**LINKÖPINGS UNIVERSITET** **14 mars 2014**
**Institutionen för teknik och naturvetenskap**
**TNM085 - Modelleringsprojekt**

# Elements

## GPU-baserad fluidsimulering i realtid

**Av**
Niklas Andersson
Gabriel Baravdish
Joakim Deborg
Kristofer Janukiewicz
Marcus Nygren

**Handledare**
Ulf Sannemo

**Examinator**
Anna Lombardi

**Sammanfattning**

I den här rapporten presenteras en Eulerisk implementation av Navier-Stokes ekvationer för att simulera rök, och i förlängningen även andra fluider. Beräkningarna görs i 3D på datorns GPU, vilket möjliggör rendering av fluiden i realtid. Rapporten i sig kan användas som ett stöd inför utförandet av en egen fluidsimulering eller som ett exempel på en enkel redogörelse av Navier-Stokes ekvationer.

# Innehåll

# 1 Introduktion

Elements är en programvara som utför fluidsimuleringar baserade på Navier-Stokes ekvationer i realtid på GPU. Programmet visar i dagsläget endast rök, men programmet är utformat för att i framtiden kunna byggas ut med fler simuleringar som exempelvis eld och vatten. Visionen med Elements är att användaren ska kunna utforska Navier-Stokes ekvationer genom att i realtid kunna experimentera med parametervärdena.

# 2 Modellering

## 2.1 Förklaring av Navier-Stokes ekvationer

Navier-Stokes ekvation, ekvation 1, och sambandet om konserverandet av massa, ekvation 2, refereras i allmänhet till Navier-Stokes ekvationer. Dessa modellerar tillsammans en fluid genom att se på fluiden som en kontinuerlig substans, det vill säga; att istället för att enskilda molekyler studeras så studers små volymelement av fluiden som kan innehålla flera molekyler. Ekvationerna i sig bygger på konservering av massa, rörelsemängd och energi och härstammar från ett antal volym- och yt-integraler över en godtycklig volym; se stycket om konservering av energi under sektion 2.2 som beskriver Reynolds Transport Theorem (RTT), ekvation 3. För en mer utförlig förklaring och djupare analys av Navier-Stokes ekvationer se referens [1].

I följande ekvationer är $\mathbf{u}$ hastigheten

$$\mathbf{u}(x, y, z, t) = \begin{pmatrix} x'(t) \\ y'(t) \\ z'(t) \end{pmatrix},$$

där $\mathbf{u}$ indirekt beror på tiden $t$. Variabeln $p$ är medelvärdet av det normala trycket i $x$-, $y$- och $z$-led över ett volymelement, $\mathbf{\Gamma}$ är en generell deriverbar spänningstensor och variabeln $\mathbf{f}$ står för yttre påverkande krafter.

$$\varrho\left(\frac{\partial}{\partial t}\mathbf{u} + \mathbf{u}\cdot\nabla\mathbf{u}\right) = -\nabla p + \nabla\cdot\mathbf{\Gamma} + \mathbf{f} \tag{1}$$

$$\nabla\cdot\mathbf{u} = 0 \tag{2}$$

## 2.2 Konservering av energi

RTT är en balansekvation som säger att förändringen av en fysisk egenskap $\psi$ (antingen skalär- eller vektor-baserad) definierad på volymen $\Omega$ ska vara lika med skillnaden mellan vad som är förändrat genom randen till $\Omega$ och vad som är förändrat inuti $\Omega$ genom eventuella källor och sänkor $\chi$. Detta i enlighet med ekvation 3 och kan tolkas som ett sätt att bevara energi.

$$
\begin{aligned}
\frac{\partial}{\partial t}\int_\Omega \psi \,\mathrm{d}V &= \int_\Omega \chi \,\mathrm{d}V - \int_{\partial\Omega} \psi\mathbf{u}\cdot\mathbf{n}\,\mathrm{d}A \quad \Leftrightarrow \\
\frac{\partial}{\partial t}\int_\Omega \psi \,\mathrm{d}V &= \int_\Omega \chi \,\mathrm{d}V - \int_\Omega \nabla\cdot(\psi\mathbf{u})\,\mathrm{d}V \quad \Leftrightarrow \\
\int_\Omega \frac{\partial}{\partial t}\psi \,\mathrm{d}V &+ \int_\Omega \nabla\cdot(\psi\mathbf{u})\,\mathrm{d}V - \int_\Omega \chi \,\mathrm{d}V = 0 \quad \Leftrightarrow \\
\int_\Omega \left(\frac{\partial}{\partial t}\psi \; &+ \nabla\cdot(\psi\mathbf{u}) - \chi\right)\mathrm{d}V = 0
\end{aligned}
\tag{3}
$$

Eftersom integralen i ekvation 3 ska vara lika med noll för alla volymer $\Omega$ erhålls ekvation 4 som blir det slutgiltiga sambandet för konservering av energi. Detta samband nyttjas i framtagandet av Navier-Stokes ekvationer genom att applicera det på massa respektive rörelsemängd.

$$
\frac{\partial}{\partial t}\psi \; + \nabla\cdot(\psi\mathbf{u}) - \chi = 0 \tag{4}
$$

## 2.3 Konservering av massa

Enligt RTT blir $\psi = \varrho$ vid konservering av massa och då det inte finns några källor eller sänkor av massa blir $\chi = 0$, vilket ger ekvation 5.

$$
\frac{\partial}{\partial t}\varrho \; + \nabla\cdot(\varrho\mathbf{u}) = 0 \tag{5}
$$

Om $\varrho$ fixeras till en konstant leder det till att $\frac{\partial}{\partial t}\varrho \; = 0$ och att $\nabla\cdot(\varrho\mathbf{u}) = 0$. Det senare villkoret leder fram till sambandet om konserverandet av massa, se ekvation 2, som ofta ackompanjerar Navier-Stokes ekvation, 1. Sambandet innebär att fluiden är inkompressibel, det vill säga att fluiden alltid behåller sin ursprungliga volym.

## 2.4 Konservering av rörelsemängd

För rörelsemängd, $p = m\mathbf{u}$, fås enligt RTT att $\psi = \varrho\mathbf{u}$, vilket ger ekvation 6.

$$
\frac{\partial}{\partial t}\varrho\mathbf{u} \; + \nabla\cdot(\varrho\mathbf{u}\mathbf{u}) - \chi = 0 \tag{6}
$$

Utveckling av ekvation 6 ger ekvation 7, där $\chi$ representerar en källa eller sänka.

$$\mathbf{u}\frac{\partial}{\partial t}\varrho \ + \varrho\frac{\partial}{\partial t}\mathbf{u} \ + \mathbf{u}\mathbf{u} \ \cdot \nabla\varrho + \varrho\mathbf{u}\cdot\nabla\mathbf{u} + \varrho\mathbf{u}\nabla\cdot\mathbf{u} = \chi \tag{7}$$

Ytterligare omskrivning av ekvation 7 ger ekvation 8.

$$\mathbf{u}\left(\frac{\partial}{\partial t}\varrho + \mathbf{u}\cdot\nabla\varrho + \varrho\nabla\cdot\mathbf{u}\right) + \varrho\left(\frac{\partial}{\partial t}\mathbf{u} + \mathbf{u}\cdot\nabla\mathbf{u}\right) = \chi \tag{8}$$

I vänsterledet i ekvation 8 innehåller den första termen sambandet om konservering av massa, ekvation 5, då $\mathbf{u}\cdot\nabla\varrho + \varrho\nabla\cdot\mathbf{u} = \nabla\cdot(\varrho\mathbf{u})$. Detta medför att denna term måste vara lika med noll, vilket innebär att kvar blir ekvation 9.

$$\varrho\left(\frac{\partial}{\partial t}\mathbf{u} + \mathbf{u}\cdot\nabla\mathbf{u}\right) = \chi \tag{9}$$

En anekdot är att utveckling av ekvation 9 ger ett samband som kan knytas till Newtons andra lag, $\mathbf{F} = m\mathbf{a}$, se ekvation 10, där $\varrho$ representerar massan $m$, $\chi$ yttre krafter $\mathbf{F}$ och där materialderivatan för $\mathbf{u}$ representerar accelerationen $\mathbf{a}$.

$$\begin{aligned}\varrho\left(\frac{\partial\mathbf{u}}{\partial t} + u_x\frac{\partial u_x}{\partial x} + u_y\frac{\partial u_y}{\partial y} + u_z\frac{\partial u_z}{\partial z}\right) = \chi \quad \Leftrightarrow \\ \varrho\ \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{u}(x,y,z,t) = \chi\end{aligned} \tag{10}$$

Genom att studera ett fluidelement i en fluid fås att kraften $\chi$ kan delas upp i två termer enligt $\chi = \nabla\cdot\sigma + \mathbf{f}$, där $\nabla\cdot\sigma$ härstammar från tryck och där $\mathbf{f}$ härstammar från yttre krafter. Cauchys spänningstensor $\sigma$ kan förenklas till $\sigma = -pI + \mathbf{\Gamma}$ , där $p$ är medelvärdet av det normala trycket i $x$-, $y$- och $z$-led, $I$ är identitetsmatrisen och där $\mathbf{\Gamma}$ är en deriverbar spänningstensor som brukligt kallas för viskositetstermen. För en fluid i vila är exempelvis $\mathbf{\Gamma} = 0$. Detta resulterar i ekvation 11 som är Navier-Stokes generella ekvation.

$$\begin{aligned}\varrho\left(\frac{\partial}{\partial t}\mathbf{u} + \mathbf{u}\cdot\nabla\mathbf{u}\right) = \nabla\cdot\sigma + \mathbf{f} \quad \Leftrightarrow \\ \varrho\left(\frac{\partial}{\partial t}\mathbf{u} + \mathbf{u}\cdot\nabla\mathbf{u}\right) = \nabla\cdot(-pI + \mathbf{\Gamma}) + \mathbf{f} \quad \Leftrightarrow \\ \varrho\left(\frac{\partial}{\partial t}\mathbf{u} + \mathbf{u}\cdot\nabla\mathbf{u}\right) = -\nabla p + \nabla\cdot\mathbf{\Gamma} + \mathbf{f}\end{aligned} \tag{11}$$

# 3 Numerisk implementation

## 3.1 Val av beräkningsmetod

Det finns två olika sätt att basera sina beräkningar på, det Euleriska synsättet och det Lagranska synsättet. Det Lagranska synsättet ser fluiden som ett partikelsystem, där en partikel kan likställas med ett fluidelement som rör på sig och som går att associera med en enskild molekyl. Till skillnad från det Lagranska synsättet där enskilda partiklar studeras, bygger istället det Euleriska synsättet på att fixerade punkter i rummet studeras. Fluidens egenskaper som exempelvis hastighet, densitet och temperatur beräknas i varje given punkt.

Vid fluidsimuleringar på GPU lämpar det sig bäst att använda sig av det Euleriska synsättet, eftersom de intressanta punkterna i rymden med fördel kan placeras i ett rutnät som direkt kan representeras av en textur. Således tillämpas det Euleriska synsättet i det här projektet.

## 3.2 Implementation av Navier-Stokes ekvationer

För simulering av rök förenklas systemet i ekvation 1. Först inses att spänningstensorn $\mathbf{\Gamma}$ är försumbar för simulering av rök. Därefter divideras höger -och vänsterledet med $\rho$, för att sedan flytta över $\mathbf{u} \cdot \nabla \mathbf{u}$ till högerledet. Resultatet blir då ekvation 12, med villkoret 2. Mer teori och bakgrund om röksimulering finns att hämta i referens [2] och [4].

$$\frac{du}{dt} = -(\mathbf{u} \cdot \nabla \mathbf{u}) - \frac{1}{\rho}\nabla p + \mathbf{f} \tag{12}$$

Ekvation 12 kan brytas ned och beräknas komponentvis. Där $\mathbf{u} \cdot \nabla \mathbf{u}$ beskriver advektion av hastigheten, $\nabla p$ tryckförändringen i rummet och $\mathbf{f}$ är en lämplig extern kraft som verkar på rök. För varje komponent uppdateras rutnätet stegvis tills alla komponenter har beräknats, för att därefter påbörja en ny beräkning.

Den första komponenten som beräknas är advektionen. Advektion av en fysikalisk kvantitet kan beräknas numerisk med olika metoder. En användbar stegmetod för fluider är Eulers implicita stegmetod. Då Eulers explicita metod tenderar att ge okontrollerbar ökning av felet så är den implicita metoden stabil för vissa valda tidssteg och enkel att implementera på GPU:n. Advektionen beräknas genom att spåra kvantiteten till en punkt bakåt i tiden. Därefter beräknas en billinjär interpolation på dem fyra närliggande värdena i rutnätet.

När adkvetion har genomförts beräknas den andra komponenten, trycket. Trycket beräknas genom att först skapa en vektor $\mathbf{w}$, som en summa av två vektorer, enligt ekvation 13:

$$\mathbf{w} = \mathbf{u} + \nabla p. \tag{13}$$

Därefter appliceras divergensoperatorn på ekvation 13 och tillsammans med villkoret i ekvation 2 blir resultatet ekvation 14, Poissons ekvation.

$$\nabla^2 p = \nabla \cdot \mathbf{w} \tag{14}$$

I ekvation 14 står $\mathbf{w}$ för hastigheten som hittills räknats ut av dem tidigare komponenterna i ekvationen. Poissons ekvation kan skrivas om till matrisform, $\mathbf{Ax} = \mathbf{b}$, där $\mathbf{A}$ byggs av Laplaceoperatorn, $\mathbf{b}$ är konstanter från $\mathbf{w}$ och $\mathbf{x}$ är vektorn med sökta element. Metoden som används för att lösa systemet är Jacobis iterativa metod. Metoden går ut på att först initiera en gissning på en lösning av systemet och sedan iterera fram en förbättrad lösning. Processen är iterativ tills lösningen konvergerar. Det finns olika numeriska metoder till att lösa Poissons ekvation, men tack vare Jacobis enkla struktur utförs beräkningarna snabbt på GPU:n, även om metoden konvergerar långsammare än andra metoder.

För en fullständig lösning på Poissons ekvation krävs hantering av tryck- och hastighets-problem som uppstår vid randen. Hastigheten i randens riktning sätts då till 0 och för trycket används Neumanns randvillkor, på så sätt att $\frac{\partial p}{\partial t} = 0$.

Slutligen uppdateras hastigheten med externa krafter. Det finns olika lämpliga externa krafter som verkar på rök. Men för enkelhetens skull har bara en kraft tagits med i implementationen, rökens lyftkraft. Det är en kraft som återfinns i verkliga sammanhang och som beror av temperaturen i rummet och gasens densitet.

Från ekvation 12 kan den externa kraften, $\mathbf{f}$, beräknas genom ekvation 15.

$$\mathbf{f}_{buoy} = -\alpha\rho\mathbf{y} + \beta(T - T_{amb})\mathbf{y} \tag{15}$$

I 15 är $\alpha$ och $\beta$ positiva konstanter så att ekvationen är fysikalisk meningsfull. Temperaturen och den initiala temperaturen är $T$ respektive $T_{amb}$. Riktningen hos kraften är $\mathbf{y} = (0, 1, 0)$. Djupare teori om framtagning av temperatur och densitet finns att se i referens [4].

## 3.3 Simulering i MATLAB

MATLAB är ett verktyg med inbyggda kommandon som underlättar beräkningar av matriser. Med programmets kommandon kan en enklare flödessimulering utföras som en förstudie och ökad förståelse. Det är dock svårt att verifiera om en simulering utförs korrekt eftersom programmet saknar visuell funktionalitet.

I MATLAB användes samma steg som beskrivs i avsnitt 3.1 Implementation av Navier-Stokes ekvationer. Till skillnad från simuleringen som beskrivs i avsnitt 3.3 OpenGL/GLSL, simplifierades det visuella utseendet för att lösa och förstå de numeriska problem som kunde uppstå. Simuleringen visade bland annat att tidsteget gav varierande resultat och var känslig för mindre ändringar. Det insågs också att $\alpha$ och $\beta$ i ekvation 15 har en kraftig påverkan på ekvationen.

## 3.4 OpenGL/GLSL

Realtidssimuleringar av fluider är mycket krävande även för dagens moderna datorer, vilket innebär att metoder för bättre prestanda är eftertraktade. En

sådan teknik är att implementera den numeriska lösningen på GPU (istället för CPU), för att utnyttja möjligheten att köra beräkningarna parallellt. Även på moderna processorer är det möjligt att utföra beräkningar parallellt på fyra eller till och med åtta kärnor, dock kan detta inte mäta sig med de upp emot 2500 beräkningskärnor som kan tillgås på ett nyare grafikkort.

För GPU-implementationen användes programmeringsspråket C++ med grafikbiblioteket OpenGL för att styra programmet på applikationsnivå. För att komma närmare hårdvaran och kontrollera hur grafikkortet utförde beräkningarna användes även språket GLSL.

För att beskriva simuleringen behövdes att antal värden sparas för varje cell i volymen. Dessa värden kan intuitivt representeras som 3D-texturer med olika många komponenter per voxel (volymelement). Till exempel så kan en hastighet i tre dimensioner lagras som en vektor med tre komponenter. För att spara en hastighet för varje cell i volymen användes en 3D-textur där varje voxel innehåller tre komponenter. För att bestämma hur många komponenter en textur skulle ha användes OpenGL-funktionen i listing 1.

Listing 1: OpenGL-funktionen för att skapa en 3D-textur.

```
void glTexImage3D(
        GLenum target,
        GLint level,
        GLint internalFormat,
        GLsizei width,
        GLsizei height,
        GLsizei depth,
        GLint border,
        GLenum format,
        GLenum type,
        const GLvoid * data);
```

De parametrar som kontrollerar antalet komponenter är internalFormat och format. I tabell 1 visas en lista över de texturer som användes, hur många komponenter de hade och vilka värden som användes på de ovan nämnda parametrarna.

Tabell 1: Lista över de olika texturerna som används.

| Storhet | Komponenter | internalFormat | format |
|---|---|---|---|
| Hastighet | 3 | GL_RGB16F | GL_RGB |
| Densitet | 3 | GL_RGB16F | GL_RGB |
| Tryck | 1 | GL_R16F | GL_RED |
| Divergens | 1 | GL_R16F | GL_RED |
| Temperatur | 1 | GL_R16F | GL_RED |

För att kunna representera hinder användes ytterligare en 3D-textur. För detta krävdes endast en textur med en komponent. Om värdet i denna komponent var större än noll ansågs den voxeln i volymen att ligga i ett hinder. För en mer omfattande beskrivning av hur texturer skapas i OpenGL se referens [3, s. 138].

För att sedan kunna uppdatera värdena i texturerna användes så kallade *Framebuffer Objects*. Med *Framebuffer Objects* kunde resultatet från grafikkortets beräkningar sparas i texturer i stället för att ritas på skärmen. Varje textur binds

till ett *Framebuffer Object* som sedan kan aktiveras före rendering. Ett problem som behövde hanteras var att OpenGL inte kan rendera direkt till en 3D-textur. Istället behövde texturen delas in i tvådimensionella skivor som sedan renderades till var för sig. För att uppnå maximal prestanda och minska belastningen på CPU användes OpenGL funktionen i listing 2.

Listing 2: OpenGL funktion för att rendera flera instanser.

```
void glDrawArraysInstanced(
        GLenum mode,
        GLint first,
        GLsizei count,
        GLsizei primcount);
```

Parametern mode talar om vilken typ av primitiv som ska renderas och sattes i detta fall till GL_TRIANGLE_STRIP. first talar om var i vertex-listan renderingen ska starta. För den parametern sattes värdet noll då renderingen skulle börja från början av listan. Eftersom det var en quad som ritades sattes count till fyra, det vill säga antalet hörnpunkter som skulle renderas. Den mest intressanta parametern är primcount då den talar om hur många instanser av som ska göras av renderingen. Då volymen delades upp i z-led sattes primcount till djupet på volymen. Vid instansrendering ökas värdet på den inbyggda vertex-shader variabel *gl_InstanceID* med ett för varje instans. För att använda den variabeln skapades en vertex-shader som sparade den till en *Output* variabel och skickade den vidare till en geometry-shader. Vertex-shader visas i listing 3.

Listing 3: Vertex-shader för att hantera instansrendering

```
layout (location = 0) in vec4 vertexPosition;
flat out int layerInstance;

void main()
{
  layerInstance = gl_InstanceID;
  gl_Position = vertexPosition;

}
```

En geometry-shader behandlar, till skillnad från vertex-shader, inte enstaka hörnpunkter utan istället hela primitiver. I detta fall användes den för att koppla en instans av renderingen till rätt texturlager. Första steget var att sätta den inbyggda variabeln *gl_Layer* till värdet på den aktuella instansen. *gl_Layer* talar om för OpenGL till vilket lager i texturen som är bunden till den aktiverade *Framebuffer Object* som renderingen ska ske. Nästa steg var att ta fram det värdet som skulle användas för åtkomst till ett lager i en bunden textur. Detta kunde inte vara samma som *gl_Layer* då varje voxel behandlas från centrum men värdet från en textur hämtas från kanten. Detta löstes genom att introducera en förskjutning på 0.5 till värdet på *gl_Layer*. Slutligen skickade geometry-shadern hörnpunkterna vidare till nästa steg i renderingen. Shadern visas i listing 4.

Listing 4: Geometry-shader för att välja rätt texturlager.

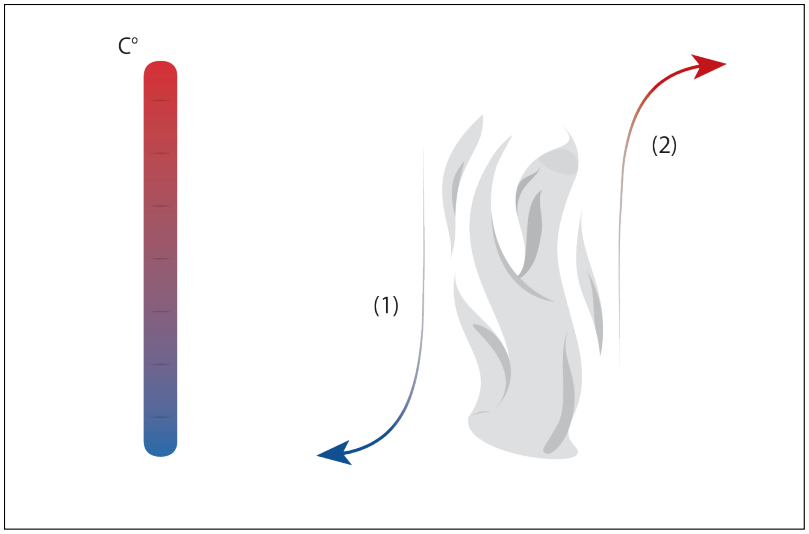
```
layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

flat in int layerInstance[3];
```

```
out float layer;

void main()
{
    gl_Layer = layerInstance[0];
    layer = float(gl_Layer) + 0.5;

    for (int i = 0; i < gl_in.length(); i++)
    {
        gl_Position = gl_in[i].gl_Position;
        EmitVertex();
    }
    EndPrimitive();
}
```

De ovan beskrivna vertex- och geometry-shaders är gemensamma för alla beräkningar som utfördes på GPU. Varje beräkning implementerades sedan som en egen fragment-shader. Nedan följer korta beskrivningar av dessa fragment-shaders i den ordning de körs i programmet. Alla konstanter som användes vid i implementationen listats i Bilaga A, tabell A1.

Advektions-termen implementerades i en shader som beräknade det nya värdet på en plats i volymen genom att använda Eulers implicita metod. Med hjälp av texturen som innehöll information om hinder kunde det avgöras om en voxel låg i ett hinder och i så fall sattes värdet där till noll.

För att beräkna lyftkraften som orsakas av temperaturen i volymen implementerades ekvation 15 i en fragment-shader. Utöver ekvation 15 implementerades även en term för rumstemperatur som gör så att en starttemperatur från början finns i hela volymen. Detta gjordes genom att jämföra om temperaturen i en punkt var mindre än rumstemperaturen och i så fall låta hastigheten vara oförändrad.




Figur 1: *Lyftkraften av rök beror på temperaturen av källan. Kallare temperaturer gör att röken sjunker, lik dimma (1) och en varmare temperatur skapar en stigande rök (2) likt den som skapas vid bränder.*

Nästa steg var att lägga till källor av olika slag. Samma fragment-shader kunde användas för att lägga till olika typer av källor så som densitet och temperatur men även för att lägga till hinder. Shadern ritade en sfär med en given radie på en given plats i en textur. Genom att använda alpha-blending kunde de nya värden blandas ihop med de gamla som redan låg i texturen och på så vis ge upphov till en källa.

Sedan beräknades divergensen av hastigheten, se högerledet i ekvation 14. Detta beräknades med central differensen av alla omkringliggande värden ur hastighetstexturen. Innan beräkningen utfördes kontrollerades om någon av de omkringliggande värde befann sig i ett hinder. Om så var fallet sattes hastigheten där till noll.

För att beräkna trycket ur ekvation 14 användes som nämnt Jacobi-iteration. I en fragment-shader implementerades en iteration som sedan kördes i en loop. Kring 20 iterationer visade sig ge ett bra resultat. På samma sätt som för divergensen användes de omkringliggande värdena fast denna gång för trycket. Ekvationen för att uppdatera trycket under en iteration visas i ekvation 16 där $\alpha$ och $\beta$ är konstanter, se tabell A1, och $b_{i,j}$ är divergensen.

$$p_{i,j} = (p_{x+1} + p_{x-1} + p_{y+1} + p_{y-1} + p_{z+1} + p_{z-1} + \alpha \cdot b_{i,j}) \cdot \beta \qquad (16)$$

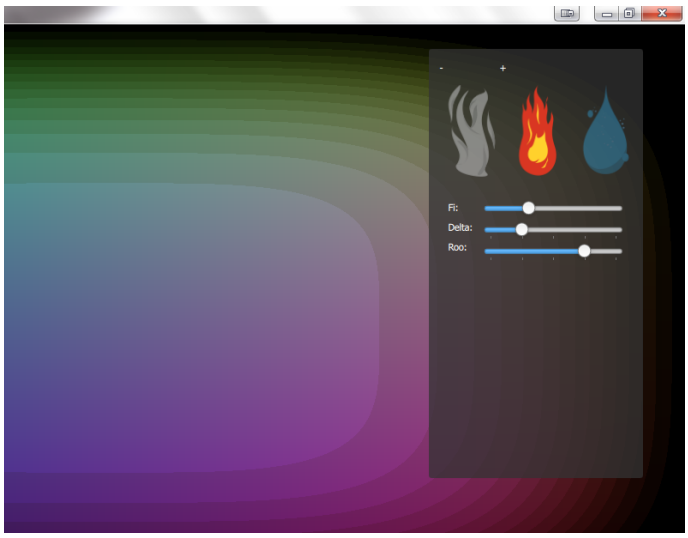
Sista steget i beräkningen var att beräkna gradienten av trycket och subtrahera den med hastigheten. Detta kommer från att den slutgiltiga hastigheten, $\mathbf{u}$, fås av att i ekvation 13 flytta $\nabla p$ till vänsterledet. Ytligare en gång hämtades alla omkringliggande värden ur tryck-texturen och dessa användes för att beräkna gradienten. Här kontrolleras även i fall någon omkringliggande cell befinner sig i ett hinder. I så fall sätts trycket till värdet i den centrala cellen och hastigheten sätts till noll.

# 4 Grafisk implementation

## 4.1 Användargränssnitt

För att utveckla och skapa ett användagränssnitt i C++ rekommenderas ofta Qt. Det är ett populärt multiplattformsbibliotek för att utveckla GUI:n, med ett egenutvecklat designskriptspråk, QML. Detta språket kommunicerar med C++ via en metod kallad ”Signals & Slots” och kan även hantera JavaScript-funktioner.

Inom projektet utvecklades fortfarande ett fungerande GUI som visade på funktionaliteten att användaren via interaktion med till exempel knappar kan kommunicera med C++-lagret. Tyvärr används inte denna implementation alls i resultatet av projektet, då vårt slutgiltiga bibliotek GLFW och Qt inte är kompatibla med varandra. Det skulle vara möjligt att använda Qt, förutsatt att man i början av projektet lägger mycket tid på att abstrahera C++-applikationen och Qt-gränssnittet, och anpassar sig efter de förutsättningar som finns.




Figur 2: *Användargränsnittet skapat i Qt som inte kom med i slutresultatet*

## 4.2 GLFW

GLFW är ett plattformsoberoende fönsterhanterings-ramverk utan onödiga finesser. GLFW har inte något inbyggt stöd för att göra GUI, men är ett mycket uppskattat ramverk bland utvecklare för att utveckla prestandatunga applikationer i OpenGL. Det som GLFW kan sköta är fönsterhantering och inmatning från bland annat mus och tangentbord.

När GLFW används vid utveckling av en OpenGL-applikationer behövs först ett annat ramverk för att ladda de OpenGL-funktioner som finns tillgängliga på systemet. För detta ändamål användes GLEW. I den här rapporten kommer inte GLEW att beskrivas i detalj. Det finns många bibliotek med samma funktionalitet och valet mellan dessa spelar inte någon större roll.

## 4.3 3D-texturer

Resultatet från den numeriska lösningen är en 3D-textur som innehåller diskreta värden för densitetens utbredning i fluiden. 3D-texturen visualiseras genom att ett 2D-plan plockas ut vid ett diskret z-värde i mitten av volymen för att sedan mappas till en quad som täcker hela *viewport*:en. Detta görs genom att rendera två trianglar som tillsammans bildar en rektangel och fyller hela skärmen. Hörnpunkterna skickas sedan till en vertex-shader som inte utför några beräkningar utan endast skickar den vidare. Det är först när de enskilda fragmenten ska behandlas som något meningsfullt sker. Här behandlas varje fragment av en fragment-shader som hämtar ett värde ur den bundna 3D-texturen beroende på fragmentets koordinater samt vilken nivå i z-led som ska visas. Viktigt att notera är att fragmentets koordinater antar värden mellan noll och fönstretsstorlek medan texturkoordinater ges mellan noll och ett. Det är alltså nödvändigt för fragment-shadern att veta om fönstret storlek så att koordinaterna kan normaliseras. Fragment-shadern syns i listing 5.

Listing 5: Fragment-shader för att visualisera ett lager ur en 3D-textur.

```
out vec3 color;
```

```
uniform sampler3D tex;
uniform float layer;
uniform vec3 dimensions;
uniform vec2 windowSize;

void main()
{
  color = texture(tex, vec3(gl_FragCoord.x / windowSize.x ,
  gl_FragCoord.y / windowSize.y, layer/dimensions.z)).xyz;
}
```

Den ovan nämnda metoden för att visualisera innehållet i en 3D-textur användes för att den är mycket lätt att implementera, dock går en stor del av meningen med att utföra simuleringen i tre dimensioner förlorad. Ett mer tilltalande alternativ hade varit att visualisera hela volymen i tre dimensioner med hjälp av volymrendering. Detta hade dock krävt betydligt mer implementation, vilket var utanför denna kurs ramar.

# 5 Diskussion

Från början gick projektet ut på att utveckla ett gränssnitt i Qt där användaren grafiskt fick modifiera parametrarna och dess värden i Navier-Stokes, med syftet att ge användaren en väldigt pedagogisk koppling mellan ekvationerna och dess visuella representation.

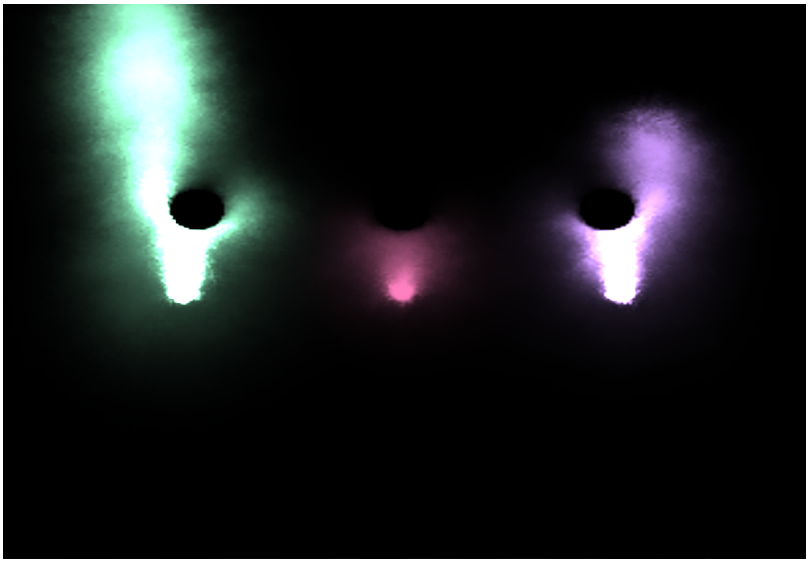
Om vi hade gjort om projektet hade vi utvecklat applikationen direkt i GLFW med tanke på att kursens tyngd låg i att utforma modellering och simulering, ej att producera en färdig produkt. Alldeles för sent i projektet upptäcktes varför det var så svårt att få en fungerande avancerad grafiska applikationer att fungera i Qt. Så fort projektet bytte fokus till GLFW gick det mycket snabbare att utveckla, även om det innebar att allt GUI-arbete bortprioriterades. Istället för ett grafiskt gränssnitt har vi skapat en konsol där det går att skriva in kommandon för att ändra på variabler och parametrar i programmet.

Med det sagt lämpar sig Qt fortfarande väl för tunga C++-applikationer som gynnas av användargränssnitt, men kraven på hur man utformar programvaran blir mycket högre och en kalkylerad avvägning rekommenderas. För utvecklingsteamet var det uppskattat att lära sig både Qt och GLFW under en och samma projekt. Det har dock fått följden att gruppen inte kommit så långt som visionen varit i något av systemen.

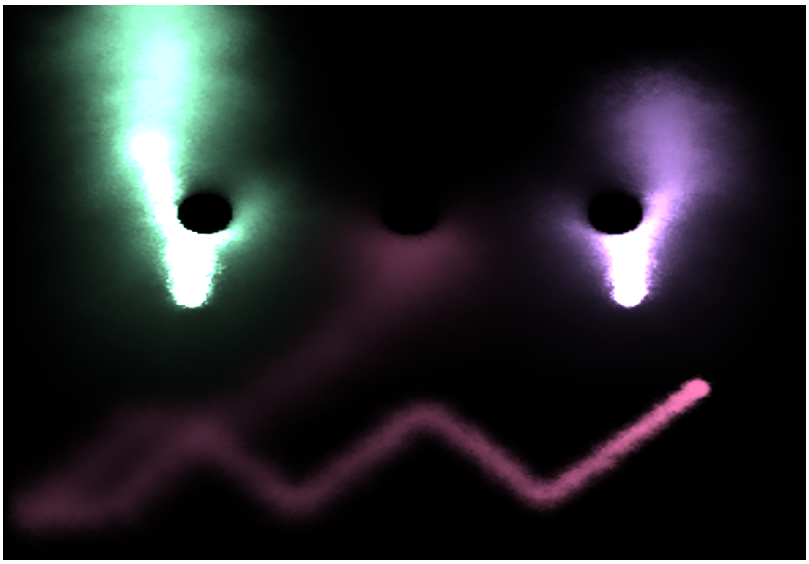
Den inledande idéen var att ha flera olika fluider, större användarkontroll, 3D-representation samt fler externa krafter representerade och dess värden modifierbara. Resultatet bedöms ändå som framgångsrikt, då de kunskapsmässiga målen för kursen överträffats och vi är nöjda med vår representation. Att göra applikationen GPU-baserad visade sig vara ett effektivt val. Simuleringen kör i realtid, så det går att interagera med den via piltangenterna, vilket inte hade varit möjligt annars.

# 6 Resultat

I figurerna 4 och 3 visas resultatet av att rendera ett lager av 3D-texturen som innehåller densitetens utbredning. De tre rökkällorna har olika densitet, där röken efter en tid breder ut sig och krockar med de cirkulära objekten. En värmekälla är placerad vid samma position som rökkällan i mitten i figur 3. Detta kan ses i figuren genom att observera hur röken på den vänstra och högra sidan rör sig i motsatt riktning till värmekällan. Implementationen tillåter en användare att styra den mittersta källan via piltangenterna, se exempel i figur 4. Genom att skriva in olika kommandon så går det att ändra på parametervärdena, exempelvis temperatur, som i detta fall är inställda enligt A1.



Figur 3: *Resultat av simuleringen vid startstadium.*



Figur 4: *Resultat av simuleringen efter att användaren har förflyttat en av punktkällorna med hjälp av piltangenterna.*

# 7 Slutord

Det finns en anledning till att fluidsimuleringar blivit så populära: det är både logiskt och visuellt stimulerande, det finns många appliceringar, och Navier-Stokes ekvationer är fortfarande, 2014, ett av de olösta matematiska millenieproblemen.

Att som studenter utforska och tillämpa dessa för att skapa denna applikation har varit roligt, utvecklande och bitvis också mycket utmanande. Om du som läsare vill göra något liknande, rekommenderar vi att läsa de referenser som använts under arbetets gång.

Förutom referenserna på nästa sida vill vi även tacka Anna Lombardi och Ulf Sannemo för deras handledning under projektets gång.

# Referenser

[1] Sebastian Böö, *Navier-Stokes ekvationer. Några enkla lösningar och tillämpningar inom meteorologi, http://www2.math.su.se/gemensamt/grund/exjobb/matte/2008/rep6/report.pdf*, Matematiska Institutionen, Stockholms Universitet, 2008

[2] Mark J. Harris, *Fast Fluid Dynamics Simulation on the GPU, GPU Gems, sida 637, Chapter 38*, NVIDIA och Addison-Wesley 2004

[3] Graham Sellers, Richard S. Wright, Jr. och Nicholas Haemel, *OpenGL Super Bible, Sixth Edition*, Addison-Wesley 2013

[4] Ronald Fedkiw, Jos Stam och Henrik Wann Jensen, *Visual Simulation of Smoke*, ACM SIGGRAPH 2001

[5] Keenan Crane, Ignacio Llamas och Sarah Tariq *Real-Time Simulation and Rendering of 3D Fluids, GPU Gems 3, Chapter 30, http://http.developer.nvidia.com/GPUGems3/gpugems3_ch30.html*, NVIDIA 2007

[6] Philip Rideout *Simple Fluid Simulation, http://prideout.net/blog/?p=58*, The Little Grasshopper 2010, 2014-03-11

# A Bilaga

Tabell A1: Lista över konstanter.

| Konstant: | Värde: |
|---|---|
| Tidssteg | 0.1 |
| Lyftkraft: $\alpha$ | 0,0125 |
| Lyftkraft: $\beta$ | 1 |
| Värmekällans temperatur vid start: $T_{amb}$ | 20 |
| Rummets temperatur vid start | 0 |
| Jacobi: $\alpha$ | -100 |
| Jacobi: $\beta$ | 0,1 |