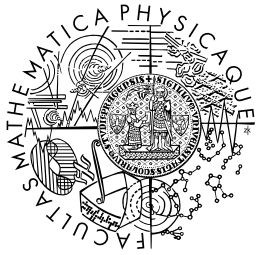


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Matěj Kripner

Traffic – hra se simulací silniční sítě

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D.
Studijní program: Informatika
Studijní obor: IPP1

Praha 2022

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Děkuji svému vedoucímu práce Mgr. Pavlu Ježkovi, Ph.D. za jeho ochotu, kontinuální podporu a rychlé řešení problémů, zejména s blížícím se termínem odevzdání. Dále děkuji své rodině za podporu v mých studiích i jinak.

Název práce: Traffic – hra se simulací silniční sítě

Autor: Matěj Kripner

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Cílem této práce byla implementace mobilní hry Traffic, ve které hráč ovládá auto a projíždí herním světem po silniční síti. Její základní princip vychází z naší vize hry simulující řízení, která by obohatila nabídku již existujících her. Tato vize se ukázala jako příliš komplexní, a proto jsme z jejích funkcionalit vybrali pouze podmnožinu, přičemž jsme dbali na zachování rozšiřitelnosti směrem k původní vizi.

Jako cílovou platformu jsme zvolili systém Android a jako implementační nástroj engine Unity spolu s jazykem C#. Ve výsledné hře je hráč v roli taxikáře, tj. převáží lidi na jimi určené destinace. Podle kvality jednotlivých jízd získává peníze a hodnocení. Při tom projíždí světem obsahujícím silnice, křižovatky, chodníky, budovy a chodce.

Herní svět je připraven na modifikaci herním návrhářem, který ne nutně umí programovat. Návrhář silnicím a chodníkům přiřazuje libovolný tvar daný Bézierovou křivkou a silnice pak spojuje do křižovatek. Tvar křižovatek se automaticky určí z tvaru navazujících silnic.

Klíčová slova: Unity, .NET, silniční síť, Bézierova křivka

Title: Traffic – Road Network Simulation Game

Author: Matěj Kripner

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: The goal of this thesis was to implement the mobile game Traffic, in which the player controls a car and navigates through a road network. The general concept of the game is based on our vision of a game simulating the experience of driving that would be different from similar existing games. This vision proved to be overly complex. Therefore, we selected only a subset of its features while ensuring that the resulting game can serve as a basis for future extension towards the original vision.

We targeted the Android operating system and used the Unity game engine and C# for implementation. In the resulting game, the player acts as a taxi driver, i.e. they pick up customers and drop them off at designated locations. The player receives game money and rating based on the quality of each ride. The game world contains roads, intersections, sidewalks, buildings and pedestrians.

The game world can be edited by a game designer with no programming skills. The designer can shape the roads and sidewalks using Bézier curves and connect roads into intersections. The shape of intersections is generated automatically.

Keywords: Unity, .NET, road network, Bézier curve

Obsah

1 Úvod **5**
- 1.1 Vize komplexní hry . . . 5
- 1.2 Podobné hry . . . 6
  - 1.2.1 #DRIVE . . . 6
  - 1.2.2 Roundabout 2: City Driving Sim . . . 6
  - 1.2.3 Car Driving School Simulator . . . 8
  - 1.2.4 Truck Simulator . . . 8
- 1.3 Cíle práce . . . 8

2 Analýza zadání **9**
- 2.1 Herní svět . . . 9
- 2.2 Chodci a herní mise . . . 9
- 2.3 Ovládání auta . . . 10
- 2.4 Audiovizuální stránka hry . . . 12
- 2.5 Uživatelské rozhraní . . . 12
- 2.6 Editor herního světa . . . 13
- 2.7 Podporované platformy . . . 13

3 Analýza **15**
- 3.1 Implementační nástroj . . . 15
- 3.2 Simulace silniční sítě . . . 16
  - 3.2.1 Spojování silnic, chodníků a domů . . . 19
  - 3.2.2 Serializace sítě . . . 19
  - 3.2.3 Renderování silnic a chodníků . . . 21
  - 3.2.4 Uniformně rozmístěné body na Bézierově křivce . . . 26
  - 3.2.5 Generování UV souřadnic pro silnice a chodníky . . . 28
  - 3.2.6 Decrossing . . . 29
  - 3.2.7 Renderování křižovatek . . . 32
- 3.3 Herní mechanismy . . . 36
  - 3.3.1 Ovládání chodců . . . 36
  - 3.3.2 Herní mise . . . 37
  - 3.3.3 Výpočet relativní polohy auta vůči silnici . . . 39
- 3.4 Rozdělení funkčnosti do modulů . . . 40
- 3.5 Rozšíření Unity editoru . . . 41

4 Herní engine Unity **43**
- 4.1 Základní popis . . . 43
  - 4.1.1 Komponenta `Transform` . . . 43
  - 4.1.2 Další důležité komponenty . . . 43
- 4.2 Programování . . . 44
  - 4.2.1 Prefabs . . . 45
  - 4.2.2 Serializace . . . 46
  - 4.2.3 Play Mode . . . 47
- 4.3 Rozšiřování Unity Editoru . . . 47

5 Vývojářská dokumentace **49**
5.1 Nástroje . . . 49
5.2 Struktura projektu . . . 49
5.2.1 Sestavení a spuštění projektu . . . 50
5.3 Struktura herní scény . . . 51
5.4 Implementace herního světa . . . 52
5.4.1 Modul `Holders` . . . 52
5.4.2 Modul `Splines` . . . 55
5.4.3 Modul `Network` . . . 57
5.4.4 Modul `Rendering` . . . 61
5.4.5 Modul `Audio` . . . 65
5.4.6 Modul `People` . . . 65
5.5 Implementace herních mechanik . . . 66
5.5.1 Fyzikální simulace auta . . . 66
5.5.2 Ovládání auta . . . 69
5.5.3 Taxi mise . . . 70
5.6 Perzistentní uložení stavu hry . . . 72
5.7 Úvodní menu . . . 72
5.8 Editor herního světa . . . 73
5.8.1 Posun, duplikace a mazání síťových prvků . . . 74
5.9 Užitečná rozšíření Unity . . . 76
5.9.1 Přidané ladící nástroje . . . 76

6 Uživatelská dokumentace – hráč **77**
6.1 Instalace hry . . . 77
6.2 Úvodní menu . . . 77
6.3 Princip hry . . . 78
6.4 Řízení auta . . . 79
6.5 Herní mise . . . 80

7 Uživatelská dokumentace – herní návrhář **85**
7.1 Otevření herního světa . . . 85
7.2 Orientace ve struktuře scény . . . 86
7.3 Přidání nové silnice nebo chodníku . . . 86
7.4 Změna parametrů objektů . . . 86
7.5 Práce s tvarem silnice nebo chodníku . . . 87
7.6 Propojení dvou nebo více silnic či chodníků . . . 88
7.7 Komponenta `Transform` chodníků a silnic . . . 91
7.8 Přidání chodníku podél silnice . . . 91
7.9 Reset navigačního meshe a uložených dat . . . 92
7.10 Řešení neplynulé práce s křižovatkami . . . 92
7.11 Vytvoření obytného domu . . . 92
7.12 Zotavení se z chyb v editoru . . . 93
7.13 Vrácení změny . . . 93

8 Závěr **95**
8.1 Budoucí rozšíření . . . 95

Seznam použité literatury **97**

Příloha A **103**

Příloha B **105**

1. Úvod

Pro některé lidi je řízení auta relaxující. Znamená ovšem vysoké náklady a potenciální nebezpečí, přičemž část lidí z různých důvodů nemůže skutečné auto řídit vůbec. Nabízí se tedy replikovat tento zážitek ve virtuálním světě.

Zatímco kvalitních PC her simulujících řízení auta je dostatek (např. *Truck Simulator*), v mobilním prostředí je situace jiná. Žádná z námi vyzkoušených mobilní her[1] nereplikuje podle našeho názoru odpočinkový aspekt řízení auta. Buď jsou tyto hry závodní (tj. hráč ve sportovním autě soutěží na závodním okruhu o co nejrychlejší čas), mají příliš restriktivní systém misí (jako např. *Roundabout 2: City Driving Sim* nebo *Car Driving School Simulator*) nebo se zaměřují jen na jeden aspekt řízení auta (řízení na izolované silnici, parkování, řešení předností na křižovatkách, apod. – jako např. *#DRIVE*). Zbývá tedy prostor pro komplexní hru cílící na mobilní zařízení, ve které je nějaký nezávodní cíl, nicméně hráč se může rozhodnout jej ignorovat a prozkoumávat otevřený svět.

Po zpřesnění naší vize takové hry dojdeme k tomu, že její implementace vysoce přesahuje rozsah bakalářské práce, a implementujeme místo toho její zjednodušenou verzi. Prvním krokem je ale popis naší původní nezjednodušené vize, ze které budeme vycházet.

1.1 Vize komplexní hry

Uvedeme, jak vypadá naše vize komplexní hry, která by mohla doplnit nabídku již existujících her simulujících řízení auta. V předchozí sekci jsme již rozhodli, že hra bude cílit na mobilní zařízení. Aby ji mohlo hrát co nejvíce hráčů, bude konkrétně cílit na nejrozšířenější operační systémy Android a iOS. Ty totiž dohromady běží na více než 99 % mobilních zařízení (konkrétně Android na 71,1 % mobilních zařízení a iOS na 27,57 % mobilních zařízení – stav k březnu 2022 podle údajů společnosti StatCounter [1]).

Hlavní myšlenkou hry je, že hráč v ní bude v roli řidiče a jeho auto bude umístěno do herního světa napodobujícího svět skutečný. Hra bude mít několik charakteristik, které uvedeme v abstraktní rovině.

1. Hra bude *otevřená*. Tím se v herní komunitě obvykle myslí dvě věci, jak uvádí herní databáze Codex Gamicus [2]. Zaprvé herní svět bude velmi rozsáhlý, případně nekonečný – toho by bylo možné docílit procedurálním generováním světa. Zadruhé hráč nebude nikterak penalizován, pokud nebude následovat nějakou herní misi. I když hra nějakou misi nabízet bude (to popíšeme dále), hráč bude mít možnost ji zcela ignorovat a „bezcílně“ projíždět světem.

2. Hra bude *komplexně* napodobovat objekty a mechanismy ze skutečného světa. Silniční síť bude sestávat z různých druhů silnic, křižovatek, kruhových objezdů, nájezdů, apod. Silnice a křižovatky budou opatřeny vertikálním i horizontálním dopravním značením a světelnou signalizací. Společně

[1] Z mobilních her jsme testovali pouze hry dostupné v Google Play, tedy pro systém Android. Vybírali jsme zejména podle uživatelských hodnocení.

s hráčem budou po silnicích jezdit auta ovládaná autonomně. Vedle některých silnic i jinde budou chodníky, po kterých budou chodit lidé. Dále bude svět obsahovat domy, parkoviště, stromy, lampy a další objekty.

3. Přes její komplexitu bude ale *jednoduché* hru hrát. To bude dáno zejména tím, že se herní svět bude podobat světu skutečnému, na jehož fungování jsou uživatelé zvyklí. To se týká např. způsobu řízení auta nebo pohybu chodců. Pokud bude v průběhu návrhu hry potřeba rozhodnout mezi dvěma variantami, z nichž jedna bude pro hráče jednodušší, pokusíme se přiklonit k té jednodušší. Cílem je, aby se nový uživatel nemusel nic učit a mohl ihned začít hrát. To znamená např. co nejvíce zjednodušit herní menu, které je pro hráče první bariérou před vstupem do hry. Příkladem může být herní menu ze hry *#DRIVE* na obrázku 1.1, ve kterém hráč ihned ví, které tlačítko stisknout pro vstup do hry. Naopak příkladem složitého herního menu je to ze hry *Truck Simulator : Ultimate* na obrázku 1.2, ve kterém musí hráč před samotnou jízdou vybrat z několika nabídek a vzít při tom do úvahy peníze, typ nákladu a délku trasy.

4. Nakonec bude hra *dlouhodobě zábavná*. Domníváme se, že i zmíněná bezcílná jízda herním světem bude v popsané hře zábavná díky komplexitě hry – hráč bude neustále interagovat s ostatními auty, s chodci nebo se světelnou signalizací na křižovatkách. Přesto by absence herní mise dlouhodobou zábavnost pravděpodobně limitovala. Hra bude proto nabízet hned několik misí, ze kterých si hráč bude moct vybrat. Jednou z nich bude mise převážení lidí, kdy cílem bude nabírat a vysazovat lidi, jako by hráč byl taxikář. Domníváme se, že tato mise bude zábavná zejména díky interakci s chodci, kteří budou hráče hodnotit za kvalitu jízdy.

1.2 Podobné hry

Pro zasazení hry popsané v předchozí sekci 1.1 do kontextu již existujících her uveďme několik příkladů her, které s ní sdílejí určité aspekty. S výjimkou *Truck Simulator* cílí všechny uvedené hry na mobilní zařízení a jsou dostupné v Google Play. U každé hry uvedeme pouze ty pozitivní aspekty, kterými se chceme inspirovat a pouze ty negativní aspekty, kterým se chceme vyhnout.

1.2.1 #DRIVE

#DRIVE je typickým příkladem toho, co považujeme za povedenou oddechovou hru, a to díky kombinaci jednoduché grafiky a příjemné hudby. Hra ovšem obsahuje velmi omezený herní svět, který sestává z jediné silnice, přičemž hráč po ní může jet pouze vpřed. Ve hře nejsou křižovatky ani chodci. Kvůli tomu se hra při delším hraní stává příliš repetitivní a není dlouhodobě zábavná.

1.2.2 Roundabout 2: City Driving Sim

Roundabout 2: City Driving Sim je otevřená hra, ve které může hráč volně prozkoumávat středně rozsáhlý svět obsahující silnice, křižovatky, chodníky, budovy, apod. Závažným problémem této hry je její herní mise, kdy jediným cílem

Obrázek 1.1: Screenshot ze hry *#DRIVE* zachycující její jednoduché herní menu.

Obrázek 1.2: Screenshot ze hry *Truck Simulator: Ultimate* zachycující její složitější herní menu.

je následovat trasu vytyčenou periodicky rozmístěnými značkami. Taková mise je příliš jednoduchá a monotónní a většinu hráčů nebude dlouhodobě bavit. Navíc pokud se hráčovo auto dotkne překážky (nezávisle na rychlosti kolize), mise se ukončí a hráč musí začít od začátku. To je po několika kolizích velmi frustrující a neodpovídá fungování aut ve skutečném světě. Dále je nevhodně řešené ovládání plynu a brzdy auta – příslušná tlačítka jsou příliš malá a při hraní je hráč snadno netrefí, protože se soustředí na silnici před sebou.

1.2.3 Car Driving School Simulator

Car Driving School Simulator je hra sloužící jako pomůcka při učení řízení auta, čímž se zásadně odlišuje od námi popsané hry. Hráč je totiž nucen absolvovat krátné výukové mise a nemá možnost samostatně jezdit po herním světě. Tyto mise navíc vyžadují od hráče řešení nezábavných problémů, jako např. startování motoru, zapínání bezpečnostních pásů nebo používání blinkrů. Hra nicméně stojí za zmínku díky velmi propracovanému hernímu světu, který obsahuje, mimo jiné, chodce, dopravní značky a horizontální dopravní značení.

1.2.4 Truck Simulator

Truck Simulator je série PC her od společnosti *SCS Software s.r.o.* Jedná se o hry nejvěrněji napodobující řízení kamionu v reálném světě. Obsahuje realistickou simulaci ovládání kamionu, simulaci ostatních aut na silnici a velmi rozsáhlý a kvalitně provedený herní svět se střídajícím se počasím. Ovládání kamionu a herní mise jsou ale velmi náročné pro začínajícího hráče. Hráč musí dbát na doplňování paliva, opravy kamionu, správu své dopravní firmy, apod.

Existuje několik klonů této hry pro mobilní zařízení. Jeden z povedených je *Truck Simulator: Ultimate*, který se svou předlohou sdílí jak skvěle propracovanou simulaci reálného světa, tak i vysokou náročnost pro začínajícího hráče a velmi komplikované uživatelské rozhraní. Proto se domníváme, že má smysl vytvořit hru, která bude jednodušší.

1.3 Cíle práce

Vytvoření hry popsané v sekci 1.1 je dlouhodobý úkol pro vícečlenný tým programátorů, herních návrhářů a grafiků. V této práci proto vybereme vhodnou podmnožinu funkcionalit této hry tak, aby výsledná hra byla zábavná a zároveň aby ji bylo možné implementovat v rámci bakalářské práce. Přitom budeme dbát na to, aby výsledné dílo bylo navrženo rozšiřitelně a mohlo v budoucnu posloužit jako základ pro vytvoření původní komplexní hry. Zejména se chceme zaměřit na simulaci silniční sítě, což považujeme za zajímavé téma. Než se pustíme do analýzy a návrhu hry, je nutné vybrat zmíněnou množinu funkcionalit, kterou bude hra obsahovat. To bude předmětem následující kapitoly.

2. Analýza zadání

Cílem této kapitoly je vybrat ze hry uvedené v sekci 1.1 vhodnou podmnožinu funkcionalit tak, aby se její implementace vešla do rozsahu bakalářské práce. Přitom chceme co nejvíce zachovat hratelnost a zábavnost hry a dbát na její budoucí rozšiřitelnost směrem k původní vizi. Výslednou zjednodušenou hru a editor jejího herního světa popíšeme v této kapitole pouze z uživatelského hlediska. Samotná implementace pak bude předmětem zbylých kapitol.

Především je potřeba zachovat základní koncept hry. Hráč bude tedy ovládat auto a projíždět silniční sítí. Pokud bude chtít, bude mít možnost provádět herní misi. Kamera bude následovat hráče a zabírat jeho auto zezadu, bude se tedy jednat o „third-person“ pohled. Celá hra se bude odehrávat ve třídimenzionálním světě.

2.1 Herní svět

Nejprve se zaměřme na podobu herního světa. Ten bude obsahovat silniční sít, síť chodníků a obytné budovy. Dekorativní objekty zmíněné v sekci 1.1, jako např. stromy nebo lampy, bude v budoucnu jednoduché doplnit, protože kromě kolizí s hráčovým autem nebudou hru nijak ovlivňovat.

Silniční síť bude sestávat ze silnic a křižovatek spojujících silnice. Každou silnici bude možné vytvarovat tak, aby její tvar přibližně modeloval tvar libovolné skutečné silnice a bude obsahovat jeden jízdní pruh pro každý z obou směrů.

Síť chodníků bude velmi podobná síti silnic, ale navíc bude napojena na obytné domy. Chodníky budou umístěny buď samostatně, nebo podél silnice. To je v souladu s původní vizí ze sekce 1.1.

Podstatnou vlastností hry popsané v sekci 1.1 je simulace aut jezdících společně s autem hráče po silniční síti. Podpora této vlastnosti je podmíněna vytvořením systému autonomního řízení auta, který bude schopen reagovat na chování ostatních aut včetně auta hráče a přizpůsobovat tomu svou jízdu. Hráč přitom bude mít možnost ovládat své auto libovolně, takže budou potenciálně vznikat složité dopravní situace. Proto se domníváme, že problém realisticky vypadající simulace autonomně se pohybujících aut je co do rozsahu téma pro další bakalářskou práci. Z toho důvodu tuto vlastnost do hry prozatím nezahrneme. Budeme nicméně dbát na to, aby komponenta zajišťující ovládání auta měla co nejjednodušší rozhraní a byla tak připravena na budoucí využití systémem autonomního řízení aut.

Protože hráč bude jediným účastníkem silničního provozu, bylo by dopravní značení a světelná signalizace de facto pouze dekorativním prvkem a tedy ve stejné kategorii jako zmíněné stromy nebo lampy. Proto dopravní značení ani světelná signalizace nebudou ve hře zahrnuty.

2.2 Chodci a herní mise

Chodci budou dekorativním doplňkem herního světa a zároveň předmětem herní mise. Budou se pohybovat výhradně po chodnících a při svém pohybu budou

sledovat nějaký konkrétní cíl, totiž některou z budov. Dále budou respektovat překážky i sebe navzájem a pokud to půjde, vyhnou se.

Z herních misí vybereme jen jednu, a to tu zmíněnou v minulé kapitole, tedy misi taxikáře. Tu nyní popišme podrobněji: hráči bude po náhodné prodlevě nabídnuto přijmutí zakázky, tedy zákazníka vyžadujícího přepravu. Součástí nabídky bude jméno zákazníka, telefonní číslo a jeho aktuální vzdálenost od auta. Pokud hráč nabídku odmítne, celý proces se po další náhodné prodlevě zopakuje. V případě přijmutí nabídky bude prvním cílem hráče vyzvednout zákazníka čekajícího na daném místě. Následně bude druhým cílem převézt naloženého zákazníka co nejrychleji a nejpohodlněji na jím určené místo.

Zákazník čekající na vyzvednutí a následně jeho cílová destinace budou barevně označeny. Kromě toho bude automaticky vypočtena nejkratší trasa po silnicích od aktuální pozice hráčova auta k požadované pozici a tato bude vodorovně vyznačena na silnici před autem. Hráč tedy nebude muset navigaci v herním světě nijak řešit – bude mu stačit následovat poskytnuté značení.

Za dokončené zakázky bude hráč sbírat peníze a hvězdičkové hodnocení na škále 1–5. Zákazníci budou hodnotit rychlost jízdy a její pohodlnost a délku zvolené trasy. Budou mít také možnost dát spropitné.

Získané peníze bude hráč ztrácet placením pokut, kterou dostane, pokud vyjede ze silnice nebo narazí do chodce. To do hry přidá zajímavý prvek rozhodování, zda upřednostnit rychlost vyřízení zakázky a zkrátit si cestu vyjetím ze silnice za cenu zaplacení pokuty. Hráč se tak bude moct vžít buď do role poctivého řidiče, nebo do role piráta silnic.

V budoucnu bude možné uvažovat o rozšíření této mise např. tak, že hráč bude mít možnost vybrat si z většího množství najednou nabízených zákazníků. Naše zjednodušení ale podle našeho názoru zásadně neubírá na hratelnosti – pokud hráči nabízený zákazník nebude vyhovovat, bude jej moct odmítnout a počkat na dalšího.

2.3 Ovládání auta

Samotné ovládání auta může být řešeno mnoha způsoby. Začněme způsobem řízení směru, tedy způsobem, jakým regulovat natočení předních kol auta. V námi vyzkoušených hrách jsou rozšířené následující přístupy:

1. Řízení dvěma tlačítky – stiskem jednoho se zatáčí doleva, stiskem druhého doprava. Tlačítka mohou buď mít vzhled šipek nebo mohou být průhledná a pokrývající celou levou, resp. pravou polovinu obrazovky.

2. Řízení nakláněním telefonu nalevo nebo napravo.

3. Řízení otáčením volantu pomocí tažení prstem.

První z těchto přístupů je pro naši hru nevyhovující, protože je v něm obtížné precizně regulovat míru zatáčení, které je pro pohyb po silniční síti důležité.

Druhý přístup nevyhovuje v tom, že při naklánění telefonu je obtížnější soustředit se na navigaci v herním světě, protože se mění relativní poloha telefonu a očí hráče. V nejkritičtějších momentech jízdy, kdy je míra zatáčení nejvyšší, je tento efekt nejrušivější. Pro zmírnění tohoto efektu by hráč spolu s telefonem

musel naklánět i hlavu, což je ale při delším hraní nepříjemné. Dále při porovnání druhého a třetího přístupu dovoluje otáčení volantem preciznější kontrolu než naklánění telefonu, protože maximální úhel natočení telefonu je mnohem menší než maximální úhel natočení volantu – zatímco volant lze otočit i o více než 360°, telefon by bylo nepříjemné otáčet o více než pravý úhel. Z těchto důvodů vybereme pro naši hru přístup s otáčením volantu, u kterého jsme neidentifikovali žádnou nevýhodu.

Dále musí mít hráč možnost určit směr jízdy auta (dopředu nebo dozadu). To je v podobných hrách řešeno jedním z následujících 2 způsobů:

1. Pedál pro přidávání plynu může být rozdělen na 2 části – jednu pro urychlení auta dopředu a druhou urychlení auta dozadu. Brždění je pak dosaženo přidáním toho plynu, který působí proti aktuálnímu směru jízdy. Zvláštní brzdový pedál v takovém případě není potřeba.

2. Hráči může být umožněno zvlášť přidávat plyn, brzdit a přepínat mezi dvěma směry jízdy (dopředu a dozadu).

Žádný z těchto způsobů nemá zásadní nevýhodu, nicméně u prvního z nich je těžší dosáhnout úplného zastavení auta, protože při zpomalování musí hráč včas uvolnit stisknutý plyn – jinak se začne pohybovat na druhou stranu. To by bylo nepříjemné při nakládání a vykládání zákazníků, kdy se hráč bude chtít zcela zastavit. Přestože má druhý způsob více ovládacích prvků, kopíruje lépe řízení skutečného auta, a proto jej nepovažujeme za složitější na pochopení. Z těchto důvodů použijeme v naší hře druhý způsob ovládání.

Hráč tak bude mít možnost zvlášť auto urychlovat a zastavovat. Z pohledu fyzikální simulace bude tedy mít možnost regulovat točivý moment působící na kola auta z motoru a z brzd. Tyto dva efekty rozebereme zvlášť:

- V případě motoru je točivý moment ve skutečných autech regulován výkonem motoru (přidáváním nebo ubíráním plynu) a zařazeným rychlostním stupněm (v případě manuální převodovky pomocí řadící páky). V naší hře musí mít hráč určitě možnost regulovat výkon motoru, jinak by nemohl uvést auto do pohybu.

  Co se týče řadící páky, jedná se o poměrně komplikovaný ovládací prvek a hráči nějakou dobu trvá, než dokáže řadit bez toho, aby ztratil přehled o dění před sebou. I zkušený hráč pak musí na řazení použít jeden ze svých prstů a dočasně se tak musí vzdát kontroly nad jiným ovládacím prvkem, jako např. volantem, plynem nebo brzdou. Přitom v námi vyzkoušených hrách, které řadící páku neobsahují, to nijak na herním zážitku neubírá (to je např. případ her *#DRIVE* nebo *Roundabout 2: City Driving Sim*). Z těchto důvodů řadící páku do naší hry nezahrneme.

  Pokud hráč nemusí volit zařazený rychlostní stupeň, nevidíme přidanou hodnotu v simulaci automatické převodovky. To opět vychází z našeho porovnání her simulujících převodovku (jako např. *Truck Simulator*) a her, které převodovku nesimulují (jako např. *#DRIVE*). Automatickou převodovku proto simulovat nebudeme.

- Brzdy budou ovládány tak, že hráč určí míru sešlápnutí brzdového pedálu (detaily o vzhledu pedálu uvedeme dále), což odpovídá řízení skutečného auta.

Hráč bude tedy pouze určovat míru sešlápnutí plynu a sešlápnutí brzdy, což je podobné řízení skutečného auta s automatickou převodovkou. To je ve většině námi vyzkoušených her řešeno dvěma tlačítky reprezentujícími pedály – jeden pro plyn a jeden pro brzdu. Tento přístup pro naši hru považujeme za nevhodný, protože není možné určit míru sešlápnutí daného pedálu. Přitom důležitou součástí jízdy po silniční síti je právě precizní ovládání plynu a brzd. Proto pro tento účel navrhneme vlastní ovládací prvek.

Vyjdeme z omezení, že nebude možné v jeden okamžik zároveň přidávat plyn a zároveň brzdit. To neubírá na hratelnosti v porovnání s dvěma oddělenými pedály pro plyn a brzdu, protože hráč má pro ovládání obou pedálu tak jako tak vyhrazený pouze jeden prst. Díky tomu můžeme spojit plyn a brzdu do jednoho ovládacího prvku, který tak bude větší a tím pádem snažší na ovládání. Bude tvořen svislým pruhem, na kterém hráč tažením prstu vymezí hodnotu. Pokud bude tato hodnota nad určitou mezí, bude její vzdálenost od této meze vyjadřovat míru sešlápnutí plynu. Pokud bude pod touto mezí, bude její vzdálenost vyjadřovat míru sešlápnutí brzdy.

2.4 Audiovizuální stránka hry

Co se týče audiovizuální stránky hry, tedy zvuků, hudby, textur a 3D meshů, nepatří tvorba těchto prvků do expertizy autora práce. Proto tyto prvky budou jednoduché co do uměleckého provedení. Budou nicméně přítomny a v budoucích verzích hry tak bude jednoduché je nahradit jejich kvalitnějšími verzemi.

Konkrétně bude ve hře hudba, zvuk motoru auta a zvukové efekty jako např. při naložení nebo vyložení zákazníka. Textury budou zpracovány velmi jednoduše a jen do té míry, do které to pomůže hratelnosti hry. To se týká textur pro silnice, chodníky a zemi, které jsou důležité, aby hráč snáze poznal, jakou rychlostí se pohybuje. Pokud ale nějakou texturu použijeme, budeme dbát na to, aby byla na podkladový objekt správně nanesena.

Co se týče 3D meshů, budeme se soustředit na kvalitní simulaci silnic, chodníků a křižovatek. Naopak zcela zanedbáme meshe pro budovy a pro auto, kde v počáteční verzi hry použijeme kvádry.

2.5 Uživatelské rozhraní

Kromě samotné hry bude mít aplikace úvodní menu. Z něj bude možné provést 3 různé akce:

1. Spustit hru. Tím se uživatel přesune do herního světa a zobrazí se mu herní ovládací prvky. Stav hry (zejména pozice auta a hráčovo skóre) bude uložen napříč spuštěními aplikace, dokud jej hráč explicitně nevyresetuje.

2. Vyresetovat stav hry. Tím se vynuluje hráčovo skóre a hráčovo auto se přesune do výchozí polohy.

3. Ukončit aplikaci.

Ze hry pak bude možné přejít zpět do úvodního menu.

2.6 Editor herního světa

Velmi důležitou součástí práce bude editor herního světa. V něm bude mít herní návrhář možnost jednoduše a bez znalosti programování vytvořit libovolnou silniční a chodníkovou síť a umisťovat budovy. Takto vytvořený svět bude pak součástí sestavení hry a hráč nebude mít možnost jej dále měnit.

Důležitou vlastností editoru bude to, že bude automaticky generovat tvary křižovatek na základě pozic navazujících silnic nebo chodníků. Detaily o této funkcionalitě uvedeme v sekci 7.

2.7 Podporované platformy

Při snaze o podporu systému iOS jsme narazili na problém, že vývojářské nástroje potřebné pro vývoj pro iOS (tj. iOS SDK) jsou dostupné pouze pro počítače od společnosti Apple – viz vývojářská sekce oficiálního webu společnosti Apple [3]. Autor přitom takový počítač nevlastní. Proto omezme cílovou platformu na systém Android. Při vývoji se ovšem budeme snažit o to, aby v budoucnu bylo co nejjednodušší zařadit iOS mezi podporované platformy.

3. Analýza

V předchozí kapitole jsme uvedli popis hry z pohledu herního návrháře. Cílem této kapitoly je provést vysokoúrovňová technická rozhodnutí o způsobu implementace.

3.1 Implementační nástroj

Protože už máme představu o podobě hry, kterou budeme implementovat, můžeme vybrat implementační nástroj. Zásadním požadavkem na tento nástroj je schopnost překládat stejný kód jak pro systém Android, tak pro systém iOS. Ačkoli systém iOS zatím podporovat nebudeme (jak jsme rozhodli v kapitole 2) chceme mít možnost tuto podporu v budoucnu přidat (to jsme uvedli v sekci 1.1). Zároveň chceme vybírat pouze z herních enginů, abychom se mohli soustředit na implementaci samotných herních mechanik a vyhnuli se řešení opakujících se problémů při tvorbě hry (renderovací pipeline, herní smyčka, systém herních objektů, systém událostí, animace, serializace a ukládání objektů a herního stavu, apod.). Zaměřme se na 4 takové enginy, které jsou uvedeny v oficiální vývojářské příručce společnosti Google [4]:

1. Defold
2. Godot
3. Unity
4. Unreal Engine 4

Z úvahy jsme vyřadili engine Defold, protože jednak je optimalizovaný spíše pro 2D hry (viz opět vývojářská příručka společnosti Google [4]) a jednak vývoj v něm probíhá v jazyce Lua (viz manuál enginu Defold [5]), který autor neovládá. Ze zbylých tří enginů jsme vyzkoušeli pouze Godot a Unity. Unreal Engine je tradičně považován za nástroj určený spíše pro větší herní společnosti, jak uvádí např. herní vývojář Ben Tristem ve svém článku Unity vs Unreal [6]. Jeho vhodnost pro náš projekt jsme proto z časových důvodů nevyhodnotili.

Herní enginy Godot a Unity jsou si velice podobné a oba podporují programování v moderním a rozšířeném jazyce C#. Zároveň se domníváme, že v obou z nich by bylo možné námi navrženou hru implementovat. Nicméně při práci s enginem Godot jsme naráželi na relativně omezenou nabídku funkcí a nepropracovanou dokumentaci. Unity naopak nabízí velmi velké množství funkcí a propracovanou dokumentaci – jak dokumentaci oficiální, tak nespočet tutoriálů dostupných online. Kvůli velmi aktivní komunitě okolo enginu Godot autor nicméně očekává, že v budoucnu se tento engine co do použitelnosti vyrovná enginu Unity.

Co se funkcionality týče, máme konkrétně např. pochybnosti o vyspělosti navigačního systému enginu Godot, který budeme nutně potřebovat při simulaci pohybu chodců. Dokumentace k aktuální verzi Godot 3.4.4 [7] totiž uvádí: *„The current navigation system has many known issues and will not always return optimal paths as expected. These issues will be fixed in Godot 4.0.“* Naopak navigační systém enginu Unity [8] podle našich experimentů funguje dobře.

Velmi podobné je porovnání nástrojů pro fyzikální simulaci vozidel s koly v enginech Godot a Unity. Godot nabízí pro tento účel třídu `VehicleWheel` [9], Unity pak třídu `WheelCollider` [10]. Dokumentace ke třídě `VehicleWheel` v aktuální verzi Godot 3.4.4 uvádí: *„This class has known issues and isn't designed to provide realistic 3D vehicle physics. If you want advanced vehicle physics, you will probably have to write your own physics integration using another PhysicsBody class.“* Tuto třídu z enginu Godot bychom tedy pro simulaci vozidla použít nemohli. Naopak podle našich experimentů funguje třída `WheelCollider` pro fyzikální simulaci vozidla dobře.

Z těchto důvodů jsme pro náš projekt zvolili engine Unity, konkrétně jeho verzi 2021.2.7f1, což byla nejnovější verze v době zahájení implementace. Pokud se čtenář dosud s enginem Unity nesetkal, doporučujeme seznámit se se stručným shrnutím jeho funkcí v kapitole 4.

3.2 Simulace silniční sítě

Ústředním prvkem herního světa bude síť silnic, po které se bude pohybovat hráčovo auto, a síť chodníků, po které se budou pohybovat chodci. Zaměřme se nejprve na síť silnic – ta sestává z jednotlivých silnic a křižovatek. Začněme rozhodnutím, jak reprezentovat tvar silnic.

Můžeme se pokusit čerpat inspiraci z postupů, které se osvědčily při projektování reálných silnic. Při projektování je třeba dbát na mnoho faktorů, jako např. bezpečnost a efektivitu jízdy, kapacitu silnice, maximální povolenou rychlost, cenu, environmentální dopad, hluk způsobený v obytných oblastech, apod., jak uvádí např. Institut dopravních inženýrů [11]. Naopak v naší hře je jediným relevantním faktorem herní zážitek. Ten se však částečně odvíjí od toho, do jaké míry působí silnice realisticky, a proto se nad projektováním vyplatí zamyslet.

Projektování silnice se konkrétně zaměřuje na tři oblasti, jak uvádí článek Geometric Desing of Roads na Wikipedii [12]:

1. Horizontální tvar silnice (*Alignment*) – tvar silnice při pohledu shora.
2. Vertikální tvar silnice (*Profile*) – stoupání a klesání silnice.
3. Parametry průřezu silnice (*Cross Section*) – počet jízdních pruhů, umístění chodníků, náklon silnice, apod.

Pro zjednodušení bude celý svět včetně silniční sítě umístěn v rovině, což je v souhladu s popisem hry ze sekce 1.1. Tím pádem nemusíme brát v potaz vertikální tvar silnice. V sekci 2.1 jsme pak určili, že každá silnice bude mít právě jeden jízdní pruh v každém směru. Volitelně bude silnice mít chodník na jedné nebo na obou stranách. Žádné další prvky, jako např. pruhy pro cyklisty, horizontální dopravní značení, přechody pro chodce nebo příkopy vedle silnice naše hra obsahovat nebude.

Zbývá rozhodnout o reprezentaci horizontálního tvaru silnice. Ten se při projektování reálných dálnic skládá z následujících tří typů úseků, viz článek Alignment and Superelevation Ministerstva dopravy státu Wyoming [13]:

1. Rovné úseky (*Straight-Line Tangents*), tj. úsečky.

2. Obloukové úseky (*Circular Curves*), tj. části kružnice.

3. Přechodové úseky (*Spiral Curves*), tj. zatáčky, ve kterých se plynule zvyšuje nebo snižuje poloměr otáčení. Jedná se tedy o přechody mezi rovnými a obloukovými úseky.

Domníváme se, že pomocí uvedených tří typů úseků lze dostatečně dobře popsat libovolný tvar silnice, i pokud tato silnice nebyla původně pomocí těchto úseků navržena. Zároveň ale takový postup návrhu silnic považujeme za relativně komplikovaný pro herního návrháře, který se s projektováním silnic dosud nesetkal. Protože nemusíme dbát na fyzikální a konstrukční omezení, pokusíme se navrhnout postup, který je typickému hernímu návrháři bližší a zároveň pomocí něj lze dostatečně dobře popsat libovolný tvar silnice.

Libovolný „rozumný“ tvar lze popsat kubickou Bézierovou křivkou – proto na takové křivky spoléhají programy pro vektorovou grafiku (např. Inkscape nebo Adobe Illustrator), standardy pro vektorovou grafiku (např. SVG) nebo popisy fontů (např. pomocí PostScript). Kubickou Bézierovou křivkou lze tedy dostatečně dobře aproximovat i libovolný tvar silnice. Otázkou je, jak pracné to je v porovnání s konstrukcí takového tvaru pomocí rovných, obloukových a přechodových úseků. Na obrázku 3.1 jsme přibližně proložili silnice na Malé Straně kubickými Bézierovými křivkami. Tím demonstrujeme, že dostatečně dobré aproximace tvaru těchto silnic lze dosáhnout pomocí malého počtu Bézierových křivek. Díky tomu celé kreslení těchto křivek zabralo autorovi pouze přibližně 5 minut. Ačkoli nejsou silnice na obrázku proloženy křivkami zcela přesně, byla by výsledná silniční síť co do herního zážitku zcela dostačující.

Navíc ovládání Bézierových křivek je poměrně intuitivní a typický herní návrhář se s nimi již setkal. Často se totiž používají pro návrh tvaru cest, řek, tras entit ovládaných hrou, apod., viz článek o Bézierových křivkách na vývojářském webu `gamedeveloper.com` [14]. Dále je práce s nimi běžná v programech pro vektorovou grafiku, jak bylo zmíněno výše.

Na obrázku 7.7 v uživatelské příručce uvidíme, že herní editor bude opravdu schopen vytvořit dobrou aproximaci této části silniční sítě.

Bézierovy křivky mají kromě velké rozšířenosti, snadného používání a flexibilitě i další dobré vlastnosti. Ty podrobněji rozebereme v sekci 5.4.2, nyní uveďme pouze jejich výčet:

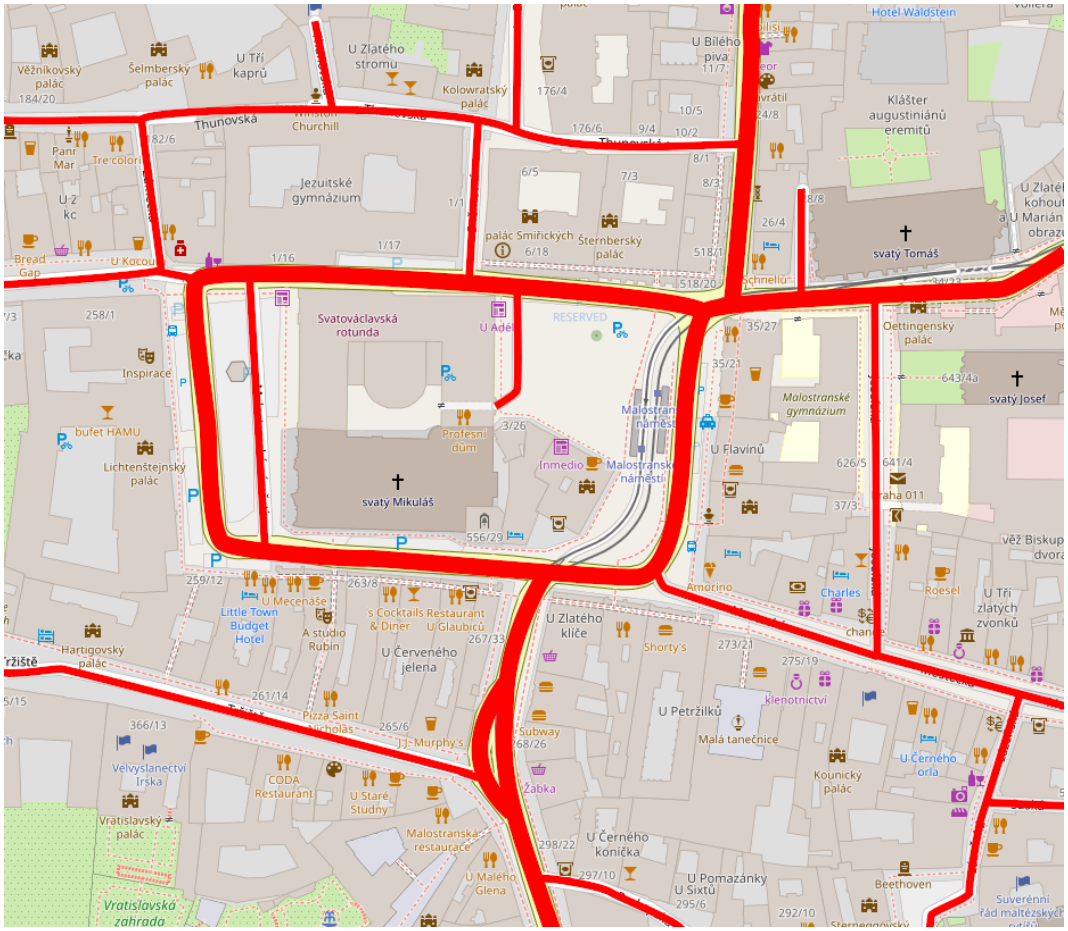
1. Interpolace bodů na Bézierově křivce je efektivní a jednoduchá díky De Casteljauovu algoritmu.

2. Lze efektivně a přesně spočítat derivaci Bézierovy křivky v každém jejím bodě a dostat tak v každém jejím bodě tečný (resp. normálový) směr.

3. Ačkoli jejich délku nelze vyjádřit uzavřeným vzorcem, existují jednoduché odhady, které pro naše účely dostačují.

Z těchto důvodů zvolíme kubické Bézierovy křivky pro popis tvaru silnic. Díky jejich flexibilitě můžeme navíc zcela stejný přístup zvolit i pro popis tvaru chodníků.

Ve zbytku práce zaveďme v kontextu kubických Bézierových křivek základní terminologii. Tvar kubické Bézierovy křivky je určen čtyřmi body, které nazveme



Obrázek 3.1: Mapa části Malé Strany se silnicemi proloženými Bézierovými křivkami.
Mapový podklad byl převzat z projektu OpenStreetMap, © OpenStreetMap contributors, jehož licenční podmínky jsou uvedeny na adrese `https://www.openstreetmap.org/copyright`. Bézierovy křivky pak byly dokresleny v programu GIMP.

kontrolní body. Tyto body mají určené pořadí a v tomto pořadí je pojmenujeme takto: *počáteční opěrný bod*, *počáteční směrový bod*, *koncový směrový bod* a *koncový opěrný bod*. Opěrným bodům budeme říkat také *kotvy*. České názvosloví nicméně není ustálené.

Dodejme, že druhým zváženým postupem byl popis tvaru silnic lomenou čarou. Abychom ale lomenou čarou dostatečně věrně popsali plynulou zatáčku, musela by se čára skládat z velmi velkého množství vrcholů. To by bylo nepraktické pro její ukládání – pokud chceme mít v budoucnu možnost vytvářet velmi rozsáhlé herní světy, znamenala by taková neefektivita problém. Proto jsme tento přístup zavrhli. Třetím možným postupem je zkombinovat předchozí dva a povolit část tvaru silnice nebo chodníku popsat lomenou čarou a část posloupností kubických Bézierových křivek. Tento postup by ovšem zásadně zkomplikoval implementaci a není jasné, zda by hernímu návrháři pomohl.

3.2.1 Spojování silnic, chodníků a domů

Některé objekty v naší hře mohou být napojené na jiné objekty. Např. dvě nebo více silnic může být spojeno do křižovatky, stejně tak chodník může být napojený na dveře budovy. Tento koncept popíšeme obecně, což pomůže zredukovat duplikaci při implementaci. Zaveďme tedy pojmy *síťový prvek* pro objekt, který může být napojený na jiné objekty a *síť* pro takto vzniklou síť objektů. Síťové objekty jsou např. silnice, chodníky nebo domy. V budoucnu to ale mohou být např. parkoviště nebo kruhové objezdy. Každý síťový prvek bude mít libovolné množství tzv. *kotev* (*anchor*), což jsou pozice, za které se prvek bude k ostatním prvkům připojovat. Záměrně zde volíme pojem, který jsme již zavedli pro opěrné body Bézierových křivek. Dále v této sekci totiž dojdeme k tomu, že bude výhodné kotvy Bézierových křivek použít jako kotvy z hlediska síťových prvků a zvolené názvosloví pak značně usnadní popis. Kotvou bude tedy např. konec silnice náležící do křižovatky nebo dveře obytné budovy.

Jak bude proces spojování probíhat z uživatelského hlediska uvedeme v uživatelské příručce v sekci 7.6. Pro jednoduchost povolíme spojit libovolné dvě různé kotvy, i když některá spojení nedávají smysl (např. napojení silnice na dveře obytné budovy). Pokud by se v budoucnu ukázalo, že taková svoboda je pro herního návrháře přílišná, lze některé kombinace jednoduše zakázat přidáním explicitní podmínky v herním editoru. Naopak chceme zachovat možnost spojovat dvě různé kotvy stejného síťového prvku – to umožní např. spojení obou konců silnice a vytvoření silničního okruhu.

Použitím kotvy ke spojení se její funkcionalita nezablokuje, takže v jednom bodě je možné spojit libovolné množství kotev. Spojení některých typů prvků má pak speciální význam. Např. spojení více silnic (resp. chodníků) v jednom bodě znamená křižovatku, čemuž je třeba uzpůsobit renderování. To popíšeme v následující sekci 3.2.3.

Dále uveďme, jak zvolíme umístění kotvy nějakého síťového prvku. Pokud se jedná např. o budovu, chceme, aby byla kotva umístěna tam, kde má daná budova dveře. Pak totiž k této kotvě můžeme napojit přístupový chodník. Uživatele v takovém případě necháme pozici kotvy zvolit ručně v editoru světa (viz opět uživatelská příručka 7), protože automaticky polohu dveří budovy určit nelze. Ovšem volit pozici kotev pro každou silnici a chodník by bylo pro herního návrháře příliš namáhavé. Lepší přístup je tyto pozice určit automaticky a případně dát uživateli možnost kotvu přidat, pokud by mu na nějaké pozici chyběla. Pro zjednodušení implementace toto provedeme trikem – kotvy silnice (resp. chodníku) budou právě kotvy odpovídající Bézierovy křivky. To splňuje naše požadavky, protože editor světa bude podporovat podrozdělení křivky na dvě a tedy přidání kotvy. Kromě jednoduchosti je výhodou tohoto přístupu to, že silnici (resp. chodník) většinou napojujeme na ostatní silnice (resp. chodníky) na jejím konci, přičemž na obou koncích Bézierovy křivky vždy kotva je.

3.2.2 Serializace sítě

Pro serializaci sítě spojení vzniklou spojováním síťových prvků použijeme serializační systém Unity popsaný v sekci 4.2.2, což je standardní způsob ukládání herního světa. Chceme přitom vzít v potaz, že v budoucích verzích naší hry bude v jeden okamžik načtena pouze část sítě. Při pohybu hráče se pak budou další

síťové prvky načítat a jiné zase mazat. Nutnost načtení celé sítě najednou by totiž značně omezovala velikost herního světa. Některé reference v síti tedy budou ukazovat na prvky, které ještě nebyly načteny, nebo už byly smazány. Taková situace se v kontextu Unity označuje jako *Cross-Scene References* a není podporována ani v nejnovější verzi Unity 2022.2, viz oficiální dokumentace této verze Unity [15]. Jedním z důvodů je, že identifikátor, který má každý herní objekt při serializaci přidělený, je unikátní pouze v rámci jedné scény, viz relevantní sekce manuálu Unity [16]. Aktuálně tedy není možné odkázat na herní objekt v jiné scéně.

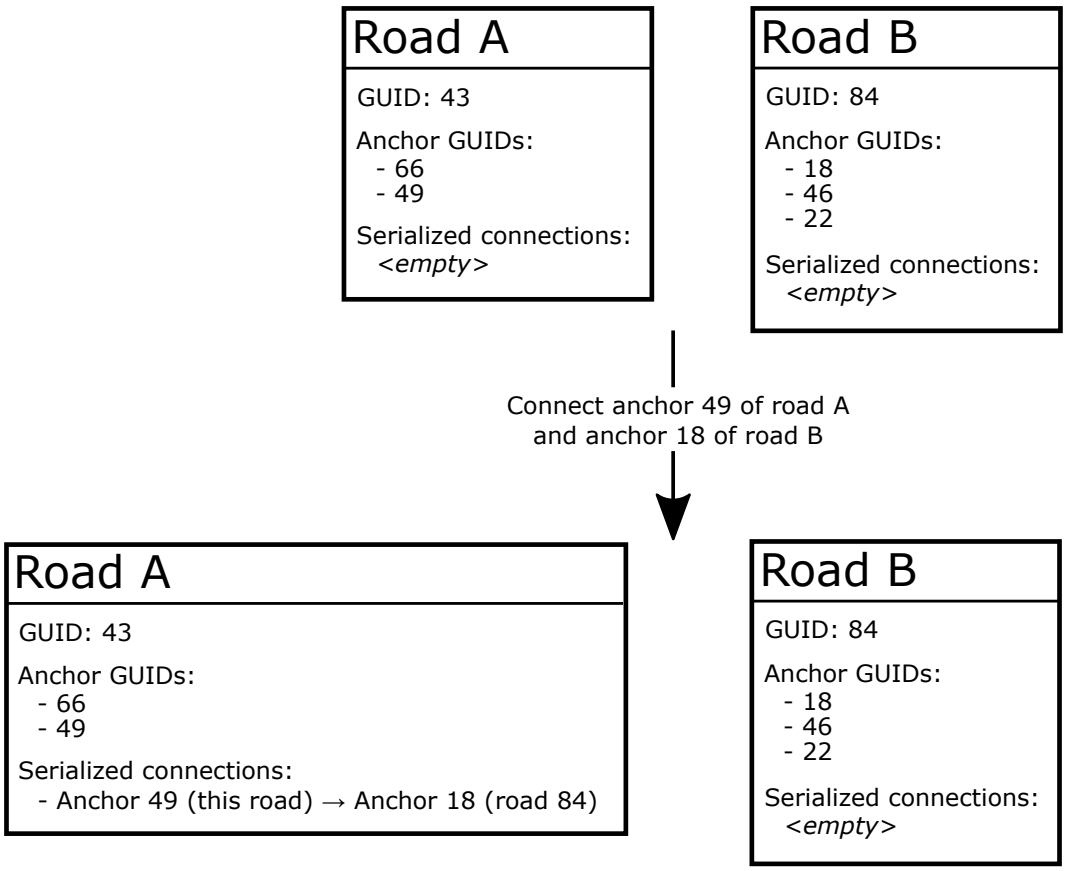
Častým řešením tohoto problému je zavedení vlastních globálně univerzálních identifikátorů (GUID) pro každý objekt, na který budeme chtít odkazovat, jak uvádí např. vývojář Unity William Armstrong [17]. Pro tento účel existuje projekt `Guid Based Reference` [18], který zavádí globální identifikátory pro herní objekty a mechanismus, jak objekt s daným identifikátorem získat. Tento projekt ovšem nepovažujeme za udržovaný, protože je umístěn pouze na webu GitHub (nikoli na standardním Unity Asset Store) a jeho poslední úprava byla provedena v roce 2020 (stav ke dni 24. 4. 2022). Proto jsme se rozhodli tento projekt v naší hře nepoužít.

Místo toho zavedeme vlastní globálně univerzální identifikátory pro každý síťový prvek a každou kotvu. K tomu využijeme strukturu `System.Guid`, což je standardní způsob zavedení takových identifikátorů (viz dokumentace této struktury [19]). Předpokládejme nyní, že spojujeme dva síťové prvky přes jejich kotvy. Vybereme jeden z nich a do něj informaci o spojení uložíme. Konkrétně do něj uložíme GUID jeho kotvy, GUID napojeného prvku a GUID napojené kotvy.

Příklad sledujme na obrázku 3.2, který znázorňuje spojení dvou silnic (všechny identifikátory jsme pro přehlednost zjednodušili). Na začátku neobsahuje ani jedna ze silnic žádná serializovaná spojení. Silnice `A` má GUID 43 a obsahuje 2 kotvy, silnice `B` má GUID 84 a obsahuje 3 kotvy. Následně spojíme kotvu 49 silnice `A` s kotvou 18 silnice `B`. Po provedení spojení se libovolně vybere jedna ze silnic (v tomto případě silnice `A`) a do ní se informace o spojení uloží tak, jak bylo zmíněno výše. Zejména zdůrazněme, že uložená informace nezahrnuje referenci na silnici `B`, ale pouze její GUID. Tím pádem nenastane problém týkající se *Cross-Scene References*, který jsme uvedli na začátku sekce.

Kdykoli pak v budoucnu budou oba prvky načtené, chceme, aby mezi nimi oboustranně vedly reference. To bude nutné např. při hledání nejkratší trasy po silnicích, kdy potřebujeme vědět, které silnice jsou na sebe napojené. Zavedení těchto referencí provedeme při deserializaci toho prvku, který bude z obou spojených prvků deserializován jako druhý. Jednoduše při načítání každého prvku zkontrolujeme, zda existuje nějaké serializované spojení z tohoto prvku do již načtených prvků, případně z již načtených prvků do tohoto prvku. Pokud ano, zavedeme příslušné reference.

Reference mezi spojenými síťovými prvky ovšem nebudeme ukládat přímo do samotných síťových prvků, protože na jednu kotvu může být napojené libovolné množství jiných kotev a reference by musela vést mezi každou dvojicí. Místo toho zavedeme koncept *křižovatek* (*junction*), kdy křižovatkou budeme v této sekci myslet seznam kotev, které jsou všechny navzájem propojeny. Příkladem může být křižovatka obsahující koncové kotvy čtyř silnic, nebo křižovatka obsahující kocovou kotvu chodníku a kotvu reprezentující dveře budovy. Z každé kotvy,



Obrázek 3.2: Proces uložení serializovatelné informace o spojení.

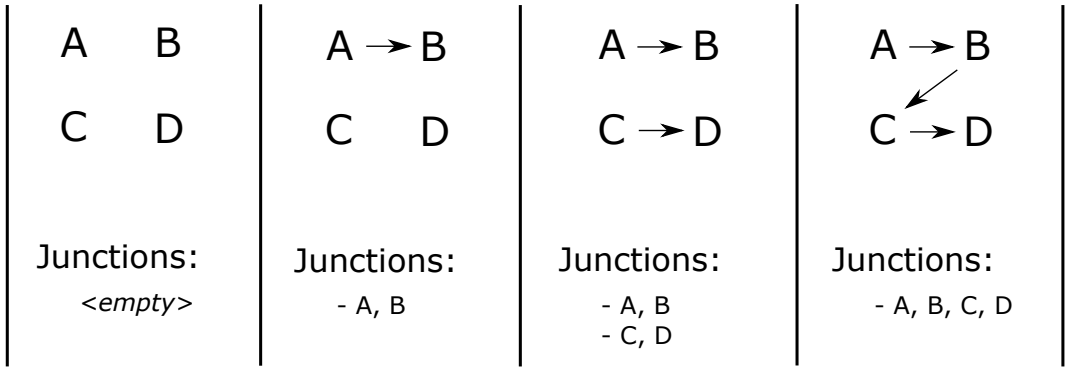
která je součástí nějaké křižovatky, pak zavedeme referenci na tuto křižovatku. Tím dostaneme možnost procházet síť v libovolném směru.

Při implementaci budeme muset dbát na to, že kvůli postupné deserializaci jednotlivých spojených prvků se může stát, že bude potřeba spojit dvě již vytvořené křižovatky do jedné. Příklad takové situace sledujme na obrázku 3.3. Ten znázorňuje postupnou deserializaci křižovatky spojující kotvy `A`, `B`, `C` a `D`, přičemž v každém kroku jsou již deserializovaná spojení označena šipkou. Nejprve deserializujeme nejprve spojení z `A` do `B` a poté spojení z `C` do `D`. Tím vzniknou 2 křižovatky. Následně deserializujeme spojení z `B` do `C`, což způsobí, že bude potřeba spojit obě již vzniklé křižovatky do jedné.

Podobně pokud nějaký prvek účastnící se spojení trvale smažeme, může být potřeba serializovaná spojení upravit tak, aby se křižovatka nerozpadla. To ovšem také není těžký problém a jeho řešení uvedeme v sekci 5.4.3.

3.2.3 Renderování silnic a chodníků

Již jsme popsali, jak budeme reprezentovat tvar silnic a chodníků a jak je budeme spojovat do křižovatek. Dále je ale potřeba určit, jak získat jejich 3D mesh. Ten bude nutný pro renderování, ale i pro detekci kolizí nebo kontrolu, na které silnici se zrovna nachází hráčovo auto. Tento problém je v podstatě totožný pro chodníky a pro silnice. Chodníky a silnice mohou být odlišně široké a vysoké a mít odlišné textury, ale mimo tyto detaily nevidíme dostatečnou přidanou hodnotu v použití odlišných algoritmů pro generování jejich tvarů. Popišme tedy situaci

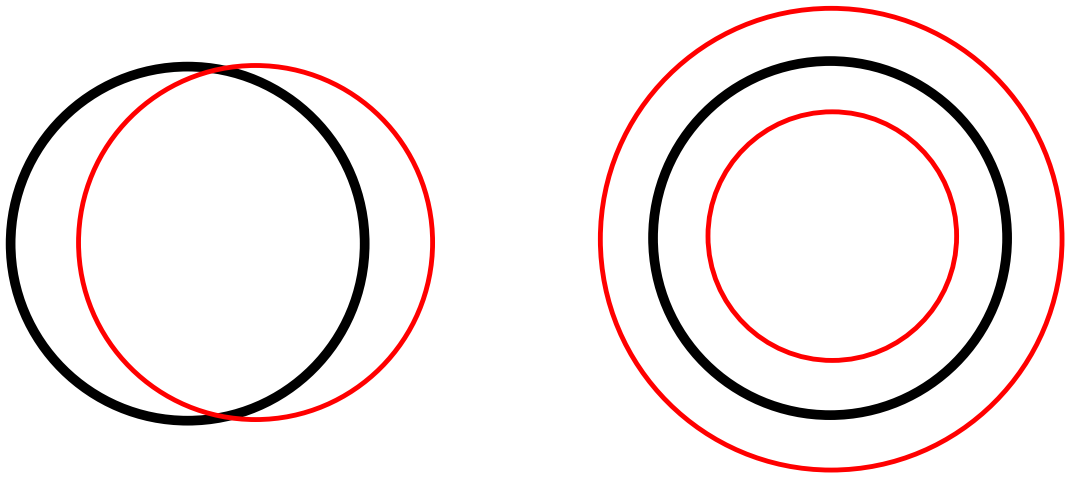


Obrázek 3.3: Postupná deserializace křižovatky, při které je potřeba spojit 2 křižovatky do jedné.

pouze pro případ silnic. Případ chodníků bude poté analogický.

V této sekci konkrétně popíšeme situaci, kdy máme pouze jednu silnici. V sekci 3.2.7 pak vyřešíme situaci, kdy je více silnic spojených do křižovatky.

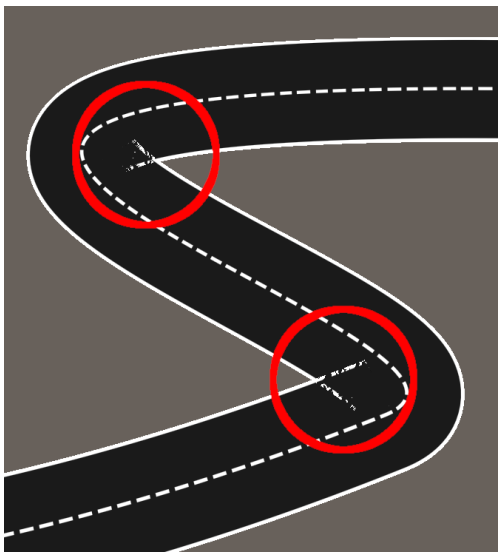
Tvar silnice je určen posloupností kubických Bézierových křivek (viz sekce 3.2). Naším cílem je vygenerovat mesh silnice. Tento problém se v podstatě redukuje na problém ve 2D, kdy chceme „posunout“ střed silnice tak, abychom dostali obě její krajnice. Zde je potřeba rozlišovat mezi *posunutím* (translací) a procesem, kterým z čáry na středu silnice vznikne krajnice silnice. Pro přehlednost nebudeme druhému z těchto procesů říkat posunutí, ale *offsetting*[1] (anglická synonyma jsou stroking nebo také extrusion). Offsettting tedy zahrnuje i určité zvětšení nebo zmenšení dané křivky. Rozdíl mezi posunutím a offsettingem na příkladu kružnice je uveden na obrázku 3.4, kde levou černou kružnici posuneme, zatímco na pravou černou silnici aplikujeme offsetting na levou i pravou stranu. Tím získáme červené kružnice.



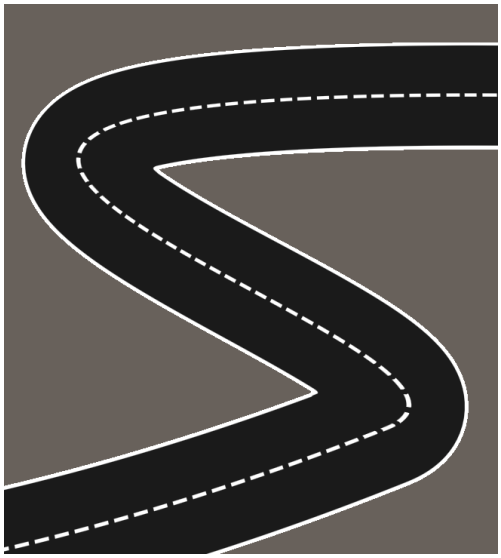
Obrázek 3.4: Porovnání posunutí kružnice (vlevo) a offsettingu kružnice (vpravo, přičemž offsetting byl proveden oběma směry).

Získané krajnice určí půdorys silnice a jeho protáhnutí do vertikálního směru už je jednoduché. Pro získání krajnic jsme vyzkoušeli následující 2 přístupy:

[1]Budeme také používat přídavné jméno offsetnutý a sloveso offsetnout.



Obrázek 3.5: Posloupnost odsazených bodů tvořících krajnici se může protínat, což způsobí artefakty jako na obrázku.



Obrázek 3.6: Odstraněním smyček tam, kde se posloupnost bodů protíná, problém vyřešíme.

1. Prvním možným přístupem je aproximovat Bézierovu křivku dostatečně jemnou lomenou čarou, čímž se poté problém značně zjednoduší. To uděláme tak, že na křivce popisující silnici vygenerujeme dostatečné množství bodů tak, aby žádné 2 nebyly příliš daleko od sebe. Jak toto zajistit vyřešíme v následující sekci 3.2.4.

   Následně pro každý z vygenerovaných bodů spočítáme tečný a potažmo normálový směr křivky v daném bodě. Poté z každého bodu vytvoříme dvě kopie, přičemž jednu posuneme v normálovém směru a druhou ve směru opačném k normálovému. Takto dostaneme dva seznamy bodů popisující jednotlivé krajnice silnice. Navíc tyto seznamy budou stejně dlouhé a body si v nich budou automaticky odpovídat, takže z nich snadno půdorys silnice snadno vygenerujeme – pro implementační detaily viz sekce 5.4.4.

   Jediná potíž, která může nastat, je, že se krajnice může sama protínat. To nastane, pokud silnice zatáčí s poloměrem příliš malým vzhledem k její šířce. Výsledkem jsou nepřirozené artefakty ve vzniklém meshi, resp. textuře, jako je vidět na obrázku 3.5.

   To vyřešíme zavedením procedury, která vezme na vstupu lomenou čáru a odstraní z ní smyčky (obrazněji řečeno „vystřihne ucha“). Tuto proceduru nazvěme `decrossing`. Použijeme ji postupně na obě krajnice každé silnice, čímž zmíněnou potíž vyřešíme. Jak bude `decrossing` fungovat vyřešíme v sekci 3.2.6. Výsledné zlepšení je pak vidět na obrázku 3.6.

2. Druhou možností, jak z tvaru silnice získat její půdorys, je vygenerovat offsettingem dvě Bézierovy křivky z křivky původní – jednu offsetnutou doprava a druhou doleva. Tyto pak budou reprezentovat krajnice silnice. Principiální problém tohoto přístupu je, křivka vzniklá offsetingem Bézie-

rovy křivky nemusí být (a až na triviální případy také není) polynomiální křivkou a tudíž ani Bézierovou křivkou – viz sekce Curve offseting v článku *A Primer on Bézier Curves* [20]. Pokud bychom se pro tento přístup přesto rozhodli, bylo by dále nutné určit, jak z krajnic vygenerovat půdorys silnice. To by bylo složitější, než v předchozím případě, kdy krajnice byly lomené čáry.

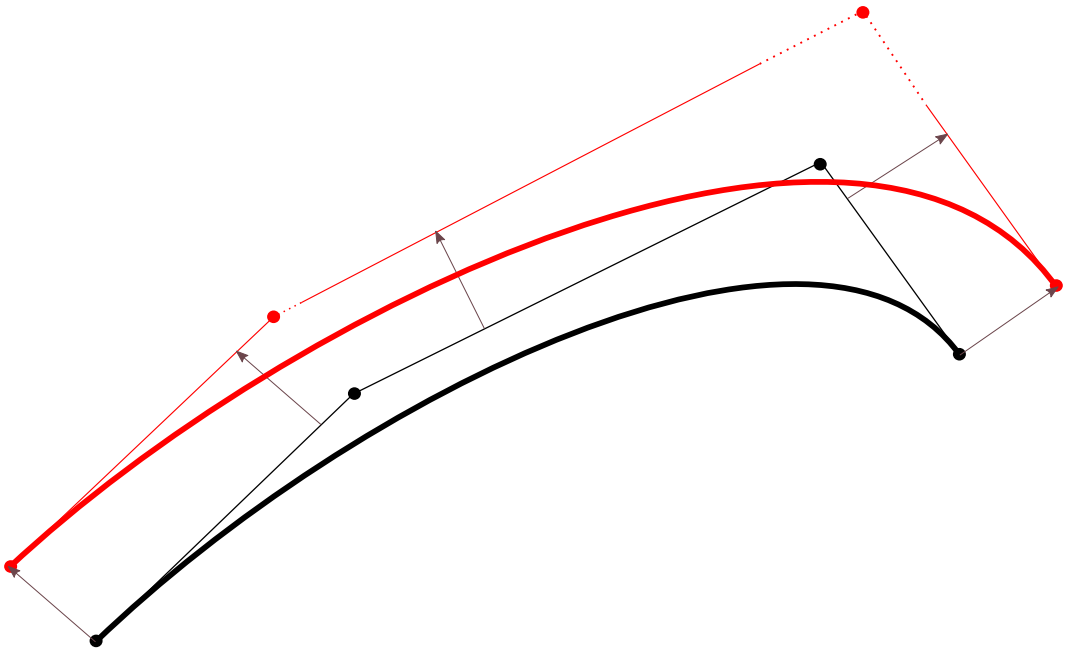
Ačkoli offsetnutou Bézierovu křivku nelze přesně popsat jinou Bézierovou křivkou, lze ji tak aproximovat. Při porovnání možných způsobů takové aproximace jsme vycházeli z článku Qualitative and Quantitative Comparisons of Offset Curve Approximation Methods [21]. Vzhledem k tomu, že postup získání krajnice uvedený v bodě 1 robustně produkuje dobré výsledky, zabývali jsme se pouze těmi postupy ve zmíněném článku, které by byly jednodušší na implementaci.

Jako jediný takový jsme vyhodnotili zjednodušenou verzi přístupu popsaného v článku Offsets of Two-Dimensional Profiles (Tiller, Hanson) [22], který je v tomto článku označován jako *Subdivision*. Jedná se o obecný algoritmus pro offsetting B-spline křivek, což je třída křivek obsahující mimo jiné Bézierovy křivky (viz článek B-spline curve z výukového textu MIT [23]).

Naše zjednodušení tohoto algoritmu spočívá ve dvou věcech. Zaprvé jeho vstupy omezíme na kubické Bézierovy křivky – jiné v našem programu nefigurují. Původní algoritmus navíc po provedení offsettingu vyhodnocuje kvalitu offsetnuté křivky a pokud ji vyhodnotí jako nedostatečnou, provede podrozdělení vstupní křivky na více kusů a celý postup zopakuje. Cílem podrozdělení je získat křivky, které jsou dostatečně jednoduché v tom smyslu, že „nezatáčejí příliš rychle“. Toto podrozdělování v naší verzi algoritmu provádět nebudeme.

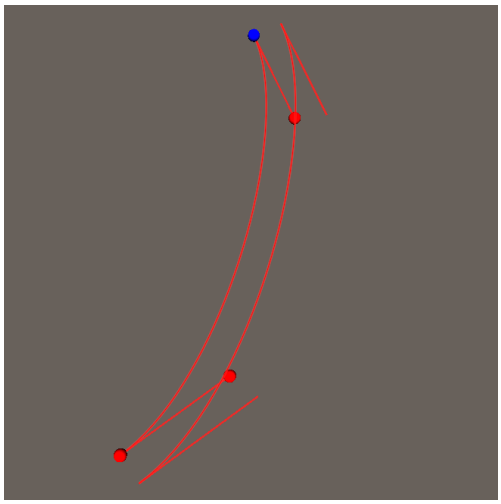
Zjednodušený algoritmus tedy bude fungovat následovně. Na vstupu mějme kubickou Bézierovu křivku určenou řídícími body $\beta_0, \beta_1, \beta_2, \beta_3$ a vzdálenost, o kterou chceme křivku offsetnout (spolu s informací, na kterou stranu chceme posouvat). Základní myšlenkou je, že o tuto vzdálenost kolmo posuneme úsečky $\beta_0\beta_1$, $\beta_1\beta_2$ a $\beta_2\beta_3$. Posunuté úsečky případně protáhneme tak, aby se sousední úsečky protínaly. Body průniku pak určí pozice nových kontrolních bodů. Sledujme tento proces na konkrétním případě na obrázku 3.7. Vstupem je černá Bézierova křivka určená černými kontrolními body. Spojením těchto bodů vzniknou tři černé úsečky. Každou z nich následně posuneme v kolmém směru o požadovanou vzdálenost, čímž získáme červené úsečky. Průsečíky protažených červených úseček jsou nové kontrolní body, které určí offsetnutou červených křivku. Ta je výstupem algoritmu.

V naší praktické implementaci musíme dále řešit případ, kdy jsou posunuté úsečky rovnoběžné nebo skoro rovnoběžné a není tedy možné získat jejich průsečík. V takovém případě je přirozené kontrolní bod posunout kolmo od jedné z úseček (nezáleží na tom, kterou vybereme, protože mají obě přibližně stejný směr). Dále v naší implementaci zpracováváme celou posloupnost křivek najednou, ne jen jednu křivku zvlášť. To však na algoritmu nic nezmění.

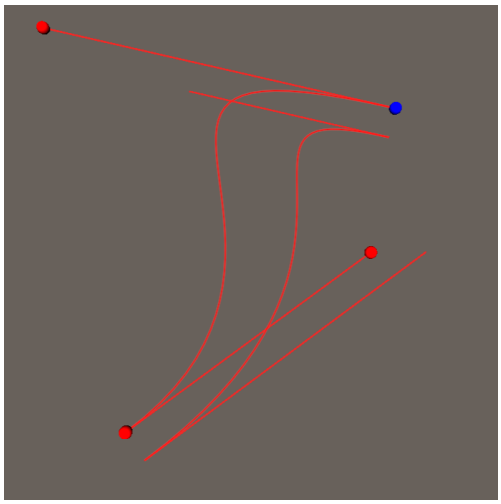


Obrázek 3.7: Ukázka Tiller-Hansonova algoritmu.

Experimentálně jsme zjistili, že křivka offsetnutá naší implementací Tiller-Hansonova algoritmu často funguje dobře, ovšem zároveň je často nepřesná, a to i v relativně jednoduchých případech. Příklad křivky, kterou naše implementace offsetne správně, je uveden na obrázku 3.8. V jiném případě, kdy je výchozí křivka stále poměrně jednoduchá, vypadá offsetnutá křivka nepřirozeně a nemůže být použita jako krajnice silnice. To je uvedeno na obrázku 3.9. Naopak použitím postupu uvedeného v bodu 1 se z obou tvarů vygenerují silnice, které vypadají přirozeně – to je uvedeno na obrázcích 3.10 a 3.11.

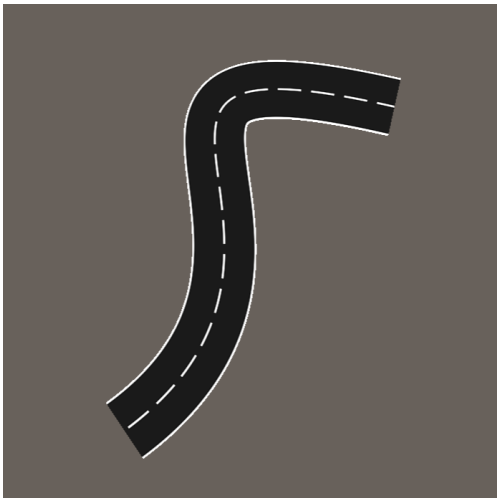


Obrázek 3.8: Příklad křivky, kterou naše implementace Tiller-Hansonova postupu posune správně.



Obrázek 3.9: Příklad křivky, kterou naše implementace Tiller-Hansonova postupu posune nepřirozeně.

Obrázek 3.10: Správně vygenerovaná silnice s tvarem daným křivkou na obrázku 3.8.



Obrázek 3.11: Správně vygenerovaná silnice s tvarem daným křivkou na obrázku 3.9.

Na základě uvedené analýzy jsme vybrali první z obou zvažovaných postupů kvůli jeho implementační jednoduchosti a robustnosti ve všech vyzkoušených případech.

3.2.4 Uniformně rozmístěné body na Bézierově křivce

V předchozí sekci 3.2.3 jsme rozhodli, že pro výpočet tvaru silnice budeme potřebovat generovat body na Bézierově křivce tak, abychom ji těmito body dostatečně hustě pokryli. Tedy aby žádné dva sousední body nebyly příliš daleko od sebe. To naštěstí není těžké – De Casteljauův algoritmus, který popíšeme v sekci 5.4.2, je dostatečně rychlý i pro velmi husté pokrytí body. Jeho nevýhodou (nebo spíše vlastností Bézierových křivek) je, že pokud začneme s uniformně rozmístěnými parametry t (viz sekce 5.4.2), nedostaneme uniformně rozmístěné body na křivce. Konkrétně v rovných úsecích budou body daleko dál od sebe než v zatáčkách. To způsobí následující 3 problémy:

1. Experimentálně jsme zjistili, že příliš jemný mesh (tj. mesh obsahující příliš velké množství bodů v malém prostoru) způsobuje problémy fyzikální simulaci enginu Unity. Konkrétně v takové situaci začnou dávat nepřesné výsledky metody pro zjišťování relativní polohy objektu vůči ostatním objektům, jako např. `Physics.OverlapBox` – zejména, pokud se objekty vůči sobě pohybují. Naší hypotézou je, že dochází k zaokrouhlovacím nepřesnostem v `single precision` datovém typu, který Unity pro reprezentaci bodů používá. Nepřišli jsme na žádný jiný důvod tohoto problému než přílišná hustota meshe, nicméně netvrdíme, že takový důvod neexistuje.

   Z toho důvodu chceme, aby měl námi vygenerovaný mesh předvídatelnou jemnost, kterou bude možné regulovat. Takovou možnost prosté využií De Casteljauova algoritmu nenabízí, viz výše.

2. V sekci 3.2.6 uvidíme, že pro posloupnost neuniformně rozmístěných bodů

se obtížněji určuje, kde tato posloupnost sama sebe protíná.

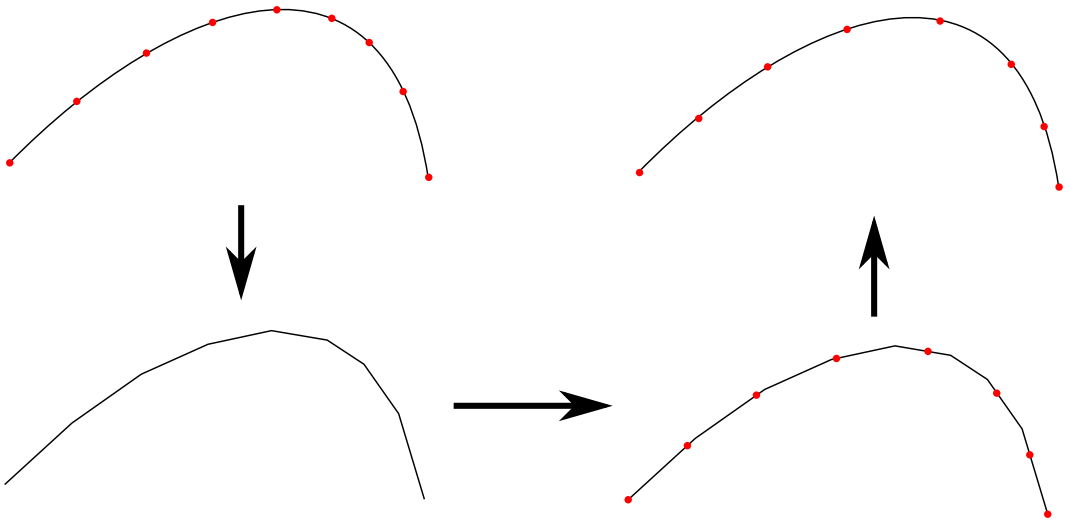
3. Získané body budeme následně používat ke generování meshe a potažmo pro generování UV souřadnic pro textury. Pokud body nejsou uniformně rozmístěné, je počítání UV souřadnic o něco obtížnější, nicméně řešitelné – to ukážeme v následující sekci 3.2.5.

Pro vyřešení všech zmíněných problémů použijeme algoritmus, který zajistí přibližně uniformní rozmístění bodů na posloupnosti navazujících Bézierových křivek. Ačkoli budou výsledné body pouze přibližně uniformně rozděleny, přesnost bude dostatečná pro všechny 3 případy.

Algoritmus bude fungovat ve dvou fázích. V první fázi vygeneruje neuniformně rozmístěné body opakovanou aplikací De Casteljauova algoritmu tak, aby tyto dostatečně hustě pokryly celou posloupnost křivek. Navíc zajistíme, aby počáteční bod první křivky i koncový bod poslední křivky byly pokryty.

V druhé fázi pak využije, že lomená čára vzniklá z vygenerovaných bodů poměrně dobře aproximuje tvar křivky. V zatáčkách, kde se toho „více děje“ jsou totiž body automaticky blíže u sebe (jak jsme to popsali výše). Vygenerujeme tedy uniformně rozmístěné body *na vzniklé lomené čáře* (což je jednoduché, pro detaily viz zápis algoritmu v pseudokódu). Tím pro každý bod získáme parametr t určený jeho vzdáleností od počátečního vrcholu lomené čáry. Na základě těchto parametrů poté body zobrazíme De Casteljauovým algoritmem zpět na křivky.

Příklad běhu algoritmu na posloupnosti o pouze jedné křivce je uveden na obrázku 3.12. Černou křivku nejprve dostatečně hustě pokryjeme červenými body, které určí lomenou čáru. Na té následně vygenerujeme uniformně rozmístěné body, které zobrazíme zpět na původní křivku.



Obrázek 3.12: Příklad běhu algoritmu pro generování uniformně rozmístěných bodů na křivce.

Dodejme, že jednodušší, ale méně efektivní verze druhé fázi tohoto algoritmu je popsána v článku *A Primer on Bézier Curves* v sekci *Tracing a curve at fixed distance intervals* [20].

Pro úplnost uvedeme zápis druhé fáze algoritmu v pseudokódu. Přitom předpokládáme, že v první fázi algoritmu si pro každý bod ukládáme jeho přibližnou

vzdálenost od začátku posloupnosti křivek. Tu vypočítáme jako vzdálenost po vzniklé lomené čáře od jejího počátečního vrcholy k danému bodu. Dále si pro každý bod ukládáme parametr t, kterým jsme bod získaly.

```
Input: sequence of sampled points P,
       target distance D
Output: sequence of uniform points Q

For each neighbouring pair u, v in P:
  While (last point in Q).Distance + D < v.Distance:
    d := (last point in Q).Distance + D
    t := u.t +
         (v.t - u.t) * (d - u.Distance) / (v.Distance - u.Distance)
    q := De Casteljau's image of t
    q.Distance := Euclidean distance of q from the last point in Q
                  + (last point in Q).Distance
    Add q to Q
```

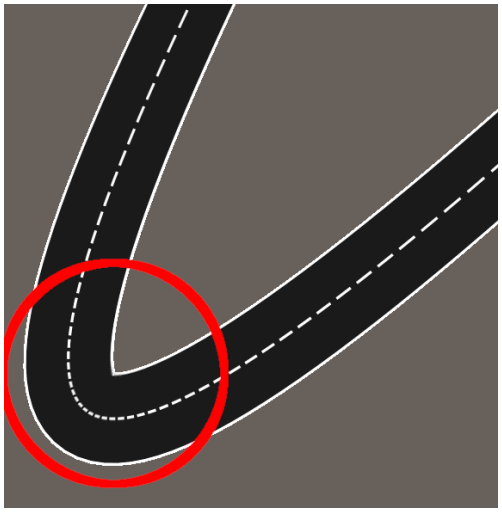
3.2.5 Generování UV souřadnic pro silnice a chodníky

Pokud by silnice a chodníky nebyly opatřeny texturami, bylo by pro hráče obtížné rozlišit silnice od chodníků, nebo dokonce silnice od země. Dále by bylo obtížnější určit, jakou rychlostí se pohybuje. Proto meshe silnic i chodníků texturou opatříme.

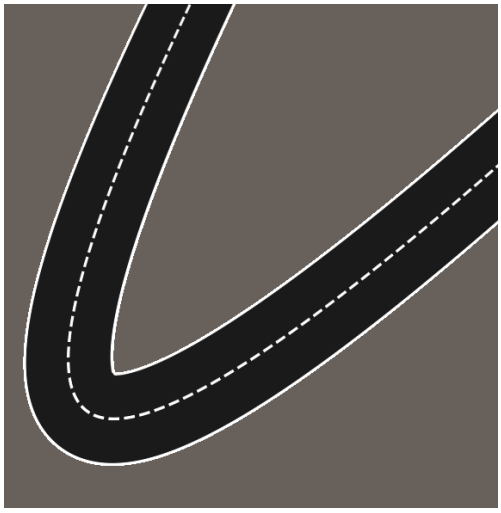
Tím pádem musíme pro každý bod meshe silnice (resp. chodníku) určit, jaká souřadnice na textuře mu odpovídá (tj. který pixel textury se na něj má nanést). Těmto souřadnicím se říká *UV souřadnice* a celému procesu pak *UV mapping*. Ve skutečnosti stačí UV souřadnice určit pro každý vrchol meshe, přičemž pro zbytek bodů meshe se provede interpolace. Pro detaily odkazujeme na článek o UV souřadnicích v manuálu k programu Blender [24].

Musíme tedy určit správnou UV souřadnici pro každý vrchol na obou krajnicích silnice. K tomu je třeba pro každý vrchol přibližně vědět, jak daleko je od začátku silnice. Protože jsme v předchozí sekci 3.2.4 rozhodli, že vrcholy meshe budou přibližně uniformně rozmístěny, není odhad vzdálenosti každého bodu problém – stačí uvážit jejich pořadí v posloupnosti vrcholů. Výsledné nanesení textury na mesh silnice je zachyceno na obrázku 3.14.

Pro doplnění předchozí sekce 3.2.4 uveďme, jak by bylo možné postupovat, pokud by body přibližně uniformně rozmístěné nebyly. Pokud bychom v takovém případě vzdálenost vrcholů od začátku silnice odhadovali pouze z jejich pořadí, byla by textura v zatáčkách „zmáčknutá“ a v rovných úsecích „natáhlá“ – to je uvedeno na obrázku 3.13. Tento problém by bylo možné vyřešit tím, že bychom vzdálenost k vrcholu po křivce odhadli vzdáleností vrcholu po lomené čáře. Jinými slovy bychom prošli popořadě vrcholy krajnice a sčítali přitom jednotlivé vzdálenosti mezi sousedy. Částečné součty nám pak dají odhad vzdálenosti každého vrcholu. To vede k výsledku, který je nerozlišitelný od obrázku 3.14.



Obrázek 3.13: Generování UV souřadnic z neuniformně rozmístěných bodů bez odhadu vzdálenosti.

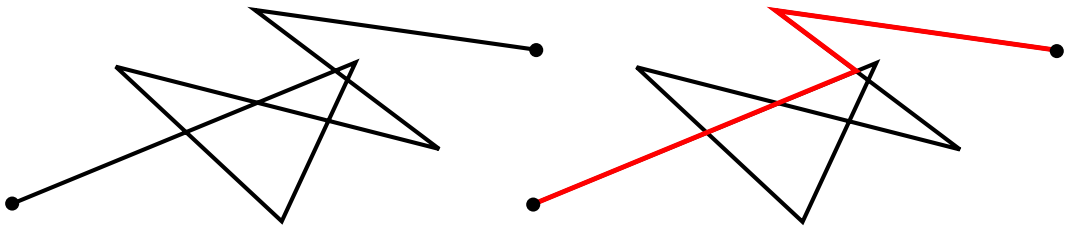


Obrázek 3.14: Generování UV souřadnic z uniformně rozmístěných bodů.

3.2.6 Decrossing

V sekci 3.2.4 jsme stanovili, že pro generování tvaru krajnice silnice je nutné umět z lomené čáry odstraňovat smyčky. Zvážili jsme 2 možné algoritmy řešící tento problém.

Zaprvé si můžeme všimnout, že se nejedná o nic jiného než o problém hledání průsečíků úseček. Lomená čára o n bodech obsahuje $n - 1$ úseček. Kdybychom identifikovali všechny jejich průsečíky (samozřejmě kromě bodů, kde na sebe sousední úsečky navazují), máme vyhráno. Pak už totiž stačí ve vzniklém grafu průsečíků (kde průsečíky a konce úseček tvoří vrcholy a úsečky se podle průsečíků rozpadnou na hrany) najít nějakou cestu z počátečního bodu lomené čáry do jejího koncového bodu (kde cesta je posloupnost vrcholů, ve které se vrcholy neopakují). Tato cesta tedy vznikne vystřižením nějakých částí původní čáry a z definice neobsahuje smyčky – tudíž je správným řešením problému. Příklad tohoto postupu je uveden na obrázku 3.15, kde vlevo je vstupní lomená čára, která je pak vpravo překryta červenou výstupní čarou. Na obrázku jsme jako výslednou cestu vybrali cestu nejkratší, což by nejspíše dělala i praktická implementace, aby minimalizovala složitost výsledku.



Obrázek 3.15: Příklad protínající se lomené čáry (vlevo) a výstupu algoritmu pro odstranění smyček (vpravo).

Rozeberme složitost algoritmu. Protože naše body budou vždy v rovině, stačí

na hledání průsečíků použít klasický postup zametání roviny přímkou – pro informace o tomto přístupu viz kniha Průvodce labyrintem algoritmů [25]. Tím dosáhneme asymptotické složitosti $O((n+p)\log n)$, kde p je počet nalezených průsečíků. V grafu průsečíků pak je $O(n+p)$ hran, takže hledání nejkratší cesty složitost nezhorší.

Jiným přístupem je všimnout si, že tento algoritmus je pro naše účely příliš obecný (např. čára na obrázku 3.15 se jako krajnice silnice nejspíše nevyskytne). Bylo by proto možné zjednodušit si práci pozorováním, že stačí řešit pouze velmi specifické instance problému. Konkrétně:

1. Vstupní lomená čára bude mít poměrně jednoduchou strukturu, protože bude reprezentovat krajnici silnice. To zejména znamená, že průsečíků bude typicky velmi málo.

2. Uvažme, co by znamenalo, kdyby lomená čára protínala sama sebe ve dvou velmi vzdálených bodech. Pak by se pravděpodobně nejednalo o ostrou zatáčku silnice, nýbrž o chybu herního návrháře, který umístil silnici tak, že protíná sama sebe. Nejenom, že takový případ se můžeme rozhodnout neřešit, dokonce je v takovém případě lepší smyčku neodstranit. Návrhář pak místo zcela nesmyslně useknuté silnice uvidí lokální artefakty v meshi, což mu více napoví, kde je chyba.

3. Posledním předpokladem bude, že vzdálenost každé dvojice sousedních vrcholů vstupní lomené čáry bude přibližně stejná. Bez tohoto předpokladu náš jednodušší algoritmus nemůže robustně fungovat (to vysvětlíme dále). Platnost tohoto předpokladu jsme zajistili v sekci 3.2.4.

Algoritmus využívající tato zjednodušení pak bude velmi jednoduchý. Nad lomenou čarou nebude uvažovat jako nad posloupností hran, ale jako nad posloupností vrcholů. Tuto posloupnost vrcholů postupně projde a pokud narazí na bod blízký nějakému dřívějšímu bodu, oznámí smyčku. Proto je důležité, aby každá dvojice sousedních vrcholů byla přibližně stejně vzdálená – jinak by sousedy blízko u sebe bylo těžké rozlišit od průsečíků a sousedy daleko od sebe by zase představovali riziko, že průsečík nezaznamenáme. Představme si např., že bychom místo přibližně uniformně rozmístěných bodů použili body nanesené De Castaljauovým algoritmem (viz 3.2.4). Pokud by se pak křivka protínala ve dvou relativně rovných úsecích, mohlo by se snadno stát, že by se k sobě žádné dva body posloupnosti dostatečně nepřiblížily a náš algoritmus by průsečík nezaregistroval.

Abychom neztratili vzájemnou korespondenci mezi body na levé a pravé krajnici, nebude algoritmus smyčky mazat – místo toho všechny body ve smyčce nahradí hodnotou daného průsečíku. Navíc bude algoritmus při hledání blízkých bodů uvažovat jen ty body, které nepotkal příliš dávno (viz bod 2 výše zmíněných zjednodušení) a které zároveň nepotkal příliš nedávno (aby nedocházelo příliš často k *false positive* případům, kdy algoritmus nahlásí sousední vrcholy jako průsečík, protože jsou dostatečně blízko u sebe). Pro následující popis sledujme pseudokód algoritmu na další straně.

Efektivita algoritmu stojí a padá na efektivitě struktury `M`. Nabízelo by se použít nějakou datovou strukturu na bázi Space partitioning – buď strukturu stromovou, jako např. Quadtree, nebo na bázi hešování. Experimentálně se ale

```
Input: sequence of points P,
       proximity threshold T,
       minimal lookback distance A,
       maximal lookback distance B
Output: sequence of points Q

M := empty set of points
For each p in P:
  Add p to Q
  If M contains a point r that satisfies all of the following:
    1. |pr| <= T
    2. r was added at least A iterations ago
    3. r was added at most B iterations ago
  Then:
    int := the middle of the line segment pr
    Set to int everything in Q starting from r
    Remove from M everything starting from r
    Add int to M
  Else:
    Add p to M
```

ukázalo, že stačí jednoduchá fronta, do které přidáváme nové body a zahazujeme ty příliš staré. Práce v herním editoru pak není zcela plynulá, nicméně upřednostníme práci na hře samotné, která je naším primárním produktem, před optimalizací tohoto algoritmu. Aby byl ale tento přístup přijatelně rychlý, je potřeba dbát při implementaci na efektivitu. Zejména je potřeba implementovat vlastní cyklickou frontu místo `System.Collections.Generics.Queue` (protože by nás příliš zpomalovalo opakované vytváření enumerátoru) a nepočítat zbytečně odmocniny ve vzdálenostech bodů. Konkrétně nefunguje následující způsob kontroly, zda je nějaký dřívější bod dostatečně blízký novému bodu:

```
if (Vector3.Distance(otherPoint, thisPoint) <= proximityThreshold) {
    // Announce intersection.
}
```

Místo toho použijeme následující způsob, který je ekvivalentní, ale rychlejší:

```
if (otherPoint.x - thisPoint.x <= proximityThreshold &&
    thisPoint.x - otherPoint.x <= proximityThreshold &&
    otherPoint.y - thisPoint.y <= proximityThreshold &&
    thisPoint.y - otherPoint.y <= proximityThreshold) {
    if ((otherPoint - thisPoint).sqrMagnitude <=
        proximityThreshold * proximityThreshold) {
        // Announce intersection.
    }
}
```

Jak odstranění odmocniny v počítání vzdálenosti, tak přidání počáteční jednodušší podmínky (která je pro nahlášení průsečíků nutná, i když ne postačující) v našem případě postřehnutelně zrychlilo práci s editorem.

3.2.7 Renderování křižovatek

V sekci 3.2.3 jsme vyřešili tvorbu meshe pro samostatně stojící silnici. Nyní se zaměřme na situaci, kdy je více silnic spojených do křižovatky. Situace pro chodníky bude opět zcela analogická. Uveďme 3 možnosti, jak při renderování křižovatek postupovat:

1. Mohli bychom mesh křižovatky vytvořit ručně ve vhodném externím nástroji, poté jej importovat do Unity, přidat do herního světa a silnice k němu napojit. To by v praxi nutně znamenalo, že bychom často jeden mesh používali pro více křižovatek – jinak by byl celý proces příliš zdlouhavý. To by mohlo uškodit variabilitě herního světa a tedy zábavnosti hry. Na druhou stranu tento postup nabízí větší flexibilitu co do tvorby speciálních křižovatek přesně podle představ herního návrháře.

2. Dále je možné generovat tvar křižovatky automaticky z tvaru spojených silnic. To je technicky netriviální, ale umožňuje mnohem rychlejší tvorbu herního světa. Kromě usnadnění práce v herním editoru by automatické generování křižovatek byl první krok k procedurálnímu generování silniční sítě. Na druhou stranu tento přístup neumožňuje upravovat jednotlivé křižovatky podle přání herního návrháře.

3. Oba předchozí přístupy lze zkombinovat, tj. ve výchozím stavu generovat tvary křižovatek automaticky, ale zároveň umožnit hernímu návrháři, aby konkrétní křižovatce přiřadil ručně vytvořený mesh.

Pro budoucí vývoj hry se jednoznačně chceme přiklonit ke třetímu přístupu, který kombinuje všechny výhody prvních dvou. V této práci se zaměříme na těžší z obou částí, tedy automatické generování tvaru křižovatek. Zároveň díky obecnosti co do spojování objektů do silniční sítě (viz sekce 3.2.1) nebude v budoucnu těžké přidat část druhou.

Vyřešme tedy problém automatického generování křižovatek. Podle toho, jak jsme spojování silnic a ostatních objektů zadefinovali v sekci 3.2.1, lze spojit libovolnou kotvu silnice s libovolnou kotvou další (ne nutně jiné) silnice. To ale znamená velké množství různých typů křižovatek, se kterými by si náš algoritmus musel umět poradit. Začneme tedy omezením problému, které co nejméně ubere na použitelnosti editoru a zároveň situaci zjednoduší. Konkrétně se omezíme na dva případy:

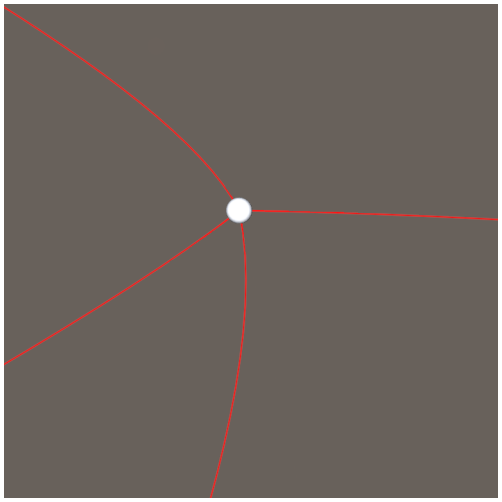
1. Křižovatky, u kterých jsou všechny silnice napojeny svým koncovým bodem (tj. koncovou kotvou).

2. Křižovatky, u kterých je jedna silnice napojena bodem jiným než koncovým a ostatní jsou připojeny koncovými body.

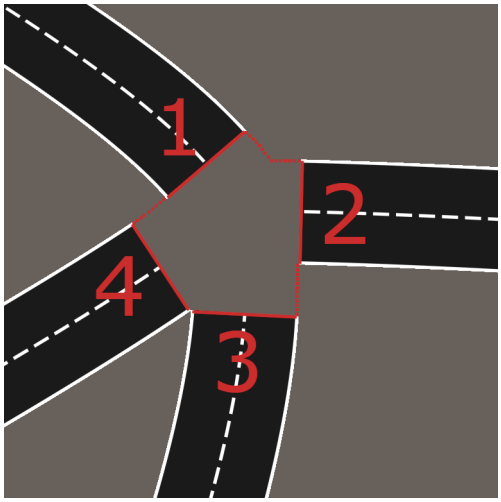
Praktická zkušenost autora s herním editorem ukázala, že toto omezení není příliš limitující.

Zaměřme se tedy nejprve na první z obou případů. Příklad vstupu našeho algoritmu je na obrázku 3.16. Pro jednoduchost zorientujeme všechny spojené silnice tak, aby začínaly v křižovatce (tedy aby jejich první kotva byla ta, přes kterou jsou spojeny). V prvním kroku algoritmu chceme zajistit, aby se meshe žádných

dvou silnic nepřekrývaly. Spojené silnice nejprve seřadíme po směru hodinových ručiček okolo spojovací kotvy. Pro každou silnici pak určíme bod, ve kterém se jako první začne dotýkat některé ze sousedních silnic. Od tohoto bodu zbytek jejího meshe odstraníme řezem kolmým na směr silnice. Stav po této operaci je vidět na obrázku 3.17 (červené symboly jsou přidané jako vysvětlivky).



Obrázek 3.16: Příklad křižovatky se 4 spojenými silnicemi. Meshe silnic i konrolní body křivek jsou pro přehlednost skryty.



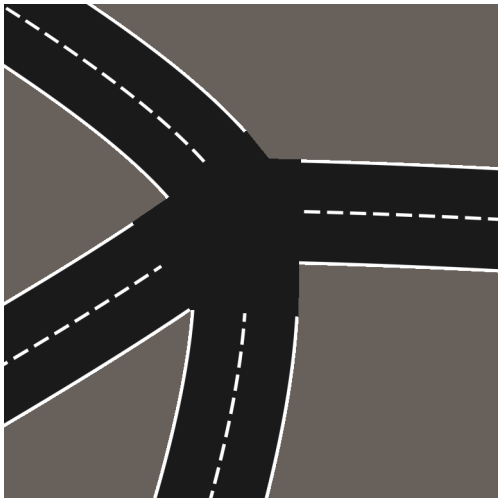
Obrázek 3.17: Pro každou sousední dvojici silnic zjistíme průsečík jejich odpovídajících krajnic. Tím dostaneme tvar křižovatky, přičemž meshe silnic náležitě ořízneme.

V druhém kroku algoritmu vezmeme konce původních a ořízlých krajnic všech silnic a vygenerujeme z nich mnohoúhelník, který bude půdorysem výsledného meshe reprezentujícího křižovatku. Výsledek je vidět na obrázku 3.18. Obrázek 3.19 pak ukazuje příklad křižovatky spojující silnice různých šířek – i na něm funguje algoritmus spolehlivě.

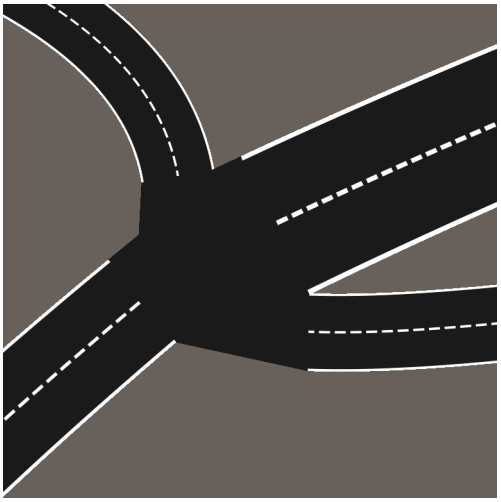
Průnik dvou sousedních silnic odpovídá průniku jejich odpovídajících krajnic. Zbývá tedy vyřešit, jak určit průnik dvou krajnic. Přesněji chceme určit takový jejich průnik, který je v určitém smyslu blízko uvažované křižovatce (pokud takový existuje). Ve větší vzdálenosti od uvažované křižovatky se totiž silnice mohou dále protínat (např. mohou být spojené další křižovatkou, nebo mohou být herním návrhářem nevhodně umístěny tak, že se protínají). Nechceme, aby tyto vzdálené interakce silnic ovlivňovaly vzhled vygenerované křižovatky.

Tento problém je podobný problému `decrossing`, který jsme vyřešili v sekci 3.2.6. Opět vyjdeme z toho, že krajnice silnice můžeme reprezentovat lomenou čarou s uniformně rozmístěnými vrcholy – takovou reprezentaci získáme při renderování silnic, jak jsme stanovili v sekcích 3.2.3 a 3.2.4. Cílem je tedy pro dvě lomené čáry najít jejich průnik, který neleží příliš daleko od začátku kterékoli z čar (pokud takový existuje).

Toho bychom opět mohli docílit obecným algoritmem pro hledání průsečíků přímek (např. technikou zametání roviny, jejíž detaily jsou uvedeny v knize průvodce labyrintem algoritmů [25]). Takový algoritmus bude ovšem muset určitě uvážit každý z vrcholů obou krajnic (ve skutečnosti bude mít dokonce horší než



Obrázek 3.18: Vygenerovaná křižovatka se 4 silnicemi.



Obrázek 3.19: Další příklad vygenerované křižovatky, tentokrát s různě širokými silnicemi.

lineární složitost – viz sekce 3.2.6). Raději využijeme toho, že uvažujeme pouze průsečíky nacházející se velmi blízko křižovatky, tedy začátku obou krajnic. Budeme jednoduše procházet všechny možné dvojice indexů `(i, j)`, kde `i` odkazuje na `i`-tý vrhol v jedné krajnici a `j` na `j`-tý vrchol v druhé krajnici. Tyto dvojice seřadíme podle vyššího z obou indexů vzestupně. Tím pádem budeme nejdřív uvažovat ty dvojice vrcholů, které jsou blízko křižovatce. Jakmile narazíme na dvojici vrcholů, které jsou dostatečně blízko u sebe, ohlásíme průsečík (ve skutečnosti se ještě podíváme na několik dalších dvojic pro případ, že by ty byly ještě blíž k sobě). Kdy jsou body „dostatečně blízko“ snadno odhadneme díky tomu, že je mezi body přiližně konstantní mezera (jako mezní hodnotu konkrétně zvolíme dvojnásobek této mezery). Musíme ovšem pamatovat na to, že průsečík nemusí existovat vůbec. Proto přidáme heuristiku, která nás v takovém případě zastaví – pokud se určitý počet iterací od sebe vrcholy pořád vzdalují, skončíme. Pro úplnost uvedeme popsaný algoritmus v pseudokódu na další straně.

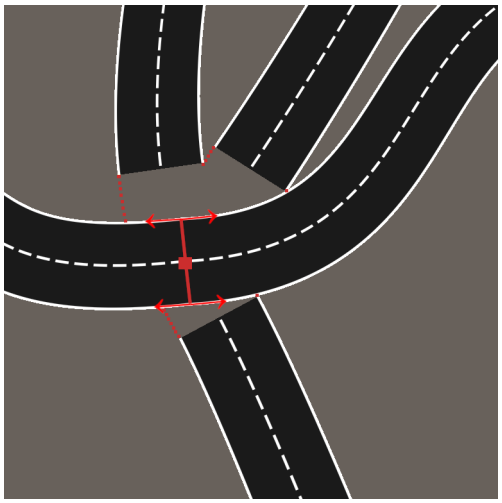
Nyní se zaměřme na druhý povolený případ, kdy křižovatka obsahuje právě jednu silnici, která není napojena svým koncovým bodem. Typickým příkladem je nájezd vedlejší silnice na hlavní, kde nájezd končí (resp. začíná), ale hlavní silnice nezačíná ani nekončí. Takových nájezdů může být v jednom místě více, a to z obou stran hlavní silnice. V následujícím popisu zůstaňme u označení *hlavní* a *vedlejší* silnice.

Zaprvé zvažme možnost převést tento problém na předchozí případ. Nabízí se pomyslně rozdělit hlavní silnici v místě spojení na dvě silnice a dále postupovat jako v předchozím případě. To je robustní způsob, jak problém vyřešit. Dokonce tak můžeme zrušit omezení na spojování silnic, které jsme zavedli na začátku sekce. Důsledkem ovšem je, že krajnice silnic mohou mít vystřižené libovolné části (do teď mohly být pouze zkráceny na koncích). To způsobí komplikace při generování meshe, kvůli kterým jsme se rozhodli tento postup do naší práce nezahrnout (nicméně program je na takové budoucí rozšíření připraven). Místo toho navrhneme postup, jak do meshe hlavní silnice vůbec nezasahovat a křižovatku

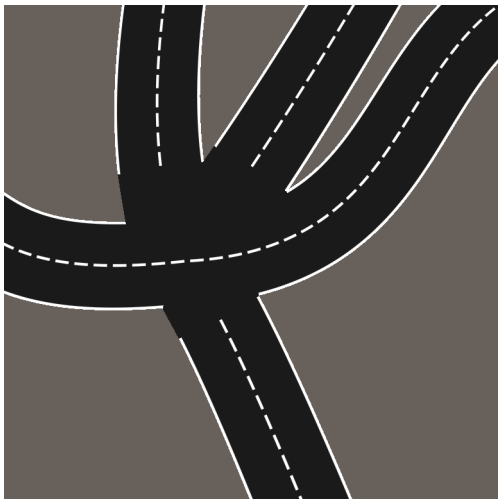
```
Input: sequence of points A,
       sequence of points B,
       distance threshold T,
       lookahead after intersection L,
       patience P
Output: indexes of elements in A and B represeting an intersection
        (if it exists)

For i from 0 up to min(A.length, B.length) - 1:
  If an intersection was detected L iterations ago, terminate.
  For j from 0 up to i:
    If (distance between A[i] and B[j]) <= T and
       no closer intersection was detected:
      Announce intersection on idexes i, j.
    Do the same check with swapped i and j.
  If the closest pair got further consecutively for the last P
  outer iterations, terminate.
```

k němu pouze „přilepit“. Toto rozhodnutí děláme čistě za účelem jednodušší implementace.

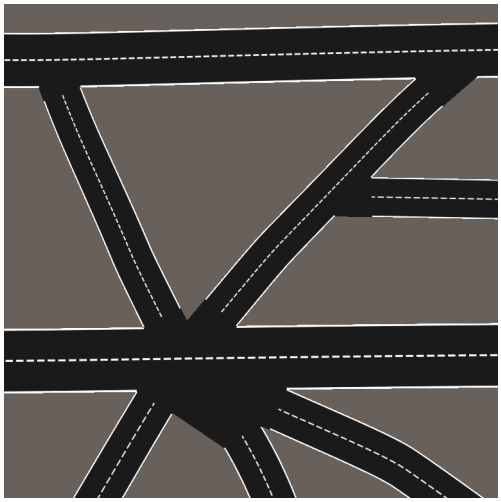
Začneme tím, že vedlejší silnice ořízneme stejně jako v předchozím případě. Poté prohledáme krajnice hlavní silnice směrem od spojovací kotvy na obě strany a poznamenáme si body, ve kterých krajnice vedlejších silnic protínají krajnici hlavní silnice. Proces je vyznačen na obrázku 3.20. Opět hledání zastavíme, pokud se od vedlejších silnic začneme příliš vzdalovat – podobně, jako v předešlém algoritmu. Do půdorysu křižovatky pak zahrneme jak mnohoúhelník tvořený konci vedlejších silnic, tak i část krajnice hlavní silnice určenou prvním a posledním dotykem vedlejších silnic.



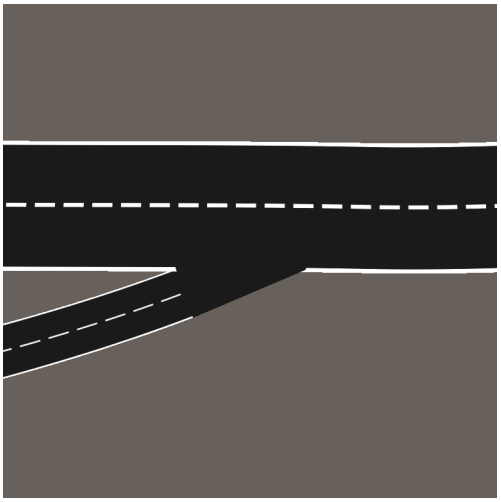
Obrázek 3.20: Po seříznutí vedlejších silnic nalezneme první a poslední bod, ve kterém vedlejší silnice protínají silnici hlavní.



Obrázek 3.21: Výsledná křižovatka skládající se ze dvou částí na obou stranách hlavní silnice.



Obrázek 3.22: Příklad komplexnějšího systému křižovatek s nájezdy různých tvarů.



Obrázek 3.23: Příklad nájezdu užší vedlejší silnice na širší hlavní silnici.

Celý postup zopakujeme pro obě strany hlavní silnice, pokud se na obou stranách nachází vedlejší silnice. Příklad dokončené křižovatky je na obrázku 3.21. Na obrázcích 3.22 a 3.23 jsou pak příklady křižovatek, kde se spojují silnice různých šířek, což algoritmus dobře zvládne.

3.3 Herní mechanismy

Dále vyřešme fungování samotné hry, která se bude na vytvořené silniční síti odehrávat. To zahrnuje pohyb chodců a hráčova auta a jejich vzájemnou interakci.

3.3.1 Ovládání chodců

Důležitou součástí herního světa jsou chodci pohybující se po chodnících. Jsou jak jediným dynamickým prvkem světa (kromě hráčova auta), tak i předmětem herní mise. Je tedy nutné, aby jejich pohyb vypadal přirozeně. K tomu je potřeba, aby chodci sledovali nějaký cíl, aby k tomuto cíli volili nejkratší možnou cestu a aby byli schopni vyhýbat se statickým překážkám i ostatním pohybujícím se chodcům. Navíc je nutné chodce do světa nějak přirozeně umisťovat. Nebylo by např. vhodné, aby se zničeho nic objevovali na chodnících.

Nejprve vyřešme problém vzniku chodců a výběr jejich cíle. Za tímto účelem zavedeme do světa koncept obytných domů. Takový dům bude herním návrhářem speciálně označen a vede k němu aspoň jeden chodník. Z každého obytného domu pak budou v náhodných intervalech vycházet chodci na navazující chodníky. Vycházení chodců z daného domu budeme modelovat Poissonovým procesem, takže časový interval mezi vznikem dvou chodců bude sledovat exponenciální rozdělení[2]. Střední hodnotu tohoto rozdělení může herní návrhář upravit a simulovat tím různě rušné domy. Pro každého chodce zvolíme náhodně rychlost jeho chůze

[2]Z praktických důvodů příliš malé hodnoty zakážeme, aby se dva chodci nevygenerovali na sobě.

z normálního rozdělení s dolní a horní mezí. Každý chodec si následně z ostatních domů dosažitelných po chodnících náhodně vybere jeden a ten bude cílem jeho pohybu.

Tím pro daného chodce získáme jeho cíl. Není také těžké vypočítat nejkratší trasu po chodnících od chodce k jeho cíli. Jakmile k cíli dorazí, odstraníme chodce ze světa. To nebude nepřirozené, protože v tu chvíli se bude nacházet u dveří obytného domu (resp. herní návrhář má možnost to tak zařídit tím, že přístupový chodník napojí k domu v místě dveří). Ovšem samotný pohyb po chodnících, které mohou mít libovolný tvar, není jednoduchý problém. Ještě těžší začne být, jakmile je vyžadováno vyhýbání se pohybujícím se překážkám. Zde využijeme výhod enginu Unity, který nabízí vlastní navigační systém. Pro fungování systému stačí meshe chodníků značit jako *walkable* a následně pro každého chodce určit pozici, na kterou má dojít. Vše ostatní – hledání nejkratší cesty, navigace po chodníku a vyhýbání se překážkám – je vyřešeno automaticky. Chodec je pro účely navigace aproximován válcem, u kterého můžeme specifikovat průměr, výšku, maximální výšku schodu, na který dokáže vystoupit, a další.

Experimentálně se ukázalo, že tento systém funguje pro účel navigace chodců dobře a výsledek vypadá přirozeně – chodci se pohybují výhradně po chodnících, navzájem se vyhýbají, zrychlují a zpomalují plynule, apod. Navíc se dá očekávat, že v budoucích verzích Unity bude navigační systém dále vylepšován.

Dodejme, že pro fungování navigačního systému je předem nutná jeho inicializace. Během ní se předpočítá dvoudimenzionální navigační mesh na povrchu chodníků (anglický název tohoto procesu je *baking*). Po tomto navigačním meshi se poté chodci pohybují. V herních světech, které jsme vytvořili, trvá tato operace jednotky sekund. Systém ji umožňuje provést v herním editoru, přičemž výsledek se serializuje a za běhu hry je ihned dostupný. Zde je nicméně nutné myslet na budoucí rozšíření naší hry, konkrétně na postupné načítání světa, případně procedurální generování nekonečného světa. V takové situaci není možné předvýpočet navigačního meshe provést v editoru. Místo toho je nutné jej provést za běhu po načtení (resp. vygenerování) nové části světa. To ale výchozí navigační systém Unity nepodporuje. Z tohoto důvodu rovnou použijeme nestandardní balíček `AI Navigation` (`com.unity.ai.navigation`), který poskytuje stejnou funkcionalitu jako výchozí navigační systém a navíc umožňuje provést zmíněný předvýpočet navigačního meshe za běhu programu. Tento balíček je nyní v experimentální verzi. Přesto jej považujeme za dostatečně spolehlivý, protože je vyvíjen společností Unity, která má v plánu jej v budoucnosti začlenit do standardního navigačního systému – viz vyjádření vývojáře Unity Maxime Plantady [26].

3.3.2 Herní mise

Zcela zásadní pro zábavnost hry jsou konkrétní detaily fungování herní mise. V průběhu mise je potřeba vybrat z chodců potencionálního zákazníka a jeho požadovanou destinaci. Domníváme se, že je přirozené vybírat za potencionální zákazníky jak chodce, kteří jsou blízko hráči, tak chodce, kteří jsou od hráče daleko. To dobře aproximuje práci reálného taxikáře, který dostává různé typy zakázek. Nicméně z chodců nemůžeme vybrat zcela náhodně – chodci se totiž pohybují jen po chodnících a od hráče nechceme, aby se pohyboval jinak než po silnicích. Potencionální zákazník se tedy musí nacházet na chodníku, který je

vedle nějaké silnice a zároveň k této silnici musí být možné dojet autem z aktuální pozice hráčova auta. Množinu chodců, kteří tyto podmínky splňují, získáme snadno prohledáváním silniční sítě od aktuální pozice hráčova auta.

Z této množiny ovšem může být nutné některé chodce vyškrtnout, protože jim nelze přiřadit vhodnou destinaci, tedy kam by jako zákazníci chtěli dovézt a odkud by pak pokračovali dál pěšky. Tato destinace musí splňovat 4 podmínky:

1. Destinace musí být chodník, protože chodci chodí jen po chodnících.

2. Destinace musí být dosažitelná po silnicích z aktuální pozice chodce, aby hráč naloženého zákazníka opravdu mohl na jeho destinaci dovézt. To tedy mimo jiné znamená, že destinace musí být chodník umístěný vedle nějaké silnice.

3. Z destinace musí být dosažitelný nějaký dům, ke kterému pak vysazený zákazník půjde – kdybychom zákazníka vysadili např. na izolovaném chodníku, musel by zůstat stát na místě nebo chodit tam a zpátky, přičemž obojí by bylo velmi nepřirozené.

4. Destinace musí být co možná nejblíže domu, do kterého pak zákazník zamíří. Jinak by po vysazení ušel kus cesty, který jsme jej mohli ještě dovézt, což by opět bylo nepřirozené.

Abychom problém zjednodušili, provedeme na začátku hry následující předvýpočet. Pro každý obytný dům od něj najdeme nejkratší cestu po chodnících takovou, že vede k některému chodníku nacházejícímu se vedle silnice. Pokaždé si koncový bod cesty na tomto chodníku označíme. Označené budou tedy ty body, které by v budoucnu mohly být zákazníkem vybrané jako destinace. Z označeného bodu pak totiž zákazník může pokračovat do obytného domu, ze kterého jsme bod označili.

Následně můžeme pro konkrétního chodce rozhodnout, zda mu dokážeme vybrat vhodnou destinaci. To uděláme tak, že prohledáme síť silnic počínaje pozicí chodce a budeme do výsledné množiny potencionálních destinací přidávat označené body, na které narazíme. Přitom si určíme maximální hloubku, do které budeme síť silnic procházet – tedy maximální délku trasy od zákazníka k jeho destinaci. Pokud je výsledná množina prázdná, nemůžeme daného chodce vybrat jako potencionálního zákazníka, protože by mu nebylo možné přiřadit destinaci. V opačném případě z výsledné množiny vybereme náhodnou destinaci a tu potencionálnímu zákazníkovi přiřadíme. Takový přístup je jednoduchý a dává destinace splňující všechny 4 uvedené podmínky.

Všemi uvedenými podmínkami dostaneme množinu chodců, kteří se mohou stát potencionálním zákazníkem. Jak z nich vybrat toho jednoho, kterého nabídneme hráči? Mohli bychom např. upřednostňovat chodce z určitých oblastí nebo pro každého chodce náhodně určit pravděpodobnost, s jakou by si zavolal taxi, kdyby měl příležitost (tedy něco jako jeho povahu). Tvrdíme, že neubereme na zábavnosti, pokud taková pravidla nepoužijeme a ze zmíněné množiny vybereme zcela náhodně. Zaprvé z chování chodců popsaného v sekci 3.3.1 plyne, že se chodci v různých oblastech vyskytují s různou hustotou. Okolo obytných oblastí s velkým množstvím domů a chodníků jich bude více, v ostatních oblastech méně. Přirozeně tak s větší pravděpodobností vybereme chodce např. z obytné oblasti,

což kopíruje chování lidí ve skutečném světě. Co se individuálních preferencí jednotlivých chodců týče, domníváme se, že chodců bude příliš mnoho na to, aby si o nich hráč něco pamatoval, takže individuální preferenci by neměly přidanou hodnotu.

3.3.3 Výpočet relativní polohy auta vůči silnici

Abychom mohli hráči zobrazit horizontální navigaci k zákazníkovi a následně k zákazníkově destinaci, potřebujeme vědět, na které silnici a kde přesně na této silnici se nachází hráčovo auto. Druhá část problému je jednoduchá – jakmile víme, na které silnici se auto nachází, máme přístup k jejímu tvaru reprezentovanému posloupností Bézierových křivek. Pak stačí nalézt ten bod na této posloupnosti, který je autu nejblíže. To provedeme již vyřešeným vygenerováním dostatečně mnoha uniformně rozmístěných bodů na jednotlivých křivkách a vybráním toho nejbližšího.

Zaměřme se nyní na první část problému, tedy jak zjistit, na které silnici se hráč nachází. Ukazuje se, že obdobný problém potřebujeme vyřešit i pro chodce. Potřebujeme totiž vědět, na kterém chodníku se chodec nachází, abychom v herní misi mohli vybrat některého z chodců, kteří jsou vedle silnice (viz sekce 3.3.2). Vyřešme tedy zjišťování polohy objektů vůči silnici nebo chodníku obecně.

Uvážili jsme následující možná řešení:

1. Využít navigační systém Unity zmíněný v sekci 3.3.1, resp. přidaný balíček `AI Navigation`. Každý agent ovládaný tímto systémem (např. chodec) má přiřazenou instanci komponenty `NavMeshAgent`. Ta nabízí veřejné pole `NavMeshAgent.navMeshOwner` obsahující instanci komponenty `NavMeshSurface`, která reprezentuje oblast navigačního meshe, na které se agent právě nachází. Pokud bychom komponentu `NavMeshSurface` přiřadili každé silnici a chodníku ve hře, mohli bychom takto rozhodnout, na které silnici či chodníku se agent nachází. Tento přístup má ale mnoho nevýhod. Původně jsme nepotřebovali, aby silnice měly komponentu `NavMeshSurface` nebo aby hráčovo auto mělo komponentu `NavMeshAgent`. Pokud bychom tyto komponenty uměle přidali, zvýšilo by to naši závislost na navigačním systému. Navíc není jasné, jak se vypořádat s tím, že by auto bylo ovládané jak navigačním systémem, tak naší vlastní herní logikou. Oficiální dokumentace `NavMeshAgent.navMeshOwner` [27] je totiž velmi stručná a na tyto otázky neodpovídá. Toto řešení proto považujeme za riskantní a zbytečně komplikované.

2. Využít detekce kolizí poskytnuté enginem Unity. Silnice, chodníky, auto i chodci určitě potřebují mít přiřazené komponenty typu `Collider`, aby u nich byly simulovány realistické kolize. Stačí tedy poslouchat události `OnCollisionEnter` a `OnCollisionExit` daných komponent typu `Collider`. Tyto události dostanou na vstup objekt účastnící se kolize, takže je jednoduché kontrolovat, zda auto narazilo do silnice a případně do které. Dalo by se očekávat, že při přejezdu z jedné silnice na druhou se vyvolá událost `OnCollisionEnter` na nové silnici a událost `OnCollisionExit` na staré silnici, což by řešilo náš problém. V praxi takto ovšem Unity nefunguje – v průběhu pobytu auta na jedné silnici se obě tyto události vyvolávají

pravidelně v nedeterministických intervalech.

3. V minulém bodě jsme narazili na to, že jsme chtěli komponentu `Collider` použít pro detekci dotyku místo detekce kolizí. Jak uvedeme v sekci 4.1.2, lepším přístupem v této situaci je vytvořit pro auto nebo chodce druhý `Collider`, který ale označíme jako `trigger`. To znamená, že sice bude generovat události při kolizích, ale nebude se účastnit fyzikální simulace. Tento `Collider` si tak můžeme dovolit posunout vůči objektu auta (resp. chodce) trochu směrem dolů, aby při kontaktu se silnicí (resp. chodníkem) bezpečně protínal `Collider` silnice (resp. chodníku). Poslouchat pak budeme události `OnTriggerEnter` a `OnTriggerExit`. V dokumentaci Unity ani na webových fórech jsme neobjevili důvod, proč by tento přístup neměl spolehlivě fungovat. Nicméně experimentálně se ukázalo, že v průběhu průjezdu např. auta po silnici se události opět vyvolávají mnohokrát za vteřinu. Domníváme se, že jde o nedostatek enginu Unity způsobený tím, že `Collider` auta je relativně malý a `Collider` silnice relativně komplikovaný.

4. Dalším přístupem je využít Unity funkci `Physics.Raycast`. Ta vyšle paprsek (tj. polopřímku) z určeného místa určeným směrem a nahlásí první objekt, do kterého narazí. Řešením by tedy bylo ptát se v pravidelných intervalech co se nachází pod autem (resp. chodcem) a kontrolovat, zda je to silnice (resp. chodník). To ovšem není robustní řešení – pokud bude např. auto jen z poloviny na dané silnici, paprsek tuto silnici může minout. To by šlo vyřešit vysíláním několik paprsků místo jednoho – třeba jeden v každém rohu auta nebo chodce.

5. Robustnějším vylepšením předchozího řešením je místo `Physics.Raycast` použít `Physics.BoxCast`, který místo jednoho paprsku „vysílá“ celý hranol. Šířku hranolu tak můžeme nastavit na šířku auta nebo chodce. Nepříjemné je, že `BoxCast` (stejně jako `Raycast`) musíme vyslat tak, aby nenarazil do samotného auta. Je tedy třeba vyslat jej přesně v prostoru mezi autem a silnicí. Jako jednodušší řešení nám přijde použít raději metodu `Physics.OverlapBox`, které zadáme krychli v prostoru a zpět dostaneme seznam objektů v něm se nacházejících. Jednoduše a robustně pak zjistíme, zda některý z objektů je silnice (resp. chodník).

Poslední řešení využívající `Physics.OverlapBox` funguje i v praxi bezchybně. Doplňme, že ve skutečnosti použijeme metodu `Physics.OverlapBoxNonAlloc`, který místo vytvoření nového pole pro uložené výsledků bere pole jako argument. Tím pádem můžeme pole pro uložení výsledku využívat vícekrát, čímž zmírníme tlak na Garbage Collector – tato operace bude totiž volána v krátkých intervalech pro každého chodce, kterých může být hodně. Nepříjemné je, že oficiální dokumentace metody `Physics.OverlapBoxNonAlloc` [28] nezmiňuje, co se stane, pokud se nalezené objekty do předaného pole nevejdou. Podle našich experimentů je však tato situace ošetřena a k chybě nedojde.

3.4 Rozdělení funkčnosti do modulů

V předcházejících sekcích jsme identifikovali 3 zásadní funkce, které bude muset objekt reprezentující silnici nebo chodník podporovat:

1. Uchovávání kontrolních bodů posloupnosti Bézierových křivek a práce s těmito křivkami.
2. Spojování s dalšími síťovými prvky.
3. Tvorba meshe pro vykreslování a detekci kolizí. S tím souvisí uchovávání šířky a výšky silnice (resp. chodníku).

Tyto 3 uvedené funkce jsou poměrně nezávislé. Konkrétně jediné závislosti jsou tyto:

1. Pro spojování s ostatními síťovými prvky je potřeba znát polohu kotev Bézierových křivek, protože ty odpovídají kotvám síťového objektu.
2. Pro tvorbu meshe je potřeba znát tvar silnice (resp. chodníku) a informace o jejím napojení na ostatní silnice (resp. chodníky) – výsledný mesh totiž musí brát v potaz tvar přiléhajících křižovatek.

Je tedy jednoduché tyto tři funkce oddělit do samostatných modulů. To uděláme ze dvou důvodů. Zaprvé vyšší modularita zvýší přehlednost výsledné implementace (využíváme Single Responsibility Principle). Zadruhé tím umožníme použít moduly i v jiném kontextu – Bézierovými křivkami bude v budoucnu vhodné popisovat horizontální dopravní značky, které ale nebudeme chtít spojovat do sítí. Podobně budovy chceme napojovat na síť chodníků, ale Bézierovy křivky ani tvorbu meshe k tomu nepotřebujeme. Dalším příkladem jsou chodníky vedoucí podél silnice, které samotné žadný tvar neuchovávají (jejich tvar je dán tvarem silnice), ale lze je napojit na ostatní chodníky a jejich mesh je stejný jako mesh jiných chodníků.

Konkrétně každý z těchto 3 modulů umístíme do samostatné Unity komponenty. Tím jasně vymezíme hranice mezi moduly a zároveň umožníme modulům komunikovat mezi sebou pomocí Unity metody `GetComponent<T>()`. Také to umožní jednotlivé komponenty v případě potřeby vypínat a zapínat – např. pokud by generování meshe bylo příliš časově náročné, může jej herní návrhář dočasně vypnout a pracovat pouze se sítí Bézierových křivek bez samotných meshů (více v sekci 7.10). Komponenty pak budeme přiřazovat herním objektům, takže např. objekt silnice bude mít přiřazeny 3 komponenty zajišťující všechny 3 uvedené funkcionality. Naopak objekt budovy bude mít přiřazenou pouze komponentu pro spojování s ostatními síťovými prvky.

3.5 Rozšíření Unity editoru

Unity Editor podporuje manipulaci s herními objekty ve 3D herní scéně (posouvání, rotaci, mazání, apod.) i úpravu jejich serializovaných parametrů. Navíc je velmi jednoduché do Unity Editoru přidávat nové funkce, což konkrétně ukážeme v sekci 4.3. Proto místo tvorby zcela nového editoru herního světa pouze přidáme funkce do již existujícího Unity Editoru. Zejména přidáme následující funkce:

- Podporu pro vytváření silnic a chodníků a manipulaci s nimi. Manipulací myslíme práci s jejich tvarem daným posloupností Bézierových křivek (posouvání, přidávání a mazání kontrolních bodů, rozdělení segmentu na dva, apod.) a jejich spojování do křižovatek.

- Přidávání chodníků podél již existující silnice.
- Napojování chodníků k budovám.
- Funkce pro výpis ladících informací o síti silnic a chodníků.

4. Herní engine Unity

V této sekci nastíníme základní vlastnosti herního enginu Unity 2021.2, které jsou potřeba k porozumění implementaci projektu. Zájemce o podrobnější porozumění odkazujeme na oficiální uživatelský manuál Unity [29] a vývojářskou příručku Unity [28].

4.1 Základní popis

Vývoj v Unity probíhá ve speciálním Unity Editoru. Ten umožňuje pracovat se scénou, tedy herním světem. Scéna obsahuje herní objekty (v této sekci budeme psát jen „objekty“) uspořádané do hierarchické struktury. Objekt tedy může mít jako rodiče jiný objekt, potom o něm mluvíme jako o potomkovi.

Základním paradigmatem v Unity je komponentová architektura. Každý objekt může mít přiřazené libovolné množství komponent nezávisle na ostatních objektech. Komponenty lze upravovat, přidávat a odstraňovat v Unity Editoru i programově. V Unity Editoru k tomu slouží okno s názvem `Inspector`.

4.1.1 Komponenta `Transform`

Každý objekt má komponentu `Transform`, která obsahuje informace o jeho pozici, otočení a zvětšení. Tato komponenta může dále obsahovat referenci na komponentu `Transform` jiného objektu, který je pak chápán jako předek. Tak vznikne výše zmíněná hierarchická struktura objektů.

V souvislosti s hierarchickou strukturou je třeba rozlišovat lokální (local) a globální (world) souřadnice. Každý `Transform` obsahuje pouze svou lokální informaci o posunutí, otočení a zvětšení vztaženou na `Transform` svého předka. Reálná globální pozice, otočení a zvětšení objektu je vypočítána jako složení všech komponent `Transform` od daného objektu přes všechny jeho předky až ke kořenu hierarchie. Např. pneumatika, která je potomkem auta, obsahuje pouze informaci o své pozici relativně ke středu auta, tedy v lokálních souřadnicích. Automaticky se pak pohybuje a natáčí společně s autem.

Pro převody z lokálních do globálních souřadnic existují 3 metody:

1. `TransformPoint` zkonvertuje bod z lokálních souřadnic do globálních.
2. `TransformVector` zkonvertuje směrový vektor z lokálních souřadnic do globálních, nebere tedy v potaz lokální posunutí.
3. `TransformDirection` zkonvertuje směr z lokálních souřadnic do globálních, nebere tedy v potaz lokální posunutí ani lokální zvětšení.

Inverzními operacemi jsou `InverseTransformPoint`, `InverseTransformVector` a `InverseTransformDirection`.

4.1.2 Další důležité komponenty

Mezi další nejběžněji používané komponenty patří:

- `Rigidbody` – Simuluje chování fyzikálního objektu, obsahuje tedy vlastnosti jako hmotnost, koeficient odporu, rychlost, úhlovou rychlost, apod.
- `MeshFilter` – Obsahuje *mesh* určený pro renderování objektu.
- `MeshRenderer` – Zajišťuje samotné renderování *meshe* obsaženého v komponentě `MeshFilter`.
- `Collider` – Předek všech tzv. colliderů, tedy komponent, které kontrolují kolizi s ostatními objekty. Pokud chceme kolize pouze detekovat, ale nechceme, aby ovlivňovaly fyzikální simulaci, označíme daný collider jako *trigger* (to buď v Unity Editoru, nebo pomocí pole `Collider.isTrigger`).
- `MeshCollider` – Collider pro objekty, jehichž tvar je dán obecným *meshem*.
- `BoxCollider` – Collider pro objekty, jejichž tvar je krychle.
- `Camera` – Objekt s touto komponentou udává mimo jiné pozici a natočení kamery, která renderuje 3D scénu na 2D obrazovku.
- `Canvas` – Pomocí této komponenty vytváříme ovládací prvky ve hře.

4.2 Programování

Programování herní logiky probíhá v jazyce C#. Unity kód překládá kompilátorem Roslyn a podporuje tedy verzi jazyka C# 9.0. Nicméně některé vlastnosti jazyka nejsou podporovány a v případě jejich použití dojde ke kompilační chybě. Mezi ty nepodporované vlastnosti, které nám při vývoji scházely nejvíce, patří

- kovariantní návratové typy při překrývání metod v potomkovi,
- init-only settery.

Pro plný výčet chybějících vlastností odkazujeme na článek *C# compiler* v manuálu Unity [30].

Pojítkem mezi naším programem a herním světem je třída `MonoBehaviour`. Díky ní můžeme vytvořit zcela nový typ komponenty. Uděláme to tak, že vytvoříme potomka `MonoBehaviour` a umístíme jej do souboru se jménem odpovídajícím jménu třídy. Tato třída je pak plnohodnotnou komponentou a můžeme s ní tak pracovat, zejména ji můžeme přiřadit k libovolnému hernímu objektu.

Potomek `MonoBehaviour` může obsahovat speciální metody (event functions), které Unity rozpoznává a přiřazuje jim význam. Tyto mohou být definované s libovolným modifikátorem přístupu, jen je potřeba dodržet signaturu. Naprostá většina těchto metod nemá parametry ani návratovou hodnotu. Uveďme ty nejdůležitější, pro kompletní výčet s detailním vysvětlením viz článek *Order of execution for event functions* v manuálu Unity [31].

`OnEnable()`
Každá komponenta může být *enabled* nebo *disabled*, což může určovat herní návrhář v Unity Editoru nebo programátor pomocí příslušných metod. Při přechodu

ze stavu *disabled* do stavu *enabled* a také při vytváření objektu s touto komponentou se na komponentě volá metoda `OnEnable`.

`Reset()`
Tato metoda se volá jednak při přidání komponenty k objektu a jednak když uživatel Unity Editoru stiskne na dané komponentě tlačítko *Reset*.

`OnValidate()`
Tato metoda se volá pokaždé, když je komponenta deserializována, nebo je změněná nějaká její serializovaná hodnota (více o serializaci v sekci 4.2.2).

`Update()`
Objekt může být *aktivovaný* (*active*) nebo *deaktivovaný* (*inactive*). Pokud je aktivovaný, volá se před vykreslením každého snímku metoda `Update()` na všech jeho *enabled* komponentách.

`LateUpdate()`
Tato metoda se stejně jako `Update` volá jednou za snímek na každé *enabled* komponentě aktivovaného objektu. Je ovšem zaručeno, že v rámci jednoho snímku všechna volání `LateUpdate` nastanou až po všech voláních `Update`. Typický případ užití této metody je pohyb kamery sledující hráče, která by měla reagovat na nejnovější polohu hráče.

`FixedUpdate()`
Tato metoda se volá na *enabled* komponentách aktivovaných objektů v přesně daných intervalech. To je v protikladu s `Update` a `LateUpdate`, u kterých není frekvence volání nijak zaručena. `FixedUpdate` je užitečná zejména pro fyzikální simulace, které mohou záviset na tom, že prodleva mezi dvěma voláními není příliš velká.

`Start()`
Tato metoda se volá na každé *enabled* komponentě před prvním voláním metody *Update*.

`OnDisable()`
Tato metoda se volá při přechodu komponenty ze stavu *enabled* do stavu *disabled* a také při zničení objektu s danou komponentou.

`OnDestroy()`
Tato metoda se volá před vykreslením v tom snímku, ve kterém objekt přestane existovat. To nastane voláním metody `UnityEngine.Object.Destroy` nebo zavřením scény.

4.2.1 Prefabs

Prefabs jsou šablony, ze kterých můžeme vytvářet herní objekty. Můžeme např. vytvořit prefab pneumatiky a poté jej čtyřikrát instanciovat v herní scéně. To má 2 výhody – zaprvé nemusíme vícekrát nastavovat parametry objektů, stačí

je nastavit jednou v prefabu. Zadruhé prefaby zůstanou provázané s vytvořenými instancemi a jakékoli změny v prefabu se projeví ve všech instancích. Naopak v každé instanci můžeme nastavení z prefabu překrýt vlastním nastavením, což se v ostatních instancích ani v prefabu neprojeví. Tím prefaby připomínají dědičnost v objektově orientovaném programování.

4.2.2 Serializace

Serializace je proces transformace herního světa do formátu, který lze uložit na disk (např. do textového formátu YAML nebo do binárního formátu). Její porozumění je nutné pro vývoj v Unity.

Unity spouští serializaci automaticky pokaždé, když v Unity Editoru uživatel provede nějakou změnu, jako např. přidání nového objektu nebo změnu hodnoty v některé komponentě. Díky tomu je při spuštění hry načten stav herního světa přesně tak, jak byl v Unity Editoru vytvořen – serializovaný stav se v tu chvíli deserializuje.

Do serializace je zahrnut každý objekt v herní scéně a všechna jeho pole (fields), jejichž datový typ je jedním z následujících:

- `UnityEngine.Object` a jeho potomci (tj. např. `GameObject`, `Component`, `Texture`, apod.)
- libovolná nestatická třída nebo struktura, pokud jsou označené atributem `Serializable`
- primitivní datové typy, výčtové typy
- pole některého z uvedených typů (nicméně pole polí, tj. *jagged array*, ani *multidimensional array* podporováno není)
- `List<T>`, kde `T` je některý z uvedených typů
- některé typy z enginu Unity, jako např. `Vector3`, `Rect`, `AnimationCurve`, a další
- další typy (viz manuál Unity [32])

Navíc aby bylo pole (field) objektu serializováno, musí splňovat následující:

- být veřejné (`public`), nebo mít atribut `[SerializeField]`
- nebýt statické nebo konstantní (`const`)
- nebýt `readonly`

Místo vlastností (properties) objektů se serializují přímo jejich *backing fields*. Pokud chceme, aby se hodnota vlastnosti serializovala, musí tedy daná vlastnost mít *setter*. Dále pokud je *backing field* automaticky vygenerovaný (*auto-implemented property*), je pro serializaci potřeba vlastnost explicitně označit pomocí `[field: SerializeField]`.

Pokud je serializovaným typem třída nebo struktura, provede se její serializace rekurzivně. Je třeba dbát na to, že výchozí způsob serializace referenčních datových typů je *inline*, tj. neserializuje se reference, ale přímo samotná hodnota. To znamená, že:

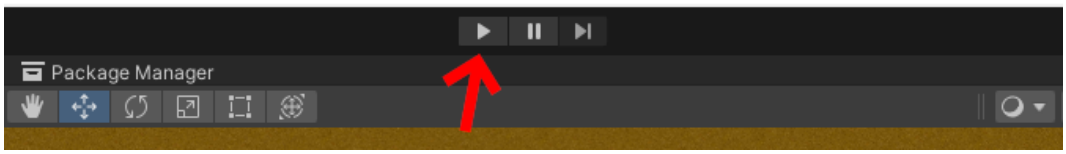
- Pokud dvě serializované reference vedly na tentýž objekt, po deserializaci bude každá ukazovat na jiný objekt.
- Pokud serializujeme cyklickou datovou strukturu, dojde k nekonečnému cyklu (který bude v určité hloubce automaticky detekován a zastaven).
- Nelze serializovat hodnotu `null`. Bohužel pokus o serializaci reference s hodnotou `null` nezpůsobí chybu, ale místo `null` se dosadí nový objekt daného typu s defaultními hodnotami polí.
- Pokud je typ odkazovaného objektu potomkem typu reference (tj. pokud využíváme polymorfismus), serializuje se pouze ta část objektu, která odpovídá typu reference.
- a další (viz manuál Unity [32])

Pokud je cílem serializovat opravdu referenci místo hodnoty, musí být dané pole označeno atributem `[SerializeReference]`. To vyřeší všechny zmíněné problémy, ale vede k vyšší časové a paměťové náročnosti serializace (viz dokumentace atributu `[SerializeReference]` [33]).

U tříd označených atributem `[Serializable]` je třeba dbát na to, že pokud obsahují konstruktor bez parametrů, použije ho Unity při deserializaci pro vytvoření deserializované instance. To znamená, že není možné v bezparametrickém konstruktoru provádět operace, které by měly proběhnout jen jednou (např. generování GUID). Tento problém řešíme tak, že bezparametrickým konstruktorům serializovatlných tříd přidáme uměle *dummy* parametr, který k ničemu nepoužijeme (je také vhodné vytvořit poté tovární metodu a konstruktor nemít veřejný, aby tento implementační detail nebyl součástí rozhraní).

4.2.3 Play Mode

Po otevření editoru je možné hru spustit v tzv. Play módu, tedy bez nustnosti provést plné sestavení aplikace. Přechod do Play módu se provede stiskem tlačíka šipky na liště v horní části okna, jak je ukázáno na obrázku 4.1. Pro přechod zpět do tzv. Edit módu stačí znovu stisknout stejné tlačítko.



Obrázek 4.1: Tlačítko pro přechod do Play módu.

4.3 Rozšiřování Unity Editoru

Unity nabízí širokou škálu možností, jak rozšířit Unity Editor podle našich představ – pro jejich kompletní popis viz sekce Extending the Editor manuálu Unity [34]. My využijeme pouze následující způsoby rozšíření Unity Editoru:

- Přidávání nových klávesových zkratek. K tomu stačí označit statickou metodu atributem `[Shortcut]` a do jeho argumentů vyplnit požadovanou kombinaci kláves. Po stisku této kombinace klávest bude zavolána příslušná statická metoda.

- Přidávání nových položek do horní lišty v Unity Editoru (té lišty, která obsahuje nabídky jako *File* nebo *Edit*. K tomu stačí označit statickou metodu atributem `[MenuItem]` a do jeho argumentu vyplnit název nové položky. Po stisku této položky bude zavolána příslušná statická metoda.

- Vytváření nástrojů pro usnadnění práce s určitým typem komponent, tedy vytváření tzv. *custom editors*. To vysvětleme na konkrétním příkladu – při práci s objektem obsahujícím komponentu `BezierSpline` (ta představuje posloupnost Bézierových křivek, viz sekce 5.4.2) můžeme chtít, aby se v herním světě vykreslily příslušné křivky spolu s kontrolními body a aby herní návrhář mohl body posouvat. Za tímto účelem vytvoříme třídu `BezierSplineEditor` dědící od třídy `Editor`, kterou navíc označíme atributem `[CustomEditor]` následujícím způsobem:

  ```
  [CustomEditor(typeof(BezierSpline))]
  public class BezierSplineEditor : Editor
  {
      // Custom editor logic.
  }
  ```

  Ve třídě `BezierSplineEditor` pak definujeme metodu `OnSceneGUI`, která se bude opakovaně volat pokud je v Unity Editoru označen objekt s komponentou `BezierSpline`. Tato metoda tak může zajistit vykreslování Bézierových křivek a kontrolních bodů, se kterými může herní návrhář manipulovat – pro obojí lze využít statické metody na třídě `Handles`, které nabízejí jak vykreslování, tak práci s ovládacími prvky.

  Pokud custom editor pracuje s herními skripty, může být vhodné tyto skripty označit atributem `[ExecuteAlways]`. V takovém případě se pak speciální metody daného skriptu (jako např. `OnEnable`, `OnDisable` nebo `Update`) volají nejenom za běhu hry, ale i v editoru v Edit módu.

5. Vývojářská dokumentace

Podařilo se nám implementovat všechny požadavky vytyčené v kapitole 2 s použitím vysokoúrovňových rozhodnutí provedených v kapitole 3. Cílem této kapitoly je uvést architekturu a implementační detaily výsledného díla.

5.1 Nástroje

Projekt používá Unity 2021.2.7f1 a jazyk C# (pro podrobnosti o integraci .NET v Unity viz sekce 4.2). Unity lze získat na oficiálním webu Unity [35], přičemž doporučujeme instalaci přes program Unity Hub, který zjednodušuje správu různých verzí Unity. Pro umožnění překladu pro systém Android je potřeba tuto při instalaci explicitně vybrat (případně to lze následně udělat v nastavení v Unity Editoru). Dále je pro používání Unity vyžadován uživatelský účet u firmy Unity (tzv. Unity ID), jehož založení je pro nekomerční účely zdarma.

Unity je dostupné pro operační systémy Windows, Mac OS a některé distribuce Linuxu. Z autorovy zkušenosti ovšem Unity Editor 2021 na distribuci Ubuntu obsahuje velké množství chyb, které značně ztěžují práci, a doporučujeme tedy zvolit pro vývoj jiný operační systém.

Projekt dále využívá následující 2 balíčky, které nejsou součástí standardní instalace Unity. Oba jsou vyvíjeny společností Unity.

- `AI Navigation` (`com.unity.ai.navigation`) – viz sekce 3.3.1.
- `TextMesh Pro` (`com.unity.textmeshpro`) – oficiálně doporučený balíček pro pokročilou tvorbu textů ve hře a v herním menu (viz manuál Unity [36]).

Pro vývoj kódu v jazyce C# jsme použili Visual Studio 2019 od firmy Microsoft. Pro detailní vysvětlení procesu propojení Visual Studia s Unity odkazujeme na oficiální návod firmy Microsoft [37].

5.2 Struktura projektu

Výsledný projekt je umístěn v příloze A v adresáři `Traffic`. Struktura projektu je založena na oddělení souborů podle jejich typu. Dále při pojmenovávání adresářů respektuje některá speciální jména, která Unity rozpoznává a přiřazuje jim význam – to se týká adresářů `Assets`, `Assets/Editor` a `Assets/Resources` (viz sekce *Special folder names* manuálu Unity [38]).

Všechny ručně vytvořené soubory se nachází v adresáři `Assets`. Ten obsahuje následující podadresáře:

- `Scripts` – Skripty v jazyce C# ovládající herní svět a herní mechanismy.
- `Editor` – Skripty v jazyce C# rozšiřující editor Unity tak, aby mohl být používán jako editor herního světa.
- `Scenes` – Popisy jednotlivých scén hry. Scénami jsou úvodního menu a samotná hra. Popis herní scény zahrnuje i předpočítaný navigační mesh.

- `Resources`
  - `Resources/Images` – Obrázky pro herní menu a ovládací prvky ve hře.
  - `Resources/Materials` – Unity Materials (druhy povrchů), které určují způsob renderování objektů, např. silnic nebo země (viz sekce Materials manuálu Unity [39]).
  - `Resources/Physic materials` – Unity Physic Materials (druhy povrchů z fyzikálního hlediska) určující fyzikální chování objektů, např. jejich statické a dynamické tření nebo odrazovost (viz sekce Physic Material component reference manuálu Unity [40]).
  - `Resources/Prefabs` – Prefabs, jako např. auto nebo pneumatika (pro vysvětlení konceptu Prefabs viz sekce 4.2.1).
  - `Resources/Textures` – Textury objektů.
  - `Resources/Sounds` – Hudba a zvukové efekty.
- `TextMesh Pro` – Soubory nestandardního balíčku `TextMesh Pro` uvedeného v sekci 5.1.

5.2.1 Sestavení a spuštění projektu

Pro instalaci hry na mobilní zařízení se systémem Android je v příloze A v adresáři `Build` připravený balíček `traffic.apk`. Pro vlastní sestavení aplikace nebo spuštění herního editoru je potřeba otevřít projekt v Unity Editoru nebo v Unity Hub (viz sekce o nástrojích 5.1).

Pro sestavení aplikace stačí v Unity Editoru otevřít nabídku `File > Build Settings` (případně pomocí klávesové zkratky `Ctrl+Shift+B`) a stisknout tlačítko `Build`. Před samotným sestavením je možné v otevřené nabídce upravit parametry sestavení, jako např. cílovou platformu.

Co se týče kompilace skriptů v jazyce C#, je rozložení zkompilovaného kódu do .NET Assemblies řízeno výchozím nastavením, kdy Unity automaticky vytvoří dva C# Project File (.csproj) – jeden pro kód z adresáře `Editor` a druhý pro veškerý ostatní kód z adresáře `Assets`. Z těchto pak vzniknou dvě assembly – `Assembly-CSharp.dll` a `Assembly-CSharp-Editor.dll`, které pak Unity používá. Navíc z assembly `Assembly-CSharp-Editor.dll` je automaticky zavedena reference na assembly `Assembly-CSharp.dll`, takže kód rozšiřující Unity editor může využívat herní skripty, což v našem případě extenzivně využíváme (opačná reference ale neexistuje). Pro více informací o těchto výchozích assemblies viz sekce Predefined assemblies manuálu Unity [41].

V otevřeném Unity Editoru je možné hru spouštět i bez kompletního sestavení, které může trvat i několik minut. Konkrétně lze hru spustit v tzv. Play Mode jediným stiskem tlačítka (viz sekce 4.2.3), což tvrá jen několik sekund. Je to tedy preferovaná varianta pro časté zkoušení hry v průběhu vývoje. V Play Mode je zároveň možné ovládat hru přes připojený mobilní telefon se systémem Android nebo iOS – stačí na mobilním telefonu nainstalovat a nakonfigurovat aplikaci Unity Remote podle oficiálního manuálu Unity [42]. Princip je takový, že Unity Editor posílá do aplikace Unity Remote snímky obrazovky a zpět získává informace o uživatelském vstupu (tj. v našem případě o dotyku na obrazovce).

Tento postup nicméně nenabízí plnohodnotný hráčský zážitek, protože snímky obrazovky jsou do mobilního telefonu posílány se sníženou frekvencí oproti standardnímu běhu hry, což se projeví menší plynulostí a kvalitou obrazu.

Dále může otevřený Unity Editor sloužit jako editor herního světa. Stačí přejít do adresáře `Assets/Scenes` a otevřít některou ze scén, jejíž název končí slovem *Level*, např. scénu `DefaultLevel`. Jak s herním editorem pracovat uvedeme v kapitole 7.

5.3 Struktura herní scény

Všechny prvky herního světa, které popíšeme v následujících sekcích 5.4 a 5.5, jsou uspořádány do herní scény reprezentující herní svět. Příkladem takové scény je scéna `DefaultLevel` v adresáři `Assets/Scenes`. Informace o správě a vytváření dalších scén uvedeme v sekci 7.1. V této kapitole uvedeme jejich strukturu, kterou všechny scény sdílejí.

Konkrétně uveďme hlavní herní objekty, ze kterých se herní scéna skládá:

- `MainCamera` – objekt kamery s komponentou `Camera`, který určuje, odkud hráč scénu pozoruje. Dále má objekt komponentu `CameraFollow`, která mění polohu kamery v závislosti na poloze hráčova auta (viz sekce 5.5.2), a komponentu `AudioSource`, ze které je přehrávána hudba. Jako jediný objekt ve scéně má kamera komponentu `AudioListener`, takže její poloha určuje i to, jak se hráči přehrávají zvuky ve scéně (viz sekce 5.4.5).

- `Light` – objekt sdružující jako své potomky objekty s komponentou `Light`, jejichž poloha určuje nasvícení scény.

- `EventSystem` – objekt s komponentami `Event System` a `Standalone Input Module`. Ty umožňují ve scéně používat prvky uživatelského rozhraní a zpracovávat uživatelský vstup. Pro více informací o těchto komponentách odkazujeme na dokumentaci balíčku *Unity UI* [43].

- `Controls` – objekt s komponentou `Canvas` sdružující prvky uživatelského rozhraní, tj. ovládací prvky uvedené v sekci 5.5.2 a dialogová okna herní mise uvedená v sekci 5.5.3.

- `Car` – objekt reprezentující hráčovo auto. Pro účely fyzikální simulace auta obsahuje komponenty `Rigidbody` a `BoxCollider`. Dále obsahuje komponentu `AudioSource`, ze které se přehrávají zvuky motoru. Samotnou fyzikální simulaci a ovládání auta zajišťuje komponenta `Car` (viz sekce 5.5.1).

- `Box` – prázdný objekt, který jako své potomky sdružuje zem, na které je herní svět umístěn a stěny zabraňující vyjetí hráče mimo zem.

- `NetworkElements` – objekt obsahující jako své potomky všechny prvky silniční sítě, sítě chodníků a budov. Obsahuje komponenty všech skupin holderů, jako např. `RoadGroup`, `SidewalkGroup`, `PersonHubGroup`, a další (pro vysvětlení konceptu skupiny holderů viz sekce 5.4.1). Dále obsahuje komponentu `NavMeshSurface`, která zajišťuje výpočet navigačního meshe na chodnících (viz sekce 5.4.6).

- `GameManager` – objekt s komponentou `GameManager`, která po spuštění hry vytvoří herní misi a spustí hudbu.
- `PersistenceManager` – objekt s komponentou `PersistenceManager`, která při spuštění hry načte její uložený stav a při ukončení hry její aktuální stav uloží, viz sekce 5.6.

5.4 Implementace herního světa

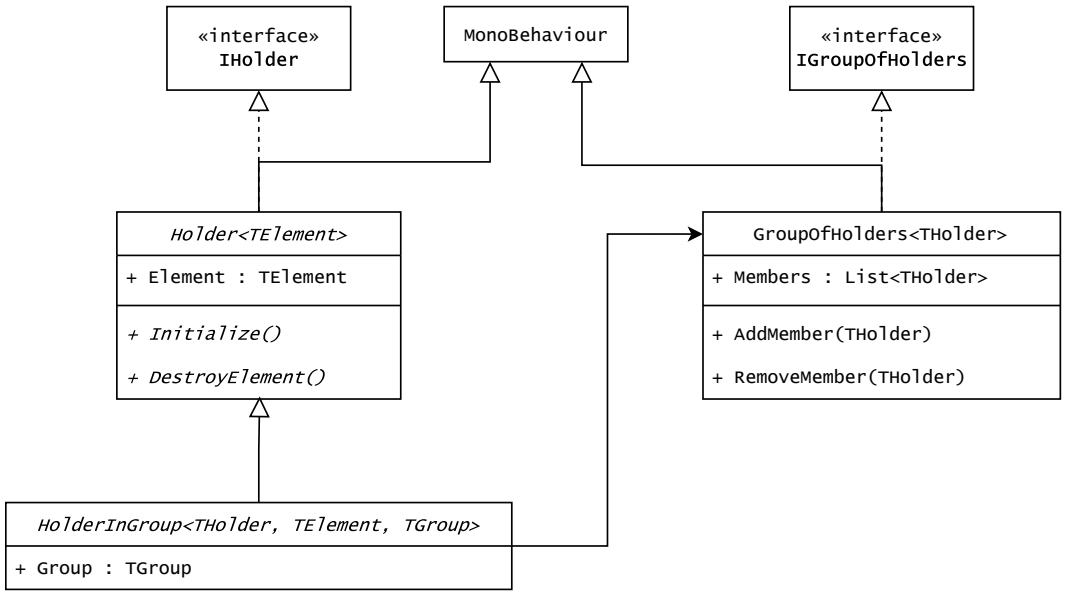
Kód ovládající herní svět (tedy silnice, chodníky, budovy a chodce) je soustředěn v adresáři `Scripts/World`. Ten obsahuje následující podadresáře, kde každý z nich představuje určitou funkcionalitu:

- `Splines` – zajišťuje uchovávání a manipulaci s posloupnostmi Bézierových křivek.
- `Network` – zajišťuje spojování síťových prvků do sítí, jak je to popsáno v sekci 3.2.1. Dále zajišťuje hledání nejkratších cest ve vzniklých sítích.
- `Rendering` – zajišťuje tvorbu meshů a UV souřadnic pro silnice a chodníky, jak je to popsáno v sekcích 3.2.3, 3.2.4, 3.2.5, 3.2.6 a 3.2.7. Dále zajišťuje renderování horizontální navigace na silnici.
- `Audio` – zajišťuje ovládání hudby a zvukových efektů.
- `People` – zajišťuje generování chodců a jejich pohyb po chodnících.
- `Holders` – shromažďuje všechny potomky třídy `MonoBehaviour` z adresáře `Scripts/World`. Tím pomáhá oddělit kód nesouvisející s Unity s kódem, který je ovládán Unity. Dále zajišťuje shromažďování holderů jednotlivých typů do skupin.

Nejprve popíšeme význam tříd v podadresáři `Holders`. Zbylé podadresáře na sobě striktně závisí jenom tak, že později uvedený závisí na dříve uvedeném. To nám umožňuje i tyto podadresáře popsat zvlášť v odpovídajících sekcích.

5.4.1 Modul `Holders`

Pro přehlednost jsme co nejjasněji definovali hranici mezi kódem nezávislým na Unity („business logikou“) a třídami dědícími od `MonoBehaviour`, jejichž životní cyklus ovládá Unity. Za tímto účelem jsme zavedli koncept tzv. *holderu*, kdy holder určité třídy je potomek `MonoBehaviour` obsahující serializovatelnou instanci dané třídy – té v tomto kontextu říkáme *element*. Např. holder třídy `Road` (tedy `RoadHolder`) obsahuje instanci třídy `Road`. V našem programu všechny holdery dědí od abstraktní třídy `Holder` (v celé sekci sledujme diagram 5.1), která je potomkem třídy `MonoBehaviour` a obsahuje vlastnost `Element`. Tím pádem je každý holder Unity komponentou a může tedy být přiřazen hernímu objektu. Ve výsledku tak např. každý herní objekt reprezentující silnici obsahuje komponentu `RoadHolder`, která obsahuje datonosnou třídu `Road`, přičemž ta nese informace o tvaru silnice, šířce silnice, apod.



Obrázek 5.1: Diagram nejdůležitějších tříd a rozhraní modulu `Holders` s vybranými vlastnostmi a metodami.

Třída `Holder` také zjednodušuje inicializaci a destrukci herních objektů tím, že implementuje Unity metody `OnEnable`, `Reset` a `OnDisable` a potomkům vystavuje už zjednodušené virtuální metody `Initialize` a `DestroyElement`. Při implementaci konkrétního holdera stačí překrýt některou nebo obě z těchto metod a není již potřeba přemýšlet o přesném fungování Unity.

Na konceptu holderu je postaven koncept skupin holderů. Jedná se o třídy implementující `IGroupOfHolders` a dědící z `MonoBehaviour`, které obsahují seznam nějakého typu holderů. Každá skupina holderů je tedy komponentou a pro správné fungování musí být přiřazena nějakému hernímu objektu.

Shromažďování holderů do skupin konkrétně funguje následovně. Pokud některý holder dědí z `HolderInGroup` (což je potomek `Holder`) a v Unity Editoru mu nebyla explicitně přiřazena skupina (tj. odpovídající potomek třídy `GroupOfHolders`), pokusí se při inicializaci tuto skupinu vyhledat v některém z rodičovských herních objektů. Např. `RoadHolder` se pokusí vyhledat `RoadGroup`. V případě nalezení sám sebe do této skupiny přidá. V opačném případě je aktuální chování takové, že se nahlásí chyba, aby se předešlo tichým selháním. Podobně při destrukci daný holder sám sebe ze skupiny odstraní.

Tím pádem má zbylý kód pro každý typ holderu k dispozici všechny načtené holdery tohoto typu. Např. seznam všech načtených síťových prvků je potřeba pro deserializaci spojení, jak je popsáno v sekci 3.2.2, podobně seznam všech chodců je potřeba při vybírání zákazníka v herní misi. Dále seznam všech silnic a chodníků je potřeba při renderování křižovatek.

Polymorfismus a CRTP

Při tvorbě obecného předka pro třídy, které obsahují nějaký serializovatelný typ, jsme narazili na problém, že výchozí serializace nepodporuje polymorfismus (viz sekce 4.2.2). Ve třídě `Holder` tedy není možné jako datový typ vlastnosti `Element` použít nějakého obecného předka všech elementů, jako např. `Sys-`

tem.Object.

Možným řešením by bylo definovat Holder.Element jako abstraktní vlastnost a v každém konkrétním holderu definovat novou vlastnost s konkrétním typem elementu. Uveďme, jak by takový přístup vypadal na zjednodušeném příladu RoadHolder.

```
abstract class Holder {
    public abstract object Element { get; protected set; }
    ...
}

class RoadHolder : Holder {
    public override object Element {
        get => Road;
        protected set => Road = (Road)value;
    }
    [field: SerializeField]
    public Road Road { get; private set; }
    ...
}
```

Tento přístup funguje, protože se serializuje konkrétní typ. Zároveň ačkoli dochází k přetypování, není pro uživatele těchto tříd nebezpečné, protože setter vlastnosti Element není veřejný. Zásadní nevýhodu ovšem vidíme v netriviální duplikaci kódu, která zvyšuje riziko programátorské chyby při tvorbě nového holdera. Snadno se totiž zapomene např. na atribut [SerializeField], bez kterého se vlastnost Road neserializuje a při deserializaci se v ní objeví hodnota null (bez jakékoli nahlášené chyby nebo warningu).

Místo toho jsme použili jiný přístup. Vycházeli jsme z toho, že Holder musí „vědět“ o datovém typu konkrétního elementu. To vynucuje přístup podobný idiomu *Curiously recurring template pattern* (často zkracované na CRTP, viz článek *Curiously recurring template pattern* na Wikipedii [44]), ve kterém je předek generický a potomek do generického parametru předá vlastní typ. V našem případu ale konkrétní holder nebude předávat vlastní typ, nýbrž typ svého elementu. Výše uvedený příklad bude pak s použitím CRTP vypadat takto:

```
abstract class Holder<TElement> {
    [field: SerializeField]
    public TElement Element { get; protected set; }
    ...
}

class RoadHolder : Holder<Road> {
    ...
}
```

Nevýhodou tohoto přístupu je, že každý holder má jiného předka a s různými holdery tedy není možné pracovat polymorfně. To jsme vyřešili zavedením rozhraní IHolder, které implementuje třída Holder. IHolder pak umožňuje přístup k elementu *read-only* vlastností ElementObject typu object. Taková rozhraní

bylo potřeba zavádět i pro některé abstraktní potomky třídy `Holder`, kteří sami slouží jako předci pro konkrétní holdery. To je příklad `INetworkElementHolder` a `ILinearNetworkElementHolder`.

Z analogických důvodů využívá CRTP koncept skupin holderů, tedy třídy `GroupOfHolders` a `HolderInGroup`. Ve zbytku kapitoly už pro přehlednost generické parametry vztahující se k CRTP nebudeme uvádět.

5.4.2 Modul `Splines`

Podadresář `Splines` obsahuje logiku pro práci s kubickými Bézierovými křivkami a jejich posloupnostmi. Samotné matematické výpočty se nachází ve formě statických metod ve statické třídě `BezierUtils`. Konkrétně se jedná o následující operace. První tři z nich podrobněji popíšeme v následující podsekci, čtvrtou jsme podrobně popsali v sekci 3.2.3.

- Interpolace na kubické Bézierově křivce dané kontrolními body pomocí De Casteljauova algoritmu.
- Počítání tečného směru v určitém bodě kubické Bézierovy křivky.
- Odhad délky kubické Bézierovy křivky.
- Posunutí posloupnosti kubických Bézierových křivek o určitý *offset* (tj. proces tvorby krajnice silnice z křivky procházející středem silnice).

Posloupnost na sebe navazujících kubických Bézierových křivek je pak reprezentována třídou `BezierSpline`. V kontextu této třídy uveďme nejprve nejdůležitější *read-only* struktury rozšiřující typový systém, které používáme pro zesílení statické kontroly kódu:

- `AnchorNumber` – pořadové číslo kotvy v posloupnosti Bézierových křivek (obálka nad typem `System.Guid`)
- `PointIdx` – pořadové číslo kontrolního bodu v posloupnosti Bézierových křivek (obálka nad typem `System.Guid`)
- `SplineSegmentView` – reprezentuje jednu křivku v posloupnosti kubických Bézierových křivek (obsahuje referenci na `BezierSpline` a pořadové číslo kotvy, kterým začíná čtveřice kontrolních bodů dané křivky)

Samotná třída `BezierSpline` obsahuje seznam kontrolních bodů všech na sebe navazujících Bézierových křivek v posloupnosti, přičemž koncový opěrný bod křivky splývá s počátačním opěrným bodem křivky další (viz terminologie zavedená v sekci 3.2). To znamená, že kontrolní body k-té křivky v posloupnosti (počítáno od 0) jsou v seznamu uloženy na indexech $3k$ až $3k + 3$ (včetně). Tím je zajištěno, že na sebe křivky vždy navazují. Nejdůležitější metody, které třída `BezierSpline` nabízí, jsou následující:

- `Interpolate` – podle vstupního argumentu `t` provede interpolaci pomocí De Casteljauova algoritmu přes všechny křivky v posloupnosti,

- `AppendSegment`, `PrependSegment` a `InsertSegment` – přidá křivku na konec posloupnosti, na začátek posloupnosti, nebo do určitého bodu v posloupnosti,
- `RemoveSegment` – odstraní danou křivku z posloupnosti,
- `SamplePoints` – vytvoří enumerátor procházející body nanesené na křivku, přičemž lze nastavit hustotu nanesení (ale mezery mezi jednotlivými sousedními body se mohou značně lišit),
- `UniformalizePoints` – vytvoří enumerátor procházející body nanesené na křivku, přičemž vzdálenost sousedních bodů je přibližně konstantní (použitý algoritmus jsme detailně popsali v sekci, 3.2.4)
- `PointClosestTo` – vrátí bod na některé z křivek, který je nejblíže určitému zvolenému bodu.

Bézierovy křivky

Uveďme stručný popis Bézierových křivek. Popíšeme je na na intuitivní úrovni, která dostačuje pro jejich implementaci, nikoli však pro jejich matematické studium. Pro důkladnější pojednání o tomto tématu doporučujeme článek A Primer on Bézier Curves [20]. Křivku chápeme jako zobrazení z intervalu $[0, 1]$ do eukleidovského prostoru – v našem případě do třídimenzionálního prostoru. Bézierova křivka řádu k je parametrizována $k + 1$ body $\beta_0, \ldots, \beta_k$ v daném prostoru – ty nazveme kontrolními body (též řídící body). Mějme tedy parametr $t \in [0, 1]$ a hledejme jeho obraz. Pro lineární Bézierovu křivku ($k = 1$) je to jednoduše lineární interpolace mezi oběma kontrolními body. Formálně můžeme psát

$$C_1(t; \beta_0, \beta_1) = t\beta_0 + (1 - t)\beta_1.$$

Kvadratickou Bézierovu křivku pak zadefinujeme rekurzivně následujícím způsobem.

$$C_2(t; \beta_0, \beta_1, \beta_2) = tC_1(t; \beta_0, \beta_1) + (1 - t)C_1(t; \beta_1, \beta_2)$$

Jinými slovy děláme lineární interpolaci dvou lineárních interpolací sdílejících jeden koncový bod. Užitečné je také představovat si trojúhelník tvořený body $\beta_0, \beta_1, \beta_2$ – tomu říkáme kontrolní mnohoúhelník. Dále si všimněme, že křivka prochází body β_0 a β_2, nikoli však bodem β_1.

Nakonec zcela analogicky definujeme kubickou Bézierovu křivku.

$$C_3(t; \beta_0, \beta_1, \beta_2, \beta_3) = tC_2(t; \beta_0, \beta_1, \beta_2) + (1 - t)C_2(t; \beta_1, \beta_2, \beta_3)$$

Kontrolním mnohoúhelníkem je v tomto případě čtyřúhelník. Popsaný postup pro vyhodnocení křivky se nazývá *De Casteljauův algoritmus*, viz relevantní výukový text Michiganské technické univerzity [45]. Doplňme, že pro tento problém existují i jiné postupy, jako např. dosazení za funkce nižšího řádu a následné použití Hornerova schématu.

V terminologii zavedné v sekci 3.2 je tedy β_0 *počáteční opěrný bod*, β_1 *počáteční směrový bod*, β_2 *koncový směrový bod* a β_3 *koncový opěrný bod*.

Jak již bylo zmíněno, jednotlivé křivky následně spojíme do posloupnosti, kde splývají odpovídající koncové body každé sousedící dvojice křivek. Jednu křivku v posloupnosti nazveme *segment*, celou posloupnost pak *spline*.

Pro lepší pochopení uvedených konceptů doporučujeme zkusit si několik křivek namodelovat v libovolném editoru pro vektorovou grafiku či v našem herním editoru.

Zbývá uvést, jak počítat derivaci křivky a délku křivky. Pro derivaci je výhodné zbavit se rekurze ve vzorci pro C_3 dosazením za C_2 a následně za C_1. Dostaneme

$$C_3(t;\beta_0,\beta_1,\beta_2,\beta_3) = t^3\beta_0 + 3t^2(1-t)\beta_1 + 3t(1-t)^2\beta_2 + (1-t)^3\beta_3.$$

Derivací podle t pak snadno dostaneme vzorec pro tečnu.

$$\frac{\partial C_3(t;\beta_0,\beta_1,\beta_2,\beta_3)}{\partial t} = 3t^2\beta_0 + (6t-9t^2)\beta_1 + (9t^2-12t+3)\beta_2 - 3(1-t)^2\beta_3$$

Délku *segmentu* bychom chtěli umět spočítat např. pro počítání nejkratší cesty k zákazníkovi, pro odhad, kolik bodů je potřeba na daný *segment* nanést při renderování, nebo abychom určili, kolikrát se má zopakovat textura silnice. Zejména v případě zjišťování počtu bodů k nanesení bychom se chtěli vyhnout aproximaci křivky lomenou čarou, protože na to už body potřebujeme mít nanesené. Chtěli bychom tedy vzorec pro délku Bézierovy křivky z jejích kontrolních bodů. Takový uzavřený vzorec ale neexistuje, což souvisí s tím, neexistuje uzavřený vzorec pro kořeny polynomu stupně aspoň 5. Pro důkaz opět odkazujeme na článek *A Primer on Bézier Curves* [20], konkrétně jeho sekce *Arc length*.

Pořiďme si tedy aproximaci délky *segmentu*. Pro segment s kontrolními body $\beta_0,\beta_1,\beta_2,\beta_3$ je dolním odhadem jeho délky $|\beta_0\beta_3|$ a horním odhadem $|\beta_0\beta_1| + |\beta_1\beta_2| + |\beta_2+\beta_3|$. Tyto dvě hodnoty tedy zprůměrujeme a výsledek prohlásíme za hledaný odhad. I když jde o odhad velmi nepřesný, prakticky se ukázalo, že je pro všechny zmíněné účely dostačující. Pro jeho jednoduchost jej proto používáme.

5.4.3 Modul `Network`

Podadresář `Network` obsahuje nástroje pro práci se síťovými prvky a sítěmi. Nejprve uveďme nejdůležitější struktury rozšiřující typový systém, které používáme pro zesílení statické kontroly kódu (podobně jako v sekci 5.4.2). De facto se jedná o *read-only* struktury, nicméně většina takto označená není, aby bylo možné je serializovat (viz sekce 4.2.2 o serializaci).

- `ElementID` – identifikátor síťového prvku
- `AnchorID` – identifikátor kotvy síťového prvku
- `Anchor` – `AnchorID` a pozice dané kotvy
- `SerializableAttachmentInfo` – serializovatelný odkaz na jinou kotvu ve stejném nebo jiném síťovém prvku, tj. dvojice `ElementID` a `AnchorID`
- `Attachment` – deserializovaný odkaz na jinou kotvu v určitém síťovém objektu, tj. dvojice `NetworkElement` a `AnchorID`

Síťové prvky a spojení

Páteřními třídami jsou třídy reprezentující síťové prvky a jejich spojení, zaměřme se tedy na ně. Tyto třídy spolu s jejich nejdůležitějšími metodami a vlastnostmi sledujme na UML diagramu 5.2.

Základní abstraktní třídou reprezentující síťový prvek je `NetworkElement`. Ta má přiřazené GUID a obsahuje informace jak o serializovaných spojeních každé kotvy (vlastnost `Junctions`), tak o deserializovaných spojeních každé kotvy (vlastnost `SerializedAttachments` – pro informace o třídě `SerializableDict` viz sekce 5.9). Dále od svých potomků požaduje implementovat zejména vlastnost `Anchors`, tedy seznam pozic a GUID všech kotev daného prvku – o tom nemůže abstraktní `NetworkElement` rozhodnout.

Deserializovaná spojení síťových prvků jsou realizována pomocí křižovatek, jak je to popsáno v sekci 3.2.2. Křižovatku reprezentuje třída `Junction` – ta tedy obsahuje seznam jednotlivých napojených kotev a metody pro práci s ním.

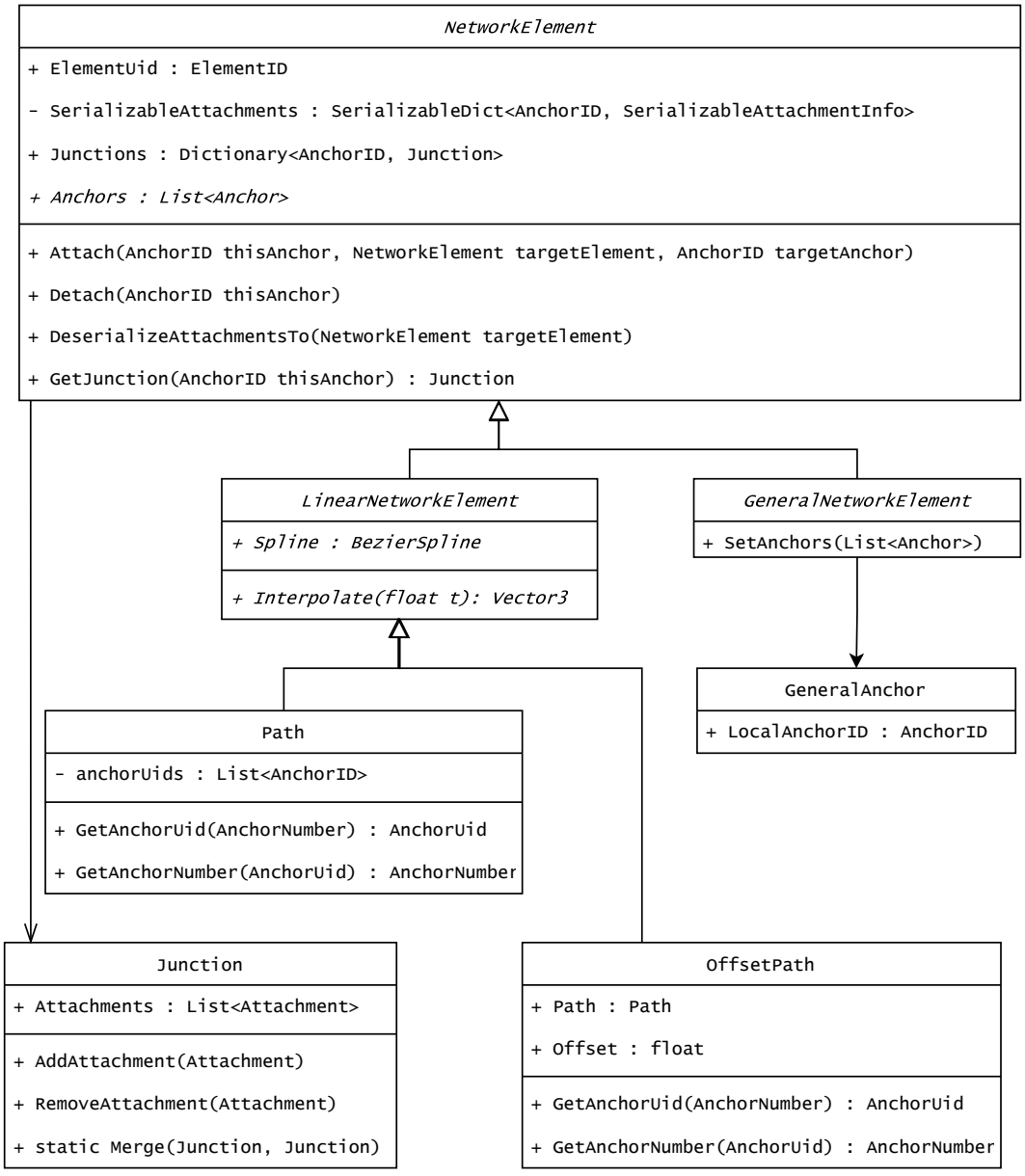
Existují 2 druhy síťových prvků:

- Síťové prvky, jejichž tvar je jednodimenzionální a kotvy jsou na něm lineárně uspořádány. Tvar je konkrétně daný instancí třídy `BezierSpline`, tedy posloupností Bézierových křivek. Tyto síťové prvky jsou reprezentované abstraktní třídou `LinearNetworkElement`. Nejjednodušším příkladem je třída `Path`, která pouze obohacuje třídu `BezierSpline` o GUID jednotlivých kotev. `Path` sleduje události vyvolané `BezierSpline` při přidání nebo odebrání kontrolních bodů a reaguje na to náležitou úpravou seznamu GUID kotev. Druhým příkladem je třída `OffsetPath`, jejíž tvar je dán posloupností Bézierových křivek posunutou o určitý *offset*. Obsahuje referenci na `Path` udávající tvar a číslo `Offset` udávající míru a směr posunutí. Příkladem využívajícím tuto třídu je chodník umístěný podél silnice, jehož tvar je dán tvarem dané silnice.

- Síťové prvky bez specifikovaného tvaru, jejichž kotvy mohou být umístěny libovolně. Ty jsou reprezentované abstraktní třídou `GeneralNetworkElement`. Příkladem využití je obytný dům, u kterého polohu kotev určuje herní návrhář. V budoucnu může být dalším příkladem např. parkoviště, kde kotvy budou umístěny u vjezdů do parkoviště. Samotná kotva prvku typu `GeneralNetworkElement` je reprezentována instancí třídy `GeneralAnchor`, která obsahuje identifikátor dané kotvy. Volba polohy kotvy probíhá tak, že k hernímu objektu s komponentou `GeneralNetworkElementHolder` přidáme jako potomka herní objekt s komponentou `GeneralAnchorHolder`. Poloha potomka pak určí polohu kotvy. Pro detailnější vysvětlení tohoto procesu s uživatelského hlediska viz sekce 7.11.

Deserializace spojení

Na konci sekce 3.2.2 jsme uvedli příklad, kdy je při deserializaci spojení potřeba spojit dvě již existující křižovatky do jedné. Popišme nyní detaily ohledně deserializace spojení podrobněji. Tato funkcionalita je soustředěna v metodě `NetworkElement.DeserializeAttachmentsTo`, která bere jako argument jiný síťový prvek a má za úkol deserializovat všechna serializovaná spojení vedoucí do tohoto prvku. Předpokládejme, že z kotvy `A` síťového prvku `P` je serializované



Obrázek 5.2: Diagram nejdůležitějších tříd modulu `Network` s vybranými vlastnostmi a metodami.

spojení do kotvy `B` prvku `Q`. Při deserializaci tohoto spojení mohou nastat 2 možnosti:

1. Kotvy `A` i `B` jsou již součástí nějakých deserializovaných křižovatek[1]. Protože po deserializaci spojení musí být `A` i `B` ve stejné křižovatce, je nutné obě již existující křižovatky spojit do jedné, a to prostou kontatenací seznamů napojených prvků obou křižovatek.

2. V opačném případě alespoň jedna z `A`, `B` nepatří do žádné křižovatky. Stačí tedy jednu kotvu přidat do křižovatky té druhé, případně vytvořit zcela novou křižovatku.

Trvalé odstranění spojení

Dále uveďme detaily ohledně trvalého odstranění nějaké kotvy z křižovatky, tedy metody `NetworkElement.Detach`. Kotvu může z křižovatky odstranit herní návrhář v herním editoru – je to operace inverzní k přidání kotvy do křižovatky. Dále je tato operace potřeba, když herní návrhář určitý prvek smaže nebo posune, poté je totiž potřeba zrušit všechna jeho spojení (více v sekci 5.8.1). Co se deserializovaných spojení týče (tedy vlastnosti `NetworkElement.Junctions`), je smazání konkrétního spojení jednoduché – stačí z křižovatky odstranit dané spojení a z `NetworkElement.Junctions` odstranit referenci na křižovatku.

Složitější je odstranění serializovaného záznamu o spojení. Pro pochopení problému si jednotlivá serializovaná spojení představme jako hrany grafu, kde vrcholy jsou kotvy všech síťových prvků. Křižovatky pak odpovídají komponentám slabé souvislosti, přičemž každá komponenta je zakořeněný strom (z každého vrcholu totiž vede nejvýše jedno serializované spojení). Smazáním spojení nechceme jednu komponentu (tj. křižovatku) rozdělit na dvě – místo toho chceme z komponenty odstranit jen jeden vrchol (konkrétně ten, na kterém byla zavolána metoda `NetworkElement.Detach`).

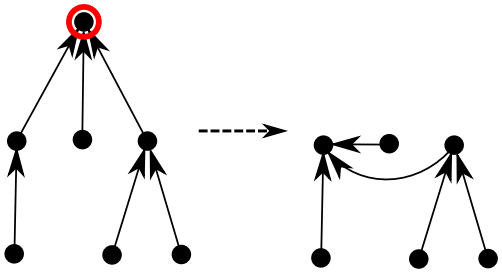
Řešíme tedy, jak ve stromu přepojit hrany, abychom jeden vrchol odstranili a zbytek stále tvořil strom. Mohou nastat 2 případy:

1. Odstraňovaný vrhol může být kořenem komponenty. Proces sledujme na obrázku 5.3, kde je odstraňovaný vrchol červeně označen. V tomto případě vybereme některého z jeho synů, označme jej `P` (syn kořene určitě existuje, protože každá křižovatka spojuje aspoň 2 kotvy). Hranu z `P` do kořene odstraníme a všechny ostatní hrany vedoucí z kořene přesměrujeme do `P`. Tím se `P` stane novým kořenem a původní kořen se stane izolovaným.

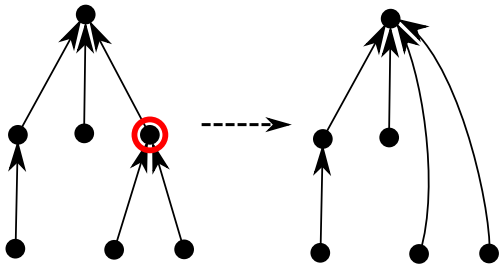
2. Opačný případ sledujme na obrázka 5.4. Pokud odstraňovaný vrchol není kořenem komponenty, má nějakého otce `Q`. V takovém případě stačí všechny hrany vedoucí do odstraňovaného vrcholu přesměrovat do `Q`.

V obou případech potřebujeme mít k dispozici vrcholy náležící do stejné komponenty, jako odstraňovaný vrchol – tedy potřebujeme, aby všechna spojení související s danou křižovatkou byla deserializována. Tento předpoklad je bezpečné učinit, protože odstraňování kotvy z křižovatky probíhá pouze v herním editoru, kde je daná část herního světa celá načtena (v současné verzi se dokonce celý herní svět načítá najednou, ale na to nelze do budoucna spoléhat).

[1] Jinými slovy `P.Junctions.ContainsKey(A) && Q.Junctions.ContainsKey(B)`.



Obrázek 5.3: Smazání kořene ze stromu tak, aby se strom nerozpadl.



Obrázek 5.4: Smazání nekořenového vrcholu ze stromu tak, aby se strom nerozpadl.

Prohledávání sítě

V kapitolách 2 a 3 jsme stanovili, že ve více různých případech je potřeba prohledat síť podle vzrůstající vzdálenosti od určité počáteční kotvy – např. při hledání nejkratší cesty pro horizontální navigaci nebo při výběru destinace zákazníka. Tato funkcionalita je soustředěna do třídy `NetworkSearch` a pro samotné prohledávání jsme zvolili Dijkstrův algoritmus. Jen je potřeba určit, jak vypadá graf, který prohledáváme. To se odvíjí od toho, zda se po nalezené cestě bude následně pohybovat chodec, nebo auto. Zatímco chodec může libovolně měnit směr pohybu, auto se na běžné silnici otočit nemůže. Proto poskytneme dva druhy vyhledávání:

1. Prohledávání bez omezení otáčení – vrcholy prohledávaného grafu jsou jednoduše kotvy síťových prvků. Z kotvy lze tedy při hledání přejít na libovolnou sousední kotvu. Zároveň žádná kotva se v nejkratší cestě nebude vyskytovat dvakrát.

2. Prohledávání s omezením otáčení – vrcholy prohledávaného grafu jsou dvojice kotva a směr. Při hledání lze přejít pouze na tu sousední kotvu, která leží v určeném směru. To znamená, že přes jednu kotvu může nejkratší cesta projít dvakrát – pokaždé v jiném směru. To odpovídá např. situaci, kdy musí auto dojet na kruhový okruh, kde se otočí a pojede stejnou cestou zpět v opačném směru.

5.4.4 Modul `Rendering`

Podadresář `Rendering` zajišťuje 2 funkcionality:

1. tvorbu meshe silnic a chodníků,

2. nanášení horizontální navigace na silnice.

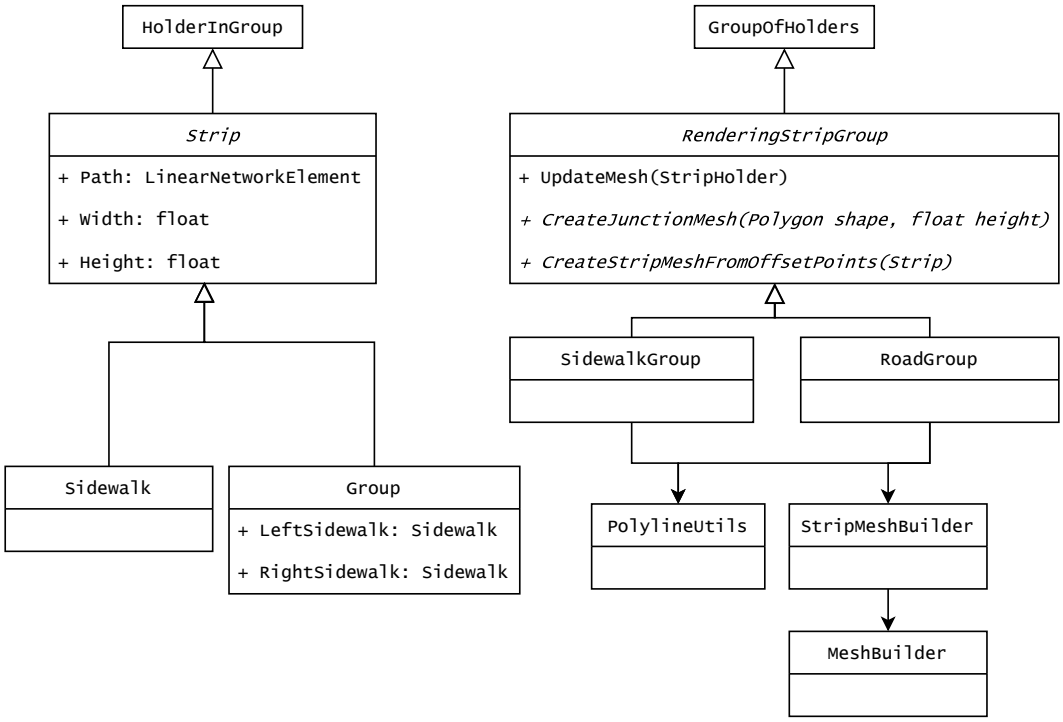
Tvorba meshe silnic a chodníků

Pro silnice a chodníky jsme vytvořili společnou abstrakci objektu, jehož tvar je určen instancí `LinearNetworkElement` a který má určitou šířku a výšku – takový objekt nazýváme *strip* (česky budeme říkat *pás*) a je reprezentován abstraktní

třídou `Strip`. Jejími konkrétními potomky jsou třídy `Road` reprezentující silnici a `Sidewalk` reprezentující chodník, jak je znázorněno na UML diagramu 5.5. Podobná abstrakce je zavedena na úrovni holderů, kde `RoadHolder` (holder třídy `Road`) a `SidewalkHolder` (holder třídy `Sidewalk`) sdílejí společného abstraktního předka `StripHolder`.

Díky funkcionalitě popsané v předchozí sekci 5.4.3 se jednotlivé pásy mohou spojovat do sítí – to konkrétně tak, že se spojují jim náležící instance `LinearNetworkElement`. Tím vznikají křižovatky, jak je známe ze skutečného světa. Při tvorbě meshe pro určitý pás je pak důležité, aby byl k dispozici i tvar všech křižovatek, kterých se účastní – ty totiž ovlivňují jeho tvar. Tím pádem je nutné mít k dispozici tvar všech napojených pásů, protože právě ty určují tvar křižovatek.

Proto jsme využili konceptu *skupin holderů* popsaných v předchozí sekci 5.4.3: `StripHolder` je potomkem `HolderInGroup` a jemu odpovídající skupina je abstraktní třída `RenderingStripGroup`. Ta má pak k dispozici všechny načtené pásy a její funkcí je tvorba meshů pro jednotlivé pásy. Jejími konkrétními potomky jsou `RoadGroup` a `SidewalkGroup` (viz opět diagram 5.5), které mohou upravovat detaily, ve kterých se liší tvorba meshe silnic od tvorby meshe chodníků. V současné verzi nicméně žádné odlišnosti neexistují a obě třídy delegují tvorbu meshe na třídu `StripMeshBuilder`. Ta dále využívá třídu `MeshBuilder` pro nízkoúrovňovou práci s meshem, jako např. přidávání polygonálních stěn nebo přidávání kvádrů. Při tvorbu meshe křižovatek je dále využita třída `PolylineUtils` zajišťující detekci průniku lomených čar a algoritmus `decrossing` popsaný v sekci 3.2.6.



Obrázek 5.5: Diagram nejdůležitějších tříd modulu `Rendering` s vybranými vlastnostmi a metodami.

Samotný princip algoritmu pro tvorbu meshe pásů a křižovatek jsme detailně popsali v sekci 3.2. V téže sekci jsme popsali výpočet UV souřadnic pro nanesení textury na mesh. Prosté nanesení textury bez použití dalších technik ovšem

produkuje nepřirozené artefakty, pokud hráč vidí dlouhou silnici pod relativně ostrým úhlem, což se v naší hře stává často. Tyto artefakty jsou zachyceny na obrázku 5.6 na první silnici zleva – středová čára silnice je na různých úsecích různě zdeformovaná, na některých úsecích dokonce zmizí úplně. Stejně tak obě krajnice jsou ve větší vzdálenosti naneseny přerušovaně.

Pro vyřešení tohoto problému lze použít tzv. mipmapy (mipmaps), tedy předpočítané zmenšené verze dané textury, mezi kterými se přechází se vzrůstající vzdálenosti objektu od kamery – viz sekce Mipmaps manuálu Unity [46]. Unity nabízí dobrou podporu pro práci s mipmapami, pro jejich zapnutí stačí v okně Inspector u dané textury zaškrtnout pole *Generate Mip Maps*. Při nanesení textury na mesh se pak pro každý bod meshe vybere nejvhodnější verze textury. Přesto ale danému bodu meshe nebude přesně odpovídat jeden pixel textury (tzv. texel) a je tedy potřeba zvolit strategii, jak získat hodnotu, která se na daný bod nanese. Unity umožňuje výběr z následujících strategií (pro informace o jejich fungování viz článek *Texture filtering* vývojářské příručky společnosti arm [47], v případě anisotropického filteringu pak sekce *Anisotropic filtering* článku *Texture filtering* encyklopedie Wikipedia [48]):

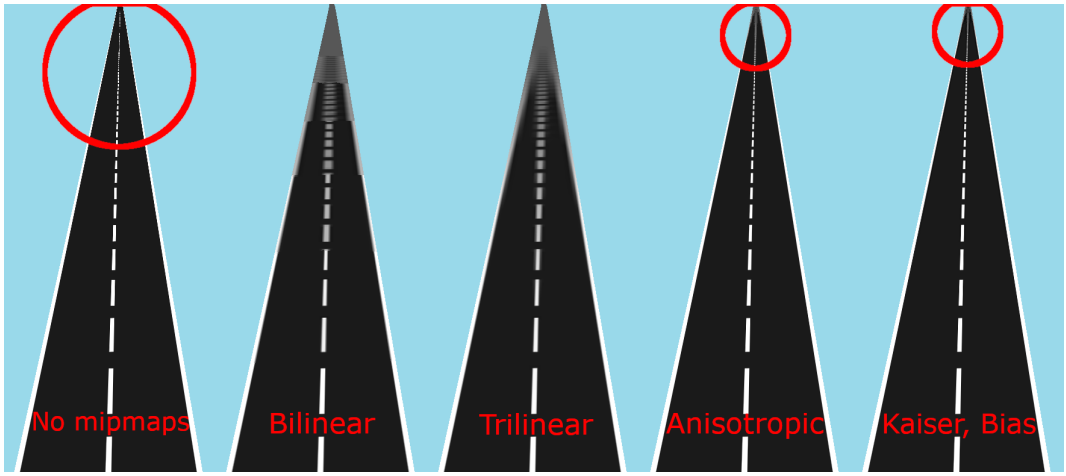
- *Point* – tato metoda dává v našem případě velmi nepřirozené ostré přechody mezi barvami a „blokový“ vzhled (na obrázku 5.6 ji proto ani neuvádíme),
- *Bilinear* – výsledek této metody je uveden jako druhý zleva na obrázku 5.6, ze kterého je zřejmý její nedostatek – textura vypadá velmi nepřirozeně při přechodech mezi mipmapami, navíc menší mipmapy obsahují příliš rozmazanou středovou čáru,
- *Trilinear* – výsledek této metody je uveden jako třetí zleva na obrázku 5.6; přechody mezi jednotlivými mipmapami již nejsou viditelné, ale menší mipmapy jsou stále velmi rozmazané,
- *Anisotropic* – výsledek této metody (konkrétně s parametrem *Aniso Level* nastaveným na nejvyšší hodnotu 16) je uveden jako čtvrtý zleva na obrázku 5.6 a z hlediska hráčského zážitku je již přijatelný.

Strategie Anisotropic filtering, která produkuje nejlepší výsledky, znamená také nejvyšší náročnost pro grafickou kartu. Nicméně na námi použitém cílovém zařízeních běží hra stále plynule a proto tuto strategii použijeme (námi použitý mobilní telefon konkrétně obsahuje čip Qualcomm SM7225 Snapdragon 750G 5G s grafickým procesorem Qualcomm Adreno 619).

Pro ještě lepší konečný výsledek požijeme dvě další techniky, které doporučuje manuál Unity:

- V nabídce *Mip Map Filtering* vybereme algoritmus *Kaiser*, který je v manuálu Unity speciálně doporučen pro případ, kdy je textura ve větší vzdálenosti rozmazaná [49].
- Použijeme atribut textury `Texture.mipMapBias`, kterým lze ovládat, v jaké vzdálenosti od kamery dojde k přepnutí na menší verzi textury. Záporné hodnoty způsobí zostření výsledku, viz dokumentace tohoto atributu [50]. Konkrétně pro `Texture.mipMapBias` zvolíme hodnotu -0.5, což je nejnižší doporučená hodnota.

Výsledek po aplikaci obou technik je uveden jako první zprava na obrázku 5.6. Relativní zlepšení je pozorovatelné až ve větší vzdálenosti od kamery. Pro lepší nahlédnutí rozdílu doporučujeme vyzkoušet různé parametry přímo v Unity Editoru.



Obrázek 5.6: Porovnání různých přístupů k nanesení textury silnice. Silnice vlevo nepoužívá mipmapy, zbytek mipmapy používá a navíc přidává postupně: bilineární interpolaci, trilineární interpolaci, anisotropní interpolaci a algoritmus Kaiser spolu s nastavením atributu `mipMapBias`.

V případě, že by v budoucích verzích hry bylo potřeba vzhled silnic ve vyšší vzdálenosti od kamery ještě vylepšit (např. kvůli tomu, že textury budou komplikovanější a příliš zmenšené budou vypadat nepřirozeně), nebo by bylo potřeba ušetřit výkon grafické karty, je možným řešením přepnout ve vyšší vzdálenosti na jednodušší verzi textury pomocí speciálně vytvořeného shaderu.

Horizontální navigace

Druhá část modulu `Rendering` je třída `HorizontalNavigation`. Ta má za úkol nanést na silnici čáru, kterou může hráč následovat, aby se co nejkratší cestou dostal k zákazníkovi nebo k zákazníkově destinaci. Význam takové navigace jsme ustanovili v sekci 2.2 a samotné hledání nejkratší cesty v síti silnic jsme popsali v sekci 5.4.3.

Podobně jako v případě krajnice silnic aproximujeme navigační čáru lomenou čarou. Pro vykreslení lomené čáry nabízí Unity komponentu `LineRenderer`, které stačí poskytnout posloupnost vrcholů požadované čáry. Tuto posloupnost získáme z nalezené nejkratší cesty, což je nějaká posloupnost kotev náležející určitým silnicím. Na křivkách určujících tvar těchto silnic použijeme metody `BezierSpline.SamplePoints` a `BezierSpline.UniformalizePoints`, čímž silnice pokryjeme uniformně rozmístěnými body. Na získané posloupnosti bodů pak provedeme tyto operace:

1. Protože začátek ani konec nejkratší cesty se nemusí nacházet přesně na nějaké kotvě, může být potřeba začátek a konec posloupnosti bodů oříznout. Tím docílíme např. toho, že navigační čára se bude zobrazovat jen před autem, nikoli pod ním nebo za ním.

2. Získané body jsou umístěny na středové čáře silnice, takže je posuneme do správného jízdného pruhu.

3. Na křižovatkách, kde se spojují jednotlivé křivky, se můžou posunuté body křížit – stejně, jako se mohou křížit krajnice dvou spojených silnic. To vyřešíme stejně jako v případě krajnic použitím algoritmu pro nalezení průsečíku dvou lomených čar, který je implementován ve třídě `PolylineUtils` (pro více informací o této třídě vic sekce 5.4.3.

Takto upravená posloupnost bodů pak slouží jako vrcholy lomené čáry nanesené na silnici.

5.4.5 Modul `Audio`

Pro přehrávání zvuků jsou v Unity určeny zejména tyto 3 třídy:

- `AudioClip` – obsahuje samotnou zvukovou nahrávku.
- `AudioSource` – komponenta přehrávající ve scéně zvuk.
- `AudioListener` – komponenta reprezentující místo, ze kterého hráč zvuky poslouchá. Ve scéně může být nejvýše jedna, v našem případě je umístěna do stejného místa jako kamera.

Relativní poloha instancí `AudioSource` a `AudioListener` pak určuje, které zvuky hráč uslyší a jak hlasitě.

Náš modul `Audio` poskytuje velmi jednoduché rozhraní pro načítání instancí `AudioClip` a jejich následné přehrávání v komponentách `AudioSource`. Lze přitom zvolit parametry přehrávání, jako např. hlasitost. Veškerou tuto funkcionalitu nabízí třída `AudioManager`, konkrétně její metoda `PlaySound`.

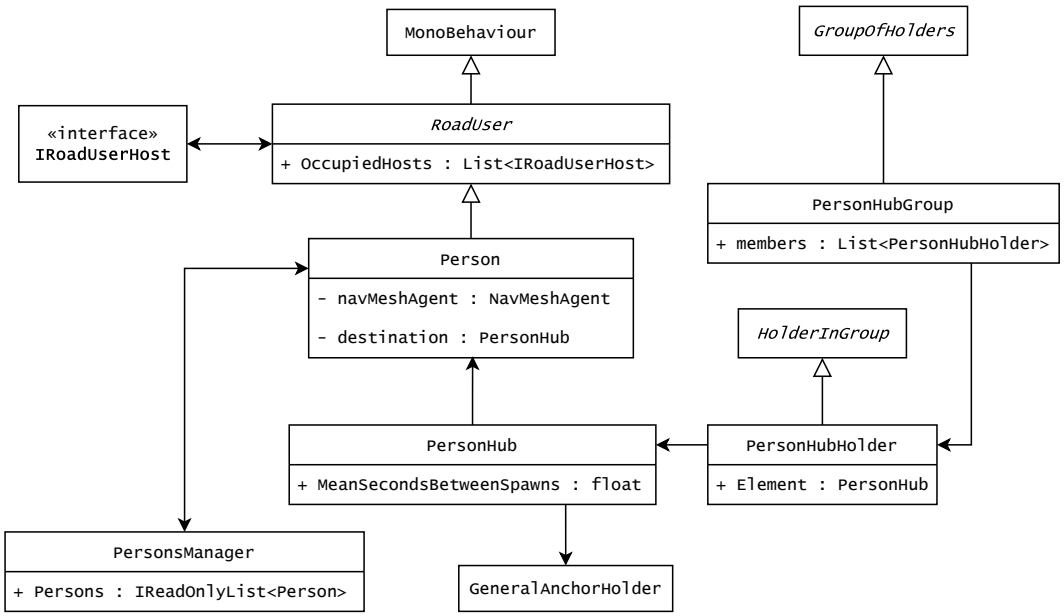
Tento modul je využíván následujícími komponentami:

- třídou `GameManager`, která na začátku hry spustí přehrávání hudby,
- třídou `Car`, která přehrává zvuk motoru, přičemž reguluje jeho hlasitost na základě míry sešlápnutí plynu,
- třídou `Person`, která přehrává zvuk výkřiku, pokud do daného člověka ve vysoké rychlosti narazí auto,
- komponentami uživatelského rozhraní, jako např. `NewCustomerOfferDialog`, která přehrává zvukový efekt při zobrazení nabídky nového zákazníka.

5.4.6 Modul `People`

Modul `People` zajišťuje generování chodců a řízení jejich pohybu. Jeho strukturu sledujme na diagramu 5.7.

Nejdůležitějšími třídami modulu jsou třída `Person` reprezentující jednoho chodce a třída `PersonsManager`, která registruje všechny existující chodce a poskytuje jejich seznam. Rozeberme zvlášť problém generování a navigace chodců:



Obrázek 5.7: Diagram nejdůležitějších tříd modulu `People` s vybranými vlastnostmi a metodami.

- Chodci se generují v souladu s návrhem uvedeným v sekci 3.3.1. Generují se tedy u obytné budovy v bodě, kde je na dům napojen chodník, přičemž každý takový bod je reprezentován herním objektem s komponentou `GeneralAnchorHolder`. Ty body, ze kterých se mají generovat chodci, označujeme jako *person hub*. Ve scéně jsou tyto body určené tak, že mají vedle komponenty `GeneralAnchorHolder` i holder třídy `PersonHub`, tedy `PersonHubHolder`.

  Samotné generování chodců pak probíhá v náhodných intervalech ve třídě `PersonHub`. Vygenerování jednoho chodce konkrétně obnáší náhodné určení jeho rychlosti a destinace. Jako možné destinace jsou přitom uváženy všechny objekty s komponentou `PersonHub`, ke kterým lze po chodnících dojít z výchozího místa chodce.

- Samotná navigace chodců je delegována na navigační systém Unity, jak jsme rozhodli v sekci 3.3.1. Každý chodec má přiřazenou komponentu `NavMeshAgent`, které pouze předá svou požadovanou destinaci.

5.5 Implementace herních mechanik

V herním světě se odehrávají dvě herní mechaniky: řízení auta a herní mise taxikáře popsaná v sekci 3.3.2. K oběma se pak vztahuje sada ovládacích prvků. Všechny tyto herní součástí popíšeme v této sekci.

5.5.1 Fyzikální simulace auta

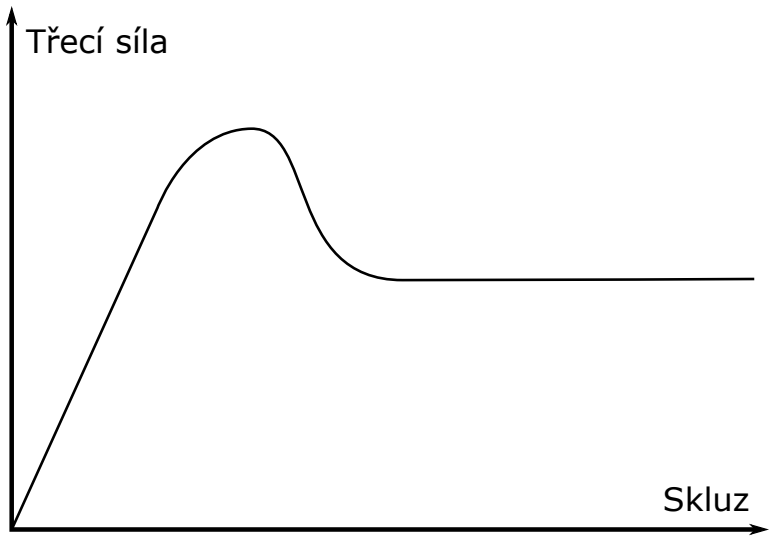
Třídy týkající se fyzikální simulace auta jsou umístěny ve jmenném prostoru `CarPhysics`. Jedná se o třídu `Car` reprezentující hráčovo auto a třídu `CarWheel`

reprezentující pneumatiku auta. Samotná simulace se opírá o komponentu `WheelCollider` enginu Unity. Stejně jako celý engine pro fyzikální simulaci v Unity je i `WheelCollider` založená na integraci open-source enginenu PhysX od společnosti Nvidia, viz dokumentace modulu pro fyzikální simulaci enginu Unity [51]. Tato komponenta je přidána ke každé pneumatice zvlášť a zajišťuje následující funkce (viz manuál komponenty `WheelCollider` [52]):

1. Detekci a simulaci kolizí pneumatiky s ostatními herními objekty, podobně jako běžný potomek třídy `Collider`.

2. Odpružení pneumatiky.

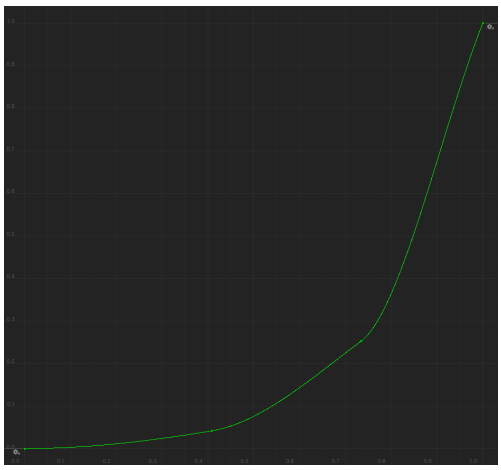
3. Tření pneumatiky o silnici ve směru dopředném a příčném. Třecí koeficient, ze kterého je vypočtena třecí síla, se v obou směrech určí pomocí tzv. *slip-based tire friction model*. To znamená, že se pro daný směr nejprve určí tzv. *skluz* (*slip*), tedy rozdíl mezi obvodovou rychlostí kola a rychlostí vozidla – pro detaily viz studijní materiál Doc. Ing. Jiřího Danzera CSc o adhezi [53]. Na základě skluzu a hmotnosti vozidla se následně určí třecí síla. Závislost třecí síly na skluzu při dané hmotnosti se nazývá skluzová charakteristika a jedná se o vlastnost pneumatiky. Konkrétní příklad skluzové charakteristiky je uveden na obrázku 5.8 převzatém z manuálu Unity [52]. Uvedená funkce stoupá přibližně lineárně až do maximální hodnoty (*Extremum Slip*), kde má derivaci 0. Následně klesá až ke své asymptotické hodnotě (*Asymptote Slip*), kde je derivace opět 0 a dále je funkce přibližně konstantní. Vztah určující tření je tak poměrně komplexní a nebylo by možné jej aproximovat použitím konstantního třecího koeficientu.



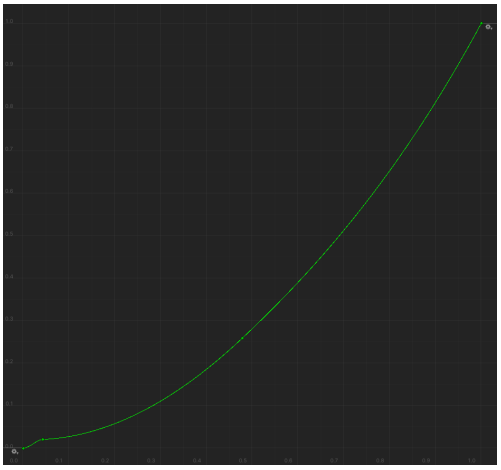
Obrázek 5.8: Typický tvar skluzové charakteristiky. Vytvořeno na základě grafu uvedeného na `https://docs.unity3d.com/2021.2/Documentation/uploads/Main/WheelFrictionCurve.png`

Komponenta `WheelCollider` je ovládána manipulací s následujícími 3 parametry:

1. `motorTorque`, tedy točivý moment, kterým je poháněno dané kolo, v jednotkách Newton metr. Znaménko hodnoty určuje směr točivého momentu.



Obrázek 5.9: Závislost točivého momentu motoru na výkonu motoru.



Obrázek 5.10: Závislost otočení kol na otočení volantu.

2. `brakeTorque`, tedy točivý moment, kterým je brzděno dané kolo, v jednotkách Newton metr. Znaménko hodnoty určuje směr točivého momentu.

3. `steerAngle`, tedy otočení kola okolo svislé osy ve stupních.

Pro každý z těchto parametrů jsme určili minimální a maximální hodnotu (viz tabulka 5.1). Hráč pak má k dispozici ovládací prvky, kterými určuje výkon motoru (*motor throttle*), výkon brzd (*braking intensity*) a otočení volantu (*steering*). Tyto 3 vstupy jsou normalizované do intervalů $[0, 1]$ v případě výkonu motoru a výkonu brzd, resp. $[-1, 1]$ v případě otočení volantu. Konkrétní podobu ovládacích prvků uvedeme v následující sekci 5.5.2. Nyní rozhodneme, jak se normalizované vstupy zobrazí na konkrétní hodnoty parametrů pro `WheelCollider`.

Jednoduché lineární zobrazení vstupů z intervalu $[0, 1]$ na parametry z intervalu $[0, \text{maximum}]$ funguje dobře jen v případě zobrazení výkonu brzd na točivý moment brzd. Experimentálně jsme zjistili, že v případě zobrazení výkonu motoru na točivý moment motoru hráč očekává, že při malých hodnotách bude ovládání citlivější. Konkrétní závislost, kterou jsme odladili pro co nejlepší herní zážitek, je uvedena na obrázku 5.9. Zobrazená křivka je přibližně složena ze tří lineárních částí se stoupajícím náklonem. Podobně u závislost otočení kol na otočení volantu hráč očekává, že při menším otočení je ovládání citlivější. Tato závislost je uvedena na obrázku 5.10. Jedná se přibližně o část paraboly. Pro vytvoření obou křivek jsme použili Unity komponentu `AnimationCurve`.

Protože je samotná fyzikální simulace již vyřešena enginem Unity, je dále zapotřebí pouze vhodně nastavit parametry auta a pneumatik. To je ovšem v případě komponenty `WheelCollider` obtížný problém, což potvrdily jak naše experimenty, tak názory jiných uživatelů na webových fórech – viz např. vlákno na téma *Good values for WheelColliders* [54].

Experimentálně jsme došli k modelu auta, který nyní popíšeme. Cílem byla přirozenost herního zážitku. Výsledné auto má 4 kola na dvou nápravách. Kola na přední nápravě jsou řiditelná, tedy se otáčí okolo svislé osy v závislosti na otočení volantu. Kola na zadní nápravě jsou hnací, tedy na ně působí točivý moment motoru vypočítaný postupem uvedeným výše. Na všechna 4 kola působí točivý

Hmotnost auta	2 000 kg
Maximální točivý moment motoru vyvinutý na 1 kolo	10 000 Nm
Maximální točivý moment brzd vyvinutý na 1 kolo	40 000 Nm
Maximální otočení kol	30°
Koeficient odporu vzduchu auta (*drag*)	0,2
Koeficient odporu vzduchu auta pro rotační pohyb (*angular drag*)	0,3
Koeficient intenzity odpružení kol (*spring*)	20000
Koeficient tlumícího efektu odpružení kol (*damper*)	3000
Koeficient tření kol v dopředném směru (*forward friction stiffness*)	1,5
Koeficient tření kol v příčném směru (*sideways friction stiffness*)	2

Tabulka 5.1: Hodnoty, které bylo v našem případě důležité nastavit pro přirozenou jízdu auta.

moment brz, pokud jsou brzdy aktivní.

Auto má Unity komponenty `Rigidbody` a `BoxCollider`. Každé kolo pak má komponentu `WheelCollider`. Těžiště auta je posunuto oproti středu auta mírně směrem dolů, jak je doporučeno pro zvýšení stability auta v manuálu komponenty `WheelCollider` [52]. Pro výčet všech nastavitelných parametrů uvedených komponent odkazujeme na manuál enginu Unity [29]. V tabulce 5.1 uvádíme námi zvolené hodnoty parametrů, které v kvalitě simulace hrály největší roli.

Všechny tyto hodnoty lze upravit v Unity Editoru.

5.5.2 Ovládání auta

Ovládací prvky auta jsou reprezentovány třídami v podadresáři `UserControl`. Hlavními prvky jsou volant, přepínání směru jízdy a sekce pro regulaci plynu a brzd. Vzhled a fungování ovládacích prvků jsme uvedli v sekci 2.3, zde uvedeme implementační detaily. Dalšími prvky zajišťující ovládání jsou tlačítko pro návrat do úvedního menu a komponenta zajišťující sledování auta kamerou z *third-person* pohledu. Popišme jednotlivé prvky popořadě:

- Pravá část obrazovky určená pro ovládání volantu je reprezentována komponentou `SteeringControlSection`, která zaznamenává hráčovo tažení prstem po této části obrazovky. Na základě toho se otáčí samotný volant reprezentovaný komponentou `SteeringWheel`, která pak informaci o otočení předává hráčovu autu. Pokud hráč prst zvedne, tedy „pustí volant“, vrací se volant samovolně do výchozí polohy, což podle našich experimentů usnadňuje řízení. V souvislosti s volantem bylo třeba nastavit dva parametry:
  - Převod mezi otočením volantu a natočením předních kol auta, tedy o kolik stupňů se musí otočit volant, aby se kola otočila o jeden stupeň okolo svislé osy. Tento parametr se nazývá *steering ratio*.
  - Úhlová rychlost, se kterou se volant vrací do výchozí polohy, pokud není hráčem držen.

Steering ratio	6
Rychlost návratu volantu	1080 °/s

Tabulka 5.2: Parametry volantu.

Hodnoty těchto parametrů, které jsme vybrali na základě našich experimentů jako nejpřirozenější pro ovládání auta, jsou uvedeny v tabulce 5.2.

- Levá část obrazovky určená pro ovládání plynu a brzdy je reprezentována komponentou `ThrustBrakeSection`. Ta zaznamenává svislou souřadnici bodu, kterého se uživatel v této části obrazovky dotkl jako posledního. Tu předává komponentě `ThrustBrakeBar` reprezentující samotný ovládací sloupec, který na jejím základě nastaví své rozměry – princip tohoto ovládacího sloupce je z návrhářského hlediska popsán v sekci 2.3 a z uživatelského hlediska v sekci 6.4.
- Tlačítko pro přepínání směru jízdy je ovládáno komponentou `DriveModeSection`, která si pouze pamatuje aktuální směr jízdy a při dotyku jej přepne.
- Tlačítko pro návrat do úvodního menu je ovládáno komponentou `ExitButton`, které při stisku použije metodu `SceneManager.LoadScene` pro načtění scény úvodního menu.
- Pohyb kamery sledující auto zezadu zajišťuje komponenta `CameraFollow`.

Pro testování hry na PC existuje pro pohodlnost i možnost ovládat auto pomocí klávesnice. Tento způsob však není optimalizovaný pro herní zážitek. Konkrétně směr lze ovládat levou a pravou šipkou, plyn a brzdy šipkou nahoru a šipkou dolu a přepínat směr jízdy lze stiskem mezerníku.

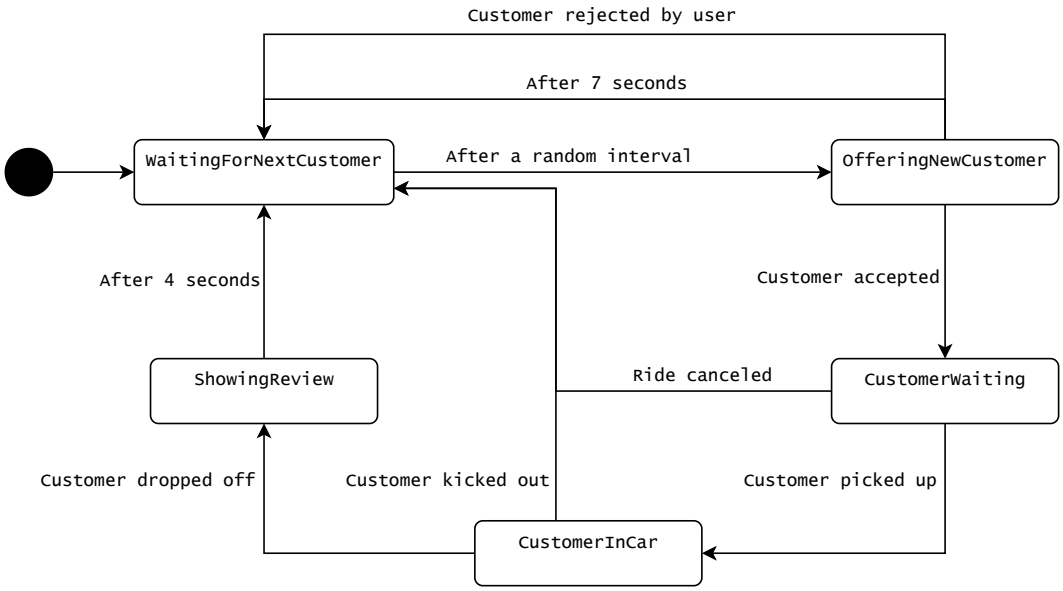
5.5.3 Taxi mise

Ve hře je implementována herní mise popsaná v sekcích 2.2 a 3.3.2. Ta je reprezentována třídou `TaxiMission`, kterou vytvoří komponenta `GameManager` při startu hry. `GameManager` také propojí instanci `TaxiMission` s ovládacími prvky mise, které jsou řízeny třídami v podadresáři `UserControl/TaxiMission`.

Samotná mise je řízena konečným automatem uvedeným na obrázku 5.11. Každý stav je reprezentován třídou implementující rozhraní `ITaxiMissionState`, jejíž název je shodný s názvem stavu. Přechod do nového stavu pak obnáší vytvoření příslušné instance a její přiřazení do vlastnosti `State` třídy `TaxiMission`.

Popišme jednotlivé stavy a přechody mezi nimi.

- Pokud zrovna není nabízen nový zákazník, ani není žádná zakázka aktivní, je mise ve stavu `WaitingForNextCustomer`. To simuluje případ, kdy taxikář čeká na novou zakázku. Dobu čekání modelujeme exponenciálním rozdělením, poté je hráči nabídnut zákazník a mise přejde do stavu `OfferingNewCustomer`. Výběr chodce, kterého nabídneme jako zákazníka, je podrobně popsán v sekci 3.3.2. Nabídnutému zákazníkovi náhodně přiřadíme jméno a telefonní číslo a vypočítáme jeho vzdálenost od hráčova auta (tj. vzdálenost nejkratší trasy po silnicích).



Obrázek 5.11: Stavový diagram herní mise.

- Ve stavu `OfferingNewCustomer` se hráči v pravém horním rohu obrazovky zobrazí dialogové okno ovládané třídou `NewCustomerOfferDialog`, které umožní novou nabídku přijmout nebo odmítnout. Pokud hráč nezareaguje do 7 vteřin, nabídka se automaticky odmítne – to kopíruje chování skutečných zákazníků, kteří se taxislužbě nedovolali. Konkrétní časový interval, po kterém se nabídka odmítne, lze spolu s dalšími parametry nastavit úpravou konstant ve třídě `TaxiMission`. Při přijetí nabídky přesune třída `NewCustomerOffer` misi do stavu `CustomerWaiting`, při odmítnutí nabídky pak zpět do stavu `WaitingForNextCustomer`.

- Ve stavu `CustomerWaiting` je hráč navigován pomocí horizontální navigace („čáře na silnici před autem“) k čekajícímu zákazníkovi. Zákazník je navíc označen šipkou, která nad ním levituje. Po celou dobu navigace je hráči v pravém horním rohu obrazovky zobrazena informace o zákazníkovi a zbývající vzdálenost, spolu s tlačítkem umožňujícím zakázku zrušit, čímž se mise přesune zpět do stavu `WaitingForNextCustomer`. Tento informační dialog ovládá třída `ActiveCustomerDialog`. Jakmile se hráčovo auto k zákazníkovi dostatečně přiblíží a zastaví, nastoupí zákazník do auta. Tím se mise přesune do stavu `CustomerInCar`.

- Ve stavu `CustomerInCar` je hráč navigován pomocí horizontální navigace k destinaci zákazníka. Stejně jako předtím zákazník je destinace označena levitující šipkou. Navíc je v hráči v pravém horním rohu obrazovky zobrazena informace o zbývající vzdálenosti spolu s tlačítkem umožňujícím zákazníka vyhodit z auta, čímž se mise přesune zpět do stavu `WaitingForNextCustomer`. Jakmile se hráčovo auto dostatečně přiblíží k zákazníkově destinaci a zastaví, zákazník zaplatí, zhodnotí kvalitu jízdy, vystoupí a odejde. Tím se mise přesune do stavu `ShowingReview`.

- Ve stavu `ShowingReview` je hráči v pravé horní čáti obrazovky zobrazeno dialogové okno reprezentované třídou `CustomerReviewDialog`. To uvádí zá-

kazníkovo hodnocení a zaplacené peníze, přičemž obojí se odvíjí od kvality a rychlosti jízdy. Hodnocení i výdělek se náležitě započítají do hráčova skóre. Po 5 vteřinách dialogové okno zmizí a mise se přesune zpět do stavu `WaitingForNextCustomer`. Tím je celý jeden běh mise dokončen.

5.6 Perzistentní uložení stavu hry

Mezi spuštěními hry je perzistentně uchována část jejího stavu. Konkrétně se uchovávají tyto informace:

1. stav herní mise – tedy dosažené hráčovo skóre (peníze a hvězničkové hodnocení), případně informace o aktuálním zákazníkovi a jeho destinaci,

2. pozice a rychlost hráčova auta,

3. pozice a destinace každého chodce.

Pro každou z těchto položek je určen jeden herní objekt, který rozhoduje, jaká data se u dané položky ukládají. Tyto objekty jsou:

1. `TaxiMission` (odpovídá za stav mise)

2. `Car` (odpovídá za pozici a rychlost auta)

3. `PersonsManager` (odpovídá za pozice a destinace chodců)

Každý z těchto herních objektů implementuje rozhraní `IPersistentDataContainer`. Komponenta `PersistenceManager` pak při spuštění či ukončení hry projde celou hierarchii herních objektů, najde všechny komponenty implementující `IPersistentDataContainer` a jejich odpovídající data načte či uloží pomocí statické třídy `PersistentStorage`.

Třída `PersistentStorage` používá formát JSON pro převedení dat do textové podoby, ve které je pak ukládá do souborů. Soubory ukládá do adresáře určeného konstantou `Application.persistentDataPath`, která je nastavena enginem Unity podle cílové platformy.

5.7 Úvodní menu

Úvodní menu je realizováno samostatnou scénou `EntryMenu`. Ta je sestavena převážně z ovládacích prvků standardního balíčku Unity pro UI (tzv. *Unity UI package* nebo také `uGUI` package – viz sekce *Unity UI* manuálu Unity [55]). Zejména jsme použili komponenty `Canvas`, `Image` a `Button`. Pro text jsme použili nestandardní balíček `TextMesh Pro` zmíněný v sekci 5.1.

Tlačítka jsou obsluhovány třídami `EntryMenuControls` a `ResetMenuControls` nacházejícími se v adresáři `EntryMenu`.

5.8 Editor herního světa

Skripty rozšiřující Unity Editor jsou umístěny ve adresáři `Assets/Editor`. Do Unity Editoru konkrétně přidávají následující prvky (pro stručný popis toho, jak Unity umožňuje rozšíření Editoru, viz sekce 4.3).

- Nástroje pro ladění hry, které popíšeme v sekci 5.9.1. Ty jsou umístěny v nabídce *Traffic Tools* v horní liště Unity Editoru a ovládá je třída `CustomTools`.
- Klávesové zkratky pro přidání nové silnice či chodníku (viz sekce 7.3 uživatelské dokumentace). Ty zajišťuje třída `CustomShortcuts`.
- Nástroje pro správu navigačního meshe a dat uložených hrou. Ty jsou konkrétně dostupné v okně *Inspector* po označení herního objektu `GameManager`. Pro jejich popis viz sekce 7.9.
- Nástroje pro práci s Bézierovými křivkami, silnicemi, chodníky a budovami. Jejich popis uvedeme ve zbytku sekce.

Rozdělení zodpovědnosti mezi *custom editory* je určeno ve smyslu sekce 3.4, tedy na práci s Bézierovými křivkami, práci se síťovými prvky a práci se silnicemi a chodníky. Tyto 3 oblasti popišme zvlášť.

- Práci s posloupností Bézierových křivek reprezentovanou třídou `BezierSpline` zajišťuje třída `BezierSplineEditor`. Ta jednak vykresluje samotnou křivku a její kontrolní body a jednak umožňuje s těmito body manipulovat. Konkrétně umožňuje posunování, přidávání a odebírání kontrolních bodů a podrozdělování křivky na 2 části (pro detaily z uživatelského hlediska viz kapitola 7).

  Prostředníkem mezi `BezierSpline` a `BezierSplineEditor` je třída `SplineViewModel` implementující návrhový vzor Model-View-ViewModel (pro popis tohoto návrhového vzoru viz článek *The Model-View-ViewModel Pattern* od firmy Microsoft [56]). Zatímco `BezierSpline` (reprezentující Model) nabízí pouze nízkoúrovňové operace pro práci s kontrolními body (změnu pozice bodu, přidání bodů na konkrétní pozici, atd.), `SplineViewModel` (reprezentující ViewModel) nabízí ty operace, které uživatel v editoru opravdu provádí. Konkrétně uživatel provádí následující operace (z uživatelského hlediska je popíšeme v sekci 7.5):

  - Přidání nového segmentu, který uživatel specifikoval pozicí konečného opěrného bodu. Počáteční a konečný směrový bod se tedy vypočítají automaticky.
  - Posunutí bodu na novou pozici. Pokud je posunovaným bodem kotva, posunou se o stejnou vzdálenost i oba sousední směrové body. Pokud je posunovaným bodem směrový bod, posune se vhodným způsobem i sousední směrový bod.
  - Podrozdělení jedné Bézierovy křivky na dvě. Při této operaci je třeba určit pozici všech kontrolních bodů nového segment.

- Práci se síťovými prvky reprezentovanými potomky třídy `NetworkElement` zajišťuje třída `NetworkElementEditor`, která vykresluje polohu kotev síťového prvku. Navíc pokud herní objekt obsahuje jak komponentu `BezierSplineHolder`, tak komponentu `NetworkElementHolder` (to je případ silnic a chodníků), upravuje třída `NetworkElementViewModel` to, jak probíhá posouvání kotev Bézierových křivek. Konkrétně pokud se posouvaná kotva přiblíží ke kotvě jiného síťového prvku, naváže se spojení mezi kotvami ve smyslu sekce 3.2.1.

- Třída `StripEditor` vyvolává přepočítání meshe pásu (tj. silnice nebo chodníku) při změně parametrů daného pásu. Jejími potomky jsou `SidewalkEditor` a `RoadEditor`. Třída `RoadEditor` navíc zajišťuje přidávání chodníků vedle silnice pomocí speciálního tlačítka v okně Inspector.

Unity Editor navíc umožňuje objekty v herním světě posouvat, duplikovat a mazat. U síťového prvku provedení některé z těchto operací znamená nutnost přepočítat spojení, kterých se daný prvek účastní (viz následující sekce 5.8.1). Pro zjednodušení by se nabízelo např. duplikaci síťových prvků zakázat, nicméně to Unity Editor nepodporuje. Navíc naše zkušenost s editorem herního světa ukazuje, že všechny tyto 3 operace jsou užitečné – pokud chce totiž návrhář vytvořit dvě silnice se stejnými parametry, lze toho nejjednodušeji dosáhnout vytvořením jen jedné a její následnou duplikací. Proto v následující sekci 5.8.1 uvedeme, jak na tyto operace se síťovými prvky reagovat.

5.8.1 Posun, duplikace a mazání síťových prvků

Pokud nějaký síťový prvek v herním editoru posuneme, zduplikujeme, nebo smažeme, přestane se prvek účastnit všech dosavadních spojení. Např. pokud posuneme silnici, musí být správně vymazána její účast ve všech křižovatkách. Při kterékoli z těchto událostí tedy smažeme všechna spojení daného prvku, tj. jak serializované záznamy o spojení, tak i dynamické reference v křižovatkách. K tomu navíc:

- Při posunutí či smazání může odstranění serializovaných záznamů o spojení způsobit, že se některé křižovatky rozpadnou a bude potřeba přesměrovat serializovaná spojení i v ostatních prvcích – viz sekce 5.4.3.

- Při duplikaci se křižovatky nerozpadnou, protože se jich bude dál účastnit ten prvek, ze kterého byl duplikát vytvořen. V duplikátu je ale potřeba změnit GUID síťového prvku a všech jeho kotev – jinak by existovaly dva prvky nebo dvě kotvy se stejným GUID.

Bohužel Unity nevyvolává žádné události při posunu, duplikaci, nebo smazání herního objektu v editoru, ani nevolá žádnou speciální metodu, kterou bychom mohli implementovat. To považujeme za chybu v návrhu Unity Editoru. Pro každou z těchto 3 situací jsme tedy museli implementovat vlastní mechanismus, který ji detekuje. Každý z těchto mechanismů spoléhá mimo jiné na to, že komponenta `NetworkElementHolder` je označena jako `[ExecuteAlways]` (pro informace o tomto atributu viz sekce 4.3).

- Pro detekci duplikace využijeme speciální metodu `MonoBehaviour.OnValidate()`, kterou implementujeme na komponentě `NetworkElementHolder`. Tato metoda se zavolá mimo jiné při každé deserializaci našeho skriptu, která nastane i při duplikaci objektu. V těle této metody získáme informaci o aktuálně zpracovávané události pomocí `Event.current`. Pokud nějaká událost aktuálně zpracovávána je, zkontrolujeme, zda se jedná o duplikaci. Přitom je třeba ošetřit, že duplikaci lze v Unity Editoru provést dvěma způsoby: buď stiskem `Ctrl+D` a nebo `Ctrl+C` následovaný `Ctrl+V`. První událost je označována jako „Duplicate“, druhá jako „Paste“. Konkrétně tedy detekce duplikace ve třídě `NetworkElementHolder` bude vypadat následovně:

```
private void OnValidate() {
    Event currEvent = Event.current;
    if (currEvent != null &&
        currEvent.type == EventType.ExecuteCommand &&
        (currEvent.commandName is "Duplicate" or "Paste")) {
        // Duplication detected.
    }
}
```

- Pro detekci posunu využijeme pole `hasChanged` komponenty `Transform`, které se automaticky nastaví na `true`, pokud se některé parametry dané komponenty změnily. Dále využijeme toho, že komponenta `NetworkElementHolder` je označena jako `[ExecuteAlways]` a její metoda `Update()` je tedy volána při každé události v editoru (viz sekce 4.3). Celá detekce posunu ve třídě `NetworkElementHolder` pak bude vypadat následovně:

```
#if UNITY_EDITOR
private void Start() {
    transform.hasChanged = false;
}
private void Update() {
    if (transform.hasChanged) {
        transform.hasChanged = false;
        // Move detected.
    }
}
#endif
```

- Pro detekci smazání nelze jednoduše využít metodu `OnDestroy()`, protože ta se volá na každém herním objektu i např. při přechodu z Play módu do Edit módu. Lze ovšem využít metodu `OnDisable()` na editorovém skriptu `NetworkElementEditor`, která se volá, když herní objekt s komponentou `NetworkElementHolder` přestane být označen. To se, mimo jiné, stane, když takový objekt smažeme. Navíc tato metoda se volá až po tom, co je herní objekt smazán, takže v ní smazání objektu můžeme kontrolovat. Detekce smazání tedy v `NetworkElementEditor` vypadá takto:

```
private void OnDisable() {
    if (target.Equals(null)) {
        // Deletion of (INetworkElementHolder)target detected.
    }
}
```

5.9 Užitečná rozšíření Unity

Pro usnadnění programování v prostředí Unity jsme implementovali několik utilitních tříd, zejména týkajících se serializace. Tyto třídy jsou umístěny v adresáři `Utils`.

Uveďme ty nejdůležitější z utilitních tříd:

- Třídy pro serializaci typů, které výchozí serializační systém Unity serializovat nedokáže. Konkrétně:
  - `SerializableGuid` pro serializaci typu `System.Guid`,
  - `SerializableDict` pro serializaci typu `System.Collections.Generic.Dictionary`,
  - `SerializableDecimal` pro serializaci typu `decimal`,
  - `SerializableNullable` pro serializaci *nullable* hodnotových typů,
  - `SerializableNullable` pro serializaci hodnoty, která může nabývat jednoho ze dvou serializovatelných typů.
- Statická třída `MathUtils` pro matematické operace, jako např. náhodný výběr z exponenciálního či normálního rozdělení.
- Statická třída `GeometryUtils` pro práci se souřadnicovými systémy, mnohoúhelníky a přímkami.

5.9.1 Přidané ladící nástroje

Námi přidané vývojářské nástroje pro ladění hry jsou dostupné v horní liště Unity Editoru v nabídce *Traffic Tools* a ovládá je třída `CustomTools`. Konkrétně jsou v nabídce dostupné tyto nástroje:

- *Recompile all scripts* – zkompiluje všechny herní skripty a načte výsledné assembly, což způsobí reinicializaci všech komponent a odstranění všech neserializovaných dat,
- *Delete and reinitialize all junctions* – smaže všechna spojení mezi síťovými prvky a znovu je deserializuje ze serializovaných záznamů,
- *Print <component> debug info* – vypíše ladící informace o dané komponentě.

6. Uživatelská dokumentace – hráč

Traffic je hra pro mobilní zařízení se systémem Android. V této kapitole uvedeme návod k její instalaci a příručku pro hráče.

6.1 Instalace hry

Pro instalaci hry je v adresáři `Build` přílohy A připraven balíček `traffic.apk`. Samotná instalace probíhá v následujících krocích:

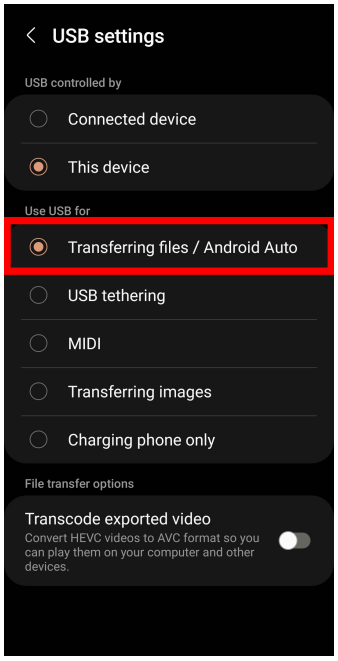
1. Pomocí USB kabelu připojíme mobilní telefon k počítači, na kterém je umístěný soubor `traffic.apk`.
2. Na mobilním telefonu přejdeme do nastavení *USB settings* a zvolíme položku *Transferring files* (*Přenos souborů*), jak je zachyceno na obrázku 6.1. Tím povolíme přenos souborů z připojeného počítače.
3. Na připojeném počítači zkopírujeme soubor `traffic.apk` do telefonu.
4. V libovolném prohlížeči souborů na mobilním telefonu vyhledáme zkopírovaný soubor `traffic.apk` a klikneme na něj, viz obrázek 6.2. V našem případě jsme použili aplikaci *My Files*.
5. Může se stát, že se zobrazí bezpečnostní upozornění týkající se instalace aplikace z neznámého zdroje. V takovém případě přejdeme v nastavení do sekce *Install unknown apps* (*Instalace z neznámých zdrojů*) a zaškrtneme políčko u námi používaného prohlížeče souborů, viz obrázek 6.3.
6. Tím dojde k instalaci aplikace, jejíž ikona se následně umístí na plochu mezi ostatní aplikace. Název aplikace je *Traffic*.

Pro uživatele zvyklé na používání terminálu dodejme, že instalaci lze provést pomocí nástroje `adb` – viz sekce *Install an app* oficiální dokumentace `adb` [57].

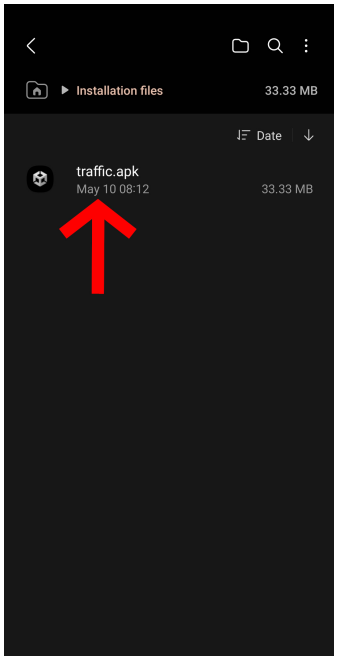
6.2 Úvodní menu

Po spuštění aplikace je hráči zobrazeno úvodní menu zachycené na obrázku 6.4. To obsahuje 4 důležité prvky:

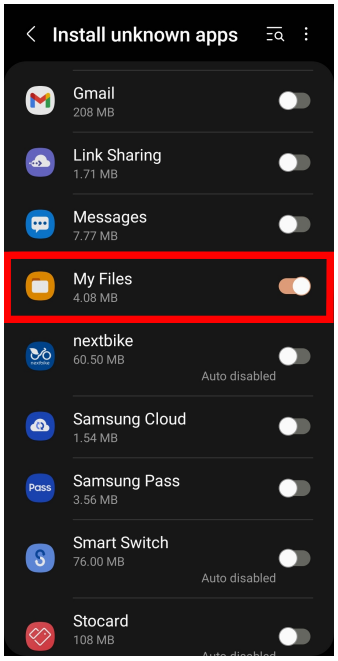
- Skóre, kterého hráč dosáhl. To sestává ze získaných peněz a hvězdičkového hodnocení – jak dosáhnout co nejlepšího skóre uvedeme dále.
- Tlačítko *PLAY*, jehož stisknutím se spustí samotná hra.
- Tlačítko *RESET*, jehož stisknutím se hráči zobrazí dialog pro vyresetování stavu hry zachycený na obrázku 6.5. Pokud hráč potvrdí vyresetování stiskem tlačítka *YES*, je vynulováno jeho skóre a jeho pozice v herním světě.
- Tlačítko *QUIT*, jehož stisknutím se aplikace ukončí.



Obrázek 6.1: Povolení přenosu souborů mezi počítačem a telefonem.



Obrázek 6.2: Instalační soubor v prohlížeči souborů na telefonu.



Obrázek 6.3: Povolení instalace aplikací z neznámých zdrojů.

Obrázek 6.4: Úvodní menu hry.

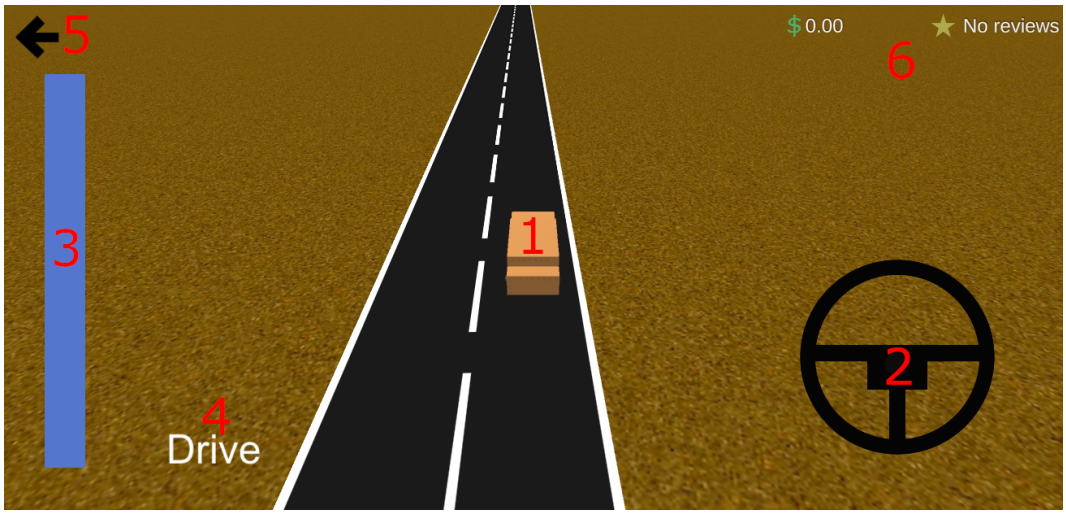
6.3 Princip hry

Po spuštění samotné hry je hráč přesunut do herního světa a zobrazí se mu ovládací prvky. Po celou dobu hry vidí hráč z *third-person* pohledu auto, které může ovládat. Cílem hry je jezdit po silnicích a pracovat jako taxikář, tedy převážet lidi na jejich destinace. To popíšeme podrobnější dále – nejprve v následující sekci 6.4 vysvětlíme, jak auto ovládat.

Obrázek 6.5: Dialog pro resetování hráčova skóre.

6.4 Řízení auta

V celé této sekci sledujme obrázek 6.6 zachycující pohled uživatele po spuštění hry.

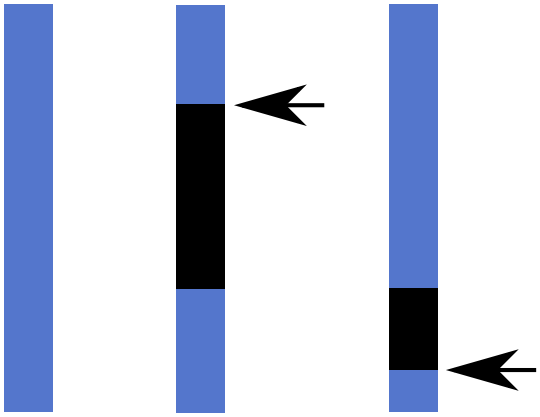


Obrázek 6.6: Pohled hráče po spuštění hry. Nejdůležitější ovládací prvky jsou očíslovány.

Pro řízení auta (číslo 1 na obrázku 6.6) má hráč k dispozici následující 3 ovládací prvky. Ty přibližně odpovídají hlavním ovládacím prvkům v reálném autě s automatickou převodovkou.

1. Volant (číslo 2 na obrázku 6.6) umístěný v pravém dolním rohu obrazovky, jehož otočení určuje orientaci předních kol auta. Tažením prstu v pravé polovině obrazovky lze volantem otáčet. Konkrétně se volant otočí o úhel, který svírá počáteční a koncový bod tažení při pohledu ze středu volantu. Manipulace s volantem dále od jeho středu je tedy pomalejší a přesnější, zatímco manipulace blíže k jeho středu je rychlejší. Pokud hráč volant nedrží (tedy zvedne prst, kterým volant ovládá, z obrazovky), vrací se volant samovolně do své výchozí pozice.

2. Plyn a brzda jsou spojeny do jednoho ovládacího prvku v podobě modrého sloupce na levé straně obrazovky (číslo 3 na obrázku 6.6). Pokud se hráč dotkne obrazovky v libovolném místě levé části obrazovky, vyplní se sloupec černou barvou až do výšky, ve které došlo k dotyku. Přitom je sloupec rozdělen na horní a dolní část, kde výplň horní části udává míru zmáčknutí plynu a výplň spodní části udává míru zmáčknutí brzdy. Příklady možného stavu sloupce sledujme na obrázku 6.7 – vlevo je stav před prvním dotykem hráče, uprostřed po přidání plynu a vpravo po sešlápnutí brzdy.



Obrázek 6.7: Ovládání plynu a brzdy, kde šipky ukazují vertikální souřadnici posledního dotyku. Zleva: není sešlápnut plyn ani brzda, je sešlápnutý plyn a je sešlápnutá brzda.

3. Přepínání směru jízdy (dopředu a dozadu) je umožněno stisknutím tlačítka v levé dolní části obrazovky (číslo 4 na obrázku 6.6). To má nápis *Drive*, pokud je zvolen směr dopředu a nápis *Reverse*, pokud je zvolen směr dozadu.

Dále může hráč přejít zpět do úvodního menu hry pomocí šipky v levém horním rohu obrazovky (číslo 5 na obrázku 6.6), čímž nedojde ke ztrátě jeho postupu. Aktuální stav peněz a hvězdičkového hodnocení je pak uveden v pravém horním rohu obrazovky (číslo 6 na obrázku 6.6).

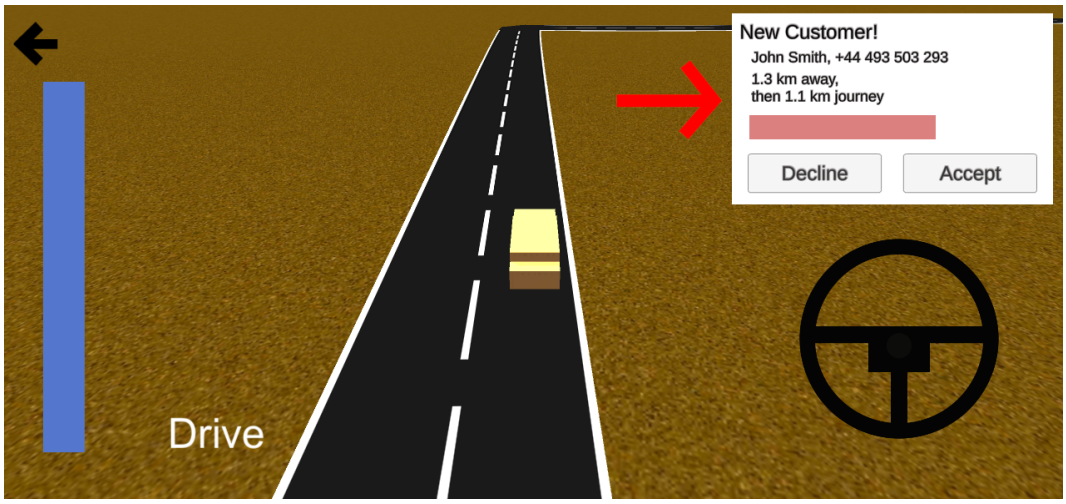
6.5 Herní mise

Cílem hry je převážet zákazníky na jejich destinace a sbírat tak peníze a co nejlepší hvězdičkové hodnocení. Zákaznící, tedy chodci, jsou ve hře zobrazeni jako žluté válce. Vyřízení jedné zakázky probíhá takto:

1. Nejprve hráč čeká, než zavolá nový zákazník požadující přepravu. V tomto mezičase může prozkoumávat herní svět.

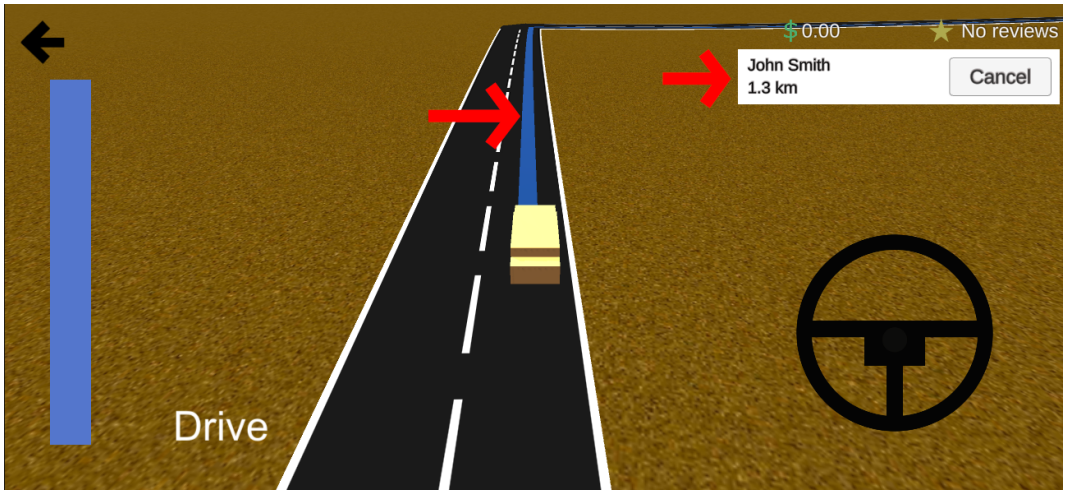
2. Nabídka nového zákazníka se po určité chvíli objeví v pravém horním rohu obrazovky, jak je zachyceno na obrázku 6.8. Součástí nabídky je jméno a telefonní číslo zákazníka, vzdálenost k zákazníkovi a vzdálenost od zákazníka k jeho destinaci. Hráč má možnost nabídku přijmout pomocí tlačítka *Accept* nebo odmítnout pomocí tlačítka *Decline*. Na rozhodnutí je vymezen určitý čas, přičemž zbývající čas je vyznačen červeným obdélníkem nad

tlačíky. Tento obdélník se postupně zmenšuje a jakmile zmizí, nabídka se automaticky odmítne. Pokud je nabídka odmítnuta, čeká hráč na dalšího zákazníka.



Obrázek 6.8: Nabídka nového zákazníka v pravém horním rohu obrazovky.

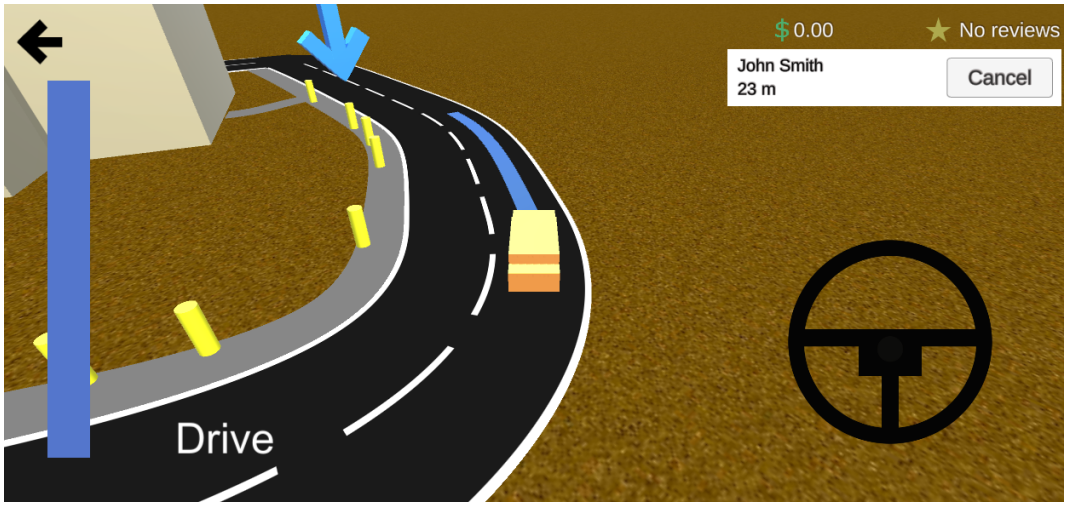
3. Při přijmutí zakázky se do pravého horního rohu obrazovky umístí informační panel se jménem zákazníka a zbývající vzdáleností k zákazníkovi. Zároveň se na silnici před hráčem zobrazí modrá navigační čára vyznačující nejkratší cestu k zákazníkovi. Oba tyto prvky jsou zachyceny na obrázku 6.9. Pokud hráč vyjede pryč ze silnice, navigační čára zmizí a objeví se až po opětovném najetí na silnici.



Obrázek 6.9: Stav po přijetí zákazníka. Zobrazen je informační panel a navigační čára.

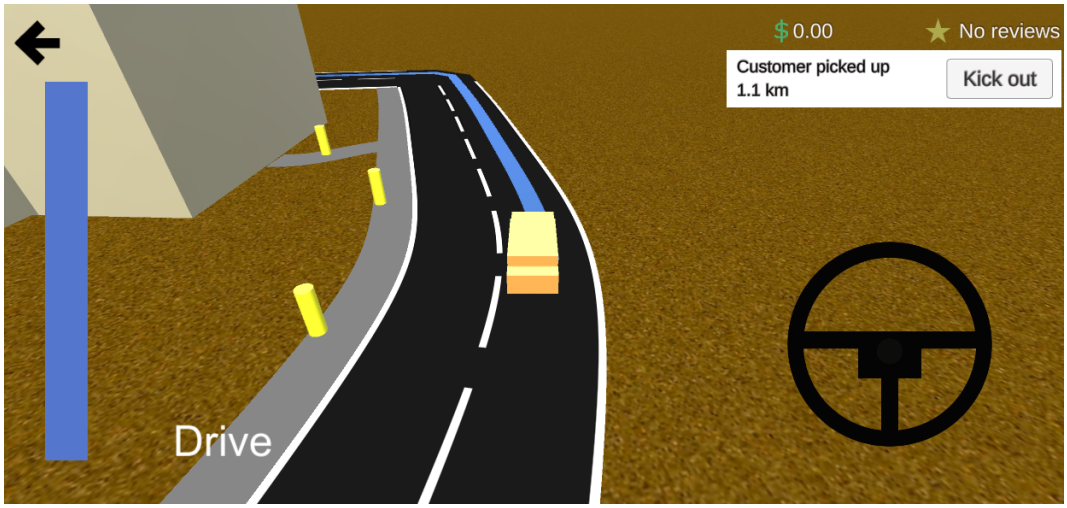
Navigační čára vede hráče tak, aby se nemusel otáčet uprostřed silnice. To znamená, že pokud jede hráč na druhou stranu, než by měl, může si neuposlechnutím navigace a otočením se uprostřed silnice zkrátit cestu. To odpovídá chování skutečných navigací, jako např. Google Maps.

Nad čekajícím zákazníkem je navíc umístěna modrá šipka, aby jej hráč mohl snáze identifikovat. To je zachyceno na obrázku 6.10.



Obrázek 6.10: Čekající zákazník označený modrou šipkou.

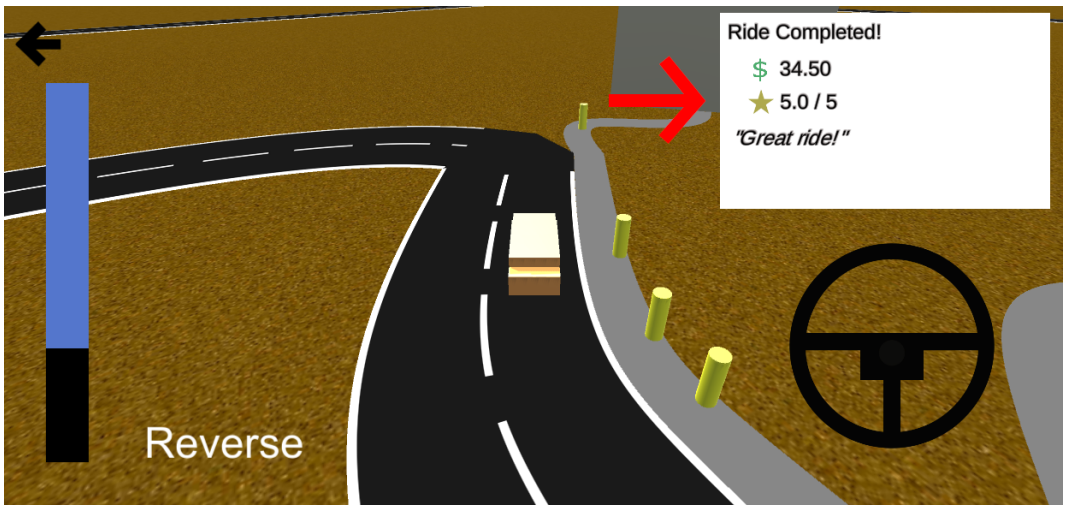
4. Jakmile hráč zastaví v blízkosti čekajícího zákazníka, nastoupí zákazník do auta. Následně se zobrazí informační panel obsahující informaci o vzálenosti k zákazníkově destinaci a tlačítko umožňující zákazníka vyhodit z auta a tím zakázku zrušit. Navíc navigační čára začne ukazovat nejkratší cestu k zákazníkově destinaci. Tento stav je zachycen na obrázku 6.11.



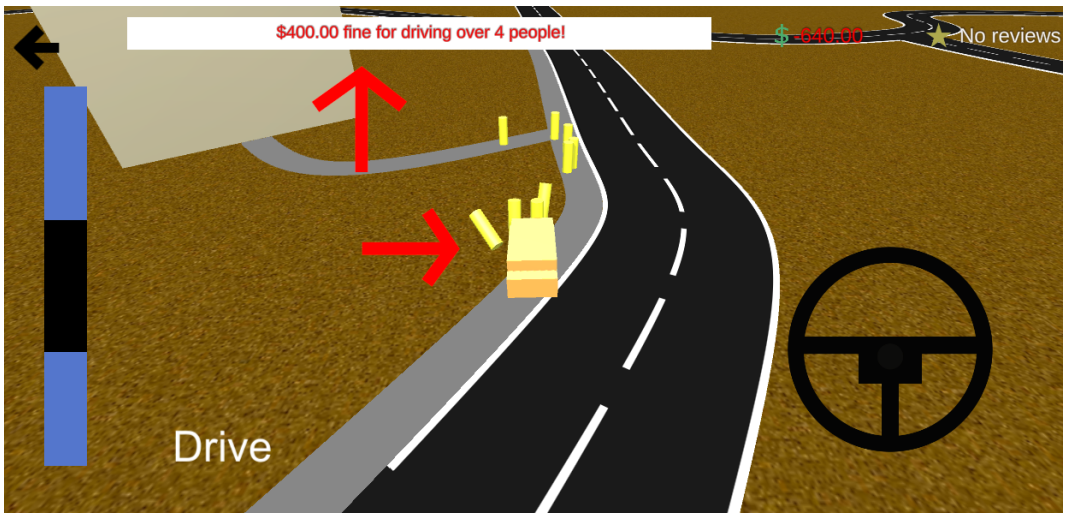
Obrázek 6.11: Stav po naložení zákazníka.

5. Jakmile hráč s naloženým zákazníkem zastaví v blízkosti zákazníkovi destinace, vystoupí zákazník z auta. Následně dostane hráč peníze a slovní a hvězdičkové hodnocení, které se odvíjí od toho, jak rychlá a zároveň pohodlná byla ujetá trasa. Informace o získaných penězích a hodnocení se zobrazí v pravém horním rohu obrazovky, jak je zachyceno na obrázku 6.12. Následně čeká hráč na dalšího zákazníka.

Pokud hráč v průběhu mise vyjede ze silnice nebo narazí do chodce, dostane pokutu, která se mu strhne z vydělaných peněz. To je zachyceno na obrázku 6.13, kde hráč narazil do 4 chodců.



Obrázek 6.12: Vyložený zákazník a informační panel s hodnocením jízdy.



Obrázek 6.13: Pokuta za srážku s chodci.

7. Uživatelská dokumentace – herní návrhář

Herní svět hry Traffic je možné editovat bez znalosti programování. Za tímto účelem jsme rozšířili Unity Editor o funkce pro práci se silniční sítí a obytnými domy. V této kapitole uvedeme návod k použití těchto rozšiřujících funkcí.

7.1 Otevření herního světa

Pro editaci herního světa je potřeba nejprve nainstalovat Unity, což jsme popsali v sekci 5.1. Součástí instalace je Unity Editor, po jehož spuštění je možné otevřít projekt Traffic, který je součástí přílohy A. Po načtení projektu je nutné v okně *Assets* přejít do adresáře `Assets/Scenes` a dvojitým kliknutím otevřít některou ze scén, jejíž jméno končí slovem *Level*.

Konkrétně je připravena scéna `DefaultLevel` s příkladem hotové silniční sítě (jejíž mapa je uvedena v příloze B) a scéna `EmptyLevel` s prázdným herním světem. Návrhář si tak může vybrat, zda editovat již hotový svět, nebo vytvořit zcela nový svět. Pokud by chtěl pracovat na více než dvou světech, může kteroukoli z připravených scén zduplikovat klávesovými zkratkami `Ctrl+C` a `Ctrl+V`, přičemž pro zduplikovanou scénu zvolí nový název. Zduplikovanou scénu je dále potřeba přidat do projektu tak, že otevřeme nastavení *File > Build Settings* a přetáhneme ji do seznamu *Scenes In Build*.

V samotné hře nemá hráč možnost si z vytvořených světů vybrat, takže tuto volbu musí provést herní návrhář. K tomu je nutné otevřít scénu `EntryMenu`, vybrat v okně `Hierarchy` objekt `Canvas > MainMenu` a následně v okně `Inspector` upravit hodnotu *Level Scene*. Do této hodnoty je nutné vyplnit jméno scény, která se má použít jako herní svět – tedy např. *„DefaultLevel“* nebo *„EmptyLevel“*.

Pokud dojde tímto způsobem ke změně výběru herního světa, je potřeba vyresetovat perzistentně uložená data. Ta se totiž vztahují k původně zvolenému světu. K tomu je nutné otevřít nově vybranou scénu, vybrat v okně `Hierarchy` objekt `GameManager` a následně v okně `Inspector` stisknout postupně tlačítka *Remove all saved data* a *Bake navigation mesh*.

Po otevření kteréhokoli herního světa jsou rozšiřující nástroje pro práci se silniční sítí automaticky dostupné. Návod k jejich použití bude předmětem následujících sekcí. Pro úvod do základních funkcionalit Unity Editoru viz oficiální tutoriál Explore the Unity Editor [58].

Rozšířený Editor jsme používali na notebooku s procesorem Intel Core i5-5200U a operační pamětí 8 GB, přičemž práce s ním byla ve většině případů dostatečně plynulá. Pokud by generování silniční sítě práci návrháře příliš zpomalovalo, lze pro nápravu využít postup uvedený v sekci 7.10.

Po editaci herního světa lze projekt znovu sestavit, což jsme popsali v sekci 5.2.1. Ve výsledném programu pak budou provedené změny přítomny.

7.2 Orientace ve struktuře scény

Pro editaci herního světa se stačí zaměřit pouze na malou část hierarchie objektů zobrazené v okně `Hierarchy`. Konkrétně jsou důležité následující objekty (pro výčet a vysvětlení všech objektů viz 5.2):

- `Car` – Pozice tohoto objektu určuje výchozí pozici hráčova auta. Aby se změna pozice projevila, je potřeba smazat perzistentně uložený stav hry, viz sekce 7.9.
- `GameManager` – Tento objekt umožňuje mimo jiné zmíněné smazání perzistentního stavu, viz opět sekce 7.9.
- `NetworkElements` – Potomky tohoto objektu jsou automaticky všechny vytvořené silnice, chodníky a domy.

7.3 Přidání nové silnice nebo chodníku

Pro přidání nové silnice či chodníku jsou možné následující 2 postupy:

- Duplikace již existující silnice či chodníku – stačí označit silnici či chodník určený k duplikaci a stisknout `Ctrl+D`. Duplikovaný objekt se vytvoří přesně na objektu původním, takže je následně potřeba duplikovaný objekt přesunout.
- Vytvoření zcela nové silnice či chodníku – silnice se vytvoří stiskem `Ctrl+W`, chodník stiskem `Ctrl+T`. Nový objekt je po vytvoření automaticky označen. Křivka určující jeho tvar je však zatím prázdná (obsahuje jen jeden kontrolní bod) a tudíž nemá daná silnice či chodník žádný mesh. Proto není vidět, pokud není označena. Její vzhled při označení je zachycen na obrázku 7.1, kde je vidět kotva a nástroj pro posouvání silnice (proč je kotva vykreslena bílo-modře vysvětlíme v následující sekci 7.5). Pokud návrhář omylem zruší označení nové silnice či chodníku předtím, než přidá aspoň jeden kontrolní bod, je třeba tuto silnici či chodník znovu označit v okně `Hierarchy`.

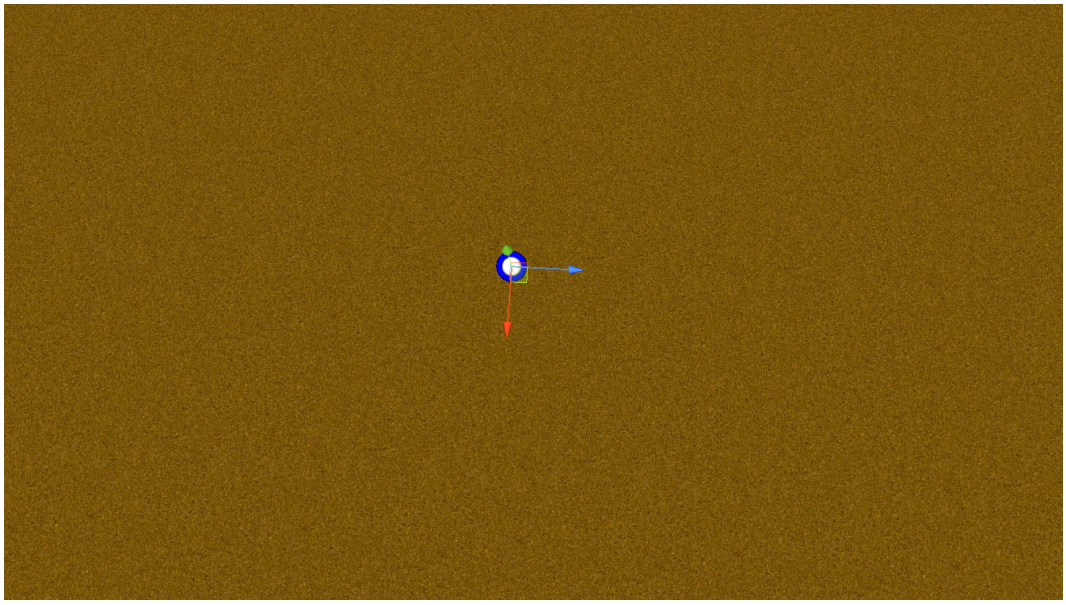
Označenou silnici či chodník lze odstranit stiskem klávesy `Delete`.

7.4 Změna parametrů objektů

Editace nastavitelných parametrů objektu (silnice, chodníku, obytného domu) probíhá v okně `Inspector`, pokud je daný objekt označen. Zaměřme se na parametry silnic a chodníků, parametry obytných domů uvedeme v sekci 7.11.

Konkrétně má každá silnice a chodník tyto nastavitelné parametry:

- Výšku (*Height*) a šířku (*Width*) vygenerovaného meshe.
- Parametr *Texture Stretch*, který určuje, jak moc je výsledná textura silnice či chodníku „natažena“ v podélném směru. Změna tohoto parametru má význam pouze v případě silnic, kde na něm závisí délka přerušovaných čar na středu silnice.



Obrázek 7.1: Vzhled silnice či chodníku ihned po jeho přidání.

- Parametr *Sampled Offset Points Granularity*, který určuje hustotu bodů nanesených na křivku při generování meshe (pro technické detaily viz sekce 3.2.3). Čím je tento parametr vyšší, tím přesnější je vygenerovaný mesh, ovšem za cenu *vyšší* náročnosti při vykreslování.

- Parametr *Approx Gap Between Offset Points*, který určuje vzdálenost, do které se body nanesené na křivku při generování meshe umístí (pro technické detaily viz sekce 3.2.4). Čím je tento parametr *nižší*, tím přesnější je vygenerovaný mesh, ovšem za cenu vyšší náročnosti při vykreslování.

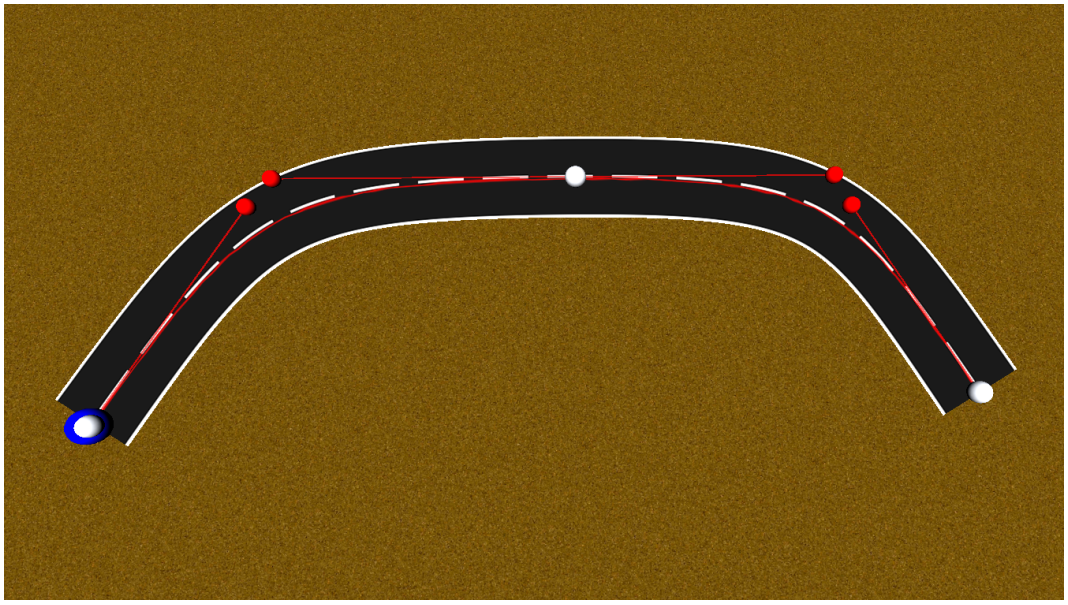
7.5 Práce s tvarem silnice nebo chodníku

Pro práci s danou silnicí či chodníkem je třeba jej nejprve označit – to buď v okně `Scene`, nebo v okně `Hierarchy`. Tím se červeně vykreslí křivka udávající tvar daného objektu spolu s jejími kontrolními body, přičemž kotvy křivky mají bílou barvu. Navíc kotva, která je označena jako konec silnice, je označena modrým okrajem. To zachycuje obrázek 7.2, na kterém je označená silnice se třemi bílými kotvami, z nichž jedna je označena modře.

Aby byly všechny zmíněné prvky správně zobrazeny, je třeba dbát na to, aby v liště v horní části okna `Scene` byla zvolena možnost *Toggle visibility of Gizmos* (což je výchozí stav).

Následně je možné provádět následující operace. Při tom je důležité, aby byla označena pouze jedna silnice či chodník.

- Přidání nové kotvy na konec silnice či chodníku – k tomu je nutné držet stisknutou klávesu `Ctrl` a levým tlačítkem myši kliknout na určité místo na zemi. Na tomto místě se objeví nová kotva, přičemž poloha dvou nových směrových bodů se určí automaticky. Nový segment křivky se připojí za koncovou kotvu původní křivky, tedy za tu, která měla okolo bílé značky modrý okraj.



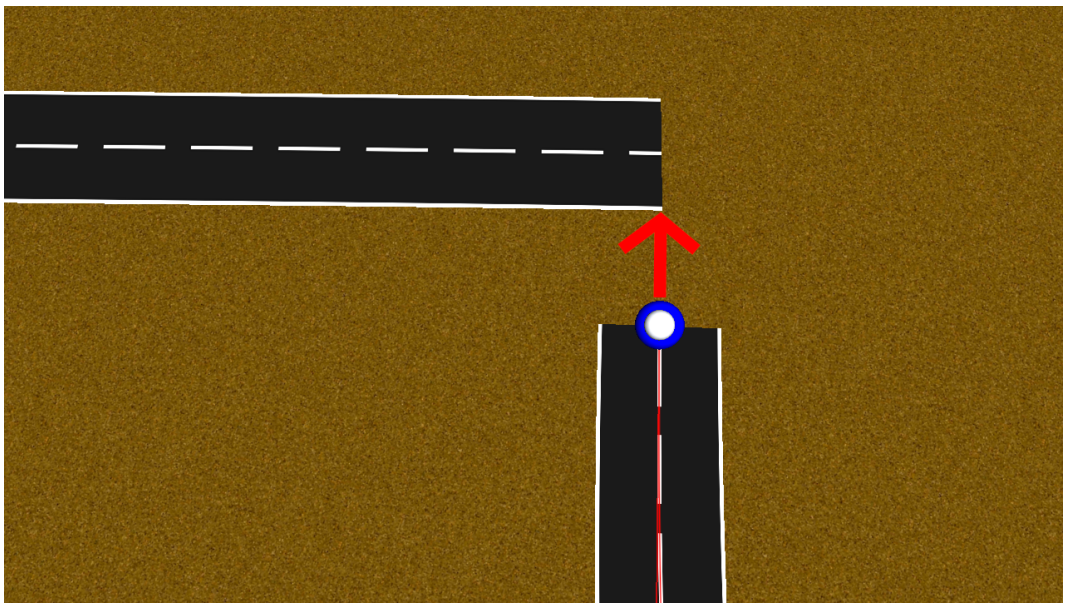
Obrázek 7.2: Vzhled silnice se třemi kotvami, pokud je označena.

- Odebrání kotvy – k tomu je nutné držet stisknutou klávesu `Shift` a levým tlačítkem myši kliknout na kotvu, kterou chceme odebrat.
- Rozdělení segmentu na dva – k tomu je nutné držet stisknutou klávesu `Ctrl` a levým tlačítkem myší kliknout na segment, který chceme podrozdělit. Tím dojde k přidání nové kotvy na místo určené myší, přičemž poloha dvou nových směrových bodů se určí automaticky.
- Prohození konců silnice, které je potřeba provést, pokud chceme přidat segment na tu stranu křivky, která aktuálně není modře označena. K tomu je nutné levým tlačítkem myší dvojitě kliknout na aktuálně neoznačený konec.
- Souřadnice kontrolních bodů silnice či chodníku je možné zadat manuálně, což však doporučujeme pouze zkušenějším uživatelům. To je vhodné např. při tvorbě kruhových úseků, kdy polohu kontrolních bodů vypočítáme a následně se chceme vyhnout nepřesnostem vycházejícím z posouvání kontrolních bodů myší. Pro zadání kontrolních bodů označené silnice je nutné přejít do okna `Inspector` a v komponentě `Spline Holder` otevřít nabídku `Element > Points`. Tím se zobrazí seznam souřadnic kontrolních bodů, který je možné editovat. Při tom je třeba dbát na to, že tyto souřadnice jsou lokální pro daný objekt (pro informace o lokálních souřadnicích viz sekce 4.1.1). Pokud chceme, aby lokální a globální souřadnice splývaly, stačí v okně `Inspector` vyresetovat komponentu `Transform`.

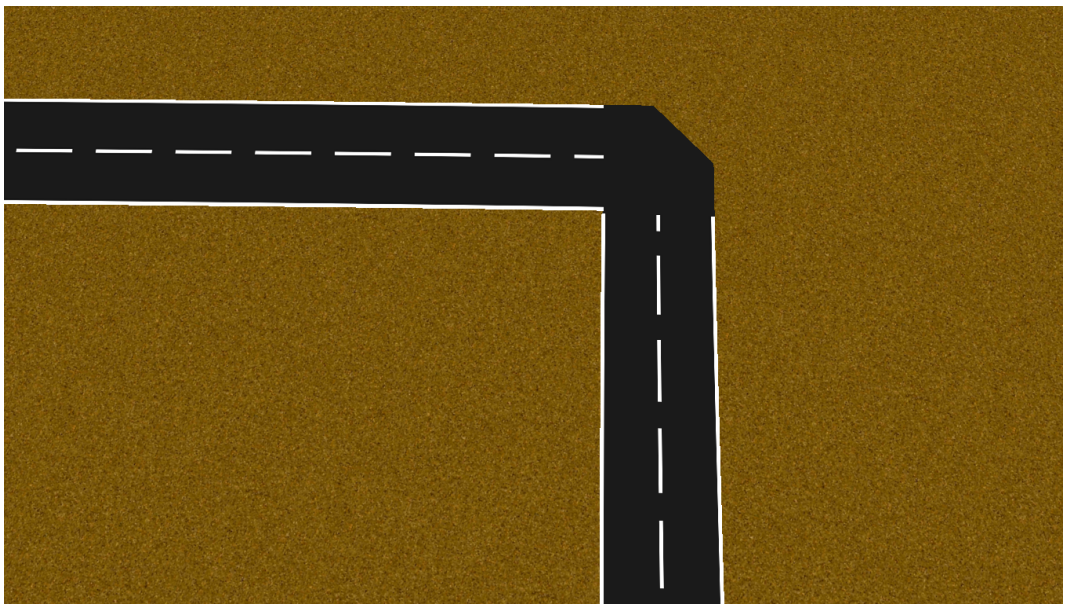
7.6 Propojení dvou nebo více silnic či chodníků

Spojování silnic a spojování chodníků probíhá zcela stejně, proto popíšeme pouze spojování silnic. Pro spojení dvou silnic je nutné kotvu jedné silnice posunout do blízkosti kotvy druhé silnice. Při dosažení dostatečné blízkosti posu-

novaná kotva „skočí“ na polohu druhé kotvy (tomuto procesu se říká *snapping*). Tím jsou kotvy spojeny a automaticky se vytvoří křižovatka. Tento proces je zachycen na obrázcích 7.3, kde kotvou jedné silnice pohybujeme směrem ke kotvě druhé silnice a 7.4, kde jsou již obě silnice spojeny.



Obrázek 7.3: Proces spojování dvou silnic, kdy kotvou jedné silnice posouváme ke kotvě druhé silnice.

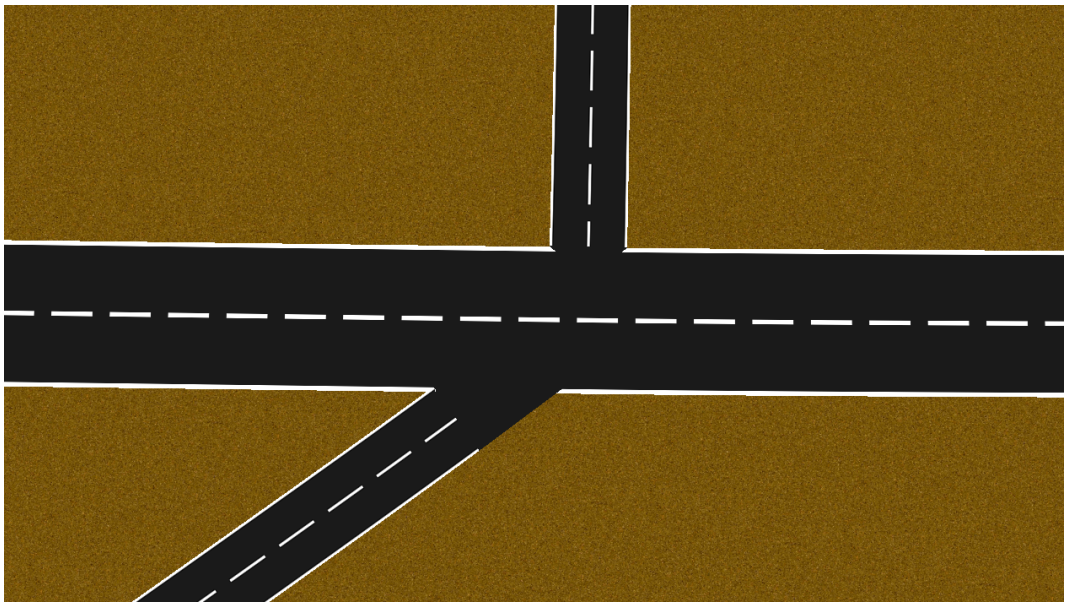


Obrázek 7.4: Dvě silnice spojené přes své koncové body.

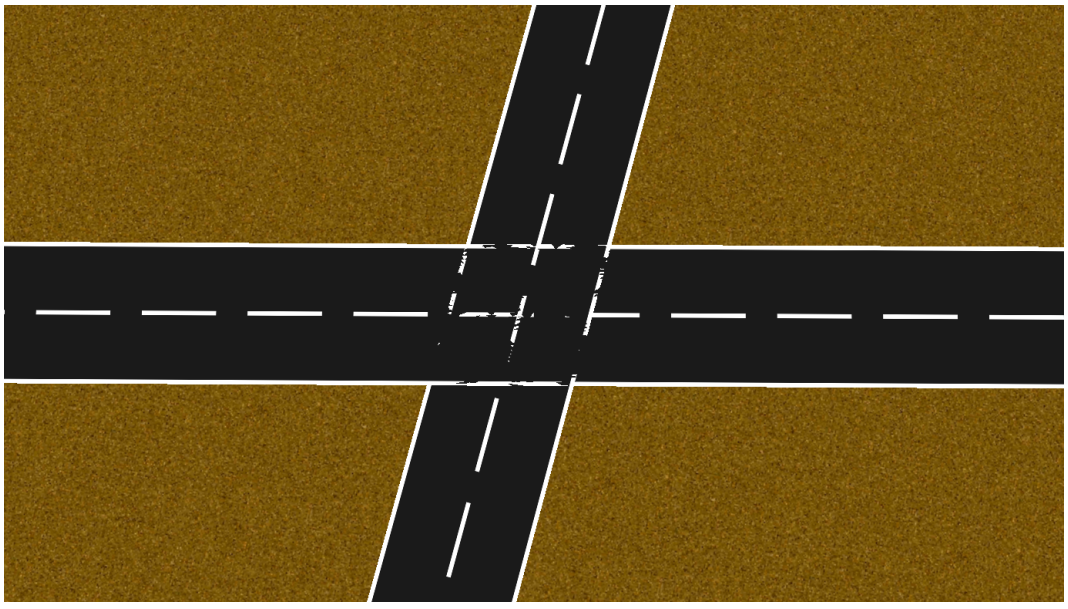
Dalším tažením jedné z kotev lze obě kotvy znovu rozpojit, čímž dojde ke zrušení křižovatky. Naopak analogickým přidáním dalších silnic vznikne křižovatka spojující více než 2 silnice.

Při spojování silnic je třeba dbát na to, aby všechny spojované silnice, s možnou výjimkou jedné z nich, byly napojeny za některý ze svých koncových bodů.

Příklad validní křižovatky, kde právě jedna silnice není napojena svým koncovým bodem, je uveden na obrázku 7.5 (konkrétně není napojena svým koncovým bodem silnice vedoucí na obrázku vodorovně). Naopak příklad nevalidní křižovatky spojující nekoncové body dvou silnic je uveden na obrázku 7.6, kde mesh obsahuje nepřirozené artefakty. Stejná situace nastane, pokud se 2 silnice překrývají jinak než pomocí popsaného procesu spojení.

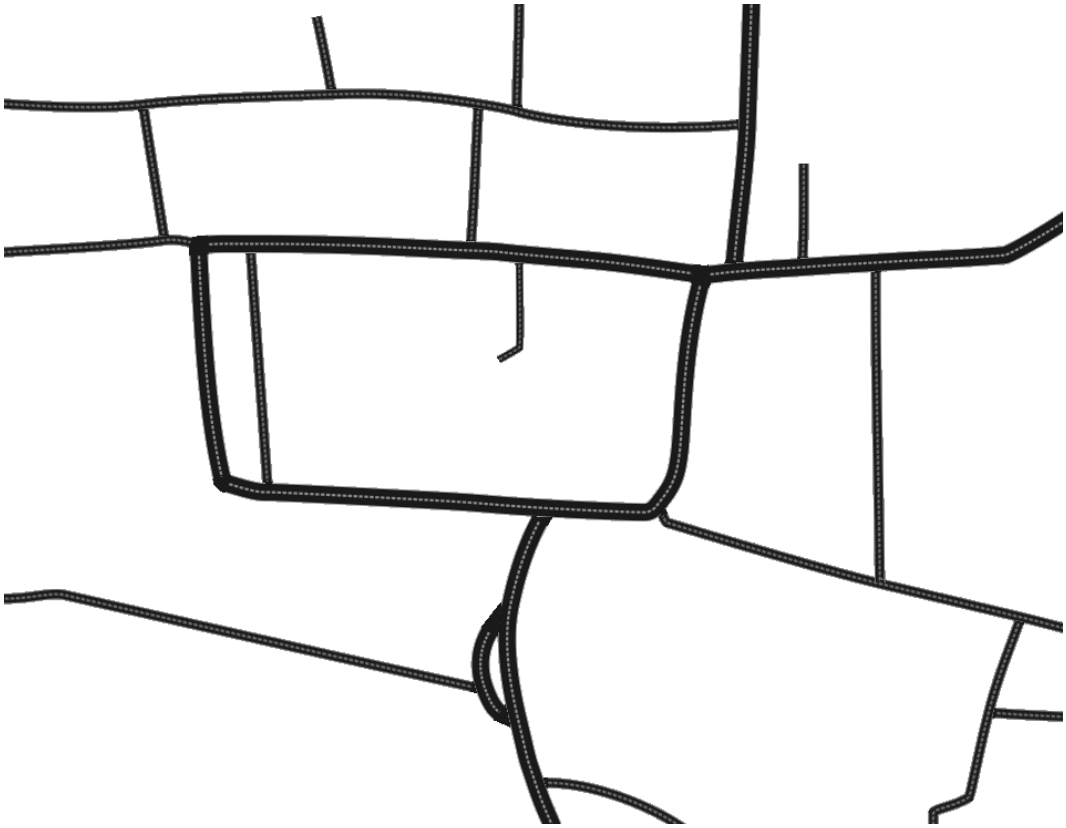


Obrázek 7.5: Křižovatka spojující tři silnice, z nichž jedna není napojena za svůj koncový bod.



Obrázek 7.6: Dvě překrývající se silnice, které nebyly explicitně spojeny přes své kotvy do křižovatky.

Příklad silniční sítě, kterou lze uvedenými operacemi vytvořit, je zachycen na obrázku 7.7. Konkrétně jde o napodobeninu skutečné silniční sítě části Malé Strany, která byla uvedena na obrázku 3.1.



Obrázek 7.7: Skutečná silniční síť z obrázku 3.1 vymodelovaná v našem editoru.

7.7 Komponenta `Transform` chodníků a silnic

Jako každý jiný herní objekt mají i silnice a chodníky komponentu `Transform`. Velmi podstatné je, že není podporována rotace ani zvětšování silnic a chodníků pomocí této komponenty. V případě, že ji návrhář omylem provede, nebude daná silnice či chodník fungovat správně. Naopak využití komponenty `Transform` k posouvání je u silnic a chodníků plně podporováno. Pokud se předtím daná silnice či chodník účastnil nějaké křižovatky, bude po posunutí z křižovatky odpojen.

7.8 Přidání chodníku podél silnice

Chodník může být umístěný podél silnice – poté je jeho tvar určený tvarem dané silnice. Pro přidání chodníku podél označené silnice je nutné v okně `Inspector` vyhledat komponentu `Road Holder` a v ní stisknout tlačíko *Add Right Sidewalk* nebo *Add Left Sidewalk* podle toho, na kterou stranu silnice chceme chodník přidat.

Smazat chodník vedoucí podél silnice lze pak standardním způsobem pomocí klávesy `Delete`. Pokud se v průběhu existence chodníku změní šířka jeho určující silnice, je potřeba chodník smazat a znovu vytvořit.

Při spojování silnic, podél kterých vedou chodníky, je třeba dbát na to, že tvar chodníků se spojením nezmění a chodník tak může „vylézt“ zpod křižovatky naproti spojované silnici. To se však děje pouze u velmi ostrých zatáček a je úkolem návrháře, aby tomuto předešel.

7.9 Reset navigačního meshe a uložených dat

Při jakékoli změně tvaru či pozice silnic, chodníků a domů a při změně výchozí pozice auta je potřeba provést následující 2 operace:

- Přepočítání navigačního meshe, který používají pro svůj pohyb chodci.
- Smazání všech perzistentně uložených dat, která obsahují mimo jiné pozici všech chodců a auta.

Obě tyto operace lze provést v okně `Inspector` při označení objektu `GameManager`. Konkrétně jde o tlačítka *Bake navigation mesh* a *Remove all saved data*.

7.10 Řešení neplynulé práce s křižovatkami

Pokud začne být manipulace se silnicemi a křižovatkami příliš neplynulá, je to pravděpodobně tím, že se meshe silnic a křižovatek negenerují dostatečně rychle. V takovém případě je možné generování těchto meshů dočasně vypnout, pracovat pouze s podkladovými křivkami a následně generování opět zapnout. Toho lze docílit vypnutím, resp. zapnutím komponenty `Road Holder` odpovídajících silnic, k čemuž slouží zaškrtávací pole v okně `Inspector` u této komponenty.

7.11 Vytvoření obytného domu

Přidání obytného domu lze docílit podobně jako přidání silnice či chodníku, tj. buď duplikací již existujícího domu, nebo vytvořením nového. Pro vytvoření nového domu slouží klávesová zkratka `Ctrl+M`, po jejímž stisknutí se nový dům objeví v herní scéně. Jeho výchozí texturu a mesh lze upravit pomocí standardních Unity komponent `Mesh Renderer` a `Mesh Filter`.

Obytnému domu lze přiřadit libovolné množství dveří, přičemž ke každým dveřím může být následně připojena kotva chodníku stejným procesem, jakým se spojují silnice nebo chodníky. Není však povoleno napojit na jedny dveře více chodníků. U dveří pak bude v průběhu hry docházet k tvorbě a zániku chodců, jak jsme popsali v sekci 3.3.1. Pro přidání dveří k domu je nutné označit daný dům a znovu stisknout zkratku `Ctrl+M`. Tím vznikne potomek objektu reprezentujícího daný dům, který bude reprezentovat nové dveře a bude mít přiřazenu komponentu `Person Hub Holder`.

Vzniklý objekt reprezentující dveře lze posouvat a následně k němu napojit chodník. Jakmile je však k němu napojen chodník, není již povoleno jej posouvat. Dále je u něj možné nastavit parametry týkající se generování chodců. Konkrétně lze nastavit tyto parametry:

- *Spawn Persons* – Zda se z těchto dveří budou generovat chodci.
- *Mean Seconds Between Spawns* – Střední hodnota exponenciálního rozdělení, kterým se řídí intervaly mezi vzniky jednotlivých chodců.
- *Min Seconds Between Spawns* – Nejkratší povolený interval mezi vznikem dvou chodců.

- *Min Seconds Between Spawns* – Nejdelší povolený interval mezi vznikem dvou chodců.

7.12 Zotavení se z chyb v editoru

Pokud dojde při práci s rozšířeným Editorem k chybě, je často možné ji opravit. Uveďme nejčastější takové chyby a možné postupy vedoucí k jejich nápravě.

- Chybová hláška *„Infinite loop detected“* znamená, že křivka určující tvar silnice nebo chodníku má nepřirozený tvar – např. obsahuje smyčku nebo příliš ostrou zatáčku. V takovém případě je řešením změnit tvar dané silnice či chodníku.

- Chybová hláška *„The spline was probably updated while not connected with this Path“* znamená, že silnice nebo chodník byly upraveny, zatímco jejich komponenta `Network Element` nebyla aktivní. To by se při běžné práci s Editorem nemělo stát a řešením je danou silnici či chodník smazat.

- Mnoho dalších chybových hlášek týkajících se silnic a chodníků umožňuje to, že po dvojitém kliknutí na danou hlášku se dotyčná silnice či chodník zvýrazní v okně `Hierarchy`. To může být nápomocné při identifikaci příčiny chyby.

- Pokud dojde k závažné chybě, případně sérii chyb, může být řešením znovu zkompilovat a načíst všechny skripty. To lze provést v nabídce *Traffic Tools* stiskem položky *Recompile all scripts*.

7.13 Vrácení změny

Většinu změn provedených v Editoru lze vrátit zpět klávesovou zkratkou `Ctrl+Z`. Pokud je ale vrácenou změnou přidání či odebrání síťového prvku nebo přidání či odebrání kotvy některé křivky, je po vrácení změny nutné v nabídce *Traffic Tools* spustit akci *Delete and reinitialize all junctions*. Tím se smažou a znovu deserializují všechna spojení síťových prvků.

8. Závěr

V této práci jsme implementovali hru Traffic pro mobilní zařízení. Při jejím návrhu jsme vyšli z vize komplexní hry simulující řízení, ze které jsme v kapitole 2 vybrali vhodnou podmnožinu funkcionalit. Při tom jsme dbali na zachování základního principu hry a na to, aby výsledná hra byla zábavná. Dále při analýze a implementaci jsme se snažili o co nejvyšší rozšiřitelnost výsledného díla směrem k původní vizi.

Všechny požadavky uvedené v kapitole 2 se nám podařilo implementovat. Co do přiblížení se čtyřem aspektům naší vize uvedeným v sekci 1.1 jsme dosáhli následujících výsledů:

- *Otevřený herní svět* – Rozšířili jsme Unity Editor o podporu snadné editace silničních sítí, čímž jsme umožnili efektivní tvorbu herních světů. Pro tvorbu rozsáhlých herních světů je však ještě potřeba implementovat postupné načítání světa, jak uvedeme dále. Navíc v naší hře není hráč nijak penalizován, pokud nesleduje herní misi a „bezcílně“ projíždí silniční sítí. Poznamenejme, že hra neumožňuje hráči vybrat si z různých levelů – o tom, který herní svět bude hra obsahovat, rozhoduje herní návrhář.

- *Komplexita herního světa* – Díky vysoké volnosti při tvarování silnic a křižovatek jsme umožnili tvorbu komplexních a málo repetitivních silničních sítí.

- *Jednoduchost pro hráče* – Herní ovládací prvky a úvodní menu jsou jednoduché na pochopení. V průběhu herní mise je hráč naváděn informačními panely a je pro něj jednoduché určit, co po něm mise požaduje. To jsme ověřili testováním hry na několika dobrovolnících.

- *Zábavnost* – Na základě zpětné vazby od hráčů, kteří hru vyzkoušeli, se domníváme, že je hra zábavná. Z časových důvodů však nebylo možné rozhodnout, zda by hra hráče bavila i po delším hraní. Domníváme se, že na to by bylo nutné přidat do hry více interaktivních prvků a umožnit hráči zřetelněji postupovat k nějakému cíli – např. tím, že by za nasbírané peníze mohl vylepšovat své auto.

Kromě samotné hry považujeme za užitečnou i analýzu a implementaci algoritmů pro práci s Bézierovými křivkami, zejména co se týče jejich aproximace lomenou čarou s uniformně vzdálenými vrcholy (sekce 3.2.4) a jejího následného offsettingu (sekce 3.2.3) a decrossingu (sekce 3.2.6). Dále považujeme tuto práci za užitečný studijní materiál pro programátora, který začíná s vývojem her v Unity.

8.1 Budoucí rozšíření

Hra Traffic byla navržena s úmyslem dalšího budoucího rozšíření. Uveďme hlavní rozšíření, na které je hra do určité míry připravena:

- *Postupné načítání herního světa podle polohy hráče*

  Aby velikost herního světa nebyla limitována operační pamětí cílového zařízení, musí být v jeden okamžik načtena pouze ta část světa, ve které se hráč nachází. Postupné načítání a opětovné mazání částí světa je tedy nutné pro tvorbu velmi rozsáhlých herních světů. Náš systém ukládání, načítání a mazání silnic je na tento problém od základu připraven, což bylo předmětem kapitol 3.2.1 a 5.4.3.

- *Procedurální generování oblastí*

  Pro zjednodušení práce herního návrháře a zrychlení tvorby rozsáhlých světů by bylo možné jednotlivé oblasti světa procedurálně generovat. Po vygenerování by měl návrhář možnost oblast dále upravit. Při procedurálním generování silniční sítě je jedním z obtížných problémů generování tvaru křižovatek na základě tvaru silnic. To náš program plně podporuje, což bylo předmětem kapitol 3.2.7 a 5.4.4.

- *Kvalitnější audiovizuální prvky*

  Výsledná hra obsahuje velmi jednoduché textury, 3D meshe a zvuky (to jsme odůvodnili v sekci 2.4). Všechny tyto prvky jsou však ve hře přítomné a lze je tak v budoucnu nahradit kvalitnějšími verzemi za minimální úpravy kódu.

- *Přidání terénu*

  V aktuální verzi se hra odehrává na ploše, takže zajímavým rozšířením by bylo přidání terénu. To by umožnilo tvorbu údolí, kopců, řek a mostů. Za tímto účelem nabízí Unity speciální modul *Terrain*. Navíc veškerý kód pro práci s Bézierovými křivkami a tvarem silnic používá jako základní typ pro reprezentaci souřadnic strukturu `Vector3` a počítá s možností, že bude v budoucnu využita i její třetí souřadnice (naopak např. algoritmus pro generování tvaru křižovatek by ale musel být upraven).

- *Ostatní auta řízená hrou*

  Pro interaktivnější herní zážitek by budoucí verze hry mohly obsahovat autonomní auta, která by jezdila po silniční síti společně s hráčem. Realistická simulace takových aut je těžký problém, jak jsme uvedli v sekci 2.1. Jako první kroky k tomuto cíli obsahuje náš program algoritmy pro hledání nejkratších cest v silniční síti (viz sekce 5.4.3) a pro detekci toho, na které silnici se auto nachází (viz sekce 3.3.3). Dále je jasně oddělena fyzikální simulace auta od ovládacích prvků, takže je umožněno ovládat auto jinak než skrz tyto prvky.

- *Rozšíření podporovaných platforem*

  Díky použití herního enginu Unity je hra připravena na budoucí podporu většího množství platforem, jako např. systému iOS.

Seznam použité literatury

[1] StatCounter. Mobile operating system market share worldwide - march 2022. https://gs.statcounter.com/os-market-share/mobile/worldwide/, 2022. Navštíveno 25. 4. 2022.

[2] Codex Gamicus. Open-world video games. https://gamicus.fandom.com/wiki/Open-world_video_games, 2022. Navštíveno 24. 4. 2022.

[3] Apple Inc. iOS 15 – Apple developer. https://developer.apple.com/ios/. Navštíveno 28. 4. 2022.

[4] Google Developers. Using a game engine on Android. https://developer.android.com/games/engines/engines-overview, 2021. Navštíveno 25. 4. 2022.

[5] Defold Foundation. Writing code. https://defold.com/manuals/writing-code/. Navštíveno 25. 4. 2022.

[6] GameDev.tv Ben Tristem. Unity vs. Unreal: Which game engine is best for you? https://blog.udemy.com/unity-vs-unreal-which-game-engine-is-best-for-you/, 2021. Navštíveno 25. 4. 2022.

[7] Ariel Manzur Juan Linietsky and the Godot community. Navigation – Godot engine documentation. https://docs.godotengine.org/en/stable/classes/class_navigation.html. Navštíveno 25. 4. 2022.

[8] Unity Software Inc. Unity – manual: Navigation and pathfinding. https://docs.unity3d.com/2021.2/Documentation/Manual/Navigation.html, 2021. Navštíveno 27. 4. 2022.

[9] Ariel Manzur Juan Linietsky and the Godot community. VehicleWheel – Godot engine documentation. https://docs.godotengine.org/en/stable/classes/class_vehiclewheel.html. Navštíveno 26. 4. 2022.

[10] Unity Software Inc. Unity – scripting API: WheelCollider. https://docs.unity3d.com/2021.2/Documentation/ScriptReference/WheelCollider.html, 2021. Navštíveno 26. 4. 2022.

[11] Institute of Transportation Engineers. Geometric design. https://www.ite.org/technical-resources/topics/geometric-design/, 2022. Navštíveno 23. 4. 2022.

[12] the free encyclopedia Wikipedia. Geometric design of roads. https://en.wikipedia.org/wiki/Geometric_design_of_roads, 2022. Navštíveno 23. 4. 2022.

[13] Wyoming Department of Transportation. Alignment and superelevation. https://www.dot.state.wy.us/files/live/sites/wydot/files/shared/Highway_Development/Surveys/Survey%20Manual/Appendix%

20D%20-%20Alignment%20and%20Superelevation.pdf, 2011. Navštíveno 23. 4. 2022.

[14] Vivek Tank. How to work with Bezier curve in games with Unity. https://www.gamedeveloper.com/business/how-to-work-with-bezier-curve-in-games-with-unity, 2018. Navštíveno 23. 4. 2022.

[15] Unity Software Inc. Unity – scripting API: EditorSceneManager.preventCrossSceneReferences. https://docs.unity3d.com/2022.2/Documentation/ScriptReference/SceneManagement.EditorSceneManager-preventCrossSceneReferences.html, 2022. Navštíveno 24. 4. 2022.

[16] Unity Software Inc. Unity – manual: Format of text serialized files. https://docs.unity3d.com/2021.2/Documentation/Manual/FormatDescription.html, 2021. Navštíveno 24. 4. 2022.

[17] William Armstrong. Spotlight team best practices: GUID based references. https://blog.unity.com/technology/spotlight-team-best-practices-guid-based-references, 2018. Navštíveno 24. 4. 2022.

[18] William Armstrong. Guid based reference. https://github.com/Unity-Technologies/guid-based-reference. Navštíveno 24. 4. 2022.

[19] Microsoft. Guid struct (System) | Microsoft docs. https://docs.microsoft.com/en-us/dotnet/api/system.guid. Navštíveno 5. 5. 2022.

[20] Pomax. A primer on Bézier curves. https://pomax.github.io/bezierinfo/, 2022. Navštíveno 16. 4. 2022.

[21] G. Elber, In-Kwon Lee, and Myung-Soo Kim. Comparing offset curve approximation methods. *IEEE Computer Graphics and Applications*, 17(3):62–71, 1997.

[22] Wayne Tiller and Eric G. Hanson. Offsets of two-dimensional profiles. *IEEE Computer Graphics and Applications*, 4(9):36–46, 1984.

[23] Wonjoon Cho Nicholas M. Patrikalakis, Takashi Maekawa. Shape interrogation for computer aided design and manufacturing, kapitola B-spline curve. https://web.mit.edu/hyperbook/Patrikalakis-Maekawa-Cho/node17.html, 2009. Navštíveno 27. 4. 2022.

[24] Blender Documentation Team. Introduction – Blender UV editor. https://docs.blender.org/manual/en/latest/editors/uv/introduction.html, 2022. Navštíveno 28. 4. 2022.

[25] M. Mareš and T. Valla. *Průvodce labyrintem algoritmů*. 1. vydání. CZ.NIC, Praha, 2017.

[26] Unity Software Inc. Experimental AI navigation package. https://forum.unity.com/threads/experimental-ai-navigation-package.1126961/, 2021. Navštíveno 22. 4. 2022.

[27] Unity Software Inc. Unity – scripting API: AI.NavMeshAgent.navMeshOwner. https://docs.unity3d.com/2021.2/Documentation/ScriptReference/AI.NavMeshAgent-navMeshOwner.html, 2021. Navštíveno 28. 4. 2022.

[28] Unity Software Inc. Unity scripting reference 2021.2. https://docs.unity3d.com/2021.2/Documentation/ScriptReference/, 2021. Navštíveno 30. 3. 2022.

[29] Unity Software Inc. Unity user manual 2021.2. https://docs.unity3d.com/2021.2/Documentation/Manual/, 2021. Navštíveno 30. 3. 2022.

[30] Unity Software Inc. Unity – manual: C# compiler. https://docs.unity3d.com/2021.2/Documentation/Manual/CSharpCompiler.html, 2021. Navštíveno 30. 4. 2022.

[31] Unity Software Inc. Unity – manual: Order of execution for event functions. https://docs.unity3d.com/2021.2/Documentation/Manual/ExecutionOrder.html, 2021. Navštíveno 30. 4. 2022.

[32] Unity Software Inc. Unity – manual: Script serialization. https://docs.unity3d.com/2021.2/Documentation/Manual/script-Serialization.html, 2021. Navštíveno 23. 4. 2022.

[33] Unity Software Inc. Unity – scripting API: SerializeReference. https://docs.unity3d.com/2021.2/Documentation/ScriptReference/SerializeReference.html, 2021. Navštíveno 30. 4. 2022.

[34] Unity Software Inc. Unity – manual: Extending the editor. https://docs.unity3d.com/2021.2/Documentation/Manual/ExtendingTheEditor.html, 2021. Navštíveno 1. 5. 2022.

[35] Unity Software Inc. Download – Unity. https://unity3d.com/get-unity/download. Navštíveno 30. 3. 2022.

[36] Unity Software Inc. Unity – manual: TextMeshPro. https://docs.unity3d.com/2021.2/Documentation/Manual/com.unity.textmeshpro.html, 2021. Navštíveno 30. 4. 2022.

[37] Microsoft. Quickstart: Configure visual studio for cross-platform development with unity. https://docs.microsoft.com/en-us/visualstudio/gamedev/unity/get-started/getting-started-with-visual-studio-tools-for-unity, 2022. Navštíveno 30. 3. 2022.

[38] Unity Software Inc. Unity – manual: Special folder names. https://docs.unity3d.com/2021.2/Documentation/Manual/SpecialFolders.html, 2021. Navštíveno 30. 4. 2022.

[39] Unity Software Inc. Unity – manual: Materials. https://docs.unity3d.com/2021.2/Documentation/Manual/Materials.html, 2021. Navštíveno 30. 4. 2022.

[40] Unity Software Inc. Unity – manual: Physic material component reference. https://docs.unity3d.com/2021.2/Documentation/Manual/class-PhysicMaterial.html, 2021. Navštíveno 30. 4. 2022.

[41] Unity Software Inc. Unity – manual: Special folders and script compilation order. https://docs.unity3d.com/2021.2/Documentation/Manual/ScriptCompileOrderFolders.html, 2021. Navštíveno 30. 4. 2022.

[42] Unity Software Inc. Unity manual – Unity Remote. https://docs.unity3d.com/2021.2/Documentation/Manual/UnityRemote5.html. Navštíveno 30. 3. 2022.

[43] Unity Software Inc. EventSystem | Unity UI | 1.0.0. https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/EventSystem.html. Navštíveno 4. 5. 2022.

[44] the free encyclopedia Wikipedia. Curiously recurring template pattern. https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern, 2021. Navštíveno 7. 5. 2022.

[45] Department of Computer Science at the Michigan Technological University Dr. Ching-Kuang Shene. Finding a point on a Bézier curve: De Casteljau's algorithm. https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/Bezier/de-casteljau.html. Navštíveno 27. 4. 2022.

[46] Unity Software Inc. Unity – manual: Mipmaps. https://docs.unity3d.com/2021.2/Documentation/Manual/texture-mipmaps.html, 2021. Navštíveno 2. 5. 2022.

[47] Arm Limited. Texture filtering. https://developer.arm.com/documentation/102449/0100/Texture-filtering, 2021. Navštíveno 2. 5. 2022.

[48] the free encyclopedia Wikipedia. Texture filtering. https://en.wikipedia.org/wiki/Texture_filtering, 2021. Navštíveno 2. 5. 2022.

[49] Unity Software Inc. Unity – manual: Texture import. https://docs.unity3d.com/2021.2/Documentation/Manual/class-TextureImporter.html, 2021. Navštíveno 2. 5. 2022.

[50] Unity Software Inc. Unity – scripting API: Texture.mipMapBias. https://docs.unity3d.com/2021.2/Documentation/ScriptReference/Texture-mipMapBias.html, 2021. Navštíveno 2. 5. 2022.

[51] Unity Software Inc. Unity – manual: Physics. https://docs.unity3d.com/2021.2/Documentation/Manual/PhysicsSection.html, 2021. Navštíveno 26. 4. 2022.

[52] Unity Software Inc. Unity – manual: WheelCollider. https://docs.unity3d.com/2021.2/Documentation/Manual/class-WheelCollider.html, 2021. Navštíveno 26. 4. 2022.

[53] Doc. Ing. Jiří Danzer CSc. Elektrická trakce 7. – adheze. *Studijní materiál Katedry výkonových elektrotechnických systémů Elektrotechnické fakulty Žilinské univerzity v Žilině*, pages 8–9, 2008. Navštíveno 26. 4. 2022 z archive.org.

[54] Unity Software Inc. a uživatelé Unity Forum. Good values for WheelColliders – Unity forum. https://forum.unity.com/threads/good-values-for-wheelcolliders.441506/, 2018. Navštíveno 26. 4. 2022.

[55] Unity Software Inc. Unity – manual: Unity UI. https://docs.unity3d.com/2021.2/Documentation/Manual/com.unity.ugui.html, 2021. Navštíveno 4. 5. 2022.

[56] Microsoft. The model-view-viewmodel pattern. https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm, 2021. Navštíveno 4. 5. 2022.

[57] Google Developers. Android debug bridge, install an app. https://developer.android.com/studio/command-line/adb, 2022. Navštíveno 5. 5. 2022.

[58] Unity Software Inc. Explore the unity editor. https://learn.unity.com/tutorial/explore-the-unity-editor-1. Navštíveno 8. 5. 2022.

Příloha A

Přehled elektronických příloh

- priloha.zip
  - Build
    - traffic.apk - sestavená aplikace pro systém Android
  - Traffic - Unity projekt vyvinuté hry
    - Assets
      - Editor - zdrojové soubory rozšíření Unity Editoru
      - Resources - zejména audiovizuální prvky hry
      - Scenes
        - EntryMenu.unity - Unity scéna úvodní obrazovky
        - DefaultLevel.unity - Unity scéna samotné hry
        - EmptyLevel.unity - Unity scéna prázdného světa
      - Scripts - zdrojové soubory hry
      - TextMesh Pro - soubory balíčku pro vykreslování textu
    - Packages - informace o použitých závislostech
    - ProjectSettings - konfigurační soubory Unity
    - Assembly-CSharp.csproj - konfigurace projektu hry
    - Assembly-CSharp-Editor.csproj - konfigurace projektu editoru
    - Traffic.sln - konfigurace celého C# solution
  - README.txt - informace o struktuře přílohy

Příloha B

Mapa výchozího levelu `DefaultLevel`