Simulation of Turbulent Flow in Computer Graphics: Applications and Optimizations

Ian Buck
Princeton University, Department of Computer Science
Independent Work for COS 397

Advisor : Perry R. Cook
www.cs.princeton.edu/~ ianbuck/turbulent

January 5, 1998

Abstract

The basis for the first part of this work was to explore what is involved in simulating turbulent flow using an already existing mathematical model. This includes implementing a complete optimized C version of the model and simulating flow models such as acoustical instruments. The second part of this work explores alternative methods of calculation for flow simulation using the graphics hardware as a mathematical engine. We present a few different methods to use the accelerated framebuffer to perform some of the calculations required in flow simulation.

Part I
Simulating Turbulent Flow

1 Introduction

Research in application of fluid flow simulation has been an active topic in computer graphics for over a decade. Such visual effects as simulated smoke, fire, and liquids are all typically based on top of a mathematical model which graphics researchers have been trying to improve through today. There have been endless papers published in the ACM SIGGRAPH proceedings in '97, '95, '93, '91, '90, and continuing back through '86 with a paper dedicated to how the swirling atmosphere of Jupiter was simulated for the movie "2010" (Yeager et al, 1986).

The primary difficulty in creating such visually stimulating effects, such as simple candle smoke is that an intensive computational simulation is required. To provide results which appear accurate, such calculations can be extensive and quite slow. For example, the "2010" Jupiter swirls required a Cray X-MP, while other simulations of candle smoke presented in the paper "Simulation of Turbulent Flow Using Vortex Particles" (INESCA '94) achieved results of 37 sec/frame for suitable results on an SGI Indigo (See Figure 1). Even the most optimized solutions for three dimensional solutions with simulation resolutions of 60x60x45 achieved 49 sec/frame (SGI Indigo2, see Figure 2). These computation times can drastically limit the complexity and resolution of the desired simulation. Furthermore, since it is a computational model, attempts to drive the simulation to faster calculation times can render the results unstable. Any new research which could improve the limits of the simulation is rapidly being explored, and is a focus of this work.

For computer simulations, two different mathematical models have been used for creating realistic turbulent flow. The first method, which was very popular early in the development of flow simulations, relies on a particle based model. Each particle represents a fixed amount of rotation about that point, called "vorticity." Particles are free to move in a simple Newtonian force model determined by the vector field defined by the vorticity (Gamito '94). These simulations can yield visually interesting results with natural swirling motion which appears similar to those found in nature. However, this model is not physically accurate since it not based actual fluid physics

Figure 1: This image was created by Manuel Noronha Gamito, "Simulation of Turbulent Flow Using Vortex Particles" using 200,000 particles in a 2D environment which resulted in a frame rate of 37 sec/frame.

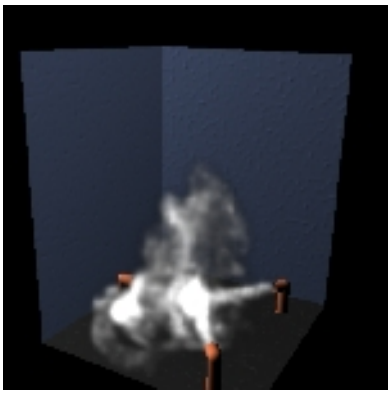

Figure 2: This image was created by Nick Fostern, "Modeling the Motion of a Hot, Turbulent Gas" (SIGGRAPH '97). He used the 3D voxel simulation used in this paper to produce this image of three jets. Frame rates here for 40x50x40 resolution was 49 sec/frame.

and cannot be used for approximating real world environments. Furthermore, interaction with fixed surrounding obstacles is undefined within this model.

The second model proposed for solving turbulent flow applies the actual physical equations which have been used for years by aeronautics engineers and other disciplines who are interested in flow calculations. Work has been done in the area, most recently, by Fostern and Metaxas to make discrete these equations into voxel approximations. By applying the necessary calculations to each voxel box, the fluid flow vectors through a medium can be determined. Not only does applying the techniques proposed provide faster frame rates than previous models, it also provides a physically accurate environment and high flexibility.

2 Navier-Stokes Mathematical Model

The Navier-Stokes equations are a set of rules defined as partial differential equations which govern all forms of fluid flow, including information regarding position, speed, pressure, and temperature. To simplify the equation, we make a certain assumptions about the model. To achieve basic turbulence, our simulation is only interested in simple flow without pressure or thermal buoyancy influencing the motion. This simplifies the equations into a singular primary expression which breaks up into three parts. The first is convection which defines basic transfer of motion; one area flows into another. The second component is drag. Here, flow is slowed by the flow surrounding it. The final component is mass conservation for pressure. Since the medium is considered incompressible, the amount of flow entering into a region must match the amount leaving.

$$\frac{d\mathbf{u}}{dt} = Convection + Drag + Mass\ Conservation$$

$$\frac{du}{dt} = -(u \cdot \nabla)u + \nu\nabla \cdot (\nabla u) - \nabla p$$

ν = Viscosity Constant
u = Velocity
p = local pressure

The constant ν specifies the viscosity of the medium. The higher the viscosity, the more drag there is in the system. The pressure gradient (∇p) allows the pressure of the gas to effect the flow. Since our model assumes that there is

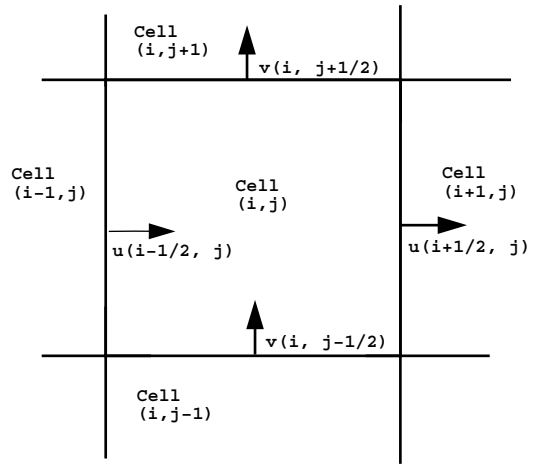

Figure 3: This diagram shows how the Fostern, Metaxas model is defined. The program keeps track of all of the edge velocities where $u_{i+1/2,j}$ is the amount of flow from cell i, j to $i+1, j$.

no compression, this term ensures mass conservation. This partial differential equation can be made discrete through the finite differences approximation, which is defined as:

$$\frac{dF}{dx} = \frac{1}{2h}(F(x+h) - F(x-h) + O(h^2))$$

The second order terms $O(h^2)$ can be assumed zero for the simulation. Applying this to the Navier-Stokes provides a discrete version of the equations which can be implemented in the software.

3 The Model

The basis for the model implemented is largely taken from work done by Fostern and Metaxas. The environment is first broken up into discrete areas. In the two-dimensional case, this can be visualized as a grid, with each grid square having dimension Δx. The simulation keeps track of the flow between adjacent squares by maintaining the "edge" velocities.

For example, $u_{i-1/2,j}$ is the value of flow across the edge shared by square (i, j) and $(i-1, j)$. Likewise $v_{i,j+1/2}$ is the flow from (i, j) to $(i, j+1)$ across their shared edge. Positive values corresponding to upward or to the right motion, therefore a positive value for $u_{i+1/2,j}$ indicates flow leaving square (i, j) and entering $(i+1, j)$. When the calculation requires an intermediate

value which is not directly maintained by the simulation, such as $u_{i,j}$ which is at the center of the square, it is simply calculated as the average of $u_{i-1/2,j}$ and $u_{i+1/2,j}$. (See Figure 3.)

The primary benefit of the Fostern and Metaxas edge scheme is that it provides for fast and efficient implementation of the Naiver-Stokes equations described below. For each time-step, the discrete Navier-Stokes calculations are performed for each grid box using velocities at each of the edges.

3.1 Convection and Drag

The convection and drag components can be calculated quite simply since, in the discrete form, the next time step only relies on the edge values for the immediate neighbors. In the simulation, these two terms are combined and reduced to into one expression for optimal efficiency. A single pass is then made of over all of the squares updating all velocities. Here is the equation for the convection and drag for the $u^{n+1}_{i+1/2,j}$ term.

$$\frac{du}{dt} = \nu\left(\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2}\right) - \frac{du^2}{dx} - \frac{duv}{dy}$$

$$u^{n+1}_{i+1/2,j} = u^n_{i+1/2,j} + \Delta t/\Delta x[(u^n_{i,j})^2 - (u^n_{i+1,j})^2 + (uv^n_{i+1/2,j-1/2})^2$$

$$+(uv^n_{i+1/2,j+1/2})^2 + (\nu/\Delta x)(u^n_{i+3/2,j} - 2u^n_{i+1/2,j} + u^n_{i-1/2,j}$$

$$+u^n_{i+1/2,j+1} - 2u^n_{i+1/2,j} + u^n_{i+1/2,j-1})]$$

3.2 Mass Conservation

Without the mass conservation term, the flow has none of the interesting features of natural turbulent flow. For example, without mass conservation, flow can cancel itself out when collided head on rather than swirling backward as expected. Unfortunately, mass conservation is not as simple to compute as the convection or drag. One way to implement mass conservation is to insure that the amount of flow going into a grid box matches the amount of flow leaving. It can be shown that the gradient of $\mathbf{u}$, the velocity contained within a square, must be equal to zero.

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

To include this in the calculation, the expression for $\nabla \cdot \mathbf{u}$ must be placed into its discrete form:

$$\nabla_{i,j} = u_{i-1/2,j} - u_{i+1/2,j} + v_{i,j-1/2} - u_{i,j+1/2}$$

Here we can see how Fostern and Metaxas' edge scheme can benefit in the calculations since all of the values in this expression are directly maintained within the simulation.

This calculated value ($\nabla_{i,j}$) specifies the amount of difference between the incoming and outgoing flow. Since this is a pressureless simulation, the edge velocities must be modified to reduce this term to zero.

Mass conservation cannot be calculated in a single pass operation similar to the convection and drag terms. For example, if $1/4\nabla_{i,j}$, was subtracted from the left and lower edges and added to the upper and right edges, the gradient would go to zero for that i, j square, however, all of the surrounding squares would no longer have a zero gradient. Therefore, mass conservation is performed through a multipass computation, stopping when the largest gradient is below a certain threshold, as follows:

- Compute all ∇ for all i, j.
- If $\nabla_{i,j} > Threshold$, subtract $1/4\nabla_{i,j}$ from $u_{i-1/2,j}$, $v_{i,j-1/2}$, add to $u_{i+1/2,j}$, and $v_{i,j+1/2}$. Repeat for all i, j.
- If $\nabla_{Max} > Threshold$, repeat from step one.

This scheme is quite simple and provides for basic mass conservation within the model. The 1/4 factor provides for a fast equilibration of mass between each square while remaining stable. (Note that this can be decreased for a smoother distribution with increases in computation time.)

3.3 Errors in Fostern and Metaxas Mass Conservation.

It should be noted that this mass conservation method is not identical to that of Fostern and Metaxas which was presented in their SIGGRAPH '97 paper. Their method used the gradient of $\mathbf{u}$ to calculate a potential field which was maintained separately from the velocities. Using a method derived from work published by Harrow and Welch, they iterated over a classical Poisson equation.

However after a week of testing, I was unable to get their solution stable. Afterward, I discovered from their web site the values provided within the paper lead to unstable results. The modification they suggested would have slowed the calculation further to the point at which I derived my own mass conservation described previously.

3.4 Rendering

While there has been a plenty of work on visualization of smoke and fluids which can be used to represent turbulent flow solutions, the primary focus of this work was to study the only the calculation of turbulent flow. Therefore, a simple graphical representation was chosen. Red areas indicated horizontal movement, while green showed vertical. To further show the effects of the resulting velocity field, massless particles are introduced to the simulation to show the actual motion described by the velocities.

3.5 Boundary Conditions

Since the algorithm is largely dependent on neighboring values, there is an immediate question regarding what is meant to happen at the boundaries. In almost all of my simulations, I allowed flow to simply fall of the end of the screen. This was done by detecting when the calculation was at the edge and using values for the (i, j) square in place of the $(i + 1, j)$ which did not exist. Although this does not provide entirely accurate results at the edge of the screen, it keeps the simulation stable and does not disturb the surrounding squares.

3.6 Flow Algorithm

The complete simulation algorithm done in a four stages: environment conditions; drag and convection calculations; mass conservation; and rendering.

- 1) First set any fixed conditions with in the simulation. For example, if there is a jet at position (a, b), set the proper values for $u_{a,b}$.
- 2) Update all of the flow values using the drag and convection calculation. Use the proper values at the boundaries.

- 3) Perform the mass conservation calculations. Calculate the gradients at each (i, j). Update the edge velocities using the gradient. Repeat until all gradients are below a specified threshold.
- 4) Update all partial motions and render grid square to the screen with the proper red and green components.

4 Basic Turbulent Flow

With a completed model, a flow simulation was done and tested. There are three main parameters within the model: Δx: Width of the grid squares, Δt: the time step between frames, and ν which is the viscosity of the medium. Also there are threshold values for the mass conservation however this does not directly modify the outcome only the accuracy and calculation time. The simplest example of the turbulent flow which illustrates the interesting swirling motion and vortices, a jet is placed in the center of the screen and at every frame is set to a fixed value. (See Figure 4.) Also, larger jets an introduce stronger vortices. (See Figure 5.)

4.1 Stability

The stability of the algorithm is dictated by the assumptions made in the discrete form of the Navier-Stokes equations. To allow the $O(h^2)$ to be assumed zero, the velocities must not allow flow to travel more than one grid square within a time step.

$$max(u, v) < \Delta x/\Delta t$$

Although this constraint seems quite simple, it can be difficult to maintain by only controlling initial conditions since turbulent flows can lead to all sorts of velocities. Furthermore, from linear analysis of the Navier-Stokes equations, it has been shown that the viscosity is also constrained:

$$\nu > (\Delta t/2)max(u^2, v^2)$$

As long as these two equations are satisfied, the simulation will remain stable. To aid in detecting instabilities in the simulation, the velocity values are checked at render time to see if they are found to be outside of acceptable limits. If they are unstable, a predefined blue color is drawn instead of the proper color value.

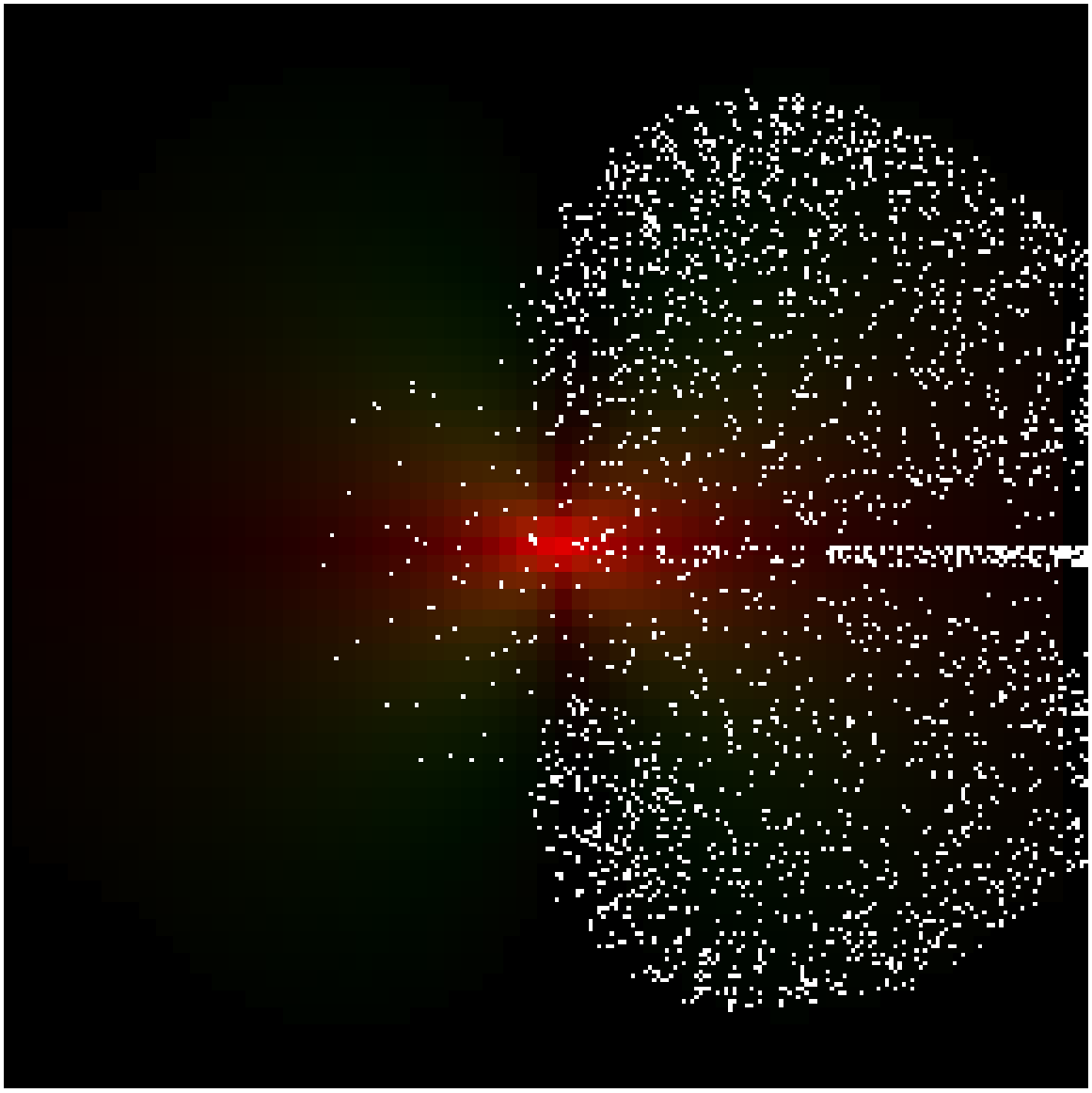

Figure 4: This image was produced after 250 iterations of the turbulent flow algorithm. The red areas indicate horizontal motion, while the green is vertical. In the center of the image there is a jet which is set to a positive 5.0 every frame. This was a 64x64 simulation, Frame rate: 0.15 sec/frame ($\Delta X = 0.05$ $\Delta t = .001$ $\nu = 0.5$)

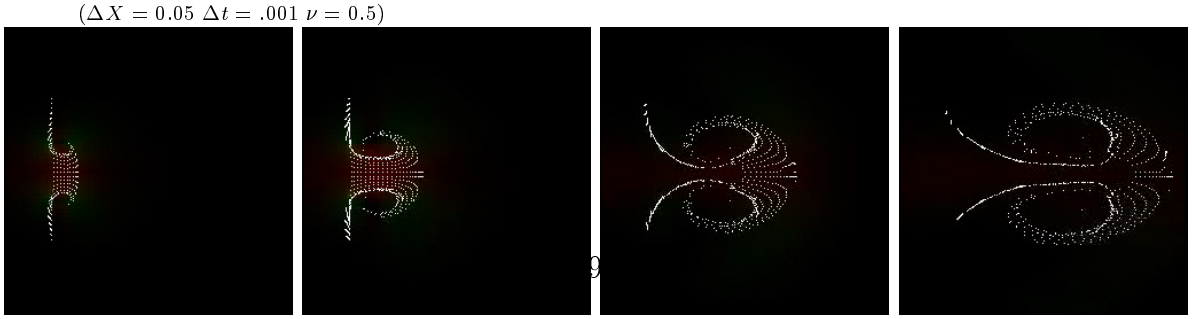

Figure 5: This animation shows the swirling nature of the gas as a jet is briefly applied. The vortices formed is a direct result of the mass conservation calculations.

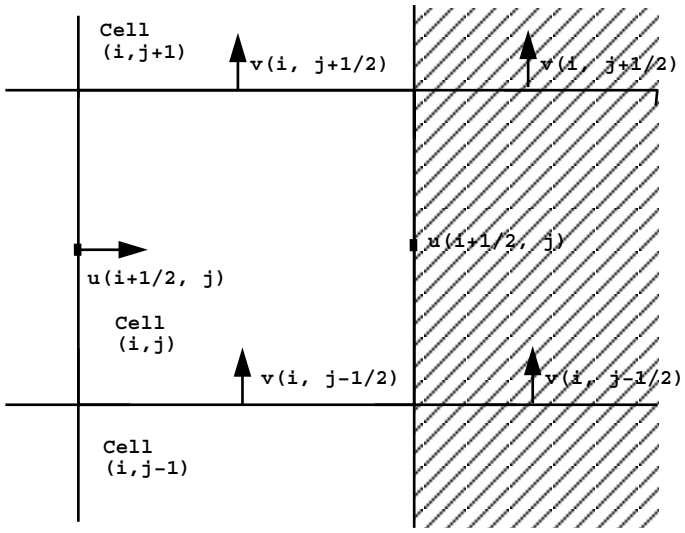

Figure 6: To include obstacles into the simulation model requires a special case on squares next to the walls. To prevent flow from going into walls, we set the u value to zero. To prevent drag, we set the v value for the wall equal to square adjacent when were doing the calculation.

5 Acoustical Models

An interesting application of this work is the study of flow within acoustical instruments. If the model is accurate, a simple instrument could be modeled to obtain an oscillating flow which could produce simulated sound waves.

5.1 Walls

Before attempting to model an instrument, the simulation model must be modified to include how the flow interacts with the walls of the instrument. The Fostern and Metaxas edge model has the flexibility to include fixed obstacles by setting the edge values to match those of the neighboring grid squares. For example, in updating the value for $u_{i+1/2,j}$ with a wall in location $(i+1, j)$, $u_{i+1/2,j}$ should be set to zero since there cannot be any flow into the wall. Furthermore in calculating the $v_{i,j+1/2}$ term, the value for $v_{i+1,j+1/2}$ of the wall should be set equal to $v_{i,j+1/2}$ so that no drag is created from the wall. (See Figure 6.)

Most of these cases can be handled in parallel with the regular drag and convection calculations. Furthermore for the mass conservation, 1/3 of the gradient is subtracted from the edge values if one of them shares a wall, 1/2

for two walls, etc.

5.2 Instrument: Basic Whistle

In order to design a working whistle, there are three different characteristics which must all be correct for an oscillation to occur at the opening. The size of the whistle chamber, size and positioning of the opening, and the size and amount on input flow, all define how the oscillation will occur.

To help in designing a working whistle, the implementation was rewritten to take a PPM file as input. Blue pixels indicated walls, red and green colors specified jets, while dark blue indicated where points where the velocity was sampled and outputed into a sound file. Using this modification, whistles could be designed with a simple editor, like xpaint, and tested. The frame rates of the simulation where high enough such that it was clear within a few minutes whether a stable oscillation was present.

In general, there were no direct methods for designing the whistle other than guesswork. The goal was primarily to have flow oscillation such that initially the majority of the flow would go inside the whistle and eventually come around to push the original source out the opening. This would weaken the flow in the whistle chamber to the point which it could no longer divert the original stream. This cycle would repeat itself and an oscillation would be present.

After plenty of trial and error, the simplest model proved to be the most successful. This single pixel input had a relatively small 16 pixel chamber with 3 pixel opening. The sound wave produced an oscillation which as nearly sinusoidal. Furthermore, to test the significance of the viscosity, ν was increased, increasing the simulated thickness of the medium, and the pitch of whistle dropped as expected. (See Figure 7.)

6 Discussion

The goal of the first part of this work was to implement a traditional turbulent flow simulation and test its capabilities. By correcting the model done by Fostern and Metaxas, this working model was accurate enough to create accurate simulation environments. The results of the oscillating whistle shows how this flow solution can be used for modeling actual systems.

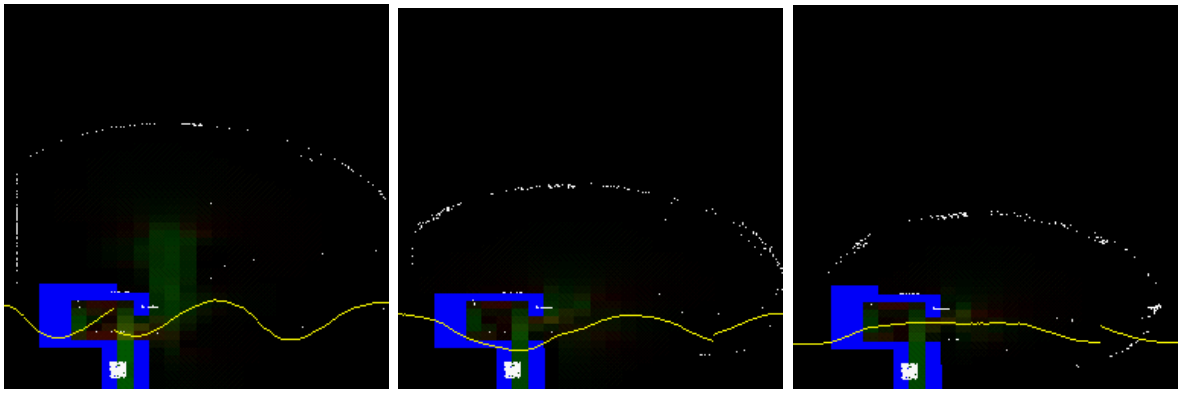

Figure 7: These images show a working whistle. The blue pixels indicate walls in the simulation while the yellow line is a graph of the velocity at the opening of the whistle over time. Here the different values of ν effected the pitch: $.002, .0025, .003$

Like all previous work done in this area, speed is always an issue. The framerates in the examples given were fast enough to watch it evolve in real time. However, increasing the resolution of the simulation can drastically slow the framerate. The second part of this work explores different methods of calculating the flow solution.

Part II
Alternative Calculation

7 Introduction

A large extent of this work explores different solutions to the work done by Foster and Metaxas. We examine alternative methods of calculating frames using graphics hardware for improved performance, an area which has been largely overlooked in the research done in flow simulation. The capabilities of graphics hardware on most workstations and even common personal computers is rapidly increasing. We show that it is possible to tap this computing resource to perform the necessary calculations for flow simulations as done in part I. Using basic imagining API provided by OpenGL, these physical calculations can be performed in parallel through the dedicated hardware. This can increase performance of these simulations, in some cases, drastically but still provide the necessary flexibility for a working solution.

Also, related to the graphics hardware extensions, this work examines the significance of number representation and its effects on the mathematical model. The simulation is also confirmed to be physically accurate by testing a acoustical model.

8 Graphics Parallels: Temperature Diffusion

There are clear parallels in the types of computation which occurs in physical simulations and computer graphics. The simplest example is temperature diffusion. This partial differential equation for diffusion is identical to the drag coefficient of the fluid model.

$$\frac{dT}{dt} = \lambda \nabla \cdot (\nabla T)$$

$$where\ \lambda = Thermal\ Conductivity$$

This equation describes how temperature disperses through a fixed medium, like a block of metal. This can be made discrete using the same finite differences method used for the Navier-Stokes equations, resulting in the expression

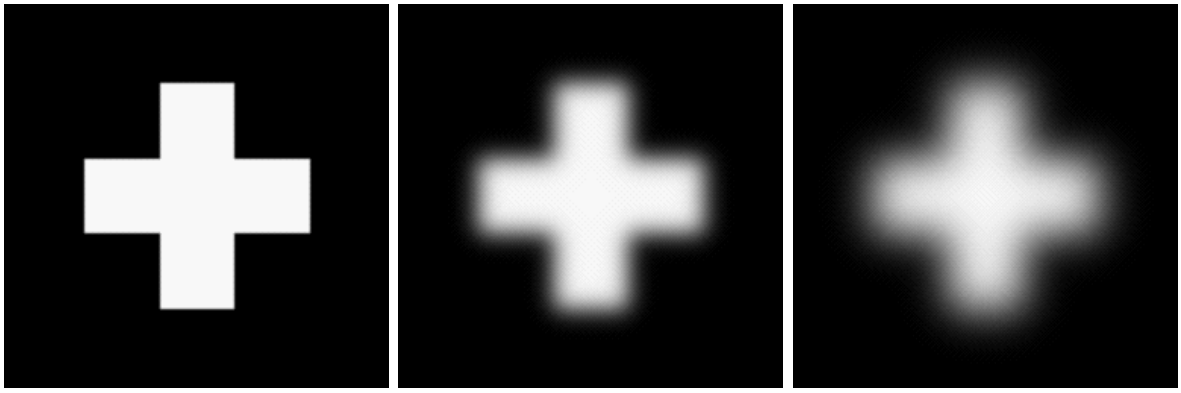

Figure 8: Here are the results of the temperature diffusion showing shots of the initial conditions, after 150 frames, and 500 frames. The intensity indicates the heat values.

$$T_{i,j}^{n+1} = T_{i,j}^{n} + k(T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n} + T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n})$$

$$where\ k = \frac{\Delta t \lambda}{\Delta x^2}$$

Using the discrete equation, the simulation can be run showing a temperature field diffusing into the surrounding medium. The luminance indicates the intensity of the heat. See Figure 8.

However without looking at the underling equations, these images appear to be a simple blurring filter. This observation suggests that simple graphics related computation could be used to do the calculations required for the simulation. In fact, the discrete equation can be rewritten into a simple convolution matrix which is familiar to graphics programmers.

$$T_{i,j}^{n+1} = \begin{pmatrix} 0 & k & 0 \\ k & 1-4k & k \\ 0 & k & 0 \end{pmatrix} T_{i,j}^{n}$$

9 Hardware Computation and Limitations

9.1 Graphics Parallels

All of the calculations involving Navier-Stokes and temperature diffusion are based upon partial differential equations involving the same mathematical

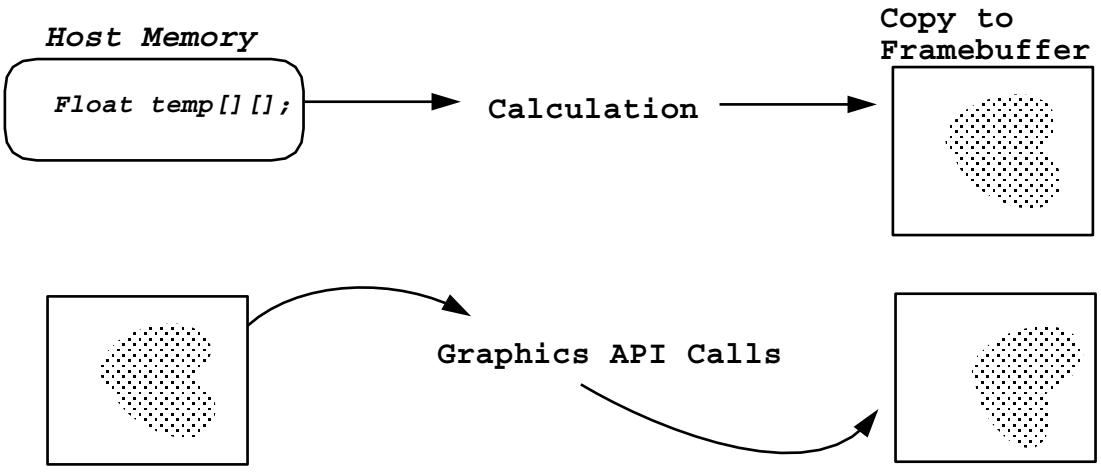

Figure 9: The inefficiency of the classical model is that two copies of the data are represented in the host memory and on the framebuffer. One is updated and then extra CPU cycles must be spent updating the frame buffer. To remove this inefficiency, we implement a method to perform all of the calculations within the framebuffer using the graphics API.

operations across each of the sample areas in the simulation. Ideally, we would like the graphics hardware to perform all of these calculations in parallel by keeping the data all inside the framebuffer and manipulating through graphics calls. See Figure 9.

Basic graphics operations, such as blending or filtering, all can be used to provide basic algebraic calculations. OpenGL provides a hardware independent API for graphics operations which can be used for these purposes.

```
glConvolutionFilter2D: Apply a nxn convolution
                       matrix across each pixel.
glMatrixMode(GL_COLOR_MATRIX): Apply a color matrix
                               across each pixel.
glBlendFunc: Modify blending to perform
             multiplication or division.
glAccum: Use the accumilation buffer to tabulate
         results.
```

There are many advantages to doing these calculations inside of the framebuffer. Primarily, the graphics hardware can accelerate the simulation since the host CPU doesn't need to be tied up. Furthermore, no extra rendering time needs to be spent placing the data onto the screen since its already

present in the framebuffer. Since the hardware is already there to perform these operations, why not use it?

9.2 Temperature Diffusion via Graphics Hardware

With the diffusion equation rewritten into matrix form, the temperature calculation can be done through the graphics hardware. In the classical implementation of the temperature diffusion model, the values are maintained within the simulation code as well as drawing them onto the screen which effectively keeps two copies of the same data. With the hardware computation, there is only a single copy of the data maintained within the framebuffer.

To calculate the temperature diffusion, the convolution filter defined above can be applied across all the values in the framebuffer using the graphics API for the machine. For OpenGL, all convolution operations occur within pixel transfer operations. The convolution filter can be specified with glConvolutionFilter2D and the pixels copied from the front buffer to the backbuffer. Within the copy, the API applies the convolution to each pixel, placing the resulting calculation into back buffer. A swapbuffer call can then switch the front and back buffers allowing for the calculation to repeat.

```
#define k .2
float filter[] = {
0.0,   k  , 0.0,
 k , 1-4*k,  k ,
0.0,   k  , 0.0};

glConvolutionFilter2D(filter);
glEnable(GL_CONVOLUTION);
glReadBuffer(GL_FRONT); // Read from the front
glDrawBuffer(GL_BACK);  // Draw to the back

... Draw any initial conditions ...

glRasterPos(1,1);
while(1) {
  glCopyPixels(0, 0, Width, Hieght, GL_COLOR);
  glxSwapbuffers(dspy, wnd);
}
```

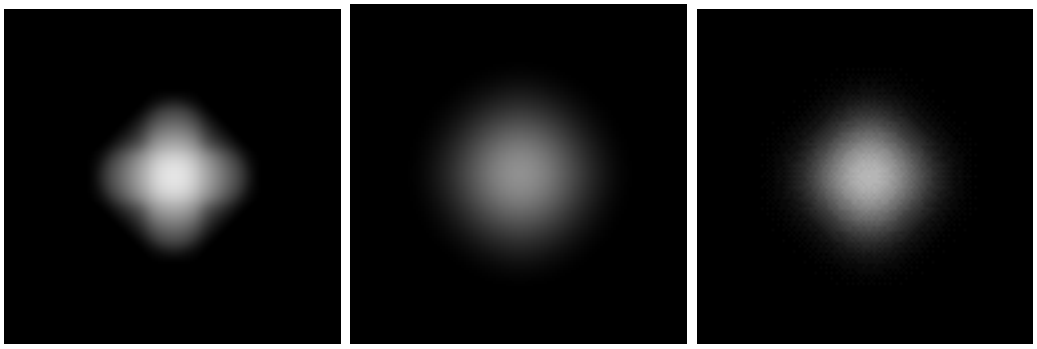

Figure 10: These results are taken from 500 iterations of the temperature diffusion model computed three different ways: (1) O2 Graphics (8 bits per channel) framebuffer calculation (2) Indigo2 framebuffer (12 bits per channel) calculation (3) 16-bit software calculations. In this simulation, the limitations of the 8 bit O2 framebuffer is quite apparent. The simulation stopped proceeding after 250 frames. The O2 simulation took only 3 seconds to reach 500 frames, while the 16-bit software took over 15 seconds to calculate and display.

This simple loop provides the same calculations as the classical calculation however all of the computation is done through the hardware API. The initial conditions are drawn to the screen through standard drawing commands and all processing it handled through the glCopyPixels call.

9.3 Hardware Results

The primary difference in calculating through the framebuffer than through standard code is that the size of the numbers representable is restricted by the number of bits available within the framebuffer. Since the temperature is represented as luminance, only the amount of bits available per color channel can be used for the simulation. For the SGI O2, 8 bit color channels are the largest available, which only allow values from 0 to 255.

Limitations Although the simulation of the temperature procedure much faster in the framebuffer than using maintaining floats for each pixel, the limited range of framebuffer drastically limits the capabilities. In the example of the cross, the simulation is limited since there is not enough bits for an adequate number range. As a result, the simulation proceeds until the convolution filter produces results with the same value which was originally present for all pixels. (See Figure 10)
These results were confirmed when the classical simulation was done using eight bit values to store the temperature values.

Furthermore, the speed of the calculations is dependent on what graphics hardware is available on the system. In this example, although OpenGL provide a hardware independent API, it does not guaranty hardware acceleration on all graphics calls. Using the profiling tools available on the SGIs, pixie and prof, there were large parts of the caculations occurring on the user CPU time for the O2. However, using the Maximum Impact graphics system on the Indigo2, a significant more of the calculation occurs within the hardware which reflects on faster calculation times even though there are more bits per pixel and a slower CPU (R44000 vs R5000).

In general, calculating diffusion through the framebuffer was significantly faster than performing it in host memory and then drawing it to the screen. The hardware limitations can restrict what can be done within the framebuffer. The primary restriction is the 8-bit number representation available. The next section explores ways to overcome this restriction by using the entire RGBA channel as continuous bits to increase the size.

10 20-bit Framebuffer Math

One way to increase the capabilities of hardware simulation calculations is to use the complete range of the hardware for the simulation. It temperature diffusion model, all of the calculations were done with only a single color channel. The calculations were easy to perform since most graphical operations treat the red, green, blue, and alpha channels independently in parallel (blending for instance is separate for the RGBA channels but occurs all at once with OpenGL).

To expand the model to use the RGBA channels as continuous bits, on an 8-bit per channel system, it is possible to represent values with 32 bits, or from 0 to 4.3 billion. With this amount of range, it is definately enough granularity to perform an adequate simulation. The difficulty is to be perform basic operations such as addition and multiplication through the graphics API.

The immediate problem with using RGBA as continuous bits is that the graphics API does not maintain a notion of carry out across the color boundaries. For example, to do an addition we can turn on blending and set the blending factors to one (glBlendFunc), and all rendering following will be the simple addition of the color values. The problem is that any overflow within a color channel is not carried over to the next color, rather they are

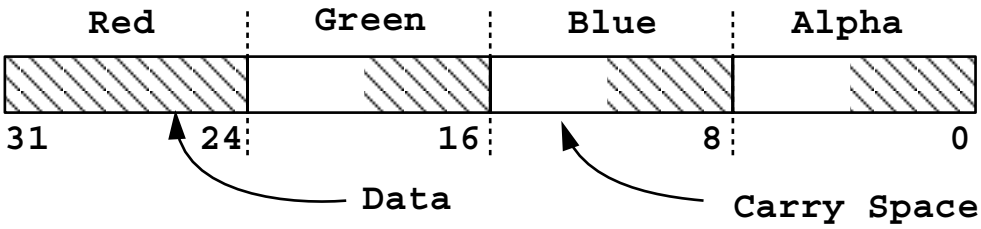

Figure 11: Here is the breakup of the 20-bit number in the framebuffer. Each color channel maintains a 4 bit buffer to handle to carry bits manually. In the form shown here, the carry space is in the upper bits so operations which result in overflow can be handled. If subtraction or division were required, this carry space would have to be shifted to the lower bits.

clamped to the maximum value.

Since the OpenGL API doesn't have any concept of carryout across color channels within blending, overflow and underflow must be dealt with manually. This is done by only using half of the bits per channel for carry and the remaining bits for the data. This is so that multiplying two numbers together there are enough bits available for the carry since multiplying two 4 bit numbers can result in at most an 8 bit number. In an 8-bit model, the remainder bits must be the upper four bits for multiplication since the value will be increasing producing overflow. The opposite is true for division; the remainder bits must be the lower four to handle the underflow. Furthermore, the extra remainder bits at the end of the number can be used for data since there are no remaining channels for carry. This leaves a total of 20 bits for computation, values ranging from 0 to 1.05 million. (See figure 11.)

10.1 20-bit Operations

To perform division and multiplication, OpenGL blending can be configured to do the required operation. The remainder is handled in four steps: getting the remainder, carrying, adding, and clearing. First the remainders are copied out of the results and placed into a separate space in the framebuffer. This can be done with an logical mask operation which only takes the remainder bits out of the color component (glLogicOp). The actual carry can then be performed with a color matrix operation which maps the color red channel to green, the green to blue, and the blue to alpha. This result can then be added back to the original image with a bit shift to push the remainder bits up to the data bits. Finally, the remainder bits can be cleared with a masking operation.

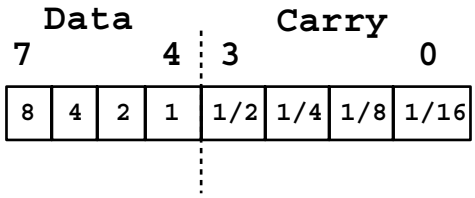

Figure 12: With the data in the upper four bits while the carry is in the lower four, the number representation is as shown is present after any divide or subtraction. The carry operation would normally would transfer the lower bits to the upper bits of the next color channel. However with division, the divisor must be placed on every channel. Since, this means that there is a restriction on the numbers representable for the divisor. Even simple numbers like .2 cannot be represented (.1875 is the closest).

These calculation could be merged together since the graphics pipeline is defined in a certain order. For example, the obtaining the remainder and the carrying could be merged since the logical mask is separate from the color matrix.

10.2 Limitations

With the increase in number representation, there becomes a problem with the numbers available for multiplication and division. To do a multiplication on the 20-bit number spread across the four 8-bit channels, we are restricted to only being able to multiply and divide by 8-bit numbers. The reason for this is that the multiplication/division operation is done through blending and must be applied by each color channel. Since each channel is defined as four bit integral and four bit fractional part (which is used as the remainder), we can only divide by values ranging from 0 to 15.9375 (1111.1111_2). (See Figure 12.)

Since we're so limited in range, the values of the constants within the equation must be selected carefully to match the representable values for the hardware.

10.3 Results

The benefit of having a larger number representation is that it can significantly improve the results of a physical simulation. Doing the temperature diffusion model using 20-bit numbers, we were able to get a solution which provided a 20 bit result, which was completely calculated through the framebuffer. (See Figure 13.)

Figure 13: This image was created using the temperature dispersion model entirely implemented through OpenGL commands. On the left is the 20-bit representation, on the right, the red channel which is the most significant byte is mapped to luminance. The middle region was used for performing the calculation on the remainder.

11 Benchmarks

One of the interesting properties of doing these calculations through the framebuffer and OpenGL is scalability. OpenGL provides a hardware independent API which specifies exactly what happens in the graphics pipeline but doesn't place any restrictions on what must be accelerated through hardware. As a result, there many variations of hardware accelerated graphics available. Here are a few of the systems tested:
Test Case: 128x128 Temperature Diffusion.
Time for 500 diffusion frames.

Test Machine: SGI Indigo2 R4400 Maximum Impact
16-bit Software Calculation: 9.2 Seconds
12-bit Framebuffer Calculation: 2.84 Seconds (High Impact: 4.72)
20-bit Framebuffer Math: 31.7 Seconds

Test Machine: SGI O2 R5000
16-bit Software Calculation: 14.42 Seconds
8-bit Framebuffer Calculation: 2.86 Seconds
20-bit Framebuffer Math: 3 Minutes, 38 Seconds

The clearly show the differences in hardware acceleration. Placing the cal-

culation inside of the framebuffer defiantly improved performance. Even the difference between the High and Maximum Impact graphics boards provides twice the performance difference.

12 Hardware Fluid Calculation

All of these methods explored in this work is applicable to hardware calculation of build simulations. The most obvious example of this is the drag component of the Navier-Stokes equations which is identical to the temperature diffusion expression. As shown above, this calculation can be performed as a straightforward convolution. The convection terms requires multiplication which can be calculated through the 20-bit framebuffer math.

The mass conservation would benefit the most from the framebuffer calculations since it requires a multipass operation. The gradient can be calculated with a simple convolution and adding back in with blending. The only complication in the model is that the gradient can be both negative and positive. An attempt was made to implement the mass conservation through hardware however there just wasn't enough accuracy in the 12-bits of the framebuffer to get acceptable results. Hopefully with more advanced graphics hardware advancements, this limitation will be lifted.

13 Discussion

A large part of this work is ahead of its time. Most of the graphics operations which were used within this work are currently considered extensions of OpenGL and not available or accelerated on certain systems (like the O2). In the next release of OpenGL, version 1.2, there will be a subsection which includes imaging operations like convolution and color matrix.

Furthermore future hardware accelerated graphics board will have increased pixel depth. Silicon Graphics is planing to release their latest graphics board in 1999 which will represent color value with a floating point format with a mantissa and significant. This could greatly improve the range of values supported inside the frame buffer.

References

[1] Fostern, N., and Metaxas, D. (1997). Modeling the Motion of a Hot Turbulent Gas. *SIGGRAPH, 1997*

[2] Verge, M.P., Causse, R., Hirschberg, A., (1995) A Physical Model of Recorder-Like Instruments *ICMC Proceedings, 1995*

[3] Fostern, N., and Metaxas, D. Realistic Animation of Liquids *University of Pensylvania*

[4] Fostern, N., and Metaxas, D. Controlling Fluid Animation *University of Pensylvania*

[5] Stam, J., and Fiume, E. (1995) Depicting Fire and Other Gaseous Phenomena Using Diffusion Processes. *ACM-0-89791-701-4/95/008, 1995*

[6] Gamito, M.N. Simulation of Turbulen Flow Using Vortex Particles. *INESCA, 1994*

[7] Yaeger, L., Upson, C., Meyers, R., Combining Physical and Visual Simulation: Creation of the Planet Jupiter for the Film 2010. *SIGGRAPH 1986*