Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Eclipse engine

Plzeň, 2010 Michal Skalský

Prohlášení

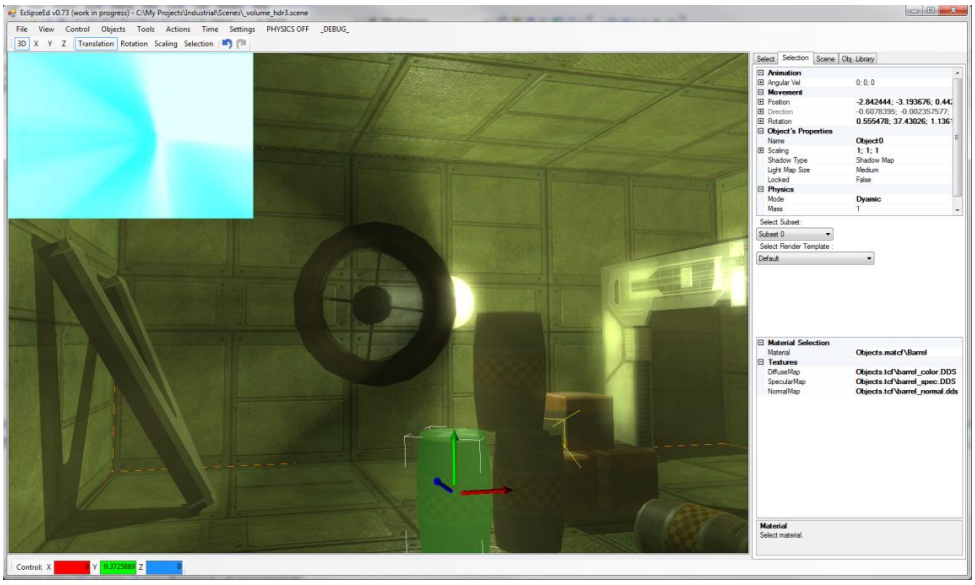
Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 13. 6. 2010 Michal Skalský,

Abstrakt

Eclipse engine

The goal of this project is to design and implement 3d game engine. There are three main parts of this document. First part servers as introduction to the game engine theory. Typical game engine architecture will be presented here. Several existing independent game engines will be discussed in the second part. And finally, design and implementation of my own game engine will be presented in the last section. The name of this engine is Eclipse and I will walk the reader through all important aspects behind Eclipse implementation. Eclipse world editor (EclipseEd) and technology demo based on Eclipse Engine (EclipseDemo) will be both presented in this last section too. EclipseEd and EclipseDemo are shown on pictures 1 and 2 respectively.



Obrázek 1: EclipseEd (Picture In Picture)

Obrázek 2: EclipseDemo

Obsah

1 Úvod

V této práci se budu zabývat herními enginy (viz 3). Uvedení do problému herních enginů bude obsahem úvodní části. Pokusím se zde stručně pokrýt některé základní pojmy, které budou dále používány. Současně zde bude také nastíněna architektura běžného herního enginu. V druhé části popíši některé už existující nezávislé herní enginy. Po této části bude následovat podrobnější popis mnou navrženého enginu. Tento engine nese jméno Eclipse a není to zcela kompletní herní engine. Konkrétně byly navrženy a implementovány následující moduly: grafika, fyzika, vstupní zařízení a skriptování.

2 Co je to hra?

V této kapitole krátce popíši, co je to hra z pohledu počítače. Ralph Koster ve své knize ***A Theory of Fun for Game Design*** definuje hru jako interaktivní zážitek, který staví hráče před sérii překážek se zvyšující se obtížností. Hráč tyto překážky poznává a postupně se je učí překonávat. A právě tento proces učení se a zlepšování je to, co považujeme za zábavu. Pro účely této práce se omezíme na běžné 2d a 3d počítačové hry jako jsou hry viděné z pohledu vlastních očí (first person shooter – FPS), akční hry viděné z pohledu třetí osoby (third person action), závodní hry, bojové hry atd. Většina těchto počítačových her by se dala popsat jako „soft real-time interactive agent-based computer simulation“. Postupně se pokusím tento termín rozebrat. Ve většině her se vezme nějaká část reálného nebo smyšleného světa. Tato část světa se matematicky modeluje, aby s ní mohl zacházet počítač. Tento matematický model je typicky výrazně zjednodušený oproti realitě (i té smyšlené), protože zjevně nemá smysl modelovat svět na úrovni atomů. Matematický model je tedy simulací (*simulation*) reálného nebo smyšleného světa. Toto zjednodušení je v počítačových hrách nesmírně důležité, protože i velmi zjednodušený model může být pro člověka téměř nerozeznatelný od reality, pokud se použije chytře. Ve většině her se dále vyskytuje řada entit (*agent*), které mezi sebou nějakým způsobem interagují. Tyto entity představují ve hrách vozidla, postavy, projektily atd. Simulace je tedy založená na interakci těchto entit (*agent-based computer simulation*). Všechny hry reagují nějakým způsobem na vstup od hráče. Jsou tedy interaktivní (*interactive*). Tato reakce na vstup od hráče probíhá u většiny her v reálném čase (real-time). Jednotlivé simulace mají určitý termín, který musí dodržet. Například se musí nakreslit třicet snímků za vteřinu, aby se dosáhlo iluze pohybu. Simulace fyziky se musí provést třeba 180 krát za vteřinu. U real-time systému, který je „soft“, nedodržení časového limitu není katastrofické. Pokles počtu snímků za sekundu zpravidla nezpůsobí smrt uživatele. Opakem soft real-time systému jsou hard systémy, u kterých nestihnutí termínu muže způsobit zranění, nebo smrt uživatele. Celá výše popsaná simulace se provádí na počítači, a tedy v diskrétním čase. Typicky je zde jedna nekonečná smyčka a v každé iteraci se postupně vyhodnocují jednotlivé

subsystémy jako je simulace fyziky, získání vstupu od uživatele, výpočet reakce uměle inteligence, vykreslení snímku na obrazovku atd.

3 Co je to herní engine?

Herní engine je software sloužící k vývoji videoher. Termín herní engine se poprvé začal objevovat v polovině devadesátých let ve spojitosti s hrami typu FPS. Dobrým příkladem byla hra Doom od Id Software. Tato hra měla rozumným způsobem oddělené jádro (vykreslování 3d grafiky, detekce kolizí, audio systém…) a vlastní náplň hry (prostředí, zvuky, pravidla hry…). Toto oddělení se ukázalo být velmi výhodné, když Id Software začal toto jádro licencovat dalším firmám. Ty se nemusely starat o nízko úrovňové aspekty hry, a stačilo jim jenom vytvořit vlastní prostředí a vlastní pravidla hry. To velmi zrychlilo a zlevnilo vývoj hry. Dnešní herní enginy se nezaměřují jenom na jeden typ hry, ale jsou více univerzální. Princip ale zůstává stejný. Herní engine představuje znovupoužitelnou část hry. Poskytuje základní i pokročilejší funkcionalitu, která je víceméně společná všem hrám. Samozřejmě neexistuje pevná hranice mezi hrou a herním enginem. Některé enginy se snaží o velkou znovu použitelnost, zatímco jiné jsou ušité na míru konkrétní hře a rozdíl mezi tím, co je ještě engine, a co už je hra, je velmi nejasný. Dále je třeba si uvědomit, že úplně universální engine neexistuje a pravděpodobně nikdy ani nebude. Dá se zde vypozorovat jistý trend. Čím pokročilejší (více vysoko úrovňové) funkce engine nabízí, tím více urychluje vývoj aplikace, ale zároveň se stává méně univerzálním. Skutečně univerzální řešení se tak dá dosáhnout jenom na velmi nízké úrovni, takový engine ale zase vývoj příliš neurychlí. Většina enginů je více či méně přizpůsobena pro potřeby určitého typu/typů hry. Pro tento typ hry poskytují vysoko úrovňové funkce a tvůrci hry se mohou omezit pouze na vlastní náplň hry. Častým jevem je také přizpůsobování existujícího enginu pro potřeby konkrétního projektu. Při tomto procesu se použije jenom část enginu, která vyhovuje a zbytek se upraví/přidá tak, aby vyhovoval vyvíjené aplikaci.

Herní engine je tedy znovupoužitelná část hry, která zrychluje a zleňuje vývoj nových her. Tento engine si tým může vyvinout sám pro potřeby několika vlastních projektů, nebo si ho může licencovat od jiné firmy.

4 Základní architektura

V této kapitole si popíši základní architekturu běžného herního enginu tak, jak je nastíněna v (1). Popíšu zde jenom hrubé rozdělení do hlavních částí. Žádné podrobnosti zde nemají smysl, protože jsou v každém enginu implementovány jinak. O architektuře herního enginu existuje jenom minimum literatury. Většina tvůrců detaily svých enginů úzkostlivě tají. Dají se tedy očekávat velké rozdíly mezi jednotlivými enginy.

Běžný herní engine se skládá z části, která bude pohánět hru (runtime) a z nástrojů, které pomáhají při vývoji. Nejdříve se podíváme na vlastní engine a v další části se zastavím nad nástroji.

4.1 Runtime

Na obrázku 3 je ukázka runtime části enginu. Stejně jako každý větší software je i engine postavený na vrstvách. Správně vyšší vrstvy závisí /využívají nižší vrstvy. Opačná závislost by se neměla objevovat. Následuje stručný popis jednotlivých vrstev z obrázku.

4.1.1 Cílový hardware

Tato vrstva reprezentuje typ hardwaru, na kterém bude možné hru spustit. Typické platformy jsou Microsoft Windows, Macintosh, Microsoft Xbox, Sony Playstation atd. Je dobré zmínit, že některé enginy jsou platformě nezávislé. Tuto funkci plní vrstva platformní nezávislosti (viz 4.1.5).

4.1.2 Ovladače

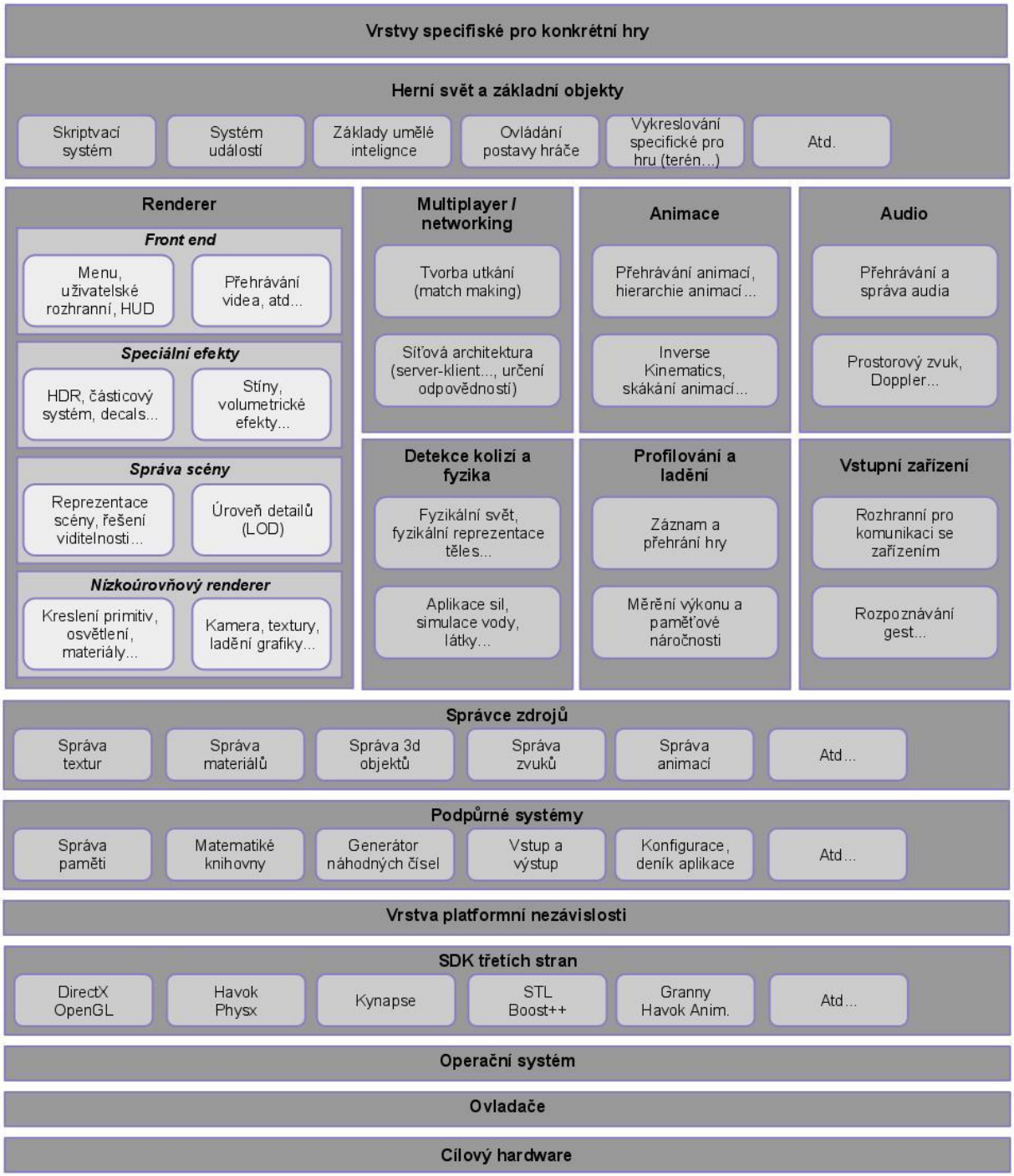
Ovladače zařízení jsou nízko-úrovňové softwarové komponenty poskytované operačním systémem nebo výrobcem hardwaru. Ovladače se starají o určitý hardware a odstiňují operační systém a vyšší vrstvy od detailů komunikace s tímto hardwarem.

4.1.3 Operační systém

Na osobním počítači je operační systém rozsáhlá aplikace poskytující tisíce funkcí. S těch velmi důležitých zmíním jenom automatickou správu paměti. Oproti tomu operační systém na konzoli je často jenom knihovna funkcí, která se přímo přeloží do vytvářené hry. S tím souvisí fakt, že taková hra plně „ovládá“ konzoli, je to jediná aplikace, která je na ní spuštěna. U nových konzolí toto už ale úplně neplatí. Jak Xbox 360, tak Playstation 3 mají jednoduché operační systémy, které mohou přerušit běh hry a nárokovat si potřebné zdroje. Nicméně ani nejnovější konzole neobsahují správu paměti a všechny hry, které na nich běží, se o tuto správu musejí starat sami.

4.1.4 SDK třetích stran

Software Development Kit (SDK) je sada vývojových nástrojů, které usnadňují tvorbu nějaké konkrétní aplikace. Většina herních enginů využívá řadu SDK třetích stran. Důvod je prostý. Stejně jako engine funguje jako znovupoužitelná část hry, tak SDK slouží jako znovupoužitelná část enginu. Každý engine potřebuje určité funkce, které jsou společné více/všem enginům. Nemá smysl tyto části implementovat neustále dokola, pokud jsou vždy stejné. Příklady některých často používaných SDK jsou uvedeny v kapitole 4.3.



Obrázek 3: Architektura enginu

4.1.5 Vrstva platformní nezávislosti

Většina herních enginů musí fungovat na více než jenom jedné platformě. Cílem je samozřejmě pokrytí co možná největší části trhu. K tomu slouží tato vrstva. Vrstva platformní nezávislosti je položena nad vrstvou hardwaru, ovladačů, operačního systému a nad vrstvou SDK třetích stran. Jejím úkolem je odstínit vyšší vrstvy od platformě závislých nižších vrstev. V praxi je to tedy společné rozhranní, které vyšší vrstvy využívají. Pro každou platformu existuje vlastní implementace tohoto roz-

hranní. To je zcela nezbytné, protože mezi jednotlivými platformami jsou obrovské rozdíly. Dokonce i standardní knihovna jazyka C je na každé platformě jiná. Bez této vrstvy by se musela napsat pro každou platformu vlastní verze enginu. To samozřejmě není moc elegantní řešení.

4.1.6 Podpůrné systémy

Sem spadají části enginu, které jsou využívány všemi dalšími vrstvami. Poskytují ty naprosto základní funkce. Patří mezi ně například:

- Správa paměti. Každá hra potřebuje rychlého a efektivního správce paměti. Ten se stará o alokaci a dealokaci paměti a musí se umět vypořádat s fragmentací paměti, ke které nevyhnutelně bude docházet. Pouze enginy, které jsou určené výhradně pro platformy s vyspělým operačním systémem, si mohou dovolit spolehnout se na správce paměti tohoto systému. Například správce paměti operačního systému Windows se ukázal jako dostatečně výkonný na to, aby byl použit v počítačové hře. Většina enginů nicméně využívá vlastní správce, které jsou navrženy přesně s ohledem na potřeby enginu.
- Matematická knihovna. Počítačové hry jsou z principu postavené do značné míry na matematických výpočtech. Proto každý engine potřebuje matematickou knihovnu. Ta poskytuje funkce pro práci s maticemi, vektory, quaterniony a mnoha geometrickými útvary.
- Vlastní datové struktury a algoritmy. Například doplnění nějakého SDK zabývajícího se datovými strukturami o další datové struktury a algoritmy nad nimi.
- Měření času. Engine potřebuje mít schopnost měřit čas ve vysokém rozlišení pro potřeby dalších systémů. Kromě měření času může obsahovat i nějaké funkce pro manipulaci s časem.
- Synchronní/asynchronní zápis/čtení ze souborů. Vedle základní práce se soubory je dnes často nutné umožnit načítání/ukládání dat asynchronně.
- Generování náhodných čísel. Funkce pro generování čísel v nějakém rozsahu, náhodných čísel s různým rozdělením, generování náhodných barev, vektorů…
- Konfigurační soubory. Čtení a zápis souboru s konfigurací enginu.
- Debuging a logging. Nástroje pro hledání chyb a manipulace s deníkem (log) aplikace.
- Chytrý ukazatel, handle, unikátní identifikátor. Použití k zabezpečení a zjednodušení práce s dynamickými objekty.

4.1.7 Správce zdrojů

Každá hra využívá mnoho dat. Potřebuje textury, 3d modely, zvuky, animace… Správce zdrojů se o tyto data musí starat a musí je zpřístupnit ostatním subsystémům. Některé enginy používají centralizovaný přístup, kdy jsou všechny zdroje dostupné přes jedno rozhranní. Jiné enginy poskytují sérii rozhranní pro různé typy zdrojů.

4.1.8 Renderer

Obrázek 4: Renderer

Vykreslování (renderování) je proces generování obrazu z výchozích datových struktur (3d modely, textury, osvětlení). Část enginu, která je odpovědná za vykreslování se jmenuje renderer. U typického enginu představuje renderer jeden z nejrozsáhlejších a nejkomplexnějších subsystémů. Renderer je část, která se dá navrhnout mnoha různými způsoby a neexistuje žádný preferovaný nebo správný způsob. Proto se renderery v různých enginech často velmi liší. Design rendereru se také výrazně mění v průběhu času. Je to dáno vývojem a proměnami grafického hardwaru. Právě design grafického hardwaru do značné míry ovlivňuje design rendereru, a díky tomu je možné vypozorovat určité společné rysy většiny moderních rendererů.

Běžný způsob návrhu rendereru je rozdělení celého systému do několika vrstev. Vrstvy jsou znázorněny na obrázku 4.

Nízko-úrovňový renderer

Nízko-úrovňový renderer neřeší, která část scény se má vykreslit. Jeho úkolem je vykreslit zadané trojúhelníky (nebo jiná geometrická primitiva) jak nejrychleji to jenom jde. Musí se přitom starat o nastavení materiálu, textur, světel a kamery. Nízko-úrovňový renderer si interně uspořádá jednotlivé objekty tak, aby se vykreslily co nejrychleji. To znamená zamezit zbytečnému přepínání materiálů a textur, zabránit zbytečným změnám stavu grafického zařízení, třídit objekty podle vzdálenosti od kamery, aby se omezilo překreslování atd. Mezi další funkce nízko-úrovňového rendereru patří správa grafického zařízení. To zahrnuje enumeraci dostupných zařízení, jejich inicializaci a další práci s těmito zařízeními. Mezi nejdůležitější funkce grafického zařízení bude patřit nastavení cíle vykreslování, prezentování vykresleného obrazu na zobrazovací zařízení a manipulace se stavem zařízení.

Správce scény

Nízko-úrovňový renderer vykreslí objekty bez ohledu na jejich umístění ve scéně, nebo bez ohledu na jejich viditelnost. Pro tyto účely existuje vrstva, která se stará o reprezentaci a správu celé scény. Z pohledu rendreru je jejím hlavním úkolem zjistit, které objekty jsou viditelné z aktuální kamery, a ty následně poslat do nízko-úrovňového rendereru. Existuje mnoho způsobů, jak toto zjistit. Pro

velmi jednoduchou scénu může být plně dostačující testovat všechny objekty na zorné pole kamery (frustum). Pro složitější scény je potřeba navrhnout efektivnější metody. Využívají se k tomu různé způsoby dělení prostoru. Toto dělení prostoru probíhá často hierarchicky pro ještě větší urychlení. Cílem všech metod dělení prostoru je rychle určit části prostoru, které nejsou ze současné kamery viditelné, protože tím vyřadíme i všechny objekty v nich obsažené. Mezi typické metody dělení prostoru patří BSP, Octree, Quadtree a Portal culling (více informací například v (2)).

Kromě dělení prostoru se v této fázi také často řeší určení objektů, které jsou zastíněné jiným objektem, a nejsou proto viditelné (Occlusion culling). Správce scény po určení viditelných objektů ještě určuje jejich množství detailů (Level of Detail - LOD). Existují dva typy. Geometry LOD (GLOD) – pro každý objekt existuje několik 3d modelů s různým množstvím detailů (různý počet trojúhelníků) a Material LOD (MLOD) – každý materiál obsahuje několik variant s různým množstvím detailů a různou výpočetní náročností. LOD pro každý objekt/materiál se obvykle určuje podle vzdálenosti od kamery. Vzdálené objekty, které nejsou téměř vidět, nemusí být vykreslovány se všemi detaily. LOD je tedy další nastroj pro zvýšení výkonu vykreslování.

Speciální efekty

Další vrstva rendereru, která se pro změnu stará o speciální efekty. Moderní engine využívá řadu efektů. Například:

- Částicový systém. Pro simulaci kouře, ohně atd.
- Systém kreslení otisků (Decals). Pro renderování stop, děr po kulkách atd.
- Následné zpracování (Post-processing). Například kreslení s velkým dynamickým rozsahem (High Dynamic Range - HDR), rozostření pohybujících se objektů (Motion Blur), korekce barev, hloubka ostrosti (Depth of Field)…

Front End

Většina enginů poskytuje funkce pro vykreslování 2d grafiky. To se používá pro následující účely:

- Zobrazení různých informací bez zakrytí vlastní scény (Heads-Up Display - HUD).
- Tvorba menu ve hře.
- Grafické uživatelské rozhranní ve hře (Graphical User Interface - GUI). Například rozhranní pro manipulaci s inventářem postavy.

Do této vrstvy se také ještě někdy zahrnuje systém přehrávání videa (FMV), pokud ho engine poskytuje.

4.1.9 Profilování a ladění

Počítačové hry musí zaručit odezvu v reálném čase. Aby bylo možné toto dosáhnut, je často nutné měřit rychlost vykonávání různých částí aplikace. Díky tomuto profilování je možné odhalit místa, která způsobují zpomalení (bottleneck) a provést patřičné optimalizace. Kromě toho musí každá hra

počítat s omezeným množstvím paměti. Pro efektivní využívání paměti se měří spotřeba paměti různých částí systému. Tato informace se dá opět využít k optimalizování. Existuje celá řada nástrojů třetích stran, které se dají použít k profilování her. Například:

- VTune od Intelu.
- PurifyPlus od IBM.
- PIX od Microsoftu. Speciálně navržené k profilování DirectX aplikací.
- PerfHUD od NVidie. Vynikající nástroj na profilování grafického výkonu. Dá se použít jenom s grafickými kartami NVidie a speciálními ovladači.

Kromě těchto „univerzálních“ nástrojů má většina enginů ještě vlastní nástroje, které umožní přesně měřit jenom určitou část kódu, nebo přesně určit paměťové nároky jednoho konkrétního subsystému. Tyto informace jsou často dostupné přímo při běhu aplikace formou výpisu na obrazovku.

4.1.10 Detekce kolizí a fyzika

Detekce kolizí je naprosto nezbytná ve většině her. Bez ní by objekty procházely jeden druhým a bylo by nemožné se jakkoliv pohybovat v herním světě. Zatímco bez detekce kolizí se prakticky žádná hra nemůže obejít, tak simulace fyziky už není nezbytnou součástí. Na tomto místě bych ještě podotkl, že simulací fyziky ve hrách se téměř vždy myslí dynamika pevných těles (Rigid Body Dynamics). Pravdou ale je, že detekce kolizí a výpočet fyziky jsou velmi těsně spjaty. Nalezené kolize se musí vyřešit jako součást simulace fyziky. Dnes už si prakticky žádná firma nepíše vlastní systém detekce kolizí a simulace fyziky. Převážná většina enginů používá už hotová řešení třetích stran, které nabízejí detekci kolizí i simulaci fyziky v jednou uceleném balení. Tři nejpoužívanější systémy jsou Havoc, PhysX a ODE (viz 4.3).

4.1.11 Animace

Každá hra, která využívá pohyblivé charaktery (lidi, zvířata, roboty…), potřebuje systém animace. Ve hrách se využívají následující typy animací:

- Sprite/texture animation
- Rigid body hierarchy animation
- Skeletal/skinned animation
- Vertex animation
- Morph targets

Všechny tyto typy animací budou stručně popsány v kapitole 6.8 . Asi nejrozšířenější způsob animování postav v moderních hrách je Skeletal himation. Animátor zde může pomocí relativně jednoduchého systému kostí (bones) vytvářet i velmi komplexní animace. Druhou velmi rozšířenou metodou je Morph target, který se používá především na animování obličejů. Animační systém musí zvlá-

dat především věrohodné kombinování a načasování různých animací. Také je důležité doladění předpřipravené animace podle okolních podmínek (např. ukotvení chodidla na nerovném povrchu). Kvalitní animace jsou pro hru důležité, protože běžný člověk je schopen poznat i drobné odchylky od reality v pohybu postav.

4.1.12 Vstupní zařízení (Human Interface Device - HID)

Aby byla hra interaktivní, potřebuje zpracovávat vstup od hráče. Mezi běžné vstupní zařízení patří:

- Klávesnice a myš.
- Gamepad a joystick.
- Další specializované zařízení jako volant, kytara, rybářský prut…

Kromě získávání vstupu od uživatele, poskytují některá zařízení také zpětnou vazbu – síla působící v určitém směru (force feedback), vibrace (rumble), nebo zvukové signály (audio feedback).

Některé herní enginy umožňují konfiguraci vstupních zařízení. Například nastavení určitých akcí na různá tlačítka nebo páčky. Mezi další funkce vrstvy vstupních zařízení může patřit rozpoznávání sekvencí a gest (sekvence tlačítek a páček aktivovaných ve správném pořadí v určitém časovém limitu).

4.1.13 Audio

Audio je pro libovolnou hru stejně důležité jako grafika nebo fyzika. Nicméně mu většinou není věnována taková pozornost. Audio systémy v herních enginech se velmi liší v tom, jak pokročilé funkce nabízejí. Od obyčejného přehrávání zvuku, přes prostorový zvuk, až k simulování Dopplerova jevu a charakteristiky prostředí. Audio systém by měl umožňovat přehrávání zvuků i hudby. Měl by umět přehrávání více zvuků najednou, opakované přehrávání atd.

4.1.14 Online Multiplayer / Networking

Mnoho her umožňuje zapojení více hráčů do stejného virtuálního světa. Multiplayer se dá rozdělit do čtyř základních kategorií:

- Single-screen multiplayer. Dva nebo více lidí hrají zároveň na jednom herním zařízení. Všechny charaktery obývají stejný virtuální svět a jsou snímány jednou kamerou, která je musí udržet všechny v záběru, nebo alespoň zřetelně naznačit, kde mimo záběr se hráč nachází. Do této kategorie spadají například hry Lego StarWars nebo Gauntlet. Existuje ještě zvláštní varianta nazvaná hot-seat. Při tomto způsobu hraní se hráči postupně střídají o jeden herní ovladač a jedno herní zařízení.

- Split-screen multiplayer. Více hráčů obývá stejný virtuální svět. Každý má svůj vlastní herní ovladač připojený k jednomu společnému hernímu zařízení. Každý má svoji kameru a obrazovka je rozdělena na několik částí, aby každý hráč viděl svoji postavu.
- Networking multiplayer. Několik počítačů je spojeno do sítě. Každý hráč hraje na svém herním zařízení.
- Massively multiplayer online games (MMOG). Stovky až tisíce hráčů mohou hrát v jednom obrovském stálém herním světě, který je hostovaný na výkonných centrálních serverech.

4.1.15 Herní svět a základní objekty

Tato vrstva obsahuje některé předpřipravené objekty, které jsou potřeba pro většinu herních projektů. Mezi obvyklé typy objektů patří například:

- Objekty statická geometrie, jako jsou budovy, silnice, terén.
- Dynamické objekty. Běžné „předměty“, ze kterých se skládá herní svět. Kameny, židle, barely. Dynamické objekty jsou někdy rozděleny na dva typy. Objekty ovládané simulací fyziky a objekty ovládané manuálně. V některých enginech je jenom jeden typ objektu a simulace fyziky se zapíná/vypíná změnou nějakého parametru.
- Objekty reprezentující charakter hráče
- Objekty reprezentující postavy ovládané počítačem (Non-Player Character)
- Zbraně
- Projektily
- Vozidla
- Dynamická a statická světla
- Kamery
- A mnoho dalších

4.1.16 Systém událostí a zpráv

Herní objekty spolu musí komunikovat. Tato komunikace se dá uskutečnit mnoha způsoby. Asi nejjednodušší způsob je, když objekt, který chce zaslat zprávu, přímo zavolá funkci objektu, který má zprávu přijmout. Další alternativou je systém založený na událostech. To je běžný způsob řešení komunikace mezi objekty. V tomto systému objekt, který odesílá zprávu, vytvoří malou datovou strukturu, zprávu/událost, která obsahuje typ zprávy, id adresáta a další parametry. Tato zpráva se předá systému událostí, který ji doručí cílovému objektu. Mezi výhody tohoto systému patří možnost ukládat zprávy do fronty a jejich postupné zpracování. Někdy je také velmi užitečné doručit zprávu až za nějaký čas po vygenerování. To je se systémem událostí velmi jednoduché.

4.1.17 Skriptovací systém

Mnohé hry využívají skriptovací systémy, aby urychlili a zjednodušili vývoj herní logiky a obsahu. Bez skriptovacího systému je nutné po každé změně znovu přeložit celou aplikaci. To je časově velmi náročné a obzvláště během ladění to může způsobit výrazné zdržení. Oproti tomu znovu přeložení krátkých skriptů netrvá příliš dlouho. Některé enginy dokonce umožňují znovu-nahrání skriptů za běhu aplikace, což ještě více zrychlí vývoj. Skriptovacím jazykům se budu ještě věnovat při popisu skriptování v Eclipse enginu.

4.1.18 Základy umělé inteligence

Dlouhou dobu patřila umělá inteligence výhradně do světa vlastní hry, a nikoli do enginu, který hru pohání. Postupně ale vývojáři začali rozeznávat jisté postupy, které jsou pro každý systém umělé inteligence stejné, a tak se část implementace začala přesouvat do herních enginů. V současné době celá řada enginů poskytuje základní stavební bloky, které usnadňují vývoj umělé inteligence. Systém umělé inteligence může poskytovat například následující funkce:

- Definování cest, po kterých se AI postavy mohou volně pohybovat, aniž by se srazily se statickou geometrií.
- Znalosti přechodů mezi jednotlivými částmi světa.
- Systém hledání cesty (Path-finding) založený na A* algoritmu.
- Zjišťování viditelnosti jiných objektů (s využitím fyzikálního enginu). Případně simulace dalších druhů vnímání (sluch…).
- Vlastní reprezentace světa. Poskytuje informace o jednotlivých entitách - o překážkách, přátelích, nepřátelích… Tyto dodatečné informace pak slouží třeba k vyhýbání se dynamickým objektům.

Jak bude popsáno v kapitole 4.3, existuje i SDK od společnosti Kynogen, který tyto základy umělé inteligence poskytuje.

4.1.19 Vrstvy specifické pro konkrétní hry

Nad výše popsanými vrstvami už jsou zpravidla subsystémy, které jsou navržené pro konkrétní hru, nebo typ hry. Jak už bylo ale v úvodu poznamenáno, nelze nakreslit jednoduše čáru mezi tím, co je herní engine, a co už je vlastní hra. V mnohých enginech budou věci specifické pro konkrétní hru zasahovat i do mnohem nižších vrstev. Záleží na tom, jak moc chce být engine univerzální.

4.2 Nástroje enginu

Každá hra potřebuje velkou spoustu dat ve formě textur, 3d modelů, animací, zvuků, skriptů a mnoha dalších. Většina herních enginů používá specifické formáty těchto dat, a je tedy nutné, aby enginy poskytovaly nástroje na tvorbu dat, nebo alespoň jejich konvertování z jiných formátů. Pro

tvorbu dat se používají speciální nástroje (Digital Content Creation - DCC). Obvykle se používají jak nástroje třetích stran, tak nástroje navržené přímo pro engine. DCC aplikace je často určená pro tvorbu jednoho typu obsahu, i když to určitě neplatí vždy. Například Maya a 3ds Max od Autodesku jsou schopné vytvářet 3d modely i animace. Photoshop od Adobe je zaměřen na tvorbu a editování textur a SoundForge se zase používá na tvorbu audio klipů. Oproti tomu některé věci se nedají vytvořit pomocí DCC třetích stran. Většina enginů například používá vlastní editory pro rozmístění objektů ve virtuálním světě. I když některé týmy k tomuto účelu úspěšně použily 3d Max, nebo jenom textový editor. Dalším typickým příkladem nástroje navrženého přímo pro engine je editor částicových efektů. Způsob, jakým engine zpracovává částicové efekty, se obvykle liší engine od enginu, a tak musí každý engine poskytovat vlastní editor.

Jak bylo uvedeno na začátku kapitoly, většina enginů používá vlastní formáty pro uložení dat. Existují dva základní důvody, proč jsou formáty používané DCC nevhodné pro přímé použití ve hrách.

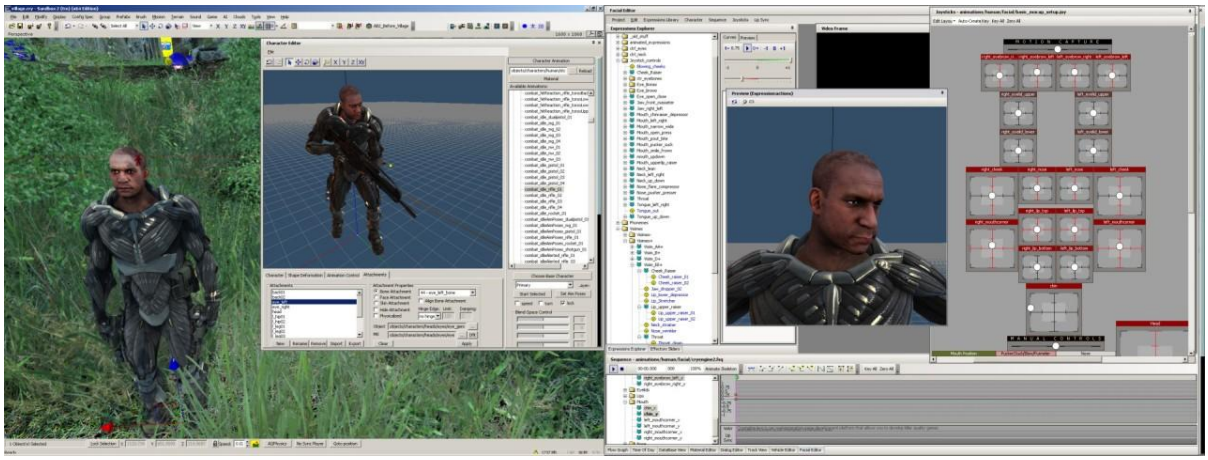
1. DCC aplikace používají formáty, které jsou většinou výrazně komplexnější, než je potřeba pro engine. Například Maya ukládá informace o objektech do orientovaného acyklického grafu (Directed Acyclic Graph - DAG) s mnoha propojeními. Ukládá historii všech změn, které byly udělány. Orientaci, pozici a velikost každého objektu ukládá jako kompletní hierarchii 3d transformací rozdělených na rotace, posunutí, změnu velikosti a zkosení. Herní engine potřebuje jenom naprosté minimum z těchto informací k tomu, aby vykreslil 3d model.
2. DCC formáty jsou typicky velmi pomalé pro načtení. Důvodem je jejich složitost, a také jsou někdy ukládané v textové podobě, která vyžaduje časově náročnou syntaktickou analýzu (Parsing).

Z výše uvedeného plyne, že musí existovat nástroje pro konvertování dat z DCC do formátů, které používá engine.

4.2.1 Architektura nástrojů

V herních enginech se používá několik odlišných metod k tvorbě nástrojů. Některé nástroje mohou být samostatné aplikace, které nejsou na enginu nějak závislé. Některé mohou být postavené na nižších vrstvách, které používá runtime enginu. Další nástroje mohou být přímo zabudované do hry. Například Quake nebo Unreal úspěšně využívají textové konzole ve hře. Ty umožňují vývojářům zadávat řadu příkazů, které poskytují možnosti ladění a konfigurace spuštěné aplikace. Další velmi užitečný postup návrhu nástrojů je postavení nástroje přímo na runtime enginu. To danému nástroji umožní kompletní přístup ke všem datovým strukturám enginu, a vyhne se tak běžnému problému,

kdy je potřeba mít dvě reprezentace každé datové struktury – jednu pro runtime engine a jednu pro vlastní nástroj. Další velmi výraznou výhodou je bezproblémový přechod z editoru do hry, protože hra už je v podstatě spuštěná na pozadí editoru. Díky tomu může designér snadno a rychle zkontrolovat, jak budou jím provedené změny přesně vypadat ve hře. Příkladem tohoto nástroje je editor



Obrázek 5: CryEngine 2 editor

Unreal enginu od firmy Epic - UnrealEd. Dalším krokem je editor, ve kterém je hra doopravdy spuštěná. To, co je vidět v editoru, bude vidět ve stejné podobě i ve hře. Tento způsob ještě více zrychluje vývoj a je to postup, se kterým se v budoucnu budeme pravděpodobně potkávat stále častěji. Příkladem takového editoru je CryEngine 2 od CryTechu (viz obr 5). Nástroje postavené přímo na runtime enginu mají samozřejmě také své nevýhody. Asi největším problémem se může stát stabilita. Pokud není engine stabilní, stávají se nástroje na něm postavené nepoužitelnými. Pokud si tým licencuje již existující engine, tak to pravděpodobně nebude problém, ale někdy se engine vyvíjí současně s hrou, a v takovém případě může dojít k nepříjemným komplikacím.

4.3 Příklady SDK

Zde stručně uvedu příklady SDK, které se používají při vývoji herního enginu. Příklady jsou seřazené podle typu SDK.

Datové struktury a algoritmy

Stejně jako jakýkoliv software i herní engine ve velké míře využívá různé datové struktury a algoritmy pro manipulaci s nimi. Následuje několik příkladů:

- STL (Standard Template Library). Standardní knihovna jazyka C++. Poskytuje řadu datových struktur, práci s řetězci, stream-based I/O a mnoho dalšího.
- STLport. Platformě nezávislá, optimalizovaná implementace STL.
- Boost. Další velmi výkonná knihovna. Velmi podobná STL.

Grafika

Asi všechny herní enginy využívají některé SDK pro kreslení grafiky.

- DirectX. SDK od Microsoftu. Jedno z nejrozšířenějších. Existují verze pouze pro platformy Windows a XBox.
- OpenGL. Velmi rozšířené SDK. Existuje pro celou řadu platforem. Je to hlavní rival DirectX.
- Libgcm. Nízko úrovňové rozhraní pro Sony Playstation 3. Vzniklo jako výkonnější alternativa k OpenGL.

Fyzika a detekce kolizí

Téměř každá hra potřebuje systém detekce kolizí. Velmi často hry využívají i skutečnou simulaci fyziky. Existuje celá řada knihoven, ale následující tři jsou nejvíce používané:

- Havok. Komerční a velmi populární řešení. Obsahuje celou řadu volitelných modulů. Obsahuje asi nejvíce funkcí.
- PhysX. Poslední dobou velmi populární SDK. Poskytuje podobnou sadu funkcí jako Havoc. Verze bez zdrojových kódů je poskytována zdarma.
- Open Dynamics Engine (ODE). Volně šiřitelná knihovna, která byla úspěšně použita v řadě her.

Animace

Zde existuje celá řada produktů. Převážná většina z nich je komerční.

- Granny. Od Rad Games Tools. Obsahuje nástroje na exportování animací a 3d modelů ze všech hlavních 3d studií (3D MAX, Maya…). Vlastní SDK pak řídí všechny aspekty animování těchto modelů.
- Havok Animation. Jeden z modulů Havoc SDK. Cílem je snadný přechod od animovaných charakterů k těm, které jsou řízené fyzikou a naopak. To je v dnešní době čím dál tím více důležité.
- Endorphin a Euphoria. Cílem těchto SDK je vytvoření realistické animace pohybu charakteru s využitím biomechanických modelů.

Umělá inteligence

Ještě nedávno umělou inteligenci řešila každá hra vlastním způsobem, protože ani neexistovala jiná možnost. Společnost Kynogen dnes ale poskytuje SDK nazvaný Kynapse.

- Kynapse. Toto SDK poskytuje nízko úrovňové funkce pro vytvoření umělé inteligence. Například hledání cesty, vyhýbání se statickým a dynamickým překážkám, a také poskytuje rozhranní mezi umělou inteligencí a animací postav.

Audio

Existují opět velmi užitečné nástroje.

- XACT. Pro DirectX existuje výborný XACT. Navíc je k dispozici zcela zdarma.
- SoundR!OT. Electronic Arts interně vyvinuli výkonný audio engine nazvaný SoundR!OT.

- Scream. Sony Computer Entertainment America poskytuje vývojářům na PS3 zvukový engine Scream.

4.4 Shrnutí

V předchozích kapitolách jsem nastínil základní architekturu běžného herního enginu. Tento text se v žádném případě nedá považovat za kompletní rozebrání problematiky. Představuje skutečně jenom velmi stručné nastínění jednotlivých subsystémů herního enginu s popisem typických věcí, za které jsou tyto subsystémy zodpovědné. Dále je třeba uvést, že neexistuje žádný všeobecně uznávaný způsob návrhu herního enginu. Existuje jistě spousta možností, jak se k návrhu enginu postavit. Většina enginů se však bude alespoň do určité míry podobat zde nastíněnému modelu.

V předchozím textu jsem popsal, z čeho se typický herní engine skládá. V následující kapitole budou stručně představeny některé existující a zdarma šířené enginy. A hned v další kapitole se přijde na řadu podrobnější pohled na Eclipse engine.

5 Příklady nezávislých enginů

Zde ve stručnosti představím několik herních enginů. Je vhodné poznamenat, že prakticky žádný nezávislý herní engine není kompletním enginem. Téměř vždy se tvůrci zaměřují jenom na některé části plnohodnotného enginu. Je tomu tak i ve všech zde uvedených příkladech.

5.1 XNA

Microsoft XNA Game Studio je snadno použitelná platforma primárně zaměřená na hráče her. XNA má hráče povzbudit k tvorbě jejich vlastních her a jejich sdílení s herní komunitou. Podobně jako YouTube podporuje tvorbu a sdílení videa. XNA je postavené na jazyce C# a Common Language Runtime (CLR). Vývoj her probíhá ve Visual Studiu nebo Visual Studiu Express. Všechno od psaní zdrojového kódu až po spravování modelů a textur je realizováno uvnitř Visual Studia. XNA podporuje dvě platformy, PC a Xbox 360. Po zaplacení jistého poplatku je možné vytvořenou hru nahrát na servery Xbox Live a sdílet s dalšími uživateli.

XNA není ani zdaleka plnohodnotným enginem. Nicméně při pohledu na výše nastíněnou architekturu můžeme pozorovat několik podobností. Cílový hardware, ovladače a operační systém je buď osobní počítač s Microsoft Windows, nebo Xbox 360. Jako SDK třetí strany využívá XNA DirectX. XNA také obsahuje vrstvu platformní nezávislosti, která je pro uživatele zcela skryta, a která umožňuje současný vývoj na obě podporované platformy. Následuje vrstva podpůrných systémů. Sem můžeme zařadit velmi rozsáhlé knihovny platformy .NET, které jsou plně k dispozici. Následuje správa zdrojů. Ta je v XNA také obsažena, a dokonce je přímo integrovaná do Visual Studia, pro snazší zacházení. Díky DirectX má XNA k dispozici také základní nástroje pro vykreslování geometrie a pro práci se vstupem.

Je tedy vidět, že XNA se dá podle mnou nastíněné architektury považovat za 3d engine, který obsahuje základní nízko-úrovňové funkce. Samozřejmě celá řada důležitých věcí zde chybí, ale to vzhledem k zaměření XNA nepředstavuje větší problém.

5.2 Ogre 3D

OGRE znamená Object-Oriented Graphics Rendering Engine. Jak název napovídá, OGRE se zaměřuje především na grafiku. OGRE je multi-platformní, flexibilní engine. OGRE využívá architekturu zásuvných modulů (plug-in), kdy je možné přes definované rozhraní snadno přidávat nové vlastnosti a funkce. Tato vlastnost z něj dělá značně modulární engine. Celkově je OGRE velice populární engine s čistým, objektově orientovaným designem. Dokazuje to i fakt, že byl použit v několika komerčních hrách. Ve stručnosti teď představím tento engine z pohledu výše popsané architektury herního enginu.

Začneme od vrstvy SDK třetích stran. OGRE přímo využívá DirectX a OpenGL a umožňuje využití prakticky libovolného dalšího SDK. Protože je OGRE multi-platformní, tak zde samozřejmě nemůže chybět vrstva platformní nezávislosti. Podporováno je DirectX, OpenGL, Windows, Linux a MacOS. Mezi podpůrné systémy patří například matematické knihovny a správce paměti včetně vlastního hledání úniků paměti (memory leak). Další vrstvou je správa zdrojů. Ta je u OGREu decentralizovaná. Existuje vzor správce, a pak jeho konkrétní implementace pro každý typ zdroje (textury, materiály…). Dále následuje renderer. OGRE obsahuje jak nízko-úrovňový, tak i vysoko-úrovňový (správa scény) renderer. Konkrétně správa scény je řešena plug-iny. Uživatelé si mohou napsat vlastní. Kromě toho je připravena implementace běžně používaných správců (BSP, OctTree…). Součástí rendereru je i podpora pro tvorbu speciálních efektů. Celý proces renderingu je skriptovatelný pomocí vlastního skriptovacího jazyka. Další podporovanou částí jsou animace. OGRE zvládá Inverse Kinematics, Skeletal Animation a skládání animací (Animation Blending). Více informací o animacích je v kapitole 6.8. Engine OGRE přímo nemá simulaci fyziky, ale má připravené rozhranní pro napojení fyzikálních enginů třetích stran (Physx, ODE…).

Z tohoto popisu je zřejmé, že OGRE je skutečně zaměřen především na grafiku. Obsahuje všechny nižší, podpůrné vrstvy a velmi propracovaný grafický subsystém. Vstupní zařízení, audio, síťová podpora a fyzika zde ale zase chybí. OGRE se nesnaží být plnohodnotným herním enginem. To je zřejmé už z názvu. Soustředí se především na grafiku, která je zde velmi propracovaná.

5.3 Irrlicht

Irrlicht engine je multi-platformní 3d engine napsaný v jazyce C++. Oficiálně podporované platformy jsou Windows, Mac OS X, Linux a Windows CE. Dále existují neoficiální porty na další systémy jako Xbox, PSP, Symbian OS, nebo iPhone. Irrlicht je mimo jiné znám také pro svoji podporu staršího hardwaru. Podobně jako OGRE je i tento engine do značné míry zaměřen především na grafický subsystém. Tím ale podobnost nekončí. Z pohledu architektury jsou oba enginy velmi podobné. Irrlicht také podporuje DirectX i OpenGL. Pochopitelně obsahuje vrstvu platformní nezávislosti, která výrazně usnadňuje vývoj na více platforem. Podobně jako OGRE má i tento engine vlastní matematické knihovny, vlastní knihovnu pro práci se souborovým systémem atp. Správa zdrojů je v Irrlichtu rozdělena na jednotlivé typy zdrojů. Je tedy decentralizovaná stejně jako v enginu OGRE. Irrlicht obsahuje nízko-úrovňový i vysoko-úrovňový renderer. Větší rozdíl je ve způsobu reprezentace scény. Irrlicht využívá hierarchický graf scény, kde jednotlivé prvky grafu mohou být například kamera, 3d model, celá úroveň… Tento graf se postupně prochází a zjišťuje se, co se má vykreslovat. Kromě toho slouží také k detekci kolizí. Irrlich dále podporuje animování 3d modelů. Konkrétně Skeletal Animation a Morph Targets. Oproti OGREu nabízí tento engine i základní detekci kolizí. Ta ovšem není náhradou

za plnohodnotný fyzikální engine. Irrlicht dále implicitně podporuje standardní vstup z klávesnice a myši. Stejně jako engine OGRE ani Irrlicht nepodporuje audio nebo síťové rozhranní.

Irrlicht engine má tedy zřejmě velmi podobnou architekturu jako engine OGRE. Nabízí několik drobností navíc, ale jinak jsou si oba enginy velmi podobné. Oba se zaměřují především na grafický subsystém, který už mají interně řešený odlišně. K enginu Irrlicht se ještě sluší dodat, že kolem něj existuje aktivní komunita, a že díky jeho otevřené formě vznikla řada přídavků. Lidé pro Irrlicht přidali například editor herního světa, podporu audia, fyziky a vytvořili neoficiální porty na další platformy.

6 Eclipse Engine

Zde začíná hlavní část této práce. Tou není nic jiného než popis Eclipse enginu. Před popisem jednotlivých částí Eclipse enginu ale nejdříve musím ujasnit, co Eclipse engine je, a co Eclispe engine není. V dalším textu budu místo zdlouhavého názvu Eclipse engine používat zkratku EE.

6.1 Úvod

V první kapitole jsem krátce předvedl, jak asi běžný engine vypadá. Účelem bylo i ukázat, že herní enginy jsou rozsáhlé a komplexní aplikace. Nebylo by moc rozumné pokusit se v rámci diplomové práce vytvořit kompletní multi-platformní herní engine se všemi funkcemi. Alespoň ne jedním člověkem. Proto jsem se rozhodl vybrat jenom některé části typického enginu. V následující kapitole se pokusím vysvětlit, o jaké části se jedná, a proč byly vybrány. Celý engine je navržen modulárně, aby se dal přizpůsobit konkrétním potřebám. Nicméně modularita v tomto případě vždy neznamená, že se dají nahradit celé subsystémy. Například u grafického enginu je nízko-úrovňový renderer pevnou součástí, protože dobře napsaný nízko-úrovňový renderer bude fungovat stejně dobře ve všech typech her, a není ho tedy potřeba upravovat. Naproti tomu správa scény a algoritmy viditelnosti se výrazně liší podle typu hry a tuto část už si může uživatel enginu implementovat sám. Obecně se dá říct, že části enginu, které budou stejné pro všechny typy her, jsou pevnou součástí enginu. Na závěr úvodu ještě jedna poznámka. Celý engine je napsaný v jazyce C++. Nejdříve ale něco krátce k historii vzniku.

6.1.1 Historie Eclipse Enginu

Pravdou je, že cesta k současné podobě enginu byla velmi dlouhá a strastiplná. Na začátku byl velmi ambiciózní sen vytvořit vlastní počítačovou hru. První kroky ke splnění tohoto snu se dají dohledat na samém konci roku 2001. Tehdy jsem začal poprvé experimentovat s grafikou a s DirectX. Po několika měsících seznamování jsem začal pracovat na své první hře. Byl to vesmírný simulátor. Bylo možné létat vesmírem v kosmické lodi, která stejně jako ostatní lodě podléhala jednoduché fyzice založené na Newtonových zákonech. Ostatní lodě byly řízeny umělou inteligencí a hráč s nimi bojoval. Jak se hra rozrůstala, a přibývaly další funkce, začaly se věci komplikovat. Chaotická správa zdrojů, žádné podpůrné funkce, vše se dělalo pevně na míru a každá změna byla velmi obtížná. Tehdy jsem pochopil co je to engine a k čemu je dobrý. V průběhu roku 2002 začal vývoj Eclipse Enginu. Brzy se objevil programovatelný grafický hardware, jehož možnosti a dopad na počítačovou grafiku mě hned uchvátil. Začal jsem mít pocit, že dokážu vytvořit stejné grafické efekty jako profesionální vývojáři her. Stala se z toho taková menší posedlost. V průběhu dalších let jsem neustále předělával svůj engine, abych dokázal to, co profesionálové. Experimentoval jsem s řadou efektů. Například dynamické osvětlení se stíny, generování a kreslení terénu včetně vegetace, kreslení realistické vodní hladiny,

Obrázek 6: Flying for Dummies. Ilustrační obrázky.

vytvořil jsem velmi flexibilní skriptovatelný částicový systém, pracoval jsem s volumetrickými světly a stíny, zkoušel jsem dokonce napsat vlastní fyzikální engine (fungoval, ale nedisponoval příliš velkou stabilitou), experimentoval jsem s nefotorealistickým zobrazením (Non-Photorealistic Rendering - NPR) a v neposlední řadě jsem zkoušel různé metody post-processingu. Během těchto experimentů se dosti chaoticky měnil i samotný Eclipse Engine. V roce 2007 jsem došel k názoru, že tato situace je už neudržitelná a začal jsem pomalu vyvíjet Eclipse Engine znovu od začátku. Tentokrát čistě a se spoustou zkušeností z předchozích projektů. Ani tak se engine nevyhnul řadě dílčích úprav, ale ty již nikdy nebyly příliš zásadního charakteru. Zajímavostí je, že engine nebyl původně navržen s podporou fyziky. Podpora fyziky byla přidána až v relativně pozdní fázi. Tady mi pomohli zkušenosti z jednoho školního projektu, kde jsem se podílel na vývoji počítačové hry Flying for Dummies (viz obr. 6). Ta disponovala komiksovou grafikou a byla postavená právě na interakci hráče s fyzikou řízeným světem. Právě při vývoji této hry jsem seznámil s fungováním fyzikálního enginu, a tyto zkušenosti jsem velmi brzy uplatnil i v Eclipse Enginu.

6.2 Navržené moduly

Cílem bylo vybrat takové související části, které by se dali dohromady použít k vytvoření funkční hry, a zároveň bylo možné tyto části v daném časovém horizontu implementovat. V první řadě jsem se rozhodl omezit se jenom na jednu platformu. Konkrétně jde o osobní počítač s operačním systémem Windows. Co všechno tedy EE obsahuje? Postupně projdu všechny vrstvy tak, jak byly popsány v základní architektuře herního enginu.

- Cílový hardware. EE běží pouze na osobních počítačích.
- Ovladače. Ovladače od poskytovatele hardwaru nebo od poskytovatele operačního systému.
- Operační systém. Microsoft Windows XP a vyšší.
- SDK třetích stran. EE používá několik SDK. Konkrétně Win32 API, DirectX 9, DirectX 10, a C++ STL. Tyto knihovny jsou nedílnou součástí enginu. Dále se ještě používá NVidia PhysX, ta není ale pevnou součástí enginu. Fyzikální subsystém je plně modulární, přistupuje se k němu přes rozhranní, a je tedy možné ho kompletně vyměnit za jiný.

- Vrstva platformní nezávislosti. Jak bylo uvedeno, EE podporuje jenom jednu platformu, a tak zde tato vrstva zcela chybí.
- Podpůrné systémy. Tato část samozřejmě nemůže chybět. Omezení na jednu platformu tady umožnilo jednoduše využít řadu pomocných knihoven a nástrojů. Například EE spoléhá na správu paměti operačního systému, využívá datové struktury a algoritmy C++ STL, jako matematickou knihovnu využívá z velké části DirectX a na další funkce se používá Win32 API.
- Renderovací engine. EE obsahuje nízko-úrovňový renderer, správu scény, speciální efekty i jednoduchý front-end.
- Profilování a ladění. EE disponuje několika nástroji pro profilování a ladění včetně plné podpory NVidia PerfHUD.
- Detekce kolizí a fyzika. Tato vrstva je v EE také obsažena.
- Animace. Animační systém jsem se rozhodl neimplementovat, přestože byl původně plánován. Problémem je, že pro prezentování libovolné části enginu, jsou zapotřebí odpovídající data. Engine bez dat je k ničemu. A jedna z věcí, které nemám k dispozici a nedokážu si vyrobit sám, jsou právě animace. A protože kvalitní animační systém je velmi rozsáhlá a náročná část enginu, nechtělo se mi trávit spoustu času implementací něčeho, co nebudu moci ani otestovat, natož pak předvést ve hře. Z tohoto důvodu EE prozatím nepodporuje animace. Animace budou přesto v jedné z následujících kapitol rozebrány, když alespoň stručně popíši principy běžně používaných animačních metod.
- Vstupní zařízení. EE nativně podporuje klávesnici a myš s tím, že přidání podpory dalších zařízení by nemělo přestavovat problém.
- Audio. Audio systém je další subsystém, který jsem se rozhodl obětovat. Tentokrát čistě z důvodu nedostatku času.
- Online multiplayer. Multiplayer není nezbytná část pro tvorbu her, a tak se zařadil na seznam vyškrtnutých subsystémů. Navíc je asi velmi obtížné vyvíjet a ladit síťový kód jedním člověkem.
- Herní svět a základní objekty. EE má nativní podporu celé řady předpřipravených objektů. Od 3d modelů, přes světla a kamery až k předpřipraveným speciálním efektům.
- Systém událostí a zpráv. Není součástí EE, protože implementace systému by představovala časovou zátěž, a přitom se dá celý systém víceméně nahradit prostým voláním členských metod.
- Skriptovací systém. Je k dispozici Eclipse Scripting Language (ESL). Je to vysokoúrovňový, procedurální, skriptovací jazyk s automatickou správou paměti. ESL není pevnou součástí EE, a je tak možné použít libovolný skriptovací jazyk. Například velmi populární jazyk Lua.

- Základy AI. AI není součástí EE v žádné podobě.
- Nástroje. Společně s EE byly vyvinuty dva nástroje. Prvním je konvertor 3d modelů z formátu .x do formátu .emf, který používá EE. Druhým nástrojem je editor herního světa (World editor). Oba nástroje budou popsány později.

V následujících kapitolách rozeberu vybrané vrstvy/subsystémy trochu více podrobně. Chtěl bych se soustředit především na popis enginu z hlediska architekty a nezabíhat příliš do detailů implementace. V některých pasážích bude ale asi pro pochopení nezbytné zabývat se částečně i implementačními detaily.

6.3 Podpůrné systémy

V této kapitole se podívám na řešení některých důležitých částí jádra celého enginu.

6.3.1 Start, ukončení subsystému

Celý engine je velmi složitá aplikace skládající se z mnoha subsystémů. Tyto subsystémy nejsou tak úplně nezávislé. Při startu enginu je zapotřebí jednotlivé subsystémy inicializovat a nastavit. Inicializace se musí provádět ve správném pořadí, protože subsystémy jsou na sobě závislé. To samé platí i o závěrečném vypínání subsystémů. Vypnutí subsystémů zpravidla probíhá v opačném pořadí než jejich inicializace. Uvedu jednoduchý příklad, na kterém bude vidět, proč záleží na pořadí. Vezmu situaci s vypínáním enginu. Budu uvažovat dva subsystémy. Správce materiálů a správce textur. Jednotlivé materiály mohou obsahovat odkazy na použité textury. Nejdříve se tedy musí ukončit činnost správce materiálů, protože při uvolňování jednotlivých materiálů se budou uvolňovat i textury, a správce textur tak musí být ještě funkční. Typický způsob, jak implementovat subsystémy v herním enginu, je pomocí návrhového vzoru jedináček (Singleton). Singleton se nejsnadněji vytváří pomocí globálních nebo statických proměnných, které mimo jiné zajišťují globální přístup. Globální a statické proměnné se inicializují těsně před vstupním bodem programu (funkce main nebo WinMain ve Windows). Problém je, že inicializace probíhá v nepředvídatelném pořadí. Engine ale nezbytně nutně potřebuje ovládat pořadí. Existuje několik způsobů, jak navrhnout singleton tak, aby bylo možné kontrolovat, kdy bude vytvořen a zničen. V EE je použit následující.

```
cTextureManager* cTextureManager::ms_pThis = NULL;
class cTextureManager : public cCollectionManager
{
private:
   static cTextureManager *ms_pThis;
   ...
public:
```

```
    //Constructor
    cTextureManager()
    {
          //make sure manager isn't running already
          Assert(!ms_pThis,"Only one instance allowed");
          ms_pThis = this;

          //Start up
          ...
    }

    //Destructor
    ~cTextureManager() { ms_pThis = NULL; ... }

    //Global access to texture manager
    static cTextureManager& Instance() { return (*ms_pThis); }
    ...
};
```

Tento design umožňuje singleton dynamicky vytvářet i zničit. Zajištění správného pořadí inicializace subsystémů už je teď snadné. Jednotlivé objekty prostě vytváříme ve správném pořadí.

```
m_pTextureMgr = new cTextureManager;
m_pMaterialMgr = new cMaterialManager;
...
```

Při ukončování stačí jednoduše prohodit pořadí.

```
SAFE_DELETE(m_pMaterialMgr);
SAFE_DELETE(m_pTextureMgr);
...
```

Tento systém určování pořadí není určitě ten nejelegantnější. Dala by se například použít prioritní fronta. Ale tento napevno daný systém má několik výhod. Je jednoduchý a dá se snadno implementovat. Na první pohled je zřejmé pořadí subsystémů. Snadno se opravují chyby v pořadí – stačí jenom prohodit dvě řádky kódu. Navíc startovní procedura enginu je zpravidla součástí nějaké inicializační funkce celého enginu, která je stejná pro všechny aplikace používající engine, a není tak potřeba hledat nějaké elegantní řešení.

6.3.2 Správa paměti

Správa paměti je velmi důležitá část každého enginu. Jak už bylo řečeno, tak EE využívá správu paměti poskytovanou operačním systémem. Správce paměti ve Windows se prokázal být dostatečně efektivní na to, aby byl použit v profesionálních herních enginech, i když je vlastní správa paměti většinou lepší řešení, a u multi-platformních enginů dokonce jediné řešení. Vlastního správce enginu jsem tedy nenavrhoval. Alespoň zde velmi krátce vysvětlím základní princip fungování vlastního správce paměti, a také se zastavím u práce s dynamickou pamětí v herním enginu.

Detaily jednotlivých správců paměti v herních enginech se nepochybně od sebe liší, ale základní princip je stejný. Správce paměti si na začátku běhu aplikace vyžádá od operačního systému všechnu dostupnou paměť. Tím získá souvislý blok paměti, se kterým může zacházet dle libosti. Jakým kon-

krétním způsobem se paměť přiděluje a vrací, závisí na zvolené implementaci. Často se například používá metoda zásobníku, kdy je paměť jeden velký zásobník. Když se načítá nová úroveň, tak se pro ni alokuje paměť. Když se úroveň ukončí, tak se paměť zase vrátí. To je velmi efektivní metoda, protože se musí měnit jenom ukazatel na vrchol zásobníků. Nevýhodou je, že není možné paměť vracet v náhodném pořadí. Musí se vracet ve stejném pořadí, v jakém byla alokována. To se někdy řeší tak, že se vracení individuálních bloků paměti zakáže. Například u dealokace úrovně se jednoduše ukazatel na vrchol zásobníku vrátí na hodnotu, na které byl před alokací úrovně. Dalším běžným rozšířením správců ve formě zásobníku jsou správci, kteří umožňují alokace z obou stran zásobníků (viz obr 7). Z jedné strany se například alokuje současná úroveň a z druhé strany se alokuje paměť potřebná pro dočasné výpočty.

Obrázek 7: Oboustranný zásobník

Správci, kteří umožňují dealokace v náhodném pořadí, musí nějakým způsobem řešit fragmentaci paměti. Opět existuje celá řada metod, jak tento problém řešit. Detaily jsou popsané v řadě publikací a nebudu se tím zde zabývat.

EE využívá správce paměti operačního systému. To znamená, že alokace nové paměti může trvat velmi dlouho - OS hledá volné místo v paměti dostatečně velké na to, aby vyhovovalo požadované alokaci. Z toho plyne jedno pravidlo. Omezit alokace na haldě (heap) na minimum a **nikdy nealokovat paměť z haldy uprostřed herní smyčky** (viz 6.4.3). Veškeré alokace by měly proběhnout v inicializační části, před spuštěním herní smyčky. EE toto pravidlo plně dodržuje. Pokud například nějaký subsystém potřebuje periodicky vytvářet a ničit nějaké objekty, tak si alokuje během inicializace větší blok paměti, a ten si pak sám spravuje. Je to potom takový správce paměti v malém měřítku. Většinou si subsystém alokuje pole nějakých objektů, a za běhu potom hlídá, které prvky pole se aktuálně používají, a které ne. Alokace i dealokace jsou pak velmi rychlé. Stačí jenom vzít objekt ze seznamu volných objektů (alokace), nebo ho přidat do seznamu prázdných objektů (dealokace).

Žádný subsystém enginu uprostřed herní smyčky nebude alokovat novou paměť na haldě. Nicméně samozřejmě neexistuje způsob, jak tomu zabránit u aplikace, která engine používá.

Se správou paměti ještě částečně souvisí kontejnery a kolekce. Ty se v herních i neherních aplikacích hojně využívají. Mezi základní typy patří dynamické pole, spojové seznamy, binární stromy, slovníky atd. EE používá STL jazyka C++. V dnešní době už asi nemá moc smysl implementovat vlastní knihovny pro datové struktury a algoritmy nad nimi. Knihovny poskytované třetími stranami mají za sebou roky vývoje a optimalizací, a jsou proto velmi efektivní. Některé týmy pouze doplňují tyto knihovny o pro ně specifické funkce. V EE jsou použity některé vlastní datové struktury, ty ale nejsou rozšířením STL. Jsou samostatné a vyvinuté na míru jejich použití (viz níže). Před použitím knihovny

třetí strany je především důležitá znalost principů, na kterých jsou jednotlivými typy kontejnerů postaveny. Nevhodná volba kontejneru muže mít velmi výrazný dopad na efektivitu daného subsystému.

6.3.3 Souborový systém

Hry jsou multimediální systémy, které potřebují ke svému chodu řadu zdrojů různých typů. Správou těchto zdrojů se budu zabývat v další kapitole. V této kapitole popíši způsob, jak jsou tyto zdroje uloženy na disku, jak je engine vyhledává, a jak s nimi zachází. EE rozlišuje několik typů zdrojů. Konkrétně:

- Textury
- Fonty
- Modely
- Efekty
- Zvuky
- Materiály
- Skripty
- Scény

Ke každému typu zdroje existuje základní cesta, základní adresář. Engine poskytuje funkci pro nalezení zdroje. Do této funkce se pošle typ požadovaného zdroje a název souboru. Funkce prohledá základní adresář pro daný zdroj, a když nic nenajde, tak prohledá postupně i všechny podadresáře. Výsledkem je úplná cesta k hledanému zdroji. Nastavení základního adresáře pro jednotlivé zdroje se dá udělat jak přímo v programu, tak i editováním konfiguračního souboru enginu. Tento jednoduchý systém umožňuje uspořádat zdroje potřebné pro aplikace přehledně do adresářů a podadresářů. Při vývoji hry pak není potřebné znát přesnou cestu ke zdroji. Stačí pouze typ a název požadovaného zdroje.

Další výhodou je snadné přemístění na jiný počítač, do jiného umístění. Nezáleží na tom, kam byla aplikace nainstalovaná. Během instalace se mohou cesty nastavit na správné hodnoty přepsáním konfiguračního souboru, nebo je může aplikace nastavit při spuštění (většinou relativně vůči umístění spouštěného souboru).

Tato část enginu je také zodpovědná za přístup k souborům. EE ale využívá přímo standardní funkce C/C++ nebo Win32 API. Jediným rozšířením je pomocná třída pro syntaktickou analýzu textových souborů.

6.3.4 Správa zdrojů

Několikrát už bylo zmíněno, že hra potřebuje celou řadu různých zdrojů k tomu, aby mohla vykonávat svou činnost. Správa zdrojů v EE poskytuje následující funkce:

- Zajištění, aby byl jakýkoliv unikátní zdroj v paměti uložen vždy pouze jednou.
- Spravuje „životnost“ zdroje. Odstraňuje zdroj, když už není nikde používaný.
- Umožňuje načtení zdroje. Zajišťuje, že je zdroj na správném místě ve správné v paměti (systémová paměť, grafická paměť…).
- Zajišťuje načítání dalších zdrojů, pokud je zdroj potřebuje. Například při načítání materiálu se musí načíst ještě textury, které materiál potřebuje.
- Umožňuje ukládání (modifikovaných) zdrojů na disk. Umožňuje shlukování zdrojů do kolekcí. Kolekce se pak zapíše do jednoho souboru.
- Umožňuje znovu-načtení modifikovaného zdroje za běhu aplikace.
- Podporuje automatické znovu vytvoření zdrojů při ztrátě grafického zařízení (viz 6.4.6).

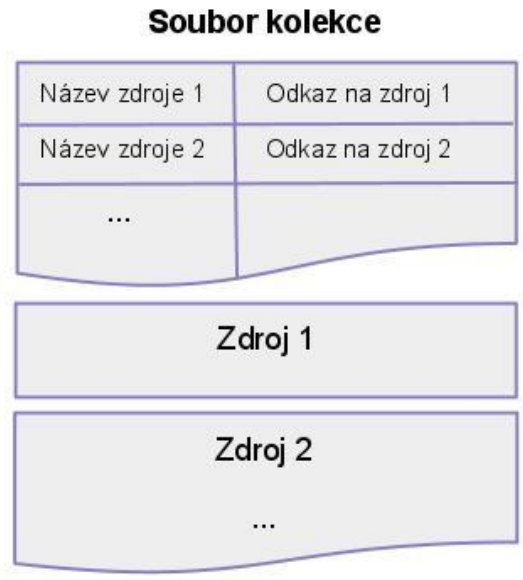
Jednotlivé funkce teď proberu trochu podrobněji. Předně je třeba uvést, že EE nepoužívá centralizovaný systém správy zdrojů. Pro každý typ zdroje existuje samostatný správce. Tito správci nicméně vycházejí ze společného předka stejně tak, jako všechny zdroje mají určitý společný základ.

Aby nedocházelo k duplikování zdrojů v paměti, využívá se systém počítání referencí. Každý zdroj si musí pamatovat, na kolika místech se používá. K tomu je nezbytné mít možnost zdroje jednoznačně identifikovat. V EE k tomu slouží název souboru, nebo název kolekce, ve které se zdroj nachází, plus název zdroje. Více o kolekcích se prozradím za chvíli. Když se poprvé načte zdroj do paměti, tak se mu počet referencí nastaví na jedna a odkaz na zdroj se vrátí volající funkci. Pokud poté přijde požadavek na ten samý zdroj, tak správce nejprve zkontroluje, jestli zdroj už není v paměti, a pokud ano, tak jenom zvedne o jednotku počet referencí a vrátí odkaz na již existující zdroj. Stejný mechanizmus se použije při vracení nepotřebných zdrojů. Když něco požádá o uvolnění zdroje, tak správce odečte jednotku od počtu referencí zdroje, a teprve když počet referencí klesne na nulu, tak to znamená, že zdroj už není nikde používaný a zdroj se uvolní z paměti.

Tento systém sice plní svůj úkol, nicméně není bez kazu. Vyžaduje zodpovědnost od uživatele enginu. Subsystémy enginu tento systém používají korektně, ale mimo engine se nedá nijak vynutit správné použití. Jde o to, že správce sice vrátí odkaz na zdroj a správně upraví počet referencí, ale poté už si se zdrojem může uživatel dělat, co chce. Když si udělá několik kopií odkazů na zdroj a bude je používat na různých místech, tak se sledovaný počet referencí nezmění a při uvolnění jednoho odkazu se zdroj uvolní a ostatní odkazy budou ukazovat na neplatnou paměť. Pro tyto případy manuálního kopírování odkazů na zdroje mají všechny zdroje funkci „AddRef“, která zvýší interní počítadlo

referencí. Je ale jenom na uživateli, aby tuto funkci použil, když kopíruje odkazy na zdroje, a také pak musí každý zdroj uvolnit. Při chybném používání může dojít k uvolnění zdroje, zatímco se ještě používá, nebo naopak se zdroj neuvolní, i když už není zapotřebí, a zabírá tak zbytečně místo. Správce nicméně před ukončením aplikace korektně uvolní všechny zdroje, které do té doby nebyly uvolněny. Systém počítání referencí zajišťuje první dva body na výše zmíněném seznamu.

K načítání a ukládání zdrojů není prakticky co říct. Jednotlivé zdroje mají své specifické formáty, které správce načte na správné místo do paměti. Všechny zdroje existují buď jako samostatný soubor, nebo jako část kolekce. Kolekce představuje soubor, který obsahuje několik zdrojů. Kolekce má několik výhod. Zvyšuje přehlednost adresáře se zdroji, když například všechny textury pro jeden objekt jsou v jednom souboru – běžný objekt bude ke svému vykreslení potřebovat tři, čtyři a více textur. Další výhodou je zrychlení načítání, pokud se načítá celá kolekce najednou. A další výhodou může být základní ochrana zdrojů před použitím bez svolení autora. Přece jenom je snazší vzít si texturu, pokud je ve formátu, který dokážu přečíst (bmp, jpg, tga…), než když je schovaná v nějakém neznámém formátu. Ochrana dat je ale opravdu jenom vedlejším efektem. Soubory kolekcí mají velmi jednoduchou strukturu a není nijak obtížné z nich data získat. Když už je řeč o vnitřní struktuře, tak ta je následující (viz obr. 8). Na začátku je nezbytná hlavička pro zjištění, jestli je soubor kolekcí, a jestli je soubor kolekcí požadovaného typu zdroje. Následuje seznam všech zdrojů ve formátu dvojic – název zdroje a jeho pozice uvnitř souboru. Tento seznam slouží k urychlení hledání jednotlivých zdrojů. Vytvářet soubory kolekcí umí každý správce. Zdroje se dají přeskupovat mezi kolekcemi i za běhu aplikace, čehož hojně využívá editor EE. Správce textur totiž nejenže vytvoří kolekci, ale i upraví všechny objekty, které tyto textury používají, aby příště používaly textury z nově vzniklé kolekce. Pro uživatele enginu navíc mezi zdroji ze samostatných souborů a zdroji z kolekcí není žádný rozdíl. Práce s oběma typy je zcela stejná. Pro identifikaci zdroje z kolekce se používá kombinace názvu kolekce a zdroje. Formát je následující: název kolekce/název zdroje. Například Objects.tcf/Table_normal.dds.



Obrázek 8: Soubor kelekce

Další velmi užitečnou funkcí správců zdrojů je schopnost znovunačtení zdroje za běhu aplikace. Tato funkce se opět využívá v editoru a značným způsobem může urychlit vývoj. V praxi to vypadá například takto. Při vývoji nějakého nového materiálu programátor edituje soubor efektu, který definuje podobu materiálu (viz 6.7.2). V editoru je objekt, na který se materiál aplikuje. Díky schopnosti

znovu načítat zdroje stačí soubor se efektem po editování uložit a editor ho okamžitě znovu načte a změny jsou okamžitě vidět. Bez nutnosti aplikaci vypínat. To samé platí pro textury, 3d modely a další zdroje. Celý proces znovu načtení je spolehlivý a velmi rychlý. Normálně by bylo potřeba najít všechny reference na daný objekt a ty nahradit novou referencí. To by bylo zdlouhavé a někdy i nemožné, protože uživatel enginu si může reference libovolně kopírovat, a engine tak nemá možnost je všechny dohledat. EE využívá toho, že se s žádným zdrojem nepracuje přímo. Všechny zdroje mají obalové třídy, které v sobě nesou vlastní zdroj, nebo informace o zdroji. Při znovunačtení se obalová třída neuvolní, takže všechny reference na ni mohou zůstat. Uvolní se pouze „vnitřnosti" obalové třídy, a ty se poté nahradí novým zdrojem. To je velmi elegantní metoda, bez negativních dopadů. Obalové třídy nebyly přidány kvůli znovu-načítání zdrojů. Obalové třídy poskytují řadu funkcí pro práci se zdroji, a navíc umožňují použít více verzí DirectX v jednom enginu (viz 6.7.1).

Poslední odpovědností správců zdrojů je obnovení zdrojů po ztrátě grafického zařízení. Tomuto problému se budu věnovat v samostatné kapitole.

6.4 Jádro a procesy

Existují dva základní způsoby jak navrhnout engine z pohledu jeho uživatele.

- Knihovna. První možností je forma knihovny. Celý engine je knihovna funkcí, ze kterých se sestaví celá aplikace. Tento postup nabízí maximální flexibilitu a kontrolu nad celým procesem. Nevýhodou je nutná dobrá znalost funkcí a jejich vztahů.
- Framework. Další možností je poskytovat engine ve formě je částečně hotové aplikace, do které programátor doplňuje části specifické pro jeho aplikaci (nebo mění původní funkcionalitu), neboli Framework. Přitom má ale omezené možnosti zasahovat do vnitřního fungování enginu/aplikace.

Eclipse engine byl navržen jako Framework. Pro použití EE je třeba odvodit své třídy od stávajících tříd a doplnit některé funkce, případně změnit některé části těchto tříd. Framework byl vybrán, protože výrazně urychluje vývoj nové aplikace za cenu obětování flexibility. Nejjednodušší aplikace postavená na EE bude zabírat par desítek řádek kódu a budou se v ní volat asi jenom čtyři funkce. Přesto bude tato mini aplikace plnohodnotná, se všemi subsystémy plně funkčními. Framework enginu se dá použít více způsoby. U řady věcí je možné si vybrat mezi řízením frameworkem, nebo vlastní implementací. V každém případě si každá aplikace musí vytvořit vlastní jádro a vlastní proces. Oba objekty jsou popsané v následujících kapitolách.

6.4.1 Jádro

Jádro reprezentuje třída cCore. Od této třídy je třeba odvodit vlastní třídu jádra a implementovat dvě virtuální funkce. Funkce OneTimeInit() se zavolá na závěr inicializace enginu. V této funkci se

minimálně musí vytvořit a registrovat jeden proces enginu (viz níže). Druhou funkcí je funkce Shutdown(). Tato funkce se zavolá při ukončování činnosti jádra a je to úplně poslední šance uvolnit případné zdroje před zničením jádra. Ve funkci Shutdown se nemusí udělat nic. Objekt jádra obaluje většinu podpůrných, nízko-úrovňových systémů. Konkrétně má na starosti následující věci:

- Spravuje (případně i vytvoří) okno, do kterého se vykresluje grafický výstup.
- Stará se o zachycení ztráty grafického zařízení a o následné zotavení (viz níže).
- Obsahuje deník aplikace.
- Spravuje procesy enginu.
- Spravuje systém cest k jednotlivým zdrojům.
- Vytváří a spravuje grafické zařízení.
- Stará se o standardní vstupní zařízení (klávesnici a myš).
- Má na starosti měření času aplikace (viz níže).
- Umožňuje načítat/ukládat grafické nastavení.
- Slouží přímo jako jednoduchý správce fontů.
- Spravuje manažery jednotlivých zdrojů.
- Stará se o herní smyčku.
- Obsahuje řadu pomocných funkcí jako je vyhledávání přesného umístění zdrojů, funkce pro prezentování scény, vykreslení trojúhelníku, nebo některé matematické funkce, které nejsou součástí žádné použité knihovny.

Mezi nejdůležitější funkce patří vytvoření/spuštění všech subsystémů. Od vytvoření správců zdrojů, přes inicializaci vstupního zařízení, až po inicializaci grafického subsystému. Například při tvorbě grafického zařízení se nejdříve pokusí načíst poslední použitou konfiguraci (ta se automaticky ukládá). Pokud žádná předchozí konfigurace není nalezena, tak se vytvoří výchozí nastavení. V obou případech se toto nastavení validuje, jestli je na daném hardwaru platné. Pokud není, tak se jádro pokusí vytvořit grafické zařízení s podobným platným nastavením.

S jádrem ještě souvisí třída cCoreObject. Je jednoduchá třída, která obsahuje statickou referenci na jádro. Většina objektů enginu je od této třídy odvozena, a má tak přes ni přístup k objektu jádra.

6.4.2 Proces

Jádro samo o sobě není nijak využitelné. Poskytuje sice přístup ke všem subsystémům, ale nedá se přes něj spustit žádný užitečný kód. A právě k tomu slouží proces. Interně třída cProcess. Proces enginu nemá nic společného s procesem operačního systému. Je to nástroj pro lepší strukturování rozsáhlé aplikace. Předně proces není nic jiného než rozhraní (třída s pouze virtuálními funkcemi). Proces představuje rozšíření jádra. Implementováním jednotlivých funkcí se vytvoří kompletní,

funkční aplikace. Vlastní implementace procesu se registruje v jádru a jádro ve správných okamžicích volá funkce procesu. Například funkce Frame je volána každý snímek a aplikace by v ní měla provést vykreslení snímku, reakci na vstup atd. Tyto funkce samozřejmě mohli být součástí jádra a žádné procesy by nebylo třeba zavádět. Důvod pro zavedení dalšího objektu je následující. Počítačové hry jsou složité aplikace, které se vedle vlastní hry skládají ještě z dalších částí, které očekávají jiné vstupy i výstupy. Například každá hra obsahuje přinejmenším menu a vlastní hru. V menu si uživatel může měnit nastavení, nahrávat uložené pozice, spouštět novou hru atd. Obě části budou vykreslovat jiný výstup, budou používat jiné zdroje a budou jiným způsobem interpretovat vstupní zařízení. Například klávesy šipek slouží k navigaci v menu i k pohybu postavy ve hře. Procesy jsou zde právě proto, aby umožnily tyto části logicky oddělit. Každý proces má vlastní data, vlastní funkce. Jeden proces může například reprezentovat herní menu, a druhý potom představuje vlastní hru. Mezi procesy se za běhu podle potřeby přepíná. Díky tomu není veškerý kód na jednom místě a celá aplikace se snáze spravuje. Navíc řada her obsahuje mnohem více částí. Klasickým příkladem mohou být různé mini-hry uvnitř her, které také vyžadují vlastní vstup a výstup.

Objekt jádra obsahuje manažer procesů s podpůrnými funkcemi. Každý nový proces se nejdříve musí registrovat, a poté už se může používat. Pro přepínání procesů se využívá zásobník. V jednu chvíli je aktivní vždy jen jeden proces. Nové (aktivní) procesy se přidávají na vrchol zásobníku, a pak se z něj zase odebírají. V tu chvíli se aktivuje proces, který byl těsně pod vrcholem zásobníku. Jinými slovy si manažer procesů pamatuje všechny procesy, které byly aktivní před přidáním současného procesu. Díky tomu se dají procesy použít například k implementaci hierarchického menu, kde každou stránka menu představuje jeden proces. Díky využití zásobníku je možné se postupně vracet přes předchozí stránky menu. Proces obsahuje následující virtuální funkce, které musí uživatel implementovat:

- OneTimeInit. Tato funkce se zavolá jednou při inicializaci herního enginu. Přesněji dojde k vyvolání této funkce po skončení funkce OneTimeInit objektu jádra (cCore). Ve funkci OneTimeInit by se měly provést jednorázové inicializace.
- RestoreDeviceObjects. Funkce RestoreDeviceObjects se zavolá po obnovení grafického zařízení (viz níže). Měla by se zde znovu inicializovat veškerá data, závislá na grafickém zařízení. Přesněji řečeno data, která jsou umístěna výhradně v grafické paměti, protože tato paměť byla přepsána jinou aplikací. EE dokáže většinu dat obnovit automaticky.
- Frame. Velmi důležitá funkce, která se zavolá každý snímek. Měl by se v ní vykreslit současný snímek, přečíst vstupní zařízení, spočítat fyzika atd.
- Shutdown. Shutdown je funkce, která se automaticky zavolá těsně před zničením procesu. Měly by se zde uvolnit veškeré zdroje.

- InvalidateDeviceObjects. Tato funkce se volá při ztrátě grafického zřízení. Je nutné uvolnit všechna data spojená s grafickým zařízením. Tedy data umístěná výhradně v grafické paměti. Stejně jako u obnovení těchto zdrojů je i jejich uvolnění z velké části automatické.
- MsgProc. Funkce na zpracování standardních zpráv systému Windows. Všechny okenní aplikace Windows spolu komunikují systémem zasílání zpráv. Implementací funkce MsgProc je možné zajistit vlastní reakce na tyto zprávy. Tuto funkci není třeba implementovat – existuje standardní implementace.
- OnEnter. Funkce, která se zavolá, když se proces stane aktivním.
- OnExit. Funkce, která se zavolá, když proces přestane být aktivním.

Proces a jádro spolu tedy vytváří ten dříve zmíněný Framework. Implementováním chybějících částí těchto tříd se vytvoří kompletní aplikace. Všechny mnohdy velmi komplikované postupy inicializace jednotlivých subsystémů jsou pro uživatele enginu skryty. Jednoduchá a funkční aplikace, která má k dispozici veškeré funkce EE se takto dá vytvořit během několika minut.

6.4.3 Herní smyčka

Každá hra, která chce dosáhnout iluze pohybu, musí generovat a zobrazovat rychle za sebou snímky. Stejně tak se i další subsystémy musí periodicky vykonávat. Jak bylo popsáno výše, tak funkce Frame aktivního procesu se volá jednou za snímek. Teď zbývá popsat konstrukci smyčky, která tuto funkci periodicky volá. Z principu potřebujeme nekonečnou smyčku, ve které se budou jednotlivé snímky vytvářet. Na konzolích může smyčka vypadat třeba následujícím způsobem.

```
//initialization
...

//loop
while(1);
{
   if(QuitButtonPressed())
         break;

   OneIterationOfGameLoop();
}
```

V případě operačního systému Windows je zapotřebí ještě zpracovávat zprávy zaslané aplikaci. Celá smyčka se tedy musí rozšířit o kontrolu příchodu zpráv a jejich zpracování. Pokud žádná nová zpráva není, tak se počítá herní snímek.

```
MSG msg;
msg.message = 0;
while(msg.message != WM_QUIT)
{
   if(PeekMessage(&msg,NULL,0,0,PM_REMOVE))
   {
         //MESSAGE HANDLING
         TranslateMessage(&msg);
         DispatchMessage(&msg);
   }
   else
   {
         OneIterationOfGameLoop();
   }
}
```

Tato smyčka už bude korektně zachytávat zprávy a ve „volném“ čase bude počítat vlastní hru. Funkce `OneIterarionOfGameLoop` z předchozího příkladu musí v EE ošetřit několik činností. Musí změřit čas od předchozího snímku. Musí hlídat, jestli se neztratilo grafické zařízení a případně se postarat o nápravu. A především musí zavolat funkci Frame právě aktivního procesu. Ještě je dobré zmínit, že jádro umožňuje buď využít předpřipravenou herní smyčku (funkce Run), nebo si programátor může smyčku vytvořit sám a periodicky volat funkci pro jednu iteraci (funkce Frame).

6.4.4 Měření času

Z předchozích kapitol musí být zřejmé, že simulace hry se provádí v diskrétních krocích. Simulovaná realita je ale spojitá. Pokud má být simulace věrohodná, musí být časové rozestupy mezi jednotlivými snímky velmi malé a je třeba znát naprosto přesně jejich délku. Engine musí umožňovat měření času s velkým rozlišením. Konkrétně EE má v jádru integrované dva časovače. MainTimer a GameTimer. MainTimer je hlavní časovač aplikace počítá celkový čas aplikace i množství času, který uplynul od posledního snímku. Všechny časy měří ve vysokém rozlišení a poskytuje časové informace pro všechny další subsystémy. GameTimer také měří celkový čas a čas od posledního snímku, nicméně k výpočtu používá hodnoty z MainTimeru. Proč jsou časovače tedy dva? MainTimer měří reálný čas aplikace, zatímco GameTimer měří čas hry. To je většinou to samé, ale začne se to lišit, když hra umožňuje nějaké manipulace s časem. Například GameTimer má přímo v sobě funkce pro nastavení jiné rychlosti plynutí času. A proč se tyto manipulace nedějí už v MainTimeru? Pravda je, že na některé subsystémy potřebujete reálný čas ať už je ten herní jakýkoliv. Uvedu příklad. Řada her umožňuje na krátký okamžik zpomalit čas. Protivníci, nebo herní prostředí se pohybuje zpomaleně a hráč má výhodu. V tomto případě je zapotřebí, aby se herní svět řídil zpomaleným časem, ale stále musí být zajištěno, aby například ovládání hlavní postavy zůstalo nezměněné, a na to je třeba reálný čas. Ve skutečnosti ve hře muže existovat celá řada časovačů pro různé účely. Takže je nezbytné mít k dispozici reálný čas aplikace. Z tohoto důvodu obsahuje jádro enginu dva odlišné časovače.

6.4.5 Pevný framerate

MainTimer má ještě jednu vlastnost. Umožňuje udržovat stejné rozestupy mezi jednotlivými snímky. Jednoduše se mu nastaví požadovaná délka jednoho snímku a on ji udržuje. Nutno poznamenat, že dokáže pouze prodloužit krátké snímky. Urychlit ty pomalé samozřejmě nedokáže. A k čemu je to dobré? Například fyzikální simulace je zpravidla stabilnější, pokud jsou mezi jednotlivými kroky simulace stejné časové rozestupy.

Poslední funkcí MainTimeru je eliminace příliš dlouhých snímků. Jde o to, že z různých důvodů může dojít k neúnosnému prodloužení rozestupu mezi dvěma snímky. Například nějaký jiný proces operačního systému krátkodobě vytíží systém natolik, že spočítání snímku trvá velmi dlouho. Dalším typickým případem je spuštění hry v debuggeru během vývoje. Běh programu se zde často přeruší. Čas nicméně běží dál a výsledný časový rozestup od posledního snímku může být v řádu minut až hodin! Počítat následující krok simulace s rozestupem několika minut od posledního není dobrý nápad. Přinejmenším simulace fyziky v takovou chvíli pravděpodobně přestane být stabilní. Pro tyto případy MainTimer omezuje maximální délku jednoho snímku. Pokud je tato hodnota překročena, tak MainTimer natvrdo nastaví časový odstup od posledního snímku na maximální délku snímku. Díky tomu je simulace stabilní a plynulá i při ladění v debuggeru.

6.4.6 Ztráta grafického zařízení

Ztráta grafického zařízení je problém postihující aplikace běžící nad DirectX 9. Ztráta grafického zařízení se projeví jako tiché selhání veškerých grafických operací DirectX. V DirectX nejsou specifikovány všechny možné příčiny ztráty grafického zařízení, ale mezi časté příčiny patří přepnutí se do jiné aplikace z celoobrazovkové aplikace, nebo spuštění systémového dialogu Windows (CTRL + ALT + DELETE). Při ztrátě grafického zařízení nemůže aplikace nic vykreslovat a musí čekat, dokud není možné zařízení obnovit. Před obnovením je třeba uvolnit veškeré zdroje v grafické paměti. Po obnovení grafického zařízení se musí tyto uvolněné zdroje znovu vytvořit. EE dokáže výše popsaný postup vykonat téměř automaticky. EE sleduje většinu zdrojů a v případě potřeby je dokáže uvolnit, a poté zase znovu vytvořit. Stejnou proceduru je třeba vykonat i při změně velikosti okna, do kterého se vykresluje. Grafické zařízení se musí resetovat s novým nastavením. Během resetování se musí také uvolnit veškerá grafická paměť. Po resetu se uvolněné objekty musí opět vytvořit. Obě, vlastně stejné operace, obstarává funkce jádra Reset. V prvním případě je volána automaticky a v druhém ji musí zavolat uživatel enginu při změně velikosti okna. Ještě ke změně velikosti okna. Speciální textury, do kterých aplikace dokáže kreslit (Render Target), umožňuje správce textur vytvářet buď s pevnou velikostí, nebo jako procento velikosti okna aplikace. Při změně velikosti okna se potom render targety vytvoří ve správném poměru k nové velikosti.

6.5 Vstupní zařízení

Subsystém vstupního zařízení jsem původně plánoval navrhnout tak, aby přes společné rozhranní bylo možné komunikovat s libovolným vstupním zařízením. To se na první pohled zdálo jednoduché. Myslel jsem, že DirectInput (část DirectX starající se o vstupní zařízení) dokáže komunikovat s libovolným zařízením. Podobně jako DirectX dokáže komunikovat s libovolnou grafickou kartou. To se ale neukázalo jako pravdivé. V dřívější verzi enginu založené na tomto chybném předpokladu bylo možné použít jenom vstupní zařízení podporované DirectInput. Funkce, ve které se reaguje na vstup, byla součástí procesu enginu (viz 6.4.2). Čtení stavu vstupního zařízení probíhalo automaticky v herní smyčce a uživatel jenom implementoval funkci, ve které se reaguje na vstup. Potom jsem ale zjistil, že některá herní zařízení DirectInput nepodporují. Dokonce i Xbox 360 Controller od Microsoftu používá vlastní ovladače (XInput). Rozhodl jsem se subsystém vstupních zařízení předělat. Relativně rychle jsem se rozloučil s myšlenkou společného rozhranní pro všechny zařízení. Rozdíly mezi jednotlivými zařízeními jsou příliš velké. Některá zařízení podporují ukládání změn svého stavu pro pozdější procházení a některá dovolují číst jenom aktuální stav. Způsob, jakým reprezentují své ovládací prvky, se liší. Rozhodl jsem se proto pro následující řešení. Čtení a zpracování vstupu není součástí herní smyčky. Pokud chce hra používat nějaké vstupní zařízení, musí si vytvořit vlastní třídu/funkce na manipulaci s tímto zařízením. Pomocí těchto funkcí musí každý snímek získat vstup ze zařízení a podle potřeby na něj reagovat. Tento systém zrovna neulehčuje přidání podpory dalšího zařízení, ale na druhou stranu umožňuje použít zcela libovolné zařízení. Navíc většina zařízení má k dispozici knihovnu funkcí, díky které je přidání podpory do enginu velmi snadné.

Přímo v EE je podpora pouze dvou vstupních zařízení. Myši a klávesnice. Pro obě zařízení existují třídy, které poskytují funkce pro práci s nimi. Myš i klávesnice je přímo součástí jádra enginu. Při startu enginu se obě zařízení inicializují a jsou připravena k použití. Pro přístup k nim obsahuje jádro funkce GetMouse a GetKeyboard. Zařízení podporují oba způsoby čtení stavu. Získání aktuálního stavu (immediate) i procházení předchozích změn stavu (buffered). Stejně jako u jiných vstupních zařízení musejí být funkce na čtení myši a klávesnice volány manuálně jednou za snímek. Kromě získání stavu je možné u obou zařízení nastavit, jestli k nim má aplikace exkluzivní přístup, a jestli má aplikace získávat data ze zařízení, i když aplikace není v popředí (cooperative level).

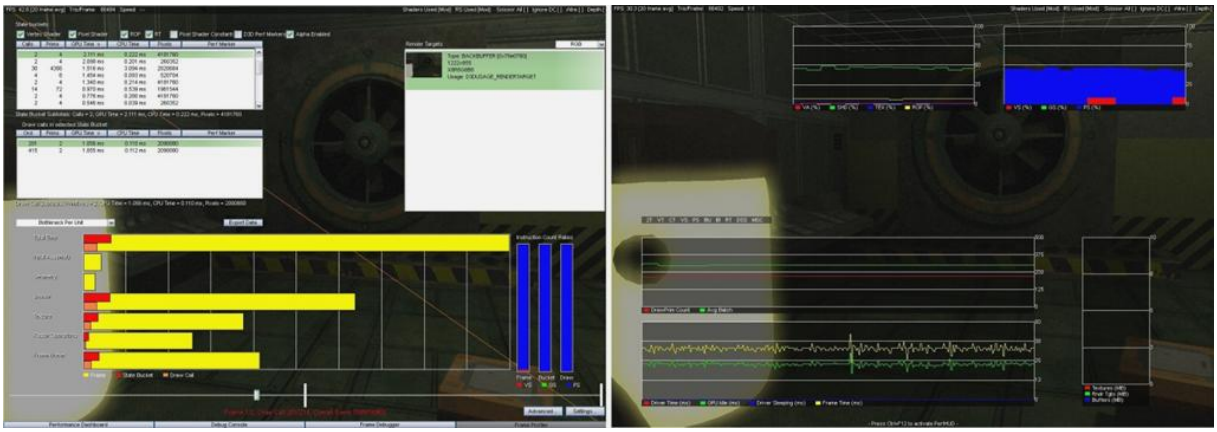
6.6 Ladící nástroje

Počítačové hry jsou nepochybně netriviální aplikace, a při jejich vývoji se proto pro jejich vyladění využívají různé nástroje. Tyto nástroje se dají rozdělit do dvou kategorií. Nástroje pomáhající odhalit a odstranit místa, která brzdí běh celé aplikace (bottleneck). A nástroje na odstraňování chyb. Existuje celá řada univerzálních aplikací na měření výkonu. My se tady ale zaměříme na nástroje přímo vyvinuté pro EE a na PerfHUD, pro který byl EE přizpůsoben.

6.6.1 PerfHUD

NVidia PerfHUD je velmi výkonný nástroj na analyzování grafických aplikací v reálném čase. PerfHUD dokáže sledovat a analyzovat obrovské množství informací o grafické části aplikace. Sleduje jak čas strávený při různých grafických operacích, tak i paměťovou náročnost. Měří data globální i na úrovni jednoho snímku. Umožňuje procházet vykreslování snímku krok za krokem. Veškeré kroky jsou přímo vidět. PerfHUD funguje jako GUI nad běžící aplikací. Změny v PerfHUDu se projeví na vykreslované scéně. Dále vykresluje různé plně konfigurovatelné grafy o výkonu. Kompletní popis možností tohoto mocného nástroje by vydal na celou knihu. Zájemce o detaily odkazuji na webové stránky společnosti NVidia.

Z hlediska enginu je nutné jenom poznamenat, že PerfHUD vyžaduje speciálním způsobem vytvořit grafické zařízení, a pro jeho použití je potřeba modifikovat zdrojové kódy aplikace. EE proto podporuje makro preprocesoru ENABLE_PERFHUD. Pokud je toto makro definováno, tak se vytvoří grafické zařízení s podporou PerfHUD. Ukázka editoru Eclipse enginu se zapnutým PerfHUD je na následujícím obrázku.



Obrázek 9: EclipseEd a PerfHUD

6.6.2 Profiler

Další nástroj pro měření výkonu je profiler EE (třída cProfiler). Tento nástroj umožňuje uzavírat části kódu do měřených úseků. Tyto úseky je možné vnořovat do sebe. Profiler měří časy v úsecích a zobrazuje výsledky na obrazovku.

6.6.3 Debug Draw

DebugDraw slouží jako náhrada oblíbeného printf (nebo nějaké jiné varianty vypisování textu). Jednou z metod, jak hledat chyby, je vypisovat v reálném čase hodnoty různých proměnných. To je ale poněkud problematické u 3d grafických aplikací. Například se objevila chyba, kdy projektil nezasáhl svůj cíl, i když by správně měl. Je možné zkoumat řadu čísel symbolizující měnící se pozici projek-

tilu a představovat si, kde se projektil nachází, to je ale pro lidský mozek velmi obtížné, až nemožné. Nebylo by lepší trajektorii projektilu nakreslit pomocí jednoduché čáry? Počítačové hry jsou kompletně ovládány matematikou. Jedna matice sice obsahuje veškeré informace o pozici, orientaci a velikosti objektu, ale člověk to z těch 16-ti čísel nevyčte. Proto vzniklo DebugDraw. DebugDraw umožňuje vykreslování jednoduchých geometrických obrazců složených z úseček. Tyto objekty prezentují srozumitelnou formou informace o 2d/3d světě. Aby bylo DebugDraw skutečně užitečné, musí umožňovat několik věcí.

- Musí být dostupné z libovolné části kódu
- Musí umožňovat vykreslování objektu po určitou dobu, protože informace, která se vykreslí pouze v jednom snímku, je nepostřehnutelná.

První bod není až tak triviální, jak se může zdát. DirectX (stejně jako většina grafických SDK) umožňuje kreslit grafiku jenom ve speciálně ohraničené části kódu. Aby bylo možné používat DirectDraw kdekoliv, musí si DebugDraw pamatovat objekty, které má vykreslit, a potom je vykreslit ve chvíli, kdy to DirectX povoluje. DebugDraw si proto vytváří kopii každého požadavku. Díky tomu je možné použít DebugDraw z jakékoliv části kódu. Když už DebugDraw uchovává všechny objekty, tak druhý bod nepředstavuje žádný problém. S každým objektem si uloží dobu, po kterou se má objekt kreslit. Objekt potom vykresluje každý snímek až do uplynutí časového limitu. Když čas objektu klesne na nulu, je objekt smazán.

V současné implementaci podporuje DebugDraw následující typy objektů:

- Úsečka. Parametry jsou koncové body a barva.
- Kvádr. Parametry jsou pozice, orientace, velikost a barva.
- Koule. Parametry jsou pozice, rádius a barva.
- Kapsle. Parametry jsou pozice, rádius, výška, orientace a barva.
- 2d text. Parametry jsou text, pozice v obrazových bodech a barva.
- 3d text. 2d text umístěný ve 3d prostoru. Parametry jsou text, 3d pozice a barva.

DebugDraw je velmi užitečný nástroj pro vizualizaci ve 3d prostoru.

6.6.4 Application Log

Posledním nástrojem na hledání chyb je deník aplikace (Application Log). Je to speciálně vyhrazený textový soubor, do kterého se zapisují události spojené s aplikací. K deníku se přistupuje přes jádro enginu, které má několik funkcí pro zapisování dat. Přes objekt jádra se může do deníku cokoliv zapisovat. Veškeré zápisy jsou také přeposlány do konzole Visual Studia. Deník se takto dá použít stejně jako printf v klasické aplikaci. EE sám deník využívá. Při startu enginu se do deníku zapíší informace o průběhu inicializace a v případě chyby se sem zapíší chybová hlášení. Deník pak vypadá například takto:

```
eclipse engine, (2010)
Started on _ 17/04/2010 | 00:47:33

[INIT]

Core: D3D Init... OK
Input: Input initialization... OK
Input: Init keyboard... OK
Input: Init mouse... OK
Graphics: Display Settings:
 - Device Type = HAL
 - Windowed = 1
 - Use Z-Buffer = 1
 - Use Stencil Buffer = 1
 - Back Buffer Count = 2
 - Refresh Rate = 60Hz
 - Back Buffer Format = X8R8G8B8_UNORM
 - Depth Stencil Format = D24_UNORM_S8_UINT
 - Vertex Processing = Software
 - Width = 1222
 - Height = 825

Graphics: Device creation... OK
Physics: Initialization... OK

[RUN]

Error: cTextureManager::LoadTextureFromFile(Ex)(), unable to load
texture: "checker.dds"

[SHUTDOWN]

(0:4:46) - exit ...
```

Deník využívají všechny subsystémy k hlášení chyb. Popravdě řečeno většina funkcí, které mohou selhat, nevrací žádné konkrétní informace o chybě. Ohlásí pouze, že selhala a chybu vypíše do deníku. To výrazně zjednodušuje návrh jednotlivých funkcí, není třeba vymýšlet a ošetřovat různé chybové stavy. To je obzvláště užitečné u vnořených funkcí, kdy je třeba chybový stav přeposílat do funkce, která je schopna na chybu reagovat. Binární chyby (Ok/Chyba) tento proces velmi zjednodušují. Samozřejmě existuje velmi elegantní způsob zachytávání chyb. Jsou to výjimky. Těm jsem se ale rozhodl vyhnout z důvodu zvýšené výpočetní náročnosti aplikace, která výjimky používá.

6.7 Grafika

Grafický subsystém je asi nejrozsáhlejší část celého enginu. Má na starosti nízko-úrovňové operace i pokročilý management scény. Postupně budou probrány jednotlivé části grafického subsystému.

6.7.1 Vrstva nezávislosti na DirectX

Jak už bylo několikrát zmíněno, EE podporuje DirectX 9 i DirectX 10. Pro podporu obou verzí jsem se rozhodl z následujícího důvodu. V době, kdy jsem začínal navrhovat EE, byly na trhu teprve první

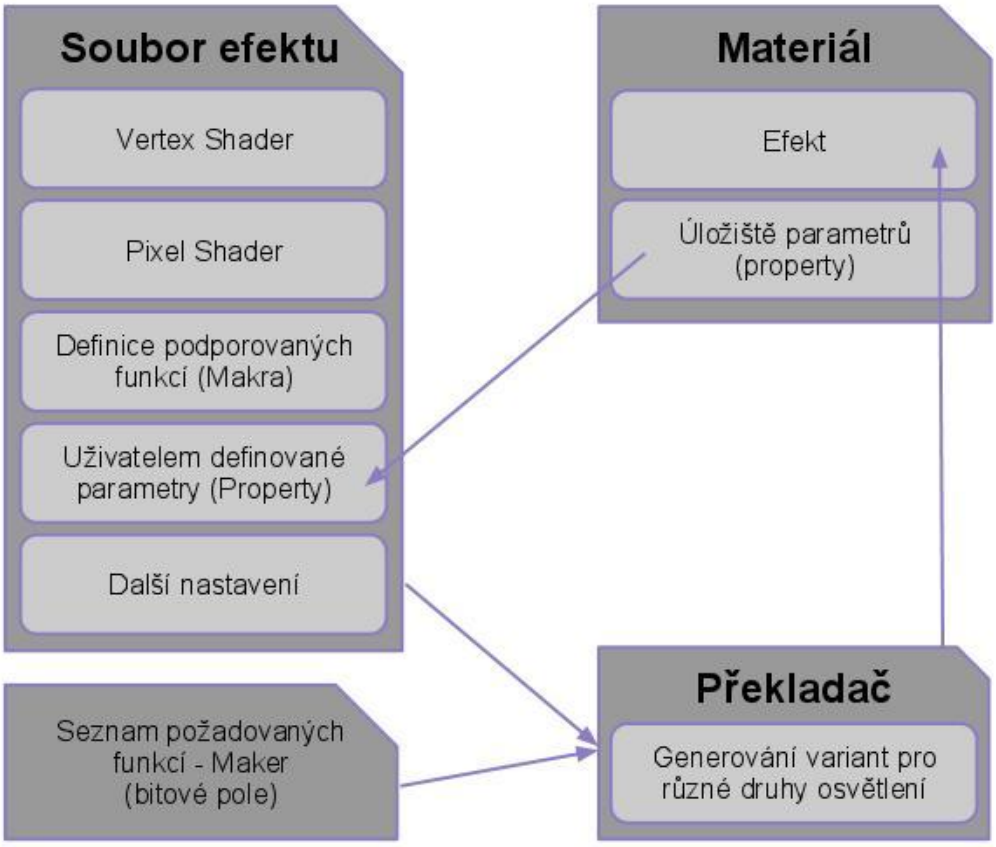
grafické karty podporující DirectX 10. A právě kvůli mizivému rozšíření DirectX 10 kompatibilního hardwaru bylo nezbytné zachovat podporu starší verze DirectX. Na druhou stranu novinky v nové verzi byly příliš lákavé, a tak EE podporuje obě verze. Aby bylo možné nějak smysluplně používat obě verze DirectX, je nutné vytvořit vrstvu, která smaže pro vyšší vrstvy rozdíly mezi verzemi DirectX. Je to stejný princip jako u vrstvy platformní nezávislosti z kapitoly 4.1.5 . Jenom v tomto případě není cílem sjednotit pro vyšší vrstvy rozdílné platformy, ale pouze rozdílné verze DirectX. Jak konkrétně tato nová vrstva vypadá. Je to jednoduché. Všechny objekty (třídy), které přímo manipulují s objekty DirectX, jsou implementované dvakrát. Jednou pro DirectX 9 a jednou pro DirectX 10. Součástí zdrojových kódů samozřejmě nemohou být obě verze stejné třídy naráz. Pro výběr jenom jedné implementace zdvojených objektů se používá preprocesor. Konkrétně k tomu slouží makro preprocesoru ECLIPSE_DIRECTX_10. Pokud je toto makro definováno, použijí se objekty postavené nad DirectX 10 a naopak. Z pohledu vyšších vrstev jsou obě verze objektů stejné. Poskytují stejné rozhranní. Konkrétní podoba tohoto rozhranní byla vytvořena tak, aby poskytovala takové funkce, které jsou uskutečnitelné v obou verzích DirectX. To nebylo vždy úplně jednoduché. Obě verze DirectX se v některých aspektech relativně dost liší. Poskytují sice podobné funkce, ale filozofie, s jakou pracují s jednotlivými objekty, je někdy značně odlišná. Rozhodl jsem se přizpůsobit EE pro novou verzi DirectX. Všechny objekty jsou tedy navrženy pro co možná nejefektivnější využití DirectX 10. U objektů, které jsou postavené nad DirectX 9, jsem potom musel dělat různé kompromisy, aby mohly komunikovat s rozhranním přizpůsobeném pro DirectX 10. Z toho důvodu nejsou všechny operace pod DirectX 9 prováděny zcela optimálně.

6.7.2 Materiály a efekty

V této podkapitole proberu velmi důležitou část grafického subsystému. Jsou jím materiály a efekty. Vykreslování geometrie je v EE možné výhradně pomocí efektů. V úvodu jenom krátce vyjasním použitou terminologii. Použité termíny vycházejí z DirectX. Shader je program běžící na grafické kartě. Tento program určuje výslednou podobu vykreslované geometrie. V DirectX 9 existují dva typy. Vertex Shader, který slouží ke zpracování bodů, ze kterých se skládá geometrie (vertex) a Pixel Shader, který slouží ke zpracování obrazových bodů (pixel). V DirectX 10 je ještě navíc Geometry Shader, který zpracovává celé trojúhelníky. Efektem bude v dalším textu myšlena kombinace vertex shaderu, pixel shaderu a případně dalších nastavení grafické karty. A nakonec materiál obsahuje efekt a konkrétní parametry, které efekt potřebuje. Parametrem může být například textura, která se aplikuje na povrch. Materiál je něco jako konkrétní instance efektu. Způsob zpracování geometrie se bere z efektu a může být pro různé materiály stejný. Vztah efektu a materiálu je nastíněn na obrázku 10, tento obrázek slouží i k ilustraci problémů v dalších dvou kapitolách. EE byl navržen tak, aby nepracoval s jednotlivými shadery, ale s celými efekty. Pro práci s efekty jsou v DirectX užitečné nástroje. Celý

efekt se dá jednoduše do enginu importovat ve formě textového souboru. Tento soubor muže kromě shaderů obsahovat i další nastavení. Konkrétní podoba souboru efektu je definovaná v DirectX. V efektech se používá programovací jazyk HLSL, který je velmi podobný jazyku C. Globálně definované proměnné se dají použít nejenom v shaderech, ale dá se k nim přistupovat i z enginu. V další kapitole vysvětlím, proč je to velmi důležité.

Efekty představují šikovný způsob, jak spojit shadery a další nastavení do jednoho objektu. Je to ale jenom první krok. Renderer EE musí řešit celou řadu problému spojenou s efekty. Postupně tyto problémy rozeberu.



Obrázek 10: Efekt a materiál

6.7.3 Materiál vs. okolní prostředí

V dobách před příchodem programovatelného hardwaru byly informace o materiálu a okolním prostředí striktně oddělené. Každý objekt měl přiřazený materiál, který definoval, jakým způsobem objekt reaguje na dopadající světlo. Informace o světle, které svítí na objekt, byly zcela oddělené a nezávislé. Materiál byl stejný bez ohledu na to, v jaké části scény se objekt nacházel, a jaká světla na něj svítila. To samé platilo pro další vlastnosti prostředí, jako je například mlha. Toto rozdělení je logické, protože objekt se může pohybovat v různých prostředích, a oddělit informace o prostředí od materiálu je velmi rozumné. Naneštěstí to není možné u programovatelného hardwaru, potažmo

efektů. Aby mohl shader spočítat výslednou podobu povrchu, musí znát veškeré informace o okolním prostředí. To není moc praktické. Jsou zapotřebí různé efekty pro stejný objekt v různých částech scény. EE se k problému staví následovně. Každý efekt existuje v několika variantách. Tyto varianty se generují automaticky interně. Každá varianta představuje určité světelné podmínky. Viz následující obrázek (obr. 11). Za běhu potom engine automaticky vybírá správnou variantu efektu podle toho, jaké světlo na objekt svítí. Efekty berou v potaz vždy jenom jeden světelný zdroj. Podle jeho typu a vlastností se nastaví správná varianta efektu. Pokud je objekt osvětlen více světly, musí se vykreslit

Obrázek 11: Variace efektu

vícekrát, pokaždé s jiným světlem a efektem. A jakým způsobem se z jednoho souboru efektu generují jednotlivé varianty? Využívá se zde maker preprocesoru, které soubory efektů podporují. Při překladu efektu engine definuje a přidává do souboru efektu různá makra, která určují konkrétní variantu efektu. Například makro ECLIPSE_SPOT_LIGHT se definuje, pokud varianta efektu reprezentuje materiál osvětlený bodovým světlem. Soubor efektu musí být napsán s využitím těchto maker. V souboru efektu tedy musí být něco jako toto:

```
#ifdef ECLIPSE_SPOT_LIGHT
…
#endif
```

Takto jsou v jednom souboru uloženy všechny varianty. Existují předpřipravené funkce, které se dají v souborech efektů použít, a které obstarají kompletní výpočet osvětlení bez ohledu na typ světla. Programátor efektu se tak může plně soustředit na naprogramování reakce materiálu na toto osvětlení. Tímto procesem se podařilo opět částečně oddělit osvětlení od definice materiálu. Uživatel enginu může v efektech definovat jenom reakci materiálu na světlo a pro výpočet osvětlení použít připravené funkce. Dále pracuje s každým efektem jako s jedním objektem, i když interně je každý efekt kolekcí efektů pro různé světelné podmínky. Výsledkem je systém, který umožňuje objektům se volně pohybovat po scéně, a vždy budou osvětleny správným způsobem. EE podporuje následující čtyři typy světel:

- Ambient light. Simulace nepřímého osvětlení.
- Directional light. Směrové světlo. Světlo nemá počátek, šíří se všude ve stejném směru.
- Point light. Bodové světlo. Vyzařuje světlo všemi směry z jednoho bodu.
- Spot light. Kužel světla. Světlo se šíří jenom v definovaném kuželu.

A dvě jejich vlastnosti:

- Projekce textury. Určuje, zda světlo promítá na své okolí nějakou texturu.
- Stín. Určuje, jestli světlo generuje stín – respektive, jestli na objekty dopadá stín generovaný ostatními objekty.

Typy světel a jejich vlastnosti vytvářejí další kombinace (celkově okolo deseti). Každá kombinace představuje jednu variantu efektu. Popravdě řečeno každé světlo má řadu dalších vlastností, ty jsou ale společné všem typům světel, a není tedy třeba vytvářet kvůli nim další varianty efektů. Takovou vlastností je například barva světla. Tolik k efektům a jejich vztahu k okolnímu prostředí.

6.7.4 Tvorba efektů

V předchozí kapitole jsem popsal tvorbu efektů z pohledu typu osvětlení. Bylo řečeno, že je možné použít předdefinované funkce, nebo si napsat efekt kompletně samostatně s využitím příkazů preprocesoru. V této kapitole představím způsob, jakým se v EE musí psát zbytek efektu, tedy reakce materiálu na spočítané osvětlení. Předně, každý efekt bude potřebovat k výpočtu řadu informací z enginu. Například se jedná o pozici kamery, transformaci objektu atd. Jak bylo řečeno, proměnné v efektech se dají nastavovat přímo z enginu. Tímto způsobem se do efektů před vykreslením objektu dostávají potřebná data. Aby engine poznal, co která proměnná znamená, využívá dodatečné informace o proměnných v efektu nazvané sémantika a anotace. Deklarace proměnné v souboru efektu vypadá třeba takto.

```
//camera position
float3 g_CameraPos : POSITION   //POSITION je semantika
<
   string Space = "World";   //anotace
   string Object = "Camera"; //anotace
>;
```

Engine si sémantiku a anotace proměnných přečte, aby identifikoval význam dané proměnné. Například z předchozí deklarace se dozví, že proměnná `g_CameraPos` reprezentuje pozici kamery ve světových souřadnicích (world space). Před vykreslením objektu s tímto efektem engine dosadí do této proměnné aktuální pozici kamery. EE rozeznává celou řadu proměnných. Všechny jsou nadefinované v předpřipraveném souboru `EclipseShared.fxh`. Tento soubor stačí vložit (příkaz preprocesoru #include je v efektech podporován) do svého souboru efektu, a rázem efekt získá přístup ke všem datům z enginu. Zde je seznam některých informací, které engine poskytuje efektům:

- Čas aplikace a doba trvání aktuálního snímku.

- Informace o kameře. Pozice, směr, dosah kamery (clip planes).
- Informace o světle. Pozice, směr, dosah, rychlost úbytku světla se vzdáleností, barvy světla, texturu, kterou světlo promítá na scénu a další.
- Transformační matice. Celá řada matic pro transformování objektů.

Všechny tyto informace sdílejí společnou paměť (Effect Pool). Díky tomu je stačí nastavit jednou a změní se ve všech efektech, to šetří čas a trochu i grafickou paměť.

Kromě těchto dat automaticky poskytovaných enginem, je možné v každém efektu definovat vlastní data – parametry (property). Jsou to opět proměnné efektu. Způsob jejich deklarace popíši za chvíli. Nejdřív vysvětlím, kdo je zodpovědný za dodání správných dat. V předchozí kapitole bylo uvedeno, že materiál je vlastně instance efektu. Vedle efektu obsahuje materiál i data, která efekt potřebuje. Tyto data se před vykreslením kopírují do parametrů efektu. Při tvorbě nového materiálu se určí efekt, ze kterého se materiál vytvoří. Materiál si přečte z efektu, jaké parametry podporuje. Pro tyto parametry si vytvoří lokální úložiště. Uživatel za běhu aplikace modifikuje přes funkce materiálu parametry efektu v tomto lokálním úložišti a před vykreslením engine překopíruje data do proměnných efektu (viz příklad níže). Vztah materiálu a efektu je znázorněn na obrázku 10. Tímto způsobem si může uživatel do efektu přidat libovolná data, která pak z aplikace mění. Typickým příkladem jsou textury použité při výpočtu. EE podporuje dva typy parametrů – číselná data a textury. Číselná data se deklarují v efektu například takto:

```
float SpecularPower : PROPERTY
<
    string Type = "Scalar";
    string Name = "Specular Power";
    float UIMin = 0.1;
    float UIMax = 128;
    string Desc = "Controls specular highlights intensity.";
> = 40;
```

Sémantika `PROPERTY` identifikuje uživatelem definovaný parametr. Anotace určují dodatečné informace. `Type` označuje typ proměnné, je podporována skalární hodnota, vektor, matice a „bool“ hodnota pro statické větvení programu na grafické kartě. `Name` je jméno parametru. Přes toto jméno je parametr dostupný. `UIMin` a `UIMax` určují minimální a maximální hodnoty parametru pro potřeby uživatelského rozhraní (např. editor materiálů). Poslední anotace je popis parametru. Opět slouží především pro potřeby editoru materiálů.

Nastavení výše popsaného parametru přes materiál může v enginu vypadat následujícím způsobem:

```
//create material
cMaterial *pMaterial = MaterialManager.CreateMaterial(...);
//get property index
INT iSpecularPower = pMaterial->GetPropertyIndex("Specular Power");
//
FLOAT fValue = 10.0f;
//set property value
pMaterial->SetProperty(iSpecularPower, &fValue);
```

Parametr typu textury se v efektu definuje následovně:

```
Texture2D NormalMap : TEXTURE
<
    string Name = "NormalMap";
    string Usage = "Normal map.";
>;
```

`Name` je opět název textury, přes který je textura dostupná. `Usage` je popis textury pro potřeby editoru materiálů.

S využitím dat poskytovaných enginem a vlastnoručně definovaých parametrů je možné dostat do efektu zcela libovolná data. Díky tomu se dá vytvořit v efektech prakticky cokoliv.

6.7.5 Supershader

Supershader (někdy také Ubershader) je termín pro shader, který v sobě obsahuje informace o více materiálech. Konkrétní shader se potom vyrobí pomocí příkazů preprocesoru, nebo nějaké podobné metody (skládání fragmentů). Postup nastíněný v kapitole 6.7.3 je svého druhu supershader – efekt obsahuje kód pro všechny typy osvětlení a konkrétní varianta se vybere pomocí preprocesoru. EE ale podporuje supershader i pro různé uživatelem definované typy materiálů v jednom souboru efektu. Nejdříve vysvětlím důvod tohoto rozhodnutí, a potom popíši, jak to celé funguje. Podpora supershaderu byla přidána proto, aby se snížila nutnost replikování stejného kódu uvnitř efektů. Vychází se zde z poznatku, že většina běžných materiálů má velkou část kódu stejnou. Například efekt, který definuje barvu povrchu, se od efektu bez barvy povrchu bude lišit v několika řádcích kódu a zbylé desítky nebo stovky řádek budou stejné. Popravdě řečeno většina běžných materiálů se dá simulovat kombinací několika málo vlastností (barva povrchu, barevná textura, textura odrazivosti světla, textura nerovnosti povrchu atd.). Nemá proto moc smysl psát desítky samostatných efektů, když budou mít většinu obsahu stejnou. Navíc změna nějaké společné části znamená editovat všechny soubory. To není zrovna praktické. Proto v EE existuje systém supershaderů. Supershader je souboru efektu, který využívá podmíněné příkazy preprocesoru. V souboru efektu je nutné provést dva kroky. Nejdříve se definují makra preprocesoru pro podmíněný překlad, a následně se tato makra použijí v těle efektu. Rozeberu to od konce, bude to snadněji pochopitelné. Část kódu efektu bude vypadat například takto:

Nejdříve definice nějaké proměnné.

```
//Diffuse color
#ifdef DIFFUSE_COLOR
    float4 DiffuseColor : PROPERTY
    <
        string Type = "Color";
        string Name = "Diffuse Color";
    > = float4(1,1,1,1);
#endif
```

A její použití někde uvnitř kódu.

```
...
//diffuse color
#ifdef DIFFUSE_COLOR
    vDiffuse *= DiffuseColor;
#endif
...
```

Definice proměnné a její použití se při překladu použije jenom tehdy, když bude v souboru definované makro DIFFUSE_COLOR. Musí se tam tedy někde vyskytovat příkaz:

```
#define DIFFUSE_COLOR
```

Tímto způsobem můžeme v jednom efektu mít řadu odlišných funkcí. Definování maker samozřejmě nemůže být v souboru napevno. Potřebujeme ho přidávat dynamicky před překladem efektu podle toho, jaké části efektu chceme při překladu použít. Makra vkládá do souboru před překladem automaticky engine podle požadavků uživatele. To funguje následovně. Aby mohl engine přidávat definice maker, musí vědět, jaké makra jsou v efektu použita. To načte před překladem z „hlavičky“ souboru efektu (o tom za okamžik). Engine má tedy seznam podporovaných maker. Uživatel se může enginu zeptat na počet maker a na jejich jména. Uživatel si vybere, jaké makra se mají při překladu použít, a engine podle toho efekt přeloží. Vybraná makra uživatel předá enginu ve formě bitového pole. Jednoduše platí, že pokud je první bit nastaven, tak se použije první makro, pokud je nastaven druhý bit, použije se druhé makro atd. Každý efekt je tedy identifikován souborem, ze kterého byl vytvořen a bitovým polem, které symbolizuje jaká makra, jaké části efektu byly při překladu použity (viz obr. 10). Konkrétně se používá 64bitů široké pole. Eclipse engine tedy podporuje u každého efektu až 64 maker.

Nyní nastal čas podívat se na způsob, jakým jsou v souboru efektu deklarována použitá makra. Výše uvedenému příkladu by odpovídala například tato deklarace:

```
string DiffuseColorMacro : Macro
<
    string TrueName = "DIFFUSE_COLOR";
    string TrueDefinition = "";
    string FalseName = "";
    string FalseDefinition = "";
> = "Diffuse Color";
```

Je to vlastně obyčejné deklarování proměnné v efektu. Stejné, jaké bylo k vidění v předchozích kapitolách. Deklaruje se proměnná DiffuseColorMacro a přiřadí se jí hodnota Diffuse Color. Pod názvem Diffuse Color bude toto makro známé v enginu. Obsahuje sémantiku Macro, aby engine rozpoznal, že se jedná o deklaraci makra. A dále obsahuje čtyři anotace. Ty určují podobu makra preprocesoru, které engine před překladem vloží do souboru. Konkrétně, pokud uživatel řekne, že se toto makro má použít, tak engine před překladem vloží následující příkaz:

```
#define TrueName TrueDefinition
```

Z předchozího příkladu to tedy bude:

```
#define DIFFUSE_COLOR
```

Pokud se makro nemá použít, engine vloží následující příkaz:

```
#define FalseName FalseDefinition
```

Z předchozího se nevloží nic, protože FalseName je prázdný řetězec.

Tyto deklarace maker musejí být na začátku souboru efektu. Jsou to pomocné informace pro engine a v efektu se nijak nepoužívají. Je možné si všimnout, že deklarace splňují pravidla pro zápis souboru efektu tak, jak jsou definována v DirectX. To znamená, že tyto deklarace nemusejí být odstraněny ze souboru před jeho překladem. Překladači efektů nebudou nijak vadit, i když se nikde nepoužívají.

Takto tedy fungují supershadery v EE. Celý proces ještě jednou shrnu. Uživatel chce vytvořit efekt z připraveného souboru. Pokud tento soubor ještě nebyl v aplikaci použit, tak engine načte úvodní informace o použitých makrech (funkcích) a uschová si je. Uživatel se může dotázat na jména jednotlivých maker, která slouží jako popis poskytované funkce. Uživatel si vybere jaké funkce (makra) se mají při překladu použít a předá je enginu ve forme bitového pole. Engine přidá definice maker preprocesoru do souboru a efekt přeloží. A jak bylo dříve řečeno, přeloží ho několikrát s dalšími internímy makry pro různé typy světel.

Zde ještě uvedu, že pokud aplikace běží pod DirectX 10, tak se do každého efektu vloží automaticky makro ECLIPSE_DIRECTX_10. Díky tomu je možné mít efekty pro obě verze DirectX v jednom souboru.

6.7.6 Kompilování efektů

Až do teď byla vždy řeč o efektech jako o textových souborech, které se za běhu aplikace přeloží. Překládání efektů za běhu má jednu podstatnou nevýhodu – časovou náročnost. Přeložení jednoho efektu je nepochybně velmi rychlé, ale problém je, že i jednoduchá scéna bude používat desítky až stovky různých efektů. Dnešní počítačové hry používají řádově tisíce efektů. Překlad takového množství efektů už se bude pohybovat v řádech minut. Navíc překládat efekty při každém spuštění má význam, jenom pokud se efekty mění. Tedy výhradně během vývoje aplikace. U finální aplikace je překlad efektů mrháním času. Z těchto důvodu podporuje EE překládání efektů do binárních souborů. Při spuštění se načte rovnou tato binární, přeložená verze efektu, což je nesrovnatelně rychlejší. Překládání efektů do binárního souboru ovšem není bez problémů. Situaci komplikuje fakt, že každý soubor efektu může nést několik různých variant efektu (viz výše). Řešení, které se nabízí, je přeložit

do souboru všechny možné varianty efektu a při načítání vybrat tu správnou variantu. Toto řešení naráží na jisté překážky. Pro EE byl napsán soubor efektu (StdMaterial.fx), který umožňuje vytvořit většinu běžných materiálů. Je to supershader s asi deseti makry. Jednoduchým výpočtem jsem zjistil, že z tohoto souboru je možné vygenerovat téměř 100 000 různých efektů (včetně variant pro různé osvětlení). Velké množství variant, ale nemá význam a nikdy se nepoužije. Navíc výsledný soubor by byl velmi rozsáhlý. Rozhodl jsem se raději pro jiné řešení. Binární soubor s přeloženými efekty se generuje postupně. Pokaždé, když se má přeložit nějaká varianta efektu, tak se zkontroluje, jestli již existuje její přeložená verze. Pokud existuje, jednoduše se načte, pokud neexistuje, tak se efekt přeloží a přeložená verze se uloží na konec binárního souboru. Textový a binární soubor mají stejné jméno, jenom jinou koncovku. Takto se postupně vytváří soubor s přeloženými variantami efektu. Jsou zde uložené jenom ty, které se skutečně používají.

6.7.7 Renderer

V této kapitole bude představen objekt scény v EE. Nejdříve vysvětlím princip renderování ve scéně. Poté přijdou na řadu nízko-úrovňové a vysokoúrovňové operace, které scéna poskytuje.

V první řadě objekt scény obsahuje seznamy všech světel, 3d modelů, kamer, render targetů a dalších objektů, ze kterých se scéna skládá, nebo které scéna potřebuje k vykreslení. Na tomto místě poznamenám, že scéna umí kreslit pouze objekty odvozené od třídy cRenderableObject. Tato třída obsahuje několik virtuálních funkcí, které je třeba implementovat. Například vrácení transformační matice, vrácení materiálu atd.

Kreslení scény probíhá v jednotlivých krocích. Zde nazvaných Scene Action (SA). Jsou definované následující typy SA:

- Geometry Pass. V tomto kroku se vykresluje geometrie – objekty ve scéně. Více o něm za chvíli.
- Clear. Clear smaže obsah současného render targetu/ů. Parametry jsou hodnoty, kterými se povrchy nově vyplní.
- Set Render Targets. Nastaví render target.
- Copy Surface. Umožňuje překopírovat jeden povrch (render target / textura) do druhého.
- Set Camera. Umožňuje změnit současnou kameru. Pro veškeré další vykreslování se použije nová kamera.
- Fullscreen Quad. Vykreslí obdélník přes celou obrazovku. Jako parametry se zadává materiál, který se aplikuje na kreslenou geometrii a volitelně cíl vykreslování (render target). Využití k postprocessingu. Všechny další SA by se dali realizovat pomocí výše zmíněných SA. Některé často používané akce byly přímo implementované jako další SA. Pro usnadnění používání EE.

- Downscale. Překopíruje zdrojovou texturu (render target) do cílové a aplikuje lineární filtr. Vhodné pro kopírování textury do menší textury. Podporuje tři varianty, a sice zmenšení na čtvrtinu (2x2 filtr kernel), na devítinu (3x3 filtr kernel) a na šestnáctinu (4x4 filtr kernel).
- Gaussian Blur. Provede na zdrojové textuře gaussovské rozostření. Je podporována řadu parametrů jako střední hodnota, odchylka nebo počet okolních vzorků (samples), které se použijí k výpočtu hodnoty každého pixelu.
- HDR. Vzorová implementace HDR. Obsahuje řadu parametrů, kterými se dá HDR nastavit.
- Volume Light. Simulace volumetrického efektu světla jako postprocesing.

Z těchto typů SA se poskládá postup vykreslování celé scény. Objekt scény během vykreslování postupně prochází jednotlivé akce tak, jak jsou za sebou definované. Posloupnost akcí se nazývá Scene Technique (ST) – technika scény. V objektu scény jich může být více. Přes rozhraní objektu scény je možné určovat, jaká technika se má kdy vykreslit (pokud vůbec).

Díky SA má uživatel enginu kontrolu nad procesem vykreslování. Uživatel kontroluje, kam se objekty vykreslují, kdy se vykreslují, z jaké kamery se vykreslují, a také má kompletní kontrolu nad post proces efekty. Díky technikám scény je možné připravit si různé postupy vykreslování. Například pro různé úrovně nastavení grafiky. Jedna technika třeba podporuje HDR a druhá je identická, ale bez vysokého dynamického kontrastu. Hráč hry si v nastavení vybere, jestli chce HDR a při vykreslování se použije patřičná technika.

6.7.8 Geometry Pass

Jak bylo naznačeno, Geometry Pass (GP) slouží k vykreslení geometrie (objektů) ve scéně. GP podporuje několik parametrů.

- Lighting (osvětlení). Pokud je osvětlení vypnuto. Vykreslí se všechny viditelné objekty ve scéně bez použití světelného zdroje. Vhodné například pro simulaci nepřímého osvětlení (ambient light), nebo pro vykreslení pouze hloubky objektů, aby při dalším kreslení nedocházelo k zbytečnému překreslování (Depth Only Pass). Pokud je osvětlení zapnuto, prochází se postupně všechna viditelná světla a pro každé světlo se vykreslí objekty, které jsou viditelné a tímto světlem osvětleny.
- Shadows (stíny). Pokud je osvětlení zapnuto a stíny jsou zapnuty, tak se pro jednotlivá světla generují automaticky ještě stíny. Povolit generování stínů musí také samotná světla. Podporovány jsou shadow mapy i stencil stíny. Konkrétní typ si volí objekty sami.
- Additive blending. Pokud je additive blending zapnuto, přidají se výsledky GP k tomu, co už bylo vykresleno dříve. Jinak předchozí hodnoty přepíší.
- Force Light. Umožňuje vnutit konkrétní světlo. Pokud je osvětlení zapnuto, tak se nebudou procházet všechny viditelné světla, ale použije se jenom toto vnucené.

- Force Material. Umožňuje vnutit všem vykreslovaným objektům konkrétní materiál.
- Force Top LOD. Umožňuje vnutit hodnotu nejvyššího detailu 3d modelů. To znamená, že žádný objekt nebude mít větší úroveň detailů, než je zadaná hodnota. Například při kreslení odrazu vodní hladiny není třeba kreslit objekty s plnými detaily.
- Index. Každý GP má přiřazený číselný index. Ten se dá libovolně měnit a více GP může mít index stejný. Slouží k přiřazení objektů ke GP (viz další kapitola).

6.7.9 Render Template

GP vykresluje veškeré viditelné objekty. To nemusí vždy vyhovovat požadavkům. Například je potřeba v jednom GP vykreslit neprůhledné objekty a v dalším průhledné. K získání kontroly nad tím, který objekt se kdy vykresluje, slouží Render Template (RT). Každý objekt má přiřazený RT. RT je pouze seznam indexů GP, ve kterých se objekty mají vykreslit. RT například říká, že se má použít v GP s indexy nula a jedna. Všechny objekty s tímto RT se vykreslí ve všech GP s indexem nula nebo jedna.

6.7.10 Nízko-úrovňový renderer

Tento renderer se použije v GP scény. Přichází do něj seznam objektů a on je musí rychle vykreslit. O vlastní vykreslení se postarají objekty nad DirectX. Renderer musí objekty před vykreslením setřídit. Prohazování textur a efektů je časově relativně náročné, a proto se mu snažíme vyhnout. Nebudu zde zabíhat do přílišných detailů, ale v zásadě se děje následující. Renderer získává z objektů materiály, které se mají použít na jejich vykreslení. Z těchto materiálů se vytváří binární strom. Materiály se stejným efektem se řadí do jednoho prvku stromu. U každého materiálu se vytváří seznam objektů, které se tímto materiálem vykreslují. Objekty u jednotlivých materiálů by se dali ještě třídit podle vzdálenosti od kamery, ale současná implementace to nedělá. Procházením tohoto stromu se zaručí, že množství změn efektů a materiálů je minimální.

6.7.11 Správce scény

Tato část je zodpovědná za určování, které objekty jsou vidět, aby se neplýtvalo výkonem na neviditelné objekty. Způsob zpracování závisí zcela na programátorovi. Objekt scény (třída cScene) obsahuje několik virtuálních funkcí. V aplikaci se musí od třídy cScene odvodit vlastní třída, která implementuje chybějící funkce. Náplní těchto funkcí je určit viditelné objekty na základě předaných parametrů. Objekt scény například přes tyto virtuální funkce požádá o objekty viditelné z určité kamery, nebo osvětlené určitým světlem. Jaký algoritmus a jaké datové struktury se k nalezení těchto objektů použijí, je čistě na programátorovi. Uživatel enginu si může vytvořit libovolnou reprezentaci scény. Libovolný algoritmus zjišťování viditelnosti. V EE existuje vzorová implementace, která používá systém portálů a místností.

6.7.12 Optimalizace

EE využívá různé optimalizace na všech úrovních enginu. Na nižších úrovních se například hlídá, aby se textura nevyměnila za tu samou. V takovém případě se žádná operace neprovede. V efektech se sdílejí proměnné. Objekty se před vykreslením třídí. Engine podporuje Geometry Level of Detail (GLOD) i Material Level of Detail (MLOD). GLOD – 3d modely existují v několika variantách, s různým množstvím detailů. Objekty dále od kamery se kreslí s menším množstvím detailů. MLOD – Každý materiál existuje v několika variantách, různě detailních variantách. Podle vzdálenosti od kamery se vybere úroveň detailů. Na vyšších úrovních se především řeší, které objekty jsou viditelné, která světla osvětlují viditelnou část scény atd. Plus řada interních algoritmů a datových struktur byla vyvinuta nebo upravena pro konkrétní potřeby. Například stavba binárního stromu v nízko-úrovňovém rederu je upravená tak, aby byla operace vkládání pro mnohé objekty zvládnutá v konstantním čase.

I přes snahu dosáhnout velké efektivity vykreslování zůstává faktem, většina věcí se v samotném enginu ovlivnit nedá. A výsledný výkon do značné míry závisí na schopnostech a znalostech uživatele enginu.

6.7.13 Shrnutí

Tímto končí popis grafického subsystému. Zaměřil jsem se v něm především na efekty, materiály a vlastní scénu. To jsou pro grafiku nesmírně důležité části, ale rozhodně to není celý grafický subsystém. Ten obsahuje desítky dalších tříd, které se starají o textury, fonty, 3d modely, světla a mnoho dalšího. Jejich popis by ale byl příliš rozsáhlí, a protože to jsou do určité míry pouze třídy, které obalují a zjednodušují práci s objekty DirectX, rozhodl jsem se je z tohoto popisu vynechat.

6.8 Animace

Ještě jednou zde zdůrazňuji, že EE v této fázi žádné animace nepodporuje. Protože ale byly animace původně plánovány a možná se v některé budoucí verzi i objeví, tak tady stručně popíši běžné typy animací používané ve hrách. Některé z nich mohou být v budoucnosti do EE přidány.

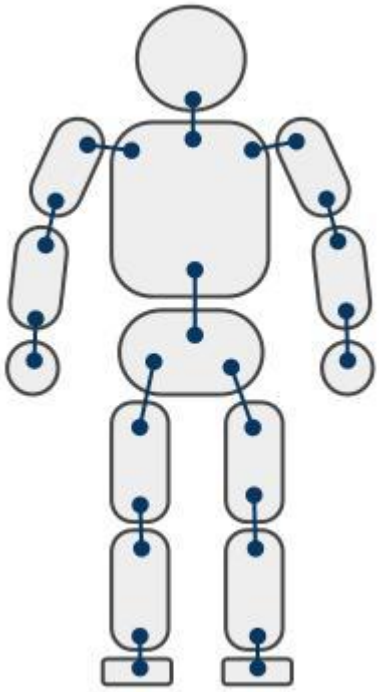
- Sprite/texture animation
- Rigid body hierarchy animation
- Skeletal/skinned animation
- Vertex animation
- Morph targets

Sprite animation představuje tradiční způsob 2d animace, kdy je například postava postupně kreslená v různých fázích pohybu. Rychlým pouštěním těchto obrázků za sebou je možné dosáhnout iluze pohybu. Sprite (malá bitmapa) se kreslí na obraz za sebou bez toho, aby bylo nutné překreslit celé pozadí. Tato jednoduchá technika se velmi často používá i v dnešních pokročilých 3d hrách. Ty-

pické použití je kreslení postav daleko od kamery, pro které je škoda plýtvat drahocenným výkonem. Například pohled na vzdálenou bitvu, ve které bojují stovky vojáků, nebo třeba diváci na stadionu, kterých je příliš mnoho na kreslení 3d modelů.

Rigid Body Hierarchical animation je asi nejstarší metoda tvorby 3d animace. Animovaný model se rozdělí na sérii samostatných 3d modelů. Například lidská postava se může rozdělit na trup, pánev, stehno, lýtko, předloktí, paže, ruce, chodidlo, hlavu. Tyto části se poskládají do logické hierarchie. Pro výše zmíněný příklad by hierarchie vypadala asi takto:

- Pánev
  - Trup
    - Pravá paže
      - Pravé předloktí
        - Pravá ruka
    - Levá paže
      - Levé předloktí
        - Levá ruka
    - Hlava
  - Pravé stehno
    - Pravé lýtko
      - Pravé chodidlo
  - Levé stehno
    - Levé lýtko
      - Levé chodidlo

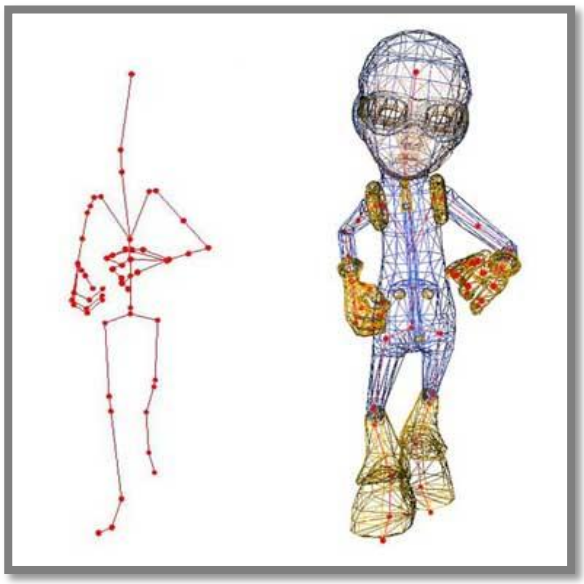


Obrázek 12: Hierarchie částí těla

Při výpočtu animace se pak každý pohyb, který se vykoná na dané části těla, provede i na všech částech v hierarchii pod ním. Tím se dosáhne správné animace všech na sobě závislých částí. Tento systém má jednu obrovskou nevýhodu. Jednotlivé části modelu nejsou nijak spojeny, a tak při jejich pohybu dochází k viditelným trhlinám. Nicméně jsou speciální případy, kde se dá takováto animace bez problému použít. Například animování různých strojů – otáčení hlavně tanku atd.

Obrázek 13: Příklady morph targetů. Převzato z (14)

Vertex animation a morph targets jsou dvě techniky, která se snaží eliminovat nevýhody Rigid Body hierarchical animation. Je třeba pohybovat s jednotlivými body modelu (vertexy), aby se trojúhelníky roztahovaly, a vynikl tak více přirozený pohyb, než při animování samostatných částí. Vertex animation dělá přesně toto. Pro každou animaci si musí pamatovat přesně, jak se který vertex pohybuje. Tím se dá dosáhnout velmi přesvědčivé animace, cena za to je ale velmi vysoká. Je třeba ukládat pozici každého vertexu v každém snímku každé animace. To představuje enormní množství dat, a tak se tato technika ve hrách moc nepoužívá. Morph targets je určité zjednodušení vertex animation. Animátor posune vertexy 3d modelu do krajních/extrémních pozic. Tyto krajní pozice (morph target) jsou jediné, co se ukládá (viz obr 13). Přesná pozice každého vertexu uprostřed animace se potom počítá lineární interpolací (LERP) několika krajních pozic. Tato metoda se příliš nehodí pro animování celé postavy (lineární interpolací se těžko aproximuje komplexní pohyb končetin), ale dá se velmi úspěšně použít k animování obličeje, který je příliš složitý (zhruba 50 svalů) pro modelování ostatními metodami.



Obrázek 14: Skeletal animation. Kostra a vlastní geometrie. Převzato z www.wazim.com/Collada_Tutorial_1.htm

Skeletal animation je metoda, která si zachovává výhody vertex animation, ale zároveň není paměťově ani výkonnostně příliš náročná. Princip je následující. Vytvoří se kostra modelu z jednoduchých kostí (jenom dvojice bodů). Jednotlivé vertexy modelu se poté přiřadí k jednotlivým kostem. Jeden vertex může být přiřazen i k více kostem. V takovém případě se většinou používají váhy, které říkají, do jaké míry vertex patří ke které kosti. Následně se místo všech vertexů animuje jenom kostra a jednotlivé vertexy kopírují pohyb přiřazených kostí. Stejně jako u Rigid body hierarchy je kostra většinou ve formě hierarchie. Skeletal animace je asi nejpoužívanější forma animace ve hrách. Příklad kostry a modelu je na obrázku 14.

Teď, když jsem popsal základní metody animování, se ještě v krátkosti zastavím u několika problémů, které animační systém musí umět řešit.

- Kombinování animací. Není možné vytvořit speciální animace pro každou situaci ve hře. Proto se vytvářejí animace základních pohybů (chůze, běh, skok, pohyb hlavy…), které se podle potřeby kombinují dohromady (animation blending). Existuje celá řada postupů, jak animace kombinovat. Nejčastěji n-dimensionální LERP a additive blending. Komplexní animace se skládají do hierarchií. Je to vcelku rozsáhlá problematika, a nebudu se tím taky zabývat. Jenom uvedu, že kombinování animací musí zajistit správné načasování jednotlivých animací, musí se postarat o to, aby se jednotlivé animace negativně neovlivňovaly, kombinací více animací nesmí dojít k nerealistickému pohybu postavy, musí zajistit věrohodné animování více nesouvisejících pohybů, protože člověk dokáže dělat více věcí najednou. Například jít a zároveň mávat rukou – tyto dvě animace se například pomocí LERPu složit nedají a musí se použít additive blending.
- Post-processing. Po vypočtení animace je v mnoha případech potřeba ještě provést nějakou formu post-processingu, který doladí animaci podle okolních podmínek. Typicky se post-processing používá při chůzi postav. Animace postavy je totiž vytvořená na rovném povrchu. Povrch, po kterém se postava pohybuje, ale už rovný být nemusí. Post processing v takovém případě zajišťuje, aby se nohy postavy vždy dotýkaly země. Podobným problémem je animace zvednutí předmětu ze země, kdy je třeba ruku nasměrovat přesně na pozici objektu. V těchto případech se často používá technika známá jako inverse kinematics (IK). Běžná animace je forma forward kinematics – známe postup animace jednotlivých kostí a počítáme výslednou pozici. U IK známe výslednou pozici a snažíme se odhalit „postup“, který nás k této pozici dovede. To přechází na problematiku minimalizace chyby, kterou se tu nebudu zabývat.
- Komprese. Další problém, který animační systém musí často řešit je komprese animačních dat. Počet animací u moderní hry může být velmi vysoký a je potřeba snížit celkové množství ukládaných dat.

Na závěr ještě poznamenám, že animační systém musí úzce komunikovat s rendererem a často i s fyzikálním enginem.

6.9 Fyzika

V této části se představím fyzikální subsystém EE. Nebudu zde zdlouhavě rozebírat principy fyzikální simulace v herních enginech. To by vydalo na vlastní knihu. Zaměřím se na způsob integrace fyziky do EE. Jak bylo dříve uvedeno, fyzikální engine není pevnou součástí enginu. Pevnou součástí je

pouze rozhranní, přes které je možné připojit libovolný fyzikální engine. Nejdřív ale lehký úvod o fyzice ve hrách obecně.

6.9.1 Fyzika ve hrách

Fyzika ve hrách se používá k simulování řady věcí. Realistický pohyb pevných těles, simulace strojů (automobily, letadla…), simulace hadrového panáka (Rag Doll), zničitelné prostředí, simulace vody, látky (oblečení, vlajky), detekce kolizí, šíření zvuku a další. Na první pohled se zdá, že fyzika by se měla používat ve všech hrách. S fyzikou je ale spojena řada problémů. Přidání realistické fyziky může ve výsledku hru poškodit. Do určité míry to záleží na typu hry. Například pro simulaci závodní hry bude fyzikální simulace přínosem. Na druhou stranu u hry řízené příběhem, může fyzika věci komplikovat. Například u destrukce nějakého objektu chceme, aby probíhal kontrolovaným způsobem (pokaždé stejně), aby nedošlo například k zatarasení cesty dál. Mezi dopady přidání fyziky do hry patří následující:

- Předvídatelnost. V animovaném světě je všechno pevně dáno. Oproti tomu fyzikální simulace je nepředvídatelná. Navíc ani simulace se stejným vstupem nedopadne pokaždé stejně kvůli numerickým nepřesnostem.
- Ladění a kontrola. Fyzikální zákony jsou sice fixní, ale ve hře je možné řadů věcí ovlivnit. Například můžeme měnit velikost gravitace. Výsledky ladění parametrů fyziky jsou ale nepřímé. Těžko se odhaduje jejich dopad a těžko se ladí.
- Umělá inteligence (AI). Navigace v dynamicky se měnícím prostředí už není tak jednoduchá. AI může nějakým způsobem používat fyziku.
- Animace. Engine musí umožňovat plynulý přechod od animace k fyzikální simulaci.
- Grafika. Zničitelné prostředí může znehodnotit předpočítaná data, jako jsou speciální textury s osvětlením a stíny (Lightmaps, Shadowmaps).
- Multiplayer. Fyzikální efekty, které ovlivňují hratelnost, musí být simulovány na serveru a replikovány na všech klientech.
- Záznam hraní. Zaznamenat hraní a následně si ho přehrát může být velmi užitečné během vývoje/testování. Implementace takového systému je mnohem obtížnější s fyzikální simulací.
- Ztráta kontroly. Při návrhu herního světa má designér mnohem menší kontrolu nad tím, co hráč uvidí a zažije.
- Zvýšená pracnost vývoje. Ve všech odvětvích přidává fyzika práci navíc. Od tvorby 3d modelů až po programování.

6.9.2 Základní principy

V této kapitole stručně vysvětlím základní princip fyzikálního enginu ve hrách. Fyzikální engine má zpravidla vlastní reprezentaci scény, ve které jsou všechny fyzikou řízené objekty. Nad těmito objekty se provádí simulace. Kvůli numerickým nepřesnostem se fyzika simuluje většinou více než jednou za snímek. Kromě vlastních těles jsou ve fyzikální scéně i různé druhy kloubů, pružiny a další objekty pro spojování těles. Využívají se například pro simulaci dveří na pantech atd. Fyzikální 3d objekty nejsou to samé, jako grafické 3d objekty. 3d objekt určený pro grafiku se skládá řádově z tisíců polygonů. Detekce kolizí u takto složitých objektů by byla příliš náročná, a tak musí být fyzikální objekty reprezentovány mnohem jednodušší geometrií. Jsou to například konvexní objekty s řádově desítkami až stovkami trojúhelníků nebo základní geometrické útvary – koule, kvádr… Objekt ve hře tedy musí mít grafický 3d model, fyzikální 3d model a řadu parametrů, jako je hmotnost, odpor tření atd.

6.9.3 Fyzikální rozhraní

Už jsem uvedl, že EE neobsahuje fyzikální engine. Obsahuje pouze rozhraní. Toto rozhraní není nic jiného než série tříd s virtuálními funkcemi, které je třeba implementovat. Samotný engine používá výhradně toto rozhraní. Uživatel enginu tedy může použít libovolný fyzikální engine, pokud dokáže komunikovat s rozhranním enginu. Toto rozhraní bylo vytvořeno na míru fyzikálnímu enginu Physx, ale protože všechny fyzikální enginy jsou si nabízenými funkcemi velmi podobné, neměl by být problém použít i jiný engine.

6.9.4 Physx

Jak bylo naznačeno v předchozí kapitole, vzorová implementace fyzikálního enginu zapojitelného do EE, je založená na fyzikálním enginu Physx od NVidie. Physx je volně šířený, snadno použitelný fyzikální engine. Physx byl použit v mnoha komerčních hrách a herních enginech. Dodává se s kvalitní dokumentací a komunita uživatelů tohoto enginu je velmi živá. To byly hlavní důvody, proč jsem si zvolil zrovna Physx.

Physx je navržen tak, aby simulace běžela ve vlastním vláknu. Synchronizace grafiky a fyziky se provádí jednou za snímek. EE engine tedy nativně podporuje rozdělení výpočtu na dva procesory. Jeden počítá fyziku a druhý vše ostatní.

6.10 Skriptování

Hlavním úkolem skriptů ve hrách je oddělit náplň herního světa a jednoduchou herní logiku od zbytku aplikace. Tyto informace se často mění, ladí a upravují. Ukázalo se, že je velmi nepraktické překládat zdrojové kódy celé aplikace kvůli změně jednoho řádku dialogu. Takových změn se pochopitelně provede během vývoje enormní množství. Skripty mají ve hrách široké uplatnění, ale přede-

vším se používají ke dvěma úkolům. Za prvé, popis vlastností, předmětů, jevů, postav obsažených ve hře. Například každá zbraň může obsahovat informace o dostřelu, velikosti poškození, které způsobuje, typu poškození atd. Druhým hlavním způsobem využití skriptů je ovládání základní herní logiky. Například při použití výše zmíněné zbraně se spustí skript, který zkontroluje, jestli postava může danou zbraň používat. Následně může spočítat na základě typu a velikosti poškození celkové zranění způsobené cílu. Stejný skript může spustit unikátní grafický, zvukový efekt. Druhý způsob je implementace herní logiky. Skript může být spuštěn po vstoupení hráče na určité místo. Takový skript odstartuje předpřipravenou událost. Například se zavřou dveře, přehraje se hudba, zobrazí se grafický efekt atd.

Skriptování v počítačových hrách se vyskytuje v několika podobách. Nejčastěji je to skriptování založené na příkazech (command based) a v podobě skriptovacího jazyka. Command based skriptování vytváří herní logiku pomocí sady předpřipravených příkazů (jdi na XY, přehraj hudbu, odeber zdraví…). Takový systém je snadno implementovatelný, ale pro složitější logiku není dostatečně flexibilní. Další možností je skriptovací jazyk. Ve skriptovacím jazyku s oboustrannou komunikací je možné vytvořit prakticky cokoliv.

Skriptovací jazyk by měl ideálně zajistit následující vlastnosti:

- Oddělenost. Skripty musí být oddělené od vlastní aplikace, aby se zabránilo neustálým překladům celé aplikace.
- Bezpečnost. Případná chyby ve skriptu by neměla narušit celou aplikaci. Skripty ve hrách nepíší výhradně profesionální programátoři, a pravděpodobnost výskytu chyb se tím zvyšuje. Například interpret jazyka může chybu včas zachytit, a zabránit tak pádu aplikace.
- Rychlost. V prostředí počítačových aplikací je důležité, aby skripty nebrzdili celou aplikaci.
- Jednoduchost. Skriptovací jazyk by ideálně neměl obsahovat zbytečně složité jazykové konstrukce. Skripty nepíší vždy programátoři. Často je to designér.
- Komunikace s hostující aplikací. Pro plné využití síly skriptů je nezbytné, aby skripty mohly volat funkce z hlavní aplikace a naopak.

6.10.1 ESL

Jak bylo dříve zmíněno. EE není přímo spojen s žádným skriptovacím systémem. Je možné k skriptování použít cokoliv. Nicméně jako součást EE byl vyvinut i skriptovací jazyk. Jazyk nese jméno Eclipse Scripting Language (ESL). Kompletní popis jazyka ESL by zabral mnoho místa, a tak zde stručně představím jeho základní vlastnosti. Zájemce o další detaily se může podívat na moji bakalářskou práci – Implementace skriptovacího jazyka (viz (3)).

Vlastnosti ESL

- Procedurální, interpretovaný, skriptovací jazyk. ESL je interpretovaný a skládá se z překladače a interpreteru (Virtual Machine).
- Interní halda spravovaná garbage collectorem. ESL disponuje automatickou správou paměti.
- Skripty použitelné jako inicializační soubory. Proměnné ve skriptu se dají číst a modifikovat v hostující aplikaci. Skripty se dají snadno použít jako inicializační sobory.
- Integrace s hostující aplikací. ESL byl navržen pro snadné a bezpečné integrování do hostující aplikace. Podporuje obousměrné volání funkcí i předávaní dat.
- C-like syntaxe. Většina programovacích (skriptovacích) jazyků současnosti do určité míry vychází ze syntaxe jazyka C. ESL není výjimkou, když jeho syntaxe představuje něco mezi jazykem C a C#.
- Register based. Interně se mezivýsledky výpočtů ukládají v registrech místo použití zásobníku. Důvodem byla snaha zvýšit rychlost výpočtu.

6.11 Nástroje

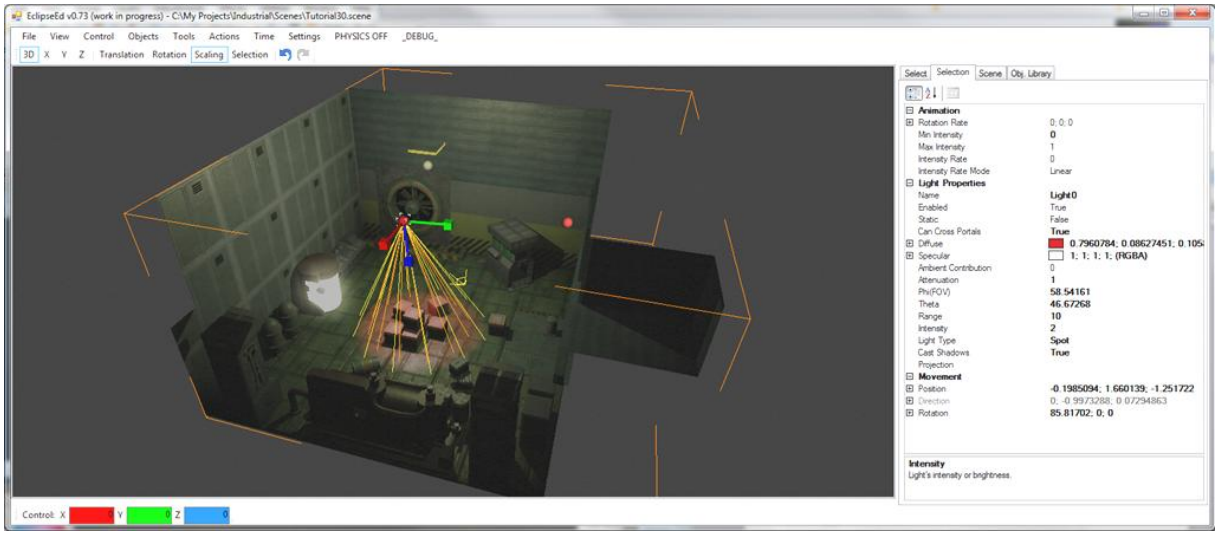
V této části představím dva nástroje, které vznikly společně s EE.

6.11.1 X2EMF

Prvním nástrojem je malá aplikace X2EMF. Je to konvertor 3d modelů. Konvertuje z formátu .x, který vytvořil jako součást DirectX Microsoft, do formátu .emf (Eclipse Mesh File), který používá EE. X2EMF je malá utilita, která disponuje náhledem 3d modelu a umožňuje některé základní operace. Změnu velikosti, změnu těžiště, výpočet normál, binormál a tangent. Dále umožňuje provést základní optimalizace modelu – odstranění nepoužitých bodů, přeorganizování bodů pro minimalizaci cache miss atd. Po volitelných úpravách se 3d model převede do formátu emf. Tento formát podporuje například GLOD, speciální reprezentaci 3d modelu pro účely generování stínu (tento model nemusí být tak detailní). I model pro generování stínů podporuje GLOD. Dále je v souboru emf fyzikální reprezentace modelu. A v neposlední řadě třeba informace o obalovém kvádru a kouli (Bounding Volumes). Generovaný soubor emf dokáže engine zpracovat.

6.11.2 EclipseEd

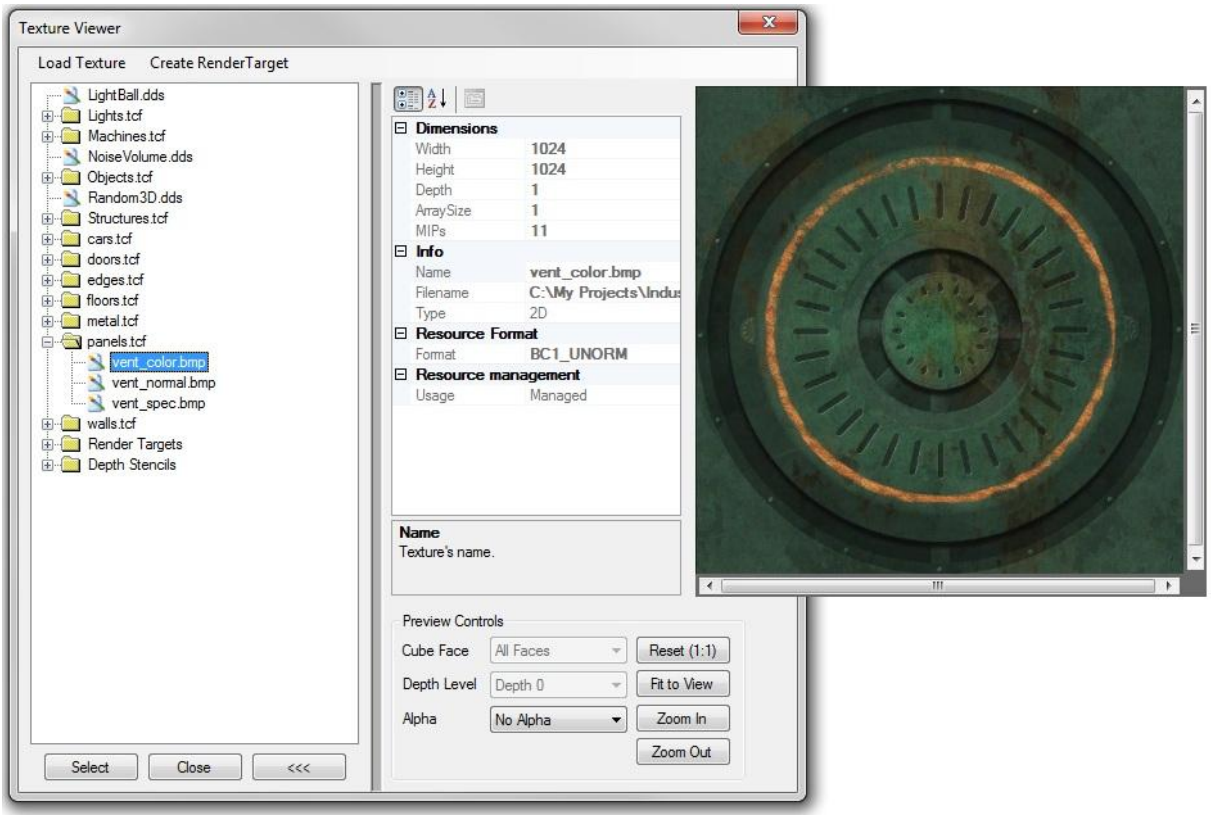
Druhým nástrojem je aplikace EclipseEd (Eclipse Editor). Je to velmi rozsáhlá aplikace, která umožňuje editovat celou řadu věcí. Od tvorby materiálů, přes rozmístění objektů na scéně, až po organizování vykreslování a tvorbu post proces efektů. Editor byl napsán podle filozofie WYSIWYG. Všechny provedené změny mají tedy odezvu v reálném čase a na vše existují náhledy. To, co je vidět v editoru, je identické s tím, co je vidět ve finální aplikaci. Vzhledem k velkému rozsahu EclipseEd zde uvedu jenom výčet funkcí doplněný o několik ilustračních obrázků. Ještě zde doplním, že EclipseEd byl napsán v jazyce C++/CLI.



Obrázek 15: Hlavní okno EclipseEd

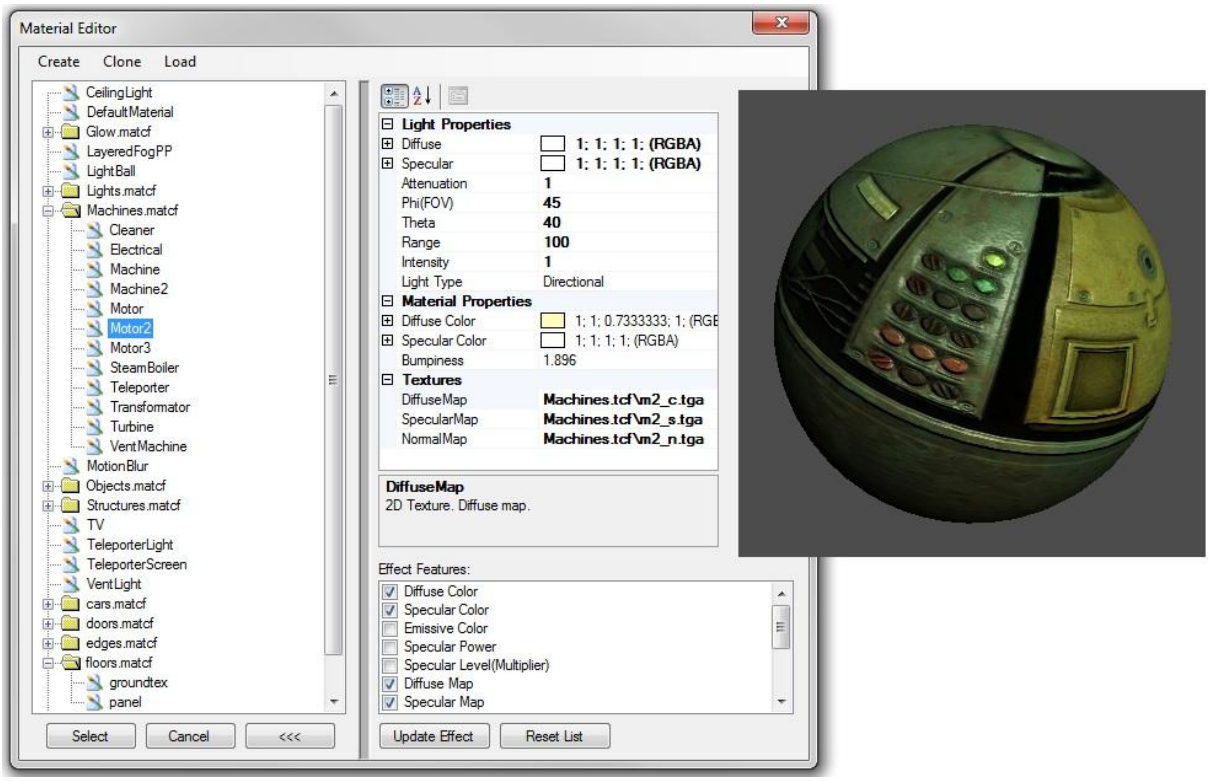
Vlastnosti:

- Podpora více projektů
  - Každý projekt má odkazy na místa, kde jsou uloženy jednotlivé zdroje, dále má vlastní knihovnu objektů a nastavení editoru
  - Každý projekt může obsahovat libovolný počet scén
  - Projekty si pamatují naposledy otevřené scény a editor si pamatuje naposledy otevřené projekty.
- Ovládání fyziky a času
  - Editor umožňuje spustit a přerušit simulaci fyziky
  - Kontrola rychlosti běhu času
- Knihovna objektů
  - Hierarchická (adresářová) struktura pro uložení „vzorů“ jednotlivých objektů
  - Vzory se dají jednoduchým přetažením myši rozmísťovat do scény
- Správa textur
  - Načítání textur ze souboru (po jedné, nebo hromadně) s možností změnit mnohé parametry ještě před načtením (rozlišení, formát, počet mip levelů…)
  - Náhled načtených textur. Podporuje speciální náhledy „cube textur“ a „volume textur“. Možnost prohlížet zvlášť barevnou část, alfa kanál, nebo jejich kombinaci.
  - Správa kolekcí. Textury je možné v editoru přeskupovat do kolekcí (případně z kolekcí do samostatných souborů).



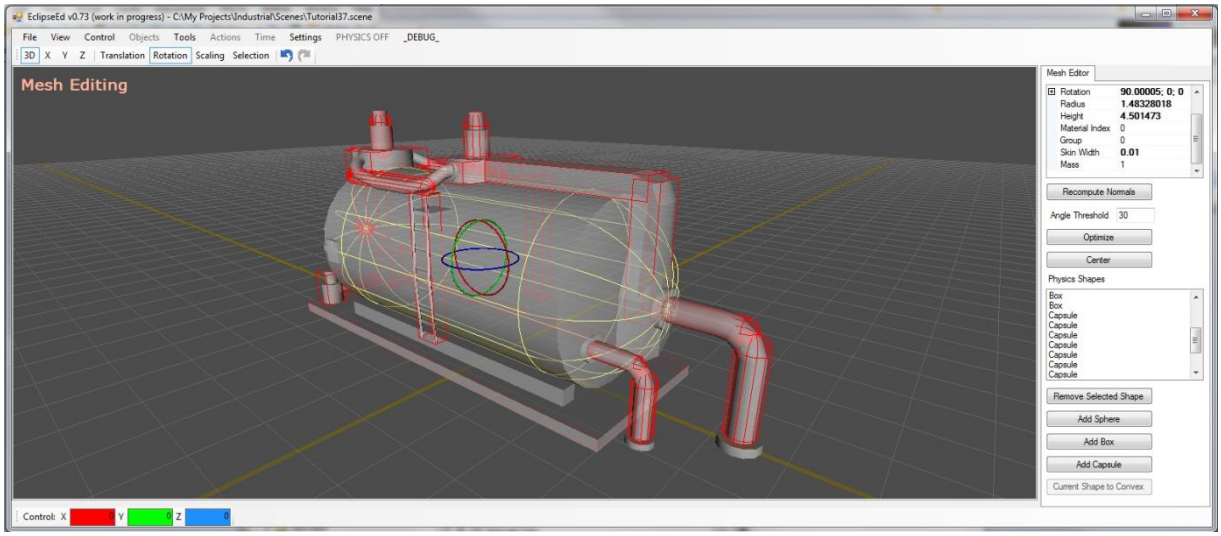
Obrázek 16: EclipseEd. Správce textur

- Správa světel
  - Editace vlastností jednotlivých světel od typu světla až po projekci textury
- Správa 3d objektů
  - Editování jednotlivých částí 3d objektů
    - Přiřazování materiálů
    - Určení, kdy se mají vykreslovat
  - Editování celých objektů
    - Změna pozice, rotace, změna velikosti, jména, informace o stínech, hmotnosti a mnoha dalších.
  - Real-time náhled pro celý objekt, nebo pro jednotlivé části
- Správa materiálů
  - Tvorba nových materiálů. Výběr vlastností, které má materiál mít a následná editace konstant, přiřazování textur. Vše včetně real-time náhledu.
  - Načítání předpřipravených materiálů ze souboru.
  - Správa kolekcí. Přeskupování materiálů z/do kolekcí.



Obrázek 17: EclipseEd. Editor materiálů

- Editace 3d modelů
    - Změna pozice, rotace, změna velikosti, tentokrát ale přímo v 3d modelu. Po těchto změnách se napevno přepočítají jednotlivé vrcholy 3d sítě
    - Přepočítání normálových vektorů – možnost odstranění/získání ostrých přechodů – parametrem je úhel, za kterým už je hrana ostrá.
    - Vytváření fyzikální obálky – tvorba ze základních fyzikálních primitiv
    - Editace „default“ fyzikálních vlastností – hmotnost, pružnost, rozložení hmotnosti…



Obrázek 18: EclipseEd. Editace 3d modelů

- Správa scény
    - Editor zobrazuje hierarchicky uspořádané všechny objekty ve scéně (3d objekty, světla, buňky, portály)
    - Možnost vybrání, označení objektu kliknutím
- Editor místností (statické geometrie)
    - Každá buňka má po vytvoření šest stěn, které se dají dále editovat
    - Možnost řezat stěny (jejich části), aby bylo možné aplikovat na různé části různé materiály
    - Možnost vyřezávat do stěn libovolně otvory
    - Funkce extrude/intrude po vybrání části stěny
    - Všechny řezy musejí být pravoúhlé. Výřez a extrude/intrude pouze pro obdélníky
    - Veškeré funkce jsou prováděné přímo v editoru pomocí myši s možností manuální korekce (přímé zadání/korekce souřadnic)
    - Přiřazení materiálu ke stěně, editace umístění textury.
    - Propojování buněk průchody (další buňky)
    - Automatické generování geometrie pro fyzikální engine
- Editor rendereru
    - Vykreslování je rozděleno na jednotlivé kroky. Např. změna render targetu, změna kamery, post-processing, vykreslení geometrie a mnoho dalšího
    - Editor může tyto kroky přidávat, mazat a editovat, a tím určovat co, kdy a jak se vykreslí.
    - Tento systém umožňuje enginu dosáhnout téměř libovolného efektu, nebo stylu vykreslování.
- Hlavní okno editoru
    - Umožňuje pomocí myši snadno zacházet s jednotlivými objekty (včetně kamery)
    - Podpora kopírování objektů pomocí myši
    - Možnost posouvat, rotovat a měnit měřítko objektů
        - To se mění v závislosti na typu objektu. Například posunutí u zdi neposune zeď, ale příslušnou. Změna velikosti u bodového světla zase mění dosah světla a velikost výseče…
        - Možnost zhruba dosáhnout požadovaného efektu pomocí myši, a následně upřesnit zadáním přesné hodnoty
    - Vybírání objektů kliknutím

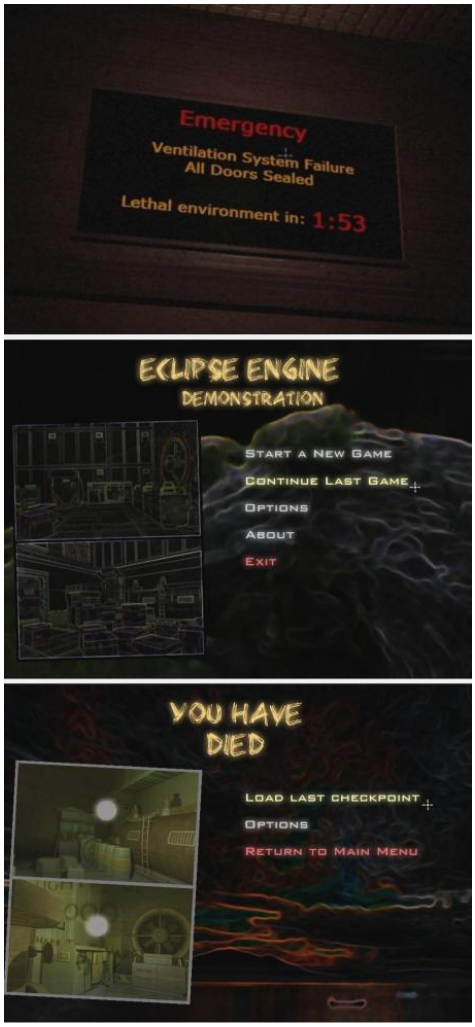
  - Několik módů ovládání kamery
  - Perspektivní i homogenní promítání
- Další vlastnosti
  - Vykreslování libovolné mřížky
  - Funkce přichytávání k mřížce (nemusí se shodovat s vizuální)
  - Zvýraznění označeného objektu + vykreslení dalších informací (rozsah světla)
  - Volitelné vykreslování objektů světel
  - Generování light map
  - Klávesové zkratky
  - Možnost vracet zpět změny – funkce undo, redo

7 EclipseDemo

Obrázek 19: EclipseDemo. Ilustrační obrázky.

EclipseDemo je malá aplikace, která má za úkol názorně prezentovat Eclipse Engine. Ideální by bylo vytvoření celé hry na Eclipse Enginu, to ale není pro jednoho člověka za pár týdnů možné. EclipseDemo je tedy krátká ukázka, která ale obsahuje vše, co by měla obsahovat plná hra, ale pouze v malém rozsahu. Na EclipseDemo se dá nahlížet jako na jednu malou úroveň z celé hry. V následující části stručně popíši, o co v EclipseDemu jde. Pro ilustraci je zde uvedeno několik obrázků.

Po spuštění aplikace se nejdříve zobrazí menu, kde je možné spustit novou hru, pokračovat v předchozí hře, měnit nastavení a přečíst si informace o aplikaci včetně popisu ovládání. Celá hra je potom viděna z pohledu vlastních očí. Hráč se ocitne v místnosti, v jakémsi skladu. Na obrazovce na stěně se postupně objevují požadavky na různé předměty. Tyto předměty musí hráč přemístit do teleportu, který se v místnosti nachází. Pokud na teleport hráč postaví správný předmět, tak dojde k jeho přenesení. V této fázi se hráč může seznámit s ovládáním a s manipulováním s předměty, které jsou řízeny reálnou fyzikou. Hráč může předměty uchopit, zahodit, odstrkovat vlastním pohybem, nebo si na ně stoupat. Při jednom transportu požadovaného předmětu dojde k nějaké chybě. Celá budova se otřese, zhasnou světla a zastaví se ventilace. Místnost se začne plnit nedýchatelným plynem těžším než vzduch (drží se při zemi, jeho úroveň postupně stoupá) a hráčovým úkolem je utéct, než se místnost naplní plynem a hráč se udusí. Hráč se musí dostat k ventilační šachtě blízko stropu. Jediným způsobem, jak se tam dostat, je postavit si z různých okolních předmětů improvizované schody. V místnosti je ale tma po výpadku světel. Naštěstí byla při poruše teleportu do místnosti přenesena koule emitující světlo zvaná Light-



Obrázek 20: EclipseDemo. Další ilustrační obrázky.

Ball. LightBall se bohužel rychle vybíjí, čímž klesá množství světla, které vyzařuje. Pro nabití je třeba LightBall uchopit a zatřást s ním. Hráči tedy dochází čas, musí se neustále starat o světlo, a k tomu musí stavět improvizované schody do bezpečí. Pokud se hráč dostane včas do ventilační šachty, tak demo úspěšně dokončil. V opačném případě následuje vypršení časového limitu a smrt.

Z technického hlediska využívá EclipseDemo reálnou simulaci fyziky a řadu grafických efektů. Mezi efekty patří například plně dynamické osvětlení včetně stínů, High Dynamic Range Rendering s automatickou adaptací na světlo, volumetrické světlo, motion blur, frame buffer distortion, grain filter a další.

Celkově toto technologické demo ukazuje, že Eclipse Engine je funkční engine, který se dá použít k vývoji počítačové hry.

8 Závěr

V této práci jsem nejdříve vysvětlil, co to vlastně je herní engine. Byla popsána architektura běžného herního enginu, a nakonec jsem i rozebral mnou vytvořený engine Eclipse.

8.1 Vývoj

Vývoj současného Eclipse Enginu pro mě byl v některých chvílích velmi obtížný. Asi nejtěžší byly samotné začátky, tedy jádro enginu. To znamenalo napsat velké množství kódu bez jakéhokoliv viditelného výsledku. Často jsem obtížně hledal motivaci a práce moc nepokračovala. V té době to navíc byla moje soukromá aktivita ve volném čase, a nic mě tedy nenutilo pokračovat. Kromě snahy splnit si svůj sen. Jak postupně přibývaly další a další funkce a engine už se dal použít k vykreslování, tak se situace o dost zlepšila a vývoj mě začal bavit. V té době začal i vývoj EclipseEd. Ten nebyl původně plánován jako plnohodnotný editor. Měl a to být jenom malá utilita pro přiřazování materiálů 3d objektům. To vše s náhledem a osvětlením. Nakonec se ale rozrostl na vcelku velký a propracovaný editor. Tato neplánovaná změna se ale trochu podepsala na interní struktuře. Funkce se postupně přidávali bez globální vize. Výsledkem jsou zdrojové kódy, které mají k přehlednosti a čistému návrhu daleko. Týká se to ale pouze editoru.

Poslední fází byl vývoj demonstrační verze. To byla s dostupnými nástroji vcelku jednoduchá práce. A hlavně velmi zábavná práce. Opět jsem si připomněl, že vytvořit si vlastní svět, a pak se v něm procházet, je velmi uspokojivá činnost. Ani tato pro mě zábavná fáze ale nebyla bez problémů. Teprve při snaze vytvořit počítačovou hru se ukázaly některé nedostatky v návrhu enginu. Nešlo o nic vážného, ale pochopil jsem, že některé věci by se s enginem dělaly obtížně. Prostě jsem se naučil další věc. Pro návrh herního enginu je dobré mít hodně zkušeností s tvorbou vlastních her.

8.2 Shrnutí

Podařilo se mi vytvořit 3d engine, který je použitelný pro tvorbu počítačové hry. To vše jsem demonstroval na krátké ukázce. Vytvoření 3d enginu tohoto rozsahu není triviální činnost a já osobně výsledek považuji za velký úspěch. Během práce na tomto projektu se mi podařilo získat velké množství zkušeností. Řadu z nich metodou pokus omyl, ale o to lépe si je budu pamatovat. EE se komplexností samozřejmě nemůže rovnat komerčním řešením, ale to co dělá, zvládá podle mě dobře a přínos ve formě zkušeností je pro mě osobně nenahraditelný.

Celý projekt se také nečekaně rozrostl. V tuto chvíli se skládá zhruba ze 100 000 řádek kódu. Z toho přibližně 53 000 spadá na engine, 31 000 na editor, 11 000 na skriptovací jazyk a zbylých asi 5 000 na EclipseDemo. Z toho je mimo jiné vidět, že většinu práce odvede engine a vlastní aplikace je relativně malá.

Vše se samozřejmě nepovedlo perfektně. Se zkušenostmi, které mám nyní, bych některé věcí asi řešil jinak, ale tak je to vždy. Také se vzhledem k rozsahu v projektu určitě vyskytují nějaké chyby.

Pravdou ale je, že samotný engine se momentálně zdá být velmi stabilní. Oproti tomu u editoru je několik známých a pravděpodobně i neznámých chyb, které komplikují jeho používání.

Použitá literatura

1. **Gregory, Jason.** *Game Engine Architecture.* Wellesley : A K Peters, 2009. ISBN: 978-1-56881-413-1.
2. **Moller, Thomas and Haines, Eric.** *Real-Time Rendering.* s.l.: A K Peters, 1999. ISBN: 1-56881-101-2.
3. **Skalský, Michal.** *Implementace skriptovacího jazyka.* Plzeň, 2008.
4. **Eberly, David H.** *Game Physics.* s.l. : Morgan Kaufmann, 2004. ISBN: 1-55860-740-4.
5. **Adams, Jim.** *Programming Role Playing Games with DirectX.* s.l. : Premier Press, 2002. ISBN: 1-931841-09-8.
6. **Dempsky, Kelly.** *Real-Time Rendering Tricks and Techniques in DirectX.* s.l. : Premier Press, 2002. ISBN: 1-931841-27-6.
7. **McCuskey, Mason.** *Special Effects Game Programming with DirectX.* s.l. : Premier Press, 2002. ISBN: 1-931841-06-3.
8. **Bergen, Gino Van Den.** *Collision Detection in Interactive 3d Environments.* s.l. : Morgan Kaufmann, 2004. ISBN: 1-55860-801-X.
9. **Buckland, Mat.** *Programming Game AI by Example.* s.l. : Wordware Publishing, 2005. ISBN: 1-55622-078-2.
10. **Varanese, Alex.** *Game Scripting Mastery.* s.l. : Premier Press, 2003. ISBN: 1-931841-57-8.
11. **Bourg, David. M.** *Physics for Game Developers.* s.l. : O'Reilly, 2002. ISBN: 0-596-00006-5.
12. **Engel, Wolfgang.** *Programming Vertex and Pixel Shaders.* s.l. : Charles River Media, 2004. ISBN: 1-58450-349-1.
13. **Kolektiv autorů.** *Game Programming Gems.* s.l. : Charles River Media, 2000. ISBN: 1-58450-049-2.
14. **Kolektiv autorů**. *GPU Gems.* s.l. : Addison-Wesley, 2004. ISBN: 0-321-22832-4.
15. **Kolektiv autorů**. *GPU Gems 3.* s.l. : Addison-Wesley, 2007. ISBN: 0-321-51526-9.
16. **Kolektiv autorů**. *Shader X3.* s.l. : Charles River Media, 2005. ISBN: 1-58450-357-2.
17. **Kolektiv autorů**. *Shader X2.* s.l. : Wordware, 2004. ISBN: 1-55622-988-7.

Přehled obrázků