Fachhochschule Köln – Abteilung Gummersbach

Fakultät Informatik

Projektarbeit

Thema:

Simulation und Visualisierung von Rauch

Prüfer: Prof. Wolfgang Konen

Name: Tobias Hermann
Studiengang: Technische Informatik
Matrikelnummer: 11032233

Inhaltsverzeichnis:

- Einleitung
- 2D und Interaktion mit einem Objekt
- 3D und Interaktion mit einem Objekt
- 2D und Erzeugung bestimmter physikalischer Phänomene
- Entwicklung des Tutorials

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 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)

2D und Interaktion mit einem Objekt (Link)

Jos Stam hat seiner Ausarbeitung ein Beispielprojekt für die 2-dimensionale Simulation beigelegt, welches jedoch keine Interaktion mit einem Objekt beinhaltet.
Um dies dann selbst zu realisieren, muss erst einmal geklärt werden, in welcher Form das Objekt gehandhabt wird. Hier benutze ich ein zusätzliches Array in der Größe der kompletten Fläche, in dem hinterlegt wird, welches Feld belegt ist und welches nicht. Somit spart man sich das Berechnen von Vektoren und Kanten.

```
static int * object;
// …
object         = (int *)   malloc ( size*sizeof(int  ) );
```

Eine bereits vorhandene Funktion zur Behandlung von Grenzen finden wir unter *set_bnd*. Diese Routine verhindert, dass der Rauch aus unseren Raum austritt. Durch eine kleine Anpassung wird nun in der zusätzlichen Funktion *set_bnd_objekt* abgefragt, welches Feld vom Objekt belegt ist, und dann wird der entsprechende Geschwindigkeitsvektor der Luft umgekehrt.

```
void set_bnd_objekt ( int N, int b, float * x )
{
        int i,j;
        for ( i=1 ; i<=N ; i++ ) {
                for ( j=1 ; j<=N ; j++ ) {
                        if ( object[IX(i,j)] > 0 ) {
                                x[IX(i,j)] = b>0 ? -x[IX(i,j)] : x[IX(i,j)];
                        }
                }
        }
}
```

Um deutlicherere Verwirbelungen hinter dem Objekt erzeugen zu können, habe ich die Möglichkeit, einen Ventilator anzuschalten, eingebaut. Somit wird der 2-D-Raum zu einer Art Windkanal, in dem das Objekt vom Benutzer mit der Maus bewegt werden kann.
Es wird ganz simpel in der Funktion für die Kantenbehandlung bei (ventilator_an == 1) eine Windgeschwindigkeit von 0.1f gesetzt.

```
void set_bnd ( int N, int b, float * x ) // b == 1 bei u ; b == 2 bei v
{
        int i;
        for ( i=1 ; i<=N ; i++ ) {
                if (ventilator_an == 1) x[IX(0  ,i)] = b==1 ? 0.1f : x[IX(1,i)];
                else                x[IX(0  ,i)] = b==1 ? 0 : 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  )] = 0.5f*(x[IX(1,0  )]+x[IX(0  ,1)]);
        x[IX(0  ,N+1)] = 0.5f*(x[IX(1,N+1)]+x[IX(0  ,N)]);
        x[IX(N+1,0  )] = 0.5f*(x[IX(N,0  )]+x[IX(N+1,1)]);
        x[IX(N+1,N+1)] = 0.5f*(x[IX(N,N+1)]+x[IX(N+1,N)]);
        set_bnd_objekt(N,b,x);
}
```

Der Ventilator kann in der Funktion key_func ein und ausgeschaltet werden.

```
case 'W': if ( ventilator_an == 0 ) ventilator_an = 1; else ventilator_an = 0;  break;
```

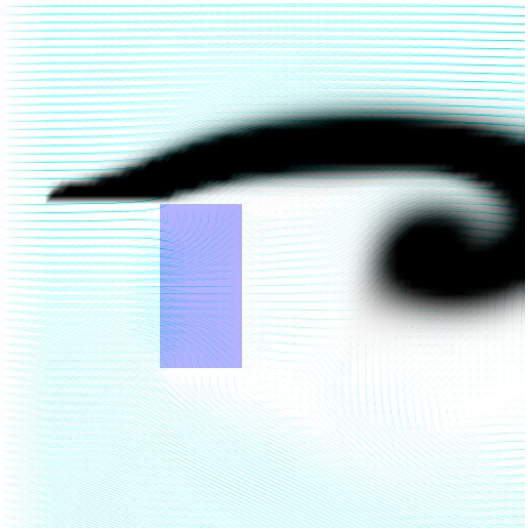


Bild 1 Bild 2

(Beide Bilder zeigen die Interaktion des Rauchs mit einem Objekt)

(Farben invertiert)

Wenn man eine wirklich korrekte Berücksichtigung der Objektränder haben wollte, müsste man nicht nur die set_bnd – Funktion anpassen, sondern auch alle anderen Bewegungsfunktionen, da sonst die Geschwindigkeitsvektoren noch ins Objekt hineinzeigen können und auch dort diffundieren. Zusätzlich müssten an den Kanten andere Druckberechnungen und an schrägen Kanten spezielle Umlenkungen über den Lot-Vektor usw. angestellt werden. Da man rein optisch jedoch auch ohne all dies eine anschauliche Simulation erhält, und die Geschwindigkeit berücksichtigt werden muss, kann man dies für unsere Zwecke vernachlässigen.

3D und Interaktion mit einem Objekt

Ein großes Problem, was sich mit der dreidimensionalen Simulation ergibt, ist die anschließende Visualisierung.
Dafür gibt es unterschiedliche Ansätze, hier einige davon:

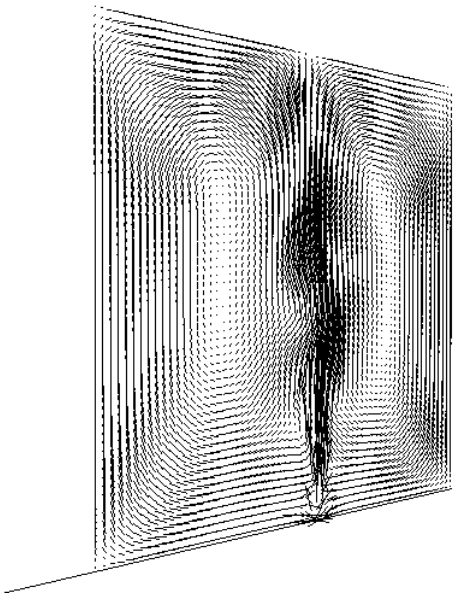
- Zwischen zwei Glasplatten
- Zusammensetzung aus Würfeln oder Flächen (quads)
- Volume Rendering http://graphics.ethz.ch/~peikert/papers/smoke_small.pdf
- Echtes Raytracing

Zuerst einmal zu einer leichteren Form der Simulation:
Die zweidimensionale Simulation **zwischen zwei Glasplatten** (Link) im dreidimensionalen Raum. Die Funktionen, die die Rauchbewegungen berechnen, sind die selben wie in der zweidimensionalen Simulation, nur die Darstellung sieht anders aus.
Hierzu musste zunächt Opengl erstmal für die dreidimensionale Darstellung initialisiert werden, und eine Camera eingeführt werden.
Ihre Startposition:

```
cameraControl.eyePosZ = 1.4;
cameraControl.eyePosX = 0.5;
cameraControl.eyePosY = 0.5;
```

Detailliertere Informationen dazu sind im Quelltext zu finden. Es handelt sich allerdings um Standard-Aufrufe, wie sie in fast jedem OpenGL-Programm vorkommen.

(Density)



(Velocity)

An die untere Kante des simulierten Bereiches habe ich eine Art „Schornstein" gesetzt. Eine Funktion, die Density und Geschwindigkeitsvektoren nach oben, mit einem gewissen Zufall behaftet, regelmäßig ausstößt.

```
static void schornstein ( float * d, float * u, float * v, float staerke )
{
	int i, j, size = (N+2)*(N+2);
	for ( i=0 ; i<size ; i++ ) {
		u[i] = v[i] = d[i] = 0.0f;
	}

	i = (int)zufallszahl(4)+2+N/2;
	j = 2;
	if ( i<1 || i>N || j<1 || j>N ) return;
	u[IX(i,j)] = 0;
	v[IX(i,j)] = staerke * force * 2;
		d[IX(i,j)] = source * staerke / 5;
	return;
}
```

So kann der Benutzer sich mit seinen Eingaben voll und ganz auf die Bewegung der Kamera konzentrieren, und in der Simulation läuft trotzdem etwas anschauliches ab.
Die Steuerungseingaben werden in den funktionen `normalKeyPressed, normalKeyReleased, specialKeyPressed und specialKeyReleased` abgefragt, die Mausbewegungen in `renderScene`.

Als nächstes folgt die **echte dreidimensionale Simulation**. (Link) Hierzu mussten als erstes einmal die Variablen und die Berechnungsfunktionen angepasst werden.

```
#define IX3(i,j,k) ( (i) + ( (N+2)*(j) ) + ( (N+2)*(N+2)*(k) ) )
#define FOR_EACH_CELL3 for ( i=1 ; i<=N ; i++ ) { for ( j=1 ; j<=N ; j++ ) { for ( k=1 ; k<=N ; k++ )
```

Zunächst bekommen die Definitionen, auf die immer wieder zurückgegriffen wird, die Erweiterung um die zusätzliche Dimension.
Dann werden die Funktionen, die die Bewegungsberechungen aufrufen, angepasst.
„w" ist jeweils der neue Vektor für die Z-Dimension

```
void dens_step3( int N, float * x, float * x0, float * u, float * v, float * w, float diff, float dt)
{
	add_source3 ( N, x, x0, dt );
	SWAP ( x0, x ); diffuse3 ( N, 0, x, x0, diff, dt );
	SWAP ( x0, x ); advect3 ( N, 0, x, x0, u, v, w, dt );
}
void vel_step3 ( int N, float * u, float * v, float * w, float * u0, float * v0, float * w0, float visc, float dt )
{
	add_source3 ( N, u, u0, dt ); add_source3 ( N, v, v0, dt ); add_source3 (N, w, w0, dt);
	SWAP ( u0, u ); diffuse3 ( N, 1, u, u0, visc, dt );
	SWAP ( v0, v ); diffuse3 ( N, 2, v, v0, visc, dt );
	SWAP ( w0, w ); diffuse3 ( N, 3, w, w0, visc, dt );
	project3 ( N, u, v, w, u0, v0 );
	SWAP ( u0, u ); SWAP ( v0, v ); SWAP ( w0, w );
	advect3 ( N, 1, u, u0, u0, v0, w0, dt ); advect3 ( N, 2, v, v0, u0, v0, w0, dt ); advect3 ( N,
	3, w, w0, u0, v0, w0, dt );
	project3 ( N, u, v, w, u0, v0 );
}
```

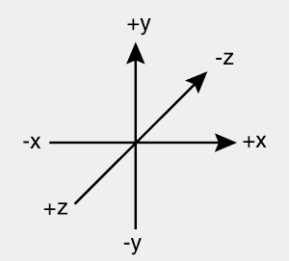
Der aufwendigste Teil der Anpassung für den dreidimensionalen Raum ist in den Funktionen `advect3` und `project3` zu finden.

Der Trick bei `advect3` liegt in der Zeile
`d[IX3(i,j,k)] = …`
Hier müssen die zuvor generierten Vektoren `s0, t0, r0, s1, t1 und r1` korrekt berücksichtigt werden.

```
void advect3 ( int N, int b, float * d, float * d0, float * u, float * v, float * w, float dt )
{
	int i, j, k, i0, j0, k0, i1, j1, k1;
	float x, y, z, s0, t0, r0, s1, t1, r1, dt0;

	dt0 = dt*N;
	FOR_EACH_CELL3
		x = i-dt0*u[IX3(i,j,k)]; y = j-dt0*v[IX3(i,j,k)]; z = k-dt0*w[IX3(i,j,k)];
		if (x<0.5f) x=0.5f; if (x>N+0.5f) x=N+0.5f; i0=(int)x; i1=i0+1;
		if (y<0.5f) y=0.5f; if (y>N+0.5f) y=N+0.5f; j0=(int)y; j1=j0+1;
		if (z<0.5f) z=0.5f; if (z>N+0.5f) z=N+0.5f; k0=(int)z; k1=k0+1;
		s1 = x-i0; s0 = 1-s1; t1 = y-j0; t0 = 1-t1; r1 = z-k0; r0 = 1-r1;
		d[IX3(i,j,k)] = s0 * t0 * r0 * d0[IX3(i0,j0,k0)] +
		                s1 * t0 * r0 * d0[IX3(i1,j0,k0)] +
		                s0 * t1 * r0 * d0[IX3(i0,j1,k0)] +
		                s0 * t0 * r1 * d0[IX3(i0,j0,k1)] +
		                s1 * t1 * r0 * d0[IX3(i1,j1,k0)] +
		                s0 * t1 * r1 * d0[IX3(i0,j1,k1)] +
		                s1 * t0 * r1 * d0[IX3(i1,j0,k1)] +
		                s1 * t1 * r1 * d0[IX3(i1,j1,k1)];

	END_FOR3
	set_bnd3 ( N, b, d );
}
```

Für die nächste Funktion ist es wichtig, bei den Nachbarn auch jeweils den in der dritten Dimension zu beachten.

Somit hat jeder Punkt 6 Nachbarn, die in die Berechnung mit einfließen.



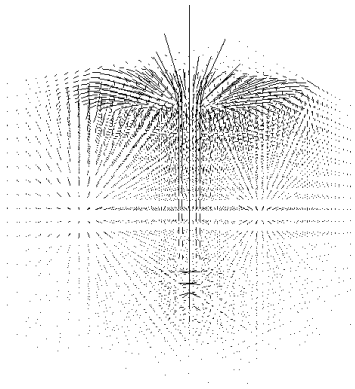
```
void project3 ( int N, float * u, float * v, float * w, float * p, float * div )
{
	int i, j, k;

	FOR_EACH_CELL3
		div[IX3(i,j,k)] = -(1.0f/2.0f)*(u[IX3(i+1,j,k)]-u[IX3(i-1,j,k)]+v[IX3(i,j+1,k)]-
		v[IX3(i,j-1,k)]+w[IX3(i,j,k+1)]-w[IX3(i,j,k-1)])/N;
		p[IX3(i,j,k)] = 0;
	END_FOR3
	set_bnd3 ( N, 0, div ); set_bnd3 ( N, 0, p );

	lin_solve3 ( N, 0, p, div, 1, 6 );
	FOR_EACH_CELL3
		u[IX3(i,j,k)] -= (1.0f/2.0f)*(float)N*(p[IX3(i+1,j,k)]-p[IX3(i-1,j,k)]);
		v[IX3(i,j,k)] -= (1.0f/2.0f)*(float)N*(p[IX3(i,j+1,k)]-p[IX3(i,j-1,k)]);
		w[IX3(i,j,k)] -= (1.0f/2.0f)*(float)N*(p[IX3(i,j,k+1)]-p[IX3(i,j,k-1)]);
	END_FOR3
	set_bnd3 ( N, 1, u ); set_bnd3 ( N, 2, v ); set_bnd3 ( N, 3, w );
}
```

In `lin_solve3` sieht es ähnlich aus, auch hier muss die Summe der Werte von sechs Nachbarfeldern berechnet werden.

```
void lin_solve3 ( int N, int b, float * x, float * x0, float a, float c )
{
	int i, j, k, k1;

	for ( k1=0 ; k1<20 ; k1++ ) {
		FOR_EACH_CELL3
			x[IX3(i,j,k)] = (x0[IX3(i,j,k)] + a*(x[IX3(i-1,j,k)]+x[IX3(i+1,j,k)]+
			x[IX3(i,j-1,k)]+x[IX3(i,j+1,k)]+x[IX3(i,j,k-1)]+x[IX3(i,j,k+1)]))/c;

		END_FOR3
		set_bnd3 ( N, b, x );
	}
}
```

Für die Darstellung habe ich die einfachste Variante (**Quads**) verwandt.
Diese Methode funktioniert ähnlich wie die Darstellung in der zweidimensionalen Ebene. Die Quadrate werden nun auch in die dritte Dimension hinein gemalt. Mit der Farbe muss man etwas tricksen.
Wenn die Farben einfach zwischen weiß und schwarz gewählt wären, würde die erste Ebene einfach alles andere verdecken (auch wenn sie keinen Rauch enthielte, also nur schwarz wäre).
Deshalb sind jetzt eigentlich alle Polygone weiß, jedoch ihre Transparenz wird durch die Density der einzelnen Felder gesteuert.
Ein Feld mit hoher Density (viel Rauch) hat wenig Transparenz und umgekehrt.
So kann eine solche Darstellung erreicht werden:

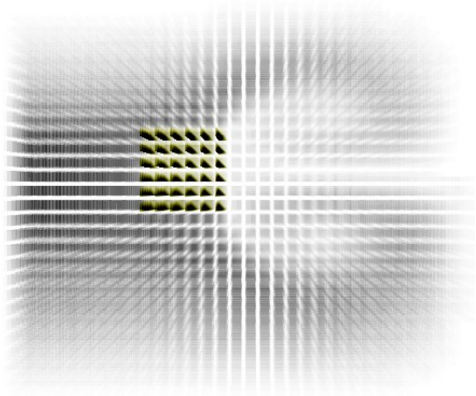
(3D-Density-Darstellung mit Quads)



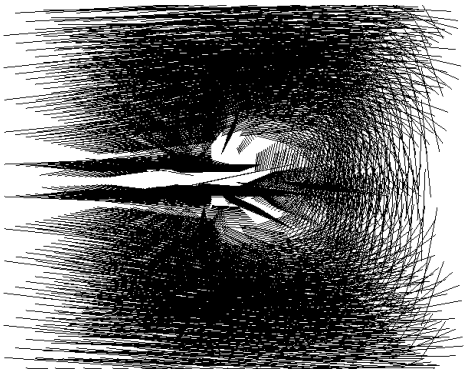
(Geschwindigkeitsvektoren)

Man erkennt die dreidimensionale Struktur des Rauches, besonders wenn man in der Simulation mit der Kamera das Objekt umkreist, jedoch ist diese optisch für Spiele nicht ansprechend genug. Hier müsste eine komplexere Darstellungsmethode verwendet werden.

Das Einbinden eines **Objektes** (Link) funktionierte nun durch Anpassen der set_bnd-funktion genau, wie bei der zweidimensionalen Variante über das Hinzufügen eines zusätzlichen Arrays, in dem hinterlegt wird, welche Voxel vom Objekt belegt sind und welche nicht.



(Objekt im 3D-Rauch)



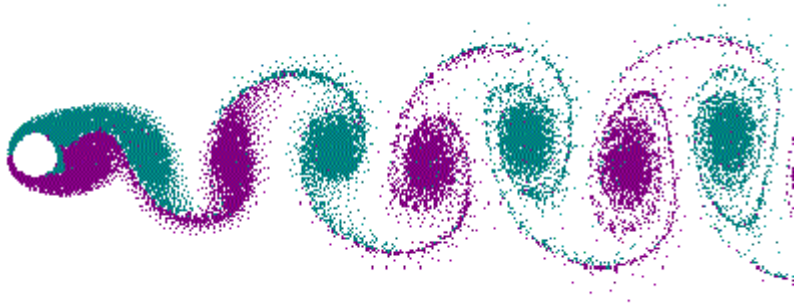
(dazugehörige Vektoren)

Volume Rendering (http://en.wikipedia.org/wiki/Volume_rendering), welches seine Anwendung hauptsächlich im medizinischen Bereich findet, sowie echtes **Raytracing** (http://de.wikipedia.org/wiki/Raytracing), was sich bis jetzt auf Grund mangelnder Geschwindigkeit der Hardware für den Echtzeit-Bereich noch nicht durchsetzten konnte, waren im Rahmen dieses Projektes wegen ihrer Komplexität nicht realisierbar.

2D und Erzeugung bestimmter physikalischer Phänomene

Abschließend habe ich versucht folgende physikalischen Phänomene im zweidimensionalen zu erzeugen:

Kármánsche Wirbelstraße: (Link)



(Bildquelle: Wikipedia)

Bei der Kármánschen Wirbelstraße handelt es sich um Verwirbelungen, die sich unter bestimmten Bedingungen hinter einem umströmten Objekt bilden können.
Dieser Effekt lässt sich im Zweidimensionalen mit der Simulation erzeugen, wenn man entweder das Objekt nach dem Programmstart einmal bewegt, um eine gewisse Anfangsunregelmäßigkeit (Asymetrie) zu schaffen, um den Anstoß für die Bildung der Kármánschen Wirbelstraße zu geben, wobei bleibt der Effekt leider nicht lange erhalten bleibt, und sich das Bild wieder recht schnell glättet,
oder durch die Einführung von Objekträndern.
Hierzu wird ein weiterer Wert benutzt, den ein Feld haben kann.

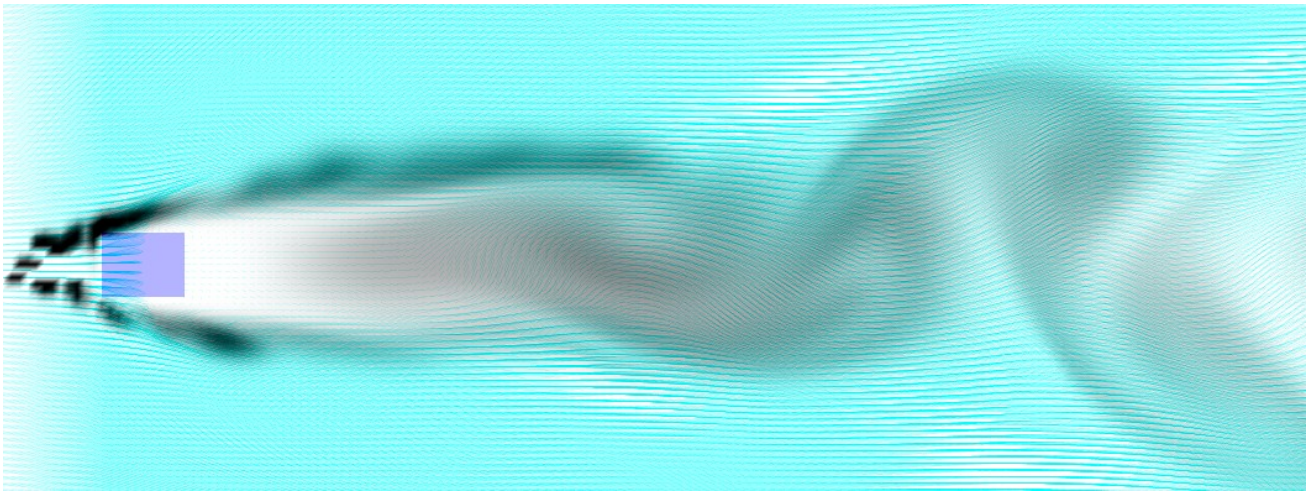
- 0 -> nicht belegtes Feld
- 1 -> Feld, dass durch ein Objekt belegt ist
- 2 -> Feld, dass durch eine Objektkante belegt ist (neu)

Die Funktion zur Erzeugung des Quaders sieht z.B. nun wie folgt aus:

```
void make_quader()
{
	int x,y;
	//int xlow=20, xhig=25;
	//int ylow=50, yhig=59;
	int xlow = obj_x - obj_s_x / 2;
	int xhig = obj_x + obj_s_x / 2;
	int ylow = obj_y - obj_s_y / 2;
	int yhig = obj_y + obj_s_y / 2;

	int i, size = (N+2)*(N+2);
	for ( i=0 ; i<size ; i++ ) {
		object[i] = 0;
	}

	for ( x = xlow ; x<xhig ; x ++) {
		for ( y=ylow ; y<yhig ; y++) {
			object[IX(x,y)] = 2;		// border of object = 2
		}
	}
	for ( x = xlow+1 ; x<xhig-1 ; x ++) {
		for ( y=ylow+1 ; y<yhig-1 ; y++) {
			object[IX(x,y)] = 1;		// inner part of object = 1
		}
	}
}
```

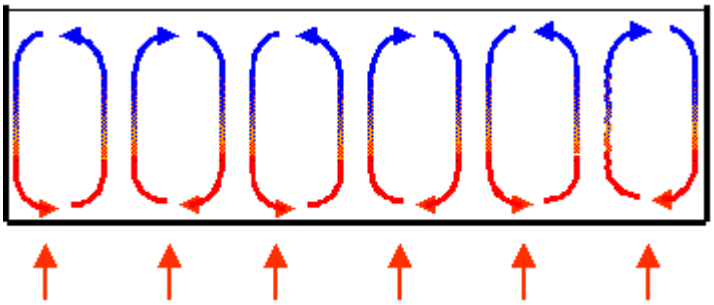
Des weiteren wird unter anderem die Funktion set_bnd_objekt angepasst:

```
void set_bnd_objekt ( int N, int b, float * x ) // b == 1 bei u ; b == 2 bei v
{
        int i,j,cnt;

        for ( i=1 ; i<=N ; i++ ) {
                for ( j=1 ; j<=N ; j++ ) {
                        if ( object[IX(i,j)] == 2 ) {
                                if (b==1) {
                                        // inverse horizontal velocity at vertical object border
                                        if (object[IX(i-1,j)]==0) x[IX(i,j)] =  -x[IX(i-1,j)];
                                        if (object[IX(i+1,j)]==0) x[IX(i,j)] =  -x[IX(i+1,j)];
                                } else if (b==2) {
                                        // inverse vertical velocity at horizontal object border
                                        if (object[IX(i,j-1)]==0) x[IX(i,j)] =  -x[IX(i,j-1)];
                                        if (object[IX(i,j+1)]==0) x[IX(i,j)] =  -x[IX(i,j+1)];
                                } else if (b==0) {
                                        // same density as active neighbour for egde-border,
                                        // average of two active neighbours for corner-border
                                        cnt=0;
                                        x[IX(i,j)] = 0;
                                        if (object[IX(i-1,j)]==0) {cnt++; x[IX(i,j)] += x[IX(i-1,j)];}
                                        if (object[IX(i+1,j)]==0) {cnt++; x[IX(i,j)] += x[IX(i+1,j)];}
                                        if (object[IX(i,j-1)]==0) {cnt++; x[IX(i,j)] += x[IX(i,j-1)];}
                                        if (object[IX(i,j+1)]==0) {cnt++; x[IX(i,j)] += x[IX(i,j+1)];}
                                        x[IX(i,j)] /= cnt;
                                }
                        }
                }
        }
}
```



(Wirbelstraße in der Simulation)

Bénard-Experiment: (Link)



(Bildquelle: Wikipedia)

Bei dem Bénard-Experiment entstehen dadurch, dass eine Viskosität von unten erwärmt wird, die auf dem Bild sichtbaren Zellen.

Die erste Vermutung war, dass man diesen Effekt durch Erhöhung der Diffusion des unteren Teils erzeugen könnte.
Allerdings tritt der Rauch dann zwar im unteren Bereich schneller in die benachbarten Zellen über, es entstehen allerdings nicht die typischen Bewegungsmuster.

Der zweite Lösungsansatz sieht wie folgt aus:
Jedes Feld bekommt eine zusätzliche Eigenschaft, die Temperatur, zugewiesen.

```
static float * temp, * temp_prev;
```

Nun wird unsere Viskosität von unten erwärmt, und damit sie nicht einfach immer heißer wird, von oben gekühlt, was z.B. beim kochenden Wasser der Wärmeabgabe nach oben an die Luft entsprechen würde.

```
void herdplatte( float * t, float avg )
{
	int i, size = (N+2)*(N+2);
	double ifloat, nfloat = N;


	for ( i=0 ; i<size ; i++ ) {
		t[i] = 0.0f;
	}
	for ( i = 2; i <= N-1; i++ ) {
		ifloat = i;
		t[IX(i,2)]=1; // unten warm
		t[IX(i,N-1)]=-1; // oben kalt

	}
}
```

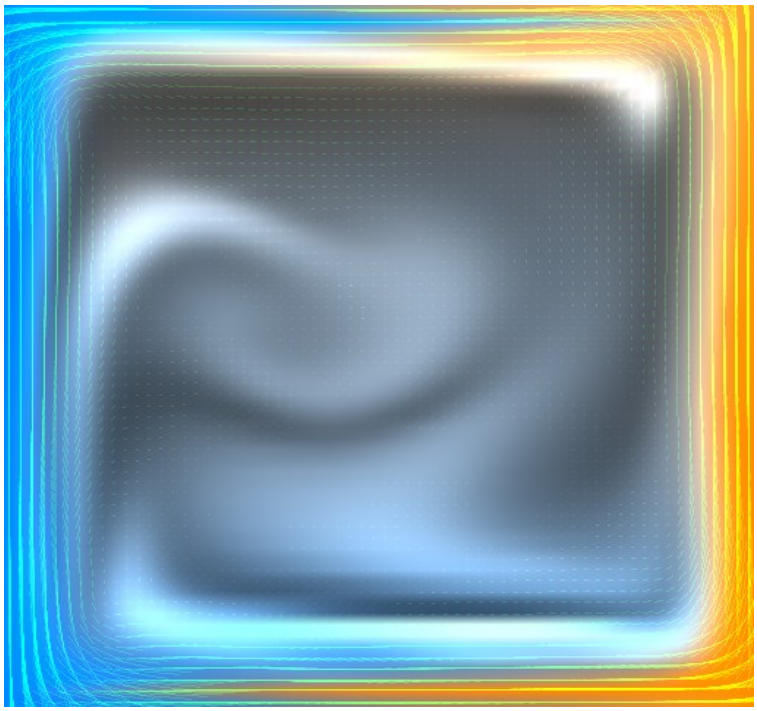
Um den Auftrieb, den eine Zelle aufgrund ihrer Temperatur gegenüber ihrer Umgebung erfährt, zu berechnen, gibt es mehrere Möglichkeiten. Den besten Effekt habe ich erhalten, wenn ich mit der Durchschnittstemperatur des gesamten Bereichs vergleiche.

```
void calc_temp_forces(float * temp, float *  v, float avg)
{
	int x, y;

	for ( x=0 ; x<=N ; x++ ) {
		for ( y=1 ; y<=N-1 ; y++ ) {
			v[IX(x,y)] += (temp[IX(x,y)] - avg) * auftriebskraft;
		}
	}
}
```

Diese Berechnung wird in den Ablauf wie folgt eingebaut:

```
static void idle_func ( void )
{
        float temp_avg;
        get_from_UI ( dens_prev, u_prev, v_prev );
        temp_avg = sum_of(temp) / (float)(N*N);
        herdplatte(temp_prev, temp_avg);
        calc_temp_forces(temp, v_prev, temp_avg);
        vel_step ( N, u, v, u_prev, v_prev, visc, dt );
        dens_step ( N, dens, dens_prev, temp, temp_prev, u, v, diff, dt );

        glutSetWindow ( win_id );
        glutPostRedisplay ();
}
```

Die Funktionen `vel_step` und `dens_step` übernehmen den Rest der Arbeit, ohne angepasst werden zu müssen.
So erhält man im Beispiel des quadratischen Arbeitsbereiches eine Bénard-Zelle, die sich nach anfänglichem chaotischen Verhalten je nach Parameterwahl stabil einpendelt.



Entwicklung des Tutorials

Abschliessend gehört zu dieser Projektarbeit die Entwicklung eines Tutorials bezüglich des Themas.
Das Tutorial zu diesem Projekt wendet sich an Programmierer, die schon Erfahrung mit OpenGL unter C++ haben. Es fasst den von Jos Stam geschriebenen Report zusammen, und baut auf diesem auf.