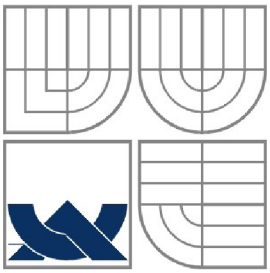
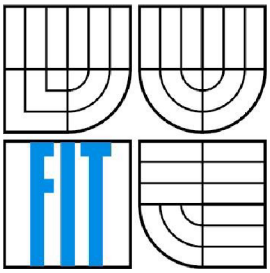


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

FYZIKÁLNÍ SIMULACE VE VIRTUÁLNÍ REALITĚ

PHYSICAL SIMULATION IN VIRTUAL REALITY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

VOJTĚCH GRÜNSEISEN

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN PEČIVA, Ph.D.

BRNO 2010

Abstrakt

Tato práce popisuje použití knihoven SDL, OSG a ODE jako nástrojů pro tvorbu her a jejich integraci do herního enginu. Je popsána obecná teorie o detekci kolizí a její role ve fyzikálním enginu. Použité knihovny jsou taktéž popsány. S použitím vyvinutého enginu je vytvořena a řízena demonstrační herní scéna obsahující tělesa pospojovaná ODE Jointy. Obsah scény a ovládání aplikace úmyslně připomíná styl first person shooter her.

Abstract

This work describes using of SDL, OSG and ODE as tools in game development and how they can be integrated to work together. Theory about collision detection and its role in physics engine is also described. Used libraries are described as well. With this engine a game scene is created and driven. This scene contains bodies connected by ODE Joints. Scene content and application controls are meant to resemble first person shooter game style.

Klíčová slova

herní engine, fyzikální engine, fyzikální simulace, simulační cyklus, ODE, OSG, SDL, detekce kolizí, kolizní odezva, tunelování, optimalizace kolizní detekce, broadphase, narrowphase , pospojovaná tuhá tělesa, herní kamera, 6DOF kamera, graf scény, HUD, herní model vzpřímeného hráčského avatara, herní model auta

Keywords

game engine, physics engine, physical simulation, simulation loop, ODE, OSG, SDL, collision detection, collision response, collision detection optimalization, broadphase, narrowphase, articulated rigid bodies, game camera, 6DOF camera, scene graph, HUD, upright player avatar game model, car game model

Citace

Vojtěch Grünseisen: Fyzikální simulace ve virtuální realitě, bakalářská práce, Brno, FIT VUT v Brně, 2010

Fyzikální simulace ve virtuální realitě

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. J. Pečivy, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Vojtěch Grünseisen
12.5.2010

Poděkování

Zde si dovoluji poděkovat panu Ing. J. Pečivovi, Ph.D. za jeho odbornou pomoc a cenné rady. Dále bych rád poděkoval vývojářům knihoven SDL, OSG a ODE za skvělou práci, kterou provádějí.

Obsah

1 Úvod

Cílem této bakalářské práce bylo spojit funkčnost knihoven OpenSceneGraph (dále jen OSG) a OpenDynamicsEngine (dále jen ODE) a vytvořit aplikaci demonstrující schopnosti obou knihoven. ODE je nejčastěji používáno k tvorbě počítačových her, proto i aplikace přiložená k tomuto dokumentu bude mít formu jednoduché hry. Herní engine pohánějící tuto hru bude taktéž navržen a vyvinut v průběhu této práce.

Při popisu použitých nástrojů (kapitola 3) bude nejvíce prostoru věnováno knihovně ODE, která je z použitých knihoven nejméně zdokumentovaná a je třeba ji přiblížit jejím případným budoucím uživatelům.

Herní scéna bude obsahovat objekty různých tvarů a velikostí. Tyto objekty spolu budou kolidovat a kolize budou měnit jejich pohybovou a rotační rychlost pomocí silových impulzů. Bude implementováno tření, závislé na materiálu kolidujících objektů.

Uživatel bude moci ve scéně ovládat některý z objektů, v takovém případě bude kamera následovat tento objekt. Kamerou bude možno hýbat také samostatně, jako se 6DOF(six degrees of freedom) kamerou. V tomto případě nebude kolidovat a nebude ovlivňována gravitací.

2 Teorie

2.1 Fyzikální engine (Physics engine)

Fyzikální engine je aplikační součást neboli modul, který v počítačové aplikaci simuluje fyzikální jevy. Jejich simulace je nutná v případě, že budujeme vědeckou nebo průmyslovou aplikaci a jde nám hlavně o přesnost výsledků. Při tvorbě interaktivních aplikací či her můžeme začlenít fyzikální engine do naší aplikace, a tím zlepšit její věrohodnost či hratelnost. Podle informací z Wiki [1] je nutné při výběru fyzikálního enginu zvolit kompromis mezi přesností a rychlostí. Přesné fyzikální enginy se zpravidla označují jako high-precision. Simulace jevů v takovém enginu trvá déle než jev samotný. To nám nevadí při simulaci vědeckého pokusu, ale pro simulaci fyziky v počítačových hrách jsou high-precision systémy nepoužitelné.

K simulacím v počítačových hrách musíme použít real-time fyzikální engine. Je tedy nutné, aby reakce na uživatelovu akci byla odsimulována téměř ihned. Více než o přesnost záleží na uvěřitelnosti a stabilitě. Implementací real-time fyzikálních enginů existuje celá řada a to jak open-source (ODE, Bullet), tak proprietárních (Havok, PhysX, CryEngine, Euphoria). Vždy je třeba volit systém (engine) podle požadavků cílové aplikace. Pokud nám stačí simulovat kinematiku, dynamiku a kolize tuhých těles, můžeme sáhnout po téměř kterémkoliv systému. Pokud vyžadujeme simulaci měkkých těles, látek a kapalin, potom nejspíše zvolíme Nvidia PhysX u kterého je navíc předpoklad, že výpočty budou hardwarově akcelerována na grafických kartách Nvidia. Open source real-time systémem, který prohlašuje[2], že podporuje simulaci měkkých těles je Bullet. ODE bohužel zvládá pouze simulace pospojovaných tuhých těles. Každý fyzikální engine typicky obsahuje integrátor, kterým aktualizuje stavy svých těles a detektor kolizí.

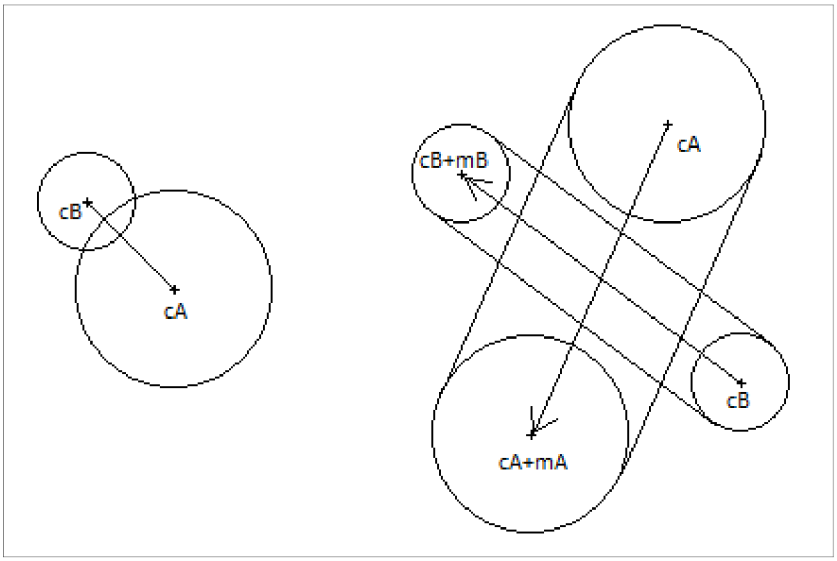
Integrátor může být v základu explicitní nebo implicitní. Ovšem například ODE používá kombinaci obou přístupů, viz ODE Wiki[3], sekce FAQ. Explicitní integrátor je obecně rychlejší, ale méně stabilní. U implicitního integrátoru je počítáno s tím, že se tělesa navzájem ovlivňují. Informace na toto téma jsem nalezl na gamedev.net fóru [4]. Dalším důležitým parametrem integrátoru je jeho řád (order). Nepřesnost simulací v ODE je také dána tím, že používá Eulerův integrátor pouze prvního řádu.

2.2 Detekce kolizí (Collision detection)

Většina informací o detekci kolizí byla získána z knihy [5], některé důležité informace pak také ze slidů [6]. Detektor kolizí tvoří nezbytnou část každého fyzikálního systému. Z principu se

ovšem jedná o součást, kterou lze použít i samostatně, tedy v prostředí bez fyzikální simulace (např. editor scény, který detekuje kolize a zabraňuje uživateli umístit objekty přes sebe).

Detektor kolizí typicky obsahuje sadu algoritmů pro testy mezi dvojicemi geometrických primitiv. Takovéto testy mohou být dvojího druhu. Buďto jednoduché, které pouze testují zda se objekty A a B nepřekrývají (overlap-test) nebo složitější, které se snaží přesně spočítat čas kolize (TOI – time of impact), pomocí sunutí (sweep-test). Pro názornost rozdílu následuje obrázek pro detekci kolize mezi dvojicí koulí ve 2D (obrázek 1).



Obrázek 1: overlap-test (vlevo) a sweep-test (vpravo) koulí ve 2D

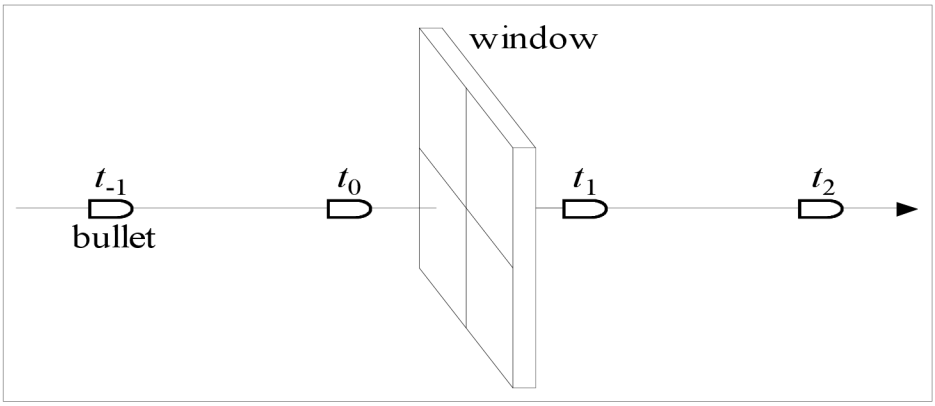
Už pro tento nejjednodušší případ kolize dvou koulí je overlap-testing mnohem rychlejší, pouze se porovná vzdálenost mezi středy koulí se součtem poloměrů. Kdežto sweep-test počitá výpočetně dražší overlap-test mezi dvěma kapslemi (obr. 1). U složitějších primitiv je výkonnostní rozdíl ještě větší. U hodně složitých geometrií jsou takové testy v reálném čase prakticky nevypočitatelné.

Fyzikální engine, jehož detektor kolizí používá sweep-testy je označován jako engine se spojitou detekcí kolizí (continuous collision detection). Kolize jsou předpovězeny předtím (a-priori) než nastanou a systém s nimi může počítat v následujícím simulačním kroku. Tento postup je doménou high-precision fyzikálních enginu. Z real-time fyzikálních enginů by měl spojitou detekci kolizí podporovat Bullet, alespoň podle informací na Wiki[2].

Naopak, pokud engine detekuje kolize pomocí overlap testů, jedná se o engine s diskrétní detekcí kolizí (discrete collision detection). Takto řeší kolize nejčastěji real-time fyzikální enginy. Diskrétní detekce funguje tak, že systém se nejdříve integruje do nového stavu a až potom (a- posteriori) zjistí, jestli se nějaké objekty nepřekrývají. Pokud tomu tak je, systém se pokusí o nápravu.

2.2.1 Tunelování (Tunnelling)

Zdrojem informací o tunelování byly taktéž slidy [6]. Při a-posteriori kolizní detekcí je simulací do systému zanešena chyba a pokud se objekty pohybovaly dostatečně rychle na to, aby v jednom simulačním kroku prošly zcela skrz sebe a při detekci kolizí už se nepřekrývají, tak chyba nebude odhalena. Tomuto jevu se říká tunelování (tunnelling – obr. 2).



Obrázek 2: Tunnelling, obrázek převzat z [6]

Pokud chceme v našem systému zamezit tunnellingu, je třeba si hlídat aby velikost rychlosti nejrychlejšího objektu (v_{max}) ve scéně za simulační krok (dt) byla menší, než tloušťka nejmenšího objektu (s_{min}):

$$v_{max} \cdot dt < s_{min}$$

Vzorec 1: prevence tunnellingu

Toto řešení nám ovšem mnohdy příliš nevyhovuje. Chceme většinou simulovat malé a rychlé objekty (například rychle letící vystřelené projektily). V takovýchto situacích se inspirujeme sweep-testem a pokusíme se spojitý pohyb objektu aproximovat jedním či více vlečnými paprsky. Paprsky musí být vystřelovány každý simulační krok z místa kde se objekt nacházel v minulém kroku do aktuální pozice. Ovšem i těleso na paprsku může protunelovat. Pokud bude cílová geometrie děravá, paprsek dírou projde, i když těleso by projít nemělo. Paprsek také nedetekuje okrajové kolize, v případě, že jej vystřelujeme ze středu objektu atd.

2.2.2 Optimalizace detekce kolizí

Často je potřeba simulovat rozsáhlé scény, ve kterých mezi sebou mohou potencionálně kolidovat všechny objekty. To znamená, že detektor kolizí by po každém simulačním kroku a pro každý objekt ve scéně volal overlap-test se všemi ostatními objekty. Tento přístup, který má časovou

složitost O (n^2) by se vyplatil pouze tehdy, pokud by opravdu všechny objekty mezi sebou kolidovaly. Takový scénář ve většině simulovaných scén nenastane s pravděpodobností hraničící s jistotou.

Druhým a mnohem méně naivním přístupem je detekci kolizí rozfázovat. V první fázi (broad-phase), se snažíme pomocí série výpočetně nenáročných testů zúžit počet potencionálně kolidujících párů na minimum, pro tyto pak ve druhé fázi (narrow-phase) provedeme výpočetně drahé overlap-testy.

Jednoduchou optimalizací, která se používá nejen pro detekci kolizí, ale také např. při renderování, raytracingu atd. , je obalit každý objekt obálkou (bounding volume) jednoduchého tvaru. Mezi nejčastěji používané obálky patří: BS (bounding sphere), AABB (axis aligned bounding box) a OBB (oriented bounding box). Z nichž BS nabízí nejrychlejší test a má tu výhodu, že je invariantní vůči rotacím. Každému overlap-testu potom bude předcházet jednoduchý test mezi obálkami kolidujícího páru objektů. Tento test se občas označuje jako mid-phase, nejedná se tedy ještě o broad phase optimalizaci.

Metody pro broadphase optimalizaci se většinou také nazývají metody prostorové organizace nebo také metody dělení prostoru (spatial partitioning methods). Takovýchto metod je celá řada a v praxi se většinou používá kombinace hned několika z nich. Vhodná kombinace metod nám pomůže se přiblížit ideální časové náročnosti O (n). Jen pro úplnost následuje výpis nejpoužívanějších metod: bounding volume hierarchies[7], sweep and prune[8], spatial hash tables, multilevel grids, quadtree.

2.3 Kolizní odezva (Collision response)

Po úspěšném detekování kolize je třeba tuto kolizi nějak obsloužit. Pokud nepoužíváme fyzikální engine, můžeme od sebe objekty odsunout. Neznáme-li hloubku zanoření, můžeme ji přibližne zjistit několika iteracemi bisekce.

Jestliže odsun provedeme ve směru opačném ke směru jejich pohybu, docílíme tak iluzi neklouzavého povrchu. Provedeme-li ho ve směru normály ke kontaktní rovině, simulujeme tak jednoduché klouzání (sliding). Tento přístup jsem si vyzkoušel v začátcích řešení této práce. Předtím, než jsem pronikl do ODE.

Pokud vyžadujeme nějaké reálnější odezvy - změny rychlostí, rotace atd., potom je výhodné využít služeb fyzikálního enginu.

3 Použité knihovny

Už při zadání práce byla pro zobrazení scény zvolena knihovna OSG[9] a pro simulaci objektů knihovna ODE[10]. Zbývalo vybrat vhodný GUI toolkit pro komunikaci s operačním systémem, vytvoření okna a zpracování událostí od uživatele. Hlavním požadavkem bylo, aby toolkit byl multiplatformní a tím spolu s multiplatformností OSG a ODE dotvořil multiplatformnost celé aplikace. Na stránkách OSG jsou k dispozici odkazy na již hotové integrace OSG s celou řadou používaných toolkitů. Z těchto toolkitů jsem měl školní zkušenosti s wxWidgets a s GLUT. WxWidgets mi přišlo jako zbytečně obsáhlé, protože jsem neplánoval vykreslovat žádné ovládací prvky operačního systému, kromě aplikačního okna. GLUT mi naopak přišel příliš minimalistický. Po prozkoumání ostatních alternativ jsem se rozhodl pro SDL.

3.1 SDL (Simple Directmedia Layer)

SDL [11] je výbornou C knihovnou často využívanou k tvorbě multiplatformních počítačových her. Bohužel osgSDL, nabízené na stránkách ODE jako integrační varianta, je několik let neaktualizované a udává kompatibilitu s OSG ve verzi pouze 1.2 (aktuální verze je 2.8). Rozhodl jsem se tedy spojit obě knihovny sám. Pro studium SDL jsem využil tutorial na root.cz [12]. Pro spojení s OSG pak návody na [13].

3.2 OSG (OpenSceneGraph)

Jedná se o vyzrálou C++ knihovnu s objemnou členskou základnou. Data organizuje do hierarchické struktury grafu scény. Uzly v tomto grafu nejenom člení graf, ale také mění vlastnosti svých potomků nebo je transformují (posouvají, rotují, mění velikost atd.). Listy, které jsou v OSG nazývány Geode, obsahují geometrii, která je transformována všemi transformacemi svých předků a její OpenGL stav je poděděn ze stavu nejbližšího předka. Je důležité si uvědomit, že graf scény není strom, tedy uzel může mít několik předků. To je s výhodou využitelné, pokud máme několik shodných geometrií, které se liší pouze transformací nebo OpenGL vlastnostmi. Například několik jinak natexturovaných a napozicovaných mrakodrapů umístěných několikrát do scény města.

Mimo výše zmíněné uzly lze do grafu scény připojovat další, specializované uzly, implementující různé efekty (level of detail, particle systém ...). Existuje také celá řada rozšiřujících nodekitů, například osgTerrain nodekity pro zobrazování rozsáhlých terénu atd.

OSG používá standardně OpenGL display listy a data se snaží do OpenGL posílat v takovém pořadí, aby minimalizovalo počet změn stavu.

Pro procházení grafu scény OSG využívá metodu návštěvníků (visitor pattern). Každý takový návštěvník je poděděn ze třídy NodeVisitor a implementuje akce, které se mají vykonat při návštěvě různých druhů uzlů. Existují i návštěvníci, kteří procházejí graf vzestupně. Pokud vyžadujeme funkčnost, kterou žádný ze standardních návštěvníků nenabízí, je velice snadné naprogramovat si svého vlastního návštěvníka.

Při vykreslení každého snímku je typicky zavolán průchod (traversal) třech návštěvníků. První takzvaný UpdateVisitor slouží k aktualizaci scény. Projde graf scény a pro každý uzel provede uživatelem definovanou akci. Druhým je CullVisitor, který vyřadí uzly které nejsou viditelné a tedy nebudou vykresleny. Například ty, jenž jsou mimo kamerou viditelný prostor nebo jsou zacloněny jinou geometrií atd. Třetí je draw traversal, který pošle data do OpenGL a vykreslí snímek.

Další užitečnou vlastností OSG je, že používá chytré ukazatele (smart pointers). Chytrý ukazatel je v OSG objekt třídy ref_ptr. Tento se pomocí C++ šablony a přetížených operátorů chová jako obyčejný ukazatel, který je navíc schopen informovat objekt na který ukazuje (objekt třídy Referenced) o tom, že na něj ukazuje. Všechny objekty v OSG, které mají být uloženy v hlavní paměti jsou poděděny ze třídy Referenced a v každém okamžiku vědí, kolik chytrých ukazatelů se na ně odkazuje. Pokud toto číslo klesne na nulu, tak zavolají svůj destruktor a uvolní se z hlavní paměti. Uzly grafu scény jsou taktéž třídy Referenced a odkazují na své potomky ref_ptry. To nám umožňuje si například uchovávat ref_ptr pouze na kořen grafu scény a pokud chceme celou scénu uklidit, stačí nám potom zrušit pouze tento ukazatel. Tedy za předpokladu, že do grafu scény neukazují jiné ref_ptry.

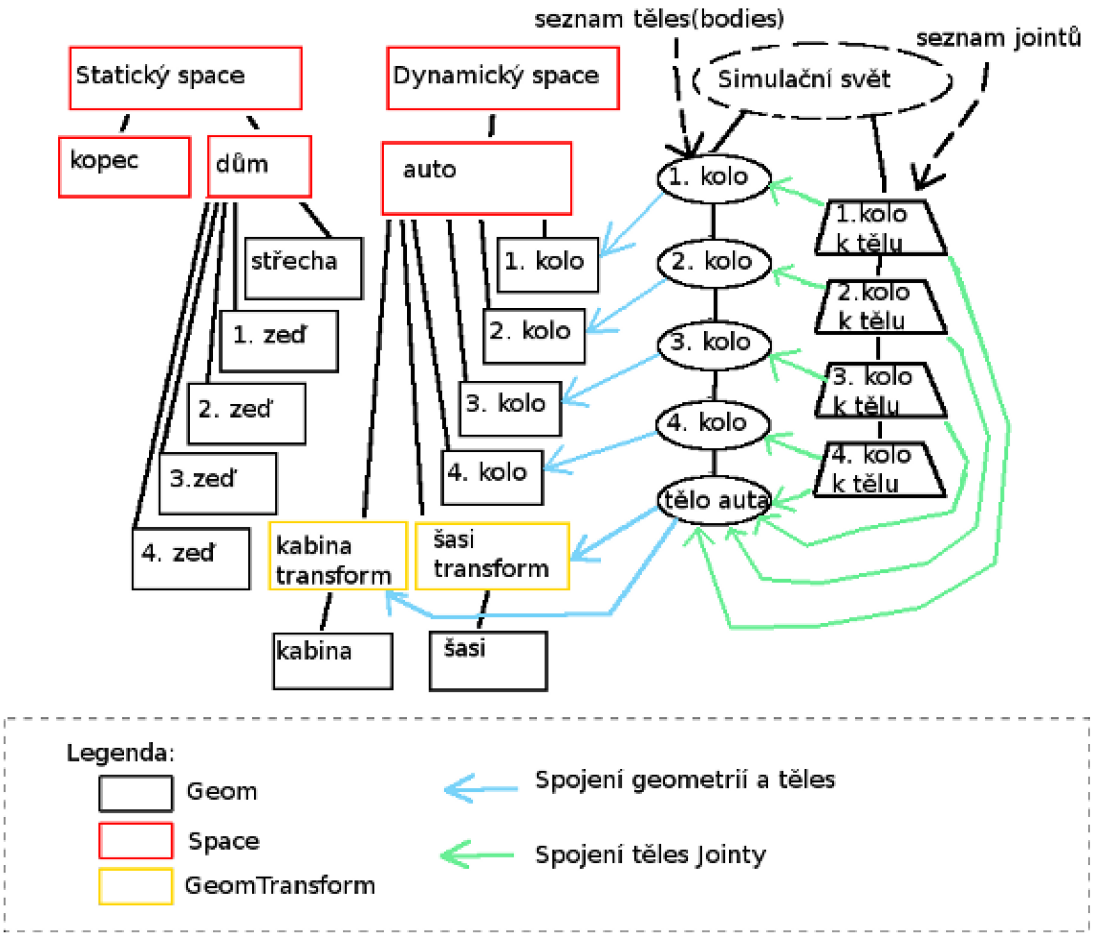
Na stránkách OSG je velké množství dokumentace a tutoriálů, ze kterých mohu doporučit hlavně OSG Quick Start Guide (OSGQSG). Podrobnější a zejména implementační detaily se mi osvědčilo hledat v oficiální online Doxygen referenční dokumentaci nebo přímo ve zdrojových souborech OSG.

3.3 ODE (Open Dynamics Engine)

ODE je open-source multiplatformní knihovna, původní API je pro jazyk C, ovšem v současnosti je k dispozici i C++ API, které je pouze objektovým zapouzdřením funkcí z C API. Zajímavostí také je, že interně je ODE psané objektově v jazyce C++ a hlavní C API je tedy zapouzdřením tohoto vnitřního objektového modelu. V případě používání veřejného C++ API se jedná už o dvojité zapouzdření. V době psaní tohoto textu je aktuální release verze ODE 0.11.1 .

ODE je real-time fyzikání engine hojně využívaný v aplikacích simulujících dynamiku a mechaniku tuhých těles a soustav pospojovaných tuhých těles (articulated rigid bodies). Nejčastěji bývá v koncových aplikacích používáno k simulaci vozidel, robotů, pohyblivých končetin atd. Současná verze není schopna simulovat aerodynamiku nebo hydrodynamiku. ODE v sobě má zabudovaný diskrétní detektor kolizí. Pro detekci kolizí s trojúhelníkovými meshi v sobě má začleněno dva detektory OPCODE a GIMPACT a dovoluje svým uživatelům si pří kompilaci ODE vybrat, který si přejí používat. Cílem ODE je být rychlé, stabilní a uvěřitelné. ODE není v žadném případě přesné. To je dáno zejména tím, že o změnu simulačních stavů se stará integrátor pouze prvního řádu. ODE je tedy ideálním nástrojem pro fyzikální simulace v počítačových hrách, kde záleží hlavně na dojmu hráče a ne na výsledné přesnosti. K pochopení ODE mi posloužila ODE Wiki[3], zejména sekce manual, kterou lzde stáhnout v podobě pdf. Velkým přinosem mi také bylo čtení ode-users Google group[14] a zdrojových souborů ODE.

3.3.1 Architektura ODE



Obrázek 3: příklad simulované scény v ODE

Architektura ODE je velice modulární. Chytře odděluje struktury pro detekci kolizí od struktury pro simulaci. To uživatelům ODE mimo jiné umožňuje použít pouze detektor kolizí nebo pouze fyzikální simulátor.

Typická struktura scény v ODE je pro názornost zobrazena na obrázku 3. Většinou simulujeme jeden svět (World), tento obsahuje seznam těles (Body) a Jointů. Dále máme obecně několik kolizních stromů. Dost často jeden pro statickou geometrii a druhý pro dynamickou geometrii. Aby byla geometrie dynamická, měla by být přiřazena tělesům, které s ní vždy po simulačním kroku pohnou.

3.3.2 Kolizní stromový prostor (Space)

Většinou když mluvíme o kolizním stromu, označujeme jej jeho kořenovým (top-level) spacem. Vytvářením kolizního stromu vytváříme prostorovou organizaci scény a optimalizujeme broadphase detekci kolizí.

Každý uzel v tomto stromu je interně poděděn ze třídy Geom. Každý takový Geom má svůj vlastní bounding AABB. ODE tedy automaticky používá bounding volume hierarchii pro optimalizaci kolizní detekce. Geom se potom dále specializuje buďto na Space, GeomTransform nebo na nějaké grafické primitivo (BoxGeom, SphereGeom, TrimeshGeom ...).

Space je uzel, který může mít poduzly, tedy člení nám kolizní strom. ODE implementuje několik druhů Spaců, každý poskytující odlišnou metodu space partitioningu. Jedná se o SimpleSpace, QuadTreeSpace, HashSpace a nově i SweepAndPruneSpace. SimpleSpace neimplementuje žádnou broadphase optimalizaci, jedná se o klasické bruteforce testování se složitostí $O(n^2)$. Je použitelný pro malý počet poduzlů (do cca 10ti) a většinou i rychlejší, protože odpadne režie pro optimalizační struktury. Například Space auto a Space dům z obrázku 3, by mohly být vytvořeny jako SimpleSpace. QuadTreeSpace organizuje AABB svých synovských uzlů do quadtree stromu a je nejefektivnější při jejich rovnoměném rozložení. Je dobře použitelný například pro organizaci statického terénu, který je osazenen kameny, stromy atd. V případě HashSpace, jsou synovské AABB rozkouskovány na menší části a vloženy do hashovací tabulky. Tento space se hodí k obalení prostoru obsahujícího dynamické a nerovnoměrně rozložené Geomy. SweepAndPruneSpace je v ODE zatím nezdokumentovaný, ale jeho využití by mělo být podobné, jako u HashSpacu.

Další možnou specializací Geomu je GeomTransform. Tento vždy zapouzdřuje jeden listový Geom a kompletně ho nahradí. Zapouzdřený Geom nemá žádné jiné vazby (na Space nebo na Body), kromě té na svůj GeomTransform. Zapouzdřený Geom poté svojí pozicí a rotací neurčuje již svoji globální transformaci, ale pouze relativní transformaci ke transformaci svého GeomTransformu.

GeomTransform se používá, pokud chceme na těleso (Body) připojit geometrii, která bude lokálně transformovaná od transformace tělesa (například při vytváření kompozitních těles) .

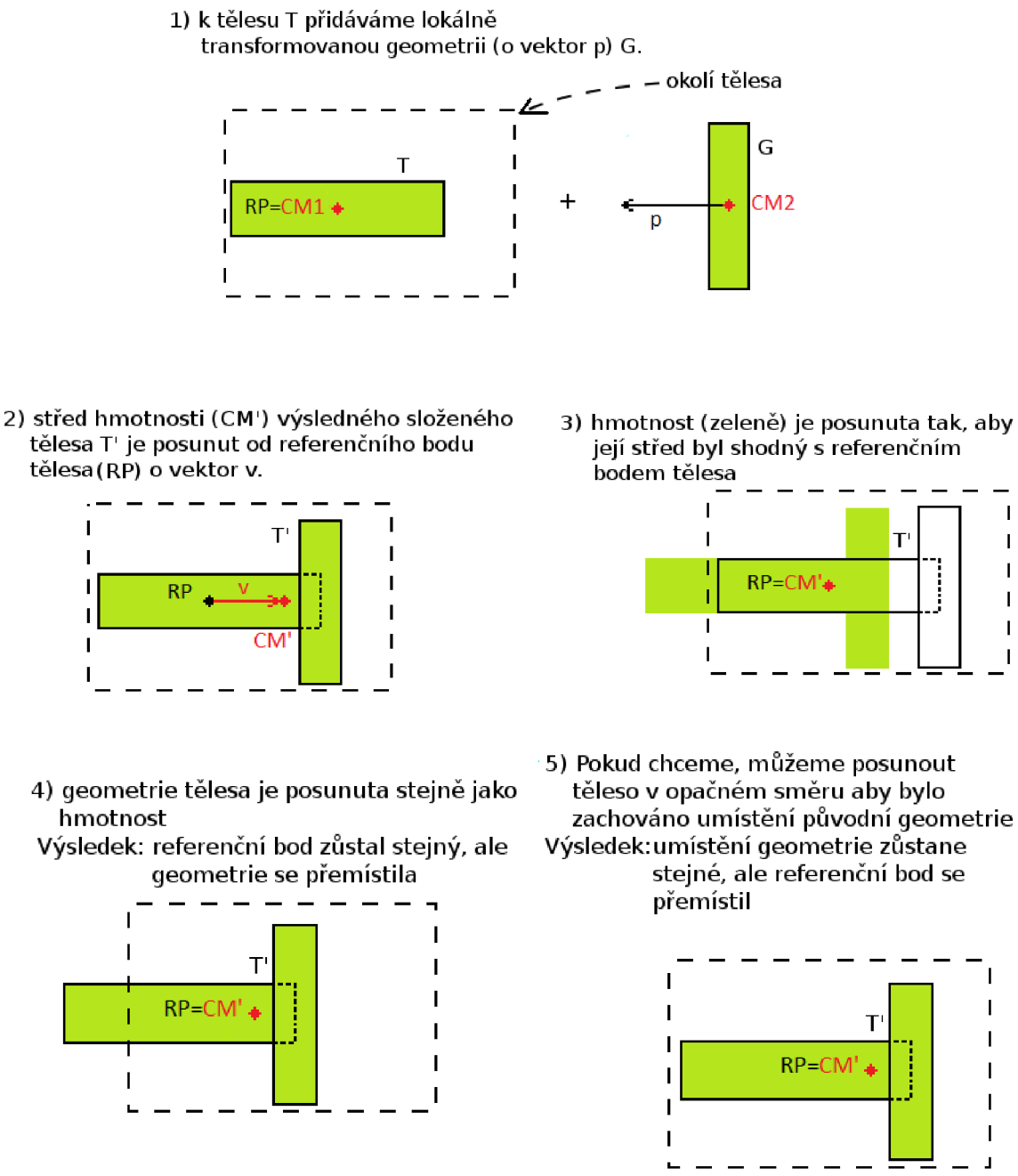
3.3.3 Simulační svět (World)

World obsahuje kompletní stav simulace. Tedy seznam simulovaných těles a seznam Jointů mezi nimi. Současné ODE nabízí dvě funkce na krokování simulace. První z nich je pomalá a paměťově náročná dWorldStep() a druhou je dWorldQuickStep(), která by měla být o poznání rychlejší zejména při simulaci rozsáhlých světů. Tyto funkce vyžadují jako parametr velikost simulačního kroku, který by měl být při každém volání stejný, tedy konstantní, po celý běh aplikace. Pokud by simulační krok byl proměnlivý, simulace by se mohla stát nepředvídatelnou a nestabilní.

Tělesa (Bodies)

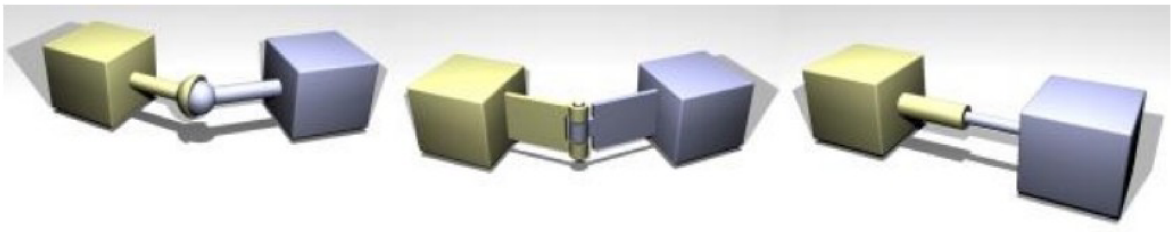
Simulační reprezentace pevného tělesa v ODE je v každém simulovaném okamžiku dána jeho globální polohou, globální rotací, vektorem lineární rychlostí (linear velocity), vektorem úhlové rychlostí (angular velocity), hmotností, maticí inerce (inertia matrix), dále pak akumulátorem síly(force) a akumulátorem kroutivé síly(torque). Pro ODE simulátor je těleso pouze referenční bod (reference point - RP) v prostoru. Energii tělesa určuje jeho lineární a úhlová rychlost. Vektor úhlové rychlosti je ve tvaru, kde velikost vektoru udává úhlovou rychlost v radiánech za sekundu a směr tohoto vektoru udává rotační osu. Akumulátory pro force a torque jsou pouze dočasné a po vyhodnocení simulačního kroku se přemění na přírůstek lineární či úhlové rychlosti. Hmotnost a matice inerce je v ODE zapouzdřena do struktury dMass a pro její vytváření a modifikaci nabízí ODE několik užitečných funkcí.

Drobný problém nastává, při pokusu vytvořit kompozitní těleso, vzniklé složením několika geometrií, kde každá geometrie má svoji dílčí hmotnost a některá z geometrí je lokálně transformovaná. ODE totiž klade jedno důležité omezení, střed hmotnosti musí být shodný s referenčním bodem tělesa. Při takovéto operaci se musíme rozhodnout, zdali chceme zachovat umístění geometrií ve scéně nebo umístění referenčního bodu tělesa. Přemístěním geometrií může dojít k nečekaným kolizím. Při přemístění referenčního bodu zase k porušení jointů. K názornosti snad pomohou následující obrázky 4-8.



Obrázky č. 4-8: Přidávání Geomu s lokální transformací k tělesu

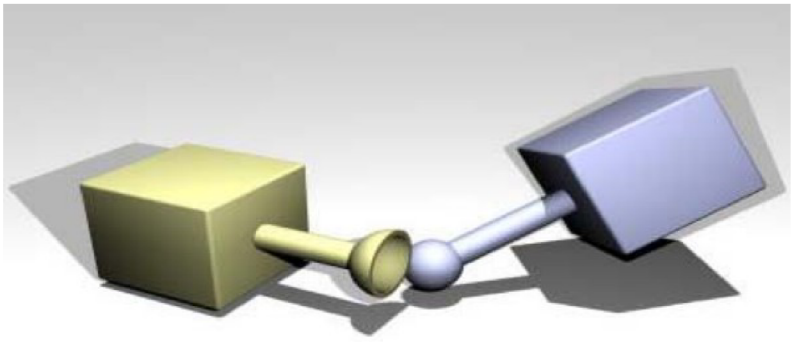
Jointy



Obrázek č.9: Typické Jointy v ODE, zleva: ball and socket, hinge, slider
obrázky převzaty z [3]

Jointy jsou hlavním simulačním mechanizmem v ODE a bezesporu hlavním důvodem, proč je ODE tak mocný nástroj. Některé typy Jointů jsou znázorněny na obrázku 9. Jointy kladou omezení (constraints) na relativní pozici a rotaci dvou těles (nebo na těleso a statický svět). Lze říci, že Jointy ubírají tělesům stupně volnosti (degrees of freedom – DOF). Constrainty Jointů jsou interně popsány soustavami matematických rovnic. Kvalitním materiálem popisujícím implementaci Jointů je [15], který také popisuje, jak vytvořit nové druhy Jointů. V ODE existuje celá řada Jointů nabízejících širokou škálu využití, pro detaily o jednotlivých Jointech doporučuji stáhnout a pročíst ODE wiki manuál. Přiložená demo aplikace ukazuje použití většiny Jointů.

Jointy je možno parametrizovat a docílit tak leckterých efektů, motorizovat je a tvořit tak pohyblivé součásti nebo omezit zbylé stupně volnosti pomocí limitů (joint stops). Pokud jsou omezení porušena, joint typicky sjedná nápravu aplikováním síly (force) nebo kroutivé sily (troque) na tělesa ve snaze dodržet omezení. Porušení může vzniknout buď akumulováním simulační chyby (znatelné u rychle se pohybujících nebo rotujících těles) nebo pokud uživatel zanese do simulace chybu tím, že spojené těleso přemístí či zarotuje bez použití sil (to je nesimulační přístup, přímé nastavování pozice a rotace těles by mělo být použito pouze při inicializaci scény). Takovéto porušení je názorné z obrázku č.10.



Obrázek č.10: Joint chyba, obrázek převzat z [3]

Způsob jakým joint řeší obnovení platnosti svých porušených omezení je také parametrizovatelný. Slouží k tomu dvojice hodnot, ERP (error reduction parameter) a CFM (constraint force mixing). Tyto hodnoty lze nastavit globálně pro celý World a také lokálně pro každý Joint (lokální nastavení má větší prioritu). Tyto hodnoty jsou bohužel závislé na simulačním kroku (při změně simulačního kroku je třeba je všude přenastavit) a pro požadovaný efekt je třeba je dlouze dolaďovat. Jednoduše řečeno, ERP určuje jakou část síly potřebné k nápravě omezení v jednom kroku joint v každém kroku použije, tedy určuje jak rychle bude obnoveno omezení. Při vysokém ERP může dojít k rozkmitání těles, pří nízkém můžou být jointy rozpojeny. CFM určuje jak moc mohou být constrainty porušeny aniž by se joint pokoušel o nápravu. Pro lepší pochopení ERP a CFM navrhuji pročíst Wiki manuál, pro jejich roli v constraint rovnicích viz [15].

3.3.4 Kolizní detekce a odezva v ODE

Uživatelé ODE, pokud chtějí funkční kolize, si musí při volání některé funkce pro kolizní detekci předat ukazatel na svoji funkci, nazývané většinou jako near callback. Ten bude volán pro dvojici Geomů (a jejich derivátů - tedy i Spaců) vždy, pokud vznikne podezření, že by Geomy mohly kolidovat. Toto podezření vznikne na základě toho, že se překrývají jejich AABB.

V této funkci si uživatel typicky ověří, jestli je některý z dvojice Geomů Spacem. Pokud ano, tak se znovu zavolá funkce pro kolizní detekci a znovu s ukazatelem na tuto funkci. Takto se rekurzivně zanořuje do kolizního stromu. Pokud se jedná o listové Geomy (nějaká geometrická primitiva), tak je vykonána narrow-phase detekce kolizí zavoláním funkce, která zkontroluje overlap primitiv. Tato funkce naplní dodanou strukturu užitečnými kolizními informacemi (collision feedback). Jedná se o kontaktní body, kontaktní normály a hloubky zanoření.

Z tohoto feedbacku se většinou nechá vytvořit Contact Joint, který je způsobem, jakým si ODE předá informace z detektoru kolizí do simulátoru. Contact Joint nabízí oproti standardnímu Jointu navíc další parametry, s jejichž použitím je možné simulovat tření, klouzání, měkkost (simulovaná pomocí ERP a CFM, nedojde tedy k žádné deformaci Geomů) a odrážení (bounce).

Z těchto odezev zaslouží největší pozornost tření. ODE nabízí dva módy tření. Z nichž první není věrohodná simulace tření, pouze nastavuje limit síly, kterou může povrch působit na jednotku normálové síly. Druhý model a o poznání přesnější, používá pyramidovou aproximaci Coulombova kuželovitého třecího modelu a dovoluje nám přímo nastavovat koeficienty tření. Pro více informací o tření v ODE doporučuji Wiki manuál nebo prohledat ode-users Google group. Pro podrobné detaily a o simulaci tření v počítačových hrách obecně viz [16].

3.3.5 Pohyb a rotace těles v ODE

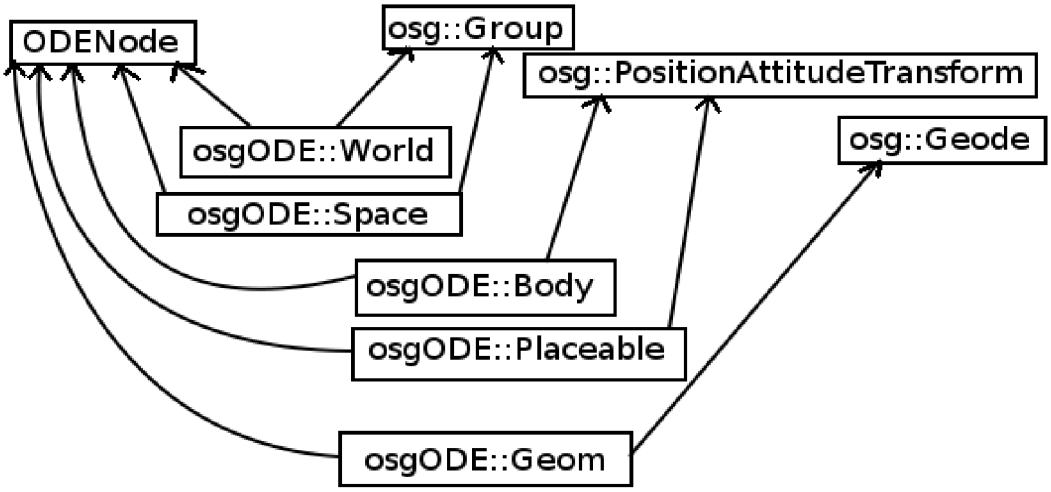
Docílit požadovaného pohybu či rotace v ODE může být někdy velice těžké. Simuluje totiž skutečnost a pohyby jsou akcelerované pomocí sil. Nejjednodušším řešením pro zamýšlený pohyb většinou bývá použití motorizovaných Jointů. Téměř každý Joint v ODE může být motorizovaný pro každou svoji osu rotace či pohybu. Pro lineární a úhlový pohyb existují dva speciální druhy Jointů, LinearMotor a AngularMotor, tyto se dají většinou úspěšně použít pro kontrolu pohybu tělesa. Pokud je ovšem požadavek specifičtější, je také možno aplikovat síly na tělesa přímo. Je-li ale působících sil více, může být složité najít správné řešení. Motorizovanému Jointu nastavíme požadovanou rychlost jeho pohybu nebo rotace a maximální sílu, kterou může za jeden simulační krok vynaložit na dorovnání této rychlosti. Takový Joint je schopen spočítat všechny působící síly a použít přesně tolik síly kolik potřebuje (pokud to není více než povolené maximum), aby zrychlil nebo zpomalil na požadovanou rychlost.

3.4 Spojení ODE a OSG

3.4.1 Cíl

Aby byla geometrie z ODE zobrazitelná v OSG, je třeba ji připojit do grafu scény. S touto geometrií v ODE pohybují tělesa. V OSG pohyb takovýchto dynamických geometrií budeme nastavovat na nějakém PAT (PositionAttitudeTransform) uzlu. Také statická geometrie bude muset být obalena PATem, pokud ji budeme chtít umístit jinám než do globálního počátku či nějak zarotovat. Nakonec jsem implementoval řešení, kdy všechny objekty z ODE, vyjma Jointů, přípojím do OSG grafu scény. Toto řešení by mělo kompletně zapouzdřit ODE implementaci ze strany OSG a takovýto wrapper jsem nazval pracovním jménem osgODE. V budoucnu bych chtěl toto řešení nabídnout komunitě OSG jako NodeKit.

3.4.2 Implementace



Obrázek 11: Diagram pro hlavní třídy osgODE wrapperu

Pro ODE objekty byly vytvořeny osgODE wrapper třídy, poděděné z abstraktní třídy ODENode, která definuje rozhraní pro připojování (attach) a odpojování (detach) do grafu scény. Objekty těchto tříd obsahují vždy příslušný ODE objekt a zapouzdřují většinu funkcí nad ním. Třídy osgODE dále dědí z některých OSG tříd. Na obrázku 11 je vidět ze kterých. Pokud je to třeba, tak přepisují virtualní metody osg::Group::addChild() a osg::Group::removeChildren() a přidávají různé kontroly toho, co se kam může připojit (například osgODE::Body nesmí být pod dalším osgODE::Body atd.) .

Třída osgODE::Placeable obsahuje obalený GeomTransform. Ten je potřeba jednak pokud chceme vytvářet kompozitní tělesa s posunutou geometrií a také pokud chceme umístit statickou geometrii jinam než do globálního počátku. V ODE je sice možné nastavovat umístění do Geomu přímo, ale v OSG potřebujeme na umístění osg::Geode nějaký PAT. Tímto PATem nám je osgODE::Placeable, který stejně jako GeomTransform v ODE lokálně transformuje geometrii. Třída osgODE::Body je taktéž poděděna z PATu a udává globální umístění a rotaci tělesa. Třída osgODE::Space žádnou transformaci nepřidává a je poděděna z osg::Group. Tato třída je také pouze abstraktní a přímo se vytvářejí až její deriváty, těmi jsou osgODE::SimpleSpace, osgODE::HashedSpace a osgODE::QuadTreeSpace. U třídy osgODE::Geom se taktéž jedná o abstraktní bázi pro specializovanější třídy obsahující různé ODE kolizní geometrie. Pro všechny kolizní geometrie jsou vytvořeny jejich zobrazitelné reprezentace v OSG jako osg::Drawable. Pro geometrii jednoduchých tvarů jsou použity osg::ShapeDrawable.

U každého geomu je možno skrýt jeho geometrickou reprezentaci a místo ní zobrazit trojúhelníkový mesh (trimesh). Toto je použito v demonstrační hře u vystřelovaných raket, které

kolidují jako kapsle (capsule), ale vypadají jako meshové rakety. V aplikaci byly použity modely .obj a složitější formáty nejsou podporovány. Výsledkem načítací funkce osg je totiž obecně nějaký graf scény, ovšem tento je v mé implementaci zploštěn (flattened) a geometrie jsou z něho připojeny přímo na osgODE::Geom. V případě TrimeshGeomu, kdy vyžadujeme trojúhelníkovou geometrii i na kolidování, jsou z OSG získány pole trojůhelníkových vertexů a indexů a tyto jsou překopírovány do ODE. Pro získání indexů z OSG geometrií je použit osg::TriangleIndexFunctor.

K aktualizaci PATů těles dochází při OSG update traversalu před vykresením každého snímku. Umístění a rotace těles jsou zde nahrány do jejich OSG reprezentací.

Většina ODE Jointů a jejich metod je také obalena do wrapper tříd. Pro detekci kolizí je vytvořena třída osgODE::CollisionFeedback, která umožňuje nastavit callback funkci, do které bude tento feedback předán k provedení kolizní odezvy.

3.4.3 Problémy a omezení

Spojením kolizní a simulační struktury ODE do jedné struktury grafu scény je potlačena jistá flexibilita ODE. Pokud například chceme přemístit geometrii spojenou s tělesem do jiného Spacu, musíme také přemístit toto těleso. Při této operaci dojde k odpojení uzlu tělesa od Worldu, což není možné provést bez zrušení tělesa a odpojení od všech jeho Jointů.

V současnosti je na ODENody kladeno omezení, že můžou mít maximálně jednoho rodiče, grafem osgODE scény je tedy strom. Těleso může obsahovat pouze osgODE::Placeable geometrii, pro případy, kdy bude třeba přepočítávat hmotnost tělesa a posouvat všemi jeho osgODE::Placeable uzly.

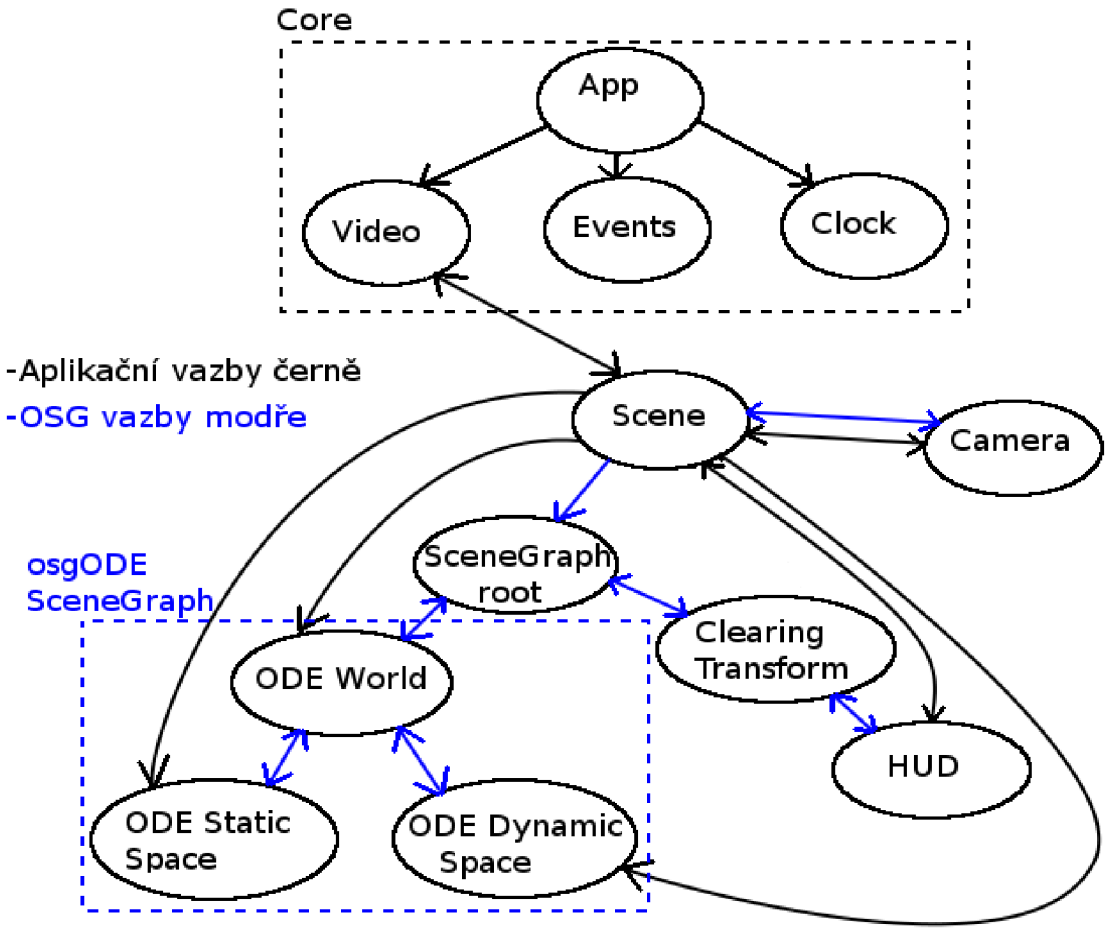
4 Demonstrační aplikace

Demonstrační aplikace je hlavním produktem této bakalářské práce. Jedná se o aplikaci naprogramovanou objektově v jazyce C++. Je vyvíjena pod Windows v multiplatformním a open-source vývojovém prostředí Code::Blocks. Pro překlad a sestavení aplikace je používán překladač GNU GCC a sestavovací program GNU Make, respektive jejich porty pro Windows ze sady MinGW.

Aplikace implementuje první verzi herního fyzikálního enginu, který bude dále vyvíjen. Motivací projektu bylo porozumět tvorbě počítačových her a navrhnout engine schopný vytvořit a řídit herní scénu.

4.1 Herní engine

4.1.1 Moduly



Obrázek 12: Schéma provázání grafu scény s moduly enginu

Engine je pro svoji rozsáhlost rozdělen do několika modulů, tyto moduly na sebe podle potřeby hierarchicky odkazují. Toto je znázorněno na obrázku 12. Moduly jsou spojeny s OSG grafem scény přes modul Scene.

4.1.2 Vstup aplikace, Core moduly

Ve vstupní funkci main() je vytvořen singleton třídy App, tento bude dostupný ze všech modulů programu voláním App::get_instance() a slouží jako kořen stromu modulů. U něho se registrují jaderné moduly (Core modules). Tyto jsou aktuálně v aplikaci tři. Jsou to moduly Video, Events a Clock. Všechny tyto moduly jsou poděděny z abstraktní třídy Module a interně využívají SDL. Ke správné inicializaci a deinicializaci SDL subsystémů slouží funkce deklarované v souboru my_sdl.h. Jaderné moduly jsou vytvořeny a parametrizovány ve funkci main(). Jejich konstruktorům jsou předány konfigurační struktury (Config), které obsahují například citlivost myši, velikost aplikačního okna, barevnou hloubku atd. Pokud Core modul inicializuje další podmoduly, pak jeho Config obsahuje hierarchicky zanořené Configy i pro tyto podmoduly.

Pokud inicializace Core modulů proběhne v pořádku, tak je zavolána funkce loop() objektu App. V této funkci je nejdříve zkontrolováno, jestli jsou registrovány a inicializovány všechny očekávané Core moduly. Pokud ano, tak je spuštěna tato aplikační smyčka:

```
clock->_start();    //start clock
while (!_last_error)
  // main loop
  {
    clock->_cycle();
    //handle events
    events->handle_events();
    //draw video frame
    video->draw();
  }
```

Smyčka iteruje dokud nedojde k nějaké chybě nebo uživatel neukončí aplikaci. V každé iteraci je zavolána funkce _cycle() modulu Clock, který si tímto aktualizuje čas od minulé iterace, což je čas, který trvalo vyrobit (odsimulovat a vykreslit) minulý snímek. Je to také čas který bude třeba odsimulovat v tomto snímku. Dále jsou zpracovány události od operačního systému a uživatele. Nakonec je zavolána funkce draw, která obsahuje simulaci i vykreslení snímku.

Modul Clock kromě počitání času snímků může také zpomalovat cyklus pomocí funkce SDL_Delay v případě, že chceme limitovat maximální počet snímků. Také umožňuje nastavovat rychlost simulace a získat simulační čas cyklu.

Modul Events obsluhuje události z klávesnice, myši a také událost při požadavku na ukončení aplikace. Po pohybu myší vždy přemístí kurzor do středu aplikačního okna.

Modul Video při své konstrukci vytvoří okno aplikace a v něm OpenGL SDL surface. Poté vytvoří a inicializuje podmodul Scene. Parametrizuje jej SceneConfigem, který je součástí jeho VideoConfigu. Ve své cyklické funkci draw() je vždy nejdříve vykonána funkce make_frame() z modulu Scene, která odsimuluje a vykreslí nový snímek, tento snímek je poté nahrán do frontbufferu pomocí SDL_GL_SwapBuffers().

4.1.3 Modul scény (Scene)

Scene modul se stará o celou zobrazovanou a simulovanou scénu a taktéž o objekt kamery. Je poděděn z osg::SceneView a jako takovému jsou mu při konstrukci nastaveny standardní osvětlení a kamera pomocí setDefaults(). Standardní kamera je ovšem ihned nahrazena aplikační kamerou třídy Camera. Následně je vytvořen kořen grafu scény. K tomu jsou připojeny dva podgrafy. Prvním je simulovaná část scény s kořenem třídy osgODE::World a dvěma potomky třídy osgODE::Space (dynamic space a static space). V druhém podgrafu scény jsou geometrie transformované do 2D projekce na obrazovku na tzv. HUD (head-up display).

Modul scény na konci své konstrukce načte obrázky pro textury a inicializuje obsah scény pomocí metod populate_hud() a populate_world().

4.1.4 Simulace

Simulaci těles v ODE je třeba provádět s konstantním časovým krokem a je potřeba ji začlenit do hlavního cyklu před vykreslení snímku, které probíhá v proměnlivých časových intervalech. Dále musí být k dispozici aparát, kterým budeme plánovat události a také provádět aktualizace těles.

Simulační cyklus

```
void Scene::make_frame()
{
  static double time_accumulator=0.0;
  unsigned int steps_this_cycle;
  Clock *clock= static_cast<Clock*>(App::get_instance()->get_module(CLOCK));

  // find out how many steps we must perform
  time_accumulator+=clock->simulation_cycle_time()/_world->step_size();
  steps_this_cycle= (unsigned int)time_accumulator; // get just whole steps
  time_accumulator-=steps_this_cycle;   // remove them from time accumulator
```

```
    for (unsigned int i=0; i<steps_this_cycle; i++ )
      // iterate through integration steps
      {
        // call ode integrator execute step
        _world->quick_step();
        _world->clear_contacts();   // empty collision joints
        _dynamic_space->collide();  // dynamic-dynamic collisions
        _dynamic_space->collide(_static_space); // static-dynamic collisions

        // call scheduled callbacks & updates
        _world->call_callbacks();
      }
    //update camera matrix
    _camera->update_view_matrix();
    // call osg update traversal -> sync from ode to osg
    update();
    // Perform culling of the scene for speed
    cull();
    // Draw the scene data to OpenGL buffers
    draw();
  }
```

Simulační cyklus je vložen do části herního cyklu prováděného Scene modulem, do funkce make_frame(), před trojici OSG traversalů (update, cull, draw) a tedy před vlastní vykreslení nového snímku. Simulační čas předchozí iterace herního cyklu (simulation_cycle_time) je pouze skutečným časem této iterace vynásobeným konstantou určující rychlost simulace. Tento čas se přičítá do tzv. časového akumulátoru (time_accumulator), který je static a umožňuje nám tedy přenášet nedělitelné časové zbytky mezi iteracemi. Z tohoto akumulátoru je v každé iteraci odebrán celočíselný násobek simulačních kroků (steps_this_cycle). Počet těchto kroků je proveden ve vnitřním simulačním cyklu.

V tomto cyklu je nejprve zavolána funkce quick_step(), která zintegruje ODE do nového stavu. Poté jsou detekovány kolize mezi ODE Geomy, které vyrobí Contact Jointy, popřípadě provedou jiné individuální odezvy. Nakonec jsou zavolány naplánované události (ODECallbacky).

Při tvorbě tohoto cyklu jsem čerpal ze skvělého článku Glenna Fiedlera [17].

Plánovač událostí

Všechny uzly v grafu scény, které jsou poděděny z abstraktní třídy ODENode si mohou, pokud jsou skrze graf připojeny k uzlu World, plánovat události. Toto je například nezbytné, pokud chceme vykonat odezvu na kolizi, která zahrnuje odstranění některých ODE objektů (těleso vybouchne po kolizi nebo jinak „zahyne“ a je třeba ho odstranit ze scény) nebo jejich přesunutí atd.

Takovouto odezvu není možno vykonat ihned v kolizním callbacku, ODE si totiž pro své potřeby uzamyká Space a s ním i všechny Geomy v něm. Odezva proto musí být naplánována a provedena později.

Mnohem častěji je ale plánovač využíván na aktualizaci (update) objektů. V tomto updatu máme většinou implementovaný nějaký kontroler který pohybuje s objektem pomocí sil nebo ODE motorizovaných Jointů. Pro dosažení požadovaného výsledku je nutné aby takovýto kontroler pracoval v konstantních intervalech pomocí periodického volání událostí.

Při implementaci plánovače jsem se inspiroval oporou ke předmětu IMS [18]. Plánovač řadí události do prioritizované fronty. Událostem je možné nastavit prioritu s jakou se mají vykonat a počet simulačních kroků, za který se mají vykonat. Pokud si callback této události přeje, může se znovu naplánovat tím, že vrátí hodnotu true. Takto jsou řešeny výše zmíněné updaty.

V mém řešení se události nazývají ODECallback, aby nedošlo k záměně s událostmi které přicházejí od operačního systému a řeší je modul Events. Následuje deklarace této struktury. Implementace těchto postupů je kvalitně komentována ve zdrojových kódech.

4.1.5 Kamera

Aplikace používá jednu kameru. Tato je poděděna ze třídy osg::Camera. Je parametrizovatelná přes strukturu CameraConfig. Zobrazuje perspektivní projekci skrze viewport o stejné velikosti jakou má aplikační okno.

Kamera je také zděděna ze třídy Controllable, která implementuje rozhraní pro rotování pomocí Eulerových úhlů, pro nastavování pohybových příznaků a vypočítání pohybového vektoru. V aplikaci je použita YPR (yaw, pitch, roll) notace Eulerových uhlů. Kde yaw je úhel otočení okolo osy Y, pitch okolo osy X a roll okolo osy Z. Aplikace uvažuje klasický OpenGL pravotočivý souřadný systém, kde poloosa -X směřuje doleva, poloosa -Y dolů a poloosa -Z dopředu. Pokud jsou všechny úhly nastaveny na nulu, potom kamera míří do poloosy -Z. Kamera umožňuje nastavovat limity pro jednotlivé rotace a v aplikaci je takto nastaven minimální pitch na $-\pi/2$ a maximální pitch na $\pi/2$. Tímto je zamezeno přetáčení, které by mohlo uživatele zbytečně mást.

Kamera může být buďto odpojená (detached) nebo připojená (attached). Jestliže je odpojená, tak nemá žádnou reprezentaci ve fyzikální simulaci, nekoliduje s geometrií ve scéně a pohybuje se konstantní rychlostí.

Má ale možnost se pomocí ODE paprsku připojit na speciální druh objektů (MyControllableObject) ve scéně, o kterých bude řeč později v kapitole o obsahu demoscény. Tyto objekty jsou také poděděny ze třídy Controllable, ale na rozdíl od kamery se pohybují fyzikálně a

akcelerovaně. Kamera se nechá svým carrier objektem „nést“, tzn. svoji pozici tedy v každém snímku nastavuje na pozici carrieru. U rotace to ale funguje opačně, objekty u kterých je to vyžadováno se snaží pomocí ODE motorizovaných jointů nebo pomocí torque dorovnat rotaci kamery. Buďto synchronně ve svém update callbacku dorovnávají absolutní rotaci kamery nebo jim také kamera poskytuje v každém snímku změny úhlů, ke kterým v tom snímku došlo (delta_yaw, delta_pitch, delta_roll).

U připojené kamery je také možnost dívat se na carrier z pohledu třetí osoby. Při tomto pohledu je z pozice carrieru vystřelován ODE paprsek směrem dozadu, který má za úkol zjistit, jestli je požadované odsunutí kamery do módu třetí osoby možné, tedy pokud v cestě posunutí není žádná geometrie. Tento paprsek je aktuálně nastaven na kolize pouze se statickým světem pro přehlednost v některých dynamických místech scény. Pokud paprsek detekuje kolizi, posunutí sníží na vzdálenost k této kolizi.

Poslední způsob, jakým kamera umožňuje vrhnout tento paprsek, je směrem dopředu, a tím zjistit pozici kam se kamera dívá. Toto je využito při nastavování směru vystřelených projektilů.

4.1.6 HUD

Uzel třídy HUD je poděděn z osg::Projection. Do grafu scény je připojen skrze uzel osg::MatrixTransform, který má v OSG nastaven referenční rámec (reference frame) na RF_ABSOLUTE, tedy neuvažuje transformace uzlů nad ním. V obrázku č. 12 je tento uzel označen jako Clearing Transform.

Samotný HUD si při vytvoření nastaví matici na 2D ortografickou projekci. Jeho stateset je pozměněn tak, aby se geometrie vždy zobrazila (vypnutím depth testu) a je povolen alpha blending po průhledné či poloprůhledné prvky. Při tomto postupu jsem se inspiroval tutoriálem ze stránek OSG o vytváření HUDu [19].

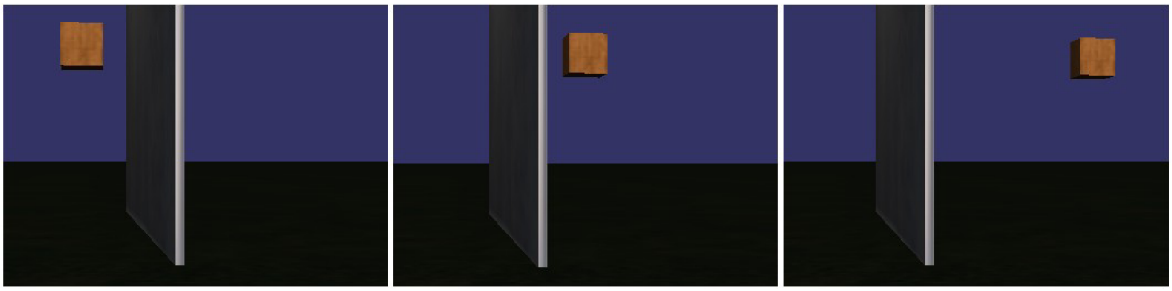
Pro připojování k HUDu jsem vytvořil dvě jednoduché třídy BoxElement a TextElement a složitější třídu OverlayWindow. OverayWindow je tvořena čtyřmi BoxElementy pro vykreslení okrajů a jedním pro vykreslení pozadí obsahu.

4.1.7 Kolize

V ODE je možné nastavit Geomům bitové pole udávající kategorie, do kterých Geom patří. Toto bitové pole má garantovanou délku 32 bitů. V mé implementaci jsem využil horních 16 bitů k určení materiálu a spodních 16 bitů k určení typu objektu do kterého Geom patří. Z 32 bitů jsem takto využil 18.

Všechnu kolizní odezvu v demonstrační scéně řeším v jedné funkci. Pro některé dvojice Geomů je vykonána speciální odezva (paprsky, zbraně při sbírání), pro zbytek Geomů je standardní odezvou vytvoření Contact Jointu. Tento je vždy parametrizován alespoň tak, aby nabízel Coulombovo tření. K tomuto účelu jsou pro všechny dvojice materiálů nastaveny koeficienty tření. Při určování těchto hodnot jsem čerpal z [20]. Pro ty dvojice materiálů, pro které se mi nepodařilo najít odpovídající koeficient, jsem ho zvolil úsudkem.

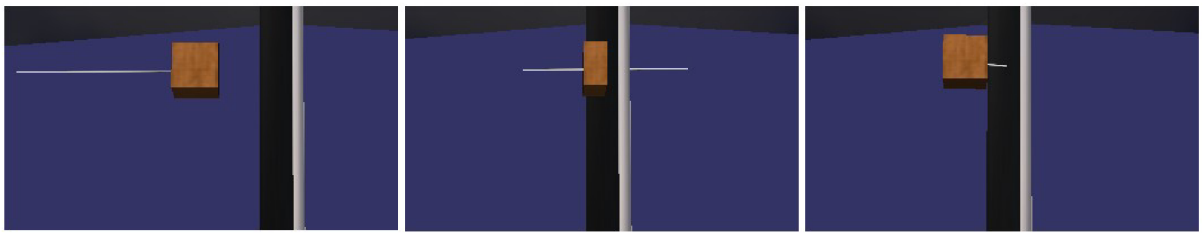
Paprsek proti tunelování



Obrázek 13: rychle letící krychle protuneluje skrz zeď

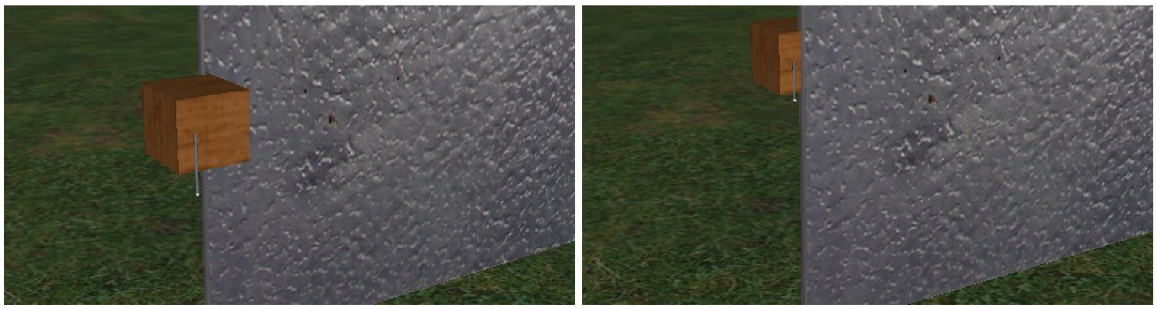
Objekty u kterých je velká pravděpodobnost, že se budou pohybovat ve vysokých rychlostech, mohou být opatřeny paprskem proti tunelování. V demonstrační aplikaci jsou takto ošetřeny vystřelované projektily (koule, krychle a rakety).

Takovémuto objektu je nastaven update, který každý krok testuje, zdali rychlost tělesa nepřevyšuje předem nastavený práh (vypočten z minimální tloušťky tělesa). Jestliže ano, pak je do scény umístěn paprsek. Tento paprsek směřuje z aktuální pozice tělesa do pozice, kde se těleso nacházelo v minulém kroku. Dále je zavolána detekce pro kolize paprsku se statickou i dynamickou simulační scénou. Pokud paprsku v cestě leží nějaké Geomy, vrátí kontaktní pozici s prvním Geomem ve svém směru. Na tuto pozici je následně přemístěno testované těleso. Ve fázi detekce kolizí bude tedy standardně rozpoznána kolize a vytvořen Contact Joint, jenž těleso vytlačí správným směrem.



Obrázek 14: Rychle letící krychle na nosném paprsku neprotuneluje skrz zeď

Tento přístup uspokojivě vyřeší většinu případů tunelování. Mohou ovšem nastat i takové okolnosti, při kterých paprsek tunelování nedetekuje (obrázek 15).



Obrázek 15: Paprsek nedetekoval tunelování skrz okraj zdi

4.2 Demonstrační scéna

Pokud geometrie ve scéně mají kolidovat nebo jsou součástí nějakého simulovaného tělesa , potom jsou připojeny do větve grafu pod uzlem osgODE::World. Ostatní geometrie (např. testovací debug geometrie) jsou umístěny na kořen celého grafu scény. Jestliže geometrie reprezentuje nějaký prvek na HUDu, je napojena pod uzel HUD.

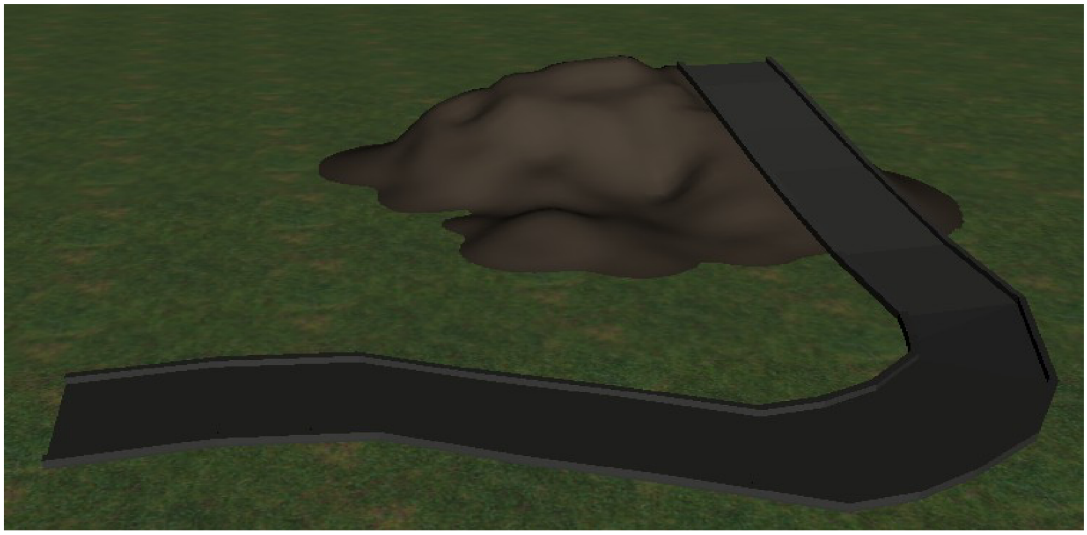
4.2.1 Simulační scéna

Objekty v simulované scéně mohou být buďto statické, v tom případě jsou reprezentovány pouze kolizní geometrií a jsou umístěny pod uzel Static Space. Nebo mohou být dynamické složené z jednoho či více pospojovaných těles. Potom jsou umístěny v grafu simulované scény pod uzel Dynamic Space.

Textury a modely

Použité textury byly staženy z [21]. Textury homogenních materiálů jsou nastaveny geometriím přímo v OSG. Pro použité jednoduché geometrie typu ShapeDrawable má OSG samo nastaveny texturovací souřadnice.

Složitější meshové geometrie byly vymodelovány v opensource programu Blender[22]. Jejich textury na ně byly naneseny tamtéž, pomocí UV mapování. Výsledek byl z Blenderu vyexportován do souborů .obj (model) a .mtl (materiál, textury). Tyto již nebyl problém načíst do OSG. Příklady použitých meshových geometrií jsou vidět na obrázcích 16 a 17.



Obrázek 16: Meshové geometrie statického kopce a dráhy

Obrázek 17: Meshová geometrie statického domku

Složitější objekty

Pro objekty, které jsou pospojované z více těles a mají implementovanou nějakou funkčnost pomocí callbacků slouží třída MyObject. Je poděděná z osgODE::SimpleSpace, pokud by tedy měl MyObject obsahovat větší množství geometrií, pak by měly být umístěny do optimalizovanějšího Spacu a ten připojen jako potomek na MyObject.

MyObject umožňuje aktivovat update na přepočítávání středu hmotnosti objektu (center of mass) pomocí callbacku. To je řešeno pomocí váhového průměru hmotnostních středů jednotlivých

komponent a dobře popsáno v [16]. K dalším funkcím MyObjectu patří například zapínání a vypínání kolizí, schovávání objektu a vyjmutí objektu ze simulace.

Speciálním případem MyObjectu je MyControllableObject, který je přizpůsoben k tomu, aby se na něho připojovala kamera.

Následuje popis zajímavých objektů v demoscéně.

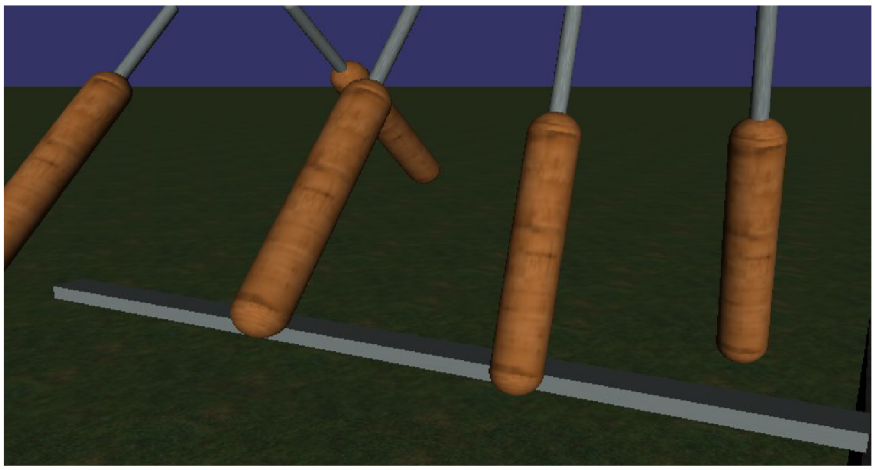
Dveře

Jednoduché spojení tělesa obsahujícího pouze geometrii kvádru a statického světa pomocí Jointu typu Hinge.

Obrázek 18: Dynamické dveře

Kyvadla

Kompozitní tělesa vytvořená spojením dvou kapslí. Jsou spojeny se statickým světem pomocí motorizovaných Hinge Jointů. V updatu je kontrolován současný úhel pro každé kyvadlo a pokud je menší než nastavené minimum nebo větší než nastavené maximum, potom je motoru kyvadla změněna rychlost na opačnou. Motor kyvadla je schopen působit velikou silou, aby nebylo možné jej přetlačit.



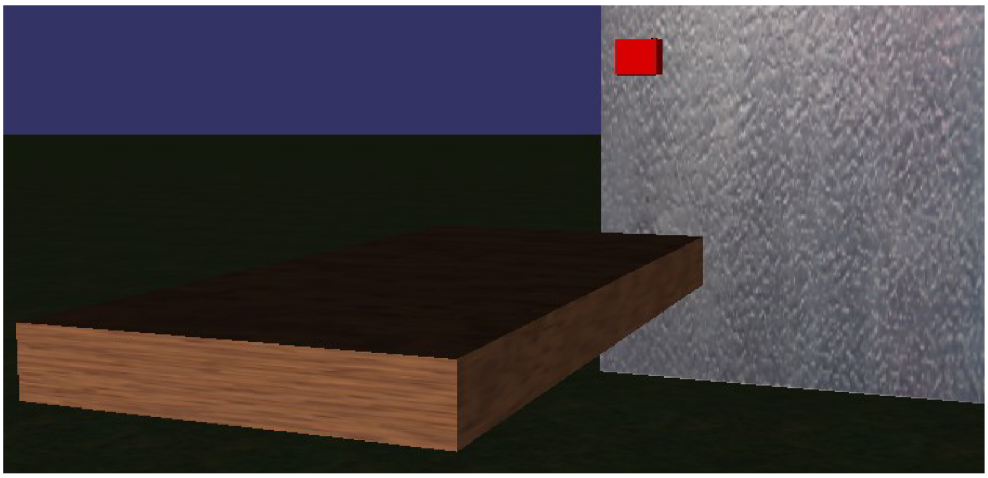
Obrázek 19: dynamická kyvadla nad kladinou

Tlačítkem ovládaná plošina

MyObject simulující pohyblivou plošinu, ovládanou tlačítkem. Tlačítko i plošina jsou tělesa obsahující kvádrovou geometrii. Obě tělesa jsou spojeny se statickým světem pomocí Motorizovaných Slider Jointů.

Update callback hlídá pozici tlačítka a pokud je posunuto na slideru za nastavený práh, provede se stisknutí tlačítka a následná obnova. Stisknutí tlačítka znamená, že se zapne či vypne pohyblivá plošina. Obnova tlačítka je provedena vytlačením tlačítka na původní pozici pomocí limitu Jointu, který je nastaven tak, že současná pozice ho porušuje a tlačítko je vytlačeno, aby znovu splňovalo omezení. Rychlost tohoto vytlačení je nastavena na relativně pomalou pomocí malého ERP. Až se tlačítko vrátí do původní polohy, obnova se zruší.

Plošina má nastavenou dobu jakou čeká na svých limitních pozicích, než se vrátí zpět. Toto je řešeno naplánováním druhého callbacku.



Obrázek 20: Tlačítko a plošina

Hráčův avatar

Hráč (player) je reprezentován často používaným modelem vzpřímené kapsle na paprsku. Pomocné informace při implementaci tohoto modelu jsem nalezl na ODE Wiki. Hráčovi ruce jsou spojeny s torzem pomocí Hinge Jointu (ramena). Torzo je spojeno se statickým světem pomocí AngularMotor Jointu. Ten je nastaven do Euler módu a jeho osy jsou nastaveny na osy globálního souřadného systému. Rotace torza okolo Z osy (roll) a X osy (pitch) jsou zakázány nastavením spodního i horního limitu AngularMotoru pro tyto osy na 0. Tímto je docíleno vzpřímenosti.

Player je poděděn ze třídy MyControllableObject a je tedy schopen nést kameru. Každý simulační krok se player snaží dorovnat pitch a yaw nastavený na kameře. Svým motorizovaným Jointem na ramenou dorovnává pitch a na jedné z os (na té jediné bez limitů) AngularMotoru dorovnává yaw.

Dále si player každý krok detekuje, jestli na něčem nestojí, vystřelováním paprsku pod sebe. Tento paprsek je nastaven tak, aby vracel feedback s nejbližším Geomem ve směru vrhu. Pokud je tento feedback vrácen znamená to, že player na něčem stojí. Jako odezva je vytvořen Contact Joint, ve kterém je paprsek v kolidující dvojici Geomů nahrazen Geomem torza. U Contact Jointu je nastaveno nízké ERP, aby player plynule přecházel přes překážky, zdolával schody atd.

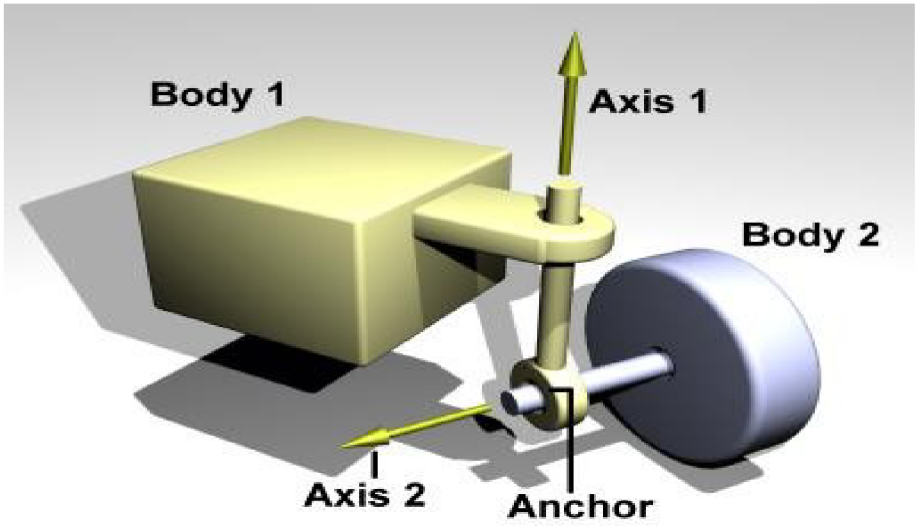
Pokud player na něčem stojí, potom může skákat. Player má nastavenou maximální výšku, do které doskočí při maximálním odraze. Odrazem se uvažuje podržení skákací klávesy (mezerník). Samotný odraz je tedy proveden až při uvolnění této klávesy. Pokud je player ve vzduchu, mění svoji rychlost pomaleji. V demoaplikaci je tato kontrola ve vzduchu (air control) nastavena na polovinu kontroly na povrchu.

Pohyb playera je řešen aplikováním sil do jeho středu hmotnosti. K tomuto účelu lze také použít LinearMotor Joint mezi torzem a statickým světem, ale lepšího výsledku se mi podařilo dosáhnout pomocí přímého aplikování sil. Když se směr pohybu playera změní (je jedno jestli ho změnil player sám nebo okolí), player se pokusí pohyb v nežádoucím směru vyrušit silou. Tuto sílu jsem nazval counter force. Rychlost pohybu playera je také upravena podle změny zanoření nosného paprsku. Toto způsobí, že při dopadu z výšky a při stoupání do svahu nebo do schodů bude player zpomalen.

Obrázek 21: Ovladatelný hráčský avatar

Auto

Druhým ovladatelným objektem demostrační scený je auto. Je složeno z pěti těles. Prvním je tělo auta, které je kompozitní a složené z dvou kvádrových Geomů (šasi a kabina). Tělo auta je připojeno ke čtyřem kolům pomocí Hinge2 jointů. Jedná se o pokročilejší druh ODE Jointu , který umožňuje nastavovat rotace okolo dvou os a skvěle se hodí na připojení kol. Hinge2 Joint je pro názornost na obrázku 22. Okolo první osy (axis1) je u auta prováděno zatáčení kol, kolem druhé osy (axis2) pak rotztáčení kol.

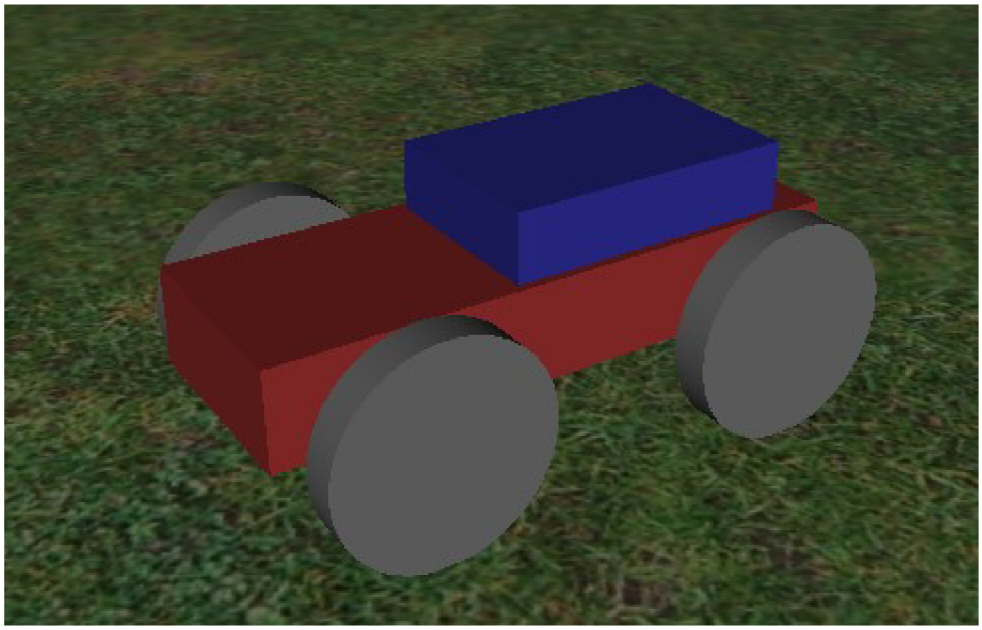


Obrázek 22: Hinge2 joint, obrázek převzat z [3]

Zatáčení i roztáčení kol je provedeno pomocí Joint motorů. Auto má pohon na všechny čtyři kola. Rychlost zatáčení je závislá na rychlosti kol. Maximální dosažitelná rychlost auta v demonstrační scéně je okolo 20 m/s.

Hlavním problémem při simulování auta v ODE je převracení nebo podkluzování v zatáčkách. Převracení se mi podařilo zmírnit implementací takzvaných sway-bars. Toto řešení jsem nalezl na ODE Wiki. Podkluzování vymizelo po vyladění parametrů na Contact Jointech pro kolize kol, zejména parametry ContactSlip1 a ContactSlip2. Dále jsem použil velice jednoduchou aproximaci funkce diferenciálu. V podstatě jen při zatáčení zrychluje vnější kola a zpomaluje vnitřní.

Auto si každý krok kontroluje síly působící na různá kola z ODE struktury JointFeedback. Tuto strukturu je možné alokovat a přiřadit každému Jointu. Pokud je síla větší než nastavený práh, inkrementuje se počitadlo poškození. Pokud poškození dosáhne určité hodnoty, kolo se rozpojí.



Obrázek 23: Ovladatelné auto

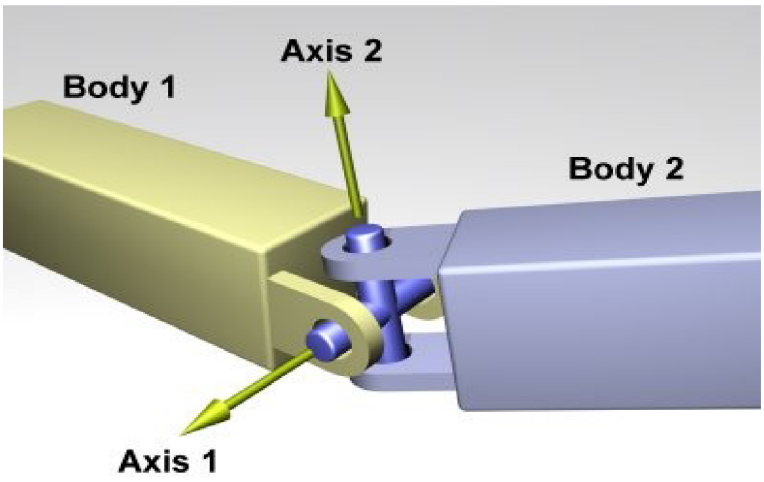
Zbraně

V demonstrační scéně jsou umístěny tři zbraně. Player je sebere kolizí s nimi. Zbraň se po sebrání připojí dvěma Jointy k jeho pravé ruce. Prvním je BallSocket pro určení pozice uchycení zbraně a druhým je AngularMotor pro nastavení rotace zbraně pomocí limitů na všech třech osách. Pokud player vlastní více zbraní, může si je přepínat klávesami 1, 2, 3. Při přepnutí zbraně je současná zbraň odpojena, vyjmuta ze simulace, jsou jí zakázány kolize a je jí zrušeno zobrazení. Nově držená zbraň je naopak připojena k pravé ruce, je zobrazena a jsou jí povoleny kolize a simulace.

První zbraní je konstruktor, tato zbraň vystřeluje levým tlačítkem železné koule o hmotnosti 10kg a pravým tlačítkem dřevěné kostky o hmotnosti 5kg. Delším podržením tlačítka je objekt vystřelen větší rychlostí. Po výstřelu je vytvořen projektil před zbraní a je na něho aplikována síla ve směru výstřelu (směr výstřelu je zjištěn pomocí paprsku kamery). Poloviční síla v opačném směru je aplikována na hráčovu pravou ruku jako zpětný ráz zbraně. Maximální impulz výstřelu je 1000N.

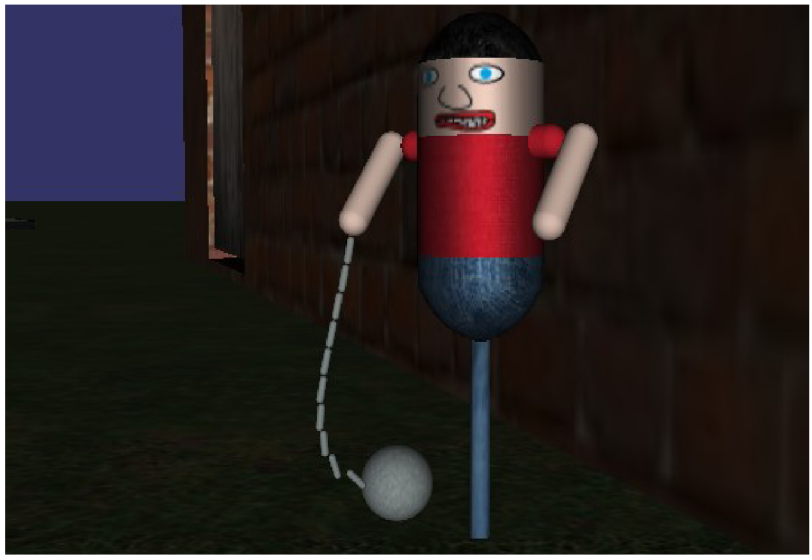
Druhou zbraní je raketomet. Ten také při výstřelu vytvoří raketu před hlavní zbraně a nastaví ji nízkou počáteční rychlost. Raketa je každý simulační krok urychlována ve směru svého letu, dokud nenarazí. Pokud nenarazí do 10ti sekund, akcelerace přestane (simulace vyhoření paliva) a raketa s pomocí gravitace spadne.

Třetí zbraní je řemdih. Demonstruje simulaci řetězu v ODE. Řetěz je tvořen z kapslí pospojovaných Universal Jointy (viz obrázek 24). Toto řešení neumožňuje článkům řetězu rotaci okolo podélné osy, ale je nejstabilnější, k jakému jsem dospěl. Zbraně jsou vidět na obrázcích 25 a 26.



Obrázek 24: Universal Joint

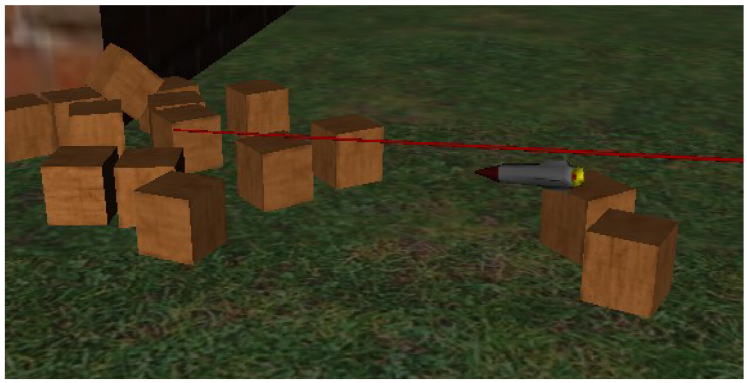
Obrázek 25: vlevo nahoře raketomet, vpravo nahoře konstuktor



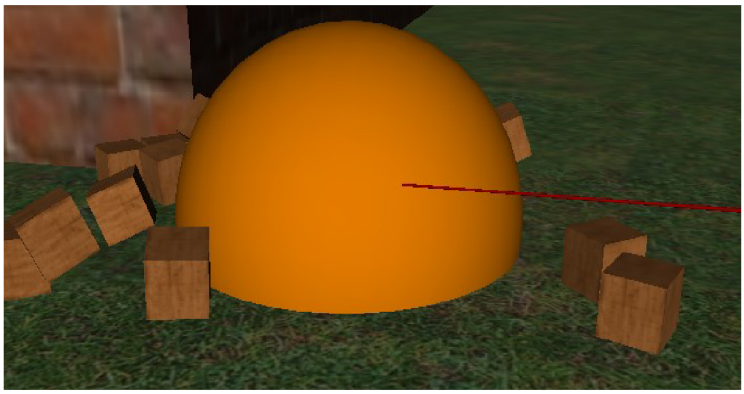
Obrázek 26: zbraň řemdih, složená z pospojovaných kapslí a koule

Exploze raket

Odezvou na kolizi rakety je naplánování callbacku, který raketu v příštím simulačním kroku odstraní a vytvoří těleso exploze. Tělesem exploze je koule, která se v každém simulačním kroku zvětšuje a mění barvu. Nepůsobí na ní gravitace a tělesa se kterými koliduje vystrkuje silou směrem od svého středu. Exploze je po odeznění odstraněna.



Obrázek 27: raleta letící k cíli



Obrázek 28: raketa byla nahrazena explozí

4.2.2 HUD prvky

Na obrazovce je zobrazeno několik informačních prvků. Je to například ukazatel počtu snímků za sekundu (fps), rychlosti kontrolovaného objektu, ukazatel poškození kol u auta a používaná zbraň u playera.

Po stisknutí F1 je zobrazeno poloprůhledné okno s ovládáním. Po stistknutí F2 se zobrazí pár užitečných rad jak získat všechny zbraně.

4.3 Ladění a optimalizace

Při vývoji aplikace byly OSG a ODE použity v debug verzích. V aplikaci je velké množství assert maker pro kontrolu hazardních stavů. Tyto lze vypnout kompilačním parametrem NDEBUG. Stejným define jsou ošetřeny i další ladící funkce. Například změna rychlosti a pauzování simulace. Pokud jsou tyto funkce vyžadovány, je třeba přeložit aplikaci bez NDEBUG.

Optimalizace byla zčásti provedena chytrým space partitioningem, ale je ještě hodně prostoru pro vylepšení. Například přestat používat pro zobrazení geometrií osg::ShapeDrawable, který používá OpenGL immediate mode. Výsledná scéna byla sestavena tak, aby běžela plynule na notebooku ASUS F3T s procesorem AMD Turion X2 1.6Ghz, 2GB RAM a grafickou kartou Nvidia GeForce Go 7600 256MB, nainstalovány byly Windows 7. Což byla také hlavní testovací sestava. Dále byla aplikace otestována na dvou dalších silnějších desktop sestavách, kde běžela bez problémů. Aplikaci se mi zatím bohužel nepodařilo otestovat na linuxu.

5 Závěr

Výsledkem této práce je aplikace demonstrující funkčnost herního engine, který zobrazuje scénu pomocí OSG a simuluje pohyb a interakce těles pomocí ODE. Všechny simulované objekty, stejně jako 2D grafické prvky zobrazené na obrazovce, byly začleněny do jednoho grafu scény. Bylo použito několik druhů ODE Jointů, pomocí kterých bylo nasimulováno několik zajímavých pospojovaných těles.

Při vypracování této práce jsem si osvojil řadu technik organizace scény, zejména pro účely zrychlení kolizní detekce. Stejně tak i metody implementace pohyblivé a ovladatelné kamery. Obecně jsem nastudoval spousty informací o tvorbě her, které se mi v mých dalších projektech budou hodit.

Herní engine, který jsem naprogramoval chci dále rozvíjet a vylepšovat. Nejprve bych rád doladil osgODE wrapper a nabídnul ho veřejnosti. Také bych rád prozkoumal možnosti síťové komunikace ve hrách a upravil svůj engine aby umožňoval multiplayer. Serializace simulačního stavu ODE a jeho posílání po síti by mohlo být zajímavé. Plánuji také vyzkoušet fyzikální engine Bullet a porovnat ho s ODE.

Dalších možností rozšíření je nespočet, například počítačem ovládané objekty s umělou inteligencí, grafické efekty (particle efekty, shader efekty) a v neposlední řadě optimalizace. Některé z těchto nápadů bych, pokud mi to bude umožněno, rád realizoval při dalším studiu na FIT. Například jako součást diplomové práce.

Literatura

Všechny zde zmíněné WWW stránky byly dostupné dne 9.5.2010 a obsahovaly zamýšlený obsah.

[1] WWW stránka - *Physics Engine*, Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Physics_engine

[2] WWW stránka – *Bullet (software),* Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Bullet_(software)

[3] WWW stránky – *Open Dynamics Engine (ODE) Community Wiki*
http://opende.sourceforge.net/wiki/

[4] Vlákno diskuzního fóra – *Implicit/Explicit Integration,* gamedev.net fórum
http://www.gamedev.net/community/forums/topic.asp?topic_id=312093

[5] Christer Ericson : *Real-Time Collision Detection,* The Morgan Kaufmann Series in Interactive 3-D Technology, Sony Computer Entertainment America, USA, 2005, ISBN: 1-55860-732-3

[6] Prezentační slajdy – Cheryl D. Seals, Ph.D. : *Collision Detection,* Auburn University,
http://www.eng.auburn.edu/users/sealscd/COMP7970/Collision/4.2%20Collision%20Detection.seals.ppt

[7] Prezentační slajdy – Johannes Mezger : *Bounding Volume Hierarchies,* University of Tübingen,
http://www.gris.uni-tuebingen.de/people/staff/jmezger/papers/bvh.pdf

[8] Prezentační slajdy – John McCutchan : *Introduction to Collision Detection,* McMaster University,
http://www.cas.mcmaster.ca/~carette/SE3GB3/2006/notes/cd_pres.pdf

[9] WWW stránky – *OpenSceneGraph*,
http://www.openscenegraph.org/

[10] WWW stránky – *Open Dynamics Engine*,
http://www.ode.org/

[11] WWW stránky – *Simple Directmedia Layer*,
http://www.libsdl.org/

[12] Online tutorial – Michal Turek : *SDL: Hry nejen pro Linux,*
http://www.root.cz/serialy/sdl-hry-nejen-pro-linux/

[13] Online tutorial – Rob Morefield, Brian Malloy : *Using SDL and OSG,*
http://www.cs.clemson.edu/~malloy/courses/3dgames-2007/tutor/

[14] Diskuzní skupina – *ode-users,*
http://groups.google.com/group/ode-users

[15] Russel Smith : *How to make new joints in ODE*, 2002,
www.ode.org/joints.pdf

[16] David M. Bourg : *Physics for Game Developers*, O'Reilly & Associates, USA, 2002,
ISBN: 0-596-00006-5

[17] Online tutorial – Glenn Fiedler : *Fix your timestep!* ,
http://gafferongames.com/game-physics/fix-your-timestep/

[18] Dr. Ing. Petr Peringer : *Modelování a simulace – IMS - Studijní opora*, 2008, Brno

[19] Online tutorial – *osgText, HUD, RenderBins,*
http://www.openscenegraph.org/projects/osg/wiki/Support/Tutorials/HudsAndText

[20] WWW stránka – *Coefficient of Friction,*
http://www.engineershandbook.com/Tables/frictioncoefficients.htm

[21] WWW stránky – *CGTextures*,
http://www.cgtextures.com/

[22] WWW stránky – *Blender*,
http://www.blender.org/

Seznam příloh

Příloha 1. DVD – obsahuje zdrojové soubory, binární soubory, modely, textury, makefile a videoukázku.