CPI 411 Graphics for Games

Real Time Simulation of 3D Fluids

This lab will cover physically based animation of fluids in 3D (specifically smoke).

A. Preamble

In real life, fluids consist of infinitesimally small particles all interfacing with one another as well as outside forces (such as gravity). To render this phenomena, there are two popular routes that graphics programmers use:

- Eulerian implementations (grid based)
- Lagrangian implementations (non-grid based, possibly particles)

This lab implements a Eulerian method that discretizes the fluid into buffers and acts on them via *compute shaders*- shaders that run outside the usual graphics pipeline and aren't restricted by vertex or fragment data. Set-up and installation of compute shaders in Monogame will be explained in **Section B**. The fluid we render is considered ***incompressible***, meaning compressing the fluid volume doesn't change the volume of the fluid (water is an example of an *incompressible* fluid).

One cell of our eulerian space represents multiple traits of the fluid: pressure, density, temperature, and most notably: *velocity*. Velocity is important as it describes the movement of objects, densities, and other quantities inside of the fluid as well as the movement of the velocity field itself. This process is named *advection* and is one of the Navier-Stokes equations- a set of partial differential equations that we solve to express the flow of incompressible, frictionless fluids.

Here are the terms in the Navier-Stokes equations we consider in simulation:

- **Advection**
    - The process of the velocity field of a fluid transporting values across the Eulerian space as well as itself.
- **Pressure**
    - Since fluids are made of small molecules, applying a force to a fluid won't immediately act on the entire fluid. The molecules where force was applied pushes neighboring molecules and so on and pressure amasses. This pressure leads to an acceleration of the fluid, which can be visualized as the molecules squishing or sloshing around.
- **Diffusion**
    - Fluids have *viscosity* values, or a measure of how resistant they are to flow. This resistance to flow diffuses the momentum across the Eulerian space, resulting in velocity from cell to neighboring cell inside the buffer.
- **External Forces**
    - Any force that is attributed to external factors. These forces can be local to a region of a fluid (slapping the surface of oobleck) or acting on the entire fluid (gravity). These are named *local* and *body* forces respectfully.

The math behind solving the Navier-Stokes equations is out of the scope of this class, if you are interested in the process of solving this equation check out GPU Gems 1 Chapter 38 as well as GPU Gems 3 Chapter 30. The underlying theory involves splitting the simulation into timesteps, where each timestep we use the current values of the fluid and their partial differential equations to calculate new values for the next step. The pressure equation is solved with a Jacobi Iterator: a solver for Poisson equations. Within these calculations we consider *boundary conditions*, or how the fluid acts at the edges of our Eulerian space.

One final note about this simulation before diving into implementation- rendering fluids comes with its own challenges and processes. The fluid we render, smoke, involves keeping track of a temperature scalar value that affects the dynamics of the fluid as well as the insertion of a Gaussian “splat” effect to make it look realistic. Rasterization-based frameworks (Monogame) don’t readily have methods to render 3D data, so we must implement a ray-marching shader to visualize our fluid.

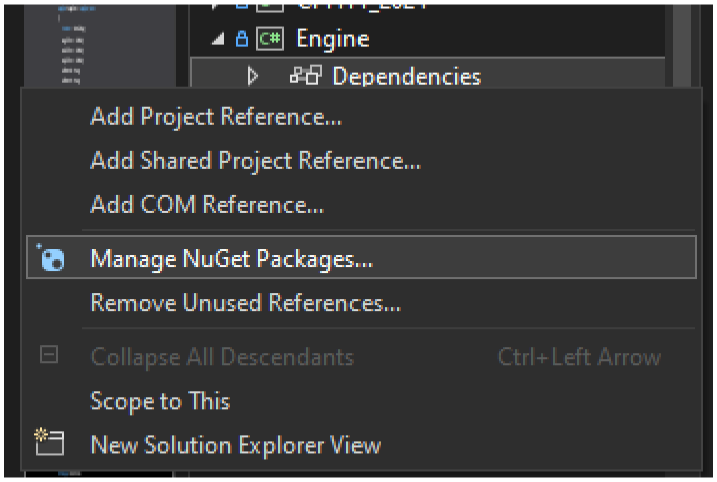
B. Monogame Setup

CPUs are excellent for intensive sequential computing tasks and program executions, whereas the GPU thrives with parallel processing due to its multithreading-oriented architecture. Due to the nature of fluid simulation where we solve the same simple equations for each grid cell (leading to millions of small computations per timestep) this computation is best done on the GPU.

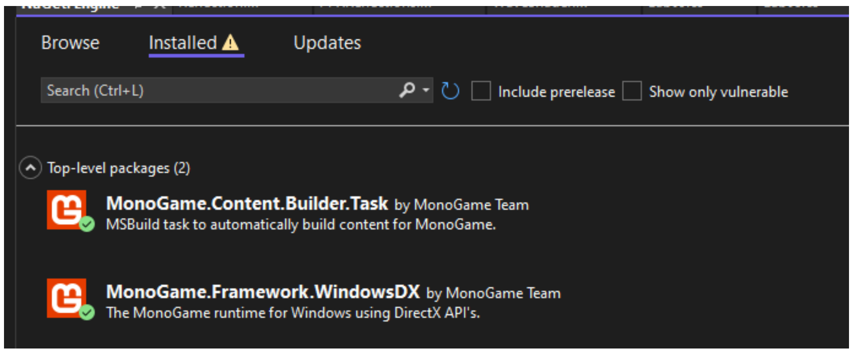
Compute shaders allow us to leverage the power of the GPU to perform these mass calculations while not being limited to the regular graphics pipeline (acting on a vertex or fragment). Another benefit of compute shaders is that they can *output* data to buffers and share memory across different threads of execution. This fits our needs perfectly and allows us to simulate at an infinitely higher framerate than a CPU would be capable of.

Base Monogame lacks the support for Compute Shaders, however a fork of Monogame (created by cpt-max) exists that supports modern graphics APIs including Compute Shaders. Switching a project over to this fork is easy, and can be done following these steps:

1. Right click the project (or Dependencies in the project) and click `Manage NuGet Packages…`



You will see your currently installed Monogame Packages here in the Installed folder

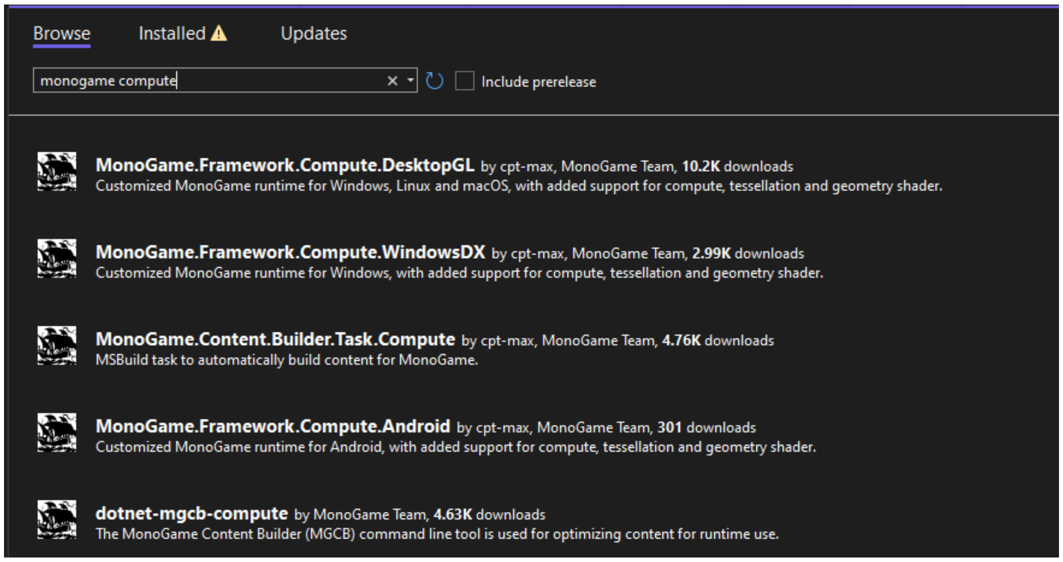


Top-level packages (2)

MonoGame.Content.Builder.Task by MonoGame Team
MSBuild task to automatically build content for MonoGame.

MonoGame.Framework.WindowsDX by MonoGame Team
The MonoGame runtime for Windows using DirectX API's.

2. Click 'Browse' and search for `monogame compute`.



MonoGame.Framework.Compute.DesktopGL by cpt-max, MonoGame Team, **10.2K** downloads
Customized MonoGame runtime for Windows, Linux and macOS, with added support for compute, tessellation and geometry shader.

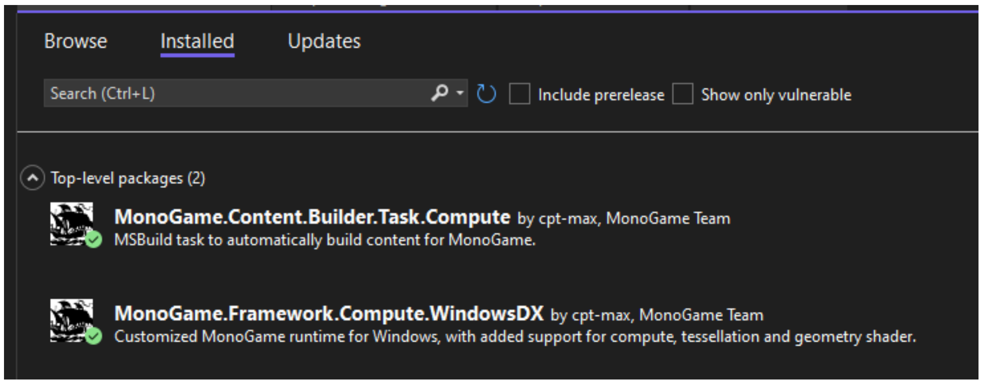
MonoGame.Framework.Compute.WindowsDX by cpt-max, MonoGame Team, **2.99K** downloads
Customized MonoGame runtime for Windows, with added support for compute, tessellation and geometry shader.

MonoGame.Content.Builder.Task.Compute by cpt-max, MonoGame Team, **4.76K** downloads
MSBuild task to automatically build content for MonoGame.

MonoGame.Framework.Compute.Android by cpt-max, MonoGame Team, **301** downloads
Customized MonoGame runtime for Android, with added support for compute, tessellation and geometry shader.

dotnet-mgcb-compute by MonoGame Team, **4.63K** downloads
The MonoGame Content Builder (MGCB) command line tool is used for optimizing content for runtime use.

Install the Compute versions of the Monogame packages you are using, they should follow the exact same naming with the word 'compute' added in. Go back to your Installed tab and uninstall the old Monogame extensions from this project (don't worry, these extensions are project based- this won't uninstall Monogame from your solution). Your installed folder should now look like this:

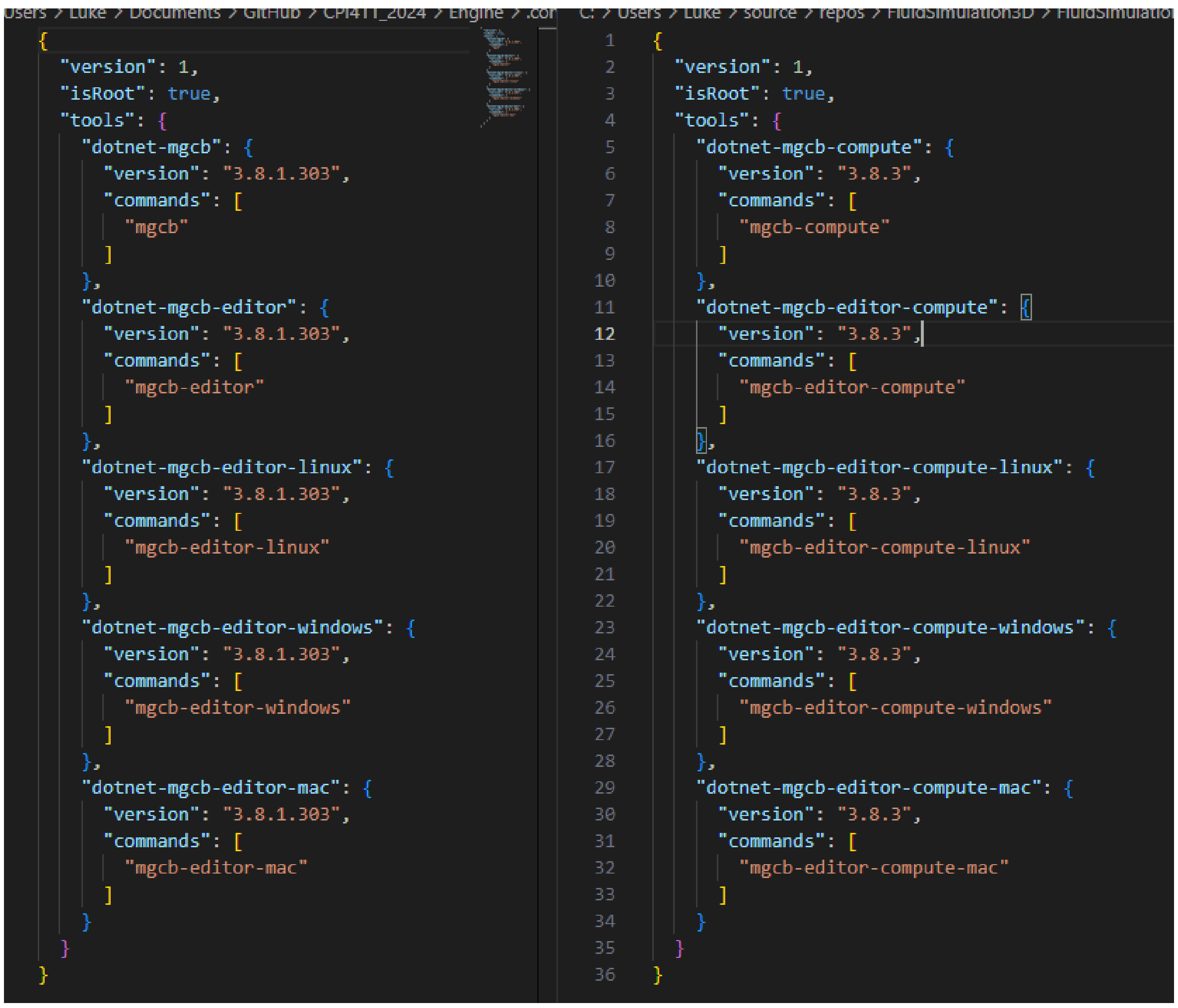


Top-level packages (2)

MonoGame.Content.Builder.Task.Compute by cpt-max, MonoGame Team
MSBuild task to automatically build content for MonoGame.

MonoGame.Framework.Compute.WindowsDX by cpt-max, MonoGame Team
Customized MonoGame runtime for Windows, with added support for compute, tessellation and geometry shader.

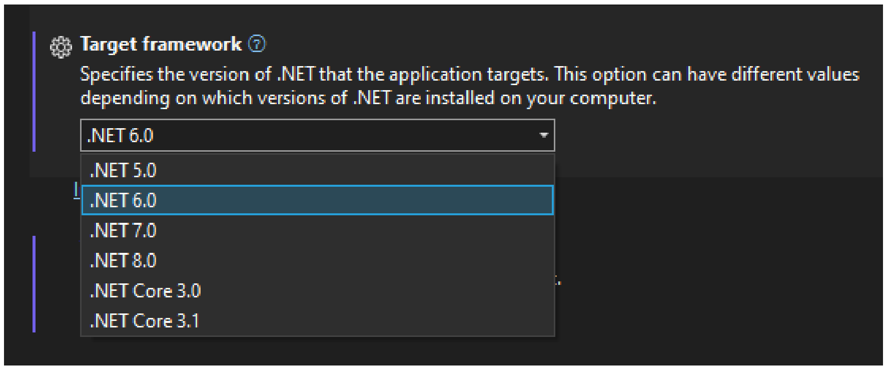
3. Locate the project in file explorer and open the .config>dotnet-tools.json file in your text editor of choice. Change the contents of the file from the left side of the image below to the right side.



```
Users > Luke > Documents > GitHub > CPI411_2024 > Engine > .con
{
  "version": 1,
  "isRoot": true,
  "tools": {
    "dotnet-mgcb": {
      "version": "3.8.1.303",
      "commands": [
        "mgcb"
      ]
    },
    "dotnet-mgcb-editor": {
      "version": "3.8.1.303",
      "commands": [
        "mgcb-editor"
      ]
    },
    "dotnet-mgcb-editor-linux": {
      "version": "3.8.1.303",
      "commands": [
        "mgcb-editor-linux"
      ]
    },
    "dotnet-mgcb-editor-windows": {
      "version": "3.8.1.303",
      "commands": [
        "mgcb-editor-windows"
      ]
    },
    "dotnet-mgcb-editor-mac": {
      "version": "3.8.1.303",
      "commands": [
        "mgcb-editor-mac"
      ]
    }
  }
}
```

```
C: > Users > Luke > source > repos > FluidSimulation3D > FluidSimulatio
{
  "version": 1,
  "isRoot": true,
  "tools": {
    "dotnet-mgcb-compute": {
      "version": "3.8.3",
      "commands": [
        "mgcb-compute"
      ]
    },
    "dotnet-mgcb-editor-compute": {
      "version": "3.8.3",
      "commands": [
        "mgcb-editor-compute"
      ]
    },
    "dotnet-mgcb-editor-compute-linux": {
      "version": "3.8.3",
      "commands": [
        "mgcb-editor-compute-linux"
      ]
    },
    "dotnet-mgcb-editor-compute-windows": {
      "version": "3.8.3",
      "commands": [
        "mgcb-editor-compute-windows"
      ]
    },
    "dotnet-mgcb-editor-compute-mac": {
      "version": "3.8.3",
      "commands": [
        "mgcb-editor-compute-mac"
      ]
    }
  }
}
```

4. Inside of Visual Studio, right-click the project file and click Properties. Change the Target Framework property from .NET 6.0 to .NET 8.0



Target framework

Specifies the version of .NET that the application targets. This option can have different values depending on which versions of .NET are installed on your computer.

.NET 6.0

C. Shader Files

Solving the Navier-Stokes equations and rendering a fluid in an alluring, realistic manner requires many compute functions to act on the buffer- so I decided the best implementation would be splitting each function into its own .fx file. These effect files are as follows:

- ApplyAdvection.fx
- ApplyBuoyancy.fx
- ApplyImpulse.fx
- ComputeBorders.fx
- ComputeConfinements.fx
- ComputeDivergence.fx
- ComputeJacobi.fx
- ComputeProjection.fx
- ComputeVorticity.fx
- **Raymarcher.fx <- The only non-compute shader**

*Note that ComputeConfinement, ComputeVorticity, and ComputeBuoyancy are used to mimic passably realistic smoke physics; Raymarcher is used to render the fluid in 3D space. The rest are used to simulate the fluid and are generally used for all fluid types.

ApplyAdvection.fx

When advecting quantities across the fluid grid, we may be tempted to use forward integration: simply taking the value of the quantity and moving it in the direction of the velocity field and then updating that cell accordingly. This approach has two problems:

1. It's unstable for large timesteps- the simulation can way overshoot and look unrealistic if it advects the quantity too far.
2. The GPU is superior at 'gather' operations rather than 'scatter' operations, or rather operations where you read from memory rather than write to different memory space than the compute kernel is working on.

The solution to this problem is to invert our calculation and use an implicit method. Rather than scattering the value of a quantity to its destination, we trace backwards and read from the grid space where the velocity could've been the last timestep, copying the data to the current cell. This method becomes more GPU efficient as well as becoming stable for arbitrary timesteps.

You may notice that tracing a velocity backwards doesn't cleanly trace to a single grid cell and oftentimes will fall between 8 neighboring grid cells. To remedy this issue, we bilinearly interpolate the values to get a clean read of the data.

One thing to note for this shader is that the Index value (notated idx) that tells us the index of the grid cell we are operating on has a strange calculation. This is because our buffer is actually stored in a 1D buffer instead of a Texture3D, and thus accessing patterns change. In the future this project could be changed to a possibly faster Texture3D however for now all shaders will use this access pattern from here on out.

Here is the ApplyAdvection.fx file:

```
//===========================================================================
// Compute Shader
//===========================================================================
#define GroupSizeXYZ 8

float4 _Size;
float _DeltaTime, _Dissipate, _Forward;

StructuredBuffer<float3> _Velocity;
StructuredBuffer<float> _Obstacles;

RWStructuredBuffer<float> _Write1f;
StructuredBuffer<float> _Read1f;

RWStructuredBuffer<float3> _Write3f;
StructuredBuffer<float3> _Read3f;

StructuredBuffer<float> _Phi_n_1_hat, _Phi_n_hat;

float3 GetAdvectedPosTexCoords(float3 pos, int idx)
{
    pos -= _DeltaTime * _Forward * _Velocity[idx];

    return pos;
}

float SampleBilinear(StructuredBuffer<float> buffer, float3 uv, float3 size)
{
    int x = uv.x;
    int y = uv.y;
    int z = uv.z;

    int X = size.x;
    int XY = size.x * size.y;

    float fx = uv.x - x;
    float fy = uv.y - y;
    float fz = uv.z - z;

    int xp1 = min(size.x - 1, x + 1);
    int yp1 = min(size.y - 1, y + 1);
    int zp1 = min(size.z - 1, z + 1);

    float x0 = buffer[x + y * X + z * XY] * (1.0f - fx) + buffer[xp1 + y * X + z * XY] * fx;
    float x1 = buffer[x + y * X + zp1 * XY] * (1.0f - fx) + buffer[xp1 + y * X + zp1 * XY] * fx;

    float x2 = buffer[x + yp1 * X + z * XY] * (1.0f - fx) + buffer[xp1 + yp1 * X + z * XY] * fx;
     float x3 = buffer[x + yp1 * X + zp1 * XY] * (1.0f - fx) + buffer[xp1 + yp1 * X + zp1 * XY] *
fx;

    float z0 = x0 * (1.0f - fz) + x1 * fz;
    float z1 = x2 * (1.0f - fz) + x3 * fz;

    return z0 * (1.0f - fy) + z1 * fy;
}

float3 SampleBilinear(StructuredBuffer<float3> buffer, float3 uv, float3 size)
{
    int x = uv.x;
    int y = uv.y;
    int z = uv.z;

    int X = size.x;
    int XY = size.x * size.y;

    float fx = uv.x - x;
    float fy = uv.y - y;
    float fz = uv.z - z;
    int xp1 = min(size.x - 1, x + 1);
    int yp1 = min(size.y - 1, y + 1);
    int zp1 = min(size.z - 1, z + 1);

    float3 x0 = buffer[x + y * X + z * XY] * (1.0f - fx) + buffer[xp1 + y * X + z * XY] * fx;
    float3 x1 = buffer[x + y * X + zp1 * XY] * (1.0f - fx) + buffer[xp1 + y * X + zp1 * XY] * fx;

    float3 x2 = buffer[x + yp1 * X + z * XY] * (1.0f - fx) + buffer[xp1 + yp1 * X + z * XY] * fx;
    float3 x3 = buffer[x + yp1 * X + zp1 * XY] * (1.0f - fx) + buffer[xp1 + yp1 * X + zp1 * XY] *
fx;

    float3 z0 = x0 * (1.0f - fz) + x1 * fz;
    float3 z1 = x2 * (1.0f - fz) + x3 * fz;

    return z0 * (1.0f - fy) + z1 * fy;
}

[numthreads(GroupSizeXYZ, GroupSizeXYZ, GroupSizeXYZ)]
void AdvectVelocity(uint3 id : SV_DispatchThreadID)
{
    int idx = id.x + id.y * _Size.x + id.z * _Size.x * _Size.y;

    if (_Obstacles[idx] > 0.1)
    {
        _Write3f[idx] = float3(0, 0, 0);
        return;
    }

    float3 uv = GetAdvectedPosTexCoords(id, idx);

    _Write3f[idx] = SampleBilinear(_Read3f, uv, _Size.xyz) * _Dissipate;
}

[numthreads(GroupSizeXYZ, GroupSizeXYZ, GroupSizeXYZ)]
void Advect(uint3 id : SV_DispatchThreadID)
{
    int idx = id.x + id.y * _Size.x + id.z * _Size.x * _Size.y;

    if (_Obstacles[idx] > 0.1)
    {
        _Write1f[idx] = 0;
        return;
    }

    float3 uv = GetAdvectedPosTexCoords(id, idx);

    _Write1f[idx] = max(0, SampleBilinear(_Read1f, uv, _Size.xyz) * _Dissipate);
}

[numthreads(GroupSizeXYZ, GroupSizeXYZ, GroupSizeXYZ)]
void AdvectMacCormack(uint3 id : SV_DispatchThreadID)
{
    int idx = id.x + id.y * _Size.x + id.z * _Size.x * _Size.y;

    if (_Obstacles[idx] > 0.1)
    {
        _Write1f[idx] = 0;
        return;
    }

    float3 uv = GetAdvectedPosTexCoords(id, idx);

    float r;
    float4 halfVolumeDim = _Size / 2;
    float3 diff = abs(halfVolumeDim.xyz - id);

    // Must use regular semi-Lagrangian advection instead of MacCormack at the volume boundaries
      if ((diff.x > (halfVolumeDim.x - 4)) || (diff.y > (halfVolumeDim.y - 4)) || (diff.z >
(halfVolumeDim.z - 4)))
    {
        r = SampleBilinear(_Read1f, uv, _Size.xyz);
    }
    else
    {
        int idx0 = (id.x - 1) + (id.y - 1) * _Size.x + (id.z - 1) * _Size.x * _Size.y;
        int idx1 = (id.x - 1) + (id.y - 1) * _Size.x + (id.z + 1) * _Size.x * _Size.y;

        int idx2 = (id.x - 1) + (id.y + 1) * _Size.x + (id.z - 1) * _Size.x * _Size.y;
        int idx3 = (id.x - 1) + (id.y + 1) * _Size.x + (id.z + 1) * _Size.x * _Size.y;

        int idx4 = (id.x + 1) + (id.y - 1) * _Size.x + (id.z - 1) * _Size.x * _Size.y;
        int idx5 = (id.x + 1) + (id.y - 1) * _Size.x + (id.z + 1) * _Size.x * _Size.y;

        int idx6 = (id.x + 1) + (id.y + 1) * _Size.x + (id.z - 1) * _Size.x * _Size.y;
        int idx7 = (id.x + 1) + (id.y + 1) * _Size.x + (id.z + 1) * _Size.x * _Size.y;

        float nodes[8];
        nodes[0] = _Read1f[idx0];
        nodes[1] = _Read1f[idx1];

        nodes[2] = _Read1f[idx2];
        nodes[3] = _Read1f[idx3];

        nodes[4] = _Read1f[idx4];
        nodes[5] = _Read1f[idx5];

        nodes[6] = _Read1f[idx6];
        nodes[7] = _Read1f[idx7];

            float minPhi = min(min(min(min(min(min(min(nodes[0], nodes[1]), nodes[2]), nodes[3]),
nodes[4]), nodes[5]), nodes[6]), nodes[7]);

            float maxPhi = max(max(max(max(max(max(max(nodes[0], nodes[1]), nodes[2]), nodes[3]),
nodes[4]), nodes[5]), nodes[6]), nodes[7]);

        r = SampleBilinear(_Phi_n_1_hat, uv, _Size.xyz) + 0.5f * (_Read1f[idx] - _Phi_n_hat[idx]);

        r = max(min(r, maxPhi), minPhi);
    }

    _Write1f[idx] = max(0, r * _Dissipate);
}

technique Tech0
{
    pass Pass0
    {
        ComputeShader = compile cs_5_0 AdvectVelocity();
    }
}
technique Tech1
{
    pass Pass0
    {
        ComputeShader = compile cs_5_0 Advect();
    }
}
technique Tech2
{
    pass Pass0
    {
        ComputeShader = compile cs_5_0 AdvectMacCormack();
    }
}
```

You may notice that there are 3 different techniques in this shader. Tech0 is called for advecting the velocity itself (advecting float3 values), Tech1 is called for advecting scalar quantities (float values) and Tech2 is a special type of advection that we can optionally use called *MacCormack* advection. This advection type reduces numerical smoothing that can lower the visual quality of the simulation at the cost of more computing power required to use it.

ApplyBuoyancy.fx

The *buoyant force* describes the influence that temperature has on smoke, notably the fact that hot smoke rises and cool smoke falls. We advect a temperature value (scalar) across the fluid, which then influences the dynamics in our ApplyBuoyancy compute shader.

Here is the ApplyBuoyancy.fx file:

```
//==============================================================================
// Compute Shader
//==============================================================================
#define GroupSizeXYZ 8

float4 _Size;
float3 _Up;
float _DeltaTime, _Buoyancy, _Weight;

RWStructuredBuffer<float3> _Write;
StructuredBuffer<float3> _Velocity;
StructuredBuffer<float> _Density, _Temperature;

[numthreads(GroupSizeXYZ, GroupSizeXYZ, GroupSizeXYZ)]
void BuoyantForce(int3 id : SV_DispatchThreadID)
{
    int idx = id.x + id.y * _Size.x + id.z * _Size.x * _Size.y;

    float T = _Temperature[idx];
    float D = _Density[idx];
    float3 V = _Velocity[idx];

    V += (_DeltaTime * T * _Buoyancy - D * _Weight) * _Up;

    _Write[idx] = V;
}

technique Tech0
{
    pass Pass0
    {
        ComputeShader = compile cs_5_0 BuoyantForce();
    }
}
```

ApplyImpulse.fx

Impulse represents the 'outside forces' part of the Navier-Stokes equations. In this simulation, we have an input position and radius that applies this force in the Impulse shader.

Here is the ApplyImpulse.fx file:

```
//============================================================================
// Compute Shader
//============================================================================
#define GroupSizeXYZ 8

float _Radius, _Amount, _DeltaTime;
float3 _Pos;
float4 _Size;

RWStructuredBuffer<float> _Write;
StructuredBuffer<float> _Read, _Reaction;

[numthreads(GroupSizeXYZ, GroupSizeXYZ, GroupSizeXYZ)]
void GaussImpulse(uint3 id : SV_DispatchThreadID)
{
    float3 pos = id / (_Size.xyz - 1.0f) - _Pos;
    float mag = pos.x * pos.x + pos.y * pos.y + pos.z * pos.z;
    float rad2 = _Radius * _Radius;

    float amount = exp(-mag / rad2) * _Amount * _DeltaTime;

    int idx = id.x + id.y * _Size.x + id.z * _Size.x * _Size.y;

    _Write[idx] = _Read[idx] + amount;
}

technique Tech0
{
    pass Pass0
    {
        ComputeShader = compile cs_5_0 GaussImpulse();
    }
}
```

ComputeBorders.fx

The ComputeBorders shader computes a buffer that specifies where ‘borders’ or impassable grid cells lie. Impassable cells are used in other compute shaders to specify actions taken on boundary conditions, such as pressure building up when it hits borders before curving outwards. Our implementation takes the simple approach of counting only the cells on the edge of the buffer as borders.

This shader could be expanded to read mesh data from other objects within a scene and mark spaces in the fluid that are obstructed, leading to a realistic fluid-hard surface interface.

**Note: This buffer is of float values due to an issue with the Monogame compute branch and bool buffers, mainly they don’t work. We use a float value of 1 for a border and a value of 0 for no border.*

Here is the ComputeBorders.fx file:

```
//============================================================================
// Compute Shader
//============================================================================
#define GroupSizeXYZ 8

float4 _Size;

RWStructuredBuffer<float> _Write;

[numthreads(GroupSizeXYZ, GroupSizeXYZ, GroupSizeXYZ)]
void Borders(int3 id : SV_DispatchThreadID)
{
    int idx = id.x + id.y * _Size.x + id.z * _Size.x * _Size.y;

    float obstacle = 0;

    if (id.x - 1 < 0)
        obstacle = 1;
    if (id.x + 1 > (int) _Size.x - 1)
        obstacle = 1;

    if (id.y - 1 < 0)
        obstacle = 1;
    if (id.y + 1 > (int) _Size.y - 1)
        obstacle = 1;

    if (id.z - 1 < 0)
        obstacle = 1;
    if (id.z + 1 > (int) _Size.z - 1)
        obstacle = 1;

    _Write[idx] = obstacle;
}


technique Tech0
{
    pass Pass0
    {
        ComputeShader = compile cs_5_0 Borders();
    }
}
```

ComputeConfinement.fx

The **Vorticity Confinement** is a computational fluid dynamics model that sets out to solve vortex dominated flows. We implement this method by first computing the vorticity from our velocity (or rotational flows) and then computing a normalized vector field from said vorticity. This shader represents the confinement part (the normalized field) of this process.

Here is the ComputeConfinement.fx file:

```
//================================================================================
// Compute Shader
//================================================================================
#define GroupSizeXYZ 8

float _DeltaTime, _Epsilon;
float4 _Size;

RWStructuredBuffer<float3> _Write;
StructuredBuffer<float3> _Vorticity, _Read;

[numthreads(GroupSizeXYZ, GroupSizeXYZ, GroupSizeXYZ)]
void Confine(int3 id : SV_DispatchThreadID)
{
    int idxL = max(0, id.x - 1) + id.y * _Size.x + id.z * _Size.x * _Size.y;
    int idxR = min(_Size.x - 1, id.x + 1) + id.y * _Size.x + id.z * _Size.x * _Size.y;

    int idxB = id.x + max(0, id.y - 1) * _Size.x + id.z * _Size.x * _Size.y;
    int idxT = id.x + min(_Size.y - 1, id.y + 1) * _Size.x + id.z * _Size.x * _Size.y;

    int idxD = id.x + id.y * _Size.x + max(0, id.z - 1) * _Size.x * _Size.y;
    int idxU = id.x + id.y * _Size.x + min(_Size.z - 1, id.z + 1) * _Size.x * _Size.y;

    float omegaL = length(_Vorticity[idxL]);
    float omegaR = length(_Vorticity[idxR]);

    float omegaB = length(_Vorticity[idxB]);
    float omegaT = length(_Vorticity[idxT]);

    float omegaD = length(_Vorticity[idxD]);
    float omegaU = length(_Vorticity[idxU]);

    int idx = id.x + id.y * _Size.x + id.z * _Size.x * _Size.y;

    float3 omega = _Vorticity[idx];

    float3 eta = 0.5 * float3(omegaR - omegaL, omegaT - omegaB, omegaU - omegaD);

    eta = normalize(eta + float3(0.001, 0.001, 0.001));

    float3 force = _DeltaTime * _Epsilon * float3(eta.y * omega.z - eta.z * omega.y, eta.z *
omega.x - eta.x * omega.z, eta.x * omega.y - eta.y * omega.x);

    _Write[idx] = _Read[idx] + force;
}

technique Tech0
{
    pass Pass0
    {
        ComputeShader = compile cs_5_0 Confine();
    }
}
```

ComputeDivergence.fx

Divergence signifies the rate at which *density* exits a region of space. The incompressibility assumption states that our fluid is *divergence free*, or that the fluid never loses any of its density over time. We must be able to compute the divergence of a field to use in the calculation of *pressure* where we ensure our divergence free conditions are kept intact.

Here is the ComputeDivergence.fx file:

```
//================================================================================
// Compute Shader
//================================================================================
#define GroupSizeXYZ 8

float4 _Size;

RWStructuredBuffer<float3> _Write;
StructuredBuffer<float3> _Velocity;
StructuredBuffer<float> _Obstacles;

[numthreads(GroupSizeXYZ, GroupSizeXYZ, GroupSizeXYZ)]
void Divergence(int3 id : SV_DispatchThreadID)
{
    int idxL = max(0, id.x - 1) + id.y * _Size.x + id.z * _Size.x * _Size.y;
    int idxR = min(_Size.x - 1, id.x + 1) + id.y * _Size.x + id.z * _Size.x * _Size.y;

    int idxB = id.x + max(0, id.y - 1) * _Size.x + id.z * _Size.x * _Size.y;
    int idxT = id.x + min(_Size.y - 1, id.y + 1) * _Size.x + id.z * _Size.x * _Size.y;

    int idxD = id.x + id.y * _Size.x + max(0, id.z - 1) * _Size.x * _Size.y;
    int idxU = id.x + id.y * _Size.x + min(_Size.z - 1, id.z + 1) * _Size.x * _Size.y;

    float3 L = _Velocity[idxL];
    float3 R = _Velocity[idxR];

    float3 B = _Velocity[idxB];
    float3 T = _Velocity[idxT];

    float3 D = _Velocity[idxD];
    float3 U = _Velocity[idxU];

    float3 obstacleVelocity = float3(0, 0, 0);

    // Possibly remove checks and directly set these values to borders as they are 0 or 1
    if (_Obstacles[idxL] > 0.1)
        L = obstacleVelocity;
    if (_Obstacles[idxR] > 0.1)
        R = obstacleVelocity;

    if (_Obstacles[idxB] > 0.1)
        B = obstacleVelocity;
    if (_Obstacles[idxT] > 0.1)
        T = obstacleVelocity;

    if (_Obstacles[idxD] > 0.1)
        D = obstacleVelocity;
    if (_Obstacles[idxU] > 0.1)
        U = obstacleVelocity;

    float divergence = 0.5 * ((R.x - L.x) + (T.y - B.y) + (U.z - D.z));
    int idx = id.x + id.y * _Size.x + id.z * _Size.x * _Size.y;

    _Write[idx] = float3(divergence, 0, 0);
}

technique Tech0
{
    pass Pass0
    {
        ComputeShader = compile cs_5_0 Divergence();
    }
}
```

ComputeJacobi.fx

Pressure is easily solvable for compressible flow- it is part of the state equation. For incompressible flow however, we obtain our pressure field through a nonlinear Poisson equation. We solve this equation using an iterator- namely the *Jacobi Iterator*. There are other iteration methods which may converge quicker, however the Jacobi is simplest and is a fine implementation.

Here is the ComputeJacobi.fx file:

```
//=============================================================================
// Compute Shader
//=============================================================================
#define GroupSizeXYZ 8

float4 _Size;

RWStructuredBuffer<float> _Write;
StructuredBuffer<float> _Pressure, _Obstacles;
StructuredBuffer<float3> _Divergence;

[numthreads(GroupSizeXYZ, GroupSizeXYZ, GroupSizeXYZ)]
void Jacobi(int3 id : SV_DispatchThreadID)
{
    int idxL = max(0, id.x - 1) + id.y * _Size.x + id.z * _Size.x * _Size.y;
    int idxR = min(_Size.x - 1, id.x + 1) + id.y * _Size.x + id.z * _Size.x * _Size.y;

    int idxB = id.x + max(0, id.y - 1) * _Size.x + id.z * _Size.x * _Size.y;
    int idxT = id.x + min(_Size.y - 1, id.y + 1) * _Size.x + id.z * _Size.x * _Size.y;

    int idxD = id.x + id.y * _Size.x + max(0, id.z - 1) * _Size.x * _Size.y;
    int idxU = id.x + id.y * _Size.x + min(_Size.z - 1, id.z + 1) * _Size.x * _Size.y;

    float L = _Pressure[idxL];
    float R = _Pressure[idxR];

    float B = _Pressure[idxB];
    float T = _Pressure[idxT];

    float D = _Pressure[idxD];
    float U = _Pressure[idxU];

    int idx = id.x + id.y * _Size.x + id.z * _Size.x * _Size.y;

    float C = _Pressure[idx];

    float divergence = _Divergence[idx].r;

    if (_Obstacles[idxL] > 0.1)
        L = C;
    if (_Obstacles[idxR] > 0.1)
        R = C;

    if (_Obstacles[idxB] > 0.1)
        B = C;
    if (_Obstacles[idxT] > 0.1)
        T = C;

    if (_Obstacles[idxD] > 0.1)
        D = C;
    if (_Obstacles[idxU] > 0.1)
```

```
        U = C;

    _Write[idx] = (L + R + B + T + U + D - divergence) / 6.0;
}

technique Tech0
{
    pass Pass0
    {
        ComputeShader = compile cs_5_0 Jacobi();
    }
}
```

ComputeProjection.fx

The projection component takes the pressure values we previously calculated along with the velocity to ensure that our fluid is still incompressible. If you imagine a fluid pushing against the edge of the surface and getting compressed, naturally it would push to the sides and move outwards. This compute shader makes our velocity take this into account and adjusts it accordingly.

Here is the ComputeProjection.fx file:

```
//================================================================================
// Compute Shader
//================================================================================
#define GroupSizeXYZ 8

float4 _Size;

RWStructuredBuffer<float3> _Write;
StructuredBuffer<float> _Pressure, _Obstacles;
StructuredBuffer<float3> _Velocity;

[numthreads(GroupSizeXYZ, GroupSizeXYZ, GroupSizeXYZ)]
void Project(int3 id : SV_DispatchThreadID)
{
    int idx = id.x + id.y * _Size.x + id.z * _Size.x * _Size.y;

    if (_Obstacles[idx] > 0.1)
    {
        _Write[idx] = float3(0, 0, 0);
        return;
    }

    int idxL = max(0, id.x - 1) + id.y * _Size.x + id.z * _Size.x * _Size.y;
    int idxR = min(_Size.x - 1, id.x + 1) + id.y * _Size.x + id.z * _Size.x * _Size.y;

    int idxB = id.x + max(0, id.y - 1) * _Size.x + id.z * _Size.x * _Size.y;
    int idxT = id.x + min(_Size.y - 1, id.y + 1) * _Size.x + id.z * _Size.x * _Size.y;

    int idxD = id.x + id.y * _Size.x + max(0, id.z - 1) * _Size.x * _Size.y;
    int idxU = id.x + id.y * _Size.x + min(_Size.z - 1, id.z + 1) * _Size.x * _Size.y;

    float L = _Pressure[idxL];
    float R = _Pressure[idxR];

    float B = _Pressure[idxB];
    float T = _Pressure[idxT];

    float D = _Pressure[idxD];
    float U = _Pressure[idxU];

    float C = _Pressure[idx];

    float3 mask = float3(1, 1, 1);

    if (_Obstacles[idxL] > 0.1)
    {
        L = C;
        mask.x = 0;
    }
    if (_Obstacles[idxR] > 0.1)
    {
        R = C;
        mask.x = 0;
    }

    if (_Obstacles[idxB] > 0.1)
    {
        B = C;
        mask.y = 0;
    }
    if (_Obstacles[idxT] > 0.1)
    {
        T = C;
        mask.y = 0;
    }

    if (_Obstacles[idxD] > 0.1)
    {
        D = C;
        mask.z = 0;
    }
    if (_Obstacles[idxU] > 0.1)
    {
        U = C;
        mask.z = 0;
    }

    float3 v = _Velocity[idx] - float3(R - L, T - B, U - D) * 0.5;

    _Write[idx] = v * mask;
}

technique Tech0
{
    pass Pass0
    {
        ComputeShader = compile cs_5_0 Project();
    }
}
```

ComputeVorticity.fx

This shader represents the 'vorticity' component of the aforementioned 'vorticity confinement.' This is the rotational forces that smoke-like fluids generate and adds a visually pleasing element to our simulation.

Here is the ComputeVorticity.fx file:

```
//=============================================================================
// Compute Shader
//=============================================================================
#define GroupSizeXYZ 8

float4 _Size;

RWStructuredBuffer<float3> _Write;
StructuredBuffer<float3> _Velocity;

[numthreads(GroupSizeXYZ, GroupSizeXYZ, GroupSizeXYZ)]
void Vorticity(int3 id : SV_DispatchThreadID)
{
    int idxL = max(0, id.x - 1) + id.y * _Size.x + id.z * _Size.x * _Size.y;
    int idxR = min(_Size.x - 1, id.x + 1) + id.y * _Size.x + id.z * _Size.x * _Size.y;

    int idxB = id.x + max(0, id.y - 1) * _Size.x + id.z * _Size.x * _Size.y;
    int idxT = id.x + min(_Size.y - 1, id.y + 1) * _Size.x + id.z * _Size.x * _Size.y;

    int idxD = id.x + id.y * _Size.x + max(0, id.z - 1) * _Size.x * _Size.y;
    int idxU = id.x + id.y * _Size.x + min(_Size.z - 1, id.z + 1) * _Size.x * _Size.y;

    float3 L = _Velocity[idxL];
    float3 R = _Velocity[idxR];

    float3 B = _Velocity[idxB];
    float3 T = _Velocity[idxT];

    float3 D = _Velocity[idxD];
    float3 U = _Velocity[idxU];

    float3 vorticity = 0.5 * float3(((T.z - B.z) - (U.y - D.y)), ((U.x - D.x) - (R.z -
L.z)), ((R.y - L.y) - (T.x - B.x)));

    int idx = id.x + id.y * _Size.x + id.z * _Size.x * _Size.y;

    _Write[idx] = vorticity;
}

technique Tech0
{
    pass Pass0
    {
        ComputeShader = compile cs_5_0 Vorticity();
    }
}
```

Raymarcher.fx

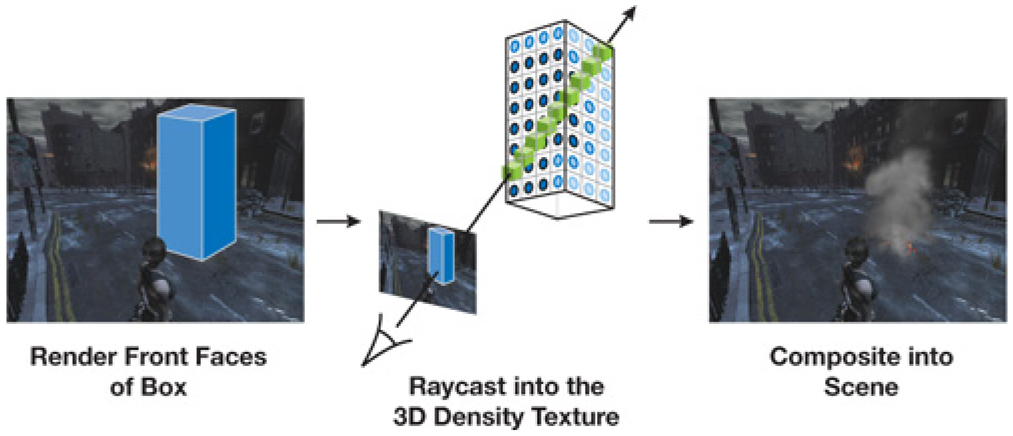
Unfortunately for us, there is no readily available way to render a three dimensional grid of values in many engines- including Monogame. A naive approach to this problem would be to render one quad for each 'slice' of the defined Eulerian grid. This works quite well actually, however from the sides of the simulation it becomes very apparent that it is a simulation as the render vanishes.

The solution to this problem is to instead render a cube (made of 6 quads) and *raymarch* through the volume for each pixel of this cube within the fragment shader. Raymarching is the technique of sending a ray through space in steps as opposed to all at once, applying some sampler function at each step.

- The ray's direction is defined as the opposite of the camera vector, or the direction from the camera to the fragment.
- The ray's start position is defined as the world space of the fragment on the box that we are rendering (if we are inside the box, it is our camera position).

This ray stops when it reaches the back of the bounding box of our fluid from our camera's position. We step an arbitrary amount of times defined by us, and at each step we do a bilinear sample of the fluid's density.

For smoke, we have a predefined *SmokeColor* parameter that determines the color of the smoke. Each step, this sample affects the alpha value of this color- meaning areas with less smoke become more see-through and areas with no smoke are completely see-through.


Render Front Faces of Box
Raycast into the 3D Density Texture
Composite into Scene

GPU Gems 3's visualization of the raymarching technique.

This raymarcher is specific to the rendering of smoke. Here is a brief description of the raymarching process for other common liquids:

- For fire, we would have to account for a time-since-lit value for realism.
- For water, we would have to account for the interface between the liquid body and air. We would also have to trilinearly sample to remove more noticeable grid artifacts.

Here is the Raymarcher.fx file:

```
#define NUM_SAMPLES 64

float4 _SmokeColor = float4(.8, 1, 1, 1);
float _SmokeAbsorption = 50;
uniform float3 _Size;

StructuredBuffer<float> _Density;

float4x4 _World;
float4x4 _View;
float4x4 _Projection;
float3 _CamPos;

struct p2f
{
    float3 Position : POSITION0;
};
struct v2f
{
    float4 pos : SV_POSITION;
    float3 worldPos : TEXCOORD0;
};

v2f VS(p2f input)
{
    v2f OUT;

    float4 worldPos = mul(float4(input.Position, 1), _World);
    float4 viewPos = mul(worldPos, _View);
    OUT.pos = mul(viewPos, _Projection);
    OUT.worldPos = worldPos.xyz;

    return OUT;
}

struct Ray
{
    float3 origin;
    float3 dir;
};

struct BoundingBox
{
    float3 Min;
    float3 Max;
};

//find intersection points of a ray with a box
bool intersectBox(Ray r, BoundingBox aabb, out float t0, out float t1)
{
    float3 invR = 1.0 / r.dir;
    float3 tbot = invR * (aabb.Min - r.origin);
    float3 ttop = invR * (aabb.Max - r.origin);
    float3 tmin = min(ttop, tbot);
    float3 tmax = max(ttop, tbot);
    float2 t = max(tmin.xx, tmin.yz);

    t0 = max(t.x, t.y);
    t = min(tmax.xx, tmax.yz);
    t1 = min(t.x, t.y);

    return t0 <= t1;
}

float SampleBilinear(StructuredBuffer<float> buffer, float3 uv, float3 size)
{
    uv = saturate(uv);
    uv = uv * (size - 1.0);

    int x = uv.x;
    int y = uv.y;
    int z = uv.z;

    int X = size.x;
    int XY = size.x * size.y;

    float fx = uv.x - x;
    float fy = uv.y - y;
    float fz = uv.z - z;

    int xp1 = min(_Size.x - 1, x + 1);
    int yp1 = min(_Size.y - 1, y + 1);
    int zp1 = min(_Size.z - 1, z + 1);

    float x0 = buffer[x + y * X + z * XY] * (1.0f - fx) + buffer[xp1 + y * X
+ z * XY] * fx;
    float x1 = buffer[x + y * X + zp1 * XY] * (1.0f - fx) + buffer[xp1 + y *
X + zp1 * XY] * fx;

    float x2 = buffer[x + yp1 * X + z * XY] * (1.0f - fx) + buffer[xp1 + yp1
* X + z * XY] * fx;
    float x3 = buffer[x + yp1 * X + zp1 * XY] * (1.0f - fx) + buffer[xp1 +
yp1 * X + zp1 * XY] * fx;

    float z0 = x0 * (1.0f - fz) + x1 * fz;
    float z1 = x2 * (1.0f - fz) + x3 * fz;

    return z0 * (1.0f - fy) + z1 * fy;
}

float4 PS(v2f IN) : COLOR
{
    float3 camPos = _CamPos;

    Ray r;
    r.origin = camPos;
    r.dir = normalize(IN.worldPos - camPos);
    BoundingBox aabb;
    aabb.Min = float3(-0.5, -0.5, -0.5);
    aabb.Max = float3(0.5, 0.5, 0.5);

      //figure out where ray from eye hit front of cube
    float tnear, tfar;
    intersectBox(r, aabb, tnear, tfar);

      //if eye is in cube then start ray at eye
    if (tnear < 0.0)
        tnear = 0.0;

    float3 rayStart = r.origin + r.dir * tnear;
    float3 rayStop = r.origin + r.dir * tfar;

    //convert to texture space
    rayStart = rayStart + 0.5;
    rayStop = rayStop + 0.5;

    float3 start = rayStart;
    float dist = distance(rayStop, rayStart);
    float stepSize = dist / float(NUM_SAMPLES);
    float3 ds = normalize(rayStop - rayStart) * stepSize;
    float alpha = 1.0;

    for (int i = 0; i < NUM_SAMPLES; i++, start += ds)
    {
        float D = SampleBilinear(_Density, start, _Size);

        alpha *= 1.0 - saturate(D * stepSize * _SmokeAbsorption);

        if (alpha <= 0.01)
            break;
    }

    return _SmokeColor * (1 - alpha);
}

technique Tech0
{
    pass Pass1
    {
        VertexShader = compile vs_4_0 VS();
        PixelShader = compile ps_4_0 PS();
    }
}
```

D. Main Program (Game1.cs)

The simulation has many parameters, and we have a few extra to manipulate the camera. Here are the required variables.

```
// needs to be the same as the GroupSizeXYX defined in the compute shader
const int ComputeGroupSizeXYZ = 8;

// Display info
const int ResolutionX = 1280;
const int ResolutionY = 720;

// Buffer constants
const int Read = 0;
const int Write = 1;
const int Phi_N_Hat = 0;
const int Phi_N_1_Hat = 1;

// Texture
const int Width = 128;
const int Height = 128;
const int Depth = 128;

// Simulation speed
const float TimeStep = 0.1f;

// Used for Jacobi compute
public int iterations = 10;

// Simulation parameters
public float vorticityStrength = 1.0f;
public float densityAmount = 1.0f;
public float densityDissipation = 0.999f;
public float densityBuoyancy = 1.0f;
public float densityWeight = 0.00125f;
public float temperatureAmount = 10.0f;
public float temperatureDissipation = 0.995f;
public float velocityDissipation = 0.995f;
public float inputRadius = 0.04f;
public Vector3 inputPos = new Vector3(0.5f, 0.1f, 0.5f);

// Compute shaders
Effect _applyAdvection;
Effect _applyImpulse;
Effect _computeBorders;
Effect _applyBuoyancy;
Effect _computeDivergence;
Effect _computeJacobi;
Effect _computeProjection;
Effect _computeVorticity;
Effect _computeConfinement;

Vector3 _size;
```

```
// Buffers
StructuredBuffer[] _density, _velocity, _pressure, _temperature, _phi;
StructuredBuffer _temp3f, _obstacles;

// Rendering
GraphicsDeviceManager _graphics;
Effect _smokeRaymarcher;
VertexBuffer _cubeVertices;
SpriteBatch _spriteBatch;
SpriteFont _textFont;
float _rotation;
Matrix _world, _view, _projection;
Vector3 _camPos;
```

The compute fork of Monogame allows you to create StructuredBuffers for use in the VRAM. Here is how you initialize them in the game program in the LoadContent function (as well as loading the other values).

```
protected override void LoadContent()
{
    _world = Matrix.Identity;
    _projection =
Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(50),
(float)ResolutionX / (float)ResolutionY, 0.1f, 1000f);

    _applyAdvection = Content.Load<Effect>("ApplyAdvection");
    _applyImpulse = Content.Load<Effect>("ApplyImpulse");
    _applyBuoyancy = Content.Load<Effect>("ApplyBuoyancy");
    _computeBorders = Content.Load<Effect>("ComputeBorders");
    _computeDivergence = Content.Load<Effect>("ComputeDivergence");
    _computeJacobi = Content.Load<Effect>("ComputeJacobi");
    _computeProjection = Content.Load<Effect>("ComputeProjection");
    _computeConfinement = Content.Load<Effect>("ComputeConfinement");
    _computeVorticity = Content.Load<Effect>("ComputeVorticity");
    _smokeRaymarcher = Content.Load<Effect>("Rendering/Raymarcher");
    _textFont = Content.Load<SpriteFont>("Text/TextFont");

    _spriteBatch = new SpriteBatch(GraphicsDevice);

    _size = new Vector3(Width, Height, Depth);

    int bufferSize = Width * Height * Depth;

    _density = new StructuredBuffer[2];
    _density[Read] = new StructuredBuffer(GraphicsDevice, typeof(float),
bufferSize, BufferUsage.None, ShaderAccess.ReadWrite);
    _density[Write] = new StructuredBuffer(GraphicsDevice, typeof(float),
bufferSize, BufferUsage.None, ShaderAccess.ReadWrite);

    _temperature = new StructuredBuffer[2];
    _temperature[Read] = new StructuredBuffer(GraphicsDevice, typeof(float),
bufferSize, BufferUsage.None, ShaderAccess.ReadWrite);
    _temperature[Write] = new StructuredBuffer(GraphicsDevice, typeof(float),
bufferSize, BufferUsage.None, ShaderAccess.ReadWrite);

    _phi = new StructuredBuffer[2];
    _phi[Read] = new StructuredBuffer(GraphicsDevice, typeof(float),
bufferSize, BufferUsage.None, ShaderAccess.ReadWrite);
    _phi[Write] = new StructuredBuffer(GraphicsDevice, typeof(float),
bufferSize, BufferUsage.None, ShaderAccess.ReadWrite);

    _velocity = new StructuredBuffer[2];
    _velocity[Read] = new StructuredBuffer(GraphicsDevice, typeof(Vector3),
bufferSize, BufferUsage.None, ShaderAccess.ReadWrite);
    _velocity[Write] = new StructuredBuffer(GraphicsDevice, typeof(Vector3),
bufferSize, BufferUsage.None, ShaderAccess.ReadWrite);

    _pressure = new StructuredBuffer[2];
    _pressure[Read] = new StructuredBuffer(GraphicsDevice, typeof(float),
bufferSize, BufferUsage.None, ShaderAccess.ReadWrite);
    _pressure[Write] = new StructuredBuffer(GraphicsDevice, typeof(float),
bufferSize, BufferUsage.None, ShaderAccess.ReadWrite);

    _obstacles = new StructuredBuffer(GraphicsDevice, typeof(float),
bufferSize, BufferUsage.None, ShaderAccess.ReadWrite);

    _temp3f = new StructuredBuffer(GraphicsDevice, typeof(Vector3),
bufferSize, BufferUsage.None, ShaderAccess.ReadWrite);

    ComputeObstacles();
}
```

*Note that we also call ComputeObstacles at the end of loading, this is where we initialize the obstacles so this step is important.

We use our Update function to update some of our shader parameter values. Here is what our Update function looks like:

```
protected override void Update(GameTime gameTime)
{
    if (Keyboard.GetState().IsKeyDown(Keys.Escape)) Exit();

    float dt_update = (float)gameTime.ElapsedGameTime.TotalSeconds;

    KeyboardState keyboardState = Keyboard.GetState();

    if (keyboardState.IsKeyDown(Keys.Right))
        _rotation += dt_update;
    if (keyboardState.IsKeyDown(Keys.Left))
        _rotation -= dt_update;

    if (keyboardState.IsKeyDown(Keys.A))
        inputPos -= new Vector3(0.01f, 0f, 0f);
    if (keyboardState.IsKeyDown(Keys.D))
        inputPos += new Vector3(0.01f, 0f, 0f);
    if (keyboardState.IsKeyDown(Keys.W))
```

```
        inputPos -= new Vector3(0f, 0f, 0.01f);
    if (keyboardState.IsKeyDown(Keys.S))
        inputPos += new Vector3(0f, 0f, 0.01f);
    if (keyboardState.IsKeyDown(Keys.Q))
        inputPos -= new Vector3(0f, 0.01f, 0f);
    if (keyboardState.IsKeyDown(Keys.E))
        inputPos += new Vector3(0f, 0.01f, 0f);

    inputPos = Vector3.Clamp(inputPos, new Vector3(0.1f), new Vector3(0.9f));

    _camPos = new Vector3((float)Math.Sin(_rotation) * 2f, 1f,
(float)Math.Cos(_rotation) * 2f);
    _view = Matrix.CreateLookAt(_camPos, Vector3.Zero, new Vector3(0, 1, 0));

    base.Update(gameTime);
}
```

To generate the cube that we raymarch through to render our fluid, we can either load a model or define vertices. This lab defines the vertices and stores them in a VertexBuffer:

```
private VertexBuffer CreateCubeVertices()
{
    var vertices = new VertexPositionTexture[36];
    Vector3 center = new Vector3(0.5f);

    // Forward face
    vertices[0] = new VertexPositionTexture(new Vector3(1, 1, 0) - center, new Vector2(1, 0));
    vertices[1] = new VertexPositionTexture(new Vector3(0, 1, 0) - center, new Vector2(0, 0));
    vertices[2] = new VertexPositionTexture(new Vector3(0, 0, 0) - center, new Vector2(0, 1));
    vertices[3] = new VertexPositionTexture(new Vector3(1, 0, 0) - center, new Vector2(1, 1));
    vertices[4] = new VertexPositionTexture(new Vector3(1, 1, 0) - center, new Vector2(1, 0));
    vertices[5] = new VertexPositionTexture(new Vector3(0, 0, 0) - center, new Vector2(0, 1));

    // Backward face
    vertices[6] = new VertexPositionTexture(new Vector3(0, 0, 1) - center, new Vector2(0, 1));
    vertices[7] = new VertexPositionTexture(new Vector3(0, 1, 1) - center, new Vector2(0, 0));
    vertices[8] = new VertexPositionTexture(new Vector3(1, 1, 1) - center, new Vector2(1, 0));
    vertices[9] = new VertexPositionTexture(new Vector3(0, 0, 1) - center, new Vector2(0, 1));
    vertices[10] = new VertexPositionTexture(new Vector3(1, 1, 1) - center, new Vector2(1, 0));
    vertices[11] = new VertexPositionTexture(new Vector3(1, 0, 1) - center, new Vector2(1, 1));

    // Top face
    vertices[12] = new VertexPositionTexture(new Vector3(1, 1, 1) - center, new Vector2(1, 0));
    vertices[13] = new VertexPositionTexture(new Vector3(0, 1, 1) - center, new Vector2(0, 0));
    vertices[14] = new VertexPositionTexture(new Vector3(0, 1, 0) - center, new Vector2(0, 1));
    vertices[15] = new VertexPositionTexture(new Vector3(1, 1, 0) - center, new Vector2(1, 1));
    vertices[16] = new VertexPositionTexture(new Vector3(1, 1, 1) - center, new Vector2(1, 0));
    vertices[17] = new VertexPositionTexture(new Vector3(0, 1, 0) - center, new Vector2(0, 1));

    // Bottom face
    vertices[18] = new VertexPositionTexture(new Vector3(0, 0, 0) - center, new Vector2(0, 1));
    vertices[19] = new VertexPositionTexture(new Vector3(0, 0, 1) - center, new Vector2(0, 0));
    vertices[20] = new VertexPositionTexture(new Vector3(1, 0, 1) - center, new Vector2(1, 0));
    vertices[21] = new VertexPositionTexture(new Vector3(0, 0, 0) - center, new Vector2(0, 1));
    vertices[22] = new VertexPositionTexture(new Vector3(1, 0, 1) - center, new Vector2(1, 0));
    vertices[23] = new VertexPositionTexture(new Vector3(1, 0, 0) - center, new Vector2(1, 1));

    // Left Face
    vertices[24] = new VertexPositionTexture(new Vector3(0, 1, 0) - center, new Vector2(1, 0));
    vertices[25] = new VertexPositionTexture(new Vector3(0, 1, 1) - center, new Vector2(0, 0));
    vertices[26] = new VertexPositionTexture(new Vector3(0, 0, 1) - center, new Vector2(0, 1));
    vertices[27] = new VertexPositionTexture(new Vector3(0, 0, 0) - center, new Vector2(1, 1));
    vertices[28] = new VertexPositionTexture(new Vector3(0, 1, 0) - center, new Vector2(1, 0));
    vertices[29] = new VertexPositionTexture(new Vector3(0, 0, 1) - center, new Vector2(0, 1));

    // Right Face
    vertices[30] = new VertexPositionTexture(new Vector3(1, 0, 1) - center, new Vector2(0, 1));
    vertices[31] = new VertexPositionTexture(new Vector3(1, 1, 1) - center, new Vector2(0, 0));
    vertices[32] = new VertexPositionTexture(new Vector3(1, 1, 0) - center, new Vector2(1, 0));
    vertices[33] = new VertexPositionTexture(new Vector3(1, 0, 1) - center, new Vector2(0, 1));
    vertices[34] = new VertexPositionTexture(new Vector3(1, 1, 0) - center, new Vector2(1, 0));
    vertices[35] = new VertexPositionTexture(new Vector3(1, 0, 0) - center, new Vector2(1, 1));

    var vb = new VertexBuffer(GraphicsDevice, typeof(VertexPositionTexture), vertices.Length,
BufferUsage.WriteOnly);
    vb.SetData(vertices);

    return vb;
}
```

Compute shaders must be dispatched from the main game file. This has a similar structure to normal shader calls, but is slightly different:

- `pass.ApplyCompute()`
- `GraphicsDevice.DispatchCompute(ComputeGroupSizeX,ComputeGroupSizeY, ComputeGroupSizeZ)`

We call these in our draw function, however due to the amount of compute shaders we must manage, the calls have been split into different functions for readability.

```
void ComputeObstacles()
{
    _computeBorders.CurrentTechnique = _computeBorders.Techniques[0];
    _computeBorders.Parameters["_Size"].SetValue(_size);
    _computeBorders.Parameters["_Write"].SetValue(_obstacles);

    foreach (var pass in _computeBorders.CurrentTechnique.Passes)
    {
        pass.ApplyCompute();
        GraphicsDevice.DispatchCompute((int)_size.X / ComputeGroupSizeXYZ,
(int)_size.Y / ComputeGroupSizeXYZ, (int)_size.Z / ComputeGroupSizeXYZ);
    }
}

void ApplyAdvection(float dt, float dissipation, StructuredBuffer[] buffer,
float forward = 1.0f)
{
    _applyAdvection.CurrentTechnique = _applyAdvection.Techniques[1];
    _applyAdvection.Parameters["_Size"].SetValue(_size);
    _applyAdvection.Parameters["_DeltaTime"].SetValue(dt);
    _applyAdvection.Parameters["_Dissipate"].SetValue(dissipation);
    _applyAdvection.Parameters["_Forward"].SetValue(forward);
    _applyAdvection.Parameters["_Read1f"].SetValue(buffer[Read]);
    _applyAdvection.Parameters["_Write1f"].SetValue(buffer[Write]);
    _applyAdvection.Parameters["_Velocity"].SetValue(_velocity[Read]);
    _applyAdvection.Parameters["_Obstacles"].SetValue(_obstacles);

    foreach (var pass in _applyAdvection.CurrentTechnique.Passes)
    {
        pass.ApplyCompute();
        GraphicsDevice.DispatchCompute((int)_size.X / ComputeGroupSizeXYZ,
(int)_size.Y / ComputeGroupSizeXYZ, (int)_size.Z / ComputeGroupSizeXYZ);
    }

    Swap(buffer);
}

void ApplyAdvection(float dt, float dissipation, StructuredBuffer read,
StructuredBuffer write, float forward = 0.1f)
{
    _applyAdvection.CurrentTechnique = _applyAdvection.Techniques[1];
    _applyAdvection.Parameters["_Size"].SetValue(_size);
    _applyAdvection.Parameters["_DeltaTime"].SetValue(dt);
    _applyAdvection.Parameters["_Dissipate"].SetValue(dissipation);
    _applyAdvection.Parameters["_Forward"].SetValue(forward);
    _applyAdvection.Parameters["_Read1f"].SetValue(read);
    _applyAdvection.Parameters["_Write1f"].SetValue(write);
    _applyAdvection.Parameters["_Velocity"].SetValue(_velocity[Read]);
    _applyAdvection.Parameters["_Obstacles"].SetValue(_obstacles);

    foreach (var pass in _applyAdvection.CurrentTechnique.Passes)
    {
        pass.ApplyCompute();
        GraphicsDevice.DispatchCompute((int)_size.X / ComputeGroupSizeXYZ,
(int)_size.Y / ComputeGroupSizeXYZ, (int)_size.Z / ComputeGroupSizeXYZ);
    }
}

void ApplyAdvectionMacCormack(float dt, float dissipation, StructuredBuffer[]
buffer)
{
    _applyAdvection.CurrentTechnique = _applyAdvection.Techniques[2];
    _applyAdvection.Parameters["_Size"].SetValue(_size);
    _applyAdvection.Parameters["_DeltaTime"].SetValue(dt);
    _applyAdvection.Parameters["_Dissipate"].SetValue(dissipation);
    _applyAdvection.Parameters["_Forward"].SetValue(1.0f);
    _applyAdvection.Parameters["_Read1f"].SetValue(buffer[Read]);
    _applyAdvection.Parameters["_Write1f"].SetValue(buffer[Write]);
    _applyAdvection.Parameters["_Phi_n_1_hat"].SetValue(_phi[Phi_N_1_Hat]);
    _applyAdvection.Parameters["_Phi_n_hat"].SetValue(_phi[Phi_N_Hat]);
    _applyAdvection.Parameters["_Velocity"].SetValue(_velocity[Read]);
    _applyAdvection.Parameters["_Obstacles"].SetValue(_obstacles);

    foreach (var pass in _applyAdvection.CurrentTechnique.Passes)
    {
        pass.ApplyCompute();
        GraphicsDevice.DispatchCompute((int)_size.X / ComputeGroupSizeXYZ,
(int)_size.Y / ComputeGroupSizeXYZ, (int)_size.Z / ComputeGroupSizeXYZ);
    }

    Swap(buffer);
}

void ApplyAdvectionVelocity(float dt)
{
    _applyAdvection.CurrentTechnique = _applyAdvection.Techniques[0];
    _applyAdvection.Parameters["_Size"].SetValue(_size);
    _applyAdvection.Parameters["_DeltaTime"].SetValue(dt);
    _applyAdvection.Parameters["_Dissipate"].SetValue(velocityDissipation);
    _applyAdvection.Parameters["_Forward"].SetValue(1.0f);
    _applyAdvection.Parameters["_Read3f"].SetValue(_velocity[Read]);
    _applyAdvection.Parameters["_Write3f"].SetValue(_velocity[Write]);
    _applyAdvection.Parameters["_Velocity"].SetValue(_velocity[Read]);
    _applyAdvection.Parameters["_Obstacles"].SetValue(_obstacles);

    foreach (var pass in _applyAdvection.CurrentTechnique.Passes)
    {
        pass.ApplyCompute();
        GraphicsDevice.DispatchCompute((int)_size.X / ComputeGroupSizeXYZ,
(int)_size.Y / ComputeGroupSizeXYZ, (int)_size.Z / ComputeGroupSizeXYZ);
    }

    Swap(_velocity);
}

void ApplyBuoyancy(float dt)
{
    _applyBuoyancy.CurrentTechnique = _applyBuoyancy.Techniques[0];
    _applyBuoyancy.Parameters["_Size"].SetValue(_size);
    _applyBuoyancy.Parameters["_Up"].SetValue(new Vector3(0f, 1f, 0f));
    _applyBuoyancy.Parameters["_Buoyancy"].SetValue(densityBuoyancy);
    _applyBuoyancy.Parameters["_Weight"].SetValue(densityWeight);
    _applyBuoyancy.Parameters["_DeltaTime"].SetValue(dt);
    _applyBuoyancy.Parameters["_Write"].SetValue(_velocity[Write]);
    _applyBuoyancy.Parameters["_Velocity"].SetValue(_velocity[Read]);
    _applyBuoyancy.Parameters["_Density"].SetValue(_density[Read]);
    _applyBuoyancy.Parameters["_Temperature"].SetValue(_temperature[Read]);

    foreach (var pass in _applyBuoyancy.CurrentTechnique.Passes)
    {
        pass.ApplyCompute();
        GraphicsDevice.DispatchCompute((int)_size.X / ComputeGroupSizeXYZ,
(int)_size.Y / ComputeGroupSizeXYZ, (int)_size.Z / ComputeGroupSizeXYZ);
    }

    Swap(_velocity);
}

void ApplyImpulse(float dt, float amount, StructuredBuffer[] buffer)
{
    _applyImpulse.CurrentTechnique = _applyImpulse.Techniques[0];
    _applyImpulse.Parameters["_Size"].SetValue(_size);
    _applyImpulse.Parameters["_Radius"].SetValue(inputRadius);
    _applyImpulse.Parameters["_Amount"].SetValue(amount);
    _applyImpulse.Parameters["_DeltaTime"].SetValue(dt);
    _applyImpulse.Parameters["_Pos"].SetValue(inputPos);
    _applyImpulse.Parameters["_Read"].SetValue(buffer[Read]);
    _applyImpulse.Parameters["_Write"].SetValue(buffer[Write]);

    foreach (var pass in _applyImpulse.CurrentTechnique.Passes)
    {
        pass.ApplyCompute();
        GraphicsDevice.DispatchCompute((int)_size.X / ComputeGroupSizeXYZ, 
(int)_size.Y / ComputeGroupSizeXYZ, (int)_size.Z / ComputeGroupSizeXYZ);
    }

    Swap(buffer);
}

void ComputeVorticityConfinement(float dt)
{
    _computeVorticity.CurrentTechnique = _computeVorticity.Techniques[0];
    _computeVorticity.Parameters["_Size"].SetValue(_size);
    _computeVorticity.Parameters["_Write"].SetValue(_temp3f);
    _computeVorticity.Parameters["_Velocity"].SetValue(_velocity[Read]);
    foreach (var pass in _computeVorticity.CurrentTechnique.Passes)
    {
        pass.ApplyCompute();
        GraphicsDevice.DispatchCompute((int)_size.X / ComputeGroupSizeXYZ, 
(int)_size.Y / ComputeGroupSizeXYZ, (int)_size.Z / ComputeGroupSizeXYZ);
    }

    _computeConfinement.CurrentTechnique = _computeConfinement.Techniques[0];
    _computeConfinement.Parameters["_Size"].SetValue(_size);
    _computeConfinement.Parameters["_DeltaTime"].SetValue(dt);
    _computeConfinement.Parameters["_Epsilon"].SetValue(vorticityStrength);
    _computeConfinement.Parameters["_Write"].SetValue(_velocity[Write]);
    _computeConfinement.Parameters["_Read"].SetValue(_velocity[Read]);
    _computeConfinement.Parameters["_Vorticity"].SetValue(_temp3f);
    foreach (var pass in _computeConfinement.CurrentTechnique.Passes)
    {
        pass.ApplyCompute();
        GraphicsDevice.DispatchCompute((int)_size.X / ComputeGroupSizeXYZ, 
(int)_size.Y / ComputeGroupSizeXYZ, (int)_size.Z / ComputeGroupSizeXYZ);
    }

    Swap(_velocity);
}

void ComputeDivergence()
{
    _computeDivergence.CurrentTechnique = _computeDivergence.Techniques[0];
    _computeDivergence.Parameters["_Size"].SetValue(_size);
    _computeDivergence.Parameters["_Write"].SetValue(_temp3f);
    _computeDivergence.Parameters["_Velocity"].SetValue(_velocity[Read]);
    _computeDivergence.Parameters["_Obstacles"].SetValue(_obstacles);

    foreach (var pass in _computeDivergence.CurrentTechnique.Passes)
    {
        pass.ApplyCompute();
        GraphicsDevice.DispatchCompute((int)_size.X / ComputeGroupSizeXYZ, 
(int)_size.Y / ComputeGroupSizeXYZ, (int)_size.Z / ComputeGroupSizeXYZ);
    }
}
```

```
void ComputePressure()
{
    _computeJacobi.CurrentTechnique = _computeJacobi.Techniques[0];
    _computeJacobi.Parameters["_Size"].SetValue(_size);
    _computeJacobi.Parameters["_Divergence"].SetValue(_temp3f);
    _computeJacobi.Parameters["_Obstacles"].SetValue(_obstacles);

    for (int i = 0; i < iterations; i++)
    {
        foreach (var pass in _computeJacobi.CurrentTechnique.Passes)
        {
            _computeJacobi.Parameters["_Write"].SetValue(_pressure[Write]);
            _computeJacobi.Parameters["_Pressure"].SetValue(_pressure[Read]);
            pass.ApplyCompute();
            GraphicsDevice.DispatchCompute((int)_size.X /
ComputeGroupSizeXYZ, (int)_size.Y / ComputeGroupSizeXYZ, (int)_size.Z /
ComputeGroupSizeXYZ);
            Swap(_pressure);
        }
    }
}

void ComputeProjection()
{
    _computeProjection.CurrentTechnique = _computeProjection.Techniques[0];
    _computeProjection.Parameters["_Size"].SetValue(_size);
    _computeProjection.Parameters["_Obstacles"].SetValue(_obstacles);
    _computeProjection.Parameters["_Pressure"].SetValue(_pressure[Read]);
    _computeProjection.Parameters["_Velocity"].SetValue(_velocity[Read]);
    _computeProjection.Parameters["_Write"].SetValue(_velocity[Write]);

    foreach (var pass in _computeProjection.CurrentTechnique.Passes)
    {
        pass.ApplyCompute();
        GraphicsDevice.DispatchCompute((int)_size.X / ComputeGroupSizeXYZ,
(int)_size.Y / ComputeGroupSizeXYZ, (int)_size.Z / ComputeGroupSizeXYZ);
    }

    Swap(_velocity);
}

void Swap(StructuredBuffer[] buffer)
{
    StructuredBuffer tmp = buffer[Read];
    buffer[Read] = buffer[Write];
    buffer[Write] = tmp;
}
```

Our raymarcher also has its own function defined as follows:

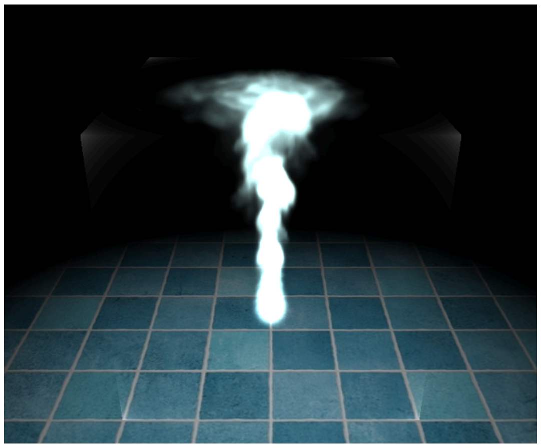
```
private void DrawFluidRaymarched()
{
    _smokeRaymarcher.Parameters["_World"].SetValue(_world);
    _smokeRaymarcher.Parameters["_View"].SetValue(_view);
    _smokeRaymarcher.Parameters["_Projection"].SetValue(_projection);
    _smokeRaymarcher.Parameters["_Density"].SetValue(_density[Read]);
    _smokeRaymarcher.Parameters["_Size"].SetValue(_size);
    _smokeRaymarcher.Parameters["_CamPos"].SetValue(_camPos);

    foreach (var pass in _smokeRaymarcher.CurrentTechnique.Passes)
    {
        pass.Apply();

        GraphicsDevice.SetVertexBuffer(_cubeVertices);
        GraphicsDevice.DrawPrimitives(PrimitiveType.TriangleList, 0, 12);
    }
}
```

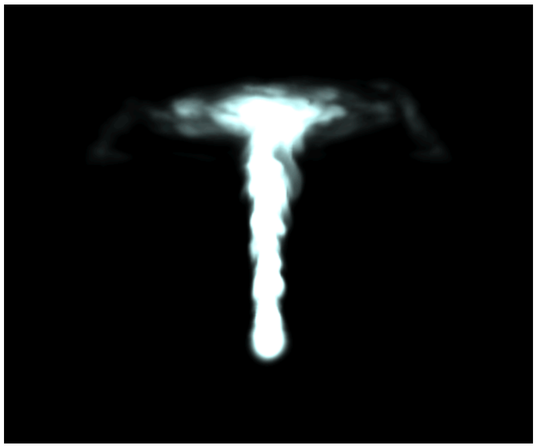
Now that all of our functions we use in our draw function are defined, we can define our draw function. The order of compute operations follows this format:

```
protected override void Draw(GameTime gameTime)
{
    float dt = TimeStep;

    ApplyAdvection(dt, temperatureDissipation, _temperature);

    // Set to true for MacCormack advection
    if (false)
    {
        ApplyAdvection(dt, 1.0f, _density[Read], _phi[Phi_N_1_Hat], 1.0f);
        ApplyAdvection(dt, 1.0f, _density[Phi_N_1_Hat], _phi[Phi_N_Hat], -1.0f);
        ApplyAdvectionMacCormack(dt, densityDissipation, _density);
    }
    else ApplyAdvection(dt, densityDissipation, _density);

    ApplyAdvectionVelocity(dt);
    ApplyBuoyancy(dt);

    ApplyImpulse(dt, densityAmount, _density);
    ApplyImpulse(dt, temperatureAmount, _temperature);

    ComputeVorticityConfinement(dt);

    ComputeDivergence();
    ComputePressure();
    ComputeProjection();

    GraphicsDevice.Clear(Color.Black);

    GraphicsDevice.RasterizerState = RasterizerState.CullNone;
    GraphicsDevice.BlendState = BlendState.Additive;

    DrawFluidRaymarched();

    base.Draw(gameTime);
}
```

E. Main Exercise

If you have managed to implement all 10 shaders and integrate them with the game file within these 34 pages, then that alone is grounds for a main exercise. Congratulations.


Smoke sim with glass and tiles


Smoke sim by itself

Both of these sims use a SmokeColor value of (.8, 1, 1, 1).

***** IMPORTANT *****
Complete the exercise in the E section, and submit a zipped file including the solution (.sln) file and the project folders to the course online site. The submission item is located in the "**Quiz and Lab**" section. Each lab has **10 points**. If you complete the exercise in class time, the full points will be assigned. The late submission is accepted just before the next class with 2 points reductions, because the solution is demonstrated in the next class.