VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

FÚZE PROCEDURÁLNÍ A KEYFRAME ANIMACE

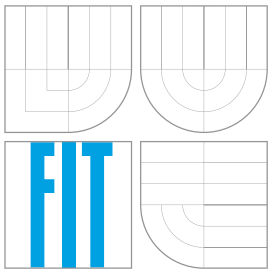
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN KLEMENT

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

FÚZE PROCEDURÁLNÍ A KEYFRAME ANIMACE

FUSION OF PROCEDURAL AND KEYFRAME ANIMATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN KLEMENT

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. LUKÁŠ POLOK

BRNO 2013

Abstrakt

Cílem této práce je implementace aplikace, v níž dochází k propojení procedurální a keyframe animace, s následnou vizualizací. Jádrem aplikace je spojení dvou různých technik animace použitých pro rozpohybování virtuální postavy. Základ pohybu je tvořen metodou klíčových snímků mezi nimiž dochází k interpolaci, a kterou v závislosti na prostředí dynamicky upravujeme pomocí animace procedurální, využívající přímé i inverzní kinematiky. Výsledkem spojením těchto dvou technik je postava interagující na okolní prostředí ve scéně. Aplikace je psána v jazyce C++, využívající knihovnu GLM, pro matematické funkce a k výsledné vizualizaci je použita knihovna OpenGL s rozšířením GLUT.

Abstract

The goal of this work is to create an application, which will combine procedural and keyframe animations with subsequent visualization. Composition of this two different animations techniques is used to animate a virtual character. To combine this two techniques one starts with interpolations from keyframe animation and then enchance them by procedural animations to properly fit into the characters surroundings. This procedural part of animation is obtained by using forward and inverse kinematics. Whole application is written in C++, uses GLM math library for computations and OpenGL and GLUT for final visualization.

Klíčová slova

OpenGL, 3D grafika, procedurální, keyframe, animace, skeletální animace, inverzní kinematika, CCD, skinning, MaxScript, vertex a fragment shader, bullet physics, bullet, kvaterniony

Keywords

OpenGL, 3D graphic, procedural, keyframe, animation, skeletal animation, inverse kinematic, CCD, skinning, MaxScript, vertex and fragment shader, bullet physics, bullet, quaternions

Citace

Martin Klement: Fúze procedurální a keyframe animace, diplomová práce, Brno, FIT VUT v Brně, 2013

Fúze procedurální a keyframe animace

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Lukáše Poloka. Uvedl jsem všechny literální prameny a publikace, ze kterých jsem čerpal

.....
Martin Klement
21. května 2013

Poděkování

Zde bych rád poděkoval Ing. Lukaši Polokovi za odborné vedení, konzultace, rady a připomínky při práci na projektu. Dále bych chtěl poděkovat všem, kteří při mně stáli a podporovali v době tvorby

Obsah

Kapitola 1

Úvod

V dnešní době dochází k velmi rychlému rozvoji informačních technologií, zrychlují se procesory, zvyšují se kapacity operačních paměti, výkony grafických karet a celkově dochází k vylepšování všech komponent uvnitř počítače. S tímto pokrokem jde ruku v ruce i pokrok v oblasti počítačové grafiky, proto si programátoři mohou dovolit zasadit do aplikace mnohem více modelů, či umožnit grafikům vytvářet je detailněji. Avšak s růstem detailnosti modelů a celkovým počtem trojúhelníku ve scéně se aplikace dostávaly do problémů v oblasti animací. U modelů, které se pohybovaly jako celek, nebyl tento problém až tak velký, na rozdíl od modelů lidí, zvířat, či jiných živočichů, u nichž při vytváření animace musí mít animátor možnost ohnout či posunout jednotlivé části modelu, bez toho aniž by ovlivnil zbylé vrcholy.

V dřívějších programech, nebyly modely až natolik propracované a skládaly se pouze z několika stovek vrcholů. Proto nebyl takový problém mít pro daný časový okamžik, uloženy aktuální souřadnice všech bodů. Jednotlivé animace nebyly ani nikterak dlouhé, většinou se jednalo pouze o krátké cykly, které se stále opakovaly (například chůze, běh, atd.). V moderních aplikacích, již není problém zobrazovat modely o tisících či deseti tisících trojúhelníků, avšak takto zaznamenané animace by zabíraly nepřijatelné množství místa v paměti a proto byla zavedena technika skeletální animace, tvořící základ na němž tato práce staví a dále ho rozvijí. Skeletální animace už svým názvem předpovídá své užití, namísto pohybování všemi vrcholy objektu, je uvnitř zasazena kostra, která odpovídá za pohyb přichyceného modelu. Čímž odpadá potřeba znalosti aktuálních souřadnic všech vrcholů v daném časovém okamžiku, a stačí nám pouze uchovávat transformace jednotlivých kostí. Svým způsobem je tato technika velmi podobná kosternímu systému u reálných živočichů, kde je každý vrchol ovlivňován pouze kostmi v jeho blízkosti, a čím blíže k některé z nich je, tím větší vliv na něj má.

Díky kostře, k níž je model přichycen, jsme schopni animace jednoduše vytvářet. Nejčastějším způsobem tvorby animací je diskrétní zaznamenání polohy a natočení kosti. Těmto záznamům říkáme keyframe (neboli klíčové snímky), mezi kterými následně interpolujeme. Jedná se o způsob, který nevyžaduje definovat pohyb kosti pro každý snímek časové osy, ale postačí nám jen několik záznamů kdy dochází k změně směru či rychlosti pohybu. Opačnou od diskrétní techniky keyframe je animace procedurální, kdy za pohyb každé kosti odpovídá její procedura (funkce). Tato funkce musí být spojitá, tzn. že pro každý časový okamžik vrací odpovídající nastavení pozice a natočení kosti. Proto se procedurální animace využívá převážně pro krátké pohybové smyčky (např. chůze, běh a jiné).

Mohlo by se zdát, že procedurální animace je pouze přežitek, jelikož pomocí animace založené na klíčových snímcích je možné vytvořit jakoukoliv sekvenci pohybů, navíc v mno-

hem kratším čase a jednodušeji. Avšak problémem animace složené z klíčových snímků je její jedno-účelnost, pro scénu kde byla vytvářena může vypadat dokonale, avšak pokud stejnou animaci zasadíme do zcela jiného okolí, pak se dostaneme do problémů, kdy například námi zobrazená postava prochází terénem, či se naopak odráží od vzduchoprázdna. S tímto problémem se setkáváme hlavně v počítačových hrách, kdy je pro každý pohyb postavy vytvořena pouze krátká smyčka, která je ve hře opakována. Odstranění tohoto problému řeší právě procedurální animace, která aktivně reaguje na prostředí ve scéně. Například při chůzi či běhu, nastavuje výšku chodidla v závislosti na terénu pod ním. Přidáním této interakce postavy s terénem, dojde ke zlepšení vizuální stránky celé aplikace a přispěje k celkovému dojmu z ní nabytého. Spojení právě těchto dvou stylů animace je hlavním tématem této práce. Procedurální animace je možné využít i pro plynulý přechod z jedné pohybové sekvence do druhé.

Následující kapitola práce se podrobně věnuje matematickým principům zobrazování scény ve 3d prostoru s využitím maticových transformací a jejich vylepšení, v rámci rotací objektů, v podobě kvaternionů. Dále jsou v textu podrobně probrány obě animační techniky, spolu s popsáním způsobu jejich vytvoření. Po získání základního přehledu v možnostech animace a pochopení jejich principů, je přistoupeno k jádru práce, tvořeném zapojením procedurální animace do techniky založené na klíčových snímcích. Vzhledem k narůstajícímu vlivu vizuální stránky vytvářených aplikací je poslední kapitola věnována právě vizualizace, jenž v dnešní době velkou měrou ovlivňuje celkový úspěch či neúspěch aplikace u koncových uživatelů.

Kapitola 2

Transformace

V 3d grafických aplikacích je velmi často potřeba transformovat vektory, udávající body v prostoru, mezi různými souřadnicovými systémy. Například souřadnice jednotlivých vrcholů modelu jsou uloženy ve svém objektovém prostoru, avšak pro výsledné zobrazení na obrazovku počítače, je potřeba tento systém transformovat do prostoru kamery. Tato kapitola se proto zaměřuje na vysvětlení matematického principu transformací a jejich maticového zpracování. V kapitole je čerpáno z [8], [9] a [16].

2.1 Lineární transformace

Lineární transformace je zobrazení z jednoho vektorového prostoru do druhého, které zachovává lineární kombinace. Což znamená, že musí splňovat následující dva požadavky:

- $$\forall v, w \in V, \text{ platí } T(v+w) = T(v) + T(w) \tag{2.1}$$
- $$\forall v \in V, r \in R, \text{ platí } T(rv) = rT(v) \tag{2.2}$$

Použití homogenních souřadnic nám umožňuje pracovat se všemi druhy základních transformací jednotně, pomocí maticového zápisu. Jelikož je skládaní transformací v maticovém zápisu reprezentováno násobením, jehož výsledkem je opět matice, je zřejmé, že i libovolnou složenou transformaci lze zapsat pomocí jediné matice. Operace násobení u matic není komutativní, proto je nutné brát v úvahu jejich pořadí při skládání. Základními lineárními transformacemi jsou translace, rotace, zkosení a změna měřítka. Translace představuje posun souřadného systému o určitý vektor a v maticovém zápisu má následující tvar:

$$T_v p = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + v_x \\ p_y + v_y \\ p_z + v_z \\ 1 \end{bmatrix} \tag{2.3}$$

Další transformací je otočení souřadného systému okolo jeho počátku. Maticový zápis rotace kolem jednotlivých os je reprezentována takto

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

$$R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

$$R_z(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.6}$$

Poslední často užívanou transformací je změna měřítka daná maticí

$$S_v p = \begin{bmatrix} v_x & 0 & 0 & 0 \\ 0 & v_y & 0 & 0 \\ 0 & 0 & v_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} v_x p_x \\ v_y p_y \\ v_z p_z \\ 1 \end{bmatrix} \tag{2.7}$$

2.2 Kvaterniony

Kvaterniony jsou další matematickou alternativou, jak lze v 3d grafice reprezentovat rotace. Jejich využití je především u aplikací obsahující skeletální animace a inverzní kinematiky. Výhodou kvaternionů oproti rotačním maticím je kromě menšího prostoru které zabírají, i jejich rychlejší zřetězení, neboť je k němu potřeba méně aritmetických operacím, ale hlavně jejich interpolace vede při vizualizaci k hladšímu průběhu animace. Kvaternion je reprezentován 4-složkovým vektor mající následující tvar

$$q =< w, x, y, z >= w + xi + yj + yk \tag{2.8}$$

Také bývají často zapsány ve tvaru $q = w + v$, kde w reprezentuje skalární část odpovídající w-složce z q, a v reprezentuje vektorovou část odpovídající x, y a z složkám z q. Můžeme tedy říct, že kvaternion je složen z úhlu otočení s okolo osy v. Sada kvaternionů je přirozeným rozšířením komplexních čísel. Násobení je definováno distributivním zákonem s následujícími pravidly pro násobení imaginárních částí i, j a k, avšak není komutativní a proto musíme dávat pozor na správné uspořádání násobených elementů.

$$i^2 = j^2 = k^2 = -1 \tag{2.9}$$

$$ij = -ji = k \tag{2.10}$$

$$jk = -kj = i \tag{2.11}$$

$$ki = -ik = j \tag{2.12}$$

2.2.1 Konverze mezi kvaterniony a maticemi

Přestože je výpočet násobení matic pomalejší a interpolace mezi kvaterniony vytváří hladší průběh animace, mají matice stále své využití. Typickým příkladem, ve kterém jsou upřednostňovány matice před kvaterniony je transformace vrcholů, kdy většina dnešních 3d API pracuje právě s rotacemi ve formě matic. Nyní si přiblížíme jak dochází k převodu kvaternionů na matice a obráceně. Proto abychom mohli kvaternion převést na odpovídající

rotační matici, musíme vyjádřit operaci $qq'q^{-1}$ (kvaterniony q'' a q' jsou odlišné od q, nejedná se tedy o jeho derivaci jak by se mohlo zdát) pomocí maticových operací. Násobení kvaternionů vypadá následovně

$$q'' = [w, v][w', v'] = [ww' - v \cdot v', v \times v' + wv' + w'v] \tag{2.13}$$

kde $\cdot$ představuje skalární součin a $\times$ je vektorový součin, vztah jenž je dále rozšířen o $[x'', y'', z'', w'']$ takto

$$x'' = yz' - zy' + wx' + xw' \tag{2.14}$$
$$y'' = zx' - xz' + wy' + yw' \tag{2.15}$$
$$z'' = xy' - yx' + wz' + wz' \tag{2.16}$$
$$w'' = ww' - xx' - yy' - zz' \tag{2.17}$$

Rozšíření můžeme vyjádřit pomocí maticového násobení, čímž získáme

$$\begin{bmatrix} w & -z & y & x \\ z & w & -x & y \\ -y & x & w & z \\ -x & -y & -z & w \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = L_q q' \tag{2.18}$$

Steným postupem můžeme výraz $q'' = q'q$ maticově vyjádřit jako

$$\begin{bmatrix} w & z & -y & x \\ -z & w & x & y \\ y & -x & w & z \\ -x & -y & -z & w \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = R_q q' \tag{2.19}$$

Pro kvaternion $q = [w, v], q^{-1} = [w, -v]/N(q)$. $N(q) = w^2 + x^2 + y^2 + z^2 = 1$ pro jednotkové kvaterniony, takže $q^{-1} = [w, -v]$. Pak po nahrazení q ya q' získáme matici

$$\begin{bmatrix} w & -z & y & -x \\ z & w & -x & -y \\ -y & x & w & -z \\ x & y & z & w \end{bmatrix} = R_q^{-1} \tag{2.20}$$

Ekvivalentní matici k operaci $qq'q^{-1}$ získáme spojením matic L_q a R_q^{-1}

$$M = L_q R_q^{-1} = \begin{bmatrix} w & -z & y & x \\ z & w & -x & y \\ -y & x & w & z \\ -x & -y & -z & w \end{bmatrix} \begin{bmatrix} w & -z & y & -x \\ z & w & -x & -y \\ -y & x & w & -z \\ x & y & z & w \end{bmatrix} = \tag{2.21}$$

Z níž po roznásobení a následném zjednodušení získáme matici, jenž odpovídá původnímu kvaternionu

$$\begin{bmatrix} 1 - 2(y^2 + z^2) & 2(xy - wz) & 2(wy - xz) & 0 \\ 2(xy + wz) & 1 - 2(x^2 + z^2) & 2(yz - wx) & 0 \\ 2(xz + wy) & 2(yz + wx) & 1 - 2(x^2 + y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = M \tag{2.22}$$

2.2.2 Konverze rotační matice na kvaternion

Vztah mezi rotační maticí a komponenty jí odpovídajícím kvaternionem jsou ukázány v předcházejícím tvrzení, z něhož můžeme odvodit následujících šest vztahů.

$$M_{1,2} + M_{2,1} = 4xy \tag{2.23}$$
$$M_{3,2} + M_{2,3} = 4yz \tag{2.24}$$
$$M_{1,3} + M_{3,1} = 4xz \tag{2.25}$$
$$M_{2,3} - M_{3,2} = 4wx \tag{2.26}$$
$$M_{3,1} - M_{1,3} = 4wy \tag{2.27}$$
$$M_{1,2} - M_{2,1} = 4wz \tag{2.28}$$

Ze zmíněných vztahů je navíc patrné, že pokud budeme znát alespoň jednu komponentu pak jsme schopni ostatní tři postupně dopočítat. Při hledání se snažíme určit právě tu komponentu, jenž má největší absolutní hodnotu. Pokud je zadaný kvaternion jednotkový pak jeho největší komponenta představuje absolutní hodnotu vždy větší jak 0.5. Jako první se pokusíme nalézt řešení pro komponentu w. K určení její absolutní hodnoty začneme s výpočtem sumy komponent, které leží na diagonále matice M, jenž vypadá následovně

$$Tr = 4 - 4(x^2 + y^2 + z^2) = 4(1 - (x^2 + y^2 + z^2)) \tag{2.29}$$

Připomeňme že jednotkový kvaternion $q = [w, v] = [cos(\theta), v' sin(\theta)]$ kde $v' = (x', y', z')$ je jednotkový vektor. Tudíž výraz můžeme vyjádřit jako

$$Tr = 4(1 - (x^2 + y^2 + z^2) sin^2(\theta)) \tag{2.30}$$

A jelikož (x', y', z') je jednotkovy vektor, pak $x'^2 + y'^2 + z'^2 = 1$ odkud získáváme vztah

$$Tr = 4(1 - sin^2(\theta)) = 4(cos^2(\theta)) = 4w^2 \tag{2.31}$$

$$|w| = Tr^{\frac{1}{2}}/2 \tag{2.32}$$

Pakliže je $Tr \geq 1$, dosadíme $4w = \pm 2Tr^{\frac{1}{2}}$ do rovnic 4, 5 a 6 a zjistíme hodnoty jednotlivých komponent.

$$x = (M_{2,3} - M_{3,2})/2Tr^{\frac{1}{2}} \tag{2.33}$$
$$y = (M_{3,1} - M_{1,3})/2Tr^{\frac{1}{2}} \tag{2.34}$$
$$z = (M_{1,2} - M_{2,1})/2Tr^{\frac{1}{2}} \tag{2.35}$$

Použití kladného či záporného základu Tr nehraje žádnou roli, jelikož kvaterniony q a $-q$ reprezentují identické natočení. Pokud však $|w| < 0.5$, pak můžeme ze zbylých komponent určit největší, pomocí prvních třech hodnot na diagonále matice M. Jako příklad vezměme předpoklad že $M_{2,2} > M_{1,1}$. Což vyjádříme jako

$$1 - 2x^2 - 2z^2 > 1 - 2y^2 - 2z^2 \tag{2.36}$$
$$-2x^2 > -2y^2 \tag{2.37}$$
$$|x| < |y| \tag{2.38}$$

Stejným způsobem bychom mohli porovnat i zbylé hodnoty na diagonále. Proto komponenta s největší absolutní hodnotou zároveň odpovídá i největší hodnotě uvnitř vektoru x, y, z. Pokud již máme největší kompomentu odečteme od ní zbylé dvě a rovnice spojíme do jediného vztahu. Pro příklad předpokládejme, že největší komponentou byla $M_{2,2}$

$$M_{2,2} - M_{3,3} - M_{1,1} = 1 - 2x^2 - 2z^2 - (1 - 2y^2 - 2z^2) - (1 - 2x^2 - 2y^2) = 4y^2 - 1 \quad (2.39)$$

Nebo-li obecně

$$v_i = \pm(M_{ii} - M_{jj} - M_{kk} + 1)^{\frac{1}{2}}/2 \quad (2.40)$$

Stejně jako u výpočtu komponenty w, nehraje roli základ který použijeme. Tudíž po zjistění odpovídající hodnoty v_i, pak dosazením získáme hodnoty v_j, v_k a w

$$v_j = (M_{ij} + M_{ji})/(4v_i) \quad (2.41)$$
$$v_k = (M_{ik} + M_{ki})/(4v_i) \quad (2.42)$$
$$w = (M_{jk} - M_{kj})/(4v_i) \quad (2.43)$$

2.2.3 Interpolace mezi kvaterniony

Jelikož jsou kvaterniony reprezentovány čtyř složkovými vektory je možné pomocí interpolace mezi nimi docílit hladkého přechodu. Při animaci objektu je možnost interpolování mezi natočením velmi důležitá, především u techniky klíčových snímků, jenž se skládá z diskrétního popisu pohybu. Nejjednodušším typem interpolace je lineární interpolace. Pro dva jednotkové kvaterniony q_1 a q_2, pak vztah vypadá následovně

$$q(t) = q_1(1 - t) + q_2 t \quad (2.44)$$

Funkce $q(t)$ sice představuje hladký přechod z vektoru q_1 do q_2, avšak nezachovává výsledný vektor normalizovaný, což můžeme opravit a získáme tento vztah

$$q(t) = \frac{q_1(1 - t) + q_2 t}{||q_1(1 - t) + q_2 t||} \quad (2.45)$$

Lineární transformace je velmi rychlá, avšak není pro interpolaci dvou rotačních vektorů ideální, jelikož nevytváří hladkou animaci. Vektor sice správně opisuje dráhu, po které cestuje, ale nezachovává v jejím průběhu konstantní rychlost. V počáteční a koncové fázi interpolace, je její změna pomalá a směrem ke středu naopak rychlost příliš vzroste.

Musíme proto využít techniku sférické lineární interpolace (zkráceně *SLERP*), která zachovává jak jednotkovou délku vektoru tak i konstantní rychlost změny úhlu v průběhu celé animace. Vztah vytvářející plynulý přechod mezi kvaterniony vypadá následovně

$$q(t) = \frac{sin(\theta)(1 - t)}{sin(\theta)} q_1 + \frac{sin(\theta)t}{sin(\theta)} q_2 \quad (2.46)$$

Kapitola 3

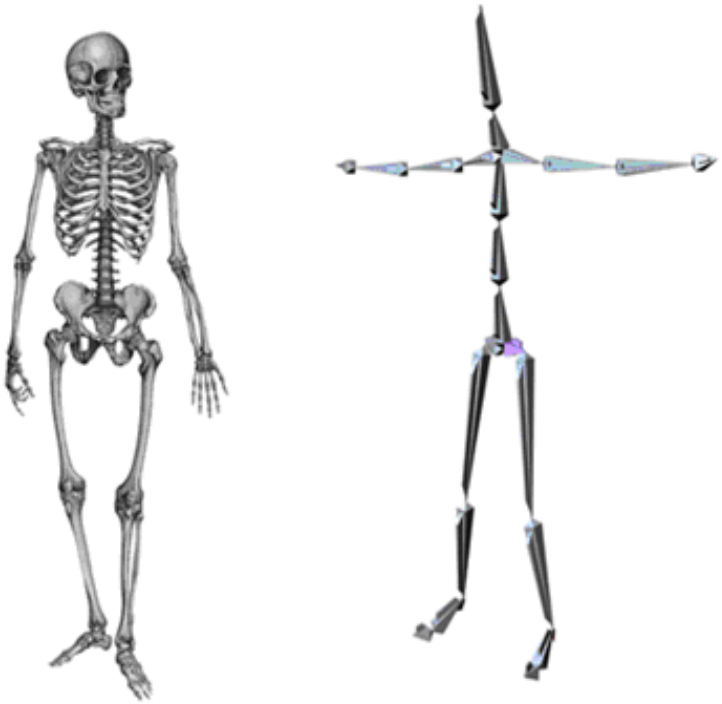
Animace

Dnes již není problém zobrazit 3D model složený z tisíce trojúhelníků. V počítačových hrách či v jiných odvětvích softwarového průmyslu je třeba, aby se objekty mohly pohybovat v reálném čase. Problém nastává u animace složitějších objektů, kde se nepohybují všechny jeho body jako celek, ale dochází k ohýbaní určité jeho části. Pak by pro všechny body modelu musely být uloženy jejich aktuální souřadnice v daném čase, což je velmi nepraktické. Pro tyto případy se používá skeletální animace kde, jak už sám název napovídá, je základem skelet neboli kostra, podle které dochází k animaci objektu a jednotlivými vrcholy objektu je hýbáno podle kostí, které jsou v jejich blízkosti. Čím blíže se nachází bod v okolí některé kosti, tím více na ní závisí jeho pohyb.

Princip pohybu virtuální kostrou je obdobný kosternímu systému u živých tvorů na Zemi, proto předtím než budeme vytvářet animaci, ať už člověka či jiného živočicha, musíme prozkoumat jeho kosterní systém. V případě špatně vytvořené kostry může docházet k nerealistickým pohybům a tím i k nerealističnosti celé animace. Skeletální animace je dnes nedílnou součástí animace, pomocí níž jsou rozpohybovány různé modely lidí, zvířat, či jiných tvorů, které grafik vytvoří. Tento systém není využít pouze pro postavy ale lze jej použít i pro vhodné objekty, které v reálném světě žádnou kostru nemají. Příkladem může být třeba obyčejný luk, kde při jeho napínání dochází k ohybu a po vystřelení se luk znovu napne. Pro ohyb lučiště nám stačí nahradit každou jeho stranu dvěma či více kostmi a máme vytvořeno reálné ohýbání. Kosterní animaci však nemůžeme využít ve všech případech, jako je různé vlnění či deformace. Proto existují i jiné způsoby, pomocí kterých jednoduše dosáhneme požadovaného výsledku. Ty však nejsou obsahem této práce.

3.1 Kostra

Jak už bylo zmíněno výše, základem skeletální animace je kostra. Kostra se skládá z kostí, které jsou mezi sebou spojeny klouby. Ty jsou obyčejně pouze rotující, ale mohou se i posouvat. Posouvání kloubů se využívá hlavně pro natahování končetin či jiných údů u kreslených nebo fantasy postav. Každý kloub může rotovat ve třech ortogonálních směrech, kterým se říká stupně volnosti kloubu (DOF). U detailní kostry lidského těla je napočítáno kolem 200 stupňů volnosti. Avšak ne všechny jsou využívány, protože každý kloub má i svá omezení na rotaci. Například loket využívá pouze jeden stupeň volnosti. Avšak běžná kostra, pro kterou se vytváří animace, nemá ani zdaleka tolik kostí jako má skutečná. Dochází k určitému vyabstrahování některých kostí, které nejsou důležité, či spojení více kostí do jedné.



Obrázek 3.1: Porovnání reálné[20] a vyabstrahované virtuální kostry

Struktura kostry musí odpovídat kosternímu systému modelovaného tvora, zároveň by měla být co nejlépe umístěna uvnitř modelu a všechny kosti musí být správně natočené. Při podcenění této fáze může dojít k nesprávným deformacím a různým jiným chybám při rozpohybování modelu. Problémovým místem mohou být například paže a jejích natočení, kdy loket kostry směřuje k zemi, ale povrch modelu postavy má loket nasměrován dozadu. Tento omyl se může stát, jelikož kosti jsou reprezentovány jen kvádry a pokud má kostra upažené ruce není na první pohled patrné, kam míří loket. Po umístění kostry se dostáváme k navázání bodů k příslušným kostem čemuž se věnuje následující podkapitola.

3.2 Kinematika

Pro postavení kostry do některé z póz, potřebujeme mít možnost pohybovat či rotovat s kostmi. Nejjednodušší možností je přímá kinematika (*forward kinematic*).

3.2.1 Přímá kinematika

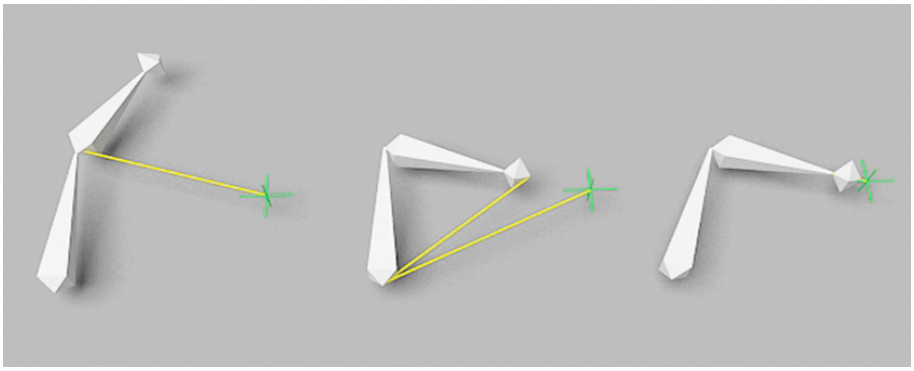
Každá kost je definována v prostoru pouze tím, kde končí její předek a jak je natočená v jednotlivých osách, a přímá kinematika mění právě tyto parametry. Příkladem jejího využití může být kost, která reprezentuje lebku a kde nepotřebujeme nic jiného než ji různě natáčet. Pohyb kostí ovlivňuje nejen její natočení ale i natočení celé struktury z ní vycházející. Přímá kinematika se využívá převážně jen pro páteř a konce končetin. Druhou možnou kinematikou je inverzní kinematika (*inverse kinematic*), ve které je navíc určitý

bod v prostoru (dále cíl), pro který se snažíme zjistit natočení kosti tak, aby poslední kost končila na souřadnicích cíle.

3.2.2 Inverzní kinematika

Fakt, že inverzní kinematika se nepočítá pouze pro jednu kost, ale pro libovolný počet vytváří velký rozdíl od kinematiky přímé. Jedná se o rychlou a robustní metodu a právě díky rychlosti muže být využita i v real-time hrách, například pro reagování na nepředvídatelné prostředí či uživatelský vstup. Inverzní kinematika může být počítána různými metodami, zde si uvedeme pouze metodu *Cyclic coordinate descent*[4], pomocí které je v projektu inverzní kinematika počítána. Jedná se o jednoduchý a rychlý algoritmus, který je i překvapivě robustní.

Každá iterace začíná u kosti, která je nejhlouběji v řetězci kostí. Tato kost je natočena tak, aby směřovala přímo na cílový bod (dále už jen cíl). Poté je otcovská kost natočena o rozdíl uhlů mezi imaginárními čarami, první od počátku otcovské kosti do konce kosti, kterou jsme natočili v minulém kroku a mezi čárou vycházící také z bodu počátku otce, ale končící v cíli. Tento postup je dále opakován pro každou kost v řetězci. Po několika iteracích přes řetězec má každá kost nastavený úhel tak, že konec řetězce je dostatečně blízko cíle. Pro animaci je důležité, které kosti budou v řetězci, pro něž se má kinematika počítat. Pokud by se jednalo o pohyb celé ruky při chůzi, tak je lepší počítat kinematiku pouze pro kosti od ramene k zápěstí. Kdybychom do výpočtů zahrnuli i kosti až po konec některého z prstů, tak by mohlo dojít k ne zcela realistickému natočení kosti. To je způsobeno tím, že nejspodnější kost je nejvíce dynamická. Proto je důležité posoudit, o jaký pohyb se bude jednat a podle toho nastavit, pro které kosti se kinematika bude počítat.



Obrázek 3.2: Ukázka jedné iterace algoritmu cyclic coordinate descent a výsledku po třech krocích iterace

Počet iterací nutných k dosažení cíle (nebo alespoň přijatelného) záleží na počátečním stavu kostí před začátkem řešení kinematiky. Jednou z možností je pamatování si určitého výchozího stavu, do kterého kosti vždy před výpočtem nastavíme. Jinou možností je, že výpočet bude vycházet ze stavu kostí, ve kterých zrovna jsou. V určitých případech je toto řešení výhodnější, například pokud se cílový bod posunul pouze o krátkou vzdálenost, dosáhne toto řešení výsledku rychleji. Avšak nejde o tak výrazný rozdíl oproti počítání kinematiky z výchozího stavu. Jelikož v prvních pár iteracích dochází k největšímu přiblížení se k cíli a postupně se přibližování zmenšuje. Pro vytvoření realistického pohybu kloubu je důležité omezení rotací. Lidské klouby také nedokáží rotovat zcela libovolně, například

lidský loket je omezen pouze na rotace v jedné ose a pouze v omezeném rozmezí od 15^o do 180^o. Proto, pokud chceme vytvářet realistickou animaci, je potřeba tato omezení nastavit.

3.3 Metody tvorby animace

Po vytvoření kostry a přichycení povrchu modelu na ni, můžeme začít animovat. Samotná animace sice může vypadat na první pohled jednoduše, ale opak je pravdou. Samozřejmě jde o délku a hlavně děj animace. Avšak i pouhé vytvoření cyklu chůze, které vypadá zcela realisticky, je velmi složité a je nutné všímat si i maličkostí a zapojení jednotlivých kostí do pohybu celého těla. Neméně důležitou součástí je neustálé hlídání váhy těla. Pokud by postava nesla těžké břemeno, musí animace chůze vypadat jinak, než když půjde s prázdnýma rukama. Grafiků, kteří umí vytvořit na první pohled dokonalé modely je už i v naší zemi celkem dost, avšak kvalitních animátorů je stále málo, nejen u nás, ale i ve světě. Hlavním důvodem tohoto nepoměru je složitost a pracnost vytváření animací. Dobrý animátor si musí všímat i sekundárních pohybů, které člověk při chůzi či jiné aktivitě podvědomě vykonává, neustále kontrolovat časování a anticipaci (předvídání) dalších pohybů a zčásti je nutné mít i talent.

Další možností pro vytvoření animací postav je stále více využívané motion-capture (*mocap*), který celý proces vývoje animace velmi urychluje. Pomocí mocapu je možné vytvořit sekvenci pohybů během několika sekund (bez započtení režie na připravení snímání), kde tu samou sekvenci by i ti nejlepší animátoři vytvářeli několik hodin. Jedná se o proces, kdy je nahráván reálný pohyb, a ten je následně přenesen na digitální model. Největším negativem tohoto přístupu je stále jeho cena a s tím související nízká rozšířenost ve veřejnosti.

S tímto problémem se snaží vyrovnat firma Animazoo[11]. Už jejich podtitulek „The future of motion capture“ souvisí se snahou minimalizovat potřebné podmínky a nároky na pořízení mocap systému. Oproti jiným mocap systémům, nejsou potřeba žádná složitá studia obsahující desítky kamer, ale stačí si pouze obléci oblek, do které jsou v místech kloubů všity senzory a postavit se na pár sekund do rozpažené pózy pro počáteční synchronizaci s kostrou modelu. Čímž je veškerá režie hotová. Samotné snímání aktuálního natočení a akcelerace v jednotlivých osách x, y a z je prováděno pomocí akcelerometrů. Princip je stejný jako u senzorů v moderních telefonech či tabletech, kde základ tvoří gyroskop a vestavěný kompas. Protože všechny senzory snímají nezávisle, říká se gyroskopickým systémům také „inerciální snímače pohybu“.

Avšak i přes takovéto moderní systémy je někdy nezbytné ruční vytváření či editace. Ať už jen pro doladění animací získaných z mocapu, či pro animace natolik jednoduché nebo krátké, že by se nevyplatilo využít tento systém (časově či cenově). Ruční klíčování někdy také může být jedinou možností pro vytváření animace pohybu, například tvorů, které v reálném světě již neexistují nebo ani neexistovali (draci a jiné příšery).

Pro tvorbu animací jsou v dnešní době nejvhodnější ty stejné programy, které se využívají pro modelování 3D objektů (3d Studio Max, Maya, Blender). Ty obsahují různé nástroje a modifikátory usnadňující samotné vytváření. Avšak nejrozšířenějším nástrojem pro animace je Maya, která k tomuto účelu byla od počátku vyvíjena. V animování je důležitý element čas. Postava se musí pohybovat neustále se stejnou dynamikou. Pokud by každý krok prováděla se zcela jinou dynamikou a rychlostí nevypadal by pohyb přirozeně. Jako nejlepší způsob při tvoření animace se uvádí ten, kde nejdříve v zajímavých okamžicích celého děje vytvoříme pózy a následně zkontrolujeme dynamiku a návaznost mezi nimi. Pokud nám připadá v pořádku, doplníme zbytek animace mezi pózami. Na stejném principu

pracuje i keyframe animace.

3.3.1 Keyframe animace

Keyframe animace je styl založený na klíčových snímcích a postupné interpolaci mezi nimi, proto už nemusíme mít zaznamenané pro jednotlivé kosti pozice a orientaci v každém časovém okamžiku, ale postačí nám pouze několik záznamů pro každou kost. Samotný výběr klíčových snímků je celkem jednoduchý a přirozený. Jedná se většinou o ty, kdy u objektu dochází ke změně směru či rychlosti pohybu. Pro ilustraci vezměme příklad, kdy postava upaží levou ruku. Pro zaznamenání takovéhoto pohybu stačí poznamenat dva klíčové snímky. První, kdy je ruka podél těla a druhý v době, kdy dochází k upažení. Všechny snímky mezi nimi doplníme interpolací mezi nimi. Z tohoto příkladu je jasná paměťová nenáročnost oproti zaznamenávání každého snímku.

Tímto způsobem jsme schopni vytvořit jakoukoli animaci a mohlo by se zdát, že ani není potřebné vymýšlet jiný způsob. Z části je tato úvaha správná. Pokud bychom chtěli vytvořit animaci, která se použije pouze v jedné dané scéně a tím její úloha končí, pak nám tento postup opravdu stačí. Ale představte si situaci, že potřebujeme animaci znovu použít, avšak v jiném prostředí, nebo pouze změnit váhu předmětu, který postava nese. Pak by takto vytvořená animace neodpovídala realitě. Docházelo by například k propadání noh do terénu, či opření se o zeď, která v nové scéně vůbec není. S těmito problémy se nejčastěji můžeme setkat u počítačových her, kde pohyb postavy je složen z několika krátkých sekvencí pohybů, které se střídají v závislosti na uživatelském vstupu. Řešením je procedurální animace, díky které jsme schopni upravit pohyb tak, aby ve zmíněném příkladu noha neprocházela terénem, ale zůstala na jeho povrchu.

3.3.2 Procedurální animace

Jedná se o způsob, kdy je výsledná animace generována v reálném čase a dovoluje nám některé věci, které by byli pomocí předefinované animace jen těžko realizovatelné či dokonce vůbec. Každé kosti odpovídá jedna funkce, které v závislosti na čase a okolí vrací odpovídající natočení kosti. Avšak i vytvoření pouze jednoho komplexnějšího pohybu (např. model chůze), je časově mnohem náročnější, než je tomu u předdefinované sekvence. Výhodou je však možnost upravení výsledného natočení v závislosti na okolnostech. Proto se dnes v průmyslu počítačových her používá jejich kombinace. Základ tvoří animace klíčových snímků, kterou se vytvoří celá sekvence a podle okolního prostředí se pouze poupraví procedurální animací.

Vezměme si jako příklad již zmíněný model chůze. Pokud postava půjde po rovném terénu, není třeba animaci z klíčových snímků nikterak upravovat. Avšak v případě členitého terénu musíme zamezit odražení se ze vzduchu respektive projití skrz terén. Proto výšku chodidla procedurálně upravíme, tak aby odpovídala výšce terénu, a částečně upravíme výšku pivotu modelu.

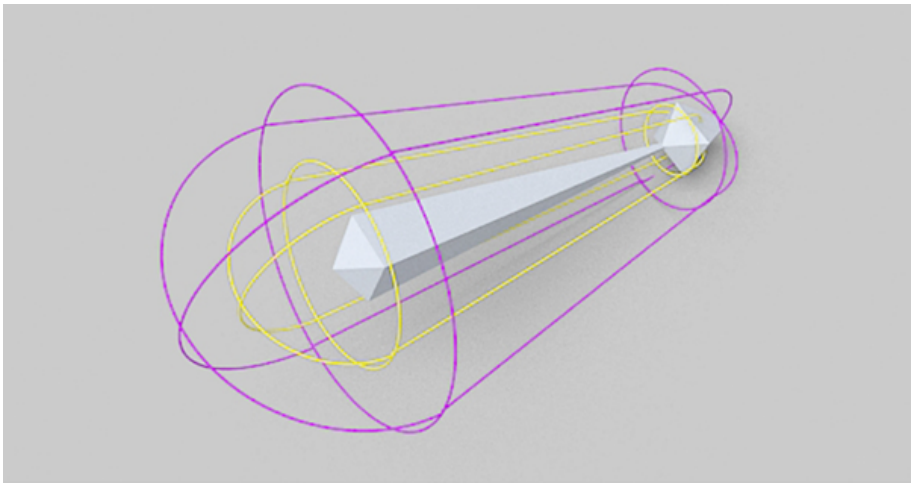
3.4 Skinning

Českým překladem slova skinning by mohlo být navléknutí kůže, avšak v české 3D grafice se žádný překlad nepoužívá, a proto i v této prácí zůstáváme u anglického slova skinning[12]. Jak již je uvedeno výše, je velmi důležité správné umístění kostry uvnitř modelu. Samotné

přichycení modelu se ve většině případů dělá v prostředí 3D editačního softwaru, které je tomuto způsobu přizpůsobeno (např. pomocí modifikátoru Skin v 3ds Max a Maya).

Po aplikování modifikátoru uživatel musí vybrat kosti, pro které se skinning bude aplikovat (pokud by uvnitř scény bylo více kosterních systémů, je třeba vybrat odpovídající). Ve druhé fázi, která je jádrem modifikátoru, dochází ke spočítání vah všem vrcholům modelu. Váhy udávají, hodnotu, kterou jsou body ovlivňovány jednotlivými kostmi a součet všech vlivů na jeden vrchol je roven jedné. Pokud bychom měli jednoduchý objekt, složený jen z několika bodů, mohli bychom každému bodu nastavit jednotlivé váhy ručně. Ale co v případě, že bychom potřebovali nastavit váhy pro model, který má desítky tisíc bodů? Z tohoto důvodu bylo třeba vymyslet algoritmus, který by toto provedl za nás.

Dnes nejrozšířenějším algoritmem je metoda obálek. Každá kost má svojí působnost na body kolem sebe a těmto okolím říkáme obálky. První obálka kosti je vnitřní, a všem bodům uvnitř ní nastaví vysokou váhu na sebe. Druhé obálce se říká vnější, je větší a tudíž působí i na vertexy, které jsou více vzdáleny. Ale čím jsou vzdálenější, tím nižší vliv na ně má, a pokud jsou vzdáleny až za rozsah obálky, nejsou už na tuto kost vázány. U každé kosti je umožněno nastavovat rozměr rozsahu vnější i vnitřní obálky jak na začátku kosti, tak u konce. Toto řešení výborně funguje pro vrcholy, které nejsou na hraně mezi kostmi. Ale právě pro body, které jsou v blízkosti více kostí, se může stát, že nalezené řešení není zcela ideální. Tudíž použití obálek je používáno pouze na počáteční nastavení vah. V případě, že chceme docílit dokonalé deformace je nutné váhy některých bodů do-upravit ručně. Ale ani to někdy nemusí stačit. Proto jsou v dnešních 3D modelačních nástrojích doplňující funkce, pomocí kterých jsou vytvářeny sekundární deformace. Příkladem může být napínání svalů, kdy při napnuté paži jsou svaly natažené a při ohybu paže dochází k jejich smrštění a zároveň k zvětšení oblasti bicepsu.



Obrázek 3.3: Ukázka vlivu kosti na vrcholy povrchu modelu, pomocí algoritmu obálek

Alternativní možností může být malování vah. Tento způsob je podobný ručnímu nastavování, jen je rychlejší, jelikož nemusíme vybírat jednotlivé vertexy a psát ručně váhy, ale vybereme si některou kost a kreslíme po modelu barvou. Čím tmavší barva je v okolí vertexu, tím větší na něj má vliv (barevné značení se může v různých aplikacích lišit například ve 3D Studiu Max je použita červená barva jako nejvíce vlivná a naopak nejméně vlivná je modrá).

Kapitola 4

Fúze procedurální a klíčové animace

Kapitola pojednává o spojení procedurální a klíčové animace. Samotný princip spojení je celkem intuitivní. Nejdříve v 3D editačním softwaru před-vytvoříme animaci některého z možných pohybů postavy v čisté scéně. Čistá scéna odpovídá pouze holému nečlenitému terénu, nejlépe zcela rovnému. Pouze pokud by v ději vytvářené animaci, mělo docházet k interakci postavy s určitým předmětem (například otevření dveří, okna a jiných), je třeba, aby právě tyto předměty byly ve scéně obsaženy. Důvodem začlenění externích objektů do čisté scény je nejen praktické hledisko, jenž animátorovi umožňuje jednodušší vytvoření pohybu, ale také z důvodu snazšího *real-time* upravení před-vytvořené animace pomocí inverzní kinematiky.

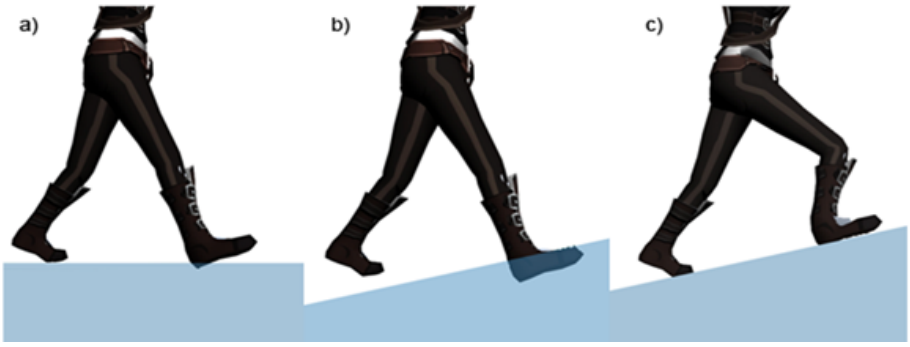
Při absenci interakce postavy a prostředí, ve kterém se pohybuje, by mohlo docházet k nerealistickému pohybu po členitém (či pouze nakloněném) terénu. Proto je nezbytné pomocí kinematik upravit pohyb končetin či dokonce celého těla. Algoritmy na vyřešení tohoto problému byly vymyšleny už velmi dávno a původně pocházely z robotiky. Dříve bylo nutné šetřit výpočetním výkonem. Proto vývojáři tento problém animací neřešili a raději investovali výkon do zobrazování grafiky či čističových systémů. Dnes již jsme v jiné době, kde si naopak nemůžeme dovolit vypustit do prodeje 3D hru,v níž by byla postava zobrazena z těsné blízkosti, a zároveň aby v jejím algoritmu nebyla vyřešena interakce s okolím. Následující podkapitola se zaměřuje a přínosy, které nám spojení procedurální a před-vytvořené animace přináší.

4.1 Přínosy spojení

Hlavním přínosem spojení dvou rozdílných typů tvorby animace charakteru již byl zmíněn a je jím právě zvýšení realističnosti výsledné animace, kterou postava provádí. Pomocí jejich spojení a kinematik můžeme vytvořit animaci, která umožňuje dynamickou interakci s předměty uvnitř scény. Můžeme například animovat přesné sáhnutí na kliku dveří, animaci skupiny postav, držení obouruční zbraně a mnoha dalších zajímavých aktivit. Samotný přínos rozdělíme na dvě kategorie podle toho, zda upravujeme jednotlivé animace nebo přechod mezi nimi.

4.1.1 Úprava animace pomocí procedurální animace

Tato kategorie je zaměřena pouze na jednotlivé animace, které jsou přehrávány, ať už v cyklu, či jen jednou. Obsahuje veškeré animace, které postava provádí, tzn. chůzi, běh, úder, skok a mnoho dalších. Jako příklad si vezměme animaci běhu, jenž byla před-vytvořena v některém z 3D editačním softwaru, pokud bychom ji pouze v cyklu přehrávali, na rovném terénu získáme očekávanou sekvenci pohybů, bez jakýchkoliv defektů. Avšak stačí, abychom terén nahnuli a chtěli přehrát běh do konce. Sekvence pohybů by byla stále stejná, což by způsobilo odrážení chodidla z místa, které je aktuálně někde nad úrovní terénu, tudíž by došlo k odrazu ze vzduchoprázdna. Naopak noha, která došlapuje před těžiště, by prorazila terén. Navíc je třeba brát v úvahu i rychlost běhu, u níž by mělo v závislosti na naklonění terénu docházet k odpovídající změně, jinou rychlostí postava poběží po rovném terénu a jinou (ve většině případů pomalejší) pokud by se jednalo o běh do kopce.



Obrázek 4.1: Ukázka a) chůze po rovném a b) nakloněném terénu bez a s c) užitím procedurální animace k nastavení výšky chodidel

Všechny tyto případy lze vyřešit pomocí zapojení procedurální animace k již stávajícímu zakladu, vytvořenému pomocí animace klíčových snímků. Procedurální animace nám navíc umožňuje v reálném čase animace upravit výslednou sekvenci pohybů tak, aby přesně odpovídala či inter-reagovala s prostředím okolo charakteru. V závislosti na daném problému se využívá jak přímá tak inverzní kinematika. Pokud bychom chtěli aby naše postavička natáčela hlavu směrem k nejbližšímu nepříteli v okolí (zároveň je třeba mít odpovídající kost omezenou v jednotlivých rotačních osách), využijeme kinematiku přímou, jenž je ideální pro takto jednoduché případy.

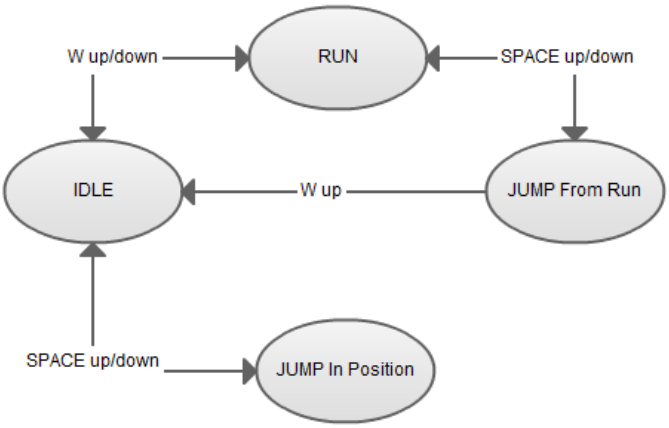
Druhý typ kinematiky se využívá, pokud máme řetězec kostí a bod v prostoru, označovaný jako koncový bod či efektor (end effector). Inverzní kinematika dopočítá natočení kostí v řetězci tak, aby poslední kost námi zvoleného řetězce končila právě v daném koncovém bodě (je-li to možné). Toho se využívá hlavně u končetin postavy, jenž se většinou skládají alespoň ze dvou kostí. Pokud aplikujeme inverzní kinematiku na kosti od pasu dolů až k chodidlu, můžeme v závislosti na výšce terénu koncový efektor řetězce zvýšit resp. snížit tak, abychom eliminovali nedostatky keyframe animace.

Tím však vzniká problém, a to je výška pivotu (jedná se o počátek kosterního sytému a reprezentuje těžiště 3D modelu). Jednoduchým řešením může být posun pivotu společně s výškou terénu a jeho aktuální výškou uloženou v animaci. Výsledek je pro většinu případů postačující, avšak jeho problémem jsou místa, kdy dochází k radikální změně výšky terénu mezi chodidly. Proto se výška povrchu v souřadnicích pivotu nebere z výškové mapy či jiného popisu, ale je dopočítána z výšek v souřadnicích nohou.

4.1.2 Přechod mezi animacemi

V předchozí podkapitole jsme vyřešili interakce jednotlivých animací s okolním prostředím, z čehož přejdeme k napojení animací za sebe. Před samotným plynulým přechodem je nutné vyřešit vlastní přechod z jedné animace na druhou, kde musíme brát ohled na to, zda daná animace může začít v průběhu jiné. Například pokud by postava v průběhu skoku dostala povel provést úkrok do strany, nelze animaci přerušit, postavu postavit do počáteční pózy a spustit animaci úkroku. Očekávaným výsledkem má být dokončení animace skoku a až poté provedení jiného pohybu (pokud mezitím nebyl zaslán jiný povel).

Simulaci přerušení můžeme modelovat například pomocí priorit, kdy nahrazení animace může být spuštěno pouze animací s prioritou vyšší. Avšak u složitějších systémů může dojít k takové závislosti mezi animacemi, kdy dopředu nastavené priority nejsou postačující a dochází k jejich dynamickému upravení podle vzniklé situace. Je zřejmé, že takovýto systém nemusí být přehledný, proto lepším řešením uvedeného problému je stavový automat, který se na začátku nastaví do počátečního stavu a v průběhu simulace dochází pouze k předem umožněným přechodům. Pro ilustraci je na obrázku 4.2 ukázán jednoduchý stavový automat, ve kterém má postava na výběr ze tří pohybů, jenž může provést pomocí stisku různých kláves.



Obrázek 4.2: Ukázkový stavový automat animací

Dále je potřebné umožnit souběžné přehrávání některých animací, například možnost provést seknutí mečem v průběhu běhu. Jelikož většina souběžných animací se skládá z jednoho pohybu v dolní a jednoho v horní části těla, pak pro docílení paralelního přehrání stačí rozdělit kostru na dvě části a pro každou část samostatně přehrát animaci.

Nyní se můžeme dostat k samotnému plynulému přechodu mezi animacemi. Nejjednodušším způsobem je interpolace mezi poslední pózou z předchozí animace a první pózou z animace nové v určitém časovém intervalu. Vhodnějším řešením by mohlo být zapojení procedurální animace i do této části animací. Avšak oproti méně složitému upravování výšky chodidla v závislosti na terénu, je pro realistický přechod nezbytné hlídat rozložení váhy těla. Pokud by předchozí animace končila na stojné pravé noze, musíme při interpolaci mezi animacemi zachovat tuto stojnou nohu a v průběhu začátku nové animace ji postupně zaměnit za nohu levou. Nevýhodou této techniky je nutnost uložení extrémního množství doplňujících informací o jednotlivých animacích, spolu s možnostmi přechodů mezi nimi.

Kapitola 5

Implementace

Po probrání teoretického základu potřebného k pochopení problematiky spojení, se nyní dostáváme k implementaci jednotlivých částí aplikace. Před tím, než se začneme věnovat pohybům a interakcím mezi objekty, je potřeba vyřešit, jak je můžeme do scény umístit. Abychom nebyli nuceni pracovat pouze se základními tělesy, jakými jsou například koule, kvádr, cylindr či jiné, dnes využíváme dnes 3D editační nástroje, díky kterým jsme schopni relativně jednouše vytvořit jakýkoliv objekt. Následnému exportu vytvořeného modelu do programu se bude věnovat první podkapitola. Poté postupně probereme implementaci jak statických modelů tak i modelů dynamických, ve kterých za jejich pohyb odpovídá jejich kostra. Detailněji se podíváme na strukturu třídy postavy, které ze základní třídy dynamického objektu vychází, ale navíc kontroluje interakci postavy s okolním prostředím.

5.1 Export modelů

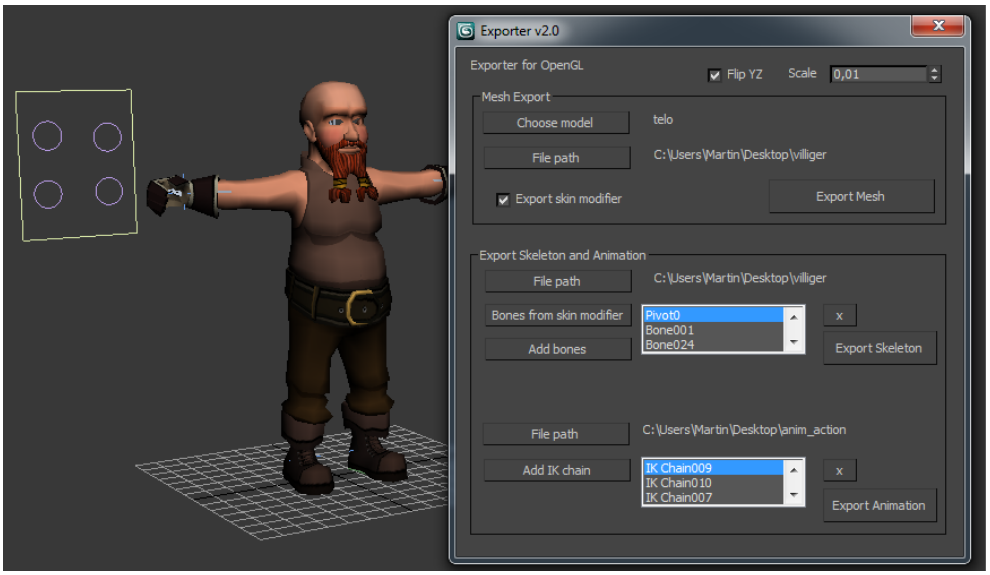
Po vytvoření modelu, jenž odpovídá našim představám, se dostáváme k potřebě jeho přenesení do našeho programu. Dnes již každý kvalitní editační software má možnost vyexportovat model do nejrůznějších formátů. Alternativně je možné si místo použití existujících exportérů vytvořit vlastní exportní skript, což je sice časově náročnější, ale lze tímto způsobem získat o objektu veškeré potřebné informace a uložit je pro nás nejefektivnějším způsobem. Metoda, při níž si studia napíší vlastní skript pro export je celkem častá, především pokud potřebují získat informace, jenž nejsou v obvyklých formátech obsaženy. Pokud je výstupní formát kvalitní může dát vzniknout i novému formátovému standartu. Příkladem je formát *.dae* původně vytvořený společností *Sony Computer Entertaiment*[5], který se postupně stal jedním z nejpoužívanějších formátů a jeho exportér je buď přímo zabudován či ve formě pluginu dostupný ve všech hlavních modelovacích nástrojích.

Z důvodů vzdělávacích a popsaných výše je pro aplikaci napsán vlastní exportér umožňující export statických i dynamických objektů včetně jejich animací. Jelikož pro vytvoření všech modelů byl použit software 3D Studio Max 2011, je exportér psán v skriptovacím jazyce MaxScript, který je tímto softwarem nabídnut. Dalším důvodem výběru právě MaxScriptu je bohatá dokumentace přímo v nápovědě programu, a rozsáhlá diskuzní fóra, kde lze nalézt řešení většiny běžných problémů s jeho používáním.

5.1.1 OpenGL Exportér v2.0

Důvodem názvu exportéru OpenGL Exportér je jeho primární určení importu modelů do programů, které pro vykreslování využívají grafickou knihovnu OpenGL. Jedná se o tex-

tový formát bez jakékoliv komprese a minimem režijního textu, okolo dat objektu. S jeho pomocí můžeme získat data, jak ze statických modelů tak i z modelů na nichž je aplikován modifikátor *Skin* společně s kostrou, jejichž spojení odpovídá za pohyb 3D modelu. Dále je s tímto skriptem umožněno exportovat animaci dynamických objektů.



Obrázek 5.1: Ukázka grafického vzhledu exportéru OpenGL Exporter v2.0

Nyní si detailněji přiblížíme co vše je skript schopen ze scény získat. Nejobecnější možností exportu je převrácení os y a z. Jelikož v 3ds Max je svislá osa vzhůru označena jako z a v OpenGL označena jako y. Druhým parametrem je měřítko, kterým se budou násobit veškeré vzdálenostní hodnoty při exportu. Vlastní export je zcela intuitivní, pouze vybereme model a nastavíme cestu, kam se má soubor uložit, pomocí připravených tlačítek. Pokud model obsahuje modifikátor *Skin* je možné jej před exportem zapnout či vypnout. Po nastavení všech potřebných parametrů vytvoříme finální soubor tlačítkem Export. Formát výstupního souboru je znázorněn v následující ukázce, a obsahuje pouze nejnutnější informace. První řádek obsahuje počet trojúhelníků vynásobený třemi, každé tři následující řádky představují právě jeden trojúhelník. Vrchol se skládá z informace o pozici, normále, texturovacích souřadnicích a v případě modifikátoru *Skin* počet kostí, na které má vazby, následovaný odpovídajícím počtem dvojic <*jméno kosti, váha*>.

Listing 5.1: Formát pro uložení vyexportovaného modelu s modifikátorem Skin

```
9366
-0.07, 1.57, -0.11, -0.87, 0.09, -0.47, 0.12, 0.14, 1, Bone1 , 1.0,
-0.06, 1.57, -0.13, -0.62, 0.09, -0.77, 0.13, 0.13, 1, Bone1 , 1.0,
-0.06, 1.59, -0.11, -0.80, -0.22, -0.54, 0.12, 0.15, 1, Bone1 , 1.0,
-0.06, 1.59, -0.11, -0.80, -0.22, -0.54, 0.12, 0.15, 1, Bone1 , 1.0,
-0.08, 1.60, 0.10, 0.92, -0.19, -0.31, 0.11, 0.15, 2, Bone1 , 0.5, Bone2, 0.5,
-0.07, 1.57, -0.11, -0.87, 0.00, -0.47, 0.12, 0.14, 2, Bone1 , 0.5, Bone2, 0.5,
-0.06, 1.59, -0.11, -0.80, -0.25, -0.54, 0.12, 0.15, 1, Bone1 , 1.0,
-0.06, 1.57, -0.13, -0.62, 0.09, -0.77, 0.13, 0.13, 1, Bone1 , 1.0,
-0.04, 1.58, -0.14, -0.20, 0.44, -0.87, 0.15, 0.14, 1, Bone1 , 1.0,
...
```

Další možností je vyextrahovat ze scény soubor popisující kostru vloženou uvnitř modelu. Zvolit kosti, které budou do kostry zapojeny, je možné buď přes vybrání modelu s modifikátorem *Skin* či přes tlačítko pro přidání jakékoliv kosti ve scéně. Případně, pokud sestava kostí obsahuje *solvery* pro inverzní kinematiku, je třeba je také přidat do spodního seznamu. Formát výstupního souboru na prvním řádku obsahuje počet pivotů (neboli kostí, které nemají předchůdce) a počet kostí v systému. Na dalších řádcích jsou nejdříve vypsány pivoty a až za nimi následují ostatní kosti. U každé kosti je zaznamenána její rotace od otcovské kosti v kvaternionech a její délka, u pivotů těmto informacím předchází jejich počáteční pozice. Základní informace o kostech soustavy jsou následovány řádkem s počet kloubů a jejich výpisem. Poslední blok souboru tvoří informace o inverzních kinematikách, nejdříve je zaznamenán počet kinematik, a následně jejich výpis obsahující: název, jméno počáteční kosti, jméno koncové kosti, jméno objektu ke kterému je kinematika přichycena a úhlu otočení (*swivel angle*).

Listing 5.2: Formát .skn obsahující údaje o struktuře kostry

```
3, 26
Pivot0 , 0.0, 0.107, 0.862, 0.506, -0.493, -0.506, 0.493, 0.054
LeftFootBone , -0.186, 0.112, 0.031, 0.478, -0.511, -0.570, 0.429, 0.158
RightFootBone , 0.18, 0.106, 0.0292, 0.607, -0.424, -0.437, 0.510, 0.160
Bone001 , -0.014, 0.706, -0.003, -0.707, 0.084
Bone024 , 0.516, -0.484, 0.448, -0.546, 0.457
...
23
Pivot0  Bone001
Bone001 Bone024
Bone024 Bone025
...
4
Chain01 , Bone021 , Bone022 , Bone022 , 180.0
Chain02 , Bone016 , Bone017 , Bone017 , 180.0
...
```

Posledním typem souboru, který je možno pomocí exportéru vytvořit je animace dynamického objektu. Skript ukládá pouze klíče, které má objekt nastaveny, jelikož očekává následnou interpolaci mezi snímky v cílovém programu. Formát souboru obsahuje na prvním řádku počet uložených kostí (objektů), spolu s délkou celé animace. Na ten navazuje vždy blok odpovídající právě jedné kosti, složený z jejího jména, a počtu zaznamenaných snímků. Každý snímek obsahuje: čas, aktální pozici (pokud se jedná o pivot) a rotaci.

Listing 5.3: Formát .ani uchovávající animace

```
3, 640
Bone001 , 2
0, 0.0, 0.0, 1.00143, 0.0, 0.0, 0.0, 1.0,
640, 0.0, 0.0, 1.00143, 0.0, 0.0, 0.707107, 0.707107,
Bone002 , 2
0, 0.0, 0.0, -0.697986, 0.716112,
640, 0.0, 0.0, -0.697986, 0.716112,
Bone003 , 4
0, 0.0, 0.0, -0.709097, 0.705111,
96, 0.242526, 0.241162, -0.666334, 0.662587,
192, 0.0, 0.0, -0.709097, 0.705111,
640, 0.0, 0.0, -0.709097, 0.705111,
```

5.2 Nahrání modelů do scény

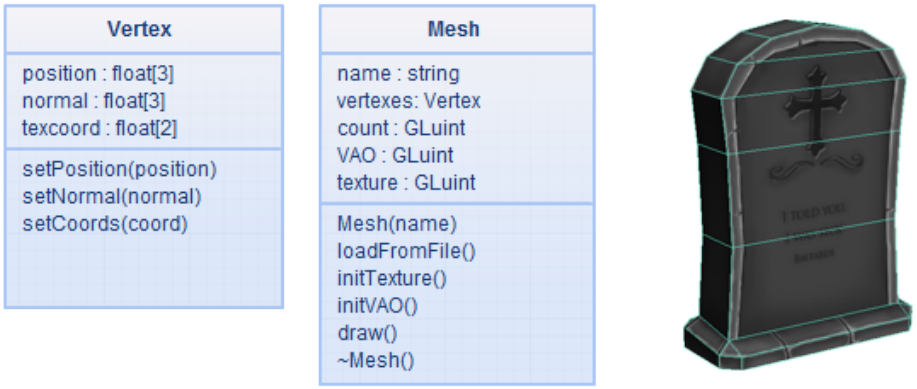
Pro minimalizování paměťových nároků obsahuje program třídu *ModelsList*. Ta spravuje veškeré modely se kterými program pracuje, a uchovává odkazy na ně. Při vytváření nového objektu se volá právě tato třída, která nejdříve prohledá svůj seznam již vytvořených objektů. Pokud vytvářený objekt nalezne vrací přímý ukazatel na něj. V opačném případě model vytvoří, a zařadí jej do svého seznamu. Seznam obsahuje dva typy modelů: statické a dynamické. Hlavním rozdílem těchto typů je to, že dynamické objekty mají navíc virtuální kostru, která ovládá jejich pohyb. Oběma se však budeme více věnovat ve zvláštních podkapitolách.

5.2.1 Třída pro statické objekty

Nejjednodušším typem objektů, se kterými aplikace pracuje jsou statické modely, které v průběhu času nemění svůj tvar a v programu jsou označeny jako třída *Mesh*. Přínos této třídy je v usnadnění práce s povrchem modelu neboli meshem označujícím síť bodů, mezi kterými jsou vytvořeny trojúhelníky. Tato třída tak obsahuje metody pro načtení modelu z vyexportovaného souboru, nahraní textury a data na grafickou kartu, a to pomocí moderních technologií OpenGL, jakými jsou například vertex buffer object a pro uložení jeho nastavení vertex array object. Následná transformace bodů pro vykreslení na obrazovku je řešena pomocí programovatelných shaderů (kvůli čemuž je potřeba OpenGL verze 3.1 či vyšší). Použité textury k modelům jsou ve formátu *.bmp* a pro jejich načtení bylo využito již existující volně dostupné třídy BMPLoader[3]. Ihned po jejím nahrání do paměti jsou navíc vygenerovány mipmapy, sloužící k snížení aliasingu při následném vykreslování.

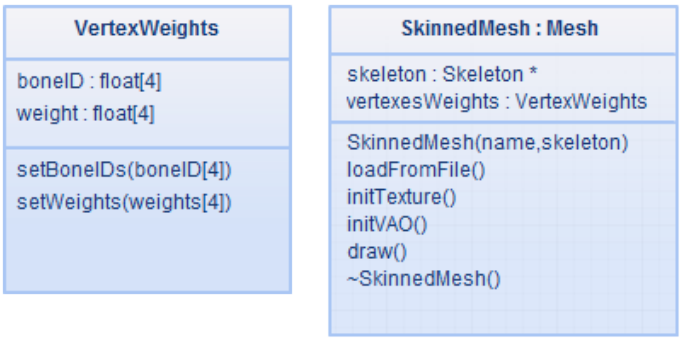
5.2.2 Třída pro dynamické modely

Dynamické modely umožňují oproti statickým nepohybovat pouze celým svým objektem, ale i jeho jednotlivými částmi. Těchto pohybů je docíleno pomocí virtuální kostry jenž je uložena uvnitř modelu, a každá její část odpovídá za předem dané vrcholy. Dynamické modely zapouzdřuje třída *SkinnedMesh*. Její struktura vychází z třídy statických modelů, avšak údaje o vrcholech jsou rozšířeny o informace určující jejich závislosti na jednotlivých kostech. Každý bod modelu uchovává čtyři dvojice tvaru<*boneID, weight*>, která udává index kosti v kostře a váhu jak moc je jí ovlivňován. Počet dvojic by mohl být i vyšší avšak většina vrcholů tento počet nevyužije ani z půlky. Případy kdy vrchol má vazby na pět



Obrázek 5.2: Třídy pro statické objekty včetně ukázkového modelu

a více kostí nastávají pouze ojediněle a často je lze jednoduše opravit odstraněním vazby s nejmenším vlivem. Navíc mají dynamické objekty ve své struktuře uložen ukazatel na odpovídající kostru, jejíž implementaci si přiblížíme v následující podkapitole.



Obrázek 5.3: Ukázka tříd pro dynamické objekty

5.2.3 Třída kostry

Třída odpovídající za veškeré operace okolo kostry je implementována pod jménem *Skeleton*. Její metody umožňují načíst kostru z externího souboru, včetně inverzních kinematik v ní aplikovaných, aktualizovat pozice a natočení jednotlivých kostí či vykreslit kompletní strukturu kostry. Pro potřeby interakce s jinými třídami je navíc implementováno vyhledávání určité kostry respektive řetězce počítající inverzní kinematiku podle zadaného jména. Ve výsledku můžeme říct, že hlavním přínosem této třídy je zaobalení všech kostí a kine-

matik, spolu s jejich metodami. Celkově docilujeme zpřehlednění a zvýšení efektivity práce nad množinou objektů. Nejdůležitější složkou jsou jednoznačně kosti, z kterých je kostra složena. Proto si přiblížíme právě jejich implementaci.

5.2.4 Třída kost

Každá kost je v prostoru jednoznačně určena jejím natočením, počáteční pozicí a její délkou. Proto i při jejím vytváření konstruktor očekává mimo jména kosti (pro následnou identifikaci) i rotaci a délku. Chybějící počáteční pozice je doplněna až následovně, v závislosti na předcházející rodičovské kosti na níž je napojena. Speciální typem jsou pivoty (středy hmoty – *center of mass*), které nemají rodiče a tudíž právě od nich celý výpočet počátečních pozic startuje. Kromě informací uchovávajících pozici v prostoru,musí navíc struktura pro kosti obsahovat odkazy na své případné následovníky a odkaz na případného rodiče. Rodičovská kost je vždy právě jedna (nejedná-li se o pivot, pak není žádná), ale odkazů na následovníky může obsahovat i více.

Důležitým faktem je v určování absolutní pozice kosti její relativní natočení od předka. Díky tomuto typu implementace je možné jednodušeji manipulovat s danými částmi struktury kostry. Například pokud by v závislosti na prostředí bylo potřeba otočit řetězcem kostí, bylo by potřeba aplikovat danou rotaci na všechny části řetězce. Avšak při implementaci tak jak je v aplikaci provedeno postačí natočit počáteční kost a při nadcházejícím přepočítání pozic bude vše natočeno správně.

Pro koncové dynamické modely s nimiž je po scéně pohybováno nejsou tyto vnitřní informace až natolik důležité. Hlavním spojením mezi kostmi a objektem jímž je hýbáno jsou transformační matice, které jsou vypočítány pomocí rekurzivního volaní metody *update()* od pivotů až po jejich nejvzdálenější potomky. Tato aktualizační metoda začíná výpočet u matice identity, na níž je aplikována translace do počáteční pozice, respektive koncové pozice předka. Posledním krokem je vynásobení zprava rotační maticí, dopočítání bodu v němž kost končí a rekurzivní zavolání funkce na potomky. Kromě transformační matice je ve třídě kosti navíc uložena inverzní matice k transformační matici z doby kdy postava byla v počáteční póze (neboli anglicky *bind pose*). Kromě seznamu kostí je v kostře obsažen i seznam řetězců řešících inverzní kinematiky. Kinematik je využíváno především pro interaktivní úpravu předdefinovaných animací.

5.2.5 Třída pro řešení inverzní kinematiky

Inverzní kinematika slouží pro dohledání natočení jednotlivých kostí řetězce, tak aby poslední člen končil v bezprostřední blízkosti zadaného bodu, pokud tedy takové řešení existuje. V závislosti na této definici je postaven i konstruktor ve vytvořené aplikaci. Jeho hlavními parametry jsou počáteční kost řetězce, koncová kost řetězce, cílový bod, jehož se kloubní napojení snaží dosáhnout a otočný úhel (anglicky *swivel angle*). Otočný úhel slouží jako doplňující informace o konečném natočení celé soustavy okolo osy mezi počátečním bodem řetězce a bodem pro který řešení hledáme.

Jelikož způsob, kterým je natočení soustavy kostí počítáno, je iterativní (jedná se o metodu popsanou již v teoretické části zprávy – *Cyclic Coordinate Descent*), mohl by jeho výpočet postupovat do nekonečna a výsledek by se limitně blížil k zadanému bodu. Abychom tomuto zamezili je nutné zvolit ukončující podmínku. Ve skutečnosti volíme podmínky dvě. Hledání ukončíme jednak, pokud se nalezené řešení dostatečně přiblíží k cílovému bodu a pak také v případě, že počet vyhledávacích cyklů přesáhne stanovenou mez.

Obrázek 5.4: Ukázka dynamického modelu a odpovídající kostry

Tímto jsme si vysvětlili implementaci všech základních elementů, na kterých je vytvořen aplikační základ a které se dále v implementaci používají. V závěru celé kapitoly je pro přehlednost přidán graf třídních a strukturních návazností, jenž vše demonstruje.

5.3 Implementace animací

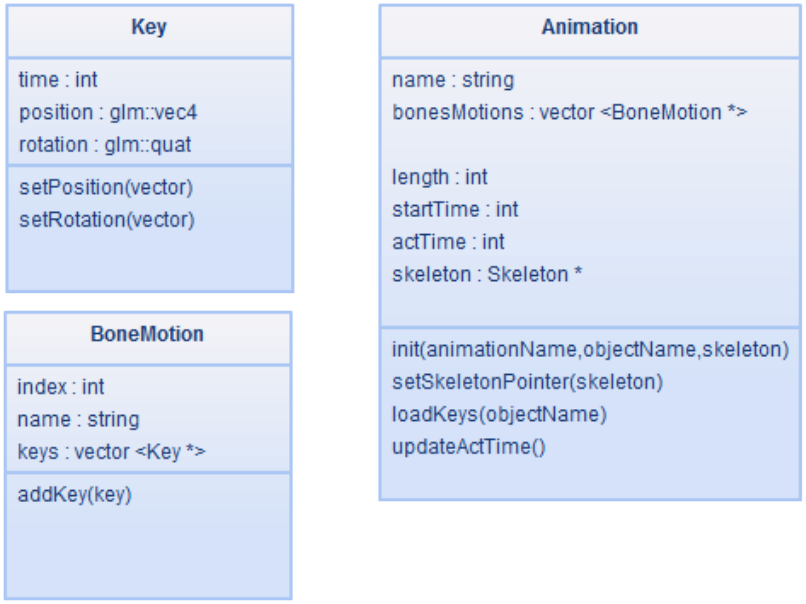
Nyní jsme již schopni nahrát do připravené scény modely a to jak statické tak i dynamické, ale zatím jsme si nic neřekli o tom, jak modely, především ty dynamické, vůbec rozpohybujeme. Tématu animací se věnuje právě tato podkapitola. V navržené aplikaci dochází ke spojení dvou odlišných typů animací. Za prvé zde máme k lasické key-frame sekvence, jenž obsahují několik klíčových snímků, mezi kterými, se provádí interpolace a za druhé pak animace procedurální, které mohou být popsány nejrůznějšími rovnicemi a dodatečnými podmínkami měnícími se v průběhu času. Každý z těchto typů má své silné stránky, ať už realističnost u klíčovaných animací, či možnost interakce s okolím u procedurálních, ale také mají své slabiny a právě spojením obou typů jsme schopni tyto slabiny minimalizovat či dokonce zcela eliminovat. Než však přistoupíme k samotnému sloučení, tak si oba typy podrobně rozebereme.

5.3.1 Keyframe animace

V předchozí kapitole jsme se dozvěděli jakým způsobem jsou animace, vytvořené v externím 3D editačním softwaru, vyexportovány do výstupního souboru, proto nyní postoupíme k popisu jejich vložení do námi prezentované aplikace. Nejelementárnějším prvkem objektu, v němž je animace uložena, je struktura *Key*, která představuje právě jeden snímek obsahující časový údaj, rotaci v kvaterionech a případně i pozici v daném čase. Celý seznam snímků popisujících kompletní pohyb kosti v průběhu animace je uložen v obecnější třídě *BoneMotion*, spolu se jménem a indexem kosti, které pohyb odpovídá.

Animace kostry jako celku je uložena v instanci třídy *Animation*, která zároveň tvoří nejobecnější třídu s níž veškeré dynamické objekty v aplikaci spolupracují. Hlavními atributy této třídy jsou: jméno animace a seznam pohybů jednotlivých kostí. Jelikož indexy pohybů jednotlivých kostí nemusí být uvnitř souboru uspořádány v zcela identickém pořadí, jako tomu je u odpovídající kostry, je nutné v době načítání hodnot kostru již znát. Samotné přidání jednotlivých snímků do patřičných struktur začíná u načtení jména kosti a vyhledání korespondujícího indexu ze seznamu kostí kostry. Poté v cyklu, jenž je spuštěn

přesně tolikrát kolik má daná kost animačních snímků, jsou ze souboru přečteny hodnoty aktuálního času a rotace v daném snímku. Výjimku tvoří kosti tvořící střed hmoty objektu neboli pivoty, u nichž je navíc uložena i absolutní pozice v době zaznamenání snímku. Kromě jednotlivých snímků je ve třídě animace navíc uloženo několik hodnot, uchovávající časové údaje. Jedná se o celkový, počáteční a aktuální čas dané animace. Tyto hodnoty jsou nepostradatelné pro zjištění v jakém časovém okamžiku se animace právě nachází a pro možné vyhledání snímků mezi kterými se má interpolovat. Kromě právě popsané metody pro načítání obsahuje třída *Animation* už jen několik metod pro změnu vnitřních hodnot, například metoda *updateActTime()*, která je zavolána před každým vyhledáním a která správně nastavuje aktuální čas animace.



Obrázek 5.5: Ukázka tříd pro interní uložení animace klíčových snímků

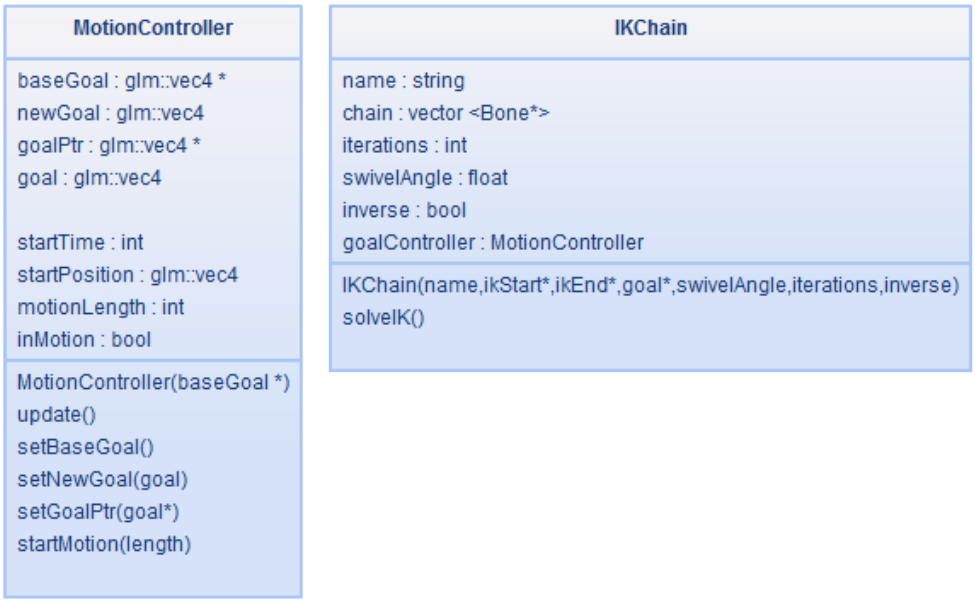
5.3.2 Procedurální animace

Rozdílným stylem animování oproti zaznamenávání klíčových snímků a jejich následné interpolaci je animace procedurální, jenž je v aplikaci využita spíše jako doplněk, který především eliminuje nedostatky ve vizuální realističnosti přehrávaných pohybů, které vznikají kvůli v důsledku předem neznámé interakci s prostředím. Základ pohybu je tedy založen na interpolaci mezi dvěmi snímky odpovídající aktuálnímu času, čímž získáme dynamický objekt, který přehrává aktuálně zvolenou animaci. Jelikož se však objekt při běhu aplikace bude pohybovat v odlišné scéně, než ve které byl definován, může docházet ke kolizím či jiným vizuálním nesrovnalostem. Proto je nutné tyto chyby odstranit, pomocí interaktivní animace.

Procedurální animace může být aplikována dvěmi různými technikami: první se nazývá dopředná (*forward kinematic*), jenž přímo nastavuje pozici kosti v prostoru. Samotná implementace i její zapojení do pohybu kostry je celkem jednoduché, pouze při každé aktualizaci pózy přepíšeme hodnoty, získané interpolací mezi snímky, na nově vypočítané z apliko-

vané kinematiky. Druhou technikou procedurálního pohybu je kinematika inverzní, jenž je výrazně složitější na implementaci i na výpočet. Typ inverzní kinematiky je v programu implementován uvnitř třídy *IKChain* a je vytvářen již při načítání kostry z exportovaného souboru. Konstruktor pro vytvoření nové instance třídy očekává sedm parametrů, jejich počet je ovlivněn především vyšší kompatibilitou s třídou kostry. Prvním argumentem konstruktoru je jméno, pro možnost jeho následného vyhledání. Následující dva argumenty udávají ukazatel na kost počáteční a koncovou mezi nimiž dojde k vytvoření řetězce s aplikovanou inverzní kinematikou, s čím souvisí i čtvrtý argument ukazatele na bod v prostoru, pro který bude kinematika počítána. Ostatní parametry nastavují hodnoty vnitřních proměnných jako například otočný úhel či počet iterací algoritmu *CCD*.

Aplikování inverzní kinematiky na kostru dynamického objektu je v zásadě stejné jako tomu bylo u dopředné kinematiky. Po nastavení vyinterpolované pózy, spustíme iterativní výpočet algoritmu, jehož výsledkem je odpovídající natočení jednotlivých elementů řetězce. Jelikož postup algoritmu *Cyclic Coordinate Descent* v 2D prostoru byl již popsán v teoretické části této práce, není potřeba detailně rozebírat jeho přepis do zdrojového kódu. Zmiňme pouze nutnost jeho úpravy při rozšíření do třetího rozměru, kdy v závislosti na úhlu sevřeném mezi počátečním a cílovým bodem okolo osy y, musíme o stejnou hodnotu pootočit celým řetězcem kolem jeho vnitřní osy (která vede mezi počátečním a koncovým bodem řetězce).



Obrázek 5.6: Ukázka tříd pro řešení inverzní kinematiky

Přidáním výpočtu kinematik při aktualizaci kostry získáváme nástroj, se kterým jsme schopni ovlivnit interakci mezi námi pohybovaným objektem a jeho blízkým okolím. V případě, že nedochází k žádné interakci je nutné aby aplikace kinematik nikterak neovlivnila původní snímkovou animaci. Proto se jako cílové body kinematik udávají pozice kloubů na konci celého řetězce, které mají za výsledek identické natočení kostí, jaké bylo před začátkem výpočtu. Naopak nastane-li kolize našeho dynamického objektu s některým jiným kolizním tělesem, je nutné cílový bod kinematiky upravit tak, aby se kolizi předešlo. Pro tyto případy

by si aplikace vystačila pouze s jedním bodem udávajícím cílový bod inverzní kinematiky, avšak představme si jeden z mnoha případů, kdy po určitý čas chceme, aby objekt opisoval pohyb získaný z vyexportované animace, následně na krátký interval převzal cílový bod z jiného objektu a na konec se opět vrátil k základnímu pohybu. Pro tyto případy je v aplikaci implementována třída *MotionController*, jenž v sobě uchovává jak ukazatel na cílový bod (v aplikaci označeno jako *basePostion*), který odpovídá klíčované animaci, tak i pozici nového cílového bodu *newPosition*. Výběr aktivní hodnoty cílové pozice v daném časovém okamžiku zajišťuje ukazatel *goalPtr*, směřující na právě jeden z nich. To aby přechod mezi cílovými pozicemi nebyl skokový, ale plynulý je zajištěno pomocí vnitřního mechanizmu uvnitř třídy. Vlastní přechod se spouští metodou *startMotion(length)*, jejíž parametr udává jak dlouho má přechod mezi pozicemi trvat a dále z metody *update()* která nám provádí aktualizaci mezi jednotlivými časovými úseky.

5.4 Implementace herních objektů

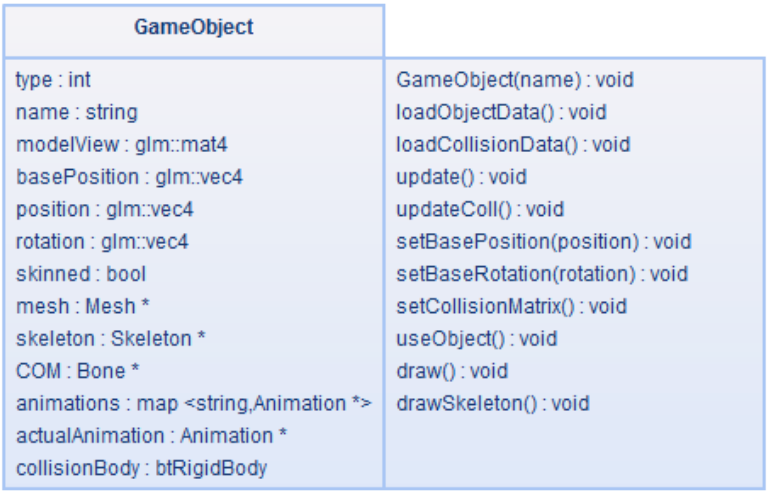
Po vysvětlení všech základních komponent, na kterých vytvořený herní svět stojí se již přesuneme ke kompaktnějším třídám, které reprezentují nejvyšší úroveň s kterou je ve vytvářené aplikaci pracováno. Jako první si popíšeme třídu *GameObject*.

5.4.1 Třída pro herní objekt

Herní objekt v sobě zaobaluje všechny výše popsané třídy a umožňuje reprezentaci jak modelů statických tak i dynamických, o kterých uchovává data polygonálního modelu a případně virtuální kostru spolu s před-připravenými animacemi. Konstruktor očekává pouze jeden povinný argument udávající jméno modelu, podle kterého je zjištěno (na základě existence souboru popisující strukturu kostry), zda se jedná o dynamický či statický model, který je následně do programu nahrán. Pokud se jedná o dynamický objekt je uvnitř konstruktoru navíc volána funkce pro načtení doplňujících dat k vytvářenému modelu. Data obsahují údaje nezbytné pro práci s animacemi, kromě jména pivotu, počtu a názvů animací jsou v souboru uloženy i data týkající se herních objektů, které jsou zároveň i aktivními objekty, jimž se více věnuje následující kapitola. Z dalších parametrů konstruktoru, které stojí za zmínění jmenujme dva čtyř-složkové vektory reprezentující absolutní pozici a natočení daného modelu v 3D prostoru scény. Obě tyto hodnoty mohou být zadány i kdykoliv později zavoláním metody *setBasePosition(position)* respektive *setBaseRotation(rotation)*. Z důvodu vyšší efektivity při následné manipulaci s herním objektem je jeho pozice i natočení přeneseno pomocí lineárních transformací do matice pojmenované *modelView*. Transformační matici využíváme především u metody *draw()*, která odpovídá za vykreslení modelu na obrazovku.

Posledním důležitým parametrem třídy herního objektu jsou animace, jenž se týkají pouze dynamických modelů. Jejich nahrání do struktury zajišťuje metoda *loadObjectData()*,v níž jsou jednotlivé animace nahrány do rozptýlené (hašovací) tabulky, s klíčem identickým názvu ukládané animace. Kromě tabulky animací si herní objekt navíc uchovává ukazatel na aktuálně přehrávanou animaci. Dynamické objekty jsou oproti statickým (pro které se spočítá transformační matice pouze při spuštění aplikace), před každým vykreslením znovu aktualizovány pomocí metody *update()*. Jedná se o nejdůležitější metodu v níž dochází i ke kontrole, zda již přehrávaná animace neskončila a neměl by se tedy ukazatel na aktuální animaci vynulovat. Jestliže animace není ukončena pokračuje aktualizace vyhledáním aktuálních dvou snímků, mezi kterými dojde k interpolaci rotací případně počátečních pozic

(jedná-li se o pivot), pro jednotlivé kosti. Výjimku tvoří střed hmoty herního objektu, který místo své počáteční pozice pohybuje celým modelem.



Obrázek 5.7: Ukázka třídy pro herní objekt

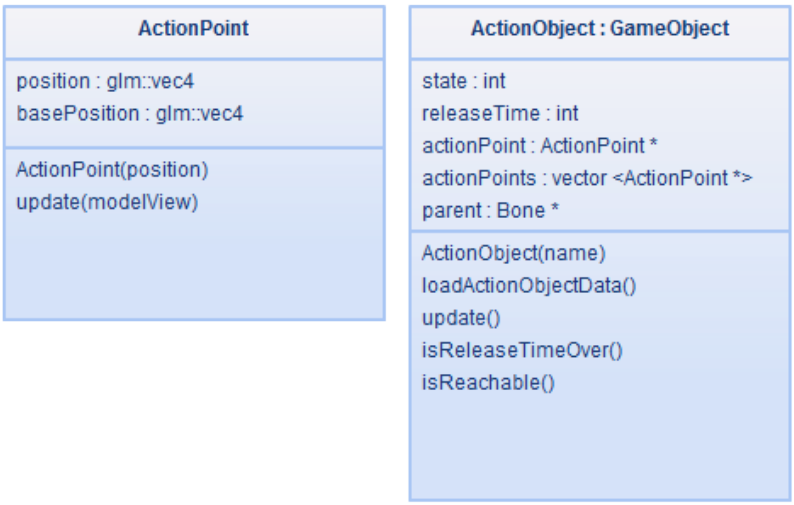
5.4.2 Třída pro akční objekt

Akční objekt vychází z třídy herního objektu, tudíž se také může jednat jak o statický tak i dynamický model a obohacuje jej o aktivní body, se kterými je umožněna následná interakce. Údaje o počáteční pozici a otcovskému objektu k němuž je aktivní bod přichycen jsou uloženy ve stejném souboru, jenž byl načten již s herním objektem. Spolu s těmito údaji dále obsahuje počet kontaktních bodů, výpis jejich pozic v prostoru a čas po kterém dojde ke konci kontaktu s akčním objektem. Jelikož s většinou těchto objektů je možné pohybovat musela být metoda aktualizace oproti hernímu objektu rozšířena o přepočítání aktuálních souřadnic všech akčních bodu. Výpočet nových souřadnic je důležitý pro další často využívanou metodu nazvanou *isReachable()*, která zkoumá, zda je některý z kontaktních bodů akčního objektu v dosažitelné vzdálenosti objektu, který metodu zavolal. Kromě vzdálenosti mezi body, metoda kontroluje i rozdíl úhlů.

Z akčních objektů vychází další více specializovaná třída *ActionObjectDoor*, pro kterou je již z názvu patrné že se věnuje implementací dveří. Jedinou metodou této třídy je přepsaná metoda herního objektu *useObject()*, která je rozšířena o stavový automat reprezentující otevírání a zavírání dveří s možnost dveře uzamknout na klíč. Jinou pomocnou třídou, která dědí z akčního objektu je *ActionObjectDestroyable*. Stejně jako třída specializovaná na dveře, tato třída přepisuje metodu *useObject()*, avšak tak že po jejím použití dojde k zničení kontaktního místa akčního objektu, čímž je simulováno ukradení předmětu z kapsy.

5.4.3 Implementace postavy

Třída reprezentující postavu se z části podobá třídě herního objektu, avšak je zároveň natolik rozdílná že při implementaci bylo rozhodnuto, že efektivnější bude vytvoření nové třídy namísto využití dědičnosti. Základní parametry uchovávající absolutní pozici objektu



Obrázek 5.8: Ukázka třídy pro akční objekty

v prostoru jsou stejné jako tomu bylo u herního objektu, mluvíme tedy o atributech uchovávajících aktuální pozici, natočení a s nimi neodmyslitelně spjatou transformační matici. Navíc je u třídy postavy přidána položka s hodnotou pozice uložené při předchozí aktualizaci, které se využívá pro výpočet herní fyziky, o níž bude řeč v poslední kapitole. Také proces uložení a manipulace s animacemi prodělal změnu oproti návrhu použitém u obecného objektu. Jednoduchý styl uchování ukazatele na právě přehrávanou animaci byl nahrazen frontou takovýchto ukazatelů, která umožňuje přidat najednou několik sekvencí snímků, jenž pak budou postupně přehrány bez nutnosti další korekce. Takto navržená organizace je velice efektivní pro zvýšení realističnosti pohybu postavy. Představme si příklad kdy s postavou není pohybováno a tudíž dochází k přehrávaní zcela stejné animace stále dokola, výsledkem je fádní animace. Avšak při použití návrhu s frontou je programátorovy nabídnuta možnost přidat naráz hned několik různých animací, které jako celek budou tvořit mnohem zajímavější výstup. Navíc přidávané sekvence, mohou být vybírány i náhodně, s jediným omezením týkajícím se jejich přesné návaznosti (ve které póze jedna animace končí, v té samé musí následující začínat).

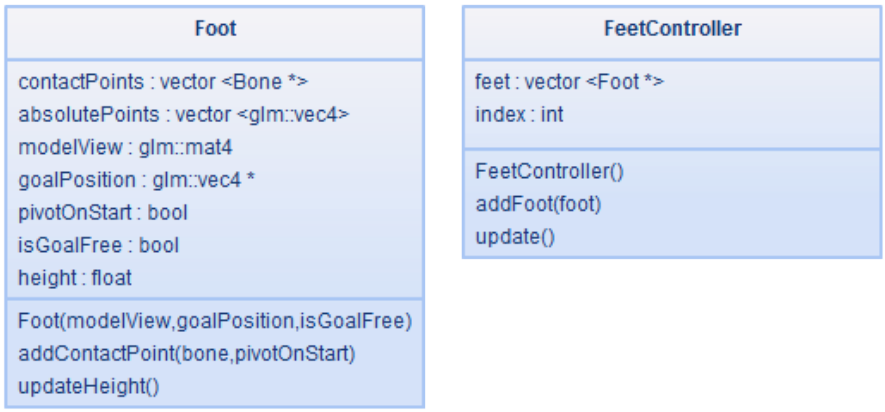
Výběr nově zařazených animací do fronty je řízen konečným automatem v metodě *setupAnimation()*, která bohužel obsahuje pouze dva stavy (jelikož z časových důvodů nebyla vytvořena animace skoku, a proto nebylo ani více stavů potřeba). Jedná se stav *IDLE*, kdy postava stojí na místě a stav *WALK*, při dochází k chůzi charakteru zvoleným směrem. Jelikož oba typy animací je možné přerušit v průběhu přehrávání, je přechod mezi danými stavy realizován, vymazáním celé fronty a následným přidáním nově zvolené animace. Přístup pouhé výměny jedné animace za druhou, produkuje skokovou změnu v póze postavy, která narušuje dojem realistického pohybu. Vzniklý problém je v aplikaci vyřešen pomocí přechodné animace, která je vložena na počátek fronty. Přechodná animace uchovává pro každou kost dva záznamy. První záznam odpovídá aktuální transformaci v které se kost nachází a druhý který je o poznání zajímavější, obsahuje transformaci, v níž se ocitne kost, po uplynutí intervalu kdy dochází k přechodu, v průběhu nové animace. Nyní když už jsme schopni přehrávat jednotlivé animace a s postavou můžeme libovolně pohybovat po celé scéně, přidáme její interakci s okolními objekty. Třída charakteru je připravena na

interakci dvěmi různými typy. Jako první si představíme interakci nohou s terénem, která je zapouzdřena do obecné třídy *FeetController*.

5.4.4 Implementace interakce s terénem

Třída *FeetController* obsahuje seznam ukazatelů na instance třídy *Foot* (importovaná postava není omezena pouze na dvě nohy, ale může jich mít i několik), pro které se při aktualizaci postavy spustí metoda *update()*. Počátečním krokem aktualizace je nalezení chodidla které je postavené nejníže v terénu, poté je celý model snížen o rozdíl mezi výškou nalezeného chodidla a výšky terénu ve středu postavy. Čímž jsme však mohli způsobit snížení ostatních končetin natolik, že nyní procházejí skrz terén. Proto na konci aktualizace spustíme cyklus kontrolující, zda opravdu ke kolizi nedošlo a případně výšku končetiny opravíme.

Zajímavostí u implementace třídy *Foot* je možnost složení chodidla z více kontaktních bodů, v kterých bude při aktualizaci hledána maximální výška. Pro lepší pochopení uvažme příklad lidského chodidla, u něhož je potřeba brát v úvahu jak výšku terénu u chodidla, tak i výsku u špičky prstů.



Obrázek 5.9: Ukázka tříd obstarávající interakci s terénem

5.4.5 Implementace interakce s akčními objekty

Další možností interakce s prostředím jsou akční objekty. Jejich implementace je podobná jako tomu bylo u ovladače nohou, kde menší elementy (v tomto případě instance tříd rukou) jsou uloženy v ovládající třídě *HandsController*. Kdy aktualizaci postavy předchází snaha nalézt akční objekt v dosažitelném okolí. Jestliže takový objekt byl nalezen, tak v průběhu hledání došlo i k nastavení aktivní ruky, kterou bude pohybováno. Pro pohyb ruky je použito již dříve popsané třídy *MotionController*, která umožňuje provést plynulou změnu mezi dvěmi cílovými pozicemi.

5.5 Kolizní model

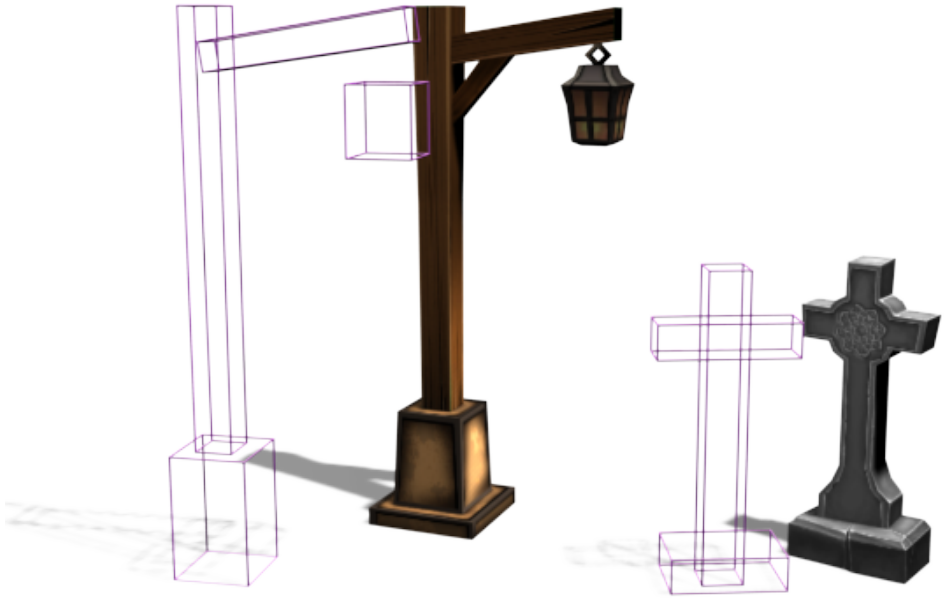
Abychom zamezili pohybu herních objektů skrz jiné objekty, bylo třeba do programu přidat fyzikální model detekující kolize. Jelikož kolize mezi objekty netvoří jádro námi prezentované aplikace, byl původní návrh řešení maximálně zjednodušen na model s pouze jedním kolizním tvarem, za který byla vybrána koule, jelikož představuje nejjednodušší prostorový objekt pro detekci kolizi. Cenou za jednoduchost a rychlost metody, byla především horší aproximace tvaru reprezentovaných modelů. A právě nepřesné popsání tvaru pomocí koulí stálo i za zamítnutím celého modelu. Jako jeho náhrada byla vybrána mnohem komplexnější fyzikální knihovna jménem *Bullet Physics*[2]. Oproti předcházející metodě knihovna Bullet nabízí pro reprezentaci herních objektů použití všech základních tvarů, od koule, krychle či kapsle až například po výškovou mapu. Kromě detekce kolizí, tato knihovna navíc obsahuje možnost diskrétní simulace kinematiky objektů. Celkově se jedná o velmi kvalitní knihovnu, která se používá i v mnoha komerčních aplikacích., ať už ve hrách (jako například *Grand Theft Auto V* či *Red Dead Redemption*), tak ve filmech (*Hancock, Sherlock Holmes*) a dokonce knihovnu nalezneme i v softwarech určených pro vytváření a editaci polygonálních modelů (*Blender, Cinema 4D*).

Zapojení fyzikálního modelu do vytvářené aplikace je implementováno uvnitř třídy *BulletWorld*, v jejímž konstruktoru dochází k inicializaci základních parametrů simulovaného světa (například nastavení gravitace), spolu s jejich následným provázáním. Třída dále obsahuje metody obstarávající vytvoření kolizních tvarů respektive dynamických objektů (rigid body). Veškeré vytvářené kolizní tvary jsou uchovány ve vnitřní hašovací tabulce, pro případ jejich opětovného použití u dalších instancí již vytvořených objektů. V aplikaci jsou rozlišeny čtyři druhy kolizních objektů:

- Objekty zcela statické - jedná se o nejjednodušší typ, který v průběhu běhu aplikace nemění svou pozici ani natočení. Typickým představiteli jsou různé sloupy, skály či stromy.
- Objekty částečně statické - pro fyzikální simulaci stále vystupují jako modely statické, avšak oproti zcela statickým, je možné měnit jejich pozici a natočení. Do této skupiny patří dveře a podobné objekty.
- Objekty plně dynamické - jsou modely, jenž jsou zcela řízeny fyzikální simulací. Většinou se jedná o menší modely, které dotváří atmosféru. Například menší krabice či kameny.
- Herní postava - posledním typem pro detekci kolizí je herní postava, s níž musí být možné pohybovat o přesně danou vzdálenost, ale zároveň musí dál interagovat s ostatními objekty. Proto je v aplikaci herní postava reprezentována tvarem kapsle, s níž je po mapě pohybováno pomocí nastavení vnitřní rychlosti v závislosti na uplynulém čase od minulé aktualizace.

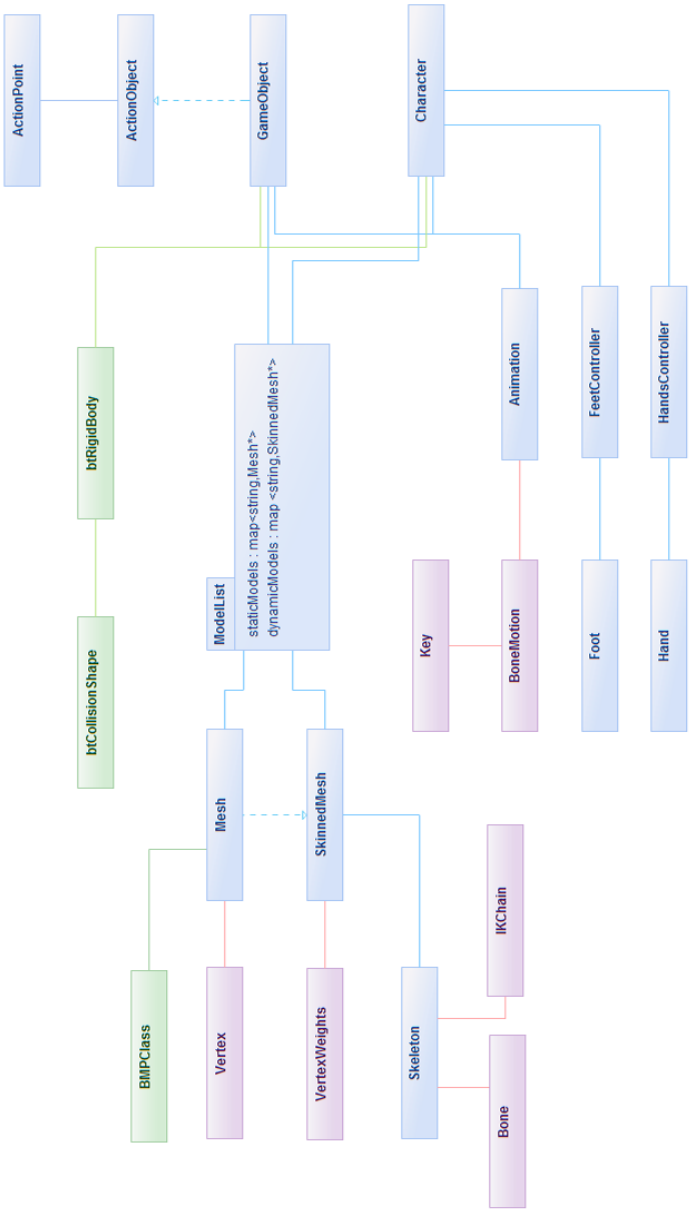
Speciálním případem statického objektu je kolizní tvar reprezentující terén, po kterém se postava pohybuje. Jeho základ je tvořen výškovou mapou, rozprostírající se po celém herním území, jenž je doplněna o další kolizní objekty které jeho tvar upřesňují. Diky popsanému spojení je možné do terénu zasadit schody, mosty a jim podobné objekty. Pro nalezení výšky terénu odpovídající daným souřadnicím je využito callback funkce *ClosestRayResultCallback* a metody *rayTest* z knihovny *Bullet*, jenž nám umožňují nalézt nejbližší kolizní bod na přímce mezi zvolenými body. Abychom zajistili, že vráceným bodem je místo kolize

paprsku s terénem musíme nastavit masku filtrující nechtěné kolize. Například pokud by byl paprsek vystřelen z vnitřku kolizního tvaru postavy, k první kolizi by došlo právě s tímto tvarem, nikoli s terénem.



Obrázek 5.10: Ukázka kolizních objektů pro vybrané modely

5.6 Diagram tříd



Obrázek 5.11: Grafické znázornění vazeb mezi jednotlivými třídami

Kapitola 6

Vizualizace

Pojem vizualizace v oblasti informatiky zahrnuje vše, co program uživateli nějakým způsobem zobrazuje. Oproti dřívějším dobám, dnes již nejsme limitováni na použití pouze jedné určité aplikace pro splnění zadaného úkonu (z důvodu její jedinečnosti na trhu), ale můžeme si vybrat ze široké palety více či méně vhodných konkurenčních aplikací. A právě vzhled může často rozhodnout o případném úspěchu aplikace mezi uživateli. Toto tvrzení platí dvojnásobně v oblasti počítačových her, kde se téměř zastavil vývoj nových žánrů, a pouze dochází k vylepšování kvality grafické stránky hry. Samozřejmě existují i výjimky, u kterých hratelnost převyšuje vše ostatní, avšak ty jsou časem nahrazeny novějšími hrami s podobnou (či dokonce identickou) hratelností, avšak s lepším grafickým provedením. Typickým příkladem mohou být hry z kolekce *Angry Birds*, které pouze pozvedly vizuální úroveň svých předchůdců, kterými byli simulace zaměřené na 2D ničení budov (často hratelné na internetu), a stal se z nich hit mezi hrami na mobilních telefonech. Kromě re-designu za jejich úspěchem stojí i masivní mediální kampaň.

To, že grafika je jedna z nejdůležitějších vlastností hry nutí programátory využívat všech moderních možností, jež realtimová 3D grafika poskytuje. Grafika počítačových her prošla za dobu své existence značným vývojem, který byl dán převážně dostupným hardwarem. Začínala tím, že si každý programátor musel vytvářet vlastní 3D renderovací engine, který byl standardní součástí programu a zpracovával se tak na procesoru. Později se objevily grafické karty, které umožňovaly přenést hlavní výpočetní zátěž na speciální paralelní hardaware a to pomocí specializovaného API. Problémem tohoto přístupu bylo, že možnosti grafických karet byly značně omezené a zvládaly jen určité předem dané typy výpočtů. To se samozřejmě časem ukázalo jako poměrně nedostačující, a tak dostaly grafické karty takzvanou programovatelnou pipeline, která umožňuje používat je do jisté míry jako obecný výpočetní paralelní koprocesor. K využití grafických karet se používají dvě hlavní API knihovny, a to jednak starší OpenGL, které je průmyslovým standardem a jde použít takřka kdekoliv od počítačů po mobilní telefony, a jednak DirectX 3D, což je proprietární technologie společnosti Mircosoft použitelná pouze na Windows, která je však součástí obecnější knihovny DirectX, která zastřešuje spoustu dalších úkolů používaných při programování počítačových her a tím ho značně usnadňuje.

Následující podkapitoly jsou věnovány multi-platformní volně šiřitelné grafické knihovně OpenGL, spolu s níž si přiblížíme i programovatelný řetězec shaderů, jenž zodpovídá za vykreslení jednotlivých objektů v naší aplikaci.

6.1 OpenGL

Grafickou knihovnu OpenGL(*Open Graphics Library*) není třeba dlouze představovat, jedná se o standard pro zobrazování 2D i 3D grafiky navržený tak, aby byl nezávislý na použitém operačním systému, grafických ovladačích a správcích oken. Proto také neobsahuje žádné funkce pro práci s okny (otevírání, zrušení, změnu velikosti), pro vytváření grafického uživatelského rozhraní ani pro zpracování událostí. Programátorské rozhraní knihovny OpenGL je vytvořeno tak, aby knihovna byla použitelná v téměř libovolném programovacím jazyce. Avšak primárně je k dispozici hlavičkový soubor pro jazyky C a C++. Z programátorského hlediska se knihovna chová jako stavový automat.

Od verze 2.1, OpenGL navíc podporuje programovatelnou pipeline, čímž došlo k zásadní změně při vykreslování objektů ve scéně. Zastaralé postupné vykreslování jednotlivých primitiv bylo nahrazeno dávkovým vykreslováním, kdy na grafickou kartu najednou pošleme všechny vrcholy spolu s jejich daty zapouzdřené v jediném bufferu, nazvaném *vertex buffer object* (dále jen VBO). Příkazy jako *glBegin, glEnd*, atd. kdy jsou informace o vrcholech přenášeny na GPU až při samotném vykreslování se nadále nepoužívají a jsou označeny jako nepodporované (tzn. že v novějších verzích knihovny už nemusejí být ani obsaženy). VBO představuje vyhrazené místo na grafické kartě, do kterého je nahrána především geometrie vykreslovaného objektu (pozice vrcholů, normály, texturovací souřadnice a jiné). Díky tomu odpadá nutnost stálého znovu nahrávání dat na grafickou kartu a dochází k výraznému zrychlení vykreslování. Zcela identicky jako s texturami, je i s buffery komunikováno pomocí identifikátorů. Proto prvním logickým krokem bude právě vygenerování jedinečného identifikátoru pomocí funkce *glGenBuffers*, odkazujícího se na daný VBO. Následně dochází k jeho aktivování příkazem *glBindBuffer* s odpovídajícími parametry, a posledním krokem je alokování místa odpovídajícího velikosti nahraných geometrických dat na GPU. Pokud jsou informace o vrcholech uloženy v souvislé struktuře jako je tomu u implementace statických objektů, pak lze alokaci i nahraní dat spojit v jediném příkaze *glBufferData*. Pokud však informace o vrcholech netvoří souvislou strukturu, stejně tak jako je tomu u dynamických objektů, jenž obsahují dodatečné informace o spojení s kostmi mimo strukturu Vertex, je nutné alokaci a nahrání rozdělit následovně:

Listing 6.1: Alokace a nahrani dat do VBO

```
glBufferData(GL_ARRAY_BUFFER,(sizeof(Vertex)+sizeof(VertexWeights))*count,
        0,GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(Vertex) * count, vertexes);
glBufferSubData(GL_ARRAY_BUFFER,sizeof(Vertex) * count,
        sizeof(VertexWeights) * count, vertexesWeights);
```

Abychom mohli data nahraná na GPU pomocí VBO vykreslit, musíme grafické kartě navíc předat strukturu uspořádání dat uvnitř bufferu. Proto OpenGL obsahuje funkci *glVertexAttribPointer*, která přesně definuje kolik hodnot a jakého typu se má načíst do zvolené proměnné spolu s jejich rozestupem v bufferu, oddělujícím informace o jednotlivých vrcholech. Abychom nemuseli při každém vykreslení stále dokola volat příkazy ukazatelů určujících uspořádání v bufferu, vznikl v OpenGL (verze 3.0) objekt který si pamatuje veškeré nastavení specifikující data vrcholů. Jedná se o *vertex array objec*t (dále pouze VAO). Jeho použití je velmi podobné postupu u VBO, kdy v prvním kroku vygenerujeme jedinečný identifikátor odkazující na nově vytvořené VAO a následně jej aktivujeme pomocí

funkce *glBindVertexArray*. Poté klasicky aktivujeme VBO a zavoláme příkazy nastavující jeho vnitřní ukazatele na informace o vrcholech. Jelikož se knihovna OpenGL chová jako stavový automat je nutné VAO opět deaktivovat, čímž zamezíme nechtěnému přednastavení interních ukazatelů do VBO. Pokud celý přenos dat na grafickou kartu a nastavení VAO proběhl správně, pak počet příkazů pro vykreslení geometrie je zredukován na pouhé zavolání příkazu aktivace VAO, příkazu vykreslení a opětovné deaktivace VAO.

Listing 6.2: Kód pro inicializaci VAO

```
void initVAO()
{
        glGenVertexArrays(1,&VAO);
        glBindVertexArray(VAO);

        GLuint VBO;
        glGenBuffers(1, &VBO);
        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex)*count,0,GL_STATIC_DRAW);
        glBufferSubData(GL_ARRAY_BUFFER,0,sizeof(Vertex)*count,vertexes);

        glEnableVertexAttribArray(shader->position);
        glEnableVertexAttribArray(shader->texCoord);
        glEnableVertexAttribArray(shader->normal);
        glVertexAttribPointer(shader->position, 3, GL_FLOAT, GL_FALSE,
                sizeof(Vertex), (void*)offsetof(Vertex, position));
        glVertexAttribPointer(shader->texCoord, 2, GL_FLOAT, GL_FALSE,
                sizeof(Vertex), (void*)offsetof(Vertex, texcoord));
        glVertexAttribPointer(shader->normal, 3, GL_FLOAT, GL_FALSE,
                sizeof(Vertex), (void*)offsetof(Vertex, normal));

        glBindVertexArray(0);
}
```

6.2 Implementace shaderů

Shadery jsou psány v jazyce GLSL, jenž je založený na syntaxi jazyka C, avšak navíc obsahuje mnoho vestavěných funkcí spojených s vykreslováním. Pro usnadnění práce se shadery je v aplikaci implementována třída *Shader*, jenž zaobaluje základní metody pro načtení programů z externích souborů spolu s kompilací a nastavením identifikátorů lokací jednotlivých proměnných. Po zavolání příkazu pro vykreslení dojde ke spuštění postupného průchodu dat uložených uvnitř VBO skrz posloupnost shaderů. Jako první dochází k paralelnímu zpracování vrcholů uvnitř vertex shaderu. Zde dochází k lineární transformaci hodnot pozic a normálových vektorů vrcholů. Kromě klasických transformací (převod do modelového či pohledového prostoru) je uvnitř vertex shaderu aplikován hardwarový skinning. Transformované vrcholy jsou dále poslány do fragment shaderu (zvaného též pixel shader), který pracuje na úrovni jednotlivých pixelů. Vstupní hodnoty pro každý bod na obrazovce jsou získány interpolací mezi předanými vrcholy. Uvnitř fragment shaderu dochází k výpočtu výsledné barvy pixelu, i proto je zde implementován osvětlovací model, obsahující jak stínování tak i osvětlení.

Navíc od verze OpenGL 3.2 může být vykreslovací řetězec obohacen o další stupeň, kterým je geometry shader. Uvnitř toho je programátorovi nově umožněno přidat či odebrat

vrcholy a tím ovlivnit zobrazovanou geometrii. Této techniky se využívá například pro generování jednoduché vegetace (především trávy) v reálném čase. Nejnovějším rozšířením řetězce shaderů je přidání teselace, která přivádí real-time změnu detailnosti modelu téměř k dokonalosti. Avšak ani jeden z těchto shaderů není v aplikaci využit.

6.2.1 Hardwarový skinning

Hlavním úkolem vertex shaderu v prezentovaném programu je transformace pozic a normál vstupních vrcholů do prostoru modelu, respektive do pohledového prostoru kamery. Avšak pro zrychlení vykreslení objektů obsahujících skeletální animaci, je i výpočet skinningu přenesen na grafickou kartu. Největším přínosem tohoto přenosu je snížení počtu výpočtů, spojených s transformacemi prováděných procesorem a s tím spojené zrychlení běhu aplikace. Každý vrchol kromě své pozice, normály a texturovacích souřadnic navíc obsahuje dva atributy čtyř hodnotových polí. Dvojice složené z hodnot z prvního a druhého pole o stejných indexech, reprezentují informaci o indexu a odpovídající váze k dané kosti, s níž je vrchol svázán. Posledním potřebným vstupem pro hardwarové urychlení je uniformní atribut obsahující pole transformačních matic, vynásobených inverzními maticemi v počáteční póze jednotlivých kostí.

Výpočet výstupní hodnoty pozice vrcholu je proveden postupným vynásobením maticemi kostí s nimiž je svázán, čímž získáme pozice, ve kterých by se vrchol ocitl v případě stoprocentní váhy k dané kosti. Nyní však využijeme skutečnosti, že součet vah je roven jedné a proto postačí výsledný bod vložit do počátku souřadného systému a v cyklu k němu přičítat souřadnice jednotlivých pozic vynásobené odpovídajícím koeficientem váhy. Oproti získání výstupních pozic vrcholů, normálové vektory nejsou ovlivněny translací v prostoru, proto je k jejich transformaci využito transformační matice omezené na rozměr 3x3, jenž uchovává pouze obecnou rotaci.

Listing 6.3: Algoritmus skinningu na GPU

```
if (skinned == true) {
        v = vec4(0.0,0.0,0.0,1.0);
        N = vec3(0.0,0.0,0.0);

        for (int i =0; i < 4; i++)
        {
                vec4 helper = transform[int(indexes[i])] * vec4(position,1);
                vec3 helperN = normalize(mat3(transform[int(indexes[i])])
                        * normal);
                v = vec4(v.x + (helper.x * weights[i]), v.y + (helper.y *
                        weights[i]),   v.z + (helper.z*weights[i]),1);
                N = vec3(N.x + (helperN.x*weights[i]), N.y + (helperN.y *
                        weights[i]), N.z + (helperN.z*weights[i]));
        }
        v = modelView * v;
        N = normalize(mat3(modelView) * N);
}else{
        v = modelView * vec4(position,1);
        N = normalize(mat3(modelView) * normal);
}
gl_Position = projection * v;
```

Jelikož vrcholy v implementované aplikaci kromě pozice a normálových vektorů nesou i texturovací souřadnice, je nutné tyto hodnoty předat dále, tak aby při rasterizaci mohlo dojít k odpovídající interpolaci. Jakmile pasterizace skončí, přejde program ke fragment shaderu.

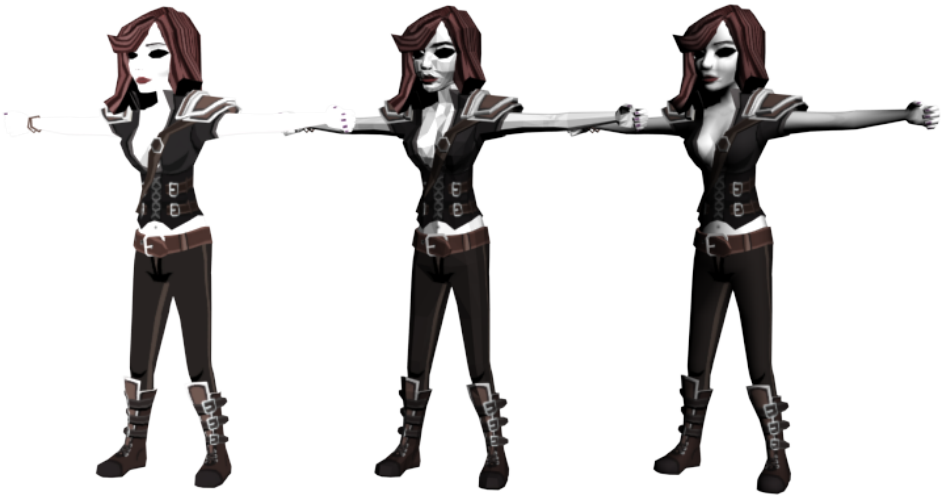
6.2.2 Stínování

Pod pojmem stínování rozumíme vykreslování barevných objektů různými odstíny barev tak, aby se zdály různě zakřivené nebo zaoblené, daly se rozlišit jednotlivé plochy a navodil se dojem hloubky. Stínování neslouží k výpočtům ani zobrazování stínů pouze zajišťuje, že objekt bude mít v různých částech různou světlost dle natočení daného místa ke zdroji světla. Stínování tvoří nedílnou součást vykreslovacího řetězce (především u vizualizace 3D objetků), kterou kdybychom vynechali, byl by model zobrazen pouze jako otexturovaná silueta. Míra ztmavení pixelu modelu závisí na normálovém vektoru zpracovávaného bodu, jenž odpovídá kolmému vektoru k tečně dané roviny a pozici oka pozorovatele. V prostorové grafice jsou využívány pouze dva standardní typy stínování. Prvním a zároveň i jednodušším způsobem je stínování ploché (*flat shading*), jenž zobrazí všechny body uvnitř polygonu ztmavené o stejnou hodnotu. Toho je docíleno vypočítáním pouze jednoho normálového vektoru pro celý vykreslovaný polygon. Výhodou této metody je její rychlost, avšak zobrazený výstup není pro většinu dnešních aplikací přípustný.

Proto byla zavedena metoda hladkého stínování (smooth shading), jenž zohledňuje zakřivení povrchu objektu a oproti předchozímu typu si nevystačí pouze s jedním normálovým vektorem ale do výpočtu zapojuje normály všech vrcholů primitiva. Proto zobrazený polygon mění úroveň ztmavení hladce napříč svým povrchem. Jedním z typů hladkého stínování je Gourandův model, kdy nejprve dojde k vyhodnocení úrovně ztmavení v jednotlivých vrcholech primitiva a při následné rasterizaci je míra ztmavení interpolována skrz vykreslovaný tvar. Výstup Gourandova stínování je pro většinu aplikací dostatečně realistický, avšak díky možnosti programovatelné pipeline, jsme nyní schopni přivést stínování k dokonalosti pomocí Phongova modelu, aniž bychom příliš zvýšili výpočetní náročnost. Phongovo stínování nahrazuje interpolaci úrovně ztmavení za interpolaci normál uvnitř polygonu, tudíž je osvětlovací model počítán pro každý pixel.

6.2.3 Osvětlení

Aplikace se snaží simulovat běžné sluneční osvětlení volného prostranství, k čemuž využívá tři nezávislé složky dle Phongova modelu. Pro získání výsledné barvy je třeba všechny tři složky sečíst dohromady. První složkou je rozptýlené světlo (*ambient light*), u nějž nelze určit směr šíření ani směr dopadu, toto osvětlení je rovnoměrně odráženo do všech směrů. Určuje nejnižší možnou úroveň osvětlení všech objektů ve scéně. Při použití pouze rozptýleného světla by každý neotexturovaný objekt měl pouze jednu barvu a byla by vidět jen jeho barevná silueta. Rozptýlené světlo slouží jako základ celého osvětlení a bývá nastaveno na nízkou intenzitu, pouze pro osvětlení míst, na které by se bodová světla nedostala. Druhá složka je difúzní světlo, jedná se o světlo, které dopadá z jednoho směru a odráží se rovnoměrně do všech směrů. Vytváří dojem plasticity hmoty. Intenzita osvětlení závisí na úhlu mezi zdrojem světla a normálou místa povrchu (ne na směru pozorovatele). Poslední složkou je odlesk (*specular light*), který reprezentuje světlo dopadající z jednoho směru a následně je odraženo do jiného směru. Odlesk vytváří dojem skleněného či kovového materiálu. V případě této složky je intenzita závislá na úhlu mezi pozicí pozorovatele a směru odraženého paprsku ze světelného zdroje.



Obrázek 6.1: Srovnání otexturovaného modelu a) bez stínování, b)s plochým stínováním, c) a s Phongovovým stínování

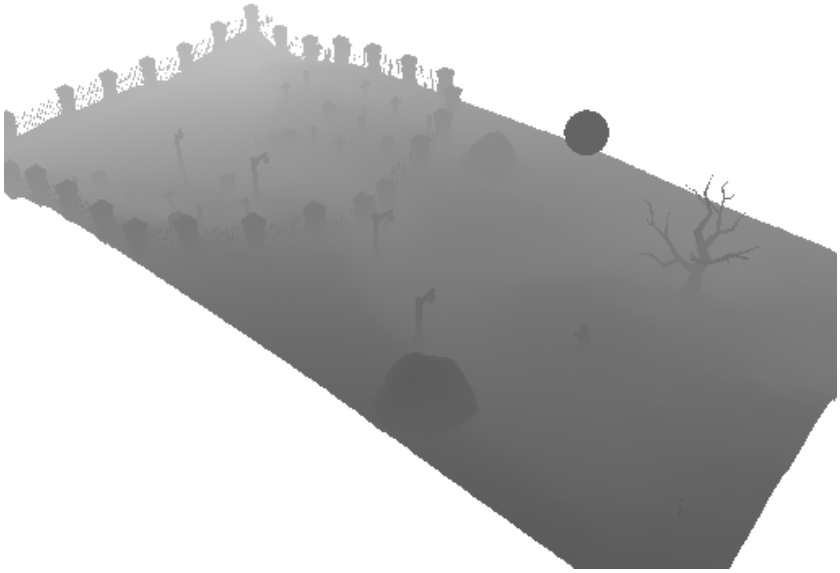
6.2.4 Stíny

Klíčovým elementem vizuální stránky moderních aplikací jsou stíny, které nejen přidávají realističnost, ale také umožňují uživateli jednoznačně určit polohu objektu v prostoru. S mohutným nárůstem výpočetního výkonu procesoru a vylepšením grafických karet je dnes možné implementovat algoritmy zobrazující dynamické stíny i přes jejich vysokou výpočetní náročnost v reálném čase. Technik, používaných pro výpočet dynamických stínů, existuje v současné době hned několik, avšak každá z nich má i své nedostatky, proto nelze jednoznačně vybrat pouze jednu, ale jsme nuceni se rozhodnout v rámci kontextu programu. Jak již bylo zmíněno v našem případě, potřebujeme simulovat pouze jedno směrové světlo, a proto byla vybrána metoda hloubkových stínových map (*depth map shadows*).

Metoda hloubkové mapy byla poprvé představena Lancem Williamsem v roce 1978[15], v jeho článku o vrhání zakřivených stínů na zakřivené povrchy. Princip metody je založen na pohledu z pozice světla (nikoli pozorovatele), kdy všechny objekty, jenž jsou z tohoto místa viditelné, jsou osvětleny a naopak místa, která vidět nejsou se ocitají ve stínu. Samotný pohled do scény je reprezentován hloubkovou mapou zobrazenou z pozice světla, která je uložena do předem připravené textury. O potřebné nastavení pro vykreslení scény do textury se v implementaci stará třída *FBO*. Při jejíž inicializaci je vygenerován obrazový rámec (*framebuffer*), jenž tvoří kontejner pro texturu obsahující rendrovaný obraz spolu s hloubkovým bufferem. Vzápětí jsou obě zmíněné komponenty vytvořeny a propojeny s obrazovým rámcem. Pro případ, že by v průběhu popsaného procesu došlo k chybě, je na konci inicializace přidána kontrola, zda je obrazový rámec v pořádku. Jestliže ano, pak již nic nebrání svázání s framebufferem a následným vykreslením scény do připravené textury.

V prvním průchodu získáváme hloubkovou mapu scény tak, že scénu zobrazíme z pozice světla a v závislosti na vzdálenosti jednotlivých objektů od této pozice je určena jejich barva. Čím jsou objekty vzdálenější, tím jejich světlost roste. Popsaného efektu zobrazení je docíleno průchodem implementovanými shadery s názvem *depthShader*. Jelikož nás u zobrazovaných fragmentů zajímá pouze jejich vzdálenost od pozice světla, postačí vertex shaderu poslat pouze souřadnice a transformační matice pro daný vrchol. Následně ve frag-

ment shaderu dojde k nastavení výsledné barvy, jenž přesně odpovídá vzdálenosti fragmentu od pozice světla (viz obrázek 6.2).



Obrázek 6.2: Ukázka hloubkové mapy zobrazované scény

V případě že uložíme získanou hodnotu rozdílu vzdálenosti do jediného kanálu textury, respektive jí zkopírujeme do kanálů všech, může být efektivita takovéhoto řešení nízká. Důvodem je omezená přesnost jednoho kanálu textury (v našem případe na 8 bajtů), proto je skutečná vzdálenost oříznuta. O mnoho lepším řešením je rozložení hodnoty vzdálenosti přes všechny čtyři složky, čímž ekvivalentně vzroste i přesnost u porovnání, zda výsledný fragment leží ve stínu či nikoliv. Rozložení do jednotlivých kanálů můžeme implementovat ve fragment shaderu pomocí speciální funkce například následovně:

Listing 6.4: Funkce pro zakódování hloubky do RGBA

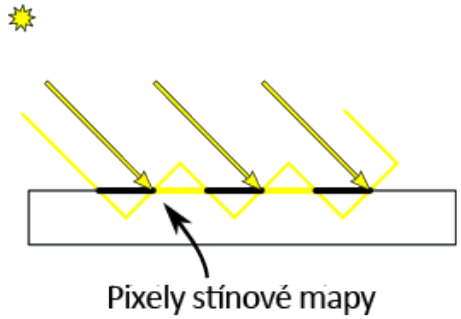
```
vec4 EncodeFloatRGBA( float v ) {
  vec4 enc = vec4(1.0, 255.0, 65025.0, 160581375.0) * v;
  enc = fract(enc);
  enc -= enc.yzww * vec4(1.0/255.0,1.0/255.0,1.0/255.0,0.0);
  return enc;
}
```

Inverzní operaci k převodu na kanály RGBA je jejich zpětné dekódování, které vypadá takto:

Listing 6.5: Funkce pro rozkódování RGBA do hloubku

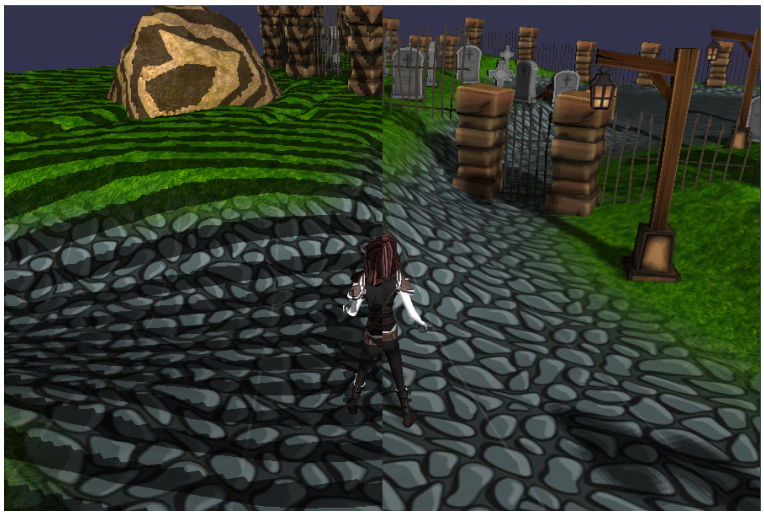
```
float DecodeFloatRGBA( vec4 rgba ) {
  return dot( rgba, vec4(1.0, 1/255.0, 1/65025.0, 1/160581375.0));
}
```

Poté, co vygenerujeme hloubkovou mapu a následně ji nahrajeme do paměti GPU, můžeme pokročit k finálnímu zobrazení scény, s již aplikovanými dynamickými stíny. Abychom zjistili, zda daný fragment leží ve stínu či nikoliv musíme nalézt odpovídající pixel ze stínové mapy. Pro jeho nalezení využijeme projekční matici světla, kterou vynásobíme absolutní pozici vrcholu uvnitř vertex shaderu. Hodnotu nalezeného pixelu dekódujeme z RGBA kanálů tak, abychom získali její skutečnou hodnotu, kterou porovnáme s nově vypočtenou vzdáleností mezi pozicí světla a fragmentu. Jestliže je nově získaná vzdálenost větší než hodnota ze stínové mapy, pak ve scéně existuje objekt, který právě zpracovávaný fragment zakrývá, a tudíž se ocitá ve stínu. Pokud bychom výpočet dále neopravovali, projevila by se chyba akné typická u metody stínových map (viz obrázek X.X). Příčinou akné je použití textury, která definuje povrch diskrétně a dochází k částečnému zastínění objektu sebou samým. Vznik problému nejlépe ilustruje následující obrázek.



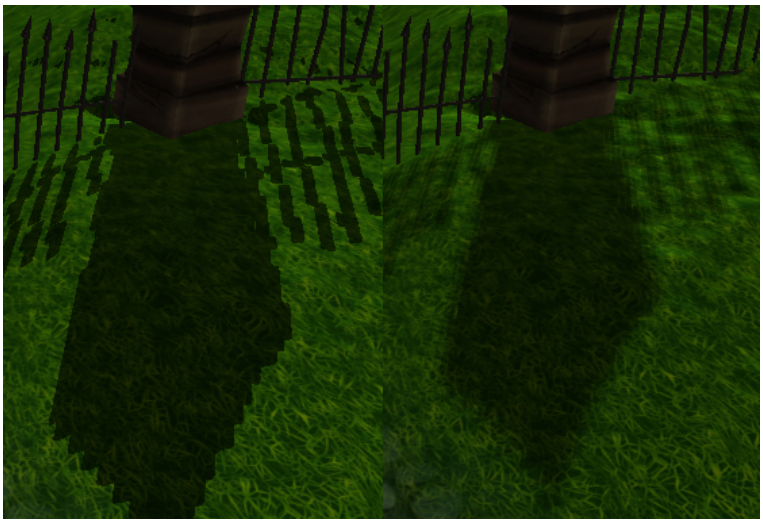
Obrázek 6.3: Grafické znázornění příčiny vzniku akné u stínových map

K odstranění této vady se běžně používá přidání tolerance chyby u porovnání vzdáleností, s kterou naopak vzniká chyba stínování u objektů, jenž jsou v těsné blízkosti. Proto je nutné zvolit velikost přidané tolerance dostatečně velkou, aby nedocházelo k akné, ale zároveň dostatečně malou aby se minimalizovala chyba stínování mezi blízkými objekty.



Obrázek 6.4: Grafické znázornění příčiny vzniku akné u stínových map

Poslední úpravou implementovaných stínů je jejich přeměna na měkké stíny (soft shadows), která sice zvyšuje výpočetní náročnost programu, avšak na druhé straně odstraňuje zubatost vzniklou nedostatečným vzorkováním objemu světla. Měkkých stínů je v aplikaci dosaženo pomocí filtrační techniky zvané percentage-closer filtering (PCF), jejíž princip spočívá v porovnání zpracovávaného fragmentu nejen s odpovídajícím pixelem ze stínové mapy, ale i s jeho okolím. Neboli pokud je daný pixel ve stínu, ale některý z bodů v jeho okolí nikoliv, pak jsme pravděpodobně narazili na přechod mezi světlem a stínem, tudíž intenzitu stínu snížíme. Technika PCF se počítá jako aritmetické průměr hodnot výsledků testů na porovnání. Velikosti okolí, které testujeme, přímo ovlivňuje jemnost přechodu. Čím větší okolí, tím je přechod jemnější, avšak na druhé straně roste výpočetní náročnost.



Obrázek 6.5: Grafické znázornění příčiny vzniku akné u stínových map

6.3 Grafický vzhled aplikace

Následujících několik obrázků je zaměřeno na grafickou prezentaci interakce postavy s okolním prostředím.

Obrázek 6.6: Ukázka interakce s akčním objektem (pohyblivou pákou)

Obrázek 6.7: Ukázka rozšíření kolizního modelu terénu o statické(chody) a dynamické objekty(malé kameny)

Kapitola 7

Závěr

V rámci tohoto projektu se nám podařilo spojit dva zcela odlišné přístupy k vytváření animace, jmenovitě animaci pomocí klíčových snímků s animací procedurální. Každý ze zmíněných přístupů má své výhody i omezení, a právě jejich spojením jsme se snažili tyto nedostatky eliminovat, abychom dosáhli zvýšení realističnosti výsledné animace. Největším přínosem představeného spojení byla možnost dynamické změny před-vytvořené animace v závislosti na okolním prostředí. Tímto jsme byli schopni docílit přesného došlapování při chůzi postavy po nerovném terénu, aniž by chodidla propadávala skrz, resp. se odrážela ze vzduchu. Možností využití však existuje mnohem více, ať už při interakci charakteru s předměty v jeho okolí (např. přesné otevírání dveří klikou, držení předmětu), až po odstranění skokového přechodu při změně mezi přehrávanými animacemi a jeho následné nahrazení přechodem plynulým. Práce nás tak zavádí do matematických principů, jenž stojí za veškerým pohybem který je v aplikaci implementován.

Jádrem práce a zároveň i jejím hlavním přínosem je navržený model fúze keyframe a procedurální animace, jenž podrobně vysvětluje způsob upravení animace založené na klíčových snímcích v závislosti na okolním prostředí v blízkosti charakteru. K velmi zdařilému upravení pohybu pomocí procedurální animace, velkou měrou přispěli právě kinematiky (dopředná i inverzní), které se podařilo napojit na virtuální kostru postavy. Dalším důležitým bodem práce bylo zapojení fyzikální knihovny Bullet, díky které mohl být tvar terénu obohacen jak o statické (například schody) tak i plně dynamické modely (menší kameny). V neposlední řadě je také vysvětlena technika stínových map, která přispívá k vysoké grafické úrovni vizualizované scény.

Aplikace jako celek tvoří rozmanitý základ určený především pro herní nadstavbu. Proto nejpravděpodobnějším směrem vývoje bude právě počítačová 3D hra, v níž hlavní roli bude představovat uživatelem ovládaná postava. Možnosti interakce charakteru s okolím jsou obrovské, od již zmíněných, které aplikace v aktuální fázi umožňuje, a jim podobných (například lezení po žebříku), až po jejich komplexní spojení s fyzikálním modelem a fyziologií těla.

Literatura

[1] Bill Baxter: Fast Numerical Methods for Inverse Kinematics[online]. `http://billbaxter.com/courses/290/html/`, 2000-02-21 [cit. 2013-05-06].

[2] Bullet Physics: Official website[online]. `http://bulletphysics.org/wordpress/`, 2013 [cit. 2013-05-06].

[3] Chris Backhouse: Chris's page: BMP Loader[online]. `http://peachieprojects.wikispaces.com/file/view/Bitmap.cpp`, 2006-10-07 [cit. 2013-05-17].

[4] Chris Welman: Inverse Kinematics and Geometric Constraints for Articulated Figure Manipulation[online]. 1993 [cit. 2013-05-03].

[5] Collada: Digital Asset and FX Exchange Schema - Official website[online]. `https://collada.org/`, 2013 [cit. 2013-05-20].

[6] Dante Treglia: *Game programming gems 3*. Charles River Media, Inc., 2002, iSBN 1-58450-233-9.

[7] Duncan Marsch: *Applied Geometry for Computer Graphics and CAD*. Springer Undergraduate Mathematics Series, 1999, iSBN 1-85233-080-5.

[8] Dunn Fletcher, Ian Parberry: *3D Math Prime for Graphics and Game Development*. Wordware Game Math Library, Wordware Pub., 2008, iSBN 978-1556229114.

[9] Eric Lengyel: *Mathematics for 3D Game Programming and Computer Graphics, Second Edition*. Charles River Media, Inc., 1999, iSBN 0-8311-3111-X.

[10] Hubert Nguyen: *GPU Gems 3*. Addison-Wesley Professional, 2007, iSBN 0-321-515-269.

[11] Jan Buriánek: *Animazoo*. Časopis PiXEL, č. 191, str. 31-33, 2012, iSSN 1211-5401.

[12] Jan Kříž: *3ds max 6: Animace a vizuální efekty*. Brno : Computer Press, 2004, iSBN 80-251-0328-5.

[13] Jan Kříž: *3ds max : Hotová řešení*. Brno : Computer Press, 2005, iSBN 80-251-0885-6.

[14] Mark A. DeLoura: *Game programming gems*. Charles River Media, Inc., 2000, iSBN 1-58450-049-2.

[15] Mark A. DeLoura: *Game programming gems 2*. Charles River Media, Inc., 2001, iSBN 1-58450-054-9.

[16] Michael E. Mortenson: *Mathematics for computer graphics application, Second Edition.* Charles River Media, Inc., 2004, iSBN 1-58450-277-0.

[17] Randima Fernando: *GPU Gems: Programing Techniques, Tips and Tricks for Real-Time Graphics.* Pearson Higher Education, 2004, iSBN 80-251-0328-5.

[18] Tomas Akenine-Moller, Eric Haines, Naty Hoffman: *Real-Time Rendering, 3rd Edition.* A.K. Peters, druhé vydání, 2008, iSBN 978-1568814240.

[19] Wolfgang Engel: *Shader X3: Advanced Rendering with DirectX and OpenGL.* Charles River Media, Inc., 2002, iSBN 0-321-22832-4.

[20] Zbyněk Mlčoch: Počet kostí v lidském těle[online]. `http://www.zbynekmlcoch.cz/informace/medicina/anatomie-lidske-telo/`, 2009-06-09 [cit. 2013-05-03].

Příloha A

Obsah DVD

Obsah přiloženého DVD je rozdělen do adresářů:

- *obrazky* - složka obsahující screenshoty z vytvořené aplikace
- *plakat* - složka obsahující obrázek plakátu prezentující vytvořenou aplikaci
- *spustitelny_soubor* - obsahuje přeloženou a spustitelnou aplikaci
  - */shaders/* - složka obsahující shadery k programu v jazyce GLSL
  - */res/* - složka obsahující zdrojové soubory (modely,textury,..)
  - $glew32.dll, glut32.dll$ - dll knihovny potřebné ke spuštění aplikace
  - *movement.exe* - spustitelný soubor
- */technicka_zprava/*
  - */tex/* - složka obsahující zdrojové soubory pro LaTeX
  - *zprava.pdf* - technická zpráva ve formátu pdf
- */zdrojove_soubory/*
  - */movement/*
    - */include/* - potřebné hlavičkové soubory
    - */bin/* - potřebné dll knihovny
    - */lib/* - potřebné lib. soubory
    - */shaders/* - shadery k programu v jazyce GLSL
    - */res/* - zdrojové soubory (modely,textury,..)
    - */src/* - zdrojové soubory (h., cpp.)
    - *movement.xxx* - nejnutnější soubory MSVS
  - *movement.sln* - soubor pro spuštění MSVS

Příloha B

Ovládání programu

- myš - ovládání natočení postavy spolu s kamerou
- w - pohyb dopředu
- s - pohyb dozadu
- a - pohyb doleva
- d - pohyb dprava
- f - použití akčních objektů
- b - natočení kamery okolo postavy vlevo
- n - natočení kamery okolo postavy vpravo
- q - výpis FPS na obrazovku
- c - zobrazení kolizních objektů knihovnou *Bullet*
- ESC - ukončení programu

Příloha C

Plakát