Rain

Sarah Tariq
stariq@nvidia.com

Month 2007

Document Change History

Version	Date	Responsible	Reason for Change
			Initial release

Abstract

Atmospheric effects such as rain and snow are important in creating realistic immersive environments and setting a mood in story telling. Rendering these effects realistically however is a hard problem, especially in real time.

This sample presents a particle system approach for animating and rendering rain streaks that works entirely on the GPU, using features that have not existed before Direct3D10. Rain particles are animated over time using *Stream Out*, and at each frame they are expanded into billboards to be rendered using the *Geometry Shader*. Finally, the rendering of the rain particles uses a library of textures stored in a *Texture Array*, which encodes the appearance of different rain drops under different viewpoint and lighting directions.

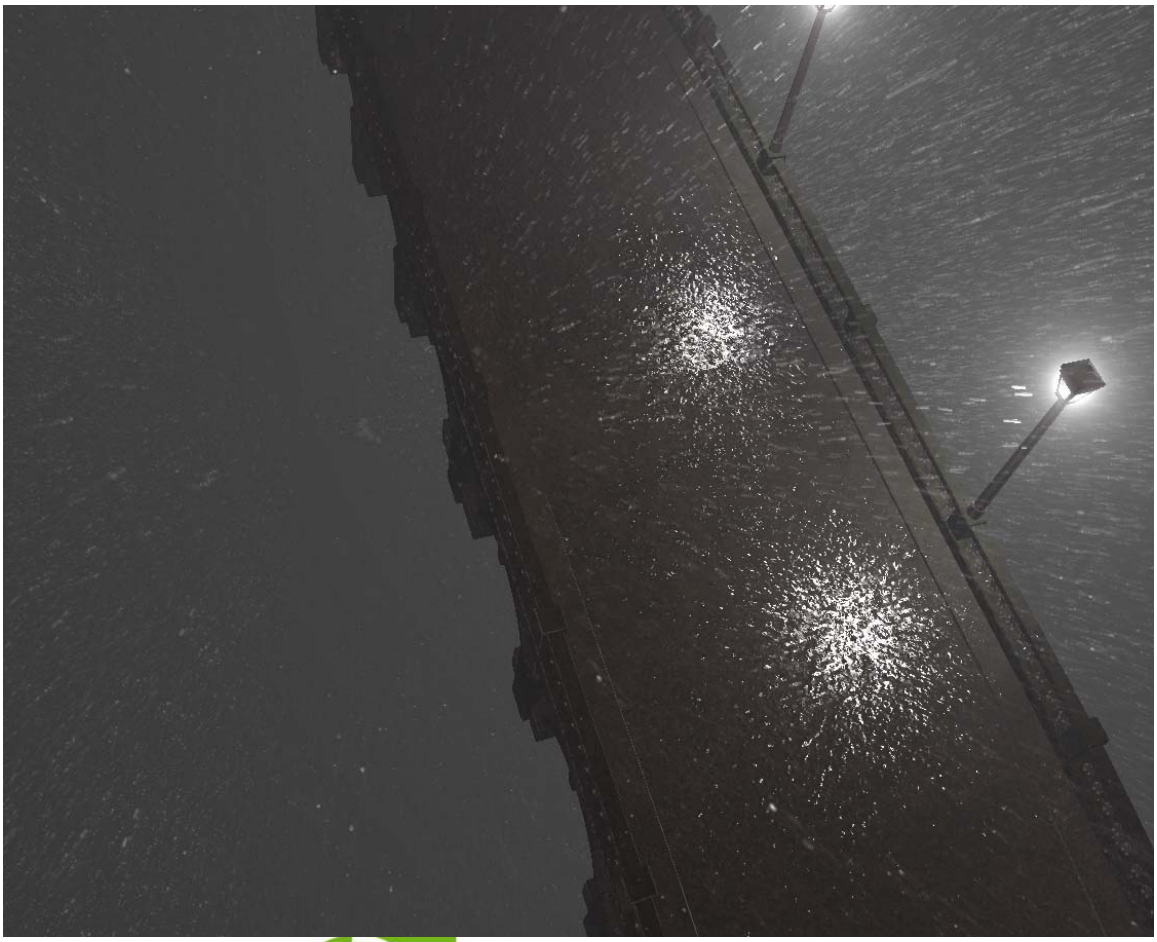

Figure 1.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com

Motivation

Rain is traditionally rendered in one of two ways; either as a particle system or as camera-centered geometry with scrolling textures.

The technique using screen-centered geometry mapped with scrolling textures is commonly used in real time applications; [Tatarchuk06] use a screen aligned quad while [Wang04] use a camera centered cone. These methods are fast but the results can look like they lack depth. It is also hard for these methods to exhibit complex dynamics like stormy wind, or respond to local lighting like street lights in the scene.

Rain can also be animated as a particle system, but in the past this approach has been considered slow for real time applications [Wang04], especially for scenes depicting heavy rainfall. Rendering rain using a particle system requires three steps:

- Animate the particles over time, applying forces such as wind and gravity
- Expand the particles into sprites to be rendered at each frame
- Render the sprites

Traditionally the animation of particles and expansion of the sprites would either be done on the CPU, or would have to be mapped to the GPU using methods like render to texture combined with vertex texture fetch. With the advent of Direct3D10 and the NVIDIA GeForce 8-series GPUs, this animation and sprite creation can now be mapped intuitively and efficiently to the GPU.

How Does It Work?

Animating Rain

Rain is animated using two vertex buffers and Stream Out. Each vertex encodes a particle. At each frame we bind one vertex buffer as input, animate the vertices in the vertex shader, and then stream them out to the other vertex buffer. At the end of the frame the buffers are swapped; see code listings 1 and 2.

```
// Setup to render a list of points; each particle is
// stored in a vertex
pd3dDevice->IASetPrimitiveTopology(
                 D3D10_PRIMITIVE_TOPOLOGY_POINTLIST );
pd3dDevice->IASetInputLayout( g_pVertexLayoutRainVertex );

// Decided which vertex buffer we are going to render from
// If this is the first frame we render from a prepopulated
// Vertex Buffer, g_pParticlesStart.
static bool firstFrame = true;
ID3D10Buffer *pBuffers[1];
if(firstFrame)
    pBuffers[0] = g_pParticleStart;
else
```

```
    pBuffers[0] = g_pParticleDrawFrom;

pd3dDevice->IASetVertexBuffers( 0, 1, pBuffers, stride,
offset );

// Point to the correct output buffer
pBuffers[0] = g_pParticleStreamTo;
pd3dDevice->SOSetTargets( 1, pBuffers, offset );

// draw: this technique is going to animate the particles
D3D10_TECHNIQUE_DESC techDesc;
g_pTechniqueAdvanceRain->GetDesc( &techDesc );
g_pTechniqueAdvanceRain->GetPassByIndex(0)->Apply(0);
pd3dDevice->Draw(g_numRainVertices , 0 );

// Get back to normal
pBuffers[0] = NULL;
pd3dDevice->SOSetTargets( 1, pBuffers, offset );

// Swap buffers
ID3D10Buffer* pTemp = g_pParticleDrawFrom;
g_pParticleDrawFrom = g_pParticleStreamTo;
g_pParticleStreamTo = pTemp;

firstFrame = false;
```

Code Listing 1. C++ code using Stream Out

```
// Construct Geometry shader with Stream Out from the
Vertex Shader
GeometryShader gsStreamOut = ConstructGSWithSO(
CompileShader( vs_4_0, VSAdvanceRain() ), "POSITION.xyz;
RAND.x; TYPE.x  " );

// The technique to animate particles uses just a Vertex
// Shader and Stream Out
technique10 AdvanceParticles
{
  pass p0
  {
   SetVertexShader(CompileShader(vs_4_0, VSAdvanceRain()));
   SetGeometryShader( gsStreamOut );
   SetPixelShader( NULL );

   SetDepthStencilState( DisableDepth, 0 );
  }
}
```

Code Listing 2. HLSL code using Stream Out for animating particles

Rendering Rain

Expanding points into sprites using the Geometry shader

At each frame, after the particles are animated, they are expanded into sprites on the GPU by using the geometry shader.

We bind the vertex buffer containing the particles to the GPU and then the geometry shader in code listing 3 is use to expand each of these into a sprite. This shader takes as input a single vertex, and outputs four vertices of the expanded sprite (two triangles in a triangle list). These vertices are *appended* to the triangle stream that was passed to the geometry shader.

```
// GS for rendering rain as point sprites.  Takes a point
and turns it into 2 triangles.
[maxvertexcount(4)]
void GSRenderRain(point VSParticleIn input[1], inout
TriangleStream<PSSceneIn> SpriteStream)
{
   PSSceneIn output = (PSSceneIn)0;

   float3 worldPos = mul( input[0].pos, g_mWorld );
   output.type = input[0].Type;
   output.random = input[0].random;

   float3 pos[4];
   // Calculate the four vertices of the sprite
   ...

   // construct the first vertex
   output.pos = mul( float4(pos[0],1.0),g_mWorldViewProj );
   output.lightDir = g_lightPos - pos[0];
   output.pointLightDir = g_PointLightPos - pos[0];
   output.eyeVec = g_eyePos - pos[0];
   output.tex = g_texcoords[0];
   // append the vertex to the stream
   SpriteStream.Append(output);

   // construct and append the other three points similar
   // to the first
   ...
   //
   // restart the triangle strip
   SpriteStream.RestartStrip();
}
```

Code Listing 3. HLSL code using the Geometry Shader for extruding points to sprites

Rendering the sprites

Rendering rain streaks is a complicated task; rain drops refract and reflect light and undergo quick deformations as they fall. When these drops are motion blurred into rain streaks by our visual systems the results are images exhibiting complex brightness patterns [Garg06]. Traditionally however, rain streaks have either been rendered as streaks of constant brightness, or using textures created by artists. Both these approaches cannot respond realistically to changes in viewing or lighting directions. [Garg06] presented a method of pre-calculating the appearance of different rain drops under varying viewpoint and illumination directions. These textures, available from http://www1.cs.columbia.edu/CAVE/databases/rain_streak_db/rain_streak.php, can be used to render raindrops under any given viewpoint and illumination direction. Other variables, such as rain velocity, and camera exposure time, can also be adjusted.

This texture database contains textures indexed by three different angles, Figure 2. θ_{view} is the angle from the view vector to its projection onto the plane perpendicular to the direction of the rain particle, θ_{light} is the angle from light vector to its projection, and ϕ_{light} the angle between the projections of the view and light vectors.

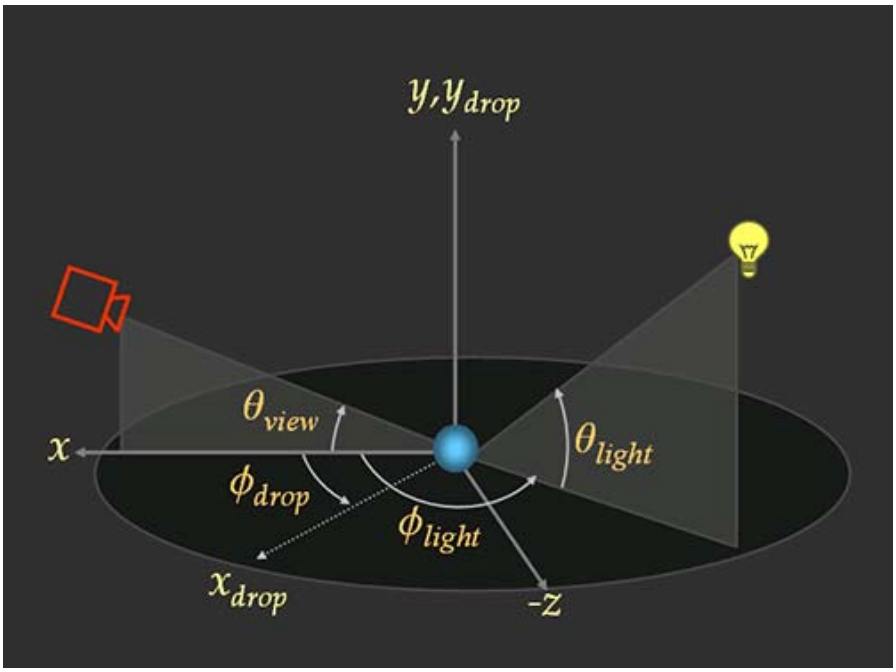

Figure 2. Angles used to index into the rain textures

In the interest of space we chose to only use a subset of the texture database available. Specifically, we forgo the variability in the θ_{view} dimension, and use only

the positive angles in the θ_{light} dimension (negative angles are mapped to their absolute value).

The images are converted to L8 DDS textures to facilitate loading. At load time they are normalized (by values encoded in a text file) and loaded sequentially into a 2D texture array. Texture arrays, a new feature in Direct3D10, are very useful for storing textures such as these. We have over 300 textures which are all distinct and require mipmapping and anisotropic filtering. Storing these textures in a texture atlas or a 3D texture would have resulted in artifacts at edges, and slower access times.

To render from these textures, in the pixel shader we use the interpolated vector to the eye and vector to the light to construct the angles θ_{light} and ϕ_{light}. These are used to index into the texture array to find the nearest 4 textures (2 nearest textures in each dimension), which are then lerped to find the final pixel color.

Rendering Fog

To enhance the look of the rainy scene we also use an accurate analytical fog model. [BoSun05]. This model correctly renders the glows around light sources, and takes the fog into account when determining the reflectance of surfaces.

Running the Sample

The sample shows a bridge with two point lights and one directional light. Both the scene and rain respond to these lights.

The directional light has no intensity falloff and its changeable parameters are the direction of the light and its intensity. The direction of the light is controlled by the left mouse button, and depicted by a small white arrow. The light response slider lets the user control how much the rain drops respond to the directional light.

Both lamps in the scene are modeled by point lights. Controls are provided to change the intensity of the point light and the range of its spotlight (if this option is chosen by the Use Spotlight checkbox). The light response slider under these controls lets the user control how much the rain drops respond to the point lights.

The rain can be frozen using the Move Particles checkbox to view the effect of changing the viewpoint and light direction on the appearance of the particles. The user can also choose not to render the particles or background, to see the speed of the GPU particle animation. The *use simple shader* checkbox can be used to compare how the streaks would look if rendered with constant brightness. The user can choose to render only a fraction of all the particles being simulated by using the Particles Drawn slider. Finally, the wind slider increases the influence of a global wind direction on the motion of the particles.

Performance

For all numbers given below the configuration of the machine used is; Intel Core 2 CPU, 2.93 GHz, 2046 MB RAM, NVIDIA GeForce 8800 GTX. For table 2 the camera is at the center of the particle system, with a field of view of 45 degrees, a screen resolution of 1280 x 1024, and the simple shader turned on.

Table 1. Performance of Stream Out for animating particles

Number of particles	200,000	1,000,000	5,000,000
Frame rate	574	257	67

Table 2. Performance for extruding and rasterizing sprites

Number of particles	200,000	1,000,000	5,000,000
Frame rate	545	126	26

Integration

This technique is relatively easy to integrate into an engine, since it requires minimal CPU intervention. The engine needs to supply the original particle positions, setup the particle animation as shown in code listing 1, and then setup the particles to render. If the rain is to follow the character, as in our demo, the engine also needs to provide the location of the camera at each frame. This position is used to re-spawn any rain particles that have fallen out of bounds.

To integrate this technique into a scene, we need to render the scene before and render the sprites with depth testing and alpha blending.

In general, animating rain using a particle system is more useful for realistic looking rain with lots of behavior (like changing wind). If a constant, light rain is desired other approaches like [Tatarchuk06] or [Wang04] might be more feasible. Another scenario where this particle system approach might not be preferable is where the character can move quickly to very different parts of the scene, since in that case the particle system might take some time to catch up to the camera.

References

- Photorealistic Rendering of Rain Streaks. Kshitiz Garg and Shree K. Nayar. ACM Transactions on Graphics (SIGGRAPH) July 2006.
- A Practical Analytic Single Scattering Model for Real Time Rendering. Bo Sun and Ravi Ramamoorthi and Srinivas G. Narasimhan and Shree K. Nayar. ACM

Transactions on Graphics (SIGGRAPH), August, 2005. http://www1.cs.columbia.edu/~bosun/sig05.htm

- Artist-Directable Real-Time Rain Rendering in City Environments. Natalya Tatarchuk and John Isidoro. Eurographics Workshop on Natural Phenomena 2006.
- Rendering Falling Rain and Snow. Niniane Wang and Bretton Wade. ACM SIGGRAPH 2004 Sketches