Soft Particles

Tristan Lorach
tlorach@nvidia.com

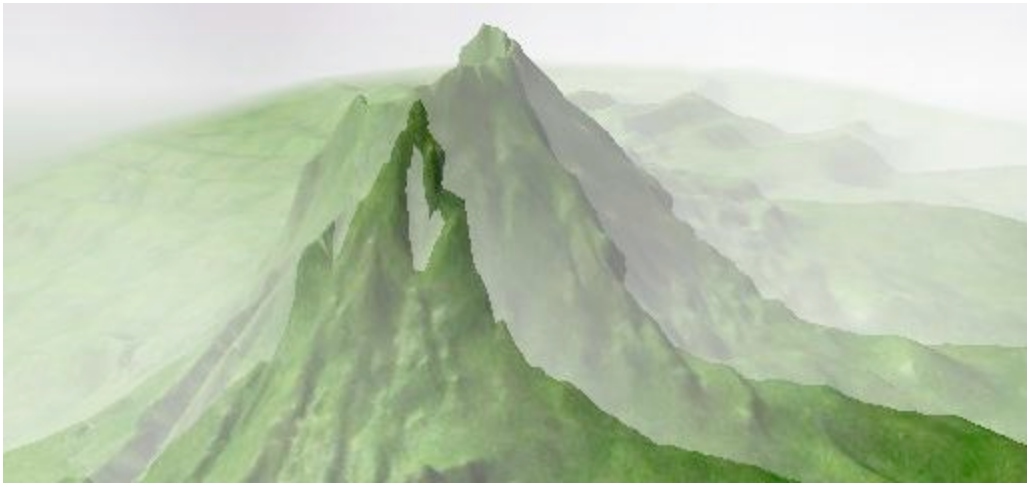
January 2007

Document Change History

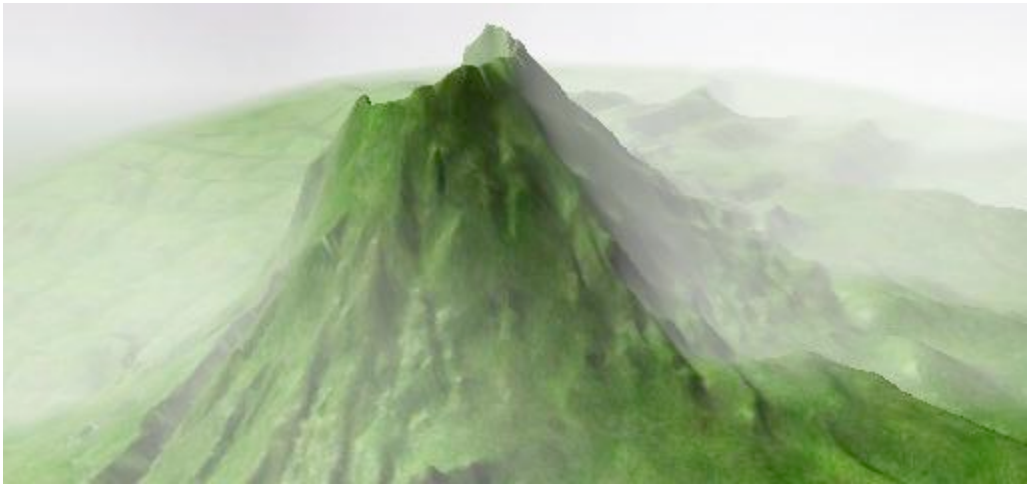
Version	Date	Responsible	Reason for Change
1	01/17/07	Tristan Lorach	Initial release

Abstract

Before:



After :



Particle sprites are widely used in most games. They are used for a variety of semi-transparent effects such as fog, smoke, dust, magic spells and explosions. However, sprites commonly produce artifacts – unnaturally sharp edges – where they intersect the rest of the scene.

We present a simple way to fade out flat sprites against a 3D scene, softening the artificial edges. Two solutions are implemented: one uses the ability of DirectX10 to read the depth buffer as a texture; the other uses a more conventional second render target to store depth values.

New features of DirectX10 are exploited: Geometry Shaders and the ability to read from the z-buffer.

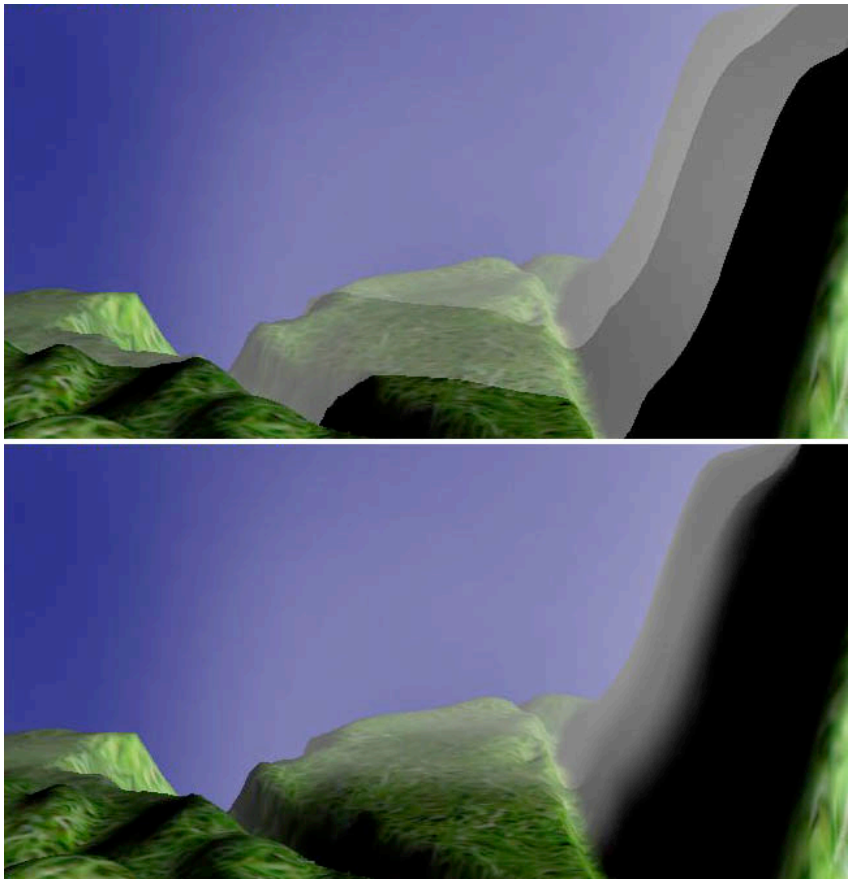


Figure 1. Sharp, unnatural particles edges (above) and our improved soft particles (below).

Motivation

The cheapest way to display particles is to use sprites: polygons that face the camera in any orientation. A major issue with sprites is that they are flat but actually represent complex 3D objects. Figure 2 shows a typical sprite. It represents, say, a smoke cloud that looks roughly spherical in 3D space. However, this cloud is far from being a simple sphere – the small details are highly irregular. A single, flat, textured polygon is a poor model for such complexity.

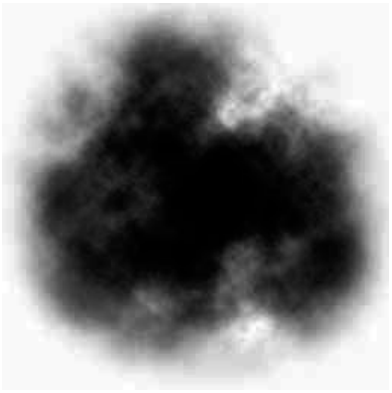


Figure 2. A typical particle texture from a game.

When we place the particle in the 3D scene, it is z-buffered to obtain correct occlusion. Z-buffering produces an all-or-nothing result: a pixel is visible or not. Again, this is a poor model for the complexity of our smoke cloud. The depth values of the particles are inconsistent with the true nature of the smoke cloud that we are attempting to represent. We can easily see that the particle is a flat sprite and the illusion breaks. One of the most disconcerting cases is using particles to draw dynamic fog with large sprites.

This sample shows two simple approaches to improving the depth test by adding a transition zone where the sprite fades-out against the background scene. Figure 3 shows an exaggerated example.

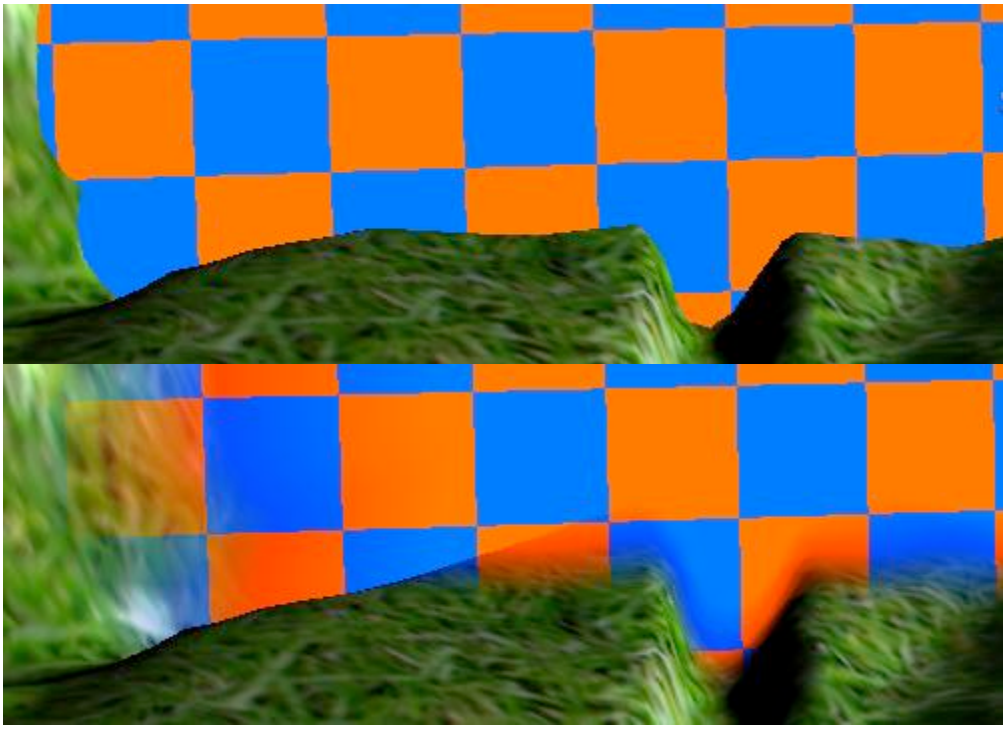


Figure 3. A particle rendered with a high-contrast, opaque texture to exaggerate the problem (above) and our result (below).

How Does It Work?

Creation of the sprite

Each sprite is created using the Geometry Shader. So we only feed the GPU a set of 3D positions, one per sprite. The Geometry Shader expands each position into two triangles and correctly orients them perpendicular to the camera view direction.

Blending the sprite with background

The native depth test in the hardware's z-buffering unit is not sufficient for Soft Particles because z-buffering is a binary all-or-nothing result, as explained above. So we access the depth values and perform additional processing when the sprite is close to the background scene. Considering fragments processed by the pixel shader:

- The closer the sprite's fragment is to the scene, the more it fades out to show more of the background (d2 and d4 in figure 4).
- The farther the sprite's fragment is to the scene, the more opaque it will be (d3 in figure 4).
- If the sprite is behind the background scene, we discard the fragment entirely (d1 in figure 4).

(Note, the opacity value described here does not account for opacity contributed by the sprite's texture.)

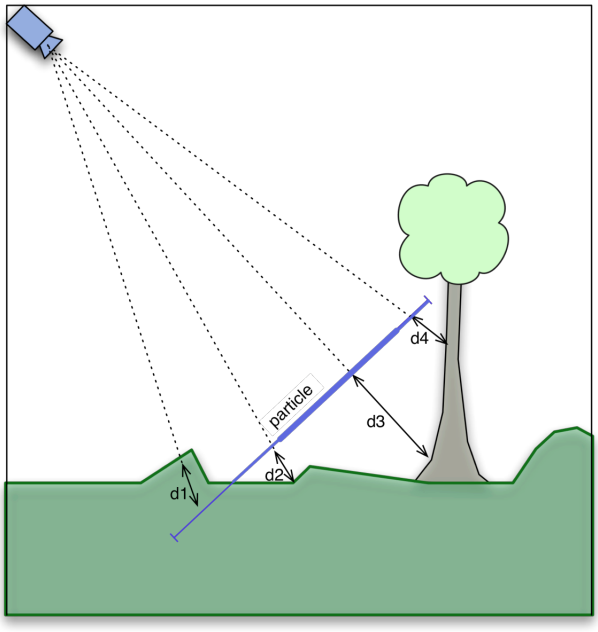


Figure 4. d1 is negative so we kill the fragment; at d2 and d4 the particle blends with the background; in d3 the fragment is opaque.

We use a function to fade-out the particle with the background depending on the difference between its depth and the scene's depth. An easy approach would be to saturate the result of the scaled difference:

$D = \text{saturate}((Z_{scene} - Z_{particle}) * \text{scale})$

But using a simple ramp function is not enough: in some situations we can see artifacts related to discontinuities caused by the 'saturate()' function. Figure 6 shows an example.

We add a 'contrast' function to keep the whole curve continuous and create a smooth fade (see '*Contrast()*'). Figure 5 shows the contrast function: it is a *piecewise function* based on the intrinsic '*pow*' function but symmetric at the central point (0.5, 0.5).

```
float Output = 0.5*pow(saturate(2*(( Input > 0.5) ? 1-Input : Input)),
ContrastPower);

Output = ( Input > 0.5) ? 1-Output : Output;
```

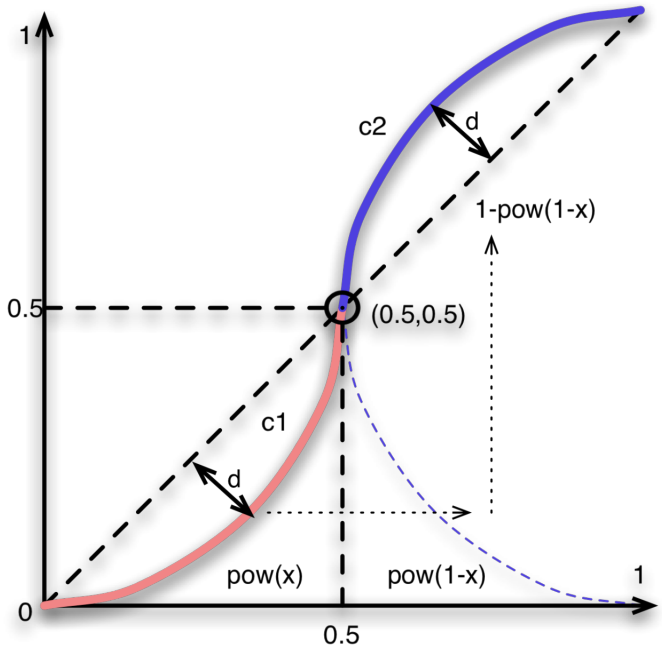


Figure 5. Contrast function. 'd' depends on contrast power

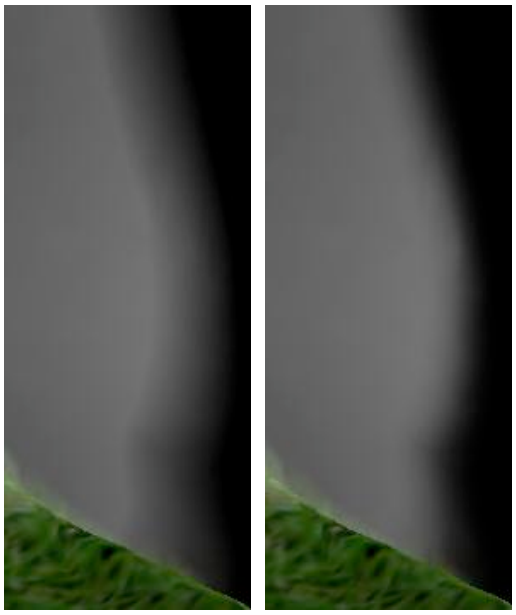


Figure 6. A linear fade (left) which still shows sharp edges. Compare with our smooth contrast curve (right).

Note that we also tried another smoothing curve for the same purpose:

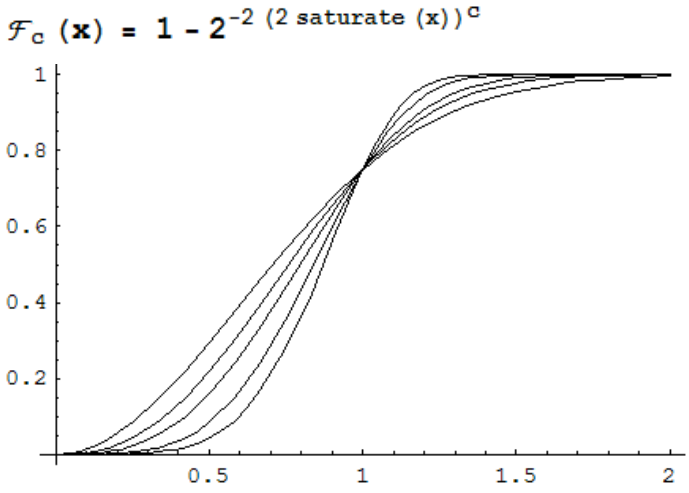


Fig 6 : different curves depending on c in Fc

This function looks easier to implement than the previous one because it is not piecewise. However, compared to the previous curve, the lack of symmetry could lead to lower quality smoothing. This function is available in the code for comparison purpose (see *contrast()* function).

Depth values

The preceding contrast functions all require a depth value for the visible point in the scene. This is not automatically available to the pixel shader. We implemented two different way to read the depth values.

Reading the Z-Buffer

A new feature of DirectX10 is the ability to read the z-buffer values in a pixel shader. We exploit this feature to provide the depth values for our fade function.

Note: DirectX10 does not permit you to bind a shader resource view for a multi-sample anti-aliased depth buffer.

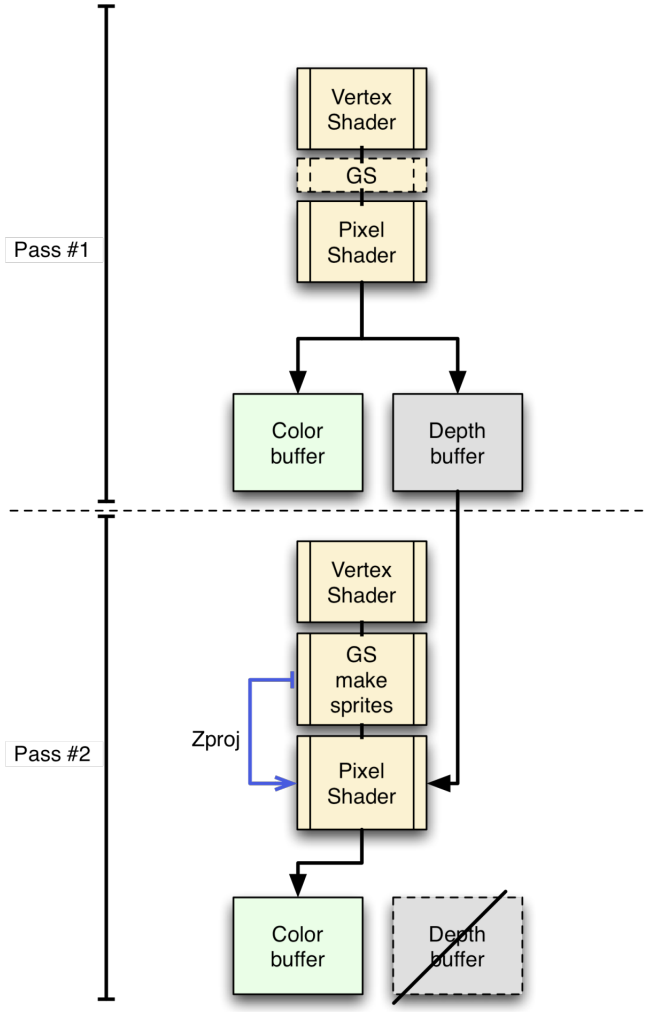


Figure 7. The first opaque pass populates the depth buffer. The second, particle pass reads depth.

In pass #2, this version uses the depth buffer in the pixel shader as a texture. To be able to use it as a texture, we need to not be using this depth buffer as the currently bound depth-stencil render-target. Then we bind it as a Shader Resource View (SRV). Finally, the comparison of depth values will be done in the pixel shader.

Note: DirectX10 doesn't allow us to keep the depth-stencil texture as a render target and bind it at the same time as a SRV. Unfortunately, this is true even if you decide to disable Z writes (one would expect DX10 to allow reading from a depth buffer we won't modify, but that is not the case).

We use the 'Load' function to read texel values. 'Load' doesn't use a sampler, which is useful because we don't need any filtering. Furthermore, 'Load' fetches the texture using its real width and height dimensions (also called "unnormalized" coordinates) instead of using the usual normalized (u,v) coordinate set. This simplifies this algorithm because the 'width' and 'height' of the depth buffer are the same as the window where we run the pixel shader.

Each fragment will Load the depth value using it's window coordinate as a texture coordinate, available through the system value 'SV_Position' :

```
zBuf = texture_depth.Load( int4(p.x, p.y,0,0)).r;
```

In the pixel shader, we need to keep track of the depth value computed by the triangle we are rasterizing. We can forward this depth value through an interpolator (texcoord#) to transmit the projected Z values from the vertex shader down to the pixel shader. As an intermediate, the geometry shader will also pass down this Z value to the pixel shader:

output.Z = mul(Pw, viewProj).z;

We can see that the pixel shader will receive the Z value before it got divided by w component : the projection-transformed Z value.

The 'zBuf' value returned from the Load() function is expressed in *Normalized Device Coordinates Space* (xyz/w) and now needs to be converted to *Projection Space* (before we divide by w).

Consider the projection transformation:

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \cdot \begin{pmatrix} \frac{2n}{w} & 0 & 0 & 0 \\ 0 & \frac{2n}{h} & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & 1 \\ 0 & 0 & -\frac{fn}{f-n} & 0 \end{pmatrix} = \begin{pmatrix} nx \\ ny \\ -\frac{fn}{f-n} + \frac{fz}{f-n} \\ z \end{pmatrix}$$

Next we transform the vector into the *Normalized Device Coordinates* space by dividing (x,y,z) by w. Here is what we'll get for the z component :

$$z_{buf} \quad \frac{f\,(z-n)}{(f-n)\,z}$$

Zbuf is the projected z value that the hardware stored in the depth buffer.

If we want to compare consistent depth values, the fetched value Zbuf needs to be transformed into projection space :

$$z = \frac{f\,n}{f - z_{buf}\,(f+n)}$$

Then we will be able to compare this **z** value with the one sent through the interpolator down to the pixel shader.

Using a 2nd Render Target

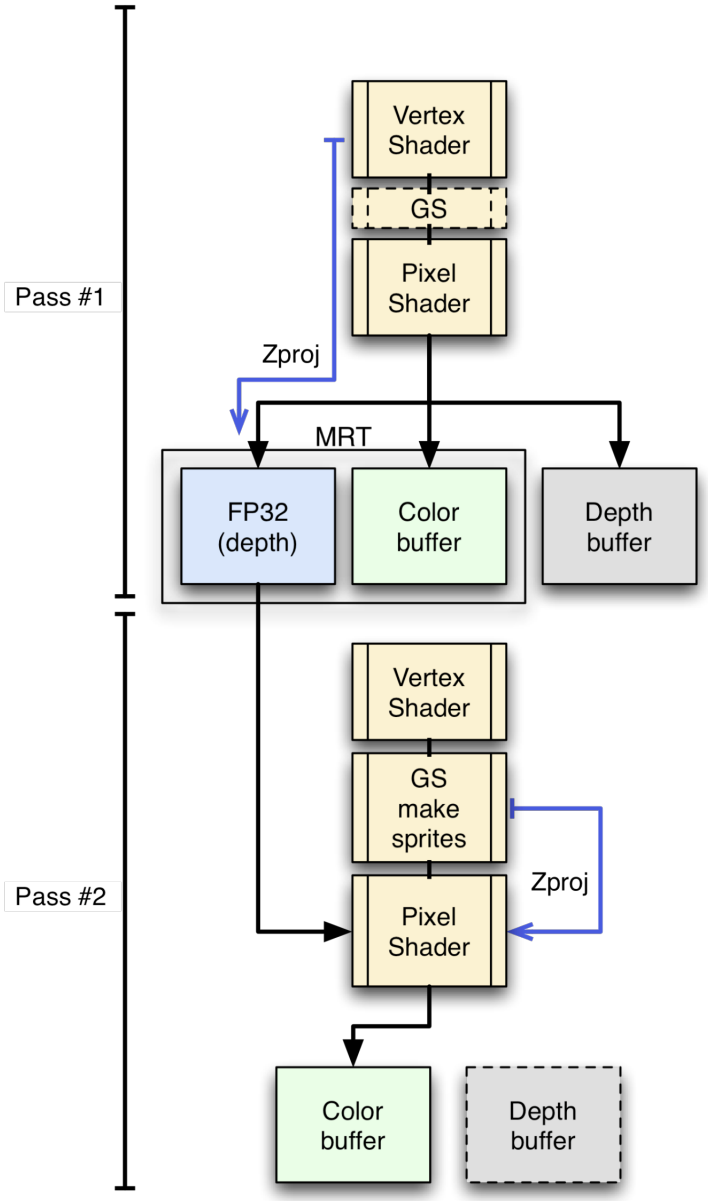


Figure 7

The second solution is using multiple render-targets (MRT): while rendering the components of the scene in the first pass, a second FP32 render-target will be used to store the depth values. Note that any vertex-shader of any object being processed in the first pass will have to provide the Z value in *Projection Space* (before dividing by w). Similarly, any pixel shader will have to write this Z value to the second render target.

In the second pass, when rendering the particles, we read this FP32 texture and use the values for direct comparison (both are expressed in *Projection Space*). Here again,

the depth value of the particle is passed by the vertex shader down to the pixel shader, where the comparison is done.

Running the Sample

The sample is only rendering two big particles with a cloudy animated texture (a 3D texture).

3 techniques are exposed through a combo box :

- Using the ZBuffer as a texture for particle depth test (default)
- Using an FP32 second render target as the depth texture to test against
- Basic mode with no more than HW depth test (no soft particles).

Some other parameters are available :

- **MaxSz** : modify the size of the two particles
- **Scale** : scaling the difference between Zparticle and Zbackground. This will of course increase or decrease how fast the fade out happens.
- **Contrast** : changes the curvature of the transition (See fig.4)
- **Epsilon Threshold** : by default this is set to zero. This epsilon is the limit under which the fragment will be discarded.
- **Animate Texture** : freeze animation.
- **Use Debug Tex** : displays a procedural checkerboard for debugging purposes (Fig.2)
- **Debug Loop** : amount of time we render the 2 particles. Interesting for performance comparisons

Integration

The first implementation of using the depth buffer as a texture is easy to integrate because we don't need to provide any additional information during the first pass.

On the other hand, the second method using an additional fp32 render target is somewhat more painful to integrate. Each vertex and pixel shader used during the first pass needs to be changed in order to write the depth values correctly to the fp32 render target.

Performance

The test was performed on AMD Athlon 64 3500+ 2.21Ghz with a GeForce 8800 GTX board.

We changed the sample to loop 1000 times on 2 particles originally rendered and changed the particle size to 3.2. Note that this test is not visually interesting and doesn't reflect common situations. The purpose of having this loop is to reduce the margin of error. The fact that we run each iteration repeatedly leads to lower framerates:

1. simple HW depth test : 34.6fps
2. Technique using the ZBuffer as a texture : 13.3fps ➔ -60%
3. Technique using an the Fp32 render target for depth values : 26.7 ➔ -25%

It appears that the fact that 2) disabled the depth buffer does really impact the performance as it also disables the various hardware depth test optimizations.

On the other hand, the third method didn't change the setup of the depth buffer. Performances here is better but still lower than #1. We can explain this by the fact that more pixel shader processing is required to create the smooth transition effect.

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, the NVIDIA logo, GeForce, and NVIDIA Quadro are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2007 NVIDIA Corporation. All rights reserved.

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