Vysoká škola báňská – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra telekomunikační techniky

Možnosti 3D enginu OGRE

Capabilities of OGRE 3D Engine

2013/14 **Jan Urubek**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jan Urubek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Možnosti 3D enginu OGRE
Capabilities of OGRE 3D Engine

Zásady pro vypracování:

V dnešní době existuje velká řada 3D enginů pro vývoj 3D aplikací. Jedním z velmi zajímavých 3D enginů v dnešní době je i OGRE (objektově orientovaný grafický renderovací engine). Cílem této práce je nastudování a popis základních možností ve spojení s modelovacím nástrojem Blender.

1. Nastudujte možnosti 3D enginu OGRE a fyzikálního enginu Bullet.
2. Zaměřte se na možnosti tohoto 3D enginu ve spojení s modelovacím nástrojem Blender, který umožňuje pohodlně vytvářet scény a modely, které lze následně využívat v OGRE.
3. Vytvořte ukázkovou aplikaci demonstrující tyto možnosti(např. stíny, průhlednost, fyzikální možnosti apod.).
4. Vše pečlivě zdokumentujte, aby bylo možné v práci pokračovat.

Seznam doporučené odborné literatury:

[1] http://www.ogre3d.org/
[2] F. Kerger. OGRE 3D 1.7 Beginner's Guide. 2010. ISBN: 978-1849512480

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Martin Němec, Ph.D.**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení studenta

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne: *4. května 2014*

.....
podpis studenta

Poděkování

Rád bych poděkoval svému vedoucímu práce, Ing. Martinu Němcovi za užitečné rady ohledně tvorby práce, za jeho podporu a pozitivní přístup při konzultacích.

Abstrakt

Tato práce je zaměřená na prozkoumání možností 3D renderovacího enginu OGRE. Za tímto účelem bude vytvořeno v jazyce C++ 10 ukázkových aplikací demonstrujících různé možnosti tohoto enginu a jejich uplatnění ve vývoji 3D aplikací a her. Součástí práce je demonstrační hra ve stylu tower defense. Pro vymodelování 3D prostředí bude rovněž aplikována možnost využití modelovacího nástroje Blender a fyzikálního enginu Bullet pro simulaci fyziky v reálném čase. Pro logickou stránku hry budou naimplementovány algoritmy uvažování nepřátel a vyhledávání cest (A* algoritmus).

Klíčová slova

OGRE 3D; C++; Blender; Bullet physics; A*; Vývoj her; Techniky implementace s použitím OGRE 3D;Tower defense;

Abstract

This works purpose is to test capabilities of 3D rendering engine OGRE. In order to test these capabilities, 10 testing aplications will be created in C++ language. Each of these aplications will demonstrate different techniques and implementation in 3D game development. The final result is going to be a strategy game. All models and the enviroment will be modeled in Blender and for the physical simulation in real time will be used Bullet engine. In case of logical structure of the game, algorithms for AI decisions and pathfinding (A star) will be also implemented.

The main goal of this work is to create a playable and user-modifiable game and subsequent improvement of the author in the development of applications and 3D graphics.

Key words

OGRE 3D; C++; Blender; Bullet physics; A*; Game development; Techniques used for implementation in OGRE 3D;Tower defense;

Seznam použitých termínů

Termín	Význam termínu
engine	Jádro počítačové hry nebo programu. V podstatě knihovna obsahující již vytvořené funkce.
renderování	Tvorba obrazu neboli vykreslování grafiky.
Direct3D	Rozhraní (API) pro práci s 3D grafikou. Komponenta sady knihoven DirectX.
OpenGL	Rozhraní (API) pro práci s grafikou.
A*	Algoritmus vyhledávání nejkratší cesty.
bump mapping	Technika texturování vytvářející iluzi nerovnosti povrchu.
reflection mapping	Technika vykreslování materiálu objektu vytvářející odraz okolí.
parallax mapping	Vylepšené techniky normálového mapování a bump mapování.
specular mapping	Technika úpravy odlesků materiálu objektu.
ambient occlusion	Metoda stínování modelů dodávající realistický dojem.
wrapper	Knihovna propojující funkce jedné knihovny s knihovnou druhou.
shader	Počítačový program sloužící k řízení jednotlivých částí GPU.
vertex	Bod v prostoru 3D grafiky. Základem všech modelů.
plane	Jednoduchý model stěny.
low-poly a high-poly	Low-poly je model obsahující nízký počet polygonů (stěn), high-poly jich naopak obsahuje mnohokrát více.
vertex program	Součást shaderu. Program, který se provede na každém vrcholu (vertexu) vstupní geometrie.
fragment program	Součást shaderu. Program, který se provede na každém pixelu scény.
SceneNode	Třída v OGRE enginu reprezentující bod v prostoru, ke kterému se mohou přichytit ostatní objekty a následně je s nimi možno pohybovat, zvětšovat je či rotovat.
Bounding-box	3D kvádr či krychle obalující model objektu.
parsování	Proces procházení textových symbolů následováno prací s hodnotami těchto textů.

iterace	Opakování určitého procesu (například cyklus while).
FPS	Počet snímků za vteřinu (z angl. Frame per second).
mass	Jednotka hmotnosti určitého fyzikálního objektu.
character controller	Sada metod většinou spjatá s ovládáním postavy.

Obsah

Úvod

Počítačová grafika a její vývoj se stávají stále více rozšířenější, jak v oblasti vytváření modelů, animací, technického modelování, tak v oblasti počítačových her. Proto vzniká mnoho renderovacích enginů, komerčních i nekomerčních.

Cílem této práce je seznámení s renderovacím enginem OGRE se spojením s modelovacím nástrojem Blender a Bullet fyzikou, prozkoumání možností tohoto enginu s kladeným důrazem na praktické otestování a implementaci v aplikacích. Práce je završena vytvořením hratelné a uživatelsky modifikovatelné hry.

Tento text je rozdělen do dvou částí. První část obsahuje stručný popis enginu včetně praktických ukázek vykonaných pokusů. Druhá část je zaměřena na popis technik a dalších možností OGRE použitých při tvorbě hry.

1 Renderovací enginy

V současné době existuje mnoho renderovacích enginů. Renderovací enginy výrazně usnadňují práci vývojářům, kteří mohou využít již naimplementované metody pro práci s grafikou a nemusí implementovat metody vlastní. Patří mezi ně např. Unreal engine, Unity3D nebo OGRE 3D.

Tyto enginy se mezi sebou liší jednak cenou (některé jsou bezplatné, některé ne), tak množstvím funkcí, které poskytují vývojářům.

1.1 Unreal engine

Unreal engine byl vyvinut roku 1998 společností Epic Games, původně pro jejich hru s názvem Unreal. Jádro tohoto enginu je napsáno v jazyce C++ a nabízí kromě funkcí vykreslování a práce s 3D grafikou, taky funkce zaměřené na vývoj her (přehrávání zvuků, character controller, atd…). Je multiplatformní a v současné době podporuje operační systémy Windows, Linux, Mac OS, konzole Xbox a PlayStation a dokonce i operační systémy pro chytré telefony jako Android či iOS.

Nejnovější verzí je Unreal Engine 4, jejíž vývoj začal již v roce 2003 a byl představen až v roce 2012. Nyní je dostupný za poplatek ve výši 19 dolarů měsíčně a pěti procent z výnosu.

Tento engine je velice oblíbený díky své struktuře a flexibilitě nejen mezi profesionály, ale také mezi příležitostnými vývojáři. Bylo v něm vytvořeno mnoho světově známých her jako např. Duke Nukem Forever, Medal of honor, Unreal tournament, Bioshock, a další…

1.2 Unity 3D

Unity 3D je multiplatformní herní engine vyvinut společností Unity technologies. Jádro tohoto enginu je napsáno v C++, ale rovněž využívá jazyku C#. V roce 2005 byl rozšířen z původní podpory OS X na multiplatformní engine. Vyniká intuitivním vývojovým prostředím a využívá Direct3D,OpenGL a OpenGL ES. Podporuje bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), dynamické stíny a mnoho dalších technik.

Unity má rovněž zabudovanou podporu fyzikálního enginu PhysX od firmy NVIDIA.

Kompletní verze enginu i s vývojovým prostředím stojí 1500 dolarů jednorázově nebo 75 dolarový poplatek každý měsíc. 30 denní zkušební verze je zdarma.

2 OGRE 3D engine

OGRE je objektově orientovaný grafický renderovací engine napsáný v C++. Engine je vytvořen tak, aby byl jednoduchý a intuitivní pro vývojáře aplikací využívající 3D akcelerace. Dokáže obsluhovat jak Direct3D tak OpenGL. OGRE je open-source. [1].

2.1 Historie

Název OGRE je zkratkou anglického Objected-oriented Graphics Rendering Engine (v překladu objektově orientovaný renderovací engine) a první verze 1.0.0 s označením Azathot byla vydána v roce 2005. Otcem tohoto projektu byl Steve Streeting, pro komunitu znám svou přezdívkou sinbad.

V současné době je OGRE ve verzi 1.9.0 označovaného jako Ghadamon a má velkou komunitu uživatelů, kteří se aktivně zapojují do vylepšování tohoto enginu a sdílejí mnoho návodů pro začínající vývojáře. Bohužel většina původních vývojářů včetně zakladatele Steva Streetinga tým OGRE opustili, avšak na druhou stranu přibylo mnoho nových vývojářů.

2.2 Softwarové požadavky

OGRE je multiplatformní engine a v současné době podporuje operační systémy : Windows, Linux, OS X, NaCI, WinRT, Windows Phone 8, iOS, a Android.

Celé jádro enginu je napsáno v C++, ale rovněž bylo komunitou rozšířeno o wrapper podporující .NET. Díky tomu může být OGRE použito s jazyky C#, Visual Basic, atd…

Minimální požadovaná verze DirectX je verze 9. Velikost RAM paměťi závisí na konfiguraci projektu (počet a velikost textur nejvíce ovlivňují RAM) a procesor by měl být kompatibilní s grafickou kartou, která by měla být aspoň na úrovni Geforce karet.

Jelikož tato bakalářská práce byla psána ve Visual Studiu 2012 a některé z projektů využívají audio knihovny openAL, je na některých počítačích nutné doinstalovat potřebné knihovny. Tyto potřebné knihovny se nacházejí na DVD.

2.3 Root a SceneManager

Nejdůležitějším prvkem v OGRE aplikaci, je objekt typu Root. Tento objekt musí být vytvořen jako první a je také posledním uvolněným při ukončení aplikace. Stará se o veškeré podtřídy tohoto enginu a propojuje je mezi sebou. Dále umožňuje načítání pluginů (.dll soubory), stará se o vytvoření hlavního okna, nastavuje renderovací subsystém (DirectX nebo OpenGL), a mnoho dalších důležitých funkcí.

SceneManager je druhým nejdůležitějším prvkem OGRE aplikace. Stará se o veškeré vytvoření entit, kamer, světel, částicových systémů, a dalších prvků obohacujících scénu. Rovněž spravuje dvě techniky vykreslování stínů objektů: stencil a texture. Přičemž technika

stencil je nativní technikou již naimplementovanou v jádru enginu, a technika texture je určena pro ty vývojáře, kteří mají zájem využít ve svém projektu vlastní shader pro vykreslování stínů.

Dále SceneManager podporuje vytvoření oblohy či pozadí scény. Pro tento účel lze využít jeden ze tří typů : skybox, skydome nebo skyplane.

2.4 Objekty

OGRE obsahuje druhy objektů, se kterými se dá pohybovat, zvětšovat či rotovat. Každý z těchto druhů dědí z rodičovské třídy MovableObject. Rovněž mohou být přichyceny na objekt typu SceneNode. Mezi tři nejpoužívanější patří: entity, světla a kamery.

2.4.1 Entity

Entity jsou nejpoužívanějšími druhy pohyblivých objektů. Je to objekt vytvořený třídou SceneManager pomocí metody createEntity. Entita sama o sobě nemůže existovat bez nadřazeného objektu SceneNode ke kterému musí být přichycena.

Základní stavebním kámenem entity je mesh. Jedná se o soubor vertexů, hran a stěn vytvarovaných do tvaru určitého objektu. OGRE využívá soubory .mesh, které obsahují všechny potřebné informace o modelu, jako rozmístění vertexů, stěn, normálů, atd… S použitím Blenderu a addonu blender2ogre mohou být do tohoto formátu modely vyexportovány.

2.4.2 Světla

OGRE podporuje 3 základní druhy světel.

- Bodové světlo (point) - pracuje tak, že zdroj emituje světlo do všech směrů, v dané vzdálenosti a útlumu.
- Reflektor (spotlight) - pracuje tak, že zdroj emituje světlo v podobě kuželu o daném vnitřním úlhlu, vnějším úhlu, vzdálenosti a útlumu.
- Směrové světlo (directional) - pracuje tak, že zdroj emituje světlo jedním směrem a osvítí tak stejnou silou všechny objekty v tomto směru.

Světlům lze nastavit intezita svítivosti, dosah, difůzní barva a spekulární barva.

2.4.3 Kamery

Kamera reprezentuje bod, ze kterého OGRE renderuje scénu. Tato scéna je vyrenderována do bufferu. Normálně je druhem bufferu hlavní okno, ale rovněž jím může být textura.

Kamery rovněž podporují dva druhy projekce: perspektivní (klasické zobrazení - objekty se zmenšují, když se od nich kamera vzdaluje) a ortografický (objekty se nezmenšuj se vzdáleností).

2.5 Částice

OGRE podporuje rovněž částicové systémy s různými variacemi možností vzhledů a chování. Tak jako u materiálů, je možno nadefinovat vzhled částicového systému do skriptu s koncovkou .particle. Zde je okomentovaná ukázka skriptu vytvořeného částicového systému :

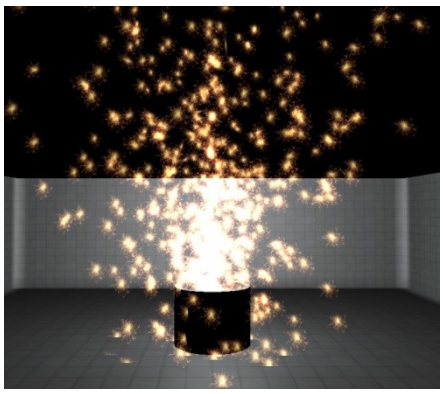
```
particle_system jiskry
{
material jiskry_mat // reference na material
particle_width 5 // šířka částice
particle_height 5 // výška částice
quota 10000 // maximalní počet částic v daný okamžik
billboard_type oriented_self // orientace dané částice


// Emmiter – místo, ze kterého se tvoří částice
emitter Point
{
angle 20 // částice bude mít náhodný úhel vystřelení ( v rozmezí
0° – 20°)
emission_rate 80 // 80 částic za sekundu
time_to_live 3 // jak dlouho bude částice existovat
direction 0 1 0 // vektor směru vystřelení
velocity_min 170 // minimální výška kam se částice vystřelí
velocity_max 200 // maximální výška kam se částice vystřelí
}
// Affectory – věci, které částici ovlivňují po dobu její
existence
affector LinearForce // Gravitace
{
force_vector 0 -200 0 // gravitační vektor (Y = -200 částice
padá dolů)
force_application add // síla gravitace se bude zvyšovat
}
affector ColourFader // Vyblednutí barev
{
red -0.25
green -0.25
blue -0.25
}
}
```

Jak lze vidět, skript musí obsahovat definici vzhledu jedné částice (referenci na její materiál, velikost a typ billboardu).

Dále musí obsahovat jeden nebo více emitorů, což jsou místa, ze kterých budou částice generovány. OGRE v současné době podporuje 6 druhů emitorů ’Point’, ’Box’, ’Cylinder’, ’Ellipsoid’, ’HollowEllipsoid’ a ’Ring’.

Poslední nepovinnou částí jsou tzv. Affectory, což jsou různé efekty, které ovlivňují částici v průběhu její existence. Affector může být například gravitační síla, nebo postupné zvětšování částice (scaler).

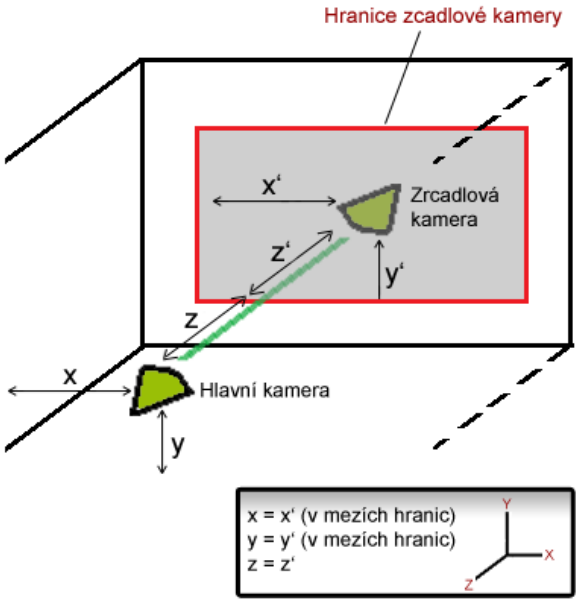


Obrázek 1.1: *Výsledný částicový systém*

2.6 Renderování do textury

Jedním z vykonaných pokusů bylo vytvořit zrcadlo, které by v reálném čase odráželo scénu, ve které se nachází. Pro tento účel byla využita třída Ogre::RenderTargetListener, která dokáže zachytit snímání kamery a vyrenderovat jej do textury.

Aby byl vytvořen efekt odrazu zrcadla, bylo nutné vytvořit druhou kameru, která zrcadlí pohyb hlavní kamery. Tento pohyb je však omezen v X a Y dimenzích na hranice objektu zrcadla (plane). V Z souřadnici není pohyb nijak omezen, jelikož indikuje přibližování a oddalování hlavní kamery od plochy zrcadla.

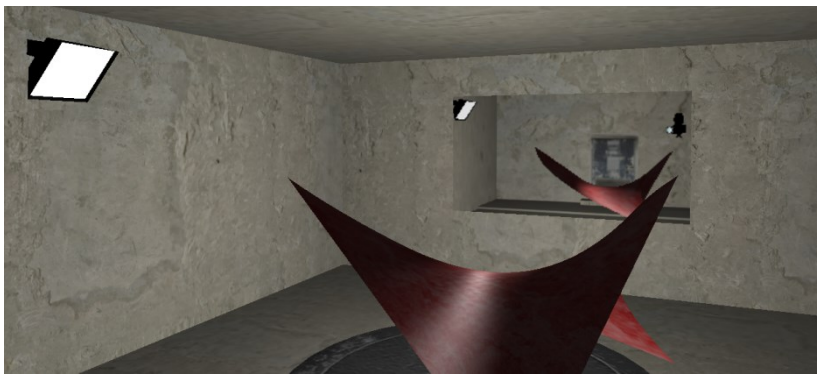


Obrázek 1.2: *Princip odrazu zrcadla*

Dále bylo nutné zaimplementovat výše zmíněnou třídu RenderTargetListener do aplikace. Proto z ní musela hlavní třída dědit. Poté byla vytvořena dynamická textura velikosti 1024x1024, která slouží jako snímač pohledu zrcadlové kamery.

```
TexturePtr tex =
TextureManager::getSingleton().createManual("dyncubemap",
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
TEX_TYPE_CUBE_MAP, 1024, 1024, 0,Ogre::PixelFormat::PF_R8G8B8,
TU_RENDERTARGET);
```

Textura je mapována jako cube_map, tudíž pro každou stěnu krychle je v reálném čase přiřazen jiný úhel pohledu zrcadlové kamery za pomocí šesti prvkového pole RenderTarget. Samotné přiřazování pohledů je prováděno v metodě preRenderTargetUpdate, která spadá pod RenderTargetListener.

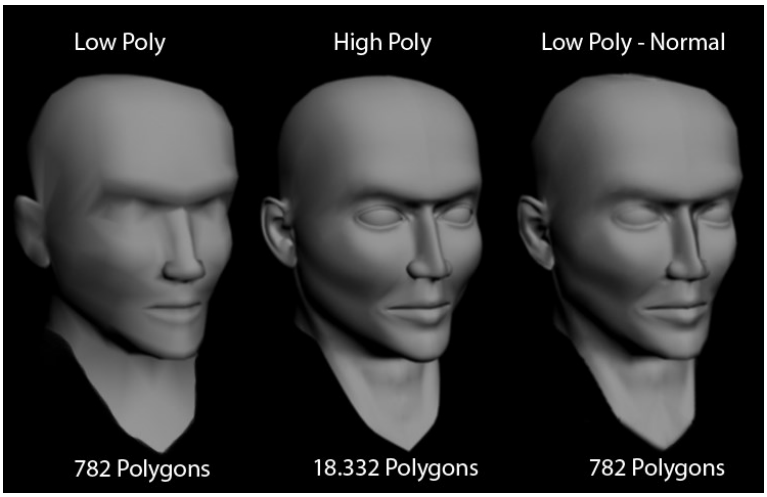


Obrázek 1.3: *Výsledný odraz v zrcadle*

2.7 Normal mapping

Využití normálového mapování v oboru herní grafiky je jednou z velice užitečných věcí. Normálové mapování navazuje pocit nerovností povrchu aplikované na nízko polygonálním (low-poly) modelu. Skutečnost, že je použit low-poly model rapidně vylepšuje FPS.

Technik vytvoření normálové mapy je více, ale nejčastěji používaná je technika vytvoření vysoko-polygonálního modelu (high-poly) a následné duplikování tohoto objektu a zredukování jeho polygonů. Poté jsou tyto dva modely porovnány a z jejich rozdílu je vytvořena normálová mapa. Tato mapa je následně aplikována na low-poly model a díky tomu simuluje detailnost high-poly modelu.



Obrázek 1.4: *Porovnání low-poly,high-poly a low-poly s normal mappingem[3]*

OGRE sám o sobě normálové mapování nepodporuje a proto je za potřebí použít shader. Reference na shader se uvádí do souboru s příponou .program, ve kterém se uvede název vertex programu a fragment programu spolu se názvem souboru. Rovněž jsou zde nadefinovány standartní vstupní parametry, které shader používá, jako například WorldViewProjection matice.

Posledním krokem je vytvořit referenci na dané vertex a fragment programy přímo v materiálovém skriptu určitého modelu.

```
vertex_program_ref Examples/BumpMapVP{
param_named_auto lightPosition light_position_object_space 0
param_named_auto worldViewProj worldviewproj_matrix
}
fragment_program_ref Examples/BumpMapFP{
param_named_auto lightDiffuse light_diffuse_colour 0
}
```

Stejným způsobem je možno využít další z technik jako specular mapování, ambient occlusion apod…

Obrázek 1.5: *Normal a specular mapping v OGRE*

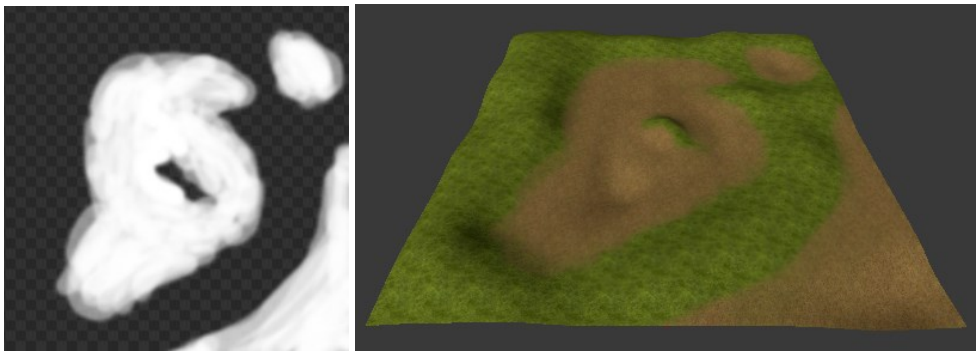
2.8 Alpha splatting

Alpha splatting je technika používaná ve většině případů na povrchy terénů. Jelikož terény nabývají velkých rozměrů, bylo by použití jedné textury na celou plochu nevhodné. Jednak by tato textura nepůsobila moc dobrým dojmem (byla by roztažená po celé ploše) a v případě velkého rozlišení této textury, by zabírala velké místo v paměti.

Alpha splatting pracuje na principu několika vrstev textur vzájemně se překrývajících a mapovaných podle určitých alfa map.

Pro vytvoření jednoduchého terénu s funkcí alpha splattingu, byl použit Blender. Nejprve byl vytvořen model samotného terénu a následně mu byly přiřazeny dvě UV mapy. První UV mapa mapuje veškeré textury aplikované na terénu - proto je nastavená tak, aby se textura vhodně opakovala. Druhá UV mapa je potřebná pro texturu alfa mapy a její mapování je nastaveno tak, aby textura pokryla veškerý povrch terénu.

Následně bylo nutné vytvořit první vrstvu textur, což je v podstatě vrstva aplikovaná na celé ploše terénu (např. textura trávy). Další vrstva reprezentuje danou alfa mapu a je nutné, aby měla v záložce Influence zatrhnutou vlastnost Stencil. Tímto Blender zajistí, aby veškeré další textury byly zobrazeny v těch místech, kde se na obrázku alfa mapy vyskytuje jakákoliv barva. Posledním krokem bylo vložit další vrstvu se stejným mapováním jako vrstva první (např. textura hlíny)



Obrázek 1.6: *Alfa mapa (nalevo), výsledek (napravo)*

Tímto způsobem byl vytvořen jednoduchý model terénu a už jen zbývalo vytvořit materiálový skript, který by umožňoval alpha splatting v OGRE aplikaci.

Jednotlivé vrstvy byly vytvořeny za pomoci passů. Zde je příklad definice jedné vrstvy:

```
pass {
      ambient 1.0 1.0 1.0
      diffuse 1.0 1.0 1.0
      lighting off
      scene_blend alpha_blend

      texture_unit {
            // Textura
            texture hlina.jpg 2d unlimited alpha
             tex_coord_set 0   // UV map 0
             }

      texture_unit {
            // Alfa mapa
            texture stencil_hlina.png 2d unlimited alpha
             tex_coord_set 1
            colour_op_ex blend_diffuse_alpha  src_current
src_texture

             }

       }
```

2.9 3D Audio

Jelikož je OGRE pouhý renderovací engine, nepodporuje sám o sobě přehrávání zvuků. Za tímto účelem vznikl wrapper nazvaný OgreOggSound. Jeho funkce je prostá, spojit funkcionalitu audio knihovny OpenAL a OGRE enginu. [2]

Samotná implementace audia v aplikaci je založena na instanci třídy OgreOggSoundManager, která vytváří a aktualizuje všechny zvuky. Inicializace této instance musela být provedena metodou init() a následně byl vytvořen nový SceneNode s názvem poslechovyNode. Tento SceneNode slouží jako bod v 3D prostoru kde se bude nacházet posluchač - tedy hlavní kamera. Proto k němu musel být přichycen Listener z instance třídy SoundManager tímto příkazem:

```
poslechovyNode->attachObject(mSoundManager->getListener());
```

Samozřejmě aby vznikl dojem z 3D zvuku (stereo a zesilování či zeslabování v závislosti na vzdálenosti kamery), musela být při každé průběhu metody frameStarted provedena aktualizace poslechovehoNode na pozici kamery a její rotaci a samotná aktualizace SoundManageru, čili tato sekvence příkazů:

```
poslechovyNode->setPosition(kameraNode->getPosition());
poslechovyNode->setOrientation(kameraNode->getOrientation());
mSoundManager->update(evt.timeSinceLastFrame);
```

Nyní už byla aplikace schopná rozeznávat zvuky v 3D prostoru, tak už jen zbývalo vytvořit nový zvuk. K tomuto účelu využívá OgreOggSound třídu OgreOggISound. Vytvoření jedné instance této třídy je prováděno za pomocí SoundManageru a jeho metody createSound, ve které se specifikuje unikátní identifikační název zvuku, název souboru se zvukem a booleanovy proměnné indikující loop (neustálé přehrávání), streamování zvuku a zda by měl být zdroj přichycen během inicializace. Jakmile je zvuk vytvořen, zbýva pouze zavolat metoda play().

Třída OgreOggSound rovněž dědí ze třídy MovableObject tak jako entity či světla. To znamená, že objekt této třídy lze přichytit ke SceneNode - tedy vytvořit zvuk ve 3D prostoru a dokonce s ním pohybovat za běhu aplikace.

Vytvoření jednoho zvuku může vypadat např. takto:

```
OgreOggISound* hudba = mSoundManager->createSound("Hudba", "The
Statler Brothers - Flowers on the Wall.ogg", false, true, true);
radioSceneNode->attachObject(hudba);
hudba->play();
```

Jelikož chybělo pouze dynamické zesilování či zeslabování zvuku na základě vzdálenosti kamery od zdroje, byl naimplementován jednoduchý výpočet této hlasitosti:

```
// Zjisteni vzdálenosti mezi kamerou a zdrojem zvuku (radio)
Ogre::Real vzdalenost =
kameraNode->getPosition().distance(radioSceneNode-
>getPosition());

// vypočítání hlasitosti
float novaHlasitost = 1.0f - (vzdalenost*0.008f);
if(novaHlasitost > 1) novaHlasitost = 1;
else if(novaHlasitost < 0) novaHlasitost = 0;
hudba->setVolume(novaHlasitost);
```

2.10 OGRE Ray a RaySceneQuery

Pro interakci s entitami, jejich označování apod. OGRE využívá třídu Ray.

Jedná se o paprsek vytvořený z určité pozice s nastaveným směrem. Tento paprsek projde veškerými bounding-boxy entit, které kolidují s jeho dráhou. Tyto entity si zaznamená, spolu s jejich vzdálenosti od bodu kde paprsek vznikl.

```
// vytvoření Ray

Ray mRay = mCamera->getCameraToViewportRay((width/2)
/float(width), (height/2) /float(height));
mRayQuery->setRay(mRay);   // přiřazení mRay do mRayQuery (pole
výsledků)

// vytvoření RaySceneQueryResult, který uchovává informace o
všech objektech zasažených Rayem

RaySceneQueryResult &result = mRayQuery->execute();

// vytvoření iterátoru

RaySceneQueryResult::iterator it = result.begin();
RaySceneQueryResult::iterator itEnd = result.end();
```

Nakonec je možné iterovat všechny výsledky a provést s nimi potřebné operace (např. označování objektů).

3 Hra

3.1 Úvod ke hře

V poslední fázi bakalářské práce vytvořena strategická hra typu tower defense.

Aby byla hratelná a nesloužila pouze k demonstraci, byla do ní navíc zaimplementována fyzika (Bullet) a další herní algoritmy (např. A* navigace nepřátel). Tvorba této hry zabrala zhruba 2-3 měsíce a byl kladen důraz na její uživatelskou modifikovatelnost (tvorba vlastních map, XML struktura prvků hry, atd..).

3.2 Scéna

3.2.1 Struktura scény

Veškeré scény byly vymodelovány v Blenderu. Aby takto vymodelovaná scéna mohla být importována do samotné hry, bylo nutné si předem nadefinovat její strukturu.

Hlavním modelem, na kterém se bude vykonávat veškeré dění je model s názvem teren. Aby byl korektně rozrastrován pomocí editoru, je nutné, aby byl v půdorysu čtvercového popřípadě obdelníkového tvaru.

Dalším nutným objektem je objekt rastru. Je to v podstatě objekt totožného tvaru, jako je terén, avšak s průhlednou texturou. Účel tohoto objektu je zobrazování kolizí, popřípadě jednotlivých políček rastru.

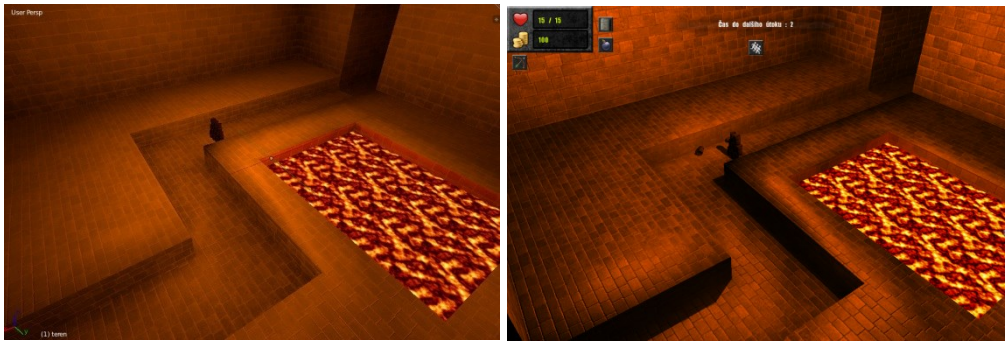
Všechny ostatní objekty slouží pouze k dekoraci scény.

3.2.2 Načtení scény v aplikaci

Nejprve bylo nutné vyexportovat celou scénu z Blenderu do formátů se kterými dokáže OGRE pracovat. K tomuto účelu byl použit addon blender2ogre. Tento užitečný addon dokáže vyexportovat všechny modely do .mesh formátu, animace do .skeleton souboru, materiály do .material souboru a nakonec informace o rozmístění, scale a rotaci do .scene souboru, což je v podstatě XML kód.

Dalším krokem bylo parsování výstupního .scene souboru. Za tímto účelem byla do projektu zaimplementována knihovna RapidXML parseru. Tento parser patří k mezi nejrychlejší parsery na světě a je zároveň nekomerční.

Postupným parsováním kódu byly vytvářeny veškeré objekty i světla ve scéně a jelikož je využita fyzikální knihovna Bullet, bylo rovněž nutné zkonvertovat statické i dynamické objekty scény na fyzikální modely (podrobněji v kapitole 3.10.2).



Obrázek 1.7: *Scéna v Blenderu (nalevo), načtená scéna ve hře (napravo)*

3.3 Editor

Pro rastrování terénu a následné určování kolizí bylo zapotřebí vytvořit editor.

Počet polí v rastru je uživatelsky nastavitelné a následné naplnění vektoru těchto polí bylo zrealizováno za pomocí OGRE třídy Ray. Algoritmus tohoto naplnění funguje tak, že nejprve jsou vypočítány rozměry jednoho políčka (šířka modelu terénu / počet políček). Poté proběhne iterace všech políček, řádek po řádku. Při jednotlivé iteraci je kamera napozicována vysoko nad pozici iterovaného políčka a za pomocí paprsku třídy Ray je určena Y pozice daného políčka. Tímto způsobem se zjistí výška terénu pro každé políčko.

Následně je již uživatel schopen označovat políčka pomocí myši a nastavovat kolize, či speciální vlastnosti. Samotné označování využívá opět třídu Ray, která vyšle paprsek tím směrem, kde se nachází myš a jakmile zasáhne terén, tak se pomocí vektoru pozice tohoto střetu najde políčko, které se na této pozici nachází.

Obrázek 1.8: *Ukázka rozrastrovaného terénu*

Výstupem editoru je soubor pol.tdx, který obsahuje informace o všech polích rastru (kolize,vektor pozice,rozměry a speciální vlastnosti).

3.4 Podpora jazyků

Za účelem jazykové univerzálnosti byla vytvořena XML struktura všech textů vyskytujících se ve hře.

Pro jednotlivé texty byla vytvořena struktura LString obsahující atributy id a text. Její nadřazenou třídou je struktura Jazyk, která obsahuje vektor těchto LStringů.

Rovněž je zaimplementován algoritmus, který vyhledá všechny tyto XML kódy, rozparsuje jejich texty, vytvoří objekt typu Jazyk s vektorem těchto textů a následně jej přidá do vektoru všech jazyků. Díky tomu mohou uživatelé přeložit celou hru do jakéhokoliv jazyka jen pouhým zkopírováním již nadefinovaného jazyka a přepsáním textů.

Všechny XML kódy jazyků byly umístěny do adresáře "res/strings".

```
<Language name="Čeština" icon="czech_icon">
      <!-- Menu -->
      <String id="newgame" text="Nová hra"/>
      <String id="settings" text="Nastavení"/>
      …
</Language>
```

3.5 GUI

Nedílnou součástí hry je GUI neboli grafické uživatelské rozhraní. Pro vytvoření a spravování GUI byla do projektu implementována knihovna ButtonGUI. Tato knihovna byla vytvořena uživatelem metaldev pro Ogre3D.org

Inicializace buttonGUI spočívá ve vytvoření dvou hlavních instancí tříd.

- buttonManager - starající se o veškeré vykreslování GUI
- inputManager - starající se o interakci s prvky GUI

Pro aktualizaci prvků GUI a případnou interakci s nimi (označování tlačíkek, klikání, atd…) bylo nutné v metodě frameStarted toto GUI aktualizovat.

```
buttonEvent * e = buttonMgr->getEvent();
while(e){
      handleButtonEvent(e);
      e = buttonMgr->getEvent();
      }

buttonMgr->update();
buttonGUI::InputManager::getSingletonPtr()->capture();
```

3.6 Nepřátelé

3.6.1 Úvod

Jedním z nejdůležitějších aspektů hry jsou nepřátelé. Momentálně byly vytvořeny a kompletně animovány za pomoci modelovacího prostředí Blenderu 4 druhy nepřátel.

3.6.2 Pohyb nepřátel

3.6.2.1 ***Konstantní rychlost***

Aby se postava nepřítele pohybovala stejnou rychlostí na jakékoliv hodnotě FPS, bylo nutné využít v metodě frameStarted atribut evt.timeSinceLastFrame, který určuje uplynulý čas od posledního snímku. Touto hodnotou byla vynásobena velikost pohybu nepřítele, a tudíž se nepřítel pohybuje konstantní rychlostí.

3.6.2.2 ***Navigace***

Pro navigaci nepřátel byl naimplementován A* algoritmus (popsán v kapitole 3.8).

Díky němu je vypočítána nejkratší možná cesta od nepřítele k cíli. Tato cesta (vektor ukazatelů na jednotlivá pole k cíli) se může lišit pro jednotlivého nepřítele na základě jeho aktuální pozice.

Pro pohyb po jednotlivých polích této cesty byl ve třídě NepritelManager vytvořen SceneNode, který reprezentuje aktuální cíl, ke kterému se nepřítel vydá. Tento cíl je napozicován na následující políčko v cestě nepřítele. Jakmile k němu nepřítel dojde, nastaví se jeho pozice na další políčko, a tento postup se opakuje, dokud nepřítel nedojde do cíle. Díky tomu nepřítel projde celou svou trasu políčko po políčku.

3.6.2.3 ***Fronta***

Aby hra obsahovala dynamické vyhledávání cesty (např. když je nepříteli postavena do cesty věž, tak tuto věž obejde), bylo nutné vytvořit funkci, která detekuje všechny nepřátele, kteří jsou touto změnou struktury mapy ovlivněni. Těmto nepřátelům byla následně vypočítána nová trasa s ohledem na již provedené změny.

Toto řešení nebylo bohužel nejlepší, jelikož mohla nastat situace, kdy šlo mnoho nepřátel v řadě za sebou a jakmile jim byla postavena do cesty věž, provedlo se hledání cesty všem naráz, což vedlo k záseku hry z důvodu mnohonásobné časové složitosti algoritmu.

K zamezení tohoto zasekávání a pro lepší plynulost hry byla vytvořena fronta nepřátel čekajících na nalezení nové cesty. Tato fronta je iterována každou 0.15 sekundy a díky tomu se vyhledávání cest rozdělí do více časových úseků, což ve výsledku zamezí zmíněnému zasekávání hry.

3.6.3 Druhy nepřátel

Pro možnost, aby hra obsahovala co nejjednodušší tvorbu různých druhů nepřátel byla vytvořena XML struktura obsahující definice každého druhu nepřítele. Tento XML kód

se nachází v adresáři "res/modely/postavy/" a je pojmenován "enemies.xml". Definice jednoho druhu vypadá například takto:

```
<Enemy>
  <Names name="pesak" mesh="pesak.mesh" material="pesak"
folder="pesak" />
  <Appearance scale="0.2" fyzHeight="0.9" fyzWidth="0.5" />
  <Others speedForRunning="1.9" flying="false" aboveGround="0"/>
</Enemy>
```

Každý takto nadefinovaný druh musí mít ve výše zmíněném adresáři svou složku obsahující veškeré data nepřitele :

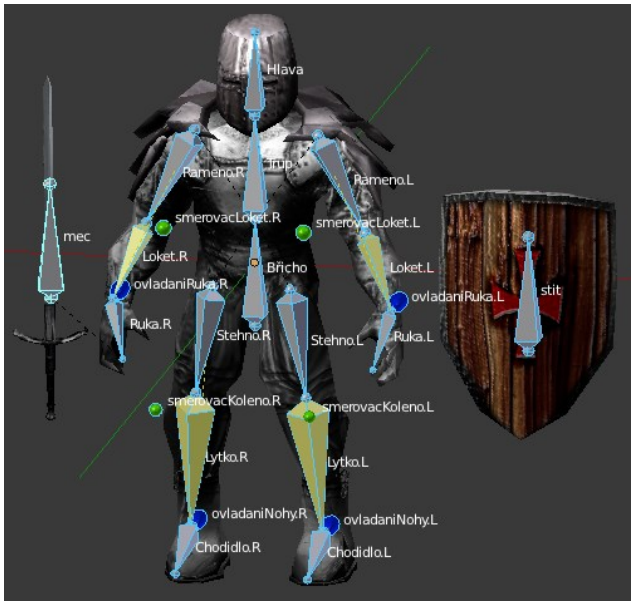
- soubor .mesh - model nepřítele
- soubor .material - OGRE skript definující materiál modelu
- soubor .skeleton - animace nepřítele
- textury

Díky této struktuře mají uživatelé možnost vytvořit vlastní druhy nepřátel v Blenderu a po následném exportu je zakomponovat do hry.

3.6.4 Animace

3.6.4.1 Vytvoření animace

Aby byla zrealizována animace postav, bylo nutné k modelu vytvořit armaturu za pomocí Blenderu. Armatura je v podstatě kostra modelu, kde každá kost ovlivňuje (deformuje) určitou část meshe.



Obrázek 1.9: *Ukázka vytvořeného nepřítele s armaturou*

Armatura z obrázku 1.9 obsahuje celkem 15 základních kostí deformujících mesh. Dále 8 ovládacích kostí, které nedeformují mesh, ale ovlivňují ostatní kosti (tyto kosti jsou znázorněny zeleně a modře) a zbylé dodatečné kosti (v tomto případě kosti pro meč a štít). Žlutě označené kosti reprezentují kosti, na které byla použita funkce inverse kinematics. Díky této funkci je zajištěna realistická simulace ohýbání kloubů a z toho vyplývající zjednodušení práce při tvorbě animací.

Samotná tvorba animace probíhala vhodným pozicováním a rotováním určitých kostí a následným zaznamenáním těchto pohybů.

Nepřátelům byly vytvořeny 4 druhy animací:

- chůze
- běhání
- smrt
- čekání

Vyexportováním byly získány soubory: .mesh - model postavy, .skeleton - informace o armatuře a animacích, .material - skript materiálu pro model

3.6.4.2 ***Načtení do OGRE***

Ke zprovoznění animací v OGRE bylo vytvořeno pole objektů třídy AnimationState. Velikost tohoto pole byla nastavena na 4, jelikož nepřítel využívá pouze 4 druhy animací. Toto pole se naplní metodou getAnimationState(Ogre::String &name), kde se v parametru name nastaví text právě přidané animace.

Pro samotné přehrávání aktuální animace je potřebná metoda addTime(Ogre::Real offset). Parametr offset značí časový posuv mezi fázemi animace, a proto se jeho hodnota nastaví na časové zpoždění mezi snímky (parametr evt.timeSinceLastFrame v metodě frameStarted). Díky tomu je přehrávání animací prováděno konstantní rychlostí na různých hodnotách FPS, stejně jako u pohybu nepřátel.

Rovněž byl vytvořen výčtový typ AnimID, ve kterém každá hodnota reprezentuje jeden určitý typ animace. Díky němu se stává práce s animacemi (např. nastavování aktuálního typu přehrávané animace) mnohem přehlednější.

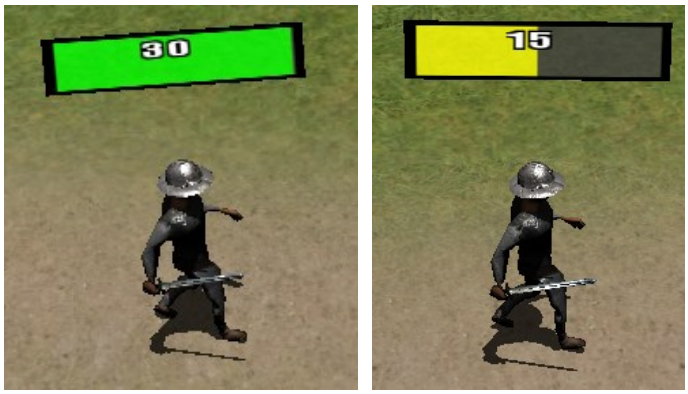
3.6.5 **Indikátory života**

Pro vizualizaci aktuálních životů nepřítele, bylo nutné vytvořit indikátor. Jako model tohoto indikátoru byl zvolen plane, který byl napozicován nad nepřítele a při každém průběhu frameStarted byl nasměrován na hlavní kameru.

Dále bylo nutné vytvořit materiál a novou texturu. Textura tohoto materiálu je nastavena na velikost 110x60 pixelů, přičemž bylo vyhrazeno 5 pixelů na každou stranu obrázku (černé okraje).

Po vypočítání procentuálního podílu aktuálních životů k životům maximalním, byly následně použity dva cykly typu for, díky kterým byl iterován každý pixel této textury a na základě porovnání jeho X souřadnice s aktuálním procentem životů, byla pro daný pixel nastavena barva.

Posledním krokem bylo vypsání textu životů na hotovou texturu. K tomuto účelu byla naimplementována metoda WriteToTexture.



Obrázek 1.10: *Indikátor života*

3.7 Naplnění kol

Naplnění kol nepřáteli bylo zrealizováno za pomocí XML skriptu. Tento skript se nachází v adresáři s mapou.

```
<Script startingGold="100" startingLifes="15" skybox="SkyBox2"
music="interier.ogg">
  <Round>
    <Unit type="pesak" health="30" speed="1" gold="10" spawn="1"
timeToSpawn="1" animSpeed="1" text="%HP%"/>
  </Round>
</Script>
```

Nově přidanému nepříteli lze nastavit počet životů, rychlost pohybu, počet zlata získaného jeho zabitím, místo ve kterém se vytvoří, čas vytvoření, rychlost animace a nakonec text v indikátoru života.

3.8 A* vyhledávání cesty

3.8.1 Úvod

Z pozorování mnoha her typu tower defense, bylo zjištěno, že jen málo z nich disponuje vyhledáváním cest pro nepřátele. Drtivá většina těchto her obsahuje již předem nadefinované cesty, které nemohou být ničím ovlivněny a to je vše.

Aby byla hra unikátní a více dynamická, byl zaimplementován algoritmus vyhledávání cest známý jako A*.

3.8.2 Popis

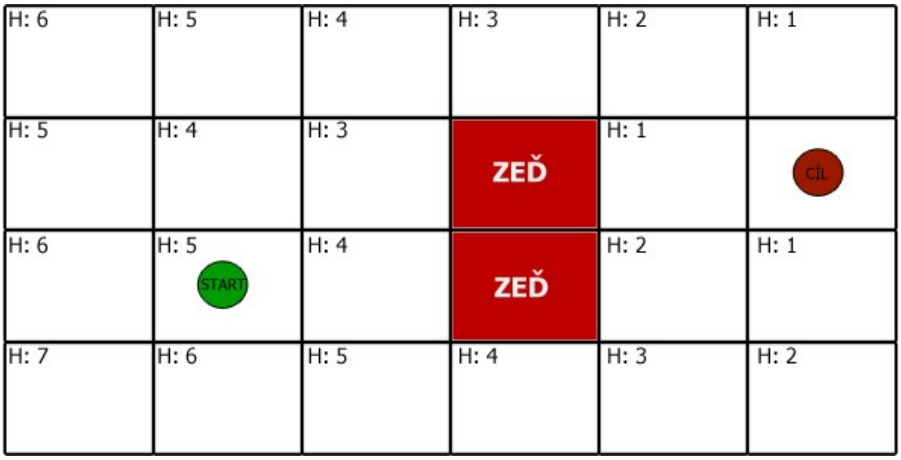
A* používá hladový princip pro nalezení optimální cesty z daného počátečního bodu do požadovaného koncového bodu. Tato cesta je nalezena na základě hodnot všech podbodů této cesty.

Postupné hledání cesty se zakládá na dvou listech (seznamech) uzlů.

- Otevřený list – obsahuje odkazy na uzly, které mohou být otestovány. Tento seznam se naplní sousedícími uzly právě testovaného uzlu. Samozřejmě se nenaplní těmi uzly, které jsou kolizní (zdi) a nebo již byly testovány (jsou v zavřeném listu). Rovněž každý nový uzel v listu dostane referenci na svého předka (na testovaný uzel).
- Zavřený list – obsahuje odkazy na uzly, které již byly otestovány.

K vyhodnocení nejkratší dostupné cesty využívá každý uzel funkce f(x),g(x) a h(x).

- h(x) – představuje heuristickou funkci. Jedná se o počet kroků mezi daným testovacím bodem a cílem, nehledě na překážky. K zjištění tohoto počtu byla použita tzv. Manhattan heuristická funkce, která v algoritmu funguje tak, že nejprve zjistí počet kroků mezi testovanými body ve vodorovném směru a následně i ve směru svislém a oba tyto výsledky sečte.



Obrázek 1.11: *Znázornění hodnot h(x) funkce*

- g(x) – představuje vzdálenost mezi počátečním a daným uzlem. Tato hodnota je kalkulována až v průběhu algoritmu. Všechny sousedící uzly v tomto případě mají stejnou vzdálenost (pro svislý a vodorovný pohyb složitost = 10 , pro šikmý pohyb složitost = 14).
- f(x) –součet h(x) + g(x)

3.8.3 **Implementace**

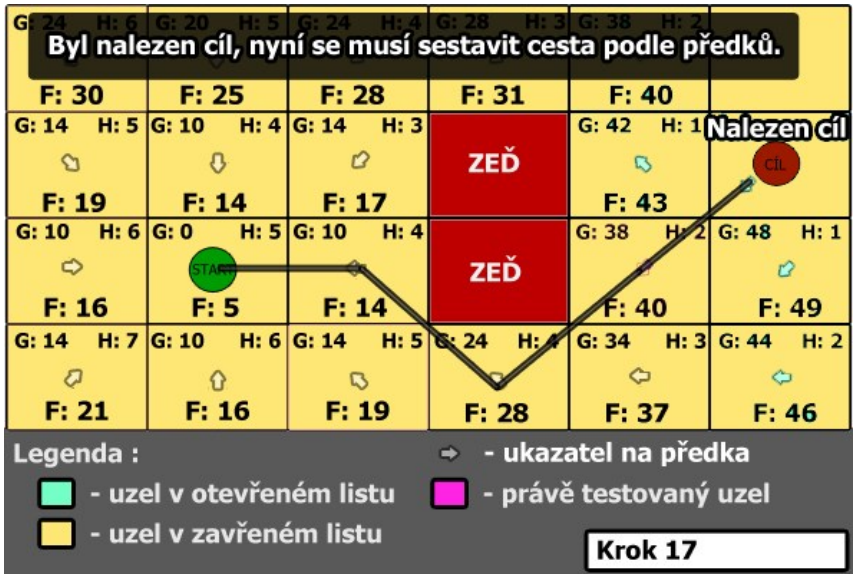
Pro práci s uzly byla vytvořena třída Policko, jež reprezentuje jeden daný uzel. Obsahuje všechny potřebné atributy G, H, F, ukazatele na sousedy, stav (pokud je kolizní nebo ne), ukazatele na předka a indikátory zda se jedná o start nebo cíl.

Při inicializaci programu se vytvoří vektor těchto políček o nastavené velikosti (A x A), kde A reprezentuje počet políček v horizontálním směru a vertikálním směru.

Nakonec byla vytvořena třída A_vyhledavac, která implementuje algoritmy samotného vyhledávání cesty.

Postupný průběh vyhledávání cesty funguje tímto způsobem:

- Jako parametry konstruktoru třídy A_vyhledavac jsou poslány ukazatelé na startovní políčko, cílové políčko a ukazatel na samotný vektor políček.
- Všem políčkům se vypočítá hodnota H za pomoci Manhattan heuristické funkce.
- Aktuálně testované políčko přidá všechny nekolizní sousedící políčka do otevřeného listu a rovněž se stane jejich předkem.
- Každému sousednímu políčku se vypočítá hodnota G a F.
- Aktuálně testované políčko se vloží do zavřeného listu a již nikdy nebude testováno.
- Poté algoritmus vyhledá v otevřeném listu políčko s nejnižší hodnotou F a nastaví jej jako aktuálně testované.
- Tento cyklus se bude opakovat tak dlouho, dokud nebude nalezeno cílové políčko nebo dokud nebude otevřený list prázdný.
- Pokud byl nalezen cíl, bude sestavena cesta pozpátku (od cíle ke startu) pomocí ukazatelů na předka každého políčka.



Obrázek 1.12: *Příklad nalezené cesty*

3.8.4 **Závěr**

Zrealizování tohoto algoritmu bylo velice přínosné, jelikož lze použít téměř v jakémkoliv žánru her. Jak od střílečeк až po strategie, všude kde je umělá inteligence a jednotky, které se potřebují přemístit z bodu A do bodu B.

V budoucnu je naplánováno obohatit jej o více komplexní uvažování. Přesněji vyhledání nejbezpečnější cesty místo té nejkratší. Zrealizováno by to mohlo být způsobem, kde by se vybrali všechny políčka v dosahu jakékoliv věže a následně by byla rapidně zvýšena jejich hodnota G. Což by mělo za důsledek vyhýbání se těmto polím, pokud by byla přístupná jiná cesta.

Obrázek 1.13: *Vizualizace A* v OGRE aplikaci*

3.9 Věže

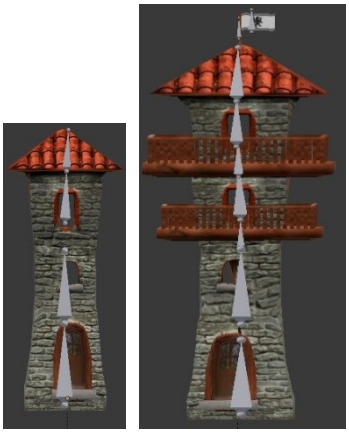
Jedním z hlavních prvků hry bylo vytvoření systému věží, které by bránili cestu všem nepřátelům. Tak jako u ostatních prvků hry byla zvolena XML struktura pro definování síly, počtu střel za sekundu, dosahu a ceny určitých věží.

Základem každé věže či zdi je rodičovská třída stavba, která implementuje veškeré potřebné atributy a metody pro práci s ní.

3.9.1 Animace stavby

Z důvodu oživení hry byly vytvořeny animace stavění věže a jakéhokoliv dalšího vylepšování této věže. Stejně jako u nepřátel je tato animace založena na armatuře. Každá z kostí armatury je přichycena za pomoci weight paint módu k určité části věže. Tyto části bylo poté potřeba zvětšovat a vhodně pozicovat, aby byla výsledným dojmem postupná stavba věže.

Následně bylo nutné v OGRE aplikaci vytvořit časování stavby či vylepšení věže.



Obrázek 1.14: *Základní úroveň věže (nalevo), nejvyšší úroveň (napravo)*

3.9.2 Projective decals

K vytvoření radiusu věže (znázorněný dosah věže vykreslený na povrchu terénu), byla použita technika zvaná (projective decals). Tato technika využívá třídu Ogre::Frustum.

```
// Vytvoření frustum

mDecalFrustumRadius = new Ogre::Frustum();
// ortografická projekce
mDecalFrustumRadius->setProjectionType(Ogre::PT_ORTHOGRAPHIC);
mDecalFrustumRadius->setAspectRatio(1.0);
// velikosti OrtoWindow jsou měněny v průběhu hry v závislosti
na dosahu věže
mDecalFrustumRadius->setOrthoWindowWidth(1);
mDecalFrustumRadius->setOrthoWindowHeight(1);
```

Následně byl vytvořen nový objekt typu SceneNode a toto frustum bylo k němu přichyceno. Po přichycení bylo nutné SceneNode napozicovat vysoko nad terén a nastavit mu rotaci kolmo dolů.

```
mProjectorNodeRadius->attachObject(mDecalFrustumRadius);
```

Nakonec bylo nutné nalézt materiál terénu, přidat mu nový pass a tomuto passu přiřadit novou texturu s nastaveným projekčním texturováním referovaným na již vytvořené frustum.

```
Ogre::TextureUnitState* texState =
passRadius->createTextureUnitState("radius.png");
texState->setProjectiveTexturing(true, mDecalFrustumRadius);
```

Obrázek 1.15: *Ukázka projective decals*

3.9.3 Vyhledávání cílů

Pro vyhledání cíle věže byl vytvořen algoritmus, který najde všechny potencionální cíle, zjistí, zda se nachází v dosahu věže a nakonec porovná jejich vzdálenost od cíle. Nepřítel s nejkratší vzdáleností je označen jako cíl a tím zůstane, dokud nezemře či neopustí dosah věže.

Navíc byly věže obohaceny algoritmem, který na základě již letících střel na nepřítele zjistí, zda nepřítel zemře. Pokud ano, nemá smysl po něm střílet novou střelu, a proto se najde nový cíl.

3.10 Bullet fyzika

3.10.1 Popis

Jelikož OGRE je pouze renderovací engine, nedisponuje tedy funkcemi, které by dokázaly simulovat gravitaci, tlakové síly čí kolize objektů. Z tohoto důvodu byl do aplikace naimplementován fyzikální engine Bullet.

Hlavním principem tohoto enginu je vytvoření fyzikálního světa a fyzikálních objektů v něm. Tyto fyzikální objekty nejsou při renderování viditelné, ale mohou k nim být přichyceny entity, které dokáží vizualizovat všechny změny pozice a rotací těchto objektů.

Aby engine dokázal aktualizovat změny ve fyzikálním světě, je nutné při každém renderování snímku zavolat metodu stepSimulation na objekt fyzikálního světa (v této aplikaci btDiscreteDynamicsWorld) s parametrem časového zpoždění od posledního renderování snímku.

OGRE podporuje rovněž druhý populární engine PhysX od NVIDIE.

3.10.2 Implementace

Pro vytvoření fyzikální scény bylo třeba vytvořit konvertor, který převede tvar viditelného objektu na tvar fyzikální. Využit byl objekt třídy StaticMeshToShapeConverter a pomocí něj byl viditelný tvar převeden na TriangleMeshCollisionShape pro statické objekty a ConvexHullCollisionShape pro dynamické objekty.

Výsledné fyzikální tvary bylo potřeba zvětšit na stejnou velikost, jaké měli viditelné tvary. Pro tento účel byl využit Bullet objekt btScaledBvhTriangleMeshShape, kterému bylo za potřebí v konstruktoru vložit fyzikální tvar a vektor zvětšení.

V tuto chvíli byl vytvořen totožný fyzikální tvar s tvarem viditelným. Dalším krokem bylo nastavit stejnou pozici i rotaci. K tomuto účelu Bullet využívá tzv. transform. Je to v objekt této třídy uchovávající informace o pozici a rotaci fyzikálního objektu ve fyzikálním světě.

Poslední částí bylo vytvořit tento nachystaný fyzikální tvar do bullet světa. K tomu byla využita již výše zmíněná metoda addRigidBody.

Tato metoda přímá tyto parametry:

- btTransform &transform – pozice a rotace fyzikálního modelu
- btCollisionShape*shape – samotný tvar fyzikálního modelu
- btScalar mass – hmotnost objektu. Díky této hmotnosti lze v Bullet fyzice rozlišit rozdíl mezi statickým a dynamickým objektem tak, že statický objekt má tuto hodnotu nulovou, kdežto dynamický ne.
- SceneNode * node – odkaz na SceneNode viditelného objektu. Je potřebný kvůli tomu, aby šla vidět projekce pohybu a změny fyzikálního modelu na model viditelný. (propojuje fyzikální model s viditelným)

Dynamické objekty se vytvářejí podobným způsobem až na to že místo TriangleMeshCollisionShape využívají ConvexHullCollisionShape a jejich mass nesmí být 0.

Obrázek 1.16: *Nepřítel před kolizí s dynamickými objekty (nalevo) a po kolizi (napravo)*

3.11 Testování hry

Hra byla testována na počítači s konfigurací :

CPU : Intel® Core™ 2 Duo 2.00 GHz

RAM : 4,00 GB

GPU : NVIDIA GEFORCE™ G102M 512 MB

Tabulka 1.1: *Naměřené výsledky testování*

Testovaná vlastnost	Hodnota
Průměrná hodnota FPS	35
Průměrné zatížení RAM paměti	500 MB
Průměrné zatížení procesoru	45%

Výsledky testování dopadly dobře, když se bralo v potaz stáří počítače a jeho aktuální stav. Velikost zatížení FPS a paměti RAM z velké části záleží rovněž na tom, jak je aktuální mapa vytvořená.

Testování rovněž proběhlo na jiných počítačích a na všech byla hra úspěšně zprovozněna po doinstalování potřebných knihoven (balíček Visual Studia 2012 a openAL knihovna).

.

Závěr

V této práci byly popsány postupy vytváření 3D aplikací v C++ s použitím OGRE enginu s implementací Bullet fyziky a modelováním v Blenderu. Vyzkoušeno bylo mnoho technik, které byly následně aplikovány při tvorbě strategické hry. Použití těchto technik je zdokumentováno a mohlo by být použito jako návod pro vývojáře, kteří začínají pracovat s OGRE 3D enginem.

Tato práce byla velice přínosná, jelikož jsem se naučil modelovat v programu Blender, včetně animování postav a dokázal jsem vytvořit za pomocí C++ a OGRE enginu hratelnou 3D hru. Rovněž jsem se naučil mnoho technik používaných při vytváření 3D scén jako např. normal mapping, animace, specular mapping, využití částic, alpha splatting, atd.

Samotná hra je nyní ve stavu, kdy by mohla být rozšířena mezi širší okruh uživatelů, kteří by mohli otestovat její funkčnost a popřípadě podali odezvu s návrhy na její zlepšení. V budoucnosti bych chtěl zlepšit výkon hry za použití Ogre třídy InstancedGeometry, se kterou jsem ještě neměl možnost pracovat. Dále bych chtěl vylepšit algoritmus pro navigaci, vytvořit více typů věží, nepřátel a nové mapy. V oblasti Bullet fyziky bych chtěl na postavy nepřátel aplikovat rag-doll fyziku.

Vesměs je v oblasti této hry ještě mnoho věcí, které by se dali vylepšit či realizovat a mám v plánu na nich pokračovat v budoucnosti.

Použitá literatura

[1] THE OGRE TEAM. OGRE [online]. 2000 [cit. 2014-03-30]. Dostupné z: http://www.OGRE3d.org/

[2] OgreOggSound [online]. 2008-08-16 [cit. 2014-04-28]. Dostupné z: http://www.ogre3d.org/tikiwiki/tiki-index.php?page=OgreOggSound

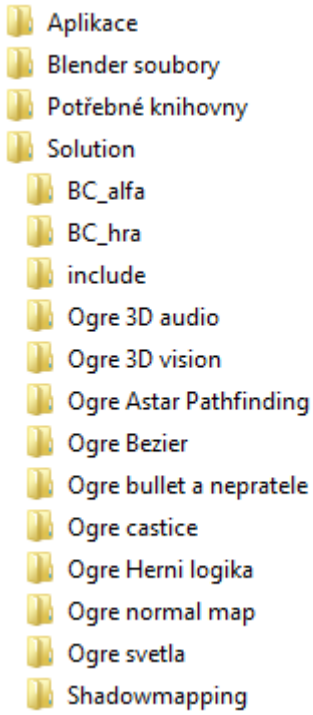
[3] Dimitri Bueno [online]. 2009 [cit. 2014-04-28]. Dostupné z: http://dimitribueno.dreamhosters.com/?p=118

Seznam obrázků

Seznam příloh

Součástí BP je DVD.

Adresářová struktura přiloženého DVD:



poznámka: Pro úspěšné spuštění všech programů je nutné zkopírovat obsah DVD na disk.

Příloha 2. - Třídní diagram