Gridfluid — 3D Grid Based Fluid Simulator

for Hewlett-Packard Scalable Visualization Array

Febr, 2008

Budapest University of Technology and Economics

Department of Control Engineering and Information Technology

Contents

Chapter 1

Introduction

One interesting and recently widely researched area of computer graphics is the simulation of fluid motion. Many phenomena that can be seen in nature like smoke, cloud formation, fire, and explosion show fluid-like behavior. Understandably, there is a need for good and fast fluid solvers both in scientific visualization and in special effects industry.

As the motion of fluids is very complex, high quality simulation in real-time is still a challenging problem. With the evolution of graphics hardware in recent years new algorithms have been developed that take advantage of the massively parallel architecture of GPUs, making it possible to see fluid dynamic simulation even in real-time.

We present a distributed GPU implementation of the fluid simulation and rendering. The algorithm is based on the Eulerian solution of the Navier–Stokes equations, and runs the simulation on a GPU cluster. The distributed implementation makes it possible to solve the equations on higher resolution data sets than in case of a single computer application while still preserving interactive frame rates.

Chapter 2

Algorithmic Background

2.1 Fluid Dynamics

The following subsections introduce the Navier–Stokes equations and their solution. The solution can be divided into several steps according to the subproblems of advection, diffusion, projection and vorticity confinement. The structure of these subsections reflect this division.

2.1.1 Equations of Fluid Motion

A fluid with constant density and temperature can be described by its velocity $\vec{u}$ and pressure p fields. These values both vary in space and in time:

$$\vec{u} = \vec{u}(\vec{x},t), \quad p = p(\vec{x},t).$$

The motion of a fluid is described by the Navier–Stokes equations:

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u}\cdot\nabla)\vec{u} - \frac{1}{\rho}\nabla p + \nu\nabla^2\vec{u} + \vec{F}, \tag{2.1}$$

$$\nabla\cdot\vec{u} = 0, \tag{2.2}$$

where ρ is the density, ν is the viscosity of the fluid, and in a Cartesian coordinate system $\nabla = (\partial/\partial x, \partial/\partial y, \partial/\partial z)$. Equation 2.1 describes the conservation of momentum while equation 2.2 states the conservation of mass (i.e. that the velocity field is divergence free). These equations should also be associated with the definition of the boundary conditions. The first term on the right side of equation 2.1 expresses the advection of the velocity field itself. This term makes the Navier–Stokes equation nonlinear. The second term shows the acceleration caused by the pressure gradient. The third term describes diffusion that is scaled by the viscosity, which a measure of how resistive the fluid is to flow. Finally $\vec{F}$ denotes the influence of external forces.

In the Navier–Stokes equations we have four scalar equations (the conservation of momentum is a vector equation) with four unknowns ($\vec{u}$, p). The pressure and the velocity fields are related according to these equations thus we can eliminate the pressure from them reducing the number of equations to three. The elimination is based on a mathematical technique, called *Helmholtz–Hodge decomposition*. According to the equation of preservation of mass, the velocity field must be divergence free. The

Helmholtz–Hodge theorem states that any vector field $\vec{w}$ can be decomposed into the sum of a divergence free vector field $\vec{u}$ and the gradient of a scalar field s that is zero at the boundary:

$$\vec{w} = \vec{u} + \nabla s. \tag{2.3}$$

So if we ignore the requirement of mass preservation during the computation, this requirement can be met later by projection step P according to equation 2.3. Rewriting this equation in another form gives us the formulae to make our field mass conserving:

$$\vec{u} = P(\vec{w}) = \vec{w} - \nabla s.$$

Using equation 2.3 and applying the divergence operator on both sides, we get:

$$\nabla^2 s = \nabla \cdot \vec{w}. \tag{2.4}$$

If we know $\vec{w}$, then scalar field s is computed by solving this equation, then projection operator $P(\vec{w}) = \vec{w} - \nabla s$ can be evaluated to obtain $\vec{u}$.

Assuming a regular grid where vector field $\vec{w}$ is available, equation 2.4 becomes a linear systems of equations in the form of $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ is a matrix, $\mathbf{x}$ is the vector of unknown samples of s and $\mathbf{b}$ is a vector of known samples describing the divergence of $\vec{w}$.

Let us apply projection operator P to the momentum preservation equation:

$$\frac{\partial \vec{u}}{\partial t} = P\left(-(\vec{u} \cdot \nabla)\vec{u} + \nu \nabla^2 \vec{u} + \vec{F}\right)$$

since $P(\vec{u}) = \vec{u}$ and $P(\nabla p) = 0$. Note that this way we could enforce the preservation of mass by eliminating the pressure from the unknowns.

The time function of velocity field $\vec{u}(t)$ is obtained by stepping time t by δt. The solution of the projected momentum preservation equation consists of several steps. Taking $\vec{u}(t)$, the velocity field at t, the terms of the right side are evaluated and added one by one, i.e. self-advected obtaining $\vec{w}_1$, modified according to the diffusion getting $\vec{w}_2$, accelerated according to external forces and vorticity confinement resulting in velocity field $\vec{w}_3$, and finally projection operator P is applied to guarantee the conservation of the mass:

$$\begin{array}{ccccccccc} & & \text{advect} & & \text{diffuse} & & \text{accelerate} & & \text{project} \\ \vec{u}(t) & \longrightarrow & \vec{w}_1 & \longmapsto & \vec{w}_2 & \longmapsto & \vec{w}_3 & \longmapsto & \vec{u}(t+\delta t). \end{array}$$

Let us consider these steps separately.

2.1.2 Advection

Advection describes how the fluid carries itself around:

$$\frac{\vec{w}_1 - \vec{u}}{\delta t} = -(\vec{u} \cdot \nabla)\vec{u}. \tag{2.5}$$

One possibility is to express $\vec{w}_1$ from this equation

$$\vec{w}_1 = \vec{u} - (\vec{u} \cdot \nabla)\vec{u}\delta t,$$

and replace differentials by finite differences. However, this type of forward Euler approach is numerically unstable. The reason of instability is that forward methods

predict the future based the present values, and each simulation step adds some error, which may accumulate and exceed any limit.

Instead a backward method is applied as proposed by Stam [7], that guarantees stability. A backward looking approach is stable since while predicting the future it simultaneously corrects the past. Thus the total error converges to a finite value.

A moving fluid transports objects with itself according to the direction of its velocity. The advection (or convection) of custom quantity Q in the fluid can be written in the following implicit form:

$$Q(\vec{x},t+\delta t) = Q(\vec{x}-\vec{u}(\vec{x},t)\delta t,t).$$

This equation means that we trace a particle backward along the path it travels, and use the quantities of this former position. This semi-Lagrangian method is called the method of characteristics.

Selecting velocity $\vec{u}(\vec{x},t)$ to be custom quantity Q we obtain the following equation for the advected velocity field:

$$\vec{u}(\vec{x},t+\delta t) = \vec{u}(\vec{x}-\vec{u}(\vec{x},t)\delta t,t).$$

2.1.3 Diffusion

The diffusion term describes how the fluid motion is damped. This damping is controlled by the viscosity of the fluid. Highly viscous fluids like syrup stick together, while low-viscosity fluids like gases flow freely. Here we also use a backward looking method, which means that the Laplacian is obtained from the future, yet unknown velocity field, and not from the current velocity field. This means that the diffusion term

$$\frac{\partial \vec{u}}{\partial t} = \nu \nabla^2 \vec{u}$$

is rewritten in the following implicit form:

$$(1-\nu\delta t\nabla^2)\vec{w}_2(\vec{x}) = \vec{w}_1(\vec{x}) \tag{2.6}$$

where $\vec{w}_1$ is the advected velocity field and $\vec{w}_2$ is the velocity field that also takes into account diffusion. The solution of such an equation is similar to that of equation 2.4 and is described in subsection 2.1.1.

As diffusion has noticeable effect on the simulation only in case of highly viscous fluids, this term can be omitted for low-viscosity fluids.

2.1.4 Vorticity confinement

Many phenomena like smoke or air mixture have high detail turbulent structures in their motion. These fine characteristics will be damped out by the numerical dissipation of the simulation. One way to restore these features is to add a pseudo-random perturbation to the velocity field. Fedkiw et al. proposed another method which added these details back only where they should appear [2]. We also used their solutions.

The first step is to compute the vorticity of the velocity field (i.e. the *curl* or *rot* of the vector field, which expresses the "rate of rotation" or the circulation density, that is the direction of the axis of rotation and the magnitude of the rotation):

$$\vec{\psi} = \nabla \times \vec{u}.$$

Then normalized vorticity location vectors are computed, which point from lower vorticity to higher vorticity areas:

$$\vec{N} = \frac{\vec{\eta}}{|\vec{\eta}|}, \qquad \vec{\eta} = \nabla|\vec{\psi}|.$$

Finally the direction and scale of the vorticity force is computed:

$$\vec{F}_{vc} = \varepsilon(\vec{N} \times \vec{\psi})$$

where ε controls the amount of the detail to be restored to the flow field. The computed vorticity force should simply be added to the velocity just like any other external forces. This term can be omitted if small scale turbulent features are not needed.

2.1.5 Adding the external and vorticity forces

Due to the external and vorticity forces the flow will accelerate, i.e. the velocity field will change proportionally to the forces:

$$\vec{w}_3 = \vec{w}_2 + (\vec{F} + \vec{F}_{vc})\delta t.$$

In our application the user can insert material inside the fluid, and also can add forces with custom direction to make the fluid flow.

2.1.6 Projection

The last step of the simulation is the projection by P:

$$\vec{u}(t + \delta t) = P(w_3).$$

2.2 Solution of the linear equations

The projection and the diffusion steps require the solution of a linear systems of equations in the form of $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. We can use Jacobi iteration to solve these equations, which has quadratic complexity, but is still the most time consuming operation of the simulation. A single iteration step requires the rendering of a full screen quadrilateral, and letting the pixel shader gather the values of $\mathbf{x}$ from neighboring pixels according to sparse matrix $\mathbf{A}$. However, where the neighboring elements are small, we can skip the gather operation, taking advantage of the early-z culling feature of the GPU [6]. The depth value is set proportionally to the maximum element in the neighborhood and to the iteration count. This way, as the iteration proceeds, the GPU processes less and less number of fragments, and can concentrate on important regions. According to our measurements, this optimization reduces the total rendering by 40%.

2.3 Visualizing the flow field

The discussed method computes the velocity field of the flow. If needed, the pressure can also be expressed from the Navier–Stokes equations. When we wish to visualize the flow, one option is to map these parameters to some color and opacity function, which in turn can be rendered by volume visualization algorithms. Instead of using

the velocity and the pressure, we can also assume that the flow carries a scalar *display variable* with itself. The display variable is analogous with some paint or confetti poured into the flow.

Using the advection formula for display variable d, the scalar field can also be updated in parallel with the simulation:

$$d(\vec{x},t+\delta t) = d(\vec{x}-\vec{u}(\vec{x},t)\delta t,t).$$

At a time, the color and opacity of a point can be obtained from the display variable using a user controlled transfer function.

2.4 Simulation on the GPU

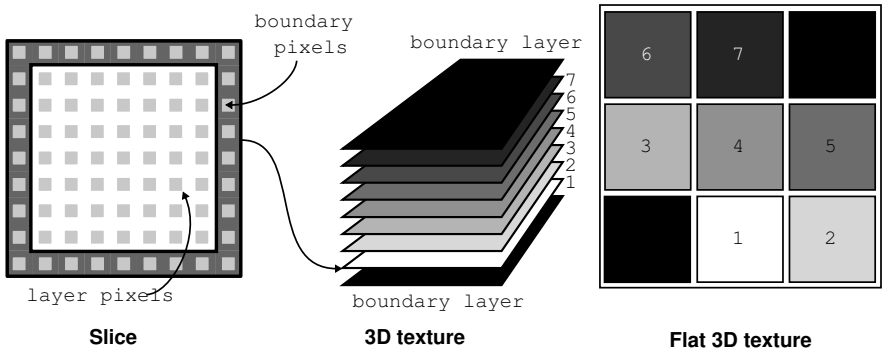

Figure 2.1: Volume slices and Flat 3D texture.

We used a collocated grid discretization to store our velocity, pressure, and density data. These variables are defined at the center of the voxels of a grid. At each time step the content of these data sets should be refreshed. As we do all the simulation on the GPU, we should store this data on the graphics card's memory in a way that it can be efficiently read and written by the graphics processor.

In GPU programming we usually store our data in textures. The problem with three dimensional data is that although the graphics cards support 3D data sets as 3D textures, no rendering is allowed to them. One can solve this problem with updating each slice of the volume into a separate texture and copy this data to the volume after refresh. This copy operation can take too much time making interactive rendering impossible for large data (this can be sidestepped by Shader Model 4 GPUs).

For efficiency reasons we used *flat 3D textures* as described by Harris [4]. A flat 3D texture tiles the slices of a 3D volume into a 2D texture (see Figure 2.1).

The result of the simulation is not only controlled by the external forces but also by the boundary conditions. These conditions describe what happens with the fluid at its boundary, i.e. on the faces of the cube encapsulating the grid volume. To store these conditions, we should extend each slice of the volume with one pixel border and we should also add two extra layers at the "top" and at the "bottom" of the volume.

Once the textures are ready, one simulation step of a layer of the volume can be done by the rendering of a quadrilateral, while boundary pixels are refreshed with rendering line primitives. As indexing a flat 3D texture needs extra work compared to

usual 3D textures, we prepare a lookup texture which helps to index the origin of any layer in the flat 3D texture. The rendered quadrilaterals also store information about the location of the layers right above and below in the flat 3D texture. Unlike in true 3D textures interpolating between two volume slices cannot be done by the texture units, thus interpolation is computed by custom shaders.

The simulation steps require to read and to write into a 3D data at the same time. Calculation one simulation step means rendering to a texture, but accessing a texture which is currently bound as a render target is not allowed. To overcome this limitation we should make a copy of our flat 3D textures — one will be sampled and one will be bound as a render target — and swap them after render (ping-ponging).

We used a 3D texture slicing rendering method to display the resulting display variable field, which means that we place semi-transparent polygons perpendicular to the view plane and blend them together in back to front order. The color and the opacity of the 3D texture is the function of the 3D display variable field.

Here we also had good use of the lookup texture to index our flat 3D textures to find the location of an arbitrary point in space in the tiled structure. Figure 2.3 shows one state of the flat 3D textures describing the velocity (right column) and display variable (left column) fields in a simulation. The top row shows these values with vorticity confinement turned off, and the bottom row shows the effect of vorticity confinement.

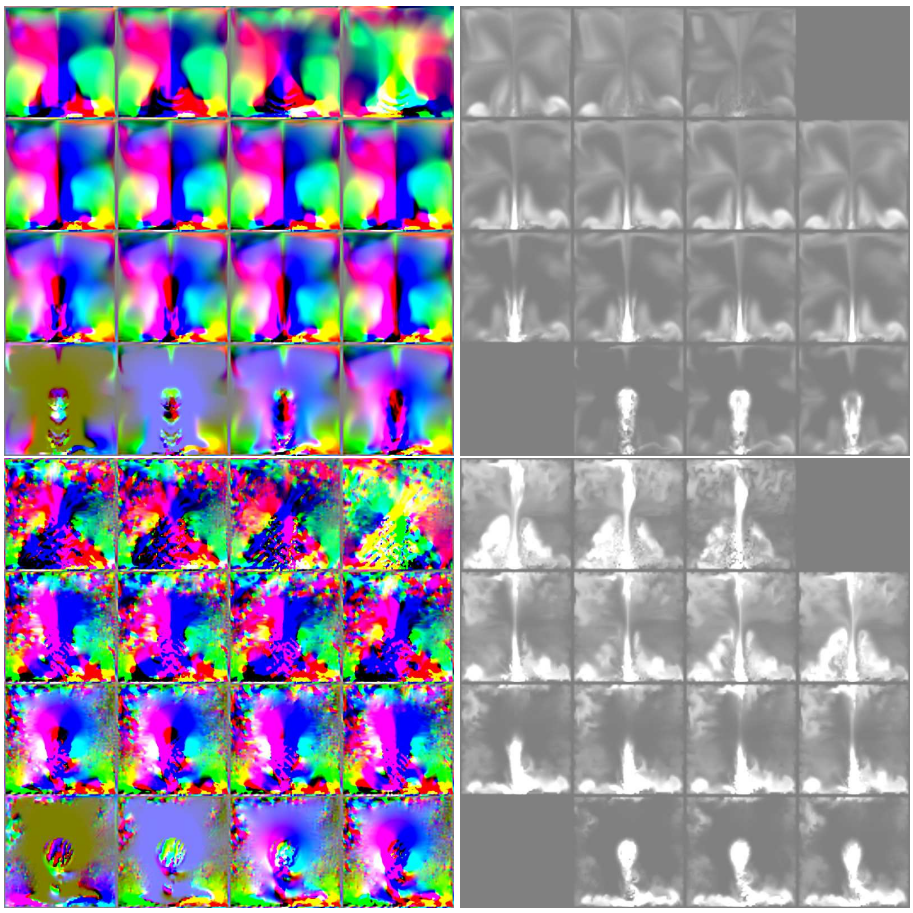

Figure 2.2: Flat 3D velocity (right column) and density (left column) textures of a simulation. The top row shows the simulation without, the bottom row with vorticity confinement.

2.5 Solid obstacles

If solid objects are present inside the fluid, the simulation should be modified. The density inside the voxels occupied by these objects should be zero, and the velocity should be set to the velocity of the object [1]. Furthermore, the boundary conditions should be properly set at the solid-fluid boundary.

An appropriate boundary condition is a *free-slip condition*, which states that fluid velocity equals to the object velocity in the direction of the boundary normal. This prevents the fluid from entering the solid object, but can freely flow along its surface (this friction is ignored). The free slip condition can be enforced after pressure projection. We should correct the result of the projection step in the following way: if the boundary voxel along a direction is a solid object, then we should use the corresponding component of its velocity. This requires the identification of voxels occupied by solid objects, and the determination of their velocity. In other words the object should be voxelized. To do this we create a separate grid containing the inside-outside information of the voxels and also the velocity values in the solid objects. We used an approach similar to [1], but we applied alpha blending instead of stencil operations. We render objects into each slice of the grid using an orthographic projection. The far clipping plane is set at infinity and the near clipping plane at the depth of the current slice. We render the object twice, once with back facing polygons and once with front facing polygons. We set up for additive blending and use value 1 for back faces and -1 for front faces. If the object is a closed shell, the result will contain a value 1 if the voxel is inside of the object and zero otherwise. If the mesh is a rigid mesh and does not deform, we can also write its velocity information into the first three components of this texture, while the alpha component stores the inside-outside information. In case of deformable objects like skinned characters special handling of the exact boundary is necessary [1].

After voxelization the effect of solid object can be added to the velocity and density fields similarly to external forces.

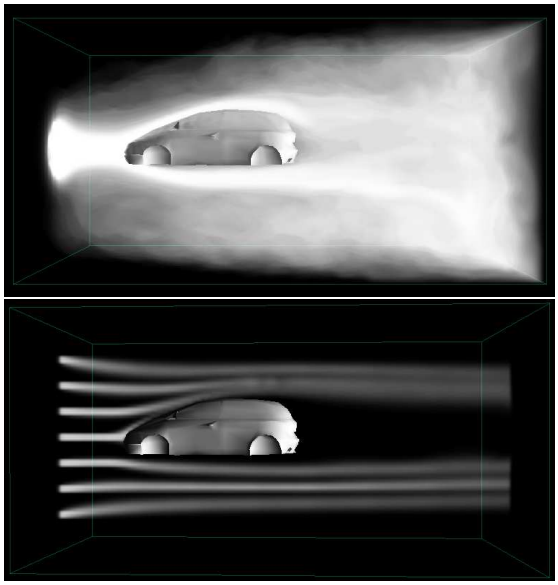

Figure 2.3: Obstacle objects in the fluid.

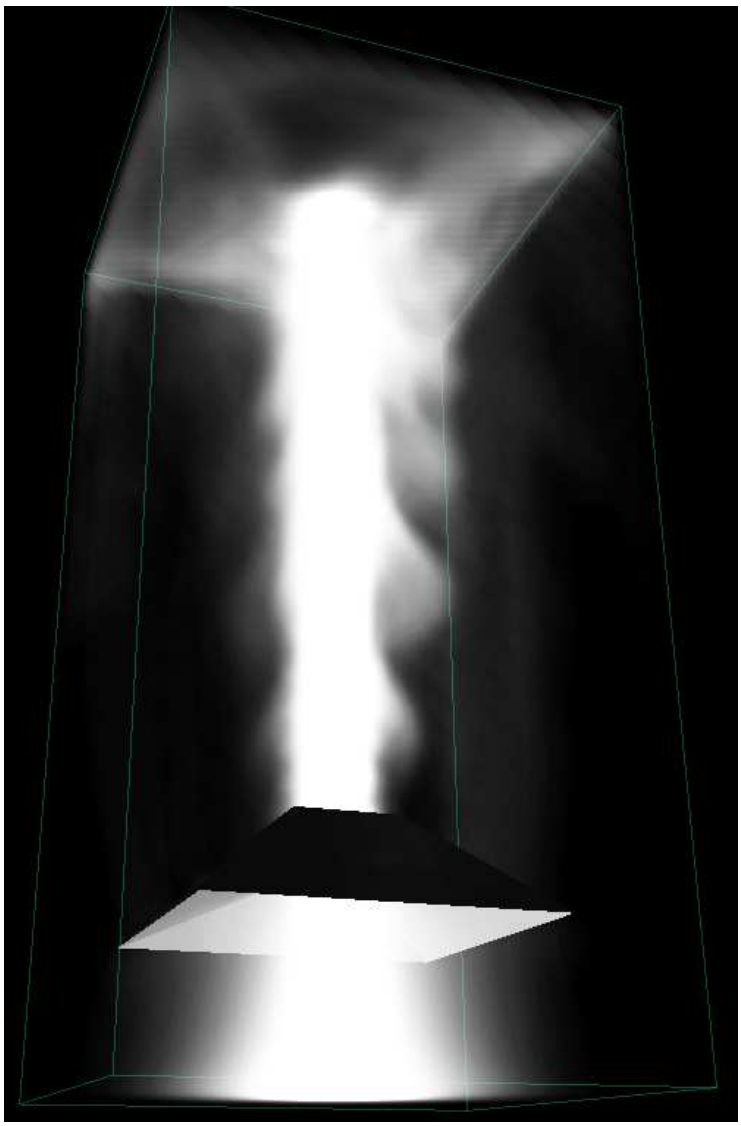

Figure 2.4: Obstacle objects in the fluid.

2.6 GPU Programs

2.6.1 Rendering

We store our 3D grid in the video memory as a 2D texture. To update the pixel values of this texture it should be bound as a render target and issue a render call that makes sure that the entire area of the viewport will be rasterised. It is usually done with rendering a so called full screen quad, which is a rectangular object covering the full viewport. As our data is a three dimensional data tiled into a two dimensional texture, and some pixels like boundary edges and boundary layers need special treatment, a single screen sized quad rendering is not suitable here.

We rendered a rectangular object for each layer of the volume, covering the pixels of the given layer in the flat 3D texture. For boundary edges we draw line primitives that cover the pixels of these edges.

Each voxel of the simulated grid has its unique texture coordinate in the flat 3D texture. If we use the texture rectangle OpenGL extension, these coordinates can be given in pixels instead of normalized coordinates. However, finding this texture coordinate for a voxel is not as easy in case of a flat 3D texture as in case of a true 3D texture, where the coordinates are the three voxel indices. In case of a tiled 3D texture one should know the resolution of the grid and the number of tiles in a row and in a column to determine in which pixel a givel voxel is located.

This indexing would cause a lot of extra work for the shaders doing the simulation calculations. To solve this problem, we should prepare some extra data to help address calculations in the shaders. Most of the calculations need to sample the neighborhood of a voxel. The neighbors in the x and y directions can easily be read since flat 3D textures preserve this neighborhood properties. However the neighbors in the upper and lower layers are located somewhere else in the flat 3D texture, depending on the layer resolution, the level of the current layer, and the number of column tiles.

As the neighbors from all directions are needed for the inner layer pixels, we also stored the coordinates of the upper and lower layers for these quadrilater' (Figure 2.5). Boundary layers only has a lower or an upper neighboring layer, so ly these coordinates should be given (Figure 2.6). Boundary edges have neighbors one direction and the neighborhood in these directions are preserved by the tiled ture, so it is enough to store the direction (Figure 2.7).

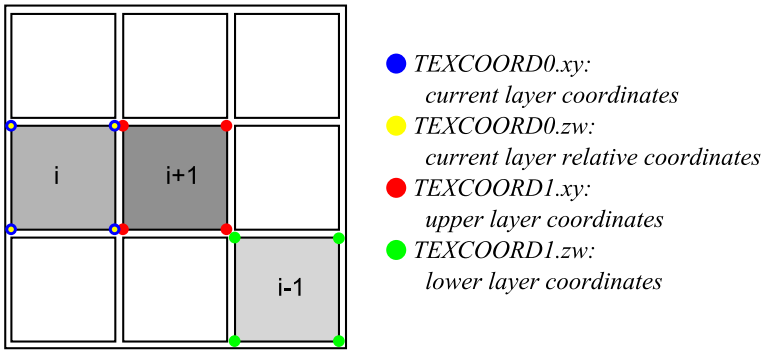

Figure 2.5: Texture coordinate setup of a quad for an inner layer.

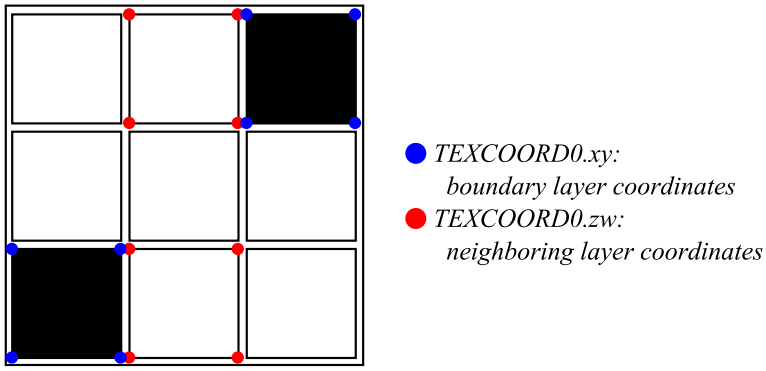

Figure 2.6: Texture coordinate setup of a quad for boundary layers.

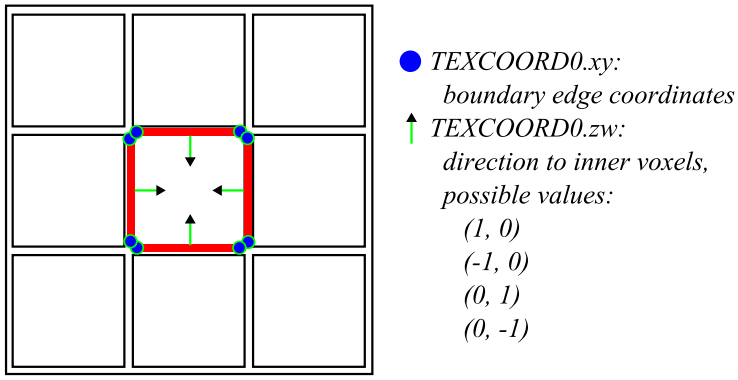

Figure 2.7: Texture coordinate setup of a line for a boundary edge.

In some cases we need to address not only a neighbor but an arbitrary voxel in the flat 3D texture. This happens during final rendering, and in case of the advection step. To easily cope with this problem we prepared a so called *Layer Offset* texture. This is a one dimensional texture with dimensions $1 \times N$ where N is the number of layers in the simulated grid (ie. the resolution along the z axis) including boundary layers. The n'th pixel of this texture stores the texture coordinates of the origin of n'th layer in the flat 3D texture (Figure 2.8).

The flat 3D texture coordinates (*flat3DCoord*) of a given voxel specified by three-dimensional coordinates *voxelCoord* can be computed with the use of the prepared offset texture (*offsetTex*) executing the following code line:

```
float2 flat3DCoord = texRECT(offsetTex, float2(voxelCoord.z, 0.5)).xy
                   + voxelCoord.xy;
```

The different simulation calculation steps require different fragment shaders but use the same vertex shader. The output of this vertex shader is listed in listing 2.1. The coordinates are set up like it was described above and shown in figures 2.5, 2.6, and 2.7.

Listing 2.2 shows the vertex shader program. The fragment shaders described in the following subsections — if not stated else — all use the output of this vertex shader.

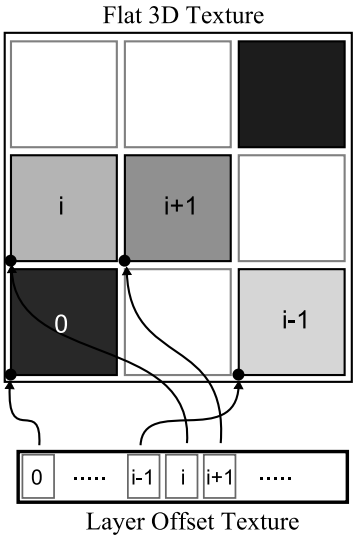

Figure 2.8: A 1D offset texture is used to easily address the origin of an arbitrary layer in the flat 3D texture.

```
struct VS_OUT {
    // vertex shader output position
    float4 hPos:POSITION;
    // texture coordinates (window coordinates)
    float4 texCoord:TEXCOORD0;
    // upper and lower layer texture coordinates
    float4 coord2: TEXCOORD1;
};
```

Listing 2.1: Vertex shader output.

We suggest the reader to refer to the descriptions and figures shown in this section whenever he gets confused about the meaning of the input values.

```
VS_OUT main_VS(
                float4 position : POSITION, //screen space coordinates
                float4 texCoord : TEXCOORD0, //texture coordinates
                float4 coord2   : TEXCOORD1 //texture coordinates
              )
{
    VS_OUT OUT;
    OUT.hPos = position;
    OUT.texCoord = texCoord;
    OUT.coord2 = coord2;
    return OUT;
}
```

Listing 2.2: Vertex shader program.

2.6.2 Boundary conditions

To set up the boundary conditions at the grid boundaries, we should refresh the pixels of the boundary layers and edges. We get the new values by simply scaling the value from the neighboring inner layer voxels. The scale value depends on the type of the boundary conditions we would like to use. For no-slip conditions of the velocity field, the scale factor should be -1, which states that the velocity becomes zero at the boundaries preventing the fluid to flow out of the grid bounds. For Neumann boundary conditions of the pressure field, this scale factor should be 1, resulting in zero gradient at the boundaries. For the density field, we should set this value to zero as no density is present outside the simulated grid.

Listing 2.3 shows the fragment shader used in boundary computation. The shader gets the scaling factor (*scale*) as a uniform floating point parameter, while it can sample the vector or scalar field of the given quantity from a flat 3D texture (*media*). The coordinates of the neighboring inner layer voxels are stored in the texture coordinates (see section 2.6.1).

```
// fragment shader running on the boundary layer pixels
float4 boundaryLayer_PS( VS_OUT IN,//vertex shader output
                         // quantity to be advected
                         uniform samplerRECT media,
                         // scale value to adjust boundary condition
                         uniform float scale
                        ):Color
{
    return scale * texRECT(media, IN.texCoord.zw);
}

// fragment shader running on the boundary edge pixels
float4 boundaryEdge_PS( VS_OUT IN,//vertex shader output
                        // quantity to be advected
                        uniform samplerRECT media,
                        // scale value to adjust boundary condition
                        uniform float scale
                        ):Color
{
    return scale * texRECT(media, IN.texCoord.xy + IN.texCoord.zw);
}
```

Listing 2.3: Setting boundary voxel values.

2.6.3 Advection

Listing 2.4 shows the fragment shader implementing backward Euler advection. The velocity field and the advected quantity field are passed as flat 3D textures (*velocity, media*). The time step is passed as a uniform float parameter (*dt*). As the shader should be able to sample an arbitrary voxel in the grid, the actual grid layer level (*l*) and the precalculated layer offset texture (*offsetTexture*) are also passed to the program (see section 2.6.1).

Since we use flat 3D textures to represent a 3D volume, the interpolation along the third coordinate needs special considerations. Interpolation along the first two coordinates is supported by the hardware, but we need to interpolate manually along the third coordinate.

```
// fragment shader running on the internal pixels of the layers during advecting a quantity
 float4 advect_PS( VS_OUT IN,//vertex shader output
                   // velocity field in a flat 3D texture
                      uniform samplerRECT velocity,
                   //advected media in a flat 3D texture
                      uniform samplerRECT media,
                   // timestep
                      uniform float dt,
                   // helper texture
                      uniform samplerRECT offsetTexture,
                   // current layer level along the z axis
                      uniform float l,
                   // resolution of one layer without boundaries
                      uniform float2 layerResolution
                    ):Color
{
    //read velocity value from velocity field, determine backward vector
    float3 vel = -texRECT(velocity, IN.texCoord.xy).rgb * dt;

    //get the layer level of the voxel after the back step
    //(l is the current level and layers are placed along axis z)
    float zSample = l + vel.z;
    // interpolation along the z direction ie. between two layers
    //are not supported by hardware in flat 3D textures
    //so need to do it manually
    //get the level of the two neighboring layers:
    float lNew1 = floor(zSample);
    float lNew2 = lNew1 + 1;
    //get the texture coordinates of their origin:
    float2 offset1 = texRECT(offsetTexture, float2(lNew1, 0.5)).rg;
    float2 offset2 = texRECT(offsetTexture, float2(lNew2, 0.5)).rg;
    //read the two values stored in the appropriate pixels:
    clampedXY = min(max(float2(-1,-1), IN.texCoord.zw + vel.xy), layerResolution + 1);
    float4 sample1 = texRECT(media, offset1 + clampedXY);
    float4 sample2 = texRECT(media, offset2 + clampedXY);
    //do the interpolation
    float4 newValue = lerp(sample1, sample2, zSample - lNew1);
    // write the new value of the quantity
    return newValue;
}
```

Listing 2.4: Advection fragment shader.

2.6.4 Pressure calculation and diffusion

As it was described before in 2.2 Both equation 2.4 and equation 2.6 are discretized Poisson equations, which are linear systems of equations in the form of $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ is a matrix, $\mathbf{x}$ is the vector of unknowns and $\mathbf{b}$ is a vector of known values. Just like in [3] we used the simplest iteration technique, the Jacobi iteration to solve these equations. One iteration step of the solution of both the pressure and the diffusion equation can be written in the following form (for details see [3]):

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

where for the pressure equation x represents $p\delta t$, b represents $\nabla \cdot \vec{w}$, $\alpha = (\delta x)^2$, and $\beta = 6$. For the diffusion equation x and b represent $\vec{w}$, $\alpha = (\delta x)^2 / v\delta t$, and $\beta = 6 + \alpha$.

Listing 2.6 shows the fragment shader code used to compute one Jacobi iteration. The vector fields of $\mathbf{x}$ and $\mathbf{b}$ are passed as flat 3D textures, while α and β are stored as uniform floating point parameters.

To calculate the pressure field, first we need the divergence of the velocity field. Listing 2.5 shows the fragment shader used to calculate the velocity divergence.

```
float4 divergence_PS( VS_OUT IN, //vertex shader output
                      uniform samplerRECT velocity // velocity field
                    ):Color
{
    // read velocity values from neighouring voxels
    float4 L = texRECT(velocity, IN.texCoord.xy + float2(-1, 0));
    float4 R = texRECT(velocity, IN.texCoord.xy + float2(1, 0));
    float4 T = texRECT(velocity, IN.texCoord.xy + float2(0, 1));
    float4 B = texRECT(velocity, IN.texCoord.xy + float2(0, -1));
    float4 U = texRECT(velocity, IN.coord2.xy);
    float4 D = texRECT(velocity, IN.coord2.zw);

    //compute divergence
    float divergence = (R.x - L.x + T.y - B.y + U.z - D.z) / 2.0;

    return divergence;
}
```

Listing 2.5: Fragment shader that calculates the divergence of the velocity filed.

```
float4 Jacobi_PS( VS_OUT IN, //vertex shader output
                  uniform samplerRECT x,
                  uniform samplerRECT b,
                  uniform float alpha,
                  uniform float beta
                ):Color
{
    // read the value of b from the current voxel
    float4 B = texRECT(b, IN.texCoord.xy);

    // read the values of x in the neighboring voxels
    float4 xL = texRECT(x, IN.texCoord.xy + float2(-1, 0));
    float4 xR = texRECT(x, IN.texCoord.xy + float2(1, 0));
    float4 xT = texRECT(x, IN.texCoord.xy + float2(0, 1));
    float4 xB = texRECT(x, IN.texCoord.xy + float2(0, -1));
    float4 xU = texRECT(x, IN.coord2.xy);
    float4 xD = texRECT(x, IN.coord2.zw);

    // zero the contribution of solid cells
    float4 xC = texRECT(x, IN.texCoord.xy);
    if(xL.a == 1) xL = xC;
    if(xR.a == 1) xR = xC;
    if(xT.a == 1) xT = xC;
    if(xB.a == 1) xB = xC;
    if(xU.a == 1) xU = xC;
    if(xD.a == 1) xD = xC;

    // calculate the next value of x
    float4 xNext = (xL + xR + xT + xB + xU + xD + alpha * B) / beta;

    return xNext;
}
```

Listing 2.6: Fragment shader of a Jacobi iteration step.

2.6.5 Projection

According to equation 2.3, the divergence free velocity field can be obtained from the divergent velocity field by subtracting the gradient of a scalar field. If we choose to use the pressure field for the scalar field we can enforce the mass preservation and eliminate the pressure from the unknowns of the Navier-Stokes momentum preservation equation (see section 2.1.1). Listing 2.7 shows the fragment shader that does the projection step. The velocity field and the pressure field are passed as flat 3D textures (w, p).

As custom obstacle objects can be placed inside the fluid, boundary conditions need special handling. Pressure needs pure Neumann conditions, which means that the gradient of the pressure field along the surface normal of the obstacle object should be zero. Velocity needs free slip boundary condition, which means that voxels near boundaries should take over the velocity of the obstacle object along its surface normal.

```
float4 projection_PS ( VS_OUT IN, //vertex shader output
                       uniform samplerRECT w, // velocity field
                       uniform samplerRECT p //pressure field
                      ) : Color
{
    // read center and neigboring pressure values
    // neighbor values needed for pressure gradient calculation
    float pC = texRECT(p,IN.texCoord.xy).r;
    float pL = texRECT(p,IN.texCoord.xy + float2 (-1, 0)).r;
    float pR = texRECT(p,IN.texCoord.xy + float2 (1, 0)).r;
    float pT = texRECT(p,IN.texCoord.xy + float2 (0, 1)).r;
    float pB = texRECT(p,IN.texCoord.xy + float2 (0, -1)).r;
    float pU = texRECT(p,IN.coord2.xy).r;
    float pD = texRECT(p,IN.coord2.zw).r;

    // read center and neigboring velocity values
    // neighbor values needed to force free slip boundary conditions
    // on solid object boundaries
    float4 wC = texRECT(w,IN.texCoord.xy);
    float4 wL = texRECT(w,IN.texCoord.xy + float2(-1, 0));
    float4 wR = texRECT(w,IN.texCoord.xy + float2(1, 0));
    float4 wT = texRECT(w,IN.texCoord.xy + float2(0, 1));
    float4 wB = texRECT(w,IN.texCoord.xy + float2(0, -1));
    float4 wU = texRECT(w,IN.coord2.xy);
    float4 wD = texRECT(w,IN.coord2.zw);

    // velocity and velocity mask forced by free slip boundary conditions
    float3 obstV = float3 (0,0,0) ;
    float3 vMask = float3 (1,1,1) ;
    // 1. eliminate incorrect pressure gradient terms
    //    (ie: pressure values in solid object should be ignored)
    // 2. calculate velocity forced by free slip boundary conditions
    if (wL.a == 1){
     pL = pC; obstV.x = wL.x; vMask.x = 0;}
    if (wR.a == 1){
     pR = pC; obstV.x = wR.x; vMask.x = 0;}
    if (wL.a == 1){
     pB = pC; obstV.y = wB.y; vMask.y = 0;}
    if (wL.a == 1){
     pT = pC; obstV.y = wT.y; vMask.y = 0;}
    if (wL.a == 1){
     pU = pC; obstV.z = wU.z; vMask.z = 0;}
    if (wL.a == 1){
     pD = pC; obstV.z = wD.z; vMask.z = 0;}

    // calculate pressure gradient
    float3 gradP = float3 (pR - pL, pT - pB, pU - pD);
    // do projection step
    float3 u = wC.xyz - gradP;
    // velocity correction due to free slip boundary conditions
    u = (vMask * u) + obstV;

    return float4 (u, 0);
}
```

Listing 2.7: Fragment shader of the projection operator.

2.6.6 Early-Z culling

The basic idea behind early-z culling is to reduce the amount of computation in regions that have little or no pressure. In small pressure regions fewer Jacobi iteration steps are sufficient while computing pressure. To enable varying iteration count per fragment, dynamic flow control is needed. Shader model 2 and earlier GPU shaders do not support true dynamic branching — and even in Shader model 3 GPUs dynamic branching can have high costs — so we need to achieve this feature in special ways. One way is to use the depth test to skip the fragments that do not meet a condition. Once the fragment shader writes the condition associated with this fragment into the depth buffer. Then The expected condition is the output of all fragments. This we the z-buffer will ignore texels not meeting the condition before the actual fragment shader program is executed.

To speed up Jacobi iteration, before the first iteration step we store a condition value for each fragment in the depth buffer. This value is proportional to the pressure values of the fragment and its neighborhood. In each iteration, we set the depth value of the rendered quads higher and let the depth test compare it to the value stored in the depth buffer, which culls out the fragments that has low pressure and do not need more iterations. This way in each iteration step more and more fragments will be culled saving us computational power.

Listing 2.8 shows the fragment shader that stores the initial depth buffer values used for early-z culling. The pressure field is passed as a flat 3D texture. The output depth is calculated from the maximum pressure value of the voxel and its neighbors.

```
void writePressureAsDepth(VS_OUT IN, //vertex shader output
                          // pressure filed
                          uniform samplerRECT pressure,
                          // color output: copied pressure field (required by our
                                implementation)
                          out float4 outPressure : COLOR0,
                          //depth output calculated from pressure
                          out float depth : DEPTH)
{
    // read center and neighbor pressure values
    float4 C = texRECT(pressure, IN.texCoord.xy);
    float4 cL = texRECT(pressure, IN.texCoord.xy + float2(-1, 0));
    float4 cR = texRECT(pressure, IN.texCoord.xy + float2(1, 0));
    float4 cT = texRECT(pressure, IN.texCoord.xy + float2(0, 1));
    float4 cB = texRECT(pressure, IN.texCoord.xy + float2(0, -1));
    float4 cU = texRECT(pressure, IN.coord2.xy);
    float4 cD = texRECT(pressure, IN.coord2.zw);

    // return center value as color (simple copy)
    outPressure = C;
    // get maximum of the sampled pressure values
    float maxC = max(max(C.r,max(cL.r,cR.r)),max(max(cT.r,cB.r),max(cU.r,cD.r)));
    // return a depth value calculated from the pressure maximum
    depth = saturate(2.0 * maxC + 0.1);
}
```

Listing 2.8: Fragment shader that writes depth values for early-z culling.

2.6.7 External Forces

The influence of external forces should be added to the velocity field after the advection step. If the velocity of external forces are stored in a flat 3D texture, this addition can be simply done with textured full screen quad rendering and additive blending. Listing 2.9 shows the fragment shader used in textured full screen quad rendering. The vector field that should be added to the bounded render target — which is also a flat 3D texture — is passed as a flat 3D texture. The values can be re-mapped (i.e. scaled and offset) before addition.

```
float4 textured_PS(
            // window space coordinates
            float2 texCoord : TEXCOORD0,
            // texture to use
            uniform samplerRECT colorMap,
            // offset and scale value to remap the texture
            uniform float2 transform
           ) : Color
{
    // all channels are mapped equally
    return texRECT(colorMap, texCoord) * transform.x + transform.y;
}
```

Listing 2.9: Fragment shader of a textured quad.

A special type of external force is vorticity confinement. It is used to add back the small scale features damped out by the advection procedure. Listing 2.10 shows the fragment shader used to calculate the vorticity of the velocity field. From the vorticity field the velocity due to small swirling forces can be calculated and later added to the velocity field. Listing 2.11 shows the fragment shader that calculates the vorticity force.

```
float4 vorticity_PS ( VS_OUT IN, //vertex shader output
                      // velocity field
                      uniform samplerRECT velocity
                      ):Color
{
    // read velocity values in neighboring voxels
    float4 L = texRECT(velocity, IN.texCoord.xy + float2(-1, 0));
    float4 R = texRECT(velocity, IN.texCoord.xy + float2 (1, 0));
    float4 T = texRECT(velocity, IN.texCoord.xy + float2 (0, 1));
    float4 B = texRECT(velocity, IN.texCoord.xy + float2 (0, -1));
    float4 U = texRECT(velocity, IN.coord2.xy);
    float4 D = texRECT(velocity, IN.coord2.zw);
    //compute and return the vorticity of the velocity field
    float3 vorticity = ((T.z - B.z) - (U.y - D.y),
                        (R.z - L.z) - (U.x - D.x),
                        (R.y - L.y) - (T.x - B.x));

    return float4 ( vorticity , 0);
}
```

Listing 2.10: Shader that computes the vorticity of the velocity field.

```
float4 vorticityForce_PS ( VS_OUT IN,//vertex shader output
                           // vorticity field
                           uniform samplerRECT vorticity,
                           //time step
                           uniform float dt
                           ):Color
{
    // vorticity scale (this can be a user adjusted parameter)
    float scale = 3.5;
    // read vorticity values in neighboring voxels
    float4 L = abs(texRECT(vorticity, IN.texCoord.xy + float2(-1, 0)));
    float4 R = abs(texRECT(vorticity, IN.texCoord.xy + float2 (1, 0)));
    float4 T = abs(texRECT(vorticity, IN.texCoord.xy + float2 (0, 1)));
    float4 B = abs(texRECT(vorticity, IN.texCoord.xy + float2 (0, -1)));
    float4 U = abs(texRECT(vorticity, IN.coord2.xy));
    float4 D = abs(texRECT(vorticity, IN.coord2.zw));
    float4 C = texRECT(vorticity, IN.texCoord.xy);
    // calculate gradient of the vorticity field
    float3 gradVort = float3(R.x - L.x, T.y - B.y, U.z - D.z);
    // calculate and return the velocity due to vorticity
    float3 V = 0;
    float3 vorticityForce = 0;
    if(dot(gradVort,gradVort))
    {
        V = normalize(gradVort);
        vorticityForce = scale * cross(V, C);
    }
    return dt * float4 ( vorticityForce , 0);
}
```

Listing 2.11: Fragment shader that computes the vorticity force.

2.6.8 Final Rendering

We render the fluid using a volume slicing method. We place quads inside the volume and render them in back to front order using alpha blending. The vertex shader scales these quads to properly enclose the volume and rotates them to always face to the camera. The object space coordinates of these quads give the three dimensional indices we should sample the density field with. As the density field is stored as a flat 3D texture, we should do the interpolation along the z axis manually. The proper indexing of the flat 3D texture is done with the use of the *Layer Offset Texture* described in section 2.6.1.

```
struct VS_OUT {
    float4 hPos : POSITION;
    float4 texCoord : TEXCOORD0;
};

VS_OUT slices_VS( float4 position : POSITION,
                  float4 texCoord : TEXCOORD0,
                  uniform float4x4 ModelView : state.matrix.modelview,
                  uniform float4x4 ModelViewProj : state.matrix.mvp,
                  uniform float4x4 invModelView : state.matrix.modelview.inverse,
                  uniform float4x4 Projection : state.matrix.projection,
                  uniform float3 gridscale, // scale
                  uniform float3 gridcenter // translate )
{
    VS_OUT OUT;

    float4 camPos = position; // center of the grid
    camPos.xyz += float3(texCoord.xyz); // make a camera aligned quad
    OUT.hPos = mul(Projection, camPos); // perspective projection
    OUT.texCoord = mul(invModelView, camPos); //remove camera transform
    // calculate grid coordinates : (0,0,0)--(1,1,1)
    OUT.texCoord -= float4(gridcenter, 0);
    OUT.texCoord *= float4(gridscale,1);
    OUT.texCoord = OUT.texCoord * 0.5 + 0.5;
    OUT.texCoord.z = 1 - OUT.texCoord.z;

    return OUT;
}

float4 slicesFlat3D_PS(VS_OUT IN,
                       uniform samplerRECT density, // density field
                       uniform samplerRECT offsetTexture, // helper texture
                       uniform float3 resolution // resolution of the grid ):COLOR
{
    // discard if we are outside the grid
    if(length(IN.texCoord.xyz - saturate(IN.texCoord.xyz)) > 0)
        discard;
    // move (0,0,0)--(1,1,1) to (1,1,1)--(resX+1, resY+1, resZ+1)
    float3 coord3D = IN.texCoord.xzy * resolution + 1;
    // do the interpolation along the z axis manually
    float l1 = floor(coord3D.z);
    float l2 = l1 + 1;
    float2 offset1 = texRECT(offsetTexture, float2(l1, 0.5)).rg;
    float2 offset2 = texRECT(offsetTexture, float2(l2, 0.5)).rg;
    float4 sample1 = texRECT(density, offset1 + coord3D.xy);
    float4 sample2 = texRECT(density, offset2 + coord3D.xy);
    float4 d = lerp(sample1, sample2, coord3D.z - l1);

    d = float4(1,1,1, d.r);
    return d;
}
```

Listing 2.12: Final volume rendering.

Chapter 3

Parallel Implementation

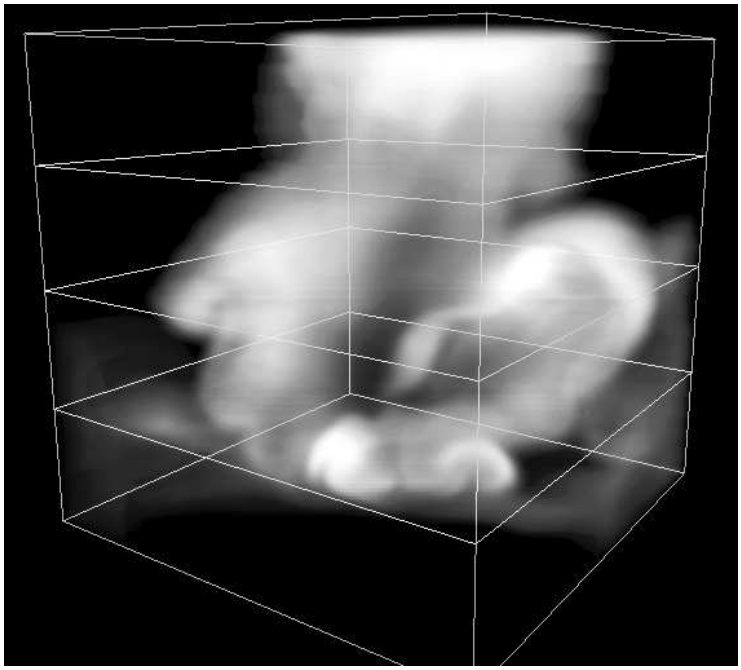

Figure 3.1: Volume rendered with our distributed rendering algorithm. The wire cubes show one part of the grid which is simulated on one node of the cluster.

Because of the data storing needs and computational cost described so far one can easily run into a point where a single computer implementation becomes insufficient. Using flat 3D textures has a disadvantage that they limit the size of the grid that can be simulated, as there is a limited texture size. Tiling the grid slices can easily result in a flat 3D texture with too high resolutions. The maximum resolution of the grid that can be computed on a Shader Model 3 GPU is $254 \times 254 \times 254$. We should note here that this grid resolution requires 4096×4096 flat 3D texture resolution since we need two extra slices and one extra pixel border in each slice for the boundary conditions.

One can step over this problem by subdividing the grid into smaller parts and store

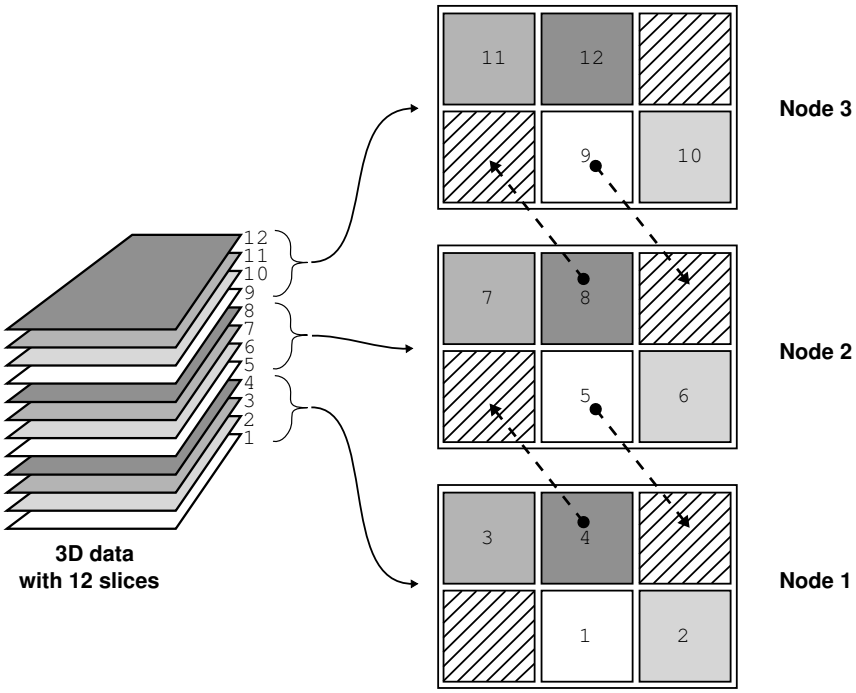

Figure 3.2: Data communication between the nodes of the cluster.

them in separate flat 3D textures, but can still run out of the GPU memory. On the other hand, the high computational cost of simulating high resolution grids does not allow interactive frame rates. The solution of these problems is the application of distributed GPGPU simulation and rendering.

We implemented the algorithm on a GPU cluster. This is a shared memory parallel rendering and compositing environment that uses the *ParaComp library*[1] of the *HP Scalable Visualization Array*. The main aspect of this library is that each host contributes to the pixels of rectangular image areas, called *framelets*. The process of merging the pixels of framelets into a single image is called *compositing*. The ParaComp library allows not only the framelet computation but also the composition to run in parallel on the cluster. In our implementation one node takes the role of the master, which means that it has additional tasks beside rendering and compositing. The master sets the proper order of the nodes for composition and displays the composited image on screen. All nodes (including the master) do simulation steps on one portion of the data set, render the subvolume, and contribute its image as one framelet. The composition of framelets is done with alpha blending in a parallel way using the *parallel pipeline algorithm* [5].

We used object space distribution of the 3D data (see Figure 3.1). We subdivided the grid and each part can be treated as a separate data set. Each node in the distributed system should send the adjacent grid slices to its neighbors as a boundary condition to be used.

In Figure 3.2 the dashed arrows show the communication between the nodes in

[1] `http://sourceforge.net/projects/paracomp`

case of a three node system. For example, the topmost node sends its lowest layer to its lower neighbor, which puts this data to its topmost boundary condition layer. This neighboring node also sends its topmost slice to its upper neighbor, which puts this data in its lowest boundary condition slice. Three data should be transferred: the adjacent slices of the velocity, pressure, and the display variable fields. We used an MPI implementation for boundary layer communication between nodes.

As we used a slicing rendering method to display the volume we should only ensure that the images produced by each node are composited in back to front order with alpha blending.

Chapter 4

Installation and Usage of Gridfluid

4.1 Installation

Both the source and the pre-built versions of the application can be found on the web site of the project[1].

4.1.1 Library dependencies

The following libraries are required by the volume rendering application:

- **paracomp**: Hewlett Packard implementation of the Parallel Compositing API (version 1.0-beta1 or later)
- **glew**: OpenGL Extension Wrangler library (version 1.3.4 or later)
- **Cg** and **CgGL** : NVIDIA Cg library
- **gl**: library implementing OpenGL API
- **glu**: OpenGL Utility Library
- **glut**: OpenGL Utility Toolkit

There are prebuilt packages for HP XC V3.2 RC1 platform for AMD64 architecture on the web site of the project for OpenGL Extension Wrangler, Cg, CgGL libraries. If one of them is missing from the target system, it can be installed in the usual way using the `rpm` package manager program:

```
# rpm -i glew-1.3.4-1.x86_64.rpm
# rpm -i glew-devel-1.3.4-1.x86_64.rpm
# rpm -i Cg-1.5.x86_64.rpm
```

The `XXX-devel-YYY.rpm` packages are only needed when the fluid simulator application is built from sources. Otherwise, only the shared libraries are to be installed.

[1] `http://amon.ik.bme.hu/gridfluid/`

The other libraries like the Parallel Compositing library, the standard C/C++ libraries, and the OpenGL libraries are platform specific and have to be installed based on the actual software stack.

4.1.2 RPM Package

The 3D grid fluid simulator (*gridfluid*) can be also installed from a prebuilt RPM[2] package in the same way:

```
# rpm -i gridfluid-0.1-1.x86_64.rpm
```

After install the necessary application files are placed in

```
# usr/local/gridfluid
```

The single computer version of the application, which does not depend on the ParaComp and MPI libraries can be installed similarly:

```
# rpm -i gridfluid_singlepc-0.1-1.x86_64.rpm
```

After install the necessary application files are placed in

```
# usr/local/gridfluid_singlepc
```

Both applications can be build from sources in the usual way using the provided make files.

4.2 Usage

4.2.1 Program Execution

After installing *gridfluid_singlepc*, it can be started with the following command:

```
$ gridfluid [OPTION]
    -res RES1xRES2xRES3      sets the grid resoltion
                             RES1, RES2, RES3 define the
                             resolution along the three axes
    -obj OBJECTFILE          sets obstacle object filename
                             OBJECTFILE is the name of the
                             object file
                             file should be in Wavefront object
                             format (.obj)
```

The object file given by the *-obj* parameter should be located in the following directory:

[2]Red Hat Package Manager

```
# usr/local/gridfluid_singlepc/media/models
```

Only a special type of Wavefront object files are supported. The file should contain a single solid triangulated mesh without any material information and should have the following file structure:

```
v x y z
..
v x y z
vt u v
..
vt u v
vn x y z
...
vn x y z
f vIdx/tIdx/nIdx vIdx/tIdx/nIdx vIdx/tIdx/nIdx
...
f vIdx/tIdx/nIdx vIdx/tIdx/nIdx vIdx/tIdx/nIdx
```

where *x, y, z* stand for the three components of the vertex positions and the normal vector, *u, v* stand for the texture coordinates of the vertices and *vIdx, tIdx, nIdx* stand for the vertex coordinate, texture coordinate, and normal vector indices, respectively.

This non-parallel rendering mode cannot be applied for simulation of grids with high resolution, because of both the memory and performance requirements. The parallel version of the simulator (*gridfluid*), that overcomes these problems have a slight difference in the way of execution.

A SLURM[3] startup script is provided to use *gridfluid* for parallel rendering. It can be invoked with the following command:

```
$ gridfluid.sh -r <renderers> -res <resolution> -obj <object file>
```

The startup script has tree parameters that should be set. The first one (*-r*) tells SLURM the number of *additional render nodes* to be allocated. The later ones (*-res* and *-obj*) have the same meaning as in case of the non parallel version of the application.

[3]SLURM is an abbreviation for Simple Linux Utility for Resource Management. It is an open-source resource manager designed for Linux clusters of all sizes. This software solution is used for HP-XC clusters.

4.2.2 User Interface

The application shows the different parameters of the fluid simulation, external force addition, and the keyboard shortcuts to adjust them on the display (see Figure 4.1). The keyboard shortcuts and their affects are the following:

- Esc Quits the program.
- Space Toggles the user interface.
- Q Toggles display mode (only available in the non parallel version). Available modes:
  - 3D density view,
  - velocity field,
  - velocity divergence field,
  - pressure field,
  - density field,
  - vorticity field,
  - vorticity force field.
- Z Turns early-z culling on/off.
- O Turns obstacle object on/off.
- I Displays/hides the obstacle object.
- C Displays/hides bounding cube of the simulated grid.
- W Turns vorticity confinement on/off.
- + / - Increases/decreases iteration count used in Jacobi iterations.
- K / L Increases/decreases number of volume slices used in final rendering. A higher number results higher quality. If this value is zero, the number of slices is determined automatically based on the resolution of the simulated grid.
- N / M Increases/decreases simulation time step. If this value is zero, the time step is set to the current frame time. If this value is too high, simulation errors can occur.
- H / J Increases/decreases viscosity.
- PageUp / PageDown Increase/decrease the injected density.

- [A] Toggles automatic external source type. Available modes are:
  - density is injected into a small spot at the bottom of the grid,
  - density is injected in a line of small spots,
  - density is injected in a grid of small spots,
  - density is injected by the user.
- [X] Turns wind tunnel force on/off. Wind tunnel force has an upward direction. If wind tunnel force is turned off, automatic sources inject upward velocities as well as densities.
- [←] / [→] / [↑] / [↓] Moves the small automatic sources on the x, y plane.
- [Y] Flips auto source line direction (it can be oriented along the x or along the y axis).
- [V] Clears the velocity field to zero.
- [D] Clears the density field to zero.
- [R] Clears both the density and the velocity fields to zero.

In 3D view the user can navigate with the following controls:

- [Left Mouse Button] + Mouse Drag Moves the camera.
- [Left Mouse Button] + [Middle Mouse Button] + Mouse Drag Zooms the camera.

If no auto sources are activated, the user can interactively add velocity and density into the fluid with the following controls:

- [Right Mouse Button] + Mouse Drag Adds density and velocity in the mouse movement direction from the center of the grid.
- [Middle Mouse Button] + Mouse Drag Adds velocity in the mouse movement direction from the center of the grid.

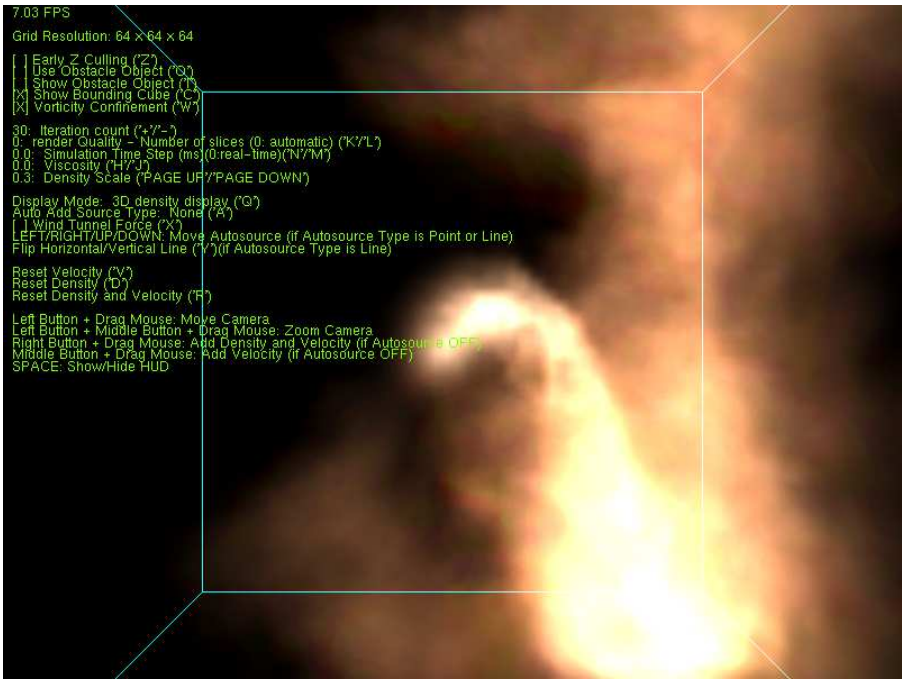

Figure 4.1: User interface of the fluid simulator.

Chapter 5

Results

For the experiments we used a *Hewlett-Packard's Scalable Visualization Array* consisting of five computing nodes. Each node has a dual-core AMD Opteron 246 processor, an nVidia Quadro FX3450 graphics controller, and an InfiniBand network adapter. One node is only responsible for compositing and managing the framelet generations and does not take part in the rendering processes, so we could divide our data set into maximum four parts.

Table 5.1 shows our results. It can be seen that using 2 nodes gives better performance than the single computer version. Using more than 2 nodes becomes useful only in case of larger data sets, for small data sets the communication between the nodes becomes a bottleneck. The N/A sign means that a $256 \times 256 \times 256$ data set cannot be simulated on a single computer since it has too high memory needs.

It is worth examining the resolutions of $64 \times 64 \times 64$ and $80 \times 80 \times 80$. The 80 resolution needs about twice as many voxels as the 64 one. It can be clearly seen that the performance of the two node implementation is the double of the performance obtained on a single node. Similarly the two node implementation nearly doubles performance in almost all cases.

Grid resolution	1 node	2 node	4 node
$32 \times 32 \times 32$	25 FPS	27 FPS	20 FPS
$64 \times 64 \times 64$	8 FPS	13 FPS	15 FPS
$80 \times 80 \times 80$	4 FPS	10 FPS	13 FPS
$100 \times 100 \times 100$	2 FPS	6 FPS	9 FPS
$64 \times 64 \times 128$	3 FPS	10 FPS	11 FPS
$128 \times 128 \times 128$	2 FPS	2.7 FPS	5.5 FPS
$64 \times 64 \times 256$	1 FPS	6 FPS	8 FPS
$256 \times 256 \times 256$	N/A	5 sec/frame	2.35 sec/frame

Table 5.1: Performance results.

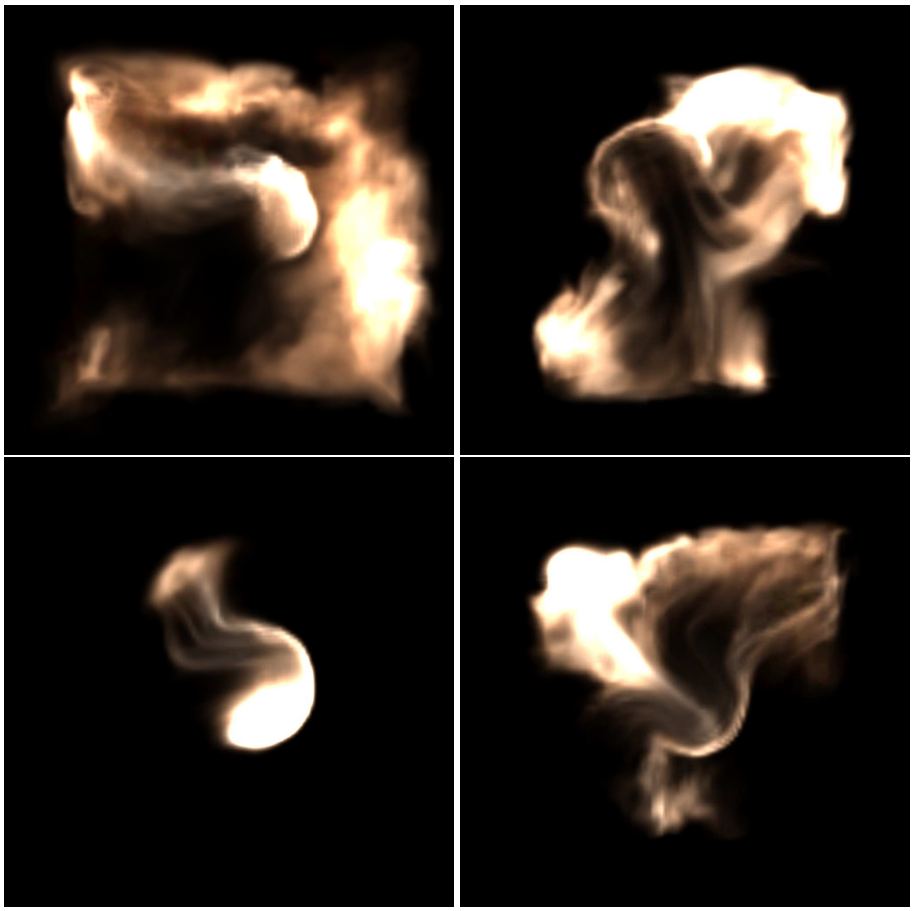

Figure 5.1: Volumetric media simulated on a $75 \times 75 \times 75$ grid.

Bibliography

[1] CRANE, K., LLAMAS, I., AND TARIQ, S. Real-time simulation and rendering of 3D fluids. In *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, 2007, pp. 633–675.

[2] FEDKIW, R., STAM, J., AND JENSEN, H. W. Visual simulation of smoke. In *ACM SIGGRAPH 2001* (2001), pp. 15–22.

[3] HARRIS, M. Fast fluid dynamics simulation on the gpu. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), ACM Press, p. 220.

[4] HARRIS, M. J., BAXTER, W. V., SCHEUERMANN, T., AND LASTRA, A. Simulation of cloud dynamics on graphics hardware. In *Eurographics Graphics Hardware'2003* (2003).

[5] LEE, T.-Y., RAGHAVENDRA, C., AND NICHOLAS, J. B. Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics 2* (1996).

[6] PEDRO V. SANDER, N. T., AND MITCHELL, J. L. Early-z culling for efficient gpu-based fluid simulation. In *ShaderX5: Advanced Rendering Techniques*, W. Engel, Ed. Charles River Media, Cambridge, MA, 2006, ch. TBD, p. TBD.

[7] STAM, J. Stable fluids. In *Siggraph 1999, Computer Graphics Proceedings* (Los Angeles, 1999), A. Rockwood, Ed., Addison Wesley Longman, pp. 121–128.