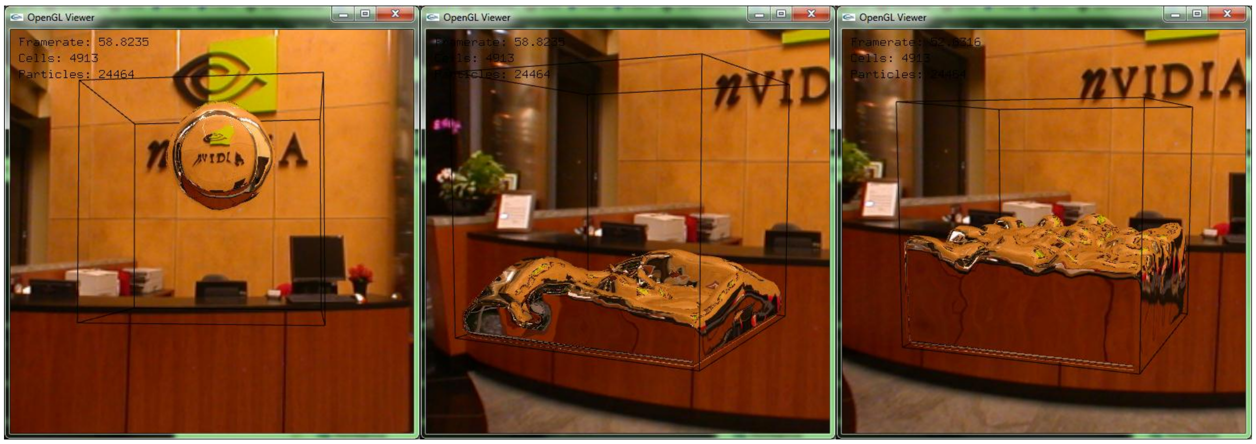
Real-Time Fluids with Advanced Shaders

Kevin Todisco



```
Framerate: 58.8235
Cells: 4913
Particles: 24464
```

Figure 1: A large scale example of the simulation. The leftmost image shows the beginning of the test case, and shows how the fluid refracts the environment around it. The middle image is taken just after the ball of water hits the bottom of the box. The rightmost image shows the fluid after it has stabilized. Notice that it appears to have gained volume, a limitation which I discuss in Section 4.

Abstract

I present a real-time implementation of realistic 3D fluid, and render it using an advanced image-space refraction method, creating the look of a fluid such as water. The water surface is constructed using the Marching Cubes algorithm, which I implement as a geometry shader.

Introduction

Fluid simulation in 3-dimensional space has long been a task which required offline computation to produce physically realistic results in both behavior and appearance. As a result, it was not possible to incorporate an advanced fluid simulation into a necessarily real-time setting such as a video game; thus, fluid in game environments, such as water, are often represented using various cheap tricks like refractive shaders applied on a plane, or simple particle and mass spring systems to represent a full body of fluid.

Recent 2D games have incorporated water as a core element, such as PixelJunk's *Shooter* or *Where's My Water* for mobile devices. Both systems are still restricted to a particle system, with various means of constructing the fluid's surface around the particles. I've found that *Shooter* did not produce visually pleasing results; the resulting fluid was clunky and had hard edges. *Where's My Water* produced a more believable look, but the particle basis could still be identified.

My method achieves realistic behavior through the use of the Marker-and-Cell (MAC) method, and it does so in 3-dimensional space in real-time. In addition, it applies a shader which approximates refraction through two surfaces to the resulting fluid mesh, achieving the more complex refractive effects of a fluid such as water. Figure 1 shows a large test case of my program.

In the next section, I will describe the previous work which I used as a basis for this project. In Section 3, I begin to describe the implementation of my project, and break this down into four parts: the parallelization of the Navier-Stokes fluid simulation, the approximated refraction shader, the extension of this refraction to dynamic scenes, and the marching cubes geometry shader. In Section 4, I discuss limitations of my method, and follow up with future work in Section 5, before giving results and concluding in Section 6.

2 Previous Work

Foster and Metaxas [1] presented an implementation of the MAC fluid simulation method, and achieved behaviorally realistic results through the solving of the Navier-Stokes fluid equations. Their computation, however, was done offline due to the hardware available to them at the time.

Nvidia Researchers [3] have recently presented a real-time implementation of 3D fluids in GPU Gems 3. Their method represents a robust approach to accomplishing the task; my approach is similar in nature, but employs different methods for both the physics simulation and rendering.

Chris Wyman [2] described an approximate image-space approach for computing complex refraction in interactive times. I employ his method and extend it to handle dynamic scene surrounding the refractive object.

My method combines techniques described in all three papers to accomplish a real-time 3D fluid.

3 Implementation

3.1 Parallel Navier Stokes

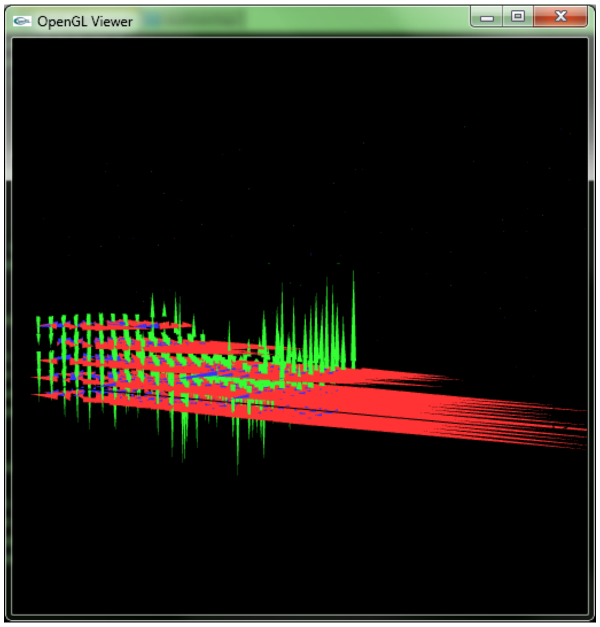


Figure 2: A visualization of the fluid simulation cell velocities.

In order to achieve real-time, the fluid simulation steps must be executed in parallel. While small test cases (i.e. less than 1000 cells and less than 10,000 particles) will run at appropriate speed in a serial implementation, large cases, which would be necessary for complex game levels, require a faster execution method.

To perform the parallelization of the physics simulation, I utilize Nvidia’s CUDA programming language, which is specifically intended for General Purpose GPU

Programming. An overview of CUDA is given in Appendix A.

The fluid simulation steps, as described by Foster and Metaxas, are easily executed in parallel. One only has to take caution that two CUDA threads do not attempt to write to the same location at the same time. Thus, since each cell's computation executes in a single thread, cells may not modify other cell's values, but instead store their intended modification internally, and this may be applied by the cell which was intended to be modified in a second pass. Simultaneous reads of data from other cells are permitted. The only exception to the ease of parallel fluid calculations is enforcement of fluid incompressibility, which I will discuss in the section on limitations. Figure 2 shows a visualization of the cell velocities simulated as part of the Navier-Stokes formulas.

3.2 Approximate Two-Surface Refraction

To properly implement Chris Wyman's technique for interactive refraction, two specific implementation details need to be noted. The first regards the manual projection of a 3-dimensional point into 2-dimensional screen space. The second involves the extraction of a pixel's depth value stored in the depth texture from the first rendering pass.

When taking a 3D point and projecting it into 2D space, the traditional method is to first multiply the point by the projection matrix. At this point, it's necessary to divide all components by w, a step which I excluded and which was ultimately the cause of some unexpected visual results. Even after screen coordinates are obtained, one additional step must be taken to map them onto a [0,1] range by dividing by the screen dimensions. Therefore the screen sizes must be passed as uniforms to the shader.

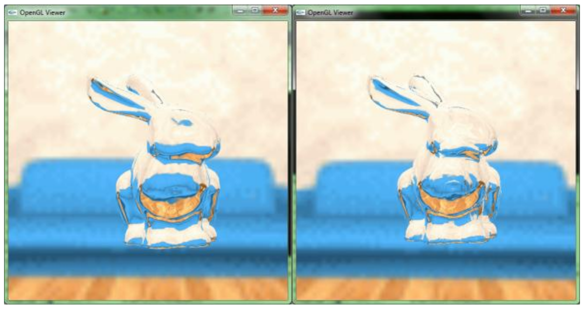


Figure 3: The image on the left shows a one-surface refraction. The right image shows refraction through two surfaces. More interesting effects can be seen in the right image.

Once the texture coordinates are computed, they are used to look up the normal and depth value in the render textures from the first rendering pass. The normal is straightforward; it is directly obtained from the color value in the texture, with no additional computation needed, save excluding the alpha value (which will be 1.0, and must be changed to 0.0). The depth however, is stored by OpenGL in a non-linear fashion. Therefore, most depth values in the texture will be close to 1.0. A computation is needed to convert this non-linear value back into a linear mapping. That computation can be accomplished elegantly with the following line:

```
float orig_d = 2*ZFar*ZNear / (ZFar + ZNear
- (ZFar - Znear)*(2*depth.x - 1));
```

The resulting value is the original depth in world units, and therefore is sufficient for the next step of the algorithm, which is computing the intersection point on the back-side of the refractive object in world space.

3.3 Extension to Dynamic Scenes

The extension of Wyman's algorithm to dynamic scenes is straightforward: render the scene around the object to a cubemap. This step therefore requires six rendering passes, one for each positive and negative primary world axis. The resulting cubemap is used in the refraction shader for color lookups once a twice-refracted ray is computed.

The exact point at which the camera should be placed, or how many viewpoints should be rendered is an area for additional research. Placing the camera at the center of the object gives good results for objects located far from the object, but poor results for objects close to it (see Figure 4). Alternatives for number of rendering passes include increasing the field-of-view to 180 degrees and rendering twice, or rendering once with a 360 degree fisheye. Doing so would alter the color lookup method in the shader, and were not included in this implementation.

3.4 Marching Cubes in a Geometry Shader

The challenge of implementing marching cubes in a geometry shader lies in the complexity of the marching cubes algorithm. If one is not careful, it's possible to exceed the instruction limit for a single shader program, even with today's hardware. It's therefore necessary to pass as much pre-computed data as possible to the shader to avoid repeated computation of static information. This data includes the various unique marching cubes cases.

In the implementation, I store the cases as a 15x256 2D texture. A texture is used to allow for the passing of large datasets into a shader program; memory allocation and shader language limitations prevent the cases being passed as a more traditional 2D array. The texture itself contains integers, and therefore OpenGL must be told to store the values as 16-bit alpha integers, to avoid a conversion to floating point. For each case of marching cubes, the 15 integers correspond to five sets of triangle vertices specified by the edge indices of the cube. For cases of marching cubes which generate less than five triangles, the data is padded with values of -1.

The shader itself emits triangle strips with a maximum of 15 vertices. Unfortunately this means that the data passed down the rendering pipeline is not optimized, since each triangle strip will consist of only one triangle. The necessity of triangle strips is dictated by the current geometry shader model of OpenGL.

4 Limitations

The current implementation is limited to representing the simulation space as a box with planar boundaries. Additional boundary velocity logic would need to be added to allow for more unique shapes of fluid flow, but the entire simulation space itself would still be best represented as a box.

My implementation also does not account for incompressibility of liquid. This is a result of the mathematical difficulties in solving all cells for zero divergence in parallel. The serial version of the program is more straightforward, as divergences are constantly changed when iterating over cells. It might be the case that more passes are required to enforce incompressibility in parallel, but this could be avoided if a one-pass solution, such as solving a matrix, could be found. Without enforcing incompressibility, the fluid has a tendency to gain in volume as time progresses.

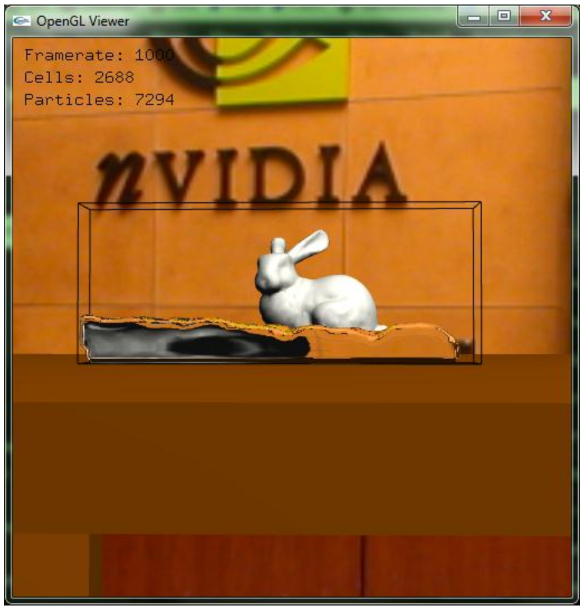


Figure 4: The bunny is improperly refracted by the water. It appears to be left of where it actually is.

In addition, because the render-to-cubemap step renders from the point of view of the refractive mesh, objects which are close to the mesh will not appear to be properly refracted, as demonstrated in Figure 4.

5 Future Work

Additional research should be conducted about methods to reconstruct the water's surface. While the Marching Cubes algorithm was successfully implemented in a geometry shader, the visuals rely on a fine rendering grid. This is a difficult task in that any CPU-GPU memory copies for large data sets must be avoided, and this is not the case with the current implementation. Also, for fine grids, where the number of primitives scales dramatically, marching cubes does not perform efficiently enough for real-time.

I would also want to look into interactivity with the fluid. As it is intended to show viability as game mechanic, interactivity with both the user and the surrounding environment is a necessity. Therefore research must be done in both user input affecting the simulation space, and the fluid's movement exerting forces on solid physics objects. While simulations involving physics objects' interacting with fluid have been implemented, to the best of my knowledge the task has not yet been carried out on the GPU.

6 Results and Conclusion

Using an NVidia GeForce GTX 560 Ti and an Intel Core i7 @ 3.40 GHz, all demonstrations ran at over 30 fps, with the largest example fluctuating between 30 and 60 fps, and all smaller examples running above 60 fps. Based on these results, I conclude that it is now feasible to use the fluid as a core game mechanic, despite the large amount of additional work required to make the fluid interactive. Given the way hardware has advanced over the past few years, new hardware may be available in the near future which can run an even more detailed simulation at faster rates.

References

[1] Foster and Metaxas, "Realistic Animation of Fluids." 1996.

[2] Chris Wyman, "An Approximate Image-Space Approach for Interactive Refraction." SIGGRAPH, 2005.

[3] Keenan Crane, Ignacio Llamas, Sarah Tariq, "Real-Time Simulation and Rendering of 3D Fluids." GPU Gems 3, 2009.

Appendix A: CUDA Overview

CUDA is a General Purpose GPU programming language with syntax almost identical to C. It easily integrates into build environments such as Visual Studio, and the CUDA compiler includes a C/C++ compiler so that programs can contain a combination of C and CUDA code.

A GPU consists of many individual processing units, all of which are much smaller than a processor one would find at the heart of a computer. When executing code on the GPU, the number of processors to be used is specified by the programmer, and all of these processors execute code at the same time. The processors used are typically abstracted into the term "blocks" or "warps."

So, in CUDA, a grid of blocks is launched, each of which will run the same computation, a kernel, which is the function written in CUDA C. Each block can also execute a certain number of threads, so each thread will run this computation, simultaneously.

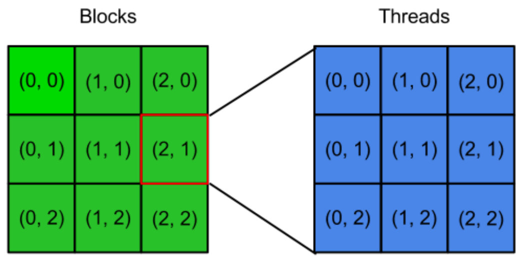


Figure 5: A visualization of a 2D grid of blocks on the GPU. Each block contains within it another 2D grid of threads. The indices of the blocks and threads may be used to construct an index into a global array.

What is most important is the indexing scheme used for the data. The blocks and threads have unique IDs from which an index can be constructed into a global memory array (see Figure 5). Hence, each thread should operate on one element of a global memory array. The mapping of threads to array locations is defined by the user, and could be anything, so long as all desired computations are made.

This style of computations allows for identical operations to be carried out simultaneously on independent elements of data. For example, computing the sum of two arrays, A and B, into an array C, each of which have size 800,000, can be accomplished conceptually in the same time as a single element sum, because each pair of elements in A and B will be summed simultaneously once delegated to different threads.

Finally, one must be cautious to avoid memory writes to the same location in global memory. As the threads execute simultaneously, it is not guaranteed which thread will make a successful write in the event that more than one thread tries to do so to the same memory location. It is highly unlikely that two threads which are competing to write will both write successfully.