Fluid simulations in OpenCL and OpenGL

The primary goal was to create a 3D fluid simulation accelerated by the GPU so it could be simulated and visualized in real time. Ultimately I failed to meet the goal of 3D due to driver limitations despite my rather current hardware. Instead I settled for a 2D simulation which still is fun too look at. Beneath the project itself, I sought to integrate OpenCL’s capabilities with my own set of C++ libraries to make harnessing the GPU easier and quicker for projects.

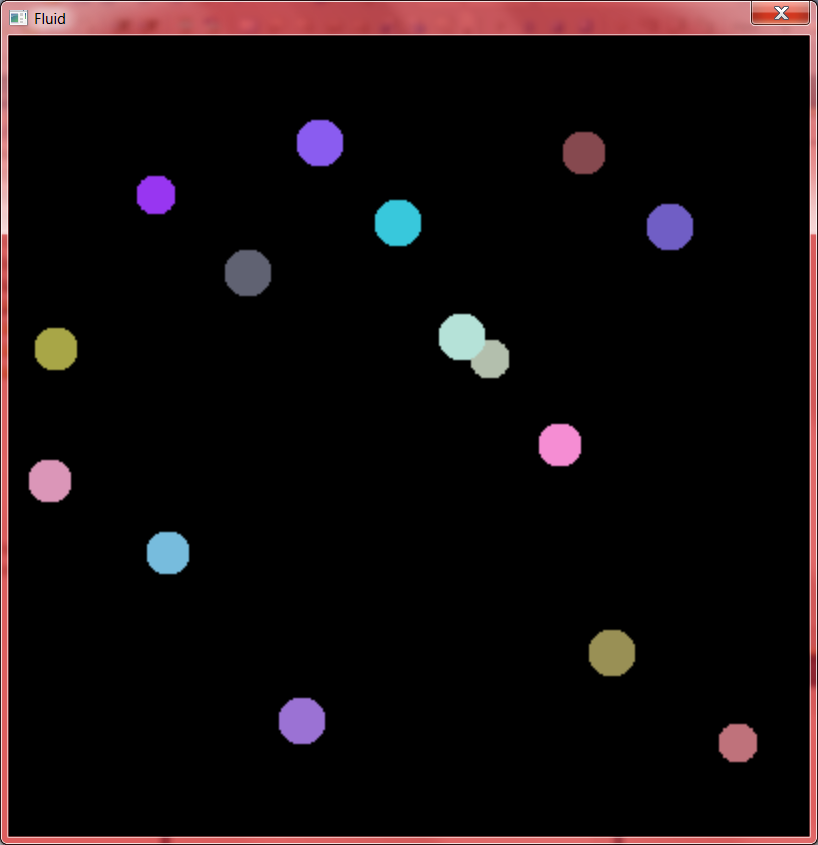
The fluid is represented by a 2D grid of vectors. The vectors represent the velocity of the fluid at points in space. The grid fits nicely with texture memory and makes the simulation well suited for the GPU. OpenCL has a limitation of not being able to read and write from the same texture. So in order to facilitate the simulation I double buffer the vectors. I have two vector textures “Velocity” and “VelocityPrev”, so when I need to modify the velocity field I swap the current velocities to “VelocityPrev” and simply write new values into “Velocity”. The velocity textures are formatted to have two components of type float and no filtering. I have not tried using half precision floating point components, but the simulation probably would not suffer from doing so. In addition to the velocity of the liquid, I have an ink layer which is a 2D grid of colors. The ink layer gives visualization to the velocities.

The simulation is broken into three steps: Diffusion, Advection, and Projection. Each of which can be performed almost entirely on the GPU. The only communication from host to device required is to invoke kernels and supply user input to the fluid. This makes worrying about latency a non-issue. Additionally since multiple steps involve solving a system of linear equations we can use the same kernel for multiple uses reducing our code size by a bit.

Diffusion or viscous diffusion is a step where the cells exchange contents with their neighbors. Diffusion is used on both the velocities and the ink. Thicker fluids like molasses exchange slowly and have a high viscosity, thinner fluids like water have lower viscosities and exchange quicker. Viscous diffusion is computed with the Laplacian operator . Which in finite difference form on a square grid is this.

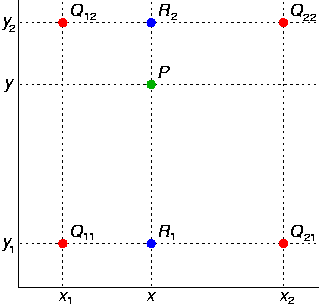
But because is a system of linear equations, writing it like that in the code will not result in a correct solution, in fact with higher viscosities the values will start to oscillate. Instead we use the Jacobi method to get an approximation. The Jacobi method involves using iteration to bring an approximation closer to the actual solution. The Jacobi equation looks like this for the 2D grid.

Where is the grid, is a grid, and and are scalar values. In the diffusion case we set to “Velocity”, to “VelocityPev”, to “VelocityPrev”, to . represents the distance between grid points, however in my simulation I simply use unit of distance between grid points. For I use but it’s an arbitrary magic number. is simply the time in between frames, which I measure in seconds. The unit of time you use for obviously effects the unit for since it’s a rate. Finally, we set to . I do this for 20 iterations, which seems to produce nice visuals while not bogging down the simulation. Increasing the iterations for diffusion doesn’t yield much more visually pleasing results; however in the second application of Jacobi more iteration produces a better effect on the visuals. Below is a picture of the ink layer before and after diffusion.

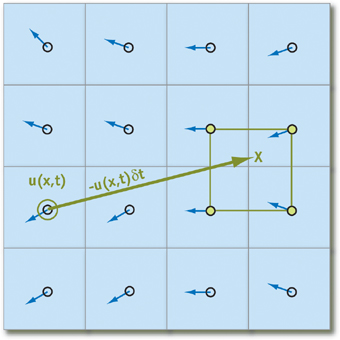


Before After

Advection is the movement of “stuff” through the liquid. The liquid and things in the liquid move along the velocities in the velocity grid. Because the velocity will mostly not line up with the discrete sampling of the grid we use bilinear interpolation to find a value at an arbitrary point in the grid.



Bilinear interpolation is used heavily in 3D graphics in order to display textures. As a result, the GPU has special support or the operation making it an especially speedy operation on texture memory. In order to avoid having to write to four grid locations by “pushing the fluid forward” we simply “pull the fluid forward.” If we were to try to move the contents of one cell forward with its velocity we’d have to update 4 cells in the grid with their new value. This would create race conditions between multiple threads trying to update cells. Instead we flip the velocity negative, use the bilinear interpolation to sample the point there and move it forward to its new location.



Below is a picture of the advection step moving the fluid forward after 1 second of simulation.



The final step is to fix the velocity grid. Liquids are for the most part incompressible, hence functioning hydraulics, after diffusion and advection the velocities don’t conserve the mass at each grid cells and thus the liquid is compressing. To fix this we run the projection step. To fix this we first calculate the divergence at each cell, which is the rate at which density is leaving a cell. We want to make the divergence 0 as that would mean the density of each cell has not changed and thus our fluid has not compressed. However, as mentioned before, after diffusion and advection divergence will not be 0. In finite difference form divergence is given as

With the Helmholtz-Hodge Decomposition Theorem we can conclude that a vector field () can be decomposed into the sum of a 0 divergence vector field () and the gradient of the pressure () in . As a result if we calculate the gradient of the pressure and subtract it from the vector field we get a vector field with 0 divergence, which is what we want.

To calculate the pressure we initialize the whole pressure field to 0, and use the Jacobi method with set to the Divergence field. set to and to . Now we have , as the earlier calculated velocity field, and we subtract the gradient of from and we get our mass conserving velocity field.

Below is a picture of the same 1 second of simulation but with the mass conserving velocities, compared to without the mass conserving velocities.



With Without

 As you can see, with the corrected velocities swirls begin to form and it looks like how a liquid behaves. As mentioned before, if we increase the number of iterations for the Jacobi method, we get a better visual result.

Part of my secondary objective for the semester project was to create a framework to make harnessing the GPU easier. I chose OpenCL for this over CUDA because it’s more hardware flexible and C++ compiler independent. However the OpenCL library leaves much to be desired in the API, it’s a C library so it doesn’t integrate well with C++’s RIAA scheme. My first task was to create C++ class wrappers around the basic objects of OpenCL. This was rather straight forward, the more interesting part came when it was time to send commands to the GPU.

Unlike CUDA the only way to interact with a device in OpenCL is through a Command Queue. Command queues hold actions like writing to device memory, reading from device memory, invoking kernels, etc… Commands in a queue can depend on other commands in the queue in a tree like fashion so events get executed in the correct order. So the name “queue” is somewhat misleading, just because events are pushed to the queue in a certain order doesn’t guarantee that that’s the order they’ll be executed in unless you set up the dependency tree correctly. Setting up a proper dependency tree in straight OpenCL quickly becomes frustrating. Most queue functions take a pointer to an array of events it’s dependent on. Like this:

clEnqueueCommand(...,&DependancyArray);

While this works on a basic API level, it quickly explodes the size of the code when doing anything large. As a solution I created the concept of an intermediate event. An intermediate event holds all the necessary information for queuing the command to the command queue, but also has methods to build the dependency tree automatically. You can establish an execution order by using the method “Then” of an intermediate event.

EventA.Then(EventB);

In addition there’s a overloaded method for encapsulating multiple events into a single event so EventC can depend on both EventA and EventB in this instance.

Do(EventA,EventB).Then(EventC);

This makes it easier to quickly build proper event dependencies without having to worry about array management. The next challenge was for kernel invocation. In CUDA a kernel call looks almost exactly like a normal function call, which I thought was a good premise. I wanted to translate that to my helper library. In normal OpenCL you have to set each argument of the kernel individually and then push an event to the command queue. So I needed to encapsulate this into something that looked like a normal function call and returned an intermediate event so I could use the aforementioned dependency management. This is where C++ really shines, because operators are so easily overridden and template meta-programming can generate custom classes based on types I was able to create a kernel template class that takes the function signature as the template argument and generates an appropriate function call operator. So to create a Kernel class that loads a kernel from a OpenCL C file, you simply do the following.

Kernel<void(int,float,float\*)> Foo(Program.GetKernel(“Foo”));

And then Foo can be used in the event queue.

EventA.Then(Foo(1,1.5,DeviceBuffer).Dim2(10,10));

Dim2 denotes the dimension of threads to call the kernel with, unless explicitly supplied OpenCL picks the block size. Having a generated function operator gives type safety to kernel calling and significantly reduces code in order to invoke one.

To allocate memory on the device you have to use a Buffer object in OpenCL. To make this easier I made another template class to make this easier. Simply supply a basic type or structure to the Buffer<> class.

class Foo;

{

float A;

int B;

};

Buffer<Foo> DeviceMemory (FooCount);

Buffer<float> MoreDeviceMemory (FloatCount);

Buffer<>’s have read and write messages that are intermediate events so they can easily be used. I also created an Image2D class which allows the allocation of texture memory, since texture memory has special formatting I couldn’t allow arbitrary data types to be used, instead there’s an ImageFormat enumeration of formats that are possible. The Image2D class also hooks in with my OpenGL wrapper (a heavy work in progress), which allows me to share textures between OpenGL and OpenCL. This is how I render the ink layer to the screen.

These helper classes are the backbone of the simulation, the make the simulation code a whole lot easier to read and error free. When built in Debug mode all OpenCL calls are checked for errors and report a line number of the origin of the error. All host and device memory used is cleaned up automatically through the destructors on the classes and helper functions make many line operations in OpenCL just a single call.

Further ways to improve the simulation would be to account for more variables of the fluid such as temperature, add the ability to place obstacles for the fluid, extending to the third dimension, and using a faster linear equation solver.