Programátorská dokumentace

Jakub Melka
Štěpán Poljak
Martin Růžička
Martin Urza

Obsah

Část I

Klientská část

1 Úvod

1.1 Instalace

Klient je napsaný v jazyce $C++$ ve vývojovém prostředí Visual Studio 2010. Nejaktuálnější verzi zdrojových kódů klienta je možné stáhnout z následující adresy.

Url: http://xp-dev.com/svn/DivergentTerritory
Login: AnonimniUzivatel
Heslo: aa4Smn1N

Kromě standardních C++ knihoven využívá klient také dvě externí knihovny: Qt a Cg. Tyto knihovny je potřeba na začátku ručně doinstalovat.

1.1.1 Qt knihovna

Klient disponuje velkým množstvím widgetů jako například okna, panely, tabulky nebo tlačítka. Proto jsme se rozhodli použít knihovnu usnadňující GUI programování. Knihovnu Qt jsme potom vybrali z několika hlavních důvodů.

1. Multiplatformnost: klient je v současné verzi kompatibilní pouze s PC vybaveným operačním systémem Windows. Případná portace na jiný OS nebo platformu by však s ohledem na použité knihovny neměla představovat zásadnější problém.

2. Stylovatelnost: GUI prvky v Qt je možné jednoduše měnit pomocí tzv. Stylesheets. Ty umožňují kompletně předefinovat vzhled jednotlivých grafických komponent a tím dodat aplikaci originální grafický styl.

V našem projektu používáme Qt ve verzi 4.7.1, nicméně problém by neměl nastat ani s vyššími verzemi knihovny. Knihovnu Qt je možné stáhnout z těchto stránek.

http://qt.nokia.com/products/

Staženou knihovnu je nakonec potřeba nainstalovat a zkompilovat. Podrobnější informace o kompilaci je možné najít například na níže uvedených stránkách.

http://thomasstockx.blogspot.com/2011/03/qt-472-in-visual-studio-2010.html

1.1.2 Cg knihovna

Cg je knihovna pro práci s shadery. Shadery jsou krátké programy běžící přímo na grafické kartě. To nám mimo jiné umožňuje programování pokročilých grafických efektů prostřednicvím vertex a pixel shaderů. Knihovnu Cg je možné stáhnout z uvedeného odkazu níže.

http://developer.nvidia.com/cg-toolkit.

1.2 Hardwarové požadavky klienta

512 MB RAM
1 GHz procesor
Grafická karta s alespoň 256 MB VRAM a podporou OpenGL 2.0
DVD mechanika
Windows 2000 nebo novější
disk 100 MB instalace

Poznámka: Uvedená sestava představuje minimální hardwarové požadavky pro úspěšné spuštění hry. Pro plynulé hraní za každé situace může být vyžadována konfigurace podstatně výkonnější.

1.3 Testovací sestavy

Klient byl úspěšně otestován na následujících hardwarových konfiguracích.

CPU	RAM	GPU	OS
Intel Core 2 Q6600	4GB	NVidia GTX470	Windows7 64bit
Intel Core i3 330M	4GB	ATI Mobility Radeon HD5470	Windows7 32bit
Intel Core 2 Q6600	4GB	NVidia 8800GTX	Windows Vista 32bit
AMD Athlon II X4 620	4GB	NVidia 8800GTS	Windows XP SP3
Intel Core 2 Q9550	4GB	NVidia 9800GT	Windows XP SP3
Intel Core 2 ???	2GB	???	Windows XP SP3
AMD Athlon 64 3000+	2GB	Ati Radeon HD 2600XT	Windows XP SP2

2 Matematematické knihovny

2.1 Matice

`Matrix4x4<T>` je třída určená pro práci s 4x4 maticemi. Uspořádání prvků matice je sloupcově dominantní:

$$Matrix4x4 < T >= \begin{pmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix}.$$

Prvky a_{12}, a_{13} a a_{14} definují posun na osách x,y,z. Rozložení prvků v matici tak můžeme napsat jako:

$$Matrix4x4 < T >= \begin{pmatrix} a_0 & a_4 & a_8 & posunX \\ a_1 & a_5 & a_9 & posunY \\ a_2 & a_6 & a_{10} & posunZ \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix}.$$

Reprezentace matice je plně kompatibilní s OpenGL maticemi a třídami `Point_3<T>` a `Vector_3<T>` , které lze maticemi jednoduše transformovat.

Třída dále nabízí základní binární maticové operace jako je sčítání, odečítání a násobení. Matici můžeme také vynásobit sloupcovým nebo řádkovým vektorem a získat tak vektor přetransformovaný danou maticí. K dispozici jsou dále metody pro nahrání nulové, jednotkové, translační, rotační a měřítkové (scale) matice. Implementovány jsou i jejich zakladní operace: rotace, posun a změna měřítka.

- $identita = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
- $posun = \begin{pmatrix} 1 & 0 & 0 & posunX \\ 0 & 1 & 0 & posunY \\ 0 & 0 & 1 & posunZ \\ 0 & 0 & 0 & 1 \end{pmatrix}$
- $rotace = \begin{pmatrix} rotaceXX & rotaceYX & rotaceZX & 0 \\ rotaceXY & rotaceYY & rotaceZY & 0 \\ rotaceXZ & rotaceYZ & rotaceZZ & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
- měřítko $= \begin{pmatrix} \text{měřítkoX} & 0 & 0 & 0 \\ 0 & \text{měřítkoY} & 0 & 0 \\ 0 & 0 & \text{měřítkoZ} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

2.2 Vektor

`Vector_3 <T>` je třída obsahující metody pro práci s vektory ve 3-rozměrném euklidovském prostoru. Obsahuje pomocné metody pro operace typu vektor-vektor i typu vektor-skalár. Dva vektory tak lze sčítat a odečítat, počítat skalární i vektorový součin nebo úhel, který spolu dva vektory svírají. K vektoru je dále možné přičítat a odečítat skaláry, případně ho násobit nebo dělit skalárem. Vektor je můžeme také rotovat, počítat jeho velikost nebo ho normalizovat na vektor jednotkové délky.

2.3 Bod

`Point_3<T>` je třída obsahující metody pro práci s body v 3-rozměrném euklidovském prostoru. Třída obsahuje metody pro operace typu bod-bod a bod-vektor. Mezi 2 body je tak možné například počítat euklidovskou vzdálenost, odečíst je od sebe a tím získat vektor rozdílu a další. K bodu můžeme přičítat nebo odečítat vektor a tím bod posouvat ve směru vektoru .

2.4 Vektor vs Bod

Třídy `Point_3<T>` a `Vector_3<T>` jsou si poměrně podobné a mají společnou část implementace. Hlavní rozdíl je v pojmenování metod a koncepci, ke které jsou určeny. Zatímco například sčítání 2 vektorů je dobře definovaná operace, sčítat 2 body nelze. Podobně je například možné určit velikost vektoru nebo vektor normalizovat, kdežto s body tyto operace nemají rozumnou interpretaci a smysl. Tento systém rozdělení je výhodný zejména ze dvou důvodů:

1. Poskytuje zvýšenou typovou bezpečnost, vektory a body jsou odlišného typu.
2. Umožňuje intuitivnější náhled na problém. Uvedeným typovým odlišením je na první pohled zřejmé, zda na proměnnou nahlížet jako na bod nebo vektor.

Interně jsou body a vektory implementovány čtveřicí atributů (x,y,z,w). Složky x,y,z udávají pozici nebo směr na jednotlivých osách. Poslední atribut w je pro body roven 1, pro vektory roven 0 a slouží pro rozšířené maticové operace tak, jak jsou definovány v OpenGL.

$Point\_3 < T >= (x, y, z, 1)$

$Vector\_3 < T >= (x, y, z, 0)$

Násobení translační 4x4 matice vektorem zanechá vektor beze změny.

$$\begin{pmatrix} 1 & 0 & 0 & posunX \\ 0 & 1 & 0 & posunY \\ 0 & 0 & 1 & posunZ \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = (x, y, z, 0)$$

Násobení bodu stejnou tranformační maticí způsobí posun bodu v závislosti na použité translační matici.

$$\begin{pmatrix} 1 & 0 & 0 & posunX \\ 0 & 1 & 0 & posunY \\ 0 & 0 & 1 & posunZ \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = (x + posunX, y + posunY, z + posunZ, 1)$$

Mezi vektory a body je samozřejmě možné provádět explicitní konverzi v případě, že je taková operace potřeba.

3 Podpora pro vykreslovaní

3.1 Proces vykreslovaní

Rendering je poměrně složitý a komplexní proces. Kompletní postup od inicializace, zobrazení až po uvolnění použitých prostředků můžeme rozdělit na několik částí.

1. inicializace OpenGL
2. nahrání textur na GPU
3. nahrání geometrie na GPU
4. nastavení kamery, světel a objektů scény
5. render scény (odkazuje se na data nahraná na GPU)
6. zobrazení vyrenderované scény v přednastaveném okně
7. smazání GPU dat
8. uvolnění ostatních OpenGL prostředků

3.2 Render třídy

3.2.1 AbstractRender

`AbstractRender` je abstraktní bázová třída zajišťující render. Primárně je třída navržena pro použití s GPU renderem. Rozhraní je tak přednostně uzpůsobeno možnostem a logice grafických karet. `AbstractRender` definuje základ společného rozhraní, které musí všechny odvozené renderovací třídy implementovat. Logiku a postup při vykreslování můžeme shrnout do následujícího seznamu.

1. nastavení parametrů třídy (kam a jak renderovat)
2. zahájení renderu nového snímku zavoláním metody `beginFrame()`

3. kreslení geometrie scény

4. ukončení renderu snímku a zobrazení výsledného obrázku zavoláním metody `endFrame()`

3.2.2 Render

`Render` třída je odvozena od `AbstractRender` , je navržena pro perspektivní rendering a implementována prostřednictvím OpenGL. Kromě základního rozhraní předepsaného třídou `AbstractRender` poskytuje třída velké množství dalších funkcí. Můžeme tak renderovat základní geometrické tvary jako je čára, kružnice, koule nebo třeba polygon. Dále je možné renderovat komplexnější geometrii jako například modely, částice, billboardy, obrázky nebo bitmapový text. Před začátkem renderu každého nového snímku je navíc možné nastavovat parametry kamery, z jejíž pohledu se scéna bude renderovat.

3.2.3 GLPhongRender

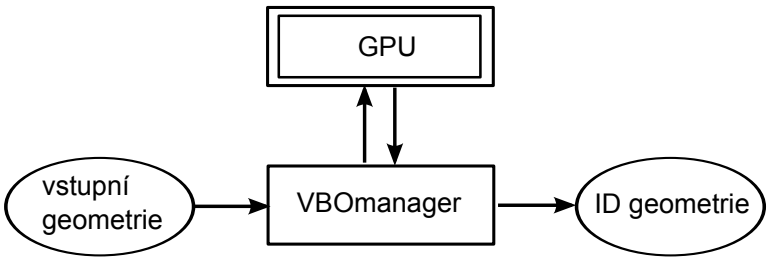
Rozšiřuje možnosti třídy `Render` o osvětlování geometrie prostřednictvím Phongova světelného modelu. Tento výpočet je prováděn per-pixel pomocí dodávaného shaderu. Shader kompatibilní s touto třídou je uložen v adresáři `kodiak/shaders/phongShaderSoftProj.cg` .

3.2.4 SoftProjRender

Render třída odvozená od předešlého `GLPhongRender` , obsahuje další pomocné kreslící funkce napsané na míru potřebám hry.

3.3 Vertex Buffer Object správce

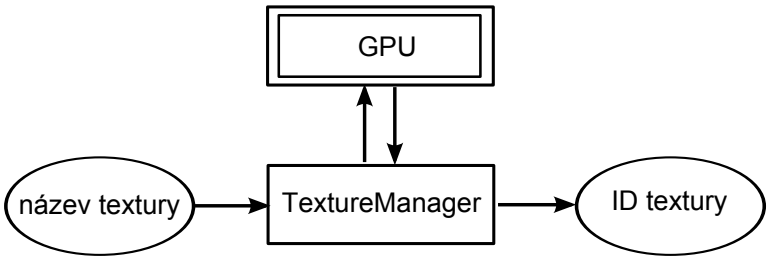
`VBOmanager` je správce dat pro práci s grafickými primitivy uloženými přímo v paměti grafické karty. Prostřednictvím této třídy může uživatel nahrávat na grafickou kartu geometrii: pole vertexů, normál a texturovacích koordinátů. K nim lze poté prostřednictvím vrácených OpenGL identifikátorů přistupovat. Tyto identifikátory jsou ukládány společně s uživatelem definovaným názvem do databáze. Prostřednictvím zadaného názvu je možné získat data uložená v databázi a dále s nimi potom pracovat.



Obrázek 1: Vertex Buffer Object: nahrání nové geometrie.

3.4 Textury

`TextureManager` je singleton třída sloužící ke správě OpenGL textur. `TextureManager` umožňuje nahrávat textury, mazat je a dále s nimi pracovat. Po vložení textury dojde k jejímu nahrání do paměti grafické karty. S nahranou texturou potom můžeme pracovat pouze nepřímo pomocí navráceného OpenGL identifikátoru. Tento identifikátor spolu s uživatelsky definovaným názvem textury se ukládá v databázi. K texturovým OpenGL identifikátorům tak můžeme s pomocí databáze přistupovat i pod uživatelem zadaným názvem a s texturami tak pohodlně pracovat.



Obrázek 2: Texture manager: práce s texturou.

3.5 Materiály

Třída `Material` je abstraktní bázová třída každého materiálu. Od ní je odvozena třída `PhongMaterial` . Ta obsahuje následující atributy materiálu.

- float ambient[3]
  RGB trojice ambientní složky materiálu.
- float diffuse[3]
  RGB trojice difůzní složky materiálu.
- float specular[3]
  RGB trojice glossy složky materiálu.
- float exponent
  exponent určující chování lesklého materiálu.

Tyto parametry jsou potřebné pro výpočet osvětlení pomocí Phongova osvětlovacího modelu, který bude popsán v jedné z následujících částí.

3.6 Phongův světelný model

Phongův světelný model aproximuje osvětlení ve scéně rozdělením na 3 části.

1. ambientní složka
2. difůzní složka
3. lesklá (glossy) složka

Ambientní osvětlení spočteme jako: $Final_{ambient} = C_a \cdot I_a$, kde

- C_a je ambientní složka materiálu
- I_a je ambientní složka světla

.

Difůzní složku osvětlení spočteme jako: $Final_{diffuse} = C_d \cdot I_d \cdot (N \cdot L)$, kde

- C_d je difůzní složka materiálu
- I_d je difůzni složka světla
- N je normála povrchu
- L je vektor světla

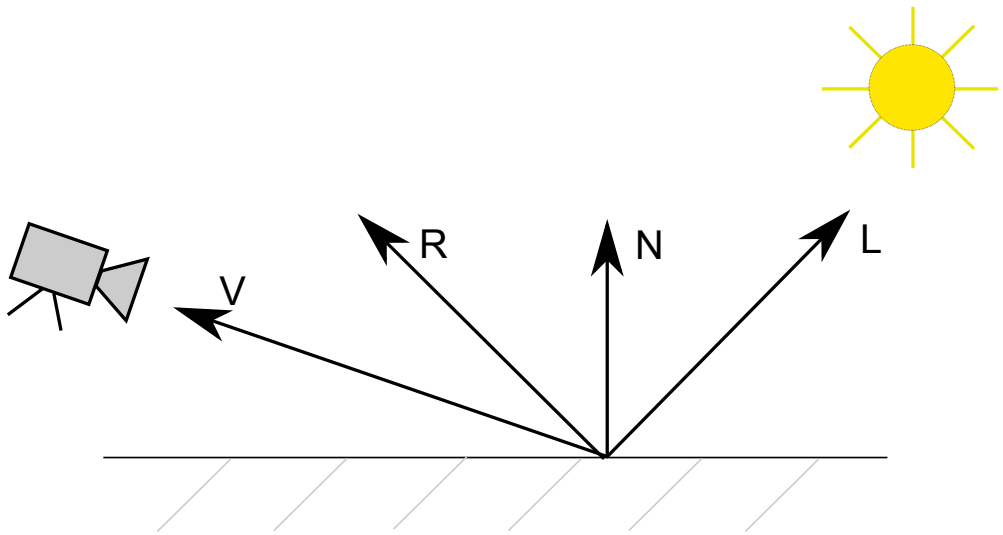
.

Glossy složka je určena vztahem: $Final_{glossy} = C_s \cdot I_s \cdot (V \cdot R)^n$, kde

- C_s je lesklá složka materiálu
- I_s je lesklá složka světla
- V je vektor pohledu
- R je vektor ideálního odrazu dopadajícího světla
- n je exponent určující další nastavení lesklosti

.

Výsledné osvětlení nakonec spočteme součtem všech tří uvedených komponent: $Final_{radiance} = Final_{ambient} + Final_{diffuse} + Final_{glossy}$.



Obrázek 3: Phongův světelný model

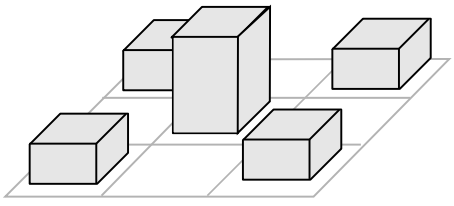
3.7 Načítání geometrie z externích souborů

Program podporuje načítání geometrie ze dvou externích formátů: výškové mapy a OBJ souboru.

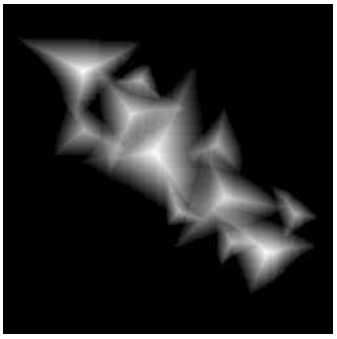
3.7.1 Výšková mapa

Výšková mapa je obvykle reprezentovaná ve formě textury. Pozice texelu společně s jeho hodnotou udává souřadnice vrcholu. Konverzi takového vstupu na vertexy, normály a texturové koordináty zajišťuje třída `HeightMapImporter` . Vrácená a zpracovaná data je potom možné nahrát prostřednictvím třídy `VBOmanager` do paměti grafické karty.

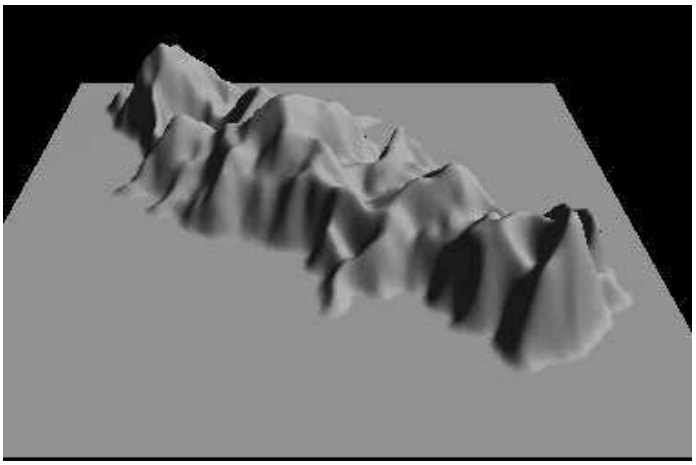
10	0	10
0	50	0
10	0	10



Obrázek 4: Schéma: data zdrojové textury vlevo a vpravo výsledná geometrie



Obrázek 5: Ukázka výškové textury...



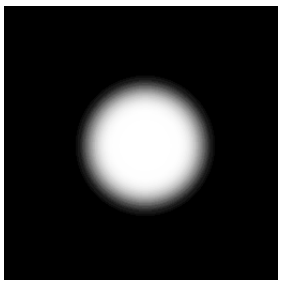
Obrázek 6: ...a z ní vygerenované geometrie.

Skriptování

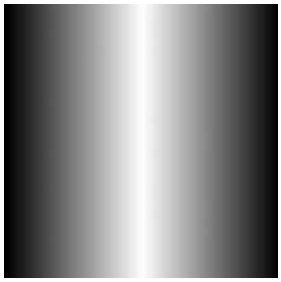
Hlavní výhodou výškových map je jejich jednoduchost. Práce s nimi je intuitivní a lze pomocí nich vytvářet rychle geometrii terénu. My jsme tento systém rozšířili ještě o systém operátorů, kdy lze výškové mapy dále upravovat. K dispozici jsou operace typu výšková mapa - výšková mapa i výšková mapa - skalár. Můžeme tak například sečíst 2 výškové mapy, výdělit jejich součet dvěma a tím získat zprůměrovanou výškovou mapu apd. Systém operátorů je doplněn o jednoduchý matematický parser, prostřednictvím kterého lze skriptovat geometrii (viz soubory `MathParser.h` a `MathTextureParser.h` `TextureOperationParser.h` .

Ukázka skriptu:

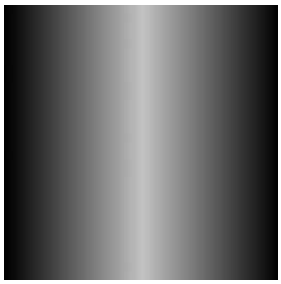
```
geometryScript "tex1.tga"+( 0.5 * "tex2.tga" )
```



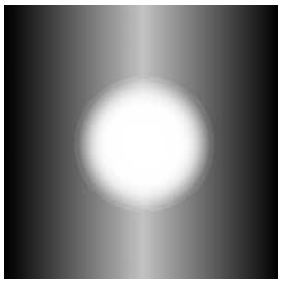
Obrázek 7: tex1.tga



Obrázek 8: tex2.tga



Obrázek 9: 0.5 * tex2.tga



Obrázek 10: tex1.tga + (0.5 * tex2.tga)

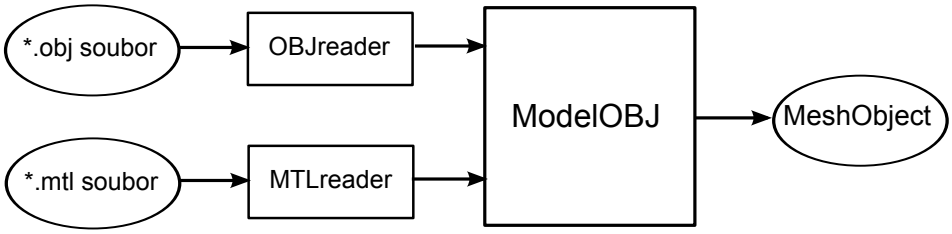
3.7.2 OBJ a MTL formát

OBJ

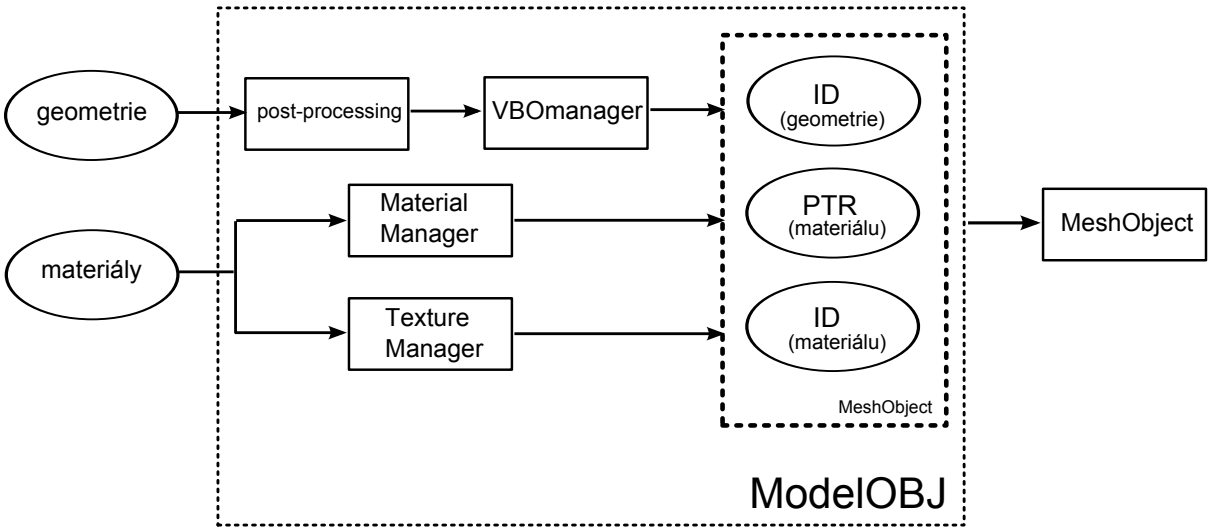
OBJ je poměrně oblíbený grafický formát, který je podporován většinou modelovacích nástrojů jako například 3D Studio Max nebo Maya. Formální specifikace formátu je k dispozici například na http://paulbourke.net/dataformats/obj/. K načítání tohoto formátu slouží třída `ModelOBJ<T>` . Ta přečte a naparsuje data ze souboru a vrátí zpracovaná data k dalšímu použití. Ty můžeme dále opět s pomocí třídy `VBOmanager` uložit na grafickou kartu.

MTL

S OBJ formátem velmi úzce souvisí MTL formát. Ten definuje materiály a geometrie specifikovná v OBJ formátu se do tohoto souboru odkazuje. Parsování obstarává třída `MTLreader` a správu materiálů potom třída `MaterialManager<T>` Podrobnosti k formátu je možné získat zde http://paulbourke.net/dataformats/mtl/. Materiál kromě atributů popisující jeho vlastnosti (viz třída `Material`) může obsahovat i názvy textur, které používá. Textury jsou podobně jako geometrie uložena ve VRAM paměti, odkud je možné s nimi při kreslení rychleji pracovat (viz výše popsaný `TextureManager`).



Obrázek 11: Proces zpracovaní *.obj a *.mtl souborů.



Obrázek 12: ModelOBJ objekt

3.8 Model

Třída `Model` slouží k reprezentaci komplexních objektů v 3-rozměrném prostoru. Pozice, velikost a orientace objektu je definována pomocí matice třídy `Matrix4x4<T>` , kterou jsme popsali v jedné z předchozích částí dokumentace. Každý model se dále skládá z jedné nebo více homogenních částí, tak zvaných Meshů. Mesh je skupina polygonů používající společnou sadu vertexů, texturovacích koordinátů, normál, materiálů a textur. Mesh reprezentujeme prostřednictvím třídy `MeshObject` . `MeshObject` obsahuje informace o použitých texturách, materiálu a geometrii objektu.

```
class Model
{
private:
        Matrix4X4<float> modelMatrix;
        vector<MeshObject> meshes;

public:
        ...
};
```

3.9 Kamera

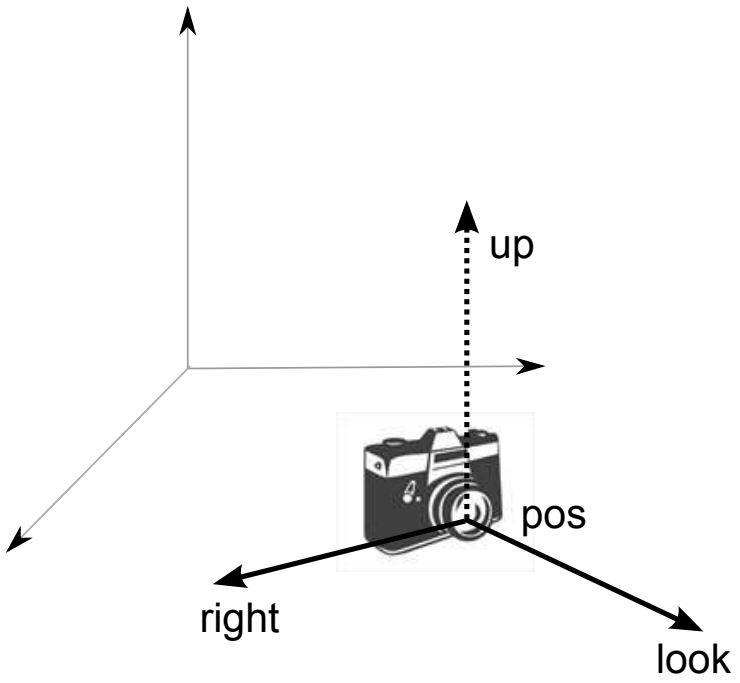
Třída `Camera<T>` slouží k práci s kamerou v 3-rozměrném prostoru. Nastavovat můžeme pozici nebo směr pohledu kamery a tím měnit pohled na scénu. Pro pohodlnější práci s kamerou nabízí třída řadu pomocných metod jako například posun kamery ve směru pohledu, otáčení kamery a další. Konfigurace kamery je určena následujícími atributy.

- $Point\_3 < T > pos$
  Určuje umístění kamery.

- $Vector\_3 < T > look$
  Určuje směr, kterým se kamera dívá.

- $Vector\_3 < T > right$
  Určuje kde je z pohledu kamery pravá strana.

- $Vector\_3 < T > up$
  Určuje kterým směrem je z pohledu kamery vzhůru. Dopočítává se automaticky jako kolmice na look a right vektor.



Obrázek 13: Kamera

3.10 Proces zpracování a zobrazení geometrie

1. Načtení geometrie. Nejprve je potřeba načíst a rozparsovat geometrii modelu ze souboru. Načítat lze geometrii ve formě výškové mapy nebo ve formátu OBJ.

2. Zpracování geometrie. Načtenou geometrii v další části převádíme do podoby kompatibilní s OpenGL Vertex Array a tu pak ukládáme prostřednictvím OpenGL VBO do paměti grafické karty. Tím, že máme geometrii uloženou přímo na GPU výrazně snižujeme objem dat, která je potřeba během renderingu přenášet mezi CPU a GPU. Tím zásadním způsobem urychlujeme kreslení geometrie.

3. Načtení materiálů a textur. V další části je potřeba načíst a zpracovat materiály a textury, které geometrie používá. Program podporuje načítání materiálů ve formátu MTL.

4. Tvorba modelu. V této chvíli máme načtenou a zpracovanou geometrii a materiály modelu. Odkazy na geometrii a materiály máme uloženy ve správci geometrie a materiálu. Můžeme tak jednoduše vytvořit model používající uloženou geometrii a materiály (viz dříve popsaná třída `Model`).

5. Kreslení modelu. Na závěr můžeme vytvořený model jednoduše vykreslit prostřednictvím dříve popsané třídy `Render` .

3.11 Částicový systém

Grafický engine je vybaven také podporou částicových systémů. Částicový systém je rozdělen na 2 části: částice a správce částic.

3.11.1 Částice

Částice je jednoduchý objekt s jehož pomocí se simulují různé objekty bez pevně dané struktury jako například kouř, sníh, tryskající voda a další.

`BasicParticle`

Je základní třída reprezentující částici. Třída obsahuje následující atributy.

- `Vector_3<float>` velocity
  Udává rychlost a směr pohybu částice.

- float acceleration
  Udává zrychlení částice ve směru pohybu.

- float lifeTime
  Životnost částice.

- `Point_3<float>` position
  Pozice částice.

Třída dále obsahuje metodu `simulate()` , která pomocí nastavených atributů částice provádí simulaci. K simulaci používáme Eulerovskou diskrétní metodu prvního řádu.

```
void BasicParticle::simulate( float dt )
{
    this->velocity += this->acceleration * dt;
    this->position += this->velocity * dt;
    this->lifeTime -= dt;
}
```

`GraphicParticle`

Je pokročilá částicová třída odvozená od `BasicParticle` . Obsahuje navíc atributy popisující částici z grafického hlediska.

- float colorOffset[4]
  Určuje, jak se bude měnit barva částice během simulace.
- float sizeOffset
  Určuje, jak se bude měnit velikost částice během simulace.
- float color[4]
  Určuje aktuální barvu částice.
- float size
  Určuje aktuální velikost částice. Částice je vždy čtvercová o délce strany size.
- unsigned int textureID
  Identifikátor textury, která se má na částici namapovat.

`GraphicParticle` rozšiřuje metodu `simulate()` o simulaci nově přidaných atributů. Dále potom přidává metodu `render()` . Ta zajišťuje vykreslení částice podle jejích parametrů do aktuálního OpenGL kontextu.

3.11.2 Správce částic

Emitor je objekt, který spravuje existující částice. Základ tvoří třída `BasicEmittor` . Ta obsahuje metody `simulate()` a `garbageRequest()` .

Metoda `simulate()` iteruje polem částic a na každou částici volá její metodu `simulate()` , kterou každá částice obsahuje. Metoda `garbageRequest()` odstraní staré, neaktivní částice.

Další metody jako například `emitNewParticles()` a jiné, je nutné dodefinovat v odvozené třídě v závislosti na konkrétních potřebách emitoru.

4 Vlastní hra

4.1 Hvězdná mapa

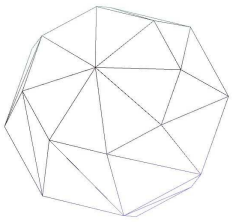
Hvězdná mapa zobrazuje vesmír v okolí aktivní lodě. Hvězdná mapa je rozdělena do dvou logických částí, režim makro a mikro mapy.

4.1.1 Mikro mapa

Zobrazuje detailní náhled na nejbližší okolí hráčovy lodě. V tomto režimu jsou vidět jednotlivé lodě a planety. Mikro mapa je spravována prostřednictvím třídy `MapFunctionality` . Ta ve spolupráci s třídou `SoftProjRender` zajišťuje renderování okolního vesmíru, jehož komponenty si v krátkosti popíšeme.

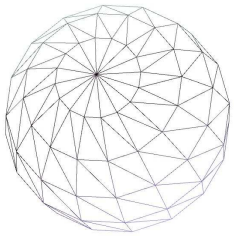
- Lodě
  Reprezentujeme je třídou `SpaceShip` . Ta obsahuje atributy potřebné pro zobrazování a simulaci lodi. Grafiku lodi zajišťuje objekt `Model` , jenž je součástí třídy.

- Planety
  Jsou reprezentovány třídou `BasicPlanet` . Třída obsahuje parametry potřebné pro zobrazování a simulaci. Geometrií planety je vždy koule a můžeme proto jednoduše generovat geometrii procedurálně. To nám umožňuje používat tzv. Progressive Level of Detail: V závislosti na vzdálenosti planety od pozorovatele měníme kvalitu geometrie [1]. Tím efektivně snižujeme počet renderovaných polygonů a tím šetříme čas GPU.

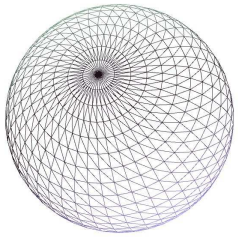


Obrázek 14: Hrubá aproximace koule 140 vrcholů

[1] Použité obrázky viz http://www.search.com/reference/Level_of_detail



Obrázek 15: Normální koule 670 vrcholů

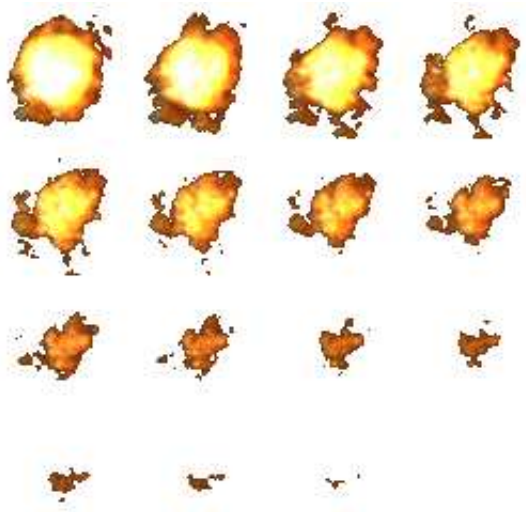


Obrázek 16: Detailní koule 5500 vrcholů

- Rakety
  Rakety jsou střely lítající mezi nepřátelskými loděmi během souboje. Reprezentuje je třída `SpaceMissile` . Podobně jako v případě lodí obsahuje třídu `Model` , která spravuje modely střel + atributy potřebné pro jejich fyzikální simulaci.

- Částice
  Lodě a rakety generují částice, ty byly obecně popsány v sekci Částicový systém. Pro účely aplikace bylo implementováno několik emitorů částic (pro každý typ lodi jiný), viz soubor `ShipEmittor.h` .

- Exploze
  Exploze jsou tvořeny sekvencí obrázků utvářející plynulou animaci výbuchu. Tyto animace realizuje třída `AnimationEffect` .



Obrázek 17: Série střídajících se obrázků tvořící animaci

4.1.2 Makro mapa

Zobrazuje prozkoumané části vesmíru. Objevené sluneční soustavy jsou symbolizovány 2D obrázkem, podobně jako lodě v okolí. Funkce makro mapy zajišťuje třída `RadarMapGame` .

4.2 Komunikace

Veškerá komunikace klienta s herním serverem probíhá přes třídy CommandClient a SocketClient. Vysokoúrovňovou komunikaci zajišťuje třída CommandClient, která je zodpovědná za posílání požadavků na server a přijímání odpovědí a aktualizací od serveru. Všechny tyto požadavky a odpovědi mají již formát herních dat a komunikace je pro klienta přes CommandClient transparentní v tom smyslu, že se nemusí starat o vytváření textových zpráv, které se ve skutečnosti na server posílají. Pro komunikaci směrem na server tedy CommandClient přijímá skutečná herní data a stará se o jejich zakódování do textového formátu, při komunikaci směrem od serveru naopak rozkódovává textové zprávy a plní herní struktury, které pak předává dále hlavní třídě MainWindow. CommandClient se tedy nikdy nestará o další akce s herní strukturou, o práci s GUI, nebo libovolné jiné rozhodování. Pouze provede dekódování a o další práci už se stará třída MainWindow. CommandClient pouze ví, jakou metodu MainWindow má volat pro který druh zprávy. Na jednu zprávu totiž může klient reagovat více způsoby, v závislosti na svém stavu, který se však v CommandClient nijak neukládá, o to vše se již stará třída MainWindow.

Veškeré požadavky klienta na server, které CommandClient přijímá, mají formu veřejných metod, protože je klient používá k vytváření svých požadavků na server. Na-

proti tomu všechny metody pro přijímání odpovědí od serveru jsou privátní, protože tyto metody klient nikdy nevolá sám. Metody pro odpovědi volá sám CommandClient vždy, když od něj klient požaduje další zprávu.

Jakou privátní metodu na dekodování zprávy má CommandClient zavolat se rozhoduje podle jejího jména. K tomu slouží třída HandlerWrapper, která mapuje jména zpráv na metody, které je dekódují. CommandClient vždy pracuje pouze v pevném intervalu, který je defaultně nastavený na 500ms a může se přenastavit metodou setTimeLimitForProcessing. Po dobu tohoto intervalu vždy vyzvedne jednu zprávu z fronty připravených zpráv, podívá se na její jméno a najde pomocí třídy HandlerWrapper odpovídající metodu na dekódování, která provede převod zprávy do herních struktur. Po dekódování zprávy CommandClient zavolá odpovídající metodu na MainWindow a předá ji dekodovaná herní data. Pokud stihne zpracovat zprávy ve frontě dřív, tak skončí aktuální zpracování až do dalšího intervalu.

Komunikace má tři hlavní vrstvy: Nejvyšší vrstvu CommandClient, která pracuje přímo s herními strukturami, střední vrstvu SocketClient, která pracuje s textovými zprávami a třídu client, která provádí skutečné zasílání a přijímání zpráv a zapouzdřuje práci se sockety. Důvod, proč je mezi třídou client a CommandClient ještě třída SocketClient, která nemá v porovnání s ostatními dvěma třídami zdaleka tolik práce je ten, že chceme maximální odstínění třídy pro práci se sockety a synchronizací a nejvyšší vrstvy, která pracuje s herními daty. Třída client díky tomu zná pouze třídu SocketClient, které zprávy předává a nebo je od ní přijímá.

4.2.1 Směr na server

CommandClient zasílá zprávy přes třídu SocketClient, která se stará o komunikaci na nižší úrovni. ClientCommand se tedy nestará o skutečné posílání zpráv, pouze zprávy zakóduje do textového formátu a předá třídě SocketClient. K tomu využívá třídu protocolWriter, která zapisuje herní struktury do textové formy zprávy, vždy jako název parametru a jeho hodnotu. ProtocolWriter však nezapisuje herní struktury jako celek, protože dokáže zapisovat jenom primitivní datové typy. Zakódování herní struktury tedy provádí CommandClient, který bere jednotlivé primitivní datové položky herní struktury a předává je třídě protocolWriter, která provede skutečné zapsání. Pokud herní struktura obsahuje složené datové typy, zapisují se opět pouze jejich primitivní datové položky. Pole a kolekce se zapisují pomocí přidávání indexů k parametrům, pro vnořené třídy se používá prefix a následně se vypisují její primitivní datové typy.

Pro zapsání zprávy přes protocolWriter využívá CommandClient především funkci beginMessage, která zahájí tvorbu zprávy a které jako typ zprávy vždy předá hodnotu messageType::MT_REQUEST, protože se jedná o požadavek na server. Dále jí předá jméno zprávy, které reprezentuje jméno požadavku na server a také se shoduje s názvem příslušné odpovědi. Následně zapisuje jednotlivé položky herních struktur pomocí metody writeValue, která vždy dostane název parametru a jeho hodnotu. Po

skončení zápisu zavolá metodu endMessage a dá jí vždy příznak true, což znamená, že se jedná o nechybovou zprávu.

Ve skutečnosti nepoužívá CommandClient protocolWriter přímo, ale používá ještě nad ním třídu requestWriter, která pro každý požadavek obsahuje vlastní metodu, která provede skutečné zakódování do textové podoby. Díky tomu je CommandClient přehlednější, protože se stará převážně o vysokoúrovňové kódování zpráv a komunikaci s MainWindow, zatímco samotné kódování provádí třída pod ním.

Vytvořenou zprávu následně CommandClient zašle na server pomocí nižší třídy SocketClient. SocketClient pouze předá zprávu nižší třídě client a uvolní alokovanou paměť pro vytvořenou zprávu.

4.2.2 Směr od serveru

Veškeré metody, které představují metody na dekódování odpovědí od serveru, mají prefix process a zbytek jejich jména se vždy shoduje s daným požadavkem na server, pokud jim nějaký odpovídá. Ne všechny zprávy od serveru totiž musí být posílány jako reakce na klienta, například serverová oznámení, že si klient má aktualizovat aktuální stav zobrazené sekce. Metody vždy přijímají ukazatel na objekt třídy message, která představuje prostřední vrstvu mezi textovým formátem a herními strukturami. Jedná se třídy obsahující mapu, která uchovává textové dvojice název – hodnota. CommandClient čte tyto zprávy a plní z nich herní struktury. Vytvoření zprávy v tomto meziformátu má na starosti třída protocolReader, která je vytváří přímo z textové podoby zprávy. CommandClient tedy nečte skutečnou textovou zprávu, ale využívá metodu translate třídy protocolWriter, aby získal rovnou meziformát, ze kterého pak pouze čte jako z mapy hodnot.

Ke čtení meziformátu využívá CommandClient třídy ValueReader. To je užitečné především k tomu, aby nemusel CommandClient opakovaně konvertovat řetězce, které jsou v mapě uložené, na skutečný datový typ, který má odpovídající parametr mít. Zároveň poskytuje metody pro čtení indexovaných hodnot, takže CommandClient zadá pouze název parametru a index a ValueReader už provede převod na skutečné jméno parametru včetně zahrnutí indexu ve standardním tvaru, čímž zamezuje vzniku chyb, který by jinak mohl vznikout při opakování stejného kódu. ValueReader se vždy vytváří na zásobníku a přijímá zprávy v meziformátu. CommandClient pak na něm volá přístupové metody, které ValueReader pouze deleguje na meziformát k získání textové hodnoty parametru a následně provede přetypování a vrátí skutečnou hodnotu.

Zprávy přijímá CommandClient od nižší vrstvy, třídy SocketClient, která využívá třídu client, která pouze zapouzdřuje práci se sockety a představuje tedy nejnižší vrstvu komunikace. Má na starosti synchronizaci při práci se sockety a odstiňuje od ní vyšší vrstvy. K tomu využívá dvě vlákna, jedno pro přijímání zpráv ze socketu a druhé pro posílání zpráv do socketu. Při odesílání zprávy jí třída CommandClient pouze předá zprávu v textové podobě pomocí metody send. O její odeslání se už

se postará třída client, která ji nemusí odeslat hned, ale může si ji nejprve zařadit do fronty. Stejně při přijmu zprávy třída client zavolá metodu receiveMessage třídy SocketClient a tím ji upozorní, že dorazila nová zpráva. Čekání na socketu na příjem zprávy tedy provádí pouze třída client a vyšší vrstvy jsou na zprávu upozorňovány v momentě, kdy nějaká dorazí.

Při přijmu zprávy tedy volá client metodu receiveMessage třídy socketClient. Předtím ji však rozkoduje do meziformátu a třídě SocketClient předá rovnou tento meziformát. SocketClient si vzniklou zprávu pouze zařadí do fronty zpráv a dál se zprávou nic nedělá. Pro správné zařazení do fronty musí na frontu nejprve získat mutex zámek, aby nedošlo k porušení synchronozace. Ve chvíli, kdy CommandClient začne pracovat v rámci svého pracovního intervalu čtení zpráv, řekne si sám třídě SocketClient o další zprávu ve frontě. V tuto chvíli opět musí nejprve získat mutex zámek na frontu.

Následně se CommandClient podívá na název požadavku a najde odpovídající metodu na rozkodování do herní struktury. Narozdíl od kódování, kde využíval třídu requestWriter, zde provádí dekódování přímo sama třída CommandClient.

Po dekódování volá metodu třídy MainWindow, která pak dále s herní strukturou pracuje.

Kroky komunikace při aktualizaci dat:

1. Uživatel zadá příkaz na aktualizaci dat nějakého widgetu, například kliknutím myší na nějaké tlačítko, nebo zadá příkaz na některou herní akci, která se musí zaslat na server, například chce postavit nový předmět v hangárech.

2. Příslušný widget vygeneruje signál, kterým signalizuje žádost o aktualizaci dat a v němž zároveň předá příslušné datové struktury, které jsou k požadavku potřebné.

3. MainWindow zachytí vygenerovaný signál do slotu, který obsluhuje daný signál.

4. MainWindow zvolí a zavolá metodu pro daný požadavek na třídě CommandClient a předá jí herní strukturu, která se k požadavku vztahuje.

5. CommandClient zavolá metodu pro daný požadavek na třídě requestWriter a předá jí herní datovou strukturu, kterou chce zakódovat.

6. RequestWriter zakóduje požadavek do textové podoby a výsledek vrátí třídě CommandClient. Pokud požadavek obsahuje nějaká data pro server, provede také jejich zakodování.

7. CommandClient zavolá metodu sendAndFreeMessage třídy SocketClient a předá ji vytvořenou textovou zprávu.

8. SocketClient zavolá metodu send třídy client, který odešle zprávu přes socket na server. Následně odstraní zadanou zprávu z paměti

Kroky komunikace směrem od serveru:

1. Server zašle klientovi textovou zprávu, například pro aktualizaci dat některé herní sekce.
2. Třída client stále čeká ve svém vlastním pracovním vlákně na socketu. Při příjmu zprávy ze socketu ji dekóduje pomocí třídy protocolReader a získá tak meziformát zprávy s mapou hodnot.
3. Třída client zavolá metodu acceptMessage na třídě SocketClient, která si zprávu zařadí do fronty zpráv.
4. CommandClient v rámci svého pracovního intervalu vyzvedne zprávu z fronty a podle jejího jména nalezne odpovídající metodu pro zpracování pomocí třídy HandlerWrapper.
5. CommandClient dekóduje zprávy z meziformátu do herní struktury tak, že čte jednotlivé parametry z mapy hodnot zprávy.
6. CommandClient zavolá pevně danou metodu třídy MainWindow, určenou pro zpracování dané zprávy a předá jí vyplněnou herní strukturu.

4.3 GUI

Hlavní třídou GUI je třída MainWindow, která zastřešuje veškeré klientské widgety, koordinuje jejich práci a zajišťuje pro ně veškerou komunikaci se serverem. Žádný widget totiž nemůže komunikovat se serverem přímo, ale musí vždy požadavky posílat přes třídu MainWindow. Stejně tak odpovědi dostává pouze od třídy MainWindow, což je užitečné zejména proto, že pouze MainWndow udržuje herní stav klienta a může proto rozhodnout, jak s obdrženou zprávou od serveru naložit.

Například při příjmu zprávy se seznamem lodí záleží na herním stavu klienta, jak s přijatými daty naloží. Například pokud je ve stavu obchodování, předá seznam widgetu pro obchodování, pokud je ve stavu bojování, předá data widgetu pro bojování. Také může přijadá data zcela odmítnout a zahodit, pokud je v takovém stavu, ve kterém daná data nepožaduje. To se může stát, pokud například zažádá o aktualizační data v jednom widgetu, ale ještě než dostane od serveru odpověď, přepne se hráč do jiného widgetu. V takovém případě jsou přijatá data zbytečná a zahodí se.

Pro widgety probíhá komunikace se serverem většinou asynchronně, což znamená, že widgety zašlou požadavek přes MainWindow a na odpověď už přímo nečekají. Při příjmu zprávy jim pak MainWindow předá aktualizační data a widgety se podle nich aktualizují. Ve většině případů si proto widgety žádný stav uchovávat nemusí a pouze na zprávy od serveru reagují a podle nich se aktualizují. V některých případech to však je přesto nutné – například při obchodování, nebo bojování.

Pro zpracování odpovědí od serveru používá MainWindow privátní metody, které

jsou označeny prefixem answer a zbytek jména odpovídá názvu příslušného požadavku. Díky tomu CommandClient ví, jakou metodu má zavolat, když naplní herní data ze zprávy od serveru.

Mnoho widgetů reprezentuje celou herní sekci, například sekci lodních štítů, zbraní, nebo větší herní sekce jako obchodování, bojování, nebo budování. Takové widgety se skládají často z dalších podwidgetů. Budeme je dále nazývat hlavní herní widgety, protože jsou mnohem komplexnější než různé menší podpůrné widgety.

Všechny takové widgety se aktualizují vždy, když uživatel vstoupí do nějaké herní sekce, kterou widget reprezentuje. Díky tomu jsou data vždy poměrně aktuální. Většina widgetů používá na svou aktualizaci jen jeden požadavek na server, který aktualizuje rovnou celý widget. Některé větší widgety, jako například obchodování, budování nebo bojování, zasílají požadavků více, protože obsahují ještě další vnořené widgety, které se mouhou aktualizovat nezávisle na ostatních widgetech, nebo proto, že není možné kvůli velikosti zasílat rovnou celý možný stav widgetu – například při souboji, kde chceme znát všechny hráčovy lodě a zároveň pro každou jeho loď také lodě v dosahu střelby. V takovém případě by bylo zbytečně náročné posílat pro všechny hráče všechny okolní lodě, protože hráč má stejně vždy vybranou jen jednu aktivní loď. Navíc by taková obrovská data rychle ztrácela na aktuálnosti. Proto se data o lodích v okolí zasílají až pro aktivní vybranou loď.

Pro aktualizaci využívají jednotlivé widgety, které reprezentují hlavní herní sekce, téměř vždy jednu aktualizační strukturu, ve které dostávají potřebná data od serveru. Podle ní aktualizují svůj stav a strukturu si uloží do té doby, než provedou další aktualizaci, což je užitečné k tomu, že mohou provádět průběžnou aktualizaci některých údajů, které se časem mění poměrně pravidělně, aniž by museli pořád se serverem komunikovat – stačí když si pamatují rychlost změny a původní stav a z něj již mohou dopočítat nový stav. To platí například u počtu surovin, nabíjení energií do některých zařízení, nebo při stavbě předmětu či lodě v hangáru, kde chceme zobrazovat postup výroby.

Část II

Serverová část

5 Úvod

Divergent Teritory je online multiplayerová hra určená pro velké množství hráčů. Dělí se na serverovou část a klientskou část, jako samostatný prvek klientské části je administrátorské rozhraní, které je určitým speciálním klientem. Server používá k uchovávání dat databázi Firebird. Klient používá knihovnu Qt a knihovnu OpenGL pro zobrazování grafiky.

5.1 Hardwarové požadavky serveru

2 GB RAM
4 jádrový procesor
DVD mechanika
Windows 2000 a vyšší
Firebird, verze 2.0.6 a vyšší
disk 50 MB instalace, cca 2 GB herní databáze

5.2 Instalace

6 Architektura serveru

Server se skládá ze dvou částí: herního enginu a vlastní hry. Herní engine zajišťuje servis herní logice a herním datovým strukturám. Tento engine lze použít i pro další hry tohoto typu.

V současné době engine umožňuje spustit jednu instanci hry na každý proces, nicméně lze ho snadno upravit tak, že bude možné spouštět více instancí hry v jednom procesu.

Protože je Divergent Territory massive multiplayer online hra, bylo nutné navrhnout architekturu serveru tak, aby velký počet zaregistrovaných či online hráčů nepředstavoval výkonostní problém.

6.1 Běh herních mechanismů

Téměř všechny herní mechanismy probíhají (poté, co je uživatel nastaví) automaticky a v reálném čase; server tedy musí simulovat mnoho paralelních dějů zároveň. Implementačně nejjednodušší by sice byla spojitá simulace, která by ale jednak příliš zatěžovala server při několika stovkách či tisích registrovaných hráčů a jednak by vytěžovala linku mezi serverem a aktuálně nalogovanými hráči, protože by jim bylo každý krok třeba posílat. Všechny děje jsou tedy simulovány tak, že si server drží stav v nějakém čase, gradient změny stavu a spočítaný čas příští změny. Když tento čas nastane, server znovu přepočítá stav, nový gradient a určí čas následující změny a tak dále. Tento způsob sice umožňuje serveru zvládnout potenciálně velké množství hráčů, avšak implementačně není úplně triviální v případě, že jednotlivé děje na sobě (někdy i cyklicky) závisí.

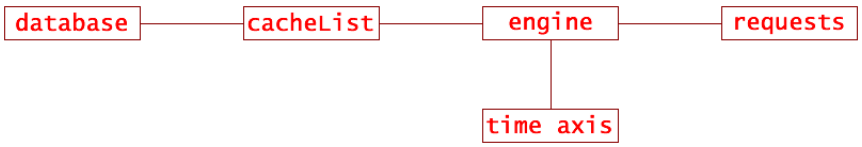
6.2 Práce s daty

Další věc, pro kterou bylo nutné architekturu serveru navrhnout, je rychlá práce s velkými objemy dat mnoha typů; některá data se často mění (základny a lodě hráčů, předměty), jiná se mění málo (uživatelské účty, výzkum), další jsou v rámci hry konstantní (hvězdy, planety). Pro tento účel používá server tzv. cacheListy, což jsou objekty, které spravují všechna data. Každý typ dat má svůj cacheList (pevnosti, hráči, předměty, vesmírné objekty, ...). Kdykoliv herní engine potřebuje nějaký objekt, předá odpovídajícímu cacheListu ID objektu (to je jednoznačné a používá se i v databázi), aby jej cacheList mohl vrátit (pokud neexistuje, vrací NULL). Uvnitř mohou cacheListy fungovat různě (jsou parametrizovatelné). Nejčastěji při startu serveru načtou všechna data z databáze do paměti, kde je uloží do vektoru, čímž je zajištěn přístup v lineárním čase. Lze je ale nastavit (případně velmi snadno přeprogramovat) na mnoho dalších typů chování, např.: data se načítají přímo z databáze a po použití se uvolní; do paměti se načítají data, která by mohla být pravděpodobně potřebná záhy (načtení pevností a předmětů hráče při jeho přihlášení); data se načtou do paměti ve chvíli, kdy jsou poprvé potřeba, pak tam zůstanou do vypnutí serveru; data se načtou, když je jich třeba, z listu se uvolní, když začne docházet paměť. Koherenci dat v paměti a databázi mohou listy též řešit mnoha způsoby: po každém použití dat je rovnou propsat; jednou za čas zapsat všechna data z listu; zapisovat data z uvolňovaných objektů; jednou za čas zapsat pouze změněná data; nezapisovat do databáze nikdy (to se hodí např. u vesmírných objektů, které se nemění, jen se z nich čte).

6.3 Řízení běhu

Celý běh serveru je řízen událostmi ze dvou front. První je časová osa, druhou tvoří requesty uživatelů. Časová osa obsahuje události, které do ní přidá server, když spočítá, že někdy v budoucnu dojde ke změně nějakého objektu (na osu přidá event s

odpovídajícím časem a ID objektu). Nabíjí-li například loď energii do štítů a server zjistí, že za minutu budou štíty plně nabité, přidá na časovou osu událost s časem "nyní + 1 minuta", kterou za minutu vyzvedne a přenastaví rozvod energie v lodi, štíty se přestanou nabíjet (změní-li se něco, co událost přiblíží či oddálí, je tato událost na časové ose přemístěna). Požadavky hráčů (z klientské části aplikace) předvídat nelze, proto se vyřizují v okamžiku, kdy přijdou. Vlákno serveru tedy stále čeká, dokud buď nepřijde požadavek po síti, nebo dokud nepřišel správný čas pro aktivaci události na časové ose. Obrázek znázorňuje, jak jsou spolu propojené jednotlivé části serveru. Engine je řízen událostmi a požadavky, data získává z cacheListů, které komunikují s databází.



Obrázek 18: Schéma architektury serveru.

6.4 Více světů

Server je navržen tak, aby na něm mohlo v jednu chvíli běžet více herních světů zároveň; každý by pak měl svou databázi a port, na kterém by poslouchal. Všechny cacheListy i časová osa patří do tříd kontextu. Aktuální kontext je schován pod ukazatelem actx, do kterého se při každém přijetí události z časové osy nebo požadavku klienta přiřadí kontext, do kterého tento klient či událost patří. Krom (neurčeného počtu) aktivních kontextů je na serveru ještě kontext globální, který obsahuje síťový server, a další věci společné pro všechny kontexty. Svět je na provozování více světů naráz sice připraven a navržen, ale nikoliv otestován; je možné, že by pro zprovoznění této možnosti bylo třeba několik dalších zásahů.

6.5 Paralelismus

Zpracovávání požadavků a tvorba odpovědí, tedy operace s řetězci s vysokou časovou náročností, probíhá plně paralelně, neboť se jedná o operace, které se nijak neovlivňují a jejich paralelní zpracování může server výrazně zrychlit. Aritmeticko-logické výpočty herní logiky či fyziky, které jsou méně náročné a jejich paralelizace by byla velmi složitá, probíhají v jednom vlákně. V případě více herních světů by běžela tvorba zpráv opět ve velkém počtu vláken a aritmeticko-logické výpočty v jednom vláknu pro každý svět. Architektura serveru potenciálně umožňuje ukládání dat z cacheListů do databáze ve zvláštním vlákně, čímž by bylo možné výkon ještě navýšit (nicméně takové vlákno zatím implementováno není).

6.6 Protokol

Server komunikuje s klienty asynchronně a bezstavově přes sockety. Chce-li klient data, vygeneruje žádost, server mu pak zpět pošle data, což není nijak odlišeno od toho, kdy na serveru dojde ke změně, o které je třeba informovat klienta. Klient přijímá od serveru data, která buď zahazuje (pokud se hráč zrovna dívá na jinou obrazovku), nebo zobrazí (pokud jsou zrovna relevantní); z tohoto pravidla existuje několik výjimek (kvůli optimalizaci). Klient nerozlišuje, zda si data vyžádal, nebo mu byla poslána jen tak.

6.7 Čas

Vzhledem k tomu, že je na serveru implementována relativně komplexní fyzika (gravitace, působení sil, zrychlování objektů, některé důsledky Keplerových zákonů, ...), je nepraktické, aby herní čas běžel rychlostí skutečného času (trvalo by moc dlouho, než by se ve hře cokoliv stalo); běží tedy rychleji (defaultně 30x rychleji, což lze změnit přepsáním jediné konstanty). Herní čas se vždy po spuštění serveru namapuje na reálný, k čemuž stačí dvojice odpovídajících časů a poměr rychlosti plynutí času ve hře a v reálném světě. Vzhledem k tomu, že je toto mapování velmi rychlé, je při vypnutí serveru (či náročnějších operacích jako je třeba generování většího vesmíru) snadné herní čas zastavit. Posunutí času vpřed nezpůsobuje větší problémy až na to, že server posunutý do budoucnosti začne zpracovávat všechny události na časové ose, které už se staly, avšak nebyly zpracovány; bude i přidávat nové, z těchto událostí vyplývající, které pak zpracuje také, až na ně přijde řada, čímž se dostane přibližně (až na nějaké detaily) do takového stavu, v jakém by byl, kdyby čas uplynul standardní cestou (každopádně bude výsledný stav konzistentní). Posunutí času zpět je nemožné; server se nejen dostane do velmi nekonzistentního stavu, ale navíc přestane fungovat, dokud se čas nevrátí přibližně do bodu, odkud byl vrácen (nicméně rozhodně plynule nenaváže - pro každý objekt ve hře začne čas plynout jindy, což způsobí další nekonzistence). Toto se týká pouze vracení herního času; návrat reálného času (například přechod z letního na zimní čas) nevadí, je-li herní čas odpovídajícím způsobem přemapován.

6.8 Sítě a vlákna

Nejnižší vrstva enginu hry zajišťuje abstrakci vláken, socketů a synchronizačních primitiv nad systémovými funkcemi. Umožňuje tak přizpůsobivost kódu v případě, že se Windows API změní, nebo v případě, že hra bude migrována na jiný operační systém. Abstrakce umožňuje používat obdobné objekty reprezentující jednotlivá vlákna, sockety a synchronizační primitiva, jaké jsou k dispozici například v Javě či C#.

6.8.1 Vlákna

Abstrakce vláken je zajištěna objektem `thread` . Do toho objektu přidáme potomka třídy `runnable` , kam dopíšeme příslušnou metodu operátoru, provádějící námi požadovanou funkčnost. Vlákno můžeme spustit hned, jak ho vytvoříme s pomocí konstruktoru (v tom případě nastavíme proměnnou `suspended` na hodnotu false). V opačném případě ho můžeme spustit později metodou `start` . Metoda `isAlive` indikuje, zda je vlákno spuštěné, nebo ne. Na každé spuštěné vlákno vždy (někdy v budoucnosti) musíme zavolat úspěšně metodu `join` a tím vlákno zničit a uvolnit jeho prostředky. Metody `resume` a `suspend` spouštějí resp. pozastavují vlákno, ale důrazně se doporučuje je nepoužívat, protože mohou vlákno pozastavit v jakémkoliv okamžiku, například když provádí zápis do souboru. Z tohoto důvodu je vhodnější používat k zastavování resp. opětovného spuštění vláken synchronizační primitiva a funkci `sleepCurrentThread` , která uspí vlákno na určitý čas.

```
class runnable
{
public:
        virtual void operator()(thread* thd) = 0;
};

class thread
{
public:
        thread(runnable* runObj, bool suspended);
        void start(void);
        void join(void);
        bool join(DWORD miliseconds);
        void resume(void);
        void suspend(void);
        bool isAlive(void);
private:
        HANDLE threadHandle_;
};
```

6.8.2 Zámky

Abstrakce zámků je zajištěna objektem `mutex` . Mutex je nejjednodušší synchronizační primitivum pro vzájemné vyloučení. Funkce `lock` zamkne kritickou sekci. Funkce `unlock` ji odemkne. Varianta funkce `lock` umožňuje zkusit kritickou sekci uzamknout a pokud se to nezdaří do určité doby, vrátí hodnotu false, pokud se povede, vrátí hodnotu true.

```
class mutex
{
```

```
private:
        HANDLE mutexHandle_;
public:
        mutex();
        ~mutex();
        void lock(void);
        bool lock(DWORD miliseconds);
        void unlock(void);
};
```

6.8.3 Semafory

Semafory jsou v jistém smyslu zobecnění zámků. Každý semafor má číselnou proměnnou. Pokud je tato proměnná kladná, zavoláním funkce `down` se odečte hodnota 1 a proces vejde do kritické sekce. V opačném případě se volání zablokuje. Funkce `up` přičte k proměnné hodnotu 1 a tím odblokuje případné procesy čekající na semaforu. Funkce `up` existuje i v časově omezené variantě - je možné zadat časový interval, po který se čeká na vstup do kritické sekce, pokud v tomto intervalu se podaří procesu vstoupit do kritické sekce, tak vrátí funkce hodnotu true, v opačném případě false.

```
class semaphore
{
private:
        HANDLE semaphoreHandle_;
public:
        semaphore(int initcnt, int maxcnt);
        ~semaphore();
        void up(void);
        void down(void);
        bool down(DWORD miliseconds);
};
```

6.8.4 Sockety

Sockety se dělí na "klientský" a "serverový". Zatímco serverový pouze přijímá spojení, klientský slouží k oboustranné komunikaci. Klientský socket má metody `sendData` a `recvData`, které slouží posílání a přijímání dat. Pokud nastane při přenosu chyba, je vyhozena výjimka. Klient se umí připojit k TCP/IP serveru pomocí metody `connectToHost`, kde se zadá jméno hosta a port. Při úspěchu vrátí hodnotu true, false jinak. K uzavření spojení slouží metoda `close`.

```
class clientSocket
{
```

```
private:
        SOCKET socketHandle_;
public:
        clientSocket(void);
        clientSocket(SOCKET socketHandle);
        bool connectToHost
                (const char* hostName, int port);
        int sendData(const char* data, int size);
        int recvData(char* buf, int size);
        void close(void);
};
```

Serverový socket má metodu `listenPort` , která slouží k poslouchání na daném portu, tj. čekání na příchozí spojení. K přijetí spojení slouží metoda `acceptConnection` , která je blokující. K uzavření socketu (a ukončení poslouchání na daném portu) slouží metoda `close` .

```
class serverSocket
{
private:
        SOCKET socketHandle_;
public:
        serverSocket(void);
        void listenPort(int port);
        void listenPort
                (int port, int maxConnections);
        clientSocket* acceptConnection(void);
        void close(void);
};
```

Uvedená implementace je nízkoúrovňová, je možné například clientSocket použít k připojení na TCP/IP server, který nepochází z tohoto projektu. Před použitím těchto objektů je nutné zavolat inicializační funkci `socketsStart` .

6.9 Funkce datum/čas

Herní engine používá svou vlastní reprezentaci datumu/času tak, aby bylo možné jednotlivá data sčítat/odečítat a šly s nimi provádět různé další operace. Datum/-čas je reprezentován v plovoucí řádové čárce datovým typem double; konstanta 1.0 odpovídá jednomu dni. Třída `datetime` obsahuje mnoho funkcí na přidávání/odečítání času, dále časoměrné funkce a také operátory, takže časy lze standardně sčítat operátorem +. Metoda `toString` převede datum/čas na textový řetězec.

```
class datetime
{
```

```
private:
        double datetime_;
public:
        datetime(void);
        datetime(double datetime);

        datetime& operator=(const datetime& other);

        const datetime operator+
                (const datetime& other) const;
        const datetime operator-
                (const datetime& other) const;

        const datetime operator*
                (const double other) const;
        const datetime operator*
                (const unsigned long long other) const;
        const double operator/
                (const datetime& other) const;
        const datetime operator/
                (const double other) const;
        const datetime operator/
                (const unsigned long long other) const;

        datetime& operator+=(const datetime& other);
        datetime& operator-=(const datetime& other);
        datetime& operator*=(const datetime& other);

        bool operator==(const datetime& other) const;
        bool operator!=(const datetime& other) const;

        bool operator<(const datetime& other) const;
        bool operator>(const datetime& other) const;
        bool operator<=(const datetime& other) const;
        bool operator>=(const datetime& other) const;

        operator double() const;

        static const datetime nanosecond(void);
        static const datetime microsecond(void);
        static const datetime millisecond(void);
        static const datetime second(void);
        static const datetime minute(void);
        static const datetime hour(void);
        static const datetime day(void);
```

```
        static const datetime month(void);
        static const datetime year(void);

        datetime& incYear(int years);
        datetime& incMonth(int months);
        datetime& incDay(int days);
        datetime& incHour(int hours);
        datetime& incMinute(int minutes);
        datetime& incSecond(int seconds);
        datetime& incMillisecond(int milliseconds);

        datetime& decYear(int years);
        datetime& decMonth(int months);
        datetime& decDay(int days);
        datetime& decHour(int hours);
        datetime& decMinute(int minutes);
        datetime& decSecond(int seconds);
        datetime& decMillisecond(int milliseconds);

    std :: string toString(void);
};
```

Dále jsou k dispozici různé funkce pro kódování/dekódování času.

```
datetime encodeDate(int year, int month, int day);
datetime encodeTime(int hours, int minutes,
        int seconds, int milliseconds);
datetime encodeDateTime(int year, int month, int day,
        int hours, int minutes,
        int seconds, int milliseconds);
datetime encodeSeconds(unsigned long long seconds);

void decodeTime(const datetime& t,
        int& hours, int& minutes,
        int& seconds, int& milliseconds);
void decodeDate(const datetime& t, int& year,
        int& month, int& day);
void decodeDateTime(const datetime& t, int& year,
        int& month, int& day,
        int& hours, int& minutes,
        int& seconds, int& milliseconds);
void decodeTime(const datetime* t,
        int* hours, int* minutes,
        int* seconds, int* milliseconds);
void decodeDate(const datetime* t, int* year,
        int* month, int* day);
```

```
void decodeDateTime(const datetime* t, int* year,
        int* month, int* day,
        int* hours, int* minutes,
        int* seconds, int* milliseconds);

bool isLeapYear(int year);
```

Existují také funkce vracející určitý počet časových jednotek mezi dvěma daty. Funkce `datediff` spočítá časový rozdíl mezi dvěma daty.

```
datetime datediff(const datetime& d1, const datetime& d2);

long long yearsBetween
        (const datetime& d1, const datetime& d2);
long long monthsBetween
        (const datetime& d1, const datetime& d2);
long long daysBetween
        (const datetime& d1, const datetime& d2);
long long hoursBetween
        (const datetime& d1, const datetime& d2);
long long minutesBetween
        (const datetime& d1, const datetime& d2);
long long secondsBetween
        (const datetime& d1, const datetime& d2);
long long millisecondsBetween
    (const datetime& d1, const datetime& d2);

long long yearsBegunBetween
        (const datetime& d1, const datetime& d2);
long long monthsBegunBetween
        (const datetime& d1, const datetime& d2);
long long daysBegunBetween
        (const datetime& d1, const datetime& d2);
long long hoursBegunBetween
        (const datetime& d1, const datetime& d2);
long long minutesBegunBetween
        (const datetime& d1, const datetime& d2);
long long secondsBegunBetween
        (const datetime& d1, const datetime& d2);
long long millisecondsBegunBetween
    (const datetime& d1, const datetime& d2);
```

Nakonec funkce `now` vrací aktuální čas.

```
datetime now(void);
```

6.10 Databázové připojení

Databázové připojení využívá knihovny IBPP[2] k připojení k databázovému serveru Firebird. Metoda `connect` se pokusí připojit k databázovému serveru se zadanými parametry, pokud se to nezdaří, vrátí hodnotu false, jinak true. Metoda `commit` potvrdí aktuální transakci, tj. všechny změny databáze provedené od posledního zavolání metody `commit`, případně `connect`, pokud byla databáze právě připojena. Metoda `disconnect` pak odpojí databázi.

```
class dbClient
{
private:
        IBPP :: Database db_;
        IBPP :: Transaction tr_;

public:
        bool connect(const std :: string& server,
                const std :: string& database,
                const std :: string& username,
                const std :: string& password);
        void commit(void);
        void disconnect(void);
};
```

Do této třídy pak lze dopisovat jednotlivé dotazy pro tu určitou hru, která je právě vyvíjena, popřípadě lze napsat potomka této třídy. Tato komponenta je stále ještě nízkoúrovňová, k abstrakci a správě databázových objektů slouží objekt "cache list" popsaný v následující sekci.

6.11 Cache objektů

Tato komponenta představuje abstrakci nad databází, dokáže spravovat databázové záznamy tak, jako by to byly objekty v operační paměti. S její pomocí lze záznamy ukládat, vytvářet, mazat a načítat. Jelikož je to abstrakce nad databází, lze ji specializovat a například některé zdroje dat vyčlenit - například statická data lze spravovat jako speciální cache list, který vůbec nespolupracuje s databází, ale má záznamy uložené přímo v kódu.

Jako základní objekt abstrakce nad záznamy v databázi můžeme považovat třídu `dataListObjectClass`. Jednotlivé specializované záznamy pak od této třídy dědí a implementují její virtuální funkce. Tento objekt má také položku `ID_`, která jednoznačně určuje daný záznam v databázi.

```
class dataListObjectClass
```

[2] www.ibpp.org

```
{
protected:
        dtIdentifier    ID_;
public:
    dataListObjectClass();
    virtual         ~dataListObjectClass();

    void            setID     (unsigned long ID);
    unsigned long   getID     () const;

    virtual void    DB_Load   (  unsigned long ID);
    virtual void    DB_Save   (  unsigned long ID);
    virtual void    DB_Insert(   unsigned long ID);

    virtual void    DB_TLoad        ();
    virtual void    DB_TSave        ();
    virtual void    DB_TInsert      ();

    virtual void    UpdateEvents    ();
};
```

K abstrakci tabulky slouží třída `cacheListClass` . Tato třída zajišťuje správu databázových záznamů a jejich reprezentaci v operační paměti. Uživatel (programátor herní logiky) se tak nemusí starat o správu objektů.

```
class cacheListClass
{
public:
    virtual ~cacheListClass();

    virtual dataListObjectClass * operator []
                (unsigned long position          );

    virtual dtIdentifier            insertItem
                (dataListObjectClass * newItem);
    virtual void                    deleteItem
                (dtIdentifier          id      );
    virtual dataListObjectClass *   access
                (dtIdentifier          id      );
    virtual void                    synchronize
                (void                          );

    virtual void  DB_Initialize
                (         void                 );

    template<typename T>
```

```
    T* getItem (unsigned long position);
    void           getObjectIds
                  (std :: deque<dtIdentifier>& ids );
    virtual void  updateEvents
                  (void                          );

        size_t getSize();
};
```

Každý záznam má své jednoznačné ID. Pomocí hranatého operátoru lze získat z cache listu příslušný záznam, případně hodnotu NULL, pokud daný záznam neexistuje. Obdobnou, resp. stejnou funkcionalitu má funkce `access` . K přidávání prvků slouží metoda `insertItem` , v okamžiku volání příslušná instance cache listu přebírá zodpovědnost za tento objekt, tj. v případě potřeby ho dealokuje. K mazání prvků slouží metoda `deleteItem` , opět, smazaný objekt není po té nutné dealokovat, neboť to udělá cache list sám. Aby se usnadnilo přetypování a také pro typovou čistotu, šablonová funkce `getItem` vrací objekt příslušného typu přetypovaný pomocí `dynamic_cast` . Funkce `getObjectIds` dokáže zjistit ID všech objektů, které jsou v dané tabulce reprezentované cache listem. Virtuální metoda `updateEvents` slouží k inicializaci objektů při prvním načtení, může, nebo také nemusí být využita. Nakonec funkce `getSize` vrátí potenciální počet prvků v cache listu a metoda `DB_Initialize` inicializuje cache list před prvním použitím (synchronizuje ho s databází).

Cache list však změny nezapíše okamžitě do databáze. Příslušné změny v objektech, jejich smazání a přidávání si pamatuje a k zápisu aktuálního stavu do databáze slouží metoda `synchronize` . Je to tak lepší proto, že server pak běží mnohem rychleji, neboť nepotřebuje neustále zapisovat změny v databázi na disk. Tento přístup je horší pro bezpečnost dat, protože v databázi jsou uložena starší data a při pádu serveru se tedy obnoví pouze starší data. To ale u online her nevadí, neboť je časté, že se třeba vezme záloha z minulého dne. Více by to vadilo například v nějakém platebním systému. Vnitřně má každý záznam u sebe údaj, který určuje platnost dat jak v databázi, tak v operační paměti. Záznam může mít u sebe následující příznak:

- `DATA_IS_EMPTY` : Data nejsou ani v databázi, ani v paměti, záznam neexistuje.
- `DATA_IN_DATABASE_ONLY` : Platný záznam je pouze v databázi, bude nutné ho načíst.
- `DATA_IN_MEMORY_ONLY` : Platný záznam je pouze v paměti, bude nutné ho vložit do databáze.
- `DATA_IN_DATABASE_INVALID` : Záznam je neplatný a je v databázi, je ho z ní nutné smazat.
- `DATA_SYNCHRONIZED` : Data záznamu jsou jak v databázi, tak v operační paměti, jsou oboje platná a identická.

- `DATA_DESYNCHRONIZED` : Platný záznam je v operační paměti, v databázi jsou zastaralá data, která budou muset být aktualizována.

6.12 Kontext hry

Kontext hry obsahuje všechny herní datové struktury a další položky nutné pro běh enginu. Z důvodu potenciální možnosti mít spuštěné více instancí hry najednou, je kontext rozdělen na globální část `global_context` a lokální část `context` . Zatímco globální část je společná pro všechny instance hry, každá instance hry má svůj lokální kontext. Engine hry si sám nastavuje příslušný kontext podle dané instance hry. Lokální kontext je možné nastavit pomocí funkce `setContextById` , zjistit identifikátor nastaveného lokálního kontextu lze funkcí `getActualContextID` a konečně získat ukazatel na lokální kontext (podle daného identifikátoru) lze funkcí `getContextById` . V současné podobě lze mít spuštěnu pouze jednu instanci hry, příslušný lokální kontext je v proměnné `actx` , globální kontext je pak v proměnné `gctx` .

```
class global_context
{
public:
    /* server */
    server srv;

    /* engine hry */
    engine eng;

    void closeClient(id clientID);
    void sendToClient(id clientID, char* message);

    void startEngine(void);
    void stopEngine(void);

    bool startGame(const std :: string& server,
        const std :: string& database,
                const std :: string& username,
                const std :: string& password,
                int port, int maxConnections);
    void stopGame();
};
```

Funkce `startGame` spustí jednu instanci hry na základě parametrů předaných této funkci. Spuštěnou hru je možné ukončit funkcí `stopGame` . Funkce `startEngine` resp. `stopEngine` jsou vnitřní funkce sloužící k spuštění, resp. zastavení jedné instance hry. Funkce `closeClient` odpojí klienta, pokud je připojen. Klientovi je

možné poslat data pomocí funkce `sendToClient` , přičemž funkce se stane vlastníkem předané zprávy, není tedy nutné řetězec odalokovat. Pokud klient s daným identifikátorem není připojen, pouze se zpráva odalokuje.

```
class context
{
public:
    /* databaze */
    dbClient db;

    /* cache listy */

    dv_time :: datetime             lastSynchronization;

    void initialize_cache_lists(void);
    void initialize_events(void);

    void synchronize(bool force = false);

    static const unsigned long index = 0;
};
```

Lokální kontext obsahuje všechny datové struktury, případně další položky, pro každou jednotlivou instanci hry. Povinně obsahuje databázového klienta `db` , datum a čas poslední synchronizace cache listů s diskem - `lastSynchronization` , index instance hry v proměnné `index` (zatím statická proměnná), funkci inicializující cache listy - `initialize_cache_lists` , funkci inicializující jednotlivé objekty - `initialize_events` a konečně funkcí `synchronize` synchronizující data v operační paměti s daty v databázi na pevném disku.

6.13 Chyby a logování

K zaznamenávání logů slouží makro `LOG` . V ladícím režimu zaznamená i funkci, soubor a řádku, na které bylo toto makro zavoláno. Makro volá jednu z funkcí `log` a `rlog` , z nichž první je určena pro ostrou verzi serveru, zatímco druhá pro ladící verzi. Funkce `lnextline` slouží k odřádkování. Všechny výše zmíněné funkce jsou plně synchronizované a lze je tedy volat z libovolného vlákna. Pokud chce uživatel napsat na konzoli něco sám, musí k synchronizaci použít funkce `lockConsoleForWrite` a `unlockConsoleForWrite` .

Z důvodu ladění server vypisuje některé informace automaticky, zatímco v ostré verzi se vypisují pouze chyby. Nastavení těchto ladících zpráv se povoluje, případně zakazuje, pomocí následujících maker:

```
#if defined(_DEBUG)
    /* verbose */
```

```
    /* globalni povoleni / zakazani ladicich zprav */
    #define DV_SERVER_VERBOSE

    #if defined(DV_SERVER_VERBOSE)
        #define DV_SERVER_VERBOSE_REQUEST
/* zobrazovani requestu                                    */
        #define DV_SERVER_VERBOSE_LOGON
/* zobrazovani prihlaseni                                  */
      // #define DV_SERVER_VERBOSE_SENDED_DATA
/* zobrazovani poslanych dat                               */
        #define DV_SERVER_VERBOSE_EVENTS
/* zobrazovani zmen eventu                                 */
        #define DV_SERVER_VERBOSE_MOBILITY_EVENT
/* zobrazovani spusteni mobility eventu                    */
        #define DV_SERVER_VERBOSE_QGPLASMA_EVENT
/* zobrazovani spusteni qgplasma eventu                    */
        #define DV_SERVER_VERBOSE_ENERGY_DISTRIBUTION_EVENT
/* zobrazovani spusteni energy distribution eventu         */
        #define DV_SERVER_VERBOSE_CONTAINERED_RESOURCES_EVENT
/* zobrazovani spusteni containered resource eventu        */
        #define DV_SERVER_VERBOSE_CHECK_CONSTRUCTION_EVENT
/* zobrazovani spusteni kontroly postaveni eventu          */
    #endif
#endif
```

Na zaznamenávání menších i kritických chyb slouží makra `errExceptionError` a `errLogicalError` . Zaznamenají soubor, funkci a řádku, ve které chyba nastala a také chybovou hlášku. V ladící verzi je možné zapnout breakpoint na místě chyby v případě, že chyba nastane.

6.14 Komunikační protokol

Komunikační protokol je čistě textový, je proto možné si ho přečíst. Na začátku zprávy je číslo s údajem, jak dlouhá je zpráva za tímto číslem. Poté následuje hlavička s typem zprávy, popř. jménem požadavku zasílaný na server/klienta, následují parametry ukončené speciálním řetězcem.

K vytváření zpráv v tomto protokolu slouží třída `protocolWriter` . Pokud chceme vytvořit zprávu, nejprve vytvoříme novou instanci této třídy. Poté zavoláme funkci `beginMessage` s požadovaným typem zprávy. Parametry lze zapisovat pomocí funkce `writeValue` , lze takto zapisovat různé číselné, řetězcové i boolean hodnoty. Nakonec zprávu ukončíme funkcí `endMessage` , která nám vrátí konstantní ukazatel na řetězec s vytvořenou textovou zprávou. Tento řetězec lze duplikovat pomocí funkce `copyMessage` .

Vytvořenou textovou zprávu je možné zpětně rozparsovat pomocí třídy `protocolReader` . Funkci `translateMessage` předáme ukazatel na řetězec s textovou zprávou a tato funkce nám vrátí instanci třídy `message` reprezentující obsah dané zprávy. Typ zprávy zjistíme pomocí funkce `getType` , jméno žádosti dostaneme funkcí `getRequestName` , druh odpovědi funkcí `getAnswerKind` a konečně mapu parametrů funkcí `getValues` .

6.15 Časová osa

V mnoha hrách jsou důležité asynchronně, na hráči nezávisle spouštěné různé typy událostí. U budovatelského typu her to může být například postavení budovy. Ke správě těchto událostí slouží časová osa. Události je možné vytvářet, registrovat a asynchronně v určitý čas spouštět. Nový typ události se vyrobí poděděním třídy `triggerEventClass` a přepsáním příslušných virtuálních funkcí. Funkce `checkEvent` spustí příslušný kód, který má být proveden v určitý čas, uživatel ji musí podědit a napsat si do ní vlastní kód. Obdobně funkce `getEventName` slouží k definování jména události. Z hlediska vnitřní logiky události je nutné si zaznamenávat identifikátor události, který získáme pomocí funkce `getEventName` . S pomocí tohoto identifikátoru lze aktualizovat čas spuštění události tím, že se tato událost smaže a nahradí se nějakou bližší událostí stejné třídy. Funkce `getServerID` slouží jako potenciální možnost rozlišit eventy pro více instancí hry běžící na jednom serveru.

Události spravuje třída `timeAxis` , ty události, které je nutné spustit, předá tato třída instanci enginu hry (viz. kapitola níže). Pomocí funkce `setEngine` nastavíme příslušný engine hry, kterému má časová osa zasílat události, které je nutné spustit. Funkcí `start` spustíme časovou osu, naopak funkcí `stop` ji zastavíme. Novou událost vsuneme funkcí `pushEventTrigger` . Časová osa si sama zjistí, zda je událost aktuální, nebo existuje jiná, kterou je nutné spustit dříve. V takovém případě tuto událost smaže, v opačném případě smaže starší událost a nahradí ji aktuální. Třídy událostí určuje proměnná `lastTriggerID` , která určuje, se kterou událostí se bude nová událost porovnávat. Pokud žádná taková událost neexistuje, je vhodné nastavit tuto proměnnou na nulu.

6.16 Vlastní engine

Engine hry tvoří jádro celé hry. Zpracovává požadavky od klientů a také asynchronní události, které mu dodá časová osa. Třídy požadavků jsou uloženy v tabulce, kterou je před spuštěním hry nutné inicializovat funkcí `init_request_table` . Samotný server lze spustit metodou `start` a zastavit metodou `stop` . Pokud chceme provádět zásahy do datových struktur hry za běhu hry, je nutná synchronizace pomocí metod `beginWork` a `endWork` . Požadavek klienta se zpracovává metodou `processRequest` , nové požadavky klienta lze přidat metodou `addRequest` , zatímco asynchronní události určené ke spuštění se přidávají metodou `addEvent` . Nové

události je možné registrovat metodou `registerEvent`, která zavolá příslušnou metodu časové osy. Funkce `isClientUpdateSuppressed`, `suppressClientUpdates` a `renewClientUpdates` slouží k potlačení zasílání aktualizací klientům.

Rejstřík