Fachhochschule Köln – Abteilung Gummersbach
Fakultät Informatik

Tutorial

Simulation und Visualisierung von Rauch

von Tobias Hermann

Einleitung

Die Hardware von PCs und Konsolen wird immer leistungsfähiger, und um diese auch ausreizen zu können, müssen für Spiele immer komplexere Algorithmen zur realistischen Simulation von Vorgängen entwickelt werden, damit dem Benutzer immer wieder bessere Szenarien präsentiert werden können.
Eine wichtige und gleichzeitig komplizierte Rolle in diesem Bereich spielt die Simulation und Visualisierung von Flüssigkeiten und Gasen.
Hierzu gibt es unterschiedliche Ansätze, von denen ich in diesem Projekt den von Jos Stam (http://www.dgp.toronto.edu/~stam/) aufgreife, den er in seinem Artikel „Real-Time Fluid Dynamics for Games" aus dem Jahre 2003 auf der Game Developer Conference vorgestellt hat. (http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/GDC03.pdf)

Dieses Tutorial richtet sich an Programmierer, die schon einmal mit OpenGL gearbeitet haben. Unser Ziel ist es, am Ende eine optisch ansprechende interaktive Simulation von Rauch im zweidimensionalen zu erhalten. Ansätze zur Erweiterung in den 3D-Bereich werden auch geliefert.

Für die Erarbeitung der Grundlagen zu diesem Thema empfehle ich die NeHe-Tutorials (http://nehe.gamedev.net/), in denen der Benutzer Schritt für Schritt an den Umgang mit dieser Schnittstelle herangeführt wird.

Grundlagen

In diesem Tutorial wird eine Lösung für das Verhalten von Viskositäten (z.B. Rauch) behandelt, die auf den Navier-Stokes Gleichungen basiert. Der Fokus wird auf Stabilität, Geschwindigkeit und gute Optik gelegt, nicht auf physikalische Korrektheit. Das Programm, zu dem auch der komplette Quelltext geliefert wird, läuft flüssig in Echtzeit auf Standard-PCs und kann z.B. in Spielen eingesetzt werden.

Die Basisgleichungen:

$$\frac{(\partial u)}{(\partial t)} = -(u * \nabla) u + v \nabla^2 u + f$$

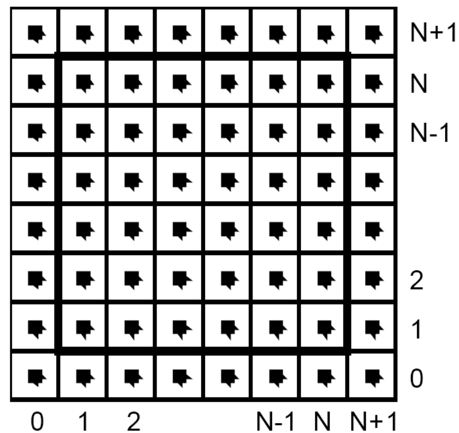
$$\frac{(\partial p)}{(\partial t)} = -(u * \nabla) p + k \nabla^2 p + S$$

Der Zustand des Rauches zu einem gewissen Zeitpunkt wird als Geschwindigkeitsfeld dargestellt. Man kann sich gut vorstellen, wie die Luft in einem Raum von einem Ventilator oder einer Heizung beeinflusst wird, und dass z.B. über der Heizung die Geschwindigkeitsvektoren der Luft nach oben zeigen würden.
Grob betrachtet sagt die obere Gleichung aus, dass die Geschwindigkeit sich in Abhängigkeit der drei Terme auf der rechten Seite der Gleichung ändert.

Ein Geschwindigkeitsfeld wird allerdings erst dann interessant anzuschauen, wenn es Rauchpartikel, Staub oder Blätter mit sich trägt. Die Bewegungen dieser Objekte werden aus der Geschwindigkeit der umgebenden Luft berechnet. Sehr leichte Objekte werden meist einfach von der Luft mitgetragen. Speziell für Rauch wäre es jedoch zu aufwendig viele einzelne Partikel zu berechnen, deshalb wird dieser einfach in Form eines Dichtefeldes dargestellt, welches Werte zwischen 0 (kein Rauch) und 1 annehmen kann.
Das Verhalten des Rauchs im Geschwindigkeitsfeld wird durch Gleichung 2 beschrieben. Es ist nicht nötig diese Gleichungen komplett zu verstehen, allerdings sollte auffallen, dass beide sehr ähnlich sind, was uns bei der Programmierung entgegen kommt.

Programmierung

Die Arrays, in denen die Geschwindigkeiten und die Dichten hinterlegt sind, kann man sich wie folgt vorstellen:



Außen um das eigentliche Arbeitsfeld ist noch ein Rand der Breite 1 definiert.
Der Einfachheit halber wird alles erst einmal für den zweidimensionalen Raum besprochen. Später folgt aber ein Ansatz für die Ausweitung in die dritte Dimension.

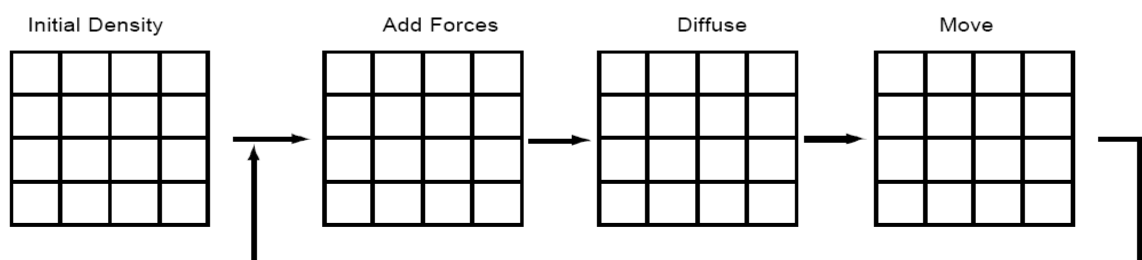
```
static u[size], v[size], u_prev[size], v_p rev[size];
static dens[size], dens_prev[size];
```

Die Arrays werden aus Geschwindigkeitsgründen eindimensional definiert. Die Elemente erreicht man über ein Makro.

```
#define IX(i,j) ((i)+(N+2)*(j))
```

Die Grundstruktur des Solvers, der die Berechnungen durchführt, sieht so aus, dass er die Werte für Dichte und Geschwindigkeit in Abhändigkeit der Ereignisse in der Umgebung aktualisiert.
Im Beispielprojekt zu diesem Tutorial kann der Benutzer mit der Maus Rauch hinzufügen und Geschwindigkeiten erzeugen. In einem Spiel könnten die Geschwindigkeiten von einem Ventilator ausgehen, einem atmenden Charakter oder einem sich schnell bewegenden Objekt. Der Rauch könnte an der Spitze einer Zigarette, eines Schornsteines oder ähnlichem entstehen.

Der Ablauf der Berechnungen läuft in dieser Reihenfolge ab:



Noch einmal zurück zur Dichtegleichung.

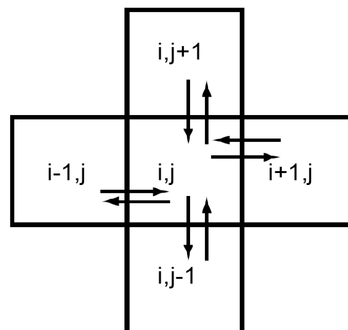
$$\frac{(\partial p)}{(\partial t)} = -(u * \nabla) p + k \nabla^2 p + S$$

Sie sagt aus, dass die Dichte vom Geschwindigkeitsfeld abhängig ist, diffundiert und durch Quellen erhöht werden kann.

Die Einbeziehung der Quellen kann leicht programmiert werden.

```
void add_source ( int N, float * x, float * s, float dt )
{
      int i, size=(N+2)*(N+2);
      for ( i=0 ; i<size ; i++ ) x[i] += dt*s[i];
}
```

Der zweite Schritt ist die Diffusion.



Der Parameter `diff` bestimmt in welcher Stärke die Diffusion zu den direkten Nachbarn stattfinden soll. Zur Berechnung wird der sogenannte Gauss-Seidel-Algorithmus benutzt.

```
void diffuse ( int N, int b, float * x, float * x0, float diff, float dt )
{
      int i, j, k;
      float a=dt*diff*N*N;
      for ( k=0 ; k<20 ; k++ ) {
            for ( i=1 ; i<=N ; i++ ) {
                  for ( j=1 ; j<=N ; j++ ) {
                        x[IX(i,j)] = (x0[IX(i,j)] + a*(x[IX(i-1,j)]+x[IX(i+1,j)]+
                        x[IX(i,j-1)]+x[IX(i,j+1)]))/(1+4*a);
                  }
            }
            set_bnd ( N, b, x );
      }
}
```

Nun zum dritten Schritt: Der Rauch (die Dichte) soll dem Geschwindigkeitsfeld folgen.

Hier bedienen wir uns der Methode des Backtracings. D.h. Es wird berechnet, woher ein Element, abhändig vom aktuellen Geschwindigkeitsfeld, seinen Dichtewert bekäme. Dieser liegt irgendwo zwischen 4 Elementen, deren Werte einfach interpoliert werden.

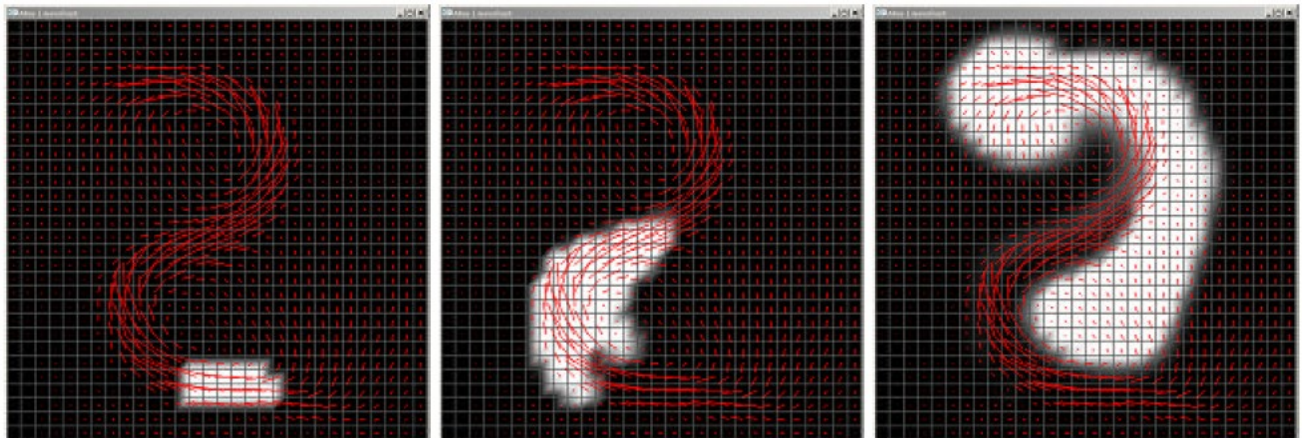
```
void advect ( int N, int b, float * d, float * d0, float * u, float * v, float dt )
{
	int i, j, i0, j0, i1, j1;
	float x, y, s0, t0, s1, t1, dt0;
	dt0 = dt*N;
	for ( i=1 ; i<=N ; i++ ) {
		for ( j=1 ; j<=N ; j++ ) {
			x = i-dt0*u[IX(i,j)]; y = j-dt0*v[IX(i,j)];
			if (x<0.5) x=0.5; if (x>N+0.5) x=N+ 0.5; i0=(int)x; i1=i0+1;
			if (y<0.5) y=0.5; if (y>N+0.5) y=N+ 0.5; j0=(int)y; j1=j0+1;
			s1 = x-i0; s0 = 1-s1; t1 = y-j0; t0 = 1-t1;
			d[IX(i,j)] = s0*(t0*d0[IX(i0,j0)]+t1*d 0[IX(i0,j1)])+
			s1*(t0*d0[IX(i1,j0)]+t1*d0[IX(i1,j1)]);
		}
	}
	set_bnd ( N, b, d );
}
```

Die Funktion dens_step ruft diese Unterfunktionen auf und führt so die Berechnungen, die die Gleichung verlangt, aus. In `x0` ist zu Anfang das Dichtefeld.

```
void dens_step ( int N, float * x, float * x0, float * u, float * v, float diff,float dt )
{
	add_source ( N, x, x0, dt );
	SWAP ( x0, x ); diffuse ( N, 0, x, x0, diff, dt );
	SWAP ( x0, x ); advect ( N, 0, x, x0, u, v, dt );
}
```

`SWAP` ist ein Makro zum Tauschen der Array-Pointer:

```
#define SWAP(x0,x) {float *tmp=x0;x0=x;x=tmp;}
```



Nun kann die Dichte problemlos dem Geschwindigkeitsfeld folgen.

Nun zum dritten Teil: Die Berechnung der Geschwindigkeit.

```
void vel_step ( int N, float * u, float * v, float * u0, float * v0,float visc, float dt )
{
	add_source ( N, u, u0, dt ); add_source ( N, v, v0, dt );
	SWAP ( u0, u ); diffuse ( N, 1, u, u0, visc, dt );
	SWAP ( v0, v ); diffuse ( N, 2, v, v0, visc, dt );
	project ( N, u, v, u0, v0 );
	SWAP ( u0, u ); SWAP ( v0, v );
	advect ( N, 1, u, u0, u0, v0, dt ); advect ( N, 2, v, v0, u0, v0, dt );
	project ( N, u, v, u0, v0 );
}
```

Bis auf die Funktion `project` wirkt alles sehr vertraut, weil wir es aus den Dichteberechnungen kennen.
Diese neue Funktion sorgt dafür, dass der physikalischen Gegebenheit, dass die Viskosität sich nicht einfach in eine Ecke des Arbeitsbereiches zusammenpressen lässt, genüge getan wird.
Wenn im Beispiel des Heizkörpers auf seiner Seite des Raumes die Luft aufsteigt, muss auf der anderen Seite zwangsläufig Luft absteigen, sonst wäre irgendwann alle Luft an der Decke und am Boden ein Vakuum, was natürlich keinen Sinn machen würde.

```
void project ( int N, float * u, float * v, float * p, float * div )
{
	int i, j, k;
	float h;
	h = 1.0/N;
	for ( i=1 ; i<=N ; i++ ) {
		for ( j=1 ; j<=N ; j++ ) {
			div[IX(i,j)] = -0.5*h*(u[IX(i+1,j)]-u[IX(i-1,j)]+
			v[IX(i,j+1)]-v[IX(i,j-1)]);
			p[IX(i,j)] = 0;
		}
	}
	set_bnd ( N, 0, div ); set_bnd ( N, 0, p );
	for ( k=0 ; k<20 ; k++ ) {
		for ( i=1 ; i<=N ; i++ ) {
			for ( j=1 ; j<=N ; j++ ) {
				p[IX(i,j)] = (div[IX(i,j)]+p[IX(i-1,j)]+p[IX(i+1,j)]+
				p[IX(i,j-1)]+p[IX(i,j+1)])/4;
			}
		}
		set_bnd ( N, 0, p );
	}
	for ( i=1 ; i<=N ; i++ ) {
		for ( j=1 ; j<=N ; j++ ) {
			u[IX(i,j)] -= 0.5*(p[IX(i+1,j)]-p[IX(i-1,j)])/h;
			v[IX(i,j)] -= 0.5*(p[IX(i,j+1)]-p[IX(i,j-1)])/h;
		}
	}
	set_bnd ( N, 1, u ); set_bnd ( N, 2, v );
}
```

Jetzt fehlt eigentlich nur noch die Erklärung zur Funktion set_bnd.
Sie legt einfach nur die Grenzen fest, die unser Arbeitsbereich hat, und aus dem Nichts austreten soll. Sie definiert sozusagen die Wände unseres Raumes.

```
void set_bnd ( int N, int b, flo at * x )
{
      int i;
      for ( i=1 ; i<=N ; i++ ) {
            x[IX(0 ,i)] = b==1 ? -x[IX(1,i)] : x[IX(1,i)];
            x[IX(N+1,i)] = b==1 ? -x[IX(N,i)] : x[IX(N,i)];
            x[IX(i,0 )] = b==2 ? -x[IX(i,1)] : x[IX(i,1)];
            x[IX(i,N+1)] = b==2 ? -x[IX(i,N)] : x[IX(i,N)];
      }
      x[IX(0 ,0 )] = 0.5*(x[IX(1,0 )]+x[IX(0 ,1)]);
      x[IX(0 ,N+1)] = 0.5*(x[IX(1,N+1)]+x[IX(0 ,N )]);
      x[IX(N+1,0 )] = 0.5*(x[IX(N,0 )]+x[IX(N+1,1)]);
      x[IX(N+1,N+1)] = 0.5*(x[IX(N,N+1)]+x[IX(N+1,N )]);
}
```

Der Aufruf der Funktionen läuft in dieser Reihenfolge ab:

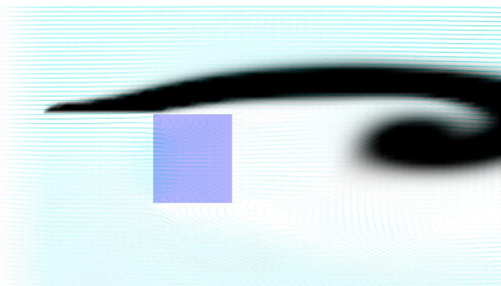
```
while ( simulating )
{
      get_from_UI ( dens_prev, u_prev, v_prev );
      vel_step ( N, u, v, u_prev, v_prev, visc, dt );
      dens_step ( N, dens, dens_prev, u, v, diff, dt );
      draw_dens ( N, dens );
}
```

Link: Ausführbare Datei der Simulation

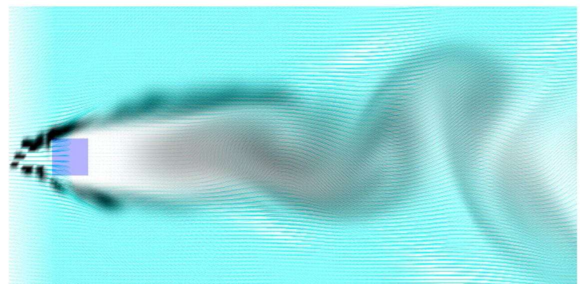
Erweiterungen

Eine mögliche Erweiterung wäre die Einführung einer dritten Dimension. Dies ist recht simpel, allerdings stellt sich die grafische Darstellung als problematisch dar. Ein Sourcecode und eine ausführbare Datei meiner Lösung liegen bei, aber Verbesserungsvorschläge sind herzlich willkommen.

Ebenfalls möglich, und im Hinblick auf Spiele sogar sehr nützlich, ist die Interaktion mit einem Objekt. (source + exe) Hierzu empfiehlt es sich, zusätzlich zu den Arrays für Rauchdichte und Geschwindigkeiten ein weiteres anzulegen, welches die Information darüber enthält, ob eine Zelle von einem Objekt besetzt ist, oder nicht. Durch eine Anpassung der set_bnd – Funktion lassen sich so auch klassische Effekte erzeugen, wie sie hier zu sehen sind.

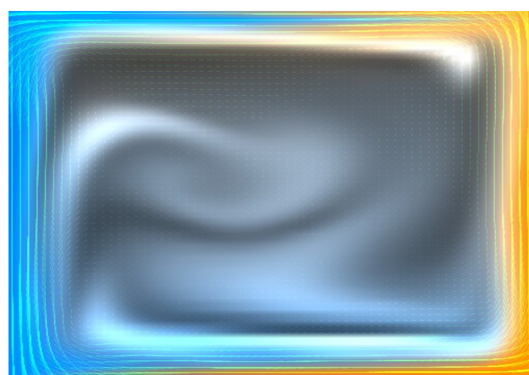


Verwirbelung



(Von Kármán vortex street)

Ebenfalls möglich ist die Simulation von Konvektionszellen, so genannte Benard-Zellen. (exe + souce)



Die beiliegenden Quelltexte für diese Effekte dürfen gerne benutzt, verbessert und weitergegeben werden.

Quelltextverzeichnisse zum Tutorial:

2D Simulation
2D Simulation mit Objekt
2D Simulation mit beweglichem Objekt
2D Simulation der Benard-Zellen
3D Simulation zwischen zwei Glasplatten
3D Simulation
3D Simulation mit Objekt

Ausführbare Dateien (.exe):

2D Simulation
2D Simulation mit Objekt
2D Simulation mit beweglichem Objekt
2D Simulation der Benard-Zellen
3D Simulation zwischen zwei Glasplatten
3D Simulation
3D Simulation mit Objekt

Literaturverzeichnis

[1] Jos Stam, "Real-Time Fluid Dynamics for Games". Proceedings of the Game Developer Conference, March 2003.
http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/GDC03.pdf

[2] Ronald Fedkiw, Jos Stam and Henrik Wann Jensen, "Visual Simulation of Smoke", In *SIGGRAPH 2001 Conference Proceedings, Annual Conference Series*, August 2001, 15-22
http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/smoke.pdf

[3] O. Staubli, C. Sigg, R. Peikert, D. Gubler: „Volume rendering of smoke propagation CFD data“, To appear in: Proceedings of IEEE Visualization '05 (Minneapolis, USA, October 23-28), 2005.
http://graphics.ethz.ch/~peikert/papers/smoke_small.pdf