# Fluid Simulation in Unity

Physics Coursework 2

By Adam Joyce

## Introduction

This project acted as an introduction to fluid dynamics. It consists of a simple two dimensional fluid dynamics demonstration in Unity. The project largely follows the methods outlined in the GPU Gems article ‘Chapter 38. Fast Fluid Dynamics Simulation of the GPU’ (1) as well as Philip Rideout’s implementation (3) in C and GLSL.

What follows is an overview of the project’s files and high level look at the code.

## Project Files

The code it split up into a number of files. ‘FluidSimulation.cs’ is the only C# file and handles the main execution of the simulation for each frame. The other code files are custom Unity .shader files each containing vertex and fragment program shaders written in CG.

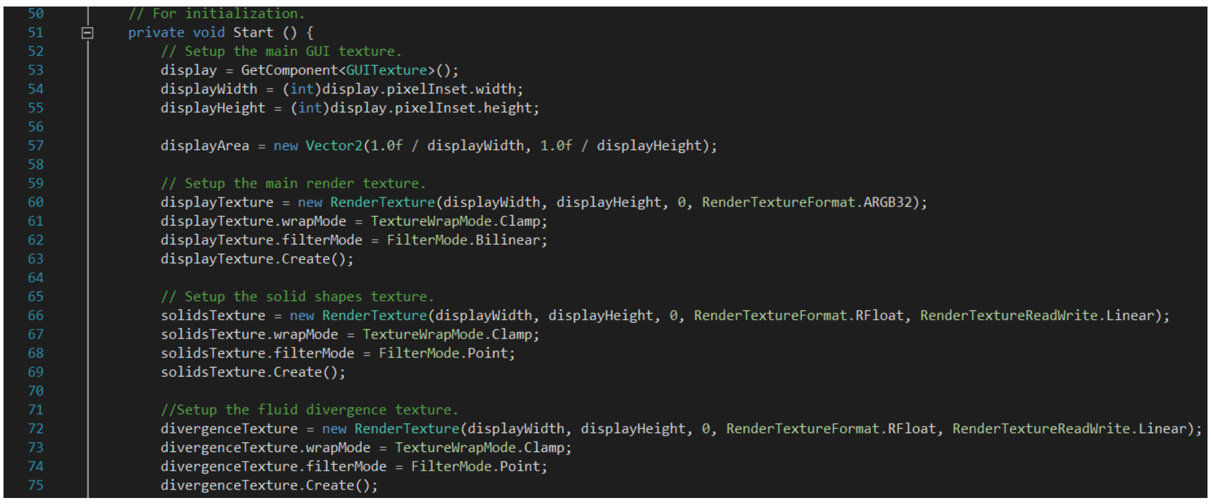
I will outline the purpose of the main functions in ‘FluidSimulation.cs’ covering the appropriate shader programs when necessary.

## Start()

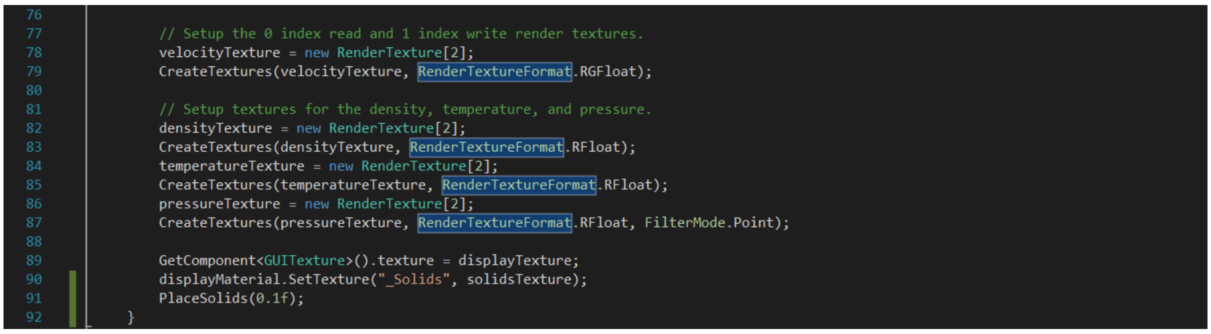
The start function is used to setup all the textures and components necessary for the simulation. I use a number of different textures as arrays to store values for certain properties of the simulation, for example the solids and divergence textures are used to track the location of solid obstacles and the divergence value of the fluid at a given location respectively.

A number of textures such as the velocity and density textures are actually texture arrays. This is because certain operations cannot be performed in place, but instead require some temporary storage to be utilised. These arrays therefore are initialised to consist of two textures, one for reading and the other writing. The ‘SwapTextures()’ function is used to swap around two textures in an array when necessary.

After the display area and textures are initliased the main display texture is assigned to its parent’s object’s texture. The display material (which houses my ‘Display.shader’) has its texture set to the solids texture and the function ‘PlaceSolids()’ is called which draws the solids to the solids texture. This leaves us in a position to start implementing the fluid simulation itself.



```
    // For initialization.
    private void Start () {
        // Setup the main GUI texture.
        display = GetComponent<GUITexture>();
        displayWidth = (int)display.pixelInset.width;
        displayHeight = (int)display.pixelInset.height;

        displayArea = new Vector2(1.0f / displayWidth, 1.0f / displayHeight);

        // Setup the main render texture.
        displayTexture = new RenderTexture(displayWidth, displayHeight, 0, RenderTextureFormat.ARGB32);
        displayTexture.wrapMode = TextureWrapMode.Clamp;
        displayTexture.filterMode = FilterMode.Bilinear;
        displayTexture.Create();

        // Setup the solid shapes texture.
        solidsTexture = new RenderTexture(displayWidth, displayHeight, 0, RenderTextureFormat.RFloat, RenderTextureReadWrite.Linear);
        solidsTexture.wrapMode = TextureWrapMode.Clamp;
        solidsTexture.filterMode = FilterMode.Point;
        solidsTexture.Create();

        //Setup the fluid divergence texture.
        divergenceTexture = new RenderTexture(displayWidth, displayHeight, 0, RenderTextureFormat.RFloat, RenderTextureReadWrite.Linear);
        divergenceTexture.wrapMode = TextureWrapMode.Clamp;
        divergenceTexture.filterMode = FilterMode.Point;
        divergenceTexture.Create();
```



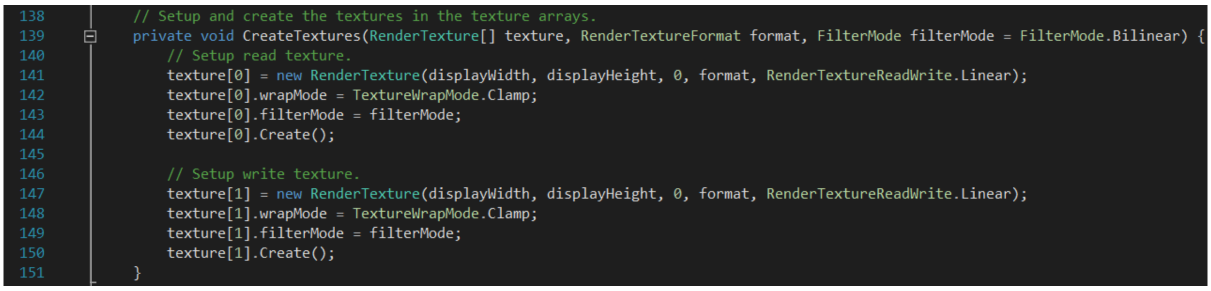
```
        // Setup the 0 index read and 1 index write render textures.
        velocityTexture = new RenderTexture[2];
        CreateTextures(velocityTexture, RenderTextureFormat.RGFloat);

        // Setup textures for the density, temperature, and pressure.
        densityTexture = new RenderTexture[2];
        CreateTextures(densityTexture, RenderTextureFormat.RFloat);
        temperatureTexture = new RenderTexture[2];
        CreateTextures(temperatureTexture, RenderTextureFormat.RFloat);
        pressureTexture = new RenderTexture[2];
        CreateTextures(pressureTexture, RenderTextureFormat.RFloat, FilterMode.Point);

        GetComponent<GUITexture>().texture = displayTexture;
        displayMaterial.SetTexture("_Solids", solidsTexture);
        PlaceSolids(0.1f);
    }
```

## Display.shader

This is a simple shader that determines the overall colour of the objects in the scene.  _MainTex acts as the density value and _Solids as the value from the solids texture.

## CreateTextures()

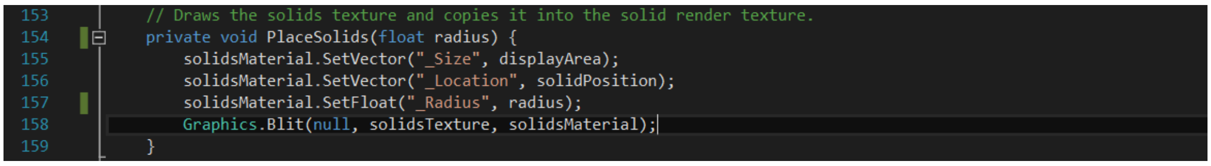
This function simply initalises the two render textures in each of the previously mentioned texture arrays.



```
    // Setup and create the textures in the texture arrays.
    private void CreateTextures(RenderTexture[] texture, RenderTextureFormat format, FilterMode filterMode = FilterMode.Bilinear) {
        // Setup read texture.
        texture[0] = new RenderTexture(displayWidth, displayHeight, 0, format, RenderTextureReadWrite.Linear);
        texture[0].wrapMode = TextureWrapMode.Clamp;
        texture[0].filterMode = filterMode;
        texture[0].Create();

        // Setup write texture.
        texture[1] = new RenderTexture(displayWidth, displayHeight, 0, format, RenderTextureReadWrite.Linear);
        texture[1].wrapMode = TextureWrapMode.Clamp;
        texture[1].filterMode = filterMode;
        texture[1].Create();
    }
```

## PlaceSolids()

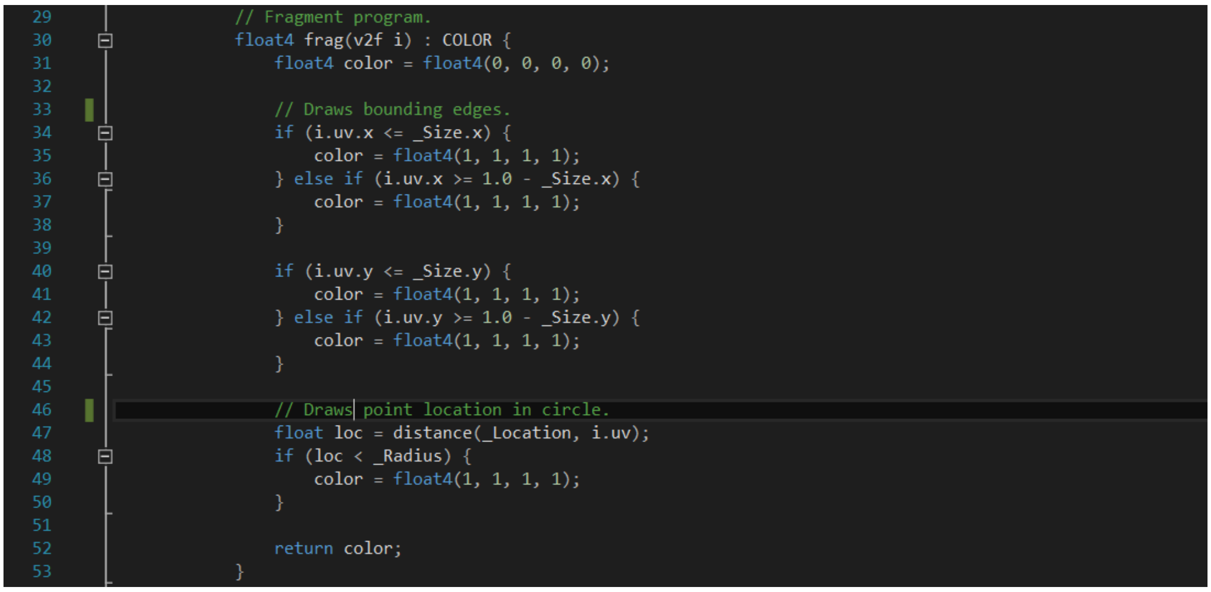
'PlaceSolids()' sets the Solids.shader variables and draws the solids into the solids render texture.



```
    // Draws the solids texture and copies it into the solid render texture.
    private void PlaceSolids(float radius) {
        solidsMaterial.SetVector("_Size", displayArea);
        solidsMaterial.SetVector("_Location", solidPosition);
        solidsMaterial.SetFloat("_Radius", radius);
        Graphics.Blit(null, solidsTexture, solidsMaterial);
    }
```

## Solids.shader

The solids shader determines which pixels are not considered 'solid'.  It checks to see if the pixel it is currently working on is in the bounds of the display area.  It also checks to see if the pixel falls within the radius of the solid circle.  If either of these conditions are found to be true the pixel is considered 'solid' and will be coloured white.



```
// Fragment program.
float4 frag(v2f i) : COLOR {
    float4 color = float4(0, 0, 0, 0);

    // Draws bounding edges.
    if (i.uv.x <= _Size.x) {
        color = float4(1, 1, 1, 1);
    } else if (i.uv.x >= 1.0 - _Size.x) {
        color = float4(1, 1, 1, 1);
    }

    if (i.uv.y <= _Size.y) {
        color = float4(1, 1, 1, 1);
    } else if (i.uv.y >= 1.0 - _Size.y) {
        color = float4(1, 1, 1, 1);
    }

    // Draws point location in circle.
    float loc = distance(_Location, i.uv);
    if (loc < _Radius) {
        color = float4(1, 1, 1, 1);
    }

    return color;
}
```

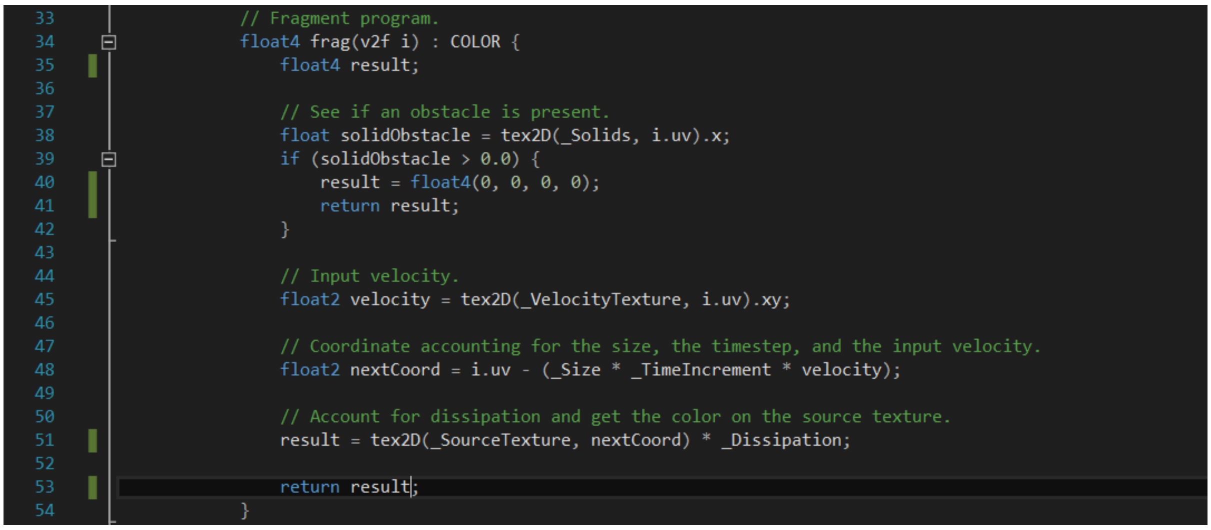
## Advect(), AddBuoyancy(), AddImpulse(), CalculateDivergence(), IterateJacobi(), SubtractGradient()

Similarly to 'PlaceSolids(), the above functions are used to set their corresponding shader variables and then draw the results into their appropriate render textures.

## Update()

This is where the main execution loop occurs. We begin by advecting the fluid density and temperature against the fluid velocity. This simulates these properties 'flowing' along with the fluid. It is also important to then advect the fluid's velocity against itself. Without this self-advection the fluid continually expands from the impulse origin with no change to its velocity creating an unnatural effect.

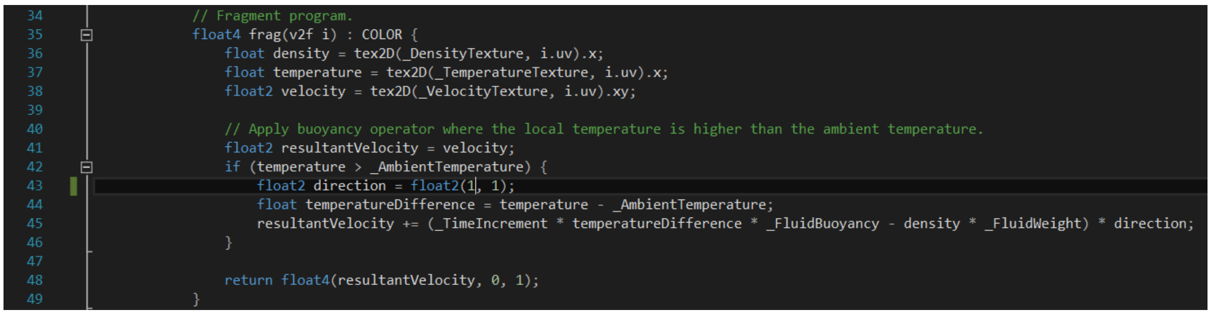
**Advection shader code:**



```
// Fragment program.
float4 frag(v2f i) : COLOR {
    float4 result;

    // See if an obstacle is present.
    float solidObstacle = tex2D(_Solids, i.uv).x;
    if (solidObstacle > 0.0) {
        result = float4(0, 0, 0, 0);
        return result;
    }

    // Input velocity.
    float2 velocity = tex2D(_VelocityTexture, i.uv).xy;

    // Coordinate accounting for the size, the timestep, and the input velocity.
    float2 nextCoord = i.uv - (_Size * _TimeIncrement * velocity);

    // Account for dissipation and get the color on the source texture.
    result = tex2D(_SourceTexture, nextCoord) * _Dissipation;

    return result;
}
```

It is important to include some kind of buoyancy in the simulation in order to simulate convection currents in the fluid. We do this by applying a buoyancy operator anywhere the local temperature is higher than the ambient temperature.

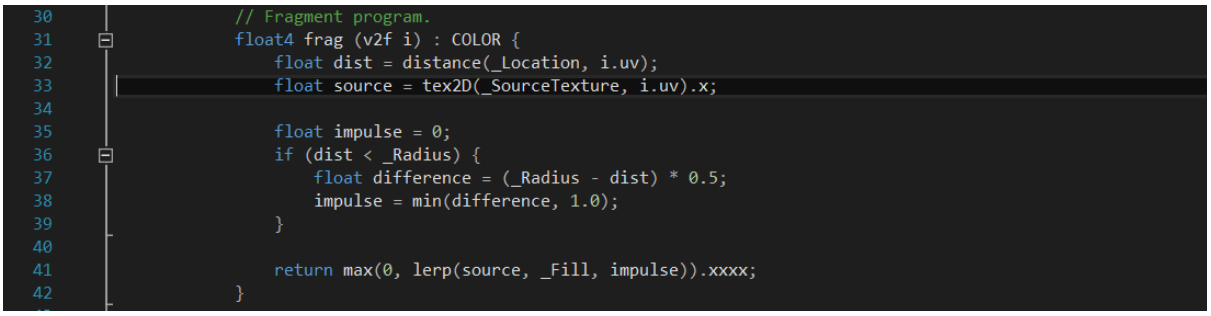
**Buoyancy shader code:**



```
        // Fragment program.
        float4 frag(v2f i) : COLOR {
            float density = tex2D(_DensityTexture, i.uv).x;
            float temperature = tex2D(_TemperatureTexture, i.uv).x;
            float2 velocity = tex2D(_VelocityTexture, i.uv).xy;

            // Apply buoyancy operator where the local temperature is higher than the ambient temperature.
            float2 resultantVelocity = velocity;
            if (temperature > _AmbientTemperature) {
                float2 direction = float2(1, 1);
                float temperatureDifference = temperature - _AmbientTemperature;
                resultantVelocity += (_TimeIncrement * temperatureDifference * _FluidBuoyancy - density * _FluidWeight) * direction;
            }

            return float4(resultantVelocity, 0, 1);
        }
```

After this, we need to add an initial impulse point. The cause of this impulse is an ‘external force’ and as such can be incorporated at a particular location using a simple impulse shader.

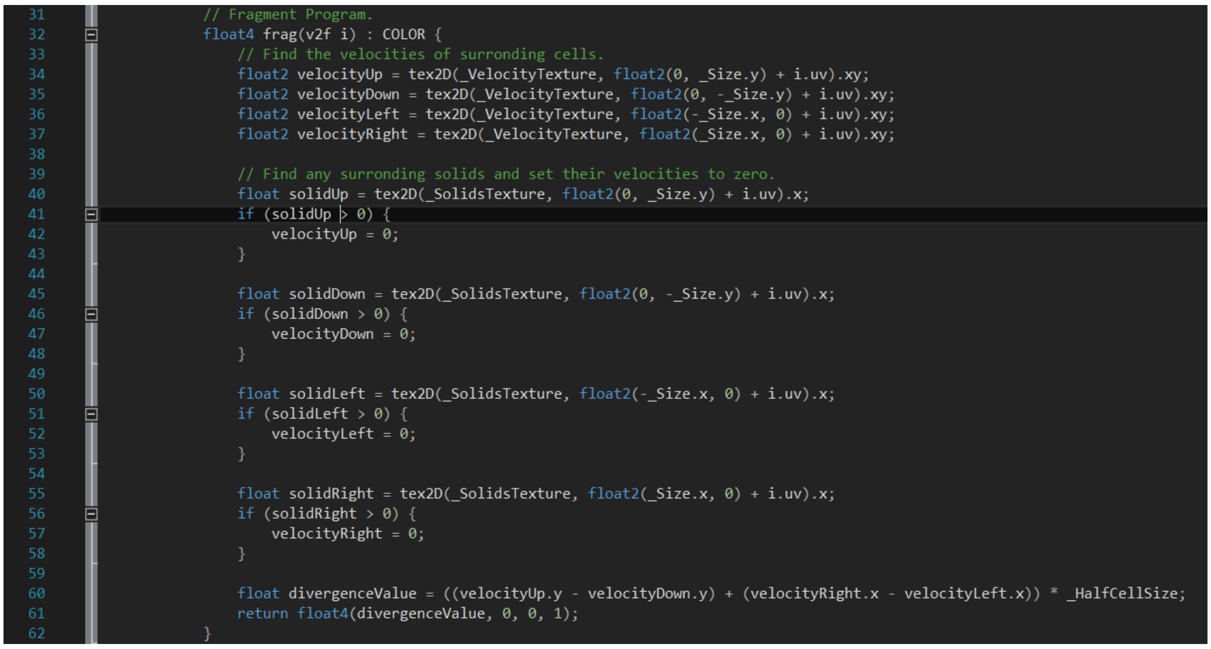
**Impulse shader code:**



```
        // Fragment program.
        float4 frag (v2f i) : COLOR {
            float dist = distance(_Location, i.uv);
            float source = tex2D(_SourceTexture, i.uv).x;

            float impulse = 0;
            if (dist < _Radius) {
                float difference = (_Radius - dist) * 0.5;
                impulse = min(difference, 1.0);
            }

            return max(0, lerp(source, _Fill, impulse)).xxxx;
        }
```

The last part of the simulation involves the projection of the fluid. We start by calculating the fluid divergence by taking the velocities of the surrounding texture ‘grid’ cells and adding the difference in each velocity axes multiplied by half our texture cell size.

**Divergence shader code:**

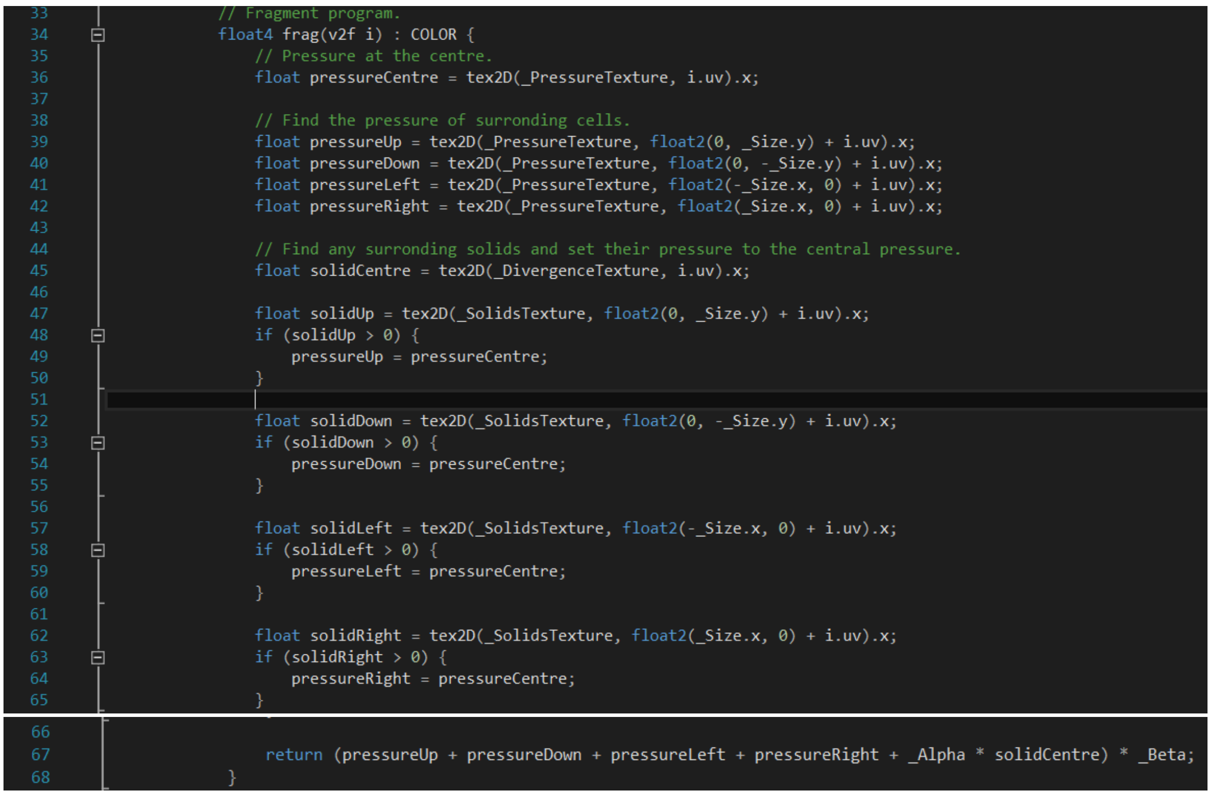


```
        // Fragment Program.
        float4 frag(v2f i) : COLOR {
            // Find the velocities of surronding cells.
            float2 velocityUp = tex2D(_VelocityTexture, float2(0, _Size.y) + i.uv).xy;
            float2 velocityDown = tex2D(_VelocityTexture, float2(0, -_Size.y) + i.uv).xy;
            float2 velocityLeft = tex2D(_VelocityTexture, float2(-_Size.x, 0) + i.uv).xy;
            float2 velocityRight = tex2D(_VelocityTexture, float2(_Size.x, 0) + i.uv).xy;

            // Find any surronding solids and set their velocities to zero.
            float solidUp = tex2D(_SolidsTexture, float2(0, _Size.y) + i.uv).x;
            if (solidUp > 0) {
                velocityUp = 0;
            }

            float solidDown = tex2D(_SolidsTexture, float2(0, -_Size.y) + i.uv).x;
            if (solidDown > 0) {
                velocityDown = 0;
            }

            float solidLeft = tex2D(_SolidsTexture, float2(-_Size.x, 0) + i.uv).x;
            if (solidLeft > 0) {
                velocityLeft = 0;
            }

            float solidRight = tex2D(_SolidsTexture, float2(_Size.x, 0) + i.uv).x;
            if (solidRight > 0) {
                velocityRight = 0;
            }

            float divergenceValue = ((velocityUp.y - velocityDown.y) + (velocityRight.x - velocityLeft.x)) * _HalfCellSize;
            return float4(divergenceValue, 0, 0, 1);
        }
```

This divergence texture can then be passed to our Jacobi iterations to help solve the pressure equation. To begin, I pass a pressure texture of all zeros (an initial ‘guess’ pressure field) to the

Jacobi iterations. In order to achieve good convergence on the solution I have used fifty Jacobi iterations in this simulation.

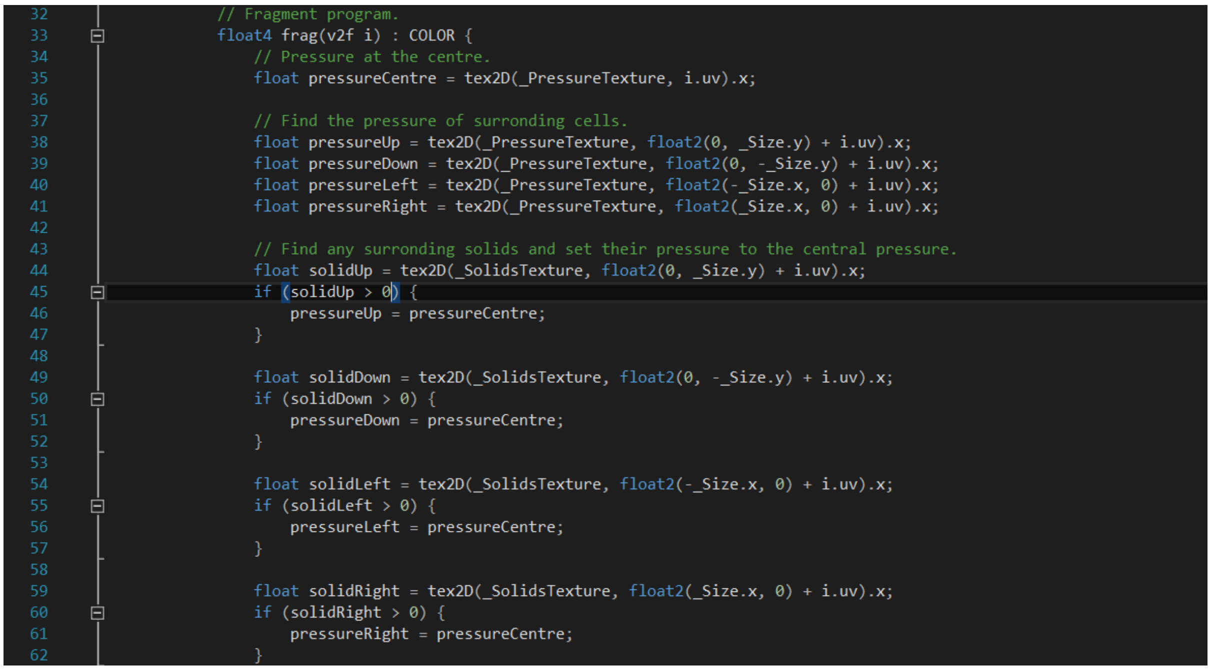
**Jacobi shader code:**



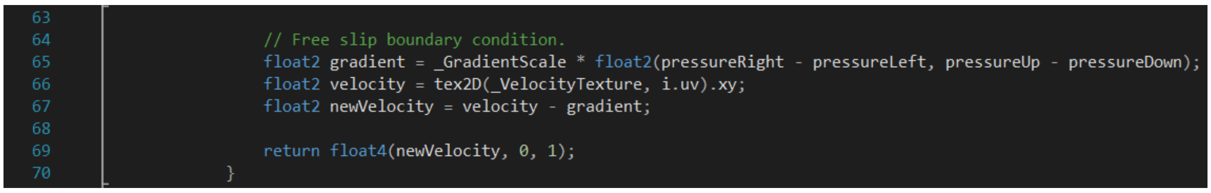
```
// Fragment program.
float4 frag(v2f i) : COLOR {
    // Pressure at the centre.
    float pressureCentre = tex2D(_PressureTexture, i.uv).x;

    // Find the pressure of surronding cells.
    float pressureUp = tex2D(_PressureTexture, float2(0, _Size.y) + i.uv).x;
    float pressureDown = tex2D(_PressureTexture, float2(0, -_Size.y) + i.uv).x;
    float pressureLeft = tex2D(_PressureTexture, float2(-_Size.x, 0) + i.uv).x;
    float pressureRight = tex2D(_PressureTexture, float2(_Size.x, 0) + i.uv).x;

    // Find any surrounding solids and set their pressure to the central pressure.
    float solidCentre = tex2D(_DivergenceTexture, i.uv).x;

    float solidUp = tex2D(_SolidsTexture, float2(0, _Size.y) + i.uv).x;
    if (solidUp > 0) {
        pressureUp = pressureCentre;
    }

    float solidDown = tex2D(_SolidsTexture, float2(0, -_Size.y) + i.uv).x;
    if (solidDown > 0) {
        pressureDown = pressureCentre;
    }

    float solidLeft = tex2D(_SolidsTexture, float2(-_Size.x, 0) + i.uv).x;
    if (solidLeft > 0) {
        pressureLeft = pressureCentre;
    }

    float solidRight = tex2D(_SolidsTexture, float2(_Size.x, 0) + i.uv).x;
    if (solidRight > 0) {
        pressureRight = pressureCentre;
    }

    return (pressureUp + pressureDown + pressureLeft + pressureRight + _Alpha * solidCentre) * _Beta;
}
```

After we have sufficiently ‘solved’ the pressure equation, the final step is to subtract the gradient of the final pressure field from the velocity texture and enforce a free slip boundary condition within the box.

Gradient shader code:



```
// Fragment program.
float4 frag(v2f i) : COLOR {
    // Pressure at the centre.
    float pressureCentre = tex2D(_PressureTexture, i.uv).x;

    // Find the pressure of surronding cells.
    float pressureUp = tex2D(_PressureTexture, float2(0, _Size.y) + i.uv).x;
    float pressureDown = tex2D(_PressureTexture, float2(0, -_Size.y) + i.uv).x;
    float pressureLeft = tex2D(_PressureTexture, float2(-_Size.x, 0) + i.uv).x;
    float pressureRight = tex2D(_PressureTexture, float2(_Size.x, 0) + i.uv).x;

    // Find any surrounding solids and set their pressure to the central pressure.
    float solidUp = tex2D(_SolidsTexture, float2(0, _Size.y) + i.uv).x;
    if (solidUp > 0) {
        pressureUp = pressureCentre;
    }

    float solidDown = tex2D(_SolidsTexture, float2(0, -_Size.y) + i.uv).x;
    if (solidDown > 0) {
        pressureDown = pressureCentre;
    }

    float solidLeft = tex2D(_SolidsTexture, float2(-_Size.x, 0) + i.uv).x;
    if (solidLeft > 0) {
        pressureLeft = pressureCentre;
    }

    float solidRight = tex2D(_SolidsTexture, float2(_Size.x, 0) + i.uv).x;
    if (solidRight > 0) {
        pressureRight = pressureCentre;
    }
```



```
            
            // Free slip boundary condition.
            float2 gradient = _GradientScale * float2(pressureRight - pressureLeft, pressureUp - pressureDown);
            float2 velocity = tex2D(_VelocityTexture, i.uv).xy;
            float2 newVelocity = velocity - gradient;

            return float4(newVelocity, 0, 1);
        }
```

## Future Development

Moving forward I am going to look into adapting my existing project into a third dimension implementation.  Joe Stam's 'Stable Fluids' SIGGRAPH paper (4) (one which the GPU Gems article heavily draws from) will likely continue to be valuable when addressing another dimension.

## Youtube Video

## Github Link

https://github.com/adamjoyce/fluid-simulation

## References

1. http://http.developer.nvidia.com/GPUGems/gpugems_ch38.html
2.
3. http://prideout.net/blog/?p=58
4. http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/ns.pdf
5. https://en.wikipedia.org/wiki/Jacobi_method
6. http://wiki.unity3d.com/index.php?title=Shader_Code