WYDZIAŁ MATEMATYKI
i INFORMATYKI
Uniwersytet Łódzki

Kierunek: Informatyka

Bartłomiej Lipiński

Numer Albumu: 402694

Symulacja zachowań pojazdów w mieście stworzona w oparciu o silnik Unity

Simulation of vehicle behavior in city created in Unity engine

Praca napisana
pod kierunkiem:
dr. Sebastiana Lindnera
w Katedrze Funkcji Rzeczywistych
Wydziału Matematyki i Informatyki
Uniwersytetu Łódzkiego

Łódź, 2024

Spis treści

Streszczenie

Celem tej pracy jest opracowanie projektu symulacji ruchu pojazdów w mieście przy użyciu silnika Unity. Główne założenia obejmują problem poruszania się pojazdów oraz zbieranie odpowiednich danych.

Praca składa się z czterech głównych rozdziałów. W dwóch pierwszych omówiono główne pojęcia związane z projektem oraz jego zamysł. W trzecim rozdziale przedstawiono ogólne działanie symulacji, natomiast w czwartym szczegółowo wyjaśniono kod programu. Ostatni rozdział dotyczy przyszłych usprawnień projektu, zawiera również wyjaśnienie, w jaki sposób dodatkowe funkcje mogą być przydatne.

Słowa kluczowe: pojazdy, symulacja, silnik Unity, spline

Summary

The aim of this thesis is to develop a vehicle traffic simulation project in a city using the Unity engine. The main assumptions include the problem of vehicle movement and the collection of relevant data.

The thesis is divided into four main chapters. The first two explain the key concepts of the project and its premise. The third chapter describes the overall functioning of the simulation, while the fourth provides a detailed explanation of the code. The final chapter focuses on potential future improvements and explains how additional features might be beneficial.

Keywords: vehicle, simulation, Unity engine, spline

Rozdział 1 Wstęp

Celem symulacji ruchu pojazdów w miejskim środowisku jest stworzenie w miarę realistycznego modelu, który odzwierciedla rzeczywiste warunki drogowe i zachowania kierowców. Wykorzystanie silnika Unity pozwala na implementację odpowiednich algorytmów i narzędzi, które wspierają realizację tego celu.

Symulację można zastosować jako:

1. **Narzędzie planowania urbanistycznego**:
    - Używane do testowania różnych scenariuszy planowania infrastruktury drogowej i oceny ich wpływu na ruch miejski za pomocą zebranych danych.
2. **Grę/aplikację rozrywkową**:
    - Użycie w grach komputerowych i symulatorach, gdzie realistyczny ruch uliczny jest kluczowym elementem.

Rozdział 2 O projekcie

Główne założenia

Główne założenia symulacji ruchu pojazdów w mieście w silniku Unity obejmują szeroki zakres technicznych aspektów, w tym:

1. **Algorytmy ruchu**:
    - Implementacja algorytmów ruchu pojazdów, które określają ich zachowanie na drodze, takie jak utrzymywanie odpowiedniego odstępu od innych pojazdów oraz prawidłowe pokonywanie skrzyżowań i zakrętów.
    - Zastosowanie różnych technik sterowania, takich jak algorytmy przyspieszania, hamowania i skręcania, aby odzwierciedlić zachowanie prawdziwych kierowców.
2. **Zachowania kierowców**:
    - Symulacja reakcji kierowców na różne sytuacje, takie jak zmiana prędkości ruchu, żeby uniknąć zderzenia.
3. **Symulacja świateł drogowych**:
    - Implementacja logiki świateł drogowych, aby regulować ruch pojazdów na skrzyżowaniach i w innych miejscach, gdzie wymagane jest kontrolowanie przepływu ruchu.

Te założenia są kluczowe dla stworzenia wiarygodnej i realistycznej symulacji ruchu drogowego, która może być wykorzystana do testowania różnych scenariuszy drogowych, badania wpływu infrastruktury drogowej na ruch uliczny, oraz analizy efektywności strategii zarządzania ruchem. Ich skuteczna implementacja pozwala na dokładne odwzorowanie rzeczywistych warunków drogowych i poprawne modelowanie zachowań kierowców.

Inspiracje

Symulacja ruchu pojazdów w miejskim środowisku stanowi jedno z kluczowych wyzwań w tworzeniu realistycznych gier symulacyjnych oraz narzędzi urbanistycznych. W takim projekcie istotne jest czerpanie inspiracji z istniejących, dobrze ocenianych symulacji miejskich, takich jak "*Cities Skylines*" 1 i 2. W tym rozdziale będzie omówione, jak te gry inspirowały projekt oraz które elementy zostały zaadaptowane w symulacji.

"*Cities Skylines*" jest grą symulacyjną, która stała się wzorem w dziedzinie zarządzania miastem. Gra oferuje zaawansowany system symulacji ruchu drogowego, który charakteryzuje się realistycznym odwzorowaniem ruchu pojazdów oraz dynamiką przepływu ruchu miejskiego. Kluczowe elementy, które zainspirowały mnie w tym projekcie, obejmują:

1. **Algorytmy ścieżek (Pathfinding)**:
    - "*Cities Skylines*" wykorzystuje zaawansowane algorytmy, takie jak A* (A-star), aby efektywnie wyznaczać trasy dla pojazdów. Jest to algorytm znajdowania najkrótszej ścieżki w grafie ważonym między dwoma dowolnymi wierzchołkami. W projekcie został zaimplementowany prostszy algorytm, który nadal pozwala na efektywne przeprowadzanie symulacji.
2. **Symulacja świateł drogowych**:
    - W "*Cities Skylines*" zarządzanie światłami drogowymi jest kluczowym elementem, który wpływa na płynność ruchu. Zaadaptowane mechanizmy zarządzania sygnalizacją świetlną, poprawiają realizm ruchu w symulacji.

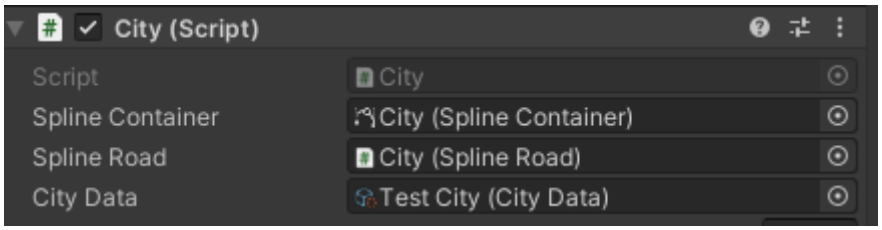
Czerpanie inspiracji z "*Cities Skylines 1*" oraz "*Cities Skylines 2*" pozwoliło na stworzenie realistycznej i efektywnej symulacji ruchu pojazdów w miejskim środowisku w silniku Unity. Implementacja działających ścieżek, i zarządzania ruchem znacząco zwiększyła realizm
i funkcjonalność symulacji. Wykorzystanie tych inspiracji pozwala na tworzenie bardziej realistycznych i angażujących symulacji miejskich, które mogą znaleźć zastosowanie zarówno w grach, jak i narzędziach do planowania urbanistycznego.

Rozdział 3 Mechanika symulacji

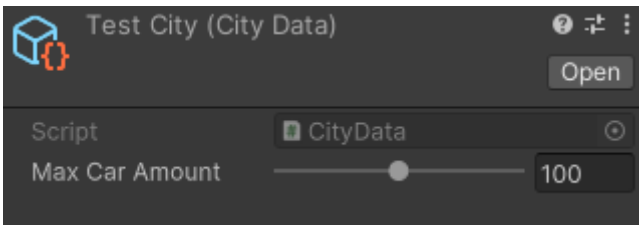
Aby ułatwić edycję i zarządzanie informacjami, projekt został zaprojektowany w sposób modułowy od samego początku. Takie podejście umożliwia łatwe dodawanie i modyfikowanie poszczególnych elementów systemu bez konieczności wprowadzania zmian w całym projekcie.

Modułowość w projekcie przynosi kilka kluczowych korzyści:

1. **Ustawienia miasta (**Rysunek 3.1 i Rysunek 3.2**)**
    - Dzięki modułowej budowie, nowe ustawienia miasta mogą być tworzone i dostosowywane bez ingerencji w resztę projektu.

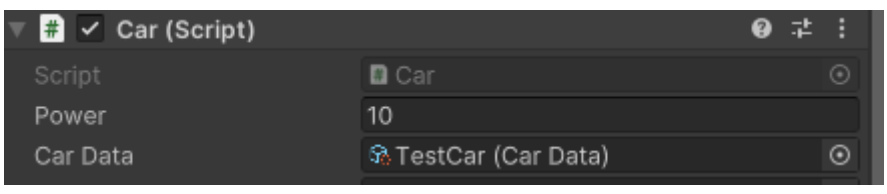


Rysunek 3.1 Zrzut ekranu przedstawiający jak moduł ustawień łączy się z samym miastem.

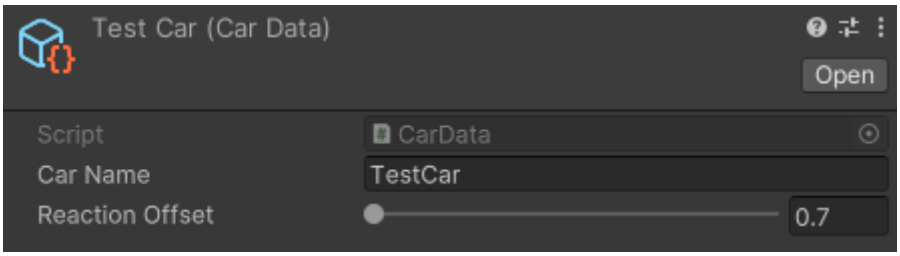


Rysunek 3.2 Zrzut ekranu przedstawiający moduł ustawień miasta.

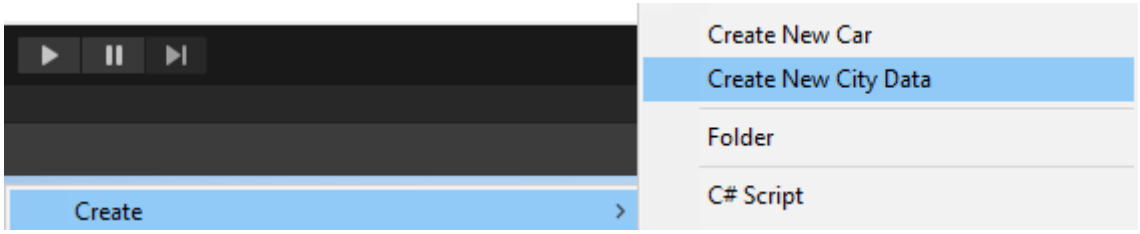
2. **Ustawienia pojazdów (**Rysunek 3.3 **i** Rysunek 3.4**)**:
    - Podobnie jak w przypadku ustawień miasta, konfiguracje pojazdów mogą być łatwo zmieniane lub dodawane.



Rysunek 3.3 Zrzut ekranu przedstawiający jak moduł ustawień łączy się z pojazdem.



Rysunek 3.4 Zrzut ekranu przedstawiający moduł ustawień pojazdu.



Rysunek 3.5 Zrzut ekranu przedstawiający łatwość tworzenia nowych ustawień.

Opis głównych elementów

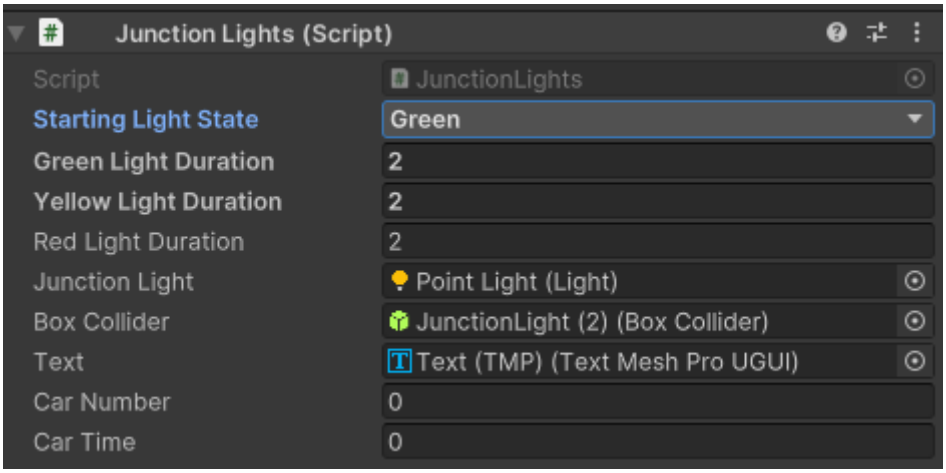
Światła

Symulacja świateł drogowych (Rysunek 3.6) w projekcie ma kilka kluczowych cech:

1. **Aktualny stan świateł**:
    - Światła mogą znajdować się w jednym z trzech stanów: zielonym, żółtym lub czerwonym.
2. **Dostosowywanie czasu trwania świateł**:
    - Użytkownik ma możliwość dostosowania czasu trwania każdego pojedynczego światła. Może zmieniać długość trwania stanów zielonego, żółtego i czerwonego, aby symulować różne warunki drogowe i sytuacje ruchu.
    - Aktualna implementacja nie pozwala na stworzenie efektu zielonej fali, natomiast można ustawić światła żeby dawały iluzję tego efektu.
3. **Zatrzymywanie ruchu w przypadku czerwonego światła**:
    - Gdy światło jest w stanie czerwonym, ruch pojazdów jest zatrzymywany.
4. **Średni czas przejazdu przez dane światło**:
    - Podczas czerwonego światła pojazdy rozpoczynają zliczanie czasu stania w korku.

- Kiedy pojazd przejeżdża przez światło, naliczony czas dodaje się do całkowitego czasu postoju wszystkich pojazdów na danej sygnalizacji. Dodatkowo zwiększa się o 1 licznik pojazdów, które przejechały przez to miejsce. Umożliwia to obliczenie średniego czasu przejazdu pojazdu.

Te funkcje umożliwiają symulację zachowania się ruchu na skrzyżowaniach, pozwalając na eksperymentowanie z różnymi ustawieniami sygnalizacji świetlnej i ocenę ich wpływu na płynność ruchu drogowego.



Rysunek 3.6 Zrzut ekranu przedstawiający komponent świateł.

Miasto

W projekcie symulacji ruchu drogowego istnieje jedno miasto, które ma kluczowe zadania:

1. **Zarządzanie infrastrukturą drogową**:
   - jest odpowiedzialne za przechowywanie listy punktów docelowych.
2. **Monitorowanie stanu świateł drogowych**:
   - Komponent zbiera dane ze wszystkich świateł drogowych w symulacji. Te informacje są wykorzystywane do wyświetlania na interfejsie użytkownika średniego czasu przejazdu przez wszystkie światła w mieście. Dzięki temu użytkownik może śledzić, jak zmienia się przepustowość na skrzyżowaniach w różnych warunkach ruchu.

- Dzięki temu użytkownik może śledzić, jak zmienia się przepustowość na skrzyżowaniach w różnych warunkach ruchu.

3. **Analiza zatoru drogowego**:
   - Średni czas przejazdu przez światła drogowe dostarcza istotnych danych na temat przepustowości dróg i skrzyżowań. Na podstawie tych informacji możliwe jest ocenienie, jak bardzo miasto jest narażone na korki oraz identyfikacja obszarów o największym prawdopodobieństwie wystąpienia zatorów drogowych.

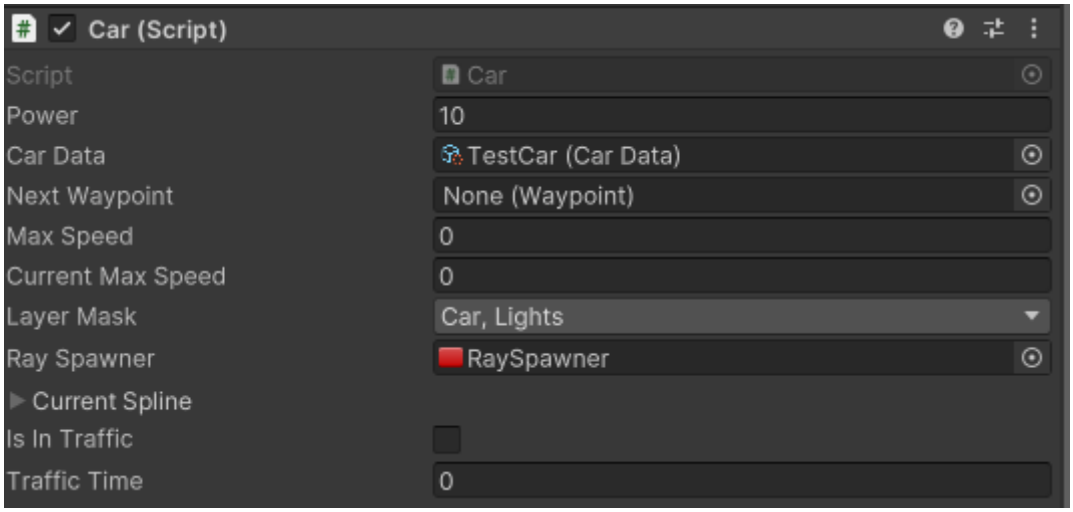
W projekcie pominięto skrzyżowania bez sygnalizacji świetlnej, ponieważ opracowanie efektywnej i realistycznej mechaniki dla skrzyżowań równorzędnych oraz podporządkowanych zwiększyłoby stopień skomplikowania. To z kolei wymagałoby znacznie więcej czasu. Skupiłem się więc na innych aspektach projektu, aby móc go ukończyć w rozsądnym czasie.

Pojazdy

Pojazdy w symulacji poruszają się za pomocą komponentu silnika fizycznego wbudowanego w Unity – Rigidbody. Jest to komponent, który nadaje obiektowi fizykę, umożliwiając mu reakcję na siły fizyczne, a przez to bardziej realistyczne odwzorowanie ich zachowania na drodze. Istnieją trzy kluczowe elementy, które charakteryzują sposób, w jaki poruszają się te pojazdy:

1. **Dostosowywanie prędkości**:
   - Silnik fizyczny umożliwia dynamiczne modyfikowanie prędkości pojazdów w zależności od warunków otoczenia. Maksymalna prędkość każdego pojazdu jest dostosowywana w taki sposób, aby uniknąć kolizji z innymi pojazdami oraz przeszkodami na drodze. Jak tylko nie jest to potrzebne, prędkość wraca ona do normalnej wartości. Jednakże robią to z pewnym ustalonym opóźnieniem, więc jest szansa na wystąpienie kolizji. Niestety nie zdążyłem tego zaimplementować, ze względu na skupienie się na innych aspektach projektu.

2. **Opóźnienie reakcji**:
    - Prędkość maksymalna jest zmieniana od razu jak tylko jest to potrzebne. Natomiast pojazd dostosowuje swoją własną prędkość z opóźnieniem, co odzwierciedla zachowanie prawdziwych kierowców na drodze.
3. **Losowanie kolejnego punktu docelowego**:
    - Aktualnie po dotarciu do końca obecnej drogi pojazd losuje sobie następną trasę z listy dostępnych.
    - Nie jest to zbyt realistyczne, ponieważ w prawdziwym świecie nie ma miasta, gdzie każdy losuje trasę. To zrobiłoby, że każda droga jest tak samo obciążona, co jest nierealistyczne. W tym projekcie nie został zaimplementowany lepszy algorytm, ponieważ skupiłem się na innych aspektach symulacji.
4. **Zakończenie trasy**:
    - Gdy samochód dotrze na koniec trasy, czyli do punktu, który nie prowadzi nigdzie, pojazd zostaje usunięty z symulacji. Ten mechanizm pozwala na cykliczne wykorzystywanie pojazdów i zapobiega gromadzeniu się niepotrzebnych obiektów w symulacji.



Rysunek 3.7 Zrzut ekranu przedstawiający komponent auta.

Ulice

System tworzenia ulic był jednym z najtrudniejszych problemów do rozwiązania. Po wielu różnych próbach zdecydowałem na znaczne ułatwienie w modyfikowaniu infrastruktury poprzez dodatkowe menu (Rysunek 3.8).

Rysunek 3.8 Zrzut ekranu przedstawiający dodatkowe ułatwienia edycji infrastruktury drogowej w mieście..

Po naciśnięciu przycisku „Generate Roads” w komponencie miasta, program przystępuje do dynamicznego tworzenia ulic w mieście. Proces ten przebiega w następujący sposób:

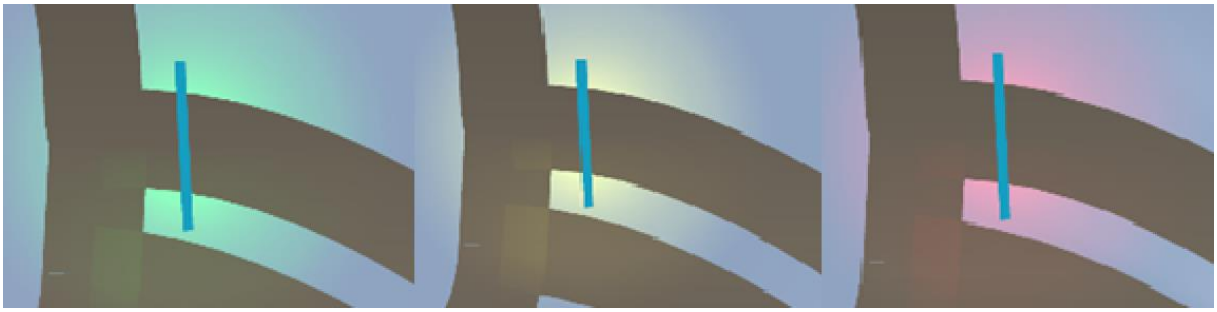
1. **Przegląd punktów docelowych**:
    - Program przechodzi po kolei przez wszystkie punkty docelowe miasta.
    - Jeżeli punkt prowadzi do jakiegokolwiek innego, skrypt tworzy podstawową drogę, korzystając z komponentu silnika Unity – Spline. Jest to komponent do tworzenia i manipulowania krzywymi, które służą do animowania ruchu obiektów w grze.
2. **Dzielenie drogi na segmenty**:
    - Droga od punktu do innego punktu docelowego jest dzielona na odpowiednio wiele segmentów, co pozwala na tworzenie realistycznych i dokładnych graficznych reprezentacji dróg w mieście.
3. **Tworzenie obiektu drogi**:
    - Na podstawie tych segmentów tworzony jest kształt drogi. Jest to realizowane poprzez tworzenie krzywej Beziera, która jest elastyczna i pozwala na wygładzanie zakrętów oraz dostosowanie kształtu drogi do topografii terenu.
4. **Dostosowywanie kształtu drogi**:
    - Jako że droga jest tworzona jako krzywa Beziera, istnieje możliwość manipulowania jej kształtem, włączając w to wyginanie i dostosowywanie trajektorii drogi. Skrypt automatycznie poprawia drogi, aby zapewnić graficzne przedstawienie wykrzywionych tras.

Ten proces dynamicznego tworzenia ulic pozwala na elastyczne dostosowywanie infrastruktury drogowej w mieście, uwzględniając zmieniające się warunki i potrzeby. Dzięki temu symulacja może reprezentować różnorodne układy drogowe.

Aspekt graficzny

Dzięki modułowej konstrukcji projektu, zmiana grafiki jest wyjątkowo prosta i nie wpływa na funkcjonalność mechanizmów symulacji. Taka architektura pozwala na łatwe aktualizacje wizualne, bez ryzyka zakłócenia działania systemu.

Światła



Rysunek 3.9 Zrzuty ekranu pokazujące graficzne działanie świateł sygnalizacyjnych podczas działania symulacji.

Miasto

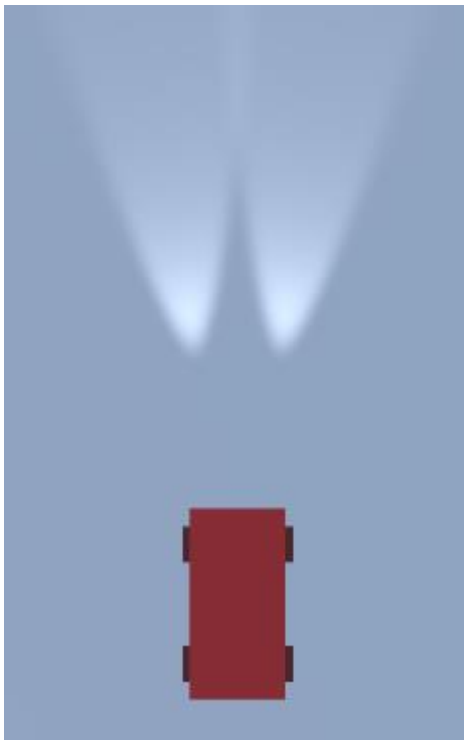
Miasto można rozbudowywać za pomocą różnych komponentów. Punkty docelowe są połączone drogami, po których poruszają się pojazdy. Sygnalizatory dodają dynamiki, zmieniając kolory świateł w określonych odstępach czasu.

Rysunek 3.10 Zrzut ekranu pokazujący północny fragment Tykocina przedstawiony w symulacji.

Rysunek 3.11 Zrzut ekranu pokazujący północny fragment Tykocina podczas trwania symulacji.

Pojazdy

Żeby poprawić widoczność w ustawionym odgórnym widoku, do pojazdów zostały dodane światła, oraz łatwa możliwość zmiany koloru karoserii (Rysunek 3.12).



Rysunek 3.12 Zrzut ekranu pokazujący graficzną interpretację pojazdu.

Ulice

System graficzny dróg stanowi wyjątek od tej reguły. Możliwe są jedynie zmiany parametrów materiału, takich jak czysty kolor i odbijanie światła. Nie zdążyłem jednak dodać działającej tekstury do dróg, dlatego, ani pasy, ani strzałki nie są wyświetlane.

Rysunek 3.13 Zrzut ekranu pokazujący graficzną interpretację drogi przed zastosowaniem wykręcania.

Rysunek 3.14 Zrzut ekranu pokazujący graficzną interpretację drogi po zastosowaniu wykręcania.

Rozdział 4 Implementacja kodu

Środowisko Unity

Mechaniki zostały zrealizowane w Unity 2022.3.3f1, a ruch pojazdów korzysta z wbudowanego silnika fizycznego. Modele zostały stworzone w Unity, a skrypty napisane w C# za pomocą Visual Studio 2022.

Własne skrypty

Punkty docelowe

Każdy punkt trasy jest opatrzony informacjami na temat innych punktów, z którymi jest połączony. Dodatkowo, użytkownik ma możliwość decydowania o tym, czy w danym punkcie mogą pojawiać się pojazdy, oraz określenia interwału czasowego, z jakim pojazdy będą pojawiać się w danym miejscu.

Światła

```
private void Awake()
{
	boxCollider = GetComponent<BoxCollider>();
	switch (startingLightState)
	{
		case LightState.Green:
			StartCoroutine(nameof(GreenLight));
			break;
		case LightState.Yellow:
			StartCoroutine(nameof(YellowLight),
			LightState.Green);
			break;
		case LightState.Red:
			StartCoroutine(nameof(RedLight));
			break;
		default:
			break;
```

```
        }
    }
    IEnumerator GreenLight()
    {
        junctionLight.color = Color.green;
        boxCollider.isTrigger = true;
        yield return new
        WaitForSecondsRealtime(greenLightDuration);
        StartCoroutine(nameof(YellowLight), LightState.Green);
    }

    IEnumerator YellowLight(LightState state)
    {
        junctionLight.color = Color.yellow;
        yield return new
        WaitForSecondsRealtime(yellowLightDuration);
        if (state == LightState.Green)
        {
            StartCoroutine(nameof(RedLight));
        }

        if (state == LightState.Red)
        {
            boxCollider.isTrigger = true;
            StartCoroutine(nameof(GreenLight));
        }
    }

    IEnumerator RedLight()
    {
        junctionLight.color = Color.red;
        boxCollider.isTrigger = false;
        yield return new WaitForSecondsRealtime(redLightDuration);
```

```
        StartCoroutine(nameof(YellowLight), LightState.Red);
}
```

Miasto

```
private void Update()
{
        SpawnCar();
        carAmount = carHolder.transform.childCount;
        int amount = junctionLightHolder.transform.childCount;
        averageJunctionWaitingTime = 0;
        foreach (Transform t in junctionLightHolder.transform)
                {
                        averageJunctionWaitingTime +=
                t.GetComponent<JunctionLights>().GetAverageTime();
                }
                averageJunctionWaitingTime =
        Math.Round(averageJunctionWaitingTime / amount, 2);
}
public void SpawnCar()
{
        if (carHolder.transform.childCount < cityData.maxCarAmount)
        {
                System.Random random = new();
                List<Waypoint> possibleStartPoints =
        startPoints.Where(w => w.canSpawn == true).ToList();
                if (possibleStartPoints.Count > 0)
                {
                        Waypoint startPoint =
                        possibleStartPoints[random.Next(possibleStartPoin
                        ts.Count)];
                         StartCoroutine(startPoint.Spawn());
                         Car car = Instantiate(carPrefab,
startPoint.transform.position, Quaternion.identity,
carHolder.transform).GetComponent<Car>();
                         Road road = startPoint.GetRandomRoad();
```

```
                car.currentSpline =
splineContainer.Splines[road.GetIndex()];
                car.nextWaypoint = road.GetWaypoint();
            }
        }
}
```

Metoda Update jest wbudowana w silnik Unity. Jest to specjalna metoda, która wywołuje się co klatkę. W przypadku tego projektu, kolejna klatka oznacza próbę stworzenia nowego pojazdu, odczyt aktualnej liczby pojazdów oraz odczyt średniego czasu oczekiwania na przejazd przez wszystkie światła.

Natomiast metoda SpawnCar sprawdza, czy aktualna liczba pojazdów jest mniejsza niż maksymalna liczba określona w ustawieniach miasta. Jeśli tak, tworzy nowy pojazd w losowo wybranym punkcie, który został oznaczony jako miejsce do tworzenia pojazdu. Następnie samochód otrzymuje losowo wybraną trasę do pokonania.

Pojazdy

```
private void Update()
{
    if (Vector3.Distance(transform.position,
nextWaypoint.transform.position) <= 0.5f)
    {
        if (nextWaypoint.GetRoads().Count != 0)
        {
            Road nextRoad = nextWaypoint.GetRandomRoad();
            HitWaypoint(City.Instance.splineContainer.Splines
            [nextRoad.GetIndex()]);
            nextWaypoint = nextRoad.GetWaypoint();
        }
        else
        {
            Destroy(gameObject);
        }
    }
```

```
     Ray ray = new Ray(raySpawner.transform.position,
transform.forward);
     if (Physics.Raycast(ray, out RaycastHit hit, 10,layerMask))
     {
          if (hit.collider.TryGetComponent(out Car otherCar) &&
     Vector3.Distance(transform.position,
     otherCar.transform.position) < 4)
          {
               if (otherCar.currentMaxSpeed < currentMaxSpeed)
               {
               currentMaxSpeed = otherCar.currentMaxSpeed;
               }
          }

          if (hit.collider.TryGetComponent(out JunctionLights
     junctionLights) && Vector3.Distance(transform.position,
     junctionLights.transform.position) < 3)
          {
               if (junctionLights.boxCollider.isTrigger ==
     false)
               {
                    currentMaxSpeed = 0;
               }
               else
               {
                    currentMaxSpeed = maxSpeed;
               }
          }
     }
     else
     {
          currentMaxSpeed = maxSpeed;
     }
```

```
        if (currentMaxSpeed == 0)
        {
            isInTraffic = true;
        }
        else
        {
            isInTraffic = false;
        }

        if (isInTraffic)
        {
            trafficTime += Time.deltaTime;
        }
}
```

Pojazd porusza się z jednego punktu kontrolnego do drugiego. Gdy osiągnie dany punkt, sprawdza, czy istnieją kolejne drogi do przebycia. Jeśli tak, losowo wybiera jedną i przemieszcza się do następnego punktu na tej drodze. Jeśli nie ma już więcej dróg, obiekt zostaje zniszczony.

Co klatkę wysyła promień do przodu, aby wykryć kolizje z innymi obiektami, takimi jak samochody lub sygnalizacje świetlne. Jeśli wykryje inny pojazd w określonej odległości, dostosowuje swoją chwilową prędkość maksymalną na podstawie prędkości innego pojazdu. Jeśli wykryje światła na skrzyżowaniach, i są one czerwone, chwilowa prędkość maksymalna jest zmieniana do 0.

Jeżeli chwilowa prędkość maksymalna jest równa 0, pojazd zaczyna zliczać czas spędzony w korku. Później jak tylko dotknie świateł, oddaje im zmierzony czas i zwiększa licznik pojazdów, które przejechały przez daną sygnalizację.

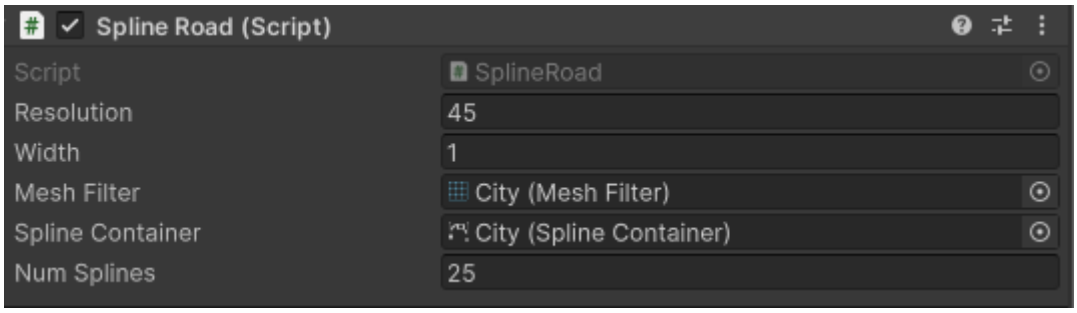
```
private void FixedUpdate()
{
    var spline = new NativeSpline(currentSpline);
```

```
        SplineUtility.GetNearestPoint(spline, transform.position,
out float3 nearest, out float t,
SplineUtility.PickResolutionMax,
SplineUtility.PickResolutionMax);

        transform.position = new Vector3(nearest.x, nearest.y,
nearest.z);

        Vector3 forward =
Vector3.Normalize(spline.EvaluateTangent(t));

        Vector3 up = spline.EvaluateUpVector(t);

        var axisRemapRotation =
Quaternion.Inverse(Quaternion.LookRotation(Vector3.forward,
Vector3.up));

        transform.rotation = Quaternion.LookRotation(forward, up) *
axisRemapRotation;

        rb.velocity = rb.velocity.magnitude * transform.forward;

        if (rb.velocity.magnitude < currentMaxSpeed)

        {

            this.Invoke(nameof(SpeedUp), carData.reactionOffset);

        }

        if (rb.velocity.magnitude > currentMaxSpeed)

        {

            rb.velocity = rb.velocity.normalized *
        currentMaxSpeed;

        }

}
```

Metoda FixedUpdate jest wbudowaną funkcją w silniku Unity. W przeciwieństwie do metody Update, która wywołuje się w każdej klatce, FixedUpdate uruchamia się w stałych odstępach czasowych. Domyślnie jest to co 0.02 sekundy, co odpowiada 50 razy na sekundę. Zaleca się używanie metody FixedUpdate do wszystkich operacji fizycznych w Unity, ponieważ silnik fizyczny aktualizuje stan fizyczny obiektów w określonych interwałach czasowych (takich jak pozycja, rotacja, prędkość i siły działające na obiekt).

W tym fragmencie pokazane jest, jak pojazd porusza się po drodze stworzonej za pomocą komponentu Spline. Wybiera najbliższy punkt na trasie, aktualizuje swoją rotację, symulując skręcanie, a następnie zwiększa swoją prędkość. Jeśli prędkość jest mniejsza niż aktualna maksymalna, wywoływana jest metoda SpeedUp z opóźnieniem ustawionym w komponencie ustawień.

Ulice



Rysunek 4.1 Zrzut ekranu przedstawiający komponent pozwalający na dynamiczne generowanie dróg.

Rozdział 5 Dalszy rozwój

Optymalizacja

Dobra optymalizacja jest kluczowa w każdym projekcie, ponieważ pozwala na zwiększenie skali bez spadku wydajności. Polega ona na dostosowaniu i usprawnieniu działania systemu, aby mógł obsługiwać większe obciążenia przy zachowaniu płynności i efektywności.

W kontekście obecnego rozwiązania, problemem jest ograniczona zdolność obsługi dużej liczby pojazdów. Przyczyną tego jest sposób, w jaki silnik fizyczny przetwarza dane. Dla każdego pojazdu silnik fizyczny wykonuje zbyt wiele obliczeń, co prowadzi do znacznego obciążenia urządzenia. Każde dodatkowe obliczenie zwiększa zapotrzebowanie na zasoby, co w rezultacie spowalnia działanie całego systemu. W skrajnym przypadku może to prowadzić do spadku wydajności, co jest szczególnie problematyczne w aplikacjach wymagających płynnego działania, takich jak symulacje.

Aby rozwiązać ten problem, należy zidentyfikować i zoptymalizować te aspekty silnika fizycznego, które generują największe obciążenie. Możliwe podejścia obejmują:

1. **Redukcję złożoności obliczeń**: Uproszczenie modelu fizycznego dla pojazdów, na przykład poprzez zmniejszenie liczby zmiennych lub parametrów, które są monitorowane i przetwarzane.
2. **Korzystanie z poziomów szczegółowości (LOD)**: Implementacja technik, które zmniejszają szczegółowość obliczeń dla pojazdów, które są dalej od kamery lub mniej istotne w danym momencie.
3. **Optymalizacja algorytmów**: Poprawa efektywności algorytmów fizycznych, aby mogły wykonywać te same zadania przy mniejszym zużyciu zasobów.
4. **Wykorzystanie równoległego przetwarzania**: Wykorzystanie technologii wielordzeniowych procesorów do rozdzielenia obciążeń obliczeniowych między różne rdzenie, co może znacząco poprawić wydajność.
5. **Inteligentne zarządzanie zasobami**: Dynamiczne alokowanie zasobów do najbardziej potrzebujących elementów systemu, co może pomóc w utrzymaniu płynnej pracy nawet przy dużym obciążeniu.

Wdrożenie tych i innych technik optymalizacyjnych pozwoli na znaczące zwiększenie liczby obsługiwanych pojazdów bez spadku wydajności, co przełoży się na lepsze doświadczenie użytkowników i bardziej efektywne wykorzystanie zasobów systemowych.

Poprawa algorytmu ruchu

Obecny algorytm ruchu pojazdów w naszej symulacji nie jest idealny. Auta nie biorą pod uwagę przeszkód innych niż te, które znajdują się bezpośrednio przed nimi. Powoduje to problemy z poruszaniem się na zakrętach, rondach i innych złożonych elementach infrastruktury drogowej.

- **Ograniczone wykrywanie przeszkód**: Obecny system używa pojedynczego promienia (raycast) wysyłanego do przodu od pojazdu, aby wykrywać przeszkody. Taki sposób wykrywania nie sprawdza obszarów po bokach lub za pojazdem, co prowadzi do problemów na bardziej skomplikowanych trasach.

Możliwe rozwiązania

1. **Zwiększenie liczby promieni (raycastów)**:
    - **Opis**: Zamiast jednego promienia, użycie kilku promieni rozchodzących się w różnych kierunkach (np. do przodu, na boki, pod kątem).
    - **Zalety**: Lepsze wykrywanie przeszkód wokół pojazdu, co pozwoli na bardziej płynne poruszanie się, szczególnie na zakrętach i rondach.
    - **Wady**: Zwiększenie liczby promieni znacząco zwiększy obciążenie obliczeniowe silnika fizycznego. Może to prowadzić do spadku płynności symulacji, szczególnie przy dużej liczbie pojazdów.
2. **Zastąpienie promieni overlapem**:
    - **Opis**: Zamiast promieni, użycie metody overlap, wbudowanej w silnik fizyczny Unity. Metoda ta tworzy niewidzialne pudełko wokół pojazdu, które sprawdza kolizje w określonym obszarze.
    - **Zalety**: Overlap może sprawdzać większy obszar nie ograniczając się tylko do jednego kierunku. Pozwala to na lepsze wykrywanie przeszkód, co może poprawić nawigację pojazdów w złożonych sytuacjach.
    - **Wady**: Chociaż overlap zmniejsza liczbę pojedynczych promieni, nadal może wymagać znacznych zasobów obliczeniowych, zwłaszcza jeśli wiele pojazdów

jest w ruchu jednocześnie. Dodatkowo, implementacja overlapu może wymagać dostosowania algorytmów ruchu, aby poprawnie interpretować dane z wykrytych kolizji.

Dodatkowo, należy zauważyć, że obecny algorytm ruchu pojazdów w naszej symulacji nie uwzględnia nachylenia terenu. Brak tego elementu może prowadzić do niewłaściwego zachowania się pojazdów, szczególnie na stromych drogach lub podjazdach, gdzie prędkość i manewrowanie mogą być ograniczone ze względu na ukształtowanie terenu.

Możliwe rozwiązania:

1. **Uwzględnienie stromości terenu**:
    - Zmodyfikowanie algorytmu ruchu pojazdów w taki sposób, aby uwzględniał stromość terenu przy określaniu prędkości i manewrowaniu pojazdów. W ten sposób można zapobiec sytuacjom, w których pojazdy próbują poruszać się po zbyt stromych drogach.
2. **Dopasowanie prędkości do nachylenia terenu**:
    - Możliwe jest dostosowanie prędkości pojazdów w zależności od nachylenia terenu. Na stromych drogach prędkość może być ograniczona, aby zapewnić bezpieczne poruszanie się pojazdów i uniknięcie utraty kontroli.

Dodatkowo uwzględnienie stromości terenu w algorytmie ruchu pojazdów może znacząco poprawić realizm symulacji i umożliwić bardziej dokładne odwzorowanie rzeczywistych warunków drogowych. Jednakże, konieczne jest przeprowadzenie odpowiednich testów i optymalizacji, aby zapewnić, że dodanie tego elementu nie wpłynie negatywnie na wydajność i płynność symulacji.

Edytor ustawień miasta i pojazdów

Obecna opcja zmieniania wartości ustawień miasta i pojazdów jest wbudowana bezpośrednio w edytor Unity, co oznacza, że aby wprowadzić jakiekolwiek zmiany, konieczny jest dostęp do całego projektu. Taki sposób pracy ma kilka istotnych wad:

1. **Problem z powtarzalnością testów**: Każda zmiana ustawień wymaga ręcznego dostępu do edytora Unity, co utrudnia przeprowadzanie spójnych i powtarzalnych

testów. Zmiany dokonywane ręcznie mogą być niedokładne lub różnić się między testami, co wpływa na wyniki i ich wiarygodność.

2. **Trudność obsługi**: Wymóg korzystania z edytora Unity do zmiany ustawień sprawia, że proces jest bardziej skomplikowany i mniej dostępny dla użytkowników, którzy nie są zaznajomieni z edytorem Unity. Ponadto, każda zmiana wymaga ponownego uruchomienia edytora, co jest czasochłonne i nieefektywne.
3. **Brak dynamicznego zarządzania**: Zmiany dokonywane w edytorze nie mogą być wprowadzane w czasie rzeczywistym podczas działania symulacji, co ogranicza możliwość dynamicznego testowania różnych scenariuszy i dostosowywania parametrów na bieżąco.

Aby rozwiązać te problemy, można wprowadzić dodatkowy interfejs użytkownika (UI) w projekcie, który umożliwiałby:

1. **Resetowanie całej sceny**: Interfejs powinien pozwalać na szybkie resetowanie sceny do stanu początkowego bez konieczności ponownego uruchamiania edytora Unity. To usprawni proces testowania i pozwoli na łatwe przeprowadzanie wielokrotnych testów w kontrolowanych warunkach.
2. **Zmiana ustawień między symulacjami**: Interfejs powinien umożliwiać użytkownikom łatwe wprowadzanie zmian w ustawieniach miasta i pojazdów bezpośrednio z poziomu aplikacji. Możliwość dynamicznego dostosowywania parametrów pozwoli na bardziej efektywne testowanie różnych scenariuszy i szybkie dostosowywanie symulacji do zmieniających się potrzeb.
3. **Intuicyjność i dostępność**: Nowy interfejs użytkownika powinien być intuicyjny i łatwy w obsłudze, nawet dla osób nieznających się na Unity. Przyjazny dla użytkownika design sprawi, że zmiany ustawień będą mogły być dokonywane szybko i bez błędów.

Wprowadzenie takiego interfejsu użytkownika znacząco poprawi efektywność procesu testowania, ułatwi obsługę i umożliwi dynamiczne zarządzanie ustawieniami symulacji, co jest kluczowe dla rozwoju i optymalizacji projektu.

Edytor dróg

Głównym problemem tworzenia siatki infrastruktury w projekcie jest jej czytelność. Za każdym razem, gdy dodawany jest punkt, trzeba wczytać wszystkie punkty docelowe od nowa. Dodatkowo resetuje to nadane przez użytkownika kształty dróg (krzywe Beziera), co znacznie wydłuża i utrudnia rozbudowę istniejącej części.

Można by to naprawić dodając kontrolę nad tym, które części dróg zostają resetowane. Jednakże nie byłem w stanie tego zrobić.

Rozdział 6 Zakończenie

Podsumowując, niniejsza praca przedstawiła proces tworzenia symulacji ruchu ulicznego z wykorzystaniem silnika Unity. Dzięki inspiracjom i obserwacjom życia codziennego udało mi się stworzyć działający projekt. Główne założenia zostały zrealizowane poprzez wdrożenie algorytmów sterowania ruchem oraz stworzenie realistycznego środowiska miejskiego. Jednocześnie napotkane trudności i wyzwania podczas realizacji projektu pozwoliły zidentyfikować obszary wymagające dalszych usprawnień. W przyszłości warto skupić się na bardziej zaawansowanych algorytmach poruszania się agentów ruchu. Dodatkowo, dobrym pomysłem byłoby ulepszenie elementów interaktywnych w trakcie działania symulacji.

Rozdział 7 Bibliografia

1. Game Dev Guide, film instruujący tworzenie dróg za pomocą pakietu spline w Unity, https://www.youtube.com/watch?v=ZiHH_BvjoGk (data wejścia 20.04.2024),
2. Unity, dokumentacja OverlapBox, https://docs.unity3d.com/ScriptReference/Physics.OverlapBox.html (data wejścia 10.05.2024),
3. Unity, dokumentacja pakietu spline, https://docs.unity3d.com/Packages/com.unity.splines@2.6/manual/index.html (data wejścia 17.03.2024),
4. Wikipedia, matematyczne wyjaśnienie spline, https://en.wikipedia.org/wiki/Spline_(mathematics) (data wejścia 21.03.2024)
5. Unity, dokumentacja komponentu umożliwiającego oddziaływanie sił fizycznych na obiekt rigidbody, https://docs.unity3d.com/ScriptReference/Rigidbody.html (data wejścia 21.03.2024),
6. Bob Nystrom, *Game programming patterns*, http://gameprogrammingpatterns.com/contents.html (data wejścia 24.03.2024),
7. Mike Geig, *Unity przewodnik projektanta gier*, HELION, Polska 2015,
8. Jacek Wesołowski, *Inżynieria Gier, Level design dla początkujących,* Wonderlang Engineering, Polska 2016

Rozdział 8 Spis ilustracji