
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ**
**DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA**


# INTERAKTIVNÍ OPENGL DEMO

**INTERACTIVE OPENGL DEMO**

**BAKALÁŘSKÁ PRÁCE**
**BACHELOR'S THESIS**


**AUTOR PRÁCE** **PATRIK CHUKIR**
**AUTHOR**

**VEDOUCÍ PRÁCE** **Ing. TOMÁŠ MILET**
**SUPERVISOR**



**BRNO 2017**

# Abstrakt

Práce popisuje implementaci jednoduché 3D hry pomocí OpenGL. Věnuje se návrhu aplikace a konkrétní implementaci, která je výstupem této práce. V textu práce je postupně popsán způsob použití knihoven *Assimp*, *Bullet* a *IrrKlang*. *Assimp* pro načítání souborů formátu *dae* a *obj*. *Bullet* pro zachytávání kolizí a fyziku scény. A *IrrKlang* jako nástroj pro 3D ozvučení. Dále se práce podrobně věnuje osvětlení, stínům a implementaci *Skeleton animation*. Část práce je také věnována použité implementaci *skyboxu* a střídání dne a noci. V poslední kapitole jsou poté popsány výkonnostní testy výsledné implementace.



# Abstract

This Bachelor's thesis describe implementation of simple 3D game by OpenGL. The attention is paid to draft of this game and specific implementation, which is output of this work. In text of thesis is described usage of libraries *Assimp*, *Bullet* and *IrrKlang*. *Assimp* for loading an *.obj* and *.dae* model file. *Bullet* for detection of collision and physics of scene. And *IrrKlang* for sounds. Furthmore thesis deals with lighting, shadows and implementation of *Skeleton animation*. Part of thesis is dedicated to implementation of *skybox* and changing day and night. In last chapter are analyzed test of game.



# Klíčová slova

OpenGL, 3D hra, Assimp, Bullet engine, IrrKlang, Skeletal animation, Shadow depth map, Phongův osvětlovací model, Perlinův šum

# Keywords

OpenGL, 3D game, Assimp, Bullet engine, IrrKlang, Skeletal animation, Shadow depth map, Phong's lighting model, Perlin's noise


# Citace

CHUKIR, Patrik. *Interaktivní OpenGL demo*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Milet Tomáš.

# Interaktivní OpenGL demo

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Mileta. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Patrik Chukir
18. května 2017


## Poděkování

Tímto bych chtěl poděkovat svému vedoucímu Ing. Tomáši Miletovi za odbornou pomoc, dále Viktorii Chomaničové a Bc. Evě Bártové za gramatickou a slohovou kontrolu, a také své rodině za podporu během tvorby.

# Obsah


**1 Úvod** **4**

**2 Teorie** **5**
2.1 Phongův osvětlovací model . . . 5
2.2 Shadow depth map . . . 5
2.3 Skeleton animation . . . 6
2.4 Perlinův šum . . . 8

**3 Návrh aplikace** **10**
3.1 Obsah aplikace . . . 10
3.1.1 Čeho všeho bude aplikace schopna . . . 10
3.2 Načítání, ukládání modelů a jejich animací . . . 11
3.3 Fyzika letu a kolize objektů . . . 11
3.4 Osvětlení a stíny . . . 12
3.5 Střídání dne a noci . . . 12
3.6 Slunce a měsíc . . . 13
3.7 Vítr . . . 13
3.8 Zvuková kulisa a zvuk výstřelu a dopadu šípu . . . 15

**4 Implementace** **16**
4.1 Diagram Tříd . . . 16
4.1.1 Třída *Scene* . . . 16
4.1.2 Třída *BulletWorld* . . . 18
4.1.3 Třída *Model* a její potomci . . . 18
4.1.4 Třídy *Skybox* a *SunMoon* . . . 18
4.1.5 Třída *Hud* . . . 19
4.1.6 Třída *Wind* . . . 19
4.1.7 Třída *Animation* . . . 20
4.1.8 Třída *Sound* . . . 20
4.2 Zajímavé nebo problémové části implementace . . . 20
4.2.1 *Player*, *Weapon* a *Projectil* . . . 20
4.2.2 Shadery . . . 21
4.2.3 Implementace Skeletal Animation . . . 21

**5 Shodnocení náročnosti a efektivnosti implementace** **24**
5.1 Náročnost a výkon . . . 24
5.1.1 Vložení mnohokrát jednoho modelu . . . 24
5.1.2 vložení velkého množství malých modelů . . . 25

5.2 Vložení několikrát středního modelu . . . . . 25
5.3 Možnosti jak rozšířit aplikaci . . . . . 26

**6 Závěr** **27**

**Literatura** **28**

**Přílohy** **30**

**A Screenshots** **31**

**B Obsah CD** **34**

# Seznam obrázků


2.1 Na obrázku vidíme náčrtek modelu s kostmi pro *Skeleton animation* ve dvou různých časech (plná pro čas t, prázdná pro čas t+1). . . . . 7
2.2 Ukázka gratientních(g) a směrových(d) vektorů pro Perlinů šum . . . . . 9

3.1 Část diagramu struktury knihovny *Assimp*[7] uchovávající informace o jedné animaci jednoho nebo více *mesh* modelů. . . . . 11
3.2 Graf střídání denní a noční oblohy, kdy poloha osy fragmentu vůči ose 0 a 1 určuje poměr míchání textur pro tento fragment. Osy 0 a 1 na základě herního času rotují a dělí den na 8 částí. . . . . 13
3.3 Větrná mapa výsledné aplikace. Šipky znázorňují směr a velikost větru. Za oběma čtverci je možno vidět větrné stíny s výrazně nižší intenzitou větru. 14

4.1 Zjednodušený diagram tříd výsledné implementace . . . . . 17
4.2 Varianty provázání *Player*, *Weapon* a *Projectil*. Šipky naznačují, které třída bud obsahovat ukazatel na druhou třídu. . . . . 21
4.3 Graf konkrétní instance struktur *aiScene*. Jsou vidět všechny tři podstruktury, ve kterých jsou uložena data potřebné k realizaci animace. Přerušované čáry spojují ekvivalentní objekty. . . . . 23

5.1 Během vkládání koulí . . . . . 25
5.2 Po ustálení koulí . . . . . 25
5.3 Během vkládání krychlý . . . . . 25
5.4 Po ustálení krychlý . . . . . 25
5.5 Během vkládání tanků . . . . . 26

A.1 Screenshot ze střelecké pozice s naplým lukem . . . . . 32
A.2 Screenshot během úsvitu, jsou zde vidět stíny vržené terči a odlesk od vycházejícího slunce . . . . . 32
A.3 Detail terče, na kterém jsou vidět vykreslené vlastní stíny . . . . . 33
A.4 Zde je vidět stín vržený zabodlým šípem v terči . . . . . 33

# Kapitola 1

# Úvod

Tvorbou počítačových her se zabývá celé odvětví průmyslu. Pracují na nich mnohačlenné vývojářské týmy, i přes to se vyvíjí měsíce a roky. Ale co vše je vlastně potřeba vytvořit, abychom se mohli projít ve virtuální 3D scéně? Právě na tuto otázku by měla tato bakalářská práce reagovat. Tato bakalářská práce by měla popsat implementaci jednoduché hry, dema, ovšem veškerá implementace bude směřována k případnému pozdějšímu rozšíření do plné hry. Vše bude řešeno buď vlastní implementací nebo *freeware/open-source* knihovnami. Práce se bude zabývat realizací základního osvětlení, načtení a vykreslení modelů, interakcí s uživatelem, dále vyřešením kolizí pomocí externí knihovny. K těmto cílům budou použity knihovny založené na *OpenGL*, jmenovitě *Assimp*[7] pro načítání, *Bullet physic engine*[2] pro zachytávání kolizí a fyzikální model a *irrKlang*[6] pro ozvučení a knihovny *glew[13], glfw[1], glm[4]* pro základní práci s grafikou. Inspirací mi byly knihy Moderní počítačová grafika[11], 3D Game Engine Desight od David H. Eberly[5] a OpenGl Průvodce programátor[12].

# Kapitola 2

# Teorie

V této kapitole bude popsána netriviální teorie, která bude později použita ve výsledné aplikaci. Primárně se zaměří na teorii implementovanou mnou, nikoliv na teorii obsaženou v knihovnách třetích stran. Hlavním zdrojem pro tuto kapitolu bude sloužit kniha Moderní počítačová grafika[11].

## 2.1 Phongův osvětlovací model

Phongův osvětlovací model předpokládá, že většina světla nedopadá do kamery přímo, ale odrazem od objektů. Toto odražené světlo se dělí na tři složky *ambientní*, *difuzní* a *spekulární*. Kde *ambietní* složka představuje světlo rovnoměrně rozptýlené v prostoru, tedy osvětluje všechny fragmenty stejně. *Difuzní* složka, pak představuje světlo od zdroje přímo dopadající na fragment. Intenzita osvětlení *difuzní* složkou se odvíjí od úhlu dopadu. Čím blíže k normále, tím více je fragment osvětlen. Poslední složka, *spekulární* neboli zrcadlová, představuje světlo odrážející se od fragmentu do kamery. Tedy její intenzita se mění na základě polohy kamery vůči úhlu dopadu světla tzn. jak moc se světlo odráží směrem ke kameře. Phongův model v tomto případě předpokládá ideální odraz, tedy úhel dopadu se rovná úhlu odrazu. Intezitu celkového ovětlení fragmentu lze obecně spočítat vzorcem:

$$I_V = I_A \vec{r_a} + \sum_{k=1}^{N} I_{L_k}[r_s(\vec{v} \cdot \vec{r_k})^h + r_d(\vec{l_k} \cdot \vec{n})][11] \tag{2.1}$$

Vzorec 2.1 je obecný pro $N$ zdrojů světla. V aplikaci bude využíváno jako zdroje světla pouze slunce, pak tedy lze vzorec zjednodušit na:

$$I_V = I_A r_a + r_s(\vec{v} \cdot \vec{r_k})^h + r_d(\vec{l_k} \cdot \vec{n}) \tag{2.2}$$

V obou vzorcích přestavuje $I_A$ intenzitu *ambientní* složky a $r_a$ její barvu. V prvním vzorci se pak sčítají všechny *difuzní* a *spekulární* složky všech zdrojů světla. V případě této bakalářské práce se tím pádem suma redukuje pouze na jeden vzorek.

## 2.2 Shadow depth map

*Shadow depth map*, českým ekvivalentem stínová paměť hloubky[11], je algoritmus na výpočet vlastních i vržených stínů hmotných těles. Vyznačuje se vysokou rychlostí výpočtu a nízkou kvalitou stínů, vyšší paměťovou náročností pro více zdrojů světla. Algoritmus

spočívá ve vytvoření hloubkové mapy, podobně jako při řešení viditelnosti pomocí *z-buffer* algoritmu, kdy je v mapě uložená vzdálenost nejbližšího bodu ke zdroji světla. Při vykreslování se porovná hloubka daného bodu s hodnotou v mapě. Pokud je daná hodnota vyšší, pak je fragment ve stínu, jestliže je ale nižší, tak ve stínu není. V důsledku použití rastrové mapy se u tohoto algoritmu projevují všechny problémy spojené s mapováním textur, jako například *aliasing*, problémy s rozlišením mapy. Dále může v důsledku nepřesnosti rasterizace vzniknout jemná odchylka. Následkem toho hloubka bodu vyjde o něco větší, než při zápisu do hloubkové mapy a začne vrhat stín sám na sebe. Algoritmus této metody se dá zapsat jako:

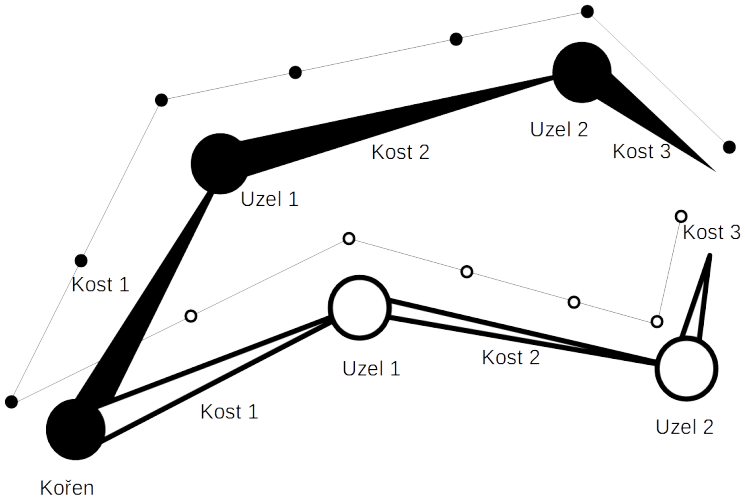
Algoritmus 2.1: Algoritmus Shadow depth map

```
for(int i=0;i<countOfLight;i++){
        drawToShadowMap(&shadowBuffer[i]);
}
draw()
//in fragment shader -- for each fragment
in vec3 coord // coordinate of fragment

// matrix for transform from model space to lightspace
uniform mat4[] LightMVP;
// array of shadow depth map
uniform sampler2DShadow ShadowSampler[];

// array of structur for light, function getLight
// return sum of ambient, diffuse and specular part
uniform lightSource light;

vec3 sum(0,0,0);
for(int i=0;i<countOfShadowMaps;i++){
        L-coord = LightMVP[i] * coord;
        float depth =texture2D(ShadowSampler[i], L-Coord.xy);
        if(depth<L-Coord.z+0.0001){
                sum += 0.5 * getLight(light[i]);
        }else{
                sum += getLight(light[i]);
}
```

Algoritmus 2.2 je psán v podstatě v *glsl*, ale tento jazyk poskytuje i funkci přímo pro provední porovnání s hodnotou v *shadow map texturProj(shadowMap,lightCoord)*. Ovšem to by nebylo moc názorné.

## 2.3 Skeleton animation

*Skeleton animation* je metoda inspirovaná stavbou lidského těla. To znamená, že společně s kostí se vždy pohybuje i od ní celá pokračující část těla. Příkladem může být pohyb stehenní kosti, který způsobí následný pohyb i kosti holení, lýtkové, kostmi kotníku a celého




Obrázek 2.1: Na obrázku vidíme náčrtek modelu s kostmi pro *Skeleton animation* ve dvou různých časech (plná pro čas t, prázdná pro čas t+1).

chodidla. Základní *Skeleton animation* využívá dvě vrstvy, síť bodů jako kůži a stromovou strukturu kostí viz. obrázek 2.1. Pokud je potřeba realističtějšího chování, používají se komplikovanější verze, kdy mezi vrstvou vrcholů a kostí je ještě jedna nebo dokonce několik vrstev simulující svalstvo. Toto vylepšení zabraňuje vzniku zlomů v kůži, či prolnutí vrcholů z jedné strany na druhou.

Ovšem pro potřeby implementace této práce dostačuje základní varianta, neboť výše zmíněné vady se projevují primárně u modelů s rotacemi o velký úhel. Pro jednoznačný popis a vykreslení jednoduché varianty *Skeleton animation* je potřeba znát:

- pro vrchol:
  - pozici, uv souřadnice, normálu
  - seznam kostí, kterými je ovlivněn
  - seznam vah, poměr ovlivnění, pro tyto kosti
- pro kost:
  - transformační matici pro přechod ze souřadní soustavy kosti do soustavy bodů, dále nazývána offsetová matice nebo matice odsazení
- pro uzel:
  - transformační matici pro přechod ze soustavy tohoto uzlu do soustavy rodičovského uzlu pro každý klíčový snímek animace

K výpočtu polohy vrcholu pro daný snímek se musí vypočítat kompletní transformační matice pro všechny kosti, kterými je vrchol ovlivněn. Matici kosti lze získat vynásobením všech matic uzlů od uzlu před danou kostí až ke kořeni, poté ještě násobeno transformační maticí kosti do souřadné soustavy bodů. Tyto matice po vynásobení příslušnými váhami a jejich sečtení, dávají výslednou transformační matici pro daný vrchol. Stačí samozřejmě

vypočítat pro každou kost matici pouze jednou v daném snímku a pak při vykreslování vrcholů k ní jenom přistoupit. Matice pro uzly se mění v závislosti na čase animace. Ukládány jsou povětšinou pouze matice pro klíčové snímky. Je-li čas animace mezi dvěma klíčovými snímky, pak je potřeba k určení matice pro jeden uzel v čase t udělat vážený průměr matic z klíčového snímku před a po:

$$M_t = \frac{M_{pred} \cdot (t - T_{po})}{M_{po} \cdot (T_{pred} - t)} \tag{2.3}$$

Konkrétní implementace výpočtu a zpracování matic je uvedena v podkapitole 4.2.3, kapitola Implementace. Vypočtou-li se všechny matice uzlů a z nich finální matice kostí, je transformační matice vrcholu dána vzorcem:

$$M_t = \sum_{0}^{n=pocetkosti} M_n \cdot vaha_n \tag{2.4}$$

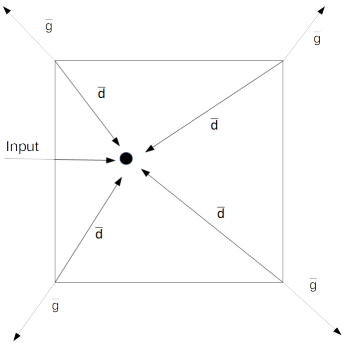
## 2.4 Perlinův šum

Perlinův šum byl navržen profsorem Kenem Perlinem v roce 1983 a v 1985 vydává text kde jej popisuje[9]. Tento šum se často používá k dosažení přirozeně náhodného jevu, jako jsou třeba letokruhy dřeva nebo skvrny na slunci. Používají se tři varianty Perlinova šumu a to 1D, 2D a 3D. 1D pro vytvoření linek, které vypadají jako kreslené rukou. 2D šum se používá například pro tvorbu 2D textur, také u výškových map náhodného terénu nebo pro zašumění normál textury pro dosažení dojmu hrbolatého povrchu. Trojrozměrná varianta šumu je asi nejzajímavější, protože je schopna generovat náhodné 3D objekty nebo prostory, například jako herní mapy s velkým počtem prvků.

Další z předností Perlinova šumu je, že profesor Perlin si jej nikdy nenechal patentovat. Díky tomu jej využívá spousta nekomerčních i komerčních aplikací. Dá se k němu dohledat spousta informací a není problém zjistit jak jej implementovat, neboť i sám profesor na svých stránkách[8] uvádí hned dvě implementace Perlinova šumu.

V roce 2001 profesor Perlin vydal článek o *Improved noise*[10], kde popisuje nedostatky předchozí varianty šumu a cestu jak nedostatky odstranit. Já jsem se rozhodl ve své aplikace využívat *Improved Noise*, který se od standardního Perlinova šumu liší jen v gradientních vektorech a upravení funkce fade(). Šum generuje náhodnou spojitou n-rozměrnou funkci. Hodnoty této funkce jsou počítány pomocí lineární interpolace a skalárního součinu gradientních a směrových vektorů. Vzorec 2.5 ukazuje variantu pro 2D šum. Kde *lerp()* představuje funkci pro lineární interpolaci a funkce *grad()* provádí skalární součin směrového vektoru a gradientního vektoru vybraného permutační funkcí p(). V každém volání *grad()* se použije jiný směrový vektor.

$$Noise(x, y) = lerp(v, lerp(u, grad(), grad()), lerp(u, grad(), grad())); \tag{2.5}$$




Obrázek 2.2: Ukázka gratientních(g) a směrových(d) vektorů pro Perlinů šum

# Kapitola 3

# Návrh aplikace

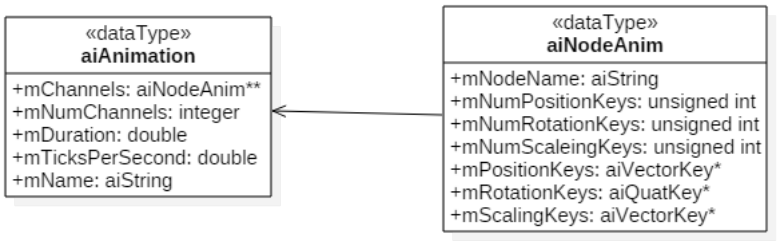
Na začátku této kapitoly bude ujasněno co vše vlastně výsledná aplikace bude umět. Dále pak v jednotlivých podkapitolách budou tyto schopnosti podrobně popsány, zmíněné možné problémy, které z nich plynou a následně budou navrhnuta řešení. Bude-li více možností jak na danou vlastnost aplikace pohlížet, popřípadě jak s ní pracovat, bude zdůvodněno proč bylo zvolené právě ono výsledné řešení. Pokud některá z vlastností aplikace bude řešena knihovnou třetích stran, bude v rámci popisu této schopnosti i navrhnuto její použití a práce s ní. V rámci tohoto popisu bude i zdůvodněno proč je vlastně použita. Určitě budou tedy zmíněny knihovny jako je *Assimp*[7] nebo *Bullet physic library*[2], dále jen *Bullet*. V kapitole 4 Implementace bude uveden výsledný diagram tříd vzniklý z tohoto návrhu.

## 3.1 Obsah aplikace

Cílem práce je vytvořit demo 3D hry, lukostřelecké střelnice. Demo bude tvořeno jen jednou scénou bez menu. Scéna bude obsahovat několik terčů a střeleckých pozic. Bude umožňovat pohyb avatárem ve vykresleném prostoru, tento hráč bude vybaven lukem a několika šípy. Hráč vždy uvidí *head-up display*, dále jen *hud*, složený z počítadla bodů, ukazatele síly a směru větru, ukazatele zbylých šípů a zaměřovače. Pro šípy bude zajištěno fyzikální chování, ovlivnění silou a směrem větru. Scéna bude osvětlena pouze sluncem, dle herního času.

### 3.1.1 Čeho všeho bude aplikace schopna

- Načítání a práce s modely a animacemi
- Fyzika letu a kolize objektů
- Osvětlení scény a stíny pro usnadnění odhadu vzdálenosti a realističnost
- Střídaní dne a noci
- Dostatečná náhrada slunce
- Vítr, plynulá změna jeho směru, poryvy, závětří
- Zvuková kulisa a zvuk výstřelu a dopadu šípu




Obrázek 3.1: Část diagramu struktury knihovny *Assimp*[7] uchovávající informace o jedné animaci jednoho nebo více *mesh* modelů.

## 3.2 Načítání, ukládání modelů a jejich animací

Modelem se rozumí skupina vrcholů (angl. *mesh*). Aby aplikace mohla model vykreslit, musí být schopna jej načíst ze souboru a uložit si ho ve vhodné reprezentaci, a vědět kterou texturu má použít. K načtení bude použita knihovna *Assimp*[7], aby bylo možno načítat více různých formátů. Model se bude ukládat jako pole vrcholů, kde vrchol bude představovat struktura. Ta bude obsahovat pozici vrcholu v souřadném systému modelu, uv koordináty, normálu, indexy a váhy kostí, na které vrchol bude navázán, více v sekci 4.2.3 o *Skeleton animation*. V tomto poli bude každý vrchol pouze jednou, neboť aplikace bude vykreslovat metodou indexovaného kreslení.

Animace jsou načítány společně s modelem ze souboru, *Assimp* je ukládá do dynamického seznamu struktur *aiAnimation*. Každá z těchto struktur představuje jednu animaci modelu a vidět ji můžeme na obrázku 3.1.

Vzhledem k tomu, že nad animací budou prováděny jen vyhledávací a čtecí operace, je toto uspořádání dostačující. Pokud by ovšem vyhledávací a výpočetní funkce byly přidány do stejné třídy, kde už jsou metody pro práci s modelem, tak by se tato třída stala velice přetíženou a těžko odladitelnou. Proto bude vhodné zabalit tuto strukturu do vlastní třídy, obohatit ji o potřebné metody a ve třídě pro model zpřístupnit jen výsledné transformační matice jednotlivých kostí.

Modely se vyskytují ve spoustě různých formátech a variacích, rozhodl jsem se pracovat pouze se dvěma formáty:

- *Wavefront* od *Wavefront Technologies* s příponou *obj* pro modely bez animací
- *COLLADA* od *Sony Computer Entertainment* s příponou *dae* pro modely s animacemi.

Ke čtení modelů je sice použit *Assimp::Importer*, který je schopen číst širokou škálu formátů, ovšem aplikace je odzkoušena pouze na těchto dvou formátech. Při jiných formátech by potom mohlo dojít k problému při přiřazování do třídy *Model* viz diagram tříd 4.1.

## 3.3 Fyzika letu a kolize objektů

V této části bude navrženo řešení výpočtu pohybu těles v gravitačním poli a výpočet projevu sil, jako například výstřel šípu. S tímto chováním i úzce souvisí kolize mezi objekty.

Fyzikální popis těchto jevů a chování je relativně složitou záležitostí a vytvoření dostatečně rychlé implementace těchto jevů by výrazně přesáhl rámec této práce, pro to jsem se rozhodl použít externí knihovnu *Bullet physic engine*[2]. Tato knihovna poskytuje i možnost řešení kolizí a jejich zachytávání.

Ovšem nepracuje přímo s modely co jsou vykreslovány, ale s vlastními objekty typu *btCollisionObject*, proto je potřeba vytvořit vazbu mezi vykreslovanou scénou a scénou, se kterou pracuje *Bullet*, povětšinou *btDiscreteDynamicWorld*. Tato vazba musí být obou směrná neboť při každém snímku se musí aktualizovat poloha vykreslovacích modelů polohou jejich fyzikálních ekvivalentů. A naopak *Bullet* vrací kolize jako pole dvojic *btCollisionObject* kdy pro každý objekt z tohoto pole se musí zavolat funkce pro reakci na kolizi.

## 3.4 Osvětlení a stíny

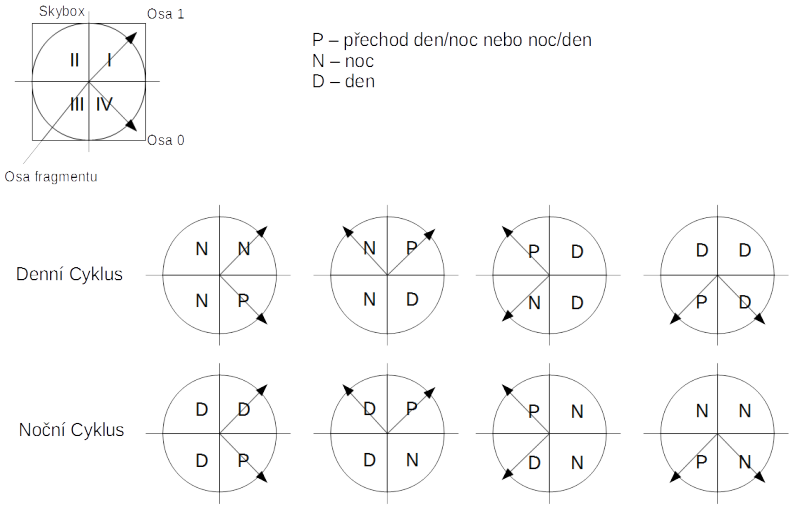
Scénu bude osvětlovat pouze jeden zdroj směrového světla představující slunce, měsíc svítit nebude, za použití Phongova osvětlovacího modelu. Jeho poloha a intenzita se bude měnit dle herní denní doby. Stíny budou vypočteny a vykresleny metodou *shadow maping* jenž je popsána v kapitole 2.2. Protože aplikace bude mít jen jeden zdroj směrového světla, je tato metoda vhodná, právě díky své rychlosti a stačí také vytvořit pouze jednu mapu. Tato mapa se bude přepočítávat pro každý snímek zvlášť, může se tedy vždy namapovat na scénu tak, aby měla co nejmenší přesah přes plošné promítnutí tělesa vymezeného pohledem kamery a zdrojem světla. Tento způsob namapování umožňuje pracovat s menší texturou, což urychlí její výpočet a zabírá tak méně paměti, nebo pracuje se stejně velkou texturou, díky čemuž se získají lepší detaily. Poloha tohoto zdroje světla bude navázána na příslušný objekt, který jej bude představovat. Společně s tímto objektem bude zdroj putovat po obloze a díky *diffusní* části světla se na základě úhlu mezi zdrojem a horizontem bude měnit i intenzita osvětlení.

## 3.5 Střídání dne a noci

Na *skybox* se budou mapovat celkem dvě textury pro denní a noční oblohu. Střídání dne a noci je způsobeno změnou poměru míchání noční a denní textury pro jednotlivé fragmenty.

Poměr pro každý fragment je dán úhlem mezi jeho osou (polopřímka ze středu přes něj) a osami 1 a 0, jak je vidět na obrázku 3.2. Pokud se fragment nachází ve stejném kvadrantu jako osa 0, je poměr vypočítán na základě jeho úhlu. Je-li úhel 45°, je poměr míchání 0:1 v neprospěch aktuálního cyklu, naopak pro -45° je poměr 1:0 pro aktuální cyklus. Mezi těmito hodnotami je potom nepřímá úměra. Ve zbylých třech kvadrantech je poměr vždy buď 1:0 nebo 0:1, odvíjí se od toho, která čtvrtina denního/nočního cyklu je:

- kvadrant u osy 1:
  - pro 1 - 3 část cyklu 0:1
  - pro 4 část cyklu 1:0
- kvadrant naproti ose 0:
  - pro 1 a 2 část cyklu 0:1
  - pro 3 a 4 část cyklu 1:0
- kvadrant naproti ose 1:




Obrázek 3.2: Graf střídání denní a noční oblohy, kdy poloha osy fragmentu vůči ose 0 a 1 určuje poměr míchání textur pro tento fragment. Osy 0 a 1 na základě herního času rotují a dělí den na 8 částí.

- pro 1 část cyklu 0:1
- pro 2 - 4 část cyklu 1:0

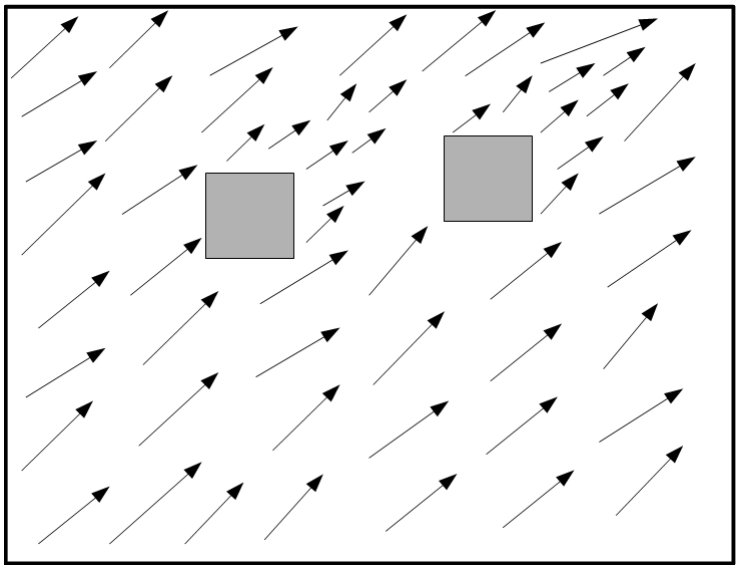
Celý den má tedy 8 částí, 4 denní a 4 noční, což trvá dvě otočení os o 360°. Aplikace při spuštění začíná v čase nula, tedy úsvitem nového dne.

## 3.6 Slunce a měsíc

Slunce a měsíc se realizují pomocí plochých modelů rotujících kolem středu *skyboxu*. Doba oběhu musí trvat celý den, tedy dvě otočení os *skyboxu*. V rámci třídy pro slunce a měsíc se bude řešit i změna směru světla příslušného objektu. Pro správný pohyb se musí zajistit dvě věci. Za prvé správné natočení plochy objektu, aby nebyl vidět jako elipsa či jako pouhá čára. A za druhé uchování vzdálenosti od středu *skyboxu*, potažmo avatara. Dosáhne se toho pomocí vektoru $\vec{a}$ mezi středem a objektem a rotací $\vec{a}$ i objektu kolem stejné osy (vyjádřená stejným předpisem, počítána vždy v relativní soustavě objektu nebo globální soustavě v případě $\vec{a}$). Poloha objektu se potom vždy nastaví podle vektoru $\vec{a}$ posunutého do aktuálního středu *skyboxu*.

## 3.7 Vítr

Vítr je na vykreslování relativně jednoduchý jev, není totiž vidět, na druhou stranu má spoustu viditelných projevů. Prozatím se bude práce zabývat pouze působením větru na šíp za letu. Vítr bude mít globální sílu a směr, který se bude v čase měnit. Takto by byl vítr všude stejný, jenže vítr se mění. Vítr je ale možno zastavit, vznikají tak závětří, jinde pofukuje více, či méně, tamhle trochu jiným směrem. Aby se dalo dosáhnout tohoto chování



Obrázek 3.3: Větrná mapa výsledné aplikace. Šipky znázorňují směr a velikost větru. Za oběma čtverci je možno vidět větrné stíny s výrazně nižší intenzitou větru.

alespoň do jisté míry realističnosti, je potřeba určit závětrná místa, větrné stíny, a pro každý bod vypočíst drobnou odchylku směru a síly, jak je znázorněno v obrázku 3.3.

K výpočtu větrných stínů se dá použít téměř jakákoliv osvětlovací metoda, protože vítr se v podstatě v ničem neliší od světla. Stále to jsou paprsky/proudy a pouze jejich projevy se liší, místo osvětlení fragmentu posunou objekt, na který dopadly. Já jsem se rozhodl použít metodu *shadow maping* a znovu využít již nutně implementované funkce pro výpočet stínů osvětlení. Tímto postupem lze získat větrné stíny, ovšem síla a směr větru se měnit nebude. Proto výsledná mapa ze *shadow mapingu* bude zašuměna, například již zmíněným *Perlinovým šumem*. Pro tuto bakalářskou práci připadají dvě možnosti. Buď zašumit vždy celou mapu, nebo pouze aktuálně čtený bod. Vzhledem k tomu, že vítr bude ovlivňovat pouze několik těles a pro každé těleso se bude číst jen jeden bod z mapy (důvod proč jen jeden bod bude vysvětlen dále). Proto bude rychlejší na výpočet použít šum vždy až při čtení bodu.

Ještě zbývá vyřešit, jak se bude vítr projevovat. K tomu lze přistoupit dvěma způsoby: jednodušším a méně reálným nebo složitým a reálnějším. Jednoduší znamená, že se vítr bude projevovat jako pouhá směrová síla, bez ohledu na dopadovou plochu tělesa. Složitý přístup znamená vypočítat dopadovou plochu a z ní odvodit výslednou sílu větru. K zjištění dopadové plochy by bylo ovšem potřeba každý fragment tělesa otestovat na jeho polohu v mapě, jestli je ve stínu nebo ne. Pokud by se tento test prováděl na CPU, došlo by k výraznému zpomalení aplikace, na GPU by k zpomalení nedocházelo, a v ničem by se to nelišilo od vykreslování stínů, ovšem standardní knihovny pro práci s GPU neumožňují přesun dat z GPU zpět do CPU. Pokud by šlo o simulátor plachetnice, tak by se určitě muselo jít složitější cestou, ale tím, že je v rámci výsledné práce tvořena lukostřelecká střelnice, vítr bude primárně ovlivňovat šípy, které mají relativně malou plochu a dalo by se říci, že stejnou ze všech stran, bude tedy dostačující pouze jednoduchá cesta, která byla popsána výše. Z toho také vyplývá proč se bude vždy číst jen jeden bod mapy pro těleso a stačí tím pádem aplikovat šum až při čtení bodu.

## 3.8 Zvuková kulisa a zvuk výstřelu a dopadu šípu

Pro výsledný efekt by bylo poněkud nezvyklé, kdyby se během hry neozývaly žádné zvuky, tento drobný detail by způsobil poloviční zážitek, i kdyby by byla hra sebelepší. Ovšem podobně jako v případě působení sil i šíření zvuku v 3D prostoru, to není zcela triviální záležitostí. Za prvé, pokud je třeba rozpoznat zdroj zvuku ve scéně, je potřeba pracovat s různou úrovní hlasitosti pro každou stranu. A za druhé přehrávaný zvuk musí běžet ve vlastním vlákně, jinak by pozastavil vlastní aplikaci, do doby než by skončilo jeho přehrávání. Z těchto dvou důvodů jsem zvolil externí knihovnu *IrrKlang*[6], která obě požadované vlastnosti poskytuje skrze jednu funkci. Vývojáři této knihovny se opravdu řídili heslem v jednoduchosti je síla a pro většinu případů stačí metoda *play3D(paht,option)*, kde *option* určuje, zda-li zvuk běží ve smyčce nebo ne.

# Kapitola 4

# Implementace

V první části této kapitoly bude popsán diagram tříd výsledné aplikace, v diagramu jsou uvedeny pouze třídy, jenž jsou nějak zajímavé nebo důležité. Třídy, které slouží jen k zapouzdření a zpřehlednění kódu budou vynechány. V druhé části této kapitoly budou podrobně popsány zajímavé problémy, na které jsem při implementaci narazil.

## 4.1 Diagram Tříd

Na obrázku 4.1 je zjednodušený diagram tříd výsledné aplikace. Výsledná Aplikace je složena ze tří hlavních částí:

1. Třída *Scene*, která zapouzdřuje celou aplikaci.
2. Třída *BulletWorld*, přes kterou je řešena veškerá fyzika.
3. Třída *Model*, která je výchozí třídou pro většinu prvků ve scéně.

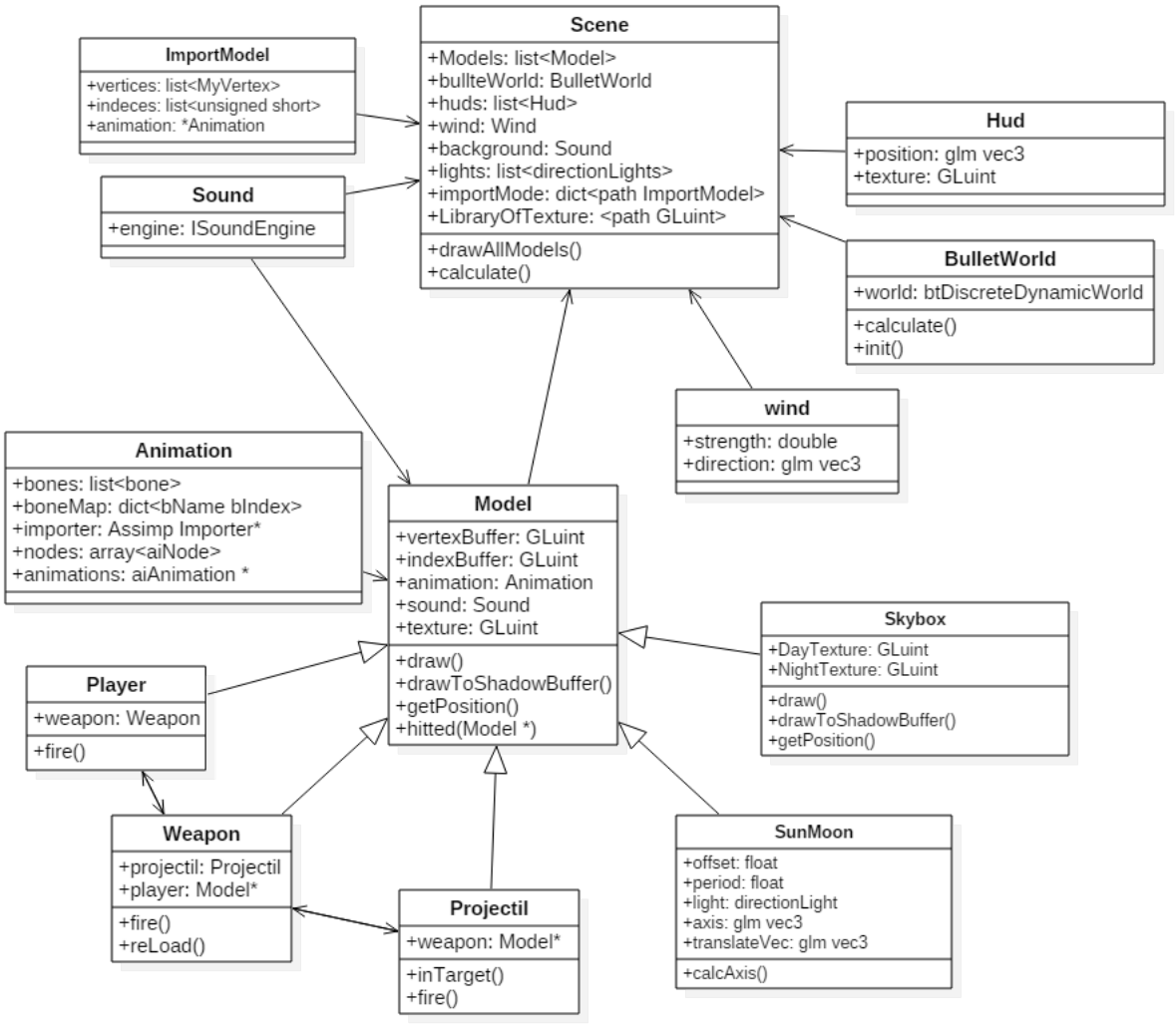
Dále pak má aplikace mnoho dalších tříd, jako třídy pro hud nebo pro práci s animacemi či zvukem, mezi zajímavější pak patří třídy *Skybox* a *SunMoon*, které mají na starosti celé střídání dne a noci a pohyb zdrojů světla.

### 4.1.1 Třída *Scene*

Třída *Scene* je ústřední třídou, z níž se přistupuje do třídy *BulletWorld* pro detekci kolizí a změny poloh modelů, na základě působení sil. Zde se také řeší jednotlivé kroky vykreslování, jako vykreslení do *wind mapy, shadow mapy* a výsledné vykreslení.

Třída *Scene* tím pádem obsahuje převážně funkce pro nastavení objektů, jejich vkládání a případné nastavení propojení mezi objekty, které spolu nějak komunikují (např.: zbraň a projektil nebo objekt a jeho fyzikální představitel). Většina chování objektů je v jejich vlastní režii, třída *Scene* volá jen funkce pro jejich nastavení při vzniku a během cyklu už pouze vykreslovací funkce.

Tato třída ještě obsahuje slovníky načtených modelů a textur. Slovník pro modely má tvar *<cestaKsouboru,ImportModel>*. Při načítání souboru se zkontroluje, zdali již není ve slovníku, jestliže je, použije se příslušný záznam. Není-li tomu tak, tak se soubor načte a zpracuje do třídy *ImportModel* a vrátí se opět záznam ze slovníku. U textur to probíhá podobně jen místo třídy *ImportModel* se použije přímo adresa textury na GPU.




Obrázek 4.1: Zjednodušený diagram tříd výsledné implementace

### 4.1.2 Třída *BulletWorld*

*BulletWorld* provádí operace nad *btDiscreateDynamicWorld* z knihovny *Bullet*[2], tedy inicializační a úklidovou (v rámci destruktoru třídy) metodu pro *btDiscreateDynamicWorld*, dále metody pro několik variant vkládání *btCollisionObject* a nejdůležitější metoda je *calculate()*, která zjistí a vrátí list aktuálních kolizí. Tato metoda je využita ve třídě *Scene*, která dále zajistí zavolání metody *hitted(Model*)* pro každý z kolizních objektů, kdy parametr této funkce je vždy objekt, se kterým došlo ke kolizi.

U této části je problém, protože není-li volání této funkce o stejné frekvenci jako tiky v *Bullet engine*, tak se některé „stejné kolize“ zachytí několikrát. Ony to nejsou stejné kolize, ale dojde-li ke střetu dvou objektů, tak doba jejich dotyku není nekonečně malá a může přesáhnout do několika tiků v *btDiscreteDynamicWorld*, což vytvoří několik záznamů o kolizi. V důsledku tohoto jevu se v seznamu kolizí některé kolize vyskytují několikrát, k ošetření dochází ve třídě *Scene*. Všechny kolize dané dvojice se ze seznamu vymažou, když třída *Scene* zavolá metodu *hitted(Model*)* pro daný pár objektů.

### 4.1.3 Třída *Model* a její potomci

Téměř všechny objekty ve scéně dědí od této třídy. Jediné které od ní nedědí je třída *Wind* a třída pro prvky hud. V třídě *Model* jsou tedy implementace základního vykreslování a obecná práce s polohou, vytváření *bufferů*, výchozí implementace metody *hitted(Model*)*. Většina potomků tyto metody dědí nezměněné.

Třídy *Weapon a Projectil* přepisují metodu *getPosition()*. Poloha *Weapon* se určuje na základě polohy přiřazené třídy *Player*. U třídy *Projectil* je to komplikovanější, neboť má tři stádia života, a to ve zbrani, v letu a v zabodnutí. V případě stadií ve zbrani a zabodnutí je poloha určována na základě modelu, ve kterém je (zbraň nebo objekt, ve kterém je zabodnut). Během letu se jeho poloha odvozuje jako u ostatních modelů, tedy od polohy jeho fyzikálního ekvivalentu. Více o vztahu těchto tří tříd bude řečeno v podkapitole 4.2.1

### 4.1.4 Třídy *Skybox* a *SunMoon*

Tyto třídy sice také dědí od třídy *Model*, jako jediné ale v podstatě dědí pouze rozhraní neboť, tři hlavní metody *draw()*, *drawToShadowMap()* a *getPosition()*, celé přepisují a ani nevlastní svůj fyzikální ekvivalent v *btDiscreteDynamicWorld*. *Skybox* má i vlastní speciální sadu *shaderů*.

Změnu denní a noční oblohy řídí funkce *calcAxis()*, která přepočítává rotace os a také na základě jejich úhlů vůči osám x a y určuje fázi denního/nočního cyklu. Ty jsou vysvětleny v kapitole 3.5. Původně se fáze cyklu měly měnit pouze na základě času, ale v tom případě by musely všechny části trvat stejně, což sice odpovídá realitě, ale nevypadá to věrohodně. Rozednívání je moc pomalé a stmívání také, jako více věrohodná se ukázala varianta, kdy je přechod mezi druhou a třetí fází denního i nočního cyklu, obrázek 3.2, trojnásobně zrychlen a zde ušetřený čas je rozložen mezi zbylé tři přechody. Úroveň tohoto zrychlení byla zjištěna empiricky, metodou pokus omyl. Tahle varianta vytváří i lepší koordinaci mezi stavem oblohy a pohybem slunce, tedy když slunce vyjde, tak se rychle rozední a pak slunce pomalu putuje po obloze během dne. Jakmile slunce zapadne, tak se rychle setmí. Při stejné rychlosti všech fází by se slunce, mělo-li stejnou dobu oběhu jako byla délka dne, drželo by se ve stejné části oblohy. Pokud ale mělo dobu oběhu delší, tak by se slunce zpožďovalo o proti rozednění v následujících dnech.

*Skybox* využívá jiný *shader*, jak již bylo zmíněno, kvůli způsobu určování poměrů pro *multiTexturing*. Kvůli výpočtu poměru je potřeba, aby *shader* vždy věděl, co je za část dne a znal osy. Na druhou stranu vůbec nepracuje se světly, proto má *skybox* i jinou *draw()* metodu. *Skybox* i *SunMoon* se do *shadow mapy* vůbec nevykreslují, proto nad nimi vůbec metoda *DrawToShadowMap()* není volána.

*SunMoon* je třída pro nebeská pohyblivá tělesa, má metody pro nastavení doby oběhu, sklonu osy vůči povrchu, posun času východu a nezačíná tak na horizontu. Podobně jako Skybox i třída SunMoon používá vlastní, v tomto případě výrazně jednoduší shader, protože nad ní se neprovádí ani výpočet stínů a osvětlení ani *Skeleton animation*. Takže by bylo zbytečné mít tak rozsáhlí shader. Z toho opět plyne jiná *draw()* metoda.

### 4.1.5 Třída *Hud*

*Hud* je výchozí třídou pro všechny prvky *head-up diplay*. Zajišťuje namapování příslušných textur na určené místo na obrazovce. Potomci této třídy jsou třídy na zobrazování stavu toulce (třída *ArrowStack*), zaměřovač (*CrossHair*) a univerzální třída na výpis řetězce (*HitsHud*).

*ArrowStack* zobrazuje pouze na obdélník, jenž je mu nastaven. Nastavená oblast obrazovky je rozdělena horizontálně na tolik oblastí, na kolik je nastaven atribut *initialCount*, zde pak zobrazí tolik textur, kolik je hodnota *countOfArrow*, každou texturu na jednu oblast. Díky tomu je znovu použitelný na n-násobné zobrazování jakékoliv textury v HUD.

*HitsHud* zobrazuje řetězec znaků pomocí textury znakové tabulky. V nastavení se určí velikost jednoho znaku a počátek řetězce. Poté se při výpisu pro každý znak spočítá obdélníček a uv souřadnice v textuře na základě *ascii* hodnoty, modulo počtem sloupců textury se získá x souřadnici a děleno počtem řádků y souřadnici. Výpočtem byl získán buď levý dolní nebo levý horní roh. Který z těchto rohů byl vypočten se odvíjí od orientace textury. Tato třída je počítána na orientaci, kdy bod [0,0] leží v levém dolním rohu a znak s hodnotou 0 je v levém horním.

### 4.1.6 Třída *Wind*

V režii této třídy je zajištění *wind map* a čtení na základě souřadnic pomocí metody *getLocalWind(glm::vec3)*. Metoda vrací proměnou typu *glm::vec3* vypočítanou vzorcem 4.1. Jinak je tato třída dosti podobná implementaci stínů. Pro vykreslování do *wind map* se používá speciální metoda ve třídě *Scene*, ale jde jen o nastavení parametrů pro zápis, jinak by se dala použít stejná metoda jako pro zakreslování do *shadow map*. V obou těchto metodách se volá pro jednotlivé modely ta stejná metoda, *drawToShadowBuffer()*.

$$\vec{v} = \overrightarrow{direction} \cdot PerlinNosice(x, y) \cdot strength \cdot shadow \tag{4.1}$$

Ve vzorci 4.1:

- direction = směr větru
- strength = síla větru
- *PerlinNoise(x,y)* způsobí zašumění směru a síly, pro náhodnější jev
- shadow je koeficient pro závětří, 1 pro volný prostor a 0,5 pro závětří

### 4.1.7 Třída *Animation*

Pokud je při načítání modelu zjištěno, že obsahuje animaci, tak je vytvořena třída *Animation* a ta je pak navázána na vznikající ImportModel. V této třídě jsou obsaženy kosti a uzly pro příslušnou animaci a odkaz na *aiAnimation* obsahuje vlastní animaci. Kosti jsou uloženy jako vektor mých struktur *Bone*, ale pole uzlů a animace využívá struktury *aiNode* a *aiAnimation*. Na obrázku 3.1 je vidět právě *aiNodeAnim*, která slouží jako mapovací třída mezi animací a polem uzlů. V případě rozšíření aplikace o možnost vytvořit více animací pro jeden model, budou se úpravy týkat pouze třídy *Animation*, více v sekci 5.3 Možnosti rozšíření. Prozatím je ale počítáno pouze s jednou animací. zšíření, zatím počítáno pouze s jednou animací.

### 4.1.8 Třída *Sound*

Třída *Sound* je určená pro práci s knihovnou *IrrKlang*[6]. Hlavním důvodem vzniku této třídy je zpřehlednění a zapouzdření kódu, tak aby *Model* při určité situaci mohl zavolat metodu *play3D(path)* a o více se nemusel starat. Knihovna *IrrKlang* umožňuje asynchronní přehrávaní, tedy přehrávání zvuků nezpožďuje scénu ani nepozastavuje výpočty aplikace.

Všechny instance této třídy by měly pracovat s jedinou instancí *ISoundEngine* z knihovny *IrrKlang*, která je inicializována v rámci třídy *Scene* a modelům ve scéně je pak předána pouze jako parametr. Důvod je ten, že pro každou instanci *ISoundEnginu* se musí nastavit poloha posluchače a většina modelů nemá přístup k poloze hráče.

## 4.2 Zajímavé nebo problémové části implementace

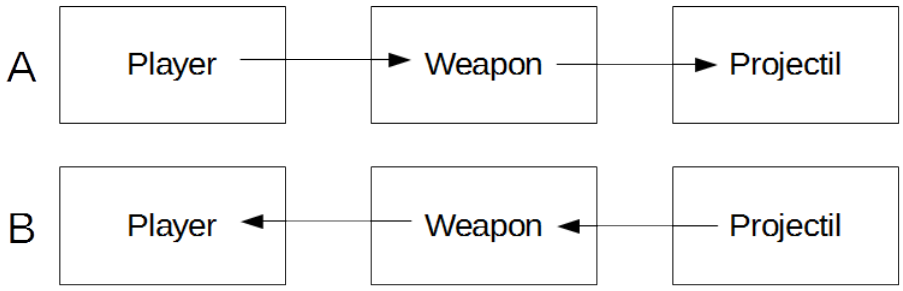
Tato podkapitola podrobněji popíše implementaci míst, kde se musel řešit nějaký implementační problém nebo bylo použito netradičního, či zajímavého řešení.

Popsáno bude řešení vztahu mezi hráčem, zbraní a projektilem, práce se *shadery* (kolik jich je použito, jak se předávají mezi instancemi. . . ), nakonec *Skeleton animation*.

### 4.2.1 *Player*, *Weapon* a *Projectil*

Při implementaci této části došlo k problému jak tyto tři třídy provázat, varianta A nebo B viz obrázek 4.2. Varianta A umožňuje řídit vše přes hráče, tedy od nejstálejší třídy (hráč se nebude pravděpodobně měnit nikdy, zatímco zbraň se občas změnit může a projektil se bude měnit po každém výstřelu). Tato varianta je příjemná pro řízení výstřelu a nabití, ovšem pro vykreslování má jednu nepříjemnou vlastnost, a to, že se musí začít od hráče a ten musí aktualizovat polohu zbraně a ta projektilu, takže je-li pořadí vykreslování opačné, mají zbraň i projektil vždy polohu o snímek zpět. Ve variantě B si o aktualizaci polohy vždy nižší prvek může říct a tím pádem nezáleží na pořadí. Ovšem při této variantě by se musel uchovávat místo hráče projektil, který se po každém výstřelu mění, také by bylo celkem komplikované vystřelit a znovu nabít, nemluvě o případě, že dojdou šípy.

Ideální varianta by byla kruhová závislost ovšem C++ tuto variantu není schopno zkompilovat. Ale pokud se v jednom směru použije konkrétní třída (*Player, Weapon, Projectil*) a ve druhém směru rodičovská třída *Model*, tak tuto obousměrnou závislost lze provést. A jeden směr tak bude sloužit k výstřelu, zatímco druhý bude sloužit k získání pozice.




Obrázek 4.2: Varianty provázání *Player*, *Weapon* a *Projectil*. Šipky naznačují, které třída bud obsahovat ukazatel na druhou třídu.

### 4.2.2 Shadery

*Shadery* jsou programy běžící na GPU a probíhají v nich vykreslovací výpočty pro jednotlivé vrcholy a fragmenty (*vertex* a *fragment shader*). Protože některé modely v jistých situacích vyžadují jiné výpočty, než jindy obsahuje aplikace hned několik sad *shaderů* (např.: pro vykreslování do *shadow mapy*, bežné vykreslení scény, *skybox*, ...). Všechny jsou k nalezení na přiloženém DVD ve složce /src/shaders.

Možností je pro každý model načíst potřebné *shadery*, zkompilovat je a používat, ale to by zabíralo hodně paměti a zpomalovalo aplikaci. A většina objektů potřebuje ty stejné *shadery*, téměř všichni potomci třídy *Model* používají *shadowMap.v/fs* a *diffuseLight.v/fs*. Tento problém jsem vyřešil tak, že tyto dva *shadery* načte a uchovává třída *Scene* a modelům předává jen jejich adresu. Když následně bylo potřeba přidat *shadery* pro prvky hud vznikl problém, pokud by se opět přidaly do třídy *Scene*, stala by se tato třída velice špatně čitelnou a vznikla by zde spousta podobných a zaměnitelných atributů. Proto pro všechny instance třídy *Hud* a jejich potomků je *shader* jako statický atribut. Takže první vzniklá instance jej načte a zkompiluje nebo se zavolá příslušná statická metoda, ostatní už jen přistupují k tomuto atributu.

Varianta se statickým atributem je pracovně mnohem příjemnější, protože model pro své vykreslení potřebuje relativně hodně proměných typu *uniform* jejiž adresu si je třeba najít a pamatovat. To se může udělat v rámci funkce, kdy model dostane adresu *shaderů* *setShader()*, ale tím vznikne několik míst, kde se napevno nastavuje konstanta (jméno proměnné v *shaderu*), což by při úpravách kódu mohlo způsobit nepříjemnosti. Na druhou stranu, předají-li se jí tyto adresy jako parametry, vznikne funkce asi s 9 parametry, což není vůbec praktické. V aplikaci je to pro zatím řešeno tak, že část se nastavuje v *setShader()* a část se předá parametrem. A tady přichází na scénu statické atributy. Pokud se totiž použije tato varianta, vše se může tím pádem nastavit v metodě *setShader()* a to pouze jednou a na jednom místě v kódu.

### 4.2.3 Implementace Skeletal Animation

Tato podkapitola rozebere způsob vyhledávání záznamů ve struktuře *aiAnimaton* a jejich následující zpracování. Nejdříve bude vysvětlena část probíhající na CPU, na GPU probíhá opravdu jednoduchý děj, jenž bude popsán na konci.

Kosti jsou uloženy ve vektoru mnou vytvořených struktur *Bone*, struktura vždy obsahuje matici odsazení a naposledy vypočítanou finální matici. Pro vykreslení objektu s animací v čase t potřebujeme znát finální matice kostí pro čas t. Na výpočet těchto matic je potřeba projít strom *aiNode* a pole *aiAnimation*, jenž jsou uloženy ve třídě *Animation*.

*Assimp* ukládá načtený soubor do struktury aiScen, jejíž příklad je vidět na obrázku 4.3. Vektor kostí z *aiMesh*, odkaz na vrchol stromu *aiNodes* z *aiScene* i odkaz na strukturu *aiAnimation* z *aiScene* jsou přeneseny do třídy Animation. K získání finálních matic pro animaci v čase t je potřeba projít strom uzlů. Pro každý uzel je potřeba vyčíst matici z pole struktur *aiNodeAnim* v *aiAnimation* a vynásobit s maticemi všech předků a předat kosti.

Rozhodl jsem se strom procházet rekurzivně a algoritmus vypadá zhruba takto (pseudokód):

Algoritmus 4.1: Algoritmus pro projití stromu uzlů

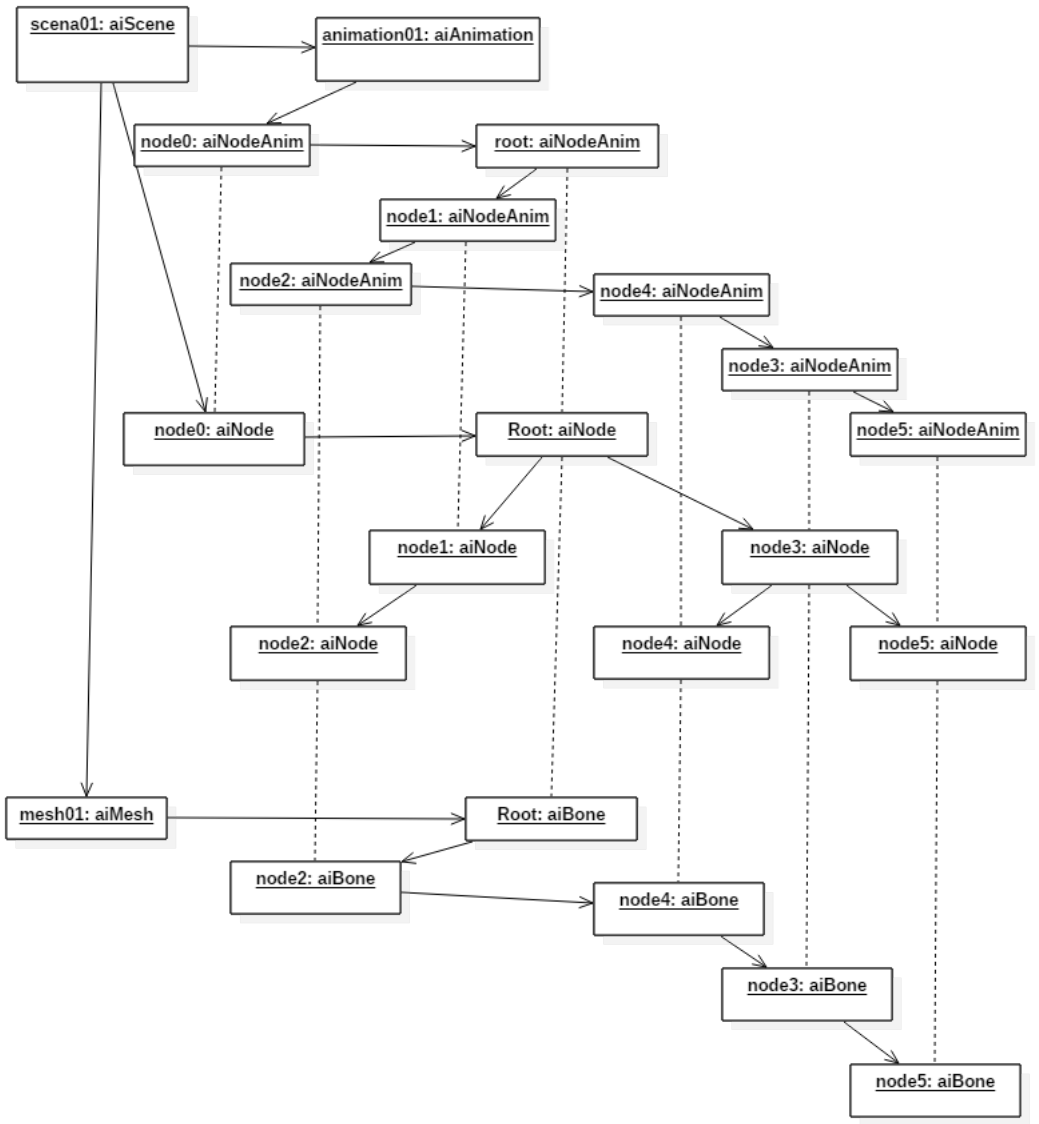
```
void updateNode(Node *n, Matrix4x4 m, time t){
        Matrix4x4 nodeM = getMatrixfromAiAnimation(n->name,t);
        Matrix4x4 toGlobal = inverse(m*nodeM);
        if(findBone(n->name)){
                Bone * bone = getBone(n->name);
                bone->final = toGloba * bone->offset;
        }
        for each(Node child in n->child){
                updateNode(child, toGlobal)
        }
}
```

Pro získání matice aktuálního uzlu je potřeba v poli *mChanels* v *aiAnimation* vyhledat záznam se stejným jménem jako má uzel. Jestliže byl nalezen správný *aiNodeAnim*, pak je potřeba z něj vyčíst vektor posunutí, kvaternion rotace a vektor zvětšení. Pro výpočet těchto prvků pro čas mezi klíčovými snímky se použije vzorec 2.3 vysvětlený v kapitole 2.3. Z těchto dvou vektorů a kvaternionu bude sestavena hledaná transformační matice.

Na GPU probíhá pouze násobení matic váhami a jejich sečtení. Tyto operace se provádí v rámci *vertex shaderů*. Výsledná matice je použita spolu s *model view projction* maticí a souřadnicemi bodu. Jak je vidět v následujícím úryvku kódu ze souboru *diffuseLight.vs*.

Algoritmus 4.2: Ukázka práce *skeletal animation* ve *vertex shaderu*.

```
void main(){
        if(boneWeight[0]>0){
                boneT      = gBones[int(boneID[0])] * boneWeight[0];
                boneT     += gBones[int(boneID[1])] * boneWeight[1];
                boneT     += gBones[int(boneID[2])] * boneWeight[2];
                boneT     += gBones[int(boneID[3])] * boneWeight[3];
        }
        vec4 p=boneT*vec4(vertexPosition_modelspace,1.0);
        gl_Position = MVP*vec4(p.xyz,1.0);
        uv=vertexUV;
        normal=(M * boneT *vec4(vertexNormal_modelspace,0.0)).xyz;
```




Obrázek 4.3: Graf konkrétní instance struktur *aiScene*. Jsou vidět všechny tři podstruktury, ve kterých jsou uložena data potřebné k realizaci animace. Přerušované čáry spojují ekvivalentní objekty.

# Kapitola 5

# Shodnocení náročnosti a efektivnosti implementace

První část této kapitoly se bude zabývat schopnostmi a výkonem aplikace, dále její chování v určitých situacích, jako je načítání nového objektu či jiná nečekaná událost, třeba jeho nenalezení. Při nepříznivé reakci budou navrhnuty možné příčiny a následné opravy.

Druhá část se bude věnovat možnými rozšířeními aplikace pro lepší výkon a vzhled. Půjde spíše o myšlenky a návrhy než rozbor konkrétní implementace nebo teorie.

## 5.1 Náročnost a výkon

Tato podkapitola se bude zabývat jak se mění náročnost na paměť a počet snímků za sekundu během a po:

- vložení mnoha stejných modelů
- vložení hodně různých modelů
- vložení několika středních modelů

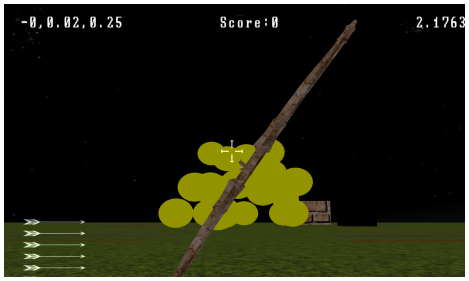
### 5.1.1 Vložení mnohokrát jednoho modelu

Pokud se opakovaně vloží jeden a ten samý model několikrát, mohlo by nastat dočasné zpomalení aplikace a zároveň dojde ke zvýšení zabrané paměti při vložení toho prvního, ale při každém dalším by se již neměla zvednout paměť ani klesnout framerate.

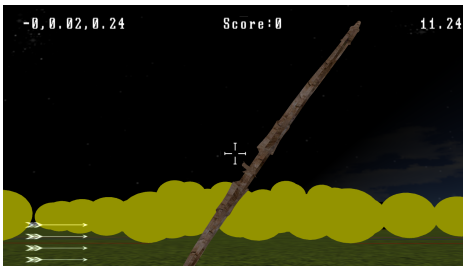
Tuto vlastnost otestuji se dvěma modely. V každém snímku vloží novou instanci objektu *sphere.obj* nebo *cube.obj* až do celkového počtu 100. U krychlí by se předpoklad měl vyplnit určitě, ale koule jsou již relativně složitý útvar (myšleno počtem vrcholů).

Výsledek pro krychle je:

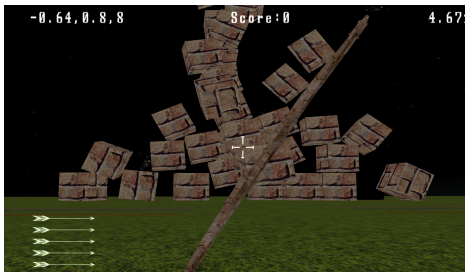
1. K navýšení paměti opravdu nedošlo. Tedy optimalizace s třídou *ImportModel* funguje jak mám.
2. Ovšem u počtu snímků za sekundu došlo k výraznému poklesu až do doby kdy se pozice krychlí neustálili. Důvodem tohoto jevu je dle mého názoru nedostatečný výkon mého CPU, pro vyhodnocení tolika kolizí v tak krátkém čase. K Snížení toho poklesu by mělo pomoct přesun vyhodnocení kolizí do samostatného vlákna.



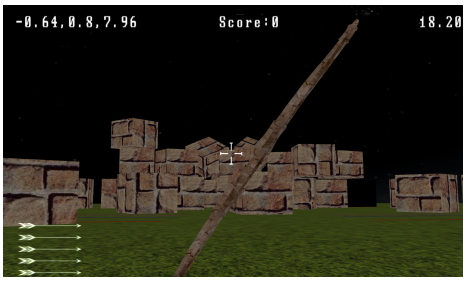

Obrázek 5.1: Během vkládání koulí




Obrázek 5.2: Po ustálení koulí




Obrázek 5.3: Během vkládání krychlý




Obrázek 5.4: Po ustálení krychlý

Výsledek pro koule je: Velice podobný jako u krychlí jen díky výrazně vyšší pohyblivosti koule byl pokles fps ještě silnější až pod jeden snímek za sekundu. A k ustálení v podstatě vůbec nedošlo.

### 5.1.2 vložení velkého množství malých modelů

V tomto případě by teoreticky mělo dojít k poklesu framrate i zvýšení zabrané paměti neboť každý přidaný objekt musí být alokován. K tomuto účelu jsem vytvořil ve složce se soubory .obj, speciální složku se sto krychlemi.

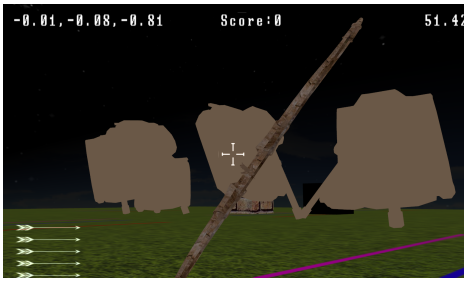
Výsledek byl neočekávaný, chování se v podstatě neliší od případu, kdy to byl pouze jeden stejný model, paměť nijak neroste a pokles počtu snímků je v podstatě stejný. Proto bude ještě vyzkoušeno i namapovat na každý objekt jinou texturu.

Při této situaci již došlo k nárůstu zabrané paměti, cca o velikost přidaných textur. Aplikaci z testů vyšla celkem špatně, pameťi není sice alokováno moc, ale jak je vidět na snímcích, tak pokles frameratu je hluboce pod přijatelnou úroveň.

## 5.2 Vložení několikrát středního modelu

Na závěr bude testováno vložení modelu tanku s přibližně 1000 vrcholy. Test se zaměří na dobu vložení a framerate.

Doba vložení tohoto modelu se pohybuje kolem 15s, ovšem tento interval se nezkracuje ani pro další instance tohoto modelu. Z toho vyplývá, že většinu doby se zpracovávají vrcholi do *btCollisionShape*, který se pro každou instanci tvoří nový. Na obrázku je sice vidět framrate 51, ale to je proto, že aplikace zamrzla.




Obrázek 5.5: Během vkládání tanků

## 5.3 Možnosti jak rozšířit aplikaci

- Vhodným rozšířením by bylo omezit oblast vykreslování a škálování úrovně detailů objektů a textur.
- Zajímavá, ale velice komplikovaná na provedení a koordinaci aplikace, je myšlenka rozdělit aplikaci do několika vláken, tak aby ošetření kolizí běželo v samostatném vlákně. Ovšem pro obsáhlejší hru naprosto nutné. *Bullet* by k tomuto měl poskytovat potřebné funkce v rámci článku *Collision Callbacks and Triggers*[3] na jejich stánkách.
- Rozšíření třídy *Animation* na více animací, tedy místo atributu *aiAnimation** na *aiAnimation***. Dále úpravu vyhledávacích a výpočetních funkcí, nejkomplikovanější na této úpravě by pravděpodobně bylo zajistit průběh několika animací zároveň, tedy finální matice kostí by musely vznikat jako součin aktuální hodnoty a hodnoty vypočítané metodou *updateNode()* a na začátku prvního výpočtu pro daný snímek nastavit matici na identitu.
- Dalším praktickým rozšířením by byla schopnost číst a tvořit dávkové soubory pro scénu. Tím by se dalo pracovat s výrazně komplikovanější scénou.

# Kapitola 6

# Závěr

Tato bakalářská práce stanovený cíl z velké části splnila. Vzniklo jednoduché demo, na kterém je možno se seznámit s několika základními grafickými metodami a prozkoumat cyklus hry. Ovšem pokud bychom jej chtěl rozšířit na plnohodnotnou hru, bylo by potřeba zapracovat většinu myšlenek z kapitoly 5.3, kde jsou jmenovány náměty k rozšíření. Jako velice důležitý krok pro další práci s aplikací vidím implementaci paralelizace pro zpracování kolizí, tak aby tento proces nezpomaloval celou aplikaci. Dále by bylo potřeba vylepšit algoritmus pro převod pole vrcholů do *btCollisionShape* nebo obecně tvorbu těchto struktur pro *Bullet*. Z tohoto nedostatku vzniká spousta nevhodného chování aplikace (například místo schodů vznikná šikmá plocha). Ale vzhledem k vlastním vstupním znalostem oboru počítačové grafiky, shledávám tuto práci jako úspěšnou, neboť otázka „Ale co vše je vlastně potřeba vytvořit, abychom se mohli projít ve virtuální 3D scéně?“ byla z velké části zodpovězena.

# Literatura


[1] Baş, C.; Shuralyov, D.; Gray, J.; aj.: *GLFW*. [Online; navštíveno 16.05.2017].
URL http://www.glfw.org/

[2] Coumans, E.: *Bullet physic library*. [Online; navštíveno 26.04.2017].
URL http://bulletphysics.org/wordpress/

[3] Coumans, E.: *Bullet physic library, článek o spouštích v případě kolizí*. [Online; navštíveno 26.04.2017].
URL http://bulletphysics.org/mediawiki-1.5.8/index.php/Collision_Callbacks_and_Triggers

[4] truck creation, G.; Riccio, C.: *OpenGL Mathematics*. [Online; navštíveno 16.05.2017].
URL http://glm.g-truc.net/0.9.8/index.html

[5] Eberly, D. H.: *3D Game Engine Design*. Morgan Kaufmann publishers, 2001, ISBN 1-55860-593-2.

[6] Gebhardt, N.; Ambiera: *IrrKlang, high level 3D audio engine*. [Online; navštíveno 16.05.2017].
URL http://www.ambiera.com/irrklang/

[7] Gessler, A.; Schulze, T.; Kulling, K.; aj.: *Open Asset Import library*. [Online; navštíveno 26.04.2017].
URL http://assimp.org/index.html

[8] Perlin, K.: *new york university mrl Ken Perlin*. [Online; navštíveno 16.05.2017].
URL http://mrl.nyu.edu/~perlin/

[9] Perlin, K.: *An Image Synthesizer*. 2017, [Online; navštíveno 13.05.2017].
URL https://pdfs.semanticscholar.org/c4e6/00dad6313f9d0b5469ff0aa2f6e9f179de93.pdf

[10] Perlin, K.: *Improving Noise*. 2017, [Online; navštíveno 13.05.2017].
URL http://mrl.nyu.edu/~perlin/paper445.pdf

[11] Žára, J.; Beneš, B.; Sochor, J.; aj.: *Moderní počítačová grafika*. Computer press, 2004, ISBN 80-251-0454-0.

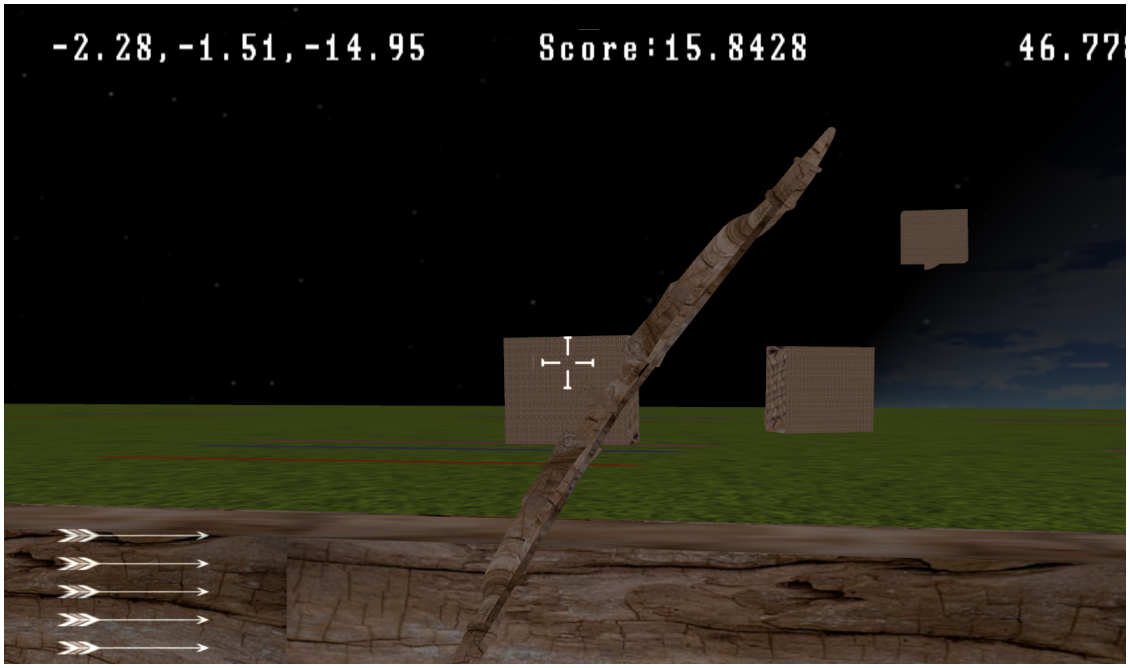
[12] Shreiner, D.; Woo, M.; Neider, J.; aj.: *OpenGL Průvodce programátora*. Computer press, 2006, ISBN 80-251-1275-6.

[13] Stewart, N.; Ikits, M.; Magallon, M.: *The OpenGL Extension Wrangler Library*. [Online; navštíveno 16.05.2017].
URL http://glew.sourceforge.net/

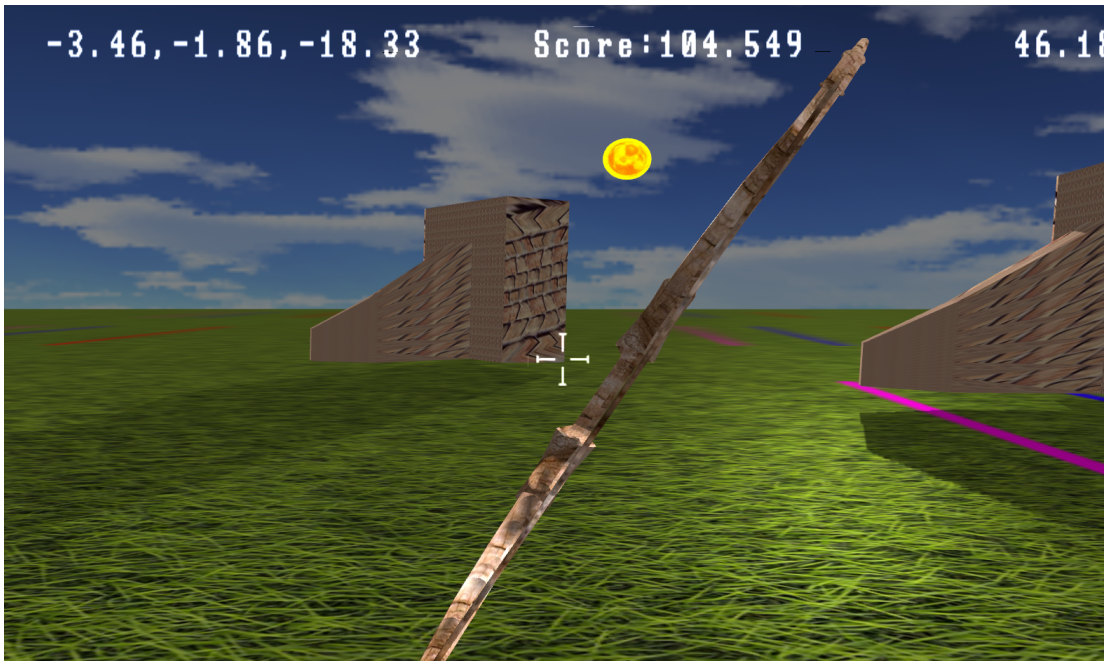
# Přílohy

# Příloha A

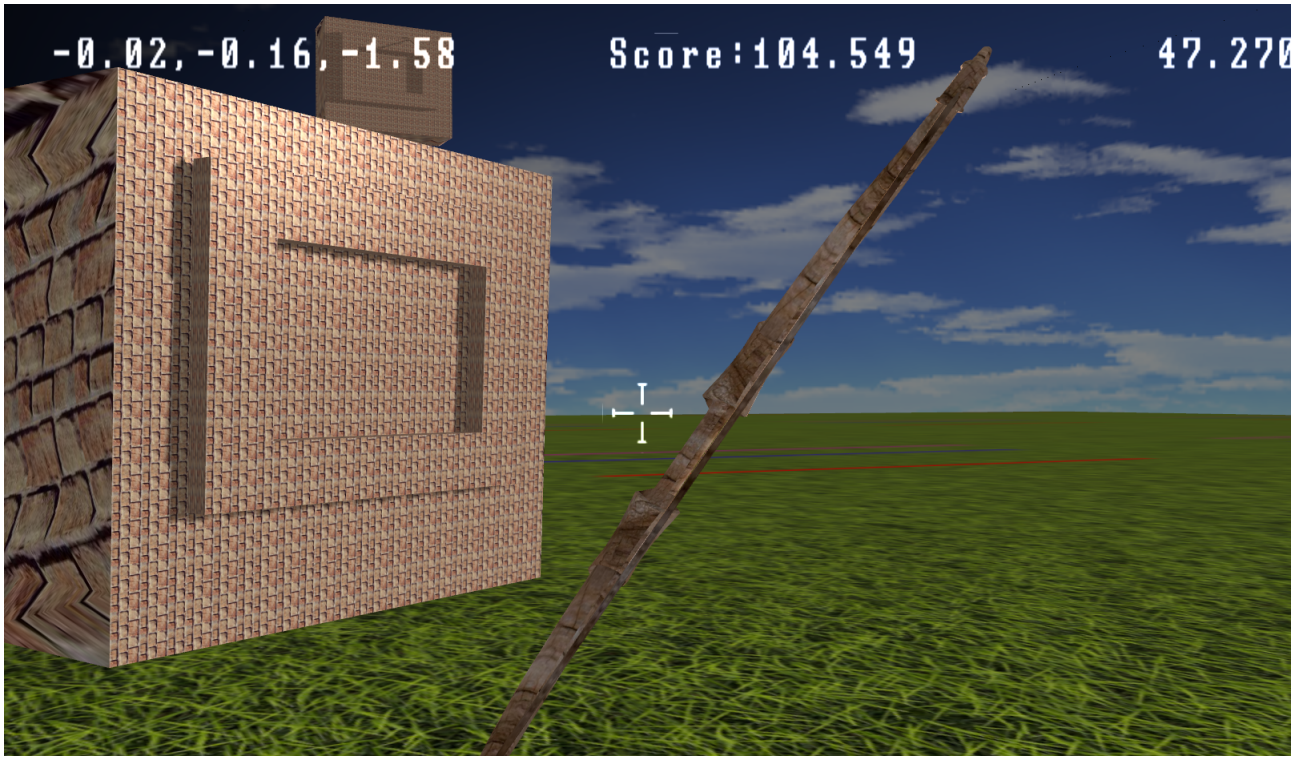
# Screenshots



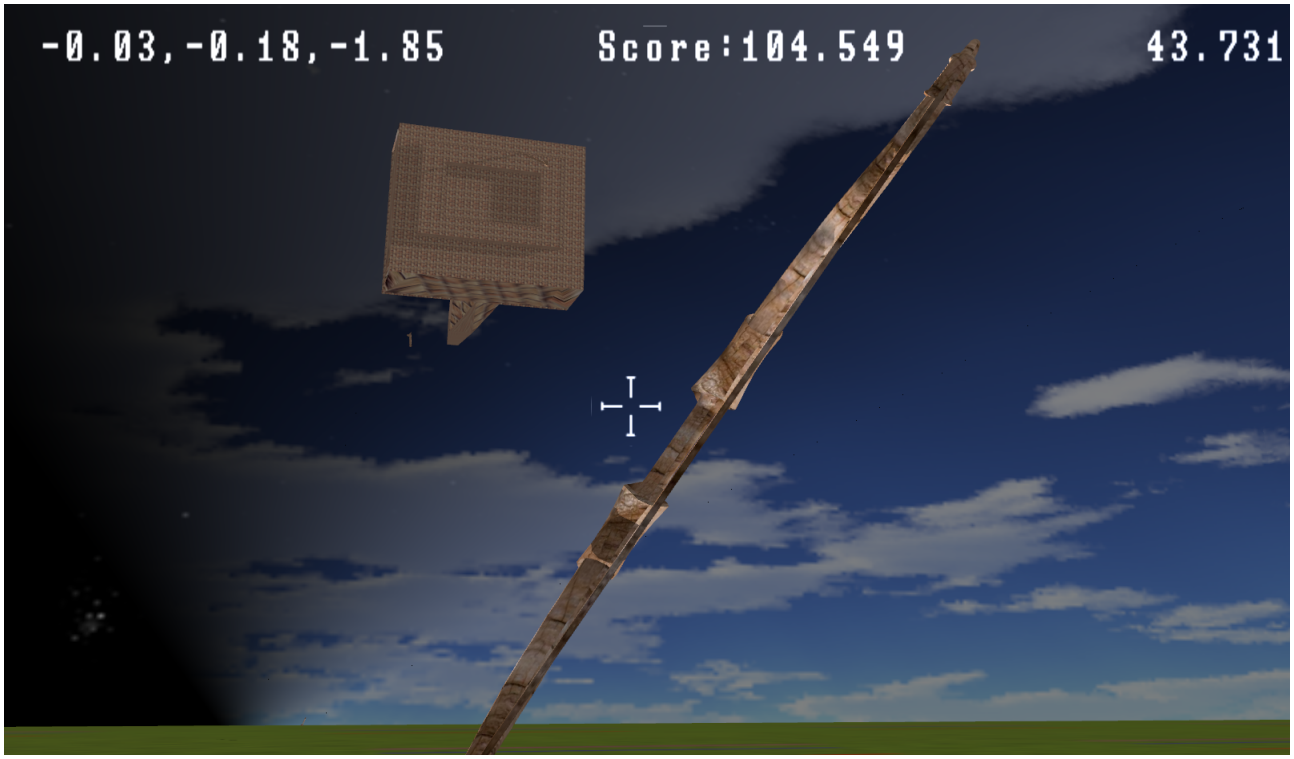

Obrázek A.1: Screenshot ze střelecké pozice s naplým lukem




Obrázek A.2: Screenshot během úsvitu, jsou zde vidět stíny vržené terči a odlesk od vycházejícího slunce




Obrázek A.3: Detail terče, na kterém jsou vidět vykreslené vlastní stíny




Obrázek A.4: Zde je vidět stín vržený zabodlým šípem v terči

# Příloha B

# Obsah CD

/ CD

- / 3rd_parties -Obsah je z převážné části tvorbou třetích stran
  - / authors.txt -zde jsou uvedeny zdroje a autoři souborů.
  - / sounds -složka obsahuje zvuky pro aplikaci
  - / libs -externí knihovny
  - / icons -modely a textury
- / doc -zdrojové soubory technické zprávy
- / src -soubory aplikace
  - / InteractiveOpenGLDemo -zdrojové kódy výsledné aplikace a spouštěcí soubor
- / technická zpráva.pdf