UNIVERSITY OF MINNESOTA DULUTH

This is to certify that I have examined this copy of a master's thesis by

Andrew Paul Norgren

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by the final

examining committee have been made.

Name of Faculty Adviser

Signature of Faculty Adviser

Date

GRADUATE SCHOOL

GPU Based Particle Dispersion Modeling with Interactive Visualization Support for Real-time Simulation

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA DULUTH

BY

Andrew Paul Norgren

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

Pete Willemsen

June 2008

Table of Contents

List of Tables

List of Figures

1. Introduction

Releasing a harmful gas in an urban environment can be a threatening attack. Knowing how a gas would spread through the city could help in the assessment of such an attack. Computational fluid dynamics (CFD) models can be used to predict where the gas will go based on the wind speed, wind direction, and other environmental factors. However, many CFD models are too slow to estimate the dispersion of a gas when the results need to be known fast or even before it would actually happen. QUIC (Quick Urban and Industrial Complex) is a system composed of QUIC-URB, used to compute the wind in the environment, and QUIC-Plume, used to model the dispersion of the gas (Williams, Brown, Boswell, Singh, & Pardyjak, 2004). This system is relatively fast compared to other CFD models, but it still isn't able to estimate the dispersion of a gas in real-time. The goal of this work is to increase the performance of QUIC to provide the estimation of particle dispersion in real-time.

Another motivation to increasing the performance of QUIC is to provide realistic wind and particle simulation for the Treadport Active Wind Tunnel (TPAWT). The TPAWT is a virtual environment being developed at the University of Utah, which consists of a 6x10 foot treadmill positioned in front of three large screens that display a virtual environment. A user moves through large virtual environments using the treadmill and is hooked up to a mechanical arm to keep them in the center. The TPAWT is used to provide realistic wind simulation acting on the user by the use of vents positioned around the user that provide airflow. When the user is moving through this virtual environment, they will feel wind just like they would in a real environment. This feeling of wind is used to increase immersion in this environment. Immersion is when a user gets the feeling that they are actually in the virtual environment.

Particle dispersion also increases immersion by modeling the effects of wind visually. Modeling the visual aspects of the wind and the physical wind in real-time help create a more realistic virtual environment.

The performance of QUIC-Plume was increased using the graphics card to provide real-time particle dispersion. Graphics cards contain highly parallel graphics processing units (GPUs) that are capable of displaying real-time 3D graphics. Their performance has been increasing fast due to the demand for better graphics in video games. GPUs are programmable using specialized languages similar to C allowing them to be used for general-purpose computations. Many scientific problems have already taken advantage of the inexpensive parallelism of graphics cards. QUIC-Plume was reworked into a GPU version that was implemented using the graphics card.

Reworking QUIC-Plume to run on the GPU created a 3D visualization tool to be used for visualizing and developing dispersion models. The GPU implementation was done using the graphics API, OpenGL, which made available all the functionality and tools of OpenGL for creating 3D visuals. Since all the particle positions are stored in GPU memory for simulating the dispersion, it is relatively fast to display all the particles in 3D space of a domain. The positions of particles can now be seen on a monitor or in a virtual environment as the simulation progresses, allowing for the behavior of particles to be watched visually while they are being advected. This feature, which became the basis for the 3D visualization tool, is a new possibility that wasn't obtainable using QUIC-Plume. Additional 3D visual tools were also created for aiding in the development of dispersion models.

These additional tools include path lines, domain layers, isocontours, and isosurfaces. A path line dynamically shows the path of a particle over time by drawing geometry, such as a line, using the positions of the particle. The domain of the simulation is a grid of 3D cells, each containing turbulence and wind parameters. A domain layer is a layer of cell values drawn in the domain as transparent layers. The layers can be drawn using planes parallel to the x-y, x-z, or y-z planes, or planes that are fully rotational. The data is encoded into colors to visualize the parameter values of cells of the domain. Isocontours are colored regions of cell values that lie in the same range drawn on layers of the domain. They can be used to visualize turbulence parameters of single layers in the domain. Isosurfaces are a 3D version of isocontours, representing a range of values throughout the entire domain as a 3D volume.

The GPU implementation of QUIC-Plume gives accurate results of particle dispersion, allows millions of particles to be simulated rather than thousands and can be seen visually in a real-time simulation. The performance is approximately 180 times faster for a simple Gaussian test case and 250 times faster for a situation with multiple buildings, providing a significant increase to allow for real-time results and visualization of particle dispersion. The 3D visualization tool created aides in the understanding and development of dispersion models.

2. Background and Related Work

When gas is emitted from the exhaust pipe of a car, it disperses through the air based on environmental factors. The individual particles that compose the gas will be blown around in the wind reflecting off of buildings and the ground. In order to model the behavior of the gas, the study of computation fluid dynamics (CFD) can be used. CFD models are often based on the Navier-Stokes equations, which are equations that can describe the motion of fluid, such as gas or liquid. Solving the full set of Navier-Stokes equations is very hard and only a few exact solutions are known (Budd, 2003). Therefore, other methods attempt to approximate the solutions to fluid flows by simplifying the Navier-Stokes equations. Still many CFD models are very computationally intensive and are not always able to provide results fast enough. Fast turn-around time is essential for quickly predicting the dispersion of dangerous chemicals through a city or for multiple scenario assessment where many simulations must be run. Fast-response modeling systems can be used to simulate fluid flow in a relatively short amount of time for emergency response applications or urban planning scenarios. QUIC (Quick Urban and Industrial Complex) is a fast-response system that models the particle dispersion of airborne gases through a city (Williams, Brown, Boswell, Singh, & Pardyjak, 2004). It is composed of QUIC-URB, used to create a 3D wind field of the area, and QUIC-Plume, used to model the flow of particles through the urban area. Although this system is a fast-response model, it still is not fast enough to provide real-time interactive rates.

"Real-time" requires the computations for a simulation to happen within the time that an actual event would occur. For example, imagine the simulation of a ball being thrown through the air. The following pseudo code illustrates the computation of the movement of the ball.

```
Initialize previous_position and time_step
Initialize total_time to 0
While application is running {
    Compute movement as a function of total_time and previous_position
    new_position = previous_position + time_step*movement
    total_time = total_time + time_step
    previous_position = new_position
}
```

The variable *movement* is the direction in which the ball is heading. To compute the movement of the ball with real-time rates, a location of where the ball will be must be calculated before the ball would have actually gotten there. A new position of the ball is computed every time step, (inside the loop). The smaller the time step means the less distance the ball will travel each loop and the more accurate the position of the ball will be. However, it is possible with a given time step that the computations of the ball's position aren't fast enough to keep the simulation running in real-time. It is also possible with a given time step that the computations of the ball are at a speed that allows for the simulation of the ball to happen much faster than it would have actually occurred. To display the simulation with real-time rates, the position of the ball must be computed with enough time left over to draw the ball on the screen. Also for smooth looking movement of the ball, a new position must be computed and drawn in less than the amount of time of the screen refresh rate, and the time step must be small enough so the distance the ball moves isn't too large each time step. Typically for a computer screen, the screen is refreshed every 1/60 of a second.

For a particle dispersion model to be simulated in real-time, thousands and even millions of particle positions must all be calculated each time step within the appropriate amount of time. Here is some pseudo code to simulate particle dispersion.

```
Initialize *previous_position* for each particle
Initialize *total_time* to 0
Initialize *time_step*
While application is running {
    For each particle, i {
        Calculate *movement* for particle, i
        $new\_position_i = previous\_position_i + time\_step*movement$
        $previous\_position_i = new\_position_i$
    }
    $total\_time = total\_time + time\_step$
}
```

For each time step, every particle's position is updated. If there are thousands or more particles, this process could take too long to be able to simulate in real-time. To speed up the process, the for loop that moves all the particles can be parallelized. Then instead of calculating the positions for each particle one by one, multiple particles can be taken care of at once. Personal computers are shifting towards multi-core processors, being built with 2,4,8, or even more cores. An 8-core machine could by used to parallelize the above loop, computing the positions of 8 particles at the same time. However, the focus of this paper is to parallelize the loop using the graphics card, which is normally used to display graphics onto a screen.

Graphics cards contain graphics processing units (GPUs), which take care of the processing needed to render images onto the computer screen. The faster this is done, the better the graphics are that can be displayed. The video game industry has been driving the performance of graphics cards in order to display better and better graphics. GPUs are vector processors that contain highly parallel stream processors, used to display real-time 3D graphics. These stream processors have been designed to work well with computations that are SIMD (single-instruction, multiple-data) computations. They are normally used for the SIMD operation of rendering images by coloring the pixels of the screen in parallel to display the image. On the most current GPUs, there are 128 stream processors capable of calculating 32-bit floating-point operations.

Researchers have taken notice of the GPU and started taking advantage of their computational power in order to solve problems that are not necessarily related to computer graphics. Because of this, General Purpose computation on Graphics Processing Units (GPGPU) has started, which is a way of using the GPU to solve non-graphics problems. Many scientific problems and simulation applications have already been solved and developed using the parallel stream processors of the GPU. Some examples of these are numerically solving the Navier-Stokes equations (Scheidegger, Comba, & Cunha, 2005), solving multi-grid problems (Goodnight, Woolley, Lewin, Luebke, & Humphreys, 2003), solving dense linear systems (Galoppo, Govindaraju, Henson, & Manocha, 2005), and cloud dynamics (Harris, Baxter, Scheuermann, & Lastra, 2003). A particle system has been designed to run entirely on the GPU, simulating a million particles at interactive rates (Kipfer, Segal, & Westermann, 2004). This particle system is similar to our GPU particle system, with the difference being that our GPU particle system uses a more sophisticated type of movement than just a basic gravity problem and has been developed to serve as a tool for engineers. Another particle dispersion model has been developed that visualizes dispersion in urban environments using a Lattice Boltzmann Model (Qui, et al., 2004). It uses the GPU to accelerate the simulation; however it does not run in real-time.

When solving a problem using a graphics card, the approach has to be designed to fit into the framework of the GPU architecture. To understand how to accomplish this, one first needs to have a basic idea of graphics and how the graphics-processing pipeline works. 3D graphics is constructed out of 3D geometry, (defined by vertices), that can be colored and textured. Without texturing, individual geometry will be one solid color. A texture is like an image that is placed on the geometry allowing, for example, a table to look like a wood table. For detailed information on texturing and creating geometry using the graphics API, OpenGL, refer to the OpenGL Programming Guide (Shreiner, Woo, Neider, & Davis, 2006). 3D graphics are displayed on a screen by taking the 3D representation of graphics and projecting it out into a 2D image to fit into the view of the screen. This process is done with a fixed set of operations, called the graphics-processing pipeline (Rost, 2006). The first stage of the graphics-processing pipeline, called Per-Vertex Operations, operates on the vertices that define the 3D geometry. Each 3D vertex is transformed from object space into eye space and given a shade of color determined by the lighting operations. Primitive Assembly is the second stage, which collects the vertices to create primitives, such as lines and triangles. The next stage, Primitive Processing, basically cuts out any parts of primitives that are outside of the user defined viewing area and transforms the vertices into 2D window coordinates. Following this is Rasterization, which decomposes the primitives into individual fragments or pixels on the screen. The next stage is Fragment Processing, where the most important operation occurring is texturing. When geometry is textured, this stage colors the individual fragments with the colors of a texture. Finally, other per-fragment operations are performed and the fragments are stored into what is called a frame buffer, which is normally output to the screen. This whole process is what is parallelized by the stream processors of a GPU for fast rendering of graphics.

Flexible functionality is provided to the GPU architecture through the use of vertex, geometry and fragment processing units called shaders. These shaders are low-level programs written using specialized languages similar to C, such as the OpenGL Shading Language or Cg, to overwrite stages of the graphics-processing pipeline. They were originally designed for graphics programmers to have more control over their applications to create better graphics and increase performance. The vertex shader overwrites the Per-Vertex Operations stage of the pipeline. It can be used to apply changes to vertices, such as changing their coordinates. The geometry shader is an added stage that falls after the Primitive Assembly stage, which takes a primitive as input, and can be used to create additional primitives. The fragment shader overwrites the Fragment Processing stage of the pipeline, allowing for complete control over the color given to each fragment. For a more detailed explanation of shaders and the graphics processing pipeline, refer to (Shreiner, Woo, Neider, & Davis, 2006) and (Rost, 2006).

The main memory structure on the GPU is textures, which are available in one, two and three dimensions. A texture is an array of vectors, where each vector, called a texel, is a color defined by a red, green, blue and alpha value, which is normally used as the colors for texturing geometry. When thinking of textures as just an array of vectors, there is no reason why the values for the vectors couldn't be represented as information other than a color, such as the position of a particle. A texture can then be used to store data, where a fragment shader can then be programmed to operate on the values of the texture in parallel. This is done by drawing geometry associated with the texture that stores the data to be operated on. The drawing can be done with the graphics state set so there is a one to one mapping between a pixel and a texel. This makes it so that the code in the fragment shader is applied to each texel of the texture. Then instead of letting the output values of the fragment shader be displayed as pixel colors on a screen, store the output values into the texels of another texture. For example, an 8x8 2D texture contains 64 texels. To create a one to one mapping between a pixel and a texel, the resolution of the display window is set to have 8x8 pixels so that each pixel is the size of each texel. A quad the size of the window is then rendered with the texture bound to it, which allows for each texel of the texture to be operated on in parallel. This functionality is what allows the above loop that calculates the movement of particles to be parallelized using the graphics card.

Solving problems on the GPU doesn't mean that all the data and operations occur solely on the GPU. In graphics programming, the data representation of graphics is created on the CPU and sent to the GPU, which then stores and processes the data to display on the screen. GPGPU also works in this same way. However, it is a good idea to limit the amount of data transfers between the CPU and GPU. The reason for this is locality of data. Processors work faster if the data is stored locally, because there isn't the extra time taken to fetch the data. Data is transferred between the CPU and GPU across the GPU "bus". The bus will act as a bottleneck if large amounts of data are being sent across it, because the bus "width" isn't very wide. This means that large amounts of data aren't able to go across at the same time. Although, it is often needed at times for the CPU to send small bits of information to the GPU for use in the shaders.

Understanding how to program graphics and use the graphics-processing pipeline to solve general-purpose applications on the graphics card takes a considerable amount of effort and is no easy task. CUDA (Compute Unified Device Architecture) is a new technology that allows for the parallel processing power of the GPU to be used without having any knowledge of graphics programming. CUDA has an API that is used along with the C programming language to allow for programmers with no graphics experience to parallelize their applications using the GPU.

3. Implementation

The objective was to increase the performance of QUIC-Plume, which is a particle dispersion model that runs entirely on the CPU. In order to increase the performance to provide real-time rates, QUIC-Plume was reworked into a dispersion model that runs on the GPU. This chapter gives a detailed explanation of how the GPU particle dispersion model works. First, an overall view of the system is outlined. Then the methodology for advecting the particles on the GPU is described. Following that is a description of how the textures are created. Then, the process of emitting particles into the simulation is explained. Finally, there is a detailed explanation of each of the dispersion models developed.

3.1 System overview

QUIC-Plume models particle dispersion using a stochastic Lagrangian random-walk approach (Williams, Brown, Boswell, Singh, & Pardyjak, 2004). This approach uses the wind field produced by QUIC-URB along with turbulent fluctuating winds to move or advect particles. As particles move through the air, they experience forces on them, with turbulence being the dominant force, controlling the movement of particles. These are the main equations that perform the advection of particles:

$$x = x_p + Udt + \frac{u'_p + u'}{2}dt$$
$$y = y_p + Vdt + \frac{v'_p + v'}{2}dt$$
$$z = z_p + Wdt + \frac{w'_p + w'}{2}dt$$

Particles advect according to a total velocity comprised of a mean wind velocity and a fluctuating velocity. The variables U, V, and W are the components of the wind field, x, y, and z are the components of the particle's position in 3D space, u', v', and w' are the components of a particle's fluctuating velocity, the subscript 'p' refers to the "previous" value of a variable, and dt is the time step. The fluctuating velocities are updated using these equations, where du', dv', and dw' are found with what are called the Langevin equations used to calculate the change in the fluctuating velocities:

$$u' = u'_p + du'$$
$$v' = v'_p + dv'$$
$$w' = w'_p + dw'$$

The fluctuating velocities are initialized to:

$$u'_p = \sigma_u x_{Random}$$
$$v'_p = \sigma_v x_{Random}$$
$$w'_p = \sigma_w x_{Random}$$

Where the variables σ_u, σ_v, and σ_w are normal turbulent stresses that are present in the flow field and X_{Random} is a random value from a set of uncorrelated, normally distributed variables with a mean of zero and a standard deviation of one.

Particles are released or emitted into the domain with an initial position given by a source. A "source" is the term given to the name of an object that emits particles. The starting position of a particle depends on the type and position of the source that is emitting it. The different types of sources include a point source, a sphere source, a line source, and a plane source. (Other possible types of sources could also be implemented). A point source is the simplest type of source that is basically a point in 3D space. Particles that get emitted from a point source are given the starting position of the location of that point source. A sphere source is a sphere that sits in 3D space, and can be a variety of sizes. Particles that get emitted from a sphere source get emitted from a random spot on the surface of the sphere. A line source is a line defined by its two end points, where particles get emitted from random positions on the line. A plane source is a rectangle defined by a corner point and a width and height, where particles are emitted from random positions on the rectangle. Particles are also given an initial fluctuating velocity when they are emitted using the above equations.

Particles will be advected through the domain, which is a 3D bounding box, until they are outside the boundary of the domain. Particle concentration values can be calculated for defined regions in the domain, by keeping track of the amount of particles in the region each time step. This step is done before advection in order to use the initial particle positions when determining particle build up over time. The visualization shows the domain and the advection of particles as it's being calculated. Here is the basic algorithm written out in pseudo code:

```
while not done{
    Update Time and Time Step
    Emit Particles
    Calculate Concentrations
    Advect Particles
    Calculate Other Per-particle Information
    Draw Visualizations
}
```

The simulation will run in the above loop until programmed to quit, or until the user decides to close the simulation. The time step for the simulation can be in real-time or fixed. When it is in real-time, it is given the amount of time it takes (in seconds) for a complete loop of operations to happen. This means that the time step will be variable, but the particles will advect based on actual time. When the time step is fixed, it is defined during initialization and the particles will advect based on a constant time step. Emitting particles is a step that is not always performed each loop, but depends on how particles are emitted and on the rate that particles are released. Calculating the concentrations is an optional step that can be set up to run during a time interval. The visuals of the simulation are user controlled and can be turned on and off. The simulation was created using the OpenGL API (Shreiner, Woo, Neider, & Davis, 2006) along with the C++ programming language, and OpenGL's Shading Language (Rost, 2006) (used to program the GPU).

3.2 Methodology

Textures are the main memory structure on the GPU and can be used for the SIMD operations of the stream processors. The particle dispersion model's main memory structure is a 2D texture used to store the positions and the fluctuating velocity of the particles. When performing advection, the GPU doesn't allow for the new positions to be stored back into the same texture as the previous positions. To accommodate for this, the data is double buffered using two textures, *previous_positions* and *new_positions*, to store the particle positions. After each time step, the roles of the two textures are swapped. This same method is used for storing the fluctuating velocities of each particle with the textures, *previous_velocities* and *new_velocities*.

The following figure illustrates how the GPU is able to advect the particles in parallel. Figure 1a shows how the GPU implementation performs advection for the particles and Figure 1b shows how the CPU implementation performs advection (which is how QUIC-PLUME works). For the GPU implementation, the particle positions are represented in 3D space using 3D coordinates, which are stored in a 2D texture. The GPU is programmed to perform the advection of multiple particles at once. The number of particles that it can process at the same time increases with the amount of stream processors available in the hardware. On the most current hardware, there are 128 stream processors available. (However, this doesn't necessarily mean the GPU will operate on 128 particles at once). For the CPU implementation, the particles are stored in a 2D array. A CPU can only calculate the advection of one particle at a time (assuming it is a single-core processor). So, the GPU is able to provide an increase in performance by advecting multiple particles at the same time, while a single-core CPU can only advect one particle at a time.

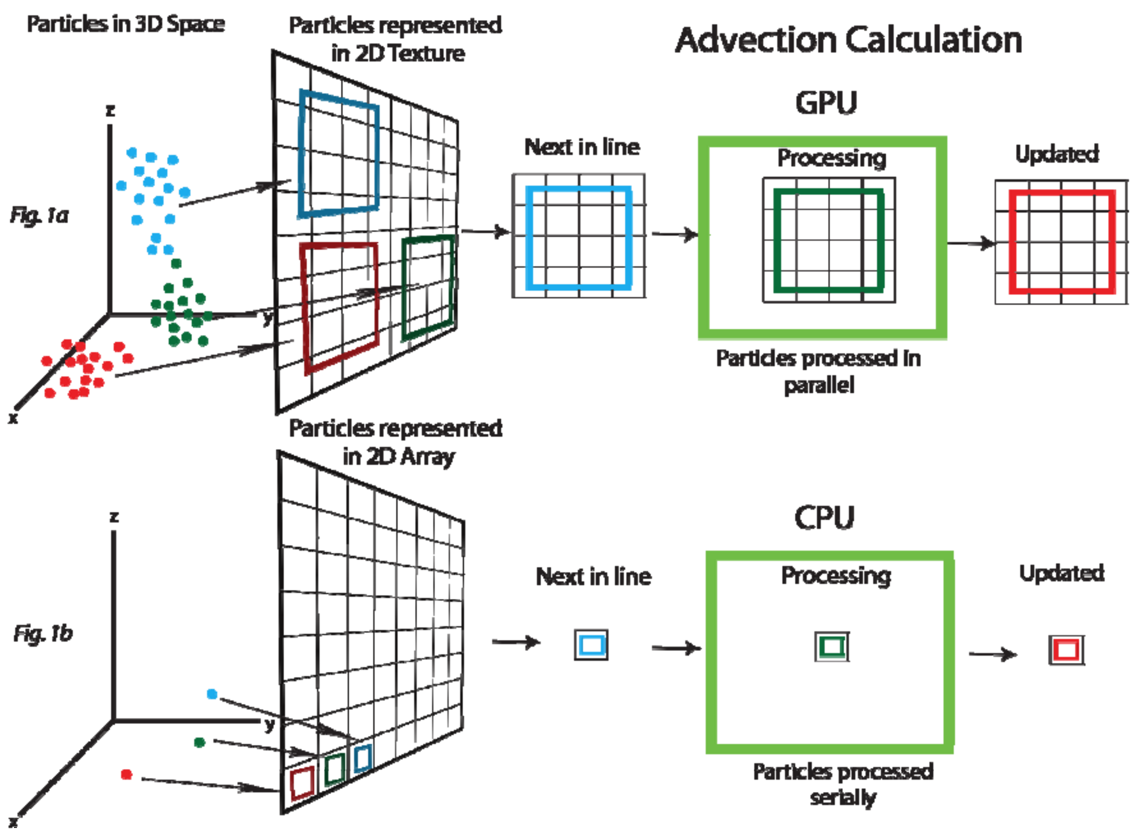

Particles in 3D Space
Particles represented in 2D Texture
Advection Calculation
GPU
Fig. 1a
z
y
x
Next in line
Processing
Updated
Particles processed In parallel
Particles represented in 2D Array
CPU
Fig. 1b
z
y
x
Next in line
Processing
Updated
Particles processed serially

Figure 3.1: Shows the difference in the advection process for the GPU and CPU.

The GPU implementation operates on the whole 2D texture of particles, meaning the fragment shader code will be applied to each texel each time step. Because of this, the *previous_positions* texture is initialized with positions that are outside of the domain for each texel. Then the fragment shader can be programmed to calculate the advection of only those particles that are in the domain.

3.3 Texture Creation

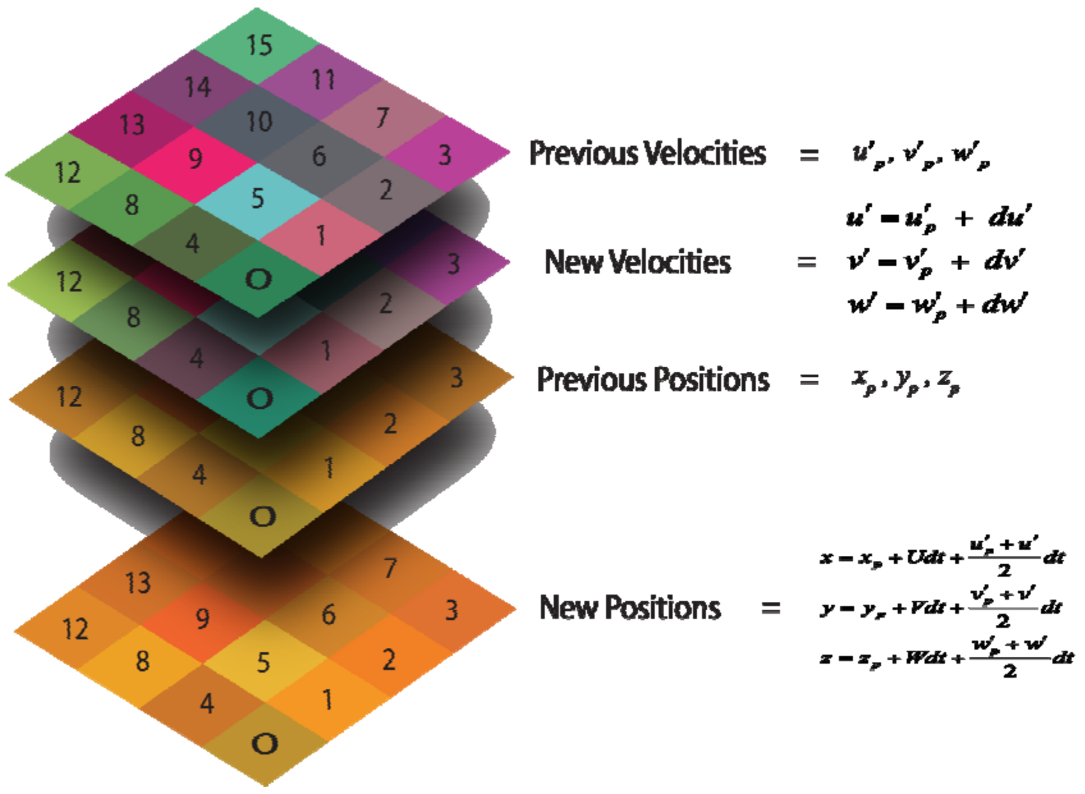

15
14
11
13
10
7
12
9
6
3
8
5
2
4
1
O
Previous Velocities = u'_p, v'_p, w'_p
New Velocities = $u' = u'_p + du'$, $v' = v'_p + dv'$, $w' = w'_p + dw'$
Previous Positions = x_p, y_p, z_p
New Positions = $x = x_p + Udt + \frac{u'_p + u'}{2}dt$, $y = y_p + Vdt + \frac{v'_p + v'}{2}dt$, $z = z_p + Wdt + \frac{w'_p + w'}{2}dt$

Figure 3.2: These rectangles are 2D textures that hold the velocities and the positions of the particles. Each rectangle is a 4x4 2D texture that represents 16 particles. Each colored square holds the attributes for an individual particle and can hold up to four values. Parallelism is achieved by programming the GPU to operate on the 2D textures.

Figure 3.2 shows how a 2D texture is represented on the GPU. Each colored square is an individual texel, which represents an attribute for one particle, storing up to four values. This figure shows that there are 16 particles being represented in the simulation using 4x4 2D textures. The size and dimension of these textures will increase when the number of particles being simulated is more (i.e. one million particles could be represented using a 1000x1000 2D texture). (A discussion of the limitation to the amount of particles is in the conclusion). The order of the numbering shown here is important, because it shows how the texture is created from a set of data, (which is stored in a 1D array on the CPU). To create one of these textures, a 1D array is used that stores all the values for the texture in a “line”. Then the dimensions specified determine the layout for the texture. For figure 3.2, the 1D array used to create a texture is of size 64, (16texels times 4 values per texel) with the values, for each of the texels, placed in the array in the numbering shown. To clarify, the first four values for texel 0 are placed one after another in the array, the four values for texel 1 are placed one after another following the four values of texel 0, and this process continues for each texel. Then by specifying a width and height value of four, the texture is created with the layout for the texels shown in Figure 3.2. This layout is important in order to access the values of a particular texel using what are called texture coordinates. The texture coordinates give the location for a specific texel in a texture similar to the way values of an array are accessed using indices. In Figure 3.2, the texels of these 2D textures can be accessed using two texture coordinates: one being the row number and the other the column number. These two values used together will give the location for one texel. For example, the texel that is numbered 6 can be accessed using the texture coordinates: s = 2 and t = 1, where s is the column number and t is the row number. (The indexing of rows and columns starts at 0).

3.4 Domain (Grid Dependent) Data

The dispersion model is simulated in a variably defined 3D domain. The movement of a particle is largely based on domain data variables. Domain data consists of a three dimensional lattice, (a 3 dimensional box made of individual cubes), the size of the domain, which has variables unique to the individual cells of the lattice. The wind field is one of the domain data variables, where each cell of the lattice is a wind velocity vector. When computing the advection of a particle, the cell value of the wind field that the particle is “in”, (in 3D space), is used in the calculation for that particle.

Domain data, such as the wind field, could be easily loaded into a 3D texture making it trivial to determine which cell of the domain the particle is in. (This could be done easily by using the floor value of the 3D coordinates of a particle as the texture coordinates of the 3D texture). However, using a 2D texture to store the domain data is a better idea, because GPUs tend to be optimized for working with 2D textures. A simple mapping can determine the cell location in the 2D texture using the 3D coordinates of a particle. The following illustration shows how the three-dimensional domain data can be stored into a 2D texture.

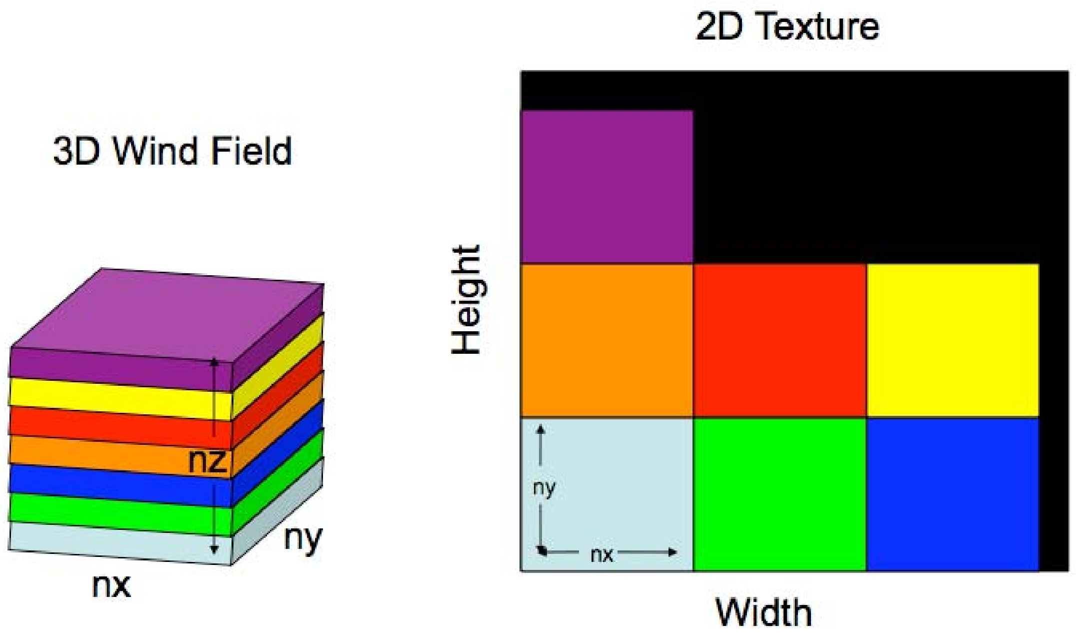


Figure 3.3: Shows how the 3D domain data is stored in a 2D texture.

Each colored block represents one layer of the domain. The domain is split up into layers by each height level of the lattice. Each layer is made up of nx by ny cells. The variables nx, ny, and nz are the domain dimensions in the x, y, and z directions respectively. The width and height of the 2D texture are even integers made to be equal and are determined by trying to minimize the dimension, while being able to fit all the layers. This is done to try and minimize the amount of space used for the texture. The following C++ code shows how the width and height value are found for the texture.

```
int total = nx*ny*nz;

width = (int) sqrt( (float)total );

int scaler;

if(ny > nx) scaler = ny;

else scaler = nx;

width = width - (width%scaler);

bool done = false;

while(!done){

        int num = width/scaler;

        if(num*num) >= nz) done = true;

        else width = width + scaler;

}

if(width%2 != 0) width++;

height = width;
```

Each layer of the domain is taken and placed side by side in the 2D texture. The order starts at the bottom of the domain and places each layer from left to right. When room is run out of in one row of the 2D texture, the next layer is placed on top of that row on the left. The next illustration shows how the domain cell that a particle is in, in the 3D domain is mapped to a texel in the 2D texture.

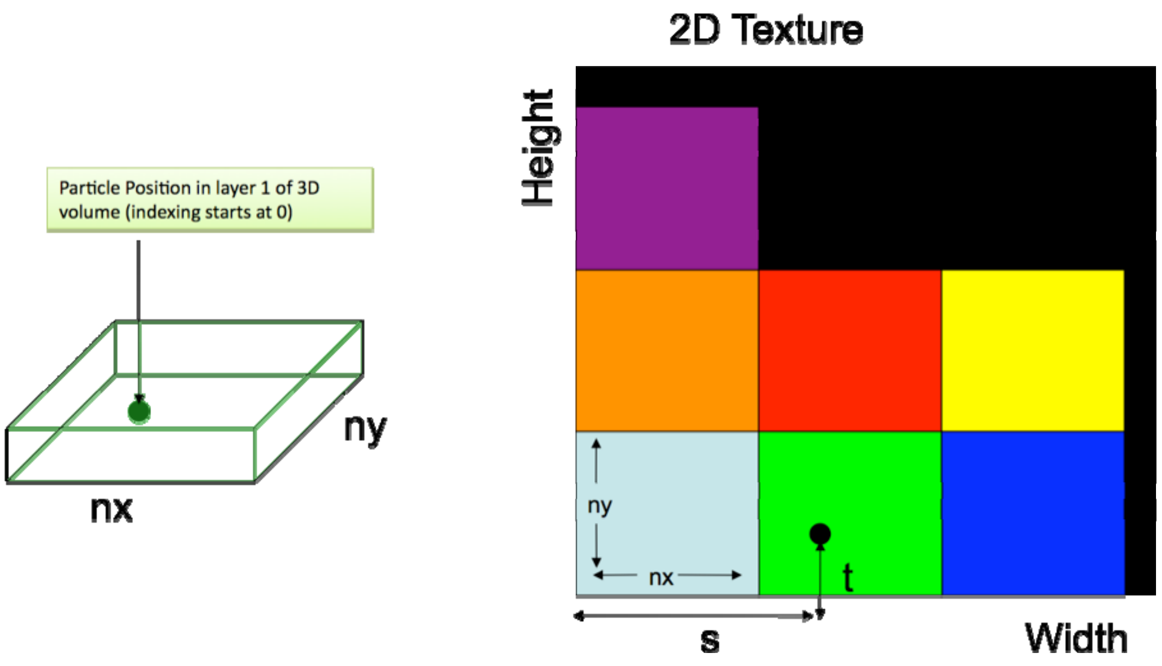

Particle Position in layer 1 of 3D volume (indexing starts at 0)
ny
nx
2D Texture
Height
ny
nx
t
s
Width

Figure 3.4: Shows the mapping of a particle position to a texel in the 2D texture found using the s and t texture coordinates.

Given the (x,y,z) position of a particle, the corresponding domain cell is found using the texture coordinates, s and t.

- x = floor(x')
- y = floor(y')
- z = floor(z')

- numInRow = (width - (width%nx))/nx
- s = x + (z%numInRow)*nx
- t = y + floor(z/numInRow)*ny

The variables, x, y, and z, are the floor values of the particle's coordinates, and the variable, numInRow, is the number of layers in one row of the 2D texture. The texture coordinates are then used to get the texel values of the domain cell using a built in texture lookup function.

Other texture coordinates can also be used to lookup surrounding cell values to compute an interpolated value for a domain parameter. The following figure illustrates the neighboring cells that could be used to calculate an interpolated value.

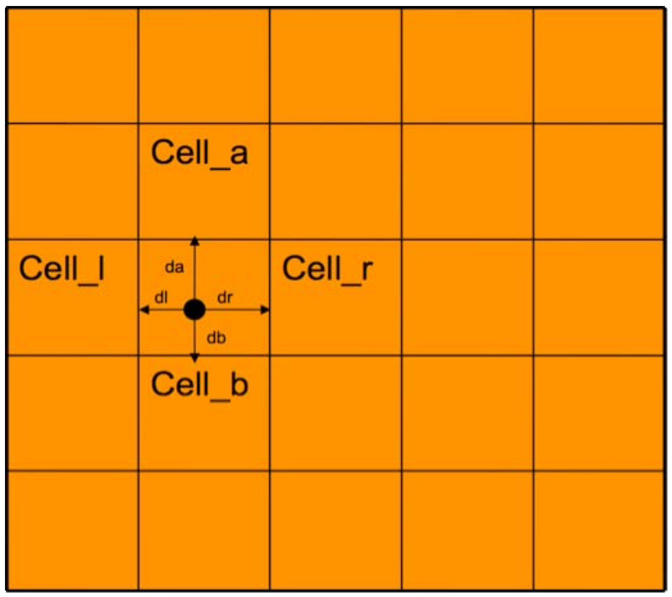


Figure 3.5:

This square represents one horizontal layer of the domain. The black point represents the position of the particle. Cell_a, Cell_b, Cell_r, and Cell_l are the surrounding cells to be used for interpolation.

The neighboring cells shown here are the neighbors with respect to nx and ny. If s and t are the texture coordinates for the current cell as computed above, then the value for Cell_r can be found using the texture coordinates, s+1 and t, Cell_l can be found using s-1 and t, Cell_a can be found using s and t+1, and Cell_b can be found using s and t-1. Note that care must be taken for cells that are on the domain boundary. The distance from the point to the neighbor cell is shown by the variables da, dl, dr, and db. The values of the surrounding cells can be weighted using the distances from the point to the neighboring cell and then averaged with the current cell value to get an interpolated value. The neighboring cells that are above and below with respect to nz can also be used for interpolation. OpenGL has built-in interpolation that will calculate a linear weighted average of the nearest 2x2 array (for 2D textures) of texels that are closest to the center of the point (Shreiner, Woo, Neider, & Davis, 2006). For 3D textures, a 2x2x2 array of texels is used and for 1D textures, 2 texels are used.

3.5 Emitting particles

A Particle is emitted into the simulation by writing its starting position into a texel in the *previous_positions* texture. This can be done by drawing a single point and setting the graphics state so that only one texel of the *previous_positions* texture can be written to. The point is drawn with the RGB color vector defined with the values of the starting position. Then a fragment shader is used to access the color values of the point and write them into the texel. Another method can be used that copies the starting position into the *previous_positions* texture using a built-in method that can update values in a texture with values from an array. The first method described was found to be more efficient. Both of these methods need to know the texture coordinates of the texel to be updated. To avoid overwriting texels that already contain the location of a particle, an ordered list is used that contains the numbers of 0 to N, where N is the number of particles available in the simulation. For each new particle emitted, a number is taken from the list and used to determine unique texture coordinates, s and t, using these equations:

- s = *number* % twidth
- t = *number* / twidth

The variable twidth is the width of the *previous_positions* texture, and the variable *number* is the number taken from the list.

Particles can be emitted from a source in different ways. One method of emission is to emit particles based on a rate of particles per second. For example, if the rate of emission is 10 particles per second and the time step is fixed at 1 second, then 10 particles are released every time step. Another option to emitting particles is to release a set number of particles each time step for a set number of time steps, which is called a continuous release. For this method to work, the simulation has to be set to run for a certain amount of time, and the time step has to be a fixed time step. Then the number of particles to release each time step is defined as:

Number to Release = (total_particles) / (duration/time step)

Where, total_particles is the number of particles to release in the simulation and duration is the amount of time to run the simulation in seconds. The last option for emitting particles is to release all the particles instantaneously. This method will release all the particles in the simulation at once at the start of the simulation.

3.6 Dispersion Models

Particle dispersion is implemented in the **Advect Particles** section of the basic algorithm. The following pseudo code shows a more detailed look at this section:

```
while not done {
    …
    For each particle, i, in the simulation {
        If particle is inside the boundary of the domain {
            Calculate movement for particle, i
            $new\_position_i = previous\_position_i + time\_step * movement$
            $previous\_position_i = new\_position_i$
        }
    }
    …
}
```

The advection calculation for each particle is done with the code of a fragment shader. There are three different dispersion models that were developed. The next three sections explain in detail each of the models.

3.6.1 Gaussian Model

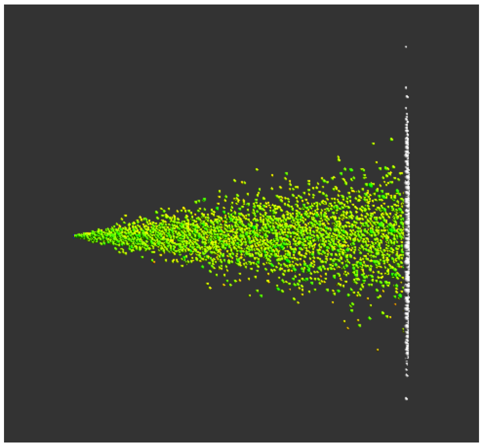


Figure 3.6:

Shows the visualization of the Gaussian dispersion model. Particles flow in a bell shape distribution.

The first model implemented was a Gaussian dispersion model with a uniform flow field. A Gaussian distribution is a function curve that is best described as the shape of a bell. For the test case, the particles are released continuously from a single point source (that is high enough that particles reach the end of the domain without hitting the ground), and flow in concentrations that create a Gaussian curve. The CPU implementation's, (QUIC-Plume's), Gaussian distribution model was validated using an existing analytical solution (Singh, Williams, Pardyjak, & Brown, 2004). Since this had already been done for the CPU implementation, all that needed to be done then was compare the GPU implementation to the CPU implementation for validation. The GPU implementation was presented at the HARMO (*Harmonisation within Atmospheric Dispersion Modelling for Regulatory Purposes*) 11 Conference in Cambridge, England in July 2007 (Willemsen, Norgren, Singh, & Pardyjak, Development of a new methodology for improving urban fast response lagrangian dispersion simulation via parallelism on the graphics processing unit, 2007).

For this test case, the change in the fluctuating velocities is calculated using the following simplified Langevin equations, which were provided by Singh and Pardyjak.

$$du' = \frac{-C_o \varepsilon}{2}\left[\lambda_{11} u' + \lambda_{13} w'\right]dt + \left(C_o \varepsilon \, dt\right)^{1/2} x_{Random}$$

$$dv' = \frac{-C_o \varepsilon}{2}\left[\lambda_{22} v'\right]dt + \left(C_o \varepsilon \, dt\right)^{1/2} x_{Random}$$

$$dw' = \frac{-C_o \varepsilon}{2}\left[\lambda_{13} u' + \lambda_{33} w'\right]dt + \left(C_o \varepsilon \, dt\right)^{1/2} x_{Random}$$

The normal turbulent stresses σ_u, σ_v, and σ_w, are defined for this test case by: $\sigma_u = 2u_*$, $\sigma_v = 1.6\ u_*$, and $\sigma_w = 1.3\ u_*$, (where u_* is a variable that is constant for each cell of the domain). λ and τ

are other turbulent stresses, formed from the gradients of the wind. τ is the symmetric Reynolds stress tensor and λ is the inverse of τ. τ is a 3x3 symmetric matrix, meaning only six components need to be stored. The Langevin equations are simplified by making some assumptions, where $\tau_{12} = \tau_{23} = 0$. This means the only components that need to be stored of the τ matrix are τ_{11}, τ_{22}, τ_{33}, and τ_{13}. The values of τ are computed with these equations: $\tau_{11} = \sigma_u^2$, $\tau_{22} = \sigma_v^2$, $\tau_{33} = \sigma_w^2$, and $\tau_{13} = u_*^2$. The values for lambda are computed using these equations: $\lambda_{11} = (1/\tau_{Det})(\tau_{22}\tau_{33})$, $\lambda_{22} = (1/\tau_{Det})(\tau_{11}\tau_{33} - \tau_{13}^2)$, $\lambda_{33} = (1/\tau_{Det})(\tau_{11}\tau_{22})$, and $\lambda_{13} = (1/\tau_{Det})(-\tau_{13}\tau_{22})$, where the determinant of τ, $\tau_{Det} = \tau_{11}\tau_{22}\tau_{33} - \tau_{13}^2\tau_{22}$. The variable, C_o, is a Universal constant that is set to 5.7. The variable, ε, is the turbulent kinetic energy (TKE) dissipation rate, which is set to u_*^3 /kh, where k=0.4 is the von Karman constant and h is the height level in the domain.

The above equations and variables are the main variables needed to perform the advection calculations for this model. The goal then is to get these variables into GPU memory so that a fragment shader can be programmed to get access to the variables it needs and perform the per particle calculations. As described in the previous section, Domain Data, the wind field velocity vectors, (U, V, and W), are domain cell dependent values which make up the wind field. For this Gaussian test case, these values are constant across the domain and are uniform in U, (meaning V and W will be zero and U will have a value that is not zero). The wind field is stored in a 2D texture, (which I'll call the *wind_field* texture), in which the method of doing this was also described in the same previous section. Since each texel in a texture can store four values and the *wind_field* texture only stores a vector of size three per texel, there is one open position available to store another value in each texel. This available spot is used to store the value of $-C_o\varepsilon/2$ for use in the equations used to update the fluctuating velocities. The tensor values, λ, are also domain cell dependent variables, that get stored in another 2D texture, called the *lambda* texture, the same way as the wind field is. For each cell in the domain, there is a λ_{11}, λ_{22}, λ_{33}, and λ_{13} value, which get stored into the four values of a texel. A 2D texture, called the *random_values* texture, the same size as the particle positions texture, is used to store the X_{Random} variables.

For the advection process, the fragment shader is invoked from the CPU that tells the GPU to calculate the advection for all the particles. The fragment shader first gets the previous position and fluctuating velocity of the particle from the *previous_positions* and *previous_velocities* textures using the texture coordinates calculated by the graphics-processing pipeline. Then the previous position of the particle is used to perform texture lookups into the *wind_field* and *lambda* textures to get the values for U, V, W, $-C_o\varepsilon/2$, λ_{11}, λ_{22}, λ_{33}, and λ_{13}. The X_{Random} variables are found by using a random texture coordinate offset passed into the fragment shader. The texture coordinate offset is added to the texture coordinates of the previous particle in the *previous_positions* texture to provide unique random variables for each particle. The new position and fluctuating velocity of the particle is then calculated and stored into the new_positions and new_velocities textures.

3.6.2 Non-Gaussian Model

The second model developed is a non-Gaussian advection of particles including particle reflection off the ground plane. The wind field is changed to a varied in U wind field in which the strength of the wind velocity component U increases with respect to height. (The components V and W remain at zero). A reference velocity and height is used in the equation to determine the value of U for each height level of the domain. The reference velocity, U_{ref}, is 7.52 m/s (meters per second) at a reference height, Z_{ref}, of 20 meters. The equation used to determine the wind velocity at each of the cells of the domain is: $U = U_{ref}(Z / Z_{ref})^p$, where Z is the height of the cell and p is set to 0.15. The Langevin equations used for calculating the fluctuating velocities are less simplified and modified with these equations, (provided by Singh and Pardyjak):

$$du' = \left[\frac{-C_o\varepsilon}{2}\left(\lambda_{11}u' + \lambda_{13}w'\right) + \frac{\partial u}{\partial z}w' + \frac{1}{2}\frac{\partial \tau_{13}}{\partial z}\right]dt + \left[\frac{\partial \tau_{11}}{\partial z}\left[\lambda_{11}u' + \lambda_{13}w'\right] + \frac{\partial \tau_{13}}{\partial z}\left[\lambda_{13}u' + \lambda_{33}w'\right]\right]\frac{w'}{2}dt + \left(C_o\varepsilon\, dt\right)^{1/2} x_{random}$$

$$dv' = \left[\frac{-C_o\varepsilon}{2}\left[\lambda_{22}v'\right] + \frac{\partial \tau_{22}}{\partial z}\left(\lambda_{22}v'\right)\frac{w'}{2}\right]dt + \left(C_o\varepsilon\, dt\right)^{1/2} x_{random}$$

$$dw' = \left[\frac{-C_o\varepsilon}{2}\left[\lambda_{13}u' + \lambda_{33}w'\right] + \frac{1}{2}\frac{\partial \tau_{33}}{\partial z}\right]dt + \left[\frac{\partial \tau_{13}}{\partial z}\left[\lambda_{11}u' + \lambda_{13}w'\right] + \frac{\partial \tau_{33}}{\partial z}\left[\lambda_{13}u' + \lambda_{33}w'\right]\right]\frac{w'}{2}dt + \left(C_o\varepsilon\, dt\right)^{1/2} x_{Random}$$

These modifications introduce the new variables required to solve the Langevin equations, $\partial\tau/\partial z$ and $\partial u/\partial z$, which are both domain cell dependent. The values of $\partial\tau/\partial z$ are the gradients of τ with respect to the vertical direction, z. The values for $\partial u/\partial z$ are the gradients of wind velocity with respect to the vertical direction, z. These new variables help obtain a better measure of turbulence. The values for σ_u, σ_v, and σ_w change to $2u_*$, $2u_*$, and $1.3u_*$ respectively. u_* is now a cell dependent value defined by the function, $u_* = k\Delta z(\partial u/\partial z)$, where k is the von Karmon constant, 0.4, and Δz is the height of the cell above ground. This modifies the values of τ and λ due to their formulations. The value of $\partial u/\partial z$ for a given cell is found using the following definitions: For all cells that are not on the top or bottom boundary of the domain, $\partial u/\partial z = (U_{k+1} - U_{k-1})/2\partial z$, (where ∂z is 1). Here U_{k+1} is the wind velocity component, U, of the cell above and

U_{k-1} is for the cell below. For cells that are "sitting" on the ground, $\partial u/\partial z = U_k/(0.5\partial z(\log(z/z_o))$, (where U_k is the value of the current cell, and z_o is the surface roughness length which is set to 0.01). For cells that are boundary to the top of the domain, $\partial u/\partial z = \partial u/\partial z$ of the cell below it. The values for $\partial\tau/\partial z$ are calculated the same way, except the corresponding τ value is used in place of U.

Two additional textures are used to store the values of $\partial\tau/\partial z$ and $\partial u/\partial z$ in the 2D textures, *tau_dz* and *duvw_dz*. These textures are created in the same way as the *wind_field* and *lambda* textures. The *lambda* and *wind_field* textures are created with the new values and all the other variables remain the same as in the previous model. When the advection step occurs, the fragment shader used for calculating the positions and velocities of the particles operates in much of the same way as in the previous model with a few additions. The new Langevin equations are used which requires two additional texture lookups into the *tau_dz* and *duvw_dz* textures to get the $\partial\tau/\partial z$ and $\partial u/\partial z$ values. These lookups are performed exactly the same way as the lookups into the *wind_field* and *lambda* textures. After the updated position and velocity vectors are computed, an additional step is used to reflect the particles off of the ground. This is a basic case for reflection, which only requires two steps. First a check is performed to see if the new position of the particle is below the ground plane. If it is, then the particle's z position component and w' velocity component is negated. The particle is now reflected off the ground plane and is back inside the domain.

3.6.3 Multiple Buildings Model

The final model developed uses the turbulence and wind field generated from QUIC, which can create flow fields for domains with buildings. This model was presented at AMETSOC

Conf Coastal Atmospheric Prediction Conference in 2007 (Pardyjak, Singh, Norgren, & Willemsen, 2007). The same Langevin equations are used as in the non-Gaussian model. There are, however, a couple of new features that were added in. The first being reflection off of multiple rectangular shaped buildings in the domain. When a new position is calculated in the fragment shader for advecting a particle, a check has to be done to see if this particle is inside a building. If it is, the new position for the particle needs to become the reflected position off the building. In order to determine if a particle is inside a building or not, a 2D texture called the *cell_type* texture is used which stores a cell type value for all the cells in the domain. (This texture is created in the same way as the *wind_field* texture). When checking to see if a particle is inside a building, a texture lookup is performed into the *cell_type* texture. If the value of this lookup shows it is in the building, then reflection calculations are performed.

In order to perform the correct reflection calculations, the plane of the building that the particle is being reflected off of needs to be known. Another 2D texture called the *buildings* texture is used to store this information. This texture is created in a different way than all the others so far. The height of the texture is the number of buildings in the domain and the width is two. Each row of this texture holds the information of one building, which makes the height of the texture equal to the number of buildings. In each row, there are two 3D vectors used to represent each building, setting the width of the texture to two. The information stored for a building is a 3D position vector and the height, width, and length (making up the second 3D vector) of the building. The 3D position vector stores an (x, y, z) position for the building, where x and z are the lower bounds of the building and y is the middle point of the building in terms of the y-axis. The height gives an upper bound of z + height for the building in terms of the z-axis, the length gives an upper bound of x + length in terms of the x-axis, and the width gives a lower bound of y – width/2 and an upper bound of y + width/2 in terms of the y-axis. This is all the information needed to represent the six planes of a rectangular shaped building. In the *cell_type* texture, a building index number is stored into the 4th component of the cell type for each cell that is inside a building. This index value corresponds directly to the row number in the *buildings* texture. Then when a particle is found to be inside a building, the 't' texture coordinate, (for looking up the building information), is given the value of the 4th component of the cell type. This provides for a way to determine which building a particle is in.

Once a particle is found to be inside a building, a texture lookup is done to get the information for that building. Then calculations are performed to find out which plane of the building to use to reflect the particle. To determine this, the following calculation is done for each plane of the building:

S = -(dot(Normal,prevPos) – dot(Normal,V)) / dot(Normal,u)

The operator "dot" used here is the dot product of the two arguments. Normal is the normal vector of the plane and V is a point on the plane. The variable, prevPos, is the previous position of the particle, which is the position outside of the building. The vector, u, is the current position inside the building minus the previous position outside the building. If the value calculated for a plane, S, is between 0 and 1, then that plane has been intersected. The plane that has the smallest S value that is between 0 and 1 is the plane that is intersected first by the plane and is the one that is used to reflect the particle. The reflected position of the particle is now calculated using the following equations:

1. Determine point of Intersection on the plane, PI
    a. PI = S*u + prevPos

2. L is the normalized vector of the difference between PI and prevPos
    a. L = normalize(PI – prevPos)
3. R is the normalized reflection direction vector
    a. R = normalize(reflect(L,Normal))
4. Length is the magnitude of the vector from PI to the position in the building
    a. Length = distance(PI, position in building)
5. Position is the reflected particle position
    a. Position = PI + (Length*R)

The functions normalize, distance, and reflect are all built-in functions on the GPU that were used. The function normalize(x) returns a vector in the same direction as the argument, x, but with a length of 1, and distance(p0,p1) returns the distance between the two arguments, (p0-p1). The function reflect(l,N) returns the reflection direction for the vector direction, l, coming into the surface whose normal is N. (The resulting direction is defined by l – 2.0*dot(N,l)*N). The type of reflection being done here is analogous to a ball with no spin on a billiards table. Once the reflected particle position is found, another check is needed to see if the particle was reflected into another building. If it is in a building, the particle will be reflected once again using the same steps. This process will repeat until a particle is not inside a building. Note that checks are also done to see if the particle got reflected under ground, and are handled by including the ground plane as one of the planes checked for intersections.

The second new feature added is what is called the Courant-Friedrichs-Lewy condition (CFL condition). This condition is for making sure that a particle does not go through a domain cell without experiencing the conditions of that cell. During a particular time step, it is possible that a particle is advected through two or more cells. Without the CFL condition, the particle's new position and velocity will only be a result of the calculations done from the original cell it was in. The particle will not be affected by the values of the other cells it went through. There may have been strong wind velocities or strong turbulent forces in the cells it passed through, which would have affected the new position and velocity of the particle. To ensure a more realistic solution for the path of a particle, the CFL condition is applied to make sure the particle experiences the forces of all the cells it passes through.

To apply the CFL condition, extra steps are added into the fragment shader that calculates the advection of particles. A particle is first advected using the values of the cell that it is currently in. If the new position for the particle has jumped two or more domain cells, then the CFL condition needs to be applied. (A particle can jump one cell, because it will not skip any cells). The time step that originally was used in the advection calculation is now divided in half and the advection is recalculated with the new time step. If the particle still travels too far, the time step is divided again. This process continues until the particle travels within the appropriate limit. Once this happens, the advection process starts again with the new position and an updated time step value that is the value left over after subtracting the time step used. With this condition, the particle could possibly be advected multiple times ensuring that the particle moves with respect to the cell values that it passes through. This also makes sure that the appropriate reflection calculations are done, making sure that a particle doesn't "skip" a reflection. For example, a particle could skip a reflection when the CFL condition isn't applied by jumping two cells, where the cell skipped contains a building wall. (This would potentially happen at a corner of a building). The CFL condition makes the advection process more robust and stable.

3.7 Calculating Concentrations

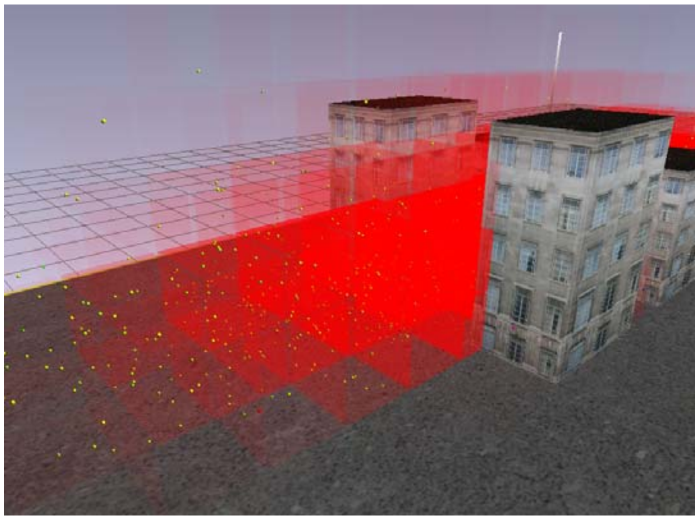


Figure 3.7

Visually shows the concentration boxes. The boxes become more solid as more particles go through them, meaning higher concentration values.

Calculating the concentrations for a simulation requires keeping track of the amount of particles that pass through specified regions in the domain. These regions are defined by “invisible” boxes that are placed inside the domain, (which we call concentration boxes). The boxes keep track of the amount of particles that are inside of it for each time step of a variably defined time interval. The concentration value at the end of the simulation for a box is the average number of particles per time step that were contained in the box. The concentration boxes are defined by a single bounding box that gets split up evenly into a variable number of boxes in the x, y, and z direction.

The process of calculating the concentrations is a slow operation relative to the fast operations of the GPU, because it involves transferring data from the GPU to the CPU. All the particle positions are defined in a 2D texture that is stored on the GPU. The locations of the concentration boxes are stored in an array of structures on the CPU. To determine which box a particle belongs to, the contents of the 2D texture storing the particle positions is read back from the GPU and stored in an array on the CPU. Then the array is traversed and if a particle’s position is inside the bounds of a group of collection boxes, an indexing is used (using the position of the particle) to determine which collection box it is in. That collection box will increment (by one) the number of particles “seen” by it. This process has to be done each time step of the time interval defined to calculate the concentrations, (which has a negative effect on the performance of the simulation due to the data transfer from the GPU to the CPU). At the end of the time interval, each collection box will contain a value, which is equal to the sum of the number of particles that were “inside” of it each time step of the time interval. Then the concentration value for each collection box will become that summed value divided by the number of time steps in the time interval.

4. Visualizations

This chapter gives a detailed explanation of the dispersion model visualizations and the 3D visualization tools created for aiding in the development of dispersion models. Here is a more detailed look at the basic algorithm found in the system overview section of Chapter 3, which can be used as a reference when reading this chapter:

```
While not done{
        **Isosurface -> create density function**
        Update Time and Time Step
        Emit Particles
        **Update Path Lines**
        Calculate Concentrations
        Advect Particles
        Calculate Other Per-particle Information
        Draw Visualizations
            ▪ **Store particle positions in vertex buffer object**
            ▪ **Store particle colors in vertex buffer object**
            ▪ **Store path lines in vertex buffer objects**
            ▪ **Isosurface -> create geometry**
            ▪ **Draw domain axes, background, buildings, and ground plane**
            ▪ **Draw colored particles**
            ▪ **Draw path lines**
            ▪ **Draw isocontours**
            ▪ **Draw domain layer**
```

- **Draw isosurface**

}

4.1 Particles

In order to draw the particles, the positions are taken from the *new_positions* texture and stored into what is called a vertex buffer object. A vertex buffer object is an array of values stored on GPU memory, which allows for the values to be used as vertices that make up geometry. An asynchronous operation is used to copy the positions from the new*_positions* texture into a vertex buffer object. Then the position values in the vertex buffer object can be drawn as points in 3D space. Using a fragment shader, the operation to render these points can be overwritten to make these points look like other objects such as spheres or even snow. Rendering the points to look like spheres creates a more appealing visual and allows for the spheres to be noticeably colored. Particle dependent information, such as a particle's direction, can be encoded into the coloring of these spheres. One color encoding that is used is a color scheme that is based on an opponent color space (Willemsen, Norgren, Singh, & Pardyjak, Integrating Particle Dispersion Models into Real-time Virtual Enviornments, 2008). However, there are other possible color schemes that can be used.

Color encoding the current particle directions is done in the section "**Calculate Other Per-particle Information"** of the basic algorithm. Another 2D texture the same size as the *new_positions* texture, called the *current_directions* texture, is used to store the color encoding of the current direction of each particle. A fragment shader is used to take as input the previous and new position of the particle from the *previous_positions* and *new_positions* textures and compute the current direction by subtracting the previous position from the new position. A color encoding can then be applied to the current direction to get a color vector and store it in the *current_directions* texture. The colors from the *current_directions* texture are taken and stored in a vertex buffer object. The particles can then be colored using the vertex buffer object as the colors for the particles.

4.2 Path Lines

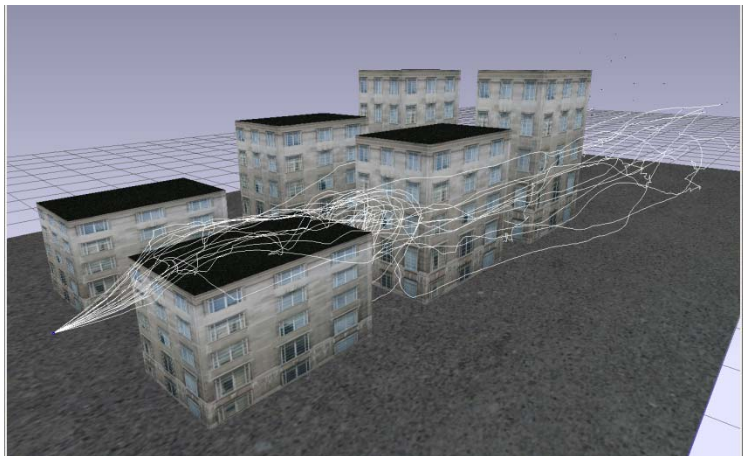


Figure 4.1:

Path Lines drawn as white lines allowing for the paths of particles to be seen dynamically.

Generating path Lines is a tool that is used to dynamically show the path of a particle over time. As particles are injected into the simulation, geometry, such as a line, can be drawn to represent the paths of the individual particles. This allows for the behavior of particles to be tracked and watched more carefully. When drawing the path line of a particle, one can more easily see the turbulence that the particle experiences, or the reflections that it encounters. This tool has proven to be very helpful in determining correct behavior of the dispersion model. It also became very useful for visual debugging, allowing for programming errors to be seen such as an incorrect or missed reflection. The first pass at creating path lines was successful in showing the paths of particles, however it decreased performance greatly due to the large data transfers between the CPU and GPU. A second pass uses a technique that provides for a way to keep the data local (on the GPU), which turned out to not have a big negative affect on the performance. The one drawback to the technique used for the second pass is that the lengths of the path lines are limited to the size of the largest width value that a 2D texture can be. The following two paragraphs describe each of the techniques used for creating the path lines.

The first pass of generating path lines uses a technique that involves lots of data transfer between the CPU and GPU. To generate a path line for a particle, every position of every time step that the particle is in the simulation is stored so that the whole path of the particle can be drawn. These positions are stored in a list using the C++ list class from the standard template library. A structure, called a *particle_position* structure, is created on the CPU to store a position using the x, y, and z coordinates. A path line for a particle is stored on the CPU using a list of *particle_position* structures, called the *position_list*. Since there are many particles that can have path lines associated with them, another list is used to store each of the position lists, called the *path_line_list*. Another structure, called an *index* structure, is created that stores a particle's texture coordinates of the *previous_positions* texture, the index value into the path_line_list, and a Boolean variable that says if a particle is outside or inside the domain. One final list, called the *index_list*, is used to store these *index* structures, where one of these structures is created for each particle that has a path line associated with it. When a particle is injected into the simulation with a path line associated with it, its initial position is stored in a new *particle_position* structure and stored in its *position_list*, and the values for the *index* structure to be placed in the *index_list* are found. The texture coordinates and initial position of a particle are determined when a particle is emitted into the simulation, (which is described in the section on emitting particles). The index value into the *path_line_list* is simply the value of an incremental counter that is used to keep track of the number of path lines, and the Boolean variable is initialized to false to say that it is inside the domain. After the initial positions are found, the following positions for each time step need to be found in order to draw the path line. To get the positions for each path line for a given time step, the particle positions in the *previous_positions* texture are transferred from the GPU to the CPU, and stored on the CPU in an array. (Note that this has to be done each time step, which has a large negative affect on performance). Then the *index_list* is traversed, and if the *index* structure's Boolean value is set to false, an indexing is used to get the position for the corresponding particle's path line. The indexing that is used is:

- index = t*width*4 + s*4

The variables, s and t, are the texture coordinates from the *index* structure, and width is the width of the *previous_positions* texture. Then the (x, y, z) position of the particle is found using that index, where the variable, position_array, is the array of positions on the CPU:

- x = position_array[index]
- y = position_array[index+1]
- z = position_array[index+2]

If the position found is outside of the domain, the *index* structure's Boolean variable is set to true so that this process doesn't have to be done the next time step for this particle. Otherwise this new position is stored in a new *particle_position* structure and put in the particle's *position_list*. This process is done for each time step that particles are in the domain with path lines associated with them, creating dynamic path lines. After the path lines are updated, they can now be drawn by traversing each of the *position_lists* in the *path_line_list*. To draw a path line, a line is drawn between each neighboring position in the *position_list*. A convenient operation implemented was a clear operation that erased all the path lines. This was done by clearing all the lists and setting the incremental counter of path lines to zero. This allowed for new path lines to be released while maintaining the visibility of the simulation. Also the speed of the simulation would significantly decrease as more path lines were generated, so by clearing them, the simulation could return to it's original speed.

The second pass used to generate path lines eliminates the large data transfers between the CPU and GPU. This is done by storing the positions of the path lines in a 2D texture, called the *path_line* texture. Each row of the texture represents a path line by storing the positions of a particle over time in the texels of the row. This means that the maximum length of a path line is equal to the width of this texture. Also, the maximum number of path lines that can be created is equal to the height of this texture. A structure on the CPU side, called the *pathIndex*, is used to represent a path line. These structures are stored in a list, called the *path_line_list*. The *pathIndex* structure stores a particle's texture coordinates of the *previous_positions* texture and the current texture coordinates of the *path_line* texture. When a particle is emitted with a path line, the texture coordinates of the *previous_positions* texture are stored in a new *pathIndex* structure. The texture coordinates for the *path_line* texture are initialized as follows: s = 0 and t = *pathNum*, where *pathNum* is a counter that is incremented each time a new path line is started. The t value corresponds to the row and the s value corresponds to the column. The *pathIndex* structure is then added to the *path_line_list*. The texture coordinates stored for the *path_line* texture are used to determine which texel to write a new position into. The s coordinate starts at 0, and is then incremented after each new position, successively storing each new position of a path line in its row of the *path_line* texture. To update the path lines, a fragment shader is used to update the position values for each path line one by one. The fragment shader takes the previous position from the *previous_positions* texture and writes that value into the *path_line* texture using the texture coordinates from the *pathIndex* structure. To draw these path lines, each row of the *path_line* texture is placed into its own vertex buffer object. Then OpenGL commands are used to tell the GPU to draw the positions of each of the vertex buffer objects as a line. Updating path lines is a dynamic process that happens every time step there are particles with path lines in the simulation. A clear operation for this technique was also implemented, which is done by clearing the *path_line_list* and initializing the counter of path lines to zero.

4.3 Domain Layers

Domain data, such as the turbulence field and wind field, can be visualized by drawing transparent layers of the domain data in the domain of the simulation. Each of the cells of the 3D gridded domain has turbulence and wind parameter values in them. These values can be determined visually by drawing colored layers of cells of the domain data in the domain. A color scheme is used to color the cells of the layers in order to provide meaning. To draw a layer of the turbulence field, a 3D texture is used to store the values of a turbulence parameter. The 3D texture is made to be the same size of the domain, which provides a direct correlation between a texel in the texture and a cell in the domain. This provides for a straightforward approach to drawing the layers. There are two different ways to specify the layer to draw of the domain.

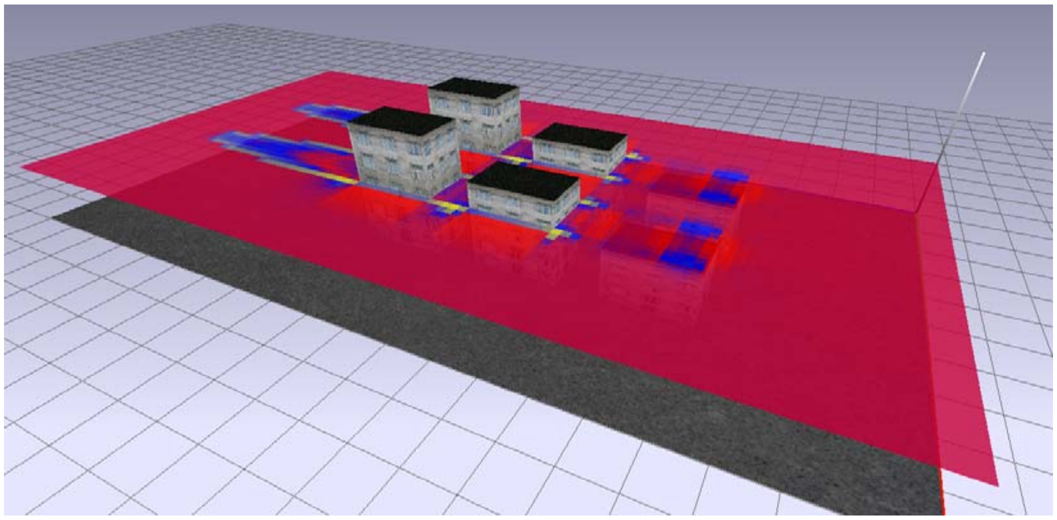


Figure 4.2: Transparent domain layer parallel to the x-y plane, which has movement along the z-axis.

The first way is to specify layers that are parallel to the x-y, y-z, or x-z plane of the domain. When drawing these layers, a quad is rendered on the screen specified by four vertices that are all on the boundary of the domain. These four vertices are determined by first deciding which plane to be aligned with, which leaves the left out axis to be the axis that the plane can move along. A value in the positive axis direction, (bounded by the domain dimensions), is used to move the layer along that axis. For example, if the domain dimensions are 100 in x by 50 in y by 20 in z, the plane chosen to be parallel to is the y-z plane, and the value chosen along the x axis is 70, then the four vertices used to draw the quad are (70,0,0), (70,0,20), (70,50,20), and (70,50,0). (These vertices are specified in the order (x, y, z)). The plane to align the layer with and movement of the layer along the left out axis is controlled by pressing keys on a keyboard. The quad is now colored using the corresponding values in the 3D texture with a color scheme. The corresponding values in the 3D texture are determined by telling OpenGL to render the quad using texture coordinates of the 3D texture. Since there is a direct correlation between a texel in the 3D texture and a cell in the domain, the texture coordinates used are the normalized vertices of the quad. Using the same example as above, the texture coordinates would be (70/99,0,0), (70/99,0,1), (70/99,1,1), and (70/99,1,0). (The normalized value of x is 70/99 and not 70/100, because the indexing of a texture starts at 0). The following figures show a layer drawn parallel to the x-z plane colored with two different built-in OpenGL texturing filter options.

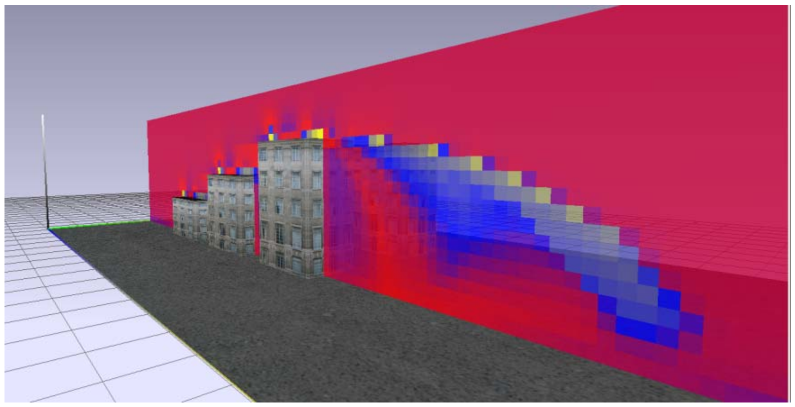


Figure 4.3: This image shows nearest neighbor texture filtering. The direct mapping of one texel to one cell can be seen, where one square in the image is colored with one texel value.

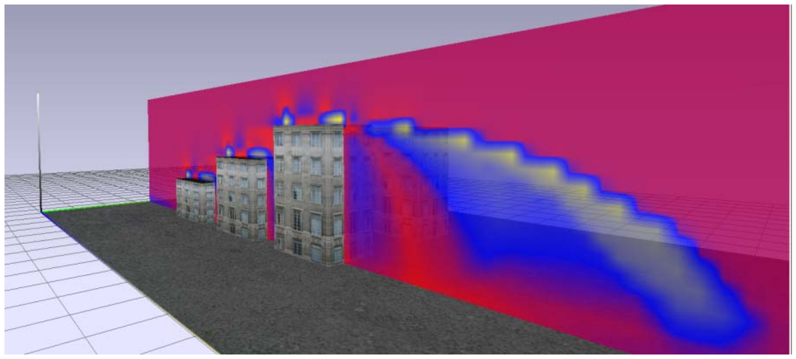


Figure 4.4: This image shows a smooth texture filtering. Pixels are colored using a linear interpolation of surrounding texel values to create a smooth blend of color.

Now in order to apply the color scheme, a fragment shader is used that takes a turbulent value from a cell and maps it into a color. The mapping takes a value and normalizes it between 0 and 1 with respect to the maximum and minimum turbulent value. There are two ways to determine the maximum and minimum values. The first is to use local values, which are the maximum and minimum turbulent value of the current layer being drawn. The second is to use global values, which are the maximum and minimum turbulent value of the whole domain.

After the normalization, the color is found by mapping the normalized value into a color between the ranges of three colors. Here is the code in the shader that does this:

```
if(t >= slider){
        t = (t – slider) / (1.0 – slider);
        color = color2 * (1.0 – t) + color3 * t;
}
else{
        t = (slider – t) / slider;
        color = color1 * t + color2 * (1.0 – t);
}
```

The variable *t*, is the normalized value, and *color* is the color that the value is mapped to. Three colors are used, *color1*, *color2*, and *color3*, to create a range of colors. The variable *slider*, is a user controlled variable, which can be changed in the range of 0 to 1 allowing for the colors of the layer to change with respect to the change in the value of *slider*. When *t* is greater than or equal to the value of *slider* and less than or equal to 1, the mapped color value is between the colors of *color2* and *color3*. When *t* is less than the value of *slider* and greater than or equal to 0, the mapped color value is between the colors of *color1* and *color2*. If the value of *t* is 1, then the mapped color value will be *color3* and if the value of *t* is equal to the value of *slider*, the mapped color value will be *color2*. If the value of *t* is between 1 and the value of *slider*, the mapped color will be an interpolated color between *color2* and *color3*. A scale is drawn (in the form of a rectangle colored with the interpolated colors) in the simulation that shows the mapping of colors to turbulent values. This allows for the values of the turbulence field to be seen, which can be a helpful tool for debugging and understanding the dispersion model. On this scale is a line drawn on the rectangle showing where the *slider* value sits. The user can change the *slider* value by clicking on the line with the mouse and dragging it across the scale. By doing this, the colors of the layer will change with respect to the *slider* value. This allows for better recognition of values in the layer where there are small changes in turbulence values.

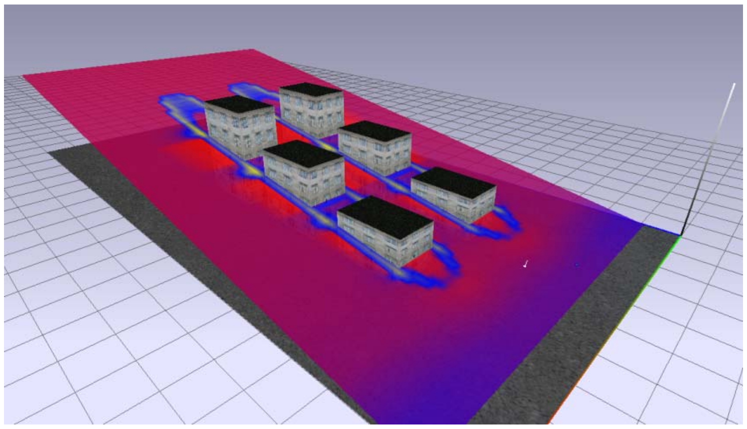


Figure 4.5:

Domain layer drawn using a rotational plane.

The second way to specify a layer to draw in the domain is to use a rotational plane that can cut the domain at any angle at any point. This allows for any possible layer to be drawn in the domain using a plane defined by a point on the plane and a normal vector that can be rotated and moved around. A point and a normal vector is all that is needed to define a plane. The point is user controlled and can be moved around to any grid intersection point in the domain using keys on the keyboard. The normal vector is also user controlled to point in any direction using keys on the keyboard. This plane, defined by a point and a normal vector, is used to intersect the domain creating intersection points on the boundary of the domain. These intersection points are then used to draw the layer. The first step of this process is to determine the normal vector. The normal vector could be initialized to anything, but it was chosen to be a vector pointing straight up in the z direction. The normal vector is then changed to point in other directions by applying a rotation to it, which causes the plane to rotate. There are three different rotations that can be applied. The first is a rotation about the local x-axis, which I'll call roll, the second is a rotation about the local y-axis, which I'll call pitch, and the third is a rotation about the local z-axis, which I'll call yaw. The reason these are local rotations is because the normal vector rotation is calculated from its current direction, and not based on the axes of the domain. Pressing keys on the keyboard is used to apply a rotation, where only one type of rotation can happen with each key press. For example, when applying a yaw rotation, the values for pitch and roll are set to zero and yaw is set to a degree of rotation. To apply a rotation to the normal vector, a transformation matrix is used which is multiplied by the normal vector (Kwon, 1998). The following calculations show how the normal vector is rotated, using the transformation matrix, A_{ij}, where i is the row and j is the column:

- A_{11} = cos(pitch) * cos(yaw)
- A_{12} = sin(roll) * sin(pitch) * cos(yaw) + cos(roll) * sin(yaw)
- A_{13} = -cos(roll) * sin(pitch) * cos(yaw) + sin(roll) * sin(yaw)
- A_{21} = -cos(pitch) * sin(yaw)
- A_{22} = -sin(roll) * sin(pitch) * sin(yaw) + cos(roll) * cos(yaw)
- A_{23} = cos(roll) * sin(pitch) * sin(yaw) + sin(roll) * cos(yaw)
- A_{31} = sin(pitch)
- A_{32} = -sin(roll) * cos(pitch)
- A_{33} = cos(roll) * cos(pitch)

The rotated normal vector components, x, y, and z, are then found with these equations, where the subscript 'p' refers to the previous normal vector:

$X = A_{11}*x_p + A_{12}*y_p + A_{13}*z_p$

$Y = A_{21}*x_p + A_{22}*y_p + A_{23}*z_p$

$Z = A_{31}*x_p + A_{32}*y_p + A_{33}*z_p$

After the normal vector is found, the intersection points of where the plane intersects the domain need to be found. The method used to find these intersection points comes from the fact that the intersection of three planes is a single point, (as long as no two planes are parallel). The rotational plane used to draw the layer is used as one of these three planes, while the other two planes are combinations of planes of the domain walls. The domain is a 3D box that has six walls (or faces). Each combination of two walls that are adjacent (or connected) is used as the two other planes in an intersection test. There are 12 different combinations of domain wall planes that potentially have intersection points with the rotational plane. All the other combination of walls of the domain can be ignored since they are parallel and won't ever intersect at a single point. The equation used to determine if the three planes intersect at a single point, P, is:

$$P =(-d_1 (n_2 \text{ x } n_3) – d_2(n_3 \text{ x } n_1) – d_3(n_1 \text{ x } n_2))/ (n_1 * (n_2 \text{ x } n_3))$$

The operator, 'x', is the cross product and the operator, '*', is the dot product. The number subscripts refer to the three planes, where d_i refers to the offset value of the i^{th} plane, and n_i refers to the normal vector of the i^{th} plane. The offset value of a plane is defined by the negative dot product of a point on the plane and its normal vector. When calculating this equation for each of the twelve combinations of domain planes with the rotational plane, a check is done first to see if the denominator is equal to zero or not. If it is equal to zero, it means there isn't a single intersection point. If it isn't equal to zero, the three planes do intersect at a single point, so the intersection point is calculated. It is possible that this intersection point could be a point outside of the domain, so only those points that are on the boundary of the domain are kept and stored in a list. The number of intersection points that can be found vary from three to six depending on how the rotational plane intersects the domain. The quad is drawn by traversing the list of intersection points. The order in which the intersection points are found and stored in a list will not always be in a clock-wise or counter clock-wise order, which is the order that OpenGL needs in order to render the quad correctly.

To get the points in a clock-wise or counter-clock-wise order, the list of points needs to be rearranged. This is done by first removing one point, P1, from the unordered list and storing it in a new ordered list (which will become the list with a correct order of points). Then the distances from each of the remaining points to P1 are calculated. The point, P2, with the minimum distance is removed and stored at the end of the ordered list. The rest of the points are selected by computing the angles that are formed using the points in the unordered list and the two points at the end of the ordered list. The point from the unordered list that creates a maximum angle is removed and stored at the end of the ordered list. More specifically, an angle is computed using a point, P, from the unordered list and the two points at the end of the ordered list, which at first will be points P1 and P2. These are the steps to compute the angle:

1. a = P1 – P2, where a is a vector created by points P1 and P2
2. b = P – P2, where b is a vector created by points P and P2
3. c = dot (a, b), where c is the dot product of vectors a and b
4. angle = arccosine(c)

This is done for each point, P, in the unordered list of points. The point, P3, that creates a maximum angle is removed from the unordered list and put at the end of the ordered list. The next point is found with the same four steps, replacing P2 and P3 with P1 and P2 respectively, (since P2 and P3 are now the two points at the end of the ordered list). This is done until there are no more points left in the unordered list, creating a list that will either be in clock-wise or counter-clock wise order. The ordered list of points is then used to draw the layer, coloring it the same way as the axis-aligned layers.

4.4 Isocontours

Isocontours are used to group areas of a domain together that have the same range of values. A weather map showing temperatures is a good example of this. The different colored regions in a weather map are isocontours that show the temperature ranges. This way, approximate temperatures can be easily determined by looking at the map. This idea was used to create isocontours for the turbulence field of the domain, allowing for approximate turbulence values to be easily determined. Isocontours are 2D regions, so in order to show them in a 3D domain, they are drawn on individual layers of the domain. These layers are drawn as quads that are parallel to the x-y, x-z, and y-z domain planes, and are colored according to the isocontour ranges. The number of isocontour regions along with the minimum and maximum turbulent value being represented determines the color of each isocontour. (Again, there can be local and global minimums and maximums). For example, if there are 5 isocontour regions, then there are 4 isocontour boundaries, each boundary being represented as a single isocontour value. The values that they receive are evenly distributed values between the minimum and maximum turbulent value. Each isocontour region gets colored according to the same color scheme that was used for drawing the domain layers.

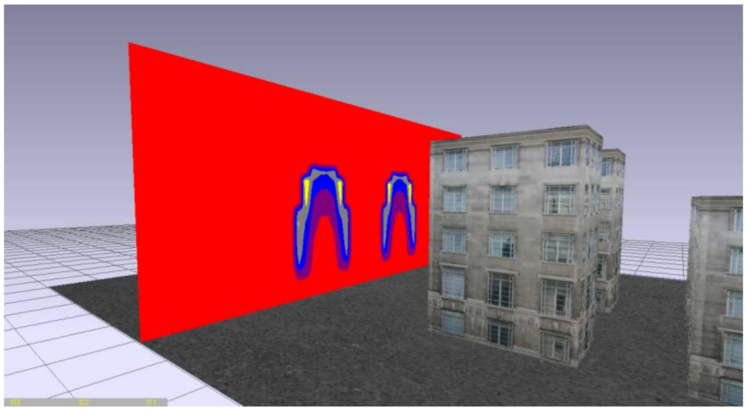


Figure 4.6: Five Isocontour regions drawn on a layer parallel to the y-z plane. Each colored region represents a range of values.

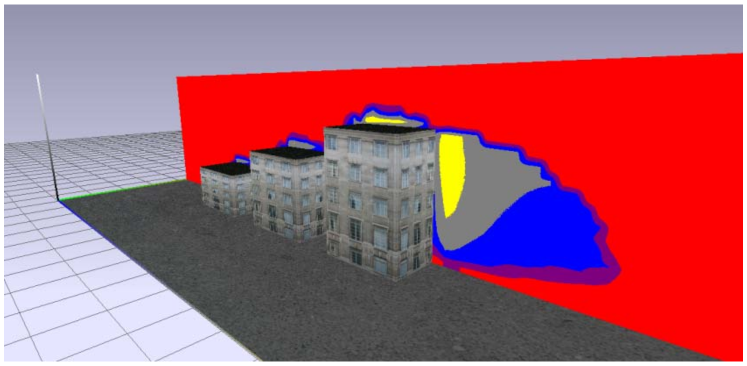


Figure 4.7: Five Isocontour regions drawn on a layer parallel to the x-z plane.

To color the isocontours, a fragment shader is used along with a 3D texture of the values of the turbulence parameter being visualized. This 3D texture is created the same way as the 3D texture used for creating the domain layers in order to provide the direct mapping between a texel in the texture and a cell in the domain. When the quad is rendered to the screen, it gets colored using a fragment shader. The fragment shader first takes the turbulent value from the texel of the 3D texture, and determines which isocontour region it is in. The isocontour line values are passed into the fragment shader so that this can be done. The isocontour line values are sorted from smallest to largest so that a number for the isocontour region can be found. This number will be 0 if the turbulent value is less than the smallest isocontour line value, 1 if it is greater than the smallest isocontour line value but less than the second smallest isocontour value, and so on. Then the color scheme used for drawing the domain layers is applied using the isocontour region number as the value to map to a color. This region number replaces the *t* variable while the *slider* variable is set to 0.5 to evenly color the regions.

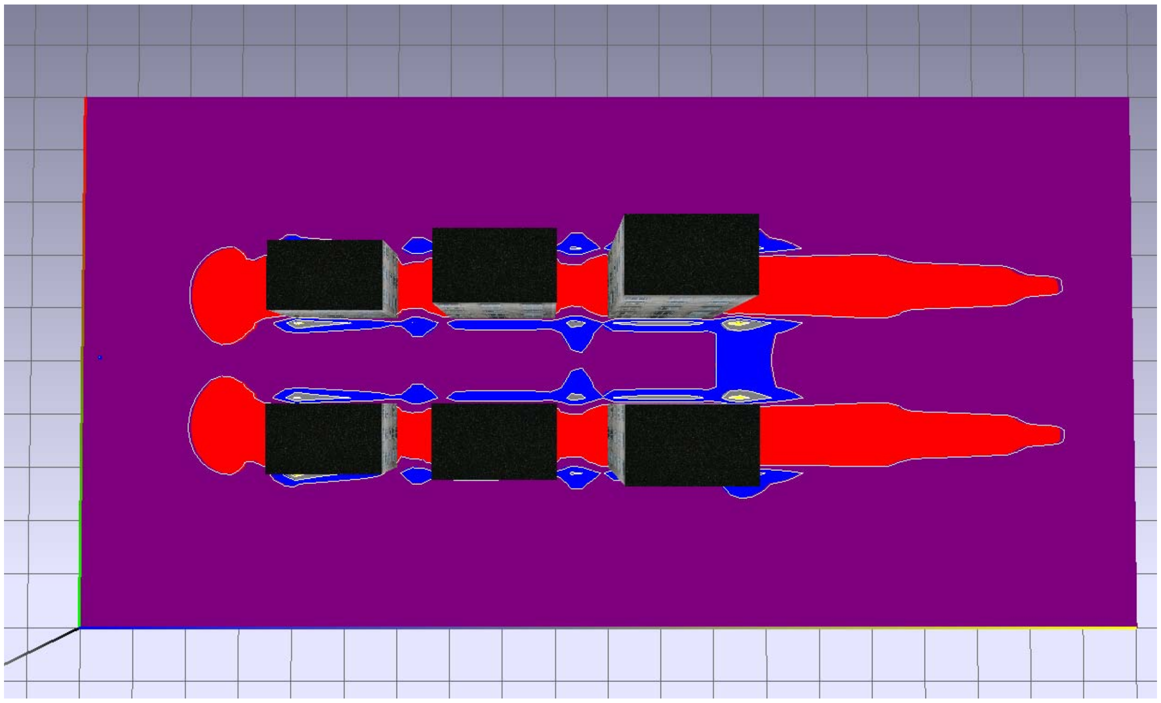


Figure 4.8: Top down view of 5 isocontour regions drawn on the ground x-y plane layer. The white lines are the isocontour boundaries drawn independent of the colored regions.

In addition to drawing the colored isocontour regions, the isocontour lines, or boundaries, can be drawn as shown in the above figure. To determine the vertices used to draw the isocontour lines, each of the cells in that layer is visited with some exception. Cells that are on the highest z value boundary, highest x value boundary, and highest y value boundary of the domain are not visited. For each cell visited, the cell to the right, the cell above, and the cell above and to the right are used to determine if any isocontour lines pass through somewhere within this group of four cells. If it is found that an isocontour line does pass through this group of four cells, an interpolation is done to determine the vertices of the line. These vertices are then stored in a list along with all the other vertices found from all of the cells visited. This list of vertices will contain all the vertices for all the isocontour lines, and can be placed in a vertex buffer object for fast rendering of the lines. The figure below shows an example layer that is parallel to the x-y plane. The cell being visited is labeled C_0, the cell to the right is C_1, the cell above is C_2, and the cell above and to the right is C_3. To determine if an isocontour line passes through these cells, a comparison is done between the values of the cells and the value of the isocontour line. If the value of the isocontour line is between the values of two neighboring cells, then that isocontour line passes between those two cells. A vertex, P, for the isocontour line can be generated between these two points using this equation:

$$P = P_i + ((\text{isovalue} - V_i) / (V_j - V_i)) * (P_j - P_i)$$

Where the variable, isovalue, is the value of the isocontour line, and V_i and V_j are the values of the two neighboring cells, C_i and C_j. P_i and P_j are the coordinates of the center of the cells of C_i and C_j.

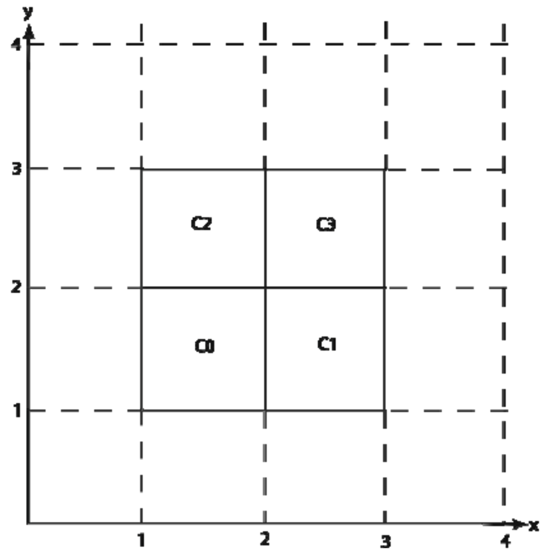


Figure 4.9:

Section of a domain layer parallel to the x-y plane. Each square is a cell of the layer holding turbulence values used for computing the isocontour boundaries.

Almost all of the time, if an isocontour line passes through a group of four cells, two or four vertices will be found, creating either one or two line segments. An even number of vertices needs to be found in order to correctly draw the isocontour lines. It is possible that an odd number could be found (in rare cases), which is taken care of by removing the last added vertex.

4.5 Isosurfaces

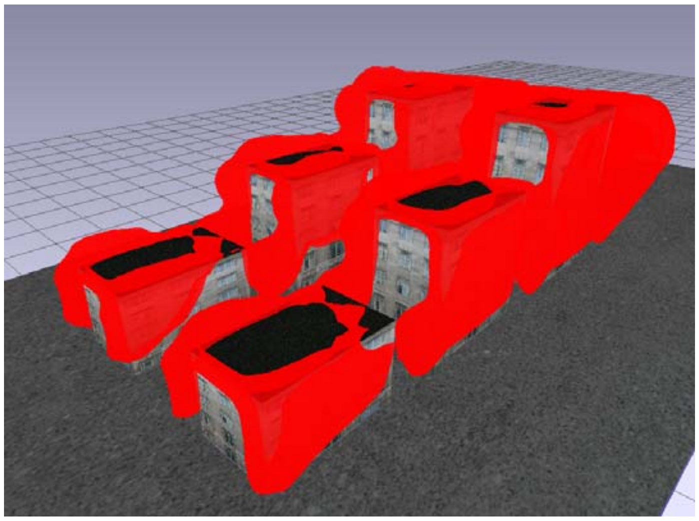


Figure 4.10:

Isosurface drawn as a solid 3D volume.

An isosurface is a 3D version of isocontours, showing a region of a range of values throughout the whole domain as a 3D volume. The algorithm used to construct an isosurface was a modified version of the marching cubes algorithm found in the book, GPU Gems 3 (Geiss, 2008). The basic idea of this algorithm is to subdivide a domain into evenly sized 3D cells and then visit each of these cells to create the 3D volume. The subdivision is variable, where the finer the mesh size means the smoother the surface is. This idea is used to create isosurfaces representing ranges of turbulence values by visiting each cell of the domain to test if it lies on a boundary between two isosurface regions. Geometry is created for the cells that lie on the boundary between two regions to make up part of the isosurface at that cell location. The geometry created from each cell creates a 3D volume. To test if a cell lies on the boundary of an isosurface, the turbulent values of the eight corners of the cell are applied to a "density function", (which is the name of the function given in GPU Gems 3). The density function is used to determine which side of a boundary the corner of a cell lies on. The density function used for finding the isosurface boundaries is:

$$\text{Density}(t) = t - \textit{isovalue}$$

The variable *t*, is the turbulent value and *isovalue* is the isosurface value, (which is a turbulent value separating two ranges of values).If the value of this density function is negative, then the turbulent value for that corner is less than the isosurface value. If it is positive, then the turbulent value for that corner is greater than the isosurface value. Each cell is given a "case number", which is a number generated from the density values of the eight corners of the cell. To generate a case number, each corner of the cell is assigned a bit value. If the density function value of the corner is positive, then it is given a bit value of 1. If the density function value is negative, it is given a bit value of 0. The case number is the bitwise OR operation of the eight bits produced. The following figure shows the numbering of the corners and edges of the cell.

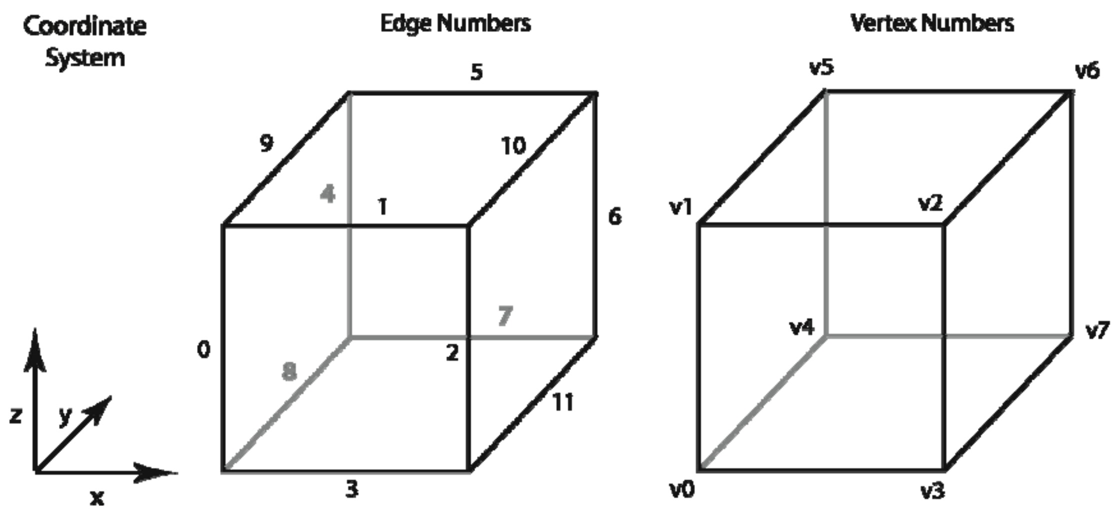


Figure 4.11: Assignment of edge numbers and vertex numbers shown for a cell, along with the orientation.

The following equation determines the case number, C:

C = v7|v6|v5|v4|v3|v2|v1|v0.

An example of this equation would be if v7, v5, v2, and v0 were bits set to 1 and the rest were set to 0. The case number, C, would be the value 1|0|1|0|0|1|0|1, which equals 165. The case number will be in the range from 0 to 255, which is used as the index into a table to determine how many triangles to draw. The number of triangles to draw will be a value from 0 to 5. This table is in the form of a 1D array holding the values of how many triangles to draw for each case number. Another lookup is performed into a table to determine the edge numbers used to build the triangles. This table is in the form of a 2D array of 3D vectors, where the case number is the row number and the column numbers are 0 to 5 (the number of triangles to draw). The edge numbers used to construct the triangles are the 3D vectors, where each 3D vector is the edge numbers for one triangle. For example, if there are 2 triangles to draw, the first triangle will be built using the edge numbers of the 3D vector in column 0 (of the appropriate row) and the second will be built with the 3D vector in column 1. The exact position to place a vertex on an edge will be an interpolated position using the density values of the cell corners of that edge. (This position will be the place along that edge where the density value is approximately 0, corresponding to the isosurface being created). There are 14 basic cases of the geometry created for a cell that lies on the boundary, where the other 240 cases are inversions and rotations (reference GPU Gems 3). The tables used to determine the number of triangles to draw and how to construct those triangles are provided in a DVD included with the book, GPU Gems 3. (The authors encourage use of these tables, so they were used for this implementation). These tables are stored on the GPU using textures.

The first step to implementing this algorithm is to compute the density values for each corner of a cell with the density function. A 3D texture is used to store the density value of each cell corner. The size of this 3D texture is one unit higher in each domain dimension, since the values being stored are for the corners of the cells. A fragment shader is used to look up the turbulence values for each corner of the cells, compute the density values and store them in the 3D texture. In order to render to (or store values in) a 3D texture, each layer of the 3D texture has to be rendered individually. The layers chosen to render at a time were horizontal layers, where each corner of a cell has the same height value. The cell corners are given a density value and stored in the corresponding texel location in the 3D texture. After computing all the density values for each corner of a cell, the second step is to build the 3D volume creating the isosurface.

The second step to implementing this algorithm is to build the triangles that create the 3D volume. To do this, each of the cells is visited one by one. A geometry shader is then used to assign a zero or one bit to each of the corners of the cells by looking up the density values in the 3D texture of density values. (Remember that a geometry shader takes a vertex as input and can create geometry from that vertex, which allows for the triangles to be created). The case number is then computed using the equation above, and used to determine the number of triangles to build. For each set of three edges, (found using the table of edges), three vertices are found using interpolation and the triangle is created from those three vertices. Instead of rendering the triangles created from the geometry shader to the screen, they are captured and stored into a vertex buffer object. This is done so that this algorithm only needs to be run once for each isosurface, storing the triangles that make up each isosurface in a vertex buffer object. Then the isosurface can be rendered to the screen using the triangles stored in the vertex buffer object.

5. Results

5.1 Validation

The GPU version of QUIC-Plume was compared to the original implementation on the CPU for validation. The comparison results for (Willemsen et al. 2007) and (Pardyjak et al. 2007) were generated by Balwinder Singh and Eric Pardyjak, engineers at the University of Utah. The comparison of concentration profiles of the GPU and CPU implementations show that the GPU implementation produces correct results. The Gaussian dispersion model was the first test case implemented and verified (Willemsen, Norgren, Singh, & Pardyjak, Development of a new methodology for improving urban fast response lagrangian dispersion simulation via parallelism on the graphics processing unit, 2007). The concentrations of particles in the domain are used as the verification method to determine if the GPU version is correct. The test case was run with 100,000 particles being emitted from a source at a continuous rate of 100 particles per second. This rate was determined by setting the time step fixed at one second, and the duration of the simulation to 1000 seconds. Particles were emitted from a sphere source with a radius of 0.2 m, located at x = 20 m, y = 49 m, and z = 70 m. The size of the domain was 100 m by 100 m by 140 m, in the x, y, and z directions respectively. The cell constant value, u_*, was set to 0.18 ms^{-1} and the uniform in U wind speed was set to 2 ms^{-1}. The concentrations were accumulated after 200 seconds from the start of the simulation for the remaining 800 seconds. They were found using collection boxes that spanned the entire domain, with 8, 100, and 140 boxes used in the x, y, and z directions.

The following figure compares the normalized concentration values, C^*, of the GPU and CPU implementation against the Gaussian solution. The graph, (a), shows the comparison along the plume centerline, where h is the height value equal to 70 m. The graph, (b), is a lateral profile at x/h = 0.7 and (c) is a vertical profile at x/h = 0.7. The GPU and CPU implementations produce concentrations that are very close to the same. They are a little different, however, which is most likely due to the difference between the random numbers used.

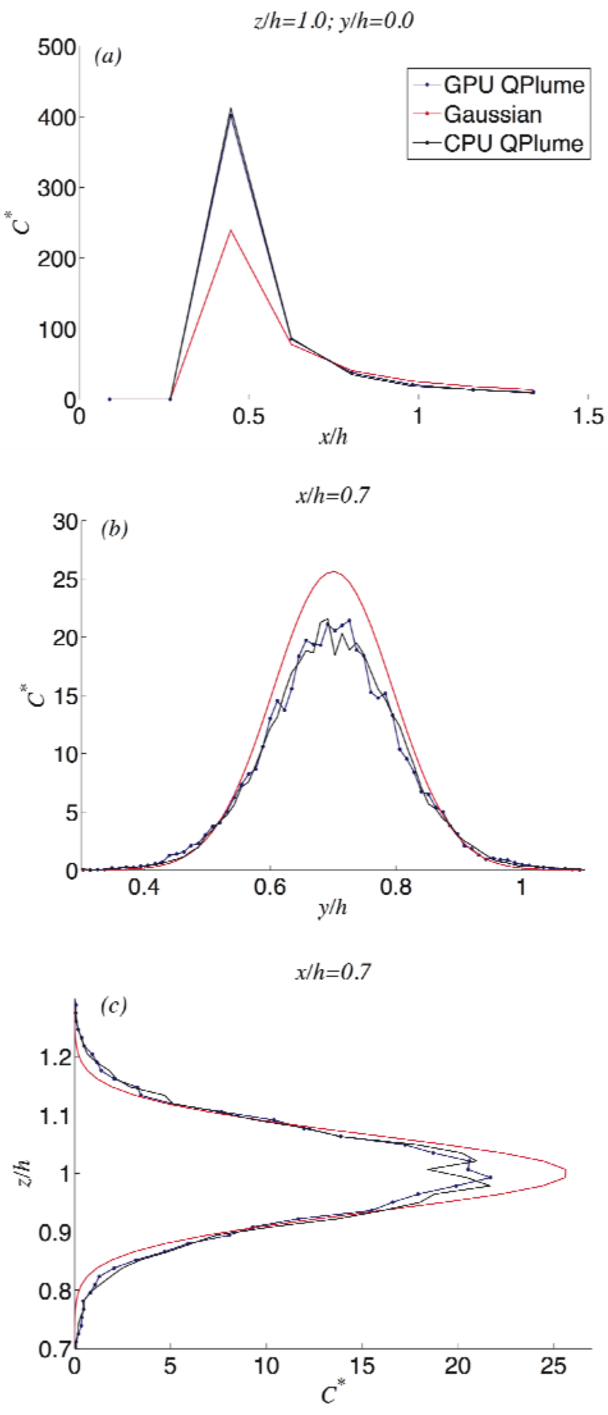


Figure 5.1:

Graphs from (Willemsen et al. 2007).

Graph (a) shows the concentration values along the plume centerline.

Graph (b) shows the concentration values of a lateral profile at x/h = 0.7.

Graph (c) shows the concentration values of a vertical profile at x/h = 0.7.

The model that introduces buildings into the domain was tested with a single cubical building placed at x = 25 m, y = 25 m, and z = 0 m with a height, H, of 10 m (Pardyjak, Singh, Norgren, & Willemsen, 2007). 50,000 particles were continuously emitted from a sphere source with a radius of 0.1 m, located at x = 5 m, y = 25 m, and z = 3m, in a domain that was 100m x 50m x 20m in the x, y, and z directions. QUIC-URB was used to generate the wind field and turbulence field for the test case. The normalized concentrations, C^*, were accumulated for a simulation that ran for 1000 seconds. The following figure shows the lateral concentration profiles found comparing the GPU and CPU implementations. The graph, (a), shows the lateral profile at x = 9.75 m and z = 2.5 m. This is at a location between the source and the building. As you can see, the concentration values are the same. The graph, (b), shows the lateral profile at x = 57.25 m and z = 2.5 m. This location shows the concentrations of particles after they've passed the building. Again, the profiles are very similar with a little difference seen due to the implementation differences. QUIC-Plume has an extra step, that is doesn't in the GPU version, when advecting the particles involving a coordinate rotation (Williams, Brown, Boswell, Singh, & Pardyjak, 2004). This could cause the slight differences in concentrations seen at the edges of the building. The third graph, (c), is a lateral profile of particle concentrations at x = 85.75 m and z = 2.5 m. The profiles are very similar again with the differences due to the difference in implementations and the difference in random numbers used.

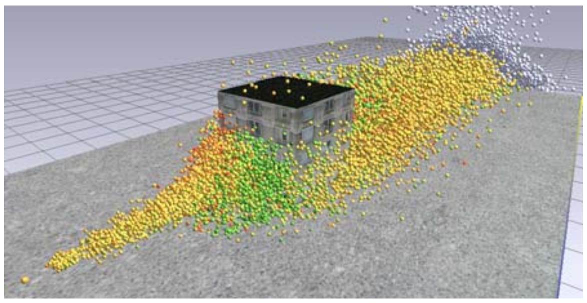


Figure 5.2:

Shows the single building test case.

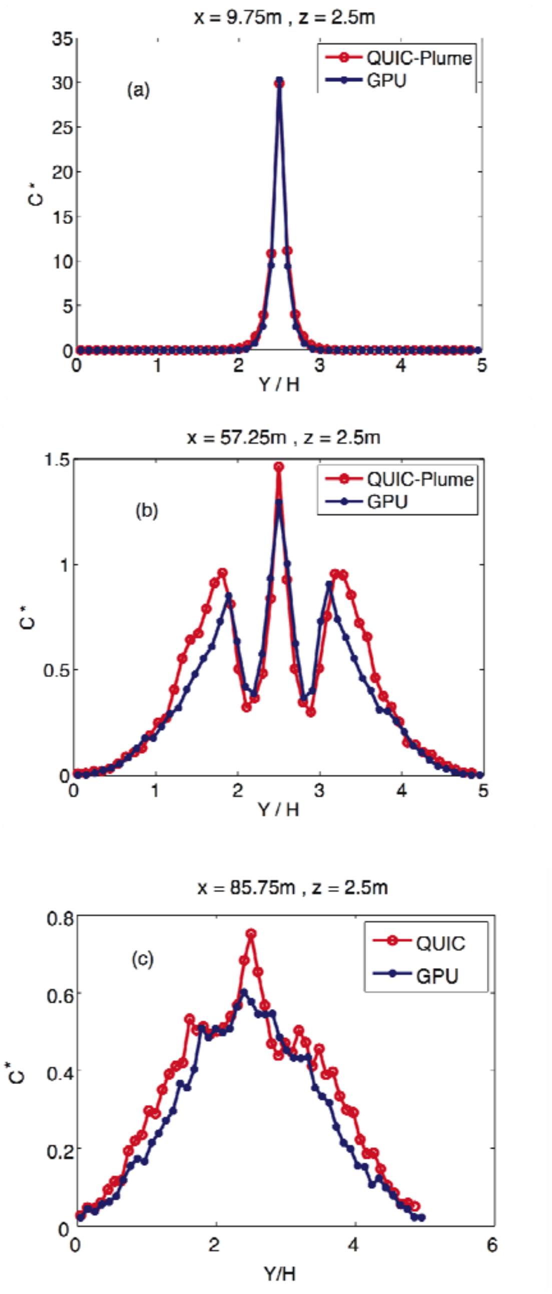


Figure 5.3:

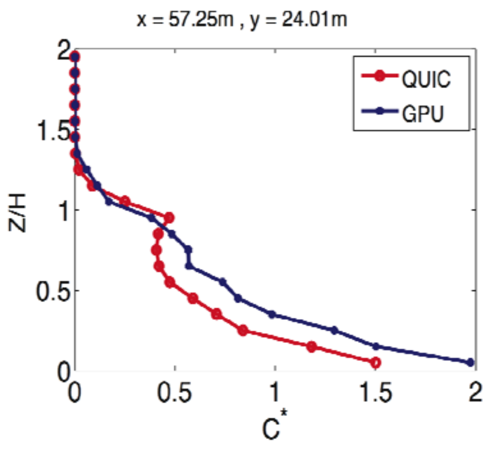
Graphs from (Pardyjak et al. 2007).

Graph (a) shows a lateral concentration profile at x = 9.75 m and z = 2.5 m.

Graph (b) shows the lateral concentration profile at x = 57.25 m and z = 2.5 m.

Graph (c) shows the lateral concentration profile at x = 85.75 m and z = 2.5 m.

The next figure compares the vertical concentration profiles of the GPU and CPU implementations. Graph (a) is vertical profile at x = 57.25 m and y = 24.01 m and graph (b) is a vertical profile at x = 85.75 m and y = 24.01 m. The results show that the concentration profiles are again very similar with the differences due to the differences in the GPU and CPU implementations.



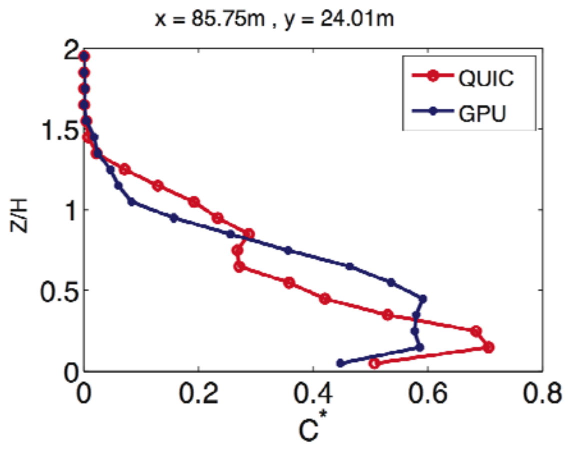


Figure 5.4:

Graphs from (Pardyjak et al. 2007).

The first graph shows a vertical concentration profile at x = 57.25 m and y = 24.01 m. The second graph shows a vertical concentration profile at x = 85.75 m and y = 24.01 m.

5.2 Performance

There is a great increase in performance obtained with the GPU implementation of QUIC-Plume. To show this, the time taken to advect N particles was averaged over 1000 time steps. The N particles were released instantaneously during the first time step in order to get the average amount of time the implementation takes to advect N particles per time step. Note that the visualizations were turned off for the GPU implementation in order to get an accurate comparison. A range of values for N was used to show the performance gain the GPU achieves. The following graph compares the performance of the CPU and GPU implementations for the Gaussian test case. The timings were taken using a 2.4 Ghz Intel Core 2 Duo Processor with an NVIDIA GeForce 8800 GTS graphics card. The results show that the GPU outperforms the CPU by up to three orders of magnitude for particle amounts of 100,000 and more (Willemsen, Norgren, Singh, & Pardyjak, Development of a new methodology for improving urban fast response lagrangian dispersion simulation via parallelism on the graphics processing unit, 2007).

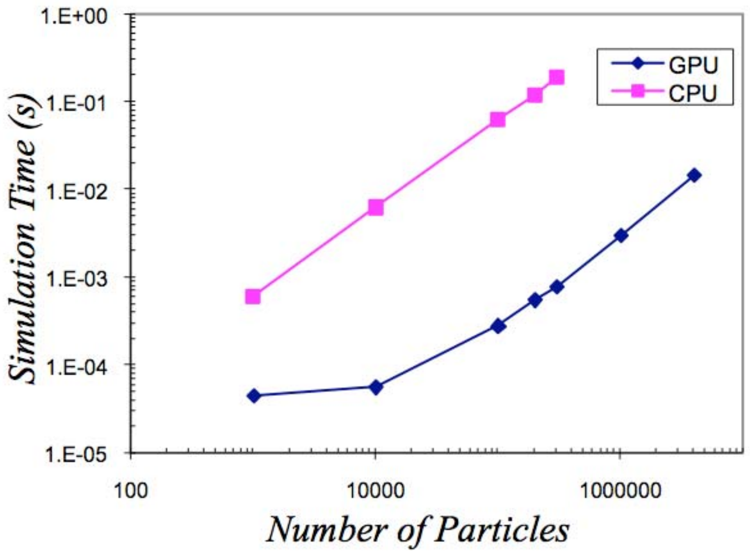


Figure 5.5:

Graph from (Willemsen et al. 2007). Compares the average time per advection step of the GPU and CPU for the Gaussian test case.

The same timing test was done to compare the performance of the single building test case (Pardyjak, Singh, Norgren, & Willemsen, 2007). The GPU performance gain for this case is a little less than the Gaussian case. This is caused by the additional textures and operations used for performing reflections, applying the CFL condition, and storing current per particle directions. The GPU implementation still provides a significant performance increase as shown in the following graph. These results are again found using a 2.4 Ghz Intel Core 2 Duo Processor with an NVIDIA GeForce 8800 GTS graphics card.

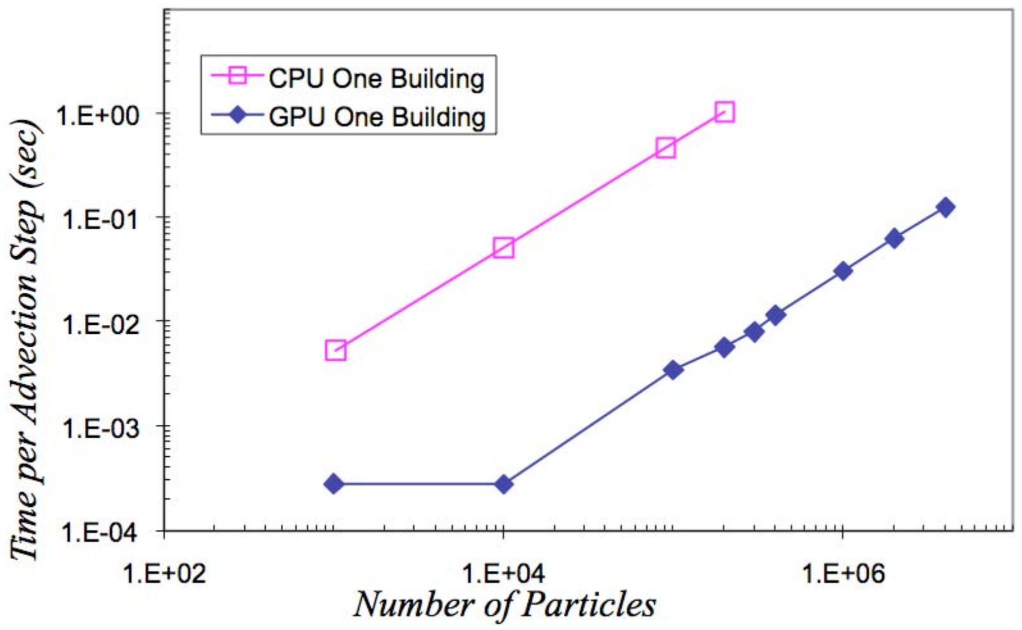


Figure 5.6: Compares the average time per advection step of the GPU and CPU for the single building test case. Graph from (Pardyjak et al. 2007).

To further evaluate our implementation, additional timing tests were used to compare the performance when increasing the amount of buildings in the domain. Here the time step was set to 0.1 seconds. Three different cases were used to compare the performance of a domain with a single building, six buildings, and thirty-five buildings for the GPU and CPU implementations. These results were obtained using an AMD Athlon 64 X2 Dual Core Processor 4800+ and an NVIDIA GeForce 8800 GTS graphics card. The single building test case described above was used for the timings for the single building case. For the six building case, the domain was 100m x 50m x 20m in the x, y, and z directions respectively with the source located at x = 5m, y = 25m and z = 3m. The buildings were placed as shown in the following figure.

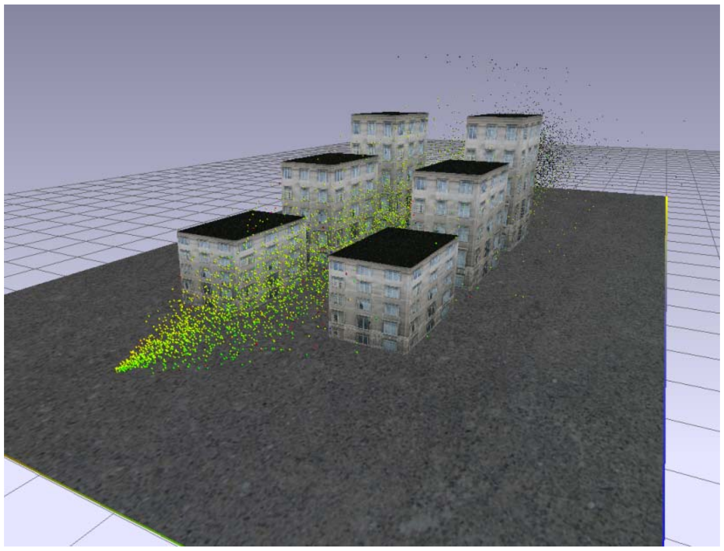


Figure 5.7:

Shows the six building case.

The thirty-five building case uses a domain 100m x 120m x 30m in the x, y, and z directions, with a line source defined by the two points, x = 10m, y = 115m, z= 3m, and x = 90m, y = 115m, z = 5m. The placement of buildings creates a dense area of buildings similar to that of a downtown area, shown in the following figure. This creates a simulation so that the particles will enter into this dense area of buildings requiring many reflections.

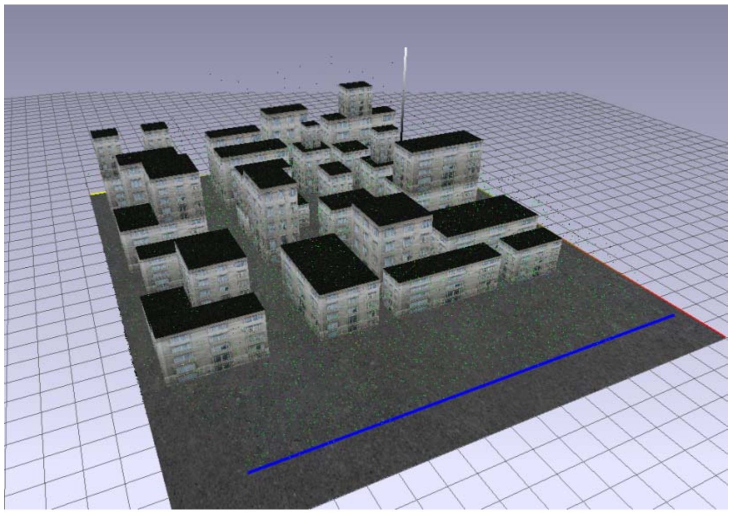


Figure 5.8:

Shows the 35 building case. The blue line is the line source emitting particles that flow into the buildings.

The timing results for these test cases are shown in the following table. The performance of the GPU implementation consistently outperforms the CPU implementation by roughly the same amount for each case. Looking at the performance of the GPU for the single building and six building cases, the performance is slightly better for the six building case. The reason for this is most likely because the single building case is set up so that most of the particles will reflect off of the building. In the six building case, many particles will flow between the middle of the buildings without having to be reflected. The performance of the thirty-five building case is slightly reduced from the other two cases, because of the large increase in the amount of reflections. However, the reduction isn't much meaning that adding buildings doesn't have a large negative affect on performance.

Table 5.1: Time per advection step (sec)

	1 Building		6 Buildings		35 Buildings	
# Particles	CPU	GPU	CPU	GPU	CPU	GPU
1,000	0.0168	0.000203	0.0133	0.000198	0.0259	0.000192
10,000	0.0558	0.000217	0.0449	0.000203	0.0696	0.000224
100,000	0.422	0.00114	0.368	0.000898	0.567	0.00126
300,000	1.24	0.00325	1.09	0.00249	1.67	0.00365
500,000	2.07	0.00532	1.82	0.00420	2.78	0.00598
1,000,000		0.0104		0.00827		0.0119
2,000,000		0.0216		0.0168		0.0234
3,000,000		0.0312		0.0251		0.0354
4,000,000		0.0479		0.0332		0.0514

6. Conclusion

It is shown that the GPU version of QUIC-Plume was successfully implemented, providing accurate results while increasing performance. This GPU implementation can be used for fast real-time results of particle dispersion and as a 3D visualization tool to help engineers in developing and understanding dispersion models. Future work includes modifications and addition of 3D tools, simulating greater amounts of particles, and allowing for additional building shapes. Also, systems with multiple GPUs could possibly be used to parallelize the particle dispersion even more and potentially provide a higher increase in performance.

When generating path lines using the second method described in section 4.2, the length of the path lines are limited by the largest width size of a 2D texture. Additional 2D textures could be used to store additional points when the amount of points stored for the path lines become larger than the width of the texture. Another modification to the path lines would be to only generate subsets of path lines in specified regions of the domain. This could be used to help see the behavior of particles when the simulation becomes cluttered with large amounts of particles.

The amount of particles available to simulate is limited by the size of 2D textures and the amount of memory on the graphics card. The maximum size of a 2D texture is 8192 by 8192 on an NVIDIA Geforce 8800, theoretically meaning that the amount of particles that can be simulated is about 64 million. However in practice, the amount of particles is also limited by the amount of memory available on the graphics card. For example, if the number of particles being simulated is 1 million, there will be five 2D textures the size of 1000x1000 (*previous_positions*, *new_positions*, *previous_velocities*, *new_velocities*, and the *random_values* texture). Each of these textures store 32-bit floats and are 15.25 MB ($1000*4*32/(8*1024^2)$). These five textures together hold 76.3 MB. Then there will also be five textures holding the domain data (the *wind_field*, *lambda*, *tau_dz*, *duvw_dz*, and *cell_type* textures). If the domain is 100m by 100m by 50m and each cell is 1 cubic meter, then each of these textures hold approximately 7.63 MB, combining to 38.1 MB. These ten textures are the textures required to run the core simulation, which requires 114.4 MB. However, there are additional textures required to color the particles, store building information, draw the isosurfaces, etc., along with additional memory used for displaying the visualization. If the amount of particles to simulate increases to 4 million using the same domain, 343.28 MB would be required. The simulation is stable with 4 million particles on an NVIDIA GeForce 8800 GTS with 640 MB of memory. 16 million particles would require 1221 MB, which is well over the size of the NVIDIA GeForce 8800 Ultra that has the largest amount of memory available at 768 MB. If the amount of memory available was not an issue, textures of size 8192 by 8192 could be used to simulate over 16 million, and additional textures could also be used to simulate even more particles by operating on each texture individually each time step.

An improvement could be made to increase the performance of the concentration calculations. Currently the process of calculating concentrations requires transferring all the particle positions from the GPU to the CPU. To speed this process up, a solution that eliminates the large GPU to CPU transfers could be implemented. This would require storing the concentration box information on the GPU in textures and being able to update the values with the use of shaders.

Bibliography

Budd, C. (2003, May). How maths can make you rich and famous: Part II. *plus magazine* (25).

Galoppo, N., Govindaraju, N. K., Henson, M., & Manocha, D. (2005). Efficient alogirhtms for solving dense linear systems on graphics hardware. *In Proceedings of the 2005 ACM/IEEE conference on supercomputing*, (pp. 1-12).

Geiss, R. (2008). Generating Complex Procedural Terrains Using the GPU. In H. Nguyen (Ed.), *GPU Gems 3* (pp. 7-37). Addison-Wesley.

Goodnight, N., Woolley, C., Lewin, G., Luebke, D., & Humphreys, G. (2003). A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware. *ACM Siggraph/Eurographics conference on Graphics hardware*, (pp. 102-111).

Gribble, C. P., & Parker, S. G. (2006). Enhancing Interactive Particle Visualization with Advanced Shading Models. *3rd Symposium on Applied perception in graphics and visualization*, *153*, pp. 111-118. Boston.

Harris, M. J., Baxter, W., Scheuermann, T., & Lastra, A. (2003). Simulation of cloud dynamics on graphics hardware. *In Proceedings of the ACM Siggraph/Eurographics conference on Graphics hardware*, (pp. 92-101).

Kipfer, P., Segal, M., & Westermann, R. (2004). UberFlow: A GPU-Based Particle Engine. *ACM Sigraph/Eurographics Conference on Graphics hardware*, (pp. 115-122).

Kolb, A., Latta, L., & Rezk-Salama, C. (2004). Hardware-based Simulation and Collision Detection for Large Particle Systems. *ACM Siggraph/Eurographics conference on Graphics hardware*, (pp. 123-131).

Kruger, J., Kipfer, P., Kondratieva, P., & Westermann, R. (2005). A Particle System for Interactive Visualization of 3D Flows. *IEEE Transactions on Visualization and Computer Graphics , 11* (6), 744-756.

Kwon, Y.-H. (1998). *Orientation Angles*. Retrieved 2008, from http://www.kwon3d.com/theory/euler/euler_angles.html

Pardyjak, E., & Brown, M. (2001). Evaluation of a fast-response urban wind model comparison to single-building wind-tunnel data. *2001 International Symposium on Environmental Hydraulics.*

Pardyjak, E., Singh, B., Norgren, A., & Willemsen, P. (2007). Using video game technology to achieve low-cost speed up of emergency response urban dispersion simulations. *AMETSOC Conf Coastal Atmospheric Prediction.*

Qui, F., Zhao, Y., Fan, Z., Wei, X., Lorenz, H., Wang, J., et al. (2004). Dispersion Simulation and Visualization for Urban Security. *15th IEEE Visualization 2004* , 553-560.

Rost, R. J. (2006). *OpenGL Shading Language Second Edition.* Addison-Wesley.

Scheidegger, C., Comba, J., & Cunha, R. (2005). Practical CFD Simulations on the GPU using SMAC. *Computer Graphics Forum , 24* (4), 715-728.

Shreiner, D., Woo, M., Neider, J., & Davis, T. (2006). *OpenGL Programming Guide.* Addison-Wesley.

Singh, B., Williams, M. D., Pardyjak, E. R., & Brown, M. J. (2004). Development and testing of a dispersion model for flow around buildings. *4th AMS Symposium on Urban Environments.*

Sunday, D. (2006). *Intersections of Lines, Segments and Planes (2D and 3D)*. Retrieved 2008, from softSurfer: http://geometryalgorithms.com/Archive/algorithm_0104/alogirhtm_0104B.htm

Trancoso, P., & Charalambous, M. (2005). Exploring Graphics Processor Performance for General Purpose Applications. *8th Euromicro Conference on Digital System Design*, (pp. 306-313).

West, M. (2007, March). Particle Fluid Dynamis: Part 1. *gamedeveloper* , 43-47.

Willemsen, P., Norgren, A., Singh, B., & Pardyjak, E. (2007). Development of a new methodology for improving urban fast response lagrangian dispersion simulation via parallelism on the graphics processing unit. *11th Intl Conf on Harmonisation within Atmospheric Dispersion Modeling.*

Willemsen, P., Norgren, A., Singh, B., & Pardyjak, E. (2008). Integrating Particle Dispersion Models into Real-time Virtual Enviornments. *14th Eurographics Symposium on Virtual Enviornments.*

Williams, M. D., Brown, M. J., Boswell, D., Singh, B., & Pardyjak, E. (2004). Testing of the quic-plume model with wind-tunnel measurements for a high-rise building. *Fifth AMS Symposium on the Urban Environment.*