

SOFTWARE-ARCHITEKTUREN

Verkehrssimulation Dokumentation

durchgeführt am
Studiengang Informationstechnik & System–Management
an der
Fachhochschule Salzburg GmbH

vorgelegt von

**Fabian Schörghofer, Andreas Reschenhofer, Lukas Altenhuber, Paul Riedl,
Mike Thomas**



Leiter des Studiengangs: FH-Prof. DI Dr. Gerhard Jöchtl

Betreuer: DI (FH) DI Roland Graf, MSc

Betreuer: DI Eduard Hirsch

Puch am, 13. Juli 2017

Inhaltsverzeichnis

1 Einführung	3
1.1 Aufgabenstellung	3
2 Software-Architektur	4
2.1 Randbedingungen	4
2.2 Kontextabgrenzung	4
2.3 Lösungsstrategie	4
2.4 Bausteinsicht	5
2.5 Laufzeitsicht	9
2.6 Verteilungssicht	10
2.7 Querschnittliche Konzepte	11
2.8 Entwurfsentscheidungen	12
2.9 Qualitätsszenarien	12
2.10 Risiken und technische Schulden	12
3 Design	13
3.1 Strassensystem	13
3.1.1 Ampeln	13
3.1.2 Straßen	13
3.1.3 Kreuzungen	14
3.1.4 Spawn	15
3.2 Fahrzeuglogik	16
3.3 Ampelsteuerung	16
3.4 Hindernisse	17
3.5 Aus- und Einfahren von Fahrzeugen anderer Gruppen	17
4 Implementierung	19
4.1 Erstellung der Welt	19
4.2 Fahrzeugsteuerung	20
4.2.1 Folgen der Strasse	20
4.2.2 Abbiegen bei Kreuzungen	21
4.3 Erkennen und Reagieren auf Ampeln	22
4.4 Fahrzeug Kollisionserkennung	24
4.5 Ampelsteuerung	25
4.6 Erstellen und Löschen von Hindernisse	26
4.7 Aus- und Einfahren von Fahrzeugen anderer Gruppen	28
5 Review einer fremden Architekturendokumentation	30

5.1 Review	30
----------------------	----

1 Einführung

Im Abschnitt 1 wird dem Leser ein Überblick der Aufgaben des Verkehrssimulationsprojekts gegeben.

1.1 Aufgabenstellung

Im Rahmen dieser Übung soll eine Verkehrssimulation realisiert werden. Dabei sollen sich diverse Verkehrsteilnehmer, zum Beispiel Autos und Busse, entsprechend der üblichen Straßenverkehrsregeln in einem gegebenen Straßennetz bewegen. Der Anwender der Simulation soll die Möglichkeit haben sowohl Simulationsparameter als auch Straßennetze modifizieren zu können. Für die Einstellung der Simulationsparameter soll eine Editor Oberfläche erstellt werden. Über diese Oberfläche hat der Benutzer die Möglichkeit vor und während der Simulation, Parameter wie die maximale Geschwindigkeit der Fahrzeuge, Beschleunigung der Fahrzeuge, Einfahrtsrate der Fahrzeuge oder Ampelschaltzeiten anzupassen. Die Simulation soll in einer zwei- oder dreidimensionalen graphischen Oberfläche dargestellt werden. Der Benutzer soll aus diversen Kartentypen, welche persistent gespeichert sein sollen, zu Beginn der Simulation auswählen können.

In den weiteren Lehreinheiten wurden weitere Aufgaben definiert. So sollte eine gruppenübergreifende Kommunikation möglich sein, Autos sollen von einer Simulation in die nächste fahren können.

Eine weitere Aufgabenstellung war die Möglichkeit ein Hinderniss in die Fahrbahn zu platzieren. Dies sollte frei möglich sein (also zur Laufzeit). Ein Auto soll dieses Hindernis umfahren können und gleichzeitig eine Kollision mit einem anderen Auto verhindern.

2 Software-Architektur

In diesem Kapitel wird die Architektur in Form des Arc42-Templates sowohl grafisch als auch textuell dargestellt.

2.1 Randbedingungen

Die Verwendung der Programmiersprache C# und die Auslagerung der Logik für geregelte Kreuzungen sind als Vorgaben für die Realisierung der Verkehrssimulation gegeben.

Es sollen verschiedene Einstellungen zur Laufzeit geändert werden können, um die Simulation entsprechend zu strapazieren. Dazu gehören:

- Anzahl der maximalen Fahrzeuge während der Simulation
- Maximale Geschwindigkeit der Fahrzeuge
- Die Zeit in der neue Fahrzeuge erstellt werden
- Verhältnis zwischen PKW und LKW

Des Weiteren soll über eine vorab definierte Schnittstelle ein Austausch von Fahrzeugen innerhalb der Gruppen erfolgen können.

2.2 Kontextabgrenzung

Einschränkungen im Detailgrad sowie im Umfang der Implementierung wurde keine vorgegeben. Es soll nur die Aufgabe mit den vorgegebenen Randbedingungen erfüllt werden. Wie diese umgesetzt werden, ist dem Projektteam überlassen.

2.3 Lösungsstrategie

Da ein Teil der Projekt Mitarbeiter bereits Erfahrungen mit Unity gemacht haben, wurde überlegt, diesen auf für die Verkehrssimulation zu verwenden. Durch die Verwendung des Unity Editors, welche als Laufzeit- und Entwicklungsumgebung für Spiele dient, wird ein Großteil bereits von der von Unity zur Verfügung gestellten "Game Engine abgenommen. Da Unity eine 2D bzw. 3D Game Engine zur Verfügung stellt wurde überlegt, welche von beiden die meisten Vorteile für eine Verkehrssimulation bringt. Da jedoch der Unterschied nur in der grafischen Darstellung liegt, was bedeutet das Logik von Autos, Straße, Ampeln usw. sowohl in 2D, als auch in 3D implementiert hätte werden müssen. Da eine Simulation mit 3D Modellen ansprechender aussieht, fiel die Entscheidung auf die 3D Implementierung.

Eine weitere Lösungsstrategie ist das Aufteilen der Aufgabenbereiche. Folgende Teilbereiche für die Implementierung wurden überlegt:

- Ampelsteuerung
- Erstellung von Straßen und Kreuzungen
- Logik und Verhalten in Fahrzeugen
- Straßenlogik (Waypoints)

2.4 Bausteinsicht

In diesem Abschnitt folgt die Beschreibung der Komponenten und in Abbildung 2.1 ist das Komponenten Diagramm zur Verkehrssimulation abgebildet.

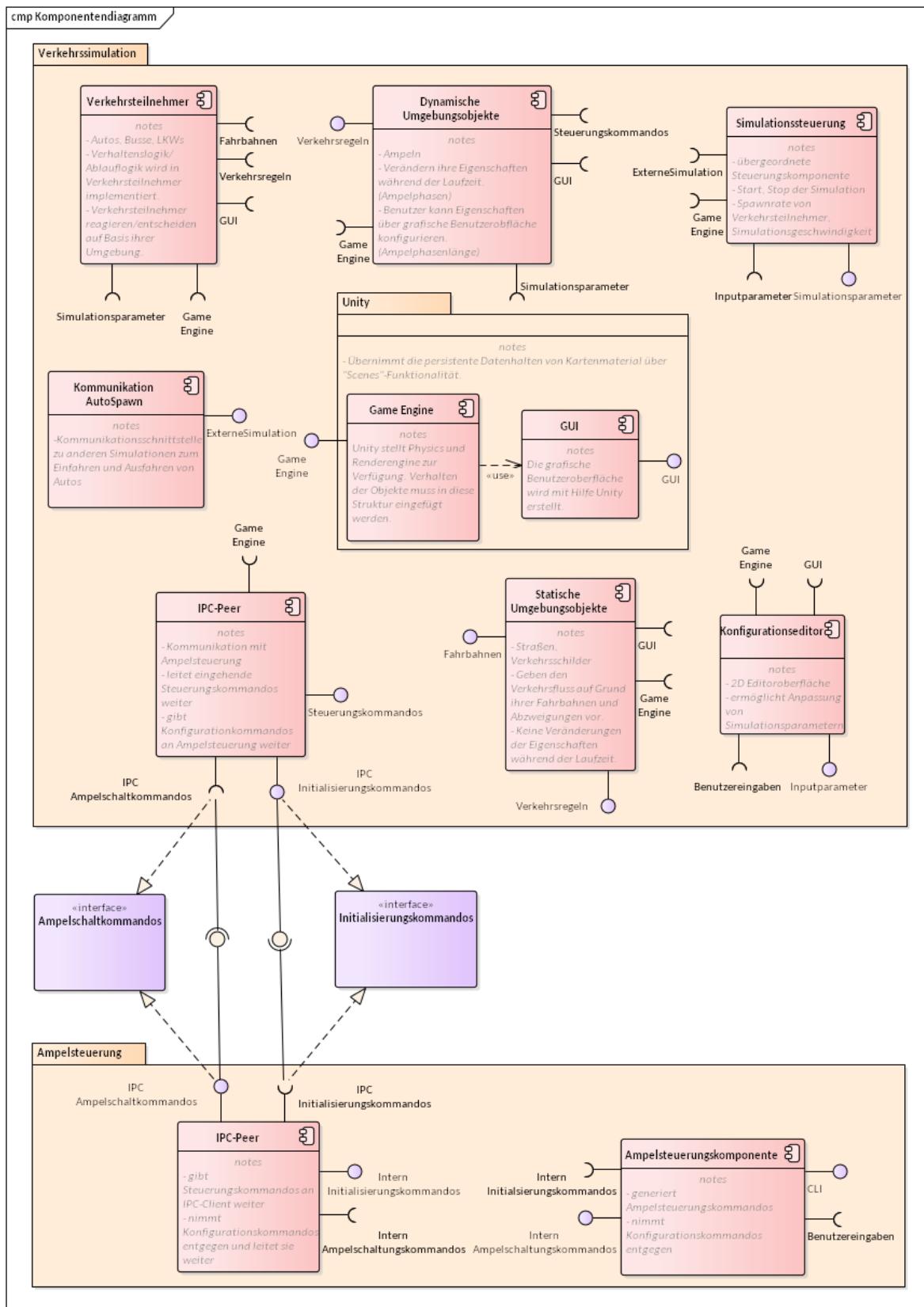


Abbildung 2.1: Komponenten Diagramm Verkehrssimulation

Verkehrssimulation und Ampelsteuerung

Sowohl Verkehrssimulation als auch Ampelsteuerung sind eigene Executables. Die Verkehrssimulation wird aus dem Unity-Projekt erzeugt und beinhaltet alle oben angegebenen Komponenten. Die Ampelsteuerung ist eine Konsolen Applikation, welche über die IPC-Komponenten mit der Verkehrssimulation kommuniziert.

Verkehrsteilnehmer

Zu diesen Komponenten gehören alle sich aktiv in der Simulation bewegenden Objekte, wie Autos, Busse und LKWs. Die einzelnen Komponenten bewegen sich autonom voneinander im Straßennetz fort. Die Komponenten scannen ihre Umgebung auf Fahrbahnen, Verkehrsregeln und andere Verkehrsteilnehmer. Auf Basis dieses Scans entscheiden sie ihre nächste Aktion, wie zum Beispiel Beschleunigen, Bremsen oder Abbiegen. Verkehrsteilnehmer können sich ausschließlich auf Fahrbahnen bewegen und beachten alle Verkehrsregeln innerhalb ihres Aktionsradius. Weiters benötigen sie Simulationsparameter, die über ihre maximale Geschwindigkeit, Beschleunigung und Bremsverhalten entscheiden. Verkehrsteilnehmer treffen ihre Entscheidungen bei jedem Renderdurchlauf der Game Engine. Die Verknüpfung der Verkehrsteilnehmer mit der GUI erfolgt abstrahiert durch Unity.

Dynamische Umgebungsobjekte

Dynamische Umgebungsobjekte sind Objekte, welche während der Simulationslaufzeit ihre Eigenschaften ändern, jedoch nicht ihre Position. Konkret sind dynamische Umgebungsobjekte Ampeln. Ampeln bieten den umliegenden Verkehrsteilnehmern ihren derzeitigen Status als Verkehrsregeln an, welche diese beachten müssen. Dynamische Elemente benötigen Simulationsparameter, welche ihre Schaltzeiten vorgeben. Konkret geben die Simulationsparameter die Phasenzeiten der Ampeln an. Weiters benötigen dynamische Umgebungsobjekte Steuerungskommandos um zwischen diversen Modi zu wechseln. Über Steuerungskommandos können Ampeln von automatischen Betrieb in einen inaktiven dauerhaft Gelb blinkenden Status gebracht werden. In jedem Renderzyklus der Game Engine wird auf neue Steuerungskommandos geprüft und falls nötig der vorgegebene Schaltvorgang eingeleitet. Die Verknüpfung der dynamischen Umgebungsobjekte mit der GUI erfolgt abstrahiert durch Unity.

Simulationssteuerung

Die Simulationssteuerung stellt eine übergeordnete Kontrollinstanz der Simulation dar. Sie legt globale Einstellungen für Simulationskomponenten zentral fest. Die Simulationssteuerung bietet Simulationsparameter für andere Simulationskomponenten an, welche diese abrufen können. Konkrete Simulationsparameter sind Schaltzeiten für Ampeln, Höchstgeschwindigkeiten für Verkehrsteilnehmer oder Spawnraten für neue Verkehrsteilnehmer. Die Simulationssteuerung benötigt Inputparameter, welche von außen die Simulationsparameter bestimmen. Die Aktualisierung der Simulationsparameter auf Basis der Inputparameter erfolgt bei jedem Renderzyklus der Game Engine.

Konfigurationseditor

Der Konfigurationseditor stellt eine zweidimensionale graphische Benutzeroberfläche dar, welche es dem Benutzer ermöglicht angebotene Inputparameter für die Simulation zu verändern. Vom Benutzer geänderte Inputparameter werden bei jedem Renderzyklus der Game Engine verarbeitet und entsprechend weiter geleitet.

IPC-Peer Verkehrssimulation

Der IPC-Peer Verkehrssimulation repräsentiert eine Instanz der Inter Process Communication zwischen den Executables Verkehrssimulation und Ampelsteuerung dar. Dieser IPC-Peer gibt Initialisierungskommandos an die Ampelsteuerung weiter. Über diese Kommandos werden die benötigte Anzahl an Ampeln mit den korrekten Initialisierungsparametern angelegt. Weiters werden Ampelschaltkommandos entgegen genommen und weiter geleitet um den Modus einer Ampel zu wechseln. Ein- und ausgehende Kommandos während in jedem Renderzyklus der Game Engine bearbeitet.

Statische Umgebungsobjekte

Statische Umgebungsobjekte repräsentieren Simulationskomponente, welche weder ihre Eigenschaften noch ihre Position während der Simulationslaufzeit verändern. Konkret sind Straßen und Verkehrsschilder statische Umgebungsobjekte. Sie bieten Fahrbahnen für die Verkehrsteilnehmer an, auf welchen diese sich bewegen können. Zusätzlich werden auch Verkehrsregeln vorgegeben, welche von den Verkehrsteilnehmer beachtet werden müssen. Die statischen Umgebungsobjekte werden von der Game Engine gerendert, jedoch sollte diese Komponente keine Logik besitzen. Die Verknüpfung der statischen Umgebungsobjekte mit der GUI erfolgt abstrahiert durch Unity.

IPC-Peer Ampelsteuerung

Die Komponente IPC-Peer Ampelsteuerung bildet die Gegenstelle der zuvor beschriebenen IPC-Peer Verkehrssimulation. Sie gibt Ampelschaltkommandos zur Gegenstelle weiter und nimmt Initialisierungskommandos entgegen. Diese Kommandos werden intern, innerhalb der Ampelsteuerung Executable an die Ampelsteuerungskomponente weiter gegeben.

Ampelsteuerungskomponente

Die Ampelsteuerungskomponente übernimmt die Verwaltung der einzelnen Ampelinstanzen. Anhand eingehender Initialisierungskommandos werden Ampelinstanzen mit den benötigten Initialisierungsparametern erstellt. Über das angebotene CLI kann der Benutzer Schaltbefehle absetzen, welche über Ampelschaltungskommandos weiter geleitet werden.

Message Queue

Die Message Queue übernimmt die Kommunikation mit den anderen Gruppen. Die Kommunikation mit den anderen Gruppen wird über einen externen Server abgewickelt, auf denen sich die Gruppen anmelden und ihre Queues abonnieren können. Ein Format, basierend auf JSON wurde zwischen den Gruppenteilnehmern definiert.

Als Protokoll wird dabei AMPQ verwendet, der dazugehörige Server, RabbitMQ implementiert diese Protokoll und ermöglicht die Übertragung von Nachrichten.

2.5 Laufzeitsicht

Die Ampelsteuerung wird beim Start der Verkehrssimulation initialisiert. Danach wird in regelmäßigen Abständen die Ampelsteuerung von der Verkehrssimulation gepolst und der Status der Ampeln abgefragt. Dieser Ablauf ist im Sequenz Diagramm in Abbildung 2.2 dargestellt.

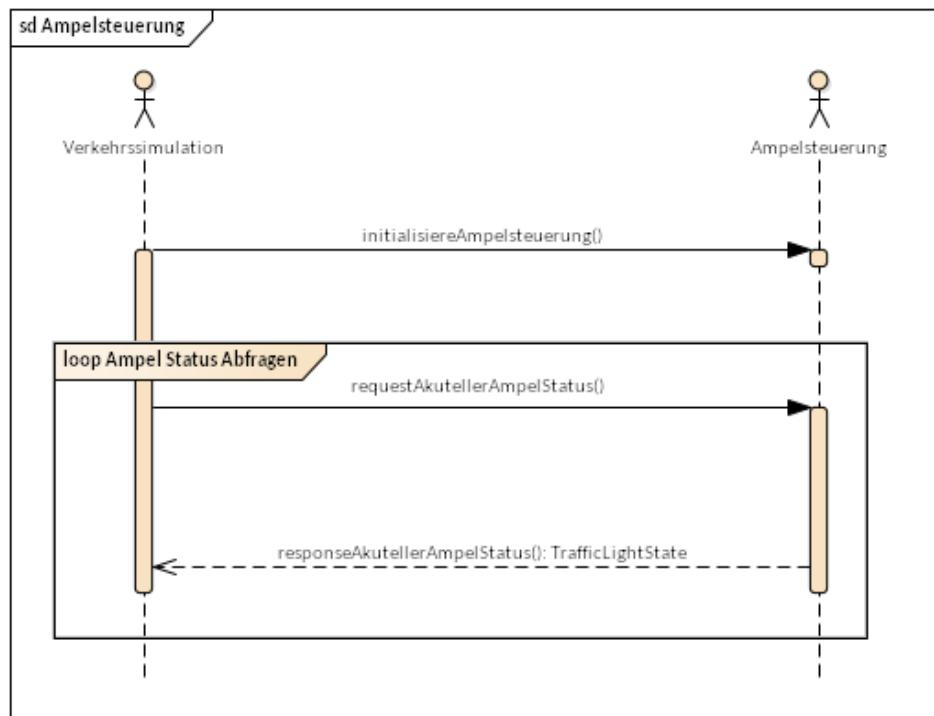


Abbildung 2.2: Sequenz Diagramm Verkehrssimulation - Ampelsteuerung

Bei der Kommunikation mit anderen Simulation zum Austausch von Fahrzeugen wird auf ein Messaging System gesetzt. Das bedeutet, dass ankommende Fahrzeuge über einen Asynchronen Push Aufruf empfangen werden und ausfahrende Fahrzeuge einen den Server gesendet werden (Abbildung 2.3).

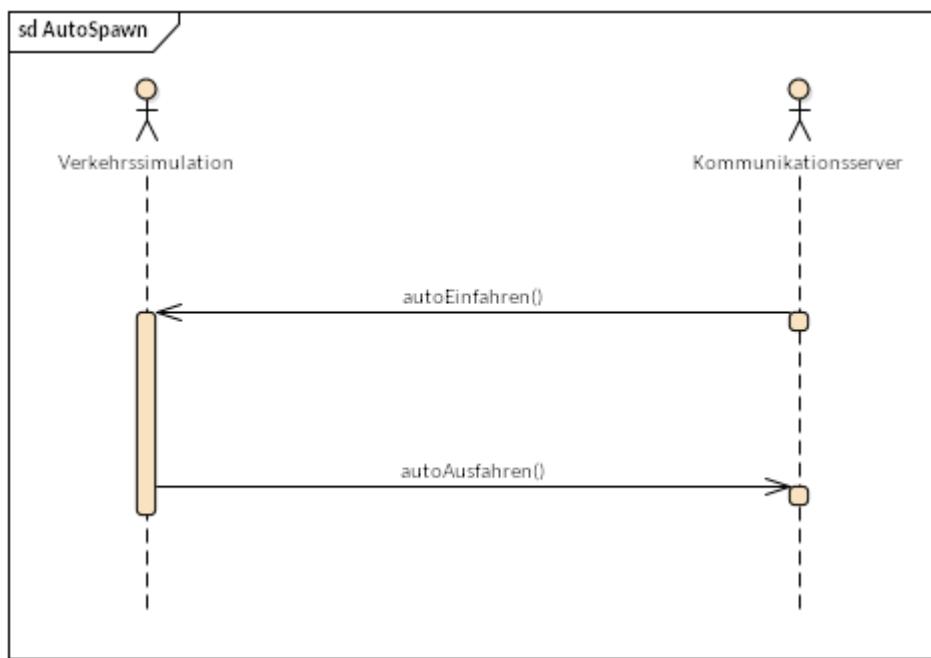


Abbildung 2.3: Sequenz Diagramm Verkehrssimulation - Ampelsteuerung

2.6 Verteilungssicht

Die Komponenten Ampelsteuerung und Verkehrssimulation sind auf zwei Geräte aufgeteilt. Die Ampelsteuerung läuft zentral auf einem Server und die Verkehrssimulation kann lokal auf einem PC ausgeführt werden. Weiters wird zur Kommunikation mit anderen Simulationen ein weiterer Server benötigt. Die Verteilung ist im Deployment Diagramm in Abbildung 2.4 abgebildet.

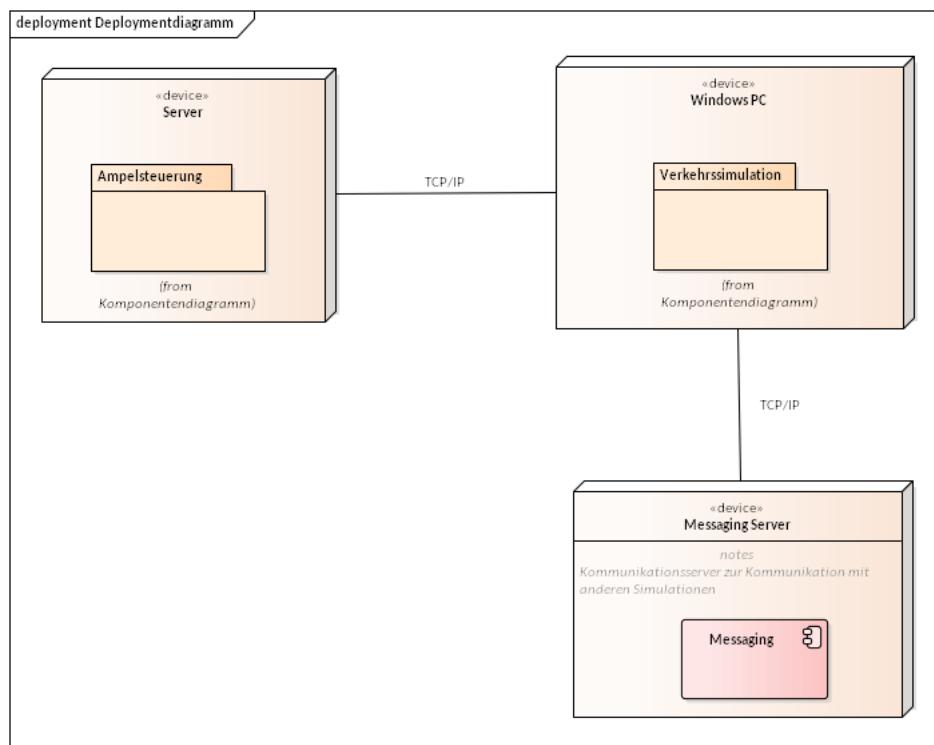


Abbildung 2.4: Deployment Diagramm der Verkehrssimulation

2.7 Querschnittliche Konzepte

Straßen und Kreuzungen werden als GameObjects abgespeichert. Dadurch wird ermöglicht, dass die erstellten Teilstücke wiederverwendet werden können. Somit kann die Erstellung der Welt per "Drag & Drop" und genaues Ausrichten erfolgen. Folgende Straßenstücke wurden in verschiedenen Längen erstellt:

- T-Kreuzung (mit Ampeln)
- X-Kreuzung (mit Ampeln)
- Spawn/Despawn-Zone
- Gerade Teilstrecke (5m, 6m, 24m, 30m, 40m, 50m, 60m)
- 90 Grad Kurve

Die damit wiederverwendbaren Teilstücke beinhalten bereits die benötigten Kollider, welche für die Fahrzeuglogik verwendet werden, sowie Shaders zum Darstellen der Straßenstücke.

Auch das Verwenden der Ampelsteuerung funktioniert über eine definierte Schnittstelle. Hierbei können einfach neue Kreuzungen mit Ampeln erstellt werden und die Ampelsteuerung regelt automatisch die neu erstellte Ampel.

2.8 Entwurfsentscheidungen

Wie in Kapitel 2.3 bereits kurz beschrieben wurde die Unity Engine als treibende Engine verwendet, da diese bereits viele Sachen implementiert, welche von uns nicht mehr berücksichtigt werden müssen (z.B. Handling wann wird welches GameObject angesprochen usw.). Ein weiterer Grund für die Entscheidung mit Unity war die schnelle und anschauliche Darstellung von 3D-Modellen, sowie die zahlreich vorhandenen Tutorials über Unity und die verschiedenen Möglichkeiten.

2.9 Qualitätsszenarien

Keine Vorhanden, da es sich hierbei nur um ein Studierendenprojekt handelt.

2.10 Risiken und technische Schulden

Da die Verkehrssimulation nun zur Genze mit der Unity-Engine funktioniert, kann des im Falle von Updates von Unity vorkommen, das API-Updates durchgeführt werden und diese nicht mehr mit den bereits eingebauten funktionierenden Funktionen übereinstimmen. Dieses führt daraufhin zu Mehraufwand, der berücksichtigt werden müsste.

3 Design

In diesem Kapitel werden die Implementierungen der einzelnen Komponenten sowie deren Schnittstellen zu anderen Komponenten dokumentiert.

3.1 Strassensystem

Das Strassensystem besteht aus drei Grundelementen den Ampeln, Strassen und Kreuzungen.

3.1.1 Ampeln

Die Ampeln wurden von uns als eigenständiges System geplant und umgesetzt. Um dieses Ziel zu erreichen erstellten wir zuerst auf Basis der Technologie TCP-Channel ein RemoteObject in Form einer eigenen DLL. Dieses RemoteObject definiert die Datenstrukturen, welche in weiterer Folge zwischen Client und Server geteilt werden. Als Serveranwendung erstellten wir eine Konsolenanwendung, welche wir auf einem öffentlich erreichbaren Linux Server mit Hilfe von Mono deployten. Als Clientgegenstellen für dieses System können verschiedene Unityinstanzen den Dienst der Serveranwendung verwenden. Sowohl am Client als auch am Server ist als DLL das RemoteObject hinterlegt. Instanzen des RemoteObjects können mit Hilfe am Server definierten Interfaces von den Clients erstellt und bearbeitet werden. Die somit erstellten Instanzen werden dann durch die TCP-Channel Technologie zwischen Client unser Server geteilt. Somit können Clients komfortabel mit ihrem lokalen Stub des RemoteObjects arbeiten ohne die Komplexität der IPC mitzubekommen.

Durch die zentrale Definition des RemoteObjects in Form einer DLL können die geteilten Datenstrukturen zentral erweitert und gewartet werden.

3.1.2 Straßen

Die Straßen sind jene Element auf denen sich die Fahrzeuge bewegen sollen. Dabei soll in diesen keine Logik enthalten sein, um der realen Welt nahe zu kommen und das System einfach zu halten. Es gibt unterschiedliche Typen von Straßenelementen. Dazu zählen Geraden und Kurven. Damit sich die Fahrzeuge auf den Straßen bewegen werden sogenannte Wegpunkte auf den Straßen platziert. Die Fahrzeuge fahren somit von einem zum nächsten Wegpunkt. In Kurven sind mehrere Wegpunkte nötig, um eine schöne Bewegung zu simulieren. Die Wegpunkte werden jeweils in eine positive und eine negative Richtung gesetzt für die beiden Fahrtrichtungen. Am Beginn jedes Straßenelements befindet sich ein Kollider, der dem Fahrzeug mitteilt, auf welcher Straßenseite er sich befindet.

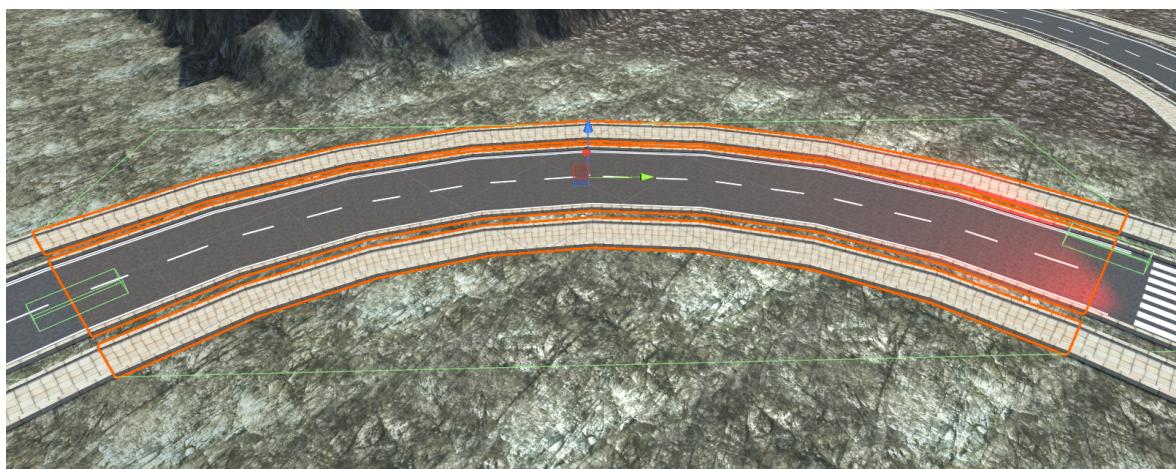


Abbildung 3.1: Collider an normaler Straße

3.1.3 Kreuzungen

Die Kreuzungen unterscheiden sich in T- und X-Kreuzungen und werden auch nur mit der nötigsten Logik ausgestattet. Dabei kann beim erstellen zwischen geregelten und ungeregelten Kreuzungen unterschieden werden. Werden geregelte Kreuzungen erstellt, wird automatisch die Ampelsteuerung initialisiert und die Kreuzung regelt auch das Darstellen der Farben für die Ampeln. Weiters besitzt jede Kreuzung beim Einfahren einen Kollider um den Fahrzeugen mitzuteilen, dass sie sich einer Kreuzung nähren. Das Fahrzeug kann danach von der Kreuzung den Ampelstatus abfragen. Weiters erhält das Fahrzeug nachdem es sich für die neue Fahrtrichtung entschieden hat die neuen Wegpunkte und Strassenelemente für die Weiterfahrt.



Abbildung 3.2: Collider an Kreuzung

3.1.4 Spawn

Hier werden je nach Einstellung der Controls die Fahrzeuge gespawnt bzw despawnt, sobald sie mit dem in Abbildung 3.3 gezeigten Collider in Berührung kommen.

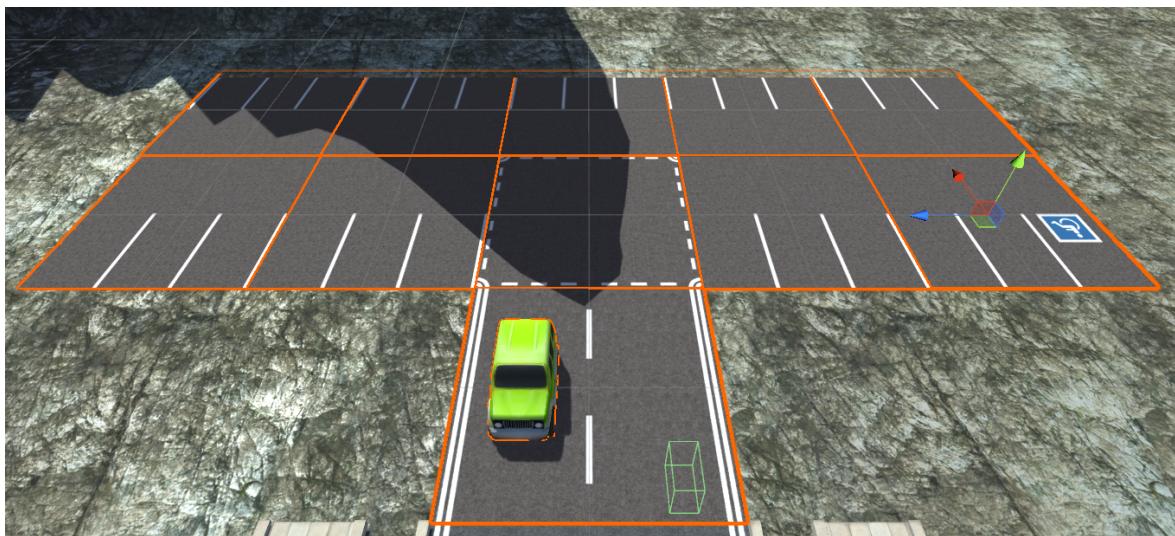


Abbildung 3.3: Spawn der Fahrzeuge

3.2 Fahrzeuglogik

Die Fahrzeuge implementieren alle ein Skript, welches für das Erkennen von Ampeln, Fahrzeugen und sonstiges Hindernissen zuständig ist. Dies ermöglicht eine völlige Kapselung und somit die Unabhängigkeit zur Straße und Kreuzung. Das bedeutet, dass Straßen und Kreuzungen nur sogenannte "Collider" Verfügbar stellen, mit denen die Fahrzeuge interagieren können. Somit bleibt die gesamte Intelligenz in den Fahrzeugen.

Diese Collider werden je nach Geschwindigkeit des Fahrzeugs länger bzw. größer (wie in Abbildung 3.4 gezeigt) um somit auf Hindernisse wie andere Fahrzeuge, Kreuzungen und sonstiges frühzeitig reagieren zu können.

Das Erkennen und Verringern der Geschwindigkeit auf andere Fahrzeuge wird mittels Raycast erledigt. Die Lösung mit den Collidern war ein Problem, da sich die Fahrzeuge, nicht mehr bewegt haben, nachdem sie kollidiert sind.

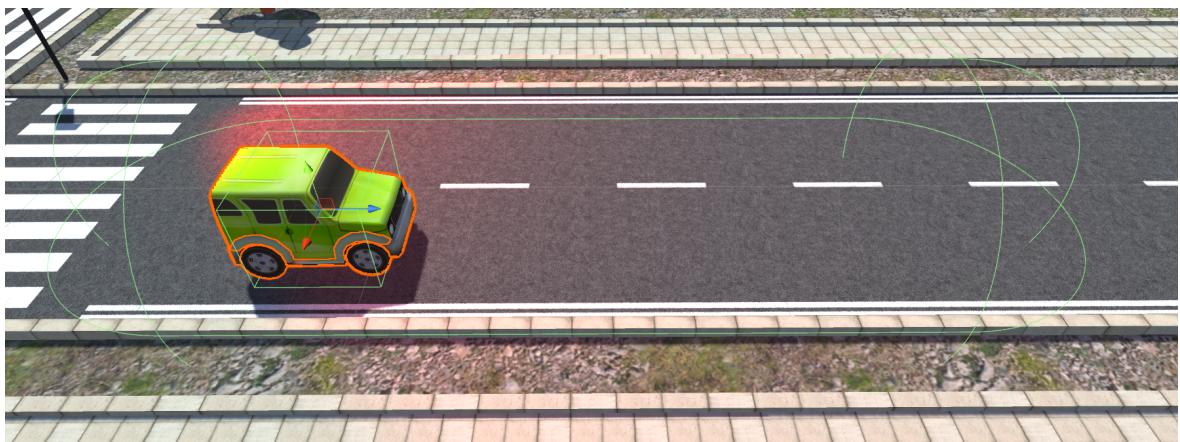


Abbildung 3.4: Collider an Fahrzeugen

3.3 Ampelsteuerung

Die zentrale Komponente der Ampelsteuerung ist das RemoteObject. Wie bereits erwähnt handelt es sich dabei um eine eigenständige Komponente, welche in Form einer DLL in diverse Projekte leicht eingebunden und zentral verwaltet werden kann. Das RemoteObject repräsentiert die Datenstrukturen und Funktionalitäten, welche zwischen Client und Server mit Hilfe der TCP-Channel Technologie geteilt werden.

Das RemoteObject bietet die Möglichkeit eine beliebige Anzahl von Kreuzungen zu erstellen und zu verwalten. Die Logik für den Aufbau der IPC-Verbindung wird dabei direkt im Client oder Server hinterlegt. Mit Hilfe von definierten Interfaces hat der Client die Möglichkeit Kreuzungen mit drei oder vier Ampeln am Server-Skeleton anzulegen. Die Kreuzungen werden entweder mit einer festgelegten Standardkonfiguration oder mit den vom Entwickler

angegeben Übergabeparametern angelegt.

Von bereits angelegten Kreuzungen können die Zykluszeiten der einzelnen Ampeln editiert werden. Weiters ist es möglich das RemoteObject zu reseten und somit den Initialzustand wiederherzustellen.

Die einzelnen Kreuzungen, welche am RemoteObject angelegt sind können separat gestartet werden. Beim Starten einer Kreuzung wird ein eigener Thread für jede Ampel der Kreuzung erzeugt in dem eine StateMachine für die Ampelschaltung abläuft. Dieser Thread läuft bis entweder die Serveranwendung terminiert wird oder die Kreuzung durch einen Reset vom Client gelöscht wird. Die StateMachine einer Ampel durchläuft die üblichen Stati und berücksichtigt dabei die Konfiguration der entsprechenden Ampel.

Mit Hilfe der ID einer Kreuzung und der Bezeichnung einer konkreten Ampel der Kreuzung kann der Status einer Ampel abgefragt werden.

Durch die Verwendung der TCP-Channel Technologie und der Definition des RemoteObjects in Form einer eigenen DLL wurde die Komplexität einer IPC sehr gut abstrahiert. Leider erfuhren wir während den Tests, dass die von uns verwendete Technologie sehr anfällig gegenüber schlechten Latenzzeiten ist. Dies wirkt sich in Form von Verbindungsproblemen zwischen Client und Server aus. Daher würde ich die Technologie TCP-Channel für IPC nur innerhalb eines lokalen Netzwerkes empfehlen.

3.4 Hindernisse

Mit einem Klick der Maus sollen überall in der Welt Hindernisse platziert werden, welche anschließend von Fahrzeugen umfahren werden müssen. Durch einen weiteren Klick auf ein Hindernis soll dieses wieder gelöscht werden.

Unity ermöglicht das Erstellen von GameObjects zur Laufzeit. Durch den Klick auf die Welt können die Koordinaten des Klicks festgestellt werden und somit das GameObject, welches bereits beim Start der Verkehrssimulation geladen wurde, platziert werden.

Durch das Halten eines Buttons auf der Tastatur soll die Erstellung von Hindernissen aktiviert werden. Dies verhindert, dass der Benutzer unbeabsichtigt zu viele Hindernisse erstellt.

3.5 Aus- und Einfahren von Fahrzeugen anderer Gruppen

Die Message Queue übernimmt die Kommunikation mit den anderen Gruppen. Die Kommunikation mit den anderen Gruppen wird über einen externen Server abgewickelt, auf denen sich die Gruppen anmelden und ihre Queues abonnieren können. Ein Format, basierend auf JSON wurde zwischen den Gruppenteilnehmern definiert.

Als Protokoll wird dabei AMPQ verwendet, der dazugehörende Server, RabbitMQ implementiert diese Protokoll und ermöglicht die Übertragung von Nachrichten.

4 Implementierung

In diesem Kapitel werden die Implementierungen der einzelnen Komponenten sowie deren Schnittstellen zu anderen Komponenten dokumentiert.

4.1 Erstellung der Welt

In der Verkehrssimulation können beliebige Strassennetze gebaut werden. Dazu stehen Verschiedene vordefinierte Elemente zur Verfügung. Dazu zählen T- und X-Kreuzungen, Kurven, Geraden und Spawn Flächen. In den Elementen sind alle benötigten Komponenten und Skripte vorhanden und diese können einfach mit Drag and Drop in die Simulation gezogen und platziert werden. Es ist nur darauf zu achten, dass alle Strassenelemente welche zusammengehören, also zwei Kreuzungen verbinden in einem übergeordneten GameObject zusammengefasst werden. Ein Beispiel ist in Abbildung 4.1 dargestellt.



Abbildung 4.1: Aufbau des Strassennetzes

Im rechten Teil der Abbildung sind die Strassenelemente abgebildet. In diesem Beispiel lautet der Übergeordnete Strassenname "Puch". In Diesem GameObject "Puch" sind alle Teilstücke der Strasse vorhanden. Diese müssen dabei in auf- oder absteigender Reihenfolge angeordnet sein.

Kreuzungen können auch beliebig mit Drag and Drop eingefügt werden. Dabei müssen die Straßen den Kreuzungen bekannt gegeben werden. Beim Klick auf das jeweilige erzeugte Element kann rechts in den Skript Einstellungen kann ausgewählt werden, ob die Kreuzung geregelt oder ungeregelt sein soll. Weiters müssen die angeschlossenen Straßen eingefügt werden. Ein Beispiel ist in Abbildung 4.2 dargestellt. Es wurde das vorher erstellte Strassen-element angeschlossen



Abbildung 4.2: Einstellungen für eine Kreuzung

Die Richtungen (Pos X, Neg Y, etc.) sind an das Koordinaten System des Kreuzungselementes angelehnt. Bei einem Klick auf die Kreuzung erscheint das Koordinaten System. Daran kann erkannt werden, an welcher Stelle die jeweilige Strasse eingetragen werden muss.

4.2 Fahrzeugsteuerung

Die Fahrzeugsteuerung ist zuständig für das Folgen der Strasse und das richtige Abbiegen des Autos. Diese Funktionen werden im Skript 'FollowWay.cs' des Fahrzeugs verwirklicht.

4.2.1 Folgen der Strasse

Wie in Kapitel 3.1.2 beschrieben befinden sich auf jeder Strasse Wegpunkte. Fährt ein Fahrzeug auf eine neue Strasse bekommt es von der Kreuzung die Richtung und das GameObject der Strasse übergeben. Das Fahrzeug bewegt sich dabei von Wegpunkt zu Wegpunkt mithilfe des in 4.1 dargestellten Codes.

```

1  dir = targetPathNode.position;
2  dir = dir - this.transform.localPosition; //Berechnung der Richtung
3
4  float distThisFrame = speed * Time.deltaTime; //Berechnung der zu
   bewegenden Distanz
5
6  transform.Translate(dir.normalized * distThisFrame, Space.World);
   //Verschieben des Objekts
7  Quaternion targetRotation = Quaternion.LookRotation(dir);
8  this.transform.rotation = Quaternion.Lerp(this.transform.rotation,
   targetRotation, Time.deltaTime * rotationSpeed); //Rotieren des
   Objekts

```

Listing 4.1: Transformation der Fahrzeuge

Die Berechnung der Verschiebung erfolgt aufgrund der aktuellen Position des Fahrzeugs und der Position des Wegpunktes. Dabei wird auch die aktuelle Geschwindigkeit des Fahrzeuges einbezogen.

Das Bewegen des Fahrzeugs findet in der 'update()' Methode statt. Diese wird regelmäßig von der Game Engine aufgerufen. Da die Abstände dieses Aufrufs variieren können wird auch die Zeit des letzten Aufrufs in die Berechnung einbezogen.

Befindet sich das Fahrzeug auf einem der Wegpunkte wird in der Methode 'getNextStreetPart()' der nächste Wegpunkt bestimmt. Es können dabei drei Fälle auftreten. Befindet sich auf der aktuellen Strasse noch ein Wegpunkt wird dieser angefahren. Befindet sich auf diesem Strassenelement kein Wegpunkt mehr, wird das nächste Strassenelement aus der von der Kreuzung erhaltenen Strasse gewählt und aufgrund der Richtung (Pos od. Neg) wird auf dem neuen Strassenelement der nächste Wegpunkt gewählt.

Ist kein neues Strassenelement vorhanden, fährt das Fahrzeug auf eine Kreuzung zu und es bekommt die nächsten Wegpunkte bei der Kollision mit den Collidern der Kreuzung.

4.2.2 Abbiegen bei Kreuzungen

Fährt ein Fahrzeug auf eine Kreuzung zu, kollidiert es mit dem an der Kreuzungs Einfahrt befindlichen Collider. Dadurch wird der in Listing 4.2 dargestellte Code ausgeführt.

```

1 public void decideWay(CrossingColliderX collider)
2 {
3     if (!isNewStreet)
4     {
5         System.Random random = new System.Random();
6         int randomNumber = random.Next(0, 3);
7         nextCrossingColliderT = null;
8         nextCrossingColliderX = collider;
9         collider.setDirection(randomNumber, this);
10    }
11
12 }
```

Listing 4.2: Abbiegeentscheidung auf einer Kreuzung

Zuerst entscheidet das Fahrzeug mithilfe einer Zufallszahl in welche Richtung das Fahrzeug fahren soll. Danach wird eine Methode des Kreuzungs Skripts aufgerufen. Als Parameter werdenr Information für die nächste Richtung (die vorher erstellte ZZ) und das eigene Objekt für ein Callback übergeben. Im Skript der Kreuzung wird aufgrund dieser Informationen durch den Aufruf einer Methode des Fahrzeugs die nächsten Wegpunkte auf der Kreuzung und der nächste Strassenabschnitt weitergeben.

4.3 Erkennen und Reagieren auf Ampeln

Das Event 'OnTriggerEnter' wird beim kollidieren mit Collidern generiert. In diesem Event, welches im 'FollowWay.cs' Skript implementiert wurde, wird überprüft, zu wem der kollidierte Collider gehört. Wird hierbei ein Ampelcollider erkannt, so wird dieser zwischengespeichert, sonst werden noch die anderen Collider auf der Kreuzung erkannt und es kommt zum Fehlverhalten im Fahrzeug. Die Bezeichnungen 'PosX', 'NegX', 'PosY' und 'NegY' beschreiben hierbei die Ampelpositionen.

```

1 void OnTriggerEnter(Collider collider)
2 {
3     GameObject itself = gameObject;
4     GameObject collidedObject = collider.gameObject;
5
6     float distance = getDistance(itself, collidedObject);
7     if((collidedObject.name.Equals("PosX") ||
8         collidedObject.name.Equals("NegX") ||
9         collidedObject.name.Equals("PosY") ||
10        collidedObject.name.Equals("NegY")) && (firstCollider == null))
11    {
12        //Here the car collided with a crossing collider
13        crossingCurrent =
14            collider.transform.parent.gameObject.transform.parent.gameObject;
15            //The current crossing?!
16        firstCollider = collidedObject;
17    }
18 }
```

Listing 4.3: Erstes erkennen einer Ampel

Nun wird das Event 'OnTriggerStay' aufgerufen, da sich das Fahrzeug immer noch im Collider der Ampel befindet. In diesem Event wird zwischen den zwei verschiedenen Kreuzungstypen, T- und X-Kreuzung unterschieden. Von diesen Kreuzungen wird sich dann eine Referenz geholt und den aktuellen Status der Ampel abgefragt. Je nach Status der Ampel verhält sich das Fahrzeug anders. Diese Verhalten wird in der 'carDecisionOnCrossing' Funktion festgelegt. Folgende Ampelverhalten wurden festgelegt:

- **Grün:** Beschleunigen
- **Blinkend Grün:** Beschleunigen
- **Gelb:** Bremsen
- **Blinkend Gelb:** Geschwindigkeit beibehalten

- **Rot Gelb:** Beschleunigen

- **Rot:** Bremsen

```

1 void OnTriggerStay(Collider collider)
2 {
3     GameObject collidedObject = collider.gameObject;
4
5     if(collidedObject.name.Equals("PosX") ||
6     collidedObject.name.Equals("NegX") ||
7     collidedObject.name.Equals("PosY") ||
8     collidedObject.name.Equals("NegY"))
9     {
10         CrossingColliderT crossT = null;
11         CrossingColliderX crossX = null;
12
13         float distance = getDistance(gameObject, collidedObject);
14         if(collidedObject == firstCollider)
15         {
16             crossT = collidedObject.GetComponentInParent<CrossingColliderT>();
17             if(crossT == null)
18             {
19                 crossX =
20                     collidedObject.GetComponentInParent<CrossingColliderX>();
21                 if(crossX == null)
22                 {
23                     return;
24                 }
25                 RemoteObject.Enum.TrafficLightsStatus trafficLightStatus =
26                     crossX.actLightState;
27                 carDecisionOnCrossing(trafficLightStatus, distance);
28             }
29             else
30             {
31                 RemoteObject.Enum.TrafficLightsStatus trafficLightStatus =
32                     crossT.actLightState;
33                 carDecisionOnCrossing(trafficLightStatus, distance);
34             }
35         }
36     }
37 }
```

Listing 4.4: Bestehende Kollision mit Ampelcollider

Die an den Fahrzeugen sitzenden Collider werden in jedem Update Aufruf je nach Geschwindigkeit des Fahrzeuges vergrößert bzw. verkleinert, so kann auf Ampeln hingebremst werden und kein abruptes Stehenbleiben erzwungen werden.

4.4 Fahrzeug Kollisionserkennung

Jedes Fahrzeug implementiert das 'FollowWay.cs' Skript welches die komplette Logik über das Verkehrsverhalten besitzt. Innerhalb dieses Skripts wird in der Update-Methode, welche in jedem Frame pro Sekunde aufgerufen wird, überprüft, ob ein Raycast mit einem anderen Fahrzeug oder Hindernis kollidiert ist. Da dieser Raycast mehrere Objekte durchdringen kann, muss in einer Liste überprüft werden ob mit einem dieser Objekte kollidiert werden soll. Ist dies der Fall, so wird die Geschwindigkeit des Fahrzeugs verringert und der Raycast wird auf die Länge, welche der Geschwindigkeit des Fahrzeugs entspricht, gesetzt. Dazu wird die Distanz des zu kollidierenden Objekts und dem Fahrzeug ermittelt. Mit der Distanz wird das Fahrzeug dann entsprechend abgebremst und wenn nötig auch zum Stillstand gebracht. Um nicht in das vorherfahrende Objekt zu fahren, wird die Hälfte der Länge des Fahrzeugs noch von der Distanz abgezogen. Damit sich das Fahrzeug nicht selbst als kollidierendes Objekt erkennt, muss der Layer des Fahrzeugs für kurze Zeit geändert werden.

```

1 private void checkRaycast()
2 {
3     // Save current object layer
4     int oldLayer = gameObject.layer;
5     //Change object layer to a layer it will be alone
6     gameObject.layer = 12;
7     int layerToIgnore = 1 << 12;
8     layerToIgnore = ~layerToIgnore;
9
10    RaycastHit[] hits;
11    hits = Physics.RaycastAll(transform.position, transform.forward,
12        raycastSize, layerToIgnore);
12    bool somethingInFront = false;
13    for(int i = 0; i < hits.Length; i++)
14    {
15        RaycastHit hit = hits[i];
16        GameObject collidedObject = hit.collider.gameObject;
17        if(collidedObject.name.Equals("jeep(Clone)") ||
18            collidedObject.name.Equals("Rock(Clone)"))
19        {
20            float distance = getDistance(gameObject, collidedObject);
21            if(gameObject.name.Equals("jeep(Clone)"))
22            {
23                distance = distance - (lengthCar / 2 + 1f);

```

```

23         }
24     else
25     {
26         distance = distance - (lengthTanker / 2 + 1f);
27     }
28     brakeWithDistance(distance);
29     mayIdrive = false;
30     somethingInFront = true;
31 }
32 if(somethingInFront == false)
33 {
34     mayIdrive = true;
35 }
36 }
37 raycastSize = speed;
38 if(raycastSize < 1)
39 {
40     raycastSize = 1;
41 }
42 // set the game object back to its original layer
43 gameObject.layer = oldLayer;
44 }
```

Listing 4.5: Erkennen von anderen Fahrzeugen und Hindernissen

4.5 Ampelsteuerung

Die Ampelsteuerung wurde als eigene Applikation entwickelt. Die Ampelsteuerung kann grundsätzlich unabhängig von der Unity-Spielwelt laufen, auch die Ampeln schalten dementsprechend autonom, auch wenn die Simulation schon beendet sein sollte.

Die Ampelsteuerung ist auch mit Mono lauffähig. Mono ist eine C#-Implementierung für unixoide Betriebssysteme. So läuft die Ampelsteuerung auch mit Linux. Auch eine Kompilierung ist möglich, mit `xbuild solution.sln` wird eine ausführbare Datei erzeugt, die sowohl mit Linux als auch Windows lauffähig ist.

In Abbildung 4.3 ist die Ausgabe der Ampelsteuerung zu sehen. Da die Ampelsteuerung gut funktionierte, wurde in die Ausgabe weniger Zeit investiert, sodass hier nur die aufrufende IP ausgegeben wird.

```
78.104.198.114 created Intersection.
78.104.198.114 created Intersection.
78.104.198.114 created Intersection.
78.104.199.112 created Intersection.
```

Abbildung 4.3: Ausgabe Ampelserver

4.6 Erstellen und Löschen von Hindernisse

Um im Environment Hindernisse zur Laufzeit zu erstellen, muss dem Terrain-GameObject ein Skript (Obstacles.cs) hinzugefügt werden. In diesem Skript findet die Abarbeitung des Inputs des Users statt.

Wie bereits in 3.4 erläutert wird zum Start der Verkehrssimulation das Hindernis geladen. Dies erfolgt über den "Load" Befehl.

```
1 private GameObject prefabLog;
2 void Start()
3 {
4     prefabLog = Resources.Load("Rock", typeof(GameObject)) as GameObject;
5 }
```

Listing 4.6: Laden des Hindernisses

Der in 3.4 beschrieben soll ein Button gedrückt gehalten werden um Hindernisse spawnen zu können. Dieser wird über einen KeyCode definiert. Im Skript wird nun in jedem Update Aufruf darauf gewartet, ob der definierte Button gedrückt und ein "MouseDown"-Event vorkommt. Mit diesem Event kann die Position der Maus auf dem Bildschirm herausgefunden werden, jedoch stimmen diese nicht mit den Welt-Koordinaten überein. Deshalb muss hier eine Umwandlung durchgeführt werden, welche mit Hilfe eines Raycasts gelöst wurde. Durch das Instanzieren wird das neu erstellte Hindernis in der Welt platziert.

```
1 private KeyCode shiftLeft = KeyCode.LeftShift;
2 if(Input.GetMouseButtonDown(0) && Input.GetKey(shiftLeft)) //Left mouse
    button clicked
3 {
4     Vector3 mousePosition = Input.mousePosition;
5     var ray = Camera.main.ScreenPointToRay(mousePosition);
6     RaycastHit hit;
```

```

7     if(Physics.Raycast(ray, out hit, 1000f))
8     {
9         Vector3 position = hit.point;    Vector3 yOffset = new Vector3(0,
10            1.5f, 0);
11         position += yOffset;
12         GameObject prefabInstance = Instantiate(prefabLog, position, new
13             Quaternion()) as GameObject;
14     }
15 }
```

Listing 4.7: Erstellen des Hindernisses

Zum Löschen eines Hindernisses muss die rechte Maustaste in Kombination mit dem vorher definierten Button verwendet werden. Mittels "Destroy" wird anschließend das erkannte GameObject wieder von der Welt gelöscht.

```

1  if(Input.GetMouseButtonDown(1) && Input.GetKey(shiftLeft)) //Right mouse
   button clicked
2  {
3      Vector3 mousePosition = Input.mousePosition;
4      var ray = Camera.main.ScreenPointToRay(mousePosition);
5      RaycastHit hit;
6      if(Physics.Raycast(ray, out hit, 1000f))
7      {
8          Vector3 position = hit.point;
9          GameObject collidedObject = hit.collider.gameObject;
10         if(collidedObject.name.Equals("Rock(Clone)"))
11         {
12             Destroy(collidedObject);
13         }
14     }
15 }
```

Listing 4.8: Zerstören des Hindernisses



Abbildung 4.4: Hindernis

4.7 Aus- und Einfahren von Fahrzeugen anderer Gruppen

Mit dem Plugin ‘Unity3D.Amqp’ (<https://github.com/CymaticLabs/Unity3D.Amqp>) für Unity ist es möglich einen RabbitMQ-Server direkt in Unity einzubinden.

Die Konfiguration der Serverdaten werden dabei direkt in den Menüeinstellungen von Unity vorgenommen. Die verwendete Konfiguration ist in Abbildung 4.5 zu sehen.

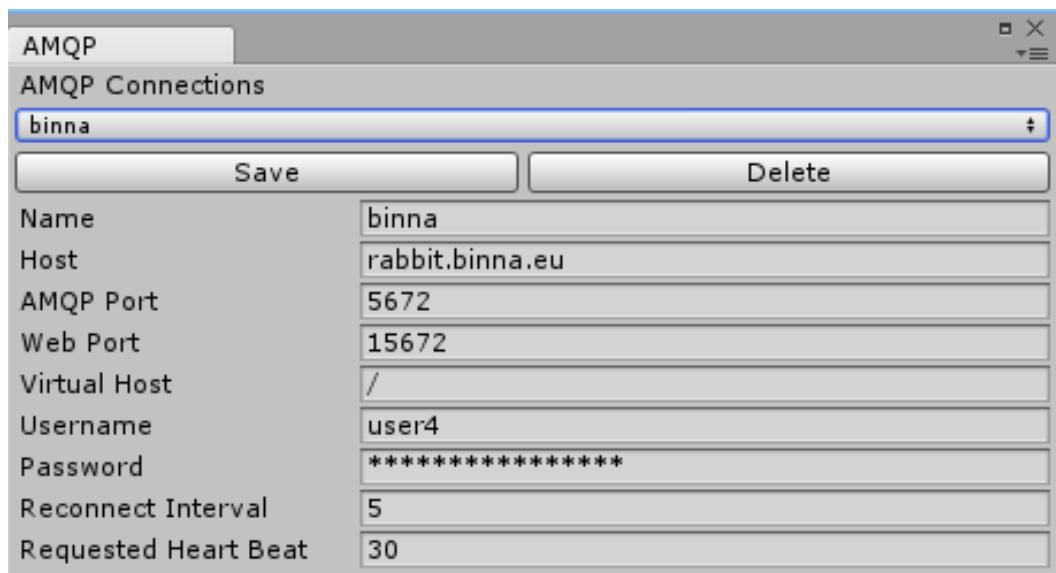


Abbildung 4.5: Einstellungen RabbitMQ

Jede Gruppe bekam einen eigenen Benutzer samt Passwort. Damit man Nachrichten empfangen kann, muss man sich auf eine ‘Queue’ subscriben.

Das Plugin stellt anschließend Methoden zur Verfügung. Eine solche Methode ist

OnMessageReceived(). Damit lässt sich eine eingehende Nachricht an ein Objekt in der Spielwelt weitergeben. Das Objekt kann daraufhin reagieren, beispielsweise ein Auto erzeugen.

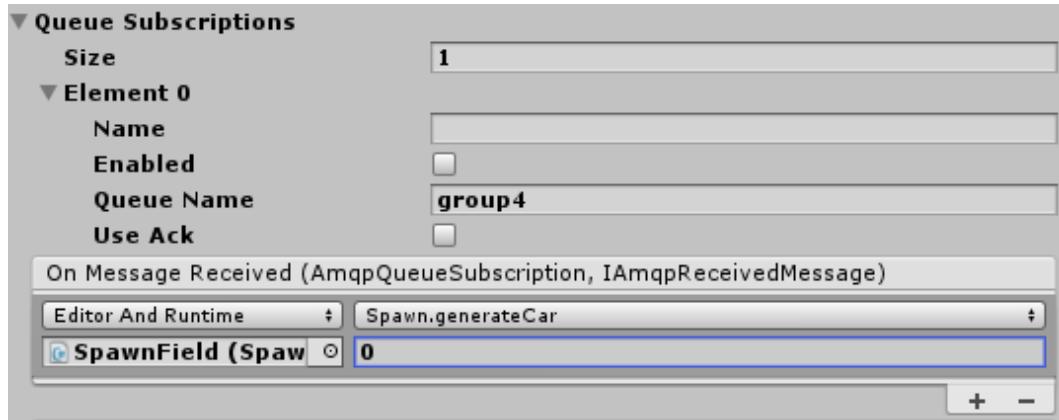


Abbildung 4.6: RabbitMQ Queue

In Abbildung 4.6 sieht man wie eine eingehende Nachricht beim Objekt SpawnField eine Methode aufruft.

5 Review einer fremden Architekturdokumentation

In diesem Kapitel wird die Architekturdokumentation der Gruppe Binna/Dorfer/Gruber/-Mühlbacher/Wieser überprüft. Dazu wurde die uns zur Verfügung gestellte Version 8 vom 27.06.2017 reviewt.

5.1 Review

Als Grundlage für die Architekturdokumentation wurde das arc42 Template verwendet. Dadurch ließ sich beim Review leicht feststellen, wie weit die einzelnen Punkte der Vorlage erfüllt wurden, oder wo es Abweichungen gab.

Der Abschnitt Einführung und Ziele enthält die Aufgabenstellung und alle geforderten Ziele. Dabei werden diese einfach und verständlich dokumentiert. Detailliertere Beschreibungen folgen erst in den nächsten Abschnitten.

Die tabellarische Form der Anforderungen/Qualitätsziele/Stakeholder ergibt eine übersichtliche Zusammenfassung. Allerdings sind manche Punkte verteilt an mehreren Stellen im Dokument beschrieben und es könnten daher Dinge übersehen werden. z.B. "Der User soll gewisse Parameter die Simulation betreffend verändern können.". Welche Parameter genau veränderbar sein sollen, findet man aber woanders.

Versionshistorie

Version	Datum	Autor(en)	Änderungen
0	03.04.2017	FS	Dokumentation erstellt (Vorlage: Martin Uray)
0.1	15.04.2017	MT	Komponentenbeschreibung
0.2	08.06.2017	AR	Erweiterung auf Arc42 Template
0.3	10.06.2017	LA	Hinzufügen 2.5 und 2.6
0.4	15.06.2017	FS	Erweiterung Ampelserver
0.5	18.06.2017	LA	Hinzufügen der Messaging Komponente
0.6	15.06.2017	AR	Fahrzeuglogik, Erkennen von Hindernissen
0.7	01.07.2017	PR	Review
0.8	12.07.2017	LA	Hinzufügen 4.2 und 4.1

Autoren:

Andreas Reschenhofer (AR), Fabian Schörghofer (FS), Lukas Altenhuber (LA), Mike Thomas (MT), Paul Riedl (PR)