HÖGSKOLAN SKÖVDE

Institutionen för kommunikation och information
Examensarbete i datalogi 30hp
C-nivå
Vårterminen 2008

Simulering av rök på GPU

Användning av GPGPU för att simulera rök

Erik Jalsborn

Simulering av rök på GPU

Examensrapport inlämnad av Erik Jalsborn till Högskolan i Skövde, för Kandidatexamen (B.Sc.) vid Institutionen för kommunikation och information. Arbetet har handletts av Mikael Thieme.

2008-06-01

Härmed intygas att allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och att inget material är inkluderat som tidigare använts för erhållande av annan examen.

Signerat: _____

Simulering av rök på GPU

Erik Jalsborn

Sammanfattning

Detta examensarbete undersöker en befintilig teknik för att simulera rök med ett partikelsystem. Tekniken utvecklas och implementeras så att beräkningar av partiklars nya positioner sker på både en CPU och en GPU. Arbetet gör undersökningar baserat på tidseffektivitet och visar att simulering av röken sker snabbare, när beräkningarna av partiklars nya positioner görs på GPU'n, istället för CPU'n.

Nyckelord: Rök, Simulering, GPGPU, Realtidssimulering, Partikelsystem, Partiklar

Innehållsförteckning

1 Introduktion

Avancerade fysikaliska fenomen som ska simuleras i applikationer, t.ex. rök, baseras ofta på komplexa algoritmer. För att dessa komplexa algoritmer ska kunna utföras inom den tidsram som gör att de uppfattas i realtid, krävs det hårdvara med hög beräkningskapacitet. I dataspel är det extra viktigt att beräkningskapaciteten är hög då det är så mycket mer än bara olika fysikaliska fenomen som ska simuleras. Exempel på sådant som tar mycket beräkningskapacitet i spel är artificiell intelligens. I de flesta dataspel implementeras rök och liknande som partikelsystem där beräkningarna sker på en CPU. Detta fungerar dock bara för partikelsystem som inte är alldeles för avancerade. I takt med att hårdvaran utvecklas kommer hela tiden nya metoder och tekniker som kräver den beräkningskapacitet som den nya hårdvaran har att erbjuda. Det finns speciell hårdvara som är till för att utföra komplexa fysikaliska beräkningar och på så vis avlasta CPU'n. Ett exempel på sådan hårdvara är fysikkortet PhysX från företaget AGEIA. Det som bland annat gör att detta kort kan utföra beräkningarna snabbt är att arkitekturen är parallell, det vill säga, många beräkningar kan utföras samtidigt.

Istället för att använda ett dedikerat fysikkort för att utföra beräkningar av olika slag kan man utnyttja en grafikprocessor (GPU). Detta eftersom en GPU också har en arkitektur som är konstruerad för att utföra parallella beräkningar. En stor fördel med att använda en redan tillgänglig GPU för att simulera ett fysikt fenomen är att konsumenten då inte behöver köpa ytterligare hårdvara, då de flesta personer som spelar nya avancerade 3D-spel redan har grafikkort med hög beräkningskapacitet. Att använda en GPU för exekvering av algoritmer och liknande som inte är menade för en CPU kallas för General Purpose computation on GPU och är ett relativt nytt begrepp i datorvärlden.

Det här arbetet undersöker en befintlig teknik för att simulera rök med ett partikelsystem, som presenteras utav Gustavsson, Engström och Gustavsson (2006). Tekniken utvecklas för att simuleras på både en CPU och en GPU. En viktig del i detta arbete är att den utvecklade tekniken ska simuleras snabbare på GPU'n än vad den gör på CPU'n. För att undersöka ifall GPU-simuleringen sker snabbare så utvärderas den utvecklade tekniken och olika scenarier genomförs. En annan viktig del av arbetet är att röken som simuleras på GPU'n ska uppföra sig likt verklig rök, något som också undersöks via utvärdering av tekniken.

2 Bakgrund

Detta avsnitt innehåller information som är viktig att känna till för att förstå arbetet. Kapitlet börjar med att beskriva vad ett partikelsystem är, för att sedan övergå till tekniken för att simulera rök som Gustavsson et al. (2006) presenterar. För att förstå tekniken är kunskap om vad ett partikelsystem är grundläggande. Därefter beskrivs termen ”General Purpose computation on GPU” (GPGPU) och några begrepp som är viktiga att känna till för att skapa en minimal GPGPU-applikation.

2.1 Partikelsystem

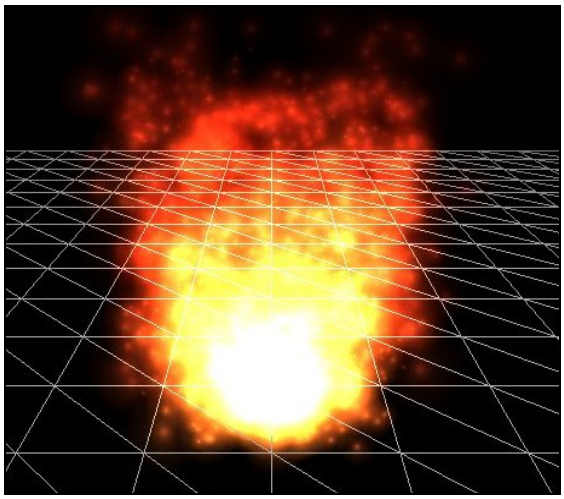
Enligt Reeves (1983) är partikelsystem en metod för att modellera ”oklara/luddiga” objekt såsom vatten, eld och moln. Metoden fungerar på ett sådant sätt att partikelsystemet innehåller en mängd partiklar som tillsammans bildar en volym. Dessa partiklar har alla samma egenskaper men med olika värden. Egenskaper som en partikel har är enligt Reeves (1983) position, hastighet, storlek, färg, genomskinlighet, form och livstid. Dessa egenskaper har partiklar oftast fortfarande idag, även fast Reeves tog fram metoden 1983.

Något som Reeves (1983) inte nämner är begreppet emitter (avsändare). Emittern är källan där nya partiklar skapas för att sedan på något sätt få exempelvis en rörelse. I de flesta dataspel och liknande applikationer är emittern osynlig. Den informationen som en emitter innehåller kan vara hur många partiklar som ska genereras per tidsenhet, vilken riktning partiklarna ska röra sig åt när dom skapas, partikelns livstid och mycket mer (Wikipedia, 2008).

När ett partikelsystem simuleras är följande ansats vanlig. Först så skapas ett antal partiklar vid emittern. Därefter appliceras någon eller några algoritmer som beräknar nya egenskaper för partiklarna. När partiklar som finns i partikelsystemet har blivit uppdaterade eller borttagna från systemet på grund av t.ex. för hög livstid, renderas partiklarna med dess uppdaterade egenskaper. När alla partiklar har renderats så görs proceduren om igen.

2.1.1 Exempel

Figur 1 visar ett partikelsystem som används för att simulera eld. I detta partikelsystem finns emittern i mitten av det vitaste i elden och partiklarna som skapas där stiger uppåt. Precis som med vanlig eld så ändras färgen och genomskinligheten på elden beroende på var den befinner sig. Två möjliga sätt att bestämma när en partikel ska försvinna från detta partikelsystem kan vara att antingen ta bort den när en viss livstid har uppnåtts eller när partikeln har nått en viss höjd. I de flesta applikationer som simulerar eld används det första sättet, nämligen livstiden.



Figur 1. Partikelsystem som simulerar eld (Jtsiomb, 2007). (Bilden får användas fritt enligt skaparen)

Ett annat exempel på ett partikelsystem som finns kallas för för statiskt partikelsystem (Wikipedia, 2008). Ett statiskt partikelsystem fungerar ungefär på samma sätt som ett vanligt partikelsystem. Som tidigare beskrivet i detta avsnitt har partiklar en livstid och det är en partikels olika egenskaper vid olika livstider, som är det centrala i ett statiskt partikelsystem. Det man gör är att man sparar undan partikelns egenskaper vid varje tidsteg som går i simulering istället för att enbart uppdatera egenskaperna. På så sätt kan man vid visualiseringen, rendera partikeln med dess egenskaper för alla tidssteg. Man ser alltså hur partikeln såg ut och var den befann sig i tidigare tidssteg. Effekten av att använda ett statiskt partikelsystem kan vara användbar om man exempelvis vill simulera hårstrån. Figuren nedan visar ett exempel på hur ett statiskt partikelsystem kan se ut. I denna figur skapas partiklarna på fem olika ställen och får sin riktning beroende på var någonstans dom skapas. Varje litet hårstrå består av en och samma partikel renderad med dess egenskaper som fås i varje tidsteg.

Figur 2. Statiskt partikelsystem som simulerar hårstrån (Halixi72, 2007). Varje bunt av hårstrån består av 1000st partiklar. (Bilden får användas med licensen *GNU Free Documentation License*, Version 1.2 eller senare)

2.1.2 Partikelsystem i dataspel

I dataspel används partikelsystem flitigt. Det som är viktigt att tänka på när man skapar ett partikelsystem som ska användas i dataspel är att simuleringen måste kunna göras i realtid. Enligt Fernando och Kilgard (2003) anses realtid i en applikation vara när uppdateringar sker 60 gånger eller mer per sekund, det uppfattas alltså för ögat som att uppdateringar sker hela tiden. Realtidssimulering innebär därmed att en simulering av något, exempelvis vatten, uppdateras minst 60 gånger per sekund. Detta leder till att en kombination av väldigt många partiklar och beräkningskrävande algoritmer inte är att föredra när beräkningarna görs på en CPU, då det kan vara svårt att uppnå realtidssimulering. I detta arbete ska, som tidigare beskrivits, tekniken som simulerar rök utvecklas för att beräkningar ska ske på en GPU, vilket förhoppningsvis kommer att ge fler uppdateringar per sekund än röken som simuleras på en CPU.

Ett slags partikelsystem som ofta förekommer i dataspel, speciellt de spel i genren First Person Shooter, är partikelsystem som simulerar blod. Figuren nedan är en del av en screenshot från spelet GTA San Andreas (Rockstar North, 2005) och visar ett exempel på hur blod kan se ut i dataspel.

Figur 3: Blodpartiklar. En karaktär blir skjuten och blodpartiklar uppkommer.

2.2 Röksimuleringen

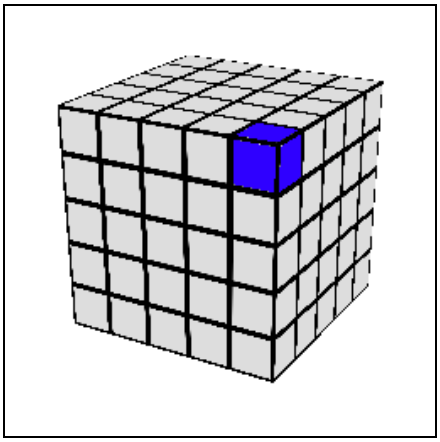
Simulering av rök och simulering av andra gaser och vätskor brukar ofta baseras på Navier-Stokes-ekvationer. Dessa ekvationer beskriver rörelsen för substanser i flytande form (Wikipedia, 2008). Det finns flera olika tekniker för röksimulering där Navier-Stokes-ekvationer har använts för att simulera rök (se exempelvis Fedkiw, Stam & Jensen; 2001 Selle, Rasmussen & Fedkiw, 2005). Ett problem som finns med dessa tekniker är att de innehåller många komplexa algoritmer som gör att det blir svårt att få simuleringen att utföras i realtid.

Tekniken som utvecklas för röksimuleringen som görs i detta arbete är baserad på tekniken som Gustavsson et al. (2006) presenterar i deras arbete. Tekniken som presenteras fokuserar på att få röken att uppföra sig så verkligt som möjligt med relativt okomplicerade beräkningar. Mindre vikt läggs alltså på att få det visuella utseendet (rendereringen) att se så realistisk ut som möjligt (till skillnad från exempelvis Selle, et al. 2005).

Enligt Gustavsson, et al. (2006) är tekniken som presenteras väldigt passande för implementation på en GPU. Det är just därför som detta arbete använder denna teknik som grund. Tekniken som Gustavsson et al. (2006) presenterar i sitt papper förklaras nedan.

2.2.1 Teknik som används

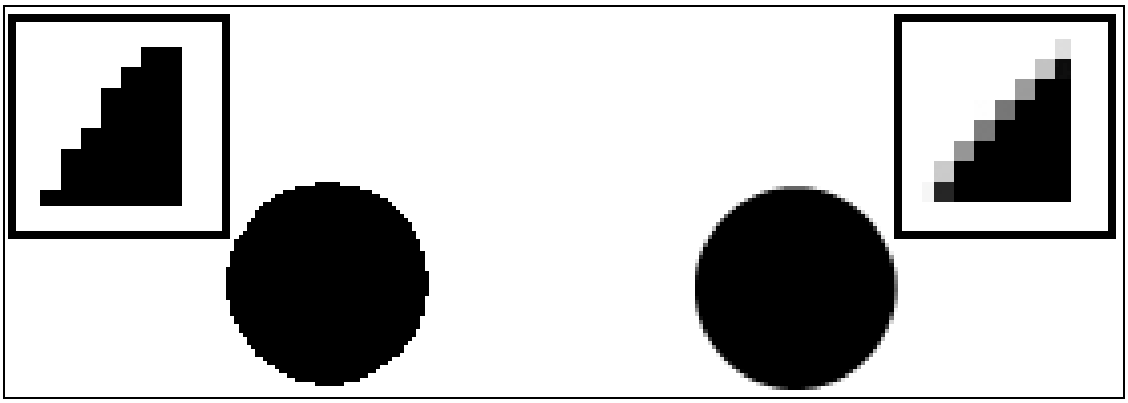
Gustavsson et al. (2006) har arbetat fram en hybrid teknik av gridbaserade och partikelbaserade metoder som använder multisampling för att bestämma hur partiklar ska röra sig. En gridbaserad teknik innebär att man skapar rutnät och använder sig av dessa rutor på något sätt. Gustavsson et al. väljer här att ha en 3D-textur av en bestämd storlek för att representera rutnätet. De kuber som bildas i 3D-texturen, som kan ses i figuren nedan, kallas för voxlar och det är i dessa voxlar som partiklar kan befinna sig i. Den data som varje voxel innehåller är den sammanlagda densiteten för de partiklar som befinner sig i voxeln. Viktigt att notera är att inga partiklar kan befinna sig utanför några voxlar. Denna teknik lämpar sig därför i en applikation där röken på något sätt ska vara instängd, exempelvis en glaskub.



Figur 4: 3D-textur med voxlar som kan innehålla partiklar. Storleken på denna 3D-textur är 5x5x5 och innehåller 125st voxlar. Det blåa i figuren är en voxel

Denna gridbaserade metod används för att man vid partikeluppdatering kan gå igenom närliggande voxlar utifrån den voxeln som partikeln är i, och utföra en algoritm som bestämmer åt vilket håll partikeln ska röra sig. Genomgången av alla närliggande voxlar görs via multisampling i tre olika nivåer som alla ligger olika långt ifrån den valda voxeln och innehåller olika många voxlar. Att man använder multisampling beror på, enligt Gustavsson et al. (2006), att det är en billig teknik då GPU'n gör detta snabbt. Antal sammanlagda samplingar som görs för de tre nivåerna är 46st.

Multisampling är en teknik där man för den nuvarande pixeln i en textur, väljer någon eller några pixlar (sampling points) en bit ifrån och gör något med de värden som dessa pixlar innehåller. I datorgrafik används multisampling bland annat för anti-aliasing. Det är en teknik som kan göra kanter mer jämna. Detta går att se i figuren nedan.



Figur 5: En cirkel utan anti-aliasing (vänster) och med anti-aliasing (höger). Det som finns innanför de två rektanglarna är uppförstoringar av en del av cirklarna.

2.2.2 Individuella partiklars uppdatering

För att uppdatera partiklars positioner används följande ekvation:

$$\mathbf{a}_i = \frac{\sum \mathbf{s}_j \times f(\rho_i)}{\sum f(\rho_i)}$$

Ekvation 1. Uppdatering partiklars position

a är den nya accelerationen som partikeln som uppdateras kommer att få. Gustavsson et al. (2006) använder accelerationen som partikelns hastighet för att ta bort behovet av att känna till både hastighet och acceleration. Detta medför att uppdateringarna går snabbare då ingen ny hastighet behöver beräknas utifrån accelerationen, utan istället kan resultatet användas för att direkt sätta en ny position.

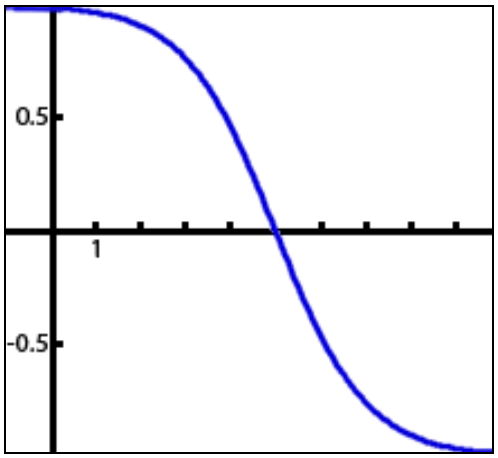
s är samplingsvektorn som används vid multisamplingen. Vektorn är i tre dimensioner och innehåller den relativa längden från den nuvarande voxeln till den voxeln som ska ingå i beräkningarna. Det innebär att en voxel som ligger rakt åt höger från den valda voxeln (samma y-värde och z-värde men med 1 mer i x-värde), hämtas med hjälp av vektorn [1,0,0].

ρ är densiteten som finns i den voxeln som samplingsvektorn hämtar. Densiteten är väldigt central i röksimuleringen då en partikels beteende bestäms av densiteten i de närliggande voxlarna som fås vid multisamplingen.

f är attraktionsfunktionen för partikeln. Den tar voxelns densitet som indata och beräknar utifrån den ifall partikeln ska attraheras eller repelleras beroende på positiv eller negativ utdata. Funktionen visas nedan och ger upphov till sigmoid-kurvan i figur 6.

$$f(\rho) = \left(\frac{2}{1 + e^{\rho - 5}} \right) - 1$$

Ekvation 2. Beräkning av attraktion



Figur 6: Sigmoid-kurvan som ges av funktionen.

Istället för att hela tiden anropa denna funktion och utföra beräkningar när partiklar ska uppdateras så använder Gustavsson et al. (2006) en 1D-textur som fungerar som en så kallad ”lookup table” för sigmoid-funktionen. Denna textur innehåller den utdata som kan returneras vid olika densitetvärden. Detta medför då att ingen beräkning behöver göras på CPU’n för funktionen.

En annan viktig del i tekniken som Gustavsson et al. (2006) presenterar är en bias-textur. Biastexturen innehåller krafter för att få partiklar att stiga och dessa krafter används bara i simuleringen för partiklar som är nyfödda, det vill säga, under en kort tid från det att partiklarna blir levande och börjar uppdateras. Värden på dessa krafter och hur länge krafterna ska användas måste bestämmas individuellt för varje partikelsystem beroende på dess utformning. De voxlar som befinner sig ovanför nuvurande partikeln vid uppdatering ger en ganska stor dragningskraft, de som ligger i samma nivå ger en lite mindre dragningskraft och de som befinner sig under ger en ännu mindre dragningskraft. Detta för att partiklarna som sagt ska stiga uppåt. Ekvationen som används vid uppdatering av partiklar som är nyfödda visas nedan.

$$\mathbf{a}_i = \frac{\sum \mathbf{s}_j \times f(\rho_i) \times bias(\mathbf{s}_j)}{\sum f(\rho_i)}$$

Ekvation 3. Uppdatering för nyfödda partiklars position

2.3 GPGPU

En CPU brukar vanligtvis utföra de beräkningar som ska göras i en applikation. Det kan vara allt från vanlig addition till mer komplexa beräkningar. Dessa beräkningar är ofta en del av en algoritm som ska utföras, t.ex. en algoritm som används för att få fram en ny hastighet av ett föremål som flyger i luften. En CPU kan utföra de mest komplexa algoritmer utan några som helst problem. Det finns dock ett problem att göra detta på CPU’n, nämligen när beräkningarna som ska göras är väldigt många för ett tidssteg och kravet på att det ska utföras i realtid finns. Detta är ett problem som bland annat uppstår i stora partikelsystem där beräkningar är krävande. Ett partikelsystem kan nämligen innehålla miljoner partiklar och att gå igenom och utföra komplexa beräkningar på alla dessa partiklar var för sig kräver hårdvara med mycket hög beräkningskapacitet, något som dagens vanliga CPU’er inte klarar av. Däremot kan man, med rätt implementation, få lite större partikelsystem att simuleras i realtid genom att använda en GPU istället för en CPU.

Att använda en GPU för att utföra beräkningar som i vanliga fall utförs av en CPU kallas för GPGPU. Fördelen med att använda en GPU är att dess arkitektur är väldigt parallell. Detta eftersom den har ett antal programmerbara processorer som kan utnyttjas med hjälp av så kallade shaders. I detta arbete används en slags shader, nämligen en fragment shader. Fragment shaders exekveras på fragment processorer som manipulerar pixlar. Ordet pixel betyder bildelement och är det minsta synliga element som kan visas på en skärm (Shreiner, Woo, Neider & Davis, 2005). En pixel innehåller färginformation. Denna information specificeras ofta i RGB eller RGBA. R står för Red, G står för Green, B står för Blue och A står för Alpha. Alpha är extra information som bestämmer en pixels genomskinlighet. Dessa värden blandas och bestämmer därmed färgen på pixeln som visas på skärmen.

I ett GPGPU-program kan man använda dessa pixlar i kombination med t.ex. fragment shaders, men istället för att specificera vanliga färgvärden för pixlarna så använder man annan data som ska manipuleras på något sätt. Det som är viktigt att tänka på när man väljer algoritmen som ska implementeras i shadern är att pixlarna som ska manipuleras har så lågt beroende mellan varandra som möjligt. Detta eftersom pixlarna manipuleras individuellt och parallellt varje gång som shadern exekveras.

Exemplet nedan åt vänster visar en loop i programmeringsspråket C++ som går igenom en 2D-array med storleken WxH där varje element i arrayen innehåller fyra värden. Vad som händer är att datan i de första och de sista elementen byter plats plats med varandra. Bytet av plats för datan sker på en CPU. Exemplet åt höger visar samma loop men är implementerat i en fragment shader, så att bytet av plats för datan sker på fragment processorer. Exekveringen kommer i detta exempel att utföras snabbare då bytena av datan i elementen sker parallellt. I just detta fall har en så kallad textur-rektangel av storleken WxH använts och varje pixel innehåller RGBA.

```
for(int i = 0; i < W; i++) {
   for(int j = 0; j < H; j++) {
      lastElement = 2D[i][j][3];
      2D[i][j][3] = 2D[i][j][0];
      2D[i][j][0] = lastElement;
   }
}
```

```
float4 change(float2 coords : TEXCOORD0,
              uniform samplerRECT val)
              : COLOR

float4 t = texRECT(val, coords);
float4 res = t;
res.x = t.w;
res.w = t.x;
return res;
}
```

Exempel 1. Byte av data i element på CPU'n (vänster) och GPU'n (höger).

2.3.1 Grunder för en simpel GPGPU-applikation

För att göra en så enkel GPGPU-applikation som möjligt finns det några koncept som ska tas hänsyn till. Dessa koncept följer nedan och är beskrivna i en tutorial utav Harris (2004).

- **Textur = arrray**. Texturer i GPGPU ses på samma sätt som arrays gör i applikationer med standardberäkning (Harris, 2004). Det innebär att man allokerar en textur som har samma dimensioner och data, som den array som innehåller datan man vill göra något med.
- **Shader = beräkningskärna**. En shader kan ses som en liten beräkningskärna som parallellt är applicerad på många fragment samtidigt (Harris, 2004).

- **1:1 pixel till texel mappning**. Man behöver 1:1 mappning från pixlar till texlar för att vara säker på att varje element i texturen blir bearbetade. För att lyckas med detta sätter man viewporten i applikationen till samma dimensioner som texturen (Harris, 2004). En viewport används för att bestämma var någonstans i fönstret rendering ska ske. Dess storlek bestäms i pixlar och får därmed en direkt koppling till fönsterstorleken.

Figur 7: En scen renderas i en viewport samma dimensioner som fönstret (vänster). Samma scen renderas i en viewport med minde dimensioner (höger).

När viewporten har ställts in gör man en ortografisk projektion. Man bestämmer här projektionsmatrisen (Harris, 2004). När man gör en ortografisk projektion specificerar man exakt var olika klipp-plan i en scen ska befinna sig. Det i scenen som befinner sig innanför dessa klipp-plan renderas medan det som befinner sig utanför inte renderas. Detta gör 1:1 pixel till texel mappning väldigt enkelt då man kan se till att bara texturen syns i viewporten.

- **Fyrkant med samma dimensioner som viewporten = dataströmgenererare** För att kunna exekvera fragment shaders måste pixlarna som ska manipuleras genereras. Genom att rita en fyrkant som har samma dimensioner som viewporten så genereras ett fragment för varje pixel. Varje fragment bearbetas sedan identiskt i fragment shadern (Harris, 2004).

- **Kopiera till textur (eng: Copy To Texture, CTT) = feedback**. När fragment shadern är färdig med beräkningarna av fragmenten, som genererades utav fyrkanten med samma storlek som viewporten, befinner sig resultaten i framebuffern. För att spara resultaten kopierar man dessa till en textur som sedan kan användas för exempelvis utritning eller ytterligare beräkningar (Harris, 2004).

3 Problem

Problemet med detta arbete är att utveckla och utvärdera en teknik för simulering av rök med partikelsystem, så att simulering sker på både en CPU och en GPU. Simuleringen som görs på en GPU ska utföras snabbare än simuleringen som görs på en CPU. Röken ska uppföra sig liknande verklig rök, det vill säga, röken ska gå från att vara smal längst ner till att bli tjockare högre upp ju längre simuleringen pågår. Efter ett tag ska röken sjunka. Ett exempel på rök som beter sig på ett sådant sätt, kan ses i figuren nedan.

Figur 8. En del av ett foto som visar stigande rök från stor brand (Nerval, 2006). (Fotot får användas fritt enligt fotografen)

Att utveckla och utvärdera en teknik för simulering av rök med ett partikelsystem som simuleras snabbare på en GPU än en CPU är viktigt att undersöka då det bidrar till att fler partiklar kan användas i partikelsystemen som simuleras på GPU'n. En annan anledning att utveckla och utvärdera tekniken är att man i realtidsapplikationer som t.ex. dataspel, vid simulering av röken på GPU'n, istället kan utnyttja CPU'n för att utföra andra avancerade beräkningar såsom AI.

Tekniken som utvecklas och utvärderas bygger på den teknik som Gustavsson et al. (2006) presenterar i sitt arbete. För att problemet ska kunna lösas delas det upp i två delmål måste uppnås.

3.1 Delmål 1: Utveckling

Det första delmålet som måste uppnås för att problemet ska lösas är utveckling som består av ett mindre och ett större steg. Det första, mindre steget, är att utveckla tekniken för röksimuleringen som Gustavsson et al. (2006) har arbetat fram, så att alla beräkningar sker på CPU'n. Detta steg är viktigt att göra först då det underlättar för nästa steg. Det andra steget är att utveckla tekniken så att beräkningarna för att uppdatera partiklars position utförs på GPU'n.

3.2 Delmål 2: Utvärdering

Det andra delmålet som måste uppnås är att utvärdera den utvecklade tekniken. Det innebär att olika sorters scenarier genomförs. Dessa scenarier används för att kunna bestämma ifall hela problemet har lösts eller om den utvecklade tekniken på något sätt behöver omarbetas. De scenarier som genomförs ska undersöka följande egenskaper hos den utvecklade tekniken:

- Tidseffektivitet: Den tid det tar att simulera den utvecklade tekniken, både på CPU'n och på GPU'n.
- Visuellt resultat: Hur den utvecklade tekniken ser ut vid rendering. Detta för att kunna se hur röken uppför sig.

3.3 Begränsningar

Det finns inget krav på att visualiseringen av röksimuleringen, alltså själva renderingen av röken, ska se så realistisk ut som möjligt. För att få röken i partikelsystemet att se ut som verklig rök, skulle olika renderingsalgoritmer behöva undersökas och utvärderas. En sådan noggrann undersökning och utvärdering är inget som görs i detta arbete då detta inte är en del av problemet.

4 Metod

Detta avsnitt beskriver några metoder som kan användas för att uppnå de två delmålen som presenterades i förra avsnittet. Först beskrivs två metoder för Delmål 1: Utveckling. Därefter beskrivs två metoder för Delmål 2: Utvärdering. I slutet av kapitlet beskrivs de metoder som har valts.

4.1 Metoder för Delmål 1: Utveckling

En metod som kan användas för att uveckla tekniken är **implementation**. För att kunna implementera partikelsystem så att uppdateringar sker på både CPU'n och GPU'n behöver det specificeras vilka olika datatyper och liknande som kommer att användas. För CPU-implementationen innebär detta att först och främst bestämma storlekar på olika arrays. För GPU-implementationen är det viktigt att specificera de olika slags texturtyper som ska användas för bra kompabilitet med shadern och grafikbiblioteket. När specificering är gjord kommer partikelsystemet att implementeras. Tack vare att specificeringen först görs blir det betydligt lättare att göra implementationen, då inget nytt behöver undersökas under denna fas.

Ytterligare en metod som kan användas för att utveckla tekniken är **intervju**. Här kan t.ex. speldesigners intervjuas för att på så sätt få information om vad som är viktigast att ha i åtanke för att utveckla partikelsystemet. Intervjuerna borde då bl.a. innefatta de olika parametrar som speldesigners vill ha för att styra partikelsystemets simulation och liknande.

4.2 Metoder för Delmål 2: Utvärdering

En metod som kan användas för utvärdering efter färdig utveckling är **experiment**. Ett antal experiment som ska genomföras specificeras då och resultaten utav dessa experiment presenteras i olika tabeller och screenshots.

Ytterligare en metod som kan användas för utvärdering är att jämföra den utvecklade tekniken med en **matematisk modell**. Exempel på en sådan matematisk modell som kan användas är Navier-stokes ekvationer. Dessa ekvationer kan användas för att simulera rök, så en jämförelse och analys mellan den utvecklade tekniken och dessa ekvationer kan vara intressant att göra.

4.3 Valda metoder

Utifrån de metoder som presenterats har följande metoder valts för problemet:

För Delmål 1: Utveckling har metoden **implementation** valts. Delmålet är att utveckla en teknik som ska simulera partikelsystem på både en CPU och en GPU. För att överhuvudtaget kunna göra detta måste tekniken implementeras. Att använda både implementation och intervju tillsammans för att utveckla teknikerna är en möjlighet, men i detta arbete är det inte relevant då det som intervju tillför inte är en del av problemet.

För Delmål 2: Utvärdering har metoden **experiment** valts. Att denna metod valts beror på att experiment är en väldigt passande metod för att undersöka framförallt tidseffektivitet men även det visuella. Experiment är också lätt att göra på något som implementerats. Att använda en matematisk modell för detta delmål är inte lämpligt eftersom tidseffektivitet baserat på simulationer på CPU'n och GPU'n inte kan undersökas.

5 Genomförande

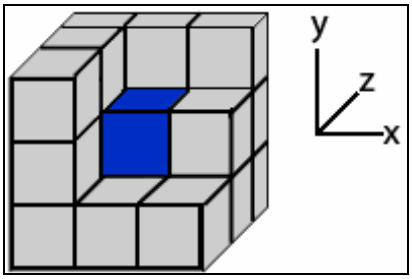
Detta avsnitt beskriver hur metoderna som presenterades har genomförts. Implementationerna görs i Visual Studio 2005 Professional och språket som används är C++. Grafikbiblioteket OpenGL och shaderspråket Cg används. Implementationen för det partikelsystemet som simuleras på GPU'n är i första hand gjord för att fungera tillsammans med grafikkortet Nvidia Geforce 6600GT.

5.1 Implementation, CPU

Partikelsystemet som simuleras på CPU'n innehåller 7st arrayer. Dessa är grid, partiklar, sigmoid, bias, sampleX, sampleY och sampleZ.

- **Grid:** Denna array representerar 3D-texturen som innehåller voxlar med olika densitet och lagrar data av typen int. Ett förutbestämt antal voxlar används som väggar för att partiklar inte ska hamna utanför 3D-griden under exekvering. Vid initiering av partikelsystemet fylls arrayen med värden baserat på den densitet som varje voxel initialt har. De voxlar som används som väggar får värdet 255 medan resterande voxlar får värdet 0, då inga partiklar vid initieringen lever. Ett exempel med en 3D-grid av storleken 3*3*3 och en vägg av storleken 1 går att se i Figur 9. En vägg med storleken **n** innebär alltså att **n** st. voxlar inåt från varje kant blir väggar. Figur 10 visar hur arrayen för denna 3D-grid ser ut i applikationen. Hur 3D-griden överförs till en array beskrivs med koden i Figur 11.

  I grundimplementationen har 3D-griden storleken 30*30*30 med en väggstorlek på 4. Arrayen får då en storlek på 27000.



Figur 9: 3D-griden med storleken 3*3*3 och väggstorleken 1. De gråa voxlarna är väggar.

{ 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 0, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255 }

Figur 10: Arrayen som representerar 3D-griden.

```
int gW,gH,gD = 3; int w = 1;
for(int x = 0; x < gW; x++) {
  for(int y = 0; y < gH; y++) {
    for(int z = 0; z < gD; z++) {
      int density = 0;
      if(x<w||x>=gW-w||y<w||y>=gH-w||z<w||z>=gD-w)
        density = 255;
      Grid[x*gH*gD + y*gD + z]=density;
    }
  }
}
```

Figur 11: Kod som visar hur arrayen skapas.

- **Partiklar:** Denna array innehåller alla de partiklar som finns i partikelsystemet. En partikel representeras av en struct som innehåller position i x,y och z, livstid (hur länge partikeln har levt/varit tillgänglig för uppdatering) och en bool med namnet alive som sätts till true när partikeln lever och därmed kan uppdateras. Vid initiering av partikelsystemet så skapas det antal partiklar som maximalt kan finnas i systemet och alive sätts till false. Varje partikel får också en slumpmässig startposition i ett litet område längst ner i mitten av 3D-griden.

```
typedef struct
{
  float x;
  float y;
  float z;
  int livstid;
  bool alive;
}Partikel;
```

Figur 12: struct som representerar en partikel

- **Sigmoid:** Denna array innehåller de y-värden (attraktionsvärden) som fås utav den sigmoidkurva som tidigare presenterats. För att få rätt utdata används arrayens olika index som indata för x-värden (densitet). Storleken på arrayen är 255 och lagrar data av typen float.

- **Bias:** Denna array representerar den bias-textur som används för att partiklar som är nyfödda ska få en extra kraft och stiga uppåt under en viss tid. Arrayen är av storleken 46 och varje element innehåller ett utav värdena 0.4, 0.8 eller 1.3.

- **SampleX, SampleY och SampleZ:** Dessa tre arrays bildar tillsammans de samplings-vektorer som används för att hämta data från rätt voxlar, utifrån voxeln som den partikel som uppdateras befinner sig i. SampleX[n], SampleY[n] och SampleZ[n] bildar en vektor. Datan i dessa arrays är av typen int och storleken för varje array är 46 (varje array innehåller alltså 46st element var). Detta eftersom multisamplings-tekniken som Gustavsson et al. (2006) presenterar hämtar data från 46st olika voxlar.

  När data ska hämtas från en voxel som t.ex. har samma y och z-värde som den nuvarande voxeln, men med ett x-värde som är 1 mer, är det vektorn innehållandes värdena 1,0,0 som används.

5.1.1 Uppdatering av partiklar

Varje ny uppdatering sätts ett antal förubestämda partiklar till alive (levande). Dessa partiklar kan därmed uppdateras och få nya positioner. Uppdateringen för varje enskild partikel ser ut som följande:

1. Minska livstiden med 1
2. Gå igenom alla 46 voxlar i en loop. I denna loop hämtas densiteten för nuvarande voxel och används sedan för att beräkna attraktionen. Därefter adderas partikelns position med attraktionen multiplicerat med samplevektorn. Om partikelns livstid är mindre än 50 (den har levt mindre än 50 uppdateringar) så multipliceras resultatet även med biasen för samplevektorn.
3. När loopen har gått igenom alla 46 voxlar så sätts partikelns nya position och densiteten för griden ändras om partikeln rör sig till en ny voxel. Därefter är det nästa levande partikels tur.

5.2 Implementation, GPU

Partikelsystemet som simuleras på GPU'n använder sig vid initiering utav samma arrays som skapades för partikelsystemet som simuleras på CPU'n. Dessa arrays överförs till texturer av en bestämd storlek genom koden nedanför.

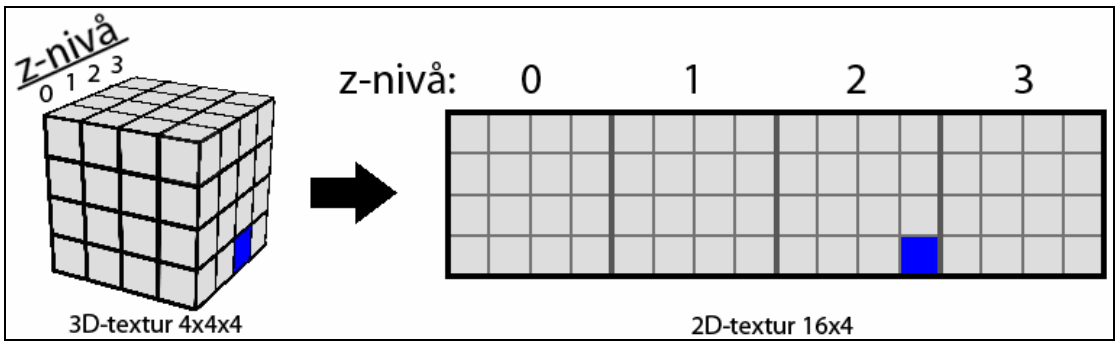
```
glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, 0, 0, width, height,
      FORMAT, TYPE, array);
```

Figur 13: Överförande från array till textur. (FORMAT är antalet färgkomponenter som finns i texturen, t.ex. RGBA. TYPE är datatypen, t.ex. GL_FLOAT)

Alla texturer som används är 2D-texturer av typen GL_TEXTURE_RECTANGLE_ARB. Att dessa typer används istället för GL_TEXTURE_2D beror på att texturkoordinaterna i en ”texturrektangel” inte är normaliserade, det vill säga, texturkoordinaterna går inte från 0-1 i bredd och höjd utan istället från 0-texturbredd och 0-texturhöjd. Detta medför att det blir lättare att hämta data från andra texturer då extra beräkningar för att hitta rätt koordinater inte behöver utföras. Sigmoid-arrayen och bias-arrayen överförs till 2D-texturer med höjden 1, dom betraktas alltså som 1D-texturer. Antal färgkomponenter som varje pixel har i dessa texturer är en, nämligen R. Det medför att dessa texturer får exakt samma dimensioner och antal element som motsvarande arrayer i CPU-implementationen.

5.2.1 Grid-texturen

3D-griden med storleken 30x30x30 delas upp i varje z-nivå och görs om till en 2D-textur med höjden 30 och bredden 30x30. Figuren nedan visar hur mappning från en 3D-textur till en 2D-textur görs i implementationen.



Figur 14. Mappning från 3D-textur till 2D-textur.

För att få data från rätt voxel i shadern, t.ex. voxeln med positionen [3, 2, 3] måste då, för att beräkna korrekt texturkoordinat i x-led, voxelns z-värde multipliceras med antal z-nivåer. Därefter ska detta värde adderas med voxelns x-värde. Kod som visar hur detta görs i shadern kan ses i figuren nedan.

```
texRECT(grid, float2(2*4 + 3, 3));
```

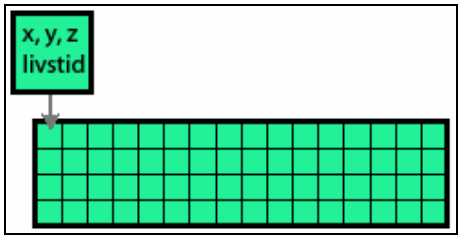
Figur 15: Hämta data från en voxels 3D-position i 2D-texturen.

5.2.2 Partikel-texturen

I CPU-implementationen användes, som tidigare beskrivits, en array innehållandes partiklar som representeras av en struct. I GPU-implementationen används istället en textur där varje pixel i texturen är en partikel. Denna texturs bredd multiplicerat med höjden är det antal partiklar som finns i systemet. En textur med bredden 500 och höjden 500 innebär att partikelsystemet kommer att innehålla 250 000 partiklar. De färgkomponenter som varje pixel har är RGBA:

- R: partikelns x-position
- G: partikelns y-position
- B: partikelns z-position
- A: livstid

Hur detta representeras i en partikeltextur kan ses i figuren nedan. Texturen som visas innehåller 64st partiklar.



Figur 16: Partikeltextur. Varje ruta representerar en partikel/pixel

Det som saknas från structen i CPU-implementationen är alltså boolen som direkt säger ifall partikeln är levande eller inte. För att kunna få med denna datan hade ytterligare en textur behövt användas vilket hade inneburit längre simulering då nya värden hade skrivits till alla pixlar i även den texturen vid varje uppdateringspass.

För att komma runt detta problem med att göra partiklar levande sätts *livstid* vid initiering till ett värde baserat på hur många partiklar som blir levande varje uppdatering. Om 10st partiklar blir levande varje uppdatering så får de 10 första partiklarna *livstid* 0. De nästkommande 10 partiklarna får *livstid* 1 och så vidare. I shadern minskar *livstid* med 1 varje uppdatering och när värdet har blivit 0 blir partikeln levande.

5.2.3 Samplevektor-texturen

Denna textur innehåller de vektorer som används vid multisamplingen. Till skillnad från CPU-implementationen som innehöll tre olika arrays, används här en enda textur av storleken 46x1. Anledningen till att en textur används är att färgkomponenterna i varje pixel bildar vektorn. De färgkomponenter som används i denna textur är RGB:

- R: vektorns relativa x-position
- G: vektorns relativa y-position
- B: vektorns relativa z-position

5.2.4 Uppdatering av partiklar

Innan partikelsystemet kan uppdateras måste alla texturer skapas och få korrekt data. Det måste skapas två partikeltexturer då den ena vid varje uppdatering används för läsning och den andra för rendering (skrivning). Dessa två texturer attachas till en framebuffer som kan användas för offscreen rendering. Det görs via koden i figur 17. ”Attach” innehåller GL_COLOR_ATTACHMENT0_EXT och GL_COLORATTACHMENT1_EXT. ”TexID” innehåller de två texturerna.

```
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, Attach[write],
GL_TEXTURE_RECTANGLE_ARB, TexID[write], 0);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, Attach[read],
GL_TEXTURE_RECTANGLE_ARB, TexID[read], 0);
```

Figur 17: Kod för att attacha texturer till framebuffer

De texturer som inte ändras under simulering, det vill säga samplevektor-texturen, bias-texturen och sigmoid-texturen, skickas till shadern som input. När denna initiering är färdig börjar simuleringen av partikelsystemet.

Varje ny uppdatering sätts 1:1 pixel till texel mappning. Detta är nödvändigt då applikationen ändrar viewporten och projektionen varje gång det visuella ska visas i fönstret. Det framebuffer object (FBO) som innehåller partikeltexturerna ”bindas” även här. Därefter möjliggörs shadern så att beräkningar kan ske. Den buffer som innehåller texturen som ska skrivas till, renderas på en fyrkant med samma dimensioner som viewporten. Partikel-texturen som ska läsas ifrån och grid-texturen som ska läsas ifrån skickas här som input till shadern. När beräkningarna är färdiga stängs shadern av. Hur detta är implementerat går att se i de 3 figurerna på nästa sida.

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);

setOneTOnePixelToTexel();

cgGLEnableProfile(m_fragmentProfile);
glDrawBuffer(attachments[m_writeTex]);

cgGLSetTextureParameter(pInput, TexID[read]);
cgGLEnableTextureParameter(pInput);
cgGLSetTextureParameter(gInput, gridTex);
cgGLEnableTextureParameter(gInput);

drawQuad();


cgGLDisableProfile(m_fragmentProfile);
```

Figur 18: Kod för att beräkningar ska kunna göras i shadern.

```
void setOneToOnePixelToTexel()
{
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluOrtho2D(0.0, texWidth, 0.0, texHeight);
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
      glViewport(0, 0, texWidth, texHeight);
}
```

Figur 19: Funktionen setOneToOnePixelToTexel()

```
void drawQuad()
{
      glPolygonMode(GL_FRONT,GL_FILL);

      glBegin(GL_QUADS);

            glTexCoord2f(0.0, 0.0);
            glVertex2f(0.0, 0.0);

            glTexCoord2f(texWidth, 0.0);
            glVertex2f(texWidth, 0.0);

            glTexCoord2f(texWidth, texHeight);
            glVertex2f(texWidth, texHeight);

            glTexCoord2f(0.0, texHeight);
            glVertex2f(0.0, texHeight);

      glEnd();
}
```

Figur 20: Funktionen drawQuad()

Den fragmentshader som används för att uppdatera alla partiklar i partikelsystemet kan ses i figuren nedan. Indata till shadern är, förutom de olika texturerna, bredden på griden och väggstorleken.

```
float4 particleFragmentShader (
      in float2 coords : TEXCOORD0,
      in uniform float gridWidth,
      in uniform float wall,
      in uniform samplerRECT particles,
      in uniform samplerRECT samples,
      in uniform samplerRECT bias,
      in uniform samplerRECT sigmoid,
      in uniform samplerRECT grid,
      out float4 res) : COLOR0
{
   float4 particle = texRECT(particles, coords);
   float3 pFinal = particle;
   particle.w--;

   if(particle.w < 0)
   {
      int density;
      float attraction;
      int currVox;
      float sumAttraction = 0;
      float3 pSum = 0;
      float3 sampleVox;
      int3 vox;
      for(currVox = 0; currVox < 46; currVox++)
      {
         sampleVox = texRECT(samples, float2(currVox, 0));
         vox = particle + sampleVox;
         density = texRECT(grid, float2(vox.x + vox.z*gridWidth,
            vox.y));
         if(density < 255)
         {
            attraction = texRECT(sigmoid, float2(density, 0));
            sumAttraction += abs(attraction);
            if(particle.w > -50)
               pSum += attraction*sampleVox*texRECT(bias,
                  float2(currVox, 0));
            else
               pSum += attraction*sampleVox;
         }
      }
      pFinal = particle + (pSum/sumAttraction);
      if(pFinal.x < wall || pFinal.x > gridWidth-wall ||
            pFinal.y < wall || pFinal.y > gridWidth-wall ||
            pFinal.z < wall || pFinal.z > gridWidth-wall)
         pFinal = particle;
   }
   res.x = pFinal.x;
   res.y = pFinal.y;
   res.z = pFinal.z;
   res.w = particle.w;
   return res;
}
```

Figur 21: Shaderkoden som används för att uppdatera partiklars position.

När första delen av simuleringen är färdig, alltså när partiklarna har uppdaterats, måste grid-texturen uppdateras så att alla voxlar får rätt densitet. Denna uppdatering sker på CPU'n då detta inte är möjligt att göra på GPU'n med den valda implementationen. Denna uppdatering görs samtidigt som den visuella delen av partikelsystemet renderas i applikationsfönstret. Renderingen är simpel och använder en polygon per partikel.

Som tidigare beskrivits kan den partikeltextur som data skrivs till inte användas för läsning. För att då kunna använda resultaten i ytterliggare simuleringspass måste de två texturerna swappas med varandra. Denna teknik kallas för "ping-pong" och är väldigt simpel. Figuren nedan visar hur funktionen, som utför detta byte mellan vilken textur som ska läsas ifrån och vilken som ska skrivas till, är implementerad.

```
void swap()
{
    write = 1-write;
    read = 1-read;

}
```

Figur 22: Implementation för ping-pong tekniken.

5.3 Experiment

För att uppnå Delmål 2: Utvärdering behövs ett antal experiment göras som sedan ska analyseras. Exakt vilka experiment som har gjorts beskrivs i detta avsnitt. Alla experiment är genomförda på en dator med följande hårdvara och mjukvara:

- Intel Pentim 4 2.80Ghz
- Nvidia Geforce 6600GT 128MB AGP
- Windows XP Professional SP2

5.3.1 Tidseffektivitet

Det har utförts två experiment baserade på tidseffektivitet. Det första experimentet jämför den tidsmässiga skillnaden det tar för det CPU-baserade partikelsystemet och det GPU-baserade att uppdateras, när dessa har samma antal partiklar. Det andra experimentet undersöker max antal partiklar som kan vara levande i partikelsystemen som simuleras på GPU'n, för att de fortfarande ska kunna simuleras i realtid.

- **Experiment 1** använder en 3D-grid med storleken 30x30x30 med en väggstorlek 4. Partikelsystemen testas med 10 000 partiklar, 50 000 partiklar respektive 100 000 partiklar. Experimentet är implementerat på ett sådant sätt så att den tid det tar för 100 uppdateringar, sparas. Antal partiklar som skapas varje uppdatering är olika för de tre partikelsystemen. För partikelsystemet med 10 000 partiklar är antalet 10, för det med 50 000 partiklar är antalet 50 och för det med 100 000 partiklar är antalet 100. Alla tre delar av experimentet exekveras tio gånger för att eliminera utomstående påverkan så gott det går. Experimentet pågår från dess att inga levande partiklar finns till dess att alla partiklar lever. Det är medelvärdet av resultaten av de tio exekveringarna som sedan sammanställs i tre tabeller.
- **Experiment 2** använder en 3D-grid med storleken 30x30x30 med en väggstorlek 4. Implementationen är gjord på ett sådant sätt att de olika partikelsystemen som valts simuleras med max antal levande partiklar i 10 iterationer. Detta för att få ett bra medelvärde på det antal uppdateringar per sekund (Frames Per Second, "FPS") som simuleringen sker i. Storleken på de partikelsystem som undersöks i detta experiment väljs efter en analys av resultaten för Experiment 1. Detta experiment har bara applicerats på den GPU-baserade implementationen. Resultatet sammanställs i en tabell.

5.3.2 Visuellt resultat

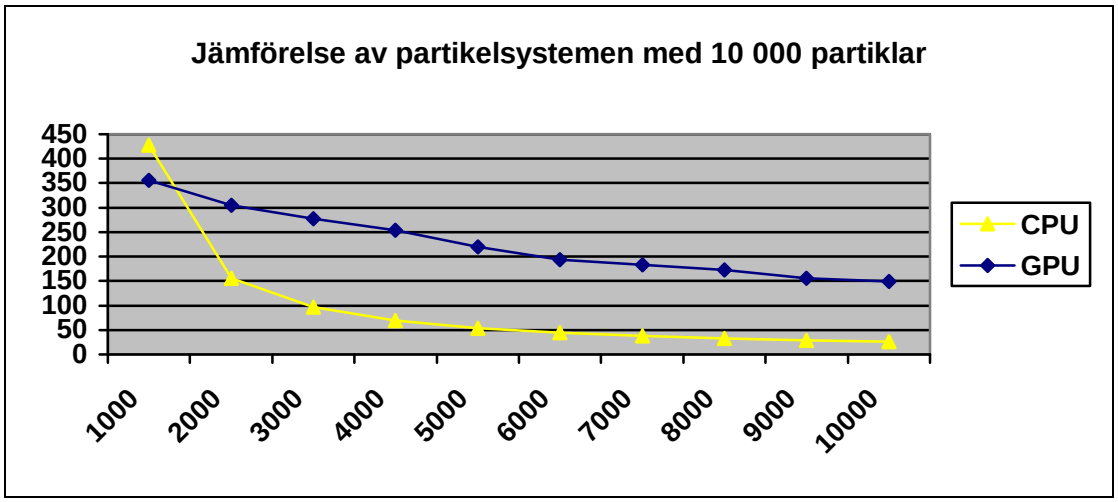
Detta experiment undersöker det visuella resultatet då ett partikelsystem simuleras. Experimentet görs för att kunna undersöka om röken som simuleras uppför sig liknande verklig rök. För detta experiment har en 3D-grid av storleken 30x30x30 med en väggstorlek 4 använts. Varje uppdatering blir fem nya partiklar levande. Resultatet visar hur partikelsystemet ser ut vid olika tidssteg i simuleringen och presenteras i form utav screenshots.

6 Resultat

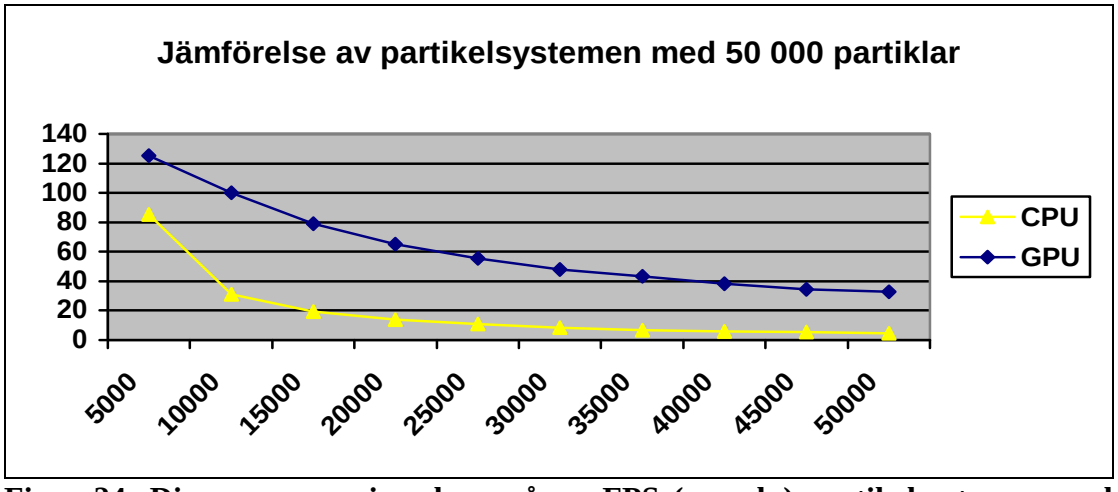
I detta avsnitt presenteras resultatet av detta arbete. Första delen innehåller resultat av det experiment som jämför tidseffektiviteten vid simuleringar av partikelsystemen. Den andra delen innehåller resultatet av experimentet som utförts för att undersöka max antal partiklar som ett partikelsystem kan innehålla, vid realtidssimulering. Den tredje och sista delen innehåller resultat av det experiment som utförts för att undersöka ifall röken som simuleras uppför sig liknande verklig rök.

6.1 Tidseffektivitet, Experiment 1

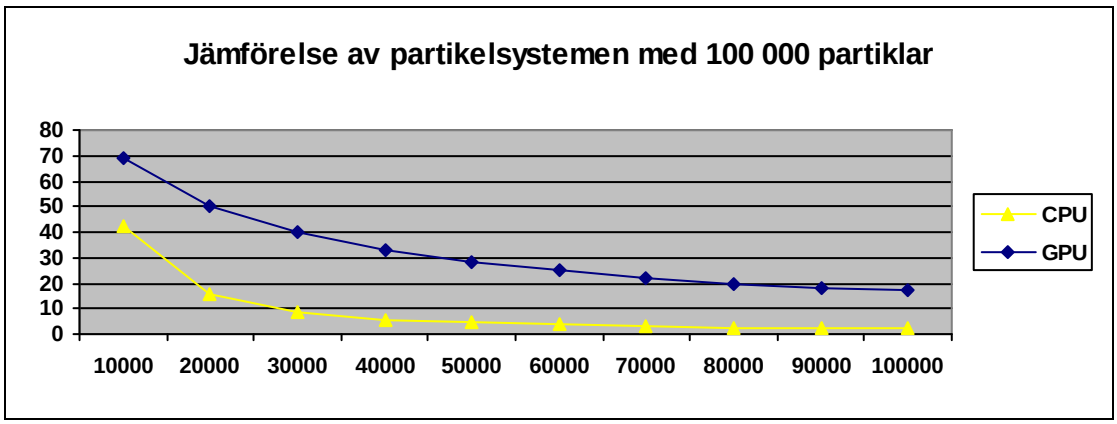
De resultat som har erhållits utav de tre delexperiment, som jämför den tid det tar för partikelsystemen att uppdateras, presenteras här nedanför i tre figurer. Varje figur innehåller ett diagram där man enkelt kan urskilja skillnaderna allteftersom fler och fler partiklar blir levande. Höga värden i avseende på FPS är positivt.



Figur 23: Diagram som visar hur många FPS (y-axeln) partikelsystemen med 10000 partiklar uppdateras i, beroende på antal levande partiklar (x-axeln).



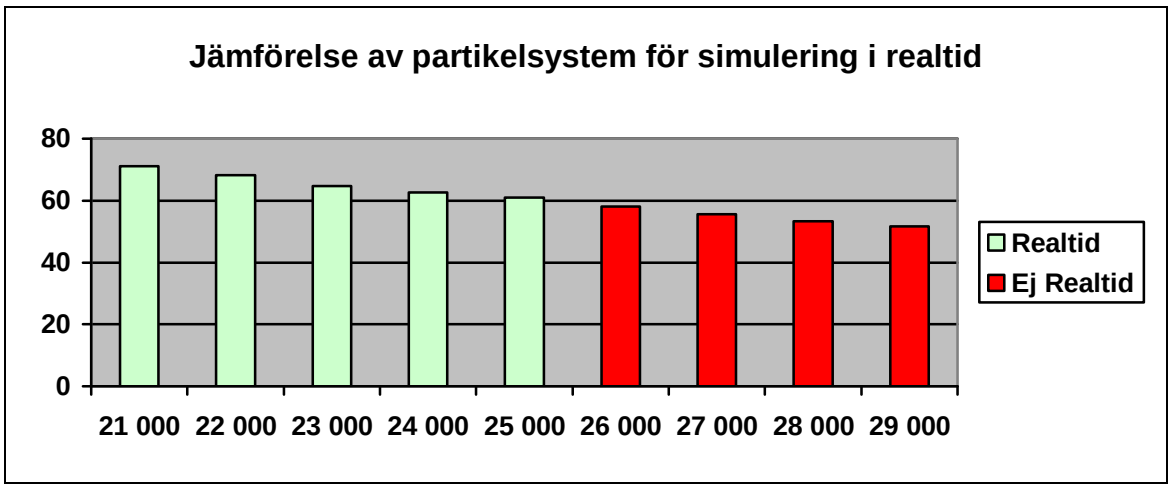
Figur 24: Diagram som visar hur många FPS (y-axeln) partikelsystemen med 50000 partiklar uppdateras i, beroende på antal levande partiklar (x-axeln).



Figur 25: Diagram som visar hur många FPS (y-axeln) partikelsystemen med 100 000 partiklar uppdateras i, beroende på antal levande partiklar (x-axeln).

6.2 Tidseffektivitet, Experiment 2

Det resultat som har erhållits utav experimentet som undersöker hur många partiklar ett partikelsystem max kan innehålla för att simulering ska ske i realtid, presenteras i diagrammet som finns i figur 26. Ett FPS-värde som är 60 eller högre innebär att partikelsystemet simuleras i realtid. I resultatet av Experiment 1 kan man se att partikelsystemet med 50 000 partiklar simuleras i realtid när antal levande partiklar är någonstans mellan 20 000 och 25 000. För detta experiment behöver dock det största valda partikelsystemet som testas innehålla mer än 25 000 partiklar, då en mindre textur används än den textur som användes i partikelsystemet med 50 000 partiklar. Detta eftersom en mindre textur ger snabbare simulationer tack vare att mindre data behöver skickas och hämtas till och från GPU'n vid varje uppdatering. Därför användes i detta experiment nio olika partikelsystem som innehöll mellan 21 000 och 29 000 partiklar.



Figur 26: Diagram som visar hur många FPS (y-axeln) de olika partikelsystemen simuleras i när alla partiklar (x-axeln) lever.

6.3 Visuellt resultat

Nedanför visas det visuella resultat som erhålls när partikelsystemet simuleras på GPU'n. Detta partikelsystem har, som tidigare beskrivits, en grid med storleken 30x30x30 och en väggstorlek 4. Vid varje uppdatering skapas det fem nya partiklar. Inga levande partiklar finns vid simuleringens början.

Uppdateringar = 50

Uppdateringar = 100

Uppdateringar = 150

Uppdateringar = 200

Uppdateringar = 300

Uppdateringar = 400

7 Analys

Genom att analysera resultaten för ”Tidseffektivitet, Experiment 1” ser man att skillnaderna i avseende på FPS, mellan CPU-implementationerna och GPU-implementationerna för varje diagram, nästan är identiska. De partikelsystem som simuleras på GPU’n har generellt sett en mycket högre FPS än de som simuleras på CPU’n.

Det diagram som skiljer sig lite från mängden är det som presenteras i Figur 23. Där ser man att det CPU-implementerade partikelsystemet har en högre FPS i början utav simuleringen. Förklaringen till att detta sker, är att det partikelsystemet som simuleras på GPU’n måste hämta och skicka data från GPU’n till CPU’n och tvärtom vid varje uppdatering, vilket är ganska kostsamt. När det då bara är ett fåtal partiklar som behöver uppdateras, vilket är fallet i början av simuleringen för detta partikelsystem, ges en fördel till partikelsystemet som simuleras på CPU’n.

En analys av resultatet för ”Tidseffektivitet, Experiment 2” visar att antalet FPS som varje partikelsystem simuleras i, reduceras i takt med att partikelsystemen blir större. De partikelsystem som innehåller 21 000 till 25 000 partiklar simuleras i realtid medan de andra fyra partikelsystemen inte simuleras i realtid. Det största partikelsystemet som man kan simulera i realtid på en plattform som har samma hårdvara som testplattformen, är alltså ett partikelsystem innehållandes 25 000 partiklar. Detta resultat bör dock bara ses som en fingervisning för max antal partiklar som kan finnas i ett partikelsystem vid realtidssimulering, då andra faktorer utanför applikationen såsom operativssystems-processer och liknande påverkar hastigheten för simuleringen.

En tolkning av det visuella resultatet visar att rökpartiklarna stiger uppåt till dess att de slår i ovansidan på griden. När denna kollision sker, sprider sig partiklarna utåt mot kanterna så att röken blir tjockare upptill. När partiklarna sedan har fått en tillräckligt hög livstid, sjunker dom. Partiklarna drar sig till områden med låg densitet och åker bort från områden med hög densitet. Det visuella resultatet påminner om verklig rök och liknar den bild (figur 8) som presenterades i problemspeciferingen.

8 Slutsats

Den ena delen av problemet i detta arbete var att utveckla en teknik som simulerar rök med ett partikelsystem på en GPU, snabbare än på en CPU. Den andra delen av problemet var att den utvecklade tekniken som simulerade rök med partikelsystem på GPU'n, skulle uppföra sig liknande verklig rök. För att kunna dra slutsatser i avseende på problemet behövdes två olika delmål uppfyllas. Det första delmålet var att utveckla tekniken för röksimulering så att beräkningarna utfördes på både en CPU och en GPU. Detta gjordes genom att använda metoden implementering. Det andra delmålet var att utvärdera den utvecklade tekniken genom att utföra olika scenarier baserat på tidseffektivitet och visuellt resultat. Metoden som då valdes var experiment.

8.1 Sammanfattning

Efter att ha analyserat resultaten från de olika experimenten kan det fastställas att de partikelsystem som simulerades på GPU'n hade en bättre FPS än de som simulerades på CPU'n. Därför kan slutsatsen göras att den ena delen av problemet i detta arbete, att simulera rök snabbare på en GPU än på en CPU, blev uppfylld.

Det visuella resultatet visar att röken som simuleras på GPU'n, uppför sig liknande verklig rök (röken går från att vara smal längst ner till att bli tjockare högre upp och efter ett tag sjunker röken). Därför kan även slutsatsen dras att den andra delen av problemet, att röken ska uppföra sig likt verklig rök, blev uppfylld.

Det är viktigt att notera att för partikelsystem med väldigt få antal partiklar (mindre än 1000 partiklar), är den utvecklade tekniken som simulerar rök med ett partikelsystem på en CPU att föredra, då tiden det tar att läsa och skicka data från och till en GPU tar för lång tid, i jämförelse med att uppdatera hela partikelsystemet på en CPU. För de lite större partikelsystemen kan slutsatsen dras att den utvecklade tekniken för att simulera rök med partikelsystem på en GPU, är en teknik som uppfyller de kriterier som problembeskrivningen har.

8.2 Diskussion

Istället för att använda en relativt billig teknik för att simulera rök, som detta arbete gör, kan man istället använda tekniker som är mer beräkningskrävande. När det gäller rök kan man då t.ex. använda sig av någon slags teknik som löser Navier-stokes ekvationer. Att använda en sådan beräkningskrävande teknik kan dock, framförallt i dataspel, ge upphov till att spelet inte simuleras i realtid om lite äldre grafikkort används. Det gäller därför, när kravet på realtidssimulering finns, att göra en begränsning för val av teknik baserat på om äldre hårdvara ska stödjas eller inte.

Att använda GPGPU för att lösa ett problem, i detta fall simulering av rök med ett partikelsystem, har visat sig vara framgångsrikt. Just detta arbete har en direktkoppling till datorgrafik. GPGPU kan användas till problem som inte har någon koppling alls till datorgrafik som t.ex. att knäcka lösenord, något som det ryska företaget ElcomSoft (2007) nyligen har gjort. Det finns en relativt nyskapad utvecklingsmiljö med namnet CUDA, som används för att göra just beräkningar på GPU'n. CUDA är utvecklat av NVIDIA och den första publika betan släpptes i februari 2007 (Nvidia, 2007). CUDA går dock bara att använda tillsammans med nyare grafikkort. De äldsta Geforce-korten som kan användas är de i 8000-serien.

8.3 Framtida arbete

Det finns många framtida studier som kan göras baserat på detta arbete. Några förslag på sådana studier följer nedan.

- En framtida studie som kan göras är att undersöka effekterna av att simulera partikelsystemen på plattformar med nyare GPU'er och annan nyare hårdvara, än den hårdvara som använts som testplattform i detta arbete. I det här arbetet användes ett Geforce 6600GT (128MB, AGP), som lanserades år 2004, och eftersom kapaciteten för vad GPU'er klarar av ökar markant för varje år, är detta en framtida studie som är väldigt intressant att göra.
- Resultaten visar att röken inte ser realistisk ut (även fast röken *uppför* sig realistiskt), eftersom varje partikel renderas på ett simpelt sätt med enbart en polygon av en bestämd färg. En framtida studie som här är intressant att göra är att undersöka lämpliga renderingsalgoritmer för partikelsystemet. Hur partikelsystemet ska renderas kan t.ex. baseras på densiteten i varje voxel eller partiklars livstid.
- En annan framtida studie som är intressant att göra är att få hela partikelsystemet att simuleras och renderas på GPU'n utan någon som helst inblandning med CPU'n. Det som då behöver undersökas är alltså hur eliminering av läsning kan göras från GPU'n till CPU'n, varje gång partiklarna ska renderas och griden ska uppdateras.
- Resultaten visar att de partikelsystem som simuleras på GPU'n lämpar sig i applikationer som kräver realtidssimulering, när antalet levande partiklar som max kan finnas i partikelsystemet är 25 000 eller mindre. Detta mått på max antal partiklar är dock bara tillförlitligt när inga andra beräkningar sker i applikationen. Därför är det intressant att implementera och testa denna teknik i några olika dataspel för att undersöka maxstorleken på partikelsystemet när andra faktorer spelar in.

Referenser

AGEIA *PhysX*. http://www.ageia.com/physx/index.html [Hämtad 08.03.14]

ElcomSoft (2007) *ElcomSoft Files Patent for Revolutionary Technique to Recover Lost Passwords Quickly* http://www.elcomsoft.com/EDPR/gpu_en.pdf [Hämtad 08.05.29]

Fedkiw, R., Stam, J. & Jensen, H.W. (2001) *Visual Simulation of Smoke* (s. 15-22). Proceedings of SIGGRAPH ‘01, ACM, 12-17 augusti, 2001, Los Angeles, Kalifornien, USA

Fernando, R. & Kilgard, M.J. (2003) *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley Professional.

Gustavsson, H., Engström, H. & Gustavsson, M. (2006) *A multi-sampling approach for smoke behaviour in real-time graphics.* In the Proceedings of SIGRAD 2006, 22-23 november, Skövde, Sverige

Halixi72 (2007) *Figur 2. Statiskt partikelsystem som simulerar hårstrån.* http://en.wikipedia.org/wiki/Image:Strand_Emitter.jpg [Hämtad 08.03.19]

Harris, M.J. (2004) *Tutorial 0: ”Hello GPGPU”*. http://www.gpgpu.org/developer/ [Hämtad 08.03.19]

Jtsiomb (2007) *Figur 1. Partikelsystem som simulerar eld.* http://en.wikipedia.org/wiki/Image:Particle_sys_fire.jpg [Hämtad 08.03.19]

Nerval (2006) *Figur 8. En del av ett foto som visar stigande rök från stor brand.* http://en.wikipedia.org/wiki/Image:Wildfiretopanga.jpg [Hämtad 08.03.18]

NVIDIA (2007) *CUDA for GPU Computing* http://news.developer.nvidia.com/2007/02/cuda_for_gpu_co.html [Hämtad 08.05.30]

Reeves, W.T. (1983) *Particle Systems-a Technique for Modeling a Class of Fuzzy Objects* (s. 91-108). ACM Transactions on Graphics (TOG), Volume 2, Issue 2 (April 1983), ACM.

Rockstar North (2005) *GTA San Andreas.*, Rockstar Games, 2005, PC-versionen http://www.rockstargames.com/sanandreas/

Selle, A. Rasmussen, N. & Fedkiw, R. (2005) *A vortex particle method for smoke, water and explosions* (s 910-914). Proceedings of ACM SIGGRAPH 2005, ACM, 2005, Los Angeles, Kalifornien, USA

Shreiner, D., Woo, M., Neider J. & Davis, T. (2005) *OpenGL Programming Guide Fifth Edition.* Addison-Wesley Professional.

Wikipedia (2008) *Navier-Stokes equations.* http://en.wikipedia.org/wiki/Navier-Stokes_equations [Hämtad 08.03.18]

Wikipedia (2008) *Particle system.* http://en.wikipedia.org/wiki/Particle_system [Hämtad 08.03.19]