Gentlemen Boxing 5 - Grymt flow on ice

Edqvist, Anders
Hellsten, Joakim
Svensson, Tord

3 juni 2003

1 Inledning

Denna rapport beskriver projektarbetet GB5, ett projekt inom ämnet datorgrafik, i kursen TSBB50 - Bildbehandling projektkurs vid institutionen för systemteknik (ISY) vid Linköpings tekniska högskola.

2 Backgrund och Administrativa detaljer

Vi implementerade spelet i C++ och använde OpenGL/GLUT för grafiken.

Under projektets gång dokumenterade vi vår källkod med Doxygen[1], en sorts JavaDoc för C++. Vi hade som policy att aldrig checka in odokumenterad kod och det var först de två sista veckorna vi bröt mot den regeln. Dokumentationen innehåller förutom förklaringar av funktioner även kodexempel som på ett illustrativt sätt visar infrastrukturen mellan olika klasser.

Alla datorer som spelet utvecklades på har inte Doxygen installerat så för att hela tiden ha tillgång till aktuell dokumentation skapade vi ett Cronjobb som varje timme checkade ut projektet, byggde dokumentationen och publicerade den på webben.

Självfallet använde vi CVS i utveckligen, vi hade vidare viss nytta av ViewCVS[2] för att klargöra ”diffar” under projektets gång.

Vi satte även upp ett webbforum för att diskutera olika problem vi stötte på under projektet. Här publicerade vi även länkar, deadlines och statusrapporter.

3 Gentlemän som slåss

I datorgrafikprojektet *Gentlemen Boxing 2* var slagskämparna hierarkiska modeller med rätblock som kroppar. Det fanns ingen gravitation eller annan fysik såsom tröghet. Kollisionshanteringen gick ut på att spela upp föranimerade ”träffanimationer” och sedan flytta isär figurerna med ett absolut avstånd.

[1] http://www.stack.nl/ dimitri/doxygen/

För att råda bot på dessa problem så såg vi oss om efter lämpliga referenser och fann [Jakobsen]. Artikeln beskriver på ett enkelt sätt hur man kan göra coola saker och eftersom spelet som artikeln baserats på är tufft så bestämde vi oss för att följa receptet.

3.1 Partikelsystem med begränsningar

[Jakobsen] föreslår att spelfigurer ska modelleras med hjälp av ett kraftfullt partikelsystem. I systemet ska knutpunkterna i modellens skelett motsvaras av partiklar med en massa. För att hålla ihop partiklarna introduceras begränsningar (eng. constraints) mellan partiklarna så att avståndet mellan dem håller en förbestämd längd. Begränsningarna tar även hänsyn till partiklarnas tyngd för att skapa en så realistisk modell som möjligt.

3.1.1 Partikelsystemsloopen

[Jakobsen] föreslår följande struktur på partikelsystemets kärna.

Först sätter funktionen `AccumulateForces` de olika partiklarnas accelerationer.

Funktionen `Vervlet` updaterar varje partikels position enligt:

$$\vec{x}' = \vec{x} + \vec{v}_x + \vec{a}_x \cdot \Delta t^2$$

`SatisfyConstraints` itererar en gång över varje begränsning och flyttar partiklarna så begränsningen uppfylls.

`CheckAndHandleCollisions` har till uppgift att finna kollisioner och att separera de inblandade partiklarna.

Idén är nu att utföra de två sista stegen flera gånger i följd för att uppnå ett stablit tillstånd utan kollisioner och utan brutna begränsningar.

```
void ParticleSystem::timeStep()
{
  AccumulateForces();
  Vervlet();
  for(...;...;...)
  {
    SatisfyConstraints();
    CheckAndHandleCollisions();
  }
}
```

3.1.2 Solida kroppar

Solida kroppar kan skapas relativt enkelt, exempelvis ger sex begränsningar på fyra partiklar en tetraeder som automatiskt ser till att dess form behålls även om en av partiklarna förflyttas. Genom att låta två kroppar dela en eller flera partiklar kan man även skapa ledade modeller.

3.2 Modellernas kroppar

Att specifiera ett skelett för våra modeller var relativt enkelt, men sen gällde det att skapa en yta också.

Först tänkte vi (precis som artikeln antydde) att vi skulle skapa ytpartiklar som kopplas till skelettpunkter via begränsningar och sedan att alla ytpartiklar ska repellera varandra. Mellan ytpunkterna skulle man sedan kunna rita polygoner. Detta visade sig förstås vara precis så svårt som det låter komplicerat, så vi beslöt oss för att rita cylindrar runt armar och ben. Bröstkorg och bål ritar vi däremot ut mellan partiklar, men för att få en tjocklek skapar vi artificiella partiklar som förflyttas en sträcka längs kroppens normal ut från skelettets partiklar.

3.3 Modellernas rörelser

Nästa steg var självklart att få modellerna att röra på sig. Här var inte [Jakobsen] särskilt hjälpsam utan säger lite slött att man ”bara” ger en hastighet till partiklarna och låter partikelsystemet skapa en rörelse och att detta enklast görs genom att ärva accelerationer från ett ”underliggande traditionellt animeringssystem”.

I Gentlemen Boxing 2 använde vi hierarkiska modeller och animerade dessa, borde inte de kunna användas som ett ”traditionellt animeringsystem”?

3.4 Hierarkiska modeller

Vi nyimplementerade generella hierarkiska modeller och skapade ett filformat för dessa, bestående av:

1. Flyttalsvariabler
2. Noder
   Noderna har en kedja av transformationer för att förändra aktuella matrisen. Varje nod har även barnnoder vilka i sin tur har transformationer som flyttar $(0, 0, 0)$ från föräldernoden- till barnnodens position.
3. Transformationer
   Dessa kan vara av rotations- eller förflyttningstyp där parametrarna bestäms av antingen konstanter och variabelreferenser.

3.4.1 Animering av den hierarkiska modellen

Genom att ändra värdena på de variabler som finns i modellen kan den animeras. Vi skapade en struktur där varje rörelsemönster var en animation som i sin tur innehöll en kedja av variabelförändringar som skulle utföras under en specifik tidsrymd. Vidare kan flera animationer ske parallellt för att skapa ”kombos”.

Abstraktionen gjorde att koden bitvis blev väldigt enkel, här en fruktad kombo!

```
HiearchialModel hm(Dictionary::fromFile("modellen.txt"));
...
hm.play("left punch");
hm.play("right punch");
```

3.5 Koppla partikelmodellen till hierarkiska modellen

Problemet hur vi ska ”ärva” accelerationer från den hierarkiska modellen var inte något [Jakobsen] berättade, dessutom skulle vi på något sätt få modellen att hålla balansen och inte ramla omkull.

3.5.1 Det puliga sättet

Då ingen i gruppen har läst varken fysik eller mekanik på högre nivå, drog vi därför till med den enda formeln för rörelse vi kommer ihåg från gymnasiet:

$$s = \frac{a \cdot t^2}{2}$$

Vi tyckte att om s är totala sträckan en nod behöver flytta sig i en posövergång och t är tiden den ska ta borde vi kunna få ut a. Dessutom är a konstant och behöver bara räknas en gång. Riktningen på $\vec{a}$ för partikeln som motsvaras av noden måste också härledas, vi testade två metoder:

1. Den riktning som noden i hierarkiska modellen förflyttar sig i vid uppdateringen.
2. Den riktning som partikel har till den position som noden har då animationen är slut.

Den andra metoden var klart bättre. Resultatet så här långt var dock helt värdelöst, p.g.a. tre faktorer:

1. Gravitationen gjorde att modellen föll omkull.
   För att lösa detta testade vi flera metoder däribland att ge alla partiklar en liten hastighet mot motsvarande nod i den hierarkiska modellen. Resultatet blev väl sådär, men det var iaf för svårt att animera och försöka tampas med effekterna av gravitationen.

2. Modellen blev instabil och vibrerade.
   Det var justerandet av begränsningarna som skapade detta, genom att flytta punkter en aning hela tiden vilket ger hastigheter i partiklarna. Vi angrep problemet på två sätt:

   (a) Genom att göra mjukare begränsningar med en felmarginal på 5%. På så sätt ger inte små (obetydliga) fel upphov till ny rörelseenergi i systemet.

   (b) Genom att hela tiden låta en del av rörelseenergin i systemet försvinna, detta genom att ändra i partikelsystemet så att en partikels hastighet avtar. Den nya uppdateringfunktionen blir då,

$$\vec{x}' = \vec{x} + \lambda \cdot \vec{v}_x + \vec{a}_x \cdot \Delta t^2 \qquad (\lambda < 1)$$

3. Eftersom accelerationen hela tiden är konstant så kommer partikeln ha som högst fart i slutet av rörelsen, detta är inte vad vi vill. Vidare så kommer rörelsen att stympas eftersom den lilla fartökning som hela tiden byggs upp kommer att ätas upp av $2b$ ovan.

   Vår funktion $a(t, s)$ kanske inte var speciellt bra när allt kommer omkring men ett byte till en linjärt avtagande $a(t, s)$ hjälpte inte.

3.5.2 Det puligare sättet

Då idén att ärva accelerationer inte gav något (i ett spel) användbart resultat bestämde vi oss för att försöka med något helt annorlunda. Vi kopplade en begränsning med ganska låg fjäderkonstant mellan varje partikel och motsvarande nod. Detta gjorde att partiklarna försökte följa efter noderna och vips löste vi problemen med balansen, vibrationerna och att maximala hastigheten var i slutet av rörelsen!

Om vi gör så att en nod (rotnoden) vid varje uppdatering flyttas till positionen för en partikel så kan vi mappa hierarkiska modellen mot partikelmodellen på ett enkelt sätt. Detta får den trevliga effekten att om hierarkiska modellen sätter sig på huk och sedan hastigt reser sig så hoppar partikelmodellen!

Genom att släppa alla begränsningar mellan noderna och partiklarna så kan man skapa en dödsscen där modellen faller till marken enligt den korrumperade fysikens lagar.

4 Texturer

För texturhantering skapade vi en centraliserad texturhanterare samt ett par bildläsningsmoduler för olika bildformat (PPM och TGA). Den centraliserade texturhanteraren aktiverar vald textur inför utritandet av objektet, avaktiverar texturen när utritandet är klart och gör det möjligt att globalt slå av och på texturstödet. Exempel:

```
// ladda texturen
PPMImage marble("resources/tests/textures/marble128.ppm");
TextureHandler* th = TextureHandler::getInstance();
TexNum tex;
tex = th->load("marble",
               marble.getData(),
               marble.getWidth(),
               marble.getHeight());
// aktivera texturstödet
th->setEnabled(true);
...
// rita ut ett texturerat objekt
th->activate(tex);
drawMyObject();
th->deactivate(tex);
```

Den ovan beskriva texturhanteraren fanns vara för grovkorning varpå ”texturklasser” lades till. En texturklass är helt enkelt en identifierare till vilken vissa attribut är knutna, så som om klassen är aktiverad och om den har några effekter associerad till sig. Detta gör det alltså möjligt att slå av och på texturering för endast vissa objekt, till exempel modellerna.

4.1 Textureffekter

Innan en textur appliceras transformeras den enligt texturmatrisen, liksom geometriska objekt transformeras enligt modelvymatrisen. Genom att manipulera, den ofta bortglömda, texturmatrisen kan man skapa enkla, snabba effekter som exempelvis moln, vatten och vissa typer av dimma.

Vi skapade bland annat textureffekter för att efterlikna moln som rör sig på himlen. Detta gjorde vi genom att först skapa en effekt som applicerar en translation på texturmatrisen (”MovingTexture”). Denna använde vi sedan på flera lager av polygoner texturerade med transparenta texturer (för att skapa en parallax-liknande effekt). Exempel:

```
MovingTexture lowerFX(0.00005,0.0005);
MovingTexture upperFX(0.000005,0.00005);

TextureHandler* th = TextureHandler::getInstance();

TexClass upperClass = th->newTexClass();
TexClass lowerClass = th->newTexClass();

TGAImage clouds("resources/tests/textures/clouds.tga");
```

```
th->addEffect(lowerClass,lowerFX);
th->addEffect(upperClass,upperFX);

TexNum tex;

tex = th->load("clouds",
               clouds.getData(),
               clouds.getWidth(),
               clouds.getHeight());

...

th->activate(tex,upperClass);
drawUpper();
th->deactivate(tex,upperClass);
th->activate(tex,lowerClass);
drawLower();
th->deactivate(tex,lowerClass);
```

4.2 Texturuppdelning

Från GB2 hade vi lärt oss att det inte är tillräckligt med en $128 * 128$ pixlar stor textur till bakgrundsbild för menysystemet. Vi började då titta på texturuppdelning (eng. texture paging) som en möjlig lösning.

Generellt, tycks texturuppdelning lösas på två sätt, antingen genom tvåpassrendering eller genom att dela upp den potientiellt komplex geometriska figuren i för texturen lämpliga delar. Eftersom vår avsikt endast var att använda stora texturer till bakgrundsbild för menysystemet beslutade vi oss för att använda en mycket förenklad variant av den senare lösningen.

Vi tog helt enkelt en bild och delade upp den[3] $(w/128) \cdot (h/128)$ ”quads” à 128*128 pixlar. När sedan bilden skall renderas renderar vi själva verket alla quads:en med rätt del av texturen applicerad på rätt quad.

5 Billboards

Vår implementation av billboards består av en centraliserad hanterare som uppdaterar billboardens orientering när så behövs (när kameran har modifierats). Den omorientering mot kameran som eventuellt behöver göras beräknas utifrån projektionsmatrisen.

För att undvika att en billboard som ligger bakom en annan billboard inte syns genom den första billboarden måste billboardsen sorteras efter avståndet mot kam-

[3] Här betecknar w bildens bredd och h dess höjd.

eran och sedan ritas ut i korrekt ordning. Detta tar också den centraliserade hanteraren hand om.

6 Partikelsystem för specialeffekter

Partikelsystem har använts sen i början av 80-talet för olika typer av effekter, såsom eld, rök, explosioner, moln etc. Första gången partikelsystem användes för specialeffekter var troligen i en sekvens[4] i Star Trek II: The Wrath of Kahn (1982), där man använde 400 partikelsystem med totalt 750 000 partiklar.

Ett partikelsystem är en samling partiklar, vilka kan vara punkter eller geometriska objekt, i rymden. Partiklarna har ett antal attribut, som position, riktning, energi, färg transparens m.m., vilka avgör hur partiklarna rör sig och hur de förändras i varje uppdatering. Partikelsystemet skapar partiklarna, bestämmer deras attribut, tar bort partiklar som inte längre behövs och uppdaterar partiklarnas position. Det är också möjligt att låta varje partikel i sig vara ett partikelsystem, t. ex. om en partikel i ett eldsystem träffar ett brännbart objekt skapar partikeln ett eldsystem där.

Som teoretisk utgångspunkt och inspiration har vi använt oss av [Lander] och [van der Burg].

6.1 Implementation

Vår implemenation av partikelsystemet för specialeffekter består av tre klasser:

- partikel (FXParticle),
- partikelsystem (FXParticleSystem)
- partikelsystemshanterare (FXParticleSystemManager)

Vi skapade med dessa klasser tre effekter: eld, eldboll och blod.

6.2 Partiklar

Partikelklassen innehåller variabler för partikelns position, föregående position, hastighet, färg (i RGBA), energi, uttoningshastighet och om den är levande eller inte.

Vi har här valet att låta partiklarna antingen ha en polygon, punkt eller linje som grafisk representation. Är partikeln en punkt är det bara att rita den, är den en linje så kan vi dra den mellan nuvarande och föregående position och är den en polygon kan vi använda nuvarande position som centrum. I de effekter vi gjorde var varje partikel två texturerade trianglar, varvid vi fick problemet att polygonen alltid måste vara roterad mot kameran för att partikeln skulle synas. Den uppenbara

[4] http://www.siggraph.org/education/materials/HyperGraph/animation/movies/genesisp.mpg

lösningen är att låta varje partikel vara en billboard, men det blir problem ur ett beräkningsperspektiv då vi kan ha tusentals partiklar samtidigt. För att dra ner på beräkningarna låter vi istället varje partikel projiceras mot kamerans plan genom att göra rotationen och skalningen i modellvymatrisen ogjorda genom att ersätta den övre-vänstra 3/3-matrisen med identitetsmatrisen.

Hastighet och riktning används uppenbarligen för att beräkna partiklarnas nya positioner. Färgen används för blending och transparens, energin för att avgöra om partikeln lever och uttoningshastigheten för att minska partikelns energi. När energin når noll antas partikeln i de flesta system vara död och kan då återinitieras om man har effekter som varar längre.

Energin använder vi som värde på alphakanalen för att låta effekterna tona ut och till slut försvinna.

6.3 Partikelsystem

Partikelsystemsklassen innehåller en lista med de partiklar systemet består av, en variabel med antalet partiklar som lever och metoder för att initiera, uppdatera samt rendera partikelsystemet samt för att avgöra om systemet har några aktiva partiklar eller inte.

Vid initeringen av partikelsystemets allokeras utrymme för den mängd partiklas som önskas, och varje partikel får värden på sina variabler. I uppdateringen förflyttas partiklarna och värdena på energi och färg förändras. Om en partikels energi har tagit slut och den kan återinitieras så sker det även här. Innan partiklarna renderas i renderingsmetoden beräknas deras position på skärmen samt projektion mot kameraplanet.

6.4 Partikelsystemshanteraren

Partikelsystemshanteraren innehåller en lista med de partikelsystem som den hanterar, och tillhandahåller metoder för att lägga till partikelsystem, och för att uppdatera och rendera de partikelsystemshanteraren hanterar.

När ett partikelsystem skapats läggs det till i listan, och när alla partiklar i systemet är inaktiva tas det bort och avallokeras. Uppdaterings- och renderingsmetoderna anropar respektive metod i respektive partikelsystemsobjekt.

6.5 Våra effekter

Med ovanstående klasser implementerade vi tre specialeffekter. Behovet av specialeffekter blev i slutändan inte så stort. Hade vi gjort fler arenor och kanske häftiga specialkrafter för figurerna hade vi kunnat använda oss av dem bättre. I de tre effekterna representeras varje partikel av en två ”trianglestrips” som ges en (halvtransparent) textur.

6.5.1 Blod

För en blodeffekt som ser bra ut klarar man sig med tio till femtio partiklar per effekt, då det snabbt blir många träffar på respektive slagskämpe. Partiklarna initieras i träffpunkten, görs röda och far iväg i slagets riktning med en viss slumpfaktor för spridning. Partiklarna tonas ut snabbt då de inte behövs längre än att de de passerat golvet. Det är relativt enkelt att lägga till kollisionsdetektering med golvet för att öka realismen, och även att låta partiklarna skapa egna system där de träffar för att simulera rinnande blod (även från väggar).

6.5.2 Eld

Ju fler partiklar som finns i en eld vid en given tid, ju större blir elden. Vi låter partiklarna få en gulvit färg vid initieringen och låter dem sedan tonas ut mot rödbrunt då de rör sig från eldens centrum för att simulera avsvalning. När en partikel helt tonats ut återinitieras den tills eldens centrum och processen börjar om från början.

6.5.3 Eldboll

För att simulera en eldboll flyttar vi punkten i vilka partiklarna initieras längs önskad vektor. I övrigt fungerar effekten i princip som eldeffekten.

7 Dataformat

Vår initiala avsikt var att använda XML som format för bland annat beskrivningen av modellerna. Vi undersökte flera tredjeparts bibliotek (bland annat xpat) men fann antingen att biblioteken var för stora [5], för ofärdiga, för krångliga eller inte portabla. Detta resulterade i att vi beslutade oss för att göra ett eget format. En ”recursive descent parser” för ett C++ influerat beskrivningsspråk med stöd för heltal, flyttal, strängar, fält och mappningar utvecklades.

Eftersom vi valde att efterapa C++: syntax för fält och attribut fick vi färgkodning och indentering ”gratis” ev Emacs om vi valde C++-mode, vilket underlättade utvecklingen av modeller enormt.

Ett smakprov på hur modellerna definieras följer här:

```
particles = [
  { name="back" start=[0.0, 0.0, 0.0] mass=10.0 color=[0.0,0.0,1.0]},
  ...
  { name="head" color=[1.0, 0.0, 0.0] mass=0.4 }
]
constraints = [
    { particle1="back" particle2="neck" restlength=1.0 type="STIFF"
                       springfactor=0.9 },
```

[5] för vår diskquota

```
    ...
  { particle1="neck" particle2="head" restlength=0.3 type="STIFF"
                     springfactor=0.9 }
]
```

8 Menysystem

För alla inställningar valde vi att utveckla ett eget menysystem, för att få ett spelliknande utseende, även om användandet av ett befintligt menysystem (eller fönstersystem) hade påskyndat utveckling hade det inte sett lika tufft ut.

Eftersom vi hade relativt stor erfarenhet av befintliga komponentbibliotek valde vi att utveckla ett liknande.

8.1 Lokalisering

I alla spel med självaktning kan man välja mellan åtminstone ett fåtal språk. Vi ville inte vara sämre så vi skapade stöd för olika språk (så kallad lokalisering). Det är influerat av Javas stöd för lokalisering fast vi har endast stöd för den rent språkliga delen (inga formateringstekniska variationer har tagits hänsyn till).

```
Locale english("resources/english");
Locale swedish("resources/svenska");
LocaleManager* lman = LocaleManager::getInstance();
LocalizedString str("Language");

lman->setLocale(&english);
cout  << "English language: " << str.getValue() << endl;

lman->setLocale(&swedish);
cout  << "Swedish language: " << str.getValue() << endl;
```

Ovanstående ger alltså följande utdata, förutsatt att ”Language” finns definierad, och korrekt översatt i båda filerna:

```
English language: Language
Swedish langauge: Språk
```

9 Timer

För att spelet inte skall gå olika fort på olika datorer skapade vi en timer som fungerar både under Solaris, Mac OS X och Windows. Timern fungerar som så att man anger hur lång tid (t) det skall förflyta innan den angivna callback-funktion anropas. Om mer tid har förflutit kommer callback-funktionen att anropas tid-som-förflutit$/t$ gånger medan tid-som-förflutit $mod\ t$ dras av t nästa gång.

Tidsprecisionen mellan plattformarna skiljer sig åt (Windows stödjer en upplösning på millisekund-nivå medan Solaris och Mac OS X på mikrosekunds-nivå) liksom gränssnittet för att utnyttja klockan.

Sedan är timern synkron, alltså, för att callback-funktionen skall kunna anropas krävs det att ”update” funktionen i timern anropas. Ju oftare den anropas desto bättre upplösning ges (med hänsyn taget till plattformen). Ett anrop till timern i GLUTs idle-funktion är således lämpligt.

En effekt av detta är att lokala fluktuationer kan förekomma men över tid så kommer callback-funktionen att anropas korrekt antal gånger.

```
int callbackData;
void callback(Timer* timer, void* arg);

//anropa callback en gång i sekunden
Timer t(TO_MICROSECONDS(1),callback,&callbackData);
t.start();

while(true)
{
  t.update();
}
```

10 Matris- och vektorbibliotek

Sen datorgrafikprojektet visste vi att det skulle vara väldigt användbart med klasser för vektorer och matriser så vi implementerade sådana. Vektorklassen innehåller mycket spännande möjligheter när det gäller var minnet som koordinaterna sparas i finns, de kan nämligen anges per referens. På så sätt kan flera vektorers koordinater sparas efter varandra i en `GLfloat[]` så man kan skicka upp dem på grafikkortet direkt!

Vi hade läst i [Wood] att det fanns användara optimeringsmöjligheter i PPC processorn för kvadratrötter och det drog vi nytta av i `Vector3::normalize()`.

10.1 Superproblemet Allan

Vi hade ett mycket besvärligt problem i vektorklassen. Det grundade sig i hurvida `*` skulle överlagras som skalär-[6] eller kryssprodukt.

En av oss definerade `*` som skalärprodukt:

```
float operator*(Vector3,Vector3);
```

Om man sedan försöker göra nedanstående så borde kompilatorn klaga:

[6] http://www.informatik.umu.se/~svps0009/files/skalarprodukt.mpg

```
Vector3 kryssp = vektor1 * vektor2;|
```

Problemet var bara att konstruktorn för Vector3 var:

```
Vector3(float x=0.0,float y=0.0,float z=0.0);
```

Skalärprodukten konverteradess alltså automatiskt om till en vektor där y och z var 0 medan x-koordinaten var sjävla skalärprodukten! Det här gjorde att en annan av oss i ett skede skrev kod i tron om att det verkligen var kryssprodukt och inte skalärprodukt. Lösningen var att använda två konstruktorer istället och vi kommer nog tänka till en extra gång innan vi använder defaultkonstruktorer i samband med aritmetiska operatoröverlagringar igen.

11 Kollisioner

Inledningsvis ville vi ha sofistikerad kollisionsdetektion mellan konvexa kroppar och jobbade därefter. [Jakobsen] ställer krav på att `CheckAndHandleCollisions()` fungerar på ett visst sätt så vi skrev en halvfärdig implementation för detta. När det sedan framgick att det var mycket obehändigt att skapa avancerade modeller och vi började använda cylindrar istället gick den kollisionshantering vid hittills jobbat på om intet.

I Gentlemen Boxing 2 dög det utmärkt att upptäcka kollisioner med hjälp av sfärer så vi beslöt att göra så även i det här projektet, dock skulle hanteringen av kollisioner bli bättre.

Längs cylindrarna, som går mellan två partiklar, så skapade vi en rad av sfärer. När en kollision mellan två sfärer upptäcks separeras sfärerna, detta gör att hela cylindern som sfären hör till förflyttas och på så sätt även ändpartiklarna. På det här sättet får vi just de berörda kroppsdelarna att röra på sig vid en kollision och behöver inte använda föranimerade kollisionsrörelser!

För att avgöra vem som tar skada och hur mycket vid en kollision så jämförde vi hastigheterna hos kolliderande sfärerna.

12 AI

Vi skapade en mycket enkel AI genom att simulera en slumpmässig relevant tangenttryckning och sedan sova en slumpmässig tid, inom ett givet intervall.

Detta skapade en ganska häftig om än något för ivrig AI som vilt slår på allt och alla. Även om det är relativt enkel att besegra så var det ett nöje att beskåda två AI-spelare som slogs.

13 Ljud

Enligt vår kravspec skulle vi ha ljud i spelet. Vår första tanke var att själva skapa ett plattformsoberoende ljud API utav olika plattformsspecifika tekniker. Vi började

med att titta på olika möjligheter för att spela upp ljud under Mac OS X och tittade på CarbonSound och AudioUnits. Tyvärr så är Apples dokumentation av ljudmöjligheterna i OS X precis lika dåliga som dokumentationen för asynkron nätverkskommunikation för allt annat ljud än NSSound. Detta gjorde att vi bestämde oss för att använda ett högnivå API istället.

13.1 OpenAL

När vi skulle välja ljud API så låg valet mellan SDL och OpenAL. Vi valde till slut OpenAL för att det skeppas som en framework och för att det är en relativt liten standard. För att verkligen förstå hur det fungerade läste vi [Creative], den officiella specifikationen [Loki] samt diverse exempelkod på internet.

13.1.1 C++-klasser för OpenAL

För att göra OpenAL än mer lättarbetat tog vi fram en del C++ klasser som kan förvandlas till ”stubbar” om inte en viss kompileringsflagga anges.

1. ALWavFile
   Klassen läser in WAV filer som sedan kan kopieras in i ljudbuffrar. För att krångla till saker och ting så finns det funktioner i OpenAL som inte är standardiserade och har olika parametrar på olika plattformar, trots att implementationerna kommer från samma tillverkare.

2. ALBuffer
   Det här motsvarar föga förvånande en ljudbuffer. Objektorienteringen passar utmärkt in i OpenAL här eftersom konstruktorn kan allokera resurser på ljudkortet som sedan destruktorn kan frigöra.

3. ALSource
   Den här klassen är en ljudkälla, den har en position, volym o.s.v. och den knyts till ett ljud via en ALBuffer.

4. ALListener
   Detta är själva lyssnarobjektet. I OpenAL finns det bara en lyssnare men i vår C++:ifiering kan det finnas flera olika lyssnare varav bara en aktiv åt gången.

5. CameraWithListener
   Det här är en smidig underklass till vår kameraklass. Den innehåller ett ALListener objekt som hela tiden synkas gentemot kamerans position och riktning.

Här är ett litet enkelt exempel på hur det kan se ut:

```
ALWavFile ljudfil("mittljud.wav");
ALBuffer buffer(ljudfil);
```

```
ALSource ljudkälla(&buffer);
ALListener jag;
jag.setPosition(theCamera.position());
...
ljudkälla.setPosition(Vector3(3,0,8));
ljudkälla.play();
```

13.2 Problem

Det kompilerar och fungerar att använda OpenAL under Cygwin på CYD-poolen. Tyvärr är det inte så att OpenGL och OpenAL går att kombinera sådär utan vidare. Vi tillbringade en hel natt på CYD med att försöka lista ut hur g++ vill ha sina argument för att det ska funka. Till slut lyckades vi och det som hjälpte var att flytta på glut32.dll från buildkatalogen och sedan flytta tillbaka den innan programmet skulle köras, varför detta hjälpte vet vi inte.

Problemet skulle nog inte uppstått om vi haft behörighet att ändra i Cygwins egna kataloger och pluppat in OpenAL den vägen, men som slutsats kan man säga att Cygwin inte är UNIX och att frameworks är 100 gånger smidigare än DLL- och LIB-filer för en utvecklare.

14 Referenser

Jakobsen, Thomas. Advanced Character Physics
http://www.ioi.dk/Homepages/thomasj/publications/gdc2001.htm

Creative OpenAL Programmer's Reference, 2001
http://developer.creative.com/articles/article.asp?cat=1&sub=31&top=38&aid=55&file=OpenALProgGuideA4.pdf

Loki, OpenAL Specificatoin And Reference, 2000
http://www.openal.org/snapshots/openal/docs/oalspecs-annote/index.html

Wood, Timothy J, Porting To Mac OS X, 2001
http://www.omnigroup.com/ftp/pub/software/Source/MacOSX/Games/GDC2001-PortingToOSX.pdf

van der Burg, John, Building An Advanced Particle System, Game Developer Magazine March 2000
www.mysticgd.com/misc/AdvancedParticleSystems.pdf

Lander, Jeff, The Ocean Spray In Your Face, Game Developer Magazine July 1998
www.darwin3d.com/gamedev/articles/col0798.pdf

SIGGRAPH 97 Conference Proceedings