Soft Particles Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

About this Document

The purpose of this document is to present a tutorial on how to implement soft particles on the PlayStation®Vita GPU.

Related Documentation

Refer to the following document for a detailed description of the soft particle implementation algorithm used in this tutorial:

- NVIDIA's white paper *Soft Particles*, by Tristan Lorach.
  http://developer.download.nvidia.com/whitepapers/2007/SDK10/SoftParticles_hi.pdf

Conventions

The typographical conventions used in this guide are explained in this section.

Hints

A GUI shortcut or other useful tip for gaining maximum use from the software is presented as a 'hint' surrounded by a box. For example:

Hint: This hint provides a shortcut or tip.

Notes

Additional advice or related information is presented as a 'note' surrounded by a box. For example:

Note: This note provides additional information.

Text

- Names of keyboard functions or keys are formatted in a bold serif font. For example, **Ctrl**, **Delete**, **F9**.
- File names, source code and command-line text are formatted in a fixed-width font. For example:

```
const float A = delta * delta;
```

Errata

Any updates or amendments to this guide can be found in the release notes that accompany the release.

1 Introduction

Overview

Typically, sprite-based particle systems are used to create effects such as fire, explosions, and smoke. However, when those particles intersect with other geometry in the scene, it creates visible hard edges, thus reducing the quality of these effects. A well-known method to address this issue is to render soft particles to soften those intersection edges and maintain visual quality. This soft particle tutorial demonstrates how to implement the soft particles algorithm on PlayStation®Vita. It uses several render targets – each serving a different purpose – so that the depth buffer values can be used when rendering the particles. The tutorial also demonstrates a method to optimally render these particles to reduce fragment processing.

Purpose

The purpose of the soft particles tutorial is to:

- Demonstrate how to create and use multiple render targets with multiple scenes
- Demonstrate how to render to an off-screen depth buffer and color buffer
- Demonstrate how to use the off-screen buffers as a texture
- Demonstrate how to use the depth values in a fragment shader to modulate the alpha component
- Demonstrate how to setup the blend equation to composite a low-resolution color buffer back to the display buffer
- Provide a useful set of program code that can be used in commercial application development for the platform

2 Theory

This chapter provides some theoretical background to implementing soft particles.

Algorithm

The soft particles algorithm used in this tutorial is discussed in NVIDIA's white paper *Soft Particles*. In this algorithm, depth values from the depth buffer are sampled per particle fragment (pixel) and compared against the fragment's depth. Based on this comparison, the algorithm sets:

- *A relatively more transparent alpha value* the closer a particle fragment depth value is to the depth buffer value
- *A relatively more opaque alpha value* the farther away a particle fragment depth value is to the depth buffer value

Obviously, if the particle fragment is behind the depth buffer value it will not be displayed (it will be depth-tested out).

A simple formula to adjust the alpha is to take the difference of the depth values and apply a scale to it:

```
depthFadeDiff = (depthBufferValue - particleDepth) * depthFadeScale;
```

However, visual artifacts can still be noticeable through this linear ramp. To create a more continuous and smoother curve, apply a `pow()` function and create symmetry around the center point (0.5, 0.5):

```
depthFade = 0.5 * pow(2*(depthFadeDiff > 0.5 ?
         (1.0 - depthFadeDiff) : depthFadeDiff), depthContrastPower);
```

3 Implementation

This chapter provides information about the implementation of the soft particles algorithm described in Chapter 2, Theory. It also describes the steps required to render particles in a way that will reduce fragment processing.

Depth Buffer as a Texture

When implementing the soft particles algorithm, the first step is to generate a depth buffer to use as a texture when rendering the particles. It includes the following parts:

- Allocate memory to store the depth values.

Before allocating memory, remember to align both the width and height by the tile size.

```
m_pPartDepthBufferData = graphicsUtilAlloc(
        SCE_KERNEL_MEMBLOCK_TYPE_USER_CDRAM_RW,
        alignedWidth*alignedHeight*4,
        UTILSMAX(SCE_GXM_TEXTURE_ALIGNMENT,
        SCE_GXM_DEPTHSTENCIL_SURFACE_ALIGNMENT),
        SCE_GXM_MEMORY_ATTRIB_READ | SCE_GXM_MEMORY_ATTRIB_WRITE,
        &m_partDepthBufferDataId);
```

- Create a depth surface with this memory block.

```
sceGxmDepthStencilSurfaceInit(
        &m_partDepthSurface,
        SCE_GXM_DEPTH_STENCIL_FORMAT_DF32,
        SCE_GXM_DEPTH_STENCIL_SURFACE_TILED,
        alignedWidth,
        m_pPartDepthBufferData,
        NULL);
```

- Set the store mode for the depth surface.

The PlayStation®Vita GPU contains on chip memory for the depth buffer. The only time that depth values are written out to memory is when the PlayStation®Vita GPU encounters a partial render. Setting the store mode will force the depth values to be written into memory after the scene has completed rendering.

```
sceGxmDepthStencilSurfaceSetForceStoreMode(
        &m_partDepthSurface,
        SCE_GXM_DEPTH_STENCIL_FORCE_STORE_ENABLED);
```

- Create a texture object with the same memory block allocated above so that it can be sampled from the fragment shader.

```
sceGxmTextureInitTiled(
        &m_partDepthTexture,
        m_pPartDepthBufferData,
        SCE_GXM_TEXTURE_FORMAT_F32M_000R,
        m_partDepthBufferWidth,
        m_partDepthBufferHeight,
        1);
```

Render Scene to Depth Buffer Only

With the depth surface and texture created, the memory now needs to be populated with depth values from the scene excluding the soft particle effects. The depth surface is the only area being rendered to, so the scene needs to only set the depth surface. The color surface parameter and sync objects are set to `NULL`.

```
sceGxmBeginScene( m_graphicsData.pContext, 0, m_pPartDepthRenderTarget, NULL,
NULL, NULL, NULL, &m_partDepthSurface );

// render scene
...
sceGxmEndScene( m_graphicsData.pContext, NULL, NULL );
```

Render Particles

The depth texture is now primed and ready to be used as a texture in our shader. Texture coordinates used to sample the depth texture are generated in the vertex shader by transforming the particles' quad *(x, y)* positions into screen space and normalizing them so that they are between 0.0 – 1.0. Notice that only the *(x, y)* positions are converted; the *z* component is still in model view projection space.

```
vPosition = mul(modelViewProj, aPosition);
vTexCoordScreen = vPosition.xyz;
vTexCoordScreen.xy = vPosition.xy / vPosition.w;
vTexCoordScreen.xy = vTexCoordScreen.xy * half2(0.5f, -0.5f) + half2(0.5f,
0.5f);
```

The algorithm formula described in Chapter 2, Theory, is implemented in the fragment shader. Thus, the newly generated texture coordinates are passed to it and used to sample the depth texture.

The first step is to calculate the delta between the particle fragment depth value and the depth buffer value. The depth texture values are brought into the same space as the particle fragment's depth value and then the delta is calculated. (The texture query format is explicitly stated as `<float4>`. This will guarantee that the texture unit will return a full 32-bit depth value in the `r` component.)

```
half depthDelta = (half)(((farPlane*nearPlane)/(farPlane –
tex2D<float4>(depthTexture, vTexcoordScreen.xy) *(farPlane-nearPlane))) -
vTexcoordScreen.z);
```

Next the scale and contrast power is applied:

```
// apply scale
half input = depthDelta * depthfade_param.x;

// apply contrast power
half output = 0.5f*pow(saturate(2*(( input > 0.5f ) ? 1.0f - input : input)),
        depthfade_param.y);
```

Now modulate the alpha component of the final output color:

```
// modulate alpha component
color.a *= (half)((input > 0.5f) ? 1.0f - output : output);
```

For additive textures that do not contain an alpha, apply the results directly to the final output color:

```
color = (color * (half)((input > 0.5f) ? 1.0f - output : output));
```

Optimized Particle Rendering

Effects such as fire, explosions, and smoke often require many overlapping transparent particles to create a believable look. However, this layering can potentially have a negative impact on performance because of the amount of fragments that need to be blended. One way to reduce the performance impact is to simply reduce the number of particle fragments.

One implementation to achieve this is to reduce the render target size that the particles are rendered to and then to composite the lower resolution color buffer on top of the display buffer. Obviously, with a lower resolution render target, the visual quality of the particles decreases. The particles become more blurry the smaller the render target is. However, effects like fire, explosion, and smoke already possess blurry particles so the visual impact is minimal.

Here are the basic rendering steps:

(1) Creating Color Buffer as a Texture

(2) Rendering Particles and Compositing the Image

Creating Color Buffer as a Texture

Creating a color buffer and using it as a texture is very similar to creating a depth buffer as described in Render Scene to Depth Buffer Only. The only differences are (a) the width and height do not have to be aligned by the tile size and (b) `sceGxmColorSurfaceInit()` is used to create the surface instead of `sceGxmDepthStencilSurfaceInit()`.

To ensure correct rendering when creating an off-screen color buffer for particle rendering, the destination color (`dstColor`) and the destination alpha (`dstAlpha`) values need to be cleared to (`0x000000FF`).

Rendering Particles and Compositing the Image

The blend equation is very important for rendering particles to the off-screen color buffer. In the traditional method of rendering particles directly to the main render target (that is, the display buffer), the blend equation for a particle's alpha component is not as important because the display buffer is the final destination for the value. In this algorithm, the particle off-screen color buffer will eventually be composited with the display buffer, so the final alpha value of the particle off-screen color buffer will be used for the blending process.

Two blend modes are explored in this tutorial when rendering particles:

- Average particle alpha blend
- Additive particle alpha blend

Average Particle Alpha Blend Equation

With average blended particles, the alpha component is used to blend the particle color with the destination color. If particles were being rendered directly to the main render target, the color blend equation would be:

```
finalColor = srcColor*srcAlpha + dstColor*(1-srcAlpha)
```

When rendering to the off-screen color buffer, the alpha needs to be preserved for the final blending of the color buffer. Thus, the alpha value needs to be a combination of all the alpha values that affect a particular fragment. To do this, the destination alpha (`dstAlpha`) is multiplied with the (1 - source alpha (`srcAlpha`)).

```
finalAlpha = srcAlpha * 0 + dstAlpha * (1-srcAlpha)
```

The sample code for the average blended particles is as follows:

```
SceGxmBlendInfo blendInfo;
blendInfo.alphaFunc = SCE_GXM_BLEND_FUNC_ADD;
blendInfo.alphaSrc = SCE_GXM_BLEND_FACTOR_ZERO;
blendInfo.alphaDst = SCE_GXM_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
```

Additive Particle Alpha Blend Equation

Typically, particles that use an additive blend mode do not contain an alpha component. If particles were being rendered directly to the main render target, the color blend equation would be:

```
finalColor = srcColor*1 + dstColor*1
```

Thus, the destination alpha is the only thing that needs to be preserved. In the following equation, the source alpha component is cancelled out by the blend factor of 0, and the destination alpha is the only component stored:

```
finalAlpha = srcAlpha * 0 + dstAlpha * 1
```

The sample code for the additive blend particles is as follows:

```
SceGxmBlendInfo blendInfo;
blendInfo.alphaFunc = SCE_GXM_BLEND_FUNC_ADD;
blendInfo.alphaSrc = SCE_GXM_BLEND_FACTOR_ZERO;
blendInfo.alphaDst = SCE_GXM_BLEND_FACTOR_ONE;
```

There is also a specific blend equation when compositing the off-screen particle buffer to the display buffer. In this blend setting, the color blend equation is the focus because it is writing the final color to the display buffer.

```
finalColor = srcColor * 1 + dstColor * srcAlpha
```

The sample code for compositing to the off-screen particle buffer is as follows:

```
SceGxmBlendInfo blendInfo;
blendInfo.colorFunc = SCE_GXM_BLEND_FUNC_ADD;
blendInfo.alphaFunc = SCE_GXM_BLEND_FUNC_ADD;
blendInfo.colorSrc = SCE_GXM_BLEND_FACTOR_ONE;
blendInfo.colorDst = SCE_GXM_BLEND_FACTOR_SRC_ALPHA;
blendInfo.alphaSrc = SCE_GXM_BLEND_FACTOR_ONE;
blendInfo.alphaDst = SCE_GXM_BLEND_FACTOR_ONE;
blendInfo.colorMask = SCE_GXM_COLOR_MASK_ALL;
```

In addition to setting the above blend-mode values, you need to clear the off-screen color buffer to `0x000000FF` before rendering the particles to it.

After the particles have completed rendering to the off-screen color buffer, simply render a full screen quad to the main render target (using the off-screen color buffer as a texture) to composite it to the display buffer. By using the blend equations and clearing the off-screen color buffer as mentioned above, the display buffer will contain the proper color.

4 Conclusion

This tutorial demonstrated the following aspects of working with the PlayStation®Vita GPU:

- Use the depth buffer to modulate the alpha to eliminate hard edges when particles intersect with the rest of the scene.
- To reduce the performance impact of rendering many overlapping transparent particles, render to a lower resolution buffer and composite the image to the final render target. Reducing the resolution by a half or a quarter typically yields significant performance gains with minimal visual impact.