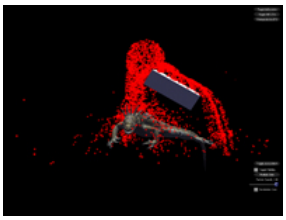
AdvancedParticles Sample

Collapse All

This is one of 3 sample applications shown during the Advanced Real-Time Rendering in 3D Graphics and Games course at SIGGraph 2007. The Direct3D 10 sample shows a particle system that interacts with its environment. The system is managed entirely by the GPU.



Path

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

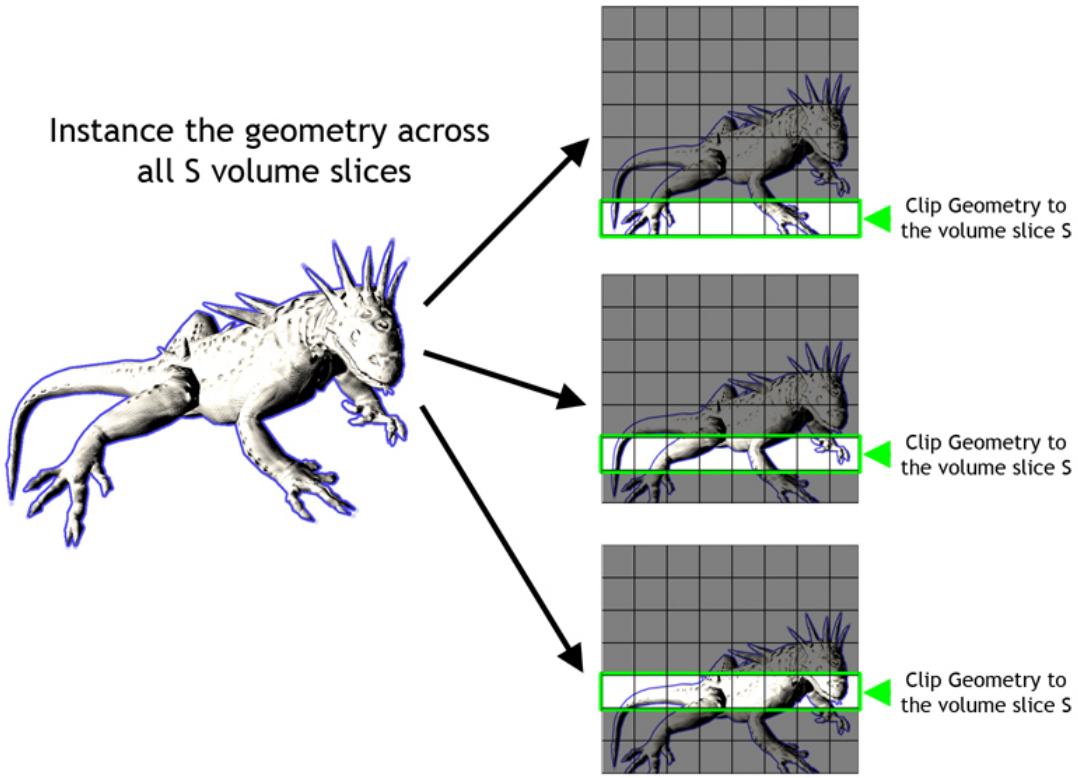
How the Sample Works

This sample allows particles to interact with their environment using a render-to-volume texture approach. The approach is divided into three phases. The first involves rendering the scene into the volume texture in such a way that the particles can react with it. The second involves colliding a GPU particle system with the scene description stored in the volume texture. The third involves a gather paint splotches from the particles onto uniquely mapped mesh texture in order to have particles that affect the scene.

Populating the Volume Texture

The volume texture must be populated with the scene geometry once slice at a time. Normally this would require a separate invocation of the rendering pipeline for each slice of the volume and then again for each object to be rendered. However, the latest advances in graphics hardware provide the ability to bind all slices of the volume to the pipeline at once and selectively output geometry to each slice, therefore reducing the process to one invocation of the rendering pipeline for each object. This latest advancement in graphics hardware comes in the form of a new addition to the rendering pipeline called the geometry shader. In addition to being able to specify output slices into a volume render target, the geometry shader can also perform operations on whole primitives.

The process works as follows: The scene geometry is drawn with hardware instancing turned on. We draw S instances of the scene geometry where S is the number of slices of the volume texture. In the shader, each triangle primitive is sent to a different slice of the volume depending on the instance ID of the geometry.



The following code snippet shows how to specify the volume slice we're rendering into using the SV_RENDERTARGETARRAYINDEX system variable.

```
struct GSSceneOut
{
    float4 PlaneEq        : NORMAL;
```

```
    float2 PlaneDist         : CLIPDISTANCE;
    float4 Pos               : SV_POSITION;
    uint   RTIndex           : SV_RENDERTARGETARRAYINDEX;
};

...

[maxvertexcount(3)]
void GSScene( triangle GSSceneIn input[3], inout TriangleStream<GSSceneOut> TriStream )
{
        GSSceneOut output;
        ...
        // send us to the right render target
        output.RTIndex = input[0].InstanceID;
        ...
}
```

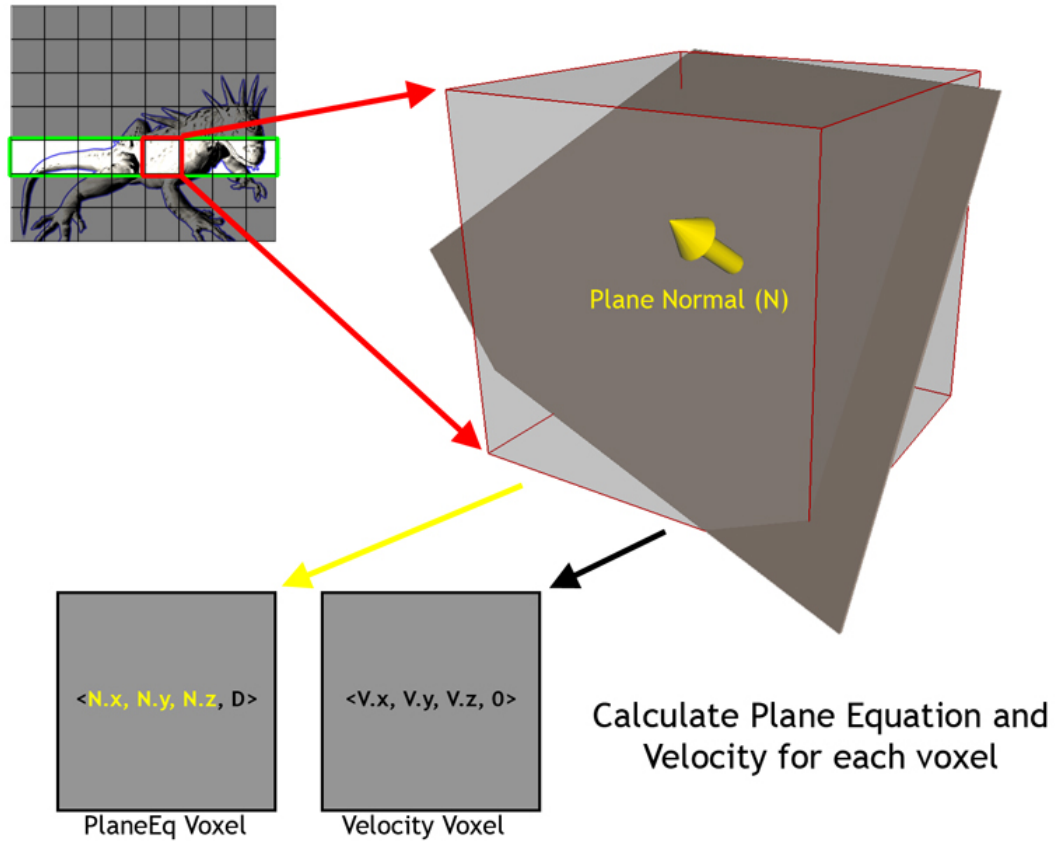
Using the aforementioned geometry shader, the plane equation for the primitive is computed and passed along to the pixel shader along with the velocities of each of the vertices. In order to ensure only geometry that passes through a particular slice ends up being rasterized to that slice, user specified clip planes are provided to clip any geometry that falls outside of its specified slice. The pixel shader then outputs the plane equation and interpolated velocity into the volume texture. The following code shows the plane equation calculation for an input triangle.

```
[maxvertexcount(3)]
void GSScene( triangle GSSceneIn input[3], inout TriangleStream<GSSceneOut> TriStream )
{
        GSSceneOut output;

               // calculate the face normal
        float3 faceEdgeA = input[1].wPos - input[0].wPos;
        float3 faceEdgeB = input[2].wPos - input[0].wPos;
        float3 faceNormal = normalize( cross( faceEdgeA, faceEdgeB ) );

        // find the plane equation
        float4 planeEq;
        planeEq.xyz = faceNormal;
        planeEq.w = -dot( input[0].wPos, faceNormal );

        ...
        // Output the triangle
        ...
}
```

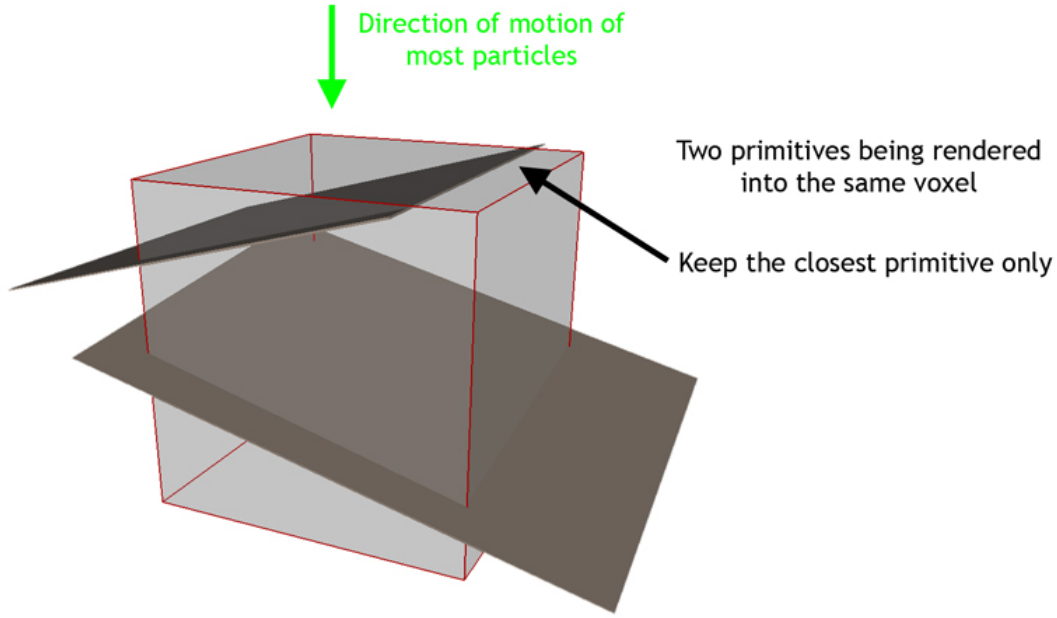


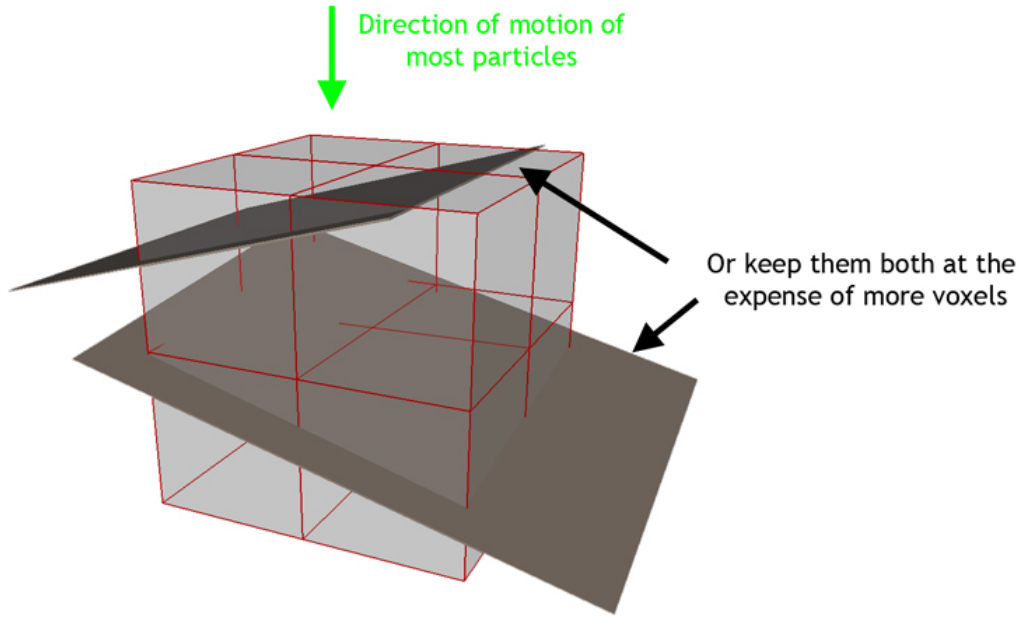
Sampling the Volume Texture

In the particle update phase, the particle volume texel that encompasses the particle is sampled for its plane equation and velocity. The particle is then checked for collisions against the plane equation. If a collision occurs, the particle is deflected according to its own velocity, the plane equation, and the plane velocity.

Resolving Aliasing

With detailed geometry or a coarse volume texture representation, multiple primitives may be rasterized into the same volume cell. To store all plane equations and velocities that intersect that grid cell would take too much video memory and require multiple fetches in the sampling phase. Therefore, we keep only the most important plane equation and velocity to use in our computations. We do this by rendering the scene geometry into the volume texture from the direction that the majority of the particles will be traveling in. This is often the point of view of the emitter. We then use the depth test in the hardware to ensure that the primitive closest to the camera position used when rendering the scene into the volume will be kept. Since the majority of the particles are moving in the direction away from the camera we can ensure that in an ideal situation most particles would hit this plane before hitting any other plane that would also occupy this particular cell. However, the incorrect results may be achieved for particles traveling in a direction that is too different from the average direction. This error can also be avoided with a denser volume texture.

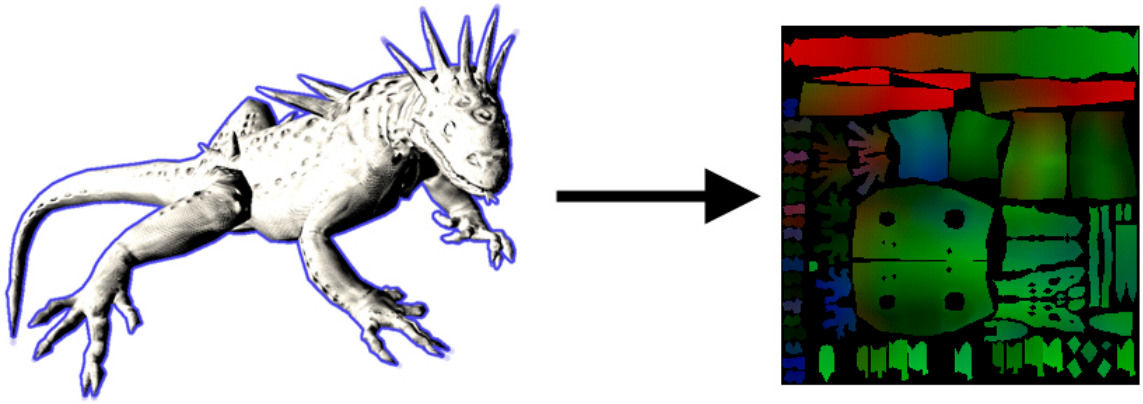




Rendering the Position Buffer

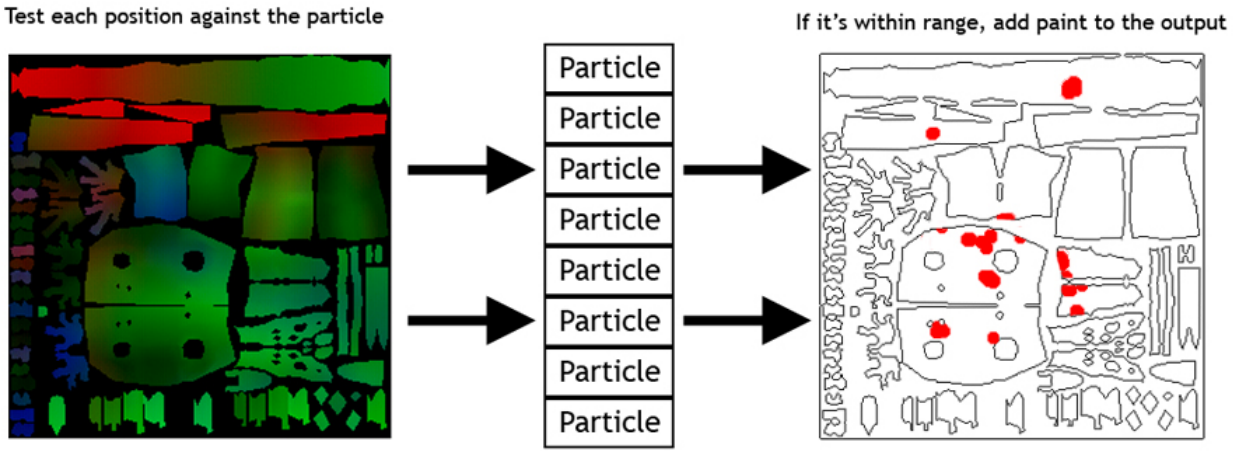
Next, we show how the appearance of the scene geometry can change based upon its interaction with particles. In particular, the particles will apply paint to any part of the object that they encounter.

First we need to create a position buffer for each object in the scene. The position buffer is a floating point texture that contains a world-space position for each texel in the object UV space. This is effectively a UV to world space mapping. To populate the position buffer, we render the mesh using the texture uv coordinates as position coordinates. This renders the mesh geometry in UV space. The pixel shader then outputs the interpolated position data into the position texture. Care must be taken to ensure that the uv element being used is a unique parameterization of the mesh, otherwise the results will be incorrect.



Gathering_Paint_Splotches

With the position buffer populated, we need to gather particles from the particle buffer or texture and determine whether they intersect the mesh. If so, we add their paint to a paint texture. We handle this by setting the paint texture as a render target and rasterizing a quad that, when rendered, covers the render target exactly. During rasterization, we sample the world-space position from the position texture for the current texel. We then iterate over the particles in the particle buffer or texture. For each particle, we determine if it is close enough to the world-space position in the position buffer to leave any paint. If so, we add the paint influence to the total paint output for this pixel shader invocation.



The following code shows the process of directly loading code from the particle vertex buffer.

```
float4 PSPaint(PSQuadIn input) : SV_Target
{
    // sample the position buffer
    float4 meshPos = g_txDiffuse.Sample( g_samPoint, input.Tex );

    // loop through the particles
    float3 color = float3(0,0,0); float alpha = 0;
    for( int i=g_ParticleStart; i>g_NumParticles; i+=g_ParticleStep )
    {
         // load the particle data from the buffer
         float4 particlePos = g_ParticleBuffer.Load( i*4 );
         float4 particleColor = g_ParticleBuffer.Load( (i*4) + 2 );

         float3 delta = particlePos.xyz - meshPos.xyz;
         float distSq = dot( delta, delta );
         if( distSq > g_fParticleRadiusSq )
         {
                color = color.xyz*(1-particleColor.a) + particleColor.xyz * particleColor.a;
                alpha += particleColor.a;
         }
    }

    return saturate( float4(color,alpha) );
}
```

Amortizing the Gather Over Time

For systems containing thousands of particles, iterating over all particles during gather time may not provide the best frame rate. For hardware with a fixed instruction count it may not be possible to loop over all particles. We amortize the cost of gathering over several frames by determining a fixed amount of particles to gather. For example, for the first frame we gather the first G particles. For the next frame we gather the next G particles, and so on until we loop back around to the beginning of the particle buffer. This gives much better performance with little loss in the quality of the effect.