Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Renderovací systém pre streamované dáta pre Ogre

Bakalárska práca

2014 **Dušan Plavák**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Renderovací systém pre streamované dáta pre Ogre

Bakalárska práca

Študijný odbor:	9.2.1 Informatika
Číslo študijného odboru:	2508 Informatika
Školiace pracovisko:	Katedra Informatiky FMFI
Vedúci práce:	RNDr. Jaroslav Janáček, PhD.

2014 **Dušan Plavák**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Dušan Plavák
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský

Názov: Renderovací systém pre streamované dáta pre Ogre
Network-streaming Rendering System for Ogre

Cieľ: Cieľom práce je navrhnúť komunikačný protokol a vytvoriť funkčné API, príslušnú dokumentáciu a tutoriál pre používateľov systému Ogre (Object-Oriented Graphics Rendering Engine), ktoré umožnia zobrazovať rôzne pohľady do aplikácie na viacerých zarideniach, ktoré sú prepojené sieťou a majú inštalovanú rovnakú aplikáciu s podporou pre streamovanie.

Vedúci: RNDr. Jaroslav Janáček, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 21.10.2013

Dátum schválenia: 23.10.2013

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.... študent

.... vedúci práce

Poďakovanie

Týmto by som sa chcel poďakovať svojmu vedúcemu RNDr. Jaroslavovi Janáčkovi, PhD. za rady, nápady a odbornú spoluprácu, ako aj za ústretovosť a ľudský prístup. Samozrejme poďakovanie patrí aj všetkým, ktorý vo mňa verili a podporovali ma.

Abstrakt

V tejto bakalárskej práci sme navrhli a implementovali doplnok pre grafický engine Ogre, ktorý pridáva podporu synchronizácie grafickej scény v aplikáciach spojených po sieti. Tento doplnok je prototyp so základnou funkcionalitou fungujúci pod rôznymi operačnými systémami a na rôznych zariadeniach.

Kľúčové slová: ogre, grafický engine, grafická scéna, synchronizácia scény, synchronizácia po sieti

Abstract

We have been designing and implementing an extension for graphics engine Ogre. This extension adds capability for synchronization of graphics scene in applications connected over a network. It is prototype with basic functionality working under different operating systems and on different devices.

Keywords: ogre, graphics engine, graphics scene, scene synchronization, synchronization over a network

Obsah

Úvod **1**

1 OGRE **2**

1.1 Čo je to OGRE 2
1.2 Vlastnosti OGRE 2
1.3 Využitie OGRE 3

2 Ciele práce **4**

2.1 Čo chceme dosiahnuť 4
2.2 Postup práce 5

3 Implementácia **6**

3.1 Analýza Ogre 6
3.1.1 Hlavné komponenty 6
3.1.2 Hlavný objekt - Root 8
3.1.3 SceneManager zblízka 8
3.2 Návrh riešenia 9
3.2.1 Obmedzenia riešenia 10
3.3 Zber informácií 10
3.3.1 Duplicitné získavanie informácie 11
3.3.2 Reprezentácia informácie 12
3.3.3 Uloženie získanej informácie 14
3.3.4 Odpočúvanie v praxi 15
3.4 Prenos informácií 16
3.4.1 Formát správ na sieti 17
3.4.2 Viacero klientov 18
3.4.3 Implementácia servera 19
3.5 Aplikácia informácií 23
3.5.1 Injektor 23
3.5.2 Všetko v jednom vlákne 24
3.5.3 Neznáme objekty 25
3.6 Používanie funkcionality 26

3.7 Kompilácia Ogre knižníc so synchronizačným rozšírením 27

4 Testovanie, výsledky, zhrnutie 28

4.1 Podporovaná funkcionalita 28

4.2 Testované platformy 29

4.3 Ďalší vývoj 30

Záver 32

Literatúra 33

Príloha A 35

Zoznam obrázkov

3.1 Hlavné komponenty Ogre, autor: Steve Streeting 7
3.2 Hierarchia dedenia triedy RenderSystem, autor: Torus Knot Software Ltd . 8
3.3 Štruktúra na uloženie príkazu 14
3.4 Formát správy 18

Zoznam ukážok kódu

3.1 definícia funkcie Light::_notifyAttached() 15
3.2 implementácia funkcie acceptClient() 20
3.3 implementácia funkcie popInformation() 21
3.4 implementácia funkcie dispatch() 22

Úvod

Grafický engine Ogre je na používanie jednoduchý framework, v ktorom sa dajú vytvárať aplikácie využívajúce akékoľvek grafické prvky. Tento framework sa používa na vytváranie rôznych druhov aplikácií, od hier až po lekárske aplikácie či simulátory určené na výučbu ľudí.

Ogre je open-source projekt a nemá platených vývojárov. Ľudia pracujú na tomto projekte zadarmo a je vyvíjaný komunitou nadšených ľudí. Táto komunita následne poskytuje aj podporu pre Ogre na fóre a irc kanály. Aj z tohto dôvodu sa na stránkach tohto projektu dajú nájsť zoznamy vecí, ktoré by uživatelia v Ogre chceli mať ale nie sú ešte implementované. V tejto práci sa pozrieme na jednu z takýchto požiadaviek a skúsime navrhnúť a implementovať riešenie.

V prvej kapitole sa bližšie pozrieme na projekt Ogre, popíšeme si jeho základné charakteristiky a využitie. Následne v druhej kapitole budeme pojednávať o cieli tejto práce a oboznámime sa s vybranou požiadavkou, ktorú budeme implementovať. Tretia kapitola obsahuje hlavnú časť práce a to implementáciu. Budeme sa tam zaoberať celým postupom vývoja, od analýzy problému, návrhu riešenia až po implementáciu a riešenia problémov vyskytujúcich sa počas práce. Na záver štvrtej kapitoly zhrnieme dosiahnuté výsledky práce a zamyslíme sa nad možným budúcim pokračovaním tejto práce.

KAPITOLA 1

OGRE

1.1 Čo je to OGRE

OGRE je skratka z anglického **O**bject-Oriented **G**raphics **R**endering **E**ngine (ďalej v texte OGRE), čo v preklade znamená objektovo orientovaný grafický vykresľovací engine. Objektovo orientovaný je nazývaným pretože jeho kód je písaný objektovo orientovaným jazykom C++. Na OGRE sa dá pozerať ako na API 3D grafického systému, ktoré nám poskytuje abstrakciu nad nižšími vrstvami zabezpečujúcimi odovzdávanie informácie priamo grafickej karte ako napríklad OpenGL alebo Direct3D. Často krát sa môžeme stretnúť s označením Ogre3d, ktoré sa používa, aby pri komunikácií nedochádzalo k nedorozumeniam, pretože Ogre sa používa aj na označenie mýtickej príšery požierajúcej deti.

1.2 Vlastnosti OGRE

Ogre sa snaží poskytnúť vývojárom aplikácií čo najväčšiu mieru abstrakcie, rozhranie ktoré je postavené na objektoch z reálneho sveta. Výsledkom je grafický engine, zameriavajúci sa na scénu aplikácie a čo najjedenoduchšiu prácu s ňou. OGRE je projekt s prístupným zdojovým kódom pod licenciou MIT, čo ho umožňuje využívať na takmer čokoľvek, rovnako aj študentom. Kód engine-u je napísaný s dôrazom na ľahkú prenositeľnosť aplikácií na základe čoho sa dá ľahko skompilovať na operačnom systéme Linux, Windows a Mac OSX, mimo iného má istú podporu aj pre platformy Android a IOS. OGRE v súčastnosti podporuje tieto vykresľovacie systémy:

- OpenGL
- OpenGL 3+
- OpenGL ES

- OpenGL ES 2
- Direct3D9
- Direct3D11

umožňuje prácu s materiálmi a písanie vlastných shader-ov v jazykoch:

- Cg
- DirectX9
- HLSL
- GLSL

a podporuje načítavanie textúr z rôznych obrázkových formátov, animácie, špeciálne efekty, particles systémy a mnoho ďalšieho.

Android platforma podporuje spúšťanie kódu napísaného v C++ čo umožňuje spúšťať aplikácie vytvorené v OGRE aj pod touto platformou. Android podporuje oficiálne NDK[1], knižnice poskytujúce rozhranie pre Android. Naneštastie oficiálne NDK nepodporuje systém výnimiek, ktorý OGRE vo veľkej miere používa v kóde. Ako riešenie prišlo neoficiálne NDK z komunity, ktoré dovoľuje používať výnimky.

OGRE ako taký nepredpokladá typ výslednej aplikácie, neobsahuje zbytočnosti ale zameriava sa na kvalitu, kvalitu písaného kódu ako aj samotnej dokumentácie. Ľudia stojaci za projektom zastávajú názor, že kvantita je niečo, čo môže prísť s časom ale kvalita sa spätne jednoducho docieliť nedá. Aj preto každá nová súčasť OGRE musí prejsť podrobnou kontrolou kódu a nové súčasti Ogre môžu akceptovať len tvorcovia projektu, ľudia čo majú za sebou prax, skúsenosti. Aj takýmto štýlom sa dosahuje stabilita a kvalita celého engine-u.

1.3 Využitie OGRE

Často krát sa ľudia pýtajú či je OGRE game engine, odpoveďou je nie. Tým, že OGRE nemá požiadavky na cieľovú aplikáciu, môže byť použitý v rôznych typoch aplikácií. Celý engine je zameraný čisto len na vykresľovanie grafiky a tvorcovia hlásia, že to tak aj vždy bude. Práve fakt, že OGRE neobsahuje zbytočné veci netýkajúce sa grafiky ho umožňuje začleniť do ktoréhokoľvek typu aplikácie. Aj preto sa tento open-source projekt používa v širokom spektre aplikácií ako napríklad námorný simulátor, rôzne 3D hry až po komerčné aplikácie zameriavajúce sa na hudbu, či lekárske programy.

[1] Native Development Kit

KAPITOLA 2

Ciele práce

2.1 Čo chceme dosiahnuť

Cieľom práce je oboznámiť sa s grafickým systémom Ogre, navrhnúť a implementovať čo najjednoduchšie API, ktoré bude zabezpečovať synchronizáciu grafickej scény v aplikáciach využívajúcich tento grafický engine. Pod zabezpečením synchronizácie scény ale rozumieme stav, kedy synchronizované aplikácie majú nie len synchronizovaný grafický výstup, ale práve naopak, dôraz sa kladie aby bol synchronizovaný obsah scény, teda objekty ktoré sa tam nachádzajú a rôzne vlastnosti danej scény.

Synchronizácia scény by mala fungovať teoreticky na ľubovolnom počte aplikácií, na rôznych zariadeniach a platformách avšak minimálne na systémoch Linux, Windows, Mac OSX.

Využitie má všade tam, kde chceme zobraziť čo najväčšiu časť scény s detailmi. Ak by sme to robili bez synchronizácie scény na viacerých zariadeniach, museli by sme všetko vykresliť na jednom zariadení a výsledok zobraziť na výstupnom zariadení, zariadeniach. Takéto vykreslenie veľkej scény môže byť náročné na výpočtové zdroje zariadenia. Ak však použijeme synchronizáciu scény na viacerých zariadeniach, stačí ak na každom zariadení vykreslíme len jednu časť výslednej scény.

Typickým príkladom použitia sú rôzne simulátory, kde zvyčajne chceme zobraziť prostredie okolo nás a teda potrebujeme vykresliť veľkú časť scény. S naším implementovaným doplnkom môžeme synchronizovať scénu a na každej aplikácií vykreslíme iný pohľad na scénu tak aby zodpovedal potrebám daného simulátora.

Použitie nie je limitované len tam, kde potrebujeme vykresliť veľkú časť scény. Ďalším typickým použitím sú rôzne hry viacerých hráčov kde sa zvykne poskytovať mož-

nosť sledovať hru bez priamej účasti. Toto sledovanie sa dá jednoducho implementovať synchronizáciou scény.

2.2 Postup práce

Dosiahnutie cieľov práce sa dá rozdeliť na nejakú postupnosť krokov ktoré treba vykonať, aby sme sa dopracovali k výsledku. V tejto práci budeme postupovať následovne:

- Analýza súčasného Ogre API a jeho možností
- Návrh riešenia
- Implementovanie riešenia
- Dokumentácia, vytvorenie príkladu použitia vytvoreného API

V provom rade sa oboznámime so Ogre engine-om. Je potrebné najskôr spoznať prostredie, v ktorom budeme pracovať. Po tom, čo dostaneme predstavu o tom, ako Ogre funguje, navrhneme riešenie ako implementovať náš doplnok do engine-u a implementujeme ho. Na záver popíšeme ako doplnok funguje a vytvoríme ukážky použitia, fungovania nášho doplnku.

KAPITOLA 3

Implementácia

3.1 Analýza Ogre

Už pri prvom kontakte s Ogre si môžeme všimnúť, že Ogre je celkom komplexný a rozsiahly systém. Pri druhom kontakte sa postupne začne ukazovať, že vďaka jeho objektovému návrhu a vhodnej dávke abstrakcie pri práci s Ogre nemusíme byť v priamom kontakte s veľkou zložitosťou celého engine-u. Rozumný nadhľad nám ponúka aj fakt, že ogre sa rozdeľuje do 3 hlavných komponentov a celý systém je dobre zdokumentovaný. Dokumentáciu nájdeme v podstate pre každú jednu funkciu, ktorú nám API ponúka.

3.1.1 Hlavné komponenty

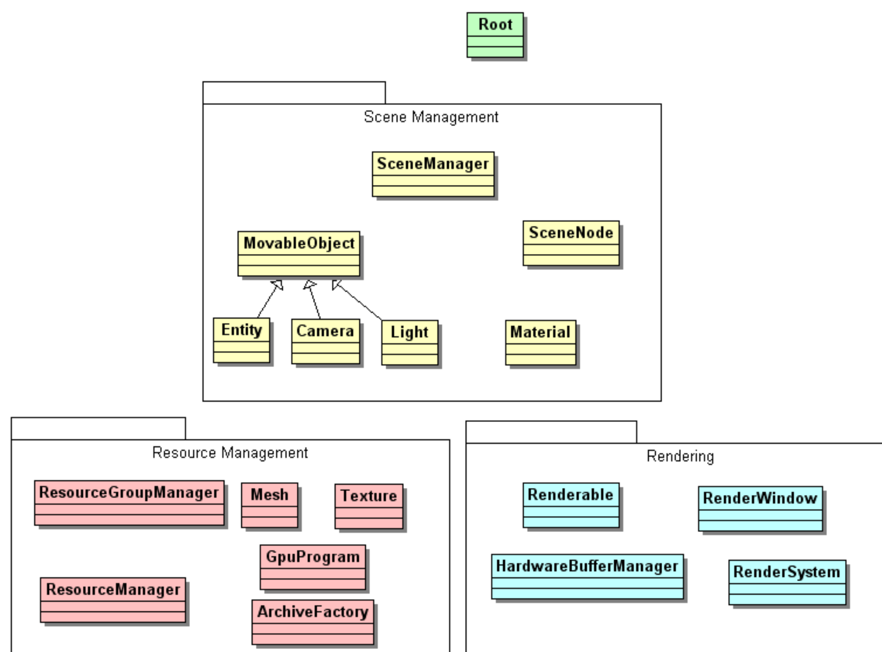
Ogre je postavené na 3 hlavných komponentoch a to:

- komponent na manažment scén
- komponent na manažment zdrojov
- komponent na vykresľovanie scény

Komponent na manažment scén zahŕňa všetky triedy reprezentujúce objekty, ktoré sa môžu ocitnúť na scéne. Do tohto komponentu sa radia aj všetky ostatné triedy, ktoré nám nejakým spôsobom ovplyvňujú scénu, menia vlastnosti objektov, či samotnej scény. Ako reprezentanta tohto komponentu môžeme považovať triedu SceneManager [22], ktorá v podstate slúži ako nejaký spojovník medzi všetkými triedami v tomto komponente. Trieda SceneManager nám poskytuje funkcionalitu umožňujúcu vytvárať objekty jednotlivých tried a umiestňovať ich na scénu, zisťovať aktuálny stav scény, jej obsah, vlastnosti.

Komponent na manažment zdrojov sa v engine-e stará predovšetkým o to, aby boli dostupné v aplikácií tie súčasti, ktoré sú uložené v externých súboroch a nie priamo v kóde. Typickým príkladom sú rôzne textúry alebo 3D modely, materiály, fonty a podobne. Tak ako pri komponente pre manažment scény sme vybrali reprezentatívnu triedu, komponent na manažment má ako reprezentanta triedu ResourceManager [20]. Na triede ResourceManager je priamo závislá trieda SceneManager, nakoľko ak chceme na scéne vytvoriť nejakú entitu, môžeme zavolať funkciu createEntity [10], ktorá ako argumenty berie názov novej entity a buď mesh objekt[1], alebo názov súboru, v ktorom je uložený, prípadne len názov samotného mesh objektu, pod ktorým sa dá vyhľadať. V prvom prípade nemusíme volať ResourceManager, nakoľko máme priamy ukazovateľ na tento objekt. V nasledujúcich prípadoch sa zavolá ResourceManager, kde sa buď mesh objekt načíta zo súboru do aplikácie a vráti sa ukazovateľ alebo v prípade poskytnutia mena mesh objektu sa tento objekt v ResourceManger-i vyhľadá a znova sa vráti ukazovateľ na tento objekt.

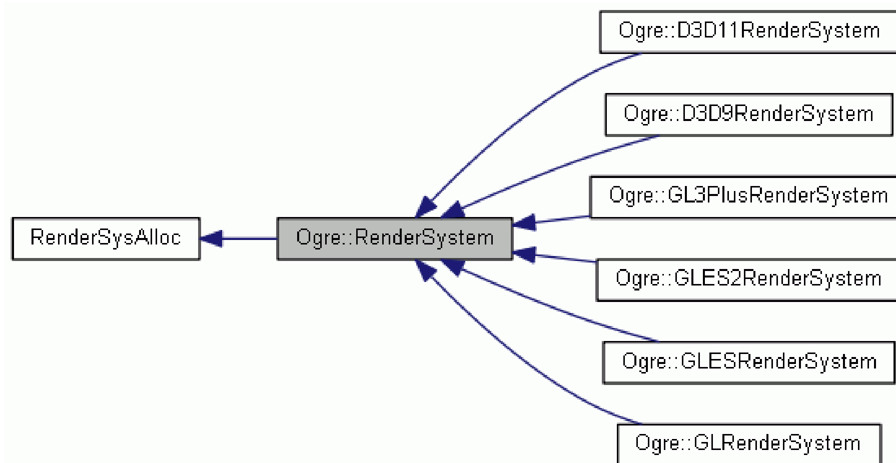
Posledný komponent slúži priamo na prácu s grafickou kartou. Triedy v tomto komponente nám zabezpečujú, že naša scéna sa zobrazí na výstupnom zariadení, bude vykreslená. Reprezentant komponentu je trieda RenderSystem [19]. V bežných aplikáciach uživatelia nepotrebujú priamo používať túto triedu, nakoľko o komunikáciu s ňou sa stará samotná trieda SceneManager a triedy pracujúce s materiálmi, objektmi na scéne. Pod bežnou aplikáciou sa myslí aplikácia nevyžadujúca si viacero okien.



Obr. 3.1: Hlavné komponenty Ogre, autor: Steve Streeting

[1]Ogre má vlastný súborový formát na ukladanie objektov v súbore s typickou koncovkou .mesh a pracuje výlučne s týmto formátom

Na záver k hlavným komponentom Ogre je treba dodať, že reprezentatívne triedy komponentov viac menej slúžia len ako triedy definujúce rozhranie pre rôzne implementácie daných tried. To následne umožnuje vytvárať jednoducho doplnky do Ogre. Fungovanie tohto systému názorne zobrazuje Obrázok 3.1.1 ktorý znázorňuje dedenie pre triedu RenderSystem, ktorá má v súčasnosti šesť implementácií:



Obr. 3.2: Hierarchia dedenia triedy RenderSystem, autor: Torus Knot Software Ltd

3.1.2 Hlavný objekt - Root

Pri budovaní aplikácie využívajúcej Ogre engine je vstupným bodom do Ogre práve root objekt [21]. Root objekt spája všetky komponenty. Vytvára sa na začiatku, a deštruktor by mal byť volaný na konci aplikácie. Skrátene, root objekt by mal byť prvým vytvoreným a posledným uvoľneným objektom. Zabezpečuje inicializáciu a konfiguráciu ostatných systémov (SceneManager a iné). Root objekt je tiež dôležitý ak naša aplikácia potrebuje ustavične prekresľovať aktuálnu scénu. V tomto prípade je potrebné zavolať funkciu startRendering() [24], ktorá zabezpečí vstup systému do cyklu. Z tohto sú len dva východiská, buď sú všetky vykresľované okná zatvorené alebo ak niektorý z registrovaných FrameListener [12] objektov požiada o zastavenie tohto cyklu. FrameListener je v podstate rozhranie, definujúce ako majú vyzerať objekty, ktoré budú upovedomené vždy pred vykreslením snímku (anglicky frame).

3.1.3 SceneManager zblízka

SceneManager je jedna z najviac využívaných tried Ogre engine-om. Nachádza sa v nej informácia o všetkom čo sa má vykresliť na výstupné zariadenie. Obsahuje zoznam kamier, svetiel a iných objektov, vyskytujúcich sa na scéne. Napriek tomu, že táto trieda je jedna z najviac využívaných Ogre engine-om, užívateľ zvyčajne k tejto triede bude najviac pristupovať len na začiatku aplikácie kde bude vykonávať nastavenie scény.

Neskôr používanie tejto triedy nie je až tak časté nakoľko ďalšie modifikácie väčšinou prebiehajú priamo na daných objektoch scény.

3.2 Návrh riešenia

Pri analýze Ogre engine-u sa ukázalo, že najviac zaujímavý pre synchronizovanie scény bude práve komponent na manažovanie scény a triedy ktoré sú v ňom obsiahnuté. Našim cieľom je synchronizovať scénu, objekty na scéne, a to takým spôsobom, že nám to umožní nie len vidieť totožný obraz na synchronizovaných aplikáciach ale aj scénu z rôznych uhlov pohľadov.

Synchronizáciu scény vieme rozdeliť na 3 menšie podproblémy a to:

- získanie informácie o zmene stavu scény / získanie stavu celej scény
- prenos získanej informácie
- aplikovanie prenesenej informácie

Jedným z možných prístupov ako riešiť synchronizáciu scény, je vytvoriť plugin do Ogre, ktorý by zabezpečoval túto synchronizáciu. Takéto riešenie je vhodné z pohľadu toho, že nijako nezasahuje do samotnej implementácie Ogre engine-u. Výhoda vyplývajúca z tohto riešenia sa stáva aj jeho nevýhodou, pretože to by znamenalo, že strácame možnosť detekovať zmenu vtedy keď nastane. Zmenu na scéne by sme mali možnosť detekovať jedine pred tým ako sa ide vykresliť nejaký snímok a to tým spôsobom, že by bolo potrebné porovnať stav scény z predchádzajúcim stavom a informáciu o tejto zmene preniesť do aplikácie, ktorá má byť synchronizovaná. Takáto detekcia zmeny sa ale ukazuje ako veľmi neefektívna, nakoľko objekty na scéne sú komplexné a súčasne ich býva veľké množstvo. Okrem časovej zložitosti je pri tomto riešení aj veľká pamäťová zložitosť. Pre detekovanie zmeny na scéne by bolo nutné si udržiavať kópiu predchádzajúcej scény aby bolo možné vytvoriť rozdiel týchto scén.

Iný prístup je modifikovať priamo implementáciu samotného Ogre engine-u, čím by sme dostali možnosť vedieť o každej zmene na scéne okamžite, po tom ako nastane. Takýto prístup je efektívny, čo sa času aj pamäte týka a prináša minimálne zaťaženie pre aplikáciu. Ukazuje sa, že tento prístup je vhodný na implementáciu a aj preto ďalej v práci budeme popisovať implementáciu práve tohto prístupu. Nevýhodou zvoleného riešenia je, že priamo treba modifikovať Ogre engine, ktorý sa snaží byť čisto grafickým engine-om, bez výnimiek.

3.2.1 Obmedzenia riešenia

Myšlienka riešenia by nemala mať prílišné obmedzenia na používanie. Avšak v tejto práci implementujeme riešenie, ktoré poskytne len čiastočnú funkcionalitu Ogre, ktorú sa bude dať synchronizovať pomocou nášho API a to z dôvodu veľkej komplexity Ogre engine-u a nedostatku času. Ogre systém obsahuje v jeho aktuálnom API vyše 1200 tried. Preto pri implementácii sa sústredíme predovšetkým na komponent pre manažovanie scén a implementujeme len istú podmnožinu celkovej funkcionality a to tak, aby výsledná implementácia poskytovala možnosť synchronizovať scény aplikácií istého typu. Zároveň budeme predpokladať, že aplikácia, respektíve obsah aplikácie, ktorý chceme synchronizovať sa na scéne ešte nenachádza a celá konfigurácia scény sa bude diať až po tom, čo aplikácie, ktoré chcú mať synchronizovanú scénu, nadviažu spolu spojenie. Toto obmedzenie zavádzame z jednoduchého dôvodu. Na to, aby sme vedeli zosynchronizovať už prednastavenú scénu, museli by sme najskôr preniesť celú scénu na druhú aplikáciu. Na prenos celej scény by sme museli podporovať synchronizáciu väčšiny funkcionality Ogre engine-u, ktorú sme už s oddôvodnením obmedzili.

Ogre v projekte dodržiava aj isté zásady a požiadavky ktoré by malo následne aj nami vytvorené API spĺňať. Jedna z takýchto požiadaviek je, že Ogre, respektíve API ktoré poskytuje je multiplatformové, čiže neviaže sa na nejakú konkrétnu platformu a teda ak aplikácia užívateľa bude používať Ogre API, mal by byť schopný túto aplikáciu skompilovať na rôznych platformách a to bez zmeny kódu. Momentálne Ogre podporuje Linux, Windows, Mac OSX, Android, IOS. Dá sa povedať, že Ogre aplikácie by mali byť schopné fungovať na systémoch kde je prístupný C++ kompilátor a aspoň jeden z renderovacích systémov: OpenGL, OpenGL 3+, OpenGL ES, OpenGL ES 2, Direct3D9, Direct3D11. Druhú zásadu, ktorá síce nie je v Ogre spomínaná ale budeme ju brať pri implementácií do úvahy je, že tento vyvíjaný doplnok by mal byť ako voliteľná súčasť Ogre, ktorá sa dá jednoducho vybrať pri kompilovaní, a preto každá zmena, doplnenie v aktuálnom API by malo byť v uzavretom podmienkovom bloku, ktorý buď bude alebo nebude braný do úvahy pri kompilovaní Ogre.

3.3 Zber informácií

V tejto sekcií sa budeme zaoberať problémom ako čo najefektívnejšie zistiť zmeny v danej scéne a ako túto informáciu reprezentovať tak aby bola ľahko aplikovateľná na klientskej aplikácií. Treba sa zamyslieť nad tým, čo pre nás znamená, že nastala na scéne nejaká zmena. V princípe to znamená, že buď nastala zmena v nejakom nastavení scény alebo nastala zmena priamo na nejakom objekte na scéne. Tieto zmeny ale mohli nastať len dvoma, pre nás prípustnými spôsobmi. Buď sa nejakému objektu respektíve atribútu nejakého objektu priradila priamo nejaká hodnota, alebo bola zavolaná nejaká

funkcia, ktorá na objekte vykonala zmeny. Ogre systém viacmenej nepodporuje priamu zmenu vlastností, priame priradenie. Preto prvá možnosť nepripadá do úvahy, čo nám vyhovuje, pretože detekovať takýto typ zmeny je zložité. Ak bol nejaký objekt zmenený volaním funkcie, potom vieme ihneď zistiť, že sa niečo zmenilo, a aj ako sa to zmenilo, ktorým volaním funkcie sa to zmenilo. Na to, aby sme vedeli robiť takúto detekciu, musíme modifikovať každú jednu funkciu, ktorej zmeny chceme byť schopní zachytávať.

Informácia o tom aká funkcia bola volaná a s akými argumentmi vo väčšine prípadov pre nás nebude dostačujúca, nakoľko tieto funkcie budú zväčša volané na objektoch, ktoré nevieme bez dodatočnej informácie jasne identifikovať. Z tohto nám vyplýva potreba posielať si aj informáciu, ktorá nám jednoznačne určí, na ktorom objekte bola daná funkcia volaná. Ako neskôr ukážeme, toto sa dosahuje komplikovane, prípadne sa to v súčasnej implementácii Ogre API ani dosiahnuť nedá, bez dodatočných zmien.

3.3.1 Duplicitné získavanie informácie

Ogre tým, že poskytuje vysokú mieru abstrakcie a zároveň sa snaží uľahčiť používanie svojho API tak ako to ide, využíva dedenie z rôznych tried. Z času na čas definuje funkcie, ktoré v podstate nerobia nič iné len prevolávajú iné funkcie a poskytujú rovnakú funkcionalitu len pod iným menom. Ak by sme teda zaznamenávali volania každej funkcie, vyústilo by to do stavu, kedy by sme takúto informáciu poslali na klientskú aplikáciu viackrát, čo by viedlo k desynchronizácií scény. V podstate na odstránenie tohto problému stačí jednoducho odchytávať iba tie volania, ktoré sú prevolávané z inak pomenovaných funkcií s rovnakým významom.

Podobný problém duplicitnej informácie nastáva aj na základe dedenia. Ak sa pozrieme napríklad na triedu SceneNode [23], ktorá dedí od triedy Node [17] a konkrétne na jej konštruktor [14], môžeme si všimnúť, že okrem nejakých interných vecí volá aj konštruktor triedy Node. Tu nastáva teda problém, pretože ak odpočúvame konštruktor oboch tried, výsledkom bude získanie informácie o zostrojení objektu SceneNode, ktorý si sám v rámci seba zavolá konštruktor triedy Node, čo nás ale už nezaujíma, pretože toto je obsiahnuté v informácií o vytvorení SceneNode objektu a získame aj druhú informáciu o vytvorení objektu Node, ktorý ale v skutočnosti nevytvárame ako samostatný objekt. Aplikovanie riešenia z predchádzajúceho problému neprichádza do úvahy, nakoľko sa tu jedná o dva rôzne objekty. Tieto síce majú istú spoločnú množinu vlastností ale nie sú totožné a môžu existovať ako dva samostatné objekty. Podstata celého problému spočíva v tom, že nám chýba kontextová informácia, konkrétne, že volanie konštruktoru triedy Node nie je priame, ale z nejakej inej funkcie.

Pri riešení nášho problému sa dajú zvoliť dva prístupy. Prvým prístupom by bolo neriešiť tento problém pri zbere informácií, ale až pri ich aplikácií na klientskej aplikácii. Pri tomto prístupe by sme museli teda pristúpiť na kontextovú analýzu prichádzajúcich informácií. Ak by sme prijali informáciu o vytvorení Node objektu, potrebovali by sme zistiť, či to bolo priame volanie alebo nejaké vnorené volanie. Bolo by potrebné pozrieť sa na informácie, ktoré sme prijali doteraz a zistiť či sa tam nenachádza informácia o volaní, ktoré vykonáva vnorené volanie Node konštruktoru. Tento prístup má hneď niekoľko nevýhod. Jedna z nich je zvýšenie komplexity kódu na klientskej aplikácií, ktorý bude interpretovať prijatú informáciu. Okrem iného sa nám tiež zvýši pamäťová náročnosť, keďže kvôli tomu aby sme vedeli vykonať kontextovú analýzu musíme si nejakým spôsobom uchovávať predchádzajúcu informáciu. Ďalšou nevýhodou je vytvorenie závislosti kódu, ktorý bude interpretovať získanú informáciu od konkrétnej implementácie systému Ogre. V prípade, že by sa v Ogre zmenila implementácia konštruktora SceneNode, ktorý by už nevolal konštruktor Node, museli by sme meniť aj implementáciu tohto interpretera. V opačnom prípade nám hrozí riziko, že budeme nesprávne interpretovať získané informácie.

Druhý prístup k tomuto problému je modifikovať implementáciu systému Ogre, a to takým spôsobom, že do volaní funkcie pridáme ďalší argument, ktorý nám bude indikovať či sa jedná o priame volanie funkcie alebo je toto volanie vnorené v nejakej inej funkcii. Tento argument môže byť boleanovská premenná. Aby sme ale neporušili spätnú kompatibilitu s aplikáciami napísanými so staršou verziou systému Ogre, tento argument pridáme ako nepovinný s predvolenou hodnotou, čo nám zaručí, že pri kompilovaní kódu postaveného na predchádzajúcej verzií tento nový argument vo volaniach funkcií nebude chýbať. Toto riešenie v podstate rieši aj prvý problém, avšak mierne menej efektívne, pretože tam kde v skutočnosti netreba odpočúvať volania funkcií, by nám toto riešenie pridalo odpočúvanie. Avšak funkcie, ktoré by sa v tomto volaní volali by mali nastavený flag, že nie sú volané priamo.

3.3.2 Reprezentácia informácie

Zistili sme teda, že na to aby sme vedeli synchronizovať scénu, stačí nám synchronizovať volania funkcií na objektoch. Aby sme dokázali urobiť identické volanie na klientskej aplikácii, potrebujeme teda tieto základné informácie:

- identifikátor funkcie, ktorá je volaná
- argumenty, s ktorými je funkcia volaná
- identifikátor objektu, na ktorom je funkcia volaná

Identifikátor funkcie

Funkciu môžeme jednoznačne identifikovať tak, že každej jednej priradíme číslo. Nakoľko je ale počet funkcií veľmi veľký, je lepšie názov reprezentovať menom, čo zlepšuje čitateľnosť kódu. V tomto prípade nám stačí použiť konštrukciu enum, ktorú nám ponúka jazyk C++, pričom spojíme číslo s menom. Zvyčajne sa ale veci, ktoré spolu súvisia nejakým spôsobom zoskupujú, čo v našom prípade znamená, že by sme chceli mať funkcie operujúce nad jedným typom objektu nejako zoskupené. V prípade, že by sme mali jeden veľký zoznam funkcií v enum-e, takéto zoskupenie by nám robilo problém, v prípade ak by sme chceli pridávať nejaké dodatočné funkcie do zoznamu neskôr. Museli by sme buď prečíslovať všetky nasledujúce funkcie v zozname alebo pridanú funkciu zaradiť do zoznamu ale s poradovým číslom, ktoré je posledné voľné. Tento prístup by ale zhoršil čitateľnosť kódu, zakaždým by sme museli hľadať posledné použité číslo. Z týchto dôvodov zavedieme kategorizovanie funkcií, takže identifikátor funkcie sa bude skladať z dvoch čísel, jedno udávajúce kategóriu a druhé udávajúce číslo funkcie v kategórií. Týmto spôsobom môžeme mať enum pre každú kategóriu zvlášť, čo nám uľahčí pridávanie nových funkcií do kategórií. Ako teda funkcie kategorizovať? Celkom logické kategorizovanie je podľa triedy, do ktorej dané funkcie patria.

Argumenty funkcie

Pri reprezentovaní argumentov funkcií veľa možností nie je. Argumenty, ktoré sú jednoduchého dátového typu reprezentujeme ich hodnotou. Dátový typ si posielať nemusíme, pretože táto informácia vyplýva z identifikátora funkcie. Tým, že vieme o akú funkciu sa jedná, vieme aj aké argumenty očakáva. Pri zložitejších dátových typoch, objektoch sa ich hodnota (poprípade stav) vyjadruje komplikovanejšie. V tomto prípade nám nič iné nezostáva, len poslať všetky dáta potrebné na zostrojenie argumentu, objektu. Jednoduchým príkladom je trieda ColourValue [9], ktorej stav sa dá reprezentovať jej štyrmi float premennými. V podstate na konštrukciu tohto objektu stačí ak budeme mať tieto 4 čísla a použijeme na klientskej aplikácií konštruktor [13] tejto triedy. Tento spôsob reprezentácie ale nie je efektívny pri komplikovaných objektoch, dátových typoch, ktoré buď obsahujú veľa premmenných alebo rôzne ukazovatele na iné objekty. Pre veľmi komplikované objekty nám stačí poslať unikátny identifikátor objektu. Treba si uvedomiť, že komplikované objekty sa budú veľmi pravdepodobne často používať a preto je výhodné ich reprezentovať unikátnym identifikátorom na rozdiel od jednoduchých dátových typov. Navyše ak by sme pri jednoduchom dátovom type ako ColorValue chceli zaviesť reprezentáciu unikátnym identifikátorom, znamenalo by to pre nás nutnosť podporovať odpočúvanie všetkých funkcií v danej triede, ktoré menia stav, hodnotu dátového typu, Čo by bolo menej efektívne, nakoľko zvyčajne by sme museli poslať klientskej aplikácii oveľa viac informácií - vytvorenie objektu, všetky volania na tomto objekte, oproti poslaniu štyroch čísel len raz.

Identifikátor objektu

Ako sme už spomínali v predchádzajúcom texte, na to aby sme volanie nejakej funkcie vedeli priradiť objektu, na ktorom bola volaná, potrebujeme nejaký jednoznačný, unikátny identifikátor. Ogre pre väčšinu objektov na scéne, o ktorých vie SceneManager objekt, má unikátne meno, ktoré sa dá použiť ako jednoznačný identifikátor. Ukazuje sa ale, že situácia u objektov, ktoré sú obsiahnuté v objektoch, ktoré majú unikátny identifikátor, je horšia. Ako príklad si môžeme uviesť objekty typu OgreEntity [11], ktorý má funkciu getName(), ktorá vracia unikátne meno. Pretože trieda Entity dedí od triedy MovableObject [16], a každý objekt typu MovableObject ktorý sa nachádza na scéne, musí mať unikátny názov, nakoľko sa ukladá do kontaineru Map v C++. Keď sa teraz pozrieme na triedu Entity, tá môže mať objekt typu AnimationState [8], ktorý má síce funkciu getAnimationName() ale tá vracia názov, ktorý nemusí byť unikátny.

Riešenie tohto problému je v podstate jednoduché. Treba navrhnúť deterministický generátor mien, ktorý bude každému objektu generovať unikátne meno. Spolu s generátorom je potrebné upraviť implementáciu súčasného Ogre API tak, aby poskytovala funkcie pre objekty na nastavenie a získanie takéhoto unikátneho mena. Takéto unikátne meno budeme teda generovať automaticky, pokiaľ užívateľ nezvolí vlastné meno pre objekt. Akonáhle zvolí vlastné meno, urobíme kontrolu či dané meno nie je ešte obsadené a ak áno, vyhodíme výnimku.
Ako si ale neskôr ukážeme, niekedy nám nebude stačiť ani táto informácia.

3.3.3 Uloženie získanej informácie

Informáciu, ktorú v odpočúvanej funkcií získame, potrebujeme niekde uložiť aby sme ju mohli neskôr spracovať, respektíve poslať klientskej aplikácií. Túto informáciu by sme mohli poslať okamžite v momente keď ju získame ale vzhľadom na to, že môže byť na server pripojených viacero klientov, posielanie informácie každému klientovi by mohlo spomaliť chod servera. Preto chceme oddeliť získavanie informácie od jej posielania klientom. Dáta, ktoré potrebujem ukladať, neskôr poslať, dajú sa nazvať príkazmi, pretože skutočne identifikujú príkazy ktoré sa majú vykonať. Preto sme vytvorili novú štruktúru, v ktorej budeme ukladať práve odpočutý príkaz. Obsah štruktúry je priamočiary, čo môžeme vidieť aj na Obrázku 3.3:

boost::uint32_t category	boost::uint32_t number	std::vector<Ogre::String> params

Obr. 3.3: Štruktúra na uloženie príkazu

Každé jedno volanie funkcie, ktorú odpočúvame si teda v tomto formáte uložíme do rady(std::queue<>) na Root objekte. Ukladanie na Root objekte je výhodné, najmä kvôli tomu, že je všade prístupný.

3.3.4 Odpočúvanie v praxi

Teraz si ukážeme ako to teda reálne vyzerá. Ako príklad nám poslúži trieda Light [15] a jej funkcia _notifyAttached() [18], ktorú môžeme vidieť v ukážke Zdrojového kódu 3.1:

Ukážka kódu 3.1: definícia funkcie Light::_notifyAttached()

```
1  void Light::_notifyAttached(Node* parent, bool isTagPoint)
2  {
3      #ifdef OGRE_NETWORK_SYNCHRONIZATION
4      if(Ogre::SceneSync::shouldIntercept())
5      {
6          boost::uint32_t cmd_category = Ogre::SceneSync::CommandsCategory::LIGHT;
7          boost::uint32_t cmd_number = Ogre::SceneSync::Light::NOTIFY_ATTACHED;
8  
9          std::vector<Ogre::String> params;
10 
11         params.push_back(this->getName());
12         params.push_back(parent->getName());
13         params.push_back(Ogre::StringConverter::toString(isTagPoint));
14 
15         Ogre::SceneSync::syncCommand(cmd_category, cmd_number, params);
16     }
17     #endif
18 
19     mDerivedTransformDirty = true;
20 
21     MovableObject::_notifyAttached(parent, isTagPoint);
22 }
```

Kód, ktorý zabezpečuje odpočúvanie sa nachádza na riadkoch 3-17. Môžeme si všimnúť podľa riadku číslo 3, že ak sa bude systém Ogre kompilovať bez definovanej konštanty OGRE_NETWORK_SYNCHRONIZATION, žiaden kód sa nepridá, čo je veľká výhoda najmä pre uživateľov, ktorí neplánujú využívať možnosť synchronizácie scény. Následne na riadku 4 sa volaním funkcie shouldIntercept() overí či sú podmienky na odpočúvanie splnené. Po zavolaní tejto funkcie sa najmä kontroluje či bola zapnutá funkcionalita synchronizácie a či sa jedná o priame volanie alebo vnorené volanie. Na riadku 6-7 nastavujeme kategóriu a číslo príkazu. Riadok 11-13 pripravuje argumenty volania príkazu. Konkrétne na riadku 11 si ukladáme informáciu na akom objekte má byť funkcia zavolaná na klientskej aplikácií. Dvanásty riadok je zaujímavý tým, že namiesto toho aby sme si uložili celý komplexný objekt typu Node, uložíme si len jeho

unikátny identifikátor a na riadku 13 uložíme argument, s ktorým bola funkcia _notify-Attached() volaná. Na záver zozbierané informácie odovzdáme funkcií syncCommand(), ktorá zabezpečuje uschovanie príkazu pokiaľ nebude odoslaný na klientskú aplikáciu.

3.4 Prenos informácií

Akonáhle sa nám nejakým spôsobom zmení scéna, chceme o tom informovať aplikácie, ktoré majú byť synchronizované, čo v praxi znamená, že potrebujeme poslať cez nejaké spojenie informáciu o tejto zmene. V našom prípade budeme mať vždy len jednu aplikáciu, jeden počítač, ktorý bude slúžiť ako vzor, podľa ktorého sa budú ostatné aplikácie synchronizovať. Môžeme teda z tejto aplikácie vytvoriť server, na ktorý sa budú všetky ostatné aplikácie pripájať ako klienti. Keďže systém Ogre je multiplatformový, musíme zabezpečiť, aby naša aplikácia bola schopná komunikovať na akomkoľvek systéme, či už sa jedná o Linux, Windows, Mac OSX, IOS alebo Android. Na docielenie tejto nezávislosti budeme používať knižnice boost [1], ktoré sú skompilovateľné pod uvedenými systémami s malou výnimkou - Androidom. Android nie je oficiálne podporovaný, pretože vývojari boost knižnice majú problém s testovaním týchto knižníc na všetkých verziách Androidu.

Aby sme zbytočne nezaťažovali hlavnú aplikáciu na serveri a ani klientov, komunikačnú časť presunieme do osobitných vlákien. Na vytvorenie nových vlákien použijeme opätovne boost knižnice, ktoré ponúkajú širokú funkcionalitu, ktorá nie je závislá na konkrétnom systéme. Následne v novom vlákne je už vytvorenie servera jednoduché. Použijeme boost::asio::io_service [7], na základe ktorej vytvoríme tcp spojenie s klientom. Akonáhle sa klient pripojí na server, môžeme začať odosielať informácie o zmene scény, ak nejaké nastali. Či nastali nejaké zmeny zistíme jednoducho tak, že sa pozrieme či rada s príkazmi je prázdna alebo nie. Ak nie, vyberieme spredu jeden príkaz a pošleme ho po sieti klientovi.

Pri posielaní príkazov cez sieť si ale treba dať pozor na problémy spojené s viac vláknovými aplikáciami. Pristupovať k nejakým dátam z viacerých vlákien je nebezpečné ak prístup nie je realizovaný atomickými operáciami. V tom prípade môže dôjsť k takzvanému data race [3], čo vedie k nedefinovanému správaniu programu. Na vyriešenie tohto problému stačí použiť štandardné riešenie - zámky, uzamykanie prístupu k dátovej oblasti. Akonáhle sme odpočuli nejakú novú informáciu a chceme ju pridať do rady príkazov, najskôr uzamkneme zámok vyhradený pre túto dátovú štruktúru a až potom ju modifikujeme. To isté spravíme aj pri odosielaní dát klientskej aplikácií. Systém zámkov nám zabezpečí, že nebudeme pristupovať k rovnakej dátovej štruktúre súčasne z dvoch rôznych miest.

3.4.1 Formát správ na sieti

Pri posielaní správ[2] po sieti je potrebné si dohodnúť nejaký formát, aby klientská aplikácia vedela ako má prijaté dáta interpretovať. V našom prípade si budeme po sieti posielať v správach informáciu o príkazoch, ktoré boli vykonané na strane servera. To teda znamená: identifikátor funkcie, identifikátor objektu, na ktorom je volaná a argumenty tejto funkcie. Keďže počet argumentov a aj ich typ je veľmi variabilný, treba zvoliť formát správ, ktoré budú vhodné práve na častú zmenu dĺžky posielanej informácie. V každej správe máme práve dve variabilné zložky, a to:

- počet argumentov
- veľkosť[3] každého argumentu

Počet argumentov

Počet argumentov je informácia, ktorá nám vyplýva z toho aká funkcia je na serveri volaná. To znamená, že informácia by sa nemusela posielať po sieti ale ukazuje sa, že hneď z niekoľkých dôvodov je lepšie si túto informáciu prenášať. Prvý z dôvodou je fakt, že síce sme sa v Ogre ešte nestretli s použitím funkcií, ktoré nemajú pevne definovanú aritu[4], to ale neznamená, že v budúcnosti sa funkcie tohto typu nebudú používať. V tomto prípade by nastal problém, pretože by sme nevedeli jasne určiť počet argumentov na základe toho, ktorá funkcia bola volaná. Druhým dôvodom prečo si posielať informáciu o počte argumentov je možnosť lepšie oddeliť sieťovú komunikáciu od interpretovania informácie. Ak by sa táto informácia neprenášala, museli by sme sa hneď pri čítaní zo siete pozerať na to aká funkcia bola volaná a následne podľa toho určiť aké ďalšie množstvo informácií patrí k danej správe. S informáciou o počte argumentov vieme jednoducho načítať jednu správu bez toho aby sme museli nejakým mechanizmom určovať rozsah správy a následne túto správu spracovávať na inom mieste.

Veľkosť argumentu

Každý typ argumentu môže mať rôznu veľkosť a navyše rovnaký typ argumentu sa môže líšiť vo veľkosti. Jednoduchým príkladom je dátový typ String. String môže obsahovať informáciu s premenlivou veľkosťou. V našej aplikácii navyše takýto typ argumentu využívame takmer v každom volaní.

Ak sa rozprávame o veľkosti argumentu treba v našom prípade myslieť aj na fakt, že aplikácia bude potencionálne využívaná na viacerých platformách, kompilovaná rôznymi kompilátormi. Problém je, že štandard jazyka C++ neurčuje veľkosť dátového

[2]jedna správa zahrňuje informáciu o volaní jednej funkcie
[3]veľkosťou označujeme počet bajtov
[4]jedná sa o funkcie s premenlivým počtom argumentov

typu, ale len rozsah dát, ktoré sa dajú reprezentovať daným typom. Výsledkom je, že ak sa na dvoch platformách opýtame funkciou sizeof() na veľkosť toho istého typu, môžeme dostať dve rôzne odpovede. Tento problém riešime tak, že používame dátový typ boost::uint32_t, ktorý nám garantuje veľkosť argumentu 4 bajty. Tento "univerzálny typ" používame na reprezentáciu kategórie a čísla funkcie, počet argumentov a ich veľkosť. Na samotné argumenty používame dátový typ String. Práve táto časť by sa dala implementovať efektívnejšie. Pri posielaní argumentov typu String je to vyhovujúce riešenie ale reálne posielame aj argumenty typu integer. Reprezentácia typu integer typom String je neefektívna, keďže dátovým typom String o veľkosti 1 bajt vieme vyjadriť číslo v rozmedzí 0 - 9, pričom do jedného bajtu vieme zakódovať až 2^8 hodnôt, čiže 0 - 255. Tento problém by sa dal vyriešiť ak by sme argumenty prenášali ako bajty použité na ich reprezentáciu a na strane klienta by sme explicitnou typovou konverziou transformovali tieto bajty na príslušný dátový typ.

V tejto práci budeme kódovať argumenty ako dátový typ string, pretože Ogre má triedu StringConverter [25], ktorá uľahčuje konverziu z rôznych dátových typov práve do typu string a naopak.

Celá sieťová komunikácia po naviazaní spojenia sa teda skladá priamo z postupností správ reprezentujúcich jednotlivé príkazy. Výsledný formát správy môžeme vidieť na Obrázku 3.4.

Obr. 3.4: Formát správy

3.4.2 Viacero klientov

Pre maximalizovanie využitia tohto synchronizačného doplnku do Ogre je vhodné navrhnúť komunikačnú časť tak, aby server zvládal pripojenie viacerých klientov naraz. Tento cieľ je možné dosiahnuť niekoľkými spôsobmi implementovania serverovej časti, ktoré si v nasledujúcej časti popíšeme. V princípe ak máme niekoľko klientov pripojených na server a informáciu, ktorú im chceme poslať, treba riešiť niekoľko problémov s tým spojenými.

Jedno vlákno, proces na klienta

Ak by sme si zvolili cestu osobitných procesov, museli by sme riešiť niekoľko problémov. Hlavný problém ale je, že procesy majú oddelený adresný priestor, čo v praxi

znamená, že nemôžu zdieľať medzi sebou dáta. Samozrejme táto situácia sa dá riešiť medziprocesovou komunikáciou, je to však komplikovanejšie na implementáciu. Pri procesoch sa nám ale vyskytne ďalší problém ak chceme, aby náš kód fungoval na viacerých platformách. Ak si zoberieme napríklad len Windows a Linux, každý z týchto systémov rieši problematiku procesov inak, čo by predstavovalo ďalší kód navyše.

V prípade zvolenia prístupu jedno vlákno pre každého klienta je potrebné myslieť na vhodný prístup k zdieľaným štruktúram. Musíme zamedziť súčasnému prístupu viacerých vlákien k zdieľaným štruktúram, ktoré vlákna menia. S týmto prístupom prichádzajú aj isté výhody. V prípade, že sa nejaký klient z nejakého dôvodu odpojí, môžeme si dovoliť istý čas čakať, či sa nepripojí, bez toho, aby sme obmedzovali ostatných klientov.

Jedno vlákno pre všetkých

Jeden z ďalších prístupov je obsluhovať všetkých klientov na jednom vlákne. V takomto prípade nemusíme synchronizovať prístup k dátovým štruktúram, nakoľko všetko prebieha na jednom vlákne. Pozor si ale treba dávať na blokujúce operácie, ktoré by nám mohli zablokovať celý server kvôli jednému klientovi. Klasické blokujúce operácie sú: čakanie na pripojenie klienta alebo odoslanie, čítanie istého množstva informácie.

Ukazuje sa ale, že moderné technológie nám dovoľujú kombinovať výhody oboch prístupov súčasne. Z prvého prístupu máme výhodu klientov ktorý sa navzájom neobmedzujú a z druhého prístupu prevezmeme výhodu prístupu k dátam len z jedného miesta. Kombinovať tieto dve výhody nám umožnuje knižnica boost::asio [5]. Táto knižnica ponúka funkcionalitu pre asynchrónne vstupno výstupné operácie. Práve tato asynchronita nám umožnuje kombinovať tieto výhody.

3.4.3 Implementácia servera

Pri implementácií servera používame knižnicu boost::asio, ktorá ponúka širokú funkcionalitu pre prácu so sieťovou komunikáciou. Pri implementácií používame tcp spojenie, ktoré nám garantuje, že dáta klientom budú poslané a zaručuje aj poradie správ, dokým nedôjde k zlyhaniu spojenia. Fakt, že správy na klienta prídu všetky a v správnom poradí je veľmi dôležitý. V prípade, že správy neprídu v správnom poradí, negarantujeme korektné správanie klientskej aplikácie a s veľkou pravdepodobnosťou môže dôjsť k desynchronizácií aplikácií. V prípade ak by sme predsa len chceli použiť udp protokol na spoľahlivej sieti, nie sú potrebné veľké zmeny v kóde. Celá funkcionalita servera je obsiahnutá v troch hlavných funkciách. Celá implementácia servera obsahuje ešte navyše konštruktor a deštruktor triedy.

Fungovanie servera sa začína v momente kedy sa vytvorí objekt typu Server. V jeho konštruktore sa inicializuje objekt io_service [6], poskytujúci funkcionalitu na komunikáciu po sieti a objekt acceptor [4] slúžiaci na prijímanie nových spojení z klientov. Následne sa vytvoria dva vlákna. V prvom vlákne sa nachádza kód funkcie popInformation() ktorá zaisťuje vybratie dát z rady do ktorej sú ukladané dáta získané z odpočúvania funkcií. Druhé vlákno slúži na obsluhu io_service objektu.

Ukážka kódu 3.2: implementácia funkcie acceptClient()

```
1  void Server::acceptClient(boost::system::error_code ec)
2  {
3      if (!ec) {
4          boost::lock_guard<boost::mutex> lk(mx);
5          connections.push_back(ss);
6      }
7
8      if (ec != boost::asio::error::operation_aborted) {
9          ss.reset(new tcp::socket(iosvc));
10         acceptor.async_accept(*ss, boost::bind(&Server::acceptClient, this, ↩
               boost::asio::placeholders::error));
11     }
12 }
```

acceptClient()

Funkcia acceptClient()(ukážka 3.2) nám slúži na obsluhovanie prichádzajúcich spojení. V podstate akonáhle objekt acceptor zaregistruje nové, prichádzajúce spojenie a príjme ho, zavolá túto funkciu. Na začiatku funkcie si uložíme socket slúžiaci na komunikáciu s klientom do zoznamu klientov, aby sme s ním neskôr vedeli komunikovať (riadok č.5), pripravíme nový socket pre ďalšieho klienta (riadok č.9) a ako vidno na riadku 10 zavoláme znova na objekte acceptor funkciu async_accept(), ktorá začne čakať na nové prichádzajúce spojenia, pričom ako argumenty jej dáme novo pripravený socket a súčasne nastavíme funkciu, ktorá sa má postarať o prijaté spojenie, na aktuálnu. V podstate vytvoríme akúsi zdanlivo nekonečnú rekurziu keď funkcia acceptClient() nepriamo volá sama seba.

popInformation()

Kód vo funkcií popInformation()(ukážka 3.3) sa vykonáva v osobitnom vlákne. Názov funkcie nám napovedá, že budeme odniekiaľ vyberať informáciu. Konkrétne to bude z rady kde ukladáme odpočutú informáciu. V tomto vlákne budeme v podstate robiť len dve veci. Vyberieme prvú položku z rady a pošleme ju na odoslanie klientom. Prvý krok musíme znova ošetriť pomocou systému zámkov, keďže k rade pristupujeme z dvoch rôznych vlákien - aktuálneho a vlákna, na ktorom sa deje odpočúvanie volaní funkcií. V kóde tejto funkcie sú zaujímavé dva riadky. Riadok 5 a riadok 27. Na riadku 5 môžeme

vidieť takzvaný interruption point, čo sa dá voľne do slovenčiny preložiť ako bod prerušenia. Tento bod nám slúži na to, aby sme vlákno vedeli z vonka nejakým spôsobom zastaviť. Knižnica boost::thread má niekoľko preddefinovaných bodov prerušenia, čo sú v podstate nejaké funkcie, ktoré keď sú zavolané, skontrolujú či nedošlo k žiadosti o prerušenie daného vlákna, ak sú prerušenia povolené. V prípade že dôjde k prerušeniu vyhodí sa výnimka, ktorá zabezpečí skončenie vlákna, ak nie je odchytená. Riadok číslo 27 informuje objekt io_service, že chceme vykonať funkciu dispatch() s uvedenými argumentami, kde argument c je odpočutý príkaz. Funkcia post() definovaná na objekte io_service nečaká na vykonanie funkcie, ktorej referenciu má ako argument, ale len zaistí jej vykonanie mimo aktuálneho vlákna, v našom prípade vo vlákne slúžiacom na obsluhu objektu io_service.

Ukážka kódu 3.3: implementácia funkcie popInformation()

```
1  void Server::popInformation()
2  {
3      for (;;)
4      {
5          boost::this_thread::interruption_point();
6
7          Ogre::Root* root = Ogre::Root::getSingletonPtr();
8          root->dataQueueMutex.lock();
9
10         bool empty = false;
11         empty = root->outgoingCommands->empty();
12
13         Ogre::SceneSynchronizer::Command c;
14
15         if(!empty){
16             c = root->outgoingCommands->front();
17             root->outgoingCommands->pop();
18         }
19         else
20         {
21             root->dataQueueMutex.unlock();
22             continue;
23         }
24
25         root->dataQueueMutex.unlock();
26
27         iosvc.post(boost::bind(&Server::dispatch, this, c));
28     }
29 }
```

dispatch()

Zasielanie dát klientom nám zabezpečuje funkcia dispatch()(ukážka 3.4), ktorá informáciu, ktorú dostane ako argument pošle všetkým pripojeným klientom. V úvode funkcie si predspracujeme dáta, ktoré máme uložené v štruktúre Command. Túto štruk-

túru serializujeme (riadok č.3) tak aby výsledok bol vo formáte, ktorý popisujeme na Obrázku 3.4. Následne preiterujeme cez všetky uložené spojenia a ak je spojenie stále otvorené, pošleme klientovi dáta, inak spojenie vymažeme(riadok 27). Na riadkoch 20-21 vidíme posielanie dát, ktoré sa znova realizuje asynchrónne aby sme neblokovali posielanie dát ďalším klientom. Funkcia async_write definovaná na objekte typu socket ako prvý argument vyžaduje úložisko, z ktorého sa majú dáta odoslať. Druhý argument je funkcia, ktorá sa má zavolať po dokončení dátového prenosu. V našom prípade máme ako druhý argument definovaný operátor Noop() s tromi argumentami. Funkcia async_write vyžaduje aby Noop() prijímala argumenty, ktoré určujú či sa vyskytla nejaká chyba a veľkosť prenesených dát. Navyše vyžaduje, aby prenášané dáta boli dostupné až kým nebudú odoslané. Na zachovanie dostupnosti dát použijeme drobnú fintu. Operátoru Noop() dodáme tretí argument a to práve dáta, ktoré odosielame. Ako vidíme na riadku 3, odosielané dáta definujeme ako shared_ptr [2] čo nám umožní tieto dáta uchovávať v pamäti aj po opustení funkcie dispatch().

Ukážka kódu 3.4: implementácia funkcie dispatch()

```
1  void Server::dispatch(Ogre::SceneSynchronizer::Command cmd)
2  {
3      boost::shared_ptr<std::string> data = boost::make_shared<std::string>(cmd.↩
           serialize());
4
5      boost::lock_guard<boost::mutex> lk(mx);
6
7      for (std::vector<boost::shared_ptr<tcp::socket> >::iterator sit = ↩
           connections.begin(); sit != connections.end();)
8      {
9          if ((*sit)->is_open())
10         {
11             boost::system::error_code ec;
12             (*sit)->remote_endpoint(ec);
13
14             if (ec)
15             {
16                 (*sit)->close();
17             }
18             else
19             {
20                 (*sit)->async_send(boost::asio::buffer(*data),
21                 boost::bind(Noop(), data /*keep-alive*/, _1, _2));
22             }
23             ++sit;
24         }
25         else
26         {
27             sit = connections.erase(sit);
28         }
29     }
30 }
```

3.5 Aplikácia informácií

Serverové časti ako odchytávanie volaní funkcií a samotné riešenie servera zabezpečujúceho komunikáciu s klientmi sme si popísali. V tejto časti sa budeme prevažne venovať klientskej aplikácií. Klientská aplikácia je pomerne jednoduchá na zostrojenie. V úvode sme uvádzali predpoklad, že na to aby sme vedeli synchronizovať scénu aplikácií na rôznych zariadeniach, musí sa na týchto zariadeniach nachádzať identická aplikácia. To nie je tak úplne pravda, pretože v skutočnosti potrebujeme mať zachovanú súborovú štruktúru aplikácie so zdrojmi, ktoré sa budú používať. Jedná sa najme o rôzne modely, textúry, materiály uložené v rôznych súboroch, prípadne nejaké iné skripty, ktoré by sme potrebovali načítať zo súboru. Ďalší, pomerne jasný predpoklad pre správne fungovanie synchronizácie je schopnosť hardware-u vykonať požadované grafické operácie. Viac menej ak na serveri použijeme niečo, čo grafická karta na klientovi nepodporuje, nemôžeme očakávať rovnaký výsledok na scéne servera a klienta. V mnohých prípadoch to môže dokonca viesť k pádu celej aplikácie.

Komunikácia so serverom

Komunikácia klienta so serverom je riešená jednoduchým spôsobom. Celú komunikáciu obsluhuje jedno vlákno, na ktorom sa pripojíme na server a čakáme na prichádzajúce informácie ktoré podobne ako na servery po odchytení uložíme do rady na neskoršie spracovanie. Pri komunikácií používame blokujúce volania boost::asio::connect(), ktoré slúži na pripojenie sa k serveru a potom už len boost::asio::read(), ktoré nám zabezpečuje čítanie informácie zo siete.

3.5.1 Injektor

Na klientskej aplikácií je hlavným komponentom pri synchronizácií scény takzvaný Injektor, ktorého reprezentuje trieda Injector. Takéto označenie sme zvolili pretože v skutočnosti to čo sa deje na klientskej aplikácií je doplňovanie informácie do aktuálnej scény na klientovi napríklad nové objekty, zmena vlastností - vkladáme, injektujeme ďalšiu informáciu.

Trieda Injektor má okrem konštruktora a deštruktora už len dve funkcie - clear() a inject(). Okrem týchto funkcií, má trieda niekoľko privátnych dátových členov, a to konkrétne hash mapy reprezentované v C++ dátovým typom std::map. Tieto hash mapy nám slúžia na ukladanie si ukazovateľov na objekty, s ktorými pracujeme na scéne. Pre každý typ objektu, ktorý sa môže vyskytnúť na scéne máme jednu hash mapu. Takáto kategorizácia ako sa neskôr ukáže nám príde vhod.

Funkcia clear() robí presne to isté čo deštruktor, uvoľní všetky použité zdroje. Táto funkcia má využitie najme vtedy ak sme synchronizovali scénu z nejakej aplikácie a chceli by sme synchronizovať scénu inej aplikácie, čiže okrem toho, že sa musíme pripojiť na iný server, musíme zmazať uložené informácie v injektore, ktoré by mohli ovplyvniť synchronizáciu novej scény.

Hlavnou funkciou na klientskej strane je funkcia inject() v triede Injector. Táto funkcia sa stará o interpretovanie informácií, ktoré nám prišli zo servera. Úlohou tejto funkcie je zistiť z prichádzajúcej informácie, ktorá funkcia bola na serveri zavolaná, na akom objekte a s akými argumentami. Všetky tieto informácie vieme jednoducho vyčítať z prijatej informácie celkom jednoducho. Jediný problém nastáva v tom, ako identifikovať objekt, na ktorom má byť funkcia zavolaná. Zo servera sme si síce poslali jednoznačný identifikátor objektu a treba si uvedomiť, že tento identifikátor platí len na serveri a klientskej aplikácie sa už netýka. V tomto momente sa nám zídu vytvorené hash mapy, do ktorých si ukladáme ukazovatele na objekty. Tieto hash mapy nám budú slúžiť ako nejaké mapovanie identifikátorov objektov na serveri na identifikátory objektov u klienta. Toto sa dá docieliť tak, že ak na serveri zavoláme nejakú funkciu, ktorá vytvára nejaký objekt, tak okrem informácie o tom, ktorá funkcia bola volaná a na akom objekte, si pošleme aj informáciu o tom ako sa novovzniknutý objekt na serveri volá a tento identifikátor použijeme ako kľúč v prislúchajúcej hash mape.

Napriek tomu, že funkcia inject() v triede Injector môže vzbudzovať dojem veľkej zložitosti, jej štruktúra je jednoduchá. V tejto funkcií robíme dve veci. Ako prvé z rady vyberieme jednu správu, ktorú chceme interpretovať. Následne interpretuje túto správu, pričom pod interpretáciou rozumieme zavolanie správnej funkcie, ktorú určíme prechodom cez dva vnorené prepínače. Prvý nám filtruje kategóriu príkazu a druhý prepínač nám filtruje číslo príkazu. Takto sa jednoducho dostaneme ku kódu, ktorý interpretuje danú správu.

3.5.2 Všetko v jednom vlákne

Prvá implementácia injektora bežala na vlastnom vlákne. Osobitné vlákno sme pre injektor vybrali kvôli tomu aby informácie prichádzajúce zo servera mohol bezodkladne interpretovať. Všetko sa zdalo byť v poriadku ale čím sa funkcionalita injektoru zväčšovala, začali sa objavovať problémy Klientská aplikácia začala padať a dôvod nebol jasný. Neskôr sa ukázalo, že voľba osobitného vlákna pre injektor zapríčiňovala tieto problémy. V podstate problém bol v tom, že sme pristupovali k istým štruktúram z viacerych vlákien. Ogre totižto neumožňuje vykonávanie akýchkoľvek operácií týkajúcich sa grafickej karty na viacerých vláknach. Z tohto dôvodu sa funkcia inject() volá z hlavného vlákna v jednom z potencionálne mnohých FrameListener-ov [12], ktoré sú

volané pred každým vykreslením snímku. Aj z tohto dôvodu funkcia inject() akceptuje jeden argument typu int, ktorý nám hovorí koľko príkazov sa má interpretovať v danom volaní. Tento argument je dôležitý, nakoľko ak bude nastavený na príliš nízke číslo, celá scéna bude viditeľne spomalená, no na druhej strane ak sa nastaví na príliš vysoké číslo, môže to spôsobiť pokles fps[5] danej aplikácie pretože interpretovanie príliš veľa správ bude pridlho zdržiavať vykreslenie snímku. V aktuálnej implementácií nechávame voľbu tohto argumentu na užívateľa, ale vhodnejšie by bolo určiť správny počet príkazov, ktoré sa majú interpretovať v jednom snímku automaticky. Toto by sa dalo docieliť pomerne jednoducho zaregistrovaním si nového FrameListener-u na serveri a posielať si správu o vykreslení snímkov. Tieto správy by nám následne označovali správy, ktoré sa majú interpretovať v jednom snímku.

3.5.3 Neznáme objekty

Ukazuje sa, že pri rozširovaní funkcionality v budúcnosti sa budeme musieť vysporiadať s ďalším problémom. Problém takzvaných neznámych objektov. Tento problém sa bude vyskytovať pri interpretácií správ prichádzajúcich zo servera na klienta, kedy sa bude jednať o správu, ktorej význam je vykonať nejakú funkciu na objekte, ktorého serverové identifikačné meno poznáme ale to sa nemapuje na žiadny objekt na klientovi. Tento problém sa môže vyskytnúť najmä pri vytvorení zložitejšieho objektu a následne modifikovaní jeho vnútorných podobjektov. Jednoduchý príklad je objekt typu Entity, napríklad nejaká postavička, ktorá sa vnútorne skladá z niekoľkých kostí a podobne. Problém nastáva v momente kedy my si prenesieme informáciu o vytvorení objektu Entity a aj jeho unikátny identifikátor zo servera na klienta. Vytváranie objektov Entity zabezpečuje funkcia createEntity() [10], ktorá ale v sebe vytvára ďalšie objekty, napríklad kosti postavičky a ich identifikátor si už neprenášame. Tento problém by bolo možné riešiť posielaním si identifikátorov všetkých objektov, ktoré daná funkcia vytvorí. To ale znamená, že ak bol objekt vytvorený v nejakom vnútornom volaní odpočúvanej funkcie, musíme tieto identifikátory objektov nejakým spôsobom “vybublať” na vrchnú úroveň odpočúvanej funkcie. Na klientskej strane by to znamenalo po interpretovaní takejto správy preiterovať cez práve vytvorený objekt a vytvoriť mapovanie všetkých jeho vnútorných objektov s unikátnymi identifikátormi zo servera. Je možné, že existuje aj vhodnejšie riešenie tohto problému, no my sa celkovo nezaoberáme týmto problémom v aktuálnej implementácií keďže našu podporovanú funkcionalitu tento problém neovplyvňuje.

[5]frame per second - počet snímkov za sekundu

3.6 Používanie funkcionality

Teraz keď už máme predstavu ako celá synchronizácia funguje, ukážeme si ako sa to celé používa. Používanie je pomerne jednoduché a nevyžaduje si znalosť ako systém funguje. Predpokladajme, že máme nejakú aplikáciu, do ktorej chceme doplniť synchronizáciu. Treba postupovať podľa nasledujúcich krokov:

1. vložiť potrebné hlavičkové súbory
2. vytvoriť objekt typu Ogre::SceneSynchronizer::Synchronizer
3. zadefinovať miesto spustenia / zastavenia servera, klienta, odpočúvania
4. vytvoriť injektor
5. zadefinovať miesto pre injektor
6. zrušiť injektor

1) Na to aby sme mohli používať funkcionalitu synchronizovania scény musíme do kódu vložiť hlavičkový súbor s definíciami funkcií, ktoré budeme používať: OgreSceneSynchronizer.h

2) Je potrebné mať vytvorenú inštanciu objektu Ogre::SceneSynchronizer::Synchronizer, ktorý nám zaobaľuje celú funkcionalitu a poskytuje jednotný prístup k nastaveniu všetkého potrebného.

3) Pod zadefinovaním miesta spustenia / vypnutia sa myslí určiť podmienku, ktorá po splnení zapne / vypne server, klienta alebo odpočúvanie. Zvyčajne sa ako podmienka zvykne využívať stlačenie nejakej klávesy, prípadne tlačítka na obrazovke. Na spustenie alebo vypnutie servera stačí zavolať funkciu toggleServer() na objekte Synchronizer. Funkcia vyžaduje jeden argument a to číslo portu, na ktorom bude server počúvať na prichádzajúce spojenia. Ako názov funkcie napovedá, pri prvom zavolaní funkcie sa server zapne, pri ďalšom vypne. Spustenie klienta je veľmi podobné, keďže opäť stačí zavolať funkciu toggleClient() na objekte Synchronizer, ktorá akceptuje dva argumenty. Prvý typu string, kde reťazec je IPv4 adresa servera a druhý argument je číslo portu, na ktorom server počúva. Na začatie odpočúvania máme funkciu startIntercept() a na ukončenie stopIntercept(), ktoré sú bez argumentov. Podobne aj uvádzané toggle funkcie sa dajú nahradiť funkciami start/stop.

4) Ak bude aplikácia slúžiť ako klientská je potrebné na nej vytvoriť injektor, čo sa robí zavolaním funkcie createInjector() na objekte Synchronizer. Funkcia createInjector očakáva jeden argument typu Ogre::SceneManager*, čo je vlastne ukazovateľ na manažéra scény, ktorú chceme synchronizovať.

5) Pod zadefinovaním miesta pre injektor sa myslí buď vytvorenie nového FrameListener-a ak ešte v našej aplikácií neexistuje alebo vybratie si jedného s existujúcich v ktorom zavoláme funkciu inject() opäť na objekte Synchronizer.

6) Posledná vec, na ktorú nesmieme zabudnúť je pred koncom aplikácie vymazať injektora a to pomocou funkcie destroyInjector() zavolanej na objekte Synchronizer. Funkcia uvoľní zdroje, ktoré injektor počas svojho pôsobenia naalokoval.

Po vykonaní týchto 6 krokov a skompilovaní kódu aplikácie, pričom musíme použiť knižnice Ogre s implementovanou podporou pre synchronizáciu, bude aplikácia schopná synchronizovať svoju scénu s inými identickými aplikáciami.

3.7 Kompilácia Ogre knižníc so synchronizačným rozšírením

Ak chceme skompilovať Ogre s rozšírením vytvoreným v tejto práci, stačí použiť modifikované zdrojové kódy obsahujúce toto rozšírenie, prípadne aplikovať patch[6] súbory nachádzajúce sa v priečinku patch_files na priloženom médiu. Toto rozšírenie bolo implementované pre Ogre verzie 1.10 a nebolo testované na novších verziách. Preto je vhodné aplikovať patch súbory práve na túto verziu Ogre. Toto rozšírenie je závislé na knižniciach boost, preto je nutné aby čase kompilácie boli prítomné v systéme. Následne stačí postupovať podľa krokov uvedených na oficiálnej stránke Ogre projektu, s tým že pri vytváraní make súborov za pomoci nástroja cmake je potrebné zadefinovať konštantu OGRE_NETWORK_SYNCHRONIZATION.

[6]patch súbory vytvorené programom Mercurial - program na správu verzií

KAPITOLA 4

Testovanie, výsledky, zhrnutie

4.1 Podporovaná funkcionalita

V súčasnej implementácií podpory synchronizovania scény odpočúvame volania vyše 60 funkcií z 15. tried, zoznam modifikovaných súborov nájdeme v prílohe A. Množina týchto funkcií nám dovoľuje synchronizovať základné operácie na scéne a bola vybraná za účelom demonštrovať a otestovať funkčnosť konceptu, tak aby sa dali použiť na vytvorenie určitých typov aplikácií.

Implementované sú funkcie, ktoré nám umožňujú vykonať základné nastavenia scény ako napríklad nastavenie svetla, tieňov, efektu hmly. Ďalej podporujeme vytvorenie objektu takmer akéhokoľvek typu a to konkrétne:

- Light
- Entity
- BillboardSet
- ManualObject
- BillboardChain
- RibbonTrail
- ParticleSystem

Následne vieme tieto objekty umiestňovať na scénu, rotovať alebo pohybovať po scéne pozdĺž osí. Vieme spájať objekty, pripájať jeden na druhý, nastavovať im viditeľnosť, textúru. Komplikovanejšie operácie s týmito objektami nedokážeme synchronizovať. Výnimkou sú len objekty typu Light, ktorým vieme nastavovať rôznu farbu,

typ, smer a dosah svetla. Rozšírenú podporu synchronizácie majú aj objekty typu Entity, pri ktorých podporujeme základnú prácu s animáciami. Objekty typu Entity sú prevažne nejaké modely načítané zo súboru, vytvorené v nejakom externom modelovacom editore ako napríklad Blender alebo 3D Max. Tieto modely častokrát obsahujú aj nejakú animáciu, ktorú dokáže model vykonávať. My tieto animácie dokážeme spustiť a pohybovať sa v nich. Každá animácia má nejaké trvanie v časových jednotkách a pod pojmom pohybovať sa v animácií myslíme nastaviť animáciu na nejakú pozíciu na časovej osi.

Súhrn tejto podporovanej funkcionality nám umožňuje synchronizovať scény apliká- cií kde sa na scéne nachádzajú nejaké objekty, pohybujú sa po scéne Synchronizovať dokážeme aj základné špeciálne efekty, ktoré sú vytvárané za pomoci ParticleSystem, avšak tu sme limitované tým, že priame modifikácie týchto efektov počas behu aplikácie nedokážeme synchronizovať. Tieto efekty bývajú zväčša uložené v súboroch ako nejaké skripty. Ak sa takýto efekt len načíta a vykoná synchronizácia je možná. Pre lepšiu predstavivosť aké typy aplikácií vieme synchronizovať uvádzame krátku ukážku na priloženom médiu v priečinku nazvanom video.

4.2 Testované platformy

Systém Ogre by mal fungovať na platformách Linux, Windows, Mac OSX, IOS a Android. Podporu synchronizovania scény sme do systému Ogre doplnili tak aby sa v tomto smere nič nezmenilo. Na sieťovú komunikáciu sme použili knižnice Boost, ktoré su multiplatformové, iné externé knižnice sme nepoužívali. Pri písaní tohto rozšírenia sme navyše nepoužívali žiadne systémovo závislé funkcie z čoho vyplýva, že integrovaná podpora synchronizácie scény by mala fungovať na vymenovaných platformách. Aby sme si to ale potvrdili, spravili sme test funkčnosti a to tak že sme Ogre s našim doplnkom skompilovali pre systém Linux, Windows a Android.

Linuxová distribúcia, pod ktorou sme systém testovali bola Linux Mint 16 a Xubuntu 14.04. Použitý Windows bol Windows 7. Mobilné zariadenie na ktorom sme testovali aplikáciu bežalo na verzií androidu 4.2.2 a procesorovej architektúre armeabi-v7. Je nutné poznamenať, že Boost oficiálne nepodporuje Android z dôvodu širokého spektra verzií tohto systému a neschopnosti pokryť všetky tieto systémy testovaním. Napriek tomu sa Boost dá skompilovať aj pre android zariadenia.

Forma testovania

Na otestovanie nášho doplnku sme použili 3 zariadenia s aplikáciou používajúcou Ogre s implementovanou synchronizáciou scény, pričom každé zariadenie používalo iný systém. Následne sme všetky tri zariadenia pripojili na jednu lokálnu wifi sieť

vytvorenou za pomoci bežného router-a. Následne jedno zariadenie vždy slúžilo ako server, na ktorom scéna aplikácie slúžila ako obraz pre ostatné zariadenia. Zvyšné dva zariadenia sa správali ako klienti, ktorí sa pripojili na server a synchronizovali scénu. Takto sa postupne zariadenia vystriedali a to tak, že každý systém bol aspoň raz serverom a klientom.

Testovanie prebehlo v poriadku a nevyskytli sa väčšie problémy. Zaznamenali sme ale zvýšenú odozvu pri synchronizovaní scény, zhruba 0.5 - 1 sekundu. Ukazuje sa že toto oneskorenie môžu spôsobovať dve veci. Prvou príčinou môže byť fakt, že odpočúvanie volaní funkcií sa vykonáva na jednom vlákne a posielanie odpočutých informácií na druhom vlákne. Keďže tieto dva vlákna majú zdieľanú dátovú štruktúru na ukladanie správ, prístup k nej riešia za pomoci zámkov. V tomto prípade ale môže nastať situácia, že jedno vlákno prevezme kontrolu nad dátovou štruktúrou a neumožní prístup k dátam vláknu, ktoré sa stará o ich odoslanie. Dôsledok môže byť oneskorené synchronizovanie scény. Riešením tohto problému môže byť implementovanie nejakého druhu kontroly prístupu k dátovej štruktúre, ktorá by zamedzovala prevzatiu kontroly nad zdieľanou dátovou štruktúrou ak druhé vlákno nezískalo k nej prístup po dlhšiu dobu.

Druhá príčina, ktorá pravdepodobnejšie spôsobuje túto odozvu, je vytváranie objektov na scéne, najmä ak je potrebné načítať modely zo súboru. Na strane klienta oneskorenie vzniká v prípade, že klientské zariadenie, na ktorom beží aplikácia je pomalšie, alebo zaneprázdnenejšie a vytvorenie objektu na klientovi trvá dlhšie ako vytvorenie objektu na serveri. Rozdiel týchto časov sa pripočítava k oneskoreniu. Tento problém efektívne nevieme riešiť, avšak užívateľ, ktorý bude používať synchronizáciu scény po sieti sa môže tomuto problému vyhnúť tým spôsobom, že objekty vytvorí o niečo skôr ako ich bude používať, zobrazovať na scéne.

4.3 Ďalší vývoj

V tejto práci sa nám nepodarilo implementovať synchronizáciu pre celú funkcionalitu systému Ogre a tak jednou z vecí kde sa dá v práci pokračovať je doplniť podporu chýbajúcej funkcionality. Takisto tento doplnok poskytuje len minimalistické rozhranie na používanie a neposkytuje takmer žiadne možnosti voľby. Užívateľ by si napríklad mohol chcieť zvoliť synchronizovať len určitý objekt, alebo vytvoriť zoznam povolených, zakázaných objektov pre synchronizáciu.

Rovnako aj komunikačný protokol sa dá vylepšiť. V aktuálnej verzií len naviažeme spojenie na daný server a port a následne po tomto spojení komunikujeme. Ako možné rozšírenie funkcionality synchronizačného doplnku je spraviť automatické detekovanie

rovnakej aplikácie na sieti, poskytnúť uživateľovi možnosť určiť si kto môže, respektíve nemôže synchronizovať aplikáciu na základe znalosti hesla.

Ďalšou zaujímavou funkcionalitou na implementovanie môže byť povolenie pripojenie sa jednej aplikácie na viacero serverov a zobrazovať tak rôzne scény zo serverov do jednej scény na klientovi.

Záver

V tejto práci sa nám podarilo navrhnúť a implementovať funkčný prototyp podpory synchronizovania scény pre Ogre. Postupne sme sa zoznámili so systémom, vnútornou architektúrou súčasného kódu Ogre, analyzovali sme rôzne možnosti implementovania danej funkcionality a na záver sme vypracovali prvú funkčnú implementáciu. Implementovaný doplnok bol úspešne otestovaný na viacerých systémoch a zariadeniach čím sme demonštrovali funkčnosť a využitie doplnku.

Popri práci sa vyskytli rôzne problémy, niektoré z nich sa podarilo vyriešiť, iné si vyžadujú dodatočnú analýzu alebo majú komplexnejšie riešenie. Hoci sa nám podarilo vyvinúť použiteľný doplnok pre istý typ aplikácií určite sme nepokryli ani zďaleka celú funkcionalitu systému Ogre a preto práca na tomto doplnku sa dá považovať za otvorenú.

Dosiahnuté výsledky v tejto práci tvoria akýsi základ ďalšej práce pre dosiahnutie výsledku, ktorý by v budúcnosti mohol byť reálne prijatý komunitou používateľov Ogre a oficiálne prehlásený za podporovanú funkcionalitu systému.

Literatúra

[1] Beman Dawes, Rene R. David Abrahams A. David Abrahams: *Knižnica Boost*. Website, 1998-2007. – http://www.boost.org/

[2] Colvin, Greg ; Beman Dawes, Peter D. Darin Adler A. Darin Adler: *SharedPointer trieda*. Website, 1999 - 2013. – http://www.boost.org/doc/libs/1_55_0/libs/smart_ptr/shared_ptr.htm#Introduction

[3] Corporation, Oracle: *Data race definícia*. Website, 2010. – http://docs.oracle.com/cd/E19205-01/820-0619/geojs/index.html

[4] Kohlhoff, Christopher M.: *Acceptor trieda*. Website, 2013. – http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio/reference/ip__tcp/acceptor.html

[5] Kohlhoff, Christopher M.: *boost::Asio trieda*. Website, 2013. – http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio.html

[6] Kohlhoff, Christopher M.: *io_service trieda*. Website, 2013. – http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio/reference/io_service.html

[7] Kohlhoff, Christopher M.: *Objekt boost::asio::service trieda*. Website, 2013. – http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio/reference/io_service.html

[8] Ltd, Torus Knot S.: *AnimationState trieda*. Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1AnimationState.html

[9] Ltd, Torus Knot S.: *ColourValue trieda*. Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1ColourValue.html

[10] Ltd, Torus Knot S.: *createEntity() funkcia*. Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1SceneManager.html#a7bf9c5edb0ae3105c546bcad39252874

[11] Ltd, Torus Knot S.: *Entity trieda*. Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1Entity.html

[12] Ltd, Torus Knot S.: *FrameListener trieda.* Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1FrameListener.html

[13] Ltd, Torus Knot S.: *Konštruktor triedy ColourValue.* Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1ColourValue.html#acc033c4828ad9f7469269933d24850f5

[14] Ltd, Torus Knot S.: *Konštruktor triedy SceneNode.* Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1SceneNode.html#a952378bf8b65301aec723e020234be6b

[15] Ltd, Torus Knot S.: *Light trieda.* Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1Light.html

[16] Ltd, Torus Knot S.: *MovableObject trieda.* Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1MovableObject.html

[17] Ltd, Torus Knot S.: *Node trieda.* Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1Node.html

[18] Ltd, Torus Knot S.: *_notifyAttached() funkcia.* Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1Entity.html#aa00c4d0ae6c87888b0188bb7356d99dc

[19] Ltd, Torus Knot S.: *RenderSystem trieda.* Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1RenderSystem.html

[20] Ltd, Torus Knot S.: *ResourceManager class.* 2013

[21] Ltd, Torus Knot S.: *Root trieda.* Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1Root.html

[22] Ltd, Torus Knot S.: *SceneManager trieda.* Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1SceneManager.html

[23] Ltd, Torus Knot S.: *SceneNode trieda.* Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1SceneNode.html

[24] Ltd, Torus Knot S.: *startRendering funkcia.* Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1Root.html#a8eda253befda1255cbfd3b298f62449e

[25] Ltd, Torus Knot S.: *StringConverter trieda.* Website, 2013. – http://www.ogre3d.org/docs/api/1.9/classOgre_1_1StringConverter.html

Príloha A

Zoznam modifikovaných súborov:

- CMake/Templates/samples.cfg.in
- CMake/Templates/samples_d.cfg.in
- CMakeLists.txt
- OgreMain/CMakeLists.txt
- OgreMain/include/OgreAnimationState.h
- OgreMain/include/OgreBillboard.h
- OgreMain/include/OgreException.h
- OgreMain/include/OgreMovableObject.h
- OgreMain/include/OgreNode.h
- OgreMain/include/OgreRoot.h
- OgreMain/include/OgreViewport.h
- OgreMain/src/OgreAnimationState.cpp
- OgreMain/src/OgreBillboard.cpp
- OgreMain/src/OgreBillboardChain.cpp
- OgreMain/src/OgreBillboardSet.cpp
- OgreMain/src/OgreEntity.cpp
- OgreMain/src/OgreLight.cpp
- OgreMain/src/OgreMesh.cpp
- OgreMain/src/OgreMeshManager.cpp
- OgreMain/src/OgreMovableObject.cpp
- OgreMain/src/OgreNode.cpp
- OgreMain/src/OgreRibbonTrail.cpp
- OgreMain/src/OgreRoot.cpp
- OgreMain/src/OgreSceneManager.cpp
- OgreMain/src/OgreSceneNode.cpp
- OgreMain/src/OgreSkeleton.cpp
- OgreMain/src/OgreViewport.cpp
- Samples/Browser/include/SampleBrowser.h
- Samples/CMakeLists.txt
- Samples/Character/include/CharacterSample.h

Zoznam vytvorených súborov:

- OgreMain/include/OgreNetwork.h
- OgreMain/include/OgreSceneSynchronizer.h
- OgreMain/include/OgreSceneSynchronizerInjector.h
- OgreMain/include/OgreSceneSynchronizerServer.h
- OgreMain/src/OgreNetwork.cpp
- OgreMain/src/OgreSceneSynchronizer.cpp
- OgreMain/src/OgreSceneSynchronizerInjector.cpp
- OgreMain/src/OgreSceneSynchronizerServer.cpp
- Samples/NetworkSample/CMakeLists.txt
- Samples/NetworkSample/include/NetworkSample.h
- Samples/NetworkSample/src/NetworkSample.cpp