VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ROZŠÍŘENÁ REALITA NAD OBRAZEM ZE STACIONÁRNÍ KAMERY

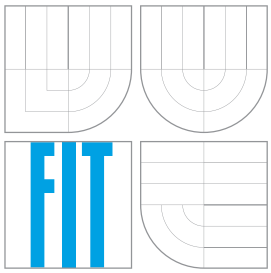
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN TINKA

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ROZŠÍŘENÁ REALITA NAD OBRAZEM ZE STACIONÁRNÍ KAMERY

AUGMENTED REALITY USING VIDEO FROM A STATIONARY CAMERA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN TINKA
AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. ADAM HEROUT, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

Tato práce se věnuje konceptu rozšířené reality nad obrazem ze stacionární kamery a jejím cílem je nabídnout prototyp interaktivního editoru, díky kterému je možné ve scéně vytvářet virtuální objekty, jako kompenzaci nedostatečných možností detekce nepohybujících se objektů v obraze ze stacionárních kamer. Výsledkem práce je editor scény vyvinutý v herním enginu Unity, jenž nabízí možnosti pro tvorbu jednoduchých objektů a jehož výstup je využitelný v projektech Unity.

Abstract

The goal of this thesis is to provide an interactive scene editor prototype as a way to compensate for the limited static object recognition capabilities of the fixed-camera-based approaches to augmented reality. The final result is an editor developed in the Unity game engine, which can be used to create simple objects and output of which is Unity project compatible.

Klíčová slova

rozšířená realita, stacionární kamera, zpracování obrazu, Unity 3D, unity, editor, INCAST

Keywords

augmented reality, stationary camera, fixed camera, image processing, Unity 3D, unity, editor, INCAST

Citace

Jan Tinka: Rozšířená realita nad obrazem
ze stacionární kamery, bakalářská práce, Brno, FIT VUT v Brně, 2015

Rozšířená realita nad obrazem ze stacionární kamery

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Ing. Adama Herouta, Ph.D. Uvedl jsem všechny prameny a publikace, ze kterých jsem čerpal.

.....
Jan Tinka
20. května 2015

Poděkování

Tímto bych chtěl poděkovat svému vedoucímu panu doc. Ing. Adamu Heroutovi, Ph.D. za jeho rady a podporu.

Obsah

Úvod

Tato práce se věnuje návrhu a implementaci prototypu editoru scény *proof of concept* infrastruktury pro tvorbu a distribuci tzv. interaktivních streamů, konceptu založeném na alternativním přístupu k rozšířené realitě využívajícího stacionárních kamer a jejich vlastností. Editor scény by měl sloužit vytváření objektů ve virtuální scéně tak, aby tato co nejblíže odpovídala scéně skutečné.

V první kapitole shrnu současný stav na poli rozšířené reality. Uvedu přístupy k ní z hlediska počítačového vidění a tyto budu dělit podle dvou kritérii. Popíši jejich principy a uvedu příklady existujících řešení. Rovněž uvedu motivaci k nasazení konceptu se stacionární kamerou, jeho silné a slabé stránky.

V druhé kapitole blíže představím koncept interaktivního streamu, na kterém je tato práce postavena, a detailněji popíši jeho části.

Ve třetí kapitole se budu věnovat hernímu enginu Unity, který jsem použil pro implementaci editoru. Popíši základní principy jeho fungování a práce s ním, důležité pro pochopení návrhu a implementace, zejména pak způsob, jakým se v něm programuje.

V poslední kapitole se pak budu věnovat návrhu a implementaci samotného editoru. V úvodu kapitoly popíši strukturu projektu v rámci Unity, po čemž následuje část o principu fungování hlavní nabídky. Ve zbytku kapitoly postupně popíši funkce nabízené editorem spolu s jejich implementací. Začnu od kalibrace a budu pokračovat od jednoduchých objektů ke složitějším, kdy se zároveň budu odkazovat k již popsané funkcionalitě. Na závěr kapitoly vysvětlím export a import výsledku práce v editoru.

V závěru textu zhodnotím výsledek práce a navrhnu rozšíření dosavadní verze editoru o novou funkcionalitu.

Kapitola 1

Přístupy k rozšířené realitě

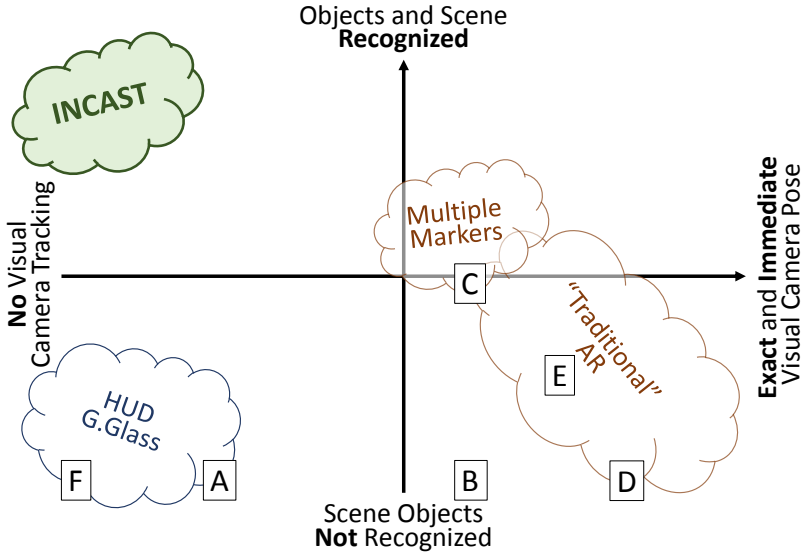
V oboru rozšířené reality se běžně uplatňují nejen techniky z oblasti počítačového vidění, ale také techniky umožňující interakci jednoho nebo více uživatelů současně. Příkladem víceuživatelského vstupu může být ovládání virtuálního prostředí rozšířené reality pomocí skutečných objektů, tedy využívající koncept tzv. *tangible user interfaces* [12, 10]. Rozšířená realita rovněž staví na vykreslovacích technikách, jejichž účelem je maximálně zastřít přechod mezi virtuální nástavbou scény a obrazem reality, ať už se jedná o realistické vykreslování, které bere v potaz existenci virtuálních objektů ve scéně a náležitě upravuje nasvětlování scény i objektů [7, 19], nebo o vykreslování nerealistické (non-realistic rendering), stylizující scénu a virtuální nástavbu stejným, nerealistickým způsobem [2]. Technik uplatňovaných v rozšířené realitě je samozřejmě více, tato práce se nicméně zaměřuje na oblast počítačového vidění a snaží se ukázat alternativní přístup k některým jeho aspektům. Z pohledu porozumění scéně technikami počítačového vidění se dají aplikace rozšířené reality řadit pomocí dvou kritérií:

1. Rozpoznávání, odhad nebo znalost okamžitého vztahu kamery ke scéně – její pozice, rotace ...
2. Rozpoznávání a porozumění obsahu scény – detekce objektů a jejich vlastností, světelné podmínky, stíny a další ...

Toto je grafický znázorněno na obrázku 1.1, kde horizontální osa značí první kritérium – od žádného vyhodnocování aktuální polohy a orientace (vlevo) do přesného vyhodnocení (vpravo); a vertikální osa značí druhé kritérium – od žádného rozpoznávání (dole) do „dokonalého" rozpoznání obsahu scény (nahoře).

1.1 Rozšířená realita s markery

Aplikace rozšířené reality vycházející z přístupů k rozpoznání vztahu kamery ke scéně založených na raných modelech speciálních značek, zvaných marker, nenabízí žádné porozumění obsahu scény. Jejich slabostí jsou i omezené možnosti odhadu pozice kamery, jehož úspěch závisí na úhlu pohledu, dobré viditelnosti a omezené vzdálenosti od markeru. Stejně tak nesmí být marker ničím zakrytý, neboť na něm odhad pozice zcela závisí. Příkladem může být konferenční systém využívající náhlavních displejů (HMD) [11]. Tato skupina je v obrázku 1.1 označena jako A.



Obrázek 1.1: Různé aplikace rozšířené reality odhadují pozici a orientaci kamery s různou přesností (horizontální osa) a rovněž jsou na různé úrovni schopné rozpoznávat obsah scény (vertikální osa). A – raný design markerů; B – vylepšené markery; C – více markerů kvůli interakci; D – mapování scény a pozice kamery bez markerů; E – počítačové vidění zaměřené na porozumění scény; F – HUD a brýle bez počítačového vidění. **zelený oblak:** Koncept INCASTu: Kamera se nepohybuje, tudíž zde není žádné rozpoznávání polohy z pohybu kamery, ale zároveň je možné velmi dobře v reálném čase analyzovat scénu.

1.2 Pokročilé markery

Aplikace vycházející z použití vylepšených markerů [10, 8, 6] mají několik zásadních výhod. Markery jsou mnohem více přizpůsobitelné a již se nemusí jednat o uměle vypadající skupinu přímek nebo čtverců, namísto toho lze použít téměř jakýkoliv design mající jisté vlastnosti, jako jsou významné rysy a hrany [22], nebo se může jednat o objekty [18]. Další výhodou je vyšší spolehlivost. Tento přístup je odolnější proti okluzi, rozmazání pohybem a je mnohem méně citlivý na vzdálenost a úhel, s jakými se kamera na marker dívá. Vylepšené markery navíc mohou pojmout více dat, aplikace si je méně pletou s jinými markery [6], mohou být pro člověka téměř neviditelné [23]. Aplikace využívající toho přístupu (B v obrázku 1.1) jsou schopny relativně přesně odhadnout pozici kamery ve scéně, vyžadují však stále ještě existenci markerů.

Markery jsou také využívány v aplikacích označených C, ve kterých neslouží jen k získání informací o kameře, ale zároveň i pro odhadování pozice objektů nebo částí těla[17, 5]. Dále také třeba coby efektivnější náhrada za zelené plátno (color separation overlay) [4]. Markery se mohou objevovat i v ostatních konceptech, kde plní doprovodnou funkci efektivního prostředku pro interakci se scénou.

1.3 Bez markerů a rozpoznávání objektů

Další skupinou jsou aplikace, které vyhodnocují pozici kamery bez použití markerů. Kamera se pohybuje v neznámém prostředí,které aplikace musí mapovat – vytvářet si jeho virtuální podobu, a zároveň znát svoji polohu v tomto prostředí. Tomuto procesu se říká *simultaneous*

localization and mapping (SLAM). Mapování prostředí se dá docílit např. pomocí hledání výrazných bodů či hran v obraze [13] nebo tzv. přímými metodami, kdy dochází ke zpracovávání všech pixelů při tvorbě hloubkových map a vztahů mezi snímky [16, 20]. Tento přístup ale nerozeznává objekty. Skupina pod štítkem D.

1.4 Bez markerů s rozpoznáváním objektů a porozuměním scéně

Dále je tu skupina aplikací E, které rovněž nevyužívají markery, ale na místo určování pozice kamery ve scéně jsou spíše zaměřeny na rozpoznávání objektů, světelných podmínek a jiných vlastností scény. Jelikož se jedná o poněkud různorodou a rozsáhlejší skupinu, uvedu několik příkladů.

Real-Time Illumination Estimation from Faces for Coherent Rendering [14]. Jedná se o metodu pro odhad světelných podmínek skutečného světa ze vzorku předem nafocených obličejů v různých světelných podmínkách, za účelem realistického nasvěcování virtuálních prvků scény. Při nasazení metody je k snímanému obličeji nalezena nejlepší shoda v datovém vzorku a virtuální objekty jsou nasvíceny podle předem vyhodnocených světelných podmínek, čímž se ušetří výpočetní výkon.

Světelným podmínkám se věnují i J. Jachnik a kolektiv [9] se svojí technikou pro zaznamenání světelných podmínek plochého povrchu bez využití speciálního vybavení. Nasbírané informace jsou použity pro vytvoření světelného pole (light field), pomocí něhož je získána odrazová mapa okolí (environment map), díky které je pak možné na lesklém povrchu odrážet virtuální objekty.

Dalším příkladem je třeba technika zaznamenávání deformací skutečných objektů. Nejdříve je model objektu automatický zpracován. Získaná data slouží jako vstup pro fyzikální simulaci, z jejichž výsledků jsou vyvozena omezení pro chování virtuálního modelu při deformaci [15].

Do této kategorie rovněž zapadá systém pro detekci a sledování 3D objektů bez textur pro mobilní telefony s fotoaparátem, který umožňuje manuální modelování objektů a nabízí automatickou detekci usnadňující inicializační fázi sledování a navázání sledování, bylo-li přerušeno[1].

Získávání znalostí o vztahu kamery ke scéně je u této skupiny aplikací a technik do jisté míry nedokonalé, neboť často závisí na pohybu objektů či změně světelných podmínek.

1.5 Bez rozpoznávání objektů a porozumění scéně

Velice odlišnou skupinou od již zmíněných je koncept mnoha průhledových (HUD) a náhlavních displejů (HMD) či brýlí, jejichž účelem je přidávat do obrazu prvky, které nereprezentují objekty reálného světa, jako jsou hodiny, údaje o rychlosti, vzdálenost k cíli, aj. Tyto systémy nezpracovávají obraz pro odhad pozice a rotace kamery. Pokud ano, vystačí si pouze s omezenou přesností. Např. navigace na delší vzdálenosti apod. Tato skupina je dobrým příkladem praktického nasazení rozšířené reality nepostavené na odhadování pozice kamery z jejího pohybu. V obrázku 1.1 je označena F.

1.6 Nehybná kamera, maximální přesnost

Tato práce se snaží prezentovat alternativní koncept rozšířené reality, ve kterém, stejně jako v poslední zmíněné skupině, nedochází k získávání informací o pohybu kamery ve scéně, neboť tato kamera je pevně umístěná. Ačkoli omezení pohybu kamery s sebou nese jisté nevýhody, jako je snížená interaktivita, dá se tohoto omezení i do značné míry využít. Jeden z hlavních pozitivních aspektů je zvýšená přesnost detekce pohybujících se objektů stejně tak jako zjišťování jejich pozice, rozměrů a jiných vlastností. Dalším je lepší celkové porozumění světelným podmínkám ve scéně, čehož je krom realistického nasvěcování virtuálních objektů možné využít i k predikci stínů. U systémů s pohybující se kamerou je zjišťování úhlu dopadajícího světla nejen náročnější (musí se vyhodnocovat v reálném čase), ale i méně přesné.

Tento koncept se snaží využít dosud nejlepšího porozumění scéně a přesné detekce objektů a uživateli aplikace založené na tomto konceptu nabídnout velice kvalitní rozšířenou realitu, ve které dochází k realistické interakci mezi skutečnými a virtuálními objekty. Jedním z úskalí při zpracovávání obrazu ze stacionární kamery je problematika detekce nehybných objektů. Této problematice se tato práce věnuje, neboť její náplň tvoří právě editor pro anotaci objektů ve scéně.

Kapitola 2

INCAST

Náplní této práce je návrh a implementace části *proof of concept* řetězce tvorby a dodávání obsahu pro rozšířenou realitu. S původní myšlenkou tohoto řetězce mě seznámil můj vedoucí pan docent Herout. Podstatou bylo využití vlastností stacionárních kamer a dat, které z nich lze získat, ke tvorbě interaktivního i neinteraktivního obsahu v podobě užitečné, či zábavné aplikace a vyzdvihnout možnosti stacionárních kamer v oboru rozšířené reality. Aplikace, jenž by byla výsledkem tohoto procesu, by měla být schopna v reálném čase zpracovávat video stream z kamery, pro kterou byla vytvořena, a náležitě přizpůsobovat obsah scény vnějším podnětům jako jsou pohyby skutečných objektů ve video streamu a uživatelský vstup, je-li aplikace interaktivní.

Později jsme z tohoto konceptu se skupinou lidí z UPGM FIT – I. Szentandrási, M. Zachariáš, R. Kajan, M. Dubská, J. Sochor a A. Herout; vycházeli, vylepšili jej, dali mu konkrétnější podobu a pojmenovali jej Interactive Camera Stream (INCAST). INCASTu byl v rámci International Symposium on Mixed and Augmented Reality (ISMAR) 2015 věnován článek [21][1], jehož jsem spoluautorem.

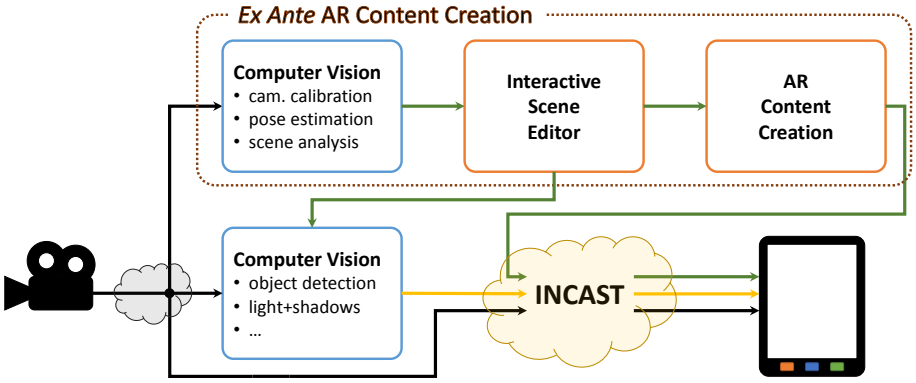
Tento koncept, je znázorněn na obrázku 2.1. Zdrojem video streamu bude nejčastěji kamera pro monitorování dopravního provozu, kamera pro vizuální analýzu, časosběrná kamera atp. Video stream je na obrázku znázorněn černými šipkami.

2.1 Tvorba aplikací

INCAST je rozdělen na 2 fáze. Tou první je tvorba cílové aplikace rozšířené reality označená jako *ex ante AR content creation*, v obrázku 2.1 znázorněná hnědým tečkovaným rámečkem. První blok této části označený modrým rámečkem je zcela automatický a spočívá ve zpracování obrazu za účelem získání údajů pro automatickou kalibraci kamery. Příkladem úspěšné automatické kalibrace může je jeden případ užití v [21], ve kterém je použito video z kamery monitorující provoz. Kalibrace je založena na získání tří ortogonálních úběžníků z nashromážděných dat o pohybu vozidel pomocí varianty Houghovi transformace zvané *diamond space*[3].

Dalším blokem je interaktivní editor scény, kterým jsem se v rámci této práce zabýval já. Jeho účelem je získání dodatečných informací o scéně jako jsou pozice různých objektů a překážek a jejich přesný tvar, fyzikální vlastnosti povrchů a další. Tyto informace je velmi těžké ze statického obrazu získat automaticky. Člověk znalý scény by v editoru

[1] Článek nebyl v době citace dosud zveřejněn.



Obrázek 2.1: INCAST pipeline. Před nasazením INCASTu je nejprve nutné vytvořit cílovou aplikaci (AR Content) a připravit serverovou část na práci s příslušným video streamem. Oranžové články řetězce jsou interaktivní, modré jsou plně automatické. Když je cílová aplikace používající INCAST spuštěna, automatické procesy počítačového vidění zpracovávají video stream z kamery a potom ho spolu s výstupními daty odesílají. *černé šipky:* video stream, *žluté šipky:* rozpoznané objekty a další data, *zelené šipky:*neměnný obsah scény z editoru a další neměnná data. Žlutý oblak reprezentuje síť pro distribuci INCAST obsahu (CDN), černý oblak pak infrastrukturu streamu kameru. [21]

anotoval různé objekty či plochy, výstup editoru by potom nabízel poměrně detailní informace o scéně. Výhodou takového výstupu je, že není omezený na jedno použití, místo toho může na jeho základě být postaveno mnoho různých aplikací pracujících s toutéž scénou. Motivací k vytvoření samostatného editoru scény je, kromě jiného(viz následující odstavec), možnost nabídnout jeho uživateli relativně jednoduchý způsob k anotaci scény namísto toho, aby pracoval např. s Unity, u kterého by musel tento potenciálně neznalý člověk investovat neúměrné množství času a úsilí do toho, aby se s ním naučil pracovat.

Třetí blok tvorby aplikace (v obrázku *AR Content Creation*) je zpracování výstupu editoru scény a přidání potřebné funkcionality. V této části již dochází ke tvorbě samotné aplikace. Jedním z důvodů, kvůli kterému jsou interaktivní editor scény a tato část rozděleny, je ,aby případní tvůrci aplikací byli oddělení od vlastníka kamery, nebo poskytovatele streamu. Tento přístup umožňuje, aby se vlastník kamery soustředil pouze na její správu, dohlížel na proces zpracování obrazu a poskytoval kvalitní model scény. Tvůrci aplikaci se potom mohou soustředit pouze na tvorbu aplikace. Tato tvorba by díky tomu, že se nemusejí starat o scénu, neměla být o nic náročnější než tvorba jakékoliv jiné aplikace a jejího obsahu.

2.2 Zpracovávání obrazu za běhu a vysílání INCASTu

Jakmile je kamera zkalibrovaná, scéna dobře vymodelovaná a cílová aplikace hotová i nasazená do provozu, může se začít vysílat INCAST. Před tím, než se stream z kamery dostane ke koncovému uživateli, musí být automaticky zpracován na serveru (spodní modrý obdélník *Computer Vision* v obrázku 2.1), kde jsou detekovány pohybující se objekty, vyhodnoceny světelné podmínky atd. Díky velmi dobré znalosti scény získané v první a druhé části fáze tvorby, je server schopen velmi přesného rozpoznávání objektů, porozumění scéně, ... Tato část může být výpočetně velmi náročná a krom toho, že by mohla pro koncová zařízení být až příliš náročná, musela by probíhat na všech zařízeních pracujících s daným INCASTem. Z tohoto důvodu je tato část umístěna na serveru, kde je kromě větších prostředků požada-

vek zpracovat každý INCAST pouze jednou popř., je-li na streamu z kamery postaveno více aplikací, pro každý právě používaný INCAST jednou.

Potom se již původní video stream spolu se zpracovanými daty (v obrázku 2.1 znázorněnými žlutou šipkou) a daty tvůrce aplikace spojí v INCAST. Tento je potom distribuován potenciálně velkému množství uživatelů prostřednictvím sítě pro doručování obsahu (CDN).

Kapitola 3

Unity

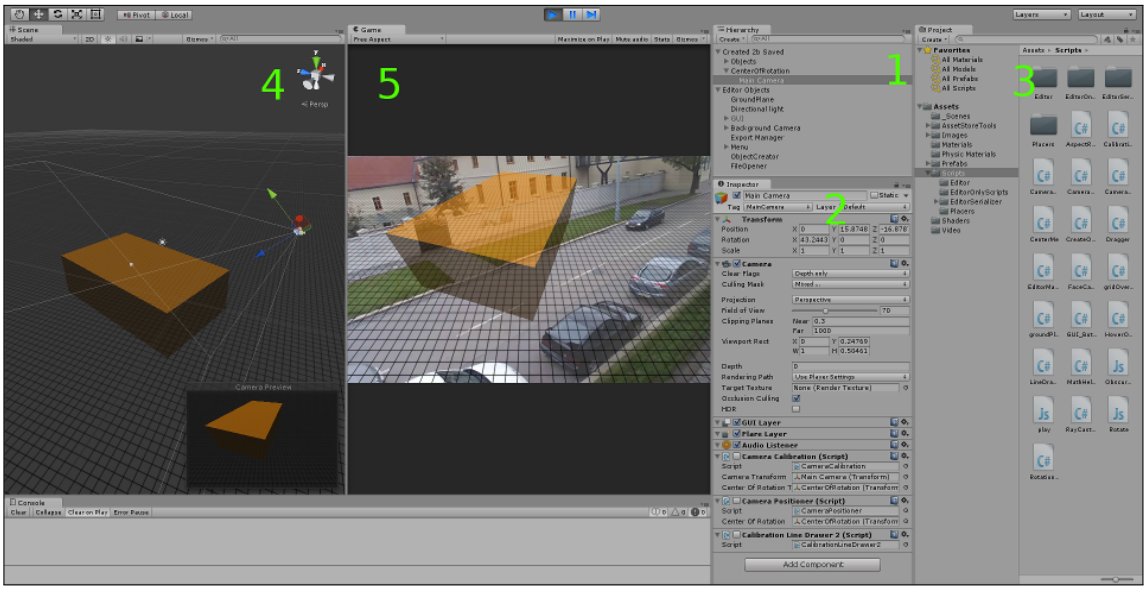
Velice podstatnou součástí této práce je Unity, neboť právě v něm je práce implementována a stejně tak je na něm závislá použitelnost jejího výstupu. V této kapitole bude představeno Unity, zmíním, proč jsem jej zvolil pro implementaci, a uvedu základní informace o jeho fungování a práci s ním, důležitě pro pochopení implementace této práce.

Při rozhodování o implementačním frameworku/enginu/prostředí jsem jako vhodné kandidáty vybíral pokud možno vysokoúrovňová řešení, s již funkčním grafickým enginem. V úvahu připadali Unity[1], Unreal Engine[2], Cryengine[3], OpenSceneGraph[4] a Ogre 3D[5]. OpenSceneGraph a Ogre3d jsou open-source projekty, u nichž je díky otevřenosti kódu možná maximální variabilita a přizpůsobivost. Problémem u těchto řešení je bohužel relativně vysoká nízkoúrovňovost (Ogre3D je samostatný grafický engine), kvůli které by bylo nutné spoustu věcí implementovat od základů. Unity, Unreal Engine a Cryengine jsou herní enginy, z nichž všechny nabízejí nějakou formu editoru scény. Tyto enginy se liší v mnoha aspektech, jako jsou licenční podmínky, dostupnost a kvalita dokumentace či tutoriálů, nabízené funkce, aj. Rozhodujícím faktorem se ale stala rychlost, s jakou lze v daných enginech dosáhnout použitelných výsledků, Unity, jenž nabízelo značnou část své funkcionality zdarma, poskytovalo mnoho vestavěných prostředků pro snadnější a efektivnější tvorbu aplikací, než tomu bylo u ostatních enginů (od prvního čtvrtletí 2015 jsou však Unity i Unreal Engine poskytovány za určitých podmínek zcela zdarma). Vzhledem k principu, jakým je tvorba aplikací v Unity koncipována, a jeho closed-source povaze jsou do jisté míry omezeny možnosti přizpůsobení výsledné aplikace (grafický nebo fyzikální engine jsou neměnné), což však v případě této práce nepředstavuje problém.

Unity je multiplatformní herní engine od společnosti Unity Technologies dodávaný s vlastním vývojovým prostředním. Hlavním účelem Unity je v první řadě vývoj her pro širokou řadu podporovaných desktopových, mobilních i konzolových platforem jako jsou iOS, Android, Windows, Linux, Mac a další. Stejně tak lze Unity použít pro tvorbu webových aplikací stavějících na WebGL nebo Unity Web Player, což je plugin do internetových prohlížečů vyvinutý Unity Technologies pro tento účel. Vývoj her a aplikací je však možný pouze na Mac OS X a Windows, ostatní platformy zatím podporovány nejsou.

Vývojové prostředí se sestává z Unity Editoru a upravené verze open source IDE MonoDevelop, kterým se zde zabývat nebudu. Unity Editor slouží k efektivní správě a tvorbě

[1] http://unity3d.com/
[2] http://www.unrealengine.com
[3] http://cryengine.com
[4] http://www.openscenegraph.org/
[5] http://www.ogre3d.org/



Obrázek 3.1: **Unity Editor:** 1 – hierarchie objektů ve scéně, 2 – komponenty vybraného objektu, 3 – *assets*, 4 – 3D scéna, 5 – náhled hry(z pohledu kamery)

jednoho projektu, což může, stejně jako u různých IDE (MS Visual Studio, MonoDevelop, NetBeans, ...), znamenat jednu aplikaci, ale možné je i vytvořit více aplikací z jednoho projektu za pomoci samostatného překladu podmnožiny scén. Scéna je reprezentací 3D scény spolu se všemi v ní se nacházejícími objekty a funkcionalitou. Z pohledu hry může scéna být mapou, misí nebo také hlavním nabídkou. Uživatel upravuje každou scénu zvlášť, může mezi nimi však libovolně přepínat.

3.1 Základný stavební prvky scény - *GameObject* a *Component*

Obsah scény tvoří objekty *GameObject*, které reprezentují libovolný (i nehmotný) objekt, a jejich komponenty *Component*. Pokud má objekt mít model, vydávat zvuky, chovat se jako pevné těleso řízené fyzikálním enginem, být animován nebo mít jakoukoli jinou funkci, přidá se mu příslušná komponenta. *GameObject* v podstatě plní pouze úlohu "schránky"na komponenty, které představují jádro funkcionality. Všechny *GameObjecty* a komponenty mohou být v aktivním stavu, nebo neaktivním. Neaktivní komponenta přestane plnit svojí funkci, neaktivní *GameObject* znamená neaktivitu všech jeho komponent a synovských objektů.

3.1.1 *Transform* – pozice, rotace...

Další základní vlastností, kterou sdílí všechny *GameObjecty* je reprezentace jejich umístění ve scéně. K tomuto účelu slouží komponenta *Transform*. Ta uchovává informace o pozici, rotaci a měřítku. Tyto jsou k dispozici ve dvojí podobě:

- globální – z pohledu souřadnicového systému scény
- lokální – z pohledu rodičovského objeku

Nepostradatelnou funkcí *Transform* je také spravování hierarchických vazeb mezi objekty. Díky hierarchické vazbě jsou synovské objekty ovlivňovány pozicí, rotací i měřítkem svého otcovského objektu. Díky tomu je při práci s editorem možné pohodlně posunout, natočit nebo zvětšit celou hierarchickou strukturu, stejně tak jako se dá těchto vlastností využít při programování.

3.1.2 *Mesh* a *Collider* – model a kolize

Modely jsou v unity reprezentovány pomocí meší. Ty sice sami o sobě nejsou komponentami, váže se k ním však skupina komponent, které je využívají. *Mesh Renderer* zajišťuje vykreslování modelu, *Mesh Filter* určuje jaká meš je použitá *Mesh Renderem* a ,*Mesh Collider* je druh komponenty *Collider*, která má na starosti kolize, a využívá meš jako kolizní model.

Komponenta *Collider* může kromě různorodých meší používat i efektivnější meše výchozích objektových primitiv, která zmíním později.

3.1.3 *Camera*

Camera je komponenta, která dělá z objektu kameru. Má vlastnosti jako zorné pole (FOV), perspektiva, vzdálenost *clipping planes* a vztah obrazu z kamery k velikosti okna, ve kterém se zobrazuje. Rovněž kamera umožňuje určit vykreslovací masku, zvolit, co se má zobrazit v místech, kde není žádný objekt a nastavit jakou bude mít obraz z kamery prioritu, je-li jich ve scéně více. Právě jedna kamera ve scéně je hlavní (kamera se štítkem *MainCamera*).

3.1.4 Další komponenty

Existuje opravdu velké množství komponent, ale ty které jsem zmínil jsou podstatné z hlediska implementace této práce. To neznamená, že by nebyly použity i jiné komponenty, např. komponenta světelného zdroje *Light* je velmi podstatná, není na ní však založený žádný z použitých algoritmů a postupů.

3.1.5 Předdefinované objekty

Aby usnadnilo práci vývojářům, nabízí Unity řadu předdefinovaných *GameObjectů* a objektových primitiv, které mají již všechny (nebo většinu) potřebné komponenty. Velice důležitá jsou jednoduchá tělesa jako krychle *Cube*, koule *Sphere*, ovál *Capsule* nebo plocha *Plane, Quad*. Všechna tato tělesa mají optimalizované meše a kolizní modely.

3.1.6 Štítky a vrstvy

Krom hierarchických struktur je možné v Unity shlukovat objekty i pomocí štítků (*tag*). Takové objekty je potom možné při programování získat pomocí speciální metody odkazující se na jméno štítku. Koncept štítků je bohužel omezen v tom, že štítky existují pouze v rámci projektu. Vrstvy se využívají, je-li z nějakého důvodu vhodné rozdělit objekty při vykreslování, *ray castingu* apod.

3.2 Programování

Má-li výsledná aplikace dělat něco jiného, než jen zobrazit padající předmět, je nutné tuto funkcionalitu naprogramovat. V Unity (verze 5.0.1f) se programuje v jazycích C# nebo JavaScript, tyto se dají v projektu i kombinovat, ne však v rámci jednoho souboru. Informace,

které uvedu v této sekci, nezávisí na programovacím jazyku, práci jsem však implementoval v C#. Programování v Unity by se dalo rozdělit na tři části. "Klasické"programování, kdy se využívají obecné principy OOP při tvorbě různých tříd, programování tříd dědících ze speciálních tříd jako jsou *MonoBehaviour* nebo *EditorWindow* a programování shaderů

3.2.1 *MonoBehaviour*

Přidává-li se nějakému *GameObjectu* uživatelem definovaná funkcionalita, dělá se to s použitím tříd dědících od *MonoBehaviour*, kterážto třída jako jediná, může být komponentou. Proces vypadá přibližně tak, že uživatel vytvoří soubor se požadovanou třídou a tento soubor v Unity Editoru přetáhne na příslušný *GameObject*, nebo komponentu třídy tohoto souboru přidá *GameObjectu* programově.

Podstatou práce s *MonoBehaviour* je využití toho, že v reakci na některé události volají jisté metody. Metoda *Update()* je volána každý vykreslovaný snímek. Metoda *OnGUI()* je volána při každém uživatelském vstupu a při jiných, interních, událostech souvisejících s GUI a vykreslováním. Metoda *Start()* je zavolána před prvním voláním *Update()* a metoda *Awake()* pak ještě dříve, na rozdíl od *Start()* se u ní ale předpokládá, že může být zavolána před inicializací všech objektů ve scéně. *OnMouseOver()* je volána, nachází-li se kurzor nad objektem s touto komponentou, *OnMouseExit()*, když kurzor objekt opustí.

3.2.2 *EditorWindow*

Krom programování výsledné aplikace Unity nabízí i možnosti, jak přizpůsobit prostředí samotného editoru. K tomuto jsou určeny třídy začínající slovem "Editor."V rámci této práce jsem využil pouze třídy *EditorWindow*, pomocí které je možné vytvořit vlastní okno v editoru.

3.2.3 Shadery

Psaní shaderu je možné dvěma způsoby. Napsat celý shader ve vysokoúrovňovém jazyce *ShaderLab* nebo v kombinaci *ShaderLab* a *Cg / HLSL*. Vlastní shadery se vyplatí využívat v exotických případech (průhledný objekt, na který lze vrhat stín), ale ve většině případů jsou více než dostatečné shadery výchozí.

3.3 *Assets* a šablony objektů v podobě *prefabs*

Všechny soubory přístupné uživateli, které se nacházejí v projektu, jako scény, skripty, obrázky, textury, materiály, modely, videa nebo zvuky jsou tzv. *Assets*.

Materiály určují jaký má daný objekt povrch, na nich závisí, co se zobrazí při vykreslování. Každý materiál má definovanou texturu nebo barvu, a shader.

Jedním z druhů těchto *Assets* jsou také *prefabs*. Jedná se o permanentní verze *GameObjectů*, jakési šablony, díky kterým je možné za běhu hry programově vytvářet předdefinované objekty. Jsou vytvářeny buď přetažení *GameObjectu* z hierarchie objektů do hierarchie *assets* nebo programově, což vyžaduje, aby byla aplikace spuštěna v Unity Editoru, tzn. nedají se vytvářen za běhu koncové aplikace.

3.4 Vstup, výstup a ukládání

Komunikace výsledné aplikace s okolním světem může probíhat několika způsoby: ovládání pomocí klávesnici, myši apod. (speciálním případem je GUI), přes soubory a přes síť (nevyužil jsem). O uživatelský vstup se stará třída *Input*, ta spravuje mapování tlačítek a os vstupních zařízení na pojmenované obecné vstupy, např. mapování kláves A a D na hodnoty -1 a 1 osy pojmenované „Horizontal." Takováto mapování mohou být téměř libovolná a aplikace vytvořené s pomocí Unity v základu zobrazí okno pro nastavení vstupů (a dalších nastavení) při každém spuštění. Třída *Input* se velice úzce váže k *Input Manageru*, jenž konfiguruje v Unity Editoru.

Od verze 4.6 nabízí Unity relativně pohodlný systém pro tvorbu GUI zvaný *UI*, který je klasicky založený na událostech. Každý akční prvek, jako je tlačítko, může při akci volat libovolný počet metod, nebo nastavovat hodnoty primitivních typů, tyto vlastnosti se nastavují v Unity Editoru. Základním stavebním kamenem *UI* je *Canvas* (plátno), který reprezentuje plochu, ke které se vážou všechny ostatní *UI* elementy, a dal by se chápat jako vrstva. Mezi další prvky, které jsem využil, patří tlačítko, text a vstupní pole. Stejně tak jako všechny ostatní věci nacházející se ve scéně, jsou i prvky *UI GameObjecty* a podle toho s nimi lze i nakládat.

Unity je postavené na platformě .NET a umožňuje využít jeho tradiční rozhraní pro práci se soubory. Pro načítání a zároveň i zpracování některých druhů souborů slouží třída *WWW*, která umí, jak je z názvu patrné, načítat data z webu přes HTTP, ale je možné ji i využít k načítání lokálních souborů. Díky *WWW* je velmi snadné načíst obrázek, neboť tento je rovnou zpracován v texturu.

Ačkoli Unity nabízí velkou škálu možností, nenabízí žádný výchozí způsob pro ukládání stavu aplikace nebo exportu obsahu scény.

Kapitola 4

Editor scény

V této kapitole popíši editor scény, uvedu z jakých částí se skládá, jaké funkce nabízí a jak se používá. Editor, který jsem implementoval je prototypem interaktivního editoru scény konceptu INCAST(kapitola 2) a slouží k relativně snadné anotaci obsahu scény, tj. vytváření virtuálních protějšků skutečných objektů, vyznačování pozemních ploch se speciálními fyzikálními vlastnostmi ale také k manuální či poloautomatické kalibraci/zjištění polohy kamery.

Hlavní funkce editoru tak, jak se nacházejí v hlavním menu jsou tyto: načtení screenshotu scény, kalibrace/umístění kamery, vytváření objektů a export scény. Funkce budu popisovat v pořadí, va jakém by se s nimi setkal za normálních podmínek uživatel.

4.1 Struktura projektu

Dříve než se pustím do popisu jednotlivých částí editoru, je vhodné abych kvůli názornosti a zasazení popisů dílčích části do širšího kontextu zmínil celkovou logickou strukturu uvnitř projektu.

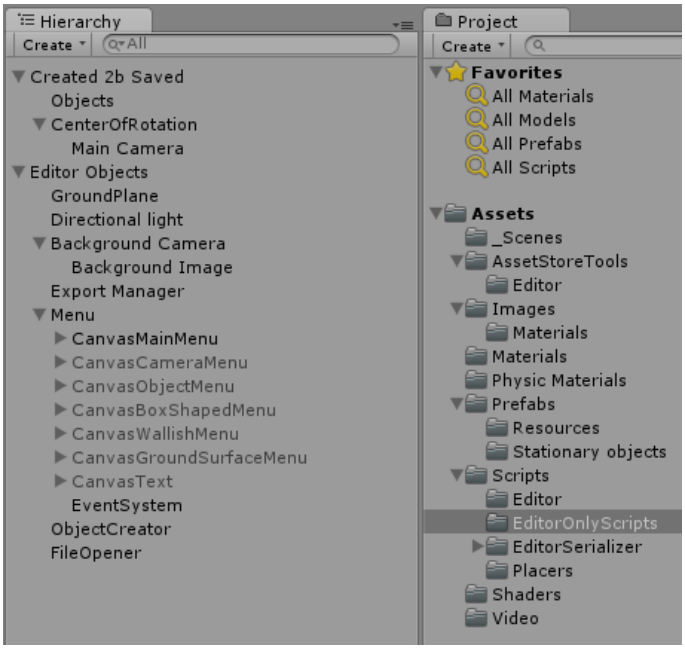
Hierarchie objektů (*GameObject*) je rozdělena na dvě skupiny, každá pod jedním kořenovým objektem. Všechny objekty, které jsou součástí editoru scény ale nebudou obsaženy v jeho výstupu, se nacházejí pod objektem *Editor Objects*, patří sem např. hlavní nabídka nebo zdroj světla. Objekty, které mají být součástí výstupu, jsou pod objektem *Created 2b Saved*, patří sem všechny objekty, které vytvoří uživatel, a kamera.

Editor scény je ve skutečnosti jedna scéna(viz kapitolu 3) a proto ne všechny z *assetů* v jejich hierarchii jsou relevantní. Podstatné jsou adresáře Materials, Prefabs a Scripts. V Materials se nacházejí materiály různých barev pro obarvení různých druhů objektů. Prefabs obsahuje *prefaby* všech objektů (*GameObject*) vytvářených za běhu aplikace. Strukturu projektu naleznete na obrázku 4.1.

4.2 Menu

Po zapnutí editoru uživatel uvidí výchozí pozadí, na něm mřížku a především hlavní nabídku (obrázek 4.2). Ta nabízí funkce vypsané výše. Uživatel může klikáním na tlačítka v menu vyvolávat akce nebo se pohybovat v hierarchii nabídek a podnabídek.

Uživatelské rozhraní jsem implementoval za pomocí Unity *UI*. Každá nabídka má svůj vlastní *canvas* a všechny nabídky jsou na stejné hierarchické úrovni pod objektem *Menu* (viditelné na obr. 4.1). Všechny funkce menu, které se vyvolají stiskem některého z tlačítek,



Obrázek 4.1: Struktura projektu v Unity Editoru. Hierarchie *GameObjectů* nalevo, hierarchie *assets* napravo. Zašedlé objekty jsou neaktivní.

Obrázek 4.2: Editor scény po spuštění. Hlavní nabídka je zvýrazněna červeně. *Load Scene Image* – načtení jiného screenshotu scény, *Camera Calibration* – podmenu pro kalibraci/umístění kamery, *Objects* – podmenu pro anotaci, *Export* – exportování scény, *Exit* – opustit editor

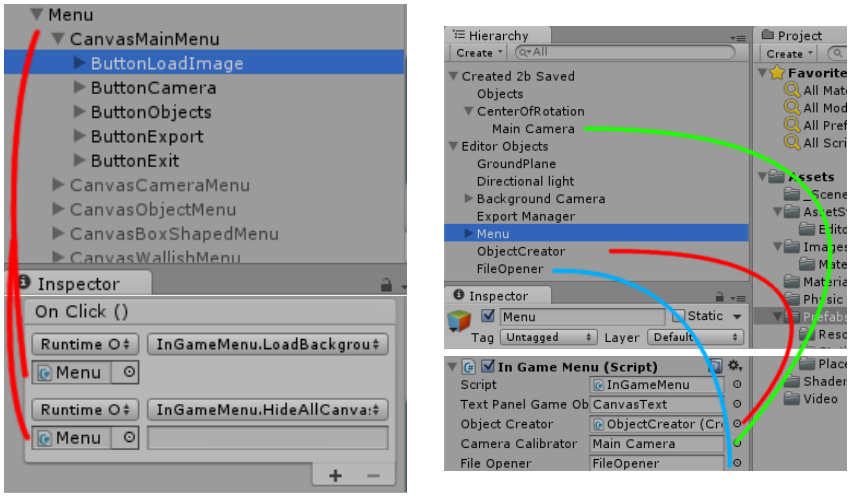
jsou implementovány ve skriptu *InGameMenu*, připojeném k objektu *Menu*, na který se všechna tlačítka odkazují(obr. 4.3).

Navigace v hlavní nabídce je řešena aktivací a deaktivací *canvasů* příslušných podnabídek. Např. na obrázku 4.3 vlevo je zvýrazněno tlačítko pro načtení nového pozadí. Toto tlačítko v reakci na stisknutí zavolá metodu *InGameMenu.HideAllCanvasesButOne*, která schová všechny *canvasy*, až na jeden, uvedený jako parametr, a předá ji prázdnou hodnotu (všimněte si prázdného políčka pod názvem metody v obr. 4.3), čímž efektivně zneaktivní a schová všechny podnabídky.

Má-li tlačítko spustit vykonávání nějaké akce, zavolá navíc i příslušnou metodu v *InGameMenu*. Ve většině případů tyto metody pouze spustí komponentu nějakého jiného objektu, který má danou činnost na starost. Až je činnost dokončena, komponenta, která ji prováděla, zavolá metodu pro znovuobjevení nabídky. Toto jsem řešil pomocí delegátů. Úryvek z *InGameMenu* s metodou *LoadBackgroundImage*:

```
public void LoadBackgroundImage(){
  fileOpener.GetComponent<BackgroundLoader>().OnFinished =
    ()=>this.HideAllCanvasesButOne("CanvasMainMenu");
  fileOpener.GetComponent<BackgroundLoader>().StartWorking();
}
```

kde *fileOpener* je *GameObject*, který se stará o dialogové okno a získání cesty k souboru, vizte vazbu mezi *InGameMenu* a *FileOpener* v obrázku 4.3 napravo. Připomínám, že všechny skripty jsou *MonoBehaviour*, tzn. že není potřeba jim předávat řízení, místo toho, se pouze aktivují a o vyvolání další činnosti se už postará např. metoda *Update*. Na stejném principu spuštění činnosti a schování všech nabídek fungují i ostatní funkční tlačítka.



Obrázek 4.3: Vazby v menu: vlevo – komponenta modře zvýrazněného tlačítka se odkazuje na *Menu*, v reakci na kliknutí volá dvě jeho metody, jedna z nichž je *HideAllCanvasesButOne*; vpravo – komponenta *InGameMenu* modře zvýrazněného menu, se odkazuje objekty mající na starost různou funkcionalitu

4.3 Obrázek v pozadí

První věc, kterou uživatel musí udělat, chce-li smysluplně používat editor scény, je nahrát do něj snímek scény z kamery, pro kterou bude scénu vytvářet. To udělá tak, že v hlavním menu

klikne na tlačítko *Load Scene Image*, které vyvolá dialog pro otevření souboru. Uživatel zvolí požadovaný obrázkový soubor a ten potom načte jako nové pozadí.

Existuje několik způsobů, jak zobrazit obrázek v pozadí a přes něj všechny virtuální objekty a GUI. Zvolil jsem možnost dvou kamer, jedna slouží pouze pro zobrazení pozadí, druhá pak vykresluje scénu (hlavní kamera). Kameru vykreslující pozadí jsem nastavil tak, aby se její obraz vykresloval až za obrazem hlavní kamery, u které jsem zvolil, aby se vykreslovaly pouze objekty ve *view frustu*, nikoliv však pozadí, nebo skybox. Obrázek je načten jako textura v *GUI Texture*(zastaralé GUI před Unity 4.6) komponentě objektu postaveného před kamerou

Jelikož kamery pracují nezávisle na sobě, načtený obrázek může mít libovolný poměr stran a velikost okna se může měnit, bylo nutné zajistit, aby hlavní kamera zabírala virtuální scénu se stejným poměrem stran jako ta zabírající pozadí. Ke kameře pozadí jsem proto připojil skript *Aspect Ratio*, který v intervalech $0,1s$ testuje, zdali nedošlo ke změně rozlišení nebo nebyl nahrán nový obrázek, a pokud došlo, upraví poměr stran obou kamer odpovídajícím způsobem.

Načítání obrázku je řešeno kaskádou „spusť skript a počkej na odpověď" skriptů. Tím prvním je z pochopitelných důvodů *InGameMenu*, to spustí *FileOpener*(viz obr. 4.3 a úryvek kódu nad ním), který otevře dialogové okno. Až se dialogové okno zavře, *FileOpener* zavolá metodu v komponentě *Aspect Ratio* kamery pozadí, předá jí cestu k souboru a tato metoda potom využije Unity třídy *WWW* k načtení a zpracování obrázku do textury, kterou posléze přepíše původní texturu v *GUI Texture* objektu s pozadím.

4.4 Mřížka, obsah scény a kalibrace

Před tím, než může uživatel začít (smysluplně) vkládat do virtuální scény objekty a povrchy, musí zajistit, aby externí (pozice a orientace) a interní (ohnisková vzdálenost) parametry hlavní kamery (od teď již pouze „kamera" nebo „virtuální kamera" ve větách se skutečnou kamerou) odpovídali skutečné kameře, ze které pochází obrázek v pozadí a která bude zdrojem video streamu. Tento krok by měl být podle INCASTu nezávislý na editoru a zcela automatický, rovněž by měl také vyřešit radiální a tangenciální zkreslení, které se pro práci v editoru neuvažuje. V tomto prototypu má nicméně uživatel možnost dosáhnout požadovaného efektu dvěma způsoby. Manuální kalibraci nebo poloautomatickou kalibrací. Obě tyto možnosti se nacházejí v nabídce pro kalibraci kamery, přístupné z hlavní nabídky.

4.4.1 Manuální kalibrace

Manuální kalibrace je probíhá tak, že uživatel směrovými klávesami WASD (toto je díky využití třídy správce uživatelského vstupu Unity nastavitelné) pohybuje s kamerou a snaží se docílit správné polohy a orientace, zároveň pak kolečkem myši mění zorné pole(FOV). Toto funguje následovně.

Ve scéně se nachází plocha s mřížkovanou texturou, která reprezentuje zem. Díky této mřížce má uživatel představu o vztahu skutečné a virtuální scény. Uživatelovým úkolem je (zdánlivě) umístit mřížku tak, aby odpovídala obrázku ze skutečné kamery. Tady se využívá toho, že proces kalibrace předpokládá, že počátek souřadnicového systému scény(skutečné i virtuální) se nachází na místě, odpovídajícím středu obrázku pozadí(videa, ze kterého pochází). Pakliže se skutečná kamera dívá na počátek souřadnicového systému, musí se na něj dívat i kamera virtuální. Vzhledem k tomu, že se kamera musí vždy dívat na jeden bod v prostoru, je její pohyb jistým způsobem omezený.

Tohoto faktu jsem využil a ve scéně jsem vytvořil objekt *CenterOfRotation*, který reprezentuje počátek souřadného systému, učinil jsem jej rodičovským objektem kamery a veškerý pohyb, který má kamera vykonat je prováděn prostřednictvím rotace tohoto objektu. Díky tomuto se kamera pohybuje po kulové ploše a stále sleduje svůj střed otáčení, umístěný v počátku. Uživateli pohybujícímu s kamerou se proto zdá, že naklání a otáčí mřížku.

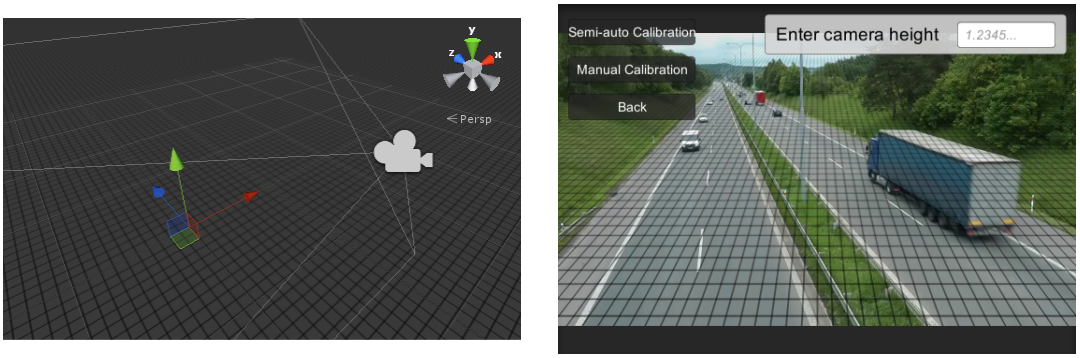
Na obrázku 4.4 se kamera nachází ve výchozím stavu, všimněte si neodpovídající mřížky v editoru napravo. Na obrázku 4.5 je správně zkalibrovaná scéna.

Kalibraci kamery mají na starosti komponenty (přesněji skripty), které jsou z důvodu přehlednosti součástí objektu kamery (obrázek 4.6). Manuální kalibraci zajišťuje komponenta *Camera Positioner*, která je spuštěna z *InGameMenu* stejným způsobem, jako načítání obrázku. Uživatelský vstup je zpracováván pomocí Unity třídy *Input* v metodě *Update* komponenty *Camera Positioner*. Krátký úryvek:

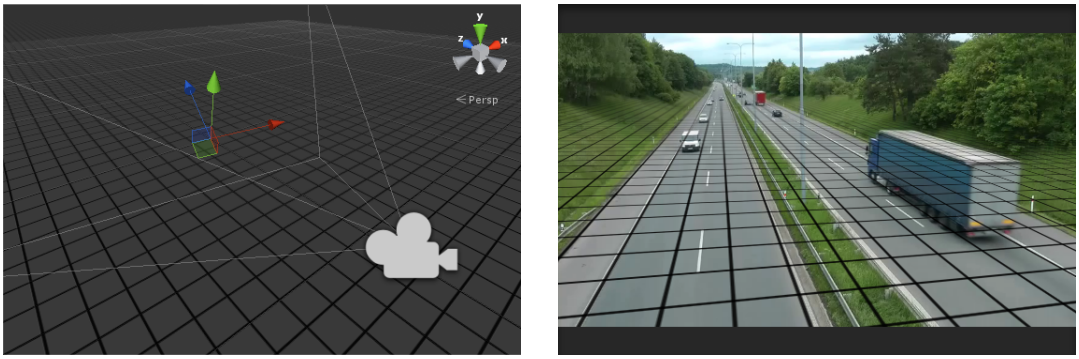
```
void Update () {
  if(Input.GetButtonDown("Fire2")){
    StopCalibrating();
  }

  if((tmpFloat = Input.GetAxis("Vertical")) != 0){
    CenterOfRotation.RotateAround(new Vector3(0, 0,0),
                                  new Vector3(-tmpFloat,0,0),
                                  Time.deltaTime*speed);
  }
```

Řetězce „Fire2" a „Vertical" odpovídají obecným vstupům, na které *Input Manager* mapuje konkrétní tlačítka, osy a jiné vstupy. Ve výchozím nastavení editoru scény reprezentuje „Fire2" pravé tlačítko myši a osa „Vertical" je řízena klávesami W a S. První *if* je příkladem konceptu, jež jsem se snažil dodržet ve všech případech, kde to bylo možné – pravé tlačítko myši značí ukončení činnosti.

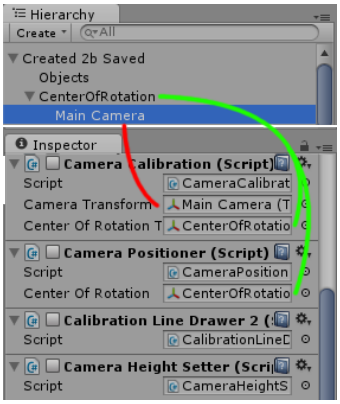
Z pohledu kalibrace je pak důležitá druhá podmínka, která testuje vstup vertikální osy; pokud tato nějaký vykazuje, dojde k rotaci *CenterOfRotation* kolem počátku podle osy X (viz obr 4.4 vlevo). Třetí parametr metody *RotateAround* zde nemá smysl rozebírat.



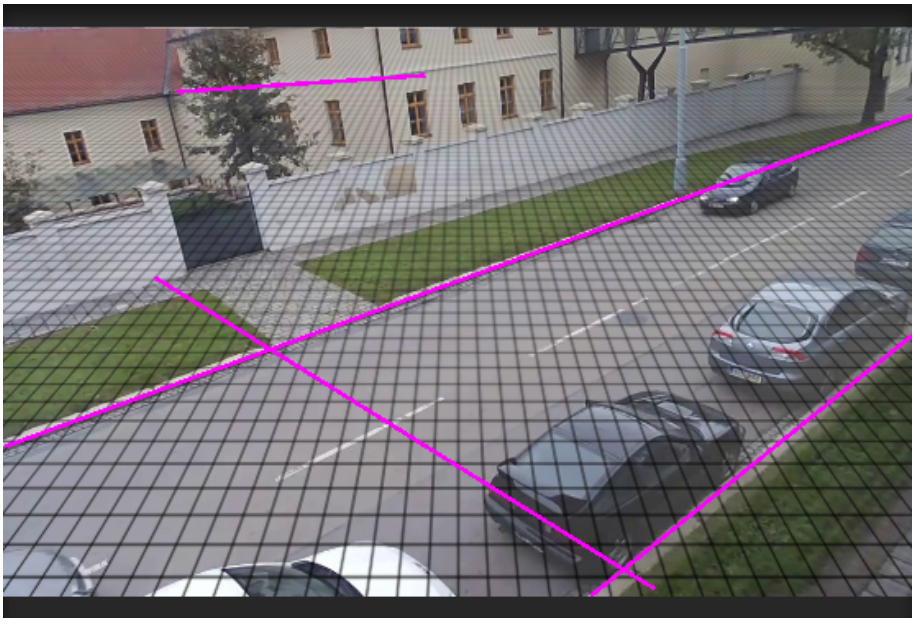
Obrázek 4.4: Výchozí stav natočení a zorného pole: Vlevo – Kamera sleduje objekt *CenterOfRotation* nacházející se v počátku souřadnicového systému. Vpravo – Editor scény v kalibračním menu.



Obrázek 4.5: Správně natočená kamera a zorné pole.



Obrázek 4.6: Skripty kamery a jejich vazby.



Obrázek 4.7: Poloautomatická kalibrace. Viditelné jsou dvě sady přímek, které na sebe jsou ve skutečném světě kolmé. Dvě přímky podél silnice jedna osa (nejčastěji Z), zbývající dvě jsou kolmé na ty první. Všimněte si rozmístění přímek, zvláště druhého páru – čím větší je na obrazovce úhel mezi rovnoběžnými přímkami, tím je výsledek přesnější

4.4.2 Poloautomatická kalibrace

Druhou možností, kterou uživatel má, je poloautomatická kalibrace, jedná se o preferovanější variantu, neboť rychlejší a snadnější. Jediné, co uživatel musí udělat, je zvýraznit dvě kolmé dvojce rovnoběžných přímek.

Nejprve se doporučuje vyznačit směr „dopředu." V Unity je dopředná osa Z a výsledná scéna tedy bude orientovaná tak, že první dvě přímky zakreslené v kalibraci budou odpovídat právě této ose. Pro kreslení přímek uživatel kliká levým tlačítkem myši a označuje tak jednotlivé body. Po prvním bodě každé přímky, se tato vykresluje od prvního bodu ke kurzoru, aby tato operace byla intuitivní. Vykreslování nevytvořené přímky má hlavní výhodu v tom, že se s její pomocí snáze kopírují hrany, přímky nebo objekty v řadě. Poloautomatická kalibrace se všemi přímkami umístěnými je vidět na obrázku 4.7.

Metoda poloautomatické kalibrace je založena na využití úběžníků. Nápad využít je k tomuto účelu jsem dostal díky metodě automatické kalibrace [3] nasazené v rámci případu užití v článku o INCASTu [21], neboť tato je také postavena na úběžnících.

Implementace poloautomatické kalibrace je rozdělena na dvě části. Tou první je kreslení přímek a získání úběžníků, tou druhou pak jejich použití pro samotné kalibrační úkony. Komponentou, která provádí druhou část je *CameraCalibration*, jenž zároveň i řídí průběh části první. Tato je spouštěna *InGameMenu* a první akcí, kterou provádí, je spuštění komponenty pro tvorbu přímek a získání úběžníků – *CalibrationLineDrawer2*. Viz obě tyto komponenty na obrázku 4.6. Stejně jako v předchozích případech i tady volající komponenta sama sebe zneaktivní nebo znepřístupní a čeká na dokončení komponenty spouštěné.

Kreslení přímek a hledání úběžníků v *CalibrationLineDrawer2* je prvním příkladem z konstrukčních skriptů založených na stavovém automatu. Jedná se o klasický koncept s jedním *switch* a *case* pro všechny stavy. Tento konstrukt se nachází v metodě *Update* a je tedy volán každý snímek. V konečném automatu se nachází osm stavů pro zaznačení každého z bodů kreslených přímek, speciální stav pro vytvoření úběžníků a tři stavy pro různé alternativy ukončení činnosti – dokončeno, selhání, zrušeno.

Body jsou vytvářeny tak, že každé volání *Update* jsou do pomocné proměnné *tmpPoint* uloženy souřadnice kurzoru na obrazovce (od 0 do 1 pro obě osy). Je-li stisknuto levé tlačítko myši, nastaví se příznak stisknutí *clicked*. V *case* odpovídajícímu současnému stavu je tento příznak otestován. Při nastaveném příznaku se pak bod použije k vytvoření instance přímky *Line*, popř. se přidá do již existující přímky, jedná-li se o její druhý bod, a nastaví stav na následující. V úryvku kódu z metody *Update* je nastaven *tmpPoint* a uveden první stav – tvorba prvního bodu:

```
tmpPoint = new ViewPortPoint (myCamera,
  myCamera.ScreenToViewportPoint(Input.mousePosition).x,
  myCamera.ScreenToViewportPoint(Input.mousePosition).y);
***
switch( state ){
 case States.ClickZp0:
  if(clicked){
    p0 = tmpPoint;
      tmpLine=new Line(p0);
    state = States.ClickZp1;
  }
  break;
```

Třídy *ViewPortPoint* a *Line* jsou pomocné třídy, které jsem vytvořil pro usnadnění práce s body a vykreslování přímek nezávislých na pozici a zorném poli kamery. Každý *ViewPortPoint* uchovává souřadnici bodu na obrazovce a umožňuje získat souřadnice bodu nacházejícího se ve scéně nějakou vzdálenost před kamerou tam, kam by se tento bod promítl při vytvoření paprsku z kamery do prostoru. *Line* v sobě uchovává dva *ViewPortPointy* a díky právě zmíněným souřadnicím je schopná vykreslovat v prostoru přímku, to provádí tak, že vytvoří ve scéně jeden *GameObject*, do nějž přidá standardní komponentu *LineRenderer*, které tyto souřadnice předá. Při každém zavolání metody *Line.Draw* jsou z bodu získány aktuální souřadnice a předány *LineRendereru*. Volání této metody se provádí u všech existující *Line* s každým vykresleným snímkem v rámci *CalibrationLineDrawer2.Update*.

Po nakreslení všech čtyř přímek se pomocí rovnic pro hledání průsečíku na rovině nalezou průsečíky obou párů, jak je uvedeno v kódu níže. Tyto průsečíky jsou rovněž instancemi třídy *ViewPortPoint* a proto z nich lze získat souřadnice v prostoru, čehož se využívá v druhé části. Po úspěšném dokončení činnosti této komponenty je zavolán příslušný delegát, očekávající dva *ViewPortPointy*, čímž je opět spuštěna činnost *CameraCalibration*, a provedena deaktivace.

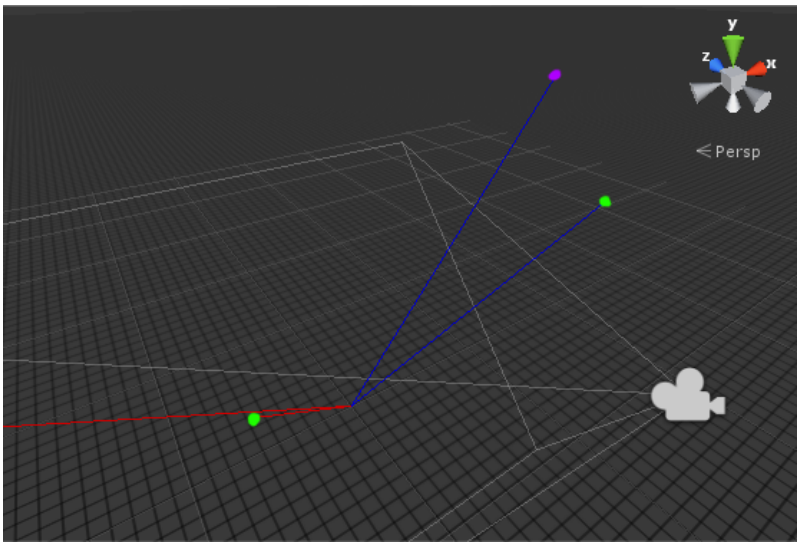
```
case States.PointsPlaced:
                        // vraci ViewPortPoint
  VanPointZ = lines[0].IntersectionWithAnotherLine(lines[1]);
  VanPointX = lines[2].IntersectionWithAnotherLine(lines[3]);
  state= States.Finished;
  break;
```

Ve druhé části je využito vytvořených úběžníků v podobě dvou *ViewPortPoint* ke kalibraci kamery. K tomu dochází ve třech fázích.

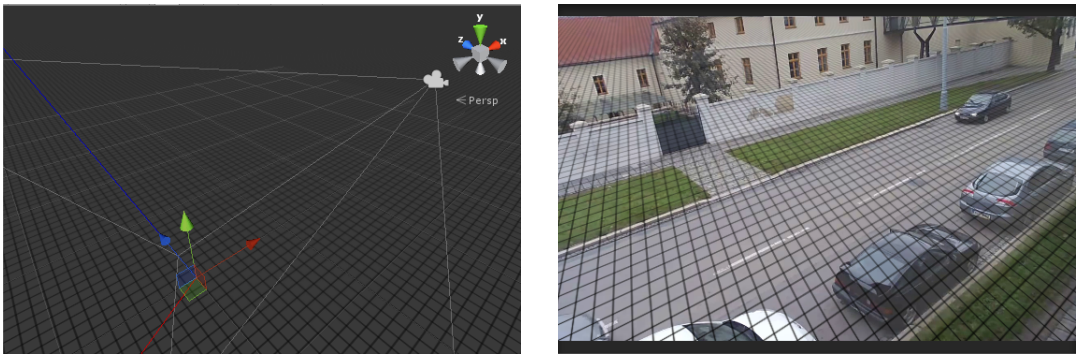
V rámci první fáze je zafixována osa Z. K tomuto účelu slouží úběžník získaný z prvních dvou přímek, s jehož pomocí je vypočítána pozice bodu ve velké vzdálenosti od kamery na místě, kam by se promítl paprsek vržení přes pixel, na kterém se nachází úběžník (toto není zcela přesné, neboť pozice úběžníku je popsána dvěma čísly s plovoucí řádovou čárkou a úběžník se navíc může nacházet mimo „obrazovku"). Zde jsem se rozhodl využít skutečnosti, že s rostoucí vzdáleností průsečíků dvou přímek, klesá úhel mezi nimi svíraný a promítnutý bod ve velké vzdálenosti (řádově 100000 jednotek souřadnicového systému) jsem využil jako poziční vektor rovnoběžný s vektorem odpovídajícím ose Z souřadnicového systému kamery. Na obrázku 4.8 je vidět modrá přímka vedoucí do fialově zvýrazněného bodu ve velké vzdálenosti, jenž byl sestrojen s pomocí zeleného bodu napravo, reprezentujícího úběžník. Tato modrá přímka je téměř rovnoběžná s osou Z SS kamery. Jelikož je tato nyní známa, je možné natočit *CenterOfRotation* tak, aby odpovídala ose Z virtuální scény. Scéna po zarovnání os je na obrázku 4.9.

Ve druhé fázi dochází k vyrovnání plochy XZ SS kamery s jejím ekvivalentem ve virtuální scéně. Toto funguje na obdobném principu jako fáze první. Bod promítnutý do dálky reprezentuje osu X nebo -X podle toho, na které „straně" se od dopředného směru (osa Z) nachází úběžník pro tuto osu. Díky tomuto bodu je možné dopočítat úhel mezi jeho pozičním vektorem a vektorem ve směru osy X SS scény a natočit *CenterOfRotation* odpovídajícím způsobem podél osy Z. V této fázi jsou již země virtuální scény a scény skutečné ve stejné rovině, viz obr. 4.10.

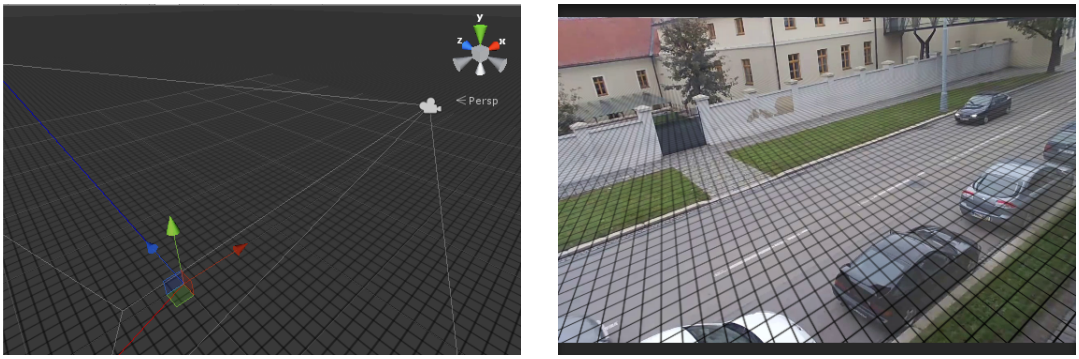
Poslední fáze je nastavení správného zorného pole *FOV*. Roviny XZ obou SS jsou sice zarovnány, osa X SS kamery však ale stále nesvírá z pohledu virtuální scény pravý úhel s osou Z. Toto jsem se rozhodl řešit iterativním přístupem, který vypadá následovně: V cyklu je



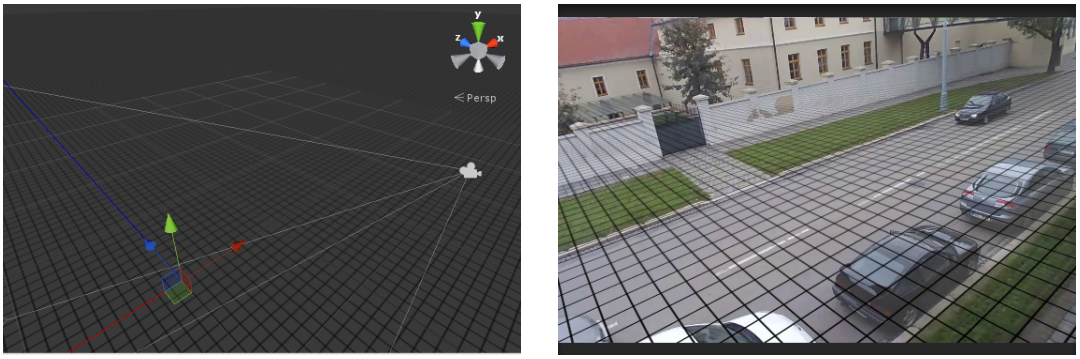
Obrázek 4.8: Poloautomatická kalibrace před zafixováním první osy. Zelené body – úběžníky v prostoru; fialový bod – vzdálený bod, promítnutí přes úběžník. Barevné přímky vedou od počátku k jednotlivým bodům: modré – týkají se osy Z; červené – týkají se X



Obrázek 4.9: Zafixovaná osa Z SS kamery. Tři barevné šipky reprezentují SS virtuální scény, modrá přímka a modrá šipka jsou zarovnány.



Obrázek 4.10: Plochy XZ scény a kamery si odpovídají/jsou rovnoběžné.



Obrázek 4.11: Kamera zkalibrována. Modrá přímka odpovídá ose Z, červená ose -X.

testována velikost úhlu svíraného mezi vektory reprezentujícími X a Z SS kamery. Pokud se tento liší od 90 ° o více než je maximální povolená chyba, přičte se k *FOV* hodnota kroku. Je-li v příštím kroku chyba větší, obrátí se znaménko kroku. Obrátí-li se znaménko kroku několikrát za sebou, znamená to, že metoda „přeskakuje" správnou hodnotu, proto se krok sníží na polovinu.

Při každé změně *FOV* se znehodnotí předchozí kalibrace. Celý tento proces (všechny fáze) nicméně probíhá periodicky s každým voláním *CameraCalibration.Update* a tudíž to nepředstavuje problém, neboť metoda konverguje ke správnému řešení.

4.4.3 Nastavení výšky kamery

Nastavení výšky, v jaké se kamera nachází, slouží k učení měřítka – poměru velikosti virtuální a skutečné scény. Není-li známa alespoň jedna délka z obsahu skutečné scény, která se nachází v obraze a nebo s ním stejně jako výška umístění kamery souvisí, není možně zjistit, jak velké objekty v obraze jsou. Např.: Kamera je správně zkalibrovaná a uživatel vytvořil objekt obalující zaparkované auto. Tento objekt by mohl mít dvojnásobnou velikost oproti originálu a uživatel by tuto skutečnost vůbec nemusel rozpoznat, kdyby nevěděl, že mřížka má základ jednu jednotku.

Tato možnost se nachází v menu kalibrace kamery (viz obrázek 4.4). Uživatel zadá číselný údaj a kamera se přiblíží nebo oddálí tak, aby výška odpovídala zadanému číslu, čímž se změní měřítko.

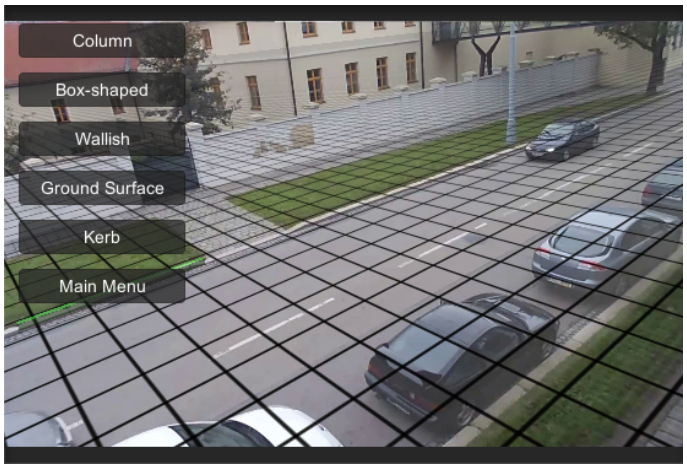
Vstupní pole je implementováno pomocí *UI InputField*, které při potvrzení vstupu zavolá metodu *SetHeightByFloatishString* komponenty *CameraHeightSetter* kamery(viz obr. 4.6). Metoda obsahuje pouze převod *stringu* na *float* a násobení vektoru skalárem.

4.5 Tvorba objektů

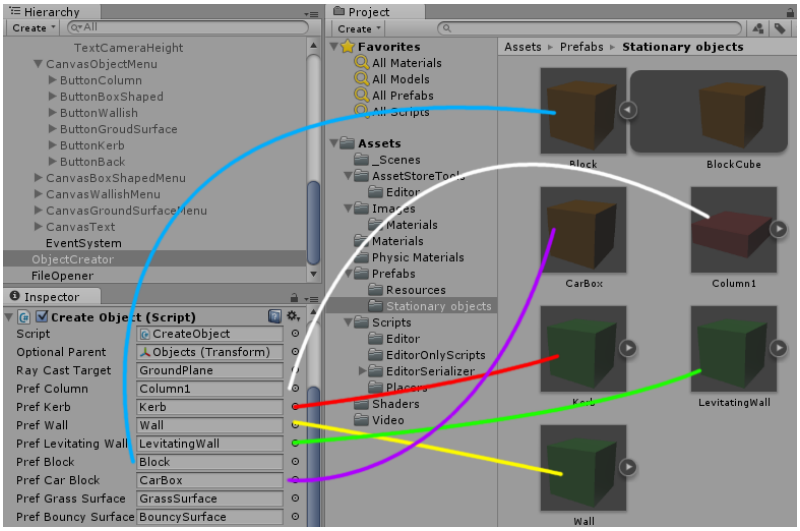
Hlavním účelem editoru scény je tvorba virtuálních objektu, které odpovídají objektům skutečným, a také označování povrchů se speciálními fyzikálními vlastnostmi, jako je travní plocha. Nabídka s výběrem objektů (obrázek 4.12) je přístupná pod položkou *Objects* hlavní nabídky.

Všechny objekty jsou vytvářený z předem připravených *prefabů*, z nichž každý je šablonou pro daný objekt. Všechny tyto *prefaby* jsou „instanciovány" skriptem pro tvorbu objektů *CreateObject* na objektu *ObjectCreator*. Tato komponenta je opět spouštěna *InGameMenu* a to pro tvorbu všech objektů, neboť funkcionalitu obstarávající manipulaci s nimi mají na starost jejich vlastní komponenty. Který objekt bude vytvořen závisí na parametru, který *InGameMenu* použije při volání *CreateObject.StartCreating*. *CreateObject* funguje tak, že v *Update* testuje vstup na stisknutí levého tlačítka myši (přesněji obecný vstup „Fire1"), pokud je toto zmáčknuto, provede se *ray cast* metodou *Physics.Raycast* s parametry takovými, aby byla zasažena pouze vrstva, ve které se nachází plocha *GroundPlane* reprezentující zem(ta, na které je textura mřížky). *Physics.Raycast* vrací informace o bodu, ve kterém došlo ke kontaktu. Na této pozici je potom vytvořen nový objekt. Vytvořenému objektu je nastaven výchozí rodič – objekt *Objects* (viz obr. 4.1) a pomocí delegátu je naplánováno opětovné spuštění *CreateObject* po dokončení tvorby objektu.

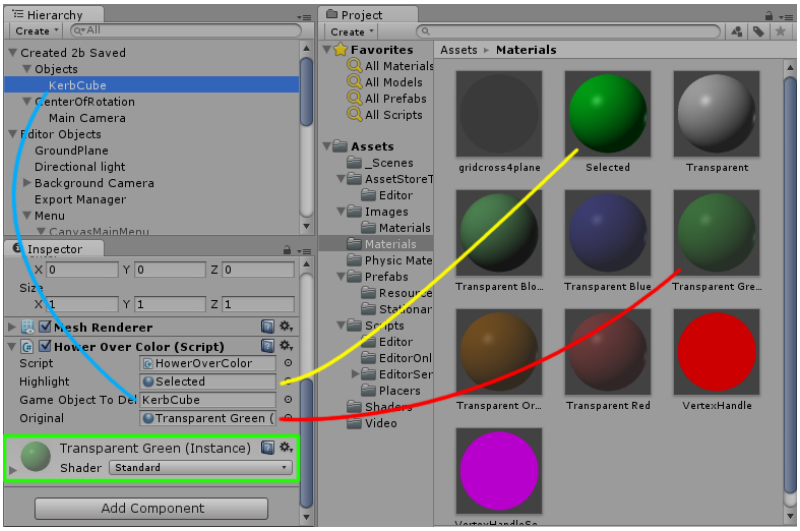
Každý vytvářený objekt má před vytvořením i po vytvoření komponentu *HoverOverColor* a přiřazený materiál udávající jeho barvu. Díky *HoverOverColor* se objekt zvýrazní při najetí myši a je jej také možné smazat. Zvýrazňování objektů je implementováno pomocí metod *MonoBehaviour.OnMouseOver* a *MonoBehaviour.OnMouseExit*, v nichž se nastaví aktuální materiál na ten s požadovanou barvou spolu s příznakem označení, jehož je využito v metodě *Update*, kde se testuje zároveň s uživatelským vstupem. Pokud uživatel zadá kombinaci Ctrl + pravé tlačítko myši, naplánuje se zničení objektu k tomuto účelu vybraného. Důvodem, proč není jednoduše zničen právě objekt, na kterém je umístěna komponenta, je to, že se tento nemusí nacházet na kořenové úrovni daného virtuálního objektu – viditelný objekt s *HoverOverColor* může být synovským objektem prázdného objektu. Tato komponenta a její vazby jsou znázorněny na obrázku 4.14.



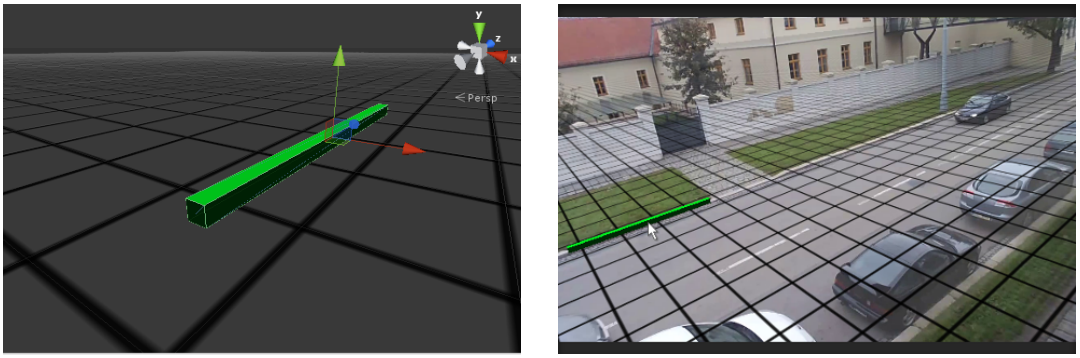
Obrázek 4.12: Nabídka pro tvorbu objektů: *Column* – sloup;*Box-shaped* – objekty ve tvaru kvádru;*Wallish* – zdi/vertikální plochy;*Ground Surface* – povrchy;*Kerb* – obrubník;*Main Menu* – návrat do hlavní nabídky.



Obrázek 4.13: Vazby mezi skriptem *CreateObject* a některými *prefaby*.



Obrázek 4.14: Vytvořený objekt KerbCube s komponentou *HoverOverColor* odkazující se na materiál s barvou pro zvýraznění (žlutá čára), původní materiál objektu(červená čára) a objekt, který bude smazán (modrá čára). Zelená rámeček – aktuálně nastavený materiál

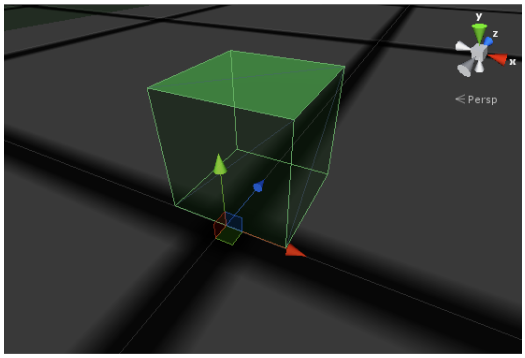


Obrázek 4.15: Vytvořený a označený obrubník.

4.5.1 Obrubník

Nejjednodušším objektem, je obrubník (Kerb). Jedná se objekt konstantní šířky a výšky, s proměnnou délkou. Vytváří se dvěma kliknutími myši, obdobně jako přímky v poloautomatické kalibraci. Jak už název napovídá, slouží objekt k vyznačování obrubníků viz krátký obrubník na obrázku 4.15.

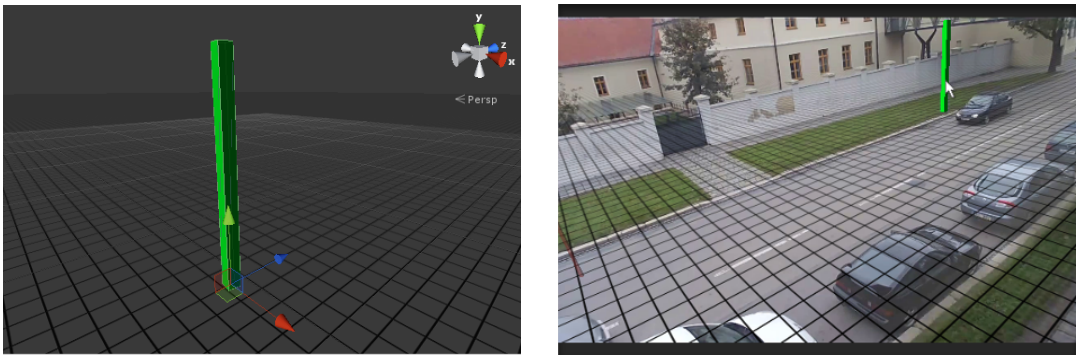
Pro tvorbu obrubníku je použita standardní *Cube*. Tato má konstantní rozměry (*scale*) a je umístěna v prázdném objektu způsobem viditelným na obrázku 4.16. Díky tomuto je možné měnit *scale* viditelné krychle pomocí prázdného otcovského objektu tak, že dochází k nárůstu rozměru krychle pouze ve kladném směru osy Z. Výchozí chování všech Unity primitiv je totiž škálování od středu objektu, tudíž na dvě strany v případě jedné osy.



Obrázek 4.16: Obrubník ve výchozím stavu. Šipky vycházejí z pozice prázdného otcovského objektu, viditelná krychle je vycentrována podle osy Z, nachází se ale pouze v kladných souřadnicích os Y a Z.

Umisťování obrubníku probíhá tak, že *CreateObject* (komponenta *ObjectCreatoru*) vytvoří objekt *Kerb*, jenž je prázdným objektem zmíněným výše. Po vytvoření je automaticky spuštěna komponenta *KerbPlacer*. V *KerbPlacer.Update* se nachází dvoustavový konečný automat v podobě *switch/case*. První stav slouží k natáčení a změně délky, druhý pak vyřadí prázdný objekt z hierarchie.

Změna délky a natáčení jsou implementovány pomocí *Physic.Raycast* na *GroundPlane*. Velikost objektu je spočtena z jeho vzdálenosti od bodu získaného *ray castem*. Pro natočení je využito metody *Transform.LookAt*, která otočí dopředný vektor lokálního souřadnicového systému (osa Z) směrem k cíli udanému parametrem podle osy udané dalším parametrem, pro který je použit normálový vektor k ploše dopadu *ray castu*.

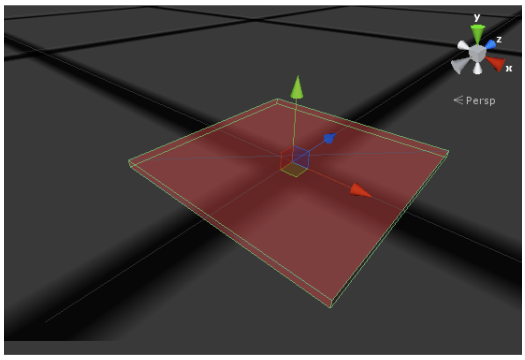


Obrázek 4.17: Vytvořený a zvýrazněný sloup.

4.5.2 Sloup

O něco složitějším objektem je sloup (Column). Jeho vlastnostmi jsou nastavitelná výška, natočení a „průměr" (nejedná se o válec, nýbrž o kvádr). Uživatel umisťuje sloup pomocí třech kliknutí myši, kde prvním vytvoří sloup na zvoleném místě, druhým potvrdí výšku a třetím natočení s průměrem. Sloup slouží k označování sloupů, kmenů apod. Příklad využití sloupu na pouliční osvětlení je na obrázku 4.17.

Podobně, jak je tomu u obrubníku, sestává i sloup ze dvou objektů, vnořený objekt tělesa, jímž je opět *Cube*, samotného sloupu je však umístěný ve svém otcovském elementu jiným způsobem (obr. 4.18). Důvodem je, že u sloupu se předpokládá jeho umístění do středu skutečného objektu a po nastavení výšky se bude měnit jeho průměr, tedy se bude zvětšovat a zmenšovat do všech stran, kromě výšky.



Obrázek 4.18: Sloup ve výchozím stavu. Šipky vycházejí z pozice prázdného otcovského objektu, kvádr je vycentrovaný podle os X a Z, nachází se však pouze v kladných hodnotách Y.

Umisťování sloupu provádí *Column1Placer*, komponenta rodičovského objektu. Opět je v ní implementován konečný automat. Tento má však již 5 tvůrčích stavů a jeden pro případ, kdy uživatel umisťování přeruší. Na rozdíl od obrubníku však sloup využívá i jiné objekty. Konkrétně se jedná o *HelpingQuad*, *instanciovaný* z *prefabu*, což je plocha určená pouze k tomu, aby ji zasáhl *ray cast*.

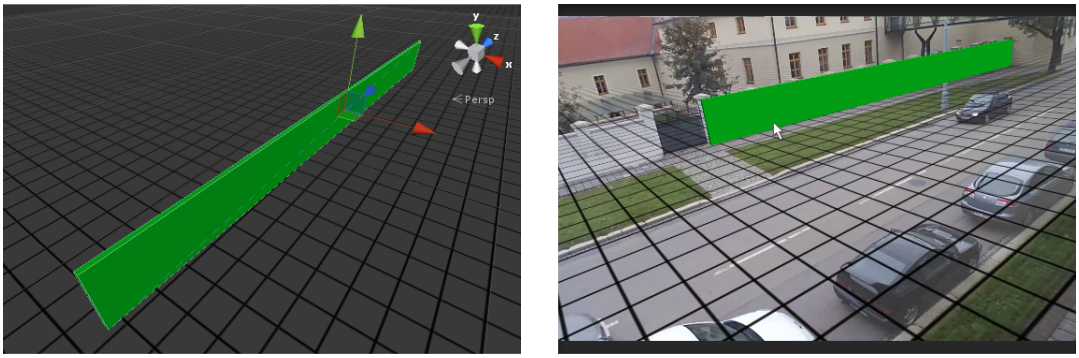
Průběh umisťování probíhá tak, že je opět nejprve vytvořen objekt sloupu s pomocí *CreateObject* a spuštěn skript pro umisťování – *Column1Placer*. V prvním stavu konečného automatu se vytvoří první *HelpingQuad*, otočený plochou směrem na kameru, ale kolmý k zemi (tzn. otočený na kameru podle osy Y). Tento *HelpingQuad* je vytvořen na stejné pozici jako sloup.

Druhý stav se čeká na kliknutí myši a periodicky (konečný automat se nachází v metodě *Update*) se provádí *ray cast* z místa kurzoru na *HelpingQuad*. Z pozice bodu dopadu se spočítá vertikální vzdálenost mezi tímto bodem a pozicí sloupu. Tato vzdálenost je pak použita pro nastavení výšky sloupu, díky čemuž tato kopíruje vertikální pozici kurzoru. Když uživatel stiskne levé tlačítko myši, přejde se do následujícího stavu.

Ve třetím stavu je zničen *HelpingQuad* a vytvořen nový, reprezentující zemi. Tento není nezbytný, neboť by místo něho mohla být použita přímo *GroundPlane*, jak je tomu u ostatních objektů, možnost nastavení průměru a rotace sloupu byla však implementována později než všechny ostatní obdobné funkce na jiných objektech a postup s *HelpingQuad* byl zvolen kvůli nezávislosti *GroundPlane* a úhlu, pod jakým může být sloup vztyčen, pokud by se měl nacházet na nakloněné rovině.

Čtvrtý stav slouží k změně průměru a otočení. Toto je implementováno podobně jako natáčení a nastavení délky u obrubníku. Každé volání *Update* se provede *ray cast* z pozice kurzoru na *HelpingQuad*. Vzdálenost mezi sloupem a bodem dopadu je použita pro nastavení *scale* v osách X a Z – tedy průměr. Zde je zvolen vhodný koeficient (0,1f – empirická hodnota), kterým je průměr vynásoben z toho důvodu, že udává-li vzdálenost mezi kurzorem a sloupem průměr, může být téměř nemožné zvolit tento správně u sloupů úzkých nebo nacházejících se v dálce. Bod dopadu je rovněž využit pro natočení sloupu tak, aby sledoval kurzor. Když uživatel klikne na levé tlačítko myši, přejde se do konečného stavu.

V konečném stavu je zničen druhý *HelpingQuad* a tato komponenta.



Obrázek 4.19: Vytvořený a zvýrazněný sloup.

4.5.3 Zeď

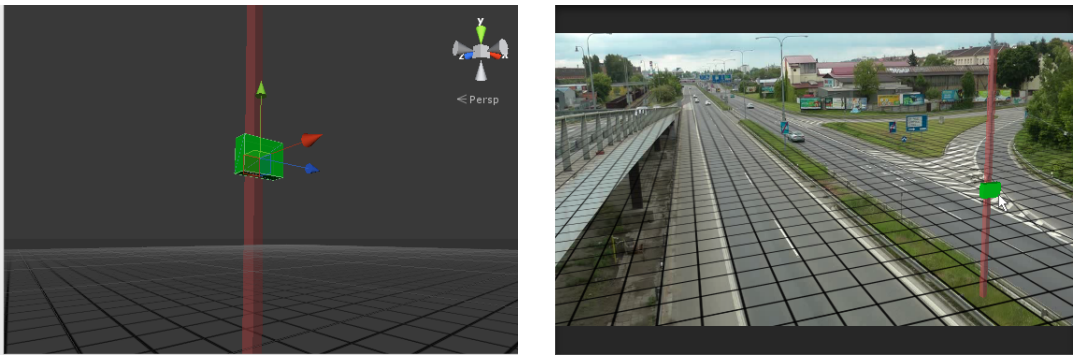
Zeď (Wall) je objekt konstantní šířky s nastavitelnou délkou a šířkou. Uživatel umisťuje zeď třemi kliknutími myši. Nejdříve vytvoří začátek zdí, potom tuto natáhne stejně jako v případě obrubníku a nakonec nastaví jeji výšku. Nejčastějším použitím zdi je z pochopitelných důvodů zeď (obrázek 4.19), hodí se však i pro jakékoliv jiné vertikální plochy.

Zeď se z pohledu implementace v mnoha ohledech podobá obrubníku. Dva zanořené objekty jsou umístěny stejně jako na obrázku 4.16. Polohování zdi obstarává skript *WallPlacer*. Vytváření začíná opět u *IngameMenu* a *CreateObject*. Konečný automat ve *WallPlacer* má čtyři stavy.

První stav nastavuje délku a natočení stejným způsobem jako *KerbPlacer*(obrubník). Po kliknutí myši se však navíc uloží poloha konce zdi do pomocné proměnné. Ve druhém stavu je vytvořena obdoba *HelpingQuad* v podobě *HelpingWall*, ty jsou vytvořeny z různých Unity primitiv (*Quad* a *Cube*), principiálně jsou však stejné.

Třetí stav slouží k nastavení výšky, toho je docíleno stejným způsobem jako u sloupu, tedy k nastavení výšky je použita vertikální vzdálenost mezi bodem dopadu *ray castu* a zdí. Zeď je reprezentována pozicí bodu uloženou v prvním stavu.

Poslední stav vyřadí prázdný rodičovský objekt z hierarchie a zničí jej spolu s touto komponentou.



Obrázek 4.20: Zvýrazněná levitující zeď před sloupem.

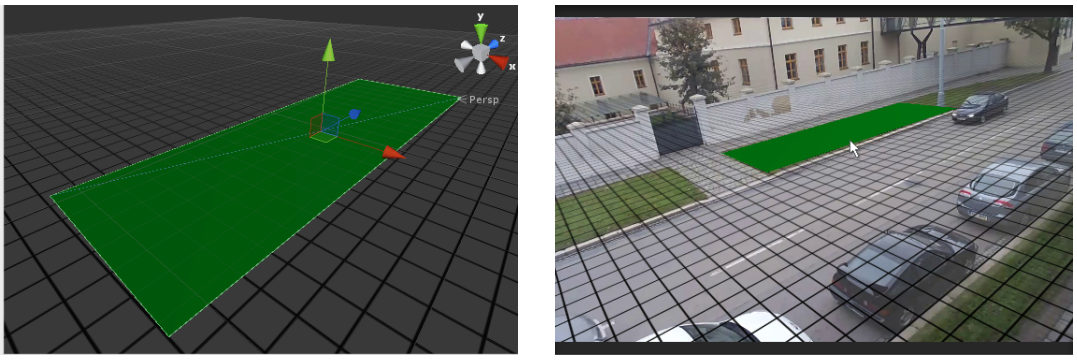
4.5.4 Levitující zeď

Levitující zeď (Levitating Wall) vychází z objektu zdi. Umísťuje se čtyřmi kliknutími myši. První dvě kliknutí slouží pro umístění základny zdi, jak je tomu u zdi obyčejné, před třetím potvrzovacím kliknutím nicméně uživatel volí výšku, ve které se základna nachází, teprve potom nastaví výšku samotné zdi posledním kliknutím. Motivací pro vytvoření levitující zdi jsou objekty jako reklamní cedule, dopravní značení, či jako na obrázku 4.20 krabice s elektronikou pro monitorovací kamery.

Jelikož se postup zahájení tvorby u všech objektů opakuje, uvedu již pouze zajímavé informace. Umísťování zdi provádí skript *LevitatingWallPlacer*.

Konečný automat v této komponentě ma oproti automatu ve *WallPlaceru* navíc jeden stav. Nový třetí stav tohoto automatu slouží k nastavení výšky základny. To probíhá stejným způsobem jako nastavování výšky s tím rozdílem, že místo *scale* objektu se mění jeho pozice v ose Y.

4.5.5 Povrchy



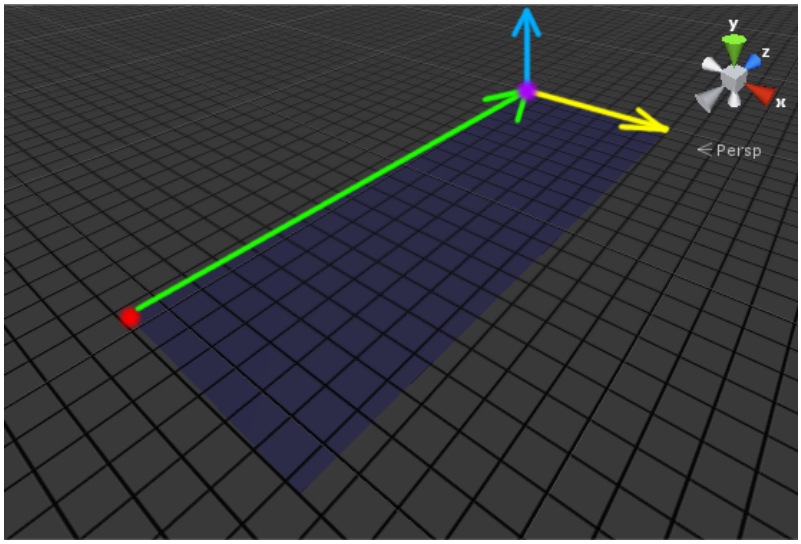
Obrázek 4.21: Zvýrazněný povrch.

Povrchy (Grass Surface, Bouncy Surface) představují způsob, jakým lze zvýraznit plocha se zajímavými fyzikálními vlastnostmi. Na obrázku 4.21 je *GrassSurface* pokrývající oblast trávníku. Ve fázi tvorby aplikace, je pak možné takto anotovaných míst využít k přidání tření, snížení odrazivosti atd. Uživatel umisťuje povrch třemi kliknutími. První opět zvolí počáteční bod, potom nastaví délku a natočení (viz obrubník) a v posledním kroku zvolí šířku.

Umísťování zprostředkovává *SurfacePlacer*. Postup je založen na konečném automatu, který má tři stavy. První stav odpovídá prvnímu stavu u zdi nebo obrubníku, jedná se o natáčení a nastavení délky pomocí *ray castu*.

Druhý stav slouží k přizpůsobení šířky. To probíhá tak, že se využije koncového bodu prvního stavu ke zhotovení vektoru popisujícího směr povrchu a následně je proveden vektorový součin mezi právě zmíněným vektorem a vektorem určeným koncovým bodem a bodem dopadu *ray castu* na *GroundPlane*. Oba tyto vektory se nacházejí v rovině země, tzn. že souřadnice Y všech jejich bodů jsou stejné, díky čemu je normálový vektor, který vznikl jejich součinem, orientovaný buďto nahoru, nebo dolů. Orientace se mění v závislosti na jakou stranu z pohledu povrchu dopadl *ray cast*. Tohoto faktu je využito při nastavování šířky (*scale* osy X), jejíž znaménko se řídí znaménkem Y souřadnice normálového vektoru. Viz obrázek 4.22.

V posledním stavu dojde k vyřazení prázdného objektu z hierarchie a jeho zničení spolu s tímto skriptem.



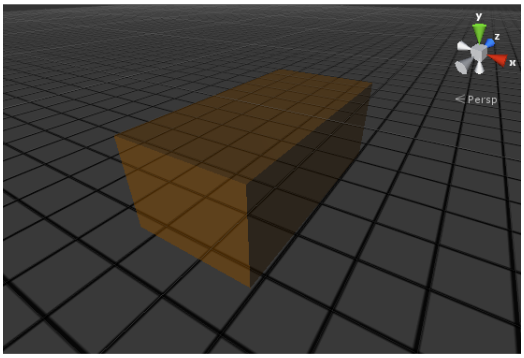
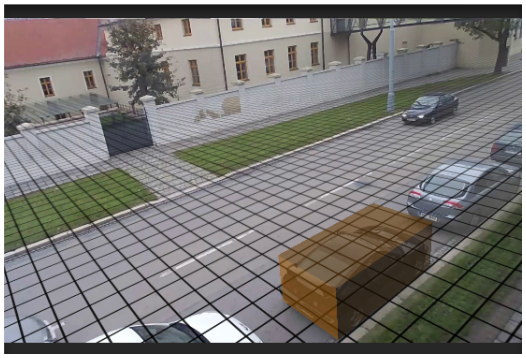
Obrázek 4.22: Nastavování šířky povrchu – vektory a body: *červená* – počáteční bod, *fialová* – koncový bod prvního stavu, *žlutá* – vektor od koncového bodu do místa dopadu *ray castu*, *modrá* – normálový vektor

4.5.6 Kvádr

Kvádr (Block) slouží k označování prostorových objektů, jejichž tvar připomíná kvádr. Takovými objekty mohou být parkující nákladní vozidla, popelnice, krabice, zastávky apod. Díky nastavitelnosti ve všech třech osách, je možné kvádrem nahradit i ostatní objekty. Ve většině případů to ovšem není výhodné, neboť může být velmi složité správně nastavit šířku zdi nebo sloupu, zvláště když se nachází daleko nebo jsou malé.

Uživatel může vytvořit kvádr čtyřmi kliknutími myši. Prvním kvádr umístí na zvolené místo, druhým potvrdí délku a natočení jeho podstavy, třetím zvolí šířku a posledním kliknutím potvrdí výšku.

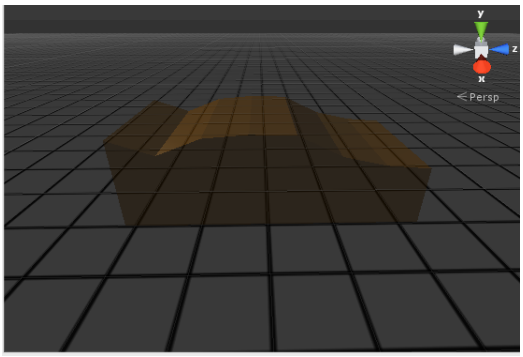
Všechny kroky pro vytvoření kvádru jsou již popsány u ostatních objektů. Umísťování je zajištěno skriptem *BlockPlacer*.

Obrázek 4.23: Automobil v kvádru.

4.5.7 Autokvádr

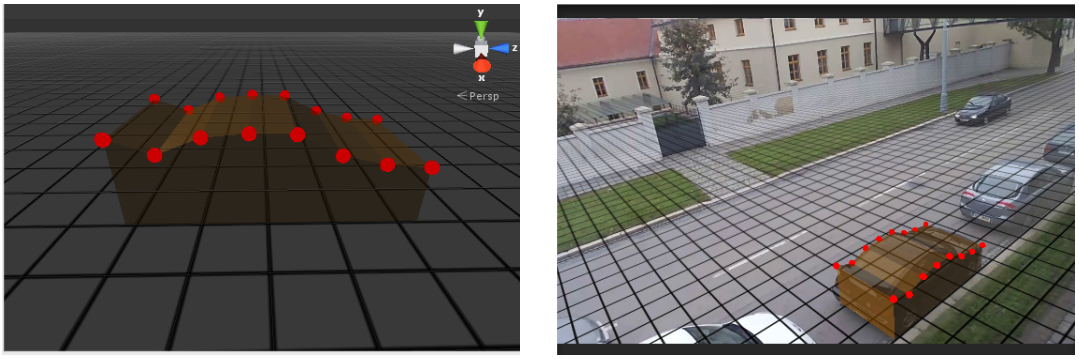
Autokvádr (CarBox, v menu Carish Block) je objekt speciálně navržený pro práci s osobními a nákladními automobily, kteréžto mají charakteristickou siluetu. Při tvorbě autokvádru nejprve uživatel vytvoří kolem vozidla kvádr pomocí čtyř kliknutí, tak jak je tomu u kvádru obyčejného, namísto dokončení akce však následuje fáze ručního polohování vrcholů horní podstavy, který je ukončen až stiskem pravého tlačítka myši. Příklad vytvořeného autokvádru je na obrázku 4.24, na obrázku 4.25 pak autokvádr před závěrečným kliknutím.




Obrázek 4.24: Automobil v autokvádru.

Všechny předchozí virtuální objekty využívaly jako model objektové primitivum *Cube*, tedy kostku. Pro účel autokvádru je však za běhu programu vygenerována meš s požadovaným počtem vrcholů a to navíc bez pomocného prázdného rodičovského objektu. Generování meše, úvodní fázi umisťování i vytváření pomocných koulí (*Sphere*) pro nastavování vrcholů má na starosti skript *CarBoxMeshMaster*. Koule jsou vytvářeny z *prefabu VertexHandle*, což je koule s materiálem konstantní barvy neberoucí v potaz světlo (*unlit*) a připojeným skriptem *VertexHandler*. Vše postupně popíši.

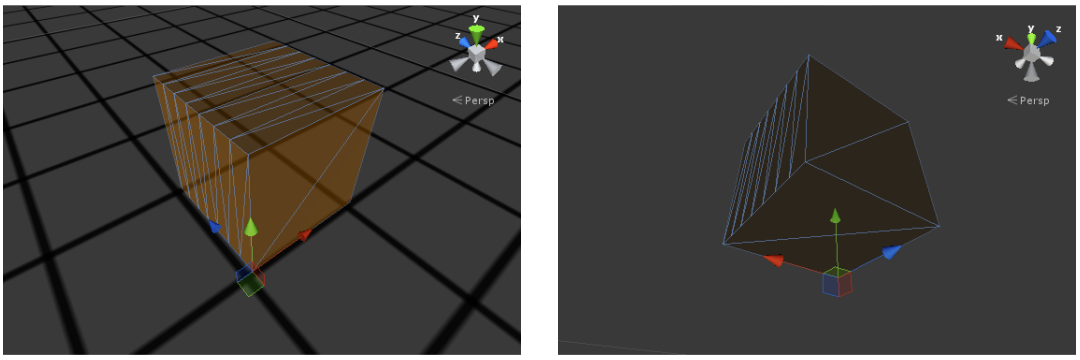
Činnost začíná vytvořením objektu autokvádru. Jakmile je objekt vytvořen, automaticky je spuštěna komponenta *CarBoxMeshMaster*, která ihned v rámci metody *start* vygeneruje meš (rozvedu později), kterou následně nahradí *placeholder* meš v komponentě *Mesh Filter*.



Obrázek 4.25: Automobil v autokvádru před dokončením. Červené body značí uchopitelné a přizpůsobitelné vrcholy.

V rámci činnosti konečného automatu v metodě *Update* jsou ve čtyřech stavech nastaveny rozměry a rotace stejným způsobem, jako je tomu u kvádru. V pátém stavu jsou vytvořeny všechny *VertexHandly* a jejich *VertexHandlerům* přiřazeny indexy vrcholů, které mají ovládat. *VertexHandlery*(komponenty) jsou navíc spárovány, díky čemuž je možné jedním *VertexHandlem* ve scéně pohybovat dvěma protilehlými zároveň. V posledním stavu se čeká na ukončení práce s *VertexHandly* stisknutím pravého tlačítka myši (resp. jakýmkoliv jiným vstupem namapovaným na obecný vstup „Fire2"). Při ukončení činnosti jsou zničeny všechny *VertexHandly* a zničí se i *CarBoxMeshMaster*.

Základní meš se sestává z pole vrcholů, pole indexu vrcholů reprezentující trojúhelníky a pole normál. Při generování meše ve *CarBoxMeshMaster* jsou nejdříve spočteny souřadnice bodů tak, aby tvořily krychli znázorněnou na obrázku 4.26. Potom jsou vytvořeny vrcholy pro každou stranu, které se každý bod zúčastní, jeden. Díky tomu, budou správně vytvářeny normály a nasvětlování bude u hran správně ostře přecházet. Dalším krokem je vytvoření pole indexu do pole vrcholů. Každé tři indexy reprezentují trojúhelník. Vrcholy jsou řazeny po směru hodinových ručiček, při pohledu z vykreslované strany. Posledním krokem je spočtení normály pro každý vrchol, což obstarává metoda *Mesh.RecalculateNormals*.



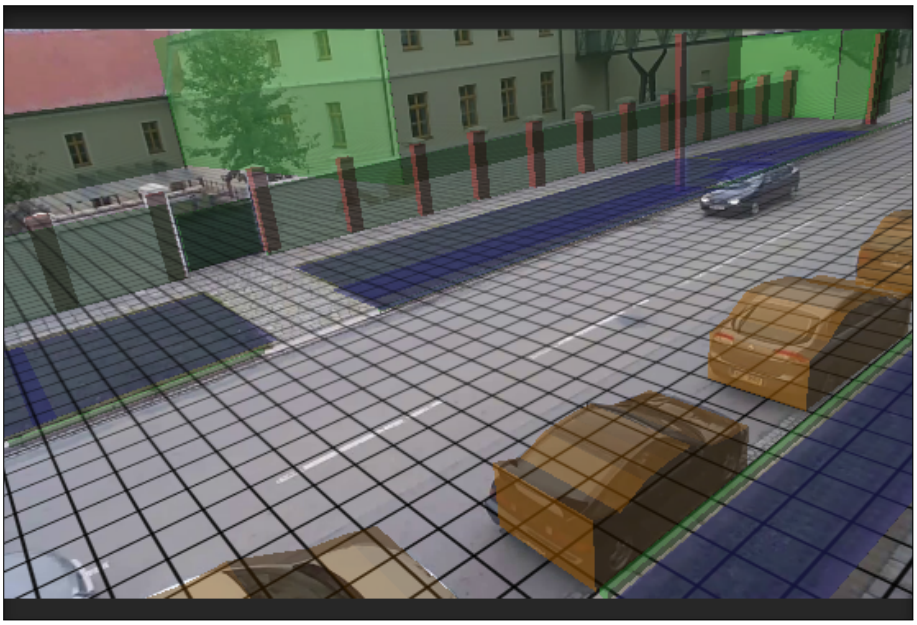
Obrázek 4.26: Vygenerovaná meš. Přední, spodní a zadní strana jsou tvořeny pouze dvěma trojúhelníky. Levá, pravá a horní strana mají každá 2∗ počet dílů trojúhelníků.

CarBoxMeshMaster umožňuje nastavit, na kolik dílů bude autokvádr rozdělen, tj. jak bude vygenerována meš. Dále *CarBoxMeshMaster* nabízí metodu *MoveVertex*, která nastaví pozici vrcholu zadaného indexem na novou hodnotu. Tuto metodu volá *VertexHandler*.

VertexHandler řídí činnost objektu *VertexHandle*. V metodě *Update* je testována kombinace vstupů a příznaku, odpovídající kliknutí a táhnutí. Táhne-li uživatel s *VertexHandle*,

Y souřadnice pozice je modifikována v závislosti na pohybu myši ve vertikálním směru, tedy nikoliv pomocí *ray castu*. Při každé takové změně pozice zavolá *VertexHandler* metodu *CarBoxMeshMaster.MoveVertex* a zároveň informuje *VertexHandler*, který s ním je spárován.

4.5.8 Příklady plně anotované scény



Obrázek 4.27: Plně anotovaná scéna: *červená* – sloupy, *modrá* – povrchy,*zelená* – zdi a obrubníky, *oranžová* – autokvádry

4.6 Export obsahu scény

Když uživatel dokončí editaci scény(viz obr. 4.27 a 4.28), bude ji chtít exportovat, aby ji mohl zaslat tvůrcům výsledné aplikace. Možnost *Export* v hlavním menu vytvoří XML soubor s obsahem scény v adresáři s editorem.

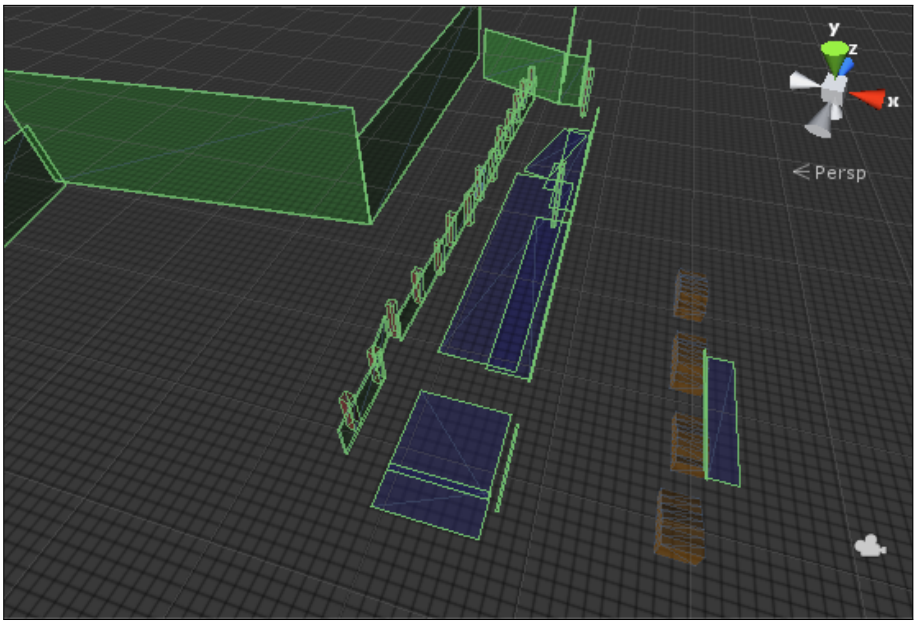
Unity nenabízí žádné vestavěné řešení pro ukládáni hry nebo snadný export dat. Původně jsem jako řešení tohoto problému zvolil známý plugin *UnitySerializer*[1] od Mika Talbota, který umožňuje právě ukládání stavu aplikace. Tento jsem využíval do doby, než jsem implementoval autokvádr – nepřišel jsem na způsob jak s tímto pluginem ukládat procedurálně generované meše, s největší pravděpodobností to však nepodporoval.

Inspirován *UnitySerializerem* jsem proto napsal jednoúčelový *EditorSerializer*, který slouží k ukládání a načítání objektů do/z XML souboru. *EditorSerializer* tvoří třídy *EditorSerializer*, *SerializableEditorData*, *DataGameObject*, *EditorFile* a *MyXmlSerializer*.

Třída *EditorSerializer* je řídící a pomocí ní, se provádí celý proces serializace. Instanci této třídy vytvoří, kořenový objekt všech vytvořených objektů ji předá a serializační metodu zavolá skript *ExportManager* objektu *Export Manager* nacházejícího se ve scéně.

MyXmlSerializer převádí objekty na XML řetězce a naopak, využívá při tom standardní třídu *XmlSerializer*.

[1] Již není dále vyvíjen a doména vypršela – http://whydoidoit.com/



Obrázek 4.28: Plně anotovaná scéna z jiného úhlu pohledu v Unity Editoru

EditorFile slouží pro operace se soubory a umožňuje provést třeba otevření, načtení a zavření v jedné metodě. Nejpoužívanější však jsou *SerializeFromFile* a *DeserializeFromFile* využívající *MyXmlSerializer*.

SerializableEditorData představuje seznam všech *DataGameObjectů*, které se mají serializovat a sám je serializovatelný.

Objekty třídy mohou být serializovány, pouze pokud není tato složená z jiných tříd. Výjimku tvoří třídy, které jsou samy serializovatelné. Komponenty v Unity vesměs serializovatelné nejsou. Tento problém řeší *DataGameObject*. Jedná se o datovou třídu s několika vnořenými třídami pro ukládání dat z komponent. Z pohledu exportování scény jsou důležitá data: pozice, rotace, měřítko, odkaz na rodičovský *Transform*, meš a vlastnosti kamery. Kromě odkazu na rodiče a meši, jsou všechny tyto informace ukládány do tříd *DataGameObject.MyTransform* a *DataGameObject.MyCamera*, které jsou serializovatelné. Meš je uložena přímo.

Problém představovalo uchování hierarchické vazby mezi objekty. To je vyřešeno tak, že si nejprve *DataGameObject* uloží skutečnou referenci na existující objekt, těsně před serializací se tento vyhledá v seznamu všech objektů, které mají být serializovány, a zjistí se jeho index v tomto seznamu. *DataGameObject* si pak tento index pamatuje.

Při deserializaci jsou potom pro všechny objekty vytvořeny příslušné *GameObjecty*, do nichž jsou vkládány nové komponenty, nebo jsou měněny hodnoty v jejich stávajících komponentách tak, aby odpovídali hodnotám v *DataGameObject*. Pro znovuvytvoření hierarchie objektů se využije uložených indexů.

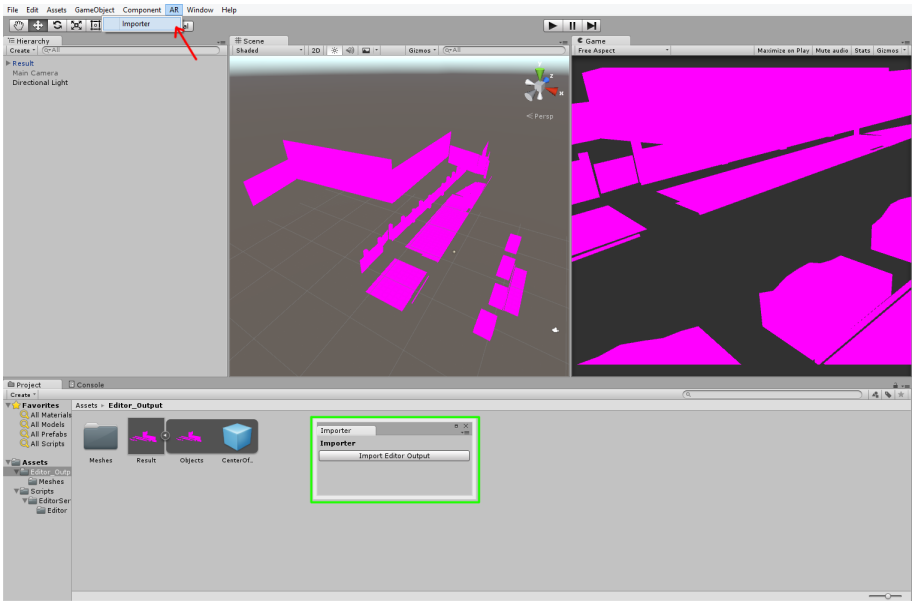
4.7 Import exportované scény

Importování scény na straně tvůrce aplikace předpokládá použití *EditorSerializeru*. K tomuto účelu byl vytvořen skript do editoru *Importer*, ten umožňuje uživateli jedním kliknutím načíst exportovanou scénu a vytvořit z ní *prefab*. Okno *Importeru* se otevírá z hlavní nabídky, záložka *AR*, viz obrázek 4.29. Díky konceptu souborů *.unityPackage*, které umožňují

vkládat *Assets* do projektu pouhým dvojitým kliknutím, je pro uživatele velmi snadné si *Importer* nainstalovat.

Importer dědí od třídy *EditorWindow*, jenž poskytuje funkcionalitu potřebnou pro přizpůsobení Unity Editoru. Jeho činnost je implementována v metodě *OnGUI*, kde se testuje, zda-li bylo stisknuto tlačítko „Import Editor Output." Pokud bylo, vytvoří se unikátní adresářová struktura ve složce *Assets*, uživatel je pomocí dialogového okna vyzván aby zvolil soubor a po jeho zvolení jsou z něj deserializována všechna data a vytvořeny *GameObjecty* tak, jak je to popsáno v předchozí sekci. Následně je kořenový objekt (včetně všech jeho potomků) pomocí *PrefabUtility* přeměněn v *prefab* a uložen do adresářové struktury.

Importer je implementován ve stejném projektu, jako editor scény, uživatel editoru scény se s ním však nesetká.



Obrázek 4.29: Nový projekt s *Importerem* a naimportovanou scénou z obrázků 4.27 a 4.28 bez textur(žádoucí chování). *Červená šipka* – možnost v menu pro otevření okna *Importeru*; *zelený rámeček* – okno *Importeru*

Závěr

V rámci této práce jsem vytvořil prototyp interaktivního editoru scény do *proof of concept* modelu INCAST prosazujícího možnosti nasazení stacionárních kamer v oboru rozšířené reality. Tento editor umí vytvářet objekty jako jsou zdi, sloupy, kvádry, částečně upravitelné hranaté objekty sloužící k modelování tvarů aut nebo vyznačovat plochy se speciálními vlastnosti. Editor rovněž umožňuje plně manuální a poloautomatickou kalibraci kamery. Dodal způsob jak výstup tohoto editoru využít v podobě *Importeru*.

Na závěr bych chtěl navrhnout vylepšení řešící některé nedostatky současné implementace. Při editaci scén se dá narazit na několik specifických problémů. Víme-li například o existenci nějakého objektu, překážky nebo budovy nacházejících se sice mimo zorné pole kamery, ale významným způsobem se projevujících ve výsledné aplikaci (např. odražení balónku nebo projektilu), není v současné době možné tyto přidat před fází tvorby výsledné aplikace (obdelník AR Content Creation 2.1), což není žádoucí, neboť se jí typicky(zatím tedy čistě hypoteticky) nezúčastňuje člověk, který obsluhoval editor (zadavatel, vlastník kamery, člověk znalý scény...). Lidé zpracovávající výstup editoru scénu znát nemusejí a případné přidávání objektů pro ně může být obtížné a neefektivní, musí-li kupříkladu dohledávat pozici ze satelitních nebo obyčejných snímků. Dalším problémem při editaci může být přesné přidávání objektů nacházejících se ve větší vzdálenosti nebo za nějakou překážkou.

První dva problémy by bylo možné vyřešit implementací funkcí *Zoom Out* resp. *Zoom In*, kdy *Zoom In* by přibližoval oblast pod kurzorem. Všechny tři zmíněné problémy by mohl elegantně řešit mód volné kamery, kdy by bylo uživateli umožněno s kamerou volně létat prostorem. Při pohybu kamery se samozřejmě ztratí ze zorného pole obraz v pozadí, toto však pro člověka znalého scény nemusí vůbec představovat problém, protože si může nejdříve část virtuální scény vymodelovat a pak se orientovat jen podle virtuálních objektů a mřížky. Obě řešení mají svá pro i proti, volný pohyb kamery je nezbytný při přesném vytváření objektů v zákrytu, přibližovací metoda se ale více hodí pro práci na delší vzdálenosti.

Za předpokladu, že projekt INCAST bude nadále pokračovat a budou vytvářeny další demonstrační aplikace či videa, mohla by, dle mého názoru, dobře působit možnost ničit pohybující se skutečné objekty, zejména silniční vozidla. Toho by mělo být možné dosáhnout mnoha způsoby. Navrhnu dva:

Pro pohybující se *bounding boxy* je relativně jednoduché spočítat vektor rychlosti pro predikci pohybu. Hodnota pixelů pod předpokládaným budoucím výskytem BB by se mohla ukládat do bufferu „s prázdnou silnicí," který by byl vozidly neustále aktualizován. Pokud by bylo vozidlo zničeno, budou pixely pod jeho současným BB nahrazeny těmi z bufferu. Druhým způsobem by mohlo být ukládání všech neměnných/málo se měnících pixelů.

Buffer by nemusel být obnovován každý snímek, ale jen jednou za čas. Důležité je mít obrázek „prázdný" na místech, kde na tom záleží. Stacionární kamery jsou pro tuto činnost více než vhodné.

Literatura

[1] Bunnun, P.; Damen, D.; Calway, A.; aj.: Integrating 3D object detection, modelling and tracking on a mobile phone. In *ISMAR*, Nov 2012, s. 273–274.

[2] Chen, J.; Turk, G.; MacIntyre, B.: A non-photorealistic rendering framework with temporal coherence for augmented reality. In *ISMAR*, Nov 2012, s. 151–160.

[3] Dubská, M.; Sochor, J.; Herout, A.: Automatic Camera Calibration for Traffic Understanding. In *BMVC*, 2014.

[4] Dubská, M.; Szentandrási, I.; Zachariáš, M.; aj.: Poor Man's SimulCam: Real-Time And Effortless MatchMoving. In *ISMAR*, 2013, ISBN 978-1-4673-4661-0, s. 1–3.

[5] Fiala, M.: Magic Mirror System with Hand-held and Wearable Augmentations. *2014 IEEE Virtual Reality (VR)*, 2007: s. 251–254, doi:http://doi.ieeecomputersociety.org/10.1109/VR.2007.352493.

[6] Fiala, M.: Designing Highly Reliable Fiducial Markers. *IEEE PAMI*, ročník 32, č. 7, July 2010: s. 1317–1324, ISSN 0162-8828.

[7] Franke, T.: Delta Light Propagation Volumes for mixed reality. In *ISMAR*, Oct 2013, s. 125–132.

[8] Herout, A.; Szentandrási, I.; Zachariáš, M.; aj.: Five Shades of Grey for Fast and Reliable Camera Pose Estimation. In *CVPR*, 2013.

[9] Jachnik, J.; Newcombe, R.; Davison, A.: Real-time surface light-field capture for augmentation of planar specular surfaces. In *ISMAR*, Nov 2012, s. 91–97.

[10] Kaltenbrunner, M.; Bencina, R.: reacTIVision: A Computer-vision Framework for Table-based Tangible Interaction. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction*, TEI '07, New York, NY, USA: ACM, 2007, ISBN 978-1-59593-619-6, s. 69–74, doi:10.1145/1226969.1226983.
URL http://doi.acm.org/10.1145/1226969.1226983

[11] Kato, H.; Billinghurst, M.: Marker Tracking and HMD Calibration for a Video-Based AR Conferencing System. In *IWAR*, 1999, ISBN 0-7695-0359-4.

[12] Kato, H.; Billinghurst, M.; Poupyrev, I.; aj.: Virtual object manipulation on a table-top AR environment. In *Augmented Reality, 2000. (ISAR 2000). Proceedings. IEEE and ACM International Symposium on*, 2000, s. 111–119, doi:10.1109/ISAR.2000.880934.

[13] Klein, G.; Murray, D.: Parallel Tracking and Mapping for Small AR Workspaces. In *ISMAR*, Nara, Japan, Nov. 2007.

[14] Knorr, S.; Kurz, D.: Real-time illumination estimation from faces for coherent rendering. In *ISMAR*, Sept 2014, s. 113–122.

[15] Leizea, I.; Alvarez, H.; Aguinaga, I.; aj.: Real-time deformation, registration and tracking of solids based on physical simulation. In *ISMAR*, Sept 2014, s. 165–170.

[16] Newcombe, R. A.; Lovegrove, S.; Davison, A.: DTAM: Dense tracking and mapping in real-time. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, Nov 2011, ISSN 1550-5499, s. 2320–2327, doi:10.1109/ICCV.2011.6126513.

[17] Pintaric, T.; Kaufmann, H.: Affordable Infrared-Optical Pose Tracking for Virtual and Augmented Reality. In *IEEE VR Workshop on Trends and Issues in Tracking for Virtual Envs*, 2007, ISBN 978-3-8322-5967-9, s. 44–51.

[18] Reitmayr, G.; Drummond, T.: Going out: robust model-based tracking for outdoor augmented reality. In *ISMAR*, 2006, s. 109–118, doi:10.1109/ISMAR.2006.297801.

[19] Rohmer, K.; Buschel, W.; Dachselt, R.; aj.: Interactive near-field illumination for photorealistic augmented reality on mobile devices. In *ISMAR*, Sept 2014, s. 29–38.

[20] Schöps, T.; Engel, J.; Cremers, D.: Semi-Dense Visual Odometry for AR on a Smartphone. In *ISMAR*, September 2014.

[21] Szentandrási, I.; Zachariáš, M.; Kajan, R.; aj.: INCAST: Interactive Camera Streams for Augmenting Surveillance Cams. In *ISMAR, 2015*, před vydáním.

[22] Vacchetti, L.; Lepetit, V.; Fua, P.: Combining Edge and Texture Information for Real-Time Accurate 3D Camera Tracking. In *ISMAR*, 2004, ISBN 0-7695-2191-6, s. 48–57, doi:10.1109/ISMAR.2004.24.
URL http://dx.doi.org/10.1109/ISMAR.2004.24

[23] Woo, G.; Lippman, A.; Raskar, R.: VRCodes: Unobtrusive and active visual codes for interaction by exploiting rolling shutter. In *ISMAR*, Nov 2012, s. 59–64.

Příloha A

Plakát