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

Bakalářská práce

Vlastnosti fyzikálního enginu ODE

Plzeň, 2011 Jiří Janeček

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne: .....

Podpis autora

Abstrakt

Properties of physical engine ODE

The target of this bachelor thesis is to learn about basis of physical engine ODE (Open Dynamics Engine), to create a set of testing programs for demonstration of basic features of ODE programming interface, to measure the accuracy of these tests by applying of extreme condition, to find the stability of the problems and to make some suggestions for their solving or their reducing at least. Results obtained by testing programs can be used for adjusting of scene parameters so that we get better findings than in case of using of ordinary adjustments. Next task is to compare ODE with other physical engines based on the results of tests. Analyze of testing tasks is described in this thesis and appropriate testing sources are also a part of this project.

Obsah

1 Úvod

Fyzikální enginy se stále častěji stávají nedílnou součásti aplikací z herního a zábavního průmyslu, robotiky a jiných odvětvích zabývajících se virtuální realitou. Nároky na přesnost, rychlost a rozmanitost simulací jsou stále větší. Současné enginy musí zvládat vice typů simulací než jen dynamiku tuhých těles, ale další simulace oblečení a kapalin. Nesmí se ale zapomínat na ty nejzákladnější části enginů. Mezi ně patří právě simulace dynamiky tuhých těles. Za dobu nejméně deset let všechny enginy udělal velký pokrok. V současné době se jedná o velmi propracované části softwaru a konkurence mezi enginy je stále větší, a proto správná volba enginu pro konkrétní aplikaci musí být také náročnější úlohou.

V první řadě bude potřeba se seznámit s programovými prostředky *ODE*. Na základě získaných informací vytvořím ukázkové úlohy používající základní funkcionalitu enginu a postup tvorby zdokumentuji. Vytvořené úlohy budou sloužit jak k demonstraci používání programového prostředí, tak ověření stability enginu a jeho chování při extrémních podmínkách. Proto se v této práci pokusím vystihnout chování fyzikálního enginu *ODE* v některých typických situacích, objevit případné projevy nestability a navrhnout možná řešení nebo omezení nalezených problémů.

Pomocích získaných výsledků testování, porovnám výsledky *ODE* s jiným enginem (jmenovitě se bude jednat o fyzikální engine *Havok*) na podobných testovacích úlohách, které vznikly jako součást jiné bakalářské práce [3]. Závěrečným výstupem porovnání obou enginů bude rozhodnutí, který z výše jmenovaných prostředků nejlépe řešil danou úlohu.

2 Open Dynamics Engine

Fyzikální engine *ODE* je svobodná knihovna určená k simulaci dynamiky tuhých těles, proto se s ním můžeme setkat v aplikacích virtuální reality, simulacích pozemních vozidel nebo chodících zvířat. *ODE* obsahuje i modul detekce kolizí, popř. lze použít i jiné obdobné knihovny. Původním autorem této knihovny je Russell Smith, a nadále je *ODE* vyvíjeno za pomocí nemalého počtu přispěvatelů. *ODE* je kvalitou srovnatelné s proprietárními konkurenty jako jsou *PhysX* a *Havok*. *ODE* má zejména výhodu v tom, že je to *svobodný software*, který je možno volně rozšiřovat a modifikovat pod licencemi *GNU Lesser General Public License* a *BSD-style license*. Dále je *ODE* snadno přenositelné na různé platformy. Naopak nevýhodou je fakt, že *ODE* využívá k výpočtu pouze *CPU*, což *ODE* značně znevýhodňuje v otázkách rychlosti a využití specializovaného výpočetního prostředí (*GPU*), kterým se může naopak chlubit *PhysX*. *ODE* je primárně napsáno v *C++*, přestože má nativní *C* rozhraní. Existuje také i objektová verze rozhraní, ta ale není tak preferovaná. *ODE* používá vysoce stabilní integrátor, proto by chyby simulace neměly růst nade všechny meze. *ODE* klade důraz na rychlost a stabilitu na úkor fyzikální přesnosti.

3 Vlastnosti

Zde uvádím některé pojmy a prvky se kterými se v *ODE* setkáte:

- Objekty (`dBodyID`) musí být součástí tzv. světa `dWorldID`. Objekt je reprezentován následujícím výčtem vlastností proměnných v čase:
  - Pozicí těžiště objektu ve světových souřadnicích.
  - Rychlostí těžiště.
  - Rotací objektu reprezentovanou $3x3$ rotační maticí.
  - Úhlovou rychlostí.

  a vlastností konstantních v čase:
  - Absolutní hmotností objektu.
  - Matice setrvačnosti (*Inertia matrix*) obsahující informaci o rozložení hmoty okolo těžiště.
- Geometrie (`dGeomID`) se vkládá do kolizního prostoru (`dSpaceID`). Kolize se testují na geometriích (`dGeomID`). Reakce na kolize je spojeny s tzv. kontakty, které se vytváří na základě zjištěných vzájemných kolizí

geometrií. Geometrie může mít přiřazené nejvýše jedno těleso. Těleso musí mít jednu a více geometrií.

- Vazba (`dJointID`) spojuje dvě tělesa. V závislosti na typu vazby může být omezen vzájemný pohyb nebo rotace obou spojených těles. Dále je možné vazbu spojit se statickým prostředím.

- *Error Reduction Parameter (ERP)* je konstanta sloužící k minimalizaci chyb ve vazbách (*joints*). Hodnota 0 znamená, že nebude použitá žádná opravující síla. Pokud je nastavena 1, potom simulace se pokusí opravit všechny chyby vazeb během následujícího simulačního kroku. Doporučené hodnoty jsou mezi 0.1 a 0.8.

- *Constraint Force Mixing (CFM)* je konstanta určující měkkost vazby. Když je *CFM* rovno nule, vazba bude tvrdá. Když se hodnota *CFM* zvyšuje, zvyšuje se i měkkost vazby. Není doporučeno používat čísla záporná a čísla větší než jedna.

Referenčním manuál [5] celé knihovny velmi podrobně popisuje další charakteristiky práce s fyzikálním enginem *ODE*.

4 Instalace a použití knihovny

ODE je nyní šířeno pouze ve formě zdrojových kódů, protože binární podoba spíše znemožňuje přizpůsobit knihovnu na různé platformy a různá nastavení. Proto první částí této kapitoly bude popis překladu *ODE* pomocí nástroje *GNU make* pro platformy *GNU/Linux* a *Windows*. V případě překladu pod *Windows* je nutné používat nástroj *MinGW* spolu s interpretačním prostředím *Unixového shellu* nazývaném *MSYS*. Samozřejmě lze přeložit *ODE* i pomocí nástroje *MSVC*, ale to není naším cílem. Podrobné informace o překladu přes *MSVC* najdete buď v oficiálním manuálu [5], nebo lze další informace získat přímo z archivu se zdrojovými kódy, kde se v kořenovém adresáři projektu nachází soubor s názvem `INSTALL`.

Pokud máte stažené zdrojové kódy z *svn* repositáře je nutné na začátek provést jeden krok navíc:

```
sh autogen.sh
```

Pokud ale používáme stabilní verzi knihovny[1] můžete předchozí krok přeskočit a rovnou provést následující příkaz:

[1] Po celou dobu projektu jsem pracoval se stabilní verzí *0.11.1*.

```
./configure --prefix=/mingw --enable-double-precision
    --with-trimesh=opcode --disable-demos
    --with-drawstuff=none
```

Tento příklad konfigurace *make* systému jsem použil k překladu knihovny pro *Windows* pomocí *MinGW/MSYS*. Důležité je zejména nastavení přesnosti čísel v pohyblivé řádové čárce *double precision*. Následuje stručný popis těch nejdůležitějších voleb:

–**disable-demos** –**without-x** umožňuje překlad pro *Mac OS X*

–**enable-double-precision** zapíná podporu dvojité přesnosti čísel v pohyblivé řádové čárce

–**with-trimesh=opcode** zapne podporu *OPCODE trimesh* (výchozí)

–**with-trimesh=gimpact** zapne podporu *GIMPACT trimesh*

–**with-trimesh=none** vypne podporu *trimesh*

–**enable-new-trimesh** zapne podporu jiného modulu *OPCODE trimesh* (používat společně s `--with-trimesh=opcode`)

Kompletní seznam použitelných voleb získáte příkazem:

```
./configure --help
```

Následuje klasická sekvence příkazů, která přeloží zdrojové soubory, a nainstaluje knihovnu do nastavené složky `prefix` (výchozí je `/usr/local`):

```
make
make install
```

kde bych u poslední řádky upozornil, že v případě *GNU/Linux*(*Ubuntu*) bude třeba zadat `sudo make install`, abyste měli dostatečná práva k zápisu do složky `prefix`.

4.1 Překlad testů pod *Windows*

Pokud chcete překládat testovací příklady pod *Windows*, je možné využít binární verzi `libode.a`[2] zadáním příkazu:

```
mingw32-make -f Makefile.win clean
mingw32-make -f Makefile.win
```

[2]Knihovnu `libode.a` jsem překládal s *MinGW gcc* verze 4.5.0.

Pro obecné používání *ODE* jsou vyžadovány následující kroky, které jsou už samozřejmě dodrženy v `Makefile` souborech obsažených projektu:

- Přidat cestu k hlavičkovým souborům (např. pro tento projekt zadáme parametr `-I./include`).
- `include <ode/ode.h>`
- Přidat statickou knihovnu `*.a` (např. `-L./lib -lode`).
- Přidat deklaraci makra `dDOUBLE` (výchozí je `dSINGLE`, ale to není podporováno testovacími příklady).

4.2 Překlad testů pod *GNU/Linux*

Při překládání testovacích příkladů pod operačními systémy *GNU/Linux* je situace mnohem jednodušší. Většina dnešních distribucí *Linuxu* obsahuje správce balíčků a instalace všech závislostí[3] se přenechává na uživateli. Na rozdíl od *Windows*, kde jsem k projektu přiložil správné verze knihoven, v *Linuxu* musíme počítat s měnícím se prostředím, tzn. knihovny mohou být jinak pojmenovány napříč všemi distribucemi. Existují totiž dvě verze knihovny *ODE* (*single* nebo *double*, podle toho jak se rozhodneme překládat). Nelze tedy zaručit že přiložený `Makefile` soubor bude správně fungovat na všech distribucích a je třeba v něm provést úpravy ohledně názvu zvolené verze knihovny. Například v některých distribucích může název knihovny obsahovat slovo *double*. Příslušná úprava `Makefile` souboru je naznačena v následujícím příkladu:

```
CXXFLAGS = ... ‘pkg-config ode-double --cflags‘ ...
LDFLAGS = ... ‘pkg-config ode-double --libs‘ ...
```

kde jsme ve skutečnosti nastavili, že informace o překladu a sestavení jsou uvedeny v souboru `ode-double.pc`, který se zpravidla nachází v adresáři `/usr/lib/pkgconfig`.

5 Ukázkové úlohy

Tato kapitola se věnuje popisu tvorby testovacích programů a zejména poukazuje na klíčové prvky každého testu. V následujících odstavcích se budu

[3]Testy jsou závislé na knihovnách *SDL*, *SDL_ttf* a *ODE*, popř. *GL* a *GLU*.

věnovat technickému popisu společných prvků všech testů, které lze také považovat za návod k postavení základní aplikace používající fyzikální engine *ODE*.

Tato práce má úmyslně testy postavené co možná nejpřesněji podle zpracování bakalářské práce [3] ve fyzikálním enginu *Havok*, abych mohl provést co nejspravedlivější srovnání obou enginů.

Před tím než cokoliv z knihovny *ODE* použijeme musíme ji předem inicializovat.

```
dInitODE2(0);
dAllocateODEDataForThread(dAllocateMaskAll);
```

Vytvoříme svět (`world`) pro výpočet dynamiky tuhých těles, a dále vytvoříme kolizní prostor (`space`). Pro seskupení kontaktů vytvoříme *joint group* a uložíme do proměnné `contactGroup`.

```
world = dWorldCreate();
space = dSimpleSpaceCreate(0);
contactGroup = dJointGroupCreate(0);
```

Kontakty jsou speciální druh vazby (*joint*) reprezentující reakci tuhého tělesa na detekovanou kolizi s jiným objektem. Kontakt je vytvořen před simulačním krokem při detekci kolizí[4]. Po dokončení simulace musíme kontakty smazat. Kontakty jsou sdružovány do skupin proto, aby bylo možné všechny kontakty rychle smazat po každém simulačním kroku.

Teď zbývá nastavit globální *ERP* a *CFM*:

```
dWorldSetERP(world, 0.2);
dWorldSetCFM(world, 1e-6);
```

Konkrétní hodnoty nastavení *ERP* a *CFM* se budou samozřejmě pro každý test měnit. V následujících kapitolách se ukáže, jak velký význam mají tyto parametry pro stabilitu i nestabilitu scény, a proto je vždy potřeba s těmito parametry pracovat opatrně. Hodnoty uvedené v předchozím příkladu můžeme považovat za doporučené, protože představují dobrý základ pro obecnou scénu. Speciální scény budou tedy vyžadovat i speciální nastavení *ERP* a *CFM*.

Před každým vykreslením jednoho snímku je volána metoda `update`, která musí vypočíst simulační krok. Nadefinujeme velikost simulačního kroku:

```
#define NB_SECONDS_BY_STEP 0.001
```

[4]Výstupem funkce řídící proces detekce kolizí je množina kontaktů uložená ve skupině `contactGroup` mezi dvěma objekty.

Nyní následuje popis metody `update`, která má následující hlavičku:

```
void RenderWindow::update(double elapsed);
```

Tato metoda je volána při každém snímku (otočka smyčky zpráv odpovídá pojmenování v angličtině - *frame*) okna. Parametr `elapsed` obsahuje čas uplynutý od předchozí smyčky událostí. V metodě `update` nadefinujeme lokální statickou proměnnou `time_cache`, kterou se řídí opakování simulačního kroku, tak aby výsledná animace odpovídala reálnému času.

```
static dReal time_cache = 0.0;
```

Další ukázka upravuje hodnotu v proměnné `elapsed`. Tento bod je důležitý hlavně proto, že zamezíme neomezenému počtu opakování simulačního kroku. Toto omezení existuje kvůli počítačům, které nemůžou v reálném čase simulaci spočítat.

```
if (elapsed > MAXIMUM_ELAPSED_TIME)
{
    elapsed = MAXIMUM_ELAPSED_TIME;
}
```

Maximální čas snímku jsem stanovil na na hodnotu 1/100 sekundy:

```
const double MAXIMUM_ELAPSED_TIME = 1.0 / 100.0;
```

Následuje hlavní část celé simulace. Do proměnné `time_cache` se přičte uplynutý čas od předchozího snímku. Uvnitř cyklu se v každé iteraci sníží hodnota `time_cache` o hodnotu simulačního kroku. Smyčka běží tak dlouho, dokud se nevyčerpá čas uložený do `time_cache`. Může se stát, že uplynutý čas není přesným násobkem délky simulačního kroku a může dojít k předčasnému nebo pozdnímu ukončení smyčky. Možná odchylka v časování je kompenzována přenosem zbylého nebo chybějícího času do dalšího snímku pomocí statické proměnné `time_cache`.

```
time_cache += elapsed;
while (time_cache >= NB_SECONDS_BY_STEP)
{
    // Detekce kolizí
    dSpaceCollide(space, this, nearCallback);
    // Jeden simulační krok
    dWorldQuickStep(world, NB_SECONDS_BY_STEP);
    // Smazat všechny dočasné kontakty
    dJointGroupEmpty(contactGroup);
    time_cache -= NB_SECONDS_BY_STEP;
}
```

Uvnitř smyčky se provádí nutné kroky utvářející celou simulaci. Nejprve detekujeme kolize mezi objekty, o což se postará metoda `dSpaceCollide` s využitím `nearCallback`. Kolize se reprezentují v podobě vazby (*joint*) nazývané kontakt. Dál proběhne výpočet simulace a na závěr vyprázdníme celou skupinu kontaktů `contactGroup`.

V metodě `nearCallback` je realizován výpočet kolize a vytvoření příslušného kontaktu. Její hlavička musí vždy vypadat takto:

```
void dNearCallback(void *data, dGeomID o1, dGeomID o2);
```

Jak už hlavička napovídá, tak tato metoda počítá kolizi mezi dvěma objekty popsanými geometriemi `o1` a `o2`:

```
dContact contact[MAX_CONTACTS]; // maximální počet kontaktů

dBodyID b1 = dGeomGetBody(o1);  // získaní BodyID podle GeomID
dBodyID b2 = dGeomGetBody(o2);

// Vyhledání všech kontaktů
int numc = dCollide(o1, o2, MAX_CONTACTS,
    &contact[0].geom, sizeof(dContact));

for (i = 0; i < numc; i++)
{
    contact[i].surface.mode = ...;
    ...
```

Funkce `dCollide` naplní pole `contact` kontakty a vrátí jejich počet `numc`. Pro každý kontakt nastavíme vlastnosti materiálu a vytvoří se dočasné vazby (*Contact Joints*).

```
    dJointID c = dJointCreateContact(world, contactGroup,
        contact + i);
    dJointAttach(c, b1, b2);
}
```

Před vypnutím programu je vhodné provést příslušný úklid voláním následujících příkazů:

```
dJointGroupDestroy(contactGroup);
dSpaceDestroy(space);
dWorldDestroy(world);
dCloseODE();
```

aby *ODE* mohlo provést uvolnění interně alokované paměti.

Všechny testy jsou vytvářeny na základě dědění funkcionality ze třídy `AbstractWindow`. Snahou bylo vytvořit prostředí, které maximální způsobem zakrývá procesy vizualizace scény a odbourává požadavek opakovat některé sekvence pro každý test zvlášť. Všechny `*.cpp` soubory představující zdrojový kód testů, obsahují pouze sekvence týkající se programování s knihovnou *ODE*. Výjimku tvoří některé moduly v adresáři **utils**, kde můžete nalézt definice tříd `Plane`, `Box` a `Sphere`, které zapouzdří některou funkcionalitu *ODE* do více přehlednějšího rozhraní. Tyto třídy dále obsahují mechanismus na kreslení příslušných objektů pomocí *OpenGL*.

5.1 Test č.0 - Box & Sphere

První test je ukázková úloha nejjednodušší simulace. Ve scéně máme pevně umístěnou desku, na kterou můžeme postupně vypouštět krychle nebo koule náhodných rozměrů. Jednotlivé objekty na sebe padají a narážení na sebe a popř. se můžou mírně odrazit.

Statickou desku vytvoříme jako krychli (`Box`) o nějakých rozumných rozměrech, s tím že některé objekty dopadající na tuto desku mohou přepadnout přes okraj ven ze scény. Zvolil jsem velikost $[10.0, 0.2, 10.0]$[5] a pozici $[0.0, -0.1, 0.0]$, aby horní plocha desky byla v nulové výšce. V konstruktoru třídy `Box` se vytváří jak geometrie objektu (v případě krychle je to *display list*) pro kreslení do *OpenGL*, tak je třeba vytvořit obdobnou instanci objektu v *ODE*. Nejprve se vytvoří těleso (*body*). Vytvořenému tělesu přiřadíme hmotnost. Pak vytvoříme geometrii, a přiřadíme k ní těleso.

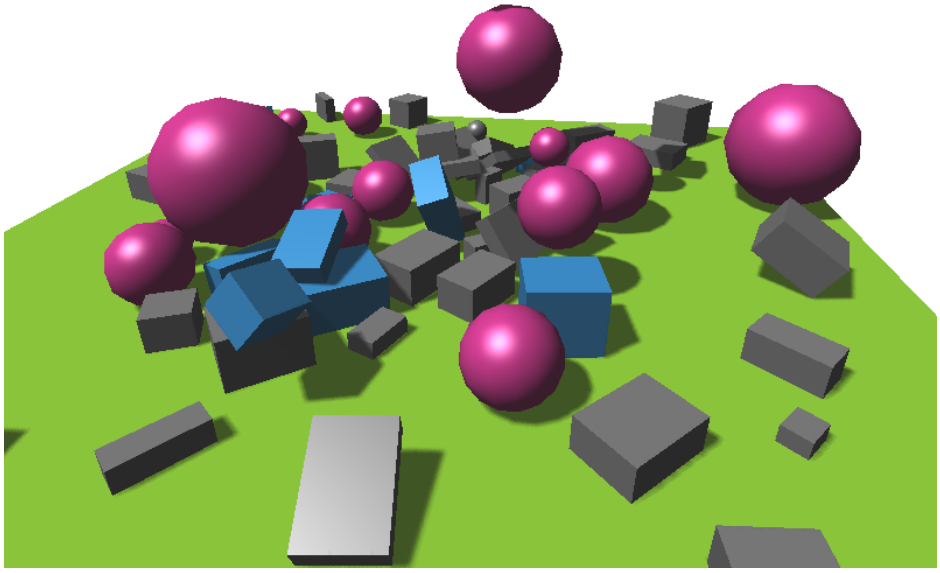
```
dMass m;
dBodyID body = dBodyCreate(world);

dMassSetBox(&m, DENSITY, width, height, depth);
dMassAdjust(&m, mass);      // celková hmotnost bude rovna mass
dBodySetMass(body, &m);

dGeomID geom = dCreateBox(space, width, height, depth);
dGeomSetBody(geom, body);
```

Dále nastavíme objekt jako statický, tím že smažeme připojené těleso nebo vytvoříme objekt popsaný pouze geometrií. Tento úkon je ukryt v metodě `setAsFixedGeom`:

[5] Rozměry jsou zapsány v pořadí: šířka v ose x, výška v ose y, a délka v ose z.



Obrázek 1: Ukázka scény z testu *Box and Sphere*

```
dGeomSetBody(geom, 0);
dBodyDestroy(body);
body = 0;
```

Metoda `setAsFixedGeom` je definována ve třídě `AbstractObject`. Výhodou dědění od třídy `AbstractObject` je fakt, že všichni potomci mají tuto funkcionalitu k dipozici právě díky předkovi.

Následuje vytváření různých objektů na pozici $[0.0, 10.0, 0.0]$. Krychle se vytváří klávesou **B**, a koule se vytváří klávesou **N**. Pohyblivé objekty vytváří stejně jako statické, jen vynecháme poslední jmenovaný krok. Dále musíme vzít v úvahu automatickou deaktivaci objektů, které se podstatě nepohybují (šetření výpočetního času).

```
dWorldSetAutoDisableFlag(world, 1);
```

Deaktivace objektu může mít ale i některá úskalí. Bez působení jiného objektu může vypnutý objekt zůstat jen tak viset v prostoru nebo zůstane v nějaké jiné nepřirozené poloze. Pouze další kolize s jiným objektem jej znovu aktivuje a uvede dočasně do přirozené polohy. K problémům může dojít ve chvíli, kdy například jeden objekt podpírá druhý objekt a ten první

je po uplynutí mezní doby smazán (protože objekty mají omezenou délku života), můžeme objekt dostat do nepřirozené polohy. Tuto vlastnost nechávám zapnutou aby dopad této metody byl patrný na první pohled. Deaktivované objekty poznáme podle šedé barvy, ostatní aktivní objekty mají barevný materiál. Pro další ušetření výkonu je kolem scény vyměřena hranice prostoru. Všechny objekty, které se za tuto hranici dostanou, jsou smazány.

5.2 Test č.1 - Jumping Balls

Úkolem tohoto testu je demonstrovat přesnost výpočtu odrazu na různých materiálových vlastnostech.

Scéna obsahuje řadu padesáti koulí padajících z výšky 7.0 nad základní rovinou. Tato řada je složena koulí se skákavostí od nejmenších hodnot až po dokonale obrazivý materiál.

Nejprve je nutné nastavit vhodné gravitační zrychlení. Zvolíme čtvrtinovou gravitaci než je na zemi a tím zpomalíme rychlost animace. Vektor zrychlení, jak je patrné z následujícího příkladu, musí směřovat v záporném směru osy y.

```
#define SURFACE_GRAVITY 9.823
dWorldSetGravity(world, 0, -0.25 * SURFACE_GRAVITY, 0);
```

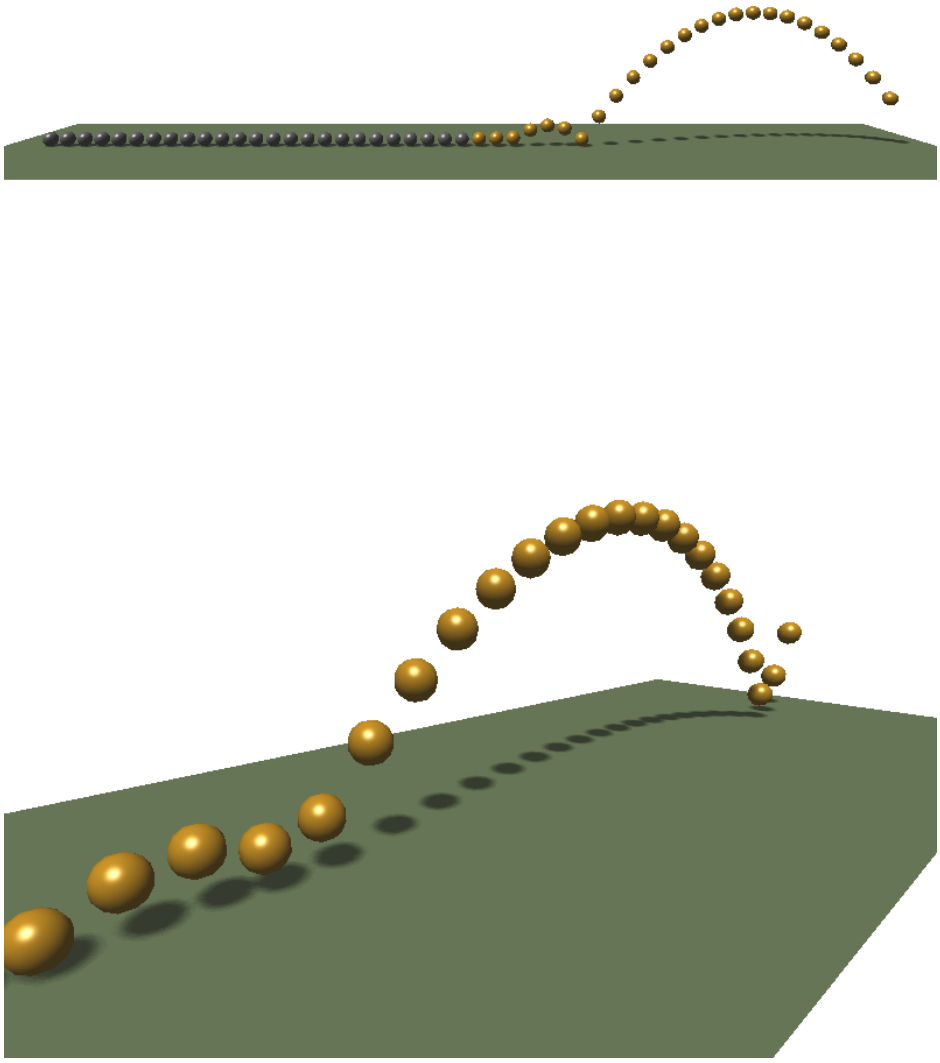
Dále vytvoříme všechny testovací koule a nastavíme u nich příslušný parametr odrazu funkcí `dGeomSetData`.

```
#define BALLS_COUNT 50
for (int i = 0; i < BALLS_COUNT; i++)
{
    dReal relPos = (dReal)i / (BALLS_COUNT - 1);    // 0..1
    balls[i] = new Sphere(world, space, 0.15, 1.0);
    ...
    balls[i]->setData(&relPos, sizeof(dReal));
}
```

Použití parametrů skákavosti najdeme v metodě `nearCallback`. Získáme ukazatel na uživatelská data a podle získané adresy uživatelská data zkopírujeme do pomocné proměnné:

```
dReal bounce;
void *p_data = dGeomGetData(o1);
memcpy(&bounce, p_data, sizeof(dReal));
```

Potom můžeme nastavit vlastnosti kontaktů:



Obrázek 2: Ukázka scény z testu *Jumping Balls*

```
for (i = 0; i < numc; i++)
{
    contact[i].surface.mode = dContactBounce;
    ...
    contact[i].surface.bounce = bounce;
    contact[i].surface.bounce_vel = 0.01;
}
```

Bitovým příznakem `dContactBounce` nastavujeme povrch jako skákavý. Síla odrazivosti je řízena parametrem `bounce`.

bounce Parametr restituce (0..1). 0 znamená, že povrch není vůbec odrazivý, 1 znamená maximální odrazivost.

bounce_vel Minimální příchozí rychlost nutná pro odraz. Rychlosti pod touto hranicí budou mít parametr restituce roven nule.

Celou scénu lze vrátit do výchozího stavu stiskem klávesy **R**.

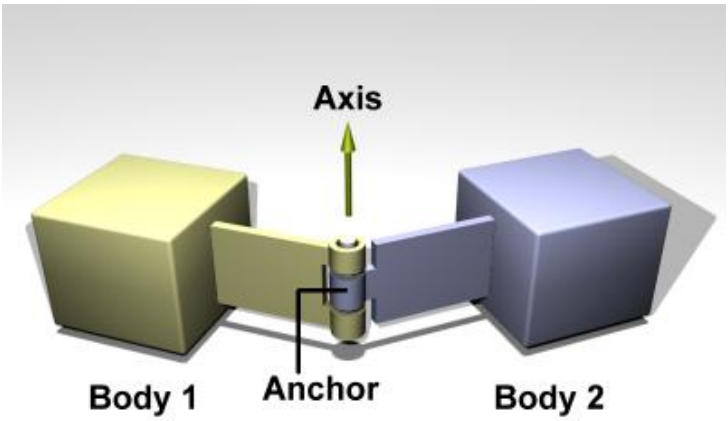
V přiloženém testu jsem *ERP* a *CFM* nastavil na nulu, i když nastavení zejména *ERP* takovýmto způsobem není obecně dobrou praktikou, protože ve většině případů by nastávaly spíše problémy. Scéna se přes takto zvolené nastavení chová podle všech představ. Nastavení *ERP* na nulu jsem zvolil proto, že u objektů s malou odrazivostí byla narušena byť jen nepatrná přesnost odrazu. *CFM* je vhodné nastavit na nulu nebo na hodnoty velmi blízko nule (čísla v řádech 10^{-6} a méně), například *CFM* přibližně rovno 0.001 způsobí úplný útlum všech koulí nezávisle na koeficientu odrazu (restituce).

Rozšiřující analýzu problému dokonalého odrazu můžete nalézt v kapitole 5.10 na straně 28.

5.3 Test č.2 - Kyvadlo

Cílem tohoto testu bylo vytvořit simulaci matematického kyvadla. Počáteční poloha kyvadla odpovídá úhlu vychýlení $\pi/2$ rad, čili kyvadlo je ve vodorovné poloze. Příklad ukazuje zejména jak vytvořit kloub (*joint*) mezi objekty a jak vytvořit složený objekt (*kompozitní objekt*), který se chová jako jedno primitivum.

Co se týká ovládání testu, je možné použít klávesu **R** na obnovení pozice a rotace kyvadla do výchozího stavu. Dále se v rámci jmenované akce přemaže soubor `Test2_output.txt` obsahující informace o stavu energií kyvadla (Energie potenciální, kinetická a celková energie systému) ve zvolených okamžicích (pravidelné vzorkování s pevným intervalem). Klávesa **F3** nastavuje hodnotu *ERP*, klávesa **F4** nastavuje hodnotu *CFM* a klávesa **F2** nastavuje



Obrázek 3: Hinge joint (převzato z *ODE* Community Wiki)

nastavuje krok, s jakým budou měněny *ERP* a *CFM*. Navíc je potřeba použití modifikátoru **Shift**, který otáčí (nastavuje zmenšení) změnu hodnot dříve uvedených.

Ve scéně je umístěna základní rovina ve výšce 0.0. Dále vytvoříme dva objekty, které budou tvořit geometrii budoucího kyvadla. Jedná se o jeden objekt krychle vytvarovaný do tvaru tyče a jeden objekt typu koule. Objekty musí být umístěny relativně vůči sobě, tak jak je budeme chtít mít posunuté, ve chvíli kdy je sloučíme. Sloučení dvou objektů lze provést metodou `attach`:

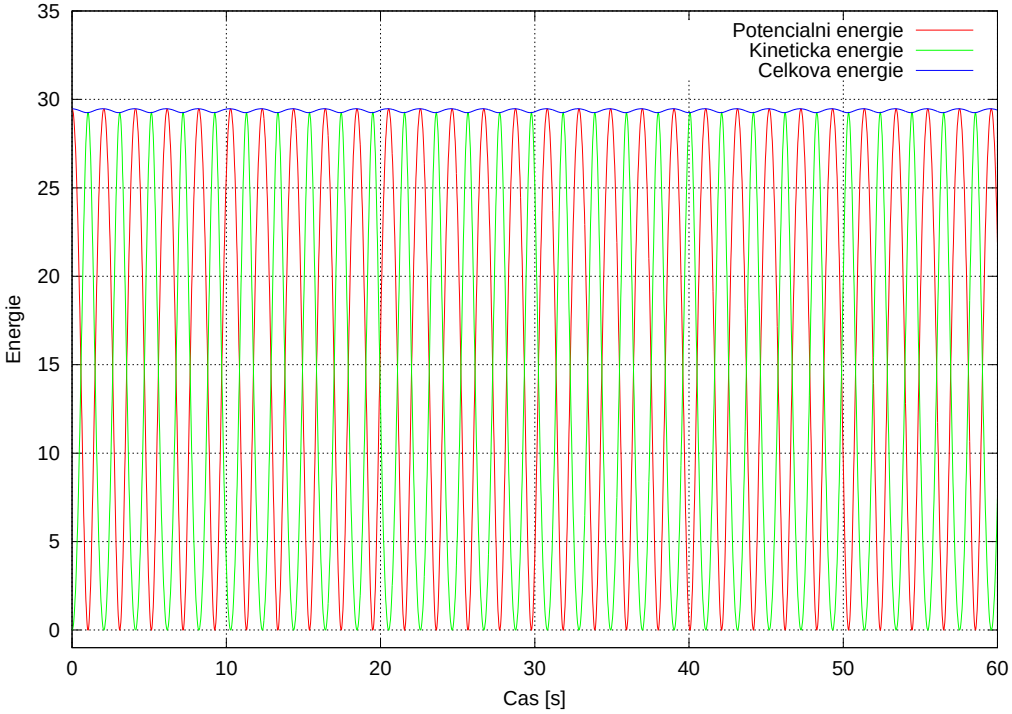
```
void AbstractObject::attach(const AbstractObject &other);
```

která má vstup jako jiný objekt třídy `AbstractObject`. Sloučení je realizováno sloučením hmotností obou těles do jednoho tělesa`dBodyID`. To znamená že všechny objekty, které obsahují stejné `dBodyID`, tvoří dohromady tzv. *composite object*, přičemž každá instance třídy `AbstractObject` obsahuje unikátní geometrii `dGeomID`. Popis implementace metody `attach` najdete v kapitole 8 na straně 35.

Zbývá jen vytvořit závěs kyvadla. Realizace probíhá, jak již bylo zmíněno, pomocí tzv. *joints*. Konkrétně použijeme vazbu nazývanou *hinge* (obrázek 3 na straně 15). Vazba se vytváří následujícím způsobem:

```
dJointID fixedHinge = dJointCreateHinge(world, 0);
```

První parametr metody vytvářejí vazbu je typu `dWorldID`. Druhý parametr je typu `dJointGroupID` označující pomyslnou skupinu vazeb. Vložená nula pak



Obrázek 4: Energie kyvadla při $ERP = 0.4$ a $CFM = 0.0$

znamená, že vytváříme samostatnou vazbu. Další krok je připojení objektů pod právě vytvořenou vazbu:

```
dJointAttach(fixedHinge, rod->getBody(), 0);
```

kde první parametr je *id* uchycení kyvadla, druhý parametr je *id* kompozitního objektu a poslední parametr je 0. Pokud je právě jeden z parametrů nulový říkáme, že je *joint* připojen ke statickému prostředí. Teď už zbývá nastavit osu otáčení (*axis*) a pozici vazby (*anchor*):

```
dJointSetHingeAnchor(fixedHinge, 0.0, 7.0, 0.0);
dJointSetHingeAxis(fixedHinge, 0.0, 0.0, 1.0);
```

Tento test dopadl úspěšně. Výsledek testu nejlépe vystihuje graf na obrázku 4 na straně 16. Červenou čárou je vyznačena potenciální energie, zelenou kinetická energie a modře je pak vyznačena celková energie (přímý součet potenciální a kinetické energie). Jak je vidět, kyvadlo si po celou dobu simulace udržuje celkovou energii na původní hodnotě. Jinak perfektní chování kyvadla kazí drobné periodické kolísání celkové energie. Zdá se že energie klesne, když je kyvadlo kolem rovnovážné polohy a po návratu do

horní úvrati se energie vrátí k normálu. Tento zvláštní jev se mi nepodařilo eliminovat. Určitě se nejedná o nějaký projev aliasingu (nedostatečného vzorkování), protože i při zaznamenání stavu při každém snímku zmiňovaný problém nezmizí. Nabyl jsem přesvědčení, že stejná chyba se také projevuje v testu 8 v kapitole 5.9 na straně 26, kde dochází k mírnému prodloužení kyvadla v těch samých místech, ve kterých tento test ztrácí energii.

Podstatným nastavením je hodnota *CFM*. Pokusy jsem zjistil, že jakákoliv jiná hodnota rozdílná od nuly způsobí buď pokles celkové energie nebo úplné zhroucení systému. Volba *ERP* není už tak důležitá, nicméně je důležité její hodnotu držet nad nulou. Hodnoty kolem 0.4 se jeví jako rozumná volba.

Výsledek obdobného testu v *Havoku* [3] nedopadl, ve srovnání s *ODE*, vůbec dobře. Celková energie kyvadla se po necelé minutě snížila až na polovinu a poté se na této hodnotě ustálila. *ODE* se při svém pravidelném kolísání energie dopouští relativní chyby[6] přibližně 0.75%.

5.4 Test č.3 - Domino

Tento test je ukázkou použití tření při kontaktu dvou objektů. Scéna je poskládána z několika destiček postavených za sebou ve spirále, tak aby při pádu jedné desky způsobil pád následující desky. Nakonec je docíleno známého řetězového efektu předávání kinetické energie. Podobného efektu je docíleno i v testu 5 (kapitola 5.6 na straně 20), jen scéna je postavena výrazně jinak.

Pro postavení takové scény stačí jen správně rozmístit desky, jak již bylo zmíněno, a nastavit vhodný koeficient smykového tření. Tření v *ODE* aproximuje tzv. *Coulombův model tření*, který je jednoduchý a efektivní k modelování tření v kontaktních bodech:

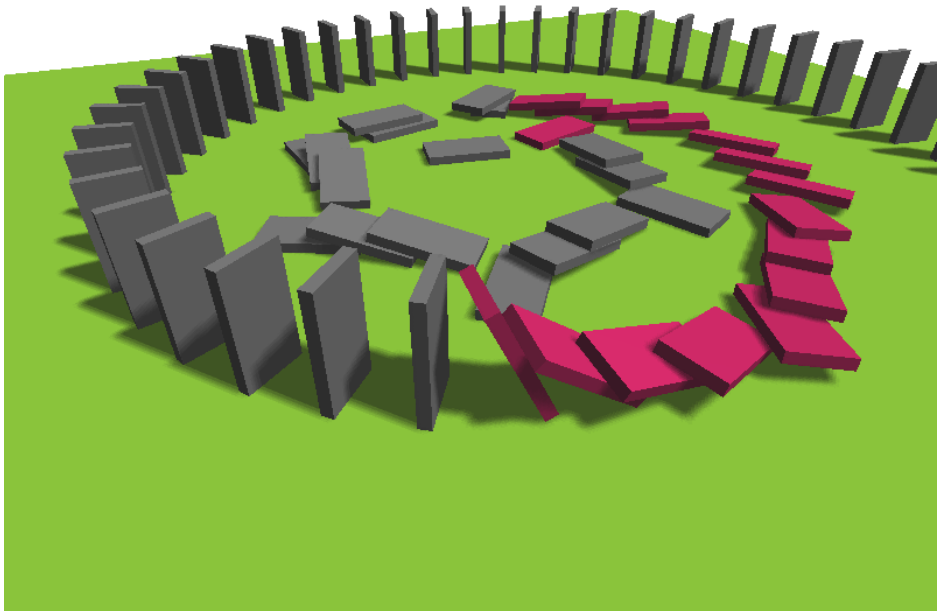
$$|F_t| \leq \mu |F_n| \tag{1}$$

kde μ je součinitel smykového tření, F_t je síla tečná k ploše, a F_n je síla kolmá na stykovou plochu. Běžně se hodnota μ nastavuje na hodnotu 1.0, přesto může být občas výhodné použít speciální hodnotu `dInfinity` označující nekonečno. Hodnota tření se nastavuje pro každý kontakt zvlášť v metodě `nearCallback`:

```
contact[i].surface.mu = 1.0;     // dInfinity
```

Zde se nastavuje proměnná `mu` představující výše zmiňovaný koeficient tření μ. Poznamenám, že volání konstrukce `contact[i]` v cyklu nastavuje parametry materiálu pro každý potenciální kontakt mezi dvěma objekty. Více

[6]Energie kyvadla dosahovala hodnoty v minimu 29.250 a v maximu 29.471. Rozdílem těchto hodnot získáme absolutní změnu energie 0.22112. Relativní změna (0.75%) se rovná hodnotě absolutní změny energie vydělené maximální hodnotě energie.



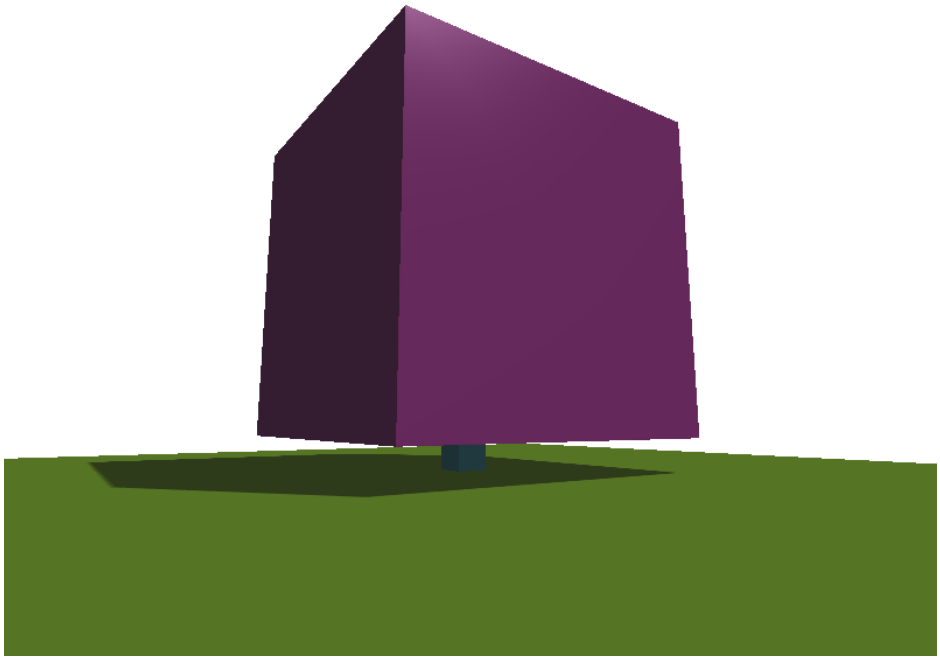
Obrázek 5: Ukázka scény z testu *Domino*

detailů najdete v na začátku kapitoly 5 na straně 6. Program domino umožňuje měnit hodnoty μ klávesami **F** pro snížení nebo **G** pro zvýšení. Klávesa **R** obnoví scénu do výchozí podoby.

Možnou komplikací tohoto testu je výpočetní náročnost simulace.[7] Pokud je například vypnuta automatická deaktivace objektů, simulace je velmi pomalá, přestože na začátku dochází jen k malému počtu kolizí. Z toho plyne, že rychlost simulace závisí zejména na počtu aktivních objektů, naproti tomu počet kolizí se zřejmě tak výrazně na výkonu neprojeví. Se zapnutou automatickou deaktivací objektů se situace výrazně zlepší, nelze však mluvit optimálním stavu. Vizualizace aktivních (označené normální barvou) a neaktivních (označené šedou barvou) objektů odhalí, že některé desky zůstanou po pádu aktivní, i když se tyto objekty vůbec nehýbou. Řešení se naskýtá v podobě dalších funkcí, které nastavují parametry okolností, za kterých mají být objekty deaktivovány. Nastavíme vyšší práh rychlosti, popř. úhlové rychlosti, od kterého považujeme objekt za nehybný:

```
dWorldSetAutoDisableLinearThreshold(world, 0.1);
dWorldSetAutoDisableAngularThreshold(world, 0.1);
```

[7] Rychlost simulace je také předmětem kapitoly 5.6 na straně 20.



Obrázek 6: Ukázka scény z testu *Different Weights*

Takto zablokované objekty se sice nehýbou (nepůsobí na ně třeba gravitace), ale nijak nejsou omezeny v reakcích na jiné objekty. Pokud dojde ke kolizi s jedním neaktivním objektem, zmíněný objekt je znovu aktivován a v následujícím simulačním kroku se normálně účastní výpočtu fyziky.

Další efektivní urychlení celé simulace umožňuje následující příkaz:

```
dWorldSetQuickStepNumIterations(world, 8);
```

Snížením počtu iterací metody *QuickStep* z 20 iterací na 8 se počet snímků za sekundu zvýšil přibližně o 20%. Další urychlení se mi podařilo dosáhnout zvýšením simulačního kroku na hodnotu 0.005. Těmito úpravami je teď test bez problémů spustitelný v reálném čase i na slabším stroji bez viditelných projevů nestability.

5.5 Test č.4 - Rozdílné hmotnosti

Tento test má za úkol vyzkoušet, jak se chovají dvě krychle různé hmotnosti a velikosti, pokud ta větší a těžší krychle je položena na tu menší a lehčí (viz obrázek 6 na straně 19).

Máte tedy ve scéně opět základní nehybnou desku a dva zmiňované objekty. Ten menší má rozměry $[0.5, 0.5, 0.5]$ a hmotnost 5.0 (kg) a větší krychle

bude vytvořena o rozměrech $[5.0, 5.0, 5.0]$ a hmotnost 5000.0 (kg). Dále je třeba vypnout automatickou deaktivaci objektů protože by to mohlo skrýt možné problémy stability.

Výsledek testu nedopadl dobře. Velká krychle poskakuje na malé. Malá krychle také neustále kmitá. Po určité době velká krychle neudrží rovnováhu a osový systém se zhroutí, poté ještě malá krychle je s kmitavým pohybem vytlačována ven a samozřejmě nakonec po uplynutí několika chvil se tak stane.

Problém je vyřešen výrazným snížením simulačního kroku na hodnotu 0.0001 (desetina původní hodnoty). Nyní je systém stabilní (osový systém je dodržen) až do hmotnosti přibližně 15000.0 (15 tun), kdy i poskakování je minimální. Pokud akceptujeme poskakování krychlí, můžeme mluvit o stabilitě systému i s krychlí o hmotnosti 75000.0 (75 tun).

Jediná věc, která se musí hlídat nezávisle na simulačním kroku, je, aby hodnota globálního *CFM* se pohybovala od 10^{-6} a méně, protože při vyšších hodnotách se objekty stávají měkkými, a to má za následek propadnutí obou krychlí pod pevnou podložku. K tomuto efektu dojte také při *ERP* menší než 0.1, proto doporučuji nechat *ERP* na 0.2.

Tak jak uvádí autor bakalářské práce [3], *Havok* 6.5 byl stabilní do hmotnosti 4000.0 (kg). Projevy nestability jsou pro oba enginy podobné. Přesto *ODE* dokáže s vhodným nastavením simulačního kroku provádět simulace mnohem přesnější než *Havok*, za cenu větší výpočetní náročnosti.

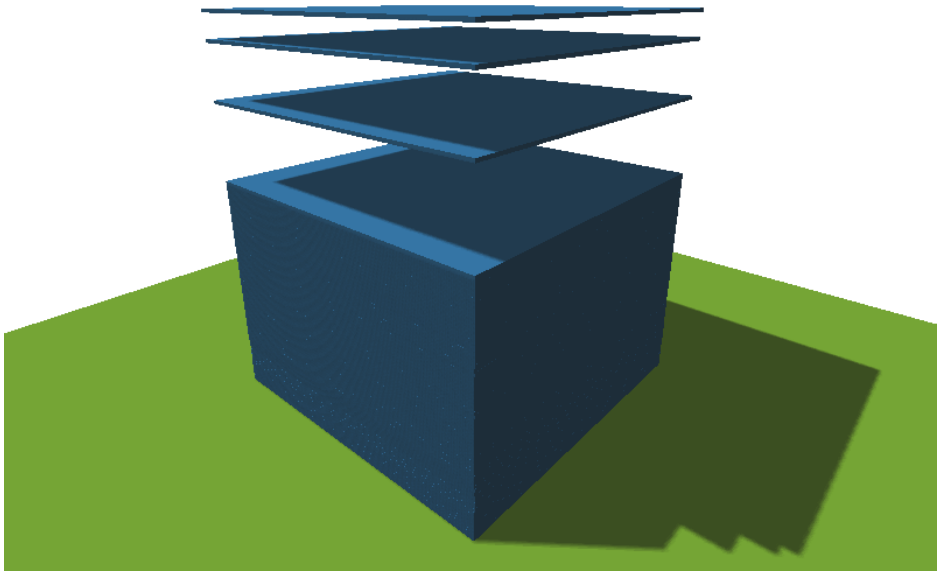
5.6 Test č.5 - Mnoho stejných kolizí

Tento příklad má za úkol ověřit, jak si *ODE* dokáže poradit s velkým počtem objektů vytvářejících mnoho stejných kolizí.

Scéna bude opět obsahovat pevnou podložku. Pomocí klávesy **N** se vytváří ploché desky o rozměrech přibližně $[5.0, 0.05, 5.0]$ ve výšce 6.0 nad podložkou. Vytvořené desky se postupně skládají na sebe. Globální *ERP* je nastaveno na základní hodnotu 0.2 a globální *CFM* jsem úmyslně nastavil na absolutní nulu. Kontakty při kolizích jsou také na základním nastavení, tj. není použita žádná z vlastností `dContactBounce`, `dContactSoftERP` ani `dContactSoftCFM`. Uvedené vlastnosti aktivujeme jako bitové příznaky v proměnné `mode`[8].

Při spuštění programu zjistíme, že se desky na sebe skládají podle očekávání, bez nějakých vedlejších projevů, jako je poskakování objektů nebo jiný vedlejší pohyb. Test se tedy do jisté míry chová velmi stabilně. Problémy se

[8]Implementace metody `nearCallback` je blíže popsána na začátku kapitoly 5 na straně 6.



Obrázek 7: Ukázka scény z testu *Lots of same objects*

neočekávaně dostaví ve chvíli, kdy počet naskládaných desek přeroste určitou mez, která je závislá na výkonu procesoru. Program se po určité době za neustálého vytváření desek výrazně zpomalí. V krajním případě program úplně zamrzne, protože je až příliš zaměstnán výpočty simulace pro takové množství objektů na relativně malém prostoru. Nároky na výkon rostou spolu s počtem vytvořených desek. Problém s náročností simulace se zdá býti neřešitelný. Výrazné zpomalení aplikace lze oddálit nastavením vyššího prahu rychlosti pro automatickou deaktivaci.

```
dWorldSetAutoDisableFlag(world, 1);
dWorldSetAutoDisableLinearThreshold(world, 0.3);
```

Výsledkem je částečně kratší doba působení horních desek na nižší. Dále je možné, díky automatické deaktivaci objektů, opětovné rozběhnutí programu, nicméně vytvoření další desky opět přivede program do dalšího a pravděpodobně delšího zamrznutí než předtím. Problém rychlosti tím tedy zdaleka nevyřešíme. Navíc lze někdy vypozorovat i další problém způsobený vyšším prahem rychlosti. Některé desky mohou být zablokovány těsně předtím než dokonale dosednou na desky spodní nebo jednoduše zůstanou viset volně v prostoru. Deaktivaci objektů je lepší raději úplně vypnout.

Rychlost simulace se může řešit pomocí algoritmů dělících prostor. Samozřejmě *ODE* má tyto prvky implementovány:

```
dSpaceID dHashSpaceCreate(dSpaceID space);
dSpaceID dQuadTreeSpaceCreate(dSpaceID space,
    dVector3 Center, dVector3 Extents, int Depth);
```

Použitím *dHashSpaceCreate* se rychlost testování výrazně zlepšila. Nyní je možné postavit dvakrát tak větší sloupec než předtím. Další urychlení na úkor stability systému jsem provedl nastavením nižšího počtu iterací metody (nastavíme klávesou **F6**) *QuickStep* z hodnoty 20 na hodnotu 10.

Zvýšení přesnosti dosáhneme nižším simulačním krokem (klávesa **F8**, popř. se stisknutým modifikátorem **Shift**), naopak zvýšení rychlosti zase dosáhneme snížením kroku. Ale pozor, nižším krokem se výrazně zvyšuje výpočetní náročnost simulace, která je už se základním krokem dost pomalá.

Hlavní problém se projeví ve chvíli, kdy nastavíme *CFM* větší než nula (např. 0.0001), sloupec s větší výškou se začne kroutit až se nakonec *zhroutí* celý systém. Jakou hodnotu zvolíme, tím jen určujeme, kdy tento sloupec spadne. Jediná správně fungující hodnota *CFM* je nula. Ostatní parametry jako simulační krok a počet iterací můžou pád sloupce pouze oddálit, ale ne úplně odstranit.

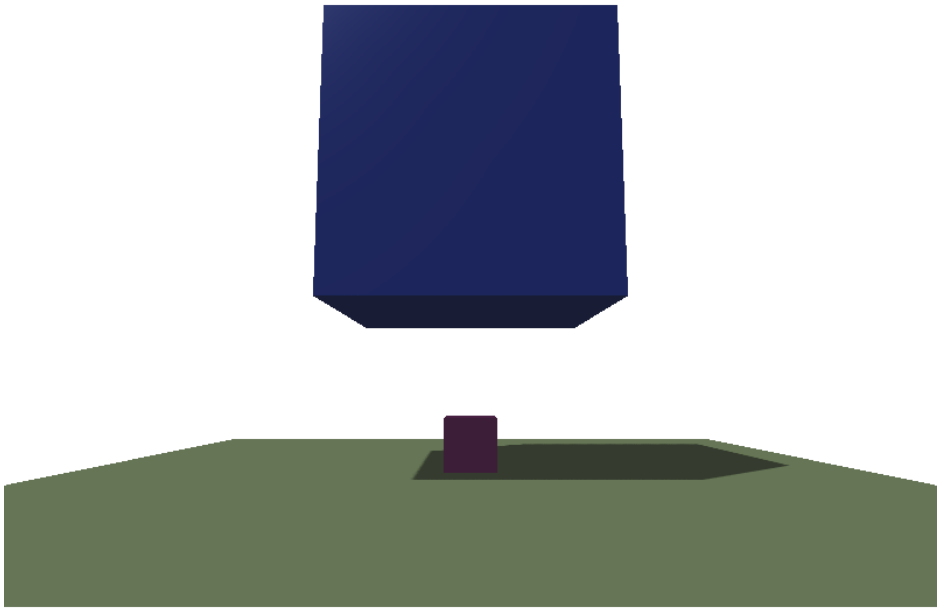
Fyzikální engine *Havok* (viz práce [3]) se v nestabilních případech choval velmi odlišně od chování sloupce v *ODE*. V *Havoku* docházelo k trhavému pohybu, dokonce některé desky byly vytlačeny ze sloupce. *ODE* se také projevuje trhavým pohybem desek, nicméně žádná deska není vytlačována a po celou dobu udržení sloupce jsou desky správně zarovnané. Jen občas některá deska je o nějaký ten milimetr posunutá mimo přesnou osu, jenže to není problém vytlačování, ale problém nepřesného dopadu.

Výsledky *Havoku* v upravené verzi jsou už v pořádku stejně jako v *ODE*. Co činí *Havok* lepší v tomto testu, je výraznější rychlost výpočtu simulace, než jak to dokázalo *ODE*.

5.7 Test č.6 - Osový systém

Nyní bude testována stabilita osového systému dvou objektů vzájemně propojených kulovým kloubem.

Ve scéně je pevná základna `Plane` na které je volně položená malá krychle o délce jedné strany 1.0 a hmotnosti jedné jednotky 1.0. Nad malou krychlí je umístěna velká krychle s délkou strany 5.0 a hmotností 5000.0. Obě krychle jsou spojeny kloubem typu *ball joint* (obrázek 9 na straně 24), který má kotevní bod *anchor* umístěn ve výšce 3.0 nad základnou. Tento kotevní bod současně leží ve spodní straně horní krychle. Očekávané chování je klidný stav



Obrázek 8: Ukázka scény z testu *Axis system*

systému, tak jak je scéna postavena. Klávesa **R** umožňuje vrátit objekty do výchozí polohy.

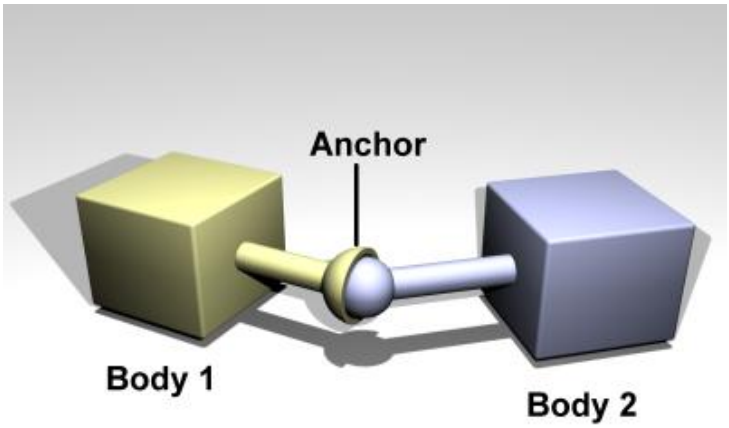
Tento test nedopadl příliš úspěšně. Výsledkem testu je zpočátku očekávaný klidný stav, nicméně po několika vteřinách horní krychle ztratí z nějakého důvodu rovnováhu a osový systém se zhroutí. Vazba mezi malou a velkou krychlí je podle viditelného výsledku bez komplikací zachována. Z vizuální stránky se simulace chová přirozeně, bez viditelného poskakování a jiného neočekávaného trhavého pohybu.

Změnou *ERP* nebo *CFM* jsem nedosáhl viditelného zlepšení. Naopak zvýšení *CFM* navíc způsobuje propadnutí obou spojených objektů pod podlahu. Dále jsem se pokusil utlumit pootočení horní krychle pomocí následující funkce:

```
dWorldSetMaxAngularSpeed(world, 0.01);
```

Scéna se samozřejmě ustálila, protože omezením úhlové rychlosti na nulu se objekt nemůže vůbec natáčet. Problém držení stabilní osy nebyl omezením úhlové rychlosti vyřešen. I když se osový systém nehýbe, stále máme co do činění se stejným problémem. Uvedený návrh tedy není uznatelný jako řešení.

Fyzikální engine *Havok* [3] si vede v tomto testu lépe, protože horní



Obrázek 9: Ball-Socket joint (převzato z *ODE* Community Wiki)

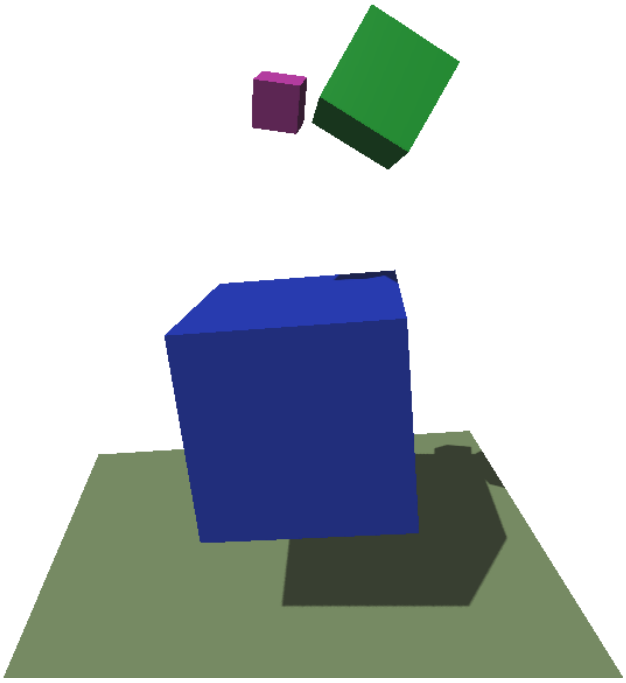
krychle udrží stabilitu osového systému. Test v enginu *ODE* je bohužel neúspěšný i přes vizuálně stabilní systém.

5.8 Test č.7 - Osový systém 2

Další test se opět zaměřuje na osový systém, tentokráte s objekty volně zavěšenými na pevném bodě. Objekty mají také rozdílné hmotnosti tak, aby byla vytvořena dostatečná zátěž na klouby (*joints*). Do nehybného závěsu je vnesen i dynamický prvek v podobě, který by měl adekvátně prověřit stabilitu celého systému.

Scéna je složena ze dvou krychlí a dvou kloubových vazeb. Ve výšce 10.0 jednotek v ose y je umístěn kloub *ball joint* spojený se statickým prostředím. Dále je ve scéně menší krychle o hmotnosti 5.0 jednotek, zavěšená pomocí zmiňovaného kloubu (stejné zavěšení jako u kyvadla, viz kapitola 5.3 na straně 14). Ve spodní straně malé krychle je další kloub tvořící spojení s velkou krychlí o hmotnosti 5000.0 jednotek. Do malé krychle je vhozena další krychle (testovací krychle) o trochu větší hmotnosti 100.0 s rychlostí 30.0 metrů za sekundu. Gravitace, *ERP* a *CFM* jsou nastaveny na standardní hodnoty jako v předchozích příkladech. Pomocí klávesy **R** můžete obnovit pozici a rychlost testovací krychle, čímž donutíme krychli znovu narazit na osový systém (krychle narazí do pivotu).

Výše popsaný systém se chová velmi přirozeně. Po nárazu testovací kostky do malé kostky (pivot) se soustava rozkývá. Čím více provádíme hodů testo-



Obrázek 10: Ukázka scény z testu *Axis system 2*

vací kostkou, soustava se kývá čím dál rychleji, což znovu potvrzuje platnost zákonu zachování energie, jako tomu bylo v testu 2 s kyvadlem (kapitola 5.3 na straně 14). Poměrné nepříjemný je efekt vysoké úhlové rychlosti pivotu, v závislosti na tom jak a kolikrát se pivot dostane do kolize s testovací krychlí. Vysoké úhlové rychlosti pivot dosáhne, když se obě krychle srazí v nevhodném natočení vůči sobě, přičemž se většina kinetické energie přenese na pivot v podobě úhlové rychlosti a ne v pohybové rychlosti, jak to nastává u jiných tvarů objektů jako je například koule. Vysoká úhlová rychlost lze snadno tlumit voláním funkce:

```
dWorldSetMaxAngularSpeed(world, 10.0);
```

kdy dosáhneme uspokojivého výsledku. Je třeba jen upozornit, že zmíněné omezení úhlové rychlosti má vliv na všechna tělesa ve scéně. Je dobré zvážit jakou vhodnou hodnotu zvolit, aby neutrpěly na realističnosti jiné objekty ve scéně, které naopak by naopak měly mít jinou úroveň omezení.

Havok je v tomto testu úspěšný po provedení některých úprav, jak uvádí autor v bakalářské práci [3]. Potěšujícím výsledkem je fakt, že v *ODE* nebylo potřeba provádět nějaké zvláštní úpravy na odstranění problémů. *ODE* je v tomto testu přirozeně stabilní. Nedá se ale předejít stavům, kdy by uživatel

změnil hodnoty parametrů *ERP* a *CFM* nevhodným způsobem, potom se nelze vyhnout problémům nestability. Ostatně to platí i pro všechny ostatní testy. Parametry *ERP* a *CFM* jsou proto v enginu *ODE* nejdůležitějšími prvky pro udržení stability scény nebo jejího zhroucení. Je třeba zacházet s těmito parametry nadmíru obezřetně.

5.9 Test č.8 - Stabilita závěsu

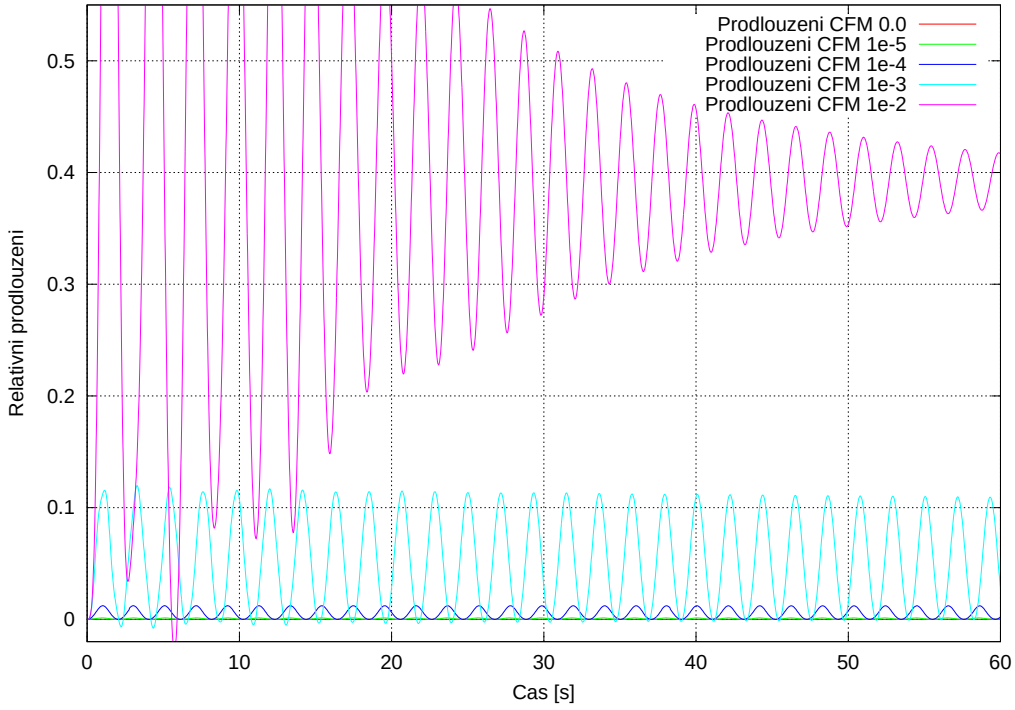
Tato kapitola se věnuje tomu, jak *ODE* dodržuje délku závěsu s velmi těžkým objektem.

Scéna je složena stejně jako v testu 2 (kapitola 5.3 na straně 14) až na jednu výjimku. Koule kyvadla je značně hmotnější než u původního příkladu (5000.0 jednotek hmotnosti). Počáteční stav kyvadla také odpovídá 90° výchylky od svislé osy. Před spuštěním simulace spočteme délku kyvadla, kterou vypočteme jako vzdálenost od středu otáčení (joint anchor) do pozice (těžiště) kyvadla. Během simulace budeme sledovat změnu délky a očekáváme, že se kyvadlo neprodlouží.

V určitých časových intervalech se zjištěná prodloužení zapisují do souboru `Test8_output.txt`. Při každém spuštění aplikace je soubor přepsán novými daty, takže si uživatel musí sám provádět zálohu zmíněného souboru jak uzná za vhodné. Stiskem klávesy **R** se kyvadlo vrátí do výchozí polohy a výstupní soubor je přepsán prázdným souborem. Klávesy **F2**, **F3** a **F4** slouží jako obvykle k nastavení *ERP* a *CFM*.

Ukázalo se, že tak, jak byla kvalita simulace ovlivňována v testu 2 v kapitole 5.3 na straně 14 konstantami *ERP* a *CFM*, budeme pozorovat obdobnou odezvu i v prodloužení. Sledováním hodnot relativního prodloužení jsem zjistil, že kyvadlo se periodicky prodlužuje a zkracuje. Obrázek 11 na straně 27 zobrazuje příklady naměřených hodnot s různým nastavením parametru *CFM*. Kyvadlo se nejvíce prodlužuje v okolí rovnovážného stavu, zatímco v místech s výchylkou $\pi/2$ je prodloužení minimální nebo dokonce nulové, pokud se systém chová stabilně. Nejedná se ale o velký problém, protože i při nastavení $ERP = 0.4$ a $CFM = 10^{-4}$ nabývá maximální chyba hodnot v řádech 10^{-3}. Nastavení CFM rovno 10^{-4} a menší můžeme považovat za bezpečný rozsah, kdy bude pro nás přesnost simulace přijatelná. Pokud budeme CFM zvyšovat přes zmíněnou doporučenou hranici, například pro volbu 10^{-3}, je možné pozorovat prodloužení přímo ve vizualizaci. Další zvyšování CFM až do řádu 10^{-2} vede ke zhroucení systému, kyvadlo propadne skrz základní rovinu, poté se vrátí zpět, ustálí se okolo nové hodnoty prodloužení a současně se přestane kývat. Tuto skutečnost vystihuje obrázek 11.

Předchozí odstavec dává pouze doporučení hodnot *CFM*, jenže i parametr



Obrázek 11: Relativní prodloužení kyvadla při $ERP = 0.4$ a CFM rovno 0, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}

ERP má na chování simulace nezanedbatelný vliv. Je pravda, že není nutné pracovat s *ERP* tak opatrně jako s *CFM*, nemůžeme ale používat hodnoty blízké nule, kde se výše popsané problémy mohou ještě znásobit. V tomto testu doporučuji *ERP* nastavit na hodnotu 0.4. Zvýšením ERP až na hodnotu 1.0 můžeme chybu snížit až na polovinu oproti nastavení *ERP* okolo 0.4.

Při ideálním nastavení ($ERP = 1.0$ a $CFM = 0.0$) lze dosáhnout velmi přesného výsledku s chybou řádově 10^{-6}. Pokud bychom použitým jednotkám přiřadili význam v metrech, lze v nejlepším případě dosáhnout chyby (relativní prodloužení) okolo 1 až $2\mu m$.

Na závěr této kapitoly bych ještě poukázal na jednu skutečnost. Hmotnost kyvadla nemá na prodloužení žádný nebo spíše nepozorovatelný vliv. Zkoušel jsem hmotnosti od 5.0 kg, přes 5000.0 kg až na velmi vysokou hmotnost 1000 tun a ve všech případech systém vykazoval stejné prodloužení.

5.10 Test č.9 - Přesnost restituce

Tento test prověřuje přesnost odrazu koule od vodorovné plochy. Cílem tedy je ověřit, zda jde *ODE* simulovat dokonalý odraz.

Scéna obsahuje kouli umístěnou ve výšce 5.0 nad základnou, která bude vlivem gravitace poskakovat na základně. Koeficient odrazivosti *bounce* (restituční koeficient) nastavíme na hodnotu 1.0 v metodě `nearCallback`. Koeficient tření není v tomto testu podstatný, ale může být nastaven na nekonečno (`dInfinity`). Gravitace je čtyřikrát vyšší než základní zemská přitažlivost, aby bylo možné rychleji ověřit stabilitu po větším množství odrazů.

```
dWorldSetGravity(world, 0, -4.0 * SURFACE_GRAVITY, 0);
```

Výsledkem scény musí být periodický pohyb koule. Během simulace budu sledovat výšku a budu zaznamenávat tyto hodnoty v místech lokálního maxima. Odchylka lokálního maxima od počátečního stavu by měla být nulová. Chyba je v příkazové řádce reprezentována třírozměrným vektorem s osami *xyz*, aby bylo možné zachytit i případně odchylky ve vodorovné rovině. Klávesou **R** lze kouli vrátit do výchozí polohy a zopakovat měření od začátku. Dále nastavení *ERP* a *CFM* se provádí pomocí již zmíněných kláves **F2** (nastavení kroku), **F3** (nastavení *ERP*) a **F4** (nastavení *CFM*).

Tento test dopadl úspěšně, protože koule skutečně poskakuje pravidelně, bez viditelné ztráty výšky. Výstupy v příkazové řádce ukazují, že koule generuje v případě ideálního nastavení $ERP = 0.2$ a $CFM = 0.0$ nulovou odchylku v bodě lokálního maxima. Výsledek tohoto testu až nadmíru uspokojivý. Přesnost obrazu není nijak ovlivňována hodnotou *ERP*. Zkoušel jsem všechny použitelné hodnoty z rozsahu 0.0 až 1.0 a podle zobrazení chyby

odrazu v čísle s plovoucí řádovou čárkou na šest desetinných míst hodnota zůstala na nule. Dále jsem nezaznamenal žádné vychýlení pohybu s osy y, tj. souřadnice x a z zůstaly od výchozího stavu nezměněny.

Zásadní vliv na přesnost odrazu má jednoznačně parametr *CFM*. Zvýšením *CFM*, byť jen nepatrně nad nulu, je odraz koule výrazně utlumen. Například když nastavíme *CFM* na 10^{-6} už po pěti odrazech se chyba odrazu pohybuje okolo 1%, po 10 odrazech přibližně 3% a po 20 odrazech chyba překročí 5%. Po shlédnutí těchto výsledků vidíme, že skutečně dokonalého odrazu zadané koule lze dosáhnou jen tehdy, kdy se *CFM* rovná přesně nule nebo zvolíme-li velmi malou hodnotu blízko nuly, pokud nějaký ten útlum potřebujeme.

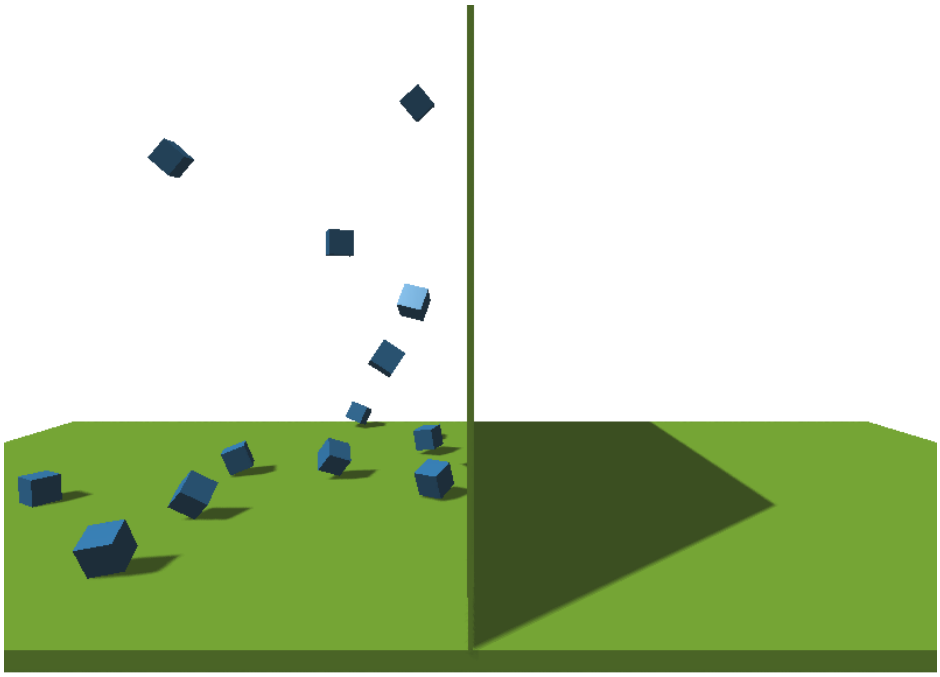
Je dobré upozornit, že pracujeme se simulačním krokem 0.001, který má zásadní vliv na to, jaký mají efekt použité hodnoty *ERP* a *CFM*. Kdybychom změnili hodnotu simulačního kroku, všechna zmíněná čísla týkající se *ERP* a *CFM* by musela být odpovídajícím způsobem upravena. Této problematice se podrobně věnuje i uživatelský manuál [5] v kapitole *Concepts*. Další článek zabývající se stejnou problematikou najdete v knize [4] v kapitole *3.4: Constraints in Rigid Body Dynamics*.

Pokud srovnáme výsledky tohoto testu ve fyzikálním enginu *ODE* s výsledky enginu *Havok*, můžeme směle tvrdit, že *ODE* se v tomto směru chová o poznání stabilněji. Jak bylo uvedeno v bakalářské práci [3], *Havok* nedokázal dosáhnout absolutně přesného odrazu i přes všechny úpravy, které autor ve své práci uvádí. Po konečných úpravách size*Havok* neztrácel amplitudu, ale naopak docházelo k mírnému zvýšení amplitudy.

5.11 Test č.10 - Maximální rychlost

Tímto posledním testem budeme zkoumat přesnost testování kolizí malých objektů s tenkou deskou ve vysokých rychlostech, přičemž cílem bude stanovení maximální rychlosti objektu, při které detekce kolize ještě dosahuje 100% úspěšnosti za zvolených podmínek.

Ve scéně je umístěna svislá deska o relativně malé tloušťce (0.05) orientovaná v ose *YZ*, která bude odrážet letící objekty. Z pozice $[-4.0, 3.0, 0.0]$ jsou vypouštěny krychle o hmotnosti 1.0 a délce jedné strany 0.4 směrem k desce. Každá krychle má počáteční rychlost dle volby uživatele. V nulové výšce je navíc umístěna vodorovná podlaha, aby bylo snadnější identifikovat, které objekty se odrazily a které prošly skrz zeď. Použitím klávesy **B** vytvoříme novou krychli letící směrem ke zdi s aktuálně nastavenou rychlostí (můžeme měnit klávesou **F5**). Opět můžete v testu libovolně nastavovat *ERP* a *CFM* klávesami **F3** a **F4**. Počet iterací se nastavuje klávesou **F6** a relaxační parametr *SOR* zase **F7**. Poslední klávesou, jak bude uvedeno později,



Obrázek 12: Ukázka scény z testu *Maximum velocity*

nejdůležitější ze všech je **F8**, která ovlivňuje velikost simulačního kroku ve funkci `dWorldQuickStep`.

Na začátek je nutné upravit některé parametry, které zejména omezují průnik krychlí do zdi:

```
dWorldSetContactSurfaceLayer(world, 0.0);
dWorldSetERP(world, 0.3);
dWorldSetCFM(world, 0.0);
```

Zkoušení jiných hodnot vedlo ke snížení hranice maximální rychlosti. Podstatné parametry jsou hlavně *CFM* a hloubka kontaktní vrstvy. Po aplikaci jmenovaných nastavení bylo možné dosáhnout až $250\,\mathrm{ms}^{-1}$ s nastaveným simulačním krokem 0.001. Další nastavitelné parametry simulace už nepomohly tento výsledek vylepšit s výjimkou jednoho, ale o tom až v dalším odstavci. Hodnota *ERP* nemá žádný vliv na přesnost, pouze může ovlivnit rychlost případného odrazu koule od zdi. Zato *CFM* může výsledky ještě zhoršit. Jak bylo již dříve zmíněno, *CFM* zvyšuje měkkost objektů a dovoluje propadnutí (penetraci) objektů do sebe. Tato vlastnost způsobuje, že objekt sice bude kolidovat se zdí, nicméně s vyšším nastavením *CFM* kontakt dovolí větší průnik místo přímého odražení objektů. Jednoduše řečeno, koule začnou propadat i v několikanásobně menších rychlostech.

Poslední parametry, o kterých jsem si myslel, že zvýší přesnost kolize, jsou počty iterací metody *QuickStep* (pro jeden simulační krok) a Relaxační parametr *SOR*:

```
dWorldSetQuickStepNumIterations(world, 20);
dWorldSetQuickStepW(world, 1.3);
```

Předchozí příklad ukazuje výchozí nastavení jmenovaných parametrů. Zkoušením různých hodnot se ukázalo, že uvedené parametry nemají žádný vliv na přesnost detekce kolizí. Počet iterací může ovlivnit nárok na výkon a přesnost odrazu už detekované kolize nebo se mohou objevit jiné projevy nestability, které za prvé nebudou tak patrné a za druhé nejsou předmětem tohoto testu. Samozřejmě že při tvorbě v praxi použitelného softwarového řešení budou hrát roli všechny aspekty stability a rychlosti výpočtu.

Naštěstí existuje jeden parametr, který jsem do teď přehlížel a nechával jej na výchozí hodnotě, protože bylo spíše nežádoucí provádět nějaké změny. Jedná se o simulační krok metody *QuickStep*. Pokud zmenším simulační krok, maximální rychlost se zvyšuje. Simulační krok lze snižovat, dokud nám stačí výpočetní výkon nebo nastanou problémy s reprezentací malých čísel v pohyblivé řádové čárce, a proto nelze exaktně stanovit konkrétní hodnotu maximální rychlosti. Velikost simulačního kroku, kterou nakonec zvolíme, je většinou dána výkonem počítače. Důkazem závislosti kroku na maximální rychlosti může být následující fakt: když je nastaven simulační krok na 0.001 dostaneme se na maximální rychlost $250\,\mathrm{ms}^{-1}$ a při kroku 0.0001 (desetina základní hodnoty) se dostaneme maximální rychlost pohybující se okolo $3630.0\,\mathrm{ms}^{-1}$. Jedná se velmi potěšující výsledek, protože zmenšením kroku na desetinu výchozí hodnoty, jsme získali přibližně desetinásobek maximální rychlosti.

Výsledek tohoto testu je tedy velmi překvapivý. V porovnání s výsledky dosaženými pomocí fyzikálního enginu *Havok* [3] jde s *ODE* dosáhnout mnohem lepších výsledků, kde jediným omezením hranice maximální rychlosti je rychlost počítače. V enginu *Havok* bylo dosaženo až $350\,ms^{-1}$ rychlosti při 100% spolehlivosti detekce kolizí.

Engine *ODE* se ve srovnání s *Havokem* choval jinak ještě v jednom smyslu. *Havok* má hranici maximální rychlosti ne tak přesně určenou jako *ODE*, protože při přiblížení k pomyslné hranici maximální rychlosti *Havok* postupně zvyšuje podíl krychlí, které prochází naskrz. Proto je hodnota maximální rychlosti zvolena tak, kdy s velkou pravděpodobností nedochází k žádnému propadnutí. U *ODE* je hranice rozpoznatelná snadno. Buď neprochází jediná krychle nebo naopak po překročení hranice maximální rychlosti prochází skrz zeď úplně všechny kostky.

6 Výsledky a zhodnocení

Celkové chování simulací v *ODE* je ve většině případů potěšující. Například test 6 (na straně 22) se mi nepodařilo zajistit chování předepsané zadáním. Osový systém se pravda choval velmi reálně, ale velká krychle půl minutě neudržela přesnou rovnováhu v ose a systém se zhroutil.

Na druhou stranu můžu jmenovat řadu testů, které se chovaly o poznání lépe než testy napsané pomocí fyzikálního enginu *Havok*. Už v testu č. 1 *ODE* prokázalo, že dokáže přesně interpretovat různé koeficienty odrazivosti. Velmi dobrou přesnost výpočtu odrazu *ODE* prokázalo i v testu č. 9. *Havok* měl v tomto směru problémy.

Test č. 4 ukázal velký potenciál pro přesnost simulace, pokud nastavujeme velikost simulačního kroku. *ODE* dokázalo udržet krychli o dvakrát větší hmotnosti, než kterou zvládl *Havok*. Nastavení simulačního kroku zásadně ovlivnilo výsledky i v posledním testu 10. Zde byl rozdíl mezi *ODE* mnohem markantnější. *Havok* i po všech vylepšeních dosáhl maximální rychlosti $350\,ms^{-1}$. S *ODE* jsem nedokázal přesně určit maximální rychlost, protože simulační krok jde zmenšovat téměř neustále, záleží na výkonu stroje. Každopádně nebylo problém dosáhnout až rychlosti kolem $3600\,ms^{-1}$.

Test 2 dokázal, že *ODE* nijak nesimuluje tlumení ve vazbách, proto nebyl problém simulovat model téměř ideálního fyzikálního kyvadla. Energie kyvadla v *ODE* se vůbec neměnila. Docházelo jen k mírnému periodickému poklesu 0.75% celkové energie. *Havok* v testu neuspěl, protože ztratil až 50% celkové energie. Nabyl jsem přesvědčení, že v *ODE* je periodické kolísání energie nějak spojeno s prodlužováním kyvadla podle testu 8.

Lepších výsledků dosahoval *Havok* v testu 5. U obou enginů se podařilo vytvořit dostatečně přesnou simulaci. *ODE* však zaostává v otázce rychlosti, protože v momentech kdy simulace v *Havoku* běžela plynule, rychlost v *ODE* na byla pod 60 snímků za sekundu (nejednalo se o simulaci v reálném čase).

Pokud budu posuzovat *ODE* z jiného úhlu, měl bych zdůraznit poměrnou jednoduchost *API* (*Application Programming Interface*). Samozřejmě je pravdou, že objektové rozhraní by nebylo při psaní v *C++* na škodu, nicméně procedurální přístup je obecně snadnější princip programování. Dále bylo v *ODE* snadné kdykoliv během běžící simulace změnit pozici, rotaci a rychlost libovolného objektu. Naproti tomu v *Havoku* jsou takové operace nemožné, protože se celá scéna musí smazat a znovu vytvářet.

Poslední nevýhodou při používání *ODE*, je poměrný nezájem o vývoj i používání této kvalitní knihovny. Vývoj *ODE* je v současné době pomalejší, než tomu bylo před několika lety. Oblíbenost klesla zejména proto, že v dnešním rychle rozvíjejícím herním průmyslu je těžké se prosadit proti velkým komerčním konkurentům jako jsou *Havok* a *PhysX*, které mají nejen pod-

poru pro simulaci dynamiky tuhých těles, ale další užitečné prvky. Dále *Havok* a *PhysX* začínají využívat pro výpočty specializovaný hardware (např. *GPU*), který výrazně urychluje výpočet. Vývojáři *ODE* zatím dlouhodobě nepočítají s úpravami knihovny pro jakoukoli hardwarovou podporou.

Všechny příklady byly testovány grafických čipech *Radeon HD 4850*, *Nvidia GTS 250* a *Intel 945GME*. Testy proběhly jak na klasickém 32-bit operačním systému, tak na 64-bit OS. Výsledky testů uvedené v této práci byly získány na operačním systému *GNU/Linux* s verzí jádra *2.6.32-30-generic*.

7 Uživatelská příručka

Pro všechny výše popsané testy platí společné ovládání pohledu. Směr kamery (pohledu) je ovládán myší, přičemž je nutné napřed tento režim řízení kamery zapnout kliknutím levého tlačítka myši (současně zmizí kurzor). Opakované klepání na tlačítko myši zapíná nebo vypíná ovládání kamery. Klávesami **W**, **S**, **A** a **D** lze pohybovat kamerou dopředu, dozadu, vlevo a vpravo v závislosti na směru kamery. Dále je možné každý program vypnout stiskem klávesy **Esc**.

Každý test bývá ovládán další skupinou kláves, které závisí na požadavcích konkrétní scény. Například klávesa **R** obnoví scénu do počátečního stavu. Klávesy **F2**, **F3**, **F4**, atd. se starají o změnu parametrů scény. Zpravidla **F2** nastavuje krok, kterým lze měnit hodnoty ostatních parametrů. Pomocí **F3** můžete měnit hodnotu *ERP* a klávesa **F4** zase mění nastavení *CFM*. Modifikátor *Shift* otočí význam všech **FX** kláves. Například kombinace kláves **Shift** + **F3** sníží hodnotu *ERP* podle aktuálně nastaveného kroku.

Dále klávesa **B**, resp. **N** vytvoří nový objekt (krychli, resp. kouli). Obnovení scény do výchozího stavu lze provést pomocí **R**. Klávesa **F1** přepíná mezi zapnutím a vypnutím zobrazení *HUD* (*heads-up display*). Uvedené ovládání není platné pro všechny testy, tzn. některé klávesové zkratky nemusí program podporovat.

Celý projekt obsažený v archivu obsahuje následující adresáře:

doc Adresář s dokumentací.

include Hlavičkové soubory knihoven *OpenGL*, *ODE* a *SDL*. Tyto soubory jsou potřebné pro překlad pod *Windows*, protože není možné spolehlivě zajistit existenci dostačujících verzí zmiňovaných závislostí.

lib Statické knihovny pro *ODE* a *SDL*. Také pouze pro *Windows*.

shaders Zdrojové soubory vertex a fragment shaderů pro výpočet osvětlení (per-pixel lighting a shadow mapping napsaný v *GLSL*).

utils Další podpůrné zdrojové soubory pro vytvoření *SDL* okna, třídy pro vykreslení geometrie, práci s *GLSL*, tak umístění objektů do *ODE* simulace.

V kořenovém adresáři projektu je třeba si povšimnout některých důležitých souborů:

Makefile Obecný makefile soubor použitelný pod *GNU/Linux* (testováno na *Ubuntu* a *ArchLinux*), popř. libovolný operační systém typu *UNIX*.

Podmínkou funkčnosti je nainstalovaný překladač *gcc-g++*, *GNU make* a *pkg-config*.

Makefile.win Makefile pro operační systém *Microsoft Windows*. Podmínkou funkčnosti je nainstalovaný překladač *MinGW* (*MSYS není třeba*) s balíčky *mingw-runtime*, *w32api*, *gcc-g++* a *mingw32-make*.

LICENSE Licence vztahující se na všechny zdrojové kódy projektu.

8 Metoda `attach`

Tato metoda vytváří kompozitní objekty. Postup který bude v této kapitole popsán, bude směřovat k vytvoření tělesa (`body`) a s jednou nebo více geometriemi (`geom`). Z toho plyne že každá instance třídy `AbstractObject` obsahuje původní geometrii a jedno těleso vzniklé kombinací původních těles. Zdrojový kód je součástí třídy `AbstractObject` ve stejnojmenném `.cpp` souboru umístěném v adresáři `utils`.

Na začátku metody jsou vedeny deklarace potřebných proměnných.

```
void AbstractObject::attach(const AbstractObject &other)
{
    dMass compositeMass;    // hmotnost vstupního tělesa
    dMass componentMass;   // hmotnost aktuálního tělesa
    dVector3 bodyPos, otherBodyPos;
    dMatrix3 bodyRot, otherBodyRot;
```

Následuje podmínka, jestli aktuální objekt je tělesem.

```
    if (body != 0)
    {
```

Dále si uložíme hmotnosti obou těles:

```
        dBodyGetMass(body, &componentMass);
        dBodyGetMass(other.body, &compositeMass);
```

Nyní se provede rotace a posun obou hmotností podle současného rozmístění těles.

```
        dBodyCopyPosition(other.body, otherBodyPos);
        dBodyCopyRotation(other.body, otherBodyRot);
        dMassRotate(&compositeMass, otherBodyRot);
        dMassTranslate(&compositeMass, otherBodyPos[0],
```

```
            otherBodyPos[1], otherBodyPos[2]);

        dBodyCopyPosition(body, bodyPos);
        dBodyCopyRotation(body, bodyRot);
        dMassRotate(&componentMass, bodyRot);
        dMassTranslate(&componentMass, bodyPos[0], bodyPos[1],
            bodyPos[2]);
```

Sečteme obě hmotnosti, přičemž výsledek je uložen v `compositeMass`.

```
        dMassAdd(&compositeMass, &componentMass);
```

Provedeme kontrolu konzistence struktury `dMass`. Hmotnost a matice setrvačnosti, musí být pozitivně definitní.

```
        if (dMassCheck(&compositeMass) != 0)
        {
```

Přiřadíme k aktuální geometrii těleso z druhého objektu (`other`).

```
            dGeomSetBody(geom, other.body);
```

Geometrii aktuálního objektu nastavíme relativní posun a rotaci od objektu `other`.

```
            dGeomSetOffsetPosition(geom,
                bodyPos[0] - otherBodyPos[0],
                bodyPos[1] - otherBodyPos[1],
                bodyPos[2] - otherBodyPos[2]);
            dGeomSetOffsetRotation(geom, bodyRot);
```

Hmotnost `compositeMass` vrátíme do lokálních souřadnic a přiřadíme ji k tělesu `other`.

```
            dMassTranslate(&compositeMass,-compositeMass.c[0],
                -compositeMass.c[1], -compositeMass.c[2]);
            dBodySetMass(other.body, &compositeMass);
```

Nakonec smažeme původní těleso `body` a přepíšeme hodnotu v proměnné hodnotou `other.body`.

```
            dBodyDestroy(body);
            body = other.body;
            composite = true;
        }
    }
}
```

Situace během kreslení geometrie objektu je velmi jednoduchá. Stačí získat pozici a rotaci geometrie:

```
dVector3 vPos;
dMatrix3 mRot;
dGeomCopyPosition(geom, vPos);
dGeomCopyRotation(geom, mRot);
```

kterou použijeme rovnou k vytvoření modelační matice *OpenGL*. Offsety pozice a rotace jsou automaticky připočítány k těžišti tělesa.

Literatura

[1] Adrian Boeing and Thomas Bräunl. Evaluation of real-time physics simulation systems. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, GRAPHITE '07, pages 281–288, New York, NY, USA, 2007. ACM.

[2] D. H. Eberly. *Game physics*, volume ISBN 1-55860-740-4. Elsevier, Inc., San Francisco, 2004.

[3] P. Karlík. *Vlastnosti fyzikalního enginu Havok Physics*. ZČU, 2009.

[4] A. Kirmse. *Game programming gems 4*. GAME PROGRAMMING GEMS SERIES. Charles River Media, 2004.

[5] Russell Smith. Ode reference manual. `http://opende.sourceforge.net/wiki/index.php/Manual`, April 2011.

[6] Zogrim. Popular physics engines comparison: Physx, havok and ode. `http://physxinfo.com/articles/?page_id=154`, April 2011.