Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Motion capture pomocí Kinectu

2019

Bc. Ondřej Zamec

Vedoucí práce: RNDr. Jan Konečný, Ph.D.

Studijní obor: Informatika, prezenční forma

Bibliografické údaje

Autor:	Bc. Ondřej Zamec
Název práce:	Motion capture pomocí Kinectu
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2019
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	RNDr. Jan Konečný, Ph.D.
Počet stran:	47
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Bc. Ondřej Zamec
Title:	Motion capture with Kinect
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2019
Study field:	Computer Science, full-time form
Supervisor:	RNDr. Jan Konečný, Ph.D.
Page count:	47
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Cílem diplomové práce je vytvořit aplikaci, která dokáže animovat model 3D postavy podle toho, jak je uživatel zaznamenáván Kinectem. Nejprve se zaměříme na kvaterniony, jelikož je to struktura, která reprezentuje rotaci. Rotace v čase je v animaci nejdůležitější položka jednoho kroku animace. Poté prostudujeme animace samotné, jelikož potřebujeme jejich porozumění k vytvoření inverzního algoritmu k algoritmu skládání transformací jednotlivých kostí modelu.

Synopsis

Objective of this diploma thesis is creating an application, which can animate a 3D model of a character accordinly to a motion capturing of the user by Kinect. Firstly, we will focus on quaternions, because it is a structure that represents rotation. Rotation in time in an animation is the most vital part of one step of the animation. Then, we shall focus on studying the animation itself, since we need knowledge about them to construct an inverse algorithm to the algorithm of composing transformations of individual bones of a model.

Klíčová slova: motion capture; animace: kvaternion

Keywords: motion capture; animation; quaternion

Děkuji doktoru Konečnému za pohotové reagování a chladnou hlavu v mém nepohotovém reagování.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

1 Úvod

V dnešní době se klasické, ruční animování postav používá méně a méně. Z velké části je nahrazeno technologií motion capture, která se dále dostává i do menších produkcí. Motion capture, zkráceně MoCap, je proces nahrávání pohybu lidských herců a používání těchto informací k animování 2D nebo 3D modelů digitálních postav [**2**]. Není to nový pojem, jak byste si mohli myslet. MoCap se používal už za Disneyho, i když v jiné podobě. Animace ve Sněhurce a sedmi trpaslících byly vytvořeny obtahováním herců, kteří předtím byli nahráni kamerou.

Průlom dnešního MoCapu se podařilo udělat Georgu Lucasovi s filmem Star Wars: Epizoda I - Skrytá hrozba. Jar Jar Binksovi se zapsal do historie jako první plně digitální postava na plátně [**3**].

Přivést MoCap na úroveň jakou známe z videí ze zákulisí, čili herci oblečení v černých elastických oblecích se zvýrazněnými body na těle, se povedlo až v roce 2002 Peteru Jacksnovi s postavou Gluma z trilogie Pána Prstenů. Předtím museli herci nahrávat animaci sami ve speciálních místnostech, mimo studio a ostatní herce. S Pánem prstenů se technologie posunula na novou úroveň a herci mohli být v jedné místnosti ve studiu, i když hráli „do záznamu“ nebo naživo.

Další zlom MoCapu nastal s Avatarem od Jamese Camerona, později vylepšený v Planetě Opic, kdy nahrávku MoCapu nebylo potřeba vykreslovat později na jiném systému, ale režisér okamžitě viděl její výsledek na CG modelu.

MoCap se dostal časem i do dalších odvětví, hlavně do herního, ale také do vojenského, sportovního, robotiky a jiných. Nicméně tyto systémy kamer a dalších doplňků jsou i dnes velmi drahé. Tato práce zkoumá možnosti, jestli je možno podobného efektu dosáhnout v amatérském prostředí s neprofesionálním zařízením.

Cílem práce je tedy průzkum a popsání animace a Kinectu. Následné vytvoření algoritmu a aplikace, která zpracuje data z Kinectu a vyprodukuje animaci, kterou bude možné zapsat do souboru. Práce se nejprve zaměří na popis kvaternionů. Ukážeme si důvody proč se kvaterniony používají pro reprezentaci rotace a jak s nimi rotovat vektor nebo bod.

Poté se zaměří na animace, co to vlastně je animace ve 3D světě, jak vypadá na logické úrovni, kterou ve výsledku nevidíme. Také práce uvede základy implementaci animace, ať čtenář může pochopit některé souvislosti.

Následně probere Kinect a data, které produkuje. S popisem algoritmu, který tyto data přetvoří do animace. A na závěr čtenáře čeká uživatelská příručka samotné aplikace. Její ovládání, rysy, možnosti a limitace.

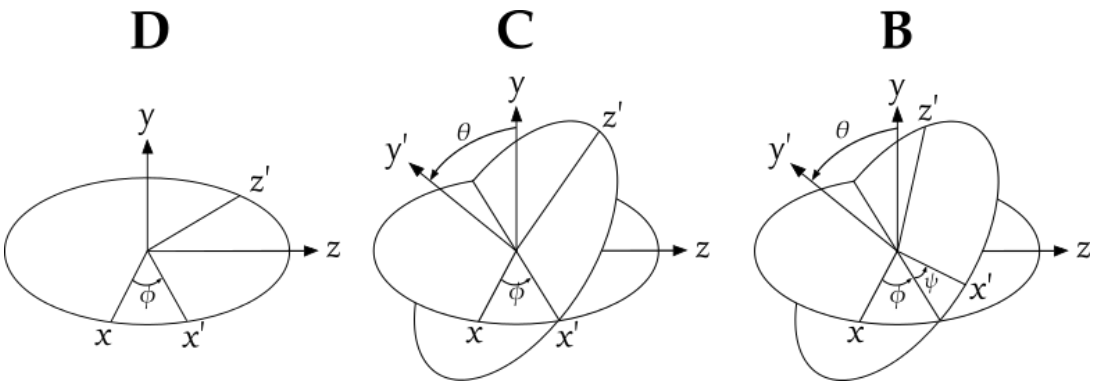
Pro správné pochopení práce se předpokládá, že čtenář zná základy grafů, vektorů, matic a základními operacemi nad nimi. Znalost počítačové grafiky není nutná, ale je výhodou, když je s ní čtenář obeznámen.

2 Kvaterniony

3D character animation (dále jen česky animace, v tomto textu se nebudeme věnovat jiným animacím) je především o rotaci částí modelu v konkrétní čas. Samozřejmě má i další vlastnosti, ale rotace hraje největší roli. Takže jak tedy u animací reprezentujeme rotaci?

2.1 Eulerovy úhly

Naivním řešením rotace v animaci by mohly být Eulerovy úhly (Eulerův souřadnicový systém). Přece jenom dle Eulerova rotačního teorému, může jakákoli rotace být popsána třemi úhly. Jestliže máme rotace zapsány jako rotační matice **D**, **C**, **B**, pak obecná rotace **A** může být napsána jako $\mathbf{A} = \mathbf{BCD}$.



Obrázek 1: Rotace podle x-konvence

Tři úhly, které určují rotační matice se nazývají Eulerovy úhly. Existuje několik konvencí pro Eulerovy úhly, kde záleží na pořadí os kolem kterých se rotace provádí. Takzvaná x-konvence, ilustrovaná nahoře, je tou nejběžnější konvencí. Často se uvádí i pod jinými názvy, např. yaw, pitch, roll odvozenými od rotace letadla. V této konvenci, rotace dána Eulerovými úhly ϕ, θ, ψ, kde

1. První rotace je o úhel ϕ kolem osy y (matice **D**).
2. Druhá rotace je o úhel $\theta \in [0, \pi]$, nyní již kolem osy x', ne x (matice **C**).
3. Třetí rotace je o úhel ψ kolem osy z' (matice **B**).

Matice odpovídající rotacím mají následující tvar:

$$\mathbf{D} = \begin{pmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix} \mathbf{C} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{pmatrix} \mathbf{B} = \begin{pmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{pmatrix}.$$

Jde vidět, že tyto matice nejsou výjimkou a nejsou komutativní, čili na pořadí záleží. Tudíž máme 6 různých způsobů/způsobů skládání rotací. A když se vědci

nedohodnou, jestli osa Y směřuje vzhůru nebo do boku, tak zde nebude jediná zaběhlá konvence. Přece jenom jako animátor nechceme řešit, jestli naše vývojové prostředí bude používat stejnou konvenci jako konečný produkt. Takže bychom potřebovali nějaký jiný matematický objekt, který by představoval rotace a pokud možno měl vhodné interpolační vlastnosti. Na řadu tedy přihází kvaterniony.

2.2 Komplexní čísla

Na kvaterniony se můžeme dívat jako na zobecnění komplexních čísel. Proto je pro nás motivací nejprve prozkoumat komplexní čísla před kvaterniony. Cílem této kapitoly je si ukázat, když vynásobíme komplexní číslo k, které pro nás může představovat vektor nebo bod ve 2D, jednotkovým komplexním číslem, tak to povede k rotaci k.

2.2.1 Definice

Je několik způsobů, jak definovat komplexní čísla. My si je nadefinujeme tak, aby byly co nejvíce podobné vektorům.

Definice 1

Uspořádaná dvojice reálných čísel $\mathbf{z} = (a, b)$ je komplexní číslo. První komponenta se nazývá reálná část a druhá komponenta se nazývá imaginární část. Dále rovnost, sčítání, odečítání, násobení a dělení jsou definovány takto:

1. $(a, b) = (c, d)$ právě když $a = c$ a $b = d$
2. $(a, b) \pm (c, d) = (a \pm c, b \pm d)$
3. $(a, b)(c, d) = (ac - bd, ad + bc)$
4. $\dfrac{(a, b)}{(c, d)} = (\dfrac{ac + bd}{c^2 + d^2}, \dfrac{bc - ad}{c^2 + d^2})$ když $(c, d) \neq (0, 0)$.

Je jednoduché dokázat, že aritmetické vlastnosti, např. komutativita, asociativita, distribuční zákony reálných čísel platí i pro komplexní čísla. Jestliže komplexní číslo má nulovou imaginární část $(x, 0)$, pak ho označujeme jako reálné číslo x a píšeme $x = (x, 0)$. Čili každé reálné číslo můžeme považovat za komplexní číslo s nulovou imaginární částí. Když vynásobíme komplexní číslo číslem reálným

$$x(a, b) = (x, 0)(a, b) = (xa, xb) = (a, b)(x, 0) = (a, b)x,$$

tak je na první pohled zjevná souvislost mezi násobením vektoru skalárem. Dále si definujeme imaginární jednotku $i = (0, 1)$. Použitím naší definice pro násobení komplexních čísel, vidíme že

$$i^2 = (0, 1)(0, 1) = (-1, 0) = -1,$$

z čehož vyplývá $i = \sqrt{-1}$.

Konjugace komplexního čísla $\mathbf{z} = (a, b)$ je značena $\mathbf{z}^*$ a dána $\mathbf{z}^* = (a, -b)$
S definovanou imaginární jednotkou se komplexní číslo a, b dá zapsat ve tvaru $a + ib$. Máme $a = (a, 0), b = (b, 0)$ a $i = (0, 1)$, pak

$$a + ib = (a, 0) + (0, 1)(b, 0) = (a, 0) + (0, b) = (a, b).$$

Teď pomocí formy $a+ib$ můžeme přepsat rovnice pro sčítání, odčítání, násobení, dělení a konjugaci:

1. $(a + ib) \pm (c + id) = (a \pm c) + i(b \pm d)$
2. $(a + ib)(c + id) = (ac - bd) + i(ad + bc)$
3. $\dfrac{a + ib}{c + id} = \dfrac{ac + bd}{c^2 + d^2} + i\dfrac{bc - ad}{c^2 + d^2}$ když $(c, d) \neq (0, 0)$
4. $\mathbf{z}^* = a - ib$.

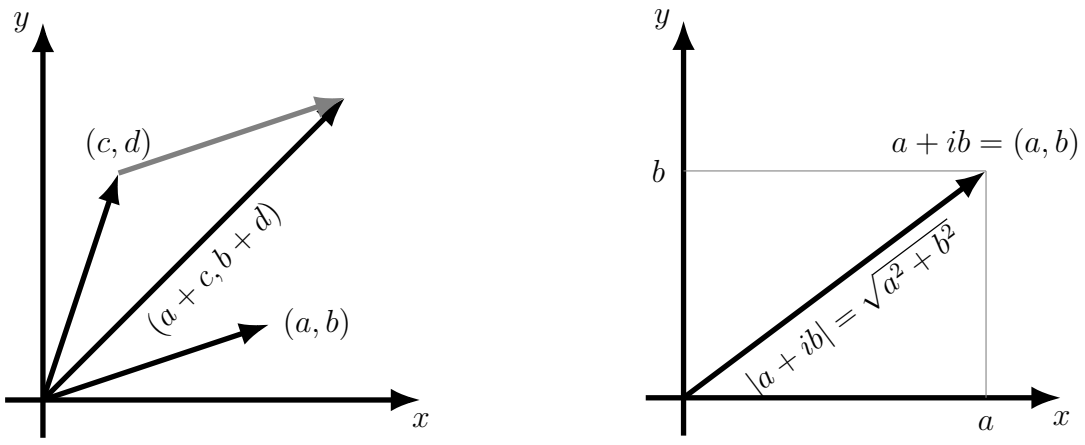
2.2.2 Geometrická interpretace

Tvar uspořádaného páru $a + ib = (a, b)$ komplexního čísla nám napovídá, že o něm můžeme geometricky přemýšlet jako o 2D bodě nebo vektoru v komplexní rovině. Dokonce naše definice sčítání a odčítání je shodná s definicemi sčítání a odčítání vektorů.

Absolutní hodnota nebo délka vektorů komplexního čísla $a+ib$ je definována jako délka vektoru, který představuje:

$$|a + ib| = \sqrt{a^2 + b^2}.$$

Pokud má komplexní číslo délku jedna, pak ho označujeme jako jednotkové (normované) komplexní číslo.



Obrázek 2: Vlevo komplexní sčítání, vpravo délka komplexního čísla

2.2.3 Polární reprezentace a rotace

Protože se na komplexní čísla můžeme dívat jako na vektory ve 2D komplexním prostoru, můžeme vyjádřit jejich komponenty pomocí polárních souřadnic:

$$r = |a + ib|$$

$$a + ib = r\cos\theta + ir\sin\theta = r(\cos\theta + i\sin\theta).$$

Pravá strana rovnice se nazývá polární reprezentace komplexního čísla $a + ib$. Zkusme vynásobit dvě komplexní čísla v jejich polárním tvaru.
Nechť $\mathbf{z}_1 = r_1(\cos\theta_1 + i\sin\theta_1)$ a $\mathbf{z}_2 = r_2(\cos\theta_2 + i\sin\theta_2)$. Pak

$$\begin{aligned}\mathbf{z}_1\mathbf{z}_2 &= r_1r_2(\cos\theta_1\cos\theta_2 - \sin\theta_1\sin\theta_2 + i(\cos\theta_1\cos\theta_2 + \sin\theta_1\sin\theta_2)) \\ &= r_1r_2(\cos(\theta_1 + \theta_2) + i(\sin(\theta_1 + \theta_2)),\end{aligned}$$

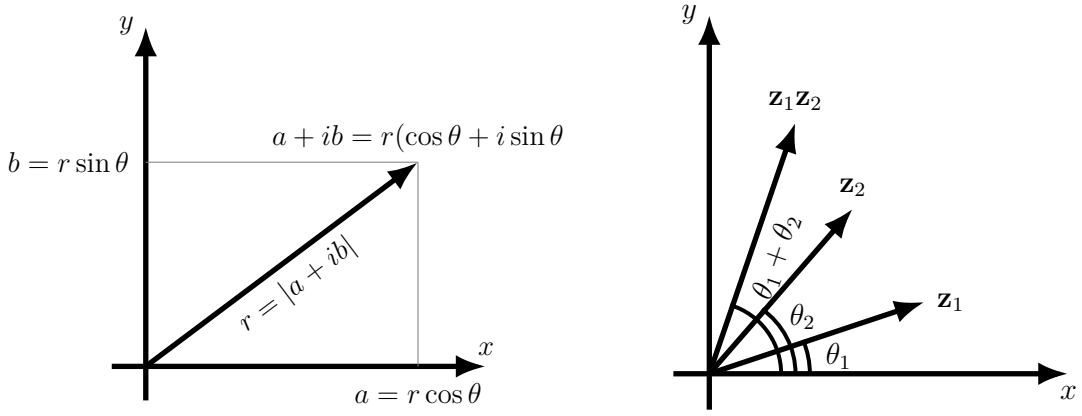
kde jsme použili trigonometrické rovnosti

$$\begin{aligned}\sin(\alpha + \beta) &= \sin\alpha\cos\beta + \cos\alpha\sin\beta, \\ \cos(\alpha + \beta) &= \cos\alpha\cos\beta - \sin\alpha\sin\beta.\end{aligned}$$

Geometricky popsáno, součin $\mathbf{z}_1\mathbf{z}_2$ je komplexní číslo představující vektor s délkou r_1r_2 a úhlem $\theta_1 + \theta_2$ na reálné ose. Když si zvolíme $r_2 = 1$, pak
$\mathbf{z}_1\mathbf{z}_2 = r_1(\cos(\theta_1 + \theta_2) + i(\sin(\theta_1 + \theta_2))$, což bude mít za následek rotaci $\mathbf{z}_1$ o úhel θ_2. Čili když se na komplexní čísla díváme jako na vektory, můžeme vyvodit následující pozorování:

Lemma 2

Násobením komplexního čísla $\boldsymbol{z}_1$ jednotkovým komplexním číslem $\boldsymbol{z}_2$ má za následek rotaci $\boldsymbol{z}_1$.



Obrázek 3: Vlevo: polární reprezentace komplexního čísla; Vpravo: součin $\mathbf{z}_1\mathbf{z}_2$ rotuje $\mathbf{z}_1$ o úhel θ_2

2.3 Teorie kvaternionů

Cílem této kapitoly je navázat na komplexní čísla a ukázat si jak speciální součin 2 kvaternionů, kde druhý je jednotkový, vyústí v rotaci vektoru nebo bodu **p**.

2.3.1 Definice a základní funkce

Definice 3

Uspořádaná čtveřice reálných čísel $\mathbf{q} = (x, y, z, w) = (q_1, q_2, q_3, q_4)$ je kvaternion (angl. quaternion). Zápis kvaternionu se běžně zkracuje jako $\mathbf{q} = (\mathbf{u}, w) = (x, y, z, w)$, $\mathbf{u} = (x, y, z)$ nazýváme imaginární (vektorovou) částí a w částí reálnou. Funkce rovnosti, sčítání, odčítání a násobení jsou definovány následovně:

1. $(\mathbf{u}, a) = (\mathbf{v}, b)$ právě když $\mathbf{u} = \mathbf{v}$ a $a = b$
2. $(\mathbf{u}, a) \pm (\mathbf{v}, b) = (\mathbf{u} \pm \mathbf{v}, a \pm b)$
3. $(\mathbf{u}, a)(\mathbf{v}, b) = (a\mathbf{v} + b\mathbf{u} + \mathbf{u} \times \mathbf{v}, ab - \mathbf{u} \cdot \mathbf{v})$.

Nechť máme $\mathbf{p} = (\mathbf{u}, p_4) = (p_1, p_2, p_3, p_4)$ a $\mathbf{q} = (\mathbf{v}, q_4) = (q_1, q_2, q_3, q_4)$. Pak $\mathbf{u} \times \mathbf{v} = (p_2q_3 - p_3q_2, p_3q_1 - p_1q_3, p_1q_2 - p_2q_1)$ je vektorový součin a $\mathbf{u} \cdot \mathbf{v} = p_1q_1 + p_2q_2 + p_3q_3$ je skalární součin. Teď můžeme zapsat kvaternionový součin $\mathbf{r} = \mathbf{pq}$ po komponentech takto:

$$
\begin{aligned}
r_1 &= p_4q_1 + q_4p_1 + p_2q_3 - p_3q_2 = q_1p_4 - q_2p_3 + q_3p_2 + q_4p_1 \\
r_2 &= p_4q_2 + q_4p_2 + p_3q_1 - p_1q_3 = q_1p_3 + q_2p_4 - q_3p_1 + q_4p_2 \\
r_3 &= p_4q_3 + q_4p_3 + p_1q_2 - p_2q_1 = q_2p_1 - q_1p_2 + q_3p_4 + q_4p_3 \\
r_4 &= p_4q_4 - p_1q_1 - p_2q_2 - p_3q_3 = q_1p_1 - q_2p_2 - q_3p_3 - q_4p_4.
\end{aligned}
$$

To také to může být vyjádřeno pomocí násobení matice vektorem:

$$
\mathbf{pq} = \begin{pmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{pmatrix} \begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{pmatrix}.
$$

2.3.2 Speciální součiny

Nechť $\mathbf{i} = (1, 0, 0, 0)$, $\mathbf{j} = (0, 1, 0, 0)$, $\mathbf{k} = (0, 0, 1, 0)$ jsou kvaterniony. Pak máme tyto součiny, některé se podobají vektorovému součinu nebo imaginární jednotce z komplexních čísel:

$$
\begin{aligned}
\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk}^2 &= -1, \\
\mathbf{ij} = \mathbf{k} &= -\mathbf{ji}, \\
\mathbf{jk} = \mathbf{i} &= -\mathbf{kj}, \\
\mathbf{ki} = \mathbf{j} &= -\mathbf{ik}.
\end{aligned}
$$

Tyto rovnice pochází přímo z násobení kvaternion, např.

$$\mathbf{ij} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \mathbf{k}.$$

V některých textech se můžete setkat s definicí podobnou komplexnímu číslu $a+ib$ pomocí předchozích speciálních kvaternionů. Definice bude podobná tomuto tvaru $p = \mathbf{i}a + \mathbf{j}b + \mathbf{k}c + d$, kde a, b, c, d jsou reálná čísla. Avšak pro naše účely ji teď nebudeme potřebovat.

2.3.3 Vlastnosti

Součin kvaternionů není komutativní, viz předchozí speciální součiny. Ale je asociativní. Toto lze vypozorovat z faktu, že součin kvaternionů lze zapsat jako součin matic, který je asociativní. Kvaternion $\mathbf{e} = (0, 0, 0, 1)$ slouží jako identita

$$\mathbf{pe} = \mathbf{ep} = \begin{pmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix}.$$

Dále také pro kvaternionový součin platí distributivní zákony přes sčítání:
$\mathbf{p}(\mathbf{q} + \mathbf{r}) = \mathbf{pq} + \mathbf{pr}$ a $(\mathbf{q} + \mathbf{r})\mathbf{p} = \mathbf{qp} + \mathbf{rp}$.

2.3.4 Převody

Nechť s je reálné číslo a $\mathbf{u} = (x, y, z)$ je vektor. Pak

1. $s = (0, 0, 0, s)$
2. $\mathbf{u} = (x, y, z) = (\mathbf{u}, 0) = (x, y, z, 0)$

Když takhle propojíme reálná čísla, vektory a kvaterniony, pak reálné číslo může představovat kvaternion s nulovou imaginární/vektorovou částí a každý vektor může představovat kvaternion s nulovou reálnou částí. Speciálně můžeme vidět, že kvaternion identity $1 = (0, 0, 0, 1) = \mathbf{e}$ sedí do tohoto převodu. Kvaternion s nulovou reálnou částí se nazývá *ryzí*. Můžeme pozorovat, že když kvaternion násobíme reálným číslem, tak je to skalární součin matic a je komutativní:

$$s(p_1, p_2, p_3, p_4) = (0, 0, 0, s)(p_1, p_2, p_3, p_4) = \begin{pmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & s \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix} = \begin{pmatrix} sp_1 \\ sp_2 \\ sp_3 \\ sp_4 \end{pmatrix}.$$

Podobně

$$(p_1, p_2, p_3, p_4)s = (p_1, p_2, p_3, p_4)(0, 0, 0, s) = \begin{pmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ s \end{pmatrix} = \begin{pmatrix} sp_1 \\ sp_2 \\ sp_3 \\ sp_4 \end{pmatrix}.$$

2.3.5 Konjugace a délka

Konjugace kvaternionu $\mathbf{q} = (q_1, q_2, q_3, q_4) = (\mathbf{u}, q_4)$ je značena $\mathbf{q}^*$ a je definována následovně:

$$\mathbf{q}^* = (-q_1, -q_2, -q_3, q_4) = (-\mathbf{u}, q_4).$$

Jinak řečeno konjugací jen znegujeme imaginární část kvaternionu. Můžeme to porovnat s konjugací komplexních čísel $\mathbf{z}^* = (a, -b)$. Konjugace má následující vlastnosti:

1. $(\mathbf{pq})^* = \mathbf{q}^*\mathbf{p}^*$
2. $(\mathbf{p} + \mathbf{q})^* = \mathbf{p}^* + \mathbf{q}^*$
3. $((\mathbf{q})^*)^* = \mathbf{q}$
4. $(s\mathbf{q})^* = s\mathbf{q}^*$
5. $\mathbf{q} + \mathbf{q}^* = (\mathbf{u}, q_4) + (-\mathbf{u}, q_4) = (0, 2q_4) = 2q_4$
6. $\mathbf{q}\mathbf{q}^* = \mathbf{q}^*\mathbf{q} = (\mathbf{u}, q_4)(-\mathbf{u}, q_4) = (-q_4\mathbf{u} + q_4\mathbf{u} + \mathbf{u} \times -\mathbf{u}, q_4^2 - (\mathbf{u} \cdot -\mathbf{u}))$
   $= (0, q_4^2 + \mathbf{u}^2) = q_1^2 + q_2^2 + q_3^2 + q_4^2 = \|\mathbf{u}\|^2 + q_4^2$.

V 5. a 6. rovnici můžeme pozorovat, že se vyhodnotí na reálná čísla, konkrétně 6. rovnici jsme využili faktu, že vektorový součin dvou souběžných vektorů je nula.

Délka (norma) kvaternionu je definována následovně:

$$\|\mathbf{q}\| = \sqrt{\mathbf{q}\mathbf{q}^*} = \sqrt{q_1^2 + q_2^2 + q_3^2 + q_4^2} = \sqrt{\|\mathbf{u}^2\| + q_4^2}.$$

Říkáme, že kvaternion je jednotkový kvaternion, jestliže má délku rovnu jedné. Délka má následující vlastnosti:

1. $\|\mathbf{q}\|^* = \|\mathbf{q}\|$
2. $\|\mathbf{pq}\| = \|\mathbf{p}\|\|\mathbf{q}\|$.

Hlavně nás zajímá druhá vlastnost. Ta nám říká, že součin dvou jednotkových kvaternionů je jednotkový kvaternion. Také pokud $\|\mathbf{p}\| = 1$, pak $\|\mathbf{pq}\| = \|\mathbf{q}\|$

2.3.6 Inverze

Protože násobení kvaternionů není komutativní, tak stejně jako u matic nemůžeme definovat dělení. Avšak každý nenulový kvaternion má inverzi. Nulový kvaternion má ve všech komponentech nuly $\mathbf{q} = (0, 0, 0, 0)$. Inverze kvaternionu $\mathbf{q}$ je kvaternion $\mathbf{p}$ takový, že $\mathbf{qp} = (0, 0, 0, 1)$

Nechť $\mathbf{q} = (q_1, q_2, q_3, q_4) = (\mathbf{u}, q_4)$ je nenulový kvaternion, pak inverze existuje, je značena $\mathbf{q}^{-1}$ a definována následovně:

$$\mathbf{q}^{-1} = \frac{\mathbf{q}^*}{\|\mathbf{q}\|^2}.$$

Je jednoduché zkontrolovat, že to skutečně je inverze, máme:

$$\mathbf{q}\mathbf{q}^{-1} = \frac{\mathbf{q}\mathbf{q}^*}{\|\mathbf{q}\|^2} = \frac{\|\mathbf{q}\|^2}{\|\mathbf{q}\|^2} = 1 = (0,0,0,1),$$
$$\mathbf{q}^{-1}\mathbf{q} = \frac{\mathbf{q}^*\mathbf{q}}{\|\mathbf{q}\|^2} = \frac{\|\mathbf{q}\|^2}{\|\mathbf{q}\|^2} = 1 = (0,0,0,1).$$

Pokud je $\mathbf{q}$ jednotkový kvaternion, pak $\|\mathbf{q}\| = 1$, tudíž platí $\mathbf{q}^{-1} = \mathbf{q}^*$
Pro kvaternionovou inverzi platí následující vlastnosti:

1. $(\mathbf{q}^{-1})^{-1} = \mathbf{q}$
2. $(\mathbf{pq})^{-1} = \mathbf{q}^{-1}\mathbf{p}^{-1}$.

2.3.7 Polární reprezentace

Nechť $\mathbf{q} = (q_1, q_2, q_3, q_4)$ je jednotkový kvaternion, pak

$$\|\mathbf{q}\|^2 = \|\mathbf{u}\|^2 + q_4^2 = 1.$$

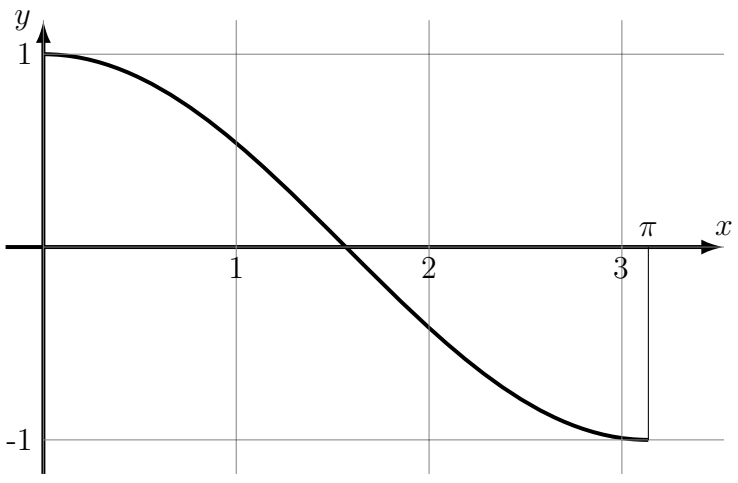
Toto implikuje $q_4^2 \leq 1 \Leftrightarrow |q_4| \leq 1 \Leftrightarrow -1 \leq q_4 \leq 1$. Obrázek 2.3.7 ukazuje, že existuje úhel $\theta \in [0, \pi]$ takový, kde platí $q_4 = \cos\theta$. Když dosadíme trigonometrickou identitu $\sin^2\theta \cos^2\theta = 1$ dostaneme

$$sin^2\theta = 1 - \cos^2\theta = 1 - q_4^2 = \|\mathbf{u}\|^2 + q_4^2 - q_4^2 = \|\mathbf{u}\|^2.$$

To implikuje

$$\|\mathbf{u}\| = |\sin\theta| = \sin\theta \quad \text{pro} \quad \theta \in [0, \pi].$$

Definujeme si jednotkový vektor $\mathbf{n}$ ve stejném směru jako $\mathbf{u}$:



Obrázek 4: Pro číslo $y \in [-1, 1]$ existuje úhel θ takový, že $y = \cos\theta$

$$\mathbf{n} = \frac{\mathbf{u}}{\|\mathbf{u}\|} = \frac{\mathbf{u}}{\sin \mathbf{u}}.$$

Z toho vyplývá $\mathbf{u} = \mathbf{n} \sin\theta$ a následně tedy můžeme psát jednotkový kvaternion $\mathbf{q} = (\mathbf{u}, q_4)$ v následující polární reprezentaci, kde $\mathbf{n}$ je jednotkový vektor:

$$\mathbf{q} = (\mathbf{n} \sin\theta, \cos\theta) \quad \text{pro} \quad \theta \in [0, \pi].$$

Například vezměme si kvaternion $\mathbf{q} = (0, \frac{1}{2}, 0, \frac{\sqrt{3}}{2})$. K převedení do polární reprezentace vypočteme

$$\theta = \arccos \frac{\sqrt{3}}{2} = \frac{\pi}{6} \text{ a } \mathbf{n} = \frac{(0, \frac{1}{2}, 0)}{\sin \frac{\pi}{6}} = (0, 1, 0).$$

Což nám při dosazení dá

$$\mathbf{q} = (\sin \frac{\pi}{6}(0, 1, 0), \cos \frac{\pi}{6}).$$

V další kapitole si ukážeme, že $\mathbf{n}$ reprezentuje osu otáčení.

Z vlastností sin a cos můžeme vypozorovat, že substitucí opačného úhlu $-\theta$ za θ je ekvivalentní znegování vektorové části kvaternionu.

$$(\mathbf{n} \sin(-\theta), \cos(-\theta)) = (-\mathbf{n}) \sin\theta, \cos\theta) = \mathbf{p}^*.$$

Omezení $\theta \in [0, \pi]$ je zde z důvodu, abychom při převodu kvaternionu $\mathbf{q} = (q_1, q_2, q_3, q_4)$ do jeho polární reprezentace dostali unikátní úhel. Nic nám nebrání, abychom zkonstruovali kvaternion takto $\mathbf{q} = (\mathbf{n} \sin(\theta + 2\pi n), \cos(\theta + 2\pi n))$, kde n je přirozené číslo. Avšak kvaternion by tím ztratil unikátní polární reprezentaci bez omezení $\theta \in [0, \pi]$

2.4 Jednotkové kvaterniony a rotace

Nyní už máme definované vše potřebné k definici rotace. Připomene si ještě dvě operace. První je inverze jednotkového kvaternionu, která se rovná konjugaci $\mathbf{q}^{-1} = \mathbf{q}^*$. Druhá je násobení dvou kvaternionů:

$$(\mathbf{m}, a)(\mathbf{n}, b) = (a\mathbf{n} + b\mathbf{m} + \mathbf{m} \times \mathbf{n}, ab - m\dot{n}).$$

2.4.1 Zobrazení rotace

Nechť $\mathbf{q} = (\mathbf{u}, w)$ je jednotkový kvaternion a $\mathbf{v}$ bod nebo vektor je 3D prostoru. $\mathbf{v}$ můžeme brát jako ryzí kvaternion $\mathbf{p} = (\mathbf{v}, 0)$. Vezměme si následující součin:

$$\begin{aligned} \mathbf{qpq}^{-1} &= \mathbf{qpq}^* \\ &= (\mathbf{u}, w)(\mathbf{v}, 0)(-\mathbf{u}, w) \\ &= (\mathbf{u}, w)(w\mathbf{v} - \mathbf{v} \times \mathbf{u}, \mathbf{v} \cdot \mathbf{u}). \end{aligned}$$

Zjednodušení dané rovnice je delší proces, takže to rozdělíme na reálnou část a vektorovou. Navíc, si zavedeme následující substituce:

$$\begin{aligned} a &= w, \\ b &= \mathbf{v} \cdot \mathbf{u}, \\ \mathbf{m} &= \mathbf{u}, \\ \mathbf{n} &= w\mathbf{v} - \mathbf{v} \times \mathbf{u}. \end{aligned}$$

Reálná část:

$$\begin{aligned} & ab - \mathbf{m} \cdot \mathbf{n} \\ &= w(\mathbf{v} \cdot \mathbf{u}) - \mathbf{u} \cdot (w\mathbf{v} - \mathbf{v} \times \mathbf{u}) \\ &= w(\mathbf{v} \cdot \mathbf{u}) - \mathbf{u} \cdot w\mathbf{v} + \mathbf{u} \cdot (\mathbf{v} \times \mathbf{u}) \\ &= w(\mathbf{v} \cdot \mathbf{u}) - w(\mathbf{v} \cdot \mathbf{u}) + 0 \\ &= 0. \end{aligned}$$

$\mathbf{u} \cdot (\mathbf{v} \times \mathbf{u}) = 0$ protože $(\mathbf{v} \times \mathbf{u})$ je ortogonální k $\mathbf{u}$ podle definice vektorového součinu.

Vektorová část:

$$\begin{aligned} & a\mathbf{n} + n\mathbf{m} + \mathbf{m} \times \mathbf{n} \\ &= w(w\mathbf{v} - \mathbf{v} \times \mathbf{u}) + (\mathbf{v} \cdot \mathbf{u})\mathbf{u} + \mathbf{u} \times (w\mathbf{v} - \mathbf{v} \times \mathbf{u}) \\ &= w^2\mathbf{v} - w\mathbf{v} \times \mathbf{u} + (\mathbf{v} \cdot \mathbf{u})\mathbf{u} + \mathbf{u} \times w\mathbf{v} + \mathbf{u} \times (\mathbf{u} \times \mathbf{v}) \\ &= w^2\mathbf{v} + \mathbf{u} \times w\mathbf{v} + (\mathbf{v} \cdot \mathbf{u})\mathbf{u} + \mathbf{u} \times w\mathbf{v} + \mathbf{u} \times (\mathbf{u} \times \mathbf{v}) \\ &= w^2\mathbf{v} + 2(\mathbf{u} \times w\mathbf{v}) + (\mathbf{v} \cdot \mathbf{u})\mathbf{u} + \mathbf{u} \times (\mathbf{u} \times \mathbf{v}) \\ &= w^2\mathbf{v} + 2(\mathbf{u} \times w\mathbf{v}) + (\mathbf{v} \cdot \mathbf{u})\mathbf{u} + (\mathbf{u} \cdot \mathbf{v})\mathbf{u} - (\mathbf{u} \cdot \mathbf{u})\mathbf{v} \\ &= (w^2 - \mathbf{u} \cdot \mathbf{u})\mathbf{v} + 2w(\mathbf{u} \times \mathbf{v}) + 2(\mathbf{u} \cdot \mathbf{v})\mathbf{u} \\ &= (w^2 - \mathbf{u} \cdot \mathbf{u})\mathbf{v} + 2(\mathbf{u} \cdot \mathbf{v})\mathbf{u} + 2w(\mathbf{u} \times \mathbf{v}). \end{aligned}$$

kde na 6. řádku jsme aplikovali trojitou součinovou identitu
$\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = (\mathbf{a} \cdot \mathbf{c})\mathbf{b} - (\mathbf{a} \cdot \mathbf{b})\mathbf{c}$ na $\mathbf{u} \times (\mathbf{u} \times v)$.

Takže $\mathbf{qpq}^*$ jsme zjednodušili na:

$$\mathbf{qpq}^* = ((w^2 - \mathbf{u} \cdot \mathbf{u})\mathbf{v} + 2(\mathbf{u} \cdot \mathbf{v})\mathbf{u} + 2w(\mathbf{u} \times \mathbf{v}), 0). \tag{1}$$

Můžeme pozorovat, že výsledek je vektor nebo bod protože reálná část je nula. Přece jenom by bylo nežádoucí udělat z bodu kvaternion. V dalších výpočtech můžeme tedy reálnou část vypustit.
A protože $\mathbf{q}$ je jednotkový kvaternion, může být zapsán takto:

$$\mathbf{q} = (\mathbf{n} \sin\theta, \cos\theta) \text{ pro } \|\mathbf{n} = 1\| \text{ a } \theta \in [0, \pi].$$

Substituováním této rovnice do 1 dostaneme:

$$\begin{aligned}\mathbf{qpq}^* &= (\cos^2\theta - \sin^2\theta)\mathbf{v} + 2(\sin\theta\mathbf{n}\cdot\mathbf{v})\mathbf{n}\sin\theta + 2\cos\theta(\sin\theta\mathbf{n}\times\mathbf{v}) \\ &= (\cos^2\theta - \sin^2\theta)\mathbf{v} + 2\sin^2\theta(\mathbf{n}\cdot\mathbf{v})\mathbf{n} + 2\cos\theta\sin\theta(\mathbf{n}\times\mathbf{v}).\end{aligned}$$

K dalšímu zjednodušení aplikujeme tyto 3 trigonometrické identity:

$$\begin{aligned}\cos^2\theta - \sin^2\theta &= \cos(2\theta), \\ 2\cos\theta\sin\theta &= \sin(2\theta), \\ \cos(2\theta) &= 1 - 2\sin^2\theta.\end{aligned}$$

$$\begin{aligned}\mathbf{qpq}^* &= (\cos^2\theta - \sin^2\theta)\mathbf{v} + 2\sin^2\theta(\mathbf{n}\cdot\mathbf{v})\mathbf{n} + 2\cos\theta\sin\theta(\mathbf{n}\times\mathbf{v}) \\ &= (\cos(2\theta)\mathbf{v} + (1-\cos(2\theta))(\mathbf{n}\cdot\mathbf{v})\mathbf{n} + \sin(2\theta)(\mathbf{n}\times\mathbf{v}).\end{aligned}$$

Teď tedy můžeme definovat zobrazení $R_{\mathbf{q}}(\mathbf{v})$, které rotuje bod nebo vektor $\mathbf{v}$ okolo osy $\mathbf{n}$ o úhel 2θ:

$$R_{\mathbf{q}}(\mathbf{v}) = \mathbf{qvq}^{-1} = \mathbf{qvq}^* = (\cos(2\theta)\mathbf{v} + (1-\cos(2\theta))(\mathbf{n}\cdot\mathbf{v})\mathbf{n} + \sin(2\theta)(\mathbf{n}\times\mathbf{v}).$$

Předpokládejme, že dostaneme osu $\mathbf{n}$ a úhel θ abychom rotovali okolo osy $\mathbf{n}$. Odpovídající kvaternion zkonstruujeme následovně:

$$\mathbf{q} = (\sin(\frac{\theta}{2})\mathbf{n}, \cos(\frac{\theta}{2})). \tag{2}$$

Pak aplikujeme zobrazení $R_{\mathbf{q}}(\mathbf{v})$ na vektor $\mathbf{v}$. Úhly dělíme dvěma, abychom si vynahradili 2θ v rovnici. Chceme rotovat jenom o θ. Ještě ukážeme jak vypadá zobrazení $R_{\mathbf{q}}(\mathbf{v})$ přepsané do matice, už bez postupu.

$$R_{\mathbf{q}}(\mathbf{v}) = \mathbf{vQ} = \begin{pmatrix} v_x & v_y & v_z \end{pmatrix} \begin{pmatrix} 1-2q_2^2-2q_3^2 & 2q_1q_2+2q_3q_4 & 2q_1q_3-2q_2q_4 \\ 2q_1q_2-2q_3q_4 & 1-2q_1^2-2q_3^2 & 2q_2q_3+2q_1q_4 \\ 2q_1q_3+2q_2q_4 & 2q_2q_3-2q_1q_4 & 1-2q_1^2-2q_2^3 \end{pmatrix}. \tag{3}$$

2.4.2 Kompozice

Předpokládejme, že $\mathbf{p}$ a $\mathbf{q}$ jsou jednotkové kvaterniony s odpovídajícími zobrazeními rotace dané $R_{\mathbf{p}}$ a $R_{\mathbf{q}}$. Nechť $\mathbf{v'} = R_{\mathbf{p}}(\mathbf{v})$, pak kompozice je dána následovně:

$$R_{\mathbf{q}}(R_{\mathbf{p}}(\mathbf{v})) = R_{\mathbf{q}}(\mathbf{v}') = \mathbf{qv'q}^{-1} = \mathbf{q}(\mathbf{pvp}^{-1})\mathbf{q}^{-1} = (\mathbf{qp})\mathbf{v}(\mathbf{p}^{-1}\mathbf{q}^{-1}) = (\mathbf{qp})\mathbf{v}(\mathbf{qp})^{-1}.$$

Protože $\mathbf{p}$ a $\mathbf{q}$ jsou jednotkové kvaterniony, jejich součin $\mathbf{pq}$ je také jednotkový kvaternion. Tedy i jejich součin $\mathbf{pq}$ představuje rotaci danou kompozicí $R_q(R_p(\mathbf{v}))$.

3 Animace

V této kapitole si ukážeme, jak se animují složité postavy (anlg. character) jako jsou lidé nebo zvířata, např. kůň. Postavy se na rozdíl od „neživých“ objektů animují daleko hůře, protože mají mnoho pohyblivých částí, které se všechny hýbou jiným směrem. I když bychom brali jenom obyčejnou chůzi, tak i zde se hýbe každá kost. Na to, jak rozpohybovat model z existujících dat se zaměříme v téhle kapitole. Jak z dat udělat animaci rozebereme až v kapitole 5.

3.1 Úvod do 3D grafiky

Protože animace je v 3D implementaci poněkud složitější a navazuje na témata, které se zpravidla probírají před animací, tak si nejprve musíme uvést některé pojmy. Nebudeme zacházet do podrobností, ale obecné povědomí bychom měli mít o základech 3D grafiky. Jeden pojem si popíšeme detailněji: world space, local space a transformace mezi nimi.

- *Vertex* je struktura, která mimo jiné obsahuje body v 3D prostoru (pozici). Pozici a další vektory, které obsahuje (normály a tangenty, pro nás nedůležité), se většinou popisují v Eukleidovském prostoru (x,y,z). Někdy se vertexem označuje sama pozice. Z kontextu se většinou dá vyvodit, který vertex máme na mysli.

- *Primitiva* jsou nejmenší prvky, které jsme schopni vykreslit. Jsou to body, úsečky a trojúhelníky vytvořené z odpovídajících vertexů. Z těchto tří primitiv se skládá každý vykreslený objekt. Body a úsečky se používají velmi zřídka.

- *Polygon* jsou většinou dva trojúhelníky, i když můžou mít složitější strukturu.

- *Mesh* je kolekce vertexů a jím odpovídajících primitiv. Mesh logicky bývá celý objekt nebo jeho část, např. ruka, kolo od auta apod. Mesh má vlastní transformační matici, viz kapitola 3.1.1.

- *Model* je kolekce meshů, které dohromady tvoří celistvý objekt. Např. model může být člověk a jemu odpovídat 5 meshů, 4 končetiny a trup. Složitější modely se většinou skládají z desítek meshů.

- *Geometrie* jsou všechny vertexy, a jejich uskupení do primitiv, meshe nebo modelu.

- *Scéna* je logicky nejvyšší prvek, je to kolekce modelů, jejich animací a světel. Stejně jako ve filmu scéna není pevně definována na velikost a může to být samostatný model, ale také i celé město.

3.1.1 L-prostor a W-prostor

Na chvíli si představte, že pracujete na filmu a váš tým má vytvořit zmenšenou verzi scény s vlakem. Konkrétně vy máte postavit most. Pravděpodobně byste ho nepostavili uprostřed studia, kde by se s tím nepracovalo nejlépe a mohli byste pokazit ostatní kulisy, které tvoří scénu. Místo toho byste na mostu dělali ve své dílně mimo scénu a přenesli ho do správné pozice až bude hotový.

Ve 3D grafice je to podobné. Místo abychom tvořili objekt se souřadnicemi relativními ke globálnímu souřadnicovému systému *W-prostor* (angl. *world space*), určujeme je vzhledem k lokálnímu souřadnicovému systému *L-prostor* (angl. *local space*). Lokální systém pravděpodobně bude nějaký vhodný souřadnicový systém umístěný vedle objektu a osy budou souběžné s objektem. Jakmile vertexy modelu byly definovány ve L-prostoru, objekt je umístěn do W-prostoru. Abychom to mohli učinit, musíme definovat, jakým způsobem jsou L-prostor a W-prostor propojeny. Tohle uděláme tak, že si určíme kde bude počátek a osy lokálního souřadnicového systému vzhledem ke globálnímu souřadnicovému systému a provedeme transformaci souřadnic. Proces změny souřadnic relativních k lokálnímu souřadnicovému systému do globálního souřadnicového systému je nazýván *world transform* a odpovídající matice je nazvána *world matrix/světová matice*. Každý objekt ve 3D scéně má vlastní světovou matici. Když bychom chtěli, tak můžeme definovat objekt přímo ve W-prostoru, jen mu nastavíme jeho světovou matici na identitu.

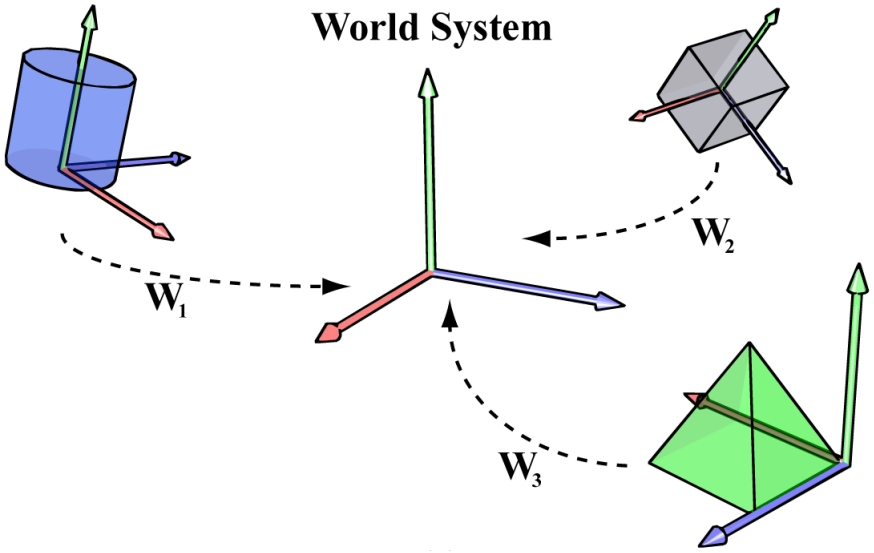
Vytváření objektu ve svém souřadnicovém systému má několik výhod:

- Je to jednodušší. Většinou objekt v L-prostoru bude mít střed v počátku souřadnicového systému a bude symetrický se směrem os. Např. rotace by byla značně těžší bez výchozího středu v počátku.
- Objekt může být použit napříč scénami. Nedává smysl abychom ho natvrdo umístili do jedné scény, když bychom ho chtěli použít ve druhé.
- Někdy vykreslujeme jeden model vícekrát v jedné scéně, ale v jiné pozici, rotaci a škálování. Např. s několika modely stromů jsme schopni vysázet celý les. Plýtvali bychom zdroji, kdybychom duplikovali stejné vertexy objektu. Tato praktika se nazývá *instancing*.

Teď si popíšeme world matrix. Ta je dána 4 vektory, $\mathbf{Q}_w = (Q_x, Q_y, Q_z, 1)$, $\mathbf{u}_w = (u_x, u_y, u_z, 0)$, $\mathbf{v}_w = (v_x, v_y, v_z, 0)$ a $\mathbf{w}_w = (w_x, w_y, w_z, 0)$, které určují, po řadě, počátek, osy x, y a z. Matice tedy vypadá takto:

$$\mathbf{W} = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ Q_x & Q_y & Q_z & 1 \end{pmatrix}.$$

Protože odhadnout tyto čtyry vektory není zrovna jednoduché, matice W se konstruuje jako skládání třech transformací. $\mathbf{W} = \mathbf{SRT}$. Kde škálovací matice



Obrázek 5: Vertexy každého objektu jsou definovány ve svém souřadnicovém systému. Navíc určujeme pozici a orientaci každého lokálního souřadnicového systému vzhledem k globálnímu souřadnicovému systému, na základě pozice a orientace, kde chceme aby se objekt nacházel.

S škáluje objekt do W-prostoru, následně rotační matice **R**, která udává orientaci L-prostoru vzhledem ke W-prostoru, a nakonec translační matice **T** udává počátek L-prostoru ve W-prostoru. Jednotlivé matice vypadají takto:

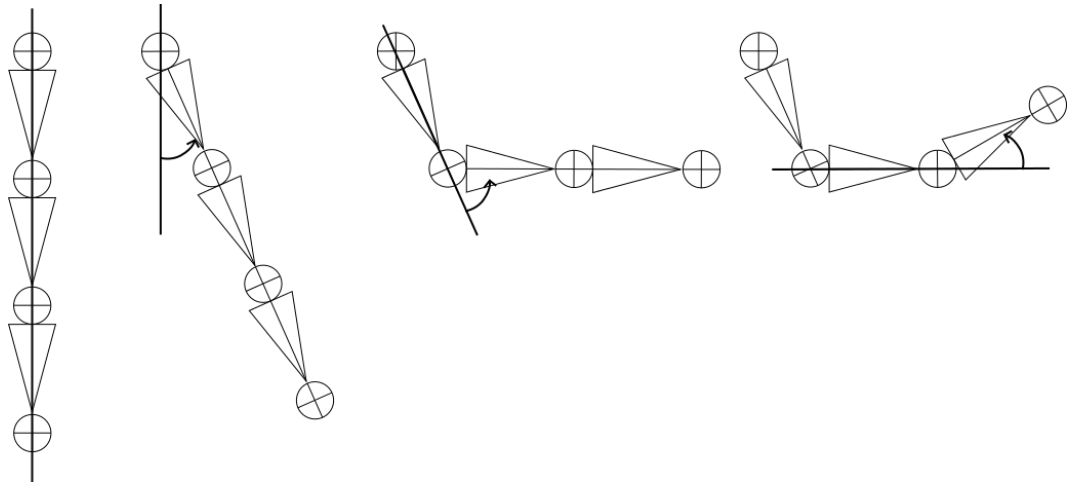
$$\mathbf{S} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

kde s_x škáluje podle osy x, s_y podle osy y a s_z podle osy z,

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ b_x & b_y & b_z & 1 \end{pmatrix},$$

kde b_x je posunutí po ose x, b_y po ose y a b_z po ose z. Když chceme posunout vektor **u** nebo bod **p**, tak je rozšíříme na 4D vektor $\mathbf{u}' = (\mathbf{u}, 0)$ nebo 4D bod $\mathbf{p}' = (\mathbf{p}, 1)$. To je z důvodu, že nechceme aby translace ovlivňovala vektory. Takhle jí nebudou ovlivněny, ale body budou posunuty, jak mají.

U rotační matice **R** máme více možností, jak ji zkonstruovat. Jedna možnost ji popsat z Eulerových úhlů, jak jsme uvedli v 2.1. Matici zkonstruujeme stejně, poté rozšíříme 4. řádek a 4. sloupec nulami, s vyjma komponentu r_{44}, kde bude 1.



Obrázek 6: Hierarchická transformace. Rodičovská transformace ovlivňuje všechny své potomky.

Druhý způsob, bližší pro animace, je zkonstruovat ji z kvaternionu. Jak jsme si ukázali v 3. Matici rozšíříme o 4. řádek a 4. sloupec stejně jako u předchozího způsobu.

3.1.2 Lineární interpolace

Je zobrazení pro dvě reálná čísla a, b a $t \in [0, 1]$ $lerp(a, b, t) = (1 - t)a + tb$. Lerp vrací číslo mezi a a b podle parametru t. Čím je menší, tím blíže je výsledné číslo k a a naopak. Zobrazení se může používat i nad vektory $\mathbf{u}$ a $\mathbf{v}$, kde komponenty výsledného vektoru jsou výsledky lineární interpolace komponentů $\mathbf{u}$ a $\mathbf{v}$. Čili ve 3D $lerp(\mathbf{u}, \mathbf{v}, t) = (lerp(u_x, v_x, t), lerp(u_y, v_y, t), lerp(u_z, v_z, t))$. Lineární interpolace se používá v animaci pro translaci a škálování mezi framy animace, viz 3.3, nebo pro různé efekty jako mlha, kde viditelnost se má se vzdáleností zhoršovat.

Pro interpolaci kvaternionů používáme tzv. sférickou lineární interpolaci. Pro kvaterniony $\mathbf{a}$, $\mathbf{b}$ a číslo t máme následující rovnici:

$$slerp(\mathbf{a}, \mathbf{b}, t) = \frac{\sin((1-t)\theta)\mathbf{a} + \sin(t\theta)\mathbf{b}}{\sin\theta} \quad \text{pro} \quad t \in [0, 1].$$

Slerp plní stejnou funkci jako lerp, ale pro kvaterniony. Čili vrací kvaternion umístěný mezi kvaterniony $\mathbf{a}$ a $\mathbf{b}$ v závislosti na parametru t. Její efekt můžeme vidět na obrázku 3.3.

3.2 Hierarchie

Hodně objektů se skládá z částí, které mají mezi sebou stromovou strukturu (rodič-potomek), kde se někteří potomci můžou hýbat nezávisle na sobě, ale jsou nuceni se hýbat, když se hýbe jejich rodič. Například, představte si horní končetinu, která se skládá z paže, předloktí a ruky. Ruka se může otáčet v izolaci

kolem zápěstí (co jí klouby dovolí), ale když se otočí předloktí kolem lokte, ruka se musí pohnout s ním. Podobně když se otočí paže, předloktí i ruka se točí s ní. Můžeme vidět jednoznačnou hierarchii objektů. Ruka je potomek předloktí, předloktí je potomek paže je potomek trupu a tak dále až se dostaneme ke kořenu kostry.

Když máme objekt v hierarchii, můžeme ho transformovat přímo do W-prostoru? Teoreticky by to šlo, ale nemělo by to kýžený výsledek. Nesmíme totiž opomenout transformace všech předků, protože ty taky ovlivňují umístění ve scéně. Každý objekt v hierarchii je modelován kolem svého souřadnicového systému s počátečním kloubem (např. loket v předloktí) jako počátkem pro umožnění rotace.

Protože všechny souřadnicové systémy existují ve stejném univerzu, můžeme mezi nimi vytvořit vazby. Konkrétně, v libovolný čas můžeme popsat každý souřadnicový systém vzhledem k souřadnicovému systému svého rodiče. (Souřadnicový systém rodiče kořene snímku F_0, viz 3.3, je globální souřadnicový systém. To jest souřadnicový systém F_0 je popsán vzhledem ke globálnímu souřadnicovému systému.) Teď když jsme vytvořili vazby mezi souřadnicovými systémy rodiče a potomka, můžeme transformovat L-prostor potomka do L-prostoru předka s transformační maticí. Tohle je stejný způsob jako local-to-world transformace. Místo abychom transformovali L-prostor do W-prostoru, transofrmujeme do L-prostoru rodiče.

Nechť $\mathbf{A}_2$ je matice, která transformuje geometrii ze snímku F_2(animace se skládá z tzv. key framů, popíšeme si v další kapitole) do F_1, nechť $\mathbf{A}_1$ je matice, která transformuje geometrii ze snímku F_1 do F_0, a nechť $\mathbf{A}_0$ je matice, která transformuje geometrii ze snímku F_1 do W. (Matice $\mathbf{A}_1$ se nazývá rodičovská (to-parent) matice, protože transformuje geometrii z lokálního souřadnicového systému do souřadnicového systému jeho rodiče.) Pak tedy, můžeme transformovat i-tý objekt v hierarchii horní končetiny do W-prostoru podle matice $\mathbf{M}_i$ definované takto:

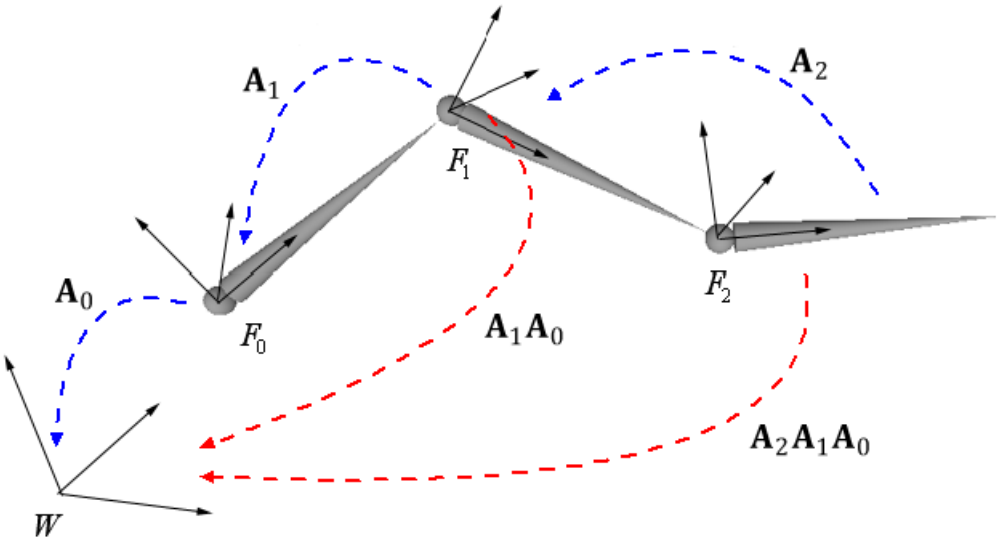
$$\mathbf{M}_i = A_i A_{i-1} \cdots A_1 A_0. \tag{4}$$

Konkrétně v našem příkladě horní končetiny, $\mathbf{M}_2 = \mathbf{A}_2\mathbf{A}_1\mathbf{A}_0$, $\mathbf{M}_1 = \mathbf{A}_1\mathbf{A}_0$ a $\mathbf{M}_0 = \mathbf{A}_0$ popořadě transformuje ruku do W-prostoru, předloktí do W-prostoru a paži do W-prostoru.

Na obrázku 3.2 je vidět, co rovnice 4 představuje graficky. V podstatě abychom transformovali objekt/kost v hierarchii horní končetiny, aplikujeme rodičovskou transformaci pro tento objekt a pro všechny jeho předky, abychom se dostali řetězcem souřadnicových systému až do toho globálního.

Příklad animace horní končetiny je jednoduchá lineární hierarchie. Ale stejným způsobem se to dá rozšířit na stromovou strukturu. Čili, W-prostoru transformaci najdeme postupnou aplikací rodičovských transformací objektu a všech jeho předků až se objekt dostane do W-prostoru. Jediný rozdíl je, že máme komplikovanější strukturu k procházení oproti lineárnímu seznamu.

Příklad složitější hierarchie máme na obrázku 3.2. Když vezmeme v úvahu levou klíční kost na obrázku 3.2, uzel je sourozenec krku a potomek horní pá-



Obrázek 7: Popisujeme každý souřadnicový systém kosti vzhledem k souřadnicovému systému svého rodiče, z čehož následně vytvoříme rodičovskou (to-parent) transformační matici, která transformuje geometrii kosti z lokálního souřadnicového systému do souřadnicového systému rodiče. Takhle postupujeme stromem nahoru až ke kořenu, čímž se dostaneme do W-prostoru.

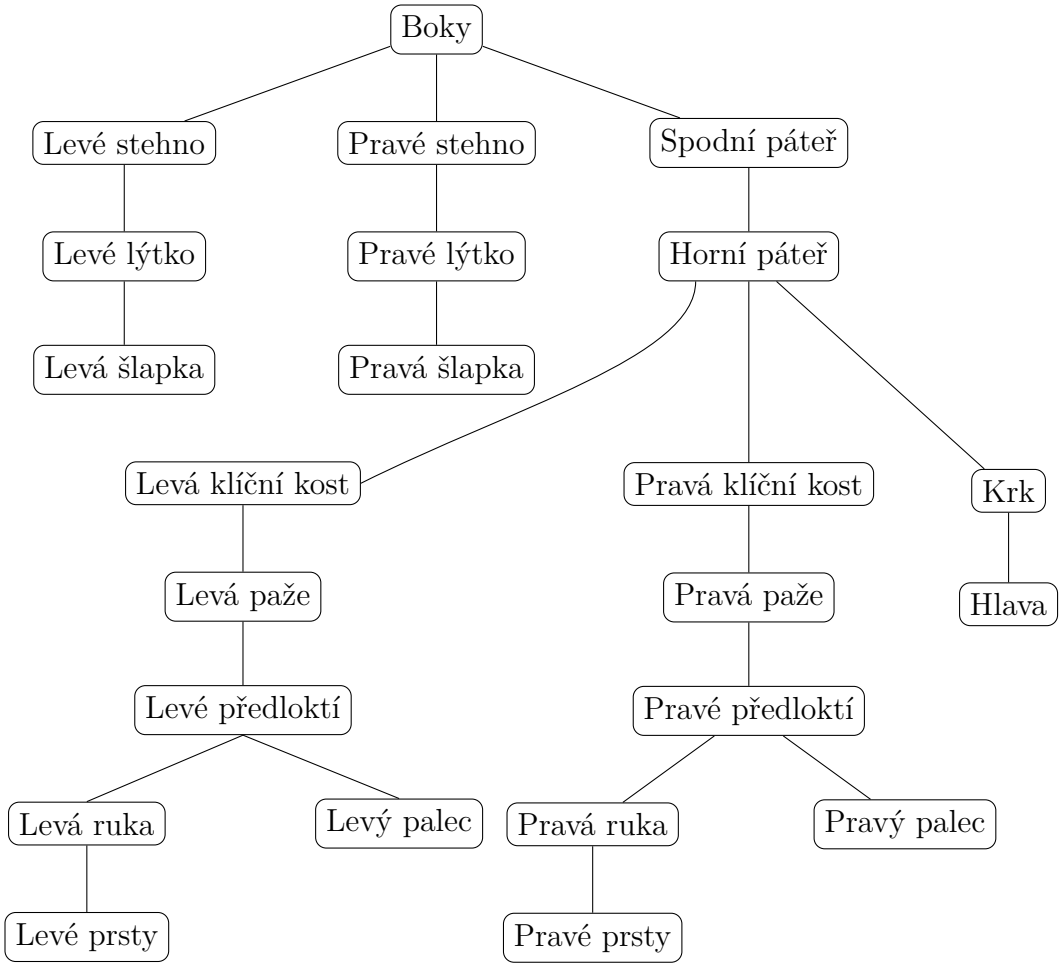
teře. Ta je zase potomek spodní páteře a spodní páteř je potomek boků. Takže world transform levé klíční kosti je tvořen skládáním vlastní rodičovské transformace, následované rodičovskou transformací horní páteře, následované rodičovskou transformací dolní páteře a nakonec následované rodičovskou transformací boků.

3.3 Key Frame

Bežná forma animace je nazývána key frame animace. *Key frame* určuje pozici, orientaci a škálování objektu v určitém okamžiku v čase. Uvnitř implementace bude mít asi takovou podobu:

```
struct KeyFrame
{
        float Time;
        Vector3 Translation;
        Vector3 Scale;
        Quaternion Rotation;
}
```

Animace kosti je potom seznam key framů uspořádaných podle času. Tvar key framů při čtení ze souboru už záleží na formátu. Například formát *m3d* je dobře čitelný pro lidi, ale hodně plýtvá zdroji:



Obrázek 8: Složitější stromová struktura dvojnohého modelu humanoida. Kořenem ve většině modelů jsou boky/kyčel. Uzly ve stejné vrstvě jsou sourozenci. Později uvidíme, že kosti z Kinectu mají velmi podobnou strukturu.

```
Bone1 #Keyframes: 76
{
  Time: 0 Pos: -0.5862113 -0.06803528 -0.06857681
  Scale: 1 1 1
  Quat: 0.4598411 -0.5545887 0.4610625 0.5180722
  Time: 0.0166666 Pos: -0.5728644 -0.276429 -0.06752014
  Scale: 0.9999999 1 1
  Quat: 0.4572482 -0.5526819 0.4629614 0.520705
  Time: 0.0333333
  Pos: -0.5844057 -0.4848228 -0.06646347
  Scale: 1 1 1
  Quat: 0.4552799 -0.5505438 0.4642273 0.5235596
                              ⋮
```

Například druhý key frame je v čase 0.01666 vteřin, s translačním vektorem $\mathbf{t} = (-0.5728644, -0.276429, -0.06752014)$, škálovacím vektorem $\mathbf{s} = (0.9999999, 1, 1)$, kde můžeme vidět nepřesnost kvůli kódování čísla s plovoucí čárkou, a nakonec rotačním kvaternionem $\mathbf{r} = (0.4598411, -0.5545887, 0.4610625, 0.5180722)$. Formát $m3d$ se v praxi nepoužívá . Je to spíše formát pro počáteční účely, protože čtečku si jsme chopni napsat sami.

Podíváme se, key frame kóduje ve formátu Collada $m3d$.

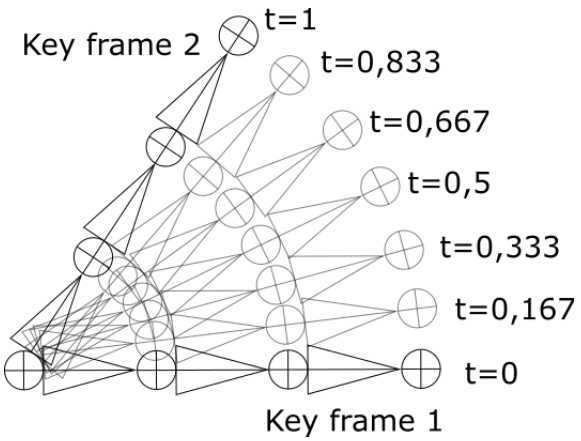
```
<animation id="arm-transform">
  <source id="arm-transform-input">
    <float_array id="arm-transform-input-array" count="40">
      0 0.033333 0.066666 0.1 0.133333 0.166667 0.2 ...
                                        ⋮

  <source id="arm-transform-output">
    <float_array id="arm-transform-output-array" count="640">
      0.638262 0.33767 0.691809 0.729258 0.367305 0.656195
      -0.659162 0.456132 -0.676542 0.674824 0.294797 -2.59935
       0 0 0 1 ...
                                        ⋮
```

Jak můžeme viděl, collada používá xml ke kódování. Díky tomuto je daleko složitější a je schopná se odkazovat ne různé prvky v souboru díky *id* tagu. Collada kóduje zvlášť čas, kde jednotlivé časy (40) můžete vidět v prvním float_array, a celé transformační matice ve druhém float_array. *count* zde znamená počet čísel, které se mají přečíst. Takže počet matic v této animaci bude $640/16 = 40$.

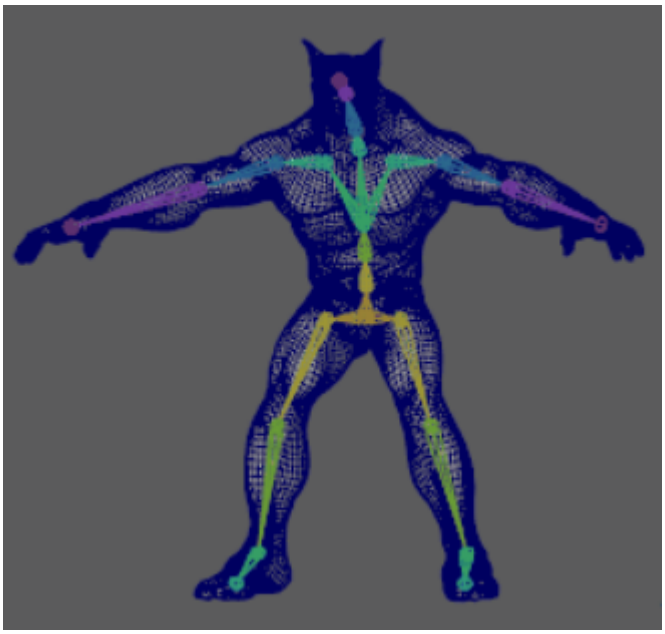
Když načteme seznam key framů, tak nám to dá zevrubnou animaci. Zbytek key framů se dopočítává interpolací (lineární pro vektory a sférickou pro kvaterniony), aby se vyrovnaly výkonu počítače, jak můžeme vidět na obrázku 3.3.

3.4 Skinned meshes

Obrázek 3.4 ukazuje mesh postavy. Zvýrazněný řetězec kostí ve figuře se nazývá *kostra*. Kostra poskytuje přirozenou hierarchickou strukturu pro řízení animačního systému postavy. Kostra je obklopena vnější *kůží* (angl. skin), kterou modelujeme jako 3D geometrii (vertexy a polygony). Původně jsou vertexy umístěny vzhledem k *B-prostoru* (angl. *bind space*), což je lokální souřadnicový systém, vzhledem ke kterému je celá kůže definována. Většinou bývá počátek $(0, 0, 0)$ ve středu těla modelu. Každá kost v kostře ovlivňuje tvar a pozici části kůže (její vertexy). Když animujeme kostru, přidružená kůže se animuje v souladu s kostrou, která zaujímá nějakou pózu.



Obrázek 9: Key frame interpolace. Key farmy určují „klíčové“ pózy animace. Interpolované hodnoty reprezentují hodnoty mezi key framy.



Obrázek 10: Mesh postavy se zvýrazněnou kostrou.

3.4.1 Předefinování kořenové transformace kostí

Jeden rozdíl oproti kapitole 3.2 je, že transformujeme z kořenového souřadnicového systému v jednom kroku. Místo abychom počítali světovou matici (to-world matrix), počítáme *kořenovou matici* (angl. to-root matrix) pro každou kost. (Transformace, která transformuje lokální souřadnicový systém kosti do souřadnicového systému kořenové kosti.) Přece jenom samotný model, který obsahuje kostry se svými transformačními maticemi, obsahuje také transformační matici.

Druhý rozdíl oproti kapitole 3.2 je, že tam jsme procházeli stromem kostí zdola-nahoru, ale v praxi je výhodnější přístup shora-dolů. Začínáme u kořene a pohybujeme se stromem dolů. Označíme n kostí přirozeným číslem $0, 1, ..., n-1$, máme následující rovnici vyjadřující kořenovou transformaci:

$$toRoot_i = toParent_i \cdot toRoot_p. \tag{5}$$

Kde p je index rodiče kosti i. Tímhle můžeme přeskočit procházení stromu nahoru od kosti, protože $toRoot_p$ zobrazuje geometrii kosti p do souřadnicového systému kořene. Jediné, co musíme udělat, je dostat geometrii souřadnicového systému kosti i do souřadnicového systému rodiče, kosti p, což obstará $toParent_i$.

Problém by mohl nastat, kdybychom se dostali k potomku dříve než k rodiči. Ale vzhledem k tomu že procházíme shora-dolů, tak tento stav nemůže nastat. Navíc je to rychlejší, protože pro kost i máme kořenovou transformaci jejího rodiče. Takže nemusíme procházet strom až ke kořenu, jsme jenom jeden krok od kořenové transformace kosti i.

3.4.2 Offset transformace

Tato a následující kapitola budou doplňkové. K animacím jednoznačně patří, ale už k jejich implementaci. Konkrétně se jedná o transformaci vertexů ve vertex shaderu, což je program grafické karty. Pro tvorbu animace nejsou tyto kapitoly důležité.

Ještě musíme vyřešit jeden problém. Ten vychází z toho, že vertexy, které jsou ovlivněné kostí, nejsou modelovány vzhledem k souřadnicovém systému té kosti. Souřadnicový systém všech vertexů je B-prostor, čili souřadnicový systém, ve kterém byl model modelován. Takže předtím, než použijeme rovnici 5, musíme převést vertexy z B-prostoru do L-prostoru kosti, která je ovlivňuje. K tomu slouží takzvaná *offset transformace*, viz obrázek 3.4.2. Matici, která reprezentuje offset transformaci najdeme zakódovanou v modelu.

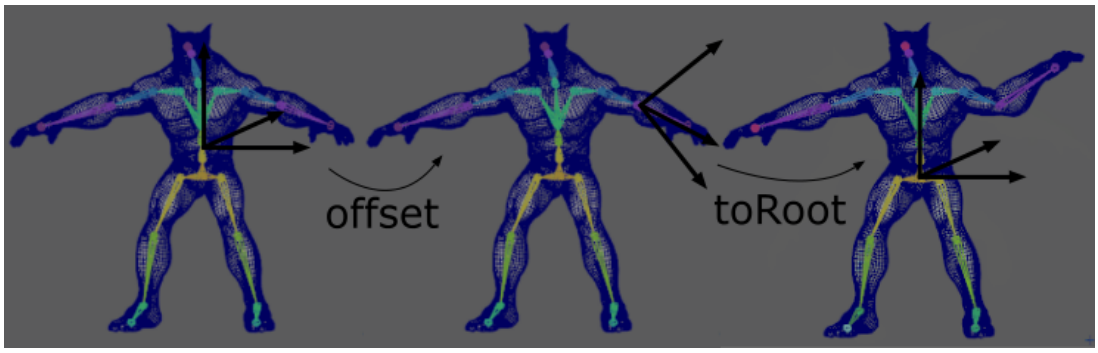
Takže transformováním vertexů offset maticí libovolné kosti B, přesuneme vertexy z B-prostoru do L-prostoru kosti B. Následně můžeme použít kořenovou transformaci kosti B, abychom je umístili do souřadnicového systému modelu v nějaké pózy animace.

Uvedeme novou transformaci, která zkombinuje offset transformaci a rodičovskou transformaci kosti, nazvanou *finální transformací*. Matematicky vypadá takto:

$$\mathbf{F}_i = offset_i \cdot toRoot_i.$$

3.4.3 Vertex Blending

Algoritmus, který animuje vertexy kůže, která pokrývá kostru, se nazývá *vertex blending*. Nebudeme se zabývat celým algoritmem, jenom si představíme ještě

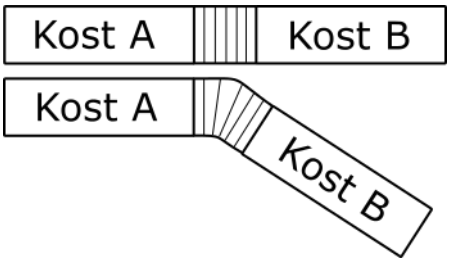


Obrázek 11: První transformujeme vertexy ovlivněné kostí z B-prostoru do prostoru kosti, která je ovlivňuje, offset transformační maticí. Potom použijeme kořenovou transformaci kostí, abychom dostali vertexy do souřadnicového systému kořene.

jeden problém spojený s animací, čímž jsou samotné klouby. Ty jsou totiž ovlivněné více kostmi, konkrétně těmi, které spojuje.

To vyřešíme tak, že vertexům (struktuře, ne pozici) přidáme váhy a indexy, které kosti je ovlivňují. Cílem bude vytvořit transformační matici pro vertex, která by byla vážený průměr transformačních matic kostí, které vertex ovlivňují. Tyto váhy se tvoří s modelem, my je budeme už jenom používat.

V praxi se používají nanejvýš 4 váhy na vertex, takže z toho budeme vycházet. Každý vertex jednolitého meshe, bude pro naše účely obsahovat translační vektor, 4 indexy kostí ovlivňující vertex a 4 váhy, které určují, jakou daná kost má vliv na vertex.



Obrázek 12: Ohyb kostí v kloubu.

Jednolitý mesh, jehož vertexy mají takovou podobu je připravený na vertex blending a nazývá se *skinned mesh*. Pozice vertexu $\mathbf{v}'$ libovolného vertexu $\mathbf{v}$ v souřadnicovém systému kořenové kosti (do W-prostoru převádíme až nakonec) se vypočítá jako rovnice váženého průměru transformací:

$$\mathbf{v}' = w_0\mathbf{v}\mathbf{F}_0 + w_1\mathbf{v}\mathbf{F}_1 + w_2\mathbf{v}\mathbf{F}_2 + w_3\mathbf{v}\mathbf{F}_3,$$

kde $w_0 + w_1 + w_2 + w_3 = 1$ a $\mathbf{F}_1, ..., \mathbf{F}_3$ jsou finální transformace kostí (které jsme představili v minulé kapitole). Takže transformujeme pozici vertexu $\mathbf{v}$ všemi finálními transformacemi kostí, které ho ovlivňují. Potom vezmeme vážený průměr transformovaných bodů, abychom spočetli konečnou podobu pozice vertexu $\mathbf{v}$'.

A to je vše. V této kapitole jsme si tedy ukázali, jak v animaci hýbeme kostmi,

které zase ovlivňují jednotlivé vertexy přidruženého meshe. V následujících 2 kapitolách si ukážeme body, které dostáváme na vstup, a jak z oněch bodů prostoru vytvoříme animaci.

4 Kinect

Kinect je série produktů s detekcí pohybu vyvinutých Microsoftem pro herní konzole Xbox 360 a Xbox One a později pro Windows PC. Je založen na stylu periferie webkamery, umožňuje to uživatelům ovládat a reagovat s jejich konzolí nebo počítačem bez potřeby herního ovladače použitím gest a mluvených příkazů [**15**].

První generace Kinectu pro Xbox 360 byla představena koncem roku 2010 v pokusu rozšířit hráčskou základnu. Začátkem roku 2012 byla vydáno SDK pro podporu PC na vývoj komerčních aplikací. SDK měla umožnit vývojářům psát aplikace pro Kinect v C++, C# nebo Visual Basic .NETu.

Kinect pro Xbox One, nová verze s rozšířenými možnostmi byla vydána v roce 2013. Hardware *Kinect for Windows v2* a příslušná SDK byly vydány o rok později.

Kvůli špatným recenzím a nízkým prodejům, Microsoft přestal vyrábět a prodávat i verzi Kinectu pro Xbox One v říjnu roku 2017.

Duchovní nástupce Kinectu, *v3* se dá považovat product od Microsoftu *HoloLens* a *v4* projekt Kinect pro Azure, mířený na firemní použití.

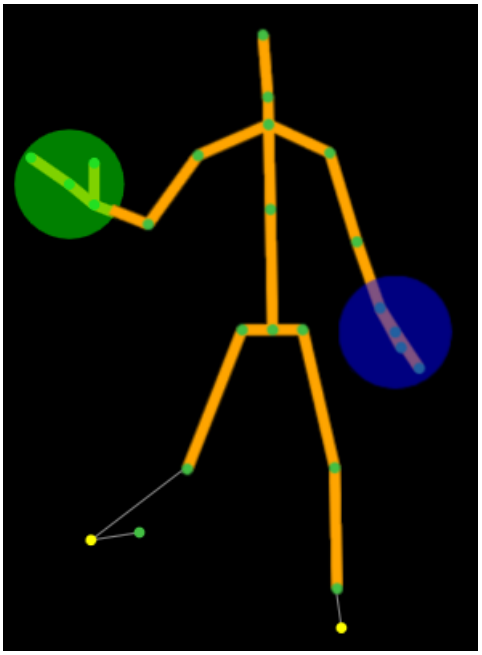
4.1 Kinect pro Xbox One

Tato vylepšená verze Kinectu, kterou budeme následně používat v aplikaci, používá širokoúhlou time-of-flight kameru. ToF kamera vrací 2D snímek vzdáleností bodů ve scéně z určitého bodu. Snímek získá měřením „letu“ světla mezi objektem a kamerou pro každý bod snímku. Kinect pořizuje snímky 30Hz frekvencí a zpracovává 2 gigabity dat za sekundu při čtení prostředí. Oproti svému předchůdci má mnohem větší přesnost, o 60% širší záběr a používá infračervený senzor k zachycení světla. Také dokáže zachytit až 6 lidí naráz. To za cenu, že nelze číst data z jednoho počítače ze dvou Kinectů. Pro nás je důležité, že dokáže detekovat pozici kloubů, gesta a výrazy tváře. Pro připojení k počítači vyžaduje speciální adaptér.

4.2 Data z Kinectu

Knihovna obsluhující data z Kinectu převádí snímky vzdáleností do několika struktur. (Nebudeme rozlišovat mezi funkcí Kinectu a jeho knihovnou.) Kde pro nás nejzajímavější struktura je *Body*. *Body* představuje data zachycená od jednoho člověka.

Nejdůležitější kolekcí obsaženou v této struktuře je seznam pozice kloubů, které Kinect zachytil. Tyto klouby (*Joint*) jsou struktury tří prvků. Jmenovitě



Obrázek 13: Data z kinectu vyobrazená na 2D plátno. Oranžové úsečky jsou kosti, zelené kruhy u kostí jsou klouby a velké kruhy u rukou představují gesta. Nezachycení kosti je vyobrazeno tenkou šedou linkou.

typ kloubu, jestli kloub byl zachycen Kinectem a pozici v prostoru. Seznam kostí z grafu 3.2 je shodný se seznamem kostí z Kinectu, až na rozdělení boků na levý bok, pravý bok a začátek páteře.

Souřadnicový systém pro body $p = (x, y, z)$ je dán fyzickým umístěním Kinectu. Ten je umístěn v počátku $0, 0, 0$. Pohyb směrem kolmo od nebo ke Kinectu představuje pohyb na z souřadnici. Ta může být jenom kladná. Pohyb nahoru a dolů představuje pohyb na y souřadnici a pohyb doleva doprava souběžne s Kinectem je pohyb na x souřadnici.

Struktura *Joint* vypadá takto:

```
struct Joint
{
  JointType jointType;
  Point position;
  TrackingState trackingState;
}
```

Body obsahuje další prvky, jako jsou gesta rukou a výraz tváře, ale pro nás nemají využití.

4.3 Omezení

Omezení Kinectu vychází z jeho primárního zaměření. To jest používání Kinectu jako součást Xboxu, kde uživatel je otočen směrem k televizi a předává Kinectu informace pomocí gest (otevřená ruka, ruka v pěst apod.) a pohybu. Problém nastane při rotaci uživatele. Kinect je totiž kalibrován na uživatele, který je orientován čelem ke Kinectu a nedokáže rozlišit mezi orientací uživatele. Čili pro Kinect není rozdíl, jestli je uživatel k němu otočen čelem nebo zády. Tím pádem Kinect nepozná rozdíl mezi pravým a levým kloubem. Vždy vrátí pravý kloub jako ten, který je snímán vpravo z pohledu Kinectu a levý kloub jako ten, co je snímán vlevo z pohledu Kinectu. To znamená, když je uživatel otočen směrem ke Kinectu, tak např. uživatelovo levé rameno je v datech reprezentováno jako pravé. Naopak, když je uživatel otočen zády ke Kinectu, tak jsou orientace kloubů shodné.

Související problémem je, že Kinect „nepředpokládá“ opačnou orientaci (zády ke Kinectu) a vrací data s daleko menší přesností. Často jsou značně mimo a pro účely animace jsou nepoužitelné.

A poslední limitací je pohyb na y (směr nahoru) souřadnici. Fyzické umístění Kinectu ovlivňuje pozice kloubů v prostoru. Nejvíce je tímto ovlivněna Y souřadnice pozic kloubů. Když máme Kinect umístěn nad středem těla, tak s rostoucí vzdáleností uživatele směrem od Kinectu budou data pozice středu těla klesat a naopak. To vychází z faktu, že když zobrazíme 3D prostor na 2D plátno, tak se objekty zmenšují směrem do středu. U počítání vektorů z bodů tento fakt nemá vliv, ale konzistentní pohyb s modelem nahoru a dolů, je velice obtížný implementovat.

5 Algoritmus

V této kapitole rozebereme inverzní algoritmus oproti animacím postav, který převádí vektory ve W-prostoru na transformaci W-prostoru do L-prostoru, z čeho pak můžeme vypočítat inverzní transformaci.

5.1 Pseudokód

Algoritmus zde předtavuje funkce *Animate*. Na jejím vstupu máme kost (uzel) *Bone*, strukturu *Body* a čas ve kterém se má pozice animace nacházet. Každá kost má seznam dětí, jednoho rodiče a inverzní kvaternionovou rotaci. *Body* obsahuje seznam pozic kloubů. Strom kostí (kostru) už máme předem vytvořenou, konkrétně v aplikaci odpovídá 23 kostem z grafu 3.2

Input:

Kost $B = \langle C, P, r \rangle$, kde C_i jsou potomci $\{C = c_1, ..., c_k\}$, P je rodič, a r je kvaternion.

Body je strukruta z Kinectu obsahující pozice kloubů.

t je čas pozice animace.

```
Animate( B , Body , t )
  direction := get corresponding normalized vector from Body ;
  direction := transform direction into original local space;
  traverse all predecessors of B :
    direction := apply parent's r_p on direction;
  rotation := get quaternion rotation between
              direction and vector u = (0,1,0) ;
  r = inverse rotation;
  keyFrame := create key frame from rotation, static data and t;
  pass keyFrame to an animation;
  call Animate( c_i , Body , t ) on every child c_i in B ;
```

Na začátku funkce vypočteme vektor *direction* z kloubů *Body*, který odpovídá dané kosti. Např pro předloktí vypočteme vektor z pozic lokte a zápěstí.

Jelikož máme souřadnice kloubů ve W-prostoru, potřebujeme je transformovat do L-prostoru. Nejdříve vektor daný klouby transformujeme z W-prostoru do výchozího L-prostoru, kde ještě nenastala žádná transformace. Většinou to bývá rotace o 90 stupňů směrem, závislé na konkrétní části těla. Více v kapitole 5.3.

Vektor *direction* reprezentuje klouby v prostoru nezávisle na předchůdcích. Potřebujeme ho tedy transformovat přes všechny souřadnicové systémy předků. Proto na *direction* aplikujeme kvaternionovou transformaci r_p pro všechny předky kosti B.

Kvaternion *rotation* vypočteme jako kvaternion představující rotaci mezi dvěma vektory, konkrétně mezi $\mathbf{u} = (0, 1, 0)$ a *direction*. Vektor $\mathbf{u} = (0, 1, 0)$ je původní směr kostí v jejich L-prostoru, proto vytváříme kvaternion z něho a *direction*. Jak se tenhle kvaternion počítá ukážeme v další kapitole.

Poté uložíme inverzní kvaternion (konjugaci kvaternionu), aby ho potomci mohli použít. Následně vytvoříme klíčový snímek a předáme ho do animace. Na závěr se funkce *Animate* zavolá na všechny potomky kosti B.

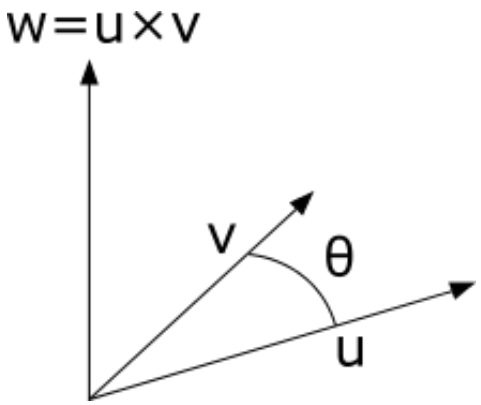
Po skončení poslední iterace funkce máme pohyby všech kostí v konkrétní čas. Jak se předá klíčový snímek animaci závisí na implementaci a není podstatný.

5.2 Výpočet kvaternionů ze dvou vektorů

Rotace je nejlépe představitelná pomocí osy, kolem které rotujeme, a úhlu. Až na extrémní případy (vektory jsou rovnoběžné) může být osa získána vypočítáním vektorového součinu dvou původních vektorů. Viz obr. 5.2. Potom úhel může být vypočítán použití vlastností vektorového nebo skalárního součinu dle:

$$\begin{aligned} \mathbf{u} \cdot \mathbf{v} &= |\mathbf{u}| \cdot |\mathbf{v}| \cos\theta \\ \|\mathbf{u}\| \times \mathbf{v} &= |\mathbf{u}||\mathbf{v}||\sin\theta| \,. \end{aligned}$$

Protože hodnota θ je vždy mezi $[0, \pi]$, tak nás zajímá jenom skalární součin. Z čehož dostaneme jednoduchý kód podle [**4**]:



Obrázek 14: Vektorový součin představující osu otáčení mez dvěma vektory.

```
Quaternion FromVector(vector3 u, vector3 v)
{
    cosTheta = dot(norm(u), norm(v));
    angle = acos(cosTheta);
    w = norm(cross(u,v));
    return fromAxisAngle(w, angle)
}
```

Funkce fromAxisAngle představuje výpočet z rovnice 2.

5.3 Transformace do L-prostoru

Nyní ještě upřesníme transformaci kosti z W-prostoru do výchozího L-prostoru, jelikož si tohle bude moci uživatel sám nastavit v aplikaci. Tohle totiž záleží na konkrétním modelu. V aplikaci jsou použity modely, které mají orientaci kostí ve W-prostoru, tak jak bychom mohli předpokládat. Kosti levé ruky, pravé ruky, trupu, nohou a chodidel jsou orientovány k vektorům popořadě

$$\begin{aligned}\mathbf{l} &= (1,0,0)\\ \mathbf{p} &= (-1,0,0)\\ \mathbf{t} &= (0,-1,0)\\ \mathbf{n} &= (0,1,0)\\ \mathbf{c} &= (0,0,1).\end{aligned}$$

Směr vektoru určuje implementace aplikace. Ta počítá vektory směrem z kořene (boků) ke koncům končetin. Kdybychom počítali vektory z opačné strany, tak vektory budou mít opačnou orientaci.

Problémem je, že některé modely nemusí mít takto orientované kosti ve W-prostoru. Např. existují modely, kde vektory směru kosti rukou mají shodný směr. Může se i stát, že jediná kost v části těla modelu (např. trupu) modelu bude mít jinou orientaci než ostatní. Proto si může uživatel v aplikaci nastavit vlastní transformaci z W-prostoru do výchozího L-prostoru.

Jelikož se jedná o rotace násobky $\pi/2$, tak se rotace kóduje změnou souřadnic (komponentů) vektorů, místo rotace kolem os x, y, z. Konkrétně v aplikaci je výchozí výpočet rotace vektoru $\mathbf{u} = (u_x, u_y, u_z)$ vyjádřen pro levou ruku, pravou ruku, trup, nohy a chodidla popořadě jako:

$$\begin{aligned}\mathbf{l} &= (-u_z, -u_x, u_y)\\ \mathbf{p} &= (u_z, u_x, u_y)\\ \mathbf{t} &= (u_x, -u_y, u_z)\\ \mathbf{n} &= (u_x, u_y, u_z)\\ \mathbf{c} &= (u_x, u_z, -u_y).\end{aligned}$$

Tedy kosti levé a pravé ruky rotujeme okolo osy z o $\pi/2$ a $-\pi/2$, kosti trupu rotujeme okolo osy x o π, kosti nohou zůstávají stejné a kosti chodidel rotujeme okolo osy x o $\pi/2$.

Když máme vektor takto transformovaný, pak ho můžeme transformovat rodičovskou transformaci všech předků.

6 Aplikace

V neposlední řadě se dostáváme k popisu samotné aplikace. Aplikace je naprogramovaná v jazyce C# a je postavená na knihovně Helix Toolkit pro WPF [**6**]. Což je knihovna, která využívá wrapper nad DirectX, SharpDX [**7**] pro vykreslování 3D grafiky. Takto kombinuje vysoký výkon 3D vykreslování s tvorbou grafického rozhraní.

6.1 Požadavky na systém

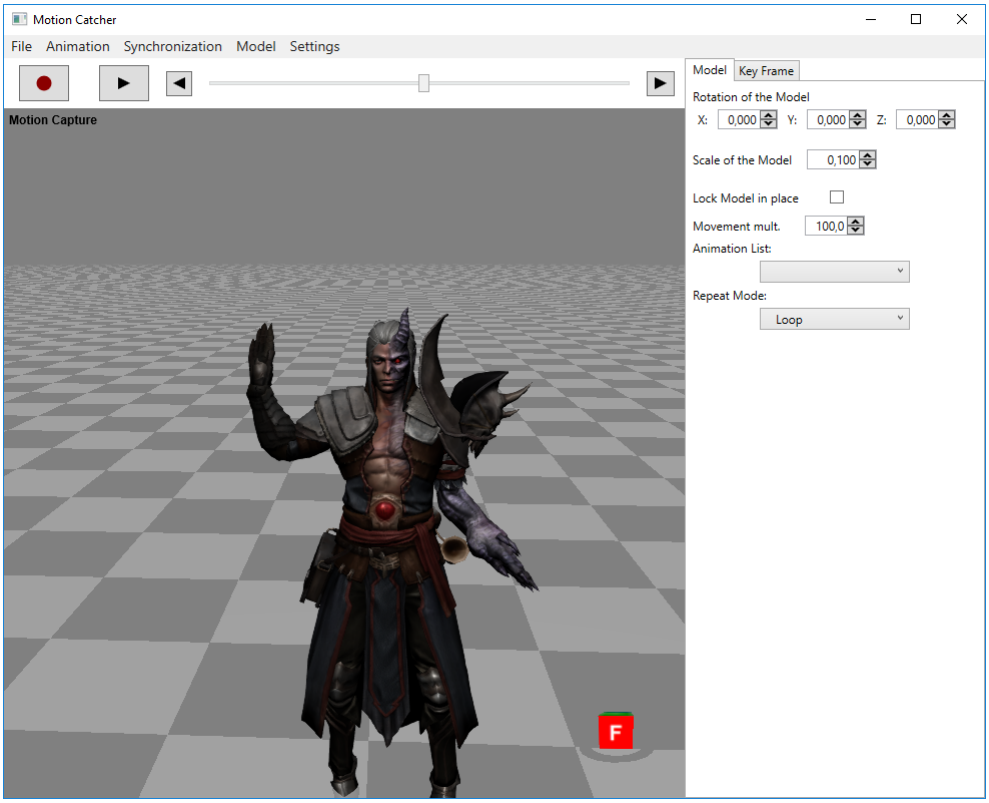
- Hlavní požadavkem pro hardware je grafická karta s podporou DirectX 11. Zbylé požadavky většina dnešních počítačů bude splňovat. Pro plnou specifikaci viz [**8**].
- USB 3.0. Starší počítače, obzvláště notebooky, sice mohou mít USB 3.0, ale i tak nemusí být dostatečně rychlá sběrnice pro provoz Kinectu.
- Windows 8, 8.1 nebo Windows 10.
- Podporu .Net Framework verze 4.6.2
- Kinect v2 a adaptér pro připojení k počítači.

6.2 Příprava

Po propojení Kinectu s počítačem přes USB 3.0, vyčkejte, než se nainstalují ovladače. Po jejich nainstalování můžete spustit aplikaci. Jestli se Kinect rozsvítil, tak se Kinect s počítačem propojili a můžete aplikaci používat. Kinect nenabízí

žádné rozhraní pro kontrolu, jestli byl počítačem nalezen nebo jestli je počítač schopen Kinect obsluhovat. Jediná kontrola je tedy fyzické světlo z Kinectu. Pro ideální Mo-Cap je třeba umístit Kinect asi jeden metr vysoko s neblokovaným výhledem. Nejlépe na kraj stolu, aby měl Kinect v záběru i dolní končetiny. Kinect lze přesouvat i po spuštění aplikace. Uživatel by se měl pohybovat asi 3 metry od Kinectu otočený čelem k němu pro nejlepší výkon.

6.3 Popis aplikace



Obrázek 15: Screenshot z aplikace.

6.3.1 Kamera

Po startu aplikace nás přivítá šedá šachovnice. To je viewport 3D scény, čili její zobrazení na 2D plochu. Kameru scény můžeme ovládat myší. Konkrétně pohyb kamery se ovládá stiskem levého tlačítka myši, rotace se ovládá pravým tlačítkem myši a zoom se ovládá rolováním kolečka myši. Kamera se dá také ovládat kliknutím na stranu krychle dole vpravo. Přesměruje to kameru na určitou orientaci ve scéně. Klávesové zkratky pro tyto rotace jsou B, F, U, D, L, R.

6.3.2 Nahrání modelu

V prvé řadě k používání aplikace je potřeba nahrát model. Pokud uživatel není obeznámen s nahráváním vlastního modelu v kapitole, viz 6.3.7, nechť načte jeden z předpřipravený modelů. To se provede kliknutím na horní liště na menu Model a dalším kliknutím na 2. nebo 3. položku. Je na výběr ze dvou modelů, Ganfaul a Akai. Po nahrání modelu se automaticky zapne Mo-Cap, čili když se uživatel bude nacházet alespoň v jednom metru od Kinectu, pak aplikace bude zachytávat jeho pohyb přes Kinect a převádět ho na animaci modelu.

6.3.3 Nahrávání animace

Jestli chce uživatel animaci nahrát, pak lze tak učinit přes stisk tlačítka s červeným kruhem nahoře vlevo. Tato sekce mezi lištou a viewportem, slouží k ovládání nahrávání. Opětovným stiskem tohoto tlačítka se nahrávání vypne. Vpravo od něho je Play tlačítko, které přehraje právě nahranou animaci od začátku do konce. Opětovným stiskem Play tlačítka se přehrávání animace vypne. Ke krokování jednotlivých klíčových snímků slouží posuvník nahoře uprostřed. Tlačítka se šipky vlevo a vpravo od posuvníku krokují animaci o jeden klíčový snímek vzad nebo vpřed.[1]

6.3.4 Úprava nahrané animace

Když má uživatel nahranou animaci, tak se odemkne menu Key Frame vpravo nahoře. Zde může uživatel ovládat jednotlivé klíčové snímky. Nahoře lze vidět pořadí a čas snímku v animaci. Pod nimi jsou 3 tlačítka: Cut Left, Cut Frame a Cut Right. Levé tlačítko odstraní klíčové snímky od začátku animace po aktuální snímek, prostřední odstraní aktuální snímek a pravé odstraní klíčové snímky od aktuálního snímku po konec animace.

Pod nimi je ovládání jednotlivých kostí v klíčovém snímku. Nejprve si uživatel vybere kost ze seznamu, kterou chce ovládat a následně upraví dle libosti. Čili může ji posunout o vektor složený z prvních 3 hodnot, druhé 3 hodnoty slouží k rotaci kosti o osy x, y a z a v poslední řadě je možnost škálovat kost ve směru os x, y a z.

Výslednou animaci si může uživatel uložit k modelu stisknutím Animation a Save Animation na horní liště. Pokud má animace stejný název jako existující, bude jí nahrazena, viz 6.3.6.

6.3.5 Transformace modelu

Model se může transformovat podle základních potřeb uživatele. Činí se tak přes menu Model vedle menu Key Frame. Lze ho rotovat podobně jako klíčové snímky. Tato možnost slouží hlavně při nahrání modelu se souřadnicí z směrem

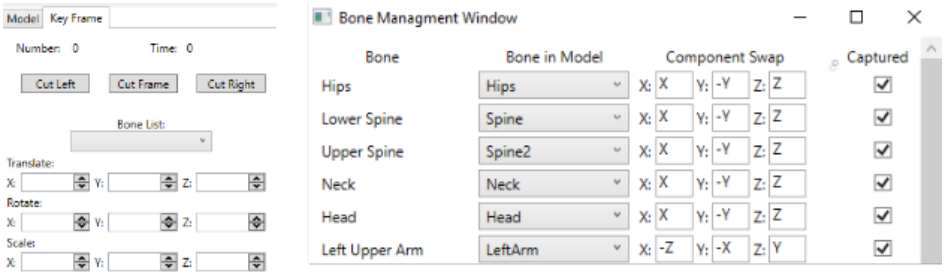
[1] V Helix Toolkitu je chyba, kdy se vykresluje samotná scéna před změnou transformačních matic. To má za následek při ovládání animace zpoždění o jednu akci, ať je jakákoliv.

vzhůru. V aplikaci je výchozí souřadnice vzhůru y. Jelikož modely nabývají různých velikostí, je často ho třeba přizpůsobit škálováním na vhodnou velikost. Tyto transformace nemají na animaci vliv.

Další funkce Lock in place už má na animaci vliv. Ta uzamkne model v počátku. Myšleno k použití k nahrávání animace, kde se očekává zajišťování pohybu modelu ze strany aplikace, která by ji využívala. Např. chůze, běh.

Další volitelnou hodnotou je násobitel pohybu. Čím větší je model v originálu (transformace modelu na tohle nemá vliv), tím ho bude méně ovlivňovat translace kostí. Proto je potřeba tuto translaci vynásobit při tvorbě animace hodnotou, aby se model pohyboval ke spokojenosti uživatele.

Poslední možností v tomto menu je přehrávání animace nahraných v modelu. Tyto animace nelze upravovat, jenom přehrávat ve způsobu vybraným v posledním seznamu: opakování nebo přehrání jenom jednou. Tento seznam ovlivňuje také přehrávání nahrané animace pomocí tlačítka Play.



Obrázek 16: Vlevo je Key Frame menu, vpravo výřez okna Bone Management

6.3.6 Lišta

Před popisem synchronizace Kinectů, kapitola 6.3.8, a nahrávání vlastního modelu, kapitola 6.3.7, si popíšeme celou horní lištu.

- File
  - Export - Exportuje celý model do binárního formátu assimpu assbin.[2]
- Animation
  - Start Capture - spustí Mo-Cap. Využití po přechodu z přehrávání nebo nahrávání animace.
  - Stop Capture - zastaví Mo-Cap.
  - Save Animation - uloží animaci k modelu, ne do souboru.
- Synchronization

[2]Knihovna/wrapper která obsluhuje knihovnu Assimp [**9**], Assimp.NET je pozadu ve vývoji za její mateřskou knihovnou a nezvládá export do jiných formátů.

  - Start Receiving - Tato instance aplikace bude označena jako server a začne naslouchat klienta. Při spuštění naslouchání se zobrazí IP adresa serveru.
  - Start Streaming - klient začne vysílat data serveru. IP adresa serveru se zadává v možnostech (Settings)
  - Stop Receiving - Server přestane naslouchat klienta.
  - Stop Streaming - Klient přestane posílat data serveru.

- Model
  - Manage Bones - umožní nastavit a upravovat chování jednotlivých kostí modelu.
  - Load Ganfaul - nahraje model Ganfaul.fbx.
  - Load Akai - nahraje model Akai.fbx.
  - Load Model from File - nahraje model ze souboru.

- Settings - spustí okno s nastavením
  - Host IP adress - IP adresa serveru, kam chce klient posílat data.
  - Port - port na který se vysílají data. Musí být stejný na obou instancích aplikace.
  - Show Skeleton - zobrazí kostru ve 3D reprezentující data z Kinectu, je podobná obrázku 4.2.
  - Animation Name - Ukládané animace se uloží pod tímto jménem.

6.3.7 Nahrání vlastního modelu

Po nahrání vlastního modelu je potřeba přiřadit kostem Kinectu kosti modelu. To lze nastavit v Model - Manage Bones. V tomto okně můžeme upravit chování kosti aplikace, viz. obrázek 6.3.5. Vlevo máme kost aplikace, vpravo od ní kost modelu vybranou ze seznamu všech kostí modelu. Dále vpravo je možnost nastavit vlastní transformaci do výchozího prostoru popsanou v kapitole 5.3. A poslední možnost je vybrat si, jestli chceme aby byla daná kost snímána. Tato možnost je především užitečná pro konce končetin. Prsty a chodidla nejsou snímány s velkou přesností a často se třepotají. Zde také můžeme pozorovat, že nezáleží na tom, když má model více kostí než Kinect. Tyto kosti prostě nebudou animované.

6.3.8 Synchronizace

Pro synchronizaci dvou Kinectů je umístěte kolmo od sebe aspoň 4 metry vzdálené. Jak již bylo zmíněno v kapitole 4.1, nelze číst data ze dvou Kinectů zároveň na jednom počítači. Proto k synchronizaci musíme mít 2 instance aplikace na dvou různých počítačích a data posílat přes místní síť. Uživatel nejprve zvolí,

která instance bude server, potom ji spustí příjímatí dat z klienta v Synchronization - Start Receiving. Vyskočí zpráva s místní IP adresou. Tu uživatel přepíše do nastavení klienta. Poté spustí na instanci klienta vysílání dat v Synchronization - Start Streaming. Jestliže se počítače v síti vidí a uživatel nastavil správně IP adresu, měli by na instanci serveru nahoře vlevo ve viewportu být vidět text „Body received from client“. Pokud ne, uživatel by měl zkontrolovat, jestli se počítače vidí v síti, např. „pingnutím“ serveru a jestli dobře přepsal IP adresu.

Jak již bylo řečeno v kapitole 4.3, data z Kinectu od uživatele, který je k němu otočený zády, jsou pro synchronizaci nepoužitelná. Takže synchronizace zde je implementována tak, že se používají data z Kinectu, ke kterému je uživatel otočen čelem. To umožňuje uživateli se otočit a stále mít použitelný záznam animace. Limitací je zde sice několika snímkový přechod, kde ani jeden Kinect dobře nevidí na uživatele a animace bude pokroucená.

Kinect kromě těla dokáže rozeznávat i obličeje. Kinect v záběru je tedy určen snímáním obličeje uživatele. Kterému Kinectu se podaří rozpoznat obličej vícekrát za určitou dobu, jeho data se dostanou do animace. Implementováno je to tak, že data ze serveru a data z klientu mají frontu, kde se zapíše, jestli byl rozpoznán obličej. Fronta nemůže nabývat více, než osmi hodnot. Když přijde devátá hodnota do fronty, tak se uvolní první, a tak dále. Čili ve frontách je počet rozpoznání obličeje každé instance za 8 snímků. Která instance má víc, její data vytvoří klíčový snímek. Když mají stejně, je upřednostněn server.

Jen z popisu se dá vypozorovat, že se tohle dá jednoduše „rozbít“, tím že se stačí podívat na Kinect, který je zády. Uživatel si tedy musí být vědom tohohle omezení.

Jiný způsob rozlišení orientace uživatele by mohl být měřit úhel mezi např. vektorem rameno - loket a vektorem rameno - zápěstí. Když by byl úhel záporný, uživatel by musel mít zlomenou ruku aby se do takového úhlu dostal ve fyzickém světě, pak je uživatel otočen zády. Ale Kinect není k tomuto určený a data ze zad uživatele čte špatně a vrátí je „pokroucená“, kde tento úhel může být kladný, i když v reálném světě tento úhel nemůže být skoro nikdy záporný.

6.4 Použité knihovny

6.4.1 Helix Toolkit

Helix Toolkit [**6**] je kolekce 3D komponentů pro platformu .NET Framework. Knihovna je postavená na základech knihovny SharpDX [**7**], která „obaluje“ nativní kód DirectX a lze ji používat v jazyce C#. Helix Toolkit se dá také používat jako rozšíření grafických knihoven WPF. Avšak její hlavní zaměření je kombinace WPF nebo UWP s ShaprDX.

6.4.2 GRPC

GRPC [**11**] je knihovna zprostředkující síťovou komunikaci, ať místní nebo webovou. Funguje napříč mnoha platforem a jazyků.

6.4.3 Assimp

Open Asset Import Library, zkráceně Assimp [**9**], je přenosná open source knihovna pro importování mnoha formátů 3D modelů. Export není tak vyvinutý jako import, nicméně se pracuje na jeho vývoji. Existuje několik wrapperů, který nativní kód napsaný v C++ obalují pro snadné použití v ostatních jazycích. Mezi nimi je i Assimp.Net [**10**], který umožňuje psát implementaci s knihovnou v C#. Helix Toolkit začal používat Assimp.Net v jeho poslední verzi.

6.5 Podobné aplikace

6.5.1 Brekel

Brekel je pseudonym vývojáře, který má přes 15 let zkušeností s vývojem motion capture [**12**]. Vyvinul několik cenově dostupných aplikací, které podobně využívají obě verze Kinectu pro Xbox, jako snímač pro motion capture. Jeho nejrozšířenější produkt je program Pro Body v2, který by se dal popsat jako rozšířením aplikace předvedené v této práci. Stejně jako tady, používá knihovny z Kinect SDK, ale staví dále na dalších algoritmech pro zlepšení výsledku.

6.5.2 Kinect Mocap

Produkt Kinect Mocap firmy iClone [**13**]. Pokročilejší implementace motion capture s Kinectem jako snímač. Dokáže například rozeznat další klouby a pohyb po podlaze. Navíc má implementovanou fyziku, takže doplňky postavy a vlasy se můžou hýbat nezávisle na uživateli.

6.5.3 iPi iPi Recorder

iPi Recorder je program vyvinutý firmou iPi Soft [**14**]. Na rozdíl od předešlých produktů se specializují na synchronizaci více kamer pro motion capture. Ale zase produkty nepracují v reálném čase, je potřeba analyzovat data nahrávek v jiném programu. Umí používat i konkurenční produkt Kinectu od Sony, Playstation Eye.

Závěr

Cílem této diplomové práce bylo vytvořit aplikaci, která by dokázala animovat libovolný 3D model humanoida podle pohybu uživatele nasnímaným Kinectem od společnosti Microsoft. Což v podstatě znamená motion capture. Pro tyto účely jsme si představili speciální strukturu, tzv. kvaternion. To je uspořádaná čtveřice, která zobecňuje komplexní čísla do 3D prostoru. Kvaterniony se ve 3D grafice používají k vyjádření rotace a také k tomuto účelu byly tak definovány. Ukázali jsme si, že to je vlastně „jen“ zakódovaná osa otáčení a úhel. Výsledkem kapitoly bylo zobrazení, díky kterému můžeme rotovat vektor nebo bod v trojrozměrném prostoru podle daného kvaternionu.

Kvaterniony jsme si představili, protože animace postav je především o rotaci jednotlivých kostí v určitý okamžik v čase. Takže v teorii jsme navázali animací postav. Což je vlastně propagace transformací souřadnicových systému ve stromu závislostí daných kostí v určitém čase. Kde výsledná pozice kosti záleží na pozici všech jejich předchůdců. Čili když pohneme kostí v prostoru, tak se hýbou všichni její potomci, ale na její předchůdce nebude mít daný pohyb vliv. Potřebovali jsme teoretickou i praktickou znalost animace, protože cílem práce je vytvořit právě inverzní algoritmus k animování. Kde v animaci skládáme výslednou transformaci kostí v závislostech na svých předchůdcích, data z Kinectu představují výsledné umístění kosti v prostoru a tak je potřeba mezi nimi vytvořit hierarchii závislostí.

Právě tento postup jsem si popsali a ukázali, že je částečně závislý na použitém modelu. Tuto závislost je uživateli umožněno ji v aplikaci popsat. Samotná aplikace pak snímá pohyby uživatele pomocí Kinectu a převádí je do animace 3D modelu. Ať už uživatel nahraje do aplikace svůj vlastní model nebo použije jeden z přiložených. Výslednou animaci je pak možno nahrát a krokovat, upravit jednotlivé pozice animace a exportovat ji do souboru, i když ne do požadovaných formátů.

Při práci s aplikací si musí být uživatel vědom jejich omezení, které pramení z omezení Kinectu. Ten dokáže dobře snímat člověka čelem k sobě, k čemuž je taky kalibrován. Uživatel by se měl vyhnout otáčení se zády ke Kinectu, ten totiž nerozpozná rozdíl mezi pravou a levou a snaží se vyčíst ze snímku člověka natočeného ke Kinectu. Což má za následek výrazně horší kvalitu animace. Toto částečně odbourá synchronizace dvou Kinectů, kde při různé orientaci hlavy natáčí uživatele jiný Kinect. Kinect sice nedokáže rozpoznat orientaci uživatele, ale je schopen rozpoznat obličeje ve svém zorném poli. Tohoto faktu jsme využili při párování dvou Kinectů.

Na závěr jsme si představili tři aplikace, které tento koncept také implementovali a zdokonalili. Kosti se tolik netřepotají, přidali dodatečné kosti a postavili model na zem. Mohl by zde být důvod studovat motion capture a vylepšit aplikaci o aproximační algoritmy. Aplikace má tedy velký potenciál být vylepšena. Avšak tomu odporuje fakt, že se Kinecty již přestaly vyrábět a prodávat. Toť ironie života.

Conclusions

The goal of this thesis was to create an application, which could animate an arbitrary 3D model of a bipedal humanoid according to the movement of the user captured by the Kinect from Microsoft company. This is basically what motion capture means. For this purpose, we have introduced a special structure, a quaterion. Which is an ordered quadruplet that generalizes a complex number into 3D space. Quaternions are used in 3D graphics to represent rotation, they were defined for this purpose after all. We showed that a quaternion is "just" an encoded rotation axis with an angle. The result of the section was a map, with which we can rotate a vector or a point in three-dimensional space by the given quaternion.

We have introduced quaternions because character animation is mainly about rotation of individual bones in a certain moment in time. So we have followed up the quaternion theory with character animation theory. Which is applying transformations of coordinate systems to every child in a tree hierarchy of bones in certain time. The result orientation of a bone depends on an orientation of all its predecessors. Which means, if we move a bone in space, then all of its children move. But it won't have an effect on its predecessors. We needed theoretical and practical knowledge of animation, because goal of this thesis is to construct a reverse algorithm to animation. In animation, we compose the resulting transformation of the bones in relation to all of its predecessors, where data from Kinect represents the final position of the bone in space. And so it is needed to create a hirearchy between them.

We have described this and showed, that it is partialy related to chosen model. User can describe this relation in the application. The application itself scans motion of the user with Kinect and transforms it into animation of a 3D model. Whether the user uploads his own model or uses one of the enclosed models. It is possible to record the final animation and step through it, adjust individual positions of the animation and export it into a file, even though they're not the required ones.

While working with the application, the user needs to be familiar about its limitations, which comes from the limitations of the Kinect itself. It can capture a human quite well when he's facing the Kinect. It is calibrated for this task in the end. The user should avoid showing your back to the Kinect, because it can't recognize the difference between right and left when it tries to read a human facing the Kinect from a picture frame. Which results in poorer quality of the animation. This is partially solved by syncing two Kinects, where a different Kinect is capturing the user while the orientation of the user's head is changing. Kinect can't recognize an orientation of the user, but it is capable of recognizing faces in its field of view. We have used this fact to sync a pair of Kinects.

In the end, we have shown three different applications, which take this concept, implements it and perfect it. The bones don't flicker that much, additional bones were added and they put the model on the ground. This could be a reason

to further study motion capture and improve the application with approximation algorithms. Therefore, the application has a potential to be improved. But this counters the fact, that Kinects have been already discontinued. Just an irony of life.

A Obsah přiloženého CD/DVD

Na samotném konci textu práce je uveden stručný popis obsahu přiloženého CD/DVD, tj. jeho závazné adresářové struktury, důležitých souborů apod.

bin/
: Adresář obsahuje spouštěcí soubor aplikace.
Cesta k němu je bin\MotionCatcher\MotionCatcher.exe

doc/
: Adresář obsahuje dokumentaci jak v souboru pdf, tak její zdrojové soubory.

src/
: Adresář obsahuje zdrojový kód aplikace.

readme.txt
: Soubor readme.txt obsahuje postup instalací aplikace a podmínky pro její spuštění. Dále obsahuje odkazy na veškeré cizí, převzaté materiály.

Reference

[1] Frank D. Luna. Introduction to 3D Game Programming with DirectX 11. First. Dules (Virginia): Mercury Learning and Information, 2012. xxxviii, 864s. ISBN 978-1-9364202-2-3.

[2] Wikipedia. Motion Capture. [online] 2019. Dostupné z: https://en.wikipedia.org/wiki/Motion_capture

[3] IGN. A brief history of motion-capture in the movies. [online] 2014-06-11 [cit. 2019-01-13]. Dostupné z: https://www.ign.com/articles/2014/07/11/a-brief-history-of-motion-capture-in-the-movies

[4] Lolengine. Beautiful maths simplification: quaternion from two vectors. [online] 2014-01-06. Dostupné z: http://lolengine.net/blog/2013/09/18/beautiful-maths-quaternion-from-vectors

[5] Mixamo. Tvorba animací a 3D modelů. Dostupné z: https://www.mixamo.com

[6] Helix Toolkit. Helix Toolkit je kolekce 3D komponentů pro .NET framework. Dostupné z: https://github.com/helix-toolkit/helix-toolkit

[7] Alexandre Mutel, SharpDX. DirectX wrapper pro C#. [online] 2017-05-29. Dostupné z: http://sharpdx.org

[8] Microsoft. Požadavky na Kinect V2. [online] 2016-08-26. Dostupné z: https://social.msdn.microsoft.com/Forums/en-US/8cb681bd-bc9b-4533-a02b-990cc9f18d15/will-my-pc-work-the-kinect-v2?forum=kinectv2sdk

[9] Assimp Development Team. Open Asset Import Library. [online] 2018-01-16. Dostupné z: http://www.assimp.org

[10] Nicholas Woodfield. Assimp .NET. C#.NET Wrapper for the Open Asset Import Library (Assimp) [online] 2018-11-28. Dostupné z: https://bitbucket.org/Starnick/assimpnet/src/master

[11] GRPC. A high performance, open-source universal RPC framework. [online] 2018-01-16. Dostupné z: https://grpc.io

[12] Brekel. Afforable Motion Capture Tools. [online] 2016. Dostupné z: https://brekel.com

[13] iClone Kinect Mocap. Real-time Kinect Motion Capture. [online] 2019. Dostupné z: https://www.reallusion.com/iclone/mocap

[14] iPi Soft, iPi Mocap Studio a iPi Recorder. [online] 2018. Dostupné z: http://ipisoft.com/

[15] Wikipedie. Kinect. [online] 2019. Dostupné z: https://en.wikipedia.org/wiki/Kinect