Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Automatická analýza pohybu ramene pro účely rehabilitace ve virtuální realitě

Plzeň 2019 Vítek Poór

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 1. května 2019

Vítek Poór

Abstract

The work is a partial project that tries to achieve the possibility of rehabilitation of the upper limb in virtual reality. The result of this work is a mechanism analyzing the phase and quality of arm abduction. In the theoretical part, the key requirements for analysis are chosen, especially motion sensing technology and data classification algorithm in real time. The practical part is based on the choice of designed and implemented analyzer usable in Unity engine environment.

Abstrakt

Práce je dílčím celkem projektu, který se snaží docílit možnosti rehabilitace horní končetiny ve virtuální realitě. Výsledkem této práce je mechanismus analyzující fázi a kvalitu abdukce paže. V teoretické části jsou vybrány klíčové požadavky pro analýzu, zejména technologie snímání pohybu a algoritmus klasifikace dat v reálném čase. V praktické části je na základě výběru navrhnut a implementován analyzátor použitelný v prostředí enginu Unity.

Obsah

1 Úvod **6**

1.1 Obsah práce 6

2 Teoretická část **7**

2.1 Virtuální realita 7

2.1.1 Oculus 7

2.1.2 HTC Vive 10

2.1.3 Další 13

2.2 Senzory 13

2.2.1 Měření dat 14

2.3 Metody klasifikace dat v reálném čase 15

2.3.1 Umělá neuronová síť 16

2.3.2 Tunel okolo naměřených dat 20

3 Realizační část **28**

3.1 Návrh řešení pro analýzu abdukce paže 28

3.1.1 Množina dat pro učení 28

3.1.2 Analýza abdukce paže 30

3.2 Implementace řešení 33

3.2.1 Struktura projektu 34

3.2.2 Pomocné struktury a rozšíření 38

3.2.3 Manipulace s naměřenými daty 40

3.3 Dosažené výsledky 42

4 Závěr **45**

Literatura **47**

1 Úvod

Práce vznikla ve spolupráci s fakultní nemocnicí Královské Vinohrady jako dílčí celek projektu, jehož výsledek má usnadňovat rehabilitaci horních končetin pacientů po vážné zdravotní újmě. Tato práce je cílena na pacienty, kteří nedokáží projevovat medicínsky správnou abdukci (pohyb směrem pryč od těla) horních končetin, natož precizně uchopit předmět.

Hlavním cílem práce je vytvoření funkčního modulu pro analýzu pohybu paže za pomoci virtuální reality pro prostředí herního enginu Unity. Takovýto modul bude schopen vyhodnocovat **kvalitu** a **fázi** pohybu paže v reálném čase. Slouží tak jako nedílná součást jádra pro výslednou aplikaci celého projektu s grafickým uživatelským rozhraním.

1.1 Obsah práce

Hlavním obsahem práce je část **teoretická** (viz. 2) a část **realizační** (viz. 3). Teoretická část předkládá potřebná fakta a znalosti pro vyhotovení výsledného analyzátoru pohybu paže. Kapitoly 2.1 a 2.2 představují základní technické možnosti **snímání pohybu** v reálném čase doplněné o jejich teoretický rozbor funkčnosti. Nakonec jsou uvedeny v kapitole 2.3 vybrané algoritmy řešící klasifikaci dat v reálném čase. Dílčí implementované algoritmy navíc obsahují detailní funkční rozbor.

Realizační část je popsána v kapitole 3.1 návrh a v kapitole 3.2 implementaci výsledného modulu pro analýzu abdukce ramene na základě jednoho z algoritmů pro klasifikaci dat v reálném časem. Ten byl vybrán v teoretické části. Nedílnou součástí takto vzniklého modulu je i jeho možná integrace do okolních systémů či aplikací. Pro integraci je klíčová znalost struktury celého projektu, jež je popsána v kapitole 3.2.1. Nakonec jsou diskutovány dosažené výsledky v kapitole 3.3.

2 Teoretická část

Kapitola teoretické části si klade za cíl seznámit čtenáře s možnými hardwarovými ale i softwarovými řešeními problému snímání pohybu paže v reálném čase. Poznatky a informace získané z této kapitoly jsou základem pro návrh a implementaci výsledného modulu pro analýzu abdukce paže popsané v realizační části 3 práce.

Nejprve teoretická část informuje o vybraných společnostech zabývajících se virtuální realitou. V podkapitole 2.1 jsou zmíněny jejich potencionálně použitelné produkty. Jednotlivé výčty možných řešeních jsou doplněny o rozbor použité technologie z pohledu snímání prostředí. Dále se v podkapitole 2.2 představuje základní funkcionalita a existující užití senzorů, použitelných v kontextu snímání pohybu.

Každá výše zmíněná podkapitola obsahuje shrnutí, které by mělo výstižně vysvětlovat, zdali popisovaná technologie bude použita pro požadovaný výsledek této práce.

2.1 Virtuální realita

Virtuální realita je v dnešní době stále rozvíjející se technologie, která umožňuje zobrazit uměle vymodelované prostředí (ať už jako iluzi reálného nebo fiktivního světa) a zapojit uživatele do něj formou vizuální interakce. Základní hardwarovou jednotkou jsou virtuální brýle neboli **headset**, které zprostředkovávají obraz. Pro lepší uživatelskou zkušenost existují další periferie, které umožňují více realistický pocit z okolního simulovaného prostředí. Tento proces se označuje jako snímání pohybu neboli **motion capture** a je pro získávání dat v reálném čase klíčovým. Ve virtuální realitě je snímání pohybu zprostředkováno periferií snímače nebo kamery. Snímají se všechny ostatní periferie včetně headsetu, ovladačů či dodatečných senzorů.

2.1.1 Oculus

Oculus VR je americká společnost spadající pod Facebook. Zaměřuje se na vývoj hardwaru a softwaru pro virtuální realitu.

Samsung Gear VR

Jedním z prvotních produktů je **Samsung Gear VR** ve spolupráci s firmou Samsung Electronics. Tyto brýle používají mobilní telefon Samsung Galaxy namísto displeje, přičemž telefon slouží jako výpočetní a rendrovací jednotka, zatímco brýle samotné jsou zprostředkovatelem zorného pole vidění a snímačem pohybu. Brýle a telefon se spolu propojují pomocí USB-C nebo micro-USB.

Oculus Rift

Dalším produktem této firmy je **Oculus Rift**. Brýle pracují s grafickou kartou počítače a společně s nimi jsou distribuovány dva ovladače a dva senzory pro snímání pohybu brýlí a ovladačů.

Pro správnou konfiguraci a ovládání vytvořila firma Oculus stejnojmenný software, který z počátku pracuje jako průvodce instalací ovladačů a ohraničení prostředí pro snímání pohybu. Po úspěšné instalaci hardwaru a integraci do pokojového prostředí slouží tento software jako úvodní obrazovka s různými možnostmi ve virtuální realitě (např.: možnost nastavení, stahování různorodých aplikací z obchodu Oculus, spouštění aplikací...).

Oculus Go

V neposlední řadě je vhodné zmínit produkt **Oculus Go**, který spojuje mobilitu z modelu Samsung Gear VR a výkonnost z modelu Oculus Rift. Tento model je distribuován jako samostatný kus hardwaru s vlastní výpočetní jednotkou. Balení obsahuje brýle a, oproti modelu Oculus Rift, zmenšené ovladače.

Jelikož jsou brýle ze své podstaty bezdrátové a nepotřebují grafickou kartu počítače, můžeme se s nimi pohybovat kdekoliv. Neobsahují ovšem rozumné snímání pohybu. Ve výsledku brýle dokáží rozeznat pouze rotaci okolo své osy, nikoliv prostoru.

Samostatnost tohoto produktu v důsledku snižuje výkon oproti modelu Oculus Rift. Zatímco Oculus Rift pracuje v rozlišení 2160x1200 pixelů rozprostřených na 90Hz OLED obrazových panelech, Oculus Go operuje v rozlišení 2560x1440 na 60Hz, nebo 72Hz LCD obrazových panelech. Dané omezení kvůli samostatnému procesoru lze vidět na snížení zobrazovací frekvence.

Balíček pro vývoj softwaru na Oculus

Aplikace podporující Oculus hardware se dají vytvářet za pomoci balíčku pro vývoj softwaru **Software Development Kit** (zkráceně **SDK**), který je volně ke stažení na oficiálních stránkách firmy Oculus. Dostupné balíčky jsou dvojího druhu: PC SDK a Mobile SDK.

PC SDK napomáhá průběhu tvorby aplikace pro produkt Oculus Rift. Práce s SDK probíhá v jazyku C++. Balíček obsahuje mimo jiné balíčky:

- Audio SDK - napomáhá nastavení zvuku pro kvalitnější uživatelský zážitek
- Platform SDK - sjednocený vizuální vjem aplikace a zjednodušený vývoj známých herních vlastností
- Avatar SDK - předdefinované objekty představující headset a ovladače propojené s reálným hardwarem.

Mobile SDK obsahuje knihovny a nástroje pro vývoj nativních mobilních aplikací pro Oculus Go a Samsung Gear VR. Mobile SDK vystavuje VrApi, které je možné integrovat jako knihovnu třetích stran a není tedy potřeba herní engine. Oproti tomu s PC SDK lze vyvíjet nativní aplikace. Dává se přednost použití herního enginu Unity nebo Unreal Engine. Oficiální webové stránky firmy Oculus obsahují rozsáhlou dokumentaci, která mimo jiné obsahuje detailní návody integrace a použití PC SDK v enginech Unity a Unreal Engine.

Srovnání

Oculus zastřešuje všechny možné druhy užití virtuální reality, od jednoduchého nahrazení výpočetní jednotky mobilním telefonem (Gear VR) přes extrémně výkonný produkt Oculus Rift propojený s desktopovým počítačem až po mobilní bezdrátové brýle Oculus Go s vlastní výpočetní jednotkou.

Při výběru hardwaru pro účely tohoto projektu je klíčová technologie snímání pohybu, kterou nabízí pouze produkt Oculus Rift. Jak již bylo zmíněno výše, tento produkt obsahuje dva senzory pro snímání pohybu. Tyto senzory jsou nastavitelné a přenositelné.

Jednou z nevýhod je jejich uspořádání v rámci pokoje, kde se provádí snímání. Vyžadují poměrně přesné nároky na správné vymezení pokojového prostředí pro snímání. Jejich nedodržení se odráží ve výsledném herním zážitku nedokonalým snímáním a zobrazováním prostředí.

Další výhodou či nevýhodou je jejich přenositelnost. Přenositelnost je skvělá při prvotní konfiguraci či pozdějšímu přeorganizování pokoje. Nevýhoda však tkví v tom, že změna jejich polohy způsobí nepřesné snímání. I za předpokladu, že je již produkt nakonfigurován a je vymezen prostor pro snímání.

2.1.2 HTC Vive

HTC Vive je balíček zařízení pro virtuální realitu, který byl vytvořen společnostmi HTC a Valve Corporation. Produkt v základním balení obsahuje headset, dva ovladače a dva senzory. K tomuto hardwaru je k dispozici, podobně jako u Oculusu software, pro instalaci a konfiguraci prostředí s názvem **VivePort**.

Headset neboli **Head Mounted Display (HMD)** sám o sobě obsahuje OLED zobrazovací panely s rozlišením 2160x1200 pixelů a obnovovací frekvenci 90Hz. Připojují se stejně jako Oculus Rift do grafické karty počítače.

Ovladače

HTC Vive v základním balení operuje se dvěma základními ovladačemi, které jsou bezdrátové a mimo klasická postraní tlačítka obsahují multifunkční trackpad a 24 senzorů. Senzory jsou rozmístěny do kruhu na konci ovladače.

Kromě výše zmíněných ovladačů dovoluje systém snímání polohy snímání dodatečných senzorů, takzvaných **Vive Tracker**ů. Tyto senzory lze připevnit na jakýkoliv fyzický předmět a vytvořit tak věrohodnější uživatelský zážitek ve virtuální realitě.

Základové stanice

Cizími slovy **Base Station** je laserový hardware pro systém snímání polohy **Lighthouse tracking system**. Využívá HTC Vive a je jeho nedílnou součástí. Jedná se o krabičku o hraně zhruba sedm centimetrů, která se zpravidla připevňuje na zeď do výšky asi dvou metrů.

Většinou se používají dvě takovéto krabičky. Ty dokáží snímat 360-stupňovitý pohyb brýlí s obrazovkou a ovladači v prostoru 5x5 metrů (ve verzi 1.0). Uživatel se může pohybovat a orientovat kdekoliv v dosahu snímačů. Společně se SteamVR tvoří klíčovou technologii pro takzvanou **full-room** zkušenost spočívajícím ve volném fyzickém pohybu prostorem pokoje.

Sledování v prostoru

Sledování ovladačů a brýlí v prostoru je dosaženo systémem Lighthouse tracking system, který je navržen a vyroben firmou Valve a je integrován do **SteamVR**. SteamVR je balíček pro vývoj software taktéž od firmy Valve. Ten usnadňuje vývoj aplikací pro HTC Vive (jde o obdobu Oculus SDK). V kontextu snímání prostoru se v rámci tohoto balíčku jedná o **SteamVR Tracking**.

Zjednodušeně řečeno, SteamVR Tracking je kompletní hardwarový a softwarový systém, který dokáže určit v reálném čase, kde se jednotlivé snímané objekty nachází v dosahu prostoru snímaného pokoje základovými stanicemi. Hardwarovým prvkem tohoto systému jsou právě základové stanice popsané výše. Sofwarovým prvkem není nic jiného nežli Lighthouse tracking system.

Výsledné snímání poté probíhá následujícím způsobem. Základové stanice zaplavují místnost neviditelnými infračervenými paprsky najednou z mnoha LED diod a dvou lazerových emitorů, které neustále rotují. Šedesátkrát za vteřinu LED diody vydají světelný impuls, který slouží jako synchronizační mechanismus. Následně se rozprostře horizontální a poté vertikální rovinný paprsek světla z lazerů. Mezitím snímané objekty (headset, ovladače ...) detekují svit LED diod a paprsek lazeru, protože jsou pokryty fotosenzory. Samotné snímání tkví v tom, že pokud snímané zařízení detekuje rozsvícení LED diod, odstartuje si interní počítadlo. To se inkrementuje do té doby, dokud zařízení nedetekuje další synchronizační impuls. Poté se využije spojitost mezi polohou existence fotosenzoru a dobou, kdy zařízení detekovalo paprsek lazeru k matematickému výpočtu exaktní relativní pozice a orientace zařízení vůči základové stanici. V podstatě výpočetní jednotka, která řeší výpočet relativní pozice senzoru ke stanici, má k dispozici průsečík dvou rovin, z nichž je schopno zjistit orientaci. Výsledná poloha v prostoru je zjistitelná pomocí další stanice, jakožto průsečík dvou přímek získaných průsečíkem rovin.

Tento systém tedy, na rozdíl od ostatních systémů, nepoužívá vizuální snímání prostoru. Díky tomu pracuje při vyhodnocování rychleji. Pokud při snímání existují alespoň dvě základové stanice, potom je systém schopen snímat velice přesné pozice a orientaci snímaných zařízení ve 3D prostoru. Přesnost je dána faktem, že každý ovladač má na sobě řádově desítky fotosenzorů, které situaci vyhodnocují šedesátkrát za vteřinu. Výsledná poloha a orientace se počítá v samotném "mozku" tohoto systému na desktopovém počítači.

HTC Vive Pro

Roku 2018 přišel na trh vylepšený model HTC Vive nesoucí název HTC Vive Pro. Významné změny oproti původnímu modelu jsou zejména v celkové vizuální podobě brýlí. Navíc se zlepšily zobrazovací panely z OLED displejů na AMOLED displeje. Zvedlo se také rozlišení z původního 2160x1200 pixelů na 2880x1600 pixelů. Obnovovací frekvence je stejná 90Hz. Z těchto důvodů se zvedly požadavky na minimální model grafické karty počítače zajišťující vykreslování obrazu.

Se změnou designu HMD přibyl interní mikrofon a slouchátka, která v předešlém modelu byla řešena externí periferií. Pro tuto práci je nejpodstatnější nová verze snímání prostředí základovými stanicemi **SteamVR Tracking 2.0**. Nová verze systému snímání prostředí rozšiřuje snímaný prostor z původních 5x5 metrů na 10x10 metrů. Nové základové stanice verze 2.0 jsou menší, tišší, levnější, spolehlivější a mají nižší příkon oproti původní verzi 1.0.

Nová verze výrazně redukovala základovou stanici tím, že odstranila LED diody a jeden ze dvou motorů vysílajících paprsek. Zbylý motor nyní dokáže vyslat horizontální i vertikální paprsky. LED diody byly odstraněny, protože byl odstraněn koncept s časovou synchronizací a počítadlem. Nyní byl přidán výstupní DATA pin, díky čemuž lze posílat signálem více komplexních dat. Tento model enkóduje a dekóduje data přímo do infračerveného světelného paprsku. Nemusí se složitě počítat pozice a orientace z počítadla.

Tyto skutečnosti mají za následek nekompatibilitu s původním modelem HTC Vive. Nicméně zpětná kompatibilita snímacího systému verze 1.0 s modelem HTC Vive Pro je zajištěna. Z podstaty věci však není vhodné kombinovat verze snímacího systému.

Závěr

I přes komplikovanější instalaci (zejména snímacího hardwaru) produktu HTC Vive i jeho novější verzi HTC Vive Pro, se tato technologie prozatím jeví jako nejvhodnější technický základ pro vyhotovení našeho projektu.

Vzhledem k ostatním technologiím, jež řeší stejný problém snímání pohybu v reálném čase, produkuje systém snímání pro Vive z podstaty své funkčnosti nejpřesnější výsledky. Nutno dodat, že z finančních důvodů je projekt vytvářen za pomoci HTC Vive a nikoliv jeho novější verze HTC Vive Pro. Nicméně tato skutečnost přímo neovlivňuje kvalitu a dosažené výsledky hotového modulu.

2.1.3 Další

Oculus a HTC Vive jsou bezpochyby v dnešní době největšími společnostmi, jež produkují kvalitní hardwarová řešení virtuální reality a nejrozšířenější sadu softwarových balíčků pro vývoj aplikací. Nicméně existují i další firmy, které řeší stejný problém či z větší části jen jeho část.

Z pohledu technologie snímání stojí za zmínku Kinect. Jedná se o technologii od společnosti Microsoft. Je vytvořena ve dvou verzích, přičemž každá z nich používá světelné paprsky a videokameru pro výsledné snímání. Na podobném principu, snímání ovladačů a headsetu, je postaveno řešení virtuální reality v produktu Playstation od firmy Sony. PlayStation navíc přidal k hardwaru pro snímání také headset a dodatečné ovladače.

Z pohledu headsetu vznikají obdoby produktu Samsung Gear VR, brýle pro chytré mobilní telefony, či Oculus Go, brýle bez technologie snímání okolí. Najdou se ale na současném trhu i brýle, které se dají integrovat s existující technologií prostorového snímání. Za zmínku stojí headsety firmy Pimax Technology, ke kterým je k dispozici volně stažitelná aplikace z oficiálních stránek firmy. Ta umožňuje konfiguraci a integraci brýlí do existující instalace HTC Vive.

Ostatní technologie nepřináší žádnou podstatnou změnu oproti HTC Vive. Nejsou vhodné pro tento projekt. Zejména výměna brýlí může sloužit jako možné vylepšení současně zvolené technologie. Problém nastává v tom, že není zajištěna stejná či lepší odezva při integraci s existující snímací technologií.

2.2 Senzory

V obecném slova smyslu je **senzor** (anglicky **sensor**) technická součástka, nebo zařízení spolupracující s nějakou elektronikou. Senzor detekuje dané událostí či změny v obecně rozdílném prostředí. Senzor slouží sám o sobě pouze jako zdroj informací, které nabývají smyslu až při jejich zpracovávání různou, z pohledu senzoru neznámou, výpočetní jednotkou (elektronikou). Druhů senzorů je celá řada. Pro účely této práce se zaměříme pouze na senzory, které nějakým způsobem dokáží zachycovat informace o průběhu pohybu.

Z tohoto pohledu lze konkrétně využít **akcelerometr**, **magnetometr** a **gyroskop** (anglicky ve stejném pořadí **accelerometer**, **magnetometer** a **gyroscope**). Akcelerometrem se získávají údaje o **zrychlení**, magnetometrem údaje o **magnetické indukci** pole Země a gyroskop popisuje **úhlovou rychlost**. Základním zdrojem informací je akcelerometr. Zbylé dva senzory

slouží pro doplnění či upřesnění chybějících informací. Trojice senzorů umožňuje na teoretické úrovni sestrojit algoritmus, který bude na základě získaných aktuálních informací určovat celkovou polohu a rotaci. Avšak uvedení takového algoritmu do praxe by produkovalo nepoužitelné výsledky, protože by vzniklá akumulovaná chyba byla moc vysoká.

Toto tvrzení můžeme potvrdit například převodem zrychlení na pozici. Proces zjištění hodnoty pozice z hodnoty zrychlení vyžaduje dvojí integraci, jelikož zintegrujeme-li zrychlení, získáme rychlost a zintegrujeme-li rychlost, získáme pozici. Navíc samotný akcelerometr produkuje malou chybu měření způsobenou vlivem okolí. Je zřejmé, že čím více výpočtů původní hodnota absolvuje, tím nepřesnější bude hodnota výsledná. V našem případě je výsledek v praxi nepoužitelný.

Na druhou stranu lze výsledný analyzátor abdukce paže postavit přímo na naměřených údajích a nezjišťovat polohu či rotaci, která by byla zatížena chybou. Není zcela jasné, zdali by takto vzniklý analyzátor produkoval kvalitní výsledky, a proto nebude v rámci této práce implementován, protože již existující mechanismy snímání pohybu s virtuální realitou produkují velice přesné výsledky.

I přes tyto skutečnosti jsou samotné senzory velice praktickým aparátem a povedlo-li by se v budoucnu zkonstruovat kvalitní analyzátor založený právě na takovýchto senzorech, byl by to značný přínos pro celý vznikající projekt. Výhody řešení analyzátoru, založeného na senzorech nezávislých na virtuální realitě, jsou především v možném vlastním zkonstruování takovýchto senzorů a zlepšení tak uživatelské zkušenosti s výsledným produktem celého projektu.

Z pohledu koncového uživatele tkví nevýhoda vybraného řešení snímání pohybu v jeho instalaci. Pokud by se podařilo zkonstruovat vlastní senzory, které by pracovaly s bezdrátovým headsetem, snížila by se tak markantně pořizovací cena potřebného hardwarového řešení. Navíc by se značně zredukoval použitý prostor společně se snadností instalace.V současné chvíli nelze vynakládat potřebný čas do hledání ideálního řešení, nýbrž je žádoucí celý projekt dokončit a začít s testováním v produkčním prostředí. Z tohoto důvodu je rozumné ponechat zvolenou technologii virtuální reality, která je již hotová a v konceptu projektu se pouze použije.

2.2.1 Měření dat

V rámci celého projektu proběhlo měření dat za použití, jak sestavy virtuální reality, tak výše zmíněných senzorů. Měření senzorů se uskutečnilo, pro jejich značný potenciál, jako budoucí náhrada sestavy virtuální reality.

Samotné měření bylo provedeno pomocí chytrých mobilních telefonů. V dnešní době drtivá většina těchto zařízení má v sobě zabudován akcelerometr, magnetometr i gyroskop. Jediné úskalí v měření dat pomocí integrovaných senzorů v mobilním telefonu je způsob jejich získávání. Mobilní telefony sice v sobě obsahují senzory, ale nikoliv software potřebný pro práci s nimi.

Za účelem snadnější práce se senzory byla použita již existující aplikace **Sensorstream IMU+GPS** [8], která je volně stažitelná z obchodu **Google Play** a je určena pro mobilní zařízení se systémem **Android verze 2.3.3 a vyšší**. S použitím správného nastavení aplikace Sensorstream jsme schopni měřená data přesměrovávat pomocí **UPD streamu** do aplikace jiné, což nám umožňuje nakládat s měřenými daty libovolným způsobem. Za tímto účelem vznikla aplikace [7], která měřená data z původní aplikace přijímá, patřičným způsobem je formátuje a nakonec ukládá do souborů. Vývoj aplikace [7] nebyl proveden v rámci této práce, proto není zahrnuta do přílohy.

Z praktického hlediska mobilní telefony nahrazují snímaná zařízení sestavou virtuální reality. Při měření je nutné připnout mobilní telefony na požadované oblasti těla koncového uživatele. Pro pohyb horní končetiny se jedná o oblast ruky, paže a hrudníku. Takto vzniklá měřená data lze teoreticky použít obdobným způsobem jako data měřená pomocí zařízení HTC Vive. I přesto, že data prakticky nejsou v současné době využívána, tvoří nedílnou součást možného budoucího procesu implementace analyzátoru založeného na senzorech.

2.3 Metody klasifikace dat v reálném čase

Obecně proces klasifikace dat zařazuje data do tříd, které jsou při samotném procesu známé a předem definované. Každý algoritmus klasifikace se obecně rozhoduje na základě takzvaného **rozhodovacího pravidla**. Rozhodovací pravidlo určuje náležitost zkoumaného prvku do konkrétní třídy (či množiny tříd). Takové pravidlo je pro každou metodu unikátní a charakterizuje ji.

Data získávána v reálném čase jsou ze svého charakteru časově rychle proměnlivá. Aby mohly metody klasifikace taková data klasifikovat, musí mít nějakou již natrénovanou a klasifikovanou množinu dat, podle kterých se rozhodují. Toto počáteční nastavení metod se označuje jako **učení s učitelem** (anglicky **supervised learning**). Jedná se o jednu z metod strojového učení aplikující se na algoritmy klasifikace dat.

Dále se tato práce uchyluje ke dvěma vybraným možnostem metod klasifikace dat použitelných v jejím kontextu řešení. Jedná se o metodu neuro-

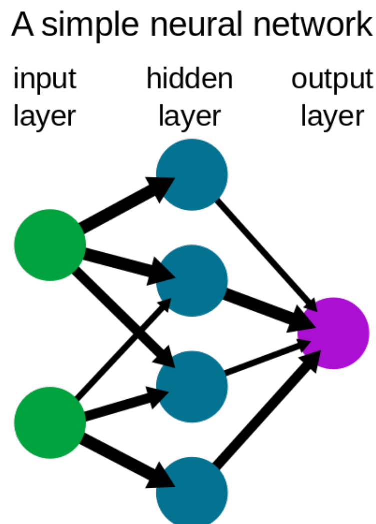
nové sítě 2.3.1 a uměle vytvořenou metodu řešení pomocí k-dimensionálního stromu okolo kterého je geometricky postaven tunelu 2.3.2. Tunel je použit při návrhu analyzátoru v realizační části 3.

2.3.1 Umělá neuronová síť

Neuronová síť (anglicky **neural network**) je z biologického hlediska síť neuronů. Neuron je buňka, jež dokáže komunikovat s jiným neuronem pomocí synapsí. Synapsí myslíme strukturu, která přenáší elektro-chemický signál mezi neurony. Spojení více takovýchto neuronových sítí tvoří rozsáhlou mozkovou síť.

Z pohledu oboru umělé inteligence a strojového učení se dá tento aparát neuronové sítě uměle převést do počítače. Vzniká tak **umělá neuronová síť** (anglicky **artificial neural network**). Umělá neuronová síť pracuje s **umělými neurony** neboli **uzly** (anglicky **artificial neurons** respektive **nodes**).

Topologicky se umělé neuronové sítě dělí do vrstev (anglicky **layers**). První vrstva se stará o vstupy. Další vrstva je takzvaně **skrytá** a zpravidla jich může být více. Skryté vrstvy obsahují jednotlivé umělé neurony. Poslední vrstva je výstupní. Symbolické ztvárnění vrstev je vidět na obrázku 2.1 (viz. [10]).



Obrázek 2.1: Ukázka vrstev neuronové sítě

Jako každou klasifikační metodu můžeme i umělou neuronovou síť učit. I u ostatních metod klasifikace, učení lze provádět takzvaně **s učitelem**

(anglicky **supervised learning**) nebo **bez učitele** (anglicky **unsupervised learning**). Z principu metody učení s učitelem potřebujeme testovací data, u nichž předem známe požadovaný výstup. Výsledné učení poté probíhá nastavováním vah a prahů jednotlivých umělých neurononů tak, aby se výstup umělé neuronové sítě co nejvíce podobal předem známému výstupu. Oproti tomu přístup metody učení bez učitele spočívá v tom, že data která máme, nejsou ručně označena a jejich vzniklý výstup se snažíme upravovat tak, aby co nejvíce odpovídal požadovanému výstupu. Umělé neuronové sítě bez učitele většinou nachází využití mimo jiné ve shlukování. Oproti tomu umělé neuronové sítě s učitelem dokážeme efektivně využít právě v metodách klasifikace.

V poslední řadě se umělé neuronové sítě dají dělit podle datového toku. Pokud výpočet algoritmu neuronové sítě končí na výstupní vrstvě, mluvíme o takzvané **dopředné umělé neuronové síti** (anglicky **feedforward artificial neural network**). Oproti tomuto klasickému návrhu sítě stojí rekurzivní neuronové sítě, kde výstupy z výstupní vrstvy slouží jako vstupy pro vrstvu vstupní.

Umělý neuron

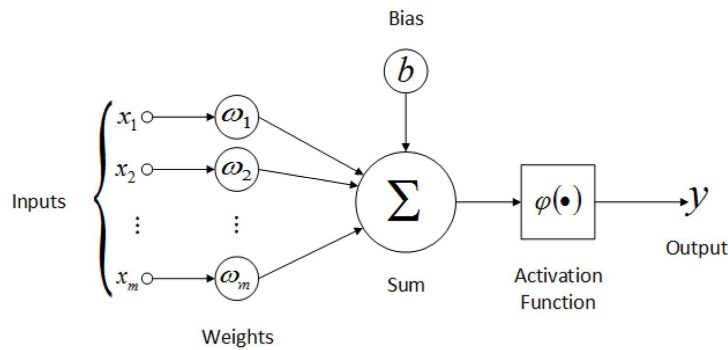
Jak již bylo zmíněno výše, umělý neuron je elementárním prvkem umělé neuronové sítě. Každý umělý neuron charakterizuje alespoň jeden vstup a právě jeden výstup. Rozhodovacím jádrem umělého neuronu je takzvaná **aktivační funkce** (anglicky **activation function** nebo **transfer function**), která je nelineární.

Všechny vstupní hodnoty jsou přenásobené odpovídajícími **váhami** a poté hromadně sečteny. Jednotlivé váhy jsou klíčovými pro požadovaný výsledek, tudíž právě váhy a **prahová hodnota** (anglicky **bias** nebo **treshold**) jsou proměnlivou složkou umělého neuronu, a tudíž i celé umělé neuronové sítě. Právě ony podléhají procesu učení. Celý tento proces lze popsat obrázkem 2.2 (viz. [1]).

Aktivační funkce

Aktivační funkce produkuje výstup samotného umělého neuronu. Snažíme se ji zvolit tak, aby výsledek byl mapován na konečný interval často mezi nulou a jedničkou. Na základě takového požadavku můžeme konstatovat, že výsledná aktivační funkce by měla být nelineární.

Výběr aktivační funkce úzce souvisí s konkrétním problémem, který má umělá neuronová síť řešit. Pravděpodobně nejjednodušší aktivační funkce je



Obrázek 2.2: Struktura umělého neuronu

pro binární klasifikátor. Taková funkce může být například **Heaviside Step** funkce, definována následovně:

$$f(x) = \begin{cases} 1, \, x \geq 0, \\ 0, \, x < 0. \end{cases}$$

Na základě výsledku součtu sumy jednotlivých vstupů pronásobených jednotlivými váhami a prahové hodnoty bude výstupem této funkce nula nebo jednička. Umělý neuron se z pohledu umělé neuronové sítě buď aktivuje, nebo neaktivuje.

Další užitečnou aktivační funkcí z pohledu klasifikace dat je funkce **Sigmoid**. Tato funkce je definována následovně:

$$f(x) = \frac{1}{1 + \mathrm{e}^{-x}}$$

Stejně jako step funkce, tak i tato funkce mapuje vstupní hodnoty na interval od nuly do jedničky. Navíc se jedná o hladkou funkci, kde se malá změna x-ové hodnoty v okolí nuly projeví v nárůstu v y-ové ose.

V poslední řadě stojí za zmínku funkce **hyperbolického tangentu** neboli **Tanh**. Funkce hyperbolického tangentu je definována následovně:

$$f(x) = \frac{2}{1 + \mathrm{e}^{-2x}} - 1$$

Tato funkce mapuje výstup na interval mezi zápornou a kladnou jedničkou. Čtenář snadno nahlédne, že tato funkce je podobná funkci sigmoid. Pro tyto dvě funkce platí následující vztah:

$$tanh(x) = 2 * sigmoid(2x) - 1$$

Hlavní rozdíl těchto dvou funkcí tkví v jejich derivaci, přičemž hyperbolický tangent má strmější derivaci než funkce sigmoid.

Umělá neuronová síť vícevrstvého perceptronu

Vícevrstvý perceptron (anglicky **multilayer perceptron**) je druh dopředné umělé neuronové sítě využívané pro klasifikaci dat. Tato síť je složena minimálně ze tří základních vrstev: vstupní, skryté a výstupní.

Všechny vrstvy, kromě vstupní, jsou složené z umělých neuronů **perceptronů** (anglicky **perceptron**), které jsou definovány mimo jiné nelineární aktivační funkcí. Ta se používá pro klasifikaci. V kontextu vícevrstvého perceptronu je jako aktivační funkce použit hyperbolický tangent respektive funkce sigmoid.

Umělá neuronová síť vícevrstvého perceptronu používá techniku učení s učitelem založenou na principu zpětného propagování chyb (**backpropagation**). Princip zpětného propagování je zjednodušeně založen na zpětné distribuci chyb vypočítané na výstupu sítě.

Shrnutí

Umělé neuronové sítě se v dnešní době hojně používají jako jistá řešení v oboru umělé inteligence a strojového učení. V kontextu řešení této práce lze využít potenciál vícevrstvého perceptronu a použít danou umělou neuronovou síť jako hledaný klasifikátor.

Klasifikátor (respektive analyzátor) by na základě vstupního **vektoru příznaků** produkoval **kvalitu** a **fázi** aktuálního pohybu. Vektor příznaků by se mohl skládal z jednotlivých složek vektorů polohy a rotace všech snímaných senzorů. Dohromady by se tedy jednalo o 24 příznaků, protože snímané senzory jsou dohromady čtyři (z oblastí ruky, paže, hlavy a hrudi) a každý je popsán tří rozměrným vektorem polohy a rotace. Výstup by byl reprezentován veličinami určující fázi a kvalitu v rozmezí od nuly do jedničky. Tedy fáze pohybu by začínala v hodnotě nula (začátek pohybu) a končila v hodnotě jedna (konec pohybu). Oproti tomu hodnota kvality by začínala v jedničce a snižovala by se s kvalitou až do nuly. Ta by znamenala naprosto špatný pohyb.

Samotné vyhodnocování hodnot fáze a kvality by bylo závislé na trénovaných datech. Jako množinu trénovacích dat by se použila data z měření, které bylo provedeno v rámci celého projektu se zařízením HTC Vive. Z těchto dat lze použít právě jeden korektní pohyb, který je mimo jiné popsán i časem. Trénování neuronové sítě by mohlo probíhat na základě těchto dat, z nichž by se vytvořil výše popsaný vektor příznaků společně s časovou značkou. Poté by při získávání hodnoty aktuální fáze mohla sloužit informace o časovém průběhu. Kvalita by mohla jít určovat normovanou vzdáleností od nejbližších trénovacích dat.

Takto navržený analyzátor pomocí neuronové sítě by mohl produkovat přijatelné výsledky, které jsou náchylné na procesu trénování a sestavení vstupního vektoru příznaků. Při sestavení vstupu se musí brát v potaz váhy jednotlivých složek vektoru, jelikož každý senzor má jiný vliv na průběh pohybu. Navíc jsou data z měření spjatá k jedné konkrétní poloze, tudíž je nutné nějakým způsobem provést transformaci k současné měřené poloze.

Za těchto okolností jsme nezkoušeli implementovat analyzátor postavený na tomto řešení. Neví se, zdali by toto řešení fungovalo v produkčním prostředí a do jaké míry by produkovalo přijatelné výsledky. Nicméně při zjištění nepoužitelnosti analyzátoru založeného na k-dimensionálním stromě 2.3.2 by bylo možné toto řešení vyzkoušet.

2.3.2 Tunel okolo naměřených dat

Metoda řešení popsaná **tunelem** vychází z myšlenky geometrické interpretace použitých naměřených dat. Principiálně se okolo dat jednoho korektního pohybu postaví tunel o poloměru, který je dán konstantou. Ve vzniklém tunelu se ze sjednocených dat postaví **k-dimensionální strom**, na jehož základě probíhá následná analýza pohybu, která produkuje hodnoty popisující **kvalitu** a **fázi** pohybu.

Tunel, jako datová struktura, obsahuje naměřená data ze všech dílčích snímaných senzorů. Ovšem výsledný k-dimensionální strom se dá postavit pouze z jedné množiny dat. Tento problém je řešen sjednocením vektorů popisujících pozici a rotaci jednotlivých senzorů (z oblasti hlavy, paže, ruky a hrudníku) do jednoho 24-dimensionálního vektoru. Proces sjednocení je doplněn o vynásobení jednotlivých složek vektorů koeficienty nastavení programu, jelikož každý senzor má odlišný vliv na průběh a analýzu pohybu. Je vhodné surová a spojená data od sebe separovat. Získá se tím možnost transformace tunelu nezávisle na změně dat stromu. Z pohledu správného určení fáze a kvality je transformace tunelu důležitá, jelikož naměřená data, okolo kterých je tunel postaven, jsou měřena v konkrétní poloze různé od polohy uživatele. Samotné transformování stromu by byla neefektivní záležitost, proto je vhodné strom stavět až před počátkem analýzy pohybu paže (respektive po poslední úplné transformaci tunelu).

Určení fáze a kvality je záležitostí použití postaveného k-dimensionálního stromu. Celá analýza je postavena na prohledávání stromu algoritmem nejbližšího souseda. Hledá se uzel stromu, jehož vzdálenost k bodu hledanému je nejmenší. Z takto nalezeného nejbližšího uzlu se na základě vzdálenosti určí kvalita pohybu. Při určování kvality se bere v ohled poloměr tunelu. Výsledná kvalita se snižuje s narůstající vzdáleností od nalezeného bodu až po

hranici tunelu, kde je nulová. U určování fáze se vychází z předpokladu, že naměřená data, z kterých je postaven strom, obsahují časovou značku. Lze tím pádem sestrojit časový interval počátku a konce pohybu. Na základě časového intervalu lze určit fázi bodu pohybu při vytváření uzlu stromu. Výsledná fáze lze určit fází nejbližšího nalezeného bodu.

K-Dimensionální strom

V počítačových odvětvích, potažmo v počítačové grafice, je k-dimensionální strom (anglicky **k-dimensional tree**) speciální datová struktura umožňující prostorové dělení obecně k-dimensionálních bodů, kde k symbolizuje počet dimenzí prostoru.

Prapůvodce k-dimensionálního stromu je **strom binární** (anglicky **binary tree**), na němž je založeno prosté dělení nanejvýš dvou potomků pro každý uzel, počínaje uzlem kořenovým. Při aplikaci metody **binárního dělení prostoru** (anglicky **binary space partitioning (BSP)**) na daný prostor při použití binárního stromu, získáváme datovou strukturu **BSP strom** (anglicky **BSP tree**), ze kterého přímo vychází k-dimesionální strom.

Metoda binárního dělení prostoru rekurzivně rozděluje daný prostor na zpravidla dvě podmnožiny polygonů či rovin, které dohromady tvoří konvexní celek daného děleného prostoru. Každý uzel, počínaje kořenem, obsahuje rovinu, která rozděluje prostor na dva podprostory. Polygony či jakékoliv objekty prostoru jsou poté obsaženy v listech stromu a tyto skupiny jsou navzájem disjunktní.

Metoda binárního dělení prostoru je obecný proces, který dělí prostor dokud se nesplní nějaká konkrétní požadovaná podmínka. Takto vzniklé dílčí podprostory BSP stromu nemají obecně žádné požadavky na orientaci daných podprostorů. Právě v této skutečnosti tkví rozdíl mezi BSD stromem a k-dimesionálním stromem. K-dimensionální strom dělí rekurzivně prostor podle souřadných os patřičných dimenzí.

Nad k-dimensionálním stromem lze definovat, kromě základních algoritmů pro vytvoření stromu a vložení/smazání prvku, **algoritmus hledání nejbližšího souseda** (anglicky **nearest neighbor search algorithm**). Algoritmus hledání nejbližšího souseda je klíčovým pro klasifikaci dat v kontextu této práce. Umožňuje vyhledání nejbližšího prvku ve stromu k prvku hledanému. Na základě jeho vzdálenosti od nejbližšího prvku ve stromu lze klasifikovat **kvalitu** hledaného řešení.

Jelikož se k-dimensionální strom jako klasifikátor pro řešení této práce obejde bez algoritmů vložení a smazání prvku, budou tyto algoritmy z analytické části vynechány. Toto tvrzení utvrzuje fakt, že klasifikátor se bude učit

z již existující sady dat prostým **vytvořením stromu** a výsledná klasifikace bude založena na algoritmu **hledání nejbližšího souseda**.

Vytvoření stromu Jak již bylo zmíněno výše, pro vytvoření stromu bude potřeba sada dat (množina k-dimensionálních bodů). Vstupní množina bodů musí být konečná a úplná, ve smyslu celistvosti. Z principu pozdější klasifikace se žádné další body nebudou přidávat za běhu programu (ačkoliv obecná implementace k-dimensionálního stromu danou akci povoluje).

Konstrukce stromu je rekurzivní proces, který ze vstupních dat a hloubky zanoření nalezne souřadnou osu, podle které se bude prostor dělit a na jejím základě nalezne ve vstupních datech medián. Mediánem je poté reprezentován prvek stromu a jeho levý potomek se určí rekurzivně s daty menšími než medián a pravý potomek obdobně s daty většími než medián.

Algoritmus konstrukce k-dimensionálního stromu nejlépe popisuje pseudokód 2.1. Jeho detailnímu popisu budou věnovány následující odstavce.

Algorithm 2.1 Konstrukce k-dimensionálního stromu

1: **procedure** GenerateTree(collection of *data*, *depth*)
2: $currentAxis \leftarrow depth$ mod k
3: $median \leftarrow$ **getMedian**$(data, currentAxis)$
4: $node \leftarrow$ **new Node**$(median)$
5: $node.Left \leftarrow$ **GenerateTree**$(data$ before $median, depth + 1)$
6: $node.Right \leftarrow$ **GenerateTree**$(data$ after $median, depth + 1)$
7: **return** *node*
8: **end procedure**

Na prvé řádce pseudokódu 2.1 definujeme proceduru *GenerateTree*, jež jako parametry přijímá kolekci dat *data* (čili obecně množinu k-dimensionálních bodů) a informaci o hloubce zanoření rekurze *depth*, protože konstrukce stromu je ze své podstaty rekurzivní proces.

Obsahem druhé řádky je získání souřadné osy, podle které se bude prostor dělit. Při výběru souřadné osy postupujeme postupně, tedy prvně dělíme podle osy x, v další fázi podle osy y, a tak dále až do počtu dimenze *k*. Pokud hloubka zanoření *depth* přesáhne počet dimenzí *k*, postupuje se opět od počáteční souřadné osy x. Například pokud bude počet dimenzí *k* roven 3 (tedy tří rozměrný prostor), potom při prvním průchodu se prostor bude dělit podle x-ové osy, dále podle y-ové osy, poté podle z-ové osy, následně znovu podle x-ové atd. Tohoto efektu dosáhneme operací modulo současné hloubky zanoření *depth* s celkovým počtem dimenzí *k*.

Řádka číslo tři obsahuje volání symbolické funkce *getMedian*, která vyhledá v současné kolekci k-dimensionálních bodů *data* medián podle současné dělící souřadné osy *currentAxis* získané v předchozím kroku. Funkce je symbolická, jelikož možných postupů získání mediánu je mnoho a konkrétní návrh implementace není podstatný. Je však důležité zmínit, že medián je sice hledán v kolekci podle hodnoty současné dělící souřadné osy *currentAxis*, ale výsledkem je zde myšlen celý bod obsahující onen nalezený medián z kolekce bodů *data*.

Dalším krokem, zachyceném na čtvrtém řádku, je vytvoření uzlu stromu *node*. Tento uzel je ve svém základu definován bodem, podle kterého je prostor dělen. V našem případě je tímto bodem hledaný *median*.

Poslední dva řádky obsahují rekurzivní volání procedury konstrukce stromu *GenerateTree* a jejich výstup přiřazují jako potomstvo vytvořeného uzlu *node*. Nejprve se rekurzivně naleznou leví potomci a posléze praví. Na levé straně od uzlu jsou uzly s body menšími než hledaný současný *median* a na prvé straně uzly s body většími než současný *median*. Nutno dodat, že třídění těchto bodů z kolekce *data* probíhá podle mediánu současné dělící osy *currentAxis*, ale metoda v rekurzy již dělí podle osy následující díky inkrementaci proměnné *depth*.

Možností konstrukce k-dimensionálního stromu je více. Tyto možnosti se liší především algoritmem výběru mediánu či samotnou reprezentací jednotlivých uzlů stromu (respektive dělením prostoru). Algoritmus výběru mediánu není pro konstrukci stromu v kontextu této práce klíčovým. Postačí nám jednoduchý algoritmus seřazení vstupní posloupnosti a výběru prostředního prvku. Oproti tomu dělení prostoru a s tím spojená reprezentace jednotlivých uzlů stromu může mít za následek nevyváženost stromu. V konstrukci stromu popsané procedurou 2.1 je použita reprezentace uzlů body, které přímo dělí prostor rovinami vedoucími skrze ně. Další možnost reprezentace uzlů je podle kolmých rovin k jednotlivým souřadným osám. V tomto případě jsou jednotlivé body obsažené až v listech stromu a jednotlivé uzly obsahují informace o osách, které dělí prostor. Tento proces může vést k nevyváženosti výsledného stromu a tato vlastnost není pro užití v klasifikátoru požadována.

Hledání nejbližšího souseda Algoritmus hledání nejbližšího souseda (anglicky **nearest neighbor search**) je v kontextu této práce stavebním kamenem výsledného klasifikátoru. Algoritmus hledá uzel s hodnotou vzdálenostně nejbližší k hledanému bodu. Prohledávání stromu je ze své podstaty rekurzivní záležitost a tento algoritmus není výjimkou.

Hledání začíná od kořene stromu. Nejprve se algoritmus rozhodne, v jaké

větvi bude pokračovat v hledání. Poté rekurzivně prohledává danou větev a hledá vzdálenostně nejbližší hodnotu uzlu k hodnotě hledané. Tento naivní přístup produkuje korektní výsledky do té doby, dokud hledaný nejbližší uzel není obsažen v opačné větvi. Z toho důvodu je potřeba při vynořování se z rekurze zkontrolovat, zda rozdíl mezi hodnotou uzlu v opačné větvi a hledaným uzlem není menší než nalezená nejbližší vzdálenost. Pokud je, poté se v prohledávání pokračuje v opačné větvi.

Postup algoritmu hledání nejbližšího souseda nejlépe vystihuje pseudokód 2.2. Následující odstavce detailně popisují jeho funkcionalitu.

Algorithm 2.2 Hledání nejbližšího souseda

```
 1: procedure Search(node, value, depth)
 2:     if node is null then
 3:         return null
 4:     end if
 5:     currentAxis ← depth mod k
 6:     if node.Value[currentAxis] > value[currentAxis] then
 7:         nextBranch ← node.Left
 8:         oppositeBranch ← node.Right
 9:     else
10:         nextBranch ← node.Right
11:         oppositeBranch ← node.Left
12:     end if
13:     tempNode ← Search(nextBranch, value, depth + 1)
14:     nearestNode ← closestDistance(value, tempNode, node)
15:     nearestDistance ← getDistance(nearestNode.Value, value)
16:     nearestValue ← value[currentAxis] - node.Value[currentAxis]
17:     if nearestDistance ≥ |nearestValue| then
18:         tempNode ← Search(oppositeBranch, value, depth + 1)
19:         nearestNode ← closestDistance(value, tempNode, nearestNode)
20:     end if
21:     return nearestNode
22: end procedure
```

První řádek pseudokódu 2.2 definuje hlavičku procedury *Search*, která přijímá tři formální parametry. Prvním z nich je uzel *node*, oproti kterému algoritmus bude hledat nejbližší vzdálenost k druhému parametru *value*, který symbolizuje hledanou hodnotu (k-dimensionální bod). Posledním parametrem není nic jiného než hloubka zanoření *depth*, používána kvůli rekurzivnímu volání procedury obdobně jako u konstrukce stromu 2.1.

Druhá až čtvrtá řádka představuje podmínku, jež testuje nulovost parametru uzlu *node*. Pokud je vstupem této procedury prohledávaný uzel, který neexistuje, procedura skončí a vrátí nulovou hodnotu. Tato podmínka je užitečná pro pozdější zjišťování nejbližšího uzlu *nearestNode* pomocnou procedurou **closestDistance**.

Na pátém řádku, stejně jako v proceduře konstrukce stromu, se deklaruje a inicializuje lokální proměnná *currentAxis*, která jednoznačně definuje současnou prohledávanou souřadnou osu. Opět se počítá pomocí operace modulo aktuální hloubky zanoření *depth* s celkovým počtem dimenzí k.

Od šestého až do dvanáctého řádku probíhá zjištění následující prohledávané větve *nextBranch* a větve jí opačné *oppositeBranch*. Pojmem větve je myšlen směr prohledávání. Fakticky se do těchto proměnných přiřazují potomci aktuálně prohledávaného uzlu *node*. Takovéto dělení je podmíněno porovnáním hodnot dle aktuální souřadné osy *currentAxis* uzlu *node* a hledanou hodnotou *value*. Hodnota na souřadné ose bodu je naznačena hranatými závorkami [] a lze si pro jednoduchost tedy představit k-dimensionální bod jako pole hodnot. Ty začínají od nultého indexu představující x-ovou souřadnou osu atd. Pokud aktuální prohledávaný uzel *node* obsahuje bod s hodnotou na souřadné ose *currentAxis* ostře větší než hledaný bod *value*, potom bude následující prohledávaná větev *nextBranch* levá, a proto jí protější směr prohledávání *oppositeBranch* bude pravý. V opačném případě se tyto směry prohodí. Následující prohledávaná větev *nextBranch* bude pravá a jí opačná větev *oppositeBranch* bude levá. Takto určený směr prohledávání je užitečný pro pozdější jednoznačné určení směru potencionálního prohledávání při vynořování z rekurze.

Poté se na třináctém řádku provádí rekurzivní hledání se stejnou hodnotou *value* v následující větvi *nextBranch*. Tentokrát se to děje ale v chronologicky následující souřadné ose. Výsledek tohoto hledání je uložen do proměnné *tempNode*. Tato proměnná se dále používá pro zjištění bližší vzdálenosti k hledanému bodu *value* procedurou **closestDistance**.

Procedura **closestDistance** je zavedena kvůli lepší čitelnosti algoritmu. Obecně vstupem se stal bod se dvěma uzly a výstupem jeden z uzlů vzdálenostně bližší ke vstupnímu bodu. Tak v podstatě pouze spočítá vzdálenosti obou uzlů ke vstupnímu bodu a vrátí ten bližší uzel. Pro pokračování v popisu celkového algoritmu to znamená, že proměnné *nearestNode* bude přiřazen buď *tempNode*, nebo *node*, a to v závislosti na blízkosti k bodu *value*. Zde se také projevuje první podmínka, která je napsána na druhém řádku, že *tempNode* může obsahovat nulovou hodnotu. V tomto případě se ještě před samotným výpočtem vzdáleností zkontroluje nenulovost obou vstupních uzlů a při zjištění neexistence jednoho z nich procedura jednoduše

vrátí ten druhý jako výstup.

Na následujícím patnáctém řádku se kalkuluje přesná vzdálenost *nearestDistance* mezi prozatím nejbližším nalezeném uzlu a hledaným bodem v pomocné proceduře **getDistance**. Tento výpočet je zde zohledněn pro přehlednost celkového algoritmu, i přestože se tento stejný proces počítal uvnitř procedury **closestDistance** v předešlém kroku.

Naplnění proměnné $nearestValue$ na šestnáctém řádku je vesměs obdoba porovnání na řádku šestém. Zjišťuje se zde rozdíl hodnot bodu hledaného *value* a bodu obsaženého uzlem *node* v konkrétní souřadné ose $currentAxis$. Tento výpočet je zde kvůli přehlednosti následující podmínky a porovnává tyto dvě hodnoty.

Poslední podmínkou algoritmu je vyhodnocení, jestli algoritmus bude prohledávat i opačnou větev. Stane se tomu tak v případě, kdy nejbližší nalezená vzdálenost *nearestDistance* je větší, nebo rovna absolutní hodnotě $nearestValue$, nebo rozdílů hodnot bodů aktuálního prohledávaného uzlu *node* a hledané hodnoty *value* podle souřadné osy $currentAxis$. Toto rozhodnutí je důležité, jelikož samotné vnořování probíhá na základě porovnávání hodnot bodů podle souřadné osy. Může ale nastat situace, kdy vybrané směry podle porovnání hodnot bodů jsou správné ale uzel s bodem s menší vzdáleností leží ve směru opačném. Proto je potřeba při vynořování se z rekurze ještě zkontrolovat, zdali uzel v opačném směru prohledávání nemá lepší rozdíl hodnot. Pokud se tomu tak stane a algoritmus se uchýlý do opačné větve, provedou se stejné operace jako na třínáctém a čtrnáctém řádku s tím rozdílem, že procedura **Search** prohledává v *oppositeBranch* a procedura **closestDistance** porovnává výsledek hledání v opačné větvi s prozatimním nejbližším nalezeným uzlem $nearestNode$.

Procedura nakonec vrací $nearestNode$, který obsahuje uzel s nejbližším hledaným bodem k bodu *value*. Nutno dodat, že samotný algoritmus je při prvním volání volán s parametry *node* jakožto kořenem stromu a s nulovou hloubkou *depth*.

Metrik pro výpočet vzdáleností je mnoho. V tomto algoritmu není explicitně definována ale ve výsledné implementaci je použita **metrika eukleidovská**, která počítá vzdálenost obecně k-rozměrných bodů odmocninou sumy kvadrátů rozdílů jednotlivých složek, neboli

$$\sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}.$$

Výpočetní složitost Při určování výpočetní složitosti se omezíme pouze na algoritmy popsané výše, tedy konstrukci stromu 2.1 a hledání nejbliž-

šího souseda 2.2, jelikož jediné tyto algoritmy jsou potřebné pro výsledný klasifikátor.

I přesto, že nám v prozatímním návrhu nezáleží na efektivnosti konstrukce stromu, je dobré vědět úskalí procesu. Jediný stěžejní a implementačně závislý proces v konstrukci stromu je hledání mediánu. Naivní metoda pro hledání mediánu je formou seřazení vstupních prvků a následné nalezení prostředního prvku. Tato metoda je závislá na efektivnosti řadícího algoritmu, který v nejlepším případě dosahuje časové složitosti $O(n * logn)$. Samotný výběr prostředního prvku se odehrává v konstantní časové složitosti. Výsledná časová složitost pro naivní hledání mediánu je tedy $O(n * logn)$. Existuje však časově efektivnější metoda nesoucí název **Median of medians**, která tento problém řeší v časové složitosti $O(n)$ (důkaz v [3]). Jelikož je medián hledán při každém zanoření, výsledná časová složitost konstrukce stromu bude větší. Zanoření v binárním stromě je logaritmické, tudíž výsledná časová složitost bude násobkem časové složitosti hledání mediánu a $O(logn)$, proto tedy v nejlepším případě dostáváme $O(n * logn)$. Nutno zdůraznit, že v současné implementaci je použito hledání mediánu naivním způsobem a výsledná časová složitost je $O(n * log^2 n)$. Paměťová složitost je potom dána počtem prvků, ze kterých je strom konstruován, tj. $O(n)$.

Při určování časové složitosti algoritmu hledáním nejbližšího souseda musíme brát v potaz hloubku zanoření a počet průchodů do opačných větvích při vynořování se z rekurze. Hloubka zanoření binárního stromu je $O(logn)$. V nejlepším případě (nenastane žádné hledání v opačné větvi) je časová složitost hledání nejbližšího uzlu $O(logn)$. V nejhorším případě se musí prohledávat i ostatní větve, všechny uzly stromu, kterých je n. Výsledná časová složitost se pohybuje mezi nejlepším případem $O(logn)$ a nejhorším možným případem $O(n)$. Dále zmiňme, že k-dimensionální strom je obecně neefektivní pro malé množství vzorků. Hledání v nejlepším případě ještě podmiňuje počet uzlů $n >> 2^k$, kde k je počet dimenzí. Tato vlastnost je dána rekurzivní podstatou algoritmu. Při málo vzorcích o velké dimenzi se musí prohledávat značně více uzlů a není dostatečně vidět efekt **prořezávání** stromu (rozhodování se jakým směrem půjdeme na základě porovnání hodnot v uzlu). Pokud tato podmínka není splněna, dochází zde k degradaci časové efektivnosti i v nejlepším případě na $O(n)$ (viz. [6]).

3 Realizační část

Kapitola realizační části si klade za cíl seznámit čtenáře s návrhem a implementací výsledného modulu pro analýzu abdukce na základě informací získaných v teoretické části.

Nejprve navrhujeme možná řešení analyzátoru v kapitole 3.1. V návrhu jsou mimo jiné také popsána data, která jsou klíčová pro jeho korektní chod. Poté se popisuje na základě návrhu jeho implementace v kapitole 3.2. U implementace zmiňujeme navíc dílčí projekty, pomocné struktury a rozšíření. Nakonec byly provedeny testy nezbytných implementovaných funkcionalit. Výsledky testů jsou shrnuty v kapitole 3.3.

3.1 Návrh řešení pro analýzu abdukce paže

Z teoretické části 2 této práce vyplývá, že jádrem výsledného modulu analyzujícího abdukci paže v reálném čase bude k-dimensionální strom 2.3.2. Připomínáme, že nad stromem je dostačující definovat pouze proceduru konstrukce stromu 2.1 a algoritmus hledání nejbližšího souseda 2.2, na jehož základě se bude provádět výsledná analýza.

Tyto dvě procedury je potřeba obalit další logikou, která bude umožňovat natrénování k-dimensionálního stromu jakožto klasifikátoru, a tím zajišťovat správnou analýzu abdukce paže. Učení klasifikátoru se zakládá na jednoduché konstrukci stromu. Algoritmus konstrukce 2.1, který je představen v teoretické části, není potřeba upravovat. Učení bude závislé na formátu datové množiny, z níž se výsledný strom následně složí. Tento formát popisujeme v podkapitole 3.1.1. Algoritmus výsledné analýzy abdukce paže již bude potřeba zkonstruovat zvlášť. Jeho podrobnému popisu a samotné definici konkrétních výstupů, které budou popisovat analýzu abdukce paže v reálném čase, je věnována podkapitola 3.1.2.

3.1.1 Množina dat pro učení

Předtím, než bude popsán formát množiny dat potřebných pro konstrukci stromu, je nutné objasnit zdroje těchto dat. Z teoretické části vyplývá použití, jakožto zdroje dat, snímacího zařízení pro virtuální realitu HTC Vive (viz. 2.1.2). Pro samotné snímání pohybu potřebujeme více snímatelných zařízeních, než je headset a ovladače. Jelikož se bude snímat pouze pohyb jedné horní končetiny, vystačíme si s jedním ovladačem umístěným v oblasti ruky

(místo ovladače lze použít jakékoliv dostupné snímatelné zařízení, protože uživatel z principu nemusí, a v drtivé většině případů nedokáže, zvládnout ovladač uchopit) a dodatečným snímatelným zařízením připnutém přibližně v oblasti středu paže. Tato dvě snímaná zařízení postačí pro informování o poloze a orientaci horní končetiny. Nedokážeme z nich ale určit orientaci vůči tělu uživatele. Celková informace o pohybu horní části těla je klíčová pro správnou abdukci, protože pacienti mají tendenci pomáhat si náklonem hrudního koše. Zmíněný častý náklon je pro korektní abdukci nepřijatelný. Z tohoto důvodu je zapotřebí dalšího snímatelného zařízení připnutého na hrudníku v oblasti prsou. Dohromady tedy máme čtyři zdroje dat v oblastech: ruky (ovladač či dodatečný senzor), paže (dodatečný senzor), hrudníku (dodatečný senzor) a hlavy (headset). Každý z těchto senzorů produkuje v reálném čase data, která lze reprezentovat prostorovými vektory polohy a rotací relativní ke snímacím zařízením.

V rámci projektu, do kterého naše práce přináší modul analýzy abdukce paže, se provedlo měření s výše popsanou sestavou HTC Vive. Výsledky měření byly zpracovány aplikací vyvinutou speciálně pro tento úkol. Aplikaci vyvinul zbytek týmu, který pracuje na projektu, do kterého přispívá i výsledek této práce. Aplikace produkuje textový soubor pro každé snímané zařízení. Textové soubory obsahují samotná data ve formátu

$$< \textit{čas} > < \textit{vektor polohy} >, < \textit{vektor rotace} >$$

vždy na novém řádku. Čas je popsán

$$< \textit{hodinou} : \textit{minutou} : \textit{sekundou.milisekundou} > .$$

Jednotlivé složky vektorů jsou desetinná čísla s tečkou a jsou od sebe odděleny čárkou.

Nutno dodat, že takto vzniklá data obsahují tři správné a tři špatné abdukce paže. Navíc i správnou abdukci paže po patnácti procentním vzrůstu. Celý tento proces snímání uživatele, ze kterého jsou vzniklá data pořízena, byl zaznamenán na videokameru.

Kdybychom vzali všechna data, sjednotili je podle času a zkonstruovali z nich strom, výsledek by nedával smysl. Pro samotné učení postačí pouze jeden správný pohyb. Za účelem extrakce dat jednoho správného pohybu ze souborů byla vytvořena aplikace v enginu Unity (viz. projekt **UnityAnimacniTest** v příloze), která mimo jiné načítá výše popsané datové soubory a na jejich základě vytváří a spouští animaci celého procesu. Extrakce jednoho správného pohybu spočívala ve vyříznutí požadovaného pohybu z vytvořené animace a uložení do externích souborů ve stejném formátu, jako byla data

načítána. Vzniklé soubory jednoho správného pohybu paže tvoří základní data pro proces učení. Navíc je aplikace rozšířena o možnost extrakce dat, které lze ručně označit pomocí modifikace animací pomocí prostředí Unity. Každá označená animace pohybu obsahuje navíc parametry fáze a kvality, které lze libovolně modifikovat v rámci ručního značení.

Nyní tedy máme k dispozici relevantní data pro konstrukci stromu ale bohužel ve čtyřech různých souborech. Takto vzniklá skutečnost nahrává ke sjednocení dat dohromady. Sjednocení je vhodné dělat za běhu programu, kdy před samotným vytvořením stromu se načtou všechny čtyři soubory a postupně se z nich vytvoří jedna datová množina. Ta obsahuje sloučené informace o poloze a rotaci každého řádku zvlášť. Tím pádem dostáváme 24-dimensionální body, pokud budeme na polohu a rotaci pohlížet jako na 6-dimensionální bod. Jediné úskalí tohoto řešení spočívá právě ve sloučení polohy a rotace do jedné veličiny. Za tímto účelem jsou představeny konstanty nastavení programu, které přenásobují jednotlivé složky vektorů polohy a rotace. Konstanty jsou unikátní pro každou složku výsledného 24-dimensionálního vektoru, protože data každého snímaného zařízení mají navzájem odlišný vliv na výsledný sloučený vektor, a tudíž je žádoucí jednotlivé vlivy patřičně upravit. Z principu by data získaná například z oblasti hrudníku neměla natolik ovlivňovat výsledné rozhodnutí jako data získaná například z ruky. Výsledkem je tedy proces učení jednoduchou konstrukcí 24-dimensionálního stromu, nad kterým bude daný klasifikátor provádět analýzu pohybu paže.

3.1.2 Analýza abdukce paže

Základem pro analýzu abdukce paže je zkonstruovaný k-dimensionální strom (v našem případě je k rovno dvaceti čtyřem). Požadavky analýzy jsou zjištění **fáze** a **kvality** pohybu. Oboje veličiny mapujeme pro přehlednost a jednoduchost na interval od nuly do jedné. Jedná se tedy o desetinná čísla.

Fáze nabývá nulové hodnoty při klidové poloze horní končetiny, tedy v momentu kdy uživatel ještě s pohybem paže nezačal. Fáze by měla lineárně růst s provádějící abdukcí paže až do jejího maxima, kdy uživatel má ruku v úrovni hlavy. V tento moment by měla fáze nabývat hodnoty maximální, v našem případě hodnoty jedna. Při addukci (pohyb do klidové polohy) paže by fáze měla opět lineárně klesat společně s paží až do nulové hodnoty. Z pohledu uživatele do počáteční polohy horní končetiny.

Kvalitu můžeme definovat vždy jako správnou. Správnost označit hodnotou jedna. Při potencionálním vychýlení pohybu paže od správného pohybu se tato hodnota bude lineárně zmenšovat až po nulovou hodnotu. Ta sym-

bolizuje naprosto chybný pohyb. Postup ohodnocování kvality vychází ze situace nastávající při počátku respektive konci pohybu. Kdybychom totiž kvalitu ohodnocovali stejně jako fázi, tedy od nulové hodnoty při počátku pohybu až do hodnoty jedna při správné úplné abdukci paže, potom bychom nebyli schopni jednoznačně určit kvalitu počátku a konce pohybu. Neměli bychom dostatečné informace o průběhu pohybu. V momentě, kdy nejsme schopni rozhodnout o kvalitě z důvodu nedostatku dat, je uživatelsky přívětivější informovat o správném pohybu (i když pohyb není plně správný) a postupně kvalitu měnit při získání již dostatečného počtu dat.

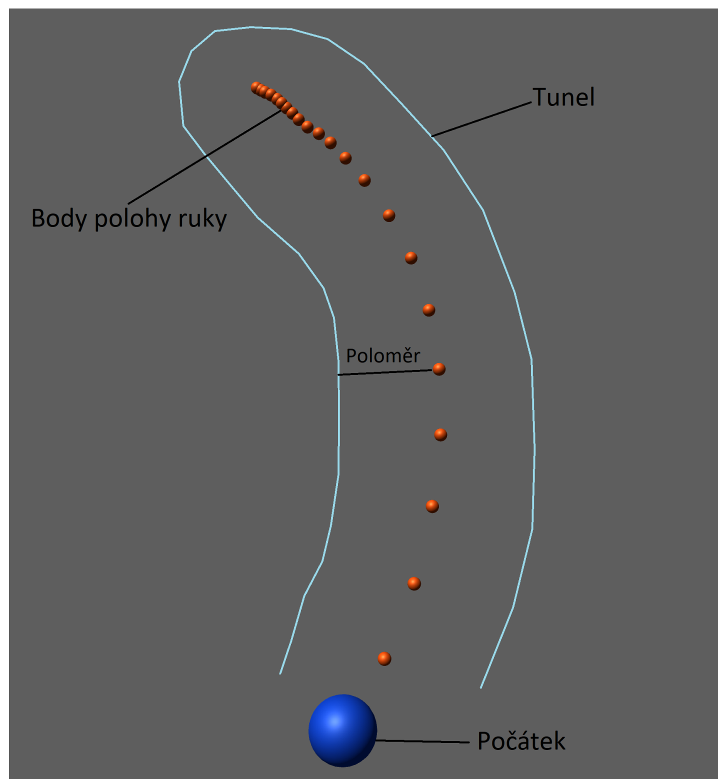
Stěžejním bodem analýzy fáze a kvality jsou získávaná data. Podobně jako je tomu u konstrukce stromu. Opakujeme, že data jsou získávána ze čtyř snímaných zařízeních. Každé zařízení ze své podstaty v libovolný čas produkuje informace o poloze a rotaci. Je nutné snímaná data shromažďovat a výslednou analýzu počítat až při explicitním zavolání dané procedury pro analýzu pohybu. Jelikož procedura pro analýzu pohybu paže prohledává strom postavený z dat správného pohybu, musí mít hledaný bod stejný počet dimenzí, jako má postavený strom. Z těchto důvodů je vhodné vystavit rozhraní, které obsahuje metodu, jež zpracovává příchozí data, a metodu, jež při zavolání vyhodnotí fázi a kvalitu z posledních úplných dat.

Je nutné, aby metoda, která má za úkol shromažďovat data, přijímala kromě informace o poloze a rotaci i informaci o typu snímaného zařízení (respektive informaci o tom, s jakou částí těla tyto data souvisí). Teprve s dodatečnou informací o typu snímaného zařízení lze seskupit shromážděná data do jednoho 24-dimensionálního bodu potřebného pro vyhledávání nejbližšího souseda ve stromu. Při shlukování dat do jednoho vektoru je důležité, aby se jejich pořadí shodovalo s pořadím jednotlivých složek v již sestaveném stromě.

V momentě, kdy je k dispozici nějaký seskupený bod, se kterým lze prohledávat zkonstruovaný strom, je možné provést výpočet jeho fáze a kvality. Za účelem získání fáze je vhodné rozšířit strukturu uzlů obsažených ve stromě o atribut nesoucí informaci o fázi hodnoty bodu uzlu. Správnost fáze uzlu vychází z předpokladu, že strom je postaven z bodů, které reprezentují jeden pohyb. Celý tento pohyb se konstruuje z datových souborů. Ty nesou mimo jiné pro každý záznam informaci o čase (v čase x pohyb počínaje a v čase y pohyb končí). Body jednoho pohybu byly získány ručním označením na základě animace veškerých dat a vyjmuty dle časového intervalu. Jednotlivé časy v souborech jsou však různé, protože snímací hardware není časově synchronizován. Tento problém lze vyřešit například zprůměrováním časů, protože se liší v řádech milisekund. Z tohoto faktu vyplývá, že při konstrukci (učení) stromu lze navíc ke každému seskupenému bodu přidat informaci o

čase, kterou lze jednoduchým způsobem namapovat na interval od nuly do jedné a získat tak fázi pohybu bodu každého uzlu stromu. Poté lze výslednou fázi hledaného bodu určit podle fáze bodu nejbližšího nalezeného uzlu k danému hledanému bodu.

Oproti tomu při získávání kvality si lze pro jednoduchost představit postavený strom jako **tunel** o poloměru definovaném konstantou nastavení programu. Příklad tunelu je popsán obrázkem 3.1, který je pořízen pomocí aplikace **TunnelTest** (obsažené v příloze), jež vznikla v rámci tohoto projektu za účelem demonstrace vizualizace naměřených bodů. Obrázek zachycuje body polohy pohybu ruky zobrazené relativně k počátku a obklopené tunelem.



Obrázek 3.1: Vizualizace tunelu nad daty pohybu ruky

Kvalita se poté určí jako vzdálenost hledaného bodu od nejbližšího nalezeného bodu uzlu, přičemž se musí zohlednit hranice tunelu. Jednoduše řečeno, čím vyšší bude vzdálenost nalezeného uzlu od hledaného bodu, tím nižší bude výsledná kvalita. Kvalita se může opět měnit lineárně až do vzdálenosti hranic (poloměru) tunelu (respektive do vzdálenosti definované konstantou).

Základním problémem konceptu tunelu jsou trénovací data, podle kte-

rých je tunel postaven. Jedná se totiž o konkrétní data vzešlá z konkrétního měření a jejich umístění v prostoru z principu neodpovídá aktuálnímu umístění snímaného uživatele. Je proto důležité, aby ještě před samotným zkonstruováním stromu, byla provedena transformace stromu na aktuální pozici uživatele. Pro efektivnější transformaci tunelu je potřeba jej reprezentovat surovými daty získanými z měření. Až při trvalém usazení uživatele na jednom místě, tedy před samotnou analyzovanou abdukcí paže, postavit z těchto dat strom. Tunel, skládající se ze surových dat, má značné výhody v urychlení procesu transformace dat a umožňuje možnou vizualizaci tunelu uživateli při jejich navzorkování. Tunel by měl být schopen pohybovat s uživatelem v reálném čase bez znatelné odezvy. Ta by při transformaci seskupených 24-rozměrných dat byla z důvodu výpočetní náročnosti znatelná. Schopnost pohybu by ovšem bylo rozumné ponechat na výpočetní jednotce grafické karty. Jednou z možností, jak toho stavu docílit, je publikování pouze navzorkovaných surových dat, na jejichž základě se v enginu Unity vizualizuje nějaký model tunelu. S modelem tunelu je možné jakkoliv pracovat v enginu Unity, nezávisle na datech obsažených v knihovním tunelu. Před samotnou analýzou postačí znát počáteční bod jednoho ze snímaných senzorů, podle kterého se transformují všechna surová data. Následně se vytvoří strom. Tímto způsobem se během uživatelské interakce s tunelem, z pohledu knihovního tunelu, provede pouze jedna transformace dat. Tento stav je žádoucí. Koncové sestavení stromu je pro jednoduchost vhodné namapovat na stisk nějakého tlačítka. Při pozdějším otestováním této funkcionality lze sestavení automatizovat (například na základě dat získávaných z oblasti hrudníku), protože díky jejich relativně malé proměnlivosti při sezení lze nejsnáze určit prostorový pohyb uživatele a odhadnout tak usazení.

3.2 Implementace řešení

Kapitola implementace řešení si klade za cíl čtenáři představit pomocné aplikace/knihovny a technické vlastnosti výsledného modulu. Ten řeší analýzu abdukce paže. Dále je stručně popsané prostředí, v němž je modul implementován, a prostředí, do kterého je modul integrovatelný. Nejdříve se objasní technické detaily projektu a následně celý koncept řešení vytvořený v rámci této práce v podkapitole 3.2.1. Dále jsou k nahlédnutí implementované pomocné metody a struktury v podkapitole 3.2.2, které usnadňují čitelnost a přehlednost výsledného zdrojového kódu. Nakonec se v textu práce zaměříme na manipulaci s naměřenými daty v rámci tunelu. Pro usnadnění interakce s naměřenými daty byla pro účely práce vytvořena kolekce, která umožňuje

jejich transformaci. Tato kolekce, společně s principem transformace celého tunelu, je popsána v podkapitole 3.2.3

Jediný požadavek na výsledný praktický produkt je integrovatelnost do prostředí enginu Unity. Unity je herní engine (respektive platforma) navržený pro vývoj 2D a 3D her, potažmo čehokoliv týkajícího se práce s 2D či 3D grafikou (animace, hry, filmy ...) od společnosti Unity Technologies. Engine Unity má jasně daná omezení, která přímo dopadají na implementaci onoho modulu.

Jedním z nich je použitý programovací jazyk. Do určité verze Unity lze použít více programovacích jazyků. I přesto jsem se rozhodl na základě článku [5] zmínit pouze programovací jazyk *c♯* (celým slovem **c sharp**). Ostatní jazyky budou v budoucích verzích označeny jako zastaralé. Právě jazyk *c♯* je zvolen jako nativní. Jedná se o objektově orientovaný jazyk od společnosti Microsoft, který je však v rámci enginu Unity interpretován jako jazyk skriptovací.

Dalším rozhodnutím, jež je zapotřebí učinit, je způsob integrace výsledného modulu do prostředí enginu Unity. Funkcionalita modulu může být do enginu Unity zavedena buď formou knihovny, která bude následně použita v interpretovatelném skriptu, nebo může být obsažena přímo ve skriptu, který dokáže engine Unity interpretovat. Jelikož engine Unity podporuje skriptovací runtime platformy **.NET verze 4.x** a API kompatibilitu s **.NET Standard 2.0**, lze použít modul jako externí knihovnu v současné verzi platfomy .NET, a tím pádem dodává této knihovně možnost integrovatelnosti do jakéhokoliv systému či aplikace podporující platformu .NET verze stejné či vyšší, než ve které je knihovna napsána.

3.2.1 Struktura projektu

Jak již bylo zmíněno výše, výsledný modul bude implementován formou knihovny funkcí, která je vytvořena nad platformu .NET. Za tímto účelem vznikl soubor řešení **AAPD_Solution.sln** (kde AAPD je zkrácenina anglických slov **A**utomatic **A**rm **P**osition **D**etection, neboli Automatická Detekce Polohy Paže), který je specifický pro vyvíjení jakýchkoliv aplikací nad platformou .NET. Celý takto vzniklý projekt je verzován v repozitáři **AAPD** [11] a je součástí přílohy. Celé řešení je vytvořeno ve vývojovém prostředí **Microsoft Visual Studio 2017**. Obsahuje doposud pět projektů. Z toho tři projekty jsou důležité pro kontext naší práce. Jedná se o projekty:

- **AAPD_Core** - Class Library (.NET Standard) projekt obsahující logiku výsledného modulu pro analýzu pohybu paže.

- **AAPD_Data** - Class Library (.NET Standard) projekt obsahující pomocné datové struktury a rozšíření.
- **KDTree** - Class Library (.NET Standard) projekt obsahující implementaci k-dimensionálního stromu.

Class Library projekt je pojmenování knihovny funkcí a .NET Standard je sada API, které .NET framework implementuje (označuje tedy do jisté míry kompatibilitu). Zbylé dva projekty jsou převážně testovací, potažmo integrační. Jedná se o testovací projekt **Tests**. Obsahuje jednotkové testy výsledného modulu a k-dimensionálního stromu. V rámci zkušební integrace a možnosti analyzování výkonu programem Visual Studio vznikl konzolový projekt **AAPD_ConsoleApp**.

V této kapitole je dále představen projekt AAPD_Core a určité závislé datové struktury z AAPD_Data, jelikož jejich znalost je klíčová pro správná použití výsledku práce. Zbytek knihovny AAPD_Data je rozebrán v podkapitole 3.2.2 a 3.2.3. Knihovnu nesoucí implementaci k-dimensionálního stromu není třeba blíže komentovat, protože popis implementovaných algoritmů byl vysvětlen v teoretické části a při použití výsledného modulu s ní uživatel explicitně nepřijde do kontaktu.

Knihovna AAPD_Core, jak již název napovídá, symbolizuje jádro tohoto projektu. Dle jmenných prostorů je členěna na **Infrastructure** a **Processors**. Mezi knihovní závislosti (anglicky **dependencies**) patří zbylé dvě klíčové knihovny, jmenovitě AAPD_Data a KDTree.

Jmenný prostor Infrastructure uchovává infrastrukturu celého řešení. V současné době obsahuje jedno rozhraní nesoucí název `IDataProcessor<T>`. Toto rozhraní je navrženo s genericky typovaným parametrem `T` a definuje tři metody. Z hlediska použití je první z nich metoda `Build(...)`, která přijímá pole datové kolekce `DataCollection<T>` na jehož základě se postaví k-dimensionální strom. Další z nich je beznávratová metoda `Input(...)`, do které vstupuje právě jeden genericky typovaný parametr `T` symbolizující vstupní data. Metoda `Input(...)` má za účel shromažďovat příchozí data, která později mají sloužit pro vyhotovení výsledné analýzy poslední beznávratové metody `Process(...)`. Při shromažďování ponecháváme pro menší paměťové nároky pouze poslední data dílčích typů. Použití v konkrétních implementacích (pokud to dává smysl) by se mělo počínat metodou `Build(...)`, která postaví strom. Poté je třeba získat potřebná data metodou `Input(...)` a jejich analýzu provádět pomocí `Process(...)`.

Metoda `Process(...)` je sice typově beznávratová, ale obsahuje dva parametry označené klíčovým slovem `out`. To značí nastavení hodnot vstupních parametrů v rámci metody. Po zavolání a korektním dokončení me-

tody budou oba dva parametry naplněny konkrétními hodnotami. Metoda Process(...) je navržena pro získání výsledné analýzy pohybu paže formou fáze (parametr phase) a kvality (parametr quality).

I přesto, že všechny tyto metody jsou navržené pro použití nezávisle na sobě, je z logiky věci žádoucí, nejprve explicitně postavit strom (je-li tomu potřeba) metodou Build(...). Dále je vhodné nashromáždit požadovaná data skrze metodu Input(...) a teprve poté se dotazovat na aktuální fázi a kvalitu metodou Process(...).

Obecně implementace rozhraní IDataProcessor<T> může zpracovávat jakákoliv data třídy Data či jejich potomků. Není vyloučeno, že by takto popsaný datový zpracovávač existoval pouze jeden. Proto vznikl jmenný prostor **Processors**, který obsahuje současné i potencionální budoucí implementace rozhraní IDataProcessor<T>. Nyní existují dvě implementace tohoto rozhraní.

Jako první implementace vznikl FileDataProcessor, který zpracovává textový soubor obsahující ručně označenou fázi a kvalitu množiny testovacích dat určených pro učení. Jak je zmíněno v podkapitole 3.1.1, jedná se o data ze snímacího systému virtuální reality. Metody Build(...) a Input(...) nejsou implementované, jelikož vstupní soubor s daty je předáván v konstruktoru a není potřeba stavět strom ani jakýmkoliv způsobem shromažďovat data. Metoda Process(...) vrací na zavolání interpolovanou fázi a kvalitu na základě hodnot vstupního souboru. Při každém požadavku o fázi a kvalitu se posune interní údaj o zpracovávaném čase o 0.1 vteřiny. Na základě tohoto času se data interpolují. Počátek je nastaven na první hodnotu vstupního souboru. Při překročení posledního možného času se interní časovač zresetuje opět na počátek.

Druhou a poslední implementací rozhraní je DataProcessor, který je součástí výsledného tunelu. DataProcessor je typovaný na model *Data* definovaný v knihovně AAPD_Data. Jedná se o konkrétní a jedinou implementaci analýzy abdukce paže popsanou v podkapitole 3.1.2.

Různé implementace rozhraní IDataProcessor<T> se od sebe liší především typem parametru T. Tento parametr je konstrukčně omezen typem třídy Data či jejích potomků a možností jejich instancování. Třída Data reprezentuje nějakou datovou strukturu zapouzdřující informace o snímaných datech. V současné době se jedná o data z jednoho konkrétního snímaného zařízení (především typ dat a jejich vektorový popis). Datový model je k nalezení ve jmenném prostoru **Infrastructure** knihovny AAPD_Data. Tento model je navržen jako schránka pro data z jednoho snímaného zařízení či senzoru a obsahuje metodu (spíše než označení metoda pro tento konkrétní člen se v kontextu jazyka c-sharp používá termín **vlastnost** respektive ang-

licky **property**) `DataType`. Její návratovou hodnotou je hodnota výčtového typu `DataType`. Ta určuje z jaké části těla pořizovaná data `Data` jsou.

Výčtový typ `DataType` je rovněž součástí knihovny AAPD_Data a je definován v jmenném prostoru **Enums**. Nabývá následujících hodnot:

- `Head` - Označení pro oblast hlavy.
- `Chest` - Označení pro oblast hrudi.
- `Forearm` - Označení pro oblast ruky.
- `Arm` - Označení pro oblast paže.
- `Unknown` - Označení pro neznámou oblast.

Z nutnosti správného transformování surových dat, respektive přípravy pro zpracování nějakou implementací rozhraní `IDataProcessor<T>`, je vytvořena třída `Tunnel<T>`, představující výsledný tunel obalující logiku analýzy. Třída `Tunnel<T>` tvoří nedílnou součást jmenného prostoru **Infrastructure** knihovny AAPD_Core. Generický parametr `T` nabývá stejných hodnot jako v implementacích rozhraní `IDataProcessor<T>`. Tunel `Tunnel<T>` charakterizuje především vlastnost *Processor*, jež je veřejná a dovoluje tak dynamicky měnit používanou implementaci rozhraní `IDataProcessor<T>`. Z potřeby zjištění, zda použitý zpracovávač je schopen analyzování (byla u něj provedena metoda `Build(...)`), vlastní tunel logickou vlastnost *CanProcess*. Tunel je sám o sobě vytvořitelný pouze z pole kolekcí `DataCollection<T>` symbolizujících naměřená data. Z těchto vstupních dat se staví strom ve zpracovávači. Hlavním pilířem tunelu se stala požadovaná transformace těchto dat. Transformace je umožněna metodou `TransformTo(...)`. Metoda vezme všechna surová data a provede transformaci relativně k bodu, který vstupuje jako parametr. Navíc za účelem vizualizace obsahuje tunel vlastnost *RawData*, jež pouze vrací kolekci navzorkovaných surových dat. Hustota vzorkování je ovlivnitelná vlastností *SampleDensity*, na jejímž základě se vybírá zpravidla *n-tý* prvek. Tunel se využije pro výsledné analyzování abdukce paže. Základním předpokladem pro správné analyzování, je mít správně transformovaná data a naplněnou vlastnost *Processor* nějakou vytvořenou implementací rozhraní `IDataProcessor<T>`.

Mezi infrastrukturu knihovny AAPD_Core lze zařadit v poslední řadě třídu publikující nastavení `Settings`. Prozatím obsahuje klíčové vlastnosti `Coefficients` a `TunnelRadius`. Vše je tvořenou statickou formou, a tudíž nastavení těchto vlastností můžeme provádět kdykoliv. Vlastnost s názvem `Coefficients` je slovník, jehož klíčem se stal typ snímaného zařízení

DataType a hodnotou pole desetinných čísel. Ta se používají při slučování poloh a rotací jednotlivých snímaných senzorů. Vlastnost TunnelRadius určuje aktuální použitý poloměr tunelu.

S takto připravenou infrastrukturou projektu implementujeme jakékoliv řešení analýzy průběhu abdukce paže. Třída Data dává sama o sobě volnost uživateli implementovat různé zdroje dat dle potřeby využitelných v jakékoliv implementaci rozhraní IDataProcessor<T> použitelného v implementovaném tunelu Tunnel<T>.

3.2.2 Pomocné struktury a rozšíření

Tato podkapitola nastiňuje možnosti použití implementovaných pomocných struktur a možnosti rozšíření obsažených v knihovně AAPD_Data. Ty značně usnadňovaly průběh vývoje. Představené objekty a funkcionalitu lze používat v jakémkoliv kontextu při integraci výše zmíněné knihovny.

Základem pro použité pomocné datové struktury se stal jmenný prostor **Models**. V současné verzi knihovny se tento jmenný prostor skládá z datových modelů Vector3D a Matrix3x3. Zmíněné modely jsou použité napříč knihovnami AAPD_Data a AAPD_Core. Jejich použití v AAPD_Data je především ve jmenném prostoru **Infrastructure**. Jmenný prostor Infrastructure obsahuje model dat Data a kolekci DataCollection<T>, jejíž generický parametr T je vymezen hierarchickou relací k modelu Data a je instancovatelný.

Datový model Data již byl představen v podkapitole 3.2.1. Navíc doplňme, že obsahuje metodu IsEmpty() s logickým návratovým typem. Jak již název napovídá, metoda testuje jestli vytvořený model naplňují relevantní hodnoty či nikoliv. Dále definuje metodu ToArray(...), která vrací nové pole reálných čísel sestavených postupně z dílčích datových vektorů. Vstupním parametrem je pole koeficientů. Díky nim se přenásobují jednotlivé složky. Za tímto účelem je vhodné použít koeficienty *Coefficients* v nastavení Settings. Tato činnost není nezbytně nutná. Datové vektory jsou obsaženy ve vlastnosti modelu *Vectors*, která je reprezentována polem struktur Vector3D. Metody IsEmpty() a ToArray(...) jsou využity v analyzátoru *DataProcessor*. Na základě metody IsEmpty() se v metodě Process(...) rozhoduje zda se shromažďují data úplná nebo ne. Poté se pomocí metody ToArray(...) převádí a sjednocují nashromážděné vektory polohy a rotace do potřebných 24-rozměrných vektorů. S tímto modelem je úzce spjatá kolekce *DataCollection*$<T>$, která je detailněji popisována v podkapitole 3.2.3.

Struktura Vector3D reprezentuje vektor reálných čísel v prostoru o třech dimenzích. Obsahuje přetížené operátory sčítání, odčítaní a násobení, které

na výstupu vytváří nový vektor. Dále pro efektivnější použití porovnání implementuje struktura rozhraní IComparable<Vector3D>. To je typické na platformě .NET pro symbolizaci porovnávatelných objektů. Navíc se přidávají do struktury statické metody (vlastnosti) vytvářející nové struktury vektoru (prázdného a po jednotlivých složkách). Takto navržený vektor je použit i v datovém modelu Data.

Struktura Matrix3x3 představuje matici reálných čísel o rozměrech 3x3. Matice je implementovaná polem polí. Kromě vytvoření matice přes uživatelem zadané hodnoty, jde matici vytvořit přes statické volání metody CreateRotation(...) nebo metody CreateFromEulerAngles(...). Metoda CreateRotation(...) vytváří matici rotace dle parametrů vektoru Vector3D (symbolizující osu otáčení) a reálného čísla (symbolizující úhel rotace). Metoda CreateFromEulerAngles(...) vytváří matici rotace eulerových úhlů z vektoru popisujícího eulerovy úhly a řetězci obsahujícím pořadí os (např.: "XYZ"). Matice se používá především při rotaci dat. Tuto problematiku popisujeme v podkapitole 3.2.3. Nad vektorem i maticí jsou postavena rozšíření, kterým bude věnován zbytek této podkapitoly.

Rozšíření mají svůj vlastní jmenný prostor **Extensions** a v podstatě se jedná o metody volatelné přímo nad proměnnou konkrétního datového typu. Doposud jsou vytvořené rozšíření:

- Vector3DExtension - rozšíření práce se strukturou Vector3D
- Matrix3x3Extension - rozšíření práce se strukturou Matrix3x3
- FloatExtension - rozšíření práce s datovým typem float jazyka c-sharp
- DoubleExtension - rozšíření práce s datovým typem double jazyka c-sharp
- CollectionExtension - rozšíření práce s kolekcemi

Rozšíření obsahují pomocné metody nad konkrétními datovými typy. Ty ale do nich nelze přímo integrovat, protože rozšíření musí být definována ve statické třídě.

Použitelné rozšíření nad vektorem Vector3D v této práci reprezentuje výpočet skalárního násobku dvou vektorů a velikosti jednoho vektoru. Skalární násobek dvou vektorů je získatelný pomocí metody DotProduct(...), do níž vstupuje jako parametr druhý vektor. To je používáno především při maticovém násobení zmíněném dále. Oproti tomu velikost vektoru je přímočará záležitost dosažitelná metodou SqrtMagnitude(). Využívána je zejména při

transformaci dat v tunelu, kdy jednotlivé parametry transformace jsou neznámé. Jedním krokem při jejich dohledávání je zjišťování úhlu, který svírají dva vektory. Z podstaty řešení úlohy hledání úhlu se tedy využívají mimo jiné i velikosti jednotlivých vektorů.

Nad maticí Matrix3x3 jsou definována rozšíření násobení Multiply(...) (jak maticí Matrix3x3, tak vektorem Vector3D) a získání eulerových úhlů z matice pomocí metody ToEulerAngles() (která vrací vektor Vector3D). Násobení i převody jsou používány především při práci s měřenými daty, respektive jejich rotacemi.

Datový typ float bylo nutné rozšířit o funkcionalitu převodů reálného čísla mezi radiány ToRadians() a stupni ToDegrees(), především kvůli snadnější práci s čísly reprezentující úhly. Jedno z nejdůležitějších rozšíření nese název Map(...). Díky němu lze mapovat reálnou hodnotu, nad kterou je voláno ze zadaného intervalu na jiný zadaný interval. Přemapování je klíčovým při konstrukci (učení) stromu a následně korektním určením fáze pohybu popsané v podkapitole 3.1.2.

Pro datový typ double bylo potřeba vytvořit pouze jedno rozšíření nesoucí název Distance<T>(...), které by určovalo vzdálenost mezi dvěma obecně n-dimensionálními body. Body jsou popsány seznamy typů T. Aby bylo určování vzdálenosti efektivní, vstupuje do metody parametr funkce určení rozdílu dvou typů T. Funkce je potřebná. Při překladu se neví nic o typech parametru T, a tudíž by dynamické přetypování za běhu programu mělo za následek vysoký nárůst neefektivnosti. Dynamické přetypování je velice náročná operace.

Poslední rozšíření byla provedena nad vybranými kolekcemi. Pro zjednodušení čitelnosti zdrojového kódu vznikly metody Fill<T>(...) obecně nad polem typů T. Metody naplňují pole buď výchozí hodnotou, nebo polem hodnot převzatých jako vstupní parametr. Používají se při převodu datových vektorů modelu Data na pole pomocí metody ToArray(...). Poslední potřebnou dodatečnou funkcionalitou nad kolekcemi je jejich seskupení do jednoho pole. Seskupení popisujeme pomocí metody Join<T>(). Ta je volatelná nad seznamem polí. Rozšíření vytvoří z původních vstupních polích nové pole typované parametrem T. Funkce je nezbytná pro slučování dat například ve zpracovávači DataProcessor.

3.2.3 Manipulace s naměřenými daty

Data měřená virtuální realitou HTC Vive jsou daty relativními vůči základovým stanicím (viz 2.1.2). Zachovávají svou informaci při pohybu i při jejich hromadné transformaci. Pokud máme nějaký záznam o pohybu a otočíme

ho například o 90 stupňů, potom by měla i takto otočená data produkovat stejné výsledky analýzy pohybu jako data originální (samozřejmě za předpokladu přizpůsobení pohybu uživatele).

Za účelem testování výsledného analyzátoru pohybu je žádoucí, abychom dokázali načtená data transformovat. Funkcionality translace a rotace jsou definovány v rámci datové kolekce `DataCollection<T>`. Translace je uskutečněna metodou `Translate(...)` a rotace metodou `Rotate(...)`. Jediným parametrem metody `Translate(...)` je vektor `Vector3D` popisující posun. Do metody `Rotate(...)` vstupuje úhel rotace, osa otáčení a bod, kolem něhož se uskutečňuje výsledná rotace. Obě metody jsou beznávratové. Zmíněné transformace manipulují přímo s načtenými daty. Data jsou získatelná vlastností kolekce *Items*. Rotace je přizpůsobena enginu Unity, kde se používá levotočivý souřadný systém a pořadí aplikace úhlů rotace je ZXY.

Kolekce má generický parametr `T`, jenž musí být instancovatelný a typově příbuzný modelu `Data`. Kolekce navíc obsahuje mechanismus načítání a ukládání dat ze souborů. Načítání funguje díky metodě `Load(...)`, která vyžaduje parametry názvu souboru s daty a typem použitého snímače. Nad vzniklou instancí můžeme uložit data do souboru opět metodou `Save(...)`. Souborová cesta je vstupním parametrem této metody. Samotná kolekce obsahuje navíc statickou metodu `LoadData(...)`. Ta vytvoří instanci kolekce z datového souboru pomocí metody `Load(...)`. Všechny metody, kromě metody statické, jsou *virtuální* a lze z každé z nich implementovat vlastní logiku v potomkovi.

Vytvořená kolekce je obrazem pouze jednoho datového souboru. V praxi to znamená, že lze vytvořit pouze kolekci odpovídající jednomu snímanému zařízení. Pro správné otočení celého prostoru je nutné takto načíst všechny datové soubory popisující pohyb a stejné akce provést nad všemi kolekcemi.

Výše zmíněné transformace nad kolekcemi jsou použité především v transformaci tunelu `Tunnel<T>` metodou `TransformTo(...)`. Tato metoda transformuje všechna načtená surová data, která jsou reprezentována pomocí kolekcí `DataCollection<T>`. Dílčí kolekce se transformují k bodu. Ten je získán jako parametr a obsahuje typ oblasti snímaného senzoru a vektor polohy. Na základě typu se vybere první prvek z kolekce obsahující stejné typy. Podle vybraného prvku se najdou parametry transformace. Ty se poté aplikují na všechny kolekce. V praxi to znamená, že lze transformovat celý tunel na základě polohy jakéhokoliv jednoho snímaného senzoru a nezávisle na tom vizualizovat tunel na základě kolekce libovolného typu. Vlastnost *RawData* zpřístupňující vzorkovaná data pro vizualizaci vzorkuje kolekce vždy při jejím zavolání. Vzorkovaná data jsou vždy aktuální s transformovanými daty a můžeme podle nich tunel vizualizovat. Avšak způsob vizualizace na základě

metody `TransformTo(...)` se nedoporučuje. Mohl by být vysoce neefektivní při velké množině dat, jelikož pro transformování vizualizovaného tunelu není žádoucí transformovat všechna data ale pouze data navzorkovaná. Navíc celá takto popsaná transformace se počítá na výpočetní jednotce procesoru.

3.3 Dosažené výsledky

Průběžné zhodnocení kvality výsledného analyzátoru nelze provést důkladně, protože celý projekt je zatím ve fázi vývoje. Nicméně je možné zhodnotit dosavadní analyzátor a jeho dílčí části. V rámci výše představeného řešení, zahrnutého v příloze, existuje projekt **Tests**, který obsahuje jednotkové testy. Testy vznikaly především pro kontrolu klíčových přidávaných funkcionalit a z časových důvodů zdaleka nepokrývají všechny možnosti užití. Testován byl především k-dimensionální strom a tunel.

U k-dimensionálního stromu bylo nutné ověřit, zda metoda konstrukce stromu staví strom korektně a jestli implementovaný algoritmus hledání nejbližšího souseda produkuje výsledek v rozumném čase. Samotná konstrukce je ověřována na základě dvourozměrných dat, u nichž předem známe výslednou stromovou strukturu. Algoritmus hledání nejbližšího souseda je porovnáván s algoritmem, který hledá nejbližší bod metodou hrubé síly. Samotné hledání oběma přístupy je provedeno 100x, přičemž jednotlivé časy běhu jsou zaznamenávány. Konečné zjištění úspěšnosti je provedeno na základě mediánu zaznamenaných časů jednotlivých algoritmů. Pro představu tabulka 3.1 porovnává časy běhů obou algoritmů na základě různorodé datové množiny.

Datová množina		Algoritmus	
velikost	dimenze	hrubá síla	nejbližší soused
58681	3	14.577	0.039
14433	24	11.300	35.642

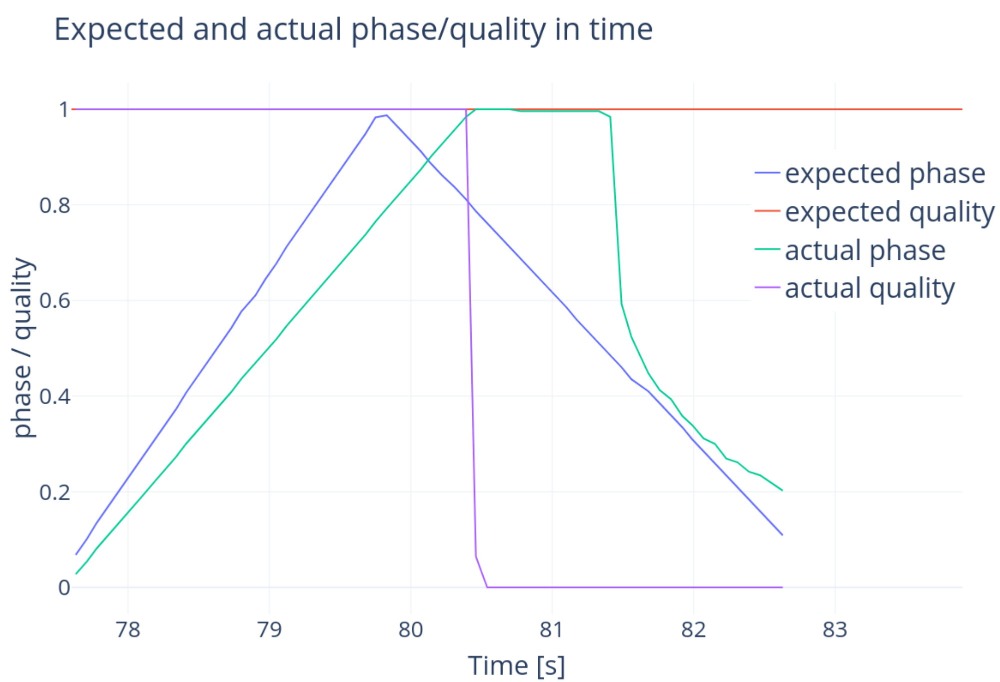
Tabulka 3.1: Porovnání časů [ms] běhů závislých na datové množině

Lze si povšimnout vlastnosti stromu závislé na velikosti dimenze a počtu vzorků, při které časová složitost algoritmu prohledávání degraduje z logaritmické na lineární. Konkrétní časy nejsou příliš zajímavé, protože jsou závislé na použitém hardwaru. Nicméně pro demonstraci efektivní implementace jsou řádově postačující při rozumně zvolené dimenzi a počtu vzorků. Navíc je vhodné podotknout, že v produkční verzi je zapotřebí použití mnohem silnějšího hardwaru, než z kterého jsou tato data pořízena. Zařízení HTC Vive má vysoké nároky, co se síly výpočetních jednotek týče.

V časové degradaci efektivnosti k-dimensionálního stromu hrají velkou roli také hledaná data. Pokud je nejbližší nalezený uzel některý z vrcholů, potom algoritmus hledání z principu funguje v logaritmické složitosti, protože se vrchol najde již v dopředném průchodu. Obdobně se dá očekávat dobrá časová efektivita i při hledání uzlů blízkým vrcholům. Nejhorším případem jsou ovšem body vzdálené, kde se skutečně časová efektivita algoritmu hledání zhoršuje. Ve výsledku algoritmus dosahuje požadované efektivnosti.

U tunelu jsou testovány především jednotlivé implementované zpracovávače a možnost jeho transformace. Zpracovávače jsou testovány zvlášť nad stejnou množinou dat, ze kterých jsou určeny. U implementovaného zpracovávače `FileDataProcessor` je testována správnost kontinuálního vracení dat. U zpracovávače `DataProcessor` je testována korektnost fáze a kvality. Poslední prozatím potřebnou vlastností vhodnou k otestování je dynamická výměna zpracovávačů v tunelu. Test je proveden nezávisle na výsledcích analýzy jednotlivých zpracovávačů, jelikož nejsou pro výsledek testu relevantní (zajímá nás pouze možnost výměny a následné bezchybné pokračování v provozu).

Nakonec byl proveden test, jehož výsledek je zachycen grafem 3.2. Graf zobrazuje hodnoty fáze a kvality dat, která popisují třetí správný pohyb ze všech naměřených pohybů. Vyobrazený průběh pohybu začíná zhruba v sedmasedmdesáté vteřině a končí kolem osmdesáté třetí vteřiny, přičemž vrchol abdukce paže je zhruba v osmdesáté vteřině. Očekávané (**expected**) hodnoty se získávají z ručně označených dat. Skutečné (**actual**) hodnoty produkuje implementovaný analyzátor `DataProcessor`. Analyzátor byl naučen pouze na abdukci pohybu (směrem nahoru) a nikoliv na pohybu celém (tedy směrem nahoru a zpět dolu). Za tohoto předpokladu lze vidět relativní správnost určení fáze a kvality na abdukci paže. Oproti tomu při addukci paže (pohybu směrem dolu) je zaznamenán prudký pokles kvality a relativní nepřesnost určované fáze. Zdánlivě nepřesné výsledky jsou nejspíše způsobeny nastavením koeficientů vlivů dílčích snímaných oblastí. Koeficienty byly použity výchozí. Neměly tedy vliv na modifikaci dat. Patřičná úprava koeficientů z časových důvodů není zahrnuta do této práce a v hledání ideálních kombinací se bude nadále pokračovat.



Obrázek 3.2: Graf očekávaných a skutečných hodnot fáze a kvality v čase

4 Závěr

Výsledek této práce je pouze dílčím celkem aplikace, která je stále ve fázi vývoje. Hotové řešení bude nejspíše podléhat potencionálním vzniklým požadavkům na úpravu. Nicméně v současné době je analyzátor postavený na principu tunelu použitelného při správném nastavení. To je výchozí především z důvodu minimálního času alokovaného na testování ve fázi vývoje. Na druhou stranu nastavení bylo navrženo tak, aby se dalo měnit i dynamicky za běhu aplikace. V pozdější fázi testování nebude potřeba měnit zdrojový kód nastavení analyzátoru.

Samotný analyzátor je postaven na formě klasifikace pomocí prohledávání **k-dimensionálního stromu**, který je obalen takzvaným **tunelem** použitým pro načtení surových dat a jejich možnou transformaci. Transformace tunelu je klíčová před zahájením analýzy. Musí se synchronizovat pozice naučených a vstupních dat.

Analyzátor je připraven na zdroj dat ze zařízení virtuální reality **HTC Vive**. Zdrojem jsou čtyři snímaná zařízení snímající oblasti hlavy, hrudi, ruky a paže. Tato data se sjednotí do jednoho 24-rozměrného bodu, pomocí kterého se poté prohledává postavený strom tunelu.

Výsledná analýza produkuje veličiny **kvality** a **fáze** pohybu normované na interval od nuly do jedné. Kvalita abdukce je inicializována hodnotou jedna, jež klesá s horšící se kvalitou pohybu. Fáze abdukce paže naopak začíná hodnotou nulovou a zvyšuje se společně s abdukcí.

Přílohy

Kompaktní disk obsahující následující adresářovou strukturu:

- Adresář **AAPD_Solution** obsahující zdrojové kódy dílčích knihoven.
- Adresář **Release** obsahující přeložené knihovny a spustitelné aplikace.
- Adresář **TestData** obsahující měřená data.
- Adresář **TunnelTest** obsahující Unity projekt pro vizualizaci naměřených bodů.
- Adresář **UnityAnimacniTest** obsahující Unity projekt pro animaci a markování naměřených dat.
- **AAPD_BakalarskaPrace** - pdf dokument s touto prací.

Literatura

[1] Ahire, J. B. *The Artificial Neural Networks Handbook: Part 4* [online]. 2018. [cit. 2018/11/11]. Obrázek struktury neuronu. Dostupné z: https://cdn-images-1.medium.com/max/2600/1*WRG_Re8vGVuHDYigtq2IBA.jpeg.

[2] Buckley, S. *This Is How Valve's Amazing Lighthouse Tracking Technology Works* [online]. Gizmodo Media Group, 2015. [cit. 2015/05/19]. Princip HTC vive snímání prostředí. Dostupné z: https://gizmodo.com/this-is-how-valve-s-amazing-lighthouse-tracking-technol-1705356768?utm_medium=sharefromsite&utm_source=gizmodo_copy&utm_campaign=bottom.

[3] Cohen, R. *My Favorite Algorithm: Linear Time Median Finding* [online]. 2018. [cit. 2018/02/15]. Metoda Medians of median. Dostupné z: https://rcoh.me/posts/linear-time-median-finding/.

[4] Durčák, I. P. *Neuronové sítě a princip jejich fungování* [online]. Verlag Dashöfer, nakladatelství, spol. s r. o., 2017. [cit. 2017/09/08]. Princip fungování neuronových sítí. Dostupné z: https://www.napocitaci.cz/33/neuronove-site-a-princip-jejich-fungovani-uniqueidgOkE4NvrWuNY54vrLeM67OeFNQh552V

[5] Fine, R. *UnityScript's long ride off into the sunset* [online]. Unity Technologies, 2017. [cit. 2017/08/11]. Programovací jazyky v Unity. Dostupné z: https://blogs.unity3d.com/2017/08/11/unityscripts-long-ride-off-into-the-sunset/.

[6] Fox, E. *Complexity of NN search with KD-trees* [online]. Coursera Inc., 2019. Efektivnost k-dimensionálního stromu. Dostupné z: https://www.coursera.org/lecture/ml-clustering-and-retrieval/complexity-of-nn-search-with-kd-trees-BkZTg.

[7] Frank, J. *Repozitář aplikace pro Sensorstream IMU+GP* [online]. 2019. [cit. 2019/04/19]. Repozirář pomocná aplikace pro získávání dat ze senzorů. Dostupné z: https://bitbucket.org/frankkuba/vr/src.

[8] Lorenz, A. *Sensorstream IMU+GPS* [online]. Google Commerce Ltd, 2013. [cit. 2013/02/08]. Aplikace pro získávání dat ze senzorů. Dostupné z: https://play.google.com/store/apps/details?id=de.lorenz_fenster.sensorstreamgps.

[9] Obermaier, Z. *Srovnání Vive a Vive Pro* [online]. EMPRESA MEDIA, a.s., 2018. [cit. 2018/06/11]. Srovnání Vive a Vive Pro. Dostupné z:

https://pctuning.tyden.cz/multimedia/hry-a-zabava/52075-htc-vive-pro-jasne-nejlepsi-set-pro-virtualni-realitu?start=3.

[10] *Neural network* [online]. Wikipedia, 2019. [cit. 2019/04/25]. Obrázek vrstev neuronové sítě. Dostupné z: https://en.wikipedia.org/wiki/Neural_network#/media/File:Neural_network_example.svg.

[11] Poór, V. *AAPD* [online]. 2019. [cit. 2019/04/27]. Repozirář projektu. Dostupné z: https://bitbucket.org/Custom666/aapd.

[12] *Unity User Manual* [online]. Unity Technologies, 2019. [cit. 2019/04/15]. Engine Unity. Dostupné z: https://docs.unity3d.com/Manual/.

[13] V, A. S. *Understanding Activation Functions in Neural Networks* [online]. 2017. [cit. 2017/03/30]. Aktivační funkce v neuronové síťi. Dostupné z: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0.

[14] Václav Matoušek, P. K. *Klasifikace, rozpoznávání a shlukování* [online]. 2019. [cit. 2019/03/23]. Umělá inteligence a rozpoznávání. Dostupné z: https://www.kiv.zcu.cz/studies/predmety/uir/predn/P4/FThema4-2neww.pdf.

[15] *Backpropagation* [online]. Wikipedia, 2019. [cit. 2019/05/16]. Wikipedia zpětné propagování. Dostupné z: https://en.wikipedia.org/wiki/Backpropagation.

[16] *HTC Vive* [online]. Wikipedia, 2019. [cit. 2019/05/08]. Wikipedia HTC Vive. Dostupné z: https://en.wikipedia.org/wiki/HTC_Vive#Vive_Pro.

[17] *k-d tree* [online]. Wikipedia, 2019. [cit. 2019/03/13]. Wikipedia k-dimensionální strom. Dostupné z: https://en.wikipedia.org/wiki/K-d_tree.

[18] *Multilayer perceptron* [online]. Wikipedia, 2019. [cit. 2019/05/07]. Wikipedia vícevrstvý perceptron. Dostupné z: https://en.wikipedia.org/wiki/Multilayer_perceptron.

[19] *Neural network* [online]. Wikipedia, 2019. [cit. 2019/05/06]. Wikipedia neuronová síť. Dostupné z: https://en.wikipedia.org/wiki/Neural_network.

[20] *Artificial neuron* [online]. Wikipedia, 2019. [cit. 2019/02/26]. Wikipedia umělý neuron. Dostupné z: https://en.wikipedia.org/wiki/Artificial_neuron.

[21] *Sensor* [online]. Wikipedia, 2019. [cit. 2019/04/09]. Wikipedia senzor. Dostupné z: `https://en.wikipedia.org/wiki/Sensor`.

[22] *Lighthouse* [online]. XinReality, 2017. [cit. 2017/07/24]. XinReality - Lighthouse tracking system. Dostupné z: `https://xinreality.com/wiki/Lighthouse`.