt *FinishPanel* klasy *GameObject*. Do zmiennej w silniku gry wpisujemy ilość wymaganych okrążeń do ukończenia wyścigu, a do obiektu przypisujemy panel pokazujący się na zakończenie gry. Zdefiniowane zostało jeszcze pole statyczne *GameFinished* o wartości *false*, które mówi czy gra jest skończona. W ciele metody *Update()* występuje instrukcja sprawdzająca czy liczba aktualnie przejechanych

okrążeń, która wynosi wartość pola *LapCompleteScript.LapsDoneFinish,* jest większa bądź równa tej określającej ilość okrążeń w wyścigu. Jeżeli warunek jest spełniony, to aktywowany jest panel na zakończenie wyścigu, upływ czasu zatrzymywany, a zmienna *GameFinished* dostaje wartość *true*, co oznacza koniec wyścigu. Polecenie *return;* zatrzymuje dalsze wykonywanie się funkcji.

Rozdział III: Sceny gry

Niniejszy rozdział poświęcony jest przedstawieniu scen występujących w grze BZRG2022. Zacznijmy od omówienia zasobów naszej gry w Unity (zob. *Rysunku 29*).



Rysunek 29. Zasoby gry

Źródło: opracowanie własne wygenerowane za pomocą Unity.

Zasoby zostały podzielone na kategorie poprzez umieszczenie ich w poszczególnych folderach (patrz *Rysunek 29*). W folderze *Materials* znajdują się materiały wykorzystane do pokolorowania podłoża torów oraz samochodu. Folder *Models* zawiera modele naszej gry pogrupowane w podkatalogach. W folderze *Cars* znajduje się zaimportowany model samochodu (potencjalnie można dodać inne modele samochodów). Podkatalog *Objects* zawiera modele wszystkich znaków, drzew oraz barierek. Natomiast podfolder *Tracks* zawiera modele torów. Folder *Scripts* przechowuje wszystkie skrypty wykorzystane do zaprogramowania logiki gry. Na *Rysunku 30* widzimy zbiór wszystkich stworzonych w grze scen.

Rysunek 30. Zbiór scen i dodatkowych plików w folderze Scenes

Źródło: opracowanie własne wygenerowane za pomocą Unity.

W folderach *BZRG2021_1* i *BZRG2021_2* znajdują się pliki odpowiedzialne za oświetlenie oraz wygenerowane cienie dla scen z torami. Zostały one wygenerowane przez Unity po naciśnięciu na przycisk *Generate Lighting* w oknie *Lighting* (patrz *Rysunek 31*). Bez wykonania tej czynności

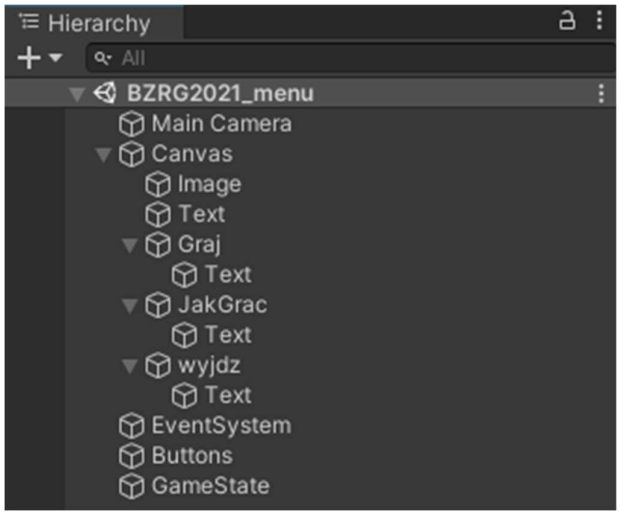
w buildzie gry nie było widać padającego na scenę światła oraz cieni. Okno *Lighting* wywołuje się poprzez *Window –> Rendering –> Lighting*.

Rysunek 31. Wygenerowanie oświetlenia

Źródło: opracowanie własne wygenerowane za pomocą Unity.

3.1 Menu główne, menu z instrukcją oraz menu wyboru toru

W tym podrozdziale omówimy konstrukcje scen menu głównego, menu z instrukcją oraz menu wyboru toru w silniku Unity. Są one bardzo podobnie skonstruowane. Zaczniemy od sceny menu głównego gry. *Rysunek 32* przedstawia hierarchiczną konstrukcję tej sceny.



Rysunek 32. Hierarchia sceny menu głównego gry

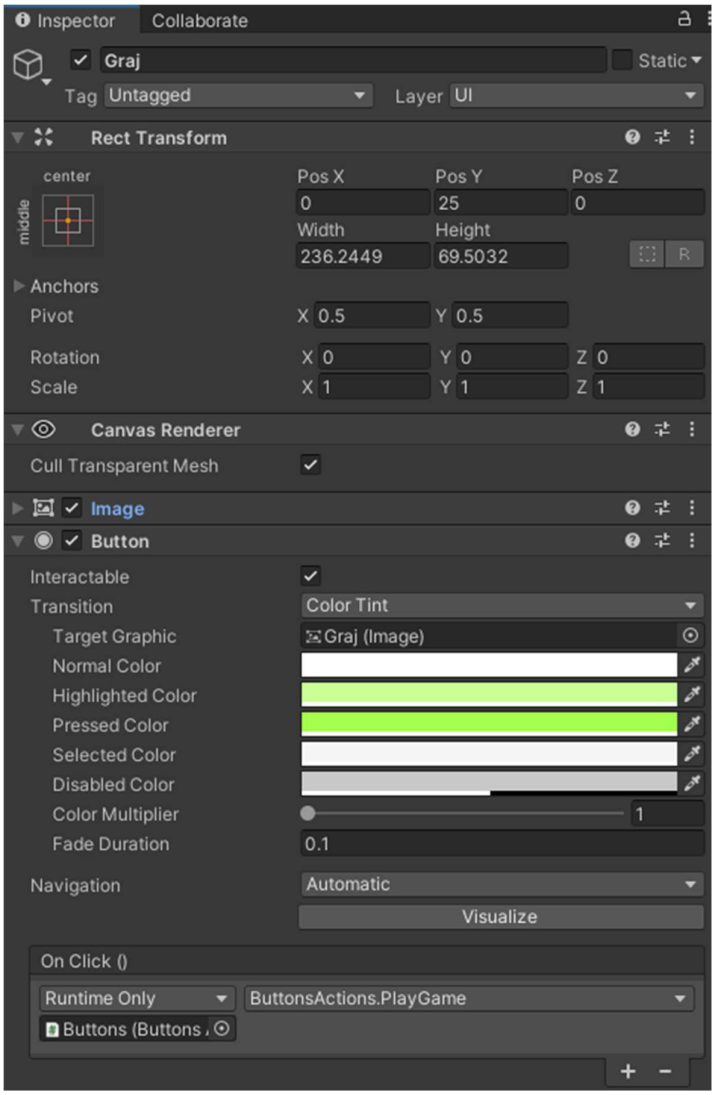
Źródło: opracowanie własne wygenerowane za pomocą Unity.

Pierwszy *GameObject* w hierarchii sceny menu głównego to *Main Camera* będący kamerą patrzącą na scenę. Obiekt *Canvas* jest odpowiedzialny za pogrupowanie wszystkich elementów interfejsu w menu głównym.

Rysunek 33. Wygląd menu głównego

Źródło: opracowanie własne wygenerowane za pomocą Unity.

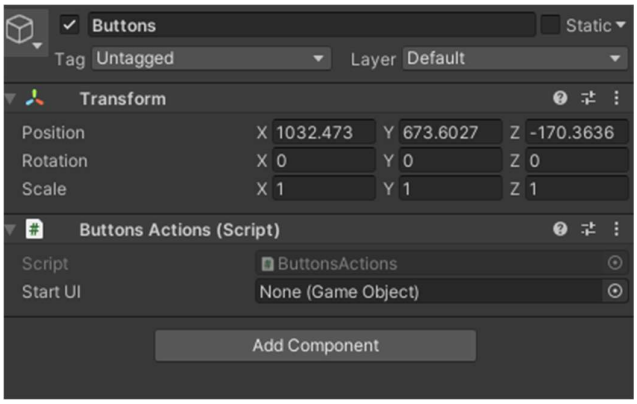
Obiekt *Image* w *Canvas* (patrz *Rysunek 32*) to czarne tło menu głównego gry (patrz *Rysunek 33*). Nazwa gry BZRG2022 została stworzona dzięki dodaniu do *Canvas* obiektu *Text* i zmianie jego koloru na zielony. *Rysunek 34* przedstawia konstrukcję przycisku „Graj" (pozostałe dwa przyciski skonstruowano w podobny sposób).



Rysunek 34. Komponenty obiektu Graj typu Button

Źródło: opracowanie własne wygenerowane za pomocą Unity.

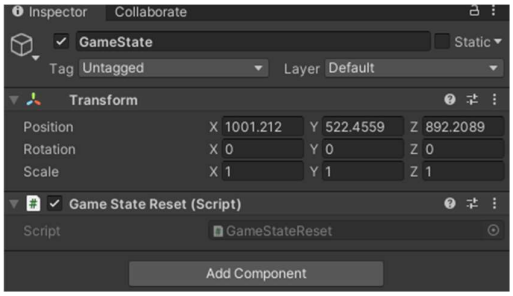
Przyciski „Graj”, „Jak grać” oraz „Wyjdź” zostały dodane do menu głównego poprzez stworzenie obiektów odpowiednio *Graj, JakGrac, wyjdz* klasy *Button* i dodanie ich do *Canvas* jako potomkowie (patrz *Rysunek 32*). Dodatkowo każdy z nich zawiera w sobie obiekt odpowiedzialny za wyświetlanie na nich tekstu. Każdy przycisk posiada te same właściwości jeżeli chodzi o rozmiar i zmianę kolorów po przybraniu różnych stanów (patrz *Rysunek 34*). *Event System* ma za zadanie przesyłać sygnały z urządzeń peryferyjnych do obiektów w aplikacji i został wygenerowany przez silnik Unity automatycznie po dołączeniu do sceny obiektu *Canvas* (patrz [32]).



Rysunek 35. Komponenty obiektu Buttons

Źródło: opracowanie własne wygenerowane za pomocą Unity.

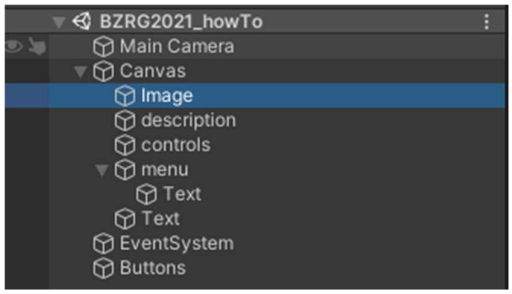
Do *Buttons* przypisany został komponent *Buttons Actions, który zawiera* omówiony w poprzednim rozdziale plik/skrypt *ButtonsActions.cs* odpowiedzialny za obsługę przycisków na scenie (patrz *Rysunek 35*). Każdy przycisk ma dodaną funkcję *On Click ()*, w której jest przypisany obiekt *Buttons* i wybrana odpowiednia funkcja (patrz *Rysunek 34*). W ten sposób umożliwiamy graczowi przemieszczanie się między scenami. Dla przycisku „Graj" wybieramy funkcję *PlayGame()*, dla „Jak Grac" – funkcję *HowToPlay()*, a dla „Wyjdź" wybieramy *Exit()*.



Rysunek 36. Komponenty obiektu GameState

Źródło: opracowanie własne wygenerowane za pomocą Unity.

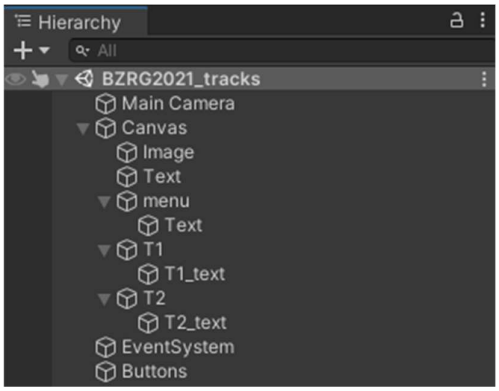
Ostatnim elementem w hierarchii sceny głównego menu jest obiekt *GameState* (patrz *Rysunek 36*). Do niego jest przypisany komponent *Game State Reset* z plikiem *GameStateReset.cs*, który resetuje stan gry po powrocie do menu głównego. Jego działanie zostało omówione w podrozdziale 2.4.



Rysunek 37. Hierarchia obiektów sceny howTo

Źródło: opracowanie własne wygenerowane za pomocą Unity.

Jak widzimy (patrz *Rysunek 37*) scena z instrukcją gry nie różni się zbyt wiele od poprzednio omówionej sceny. Posiada inny układ elementów na ekranie, czyli opis celu gry, sterowania oraz przycisk powrotu do menu głównego (zob. *Rysunek 3*). Poza tym scena nie posiada obiektu *GameState* odpowiedzialnego za resetowanie stanu gry. Nie jest on tutaj potrzebny, bo wykonanie instrukcji z klasy *GameStateReset* przewidziane jest tylko w momencie wywołania sceny z menu głównym gry.



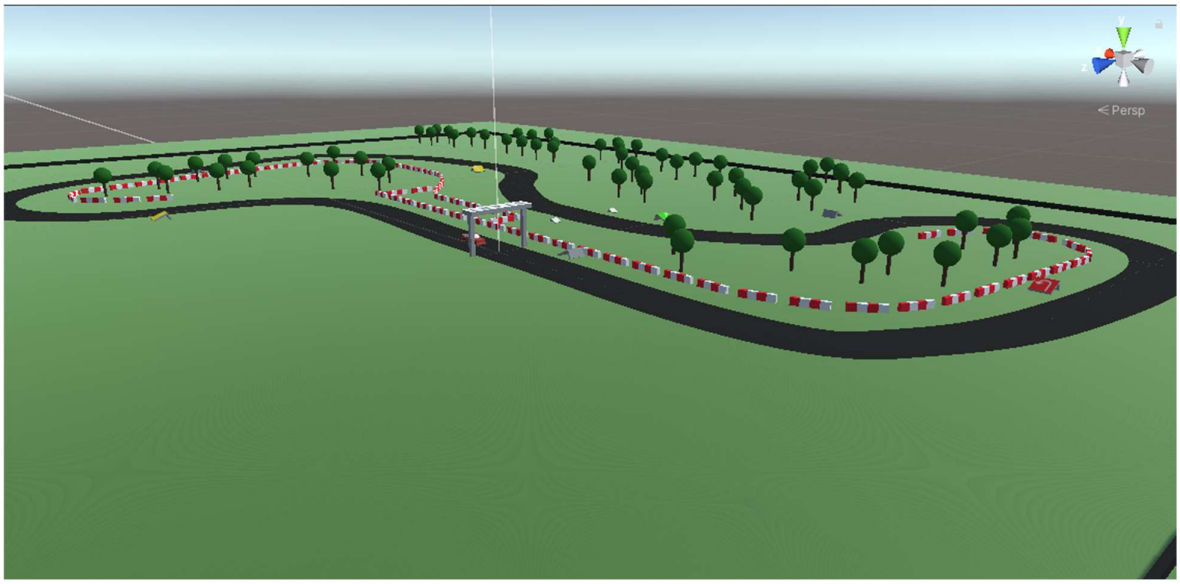
Rysunek 38. Hierarchia obiektów sceny tracks

Źródło: opracowanie własne wygenerowane za pomocą Unity.

Menu wyboru torów posiada nazwę menu w którym się znajdujemy, obrazy torów do wyboru wraz z nazwami oraz przycisk powrotu do menu głównego (patrz *Rysunek 4*). Obiekty *T1* i *T2* to zbiory obiektów odpowiedzialne za wybór konkretnego toru po ich kliknięciu. Została im przypisana funkcja *On Click ()* tak jak w przypadku przycisków nawigacyjnych. W niej ponownie został przypisany obiekt *Buttons* i wykonanie odpowiedniej funkcji przenoszącej na wybrany tor poprzez przypisany do obiektu *Buttons* skrypt *ButtonsAction.cs*.

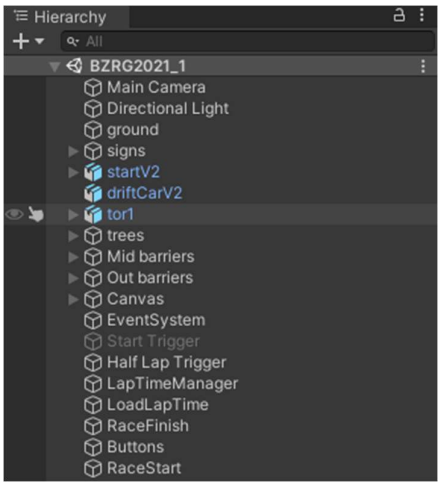
3.2 Pierwszy tor

W tym i następnym podrozdziale zostaną przedstawione projekty torów w silniku Unity. Zacznijmy od pierwszego toru przedstawionego na *Rysunku 39* i jego hierarchii (patrz *Rysunek 40*).



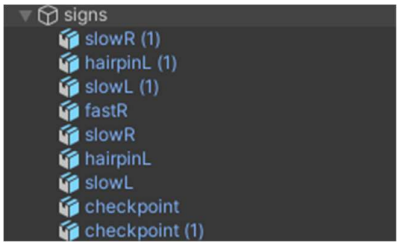
Rysunek 39. Scena pierwszego toru i jego otoczenie w Unity
Źródło: opracowanie własne wygenerowane za pomocą Unity.

Czarne paski na zewnątrz całego terenu toru to barierki uniemożliwiające wyjazd poza mapę, żeby samochód nie spadł z niej. Biało-czerwone barierki wyznaczają granicę toru w taki sposób, żeby nie było możliwości ścinania zakrętów. Dodatkowo teren został ozdobiony wcześniej przygotowanymi modelami drzew.



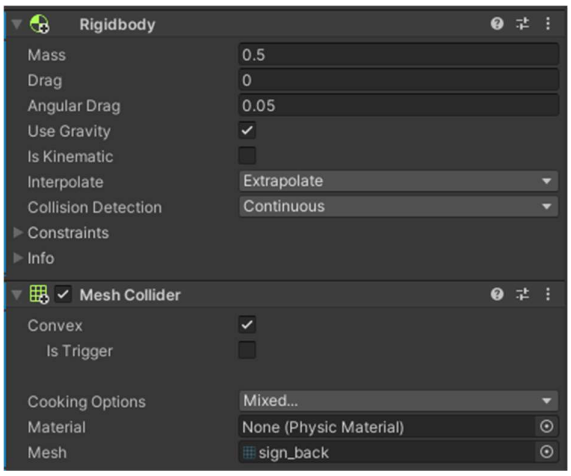
Rysunek 40. Hierarchia obiektów sceny pierwszego toru
Źródło: opracowanie własne wygenerowane za pomocą Unity.

Hierarchia sceny z pierwszym torem (patrz *Rysunek 40*) posiada wcześniej omówione obiekty takie jak *Main Camera*, *EventSystem* i *Buttons*. Element *Directional Light* to światło oświetlające całą scenę. Obiekt *ground* to zielone podłoże imitujące trawę.



Rysunek 41. Zbiór znaków na pierwszym torze
Źródło: opracowanie własne wygenerowane za pomocą Unity.

W obiekcie nadrzędnym *signs* znajdują się prefaby (patrz [33]) będące znakami, które wskazują zakręt znajdujący się za nimi. Są to znaki wcześniej wykonane w Blenderze opisane w pierwszym rozdziale niniejszej pracy. Każdy znak ma dodane komponenty *Rigidbody* i *Mesh Collider* (patrz *Rysunek 42*).



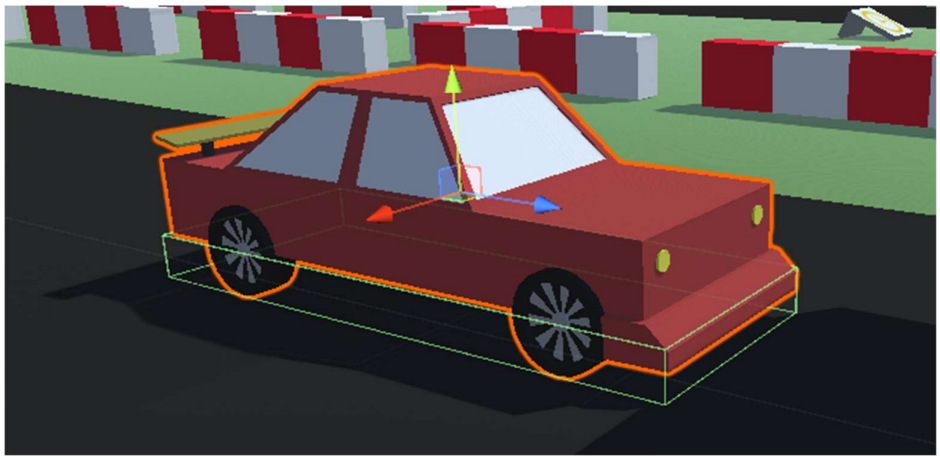
Rysunek 42. Komponenty znaków wskazujących zakręty
Źródło: opracowanie własne wygenerowane za pomocą Unity.

Pierwszy komponent odpowiedzialny jest za nadanie obiektowi praw fizyki. *Mesh Collider* definiuje na krawędziach obiektu siatkę, która po zderzeniu z innym obiektem posiadającym ten sam komponent, wykrywa kolizję. Na przykład wjeżdżając samochodem, który też posiada ten sam komponent, w znak będzie skutkować jego przesunięciem przy czym samochód zostanie spowolniony.

Prefab *startV2* to znak postawiony na linii startu. Po rozwinięciu go w hierarchii widzimy, że składa się on z dwóch *mesh'ów* tak jak to zostało wcześniej przygotowane w Blenderze. Jak już zostało wspomniane wcześniej *GameObject* samochodu gracza jest nazwany *driftCarV2*.

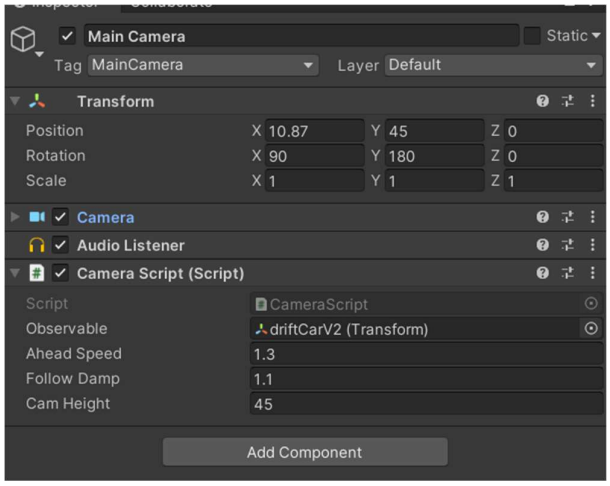
Rysunek 43. Prefaby znaku startu, samochodu i linii toru w hierarchii sceny
Źródło: opracowanie własne wygenerowane za pomocą Unity.

Obiekt *driftCarV2* posiada komponent *Car Behaviour z* podpiętym skryptem *CarBehaviour.cs* oraz komponenty *Box Collider* i *Rigidbody*. *Box Collider* oprócz wyznaczenia strefy wykrywania kolizji (patrz *Rysunek 44*) wyznacza również środek ciężkości pojazdu.



Rysunek 44. Samochód wraz z siatką Box Collider
Źródło: opracowanie własne wygenerowane za pomocą Unity.

Żeby samochód nie przewracał się podczas skręcania środek ciężkości został obniżony poprzez zmniejszenie wielkości *Box Collider* wzdłuż osi Z. Dodatkowo położenie, skala i obrót pojazdu zostały przypisane do pola *Observable* klasy *CameraScript* określonej w skrypcie *CameraScript.cs.* Z kolei skrypt ten przypisany został do kamery na scenie (patrz *Rysunek 45*).

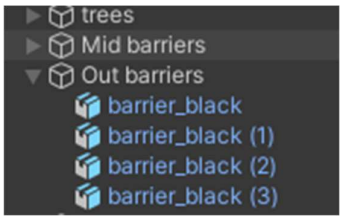


Rysunek 45. Komponenty kamery

Źródło: opracowanie własne wygenerowane za pomocą Unity.

Prefab *tor1* (patrz *Rysunki 40* oraz *43*), tak jak w Blenderze, składa się z krzywej *Bezier* oraz szeregu obiektów *Plane* ułożonych wzdłuż tej krzywej. Jest to wynikiem importowania odpowiedniego pliku FBX toru (utworzonego w Blenderze) do naszego projektu w silniku Unity (zob. też podrozdział 1.1).

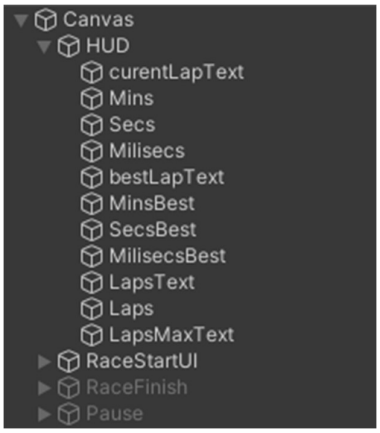
Obiekty *trees*, *Mid barriers* i *Out barriers* to pogrupowane zbiory drzew, barier wyznaczających granicę toru oraz barier blokujących wyjazd samochodu poza mapę (patrz *Rysunki 40* oraz *46*).



Rysunek 46. Zbiór drzew, barier toru i barier świata w hierarchii sceny

Źródło: opracowanie własne wygenerowane za pomocą Unity.

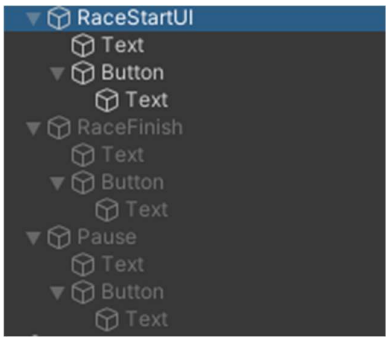
Hierarchia HUD na scenie pierwszego i drugiego toru wygląda tak samo (zob. *Rysunek 47*).



Rysunek 47. Elementy HUD scen obu torów

Źródło: opracowanie własne wygenerowane za pomocą Unity.

Obiekt HUD ma przypisany komponent *Pause Script* klasy *PauseScript*, w którym przypisany jest zbiór obiektów *Pause* będący menu pauzy. Elementy od *curentLapText* do *LapsMaxText* włącznie są wyświetlane po lewej górnej stronie ekranu (patrz *Rysunek 5*). Obiekt *currentLapText* prezentuje napis „Czas okrążenia", *bestLapText* – napis „Najlepsze okrążenie", a *LapsText* – napis „Okrążenie". Element o nazwie *LapsMaxText* odpowiedzialny jest za wyświetlenie na HUD liczby okrążeń, którą trzeba pokonać, aby skończyć wyścig. Obiekty *Mins*, *Secs*, *Milisecs* są przypisane do komponentu *Lap Time Script* obiektu *LapTimeManager* (zob. *Rysunek 52*), a *MinsBest*, *SecsBest*, *MilisecsBest* i *Laps* komponentu *Load Lap Time* obiektu *LoadLapTime* (zob. *Rysunek 53*). Obiekty *LapTimeManager* i *LoadLapTime* mają do siebie przypisane skrypty odpowiedzialne za zliczanie aktualnego okrążenia oraz najlepszego okrążenia (zob. *Rysunki 52* oraz *53*). Oba obiekty bardziej szczegółowo zostaną opisane w dalszej części tego rozdziału.



Rysunek 48. Menu startu, końca wyścigu i pauzy

Źródło: opracowanie własne wygenerowane za pomocą Unity.

Obiekt *RaceStartUI* reprezentuje menu startowe gry pokazujące się po wczytaniu toru (patrz *Rysunki 5* oraz *48*). Zawiera ono obiekt *Text* z napisem „Kliknij START aby rozpocząć sesję" oraz przycisk z napisem „START". Przycisk ma przypisaną funkcję *On Click ()*, która odnosi się do metody *StartRace()* obiektu *Buttons.*

Rysunek 49. Okno informujące o końcu wyścigu
Źródło: opracowanie własne wygenerowane za pomocą Unity.

Na początku wyścigu obiekt *RaceFinish* jest nieaktywny, ponieważ ma się pokazywać tylko i wyłącznie po ukończeniu wyścigu. Obiekt ten zawiera również okno z napisem „KONIEC WYŚCIGU" oraz przycisk „Powrót do menu" (zob. *Rysunek 49*). Przycisk ten działa na podobnej zasadzie jak przycisk „START". Różni się tym, że przy wywołaniu *On Click ()* wykonuje metodę *MainMenu()* obiektu *Buttons*.

Ostatnim oknem, które omówimy, jest okno pauzy reprezentowane przez obiekt *Pause*. Zawiera ono tekst „Pauza" oraz przycisk „Powrót do menu" (patrz *Rysunek 6*). Menu pauzy nie ma dodanej funkcji *On Click ()*. Skrypt *PauseScript.cs* odpowiedzialny za jego działanie został przypisany do komponentu *Pause Script* obiektu *Canvas*.

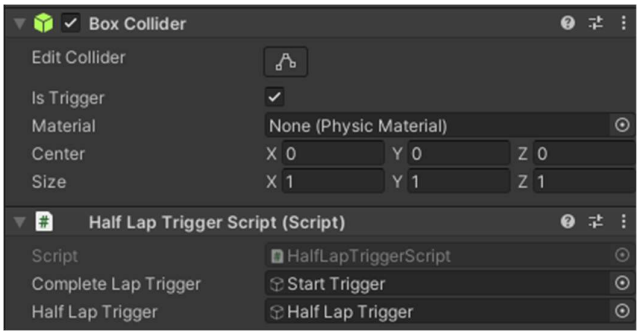
Przejdźmy teraz do omówienia obiektu *Start Trigger*, który ustawia punkt kontrolny na linii startu (patrz *Rysunek 50*). Przy załadowaniu sceny jest on nieaktywny (patrz *Rysunek 40*). Obiekt ten aktywuje się gdy gracz przejedzie przez punkt kontrolny znajdujący się w połowie toru. Do *Start Trigger* dodany został komponent *Box Collider* (patrz *Rysunek 50*), który przy kolizji aktywuje działanie skryptów odpowiedzialnych za zliczanie czasu okrążeń omówionych w drugim rozdziale.

Rysunek 50. Komponenty obiektu Start Trigger

Źródło: opracowanie autora na podstawie [24] z własnymi poprawkami.

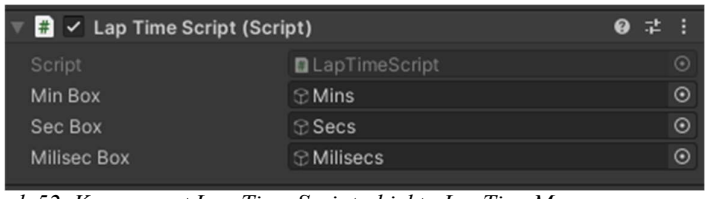
Obiekt ma też podpięty komponent *Lap Complete Script* zawierający skrypt *LapCompleteScript.cs*. W nim zostały wskazane obiekty odpowiedzialne za wyświetlanie czasu okrążenia, ilości pokonanych okrążeń oraz drugi punkt kontrolny. Obiekt ten wygląda podobnie jak ten z *Rysunku 28*.

Half Lap Trigger to obiekt znajdujący się w miejscu punktu kontrolnego w połowie trasy (zob. *Rysunek 51*). Tak jak *Start Trigger* posiada ten sam *Box Collider*. Obiekt został rozszerzony o komponent *Half Lap Trigger Script* zawierający skrypt *HalfLapTriggerScript.cs* omówiony w drugim rozdziale. W nim zostały wskazane obiekty będące punktami kontrolnymi na linii startu i w połowie trasy.



Rysunek 51. Komponenty przypisane do obiektu Half Lap Trigger

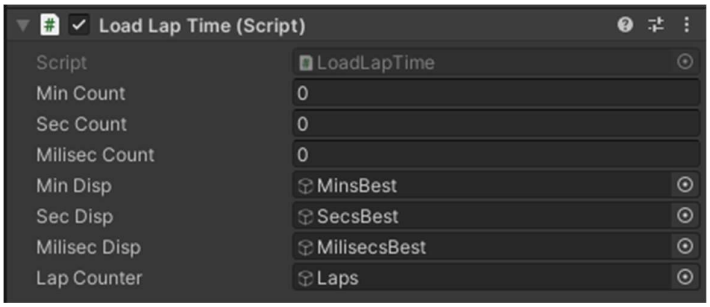
Źródło: opracowanie autora na podstawie [24] z własnymi poprawkami.



Rysunek 52. Komponent Lap Time Script obiektu LapTimeManager

Źródło: opracowanie autora na podstawie [24] z własnymi poprawkami.

Patrząc na *Rysunek 52* możemy zauważyć, że dla obiektu *LapTimeManager* do pól w komponencie *Lap Time Script* zostały przypisane obiekty odpowiedzialne za wyświetlanie milisekund, sekund i minut.

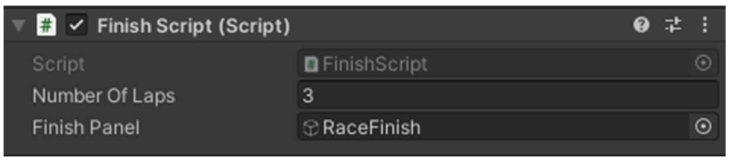


Rysunek 53. Komponent Load Lap Time obiektu LoadLapTime

Źródło: opracowanie autora na podstawie [24] z własnymi poprawkami.

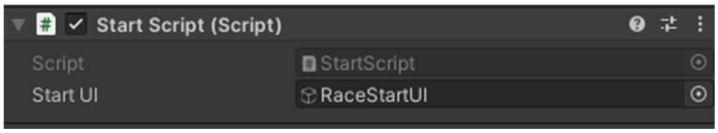
Obiekt *LoadLapTime* został wzbogacony o komponent *Load Lap Time* klasy *LoadLapTime* (patrz *Rysunek 53*) odpowiedzialny za wyświetlanie najlepszego czasu okrążenia danej sesji oraz ilości pokonanych okrążeń. Zostały w nim wskazane pola minut, sekund, milisekund oraz ilości okrążeń.

Obiekt *RaceFinish* jest elementem odpowiedzialnym za wyświetlenie okna zakończenia wyścigu po przejechaniu określonej liczby okrążeń (patrz *Rysunki 40 i 54*). W polu *Number of Laps* możemy ustalić ilość okrążeń potrzebnych do ukończenia wyścigu. W polu *Finish Panel* wskazujemy przygotowane wcześniej okno HUD informujące o ukończeniu wyścigu.



Rysunek 54. Komponent Finish Script obiektu RaceFinish

Źródło: opracowanie własne wygenerowane za pomocą Unity.



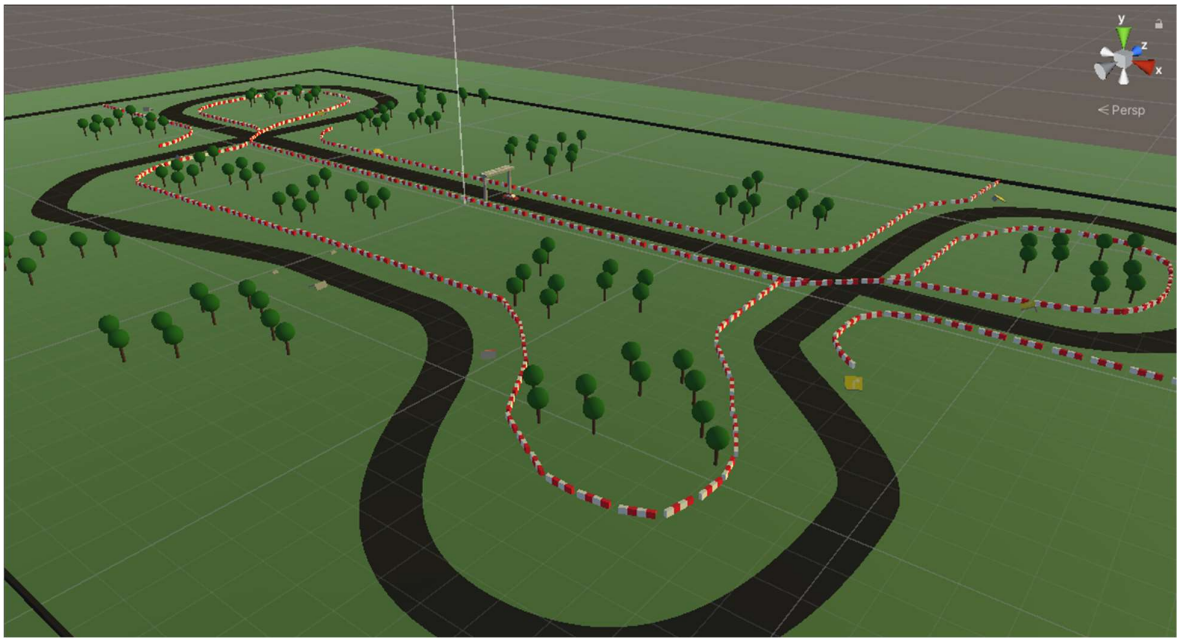
Rysunek 55. Komponent Start Script obiektu RaceStart

Źródło: opracowanie własne wygenerowane za pomocą Unity.

Do obiektu *RaceStart* został dodany komponent *Start Script* z przypisanym do niego skryptem *StartScript.cs*, oraz obiektem *RaceStartUI*, a w nim okno HUD z *Rysunku 5* pojawiające się po załadowaniu sceny (patrz *Rysunek 55*).

3.3 Drugi tor

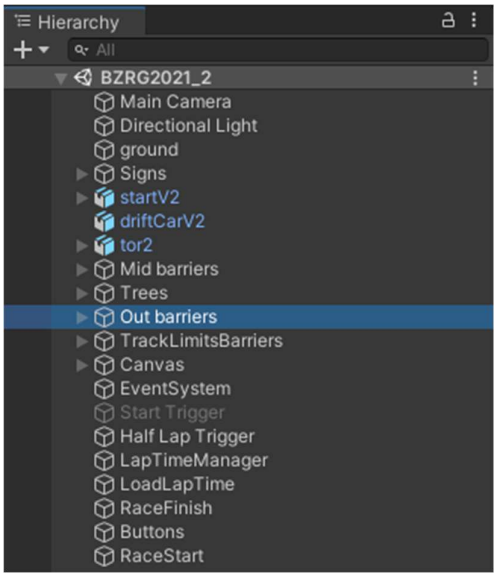
Do omówienia pozostała ostatnia scena gry, czyli drugi tor (patrz *Rysunki 56* i *57*).



Rysunek 56. Scena drugiego toru i jego otoczenie w Unity

Źródło: opracowanie własne wygenerowane za pomocą Unity.

Drugi tor jest większy od pierwszego i ma inny układ zakrętów (por. *Rysunki 36* i *56*). Posiada więcej drzew i barierek.



Rysunek 57. Hierarchia obiektów sceny drugiego toru

Źródło: opracowanie własne wygenerowane za pomocą Unity.

Hierarchia drugiego toru jest bardzo podobna do tej z pierwszego (por. *Rysunki 40* i *57*). Ma podobnie pogrupowane różne typy obiektów. Obiekt *TrackLimitsBarriers* reprezentuje te same barierki zapobiegające cięciu zakrętów co *Mid barriers* ze sceny pierwszego toru. Również oświetlenie sceny zostało lekko zmienione w porównaniu do toru pierwszego. Wszystkie pozostałe elementy mają takie same komponenty, parametry i logikę za nimi stojącą.

Podsumowanie

Projektowanie każdej gry komputerowej, nie ważne jak bardzo skomplikowanej, wymaga od twórcy wcześniejszego przemyślenia i przygotowania jej koncepcji. Nie inaczej było w przypadku tworzenia trójwymiarowej gry wyścigowej, której projekt stanowi integralną część niniejszej pracy dyplomowej. Na etapie projektowania naszej gry, która nazywa się BZRG2022, ważne było uwzględnienie silnika gier, którym posługiwaliśmy się. W naszym przypadku był to silnik Unity (wersja 2020.3.24f1). Posługiwanie się Unity wymaga nie tylko posiadania wiedzy na temat samego Unity, ale także wiedzy na temat trójwymiarowej grafiki komputerowej, jej matematycznych aspektów, zjawisk fizycznych zachodzących przy wyścigu, oraz umiejętności programowania w języku C#. Program Blender okazał się dość wygodnym i wydajnym narzędziem do tworzenia modeli 3D na potrzeby naszego projektu. W trakcie tworzenia gry napotkaliśmy wiele przeszkód i problemów (np. związanych z poprawnym zliczaniem okrążeń czy też czasu). Wyżej wymienione umiejętności pomogły w ich rozwiązaniu i stworzeniu gry wyścigowej zgodnie z zamierzeniami autora niniejszej pracy. Należy podkreślić, że znajomość programowania zorientowanego obiektowo, jest niezbędna nie tylko w celu implementacji logiki oraz interfejsu użytkownika gry, ale również przy analizowaniu dokumentacji silnika Unity.

Bibliografia

1. Geig, M., *Unity: przewodnik projektowania gier: 24h do własnej gry*, Wydawnictwo HELION, 2015.
2. Ross, J., *Unity i C#: praktyka programowania gier*, Wydawnictwo HELION, 2020.
3. Simonds, B., *Blender: praktyczny przewodnik po modelowaniu, rzeźbieniu i renderowaniu*, Wydawnictwo HELION, 2014.

4. https://docs.unity3d.com/ScriptReference/MonoBehaviour.html (data dostępu: 10.11.2021)
5. https://docs.unity3d.com/2021.2/Documentation/ScriptReference/Quaternion.html (data dostępu: 10.11.2021)
6. https://docs.unity3d.com/ScriptReference/Rigidbody.html (data dostępu: 10.11.2021)
7. https://docs.unity3d.com/Manual/CameraRays.html (data dostępu: 10.11.2021)
8. https://docs.unity3d.com/ScriptReference/Plane.html (data dostępu: 10.11.2021)
9. https://docs.unity3d.com/ScriptReference/Camera.ScreenPointToRay.html (data dostępu: 10.11.2021)
10. https://docs.unity3d.com/ScriptReference/Vector3.html (data dostępu: 10.11.2021)
11. https://docs.unity3d.com/ScriptReference/Ray.GetPoint.html (data dostępu: 10.11.2021)
12. https://docs.unity3d.com/ScriptReference/Mathf.Atan2.html (data dostępu: 10.11.2021)
13. https://docs.unity3d.com/ScriptReference/Quaternion.Euler.html (data dostępu: 10.11.2021)
14. https://docs.unity3d.com/ScriptReference/Rigidbody-velocity.html (data dostępu: 10.11.2021)
15. https://docs.unity3d.com/ScriptReference/Input.GetMouseButton.html (data dostępu: 10.11.2021)
16. https://docs.unity3d.com/ScriptReference/Time-fixedDeltaTime.html (data dostępu: 10.11.2021)
17. https://docs.unity3d.com/ScriptReference/Transform-rotation.html (data dostępu: 10.11.2021)
18. https://docs.unity3d.com/ScriptReference/Quaternion.Lerp.html (data dostępu: 10.11.2021)
19. https://docs.unity3d.com/ScriptReference/Mathf.Clamp.html (data dostępu: 10.11.2021)
20. https://docs.unity3d.com/ScriptReference/Component.GetComponent.html (data dostępu: 10.11.2021)
21. https://docs.unity3d.com/ScriptReference/Transform-position.html (data dostępu: 10.11.2021)
22. https://docs.unity3d.com/ScriptReference/Time-deltaTime.html (data dostępu: 10.11.2021)
23. https://www.youtube.com/watch?v=pfqY3J6jsc4&list=PLZ1b66Z1KFKgkE9ji0tF2iDO0LGxmlwIm&index=5 (data dostępu: 20.12.2021)
24. https://www.youtube.com/watch?v=hEGCoV_X7Io&list=PLZ1b66Z1KFKgkE9ji0tF2iDO0LGxmlwIm&index=6 (data dostępu: 20.12.2021)

25. https://www.youtube.com/watch?v=WlWhWjALrso&list=PLZ1b66Z1KFKgkE9ji0tF2iDO0LGxmlwIm&index=10 (data dostępu: 20.12.2021)
26. https://www.youtube.com/watch?v=Rkk3LBGlLyI&list=PLZ1b66Z1KFKgkE9ji0tF2iDO0LGxmlwIm&index=11 (data dostępu: 20.12.2021)
27. https://www.youtube.com/watch?v=PyJCz58V7JQ (data dostępu: 20.12.2021)
28. https://docs.unity3d.com/ScriptReference/PlayerPrefs.html (data dostępu: 20.12.2021)
29. https://docs.unity3d.com/ScriptReference/Time-timeScale.html (data dostępu: 21.12.2021)
30. https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.LoadScene.html (data dostępu: 21.12.2021)
31. https://docs.unity3d.com/ScriptReference/Input.GetKeyDown.html (data dostępu: 21.12.2021)
32. https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/EventSystem.html (data dostępu: 3.05.2022)
33. https://docs.unity3d.com/Manual/Prefabs.html (data dostępu: 3.05.2022)