

UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Animační systém pro 3D aplikace

Vít Košta

Bakalářská práce

2012

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2011/2012

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Vít Košta**
Osobní číslo: **I09154**
Studijní program: **B2646 Informační technologie**
Studijní obor: **Informační technologie**
Název tématu: **Animační systém pro 3D aplikace**
Zadávající katedra: **Katedra informačních technologií**

Z á s a d y p r o v y p r a c o v á n í :

Tématem práce jsou vybrané metody animací, využívané při vývoji her.

V teoretické části budou popsány základní metody animací, které se nejčastěji používají ve 3D aplikacích. Jedná se především o animaci pohybu po zadané křivce, skeletální animaci, změnu tvarů (geometry morphing) a další. V teoretické části budou rovněž popsány základní principy použití knihovny OpenGL pro vývoj grafických aplikací.

Cílem praktické části je návrh a implementace animačního systému, který tyto metody bude používat. Implementační část bude realizována pomocí programovacího jazyka C++ s využitím OpenGL API. Součástí praktické části bude ukázková aplikace, která bude využívat možnosti navrhnutého animačního systému.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování bakalářské práce: tištěná/elektronická

Seznam odborné literatury:

EBERLY, David H. 3D Game Engine Design : a practical approach to real-time computer graphics. 2nd ed. Amsterdam : Morgan Kaufman, 2007. 1018 s. ISBN 0-12-229063-1, ISBN 978-0-12-229063-3.

ADAMS, Jim. Advanced Animation with DirectX. Boston (Mass.) : PREMIER PRESS, 2003. 452 s. ISBN 1-59200-037-1.

Vedoucí bakalářské práce:

Ing. Petr Veselý

Katedra softwarových technologií

Datum zadání bakalářské práce:

16. prosince 2011

Termín odevzdání bakalářské práce:

11. května 2012



prof. Ing. Simeon Karamazov, Dr.
děkan



L.S.



Ing. Lukáš Čegan, Ph.D.
vedoucí katedry

V Pardubicích dne 30. března 2012

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 11. 5. 2012

Vít Košta

Anotace

Tato práce se zabývá animačními technikami, které se používají především v počítačových hrách a 3D modelovacích a animačních aplikacích. Mezi tyto techniky patří klíčovaná animace, pohyb po křivce, skeletální animace a morfování tvarů. Také je představena knihovna animačního systému, která všechny tyto techniky používá. Součástí práce je i ukázková aplikace, která využívá animační systém a OpenGL API a ukazuje, jak využít GPU pro akceleraci některých technik animace.

Klíčová slova

Animace, klíčovaná, cesty, křivky, skeletální, morfování

Title

Animation system for 3D applications

Annotation

This work is focused on animation techniques mainly used in computer games and 3D applications for artists. These include key-framed animation, curve paths, skeletal animation and mesh morphing. This work also presents animation system library, which uses all these techniques. Also demonstration application is important part of this work. The demo app uses OpenGL API and shows how to use GPU to accelerate some animation techniques.

Keywords

Animation, skeletal, morph, curves, key-frames, paths

Obsah

Seznam zkratk	8
Seznam obrázků	9
Úvod	10
1 Křivky	11
1.1 Lineární interpolace	11
1.2 Polynomiální interpolace	12
1.3 Hermitovská interpolace	13
1.4 Spline křivky	14
1.5 Kubický spline	14
1.6 Cardinal spline	15
1.7 Catmull–Rom spline	15
1.8 Kubická Bézierova křivka	16
1.9 Spojování křivek	17
2 Klíčovaná animace	18
2.1 Využití křivek v klíčované animaci	19
3 Pohyb po cestách	20
3.1 Reparametrizace křivek	20
4 Morfování tvarů	23
5 Skeletální animace	24
5.1 Kostí a jejich hierarchie	24
5.2 Klíčovaná skeletální animace	25
5.3 Transformace objektů pomocí kostí	25
5.4 Kvaterniony	26
5.4.1 Základní vlastnosti	26
5.4.2 Sférická lineární interpolace	27
6 Animační systém	28
6.1 Úvod do systému	28
6.2 Požadavky na animační systém	28
6.3 Základní vlastnosti	28
6.4 Struktura systému	29
6.5 Controller	29
6.6 Controller manager	29
6.7 Graph a path	30
6.8 Graph, path controllers	30
6.9 Míchací controllery	30
6.10 Ostatní	31
6.10.1 Controller Lokat	31
6.10.2 Controller pro spouštění událostí	31
6.10.3 Controller schodové funkce	32
6.11 Mesh morph	32
6.12 Skeletální animace	32
6.12.1 Kostra a kosti	33
6.12.2 Animace	33
6.12.3 Animation manager	34
6.12.4 Animation Mixer	34
6.12.5 Kanály	34
7 Ukázková aplikace	36

7.1	OpenGL API	36
7.2	Základní postup vytvoření OpenGL okna ve Windows	36
7.3	Seznámení s JK2eLite	38
7.3.1	Základní části knihovny JK2eLite	38
7.4	Implementace morfování a skinningu v GLSL	40
7.5	Samotná aplikace	42
	Závěr	43
	Zdroje	44
	Popis formátu souboru JK2	45
8	Popis formátu souboru JK2A	46
9	Popis formátu souboru JK2MORPH	46

Seznam zkratek

1D	jednorozměrný
2D	dvojměrný
3D	tříměrný
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GLEW	wrangler extension library
OpenGL	Open Graphics Library
API	Application Programming Interface
MSVC	Microsoft Visual C++
GLSL	OpenGL Shading Language
UBO	Uniform Buffer Object
TBO	Texture Buffer Object
VBO	Vertex Buffer Object
RAM	Random-access memory
VRAM	Video random-access memory
ORT	Out of Range Type

Seznam obrázků

Obr. 1 2D klíčovaná animace	18
Obr. 2 změna tvaru v závislosti na kladné a záporné váze morph targetu	23
Obr. 3 kosterní hierarchie	24
Obr. 4 transformace pomocí kosti	26
Obr. 5 vazby tříd controllerů	29
Obr. 6 struktura systému skeletální animace	33

Úvod

Tato práce se zabývá animačními technikami a implementací animačního systému, ve kterém jsou všechny zde představené techniky obsaženy. Tento animační systém může být využíván 3D aplikacemi. Je důležité také poznamenat, že to co je zde předkládáno jako animace pro 3D aplikace lze využít i ve 2D aplikacích. Animace jsou velmi důležitou součástí interaktivních aplikací. Jejich hlavní význam je hlavně v tom, že animovaný výstup ať už ze simulace nebo jakékoli jiné aplikace dokáže daleko lépe vysvětlit význam popisovaného děje.

V této práci se používá vlastní definice animace jako „činnost, která způsobuje na základě změny času změnu stavu (např. tvaru, pozice) nějaké entity (např. částice, 3D objektu). Můžeme si všimnout, že v této definici není ani zmínka o tom, jak bude výsledný objekt zobrazen. Ujasnění této definice je velmi důležité, protože pod slovem „animace“ si lze představit mnoho různých věcí. Jednou takovou představou je, že animace je přímo graficky vyjádřena. Tato představa je tedy v kontextu této práce mylná.

Je také vhodné si představit, co je v rámci této práce myšleno 3D aplikacemi. Jedná se především o počítačové hry a modelovací a animační nástroje, které mohou sloužit k vytváření grafických podkladů a animací pro hry. Takže hlavním záměrem této práce je využití animačních metod v rámci počítačových her. V současné době již není možné hry chápat jako něco co stojí mimo společnost – počítačové hry se nyní stávají velmi důležitou formou zábavy. Ale hry nejsou jenom zábava, za hrami se skrývá mnoho pokročilých grafických, simulačních a jiných softwarových technologií. Proto je výhodné se hrami zabývat i z jiných důvodů, než jen ukojení touhy po zábavě.

V počítačových hrách není nutné, aby vše fungovalo podle skutečnosti, ale hlavně musí vše vypadat co nejlépe. Tyto metody tedy využívají všechna data, která jsou již ručně předpřipravena, žádná simulace v nich neprobíhá. Hodí se tedy spíše pro statické scény, ale techniky zde představené mohou být dále rozšířeny, aby a nějakou formu simulace mohly využívat.

Jak již bylo poznamenáno, animační systém se nestará o to jak objekt zobrazit, ale o to jak objektu změnit nějakou jeho charakteristickou vlastnost. O zobrazení se musí starat klientská aplikace, která dokáže všechny tyto změny reflektovat do grafické podoby. V této práci je představena aplikace založená na grafické knihovně OpenGL, programovacím jazyku C++ a platforma pro aplikaci je operační systém Microsoft® Windows®. Programovací jazyk C++ byl zvolen především díky tomu, že programátor má plnou kontrolu nad tím co jak bude probíhat.

OpenGL je velmi výkonná moderní grafická knihovna. Práce s ní je velmi jednoduchá a proto je vhodné začít cestu poznávání 3D grafiky právě s touto knihovnou. Jedním z cílů je také ukázat jak využít GPU k akceleraci některých animačních technik. Proto, aby byl animační systém nezávislý na použité grafické knihovně, nemůže přímo obsahovat podporu využití GPU. Toto vše zůstává na implementaci klientskou aplikací. OpenGL přímo využívá svůj vlastní vyšší programovací jazyk (GLSL), pomocí kterého lze naprogramovat nové programy, které budou spouštěny na grafické kartě. Využitím těchto možností lze dosáhnout značného zvýšení rychlosti aplikace, což je pro počítačové hry kritická vlastnost, kterou nelze opomenout.

1 Křivky

Křivky hrají jak v počítačové grafice, tak v animaci velmi důležitou roli. Především díky tomu, že existuje omezená paměť, do které není možné, ani vhodné, vložit velmi mnoho bodů, které by vyjadřovali požadovaný tvar. Z tohoto důvodu je nutné mít k dispozici metody, kterými lze tyto pozice dopočítávat pomocí matematických funkcí.

Křivky lze vyjádřit několika způsoby – explicitně (pomocí matematické funkce, $y = f(x)$), implicitně (tvar $F(x, y) = 0$) a parametricky. V počítačové grafice a animaci obecně je velmi výhodné používat parametrické vyjádření křivek, především díky tomu, že umožňuje postupný výpočet křivek a také protože u složitějších křivek nelze snadno najít matematickou funkci, která by ji popisovala. Dvourozměrná parametrická křivka (může být i více rozměrná) je obecně vyjádřena jako

$$P(t) = [x(t), y(t)].$$

Důležitou vlastností, které u křivek sledujeme, je také tečný vektor. Tento vektor lze získat první derivací podle parametru t . V případě parametrických křivek opravdu není derivací bod, ale vektor. Směr tohoto vektoru udává směr tečny a jeho velikost udává rychlost v tomto bodě.

Dále budou představeny nejzákladnější křivky, které lze použít ať už pro interpolaci (aproximaci) hodnot animace nebo vytváření různých tvarů a cest.

1.1 Lineární interpolace

Lineární interpolace je jednou z nejpoužívanějších metod interpolace. I když se může zdát, že se nic v přírodě nepohybuje striktně lineárně, v počítačové animaci má toto zjednodušení svoje místo. Pro výpočet je potřeba využívat pouze operace sčítání a násobení, výpočet této interpolace je proto velmi rychlý. Právě proto síla lineární interpolace spočívá v jednoduchosti.

Parametrická definice je dána

$$X = A + (B - A) \cdot t$$

nebo ve formě bez vektoru

$$X = A \cdot (1 - t) + B \cdot t,$$

kde A , B jsou body, kterými chceme interpolovat, $B - A$ je vektor, který směřuje od bodu A do bodu B a t je parametr, který udává jak „blízko“ jsme u bodu A nebo B . Parametr t může nabývat hodnot od $-\infty$ do ∞ , kdy se interpoluje na celé přímce, kterou body A a B tvoří. Častější je omezení parametru t na interval 0 až 1 , což zajistí, že se bude interpolovat pouze na úsečce AB .

Najít směr i velikost tečného vektoru, je v případě lineární interpolace velmi snadné, jedná se přímo o směr a velikost vektoru $B - A$. Využití první derivace toto tvrzení potvrzuje. V každém bodu úsečky má stejnou velikost i stejný směr, z čehož plyne, že rychlost je konstantní a zrychlení je nulové.

Lineárními segmenty lze dokonce aproximovat i daleko složitější křivky. Tato metoda je, ale na vytváření křivek ručně velmi pracná, a proto je nutné využít křivky vyšších řádů, které s tímto problémem velmi pomohou.

1.2 Polynomiální interpolace

Polynom je definován

$$P_n(x) = \sum_{i=0}^n a_i \cdot x^i = a_0 + a_1 \cdot x + \dots + a_{n-1} \cdot x^{n-1}.$$

$P(x)$ určuje polynom řádu n , a_i koeficienty polynomu. Z této definice tedy také vyplývá, že lineární interpolace je podmnožinou polynomiální, kvůli její důležitosti je ale vhodné se jí zabývat odděleně. Proto se již dále budeme zabývat pouze polynomy vyšších řádů.

Danými body lze proložit polynom, který bude všemi těmito body procházet. Mezi nejčastěji používané polynomy, které lze k tomuto účelu použít, je kubický polynom. Taková kubická interpolace je parametricky vyjádřena jako

$$P(t) = (t^3, t^2, t, 1) \cdot N \cdot \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{pmatrix} = (t^3, t^2, t, 1) \cdot N \cdot P,$$

kde N je bazová matice, P je vektor bodů, kterými je polynom proložen.

Kdyby bylo dáno n bodů museli bychom řešit n rovnic o n neznámých, naštěstí ale existuje snazší způsob jak dosáhnout stejného výsledku – Lagrangeův polynom. Lagrangeův polynom byl definován jako

$$P_n(x) = \sum_{i=0}^n y_i \cdot \frac{\prod_{i \neq j} (x - x_j)}{\prod_{i \neq j} (x_i - x_j)}$$

Aby byl výpočet korektní, musejí být body seřazeny podle hodnoty x souřadnice od nejmenší po největší a nesmí existovat žádné body, které by měly stejnou x souřadnici.

Z této definice, lze vyjádřit parametrické vyjádření polynomu řádu n pro $n + 1$ bodů, kde $P(t_0) = P_0$, $P(t_1) = P_1$, ..., $P(t_n) = P_n$ a $0 \leq t_i \leq 1$ je dáno vztahem

$$P(t) = \sum_{i=0}^n P_i \cdot \frac{\prod_{j \neq i} (t - t_j)}{\prod_{j \neq i} (t_i - t_j)}$$

Značnou nevýhodou při používání polynomiální interpolace je velké rozkmitání křivky vyšších řádů nebo pokud jsou interpolované body příliš blízko u sebe. Kvůli tomu je vhodné využívat křivky, u kterých je řád polynomu co nejnižší. Nyní je tedy již jasné, proč je kubický polynom tak využíván. Další problém je malá kontrola nad tvarem křivky. Tvar lze měnit jen přidáváním nových uzlových bodů, což je velmi nepohodlné.

1.3 Hermitovská interpolace

Při použití lineární interpolace nebo polynomiální nemáme velkou kontrolu nad tvarem křivek z nich vytvořených. Jak již bylo uvedeno, abychom mohli ovlivnit tvar těchto křivek musíme přidávat stále nové body, a ani to nezajistí, že dostaneme požadovaný tvar. Proto byly vytvořeny křivky, které toto nevyžadují, a jejich tvar lze řídit jinými způsoby. Jednou z takovýchto křivek je právě Hermitovská křivka.

Je interaktivní [1], což znamená, že lze měnit tvar křivky, aniž bychom museli měnit polohu bodů, mezi kterými interpolujeme. Toto nelze tvrdit o lineární ani polynomiální interpolaci.

Kubická Hermitovská křivka je definována pomocí 2 řídicích bodů a 2 tečen. Interpolace z bodu P_0 do bodu P_1 je řízena velikostí a směrem tečen. Čím je tečný vektor delším, tím větší tendenci má křivka pokračovat v tom směru. Parametricky je definována jako každá jiná parametrická křivka pomocí

$$P(t) = (t^3, t^2, t, 1) \cdot H \cdot \begin{pmatrix} P_0 \\ P_1 \\ P'_0 \\ P'_1 \end{pmatrix} = (t^3, t^2, t, 1) \cdot \begin{pmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} P_0 \\ P_1 \\ P'_0 \\ P'_1 \end{pmatrix},$$

kde H je báze Hermitovské interpolace, P_0 , P_1 body kterými chceme interpolovat, P'_0 tečna procházející bodem P_0 a P'_1 tečna procházející bodem P_1 . Lze dokázat, že derivace v bodech P_0 a P_1 je rovna P'_0 a P'_1 .

1.4 Spline křivky

Spline křivky řeší problém s rozkmitáním křivky při použití Lagrangeova polynomu. K tomu je využíváno několik spojených segmentů jednoduchých křivek, zpravidla kubických. Definice spline křivek jak je uvedena v [1]:

„Spline je soustava polynomů stupně k takových, že jsou hladce spojeny v uzlových bodech. V každém uzlovém bodu jsou dva polynomy spojeny tak, že jejich první derivace mají stejnou hodnotu. Je také nutné, aby všechny derivace až do $k - 1$ derivace měly stejnou hodnotu v daném bodě.“.

Celá kategorie spline křivek se dá rozdělit do dvou kategorií

- aproximační,
- interpolační.

Podrobněji budou představeny jen některé spline křivky, ale existuje mnoho dalších např. B-spline (aproximační), NURBS (aproximační), Kochanek–Bartels spline a jiné.

1.5 Kubický spline

Skládá se z $n - 1$ Hermitovských segmentů, které jsou hladce spojeny. Pokud je dáno n bodů, vytvoříme si $n - 1$ dvojic $(P_0, P_1), (P_1, P_2), \dots, (P_{n-2}, P_{n-1})$, na které použijeme Hermitovskou interpolaci. Proto, abychom mohli takovou křivku sestavit, je nutné k zadání n bodů přidat tečný vektor v bodě P_0 a bodě P_{n-1} . Všechny ostatní tečné vektory nutné pro sestavení Hermitovských segmentů, budou dopočítány podle definice spline křivek. Těchto neznámých tečen je pouze $n - 2$, protože je nutné, aby si tečné vektory odpovídali jak svou velikostí, tak svým směrem ve spojích jednotlivých segmentů. Použitím této metody sestavení křivky dosáhneme toho, že bude interaktivní, tvar lze měnit pomocí tečných vektorů na začátku a na konci křivky. Přesný postup výpočtu vnitřních tečných vektorů, lze snadno odvodit nebo lze použít odvození ukázané v [1].

Je také možné zvolit druhé derivace v počátečním a koncovém bodě tak, aby se rovnaly nule, tím odpadne nutnost zadávat tečné vektory a zajistí malé zakřivení celé nové křivky. Postup výpočtu je poté obdobný jako v první variantě.

1.6 Cardinal spline

Cardinal spline vychází opět z Hermitovské interpolace a odstraňuje některé nevýhody kubické spline, jako například nutnost řešit velké množství rovnic, které je závislé na počtu uzlových bodů. Vše ovšem za cenu, že si druhé derivace v každém bodu nejsou rovny. Tato křivka tedy nesplňuje definici spline křivek a proto se nejedná o spline křivku i přesto se nazývá „spline“, což může být matoucí. Tento název ji zůstal hlavně z důvodu své velké podobnosti se spline křivkami.

Pokud je zadáno n bodů, kterými má křivka procházet, je nejprve nutné, je setřídít do $n - 3$ čtveřic $[P_0, P_1, P_2, P_3], [P_1, P_2, P_3, P_4], \dots, [P_{n-4}, P_{n-3}, P_{n-2}, P_{n-1}]$. Poté se projdou všechny tyto dvojice a sestaví se segmenty křivky tak, že P_{i+1}, P_{i+2} se stanou počátečními body příslušného segmentu, a tečné vektory v těchto bodech budou dány jako $s \cdot (P_{i+2} - P_i)$ a $s \cdot (P_{i+3} - P_{i+1})$, kde $0 \leq i \leq n - 1$. Tento výběr tečných vektorů zajistí, že křivka bude spojena hladce. Jeden segment Cardinal spline je tedy parametricky definována jako

$$P(t) = (t^3, t^2, t, 1) \cdot \begin{pmatrix} -s & 2-s & s-2 & s \\ 2s & s-3 & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} P_i \\ P_{i+1} \\ P_{i+2} \\ P_{i+3} \end{pmatrix},$$

kde parametr s ovládá délku tečných vektorů, i je zadáno stejně jako v předešlém příkladu.

1.7 Catmull–Rom spline

Jedna z nejpožívanějších křivek v počítačové animaci vůbec, hlavně díky svým vlastnostem a tvarem. Používá se především pro pohyb objektů po definované cestě. Neprochází počátečním a koncovým bodem, ale tyto body mají na tvar křivky vliv. Tento jev lze odstranit zdvojením počátečního a koncového bodu. Jedná se o speciální případ Cardinal spline, kdy je s rovno 0.5.

Parametricky je tedy definována

$$P(t) = (t^3, t^2, t, 1) \cdot \begin{pmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} P_i \\ P_{i+1} \\ P_{i+2} \\ P_{i+3} \end{pmatrix},$$

kde $0 \leq i \leq n - 1$ (n = počet bodů).

První derivace je definována

$$P'(t) = (t^3, t^2, t, 1) \cdot \begin{pmatrix} 0 & 0 & 0 & 0 \\ -1.5 & 3 & -3 & 1.5 \\ 2 & -5 & 4 & -1 \\ -0.5 & 0 & 0.5 & 0 \end{pmatrix} \cdot \begin{pmatrix} P_i \\ P_{i+1} \\ P_{i+2} \\ P_{i+3} \end{pmatrix},$$

kde i má stejné vlastnosti jako u parametrické definice.

Z čehož plyne, že rychlost není konstantní a toto přináší některé problémy, které budou představeny v části zabývající se cestami.

1.8 Kubická Bézierova křivka

Všechny doposud představené křivky, procházely všemi kontrolními body a jednalo tedy o interpolaci. Bézierova křivka neprochází všemi body a je tedy aproximační. Obecně je tato křivka definována

$$P(t) = \sum_{i=0}^n P_i \cdot B_i^n(t), \quad 0 \leq t \leq 1,$$

kde P_i jsou body, které tvoří obalový polygon křivky a $B_i^n(t)$ jsou Bernsteinovy polynomy, dané vztahem

$$B_i^n(t) = \binom{n}{i} \cdot t^i \cdot (1-t)^{n-i}$$

Opět je její kubická varianta, díky své jednoduchosti, velmi používána. Je parametricky definována jako

$$P(t) = (t^3, t^2, t, 1) \cdot \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix}.$$

Body P_1 a P_2 můžeme označit jako kontrolní, protože jimi křivka neprochází. Protože bude dále třeba používat první derivaci, je vhodné ji ukázat

$$P'(t) = (t^3, t^2, t, 1) \cdot \begin{pmatrix} 0 & 0 & 0 & 0 \\ 3 & -9 & 9 & -3 \\ -6 & 12 & -6 & 0 \\ 3 & -3 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix}.$$

1.9 Spojování křivek

Křivky se často skládají ze segmentů a proto je nutné mít přehled o tom, jakým způsobem lze takové segmenty spojovat. Jednou z možností jak spoje popsat je tzv. parametrická spojitost. Rozlišuje se několik typů této spojitosti [1]:

- C^0 – koncový bod segmentu a počáteční bod segmentu je stejný,
- C^1 – první derivace jsou ve všech bodech spojitě,
- C^2 – druhé derivace jsou ve všech bodech spojitě,
- C^n – n -té derivace jsou ve všech bodech spojitě.

Jinak řečeno, C^0 znamená, že se pozice nezmění skokem, C^1 směr a velikost rychlost, C^2 zrychlení. Pokud je křivka C^n spojitá, zjevně splňuje spojitost od C^0 po C^n . Křivka se považuje za hladkou, pokud splňuje alespoň C^1 , čím vyšší C spojitost tím je křivka hladší.

Lineární segmenty lze spojovat tak, aby splňovali všechny typy C spojitosti velmi snadno, avšak dosáhne se pouze toho, že budou všechny segmenty na jedné přímce, což nemá žádné opodstatnění. U spline křivek, které jsou stupně k , je již z definice spojitost jednotlivých segmentů zaručena až do C^{k-1} . Spojování kubické Bézierovy křivky je o trochu zajímavější. Pokud chceme, aby byly dvě kubické Bézierovy křivky hladce spojitě je nutné, aby koncový bod první křivky byl shodný s počátečním bodem druhé křivky a musí být středem úsečky mezi posledním kontrolním bodem první křivky a prvním kontrolním bodem druhé křivky.

Protože jsou typy spojení C velmi restriktivní, např. z geometrického hlediska vypadá napojení tak, že tečné vektory mají stejný směr, ale mají různou velikost, tedy není C^1 spojitá, i když se na první pohled zdá, že je. Proto existuje další kategorizace spojitosti, tzv. geometrické [1]:

- G^0 – koncový bod segmentu a počáteční bod segmentu je stejný,
- G^1 – první derivace mají v bodě spojení stejný směr,
- G^2 – druhé derivace mají v bodě spojení stejný směr,
- G^n – n -té derivace mají v bodě spojení stejný směr.

2 Klíčovaná animace

Klíčovaná animace je jednou z nezákladnějších animačních metod, které se pro animaci používají. Spočívá ve vytvoření značek na časové ose, kde každá značka představuje stav objektu v označeném čase. Tyto stavy mohou představovat změnu polohy, tvaru, obecně změnu nějaké vlastnosti, která daný objekt popisuje. S dostatkem představivosti, lze považovat za objekt obrázek a jako změnu jeho „tvaru“ nějaký jiný obrázek, který může být ve vztahu s předešlým obrázkem (obr. 1). Právě ve vztahu s touto představou se lze také setkat s výrazem key-frame, kde se nepoužívá přímo časová osa, ale osa snímků (frame – toto označení vychází z políčka filmu). Čas tohoto snímku je poté vypočítán podle další hodnoty, která udává počet snímků za sekundu (frames per second – FPS). Mezi dvěma snímky je tedy konstantní časová vzdálenost.



Obr. 1 2D klíčovaná animace

Samotná animace je založena na postupném procházení klíčů a měnění stavů objektů. Podle aktuálního času se najdou dva sousední klíče, mezi kterými právě aktuální čas leží. První klíč, který má svůj čas větší než aktuální čas bude nazván jako následující klíč a nejbližší klíč s nižším časem bude nazván jako aktuální klíč (všechny klíče jsou seřazeny podle času).

Existuje zde i několik problémů, které s sebou klíčovaná animace přináší. Pokud není dostatek klíčů, z čehož vyplývá, že jsou od sebe klíče v čase velmi vzdálené, jsou vidět ostré skoky mezi stavy objektů a taková animace nevypadá dobře. Řešení tohoto by se mohlo nabízet: Proč tedy nevytvářet všechny klíče? Ano, u animace obrázků není jiná možnost než mít každý snímek jiný obrázek, ale pro změny např. pozic by bylo třeba neuvěřitelně mnoho klíčů. Vytvářet tolik klíčů ručně by bylo velmi pracné, paměťově velmi náročné a v praxi takřka nepoužitelné. Možnost jak se tomuto problému vyhnout, je použití interpolace a dopočítávat podle nějaké funkce body mezi jednotlivými klíči – využít tedy již známé parametrické křivky.

2.1 Využití křivek v klíčované animaci

Je si třeba uvědomit jednu důležitou vlastnost klíčované animace – čas roste lineárně. Nejčastěji používanými křivkami pro dopočítávání hodnot klíčů jsou lineární a kubické Bézierovy. Jako body (klíče), které se používají pro interpolaci, se použijí dvojice čas a hodnota – tedy např. bod $P = [time, value]$.

U lineární interpolace je situace vcelku snadná, čas roste lineárně (osa x) i parametr roste lineárně, není třeba nic upravovat, stačí jen aktuální čas převést na interní hodnotu parametru segmentu, pomocí jednoduchého vzorce

$$t_{local} = \frac{time_{curr} - B_{time}}{B_{time} - A_{time}}, \quad 0 \leq t_{local} \leq 1, \quad (1.1)$$

kde $time_{curr}$ je aktuální čas, B_{time} je čas následujícího klíče, A_{time} je čas současného klíče.

Při použití kubických Bézierových křivek je situace poněkud složitější. Křivka jednoho segmentu mezi současným (A) a následujícím (B) klíčem je definována jako:

$$P(t) = (t^3, t^2, t, 1) \cdot \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} A \\ C_1 \\ C_2 \\ B \end{pmatrix},$$

kde C_1 , C_2 jsou kontrolní body, které ale nelze považovat za klíče, i když mají stejné vlastnosti, tedy jsou definovány pomocí dvojice čas a hodnota. Pokud nejsou hodnoty x souřadnice (času) kontrolních bodů uzavřeny mezi časy klíčů A a B, mohlo by se stát, že by jednomu času odpovídalo více hodnot. Na tento problém si je potřeba při vytváření Bézierových křivek dát pozor.

Fakt, že čas roste lineárně, přináší ve využití této křivky několik problémů. Za parametr křivky t nelze použít přímo čas, tak jak bylo ukázáno v případě lineární interpolace. Ale i přesto lze využít parametrické vyjádření. Protože známe čas, který v našem případě značí x pozici bodu na křivce. Z tohoto vztahu může dopočítat hodnotu parametru pomocí řešení rovnice. Lze například využít metodu tečen (metoda Newton–Raphson).

Je tedy dána kubická Bézierova křivka a aktuální čas

$$x(t) = (t^3, t^2, t, 1) \cdot \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} A_{time} \\ C_{1x} \\ C_{2x} \\ B_{time} \end{pmatrix} = time_{curr},$$

kde A_{time} , B_{time} jsou časy současného a následujícího klíče, C_{1x} , C_{2x} jsou „časy“ kontrolních bodů.

Jedinou neznámou je parametr t , pro jeho výpočet použijeme metodu tečen, jako vhodný kandidát na počáteční hodnotu se jeví hodnota parametru, odvozená ve vztahu (1.1)

$$t_1 = t_{local} - \frac{x(t_{local})}{x'(t_{local})}$$

a pokračuje se dalšími hodnotami

$$t_{i+1} = t_i - \frac{x(t_i)}{x'(t_i)}.$$

Opakováním můžeme hodnotu zpřesňovat. Obvykle se používají dvě podmínky, kdy lze již hledání ukončit – použije se maximální počet iterací nebo pokud je současná hodnota parametru menší než nějaká zvolená hodnota. Tímto způsobem dostaneme parametr t , který zhruba odpovídá aktuální hodnotě času (na ose x) a použijeme ho pro výpočet hodnoty klíče. Numerické řešení rovnic nemusí být ideální, a pokud nebude postačovat rychlost (ve většině případů bude), je třeba nalézt jiné způsoby jak tento problém řešit.

3 Pohyb po cestách

Pokud je u klíčované animace klíčovou hodnotou čas, ve kterém se má v daném bodě částice nacházet, u cest je touto hodnotou pouze pozice. K čemu takto definované cesty použít? Použití pro herní aplikace je velmi omezené. Většinou je vhodné tento typ cest použít pro pohyb kamery, tedy v předem připravené animaci, do které není vhodné interaktivně zasahovat. Nevhodný příklad použití se sám nabízí, např. pohyb vozidla po cestě. Problém spočívá v tom, že vozidlo potřebuje reagovat na dynamické překážky a upravovat svou rychlost podle tvaru křivek a působení sil. Takovouto simulaci, zde prezentované metody nenabízí.

3.1 Reparametrizace křivek

Pokud použijeme jednoduché přičítání parametru t pomocí stejně dlouhých kroků, zjistíme, že na pohybu částice po cestě, zdánlivě konstantní rychlostí, něco nesedí. Je jasné vidět, že částice na kratších úsecích zpomaluje a na dlouhých velmi zrychluje. Toto je způsobeno tím, že rychlost objektu je ovlivněna velikostí tečného vektoru křivky v daném bodě (jedná se o již zmiňovanou rychlost křivky). Aby byl pohyb objektu opravdu konstantní, je nutné, aby velikost tečného vektoru byla nezávislá na parametru t – tedy konstantní. Protože se většinou cesty skládají z více úseků, je také nutné zajistit to, aby tato velikost vektoru byla ve všech segmentech stejně velká.

Řešení tohoto problému pro lineární interpolaci je snadné, spočítáme délku úseku a místo toho, abychom použili přičítání parametru t u všech segmentů pomocí stejně dlouhých

kroků, použijeme velikost kroku, která bude závislá na velikosti daného úseku. Za parametr t tedy zvolíme:

$$t = \frac{s - l_1}{l_2 - l_1},$$

kde s je vzdálenost od počátku křivky, l_1 je vzdálenost od počátku křivky prvního uzlového bodu, která je větší nebo rovna s , l_2 je vzdálenost od počátku křivky prvního uzlového bodu, která je menší nebo rovno s . Tímto způsobem získáme parametr t , který bude $0 \leq t \leq 1$.

Protože toto řešení funguje dobře pouze u lineárních křivek, je nutné přijít s obecnější formulací řešení tohoto problému. Takové řešení je uvedeno v [8]. V případě křivek vyšších řádů je nutné křivku reparametrizovat, nebo-li najít takovou funkci, která bude závislá na délce oblouku (arc-length), místo parametru t . Snadno můžeme najít funkci, která bude závislá na parametru t a bude popisovat vzdálenost jednoho uzlového bodu segmentu od druhého:

$$s = g(t) = \int_0^t \sqrt{(x'(t))^2 + (y'(t))^2 + (z'(t))^2} dt .()$$

Naším cílem je, ale získat inverzní funkci, která bude mít za parametr vzdálenost s

$$t = g^{-1}(s) .$$

Poté stačí tento parametr využít a získat bod X , který odpovídá vzdálenosti s na křivce

$$P(g^{-1}(s)) = X .$$

Protože nalezení $g^{-1}(s)$ v uzavřené formě je takřka nemožné, je nutné využít numerické metody. Hledáme tedy takové t , pro které bude platit

$$0 = g(t) - s ,$$

Jedná se o rovnici, kterou může opět numericky řešit s pomocí již známé Newton-Rapson metody a můžeme najít parametr t , který odpovídá této vzdálenosti od počátku křivky. Protože obecně neexistuje analytické řešení integrálu ve funkci $g(t)$, je třeba využít numerického integrování (např. Rombergova metoda [9]). Derivace této funkce je současně velikostí rychlosti křivky v daném bodě. Za počáteční odhad parametru t (t_0) můžeme zvolit stejnou hodnotu jako v případě lineární cesty. Vztah je tedy dán

$$t_{i+1} = t_i - \frac{g(t_i) - s}{g'(t_i)} .$$

Toto řešení lze provádět během každého volání, avšak není to vhodné. Je možné proložit výchozí křivky jinou, která ji co nejvíce odpovídá, ale má mezi všemi svými úseky stejnou vzdálenost.

4 Morfování tvarů

Ve 3D grafice nemůžeme vystačit jen s jednotlivými pozicemi a 2D obrázky. Musíme si zavést shluky bodů spojených do polygonu, které poté nějakým způsobem rasterizujeme. S tímto přichází nutnost mít nějakou možnost takové objekty animovat. A není nutné vůbec přicházet s něčím novým. Stále vystačíme s interpolacemi a body v prostoru (vertexy).

Nejprimitivnější metodou, jak animovat objekty, je interpolovat mezi jedním tvarem na jiný. Tato technika se nazývá morfování meshu (tvarů), lze nalézt i jiné názvy, která ale stále vyjadřují totéž – blend shapes, morph targets. V počítačových hrách byla tato technika hojně využívána zhruba do roku 2000 (a stále je). Mezi hlavní výhody, oproti jiné metodě, která bude představena později, patří rychlost výpočtů – stačí jen lineárně interpolovat mezi pozicemi vertexů.

Vytvořit morfování tvarů je velmi snadné, stačí vzít nějaký model, tento označit jako základní tvar (base mesh) a poté nějakým způsobem měnit pozice vertexů modelu a tak vytvářet cílové meshe (target mesh), do kterých chceme transformovat base mesh. Je nutné si uvědomit, že base a daný target mesh musejí mít shodný počet vertexů.

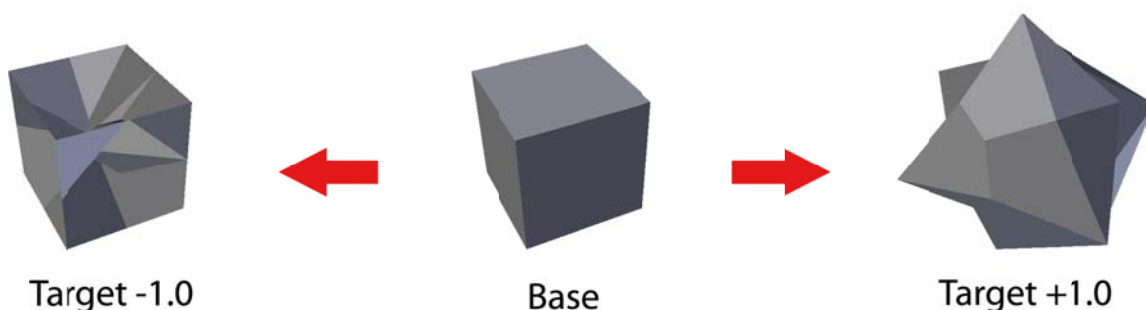
Jakmile jsou všechny tyto tvary vytvořeny, již nic nebrání tomu je využít pro animaci. Vertexy target meshů je vhodné ukládat jako relativní pozice vzhledem k base mesh.

$$offset = vertex_{base} - vertex_{target}$$

Offset udává změnu pozice vertexu, který se nachází v target mesh, oproti odpovídajícímu vertexu v base meshe. Dále target meshům přidělíme váhy, které budou udávat, jak hodně se má příslušný target mesh projevit na výsledném tvaru. Váhy mohou být i záporné, čímž získáme přesný opačný tvar, než měl původně vytvořený target mesh. Pokud je tedy dáno n target meshů, výslednou pozici vypočítáme jako

$$vertex = vertex_{base} + \sum_{i=0}^n w_i \cdot offset_i$$

kde w_i je váha target meshe i , $offset_i$ je vertex, který odpovídá $vertex_{base}$ v target mesh i .



Obr. 2 změna tvaru v závislosti na kladné a záporné váze morph targetu

Všechny váhy lze přidat do klíčované animace a poté jimi interpolovat a získat tak jednoduchou animaci.

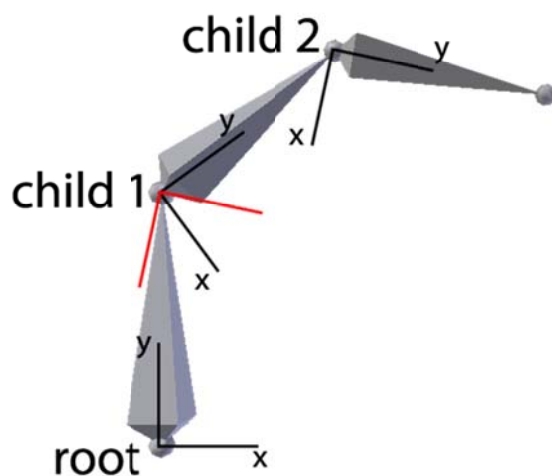
5 Skeletální animace

Používání morfování objektů, které bylo představeno v minulé části, má spoustu nevýhod. Pokud chceme animovat hrubý pohyb živočichů (tzn. pohyb větších částí těla), morfování objektů značně selhává. Nutnost vytvářet takové animace pouze ručně je velmi zdoluhavé a pracné. Přesně tento nedostatek se snaží odstranit skeletální animace. Jejich použití je velmi intuitivní a spočívá ve vytvoření kosterní hierarchie, podle které se budou transformovat jednotlivé části objektu. Pro animátora se poté práce sníží jen na rotace kostí, kterých se v modelu nachází výrazně méně než všech vertexů, s kterými by musel pohybovat v případě použití morfování objektů. Existují samozřejmě i případy, kde skeletální animace také selže, a je nutné použít morfování. Mezi tyto případy patří především animace výrazů obličeje nebo nějakých měkkých objektů. Samozřejmě lze obě techniky spojit a získat tak to nejlepší z obou technik – např. celé lidské tělo se bude animovat pomocí kostí, část obličeje také, a zbytek již bude tvořen morph targety. Použití skeletálních animací je možné jak ve 2D tak ve 3D.

5.1 Kostí a jejich hierarchie

Každá kostra obsahuje kosti (bone) a klouby (joint), dále musí obsahovat kořenovou kost, ze které všechny ostatní kosti vycházejí. Pomocí kloubů se na sebe napojují a platí, že každá kost kromě root může mít nejvíce jednoho rodiče.

Všechny kosti jsou popsány relativní maticí, která určuje jejich relativní transformaci vzhledem ke své rodičovské kosti, dále absolutní maticí, která určuje transformaci vzhledem ke světu. Výchozí absolutní transformační matice kosti se nazývá bind pose.



Obr. 3 kosterní hierarchie

Pokud změníme transformaci některé kosti, která má k sobě napojeny jiné kosti, je třeba upravit všechny absolutní matice kostí, které se nacházejí v hierarchii pod ní (kosti, které

mají za rodiče nějakou kost, která má jako rodiče právě upravenou kost). Toho lze dosáhnout jednoduchým rekurzivním voláním, které začne se změněnou kostí a pokračuje dále v hierarchii. Z tohoto důvodu je tedy nutné znát relativní pozici vzhledem k rodičovské kosti, aby bylo možné snadno řetězově počítat matice. Nová absolutní matice kosti, která má rodiče, bude dána jako

$$M_{absolute} = M_{parent} \cdot M_{relative} ,$$

kde $M_{absolute}$ je výsledná matice, $M_{relative}$ je relativní matice dané kosti vzhledem k rodiči, M_{parent} je absolutní matice rodiče.

Jestliže máme k dispozici pouze data, která udávají transformační matice kostí pouze vzhledem ke světu, výpočet relativních matic kostí vzhledem k rodiči je snadno dopočitatelný pomocí

$$M_{relative} = M_{parent}^{-1} \cdot M_{absolute} ,$$

kde $M_{absolute}$ je absolutní matice dané kosti, $M_{relative}$ je výsledná relativní matice vzhledem k rodiči, M_{parent}^{-1} je inverzní absolutní matice rodiče.

5.2 Klíčovaná skeletální animace

Opět můžeme využít klíčovanou animaci, abychom získali jednotný pohyb. Jediný rozdíl v tom, co bylo doposud představeno, je nepoužití bodů, ale matic. Není v tom ale žádný problém, vše funguje naprosto stejně. V tomto případě je možné zvolit za interpolační metodu lineární interpolaci matic. Podle klíče spočítáme novou relativní matici pomocí známého vztahu

$$M = M_{curr}(1 - t) + M_{next} \cdot t ,$$

kde M je výsledná interpolovaná matice, M_{curr} matice aktuálního klíče, M_{next} matice následujícího klíče, t je lokální čas klíče $t \in \langle 0,1 \rangle$. Po tomto kroku je nutné aktualizovat hierarchii kostí.

5.3 Transformace objektů pomocí kostí

V případě, že již je kostra vytvořena, může se přejít k její využití – transformace jiných objektů pomocí kostí, které jsou připojeny k těmto objektům. Obvykle je možné mít k jakémukoli objektu připojeno neomezeně kostí a k nim přiřadit váhu s jakou mají ovlivňovat pozici objektu. Těmito objekty může být naprosto cokoliv. Ať už objekt jako celek, tvořen mnoha vertexy a trojúhelníky, nebo jeden samotný vertex. Všechny tyto možnosti mají stejné opodstatnění a přistupuje se k nim naprosto stejným způsobem. Takže je jedno, jestli transformujeme jediný vertex nebo celou skupinu vertexů.

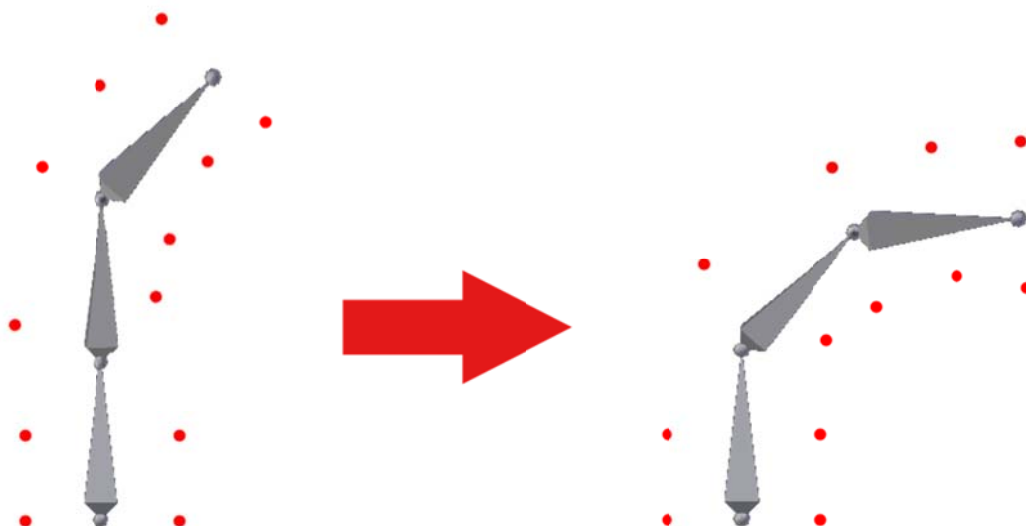
Provádět takové transformace je velmi jednoduché. Stačí pouze využít tento vzorec:

$$M_{final} = M_{absolute} \cdot M_{bindpose}^{-1},$$

kterým získáme M_{final} , což je relativní matice vzhledem k bind pose, protože potřebujeme přesunout střed rotace objektu do pozice kosti, jinak by se objekty otáčely kolem středu souřadnic.

$$pos = pos_{base} \cdot \sum_{i=0}^n w_i \cdot M_i,$$

kde pos je výsledná pozice po transformaci, pos_{base} je výchozí pozice, w_i je váha kosti, M_i je final matice odvozená výše a n udává počet kostí, které jsou spojeny s daným objektem.



Obr. 4 transformace pomocí kostí

5.4 Kvaterniony

Doted' vše funguje jak má a vše vypadá i vcelku dobře. Ale lineární interpolace rotačních matic nemusí vždy přinést dobré výsledky, protože úhlová rychlost není konstantní, proto je nutné vyzkoušet i jiné možnosti, které jsou k dispozici, pro interpolaci rotací. Tyto nové možnosti mohou poskytnout kvaterniony. Definice zde uvedené vycházejí především z popisu a definic, tak jak jsou uvedeny v [2] a [4]

Kvaterniony jsou čtyř dimenzionálním rozšířením komplexních čísel.

5.4.1 Základní vlastnosti

Definice

$$q = w + x \cdot i + y \cdot j + z \cdot k = [w, \vec{v}] = [w, (x, y, z)]$$

Kde x, y, z, w jsou reálné složky a i, j, k imaginární složky. Kvaternion lze také definovat uspořádanou dvojici skalární (w) složky a trojrozměrné vektorové složky (\vec{v}). Vlastnosti imaginárních složek jsou stejné jako u komplexních čísel v rovině, tedy $i^2 = j^2 = k^2 = -1$. Dále lze tyto složky mezi sebou násobit a to tak, že $i \cdot j = -j \cdot i = k$, $j \cdot k = -k \cdot j = i$, $k \cdot i = -i \cdot k = j$.

Sčítání a odčítání je definováno jako prosté sečtení/odečtení všech složek

$$q_0 + q_1 = (w_0 + w_1) + (x_0 + x_1) \cdot i + (y_0 + y_1) \cdot j + (z_0 + z_1) \cdot k$$

Násobení probíhá vynásobením všech členů jednoho kvaternionu, všemi členy druhého kvaternionu

$$q_0 \cdot q_1 = (w_0 w_1 - x_0 x_1 - y_0 y_1 - z_0 z_1) + (w_0 x_1 + x_0 w_1 + y_0 z_1 - z_0 y_1) \cdot i + (w_0 y_1 - x_0 z_1 + y_0 w_1 + z_0 x_1) \cdot j + (w_0 z_1 + x_0 y_1 - y_0 x_1 + z_0 w_1) \cdot k$$

Je třeba si uvědomit, že násobení kvaternionů není komutativní, jak lze vidět již z násobení imaginárních složek mezi sebou.

Konjugovaný kvaternion

$$q^* = [w, -\vec{v}] = w - x \cdot i - y \cdot j - z \cdot k$$

Velikost (norma) kvaternionu

$$\|q\| = \sqrt{x^2 + y^2 + z^2 + w^2}$$

Kvaterniony mohou vyjadřovat rotace, pomocí zadaného úhlu a osy rotace. Takto vytvořené kvaterniony, lze následně převést na klasickou rotační matici 3x3 a použít ji normálním způsobem pro transformace objektů. Proč lze tyto kvaterniony použít pro rotace je dokázáno v [3] a způsob převodu na matici lze nalézt tamtéž.

5.4.2 Sférická lineární interpolace

Kvaterniony lze také lineárně interpolovat pomocí stejných vzorců, které byly již několikrát ukázány. Bohužel, i když je tato metoda lepší než pouhé interpolace matic, nemá také konstantní úhlovou rychlost. Proto byla odvozena sférická lineární interpolace, která je definována jako

$$Slerp(q_0, q_1, t) = q_0 \cdot (q_0^* \cdot q_1)^t$$

Interpoluje se po nejkratším oblouku mezi q_0 a q_1 na jednotkové kvaternionové kouli a úhlové zrychlení je konstantní.

6 Animační systém

6.1 Úvod do systému

Systém je v podobě statické knihovny. Tato knihovna byla nazvána JK2eA. Animace v pojetí JK2eA – nestará se o to jak data zobrazit, ale jakým způsobem je změnit. Pojem klientská aplikace znamená aplikaci, která využívá služeb knihovny.

6.2 Požadavky na animační systém

Předtím než přistoupíme k vývoji, je vhodné si stanovit požadavky, které budeme od systému požadovat:

- modulárnost,
- jednoduchost,
- obecnost.

Modulárnost v tomto případě znamená, že lze snadno vkládat do animačního systému moduly tak, aby nebylo nutné zásadněji měnit strukturu systému.

Animační systém by měl být natolik obecný, aby nezáleželo na tom, jakým způsobem budou data zobrazována.

Tyto vlastnosti, však mohou mít i své negativní důsledky. Je vždy výhodné využít přesně specializovaný systém, který maximálně využije možností hardwaru.

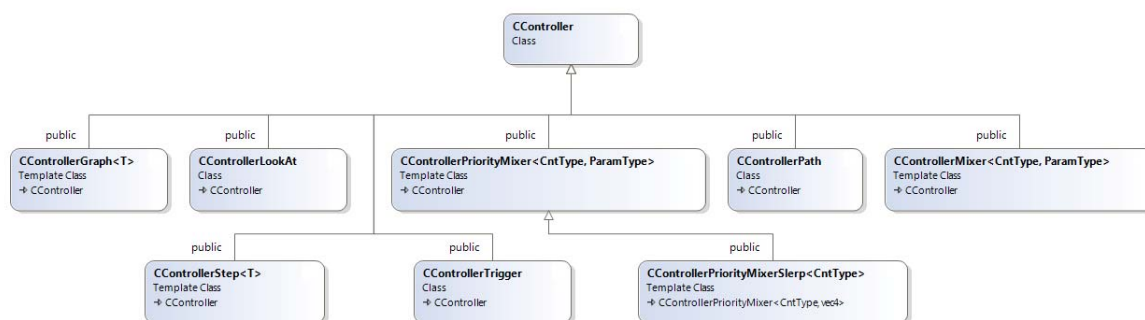
6.3 Základní vlastnosti

Základní stavební jednotkou tohoto systému je tzv. controller [6]. Controller zde obecně znamená objekt, který na základě nějakých vstupních dat a času, změní jiná data. Vstupní data jsou tedy oblastí paměti a nazývají se **animation data**. Data, která se mají měnit, se nazývají **parameter**. Systém ovšem nevyžaduje, aby existovala animation data a nemusí existovat ani parameter. Tento koncept plní současně první a druhý požadavek na systém – modulárnost a jednoduchost.

Můžeme uvést příklad, jak takový controller funguje. Je dána nějaká křivka a je požadováno, aby byla měněna pozice nějakého bodu, na základě času. Controller dostane ukazatel do paměti, kde se nachází data, která mají být na základě této křivky změněna. Zjevně se jedná o pozici nějakého bodu. Controller tedy vezme čas, vypočítá, kde se aktuálně nachází na křivce a takto získanou pozici zapíše do paměti, kde se nachází připojený bod. Takto jednoduše fungují všechny controllery, které tento systém obsahuje.

6.4 Struktura systému

Diagram tříd



Obr. 5 vazby tříd controllerů

6.5 Controller

Bázová třída pro všechny ostatní controllery. Obsahuje čistě virtuální metodu Update, která se stará o aktualizace příslušného controlleru. Každý controller má také definovaný svůj lokální čas. Controller má definované stavy, ve kterých se může nacházet

- active – je aktivní může volat tělo metody update,
- sleeping – jeho lokální čas roste, ale neprovádí se nic více,
- waiting – je pasivní, čeká,
- finished – skončil svou činnost.

Také se musí nastavit, co by se mělo provést, pokud je lokální čas mimo definovanou časovou oblast controlleru (tzv. ORT typy). Definuje se několik ORT

- loop – lokální čas se nastaví na nulu a činnost controlleru se opakuje,
- finish – změni svůj stav na finished,
- random – lokálnímu času bude přiřazena nová náhodná hodnota.

Dále obsahuje speciální atribut bPassive udává zda se má, pokud je zaregistrován v controller manageru, volat metoda update.

6.6 Controller manager

Spravuje registrované controllery, manager je jen jeden a jedná se o Singleton vlase [5]. Není vyžadováno, aby byly všechny controllery registrovány v manageru, ale poté je nutné, aby klientská aplikace všechny operace nad controllery zajišťovala sama. Mezi hlavní funkce manageru patří centrální aktualizace všech zaregistrovaných controllerů. Všechny controllery, které se registrují do manageru musejí být vytvářeny dynamicky a manager se sám stará o uvolnění takto alokované paměti, pokud již controller není používán.

Manager funguje také jako Observer (Observer patern [5]) a hlídá si, zda již některé parametry controllerů jsou neplatné. Proto je nutné při registraci uvádět Observable objekt, který dá vědět manageru, pokud byl parametr zrušen. Pokud se tak stane, automaticky budou všechny controllery, které tento parametr využívají smazány.

Protože se předpokládá, že všechny controllery mohou mít řízen nějaký svůj atribut jiným controllerem, je již v básové třídě zakomponován observable objekt, který upozorní všechny zaregistrované controllery při volání destruktoru příslušného controlleru. Klientská aplikace by tedy měla mít vytvořené observable objekty podobným způsobem.

6.7 Graph a path

Jedná se o podpůrné objekty, které jsou využívány graph a path controllery jako animation data. Graph složí k použití křivek v klíčované animaci. Poskytuje interpolace lineární, polynomiální, Catmul-Rom a kubickou Bézierovu aproximaci. Třídy odvozené od CCurveGraph jsou generické a může tedy jako hodnota uzlu sloužit jakýkoliv objekt, který má definovány příslušné operátory (+, -, *).

CCurvePath objekty nejsou generické a využívají pouze 3D souřadnice. A jsou uzpůsobeny pro využití křivek pro cesty. Takže místo interpolace podle času, používají interpolace podle délky. Jsou implementovány pouze dva typy cest – lineární, Catmull-Rom.

6.8 Graph, path controllers

Graph controller využívá jako animační data CCurveGraph a jako parametr musí použít stejný typ, jaký se používá právě u tohoto objektu. V metodě Update nedělá nic jiného než, že mu roste lokální čas a používá ho jako vstupní parametr pro metodu Interpolace, připojeného grafu.

Path controller funguje obdobně. Jeho animačními daty je CCurvePath a parametrem pozice a směr. Směr udává aktuální směr tečného vektoru na křivce a jeho zadání je nepovinné. Těleso se tak může pohybovat po cestě a otáčet se jiným způsobem (např. pomocí controlleru LookAt). V metodě Update se podle rychlosti a aktuálního lokálního času vypočítává dráha, která se stává vstupním parametrem metody GetValue připojené CCurvePath.

6.9 Míchací controllery

Míchání je definováno takto

$$P_{res} = \sum_{i=0}^n w_i P_i,$$

kde P_{res} je výsledná hodnota parametru, n je počet zaregistrovaných controllerů, w_i je váha controlleru a P_i je aktuální hodnota parametru, který příslušný controller používá.

Všechny controllery, které se zaregistrují do míchacích controllerů, budou mít nastavený atribut `bPassive` na `true`. O jejich aktualizaci se totiž stará sám mixer ve své metodě `Update`. První vložený controller do mixeru udává, jaký parametr musejí mít ostatní controllery, které budou vloženy později. Nelze tedy vložit do jednoho mixeru rozdílné parametry, u kterých by míchání nemuselo mít smysl.

Dále je nutné při vložení udat váhu. Váha se udává jako ukazatel na paměť a lze tedy využít parametr, na který je již nějaký controller navázán a takto dynamicky váhu měnit. Váha není nijak shora omezena a může být klidně větší než jedna, ale ne menší než nula. Avšak to neznamena, že součet všech vah je větší než jedna. Při každé aktualizaci se používá pomocná proměnná, která udává zbývající váhu ($z\ 1$), pokud by již odečtení váhy, nějakého controlleru způsobilo, že váha bude menší než nula ostatní controllery se přeskočí a na controller, který toto způsobil bude použita dočasně zbývající váha. Tato vlastnost umožňuje, aby jeden controller nebo skupina controllerů převzala naprostou kontrolu nad parametrem a ostatní controllery v mixeru neměli žádnou váhu na tom jak bude výsledný parametr vypadat. Zde by mohl nastat problém, pokud by bylo třeba přesně určit pořadí, v jakém se má odebírat od aktuálně zbývající váhy. Proto systém rozlišuje dva typy mixerů:

1. normální (`CControllerMixer`),
2. prioritní (`CControllerPriorityMixer`).

Normální používá pořadí, v jakém byly controllery do mixeru registrovány a prioritní, jak již název napovídá, používá pořadí podle nastavené priority jednotlivých controllerů.

V systému také existuje specializovaný prioritní míchací controller, který nepoužívá lineární míchání, ale sférickou interpolaci kvaternionů. Tento mixer používá hlavně `Skeletal Animation Mixer`, který bude popsán v části o skeletální animaci.

6.10 Ostatní

6.10.1 Controller Lokat

Někdy je třeba provést rotaci objektu, tak aby se „díval“ na objekt jiný. Právě k tomuto slouží `ControllerLookAt`.

Parametrem pro něj je transformační matice pozorovacího objektu a pozice pozorovaného objektu. Dále lze nastavit některé vlastnosti, jak bude probíhat pozorování. Jedná se o způsob, jakým bude vytvářena rotační matice (směr může být invertován)

Nepodporuje omezení rotace, takže se objekt bude otáčet ve všech osách.

6.10.2 Controller pro spouštění událostí

Tento controller je velmi zvláštní a na první pohled nezapadá do systému. Jedná se o výjimku z pravidla, že controller obsahuje parametr. Tento žádný neobsahuje, protože jedinou jeho prací je spouštění událostí v určený čas. Může buď využívat svůj interní čas, nebo interní čas jiného controlleru. Animačními daty jsou pro něj čas v sekundách, kdy se má spustit a funkční objekt, který popisuje, co se má spustit. Implementace takového objektu je plně v režii klientské aplikace nebo uživatele v rámci animačního systému, stačí, aby byl odvozen od rozhraní, které je poskytováno.

Jeho hlavní zamýšlenou funkcí je zapínání akcí ve skeletální animaci. Pod tím si lze představit spouštění zvuků, částicových efektů, skriptů a podobně. Avšak není nijak omezen a může spouštět cokoli kdykoliv.

6.10.3 Controller schodové funkce

Velmi jednoduchý controller, který mění parametr podle zadané schodové funkce. Parametrem může být pouze hodnota float nebo int. Animační data popisují schodovou funkci; jedná se o počáteční hodnotu, velikost kroku, časovou prodlevu mezi kroky a celkový počet kroků.

6.11 Mesh morph

Morfování objektů je velmi jednoduchá záležitost a není k ní potřeba nějaká složitá struktura. V zásadě se skládá ze dvou částí – CMorphTargetMesh, CMorphTargetController.

CMorphTargetMesh se stará o načítání souborů, které obsahují informace o morph targetech a toto data následně zprostředkuje klientské aplikaci, aby si je mohla uložit do grafické paměti.

CMorphTargetController je velmi jednoduchý, pouze řídí váhy jednotlivých morph targetů, které byly načteny. Umožňuje plynule aktivovat a deaktivovat morph targety.

O všechno ostatní se musí starat klientská aplikace, protože se předpokládá, že aktualizace meshe bude provedena na GPU.

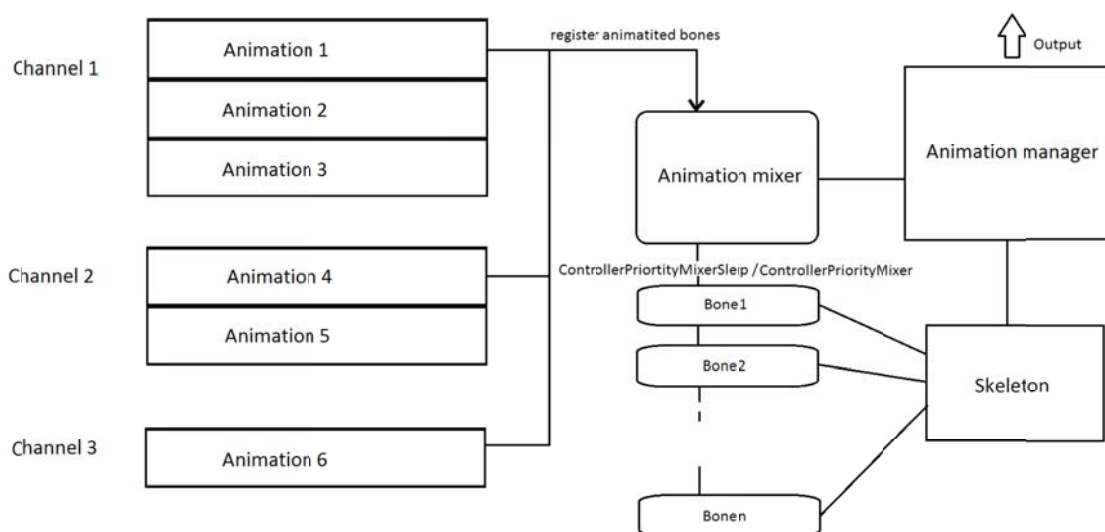
6.12 Skeletální animace

Skeletální animace je hlavní částí celého animačního systému a proto ji byl věnován největší prostor. Tato důležitost vychází z cílení nasazení systému pro počítačové hry, kde je animace humanoidních postav velmi důležitá.

Současná implementace skeletální animace umožňuje spouštět pouze klíčované animace a tyto míchat mezi sebou.

Animační systém pouze zprostředkovává aktuální matice kostí, o skinning (změnu pozic vertexů) se musí starat klientská aplikace. Tato nepřímost animace vychází z předpokladů, že se transformace vertexů (nebo čehokoli jiného) bude provádět, podobně jako u morfování, přímo na GPU.

Obrázek (Obr. 6) ukazuje, jak vypadá struktura systému skeletální animace. Využívají se zde koncept kanálů a mixeru. Podrobněji bude popsán dále.



Obr. 6 struktura systému skeletální animace

6.12.1 Kostra a kosti

O kosti se stará třída CBone a o kostry třída CSkeleton. Kosti mají implementováno (jak bylo uvedeno v části 5.1) vše co je nutné k popsání kosti. Navíc obsahují kvaternion, který udává aktuální relativní rotaci kosti vzhledem k rodiči (pokud ho má). Používá se k interpolaci a může být použit jako parametr pro CControllerGraph, který má za své animační data CCurveGraphSlerp, tedy pro klíčovanou animaci.

V kostře se shromažďují kosti, které vytvářejí příslušnou kosterní hierarchii. Kostru lze načíst z příslušného místa v JK2 souboru. Stará se také o aktualizace hierarchie kostí voláním metody UpdateBones, kde se přepočítá absolutní matice všech kostí, i těch na kterých žádná změna nikdy neproběhla. Kvaternion kosti je převeden na matici a použit k výpočtu absolutní matice. Pokud chceme vložit do kostry novou kost, která má rodiče, je nutné, aby rodič byl do kostry vložen předtím, než příslušná kost. Pokud se rodičovská kost nenachází v příslušné kostře, kost do kostry nebude vložena.

6.12.2 Animace

Na CControllerSkeletalAnimation je možné nahlížet, jak již název napovídá, jako na controller, ale není doporučeno ho tímto způsobem používat. Parametrem pro tento controller by byl celý mesh nebo model, na který chceme aplikovat skeletální animaci. Animační data jsou klíče grafů translace a rotace. Bohužel toto použití má jednu velkou nevýhodu – nemožnost snadného míchání. Vše kolem by musela řešit klientská aplikace. Proč tedy vše toto nezakomponovat přímo do animačního systému? Animace je tedy možno používat více způsoby a záleží na uživateli, k čemu se rozhodne. Doporučení je jasné – využít Animation manager.

Protože se jedná o složený controller, který obsahuje nějaké další podřízené controllery, je nutné zajistit, aby při nastavování stavů, ORT a dalších metod svázaných s controllerem, byla všechna tato nastavení provedena také na všech podřízených controllerech.

Data pro animace jsou ukládána v jednotlivých souborech, které obsahují klíče pro jednotlivé kosti (soubor JK2A – viz Příloha A). Pro každou kost, na kterou se odkazuje soubor s animací, se vytvoří dvojice CCurveGraphLinear a CControllerGraph pro posuny kostí a dvojice CCurveGraphSlerp a CControllerGraph pro rotace kostí.

6.12.3 Animation manager

CSkeletalAnimationManager funguje jako vyšší vrstva mezi skeletálními animacemi. Dal by se popsat jako shromaždiště, které sjednocuje kostru, animace, kanály. Má také společné znaky s třídou CControllerManager jako je centrální aktualizace všech objektů, které obsahuje. Každý manager musí mít přiřazenou kostru a to pouze kostru, která je načtena ze souboru. Počet kanálů je konstantní, obsahuje osm kanálů (nic ale nebrání tomu, aby jich bylo více), lze tedy spustit „současně“ až osm animací (není to úplně pravda, protože některé animace mohou být zcela překryty jinou a nemají na výsledný tvar žádný vliv).

Tato část by se v animačním systému nemusela vůbec nacházet a mohla by být plně ponechána na implementaci klientskou aplikací, protože vše potřebné k vytváření skeletálních animací lze složit z controllerů již představených.

Další funkcí je zjednodušení používání skeletální animace. Díky tomu lze pouhým voláním metody ActivateAnimation, animace spouštět, míchat a vypínat (překrýt právě aktivovanou animací), navíc lze i úplně vypnout příslušný kanál použitím metody DeactivateChannel.

6.12.4 Animation Mixer

Protože je CControllerSkeletalAnimation složený controller a obsahuje controllery, které používají rozdílné parametry, není možné použít standardní míchací controllery. CSkeletalAnimationMixer řeší právě tento problém. Protože v animacích se již používají jednoduché controllery, je možné na ně použít standardní mixery. Proto animace musí zaregistrovat všechny svoje vnitřní controllery do mixeru, voláním své metody ToMixer, kde jedním z parametrů je právě Animation mixer. V metodě ToMixer se volá metoda mixeru AddBoneController, který přiřadí controllery vázané na stejné kosti do jednoho správného standardního mixeru. Pro každou kost z referencované kostry je vytvořen jeden míchací controller.

V rámci jednoho Animation manageru je pouze jeden Animation mixer.

6.12.5 Kanály

Třída, která zprostředkovává kanály, se jmenuje CSkeletalAnimationChannel. Kanály jsou v Animation manageru seřazeny podle priorit, kde kanál 0 má nejmenší prioritu a kanál N největší. Toto znamená, že pokud běžící animace v kanálu s nižší prioritou upravuje matice stejné kosti jako běžící animace s vyšší prioritou, bude transformace od animace s nižší prioritou úplně ignorována. Tímto způsobem lze poté vytvořit animace, kde ve vrstvě s nejnižší prioritou běží animace chůze, která samozřejmě mění i kosti horních končetin a ve vrstvě s vyšší prioritou běží animace mávání rukou. Takto byla vytvořena úplně nová

animace chůze a mávání. Kdyby animační systém nepodporoval takovéto míchání, musela by se animace vytvořit ručně.

Každý kanál může obsahovat až n animací, ale současně můžou běžet zatím pouze dvě. Většinu času ovšem v kanálu běží jen jedna animace, dvě pouze při přechodu z jedné do druhé.

Každé vložené animaci se přiřadí váha s jakou budou její vnitřní controllery ovlivňovat výslednou pozici kosti. Kanál obsahuje dva základní odkazy na animace – blend in (animace, která se má plynule začít přehrávat), blend out (animace, která se má plynule přestat přehrávat). Přejchod mezi těmito animacemi je lineární. Váha blend in animace roste váha blend out klesá. Až je váha blend out animace nulová, blend in animace se nastaví jakou blend out a blend in se nastaví prázdný odkaz. Výměna těchto odkazů je proto, aby při další aktivaci animace bylo hned jasné, která se má přestat přehrávat. Změny vah se provádějí, pouze pokud jsou oba odkazy nastaveny na platné animace.

7 Ukázková aplikace

7.1 OpenGL API

OpenGL („Open Graphics Library“) je grafické API spravované organizací Khronos Group, které slouží jako softwarové rozhraní pro grafický hardware. Lze tedy ke kreslení využívat přímo grafickou kartu, aniž bychom museli nějakým závažným způsobem zapojovat do rasterizačních metod CPU. Obsahuje mnoho příkazů, které slouží k definici geometrických 2D nebo 3D objektů a řízení jak budou tyto objekty vykresleny. Dále lze najít příkazy, které pracují přímo s vytvořenými pixely – mohou pixely číst nebo přímo zapisovat.

OpenGL je podporováno na širokém množství hardwaru napříč všemi platformami, dokonce i mobilními (OpenGL ES) nebo existuje verze OpenGL přímo pro webové prohlížeče (WebGL). Jednou z mnoha výhod OpenGL jsou rozšíření, která nemusejí být uvedeny přímo v základní specifikaci OpenGL a výrobci hardware mohou implementovat speciální rozšíření, která využívají nějaké speciální vlastnosti jejich grafického hardwaru.

Již od verze 3.1, OpenGL přímo využívá moderní programovatelnou grafickou pipeline. Starší verze (od 2.x) toto mohly, ale nemusely využívat. Bylo možné například využít přímo systém per-vertex osvětlení, což je v novějších verzích ponecháno na vlastní implementaci uživatele. Dále již není možné využívat tzv. immediate mod (přímé kreslení), kde se používali mezi dvojicí procedur `glBegin`, `glEnd` kreslicí příkazy `glVertex` atd. Všechny staré příkazy jsou označeny jako zastaralé a je tudíž vhodné se jim vyhýbat. OpenGL, ale dává možnost pomalu přecházet ze starších verzí, zavedením *compatibility profile* a *core profile*. V *compatibility profile* lze kombinovat jak nové funkce a procedury, tak staré. V *core profile* již nelze tyto procedury dále používat. V době psaní této práce se nachází OpenGL ve verzi 4.2.

Protože OpenGL žádným způsobem nepoužívá koncept objektově orientovaného programování, je ponecháno na uživateli API, aby si vše vytvořil sám. K těmto účelům byla vytvořena vlastní knihovna JK2eLite, která, mimo jiné, poskytuje objektový přístup k OpenGL příkazům. Tato knihovna byla vytvářena pro OpenGL 3.3 core profile.

7.2 Základní postup vytvoření OpenGL okna ve Windows

Protože OpenGL API nefunguje samo o sobě, je nutné mít další API, která připraví a budou spravovat okna, do kterých lze kreslit. Ve Windows se toto API nazývá WGL. Popis všech následujících funkcí vychází z [10], pokud není uvedeno jinak.

Následující popis popíše postup při vytváření okna v C++ (prostředí MSVC) a WinAPI. Předtím než je vůbec možné začít používat WGL je nutné vytvořit standardní okno ve WinAPI. A příslušné hlavičkové soubory, které by měly být součástí standardních hlavičkových souborů MSVC – „gl.h“.

Po vytvoření okna (volání `CreateWindow`) musíme nejprve vytvořit Pixel Format Descriptor (PFD), který popisuje, jakým způsobem budou na obrazovce zobrazovány pixely. Nastavuje se zde např. barevná hloubka, depth buffer a mnoho dalších. Podle PFD

bude vybrán pixel formát (ChoosePixelFormat), který se nejlépe přiblíží k definici a je podporován daným zařízením.

```
PIXELFORMATDESCRIPTOR pfd;  
HDC hdc = GetDC(hWnd);  
int pf = ChoosePixelFormat(hdc, &pfd);  
SetPixelFormat(hdc, pf, &pfd);
```

Po nastavení pixel formátu je třeba vytvořit tzv. render kontext (RC), spojit ho s draw kontextem a nastavit jako aktuální. Právě zde je místo jak nastavit zmíněné profily kompatibility OpenGL 3.2 a vyšší.

Protože standardní knihovna poskytovaná s MSVC nepodporuje OpenGL vyšších verzích, je nutné buď použít hlavičkový soubor, ve kterém jsou uvedeny všechny procedury, funkce, konstanty, atd. a propojit si tyto s dynamickou knihovnou opengl32.dll ručně. Většinou je tato knihovna distribuována spolu s ovladači grafické karty, není nutné využívat žádné aktualizace Windows. V této práci není využíváno ruční propojování, ale využívá se zde open-source knihovna GLEW¹, která vše zařídí automaticky.

Pokud tedy, již máme připravena rozšíření pro OpenGL 3.3 můžeme přistoupit k vytvoření kontextu pro tuto verzi. Provádí se to voláním funkce wglCreateContextAttribsARB, která vrací nový render kontext, pokud se podařilo vytvořit kontext, pokud ne znamená to, že grafická karta nepodporuje tuto verzi OpenGL a není možné dále pokračovat.

Používá se zde pole, které popisuje verzi OpenGL (WGL_CONTEXT_MAJOR_VERSION, WGL_CONTEXT_MINOR_VERSION) a nastavení profilů kompatibility.

```
int GL33[] = {WGL_CONTEXT_MAJOR_VERSION_ARB, 3,  
             WGL_CONTEXT_MINOR_VERSION_ARB, 3,  
             WGL_CONTEXT_FLAGS_ARB,  
             WGL_CONTEXT_FORWARD_COMPATIBLE_BIT_ARB,  
             WGL_CONTEXT_PROFILE_MASK_ARB,  
             WGL_CONTEXT_CORE_PROFILE_BIT_ARB,  
             0  
};  
HGLRC hRC = wglCreateContextAttribsARB(hDC, 0, attribListGL33);
```

Po tomto kroku již můžeme začít používat OpenGL příkazy. Obvykle se zde provádí inicializace různých nastavení jako depth testu, projekční matice atd. Po všech těchto prvotních nastavení můžeme již přistoupit k renderovací smyčce. Jedná se o nekonečný cyklus, kde na konci každého průchodu je volána procedura wglSwapBuffers, které pošle na grafickou kartu, a provede výměnu front a back bufferu, kdy se z back bufferu stane front buffer (double buffering). OpenGL nativně podporuje pouze double buffering, toto

¹ The OpenGL Extension Wrangler Library – <http://glew.sourceforge.net/>

omezení lze obejít vytvořením vlastních frame bufferů, ale toto řešení není příliš vhodné. Většinou existuje podpora triple buffering již od výrobce grafické karty. Před skončením aplikace je nutné po sobě uklidit, smazat dříve vytvořený render context.

```
wglMakeCurrent(hDC, 0);  
wglDeleteContext(hRC);
```

7.3 Seznámení s JK2eLite

Knihovna JK2eLite zapouzdřuje OpenGL volání do tříd, stará se o zobrazení modelů a také přímo využívá, zde navrhnutý animační systém pro animování modelů. Ve vyšší verzi obsahuje také podporu pro částicové systémy, GUI, fyzikální engine, podporu přehrávání zvuku atd. Vzhledem k využití v této ukázkové aplikaci nejsou tyto části nutné, a proto stačí jen ochuzená verze. Dále již bude popsána jen funkčnost jednotlivých částí, které zapouzdřují volání OpenGL, ne přesné názvy funkcí, které se zde používají.

7.3.1 Základní části knihovny JK2eLite

Renderer

Slouží jako vstupní bod pro OpenGL – inicializace, kreslení primitiv (body, čáry), výchozí shader, uchovávání matic (projection, modelview).

OGL_Buffer

OGL_Buffer slouží k alokování paměti přímo na grafické kartě. Data, která jsou často používána, je vhodné vložit přímo do VRAM a ušetřit tak čas, který by byl nutný k častému přenosu dat z RAM do VRAM. Obvykle není nutné přímo používat OGL_Buffer, protože máme k dispozici další objekty v rámci JK2e, které ho již používají a upřesňují jeho použití.

OGL_Shader

Slouží k vytváření shaderů (programů, které se spouštějí na grafické kartě). OpenGL 3.3 rozlišuje tři základní části shaderů:

- Vertex shader – aplikuje se na jednotlivé vertexy
- Geometry shader – aplikuje se na celé primitivy (čáry, trojúhelníky, atd.)
- Fragment shader (v DirectX nazýván pixel) – aplikuje na každý pixel

Vytvořit shader program, lze snadno zavoláním metody CreateShader a vložením argumentů, které znamenají cestu k textovým souborům se zdrojovými kódy jednotlivých shaderů. OpenGL využívá vlastní vyšší programovací jazyk pro psaní shaderů – GLSL („OpenGL Shading Language“). Tyto části jsou zkompileovány a spojeny do jednoho programu. Těchto programů může existovat více, a pokud chceme nějaký program použít, využijeme metodu BindShader.

OGL_Texture

Načte textury ze souboru (bmp, tga, jpg) a vytvoří texturu, kterou lze následně používat. OpenGL podporuje více druhů textur (1D, 2D, 3D, cube), ale OGL_Texture umí pracovat pouze s 2D texturami. Pro texturu se také implicitně vytvářejí mip-mapy. Je možné také nastavovat různé typy filtrování textury (bilinear, trilinear, aniso, atd.).

Pro vytvoření textury se používá metoda LoadTexture. A pokud chceme texturu použít, musíme využít metodu BindTexture. Zpravidla je nežádoucí a zbytečné načítat do paměti několik stejných textur. Právě kvůli tomu je vhodnější použít centrální manager, který se stará o textury a požádat ho o načtení příslušné textury – metoda manageru AddTexture. Bude nám vrácen ukazatel na texturu a můžeme si být jisti tím, že se nebude plýtvat zdroji.

OGL_VBO

Do Vertex Buffer Object se ukládají data nutná k zobrazení objektů – vertexy a trojúhelníky. Trojúhelník je definován pomocí tří pořadových čísel vertexů. Vertex je obvykle nutné charakterizovat více atributy než je pouhá souřadnice. Mezi tyto atributy patří např. pozice vertexu v textuře (UV), barva, atd. OGL_VBO nabízí několik šablon vertexů s atributy – jednoduchý (pouze souřadnice, NoUV), s UV (Normal), s podporou skeletální animace (Skinned). Nový VBO se vytváří voláním příslušné metody CreateX, kde místo X je NoUV, Normal nebo Skinned. Jako argument je jednorozměrné pole vertexů, které musí dodržet pořadí atributů, např. jeden Normal vertex je definován pěticí hodnot typu float v pořadí x, y, z, u, v.

Po vytvoření se používá metoda Draw, která vykreslí zadaný počet trojúhelníků na obrazovku.

OGL_TBO

Texture buffer object má se standardním chápáním textur v OpenGL pramálo společného a název „texture“ může být zavádějící. Jedná se spíše o 1D buffer ke kterému lze přímo přistupovat v rámci shaderu. Toto je velmi důležité – získáme možnost číst velké objemy dat přímo v shader programu.

OGL_UBO

Jako spoj mezi shader programem, OpenGL a aplikací existují tzv. uniform proměnné (uniform, protože nemění svoji hodnotu v závislosti na zpracovávané primitivě). Tento koncept dále rozšiřuje Uniform Buffer Object. Jedná se o klasický buffer, který lze připojit k více shader programům jen jednou a pokud se na něm provedou nějaké změny, uvidí je okamžitě všichni. Toto je velká výhoda oproti využívání standardních jednoduchých uniform proměnných, které je nutné pro každý shader program nastavovat zvlášť. Jednou z výhod UBO je tedy možnost vytvořit buffer s globálními proměnnými (v OpenGL je vhodné UBO využít pro projection a modelview matice).

CMODEL

Slouží k vytváření a kreslení modelů. Používá vlastní formát (JK2 – viz Příloha A). Využívají se zde další třídy CMesh, CMaterial. Ze základních představených částí využívajících OpenGL se používá OGL_VBO, a OGL_TEXTURE.

Model přímo využívá animační systém – především část zajišťující skeletální animaci (CSkeletalAnimationManager) a morfování tvarů. Protože je nevhodné načítat stejné modely několikrát (podobně jako u textur), byl k tomuto účelu vytvořen manager, který spravuje všechny dosud načtené modely – ModelManager. Využitím metody manageru AddModel, buď model vytvoří (pokud ještě neexistuje) a vrátí ukazatel na tento model nebo přímo vrátí ukazatel na již existující model.

7.4 Implementace morfování a skinningu v GLSL

Jak již bylo zmíněno v části popisující animační systém, některé animace jsou ponechány na implementaci klientské aplikaci s vhodným využitím GPU. Vše se provádí ve vertex shaderu.

Skinning

Jediné co je potřeba vytvořit jsou uniform proměnné, které budou obsahovat matice kostí. Bohužel, je možné využít jen omezené množství pro uchování proměnných. Dále každý vertex jako svůj atribut obsahuje váhu (in_weights) a indexy kostí pro tyto váhy (in_bones).

```
uniform mat4 bones[32];
void main(void)
{
    gl_Position = projection * modelview *
        ( (in_weights.x * bones[int(in_bones.x)]) +
          (in_weights.y * bones[int(in_bones.y)]) +
          (in_weights.z * bones[int(in_bones.z)]) +
          (in_weights.w * bones[int(in_bones.w)]) ) * in_position;
}
```

Jak je vidět jedná se jen o pouhé aplikování vzorce z části 5.3. Takto provedený skinning je mnohem rychlejší než kdybychom vše museli provádět pomocí CPU a zapisovat do dalšího VBO. Avšak pokud bychom chtěli takto změněné modely vykreslovat vícekrát, nemuselo by být vykreslování rychlejší než skinning jednou provést na CPU a již využívat znovu a znovu. Takto omezení bychom byly pouze při použití nižší verze OpenGL (2.x), kde provedení skinningu na grafické kartě funguje také, ale modernější grafické karty podporují funkci stream out tzn. zapsání do bufferu přímo na GPU. Takto tedy ušetříme spoustu času a můžeme z nově zapsaného bufferu, kreslit kolikrát chceme.

Morfování

U morfování je situace trochu komplikovanější, protože si potřebujeme pomatovat, offsety vertexů všech aktivních morph targetů. Zde nastává prostor pro TBO, do kterého všechny offsety všech zaregistrovaných (ne pouze aktivních) uložíme. Bohužel, řešení že si nahrajeme všechny morph targety do TBO není příliš vhodné, ale vzhledem k představě jak funguje morphing na GPU je to dostačující. Také je nutné využít formát textury RGBA32F (4*float), což znamená, že na jeden offset potřebujeme šestnáct bytů. Správně

lze usuzovat, že by mohlo stačit dvanáct bytů ($3 \times \text{float} = x, y, z$), bohužel v používané OpenGL verzi (3.3) toto není možné, avšak verze 4.x již toto omezení odstraňuje a je možné použít formát RGB32F.

Když jsou již připravené offsety vertexů, je již nutné pouze vytvořit buffer ve kterém budou uloženy aktuální váhy morph targetů. K tomuto účelu bude použit UBO. Samozřejmě, že je možné využít také TBO, ale je třeba počítat s tím, že přenos dat do TBO může být pomalejší než přenos dat do UBO. Je třeba zvážit všechna pro a proti a vyzkoušet co je pro cílovou aplikaci nejvhodnější.

```
#define MAX_BLEND_SHAPES 5 // = 20

uniform samplerBuffer buffer;
uniform int blend_shapes_count;
uniform int vert_count;
layout(std140) uniform blend_shapes
{
    vec4 weight[MAX_BLEND_SHAPES];
};

void main()
{
    vec4 final_position = in_position;
    for(int i = 0; i < blend_shapes_count; i++) {
        vec4 offset = texelFetch(buffer, gl_VertexID + i*vert_count);
        final_position.xyz += offset.xyz * weight[i / 4][i % 4];
    }

    gl_Position = projection * (modelview * final_position);
}
```

Lze si všimnout, že se zde používají i jednoduché uniform proměnné, které udávají skutečný počet morph targetů (`blend_shapes_count`), počet vertexů, abychom mohli zjistit, kde v TBO začíná nový morph target. UBO se zde jmenuje `blend_shapes` a jsou v něm uloženy aktuální váhy po čtveřicích. Uniform proměnná typu `samplerBuffer` určuje v jaké texturovací jednotce je připojen TBO. Samotná logika kódu je velmi jednoduchá – pomocí vnitřní funkce GLSL `texelFetch`, přečteme data uložená v TBO a poté je pomocí známých vzorců z části 4 aplikujeme.

7.5 Samotná aplikace

Ukázková aplikace si neklade za cíl nic více, než ukázání jednoduché scény, kde jsou animovány určité objekty. Kameru v aplikaci je buď možné ovládat ručně (pomocí kláves W,S,A,D a myši) nebo nechat pohybovat po předem definované cestě. Tyto módy se přepínají klávesou F2.

Přehled ukázek možností animačního systému v aplikaci

- pohyb po lineární křivce,
- pohyb po Catmull-Rom křivce,
- změna rychlosti pohybu po křivce na základě klíčovaného grafu,
- změna barev na základě klíčovaného grafu,
- skeletální animace s pohybem po křivce
- morph targets.

V aplikaci byl použit model člověka, který byl vygenerován aplikací MakeHuman² a motion capture data (chůze, běh) z databáze Carnegie Mellon University³.

² <http://www.makehuman.org/>

³ <http://mocap.cs.cmu.edu/>

Závěr

V rámci této práce bylo dosaženo všech cílů, které byly stanoveny – implementace animačního systému, který bude využívat morfování objektů, skeletální animace a pohyb po cestách. Bohužel, ale v současné podobě není animační systém připraven k tomu, aby byl využit v jakékoliv hře. Pro testovací účely slouží velmi dobře, ale pro využití ve hrách má ještě spoustu nedostatků.

Díky dlouhodobé přípravě a návrhu, neprošel animační systém žádnými drastickými změnami a jeho podoba a struktura se od začátku vývoje příliš nebo vůbec nezměnila. Přesně toto je jedním z mnoha přínosů, které celý tento vývoj přinesl. Důležitost přípravy. A taková příprava nemusí být ani příliš detailní, důležité je znát spíše makro detaily (komunikace částí systémů) než mikro detaily (podoba tříd).

Během vývoje bylo věnováno enormní úsilí do vybudování editoru, ve kterém se měly animace interaktivně vytvářet. Této aplikaci bylo věnováno mnohem více času, než celému animačnímu systému a i přesto nebyla dopracována do funkční podoby, která by prezentovala možnosti celého systému. Editor se proto nestal přímo ukázkovou aplikací této bakalářské práce. Právě kvůli tomuto špatnému rozhodnutí nemohlo být věnováno více času animačnímu systému, který by si to jistě zasloužil.

Protože celý projekt animačního systému je součástí dlouhodobého plánu vývoje zde představené knihovny JK2e, vývoj bude nadále pokračovat. Prvním vylepšením bude přepracování části, která se stará o skeletální animaci, tak aby mohla využívat fyzikální engine a inverzní kinematiku. Dále dopracování editoru a systému, tak aby mohla být implementována podpora animačních stromů, což velmi usnadní míchání skeletálních animací. Velmi důležitou částí je také nutnost mít přímou podporu pro animaci postav. Proto bude vytvořen modul animace postav, který bude využívat skeletální modul a bude snadno zprostředkovávat vše, co je nutné proto, aby animace postav vypadali co nejlépe. V současné době více jádrových procesorů je také nutné využívat paralelní zpracovávání. Animační systém by nemělo být až tolik obtížné paralelizovat, právě proto se v něm nachází manažer controllerů, který se měl starat o přiřazování vláken jednotlivým controllerům. Bohužel toto nebylo implementováno, ale předpokládá se, že v blízké budoucnosti bude. Další rozvoj zaznamená také využívání OpenGL. Protože je celý projekt JK2e plánován na velmi dlouhou dobu a využívání zastaralých technik není výhodné, bude proveden přechod z OpenGL 3.3 na OpenGL 4.x.

Celkově práce přinesla mnoho nových zkušeností a znalostí, které bude možné dále rozvíjet. Lze jen doufat, že v brzké době všechny tyto zkušenosti přinesou konečně možnost využití v aplikacích, díky kterým bylo veškeré toto úsilí vynaloženo – počítačových her.

Zdroje

- [1] SALOMON, David. *Curves and surfaces for computer graphics*. New York: Springer, c2006, 460 s. ISBN 03-872-4196-5.
- [2] EBERLY, David H. . *3D game engine design : a practical approach to real-time computer graphics*. 2nd ed. Amsterdam : Morgan Kaufman, 2007. 1018 s. ISBN 0-12-229063-1, ISBN 978-0-12-229063-3.
- [3] ADAMS, Jim. *Advanced Animation with DirectX*. Boston (Mass.) : PREMIER PRESS, 2003. 452 s. ISBN 1-59200-037-1.
- [4] DAM, Erik B., Martin KOCH, Martin LILHOLM, *Quaternions, Interpolation and Animation*. Technical Report DIKU-TR-98/5. Copenhagen: University of Copenhagen, 1998.
- [5] ECKEL, Bruce. *Myslíme v jazyku C++: knihovna zkušeného programátora*. 1. vyd. Praha: Grada Publishing, 2006, 608 s. ISBN 80-247-1015-3.
- [6] HILDEBRAND, Antonín. *Návrh animačního systému pro 3D Engine*. [video] Programátorské večery [online]. Audiovizuální centrum studentů ČVUT, 2005 [cit. 2012-05-11]. Dostupné z WWW: <http://www.avc-cvut.cz/akce/mgr-antonin-hildebrand-illusion-softworks-navrh-animacniho-systemu-pro-3d-engine>
- [7] SEGAL, Mark, Kurt AKELEY. *The OpenGL Graphics System: A Specification (Version 4.2)*, 2011
- [8] EBERLY, David H. *Moving Along Curve with Specified Speed*. [online]. 2009 [cit. 2012-05-11] Dostupné z <http://www.geometrictools.com/Documentation/MovingAlongCurveSpecifiedSpeed.pdf>
- [9] KAW, Autar. *Romberg Rule*. [online]. 2009 [cit. 2012-05-11]. Dostupné z http://numericalmethods.eng.usf.edu/mtl/gen/07int/mtl_gen_int_txt_romberg.pdf
- [10] WGL and Windows Reference. *MSDN* [online]. 2012 [cit. 2012-05-11]. Dostupné z: <http://msdn.microsoft.com/en-us/library/ee872083%28v=vs.85%29>
- [11] LORAH, Tristan. *DirectX 10 Blend Shapes: Breaking the Limits*. GPU Gems 3 [online]. 2011 [cit. 2012-05-11]. Dostupné z http://http.developer.nvidia.com/GPUGems3/gpugems3_ch03.html

Příloha A – Formáty používaných souborů

Popis formátu souboru JK2

(tučným písmem jsou označeny bloky souboru, kurzívou popis)

hlavička(„JK2“)	– 3*char
verze	– 2*byte
počet meshů	– unsigned short
počet animací	– unsigned short
počet materiálů	– unsigned short
počet kostí	– unsigned short

MESH

nazev – délka názvu * char + ukončovací znak

Rozdělení podle typů:

- *NonGeom mesh:*

obalovy kvadr min	– 3*float
obalovy kvadr max	– 3*float
rotace (normalizovany kvaternion)	– 3*float
- *Geom mesh:*

ID materialu	– unsigned short
počet vertexu	– unsigned short
VERTEXY	
vertex	– 3*float
uv	– 2*float
počet faců	– unsigned short
FACY	
face	– 3*unsigned short
- *Skinned geom mesh:*

ID materialu	– unsigned short
počet vertexu	– unsigned short
VERTEXY	
vertex	– 3*float
uv	– 2*float
flag	– byte
váha + id	– max 4*float
počet faců	– unsigned short
FACY	
face	– 3*unsigned short
- *Collision mesh:*

ID materialu	– unsigned short
počet vertexu	– unsigned short
VERTEXY	
vertex	– 3*float

počet faci	– unsigned short
FACY	
face	– 3*unsigned short

INFO

typ kolize – byte

MATERIAL

nazev	– délka názvu * char + ukončovací znak
id typu	– unsigned short
nazev souboru zakladni textury	– délka názvu * char + ukončovací znak
flag	– byte
doplnkove textury (normal, height, specular)	– max 3*(délka názvu*char + ukončovací znak)

KOSTI

nazev	– délka názvu*char + ukončovací znak
id rodice(-1 není)	– int
pozice	– 3*float
rotace (normalizovany kvaternion)	– 3*float

ANIMACE

Ve verzi 0.5 není podporováno.

Popis formátu souboru JK2A

celkový počet framů	– int
počet klíčovaných kostí	– unsigned short

KLÍČE KOSTÍ

id kosti	– int
počet klíčů pozic	– int

KLÍČE POZIC

čas(frame)	– float
pozice	– 3*float
počet klíčů rotací	– int

KLÍČE ROTACÍ

čas(frame)	– float
rotace (normalizovany kvaternion)	– 3*float

Popis formátu souboru JK2MORPH

Hlavička („JK2MO“)	– 5*char
Verze	– 2*byte
Počet morph targetů (n)	– int
Počet vertexů v jednom morph targetu (m)	– unsigned short
Data offsetů	– n*m*4*float