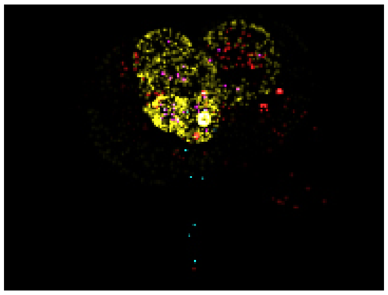
ParticlesGS Sample

⊟ Collapse All

This sample implements a complete particle system on the GPU using the Direct3D 10 Geometry Shader, Stream Output, and DrawAuto.



Path

Source	*SDK root*\Samples\C++\Direct3D10\ParticlesGS
Executable	*SDK root*\Samples\C++\Direct3D10\Bin\x86 or x64\ParticlesGS.exe

How the Sample Works

Particle system computation has traditionally been performed on the CPU with the GPU rendering the particles as point sprites for visualization. With Geometry Shaders, the ability to output arbitrary amounts of data to a stream allows the GPU to create new geometry and to store the result of computations on existing geometry.

This sample uses the stream out capabilities of the geometry shader to store the results of particles calculations into a buffer. Additionally, the Geometry Shader controls particle birth and death by either outputting new geometry to the stream or by avoiding writing existing geometry to the stream. A Geometry Shader that streams output to a buffer must be constructed differently from a normal Geometry Shader.

When used inside an FX file

```
//--------------------------------------------------------------------------------------
// Construct StreamOut Geometry Shader
//--------------------------------------------------------------------------------------
geometryshader gsStreamOut = ConstructGSWithSO(compile gs_4_0 GSAdvanceParticlesMain(),
                                               "POSITION.xyz;
                                               NORMAL.xyz;
                                               TIMER.x;
                                               TYPE.x" );
```

When used without FX

```
//--------------------------------------------------------------------------------------
// Construct StreamOut Geometry Shader
//--------------------------------------------------------------------------------------
D3D10_STREAM_OUTPUT_DECLARATION_ENTRY pDecl[] =
{
        // semantic name, semantic index, start component, component count, output slot
        { L"POSITION", 0, 0, 3, 0 }, // output first 3 components of "POSITION"
        { L"NORMAL", 0, 0, 3, 0 }, // output the first 3 components of "NORMAL"
        { L"TIMER", 0, 0, 1, 0 }, //  output the first component of "TIMER"
        { L"TYPE", 0, 0, 1, 0 }, //  output the first component of "TYPE"
};

CreateGeometryShaderWithStreamOut( pShaderData, pDecl, 4, sizeof(PARTICLE_VERTEX), &pGS );
```

Particle Types

This particle system is composed of 5 different particle types with varying properties. Each particle type has its own velocity and behavior and may or may not emit other particles.

Launcher Particles

Launcher particles do not move and do not die. They simply count down until they can emit a Shell particle. Once they have emitted a Shell particle, they reset their timer.

Shell Particles

Shell particles are single particles that are given random velocities and launched into the air by Launcher particles. They are meant to represent fireworks shells before they explode. When a Shell particle reaches the end of it's lifespan it does not re-emit itself into the system. Instead, it emits several Ember1 and Ember2 type particles.

Ember1 Particles

Ember1 particles are emitted from Shell particles when they explode. Ember1 particles have short lifespans and fade out as their timer counts down. When their timer reaches zero, these particles do not re-emit themselves into the system and effectively "die."

Ember2 Particles

Ember2 particles are also emitted from Shell particles when they explode. Unlike Ember1 particles, when Ember2 particles reach the end of their lifespans they emit Ember3 particles. These are the source of the secondary explosions in the system.

Ember3 Particles

Ember3 particles are similar to Ember1 particles except that they are of a different color and have a shorter lifespan.

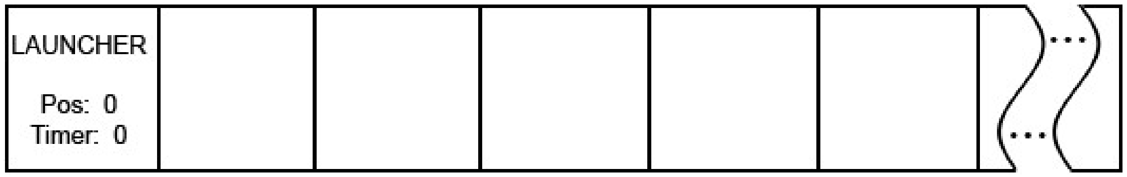
Handling Particles in the Geometry Shader

Particles are handled entirely on the GPU by the Geometry Shader. Instead of going to the rasterizer, the vertex data passed into the geometry shader is output to another vertex buffer. After an initial seeding of the vertex buffer with LAUNCHER type particles, the system can sustain itself on the GPU with only per-frame timing information coming from the CPU.

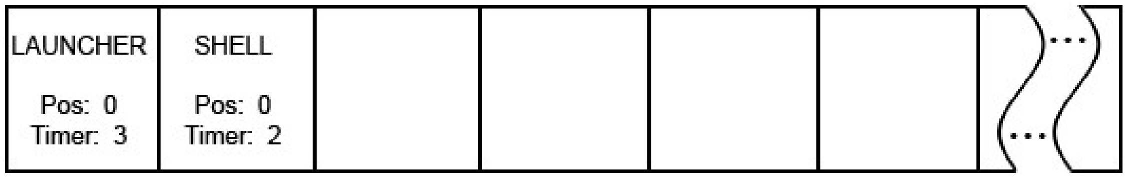
The sample uses 3 buffers consisting of vertex data to facilitate a fireworks particle system. The first stream contains the initial particles needed to "seed" the system. It is used only once during the first frame that the particle system is active. The second and third buffers are ping-pong buffers and trade off being streamed to and rendered from every other frame.

The particle system works in the following manner:

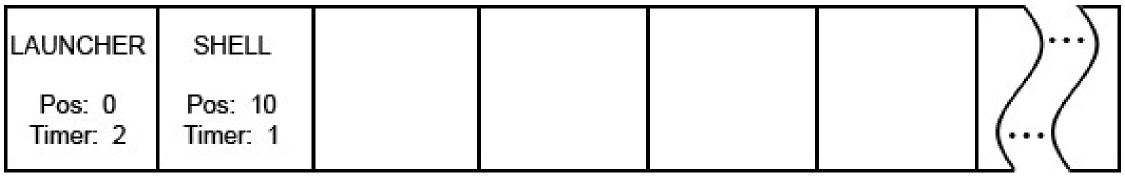
1. The seed buffer is filled with an initial launcher particle



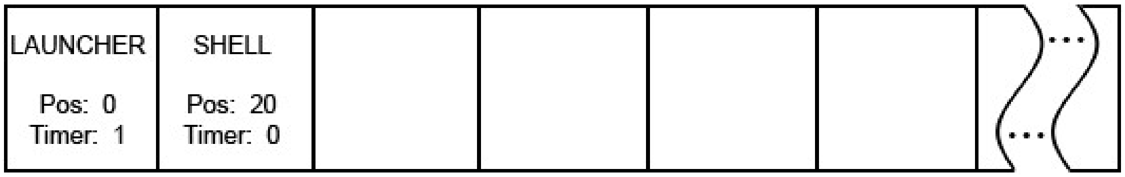
2. The first time through the GS, the GS sees that the LAUNCHER is at 0 and emits a SHELL at the launcher position. NOTE: Because the particle system is rebuilt every pass through the GS, any particles that are necessary for the next frame need to be emitted, not just new particles.



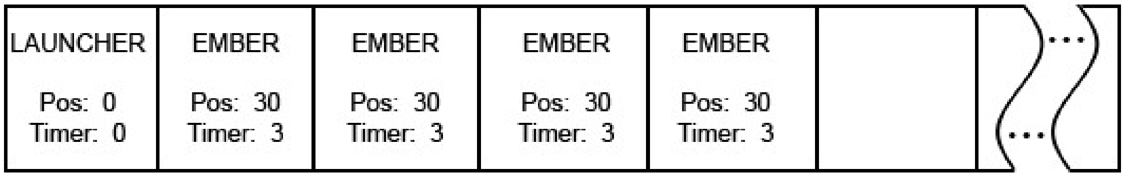
3. The second time through the GS, the LAUNCHER and SHELL timers are decremented and the SHELL is moved to a new position.



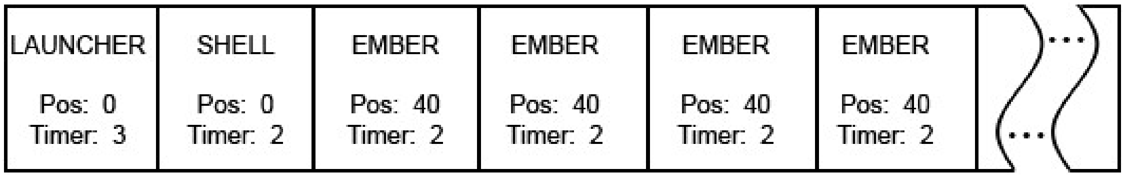
4. The SHELL timer is decremented to 0, which means that this is the last frame for this SHELL.



5. Because its timer is at 0, the SHELL is not emitted again. In its place, 4 EMBER particles are emitted.



6. The LAUNCHER is at zero, and therefore must emit another SHELL particle. The EMBER particles are moved to a new position and have their timers decremented as well. The LAUNCHER timer is reset.



Knowing How Many Particles Were Output

Geometry Shaders can emit a variable amount of data each frame. Because of this, the sample has no way of knowing how many particles are in the buffer at any given time. Using standard Draw calls, the sample would have to guess at the number of particles to tell the GPU to draw. Fortunately, DrawAuto is designed to handle this situation. DrawAuto allows the dynamic amount of data written to streamout buffer to be used as the input amount of data for the draw call. Because this happens on the GPU, the CPU can advance and draw the particle system with no knowledge of how many particles actually comprise the system.

Rendering Particles

After the particles are advanced by the gsStreamOut Geometry Shader, the buffer that just received the output is used in a second pass for rendering the particles as point sprites. The VSSceneMain Vertex Shader takes care of assigning size and color to the particles based upon their type and age. GSSceneMain constructs point sprites from the points by emitting a 2 triangle strip for every point that is passed in. In this pass, the Geometry Shader output is passed to the rasterizer and does not stream out to any buffer.

© 2010 Microsoft Corporation. All rights reserved.
Send feedback to DxSdkDoc@microsoft.com.
Version: 1962.00