Mendelova univerzita v Brně
Provozně ekonomická fakulta

Vizualizace virtuálních objektů v reálném prostoru s využitím nástroje Unity

Diplomová práce

Vedoucí práce:
Ing. David Procházka, Ph.D.

Radek Mikulka

Brno 2015

Tímto bych velice rád poděkoval Ing. Davidu Procházkovi, Ph.D., vedoucímu mé diplomové práce, za cenné rady a trpělivost, které mi poskytl při psaní této práce. Dále bych rád poděkoval své rodině, přítelkyni a všem účastníkům testů použitelnosti aplikace.

Čestné prohlášení

Prohlašuji, že jsem tuto práci: **Vizualizace virtuálních objektů v reálném prostoru s využitím nástroje Unity** vypracoval/a samostatně a veškeré použité prameny a informace jsou uvedeny v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*

Jsem si vědom/a, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 Autorského zákona.

Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

V Brně dne 30. listopadu 2016 _____

Abstract

Mikulka, Radek, Visualization of virtual objects in the real space using Unity, Master thesis. Brno, 2016.

This master thesis is focused on visualization of virtual objects in the real space. The introductory section provides a brief description of Kinect and Unity technologies. Moreover, similar state-of-the-art projects are outlined with regard to the used methodics of interaction. Subsequently, a review of useful libraries is provided. On the basis of these information, the methodics of my project is formulated. The subsequent part of the thesis deals with the implementation of experimental application in the C# programming language. Finally, the application is evaluated and there is a discussion of its pros and cons.

Keywords

augmented reality, Microsoft Kinect, Unity3D, C#, image processing

Abstrakt

Mikulka, Radek, Vizualizace virtuálních objektů v reálném prostoru s využitím nástroje Unity, Diplomová práce. Brno, 2016.

Diplomová práce se věnuje vizualizaci virtuálních objektů v reálném prostoru. Úvodní část obsahuje stručný popis použité technologie Kinect a Unity3D. Dále je posouzen aktuální stav řešené problematiky, kde se práce zaměřuje na metodiky k řešení podobných projektů. Následně je provedena rešerše použitelných knihoven. Poslední část se zabývá vlastní realizací řešení daného problému v programovacím jazyce C#. Nakonec je aplikace zhodnocena a jsou diskutovány její silné a slabé stránky.

Klíčová slova

rozšířená realita, Microsoft Kinect, Unity3D, C#, zpracování obrazu

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

V posledních desetiletích se díky masivnímu rozvoji výpočetních kapacit počítačů rozvíjely také jejich vykreslovací schopnosti a možnosti zpracovávat větší a větší objemy dat v kratším čase. Postupně se možnosti vykreslování posouvaly přes vykreslování dvourozměrné grafiky ke grafice trojrozměrné, kterou dnes již dovedli k vysoké realističnosti. Dalším logickým krokem bylo posunout hranice z vykreslování trojrozměrné grafiky na dvourozměrné obrazovky, na projekci virtuálních scén do reálného prostoru.

Technologie dnešní doby umožňuje vytvoření ať už věrné, nebo jen abstraktní kopie našeho okolí a přispívá tak různými způsoby ke zpracování, či rozšíření světa okolo nás. Principy virtuální, či rozšířené reality se využívají v mnoha odvětvích, zábavním průmyslem počínaje a výrobním průmyslem konče. Většina lidí při otázce, co je rozšířená realita, stále tápe, ale není to způsobeno tím, že by si neuměli představit, o co se vlastně jedná, ale je to spíše tím, že virtuální a rozšířená realita se stala tak běžnou součástí okolního světa, že ji již ani nevnímáme jako něco speciálního.

Virtuální realita má bezesporu své největší využití v herním průmyslu, kde každou chvíli sledujeme nové a nové úspěchy a překonávání technologických mezí, jako jsou např. *HTC Vive*, *Oculus Rift*, *Kinect* apod., ale má dnes již své pevné místo v kapse většiny lidí bezpečně uschovaná v jejich mobilním telefonu. Pokaždé, když využíváme auto navigaci, nebo když si zobrazíme kavárny v našem blízkém okolí, tak náš telefon využívá kromě dalších specifických mechanik, jako je například lokalizace pozice principů, které jsou závislé na principech virtuální rekonstrukce okolního světa z různorodých zdrojů vstupních dat.

2 Cíl práce

Hlavním cíle je navrhnout takovou aplikaci, která bude schopna načítat data z nástroje Kinect a tato data následně transformuje do takové podoby, aby na jejich základě mohla být co nejvěrněji virtuálně zrekonstruována referenční scéna. Nad těmito daty se bude odehrávat referenční hra, jejíž objekty budou pomocí dataprojektoru promítána na stejné prostory, z nichž byla scéna původně zrekonstruována. V zájmu dosažení tohoto cíle bude nezbytné splnit následující dílčí cíle:

- Prozkoumat technické možnosti nástroje Microsoft Kinect a stanovit nejvhodnější způsob jejich využití
- Srovnat využitelné knihovny a zvolit, která bude nejvhodnější pro realizaci
- Provést rešerši stávajících řešení a prozkoumat technologie, které tato řešení využívají
- Implementovat řešení, které bude schopno opravit a filtrovat surová vstupní data Kinectu a tato data následně pomocí projekce vizualizovat na vhodné referenční prostory
- Implementovat vzorovou aplikaci, pomocí které bude prověřena kvalita a robustnost implementace
- U testovacího vzorku uživatelů senzoricky prověřit kvalitu vizualizovaných filtrovaných dat a celkovou kvalitu aplikace
- Zhodnotit a diskutovat řešenou problematiku jak z ohledu technického, tak ekonomického

3 Realizace rozšířené reality v kombinaci s nástrojem Kinect

Pojem virtuální realita se stal v posledních několika desetiletích velice populární a to nejen pro vědeckou obec, ale díky vědecko-fantastické kinematografii se již posledních několik dekád dostává do povědomí i běžných lidí. Pojmem virtuální realita je nejvýstižněji popsán větou *„zcela umělý, počítačově generovaný svět, kde smysly jako zrak, sluch, čich atd. jsou řízeny počítačem"* (Bimber, et al., 2005).

Rozšířená realita je s realitou virtuální velice úzce spjata. Někteří definují rozšířenou realitu jako specifický druh virtuální reality (P. Milgram, et al., 1994), jiní tvrdí, že rozšířená realita je mnohem obecnější pojem než virtuální realita a vidí virtuální realitu spíše jako zvláštní případ rozšířené reality (P. Milgram, et al., 1994). Ať už je jakákoli definice pravdivá, tak však stále platí fakt, že virtuální a rozšířená realita jsou sice pojmy, které sdílejí mnoho společného, avšak se jedná o dvě principiálně naprosto rozdílné věci. Rozdílem mezi virtuální a rozšířenou realitou je především ten, že rozšířená realita pracuje nad reálnými daty a ne nad daty čistě virtuálními, a tedy jakýmsi způsobem rozšiřuje již stávající realitu. Z tohoto důvodu se mnozí vývojáři rozšířené reality obávají, protože je náročnější, než realita virtuální, protože data jsou načítána z reálných prostor a není možné si je libovolně upravovat a usnadňovat si tak různé implementačně složitější situace.

Na Obr. 1 je možné vidět příklad rozšířené reality a její věrohodnosti, kde jsou reálné prostory, jenž čítají samotnou místnost virtuálně obohacenu o postavičku robota.

Obr. 1 Ukázka rozšířené reality
Zdroj: Bimber, et al., 2005

3.1 Kinect

Kinect je zařízení pro snímání vnějšího prostoru vyvinuté společností Microsoft. Poprvé byl představen na festivalu E3 (*Electronic Entertainment Expo*) v červnu roku 2009, tehdy ještě pod pracovním názvem *Project Natural* (Beaumont, 2009). Hlavním záměrem, proč byl Kinect vyvinut, bylo vytvořit nové vstupní zařízení, které by bylo schopné nahradit herní ovladače herních konzolí *Xbox 360* a *Xbox One*. Snahou Microsoftu bylo vytvořit takové zařízení, díky kterému bude uživatel schopen ovládat jejich konzole pomocí zvuků a gest, bez nutnosti dotykově ovládat další vstupní zařízení, čímž by jejich konzole získaly značnou konkurenční výhodu na hráčském trhu. První generace tohoto zařízení byla uvedena v listopadu roku 2010 taktéž na konferenci E3, kde proběhlo předem oznámené představení pod názvem *World premiere ‚Project Natal' for Xbox 360 experience* (Microsoft, 2010), kde byl také odhalen dodnes využívaný název Kinect. Slovo Kinect je spojení dvou anglických slov *kinetic* (pohybový) a *connection* (propojení) (Toulouse, 2010). Tato první generace Kinectu byla tehdy ještě kompatibilní pouze s herními konzolemi Microsoft. Technologie poté čekala dlouhých čtrnáct měsíců až do února 2012 (Eisler, 2012), kdy byla oficiálně vydána i pro PC, což značně urychlilo vývoj technologií založených na tomto zařízení a teprve umožnilo ukázat všechny možnosti, které Kinect uživatelům poskytuje.

V roce 2011 Microsoft vydal první SDK (*Software Development Kit*) (Orland, 2011), které bylo pro nekomerční účely volně dostupné široké veřejnosti. Vydání SDK, spolu s poměrně nízkou cenou Kinectu, oproti jiným zařízením schopným snímat prostor, vedlo k tomu, že se Kinect velmi rychle stal jednou z nejvyužívanějších platforem pro nejrůznější experimenty nad prostorovými daty. To přispělo k tomu, že po prvních 60 dnech se prodalo více než 8 milionů kusů, což Kinectu v únoru roku 2011 zajistilo příčku v Guinnessově knize rekordů za nejrychleji se prodávající spotřební elektroniku (Orland, 2011).

Senzor Kinectu je vybaven čtyřmi základními komponentami. Jedná se o pole mikrofonů a RGB, hloubkový a infračervený senzor.

V dnešní době jsou na trhu dvě odlišné verze zařízení Kinect. Hlavní rozdíly mezi těmito verzemi jsou především v rozlišení, které jsou senzory Kinectu schopny zpracovávat, v počtu osob, které může zařízení simultánně zpracovávat, kvalitě tohoto zpracování a ve vzdálenostních omezeních senzorů. Tyto rozdíly jsou blíže popsány v následující tabulce.

	Kinect 1.0	Kinect 2.0
RGB kamera (pixely)	1280 × 1024, 640 × 480	1920 × 1080
Hloubková kamera (pixely)	640 × 480	512 × 424
Maximální vzdálenost (metry)	4,0	4,5
Minimální vzdálenost (metry)	0,8	0,5
Horizontální zorné pole (stupně)	57	70
Vertikální zorné pole (stupně)	43	60
Polohovací motorek	Ano	Ne
Počet kloubů na kostře	20	26
Maximální počet v jeden okamžik snímaných koster	2	6
USB	2.0	3.0

Tab. 1 Srovnání Kinectu v1 a v2
(Pagliari, et al., 2015)

RBG kamera zachycuje dvourozměrný obraz snímané scény. Informace o každém bodu je načtena v podobě tří hodnot, kde každá hodnota reprezentuje složku daného barevného kanálu (R – červená, G – zelená, B – modrá). Tato data jsou primárně určena k detekci postav ve scéně nebo k detekci detailů na postavě, kde nejvýznamnějším z nich je snímání obličeje postavy.

Pro získání hloubkových souřadnic scény využívá Kinect dvou senzorů a to infračerveného a monochromatického CMOS (*Complimentary metal-oxide semiconductor*) senzoru (Crawford, 2010). Pro detekci objektů využívá Kinect principu, kdy ještě před sejmutím hloubkového snímku pokryje infračervený senzor celé zorné pole infračervenou mapou souřadnic, neviditelnou lidským okem, jejíž údaje se od povrchu odrážejí zpět k senzorům Kinectu.

Obr. 2 Ukázka infračerveného mapování pomocí Kinectu v1
Zdroj: Flatley, 2010

Odraz těchto infračervených paprsků je vyhodnocován monochromatickým CMOS senzorem, kde probíhá zformování hloubkové mapy. Celý tento proces pracuje na principu *time-of-flight*, kde se diferencuje vzdálenost bodu v prostoru. Tento systém měří fázový posun modulovaného signálu a počítá tak hloubku za použití následujícího vzorce, kde proměnná *d* udává hloubku, *c* rychlost světla a f_{mod} je modulační frekvence.

$$2d = \frac{phase}{2\pi} \cdot \frac{c}{f_{mod}}$$

Obr. 3 Vzorec pro výpočet hloubky
Zdroj: Pagliari, et al., 2015

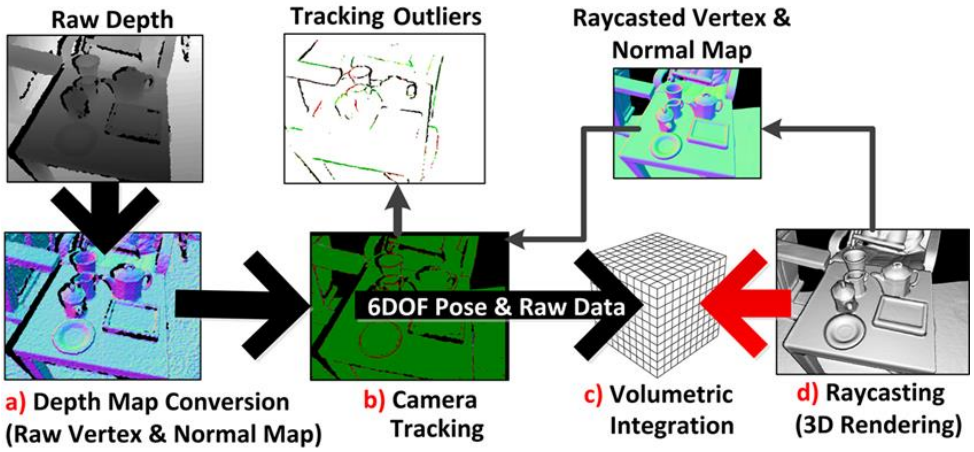
Kinect získává data na několika frekvencích, čímž je značně zpřesněno měření hloubky. Frekvence, které využívá senzor Kinectu v2, jsou přibližně 120 MHz, 80 MHz a 16 MHz (Pagliari, et al., 2015). Princip spoléhá na měření rozdílu mezi dvěma frekvencemi, kde každá z nich obsahuje část vracejícího se infračerveného světla. Platí tedy, že i za předpokladu, že scéna je neadekvátně osvětlena, tak je vždy zaručena adekvátní kvalita načtení scény. Pod účinky přímého slunečního záření je tento princip značně narušen, protože je nemožné korektně diferencovat množství odraženého infračerveného světla (Ruben, 2015). Získaná mapa zobrazuje každý bod v prostoru pomocí tří souřadnic, kde každá z nich reprezentuje jednu osu, přičemž na ose *x* a *y* je zanesena informace o pozici bodu na dvourozměrné hloubkové mapě a na ose *z* je zaznamenán údaj o hloubce v daném bodě.

V dnešní době je možné nalézt řadu projektů zabývajících se virtuální nebo rozšířenou realitou, jejichž základním stavebním kamenem je právě zařízení Kinect. Některé z těchto projektů v mnoha ohledech převyšují ty ostatní a výsledky, které dokazují, jsou mnohdy převratné.

3.1.1 Kinect Fusion

Kinect Fusion je nástroj vytvořený společností Microsoft a je určen ke skenování 3D modelů za pomoci zařízení Kinect. Uživateli je umožněno postupně načítat celou scénu pohybem senzoru Kinectu okolo skenovaného objektu a tím získat plnohodnotný trojrozměrný model. Nevýhodou je hardwarová náročnost nástroje. Pro využívání Kinect Fusion musí uživatel vlastnit grafickou kartu podporující DirectX 11, aby bylo možné zároveň data načítat, zpracovávat a vykreslovat.

Knihovny pracují na principu, kdy rekonstruují model pomocí hloubkové mapy sejmuté z více úhlů pohledu. Knihovny diferencují lokální pozici a rotaci na základě snímku sejmutého z kamery. Data z těchto snímků jsou následně zprůměrována a vzájemně slícována do jedné voxelové mapy. Průběh tohoto procesu je vizualizován na Obr. 4 (Microsoft, 2015).



Obr. 4 Ukázka principu zpracování dat pomocí Kinect Fusion
Zdroj: Microsoft, 2016

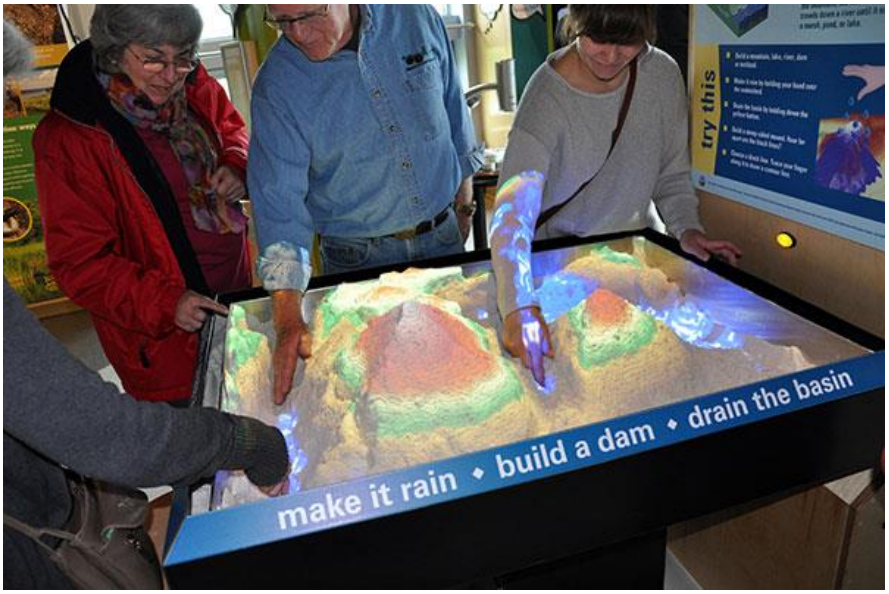
3.1.2 Sandbox

Sandbox využívá dataprojektoru a vstupního zařízení, které je schopné načítat prostorová data, ukotveného na konstrukci nad speciální nádrží s pískem. Dataprojektor i hloubkový senzor jsou zkalibrovány tak, aby si jejich zorná pole navzájem odpovídala.

Uživateli je poté umožněno, aby z písku modeloval různorodé tvary, přičemž hloubkový senzor detekuje vzdálenost k takto vzniklým útvarům. Na základě těchto dat je zrekonstruován trojrozměrný model, který je obarven v závislosti na pomyslné nadmořské výšce. Výsledné barvy jsou pomocí dataprojektoru promítány na písek, čímž vzniká iluze topografické mapy. Uživateli je potom umožněno tuto mapu v reálném času upravovat tím, že modifikuje reliéf písečného terénu.

V případě, že senzory detekují objekt (například ruku) ve specifické výšce nad povrchem, tak je z tohoto objektu vyvolán virtuální déšť, který se projevuje jako modrá dynamicky se měnící textura na písku. Tato textura se adaptuje na základě povrchu a postupně se rozpíná po takových plochách, aby bylo rozpínání vodní

plochy přirozené a odpovídalo běžnému chování vodního toku reálného světa. Virtuální voda poté pomalu mizí jakožto simulace vsakování do půdy.
(Reed, et al., 2015)



Obr. 5 Ukázka sandboxu
Zdroj: Reed, et al., 2015

3.1.3 A realistic Augmented Reality Racing Game using a Depth-Sensing Camera

Tento projekt se zabývá vylepšením standardních principů rozšířené reality a korektním pozicováním virtuálních objektů spolu s objekty reálnými. Většina projektů rozšířené reality se zabývá pouze umístěním virtuálních objektů, ale již dále neřeší, zda se tyto objekty chovají realisticky v závislosti na prostředí, v němž jsou umístěny. Výsledkem pouhého vykreslování virtuálního objektu na markeru v reálné scéně je často chybné zobrazení těchto objektů a virtuální objekty se mnohdy jeví, jako by se vznášely nad povrchem, nebo kolidovaly s jinými objekty reálné scény. Tento projekt se zabývá řešením těchto situací a za pomoci zařízení Kinect zkoumá prostory, do nichž mají být zasazena virtuální data. Nad těmito daty provádí různorodé korekce výsledné pozice, a virtuální objekty následně umisťuje na korektní pozice s ohledem na vlastnosti referenční scény.
(Clark, et al., 2011)

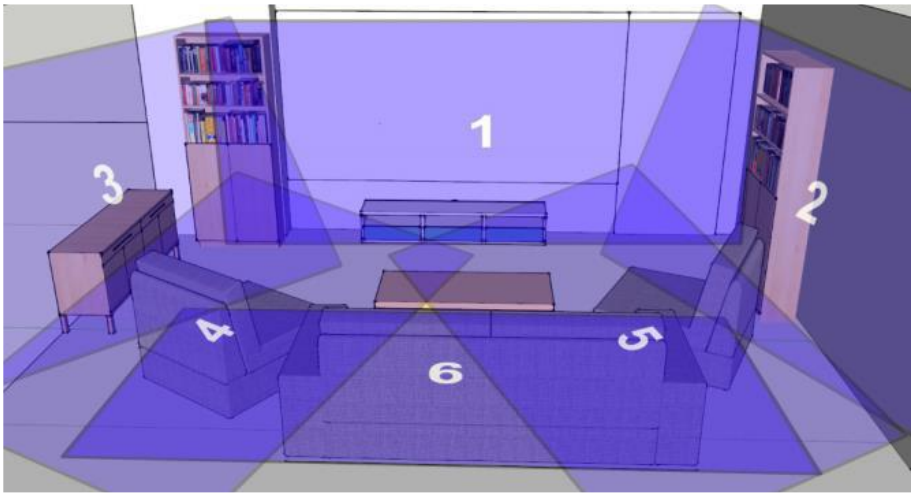
Obr. 6 Ukázka korektní pozice virtuálních objektů
Zdroj: Clark, et al., 2011

3.1.4 RoomAlive

RoomAlive je bezesporu jeden z technologicky nejsofistikovanějších projektů rozšířené reality, který je možné dnes najít. Jedná se o projekt, realizovaný vývojovým týmem Microsoftu. RoomAlive je *proof-of-concept* prototyp, který je zkonstruován tak, aby byl schopen proměnit jakoukoli místnost na virtuální hřiště, na němž jsou realizovány různorodé projekce.

Projekt silně vyčnívá především v technologii kalibrace a dynamického prostorového mapování. RoomAlive umožňuje značnou míru interakce uživatele s prostředím. Hráč se může pohybovat po prostředí, dotýkat se věcí, zvedat objekty apod.

RoomAlive pracuje na principu kombinace několika hloubkových senzorů s dataprojektory, které mají předem sjednocené *field-of-view* (zorné pole) s těmito senzory, kde všechna tato zařízení jsou propojena s jedním počítačem. Vzorový projekt, na kterém Microsoft demonstruje tuto technologii, se odehrává v místnosti o rozměrech 18 stop na 12 stop, kde projekce pokrývá tři vztyčné plochy a podlahu za pomoci šesti projektorů a k nim připevněných hloubkových senzorů, jak je možné vidět na Obr. 7.



Obr. 7 Konstrukce projektu RoomAlive
Zdroj: Jones, et al., 2014

Celý projekt je realizován pomocí herního enginu Unity3D, který je využit pro vykreslování projekce, simulaci virtuální fyziky objektů a perspektivní korekce s ohledem na pozici pozorovatele ve scéně.

Projekt vyčnívá především díky skvěle zvládnutému principu kalibrace a řešení problematiky vzájemně se překrývající projekce více projektorů. Díky tomu, že projektory jsou upevněny na pevné konstrukci, tak tvůrci přesně znají pozice ostatních zařízení a jsou tedy schopni za velice vysoké přesnosti řešit slícování hloubkových map, které jim poskytují hloubkové senzory. Tento princip umožňuje sestavit naprosto věrnou virtuální podobu referenční scény ve virtuálním prostoru a za pomoci Unity s ní libovolně manipulovat. Dále určitě stojí za zmínku princip, kterým vývojáři komunikují mezi jednotlivými Kinecty a jejich pomocí načítají kostry uživatelů scény, čímž řeší problém vzájemného si stínění, které je v případě využití jen jednoho Kinectu kamenem úrazu u spousty projektů.

Výsledkem je tedy velice precizní projekce rozšířené reality, která umožnuje interakci několika uživatelům najednou a jedná se momentálně asi o nejlépe zpracovaný volně dostupný projekt rozšířené reality této doby.

Obr. 8 Ukázka projektu RoomAlive
Zdroj: Jones, et al., 2014

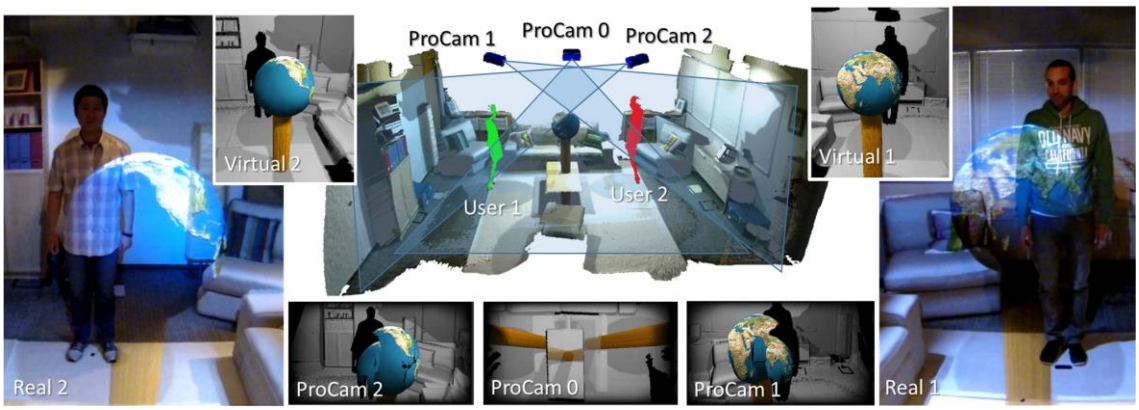
3.1.5 Dyadic Projected Spatial Augmented Reality

Jedná se o projekt zaštítěný společností Microsoft, jehož hlavním účelem je vykreslování obrazu do prostoru s ohledem na pozici pozorovatele.

Tento projekt využívá dataprojektoru, ke kterému je připevněno zařízení Kinect pro načítání prostorových dat, na jejímž základě je následně vytvořena virtuální reprezentace zkoumané scény. Kinect je dále využíván ke sledování pozice a rotace hlavy uživatele a s ohledem na takto získaná data je prováděna perspektivní korekce dat nad virtuálním modelem načtené scény.

Projekt může být využíván buď jedním, nebo i dvěma uživateli najednou. Proto, aby bylo možné současné, avšak zcela rozdílné projekce pro každého uživatele zvlášť, je nezbytné, aby každý uživatel stál sám pod protějším dataprojektorem. Zorné pole uživatele je poté promítáno jak na statické objekty ve scéně, tak i na tělo druhého uživatele, jak je možné vidět na Obr. 9. Uživatelům je umožněna i vzájemná interakce s virtuálními objekty ve scéně. Tento mechanismus je v tomto projektu demonstrován například tím, že si uživatelé mohou mezi sebou házet virtuální tenisové míčky, nebo jiné předměty. (Benko, et al., 2014).

Tento projekt všeobecně potvrzuje, že je možné vykreslovat odlišný obraz pro různé uživatele a přitom zachovat integritu virtuálních dat, čímž jednoznačně posunuje možnosti využitelnosti prostorové rozšířené reality na další úroveň.



Obr. 9 Ukázka dynamické projekce s prostorovou korekcí
Zdroj: Benko, et al., 2014

3.2 Unity

Unity3D je multiplatformní engine primárně určený k vývoji her vyvinutý společností *Unity Technologies*, je však díky své multiplatformnosti využíván k implementaci různorodých aplikací. Jádro unity je napsáno v jazyce *C++*, ale interní skripty aplikace musejí být napsány v jazyce *C#*, *JavaScript*, nebo *Boo*. Na Unity je také možné snadno připojit vesměs jakékoli již zkompilované *C++* skripty uložené v podobě dll. Unity je postaveno tak, aby bylo kompatibilní se všemi hlavními grafickými API, jako jsou *OpenGL*, *OpenGL ES*, nebo *Direct3D*, kde výběr grafického API závisí pouze na volbě uživatele.

Unity je pro nekomerční účely poskytováno zcela zdarma pouze v mírně omezené verzi. Tato omezení se však týkají vesměs pouze designových nebo analytických funkcí, které většina uživatelů nepotřebuje a pokud ano, tak je možné tyto knihovny nahradit knihovnami třetích stran s podobnou funkcionalitou. Díky tomu se Unity stalo jedním z nejpoužívanějších enginů pro vývoj her. Tento fakt je dokumentován tím, že dle statistik mají různé produkty vyrobené na enginu Unity3D přes 5 bilionů stažení, pracuje na něm 34 procent z nejlepších tisíce zdarma poskytovaných mobilních her a s hrami na Unity se již setkalo přes 770 milionů hráčů (Unity Technologies, 2016).

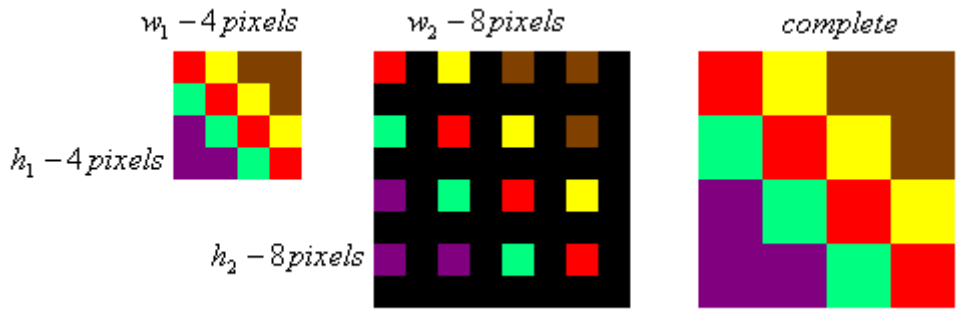
Unity se jakožto nástroj primárně určený k vývoji her řídí vlastními technikami vykreslování dat. Sítě polygonů, které je možné pomocí Unity renderovat jsou omezeny počtem 65 000 vrcholů pro jeden model. Tento fakt komplikuje situace, kdy uživatel chce vykreslovat data na základě výškových map, nebo jiných struktur s vysokou hustotou dat. Takováto data si uživatel tedy musí sám nějakým způsobem upravit, aby maximální počet údajů, které mají být převedeny na vrcholy, odpovídal maximálnímu možnému zpracovatelnému počtu vrcholů.

3.2.1 Algoritmy pro změnu velikosti bitmap

V případě, že je záměrem rekonstruovat sítě polygonů za pomoci Unity na základě výškové mapy v podobě bitmapy, je nezbytné zajistit, aby počet polygonů této bitmapy byl menší než maximální možný počet vrcholů pro rekonstrukci pomocí Unity. Za tímto účelem je nezbytné zvolit vhodný algoritmus pro změnu velikosti bitmap.

Algoritmy pro změnu velikosti bitmap jsou velmi diskutovaným a frekventovaným tématem. Existují desítky různých algoritmů, jejichž pomocí se dá dosáhnout různě kvalitních výsledků. Využitelnost těchto algoritmů je rozlišná a záleží na mnoha faktorech, jako maximální přípustná časová složitost, typ zpracovávaných dat, požadovaná úroveň změny velikosti apod.

Změna bitmapy je proces, kdy je na základě referenční vstupní bitmapy a požadovaného výstupního měřítka zkonstruována její zvětšená, nebo zmenšená verze. Při zvětšování obrázků jsou vytvořena prázdná místa ve zvětšené verzi původního obrázku. Tato prázdná místa jsou následně nahrazena interpolovanými hodnotami. Na Obr. 10 je vizualizován celý tento proces, přičemž algoritmem pro interpolaci je interpolace nejbližším sousedem (John, 2007).

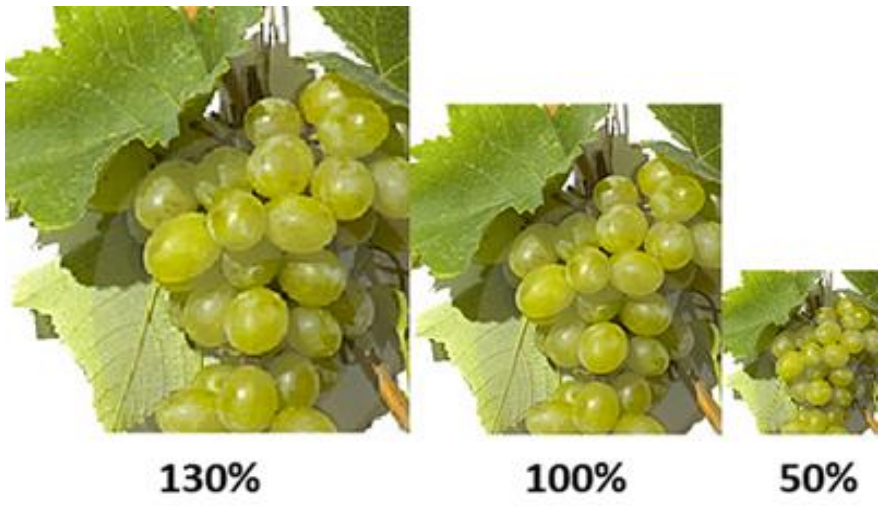


Obr. 10 Ukázka principu pro změnu velikosti obrázku
Zdroj: John, 2007

3.2.2 Interpolace nejbližším sousedem

Interpolace nejbližším sousedem je jednou z nejjednodušších a nejrychlejších druhů interpolace a využívá se především v těch případech, kde je rozhodující rychlost před kvalitou (např. generování miniatur ve fotoalbech). Algoritmus pracuje na principu, kdy zkoumá okolí body zkoumaného prvku a přiřadí mu takovou hodnotu, kterou nabývá nejbližší prvek.

Při zmenšení vstupních dat často vzniká okem viditelné zostření hran obrázku, protože neprobíhá žádná interpolace pixelů, o které je výsledná bitmapa ošizena, a hodnoty těchto pixelů jsou jednoduše anulovány.



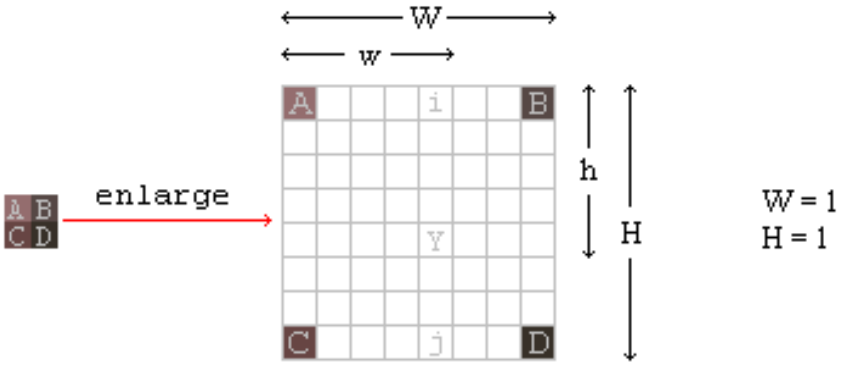
Obr. 11 Ukázka změny velikosti pomocí interpolace nejbližším sousedem
Zdroj: John, 2007, upraveno autorem práce

3.2.3 Bilineární interpolace

Bilineární interpolace funguje podobně jako interpolace nejbližším sousedem na principu srovnání s přilehlými pixely, ale nepřejímá jejich hodnotu přímo, nýbrž ji

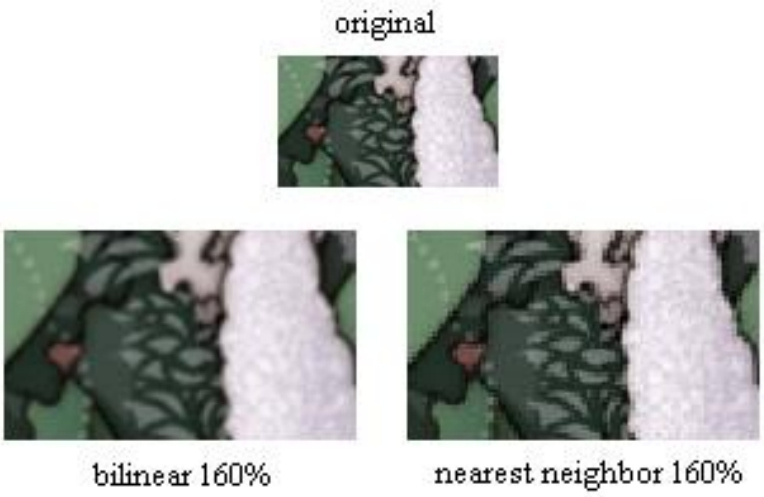
lineárně interpoluje. Výsledná hodnota se vypočte za pomoci kombinace dvou lineárních interpolací a to interpolací po ose *x* a *y*.

Při zmenšení obrazu nejsou hodnoty pixelů jednoduše anulovány, ale jejich hodnota je zanesena do hodnoty výsledného pixelu, čímž je docíleno hladkého přechodu při zmenšení. Naopak při zvětšení obrázku jsou hodnoty nově vzniknutých pixelů interpolovány mezi dvěma nejbližšími pixely tak, jak je ilustrováno na Obr. 12.



Obr. 12 Princip zvětšení pomocí bilineární interpolace
Zdroj: John, 2007

Bilineární interpolace je rychlá a poskytuje velmi kvalitní výsledky. Největším problémem při využití tohoto algoritmu je především optická ztráta ostrosti a hloubky zpracovávané bitmapy.

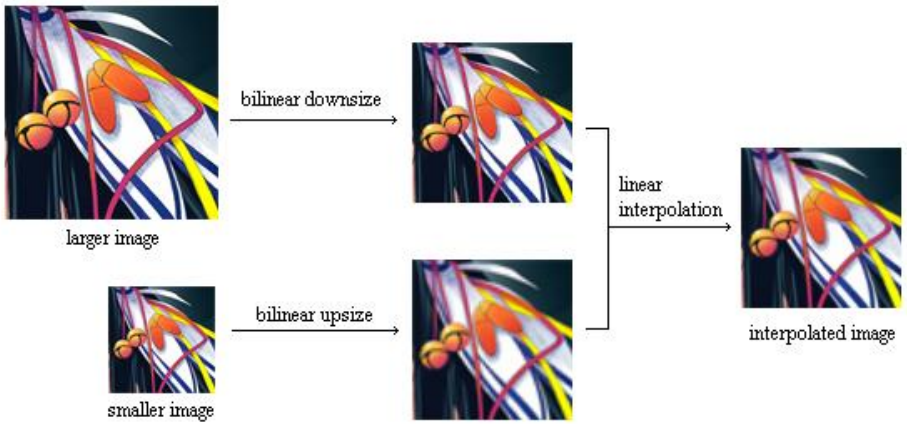


Obr. 13 Srovnání změny velikosti bilineárního zvětšení a zvětšení pomocí nejbližšího souseda
Zdroj: John, 2007

3.2.4 Trilineární interpolace

Trilineární interpolace je nadstavba nad bilineární interpolací, jejíž nejčastější využití je pro tvorbu *mipmap* (předem generovaná pyramidová sekvence obrázku v různých velikostech). Tato metoda vyžaduje pro svou funkci dvě verze vstupních dat o různých velikostech, přičemž platí, že druhá verze vstupních dat by měla mít poloviční velikost oproti první.

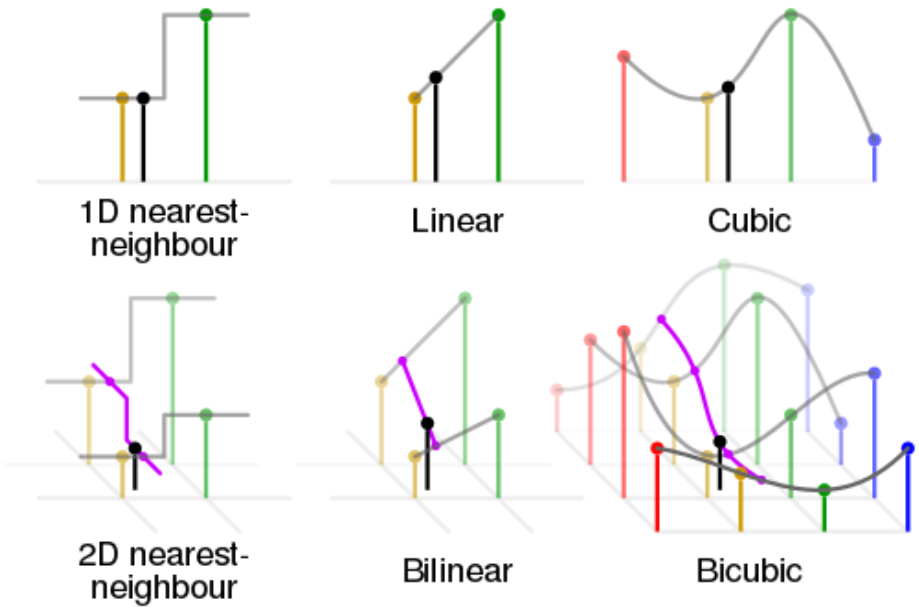
Princip trilineární interpolace je takový, že nejdříve je pomocí bilineární interpolace vytvořena zmenšená verze většího obrázku a poté je taktéž za pomoci bilineární interpolace vytvořena zvětšená verze menšího obrázku. Oba takto nově vzniklé obrázky jsou mezi sebou následně podrobeny lineární interpolaci. Výstupem této kombinace obou vytvořených bitmap je již finální výstupní bitmapa.



Obr. 14 Princip trilineární interpolace
Zdroj: John, 2009

3.2.5 Bikubická interpolace

Bikubická interpolace je z vybraných popsaných metod bezesporu nejkvalitnější metodou pro změnu velikosti obrázku, leč její kvalita je vykoupena její časovou složitostí. V porovnání s bilineární interpolací, která v jeden okamžik zpracovává pouze 4 pixely (2×2), tak bikubická interpolace zpracovává 16 pixelů (4×4). Získání hodnoty daného pixelu není vyhodnocován pomocí lineární interpolace, ale může být realizována různými algoritmy, díky kterým je pro stejná data možné dosáhnout rozdílných výsledků. Příkladem může být například *Lagrangeova interpolace*, *Bikubická konvoluce*, nebo algoritmus *Cubic spline* (kubická křivka) (Keys, 1981).

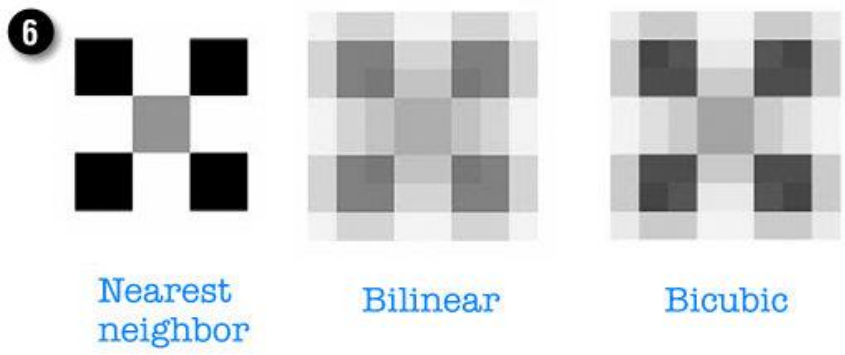


Obr. 15 Vizualizace principu bikubické interpolace
Zdroj: Cmglee, 2007

3.2.6 Shrnutí

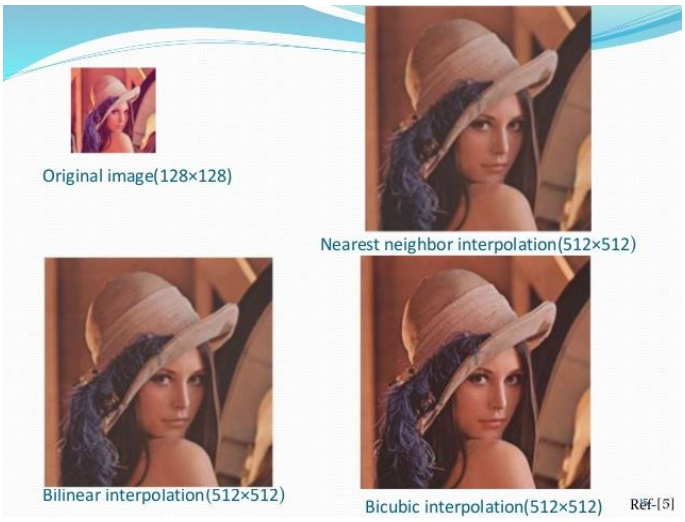
Uvedené algoritmy patří mezi nejznámější a nepoužívanější algoritmy pro změnu velikosti obrázků. Každý z těchto algoritmů poskytuje výstupy rozdílné kvality a rozdílné časové složitosti.

Výkonnostně nejsložitějším algoritmem je algoritmus pro bikubickou interpolaci, ale za to by měl být schopen poskytovat nejlepší možné výsledky. Dalším algoritmem, který poskytuje velice kvalitní výstupy, je algoritmus pro trilineární interpolaci, který je bohužel ale technologicky omezen tím, že potřebuje více vstupních dat, které ve většině případů nejsou v adekvátní kvalitě na začátku k dispozici. Z tohoto důvodu je často při využívání trilineární interpolace například pro tvorbu mipmap v Unity využit alternativní přístup, kdy je nejdříve ze vstupních dat pomocí bikubické interpolace vytvořena zmenšená bitmapa a další snímky jsou již dále vytvářeny trilineárním algoritmem. Nejpoužívanější metodou je nejspíše bilineární interpolace, protože poskytuje nejspíše nejlepší poměr mezi výkonem a výsledným efektem, ale je nutné mít na paměti, že toto tvrzení nemusí platit ve všech případech. Nejméně kvalitní, ale bezesporu nejrychlejší je metoda pomocí nejbližšího souseda, která však pro mnohá data vytváří příliš hrubé přechody. V některých případech však právě tato metoda může poskytovat nejlepší výsledky, jak můžeme vidět na Obr. 16.



Obr. 16 Srovnání algoritmů pro změnu velikostí bitmap 1
Zdroj: Bouton, 2003

Obr. 16 demonstruje data, kde je na první pohled vidět rozdíl mezi jednotlivými algoritmy. Je však nutné podotknout, že se jedná o velice specifickou situaci, která v praxi nastává jen velice zřídka. Ve většině případů jsou zpracovávána data z reálného světa, u kterých už rozdíl mezi algoritmy není tak zřetelný jako v předcházejícím případě. Příkladem takovýchto dat může být bitmapa, která je uvedena na Obr. 17.



Obr. 17 Srovnání algoritmů pro změnu velikostí bitmap 2
Zdroj: Alokahuti, 2015

Pro výběr nejvhodnějšího algoritmu není žádná definovaná metodika pro výběr nejoptimálnější metody pro změnu velikosti, protože jak je zřejmé z předchozích ukázek, tak není možné předem predikovat, který algoritmus bude poskytovat nejlepší výsledek. Je možné nalézt taková data, pro která bude bikubická interpolace zcela nevhodná a naopak bilineární bude poskytovat velice uspokojivé výsledky apod.

4 Srovnání knihoven pro Microsoft Kinect

Poslední kapitola rešeršní části práce se zabývá knihovnami schopnými komunikovat s nástrojem Kinect. V závěru této kapitoly je srovnání vybraných knihoven a zhodnocení jejich využitelnosti.

4.1 OpenKinect Project

OpenKinect je otevřená komunita asi 2 000 lidí, kteří se zajímají o práci se zařízením Kinect a o jeho integraci s počítačem. Skupina pracuje na zdarma poskytovaných knihovnách s názvem *libfreenect*, které umožňují interakci Kinectu s platformami *Windows*, *Mac* a *Linux*. Zdrojové soubory jsou poskytovány pod *Apache20*, nebo volitelně pod *GPL2* licencí.

Knihovna je napsána v jazyce C a poskytuje dva druhy API a to *High level* a *Low level*. Knihovna obsahuje *wrappery* pro celou řadu programovacích jazyků, jako jsou *C*, *C++*, *.NET*, *Java*, nebo *Phyton*. Členové komunity také pracují na různých API pro propojení s *OpenCV*, *MATLABEM*, *LabView* atd.

Pro knihovny také existují různé nadstavby. Jedná se především o nadstavbu *Record*, která umožňuje nahrávání získaných dat a jejich ukládání na disk, nebo *Fakenect*, což je simulátor rozhraní Kinectu, který umožňuje uživateli přehrávat uložená data a díky tomu pracovat, i když zrovna není Kinect připojen (Open Kinect, 2012).

Knihovny disponují dosti krátkou a zmatečnou dokumentací a uživatel je nucen informace hledat spíše u externích zdrojů. Projekt se kdysi slavil ve velké komunitě a na své oficiální Wiki poskytuje odkazy na různá fóra, IRC a další komunikační kanály, na nichž však již k datu zpracovávání této práce nebyla viditelná žádná větší aktivita. Stejně tak to vypadá se zdrojovými kódy v oficiálním repositáři na serveru *GitHub*, kde je v posledních letech vidět velice nízká až žádná aktivita.

Obr. 18 Logo OpenKinect
Zdroj: Open Kinect, 2012

4.2 OpenNI

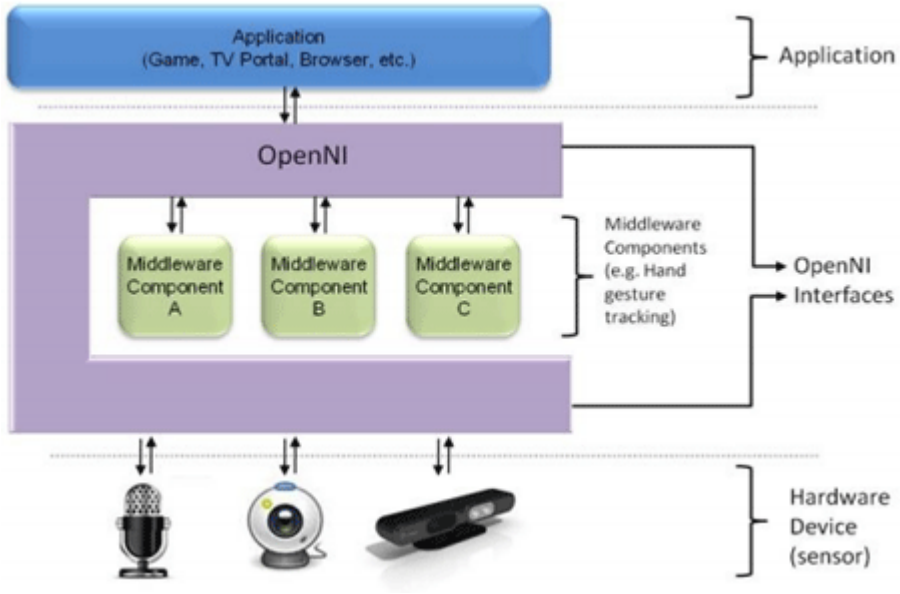
Open Natural Interaction (zkráceně jen OpenNI) je open source projekt, který se zabývá rozvojem technologií na poli zařízení umožňujících interakci pomocí přirozeného uživatelského rozhraní. V počátcích byl tento projekt pod křídly společnosti *PrimeSence*, ale poté byl odkoupen společností *Apple* a v roce 2014 byly zrušeny jeho tehdy oficiální webové stránky OpenNI.org (Armstrong, 2014).

OpenNI poskytuje open source API, které je možné ovládat pomocí jazyků C++ a C# umožňuje:

- Rozpoznávání zvukových příkazů
- Gesta rukou
- Sledování pohybu těla

Knihovny jsou k dostání ve dvou verzích, přičemž pouze knihovny první verze jsou schopny komunikovat se zařízením Kinect. Knihovny druhé generace jsou určeny pro společnost ASUS a pro jejich Wavi Xtion.

Knihovny umějí komunikovat s celou řadou zařízení a nejsou tedy zcela jednoúčelové jako ostatní knihovny, kterými se tato práce zabývá. Knihovny fungují na tzv. principu *Middleware*, což je princip pro implementaci komunikace mezi vstupy a výstupy, jak ukazuje Obr. 19.



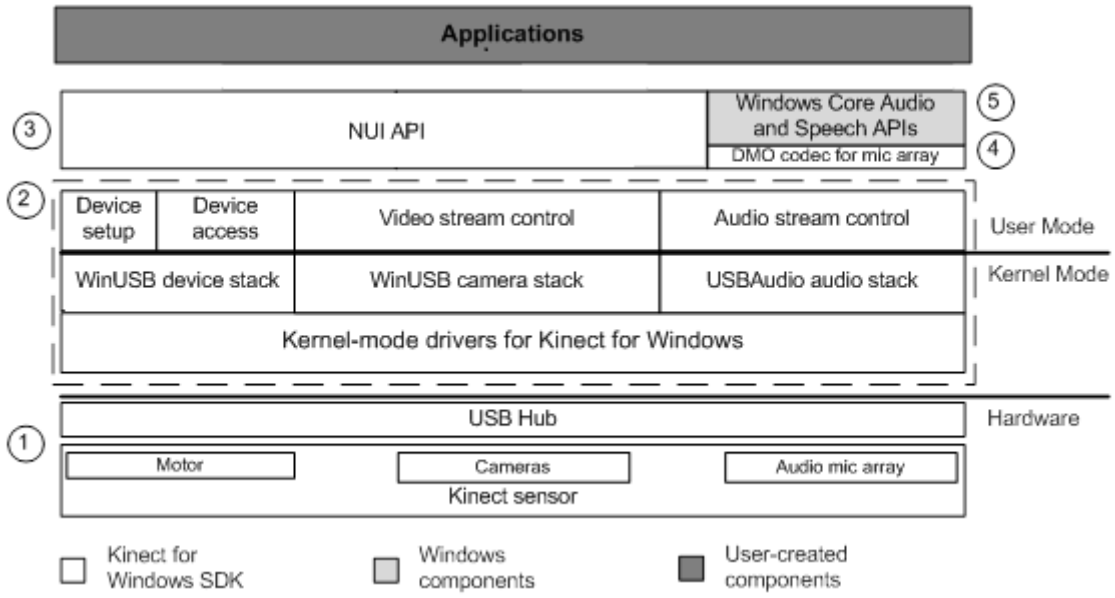
Obr. 19 Princip funkce OpenNI
Zdroj: Loriot, 2011

4.3 Microsoft Kinect SDK

V červnu roku 2011 Microsoft oficiálně vydal knihovnu *Kinect for Windows SDK* pro Kinect verze jedna. Knihovna je volně ke stažení pro nekomerční využití. K dnešnímu datu je knihovna ke stažení na oficiálních stránkách Microsoftu ve verzi *2.0.1410.19000* a byla naposledy aktualizována 21. 10. 2014. Pro využívání této knihovny musí uživatel nainstalovat minimálně *.NET Framwork verze 4*. Knihovna běží pouze na operačních systémech Microsoftu a to konkrétně na systémech *Windows 7*, *Windows 8*, *Windows 8.1*, nebo *Windows Embedded Standard 7*.

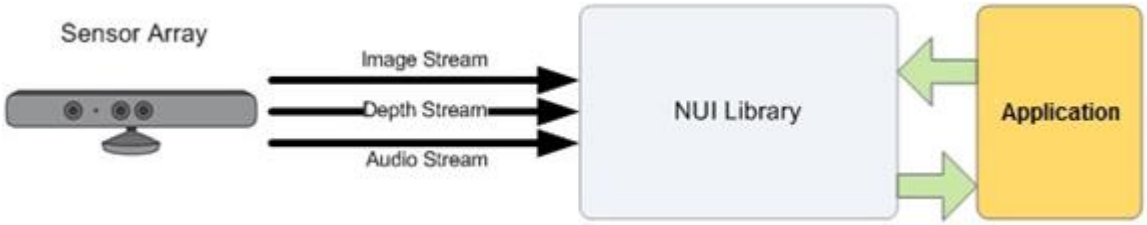
Knihovna je velice kvalitně zdokumentována a má velmi aktivní komunitu. Pro interakci s těmito knihovnami je možné využít jazyk *C#*, *C++*, nebo *Visual Basic* (Microsoft, 2016).

Knihovny komunikují se zařízením Kinect pomocí architektury, která je ilustrována na Obr. 20.



Obr. 20 Architektura Kinect for Windows SDK
Zdroj: Microsoft, 2014

Knihovny umožňují interakci s barevným, infračerveným i hloubkovým senzorem a také se všemi mikrofony zařízení.



Obr. 21 Ukázka interakce s aplikací
Zdroj: Microsoft, 2014

4.4 Srovnání

Všechny uvedené knihovny jsou schopny komunikovat se zařízením Kinect, ale každá z nich má své silné a slabé stránky.

Po stránce multiplatformnosti jednoznačně vedou knihovny OpenNI a OpenKinect, protože je možné tyto knihovny provozovat na všech běžných platformách, kdežto oficiální SDK Microsoftu běží pouze na operačním systému Windows.

Co se týká kompatibility programovacích jazyků je jednoznačně nejlepší OpenKinect, který obsahuje *wrappery* pro nejvíce programovacích jazyků. OpenNI a Microsoft SDK podporují pouze jazyky z rodiny Microsoftu.

Všechny knihovny mají přístup k RGB kameře, avšak s ní dokážou pracovat v různých rozlišeních. Jediná knihovna, která dokáže získávat vstupní RBG snímky v HD rozlišení, je knihovna Microsoftu.

Co se týče zvuků, tak nejlepší podporu má Microsoft SDK. Ostatní knihovny sice umějí pracovat se zvukovým vstupem, ale pouze Microsoft umí pracovat se všemi čtyřmi vstupy.

Po stránce dokumentovanosti jednoznačně vedou knihovny Microsoftu. Zbylé knihovny mají dokumentaci velice chabou a většinu informací musí uživatel hledat na různých fórech, v článcích nebo v diskuzích. Co se týká uživatelské aktivity, tak knihovny OpenNI a OpenKinect se již netěší přílišné aktivitě uživatelů, kdežto na fórech Microsoftu jsou uživatelé velice aktivní a často do diskuzí přispívají i lidé přímo od Microsoftu.

Co se týká složitostí integrace s nástrojem Unity, tak jednoznačně vede Microsoft SDK, který přímo poskytuje speciální zkompilované knihovny, které jsou přímo určeny pro interakci s enginem Unity a obsahují i řadu okomentovaných vzorových aplikací, z nichž se může uživatel mnohému naučit.

	OpenNI	**OpenKinect**	**Microsoft Kinect SDK**
Platformy	Windows, Linux, Mac	Linux, Windows, Mac	Windows
Programovací jazyky	C#, C++	C, C#, Jana, Python	C#, C++
Počet kloubů	20	-	26
Počet koster	-	-	6
RGB kamera	Ano	Ano	Ano
Hloubková kamera	Ano	Ano	Ano
Infračervená kamera	Ano	Ano	Ano
Využití mikrofonu	Ano	Ne	Ano
Rozpoznání řeči	Ano	Ne	Ano
Oficiální dokumentace	Ne	Ne	Ano

Tab. 2 Srovnání knihoven pro Kinect

5 Metodika práce

V zájmu úspěšné realizace vizualizace virtuálních objektů nad reálnými daty budou muset být provedeny následující kroky:

Propojit platformu Unity3D s ovladačem Kinect

Bude nezbytné na základě rešerše dostupných knihoven pro manipulaci s Kinectem nalézt knihovnu umožňující komunikaci mezi platformou Unity3D a ovladačem Kinect. Knihovna musí být ve zkompilované podobě ve formátu dll (Dynamic-Link Libraries), aby bylo možné tuto knihovnu propojit s herním enginem Unity3D. Knihovna musí být schopna pracovat jak s hloubkovým senzorem, tak s barevným senzorem Kinectu a měla by být schopna navzájem mapovat vstupní snímky mezi těmito dvěma vstupy a umožnila tak jejich převoditelnost. Knihovna musí být stabilní, optimalizovaná a patřičně zdokumentovaná. Hlavním parametrem je, aby knihovna umožňovala přístup přímo k hrubým datům načteným přímo ze senzoru, protože v zájmu dosažení maximální kvality filtrace bude nutné data upravit vlastními opravnými algoritmy.

Připravit referenční prostory a kalibrovat dataprojektor

Bude sestrojena referenční scéna, nad kterou bude aplikace testována. Scéna by měla být dostatečně rozmanitá, ale zároveň by neměla obsahovat zbytečně mnoho objektů, aby výsledná ukázková aplikace měla dostatek prostoru pro interakci na volných plochách. Nezbytné bude navrhnout intuitivní rozhraní, než si uživatel bude schopen co nejjednodušeji a nejrychleji kalibrovat dataprojektor vůči čtečce Kinectu, aby bylo dosaženo maximální kvality výsledné projekce a obraz adekvátně lícoval s daty získanými ze senzorů.

Načíst, vyfiltrovat a opravit chyby ve vstupních datech

Bude implementováno rozhraní, které aktivuje Kinect a následně bude s jeho pomocí možné načítat surová hloubková a barevná data ze senzorů. Hrubá vstupní data bude nejprve nutné načíst do vlastních jednoduše přístupných struktur, aby bylo možné s nimi manipulovat. Následně budou implementována opravná opatření nad daty, která Kinect vyhodnotí jako chybná, nebo nespolehlivá. Následně budou data podrobena několika sériím různých validačních a aproximačních algoritmů, které budou mít za úkol co nejpreciznějí eliminovat šum nad vstupními daty, či opravit chybové záznamy. Celý tento proces bude několikrát zopakován nad několika načtenými snímky a tato opravená data budou následně mezi sebou zprůměrována, aby bylo dosaženo co nejspolehlivějšího výsledku a minimalizovala se tak strojová chyba snímaných vstupních dat.

Vizualizovat získaná data

Surová data musejí být převedena do bitmapové podoby, aby mohla být snadno vizualizována a převedena do korektního měřítka. Vizualizace bude nezbytná, aby bylo následně možné senzorickými testy prověřit efektivitu filtrace před zkonstruováním projekční scény.

Navrhnout uživatelské GUI

Musí být navrženo intuitivní GUI (Graphical User Interface), pomocí kterého si uživatel bude moci případně poupravit výslednou normalizovanou výškovou mapu a upravit umístění virtuální kamery ve scéně zrekonstruované pomocí dat z výškové mapy. Skrz toto rozhraní bude mít uživatel také dále možnost ovlivňovat funkce implementované vzorové aplikace.

Navrhnout vzorovou hru pro interakci nad daty

Na základě výstupní výškové mapy bude vytvořena síť polygonů, která bude sloužit jako neviditelný terén, ve které se uživatelé budou moci virtuálně pohybovat. Tato síť bude před uživateli skryta, aby nijak neovlivňovala výslednou projekci, a bude sloužit především k omezení pohybu hráčů po scéně. Hra se tedy bude odehrávat pouze na ploše zorného pole hloubkového senzoru Kinectu. Hra bude navržena pro dva hráče a bude ovládána vstupem z jedné klávesnice. Hráči se budou moci pohybovat po objektech umístěných ve scéně, ale tuto scénu nebudou moci nijak opustit. Cílem hry bude eliminovat soupeřova hráče.

6 Návrh a implementace

6.1.1 Integrace Kinectu

Pro úspěšné získání dat z ovladače Kinect musejí být provedeny dva základní úkony. Prvním z nich je především aktivace senzoru Kinectu a následné zpřístupnění zdrojů, ze kterých má zařízení čerpat data. Aplikace pro práci potřebuje primárně hloubková data, s nimiž po jejich korektuře pracují všechny další algoritmy. Vedle hloubkových dat je také nutné mít načtena data uchovávající informace o barvě. Primární určení barevného snímku je ke kalibraci zorného pole dataprojektoru a zorného pole hlubokého senzoru Kinectu. Tento barevný snímek je následně přemapován na snímek hloubkový a takto upravený výstup je uživateli zobrazován na standardním výstupu. Na základě takto upraveného snímku je uživateli umožněno napozicovat dataprojektor tak, aby výstup dataprojektoru co nejlépe odpovídal vstupu hloubkového senzoru. V konečném důsledku je tedy snímač Kinectu nastaven tak, aby byl schopen načítat data jak o barvě, tak o hloubce.

Implementace aktivace a následného získávání dat z herního ovladače Kinect pomocí nástroje Unity3D je mírně odlišná oproti referenčnímu kódu v dokumentaci poskytované k nástroji Kinect Microsoftem. Standardní kód pro načítání dat z Kinectu uváděný Microsoftem spoléhá na událost `MultiSourceFrameArrived`, která je napojena přímo na čtečku. Kinect interně načítá data ve vlastním vlákně a po jejich načtení provolá výše zmíněnou událost. Tato událost je určena k notifikaci, že data byla načtena a jsou připravena k použití. Platforma Unity pracuje z mnoha důvodů na mírně zjednodušené verzi *.NET 3,5* a využívá kompilátor *Mono*, což v některých případech znemožňuje přímou kompatibilitu Unity s knihovnami třetích stran, přičemž jednou z nich je právě i využitá knihovna *Kinect for Windows SDK 2.0*. Při práci s Unity je nový uživatel omezen především tím, že Unity se řídí vlastními principy zpracování vláken, které se liší od standardních principů, které využívá Microsoft ve svých knihovnách, které poskytuje pro propojení nástroje Kinect s aplikacemi. Poskytované knihovny jsou již ve zkompilované podobě dll (*Dynamic-link library*) knihoven, a není je tedy možné pozměnit.

Z důvodu nekompatibility vícevláknového zpracování v Unity není tedy možné využívat výše zmíněnou klíčovou událost `MultiSourceFrameArrived`. Aplikace tedy využívá vlastní funkcionalitu, která hlídá, zda jsou tato data již inicializována a teprve až poté vyvolá příslušné akce pro práci nad těmito daty. Tímto způsobem je v plném rozsahu suplována výše zmíněná notifikační událost.

Samotná inicializace je provedena pomocí metody `Start`, což je interní metoda Unity. Tato metoda je automaticky vyvolávána vždy před vykreslením prvního snímku aplikace na všech skriptech, které dědí z mateřské třídy `MonoBehaviour`, kterou Unity využívá k detekci svých vlastních skriptů. Na těchto skriptech Unity posléze provolává také různé další události, jako je například metoda `Update`, která je blíže popsána v následujícím textu.

Prvotní inicializace je provolána na třídě `KinectInput,` což je třída, která má za úkol kompletně spravovat všechny vstupy z ovladače Kinect (viz Kód. 1).

```
private void Start()
{
  this.Sensor = KinectSensor.GetDefault();
  if (this.Sensor != null) {
    if (!this.Sensor.IsOpen) {
      this.Sensor.Open();
    }
    this.reader = this.Sensor.OpenMultiSourceFrameReader
    (
      FrameSourceTypes.Color |
      FrameSourceTypes.Depth |
    );
  }
  if (!this.Sensor.IsAvailable) {
    Debug.LogError("Kinect není připojen!");
  }
}
```

Kód. 1 Inicializace Kinectu

Pro získání snímku aplikace využívá metodu `Update`, kterou Unity interně vyvolává před vykreslením každého snímku, čímž je docíleno toho, že operace nad nečtenými daty mohou být započaty v podstatě ihned poté, kdy externí knihovny Kinectu načtou kompletní data, jak je možné vidět na ukázce Kód. 2.

```
private void Update()
{
  MultiSourceFrame reference = this.reader.AcquireLatestFrame();
  if(reference == null) return;
  using (DepthFrame depthFrame = reference.
         DepthFrameReference.AcquireFrame()) {
    if (depthFrame != null) {
      this.LoadData(depthFrame);
      depthFrame.Dispose();
    }
  }
  using (ColorFrame depthFrame = reference.
         ColorFrameReference.AcquireFrame()) {
    if (depthFrame != null) {
      this.LoadData(depthFrame);
      depthFrame.Dispose();
    }
  }
}
```

Kód. 2 Načítání dat

Metody `LoadData` slouží k extrakci surových dat ze vstupního snímku, který nám poskytuje daný senzor. Kopie těchto dat musejí být uloženy ve vlastních struktu-

rách, protože na každém sejmutém snímku, který Kinect vrací, musí být bezpodmínečně zavolána metoda `Dispose`, který zajišťuje uvolnění načtených dat. Pokud by tato metoda nebyla provolána, potom by již nikdy nedošlo k načtení dalšího snímku z daného senzoru.

6.1.2 Kalibrace dataprojektoru

Aby bylo možné korektně promítat data, je nejprve nutné správně napozicovat dataprojektor. Za tímto účelem je implementováno rozhraní, které může uživatel za tímto účelem využít.

Kalibrace je realizována na tom principu, že na výstupu dataprojektoru je promítána jednolitá barva, kdežto na standardním výstupu připojeného počítače je zobrazován barevný obraz viditelný v zorném poli hloubkového senzoru. K obrazu na standardním výstupu je navíc přidána průhledná mřížka, aby vylo lépe vidět, kde jsou umístěny jednotlivé objekty ve scéně a uživateli byla tak kalibrace usnadněna.

Problémem je, že načtená surová barevná data momentálně ještě nejsou ve stavu, aby mohla být za uvedeným účelem využita. Primárním problémem těchto dat je, že jsou sejmuta z barevného senzoru, jenž je na zařízení Kinect umístěn asi 10 cm vedle hloubkového senzoru, který snímá surová hloubková data. Mezi hloubkovým a barevným snímkem tedy vzniká rozdíl způsobený odlišnými pozicemi, ze kterých je snímek sejmut. Dalším problémem je, že hloubková data a barevná data jsou jak v jiném rozlišení, tak mají i rozlišný poměr stran. Informace o rozlišení poskytují přímo knihovny Kinectu spolu se sejmutým snímkem. Tyto hodnoty se neliší snímek od snímku, a proto s nimi aplikace pracuje jako s konstantami. Hodnoty těchto konstant udávají rozlišení v pixelech, kde pro Kinect v2 nabývají hodnoty 1 920×1 080 pro barevná data a hodnoty 512×424 pro hloubková data. Vzhledem k rozlišnému rozlišení a poměru stran musejí být tedy data nejdříve sjednocena. Knihovny Kinectu za tímto účelem poskytují třídu `CoordinateMapper`, která vrací koordináty, které nabývají měřitelných hodnot v případě, že údaje z barevného snímku spadají mezi hranice hloubkového snímku. Pokud koordináty nespadají do hloubkového snímku, potom nabývají hodnot kladného, nebo záporného nekonečna.

Metody třídy `CoordinateMapper` mohou být využity až v okamžiku, kdy jsou zcela načtena jak hloubková, tak barevná data. Standardně jsou tyto metody provolávány po provolání události `MultiSourceFrameArrived`, kterou aplikace z předešle uvedených důvodů nemůže využít, proto opět kontroluje stav načtení dat na každém provolání metody `Update` a veškerá data si po úspěšném načtení uloží do pole. Při každém provolání této metody je tedy nejdříve zkontrolována neprázdnost proměnných `depth` a `color`, což jsou reference na třídy, v nichž si aplikace interně ukládá veškerá načtená data. Pokud jsou data již úspěšně načtena, je možné bezpečně provolat metodu `MapDepthFrameToColorSpace`, což je klíčová metoda, které je v prvním parametru předáno pole surových dat načtených hloubkovým senzorem a ve druhém parametru je předáno pole typu `ColorSpacePoint`, což je `struct`, který knihovny Kinectu využívají k uchování dat o pozici

daného pixelu na barevném snímku. Tento `struct` obsahuje pouze dvě proměnné a to proměnnou `X` a `Y`, které odpovídají souřadnicím daného pixelu barevné mapy v hloubkové mapě, přičemž pokud barevná informace nemá pozici, na niž by se mohla promítnout, potom je hodnota automaticky nastavena na kladné nebo záporné nekonečno. Metoda poté hodnoty svého výstupu zapíše přímo do pole, které jí bylo předáno v druhém parametru (viz Kód. 3).

```
private void Update()
{
  if (this.depth != null && this.color != null) {
    if(this.ColorSpacePoints == null)
      this.ColorSpacePoints = new ColorSpacePoint
                              [this.depth.Raw.Length];
    this.Sensor.CoordinateMapper.MapDepthFrameToColorSpace
    (this.depth.Raw, this.ColorSpacePoints);
  }
}
```

Kód. 3 Načítání pole barevných koordinátů

Po načtení těchto dat zná již aplikace vše potřebné k tomu, aby mohla sestavit barevný snímek, který bude odpovídat snímku sejmutému z hloubkového senzoru. Pro konstrukci tohoto snímku aplikace využívá metodu `GetMappedTexture` (viz Kód. 4), které jsou v parametru předány všechny nezbytné vstupní informace. Metoda vrací objekt typu `Texture2D`, což je interní formát Unity pro uchovávání bitmapy. Metoda musí být vysoce optimalizována, protože pracuje s velkým množstvím dat a je nutné, aby mohl probíhat v extrémních případech i 30krát za sekundu, protože snímací frekvence Kinectu může být za optimálních podmínek až 30 fps (*frames per second*) (Microsoft, 1016). Za tímto účelem aplikace pracuje s daty na úrovni jednotlivých bitů bitmapy a až v poslední fázi načte data do návratového typu `Texture2D`, což je technika, která se oproti přímému definování barvy jednotlivých pixelů prokázala jako mnohem optimálnější po stránce výkonnosti.

Metoda pracuje na principu, kdy si nejdříve vytvoří objekt finální textury, která bude zároveň i jejím konečným výstupem. Této textuře nastaví výšku, šířku, formát uložených bitů a zakáže pro tuto texturu automatické generování mipmap. Následně je vytvořeno pole bytů, které musí být čtyřikrát delší, než je vstupní pole barevných koordinátů. Tato délka je dána faktem, že pole koordinátů nese na každém indexu informaci o celém pixelu a pro uložení kompletní barevné informace pixelu pomocí typu `byte` musí být informace uložena v podobě čtyř hodnot pro každý kanál zvlášť (RGBA). Dále jsou v cyklu procházena veškerá vstupní data a pro každý koordinát je provedena kontrola, zda se vyskytuje na barevné mapě. Pokud tomu tak je, potom je hodnota celého pixelu, na nějž koordinát odkazuje, zahrnuta do výsledného pole hodnot a pokud ne, tak je daný pixel přeskočen. Tím, že byl pixel přeskočen, mu je vlastně ponechána jeho defaultní hodnota, což je nula, která odpovídá barvě bílé. V poslední fázi jsou již data jen načtena do výstupního návratového typu pomocí dvou metod Unity `LoadRawTextureData` a `Apply`.

```
public Texture2D GetMappedTexture(ushort[] pDepth, byte[] pColor,
ColorSpacePoint[] pColorSpace, Dimension pDepthDim, Dimension pColor-
Dim)
{
  Texture2D result = new Texture2D(pDepthDim.Width,
  pDepthDim.Height, TextureFormat.RGBA32, false);
  byte[] raw = new byte[pColorSpace.Length * 4];
  int ind = 0;
  for (int i = 0; i < pColorSpace.Length; i++) {
    int colorX = (int)Math.Floor(pColorSpace[i].X + 0.5f);
    int colorY = (int)Math.Floor(pColorSpace[i].Y + 0.5f);
    if (colorX >= 0 && (colorX < pColorDim.Width)
        && (colorY >= 0) && (colorY < pColorDim.Height)) {
      int colorImageIndex = ((pColorDim.Width * colorY)
      + colorX) * 4;
      raw[ind++] = pColor[colorImageIndex];
      raw[ind++] = pColor[colorImageIndex + 1];
      raw[ind++] = pColor[colorImageIndex + 2];
      raw[ind++] = pColor[colorImageIndex + 3];
    }
    else {
      ind += 4;
    }
  }
  result.LoadRawTextureData(raw);
  result.Apply(false);
  return result;
}
```

Kód. 4 Namapování hloubkového snímku na barevný

Takto zpracovaný barevný snímek již svými parametry plně odpovídá hloubkovému snímku, a proto může být po přidání orientační mřížky odeslán na standardní výstup k vykreslení. Touto technikou je tedy dosaženo toho, že je na standardním výstupu vidět obraz, který je viditelný hloubkovým senzorem Kinectu.

Obr. 22 Ukázka standardního výstupu počítače při procesu kalibrace

6.1.3 Zpracování hloubkových dat

Hloubkový snímek, který je získán z Kinectu, obsahuje veškerá data, která jsou zapotřebí k vytvoření výškové mapy referenčního prostředí, avšak tato data obsahují mnoho neadekvátně vyhodnocených, či vyloženě chybných údajů. Data jsou poskytována v podobě pole datového typu `ushort` a udávají vzdálenost bodu v milimetrech.

Aby bylo možné s těmito daty dále pracovat, je nezbytné zcela rozumět jejich internímu obsahu a stanovit, jakou informaci přesně je nezbytné z těchto dat vytěžit. Vzhledem k tomu, že cílem je získání výškové mapy, kterou bude možné co nejpřesněji vizualizovat, je nezbytné z dat vytěžit takovou informaci, kterou bude možné ohodnotit barvou.

Vzhledem k tomu, že navržená aplikace mnohonásobě prochází celá vstupní data a provádí nad nimi různorodé úpravy, je nezbytné, aby barevná data byla uložena v co nejoptimálnější struktuře. V průběhu vývoje byla vyzkoušena řada metodik a formátů pro ukládání barevných údajů, kde vítězně vzešla metodika uložení všech surových dat do jednorozměrného pole datového typu `byte` s tím, že informace o výšce a šířce vstupní mapy, jejíž obsah data definují, si aplikace uchovává v samostatné struktuře mimo toto pole. Tato metodika je vybrána kvůli její rychlosti a nízké paměťové náročnosti, protože u ostatních testovaných struktur byly u takto dlouhých dat velké problémy spojené se zátěží *Garbage Collectoru* a i skrz to, že vesměs veškeré úkony probíhají hned v několika vláknech, byla rychlost zpracování dat neakceptovatelná.

Výstupem Kinectu je tedy pole typu `ushort`, ale po obarvení je nezbytné pole typu byte, takže musí být provedeno co nejšetrnější přetypování z typu `ushort` na `byte`. Brzké explicitní přetypování již v této fázi však není vhodné, protože nad daty typu `ushort` se dá aplikovat mnohem více vyhlazovacích a korekčních mechanismů.

Nejdůležitějším krokem je identifikace nepřesně vyhodnocených údajů. Za tímto účelem SDK Kinectu poskytuje informace o minimální a maximální spolehlivé vzdálenosti, kterou je Kinect schopen zachytit. Na data je tedy aplikována filtrace, která nalezne veškerá neadekvátně vyhodnocená data a přiřadí jim hodnotu nula, aby mohla být snadněji identifikována a zpracována v dalším kroku korekce dat. O tuto operaci se stará metoda `FilterMinMax`, které je v parametru předáno pole hodnot, nad kterými má být provedena filtrace a údaj o dolní a horní mezi, který poskytuje SDK Kinectu. Tento proces je možné vidět pod ukázkami Kód. 5 a Kód. 6.

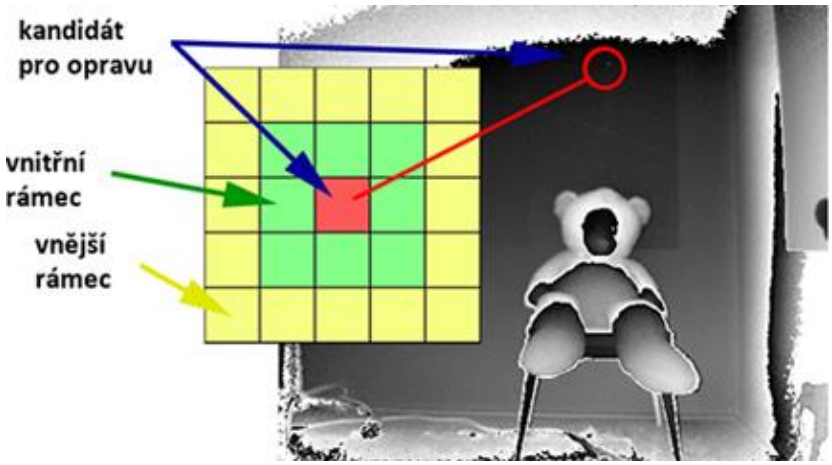
```
private void FilterMinMax(ushort[] pData, ushort pMin, ushort pMax)
{
  Parallel.For(0, pData.Length, i => {
    pData[i] = this.CorrectDepth(pMin, pMax, pData[i]);
  });
}
```

Kód. 5 Filtrace hloubky dat

```
private ushort CorrectDepth(ushort pMin, ushort pMax, ushort pValue)
{
  if (pValue <= pMin || pValue >= pMax) return 0;
  return pValue;
}
```

Kód. 6 Ohraničení hloubkového údaje

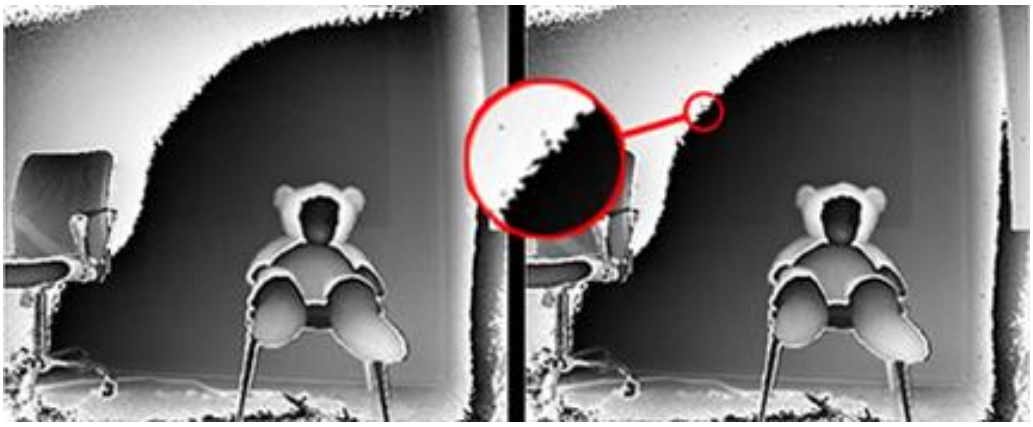
V nadcházejícím kroku jsou data znovu prohledána, ale tentokrát s tím rozdílem, že se aplikace blíže zaměřuje právě na takové údaje, které jsou rovny nule. V případě, že je takovýto údaj nalezen, jsou prohledány jednak jeho přímo sousedící hodnoty a také jeho nepřímo sousedící hodnoty, mezi kterými jsou vyhledávány nenulové hodnoty, jak je ilustrováno na Obr. 23.



Obr. 23 Filtrace pomocí vnitřního a vnějšího rámce

Princip algoritmu je ten, že počty nalezených nenulových hodnot jsou sčítány a tyto výsledné součty jsou nakonec srovnány vůči předem definované mezi, při jejímž překročení je provedeno zprůměrování okolních nenulových hodnot a výsledná hodnota je přiřazena zkoumanému bodu. Touto operací je vyhodnoceno, zda je hodnota bodu skutečně chybná a jedná se tedy s největší pravděpodobností o šum, nebo se jedná o objekty, které Kinect načetl správně, ale nebyly v adekvátní vzdálenosti pro zpracování.

Data jsou následně několikrát po sobě podrobena zmíněnému algoritmu, aby byla dosažena maximální možná úroveň korekce.



Obr. 24 Srovnání dat před a po filtraci pomocí vnitřního a vnějšího rámce

Samotný algoritmus je realizován metodou `FilterPixelByBounds` (viz Kód. 7), která na vstupu očekává veškerá hloubková data a údaje o výšce a šířce snímku, ze kterého byla hloubková data sejmuta. Vzhledem k délce metody je uvedena pouze její zjednodušená podoba v pseudokódu.

Metodě jsou v parametru předána surová vstupní data v podobě pole a také výška a šířka, které udává rozměry dvourozměrné reprezentace tohoto pole při převodu na bitmapu. Princip algoritmu je takový, že pomocí dvou vnořených cyklů prochází jak výšku, tak šířku pomyslné bitmapy a každý jeden pixel této bitmapy je podroben kontrole, zda je jeho hodnota nulová nebo ne. Pokud nesplňuje tuto podmínku, tak je jeho hodnota ponechána beze změny, ale pokud ano, potom přicházejí na řadu další korekční mechanismy. U takovéhoto bodu jsou v cyklu procházeny všechny body jeho vnitřního i vnějšího rámce (viz Obr. 23). U každého zpracovávaného bodu v kterémkoli z těchto rámců je za předpokladu, že je jeho hodnota nenulová, ukládána do dvourozměrné datové struktury, kde první položka udává hodnotu bodu a druhá položka počet již nalezených bodů, které nabývají stejné hodnoty. Pokud je tedy hodnota stejná, jako je hodnota zpracovávaného bodu, tak je pouze navýšena častost tohoto výskytu, čímž je dosaženo, že všechny hodnoty v této struktuře jsou unikátní a liší se jen podle častosti jejich výskytu. Aplikace si zároveň do proměnných `inter` a `outer` zapisuje, kolik nenulových hodnot bylo nalezeno ve vnějším rámci a kolik ve vnitřním. Nakonec, až je okolí nulového bodu zcela zpracováno, tak je realizován poslední krok algoritmu, kde

jsou hodnoty proměnných `inter` a `outer` srovnány vůči konstantě, která stanovuje minimální počet nenulových hodnot, který musí být nalezen, aby mohla být změněna hodnota zpracovávaného bodu. Pokud data schvalují uvedenou podmínku, je v konečné fázi hodnota zkoumaného bodu nahrazena takovou hodnotou, která měla v okolních rámcích nejvyšší častost. Princip tohoto algoritmu byl inspirován metodami využitými v práci *Smoothing Kinect Depth Frames in Real-Time* (Sandford, 2012).

```
FilterPixelByBounds(array[] data, int width, int height)
array[] result
for ya = 0 to height do
  for xa = 1 to width do
    array[,] filter // pole hodnot a častostí jejich výskytu
    int index = xa + ya * width    // index v jednorozměrném poli
    if data[index] == 0 then
      // počet nenulových hodnot ve vnitřním nebo vnějším rámci
      int inner = outer = 0
      for yb = -2 to 3 do
        for xb = -2 to 3 do
          if yb != 0 || xb != 0 then
            int index2 = (ya+yb)*width+(xa+xb)
            if(data[index2] == 0) then continue
            if filter.Contains(data[index2]) then
              filter[data[index2]]++
            else
              filter.Add(data[index2])
            if yb!=2 && yb!=-2 && xb!=2 && xb!=-2 then
              inner++
             else outer++
       if inner>thresh || outer>thresh then
       int id = 0
      for i = 1 to filter.lenght do
         if filter[id] < filter[i] then
          id = i
  result[index] = filter[id]
  else result[index] = data[index]
```

Kód. 7 Filtrace pomocí rámců
nezjednodušená podoba kódu: Příloha A.1

Doposud uvedený blok algoritmů, který tedy čítá nejdříve vyfiltrování chybně vyhodnocených dat a poté detekci a korekci těchto chyb, je proveden hned několikrát nad různými načtenými vstupními snímky. Takto filtrovaná výsledná data si aplikace ukládá do interní paměti a po zpracování příslušného počtu vstupních snímků jsou všechna tato data podrobena dalšímu algoritmu, který tato data navzájem normalizuje. Díky tomuto algoritmu je dosaženo dvou významných vylepšení. Především je eliminována možnost, že by data načtená Kinectem byla vadná. Tato situace by mohla nastat v případě, že by aplikace zpracovávala jen jednou sejmutá

vstupní data, která mohla být z jakéhokoli důvodu poškozena. Dále je docíleno dalšího navýšení průměrné přesnosti zkoumaných dat, protože při vyšším počtu sejmutých snímků je jen velmi malá pravděpodobnost, že by data zkoumaného pixelu byla na všech načtených snímcích sejmuta nekorektně.

O uvedené zprůměrování se stará metoda `CreateAverageDepthArray` (viz Kód. 8). Tato metoda přejímá v parametru dvourozměrné pole, které obsahuje již předchozími algoritmy upravená data několika po sobě jdoucích sejmutých snímků. Dále dostává opět informaci o výšce a šířce dvourozměrné bitmapy, kterou je možné nad každým sejmutým snímkem zkonstruovat. V prvním cyklu algoritmu je zhotoveno nové pole, jehož každý index obsahuje sumu hodnot ze všech vstupních polí na stejném indexu. Toto pole je následně znovu celé projdeno a na základě každé hodnoty a počtu vstupních polí je vypočítán podíl, který je zapsán do výstupního pole.

```
int[] CreateAverageDepthArray(int[][] depthArray, int width,
int height)
  // pole obsahující sumu hodnot všech zpracovávaných snímků
  int[] sumDepthArray
  int[] averagedDepthArray
  for i = 0 to depthArray.Length do
    for y 0 to height do
      for x = 0 to width do
        int index = x + y * width
        int val = depthArray[i][index] * count
        sumDepthArray[index] += val
  for y = 0 to height do
    for x = 0 to width do
      int index = x + y * width
      averagedDepthArray[index] = sumDepthArray[index]
                                  / depthArray.Lenght
      return averagedDepthArray
```

Kód. 8 Normalizace dat mezi více snímky
nezjednodušená podoba kódu: Příloha A.2

Nyní jsou data již normalizována a je z nich odstraněn veškerý šum. Proto již mohou být data přetypována na `byte`, čímž zároveň proběhne převod z výškové informace na informaci o barvě.

`Ushort` je datový typ, který je schopen nést hodnotu maximálně 65 535, která je uložena v 16 bitech, kdežto `byte` je mnohem menší datový typ schopný nést maximálně hodnotu 255 a jeho celková délka činí 8 bitů.

Vzhledem k rozdílným bitovým délkám není možné využít implicitní bezeztrátové přetypování, ale musí být využito explicitní přetypování, které by však na aktuálních datech zapříčinilo ztrátu některých významných bitů a výsledná hodnota by byla naprosto znehodnocena. Oficiální dokumentace Kinectu udává, že maximální vzdálenost, na kterou je ovladač Kinect schopen spolehlivě rozpoznat pozici bodu, je 45 000 milimetrů, což odpovídá bitovému přepisu 0001000110010100.

Z maximální hodnoty, kterou mohou vstupní data obsahovat, je zřejmé, že první tři bity jsou zcela redundantní, protože jejich obsah přesahuje maximální spolehlivou vzdálenost snímače a tyto bity mohou být tedy anulovány.

Z dokumentace dále vychází i minimální vzdálenost, na niž je Kinect schopen spolehlivě rozpoznat vzdálenost bodu a jedná se o vzdálenost 500 milimetrů. Z tohoto faktu vyplývá, že všech spodních osm bitů je taktéž redundantních.

Pro nejkvalitnější možnou extrakci relevantních dat je použit algoritmus, který posune bity o 5 jednotek napravo, čímž jsou vyfiltrovány redundantní první tři bity, ale přitom jsou zachovány první tři bity z nižšího bitu, čímž je dosaženo přetypování bez jakékoli ztráty relevantních dat.

Obr. 25 Vizualizace bitového posunu

Obr. 26 Konečný výstup všech opravných algoritmů

6.1.4 Rekonstrukce scény

Data jsou nyní tedy vyfiltrována, je odstraněn šum, jsou opraveny chybné sektory a každá výšková hodnota je jednoduše vizualizovatelná. V druhé fázi již tedy přichází na řadu sestavení sítě polygonů, a tím vytvoření virtuální 3D reprezentace načtené scény. Unity jakožto platforma, která je primárně určena k vývoji her, má velice jednoduché rozhraní pro vytváření 3D objektů, které má však svá jistá omezení. Největším omezením je maximální počet polygonů, který připadá na jeden objekt a to 65 000 (Unity3d verze 5.4.3f3). Vstupní data hloubkového senzoru jsou však mnohem delší. Vstupní data mají rozlišení 512×424, z čehož vyplývá, že pro věrný přepis každého bodu na vertex by bylo zapotřebí 217 088 vertexů, což není za pomoci Unity3D technologicky možné. Data se tedy musejí nejprve zmenšit. Cílem je, aby nutná komprese byla co nejnižší, proto je nová výška a šířka vypočtena tak, aby se výsledný počet vertexů co nejvíce přiblížil maximálnímu možnému po-

čtu vertexů zpracovatelného v Unity3D, při zachování původního aspektu výšky a šířky. Tohoto je docíleno díky metodě `CalcWidthAndHeight` (viz Kód. 9), která v parametru přijímá `struct`, který nese informaci o aktuální výšce a šířce hloubkové mapy a vrací nově vypočtenou výšku a šířku, jejíž součin je menší než maximální možný počet polygonů sestavitelný za pomoci Unity3D.

```
private Dimension CalcWidthAndHeight(Dimension pDim)
{
  float raito = (float)pDim.Width / (float)pDim.Height;
  // zjednodušená rovnice x * x * raito = max verts
  float x = Mathf.Sqrt(MaxUnityVertertexes / raito);
  return new Dimension(Mathf.RoundToInt(x * raito),
                       Mathf.RoundToInt(x));
}
```

Kód. 9 Výpočet zmenšeného rozlišení

Dále je nezbytné výškovou mapu zmenšit do vypočtené velikosti. Pro tento účel aplikace využívá statické metody `ThreadedScale` (viz Kód. 10), která na principech bilineární interpolace provede zmenšení dat. Byly testovány i jiné algoritmy, jako *k*-NN (*k-nearest neighbor*), nebo *box sampling*, ale algoritmus pro bilineární interpolaci nakonec tyto metody předčil díky nejlepšímu poměru výsledné kvality a rychlosti při vícevláknovém zpracování.

Aplikace v prvotní fázi nejdříve na základě počtu jader procesoru rozdělí data na celky, které budou paralelně zpracovávány. Tyto celky jsou následně v parametru odeslány metodě `BilinearScale` (viz Kód. 11), ve které jsou data dále zpracovávána.

Vzhledem k tomu, že se jedná o statickou metodu, která zpracovává data ve více vláknech, je nezbytné veškeré vstupní informace ukládat do proměnných samotné třídy. V následujícím bloku kódu je ukázána inicializace těchto dat a u jednotlivých proměnných je v komentářích uvedeno, co daná proměnná udává. Pro realizaci vícevláknového zpracování je využito tříd jazyka *c#* pro vícevláknové zpracování. Jmenovitě se jedná o třídy `Mutex` a `Thred`, jejichž bližší specifikace je popsána v dokumentaci MSDN (*Microsoft Developer Network*). Metoda `Start` třídy `Thread` přijímá v parametru objekt, do nějž aplikace odesílá svou interní třídu `ThreadData`, což je jednoduchá datová struktura s proměnnými `start` a `end`, které udávají rozsah indexů zpracovávaného pole dat, který je přidělen pro dané vlákno.

```
static void ThreadedScale (Texture2D pTex, int pNewWidth, int pNe-
wHeight)
{
  texColors = pTex.GetPixels(); // pole barev vstupní bitmapy
  // pole barev výstupní bitmapy
  newColors = new Color[pNewWidth * pNewHeight];
  // měřítko výsledné
  ratioX = 1.0f / ((float)pNewWidth / (pTex.width - 1));
```

```
  ratioY = 1.0f / ((float)pNewHeight / (pTex.height - 1));
  w = pTex.width;  // šířka vstupních dat po ose X
  w2 = pNewWidth;  // šířka výstupních dat po ose X
  // počet jader procesoru
  int cores = Mathf.Min(SystemInfo.processorCount, pNewHeight);
  int slice = pNewHeight/cores; // rozsah indexů pro každé jádro
  finishCount = 0; // počet dokončených vláken
  mutex = new Mutex(false);
  for (int i = 0; i < cores - 1; i++) {
    ThreadData threadData = new ThreadData(slice*i,slice*(i+1));
    Thread thread = new Thread(BilinearScale);
    thread.Start(threadData);
  }
  while (finishCount < cores) Thread.Sleep(1);
  pTex.Resize(pNewWidth, pNewHeight);
  pTex.SetPixels(newColors);
  pTex.Apply();
}
```

Kód. 10 Změna velikosti bitmapy

Metoda `BilinearScale` postupně prochází celý jí přidělený blok dat a lineárně interpoluje data po ose X a Y. Výslednou barvu vypočtenou pomocí metody `Color-LerpUnclamped` ukládá do globální proměnné `newColors`, ze které je v konečném důsledku zrekonstruována výsledná bitmapa.

```
void BilinearScale (ThreadData threadData)
  for int y = threadData.start to threadData.end do
    int yFloor = y * ratioY
    int y1 = yFloor * w // w - šířka vstupních dat po ose X
    int y2 = (yFloor+1) * w
    int yw = y * w2 // w2 - šířka výstupních dat po ose X
    for int x = 0 to w2 do
      int xFloor = x * ratioX;
      float xLerp = x * ratioX - xFloor;
      newColors[yw + x] = ColorLerpUnclamped(
      ColorLerpUnclamped(texColors[y1 + xFloor],
        texColors[y1 + xFloor+1], xLerp),
      ColorLerpUnclamped(texColors[y2 + xFloor],
        texColors[y2 + xFloor+1], xLerp),
      y*ratioY-yFloor)
    // ukončení mutextu a navýšení počtu dokončených vláken
    mutex.WaitOne()
    finishCount++
    mutex.ReleaseMutex()
```

Kód. 11 Algoritmus bilineární změny velikosti bitmapy
nezjednodušená podoba kódu: Příloha A.3

Pro interpolaci je využívána metoda `ColorLerpUnclamped` (viz Kód. 12), což je metoda, která oproti standardně používané metodě Unity `Color.Lerp` umožňuje lineární interpolaci v její neohraničené podobě. Algoritmus neohraničené lineární interpolace dvou hodnot vychází ze standardního vzorce pro lineární interpolaci dvou čísel `x` a `y` hodnotou `t`, kde `t` bývá většinou ohraničena od nuly do jedné. Tento vzorec je možné vyjádřit rovnicí `x + (y – x) * t`, kde `t ∈ (0, 1)` a z něj také vychází princip následujícího algoritmu, pouze s tím rozdílem, že `t` není omezeno hranicemi. Metoda tuto interpolaci provede pro každý barevný kanál zvlášť.

```
private static Color ColorLerpUnclamped (Color pColor1, Color pColor2,
float pValue)
{
  return new Color(pColor1.r + (pColor2.r - pColor1.r) * pValue,
         pColor1.g + (pColor2.g - pColor1.g) * pValue,
         pColor1.b + (pColor2.b - pColor1.b) * pValue,
         pColor1.a + (pColor2.a - pColor1.a) * pValue);
}
```

Kód. 12 Neohraničená lineární interpolace barvy

Nad takto upravenou výškovou mapou je již možné zrekonstruovat síť polygonů. Aby bylo možné přenést výškové informace z výškové mapy do sítě polygonů, musí být nejdříve vytvořena plocha o odpovídajícím počtu vrcholů a trojúhelníků. Dále je pro síť polygonů vypočtena i mapa UV koordinátů, kterou aplikace sice sama k ničemu nevyžívá, ale je díky tomu značně navýšena použitelnost aplikace za jinými účely, než je zamýšlený cíl práce. Díky tomuto vylepšení je velice jednoduché aplikaci předělat na jiný podobný produkt, jako může být například *Sandbox*, nebo *RoomAlive*. Za účelem vytvoření základní plochy aplikace využívá metodu `CreateMesh` (viz Kód. 13), která realizuje všechny výše uvedené úkony.

```
private void CreateMesh(Texture pMap)
{
  this.mesh = new Mesh();
  this.meshPlane.mesh = this.mesh;
  this.CalcWidthAndHeight(pMap);
  this.verts = new Vector3[this.width * this.height];
  this.uv = new Vector2[this.width * this.height];
  this.trits = new int[6*(this.width - 1) * (this.height - 1)];
  this.CalculateMesh(this.height, this.width, this.verts,
  this.trits, this.uv);
  this.mesh.vertices = this.verts;
  this.mesh.uv = this.uv;
  this.mesh.triangles = this.trits;
  this.mesh.RecalculateNormals();
}
```

Kód. 13 Inicializace sítě polygonů

Předchozí algoritmus využívá pro svou funkci metodu `CalculateMesh` (viz Kód. 14), což je metoda, která naplní pole jí předané v parametru defaultními daty. Princip je ten, že data jsou procházena po jejich výšce a šířce a každému vrcholu v prostoru je přiřazena jeho korektní pozice na ploše, index v UV mapě a jeho návaznosti na přilehlé vrcholy, čímž je jeho hodnota zanesena i do pole trojúhelníků pro vykreslování.

```
void CalculateMesh(int height, int width, Vector3[] verts, int[]
trits, Vector2 uv)
int triangleIndex = 0;
  for int y = 0 to height do
    for int x = 0 to width do
      int index = y * width + x;
      verts[index] = new Vector3(x, -y, 0f);
      uv[index] = new Vector2(x / width, y / height);
      if x != (width - 1) && y != (height - 1) then
        int topLeft = index;
        int topRight = topLeft + 1;
        int bottomLeft = topLeft + width;
        int bottomRight = bottomLeft + 1;
        trits[triangleIndex++] = topLeft;
        trits[triangleIndex++] = topRight;
        trits[triangleIndex++] = bottomLeft;
        trits[triangleIndex++] = bottomLeft;
        trits[triangleIndex++] = topRight;
        trits[triangleIndex++] = bottomRight;
```

Kód. 14 Vytvoření sítě polygonů
nezjednodušená podoba kódu: Příloha A.4

Takto vytvořená plocha již obsahuje veškeré potřebné údaje a posledním krokem je doplnit výškovou informaci. V následující metodě (viz Kód. 15) je kromě zanesení výškové informace pro ilustraci opět uveden i způsob, kterým je možné data obarvit, a tím získat plnohodnotnou reprezentaci prostoru včetně korektních a správně namapovaných barevných údajů.

```
private void RefreshData(Texture2D map, int frameWidth, int rawLenght)
{
  ColorSpacePoint[] colorSpace = new ColorSpacePoint[rawLenght];
  for (int y = 0; y < map.height; y++) {
    for (int x = 0; x < map.width; x ++) {
      int smallIndex = y * map.width + x;
      this.verts[smallIndex].z = map.GetPixel(x,y).r;
      ColorSpacePoint colPoint = colorSpace[(y*frameWidth)+x];
      this.uv[smallIndex] = new Vector2(colPoint.X / map.width,
                                        colPoint.Y / map.height);
      }
    }
```

```
  this.mesh.vertices = this.verts;
  this.mesh.uv = this.uv;
  this.mesh.triangles = this.trits;
  this.mesh.RecalculateNormals();
}
```

Kód. 15 Přepočet sítě polygonů

Obr. 27 Obrázek vygenerované sítě polygonů

6.1.5 Implementace ukázkové aplikace

Ukázková aplikace se celá odehrává nad virtuálním terénem, který je zrekonstruován na základě reálných objektů ve scéně, přičemž uživateli je navíc umožněno si hloubku tohoto terénu virtuálně upravit. Ukázková aplikace je hrou pro dva hráče, kde každý hráč ovládá svou postavu, přičemž oba hráči ovládají svou postavu za pomocí jedné klávesnice. Ve hře jsou kromě virtuálních hráčů i jiné virtuální objekty.

Po spuštění se aktivuje událost, která zajistí, že terénem začne prolétat virtuální letadlo, které na náhodných pozicích vypustí do scény virtuální objekty (jednotlivé hráče a lékárny), které se díky fyzikálnímu systému Unity postupně snášejí k zemi, dokud nedopadnou na virtuální terén. Cílem hry je porazit protivníkova hráče, kde za tímto účelem hráči vyžívají reálné objekty scény, za kterou se mohou schovávat, nebo se po ní pohybovat.

Ukázková aplikace přistupuje ke konstrukci virtuálního terénu mírně odlišným způsobem než výše uvedený princip vykreslování virtuální sítě polygonů, který data vykresluje přímo na základě výškové mapy. Vzhledem k tomu, že virtuální terén je před uživatelem skryt a je tedy využíván jen pro zhotovení kolizních bloků, je nejdříve výšková mapa zjednodušena, aby kolizní bloky byly co nejhladší a neskládaly se ze zbytečně složitých útvarů. Tohoto je dosaženo pomocí metody `SimplyfyImage` (viz Kód. 16). Tato metoda pracuje na principu, kdy projde celou hloubkovou mapu tak, že namísto toho, aby ji procházela pixel po pixelu, přeskakuje pixely o délku předem definovaného kroku. Hodnoty všech pixelů, které spadají do rozsahu daného kroku, zprůměruje. Hodnoty všech těchto pixelů jsou následně

nahrazeny vypočtenou průměrnou hodnotou, čímž je dosaženo značného zjednodušení vstupních dat, jak je možné vidět na následujícím obrázku.

```
void SimplyfyImage(Texture2D tex)
for int y = 0 to tex.height do
  y += this.step
  for int x = 0 to tex.width do
  x += this.step
    float sum = 0f
    for int y2 = 0 to this.step do
      for int x2 = 0 to this.step do
        if x + x2 > tex.width || y + y2 > tex.height then continue
        sum += tex.GetPixel(x + x2, y + y2).r
    float val = sum / (this.step * this.step)
    Color col = new Color(val, val, val, 1f)
    for int y2 = 0 to this.step do
      for int x2 = 0 to this.step do
        if x + x2 > tex.width || y + y2 > tex.height then continue
        tex.SetPixel(x + x2, y + y2, col)
```

Kód. 16 Algoritmus zjednodušení hloubkové mapy
nezjednodušená podoba kódu: Příloha A.5

Obr. 28 Původní hloubková mapa a zjednodušená hloubková mapa

Aby bylo uživateli umožněno rozhodnout, které objekty chce do výsledného virtuálního terénu zahrnout, je implementováno grafické rozhraní, díky kterému si uživatel může stanovit jistý hloubkový limit, což je mezní hodnota, na základě které se modifikuje hloubková mapa. Takto upravená hloubková mapa v konečném důsledku obsahuje pouze dvě barvy, přičemž jedna barva odpovídá maximální hloubce dat a druhá barva odpovídá hloubce nula. Tato funkce je realizována metodou `ColorzeData` (viz Kód. 17), která jako vstupní parametry přijímá hrubá výšková data a rozlišení výsledné textury. Metoda projde veškerá data, srovná je s proměnnou `normalizedMaxDepth`, což je proměnná, kterou uživatel sám nasta-

vuje pomocí grafického GUI. Na základě této proměnné je následně pro každý pixel přiřazena jeho výsledná barva, která demonstruje jeho hloubku.

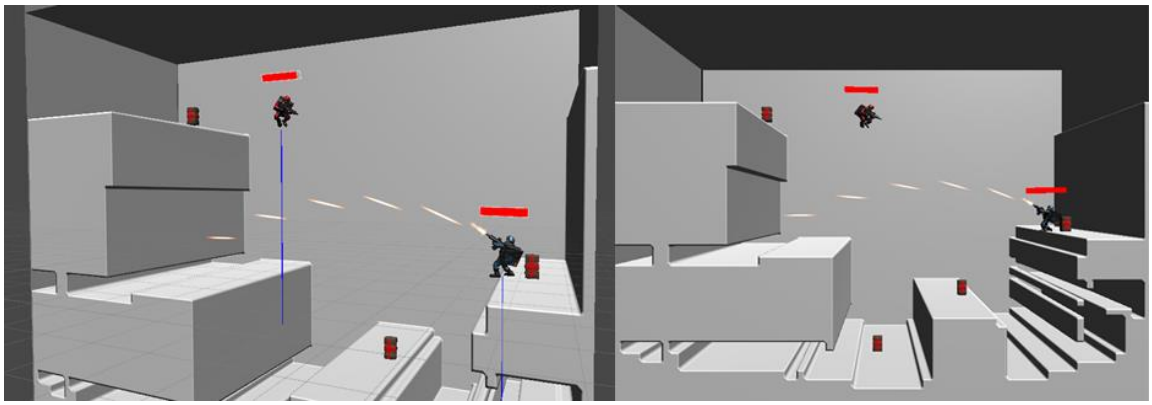
```
private Texture2D ColorizeData(byte[] data, int width, int height)
{
  Texture2D map = new Texture2D(width, height,
                                TextureFormat.RGB24, false);
  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      int smallIndex = y * width + x;
      if (pData[smallIndex] <= this.normalizedMaxDepth)
        map.SetPixel(x, y, Color.blue)
    else
      map.SetPixel(x, y, Color.red);
  }
  map.Apply(false);
  return map;
}
```

Kód. 17 Algoritmus pro obarvení hloubkové mapy

Zmíněné grafické rozhraní obsahuje tři posuvníky. Díky prvnímu posuvníku je uživateli umožněno určit avizovanou mezní hodnotu a výsledná zpracovaná hloubková textura je mu zobrazovaná jak na výstupu dataprojektoru, tak v okně náhledu, jak je možné vidět na Obr. 29. Další dva posuvníky slouží k případné úpravě horizontální a vertikální pozice kamery ve virtuální scéně, což je nezbytné v případě, že vzdálenost mezi senzorem Kinectu a dataprojektorem je v reálném světě příliš velká a vznikala by kvůli tomu nepřesnost.

Obr. 29 Ukázka grafického GUI

Pomocí takto upravené hloubkové mapy je následně vygenerován terén za pomoci metody `CreateMesh`, která je popsána v předcházející podkapitole. K tomuto virtuálnímu terénu jsou přidány ještě další čtyři rovné plochy, které fungují jako jakýsi rám okolo celého terénu, který zajišťuje, aby se hráč nikdy nemohl dostat za hranice tohoto terénu a tedy nebylo za žádných okolností možné opustit zorné pole dataprojektoru. V posledním kroku je objektu zhotoveného terénu vypnuta komponenta `MeshRenderer`, což je klíčová komponenta, kterou Unity využívá k vykreslování jeho polygonů. Díky tomu je zajištěno, že ač terén ve virtuální podobě stále existuje a aplikace může i nadále těžit ze všech s tímto faktem spjatých funkcionalit, jako jsou fyzikální výpočty, tak není viditelný na výstupu dataprojektoru, ale veškeré virtuální objekty, které s tímto virtuálním terénem jakkoli interagují, vykreslovány jsou a je zachována jejich funkcionalita.



Obr. 30 Ukázka virtuální reprezentace z pohledu Unity3D

O ovládání průběhu celé hry se stará třída `GameController`, což je jediný singleton v celém projektu. Tato třída se stará o vytvoření hráčů, objektů, které mohou hráči sbírat, restart hry při smrti jednoho z hráčů, ukončení hry apod.

Pro start i restart hry je využívána metoda `InitGame`. Tato metoda nejdříve vymaže všechny dynamické objekty scény, které se uvedenému singletonu na principu návrhového vzoru obeserver v průběhu hry registrují a poté vytvoří virtuální letadlo, které přeletí nad virtuálním terénem a na náhodných pozicích vytvoří všechny virtuální objekty pro celou hru.

Obr. 31 Ukázka běžící ukázkové aplikace

Hra sama o sobě používá různé pokročilé metody a techniky, které umožňuje engine Unity3D, jako jsou míření hráčů pomocí inverzní kinematiky, vícevrstvé animace pro oddělení animace horní a dolní poloviny těla hráčů, přímá úprava hashovaných hodnot na shaderu, která se využívá pro záblesky při výstřelech se zbraní, částicové efekty pro simulaci ohně vycházejícího z tryskových batohů hráčů apod. Tyto techniky přesahují rozsah a stanovené cíle této práce a jejich popis nebo přiložení do příloh práce by příliš navýšilo délku textu. Z tohoto důvodu jsou k nalezení pouze v podobě čistého kódu na CD ze zdrojového souboru, přiloženému k této práci.

7 Diskuze

Tato kapitola se zabývá vyhodnocením realizovaného projektu. Rozebrána jsou zejména technologická omezení dané platformy Unity a nástroje Kinect. V závěru následuje návrh pro možná budoucí vylepšení aplikace a možnosti pro využití zkoumaného řešení v praxi.

7.1 Realizace

Navržená aplikace je schopna velice přesného a precizního převodu reálného světa do jeho virtuální podoby. Největším úskalím tohoto převodu bylo kromě problémů dílčích, jako byla integrace knihoven Kinectu s enginem Unity, nebo nalezení vhodných prostor pro realizaci vzorové aplikace, především oprava, normalizace a filtrace vstupních dat, která Kinect poskytuje. Data poskytovaná senzory jsou podrobena řadě algoritmů, které značně vylepšují jejich přesnost a zajišťují, aby bylo maximálně možně využito potencionálu nástroje Kinect. Snahou je, aby vizualizovaná data byla pokud možno co nejpřesnější, proto bylo nezbytné pracovat s těmito daty v jejich nativní nezmenšené podobě, a mít na paměti, že každý údaj, který je chybně vyhodnocen, může ve výsledku způsobit značnou nepřesnost na ve finále zkonstruované síti polygonů. I přesto, že je polygonová síť uživateli skryta, tak na jejím základě jsou vypočteny kolizní bloky, které jsou naprosto nezbytné pro adekvátní funkčnost výsledné vzorové aplikace, kde veškerý pohyb ve scéně je realizován za pomoci zabudovaného fyzikálního enginu Unity, a případné chyby nad těmito daty by mohly zapříčinit nepředvídatelné vzorce chování tohoto systému.

V průběhu vývoje bylo nutné čelit mnoha technologickým omezením, kde některá z nich se aplikaci podařilo vcelku obstojně překonat, avšak za cenu snížení věruhodnosti dat, protože chybná data musela být podrobena predikčním algoritmům. Tyto situace nastaly především v případech, že byla vyhodnocena data, jejichž hodnota neodpovídala minimální, nebo maximální spolehlivé vzdálenosti od senzoru.

Při testování implementace v jiných než laboratorních podmínkách bylo identifikováno několik technologických omezení, která se nepodařilo nijak odbourat a soudě dle rešerše, která byla pro tyto specifické situace zpětně provedena, není v momentální době ani technologicky možné se jim vyhnout, nebo by korektura těchto dat výrazně překračovala rámec zpracovávané problematiky. Nejvýznamnějším problémem je, že senzory Kinectu neadekvátně vyhodnocují výškové údaje za předpokladu, že Kinect není umístěn kolmo k dominantní ploše, nad kterou má být realizována samotná projekce. Při naklonění těchto senzorů se projevuje značná chybovost dat. Dalším velkým problémem je omezení prostor, nad kterými může aplikace pracovat. Kinect sám o sobě není technologicky postaven na to, aby byl schopen správně pracovat ve venkovním prostředí, kde jsou snímaná data podrobena slunečnímu ultrafialovému záření, protože je takto naprosto anulována věrohodnost odrazu infračervených paprsků, které vysílá sám Kinect. Dále bylo zjiště-

no, že senzory nedokáží spolehlivě detekovat různé specifické typy materiálů, jako jsou například zrcadlo, lesklé projekční plátno, nebo sklo.

Výsledná aplikace byla otestována v laboratoři na testovacím vzorku uživatelů, který čítal sedm osob. Všichni z testovaných si ve většině případů dokázali bez větších problémů kalibrovat projekci a aplikaci nad jimi navrženými referenčními prostory spustit a testovat. V průběhu testování nebyla zaznamenána žádná chyba a zpětná vazba testovacích subjektů byla ve všech případech kladná.

7.2 Možná vylepšení

I přesto, že aplikace splňuje všechny definované cíle, tak existuje několik funkcionalit, které by bylo rozumné rozšířit, aby mohla být navýšena užitnost implementace.

Aplikace sama o sobě je navržena tak, aby její případná nadstavba byla co nejjednodušší. Aplikace již i v aktuální podobě implementuje různé funkcionality, které sice nejsou její podstatou a v projektu nejsou využívány, leč umožňují případně velice rychlé předělání funkcionality aplikace. Příkladem může být mapování UV koordinátů pro výslednou polygonovou síť, díky čemuž by bylo možné aplikaci velice rychle předělat například na *Sandbox*, nebo *RoomAlive*.

Bezesporu nejdůležitějším vylepšením by byla automatizace kalibrace projektoru, které prozatím probíhá manuálně. Pro takovéto vylepšení by bylo nezbytné rozšíření hardwarových prostředků o další snímací zařízení, které by bylo schopné nejspíše na principu detekce markérů rozpoznat pozici Kinectu vůči dataprojektoru v prostoru scény a na základě těchto údajů by automatizovalo kalibraci. Dalším vylepšením by mohlo být automatické pozicování pohledové kamery uživatele, které by muselo být taktéž realizováno za pomoci dalšího snímacího zařízení. Posledním vhodným vylepšením by mohlo být upravit implementaci tak, aby bylo možné načtenou scénu dynamicky upravovat i za běhu aplikace, protože stávající implementace pracuje na principu, kdy je scéna sejmuta a vyhodnocena pouze na začátku a poté již pracuje s neměnnou vnitřní reprezentací virtuální scény.

7.3 Možnosti využití a zhodnocení ekonomických aspektů

Kritickým kamenem úrazu využitelnosti je především časová náročnost nutná ke zpracování dat, z čehož vyplývá, že využití implementovaného řešení v reálném čase, například pro zábavní průmysl, kde je nezbytné několikrát za sekundu data znovu načítat a provádět nad nimi různorodé akce, není vhodné, protože časová složitost je i přesto, že aplikace vesměs veškeré úkony provádí ve vláknech a zátěž Garbage Collectoru je silně optimalizována, příliš vysoká. Naopak při využití buď pro vědeckou oblast, nebo pro podniky zabývající se co nejpřesnějším převodem dat do virtuální podoby (pro 3D tisk, modelování 3D objektů, skenování modelů map, budov apod.), kde časová složitost buď není klíčovým parametrem, nebo disponují dostatečně výkonným hardwarem, je reálné. Významné využití by mohlo být také například v geografickém oboru, kde se mnohdy řeší problémy vzájemné

viditelnosti. Problém viditelnosti je komplexně vyřešen například i v navržené vzorové aplikaci při střelbě hráčů mezi sebou.

Posledním uvedeným a nejspíše nejsilnějším, ne však posledním, aspektem využitelnosti je především řešení problémů v oboru projekce pomocí dataprojektorů. Problém v tomto oboru nastává v případě, že je nutné provádět projekci na ne zcela rovné ploše, a obraz je tím pádem deformován. Implementované řešení však nabízí možnost korektního UV mapování načtené scény, a tím umožňuje korektní vykreslení snímku i na nerovné povrchy.

Uvedené možné způsoby využití je možné implementovat za relativně nízkých nákladů, protože zařízení Kinect je samo o sobě považováno za jeden z nejlevnějších prostorových senzorů vůbec. Finančně nákladnější část tvoří dataprojektor, protože je nutné používat nadstandardní zařízení schopné promítat na blízkou vzdálenost a musí poskytovat široké spektrum možností nastavitelnosti zorného pole.

8 Závěr

Cílem diplomové práce bylo vizualizovat virtuální objekty v reálném prostoru s využitím nástroje Unity. Pro splnění hlavního cíle bylo nutné splnit cíle dílčí. Prostudovat existující projekty založené na vizualizaci prostřednictvím projekce. Analyzovat a srovnat knihovny pro nástroj Kinect. Implementovat funkcionalitu realizující opravná, filtrační a normalizační opatření nad daty získanými ze senzorů a rekonstruovat virtuální scénu generovanou z těchto dat. Navrhnout demonstrační aplikaci, která za pomoci projekce prověří kvalitu a spolehlivost virtuálních dat. Vyhodnotit a diskutovat výsledné řešení.

V úvodní teoretické části práce bylo rozebráno zařízení Kinect po stránkách jeho vývoje, schopností, principu funkce a omezení. Dále byla prozkoumána stávající řešení podobné problematiky, která ke své funkci využívají zařízení Kinect.

Dále byl popsán herní engine Unity3D, který byl analyzován především za účelem, zda bude poskytovat veškeré nástroje nutné k dosažení stanoveného cíle. Díky této rešerši bylo zjištěno, že Unity3D má svá omezení, která vedou k nutnosti zmenšovat hloubkovou mapu, ze které se rekonstruuje síť polygonů. Na základě tohoto zjištění byla provedena rešerše algoritmů, pro změnu velikostí bitmapy, aby bylo možné pro dané cíle identifikovat nejvhodnější algoritmus.

Následně byla provedena rešerše aktuálně poskytovaných knihoven, schopných komunikace se zařízením Kinect. Tyto knihovny byly zhodnoceny z několika klíčových aspektů a byly srovnány jejich silné a slabé stránky.

Na základě stanovené metodiky se přešlo k samotné implementaci. Po úspěšném propojení zařízení Kinect a enginu Unity3D, za použité knihovny *Kinect for Windows SDK 2.0*, která byla na základě rešerše knihoven stanovena jako nejvhodnější, se přešlo již k samotné realizaci zamýšleného cíle. Byly implementovány korekční algoritmy, které značně navyšují přesnost hloubkových dat a prověřují jejich validitu.

Posledním krokem vlastní práce bylo zhotovení vzorové aplikace, která je schopná načítat prostorová data, obohatit je o virtuální objekty a v takto upravené podobě je promítat pomocí dataprojektoru připojeného k počítači. Celý proces byl v konečném důsledku diskutován, kde byly zhodnoceny jak jeho silné, tak slabé stránky, byly navrženy možnosti pro případná vylepšení a byl zhodnocen přínos a možná využití zkoumané problematiky v praxi.

Stanovené cíle byly tedy v konečném důsledku splněny a byla i implementována řada vylepšení nad rámec těchto cílů.

9 Literatura

MICROSOFT, *Kinect for Windows Sensor Components and Specifications*. Msdn.microsoft.com [online]. [cit. 2016-11-27]. Dostupné z: https://msdn.microsoft.com/en-us/library/jj131033.aspx

L. FLATLEY, JOSEPH. *Visualized: Kinect + night vision = lots and lots and lots of dots (video)* [online]. 11. 08. 10 [cit. 2016-11-28]. Dostupné z: https://www.engadget.com/2010/11/08/visualized-kinect-night-vision-lots-and-lots-and-lots-of-do/

PAGLIARI, DIANA A LIVIO PINTO. *Calibration of Kinect for Xbox One and Comparison between the Two Generations of Microsoft Sensors*. Sensors [online]. 2015, 15(11), 27569-27589 [cit. 2016-11-28]. DOI: 10.3390/s151127569. ISSN 1424-8220. Dostupné z: http://www.mdpi.com/1424-8220/15/11/27569/

UNITY TECHNOLOGIES. *Unity - Fast Facts*. Unity3D [online]. 2016 [cit. 2016-11-28]. Dostupné z: https://unity3d.com/public-relations

BIMBER, OLIVER A RAMESH. RASKAR. *Spatial augmented reality: merging real and virtual worlds*. Wellesley, Mass.: A K Peters, c2005. ISBN 1568812302.

P. MILGRAM, H. TAKEMURA ET AL. „*Augmented Reality: A Class of Displays on the Reality-Virutality Continuum.*" In Proceedings of SPIE: Telemanipulator and Telepresence Technologies 2351 (1994), 282–292.

P. MILGRAM AND F. KISHINO. „*A Taxonomy of Mixed Reality Visual Displays*." IEICE Transactions on Information Systems E77-D 12 (1994), 1321–1329.

BEAUMONT, CLAUDINE. *E3 2009: Microsoft launches Project Natal Xbox 360 controller-free games system*. In: The Telegraph [online]. Los Angeles: Claudine Beaumont, 2009 [cit. 2016-11-29]. Dostupné z: http://www.telegraph.co.uk/technology/e3-2009/5424429/E3-2009-Microsoft-launches-Xbox-360-controller-free-games-system.html

MTV Networks Announces Groundbreaking Programming Partnership With Microsoft for E3 Expo 2010. In: Microsoft [online]. New York: Microsoft, 2010 [cit. 2016-11-29]. Dostupné z: http://news.microsoft.com/2010/05/13/mtv-networks-announces-groundbreaking-programming-partnership-with-microsoft-for-e3-expo-2010/#sm.00014ekufvv5zezhy121j269njsbx#5eSJGxV7vMPCjpl8.97

TOULOUSE, STEPHEN. *What's in a name? Everything and nothing, depending*. In: Stepto [online]. Stephen Toulouse, 2010, Archivováno z http://stepto.com/Lists/Posts/Post.aspx?ID=620 [cit. 2016-11-29]. Dostupné z: https://web.archive.org/web/20100823065927/http://stepto.com/Lists/Posts/Post.aspx?ID=620

EISLER, CRAIG. *Starting February 1, 2012: Use the Power of Kinect for Windows to Change the World*. In: Microsoft [online]. Microsoft, 2012, Archived from http://stepto.com/Lists/Posts/Post.aspx?ID=620 [cit. 2016-11-29]. Dostupné z: https://blogs.msdn.microsoft.com/kinectforwindows/2012/01/09/starting-february-1-2012-use-the-power-of-kinect-for-windows-to-change-the-world/

ORLAND, KYLE. *Microsoft Announces Windows Kinect SDK For Spring Release*. In: Gamasutra [online]. Kyle Orland, 2011, Archived from http://stepto.com/Lists/Posts/Post.aspx?ID=620 [cit. 2016-11-29]. Dostupné z: http://www.gamasutra.com/view/news/33136/Microsoft_Announces_Windows_Kinect_SDK_For_Spring_Release.php

ORLAND, KYLE. *Microsoft: Kinect Hits 10 Million Units, 10 Million Games*. In: Gamasutra [online]. Kyle Orland, 2011, Archived from http://stepto.com/Lists/Posts/Post.aspx?ID=620 [cit. 2016-11-29]. Dostupné z: http://www.gamasutra.com/view/news/33136/Microsoft_Announces_Windows_Kinect_SDK_For_Spring_Release.php

CRAWFORD, STEPHANIE. *How Microsoft Kinect Works*. In: HowStuffWorks [online]. 2010 [cit. 2016-11-29]. Dostupné z: http://electronics.howstuffworks.com/microsoft-kinect2.htm

SEGGERS, RUBEN. *People Tracking in Outdoor Environments: Evaluating the Kinect 2 Performance in Different Lighting Conditions*. Amsterdam, 2015. Bakalářská práce. University of Amsterdam. Vedoucí práce R. Bakker.

JOHN. *Nearest Neighbor Image Scaling*. In: Tech-Algorithm [online]. 2007 [cit. 2016-11-29]. Dostupné z: http://tech-algorithm.com/articles/nearest-neighbor-image-scaling/

JOHN. *Bilinear Image Scaling*. In: Tech-Algorithm [online]. 2009 [cit. 2016-11-29]. Dostupné z: http://tech-algorithm.com/articles/bilinear-image-scaling/

JOHN. *Trilinear Interpolation Image Scaling*. In: Tech-Algorithm [online]. 2009 [cit. 2016-11-29]. Dostupné z: http://tech-algorithm.com/articles/trilinear-interpolation-image-scaling/

KEYS, R. *Cubic convolution interpolation for digital image processing*. IEEE Transactions on Acoustics, Speech, and Signal Processing [online]. 1981, 29(6), 1153-1160 [cit. 2016-11-29]. DOI: 10.1109/TASSP.1981.1163711. ISSN 0096-3518. Dostupné z: http://ieeexplore.ieee.org/document/1163711/

CMGLEE. *Bicubic interpolation*. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2007 [cit. 2016-11-29]. Dostupné z: https://en.wikipedia.org/wiki/Bicubic_interpolation

ALOKAHUTI. *Super Resolution*. In: SlideShare [online]. 2015 [cit. 2016-11-29]. Dostupné z: http://www.slideshare.net/alokahuti/super-resolution-46106868

BOUTON, DAVID. *Anti-Aliasing and Resampling Artwork - Part 2*. In: Graphics [online]. 2003 [cit. 2016-11-29]. Dostupné z: http://www.graphics.com/article-old/anti-aliasing-and-resampling-artwork-part-2

OPEN KINECT. *OpenKinect* [online]. 2012 [cit. 2016-11-30]. Dostupné z: https://openkinect.org/wiki/Main_Page

ARMSTRONG, ALEX. *OpenNI To Close.* In: I Programmer [online]. 2014 [cit. 2016-12-01]. Dostupné z: http://www.i-programmer.info/news/194-kinect/7004-openni-to-close-.html

LORIOT, YANNICK. *Kinect: How to install and use OpenNI on Windows – Part 1*. In: Yannick Loriot - Blog [online]. 2011 [cit. 2016-12-01]. Dostupné z: http://yannickloriot.com/2011/03/kinect-how-to-install-and-use-openni-on-windows-part-1/

Kinect for Windows Architecture. In: Microsoft [online]. 2014 [cit. 2016-12-01]. Dostupné z: https://msdn.microsoft.com/en-us/library/jj131023.aspx

Kinect for Windows SDK 2.0. In: Microsoft [online]. [cit. 2016-12-01]. Dostupné z: https://www.microsoft.com/en-us/download/details.aspx?id=44561

JONES, BRETT, LIOR SHAPIRA, RAJINDER SODHI, et al. *RoomAlive. Proceedings of the 27th annual ACM symposium on User interface software and technology* - UIST '14 [online]. New York, New York, USA: ACM Press, 2014, 637-644 [cit. 2016-12-01]. DOI: 10.1145/2642918.2647383. ISBN 9781450330695. Dostupné z: http://dl.acm.org/citation.cfm?doid=2642918.2647383

BENKO, HRVOJE, ANDREW D. WILSON, FEDERICO ZANNIER, et al. *Dyadic projected spatial augmented reality*. Proceedings of the 27th annual ACM symposium on User interface software and technology - UIST '14 [online]. New York, New York, USA: ACM Press, 2014, 645-655 [cit. 2016-12-01]. DOI: 10.1145/2642918.2647402. ISBN 9781450330695. Dostupné z: http://dl.acm.org/citation.cfm?doid=2642918.2647402

Kinect Fusion. In: Microsoft [online]. 2015 [cit. 2016-12-03]. Dostupné z: https://msdn.microsoft.com/en-us/library/hh855389.aspx

REED, SARAH, SHERRY HSI, OLIVER KREYLOS, M. YIKILMAZ, LOUISE KELLOGG, S. SCHLADOW, HEATHER SEGALE A LINDSAY CHAN. *Augmented Reality Turns a Sandbox into a Geoscience Lesson*. Eos [online]. 2016, 97, - [cit. 2016-12-03]. DOI: 10.1029/2016EO056135. ISSN 2324-9250. Dostupné z: https://eos.org/project-updates/augmented-reality-turns-a-sandbox-into-a-geoscience-lesson

CLARK, ADRIAN, THAMMATHIP PIUMSOMBOON, OLIVER KREYLOS, M. YIKILMAZ, LOUISE KELLOGG, S. SCHLADOW, HEATHER SEGALE A LINDSAY CHAN. *A realistic augmented reality racing game using a depth-sensing camera*. Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry - VRCAI '11 [online]. New York, New York, USA: ACM Press, 2011, 97, 499- [cit. 2016-12-03]. DOI: 10.1145/2087756.2087851. ISBN

9781450310604. ISSN 2324-9250. Dostupné z: http://dl.acm.org/citation.cfm?doid=2087756.2087851

SANFORD, KARL. *Smoothing Kinect Depth Frames in Real-Time*. In: CodeProject [online]. 2012 [cit. 2016-12-11]. Dostupné z: https://www.codeproject.com/articles/317974/kinectdepthsmoothing

Přílohy

A Vybrané zdrojové kódy

1. Filtrace pomocí rámců

```
private void FilterPixelByBounds(ushort[] pData, Dimension pDimension)
{
  if (this.smoothDepthArray == null)
  {
    this.smoothDepthArray = new ushort[pData.Length];
  }
  int widthBound = pDimension.Width - 1;
  int heightBound = pDimension.Height - 1;
  // paralelně projdeme data
  Parallel.For(0, pDimension.Height, depthArrayRowIndex =>
  {
    for (int depthArrayColumnIndex = 0;
    depthArrayColumnIndex < pDimension.Width; depthArrayColumnIndex++)
    {
      int depthIndex = depthArrayColumnIndex +
                       depthArrayRowIndex * pDimension.Width;
      // hledáme data, která kinect nedokázal vyhodnotit
      if (pData[depthIndex] == 0)
      {
      // raw data jsou uložena v jednorozměrném poli a indexy
      v bitmapě si musím dopočítat
        int x = depthIndex % pDimension.Width;
        int y = (depthIndex - x) / pDimension.Width;
        Dictionary<ushort, byte> filterCollection =
                    new Dictionary<ushort, byte>();
        int innerBandCount = 0;
        int outerBandCount = 0;
        // projdeme 5x5 okolí zkoumaného bodu
        for (int yi = -2; yi < 3; yi++)
        {
          for (int xi = -2; xi < 3; xi++)
          {
            // údaj na 0x0 je zkoumaný bod, proto ho nezahrneme
            if (xi != 0 || yi != 0)
            {
              // vypočtu absolutní pozice
              int xSearch = x + xi;
              int ySearch = y + yi;
              // omezím hranice v případě, že zpracovávám krajní body
              if (xSearch >= 0 && xSearch <= widthBound &&
                  ySearch >= 0 && ySearch <= heightBound)
              {
                int index = xSearch + ySearch * pDimension.Width;
```

```
                if (pData[index] != 0)
                {
                  for (int i = 0; i < 24; i++)
                  {
                    if (filterCollection.ContainsKey(pData[index]))
                    {
                      filterCollection[pData[index]] =
                      (byte)(filterCollection[pData[index]] + 1);
                    }
                    else
                    {
                      filterCollection.Add(pData[index], 0);
                    }
                  }
                  // uložíme, ve kterém bandu jsme nenulový
                  prvek nalezli
                  if (yi != 2 && yi != -2 && xi != 2 && xi != -2)
                    innerBandCount++;
                  else
                    outerBandCount++;
                }
              }
            }
          }
        }
        // pokud data splňují treshold omezení, tak provedu
        výpočet hodnoty
        if (innerBandCount >= this.innerBandThreshold ||
            outerBandCount >= this.outerBandThreshold)
        {
          float sum = 0f;
          ushort count = 0;
          foreach (KeyValuePair<ushort, byte> filter
                   in filterCollection)
          {
            sum += filter.Key * filter.Value;
            count += filter.Value;
          }
          this.smoothDepthArray[depthIndex] = (byte)(sum / count);
        }
      }
      else
      {
        this.smoothDepthArray[depthIndex] = pData[depthIndex];
      }
    }
  });
  pData = this.smoothDepthArray;
}
```

2. Algoritmus pro zprůměrování snímků

```
private ushort[] CreateAverageDepthArray(ushort[][] pDepthArray, Di-
mension pDim)
{
  int[] sumDepthArray = new int[pDim.Width * pDim.Height];
  ushort[] averagedDepthArray = new ushort[pDim.Width * pDim.Height];
  int denominator = 0;
  int count = 1;
  for (int i = 0; i < pDepthArray.Length; i++)
  {
    Parallel.For(0, pDim.Height, y =>
    {
      for (int x = 0; x < pDim.Width; x++)
      {
        int index = x + y * pDim.Width;
        int val = pDepthArray[i][index] * count;
        sumDepthArray[index] += val;
      }
    });
    denominator += count;
    count++;
  }
  Parallel.For(0, pDim.Height, y =>
  {
    for (int x = 0; x < pDim.Width; x++)
    {
      int index = x + y * pDim.Width;
      averagedDepthArray[index] = (ushort)(sumDepthArray[index] /
                                  denominator);
    }
  });
  return averagedDepthArray;
}
```

3. Bilineární změna velikosti

```
public static void BilinearScale (System.Object pObj)
{
  ThreadData threadData = (ThreadData) pObj;
  for (int y = threadData.start; y < threadData.end; y++)
  {
    int yFloor = (int)Mathf.Floor(y * ratioY);
    int y1 = yFloor * w;
    int y2 = (yFloor+1) * w;
    int yw = y * w2;
    for (int x = 0; x < w2; x++)
    {
      int xFloor = (int)Mathf.Floor(x * ratioX);
      float xLerp = x * ratioX - xFloor;
      newColors[yw + x] = ColorLerpUnclamped
                          (ColorLerpUnclamped(texColors[y1 + xFloor],
                          texColors[y1 + xFloor+1], xLerp),
                          ColorLerpUnclamped(texColors[y2 + xFloor],
                          texColors[y2 + xFloor+1], xLerp),
                          y*ratioY-yFloor);
    }
  }
  mutex.WaitOne();
  finishCount++;
  mutex.ReleaseMutex();
}
```

4. Algoritmus zhotovení sítě polygonů

```
private void CalculateMesh()
{
  int triangleIndex = 0;
  for (int y = 0; y < this.height; y++)
  {
    for (int x = 0; x < this.width; x++)
    {
      int index = y * this.width + x;
      this.verts[index] = new Vector3(x, -y, 0);
      this.uv[index] = new Vector2(x / (float)this.width,
                       y / (float)this.height);
      if (x != (this.width - 1) && y != (this.height - 1))
      {
        int topLeft = index;
        int topRight = topLeft + 1;
        int bottomLeft = topLeft + this.width;
        int bottomRight = bottomLeft + 1;
        this.trits[triangleIndex++] = topLeft;
        this.trits[triangleIndex++] = topRight;
        this.trits[triangleIndex++] = bottomLeft;
        this.trits[triangleIndex++] = bottomLeft;
        this.trits[triangleIndex++] = topRight;
        this.trits[triangleIndex++] = bottomRight;
      }
    }
  }
}
```

5. Algoritmus zjednodušení bitmapy

```
private void SimplyfyImage(Texture2D pTex)
{
  for (int y = 0; y < pTex.height; y += this.step)
  {
    for (int x = 0; x < pTex.width; x += this.step)
    {
      float sum = 0f;
      for (int y2 = 0; y2 < this.step; y2++)
      {
        for (int x2 = 0; x2 < this.step; x2++)
        {
          if(x + x2 > pTex.width || y + y2 > pTex.height) continue;
          sum += pTex.GetPixel(x + x2, y + y2).r;
        }
      }
      float val = sum / (this.step * this.step);
      Color col = new Color(val, val, val, 1f);
      for (int y2 = 0; y2 < this.step; y2++)
      {
        for (int x2 = 0; x2 < this.step; x2++)
        {
          if (x + x2 > pTex.width || y + y2 > pTex.height) continue;
          pTex.SetPixel(x + x2, y + y2, col);
        }
      }
    }
  }
}
```

B CD s výslednou aplikací a zdrojovými soubory