LINKÖPINGS UNIVERSITET

Modelleringsprojekt 2013

Gruppmedlemmar:
Simon BERGSTRÖM
Pär ERIKSSON
Oscar IVARSSON
Jimmie BERG
Robin BERNTSSON

Examinator:
Dr. Anna LOMBARDI

13 mars 2013

Sammanfattning

Innehåll

Figurer

Tabeller

Kapitel 1

Inledning

1.1 Syfte

Målet med detta projekt var att åstadkomma en applikation som simulerar rök. Vissa krav fastställdes: simuleringen skall ske i tre dimensioner, den skall vara interaktiv och renderas i realtid. Ett sådant projekt delas naturligt upp i två delar, varav det ena ligger något utanför kursens krav. Den första delen avser simulering av ett fysiskt grundat fenomen, vilken denna kurs erfordrar. Den andra omfattar den rendering som måste äga rum för att transformera siffror till rök, eller åtminstone till något som ser ut som rök.

Vi valde att arbeta med båda problem parallellt. Det ansågs ofrånkomligt att vissa personer håller sig på sin respektive kant, men vi ville minimera detta i största möjliga mån. Det var ett mål att alla partner skulle ingå i- och vara en del av alla moment i projektet.

Vid simulering av flöden förekommer två olika tillvägagångssätt. Antingen generaliseras flödet till ett antal partiklar med tillhörande fysiska egenskaper (position, hastighet et cetera), eller så definieras flödets densitet och hastighet i separata rutnät i lämplig dimension. Då dessa metoder hyser inneboende (och ofrånkomliga) för- och nackdelar används vanligen en hybrid vid kostsamma produktioner, där det bästa från båda världar kan utnyttjas. För vårt syfte var det endast nödvändigt att använda en metod.

1.2 Generell metod

I och med att syftet för detta projekt var att åstadkomma en utseendemässigt korrekt simulering av rök var vi tvungna att ta ställning till vilket tillvägagångssätt som var lämpligt för att angripa problemet. Utgångspunkten för all flödessimulering är densamma, nämligen lösning av Navier-Stokes satser (2.1). Skillnaden mellan de föregående metoderna är nämnvärda och valet därav påverkar resultatet mycket.

$$\begin{cases} \frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla \rho = \vec{g} + \upsilon \nabla \cdot \nabla \vec{u} \\ \nabla \cdot \vec{u} = 0 \end{cases} \tag{1.1}$$

Vi valde såkallad Eulerian-method. Vid detta tillvägagångssätt kvantiseras en kontinuerlig yta eller rymd till ett antal rutor eller kuber, beroende aktuell dimension. Dessa rutor eller kuber kallas voxels, och varje enskild sådan innehåller information som avser flödets densitet och hastighet i aktuellt område. Dessa värden används sedan för att representera röken vid eventuell rendering. För det absolut enklaste fallet av rendering representeras densitetsvärdet av en färg som fyller motsvarande voxel.

1.3 Verktyg

1.3.1 Program

- **Matlab** - Användes till att skapa en första simulering av tvådimensionell rök.
- **Codeblocks** - Programmeringseditor som tillämpades för den slutgiltiga implementationen.

1.3.2 Språk/Bibliotek

- **C++** - Det huvudsakliga programmeringsspråket.
- **OpenGL** - Grafiken ritades upp i ett OpenGL-fönster.
- **GLSL** - Renderingen gjordes på grafikkortet med OpenGL Shading Languague.
- **OpenCL** - Parallelprogrammering på grafikkortet och som utfördes i fluidlösaren skrevs i OpenCL.
- **AntTweakBar** - Bibliotek som användes till att skapa GUI:t.
- **Latex** - Typsättningssytem för rapportskrivning.

Kapitel 2

Flödessimulering

2.1 Särskild metod

För att alstra naturtrogen rök krävs ytterligare termer än de som förekommer i Navier-Stokes ekvationer. När det gäller att implementera rökspecifika termer finns det många erkända processer att välja bland, alla med varierande svårighet. Avvägningen som får göras är den mellan komplexitet och gott resultat. Huvudsakligen skall ett gott resultat uppnås utan arbetsbelastning som ligger utanför kursens förväntningar.
Utöver den absolut simplaste flödesberäknaren möjlig valde vi att implementera lyftkraft och temperatur, vilka är mycket lämpliga för denna kurs då de har rötter i den fysiska världen, och gör även att resultatet blir finare.

2.2 Antaganden

Eftersom att syftet med detta projekt var att skapa en naturligt rotad simulering var vi skyldiga att städse anknyta varje steg i den matematiska processen till motsvarande verkligt förekommande fenomen. Således var vi tvungna att göra vissa antaganden innan vi skred till verket.

Vi var tvungna att definiera vilka parametrar som kan anses vara föränderliga. Dessa ansågs vara rökens utbredningshastighet, viskositet (motstånd), och initialvärden (hastighet, densitet, temperatur, position). Relevanta och konstanta koncept innefattas av Navier-Stokes satser, gravitation och utväxling av temperatur. Dessa beror till viss del på de parametrar nämnda i föregående stycke.

2.3 Approximationer

Egentligen är även omgivande luft en gas som borde hanteras på vederbörligt vis, men enligt etablerade metoder är det ibland enklare att generalisera denna till ett stillastående vakuum med blott en parameter för temperatur och tumma på knapparna tills flödet beter sig som om den hade omgivits av luft.

För en estetiskt tilltalande simulering är det viktigt att låta röken vara omsluten av ogenomträngliga väggar så den kan studsa runt och interagera med sig själv. Om vår modell skall vara fysiskt korrekt får rummet ej läcka, all massa måste bevaras. Varken energi eller massa kan

egentligen upphöra, men detta krav var vi emellertid tvungna att bryta då numeriska lösningar och iterativa approximationer aldrig blir exakta i detta sammanhang, vilket leder till att flödets totala massa minskar för varje uträkning. Det finns masskonserverande processer i vår lösning, men det kommer alltid att förekomma ett visst läckage.

Även våra termer för temperatur och lyftkraft strider till viss del mot fysiska begrepp. Temperaturutväxling sker, precis som man kan förvänta sig, men inte i noggrann samklang med vad fysikens lagar föreskriver. Det samma gäller för tyngd- och lyftkraft, de finns gott och väl med i beräkningarna, men endast mycket approximativt.

Ett sista förbehåll till simulatorns fysiska riktighet är avsaknaden av SI-enheter. Det är möjligt att normalisera alla parametrar till verkliga storheter i ett sådant här projekt, men vi fattade beslutet att detta vore mycket jobb för lite utväxling. Alltså lät vi alla variabler vara enhetslösa, med undantag för temperatur som faktiskt angivits i Kelvin. De interval som ansågs vara lämpliga för de olika parametervärdena fastslogs genom ”trial-and-error”.

2.4 Matematiska metoder

Det tillvägagångssätt som används bygger på så kallad splittring[2] en komplicerad ekvation delas up i sina komponentdelar och löses var för sig.

$$\frac{\mathrm{d}q}{\mathrm{d}t} = f(q) + g(q) \tag{2.1}$$

$$\begin{cases} \widetilde{q} = F(\Delta t, q^n) \\ q^{n+1} = G(\Delta t, \widetilde{q}) \end{cases} \tag{2.2}$$

Detta är fortfarande en algoritm av första ordningens noggrannhet[2]. Motivet till att göra detta är att integrations metoderna F() och G() kan väljas oberoende av varandra vilket tillåter att den bästa lösningsmetoden används för varje del av ekvationen.

Ekvation (2.1) delas upp i:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = 0 \tag{2.3}$$

$$\frac{\partial \vec{u}}{\partial t} = \vec{g} \tag{2.4}$$

$$\frac{\partial \vec{u}}{\partial t} = \upsilon \nabla \cdot \nabla \vec{u} \tag{2.5}$$

$$\begin{cases} \frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho} \nabla \, \rho = 0 \\ \nabla \cdot \vec{u} = 0 \end{cases} \tag{2.6}$$

Sedan löses varje ekvaton för sig med hjälp av resultatet från föregående ekvation.

I två av funktionerna kommer en Poission ekvation $\nabla^2 \vec{q} = \vec{b}$ behövas lösas. På diskret form är detta ett linjärt system $\mathbf{A}\vec{x} = \vec{b}$ där $\mathbf{A}$ är den diskreta motsvarigheten till Laplaceoperatorn ∇^2 . Detta kan löses genom att ta inversen av $\mathbf{A}$ men då matriserna till största delen innehåller nollor är det onödigt beräkningstungt. Istället skriver vi om det enligt (2.7).

$$x_{i,j,k}^{(n+1)} = \frac{x_{i+1,j,k}^{(n)} + x_{i-1,j,k}^{(n)} + x_{i,j+1,k}^{(n)} + x_{i,j-1,k}^{(n)} + x_{i,j,k+1}^{(n)} + x_{i,j,k-1}^{(n)} + \alpha \cdot b_{i,j,k}}{\beta} \tag{2.7}$$

2.5 Projektion

Projektions steget är det steg som garanterar att det resulterande hastighetsfältet $\vec{u}^{n+1}$ är divergensfritt. Ekvation (2.6) innehåller två okända p och $\vec{u}^{n+1}$, detta löses genom att diskretisera (2.6) vilket resulterar i:

$$\vec{u}^{n+1} = \vec{u}^{n} - \frac{dt}{\rho}\nabla p \tag{2.8}$$

Multiplikation med ∇ på båda led av 2.8 i kombination med divergensvillkoret leder till:

$$\nabla \cdot \vec{u}^{n} = \frac{dt}{\rho}\nabla^2 p \tag{2.9}$$

Ekvation (2.9) är en Poission ekvation och kan lösas genom att använda(2.7) med $\alpha = \frac{dx^2 \cdot \rho}{dt}$ och $\beta = 4$. Det resulterande tryckfältet sätts in i (2.9) och det nya divergensfrita hastighetsfältet räknas ut.

2.6 Advektion

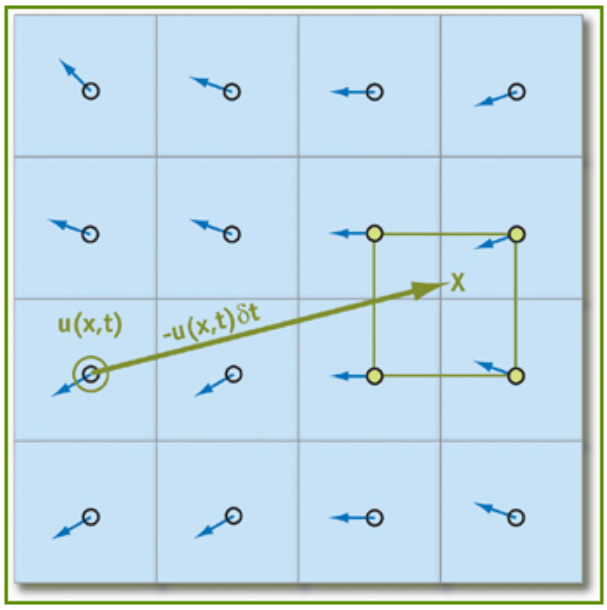
I advektionssteget används en semi Langerisk metod där varje cell förflyttar sig i hastighetetsfältet som om den vore en tänkt partikel. Dom nya värdena för (2.3) skulle kunna räknas ut med Euler's explicita metod.

$$\vec{r}(t + dt) = \vec{r}(t) + dt \cdot \vec{u} \tag{2.10}$$

Detta ger dock uphov till vissa problem och för stora dt leder det till att systemet blir instabilt. Det är då bättre att använda sig av en implicit metod [3].

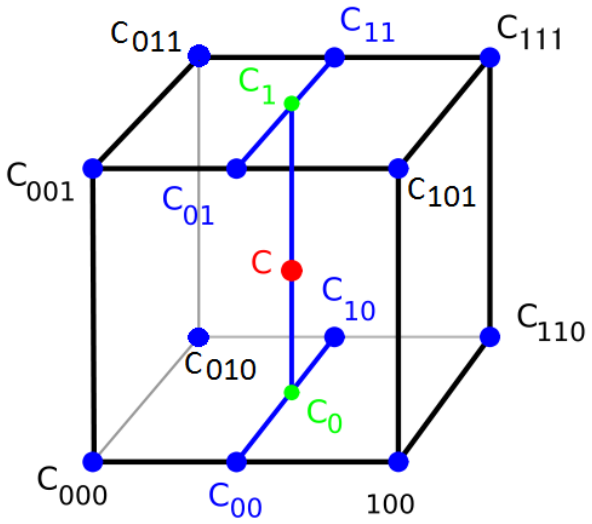
$$q(\vec{x}, t + dt) = q(\vec{x} - \vec{u}(\vec{x}, t), t) \tag{2.11}$$

Snarare än att förutse vilken position en partikel lär hamna efter ett steg i tidsled görs det omvända och positionsutvecklingen följs baklänges; processen bestämmer vilken partikel som hade blivit positionerad på aktuell plats från ett steg i bakåt i tiden.



Figur 2.1: Visualisering av advection i två dimensioner. [4]

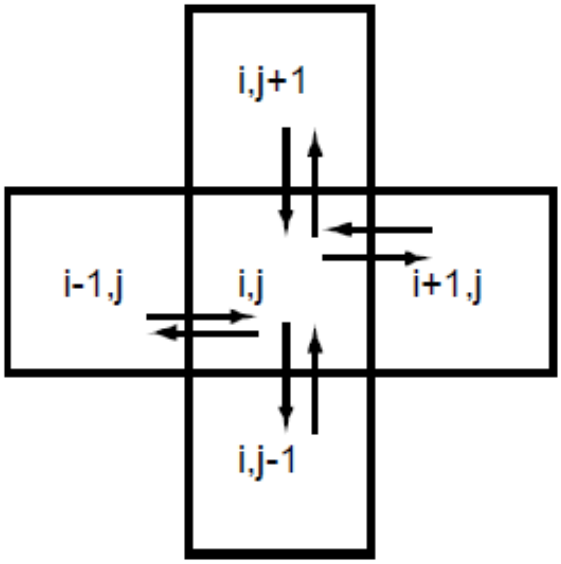
För att få värdet i den erhållna punkten utförs en trilinjär interpolation mellan de åtta omkringliggande punkterna.



Figur 2.2: Interpolation i tre dimensioner.

2.7 Diffusion

Diffusions ekvationen (2.5) beskriveren utjämning av voxelvärdena. Konstanten ν beskriver hur snabbt denna utjämning sker.



Figur 2.3: Diskret visualision av diffusion i två dimensioner

Samma metod som användes för advektion används även här. Ekvation (2.5) diskretiseras och skrivs på implicit form

$$(\mathbf{I} - \nu dt \nabla^2)\vec{u}(\vec{x}, t + dt) = \vec{u}(\vec{x}, t) \tag{2.12}$$

Där **I** är enhetsmatrisen. Detta är en Poission ekvation för hastigheten och kan skriva på samma form som 2.7 med $\alpha = \frac{dx^2}{\nu dt}$ och $\beta = 4 + \alpha$.

2.8 Gränsvilkor

Alla differentialekvationer definierade på ett slutet område kräver villkor på begränsningsytan för att ha en unik lösning. Då inget rök ska kunna lämna arbetsytan så måste vissa villkor vara uppfyllda vid de yttersta voxlarna.

1. Flödets hastighet relativt gränsytan måste vara noll $\vec{n} \cdot \vec{u} = 0$.
2. Tryckförändringen måste vara noll i gränsytans normalrikting, d.v.s. $\frac{\partial p}{\partial \vec{n}} = 0$.

2.9 Lyftkraft

Ett flöde som simulerats med hjälp av de termer som redan är nämnda kan anses vara ”obestämt”. Det finns ingen fysikalisk motsvarighet i vår omgivning, för att göra specifika gaser eller vätskor måste flera termer adderas. Av dessa fanns flera kandidater för röksimulering. Vi valde att lägga till blott en term, vilken konstituerar lyftkraft, något som på egen hand gör att flödet går från att vara obestämd till rökliknande.

Denna term simulerar det faktum att rökens temperatur och densitet skiljer sig från den omgivande luften. Pågrund av gravitation kommer röken stiga med avtagande hastighet vartefter

röken skingras. För att simulera detta definierades ett fält med samma struktur som vårt densitetsfält, fast för temperatur. Enligt samma processer breder temperatur ut sig i rummet, men detta är inget som syns direkt, utan spelar snarare en roll i kalkylering av den lyftkraft som verkar på flödet.

Lyftkraft approximerades som en funktion av temperatur och densitet i en viss punkt, och denna adderades till motsvarande ”uppåt-komponent” i vårt hastighetsfält.

$$F = (-\kappa d + c(T - T_0)) \tag{2.13}$$

dimensionslösa konstanterna κ och c skalar effekten av temperatur och densitetsskillnader mellan flödet och luften[1]. Dessa bestämdes också mer eller mindre genom ”trial-and-error”. Konstanten k skalar densitetens inverkan på lyftkraft och borde därför vara definierad som någon slags invers av kg/m3, men i och med att densiteten i sig inte är angiven som en fysisk storhet hade även k blivit arbiträr i någon mening. Temperatur angavs dock enligt Kelvin-skalan där den omgivande luften är rumstempererad medan röken hundra grader varm.

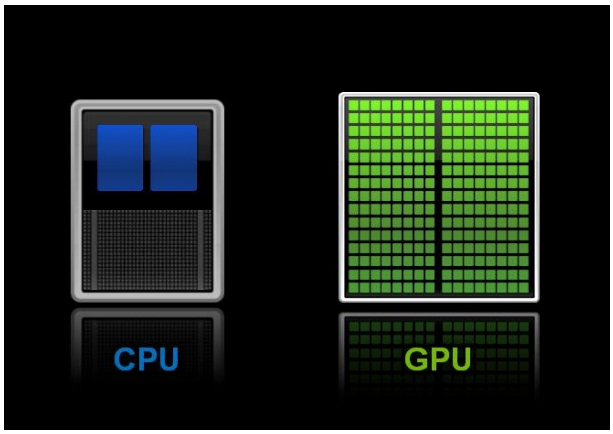
Kapitel 3

Implementering

3.1 Matlab

Den första versionens flödessimulering skrevs och implementerades i Matlab. Resultatet renderades som ett tvådimensionellt flöde där varje voxel i 2D-planet innehöll en enda intensitet. Koden och kunskapen som erhölls därifrån tillämpades senare vid övergång till C++ och tre dimensioner.

3.2 Parallelprogrammering



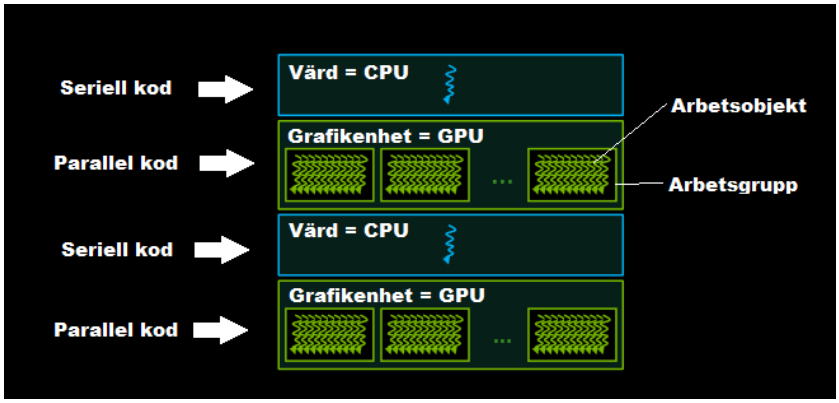
Figur 3.1: Skillnaden mellan CPU och GPU [6]

Figur 3.1 illustrerar datorns processor (CPU) samt grafikenhet (GPU). CPUn består av ett förhållandevis litet antal kärnor medan GPUn består av flera hundra. För att kunna hantera mängden beräkningar som flödesberäknaren kräver och samtidigt rendera röken i realtid var vi tvungna att tillämpa parallelprogrammering på grafikkortet. Detta möjliggör tusentals beräkningar vilka utföras samtidigt snarare än att de utförs i en seriell följetong. Detta minskar behandlingstiden och bidrar till fler bilder per sekund.

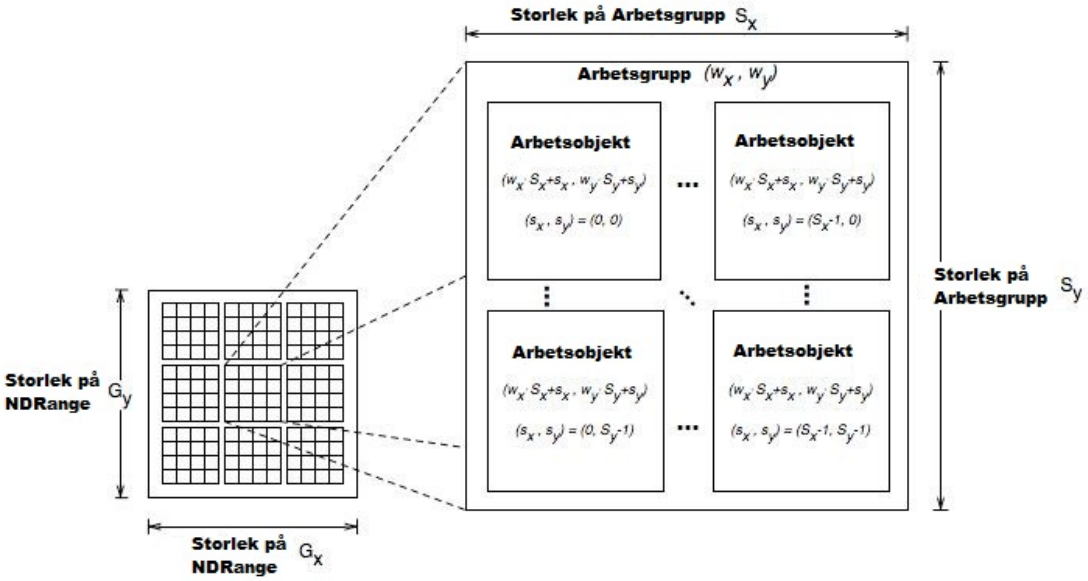
[1]Resultatet kan ses på `http://www.youtube.com/watch?v=NHDkpWNJS-g`

3.2.1 OpenCL

OpenCL (Open Computing Language) är ett plattformsoberoende ramverk som gör parallellprogrammering möjligt. Detta tillämpades i applikationen. Ramverkets struktur är består av två separata delar. Den ena utgör en CPU-baserad värd som skapar och allokerar minnesobjekt samt hanterar kommunikation till grafikenheten. Den andra delen består av såkallade kernels. En grafisk beskrivning kan ses i figur 3.2 Dessa är funktioner skrivna i språket OpenCL C och som körs på grafikenheten, dessa användes för att dirigera beräkningar till GPU som annars hade blivit utförda på CPU.



Figur 3.2: Arbetssekvens för parallelprogrammering i OpenCL [6]



Figur 3.3: Tvådimensionell NDRange [7]

När en kernel körs skapas ett heltalsindex med en storlek som motsvarar antalet beräkningar som ska genomföras. Det här indexet kallas NDRange [5] (figur 3.5) och är uppdelad i arbetsgrupper som i sin tur består av arbetsobjekt. Arbetsobjekten genomför en beräkning var, alltså körs kernelfunktionen en gång för varje sådant objekt och allt detta görs samtidigt.

3.3 Flödesberäknaren

Varje gång flödessimuleringen itereras tas ett tidssteg vardera för densitet, hastighet samt temperatur. Ett sådant tidssteg kallar sedan på flödesberäknarens olika delar (Kodexempel 3.1-5) och exekverar dessa i korrekt ordning. Kodexempel 3.3 skiljer sig från övriga då denna funktion körs som kernel på grafikkortet.

Kodexempel 3.1: Advektion

```
\label{kod1}
Advektion
{
    // Storlek på steg
    dt0 = tidssteg * gridstorlek;

    for ( Varje voxel i gridet )
    {
        // Hitta Voxel-koordinater bakåt i tiden
        [i0, j0, k0] = [i, j, k] - dt0 * voxelVärden(i,j,k);

        if ( Voxel är vid gräns )
        {
            // Flytta in [i0, j0, k0] innaför gränserna.
        }

        // Nya voxelVärden interpoleras från närliggande värden
        voxelVärden(i,j,k) = gamlaVoxelVärden(i,j,k);
    }
}
```

Kodexempel 3.2: Projektion

```
Projektion
{
    for ( Varje voxel i gridet )
    {

        Pa = voxelVärden_Hastighet_u(i+1,j,k);
        Pb = voxelVärden_Hastighet_u(i-1,j,k);
        Pc = voxelVärden_Hastighet_v(i,j+1,k);
        Pd = voxelVärden_Hastighet_v(i,j-1,k);
        Pe = voxelVärden_Hastighet_w(i,j,k+1);
        Pf = voxelVärden_Hastighet_w(i,j,k-1);

        gamlaVoxelVärden_Hastighet_v(i,j,k) = - 0.5 *
            (Pa - Pb + Pc - Pd + Pe - Pf) / gridstorlek;
         gamlaVoxelVärden_Hastighet_u(i,j,k) = 0;
    }

    Poissonlösare();

    for ( Varje voxel i gridet )
    {
        Pa = gamlaVoxelVärden_Hastighet_u(i+1,j,k);
        Pb = gamlaVoxelVärden_Hastighet_u(i-1,j,k);
```

```
        Pc = gamlaVoxelVärden_Hastighet_u(i,j+1,k);
        Pd = gamlaVoxelVärden_Hastighet_u(i,j-1,k);
        Pe = gamlaVoxelVärden_Hastighet_u(i,j,k+1);
        Pf = gamlaVoxelVärden_Hastighet_u(i,j,k-1);

        voxelVärden_Hastighet_u(i,j,k) -= 0.5 *
            gridstorlek * (Pa - Pb);
        voxelVärden_Hastighet_v(i,j,k) -= 0.5 *
            gridstorlek * (Pa - Pb);
        voxelVärden_Hastighet_w(i,j,k) -= 0.5 *
            gridstorlek * (Pa - Pb);
    }
}
```

Kodexempel 3.3: Poissonlösare

```
Possionlösare (kernelfunktion) // Löser systemet linjärt
{

    // Hämtar rätt index ur NDRange
    int i = get_global_id(0) + 1;
    int j = get_global_id(1) + 1;
    int k = get_global_id(2) + 1;

    for ( Jacobiiterationer )
    {
        if ( i,j,k befinner sig innanför gridet )
        {
            Pa = voxelVärden(i+1,j,k);
            Pb = voxelVärden(i-1,j,k);
            Pc = voxelVärden(i,j+1,k);
            Pd = voxelVärden(i,j-1,k);
            Pe = voxelVärden(i,j,k+1);
            Pf = voxelVärden(i,j,k-1);

            voxelVärden = gamlaVoxelVärden(i,j,k)
                + (Pa + Pb + Pc + Pd + Pe + Pf) / 6;
        }
    }
}
```

Kodexempel 3.4: Lyftkraft

```
Lyftkraft
{
    for ( Varje voxel i gridet )
    {
        voxelVärden_Hastighet_v(i,j,k) += rökFaktor *
            voxelVärden_Densitet(i,j,k) -
            temperaturFaktor *
            (voxelVärden_Temperatur(i,j,k) - luftTemperatur);
    }
}
```

Kodexempel 3.5: Gränser

```
Gränser // För ett par sidor av kuben
{

    for ( gridstorlek + 1 )
    {
        for ( gridstorlek + 1 )
        {
            // Kolla om de voxelvärden som ligger vid den kanten
            // påverkas utav hastighet i motsatt riktning
            if ( Voxelvärden påverkas )
            {
                voxelVärden(0,j,k) = - voxelVärden(1,j,k);
                voxelVärden(gridstorlek+1,j,k) =
                    - voxelVärden(gridstorlek+1,j,k);
            }
            else
            {
                voxelVärden(0,j,k) = voxelVärden(1,j,k);
                voxelVärden(gridstorlek+1,j,k) =
                    voxelVärden(gridstorlek+1,j,k);
            }
        }
    }

    // Beräkna nya värden för alla hörnen, ett exempel
    voxelVärden(0,0,0) = (voxelVärden(1,0,0) + voxelVärden(1,0,0) +
    voxelVärden(1,0,0) + voxelVärden(1,0,0)) / 3;
}
```

3.4 Volymrendering i realtid

För att visualisera genomskinliga volymer används en metod vid namn volymrendering. Denna renderingsmetod gör det möjligt att visualisera såväl yta som insida av en fylld tredimensionell modell. Vanligen delar man upp detta begrepp i två områden: texturbaserad volymrendering eller ”ray casting” (med brist på svensk motsvarighet).

Den senare anses vara modern och effektiv, vilket gjorde att denna valdes som slutlig renderingsmetod för projektets rökmoln.

3.4.1 Ray Casting

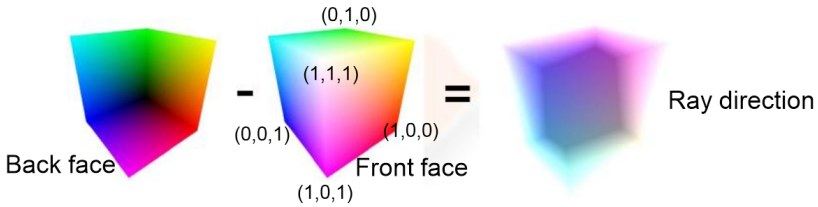
Tekniken innebär att för varje pixel generera en stråle med utgångspunkt i simulationens betraktningsposition. Strålen penetrerar en kubisk volym vars storlek och position faller i linje med rökens densitetsvolym. Här interpoleras värden utifrån 3D-data (i vårt fall densitetsvärden) och dess position i volymen. Pixelns slutgiltiga värde blir summan av dessa interpolationer.

Algoritmen kan sammanfattningsvis beskrivas med följande schema[9]:

1. Framsidan av en tänkt begränsningsyta renderas ut till en textur för att representera startpunkterna.

2. Baksidan av samma begränsningsyta renderas ut till en annan textur för att representera stoppunkterna. Avståndet och riktningen kan enkelt räknas ut genom skillnaderna mellan dessa punkter (figur 3.4).

3. För varje startpunkt genereras en stråle som penetrerar volymen längs riktningsvektorn tills strålen har lämnat volymen.



Figur 3.4: Generering av strålens riktning [10]

Kodexempel 3.6: Tvådimensionell texturhantering

```
\label{texturHantering}
GLuint textureHandle;
glGenTextures(1, &textureHandle);
glBindTexture(GL_TEXTURE_2D, textureHandle);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, width, height);
```

Texturhantering i OpenGL fungerar genom att ett ”handtag” för texturen alstras och en särskild textur blir bunden till sitt handtag (se kodexempel 3.6). De texturdata som behövs vid rendering kan sedan hämtas ut från detta handtag, som i sin tur förmedlar lämplig textur.

Två texturer som representerar start och slutpunkter genereras, i det första passet bortses den främre ytan och i det andra passet den bakre ytan. Densiteten binds således till en 3D-textur och uppdateras parallellt med flödesberäknaren och informationen interpoleras (se kodexempel 3.7).

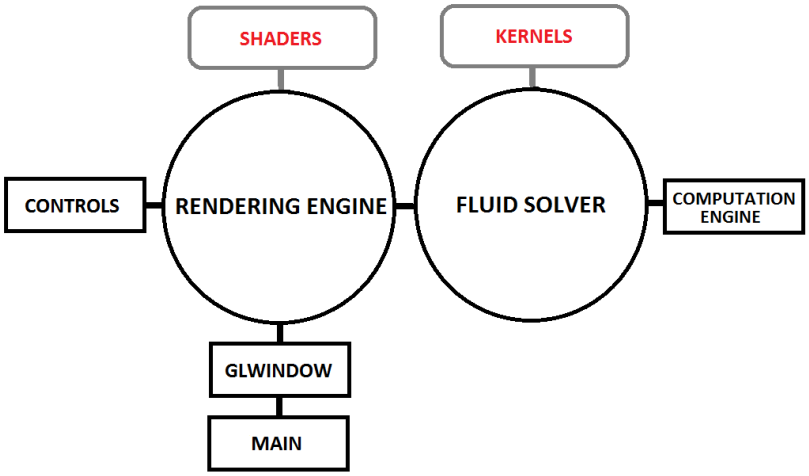
Kodexempel 3.7: Interpolering i fragment shader

```
while(stråle befinner sig i volymen)
{
    // Räkna ut densiteten utifrån 3D-data och strålens position
    float densitet = texture(rökdata, position).x;

    // Addera densiteten till en färgvektor
    Färgvektor += densitet;
}
```

3.5 Kodstruktur

Applikationen byggdes upp enligt figur 3.5. Vid exekvering skapas initialt ett fönster i GLWindow-klassen där även huvudloopen ligger. Applikationen byggdes därefter upp genom två huvudklasser där större delen av funktionaliteten ligger. Rendering Engine sköter rendering av rökmolnet samt användargränssnittet. Denna kommunicerar även med de shaders som finns samt den andra huvudklassen som i denna rapport refereras till som flödesberäknaren. Denna klass behandlar tillsammans med kernelfunktioner alla beräkningar på flödet.



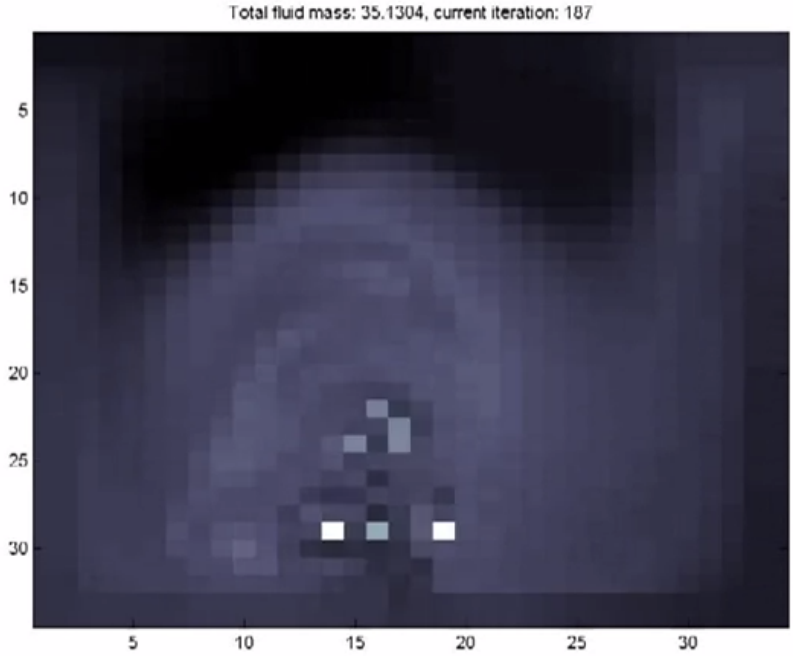
Figur 3.5: Tvådimensionell NDRange

Kapitel 4

Resultat

4.1 Förstudier i MATLAB

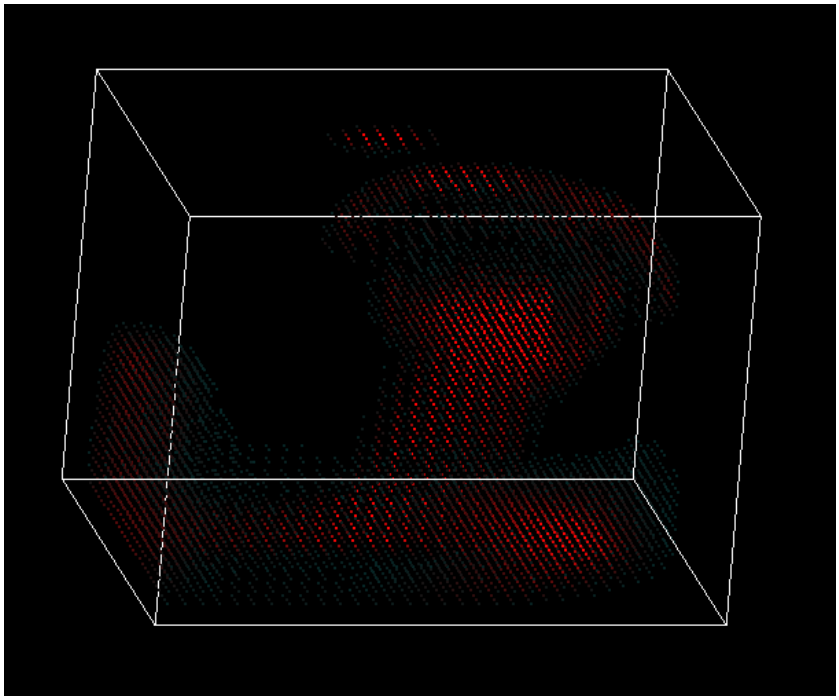
En tvådimensionell flödesberäknare i MATLAB togs fram, figur 4.1, den simulerade densitetens utbredning.



Figur 4.1: 16x16 rutor

4.2 Implementering av Flödesberäknaren i C++ och OpenGL

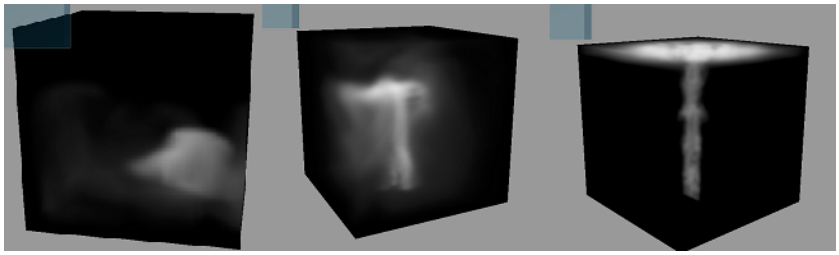
Den tidigare flödesberäknaren utvidgades till tre dimensioner, och visualiserades utav OpenGL's inbyggda funktion för att rendera punkter, figur 4.2.



Figur 4.2: 32x32x32 voxlar

4.3 Rendering med OpenGL och GLSL

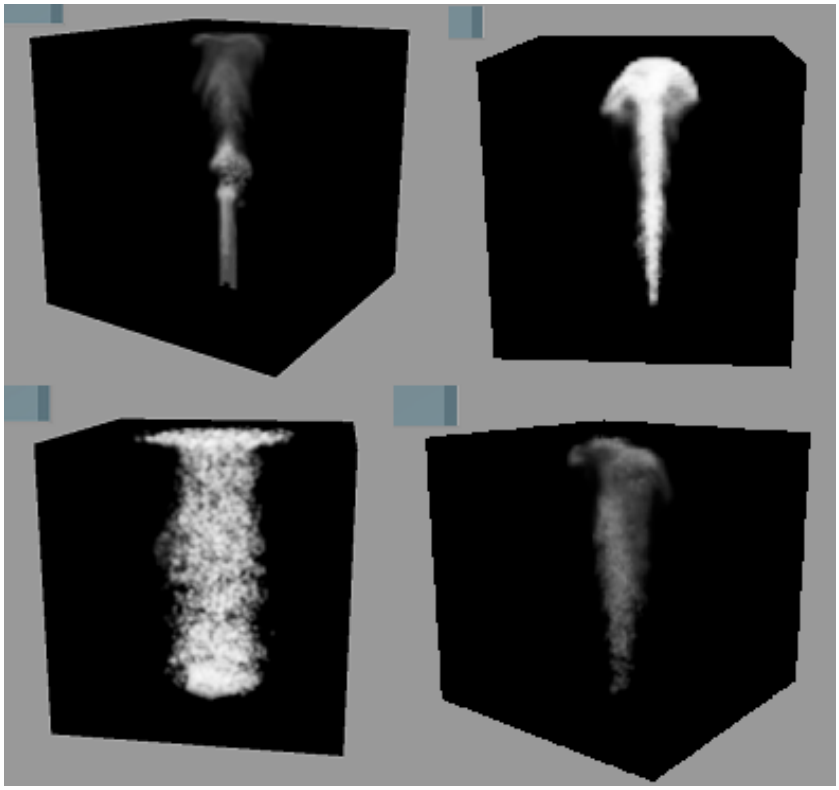
Samma flödesberäknare som tidigare användes och volymrenderingsmetoden ray casting implementerades, simuleringen renderades i realtid. Resultatet kan ses i figur 4.3.



Figur 4.3: 32x32x32 voxlar

4.4 Resultat med OpenCL

Genom integration av OpenCL i flödesberäknaren kunde fler bilder per sekund i samma upplösning framkallas. Upplösningen kunde således ökas och fortfarande köras i realtid enligt figur 4.4 och tabell 4.1.



Figur 4.4: 64x64x64 voxlar

Tabell 4.1: Bilder per sekund vid exekvering

	Utan OpenCL	Med OpenCL
32x32x32 voxlar	16 FPS	34 FPS
64x64x64 voxlar	2 FPS	8 FPS

Kapitel 5

Diskussion

Om arbetet skulle fortskrida finns ett antal områden att förbättra. Det största området är att förbättra röken i sig och få den mer realistisk. Förutom att implementera bättre interpolation (Catmull-Rom), som resulterar i mer exakta värden, kan en mer avancerad metod för att spara värden i gridet användas. MAC staggered grid är en sådan metod som sparar hastighetsvärden på sidorna istället för i voxelns centrum. Den undviker i och med det vissa problem som kan uppstå vid nuvarande metod.

Med en mer exakt interpolation kan denna användas vid fler tillfällen utan att felaktiga värden skulle uppstå. Då hade en Runge-Kuttametod kunnat tillämpas vid advektionen istället för nuvarande implicita Eulermetod vilket skulle generera en mer exakt metod att lösa stegen i tidsled. MacCormacks schema är också en välkänd metod som skapar större noggrannhet genom att gå två steg i advektionen, ett prediktions- och ett korrigeringssteg.

En viktig parameter att inkludera vid simulering av realistisk rök är vorticitet. Denna parameter är definierad som rotationen av en strömmande fluids momentana hastighetsvektor i varje punkt [8]. Förutom att den inte är helt trivial att implementera så är den beräkningstung.

Objektkollision är ett väldokumenterat område som förmodligen skulle kunna implementeras relativt enkelt. Ett sätt är att skapa två texturer där den ena innehåller information om var objektet befinner sig och den andra hur objektet påverkar hastigheten vid kollision med flödet.

Tabell 4.1 visar att parallelprogrammering på grafikkortet kan förbättra prestandan avsevärt. Trots att en relativt liten del av flödesberäknaren exekveras parallellt på grafikkortet bidrar den till att röken kan simuleras med en högre upplösning i realtid. Detta ger en mer realistisk rök med fler och synligare detaljer. Vid detta område finns många områden som kan optimeras och förbättras. Dels skulle en större del av flödesberäknaren kunna köras på grafikkortet genom vidare implementation men även förbättringar i minneshantering och lagring utav data skulle förmodligen medföra fler bilder i sekunden vid en viss upplösning. Ett exempel kan vara att använda grafikkortets separata texturcache för att snabbare läsa och skriva till minnet.

Detta projekt gav upphov till en applikation som simulerar rök. Rök är ett fysiskt förekommande fenomen, men uppfattningen av hur pass fysiskt simuleringen är får ofta subjektiv. Det är svårt att objektivt fastställa hur pass ”bra” en rök är, särskilt i detta fall. Vanligen kan en simulering jämföras med mätdata som hämtats från den faktiska verklighet som simuleringen vill avbilda; i detta fall har många approximationer gjorts och således finns det inga data att

jämföra med experimentell data.

Att studera bromssträckan för en bilsimulator med bromssträckan för motsvarande bil i verkligheten leder till en objektiv slutsats om simulatorns förmåga. Vårt fall skiljer sig från detta genom att det inte finns något naturligt sätt att jämföra med verkligheten. Detta beror delvis på att röken inte är fastställd som någon särskild, det skulle kunna vara såväl vattenånga som brandrök. Dessutom är det i slutändan viktigast att röken är vacker för människan, vilket är högst subjektivt. Slutsatser om simuleringens kvalitet får alltså ligga i betraktarens öga.

Källförteckning

[1] Fernando R *GPU Gems* 2004: Addison-Wesley Professional.

[2] Bridson R *Fluid Simulation for Computer Graphics* 2008: A K Peters/CRC Press.

[3] Stam J *Real-Time Fluid Dynamics for Games* 2003, pages 1-17.

[4] Randima F *GPU Gems* 2004, available at `http://http.developer.nvidia.com/GPUGems/elementLinks/fig38-03.jpgl`.

[5] Verigakis N *Eulerian Smoke Simulation on the GPU MSc Computer Animation and Visual Effects Master Thesis* 2011: N.C.C.A Bournemouth University.

[6] Woolley C *Introduction to OpenCL* 2010: NVIDIA CORPORATION, Developer Technology Group, available at `http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/06-intro_to_opencl.pdf`.

[7] Trevett N, Zeller C *OpenCL on the GPU* 2009: NVIDIA CORPORATION, available at `http://www.nvidia.com/content/GTC/documents/1409_GTC09.pdf`.

[8] Fernando R *Vorticitet*: Nationalencyklopedin, available at `http://www.ne.se/lang/vorticitet`.

[9] Hadwiger M *GPU-Based Ray Casting* SIGGRAPH 2009 .

[10] Venkataraman S *4D Volume Rendering NVIDIA CORPORATION 2009* `http://www.nvidia.com/content/GTC/documents/1102_GTC09.pdf`.