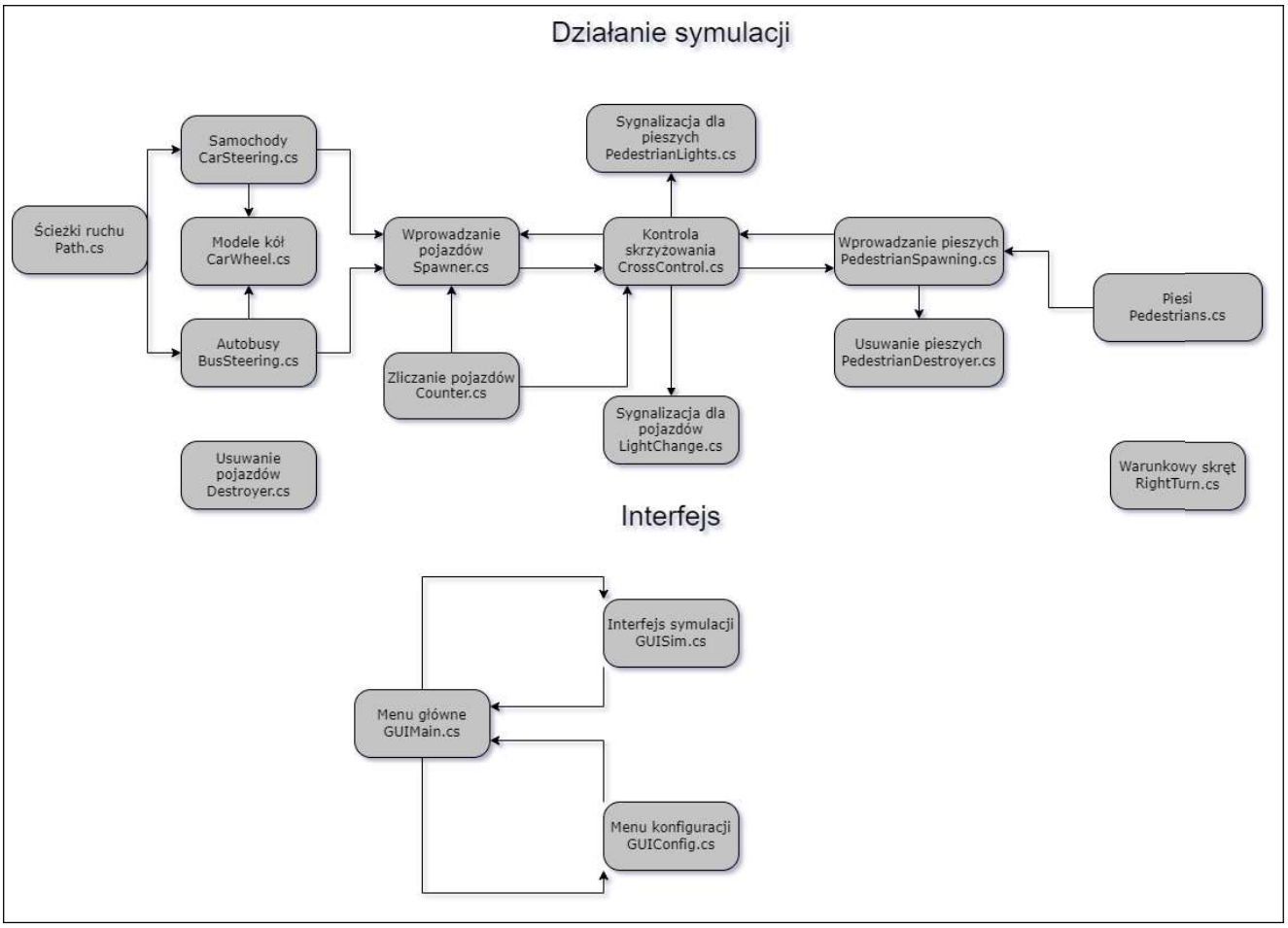
3.7 Oprogramowanie modelu

Po połączeniu wszystkich modeli i zbudowaniu całej sceny (rys. 3.18) model zawierał już wszystkie potrzebne obiekty do oprogramowania i możliwe było napisanie skryptów umożliwiających działanie całego modelu.

Rysunek 3.18: Ostateczny wygląd skrzyżowania ze wszystkimi modelami

Pomimo tego, że sam silnik *Unity* został napisany w języku C++, skrypty tworzone do programowania projektów można pisać w 2 innych językach. Pierwszym z nich jest język stworzony specjalnie na potrzeby użytku w IDE *Unity* (integrated development environment - ang. zintegrowane środowisko programistyczne) i jest to *UnityScript*. Został oparty na podstawie *JavaScript*, więc jest to doskonała opcja dla użytkowników, którzy mają doświadczenie z jego pierwowzorem. Drugą opcją, oraz opcją wykorzystaną w tym projekcie jest C#. Jest to język programowania będący standardem w branży, szczególnie w oprogramowaniu związanym z firmą *Microsoft*. Pomimo tego, że aplikacja silnika pozwala na modyfikację rożnych zmiennych znajdujących się w kodzie, to nie posiada ona wbudowanego edytora i potrzebuje zewnętrznej aplikacji do osiągnięcia pełnej funkcjonalności. Ze względu na wspomnianą już dużą kompatybilność z aplikacjami korporacji *Microsoft* oraz wbudowanym wsparciem dla *Unity*, przy tworzeniu tej pracy wykorzystany został edytor *Microsoft Visual Studio 2019* w wersji *Enterprise* zapewnionej przez Politechnikę Poznańska na podstawie jej udziału w programie „Microsoft Azure Dev Tools for Teaching”.

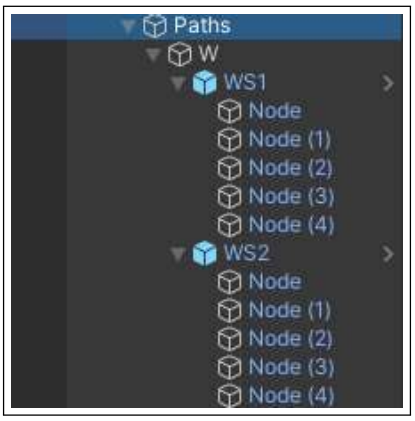
Programowanie w *Unity* nie odbywa się w jednym miejscu, ponieważ każdy rodzaj obiektu potrzebujący opisu powinien posiadać swój własny skrypt. Z tego względu w projekcie modelu znajduje się 17 plików z kodem. Wszystkie elementy nie są jednak całkowicie oddzielne i występuje pomiędzy nimi wiele zależności koniecznych do prawidłowego działania. Dokładniejsze szczegóły połączeń zostaną omówione podczas opisywania konkretnych modułów, ale ich ogólny schemat widać na rysunku 3.19 poniżej.



Rysunek 3.19: Ideowy schemat wszystkich połączeń między skryptami wraz z ich głównymi zadaniami

Path.cs

Na samym początku programowania ruchu pojazdów należało zdecydować w jaki sposób podejść do tego zadania. Pierwszą możliwością byłoby matematyczne opisanie wszystkich możliwych ścieżek, którymi uczestnicy ruchu mogą się poruszać. Takie rozwiązanie byłoby niezwykle czasochłonne, a niektóre ze ścieżek mogłyby się okazać dosyć skomplikowane. Dodatkowo w takim koncepcie trudno wprowadzać jakiekolwiek poprawki, a proces ewentualnego rozwoju projektu byłby niesamowicie mozolny. Z tych względów zdecydowano się na nawigacje na podstawie punktów istniejących w przestrzeni trójwymiarowej modelu. Obiekty w *Unity* można łączyć ze sobą co daje możliwość stworzenia obiektu „rodzica", który zawiera w sobie wszystkie punkty posiadające odpowiednie współrzędne (rys. 3.20).



Rysunek 3.20: Struktura ścieżek w edytorze *Unity*

W teorii żaden z elementów struktury nie wymaga żadnego dodatkowego programowanie, aby takie rozwiązanie działało. W celu dogodnego tworzenia ścieżek warto je jednak zwizualizować w samym edytorze projektu za pomocą skryptu Path.cs. Większość kodu z tego pliku wywoływane jest w funkcji OnDrawGizmosSelected() co powoduje, że ścieżka jest widoczna jedynie w oknie edycji oraz tylko wtedy kiedy obiekt jest zaznaczony. Zaczęto od zdefiniowania koloru rysowania (wcześniej wybranego w edytorze *Unity*), stworzono listę oraz pobrano i umieszczono w niej koordynaty ze wszystkich obiektów znajdujących się w „rodzicu". Następnie dodano wszystkie współrzędne do powstałej listy zachowując ich początkową kolejność, ale pominięto lokalizację głównego elementu, ponieważ nie jest ona związana z samą ścieżką. Ostatnia pętla pobiera z listy umiejscowienie aktualnego i poprzedniego punktu, aby narysować pomiędzy nimi linię w ustalonym kolorze, oraz kulę w miejscu aktualnego węzła (rys. 3.21). Dzięki tym wszystkim zabiegom tworzenie kolejnych punktów wymaga jedynie duplikacji poprzedniego obiektu oraz przesunięcie go w oknie edytora na pożądane miejsce.

Rysunek 3.21: Wizualizacja ustalonych ścieżek

CarSteering.cs oraz BusSteering.cs

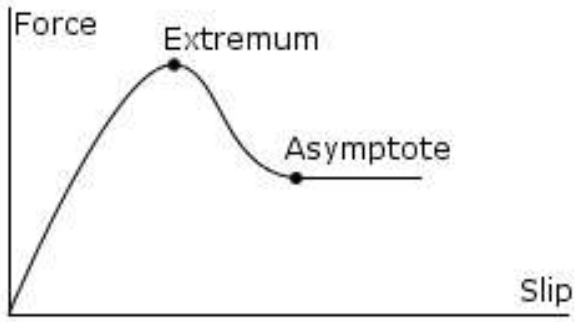
Posiadając ścieżki ruchu dla pojazdów należało przejść do opisania ich ruchu. *Unity* jest bardzo uniwersalnym narzędziem i istnieje wiele różnych sposobów na uzyskanie tego efektu. Najprostszym z nich jest zwykłe zmienianie współrzędnych poruszanego obiektu. Metoda ta jest wystarczająca dla bardzo prostych ruchów jednak zwykle zależna jest od takich elementów jak ilość wyświetlanych klatek. Powoduje to duże trudności przy poruszaniu się z różnymi prędkościami czy nadawaniu obiektowi bardziej skomplikowanych trajektorii.

Do wszystkich używanych struktur można dodawać wbudowane w *Unity* komponenty, które wymuszają najróżniejsze zachowania w projekcie. Jednym z takich komponentów jest *Rigid Body* (bryła sztywna). Powoduje on, że na zawierającą go strukturę może działać wbudowany silnik fizyczny. Daje to programiście dużo możliwości, ponieważ można symulować wpływy najróżniejszych sił na obiekty z modyfikowalnymi masami czy ustalonymi wartościami tarcia. Dzięki temu wszystkimi obiektami można także poruszać po prostu przykładając do nich siłę w odpowiednim miejscu. Daje to użytkownikowi dużo większą kontrolę nad ruchem z uwzględnieniem zewnętrznych czynników takie jak siła grawitacji. Aby bryła sztywna była w stanie wchodzić interakcje z innymi podobnymi obiektami należy dodać komponent typu *collider* pozwalający wykrywać zderzenia. Komponenty te występują w wielu kształtach (np. sześcian, kula, płaska powierzchnia), aby móc je dopasować jak najlepiej do struktury, z którą są wykorzystywane. Jeżeli posiada ona model trójwymiarowy można nawet wykorzystać jej siatkę za bazę wszelkich zderzeń. Po kolizji dwóch lub większej ilości obiektów można pozwolić działać silnikowi fizycznemu i obliczyć jej skutki albo opisać swoje własne za pomocą skryptu. Łącząc te dwa komponenty można uzyskać niezwykle potężne narzędzie pozwalające na stworzeniu wielu kontrolerów ruchu obiektów, jednak wadą

tego rozwiązania jest ilość pracy jakiej wymaga jego wdrożenie. Warto z niego korzystać jedynie kiedy dąży się do bardzo specyficznych efektów, których nie da się uzyskać mniej czasochłonnymi metodami.

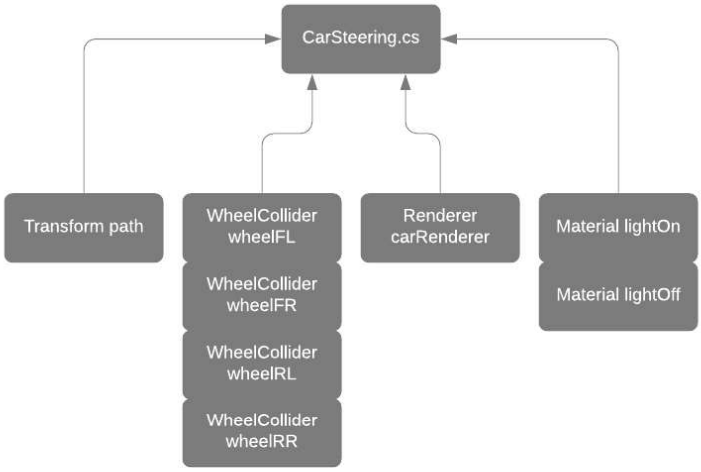
Uproszczeniem powyższej metody jest wykorzystanie komponentu *Character Controller* [46]. Jest on swoistym połączeniem elementów omówionych wcześniej, ale bierze z nich jedynie właściwości oraz metody potrzebne do sterowania poruszaniem obiektu. Dodawane jest także wiele parametrów oraz funkcji stworzonych szczególnie z myślą o opisywaniu ruchu. Jest to niezwykle wygodne narzędzie, ale niestety nie nadaje się do poruszania pojazdami. Jego głównym zamysłem od początku było kontrolowanie postaci przez co zawarty *collider* ma kształt kapsuły i w przypadku samochodu jest bardzo niepraktyczny. Większość nowych funkcji także kierowana jest jedynie dla postaci, więc przykładowa możliwość poruszania się po schodach nie jest ani trochę przydatna dla modeli samochodów.

Istnieje jeszcze jedno rozwiązanie problemu poruszania się samochodów, które doskonale nadaje się do tego zadania. *Unity* posiada także komponent o nazwie *Wheel Collider* [47], który nie działa tak jak wszystkie wspomniane wcześniej elementy tego typu. W rzeczywistości umożliwia on pełną symulację fizyczną koła, którą można użyć do opisywania pojazdów takich jak samochody. Aby móc z niego korzystać cały pojazd musi już być ustanowiony bryłą sztywną, a następnie należy dodać *Wheel Collider* [47] w miejsce każdego koła. Komponent ten sam w sobie nie posiada reprezentacji graficznej, ale można go umieścić tam gdzie istnieją modele kół, a następnie w kodzie powiązać ich lokalizację oraz wszystkie możliwe rotacje. Oczywiście *Unity* nadal bierze pod uwagę silnik fizyczny więc każdemu kołu należy nadać odpowiednią masę, promień czy działające na nie tłumienie. Dodatkowo silnik symuluje tradycyjne zawieszenie składające się ze sprężyny i tłumika co daje użytkownikowi dostęp do ustawiania zakresu ruchu, a także wartości sprężystości i tłumienia. Bardzo ważne dla zachowania realistycznego poruszania się jest odpowiednia konfiguracja tarcia działającego na koło zarówno w kierunku poruszania się jak i w kierunku bocznym. Ustalenie tej charakterystyki sprowadza się do umiejscowienia punktów *Extremum* (maksymalnej wartości tarcia statycznego), *Asymptote* (wartości tarcia kinetycznego) na poniższym wykresie (rys. 3.22) oraz ustalenie jego skali za pomocą zmiennej *Stiffness*.



Rysunek 3.22: Charakterystyka tarcia kół [48]

Cały system poruszania się samochodów zawarty został w pliku CarSteering.cs i aby umożliwić jego działanie należy dodać go jako nowy komponent do wybranego modelu samochodu. Każda przywołana instancja skryptu (przyporządkowana do pojedynczego pojazdu) potrzebuje do prawidłowego działania odnośników do 8 innych obiektów (rys. 3.23):



Rysunek 3.23: Elementy przyjmowane przez CarSteering.cs

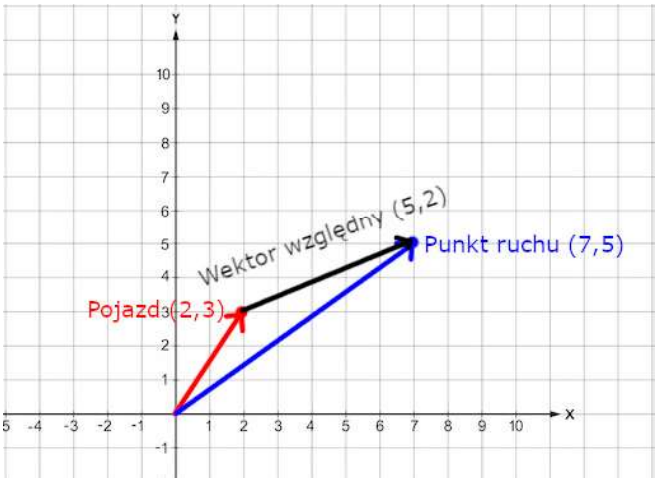
Transform path to już wspomniany obiekt „rodzic" zawierający współrzędne wszystkich punktów ścieżki, którą powinien poruszać się wybrany samochód. Potrzebne są także wszystkie elementy z komponentami *WheelCollider* [47] z podziałem na każde konkretne koło samochodu. Jest to konieczne, aby oddzielić koła przednie odpowiedzialne za sterowanie od tylnych. *carRenderer* jest obiektem odpowiedzialnym za graficzne ukazywanie modelu samochodu na ekranie, a oba elementy typu *Material* są mu przekazywane w celu ukazywania zapalonych oraz zgaszonych tylnych świateł.

Sam skrypt składa się z 7 funkcji :

- ApplySteer()
- Drive()
- CheckWaypointDistance()
- Braking()
- Sensors()
- Start()
- FixedUpdate()

Start() jest funkcją dostarczoną przez *Unity* i uruchamia się ona w momencie inicjalizacji danego skryptu. Zwykle wykorzystywana jest do ustalenia startowych wartości wszystkich zmiennych, ale w tym przypadku wykorzystana jest także do utworzenia tablicy zawierającej kolejne punkty ścieżki. Reszta funkcji wywoływana jest w innej domyślnej dla silnika funkcji czyli FixedUpdate() uruchamianej niezależnie od aktualnej liczby klatek na sekundę programu w domyślnej częstotliwości 50 Hz.

Do kierowania pojazdem służy ApplySteer() (rys. 3.25). Cały ruch odbywa się na płaskiej powierzchni więc przy obliczeniach bierze się pod uwagę jedynie 2 wymiary. Znając aktualne współrzędne samochodu oraz współrzędne kolejnego punktu ścieżki trzeba przejść do zaprezentowania ich jako wektory. Aby przenieść nasz model do pożądanej lokalizacji należało obliczyć wektor względny czyli różnicę pomiędzy tymi dwoma wektorami (rys. 3.24).



Rysunek 3.24: Wyznaczanie wektorów

Do obliczeń wykorzystano funkcje wbudowaną w *Unity*, a dokładniej InverseTransformPoint() [49], która oblicza różnicę pomiędzy obiektem, na którym została wywołana, a obiektem podanym jako dane wejściowe. Posiadając wektor względny można przejść do obliczenia kąta sterowania przednich kół samochodu używając następującego wzoru (3.1):

$$\frac{\text{współrzędna „X” wektora względnego}}{\text{długość wektora względnego}} * \text{maksymalny kąt sterowania} \tag{3.1}$$

Wynikiem dzielenia jest liczba mieszcząca się w zakresie [-1,1], więc aby osiągnąć odpowiedni kąt sterowania mnoży się go przez liczbę reprezentującą maksymalny kąt ustawiony w modelu. Na końcu wykonuje się jedynie interpolację z wykorzystaniem funkcji Mathf.Lerp() [50], dzięki czemu uzyskany ruch skrętu kół jest całkowicie płynny. Całość wygląda w skrypcie w następujący sposób:

```
1 odwołanie
private void ApplySteer()
{
    Vector3 relativeVector = transform.InverseTransformPoint(nodes[currentNode].position);
    float newSteer = (relativeVector.x / relativeVector.magnitude) * maxSteerAngle;

    wheelFL.steerAngle = Mathf.Lerp(wheelFL.steerAngle, newSteer, Time.deltaTime * turnSpeed);
    wheelFR.steerAngle = Mathf.Lerp(wheelFR.steerAngle, newSteer, Time.deltaTime * turnSpeed);
}
```

Rysunek 3.25: Funkcja ApplySteer()

Funkcja Drive() służy do napędzania kół. Dzięki promieniowi koła oraz jego prędkości obrotowej wylicza aktualną prędkość liniową. Każdy samochód posiada ustalone maksymalne wartości osiąganej prędkości oraz momentu przekazywanego na koła, który ma wpływ na przyspieszenie. Oczywiście wartości te należy skalibrować, aby ruch na modelu skrzyżowania odbywał się w tempie podobnym do rzeczywistego. Niestety, w *Unity* żadna z wartości nie jest bezpośrednio odniesiona do prawdziwych jednostek i nie można po prostu przenieść prawdziwych prędkości osiąganych przez samochody. Aby uniknąć tego problemu, w tej pracy zdecydowano się na zmierzenie czasów przejazdów przez skrzyżowanie na każdym z pasów oraz kalibrację wartości na ich podstawie (tablica 3.1). Do tego celu wykorzystano jedno z 15 minutowych nagrań, a następnie wyciągnięto wartość średnią ze wszystkich wykonanych pomiarów. Aby odpowiednio skalibrować model najpierw zmierzone zostały przejazdy rozpędzonych pojazdów, które nie musiały zwalniać przed skrzyżowaniem z powodu czerwonego światła. Po skalibrowaniu prędkości maksymalnej modelu zmierzono także długości przejazdów samochodów zaczynających od całkowitego zatrzymania. Do prawidłowej kalibracji przyspieszenia zawsze wykorzystywano samochody znajdujące się bezpośrednio przy sygnalizatorze, ponieważ tylko one nie nabierały żadnej dodatkowej prędkości przed

wjazdem na skrzyżowanie. Posiadając wszystkie te dane prędkości maksymalne oraz momenty przekazywane na koła zostały tak skalibrowane, aby czasy przejazdu w modelu mieściły się w zakresie ±0.3 sekundy od wyliczonych średnich. Drive() napędza wszystkie koła samochodu z wybranym momentem jeżeli aktualna prędkość jest mniejsza od prędkości maksymalnej oraz auto aktualnie nie hamuje. W przeciwnym wypadku na koła podawany jest moment zerowy.

Tablica 3.1: Tabela zawierająca wszystkie wykonane pomiary (czasy podane w sekundach)

Kierunek jazdy	Prosto		Zachód prawo		Wschód lewo		Południe lewo		Południe prawo	
Pomiar	Vmax	V0	Vmax	V0	Vmax	V0	Vmax	V0	Vmax	V0
1	2,95	6,79	2,89	5,1	6,48	8,17	5,23	8,95	4,91	5,1
2	2,42	5,84	5,11	4,66	7,19	8,11	5,11	7,25	4	4,98
3	2,48	6,54	3,93	4,72	6,15	8,7	5,48	7,45	3,86	5,69
4	2,95	5,62	4,27	5,13		8,3	5,16	7,59	3,59	
5	2,68	6,28	3,48	6,16		8,04	5,7	7,44	3,57	
6	2,29	5,5	3,39	5,45		7,91	5,56	5,89	3,78	
7	3,34	6,5	4,14	6,01		10,27	6,02	8,56	4,77	
8	3,41	6,05	4,33	5,97			5,2		4,26	
9	2,94	6,63	4				5,61		3,99	
10	3,13	5,56	3,81				6,01		4,13	
Średnia	2,86	6,13	3,94	5,4	6,61	8,5	5,51	7,59	4,09	5,26

Funkcja CheckWaypointDistance() odpowiedzialna jest za sprawdzanie odległości między samochodem a kolejnym punktem ścieżki. Kiedy ta odległość spadnie poniżej wybranej wartości (w tym przypadku 2.5) zmienna odpowiedzialna za dobór współrzędnych z tablicy zostanie powiększona o 1 i pojazd zacznie poruszać się w nowym kierunku.

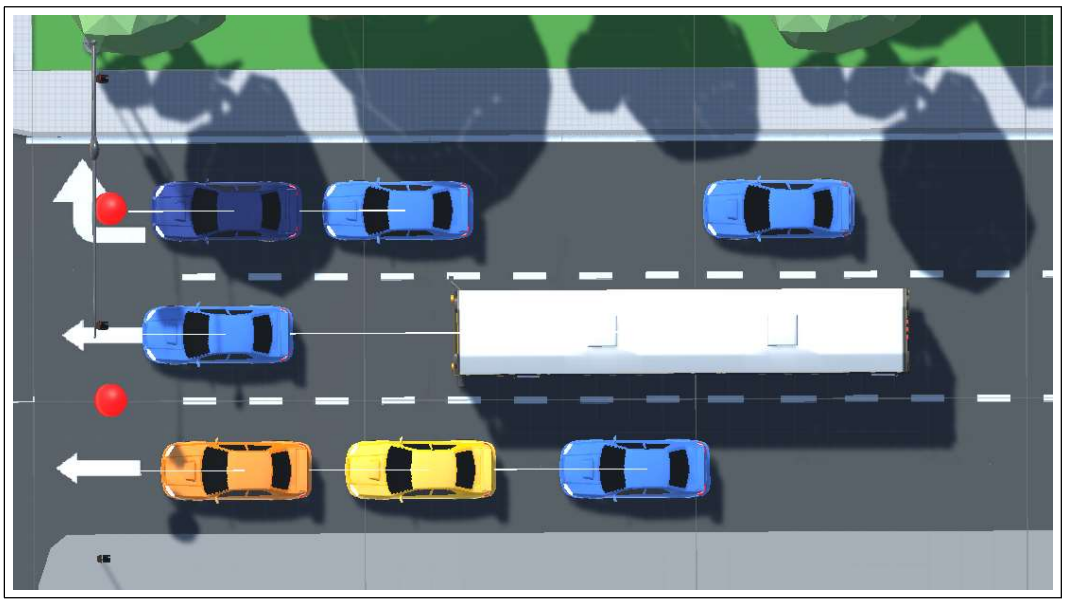
Kolejna funkcja czyli Braking() odpowiedzialna jest za załączanie hamulców. Kiedy odpowiednia zmienna zostanie aktywowana na wszystkich elementach Wheel Collider uruchomią się hamulce, które wyłączą się kiedy zmienna z powrotem przyjmie stan zerowy. Dodatkowo funkcja zmienia także materiał na tylnych światłach samochodu przez co uruchamiają się światła stopu (rys. 3.26). Brak tego elementu jest jedynym co odróżnia skrypt sterujący samochodami od BusSteering.cs sterującego autobusami.

(a) Wyłaczone hamulce **(b)** Załączone hamulce

Rysunek 3.26: Działanie świateł

Ostatnią funkcją w skrypcie CarSteering.cs jest Sensors(). Wykorzystuje ona wprowadzone przez *Unity* elementy typu Raycast [51] do wykrywania przeszkód i sterowania hamulcami. Ich działanie polega na wypuszczeniu promienia na ustaloną odległość i uruchamianiu funkcji kiedy natrafi on na jakąś przeszkodę (rys. 3.27). Aby

Raycast zadziałał, obiekt, na który natrafi musi posiadać jakiś rodzaj elementu Collider. W przypadku Sensors() sprawdzane są także tagi obiektu trafionego promieniem. Są to dodatkowe atrybuty, dzięki którym obiekty można grupować nawet jeżeli mają całkowicie odmienne nazwy. Funkcja musi je sprawdzić, ponieważ samochód nie powinien rozpoczynać hamowania w przypadku natrafienia na elementy usuwające samochody na końcu ich tras oraz liczniki.



Rysunek 3.27: Promienie widoczne w trybie edycji

Oprócz samego uruchamiania hamulców funkcja może także wpływać na ich działanie. Raycasty oprócz samego wskazania obiektu, na który trafiły, dają dostęp także do odległości w jakiej się znajduje. Korzystając z tego dystansu w Sensors() zmieniana jest także siła hamowania, która jest odwrotnie proporcjonalna to odległości od przeszkody. Innym ważnym elementem, który nabiera znaczenia przede wszystkim na pasach prowadzących na wprost (ze względu na wyższe prędkości) jest skracanie długości promieni po pierwszym hamowaniu do zatrzymania. Dzięki wprowadzeniu takiej mechaniki samochody dojeżdżają do siebie na bliższe odległości i nie pozostawiają dużych luk podczas zjazdu ze skrzyżowania. W funkcji Sensors() (rys. 3.28) załączona jest zmienna odpowiedzialna za obsługę tej funkcji, ale sama zmiana długości musi być już uruchamiana w FixedUpdate().

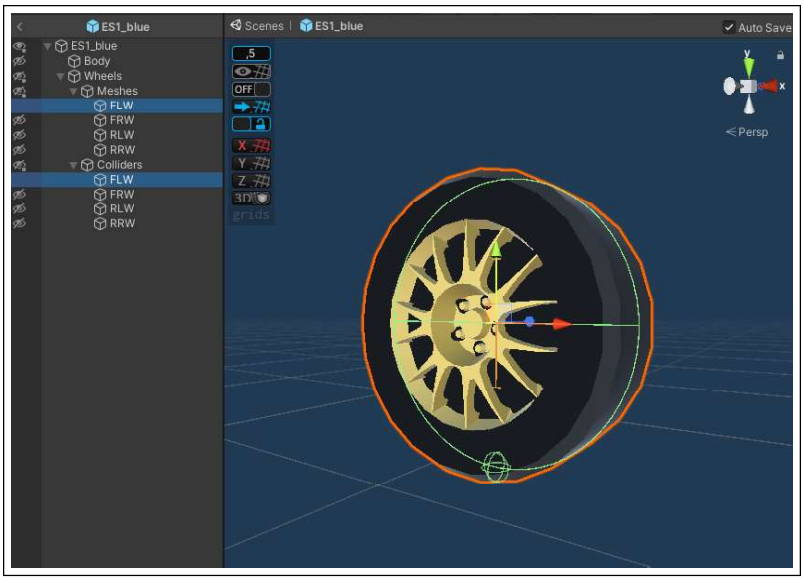
```
1 odwołanie
private void Sensors()
{
    RaycastHit hit;
    Vector3 sensorStartPos = transform.position;
    sensorStartPos.y += 1f;

    if (Physics.Raycast(sensorStartPos, transform.forward, out hit, sensorLength)
    && hit.transform.tag != "destroyers" && hit.transform.tag != "counters" && hit.transform.tag != "rightTurn")
    {
        Debug.DrawLine(sensorStartPos, hit.point);
        maxBrakeTorque = 5500 / Vector3.Distance(transform.position, hit.point);
        isBraking = true;
        if(currentSpeed == 0)
        {
            brakedOnce = true;
        }
    }
    else
    {
        isBraking = false;
    }
}
```

Rysunek 3.28: Funkcja Sensors()

CarWheel.cs

CarSteering.cs odnosi się jedynie do komponentów Wheel Collider, które są dla silnika *Unity* elementami fizycznymi, ale są one całkowicie niewidoczne. Graficzne modele samochodów posiadają osobne kół, ale przy zastosowaniu tylko jednego skryptu pozostaną one całkowicie nieruchome względem reszty. Aby modele poruszały się w sposób odpowiadający sterowanemu pojazdowi, stworzony został krótki skrypt CarWheel.cs. Należy dodać go do każdego z 4 kół i powiązać je z odpowiadającym im Wheel Colliderami (rys. 3.29). Dzięki temu zabiegowi koła zmieniają pozycję, skręcają oraz kręcą się zgodnie z symulacją zapewnianą przez poprzedni skrypt.



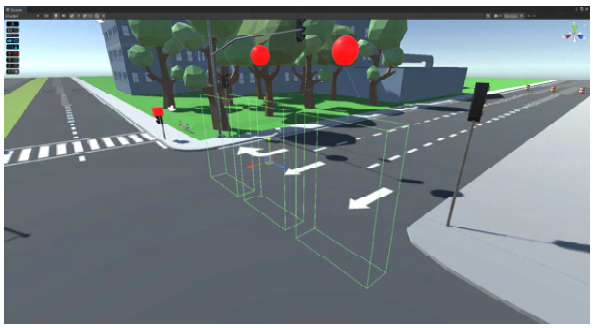
Rysunek 3.29: Graficzny model koła oraz przypisany mu Wheel Collider widoczne w hierarchii samochodu

Counter.cs

Aby umożliwić dynamiczne sterowanie skrzyżowaniem za pomocą algorytmu konieczne jest utworzenie liczników, które będą śledzić liczbę pojazdów znajdującym się na każdym z pasów. Implementacja tej funkcji ma miejsce w pliku Counter.cs. Skrypt ten składa się z 2 zmiennych, które redukowane będą podczas zjeżdżania aut z pasów ruchu oraz jednego Collidera (rys. 3.30 b). Komponent ten w przeciwieństwie do kół czy modeli samochodów nie jest elementem fizycznym, ponieważ został ustawiony w tryb Trigger. Silnik fizyczny ignoruje wtedy całkowicie wszelkie zderzenia, ale programista nadal ma dostęp do funkcji wykrywających początek czy koniec kolizji. Zgodnie z tym Counter.cs wykorzystuje funkcję OnTriggerEnter() (rys. 3.30 a) aby wykryć przejeżdżający pojazd, a następnie według tagu decyduje czy należy z pasa ruchu odjąć samochód czy autobus. Warto już na tym etapie zaznaczyć, że dostęp do zmiennych będą miały także obiekty wprowadzające nowe pojazdy na skrzyżowanie więc liczby te zawsze będą zgodne ze stanem faktycznym.

```
Unity Message | Odwołania: 0
private void OnTriggerEnter(Collider collision)
{
    if (collision.gameObject.tag == "cars")
    {
        carsCount--;
    }
    else if (collision.gameObject.tag == "buses")
    {
        busesCount--;
    }
}
```

(a) Funkcja OnTriggerEnter()

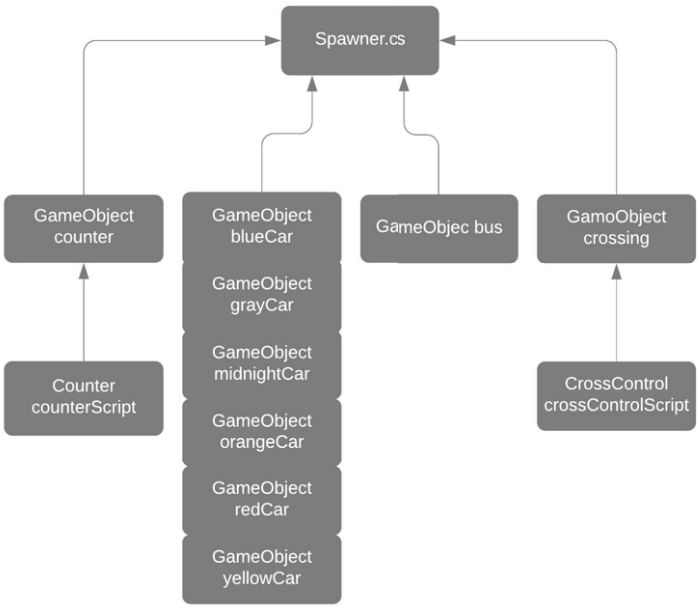


(b) Liczniki znajdujące się przy wjeździe na skrzyżowanie

Rysunek 3.30: Elementy działania Counter.cs

Spawner.cs

Bardzo ważnym elementem projektu jest wspomniany powyżej skrypt Spawner.cs umieszczający nowe pojazdy do modelu. Podobnie do elementu odpowiedzialnego za sterowanie do działania potrzebne są następujące zewnętrzne obiekty (rys. 3.31):



Rysunek 3.31: Elementy przyjmowane przez Spawner.cs

Poprzez GameObject counter otrzymywany jest dostęp do skryptu zawierającego ilości pojazdów na pasie, a poprzez GameObject crossing przekazuje się główny skrypt odpowiedzialny za sterowanie całym skrzyżowaniem. Należy także przekazać wszystkie obiekty, które będą wprowadzane do modelu. Muszą one być wcześniej odpowiednio przygotowane dla każdego pasu ruchu oraz zapisane jako assety. Z tego względu nie znajdują się one na stałe w hierarchii projektu. Spawner.cs jest dosyć złożonym skryptem i składa się z następujących funkcji:

- Start()
- Update()
- StartReading()
- WaitForSpawn()
- SpawningCar1()
- ChangeToBus()
- SpawningBus()
- ChangeFromBus()
- SpawningCar2()

W funkcji inicjalizacyjnej tworzone są dwie listy. Po przetworzeniu nagrań system wizyjny przekazuje do modelu informacje w postaci 3 cyfrowych kodów dla każdego pasa ruchu i dlatego pierwsza z list jest zapełniana 3 cyframi wskazującymi ile pojazdów ma się pojawić, a do drugiej dodawane są wszystkie modele samochodów. Ważne jest także wywołanie SpawningCar1(), SpawningBus() oraz SpawningCar2() poprzez funkcję InvokeRepeating(). Oprócz elementów do wywołania przyjmuje ona wartości liczbowe i podczas działania programu wywołuje wybrane funkcję w wyznaczonych interwałach czasowych. Oznacza to, że funkcje bezpośrednio odpowiedzialne

za wprowadzanie samochodów będą uruchamiane raz na ustaloną liczbę sekund.

Update() sprawdza jedynie czy zmienna odpowiedzialna za czytanie kodu jest w stanie TRUE. Jeżeli tak to zmienna ustawiana jest na FALSE oraz uruchamiana jest funkcja WaitForSpawn(). Dzięki temu, że jest ona typu *IEnumarator* wywoływana jest poprzez StartCoroutine(). Oznacza to, że będzie działać równolegle do reszty programu co konieczne jest w przypadku wprowadzania opóźnień czasowych, ponieważ jeżeli taka operacja wykonywana była w standardowy sposób cała aplikacja zamierałaby na czas oczekiwania.

StartReading() jest bardzo prostą funkcją i służy jedynie, aby zewnętrze skrypty mogły rozpocząć procedurę sczytywania kodu podanego przez system wizyjny.

Sama funkcja WaitForSpawn() (rys. 3.32) rozpoczyna się od polecenia czekania aż wszystkie poprzednie procedury wprowadzające nowe pojazdy na skrzyżowanie zostaną zakończone. Taki zabieg jest konieczny, ponieważ w przeciwnym wypadku program mógłby próbować zainicjować dwie procedury jednocześnie, co skończyłoby się błędami. Następnie na podstawie numeru identyfikacyjnego nadanego każdemu obiektowi ze skryptem Spawner.cs pobierany jest odpowiedni kod z listy skryptu CrossControl.cs. Kod ten jest typu *string* więc należy przekonwertować go na liczbę całkowitą, aby następnie uzupełnić listę spawnNumber, z której będą pobierane wartości do inicjalizowania kolejnych obiektów. Po tych zabiegach rozpoczęta jest procedura wprowadzania pojazdów poprzez zmianę odpowiedniej zmiennej.

```
1 odwołanie
IEnumerator WaitForSpawn()
{
    yield return new WaitUntil(() => (isSpawningCar1 == false && isSpawningBus == false && isSpawningCar2 == false));
    spawnCode = crossControlScript.spawnCodes[id];
    for (int i = 0; i < spawnCode.Length; i++)
    {
        spawnNumber[i] = (int)(spawnCode[i] - '0');
    }
    spawnCode = null;
    if (spawnNumber != null)
    {
        isSpawningCar1 = true;
    }
}
```

Rysunek 3.32: Funkcja WaitForSpawn()

Cały proces odbywa się na podstawie wspomnianych 3-cyfrowych kodów, ponieważ taki system umożliwia umieszczenie autobusu pomiędzy samochodami w wymaganym miejscu. Pierwsza i ostatnia cyfra kodu odpowiedzialne są za liczby samochodów, a cyfra środkowa zawsze za ilość autobusów, przy czym cały kod czytany jest od początku. Przykładowo kod „312” spowodowałby pojawienie się 3 samochodów, a następnie 1 autobusu śledzonego przez 2 kolejne samochody. Jeżeli chcielibyśmy aby autobus był pierwszy w kolejce wtedy kod przyjąłby postać np. „015”, a jeżeli miałby być ostatni kod wyglądałby np. „310”. W przypadku pasów, na których nie pojawiają się autobusy, wystarczy używać jedynie pierwszej cyfry kodu do poprawnego działania.

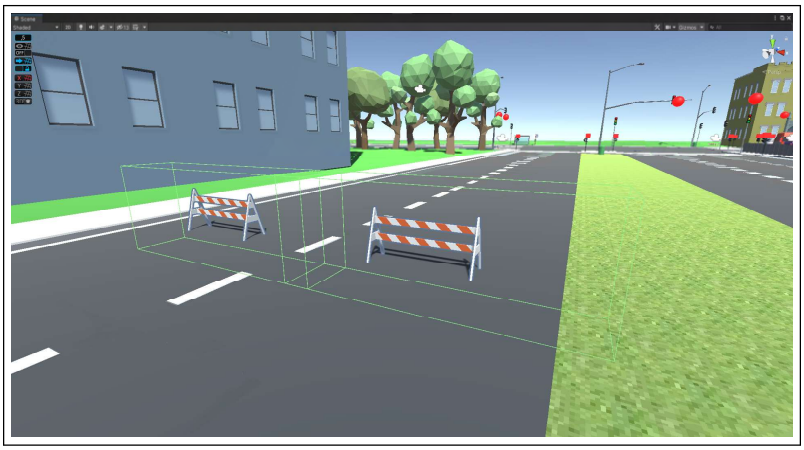
Każdy z 3 etapów procedury posiada odpowiednia zmienną odpowiedzialną za jego aktywację. Kiedy zostanie uruchomiony, funkcja sprawdza odpowiednią liczbę z kodu i działa zależnie od jej wartości. Każdy pas w symulacji ma ograniczoną liczbę miejsca między punktem wprowadzania pojazdów a samym skrzyżowaniem, więc jeżeli liczba aut do wprowadzenia jest większa od 6 to program ustawia ją na liczbę maksymalną czyli 6. Jeżeli liczba mieści się w zakresie (0,6] program przechodzi do przekazania odpowiedniego pojazdu. W przypadku samochodów zaczyna od wylosowania liczby na podstawie, której wybierze kolor wprowadzanego samochodu. Na podstawie tego numeru z listy modeli wybierane jest auto i jego instancja pojawia się w wybranym wcześniej punkcie (rys. 3.33). To także w tym miejscu do skryptu Counter.cs przesyłana jest informacja o pojawieniu się kolejnego pojazdu i liczba przyjęta z kodu zmniejszana jest o 1. Kiedy wszystkie pojazdy już się pojawią i liczba ta osiągnie 0 załączany jest kolejny etap procedury jedną funkcją ChangeToBus(), albo ChangeFromBus(). Wprowadzają one krótkie opóźnienie pomiędzy etapami, które jest konieczne, ponieważ modele autobusów są zdecydowanie

większe od samochodów i przyspieszają w inny sposób. W przypadku braku opóźnienia autobusy mogłyby swoim modelem ingerować w pojawiające się samochody wypychając je z przypisanego pasa ruchu.

Rysunek 3.33: Graficzne reprezentacje punktów wprowadzania pojazdów

Destroyer.cs

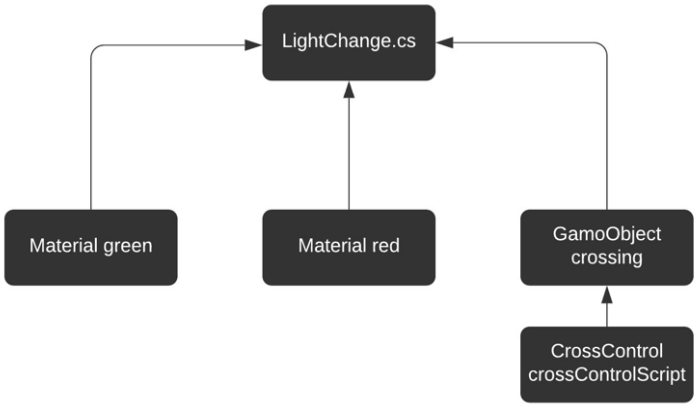
Każda istniejąca instancja (czyli pojedynczy obiekt) pojazdu potrzebuje mocy obliczeniowej komputera do odpowiedniego działania, więc aby program działał z zadowalającą prędkością należy usuwać pojazdy, które już zjechały ze skrzyżowania i nie są potrzebne do dalszej symulacji. Do tego celu służy skrypt Destroyer.cs. Jego działanie jest bardzo proste i mieści się w 3 linijkach kodu. Po wykryciu kolizji (rys. 3.34) z innym obiektem ma on za zadanie sprawdzić tagi tego obiektu. Jeżeli wynika z nich, że jest to samochód albo autobus obiekt natychmiast usuwany jest z symulacji.



Rysunek 3.34: Graficzne reprezentacje punktów usuwania pojazdów wraz z widocznymi colliderami

LightChange.cs

Obsługą obiektów reprezentujących sygnalizację świetlną zajmuje się kod z pliku LightChange.cs. Nie jest to bardzo rozbudowany skrypt, ale do działania nadal potrzebuje 3 zewnętrznych elementów (rys. 3.35):



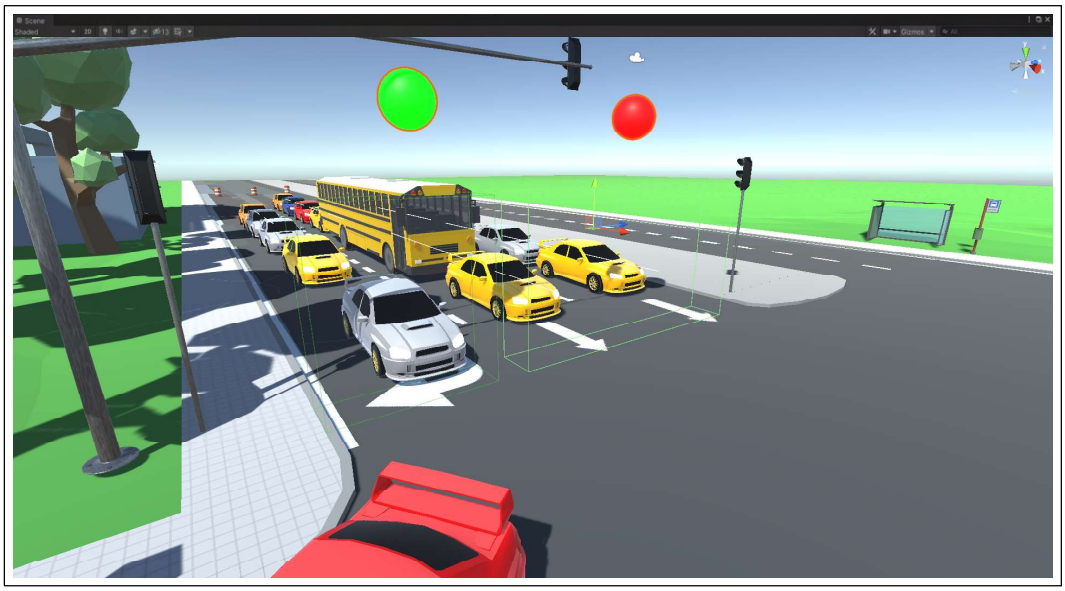
Rysunek 3.35: Elementy przyjmowane przez LightChange.cs

Tak jak w poprzednim skrypcie konieczne jest odwołanie do obiektu zawierającego element sterujący skrzyżowanie, ponieważ to tam przechowywane są zmienne odpowiedzialne za zachowanie sygnalizacji świetlnej. Dodać należy także materiały zawierające kolory sygnalizacji świetlnej czyli zielony oraz czerwony. Kolor materiału można zmieniać także na bieżąco w kodzie, ale tworząc materiały z poziomu edytora *Unity* można osiągnąć lepsze wizualnie rezultaty, które następnie zostaną powiązane z każdą instancją sygnalizacji. Bardzo ważne jest także ustalenie numeru identyfikacyjnego każdego sygnalizatora, aby mógł on korzystać z odpowiedniej zmiennej do ustalaniu stanu świateł. Sam kod składa się z 4 głównych funkcji

- Start()
- Update()
- GreenLight()
- RedLight()

Podczas inicjalizacji kodu pobierany jest skrypt sterujący skrzyżowaniem oraz Collider dodany do sygnalizacji, który będzie służył za element zatrzymujący samochody. Wszystkie pojazdy wykrywają go poprzez wspomniany już system Raycastów co umożliwia im dynamiczne reagowanie na zmieniający się stan sygnalizatora.

W funkcji Update() sprawdzany jest numer identyfikacyjny sygnalizatora, aby na jego podstawie śledzić stan zmiennej odpowiedzialnej za dany fragment skrzyżowania (rys. 3.37 a). Jeżeli jest ona prawdziwa uruchamiane jest światło zielone a Collider jest wyłączany umożliwiając przejazd samochodów. W przeciwnym wypadku Collider jest z powrotem aktywnym elementem, a światło świeci się na czerwono (rys. 3.36).



Rysunek 3.36: Sygnalizacje świetlne z załączonym oraz wyłączonym elementem kolizyjnym

Funkcje GreenLight() (rys. 3.37 b) oraz RedLight() służa do zmiany materiału danego sygnalizatora. Nie są one jednak identyczne, ponieważ to na poziomie sygnalizatora zaimplementowany jest czas międzyzielony. Aby podczas zmian stanów sygnalizatorów samochody wjeżdżające na skrzyżowanie nie uderzyły tych zjeżdżających należy wprowadzić opóźnienie w zapaleniu kolejnego zielonego światła. Z tego powodu GreenLight() jest funkcją typu IEnumarator co daje jej dostęp do możliwości działania równoległego z resztą programu i odczekania ustalonego okresu przed egzekucją reszty kodu.

```
case 6:
    if (crossControlScript.southLeft)
    {
        if (flag == false)
        {
            StartCoroutine(GreenLight());
            flag = true;
        }
    }
    else
    {
        flag = false;
        StopAllCoroutines();
        RedLight();
    }
    break;
```

(a) Instrukcja typu case sprawdzająca numer identyfikacyjny

```
IEnumerator GreenLight()
{
        yield return new WaitForSeconds(5f);
        this.GetComponent<Renderer>().sharedMaterial = green;
        lightCollider.enabled = false;
}
```

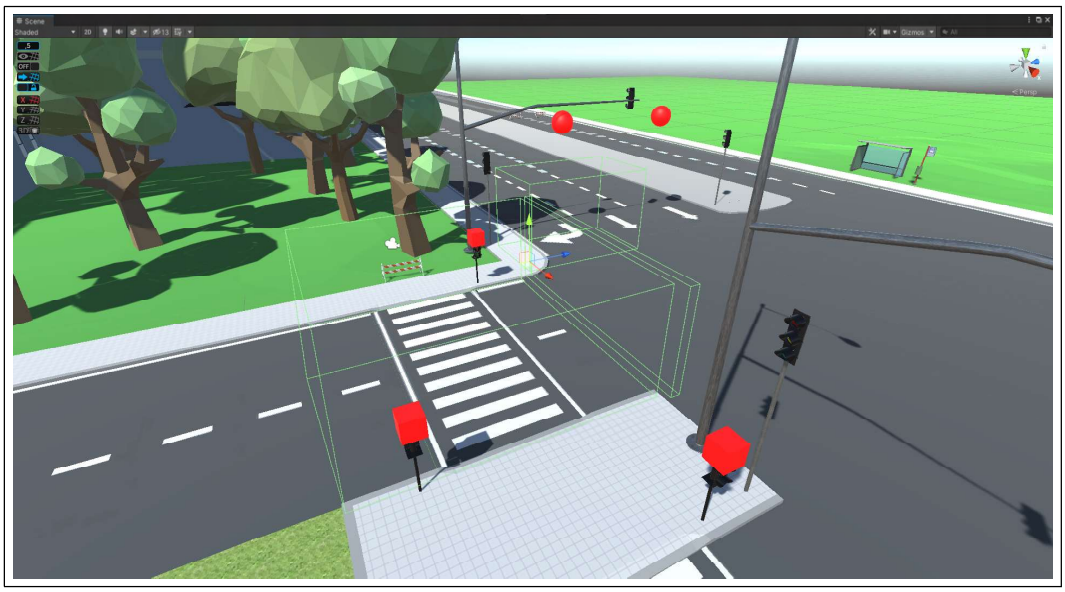
(b) Funkcja załączająca zielone światło

Rysunek 3.37: Elementy działania LightChange.cs

RightTurn.cs

Prawoskręt z ulicy Baraniaka jest realizowany za pomocą sygnalizatora S-2 czyli potocznie nazywanej „zielonej strzałki”. Oznacza to, że pozwolenie na przejazd jest warunkowe i sygnalizacja dla pieszych również może dawać zielone światło. W takim przypadku kierowca powinien się zatrzymać i przepuścić wszystkich pieszych przed kontynuowaniem dalszej jazdy. Aby osiągnąć tą funkcjonalność sygnalizacji konieczny był dodatkowy kod zawarty w skrypcie RightTurn.cs. Składa się on z 3 elementów typu Collider (rys. 3.38). Pierwszy z nich obejmuje przejście dla pieszych i jego zadaniem jest wykrywanie pieszych. Do tego celu wykorzystamy funkcję *Unity* czyli OnTriggerStay(). Aktywuje ona zawarty w niej kod nie tylko w momencie wykrycia kolizji, ale przez cały okres jej trwania. W tym przypadku użyta jest do aktywowania 2 pozostałych Colliderów których zadaniem jest wyhamowanie samochodów przed przejściem. Blokada pojazdów jest zrealizowana w dwóch etapach, ponieważ pierwszy z elementów zatrzymuje samochody przy wjeździe na skrzyżowanie i nie zadziałałby dla samochodu, który już zdążył go minąć. Aby obsłużyć takie przypadki wprowadzony został także drugi element, który zapew-

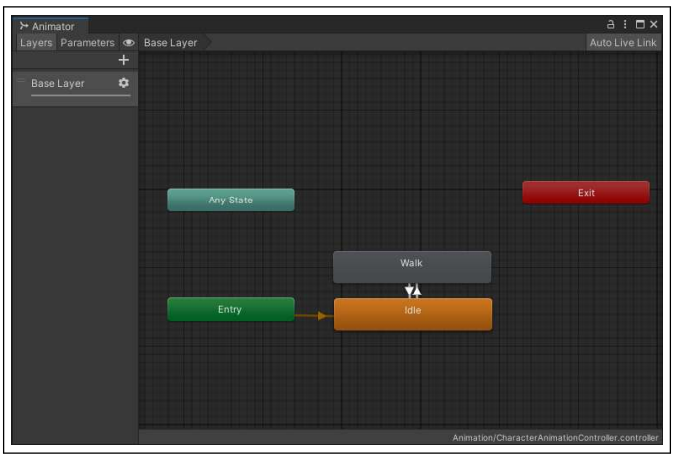
nił całkowite bezpieczeństwo wirtualnych pieszych. Aby dezaktywować blokadę użyto funkcji OnTriggerExit(), która aktywuje kod w niej zawarty w momencie wykrycia końca kolizji z pieszymi. Kod ten dezaktywuje Collidery i umożliwia w ten sposób dalszy przejazd samochodom.



Rysunek 3.38: Wszystkie 3 elementy kolizyjne przy prawoskręcie

Pedestrians.cs

Zachowanie pieszych zostało opisane w dużo prostszy sposób niż pojazdów. Ich ruch jest na tyle nieskomplikowany, że nie było konieczności sięgać po gotowy kontroler postaci. Szybszym rozwiązaniem okazało się napisanie własnego kontrolera, którego całe zachowanie zostało zawarte w dwóch podstawowych funkcjach Start() oraz FixedUpdate(). Postacie wykorzystane w modelu wymagały jednak przygotowania prostego schematu animacji. W *Unity* są one obsługiwane dzięki systemowi Mecanim (rys. 3.39), który działa na zasadzie maszyny stanów. W modelu użytkowane są 2 animacje (animacja chodzenia oraz animacja bezczynności), więc maszyna posiada jedynie 3 stany - po jednym na animacje oraz stan wejściowy. Tworzone są także zmienne, za pomocą których można kontrolować przechodzenie pomiędzy wszystkimi stanami. Tak przygotowany kontroler animacji dodano do komponentu Animator każdej utworzonej postaci.



Rysunek 3.39: System Mecanim widoczny w *Unity*

Do sterowania postacią wykorzystano komponenty Rigidbody oraz Collider w podobny sposób jak wspomniano wcześniej. Podczas inicjalizacji skryptu tworzone są zmienne zawierające bryłę sztywną oraz animator pieszego. Cała reszta kodu została napisana w funkcji FixedUpdate(). Tak jak inne elementy skrzyżowania, w celu zmniejszenia liczby plików w projekcie i obsłużenia ruchu wszystkich pieszych jednym skryptem wykorzystano

numery identyfikacyjne wybierające ruch w odpowiednim kierunku. Piesi, tak jak pojazdy, wykorzystują Raycasty w celu zatrzymywania się przed przejściem podczas czerwonego światła na sygnalizacji świetlnej (rys. 3.40). Została zaimplementowana logika pilnująca, żeby piesi nie zatrzymywali się podczas spotkania postaci z przeciwka, znaków będących jednocześnie elementami wprowadzającymi nowych pieszych jak i usuwającymi pieszych, którzy pokonali całe przejście. Wybór kierunku postaci odbywa się poprzez ustalenia wektora ruchu. Skrzyżowanie zostało zbudowane równoległe do osi współrzędnych więc wszystkie 2 kierunki oraz 4 zwroty są dosyć proste i wyglądają w następujący sposób:

- Vector3 xPlus = new Vector3(1, 0, 0);
- Vector3 xMinus = new Vector3(-1, 0, 0);
- Vector3 zPlus = new Vector3(0, 0, 1);
- Vector3 zPlus = new Vector3(0, 0, 1);

Rysunek 3.40: Poruszający się oraz stojący piesi

Sam ruch odbywa się za pomocą funkcji MovePosition() [52]. Na jej wejście należy podać pozycję, na którą należy przemieścić obiekt. Jej kod wywoływany jest w regularnych interwałach czasowych, więc obliczanie nowej pozycji musiało odbywać się w sposób ciągły i automatyczny co można było osiągnąć poprzez użycie wzoru (4.2):

$$\text{body.position} + \text{Vector3.normalized} * \text{speed} * \text{Time.fixedDeltaTime} \tag{3.2}$$

gdzie:

$body.position$ – aktualna pozycja postaci
$Vector3.normalized$ – znormalizowany wektor kierunku i zwrotu ruchu
$speed$ – zmienna pozwalająca na regulację prędkości ruchu
$Time.fixedDeltaTime$ – różnica czasowa pomiędzy wywoływaniem FixedUpdate()

Prędkości ruchu pieszych musiały odpowiadać rzeczywistym tak samo jak w przypadku pojazdów. Z tego względu wykonano procedurę kalibracji w podobny sposób jak wcześniej. Z 15-minutowego filmu wyciągnięto wszystkie przypadki przejść przez pasy, w których wyraźnie widać moment wejścia i zejścia z jezdni. Następnie podzielono je na pokonanie 2 oraz 3 pasów samochodowych oraz obliczono średnią wartość ze wszystkich pomiarów czasowych (tablica 3.2). Wartości prędkości przypadające na pokonanie pojedynczego pasa różniły się więc model skrzyżowania został tak skalibrowany aby mieścić się pomiędzy dwoma wyznaczonymi średnimi czasami.

Tablica 3.2: Pomiary czasów przejść pieszych w sekundach

Pomiar	2 pasy	3 pasy
1	6,66	8,95
2	7,07	8,69
3	6,65	7,79
4	7,12	7,59
5	5,11	11,02
6	7,71	7,6
7	7,24	6,85
8	5,89	6,68
9		9,8
10		9,29
Średnia	6,68	8,43

PedestrianSpawning.cs

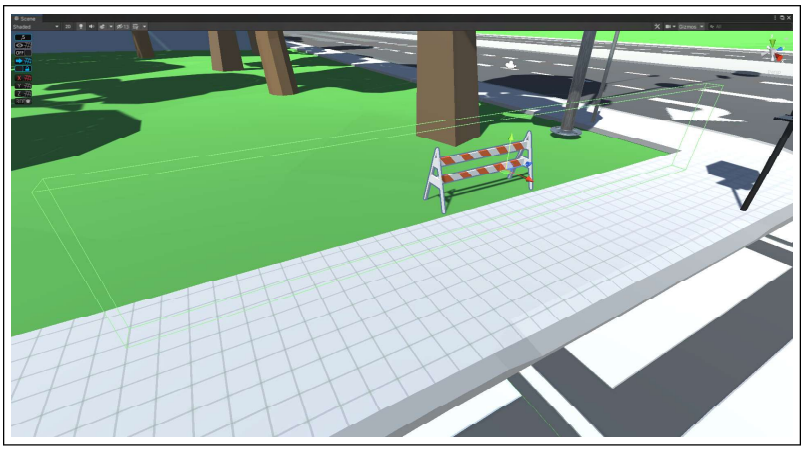
Plik odpowiedzialny za wprowadzanie pieszych do symulacji został w dużej mierze oparty na systemie wykonującym to samo zadanie z pojazdami. Posiada on 2 modele postaci do dyspozycji więcej od Spawner.cs, ale nie musi posiadać połączenia z żadnym dodatkowym licznikiem kontrolującym ilość pieszych, ponieważ ta funkcjonalność jest wykonywana wewnętrznie. Nadal jednak potrzebne jest połączenie ze skryptem kontrolującym skrzyżowanie. Plik ten zawiera 4 funkcje:

- Start()
- Update()
- StartReading()
- Spawning()

Podczas inicjalizacji pobierany jest skrypt CrossControl, ponieważ wprowadzanie pieszych także oparte jest na numerach identyfikacyjnych. Wszystkie przygotowane modele pieszych dodawane są także do odpowiedniej listy, która zostanie wykorzystana przy losowaniu jednego z nich. Po raz kolejny wykorzystana jest także funkcja InvokeRepeating(), przy pomocy której wywoływana cyklicznie będzie funkcja Spawning().

W funkcji Update() ciągle sprawdzona jest zmienna odpowiedzialna za rozpoczęcie odczytywania kodu. Jeżeli zostanie aktywowana poprzez funkcje StartReading() to kod jest pobierany oraz konwertowany z zmiennej typu string na typ int. W przypadku pieszych nie ma potrzeby korzystania z 3-cyfrowego kodu więc ostatecznie pobrana zostaje tylko jedna liczba.

Funkcja Spawning() również oparta jest na jej odpowiedniku do wprowadzania samochodów. Główną różnicą jest implementacja różnych miejsc pojawianie się postaci. Od punktu centralnego obiektu został wyznaczony prostokąt, którego wierzchołki wyznaczają granice, w których może pojawić się pieszych. Na początku procesu wprowadzania nowej instancji pieszego jego współrzędne są losowane. Następnie, podobnie jak przy pojazdach, losowany jest także model z listy przygotowanych. Sano wprowadzenie odbywa się przy pomocy komendy Instantiate(), do której należy podać inicjowany obiekt, jego pozycję oraz rotację. Po wykonaniu tych czynności liczba w liczniku zostaje zwiększona o 1, a cyfra z kodu zmniejszona. Tutaj także zawarte zostało zabezpieczenie, aby jednorazowo nie pojawiła się zbyt duża liczba pieszych. Warto zaznaczyć, że dla pieszych jeden obiekt graficzny reprezentuje ich miejsce wprowadzenie oraz jednocześnie miejsce usuwania pieszych nadchodzących z przeciwnej strony skrzyżowania (rys. 3.41).



Rysunek 3.41: Obiekt wprowadzający oraz usuwający pieszych z widocznym elementem kolizyjnym

PedestrianDestroyer.cs

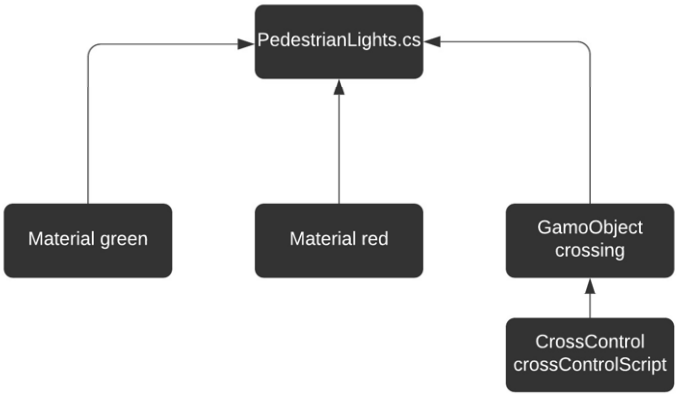
Plik usuwający pieszych odnosi się jedynie do PedestrianSpawning.cs, aby mieć możliwość zmniejszenia liczby pieszych kiedy zejdą ze skrzyżowania. Kod jest aktywowany poprzez OnCollisionEnter() (rys. 3.42), czyli w momencie wykrycia kolizji i następnie sprawdzany jest tag wykrytego obiektu. Elementy niszczące także posiadają numer identyfikacyjny i na jego podstawie usuwają z modelu tylko pieszych wprowadzanych po przeciwnej stronie przejścia przy którym się znajdują. Taki zabieg jest konieczny, ponieważ w przeciwnym przypadku usuwani byliby piesi, którzy jeszcze nie zdążyli przejść na drugą stronę.

```
Unity Message | Odwołania: 0
void OnCollisionEnter(Collision coll)
{
    switch (id)
    {
        case 0:
            break;
        case 1:
            if (coll.gameObject.tag == "pedestrianSW")
            {
                Destroy(coll.gameObject);
            }
            break;
```

Rysunek 3.42: Fragment kodu sprawdzający numer identyfikacyjny oraz tag pieszego

PedestrianLights.cs

Obsługa sygnalizacji dla pieszych wymaga przekazania tych samych obiektów co obsługa samochodów (rys. 3.43):



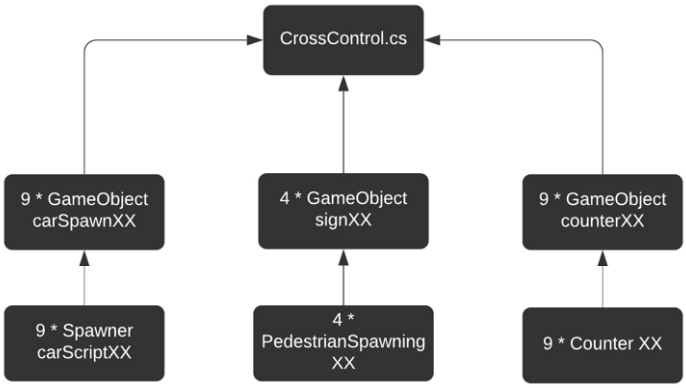
Rysunek 3.43: Elementy przyjmowane przez PedestrianLights.cs

Cztery z funkcji znajdujących się w pliku posiadają te same nazwy oraz działanie co w przypadku LightChange.cs. Jedyną zmianą jest wprowadzenie dodatkowej funkcjonalności wykrywania kolizji. W przeciwieństwie do samochodów sygnalizacja świetlna dla pieszych znajduje się po obu stronach przejścia (rys. 3.44), więc dwa numery identyfikacyjne odnoszą się do jednej zmiennej w skrypcie CrossControl.cs. Kiedy już oba sygnalizatory działają razem pozostaje problem przejścia podczas zmiany kolory sygnalizacji na czerwony. Pieszy, który już znajduje się na pasach musi mieć możliwość zejścia z nich nawet pomimo załączonej blokady. Aby to rozwiązać wykorzystana została funkcja OnCollisionEnter(), która na podstawie numeru identyfikacyjnego danego sygnalizatora sprawdza czy tag pieszego zgadza się z punktem wprowadzania naprzeciwko. Jeżeli tak, to funkcja IgnoreCollision() ignoruje skutki kolizji i pozwala pieszemu zejść z jezdni.

Rysunek 3.44: Sygnalizatory dla pieszych w obu stanach wraz z widocznymi elementami kolizyjnymi

CrossControl.cs

CrossControl.cs jest plikiem odpowiadającym za sterowanie całym modelem skrzyżowania. Do wykonywania tego zadania potrzebuje znaczącej ilości odnośników do wszystkich informacji o ruchu w symulacji. Sumarycznie skrypt ten należy odnieść do 22 innych obiektów (rys. 3.45):



Rysunek 3.45: W miejscu XX znajduje się odnośnik do danego pasa ruchu

Powiązanie do wszystkich 9 obiektów wprowadzających pojazdy jest konieczne, ponieważ trzeba przez nie przyjąć komponenty ze skryptami Spawner.cs. Posiadając te pliki możliwe jest zainicjowanie metody StartReading() inicjując proces czytania kodu w każdym z punktów. Aby mieć możliwość śledzenia liczby należało dodać również wszystkie liczniki oraz ich skrypty Counter.cs. Informacja ta była potrzebna w algorytmie sterującym skrzyżowaniem, który również został zaimplementowany w tym pliku. Obsługa pieszych nie wymagała tylu dodatkowych elementów, ponieważ wystarczyło przyjąć jedynie 4 punkty wprowadzające. Dostęp do skryptów PedestrianSpawning.cs zapewnia dostęp zarówno do metody wprowadzającej jak i liczby pieszych czekających na

przejście.

Skrypt posiada zmienne odpowiedzialne za sterowanie skrzyżowaniem i to do nich odnoszą się wszystkie sygnalizacje. Dodatkowo program może działać w kilku różnych trybach i ich ostateczny wybór także dokonuje się w tym pliku. Po pierwsze sterowanie może być wykonywane manualnie, przez użytkownika, lub automatycznie przez algorytm i decyduje o tym zmienna automaticControl. Dodatkowo istnieje także wybór sposobu pobierania danych od systemu wizyjnego poprzez zmienną automaticRead. Ta funkcjonalność także może odbywać się automatycznie, czyli w określonych interwałach czasowych, lub może być wywoływana w dowolnym momencie przez użytkownika. Zmienne te mogą być zmieniane na ekranie konfiguracyjnym, który musiał zostać stworzony jako dodatkowa scena w projekcie. Obiekty oraz skrypty nie mogą być przekazywane między scenami więc zamiast odnoszenia się do odpowiednich obiektów obie zmienne są typu static. Oznacza to, że istnieją tylko w pojedynczej instancji, ale może się do nich odnieść każdy skrypt w projekcie, nawet taki wywoływany poprzez inną scenę.

Pomimo obsługiwania takiej dużej liczby obiektów CrossControl.cs składa się jedynie z 5 funkcji:

- Start()
- ReadFile()
- GetDigit()
- Update()
- AutomaticCrossControl()

przy czym ostatnia z nich odpowiada za algorytm sterujący skrzyżowaniem i zostanie omówiona w następnym rozdziale.

Podczas uruchamiania skryptu oprócz zmiennych koniecznych do działania inicjalizowana jest także ścieżka, prowadząca do folderu z danymi aplikacji. W tym miejscu program oczekuje pliku tekstowego z informacjami od systemu wizyjnego. Jego nazwa domyślnie ustawiona jest na „test.txt”, ale może zostać zmieniona na ekranie konfiguracji poprzez zmienną typu static. Program podczas startu tworzy także wzorową listę typu string zapełnioną samymi „0”. Podczas działania zostaje ona zapełniona odpowiednimi kodami odebranymi od systemu.

Funkcja ReadFile() (rys. 3.46 a) odpowiada za czytanie pliku tekstowego oraz zapełnienie zainicjalizowanej listy odpowiednimi kodami. Do wykonania tego zadnia wykorzystana została klasa StreamReader [53] wbudowana w język C. Czyta ona wyznaczony plik linia po linii przekazując każdą z nich do odpowiedniego elementu listy z kodami.

```
1 odwołanie
void ReadFile()
{
    using(StreamReader sr = new StreamReader(path))
    {
        for (int i = 0; i < 13; i++)
        {
            spawnCodes[i] = sr.ReadLine();
        }
    }
}
```

(a) Funkcja ReadFile()

```
1 odwołanie
public int GetDigit(int seconds)
{
    string temp = seconds.ToString();
    int[] array = new int[temp.Length];
    for (int i = 0; i< array.Length; i++)
    {
        array[i] = (int)(temp[i] - '0');
    }
    return array[array.Length-1];
}
```

(b) Funkcja GetDigit()

Rysunek 3.46: Elementy działania CrossControl.cs

Aby umożliwić automatyczne czytanie pliku z danymi od systemu wizyjnego należało ten proces zsynchronizować. Skrypty zapisujące dane do pliku oraz odczytujące je nie mogą mieć jednoczesnego dostępu, ponieważ jest to niemożliwe i spowoduje błędy w działaniu. Aby temu zapobiec postanowiono wprowadzić synchronizację

na podstawie cyfry jedności sekund czasu systemowego. Do ich uzyskania służy funkcja GetDigit() (rys. 3.46 b). Na wejściu przyjmuje ona liczbę, która zostaje przekonwertowana na string oraz podana do zmiennej tymczasowej. Dzięki tej konwersji i użyciu odpowiedniej pętli można rozpisać osobno każdą z cyfr początkowej liczby, przekonwertować z powrotem na typ int oraz zapisać do listy. Na koniec wystarczy jedynie wywołać odpowiedni element z listy, aby uzyskać dostęp do oczekiwanej liczby jedności.

W funkcji Update() pobierane są sekundy z czasu systemowego komputera, na którym uruchomiony jest program. Następnie przekazywane są one do GetDigit() i tworzona jest zmienna zawierająca jedynie aktualną cyfrę jedności. Jeżeli uruchomione jest automatyczne odczytywanie, w każdym momencie kiedy jedności zmienią się z 0 na 1 lub 5 na 6 (czyli kiedy liczba sekund zmieni się z 0 na 1, 5 na 6, 10 na 11, 15 na 16 itd.) załączona zostanie zmienna „read". Kiedy jest ona w stanie *TRUE* wywołany zostanie skrypt ReadFile() oraz metody StartReading() na każdym elemencie, który posiada nowe obiekty (pieszych lub pojazdy) do wprowadzenia na skrzyżowanie. Aby czynność ta wykonała się tylko jeden raz na jej koniec „read" zmieniane jest z powrotem na *FALSE*. Dodatkowo Update() sprawdza czy naciśnięty został przycisk „Escape" co wprowadza program do głównego menu oraz uruchamia AutomaticCrossControl().

GUI (Interfejs graficzny użytkownika)

Aby gotowa aplikacja mogła być użytkowana bez środowiska *Unity* należało także wprowadzić prosty interfejs użytkownika, który pozwoli na konfigurację modelu na każdym komputerze i przez dowolnego użytkownika. Każdy wyświetlany ekran jest w środowisku zbudowany na osobnej scenie i posiada skrypt odpowiedzialny za kontrolowanie GUI. Pliki te nie są zbyt skomplikowane i zwykle odnoszą się do funkcji przycisków dlatego zostaną omówione jedynie funkcjonalności ekranów bez zagłębiania się w obsługujący je kod.

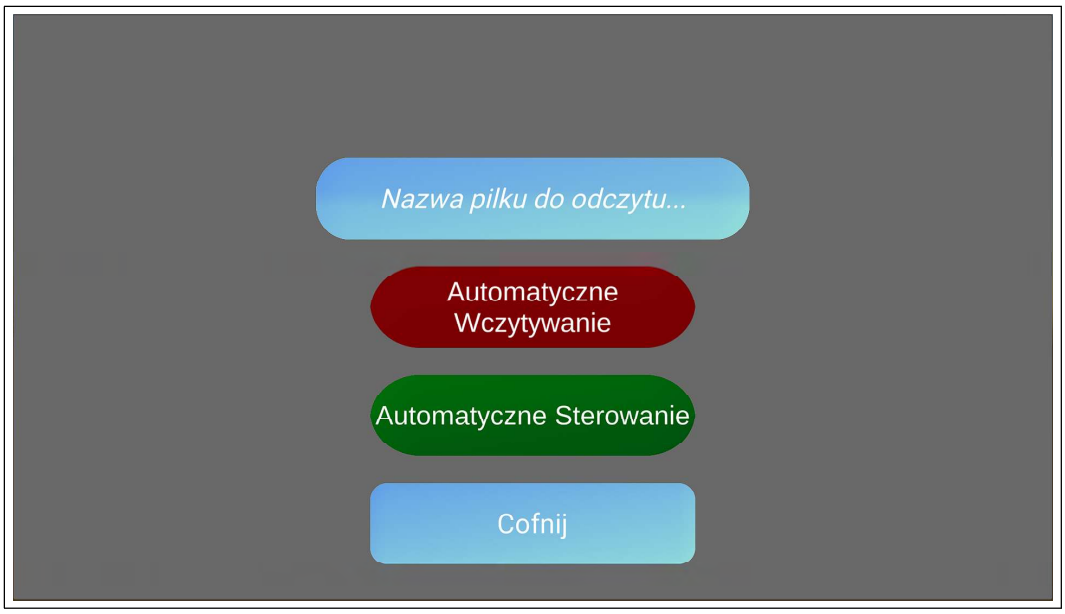
GUIMain.cs

Rysunek 3.47: Menu główne

Po uruchomieniu programu użytkownika ukazuje się menu główne (rys. 3.47). Znajdują się tutaj 3 przyciski:

- „Start" – uruchamia symulację
- „Konfiguracja" – przechodzi do menu konfiguracji
- „Wyjście" – zamyka program

GUIConfig.cs



Rysunek 3.48: Menu konfiguracji

W pierwszym polu menu konfiguracyjnego (rys. 3.48) można wpisać nazwę pliku tekstowego (bez rozszerzenia „.txt"), z którego program ma odczytywać dane. Plik jednak musi się znajdować w podfolderze o nazwie „Symulacja Skrzyzowania_Data" znajdującym się w folderze z aplikacją. 2 przyciski znajdujące się się pod polem tekstowym zmieniają zmienne odpowiedzialne za uruchamianie automatycznego wczytywania pliku oraz automatycznego sterowania skrzyżowaniem. Kiedy zmienne te są załączone przyciski przyjmują kolor zielony, a kiedy są wyłączone, przyjmują kolor czerwony. Ostatni przycisk „Cofnij" prowadzi użytkownika z powrotem do menu głównego.

GUISim.cs

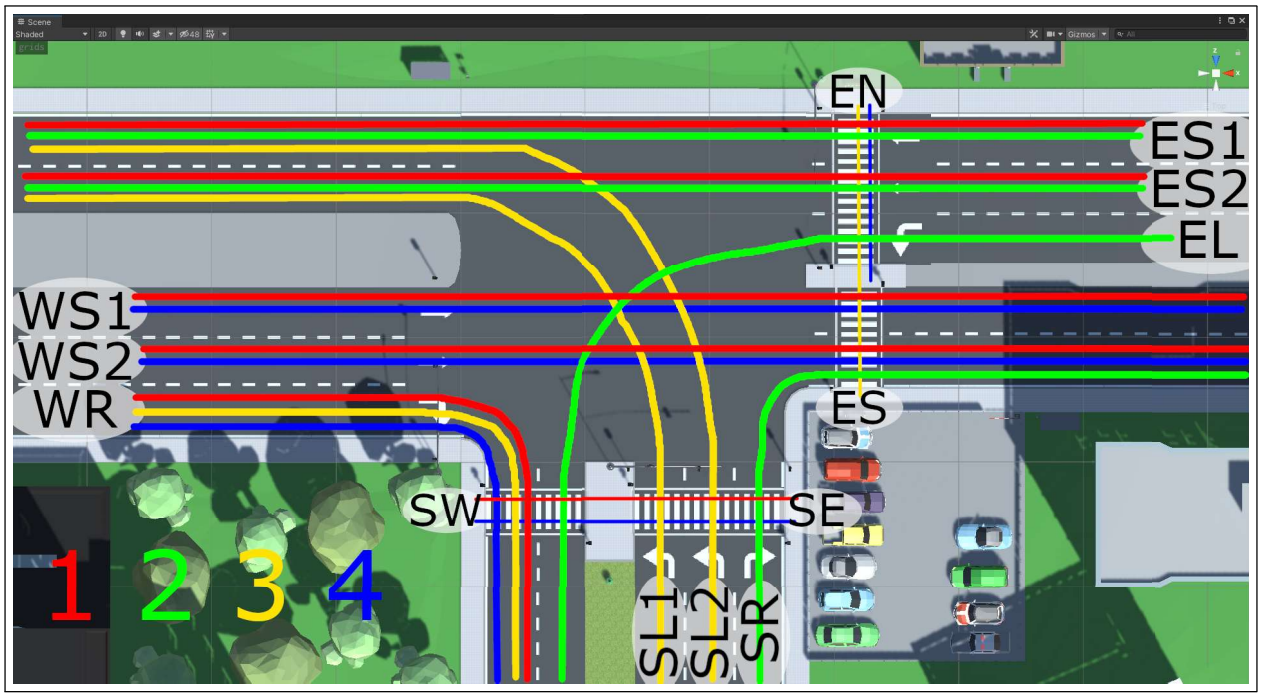
Rysunek 3.49: GUI podczas symulacji

Interfejs podczas symulacji (rys. 3.49) zależy od wcześniej ustalonej konfiguracji programu. Jeżeli algorytm sterowania jest wyłączony na ekranie pojawiają się niebieskie okręgi. Wskazują one na sygnalizatory świetlne

lub przejścia dla pieszych. Po ich wciśnięciu stan tego na co wskazują zostanie zmieniony (zmianę ze światła czerwonego na zielone nadal opóźnia czas międzyzielony). Jeżeli wyłączone jest także automatyczne czytanie danych, w prawym górnym rogu ekranu pojawi się niebieski przycisk „Wczytaj Dane”, którego każdorazowe naciśnięcie spowoduje włączenie procedury czytania danych przez program.

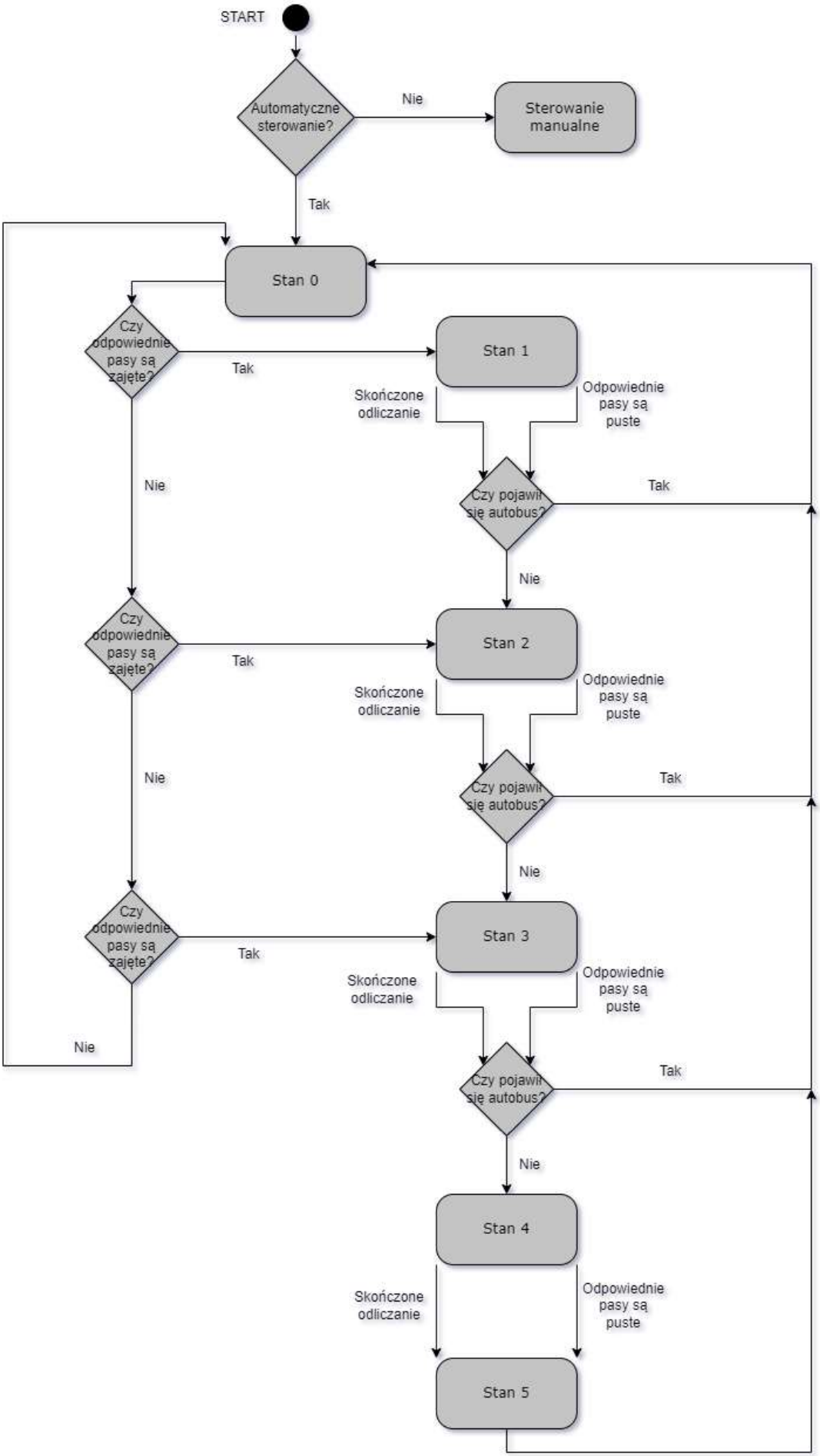
3.8 Algorytm sterowania skrzyżowaniem

Algorytm sterowania skrzyżowaniem został oparty na idei maszyny stanów. Maszyna stanów to automat definiujący zachowanie systemów dynamicznych jako maszyny o skończonej liczbie stanów i skończonej liczbie przejść pomiędzy tymi stanami [54]. Maszyna wykorzystana w tej pracy posiada 6 stanów, ale tylko stany od 1 do 4 zawierają w sobie instrukcje załączające odpowiednie sygnalizacje świetlne. Przepuszczane kierunki ruchu podczas każdego z tych stanów zostały ukazane na rysunku 3.50 (zgodnie z oznaczonymi w lewym dolnym rogu kolorami). Na tej samej grafice można zauważyć krótkie nazwy każdego z pasów, które były wykorzystywane podczas tworzenia algorytmu i zostaną wykorzystane do jego opisu.



Rysunek 3.50: Wizualizacja przepuszczanych obiektów w odniesieniu do stanów

Pierwszym stanem zdefiniowanym w maszynie jest stan „0”. Sprawdza on zajętość przejść dla pieszych oraz pasów ruchu obsługiwanych przez dany stan. Sprawdzanie zawsze zaczyna się od stanu „1”, a jeżeli brakuje w nim obiektów do przepuszczania, kontynuowane jest do stanów następnych (stan „4” nie jest sprawdzany, ponieważ dubluje on ścieżki ruchu ze stanu „1”). Po wykryciu uczestników ruchu automat przechodzi do odpowiedniego stanu i załącza sygnalizacje. W dalszym etapie sprawdzane są pasy unikatowe dla załączonego stanu (np. SL1 oraz SL2 dla stanu „3”). Jeżeli będą one puste (cały ruch zostanie przepuszczony) albo od załączenia sygnalizacji minie 40 sekund, przepływ ruchu zostanie zamknięty. W tym miejscu maszyna sprawdza czy na pasie WS2 albo ES1 pojawił się jakiś autobus. Jeżeli tak, to stan ustawiany jest na „0”, a autobusy zostaną przepuszczone w następnej kolejności, ponieważ ich pasy są obsługiwane przez stan „1”. Jeżeli autobusy nie zostaną wykryte załączony zostanie następny w kolejności stan. Ostatni z nich, czyli stan „5”, nie załącza żadnych sygnalizacji, a jedynie sprawdza warunki końca stanu „4” i po ich spełnieniu przechodzi na początek procesu. Cały schemat działania maszyny stanów został przedstawiony na rysunku 3.51.



Rysunek 3.51: Diagram maszyny stanów