FluidCS11

<https://github.com/walbourn/directx-sdk-samples>

This is the DirectX SDK's Direct3D 11 sample updated to use Visual Studio 2012 and the Windows SDK 8.0 without any dependencies on legacy DirectX SDK content. This sample is a Win32 desktop DirectX 11.0 application for Windows 10, Windows 8.1, Windows 8, Windows 7, and Windows Vista Service Pack 2 with the DirectX 11.0 runtime.

**This is based on the legacy DirectX SDK (June 2010) Win32 desktop sample. This is not intended for use with Windows Store apps, Windows RT, or universal Windows apps.**

# Description



This sample illustrates how to implement a simple optimized 2D particle fluid simulation. It uses a technique called Smoothed Particle Hydrodynamics to perform the fluid simulation based on the paper “Particle-Based Fluid Simulation for Interactive Applications” by Matthias Müller. To further optimize the simulation for the GPU, a spatial grid is implemented based on the article “Broad-Phase Collision Detection with CUDA” by Scott Le Grand.

## Fluid Simulation

Smoothed Particle Hydrodynamics is a simple technique for simulating fluid flows using particles. The basic idea is to use the neighboring particles within a search radius to determine properties at the current location using a smoothing kernel. A particle which is closer to the current location will have a greater influence than a particle which is near to the search radius’s edge. For a full explanation of this algorithm and the equations, please see the paper “Particle-Based Fluid Simulation for Interactive Applications” by Matthias Müller.

## Simple N2 Algorithm and Optimized Shared Memory N2 Algorithm

The easiest way to perform the simulation is with a simple N2 search. The compute shader will execute one thread per particle and iterate over all other particles looking for neighbors within the desired search radius. However, this technique is extremely expensive since we only care about particles within a relatively small neighboring area.

We can further optimize this by utilizing shared memory to reduce the required memory bandwidth. Each thread within the thread group will read one potential neighbor into shared memory, and then all of the threads can access the neighbors from the common shared memory. Please see the NBodyGravityCS11 sample for more information on both of these techniques. This helps improve the performance greatly, but this is still only useful for a small number of particles.

## Optimized Spatial Grid

Because each particle only needs to intact with its neighbors within a small radius, a full N2 search of all neighbors is excessive. If we can limit the work to only comparing nearby neighboring particles, then we will greatly reduce the required memory bandwidth. Since our search radius has a maximum size, we can do this by dividing the world into an even grid that is the same size as the search radius. Our search will be limited to just the occupants of the current particle’s grid cell plus the occupants of the neighboring eight cells.

### Calculate the Grid Cell ID

The first step is to determine which grid cell each particle is in. Particles that occupy the same grid cell will share the same grid cell ID. A very simple combination of the cell’s X and Y position can be used to uniquely identify each cell. This information along with the particle’s ID will be stored in a structured buffer for the next step.

### Sort the Grid IDs

Next we need to sort the structured buffer of grid cell IDs and particle IDs. This will have the effect of organizing all of the particles that occupy the same cell next to each other in memory. For this sample, we used the bitonic sort from the CompuerShaderSort11 sample.

### Find the Grid Indices

We have constructed a sorted list of every particle and the cell ID it occupies, but we do not have a way to index that information based on a cell ID yet. The next step will search the list for the start and end location of each cell and store that information. To do this, we execute one thread per element of the list. If the element to the left in the list has a different grid cell ID, then this must be the first occupant in the cell. Recall that we sorted this data in the previous step. Similarly, if the element to the right has a different ID, then we are the last occupant. By recording the start and end index for each cell ID we can easily lookup all of the particles in a given cell in the next step.

### Perform the Simulation

Lastly, we must perform the simulation. The core of the simulation is unchanged; however, this time we need only search for potential neighbors within the adjacent cells. All we need to do is calculate our grid cell ID and the cell IDs of the neighboring cells. Using this we can look up the particles in those cells by looking at the indices built in the previous step. Then we check if the particle is within the search radius and perform the simulation same as before.

# Dependencies

DXUT-based samples typically make use of runtime HLSL compilation. Build-time compilation is recommended for all production Direct3D applications, but for experimentation and samples development runtime HLSL compilation is preferred. Therefore, the D3DCompile\*.DLL must be available in the search path when these programs are executed.

* When using the Windows 8.x SDK and targeting Windows Vista or later, you can include the D3DCompile\_46 or D3DCompile\_47 DLL side-by-side with your application copying the file from the REDIST folder.

%ProgramFiles(x86)%\Windows kits\8.0\Redist\D3D\arm, x86 or x64

%ProgramFiles(x86)%\Windows kits\8.1\Redist\D3D\arm, x86 or x64

%ProgramFiles(x86)%\Windows kits\10\Redist\D3D\arm, x86 or x64

# More Information

[Where is the DirectX SDK (2015 Edition)?](https://walbourn.github.io/where-is-the-directx-sdk-2015-edition/)

[DXUT for Win32 Desktop Update](https://walbourn.github.io/dxut-for-win32-desktop-update/)

[Games for Windows and DirectX SDK blog](https://walbourn.github.io/)