Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra kybernetiky

Diplomová práce

Vývoj rozhraní pro interaktivní distribuovanou hru s ovládáním pomocí 2D i 3D obrazových senzorů

Autor:
Bc. Jan Holeček

Vedoucí práce:
Doc. Ing. Miloš Železný Ph.D.

Plzeň, 2014

Prohlášení

Předkládám tímto k posouzení a obhajobě diplomovou práci zpracovanou na závěr studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím odborné literatury a pramenů, jejichž úplný seznam je její součástí.

V Plzni dne 16. května 2014

....
vlastnoruční podpis

Poděkování

Na tomto místě bych rád poděkoval především **Doc. Ing. Miloši Železnému Ph.D.** za vedení této diplomové práce a poskytnutí odborných konzultací. Dále potom **Ing. Miroslavu Jiříkovi** a **Ing. Tomáši Rybovi** za cenné odborné rady a pomoc při samotném vývoji rozhraní.

Také bych rád poděkoval **Ing. Jakubu Vítovi** za vytvoření prvního prototypu síťové komunikace, **Ing. Elišce Hamáčkové** a **Aleně Holečkové** za jazykovou úpravu a **Bc. Milanovi Jirků** za dlouhodobé zapůjčení zařízení Kinect. V neposlední řadě bych na tomto místě rád poděkoval svojí rodině, blízkým a přátelům za podporu při celém mém studiu.

Abstrakt

Tato diplomová práce se zabývá návrhem rozhraní určeného pro snadný vývoj interaktivních distribuovaných aplikací a her s využitím 2D a 3D obrazových senzorů. V úvodních částech je prezentován stručný teoretický úvod do distribuovaných systémů, technologií TCP/IP a zařízení Kinect. Následuje popis použitých technologií, díky kterým bylo možno navrhnout jednotlivé komponenty samotného rozhraní pro přenos dat mezi uživateli a pro zisk a zpracování obrazových informací ze zařízení Kinect. Stěžejní část práce obsahuje popis návrhu a funkčnosti těchto komponent i rozhraní jako celku. V práci je také prezentována interaktivní hra, pomocí které je výše zmíněné rozhraní testováno. Testy provedené na této aplikaci potvrdili použitelnost a funkčnost rozhraní v reálných podmínkách.

Klíčová slova

interaktivní hra, distribuované systémy, rozhraní, Enet, Kinect, zpracování obrazu, prostorová informace

Abstract

This thesis deals with a design of an interface earmarked for development of interactive distributed systems and games using 2D and 3D vision sensors. At the beginning a brief theoretical introduction to distributed systems, TCP/IP technology and Kinect device is presented, followed by a description of applied technologies which allowed to design individual components of the interface for data transmission between users and for obtaining and processing image information got from the Kinect device. The crucial part of the thesis contains a description of a proposal and functionality of these components and the interface as itself. In the thesis there is also presented an interactive game by which is the above-mentioned interface tested. Tests performed at this application have confirmed the availability and functionality of the interface in real terms.

Keywords

interactive game, distributed systems, interface, Enet, Kinect, image processing, spatial information

Obsah

1 Úvod **1**

2 Distribuované systémy **4**
2.1 Vývoj distribuovaných systémů 4
2.2 Centralizovaný topologický model 6
2.3 Decentralizovaný topologický model 7
2.4 Topologický model s částečnou centralizací 8

3 Architektura TCP/IP **10**
3.1 TCP 11
3.2 UDP 12

4 Zařízení Kinect **13**
4.1 Historie a vývoj 13
4.2 Funkce a principy 14

5 Použité technologie **18**
5.1 Python 18
5.2 PyKinect 19
5.3 PyGame 19
5.4 OpenCV 20
5.5 Enet (PyEnet) 20
5.6 Sockets 21

6 Rozhraní **22**
6.1 Výběr vhodného topologického modelu 22
6.2 Struktura rozhraní 23
6.3 Třídy Game a OurGame 25
6.4 Distribuovaní stavů mezi klienty a serverem 28
6.4.1 Třída StateMode 29
6.4.2 Třída State 30

6.4.3 Třída ClientServer 32
6.4.4 Třída GameServer 33
6.4.5 Třída GameClient 35
6.5 Sběr dat z obrazových senzorů 36
6.5.1 Třída KinectTrack 37
6.5.2 Třída Kamera 40
6.5.3 Pomocné metody pro práci s obrazovými daty 41
6.6 Přenos obrazové informace 43
6.6.1 Třída VideoServer 44
6.6.2 Třída ManagingClient 46
6.6.3 Třída VideoClient 47
6.7 Ostatní pomocné metody a spuštění programu 48
6.8 Zhodnocení rozhraní 50

7 Testovací aplikace **51**
7.1 Principy a mechanismy testovací hry Dots 51
7.2 Stavy hry 52
7.3 Výpočty na straně klientů 53
7.4 Výpočty na straně serveru 53
7.5 Kostra 55
7.6 Zpracování videa 55
7.7 GUI 58
7.8 Testování a zhodnocení hry Dots 59

8 Závěr **62**

A Struktura přiloženého CD **66**

Seznam obrázků

1.1 Zjednodušené schéma interaktivní komunikace pomocí obrazové a prostorové informace. 2

2.1 Model host-terminál. 5
2.2 Model klient-server. 6
2.3 Topologický model centralizovaného distribuovaného systému. 7
2.4 Topologický model decentralizovaného distribuovaného systému. 8
2.5 Topologický model distribuovaného systému s částečnou centralizací. 9

3.1 Vrstvy architektury TCP/IP. 10
3.2 Schéma zapouzdření dat v TCP/IP. 11

4.1 Umístění obrazových senzorů na zařízení Kinect. Podkladová fotografie byla převzata z www.microsoft.com. 14
4.2 Zobrazení sítě bodů v prostoru. Převzato z [Hoiem 2012] . . . 15
4.3 Ukázka hloubkové mapy pořízené zařízením Kinect. Převzato z [Hoiem 2012] 16
4.4 Ukázka nalezení jednotlivých částí těla na hloubkové mapě uživatele. Převzato z [Hoiem 2012] 16
4.5 Přehled sledovaných částí těla. 17
4.6 Rozfázovaný pohyb celé kostry. 17

6.1 Schéma architektury rozhraní. 24
6.2 Schéma tříd Game a OurGame. 25
6.3 Schéma bloku distribujicího stavy mezi serverem a klienty. . . 28
6.4 Schéma přenášení stavů mezi serverem a dvěma klienty. 29
6.5 Schéma bloku poskytujícího informace z obrazových senzorů. . 37
6.6 Schéma bloku přenášejícího video mezi klienty. 43
6.7 Schéma činnosti tříd VideoServer a ManagingClient. 44

7.1 Schéma zpracování obrazových dat odesílatelem a příjemcem. . 57
7.2 Ukázka použití filtrů 58

7.3 GUI s popisem jednotlivých prvků. 58
7.4 První testovací dvojice. 60
7.5 Druhá testovací dvojice. 61

Kapitola 1

Úvod

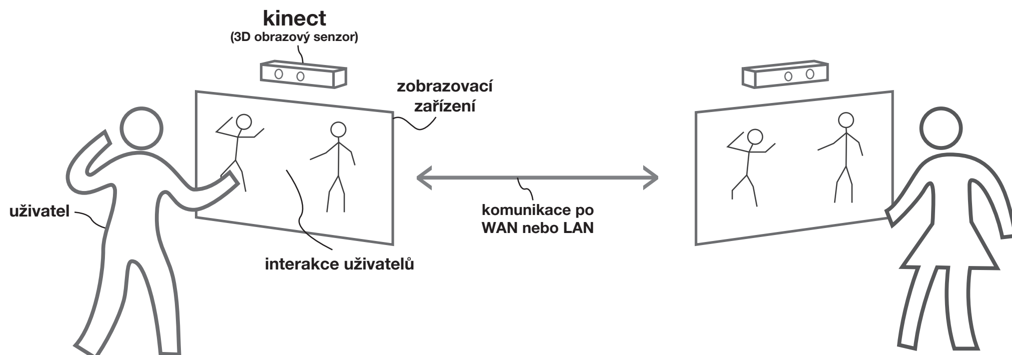
Spolu s rozvojem a zrychlením počítačových sítí (především světové sítě Internet) se rozvíjí i způsoby vzájemné komunikace vzdálených uživatelů. Díky nárůstu propustnosti sítí je možné posílat nejenom textové informace, ale i informace zvukové, obrazové, především ve formě videa, a další[1]. V současných systémech poskytujících instant messaging a VoiP telefonii (např. Skype nebo Google Hangouts) jde především o cílenou interakci uživatelů, kteří si vyměňují rychlé textové informace nebo komunikují pomocí zvuku a videa.

Projekt CityGate, jehož posláním je propojení českého města Plzeň a belgického města Mons (hlavní evropská města kultury pro rok 2015), se naopak zaměřuje na náhodnější komunikaci uživatelů, kteří se neznají a mezi kterými existuje určitá jazyková bariéra. Tato komunikace nemůže probíhat pomocí psaného nebo mluveného slova, a tedy musí být využity i jiné druhy informace, než které nám poskytuje klávesnice nebo mikrofon. Vhodnou neverbální informací může být informace obrazová nebo informace prostorová. Celkový charakter komunikace by se také podstatně lišil od komunikace klasické a bylo by nasnadě tuto komunikaci nazývat spíše interakcí.

Cílem této práce, která je součástí výše zmíněného projektu CityGate, je navrhnutí rozhraní umožňujícího rychlou implementaci distribuovaných aplikací, které dovolí vzdáleným uživatelům spolu interagovat pomocí obrazové a prostorové informace (viz obrázek 1.1). Ideálním nástrojem pro získání této informace by mohlo být zařízení Kinect od společnosti Microsoft, které poskytuje údaje o kostře až šesti uživatelů, hloubkovou mapu snímaného okolí a klasický záznam z kamery až do rozlišení 1280 na 1024 obrazových bodů.

[1] Myšleno z hlediska výstupu ze systému. Z hlediska charakteru informace v síti se vždy jedná o digitální informaci ve formě počítačových bitů

Typickými aplikacemi pro toto rozhraní mohou být především causal hry[2] nebo umělecké objekty, jejichž součástí je společná interakce náhodných a v prostoru vzdálených lidí.



Obrázek 1.1: Zjednodušené schéma interaktivní komunikace pomocí obrazové a prostorové informace.

Problém samotného návrhu lze kvalifikovat do několika samostatných kategorií. První z nich spadá do kategorie návrhu distribuovaných systémů, kde je potřeba zaručit rychlou a bezproblémovou komunikaci mezi uživateli. Druhou kategorií je zpracování obrazové a prostorové informace, kde největší problém představuje získání a případná příprava obrazové a prostorové informace pro odeslání ostatním uživatelům. Třetí kategorií je samotný návrh rozhraní, které by mělo být nejenom dostatečně modulární pro budoucí úpravy a rozšiřování funkčnosti, ale mělo by být především snadno přístupné a ovladatelné pro vývojáře, kteří budou toto rozhraní využívat.

Druhá kapitola této diplomové práce seznamuje čtenáře se základní teorií a topologií distribuovaných počítačových systémů. Kapitola třetí se zabývá komunikací jednotlivých entit distribuovaných systémů (protokol TCP/IP). Kapitola čtvrtá seznamuje čtenáře se zařízením Kinect, které obsahuje 2D a 3D obrazové senzory. Pátá kapitola poskytuje popis konkrétních použitých softwarových technologií: skriptovacího jazyka Python, knihovny PyKinect pro komunikaci se zařízením Kinect, rozhraní pro jednodušší implementaci

[2]Hry pro příležitostné hráče, které vyžadují žádnou nebo nepříliš velkou zkušenost s hraním počítačových her

her PyGame, open-source projektu pro zpracování obrazu OpenCV a v neposlední řadě technologie pro komunikaci PyEnet a Sockets. Následující šestá kapitola popisuje samotný návrh rozhraní pro interaktivní distribuované aplikace. Poslední sedmá kapitola představuje testovací aplikaci využívající toto rozhraní a zhodnocení dosažených výsledků.

Kapitola 2

Distribuované systémy

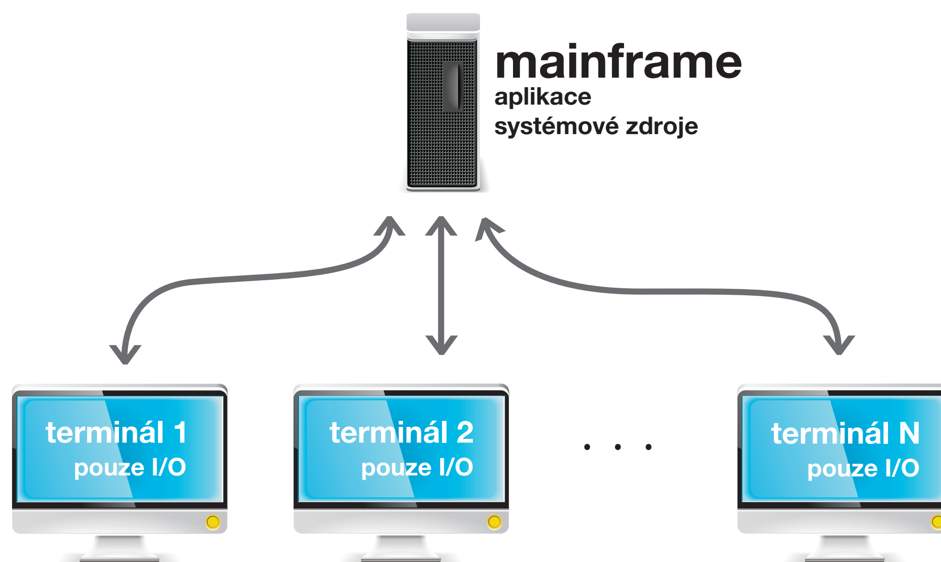
Andrew Tanenbaum definuje v [Tanenbaum 2007] distribuovaný systém:

> *A distributed system is a collection of independent computers*
> *that appear to its users as a single coherent system.*

Tento popis distribuovaného systému jako systému, který běží na oddělených strojích, ale uživateli se jeví jako jeden koherentní systém, je výstižný, ale také velmi široký, a proto je důležité zabývat se teorií distribuovaných systémů podrobněji. Nejprve bude nastíněna stručná historie těchto systémů a následně budou popsány nejběžnějších topologické modely, které charakterizují uspořádanost vazeb mezi entitami distribuovaných systémů.

2.1 Vývoj distribuovaných systémů

Počátek distribuovaných systémů se objevuje ve snaze maximálního využití nákladných výpočetních prostředků. Jedná se o dávkové zpracování, a především o systémy postavené na modelu host-terminál [Klimeš]. V modelu host-terminál vystupuje jeden (zpravidla nákladný) sálový nebo střediskový počítač (tzv. mainframe), který poskytuje aplikace a systémové zdroje (paměť, CPU apod.), a N terminálů, které těchto aplikací a systémových zdrojů využívají, jak je možné vidět na obrázku 2.1. Terminály slouží pouze k zadávání vstupů a k vykreslení výstupů. Tento systém byl výhodný díky snadnější údržbě, kdy bylo potřeba udržovat aplikace pouze v jednom fyzickém stroji.



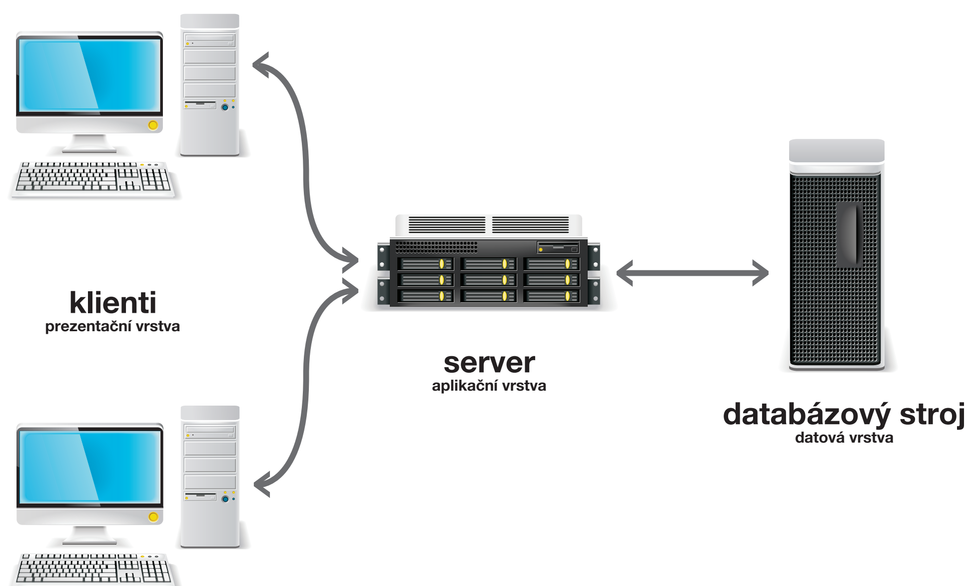
Obrázek 2.1: Model host-terminál.

Na konci 70. let, díky klesající ceně osobních počítačů, se od sdílení začalo upouštět a model host-terminál byl nahrazen modelem izolovaných počítačů [Klimeš]. V tomto modelu má každý uživatel svůj vlastní osobní počítač, který není nijak spojen s ostatními uživateli. Poměrně rychle se ukázala velká nevýhoda tohoto řešení, kdy byl uživatel odkázán pouze sám na sebe. I veškerá údržba musela být provedena na každém zařízení zvlášť. Bylo tedy nutné se určitým způsobem vrátit zpět a najít přijatelný kompromis mezi modelem host-terminál a modelem izolovaných počítačů [Klimeš].

Tento kompromis byl nalezen v modelu klient-server, kdy klientská část zajišťuje uživatelské rozhraní a serverová část se stará o zpracování dat a jejich uložení (tzv. dvouvrstvá architektura) [Klimeš]. Model má oproti modelu host-terminál výhodu ve velké úspoře přenášených dat, a může tedy dobře pracovat nejenom v sítích LAN[1], ale i v sítích WAN[2]. Dnes bývá dvouvrstvá architektura zpravidla nahrazována architekturou třívrstvou, kde dochází k oddělení aplikační vrstvy (zpracování dat) a datové vrstvy (uložení dat), viz obrázek 2.2.

[1]Local Area Network - počítačová síť pokrývající malé území, např. budovu firmy, školy nebo byt.

[2]Wide Area Network - počítačová síť pokrývající velké území, které může přesahovat hranice států. Nejznámější WAN sítí je celosvětová síť Internet.

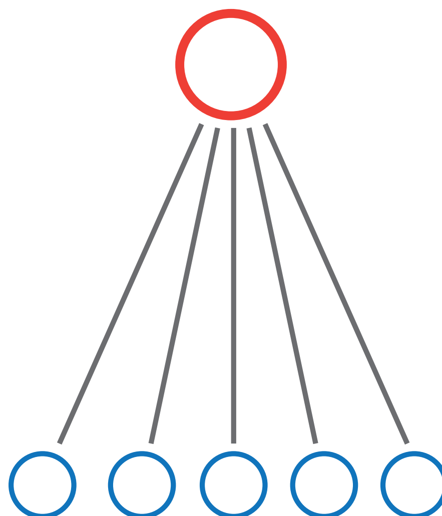


Obrázek 2.2: Model klient-server.

2.2 Centralizovaný topologický model

Model klient-server je typickým příkladem centralizovaného systému, tedy systému, kde se vyskytuje jediná nadřazená entita a ostatní entity jsou jí podřízené a komunikují pouze s touto nadřazenou entitou (serverem), viz obrázek 2.3. Tento topologický model vyžaduje jednodušší údržbu, je velmi koherentní[3], ale hůře toleruje havárie, protože v případě havárie serveru je jediná autorita ztracena a síť dále nemůže fungovat [Minar 2001].

[3]Informační koherence vyjadřuje, jakou mírou se může důvěřovat datům nalezených v síti [Minar 2001]



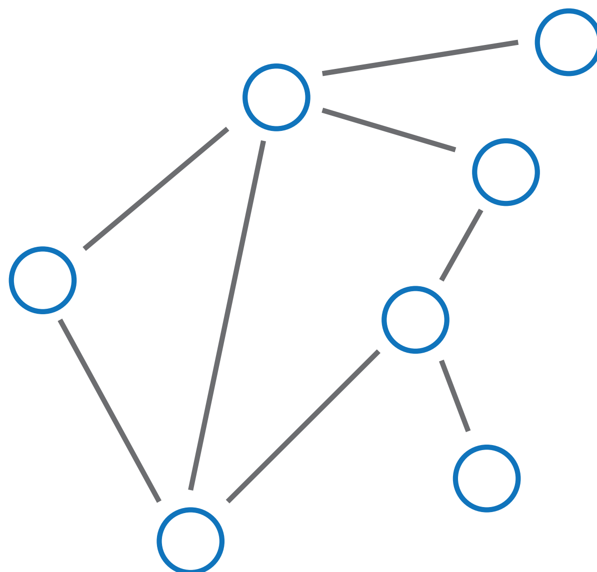
Obrázek 2.3: Topologický model centralizovaného distribuovaného systému.

Klasickým příkladem systému klient-server může být webový server připojený do sítě Internet, který poskytuje klientům, kteří se k němu připojili, informace z webových stránek, které jsou na něm umístěny[4].

2.3 Decentralizovaný topologický model

Určitým protipólem centralizovaných topologických modelů typu klient-server je decentralizovaný model typu peer-to-peer, kde v síti nevystupuje žádná nadřazená entita, ale všechny uzly sítě jsou si rovny (viz obrázek 2.4). Model disponuje většinou opačnými parametry než model centralizovaný. Je málo koherentní a velmi špatně se udržuje, ale disponuje jednoduchou rozšiřitelností a je velmi odolný k haváriím. Decentralizovanou síť je také těžké ovládnout, protože neobsahuje jednu nadřazenou entitu, a tedy neobsahuje jedno zranitelné místo [Minar 2001].

[4]Myšlen tento konkrétní jednoduchý příklad s jedním webovým serverem. Rozhodně nelze nazírat na světovou síť Internet jako na centralizovaný systém



Obrázek 2.4: Topologický model decentralizovaného distribuovaného systému.

Příkladem peer-to-peer sítě může být v současné době velmi populární síť virtuální měny Bitcoin, u které je tato decentralizace klíčová, jelikož je prakticky nemožné, aby jiná entita (například vláda nějaké země) síť ovládla. V tomto konkrétním příkladě by tato entita musela vlastnit minimálně polovinu výpočetního výkonu sítě[5].

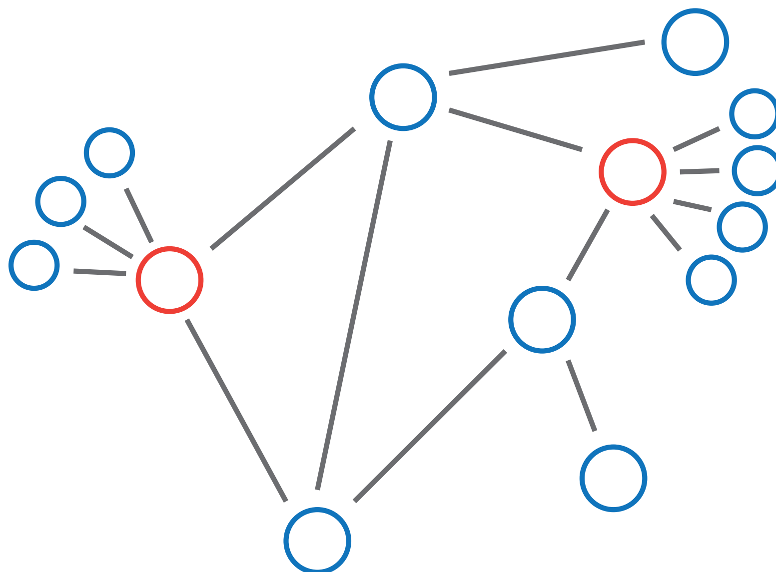
Je nutno zdůraznit, že čistokrevné peer-to-peer sítě nejsou příliš běžné a většinou určité nadřazené entity obsahují. Většinou slouží k navázání komunikace mezi jednotlivými uzly.

2.4 Topologický model s částečnou centralizací

Kombinací předchozích dvou modelů dostáváme decentralizovaný model, ve kterém se ale vyskytují entity, které jsou pro část modelu nadřazené, a

[5]Síť je založena na potvrzování transakcí ostatními uživateli sítě a vždy věří straně s větším výpočetním výkonem. Je vycházeno z předpokladu, že většina účastníků (tedy většina výpočetního výkonu) je slušná a nesourodá, a nebude tedy schválně stejným negativním způsobem ovlivňovat celou síť

tato část modelu se chová jako centralizovaný model (obrázek 2.5). Díky této vlastnosti si systém poměrně zachovává vlastnosti decentralizovaného modelu, ale díky centralizovaným částem jsou jeho data více koherentní [Minar 2001].



Obrázek 2.5: Topologický model distribuovaného systému s částečnou centralizací.

Příkladem takto postaveného systému může být dřívější architektura sítě Skype (VoiP telefonie, videohovory, instant messaging), kde někteří uživatelé, kteří disponují kvalitním připojením, rychlým počítačem a především veřejnou IP adresou, přebrali roli takzvaných supernodů[6]. Tyto super uzly poté v síti figurovaly jako nadřazené entity pro okolní počítače bez veřejné IP adresy [Wang 2005].

[6]Do této role se uživatel nedostával zcela vědomě a z vlastní vůle.

Kapitola 3

Architektura TCP/IP

V distribuovaných počítačových systémech hraje velkou roli transport dat mezi jednotlivými entitami. V celosvětové síti Internet je hlavním komunikačním protokolem (rodinou protokolů) TCP/IP neboli Transmission Control Protocol/Internet Protocol, jehož architektura se skládá ze 4 vrstev [Sochor 2013] (jak znázorňuje obrázek 3.1):

Obrázek 3.1: Vrstvy architektury TCP/IP.

aplikační vrstva Tato vrstva určuje, jakým způsobem (v jakém formátu) mají být data přenášená z (respektive do) koncových aplikací.

transportní vrstva Transportní vrstva zajišťuje spojení mezi aplikační vrstvou a síťovou vrstvou, kdy adresuje data konkretní aplikaci pomocí portů. Toto adresování probíhá pouze v rámci jednoho počítače [Dostálek 2000]. Toto spojení může být zajištěno protokolem UDP (viz podkapitola 3.2), nebo protokolem TCP (viz 3.1).

síťová vrstva (IP) Tato vrstva zajišťuje přenos mezi vzdálenými počítači pomocí protokolu IP (adresy IP). Data předává pomocí tzv. paketů[1], které se skládají z meta (řídících) a uživatelských dat.

vrstva síťového rozhraní Vrstva zajišťuje spojení s fyzickým přenosovým médiem. TCP/IP přímo nedefinuje protokol síťového rozhraní, ale pouze návaznost předchozí síťové vrstvy na služby vrstvy síťového rozhraní, které už používají své vlastní protokoly.

Aby mohla vyšší vrstva používat služby nižší vrstvy nebo nižší vrstva předat data vyšší vrstvě, musí být data z jednotlivých vrstev zapouzdřena. Zapouzdření probíhá od nejvyšší vrstvy k nejnižší, kdy tzv. protokolová datová jednotka (PDU[2]) je v nižší vrstvě opatřena příslušnými metadaty a je vytvořená nová protokolová datová jednotka, která je opět předána vrstvě nižší. V nejnižší vrstvě je odeslána příjemci, který ji opačným způsobem dopraví do vrstvy nejvyšší. Tento postup zapouzdření je naznačen na obrázku 3.2.

Obrázek 3.2: Schéma zapouzdření dat v TCP/IP.

Vrstva síťového rozhraní a síťová vrstva IP poskytují tzv. nespolehlivý přenos, kdy není zaručeno, že data od jednoho uživatele k druhému budou doručena nebo budou doručena ve správném pořadí. Spolehlivého přenosu je možné dosáhnout v transportní vrstvě pomocí protokolu TCP. V případě požadavku na nespolehlivý přenos je možné v transportní vrstvě využít protokol UDP.

3.1 TCP

Transportní protokol TCP (Transmission Control Protocol) transportní vrstvy architektury TCP/IP poskytuje spolehlivý datový přenos mezi dvěma počítači. Jedná se o protokol spojového charakteru, je tedy nutné před samotným přenosem dat ustanovit spojení a po ukončení přenosu opět spojení ukončit.

[1] Někdy označovaných také jako datagram.
[2] Protocol Data Unit

K navázání dochází pomoci tzv. handshakingu[3], který je v případě TCP třífázový [Sochor 2013].

Přenos dat probíhá pouze mezi dvěma body a většinou duplexně (obousměrně). TCP protokol rozděluje proud bajtů z aplikace na menší díly (pakety) podle velikosti MTU[4] a tyto pakety následně odesílá druhé straně, která jejich doručení musí potvrdit. Pokud odesílatel nedostane potvrzení v určité době[5], odešle data znovu. Příjemce dat přeuspořádá obdržené pakety podle správného pořadí a předá je aplikační vrstvě opět ve formě proudu bajtů. Aplikačním vrstvám se tedy jeví celý přenos pomoci TCP jako přenos proudu bajtů a nevnímají rozdělení dat na menší díly [Peterka].

Potvrzování přijatých paketů a jejich číslování má velkou výhodu v bezpečnosti přenosu, protože protokol zaručuje nejenom doručení, ale i správné pořadí. Nevýhodou je větší náročnost přenosu, protože jsou posílány i tyto redundatní údaje. V případě neobdržení jednoho paketu je celý přenos zdržen čekáním na tento paket, což může být při určitých aplikací bráno také jako nevýhoda.

3.2 UDP

Transportní protokol UDP neposkytuje na rozdíl od TCP garanci doručení. Díky této vlastnosti je celý protokol pouze jednoduchá nadstavba nad nižší síťovou IP vrstvou. Data jsou balena do IP-datagramů, které jsou díky charakteru přenosu mnohem jednodušší než pakety protokolu TCP [Sochor 2013].

Jedná se o nespojový způsob přenosu. Není tedy potřeba před samotnou komunikací vytvářet jakékoliv spojení a v případě potřeby může jakákoliv entita okamžitě komunikaci přerušit [Sochor 2013] (respektive přestat se komunikace účastnit). Také na rozdíl od TCP protokolu umožňuje broadcast (všesměrový) a multicast (více příjemců) přenos. Zjednodušeně řečeno, počítač odesílající data se vůbec nezajímá o to, zda ostatní počítače data přijaly a zda jsou ve správném pořadí.

Díky těmto vlastnostem je protokol UDP vhodný u aplikací, které nepotřebují všechna data, ale operují zpravidla pouze s těmi nejnovějšími.

[3] Z anglického výrazu pro potřesení rukou.

[4] Maximum Transmission Unit. Maximální velikost paketu přenesených po TCP/IP síti. Tato hodnota je většinou 1500 bajtů.

[5] Tato doba je statisticky vypočítávána z časů předchozích přenosů [Sochor 2013].

Kapitola 4

Zařízení Kinect

Zařízení Kinect od společnosti Microsoft slouží primárně pro ovládání her a aplikací pomocí pohybu lidského těla a hlasu. Existují dvě vývojové větve: Kinect 360, který je primárně určený pro platformu Xbox od stejné společnosti, a Kinect for Windows, který je určen pro vývoj aplikací na platformě Windows. Obě platformy jsou velmi podobné a liší se pouze v detailech a licenčních ujednáních.

4.1 Historie a vývoj

Zařízení (Kinect 360) bylo poprvé představeno v roce 2009 na veletrhu E3[1] pod kódovým označením Project Natal [Lowensohn 2011]. Technologie pro zpracování hloubky obrazu (včetně hardwaru) Microsoft licencoval od společnosti PrimeSense [MacCormick], která také vyráběla vlastní zařízení na detekci lidského pohybu. V listopadu roku 2010 je zařízení Kinect poprvé uvedeno do prodeje [Lowensohn 2011].

Velký zlom ve vývoji přichází na jaře roku 2011, kdy je uvolněn nekomerční SDK (Software Development Kit)[2] [Knies 2011], díky němuž má každý vývojář přístup ke všem hlavním funkcím. V současné době je tato SDK ve verzi 1.8. V listopadu 2013 byla představena nová verze snímače (jako součást herní konzole Xbox One) pod jménem Xbox One Kinect. Kinect for Windows ve verzi 2.0, který bude vycházet ze zařízení Xbox One Kinect, je plánován na léto 2014 [Kerkhove 2014].

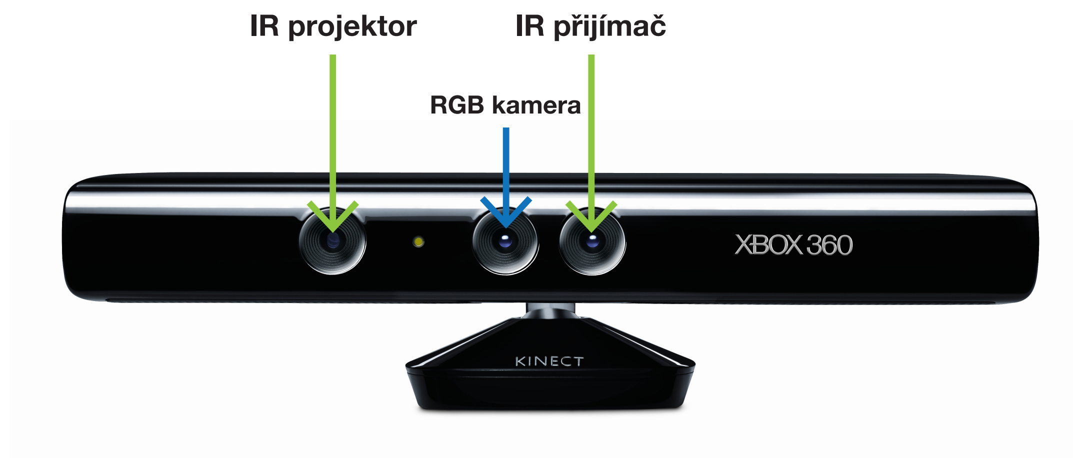
[1] Veletrh Electronic Entertainment Expo je každoročně pořádán v Los Angeles v USA.

[2] Komerční varianta spolu s vývojovou větví Kinect for Windows byla vypuštěna už v lednu téhož roku.

4.2 Funkce a principy

Vzhledem ke komerční povaze zařízení Kinect jsou detaily o přesné funkčnosti a některých specifikacích neveřejné [MacCormick] a část znalostí o fungování zařízení je odvozena komunitou zkoumáním patentů a reverzním inženýrstvím. Je tedy možné, že následující údaje se nemusí zcela přesně shodovat se skutečností.

Kinect se skládá z infračerveného (IR) projektoru, infračerveného přijímače, RGB senzoru a čtyř mikrofonů[3]. RGB senzor umožňuje záznam videa až do rozlišení 1280 na 1024 obrazových bodů[4] o snímkovací frekvenci 30Hz [Microsoft].



Obrázek 4.1: Umístění obrazových senzorů na zařízení Kinect. Podkladová fotografie byla převzata z www.microsoft.com.

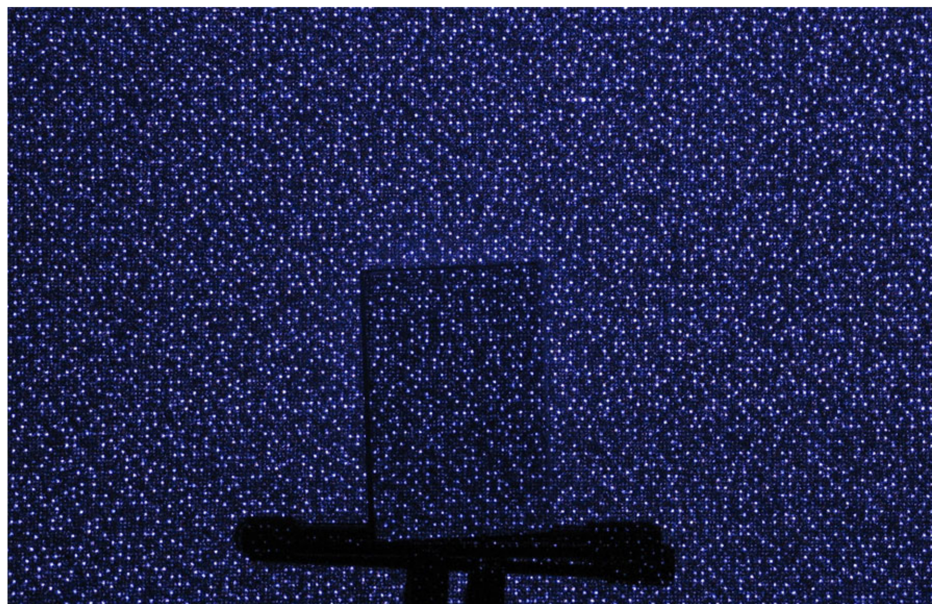
Další funkcí Kinectu je zjišťování hloubky prostředí, která tvoří základ pro sledování pohybů (kostry) uživatelů [Zhang 2012]. Ke zjištění hloubky prostoru jsou využity infračervený projektor a infračervený senzor [Zhang 2012] [MacCormick], které jsou od sebe vzdáleny necelé 3 centimetry (viz obrázek 4.1). Při klasickém zjišťování hloubky scény pomocí dvou od sebe vzdálených kamer poskytuje každá kamera lehce odlišný obraz. Na základě těchto odliš-

[3]Práce se zvukem není obsahem této práce a téma zpracování zvuku nebude dále zmíněno.

[4]Oficiální stránky společnosti Microsoft viz:http://msdn.microsoft.com/en-us/library/jj131033.aspx udávají jako maximální velikost obrazu 1280 na 960 obrazových bodů, ale knihovna PyKinect umožňuje výše zmíněných 1280 na 1024 obrazových bodů.

ností je možné vypočítat vzdálenost jednotlivých objektů scény (hloubku prostoru) [Zhang 2012]. Tento způsob ale vyžaduje správné spárování obou obrazů, což může být velmi složité u objektů, které nemají výrazné textury, mají opakující se textury, nebo pokud jsou snímky pořízeny v nepříznivých světelných podmínkách (například v šeru) [Hoiem 2012].

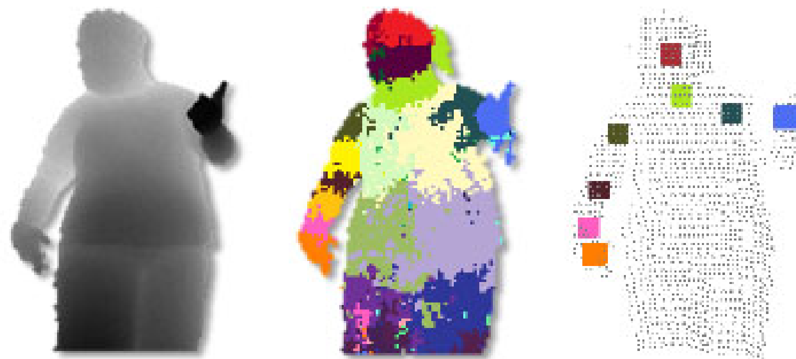
Zařízení Kinect tyto problémy obchází nahrazením jedné kamery infračerveným projektorem, který zobrazuje v prostoru síť bodů (viz obrázek 4.2), kde každý hlavní bod má okolo sebe množinu menších bodů, které jsou svým rozmístěním pro každý hlavní bod unikátní a je tedy možné určit, o který konkrétní bod se jedná a kde se nalézá [Zhang 2012]. Pomocí infračerveného snímače je tato síť zpětně (po promítnutí do prostoru) vyhodnocena a díky odlišnostem oproti původní síti (změna rozestupu mezi sousedními hlavními body apod) je možné vytvořit hloubkovou mapu (viz obrázek 4.3). Tento způsob poskytuje oproti tradičnímu způsobu dobré výsledky i při nevýrazných nebo opakujících se texturách nebo ve špatných světelných podmínkách [Hoiem 2012].



Obrázek 4.2: Zobrazení sítě bodů v prostoru. Převzato z [Hoiem 2012]

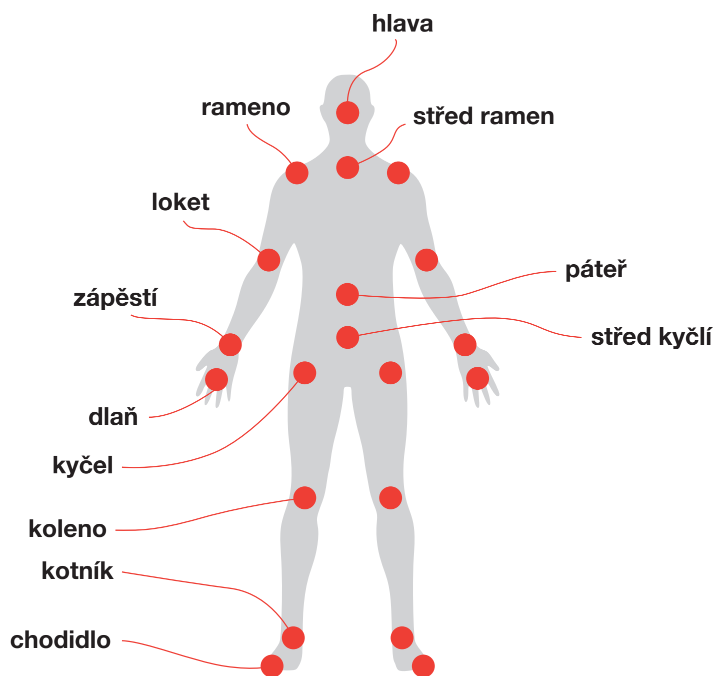
Obrázek 4.3: Ukázka hloubkové mapy pořizené zařízením Kinect. Převzato z [Hoiem 2012]

Takto vytvořená hloubková mapa umožňuje Kinectu nalézt ve scéně lidské postavy a estimovat pozice částí jejich těl v prostoru [Hoiem 2012] (obrázek 4.4). Celá problematika procesu estimace částí těl uživatelů je velmi rozsáhlá a přesahuje téma této práce. Přiblížení tohoto procesu je možné nalézt v [Hoiem 2012] nebo v [MacCormick].



Obrázek 4.4: Ukázka nalezení jednotlivých částí těla na hloubkové mapě uživatele. Převzato z [Hoiem 2012]

Kinect umožňuje sledování 20 bodů na těle uživatele. Tyto body představují především koncové části těla nebo místa ohybu (klouby). Přehled těchto bodů je vyobrazen na obrázku 4.5. Na obrázku 4.6 je potom ukázka rozfázovaného pohybu kostry, která vznikla logickým spojením těchto bodů.



Obrázek 4.5: Přehled sledovaných částí těla.

Obrázek 4.6: Rozfázovaný pohyb celé kostry.

Kapitola 5

Použité technologie

Rozhraní pro tvorbu interaktivních distribuovaných aplikací využívá celou řadu volně dostupných technologií. Tato kapitola se bude zabývat popisem nejdůležitějších z nich.

5.1 Python

Objektově orientovaný skriptovací[1] jazyk Python je ideální pro rychlý vývoj aplikací [Švec 2002]. Byl navržen v roce 1991 Guidem van Rossuem, je implementovaný pomocí jazyka C (někdy je tedy označován jako CPython) a v dnešní době je dostupný již ve verzi 3.4. Kvůli podpoře řady knihoven třetích stran, které nejsou s touto verzí kompatibilní, je stále velmi populární verze 2.7, která byla použita při implementaci rozhraní pro interaktivní distribuované aplikace, které je tématem této práce.

Syntaxe jazyka Python je oproti běžným kompilovaným jazykům (Java, C++ apod.) zjednodušena na minimum. Chybí tedy deklarování typu proměnných a argumentů funkcí (podobně jako v jazyku PHP) a pro seskupování výrazů je použito pouze odsazování (čímž je programátor nucen k jisté přehlednosti programu). Pro rozsáhlejší programy jsou ale stále k dispozici jmenné prostory, odchytávání výjimek apod. Jeho funkčnost je možné rozšířit pomoci jazyka C, ve kterém je napsaný [Švec 2002]

Jazyk Python tedy kombinuje jednoduchost skriptovacích jazyků a použitelnost jazyků kompilovaných. Díky těmto vlastnostem se může vývojář interkativní distribuované aplikace více soustředit na samotnou funkčnost

[1] Označení pro interpretované jazyky, které se vyznačují vysokou mírou expresivity a minimalistickou syntaxí

aplikace, a nikoliv na způsob, kterým bude tato funkčnost vytvořena.

Odkaz na stránky projektu: https://www.python.org/

5.2 PyKinect

Knihovna PyKinect (určená pro Python 2.7) poskytuje propojení mezi jazykem Python a SDK pro Kinect (viz podkapitola 4.2). Touto knihovnou je tedy zaručen poměrně bezproblémový přístup ke všem obrazovým funkcím zařízení Kinect přímo z Pythonu. Knihovna podporuje sledování kostry až šesti uživatelů najednou, hloubkovou mapu v rozlišení 320 na 240 obrazových bodů a video v maximálním rozlišení 1280 na 1024 obrazových bodů při 30 FPS. Knihovna velmi vyzdvihuje velkou propojenost s knihovnou PyGame (viz 5.3), ale toto propojení není nutnost a knihovnu PyKinect je možné využít pouze jako poskytovatel dat ze senzorů.

Výhodou knihovny je jednoduchý přístup ke všem potřebným obrazovým datům a sběr všech dat na oddělených vláknech. Nevýhodou je v současné době absence rozumné dokumentace a zaměření především na OS Windows, protože knihovna využívá přímo Kinect for Windows SDK. Alternativou mohou být knihovny OpenNI NITE 2, ale jejich vývoj je od odkoupení společnosti PrimeSense společností Apple velmi nejistý a v současné době není možné tyto knihovny získat z oficiálních zdrojů.

Odkaz na stránky projektu:
http://pytools.codeplex.com/wikipage?title=PyKinect

5.3 PyGame

PyGame je knihovna (přesněji sada knihoven), která je navržená především pro snadnou implementaci jednoduchých počítačových her v jazyku Python. Projekt vzniknul v roce 2000 jako pokračování zaniklého projektu PySDL [Shinners].

Rozhraní pro vývoj distribuovaných interaktivních her, jehož vývojem se zabývá tato práce, není přímo spojené s knihovnou PyGame, ale tato knihovna je využita při tvorbě testovací aplikace (viz kapitola 7) a díky své jednoduchosti je velmi vhodná pro vývojáře, kteří budou využívat toto rozhraní. Srovnatelnou alternativu pro tvorbu her můžeme najít například

v knihovnách XNA od společnosti Microsoft, které jsou ale určeny pro jazyk C#.

Odkaz na stránky projektu: http://www.pygame.org/news.html

5.4 OpenCV

OpenCV je open-source knihovna pro počítačové vidění[2], která umožňuje jednoduše implementovat i sofistikované systémy pro zpracování digitálního obrazu. Její základ je napsaný v jazycích C a C++ a je přístupná i pro jazyky C#, Python, Matlab a další [Bradski 2008].

Knihovna obsahuje nejenom nástroje pro filtraci obrazových dat, segmentaci a podobné základní úlohy zpracování obrazových dat, ale také sofistikovanější metody pro identifikaci předmětů, sledování pohybu, rozpoznávání obličejů. Knihovna patří k velmi populárním a počet stažení dosáhl více než 7 milionů (údaj z oficiálních stránek projektu ze dne 7. 5. 2014). Alternativou v jazyku Python by mohla být knihovna PIL (Python Image Library), která ale nedosahuje takového rozsahu jako knihovna OpenCV.

Odkaz na stránky projektu: http://opencv.org/

5.5 Enet (PyEnet)

Projekt Enet (názvem PyEnet je označována verze pro jazyk Python) vytváří nadstavbu nad protokolem UDP (viz 3.2), která byla dříve zamýšlena pouze jako součást hry pro více hráčů s názvem Cube. Protokol TCP byl pro tuto konkrétní potřebu málo vhodný, protože způsoboval větší latenci. Oproti tomu protokol UDP nezaručuje spolehlivý přenos dat.

Enet, jakožto nadstavba nad protokolem UDP, se snaží spojit výhody obou protokolů a vytvořit tedy protokol nový, který bude mít nízkou latenci, umožňovat broadcastové vysílání dat a také zaručovat spolehlivost dat, pokud bude vyžadována. Je umožněno posílat data s garancí doručení a správného pořadí (stejně jako u protokolu TCP), ale také pouze s garancí doručení, pouze s garancí pořadí nebo nespolehlivě jako u protokolu UDP.

[2] Písmena CV v názvu znamenají computer vision, v překladu počítačové vidění.

Odkaz na stránky projektu Enet: http://enet.bespin.org/

Odkaz na knihovnu PyEnet: https://code.google.com/p/pyenet/

5.6 Sockets

Sockets (někdy také pod názvem BSD Sockets) je nízkoúrovňová technologie, která zprostředkovává vzdálenou komunikaci mezi dvěma procesy, které mohou probíhat na různých počítačích. Komunikace může probíhat buď spolehlivým protokolem TCP, nebo nespolehlivým protokolem UDP. Komunikace má charakter server-klient.

V prostředí Python je technologie přístupná v modulu sockets, který je standardní součástí tohoto programovacího jazyka.

Kapitola 6

Rozhraní

S využitím výše zmíněného teoretického základu a znalostí o technologiích bylo vyvinuto rozhraní pro tvorbu interaktivních aplikací využívajících zařízení Kinect a webové kamery. Tato kapitola se zabývá popisem architektury tohoto rozhraní. U popisu metod jsou z důvodu přehlednosti v některých případech vynechány nedůležité vstupní parametry.

6.1 Výběr vhodného topologického modelu

Před samotným návrhem je nutné rozhodnout, který topologický model nejvíce vyhovuje nárokům pro distribuovanou interaktivní aplikaci. Je třeba brát zřetel na specifické vlastnosti budoucích aplikací:

- Aplikace bude obsahovat malý počet entit. Nejčastěji dva uživatele, kteří budou interagovat mezi sebou.
- Data by měla být přenášena co nejrychleji.
- Především pro hry by byla výhodná existence entity, která by poskytovala informační koherenci, tedy byla by v pozici určitého rozhodčího mezi interagujícími uživateli[1].

Malý počet entit vyřazuje z výběru topologický model s částečnou centralizací, protože takovýto distribuovaný systém by v našem případě nebylo možné ani sestavit. Decentralizovaný model (peer-to-peer) by zajišťoval nejkratší možné spojení mezi dvěma entitami, ale v případě více entit by objem

[1] Díky zpoždění vyvolanému nenulovou dobou přenosu dat mohou nastat situace, kdy se stejný stav hry jeví různým hráčům jiný, a je potřeba rozhodnout, jaký stav byl v daném okamžiku správný.

dat odeslaných každou entitou ostatním entitám velmi rychle rostl a každá entita by odesílala stejná data několikrát (všem ostatním entitám). Tento model by také neobsahoval žádnou nadřazenou entitu, která by mohla řídit celý průběh[2].

Pro potřeby interaktivní distribuované aplikace se jeví jako velmi výhodný model klient-server. Tento topologický model obsahuje nadřazenou entitu (server), která může zajišťovat informační koherenci a může obstarávat stavy aplikace, které musí být pro všechny klienty stejné. V případě dvou klientů (nejčastější případ) je nevýhodou dvojnásobná datová náročnost pro server, protože musí posílat informace všem klientům. V případě většího počtu klientů je náročnost zpravidla menší než u modelu peer-to-peer.

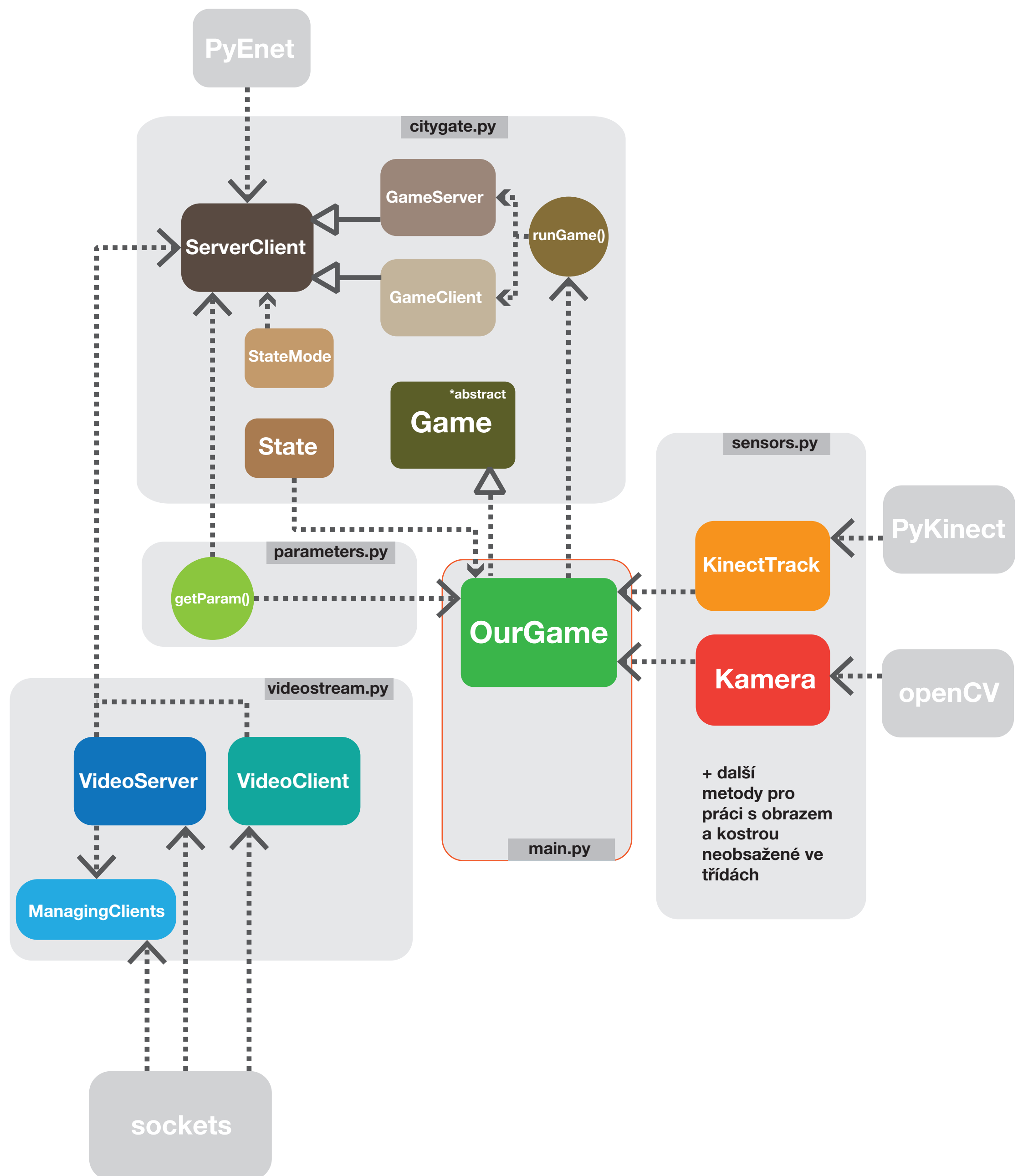
Po zvážení výše zmíněných vlastností různých topologických modelů bylo rozhodnuto použít model klient-server.

6.2 Struktura rozhraní

Struktura vytvořeného rozhraní se skládá z několika vzájemně propojených bloků, které tvoří samostatnější celky. Blok *citygate* (soubor citygate.py) se stará o distribuování stavů mezi klientem a serverem pomocí knihovny PyEnet. Tento blok byl vytvořen ve spolupráci s **Ing. Jakubem Vítem**, který vytvořil jeho první prototyp. Blok *videostream* (soubor videostream.py) se stará o výměnu videa mezi klienty pomoci knihovny Sockets a blok *sensors* (soubor sensors.py) se stará o sběr dat z obrazových snímačů. Dále je obsažen blok *parameters* (soubor parameters.py), který zajišťuje sběr parametrů při spuštění z příkazové řádky a v neposlední řadě blok *main* (soubor main.py), který propojuje samotnou uživatelskou aplikaci se zbytkem systému. Na obrázku 6.1 je vidět schéma celé architektury[3].

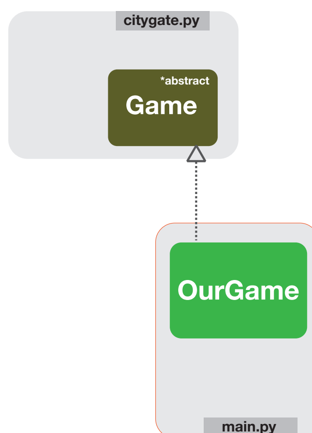
[2]Bylo by samozřejmě možné určit jednu entitu jako rozhodující, ale v takovémto případě by měla tato entita určitou výhodu nad ostatními. Také by se už nejednalo o plně decentralizovaný model.

[3]Jazyk Python je oproti ostatním běžným jazykům (Java, C++ a další) velmi volný a zdrojový kód může obsahovat metody, které nejsou v žádné třídě, třídy ve třídách a další, v jiných jazycích netypické konstrukce. Klasický UML diagram tříd by byl příliš nepřehledný a vhodnější bude tento diagram tříd, který z UML vychází, ale zcela se nedrží všech předepsaných pravidel.



Obrázek 6.1: Schéma architektury rozhraní.

6.3 Třídy Game a OurGame



Obrázek 6.2: Schéma tříd Game a OurGame.

Třída `OurGame` je realizace abstraktní třídy `Game` (viz schéma 6.2). Třída `Game` (respektive její realizace) je jedinou třídou, se kterou by měl uživatel (programátor aplikace) přijít do styku. Obsahuje následující metody:

createGameState(state) Slouží k vytváření serverových stavů (viz kapitola 6.4), které jsou jedinečné a rozhoduje o nich jedině a pouze server. Příklad definování serverového stavu:

```
state.addParam('time', 'float')
```

V tomto případě představuje `time` název proměnné a `float` definici datového typu.

createPlayerState(state) Metoda vytváří klientské stavy (viz kapitola 6.4), které jsou pro každého klienta rozdílné. Příklad definování serverového stavu:

```
state.addParam('score', 'int',True)
```

V tomto případě představuje `score` název proměnné, `int` definici datového typu a boolovská hodnota určuje, zda je stav vlastněn serverem, či nikoliv (viz 6.4.2).

updateGameState(state, playerState, client) Metoda slouží k aktualizaci serverových stavů definovaných pomoci funkce `createGameState(state)` nebo k aktualizaci klientských stavů vlastněných serverem. Příklad inkrementace serverového stavu:

```
state['numDots'] += 1
```

Většinou je potřeba, aby tuto proceduru prováděl pouze server a kód je potřeba opatřit podmínkou:

```
if client.isServer==True:
```

V případě, že je potřeba (například kvůli pomalé odezvě serveru) vypočítávat náhradní data offline[4], můžeme tento kód opatřit podmínkou opačnou:

```
if client.isServer==False:
```

Z čistě technického hlediska by bylo možné tuto část definovat i v jiné metodě (například v metodě `render()`), ale pro logickou přehlednost kódu je toto místo ideální. Pro občasné potřeby jsou metodě přístupné i klientské proměnné (`playerState`) a další informace (pomocí proměnné client).

updatePlayerState(playerId, state, gameState) Tato třída se chová podobně jako předchozí třída `updateGameState` s tím rozdílem, že nedochází k aktualizaci serverových stavů, ale stavů klientů, a to takových, které nejsou vlastněny serverem. Každý klient může aktualizovat pouze své stavy, a nemůže tedy ovlivňovat stavy jiných klientů. Tato procedura může být prováděna serverem, ale ve většině případů je potřeba, aby příslušný kód prováděli pouze klienti. Kód je tedy nutné opatřit podmínkou:

```
if not idPlayer==None:
```

[4]Například při letu míčku, kdy klienti obdrží pozici míčku pouze každou vteřinu

Pomocí vstupní proměnné `playerId` je také možné aktualizovat hráčské stavy pro každého hráče zvlášť[5]. Dále je k dispozici informace o serverových stavech.

serverIni() Tato metoda slouží serveru k inicializaci potřebných proměnných[6]. Je provedena pouze jednou na začátku, před hlavní smyčkou programu, a pouze serverem.

createGUI() Metoda velmi podobná předcházející metodě `serverIni`. Je také prováděna pouze jednou na začátku před hlavní smyčkou programu, ale pouze klienty. Jak už název napovídá, je vhodná především k inicializaci GUI neboli grafického uživatelského rozhraní.

render() Metoda `render` je vykonávána pouze klienty, a protože je prováděna při každém průchodu[7] hlavní smyčky programu, je vhodná pro renderování výstupu na zobrazovací zařízení uživatele. Je možné ji použít i k ostatním potřebným výpočtům klientů, ale logičtějším místem je metoda `updatePlayerState`.

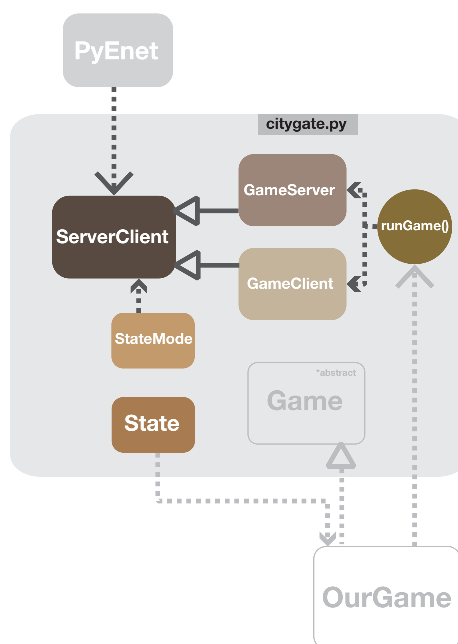
handleUserInput() Metoda slouží ke zpracování uživatelských vstupů. V současné podobě je tato metoda shodná s metodou `render`. Důvod tohoto rozdělení je nejenom v logickém oddělení, ale také v možnosti nastavit pro metody `render` a `handleUserInput` jinou periodu vykonávání v hlavní smyčce programu.

[5]Ve většině případů je výhodnější k tomuto účelu použít serverový stav, který řídí hru a může například rozhodovat, jaký hráč (klient) je útočník a jaký je v pozici obránce.

[6]Nikoliv však k inicializaci serverových stavů, které se inicializují v metodě `createGameState`

[7]Je možné uzpůsobit program tak, aby nebyla prováděna pokaždé, ale i s jinou periodou, například při každém druhém průchodu.

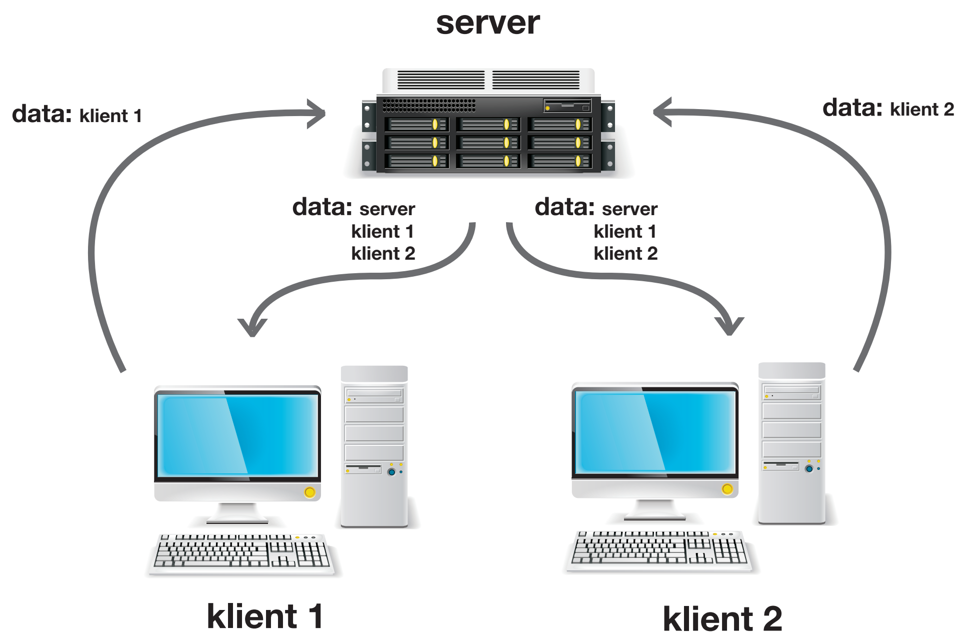
6.4 Distribuovaní stavů mezi klienty a serverem



Obrázek 6.3: Schéma bloku distribujicího stavy mezi serverem a klienty.

Výměna stavů mezi jednotlivými klienty a serverem je zprostředkována blokem citygate (soubor citygate.py), který využívá knihovnu PyEnet popsanou v podkapitole 5.5 (schéma na obrázku 6.3). Základní princip je v přenosu klientských dat (stavů) od klientů na server, který je broadcastovým způsobem pošle zpět klientům. Klient tedy dostává informace o serveru a o všech klientech, včetně sebe (viz obrázek 6.4). Zde je jistá nevýhoda v redundantnosti dat (klient obdrží i data, která sám poslal), ale tento způsob pomáhá jednodušší implementaci a přehlednosti.

Stavy v aplikaci mohou být dvojího typu. Prvním typem je serverový stav, který je pouze jeden a je plně vlastněn serverem. Tento typ je ideální pro jedinečné stavy, které jsou platné pro všechny klienty. Příkladem může být pozice míčku, který hráč přihrává druhému hráči. Pro oba hráče musí mít míček stejné souřadnice a ani jeden z klientů nemůže přímo měnit pozici míčku.



Obrázek 6.4: Schéma přenášení stavů mezi serverem a dvěma klienty.

Druhým typem je klientský stav, který je pro každého klienta unikátní (každý klient má svojí hodnotu tohoto stavu). Tento stav může být plně vlastněn klientem nebo plně vlastněn serverem. Pokud je stav vlastněn klienty, jsou stavy plně v kompetenci jednotlivých klientů. Případem využití jsou stavy, které má každý klient a jejichž hodnotu určují samotní klienti, například pozice ruky v prostoru jednotlivého hráče. Pokud je klientský stav vlastněn serverem, má každý klient také svojí hodnotu tohoto stavu, ale tato hodnota je plně řízená serverem. Dobrým příkladem použití může být skóre jednotlivých hráčů, kdy každý hráč má svůj jedinečný počet bodů, ale tyto body musí být plně v kompetenci nadřazené entity, tedy serveru.

6.4.1 Třída StateMode

Třída `StateMode` definuje tři módy, ve kterých se může (z pohledu jedné entity) stav nacházet. Tyto módy jsou:

- `SERVER`: Pro server se v tomto módu nacházejí všechny stavy, se kterými operuje.

- `CLIENT_LOCAL`: Pokud je pro klienta stav v tomto módu, jedná se o jeho stav. Tento stav může být tímto klientem vlastněn nebo může být vlastněn serverem.

- `CLIENT_REMOTE`: Stavy v tomto módu jsou pro klienta všechny stavy, které nejsou jeho. Může se jednat o serverové stavy nebo stavy jiných klientů (vlastněné serverem či nikoliv)

Zavedením těchto stavových módů se především dosáhne větší přehlednosti při rozlišování vlastnických práv u jednotlivých stavů.

6.4.2 Třída State

Třída `State` definuje strukturu, ve které jsou uchovávány stavy, a definuje také operace s touto strukturou. Stavy jsou uchovávány jako asociativní pole[8] (proměnná `values`), kde klíčem je název stavu a hodnotou hodnoty stavu. Protože je těchto hodnot stavu více, jsou tyto hodnoty uchovávány také jako asociativní pole, a tedy hodnotou v asociativním poli stavu je také asociativní pole. Příklad naplnění struktury konkrétního stavu:

```
self.values['time'] = {
        'value': 10.5,
        'type': 'float',
        'owner': True
    }
```

Zde je `time` název stavu, `value` udává hodnotu stavu, `type` je jeho datový typ a `owner` určuje, zda se jedná (v případě boolovského True) o stav vlastněný serverem (ať už serverový stav nebo klientský stav vlastněn serverem), nebo o (v případě boolovského False) klientský stav vlastněn klientem. Datový typ nemusí být pouze základní, ale je možné vložit i složitější datovou strukturu, například seznamy nebo tuple struktury, což je v našem případě výhodné pro posílání souřadnic o kostře.

Mezi hlavní metody třídy `State` patří:

addParam(name, type, ownedByServer, defaultValue) Tato metoda umožňuje vkládat nové druhy stavů. Podle módu a boolovské proměnné zadané při vytváření stavu se rozhodne, zda je stav vlastněn entitou, která ho vytváří viz kód:

[8] Asociativní pole (označované také jako slovník) tvoří neuspořádané dvojice klíč a hodnota. Klíč musí být v rámci tohoto asociativního pole jedinečný.

```
owner = False
if (ownedByServer and self.mode == StateMode.SERVER) or (not
     ownedByServer and self.mode == StateMode.CLIENT_LOCAL):
  owner = True
self.values[name] = {
  'value': defaultValue,
  'type': type,
  'owner': owner
  }
```

__setitem__(self, key, value) Klasická setovská metoda definuje, jakým způsobem bude konkrétnímu stavu (podle klíče `key`) přiřazena hodnota (`value`), viz následující kód:

```
if not key in self.values:
      print "Key does not exist"
      print self.values
    self.values[key]['value'] = value
    }
```

Přiřazena je tedy pouze samotná hodnota (`value`), nikoliv datový typ (`type`) a vlastník (`owner`). Programátor využívající třídu `Game` (viz 6.3) má tedy přehlednější a jednodušší práci, protože datový typ a vlastník se definují pouze na začátku a dále není potřeba tyto hodnoty měnit. Pokud není klíč obsažen, metoda skončí chybou.

__getitem__(self, key, value) Definování getovské metody umožňuje rychlejší přístup k hodnotě stavu, protože programátor využívající třídu `Game` (viz 6.3) nemusí ze stavu extrahovat hodnotu (`value`), ale tuto hodnotu pro něj představuje samotná proměnná stavu.

formatPacket() Metoda `formatPacket` připraví data na odeslání pomocí pythonovské knihovny Pickle. Tato knihovna umí serializovat struktury, což v našem případě znamená, že dokáže z asociativního pole, jehož hodnoty jsou další asociativní pole, vytvořit proud bytů, který se hodí k přenosu po síti. Z jednotlivých stavů jsou brány pouze hodnoty stavu, protože ostatní účastníci ví, jaká data budou přijímat.

updateFromPacket() Slouží k uložení nových hodnot stavů (do proměnné `values`) obdržených od serveru. Metoda obdržená data deserializuje, a reprezentuje tedy opačnou funkci než předcházející metoda

formatPacket. Metoda také kontroluje, zda mají klienti definované stejné stavy. V opačném případě vypíše chybu.

6.4.3 Třída ClientServer

Tato třída není používána přímo, ale slouží jako rodič pro třídy `GameServer` (viz 6.4.4) a `GameClient` (viz 6.4.5). Třída definuje dvě důležité proměnné:

- `gameState`
- `players`

V těchto proměnných si každá entita (server i klient) udržuje informaci o serverových stavech a stavech všech klientů (i svém vlastním). Metody sloužící k samotnému připojení uživatelů na server a posílání dat definuje jako prázdné (metody `onClientConnect`, `onClientDisconnect`, `onMessageRecieved` a `sendData`) a definuje pouze metodu `updatePlayers`, a především hlavní smyčku programu, která je obsažena v metodě `run`:

run() Metoda obsahuje hlavní smyčku programu, která je vykonávána serverem a každým klientem. Před samotnou smyčkou dojde k vytvoření serverových stavů (v případě serveru v módu `SERVER` a v případě klienta v módu `CLIENT_REMOTE`), vytvoření potřebných proměnných a inicializaci video přenosu (více k přenosu obrazu v kapitole 6.6):

```
if self.isServer:
    self.gameState = self.game.createGameState(StateMode.SERVER)
    #SERVER INI
        self.game.serverIni()
    #TCP VIDEO SERVER
        self.videoServer.start()

else:
    self.gameState = self.game.createGameState(StateMode.CLIENT_REMOTE)
    #GUI
        self.game.createGUI()
    #TCP VIDEO CLIENT
        self.videoClient.start()
```

Dále následuje hlavní while smyčka. Hned na začátku smyčky se zkontroluje, zda v PyEnetu nenastala nějaká událost (event). Odchytávány jsou tyto události:

- `EVENT_TYPE_CONNECT`: Tento typ události značí, že došlo k připojení klienta.
- `EVENT_TYPE_DISCONNECT`: Tento typ události značí, že došlo k odpojení klienta.
- `EVENT_TYPE_RECEIVE`: Tento typ události značí, že došlo k obdržení dat.

Kód pro odchytávání událostí vypadá následovně:

```
    while True:
        event = self.host.service(0)
        if event == None:
          break
        elif event.type == enet.EVENT_TYPE_CONNECT:
          self.onClientConnect(event.peer)
        elif event.type == enet.EVENT_TYPE_DISCONNECT:
          self.onClientDisconnect(event.peer)
        elif event.type == enet.EVENT_TYPE_RECEIVE:
          msg = pickle.loads(event.packet.data)
          self.onMessageRecieved(msg)
```

Následně (už mimo smyčku odchytávající události) dojde k zavolání metody `sendData`, která zajišťuje odeslání dat. Následují metody, které jsou vykonané pouze klienty. První je `handleUserInput` a `render` (viz 6.3) a metody pro posílání obrazu mezi klienty (viz 6.6). Kód této části:

```
if not self.isServer and self.clientId != None:
  self.game.handleUserInput(self.players[self.clien ...)
  self.game.render(self)
  #VIDEO IN
  if videoClient.videoIn.empty():
    videoClient.videoIn.put(self.game.videoIn())
  #VIDEO OUT
  if not videoClient.videoOut.empty():
    self.game.videoOut(videoClient.videoOut.get())
```

6.4.4 Třída GameServer

Třída `GameServer` dědí od třídy `ClientServer` a je používána pouze entitou, která je serverem. Předefinovává většinu tříd ze třídy `ClientServer` a definuje několik vlastních:

create() Metoda `create` vytváří na zadané IP adrese server, který poslouchá na zadaném portu. IP adresa a číslo portu se získávají z bloku parameters. Zdrojový kód pro vytvoření serveru:

```
self.host = enet.Host(enet.Address(ipAddress, port), 16, 0,
    0, 0)
```

Dále metoda vytváří server pro přenášení obrazu mezi klienty. Tento server používá stejnou IP adresu jako server pro stavy, ale poslouchá na jiném portu.

onClientConnect() Metoda při zachycení nového klienta vytvoří na serveru jeho stavy jako položku v seznamu `players`. Tento seznam uchovává stavy všech připojených hráčů.

onClientDisconnect() Tato metoda má opačnou funkci než metoda předchozí. Při odpojení klienta tedy vymaže jeho stavy z proměnné `players`.

sendData() Metoda zajišťuje broadcastové rozesílání stavů klientům. Rozesílaná zpráva je asociativní pole, které obsahuje název akce, id klienta a samotný stav. Akce značí, zda jde o aktualizaci hráčských stavů (`player_update`) nebo o aktualizaci stavů serverových (`game_update`). Celé pole je serializované pomocí výše zmíněného Pickle viz kód:

```
for i, s~in self.players.iteritems():
      msg = pickle.dumps({
          'action': 'player_update',
          'clientId': str(i),
          'state': s.formatPacket()
        });
      packet = enet.Packet(msg,3)
      self.host.broadcast(0, packet)
```

Dále je rozeslána zpráva o serverových stavech:

```
msg = pickle.dumps({
          'action': 'game_update',
          'time': self.localTime,
          'state': self.gameState.formatPacket()
    });
packet = enet.Packet(msg,3)
self.host.broadcast(0, packet)
```

onMessageRecieved() Metoda `onMessageRecieved` definuje, jakým způsobem má server naložit s daty, která obdrží od klientů. Pokud server obdrží zprávu s označením `client_update`, aktualizuje tímto stavem příslušného klienta ve svojí proměnné `players`, viz kód:

```
1 if msg['action'] == 'player_update':
      clientId = int(msg['clientId'])
3     self.players[clientId].updateFromPacket(msg['state'])
```

6.4.5 Třída GameClient

Třída `GameClient`, stejně jako třída `GameServer`, dědí od třídy `ClientServer`, ale je používána klienty. Definuje následující metody:

create() Metoda `create` vytváří spojení se serverem na zadané IP adrese a čísle portu, viz následující kód:

```
1 self.host = enet.Host(None, 32, 2, 0, 0)
      self.peer = self.host.connect(enet.Address(ipAddress,
    port), 1)
3     print "Game client connecting to " + ipAddress + ":" +
    str(port)
```

Dále vytváří spojení se serverem zajišťující obrazový přenos mezi klienty:

```
1 self.videoClient = videoClient(ipAddress)
```

onClientConnect() Metoda při připojení hráče k serveru vytvoří jeho klientské stavy a vloží je do seznamu `players` viz kód:

```
1 self.players[self.clientId] = self.game.createPlayerState(
    StateMode.CLIENT_LOCAL)
```

Stavy jsou vytvářeny v módu `CLIENT_LOCAL`, protože jsou to stavy tohoto konkrétního klienta.

sendData() Metoda `sendData` zajišťuje odesílání dat směrem k serveru:

```
s~= self.players[self.clientId]
msg = pickle.dumps({
  'action': 'player_update',
  'clientId': str(self.clientId),
  'state': s.formatPacket()
  });
packet = enet.Packet(msg,3)
self.peer.send(0, packet)
```

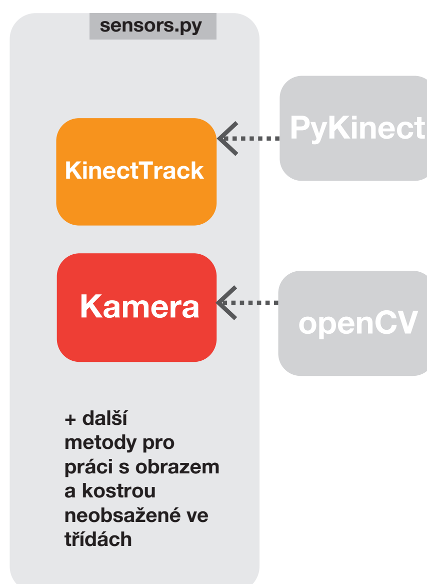
Je možné si povšimnout, že oproti serverové verzi metody nedochází k odeslání dat ostatním klientům.

onMessageRecieved() Tato metoda definuje zacházení s obdrženými daty. Pokud je název akce v obdržených datech `player_update`, jsou aktualizovány stavy příslušného hráče podle id. Pokud se id nenachází v seznamu `players`, dojde k vytvoření nového záznamu v `players` a vytvoření stavů nového hráče. Tyto stavy jsou v módu `CLIENT_REMOTE`, jelikož pro toho klienta nejde o jeho stavy. V případě akce `game_update` dojde k aktualizaci serverových stavů v proměnné `gameState`. Kód této části:

```
if msg['action'] == 'player_update':
  clientId = int(msg['clientId'])
  if not clientId in self.players:
    self.players[clientId] = self.game.createPlayerState(
   StateMode.CLIENT_REMOTE)
  self.players[clientId].updateFromPacket(msg['state'])
if msg['action'] == 'game_update':
  self.gameState.updateFromPacket(msg['state'])
```

6.5 Sběr dat z obrazových senzorů

Pro potřeby vývojářů aplikací v tomto rozhraní byly vyvinuté podpůrné prostředky pro práci s obrazovými senzory. V současné době blok sensors (soubor sensors.py) obsahuje třídu `KinectTrack` pro komplexní práci se zařízením Kinect, poskytující video, informace o kostrách uživatelů a hloubkové mapy, a třídu `Kamera`, poskytující základní nástroje pro práci s webkamerou. Dále obsahuje blok několik volných metod (nenacházejících se v žádné třídě), které umožňují zpracovávat obrazové informace obdržené od jiných klientů.



Obrázek 6.5: Schéma bloku poskytujícího informace z obrazových senzorů.

6.5.1 Třída KinectTrack

Třída `KinectTrack`, která využívá knihovnu PyKinect (viz 5.2), umožňuje vývojáři snadnější a pohodlnější přístup k informacím ze zařízení Kinect. Pomoci této třídy je možné získávat data o kostře uživatele[9], obrazový záznam z kamery až do rozlišení 1280 na 1024 obrazových bodů a hloubkovou mapu snímaného prostoru v rozlišení 320 na 240 obrazových bodů.

PyKinect umožňuje každou část (video, hloubka, kostra) spustit samostatně a na samostatném vlákně. Například v případě nevyužití videa a hloubkových map je tedy ušetřen výpočetní výkon a probíhá pouze sběr dat o kostře. Nejprve je nutné jednotlivé části inicializovat. Inicializace videa:

```
def startVideo(self, width=640, height=480):
    self.kinect.video_frame_ready += self.video_frame_ready
    if width==1280:
        res = nui.ImageResolution.Resolution1280x1024
    else:
        res = nui.ImageResolution.Resolution640x480
        self.kinect.video_stream.open(nui.ImageStreamType.Video, 2,
    res, nui.ImageType.Color)
```

[9]V současné době umožňuje třída získat informaci o jednom uživateli. Samotný Kinect a knihovna PyKinect umožňují sledovat až šest uživatelů.

Inicializace hloubkové mapy:

```
def startDepth(self):
        self.kinect.depth_frame_ready += self.depth_frame_ready
        self.kinect.depth_stream.open(nui.ImageStreamType.Depth,
    2, nui.ImageResolution.Resolution320x240, nui.ImageType.Depth
    )
```

Inicializace kostry:

```
def startSkeleton(self):
        self.kinect.skeleton_engine.enabled = True
        self.kinect.skeleton_frame_ready += self.
    skelet_post_frame
```

Dále je nutné popsat metody `video_frame_ready`, `depth_frame_ready` a `skelet_frame_ready`, které definují, jak se má s jednotlivým snímkem videa, hloubkové mapy nebo kostry naložit dále. Každý snímek je vložen do svojí fronty (struktura ze třídy `Queue`), a to pouze v případě, že je fronta prázdná. Předejde se tak zbytečnému plnění fronty, kterou nikdo nevyužívá. Z těchto front už je možné jednotlivé snímky odebírat přímo do třídy `Game`. V případě kostry je metoda `skelet_frame_ready` definovaná velice prostě:

```
if self.skeletonQ.empty():
    skelet = self.getSkeleton(frame)
    self.skeletonQ.put(skelet)
```

Zde je nejprve kostra upravena metodou `getSkeleton`, která je popsána níže. V případě videa a hloubkové mapy je výhodné (pro pozdější použití) převést snímek do formy, která je vhodná pro zpracování pomocí OpenCV. V případě videa:

```
video = numpy.empty((hight, width,4), numpy.uint8)
        frame.image.copy_bits(video.ctypes.data)
        im = cv2.cvtColor(video, cv2.COLOR_BGRA2BGR)
        if self.videoQ.empty():
            self.videoQ.put(im)
```

Zde stojí za povšimnutí, že numpy struktura (kterou je reprezentován obraz v OpenCV) má prohozené řádky a sloupce. Podobný převod je i u hloubkové mapy:

```
depth = numpy.empty((240,320,1), numpy.uint16)
        frame.image.copy_bits(depth.ctypes.data)
        im = cv2.cvtColor(depth, cv2.COLOR_BGRA2BGR)
        if self.depthQ.empty():
            self.depthQ.put(Image.fromarray(im))
```

Pomocí třídy `KinectTrack` by bylo možné obdržet data o kostře přesně tak, jak je poskytuje PyKinect, ale tato forma není příliš přehledná a pro účely zjednodušení návrhu interaktivních aplikací byly vyvinuty metody pro snadnější uchopitelnost dat z kostry.

Metody pro práci s kostrou:

findPlayer () Knihovna PyKinect bohužel nově nalezenou kostru přiřazuje náhodně do seznamu šesti možných koster. Tato metoda vrací index, na kterém se v tomto seznamu kostra nalézá. Pokud je v seznamu koster více, je vracen jen index kostry, která se objevila jako první. Pokud není nalezena žádná kostra, je vraceno číslo -1.

getSkeleton(frame,scaled=True) Tato metoda poskytuje celou kostru (skeleton) jednoho hráče[10], a to ve formátu seznamu. Struktura tohoto seznamu spočívá ve střídání názvů kostí (části skeletonu) a příslušných hodnot jejich pozic, viz příklad:

```
...
'Head',
[102.3, 20.3, 135],
'ShoulderLeft',
[57.2, 51.3, 80],
...
```

Metoda nejprve nalezne kostru uživatele pomocí metody `findPlayer` a následně vytvoří seznam celé kostry podle výše zmíněné struktury. Pozice jednotlivých kostí je možné získat neškálovaně (`scaled=False`), kdy jsou hodnoty pozic takové, jak je reprezentuje sám Kinect, nebo škálovaně (`scaled=True`), kdy jsou hodnoty pozic škálované například podle velikosti okna, ve kterém

[10] V současné době třída `KinectTrack` umožňuje sledovat pouze jednu kostru v jeden okamžik, což odpovídá nejčastějšímu použití. Pro podporu více koster (kterou Kinect umožňuje) by bylo nutné metodu upravit.

se kostra uživateli zobrazuje. Škálování probíhá pomocí funkce `nui.SkeletonEngine.skeleton_to_depth_image`, jejímž vstupem jsou pozice a velikost (výška a šířka) plochy, na kterou se má kostra škálovat v obrazových bodech[11].

findJoint(self,jointId,scaled=True) Tato metoda (narozdíl od metody `getSkeleton`) vrací pozice jednotlivých kostí, nikoliv celé kostry. Je obzvláště výhodná pro aplikace, které se ovládají pouze malým počtem kostí (typicky pouze dlaně), a není tedy potřeba získávat celou kostru. Metoda vrací pozice neškálovaně nebo škálovaně.

skeletonToString(skeleton) Metoda převádí strukturu celé kostry (poskytnuté metodou `getSkeleton`) do jednoho řetězce. Tento řetězec je výhodný k posílání celé kostry dalším uživatelům[12]. K oddělení jednotlivých částí slouží oddělovače:'/', '|' a ',' viz příklad:

```
...|Head/102.3,20.3,135||ShoulderLeft/57.2, 51...
```

Opačná metoda převádějící řetězec na strukturu kostry se nenachází ve třídě `KinectTrack`, protože uživatel, který data přijímá, nemusí mít vytvořenou instanci třídy `KinectTrack`, která je úzce spojena s fyzickým připojením zařízení Kinect. Tato metoda je popsána v 6.5.3.

isSkeleton(skeleton) Metoda vrací boolovskou hodnotu, zda Kinect sleduje nějakou kostru, nebo nikoliv.

destroy() Tato metoda ukončuje vlákno, které se stará o sledování kostry.

6.5.2 Třída Kamera

Třída `Kamera` poskytuje vývojáři základní práci s webovou kamerou. Tato třída obsahuje pouze dvě metody, které využívají prostředky pro práci s kamerou poskytnuté knihovnou OpenCV:

loadKamera(source=0) Tato metoda inicializuje webovou kameru pomocí knihovny OpenCV a uloží ji do své proměnné `kamera`:

```
1  self.kamera = cv2.VideoCapture(source)
```

[11]Škálované je pouze informace o ose X a Y. Informace o ose Z zůstává stejná.

[12]Kostru je samozřejmě možné poslat i jako jednotlivé integery (případně jako float) nebo jako strukturu tuple.

Pokud je `source` nastaveno na 0, je inicializována integrovaná kamera. V případě nastavení na 1 je inicializována kamera externí.

getImage(source=0) Metoda navrací aktuální snímek z kamery a vkládá ho do numpy struktury, která je vhodná pro OpenCV:

```
ret, im = self.kamera.read()
im = numpy.array(im)
im = cv2.cvtColor(im, cv2.COLOR_RGB2BGR)
return im
```

6.5.3 Pomocné metody pro práci s obrazovými daty

Velká část metod pro práci s obrazovými senzory a daty se přímo neváže na instance tříd `KinectTrack` a `Kamera`, protože je důležité, aby mohly být využívány všemi (především klienty), bez rozdílu připojeného hardwaru. Popis metod:

stringToSkeleton(string) Tato metoda převede řetězec (nejčastěji obdržený od druhého klienta) na strukturu popisující kostru (tato struktura je popsaná v 6.5.1).

saveJpegToMemory(img,com=0,string=0) Z důvodu datového objemu obrazových dat přenášených v systému (přenos je blíže popsaný v 6.6) je výhodné přenášený obraz komprimovat. Pro tuto kompresi je možné použít kompresní metodu JPEG, kterou umí využít knihovny OpenCV nebo PIL (Python Image Library). Bohužel tyto knihovny umožňují použít JPEG kompresi pouze při ukládání na disk, což není z důvodu malé rychlosti zápisu a čtení z pevného disku v reálné situaci dobře použitelné. Je ale možné využít pythonovskou knihovnu StringIO, která umožňuje zapsat tato data nikoliv na pevný disk, ale přímo do paměti. Tento proces provádí následující kód:

```
img = Image.fromarray(img)
imgJpeg = StringIO.StringIO()
img.save(imgJpeg, "JPEG", quality=com)
imgJpeg.seek(0)
```

Komprese obrazových dat je prováděna pomocí knihovny PIL. Proměnná `quality` značí kvalitu komprese, kdy 100 je nejlepší a datově nejnáročnější a 0 nejhorší a datově nejméně náročná kvalita. Následně

jsou vrácena obrazová data uložená v paměti nebo přímo převedená na řetězec, který se hodí k přenosu mezi klienty:

```
if string == 0:
    return imgJpeg
elif string == 1:
    imgString = imgJpeg.getvalue()
    return imgString
```

makeJpegFromMemory(string) Metoda vytváří v paměti ze vstupního řetězce obraz ve formátu JPEG pomoci knihovny StringIO:

```
imgJpeg = StringIO.StringIO()
imgJpeg.write(string)
imgJpeg.seek(0)
return imgJpeg
```

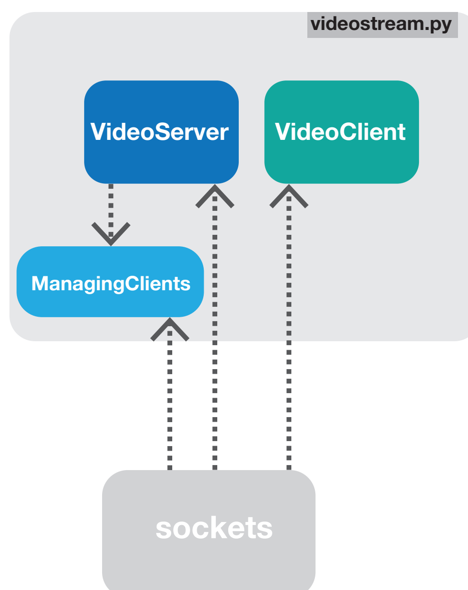
Tato metoda tedy umožňuje z řetězce obdrženého od jiného klienta opět vytvořit obraz v paměti.

makeCvImageFromMemory(string) Metoda je funkčností podobná předchozí, ale z obdrženého řetězce vytváří obraz v numpy struktuře, který umí využít knihovna OpenCV:

```
nparr = numpy.fromstring(string, numpy.uint8)
img_np = cv2.imdecode(nparr, cv2.CV_LOAD_IMAGE_COLOR)
return img_np
```

Díky převodu přímo do numpy struktury se lépe využijí metody zpracování obrazu, které poskytuje knihovna OpenCV.

6.6 Přenos obrazové informace



Obrázek 6.6: Schéma bloku přenášejícího video mezi klienty.

Bylo by logické použít výše zmíněný způsob přenosu informací mezi entitami (viz podkapitola 6.4) pro všechny druhy informace, tedy i pro informaci obrazovou[13]. Obrazová informace má ale oproti informaci o stavech jiný charakter. Je především datově mnohem náročnější než informace o stavech[14] a není u ní zpravidla potřeba nadřazená entita. Zjednodušeně řečeno, je nutné pouze dopravit obrazovou informaci mezi klienty[15] a zaručit, aby každý uživatel obdržel od druhého uživatele co nejnovější data.

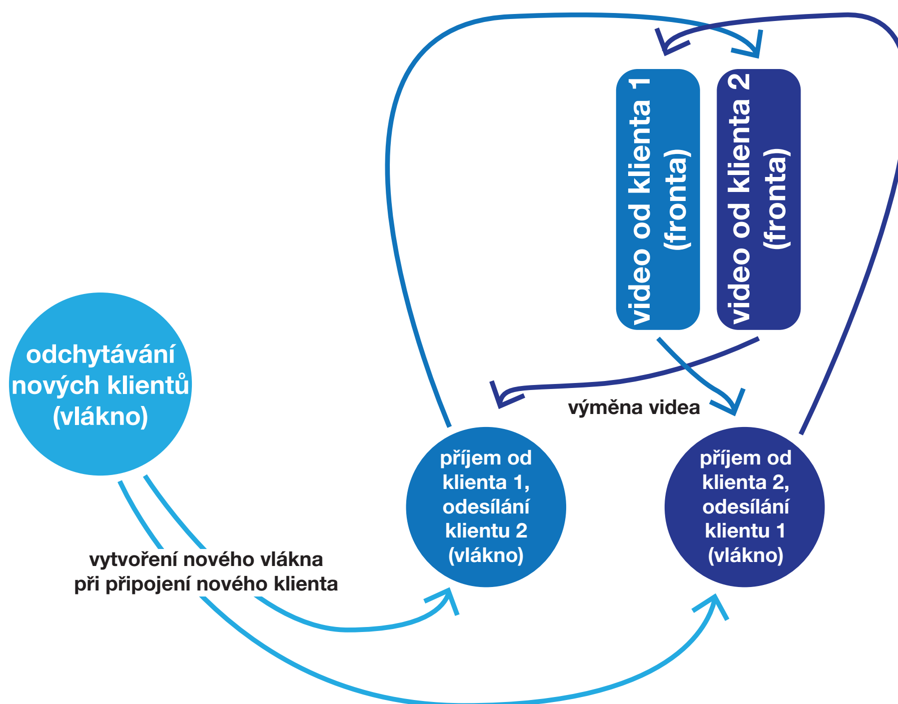
Při využití způsobu, kterým se přenášejí informace o stavech, se objevuje nevýhoda broadcastového systému, kdy jsou všechna data posílána všem. U stavových, méně objemově náročných dat to nepředstavuje problém, a naopak to jistou mírou pomáhá ke snadnější implementaci a přehlednosti dat.

[13]Myšleno video, nikoliv například data o kostře, která se z určitého úhlu pohledu dají také považovat za data obrazová.
[14]Při běžných aplikacích.
[15]V současné době je v rozhraní umožněný přenos obrazové informace pouze mezi dvěma klienty a v dalším textu bude tedy diskutována pouze tato možnost

U obrazové informace je většinou objem dat řádově větší a bylo by zbytečné takto objemná data posílat všem účastníkům.

Z těchto důvodů bylo tedy rozhodnuto vytvořit systém na distribuci obrazové informace nezávisle na systému distribuce stavových dat (viz schéma 6.6). Tento systém stále obsahuje nadřazenou entitu (server), ale pouze z důvodu, že podřazené entity (klienti) nemusí být v síti přímo viditelné a je potřeba prostředník, který zařídí výměnu obrazových dat mezi klienty. Nevýhodou zapojení serveru při přenosu obrazové informace je větší objem dat přijímaných a odesílaných serverem. Měl by tedy mít lepší konetivitu (rychlost připojení) než klienti. Oproti distribuci stavů (knihovna PyEnet) je tento systém postaven na technologii Sockets popsané v podkapitole 5.6.

6.6.1 Třída VideoServer



Obrázek 6.7: Schéma činnosti tříd VideoServer a ManagingClient.

Třída `VideoServer` se stará o připojení jednotlivých klientů a výměnu obrazových dat mezi nimi, jak naznačuje schéma 6.7. Nejprve je nutné vytvořit pomocí Sockets TCP server, který bude přijímat klienty na určité IP adrese a určitém portu. IP adresa je shodná s IP adresou stavového serveru, ale číslo portu je odlišné. Kód pro vytvoření TCP serveru:

```
self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
self.sock.bind((host, port))
self.sock.listen(2)
```

Parametr `socket.SOCK_STREAM` v `socket.socket` značí TCP server a parametr listen udává maximální počet klientů, kteří se mohou připojit na server. Kromě inicializace obsahuje třída `VideoServer` dvě metody:

run() Metoda `run` je spouštěna příkazem:

```
self.server.start()
```

Tento příkaz je vykonán ve třídě `ClientServer` (blíže popsané v 6.4.3) a metoda `run` je spuštěna v novém vlákně, protože je nutné oddělit tuto smyčku od hlavní smyčky programu ve třídě `ClientServer`. Na vině je čekání na připojení nových klientů, které by jinak blokovalo vykonávání hlavní smyčky programu:

```
self.conn, self.addr = self.sock.accept()
```

Po úspěšném připojení klienta je opět v novém vlákně vytvořena nová instance třídy `ManagingClient`:

```
thread = managingClient(self.conn, self.idClient)
thread.daemon = True
thread.start()
```

Nová vlákna jsou spuštěna v módu daemon. Tento mód umožní automatické ukončení vláken při ukončení činnosti hlavního programu.

stop() Tato metoda ukončí činnost serveru.

6.6.2 Třída ManagingClient

Tato třída má na starosti příjem obrazových dat (na straně serveru) od jednotlivých klientů a odesílání dat klientům opačným. Nová instance této třídy je spuštěna z třídy `VideoServer` jako samostatné vlákno pro každého nově připojeného klienta (vláken je tedy stejný počet jako připojených klientů). Zde vzniká problém výměny obrazových dat mezi jednotlivými vlákny. K tomuto účelu se velmi hodí pythonovská knihovna Queue, která umožňuje konstrukce front, které zabraňují nebezpečným operacím jako zápis dvou vláken na jedno místo v paměti v jeden okamžik. Pokud jedno vlákno vkládá data do fronty, je tato fronta uzamčena a jiné vlákno nemá možnost s frontou manipulovat[16]. Třída `ManagingClient` obsahuje (kromě inicializační metody) jedinou metodu:

run() Metoda `run` má na starosti příjem obrazových dat (ve formě řetězce), jejich vkládání do příslušné fronty podle ID klienta a odesílání obrazových dat druhému klientovi. Příjem dat od druhého klienta je prováděn pomocí metody `recv` (ze třídy `Sockets`):

```
1 data = self.conn.recv(self.buffersize)
```

Následně jsou data uložena do fronty (v tomto příkladě se jedná o klienta s ID 0). Data jsou uložena pouze v případě, že je fronta prázdná. Zamezí se tak hromadění zbytečných dat.

```
1 if videoFrom0.empty():
    videoFrom0.put(data)
```

K odeslání dat (z fronty druhého klienta) dochází pomocí metody `send` (z knihovny Sockets). V tomto příkladě klientovi s ID 1.:

```
  if not videoFrom1.empty():
2     self.conn.send(videoFrom1.get())
      break
```

[16] Je tedy důležité zajistit, aby fronta nebyla kontinuálně obsazena jedním vláknem, čímž by blokovala přístup ostatním vláknům. Tohoto je možné dosáhnout například krátkým uspáním vlákna (jednotky milisekund) ve chvíli, kdy neoperuje s žádnou frontou.

Pokud nejsou žádná obrazová data k dispozici po dobu 0,04 sekundy, je místo obrazu odeslán řetězec 'empty'. Tento mechanismus je přítomen z důvodu nutnosti odeslání dat, protože bez tohoto odeslání není možné další data přijmout. Pokud by druhý klient neobdržel žádná data, nemohl by žádná data vysílat. Kód tohoto mechanismu:

```
if timeDiff > 0.04:
    self.conn.send('empty')
    break
```

Druhý klient už může podle obsahu řetězce snadno poznat, zda se jedná o obrazová data nebo o 'empty' řetězec. Celý tento proces příjmu a odeslání se stále opakuje, dokud se klient neodpojí nebo není ukončen server.

6.6.3 Třída VideoClient

Třída `VideoClient` slouží klientovi k připojení na server a k příjmu a odeslání obrazových dat z tohoto serveru, respektive na tento server. Data k odeslání a data přijatá jsou vkládána do dvou front: `videoOut` a `videoIn`. Fronta `videoOut` je plněna metodou `videoOut` ze třídy `Game` (viz 6.3) a fronta `videoIn` je plněna daty, která jsou obdržena od druhého klienta.

Nejprve se klient připojí k serveru pomocí příkazu:

```
self.sock.connect((host, port))
```

Po inicializaci klienta je v metodě `run` třídy `ClientServer` vytvořeno nové vlákno s metodou `run` z třídy `VideoClient`. Metoda `run` je (kromě inicializace) jedinou metodou třídy `VideoClient`:

run() Tato metoda odesílá a přijímá obrazová data od serveru (tedy od druhého klienta). Příjem dat je obdobný jako u serveru. Obrazová data jsou přijata metodou `recv` a vložena do příslušné fronty:

```
imgStr = self.sock.recv(self.buffersize)
if self.videoOut.empty():
    self.videoOut.put(imgStr)
```

Data jsou opět vložena pouze v případě, že je fronta prázdná a všechna předchozí data už byla odebrána. Odeslání dat je opět podobné jako u serveru:

```
if not self.videoIn.empty():
    imgStr = self.videoIn.get()
```

Pokud nejsou žádná data k odeslání, je opět odeslán řetězec 'empty'.

6.7 Ostatní pomocné metody a spuštění programu

Rozhraní obsahuje několik dalších metod, které nejsou součástí žádné třídy a chybí tedy v předchozím výčtu. Popis příslušných metod:

runGame(game) Tato metoda vytváří instance tříd `GameServer` a `GameClient`, které obsahují hlavní smyčky programu. Metoda je obsažená v souboru citygate.py a je spuštěna přímo z mainové metody (soubor main.py) s parametrem `game`, který představuje instanci třídy `OurGame`. Tato metoda nejprve kontroluje první parametr z příkazové řádky:

```
if len(sys.argv) < 2:
    print >> sys.stderr, "Missing mode parameter"
    printHelp()
    sys.exit(1)

if sys.argv[1] != "client" and sys.argv[1] != "server":
    print "Unknown mode option (either 'server' or 'client')
  "
    sys.exit(1)

if sys.argv[1] == "server":
    isServer = True
else:
    isServer = False
```

Pokud je parametr "server", je vytvořena instance třídy `GameServer`:

```
if isServer:
    gameObj = GameServer(game)
```

V případě "client"je vytvořena instance třídy `GameClient`:

```
if isServer:
    gameObj = GameServer(game)
```

Dále je vytvořeno připojení pomocí metody `create` (z třídy `GameServer`, respektive `GameClient`):

```
gameObj.create()
```

Následně, pokud nedošlo k přerušení programu vstupem z klávesnice, je zavolána metoda `run` (z třídy `GameServer`, respektive `GameClient`), nebo metoda `stop`, pokud došlo k přerušení programu z klávesnice:

```
try:
  gameObj.run()
except KeyboardInterrupt:
  gameObj.stop()
```

getParam(name) Metoda `getParam` (v souboru parameters.py) navrací podle parametru `name` hodnotu parametrů z příkazového řádku, které následují po prvním parametru, který určuje, zda je entita server nebo klient. K získání těchto parametrů slouží pythonovský parser getopt. Definovány jsou tři parametry:

- –address(-a): Udává IP adresu serveru.
- –port(-p): Udává číslo portu pro stavový server.
- –videoport(-v): Udává číslo portu pro video server.
- –videoport(-k): Značí, zda je využíván Kinect. Může fungovat jako přepínač mezi ovládáním Kinectem a webkamerou.

Vývojář si samozřejmě může jednoduše dodefinovat podle potřeby svoje další parametry a zavoláním této metody k nim jednoduše přistupovat.

Samotné spuštění aplikace probíhá v příkazovém řádku. Ukázka spuštění serveru s parametry:

```
main.py server -a 192.168.0.1 -p 690 -v 52000
```

Takto vytvořený server bude přijímat klienty na IP adrese 192.168.0.1. Jeho stavová část bude poslouchat na portu 690 a část přenášející video na portu 52000. Podobným příkazem dojde k vytvoření klienta:

```
main.py client -a 192.168.0.1 -p 690 -v 52000 -k 1
```

Tento klient se pokusí připojit k serveru s IP adresou 192.138.0.1, stavy bude přijímat a odesílat na portu 690 a video na portu 52000. Ke klientovi je připojen Kinect.

6.8 Zhodnocení rozhraní

Kvalitu výše navrženého rozhraní je možné diskutovat pomocí kladů a záporů, které tento návrh přináší.

Kladné vlastnosti:

- Jednoduchost návrhu aplikace - Systém stavů umožňuje vývojáři poměrně rychlý vývoj aplikace. Také jazyk Python tomuto rychlému a jednoduchému návrhu napomáhá.
- Modularita - Díky objektovému návrhu a logickému rozdělení funkčnosti je možné rozhraní poměrně jednoduše dále upravovat a měnit nebo přidávat jeho funkčnost.
- Práce s obrazovými senzory - Jednoduchý přístup k obrazovým senzorům, především k zařízení Kinect.
- Přítomnost serveru - Výskyt autority pro ostatní entity. Zjednodušuje připojení entitám, které nejsou přímo viditelné v síti (neveřejná IP).

Záporné vlastnosti:

- Přítomnost serveru - Přes tento bod prochází mnohem více dat než přes ostatní entity, a vyžaduje tedy zpravidla lepší konektivitu.
- Redundantnost ve stavech - Broadcastový systém odesílání stavů ze serveru zapříčiňuje obdržení jistých redundantních dat klienty. Tato data jsou naštěstí poměrně malá a nepředstavují velký problém.
- Přenos videa pouze pro dva klienty - V současném rozhraní je umožněn přenos videa pouze mezi dvěma klienty. Pro podporu více klientů je nutné třídu upravit.

Kapitola 7

Testovací aplikace

Pomocí rozhraní popsaného v kapitole 6 byla vyvinuta testovací hra s názvem Dots[1], jejímž cílem je především otestovat rozhraní z hlediska funkčnosti a přístupnosti při vyvíjení aplikace. Princip hry je popsán v následující podkapitole. Dále budou ukázány vybrané návrhové detaily (především ty, které mají spojitost s rozhraním) z hry Dots. Nebude zde tedy diskutován návrh celý.

7.1 Principy a mechanismy testovací hry Dots

Hra Dots je hra pro dva hráče ovládaná pomocí zařízení Kinect nebo pomocí počítačové myši. Hráči soupeří mezi sebou v umisťování a sbírání kruhových značek z obrazovky. První hráč je v roli útočníka a pomocí dlaní (pozici dlaní vidí na obrazovce) umisťuje kruhové značky na obrazovku. Umístění značek probíhá přiblížením dlaně směrem ke Kinectu. Druhý hráč, obránce, se snaží umístěné značky pomocí dlaní sebrat (pozice je opět zobrazená na monitoru). Při sběru už nemusí přibližovat dlaň, stačí se značky dotknout.

Vytvořené značky mají omezenou životnost a za určitou chvíli zmizí a není je tedy možné sebrat. Čas zbývající do zmizení je naznačován změnou barvy (od zelené po tmavě růžovou). Pokud obránce nestihne značku sebrat, dostává útočník bod. V opačném případě obdrží bod obránce. Tento mechanismus nutí útočníka rychle umisťovat značky na komplikovaných místech. Počet značek, které existují v jeden okamžik, je omezen.

K jisté interakci dochází také pomocí videa, kdy hráč vidí video s protivníkovým obličejem v centru obrazovky. Po uplynutí stanovené doby se role

[1] V překladu "body"nebo "puntíky".

obrátí a hráč, který dříve útočil, se brání proti hráči, který předtím bránil. Skóre se zachovává po celou dobu hry. Hra není nijak ukončena a hráči mohou hrát teoreticky nekonečně dlouho.

7.2 Stavy hry

Jednou z nejdůležitějších částí návrhu je vybrání stavů, které vhodně reprezentují mechanismy hry. Je také nutné rozhodnout, jaká entita bude o těchto stavech rozhodovat.

Prvním množinou stavů jsou stavy serverové (viz 6.4). Tyto stavy jsou tři:

dotsS Tento stav reprezentuje stav všech živých (kterým nevypršel čas) kruhových značek[2]. Je jedinečný a pro všechny hráče stejný. Ve struktuře je jejich pozice a časová známka udávající, kolik času jim zbývá do konce životnosti. Tato časová známka nereprezentuje přímo čas, ale ukazuje stav, ve kterém se značka nachází a díky němuž je možné určit barvu značky.

timeS Stav udává čas jednotlivých kol (výměn rolí) hry. Tento stav musí být pro všechny zúčastněné stejný a plně řízen nadřízenou entitou.

attackerS Tento stav udává, kdo z hráčů je právě útočník a kdo obránce.

Další množinou jsou klientské stavy vlastněné klienty:

handleftC Stav udává pozici levé dlaně klienta jako tuple strukturu. V případě útočícího hráče údaj reprezentuje pozici, kde byla levá dlaň naposledy přiblížena ke Kinectu. Tento stav je pro každého hráče jiný a je těmito hráči plně vlastněn.

handrightC Stejný jako předchozí stav, ale pro pravou dlaň hráče.

Poslední skupinou stavů jsou stavy klientů plně vlastněné serverem. Takovýto stav obsahuje hra pouze jeden:

scoreCS Tento stav určuje dosažené skóre každého hráče. Musí být pro každého hráče jedinečné, ale plně pod kontrolou serveru.

[2]Písmeno S na konci všech serverových stavů pouze udržuje větší přehlednost ve stavech. Stejně tak písmeno C u klientských stavů vlastněných klienty a CS u klientských stavů vlastněných serverem.

7.3 Výpočty na straně klientů

Úkolem klientů je (kromě zobrazování údajů uživateli) poskytování pozice dlaní serveru. V případě útočníka poskytuje poslední pozici dlaní přiblíženou ke Kinectu, což odpovídá mechanismu umístění kruhové značky, viz kód z metody `updatePlayerState` pro levou dlaň:

```
state['handleftC'] = self.handLeft

if self.hlava[2]-self.leva[2]>400 and self.nextDotLC:
    self.timeNextPut = time.time()
    self.handLeft = [self.leva[0],self.leva[1]]
    self.nextDotLC = False
if not self.hlava[2]-self.leva[2]>400 and not self.nextDotLC and
   (time.time() - self.timeNextPut) > 0.5:
    self.nextDotLC = True
```

Dostatečné přiblížení dlaně je indikováno dostatečným rozdílem vzdálenosti této dlaně a hráčovy hlavy. Tím je zaručená použitelnost při libovolné vzdálenosti hráče od Kinectu. Kód obsahuje mechanismus, který zabraňuje vytvoření více značek při jednom přiblížení dlaně a také jistý minimální čas, který musí uběhnout mezi vytvořením dvou značek. Tímto je zabráněno vytváření velkého množství značek v jeden okamžik.

Princip je obdobný pro pravou ruku a rovněž pro alternativní ovládání pomocí počítačové myši (myší je "přiblížení"provedeno pomocí stisku levého tlačítka). V případě bránícího hráče, kterému stačí se značek pouze dotknout, je kód pro levou dlaň jednodušší (u ovládání myší je sběr značek proveden opět stisknutím levého tlačítka):

```
state['handleftC'] = [self.leva[0],self.leva[1]]
```

Výše popsané mechanismy tedy pouze odešlou serveru pozice dlaní a nerozhodují o umístění značek. O umístění značek se stará sám server.

7.4 Výpočty na straně serveru

Na straně serveru probíhá přidávání nových značek podle údajů od klientů, změna časových stavů jednotlivých značek a odebrání starých značek. Nejprve jsou zpracovány údaje od klientů (pozice dlaní):

```
if len(self.dots)< self.maxDots and not p['handleftC'] == 0 and
   not p['handleftC'] == self.oldLeftHand:
       self.dots.append([p['handleftC'],time.time()])
       self.oldLeftHand = p['handleftC']
```

Údaj z dlaní útočícího hráče je vložen do struktury `dots`. Proměnná `oldLeftHand` zabraňuje vložení stejné značky dvakrát (klient stále zasílá poslední aktivní pozici dlaně) a proměnná `maxDots` určuje, zda nebyl překročen maximální počet značek. Dále následuje zpracování pozic dlaní bránícího hráče a případné odstranění značky ze struktury `dots`:

```
if not self.isDot(p['handleftC'],self.dotsOut)==-1:
       self.dots.remove(self.dots[self.isDot(p['handleftC'],self.
   dotsOut)])
       p['scoreCS']+=1
```

Zda je pozice dlaně dostatečně blízko některé značky, určuje metoda `isDot`. Pokud je pozice dlaně přibližně shodná s pozicí některé značky, je tato značka vymazána ze struktury `dots` a skóre hráče je inkrementováno o jedničku. Následuje odstranění značek, kterým vypršel čas:

```
for index in range(len(self.dots)):
    if time.time() - self.dots[index][1] >4:
        for i,p in client.players.iteritems():
            if state['attackerS'] == i:
                p['scoreCS']+=1
        self.dots.remove(self.dots[index])
        index = 0
```

Algoritmus projde strukturu `dots` a vyřadí značky, které jsou starší než požadovaná hodnota (v tomto případě 4 vteřiny). Pokud je nějaká značka tímto způsobem vyřazena, dojde k inkrementaci skóre útočícího hráče o jedničku. Předposledním krokem je převod struktury `dots` na strukturu `dotsOut`, která je struktuře `dots` podobná, ale jinak reprezentuje časové známky:

```
for index in range(len(self.dots)):
    pos = self.dots[index][0]
    timeStamp = 3
    #NEW TIMESTAMP
    timeDif = time.time() - self.dots[index][1]
    if timeDif >0 and timeDif < 2 :
        timeStamp = 0
```

```
    if timeDif > 2 and timeDif < 3 :
        timeStamp = 1
    if timeDif >3 :
        timeStamp = 2
    self.dotsOut.append([pos,timeStamp])
```

Časová známka je nahrazena pseudo časovou známkou, která určuje stav značky, podle které může klient určit barvu vykreslení značky na obrazovku. Posledním krokem je vložení této struktury do stavové proměnné:

```
state['dotsS'] = self.dotsOut
```

7.5 Kostra

K získání dat o pozici kostry hráče je použita metoda `getSkeleton` ze třídy `kinectTrack`, ze které jsou vybrány pozice obou dlaní a pozice hlavy hráče. Hodnoty os X a Y jsou škálované vzhledem k velikosti okna, ve kterém se hra na monitoru odehrává. Příklad zisku pozice hlavy hráče a uložení do proměnné hlava:

```
if not self.kinect.skeletonQ.empty():
    skelet = self.kinect.skeletonQ.get()
    self.hlava[0] = int(skelet[7][0])
    self.hlava[1] = int(skelet[7][1])
    self.hlava[2] = int(skelet[7][2]*1000)
```

Informace o pozici kostry je převáděná na datový typ Integer, protože obrazové body monitoru není možné dělit. V hodnotách osy Z je bohužel tímto ztracená velká přesnost, a proto je tato hodnota v našem případě násobena před převodem číslem 1000.

7.6 Zpracování videa

Systém přenosu obrazové informace mezi klienty potřebuje jako vstup řetězec obsahující obrazová data[3], která jsou odeslána druhému klientovi, a na výstupu poskytuje stejný řetězec od druhého klienta. Je tedy pouze na vývojáři interaktivní aplikace, jakým způsobem obrazová data zpracuje. V případě

[3] Teoreticky je tedy možné posílat tímto systémem jakákoliv data ve formě řetězce.

testovací hry Dots byl pro zpracování obrazu vybrán formát MJPEG[4]. K tomuto video formátu poskytuje rozhraní sadu metod (viz podkapitola 6.5.3).

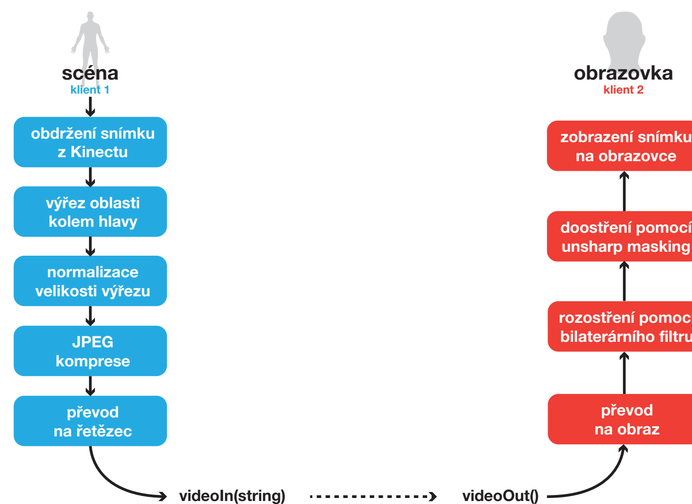
V případě testovací hry Dots je nejprve nutné v metodě `videoIn` pořídit snímek z Kinectu o velikosti 640 na 480 obrazových bodů[5], vyříznout z něho hlavu hráče, vhodně snímek komprimovat a převést na řetězec (schéma celého procesu na obrázku 7.1). K pořízení snímku lze samozřejmě využít třídu `KinectTrack` a k vyříznutí hlavy je použita vlastní metoda `cropHead`, jejímiž vstupními parametry jsou pozice hlavy z Kinectu a velikost ořezu v obrazových bodech:

```
1 if not self.kinect.videoQ.empty():
      if not self.hlava[0] == 0:
3         img = self.kinect.videoQ.get()
          img = self.cropHead(self.hlava, int(65000/self.hlava
      [2]),img)
5         img = cv2.resize(img, (self.hlavaSize, self.hlavaSize),
       0, 0, interpolation = cv2.INTER_LINEAR)
          videoString = saveJpegToMemory(img,40,1)
```

K výřezu hlavy je také použita informace o vzdálenosti hlavy od Kinectu (osa Z), díky čemuž může výřez obsahovat stále stejnou část tváře nezávisle na pozici hráče (jeho reálné vzdálenosti od Kinectu). Hodnota osy Z stoupá se vzdáleností hlavy od Kinectu, a je tedy nutné aby ve vzorci pro velikost výřezu byla jako jmenovatel. Protože je změna hodnoty osy Z poměrně lineární, je možné použít jako čitatel konstantu. V tomto případě byla jako nejlepší empiricky nalezena konstanta 65000. Před samotnou kompresí byla upravena velikost obrazu (100 na 100 obrazových bodů). Komprese probíhá pomocí metody `saveJpegToMemory`, která vytvořený JPEG rovnou převede na řetězec. JPEG komprese obrazu je kompresí ztrátovou, tedy při zmenšení dat dojde i ke ztrátě kvality obrazu.

[4]Motion JPEG je formát videa, kde je každý jednotlivý snímek reprezentován obrázkem ve formátu JPEG. Každý jednotlivý snímek je tedy klíčový a nepotřebuje žádný snímek předchozí. Tento formát je využíván při VOIP video přenosech.

[5]Se standardní velikostí 1280 na 960 obrazových bodů měly některé počítače, na kterých byla hra testována, výpočetní problémy.



Obrázek 7.1: Schéma zpracování obrazových dat odesílatelem a příjemcem.

Uživatel na druhé straně musí pomocí metody `makeCvImageFromMemory` převést řetězec opět na obraz (v tomto případě na obraz vhodný pro knihovnu OpenCv). Pokud je obraz značně komprimován, je výhodné použít metody pro jeho vylepšení[6]. Pro odstranění rušivých artefaktů vzniklých JPEG kompresí a šumu z kamery je nejprve obraz mírně rozostřen pomocí bilaterálního filtru. Tento nelineární filtr oproti Gaussovu filtru více zabraňuje narušení hran [Bradski 2008]. Více o filtru v [Bradski 2008] na straně 113. Praktické provedení pomocí OpenCv (parametry byly nalezeny empiricky):

```
img = cv2.bilateralFilter(img,9,75,75)
```

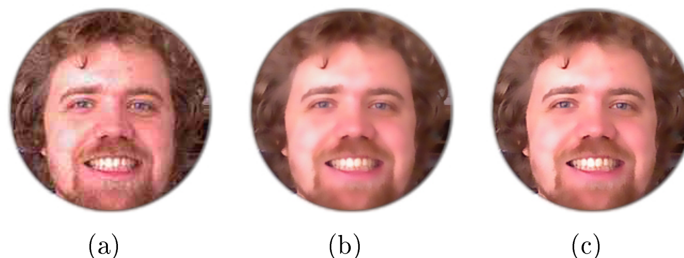
Tímto jsou v obrazu potlačeny rušivé elementy, bohužel je tím ztracena ostrost. Tu je možné získat použitím nějakého doostřovacího algoritmu, například pomocí techniky zvané unsharp masking[7]. Princip spočívá v odečtení rozostřené (pomocí Gaussova filtru) kopie obrazu od původního [Šonka 2008]. Více o tomto filtru v [Šonka 2008] na straně 134. Praktická realizace pomocí OpenCv (parametry byly nalezeny empiricky):

```
1 tmp = cv2.GaussianBlur(img,(5,5),5)
cv2.addWeighted(img, 1.5, tmp, -0.5, 0, img)
```

[6]Tyto metody rozhodně nerekonstruují informace ztracené při kompresi, ale jejich smysl spočívá k zisku přirozenějšího obrazu bez rušivých parametrů.

[7]Termín se nepřekládá do češtiny

Praktická ukázka použitých filtrů na výřez obličeje je zobrazena na obrázku 7.2.



Obrázek 7.2: (a) původní obraz po silné JPEG kompresi, (b) rozostřený obraz a (c) Doostřený obraz.

7.7 GUI

Grafické uživatelské rozhraní bylo vytvořeno pomocí knihovny PyGame a jeho jednotlivé části jsou popsány na obrázku 7.3.

Obrázek 7.3: GUI s popisem jednotlivých prvků.

7.8 Testování a zhodnocení hry Dots

Jak již úvod této kapitoly napověděl, hra Dots testovala především přístupnost a funkčnost samotného rozhraní. Základní implementace hry zabrala přibližně 16 hodin čistého času (ladění drobných problémů nebylo započítáno) a většinu z této doby bylo využito na vývoj a testování sledování hráčovy hlavy a tvorbu GUI. Samotné přímé napojení na rozhraní zabralo času minimálně. Zde je ale nutné zdůraznit, že tvůrce hry, jakožto tvůrce samotného rozhraní, byl s jeho principy dobře obeznámen a novému vývojáři by zřejmě zabralo nějaký čas se s rozhraním seznámit.

Následně byla testovací hra otestována v několika sítích[8] :

localhost Všechny entity (server a dva klienti) byly spuštěny na PC 1 (viz označení v poznámce pod čarou). Z důvodu nemožnosti připojení dvou Kinectů k jednomu počítači byl jeden nahrazen webovou kamerou zabudovanou přímo v notebooku bez možnosti sledovat obličej. V tomto případě vše běželo bez problémů a zpoždění.

LAN (Ethernet) Síť LAN spojená kroucenou dvoulinkou. Server byl umístěn na PC 3, klienti na PC 1 a PC 2. Spojení zajišťoval switch TP-Link. V tomto případě byl výsledek stejný jako v předcházejícím případě.

LAN (Wifi) Stejná konfigurace jako v předchozím případě, pouze PC 1 a PC 2 komunikovaly se switchem TP-Link pomocí Wifi AP. Při průměrné kompresi videa byl přenos plynulý bez viditelného zpoždění. Pouze při nejkvalitnější kompresi videa začal být obraz trhanější.

WAN Server a klient 1 byli umístěny na PC1 a kroucenou dvoulinkou připojeni do sítě Internet poskytovatelem PilsFree (reálná symetrická[9] rychlost kolem 20 Mbit/s). Druhý klient byl umístěn na PC 2 a připojen kroucenou dvoulinkou do sítě Internet poskytovatelem UPC (reálná nesymetrická rychlost kolem 10 Mbit/s download a 1 Mbit/s upload). Při tomto testu bylo možno pozorovat jisté zpoždění odezvy dané vzdáleností jednotlivých entit, které ale nemělo na průběh hry žádný vliv.

Předchozí testy ukázaly, že rozhraní nejenom funguje v reálných podmínkách, ale že je možné v něm poměrně dobře vyvíjet aplikace. Jako jistá nadstavba byl provedený poslední test, který se již nezabýval rozhraním, ale

[8]Specifikace a označení jednotlivých prvků: PC 1 - notebook HP ProBook 4340s, PC 2 - notebook Toshiba L750, PC3 - desktop Intel Core 2 Duo, Wifi AP - Asus WL-300g, switch TP-Link SF1005D

[9]Stejně rychlý upload jako download.

použitím samotné testovací hry v reálných podmínkách na reálných uživatelích. Cílem experimentu bylo pozorovat hráče, jestli dokáží hru ovládat a zda dokáží pochopit její principy, o kterých nedostali dopředu žádné informace. Tohoto testu se zúčastnili čtyři jedinci, z čehož pouze jedna osoba měla vysokoškolské vzdělání technického směru.

Obrázek 7.4: První testovací dvojice.

První dvojici (oba jedinci měli mírné zkušenosti s hraním her, ale žádné zkušenosti s ovládáním pomocí vlastního těla) byl ponechán čas dvě minuty, aby zkusili sami pochopit mechanismy hry. Výsledek byl velmi rozpačitý a bodové ohodnocení, kterého bylo dosaženo, bylo možné připsat pouze na vrub náhody. Největší problém činil mechanismus umístění značek a hráč, který bránil, svojí roli nepochopil. Také mechanismus výměny rolí zůstal nepochopen a hráčům ještě více ztěžoval pochopení hry. Po těchto dvou minutách bylo dosažené skóre pouze 6:3. Následně byly v rychlosti vysvětleny principy hry. V ten okamžik přestali mít hráči jakékoliv problémy s pochopením mechanismů a ovládáním. Po dalších 6 minutách bylo dosaženo skóre 117:110.

K druhé skupině hráčů (jedna osoba z této skupiny měla velké zkušenosti s hraním her a mírné zkušenosti s ovládáním tělem, druhá osoba měla velmi malé zkušenosti s hraním her a žádné s ovládáním tělem) bylo přistoupeno naprosto stejným způsobem. Bez znalosti mechanismů byla podobně bezradná jako skupina předešlá. Pouze hráč, který měl věší zkušenosti s hraním her, byl blízko odhalení mechanismu umístění značek, ale již neodhalil me-

chanismus změny rolí. Dosažené skóre po třech minutách bylo 7:2, pro hráče s většími zkušenostmi s hraním počítačových her. Po vysvětlení principů neměli hráči už téměř žádné problémy s pochopením a ovládáním. Pouze hráč s větší znalostí počítačových her se při umisťování nakláněl výrazně dopředu a tím občas znemožňoval umístění značky (jeho dlaně nebyly tak daleko od hlavy jako u normálně stojícího hráče). Po dalších šesti minutách se skóre zastavilo na velmi vyrovnaných hodnotách 105:104 pro hráče s větší zkušeností.

Obrázek 7.5: Druhá testovací dvojice.

Po pochopení principů hry dokázali všichni hráči hru bez problémů ovládat (ačkoliv byl tento způsob ovládání pro ně nezvyklý) a soupeřit mezi sebou. Je nutné zdůraznit, že smyslem tohoto testu nebylo získat statisticky významná data (testovací vzorek byl příliš malý), ale spíše obdržet prvotní nástřel o přístupu k návrhu těchto aplikací, který pak může být promítnut do úpravy této testovací hry nebo do aplikací budoucích. Test poměrně dobře ukázal, že pokud nejsou principy hry naprosto intuitivní a zřejmé, mohou být hráči bezradní. Možným řešením tohoto problému by mohlo být zjednodušení principů na minimum, přítomnost tutoriálu před samotným hraním nebo poskytnutí různých instrukcí hráči (například ve formě mluveného slova) počítačem při hraní. Po stránce technické fungovala hra bez problémů.

Kapitola 8

Závěr

Cílem této práce bylo navrhnout rozhraní pro snadný vývoj distribuovaných aplikací a her s využitím 2D a 3D obrazových senzorů. Na rozhraní byly kladeny nároky nejenom funkční, kdy byla vyžadována rychlá a bezproblémová komunikace mezi uživateli, ale také nároky na snadný vývoj aplikací v tomto rozhraní a na modularitu, která může umožnit jeho další vývoj a rozvoj.

V rámci práce byla nejprve prostudována teorie distribuovaných systémů, která umožnila vybrat pro rozhraní vhodný topologický model. Jako nejvhodnější byl zvolen model klient-server, jehož hlavní výhodu představuje přítomnost nadřazené entity. Dále byly prostudovány principy TCP/IP protokolu, které osvětlily především rozdíly mezi spolehlivým a nespolehlivým přenosem dat po sítích LAN a WAN. Následovalo teoretické i praktické seznámení se zařízením Kinect jakožto hlavním poskytovatelem obrazových a prostorových dat.

Na základě této teorie bylo v programovacím jazyce Python vyvinuto rozhraní pro tvorbu interaktivních distribuovaných aplikací, které se skládá ze tří hlavních modulů. První modul zajišťuje přenos stavových proměnných mezi klienty a serverem. Modul byl postaven na knihovně PyEnet, která poskytuje vlastní protokol pro přenos dat, jenž spojuje vlastnosti protokolů TCP a UDP se zaměřením na síťové hry. Druhý modul obstarává přenos obrazových dat mezi dvěma klienty. Z důvodů větší datové náročnosti nebylo výhodné obrazová data transportovat pomocí modulu pro přenos stavových proměnných, který rozesílá klientům data ze serveru broadcastovým způsobem, ale byla využita technologie Sockets a protokol TCP. Samotný modul nedefinuje formát videa. Ten je plně v kompetenci vývojáře, který poskytuje modulu pouze obrazová data ve formě řetězce. Třetí modul zajišťuje snadný přístup k obrazovým a prostorovým informacím ze zařízení Kinect pomocí knihovny

PyKinect, která využívá oficiální SDK od společnosti Microsoft a je přímo určená pro jazyk Python. Modul poskytuje data o jedné sledované kostře, hloubkovou mapu prostředí a jednotlivé snímky z RGB senzoru. Modul také zajišťuje základní práci s webovou kamerou pomocí knihovny OpenCV a poskytuje metody pro práci s videoformátem MJPEG (především v návaznosti na modul pro přenos obrazových dat).

Pro potřeby otestování rozhraní v reálných podmínkách byla vyvinuta jednoduchá interaktivní hra. První test představoval už samotný vývoj hry, kdy byla testována přístupnost rozhraní pro vývojáře. Jediným menším problémem, který přímo souvisel s implementací do rozhraní, bylo rozhodnutí, které stavy budou hru popisovat a kdo tyto stavy bude vlastnit a určovat. Samotná implementace stavů už byla velmi rychlá a intuitivní. Dále byla testována funkčnost rozhraní na rozdílných síťových konfiguracích (localhost, LAN a WAN). Ve všech případech bylo rozhraní funkční a hra byla vždy hratelná.

Posledním testem, který představoval již jistou nadstavbu nad tématem této práce, bylo použití testovací hry s reálnými osobami, které o principech hry neměly (alespoň ze začátku) žádnou apriorní informaci. Účelem tohoto testu bylo pozorování, nakolik je hra intuitivní a zda se dají její principy rychle osvojit. Testy naznačily, že hra intuitivní příliš není a všechny z testovaných osob měly bez dalších informací velké problémy hru ovládat a interagovat s protihráčem. Po rychlém ústním vysvětlení principů hry už testované osoby neměly s ovládáním téměř žádné problémy a mohly soupeřit s protihráčem. Tento test naznačil velké nároky na kvalitu designu interaktivních aplikací, které by měly být nejenom zábavné, ale především lehce pochopitelné i bez předchozí instruktáže.

Z předchozích výsledků vyplývá, že rozhraní je nejenom funkční, ale je také možné pomocí něj jednoduše vyvíjet interaktivní distribuované aplikace a díky přiloženým knihovnám také jednoduše využívat zařízení Kinect. Cíle práce, které byly nastíněny v úvodu, byly tedy splněny. V budoucnu by bylo určitě přínosem obohatit rozhraní o další funkčnost, především o podporu modernějších video kodeků a podporu více zařízení pro zisk obrazových a prostorových dat.

Literatura

[Bradski 2008] BRADSKI, Gary a KAEHLER, Adrian. *Learning OpenCV*. O'Reilly Media, first edition, 2008. ISBN: 978-0-596-51613-0.

[Dostálek 2000] DOSTÁLEK, Liboor a KABELOVÁ, Alena. *Velký průvodce protokoly TCP/IP a systémem DNS*. Computer Press, Praha, 2. aktualizované vydání, 2000. ISBN 80-7226-323-4.

[Hoiem 2012] HOIEM, Derek. *How the Kinect Works* [slidy]. University of Illinois [cit.: 28.4.2014]. Dostupné z: `http://courses.engr.illinois.edu/cs498dh/fa2011/lectures/Lecture%2025%20-%20How%20the%20Kinect%20Works%20-%20CP%20Fall%202011.pdf`

[Klimeš] KLIMEŠ, Cyril. *Distribuované systémy* [online]. Ostravská univerzita v Ostravě, Přírodovědecká fakulta, Katedra informatiky a počítačů [cit.: 3.4.2014]. Dostupné z: `http://www1.osu.cz/~prochazka/ds/SkriptaKlimes.pdf`

[Kerkhove 2014] KERKHOVE, Tom. *Introduction to Kinect, releasing this summer and support for Unity & Windows Store apps* [online]. [cit.: 28.4.2014]. Dostupné z: `http://www.kinectingforwindows.com/2014/04/07/build-2014-introduction-to-kinect-releasing-this-summer-and-support-for-unity-windows-store-apps/`

[Knies 2011] KNIES, Rob. *Academics, Enthusiasts to Get Kinect SDK* [online]. [cit.: 28.4.2014]. Dostupné z: `http://research.microsoft.com/en-us/news/features/kinectforwindowssdk-022111.aspx`

[Lowensohn 2011] LOWENSOHN, Josh. *Timeline: A look back at Kinect's history* [online]. cnet.com [cit.: 28.4.2014], Dostupné z: `http://www.cnet.com/news/timeline-a-look-back-at-kinects-history/`

[MacCormick] MACCORMIC, John. *How does the Kinect work?* [slidy]. [cit.: 28.4.2014]. Dostupné z: `http://users.dickinson.edu/~jmac/selected-talks/kinect.pdf`

[Microsoft] Microsoft. *Kinect for Windows Sensor Components and Specifications* [online]. [cit.: 28.4.2014]. Dostupné z: http://msdn.microsoft.com/en-us/library/jj131033.aspx

[Minar 2001] MINAR, Nelson. *Distributed Systems Topologies* [online]. 12/14/2001 [cit.: 5.4.2014]. Dostupné z: http://www.openp2p.com/pub/a/p2p/2001/12/14/topologies_one.html

[Peterka] PETERKA, Jiří. *TCP a UDP* [online]. eArchiv.cz [cit.: 27.4.2014]. Dostupné z: http://www.earchiv.cz/anovinky/ai1864.php3

[Shinners] SHINNERS, Pete. *Python Pygame Introduction* [online]. [cit.: 4.5.2014]. Dostupné z: http://www.pygame.org/docs/tut/intro/intro.html

[Sochor 2013] SOCHOR, Tomáš. *Počítačové sítě 1* [online]. Ostravská univerzita v Ostravě, [cit.: 27.4.2014]. ISBN 978-80-7464-269-2. Dostupné z: http://projekty.osu.cz/svp/opory/prf-sochor-pocitacove-site-1.pdf

[Šonka 2008] ŠONKA, Milan, HLAVÁČ, Václav. a BOYLE, Roger. *Imaga Processing, Analysis, and Machina Vision.* Thomson Learning, third edition, 2008. ISBN: 10: 0-495-24438-4

[Švec 2002] ŠVEC, Jan. *Učebnice jazyka Python (aneb Létající cirkus)* [online]. [cit.: 29.4.2014]. Dostupné z: http://www.py.cz/tut-2.2.pdf

[Tanenbaum 2007] TANENBAUM, Andrew S. a STEEN, Maarten van. *Distributed Systems: Principles and Paradigms.* Prentice Hall, second edition, 2007. ISBN-10: 0132392275.

[Wang 2005] WANG, Hao. *Skype VoIP service - architecture and comparison* [článek]. Institute of Communication Networks and Computer Engineering, University of Stuttgart, 2005. Dostupné z: http://94.23.146.173/ficheros/bfea9a19a5b237024399d1c606e8b7e5.pdf

[Zhang 2012] ZHANG, Zhengyou. *Microsoft Kinect Sensor and Its Effect.* IEEE Multimedia Volume:19, Issue: 2, 2012. str. 4-10

Příloha A

Struktura přiloženého CD

- src [složka] - zdrojové kódy rozhraní a testovací hry
  - img [složka] - obrazové zdroje pro GUI testovací hry
  - citygate.py
  - main.py
  - parameters.py
  - sensors.py
  - videostream.py
- HolecekDP.pdf - tato diplomová práce ve formátu PDF