3D Fluid Simulation and Rendering

A project for the Real-Time Graphics Programming course

By Gianluca Iacchini

Introduction

This document describes the fluid simulation project made for the UNIMI course of Real-Time Graphics Programming.
The simulation is based on the following resources:

- Real-Time Simulation and Rendering of 3D Fluids
  https://www.cs.cmu.edu/~kmcrane/Projects/GPUFluid/paper.pd
- Fast Fluid Dynamics Simulation on the GPU
  https://developer.nvidia.com/gpugems/gpugems/part-vi-beyond-triangles/chapter-38-fast-fluid-dynamics-simulation-gpu
- WebGL-Fluid-Simulation
  https://github.com/PavelDoGreat/WebGL-Fluid-Simulation
- Fluid-sim by bhanathan
  https://github.com/bhatnathan/fluid-sim

While the project structure and utility classes are for the most part based on:

- Hazel Engine
  https://github.com/TheCherno/Hazel
- Learn OpenGL
  https://learnopengl.com/

Simulation

There are two main ways to approach the simulation of fluids: the Lagrangian viewpoint and the Eulerian viewpoint.
The Lagrangian approach uses particles to track the fluid motion. Each point of the fluid can be thought as a separate particle with its own properties such as velocity, temperature or dye color.
The Eulerian approach looks at fixed points in space and keeps track of how the fluid quantities change, this is usually done utilizing a grid.

The approach that was used in the project is the Eulerian viewpoint, that is due to the fact that GPUs are well suited to work with grids. In this way we can also take advantage of GPU high performance through parallelism, as well as use textures to store our data.

Navier-Stokes Equations

A fluid flow can be described using the incompressible Navier-Stokes equations, a set of partial differential equations (PDEs) usually written as:

$$\frac{\delta \vec{u}}{\delta u} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla \cdot \nabla \vec{u}$$
$$\nabla \cdot \vec{u} = 0$$

They can be rewritten as

$$\frac{\delta \vec{u}}{\delta u} = -(\vec{u} \cdot \nabla \vec{u}) - \frac{1}{\rho} \nabla p + \nu \nabla^2 \vec{u} + F$$
$$\nabla \cdot \vec{u} = 0$$

To better understand the equation we will be splitting it into multiple pieces.
The first part $-(\vec{u} \cdot \nabla \vec{u})$ is called advection. Advection is when the velocity of the fluids causes it to transport quantities along with the flow, such as temperature or some dye moving into the fluid. Velocity itself can also be advected like any other quantity, in this case we talk about self-advection.

The second part $-\frac{1}{\rho} \nabla p$ takes pressure into account. When a force is applied to the fluid it does not instantly propagate through the entire volume, it instead forces the molecules to push into each other.

The final part $\nu \nabla^2 \vec{u} + F$ is about external forces (such as gravity). They can be applied to a specific region of the fluid or they can be applied to the entire volume.

Note that the *nabla* operator (∇) is used for gradient (∇), divergence ($\nabla \cdot$) and Laplcian (∇^2) operations, all of which use derivatives. Because we are working in a discrete space, whenever a derivative function is needed, we will be using its finite-difference version instead.

Algorithm

The algorithm for the basic simulation of a fluid is a series of steps repeated each frame

```
void update()
{
    u = advect(u);
    // Advect other quantities here

    u = addForces(u); // Vorticity, Gravity, Impulse, etc.

    p = computePressure(u);
    u = subtractPressureGradient(u, p);
}
```

Each step updates the velocity, which in turn will advect the other quantities (and itself) along the flow.

We will now jump into the implementation of the algorithm, note that in this document we will not talk about the theory behind the simulation, since it is beyond the scope of this project; however we may touch on it briefly when talking about its implementation or the technical choices made.

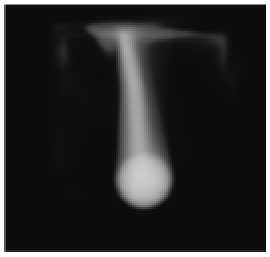
Simulation Implementation

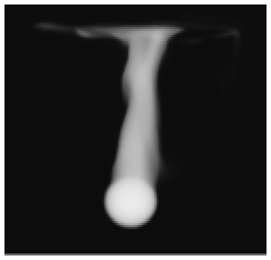
Textures

Because we are working on a 3D simulation, we need to store data in a three-dimensional grid. In a CPU application this would’ve been done by using an array, however since we are working with the GPU we will be using 3D textures to store the necessary information.

To correctly simulate the fluid we need at least two textures: one for velocity and one for pressure. In reality we will be using many more than that, some of the other textures that we will be using are a dye texture (like the name suggests, dye is a color being carried around the fluid; without it we wouldn’t be able to visualize the simulation), a temperature texture (used to compute external forces such as buoyancy) and other textures used to simplify computation in intermediate steps (e.g. a velocity divergence texture so we don’t have to compute it every time).

Now that we have chosen the data structure we want to use, we have to ask ourselves how big the simulation grid (and therefore the textures used to store its data) should be. A bigger grid will produce more accurate results however it will be more resource intensive.


A fluid simulation using a 64 cell grid


A fluid simulation using a 128 cell grid

For our simulation we chose a cubic grid with a side of size 128, however nothing stops us from choosing a different size or even a rectangular box shape.

Ping-pong textures

In almost all of the steps of our simulation we will need to update a texture data based on its previous contents. In a CPU application this would not pose a problem, since we could simply update a variable with a regular assignment like this

```
A[i,j,k] = A[i,j,k] + someFunction(arg1, arg2, ...);
```

However this is not possible with textures, as reading from and writing to the same texture in a single render pass would result in undefined behavior.
There is a simple solution to this problem: ping-ponging textures. With ping-ponging we create two textures for each information that we want to store, a read texture and a write texture. Whenever we need to read data from a texture we will read it from the read texture and whenever we need to perform a write operation we will perform it on the write texture; then after each write we swap the two textures.
In a CPU application it would be similar to using an extra variable

```
writeA[i,j,k] = readA[i,j,k] + someFunction(arg1, arg2, ...);
swap(readA, writeA);
```

It doesn’t matter how many times we perform a read and a write, we only ever need two textures.

In our project this is done by creating two framebuffers, each of which has a color texture attached to it. Having two different framebuffers instead of just two textures helps to keep the code more tidy and is also faster than switching color attachments.

Shaders

All the steps presented in the algorithm section must be run by the GPU, therefore they will be implemented in program shaders. Before the simulation starts we initialize a screen-size quad with the following vertices

```
float quadVertices[] = { // vertex attributes for a quad that fills the
entire screen.
    //positions   //tex coords //grid coords in range [0-GridSize]
   -1.0f, -1.0f,  0.0f, 0.0f,  0.f, 0.f,
    1.0f, -1.0f,  1.0f, 0.0f,  gridDimension.x, 0.f,
   -1.0f,  1.0f,  0.0f, 1.0f,  0.0f, gridDimension.y,
    1.0f,  1.0f,  1.0f, 1.0f,  gridDimension.x, gridDimension.y,
};
```

Whenever we need to write to one of the simulation textures, we do so using instancing. The number of instances is equal to the depth of the grid.

Let's now see some of the shaders used for the simulation.

`basicFluid.vert`: a simple vertex shader used by all the simulation steps, it takes as input the vertices of the quad, it then sets the vertex positions to fill the whole screen and passes the texture coordinates and grid coordinates to the geometry shader along with the **`gl_instanceID`** number; which will be used to compute the depth for both texture and grid coordinates.

`layeredRender.geom`: a simple geometry shader used by all the simulation steps, it takes triangles as input and returns triangle strips as output. Before returning each vertex it sets their texture coordinates depth and grid coordinates depth using the instance number obtained by the vertex shader.

`advection.frag`: performs the advection steps. As we introduced in the previous section, advection is the process by which the fluid velocity transports itself (self-advection) and other quantities through the fluid. Imagine we have a particle at each grid point location, we want to know where it will end up after a certain amount of time δt. We multiply the particle velocity by the time δt to find out where this imaginary particle ends up, then we update the new grid point location with the particle starting position quantity.
There are, however, two problems when implementing this approach. The first is that when given a δt big enough the simulation may become numerically unstable, the second is that due to how the GPU works, we can only update the current fragment we are working on. Therefore we will invert the problem: we trace back where the imaginary particle started and update our current quantity with the particle starting grid cell quantity; this is in line with how the GPU works and it can be demonstrated that updating the quantity this way will not “blow up” our simulation, no matter how big δt is.

`jacobi.frag`: In order to solve the pressure equation we will be using an iterative solution called Jacobi iteration. We run the **`jacobi.frag`** shader multiple times each frame, giving pressure obtained in the last iteration as an input for the next. With more iteration the results will be more accurate but will also be more expensive to calculate, for our simulation we are using

100 as the default number of iterations.

`gradientSubtraction.frag:` This is the last step of the simulation, this shader updates the velocity to account for the newly computed pressure.

We won't go any further into the individual simulation shaders, as they are simply a translation of the Navier-Stokes equation into GLSL code.

In the client code we wrap each shader with a method. The methods are responsible for binding the correct shaders, framebuffers and texture units.

```
void fluidStep(PingPongFramebuffer& fluidData){
    glBindVertexArray(quadVAO);
    glViewport(0,0, fluidData.textureSizeX, fluidData.textureSizeY);
    glBindFramebuffer(GL_FRAMEBUFFER, fluidData.writeBuffer);

    currentStepShader.Use();
    currentStepShader.SetInt(“textureRead”, 0);
    glBindTexture(GL_TEXTURE_2D, fluidData.readTexture);

    glDrawElementsInstanced(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0,
fluidData.tetureSize.z);
};
```

Rendering

The result of the simulation is a collection of data stored in multiple 3D textures, however we have no way to view it yet.
To visualize the fluid we will use a ray-marching fragment shader to march through the dye texture.

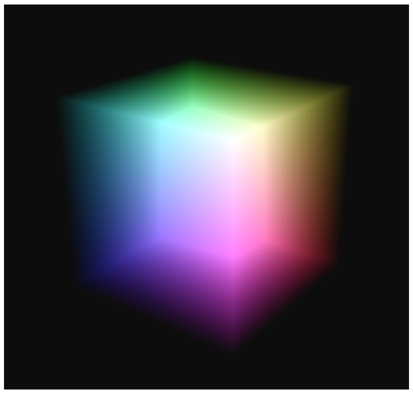
Raycasting

In order to perform ray-marching, we have to know where the ray is entering the volume, in which direction it is marching and how many samples to take.
We start by creating a volume box; this box will represent the fluid volume in our world, then we create a screen-size texture called *rayData*, we want to encode, for each pixel of this texture, the entry point of the ray (in texture space) and the depth through the volume that the ray traverses.
To get this information we draw the back faces of our volume into the *rayData* texture with a shader that outputs the distance from the eye to the fragment in the alpha channel. We then run a similar shader for the front faces, but we also return the texture-space coordinate of the

fragment in the output RGB channels. Furthermore, before drawing the front faces, we enable subtractive blending in order to satisfy the following equations:

```
rayData.rgb = FrontFaces.rgb
rayData.a = BackFaces.a - FrontFaces.a
```

In this way the *rayData* texture will contain all the information we want; we are only missing the ray direction, which can be computed by subtracting the ray entry point from the eye position (in texture space).


Representation of ray data texture

Raymarching

We can now proceed with the ray-marching algorithm. We render the volume by drawing a screen-size quad; we then look into the *rayData* texture to find out which pixels we need to ray-cast through and we compute the ray direction using the ray entry point and the eye position.
We march through the ray with a step size of half a voxel; at each step along the way we sample values from the dye texture and then we blend them front to back, in this way we can terminate ray casting early when the color saturates (`FinalColor.a > 0.99f`).
The number of samples that the ray-marching algorithm uses is proportional to the depth through volume that the ray traverses.

Compositing

We now have a fully working and visible fluid simulation, there are however two main problems to solve. The first is that the rays march through the volume even if they encounter other scene objects on the way. This means that the fluid will be visible even when it is covered by other opaque scene geometry.
The second problem appears evident when the camera enters the volume box. Because we are computing the ray direction as the distance between the eye position and the front faces; if the camera is clipping the volume then the front faces will not be rendered, thus we will not be able to compute the ray direction.

Occlusion culling

There is an easy solution to scene geometry blocking the fluid volume box. We compare the distance between the eye and the front faces fragments to the distance between the eye and the closest scene object.
If the distance between the eye and the front faces is greater, then that fragment is occluded; we mark this fragment by setting the R channel to a negative value; whenever we find a marked fragment in the ray-marching shader we don't cast any rays for that fragment.

Clipping

Dealing with the camera clipping with the volume box is a bit more complicated. We start by marking the fragments for which the front faces were not rendered by outputting a negative value in the G channel. We then convert the camera near plane position to the volume box texture space; we can do this by using the inverse of the volume box MVP matrix.
The ray direction can now be computed for the marked fragments using the eye position and the near plane position; we also have to correct the depth through the volume that the ray traverses by subtracting from it the distance between the eye and the near plane.

Rendering Implementation

Let's dive a bit into the implementation of the fluid rendering. We start by binding a framebuffer in the main render loop before the scene is drawn. This framebuffer has a depth buffer component attached to it, which we will use in order to implement occlusion culling for the fluid.

In the fluid render function we begin by filling the *rayData* texture, to do so we draw a cube; each vertex of the cube is composed only by its position.
We render the back faces of the cube first and we draw them into the *rayData* texture, we then enable blending with the following parameters to achieve subtractive blending

```
glEnable(GL_BLEND);
glBlendEquation(GL_FUNC_REVERSE_SUBTRACT);
glBlendFuncSeparate(GL_ONE, GL_ZERO, GL_ONE, GL_ONE);
```

After that we render the front faces of the cube.

The raycasting shader is quite simple; the vertex shader is a basic program that simply multiplies the cube vertices by its MVP matrix.
The fragment shader returns the cube texture coordinates in the RGB channels, and the distance between the eye and the current fragment in the alpha channel. This is also where we check for occlusion and clipping

```
void main()
{
      float depthValue = texture(depthTexture, gl_FragCoord.xy /
screenSize).x;
      float linearDepth = LinearizeDepth(depthValue);

      // We set the back face to a negative green value. If the front face
is clipped (e.g. camera is inside the simulation box)
      // Then the negative value will remain; otherwise the RGB channels
will be overwritten with the front faces values;
      if (isBackFace)
      {
            FragColor.xyz = vec3(0.f, -1.0f, 0.0f);
            // Returns eye to fragment distance in the alpha channel
            FragColor.w = rayData.depth;
      }
      else
      {
            // This condition is used to check if an object is between the
camera and the fluid, if that's the case than a negative value will be
written to the texture red channel (we are outputting a positive value
            // because we are using subtractive blending).
            if (linearDepth < rayData.depth)
            {
                  FragColor = vec4(1, 0, 0, 0);
                  return;
            }

            // Returns the tex coords position, adjusted to account for
subtractive blending
            FragColor.xyz = -rayData.posInGrid;
            // Returns eye to fragment distance in the alpha channel
            FragColor.w = rayData.depth;
      }
}
```

Now that we have our *rayData* texture ready, we can proceed with the ray-marching shader. This is done by drawing a screen-size quad, once again the vertices of the quad contain only their position in NDC.
The vertex shader is quite short, it sets the `gl_Position` with the vertices data and outputs to the fragment shader the position of the near plane in the volume box object space; this is achieved by multiplying the vertex position by the inverse of the volume box MVP matrix; as explained in the clipping section, this is done in order to compute the ray direction when the camera is inside the volume box.

```
void main()
{
	// Basic position for a full screen quad
	rayCastData.pos = vec4(aPos, 0.0f, 1.f);
	// Computes the near plane position to the volume box object space
	rayCastData.posInGrid = (invModelViewProj * vec4(aPos.xy * zNear, 0,
zNear)).xyz;

	gl_Position = rayCastData.pos;
}
```

The fragment shader is where we have the actual ray-marching algorithm. It starts by checking whether the current fragment is occluded by scene geometry or if the camera is clipping the volume box

```
vec4 outColor = vec4(0);
vec4 rayData = texture(rayDataTexture, gl_FragCoord.xy / RTSize);

// Check if fragment is occluded by scene geometry
if (rayData.x < 0)
{
	// if true then don’t cast any rays for this fragment
	FragColor = outColor;
	return;
}

// Check if camera is clipping the volume box
if (rayData.y < 0)
{
	// Sets the starting point of the ray as the corresponding fragment
of the near plane
	rayData.xyz = (rayCastData.posInGrid / gl_FragCoord.z);
	// Corrects the depth through the volume
	rayData.w = rayData.w - zNear;
}

vec3 rayOrigin = rayData.xyz;
float rayLength = rayData.w;
```

The core of the algorithm is given by the for loop

```
for (i; i < nSamples; i++)
{
      DoSample(1, O, outColor);
      O += stepVec;

      if (outColor.a > 0.99f)
            break;
}
```

The number of samples is proportional to the depth through the volume that the ray traverses and the grid dimension

```
float fSamples = (rayLength / gridScaleFactor * maxGridDim) * 2.0f;
int nSamples = int(floor(fSamples));
```

A single step is proportional to half a voxel of the grid.

```
vec3 stepVec = normalize((rayOrigin - eyeOnGrid) * gridDim) * recGridDim *
0.5f;
```

The **`DoSample(1, O, outColor)`** function is where we do the blending and it is defined as follows

```
// Sample dye texture at the specified position.
void DoSample(float weight, vec3 O, inout vec4 color)
{
      vec3 texcoords;
      vec4 sampleColor;
      float t;

      // Sample dye texture
      texcoords = vec3(O.x, O.y, O.z);
      sampleColor = weight * texture(dyeTexture, texcoords);
      sampleColor.a = (sampleColor.a) * 0.1f; //Opacity modulator

      // Blend color
      t = sampleColor.a * (1.0f - color.a);
      color.rgb += sampleColor.rgb + t;
      color.a += t;
}
```

This function is not only called once per loop, but also at the end of the shader and only if the loop did not end early.

```
if (i == nSamples)
{
      DoSample(fract(fSamples), O, outColor);
}
```

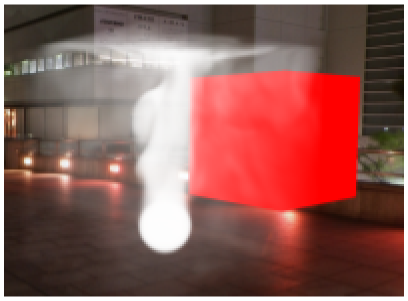
This last function call is used to reduce any banding artifacts that may appear. Another trick to minimize the banding is to use a jitter texture, in our project we use a 256x256 texture to jitter the samples along the ray direction.

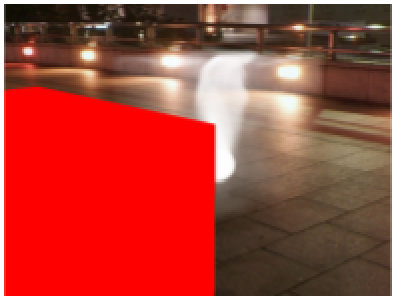
```
float offset = texture(jitterTexture, gl_FragCoord.xy / 256.f).x;
vec3 O = rayOrigin + stepVec * offset;
```

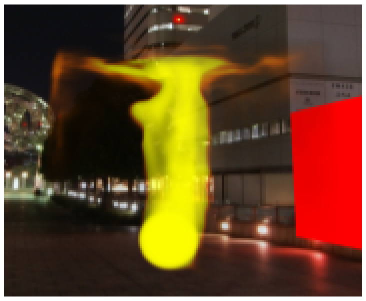
The fragment shader will return the final color of the simulation, we then blend it to screen to using alpha blending

```
glEnable(GL_BLEND);
glBlendEquationSeparate(GL_FUNC_ADD, GL_FUNC_ADD);
glBlendFuncSeparate(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_ONE, GL_ZERO);
```

We can now visualize the fluid simulation


Smoke in front of scene geometry


Smoke occluded by scene geometry


Simulation of a flame

Performance

The performance of the simulation depends on multiple factors. We measure performance by looking at the average frame time and the total memory used.

Memory usage

Since we are working with a lot of textures, we have to consider how much memory we are using. Texture sizes are based on either the simulation grid dimension or the application window size.
We use a total of 15 textures for the physic simulation, each of them is a 3D texture with a size of **`(GridDimension x GridDimension x GridDimension)`**

```
phi_hat = Framebuffer::CreateDoubleFramebuffer(gridDimension, 4);
phi_hat_1 = Framebuffer::CreateDoubleFramebuffer(gridDimension, 4);
velocity = Framebuffer::CreateDoubleFramebuffer(gridDimension, 3);
divergence = new Framebuffer(gridDimension, 1);
curl = new Framebuffer(gridDimension, 3);
dye = Framebuffer::CreateDoubleFramebuffer(gridDimension, 4);
temperature = Framebuffer::CreateDoubleFramebuffer(gridDimension, 1);
pressure = Framebuffer::CreateDoubleFramebuffer(gridDimension, 1);
obstacleBounds = new Framebuffer(gridDimension, 1);
```

As discussed in the texture section, we need texture ping-ponging whenever we need to read and write to the same texture in a single render pass; therefore each framebuffer initialized with the **`Framebuffer::CreateDoubleFramebuffer(gridDimension, nChannels)`** method stores two textures instead of just one.
All the textures in the project are created with a size of 32 bit per channel; there are a combined total of 17 channels between the ping-pong textures, and a combined 5 channels for the single ones. The total memory used by the physic simulation with a grid of side 128 is then

```
(17 * 2 + 5) * (128 * 128 * 128) * (32 bit) = 2,617,245,696 bit
```

Which translates to about 327MB.

We can greatly reduce the memory needed by reducing the grid dimension. This will drastically improve both the memory usage and the frame time (more on this in the next section) at the cost of the simulation accuracy; likewise we can also reduce the size of the texture channels.

For example, a simulation with a grid of size 64 and 32 bit textures would only need 40.9MB of memory, a grid of size 128 with 16 bit textures would require 163MB while the memory sufficient to simulate a fluid with a 64 cell grid and 16 bit textures is only 20.4MB.

On top of these we also have the textures needed for the ray-marching algorithm. These texture sizes depend on the application window size.

```
rayData = new Framebuffer(screenSize, 4);
```

```
depthBuffer = new Framebuffer(screenSize, 4);
```

Note that the depthBuffer actually contains two textures: a 4 channel color texture and a 1 channel depth buffer texture.
The total memory used by these textures for a window of size 1920x1080 is ~74.6MB. Using 16 bit textures it can be reduced to 37.3MB.

Frame time

The application average frame time depends on three factors: simulation grid size, application window size and the number of Jacobi iterations.

The average frame time using a grid of side 128 with 100 Jacobi iterations on a 1920x1080 application is ~13.33ms

Reducing the grid dimension is by far the best way to improve the frame time (and also the memory usage as we have seen in the previous section); reducing the grid to a size of 64 results in an average frame time of ~4.20ms.

A small improvement can be obtained by reducing the application window size, this is due to the ray-marching shader having to cast less rays. Reducing the screen size to 800x600 will improve the frame time to an average of 10.50ms.

Finally let’s look at the number of Jacobi iterations, if we halve them to 50 we get an average frame time of 9.50ms. The simulation accuracy lost in this way is not too big, 50 iterations per frame is still good enough for a generic fluid; however this is not the case for liquids. If we were to use too few iterations for liquids simulation the results would be disastrous.