Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského, Bratislava

Animácia postáv pomocou kostry

(Bakalárska práca)

Stanislav Fecko

Odbor: 9.2.1 Informatika
Vedúci: RNDr. Martin Samuelčík

Bratislava, 2010

Čestne prehlasujem že som túto prácu vypracoval sám za pomoci uvedenej literatúry, konzultácií s vedúcim práce a s ľuďmi spomenutými v poďakovaní.

Bratislava, 11.jún 2010 .....

Týmto sa chcem poďakovať vedúcemu práce pánovi RNDr. Martinovi Samuelčíkovi za odborné konzultácie. Tiež by som rád vyslovil vďaku rodičom a súrodencom za podporu, spolužiakom za rady a aj všetkým ktorí akokoľvek inak pomohli, alebo aspoň chceli.

Abstrakt

Autor:	Stanislav Fecko
Názov práce:	Animácia postáv pomocou kostry
Škola:	Univerzita Komenského v Bratislave
Fakulta:	Fakulta matematiky, fyziky a informatiky
Katedra:	Katedra informatiky
Vedúci práce:	RNDr. Martin Samuelčík
Miesto:	Bratislava
Počet strán:	29
Dátum:	11.6.2010

Abstrakt: Práca na začiatku stručne oboznamuje s možnosťami animácie, neskôr nasleduje detailnejší popis techniky využívajúcej na tento účel vnútornú kostru postavy. Tento prístup sa dnes často využíva na animáciu virtuálnych postáv. Vďaka využitiu potenciálu GPU na výpočet potrebných transformácií je technika použiteľná aj na animáciu v reálnom čase. V práci prezentujem svoju implementáciu metódy a riešenia problémov na ktoré som po ceste narazil.

Kľúčové slová: grafika v reálnom čase, OpenGL, animácia postáv, kostra

Abstract

Author:	Stanislav Fecko
Thesis title:	Skeleton-driven character animation
University:	Comenius University, Bratislava
Faculty:	Faculty of Mathematics, Physics and Informatics
Department:	Department of Computer Science
Advisor:	RNDr. Martin Samuelčík
Miesto:	Bratislava
Thesis length:	29
Date:	11.6.2010

Abstract: The thesis first briefly describes various techniques used for animation of digital objects. Then a method using skeleton is introduced and explained in detail. Nowadays, this approach is frequently used to animate virtual characters. Taking advantage of massive computing power of modern GPUs makes the technique suitable for animations in real time. Throughout the thesis I present my own implementation of the method as well as solutions of diverse problems encountered in the course of the work.

Keywords: real-time graphics, OpenGL, character animation, skeleton

Predhovor

Na animáciu virtuálnych postáv sa dá použiť niekoľko techník, ale najčastejšie je to prípad animovanej kostry, na základe ktorej sa následne hýbe celý model. Téma ma zaujala pre jej široké možnosti využitia v grafike. Táto metóda umožňuje s miernymi úpravami animovať nielen postavy, ale aj vegetáciu, či deformácie všeobecných objektov. Prezentujem tu implementáciu naplno využívajúcu výkon GPU. Tým sa táto techika dá využiť aj na animáciu postáv v reálnom čase.

Ako súčasť práce som naprogramoval aplikáciu schopnú načítať model a kostru a následne využijúc spomenutú metódu modelom hýbať podľa pózy kostry. Prínosom práce by malo byť zhrnutie potrebných matematických znalostí potrebných na naprogramovanie podobného systému, ako aj opis dátových štruktúr a algoritmov ktoré som využil.

Obsah

Zoznam použitých obrázkov

Úvod

Cieľom tejto práce je oboznámenie sa s možnosťami animácie trojrozmerných objektov, najmä virtuálnych humanoidov. Využíva sa pritom vnútorná kostra, ktorej pohyby sú buď definované sekvenciou kľúčových snímkov, alebo môže byť napríklad počítaná v reálnom čase na základe fyzikálnej simulácie. Animovaný objekt sa musí na kostru v základnej póze istým spôsobom naviazať a potom je schopný sledovať jej pohyby. Technika sa často využíva napríklad vo filmovom priemysle, pretože s využitím dostatočne detailnej kostry a jej kvalitnej animácie produkuje veľmi realistické deformácie ľubovoľnej podobnej postavy na ktorú sa aplikuje. Toto sa dá využiť okrem iného aj v rôznych masových scénach, kde by bolo využitie hercov v takomto počte nepraktické alebo neekonomické. Dnes už dokonca vznikajú aj celovečerné filmy hrané virtuálnymi hercami, a zároveň vo vysokej filmovej kvalite. Využitie shaderov grafickej karty na najnáročnejšiu časť výpočtu umožňuje tieto animácie počítať s miernymi obmedzeniami aj v reálnom čase, čo z metódy spravilo veľmi populárny nástroj aj pre tvorcov počítačových hier. Takéto animácie sa síce kvalitatívne nevyrovnajú filmovým, ale ich výhoda je v tom, že čas na renderovanie takejto postavy sa počíta v milisekundách.

V práci prezentujem riešenie využívajúce knižnicu OpenGL a deformácie pokožky počítam vo vertex shaderoch. Vďaka tomuto presunutiu výpočtov na GPU je aplikácia schopná počítať pohyby v reálnom čase. Využívam modely načítavané z formátu Truespace od firmy Caligari, kostru a jej animáciu uskladňujem vo vlastnom binárnom formáte. Tiež som na ilustrovanie niektorých častí textu použil okrem záberov zo svojho programu aj obrázky z filmu Avatar (avatarmovie.com), z hier Zaklínač (tw1.thewitcher.com; v poľskom origináli Wiedźmin) a World of Warcraft (worldofwarcraft.com) a zo stránky joshuadavidbrown.com. Použité modely sú jedny z hotových vzorov v programe TrueSpace. Na ich načítavanie som použil tutoriál [7].

1 Základné pojmy

Na začiatok by som uviedol zopár pojmov ktoré budem v práci používať a ku každému napíšem, čo pod ním chápem, aby sa tak snáď predišlo prípadným nedorozumeniam či nejasnosti textu. K niektorým uvádzam aj anglický ekvivalent pojmu. V texte sa budem snažiť používať slovenské termíny, ale v prípade záujmu o tému sa môžu hodiť aj anglické, keďže väčšina materiálov je v tomto jazyku.

VRCHOL (VERTEX) - bod v trojrozmernom priestore reprezentovaný vektorom, a to buď trojzložkovým (bežnejšie), alebo štvorzložkovým (rozšírený zápis; homogénny). Prevod medzi týmito zápismi je nasledovný: $[x, y, z, w] = [\frac{x}{w}, \frac{y}{w}, \frac{z}{w}]$

TROJUHOLNÍKOVÁ SIEŤ, TRIANGLE MESH; MESH, alebo jednoducho aj MODEL - množina trojuholníkov reprezentovaných VRCHOLMI (tormi). Najčastejšie sú tieto vrcholy zdieľané viacerými trojuholníkmi. V tejto sieti potom stranu každého trojuholníka nazývame HRANOU (EDGE) a trojuholník STENOU (FACE). Sieť sa môže skladať aj zo všeobecnejších mnohouholníkov, avšak ja budem v práci robiť len s trojuholníkovou sieťou.

KOSŤ (BONE) - základná stavebná jednotka kostry, má pri nej význam hovoriť o dĺžke a rotácii (voči predchádzajúcej kosti). Kostra je potom stromovitá štruktúra kostí, ktoré sú navzájom spájané KĹBMI (JOINT). Pre každú kosť existuje predchádzajúca (PARENT) kosť a prípadne tiež nejaký počet nasledovných. Jediná kosť bez predka sa nazýva KOREŇOM (ROOT) kostry.

ANIMÁCIA - proces postupného pohybu alebo zmien tvaru v závislosti na čase. Väčšinou ide o prípad, že poloha (príp. iný parameter) je definovaná v niekoľkých kľúčových časových okamihoch a následne v procese animácie je úloha dopočítať túto polohu aj pre body v čase, kde hodnota nie je explicitne definovaná (napríklad ak ju potrebujeme v čase 1,42s a je zadaná len v časoch 0s a 3s).

1.1 OpenGL

Keďže táto práca intenzívne využíva OpenGL a aj v texte sa mnohokrát spomína, bolo by asi dobré začať tým že čo a načo OpenGL je.
Už dávnejšie vznikali rôzne špecializované súčasti počítačov zamerané na podporu grafiky. Ich hlavnou úlohou bolo urýchľovať niektoré náročné grafické výpočty. Za týmto účelom boli výkonnejšie v istom type úkonov, avšak na rozdiel od centrálneho procesora výrazne menej flexibilné. Ich využitie teda zahŕňalo rozdelenie úloh na tie ktoré sa budú vykonávať na hlavnom procesore

a tie, ktorých sa ujme prídavné zariadenie. Avšak tieto zariadenia boli často natoľko unikátne, že pri vývoji programov bolo treba pre každé písať osobitné ovládače a rozhranie. Po niekoľkých pokusoch o akési zjednotenie v tejto oblasti a vytvorenie štandardu, ktorý by bol všeobecne dodržiavaný, vzniklo v roku 1992 konzorcium *OpenGL Architecture Review Board*, ktoré vytvorilo špecifikáciu OpenGL. V roku 2004 vznikla verzia 2.0, v roku 2008 verzia 3.0 a v roku 2010 verzia 4.0 tejto špecifikácie.

Teda OpenGL je rozhranie medzi grafickým hardvérom a aplikáciou, ktorá jeho služby využíva. Rozhraním je v tom zmysle, že program volá štandardom definované funkcie a očakáva tým istým štandardom definované správanie sa týchto funkcií. Je na každom výrobcovi grafickej karty, aby s ňou dodal ovládače implementujúce dané funkcie na konkrétnej karte. Ako sa dá všimnúť pri kupovaní grafickej karty, vždy sa uvádza verzia OpenGL, ktorú karta podporuje. To znamená, že hardvér by mal byť schopný vykonávať funkcie danej verzie špecifikácie a teda programy využívajúce tieto funkcie by ich mali mať k dispozícii. Mali... Stáva sa, že nie úplne všetko je realizované na karte, ale emulované softvérovo, čiže za využitia CPU.

Možno zrozumiteľnejšie bude funkciu tohto rozhrania popísať ako istého kvázi autonómneho agenta, ktorému z hlavného programu posielame stručné príkazy a on ich potom preloží hardvéru grafickej karty, ktorá ich vykoná. Vstupom pre OpenGL sú teda nejaké dáta a popis čo s nimi robiť. Najčastejšie sú týmito dátami body priestoru spolu s ich niekoľkými vlastnosťami (ako napríklad farba, alebo normálový vektor) a popis väčšionu špecifikuje, akým spôsobom sa zadané body majú skladať do trojuholníkov. V procesoroch grafickej karty sa na jednotlivé vrcholy aplikujú rôzne transformácie zodpovedné napríklad za správnu prespektívu a pozíciu kamery, až sa nakoniec 3D trojuholník premietne na obrazovku. Potom sa každý z trojuholníkov rozbije na jednotlivé pixely a počíta sa ich výsledná farba. Nakoniec sa táto farba zapíše do časti pamäte, ktorá sa ľahko predstaví ako výsledné plátno, na ktoré sa používateľ aplikácie pozerá.

Ako jednoduchý príklad by mohlo slúžiť nakreslenie obdĺžnika. Ten je definovaný štyrmi vrcholmi. Zadajú sa ako priestorové vektory, čiže každý ako trojica čísel. Keďže OpenGL pracuje s homogennými súradnicami bodov, tieto čísla sú doplnené štvrtou zložkou: jednotkou. Následne je každý z vrcholov vynásobený afinnou maticou zodpovednou za správne premietnutie z 3D do 2D. Potom sa musí určiť, ako sa tieto štyri vrcholy poskladajú do trojuholníkov. Na základe informácie, že chcem kresliť obdĺžnik, vieme z nich utvoriť dve trojice: 123 a 341 (kde číslo označuje poradové číslo v akom bol daný vrchol zadaný). Pre každý z týchto dvoch trojuholníkov si teraz zistíme, ktoré pixely na obrazovke zaberá a pre každý z nich budeme počítať jeho farbu.

Na dosiahnutie rôznych výsledkov, vhodnejšieho správania alebo efektov, môže programátor rôzne časti tohoto procesu meniť. Niektoré viac, niektoré menej. No a tento projekt sa do značnej miery zaoberá úpravou jedného z preddefinovaných štádií transformovania vrcholov tak,

aby sa v konečnom dôsledku virtuálna postavička hýbala. Na toto využijeme takzvané SHADERY. Ide vlastne o krátke a relatívne jednoduché programy, ktorými sa dajú nahradiť pôvodné štádiá spracovania vstupných dát. V súčasnosti rozlišujeme tri typy shaderov - geometrické, vrcholové a pixelové (geometry, vertex and pixel shaders). Kým prvé z nich menia už najskoršiu fázu, a to zadávanie vrcholov (majú možnosť generovať nové vrcholy a posielať ich ďalej na spracovanie), druhé upravujú transformáciu vrcholov (na vstupe majú 3D vrchol a na výstupe obvykle 2D). Tretia skupina shaderov sa zase stará o výpočty v poslednej fáze, a to keď sú už trojuholníky rozdelené na jednotlivé pixely. Jedno z možných využití je výpočet jasu pixela na základe informácie o pozíciách svetiel a orientácii tieňovaného povrchu.

Geometrické shadery sa zatiaľ využívajú len zriedkavo, okrem iného aj preto že ich podporujú len najnovšie grafické karty. Ale dva zvyšné druhy shaderov sa v praxi využívajú veľmi často a ich vhodnou kombináciu sa dosahujú všemožné efekty. Aby sa tieto časti dali programovať, je potrebné poslať do grafickej karty krátky program, ktorý tam bude následne bežať. Program sa posiela ako text, ovládač ho preloží pre konkrétny hardvér a nahrá na správne miesto. Staršie verzie shaderov vyžadovali tento program v podstate v asembleri. Odhliadnuc od toho že takýto program je citeľne menej prehľadný (so všetkými nevýhodami ktoré sú s tým spojené), má aj inú nevýhodu. Mohlo by sa zdať, že program v asembleri má potenciál byť vysoko efektívny, a teda relatívne rýchly. Avšak nie je karta ako karta a to už nehovoriac o kartách od rôznych výrobcov. Preto kód, ktorý je na jednej karte optimálny, môže byť pre druhú nevhodný. Novší prístup k shaderom preto spočíva v tom, že OpenGL neočakáva asemblerový program, ale program napísaný vo vyššom jazyku. Tento je driverom špecifickým pre konkrétnu grafickú kartu preložený do jednotlivých inštrukcií a až tak posunutý ďalej. Toto medzištádium umožňuje výrobcom kariet napísať na každý model karty osobitný prekladač. Taký, ktorý bude využívať všetky možnosti daného hardvéru a bude generovať programy optimálne pre ten ktorý konkrétny typ karty.

Tým sme získali dve výhody naraz - možnosť písať shadery nejakým rozumným spôsobom a zároveň aj relatívne vysokú optimalizáciu tohoto kódu. Spomínaný vyšší jazyk v ktorom sa programy píšu sa nazýva **OpenGL Shading Language**, známejší pod skratkou **GLSL**. Ide o jazyk veľmi podobný jazyku C, má rovnakú syntax a podobné dátové typy. Je rozšírený o vektory a matice, rôzne operácie s nimi a navyše definuje niekoľko špeciálnych typov premenných.

1.2 GLSL jazyk

Každý shader je samostatný program, ktorý má podľa typu (či pracuje na úrovni vrcholov alebo pixelov) definované isté vstupy a výstupy. Oveľa viac, detailnejšie a presnejšie sa dá o GLSL dočítať napríklad v oficiálnej špecifikácii tohoto jazyka [3]. Tu iba upozorním na jednu (dôležitú) zaujímavosť ktorá mierne vychádza nad rámec jazyka C.

Ide o dva druhy premenných - **uniform** a **varying**. Nebudem to radšej prekladať, lebo aj tak sa iné názvy nepoužívajú. Čo o nich stačí vedieť je to, že slúžia na komunikáciu medzi rôznymi shadermi. Keď využívame vertex a pixel shadery, vo väčšine prípadov potrebujeme na výpočet farby pixela aj nejeké vstupné informácie. Jednými z nich sú uniform premenné, ktoré slúžia na zásahy priamo z hlavného programu dovnútra shadera. To preto, že hodnoty týchto premenných môžeme meniť priamo z aplikácie. Ako príklad takéhoto využitia môže byť napríklad súčasná pozícia kamery. Tá sa pri každom vykreslení scény mierne mení a zakaždým upovedomíme shader o jej aktuálnej polohe. Využívame na to špeciálnu funkciu glUniform popísanú okrem iného v špecifikácii OpenGL. Druhá kategória sú varying premenné, ktoré zasa slúžia na prenos informácií z vertex shadera do pixel shadera. Najbežnejším parametrom ktorý chceme vedieť pre každý pixel, je normálový vektor plochy v tom danom bode. Tu si treba uvedomiť, že aj keď máme zadaný trojuholník s informáciou o normálach v každom vrchole, budeme potrebovať zistiť tento vektor aj pre pixel v strede trojuholníka. Preto sa tu využíva bilineárna interpolácia hodnôt v rohoch trojuholníka. Týmto parametrom môže tiež byť napríklad farba. Vďaka interpolácii jej hodnôt po ploche trojuholníka dostaneme pekný plynulý prechod. Preto napríklad farbu definujeme ako varying premennú.

Nemyslím si, že má význam ísť do detailov. V prípade záujmu odporúčam nedozierne more tutoriálov na internete, ako aj spomínanú oficiálnu špecifikáciu. Nakoniec už len doplním, že jedna z dôležitých častí tejto bakalárskej práce spočíva vo vertex shaderi, kde budeme počítať deformácie pokožky pre každý vrchol modelu. Z týchto vrcholov sa následne vyskladajú trojuholníky a v konečnom dôsledku sa bude postava hýbať.

1.3 Matematické základy

Po absolvovaní povinného predmetu Algebra(1) by už malo stačiť doplniť pár detailov. Napríklad sa oboznámiť s pojmom kvaternión, ktorý bude v práci reprezentovať rotácie.

1.3.1 Komplexné čísla a rotácie v 2D

Skôr než budeme detailnejšie rozprávať o kvaterniónoch, spomenul by som, prečo vôbec siaham po takýchto číslach. Je to preto, že ak sa na komplexné čísla a kvaternióny pozrieme zo správneho

uhla, zistíme že nám vedia celkom pekne vyjadrovať rotácie. Začnime teda tými jednoduchšími: komplexnými číslami.

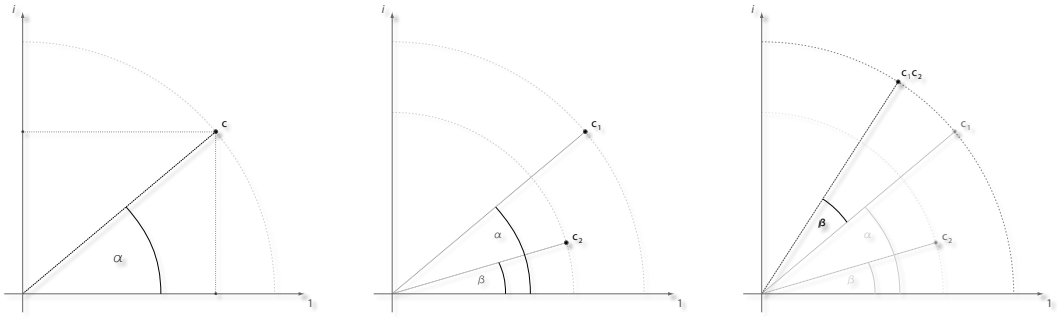
Z vlastnosti imaginárnej jednotky $i^2 = -1$ vyplýva, že ak vynásobíme dve komplexné čísla, dostaneme

$$(a + bi)(c + di) = (ac - bd) + i(ad + bc).$$

Čo je na ich násobení zaujímavé však zistíme, až keď si číslo $c = a + bi$ zapíšeme v polárnej sústave súradníc (pri komplexných číslach tiež známa ako geometrická reprezentácia) ako $c = r(\cos\alpha + i\sin\alpha)$, kde r je dĺžka vektora a α je uhol zvieraný s reálnou osou. Vtedy uvidíme, že vynásobením dvoch čísel c_1 a c_2 dostávame

$$(r_1(\cos\alpha + i\sin\alpha))\,(r_2(\cos\beta + i\sin\beta)) \;=\; r_1r_2(\cos(\alpha+\beta) + i\sin(\alpha+\beta)).$$

V argumentoch oboch goniometrických funkcií výsledku dostávame ten istý uhol, a to $\alpha + \beta$. Dĺžka výsledného čísla je síce súčin r_1r_2, ale ak sa dohodneme, že $r_2 = 1$, tak sme dostali prekvapujúco pekný výsledok: komplexné číslo rovnakej dĺžky ako c_1 (čiže r_1), ibaže otočené o β okolo stredu. Inak povedané, násobením čísla c_1 číslom c_2 sme dostali c_1 zrotované o uhol β.



Obr. 1: Zľava vizualizácia komplexného čísla v Gaussovej rovine, čísla c_1 a c_2, a ich súčin (za predpokladu $|c_2| = 1$).

Každé komplexné číslo c sa dá chápať ako dvojrozmerný vektor v komplexnej rovine (jej bázu nám budú tvoriť reálna jednotka 1 a imaginárna jednotka i), často tiež nazývanej Gaussovou rovinou. V takomto ponímaní sa už dá spomenuté násobenie chápať úplne ako rotácia vektora. Teda ak máme číslo c reprezentujúce vektor v, tak na jeho rotovanie o uhol φ okolo stredu ho stačí vynásobiť číslom $(\cos\varphi + i\sin\varphi)$.

1.3.2 Kvaternióny a Rotácie v 3D

Kvaternióny sú čísla v istom zmysle dosť podobné na komplexné čísla. To preto, lebo kým komplexné číslo má jednu imaginárnu časť $(a + bi)$, kvaternión ich má tri $(a + bi + cj + dk)$. Aj v ostatných vlastnostiach sú si navzájom značne podobné. Napríklad, ako sme spomínali, imaginárna jednotka i má vlastnosť že $i^2 = -1$. Nuž, v prípade kvaterniónov platí obdobný vzťah: $i^2 = j^2 = k^2 = -1$. Tentoraz však existujú aj nasledovné rovnosti:

$$\begin{aligned} ij &= k \qquad & ji &= -k \\ jk &= i \qquad & kj &= -i \\ ki &= j \qquad & ik &= -j \end{aligned} \tag{1}$$

V práci budem používať vektorovú reprezentáciu kvaterniónov, a teda kvaternión $a+bi+cj+dk$ zapíšem ako vektor (a, b, c, d). Pri takomto spôsobe ich definovania je asi ešte dobré spomenúť, že každý kvaternión sa často delí na významovo osobitné časti - na skalárnu s a vektorovú $\overrightarrow{\mathbf{v}}$. Skalárnou označujeme hodnotu a, kým za vektorovú považujeme vektor (b, c, d). Kvaternión sa teda potom dá zapísať aj ako

$$a + bi + cj + dk \quad = \quad \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} \quad = \quad \begin{pmatrix} s \\ \overrightarrow{\mathbf{v}} \end{pmatrix},$$

a čo je pekné, reprezentuje nám práve rotáciu o uhol $2\arccos(s)$ okolo vektora $\overrightarrow{\mathbf{v}}$. Ďalej sa budeme zaoberať len kvaterniónmi, ktorých dĺžka v bežnom vektorovom ponímaní je 1 (*normalizované*). V opačnom prípade transformácia ktorú kódujú nebude zachovávať vzdialenosti, ale bude zväčšovať, resp. zmenšovať transformované objekty.

Počas práce budeme dosť intenzívne tieto kvaternióny medzi sebou násobiť. Ak si zapíšeme dva kvaternióny $u_0 + u_1 i + u_2 j + u_3 k$ a $v_0 + v_1 i + v_2 j + v_3 k$, roznásobíme ich ako každý iný mnohočlen mnohočlenom, tak po využití vlastností (1) dostaneme takýto vzorec pre súčin dvoch kvaterniónov:

$$\begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} u_0v_0 - u_1v_1 - u_2v_2 - u_3v_3 \\ u_0v_1 + u_1v_0 + u_2v_3 - u_3v_2 \\ u_0v_2 + u_2v_0 + u_3v_1 - u_1v_3 \\ u_0v_3 + u_3v_0 + u_1v_2 - u_2v_1 \end{pmatrix}, \tag{2}$$

alebo tiež

$$\begin{pmatrix} s_1 \\ \overrightarrow{\mathbf{v}_1} \end{pmatrix} \begin{pmatrix} s_2 \\ \overrightarrow{\mathbf{v}_2} \end{pmatrix} = \begin{pmatrix} s_1 s_2 - \overrightarrow{\mathbf{v}_1} \cdot \overrightarrow{\mathbf{v}_2} \\ s_1\overrightarrow{\mathbf{v}_2} + s_2\overrightarrow{\mathbf{v}_1} + \overrightarrow{\mathbf{v}_1} \times \overrightarrow{\mathbf{v}_2} \end{pmatrix}. \tag{3}$$

Ďalšia vec, ktorú budeme pri výpočtoch potrebovať, je pojem združeného kvaterniónu. Označujeme ho hviezdičkou. Ak máme kvaternión $q = (s, \overrightarrow{\mathbf{v}})$, tak k nemu združený kvaternión q^* bude

$$q^* = \begin{pmatrix} s \\ -\overrightarrow{\mathbf{v}} \end{pmatrix}$$

Teraz by sme už mali vedieť všetko nevyhnutne potrebné k samotnému rotovaniu vektorov pomocou kvaterniónov. Ak teda potrebujem otočiť vektor $\overrightarrow{\mathbf{v}}$ kvaterniónom q, robím to tak, že $\overrightarrow{\mathbf{v}}$ rozšírim o štvrtú zložku, ktorú nastavím na 0. Potom nasleduje násobenie kvaterniónov:

$$w = q \ u \ q^*. \tag{4}$$

No a nakoniec treba z kvaterniónu w spätne získať transformovaný vektor. Tým bude vektorová časť kvaterniónu w.

Viac o kvaterniónoch sa dá nájsť v [1]. V texte je na prakticky piatich stranách zhrnutého veľmi veľa o kvaterniónoch a ich použití. Ja som z tohto zdroja dosť intenzívne čerpal.

2 Animácia postavy

2.1 Motivácia

Už dávnejšie sa v počítačových hrách objavili okrem kariet či padajúcich kociek aj postavy. Avšak ukazuje sa, že ľudské oko je na niektoré nedokonalosti oveľa háklivejšie ako na iné. A pohyby tela sú jednou z tých problematických vecí, kde si všimneme aj pomerne malé chybičky. A tak spočiatku tieto postavy v rôznych hrách boli nahrádzané prácne predspracovávanými dvojrozmernými obrázkami a z hľadiska realizmu boli pochopiteľne na veľmi nízkej úrovni.

Časom sa výpočtový výkon bežných počítačov výrazne zvýšil a s možnosťou hardvérového urýchľovania na špecializovaných grafických koprocesoroch sa už dalo uvažovať o vdýchnutí tretej dimenzie do plochého sveta hier. Opäť, pokrok bol postupný, od jednoduchých hranatých postáv až po dnešné, od reality niekedy ťažko rozoznateľné modely. Pohyby týchlo ľudí tak isto prechádzajú zmenami, od trhaných a nie veľmi presvedčivých pohybov súperov v starších strieľačkách, až po do detailov vypracované animácie, ktoré sú realizmom porovnateľné s filmovou kvalitou.

Dnes už možno vidieť virtuálne postavy naozaj na mnohých miestach, od počítačových hier, cez filmy, kde často nahrádzajú živých kaskadérov, alebo dokonca dostanú i hlavnú úlohu, až po automatických sprievodcov online múzeami a galériami, vidíme ich každý deň v reklamných šotoch... Na dosiahnutie všetkého tohoto bolo treba vyvinúť metódy ako postavy rozhýbať a doslova im tak vdýchnuť život.

Obr. 2: Najčastejšie využitie virtuálnych animovaných postáv: vo filmoch (Avatar) a v hrách (Zaklínač).

2.2 Kostra

Pochopiteľne, animácia pomocou kostry nie je jediný spôsob ako dosiahnuť to, čo chceme. Napríklad jednoduchšie si je v pravidelných časových intervaloch zapamätať celú trojuholníkovú sieť postavy. Následne keď chceme takýto pohyb renderovať, stačí nám vykresliť model časovo najbližšie k súčasnému stavu. Ako som spomenul, ide o naozaj veľmi jednoduchý spôsob, čo so sebou, žiaľ, prináša aj viacero nepriaznivých vlastností tohoto prístupu. Závažný tu je problém, že postava sa takto nemôže hýbať plynulo, ale vždy iba skokovito. Ďalšia nemenej nepríjemná skutočnosť je, že takáto reprezentácia animácie vyžaduje masívne predspracovanie dát a už neumožňuje nijako animáciu meniť počas behu programu. No a v neposlednom rade sú nároky na uskladnenie takejto animácie nepríjemne vysoké.

Netreba sa nechať pomýliť faktom, že dnešné disky sú už často terabajtových veľkostí a domnievať sa že takéto šetrenie miestom je preto prežitok. Áno, je pravda že disky sú veľké, ale je hneď niekoľko dôvodov, prečo to nerieši situáciu. Po prvé, postava môže mať aj desaťtisíce trojuholníkov, takže jej geometrická reprezentácia môže zaberať aj desiatky až stovky kilobajtov. To nie je veľa. Ale pamätať si takýto objem dát povedzme 25 krát za sekundu už zvyšuje veľkosť na niekoľko mega za sekundu animácie jednej konkrétnej postavy. Po druhé, na jednej animácii nemusí byť rozdiel osudne veľký, ale keď je animácií sto alebo tisíc, tak už aj celkový objem bude dostatočne povšimnuteľný. A to vzhľadom na fakt, že každá postavička potrebuje osobitnú animáciu, nie je nijako prehnaný počet. Takže sa už dostávame ku gigabajtom a to máme ešte len geometriu, nehovoriac o textúrach či zvukoch. Po tretie, aby sa takáto postava renderovala, najprv treba daný model z disku prečítať a následne poslať do grafickej karty, čo v prípade že tade budeme tlačiť našich niekoľko mega za sekundu za každú postavu v scéne, zrazí rýchlosť renderovania na neúnosne nízke hodnoty. Stručnejšie povedané, tento prístup je veľmi neefektívny.

Lepší spôsob ako počítaču popísať animáciu je vytvoriť sériu takzvaných kľúčových snímkov. Potom to funguje tak, že pre každý vrchol modelu animácia obsahuje záznam o jeho polohe v daných kľúčových časoch a následne sa medzi týmito polohami interpoluje. Takýto prístup je vizuálne výrazne uspokojivejší, pretože pohyb už nie je trhaný, ale plynulý. Animácia tiež zaberie výrazne menej miesta, pretože kľúčové snímky, ako už aj slovíčko *kľúčový* navráva, sú umiestnené len v časových bodoch ktoré sú niečim zaujímavé, význačné, čiže kľúčové. Napríklad na dostatočne rovný pohyb stačí pomerne málo kľúčových snímkov bez viditeľnej straty kvality animácie a pritom sa tým ušetrí značná časť dát.

Tento prístup má podobnú značnú nevýhodu oproti animácii kostrou, a to že pohyb je fixovaný na jeden konkrétny model. Ak teda máme v scéne desať rôznych postáv, pre každú budeme musieť vytvoriť osobitnú animáciu, aj keď pohyb ktorý vykonávajú je prakticky rovnaký. Ďalším

nedostatkom je to, že meniť animáciu na základe spätnej väzby od hráča je prakticky nemožné. Napriek tomu je táto metóda pomerne vhodná na animáciu geometricky jednoduchých objektov, ktoré vykonávajú nejaké pomerne jednoduché pohyby a kde preto nemožnosť ovplyvňovať priebeh animácie nie je nevýhodou. Napríklad na animované trojrozmerné užívateľské prostredie je táto metóda dosť vhodná.

Obr. 3: Mnoho jednoduchých animácií ktoré nevyžadujú interakciu je realizovaných key-frame spôsobom. Napríklad predpokladám že aj tieto zástavy (World of Warcraft).

Nakoniec sme sa dostali až po metódu animovania na základe vnútornej štruktúry, ktorú po vzore z prírody nazvime kostrou. Hlavnou hnacou silou je možnosť vykonávať niektoré výpočtovo náročné operácie na trojuholníkovej sieti priamo na grafickej karte, kde sú na tento účel k dispozícii tisíce paralelných procesorov. Aby to nevyzeralo zase príliš ružovo, treba upozorniť na nevýhodu tohoto postupu, a to že tieto výpočnty sú často limitované na prácu s veľmi okliešteným množstvom vstupných dát. Napríklad kým pri práci na CPU môžeme ľubovoľne pristupovať ku všetkým vrcholom modelu, vo vertex shaderi, ktorý na tieto výpočty používame, máme prístup iba ku jednému bodu a musíme sa zaobísť bez informácie o ostatných. K dispozícii máme len zhora ohraničené množstvo pamäte, ktorú môžeme využiť na rôzne vstupné dáta. A práve preto použijeme kostru.

Najdôležitejšia vec, ktorú si tu treba uvedomiť je to, že kostra má bežne o zhruba dva až tri rády menej kostí ako model vrcholov. Má teda dostatočne malý dátový obsah na to, aby sa dala celá poslať ako vstupná informácia pre shader, ktorý bude animáciu počítať. A tak si nebudeme pamätať polohy v kľúčových časoch pre všetky vrcholy, ale len pre kosti a následne na základe nejakého výpočtu transformujeme celý model podľa polohy kostry. Na mieste by

teraz bola otázka či sto až tisícnásobným zredukovaním dát nestrácame príliš veľa informácie. Najdiplomatickejšia odpoveď je že závisí... Samozrejme, pomocou kostry sa nedá reprezentovať ľubovoľná animácia. A to práve kvôli najväčšej výhode tejto metódy, čiže akémusi zaškatuľkovaniu vrcholov do menších skupiniek, pričom v rámci jednej skupiny sa k nim správame takmer rovnako. Tým sa do istej miery stráca jedinečnosť každého bodu modelu za cenu výrazného urýchlenia výpočtu. Takže napríklad zmeny ktoré menia proporcie častí modelu sa tu reprezentujú veľmi zložito, alebo sa dosiahnuť vôbec nedajú. Predošlým spôsobom sa popísať dajú, ale keďže nás zaujíma pohyb ľudí, prípadne iných živočíchov, tieto problémy riešiť nemusíme, naopak, prístup nám zabezpečí, že vrcholy ramena budú pekne držať pokope, aj keď sa budú pohybovať. Preto na animácie aké plánujeme robiť je spôsob cez kostru vynikajúci.

Nuž a prejdime teda ku výhodám takéhoto spôsobu reprezentácie a počítania animácie. V prvom rade je veľmi rýchly. Je to preto, že hrozivo veľa výpočtov sme presunuli na GPU, ktorá je na to stavaná a oveľa výkonnejšia ako hlavný procesor.
Ďalšia výhoda je, že animácia sa dá jednoducho ovplyvňovať priamo za behu programu. Kým meniť polohy všetkým bodom modelu je výpočtovo nepríjemne náročné, pozmeniť zopár kostičiek je jednoúché a rýchle. Takže napríklad simulácia fyziky pri pohybe postavy sa dá bez väčších komplikácií počítať v reálnom čase. No a snáď najväčšia výhoda je neuveriteľná flexibilita čo do rôznorodosti animovaných postáv. Ak vytvoríme jednu animáciu ľudskej postavy, uložíme si ju ako sekvencie pohybujúcej sa kostry, môžeme neskôr túto kostru aplikovať na mnoho iných modelov postáv a animácia na ne bude použiteľná a bude vyzerať veľmi prirodzene. Ak máme napríklad animáciu jednej tancujúcej postavy, nie je problém túto animáciu použiť na rozhýbanie stoviek tanečníc a tanečníkov otváracieho ceremoniálu na olympiáde.

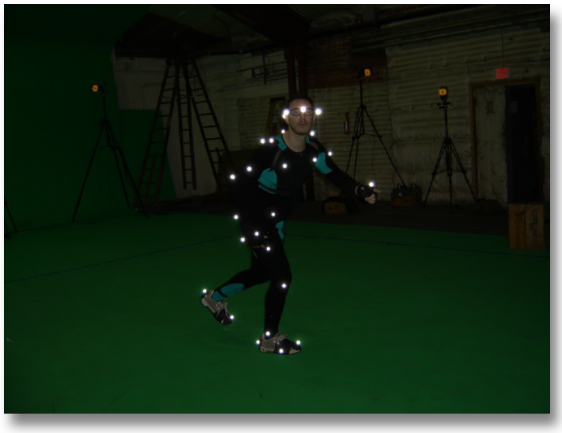
2.3 Zadávanie hodnôt kĺbov

Kostra umožňuje hýbať virtuálnou postavou. Ale aby tento pohyb bol čo najpresvedčivejší a najuveriteľnejší, treba aby také boli aj pohyby kostry. Teda aby bola dodržaná istá dynamika ľudského tela, alebo vzájomná súvislosť pohybov. Napríklad pohyby rúk sú často doprevádzané, hoci i miernymi, pohybmi zvyšných častí tela. Alebo povedzme pri obyčajnej chôdzi takmer niet kosti ktorá by mala ostať v pokoji. Editovanie takto realistických pohybov manuálne je prakticky nemožné, a to nielen preto, že by bol proces nesmierne časovo náročný. Hovoríme tu totiž o detailoch, ktoré zapísať do jazyka rotácií kostí nie je práca pre človeka.

Na dosiahnutie kvalitných animácií sa používajú rôzne techniky. Spomenul by som dve najznámejšie a zhruba popísal, v čom spočívajú. Jednoduchšou na realizáciu a ovládanie je definovanie animácií pomocou takzvanej inverznej kinematiky. V takomto prípade človek, ktorý animáciu vytvára, neriadi osobitne každý kĺb a nenastavuje jeho rotáciu, ale nastavuje iba polohy

a orientácie koncových bodov. Čiže napríklad ruky. Kúzlo postupu je v tom, že ostatné dáta sa dopočítajú na základe pomocných dát. Sú nimi uhlové ohraničenia jednotlivých kĺbov. Napríklad koleno sa dá pomerne ľahko ohnúť len jedným smerom. Je to dané tvarom kostí a ich vzájomným spojom. Aby sme výpočet nezaťažovali týmito údajmi, zadáme iba isté hraničné uhly, po ktoré je daný kĺb schopný sa otočiť. A zvyšné je už práca počítača. On nájde čo najprirodzenejšiu konfiguráciu predošlých kostí tak, aby bola nakoniec ruka tam kde ju chceme mať. Ako vidno, sú tu potrebné dáta, ktoré síce tiež treba zadať, ale sú pre človeka stále tie isté. Takže ich stačí zadať raz a použijú sa na prepočet všetkých potrebných animácií. No a nakoniec sa vypočítané pozície kostry opäť zapíšu ako kľúčové snímky a pri animácii sa interpolujú.

Ďalší spôsob ako zadať animáciu je systém *motion capture*. Je oveľa náročnejší ako vyššie spomenutý prístup, ale produkuje najkvalitnejšie výsledky. Jeho základná myšlienka spočíva v tom, že kostru v jednotlivých časoch nezadáva animátor pomocou myši a klávesnice, ale živý herec svojimi vlastnými pohybmi. A teda výsledná animácia sú presne nasnímané pohyby reálneho človeka, čo by malo zaručiť vysoký realizmus. Realizácia sa môže líšiť, ale najčastejšie ide o biele bodky na tele herca v čiernom, prípadne inak označené body na hercovom povrchu. Tieto body sa počas snímanej animácie zachytávajú niekoľkými kamerami dookola scény. Nasnímané sekvencie sú prehnané počítačom, ten nájde 3D polohy bodov a na ich základe je schopný zostaviť súčasnú polohu kostry. No a z tej už vie vypočítať potrebné rotácie. Tento spôsob, ako som spomínal, má potenciál na neuveriteľne kvalitné a dynamické animácie, ale na jeho realizáciu treba pomerne drahé vybavenie. Preto sa používa iba v profesionálnych štúdiách, ako napríklad pri tvorbe filmov či počítačových hier.



Obr. 4: Motion capture: herec s bielimi bodmi rozmiestnenými po tele. Mierne upravený obrázok pochádza zo stránky http://www.joshuadavidbrown.com/home.html

3 Popis implementácie

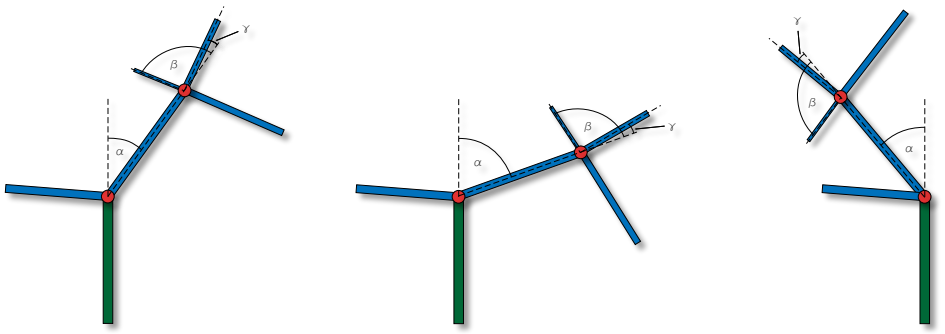
3.1 Dátové štruktúry

Snáď by stálo za reč v krátkosti popísať základné dátové štruktúry, ktoré som pri svojej implementácii použil. Nuž, začnime od základných tehličiek. Vzhľadom na to že OpenGL, na ktorom je celý projekt postavený, používa 32-bitové desatinné čísla, používam aj ja na výpočty typ GLfloat (**1**bit znamienko, **8**bit exponent, **23**bit mantisa). Z nich sa následne hierarchicky skladajú tamer všetky štruktúry, začínajúc vektormi, maticami, kvaterniónmi, ďalej z týchto postupne poskladané väčšie a väčšie štruktúry. Vo všeobecnosti sa snažím v prípade štruktúr na nižšej úrovni vyhýbať objektom a radšej dáta držím v poliach, avšak pre štruktúry hierarchicky dosť vysoko rád využijem pohodlie objektového prístupu.

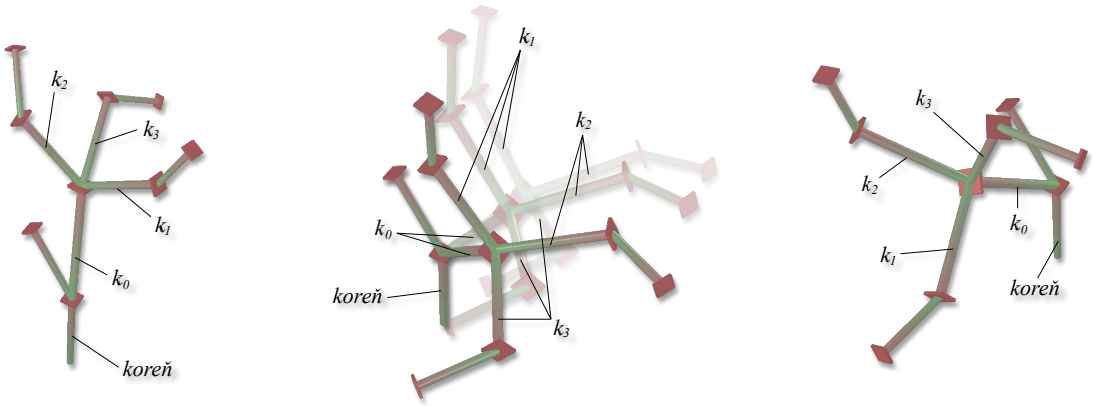
Rotácie vo svojom projekte reprezentujem kvaterniónmi, čo má niekoľko nezanedbateľných výhod. Ako prvá, a dosť dôležitá, je že zaberajú citeľne menej pamäte ako ekvivalentná rotačná matica. Už len týmto ušetrím 5 floatových čísel, čo vzhľadom na fakt, že tieto dáta musím zakaždým uploadovať do vertex shadera, je príjemné ušetrenie. Pri rôznych výpočtoch môže prípadne dochádzať ku hromadeniu chýb spôsobených konečnou presnosťou reprezentácie desatinných čísel v počítači. Toto však môže mať v prípade transformácie za následok že nebude iba rotovať, ale aj naťahovať alebo skracovať kosti. A ak sa takáto chyba vyskytne blízko koreňa, tak na okrajoch už môže byť pozorovateľná. Nuž a práve tu je výhoda kvaterniónovej reprezentácie rotácií - tieto stačí veľmi jednoducho normalizovať a aj keď nepresnosť rotácie sa tým zrejme neopraví, predídeme (určite bolestivému) naťahovaniu kostí. V prípade matíc by sme museli podstupovať pomerne komplikovaný proces ortogonalizácie, v porovnaní s čím je normalizácia smiešne jednoduchý a rýchly úkon.

Popíšme si teraz kostru. Pri nej je rozhodujúca množina kostí. O každej si pamätáme zopár jej vlastností a údajov o príslušnosti do hierarchie. Najdôležitejšie sú dĺžka, rotácia voči predošlej kosti, index predchodcu a indexy potomkov. Tak isto sa tu ale uskladňujú aj informácie o absolútnej priestorovej polohe kosti (dva vektory, začiatok a koniec), orientácii kosti voči defaultnému smerovaniu (to je smerom nahor [0,1,0]), ale aj krátky názov kosti. Posledný parameter pochopiteľne vôbec na nič nevplýva, ale v situácii, keď je kostra dostatočne košatá, môže správne pomenovanie pomôcť pri orientácii.

Kostra je primárne definovaná hierarchiou, dĺžkami kostí a relatívnymi rotáciami. Čiže vzťahmi otec-syn medzi jednotlivými kosťami, dĺžkou a rotácii voči predchádzajúcej kosti. Toto sa najľahšie predstaví v 2D, kde rotácia znamená jediný jednoduchý uhol. V práci sa však treba vysporiadať s uhlami v priestore. Preto na ich opis nestačí jedno číslo, ako v 2D, ale použijeme už spomenuté a opísané kvaternióny.



Obr. 5: Reprezentácia kostry - relatívne uhly v 2D. Podstatné je že sa mení len uhol α, kým uhly β a γ ostávajú nezmenené.



Obr. 6: Reprezentácia kostry - relatívne uhly v 3D. Opäť si všimnime že sa mení len uhol medzi koreňom a k_0, ale uhly medzi ostatnými kosťami nemeníme.

Ďalšia dôležitá štruktúra bez ktorej by sme nemali čo animovať, je samotný model. Reprezentovaný je už objektom, pozostáva z niekoľkých polí, z ktorých najdôležitejšie je pole 3D vektorov. Toto nesie informácie o pozíciách vrcholov. V osobitnom poli sú dáta popisujúce ktorý vrchol je súčasťou ktorého trojuholníka. Pamätám si tam aj iné dáta, ktoré sa využívajú pri renderovaní postavy, ako napríklad normály, ale z hľadiska našej animácie je ešte dôležité spomenúť niekoľko ďalších. V jednom poli si pamätám indexy a váhy kostí, na ktoré sa daný vrchol viaže a v ďalších dvoch pretransformované pozície vrcholov a normál. Ide o to, že pri procese naväzovania sa modelu na kostru sa pre každý vrchol nájde niekoľko najvplyvnejších kostí a následne sa

poloha tohoto vrcholu uloží z pohľadu daných kostí. No a to spomínané niekoľko je v mojom prípade 4. Teda každý vrchol modelu je v každom čase ovplyvňovaný najviac štyrmi kosťami. No a preto ‚najviac' a nie ‚práve', lebo ak sú niektoré z váh nulové, tak je bod prakticky transformovaný menším počtom kostí.

Napokon ešte popíšme ako reprezentujem samotnú animáciu kostry. Tentokrát je to pole kvaterniónov. Iné parametre kostí si tu pamätať netreba, lebo tie sa už počas behu animácie nemenia (ako napríklad dĺžka či indexy potomkov). To preto, lebo tie sa počas animácie už nemenia.

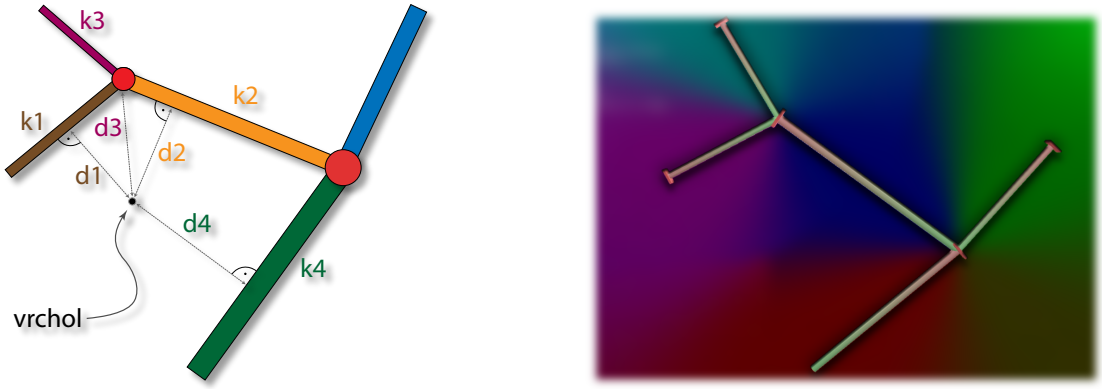
No a pre každú z týchto rotácií si ešte pamätám jeden GLfloat popisujúci čas. Tento znamená, že presne v tomto čase bude mať kostra práve takéto rotácie kostí. Nuž ale keďže väčšinou potrebujem nie jednu z týchto hodnôt, ale nejakú spomedzi nich, tak tieto hodnoty interpolujem. Dáta skladujem v osobitnom objekte nazvanom SkeletonAnimation, nezávisle na objekte samotnej kostry. Má to takú výhodu, že viacero kostier takto môže zdieľať tú istú animáciu (a prípadne bežať nejakú navzájom rôznu fázu). Vtedy keď má kostra záujem sa pohnúť, jednoducho zavolá na sebe jednu metódu a tá nastaví rotácie kostí podľa animácie a požadovaného času.

3.2 Naviazanie model - kostra

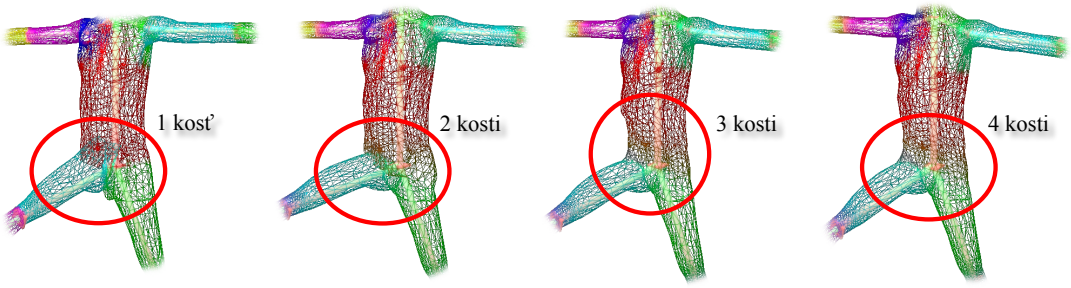
V práci používam modely generované programom TrueSpace od firmy Caligari, ktoré viem zo súboru načítať a použiť pre účely animácie. Kostry si zase uskladňujem vo vlastnom formáte, do ktorého sa dajú ľahko ukladať a z ktorého ich aj načítavam. Keď už teda máme načítaný model aj kostru, treba ich spojiť. Ide vlastne o naučenie modelu, ako má na pohyby kostry reagovať. Ja túto fázu riešim tak, že pre každý vrchol nájdem štyri najbližšie kosti. Vzdialenosti počítam ako vzdialenosť bodu od úsečky a na základe tejto vzdialenosti následne nastavím váhy vplyvu danej kosti. Váha rastie nepriamo úmerne s kvadrátom vzdialenosti a nakoniec sú tieto váhy normalizované, tj. zabezpečím, aby v súčte dávali jednotku.

Skúšal som aj rôzne obmeny výpočtu týchto váh, napríklad že boli nepriamo úmerné prvej mocnine vzdialenosti, alebo až odmocnine. Výsledky sú v zásade podobné, avšak je viditeľný rozdiel v plynulosti prechodov jednej zóny vplyvu do druhej. Kým s použitím druhých mocnín vdzialeností v prvom prípade sú tieto zóny menšie, a teda končatiny sa deformujú len v bližšom okolí kĺbov, v prípade funkcií pomalšie klesajúcich so vzdialenosťou sa deformujú až príliš plynulo, čo má za následok, že aj niektoré pôvodne rovné úseky sa zakrivujú.

Ďalší faktor s ktorým som poexperimantoval je počet kostí na ktoré sa môže jeden vrchol naviazať. Začínal som s dvomi. Postava sa už vedela hýbať (a vedela by to aj s viazaním 1 kosť / 1 vrchol), ale vznikali na nej oku nie príliš lahodiace chyby: roztrhnutia, vtláčanie jednej časti tela do inej... Preto som program rozšíril na 4 kosti / 1 vrchol a značná časť týchto problémov zmizla.



Obr. 7: Najbližšie kosti a ich vzdialenosti. Vľavo je nákres s vyznačenými vzdialenosťami pre každú z kostí k_1 až k_4. Za takéhoto rozloženia by mala najväčšiu váhu kosť k_1, potom k_2, následne k_3 a napokon k_4. V pravo je ukážka z programu ako by za daného usporiadania kostí vyzeralo rozloženie zón vplyvu.



Obr. 8: Porovnanie vplyvu počtu kostí/vrchol na kvalitu ohybu kĺbov.

Avšak treba spomenúť dva fakty. Jednak že nie všetky nedostatky sa dajú riešiť jednoduchým zvýšením tohto počtu. A tiež to, že tým pribudol nový problém. Nie obzvlášť hrozivý, ale nový. A to taký, že v niektorých prípadoch má vrchol tendenciu viazať sa na priveľa kostí. Tým nemys-

lím na viac ako 4, ale viac ako by sme si priali. Napríklad ak je lýtko ľavej nohy ovplyvňované pravým lýtkom. Je pravda, že v čase viazania boli pomerne blízko, ale zároveň nechceme, aby pohyby ľavej nohy ovplyvňovali takýmto spôsobom pravú. Dá sa to riešiť rôzne. Buď rôznymi komplikovanými metódami, ktoré berú pri výpočte do úvahy nie jednoduchú euklidovskú vzdialenosť bodu a úsečky, ale dĺžku najkratšej cesty od vrcholu ku kosti **vovnútri** modelu. Alebo druhá možnosť je mať na začiatku model dostatočne upažený, rozkročený, prípadne s roztiahnutými prstami... Tretia možnosť je na animáciu použiť detailnejšiu kostru (viac kratších kostí). Ja som sa snažil použiť druhý spomenutý prístup.

Takže, štyri indexy a štyri váhy si potom uložím spolu s dátami modelu. Navyše však prepočítam aj pozície všetkých vrcholov do sústavy každej z kostí, na ktoré sa viažu. To znamená že od pôvodnej pozície vrcholu odpočítam začiatok kosti a otočím ho rotáciou inverznou ku orientácii kosti. Takto prepočítam vrchol aj podľa druhej až štvrtej kosti a opäť, uložím si výsledky. Čo si tu treba uvedomiť je, že nestačí nám pretransformovať len samotné pozície vrcholov, ale aj všetky ostatné geometrické vlastnosti s nimi spojené, ako napríklad často býva informácia o normálovom vektore, či dotyčnici. V tejto práci nevyužívam normal-mapping (proces pri ktorom sa na povrchu objektu podľa nejakej textúry mierne menia normálové vektory, čo potom často dodá plastickejší vzhľad), preto nebudem potrebovať dotyčnice, ale keďže objekty v scéne sú tieňované, normály budem musieť tak isto previesť do sústavy kostí. V tomto prípade to bude o to jednoduchšie, že vektor iba rotujem.

3.3 Prepočítavanie kostry

Jedna z možností je shaderu poslať iba relatívne rotácie v kĺboch a dĺžky kostí. On by si potom našiel pre každú kosť jej predka, potom jeho predka a zase jeho predka a až by prišiel po koreň, po ceste by si násobil kvaternióny a nakoniec by mal orientáciu danej kosti. Riešenie ktoré používam ja je prakticky úplne rovnaké, ibaže sa vykonáva na inom mieste a menej krát. Ja prejdem celým stromom kostry ešte v aplikácii a shaderu už pre každú kosť pošlem výslednú orientáciu. Toto beží na CPU, ale nie je to problém. Kostra je totiž veľmi malá a bude to rýchlo hotové. Výhodou takéhoto prístupu je jednak odbremenenie GPU, ale hlavne výrazné ušetrenie výpočtov. Problém je totiž v tom, že vertex shader sa spustí pre jeden vrchol a potom skončí. Potom sa síce spustí pre ďalší vrchol, ale už si nebude pamätať nič z predošlého behu. A preto bude kostrou prechádzať hore ku koreňu znova a znova, pre každý vrchol modelu. Avšak kým pracujem v hlavnom programe a mám pohodlný prístup k množstvu pamäti, môžem celou štruktúrou prejsť len raz, výsledky si priebežne ukladať a nakoniec ich shaderu odovzdať. Takže za cenu minimálneho zaťaženia centrálneho procesora výrazne urýchlim výpočet vrcholov vo vertex shaderi. Navyše, keďže sa pri prepočte pozícií a rotácií nevyužívajú nijaké OpenGL

volania, dajú sa tieto výpočty odsunúť na vedľajšie vlákno a zatiaľ robiť niečo iné. Takže veľmi jednoducho sa dá na začiatku renderovacieho cyklu spustiť aktualizácia všetkých kostier používaných v scéne a kým sa dostaneme k ich použitiu, najpravdepodobnejšie už budú hotové.

3.4 Výpočet v shaderi

Napokon sme vo fáze, že nastupuje vertex shader. Tomu program poslal (spôsob je načrtnutý v časti 1.2) zopár vstupných dát: 4x pozíciu vrchola (p_i), 4x normálový vektor (n_i), 4x index kosti, 4x váhu (w_i) a v neposlednom rade celú kostru. Tá však nesie iba mninimum potrebných údajov a to pre každú kosť pozíciu jej začiatku a orientáciu (kvaternión reprezentujúci odklon od [0,1,0]). A jeho úlohou je poslať na výstup 1x pozíciu vrchola a 1x normálu.

Podľa štyroch indexov si v kostre nájdeme potrebné údaje. Budú to 4D kvaternióny orientácie: q_0 až q_3, 3D vektory polohy kosti t_0 až t_3. Keďže máme v shaderi definovanú funckiu rotácie vektora kvaterniónom, môžeme transformáciu zapísať takto:

$$p = \sum_{i=0}^{3} ((\ \mathrm{rotateQ}(\ p_i, q_i\ )\ + t_i) * w_i)$$
$$n = \sum_{i=0}^{3} (\ \mathrm{rotateQ}(\ n_i, q_i\ )\ * w_i)\,,$$

kde p je výsledná pozícia vrchola a n je výsledná normála v tomto bode. Okrem iného tu využívame fakt, že súčet váh w_0 až w_3 je 1, a teda ide o vážený priemer vektorov ovplyvnených jednotlivými kosťami.

3.5 Relevantné časti kódu

3.5.1 Aktualizácia pozícií kostí

Úlohou tejto metódy je z relatívnych rotácií a dĺžok kostí vypočítať jednak ich absolútne pozície v priestore, čiže začiatok a koniec a jednak orientáciu voči vopred stanovenému vektoru (0,1,0).

```
procedure Skeleton.computeBonesPositions;
    procedure findBone( index: Integer );          //vypočíta pozíciu a orientáciu kosti
    const defaultOrientation: vector3D = ( 0,1,0 );
    var direction: vector3D;
        i, childId: Integer;
        orientation: quaternion;
    begin;
        orientation := bones[ index ].orientation;     //orientácia predchodcu kosti index
        orientation := multiplyQuaternion( orientation, bones[ index ].transform );
            //teraz je to už výsledná orientácia kosti index
        direction := rotateVector( defaultOrientation, orientation );     //smerový vektor kosti
        direction := multiplyVector( direction, bones[ index ].boneLength );     //...vynásobený jej dĺžkou


        bones[index].E := addVector( bones[index].B, direction );          //koniec = začiatok + smer
        bones[index].orientation := orientation;


        for i := 0 to Length( bones[ index ].children )-1 do begin;          //pre každého z potomkov...
            childId := bones[ index ].children[ i ];          //childId je teraz index dieťaťa
            bones[ childId ].B := bones[ index ].E;          //potomok začína kde rodič skončil
            bones[ childId ].orientation := orientation;          //a zatiaľ má aj jeho orientáciu
            findBone( childId );          //pokračuje do hĺbky
        end;
    end;

begin;
        //kontrola validnosti kostry a tak
    ...
    bones[ root ].B := position;          //pozícia začiatku koreňa
    bones[ root ].orientation := identityQuaternion;
    findBone( root );
end;
```

3.5.2 Uploadovanie kostry a samotné vykresľovanie

V prvej fáze treba poslať kostru, čiže začiatok každej kosti a jej orientáciu, do vertex shadera. V druhej fáze renderujeme model. Vidno tu iba volanie procedúry, ale v konečnom dôsledku sa aj tak iba zavolá taskList - lebo v týchto mám uložený celý model. V momente pripnutia modelu na kostru ho vytvorím a potom už iba volám. Je to tak rýchlejšie.

```
     //upload kostry do shadera
engine.useProgramObject( animationPO );    //povieme OpenGL aby teraz používal náš shader
for i := 0 to Length( Bonie.bones )-1 do begin    //za každú jednu kosť kostry...
        //v shaderi mám definované pole boneRotation[]. Najprv získam adresu i-teho prvku...
    uniPos := getUniformLocation( animationPO, 'boneRotation['+inttostr(i)+']' );
        //... a potom doň zapíšem aktuálnu hodnotu
    glUniform4fv( uniPos, 4, @( Bonie.bones[ i ].orientation ) );
        //tak isto aj s bonePosition[]
    uniPos := getUniformLocation( animationPO, 'bonePosition['+inttostr(i)+']' );
    glUniform3fv( uniPos, 3, @( Bonie.bones[ i ].B ) );
end;

    //renderovanie modelu (SAM sa volá náš model - ako skratka od SkeletonAnimatedModel)
if not Assigned( SAM )then begin;    //iba pre istotu, ak by došlo k nejakej chybe
    log('Error: Mesh does not exist.');    //zapíš do logu hlásenie a skonči
    halt;
end;

drawSAMesh( SAM );    //vyrenderujeme model
turnOffShaders;    //a vypneme shader
```

3.5.3 Rotácia vo vertex shaderi

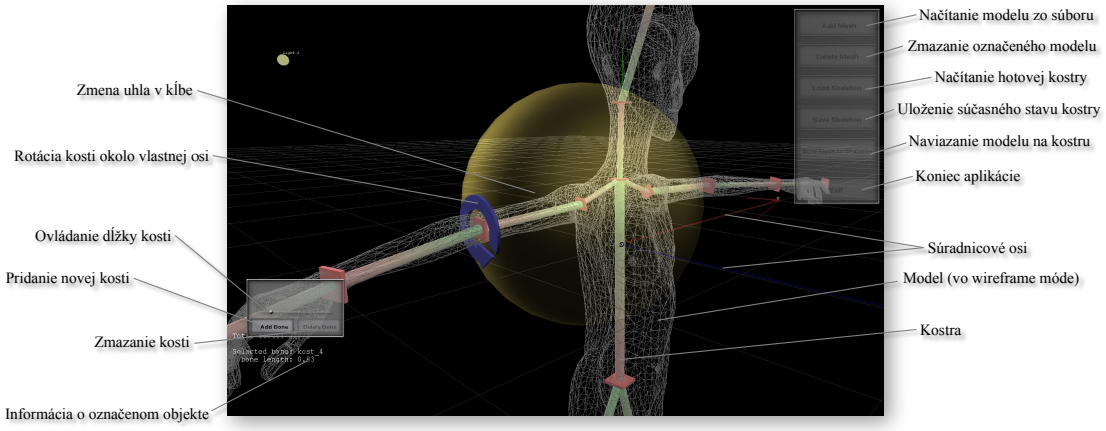
Vychádzam zo vaťahov (3) (vďaka vektorovo orientovanému zápisu je toto vyjadrenie na karte o torchu rýchlejšie ako (2)) a (4). Dá sa to zapísať aj krajšie, ale ja som sa snažil minimalizovať množstvo volaní podfunkcií a rozpísal rotáciu nasledovne:

```
vec3 rotateVector( vec3 v, vec4 q ){       //zrotuje vektor v kvaterniónom q
   vec4 result, vector;
      //zamaskujem v za kvaternión vector
   vector = vec4( 0,v );
      //násobenie q * vector
   result.x = q.x * vector.x - dot( q.yzw, vector.yzw );
   result.yzw = q.x * vector.yzw + vector.x * q.yzw + cross( q.yzw, vector.yzw );
      //q si nahradíme jeho združeným
   q.yzw = -q.yzw;
      //násobenie result * q
   vector.x = result.x * q.x - dot( result.yzw, q.yzw );
   vector.yzw = result.x * q.yzw + q.x * result.yzw + cross( result.yzw, q.yzw );
      //odmaskovanie a return
   return vector.yzw;
}
```

4 Výsledky

4.1 Ovládanie programu

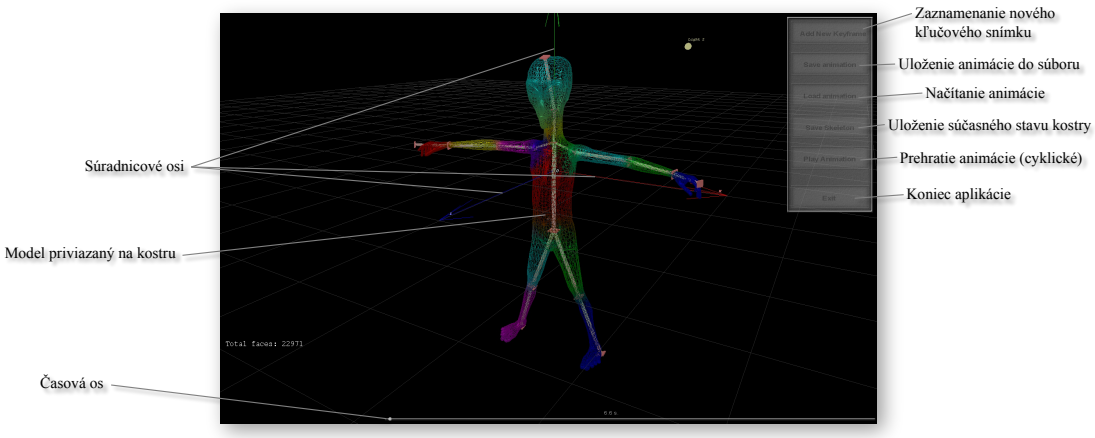
Práca s programom sa dá rozdeliť na dve hlavné časti - pred a po naviazaní modelu na kostru. V tej prvej sa dajú do scény načítať modely a kostry nezávisle na sebe. V tejto fáze sa dá načítaná kostra ľubovoľne meniť, dajú sa pridávať, odoberať, predlžovať a skracovať kosti. A samozrejme dajú sa meniť rotácie v kĺboch. Na načítavanie slúži niekoľko tlačidiel na pravom paneli. Úpravy kostry sa realizujú osobitne pre každú kosť. Pri kliknutí pravým tlačidlom myši na kosť sa táto prepne do editovacieho módu a v ľavej časti okna sa ukáže malý panel slúžiaci na úpravu danej kosti. Dĺžka sa dá nastaviť posuvnou lištou, dvoma tlačidlami je možné pridať kosť „potomka", alebo označenú kosť zmazať. Transformácie v kĺboch sa dajú meniť priamo v 3D a to pomocou priesvitnej sféry, ktorá sa v režime úprav zobrazí okolo začiatku označenej kosti. Kliknutím na sféru sa kosť nasmeruje týmto smerom. Aby sa dali meniť všetky tri zložky rotácie, na konci kosti je umiestnená malá šípka, kliknutím na ktorú sa kosť točí okolo vlastnej osi.



Obr. 9: Prvá fáza - vzhľad okna, objekty v scéne a ovládacie prvky.

Keď je už na scéne model ktorý chceme hýbať a aj kostra je podľa našich predstáv, môžeme prejsť k druhej fáze. Urobíme tak jedným kliknutím, ktoré má za následok práve už spomínaný proces naviazania modelu na kostru. Tieto väzbové dáta sa už neskôr nemenia. Keďže program má byť hlavne ilustratívny, model sa zafarbí podľa zón vplyvu jednotlivých kostí. Povrch potom pochopiteľne nevyzerá realisticky, ale zato tak dobre vidno napríklad ako plynulo prechádzajú váhy kostí pozdĺž kĺbu. Toto je samozrejme len pomôcka pri vnímaní procesu samotného deformovania a dá sa ľahko prestaviť na nejaké krajšie zafarbenie.

V tejto, druhej, fáze sa už dá kostra meniť len obmedzene. Nedajú sa ani pridávať, ani odoberať kosti a takisto ani upravovať ich dĺžky. Čo sa však meniť dá, sú rotácie v kĺboch. Tie sa menia rovnakým spôsobom ako v predošlej fáze s tým rozdielom, že tentokrát už model sleduje pohyby kostí. Aby sme pristúpili k samotnej animácii, je potrebné mať niekoľko kľúčových snímkov. Teraz využijeme aj lištu v dolnej časti okna. Je to časová os. Keď chceme pridať kľúčovú pozíciu, stačí nastaviť na lište správny čas, popohybovať kostrou a nakoniec kliknúť tlačidlo na zaznamenanie kľúčového snímku. Ten sa zaznamená do vyššie opísanej štruktúry a pri najbližšej príležitosti bude započítaný do interpolovanej polohy kostry.



Obr. 10: Druhá fáza - vzhľad okna, objekty v scéne a ovládacie prvky.

4.2 Slabé miesta

Ako nič v prírode, ani tento postup nie je dokonalý. Vyskytujú sa pri ňom niektoré chyby či chybičky, na ktoré je poctivé upozorniť. Prvá vec, ktorú treba spomenúť, sú sploštieniny na kĺboch. Čím väčší uhol zvierajú kosti oproti pôvodnému štádiu, tým je tento efekt viditeľnejší, a teda rušivejší. Uvažoval som nad interpoláciou vyššieho rádu, ale neukázalo sa to byť dostatočným riešením. Odhliadnuc od faktu, že by sa tým rendering ešte spomalil, má takýto spôsob i iný nedostatok: ak nie sú kosti idúce za sebou, ani kubická krivka neurobí to, čo potrebujeme. Napríklad v prípade že by sme mali pomerne kvalitne vypracovanú dlaň, tak vrcholy zo zóny vplyvu záprstných kostí práve potrebujú lineárnu interpoláciu. Dotyčnice kostí by totiž smerovali rovnakým smerom (ku končekom prstov) a v tejto zóne to je zjavne nežiadúce. Ďalší problém nastáva, ak sa kosť točí v smere okolo vlastnej osi o uhol blízky 180°. Vtedy sa daná končatina v istej zóne vcucne do jedného bodu.

Obr. 11: Nedostatky súčasného prístupu: sploštenia (vľavo) a zaškrtenia (vpravo).

Je to spôsobené tým, že vo vertex shaderi sa lineárne interpolujú už rotované vektory. Pekné riešenie pre končatiny by bolo neinterpolovať transformované pozície, ale samotný kvaternión a tým následne rotovať vstupný vrchol. Avšak tento prístup zlyháva ak kosti, na ktoré sa vrchol viaže, nie sú nasledovné, čo sa žiaľ stáva často. Iný spôsob využívajú autori článku [4]. Spočíva v pridávaní pomocných kĺbov na miesta, kde takéto poruchy vznikajú. Tak aj pre 180° uhol bude každý z kĺbov pretočený len o 90°, čo zvláda bez väčších problémov. V prípade záujmu odporúčam prečítať.

4.3 Rýchlosť

Na testovanie som používal zostavu s Windows XP Home Edition SP3, procesorom Intel Core2 Duo E6550 @ 2.33Ghz, pamäťou 2GHz DDR2, grafikou ATI Radeon HD 2400 s 512MB pamäte a 40 shader procesormi. Program bežal v 1680 x 1050 px móde bez antialiasingu. Samotné užívateľské prostredie spolu s obsluhou všetkých nevyhnutných vecí ohľadom bežania programu zaberajú cca 8ms. Model, ktorý som používal, má približne 12000 vrcholov a 23000 trojuholníkov. Po načítaní modelu do programu trvá renderovanie jedného obrázku približne 16ms, po naviazaní na kostru a teda s aplikáciou animácie táto doba vzrastie na 18ms (čo zodpovedá 56 vykresleniam za sekundu).

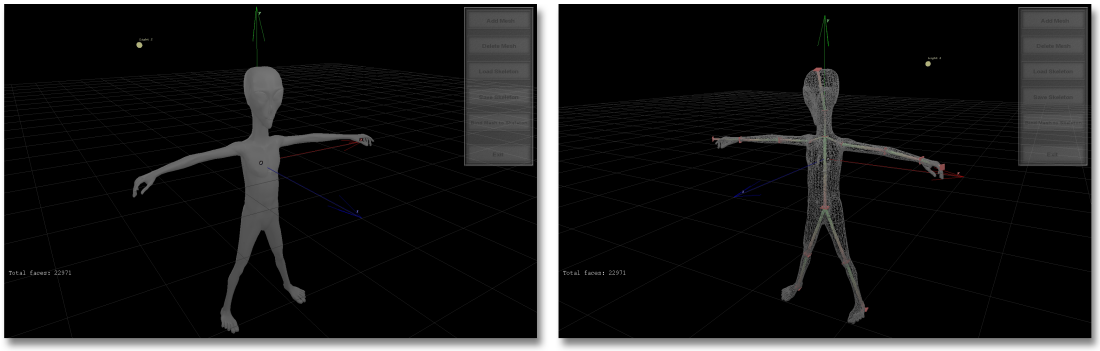
Skúšal som program s niekoľkými, rôzne zložitými, modelmi. Všetky vznikli z pôvodného mimozemšťana buď zdvojením všetkých vrcholov a hrán, alebo naopak, zredukovaním ich počtu. Na tento účel som použil nástroj programu TrueSpace 'Polygon Reduction Tool'. Tu sú namerané

výsledky:

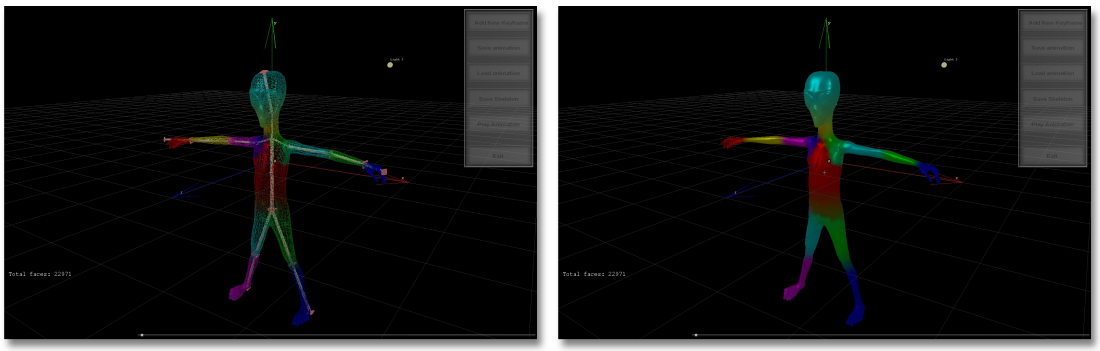
model	# vrcholov	# trojuholníkov	čas	fps
redukovaný (2)	547	1362	13 ms	77
redukovaný (1)	2501	5539	16 ms	63
pôvodný	11735	22971	18 ms	56
dvojnásobný	23470	45942	27 ms	37
štvornásobný	46940	91884	45 ms	22
šesťnásobný	70410	137826	63 ms	16

4.4 Výsledné obrázky

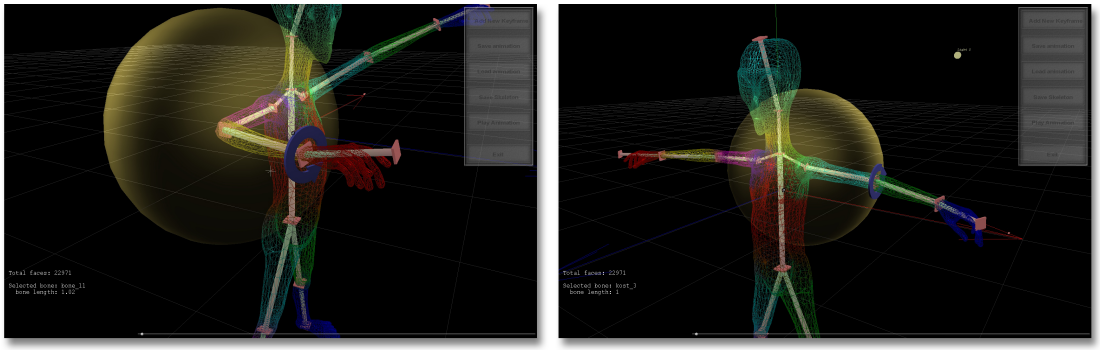
Pripájam zopár screenshotov z aplikácie čo som vytvoril. Prvé 2x tri zobrazujú užívateľské prostredie, načítavania a fázy práce s aplikáciou, kým zvyšné zachytávajú konkrétne pózy ktoré postava v programe zaujala. Tieto obrázky sú vždy vo aj vo wireframe verzii, aj v móde celistvého modelu. Na niektorých je kvôli krajšiemu obrázku vypnuté renderovanie kostry.



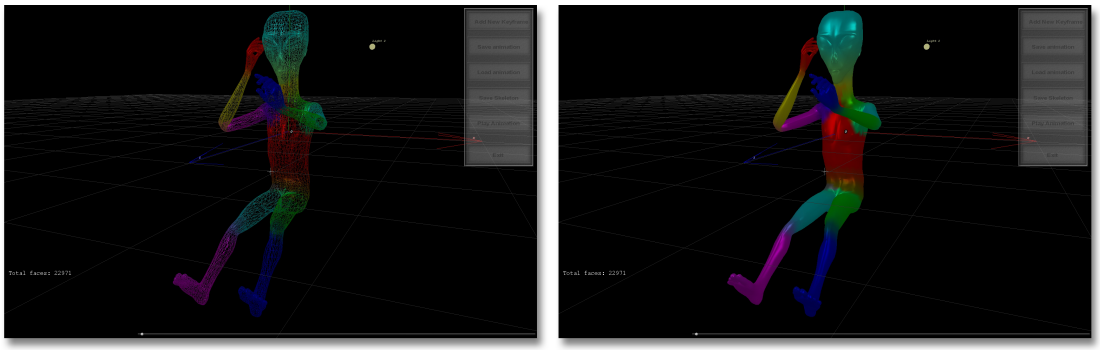
Obr. 12: Načítanie modelu a kostry do programu.



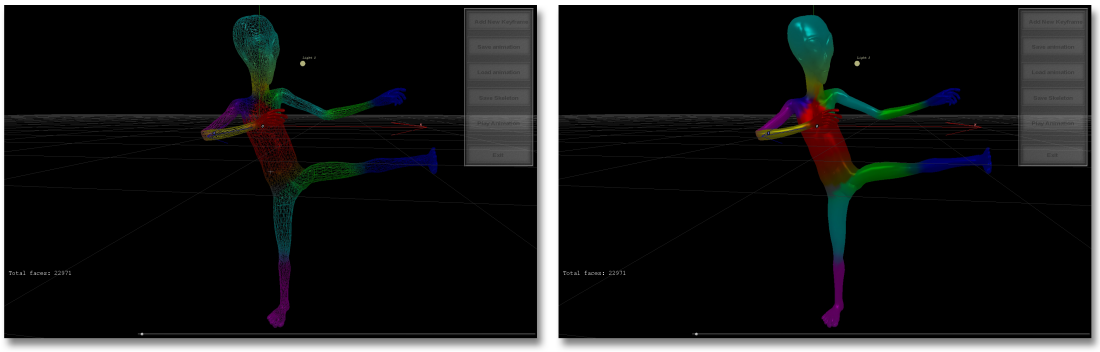
Obr. 13: Naviazanie modelu na kostru.



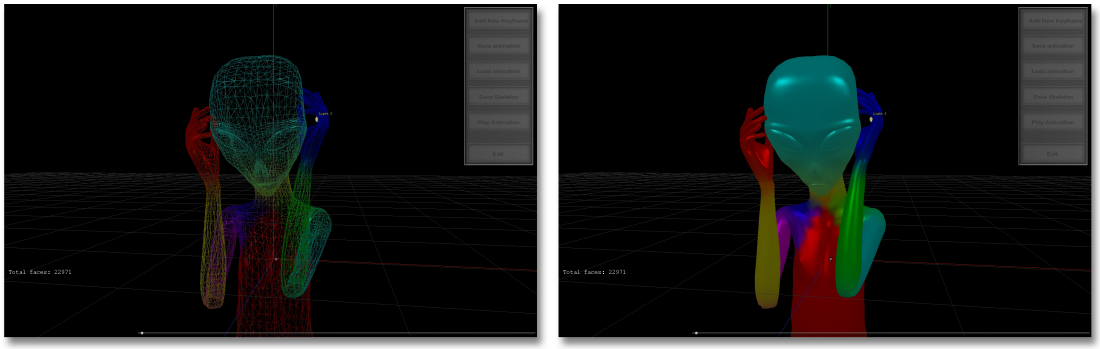
Obr. 14: Úprava rotácií v kĺboch.



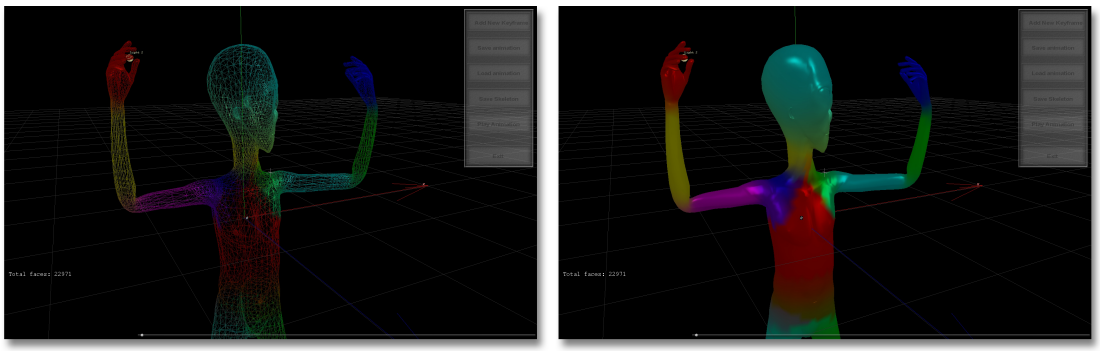
Obr. 15: Póza 1.



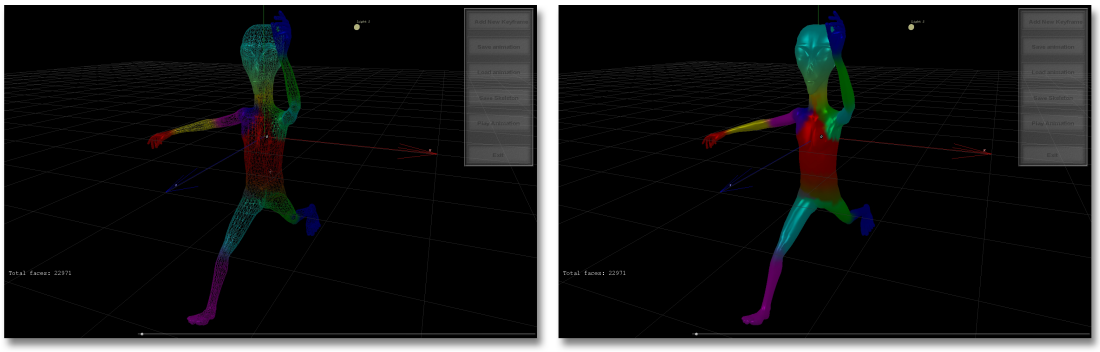
Obr. 16: Póza 2.



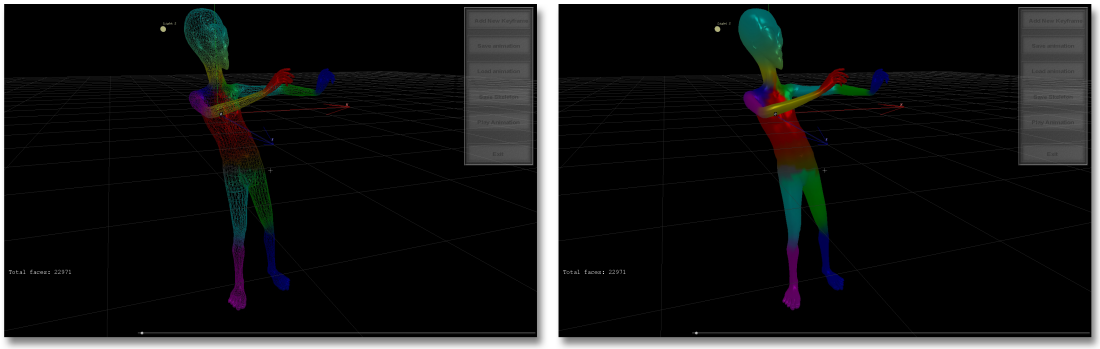
Obr. 17: Póza 3.



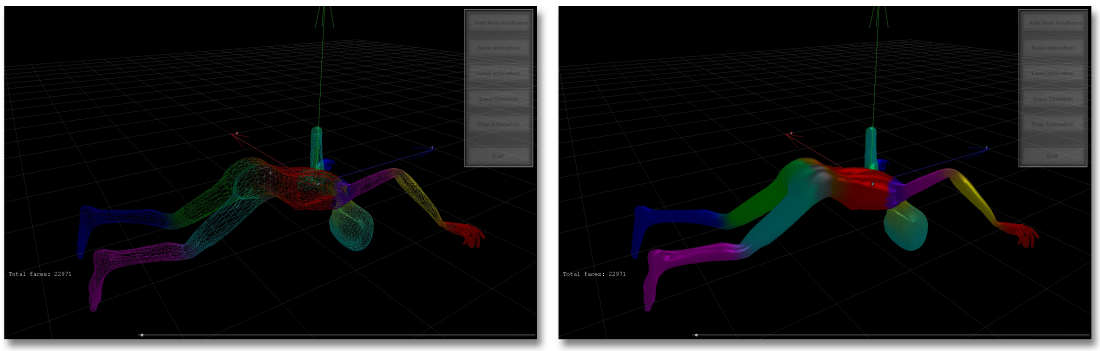
Obr. 18: Póza 4



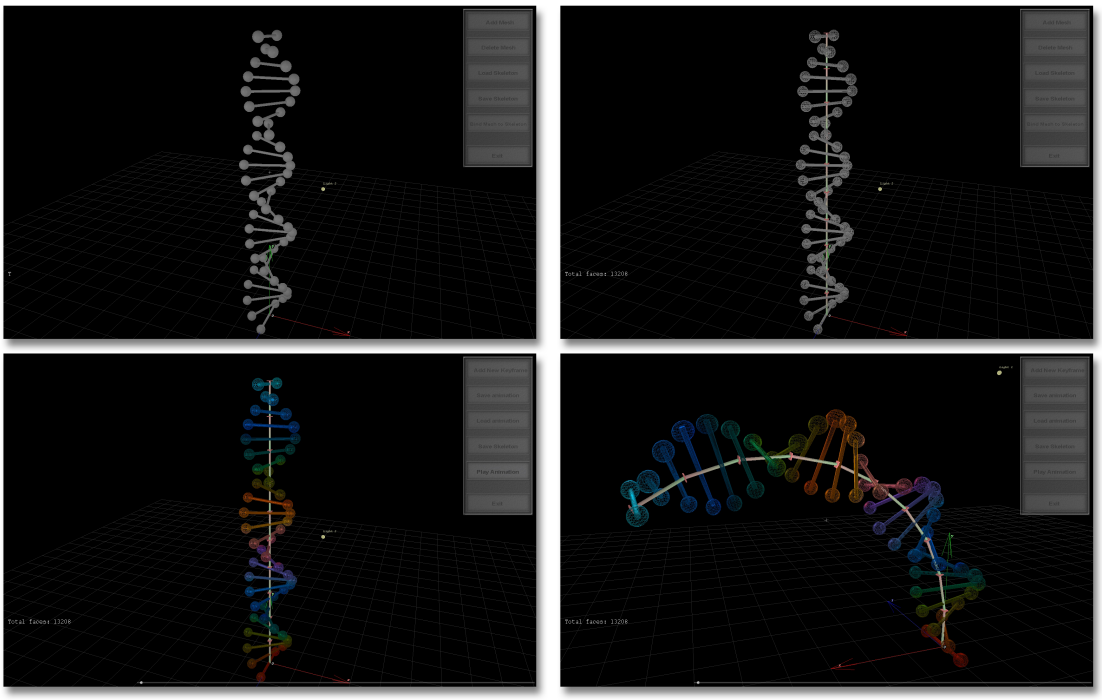
Obr. 19: Póza 5.



Obr. 20: Póza 6.



Obr. 21: Póza 7



Obr. 22: Keďže som spomenul že metóda sa dá použiť nielen na pohyb postáv, tak ako malý príklad ukážem, ako sa dvojzávitnica DNA dá animovať pomocou kostry.

Záver

Odprezentoval som metódu často využívanú na animáciu postáv aj v interaktívnych aplikáciách. Načrtol som matematický podklad, uviedol som, ako sa pomocou komplexných čísel dajú reprezentovať rotácie v rovine, tiež som sa dostal k ich rozšíreniu v podobe kvaterniónov. Opísal som, ako sa nimi dajú kódovať trojrozmerné rotácie a ako vyzerá výpočet otočeného vektora. Popísal som algoritmy, ktoré som využíval a implementáciu dôležitých častí programu, ako napríklad naviazane modelu a kostry, prepočet pozícií kostí, alebo transformáciu vo vertex shaderi. Spomenul som ohraničenia a nedostatky tohoto prístupu, načrtol som ich možné riešenia a odkázal na niekoľko článkov, ktoré sú už nadstavbou nad jednoduchý systém popísaný v tejto práci a riešia už chyby, ktoré ja ignorujem. Tiež som ukázal niekoľko príkladov póz, ktoré postava v mojom programe zaujala, ako aj príklad poukazujúci na fakt že animovaný objekt nemusí byť nevyhnutne postava. Som si samozrejme vedomý, že je stále čo vylepšovať, jednak kvalitu ohybu v kĺboch, kde za istých podmienok vznikaú nepekné artefakty, ako aj rýchlosť tohoto procesu. Naprogramoval som aplikáciu, ktorá demonštruje postupy popísané v tomto texte a ktorá je súčasťou bakalárskej práce. Celkovo si myslím, že práca splnila moje očakávania, dosiahol som, čo som si na začiatku zaumienil. Dúfam, že splnila aj očakávania čitateľa.

Ďakujem za pozornosť.

Literatúra

[1] *Simo Särkkä*. 2007. Notes on Quaternions (http://www.lce.hut.fi/ ssarkka/pub/quat.pdf).

[2] *Mark Segal, Kurt Akeley*. 2008. **The OpenGL Graphics System: A Specification** (Version 3.0 - September 23, 2008).

[3] *John Kessenich*. 2006. **The OpenGL Shading Language** (Language Version: 1.20).

[4] *Alex Mohr, Michael Gleicher*. 2003. **Building Efficient, Accurate Character Skins from Examples**. SIGGRAPH 2003.

[5] *Rich Wareham, Joan Lasenby*. 2008. **Bone Glow: An Improved Method for the Assignment of Weights for Mesh Deformation**. Articulated Motion and Deformable Objects, str. 63-71.

[6] *Doug L. James, Christopher D. Twigg*. 2005. **Skinning mesh animations**. SIGGRAPH 2005.

[7] *Dave Kerr*. Tutoriál na nehe.gamedev.net/data/articles/article.asp?article=01

[8] *Rôzni autori*. Stránka nehe.gamedev.net s peknými lekciami na naučenie sa OpenGL.