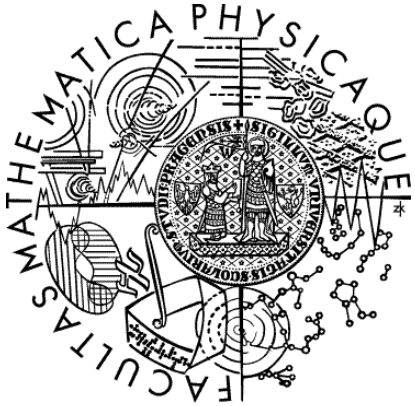
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Petr Šebor

Skeletální animace

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Josef Pelikán

Studijní program: Informatika, Aplikovaná informatika

2006

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 30. května 2006

Obsah

Název práce: Skeletální animace
Autor: Petr Šebor
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: RNDr. Josef Pelikán
e-mail vedoucího: Josef.Pelikan@mff.cuni.cz

Abstrakt: Skeletální animace a prostorová optimalizace animačních dat je v současné počítačové grafice stále živé téma a otevřený problém. Cílem této práce je nalezení jednoduchého, efektivního a snadno implementovatelného algoritmu, který poskytne řešení problému za pomoci aproximace zdrojových dat neuniformními neracionálními B-splines s výhledem na budoucí práci, jak dále tuto metodu zlepšovat a zefektivňovat.

Klíčová slova: Skeletální animace, B-spline, Aproximace animačních dat.

Title: Skeletal animations
Author: Petr Šebor
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Josef Pelikán
Supervisor's e-mail address: Josef.Pelikan@mff.cuni.cz

Abstract: Skeletal animation and spatial optimization of animation data is still an open problem in contemporary computer graphics. The contribution of this work is to provide a functional, effective and easily implementable solution of the optimization problem with the help of the keyframe fitting and approximation by non uniform non rational B-splines with an outlook on what improvements can make this method even more flexible and useful.

Keywords: Skeletal animation, B-spline, Animation data approximation.

1. Úvod a motivace

Cílem této práce je zkoumání a realizace základních algoritmů skeletální animace a nalezení algoritmu paměťové optimalizace animačních dat. Metoda aproximace animačních dat funkcemi je neustále živé téma a bývá předmětem nesčetných debat na elektronických diskuzních fórech.

Tato práce navazuje na můj ročníkový projekt, který se soustředil výhradně na samotné pricipy skeletálních animací. Prostorová optimalizace animačních dat jejich aproximací funkcemi je úloha, která mě oslovovala již delší dobu nejenom z osobního, ale i z profesionálního hlediska. V praxi se setkávám s animacemi, které svojí velikostí začínají v některých případech narážet na paměťové limity, a tak začala být otázka řešení problému optimalizací velmi aktuální. Ve světě existuje velice málo použitelných implementací a jakékoliv informace se na toto téma získávají jen velmi těžko. V podstatě jediné zdroje informací, které mi poskytly dostatek indicií k úspěšné implementaci byla výhradně různá diskuzní fóra a skupiny; ani tam ale nepanoval jednotný názor na možné postupy, které vedou k úspěšnému cíli.

2. Problematika

2.1. Animace

Mluvíme-li v trojrozměrné grafice o animaci, máme na mysli proces provádějící modifikaci bodů trojúhelníkové sítě (triangle mesh), který reprezentuje povrch animovaného objektu. My se budeme zabývat pouze jednou metodou, která se nazývá „skeletální animace“. Existují ještě další metody, jako například keyframe animace, které jsou ale z dnešního hlediska dávno překonané, protože kladou příliš veliké nároky na hardware.

2.2. Skeletální animace

Název „skeletální“ skutečně vystihuje podstatu, na jejímž pricipu tento algoritmus funguje. Motivace je zhruba následující: budeme animovat jednoduchou kostru, na kterou je (prozatím) blíže nespecifikovaným způsobem navázán trojúhelníkový mesh. Způsob, jakým je kostra za běhu modifikována, nám dává společně s ostatními daty jednoznačný návod, jak modifikovat trojúhelníkovou síť na povrchu objektu.

Pojďme se nejprve podívat, jak taková kostra může vypadat, a co vlastně reprezentuje. Kostra modelu v podstatě není nic jiného než stromová datová struktura (nemůže tedy obsahovat žádné cykly), která má právě jeden kořen. Dále také platí, že každá kost kromě kořenové má právě jednu nadřazenou kost (parent bone) a každý takový uzel našeho stromu reprezentuje lokální transformaci pro celý svůj podstrom. Dá se také říci, že v každé kosti je uložena informace o tom, jak daleko a pod jakým náklonem je k této kosti připojena kost další. Podívejme se například, co by se stalo, kdybychom v polovině naší stromové struktury pootočili jednou kostí. Znamenalo by to, že jsme tím pádem ovlivnili natočení všech kostí v celém podstromu, který je na námi modifikované kosti zavislý. Pokud bychom měli kostru v nějaké základní poloze a dále předpis, jak v čase modifikovat natočení jednotlivých kostí, rázem bychom získali funkční systém, který je schopen poskytovat data pro algoritmus zodpovědný za deformaci trojúhelníkové sítě.

2.3. Animační data

Animační data pro skeletální animaci jsou bežně ukládána stejným způsobem jako data pro keyframe animaci po jednotlivých snímcích s určitou frekvencí. Na data jednoho snímku se můžeme dívat jako na soubor jednotlivých kostí, kde každá kost obsahuje translaci a rotaci kolem jednotlivých os souřadného systému. Na každý takový snímek se můžeme dívat jako na stavový vektor

$$\theta = \{x_1, y_1, z_1, \alpha_1, \beta_1, \gamma_1, x_2, y_2, z_2, \alpha_2, \beta_2, \gamma_2, \ldots\},$$

kde index určuje příslušnost k dané kosti a jednotlivé prvky vektoru tvoří části každé kosti. Nejprve posunutí v lokálním prostoru nadřazené kosti, a dále pootočení v lokálním prostoru nadřazené kosti.

2.4. Kvaterniony

Na chvíli odbočíme ke struktuře, která nám významným způsobem zjednoduší práci s rotacemi jednotlivých kostí. Pokud se všeobecně bavíme o obecné rotaci, většinou si ji představíme jako trojici rotací kolem jednotlivých os souřadného systému. Podle zažitých konvencí se většinou předpokládá, že se rotace kolem jednotlivých os provádějí v pořadí X, Y a nakonec Z, můžeme se ale setkat i s výjimkami, kde se předpokládá zcela jiná konvence. Jedná se o takzvané eulerovy rotace, a jak je na první pohled vidět, rotace zadaná jako seznam rotací kolem jednotlivých os může být velice nejednoznačná, přestože jejich použití může na první pohled vypadat jako zřejmé a intuitivní. Můžeme se ale setkat s dalšími komplikacemi, kdy budeme potřebovat například plynule rotovat objekt za použití rotací kolem všech tří os. Fakt, že bychom interpolaci rotace vyjádřenou pomocí eulerových úhlů prováděli po složkách, může vést k tomu, že se výsledný objekt nebude otáčet přirozeně a bude v prostoru opisovat zcela nepřirozené křivky.

Rotace se také dá vyjádřit pomocí kvaternionů. Jedná se vlastně o čtyřrozměrné komplexní číslo s jednou reálnou a třemi imaginárními složkami. Jak přesně je kvaternion definován, jak funguje a jak se s ním pracuje můžeme zjistit například v knize Marka a Alana Watt [4] nebo na MathWorldu [5]. Je třeba zdůraznit, že

od této chvíle budeme mluvit výhradně o rotacích reprezentovaných pomocí jednotkových kvaternionů. Rotaci vyjádřenou pomocí kvaternionu si můžeme představit jako rotaci kolem jednotkového vektoru v trojrozměrném prostoru. Převod takové rotace do podoby kvaternionu je velice triviální.

$$q=(s,\boldsymbol{v})=(w;x,y,z)=(\cos(\frac{1}{2}\gamma),\bar{n}\sin(\frac{1}{2}\gamma))$$

Nejdůležitější je asi „přátelské“ chování kvaternionů vzhledem k interpolaci. (Připomeňme, že pod pojmem interpolace se rozumí plynulý přechod mezi dvěma kvaterniony.) Samotných interpolačních metod existuje několik, my si však povíme pouze o dvou z nich.

3. Interpolace animačních dat

3.1. Lineární interpolace vektorů

Pokud chceme interpolovat data, která jsou reprezentována pouze jako vektor (což je například translace), budeme výhradně používat metodu lineární intepolace.

$$\boldsymbol{lerp}(p_0, p_1; t) = (1-t)\,p_0 + t\,p_1$$

Pro kvaterniony se běžně používá operace *slerp*, my si ale ukážeme, že si vystačíme s metodou *lerp* i u kvaternionů.

3.2. Sférická lineární interpolace kvaternionů

Na jednotkový kvaternion **q** reprezentující rotaci se též můžeme dívat jako na souřadnici na povrchu čtyřrozměrné koule. Jedna z vlastností kvaternionů je ta, že **q** a **-q** reprezentují identickou rotaci, oba dva se ale nacházejí na opačné polokouli jednotkové koule. Při interpolaci se postupně dobíráme k souřadnicím, které leží na oblouku mezi oběma interpolovanými kvaterniony. Existují dvě možnosti, po kterém oblouku se bude pohybovat náš interpolovaný bod – buďto se bude pohybovat po nejkratším nebo po nejdelším možném oblouku mezi dvěma takovými body. Je na nás, abychom se před započetím interpolace rozhodli, který oblouk si vybereme, ale musíme nejprve ověřit, na které polokouli se **q** resp **-q** nachází.

Nejjednodušší metodou pro vyhodnocení příslušnosti kvaternionu ke každé z polokoulí je spočtení úhlu, který mezi sebou dva testované kvaterniony svírají. Protože jsou oba kvaterniony jednotkové, úloha se nám zjednoduší na pouhý skalární součin.

$$\begin{aligned}\cos(\Omega) &= \frac{p_w q_w + p_x q_x + p_y q_y + p_z q_z}{\sqrt{(p_w^2 + p_x^2 + p_y^2 + p_z^2)}\sqrt{(q_w^2 + q_x^2 + q_y^2 + q_z^2)}} \\ \cos(\Omega) &= p_w q_w + p_x q_x + p_y q_y + p_z q_z\end{aligned}$$

Přičemž **p** a **q** jsou dva kvaterniony figurující v interpolaci. Pokud chceme cestovat po nejkratším oblouku, musíme pokaždé ověřit obě dvě varianty **q** a **-q** a zvolit takovou, pro kterou jsme obdrželi ve funkci $\cos(\Omega)$ kladný výsledek. Všimněme

si, že pro určování příslušnosti k polokouli není třeba vyhodnocovat samotné Ω, stačí nám pouze znaménko skalárního součinu. Dále si musíme uvědomit, že existují ještě další dvě vzájemné konfigurace kvaternionů, se kterými se při interpolaci musíme umět vypořádat. První problém nastane, když chceme interpolovat mezi dvěma stejnými kvaterniony. Výsledkem totiž je vždy tatáž hodnota. Pokus o použití operace **slerp** by končil běhovou chybou z důvodu dělení nulou. Podobný problém by nastal, kdybychom se pokoušeli interpolovat mezi kvaterniony **q** a **-q**. V tomto případě tento problém převedeme na problém interpolace po nejkratším oblouku, a tím pádem interpolace dvou stejných kvaternionů. Ta se dá, jak si za chvíli ukážeme, vcelku jednoduše ošetřit.

Operace sférické lineární interpolace je přesně definovaná v internetové encyklopedii Wikipedia [8] a vypadá následovně:

$$\boldsymbol{slerp}(q_0, q_1; t) = \frac{\sin(1-t)\Omega}{\sin\Omega} q_0 + \frac{\sin t\Omega}{\sin\Omega} q_1$$

Hodnotu Ω zjistíme použitím funkce $\arccos(\cos(\Omega))$ na výsledek $\cos(\Omega)$, který jsme obdrželi výše.

Z výše uvedené funkce vyplývá, jaká situace nastane, jestliže je jmenovatel blízký nule. Jsou to přesně ony dvě situace, které jsme si popsali výše.

$$\begin{aligned}
\lim_{\Omega\to 0} \boldsymbol{slerp}(q_0, q_1; t) &= \lim_{\Omega\to 0}\left(\frac{\sin(1-t)\Omega}{\sin\Omega} q_0 + \frac{\sin t\Omega}{\sin\Omega} q_1\right) \\
&= (1-t)q_0 + t q_1 \\
\lim_{\Omega\to 0} \boldsymbol{slerp}(q_0, q_1; t) &= \boldsymbol{lerp}(q_0, q_1; t)
\end{aligned}$$

Pro případ, že je vzdálenost oblouku, po kterém se budeme pohybovat, limitně blízká nule, stává se ze sférické lineární interpolace pouze interpolace lineární. Tento trik se v praxi užívá zcela běžně. Ve chvíli, kdy se $\sin(\Omega)$ nebezpečně přiblíží nule, provedeme místo operace *slerp* operaci *lerp*.

Slerp má pro nás dvě velmi zajímavé vlastnosti. První je ta, že výsledek interpolace cestuje mezi dvěma kvaterniony při konstantním růstu koeficientu **t** konstantní rychlostí. Druhá, avšak nepříjemná vlastnost, je, že operace Slerp není komutativní.

Pokud budeme provádět operaci Slerp v nahodilém pořadí mezi několika různými kvaterniony, neobdržíme stejný výsledek.

3.3. Lineární interpolace kvaternionů

Oproti sférické lineární interpolaci má obyčejná interpolace několik výhod. Nejdůležitější z nich je, že operace *lerp* je na rozdíl od *slerp* komutativní. Můžeme ji tedy provádět v libovolném pořadí, a přesto dostaneme stejný výsledek. Operace *lerp* je navíc oproti s*lerp* velice levná operace; vyhodnocování goniometrických funkcí bývá obvykle poměrně drahé.

$$\boldsymbol{lerp}(q_{0,}q_1;t) = (1-t)q_0+t\,q_1$$

Lineární interpolace však trpí jedním nedostatkem, který se ale nakonec ukáže jako ne zcela zásadní, viz [1]. Tím nedostatkem je nekonstantní rychlost pohybu po oblouku čtyřrozměrné koule při interpolaci. Rotace, kterou tento kvaternion představuje, bude tedy jinak rychlá na začátku, na konci nebo třeba uprostřed interpolace. Důvod je ten, že se interpolovaný bod pohybuje ne po oblouku, ale lineárně po tětivě mezi čtyřrozměrnými body, které kvaternion reprezentují (a tudíž přestává být jednotkovým kvaternionem a je ho třeba pokaždé renormalizovat). Po normalizaci výsledku obdržíme kvaternion, který nakonec prochází po identickém oblouku jako při operaci *slerp*, jen jinou rychlostí. Pro naše potřeby je ale tento fakt v podstatě zanedbatelný, neboť se neprojevuje v takové míře, aby to jakkoliv viditelně rušilo při samotné animaci objektů.

4. Skinning

Skinning je označení procesu, při kterém se aplikují data animované kostry na trojúhelníkový mesh. Ukážeme si, jak to funguje.

Každý bod trojúhelníkového meshe obsahuje seznam kostí, které mají šanci ho ovlivnit, a také váhu kosti v rozsahu $\langle 0;1 \rangle$. Suma vah všech kostí v každém bodě musí být dohromady rovna jedné, abychom zaručili, že se výsledný bod netransformuje na nepředvídanou pozici.

Objekty, na kterých budeme provádět skeletální animace, mají z aplikace, v níž byla data vytvořena, uloženou základní kofiguraci kostry, které se také jinak říká *bind póza*. Pomocí ní totiž můžeme transformovat každý bod trojúhelníkového meshe do souřadného systému, který každá kost reprezentuje. Jakákoliv póza odlišná od bind pózy se nazývá *aktuální póza.*

Základní princip skinningu je v první řadě transformovat každý bod ze souřadného systému objektu do souřadného systému každé kosti, která má na takový bod vliv, a to za pomoci inverzní transformace, kterou daná kost *bind pózy* reprezentuje. Je to z toho důvodu, že je bod v tomto prostoru vzhledem k pohybům dané kosti invariantní - nemění vzhledem ke kosti svou polohu ani orientaci. Kost se pak díky animaci může pohybovat zcela libovolně, ale poloha bodu se v jejím souřadném systému nezmění. Ve chvíli, kdy známe aktuální pózu kostry, transformujeme každý takový bod z lokálního souřadného systému kosti transformací *aktuální pózy*, kterou taková kost reprezentuje. Tím obdržíme opět bod v souřadném systému objektu, ale modifikovaný animovanou kostrou.

$$
\begin{aligned}
B_i &= B_{(i,\,rotation)} B_{(i,\,translation)} \\
P_i &= P_{(i,\,rotation)} P_{(i,\,translation)} \\
v'_j &= \left(\prod_{i=0}^{j} P_i\right)\left(\prod_{i=0}^{j} B_i\right)^{-1} v
\end{aligned}
$$

B jsou kosti obsažené v bind póze, P jsou kosti aktuální pózy. Výsledkem je bod v souřadném systému objektu, který byl transformován do souřadného systému každé jednotlivé kosti, jež na něj měla vliv nejprve transformací bind pózových kostí

a následně transformován zpět do souřadného systému objektu. Tentokrát ale pomocí transformací kostí z aktuální pózy.

Pokud byl bod ovlivněn více nežli jednou kostí, výsledek bude roven váženému součtu všech možných transformací.

$$\bar{v} = \sum_{j=0}^{n} v'_j w_j \quad \sum_j w_j = 1$$

5. Míchání animací

Vlastnost, která přidává skeletálním animacím na flexibilitě, je možnost míchat mezi sebou zcela libovolné animace. Slovo „libovolné“ však nelze brát úplně doslova, je třeba mít na paměti, že tato metoda má své hranice. Pokud bychom mezi sebou chtěli míchat dvě různé animace, jejichž sjednocením je něco, co nedokážeme ani sami rozumně popsat, nelze potom čekat, že výsledek takto smíchaných animací bude k něčemu užitečný. Jako příklad můžeme uvést dvě různé animace lidské postavy, z nichž jedna představuje postavu rychle běžící a druhá postavu, která dělá dřepy. Pokud budou smíchané „ve stejném poměru“, výsledky budou zcela jistě dosti nesmyslné.

Potřebujeme tedy najít algoritmus, který nám dovolí smíchat mezi sebou několik různých póz podle toho, jak moc velký význam dané póze přiřkneme. Síla přehrávané animace může hrát svou roli například ve chvíli, kdy se snažíme přejít od jedné animace ke druhé. Animace pak můžeme přehrávat současně, první s maximální silou a druhou s nulovou. Postupně potom můžeme sílu první animace odebírat a stejnou měrou přidávat animaci druhé.

Samotné míchání animací se provádí tak, že provedeme lineární interpolaci jednotlivých složek kostí kostry. Díky tomu, že používáme lineární interpolaci všech složek se nemusíme obávat o pořadí, v jakém animace do procesu interpolace vstupují. Jediné, co nám zbývá určit, je interpolační koeficient, který může nabývat pro každou kost zcela různých hodnot. Vzhledem k tomu, že animace nemusí nutně pokrývat celou kostru, ale pouze její podčást, musíme provádět výpočet interpolačních koeficientů nezávisle pro každou kost.

Algoritmus výpočtu interpolačních koeficientů je uveden a vysvětlen v plném rozsahu ve článku na internetovém serveru Gamasutra [3]. V podstatě se ale jedná o převod faktorů resp. sil jednotlivých animací do normalizovaného tvaru (na váhy) tak, aby byl součet všech vah na každé kosti roven přesne jedné. V tom případě si totiž můžeme dovolit nad daty provést konvexní kombinaci. Pokud jsme schopni provádět pouze operaci interpolace, dozvíme se ve výše zmíněném článku i postup jak jednotlivé váhy převést na interpolační koeficienty.

6. Optimalizace animačních dat

Asi nejefektivnější metodou paměťové optimalizace animačních dat je jejich aproximace pomocí nějaké funkce.

Podívejme se tedy pořádně na to, jak vypadají data, nad kterými chceme dělat aproximaci. Na množině všech animačních snímků musíme sledovat chování jednotlivých kostí v čase napříč těmito snímky a kosti považovat při aproximaci za samostatná a nezávislá data, kde informace o vztahu k nadřazené kosti nemá vůbec žádný vliv. Ačkoliv praxe bývá obvykle trochu jiná, my můžeme pro začátek v nejjednodušším případě uvažovat pouze kosti, které obsahují jen rotace a translace. Jedná se tedy o dva vektory, z nichž jeden má tři a druhý čtyři složky. Přestaňme na okamžik o kostech uvažovat jako o datové struktuře, která obsahuje rotaci a pozici, vraťme se pro přehlednost k zápisu pomocí stavového vektoru, tentokrát ale pro jednu jedinou kost.

$$\theta = \{x, y, z, \alpha, \beta, \gamma, \delta\}$$

Pokud bychom se soustředili pokaždé pouze na jednu složku stavového vektoru napříč všemi animačními snímky, obdržíme 7 různých samostatných vektorů, jako je následující příklad:

$$\theta_{\alpha} = \{\alpha_{1,} \alpha_{2,} \alpha_{3,} \ldots, \alpha_{n}\}$$

Pro další příklady si zaveďme pomocnou proměnnou t, která nabývá hodnot z intervalu $\langle 0;1 \rangle$, kde hodnota $t=0$ odpovídá prvnímu snímku a hodnota $t=1$ snímku poslednímu. Díky povaze dat se můžeme pokusit najít takovou funkci, která vydá v čase t_i (odpovídajícímu každému snímku **i**) hodnotu, která se co nejvíce blíží původní hodnotě α_i. Navíc hledáme funkci, která bude definovaná na celém intervalu t a bude mít spojité derivace alespoň druhého řádu. Tím zaručíme, že se výsledky vyhodnocování takové funkce nebudou chovat nijak zásadně zle vzhledem k našim potřebám co nejplynulejších animací. Spojité derivace až do druhého řádu nám zaručují vlastnost, že změna rychlosti interpolovaného bodu bude prováděna plynule a ne ve skocích.

Výše v textu jsme se zmiňovali o tom, že se jednotlivé složky vektoru chovají na malém okolí v čase náhodně vybraného snímku přibližně lineárně. Už zde bychom mohli přijít s jednoduchým řešením, že by bylo možné definovat po částech funkci, která je na jednotlivých intervalech $\langle t_{i-1}; t_i \rangle$ lineární. Je ale asi zřejmé, že taková funkce hrubě porušuje výše uvedené předpoklady o spojité druhé derivaci. Na hranicích intervalů je totiž v podstatě jisté, že nebude spojitá ani derivace první.

Co bylo ale na této metodě zaznamenáníhodné, byl onen nápad poskládat výslednou funkci po částech z několika dalších a mnohem jednodušších, jako jsou například kvadratické a kubické polynomy nebo dokonce polynomy vyšších stupňů.

Jedna z takových pěkných funkcí, které se samy nabízí, jsou Bézierovy křivky [6]. Tyto funkce nám již zaručují jistou míru spojitosti na intervalu, na kterém jsou definovány, ale zároveň přinášejí mnoho nevýhod, které se pokusíme krátce shrnout.

Bézierovy křivky jsou definované pomocí takzvaných řídících bodů a bázových funkcí.

$$C(u) = \sum_{i=0}^{n} B_{n,i}(u) P(i)$$

$$B_{n,i}(u) = \frac{n!}{i!(n-i)!} u^i (1-u)^{n-i} = \binom{n}{i} u^i (1-u)^{n-i}$$

$P(i)$ jsou kontrolní body, $B_{n,i}(u)$ jsou bázové funkce Bézierovy křivky a jsou to takzvané Bernsteinovy polynomy. Jak je možné z definice nahlédnout, Bézierova křivka stupně *n* je definována *n+1* kontrolními body a hodnota $C(u)$ na každém z nich závisí na celém svém rozsahu od nuly do jedné a to s sebou nese celou řadu nevýhod. Pokud chceme, aby nás Bézierovy křivky poslouchaly (ohýbaly se pokud možno dostatečně blízko kolem kontrolních bodů), nemohou nabývat velkého stupně. Pokud bychom si Bézierovy křivky zvolili jako stavební kámen pro aproximaci a výslednou aproximační funkci po částech pomocí nich definovali, nutně bychom se nevyhnuli úloze řešit, co se má stát na hranicích jednotlivých křivek, a museli se postavit čelem k problému jejich navazování. Není to neřešitelná úloha, ale každopádně přináší jistou míru nepohodlí, kvůli kterému stojí za to zkoumat dál.

6.1. Non Uniform Non Rational B-Splines

Ve zkratce NUNRBS. B-spline křivka je definovaná velmi podobně jako Bézierova křivka. Zásadně se ale liší v definici bázových funkcí.

$$C(u)=\sum_{i=0}^{n} N_{i,p}(u)P(i)$$

$$\begin{aligned} N_{i,0}(u) &= 1, pokud \quad u_i \le u \le u_{i+1}, \quad jinak\, 0 \\ N_{i,p}(u) &= \frac{u-u_i}{u_{i+p}-u_i} N_{i,p-1}(u) + \frac{u_{i+p+1}-u}{u_{i+p+1}-u_{i+1}} N_{i+1,p-1}(u) \end{aligned}$$

Malé p označuje stupeň křivky, N jsou bázové funkce a u_i jsou hodnoty takzvaného uzlového vektoru (knot vector), což je pomocný vektor, který slouží jako další faktor při určování tvaru křivky. Jsou v něm zaznamenány údaje o časovém rozložení jednotlivých kontrolních bodů vzhledem ke křivce a právě díky tomu, že jednotlivé uzly nejsou v uzlovém vektoru rozmístěny rovnoměrně se těmto křivkám říká neuniformní. Na první pohled to možná není úplně zřejmé, ale počet kontrolních bodů u tohoto typu křivky není závislý na jejím stupni. Čili, přestože budeme například pracovat s kubickou B-spline křivkou (křivka stupně 3), můžeme si pro její definování zvolit naprosto libovolný počet kontrolních bodů (existuje však spodní limit, který říká, že počet kontrolních bodů musí být alespoň stupeň $p + 1$). Další velice pěkná a důležitá vlastnost je, že nejvyšší počet nenulových bázových funkcí je též roven stupni *B-spline* + *1*. V praxi to znamená, že i v situaci, kdy budeme pracovat s křivkou relativně nízkého stupně definovanou na velikém množství kontrolních bodů, v každém okamžiku u budeme pracovat nanejvýše se čtyřmi sousedními kontrolními body. Jediný limitující faktor je vyhodnocování bázových funkcí v bodě u .

Pro rychlejší vyhodnocování funkce $C(u)$ se dá použít De-Boorův algoritmus [7], který funguje na podobném principu jako je vkládání nového bodu do B-spline. O nejvyšší rychlost nám ale v další kapitole nepůjde.

6.2. Optimalizační úloha a řešení

Před samotným zformulováním úlohy si musíme uvědomit, co máme vlastně za vstupy, jaká jsou naše data a co očekáváme na výstupu. Budeme-li se snažit nalézt takovou křivku, která prochází pokud možno co nejpřesněji skrze naše zdrojová data, máme polovinu odpovědi na dosah ruky. Stačí se podívat na definici B-spline a položme ten problém obráceně. Mějme množinu vzorků $S(u)$ a nějaký adekvátní uzlový vektor. Tento budiž dobře definovaný - neobsahuje žádné vícenásobné uzly a splňuje následující podmínku:

$$m = n + p + 1$$

Ta říká, že počet uzlů je roven počtu kontrolních bodů + stupeň křivky + 1.

Množina výsledků $S(u)$ reprezentuje data, která jsme získali rozdělením stavového vektoru každé kosti na jednotlivé složky. Budeme hledat takovou minimální množinu kontrolních bodů B-spline křivky, která bude společně s uzlovým vektorem co nejvěrněji kopírovat zdrojová data. Algoritmus bude hledat řešení s minimální množinou kontrolních bodů, tu postupně doplňovat a zkoumat, zda-li se už konečně „trefil“. V případě, že ne, zvětšíme množinu kontrolních bodů a uzlů o jedna a pokusíme se o celý proces ještě jednou. Poté, co určíme řešení úlohy a obdržíme novou množinu kontrolních bodů, budeme pomocí nich a uzlového vektoru vyhodnocovat funkci C ve stejných časových intervalech, jako byla nasnímána původní data, a budeme měřit kvadratickou odchylku. Pokud tato nepřekročí nějakou pevně stanovenou mez, můžeme nalezenou množinu kontrolních bodů prohlásit za akceptovatelné řešení a algoritmus vrací úspěch.

$$C(u)=\sum_{i=0}^{n} N_{i,p}(u)P_i$$

Pro aproximaci volíme stupeň $p=3$ (kubická B-spline) a spočítáme matici koeficientů N o takových rozměrech, kde počet řádků bude roven počtu vzorků a počet sloupců bude roven počtu kontrolních bodů.

$$\begin{pmatrix} N_{0,0}(u_0) & N_{0,1}(u_0) & \cdots & N_{0,p}(u_0) & 0 & \cdots & 0 \\ \vdots & & & & & & \vdots \\ 0 & \cdots & \cdots & 0 & N_{i-p,p}(u_n) & N_{i-p+1,p}(u_n) & \cdots N_{i,p}(u_n) \end{pmatrix}$$

Obdržíme tak matici, která má podle diagonály pás koeficientů šířky *p+1*.

Index i je odkaz na prvek uzlového vektoru a označuje takový uzel, kterým začíná interval, jehož je u prvkem.

Obecně je úloha formulovaná následovně:

$N\vec{P}=\vec{S}$, kde N je matice koeficientů bázových funkcí $N_{i,p}$ pro jednotlivé vzorky dat, $\vec{P}$ je vektor kontrolních bodů a $\vec{S}$ je vektor vzorků.

Naším úkolem je najít takový vektor $\vec{P}$, který minimalizuje funkci $min|N\vec{P}-\vec{C}|^2$ (reziduální vektor). Máme tedy úlohu, která vede k řešení pomocí metody nejmenších čtverců.

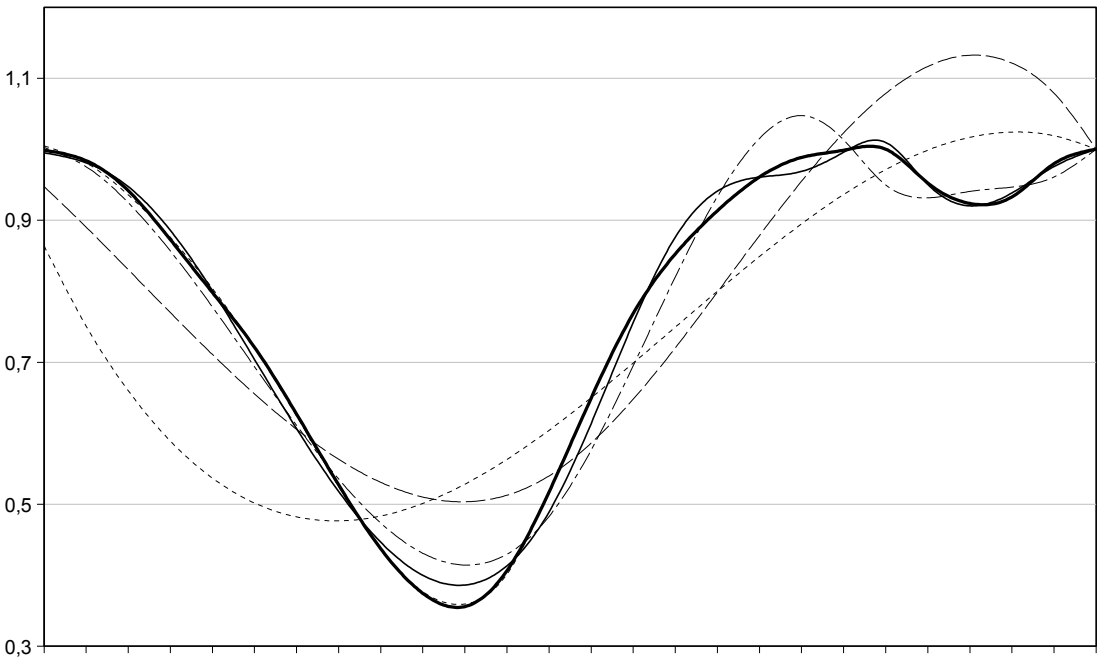
6.3. Implementace

Pro naše potřeby připadaly v úvahu dva algoritmy, které řeší naši úlohu, ve které hledáme vektor $\vec{P}$ s co nejméně prvky takový, který je zároveň řešením rovnice $N\vec{P}=\vec{S}$, platí pro něj podmínka, že ze všech možných $\vec{P}$, které řeší předchozí rovnici tento bude navíc splňovat podmínku $min|N\vec{P}-\vec{S}|^2$ a dotřetice takové $\vec{P}$, pro které bude $|C(u_i)-S_i|^2<T$, kde T je námi specifikovaná úroveň tolerance.

Z dostupného souboru funkcí knihovny GSL (GNU scientific library) jsme vybrali metodu *singular value decompositon* (dále jen *svd)*, která je nejstabilnejší, a nejpřesnější (ale taky zároveň nejpomalejší) a poskytne nám takové řešení, jež splňuje první dvě podmínky, které na celý algoritmus klademe. Splnění třetí podmínky (ověření, zda se chyba mezi vzorkem a funkcí ještě stále vejde pod určitou mez) již musíme ověřit sami. Pokud *svd* nalezne řešení, vrátí nám vektor kontrolních bodů, u kterých lze poměrně snadno ověřit, jak dostatečně přesně aproximuje námi poskytnutý vektor nasmímaných vzorků animačních dat, a to tak, že porovnáme výsledky nově vzniklé křivky s původními vzorky.

6.4. Výsledná data

Následuje ukázka výstupu jednotlivých iterací aproximačního algoritmu pro jednu složku rotačního kvaternionu pro holenní kost u běžící humanoidní postavy.



Aproximace zdrojových dat

Nejvýraznější čarou je označená množina vzorků, což jsou data, jejichž aproximaci hledáme. Jednotlivé přerušované čáry ukazují, jak jsme se k nim blížili za pomocí naší aproximační B-spline křivky po každé iteraci. Poslední iterace již v podstatě přesně kopíruje vzorová data. (Pro nepřehlednost byly vynechány data tří předposledních iterací.)

Vzhledem k tomu, že se kostra animuje typicky pouze pomocí rotací jednotlivých kostí (zřídka kdy se kosti natahují do délky), jsou v podstatě všechny translace aproximované pouze B-spline křivkou která je definovaná na minimálním množství kontrolních bodů a představují pouze přímku. Pro velké animace to znamena zaznamenání hodnou úsporu. Animace postavy které mám k dispozici pro testy dokonce namají translaci použitou ani jednou. Oproti předešlému stavu, kdy byly všechny informace kosti uložené v každém snímku jsme zde i pří použití B-spline

křivek stupně 3 výrazně ušetřili. Původní testovaná animace má přibližně 30 snímků a 70 kostí. Keyframe animace tak zabere 30 * 70 * 3 * 4 = 25300 bajtů (30 snímků, 70 kontrolních bodů po 3 floatech, datový typ float zabere 4 bajty), B-spline aproximace nám zabere pouze 70 * (4 * 3 * 4 + 6 * 4) = 5040 bajtů (70 kostí, na každou kost 4 kontrolní body * 4 floaty * 4 bajty na float + uzlový vektor velikosti 6 floatů * 4 bajty na float). V případě, že by počet snímků kostry vzrostl, rostla by pro keyframe animaci stejnou měrou i datová struktura která jednotlivé translace obsahuje. Aproximovaná data křivkou by však zůstala konstantně veliká.

Rotační data mají trohu jiný charakter – mají tendenci se měnit mnohem častěji, než translace - přesto ale bývá počet změn u ručně animovaných modelů velice malý, jiná situace by pravděpodobně nastala, pokud bychom se snažili aproximovat data pořízená například z motion capture zařízení.

7. Budoucí práce

Při aproximaci animačních dat pomocí křivek jsme použili výhradně B-spline třetího stupně. Dále jsme předpokládali, že data, se kterými pracujeme jsou spojitá (respektive, vzešla ze spojité předlohy) minimálně do druhé derivace. Je ale zřejmé, že pro některé animační kanály by bylo mnohem výhodnější, kdybychom zvolili B-spline nižšího stupně. To proto, aby se urychlilo vyhodnocování bodů v čase při opětovném přehrávání. Bylo by též příjemné, kdyby se tento proces odehrával zcela automaticky bez zásahu člověka. Možný postup by mohl vypadat zhruba tak, že se pokusíme proložit množinu vzorků nejdříve pomocí křivek s co nejnižším možným stupněm a teprve pokud nevyhoví předem zvolené toleranci, nasadit B-spline s vyšším stupněm. Další zajímavou podúlohou je docílit, abychom i v prostředí, kde máme k dispozici pouze B-spline, mohli používat data, která mají nespojitost u nižších derivací (například poskakující míč). A v neposlední řadě též situace, kdy není dokonce spojitá ani první derivace a kdy se objekt místo plynulého pohybu ve scéně pouze přemisťuje z místa na místo.

Optimalizační algoritmus postupuje od nejhrubšího řešení a pokouší se postupným dodefinováváním B-spline hledat přijatelné řešení. Bylo by zajímavé zkusit opačný přístup, kdy začínáme s co největším množstvím kontrolních bodů a postupně tuto množinu redukujeme, dokud nedostaneme řešení, které již nevyhovuje.

Další kroky by bylo možné podniknout na poli kvantizace dat, do menších, případně chytřejších datových typů. Například kvaterniony se dají konvertovat do bitových polí tak, že se dají v relativně slušné přesnosti reprezentovat pouze 32-bitovým číslem [2].

8. Závěr

Nalezené řešení se ukázalo jako dostatěčně funkční a v praxi okamžitě použitelné. Zajímavé je, že tato metoda dává docela věrohodné výsledky i při relativně vysoké hladině tolerance na minimální množině kontrolních bodů. Je to způsobeno tím, že u některých trojrozměrných modelů ani tak nezáleží na tom, jak moc přesně jsme schopni zdrojová data aproximovat, ale postačí, když se jednotlivé kosti hýbou alespoň přibližně správně. Jelikož nejsou všechny objekty ve scéně běžně zobrazované v dostatečné blízkosti kamery a stačí, aby pouze budily dojem, že se v určité vzdálenosti hýbou přibližně dobře, může být tento poznatek vcelku důležitý, pakliže chceme za každou cenu šetřit místem v paměti. Při interpolaci translací jsme navíc dokázali při přehrávání animací vyhladit křivky, po které se kosti mohou pohybovat. Při keyframe animaci jsou totiž translace běžně vyhodnocované pouze pomocí linární interpolace. Pokud by hladké přechody nebyly žádoucí, můžeme jim samozřejmě celkem snadno předejít, důležité ale je, že máme možnost volby.

Přestože se řešení nakonec ukázalo jako poměrně jednoduché, strávil jsem relativně dlouhou dobu při jeho hledání ve slepých nebo alespoň zdánlivě těžko schůdných uličkách při výběru vhodných aproximačních funkcí. Bylo docela krátkozraké, že jsem zpočátku nevěnoval dostatečnou pozornost testům jednotlivých funkcí, hlavně těch, které přímo souvisely s výpočty na B-spline křivkách. Bylo nakonec velmi časově náročné odhalit a odstranit některé chyby, které by pomohly testy hravě identifikovat. Na začátku jsem se také rozhodoval, zda-li budu používat nějaký volně dostupný balík pro řešení metody nejmenších čtverců. Přecenění vlastních schopností v touze implementovat si všechno úplně sám mě též stálo nemalé množství času a nakonec stejně přivedlo zpátky k již hotové implementaci v GNU Scientific Library, která mi navíc umožnila vyzkoušet i jiné metody řešení než Singular Value Decomposition.

Použitá literatura

[1] BLOW, Jonathan. *Understanding Slerp, then not using it*. *The Inner product* [online]. Aktualizováno 26. 2. 2004 [cit. 2006-05-28]. Dostupné z: http://number-none.com/product/Understanding%20Slerp,%20Then%20Not%20Using%20It/.

[2] DeLOURA, Marc. *Game programming Gems 2*. Hingham: Charles River Media, 2000. ISBN: 1-58450-049-2.

[3] EDSAKK, Jenny. Animation Blending: Achieve Inverse kinematics and more. Designer's Notebook, Gamasutra [online]. Publikováno 4.7.2003 [cit. 2006--05-30]. Dostupné z: http://www.gamasutra.com/features/20030704/edsall_01.shtml

[4] WATT, Alan, WATT, Mark. *Advanced Animation and Rendering Techniques: Theory and Practice.* New York: ACM Press, 1992. ISBN: 0-201-54412-1.

[5] WEISSTEIN, Eric W. *Quaternion. Mathworld, A Wolfram Web Resource*. [online]. Aktualizováno 3. 3. 2004 [cit. 2006-05-28]. Dostupné z: http://mathworld.wolfram.com/Quaternion.html.

[6] Wikipedia Contributors. *Bézier curve*. Wikipedia, The Free Encyclopedia [online]. Aktualizováno 30. 5. 2006 [cit. 2006-05-30]. Dostupné z: http://en.wikipedia.org/wiki/Bezier_curve.

[7] Wikipedia Contributors. *De Boor's algorithm. Wikipedia, The Free Encyclopedia* [online]. Aktualizováno 26. 5. 2006 [cit. 2006-05-30]. Dostupné z: http://en.wikipedia.org/wiki/De_Boor_algorithm.

[8] Wikipedia Contributors. *Slerp*. Wikipedia, The Free Encyclopedia [online]. Aktualizováno 8. 5. 2006 [cit. 2006-05-28]. Dostupné z: http://en.wikipedia.org/wiki/Slerp.

Příloha A – knihovna UMD, uživatelská dokumentace

Úvod

Jádrem celého projektu Skeletálních animací je knihovna UMD, jejímž úkolem je správa a zpracování animačních dat. Knihovna UMD je napsána v C++, je nezávislá na procesorové architektuře a je napsaná dostatečně portabilně tak, aby byla kompilovatelná na různých platformách s překladači GCC 3.2 a vyšší, MSVC 7.1 a vyšší nebo kompatibilních překladačích.

Externí závislosti

GSL – GNU Scientific Library. Knihovnu využívá UMD pro řešení úlohy nalezení řešení soustavy lineárních rovnic metodou nejmenších čtverců. Z několika nabízených metod používáme tu nejstabilnější, a to Singular Value Decomposition.

PIX - Prism intermediate format loaders. Knihovna PIX slouží jako nástroj pro načítání animačních a dalších dat z textové podoby. Jejím výstupem je binární datová struktura, se kterou můžeme v UMD velmi lehce pracovat.

P3CORE – Soubor abstrakčních funcí nad operačním systémem a různými implementacemi LIBC. Tato knihovna poskytuje základní funcionalitu ne zcela nepodobnout knihovne STL (Standard Template Library), jako jsou dynamická pole nebo obousměrné spojové seznamy, ale zároveň také funcionalitu zastřešující rozdíly mezi různými operačními systémy, jako jsou například OS Windows, nebo OS Linux.

Pose (umd_pose.h)

Centrálním stavebním kamenem celé knihovny je abstraktní třída *pose_t*. Jejím jediným úkolem je zastřešovat složitější generátory dat pro skeletální animace. Abstraktní interface třídy zahrnuje metody pro práci s časem dané pózy a funkce, pomocí nichž je možné ptát se na data jednotlivých kostí, které pro nás potomek *pose_t* připraví.

Dynamic pose (umd_dynamic_pose.h)

Dynamic pose je jednoduchá specializace třídy *pose_t* a slouží hlavně jako úložiště již zpracovaných dat. Pomocí této pózy, kde jsou uložená již všechna potřebná data pro animaci, jsme nakonec schopni uvést náš trojrozměrný model do pohybu.

Motion (umd_motion.h)

Motion je v podstatě sekvenční póza. Skládá se z jednotlivých snímků animace v čase a je schopná vracet data vázaná ke konkrétnímu snímku v závislosti na nastaveném čase. Jednotlive složky v kostech jsou lineárně interpolovány.

B-spline motion (umd_bspline_motion.h)

B-spline motion je centrální prvek knihovny UMD. Tato třída je chopná převzít data třídy *motion_t*, ta aproximovat pomocí B-spline třetího řádu a výsledky uložit ve vlastní datové struktuře, se kterou je samozřejmě schopna dále pracovat a data popsaná v křivkách na požádání vyhodnocovat a vracet volajícímu.

Skeleton (umd_skeleton.h)

Skeleton je jednoduchá datová struktura, která udržuje informaci o hierarchické závislosti jednotlivých kostí. Pomocí této struktury jsme schopni provádět transformace z a do lokálního prostoru každé kosti, bez znalosti hierarchie kostí v kostře bychom nedokázali nic.

Instance (umd_instance.h)

Instance je abstraktní datová třída, která slouží jako univerzální interface k jednotlivým instancím animovaných objektů. U této třídy zatím není zcela jasné, zda-li budou data animována pomocí skeletální animace či nikoliv.

Skeletal instance (umd_skeletal_instance.h)

Potomek instance, který má již jasně danou úlohu; reprezentovat výsledek, který pochází z několika různých skeletálních animací. Je zodpovědný za řízení času pro jednotlivé struktury typu *motion_t* a také obsahuje strukturu *dynamic_pose_t*, kam si pokaždé ukládá výsledky různých animací. Skeletal instance je třída, se kterou

budeme typicky pracovat, pakliže chceme animovat jakékoliv objekty. Této struktury se můžeme dotazovat na finální konfiguraci jednotlivých kostí a zároveň jsme také schopni data jednotlivých kostí ještě před použitím jakkoliv modifikovat.

Data convertors (umd_convertors.h)

Datové konvertory jsou balíkem pomocných funkcí, které pro komunikaci s knihovnou PIX. Slouží k načtení a předzpracování dat tak, aby byly bez větších rozmyslů a komplikací přímo použitelné v knihovně UMD. Tento modul se také stará např. o korekci nesprávně položených kvaternionů na odvrácených hemisférách v jedné animaci.

Používáme UMD

Knihovna UMD umí pracovat s několika různými zdroji dat. První z nich je datový zdroj s příponou .PIS, který obsahuje „intermediate skeleton“. Jedná se o informace o kostře, její hierarchii a dále také obsahuje data takzvané základní pózy, která se během animace nemění a zůstává konstantní.

Následují animační data, uložená v souboru s příponou .PIA. Tato datová struktura obsahuje jednotlive snímky celé animace, která se vztahuje vždy k nějaké podmnožině kostí celého objektu. (Klidne může pokrýt i celý objekt).

Knihovna UMD umí pracovat jednak s animačními daty zakódovanými do jednotlivých animačních snímků, umí ale též převést a aproximovat tuto datovou strukturu do funkcionálního popisu pomocí B-spline křivek. Postup, kdy chceme animaci převést na křivky je zhruba následující. Vytvoříme skeletální instanci, která vyžaduje jako jediný argument jméno .PIS souboru, ve kterém nalezne popis kostry a dalších dat, uvedených výše. Tato struktura je nám schopná vracet ukazatele na *motion_t*, pakliže jí o to požádáme funkcí, která slouží o načtení jednoho zdroje animací ze souboru .PIA. Tyto zdroje animaci pak můžou být sdíleny různými instancemi.

Pakliže chceme načtenou animaci převést do křivkového popisu, stačí, abychom vytvořili objekt typu *bspline_motion_t* a zavolali jeho inicializační metodu *build_from()* s parametrem typu ukazatel na *motion_t*. Se třídou *bspline_motion_t*

můžeme dále nakládat jako s každou další libovolnou pózou, protože sdílí abstraktní interface.

Abychom mohli přehrávat jednu nebo více animací, musíme tyto zaregistrovat u nějaké třídy typu *skeletal_instance_t*, která bude sloužit jako úložiště pro výsledky smíchaných zaregistrovaných animací. Jednotlivým animacím lze nastavit individuální faktor, který se při míchání použije jako váha animace. Faktory nemají předem daný žádný nárok na velikost. Míchací algoritmus vždy sečte všechny faktory pro každou kost zvlášť dohromady, touto sumou vydělí jednotlivé faktory a tak dosáhne toho, že je suma všech vah rovna jedné a zároveň neklade žádné zvláštní nároky na vyšší vrstvu.

Nezbývá nám nic jiného, než v pravidelných intervalech volat metodu *update()* třídy *instance_t* a dotazovat se jí na výsledky animace. Tyto pak můžeme zcela libovolně používat pro další práci v jiných vrstvách.