VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

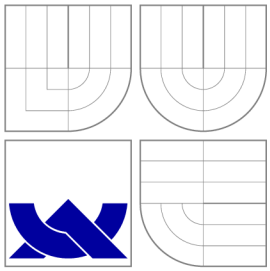
VIZUALIZACE A SIMULACE V GRAFICKÉ SCÉNĚ S VYUŽITÍM KNIHOVNY DELTA3D

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

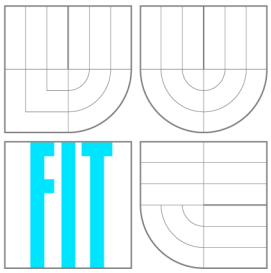
AUTOR PRÁCE
AUTHOR

LUKÁŠ MARTANOVIČ

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VIZUALIZACE A SIMULACE V GRAFICKÉ SCÉNĚ S VYUŽITÍM KNIHOVNY DELTA3D

VISUALISATION AND SIMULATION IN GRAPHICS SCENE USING DELTA3D

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LUKÁŠ MARTANOVIČ
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN PEČIVA, Ph.D.
SUPERVISOR

BRNO 2011

Abstrakt

Účelem práce je popis tvorby vizualizační a simulační aplikace s využitím knihovny Delta3D. V první kapitole se nachází popis struktury knihovny, následovaný seznámením se základními mechanizmy a prostředky enginu. Obsahem druhé a třetí kapitoly je následně příklad ruční tvorby jednoduché grafické scény a popis tvorby rozsáhlejší aplikace využívající širší spektrum možností poskytovaných knihovnou Delta3D.

Abstract

Goal of this thesis is to describe the process of creating a visualisation and simulation application using Delta3D. First chapter contains description of library structure, followed by familiarization with basic mechanics and features of the engine in chapter two. Content of third and fourth chapter is an example creation of simple graphics scene and description of development of more advanced application using wider spectrum of features offered by Delta3D library.

Klíčová slova

Delta3D, herní engine, simulační engine, 3D počítačové hry, tvorba počítačových her, vizualizace, simulace

Keywords

Delta3D, game engine, simulation engine, 3D computer games, computer game development, visualisation, simulation

Citace

Lukáš Martanovič: Vizualizace a simulace v grafické scéně s využitím knihovny Delta3D, bakalářská práce, Brno, FIT VUT v Brně, 2011

Vizualizace a simulace v grafické scéně s využitím knihovny Delta3D

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Pečivu, Ph.D.

.....
Lukáš Martanovič
14. května 2011

Poděkování

Rád bych poděkoval panu Ing. Janovi Pečivovi, Ph.D. za vedení, směrování, ochotu podělit se o svoje skušenosti a pomoc při tvorbě této práce.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Či už sa jedná o počítačové hry alebo armádne výcvikové simulácie, vývoj takto komplexných aplikácií by bol neúmerne nákladný a zložitý bez existencie vysokoúrovňových vývojových nástrojov, takzvaných *enginov*. Moderných 3D vizualizačných a simulačných enginov v súčasnosti existuje niekoľko desiatok, či dokonca stovák, veľkou nevýhodou značného množstva z nich však zostáva aj napriek širokým možnostiam, ktoré pri vývoji aplikácií poskytujú, uzavretosť ich kódu, resp. veľmi obmedzený prístup k funkcionalite na nízkej úrovni či nemožnost úprav a prispôsobenia samotného enginu. Jednou z alternatív, uberajúcich sa cestou integrácie rôznych open-source iniciatív na pôde vizualizácie a simulácie, je aj projekt Delta3D.

Jadrom obsahu tejto práce je popis herného a simulačného enginu Delta3D, jeho súčastí v aktuálne dostupnej distribúcii, mechanizmov a princípov fungovania a ukážka jeho použitia na príklade jednoduchej demonštračnej aplikácie (viď prílohy).

Kapitola 2

Delta3D

Projekt Delta3D vznikol a ústredie jeho vývoja sa v súčasnosti nachádza na Modeling, Virtual Environments and Simulation (MOVES) Institute pri Naval Postgradual School v Monterey, Kalifornia, USA. Prvá oficiálna verzia bola zverejnená v roku 2004 a vývoj odvtedy neustále pokračuje, pričom na ňom spolupracujú inštitúcie a jednotlivci z celého sveta. Pôvodným zámerom projektu bolo zjednotenie rôznych open-source iniciatív týkajúcich sa simulácie a vizualizácie pre potreby tvorby takzvaných *„Serious Games“* („vážnych hier“), teda výukových a výcvikových simulácií pre potreby námorníctva Spojených štátov amerických. V súčasnosti je výstup projektu, teda samotný engine, využívaný mnohými vládnymi aj súkromnými spoločnosťami a jednotlivcami po celom svete (niekoľko najvýznamnejších príkladov je možné nájsť na oficiálnych internetových stránkach projektu – [6]).

Obrázek 2.1: Delta3D

Jadrom je integrácia rôznych open-source projektov, ako OpenSceneGraph (OSG), OpenAL, Character Animation Library (Cal3D) a Open Dynamics Engine (ODE) a projektov ako Qt, Crazy Eddie's GUI, či Game Networking Engine (GNE) v jednotnom vysokoúrovňovom API, ktoré by zastrešovalo všetky jednotlivé zložky, zároveň ich však ponechalo voľne prístupné a umožnilo vytvárať aplikáciu od najnižšej úrovne. Dôraz je pritom kladený na prenositeľnosť aplikácií – oficiálne prebieha vývoj pre platformy Windows a Linux, existuje však aj podpora pre Mac OSX. Okrem spomenutých súčastí prináša množstvo nástrojov zjednodušujúcich výstavbu aplikácie. Najvýznamnejším spomedzi nich sú Simulation, Training and Game Editor (STAGE – bližší popis 2.2.4 na strane 14), Particle editor, či samostatný Object viewer.

Projekt momentálne zastrešuje GNU Lesser General Public License. U jednotlivých zložiek sa toto môže lýšiť, všetky sú však voľne šíriteľné.

2.1 Architektúra

Delta3D sa v zásade skladá z pomerne rozsiahleho množstva vzájomne previazaných frameworkov členených podľa typu funkcionality, ktorú zaobstarávajú. Kompletná pravidelne

Obrázek 2.2: Niektoré integrované projekty

aktualizovaná dokumntácia k API je voľne prístupná k nahliadnutiu na internetových stránkach projektu [9]. Závislosti jednotlivých „balíčkov“ znázorňuje obrázok 2.3 na strane 7.

2.1.1 Súčasti

dtABC

Application Base Components zabezpečuje, ako už názov napovedá, samotný beh aplikácie, ale napríklad aj prácu s jej oknom a podobne. Obsahuje tiež komponenty pre prácu s počasím (viditeľnosťou, oblačnosťou, ...).

dtAudio

Vysokoúrovňová zvuková funkcionalita a rozhranie pre prácu so systémom OpenAL.

- 2D/3D zvuk.
- Úplná kontrola zvuku – modulácia, pozícia, pozícia „bodu načúvania“, dopplerov efekt, prehrávanie.

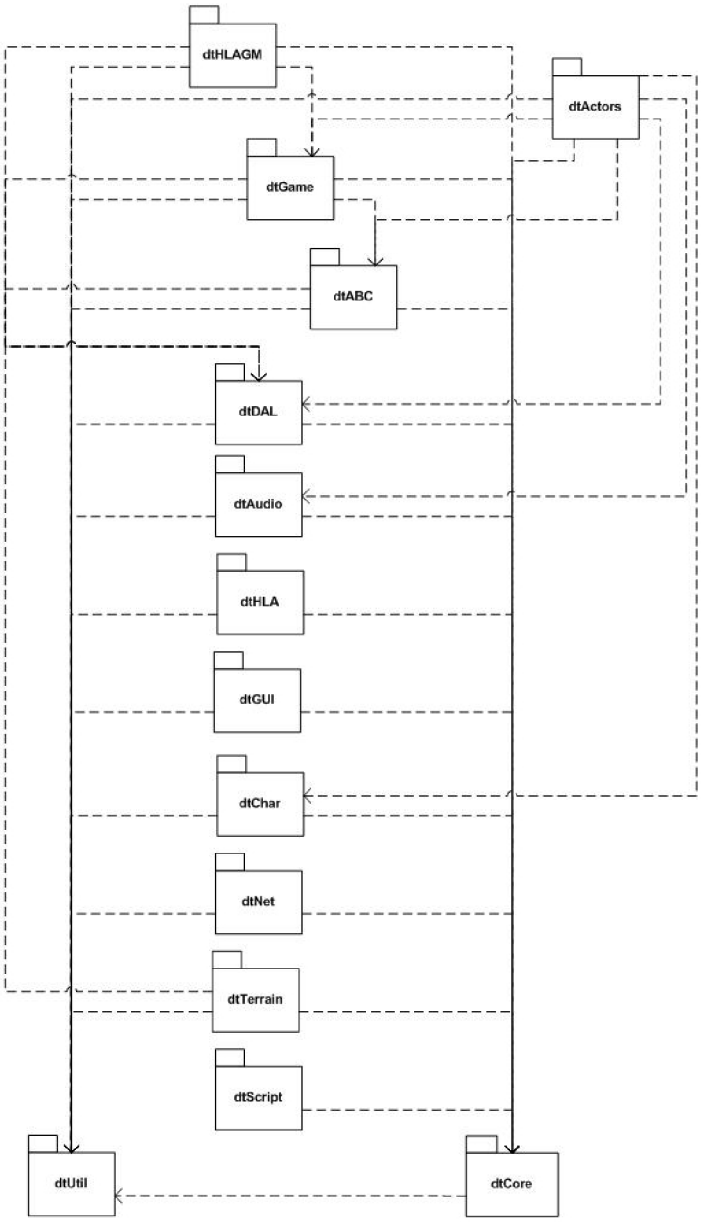
dtCore

dtCore je framework pre základnú prácu s vizualizáciou v 3D grafickej scéne. Zapuzdruje samotné renderovanie scény, k čomu využíva OpenSceneGraph, resp. OpenGL.

- Mapovanie I/O zariadení a na nich závislé pohybové modely.
- Kamera (vrátane práce s viacerými pohľadmi na scénu, resp. s viacerými oknami súčasne).
- Osvetlenie, environment, použitie particle systémov.
- Fyzika objektov v grafickej scéne, výpočet vzájomných kolízií.

dtDAL

DAL alebo Dynamic Actor Layer zabezpečuje správu projektov, spracovanie máp vytvorených v editore STAGE, manažment knižníc a prístup k vlastnostiam a logike implementácie Actors prostredníctvom Actor Proxy (2.2.2 – 12, resp. 2.2.3 – 13).



Obrázek 2.3: Medzitriedne závislosti. Súčasti sú závislé na tých, ku ktorým od nich smerujú šípky.[10]

dtGame

dtGame tvorí komplexnú architektúru pre tvorbu simulačnej, resp. hernej aplikácie.

- Manažment Game Actors.
- Game Manager a jeho prispôsobenie alebo rozšírenie pomocou užívateľských komponent (2.2.1 – 9).
- Infraštruktúra komunikácie herných objektov - tvorba, odosielanie, routovanie a spracúvanie správ, informácie o herných udalostiach, úlohách.

dtGUI

V súčasnosti sa práca s GUI v Delta3D sústreďuje okolo CEGUI, čo je aj jadrom tohto frameworku. Zámerom vývojového týmu je však aj čoraz rozsiahlejšia spolupráca s Qt.

dtHLAGM

Tento framework poskytuje interface ku HLA networkingu. Sem spadá integrácia s uživateľsky špecifickým RTI, prepočet koordinátov, komunikácia HLA entít a pod.

dtNet

Vysokoúrovňové API pre tvorbu multiplayer súčastí hier alebo simulácií. Umožňuje spoľahlivý/nespoľahlivý prenos dát v rámci klient/server architektúry.

dtPython

dtPython umožňuje prístup k API Delta3D skrze skriptovací interface Pythonu. Framework je závislý na knižniciach Pythonu a knižnici Boost, ktoré niesú súčasťou distribúcie Delta3D a je teda nutné zabezpečiť ich prítomnosť pre jeho využitie.

dtTerrain

Rozsiahla práca s terénom v grafickej scéne. Primárnym určením je práca s veľkými a/alebo dynamicky tvorenými hernými scénami.

- Načítanie, renderovanie a dekorácia terénov.
- Procedurálne umiestňovanie vegetácie, SOARX algoritmy pre dynamické renderovanie LOD.
- Paged terrain, GEOTIFF.

dtUtil

Pomocná funkcionalita, ako zápis rôznych logov, spracovanie XML súborov, súborových ciest alebo manažment knižníc.

Ďalšie súčasti

Doposiaľ uvedené súčasti enginu Delta3D tvoria akési jadro, umožňujúce vytvorenie plnohodnotnej aplikácie a sú najrozsiahlejšie a najkomplexnejšie. Okrem nich je však súčasťou projektu aj niekoľko ďalších:

- dtActors – Knižnica Actorov reprezentujúcich väčšinu tried obsiahnutých v dtCore, resp. dtABC.
- dtAI – Framework pre prácu s umelou inteligenciou, A* algoritmy a pohyb medzi waypointmi v scéne.
- dtAnim – Animácia skeletálnych modelov s využitím Cal3D.
- dtDIS – Framework pre podporu protokolu DIS.

2.1.2 Delta3D-Extras

Akýmsi rozšírením iniciatívy Delta3D je zjednotenie mnohých projektov vychádzajúcich z Delta3D, ktoré však niesú vyvýjané alebo udržiavané ako súčasť iniciatívy pod súhrnným názvom Delta3D-Extras. Jedná sa viac-menej o záležitosť komunity umožňujúcu vzájomné zdieľanie rozšírení a projektov, ktoré z Delta3D vychádzajú. Patria sem napríklad:

- dtVRPN – Využitie Virtual Reality Peripheral Network.
- dtChar – Projekt zaoberajúci sa animáciou skeletálnych modelov s pomocou ReplicantBody.
- dtScript – Začlenenie JavaScriptu.

a mnoho ďalších.

Spločnú základňu pre Delta3D-Extras je možné momentálne nájsť ako projekt hostovaný na SourceForge [7].

2.1.3 Dostupná distribúcia

Najaktuálnejšou release verziou enginu je verzia 2.5.0, vydaná v novembri 2010. Je dostupná v podobe inštalátora predkompilovaného SDK pre Visual Studio 2008 alebo v podobe balíčka zdrojových kódov.

Súčasťou distribúcie sú všetky vyššie spomenuté súčasti a ich závislosti (dependencies), ako aj pomerne rozsiahla zbierka príkladov demoštrujúcich jednotlivé možnosti enginu a vzorové dáta na ktorých je možné testovať vytvárané aplikácie – niekoľko predpripravených 3D modelov, textúr, zvukov a ďalšieho obsahu.

2.2 Mechanizmy

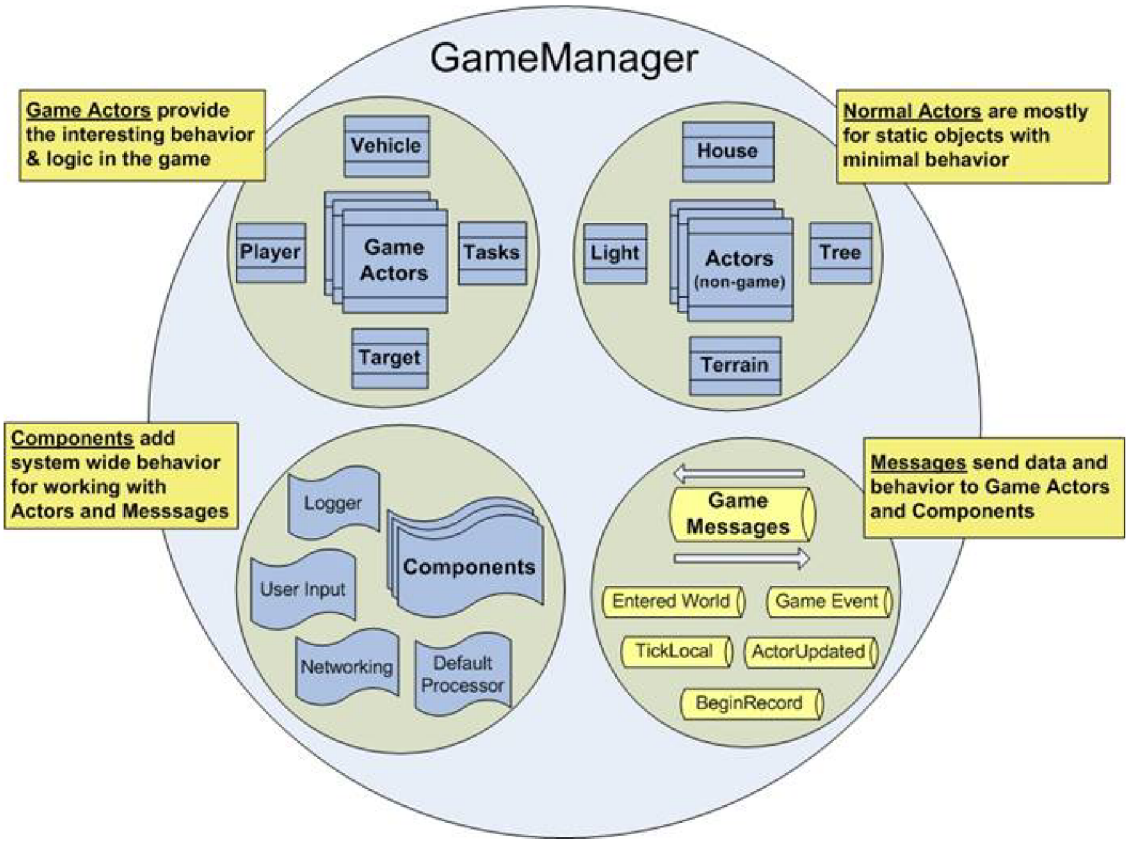
Pred tým, ako je možné pustiť sa do popisu štruktúry a procesu tvorby aplikácie v prostredí Delta3D, je vhodné zamerať pozornosť na princípy fungovania a mechanizmy poháňajúce činnosť enginu ako takého. Obsahom nasledujúcej kapitoly je teda rozbor vnútornej činnosti a prvkov enginu, ktorých využitie je v princípe nutnosťou pre akúkoľvek pokročilejšiu aplikáciu.

Schémy a diagramy využité v tejto kapitole boli prebraté z oficiálnych stránok projektu Delta3D [5].

2.2.1 Game Manager

Game Manager je možné zjednodušene označiť ako to, čo sa bežne rozumie pod „logikou herného enginu“. Jedná sa o jadro, starajúce sa o manažment herných máp, entít v nich obsiahnutých (Actors, Game Actors – 2.2.2) vrátane ich priestorovej a logickej organizácie, komponent a mechanizmu správ a komunikácie medzi nimi. V dôsledku teda predstavuje akúsi abstraktnú vrstvu, umužňujúcu sa pri vývoji aplikácie odtrhnúť od mechanizmov nižšej úrovne (napr. pre-frame, frame a post-frame eventy) a sústreďovať sa na špecifické fungovanie konkrétnej hry alebo simulácie.

Samotný Game Manager (ďalej v texte ako GM) v neupravenej podobe vykonáva akcie spoločné všetkým aplikáciám. Špecifickú funkcionalitu, pravidlá hry či simulácie, ako aj logiku správania sa herných entít, je samozrejme nutné dotvoriť. Architektúra GM našťastie umožňuje jeho jednoduchú rozšíriteľnosť a prispôsobivosť. Autori projektu sami na



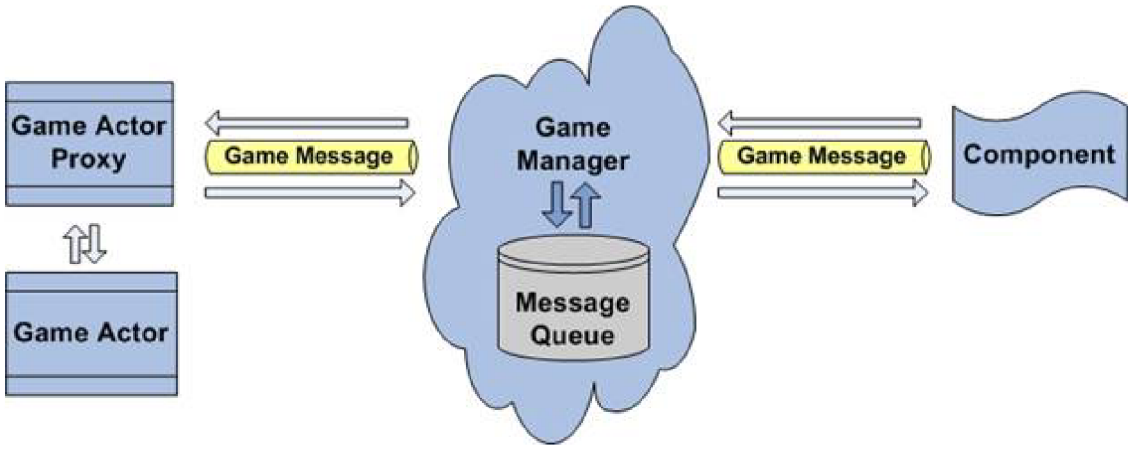
Obrázek 2.4: Štruktúra Game Manageru

oficiálnych internetových stránkach odporúčajú pokiaľ je to možné vyhýbať sa tvorbe tried odvodených od GameManager a jeho funkcionalitu radšej prispôsobovať pomocou vlastných účelovo špecifických komponent [8]. Takto je zaručená vyššia miera flexibility pri využívaní možností GM (bližší popis viď 2.2.1 – 12).

Mechanizmus zasielania správ

Základnou metódou komunikácie medzi všetkými Componentmi a Actormi obsiahnutými v aplikácii je *Game Message*. Je možné ich využť či už pre výmenu dát, zmenu hodnôt vlastností jednotlivých Actorov, oznamy o zmene ich stavu, alebo výskyte rôznych dôležitých herných udalostí. Pre rôznorodé informácie o stave simulácie a udalosti, ktorých výskyt je v réžii samotného GM, ako napríklad zmena stavu načítania hernej mapy, vypršanie časovača, či informácia o začatí cyklu simulácie, využíva engine množstvo predpripravených typov správ (napr. INFO_MAP_LOADED, INFO_ACTOR_CREATED, INFO_TIMER_ELAPSED atp.). Samozrejme je možné došpecifikovať vlastné typy, nesúce informácie špecifické pre nami vytváranú aplikáciu.

Tvorba, zasielanie a správne doručenie správ všetkým adresátom je primárnou úlohou Game Manageru. Všetky užívateľské typy správ, ktoré plánujeme v aplikácii využívať, je nutné najprv registrovať v *MessageFactory*, ktorá je súčasťou GM. Potom môže ktorýkoľvek Actor (prostredníctvom svojho Proxy) alebo Component jednoducho vytvoriť správu volaním `CreateMessage()`, kde ako parameter uvedieme jeden z typov správy, ktoré MessageFactory pozná. Po vytvorení správy a jej prípadnej úprave (napríklad parametrov identifi-



Obrázek 2.5: Mechanizmus zasielania správ

kujúcich odosielateľa alebo Actora, o ktorom konkrétna správa je) volaním `SendMessage()` alebo `SendNetworkMessage()` (ak chceme, ako už názov napovedá, odoslať správu nie len lokálne) dosiahneme jej zaradenie do fronty správ čakajúcich na odoslanie a GM sa následne postará o jej doručenie všetkým odberateľom daného typu správ, akonáhle dôjde na rad. Components odoberajú všetky správy, naproti tomu Actos sa musia pre odber každého typu správy registrovať.

Spracovanie správ v komponente potom zabezpečuje metóda `ProcessMessage()` (resp. `DispatchNetworkMessage()`), kde dôjde k odfiltrovaniu podľa typu doručenej správy a vyvolaniu zodpovedajúcej reakcie.

Rovnako prostredníctvom metódy `ProcessMessage()` reagujú na príjem správ aj Actors. Pre všetko, čo je nutné vykonávať periodicky (najčastejšie s každým frameom) slúži odchytávanie a spracúvanie tzv. Tick správ (2.2.1 – 11). Registráciu pre odber jednotlivých typov správ pre daný druh Actora, ako aj určenie alternatívnej obslužnej metódy pre konkrétnu správu (implicitne je to už spomenutá `ProcessMessage()`), zabezpečuje jeho Proxy v rámci metódy `OnEnteredWorld()` (2.2.2 – 12).

Tick

Tick je v terminológii autorov Delta3D vlastne jeden krok simulačného cyklu, teda hlavného cyklu behu aplikácie. Ak neberieme do úvahy viacvláknové simulačné cykly alebo optimalizačné prispôsobenia enginu, zodpovedá prakticky vykresleniu jedného framu.

Ak teda niektorý z Actorov prítomných v aplikácií musí reagovať pri každom „otočení cyklu“ (napr. kontrola a reakcia pri detekcii kolízií), musí existovať spôsob, ktorým ho GM vyzve k vykonaniu každého ďalšieho kroku. Túto výzvu zabezpečuje Game Manager rozosielaním správ TICK_LOCAL, resp. TICK_REMOTE, ktorých obsahom je okrem iného napríklad aj aktuálny simulačný čas. Tick správy niesú spracúvané tak ako štandardné správy pomocou `ProcessMessage()`, ale majú vyhradené metódy `OnTickLocal()`, resp. `OnTickRemote()`.

Mechanizmus rozosielania správ je pre potreby „tikania“ využitý najmä kvôli snahe zabrániť zbytočnému volaniu obslužných metód pre tick na Actors, ktorý reagujú len na špecifické udalosti, alebo sú celkom statický (napr. rôzne dekoračné objekty v scéne apod.). Preto GM nevyužíva jednoducho volanie metód `OnTickLocal()` či `OnTickRemote()` u každého Actora, ale rozosielanie správ, pre ktoré sa záujemcovia musia registrovať.

Game Manager Component

Game Manager Component (zjednodušene Component) sa dá chápať ako aplikačne špecifické rozšírenie Game Manageru. Využíva sa pre vykonávanie akcií a reakcie na podnety, ktoré sú „celoaplikačného rozmeru“ a nestačí ich vykonať jednotlivými hernými entitami, resp. o zabezpečenie funkcionality, ktorá má byť prítomná (z pohľadu koncového užívateľa) od spustenia aplikácie až po jej ukončenie. Jedná sa napríklad o reakcie na užívateľský vstup, sieťové rozhranie, kresbu grafického užívateľského rozhrania, či vynucovanie pravidiel hry a jej vedenie. Každý Component je odberateľom všetkých správ, ktoré prejdú Game Managerom. Ako už bolo spomenuté, príčinou existenicie konceptu Componentov je najmä flexibilita pri vývoji aplikácií, kedy jadro GM zostáva stále rovnaké, správanie a logiku aplikácie je ale možné jednoducho rozšíriť a prispôsobiť.

Každý komponent, s ktorým má Game Manager spolupracovať, je nutné ručne pripojiť. Deje sa tak zväčša v rámci vykonávania `OnStartup()`, čo je jedna z inicializačných metód Game Manageru (bližie viď 4.6 – 38). Dôležitým komponentom, ktorý je nevyhnutnou podmienkou pre správne fungovanie každej aplikácie je *Default Message Processor* (DMP), ktorý zabezpečuje spracovanie a prácu so základnými správami, ako sú informácie o načítaní hernej mapy, vstupe Actorov do hry a podobne. Aj keď je toto vlastné v zásade všetkým aplikáciám postaveným na využívaní GM, nieje DMP jeho integrálnou súčasťou, aby sa umožnila prispôsobivosť aj v tejto oblasti.

Každému Componentu je pri jeho pripájaní ku GM nutné priradiť vhodnú prioritu. Priorita Componentov určuje poradie, v ktorom budú spracúvať správy od GM. Existuje päť úrovní – HIGHEST, HIGHER, NORMAL, LOWER a LOWEST. Tak napríklad spomenutý DMP zabezpečuje nevyhnutné služby, bez ktorých by aplikácia nemohla správne fungovať a vykonáva to, na čom zväčša závisia reakcie ostatných Componentov, preto je vhodné určiť mu najvyššiu prioritu.

2.2.2 Actor, Actor Proxy

Actor, Game Actor

Pod pojmom *Actor* si je možné predstaviť v zásade akýkoľvek herný objekt, s ktorým má byť schopný pracovať editor STAGE, či už sa jedná o jednoduchý statický model, ovládateľné vozidlo, hráčskeho avatara alebo spínač reagujúci na blízkosť hráča. *Game Actor* je potom taký typ Actora, ktorý je navrhnutý s ohľadom na spoluprácu s Game Managerom. Základný rozdiel spočíva v tom, že kým Actor je aj akýkoľvek neherný, viac či menej statický objekt, pojem Game Actor označuje Actorov, ktorý su v aplikácii nejakým spôsobom aktívni, majú schopnosť reagovať na správy a tick a interagovať so svojim okolím. Actor (všeobecne) môže byť z pohľadu Game Manageru chápaný ako lokálny (operuje a manipuluje s ním priamo tento GM), alebo vzdialený (remote), kedy ho GM vidí a vníma, ale Actor samotný je manipulovaný iným GM. Rozdiel medzi Actorom a Componentom potom možno chápať tak, že kým Actor je objekt, ktorý sa „fyzicky“ nachádza v konkrétnej hernej mape, Component je mimo „reality hry“, rozprestretý na úrovni celej aplikácie.

Actor Proxy

Rozhranie medzi vnútornou štruktúrou Actora, resp. Game Actora a Game Managerom alebo STAGE tvorí *Actor Proxy*. Prítomnosť rozhrania je nutná, keďže Actor môže predstavovať prakticky akýkoľvek reálny či fiktívny element reálneho sveta od statického objektu po pohyblivé objekty s vlastnou umelou inteligenciou a množstvom vlastností a schopností.

Zároveň platí, že tak GM, ako aj STAGE pracujú s jednotlivými entitami tvoriacimi hernú mapu generický a nieje možné, aby mali informácie o vnútornom fungovaní každého jednotlivého užívateľského typu Actora. Proxy teda sprístupňuje dátovú štruktúru Actora, ku ktorému prislúcha, pomocou množiny vlastností (properties) a k nim prináležiacim metódam prostredníctvom jednoduchého rozhrania založeného na metódach `SetProperty()`, resp. `GetProperty()`. Na podobnom princípe je založené aj priradzovanie zdrojov (resources) jednotlivým Actorom, ako napr. konkrétny model, či zvukový súbor, prostredníctvom `SetResource()`, resp. `GetResource()`.

Okrem sprístupnenia vlastností Actora pre potreby editoru (k čomu slúži metóda `BuildPropertyMap()`) medzi ďalšie dôležité úlohy Actor Proxy patrí tiež registrácia pre odber jednotlivých typov správ od GM (čo prebieha zväčša popri inicializácii v rámci metódy `OnEnteredWorld()`) a samotná inštanciácia Actora (`CreateActor()`, v rámci ktorej je obvykle vytvorený a podsunutý metóde `SetActor()`).

Vytvoreného Actora je potom po náležitej registrácii (k tomuto účelu slúži trieda `ActorsPluginRegistry`, resp. jej derivát, obsahujúci objekt `mActorsFactory`, ktorý je nutnou súčasťou každej užívateľskej knižnice) možné nielen obsluhovať za pomoci GM, ale je tiež možné priamo ho vkladať a hodnoty jeho sprístupnených atribútov upravovať pomocou editoru STAGE (viď 2.2.4). Zoznam všetkých Actorov v aktuálnej mape je potom kedykoľvek prístupný a prehľadávateľný v aplikácii cez volanie príslušných metód Game Manageru.

Napriek tomu, že od prvých verzií návrhu došlo k niekoľkým zmenám, základný popis návrhu a fungovania Game Manageru, vrátane popisu ideí stojacich za konceptom komunikácie, Actors a Components obsahuje Software Design Document k verzii 1.0 z októbra 2005 [4].

2.2.3 DAL - Dynamic Actor Layer

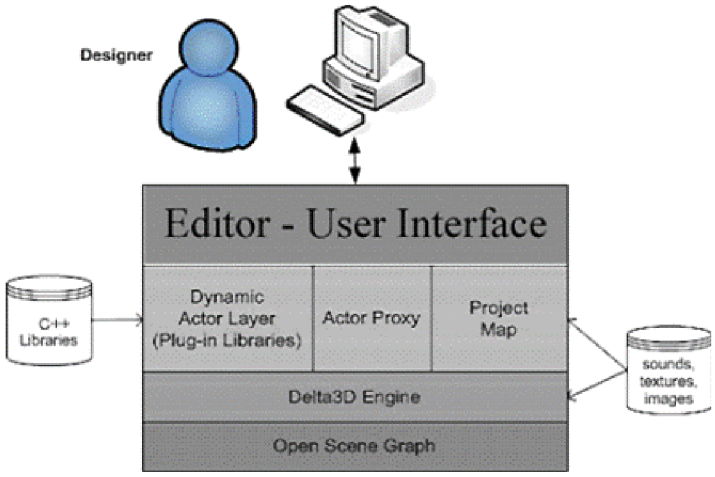
Ako vyplýva z horeuvedeného, editor STAGE ani Game Manager v zásade nemajú možnosť vidieť vnútornú štruktúru jednotlivých súčastí tvorenej aplikácie a operujú len s tým, čo je v rámci istých pravidiel zviditeľnené prostredníctvom špecifických Actor Proxy. Mechanizmy, ktoré ale rozoznajú a integrujú užívateľské knižnice obsahujúce tieto prostiredky, umožňujú ich správu, rozoznanie a manipuláciu (a tým okrem iného znovupoužiteľnosť už vytvorených typov Actorov) sa súhrnne označujú ako *Dynamic Actor Layer*, skrátene DAL. Podrobným rozborom vrátane diagramov tried a sekvenčných diagramov popisujúcich štruktúru a prácu DAL sa zaoberá Software Design Document k verzii 1.2 z augusta 2005 prístupný na internetových stránkach projektu Delta3D [2]. DAL je možné na základe poskytovanej funkcionality rozčleniť na tri základné časti: Library Manager, Actor Proxy a Project/Map.

Library Manager

Library Manager je časť zodpovedná za správu pripájaných užívateľských knižníc. Jeho súčasťou je machanizmus rozoznávajúci, aké typy užívateľských objektov (Actorov) sú v rámci knižnice dostupné a ako ich vytvoriť.

Actor Proxy

Actor Proxy je, ako už bolo spomenuté, prostriedok komunikácie medzi GM (alebo STAGE) a užívateľskými objektmi – Actormi. (Detailnejšie viď 2.2.2 – 12.)



Obrázek 2.6: Dynamic Actor Layer

Project/Map

Treťou úlohou DAL je správa projektu ako takého. To zahŕňa manažment dostupných zdrojov (modelov, zvukových a obrazových súborov, textúr, particle systémov, . . .) v adresárovej štruktúre projektu a zabezpečenie ich sprístupnenia z pohľadu editoru STAGE (resp. Game Manageru), načítanie a ukladanie máp a konfiguračných súborov.

2.2.4 STAGE

Simulation, Training and Game Editor je hlavným pomocným nástrojom pre tvorbu plnohodnotných aplikácií v prostredí Delta3D umožňujúcim jednoduché a rýchle vyskladanie hernej mapy.

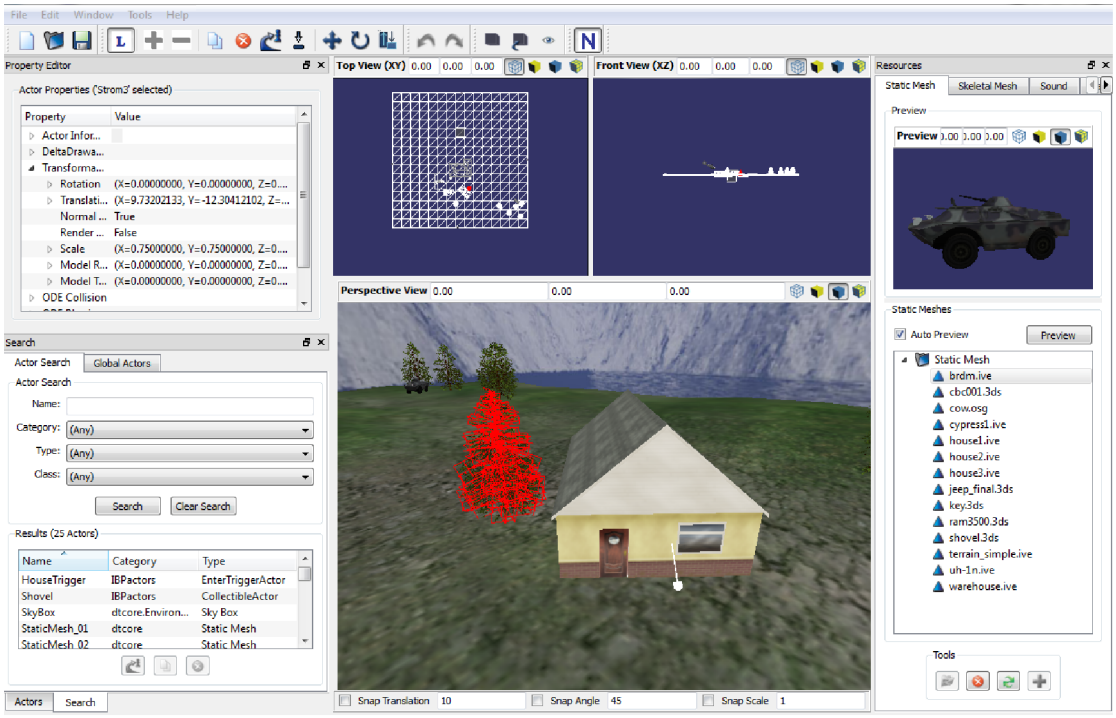
Prvým krokom pri tvorbe aplikácie s využitím editoru STAGE je zvolenie kontextu, resp. vytvorenie nového projektu. Ten určuje cesty, kde sa budú vytvárať všetky v ňom využité mapy a zdroje k tomu použité, čím určuje implicitnú adresárovú štruktúru projektu a zároveň jednotlivé projekty odlišuje. Akonáhle je zvolený kontext projektu, je pre sprístupnenie väčšiny funkcionality editoru nutné vytvoriť aspoň jednu hernú mapu.

Mapa samotná je uložená v XML formáte (.dtmap) skladajúcom sa z troch základných častí. Prvou je hlavička nesúca informácie o verzii použitého editoru, časovú známku a pod., nasleduje špecifikácia jednotlivých knižníc využitých v danej mape a nakoniec sekcia Actors popisujúca objekty v nej vložené prostredníctvom vlastností (properties) a ich hodnôt. [3]

Prostredie editoru

Po vytvorení hernej mapy sa sprístupnia možnosti jej editácie. Prostredie hlavnej obrazovky editoru ukazuje obrázok 2.7 na strane 15.

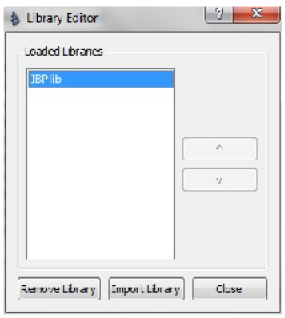
V pravej časti sa nachádza *resource editor*, prostredníctvom ktorého je možné k projektu pripájať všetky dáta, ktoré do neho plánujeme začleniť – hotové modely (aktuálna verzia podporuje formáty .ive, .osg, .3dm, . . .), textúry, zvuky, particle systémy a podobne. Umožňuje tiež rýchly náhľad na práve vybraté zdroje. Vložené zdroje sa ukladajú v adresári, ktorý bol zvolený ako kontext projektu pri jeho zakladaní, prehľadne usporiadané podľa typu. Samostatný adresár potom obsahuje jednotlivé herné mapy obsiahnuté v projekte.



Obrázek 2.7: Hlavné okno editora STAGE

V strednej časti je potom súbor náhľadov na aktuálne tvorenú mapu. Implicitne je rozdelený na štyri časti, reprezentujúce pohľad zhora (rovina XY), zpredu (XZ), zboku (XY) a perspektívny náhľad. Toto rozloženie je samozrejme prispôsobiteľné. Pomocou voľného približovania/oddaľovania, natáčania a posunu náhľadov je možné jednoducho sa zorientovať v mape a získať predstavu o v aplikácii zobrazovanej scéne.

Ľavá časť obsahuje knižnicu actorov, ktorých je možné pri tvorbe použiť, zoznam a vyhľadávač všetkých actorov nachádzajúcich sa v mape a v hornej časti *Property editor* umožňujúci konfiguráciu vlastností v mape sa nachádzajúcich actorov, ktoré boli sprístupené prostredníctvom ich proxy. Absolútnou nutnosťou je potom import knižníc obsahujúcich užívateľské typy Actorov (2.2.2), pre ktorý editor ponúka veľmi jednoduché rozhranie.



Obrázek 2.8: Pridávanie užívateľských knižníc do editora STAGE

Kapitola 3

Ručná tvorba scény

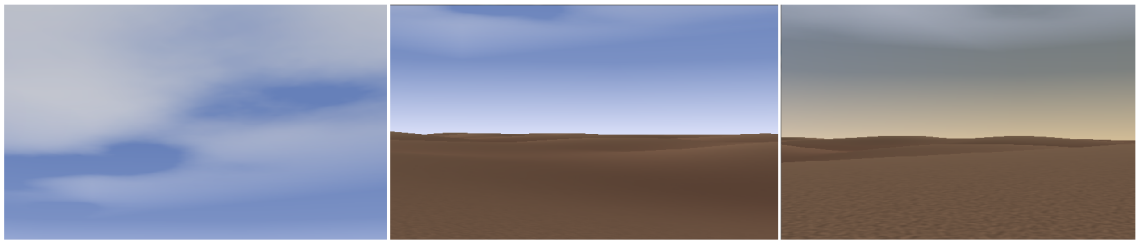
Pre zoznámenie sa s princípmi tvorby a zobrazenia jednoduchej grafickej scény a lepšie pochopenie mechanizmov nižšej úrovne, ktoré za nimi stoja, je vhodné bližšie sa pozrieť na spôsob, ako jednoduchú aplikáciu vytvoriť takpovediac „ručne“, bez využitia možností poskytovaných editorom STAGE a Game Managerom.

3.1 Application a Main

Základom, na ktorom je postavená takmer každá Delta3D aplikácia, je trieda `dtABC::Application`, ktorá obsahuje všetky nevyhnutné prostriedky pre zobrazenie a konfiguráciu okna aplikácie, vytvorenie a zobrazenie grafickej scény a užívateľský vstup.

Pre jednotlivé objekty, ktoré chceme vložiť do zobrazovanej scény, využijeme triedu `dtCore::Object`, ktorá umožňuje reprezentovať virtuálny objekt, ktorý je zobraziteľný, pohybovateľný a má definované fyzikálne vlastnosti ako hmotnosť, ťažisko a podobne.

Knižnica Delta3D poskytuje pre pohyb objektov v grafickej scéne niekoľko predpripravených pohybových modelov (*MotionModel*), ktoré zabezpečujú transformáciu objektu v priestore na základe užívateľského vstupu. Tieto napodobňujú modely pohybu typické pre rôzne druhy herných aplikácií, ako je pohyb kamery v strategických hrách videných zvrchu a podobne. Pre názornosť v tomto príklade využijeme `dtCore::FPSMotionModel`, ktorý je typický pre akčné hry videné z pohľadu prvej osoby a umožňuje pohyb vpred a vzad so sledovaním terénu a úkroky do strán s využitím klávesnice a voľné rozhliadanie sa (resp. rotáciu) pomocou myši.



Obrázek 3.1: Terén a prostredie: SkyDome, InfiniteTerrain, Efekty dennej doby a počasia.

Ako povrch, na ktorom sa budú nachádzať objekty, ktoré do scény umiestnime, je možné využiť obyčajný Object, ktorému priradíme vhodný model terénu. Pre jednoduchosť ukážeme využitie nekonečného generovaného povrchu (`dtCore::InfiniteTerrain`).

Osvetlenie a viditeľnosť, resp. rôzne efekty oblohy a prostredia všeobecne, ako napríklad hmlu zabezpečuje trieda `dtCore::Environment`.

V prípade, že sa rozhodneme scénu oživiť napríklad pridaním zvuku (povedzme hukot motora naštartovanej helikoptéry, okolo ktorej budeme prechádzať), poslúži nám pre jeho načítanie a konfiguráciu jeho prehrávania `dtAudio::Sound`.

Pre zostavenie scény sa obvykle využíva metóda `Config()`, ktorá zabezpečuje inicializáciu aplikácie. Ak by sme chceli začleniť funkcionalitu podobnú Tick bez využitia Game Manageru, resp. pre čokoľvek, čo je nutné vykonávať s každým frameom pri behu aplikácie, je možné zasiahnuť do simulačného cyklu. Na základnej úrovni sa tento skladá z krokov *EventTraversal*, *PostEventTraversal*, *PreFrame*, *CameraSynch*, *FrameSynch*, *Frame*, a *PostFrame*. Zásahy je vhodné robiť najmä v časti PreFrame, v ktorej obvykle prebieha takmer všetká aplikačná logika a kde inak väčšinu svojich úloh vykonáva aj Game Manager, prípadne PostFrame. EventTraversal a Frame sú naopak vyhradené pre úkony OpenSceneGraphu a nieje vhodné do nich zasahovať. Trieda, ktorá bude tvoriť základ aplikácie potom môže vyzerať napríklad takto:

```
class ExampleApp : public dtABC::Application
{
public:
        ExampleApp(const std::string& configFilename = ''config.xml'');

        virtual void Config();
        virtual void PreFrame(const double deltaFrameTime);
protected:
        ~ExampleApp();
private:
        dtCore::RefPtr<dtCore::InfiniteTerrain> mTerrain;
        dtCore::RefPtr<dtCore::Environment> mEnvironment;
        dtCore::RefPtr<dtCore::Object> mHelicopter;
        dtCore::RefPtr<dtAudio::Sound> mEngineSound;
        dtCore::RefPtr<dtCore::FPSMotionModel> mFPSMM;
};
```

`dtCore::RefPtr` je template pre smart pointer starajúci sa o počítanie referencií na odkazovaný objekt a je dobrou praxou ho využívať. Funkcia main potom typicky vyzerá nasledovne:

```
int main()
{
        dtCore::SetDataFilePathList( ''..;'');
        dtCore::RefPtr<ExampleApp> app = new ExampleApp();
        app->Config();
        app->Run();
        return 0;
}
```

Po úvodnom nastavení ciest pre vyhľadanie dátových zdrojov (modelov a zvukov), ktoré využijeme pri zostavovaní scény a vytvorení inštancie našej triedy nasleduje konfigurácia okna a samotné zostavenie scény volaním metódy `Config()` a napokon volaním `Run()` dôjde k spusteniu behu nekonečného cyklu aplikácie.

3.2 Zostavenie scény

Zostavenie scény prebieha v rámci vykonania metódy `Config()`. Tá má jediný argument, ktorým je súboru s konfiguračnými dátami. Jedná sa o XML dokument obsahujúci nastavenia ako základná veľkosť okna aplikácie, informácia, či sa bude pracovať vo fullscreen móde a podobne.

Aby sme mohli v aplikácii využiť zvukové efekty, musíme najskôr vytvoriť a konfigurovať rozhranie k OpenAL, čo je súčasť enginu zodpovedná za prácu so zvukom na základnej úrovni.

```
dtAudio::AudioManager::Instantiate();
dtAudio::Listener* ear = dtAudio::AudioManager::GetInstance().GetListener();
GetCamera()->AddChild(ear);
```

Ako je vidieť, nevytvárame priamo objekt `dtAudio::Listener`, ale pracujeme len s jeho rozhraním, ktoré je súčasťou AudioManageru. Samotný Listener je transformovateľný objekt (odvodený z triedy `dtCore::Transformable`) a preto ho môžeme priradiť ako potomka akémukoľvek inému transformovateľnému objektu, čím zaručíme ich previazanosť. Ak teda v tomto vzťahu rodič napríklad zmení polohu v scéne, spločne s ním zmenia svoju polohu aj jeho potomkovia. Keďže v tomto prípade budeme pohybovať kamerou a chceme vnímať aj zvuky z jej pozície, vytvoríme medzi ňou a Listenerom takýto príbuzenský vzťah.

Následne môžeme prístúpiť k tvorbe scény. Vytvoríme najskôr terén a následne Environment.

```
mTerrain = new dtCore::InfiniteTerrain;
mTerrain->SetVerticalScale(15.0);
mTerrain->Regenerate();

mEnvironment = new dtCore::Environment();
mEnvironment->SetDateTime(2011, 4, 20, 2, 0, 0);
mEnvironment->AddEffect(new dtCore::SkyDome());
mEnvironment->AddEffect(new dtCore::CloudPlane(6,0.5,6,1,.3,0.96,256,1800));
```

Keďže využívame generovaný nekonečný terén, nastavíme jeho „hornatosť“ pomocou `SetVerticalScale()`, aby sme sa vyhli vygenerovaniu plochej dosky. Následne vytvoríme objekt reprezentujúci Environment. V ňom nastavíme dennú dobu, čo určuje osvetlenie scény a pridáme efekty oblohy a oblakov.

Po vygenerovaní terénu a prostredia môžeme vytvoriť objekty, ktoré chceme do scény umiestňovať.

```
mHelicopter = new dtCore::Object(''Helicopter'');
mHelicopter->LoadFile(''uh-1n.ive'');
mHelicopter->SetTransform(dtCore::Transform(0.f, 0.f, 0.f, 0.f, 0.f, 0.f));

GetScene()->AddDrawable(mTerrain.get());
GetScene()->AddDrawable(mEnvironment.get());
GetScene()->AddDrawable(mHelicopter.get());
```

Vytvoríme teda objekt `dtCore::Object`, ktorý bude reprezentovať v tomto prípade helikoptéru nachádzajúcu sa v scéne. Priradíme tomuto objektu model, ktorý sa bude zob-

razovať, nastavíme jeho polohu a rotáciu pomocou metódy `SetTransform()` a napokon všetky doposiaľ vytvorené objekty umiestnime do scény, čím vlastne enginu povieme, že si ich prajeme vykresľovať.

Následne môžeme pridať k helikoptére zvuk motora. O vytvorenie zvuku požiadame AudioManager, ktorý ho tým zároveň u seba zaregistruje. Nastavíme vlastnosti ako opakovanie a silu či dosah zvuku, spustíme jeho prehrávanie a pridáme ho do scény ako potomka nami vytvorenej helikoptéry.

```
mEngineSound = dtAudio::AudioManager::GetInstance().NewSound();
mEngineSound->LoadFile(''helo.wav'');
mEngineSound->SetLooping(true);
mEngineSound->SetRolloffFactor(0.15f);
mEngineSound->Play();
mHelicopter->AddChild(mEngineSound.get());
```

Následne vytvoríme MotionModel, ktorému ako parametre určíme model klávesnice a myši využívaný enginom Delta3D a špecifikujeme vlastnosti ako rýchlosť chôdze, výšku, ktorú má udržiavať nad terénom a podobne. Následne mu priradíme objekt ktorým chceme pohybovať pomocou `SetTarget()` (v tomto prípade kameru) a scénu, v ktorej sa bude pohybovať pomocou `SetScene()`.

```
mFPSMM = new dtCore::FPSMotionModel(GetKeyboard(), GetMouse(),
                10.0f, 1.5f, 5.0f, 1.8f, 0.7f, true, false);
mFPSMM->SetTarget(GetCamera());
mFPSMM->SetScene(GetScene());
```

Nakoniec ešte zavoláme rodičovskú triedu, aby sme náhodou nevynechali nejaké dôležité inicializácie.

```
Application::Config();
```

Tým vlastne uzatvoríme základnú konfiguráciu scény.

3.3 Kompilácia a spustenie

Takto vytvorená aplikácia je preložiteľná mnohými rúznymi prekladačmi a prenositeľná medzi operačnými systémami. Oficiálne podporovaný je vývoj pre MS Windows a Linux, existujú však aj projekty určené pre Mac OS X. Prenositeľnosť a tvorbu projektových súborov, resp. Makefile prispôsobených konkrétnej platforme a prekladaču zabezpečuje plná kompatibilita s build systémom CMake. Jednoduchá aplikácia, ktorej vytvorenie sme si práve priblížili, má podobu jediného spustiteľného súboru. Nutná je samozrejme prítomnosť využitého modelu v adresárovej štruktúre tak, aby zodpovedala dátovým cestám špecifikovaným v aplikácii.

Výsledkom je potom scéna podobná tejto:

Obrázek 3.2: Jednoduchá ručne vytvorená scéna

Kapitola 4

Rozsiahlejšia demoaplikácia

Delta3D dáva tvorcom simulácií a aplikácií, ako bolo ukázané v predchádzajúcom príklade, voľné ruky a plnú kontrolu nad riadením behu, komunikáciou objektov a tvorbou scény. Z hľadiska efektivity je však ručný prístup, kedy by bolo nutnosťou takpovediac „písať“ umiestnenie každého jednotlivého objektu do scény, nehovoriac o pokročilejšej funkcionalite, pre tvorbu rozsiahlejších aplikácií prakticky nepoužiteľný. Preto je výhodné plne využívať možnosti poskytované enginom, vrátane pomocných nástrojov, ktoré sú súčasťou distribúcie a ponechať si tak možnosť sústreďovať sa na vyššiu úroveň a zaujímavejšie súčasti zamýšľanej hry, simulácie či vizualizácie.

Aj keď systém Delta3D umožňuje pristupovanie k svojim nízkoúrovňovím zložkám, a teda okrem iného aj tvorbu či úpravu jednotlivých zdrojov, napríklad úpravu grafických modelov pomocou prostriedkov poskytovaných OpenSceneGraphom, z hľadiska efektivity vytvárania a prehľadnosti vnútornej štruktúry aplikácie je výhodné pred začiatkom samotnej tvorby venovať priestor zhromažďovaniu a usporadúvaniu resourcov, či už sa jedná o modely, zvukové súbory, shadere, textúry, particle systémy, alebo obrázky a fonty využívané pri tvorbe grafického užívateľského rozhrania. STAGE napríklad umožňuje použitie predpripravených kompletných modelov vo formátoch .ive, .osg, .3ds, . . .

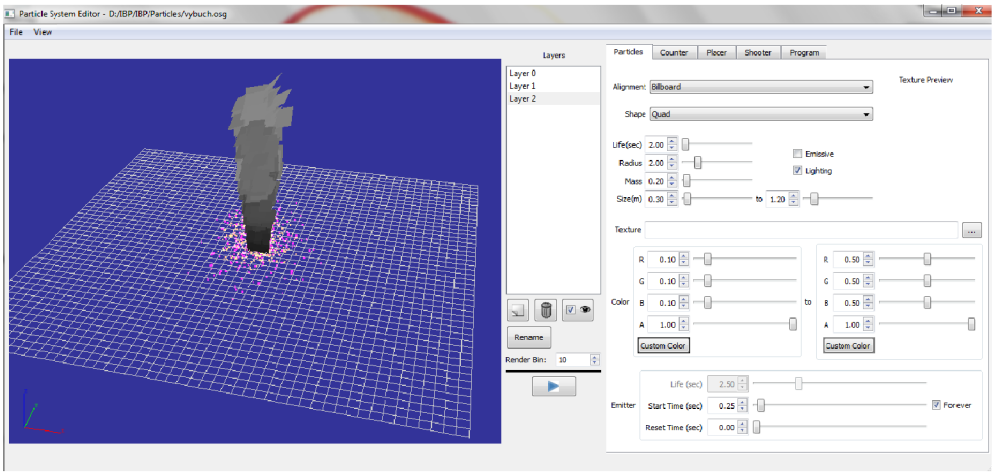
4.1 Pomocné nástroje

Okrem editoru STAGE určeného pre komplexnú tvorbu herných máp a konfiguráciu do nich umiestňovaných Actorov poskytuje distribúcia Delta3D niekoľko ďalších nástrojov pre správu a tvorbu zdrojov využívaných v tvorených aplikáciách.

4.1.1 Particle Editor

Particle Editor umožňuje jednoduché a intuitívne vytváranie a editáciu particle systémov od jednoduchého odpálenia niekoľkých častíc po komplexné efekty ohňa čí dymu, vrátane definovania fyzikálnych vlastností a síl pôsobiacich na častice. Škála možností a nastavení je pomerne široká a užívateľ má možnosť okamžitého náhľadu na výsledok. Bližší popis poskytuje manuálový dokument k Particle Editoru [1].

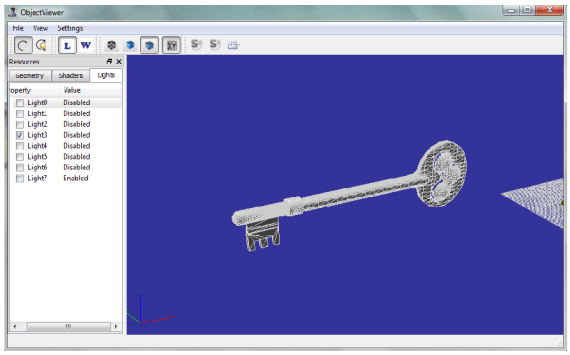
V rámci popisovanej demonštračnej aplikácie bol Particle Editor využitý pre tvorbu jednoduchých efektov prachu za pohybujúcim sa vozidlom (viď 4.5 – 28) a výbuchu pri ukončení hry.



Obrázek 4.1: Particle Editor

4.1.2 Object Viewer

Ďalším pomocným nástrojom, vhodným pre správu zdrojov, je jednoduchý *Object Viewer*, pomocou ktorého je možné prezerať používané objekty s rôznymi možnosťami nasvietenia a aplikácie shaderov.



Obrázek 4.2: Object Viewer

4.2 Správy

Pre potreby komunikácie medzi Actormi a Componentmi v prezentovanej aplikácii sa, okrem tých implicitne prítomných ako súčasť Game Manageru, využíva pomerne rozsiahly repertoár užívateľských typov správ. Tieto typy boli vytvorené pre reprezentáciu aplikačne špecifických významných herných udalostí, alebo zmien stavu aplikácie. Stručné predstavenie ponúka tabuľka B.1.

Samotná registrácia správ u Game Manageru prostredníctvom MessageFactory prebieha bezprostredne po spustení aplikácie. Celý proces je podrobnejšie ukázaný a popísaný v sekcii venujúcej sa logike spustenia aplikácie (viď 4.6 – 38).

STATE_MENU	Informácia, že má byť zobrazené úvodné menu
STATE_GAME	Nastáva beh hry
STATE_DEBRIEF_F	Hráč ukončil hru úspešne
STATE_DEBRIEF_S	Hráč ukončil hru neúspešne
VEHICLE_ENTERED	Hráč nastúpil do vozidla
VEHICLE_LEFT	Hráč opustil vozidlo
VEHICLE_MOVES	Vozidlo sa začalo pohybovať
VEHICLE_STOPPED	Pohyb vozidla sa zastavil
INTERSECTED	Hráč ukazuje na použiteľný predmet
NOT_INTERSECTED	Hráč už neukazuje na použiteľný predmet
ITEM_COLLECTED	Hráč zodvihol predmet
TOGGLE_RADAR	Zapnutie/vypnutie „detektoru“
STATE_PAUSED	Pozastavenie hry
GAME_RETURN	Návrat do hry
RESET	Hráč predčasne opustil hru
SHOW_HINT	Zobrazenie hint správy
HIDE_HINT	Čas ukončiť zobrazovanie hint správy
HOUSE_ENTER_POSSIBLE	Hráč spustil trigger umožňujúci vstup do budovy
HOUSE_ENTER_IMPOSSIBLE	Hráč odišiel z dosahu triggeru

Tabulka 4.1: Užívateľské správy

4.3 Actors

Pre potreby prezentovanej demonštračnej aplikácie bolo vytvorených niekoľko nových typov Actorov, predstavujúcich rôzne typy interaktívnych objektov „fyzicky“ prítomných v scéne, ako aj entitu hráča samotného.

playerActor

Prvým a najdôležitejším Actorom prítomným v predstavovanej aplikácii je *playerActor*, zastupujúci entitu hráča vo virtuálnom svete. Prostredníctvom tohto typu hráč môže pohybovať kamerou a Listenerom, teda sledovať scénu a počúvať zvuky z pozície prvej osoby. Rovnako tento actor umožňuje hráčovi objavovať a používať interaktívne predmety nachádzajúce sa v prostredí hry.

Štruktúra triedy playerActor (s vynechaním nedôležitých pomocných metód) potom vyzerá nasledovne.

```
class IBP_EXPORT PlayerActor : public dtGame::GameActor
{
public:
        PlayerActor(dtGame::GameActorProxy &proxy);
        ~PlayerActor();

        //spracovanie správ
        virtual void OnMessage(dtCore::Base::MessageData *data);
        virtual void OnTickLocal(const dtGame::TickMessage &tickMessage);
        virtual void ProcessMessage(const dtGame::Message &message);
```

```
        void Init();        //inicializácia
        void Intersect();              //picking objektov v okolí
        void Update();                    //riesnie kolizii
private:
        bool collision;
        dtCore::Transform oldPosition, newPosition;
        dtCore::RefPtr<dtCore::Isector> IntersectRay;
};
```

Ako vidieť, playerActor používa pre rôzne správy až tri rozličné metódy. Využitie nízkoúrovňovej `OnMessage()` je nyvhnutnosťou pre spracovanie a reakciu na kolízie hráča s objektmi v scéne. Engine používa pre detekciu kolízií Open Dynamics Engine. Systém sleduje a vyhodnocuje, či objekt kolidoval, ak má tento natavenú vlastnosť `CollisionDetection`, resp. ak je mu priradený nejaký tvar kolíznej geometrie. Ten môže byť rôzny od jednoduchej gule s určitým polomerom obklopujúcej objekt až po komplexný tvar počítaný na základe objektu priradeného modelu.

- `OnMessage()` slúži na spracovanie nízkoúrovňových správ od fyzikálneho enginu a výskyte kolízie v scéne. Systém takto informuje o všetkých kolíziách odohrávajúcich sa v scéne, nevyhnutné preto je pomocou parametrov správy rozlíšiť, či je hráč jedným z kolidujúcich objektov. Ak áno, jediným spracovaním v tejto metóde je nastavenie flagu (alebo teda prepínača) collision.

- `OnTickLocal()`, starajúci sa o reakciu na Tick, v prípade, že sa hráč práve nenachádza vo vozidle, volá metódy `Update()` a `Intersect()`, ktoré zabezpečujú konkrétnu reakciu na výskyt kolízie, resp. kontrolu výskytu interaktívnych predmetov v zornom poli hráča v jeho bezprostrednej blízkosti.

- `ProcessMessage()` zodpovedá za spracovanie užívateľských správ – konkrétne ITEM_COLLECTED, VEHICLE_ENTERED, VEHICLE_LEFT a INFO_TIMER_ELAPSED. Reakciou na tieto správy je nastavenie príznakov o zozbieraní predmetov, reakcia na vstup do vozidla, resp. výstup z vozidla a pomocná funkcionalita pri zobrazovaní nápovedy ako reakcie na určité úkony vykonané hráčom.

Pre potreby reakcie na kolíziu hráča s okolím si playerActor uchováva informáciu o svoje aktuálnej a poslednej dobrej (pred výskytom kolízie) polohe. Metóda `Update()` potom buďto premiestni hráča na pozíciu pred kolíziou (čo je vzhľadom na frekvenciu vykonávania kontroly nebadateľný posun), alebo aktualizuje údaje o aktuálnej pozícii, čím sa zabezpečí, že hráč nemôže prechádzať cez predmety vo svojom okolí. Zjednodušene:

```
void PlayerActor::Update()
{
        if(collision){
                SetTransform(oldPosition);
                SetCollision(false);
        }else{
                oldPosition = newPosition;
                GetTransform(newPosition);
        }
}
```

Zmena polohy hráča je zabezpečená využitím `dtCore::FPSMotionModel`, ktorý ovládaním a správaním zodpovedá pohybu hráča typickému pre First Person Shooter hry.

Rozoznávanie aktívnych predmetov v okolí hráča a umožnenie základnej interakcie s nimi vykonáva intersekčný vektor `dtCore::Isector`. Tomu pri inicializácii hráča priradíme zvolenú vzdialenosť od hráča, ktorá nás ešte zaujíma a informujeme ho o scéne, v ktorej si intersekcie prajeme zisťovať.

```
IntersectRay->SetScene(&GetGameActorProxy().GetGameManager()->GetScene());
IntersectRay->SetLength(3.f);
```

Potom pri každom volaní metódy `Intersect()` zresetujeme a aktualizujeme polohu a smerovanie Isectoru na základe údajov o transformačnej matici Actora.

```
IntersectRay->Reset();

osg::Vec3 trans, rot;
osg::Matrix rotMatrix;
newPosition.GetTranslation(trans);
newPosition.GetRotation(rotMatrix);
rot.set(rotMatrix(1,0), rotMatrix(1,1), rotMatrix(1,2));

IntersectRay->SetStartPosition(trans);
IntersectRay->SetDirection(rot);
```

Následne voláme metódu `Update()` Isectoru a ak uspeje (došlo k „prieseku“), pozrieme sa na najbližší objekt. V opačnom prípade vyšleme správu NOT_INTERSECTED. Opäť zjednodušene:

```
if(IntersectRay->Update())
{
      dtCore::DeltaDrawable *item = IntersectRay->GetClosestDeltaDrawable();
      if(item != NULL)
      {
            InteractiveActor *ia = dynamic_cast<InteractiveActor*>(item);
            if(ia){
                  //vyslanie spravy INTERSECTED
            }
      }else{
            //vyslanie spravy NOT_INTERSECTED
      }
}
```

Ak sa jedná o objekt odvodený od triedy InteractiveActor, vyšleme správu INTERSECTED obsahujúcu unikátne id daného objektu.

Dôležitým spôsobom interakcie hráča s okolím je tiež možnosť využiť „detektor“ pokladov (4.3 – 29) v exteriérovej, resp. svetlo „baterky“ v interiérovej mape. Tieto prvky ale niesú integrálnou súčasťou Actora reprezentujúceho hráča.

Keďže v aplikácii využívame prechod medzi viacerými mapami, hráčov inventár je nutné simulovať v niektorom z Componentov, aby nedošlo k strate jeho obsahu pri zmene mapy. Podrobnejšie je problém popísaný v časti zaoberajúcej sa *InputComponentom* (4.5.2 – 36).

InteractiveActor

InteractiveActor, ako je očividné už z názvu, označuje Actora, s ktorým je možné z pohľadu hráča interagovať. Jedná sa o bázovú triedu pre špecifickejšie typy Actorov – collectibleActor, treasureActor a vehicleActor. Tento typ sám sa v praxi v žiadnej vytvorenej mape nevyskytuje, jeho hlavným účelom je rozlýšenie aktívneho predmetu v hráčovom okolí od statických objektov a následné umožnenie rôznej interakcie v závislosti na konkrétnom deriváte.

Actor je odvodený z triedy `dtGame::GameMeshActor`, čo je Game Actor, ktorému je možné priradiť 3D model zabezpečujúci jeho „fyzickú“ prítomnosť v realite hry.

collectibleActor

Prvým typom Actora odvodeným od InteractiveActor je zodvihnuteľný predmet predstavujúci *collectibleActor*. Tohto typu sú predmety v scéne, ku ktorým hráč môže pristúpiť a po ich aktivácii dôjde k ich pridaniu do hráčovho inventára a odobratiu zo scény. Jedná sa zväčša o objekty, ktoré hráč „použije“ neskôr v priebehu hry. Takto je to v predstavovanej aplikácii, je to však záležitosť logiky a pravidiel hry, nejedná sa teda o nevyhnutnosť.

Actor reaguje na správu ITEM_COLLECTED, kde podľa jej súčasti `AboutActorId` určí, či je učená jemu a ak áno, dôjde k odstráneniu zo scény. To je zabezpečné volaním metódy `SetActive()` s parametrom `false`, čím sa oznámi enginu, že tento objekt si ďalej neprajeme zobrazovať. Je tiež odstránená kolízna geometria objektu volaním `ClearCollisionGeometry()`.

V demonštračnej aplikácii je collectibleActor využitý v podobe kľúča od truhlice a lopaty.

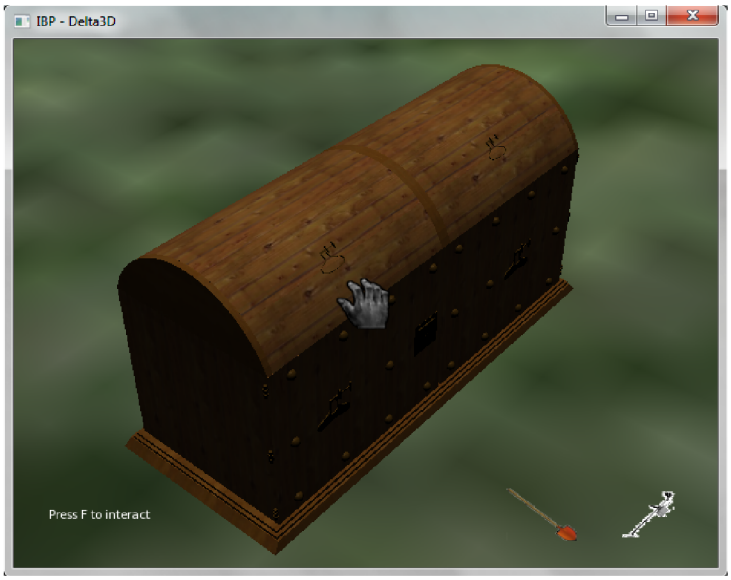
Obrázek 4.3: collectibleActor: Zodvihnutie objektu lopaty. Umiestnenie kľúča v scéne.

treasureActor

Ďalší typ, *treasureActor*, predstavuje zakopaný poklad, ktorého nájdenie je hlavnou úlohou hráča. Typ je opäť odvodený od InteractiveActor.

Actor sa neregistruje pre žiadne správy, všetka interakcia s ním je priamo v réžii InputComponentu a je závislá na stave hráčovho inventáru, teda na tom, ktoré predmety už hráč vo svete hry zozbieral či použil.

Pre navedenie hráča k pozícii pokladu slúži radarActor (viď 4.3 – 29). Ak hráč pozíciu pokladu objaví a pokúsi sa ho použiť skôr, ako si do inventáru zaradí lopatu, aplikácia zareaguje výpisom jednoduchej správy s nápovedou o nutnosti tohto úkonu. Po získaní lopaty prebieha interakcia v dvoch fázach. Prvou je vykopanie pokladu, kedy sa tento „zviditeľní“ (vysunie sa nad povrch terénu). Druhou je otvorenie truhlice. To sa môže podariť, ak má hráč pri sebe kľúč, čo vyústi v úspešné ukončenie hry. Ak sa však pokúsi



Obrázek 4.4: treasureActor

truhlicu otvoriť aj bez kľúča, dôjde k výbuchu nástražného systému, o možnosti čoho je hráč po vykopaní pokladu vopred upovedomený ďalšou krátkou nápovedou.

Výbuch samotný je znázornený odpálením jednoduchého particle systému a prehratím zvuku detonácie, načo je hráč znehybnený vypnutím MotionModelu, ktorý zabezpečuje jeho pohyb a dochádza k zobrazeniu ukončovacej obrazovky informujúcej o prehre a umožňujúcej návrat do hlavného menu.

Particle systém a zvuk výbuchu sú vytvárané dynamicky.

```
dtCore::RefPtr<dtAudio::Sound> explosionSound;
dtCore::RefPtr<dtCore::ParticleSystem> explosionSystem;
```

Pre ich inicializáciu slúži metóda `Init()` volaná v rámci inializácií po načítaní mapy obsahujúcej tento typ Actora.

```
void TreasureActor::Init()
{
        explosionSound = dtAudio::AudioManager::GetInstance().NewSound();
        explosionSound->LoadFile(''Sounds/exp57.wav'');
        explosionSound->SetLooping(false);
        AddChild(explosionSound);

        explosionSystem = new dtCore::ParticleSystem(''explosion'');
        explosionSystem->LoadFile(''Particles/vybuch.osg'');
        explosionSystem->SetEnabled(false);
        explosionSystem->ResetTime();
        AddChild(explosionSystem);
}
```

Následne interakciu zabezpečuje volanie metód `Use()`, resp. `Explode()` v závislosti na aktuálnom stave hráčovho inventáru, ako aj samotného pokladu z InputComponentu ako reakcia na stlačenie príslušného tlačidla užívateľom.

vehicleActor

Ďalším (a posledným) typom odvodeným od InteractiveActora je *vehicleActor*. Jedná sa o ovládateľné vozidlo slúžiace k urýchleniu presunov hráča po exteriérovej mape.

Obrázek 4.5: vehicleActor v pohybe.

Vozidlo, rovnako ako hráč pohybujúci sa „po vlastných“, musí byť schopné reagovať na kolízie s objektmi nachádzajúcimi sa v hernom svete. Mechanizmy zabezpečujúce kolidovanie sú prakticky totožné s tými použitými v type playerActor. Rovnako aj sledovanie terénu a udržiavanie sa v konštantnej výške nad ním je opäť riešené MotionModelom zabezpečujúcim pohyb ako taký. Podobne particle systém prachu a zvuk motora sú inicializované rovnakým spôsobom, ako podobné prvky u iných typov Actorov.

Keďže sa v zásade jedná len o interaktívny ovládateľný pohyblivý objekt, nemusí sa po aplikovaní vhodného MotionModelu jednať len o vozidlo pozemné, ale pokojne aj o vodné či vzdušné. Preto je súčasťou nastaviteľných properties aj prepínač `isLand`, ktorý zabezpečuje pridanie particle systému simulujúceho prach za pohybujúcim sa pozemným vozidlom. V prípade prezentovanej aplikácie sa jedná o pohyb po teréne, bol teda zvolený `dtCore::WalkMotionModel` s istými obmedzeniami (napríklad zákaz úkrokov do strán a pod.). Pridanie vlastnosti *isLand* tak, aby bola nastaviteľná v prostredí editoru STAGE potom prebieha v metóde `BuildPropertyMap()` prislúchajúcej k Proxy tohto Actora.

```
void VehicleActorProxy::BuildPropertyMap()
{
        const std::string GROUP = ''VehicleActor'';
        VehicleActor &va = static_cast<VehicleActor&>(GetGameActor());
        InteractiveActorProxy::BuildPropertyMap();

        AddProperty(new dtDAL::BooleanActorProperty(''isLand'',''isLand'',
                dtDAL::MakeFunctor(va, &VehicleActor::SetIsLand),
                dtDAL::MakeFunctorRet(va, &VehicleActor::GetIsLand),
                ''Sets/gets whether actor is a land vehicle'', GROUP));
}
```

Najpodstatnejším je samozrejme `AddProperty()`, kde je okrem názvu či skupiny vlastností, do ktorej práve pridávaná spadá, aj priamo určené, ktoré metódy samotného Actora (ktorého rozhranie Proxy zabezpečuje) budú použité pre nastvenie, resp. zistenie hodnoty konkrétnej vlastnosti v editore.

Actor reaguje na správy VEHICLE_ENTERED, VEHICLE_LEFT, VEHICLE_MOVES a VEHICLE_STOPPED. Samozrejmou je schopnosť reakcie na Tick, kde okrem riešenia kolízií dochádza aj k spúšťaniu particle systému prachu.

Reakciou na správy VEHICLE_ENTERED a VEHICLE_LEFT je nastavenie prepínača `isActive`, spúšťajúceho detekciu kolízií a tiež spustenie, resp. ukončenie prehrávania zvuku motora. V prípade, že sa jedná o pozemné vozidlo, reakciou na VEHICLE_MOVES a VEHICLE_STOPPED je prepnutie prepínača `isMoving`, riadiaceho odpálenie particle systému v rámci metódy `OnTickLocal()`.

radarActor

radarActor predstavuje akýsi „detektor“, ktorý umožňuje jednoduchšiu lokalizáciu pokladu v hernej mape. Keďže je potrebný len v jednej z herných máp, konkrétne v tej, v ktorej sa poklad skutočne nachádza, nieje integrálnou sučasťou hráčskej entity, pri incializácii je však medzi nimi vytvorený príbuzenský vzťah, ktorým zabezpečíme dojem, že to tak je (spoločný pohyb po mape). Po svojom zapnutí užívateľom mení rýchlosť a intenzitu zvukovej signalizácie na základe vzdialenosti pokladu od svojej pozície v priestore, čím hráčovi napovedá, či sa k pozícii pokladu približuje, alebo sa naopak vzdaľuje (systémom teplo/zima).

Actor odoberá správy TOGGLE_RADAR a tiež reaguje na Tick. Reakciou na správu TOGGLE_RADAR je prepnutie prepínača `enabled`, na základe hodnoty ktorého je následne umožnené spustenie reakcie na Tick správy. V rámci metódy `OnTickLocal()` potom prebehne vlastný prepočet vzdialenosti a inkrementácia čítača o príslušnú hodnotu. Po naplnení čítača dôjde k prehratiu zvuku pípnutia, čím je dosiahnutý želaný efekt a čítač vynulujeme.

Inicializácia zvuku, ako aj registrácia pre odber potrebných správ, prebieha rovnako ako v už uvedených príkladoch iných typov Actorov.

enterTriggerActor

Súčasťou tohto Actora je `dtABC::ProximityTrigger` (teda spínač reagujúci na blízkosť) a taktiež objekt `EnterAction`, ktorý je odvodený od `dtABC::Action`, čo je vlastne udalosť vyvolateľná napríklad práve zopnutím spínača. V tomto konkrétnom prípade je akciou, ktorú vyvolá kolízia hráča a Triggeru vyslanie správy HOUSE_ENTER_POSSIBLE, čo predstavuje fakt, že hráč sa priblížil ku vstupu do budovy, resp. výstupu z nej (je takpovediac „na dosah dverí“).

Tento Actor teda spracúva správy HOUSE_ENTER_POSSIBLE a Tick. Prvá slúži na zopnutie prepínača umožňujúceho vstup do budovy, čím sa zároveň začne s periodickým prepočtom vzdialenosti hráča od vstupu do budovy v rámci metódy `OnTickLocal()` a po prekročení istej hranice k opätovnému vypnutiu prepínača a tým znemožneniu vstupu. Pre potreby výpočtu si pri svojej inicializácii Actor uchová informáciu o svojej polohe a unikátne id objektu, kolízia s ktorým má spôsobiť spustenie pripojenej akcie.

IBPgameActorsRegistry

Aby editor STAGE, resp. Game Manager našiel, rozoznal a bol schopný pracovať s vytvorenými typmi Actorov, je nutné v knižnici vytvoriť akýsi prístupový a registračný bod. V architektúre aplikácie v prostredí Delta3D slúži tomuto účelu trieda `dtDAL::ActorPluginRegistry`, resp. v tomto prípade jej derivát `IBPgameActorsRegistry`.

```
class IBP_EXPORT IBPgameActorsRegistry : public dtDAL::ActorPluginRegistry
{
public:
        IBPgameActorsRegistry();
        virtual void RegisterActorTypes();
private:
        dtCore::RefPtr<dtDAL::ActorType> PlayerActorType;
};
```

Okrem metódy `RegisterActorTypes()`, v ktorej prebieha vlastná registrácia, je nutné pričleniť ešte dve dôležité funkcie. Pri importe knižnice v nej totiž STAGE vyhľadá funkcie `CreatePluginRegistry()` a `DestroyPluginRegistry()`. Deje sa tak z dôvodu zaistenia validity knižnice. Je nutné deklarovať ich ako `extern ''C''` aby ich bol editor schopný správne spracovať. Proces registrácie Actora bude ukázaný na príklade jedného typu, konkrétne playerActor.

```
extern ''C'' IBP_EXPORT dtDAL::ActorPluginRegistry* CreatePluginRegistry()
{
        return new IBPgameActorsRegistry();
}

extern ''C'' IBP_EXPORT void DestroyPluginRegistry(dtDAL::ActorPluginRegistry *reg)
{
        delete reg;
}

void IBPgameActorsRegistry::RegisterActorTypes()
{
      PlayerActorType = new dtDAL::ActorType(''PlayerActor'',
              ''IBPactors'', ''Actor representing player entity'');

      mActorFactory->RegisterType<PlayerActorProxy>(PlayerActorType.get());
}
```

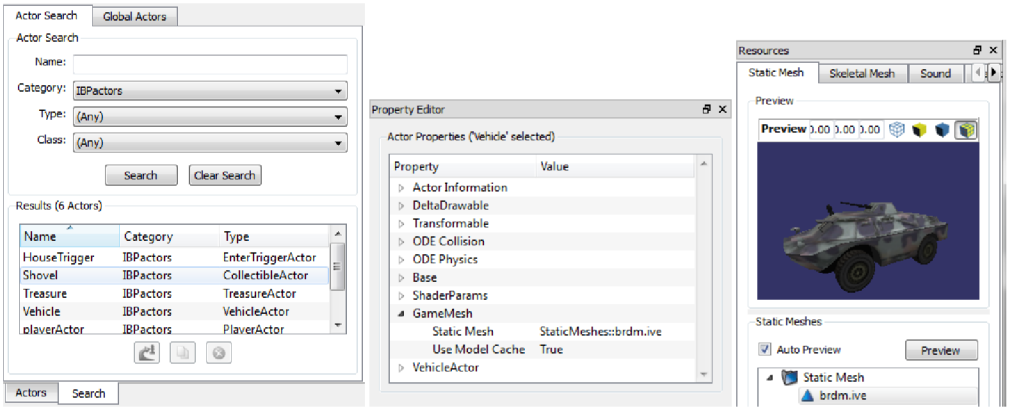
Týmto jednoduchým spôsobom môžeme v rámci jednej knižnice vytvoriť a u ActorFactory registrovať ľubovoľné množstvo Actorov.

4.4 Mapy

4.4.1 Tvorba

Vytvorenie hernej mapy, reprezentujúcej scénu v samotnej aplikácii, je s využitím editoru STAGE veľmi užívateľsky príjemné. Po zvolení kontextu, pripojení knižníc a vytvorení mapy ako takej (viď 2.2.4) môžeme okamžite prejsť k jej plneniu a zostavovaniu scény. Ak chceme do mapy vložiť napríklad obyčajný statický model, urobíme to pomocou niekoľko

málo jednoduchých krokov. Najskôr z knižnice actorov vyberieme vhodný typ, ktorý bude tento objekt reprezentovať (pre naše potreby napr. *Static Mesh*) a zvolíme vytvoriť (Create Actor). Tým dôjde k umiestneniu do mapy a actor sa objaví ako v náhľade na scénu, tak v zozname v nej obsiahnutých actorov. Ak by sa jednalo o typ, kde konkrétne umiestnenie nedáva logicky zmysel (napr. *SkyBox*), objaví sa v zozname *Global Actors*. Teraz je možné práve vložený objekt vybrať (označením priamo v náhľade na scénu, resp. zvolením zo zoznamu) a konfigurovať ho pomocou Property editoru. V tomto prípade by to zahŕňalo napr. priradenie modelu spomedzi zdrojov prístupných v projekte a nastavenie jeho transformácie (polohy, rotácie, scale). Obdobným spôsobom je teda možné vyskladať pomerne zložité mapy a jednoducho a rýchlo tak pripraviť základ pre grafické scény v tvorenej aplikácii.



Obrázek 4.6: Stage – súčasti: Actors, Property editor, Resource editor.

4.4.2 Prepojenie s vlastnou aplikáciou

Vytvorenú mapu je možné v aplikácii využiť ručným vložením:

```
dtDAL::Project::GetInstance().SetContext(''..'');

dtDAL::Map &Map = dtDAL::Project::GetInstance().GetMap(''Map'');
dtDAL::Project::GetInstance().LoadMapIntoScene(Map, *GetScene());
```

prípadne po správnom nastavení kontextu ponechať tieto kroky na Game Manager a správu herných máp využiť jeho metódy `ChangeMap()`, `GetCurrentMap()`, či `CloseCurrentMap()`. Game Manager potom umožňuje pristupovať k Actorom umiestneným v aktuálne používanej mape na základe rôznych kritérií – či už o výber konkrétneho Actora prostredníctvom jeho unikátneho id (reprezentovaného pomocou `dtCore::UniqueId`) alebo mena, prípadne k skupinám Actorov v závislosti na ich type, bázovej triede alebo ku všetkým Actors, resp. Game Actors, ktorí sú v mape obsiahnutí. Príklad vytvorenia zoznamu všetkých Game Actorov (resp. ich Proxy, pomocou ktorých Game Manager Actorov rozoznáva) v pripojenom Componente:

```
std::vector<dtGame::GameActorProxy*> proxies;
GetGameManager()->GetAllGameActors(proxies);
```

Následne je možné štandardne spracúvať vytvorený zoznam, Actorov ďalej triediť a pristupovať k nim volaním `GetActor()` na Proxy tých, s ktorými si prajeme pracovať.

4.4.3 Použité mapy

V aplikácii má hráč možnosť (resp. pre úspešné splnenie herných úloh a ukončenie hry víťazstvom nutnosť) vstúpiť do jednej z budov. Interiér budovy je predstavovaný modelom skladiska, ktorý je súčasťou distribúcie Delta3D. Presun medzi budovou a exteriérom by bolo možné riešiť aj v rámci jedinej hernej mapy – napríklad umiestnením modelu skladiska pod terén alebo inak vhodne tak, aby nebol hráčom viditeľný pri prechádzaní exteriérom a následnou prostou „teleportáciou“ hráča na zodpovedajúcu pozíciu v prípade, že vykoná vstup. Aby však bolo možné demonštrovať prácu Game Manageru pri zmene mapy, ako aj možnosti a komplikácie z toho vyplývajúce, interiér je v aplikácii predstavovaný samostatnou mapou.

Exteriér

Prvou mapou, v ktorej sa hráč objaví po spustení novej hry, je exteriérová mapa. Je tvorená niekoľkými modelmi domov a stromov v jednoduchom teréne. Predstavuje malú opustenú osadu, v okolí ktorej by malo byť možné nájsť poklad.

Obrázek 4.7: Exteriérová mapa – STAGE

Keďže mapa nieje tvorená nekonečným terénom, ale len pomerne obmedzenou plochou na ktorej sa nachádzajú z pohľadu hry zaujímavé a významné miesta a predmety, bolo nutné zabezpečiť, aby sa hráč nemohol – či už úmyselne, alebo nedopatrením – dosať mimo vyhradenú oblasť a takpovediac sa „prepadnúť do ničoty“. Terén, po ktorom sa hráč pohybuje, je preto ohradený neviditeľnými stenami, teda objektmi, ktoré síce netvorí žiaden zobraziteľný model, ale disponujú kolíznou geometriou a blokujú tak pohyb hráča mimo oblasť terénu.

Obloha a osvetlenie je tvorené globálnym Environment Actorom, namiesto SkyDome a CloudPlane, ktoré boli ukázané pri ručnej tvorbe aplikácie, je v tomto prípade využitá alternatíva SkyBox, kde je vlastne celá scéna umiestnená do vnútra boxu (alebo kocky), ktorého steny sú navzájom nadväzujúce textúry krajiny a oblohy a tvoria tak akýsi panoramatický efekt.

V tejto mape je umiestnených niekoľko interaktívnych predmetov – menovite lopata, ktorú hráč môže zobrať a použiť na vykopanie pokladu, poklad sám o sebe, ovládateľné vozidlo pre zrýchlenie presunu hráča po mape, trigger umožňujúci vstup do jednej z budov (formou zmeny mapy, ako už bolo spomenuté) a radar (alebo detektor) pre vyhľadanie pozície pokladu.

Interiér budovy

Druhá mapa, reprezentujúca interiér jednej z budov nachádzajúcich sa v prvej mape, je tvorená jediným modelom skladiska, obsahujúcim niekoľko miestností a chodieb, v ktorých sa hráč môže voľne pohybovať.

Obrázek 4.8: Exteriérová mapa – ingame

Mapa obsahuje interaktívny predmet kľúč, ktorý je z hľadiska hry významný a úlohou hráča v tejto mape je práve jeho nájdenie. Pre ozvláštnenie tejto úlohy je vnútro skladiska ponorené do tmy a hráč si pre uľahčenie navigácie v jeho priestoroch a objavenie hľadaného predmetu môže svietiť jednoduchou „baterkou“. Baterku je simulovaná pomocou neupraveného `dtCore::Spotlight`.

Keďže model skladiska je uzavretý, žiadne špeciálne bariéry, ktoré by hráča udržali v priestore určenom pre hru, niesú potrebné.

4.5 Components

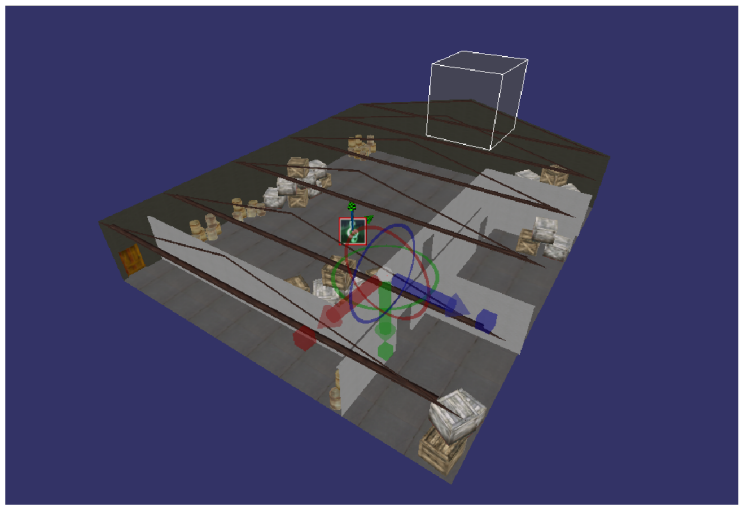
Súčasťou demonštračnej aplikácie sú, ako už bolo spomenuté, dva vlastné Componenty. Z hľadiska logiky aplikácie bolo nutné nad rámec základnej funkcionality poskytovanej GM zabezpečiť kresbu GUI v podobe úvodného menu, obrazoviek s nápovedou či hráčskeho inventára atď., k čomu bol vytvorený *GUIComponent*. Pre spracovanie užívateľského vstupu a umožnenie komfortného ovládania aplikácie, ako aj udržiavanie informácií o priebehu a vynucovanie pravidiel hry si vyžiadalo vznik *InputComponentu*.

4.5.1 GUIComponent

Pre kresbu grafického užívateľského rozhrania je v systéme Delta3D najvyužívanejší *Crazy Eddie's GUI System* (skrátene CEGUI). Súčasťou distribúcie je niekoľko základných fontov a grafických schém využiteľných pri príprave aplikácie. Konkrétne sa jedná o font DejaVuSans a schému WindowsLook, pracujúcu s imagestemi a looknfeel súbormi štýlu WindowsLook. Pre potreby prezentovanej aplikácie bolo vytvorených niekoľko ikon vo formáte Targa a imagesetov, čo sú popisné súbory vo formáte XML umožňujúce pričlenenie vytvorených obrázkov k schéme WindowsLook.

Jednotlivé obrazovky a okná (`CEGUI::Window`) alebo tlačidlá (`CEGUI::PushButton`), ktorými sú tvorené, spoločne vytvárajú stromovú štruktúru, v ktorej každá významná obrazovka (úvodné menu, obrazovka nápovedy, hráčsky HUD atp.) tvorí jednu vetvu. Prepínanie obrazoviek teda prebieha jednoducho ako ukrývanie či zobrazovanie koreňových uzlov jednotlivých obrazoviek. Pri vytvorení samotného Componentu okamžite dochádza k výstavbe všetkých obrazoviek. štruktúra triedy GUIComponent potom zjednodušene:

```
class IBP_EXPORT GUIcomponent : public dtGame::GMComponent
{
public:
```



Obrázek 4.9: Interiérová mapa – STAGE

```
        GUIcomponent(dtCore::DeltaWin &win,
                dtCore::Keyboard &keyboard,
                dtCore::Mouse &mouse,
                const std::string &name);
        virtual void OnAddedToGM();
        virtual void ProcessMessage(const dtGame::Message &message);

private:
        void BuildMainMenu();
        //reakcia na tlacidlo NEW
        bool OnNew(const CEGUI::EventArgs &e);

        //okna a tlacidla tvoriace strukturu rozhrania
        CEGUI::Window *mMainWindow;
        CEGUI::Window *mMainMenu;
        CEGUI::PushButton *mNew;
        //DeltaDrawable pre rendrovanie GUI
        dtCore::RefPtr<dtGUI::CEUIDrawable> mGUI;
};
```

Prvým krokom je načítanie súboru s popisom grafickej schémy a vytvorenie koreňového uzla celého GUI.

```
mGUI = new dtGUI::CEUIDrawable(&win, &keyboard, &mouse);

std::string schemefile(osgDB::findDataFile(''WindowsLook.scheme''));
CEGUI::SchemeManager* sm = CEGUI::SchemeManager::getSingletonPtr();
sm->loadScheme(schemefile);

CEGUI::WindowManager* wm = CEGUI::WindowManager::getSingletonPtr();
mMainWindow = wm->createWindow(''DefaultGUISheet'', ''root'');
CEGUI::System::getSingleton().setGUISheet(mMainWindow);
```

Následne je možné vytvoriť celú štruktúru tvoriacu obrazovky grafického rozhrania. Vytvorenie tlačidla spúšťajúceho novú hru a pripojenie náležitej funkcionality potom vyzerá nasledovne.

Obrázek 4.10: Interiérová mapa – ingame

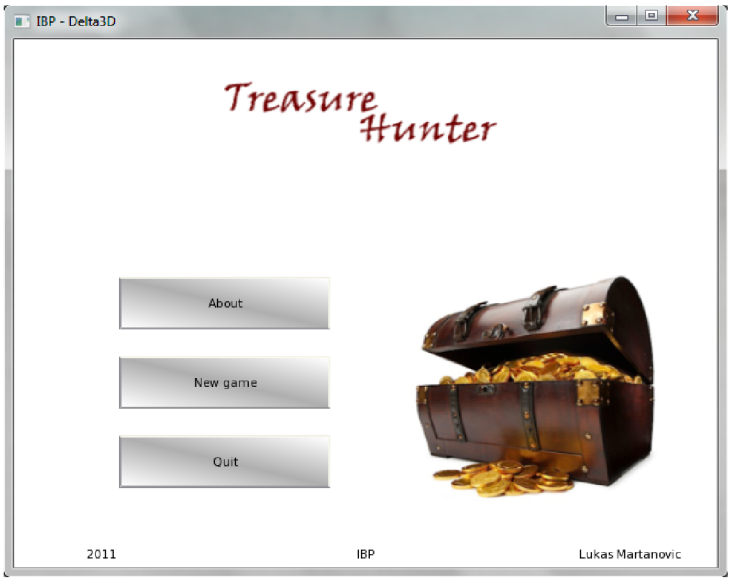
```
mNew = static_cast<CEGUI::PushButton*>
                (wm->createWindow(''WindowsLook/Button'', ''newButton''));
mNew->setText(''New game'');
mNew->setSize(CEGUI::UVector2(cegui_reldim(0.3f),cegui_reldim(0.1f)));
mNew->setPosition(CEGUI::UVector2(cegui_reldim(0.15f),cegui_reldim(0.6f)));
mMainMenu->addChildWindow(mNew);

mNew->subscribeEvent(CEGUI::PushButton::EventClicked,
                CEGUI::Event::Subscriber(&GUIcomponent::OnNew, this));
```

Ako je možné vidieť, po určení typu dochádza k jednoduchému textovému nastaveniu vlastností, ako je veľkosť relatívna k veľkosti okna, zarovnanie či umiestnenie v rámci okna, alebo popiska a nakoniec k pripojeniu do stromovej štruktúry. Rovnakým alebo podobným spôsobom sú potom vytvorené všetky prvky grafického rozhrania. Tlačidlám ešte volaním metódy `subscribeEvent()` priradíme obsluhu. V tomto prípade reakciou na stlačenie tlačidla *New game* bude volanie metódy `OnNew()`, ktorá zabezpečí príslušnú obsluhu.

Pokyn pre vykresľovanie vytvoreného rozhrania sa enginu zadáva v rámci metódy `OnAddedToGM()`, ktorá je volaná okamžite po registrácii Componentu u Game Manageru (viď 4.6 – 38).

```
void GUIcomponent::OnAddedToGM()
{
        GetGameManager()->GetApplication().AddDrawable(mGUI.get());
}
```

GUIComponent, ako každý typ Componentu, dostáva od Game Manageru všetky správy, ktoré sa počas behu aplikácie vyskytnú. Najdôležitejšími z pohľadu zobrazovania GUI sú samozrejme tie, ktoré informujú o zmene stavu aplikácie (ako je ukončenie hry, či jej pozastavenie), ďalej informácie o zozbieraní predmetov v scéne, ktorých ikony sa následne zobrazujú v hráčovom HUD a správy o výskyte interaktívneho predmetu pred hráčom, čo spôsobuje zobrazenie ikony ruky v strede a stručnej nápovedy v ľavom dolnom rohu obrazovky. Po učinení dôležitých krokov počas postupu hrou, ako sú zodvihnutie jednotlivých predmetov, resp. interakcia s pokladom, sa hráčovi taktiež zobrazuje stručná nápoveda, čo by mal ešte vykonať, resp. varovanie pred možnosťou, že poklad je chánený pascou. Zobrazovanie HUDu ilustruje obrázok 4.4 na strane 27.



Obrázek 4.11: GUI – Úvodné menu.

4.5.2 InputComponent a riadenie aplikácie

InputComponent, ako už bolo spomenuté, okrem spracúvania užívateľského vstupu zabezpečuje aj inicializáciu Actorov po načítaní alebo zmene mapy a taktiež samotnú hernú logiku. Keďže je v dôsledku toho štruktúra triedy pomerne rozsiahla, obmedzíme sa pri popise na najdôležitejšie súčasti a funkcionalitu.

```
class IBP_EXPORT InputComponent : public dtGame::BaseInputComponent
{
public:
        virtual void ProcessMessage(const dtGame::Message& message);
        virtual bool HandleKeyPressed(const dtCore::Keyboard* keyboard, int key);
        virtual bool HandleKeyReleased(const dtCore::Keyboard* keyboard, int key);

        dtCore::UniqueId GetLastInteraction();
private:
        void ChangeMap();
        void SetupMotionModel();
        void SetupPlayer();
        void SendInteractionMessage(const dtGame::MessageType & type);

        bool inv_key, inv_shovel;
        dtCore::RefPtr<dtCore::FPSMotionModel> fmm;
        dtCore::RefPtr<dtCore::WalkMotionModel> wmm;
        InteractiveActor *intersectedObject;
        dtCore::UniqueId lastInteraction;
};
```

Ako vidieť, v aplikácii sú využite dva rôzne MotionModely – `dtCore::FPSMotionModel` pre „peší“ pohyb hráča typický pre FPS hry a `dtCore::WalkMotionModel` pre pohyb vozidla. Keďže aplikácia vyžaduje prechod medzi viacerými hernými mapami, v dôsledku čoho nieje možné uchovávať inventár priamo ako súčasť playerActora (viď nižšie) a zároveň je inventár v tomto prípade pomerne kompaktný, pre jeho simuláciu bolo zvolených niekoľko premenných typu `bool` uchovávaných v rámci InputComponentu. Pre potreby interakcie so svetom, napríklad vkladanie identifikačných údajov aktivovaného objektu do správ, slúži premenná lastInteraction.

Zmena mapy

Zmenu mapy za behu hry vykonáva Game Manager v niekoľkých krokoch, ktorých vykonanie zaberie niekoľko frameov:

- Na začiatku rozlýši, či už nejaká mapa načítaná je a v závislosti na tom rozošle správu INFO_MAP_CHANGE_BEGIN (ak nieje), alebo INFO_MAP_UNLOAD_BEGIN.
- Akonáhle ukončí prípadné uzatváranie mapy, rozošle správu INFO_MAP_UNLOADED, nasledovanú správou INFO_MAP_LOAD_BEGIN pred zahájením samotného načítania.
- Ukončenie a pripravenosť mapy oznámi potom rozoslaním správy INFO_MAP_LOADED a úplne na záver INFO_MAP_CHANGED.

INFO_MAP_CHANGED a INFO_MAP_LOADED sú z hľadiska logiky behu aplikácie rovnocenné a po obdržaní ktorejkoľvek z nich je možné zahájiť prácu s načítanou mapou.

Pri prechode do novej mapy Game Manager zruší všetkých Actorov a asociované dáta z predošlej, práve opúšťanej mapy. Keďže hra, ktorej tvorbu popisujeme, umožňuje viacnásobný prechod medzi mapami, je nutné zabezpečiť, aby hráč napríklad po návrate do exteriéru našiel herný svet v takej podobe, v akej ho pred vstupom do budovy zanechal. Nebolo by žiadúce, aby sa opäť objavili predmety, ktoré z mapy už boli odobrané, vykopaný poklad sa opäť zakopal alebo vozidlo, na ktorom sa priviezol teleportovalo na východziu pozíciu. Preto je nevyhnutné implementovať logiku a mechanizmy, ktoré sa postarajú o konzistenciu.

Inicializácia Actorov nachádzajúcich sa v hernej mape a prípadné prispôsobenie hráčom už vykonaným aktivitám po zmene mapy je tiež vykonávaná v tomto Componente. Pre tento účel slúži metóda `ChangeMap()`, ktorá v závislosti na práve načítanej mape a prepínačoch informujúcich o obsahu hráčovho inventáru či zmene polohy vozidla volá príslušné inicializácie jednotlivých interaktívnych Actorov nachádzajúcich sa v scéne. Takto je zabezpečené správne rozmiestnenie objektov, „naštartovanie“ detekcie kolízií inicializáciou dátových zložiek playerActora či vehicleActorov, ako aj previazanie prípadne prítomného radarActora alebo „baterky“ vo forme svetelného zdroja s entitou hráča. Dôležitým je tiež prepojenie kamery a tým aj objektu Listenera s entitou hráča.

Špecifickým krokom pri inicializácii interiérovej mapy je potom zrušenie globálneho osvetlenia a jeho nahradenie jediným svetelným zdrojom – hráčovou baterkou.

Interakcia hráča so svetom

Užívateľ má možnosť okrem pohybu v scéne vykonávať viacero činností, ako je presun medzi mapami, aktivácia vozidla alebo pokladu, zbieranie predmetov alebo zapínanie a vypínanie „baterky“, resp. „radaru“. K tomuto samozrejme využíva štandardne klávesnicu, vstupy z ktorej spracúvajú metódy `HandleKeyPressed()` a `HandleKeyReleased()`. Reakciou na väčšinu kláves je špecifická akcia, napríklad `R` v exteriérovej mape riadi len a len zopínanie radaru, preto je možné ako obsluhu vykonať jednoducho zostavenie a odoslanie príslušnej správy, na ktorú potom zareagujú dotknuté objekty.

Zložitejšou je však aktivácia interaktívnych predmetov, ktorá je kontextovo závislá a v niektorých prípadoch si vyžaduje mierne zložitejšiu obsluhu. Takáto interakcia obvyle prebieha stlačením klávesy `F` vo chvíli, keď sa hráč nachádza pred interaktívnym objektom a hra ho na možnosť jeho použitia upozorní. Vstup do budovy je hráčovi umožnený, len ak sa nenachádza vo vozidle. Reakciou je zopnutie kontrolných prepínačov a volanie metódy `ChangeMap()` Game Manageru.

Ak má hráč práve v zornom poli použiteľný objekt (InputComponent obdržal správu ITEM_INTERSECTED), Component si uchováva id tohto objektu ako aj ukazateľ naň. Ak sa hráč rozhodne interagovať, je potom možné jednoducho rozlíšiť, o aký typ sa jedná – collectibleActor, treasureActor, alebo vehicleActor.

- *collectibleActor* – Reakciou je vyslanie správy ITEM_COLLECTED s identifikáciou objektu, na čo reaguje ako zasiahnutý actor (svojím „odstránením sa“), tak GUIComponent zobrazením príslušnej ikony a nápovedy. Ďalej dochádza k zmene hodnoty premennej v inventári.

- *treasureActor* – Najskôr je nutná kontrola, v akom stave sa poklad nachádza, teda či už bol vykopaný. V prípade, že nie, prebehne kontrola hráčovho inventáru pre prítomnosť lopaty a buďto dôjde k vyvolaniu zobrazenia nápovedy, alebo k použitiu pokladu. Ak už je vykopaný, v závislosti na prítomnosti alebo neprítomnosti kľúča v hráčovom inventári dôjde k úspešnému, resp. neúspešnému ukončeniu hry. Ak sa jedná o neúspech, je vyvolaný výbuch pokladu, znehybnený hráč vypnutím jeho MotionModelu a nastavený časovač, po ktorého uplynutí dôjde k zobrazeniu záverečnej obrazovky informujúcej o prehre. Časovač je zaradený, aby bolo hráčovi umožnené vidieť výbuch a nedošlo k okamžitému vytrhnutiu z reality hry.
- *vehicleActor* – Nastúpenie do vozidla spočíva vo vypnutí kolízneho modelu hráča, ako aj MotionModelu, ktorý zabezpčuje jeho pohyb po scéne. Ďalej dôjde k zapnutiu MotionModelu reprezentujúceho pohyb vozidla a jeho priradeniu ku konkrétnemu vehicleActorovi. Hráč je následne „presunutý“ za vozidlo a je vytvorený príbuzenský vzťah, čím je zabezpečený pri ovládaní vozidla pohľad zozadu (keďže kamera je už pripojená a k entite hráča).

Jedinou akciou, ktorú môže hráč vyvolať počas ovládania vozidla, je vystúpenie. Spočíva vlastne v reverzii akcií učinených pri nastupovaní.

4.6 Logika spustenia aplikácie

Aplikácia pozostáva z dvoch základných častí – spustiteľného binárneho súboru a dynamických knižníc obsahujúcich všetky komponenty, actors a logiku aplikácie (jedná sa o tie isté knižnice, ktoré sú použité pri tvorbe hernej mapy v editore STAGE).

V knižnici dôjde ako k prvému k vyhľadaniu triedy odvodenej od `dtGame::GameEntryPoint` a v nej k postupnému volaniu metód `Initialize()`, `CreateGameManager()` a `OnStartup()` v tomto poradí.

- V metóde `Initialize()` obvykle dochádza k spracovaniu parametrov príkazového riadka. Je to tiež ideálne miesto pre akúkoľvek konfiguráciu, ktorú je nutné vykonať prv ako čokoľvek iné. Obvyklým je nastavenie kontextu projektu a dátových ciest (ktoré obvykle z priečinka predstavujúceho kontext projektu vychádzajú), resp. konfigurácia okna aplikácie.

```
dtDAL::Project::GetInstance().SetContext(''..'');
dtCore::SetDataFilePathList(dtDAL::Project::GetInstance().GetContext());

app.GetWindow()->SetWindowTitle(''IBP - Delta3D'');
app.GetWindow()->SetFullScreenMode(false);
```

- `CreateGameManager()`, ako už názov napovedá, slúži pre vytvorenie samotnej inštancie Game Manageru. Preťažovanie tejto metódy je nutné len v prípade tvorby vlastnej odvodenej triedy, inak vznikne štandardne `dtGame::GameManager`.
- `OnStartup()` je potom začiatkom samotnej vlastnej činnosti aplikácie. V tejto časti dochádza k vytvoreniu a pripojeniu komponent, registrácii užívateľských typov správ, nastaveniu kontextu projektu a načítaniu mapy. Demonštračná aplikácia využíva okrem DefaultMessageProcessoru dve ďalšie aplikačne špecifické Componenty, konkrétne `GUIComponent` pre vykreslľovanie GUI (viď 4.5.1 – 33) a `InputComponent` pre spracovanie užívateľského vstupu a zároveň ako jadro starajúce sa o pravidlá hry a pod. (viď 4.5.2 – 36).

Skrátený príklad obsahu preťaženej metódy `OnStartup()`:

```
void IBPGameEntryPoint::OnStartup(dtGame::GameApplication &app)
{
        dtAudio::AudioManager::Instantiate();
        app.GetCamera()->AddChild(dtAudio::AudioManager::GetListener());

        dtGame::GameManager& gameManager = *app.GetGameManager();

        gameManager.GetMessageFactory().RegisterMessageType<dtGame::Message>
        (MyMessageType::STATE_MENU);

        dtCore::RefPtr<dtGame::DefaultMessageProcessor> dmp =
        new dtGame::DefaultMessageProcessor(''DefaultMessageProcessor'');
        dtCore::RefPtr<GUIcomponent> GUIcomp =
        new GUIcomponent(*app.GetWindow(), *app.GetKeyboard(),
                        *app.GetMouse(), ''GUI'');
        dtCore::RefPtr<InputComponent> InpComp = new InputComponent(''Input'');

        gameManager.AddComponent(*dmp,
        dtGame::GameManager::ComponentPriority::HIGHEST);
        gameManager.AddComponent(*InpComp,
        dtGame::GameManager::ComponentPriority::NORMAL);
        gameManager.AddComponent(*GUIcomp,
        dtGame::GameManager::ComponentPriority::NORMAL);

        gameManager.ChangeMap(''Map'');

        dtCore::RefPtr<dtGame::Message> msg =
        gameManager.GetMessageFactory().CreateMessage(MyMessageType::STATE_MENU);
        gameManager.SendMessage(*msg);
}
```

Keďže je v aplikácii využitá práca so zvukom, hneď na začiatku dochádza k inicializácii AudioManageru a vytvoreniu väzby medzi kamerou a Listenerom, čím je zabezpečené, že zvuky k hráčovi doliehajú akoby sa nachádzal na pozícii kamery v scéne. Následne registrujeme u MessageFactory, ktorá je súčasťou Game Manageru, všetky správy, ktoré budeme používať pre komunikáciu medzi objektmi.

V ďalšom kroku vytvoríme a pripojíme Componenty. DefaultMessageProcessoru je priradená najvyššia priorita, ostatné Componenty sú postavené na rovnakú úroveň.

Napokon GameManageru povieme, že na úvod si prajeme načítať mapu s menom „Map“ a vytvoríme a odošleme správu o tom, že aplikácia sa momentálne nachádza v stave úvodného menu, na čo zodpovedajúco zareagujú Componenty – dôjde k načítaniu a inicializácii Actorov obsiahnutých v prvej mape a k zobrazeniu úvodného menu. Od tejto chvíle je potom beh aplikácie plne v rukách užívateľa a hra sa môže začať.

Kapitola 5

Záver

Výstupom práce je jednoduchá aplikácia demonštrujúca využitie základných možností a mechanizmov herného a simulačného enginu Delta3D a podrobný popis týchto mechanizmov, ako aj samotnej tvorby aplikácie. Projekt osvetľuje základné zákonitosti fungovania moderných herných enginov a zároveň upozorňuje na potenciál spájania a využívania výstupov rôznych opensource iniciatív na poli vizualizácie a simulácie.

Možnosti poskytované knižnicou, ako aj knižnica samotná, sú ozaj rozsiahle a rôznorodé. Ponúka sa tak široké pole pre smerovanie vývoja aplikácií, či už v podobe rozsiahlych simulácií, realistického zobrazovania alebo tvorby pokročilých 3D počítačových hier. Využiteľnosť a miera uplatnenia enginu Delta3D v praxi má aj vďaka samostatnému životu a neustálemu zdokonaľovaniu sa začlenených projektov potenciál narastať.

Prípadné pokračovanie a rozširovanie projektu (ako po stránke objemovej, tak v zmysle začleňovania pokročilejších grafických možností a prepracovanejších herných pravidiel) môže viesť k vzniku plnohodnotnej hernej alebo simulačnej aplikácie a ponúka zaujímavé horizonty kreatívnej realizácie.

Literatura

[1] Graphical Particle Editor Reference [online]. http://www.delta3d.org/filemgmt_data/files/Particle%20system.pdf, [cit. 2011-04-10].

[2] BMH Associates, Inc.: Delta3D Level Editor: Dynamic Actor Layer version 1.2 (Software Design Document) [online]. http://www.delta3d.org/filemgmt/visit.php?lid=21, 2005-08-29 [cit. 2011-04-10].

[3] BMH Associates, Inc.: Delta3D: Project & Map: Load, Save and Archive version 1.2 (Software Design Document) [online]. http://www.delta3d.org/filemgmt/visit.php?lid=19, 2005-08-29 [cit. 2011-04-10].

[4] BMH Associates, Inc.: Delta3D Game and Simulation Engine: Game Manager version 1.0 (Software Design Document) [online]. http://www.delta3d.org/filemgmt/visit.php?lid=21, 2005-10-25 [cit. 2011-04-10].

[5] WWW stránky: Delta3D – Open source gaming & simulation engine. http://delta3d.org/, [cit. 2011-04-10].

[6] WWW stránky: Delta3D – Partners. http://www.delta3d.org/article.php?story=20041110112606610&topic=about, [cit. 2011-04-10].

[7] WWW stránky: Delta3D – Extras. http://sourceforge.net/projects/delta3d-extras/, [cit. 2011-05-01].

[8] WWW stránky: Delta3D – GM Tutorial part 1 and 2. http://www.delta3d.org/article.php?story=20060620123144266&topic=tutorials, [cit. 2011-05-01].

[9] WWW stránky: Delta3D: Delta3D API Documentation. http://delta3d.sourceforge.net/API/html/index.html, [cit. 2011-05-08].

[10] WWW stránky: Delta3D – Delta3D Knowledge Base. http://www.delta3d.org/article.php?story=20051207101455773&topic=docs, [cit. 2011-05-09].

Dodatek A

Obsah CD

Obsahom priloženého CD sú zdrojové súbory vytvorené autorom tohto textu ako aj konfiguračný súbor pre systém CMake umožňujúci preklad týchto súborov za predpokladu prítomností knižníc Delta3D. CD tiež obsahuje kompletnú adresárovú štruktúru so spustiteľným súborom a všetkými knižnicami a dátami nutnými k bezproblémovému behu aplikácie. Aplikácia je určena pre operačný systém Windows (testovanie prebehlo na platformách Windows 7 a Windows XP).

Všetky obsiahnuté súbory boli buďto vytvorené priamo pre potreby tejto aplikácie, sú súčasťou distribúcie Delta3D, alebo sú voľne dostupné na internete ako stock.

Adresárová štruktúra zahŕňa tento obsah:

- Zdrojové kódy vytvorené pre potreby tejto práce.
- Spustiteľná aplikácia a všetky jej súčasti vrátane externých knižníc nutných k spusteniu aplikácie.
- Pdf verzia tohto dokumentu pod názvom *IBP.pdf*, zdrojové súbory potrebné pre znovuvytvorenie tejto práce.

Dodatek B

Manuál

B.1 Pozadie hry

Úlohou hráča je nájsť, vykopať a otvoriť truhlicu s pokladom nachádzajúcu sa v okolí malej opustenej osady. K tomuto účelu je mu umožnené, resp. je od neho požadované, aby preskúmal herný svet a pokúsil sa s ním vhodne interagovať.

B.2 Úlohy

Hráč musí pre úspešné ukončenie hry vykonať niekoľko krokov:

- Nájsť a zobrať si lopatu, ktorá sa nachádza medzi domami.
- Použiť vchod do najväčšej budovy a vstúpiť dovnútra.
- Vo vnútri budovy nájsť a zobrať si kľúč. V tme si môže svietiť baterkou.
- Vrátiť sa von a za pomoci zvukovej signalizácie (detektoru) nájsť poklad.
- Využiť zozbierané predmety k vykopaniu a odomknutiu pokladu.

V rámci aplikácie je z hlavného menu a aj kedykoľvek v priebehu hry zobraziteľná obrazovka informujúca o ovládaní hry a hráčových úlohách.

B.3 Ovládanie

Ovládanie aplikácie je realizované prostredníctvom klávesnice a myši.

Peší pohyb	W, A, S, D
Pohyb vozidla	Kurzorové klávesy
Interakcia s prostredím	F
Detektor (radar) v exteriéri	R
Baterka v interiéri	L
Prerušenie hry/Nápoveda	Esc

Tabulka B.1: Ovládanie aplikácie