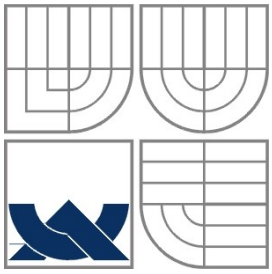
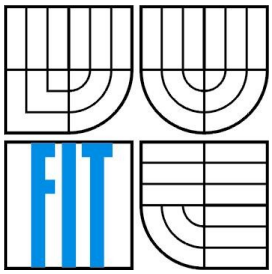


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

FYZIKÁLNÍ SIMULACE V POČÍTAČOVÝCH HRÁCH

PHYSICAL SIMULATION IN COMPUTER GAMES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

BC. JIŘÍ DOČKAL

VEDOUCÍ PRÁCE
SUPERVISOR

ING. ADAM HEROUT, PH.D.

BRNO 2010

Abstrakt

Práce se zabývá problematikou moderních herních enginů se zaměřením na fyzikální simulace a částicové systémy. Nabízí přehled použitelných architektur pro vývoj herního enginu. Poskytuje charakteristiku jeho nejdůležitějších logických modulů jako scénový graf, správa zdrojů nebo vykreslování. Popsány jsou také dnešní nástroje pro fyzikální simulaci ve hrách. Hlavní část práce je soustředěna na návrh a implementaci vlastního herního enginu C3D, který využívá možností fyzikálního enginu NVIDIA PhysX. Práce nabízí moderní techniky, které vycházejí z autorových zkušeností.

Abstract

The thesis is concerned with modern game engines, focusing on physical simulation and particle systems. It offers usable architectures overview for a game engine development. The thesis provides characteristic to the most essential game engine's logical modules as scene graph, resource management or rendering. Today's tools used for physical simulation in games are also described. Main part of the thesis concentrates on design and implementation of its own C3D game engine which exploits capabilities of the NVIDIA PhysX physical engine. The thesis includes modern techniques rising from author's gained experience.

Klíčová slova

Herní engine, NVIDIA PhysX, aktor, spoj, fyzikální simulace, měkké částice, částicový systém

Keywords

Game engine, NVIDIA PhysX, aktor, joint, physical simulation, soft particles, particle system

Citace

Dočkal Jiří: Fyzikální simulace v počítačových hrách, diplomová práce, Brno, FIT VUT v Brně, 2010

Fyzikální simulace v počítačových hrách

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Adama Herouta, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jiří Dočkal
26.5.2010

Poděkování

Je mojí milou povinností vyjádřit alespoň takto velkou vděčnost a pokoru svému vedoucímu panu Ing. Adamu Heroutovi, Ph.D. za jeho zkušené rady, přátelský přístup, ochotu a motivaci k další práci.
Také děkuji tvůrcům grafických modelů lodí pro technologické demo.
Tie Fighter: James Bassem (www.jrbassett.com)
X-Wing: Dr. Harry Chang, Don Showalter, Matt Walton, Jose Gonzales Pareja, Matt Allen.

Obsah

1 Úvod

Fyzikální simulace ve hrách se během posledních několika let stala velmi důležitou. Efekty, jako jsou bortící se domy, obrovské exploze, deformace objektů, turbulence, kouř nebo tekutina zvýšily realističnost her na vysokou úroveň. Současně je však výpočet těchto efektů velmi náročný a CPU je v dnešních herních enginech vytíženo jinými úlohami. Na druhou stranu fyzikální simulace zahrnuje stejné výpočty pro velký počet objektů a stává se tak vhodným kandidátem pro umístění alespoň části výpočtu na moderní GPU, které jsou dnes mocnými paralelními procesory.

Cílem práce je prostudovat architekturu herních enginů a souvisejících nástrojů pro fyzikální simulaci ve hrách. Získané poznatky následně využít při návrhu a implementaci vlastního herního enginu, který bude integrovat fyzikální engine NVIDIA PhysX. Implementovanou funkcionalitu prověřit demonstrační aplikací. Práce nechce čtenáři nabídnout vyčerpávající návod k použití NVIDIA PhysX, snaží se spíše poskytnout rady, které vycházejí z autorových zkušeností. Cílem práce je také zhodnotit realističnost fyzikálních simulací ve hrách.

Následující kapitola popisuje poznatky získané studiem moderních herních enginů. Je vysvětlena podstata a architektura herních enginů. Charakterizovány jsou logické moduly enginu jako správa paměti, zdrojů, vykreslování a samozřejmě také nástrojů pro fyzikální simulace. Třetí kapitola detailně vysvětluje návrh a implementaci vlastního herního enginu s důrazem na integraci, vlastnosti a zákoutí využitých modulů fyzikálního enginu PhysX. Konečně čtvrtá kapitola poskytuje cenné zkušenosti získané při práci s použitým fyzikálním systémem a provádí čtenáře implementací demonstrační aplikace. Ve třetí a čtvrté kapitole najdeme podněty a nápady pro další práci na projektu stejně jako zhodnocení dosažených výsledků.

2 Herní engine

Herní projekt je dnes obrovskou aplikací, která v sobě kombinuje téměř všechny domény vývoje aplikací. To o jak rozsáhlých projektech je tu řeč si ukážeme na příkladu. Bioshock od společnosti 2K Boston/2K Australia se stal hitem na konci léta roku 2007. Na projektu pracovaly dva týmy vývojářů. Jen použitý engine, Unreal Engine 3, obsahuje okolo dvou miliónů řádek kódu [LOWENSOHN10]! Aby byl čas vývoje takové aplikace přijatelný hlavně z ekonomického pohledu, je nezbytný kvalitní návrh projektu. Vzhledem k velikosti projektu je využíváno objektového návrhu software. S ním pak souvisí i volba implementačního jazyka. Největší roli zde hrají faktory ovlivňující výslednou rychlost aplikace. Z tohoto důvodu jsou voleny jazyky kompilované. Nejpoužívanější je stále jazyk C++ [GAMEWIKI10], i když jsou i jiné možnosti.

V dřívější době, když nebyly herní projekty tak rozsáhlé, zejména z důvodu slabšího výkonu hardware, byla hra tvořena jedním celistvým programem. Každá taková hra byla originálem a vývoj začínal vždy od nuly [ENGINE10]. Díky tomu co dnes umožňuje výkon počítačů, vzrostla potřeba znovupoužití kódu. Na nejvyšší úrovni pohledu na hru je hra rozdělena na dvě základní části: logiku hry a herní engine. Při tomto modelu pak logika hry využívá vlastností a schopností enginu k vytvoření herního světa a zákonitostí, které v něm fungují. Zmíněný model společně s daty tvoří hru.

Samotný engine lze od herní logiky oddělit až do té míry, že se engine stává samostatným frameworkem, který lze opakovaně použít s nutnými úpravami v nových hrách. Jeho vývoj je však náročný jak na čas, tak na finanční prostředky a lidské zdroje. Navržení kvalitního enginu je nesmírně obtížné a náročné. Je zapotřebí mnoha zkušeností z práce na podobných projektech a obrovský rozhled v této široké problematice. Proto se rozvinul trh s licencemi enginů [ENGINE10]. Enginy prověřené kvalitní hrou jsou velmi ceněné – až miliony dolarů [ENGINE10]! To, jak těžké je takový špičkový engine vytvořit je zřejmé. Komerčně úspěšných enginů je ovšem velmi málo řádově do deseti. Úspěšný engine musí být především stabilní, snadno rozšiřitelný, skvěle optimalizovaný a musí zvládnout obrovské množství požadavků vyplývajících z daného žánru vyvíjené hry. Zde je důležité říci, že neexistuje univerzální engine, který by se dal bez jakýchkoli úprav nasadit do každého herního projektu. Žádný engine současnosti zatím nevyhovuje všem stanoveným požadavkům stoprocentně. Nejblíže se, dle mého názoru, k tomuto cíli zatím přiblížilo vývojové studio Epic Games, Inc. se svým enginem Unreal Engine 3. Uvedený engine se asi jako jediný v současné době vypořádává se všemi myslitelnými aspekty nejlépe a poskytuje kompletní vývojové

prostředí jak pro programátory, tak pro grafiky. Důkazem budiž jeho velká obliba mezi profesionálními vývojovými týmy a nespočet vydaných her.

Obrázek 2.1 - Unreal Engine 3 v akci

Velkým problémem při snaze o vytvoření univerzální knihovny je právě optimalizace všech aspektů enginu. Je nezbytné, aby výpočet vždy proběhl v dostatečně krátké době, která je požadována pro interaktivní zábavu.

Díky dnešnímu výkonu počítačů by bylo nesmyslné vyvíjet 2D herní engine. Taková hra by nevyužila všech možností moderních počítačů a byla by odsouzena ke komerčnímu neúspěchu i přes svoji výtečnou hratelnost a nápad. Proto se ve své práci budu věnovat pouze těm nejmodernějším řešením 3D enginů. Důležité je říci, že každá z podkapitol by bez nejmenších problémů posloužila jako samostatné a zcela vyčerpávající téma např. pro disertační práci. Výklad jsem omezil na nejdůležitější aspekty, tak jak dovoluje rozsah této práce.

2.1 Architektura

Architektura každého softwarového produktu má zásadní vliv na dosažené výsledky. Při chybném návrhu jsou ztráty (jak časové, tak finanční) obrovské. Architektura je totiž základním stavebním kamenem celého systému. Pokud musíte v průběhu stavby domu změnit základy, je to změna vyžadující změnu všech vašich dalších plánů. Návrh architektury si tedy zaslouží velkou pozornost. Architektura systému musí splňovat veškeré požadavky na ní kladené a toto musí být možno jasně

prokázat. Navrhnout kvalitní architekturu splňující ty nejnáročnější požadavky je netriviální a změnám se v průběhu vývoje nejspíše nikdy nevyhneme.

2.1.1 Požadavky na architekturu

Prvně architektura musí co nejvíce oddělit herní logiku od enginu. Jednotlivé komponenty by mělo být možné vyvíjet a testovat nezávisle na sobě. Celý engine by měl být funkční i bez některých komponent. Čili lze tyto komponenty měnit nezávisle na ostatních. Jednotlivé domény enginu je důležité co nejvíce oddělit: grafika, fyzika, umělá inteligence, zvuk, síť atd. Díky tomu se pak jednotliví vývojáři mohou plně specializovat pouze na svoji doménu znalostí. Vzhledem k náročnosti to ani jinak zvládnout nelze. Neméně důležitá je i flexibilita. Znovupoužití enginu si tedy zcela jistě vyžádá úpravy. Kritickým požadavkem je rozšiřitelnost původní verze softwaru. To opět úzce souvisí se opakovaným použitím enginu pro vývoj dalších her. Každá další hra si určitě vyžádá implementaci nových myšlenek a funkčností. Nové funkce enginu by měly být integrovatelné s minimálními změnami systému a to pouze na té nejvyšší úrovni. Zásahy do architektury jsou absolutně vyloučeny. Navíc musí zůstat zachována předchozí funkčnost beze změn. To se týká i dalších vlastností enginu. Kvalitní engine je i po přidání nových možností stále stabilní, stejně efektivní. Shrňme základní požadavky:

- Čisté oddělení herní logiky od enginu
- Co největší nezávislost jednotlivých komponent enginu
- Flexibilita
- Rozšiřitelnost

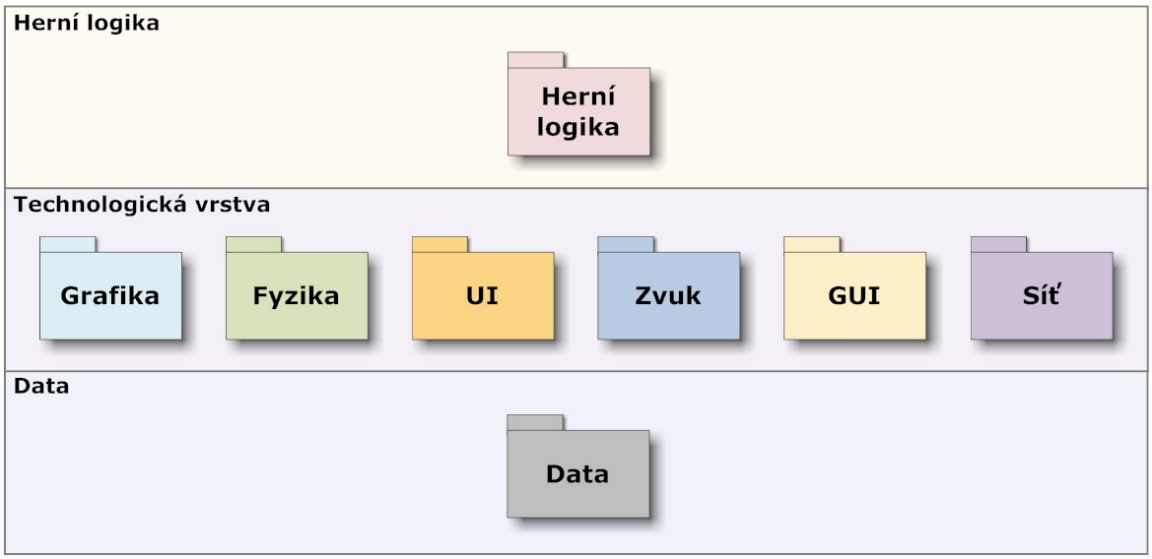
Mohli bychom se ptát, proč není mezi požadavky uvedena efektivita a další kvalitativní vlastnosti. Efektivita a kvalita, ale opravdu nejsou požadavky, pokud se o hry jedná. Samozřejmě se je na dodržení těchto základních vlastností každého dobrého enginu kladen velký důraz, ale jsou tak naprosto zřejmé, že jsem je mezi požadavky na architekturu engine nezařadil. V praxi jistě nevyužijeme engine, který je sice stabilní, perfektně rozšiřitelný a upravitelný, ale neposkytuje dostatečný výkon.

2.1.2 Kandidáti pro architekturu enginu

Z abstraktního pohledu je každá architektura založena na jednotlivých komponentách. Ty mají svá specifická omezení stejně tak komunikace mezi nimi. Zbývající část této podkapitoly byla převzata z [PLUMMER04].

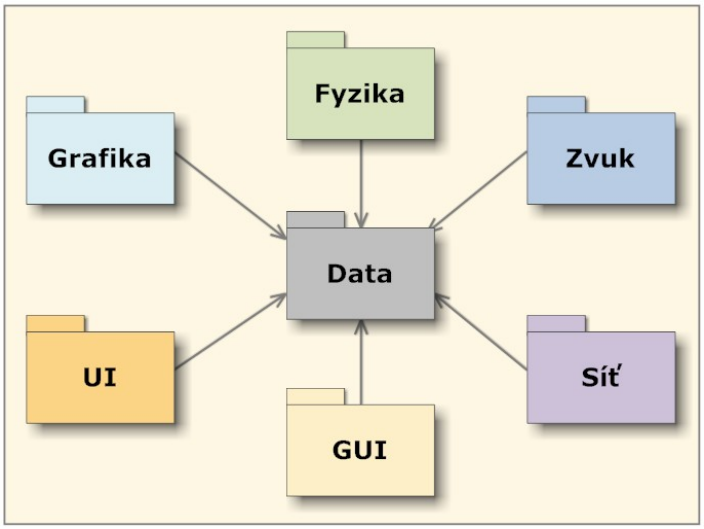
Vícevrstevná architektura rozděluje funkcionalitu systému do logicky oddělených vrstev. Každá vrstva pak slouží jako služba pro vrstvu nad ní. Znovupoužití kódu je zde docíleno tím, že je

vlastní logika aplikace umístěna v nejvyšší vrstvě. Nižší vrstvy tak lze použít pro vývoj nové aplikace. Díky vrstvám je splněn také požadavek na modularitu systému. Schéma architektury vidíme na obrázku 2.2 – Vícevrstevná architektura.



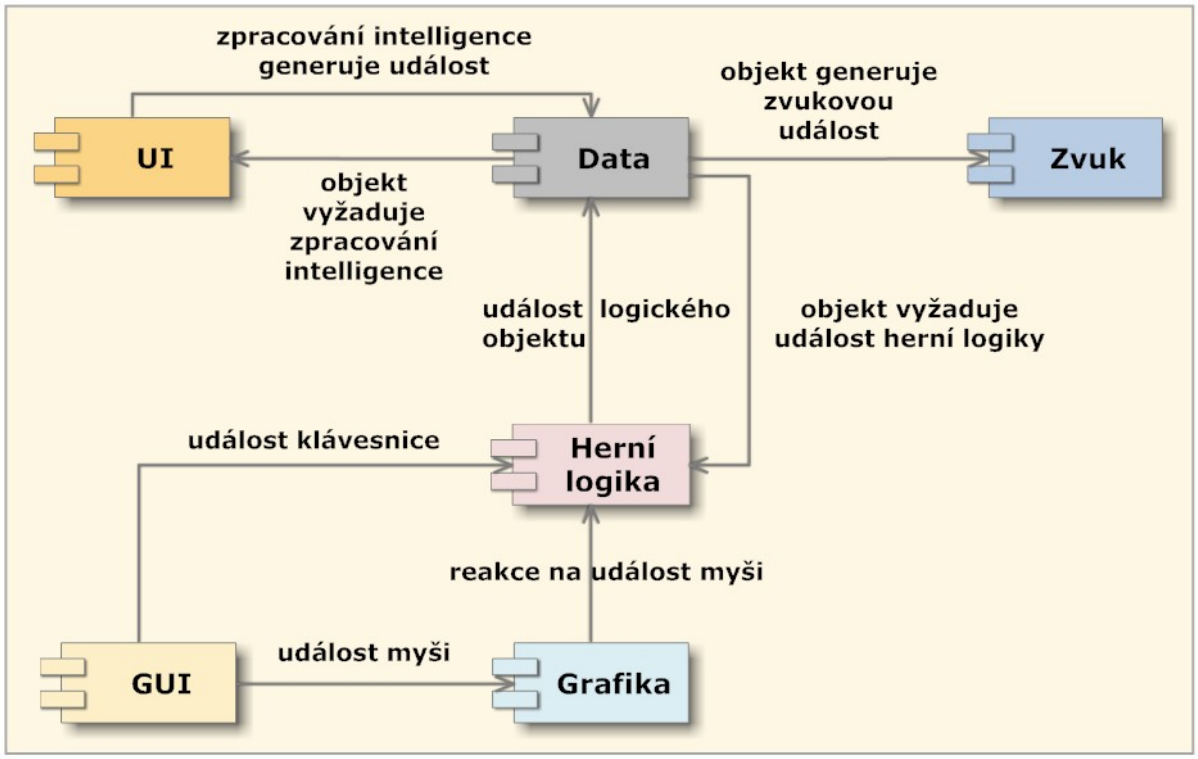
Obrázek 2.2 - Vícevrstvá architektura

Dalším kandidátem je datově centralizovaná architektura. Jedná se o architekturu s centrálním datovým skladem (úložištěm), nad nímž operují klientské moduly. Hlavní výhodou takto navržené architektury enginu je nezávislost jednotlivých klientů. Integrace nového klienta je pak velmi praktická, nenáročná a nemá vliv na ostatní klienty. Schéma architektury vidíme na obrázku 2.3 – Datově centralizovaná architektura.



Obrázek 2.3 - Datově centralizovaná architektura

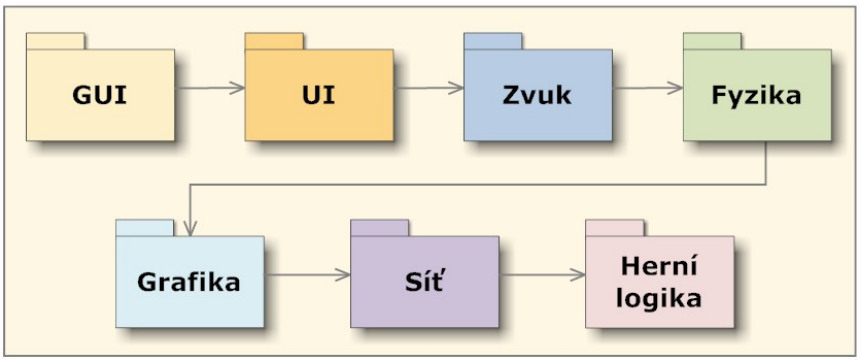
Nezávislé procesy komunikující mezi sebou pomocí mechanismu zasílání zpráv je definice architektury nezávislých komponent. Jednotlivé komponenty mají registrovány zprávy, které chtějí přijímat a zpracovávat. Velkou výhodou je, že systém může fungovat s prakticky libovolným počtem komponent. Správné užití této architektury pak dovoluje systému libovolně přidávat nebo odebírat funkcionalitu. Nevýhodou je velká režie spojená s komunikací komponent pomocí zasílání zpráv. Schéma architektury vidíme na obrázku 2.4 – Architektura nezávislých systémů.



Obrázek 2.4 - Architektura nezávislých systémů

Data Flow architekturu tvoří v podstatě „roura“, kterou procházejí zpracovávaná data. V každém spoji roury je umístěn blok pro transformaci vstupních dat na výstupní. Výhodou je jistě to, že výkonné bloky mohou být velmi elementární. Takový systém lze pohodlně upravovat a jednoduše pochopit. Dále je možné velmi efektivně přidávat další výpočetní stupně. Tím dosáhneme snadné rozšiřitelnosti systému. Při větším zásahu do systému lze spatřit některé nevýhody. Nelze jednoduše změnit pouze jednu komponentu a dosáhnout nového výsledku. Taková změna často vyžaduje změnu i v jiných závislých komponentách.

Schéma architektury vidíme na obrázku 2.5 – Data flow architektura.



Obrázek 2.5 - Data Flow architektura

Cílem architektury systému systémů je efektivně integrovat množství komplexních systémů. Podnětem je existence systémů zcela odlišných domén pracujících jako jeden celek. Např. grafika, fyzika, zvuk apod. jsou naprosto rozdílné domény, které vyžadují vlastní přístup. Ve výsledku však musí být schopny pracovat společně. Výhoda je zde zřejmá. Lze velmi efektivně oddělit doménové znalosti a vyvíjet jednotlivé systémy nezávisle na sobě. Nevýhodou je opět komunikace mezi systémy pomocí mechanismu zasílání zpráv.

Je zřejmé, že žádná z nabízených architektur nebude plně uspokojovat vysoké nároky na moderní enginy. V konečné fázi vývoje tak bude použito hybridní architektury, který bude využívat předností dostupných architektur. Jde právě o to, jak vyvážit všechny požadavky a zajistit jejich dostatečné uspokojení. Zde vzniká obrovský prostor pro optimalizace všech funkčních jednotek systému.

2.2 Logické moduly moderních enginů

Jak jsme si již řekli v požadavcích na architekturu, je oddělení jednotlivých funkčních domén jedním z důležitých požadavků na herní engine. Analýzou případů užití jakékoli moderní počítačové hry můžeme snadno identifikovat ty nejzákladnější. V následujícím textu si je přiblížíme.

2.2.1 Scénový graf

Ty tam jsou doby, kdy počet objektů ve scéně nepřekračoval řád desítek. Dnešní virtuální světy obsahují tisíce objektů. Uložení herních objektů do pole a následné sekvenční procházení všech jeho prvků je dnes naprosto neefektivní a nedostačující, ale zároveň jednoduchá metoda, jak vykreslit výslednou scénu. Hlavní nevýhodou tohoto přístupu bylo vykreslování všech objektů dané scény bez

ohledu na to, zda je daný objekt viditelný z hráčovy perspektivy. Řešením je právě graf scény. Graf scény je n-rozměrná stromová struktura nebo acyklický graf. Každý rodičovský uzel může obsahovat nula až n potomků. Transformační matici nese buď každý uzel, nebo speciální transformační uzel. Podle této matice jsou počítány aktuální transformace všech potomků [SCENEGRAPH10]. Každý uzel dále obsahuje obalové těleso. Jedná se o analyticky vyjádřený geometrický útvar nejčastěji osově zarovnaný kvádr. Všichni potomci daného rodičovského uzlu, spadají svou aktuální pozicí do obalového tělesa rodiče. Díky tomu se minimalizuje počet testovaných objektů na viditelnost vůči kameře. Pokud je totiž rodičovský uzel v záběru kamery, pak i všichni jeho potomci jsou v dané scéně viditelní a nemusí se na viditelnost testovat. Tento přístup k určování viditelnosti objektů předpokládá velký počet statických objektů, které nemění svoji pozici v herním světě [EBERLY05]. Velmi dynamické scény vyžadují buď reorganizaci grafu, nebo lze použít známé stromové algoritmy pro dělení prostoru jako např. BSP strom nebo quad strom.

Vnitřní uzly mohou mít mnoho dalších použití. Mezi nejpoužívanější patří výběrový uzel sloužící k výběru potomka. Uplatnění najde např. při výběru úrovně detailu zobrazovaného objektu. Dále je možno používat událostní uzly, které generují nějakou definovanou událost při průchodu uzlem.

Geometrická data uchovávají listové uzly. Spolu s daty o geometrii objektu obsahuje listový uzel další potřebná data pro vykreslení, jako jsou textury a materiály. Obrovský prostor pro optimalizace lze nalézt v procesu zpracování grafu. Je nutné aktualizovat všechny transformační matice, přepočítat obalová tělesa a udržet tak strom konzistentní.

Graf scény je velmi podstatnou součástí enginu. Má velký vliv na výkon a jeho možnosti. Navrhnout a optimalizovat graf scény je opět netriviální.

2.2.2 Grafický modul

Hlavním posláním grafického modulu je bezpochyby vykreslení určité scény. Dnešní enginy využívají pro vykreslení scény výhradně grafické akcelerátory. Komerční engine, který by ještě dnes poskytoval možnost softwarového vykreslování scény, určitě nenajdete. K samotnému vykreslování dat slouží shadery. Enginy dnes ani neumožňují použití fixní grafické pipeline. Grafický modul musí dostatečně rychle zásobovat grafický procesor daty a spravuje jeho nastavení. Asi nejdůležitějšími požadavky jsou výsledná kvalita obrazu společně s dostatečnou rychlostí zpracování a maximálním vytížením grafického akcelerátoru. Aby byly hardwarové zdroje efektivně využity, je nutné zasílat pouze data, která se budou skutečně vykreslovat. Pak je možné aplikovat výpočetně náročnější shader programy a docílit tak vyšší kvality obrazu.

Renderer je zodpovědný za vykreslení výsledného obrazu scény. Vstupem rendereru je obvykle kamera a graf scény. Princip vykreslování je následující [EBERLY05]:

- Urči viditelné objekty
- Seřaď (viditelné) objekty
- Pro každé světlo vykresli ovlivněné objekty
- Seřaď a vykresli průhledné objekty
- Aplikuj následné zpracování výsledného obrazu (post-processing)

Kromě dělení scény pro určení viditelných objektů (viz. 2.2.1 Scénový graf) využívají moderní enginy také occlusion culling neboli odstranění objektů zastíněných jiným objektem. Seřazením viditelných objektů chceme docílit co nejméně častých změn v nastavení stavů grafického procesoru. Pro každé světlo většinou generujeme v prvním průchodu stínové mapy. Ve druhém průchodu pak vykreslujeme objekty i s texturami a materiály (shadery). Průhledné objekty musíme nejprve seřadit dle vzdálenosti od kamery, aby byl výsledek korektní. Následné zpracování výsledného obrazu aplikuje (pomocí obrazových shaderů) efekty jako např. hloubku pole (depth of field), rozmazání rychlých pohybů (motion blur) nebo žár (glow). Celý proces je znám jako dopředné vykreslování (forward rendering).

Dopředné vykreslování trpí hlavně složitou správou scény. Je nutné mnohanásobné procházení grafu a opakování již provedených operací jako výpočet transformací, texturové filtrace, dekomprese normálových map atd. Tyto neduhy odstraňuje technika odloženého vykreslování (deferred rendering). V průběhu odloženého vykreslování se nemusíme starat o žádné osvětlování. Technika nejčastěji využívá MRT (multiple rendering target). Normály, atributy osvětlení, pozice vertexů atd. ukládáme do separátních map v jednom průchodu scénou. Nad všemi mapami pak aplikujeme osvětlovací shader [SHISHKOVTSOV05]. Výhod je hned několik. Hlavně, všechny výpočty jsou prováděny v obrazovém prostoru. Výsledkem je stálý výkon a prakticky žádné omezení v počtu světel. Zpracování průhlednosti zůstává většinou podobné dopřednému vykreslování. Technika je zatím využívána krátce (nápad pochází z počátku devadesátých let) a podporuje ji jen několik herních enginů. Její výhody jsou ale značné. V budoucnu se s ní budeme určitě stále více setkávat. Ukázku vidíme na obrázku 2.6 – Deffered rendering (CryEngine 3).

Obrázek 2.6 - Deffered rendering (CryEngine 3)

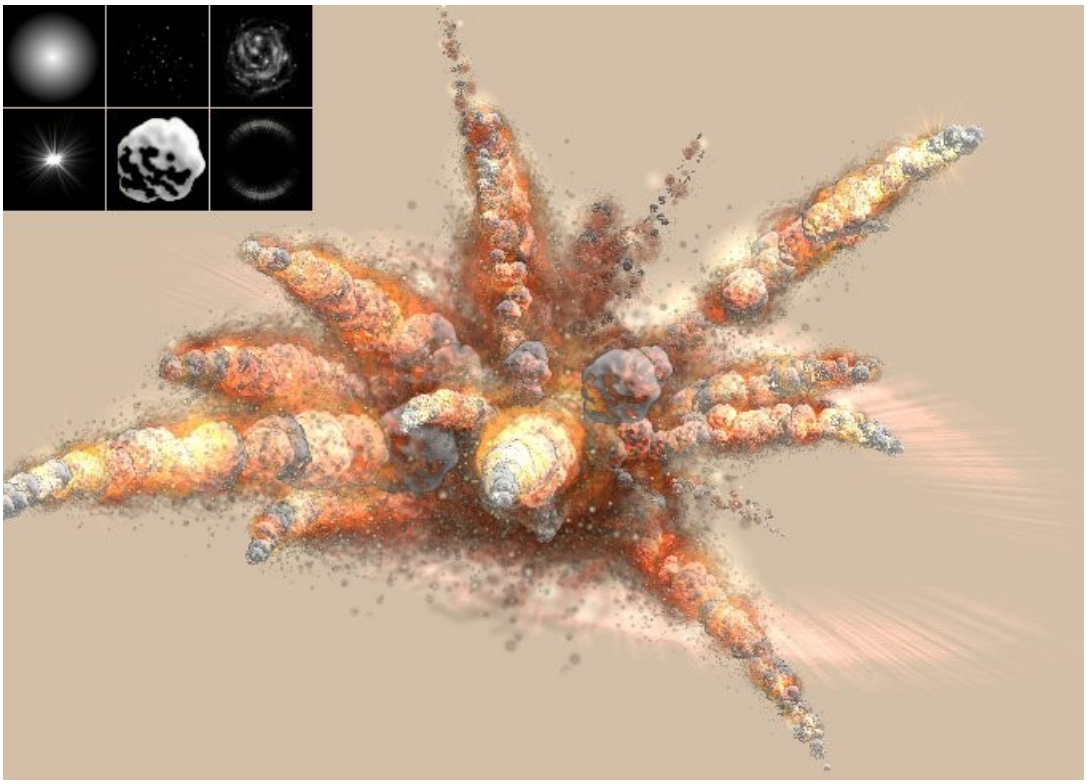
Pro implementaci dnešních rendererů lze prakticky použít dvě API. OpenGL a DirectX. OpenGL zaostává ve vývoji, avšak pokud plánujete aplikaci použitelnou na více platformách, je volba OpenGL nutností. V API DirectX firmy Microsoft se mnohem flexibilněji projevují nové možnosti grafických procesorů. Navíc díky těsné vazbě DirectX na operační systém (pouze platforma Windows), může API poskytnout oproti OpenGL také funkcionalitu pro vytváření okna aplikace, zpracování vstupních událostí apod. I vzhledem k rozšířenosti platformy Windows je dnes právě DirectX nejčastější volbou pro implementaci rendereru.

2.2.3 Částicový systém

Částicový systém je nedílnou součástí každého moderního enginu. Jeho uplatnění je velice široké. Od ohně, exploze a tornáda přes vodu až po různé „magické“ efekty. Systémy částic používané v dnešních herních enginech jsou velice robustní a lze nastavit jakýkoli myslitelný parametr pro dosažení co nejširší škály efektů [PARSYSTEM10].

Efekt tvoří velké množství jednotlivých částic. Počet se obvykle pohybuje od desítek do tisíců částic. Každá částice má svoji hmotnost, velikost, vektor aktuální rychlosti, rotaci a další parametry. Dnešní částicové systémy počítají i s vlivem prostředí na systém např. větrem. Toho je dosahováno pomocí silových polí, které působí na všechny částice systému. Nejzákladnějším silovým polem je gravitace, která uděluje každé částici konstantní zrychlení. Při dnešním výkonu hardwaru přicházejí ke slovu také síly působící mezi jednotlivými částicemi a mezi částicemi a okolím. Pro obrovský

počet částic je nemožné tyto výpočty provádět na centrálním procesoru. Výpočet se umísťuje na dnešní ultra výkonné grafické procesory a to buď formou specializovaných shaderů [LATTA04] nebo knihoven pro výpočet fyziky na GPU [PHYSX08].



Obrázek 2.7 - Částicový systém v akci

Pro vykreslení se používají hlavně techniky založené na point-sprite metodě známé i jako billboarding. I dnešní technologicky nejvyspělejší herní enginy využívají čtverce natočené kolmo k pozorovateli pro vykreslení částicových efektů jako kouř, oheň, dynamická mlha nebo déšť. Důvodem je vysoký výkon a snadnost implementace. Díky tomu, že jsou stále natočeny kolmo k pozorovateli, dokáží 2D plochy spolu s alfa mícháním vytvořit iluzi substance vyplňující 3D prostor. Kvůli průhlednosti je nutné před odesláním částic grafické kartě částice nejdříve seřadit podle vzdálenosti od kamery. Pro generování ploch s výhodou využijeme geometry shader programu, který čtverec vygeneruje přímo na grafickém procesoru. GPU jsou pak procesorem zasílány pouze aktuální pozice částic. Technika point-spritů s sebou naproti rychlosti přináší, jako vždy, některé problémy. Mezi ty patří např. odstranění ostré hrany při kontaktu částice s okolím (viz. kapitola 3.4.1 Animated Depth Soft particles)

Částice můžeme také vykreslit jako tzv. methaball čili v podstatě koule. Následně aplikujeme filtry, které povrch rozmažou a vytvoří tak celistvý povrch. Této techniky se využívá především při vykreslování tekutin.

Obecně je vždy problémem osvětlení a stínování částicového efektu. Dalšími technikami pro vykreslení efektu jsou raycasting, splatting nebo marching cubes. Tyto techniky se však vyskytují v oblasti herních enginů zatím pouze sporadicky. Neposkytují totiž požadovanou rychlost výpočtu.

2.2.4 Fyzikální modul

Vývoj fyzikálního systému je nesmírně náročný. Herní enginy proto integrují již existující fyzikální enginy namísto tvorby vlastního nového systému. Nejpoužívanější jsou fyzikální enginy Havok [HAVOK10] a NVIDIA PhysX [PHYSX08]. Oba produkty nabízejí detekci kolizí, simulaci pevných těles, tekutin (fluidů), silových polí, spojů (omezení), tkanin, vozidel, deformací a v neposlední řadě simulaci postav. Animátorům a programátorům oba produkty nabízejí pro tvorbu fyzikálního popisu scény a ladící prostředky pro vizualizaci a analýzu výkonu. Škála fyzikálních efektů je limitována výkonem hardware a potažmo naší představivostí.

Výhodou PhysX oproti Havok enginu je volná dostupnost binární verze API a především možnost GPU akcelerace implementovaná na platformě CUDA [PHYSX08]. Výhodu GPU akcelerace bohužel v současné době limituje fakt, že je dostupná pouze na grafických kartách firmy NVIDIA. To zamezuje masivnímu nasazení PhysX enginu ve hrách. Herní studia si nemohou dovolit vytvářet hry pouze pro NVIDIA GPU. PhysX je tak používán jen pro extra efekty. Čas ukáže, jestli se dočkáme podpory GPU akcelerace PhysX i na kartách firmy AMD/ATI nebo zda výpočet zůstane na bedrech CPU. Dle mého názoru má před sebou GPU akcelerace fyzikální simulace ve hrách velkou budoucnost.

2.2.5 Správce zdrojů

Správce zdrojů je zodpovědný za nahrávání potřebných grafických dat do systému a za jejich odstranění pokud již nejsou zapotřebí – nepoužívají se. Jedná se o geometrická data modelů, textury, animace apod. Jde o velké objemy dat. Správce musí být schopen v rozumném čase tyto data nahrát z pevného disku. Nahrání dat bylo dříve součástí fáze nahrávání části herního světa - úrovně. Jakmile hráč splnil všechny podmínky definované hrou pro postup do další úrovně, byly nepotřebné zdroje odstraněny a nahrazeny novými. Hra tak po jistou dobu nemohla kontrolovat pozornost hráče. Pokud navíc hráč vlastnil méně výkonný počítač, stával se přechod hry z jedné úrovně do druhé značně zdlouhavý a někteří hráči prostě nevydrželi čekat a hru raději nehráli. Při dnešním výkonu počítačů je snaha tento nepříjemný proces odstranit [DICKHEISER06].

Data potřebná pro herní svět jsou nahrávána za běhu aplikace. Jakmile hráč vstoupí do další sekce, tak již nevidí starou známou obrazovku „Loading“, ale data jsou průběžně odstraňována a nahrazována novými. Hra pak hráči nedá vydechnout a umožní větší požitek ze hry. Hra v jistém

okamžiku předpokládá, že se hráč bude dále pohybovat jistým směrem. Nahraje tedy data dopředu ještě před tím, než budou použita. Systém musí zajistit, aby nově požadovaná data byla co nejdříve k dispozici. Výběr zdroje k odstranění (při nedostatku paměti) řídí politika správce. Nejčastěji se odstraní zdroj, který byl nejdéle nepoužitý (LRU). Samotné nahrávání zdrojů probíhá v odděleném vlákně. Zdroje musíme priorizovat, protože např. terén musí být nahrán jistě dříve než třeba „bezvýznamný“ detail domu v pozadí, aby hráč nepostřehl spuštěný proces nahrávání dat [MCSHAFFRY03].

Přítomnost zdroje ve správci můžeme buď při každém požadavku klasicky otestovat, nebo lze v případě nepřítomnosti poskytnout aplikaci dočasný zdroj (např. textura se čtyřmi zcela průhlednými pixely). Po dokončení vlastního nahrávání tento dočasný zdroj nahradíme skutečným a neustálé testování přítomnosti tak zcela odpadne, protože zdroj je vždy k dispozici [KIRMSE04].

2.2.6 Správa paměti

Správa paměti je důležitou součástí moderního enginu. Má zásadní vliv na celkovou rychlost systému. Pokud např. vaše vesmírná loď právě zničila nepřátelský cíl, budete očekávat mohutnou explozi. Ta je realizována vlastním částicovým systémem. Pro jeho vytvoření je nutno alokovat paměť pro všechny částice systému. Pokud tuto činnost ponecháme na starost operačnímu systému, použije se systém virtuální paměti a mechanismus stránkování. To zahrnuje nalezení bloku paměti o požadované velikosti a další, z pohledu engine, časově náročné operace. Vytváření velkého počtu malých bloků paměti vede také na nepříjemnou fragmentaci paměti. Protože jsou v každém snímku aktualizovány stavy všech instancí, je výpadek stránky při získávání přístupu k instanci běžnou záležitostí. Výkon celého systému rapidně klesá.

Základní myšlenka řešení spočívá v alokaci velkého bloku paměti pro hlavní objekty předem (při inicializaci systému). Vytvořený úsek paměti je spravován paměťovým správcem. Každá operace `new` pak vyvolá pouze nalezení volné položky v alokovaném bloku [KIRMSE04]. Úspora výpočetního času je značná. Dnešní enginy používají velice sofistikované paměťové správce, které optimalizují využití paměti jak operační, tak paměti grafického akcelerátoru [DELOURA01]. Na něj mohou být nahrána např. geometrická data, o kterých víme, že budou použita pro každý snímek. Tím se dále snižuje i množství dat přenášených mezi centrálním procesorem a grafickým procesorem [WRIGHT–LIPCHAK05].

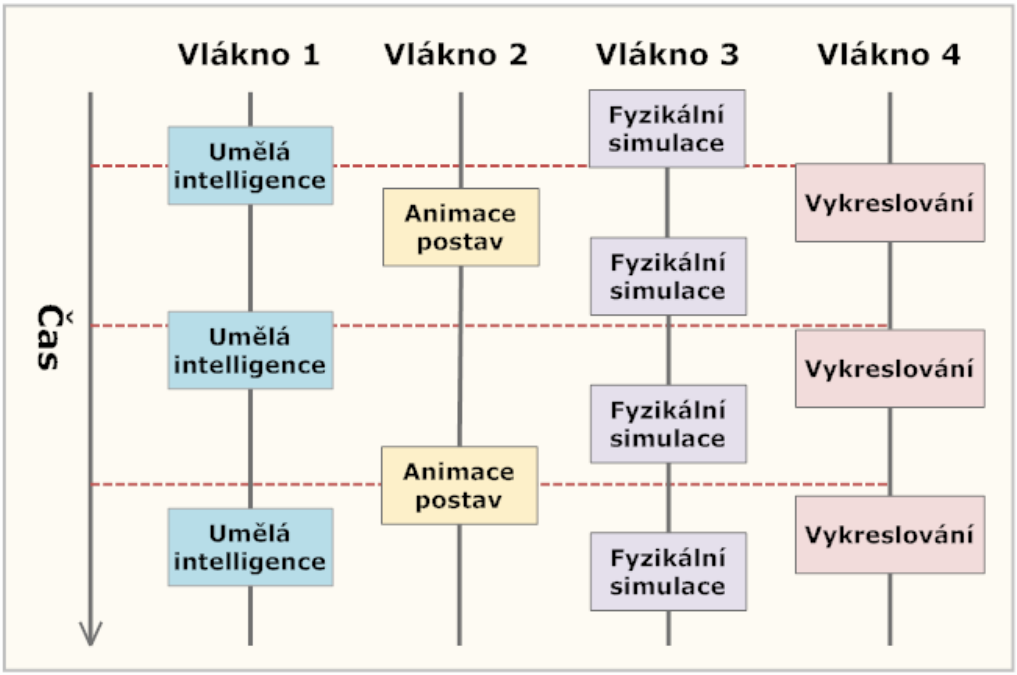
Se správou paměti souvisí i životní cyklus objektů. Objekt, který již není využíván, je důležité z paměti odstranit. Pro tyto potřeby se téměř výhradně užívá tzv. chytrých ukazatelů (smart pointer). Pro tento model je zapotřebí správně navrhnout objektový systém. Většinou se volí struktura s jedním předkem pro dosažení maximálního výkonu [EBERLY05].

2.2.7 Herní smyčka

Každý herní engine takovou smyčku obsahuje. Smyčka je jádrem systému, ve kterém probíhají všechny výpočty. Po inicializaci enginu je tato smyčka spuštěna. Provádí aktualizace všech vnitřních částí enginu. Výsledkem je vykreslený obraz dle aktuálních hodnot herních objektů.

S příchodem vícejádrových centrálních procesorů vyvstala potřeba využití jejich výkonu. Použití vícevláknového zpracování zásadním způsobem ovlivní návrh celého systému. Je zapotřebí přizpůsobit aktualizační proces enginu. Základní myšlenkou je spouštět samostatná vlákna pro jednotlivé moduly enginu. Problémem je docílit maximálního vytížení všech jader procesoru a co nejmenší závislosti mezi jednotlivými moduly. Synchronizace totiž znamená čekání.

Opět existuje mnoho řešení těchto problémů. Jedno z nich vidíme na obrázku 2.8 – Možné řešení vícevláknového zpracování.



Obrázek 2.8 - Možné řešení vícevláknového zpracování

V tomto modelu jsou moduly umělé inteligence, animace postavy, fyzikální simulace a proces vykreslování mapovány do oddělených vláken. Každé vlákno provádí své výpočty z posledních známých výsledků. Každý model je tedy aktualizován s jinou frekvencí. Implementace tohoto modelu je relativně jednoduchá, přesto poskytne výrazné navýšení výkonu při použití vícejádrového procesoru [GABB–LAKE05].

3 C3D Engine

Zvolení jména enginu je příjemný proces. Jde o ryze český produkt, dovolil jsem si tedy použít diakritiku. Engine dostal název Čočkus3D, zkráceně C3D.

Celý projekt je implementován v jazyce C++ a využívá řady knihoven. Všechny knihovny jsou multiplatformní a volně dostupné jak pro komerční tak nekomerční vývoj. Jejich krátký popis s odkazy na internet naleznete v příloze č.3. Na přiloženém CD naleznete programovou dokumentaci, která může posloužit pro rychlé nahlížení do popisované implementace.

Hierarchie dědičnosti enginu má jediného předka a tím je třída `RCObject`. Třída `RCObject` implementuje podporu pro chytré ukazatele pomocí knihovny `boost::intrusive_ptr`. Každý objekt enginu je tedy chytrým ukazatelem. Vedle usnadnění práce s pamětí, poskytuje tento model také implicitní sdílení dat, které využijeme pro geometrická data, textury a jiné velké objemy dat.

3.1 Scénový graf

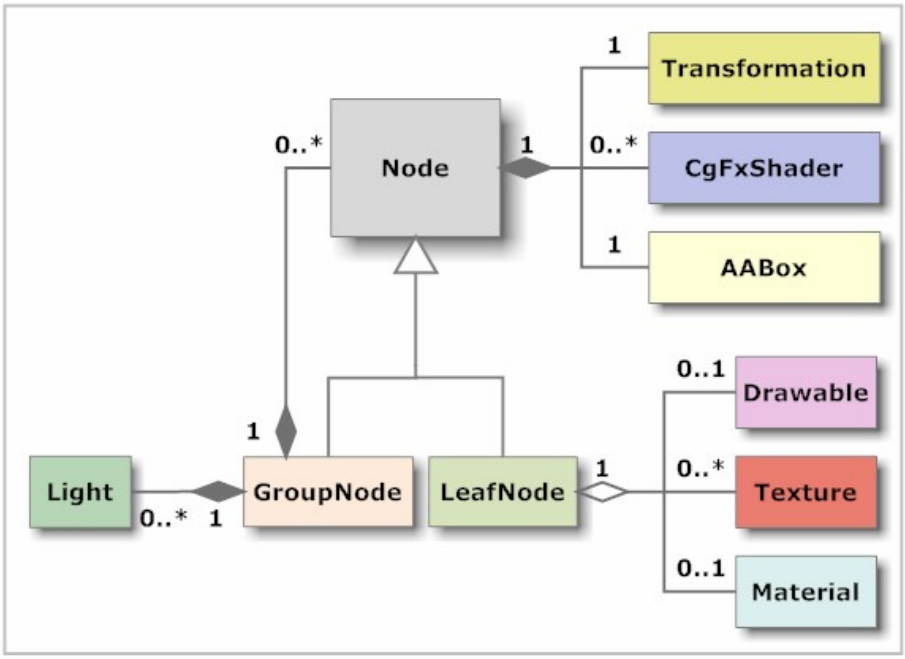
Scénový graf tvoří základní kámen celého C3D enginu. Uchovává veškeré informace nutné pro výslednou scénu. Každé rozhodnutí v době návrhu má přímý vliv na možnosti implementace dalších vlastností enginu a stává se tak klíčovou záležitostí. Tvorba kvalitního návrhu je velmi obtížná a vyžaduje velkou dávku zkušeností, jak jsem se sám přesvědčil. Během vývoje projektu byl design grafu několikrát upraven, což přineslo další časově velmi náročné úpravy. Pojďme se nyní podívat na finální návrh podrobně.

3.1.1 Návrh

Návrh si ukážeme na obrázku 3.1 – Diagram tříd scénového grafu. Vychází z kompozitního návrhového vzoru (Composite) [GAMMA95]. Tento vzor umožňuje vytváření n-ární stromové struktury, což přesně odpovídá definici struktury scénového grafu.

Každý uzel obsahuje čistý (raw) ukazatel na svého předka, aby bylo možné procházet graf i ze zdola nahoru. Ukazatel musí být čistý, jinak by došlo k cyklické závislosti. Každý uzel také obsahuje obalovací těleso ve formě osově zarovnaného kvádru. Dále musíme rozhodnout, kde budou uloženy transformace. Máme dvě možnosti. První je uložit informace o transformacích do speciálního typu uzlu např. `TransformationNode`, který by dědil z `GroupNode` . Druhou možností je uchovávat transformace v každém uzlu grafu. Oba přístupy jsou takřka rovnocenné a umožňují vystavět sémanticky shodný strom. Nevýhodou druhého způsobu je vyšší paměťová náročnost. Všechny typy

odvozené od třídy `Node`, budou obsahovat informace o transformacích a to i v případě, že pro ně nebudou relevantní, tj. nebudou nikdy použity. Příkladem takového typu uzlu je např. uzel pro výběr vhodné úrovně detailů. I přes vyšší paměťovou náročnost byl zvolen tento druhý přístup. Tedy uchování transformací v každém uzlu. Transformace zapouzdřuje třída Transformation implementovaná pomocí knihovny GMTL (viz. příloha č.3).



Obrázek 3.1 – Diagram tříd scénového grafu

Světla jsou uložena ve skupinovém uzlu (`GroupNode`) a ovlivňují vždy podgraf daného uzlu. Stejnou sémantiku mají i shadery. Narozdíl od světel je můžeme přiřadit všem typům uzlu grafu.

Třída `LeafNode` reprezentuje listový uzel grafu. Uchovává pole textur, abychom byly schopni využívat technik jako např. normálového mapování nebo multi-texturování. Listový uzel obsahuje maximálně jeden materiál a konkrétní reprezentaci geometrických dat.

3.1.2 Zpracování

Nad grafem scény jsou prováděny různé operace. Jedná se o vykreslování scény, aktualizace grafu dle výsledků fyzikálních výpočtů, aktualizace nejrůznějších stavů atp. Jaké jsou tedy požadavky návrhu?

Efektivitu musíme vyřadit. Již z povahy projektu, tedy aplikace počítané v reálném čase, jasně plyne, že efektivní zpracování není požadavkem, ale nutnou vlastností návrhu. Skutečným požadavkem zůstává rozšiřitelnost. Naprosto běžným příkladem, dokládajícím potřebu snadné rozšiřitelnosti, může být implementace vykreslování scény pomocí jiného grafického API. V tomto případě budeme chtít přidat nový způsob vykreslování scény, ale zároveň chceme zachovat původní

ideálně bez jakýchkoliv změn ve starém kódu. Uživatel si tak bude moci vybrat jaké API má být použito pro vykreslování.

Nejjednodušší řešení je implementovat funkčnost pomocí virtuálních funkcí umístěných v uzlech scénového grafu.

```
class C3D_API Node : public Object
{
//...
public:
    virtual void RenderOpenGL()=0;
    virtual void RenderDirectX()=0;
    virtual void UpdatePhysics()=0;
    virtual void Update(float fDelta)=0;
//...
};
```

Toto řešení je naprosto nepoužitelné. Přidání dalšího zpracování grafu si vynucuje změnu ve všech třídách scénového grafu. Navíc nelze v průběhu zpracování grafu ukládat žádné dočasné informace platné pro více uzlů grafu. Vhodnějším řešením je oddělit informace od operací [EBERLY05]. Takto:

```
class C3D_API Node : public Object
{
//...
public:
    virtual void Render(IRenderer* pRenderer)=0;
    virtual void UpdatePhysics(IPhysicsSystem* pPhysics)=0;
    virtual void Update(float fDelta)=0;
//...
};

class IRenderer : public Object
{
//...
public:
    virtual void RenderGroupNode(GroupNode* pGroupNode)=0;
//...
};
```

Implementace by pak vypadala následovně:

```
void GroupNode::Render(IRenderer* pRenderer)
{
    pRenderer->RenderGroupNode( this );
};
```

Analyzujme. Díky oddělení informace od operace dostáváme daleko větší možnosti rozšíření funkcionality. Není již nutné měnit deklarace všech tříd scénového grafu za účelem přidání nové možnosti vykreslení scény. Problémem však stále zůstává přidání naprosto nové operace nad grafem, se kterou nebylo nebo ani nemohlo být v původním návrhu počítáno. Např.: `UpdateSound()`, která by zpracovala aktualizaci zvukové složky. Dále pak v porovnání s první verzí dochází k větší režii při

zpracování. Každá jednotlivá operace nad konkrétní instancí uzlu zahrnuje volání dvou virtuálních metod v porovnání s jednou v předešlé verzi. Uvážíme-li dále velký počet typů uzlů grafu a v praxi často obrovský počet instancí uzlů (i tisíce), může mít tato režie neblahý dopad na výkon systému[1]. Tento návrh tedy také není přesně to, co hledáme.

Hlavním problém předchozích návrhů je neznalost typu uzlu v době běhu aplikace. Tuto chybějící informaci nahrazujeme použitím virtuálních metod. Kdybychom dokázali efektivně dynamicky určit typ zpracovávaného uzlu grafu, bude návrh splňovat veškeré požadavky. Operace nad grafem by pak byly prováděny velmi jednoduše. Stačí získat typ uzlu a vyvolat příslušné zpracování. Oddělení informace od operace bude pak optimální. Ve třídách uzlů se již nebudou vyskytovat žádné metody spojené s operacemi nad grafem. Tedy žádný `Update()`, `Render()`, atp. Jak ale určit typ uzlu? C++ toto nativně neumožňuje. Můžeme implementovat RTTI systém. Ten nám poskytne textové názvy typů jednotlivých tříd scénového grafu a dovoluje nám zjistit i jméno třídy, ze které je daný typ odvozen [EBERLY05]. Takový systém nabízí až příliš informací a navíc v řetězcovém formátu. Použití by vedlo k neustálému porovnávání řetězců, což výrazně snižuje výkon. Pro naše účely stačí uchovat pouze číselnou reprezentaci typu objektu, která bude unikátní v rámci celého systému. Nazvěme ji objektovým kódem (`ObjectCode`).

```
enum ObjectCode
{
    OC_INVALID = 0xFFFFFFFF
    , OC_OBJECT = OC_INVALID
    , OC_NODE = OC_INVALID
    , OC_GROUPNODE = 0
    , OC_LEAFNODE
    //...
};
```

Objektový kód uložíme do každého objektu systému.

```
class C3D_API Object : public Utils::RCObject
{
public:
    Object() : Utils::RCObject() { m_uiObjectCode = OC_OBJECT; }
    /// Get object type unique code.
    C3D_FORCE_INLINE unsigned int GetObjectCode()
                                                { return m_uiObjectCode; }
    //...
protected:
    unsigned int m_uiObjectCode;
};
```

[1] Pro proces vykreslování toto jistě není problém, protože samotné vykreslování bude jistě potřebovat nepoměrně více procesorového času. Pro procesy jako je například pouhá aktualizace transformací po dokončení fyzikální simulace už režie spojená s velkou virtuální tabulkou hraje svoji roli.

Třídu, jež bude definovat operaci nad grafem, pojmenujme Traverserem (třída `Traverser`). Její členské metody budou implementovat operaci nad konkrétními typy uzlů. Třída implementuje návrhový vzor Visitor [GAMMA95].

```
class C3D_API Traverser : public Object
{
public:
    Traverser();
    ~Traverser()=0;
    //...
    virtual void Apply()=0;
    void SetRoot(NodePtr pNode);
    //...
protected:
    void TraverseGroup(GroupNodePtr pGroup);
    void TraverseLeaf(LeafNodePtr pLeaf);
    //...
};
```

Neustále však musíme porovnávat, tentokrát místo řetězců celá čísla. Efektivita bude sice lepší, ale stále nedostačující. V tuto chvíli nám pomohou ukazatele na funkce. Pole ukazatelů můžeme v konstruktoru třídy `Traverser` nastavit na členské metody.

```
Traverser::Traverser(void) : Object()
{
    RegisterCallback(OC_GROUPNODE, &Traverser::TraverseGroup);
    RegisterCallback(OC_LEAFNODE, &Traverser::TraverseLeaf);
    //...
};
```

Při procházení grafu scény `ObjectCode` poslouží jako index do pole ukazatelů a vyvolá tak operaci nad konkrétním typem uzlu.

```
C3D_FORCE_INLINE void Traverser::TraverseObject(ObjectPtr pObject)
{
    unsigned int iCode = pObject->GetObjectCode();
    C3D_ASSERT(iCode!=OC_INVALID, "RequestToTraverseInvalidObject!");
    //volání registrovaní metody traverseru
    C3D_ASSERT(m_FuncTbl[iCode], "ObjectIsNOTregisteredForTraversing!");
    (this->*(void(Traverser::*)(ObjectPtr))m_FuncTbl[iCode])(pObject);
};
```

Zhodnoťme výsledný návrh. Zpracování instance uzlu teď zahrnuje pouze získání příslušného objektového kódu uzlu (inline) a volání ukazatele na členskou metodu. To je určitě efektivnější než dvě virtuální volání v předchozí verzi. Všimněme si, že obslužné metody Traverseru již nejsou nadále deklarovány jako virtuální. Dalším příjemným důsledkem vyplývajícím s použití ukazatelů na funkce jsou lepší možnosti řízení přístupu k metodám Traverseru. Obslužné metody jsou deklarovány jako

chráněné. Není již důvod, proto aby byly veřejné. To zjednodušuje klientský kód a omezuje možnost vzniku chyby z důvodu špatného použití třídy. Rozšíření operací nad grafem není žádným problémem a nemá vlastně žádná omezení. Jakoukoli operaci nad grafem je nyní jednoduché přidat pouhým odvozením od abstraktního rozhraní Traverseru a implementováním potřebných obslužných metod. Navíc pokud se změny týkají třeba jen jednoho typu uzlu, jsou pochopitelně k dispozici implementace v nadřazených třídách. Není nutná žádná změna ve starém kódu a ostatní operace zachovávají stejnou funkcionalitu. Návrh tedy splňuje námi stanovené požadavky.

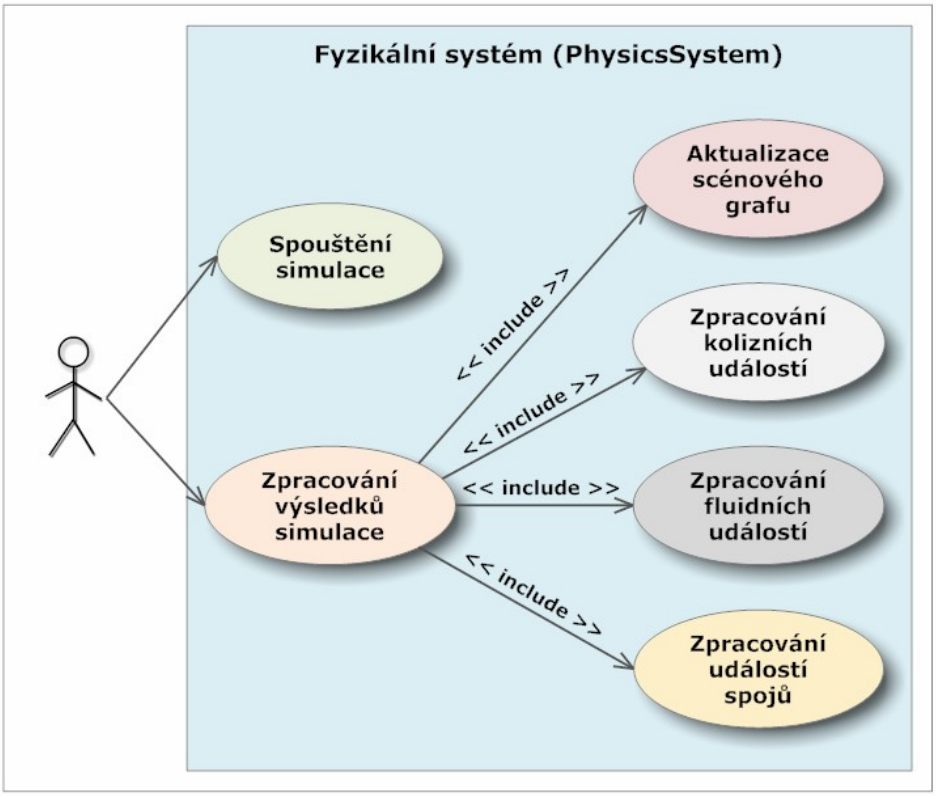
3.2 Fyzikální systém - PhysX

Fyzikální systém C3D engine je implementován pomocí NVIDIA PhysX SDK ve verzi 2.8.1 (dále jen PhysX). C3D engine v současnosti podporuje simulaci pevných těles, spojů (joint), fluidů a silových polí. PhysX jistě nabízí mnohem více. Další práce na fyzikálním systému C3D enginu by proto zákonitě pokračovala v integraci dalších modulů. Velmi zajímavé jsou např. možnosti měkkých těles (soft bodies). Kapitola uvádí nejdůležitější vlastnosti vybraných modulů PhysX a jejich integraci v C3D enginu. Na konkrétní použití a doporučení, stejně tak jako na problémy se podíváme v kapitole 4 Technologické demo.

Z pohledu programátora je NVIDA PhysX SDK knihovna či balík tříd pro simulaci pevných těles v reálném čase. Interně je knihovna implementována hierarchií tříd v C++. Navenek exportuje pouze abstraktní rozhraní základních tříd objektové struktury. Jména všech tříd PhysX začínají prefixem `Nx`. Pro aplikačně závislou funkcionalitu, např. zpracování kolizních informací nebo dalších událostí, PhysX poskytuje mechanismus zpětných volání. PhysX využívá daty řízený přístup pro vytváření objektů. Místo toho, aby uživatelský kód vytvořil objekt a poté ho metodami nastavoval, jsou veškerá data nutná pro správnou inicializaci objektu nastavena v deskriptoru (datová třída). Z deskriptoru je objekt vytvořen jedním voláním příslušné metody [PHYSX08].

Celkově je, dle mého soudu, PhysX velmi kvalitně navržen a díky poskytnutému balíku tutoriálů a dokumentace, ho lze v rozumném čase dobře pochopit. Některé oblasti jako třeba ladění výkonu jsou naproti tomu popsány málo a další problémy se objevují při konkrétní práci.

Fyzikální systém C3D zapouzdřuje práci s PhysX. Vstupem do systému je třída `PhysicsSystem`. Hlavní úkoly jsou spouštění simulace, zpracování výsledků simulace a správa fyzikální scény. To znázorňuje obrázek 3.2 – Případy užití fyzikálního systému C3D enginu.



Obrázek 3.2 – Případy užití fyzikálního systému C3D enginu

3.2.1 Integrace

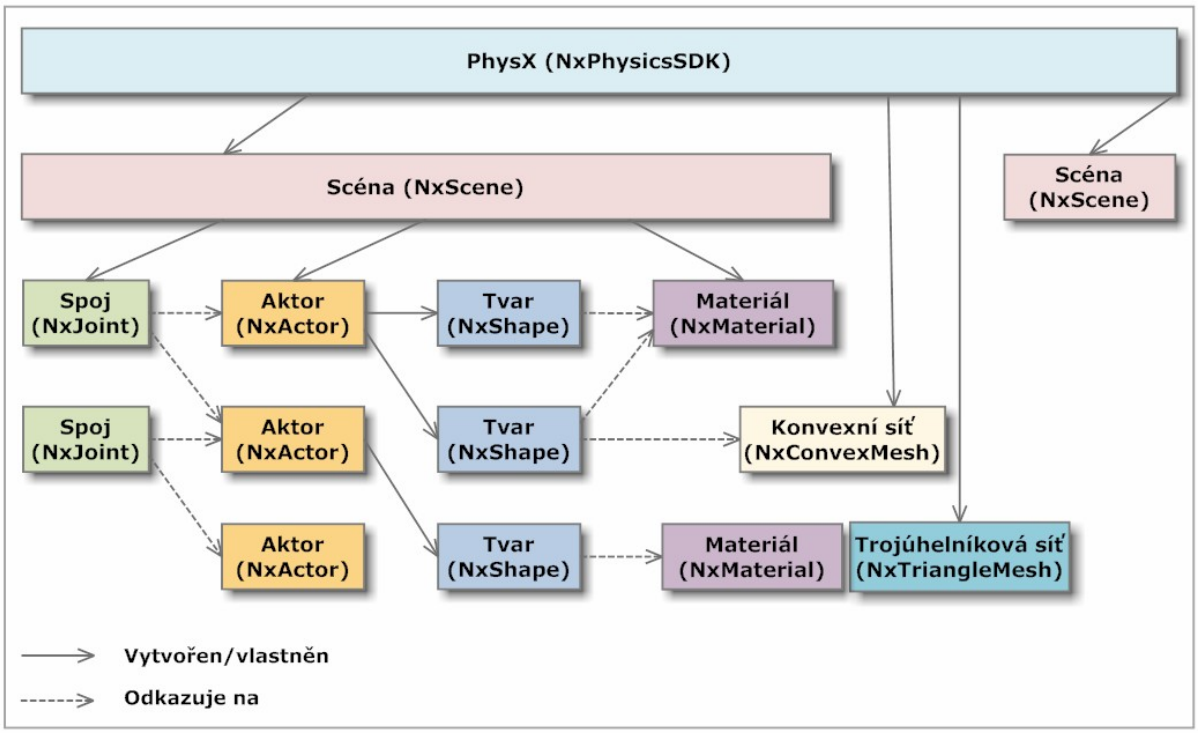
Filozofie PhysX nazývá pevné těleso aktorem (Actor). Aktory jsou protagonisté fyzikální simulace. PhysX dále rozlišuje aktory na statické a dynamické. Statické aktoy mají pevnou pozici, což umožňuje provádět optimalizace výpočtu simulací. Dynamické aktory pak musí mít definováno tělo. To specifikuje vlastnosti jako velikost a lokální pozici hmotnosti apod. [PHYSX08].

Každý hráč může mít přiřazený tvar (shape) – žádný, jeden či více. Tvar definuje fyzikální prostor hráče při simulaci a je základem pro fyzikální výpočty. Aktor bez žádného tvaru tedy nemůže kolidovat s žádným dalším hráčem ve scéně. Pozice tvaru je relativní k aktoru. Pro specifikaci fyzikálního prostotu máme k dispozici jednak základní druhy tvarů jako jsou kvádr, kapsule, koule nebo komplexnější konvexní trojúhelníková síť až po nejobecnější i nekonvexní trojúhelníkovou síť. Tvary, přiřazené jednomu konkrétnímu hráči, se mohou překrývat. Mezi tvary aktoru je totiž ve výchozím nastavení vypnutá detekce kolizí. Pro aktor, který je složen z více tvarů, se automaticky vytvoří obalovací tvar s cílem optimalizovat detekci kolizí [PHYSX08].

Aktory lze k sobě vázat spoji (Joint) a vytvářet tak komplexní celky. Joint propojuje vždy dva aktory, ale každý aktor může být spojen s dalšími pomocí jiných spojů. Spojů definuje PhysX několik

druhů. Každý vytváří vazbu s různými možnostmi omezení vzájemného pohybu. Např.: fixní spojení nedovoluje žádný pohyb jednoho hráče vůči druhému. Naopak s 6DOF (6 degree of freedom) spojem máme úplnou svobodu při definování omezení relativního pohybu. Mezi aktory s definovaným spojem je opět vypnuta detekce kolizí (v základním nastavení) [PHYSX08].

Samotná fyzikální simulace probíhá vždy v rámci scény. Celou architekturu vidíme na obrázku 3.3 – Architektura PhysX.



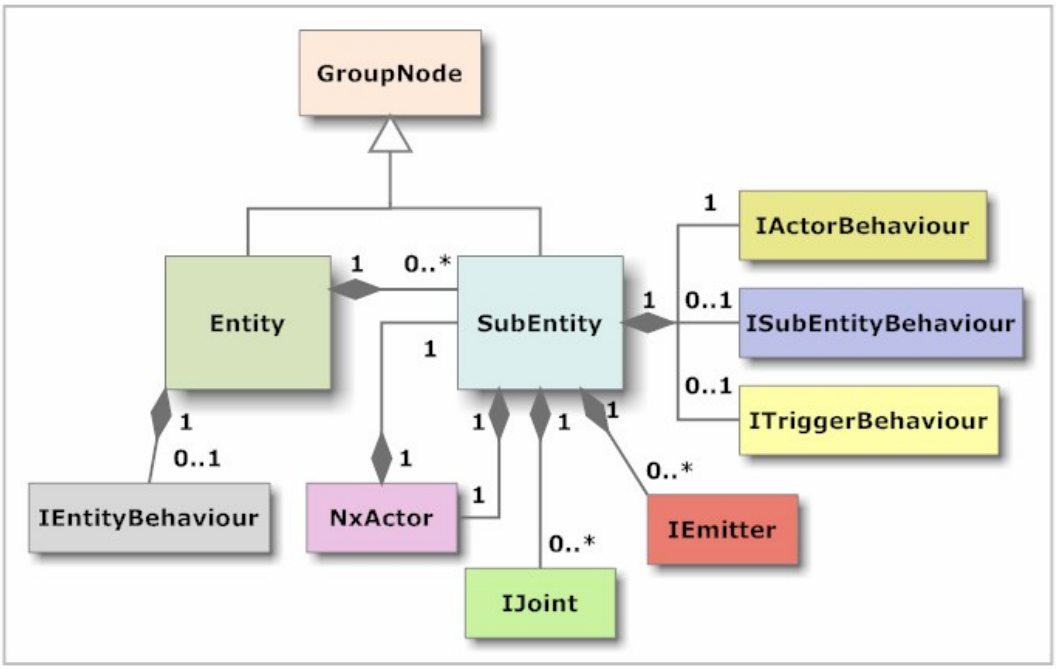
Obrázek 3.3 – Architektura PhysX

Motivaci návrhu integrace vysvětlím na příkladu. Představme si nyní vesmírnou bitevní loď. Naše loď je obrovská. Má velký trup, vzadu raketové motory a po stranách laserová děla. Laserové dělo má zcela zřejmě jiné fyzikální vlastnosti a také chování. Navíc hráč může dělo poničit nebo úplně zničit tak, že exploduje nebo se odlomí od lodi. Loď o tomto musí být informována. Již totiž nelze střílet všemi děli a také „zdraví“ lodě se musí adekvátně změnit. Zničené dělo se také bude zcela jistě chovat jinak než před svým zničením. Z pohledu enginu je naše loď reprezentována jednak graficky (geometrie, textury, materiál, ...) a fyzikálně pomocí aktorů. Obecně takový herní objekt nazveme entitou a její část subentitou. Z příkladu vycházejí požadavky:

- Entita je složena z více různých částí
- Entita má přístup ke svým částem
- Entita má chování

- Subentity můžeme přidávat a odebírat
- Subentita může mít jiné fyzikální vlastnosti
- Subentita má chování
- Subentita i entita může měnit svoje chování

Výsledný návrh vidíme na Obrázek 3.4 – Návrh integrace PhysX.



Obrázek 3.4 – Návrh integrace PhysX

Třída `SubEntity` modeluje část entity. Jde o obálku (wrapper) [GAMMA95] okolo rozhraní `NxActor`. To umožňuje specifikovat fyzikální vlastnosti nezávisle na entitě, ke které patří. Třída je odvozena od uzlu `GroupNode`. Fyzikální model lze tedy dále graficky reprezentovat libovolně. Všimněte si odkazu ze třídy `NxActor` na třídu `SubEntity`. Při vytváření entity (viz. kapitola 3.3.4 Entitní systém) je ukazatel na subentitu nastaven do `NxActor::userData` a používá se při zpracování výsledků simulace.

Entity budeme skládat z určitého počtu subentit. Pro propojení jednotlivých částí v jeden celek využijeme spojů (Joint). Spoje zapouzdřuje třída `PhysXJoint`, která definuje abstraktní rozhraní `IJoint`. Opět implementuje obal, v tomto případě okolo rozhraní `NxJoint`.

Vraťme se na chvíli k našemu příkladu vesmírné lodě. Pokud hráč zasáhne laserové dělo lodě, mohli bychom chtít, aby se dělo při dalším zásahu odtrhlo od lodě. Toho dosáhneme, když při prvním zásahu děla zmenšíme sílu nutnou pro přetržení spoje. K tomu nutně potřebujeme mít přístup na příslušné spoje subentity. Jak nepřímo ukazuje obrázek architektury PhysX, lze u spoje zjistit,

které aktory propojuje. Opačnou akci ale PhysX nedovoluje. Nelze přímo získat spoj pomocí aktoru. Každá subentita proto uchovává ukazatele na `IJoint`.

Třída Entity modeluje celkový pohled na herní entitu. Dědí z uzlu `GroupNode`. Díky tomu můžeme entitu přidat nebo odstranit ze zpracování. PhysX nedovoluje zadat transformační matici aktora relativně. I když je subentita synovským uzlem entity v grafu scény, transformace subentity je vždy globální (absolutní). Při změně transformace entity tedy musíme adekvátně změnit transformace všech subentit entity. Aktory subentit entity mají různé počáteční transformace. To vede na situaci, kdy transformační matice entity zůstává stále identická a podgraf scény reprezentující entity je degradován na pole subentit. Stále ale těžíme ze stromové struktury scénového grafu, protože můžeme nadále specifikovat např. efekt, který je vztažen na celou entitu.

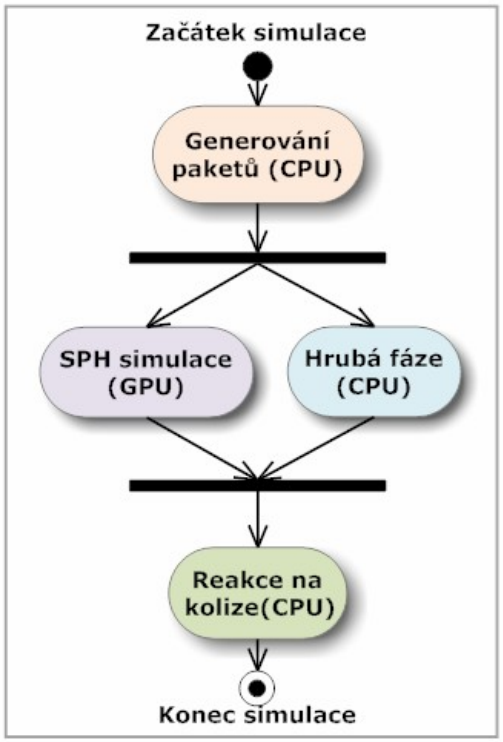
Systém chování stojí na komponentovém návrhu. Odvozením od patřičných rozhraní jsme schopni definovat chování objektů v různých situacích. Např. pro subentitu definujeme rozhraní `IActorBehaviour`, které je voláno v důsledku vzniku kolize. Logiku aplikace definujeme pomocí rozhraní `ISubEntityBehaviour` a `IEntityBehaviour`. Pro entitu máme k dispozici rozhraní `IEntityBehaviour`, chování spoje v okamžiku přetrhnutí můžeme definovat odvozením z rozhraní `IJointBehaviour` atp. Důležité je, že díky komponentovému návrhu zůstává reprezentace entity (grafická, fyzikální) oddělena od jejího chování v herním světě. Navíc implementací rozhraní můžeme poskytnout neomezený počet a chování lze dynamicky měnit. Získáváme tak velmi silný nástroj pro řízení životního cyklu entity i jejích částí.

3.2.2 Částicový systém - Fluidy

Fluidy dovolují simulaci efektů jako je kouř, exploze, tekutina atp. pomocí částicového systému a emiterů. Fluidní simulace probíhá ve třech režimech. Jistě nejzajímavějším je SPH simulace, tedy Smoothed Particle Hydrodynamice, česky označovaná jako vyhlazená hydrodynamika částic. Implementace SPH simulace je ve PhysX založena na Navier-Stoke rovnicích, které popisují pohyb částic fluidu v závislosti na rychlosti, hustotě a tlaku během času [SCHIRM–HARRIS10]. SPH bere v úvahu i síly působící mezi jednotlivými částicemi. Tyto síly jsou modelovány pomocí silových polí (force filed). Každá částice jich má přiřazeno několik. Např. tlakové silové pole způsobuje „odrážení“ částic od sebe. Povrchové napětí PhysX modeluje přitažlivými silami mezi částicemi určité hmotnosti. Tento model vede potencionálně na $N \cdot (N-1)$, (N - počet částic fluidu) výpočtů! PhysX toto řeší definicí okolí částice tedy prostoru, v němž další částice působí na právě zpracovávanou částici. Interakce mezi částicemi lze úplně vypnout nebo nastavit fluid do mixovaného režimu.

Simulace pak počítá jen některé tlakové charakteristiky. Různé fluidy na sebe nemohou v použité verzi PhysX vzájemně působit [PHYSX08].

Aby byla detekce kolizí částic se statickými a dynamickými objekty ve scéně dostatečně efektivní, PhysX generuje nejprve pakety – skupiny částic. Paket můžeme chápat jako speciální druh tvaru. V hrubé fázi (broad phase) následně probíhá samotná detekce potencionálně kolidujících objektů. Proces simulace fluidů vidíme na obrázku 3.5 – Proces simulace fluidů ve PhysX.



Obrázek 3.5 – Proces simulace fluidů ve PhysX

Z obrázku je patrné, že SPH simulace je prováděna na GPU, zatímco CPU současně detekuje kolize částic s objekty scény. Výsledky SPH simulace jsou následně staženy z GPU pro aktualizaci objektů.

Model používaný ve PhysX je vzhledem k realitě velmi zjednodušený. „Reálná“ simulace hydrodynamiky kapalin a plynů je výrazně náročnější. Vede na řešení lineárního systému o šestnácti miliónech neznámých a desítky minut výpočtu pro jeden snímek [MÜLLER07]!

Třídou implementující fluidy ve PhysX je `NxFluid`. C3D implementuje třídu `ParticleEffect` jako obálku okolo `NxFluid`. Pro vykreslení částic musíme znát jejich aktuální pozici, život a další atributy. Za tímto účelem nastavujeme fluidu uživatelsky alokovanou paměť prostřednictvím třídy `NxParticleData`. Při synchronizaci výsledků simulace jsou tyto bufery naplněny aktuálními daty. Stejným způsobem lze získat identifikátory nových nebo zrušených částic. Lze tak např. náhodně měnit počáteční rotaci částice. V každém snímku můžeme také simulované

částice aktualizovat metodou PhysX `NxFluid::updateParticles()`. Uživatelsky alokovanou paměť naplníme hodnotami síly pro částice, které chceme ovlivnit. Tento proces lze využít pro kontrolu tvaru celého efektu a dovoluje tak simulovat třeba klasický hřib při výbuchu bomby.

`ParticleSystem` je v C3D třídou, která spravuje jednotlivé efekty resp. fluidy. Vytváření efektů implementuje metoda `ParticleSystem::Create()` a funguje takto:

- Vytvoř nový PhysX fluid a nastav jeho parametry.
- Inicializuj bufery pro grafická data.
- Vytvoř a nastav objekt typu `ParticleEffect` pomocí nového fluidu.
- Vyvolej uživatelskou inicializaci.

Vstupem je jméno nového efektu a zdroj s fyzikálním popisem. Nový fluid je automaticky nastaven pro simulaci na GPU, pokud konkrétní počítač disponuje vhodnou grafickou kartou.

Inicializaci provádíme implementací rozhraní `ParticleEffectInitCallback`. Instanci přiřadíme částicovému systému (`ParticleSystem`). Při inicializaci budeme chtít efektu přiřadit texturu částic a kontrolér, který bude měnit vzhled částic. Současná implementace kontroléru dovoluje u efektu řídit velikost, barvu, průhlednost, rotaci a animaci částic. Hodnoty jednotlivých atributů definujeme jako funkci zbývajícího životního času částice (viz. kapitola 4.3 Efekty).

Do fluidů produkují nové částice emitery (`NxEmitter`). Jeden fluid může mít přiřazeno mnoho emiterů. Emitery definují hlavně počáteční vlastnosti emitovaných částic jako směr, zrychlení a tempo vzniku nových částic. Při zpracování informací načtených do nastavených buferů však v současném API PhysX nelze určit, který emiter částici vytvořil. Nemůže tedy např. přiřadit částicím jednoho emiteru jinou texturu než částicím druhého emiteru pokud emitují částice stejnému fluidu. V praxi tak musíme např. pro trosky při výbuchu použít jiný fluid než pro samotný výbuch. Na druhou stranu budeme pravděpodobně chtít, aby se částice trosek chovaly jinak než částice výbuchu.

Emiter lze přichytit ke tvaru aktora. Jde o velmi užitečnou vlastnost, kterou využijeme např. pro simulaci kouře za letící raketou. PhysX nedovoluje pomocí aktoru získat připojené emitery. To se jistě může hodit. V C3D enginu proto máme k dispozici metody `SubEntity::AttachEmitter()` a `SubEntity::GetEmitter()`.

3.2.3 Silová pole

Na pevná tělesa a látky (ve smyslu textilu nebo plátna) lze aplikovat síly na konkrétní objekty. Pokud je objektů více, vyžaduje to pro každý objekt vždy volání API PhysX v každém snímku. Režii lze

předejít využitím silových polí PhysX. Jedná se o fyzikální objekt tak jako třeba aktor, který má vliv na aktory (pevná tělesa), látky, měkká tělesa a fluidy. Silové pole tvoří skupina tvarů (stejně jako aktor), které definují prostor působení pole. Výslednou sílu, která působí na objekty uvnitř silového pole, definuje funkce jádra. PhysX nabízí lineární funkci nebo můžeme definovat funkci vlastní. U lineární určujeme konstantní část výsledné síly, dále lineární a kvadratický pokles síly se vzdáleností, šum a můžeme i přímo řídit výslednou pozici a rychlost objektu v silovém poli [PHYSX08]. Dosáhnout tak lze pestré škály efektů jako vítr, turbulence, antigravitaci apod., záleží jen na fantazii. C3D implementuje podporu pro silová pole PhysX třídou `ForceField`.

3.2.4 Simulace

Simulace je vždy prováděna v rámci scény. Nastavení scény určuje, jak bude simulace probíhat. Důležitou roli pro stabilitu simulace hraje volba časování. PhysX poskytuje časování fixní a variabilní. Při fixním časování je uplynulý čas interně rozdělen na zadaný počet kroků o zadané maximální délce. Pokud je uplynulý čas delší, je zbytek akumulován a přidán při další simulaci k aktuálnímu času snímku. Při variabilním časování PhysX nedělí čas snímku na kroky a ponechává tak zodpovědnost za stabilitu na aplikačním kódu. Jestliže aplikujeme v každém snímku sílu na nějaký aktor s cílem dosáhnout konstantní rychlosti, musíme si uvědomit, že pokud každý snímek trvá různou dobu, bude výsledná rychlost při aplikování stejné síly také různá. Aplikovanou sílu je nutné škálovat vůči referenční síle dle doby snímku [PHYSX08].

PhysX je navržen pro vícevláknové zpracování. Pro simulaci scény PhysX dovoluje nastavovat snad všechny myslitelné aspekty. Jsme schopni volit počet vláken, jejich prioritu a dokonce i spřažení vláken s procesory. Ve výchozím nastavení běží simulace ve vlastním vlákně odděleně od aplikačního (hlavního) vlákna [PHYSX08].

Metoda `PhysicsSystem::Simulate()` spustí simulaci aktuální fyzikální scény. Realizována je dvěma voláními PhysX.

```
void PhysicsSystem::Simulate(float dAppTime)
{
    //Neblokující volání
    m_pScene->simulate(dAppTime);
    //Zajisti zaslani všech potřebných dat simulačnímu vláknu
    m_pScene->flushStream();
};
```

Detekce kolizí probíhá v rámci všech tvarů ve scéně (ne aktorů). To vede potencionálně na $N \cdot N / 2$ detekcí kolize mezi dvěma tvary. PhysX proto automaticky rozděluje prostor, který tvary zabírají. Rozdělení prostoru probíhá v tzv. hrubé fázi (broad phase) detekce kolizí. Seznam tvarů

scény je seřazen tak, aby detekce probíhala pro daný tvar pouze s tvary v jeho blízkém okolí. Seřazení tvaru je výpočetně velmi náročné [PHYSX08].

3.2.5 Zpracování výsledků simulace

Pro synchronizaci simulačního a aplikačního vlákna slouží ve PhysX metoda `NxScene::fetchResults()`. Musí být volána vždy v páru s metodou `NxScene::simulate()` a vždy jako první po spuštění simulace. Metoda pracuje následovně:

- Zkontroluj, zda bylo simulační vlákno dokončeno.
- Vyvolej obsluhu uživatelských zpětných volání.
- Aktualizuj stavová data v buferu a prohoď stavové bufery, aby byl další simulační krok prováděn na jiném buferu.
- Zpřístupni bufer s výsledky pro zápis z aplikačního vlákna.

Systém zpětných volání dovoluje aplikačnímu kódu reagovat na kolize objektů, události fluidů nebo spojů. Zpětná volání jsou ale obsloužena před synchronizací výsledků simulace! Zcela reálnou reakcí na kolizi v herním světě však může být např. odstranění rakety, která narazila s mohutným výbuchem do domu. V tuto chvíli ale chceme z aplikačního vlákna zapisovat do ještě nesynchronizovaných buferů. Výsledkem je nedefinované chování. Dokumentace k PhysX pouze doporučuje nevytvářet nebo neodstraňovat objekty během obsluhy zpětných volání. Dle mého názoru je to velmi scestné doporučení. V praxi je nutné všechny události uložit a zpracovat až po synchronizaci simulačního vlákna s aplikačním. Implementovaná rozhraní pro obsluhu událostí přiřazujeme vždy scéně. Pokud simulujeme více scén současně, můžeme zpracovat vzniklé události pro každou scénu jinak.

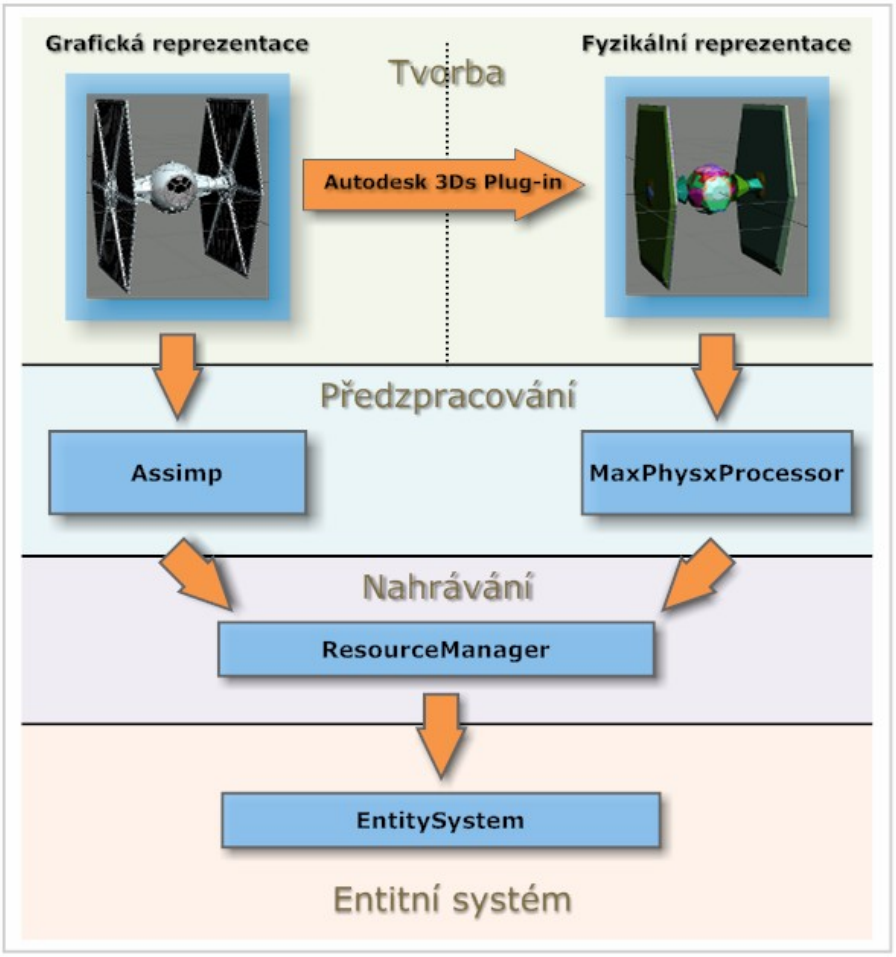
Informace o kolizích PhysX exportuje přes rozhraní `NxUserContactReport`. Rozhraní v C3D implementuje třída `PhysicsContactReport`. Pomocí NxContactStreamIterator jsou pro kolidující aktory získány konkrétní kolidující tvary a bod dotyku. Uložena je také událost jako taková. Tedy, zda se jedná o počáteční, vnitřní či koncový dotyk nebo byla překročena stanovená hranice síly. Po synchronizaci výsledků simulace jsou uložené kolizní informace zpracovány. Každá subentita musí mít přiřazenou implementaci rozhraní `IActorBehaviour`. Přes `NxActor::userData` získáme příslušnou subentitu a podle druhu kontaktu následně vyvoláme metodu rozhraní `IActorBehaviour`.

Aktuální transformace aktorů musíme zapsat do jejich subentit pro proces vykreslování. Díky návrhu operací nad grafem scény v C3D je realizace velmi jednoduchá a přímá. Vytvoříme nový traverser, `PhysXTraverser` (zděděním ze třídy `Traverser`), který při zpracování uzlu

`SubEntity` zkopíruje transformační matici aktora do transformační matice uzlu. Tímto dosáhneme nezávislosti vykreslování na fyzikální reprezentaci. Popsanou funkcionalitu zpřístupňuje metoda fyzikálního systému `PhysicsSystem::Update()`.

3.3 Entitní systém a aktiva

Kompletní proces vzniku entity v C3D enginu je znázorněn na obrázku 3.6 – Proces vzniku entity.



Obrázek 3.6 - Proces vzniku entity

3.3.1 Tvorba

Obecně lze pro tvorbu samotných dat použít jakýchkoliv nástrojů. Jejich výstup však musí nutně být ve formátu podporovaném fází předzpracování. Pro grafickou reprezentaci lze použít velký počet formátů díky implementaci předzpracování pomocí knihovny Assimp (viz. příloha č.3). Pro fyzikální reprezentaci je nutné, aby použitý nástroj exportoval fyzikální popis do formátu NxuStream2.

3.3.2 Předzpracování

Vytvořená vstupní data je nutné předzpracovat. Čili transformovat do formátu vhodného pro engine. Transformace grafické reprezentace probíhá pomocí knihovny Assimp, jak bylo nastíněno v předchozí podkapitole 3.3.1 Tvorba. Assimp importuje všechny podporované formáty do interní struktury podobné grafu scény. Na tuto strukturu aplikuje následné zpracování, které zahrnuje např. odstranění duplikátních materiálů a vrcholů, ale také přepočítání indexů do pole vrcholů pro zlepšení lokality vrcholů v cache.

PhysX plug-in for 3ds MAX exportuje popis ve formátu NxuStream2. NxuStream2 je knihovna dodávaná ve zdrojovém tvaru spolu s NVIDIA PhysX SDK [PHYSX08]. Knihovna pracuje nad kolekcí deskriptorů (`NxuPhysicsCollection`) objektů PhysX. Zapouzdřuje ukládání a nahrávání z binárního, xml a collada formátu. Velmi užitečná je její schopnost instancovat kolekce nebo její části. Pro předzpracování fyzikálního popisu vznikl nástroj MaxPhysXProcessor. Jedná se o konzolovou aplikaci, jejíž základem je právě knihovna NxuStream2. Nástroj usnadňuje převod mezi formáty NxuStream2, provázání grafické a fyzikální reprezentace pro entitní systém (viz. 3.3.4 Entitní systém) a převod D6 spojů na fixní spoje. Pokud nástroj spustíme bez parametrů, vypíše nám podrobnou nápovědu. Pro převod mezi formáty lze zadat jméno vstupního souboru a jméno výstupního souboru s příponou odpovídající požadovanému formátu NxuStream2.

Knihovna NxuStream2 definuje pro každý objekt PhysX (aktor, spoj, fluid, atd.) tzv. uživatelské vlastnosti (`userProperties`) ve formě řetězce. Důležité je, že tento řetězec je předán do rozhraní při vytváření fyzikálního objektu. Uživatelských vlastností využívá především entitní systém (viz. 3.3.4 Entitní systém). Pokud v MaxPhysXProcessoru zapneme automatické propojení grafické a fyzikální reprezentace, je část jména aktoru zapsána do uživatelských vlastností. Částí jména rozumíme podřetězec jména od jeho počátku po první výskyt tvrdé mezery nebo konce jména. Např. pro jméno aktoru „LeftWing_Actor“ bude do uživatelských vlastností zapsán řetězec „LeftWing“.

3.3.3 Nahrávání

Vstupem do systému zprávy zdrojů je třída `ResourceManager`. Poskytuje jednoduchý mechanismus pro získání určitého zdroje. Zdrojem rozumíme texturová data, geometrické informace, fyzikální popis atp. Zdroj je při svém prvním použití nahrán a uložen do hash mapy dle jména souboru zdroje.

Získání grafické reprezentace entity zajišťuje metoda `ResourceManager::GetGraphicsPrototype()`. Ta nahraje kompletní grafický popis

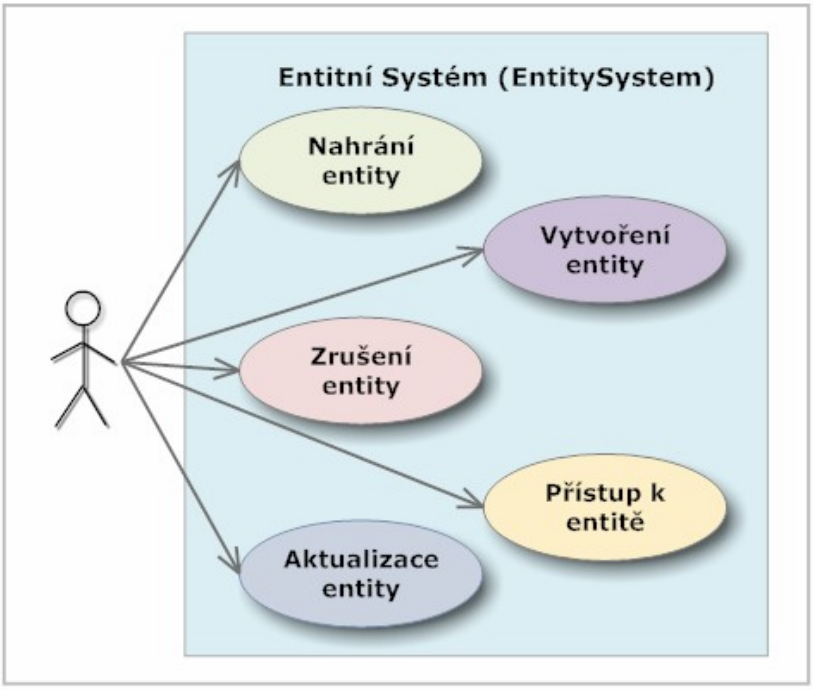
entity včetně textur, materiálů, geometrických dat ve formě části grafu scény. Tato část grafu funguje opravdu jako grafický prototyp pro vytváření entit entitním systémem, jak uvidíme dále. Proces využívá předzpracování geometrických informací knihovnou Assimp, texturová data jsou nahrána pomocí knihovny DevIL (viz. příloha č.3).

Získání fyzikálního popisu realizuje metoda `ResourceManager::GetPhysics()`. Využívá funkcí knihovny NXUStream2, konkrétně `LoadCollection()`. Kolekce je opět uložena do hash mapy dle jména souboru.

Tato implementace je opravdu velmi jednoduchá a pro komerční hru zcela nedostačující. Nahrávání totiž probíhá synchronně. Při přechodu z jiné herní úrovně to jistě nevadí. Ale nahrávání dodatečných zdrojů v průběhu vlastního hraní je prakticky nemožné, neboť dojde k pozastavení celé simulace. Řešení situace bylo popsáno v kapitole 2.2.5 Správce zdrojů. Předzpracování grafické části entity probíhá dynamicky, tedy až při běhu aplikace. To pochopitelně vede k výraznému prodloužení času nutného pro nahrání zdroje. Tento nedostatek odstraní implementace virtuálního souborového systému spolu s podporou serializace objektů systému [DELOURA01]. Na současnou implementaci se tak můžeme dívat jako na počátek opravdového správce zdrojů.

3.3.4 Entitní systém

Funkcionalitu implementuje třída `EntitySystem`. Obrázek 3.7 – Diagram užití entitního systému shrnuje hlavní úkoly entitního systému.



Obrázek 3.7 – Diagram užití entitního systému

Nejprve musíme entitu nahrát metodou `EntitySystem::LoadEntity()`. Nedělá nic jiného, než že pomocí správce zdrojů (`ResourceManager`) nahraje fyzikální popis a grafickou reprezentaci.

Daleko zajímavější je vytváření entity implementované metodou `EntitySystem::CreateEntity()`. Algoritmus pracuje následovně:

```
Vytvoř entitu()

Vstup: grafický prototyp a kolekce PhysX deskriptorů (fyzikální popis)
Výstup: nová instance entity

for všechny aktory v kolekci
{
    Vytvoř aktor
    Vyhledej GroupNode v grafickém prototypu, ke kterému aktor patří
    Vytvoř a nastav novou subentitu pomocí nalezeného uzlu a aktoru
    Klonuj podgraf nalezeného uzlu do nové subentity
    Přidej subentity do entity
}
Vlož entitu do aktuálního grafu scény
Zpětné volání uživatelské inicializace entity
```

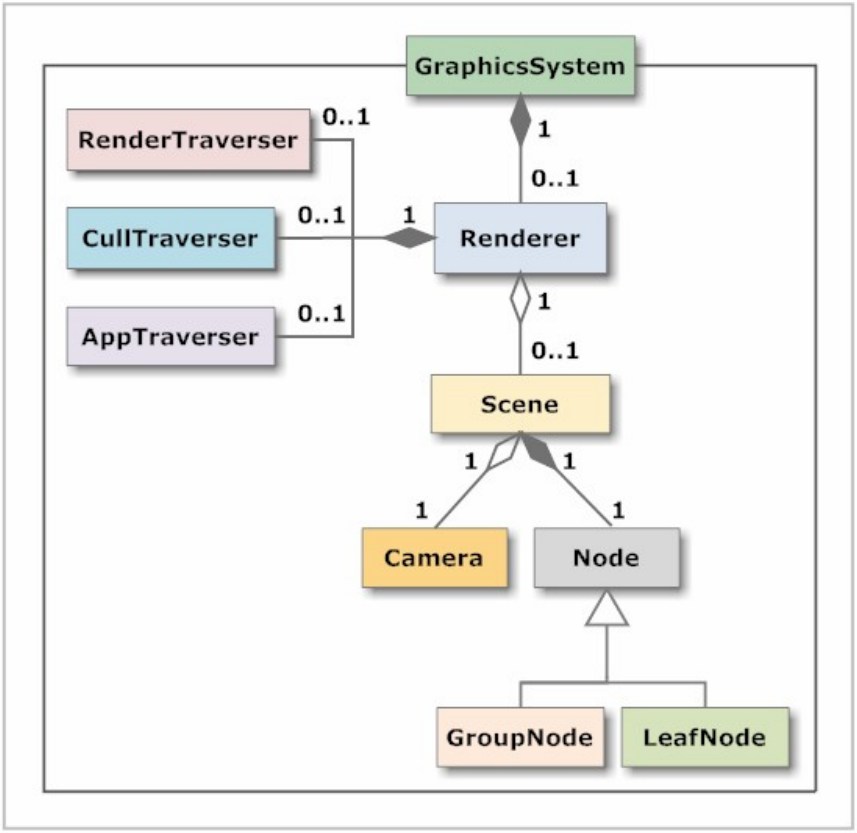
K vytvoření aktoru v kolekci využíváme schopnosti knihovny NxuStream2 instanciovat objekty PhysX. NxuStream2 dovoluje implementovat rozhraní `NXU_userNotify`, které je voláno při vytváření. Pro každý objekt PhysX (aktor, spoj, fluid, atd.) poskytuje rozhraní dvě metody. První je volána pro deskriptor objektu. Dovoluje upravit deskriptor před vytvořením samotné instance nebo jeho vytvoření úplně potlačit. Druhá metoda je vyvolána po vytvoření objektu a poskytuje šanci k dalšímu zpracování nového objektu, což je náš případ. Třetím parametrem všech metod rozhraní `NXU_userNotify` jsou uživatelské vlastnosti (`userProperties`). Implementace vytvoření entity předpokládá, že tento parametr nese jméno uzlu v grafickém prototypu, které přísluší aktuálně procesovanému aktoru. Uzel je vyhledán průchodem prototypu do hloubky. Implementace nezaručuje unikátnost jména uzlu v grafu. Nalezen je první uzel s požadovaným jménem a musí být typu `GroupNode`.

Nyní vytvoříme novou instanci subentity (třída `SubEntity`). Pomocí přiřazovacího operátoru zkopírujeme obsah nalezeného uzlu typu `GroupNode` (třída `SubEntity` dědí právě ze třídy `GroupNode`) do subentity. Potomky nalezeného uzlu naklonujeme pomocí členské metody `Object::Clone()`. Každý typ v systému musí tuto metodu implementovat. Dále přiřadíme aktor nové subentitě a uživatelský ukazatel aktoru (`NxActor::userData`) nastavíme na subentitu, aby bylo umožněno zpracování subentit fyzikálním systémem.

Nová entita vyžaduje další nastavení, než bude možné její použití. Jde hlavně o nastavení chování subentit a entity. Inicializace entity musí být ponechána na uživatelském kódu a lze ji implementovat prostřednictvím rozhraní EntityInitCallback. Vstupem je pochopitelně nová entita a na základě jejího typu můžeme provést inicializaci.

3.4 Grafický systém

Funkcionalitu systému zpřístupňuje třída `GraphicsSystem`. Implementuje návrhový vzor Facade [GAMMA95]. Třída poskytuje vstupní bod pro práci s grafem scény, pro vykreslování a implementuje také aktualizaci systému. Návrh grafického systému popisuje obrázek 3.8 – Diagram tříd grafického systému.

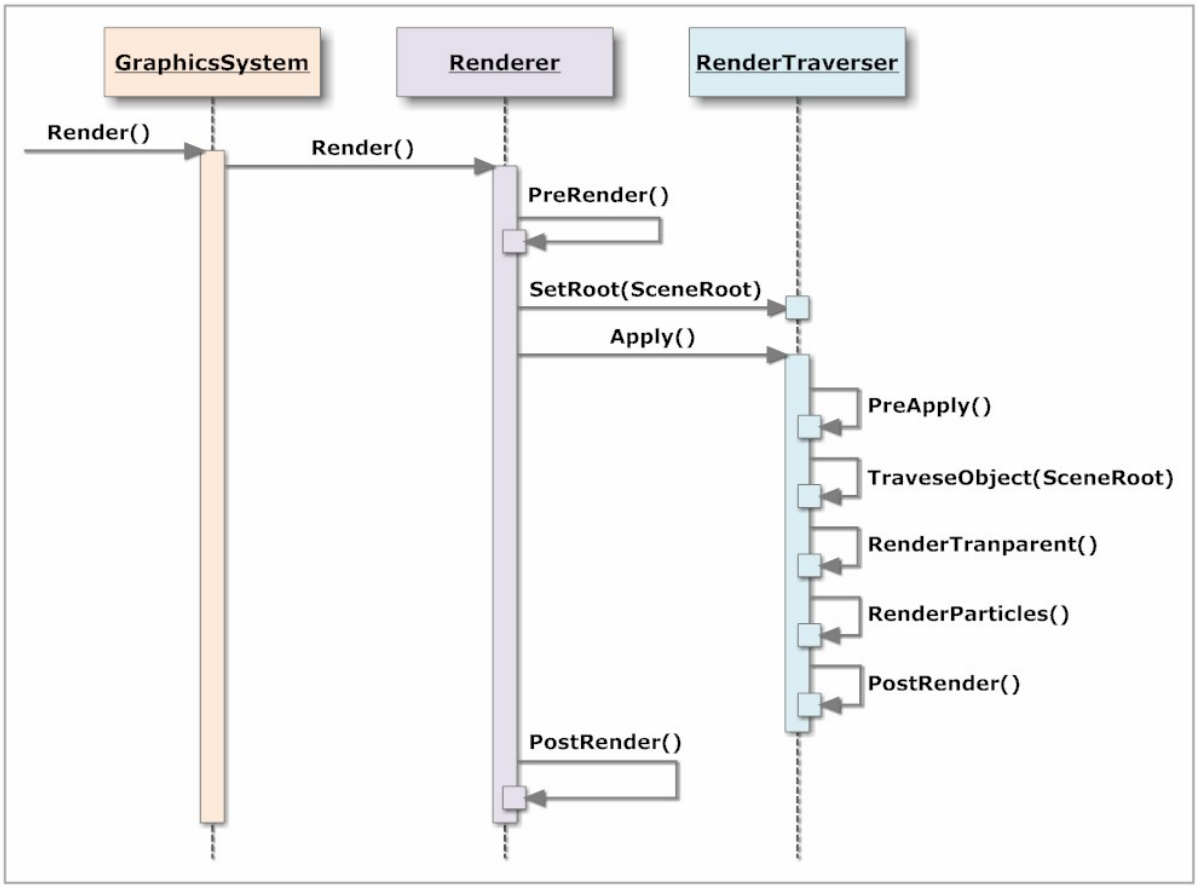


Obrázek 3.8 – Diagram tříd grafického systému

Grafický systém odpovídá hlavně za proces vykreslování. Implementováno bylo klasické dopředné vykreslování pomocí OpenGL API. Hlavní důraz je kladen na kvalitní vykreslování částicového systému. Funkcionalitu zapouzdřují hlavně třídy `OpenGLRenderer` a `OpenGLTraverser`. Při inicializaci rendereru odkazem na okno aplikace vytvoříme OpenGL kontext [WRIGHT–LIPCHAK05] a Cg kontext. Dále je pomocí knihovny GLEW (viz. příloha č.3)

detekována přítomnost potřebných rozšíření a jsou nastaveny základní atributy OpenGL stroje. Pokračujme inicializací traverseru metodou `OpenGLTraverser::Init()`. Ta rezervuje paměť pro hloubkovou texturu a nastaví její parametry. Poté je vytvořen shader pro vykreslení částicových efektů (viz. následující podkapitola 3.4.1 Animated Depth Soft particles).

Samotný proces vykreslování znázorňuje Obrázek 3.9 – Vykreslovací sekvence.



Obrázek 3.9 – Vykreslovací sekvence

Scénový graf je procházen do hloubky. Při průchodu stromem jsou vykreslovány pouze neprůhledné objekty (`TraverseObject()`). Geometrie se při prvním vykreslování nahraje do statického vertex bufer objektu (VBO) [WRIGHT–LIPCHAK05]. Průhledné objekty jsou uloženy do pole pro další zpracování stejně jako částicové efekty. Traverser provádí během vykreslování jen základní optimalizace. Kontroluje, zda není nastavován již nastavený (identický) materiál či textura. Neprobíhá tedy žádné řazení uzlů pro dosažení nižší režie. Transformace jsou ukládány na zásobník. V každém procházeném uzlu je vrchol zásobníku násoben transformací uzlu a výsledek je opět uložen na vrchol. Rozdílné zpracování probíhá v listovém uzlu. Výsledek skládání transformací není uložen na vrchol, ale ihned nastaven OpenGL stroji voláním `glLoadMatrixf`. Detekce viditelnosti také nebyla implementována. Nutno však říci, že návrh počítá i s těmito požadavky a funkčnost lze díky

tomu snadno implementovat. Jako poslední jsou vykresleny průhledné objekty (`RenderTransparent()`) a částicové systémy technikou měkkých částic (`RenderParticles()`).

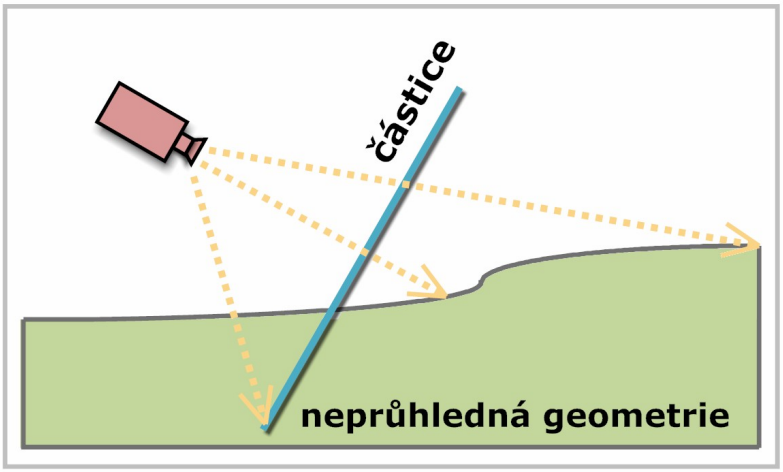
3.4.1 Animated Depth Soft particles

Nevýhodou techniky point-spritů jsou problémy s artefakty, které vycházejí ze samotné podstaty této metody. Snažíme se totiž objemová data nahradit určitým počtem velkých ploch. Ty pak mohou zasahovat do geometrie v pozadí. Důsledkem je výrazná průsečnice billboardu s geometrií v případě průniku. To je způsobeno „ostrostí“ hloubkové funkce (depth function). Fragment buď je, nebo není odstraněn z dalšího zpracování podle výsledků hloubkové funkce. Neexistuje nic mezi tím.

Implementovaná metoda se snaží toto odstranit provedením dalších akcí nad hloubkou fragmentu. Využijeme alfa míchání.

- Čím blíže je fragnment ke scéně v pozadí, tím větší bude jeho průhlednost
- Čím dále je fragment od scény v pozadí, tím bude jeho průhlednost menší
- Fragment za scénou nebude vykreslen

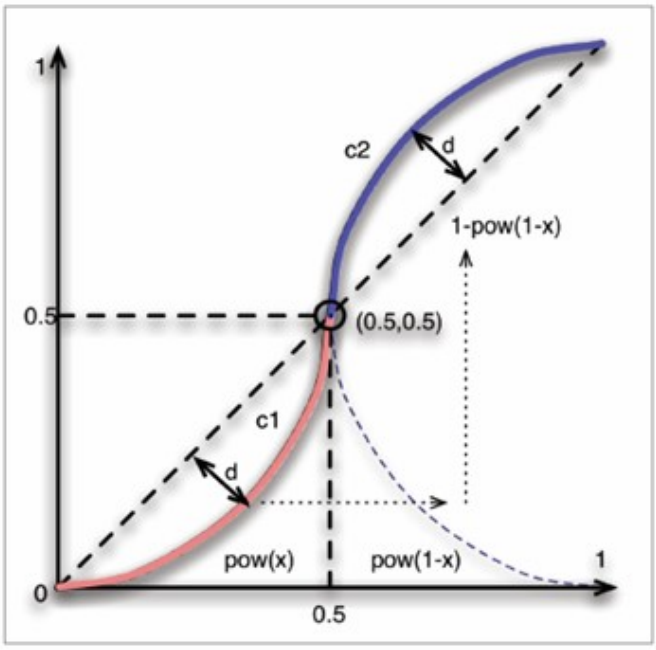
Celou situaci ilustruje obrázek 3.10 – Situace s klasickou částicí.



Obrázek 3.10 – Situace s klasickou částicí

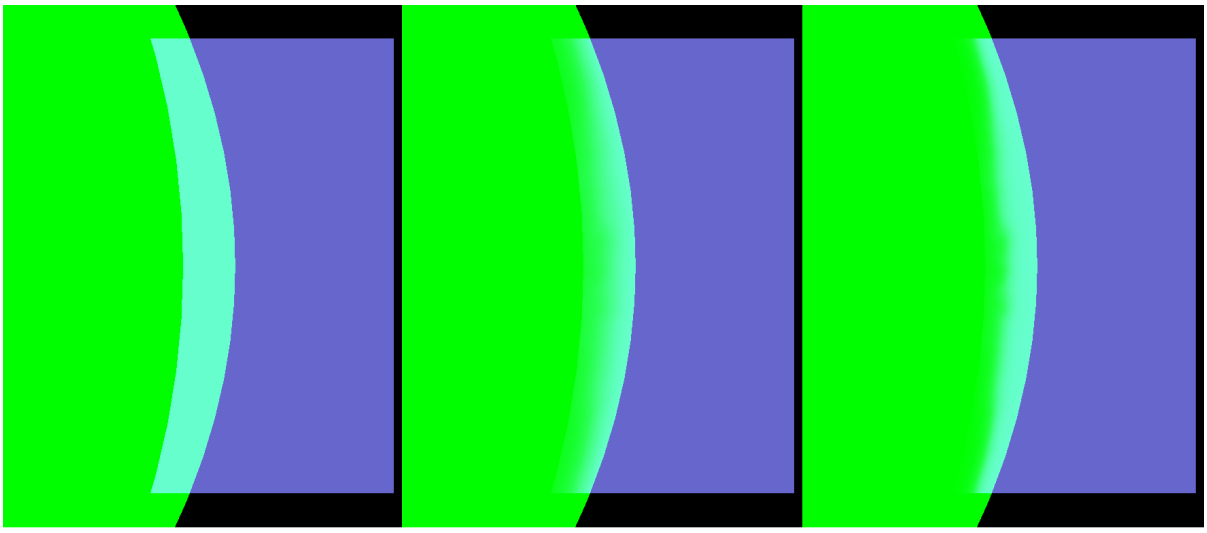
Z rozdílu hloubek nyní vypočítáme koeficient, který bude modulovat alpha hodnotu zpracovávaného billboardu. Jednoduchým přístupem k výpočtu koeficientu je saturovat škálovaný rozdíl hloubek. Tento přístup však neposkytuje uspokojivou kvalitu. V některých případech se objevují

nespojitosti, způsobené saturační funkcí. Použita je kontrastní funkce. Ta poskytuje hladší průběh a je symetrická [LORACH07]. Její průběh vidíme na obrázku 3.11 – Průběh kontrastní funkce.



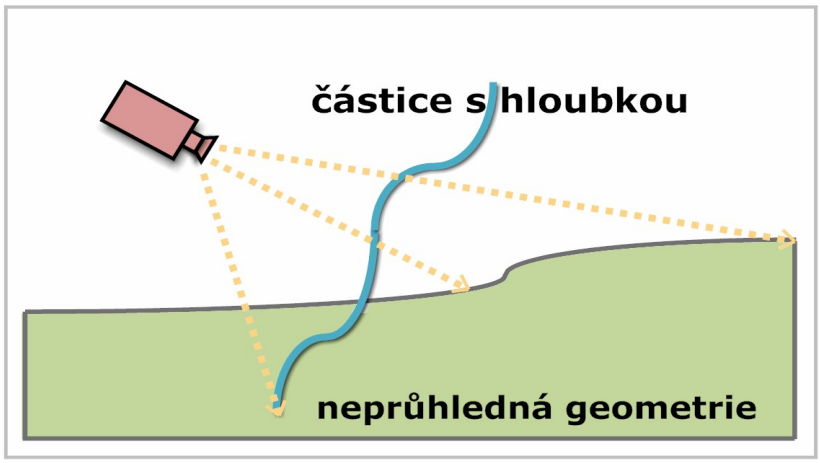
Obrázek 3.11 – Průběh kontrastní funkce

Výsledky aplikace kontrastní funkce vidíme na obrázku 3.11 – Kontrastní funkce v akci.



Obrázek 3.12 – Kontrastní funkce v akci (vlevo klasická hloubková funkce, uprostřed a vpravo kontrastní funkce s kontrastem 6 a 15)

Získaný koeficient útlumu alpha hodnoty je dále modulován hloubkou fragmentu [SOFTPAR10]. Odtud slovo Depth v názvu techniky. Hodnota je uložena v alfa kanálu textury spritu. Důvodem použití je snaha přiblížit se ještě více volumetrické povaze simulovaných efektů a zvýšit tak kvalitu. Novou situaci vidíme na obrázku 3.13 – Situace částice s hloubkou.



Obrázek 3.13 – Situace částice s hloubkou

Generování point-spritů je vhodnou úlohou pro geometry shader [GPUGUIDE08]. Geometry shader implementuje rotaci spritu a v texturové souřadnici zasílá hloubku spritu. Zasílání hloubky je nutné, protože pixel shader má přístup pouze k obrazovým (2D) souřadnicím současného fragmentu.

Technika měkkých částic utlumuje fragment na základě rozdílu hloubky fragmentu a scény na pozadí. Pixel shader tedy musí mít přístup k hloubkové mapě scény (depth map). Mapu lze získat použitím FBO (frame buffer object), PBO(pixel buffer object) nebo lze zkopírovat potřebné informace přímo ze snímkového buferu. FBO je více vhodné pro získání hloubkové mapy při pro generování stínů, PBO je zase výhodnější použít pro MRT (multiple render target) rendering. Důvodem vyřazení těchto technik je nutnost opakovaného zasílání geometrických dat ke zpracování na GPU. Nelze totiž vykreslovat naráz do frame buferu a FBO nebo PBO. V našem případě nám totiž stačí hloubková mapa z pohledu kamery. Implementace nejprve při inicializaci (`OpenGLTraverser::Init()`) rezervuje v paměti prostor pro dvojrozměrnou texturu o stejném rozlišení jako snímkový bufer. Při samotném vykreslování jsou aktualizována data textury současnými hodnotami hloubky scény pomocí OpenGL volání `glCopyTexSubImage2D` [WRIGHT–LIPCHAK05].

Získaná mapa však obsahuje informace o hloubce v normalizovaném clip prostoru – po vydělení váhou bodu. Hloubka spritu, zasílaná geometry shaderem, je ale v prostoru projekčním. Hloubku scény musíme tedy nutně převést zpět z clip do projekčního prostoru – před vydělením váhou souřadnice.

Násobením souřadnic bodu (*x,y,z,w*) v modelovém prostoru projekční maticí a následným dělením váhou bodu získáme pro souřadnici *z* následující vztah [LORACH07]:

$$z_{buf}\,\frac{f\cdot(z-n)}{(f-n)\cdot z}$$

, kde *f* - far plane, *n* - near plane, *z* - z souřadnice v projekčním prostoru. Z toho pro *z* v projekčním prostoru platí:

$$z=\frac{f\cdot n}{f-z_{buf}\cdot(f+n)}$$

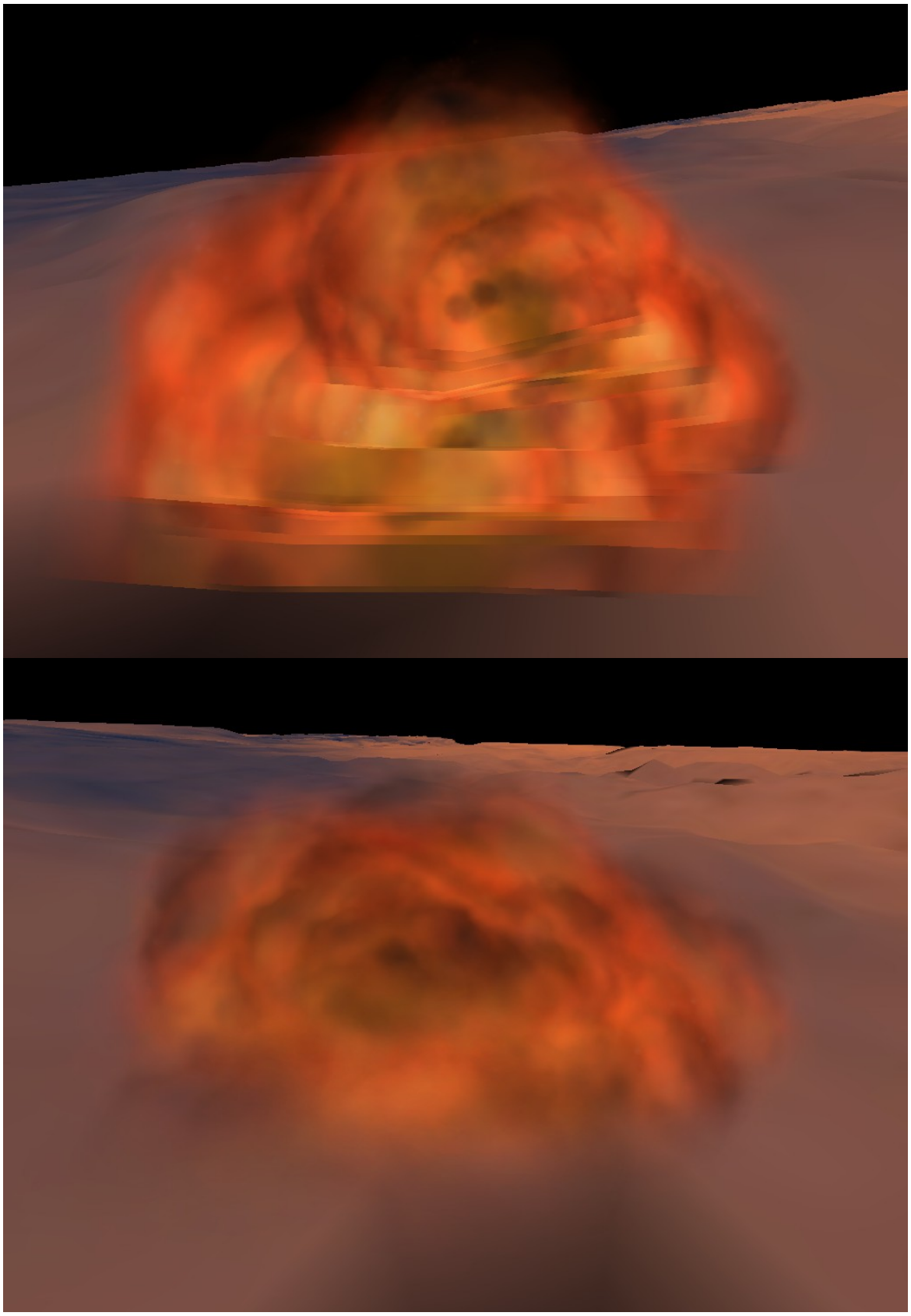
Tuto hodnotu již můžeme použít pro samotný výpočet koeficientu.

Hlavní výpočet realizuje pixel shader. Pokud efekt, např. výbuch, zabere velkou část obrazu, zvýší se výrazně výpočetní náročnost a výkon rapidně klesne. To je způsobeno velkým počtem pixelů, které musí pixel shader zpracovat. Řešením je vykreslovat částice do zvláštního obrazového buferu, který má menší rozlišení než hlavní obrazový bufer. Manipulací s jeho velikostí je možné měnit poměr mezi výkonem a kvalitou zobrazení. Nevýhodou je ale nutnost převzorkování mapy s hloubkou scény do menšího rozlišení, což jistě přinese artefakty [CANTLAY07].

Implementovány byly dvě techniky CgFx efektu. Obě využívají techniku Depth Soft Particles. První aplikuje na sprite vždy stejnou texturu. Ta druhá využívá pole textur k docílení animace. Textury pro animaci jsou ukládány do 2D texturového pole (2D texture array). Oproti 3D textuře nám poskytuje svobodu při filtraci mezi texturovými vrstvami. Automatická trilineární filtrace u 3D textury často nedostačuje. Další nevýhodou 3D textury je, že vyžaduje, aby byl počet vrstev mocninou dvou. Důsledkem je vyšší paměťová náročnost v případě, že je počet vrstev menší. Ostatní vrstvy nejsou použity, ale paměť je pro ně stejně alokována[1].

Implementovaná technika je ve svém principu jednoduchá, výsledek je však více než uspokojivý, jak můžeme vidět na obrázku 3.14 – Implementovaná technika měkkých částic v akci. Ostré hrany jsou zcela odstraněny, celková kvalita je mnohem vyšší. Další práce by jistě směřovala k osvětlení částic a generování stínu efektu.

[1] Tato nevýhoda vymizí, pokud implementace OpenGL v ovladači grafického adaptéru exportuje řetězec rozšíření ARB_texture_non_power_of_two.

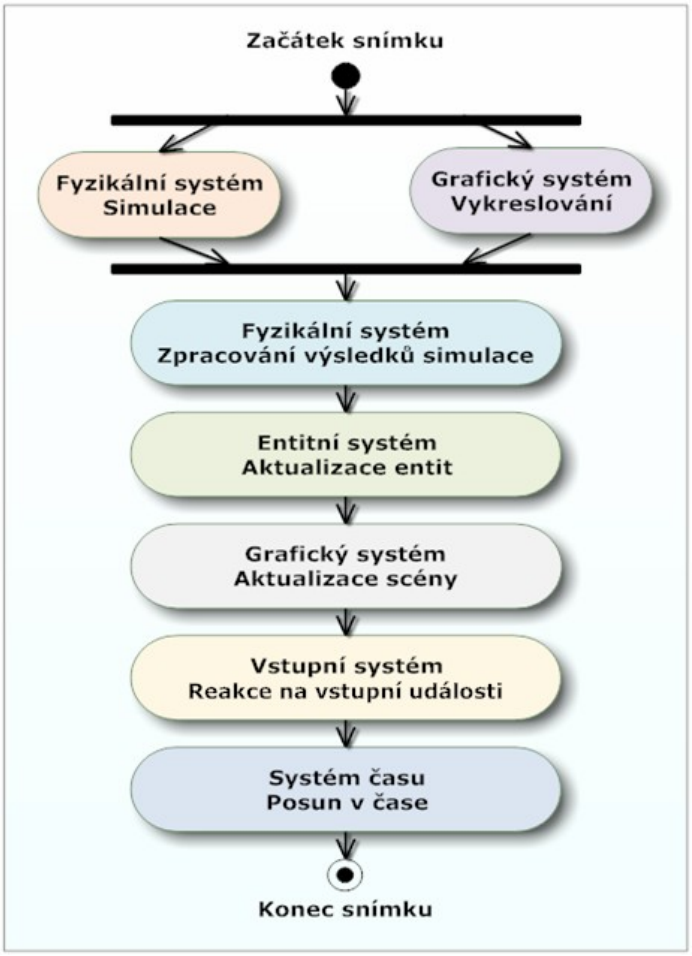


Obrázek 3.14 – Implementovaná technika měkkých částic v akci (nahoře vypnuta dole zapnuta)

3.5 Výsledná architektura a herní smyčka

Všechny systémy operují nad scénovým grafem. Implementovány jsou jako singleton [GAMMA95]. Šablona singletonu zaručuje správné odstranění systému z paměti i před koncem aplikace [DELOURA00]. Máme tak úplnou kontrolu nad životním cyklem systémů. Systémy jsou na sobě nezávislé. Pomocí C3D enginu je tedy možné vytvořit např. prohlížeč grafických modelů bez nutnosti inicializovat fyzikální systém. Pro hru však budou s největší pravděpodobností vytvořeny instance všech systémů. Architektura dovoluje implementaci různých např. fyzikálních systémů. Jeden může být pro výkonné PC, další pro starší hardware, kde budou fyzikální výpočty značně zjednodušeny. Uživatel si pak může zvolit, který chce použít, aniž by to mělo vliv na ostatní systémy a funkcionalitu enginu jako celku. Výsledná architektura je datově centralizovaný systém systémů.

Herní smyčka, dokumentuje ji obrázek 3.15 – Herní smyčka, je částečně asynchronní. Paralelně probíhá výpočet dalšího kroku fyzikální simulace, zatímco je vykreslována scéna.



Obrázek 3.15 – Herní smyčka

4 Technologické demo

Další kapitoly a odstavce popisují implementaci technologického dema. Demo slouží pro demonstraci implementovaných vlastností C3D enginu, rovněž ověřilo PhysX engine v praxi a poskytlo prostor k experimentování s jeho rozmanitými schopnostmi a nastaveními. V textu jsou popsány pouze parametry a příznaky PhysX, které byly měněny nebo hrají výraznou roli při simulaci. Popis všech parametrů lze nalézt v dokumentaci k PhysX, která je umístěna na přiloženém CD. Celé technologické demo implementuje třída `Demo`.

Demo je hratelné a můžeme ho považovat za základ hry (viz. obrázek 4.1 – Ukázka z dema). Jde klasickou arkádovou akční hru tzv. shoot'em up. Hráč ovládá šipkami pohyb vesmírné lodě X-Wing a mezerníkem střílí rakety do nepřátelských lodí. Nepřátele představují lodě Tie Fighter. Zásah nepřítele raketou vyvolá mohutný výbuch a zničená loď se řítí k povrchu Marsu. Nepřátelé v současnosti nestřílí proti hráči rakety.

Obrázek 4.1 – Ukázka z dema

4.1 Inicializace

C3D engine musíme nejprve inicializovat. Inicializaci implementuje metoda `Demo::InitEngine()`. První musí být vytvořen logovací systém, neboť ho využívají všechny ostatní systémy. Konstruktoru je předáno jméno xml konfiguračního souboru v aktuálním pracovním adresáři. Následuje vytvoření správce zdrojů. Ten některé další systémy využívají již při vytváření. Na dalším pořadí nezáleží.

```
try { //Logovací systém
    new Log::LogSystem(Log::ConfigFileReader("LogSystem.config.xml"));
}
catch(...) {
    exit(1);      //Výjimka je ošetřena logovacím systémem
}

m_pResourceManager = new Engine::ResourceManager;  //Správce zdrojů
m_pParticleSystem = new Engine::ParticleSystem;    //Částiový systém
m_pInputSystem = new Input::InputSystem;           //Systém vstupů
m_pTimeSystem = new Engine::TimeSystem;            //Systém času
m_pPhysicsSystem = new Physics::PhysicsSystem;     //Fyzikální systém
m_pGraphicsSystem = new Graphics::GraphicsSystem;  //Grafický systém
m_pEntitySystem = new Engine::EntitySystem;        //Entitní systém
```

Grafický systém vyžaduje ještě inicializaci. Metoda `GraphicsSystem::Init()` vytvoří výchozí OpenGL renderer a příslušné traversery (aplikační a vykreslovací). Nyní potřebujeme inicializovat samotný renderer. K tomu musíme nutně vytvořit okno. Demo vytváří okno pomocí knihovny SDL.

```
g_GraphicsSystem.Init();           //Vytvoří výchozí renderer
HWND hWin = InitWindow();          //Vytvoř okno
//Inicializace rendereru
RendererPtr pRender = g_GraphicsSystem.GetCurrRenderer();
pRender->SetWindow(hWin);          //Nastav okno pro vykreslování
pRender->Init();       //Vlastní inicializace (OpenGL context, ...)
pRender->Reshape(0,0, 1024, 768);  //Vykresluj na celé okno
```

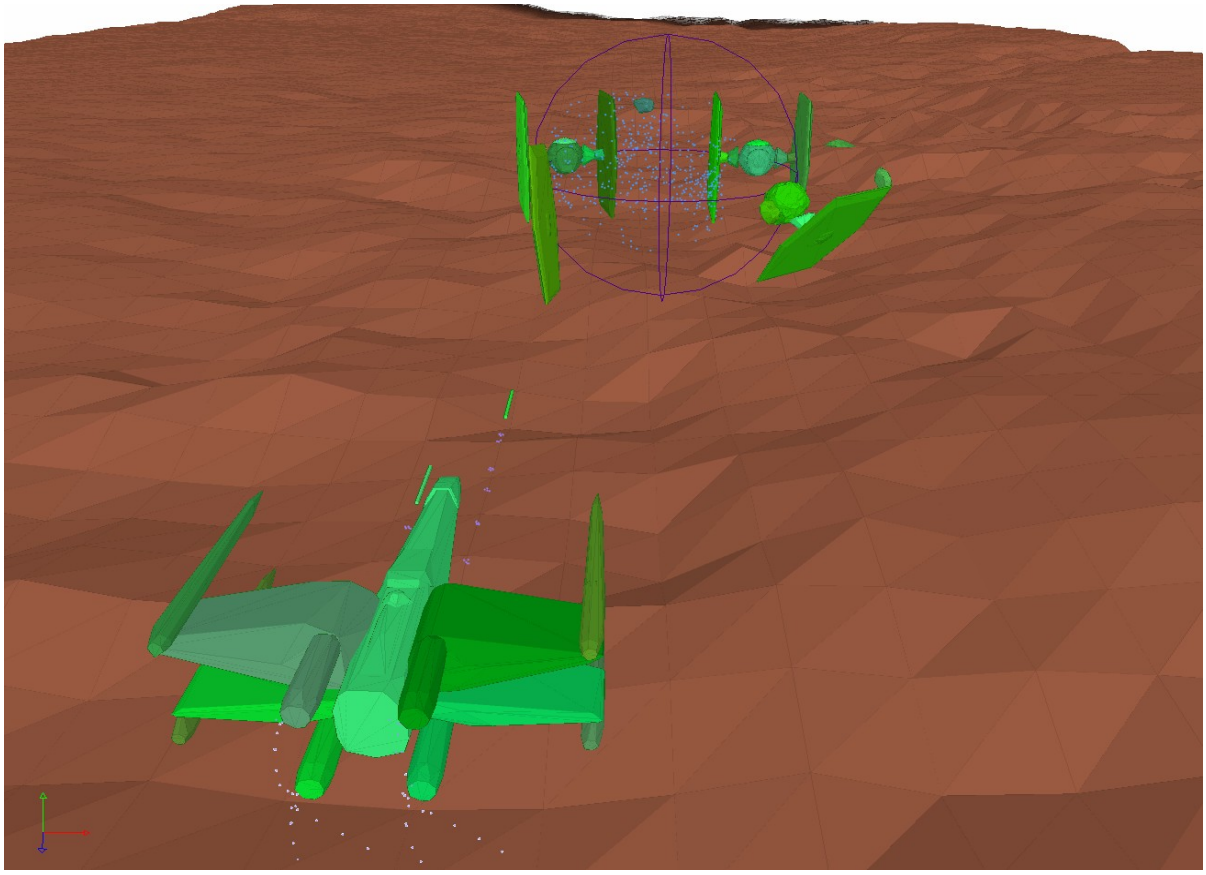
Inicializace fyzikálního systému zahrnuje:

- konstrukci hlavního objektu PhysX
- přiřazení zpětného volání pro obsluhu chyb PhysX
- detekci přítomnosti grafické karty pro možnou hardwarovou akceleraci
- pokus o připojení k nástroji PhysX Visual Debugger

PhysX Visual Debugger je volný nástroj firmy NVIDIA pro vizualizaci PhysX scény. Během vývoje enginu a dema se velmi osvědčil a doporučuji ho při práci s PhysX využívat. Připojení probíhá na adresu localhost s portem 5425.

```
g_PhysicsSystem.Init();    //Vytvoří PhysXSDK
```

Na následujícím obrázku 4.2 vidíme zobrazení scény dema nástrojem PhysX Visual Debugger.



Obrázek 4.2 – Scéna dema ve PhysX Visual Debugger

Jako poslední nastavíme výchozí hodnoty sytému vstupních událostí.

```
//Inicializace systému vstupů
g_InputSystem.SetWindow(hWin);
g_InputSystem.Init();
```

V tuto chvíli máme přístupnu veškerou implementovanou funkcionalitu C3D enginu a potažmo i PhysX enginu.

4.2 Scéna

Než začneme sestavovat scénu dema, musíme nahrát potřebné zdroje. Implementace v metodě `Demo::LoadAssests()` je velmi jednoduchá a využívá služeb správce zdrojů (`ResourceManager`) a entitního systému (`EntitySystem`).

Sestavení scény dema implementuje metoda `Demo::SetupScene()`. V první řadě alokujeme instanci grafické scény a novou scénu přiřadíme grafickému systému.

```
//Grafická scéna obsahuje po vytvoření nového kořen scénového grafu
Graphics::ScenePtr pScene(new Graphics::Scene);
g_GraphicsSystem.GetCurrRenderer()->SetScene( pScene );
```

Vytvoříme kameru a tu přiřadíme scéně. Alokujeme a nastavíme hodnoty klíčových snímků. Vytvoříme animační kontrolér a přiřadíme klíčové snímky. Spuštěný kontrolér přidáme do kamery.

```
//Kamera
m_pCam = new Camera;
m_pCam->RotateXrad( gmtl::Math::PI*0.28f );
pScene->SetCamera(m_pCam);

//Nastavení pohybu kamery
PositionKeyFrames* keyFrames = new PositionKeyFrames;
keyFrames->resize(2);
keyFrames->at(0).fTime = 0;
keyFrames->at(0).vKey.set(0, -110, 0);
keyFrames->at(1).fTime = 230;
keyFrames->at(1).vKey.set(0, -110, 4000);

//Animační kontroler
CameraKeyFrameController* animCont = new CameraKeyFrameController;
animCont->SetPositionKeyFrames( keyFrames );
animCont->Start();

m_pCam->AddController( animCont );
```

Scénu nasvítíme bodovým zdrojem světla.

```
//Světlo
PointLightPtr pLight(new PointLight);
pLight->SetEnable();
pLight->SetPosition(0, 250, 800);
pLight->SetDiffuseColor( 1.f, 0.8f, 0.5f, 1.f );

pScene->GetRoot()->AddLight( pLight );
```

V tuto chvíli máme nastavenou grafickou část scény. Nyní přichází na řadu fyzikální scéna PhysX. Nastavený deskriptor předáme fyzikálnímu systému.

```
//Vytvoř scénu PhysX
Physics::SceneDesc sceneDesc;
sceneDesc.gravity = NxVec3(0.f, -9.8f, 0.f);
sceneDesc.flags |= NX_SF_FLUID_PERFORMANCE_HINT;
sceneDesc.backgroundThreadCount = 4;

Physics::ScenePtr pFyScene = g_PhysicsSystem.CreateScene(sceneDesc);
```

Deskriptor je v konstruktoru automaticky nastaven na výchozí hodnoty. To, mimo jiné znamená, že je nastaveno fixní časování. Při variabilním časování byla simulace v případě dema velmi nestabilní. Chování se měnilo s každým spuštěním aplikace. Spoje se natahovaly a přetrhávaly. Odtržené části entity poletovaly prostorem. Fixní časování touto nestabilitou netrpí a osvědčilo se. Mohu tak potvrdit pravdivost doporučení v dokumentaci PhysX [PHYSX08]. Simulace pevných těles scény byla ponechána v softwarovém režimu[1]. Scéna dema totiž v jednu chvíli obsahuje pouze desítky pevných těles (aktorů). Režie spojená s přenosem dat z/do GPU převyšovala vlastní dobu výpočtu a GPU nebylo dostatečně vytíženo. Výsledkem je kratší doba výpočtu na CPU. Výhod GPU akcelerace PhysX, tak využijeme hlavně při simulaci komplexních scén dnešních her a pokročilých modulů jako jsou simulace fluidů. Scéně je také nastaven příznak `NX_SF_FLUID_PERFORMANCE_HINT`. Příznak povoluje rychlejší, ale méně přesnou kolizi fluidů se statickou geometrií. Detekce kolizí se statickou geometrií pak probíhá o jeden krok simulace pozadu. Scéna dema modeluje fyziku terénu statickým aktorem s komplexní trojúhelníkovou sítí (~50000 trojúhelníků). Přesto po nastavení příznaku nebylo pozorováno žádné výrazné urychlení výpočtu. Vlastnost NxSceneDesc::`backgroundThreadCount` určuje počet vláken pro zpracování úloh na pozadí jako např. různé předzpracování dat zasílaných na GPU.

Vytvořené scéně dále nastavíme pravidla pro kolizní skupiny. Každý aktor můžeme přiřadit do kolizní skupiny. Pravidla určují, zda se mezi dvěma aktory různých skupin mají detekovat kolize.

```
//Nastavení kolizních skupin
pFyScene->setGroupCollisionFlag(1, 2, false);
pFyScene->setGroupCollisionFlag(0, 1, true);
pFyScene->setGroupCollisionFlag(0, 2, true);
```

Např. Tie Fighter bude patřit do skupiny nula. Ta koliduje se všemi skupinami aktorů. Naproti tomu raketová střela nekoliduje s X-Wing. Raketa patří do skupiny jedna a X-Wing do skupiny dva.

Nakonec systémům C3D enginu musíme nastavit implementovaná rozhraní pro uživatelská zpětná volání. Ty, jak už víme, obsluhují inicializaci entit, částičových efektů a silových polí.

```
//Uživatelská zpětná volání
g_PhysicsSystem.SetForceFieldInitCallback( new ForceFieldInit );
g_ParticleSystem.SetEffectInitCallback( new ParticleEffectInit );
g_EntitySystem.SetInitEntityCallback( new DemoInitEntity );
```

Scénu máme vytvořenou jak graficky, tak fyzikálně. Zbývá naplnit scénu efekty a entitami.

[1] Nastavení platí opravdu jen pro simulaci pevných těles. Např. fluidy můžeme nadále simulovat na GPU.

4.3 Efekty

Efekty technologického dema realizují tři fluidní částicové efekty. Jeden pro výbuchy raket, další pro simulaci kouře za raketou a poslední pro efekt trysek raketových motorů hráčovy lodě (X-Wing). Vytvoříme je velmi jednoduše pomocí částicového systému C3D enginu.

```
g_ParticleSystem.CreateEffect( "X_Rocket_Explode",
                                    "fluids\\rocket_explode_fluid.xml");
g_ParticleSystem.CreateEffect( "X_Rocket_Smoke",
                                      "fluids\\rocket_smoke_fluid.xml");
g_ParticleSystem.CreateEffect( "XWing_Noozle",
                                      "fluids\\xwing_noozle_fluid.xml");
```

Částicovému systému jsme již nastavili inicializační zpětné volání efektů. Při vytváření efektů chceme nastavit texturu a kontroler, který bude dle času života částice měnit její grafické vlastnosti. Toto si ukážeme na efektu trysek raketových motorů viz. `ParticleEffectInit::InitXwingNoozle()`. Efektu nejprve nastavíme texturu částic.

```
Graphics::TexturePtr pTex( new Graphics::Texture);
pTex->SetImage( g_ResourceManager.GetImage( "ex.png" ) );
pEffect->AddTexture( pTex );    //texturu přiřadíme efektu
```

Nyní přejdeme ke křivkám. Definujeme známé body křivky, vytvoříme samotnou interpolační křivku a nastavíme jí interpolátor. Implementován byl interpolátor lineární a kosinový.

```
//Definice známých bodů křivky alpha hodnoty
RealPoints<2>* alphaPoints = new RealPoints<2>;
alphaPoints->push_back( Vec2f(0.1f, 0.0f) );
alphaPoints->push_back( Vec2f(1.f, 0.6f) );
//Interpolační křivka
InterpolatedCurve<2>* alCur = new InterpolatedCurve<2>;
alCur->SetPoints( alphaPoints );
alCur->SetInterpolator( new CosinInterpolator );//interpolátor
```

Vždy se interpoluje zbývající životní čas částice. Implementace předpokládá, že známé hodnoty zbývajícího času života částice jsou uloženy v první komponentě známých bodů. Stejně definujeme křivky pro velikost, barvu, rotaci a animaci částice.

Zbývá založit nový kontrolér, přiřadit mu definované křivky a kontrolér nastavit částicovému efektu.

```
//Kontrolér částicového efektu
ParticlesEffectController* controller = new ParticlesEffectController;
controller->SetAlphaCurve( alCur );
pEffect->AddController( controller );
```

Kombinací proměnlivých atributů částice v čase jejího života s technikou animovaných měkkých částic a SPH simulací se podařilo dosáhnout více než uspokojivých výsledků, jak vidíme na obrázku 4.3 – Exploze nepřátelských lodí.

Obrázek 4.3 - Exploze nepřátelských lodí

Podívejme se nyní na fyzikální stránku věci blíže. Simulace a detekce kolizí fluidů PhyX je nesmírně náročná na výkon hardware. Následující řádky se vztahují k výpočtu prováděném na GPU. Dříve než se pustíme do podrobností, musím zmínit, že popsané zkušenosti vycházejí z použité grafické karty NVIDIA GeForce 8600GT. V úvahu tedy musíme brát vylepšení a optimalizace architektury čipů při obecných výpočtech (GPGPU). U novějších řad grafických karet NVIDIA tak mohou být některé neduhy odstraněny a celkový výkon bude jistě několikanásobně vyšší. Pro ladění výkonu byl použit volně dostupný nástroj AgPerfMon. Díky robustnímu událostnímu systému PhysX máme k dispozici profilovací informace o všem, co se v naší scéně stalo.

Nyní si v krátkosti představíme ty nejdůležitější parametry fluidu. Shrnuje je tabulka 4.1.

Parametr	Zkratka	Popis
kernelRadiusMultiplier	KRM	Spolu s parametrem restParticlesPerMeter kontroluje rádius vlivu částice na ostatní částice fluidu.
restParticlesPerMeter	RPM	Kubická hodnota tohoto parametru definuje počet částic v krychlovém metru, při kterém fluid přechází do „odpočinku“, čili není prováděna jeho simulace, dokud se počet částic nezmění.
packetSizeMultiplier	PSM	Kontroluje velikost paketu, ve kterých probíhá simulace částic.
motionLimitMultiplier	MLM	Definuje maximální vzdálenost, kterou může částice urazit během jednoho kroku simulace relativně k ostatním částicím fluidu.
collisionDistanceMultiplier	CDM	Definuje vzdálenost mezi částicemi a kolizní geometrií kde dochází ke kolizi.

Tabulka 4.1 – Základní parametry fluidu

Jak už jsme si řekli v kapitole 3.2.2 Částicový systém - Fluidy, probíhá detekce kolizí částic s okolím nahráváním paketů kolizních dat v hrubé fázi. Seřazení tvarů scény je časově náročné. Nejhorším případem je častý vznik a zánik paketů ve scéně společně s jejich velkým počtem. To vede k velmi častému přepočítání hrubé fáze a výkon rapidně klesá. Pokud paket obsahuje příliš málo částic (příliš velký počet paketů) nebo příliš mnoho částic (příliš malý počet paketů), výkon zřetelně klesá [PHYSTIPS09]. Moje zkušenosti ale jasně říkají: čím méně paketů, tím lépe. Problémem je scénář, kdy do jednoho fluidu emituje nové částice mnoho emiterů současně. Pakety jsou totiž brány v úvahu pro každý emiter zvlášť. S každým novým emiterem tedy zákonitě vznikají nové pakety. Při ladění výkonu fluidů nebylo těžké dosáhnout většího počtu současně simulovaných částic, pokud byl použit jen jeden emiter. Dalším úskalím je prostorové rozložení částic a jejich rychlost. Pokud simulace probíhá na malém prostoru, nedochází k častému přepočítání hrubé fáze a výkon je skvělý. Když jsou částice rozprostřeny do velkého prostoru a jejich pozice se často a výrazně mění, je hrubá fáze přepočítána velmi často. Výkon náhle prudce klesá, a to i pokud emitujeme částice pouze jedním emiterem. Nejhorší případ nastává v kombinaci těchto scénářů. Technologické demo je toho důkazem. Na raketových střelách je vždy přichycený emiter a střela se spolu s ním rychle pohybuje (viz. následující kapitola 4.4 Entity). Emitované částice jsou rozprostřeny na velkém prostoru. Navíc

v jednu chvíli je aktivních i deset takových emiterů. Výsledkem je rapidní pokles výkonu. Režim simulace, tedy zda je zapnuta SPH simulace či nikoliv, nemá v tomto případě na výkon významný vliv, pokud není nastaven velký rádius vlivu částic na ostatní. Pak se úměrně zmenší velikost paketů a jejich počet se zvýší. Ladění výkonu by jistě vyžadovalo další testy. Řešením tak zůstalo snížení maximálního počtu simulovaných částic fluidu. Dle mé zkušenosti výkon simulace lineárně klesá s přibývajícím počtem částic až do doby, kdy GPU přetížíme. Výkon v této chvíli rapidně spadne a simulace částicového systému se stane úzkým hrdlem celého enginu.

Pro zajištění stabilního výkonu lze fluid nastavit do tzv. prioritního režimu. Prioritní režim limituje maximální počet současně simulovaných částic. Pokud by při emisi nových částic mělo dojít k překročení maximálního počtu, jsou nejstarší částice použity pro ty nové [PHYSXTIPS09]. Adekvátně k maximálnímu počtu částic musíme nastavit i emitery částicového efektu. Zajímají nás především parametry tempa (rate) emise, maximum emitovaných částic a životnost částic emiteru. Musíme si uvědomit, že pokud v každém snímku vytvoříme tři emitery, které budou chrlit tisíce nových částic za sekundu, pak velmi rychle dosáhneme maxima simulovaných částic fluidu. Ten začne staré částice odstraňovat (využívat pro nové) a může se tak stát, že nám před očima náhle vymizí třeba celý výbuch. To bude vypadat velmi nepřirozeně. Nabízí se dvě možná řešení. Můžeme zpomalit tempo vzniku nových částic, nebo snížit maximální počet emitovaných částic, a nebo jejich životnost. Vždy záleží na konkrétní situaci. Cílem je, aby se počet simulovaných částic pohyboval jen těsně nad stanoveným maximem a tyto artefakty tak nebyly pozorovatelné. Toto můžeme dále podpořit stanovením počtu rezervovaných částic. Maximum tak bude sníženo, ale nové částice se vždy objeví, aniž by způsobily zánik velkého počtu starých částic.

Hodnoty prametrů fluidů dema shrnuje tabulka 4.2 – Nastavení fluidů dema.

Fluid	Parametry						
	Maximální počet částic	Rezervovaný počet částic	KRM	RPM	PSM	MLM	CDM
Exploze	3000	900	1	10	128	8	0.12
Kouř	4000	1000	2	5	256	0.2	0.12
Trysky	600	120	1	30	256	1.9	0.12

Tabulka 4.2 – Nastavení fluidů dema

Všechny částicové efekty dema jsou nastaveny v prioritním režimu. Fluidy pro exploze a kouř za raketou jsou simulovány v SPH režimu. Fluid trysek má interakci částic vypnutu a chová se tak jako klasický částicový efekt. Stále ale probíhá detekce kolizí s objekty scény. Kouř má navíc nastavenu malou externí akceleraci, takže pomalu stoupá vzhůru. Fluidu nastavujeme spoustu dalších parametrů. Např. tuhost, viskozita, dynamické a statické tření, přilnavost, útlum rychlosti pohybu

částic, přitažlivá síla mezi částicemi a mnoho dalších. Pečlivým nastavováním parametrů jsme schopni dosáhnout nespočet efektů. Zároveň ale musím říci, že vyladění jen základních parametrů simulace vyžaduje opravdu hodně trpělivosti. Hodnoty základních parametrů emiterů fluidů dema shrnuje další tabulka 4.3 – Emitery fluidů dema.

Emiter	Parametry		
	Maximální počet částic	Tempo vzniku	Životnost částic
Exploze	100	100	1.2
Kouř	420	140	0.4
Trysky	150	30	0.8

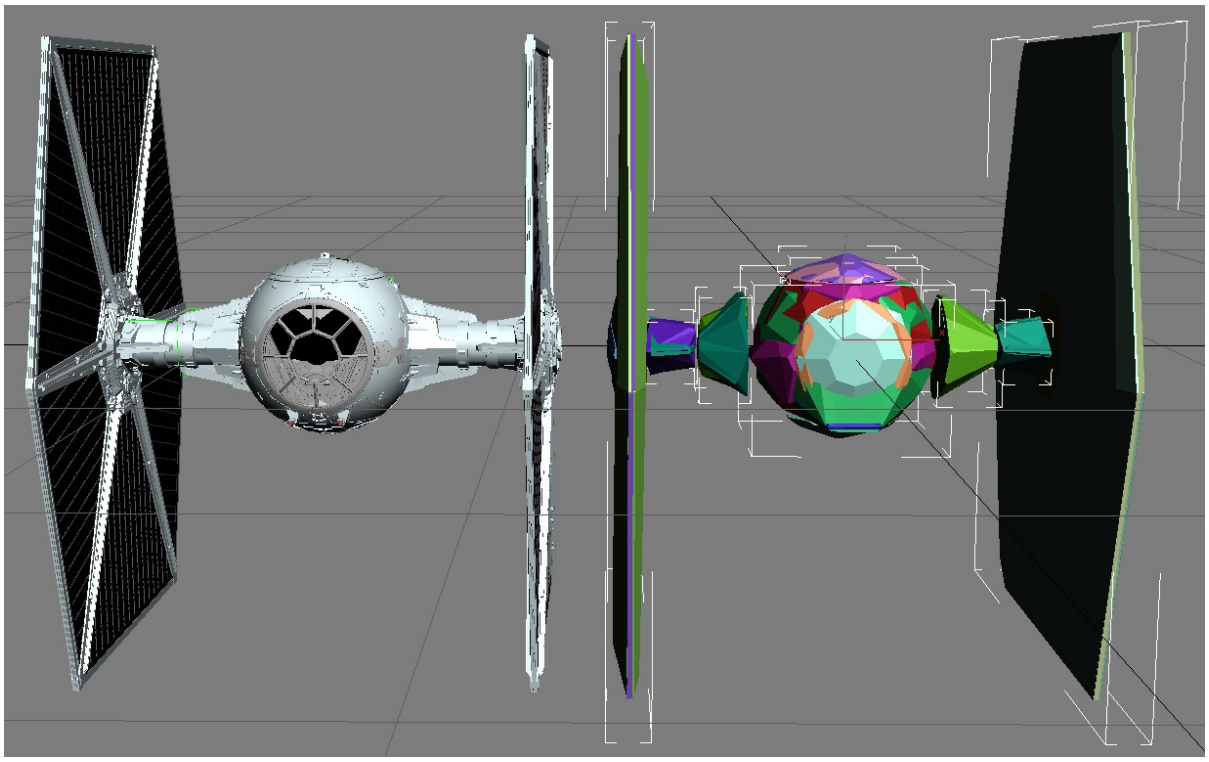
Tabulka 4.3 – Emitery fluidů dema

4.4 Entity

Fáze tvorby dat byla pro technologické demo realizována pomocí vyspělého prostředí Autodesk 3ds Max 2009 ve zkušební Trial verzi. Tímto nástrojem lze vytvořit grafickou reprezentaci. Fyzikální reprezentace byla vytvořena pomocí NVIDIA PhysX plug-in for 3ds MAX ve verzi 1.0.6. Modul dovoluje sestavit kompletní fyzikální popis pro PhysX včetně spojů, fluidů, silových polí a tento popis serializovat do NxuStream2 formátu. Fyzikální popisy všech entit byly pro účely studia ponechány v xml podobě NxuStream2 formátu a obsahují veškerá počáteční nastavení fyzikální reprezentace entity.

Věčným problémem při využití 3ds MAX spolu s OpenGL je souřadný systém. Studio používá levoruký, kdežto OpenGL pravoruký. Řešení se nabízí několik. Převod lze realizovat ve fázi předzpracování nebo ještě lépe vlastním plug-inem pro export dat ze 3ds MAX. Pro technologické demo jsem zvolil nejjednodušší řešení: otočit fyzikální reprezentaci entity o 90 stupňů okolo osy *x*. Transformace aktorů jsou použity i pro grafickou reprezentaci a výsledek je tak korektní.

Nejzajímavější entitou technologického dema je bezesporu Tie Fighter. Snažíme se dosáhnout zničitelné entity, která se při zásahu raketou rozpadne na menší části. Model entity vidíme na Obrázek 4.4 – Tie Fighter konfigurace. Bílé ohraničení fyzikálního modelu odpovídá jednotlivým aktorům. Tie Fighter tvoří celkem 9 aktorů. Aktory obsahují různý počet tvarů. Všechny tvary jsou konvexní trojúhelníkové obálky (convex hull) s maximálně 32 vrcholy, vytvořené zásuvným modulem PhysX. Fyzikální model je tedy zjednodušením grafického. Jistě by šlo použít obecnou trojúhelníkovou síť shodnou s grafickou reprezentací. Současná implementace PhysX ale nedetekuje kolize mezi dvěma takovými tvary. Jednotlivé aktory jsou propojené spoji. Budeme využívat možnosti přetržení spoje.



Obrázek 4.4 – Tie Fighter konfigurace (vlevo je grafická verze vpravo fyzikální)

Pro Tie Figter chceme použít pevný (fixní) spoj. PhysX plug-in podporuje „pouze“ 6DOF (NxD6Joint) spoj a ne pevný (NxFixedJoint). Uvozovky jsou namístě, protože 6DOF spojem jsme schopni modelovat všechny druhy spojů. U 6DOF spoje lze uzamknout pohyb a rotaci ve všech osách. Uzamčený 6DOF spoj se ale překvapivě nechová shodně jako pevný spoj. Někdy dochází k nechtěnému natahování a rotaci ve spoji. Řešením je převést takto nastavené spoje na fixní ve fázi předzpracování. MaxPhysXProcessor na to samozřejmě pamatuje.

Možná jste si všimli, že tvary aktorů byly upraveny, aby se tvary jednoho aktoru nepřekrývaly (ani nedotýkaly) s tvary jiného aktoru. Ale proč? Vždyť pokud by se překrývaly, můžeme mezi propojenými aktory vypnout detekci kolizí. Když se spoj přetrhne, přestane existovat nastavení kolizí mezi aktory spoje. To je logické a navíc i žádoucí. Určitě chceme, aby odtržená část entity mohla kolidovat se zbytkem entity. Když však se tvary aktorů překrývají, pak PhysX v okamžiku přetržení spoje okamžitě generuje kolize. Výsledkem je dost nepřirozené odmrštění aktorů od sebe. Tvary se tedy v našem případě překrývat nesmí.

Fyzikální model entity použitý v demu má k reálnému velmi daleko. Již samotný fakt, že jde o zjednodušení již zjednodušeného grafického modelu je toho důkazem. Vetší přiblížení k realitě by se dalo dosáhnout použitím tkanin (cloth) PhysX. Trojúhelníková síť tvarů by přesně odpovídala

grafickému modelu a navíc by byla prováděna simulace napěťových sil tvaru. Dosáhli bychom tak deformací a úlomků částí lodě.

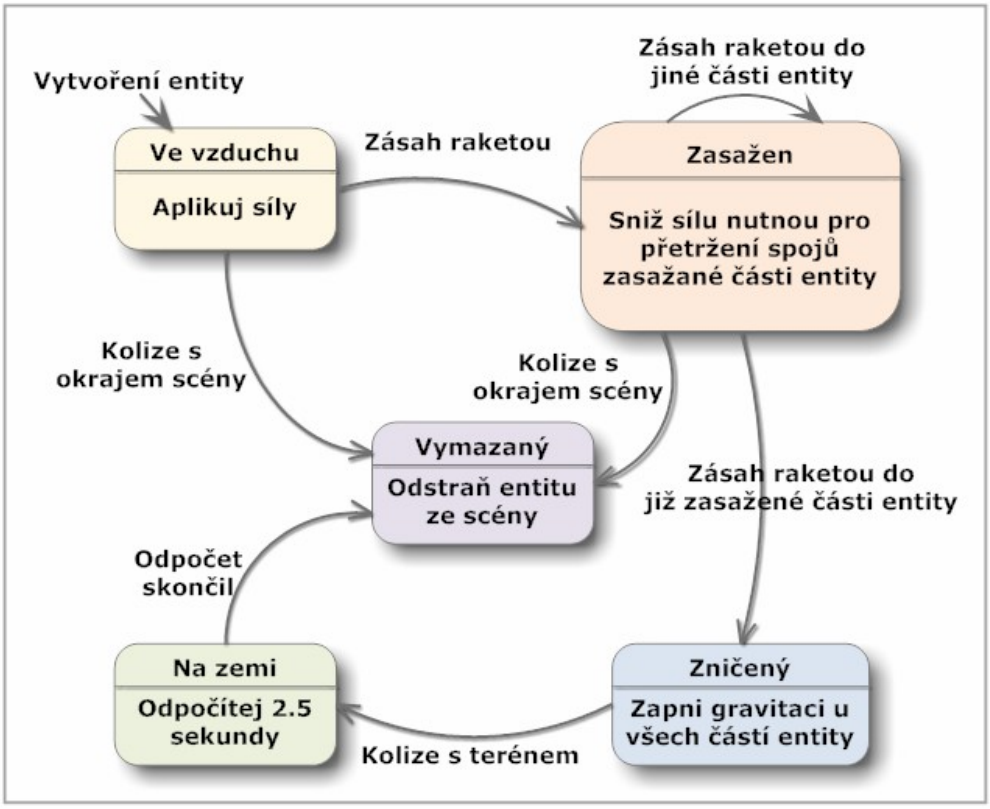
Pro vytvoření entit dema využijeme entitního systému.

```
m_pXwing = g_EntitySystem.CreateEntity( "Player", XWING );  //X-Wing
m_pXwing->SetWorldPosition(0, 20, -30);  //výchozí pozice

g_EntitySystem.CreateEntity( "Mars", MARS );  //Terén

CreateSceneBound();  //Fyzikální okraj scény
```

Chování entity vidíme na Obrázek 4.5 – Diagram chování entity Tie Fighter.



Obrázek 4.5 – Diagram chování entity Tie Fighter

Chování přiřazujeme v inicializaci entity viz. `DemoInitEntity::TieFighterInit()`. Tie Fighter se po vytvoření pohybuje podél záporné osy *z*. Všem částem entity nastavíme lineární a úhlové tlumení působících sil (linear, angular damping). Cílem je jednak simulace odporu vzduchu a také zamezení velké výchylky pohybu entity od své dráhy při zásahu raketou. Hodnoty jsou proto dost vysoké.

```
void DemoInitEntity::TieFighterInit(EntityPtr pEntity)
{
    //Pro všechny SubEntity
    pSub->GetActor()->setLinearDamping(5);      //Lineární útlum sil
    pSub->GetActor()->setAngularDamping(7);     //Úhlový útlum sil
    for(int j=0; j<pSub->GetJointCnt(); ++j)
    {
        pSub->GetJoint(j)->SetUnbreakable();
        pSub->GetJoint(j)->SetJointBehaviour(new OthersJointBehaviour);
    }
    //...
};
```

Pohyb entity realizujeme aplikací síly v každém snímku na všechny subentity (aktory). Každá část má nastavenou jinou hmotnost. Aplikované síly musí být pro různé části různě velké, aby výsledkem byla stejná rychlost částí. Sílu bychom v našem případě mohli aplikovat také pouze na prostřední část entity (díky souměrnosti entity okolo jejího středu) a vlastně jí tak táhnout požadovaným směrem jako za provaz. V tu chvíli se ale fixní spoje začnou trochu prohýbat, což nevypadá přirozeně.

Všechny spoje entity jsou inicializovány jako nepřetrhnutelné. Toho dosáhneme nastavením síly nutné pro přetržení na hodnotu `NX_MAX_REAL`.

```
void PhysXJoint::SetUnbreakable()
{
    m_pNxJoint->setBreakable( NX_MAX_REAL, NX_MAX_REAL );
};
```

Aktory entity mají po inicializaci vypnutou gravitaci. Při přetržení spoje, dalším zásahem raketové střely, gravitaci opět zapneme a musíme také snížit lineární a úhlový útlum. Entitě odstraníme její celkové chování, takže již nejsou aplikovány síly pohybu a části entity padají k zemi. Návratovou hodnotou řídíme, zda bude přetržený spoj odstraněn či nikoliv.

```
bool OthersJointBehaviour::OnBreak()
{
    //...
    Engine::Entity* p = m_pOwner->GetFirstSubEntity()->GetEntity();
    p->SetEntityBehaviour( NULL );
    //Pro Všechny SubEntity
    pSub->EnableGravity();
    pSub->GetActor()->setAngularDamping( 1.f);
    pSub->GetActor()->setLinearDamping( 0.05f );
    //...
};
```

Jakmile Tie Fighter dopadne na terén, nastavíme nové celkové chování entity, které po dvou a půl sekundách entitu kompletně odstraní ze všech systémů.

Destruktivní chování entity lze dále vylepšit. Entita se rozpadá na předem vytvořené aktory. Ty ale stále obsahují více než jeden tvar. Aktor lze např. při dopadu na terén dále fragmentovat oddělením jeho tvarů a docílit tak rozpadu části entity na ještě menší kousky. V principu bychom při zpracování kolizní události museli zasažený tvar aktoru uložit do deskriptoru a tento deskriptor předat deskriptoru nového, fragmentačnímu aktoru. U části entity bychom pak byli nuceni přepočítat hmotnost a její těžiště. Časté zakládání většího počtu nových aktorů by ale způsobilo přepočítání seznamu potencionálně kolidujících objektů (broad phase). Řešením je vytvořit pro všechny tvary entity aktory předem s tím, že by u nich byla vypnuta kolizní reakce. Také bychom museli nutně aktualizovat jejich pozice, aby odpovídaly pozicím tvarů entity. Popsaný princip je v C3D implementován, ale není odladěn a tak není použit.

Fyzikální popis hráčovy lodě (X-Wing) je velmi podobný jako u entity Tie-Fighter. Tvoří ji deset aktorů propojených pevnými (fixními) spoji. Spoje jsou při inicializaci v metodě `DemoInitEntity::XWingInit()` opět nastavené na nepřetržitelné, avšak kolize nevyvolá žádnou změnu. Loď se tak chová jako nezničitelná. Pohyb enitity je opět realizovaný opakovaným aplikováním sil ve směru pohybu.

V inicializaci zbývá nastavit efekt „ohně“ vycházejícího z trysek raketových motorů lodě. Nejdříve získáme deskriptor nastavení emiteru.

```
//Získej zdroj s popisem efektu a emiteru
NXU::NxuPhysicsCollection* pColl =
        g_ResourceManager.GetPhysics("fluids\\xwing_noozle_fluid.xml");
//Získej descriptor emiteru
Physics::EmitterDesc desc;
pColl->mScenes[0]->mFluids[0]->mEmitters[0]->copyTo(desc,
                                                    NXU::CustomCopy());
```

Deskriptorem vytvoříme emiter. K tomu musíme získat patřičný částicový efekt. Emiteru přiřadíme chovaní.

```
//Získej efekt (fluid) ohně motorů
ParticleEffectPtr pEff = g_ParticleSystem.GetEffect("XWing_Noozle");
//Vytvoř emiter
IEmitterPtr pEmitLU = pEff->CreateEmitter( desc );
//Chování
pEmitLU->SetEmitterBehaviour( new XWingNozzleEmitterBehaviour );
```

Chování emiteru reaguje na událost fluidu PhysX. Fluidní události jsou generovány vždy, když je dosaženo maximálního počtu částic fluidu nebo emiteru [PHYSX08]. Pro fluid to znamená, že v tu chvíli probíhá simulace nastaveného maximálního počtu částic. Pro emiter je význam odlišný. Znamená, že byl emitován pro daný emiter nastavený maximální počet částic. Ne všechny částice jsou v tu chvíli simulovány. Některé již zanikly, protože vypršel jejich čas života. V důsledku události

přestane emiter vytvářet nové částice. Pokud chceme, aby emiter vytvářel nové částice nepřetržitě, musíme při obsluze události emiteru vynulovat (resetovat) jeho rezervoár (zásobník).

```
bool XWingNozzleEmitterBehaviour::OnMaxParticlesEmitted()
{
    //Vynuluj reservoár emiteru
    ((Emitter*)m_pOwner)->GetNxEmitter()->resetEmission(100);
    return false;
}
```

Návratovou hodnotou specifikujeme, zda má být emiter po obsluze události odstraněn ze systému (true) nebo ne (false). Emiter je vytvořený a nastavený. Zbývá ho přichytit k motoru lodi a spustit emisi částic.

```
IEmitterPtr pEmitLU = ...
//Získej patřičnou část entity
SubEntityPtr pSubLU = pEntity->GetSubEntity("LU_Wing");
C3D_ASSERT(pSubLU,"");
//Připoj emiter
pSubLU->AttachEmitter( pEmitLU, "LU_Nozzle_Shape" );
pEmitLU->Start();  //Spusť emisi
```

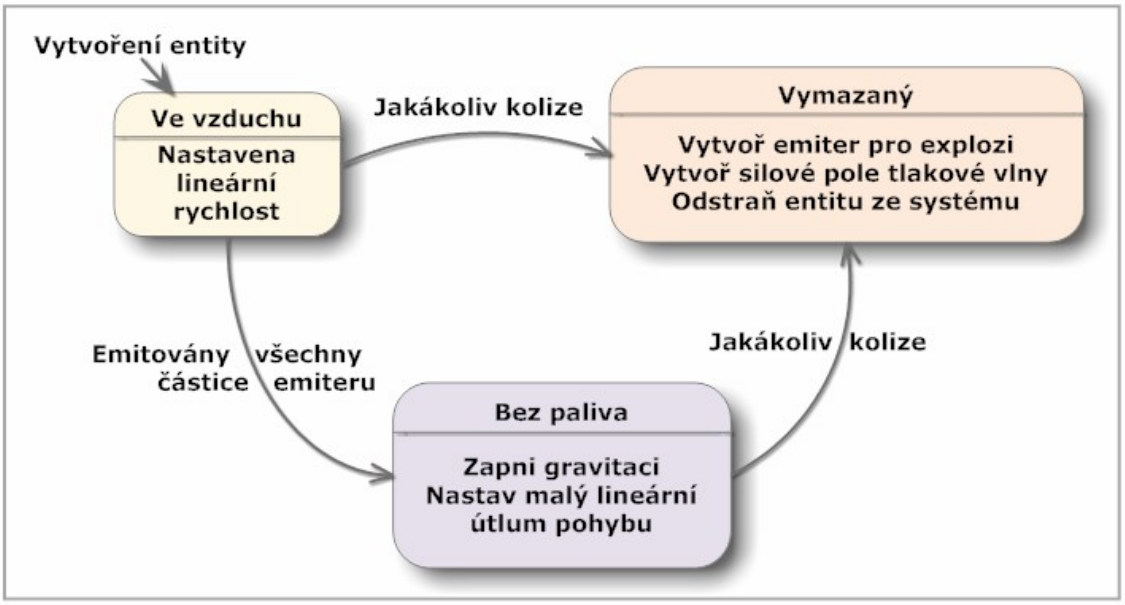
Hráč stiskem mezerníku střílí raketové střely. Samotná střelba pouze vytváří novou raketu a nastaví ji do správné pozice relativně k lodi hráče (viz. `Demo::Fire()`). Střelu fyzikálně reprezentuje jeden aktor s tvarem kvádru. Protože střela nemá nastavený žádný útlum pohybu, můžeme její pohyb vesmírem realizovat nenulovou lineární rychlostí. Hodnotu nastavujeme tělu aktoru přes vlastnost linearVelocity (`NxBodyDesc::linearVelocity`). Vytvoření střely zahrnuje vznik emiteru pro částicový efekt (fluid) kouře za střelou (viz. `DemoInitEntity::RocketInit()`). Kód je velice podobný tvorbě emiterů pro motory entity X-Wing. Rozdíl je v chování emiteru střely. Po emitování nastaveného maxima částic střele vynulujeme lineární rychlost a zapneme gravitaci spolu s velmi malým útlumem pohybu.

```
bool RocketEmitterBehaviour::OnMaxParticlesEmitted()
{
    SubEntityPtr pSub = m_pOwner->GetAttachedSubEntity();
    pSub->EnableGravity();
    pSub->GetActor()->setLinearDamping(0.5);

    return IEmitterBehaviour::OnMaxParticlesEmitted();  //return true
}
```

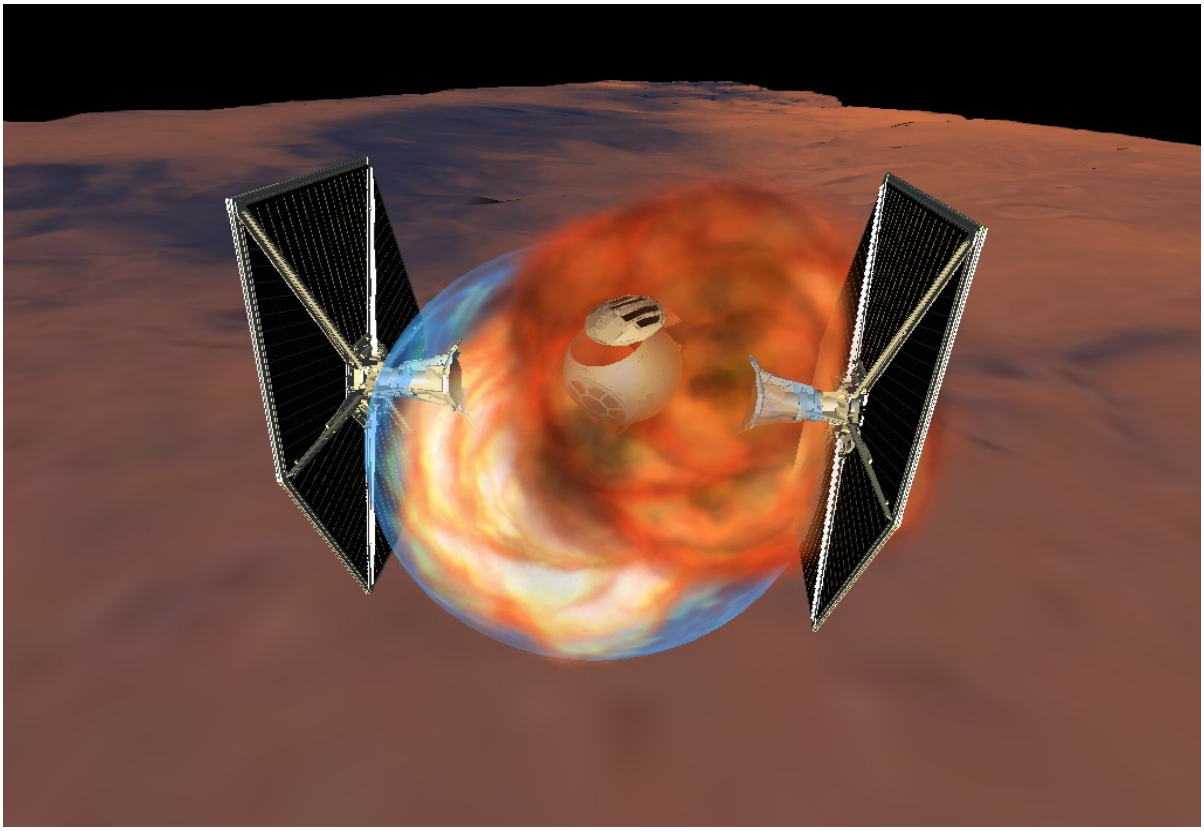
U střely chceme jistě reagovat na vzniklé kolize. Protože bude střela při jakékoli kolizi odstraněna z herního světa, nezajímají nás kolizní události generované při vnitřním dotyku ani při konci kolize. Aktoru tedy povolíme ve vlastnosti NxActorDescBase::contactReportFlag pouze příznak `NX_NOTIFY_ON_START_TOUCH`. V obsluze kolizní události opět vytvoříme emiter, tentokrát pro

částicový efekt exploze a také vytvoříme silové pole simulující tlakovou vlnu. Celé chování raketové střely vidíme na obrázku 4.6 – Diagram chování raketové střely.



Obrázek 4.6 – Diagram chování raketové střely

Celou situaci vidíme na obrázku 4.7 – Explodující Tie Fighter.



Obrázek 4.7 – Expolující Tie Fighter

Terén modeluje demo statickým aktorem s jediným tvarem. Tvar je typu obecné trojúhelníkové sítě (NxTriangleMeshShape) a tvoří ho zhruba padesát tisíc trojúhelníků.

Okraj scény definuje demo pouze fyzikální reprezentací. Jde o statický aktor s jediným tvarem typu kvádr (`NxBoxShape`). Úkolem okraje je detekce entity, která svojí pozicí opouští herní svět. Půjde především o entitu Tie Fighter. Pokud ta svým pohybem překročí okraje scény, můžeme ji z herního světa odstranit. Pro tyto účely lze jakýkoliv tvar aktoru nastavit jako spínač (trigger). Při vstupu, výstupu nebo setrvání jiného tvaru ve tvaru nastaveném jako spínač je generována patřičná událost. Tvar vytvoříme tentokrát ručně přímo aplikačním kódem:

```
//Vytvoř tvar (box) okraje scény
NxBoxShapeDesc boxDesc;
boxDesc.dimensions = NxVec3(400, 76, 2100);//Velikost
boxDesc.localPose.t.set(0, 0, -2100);//Lokální pozice vůči aktorovi
//Nastav generování události při opuštění tvaru
boxDesc.shapeFlags |= NX_TRIGGER_ON_LEAVE;
```

5 Závěr

Prostudoval a popsal jsem architekturu herních enginů a nejpoužívanější nástroje pro fyzikální simulace ve hrách. Zhodnocena je také jejich realističnost. Navrhl jsem a implementoval herní engine, který podporuje fyzikální simulace pevných těles, spojů, silových polí a částicových systémů (fluidů) pomocí knihovny PhysX. Výstupem projektu je, kromě C3D enginu, implementované technologické demo, které vhodně demonstruje všechny vybrané techniky formou interaktivní aplikace. Prezentovaný postup tvorby zničitelných herních entit účinně kombinuje simulaci pevných těles a spojů. Spolu s takřka neomezenými možnostmi definice chování entity výrazně přispívá k dynamičnosti herního světa. Částicové efekty dosahují díky simulaci na GPU značně velkých počtů částic. Realističnost je dále podpořena výpočtem interakcí částic a kolizemi částic s okolím. O kvalitní vykreslení efektů se stará vylepšená technika animovaných měkkých částic implementovaná CgFx shaderem. Celý částicový systém je tak připraven k okamžitému použití v herních projektech.

Literatura

[CANTLAY07] CANTLAY, Iain. *GPU Gems 3 : 3D and General Programming Techniques for GPUs* [online]. Kendallville (Indiana) : Addison-Wesley, December 2007 [cit. 2010-05-18]. Chapter 23. High-Speed, Off-Screen Particles, s. . Dostupné z WWW: <http://http.developer.nvidia.com/GPUGems3/gpugems3_ch23.html>.

[DELOURA01] DELOURA, A. Mark. *Game programming gems 2*. 1st edition. Hingham : Charles River Media, 2001. 575 s. ISBN 1-58450-054-9.

[DELOURA00] DELOURA, A. Mark. *Game programming gems*. 1st edition. Hingham : Charles River Media, 2000. 614 s. ISBN 1-58450-049-2.

[DICKHEISER06] DICKHEISER, Michael. *Game programming gems 6*. Boston (Massachussets) : Charles River Media, 2006. 695 s. ISBN 1-584-50450-1

[EBERLY05] EBERLY, H. David. *3D game engine architecture : engineering real-time applications with wild magic*. 1st edition. Boston : Morgan Kaufmann Publishers, 2005. 735 s. ISBN 0-12-229064-X.

[ENGINE10] *Wikipedia* [online]. 18.01.2003, 20.05.2010 [cit. 2010-05-23]. Game engine. Dostupné z WWW: <http://en.wikipedia.org/wiki/Game_engine>.

[GABB–LAKE05] GABB, Henry; LAKE, Adam. *Gamasutra* [online]. November 17, 2005 [cit. 2010-05-16]. Threading 3D Game Engine Basics. Dostupné z WWW: <http://www.gamasutra.com/features/20051117/gabb_01.shtml>.

[GAMEWIKI10] *The Game Programming Wiki* [online]. 14.10.2004, 17.05.2010 [cit. 2010-05-23]. Game Engines. Dostupné z WWW: <http://gpwiki.org/index.php/Game_Engines>.

[GAMMA95] GAMMA, Erich. *Design patterns :elements of reusable object-oriented software*. 1st edition. Massachusetts : Addison-Wesley, 1995. 395 s. ISBN 0201633612.

[GPUGUIDE08] *GPU Programming Guide : GeForce 8 and 9 Series* [online]. [s.l.] : NVIDIA Corporation, December 19, 2008 [cit. 2010-05-23]. Dostupné z WWW: <http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide_G80.pdf>.

[HAVOK10] *Havok* [online]. c2010 [cit. 2010-05-23]. Dostupné z WWW: <http://www.havok.com/>.

[KIRMSE04] KIRMSE, Andrew. *Game programming gems 4*. 1st edition. Hingham : Charles River Media, 2004. 703 s. ISBN 1-58450-295-9.

[LATTA04] LATTA, Lutz. *Gamasutra* [online]. 28.07.2004 [cit. 2010-05-23]. Building a Million-Particle System. Dostupné z WWW: <http://www.gamasutra.com/features/20040728/latta_01.shtml>.

[LORACH07] LORACH, Tristan. *NVIDIA Developer* [online]. Verze 1. 2007-01-17 [cit. 2010-03-29]. Soft Particles. Dostupné z WWW: <http://developer.download.nvidia.com/whitepapers/2007/SDK10/SoftParticles_hi.pdf>.

[LOWENSOHN10] LOWENSOHN, Josh. *Cnet* [online]. March 9, 2010 [cit. 2010-05-23]. How Epic fit the Unreal Engine into the iPhone. Dostupné z WWW: <http://news.cnet.com/8301-27076_3-20000214-248.html>.

[MCSHAFFRY03] MCSHAFFRY, Mike. *Game Coding Complete*. Scottsdale (Arizona) : Paraglyph Press, Inc., 2003. Loading and Caching Game Resources, s. 251-283. ISBN 1-932111-75-1.

[MÜLLER07] *Matthias Müller-Fischer* [online]. 2007 [cit. 2010-04-19]. Real Time Fluids in Games. Dostupné z WWW: <http://www.matthiasmueller.info/talks/gameFluids2007.pdf>.

[PARSYSTEM10] *Wikipedia* [online]. 05.12.2003, 15.04.2010 [cit. 2010-05-23]. Particle system. Dostupné z WWW: <http://en.wikipedia.org/wiki/Particle_system>.

[PHYSX08] PhysX documentation. *PhysX Documentation* [online]. 2008 [cit. 2009-05-25]. Dostupný z WWW: <http://developer.download.nvidia.com/PhysX/2.8.1/PhysX_SDK_2.8.1_build_13_for_PC.msi>.

[PHYSXTIPS09] *NVIDIA Developer Zone* [online]. 18.3.2009 [cit. 2010-04-02]. PhysX Tips. Dostupné z WWW: <http://developer.nvidia.com/object/physx_tips.html>.

[PLUMMER04] PLUMMER, Jeff. *A flexible and expandable architecture for computer games*. Tempe, Arizona, 2004. 382 s. Diplomová práce. Arizona State University. Dostupné z WWW: <http://www.gamecareerguide.com/education/theses/20051018/plummer_thesis.pdf>.

[SCENEGRAPH10] *Wikipedia* [online]. 27.12.2002, 02.03.2010 [cit. 2010-05-23]. Scene Graph. Dostupné z WWW: <http://en.wikipedia.org/wiki/Scene_graph>.

[SCHIRM–HARRIS10] SCHIRM, Simon; HARRIS, Mark. *Game Programming Gems 8*. Boston (Massachussets) : Course Technology, a part of Cengage Learning, 2010. Fast GPU Fluid Simulation in PhysX, s. 529-541. ISBN 1-58450-702-0.

[SHISHKOVTSOV05] SHISHKOVTSOV, Oles. *GPU Gems 2 : programming techniques for high-performance graphics and general-purpose computation* [online]. Taunton (Massachusetts) : Addison-Wesley, April 2005 [cit. 2010-05-18]. Chapter 9. Deferred Shading in S.T.A.L.K.E.R, s. . Dostupné z WWW: <http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter09.html>.

[SOFTPAR10] *Windows Developer Center* [online]. 2010 [cit. 2010-03-29]. SoftParticles Sample. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ee416430(VS.85).aspx>.

[WRIGHT–LIPCHAK05] WRIGHT, Richard S.; LIPCHAK, Benjamin. *OpenGL Superbible*. 3rd edition. Indianapolis : Sams Publishing, 2005. 1173 s. ISBN 0-672-32601-9.

Seznam příloh

Příloha 3. Knihovny použité při implementaci projektu

Knihovna	Verze	Popis
NVIDIA PhysX SDK	2.8.1	Fyzikální engine. http://developer.nvidia.com/object/physx_downloads.html
Generic Math Template Library (Gmtl)	0.6.0	Matematická knihovna speciálně navržená pro maximální výkon v interaktivních aplikacích. http://ggt.sourceforge.net/
Open Asset Import Library (Assimp)	1.0	Knihovna pro konverzi grafických dat pro herní enginu. http://assimp.sourceforge.net/
Simple DirectMedia Layer (SDL)	1.2.14	Multimediální nízkoúrovňová knihovna. http://www.libsdl.org/
Cg toolkit	2.2	Knihovna pro shadery v jazyce Cg. http://developer.nvidia.com/object/cg_toolkit.html
The OpenGL Extension Wrangler Library (GLEW)	1.5.1	Poskytuje dynamický mechanismus pro zjištění podporované verze OpenGL a jeho rozšíření. http://glew.sourceforge.net/
Developer's Image Library (DevIL)	1.7.8	Knihovna pro nahrávání obrovského počtu obrazových formátů a práci s nimy. http://openil.sourceforge.net/
boost	1.41.0	Zdrojové knihovny pro široké spektrum použití. http://www.boost.org/
OpenGL	2.0	Grafické API. http://www.opengl.org/
Object Oriented Input System (OIS)	1.2.0	Přístup k všem druhům vstupních zařízení. http://sourceforge.net/projects/wgois/
Templatized C++ Command Line Parser Library (TCLAP)	1.2.0	Malá, flexibilní knihovna pro definici a přístup k parametrům příkazové řádky. http://tclap.sourceforge.net/
TinyXML++ (ticpp)	NA	Objektová nadstavba knihovny TinyXML pro práci s XML soubory. http://code.google.com/p/ticpp/