Technical Brief

Shadowing Strategies for HLSL and *FX Composer*

DEVELOPMENT

Table of Contents

Shadowing Strategies 1
Overview 1
Shadow Transforms 2
Using Shadow Maps 3
Special "Roll Your Own" Maps 8
Using Shadow Stencil Volumes 9
Soft Stencil Shadows 12

Shadowing Strategies for HLSL and *FX Composer*

Kevin Bjorke
kbjorke@nvidia.com

Shadowing Strategies

Shadows are important for convincing 3D scenes. Modeling, texturing and surface shading can describe objects, lighting can reveal their broad form, but it is shadows that present to a viewer's eye the spatial relationships *between* the objects in a scene, or the time of day, and the hard or soft qualities of the light. The density, angles, and color of shadows can define time of day, can hide a friend or an enemy, and have a tremendous impact on the mood and feeling of any picture, movie, or game scene.

Modern GPUs are capable of creating and using shadows by a variety of different techniques. While hardly exhaustive, this technical brief will help the developer in understanding and creating the most common forms of shadows used in games and related realtime applications.

Overview

There are many shadow algorithms, and new ones appear in Siggraph sketches, GDC papers, and other academic and industry venues every year. In practice, we can distinguish between two general classes of shadowing methods and shadows: *direct* shadows, which are cast from a specific small light source, and *indirect* shadows, which are created by obscuration of light coming from all directions throughout the scene.

Shadows are often classified as "hard" or "soft" – it's true that indirect shadows will always be soft, while hard shadows must by definition be from a direct source. But the variations in softness are broad. It's wise not to get too caught-up in these labels, since all shadowing algorithms describe approximations to real-world physics. In the real world, shadows don't exist *per se* – they appear from the *lack* of incoming light. But computer graphics rarely tries to exactly emulate the full range of real-world physics! Instead, as with most computer graphics, we want algorithms that will provide us with useful and fast results for a "99% correct" solution.

This brief focuses primary on direct shadowing algorithms. The two dominant forms of direct shadowing used in computer graphics for GPU rendering are *shadow maps* and *shadow volume stenciling.*

Indirect shadowing, at least in terms of pixel and vertex shaders, is often pre-calculated and involves interpolation of values between vertices – the shader load is small but the CPU load large. Some soft shadow algorithms involving complex use of direct shadows will be mentioned here, but indirect shadowing is beyond the scope of this document.

Shadow Mapping Overview

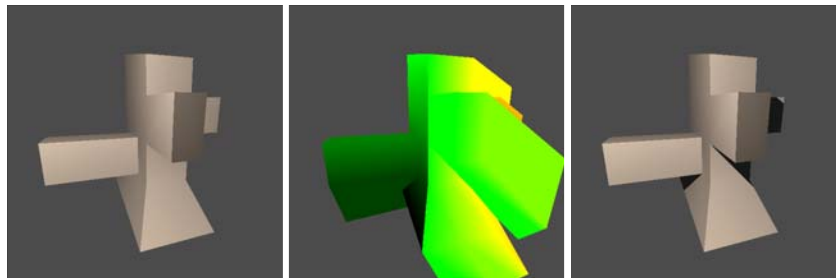
The general principle of shadow mapping is simple: render a view of the shadowing object from the point of view of the light source, record the depth at each pixel, and then use that depth when rendering the final image – for each visible point on the

rendered surface, compare its distance to the light source with the distance calculated from the (projected) map of depths. If the stored depth is less than the calculated distance, then that point is in shadow.

Shadow Volume Stenciling Overview

Stencil shadowing uses a special version of the geometry of the shadowing object, a *shadow volume*. The shadow volume is created by comparing object polygons to the direction of light. Polygons facing the light are unchanged, but polygons facing away from the light are extruded far away (often to infinity). This creates a polygonal volume that corresponds to the shadowed area. The volume is rendered multiple times, always from the normal scene-camera point of view: first to create a valid scene Z buffer, then with frame-buffer stenciling enabled, rendering surfaces that face away from the camera (incrementing the stencil buffer) and then surfaces facing the camera (decrementing). Finally, the scene is rendered with a normal surface shader, with stencil testing set so that pixels that were inside the shadow volume (whose stencil counts will be nonzero) will simply not render at all – only pixels outside the shadow will be rendered.

Below are views showing a simple geometric object, an extruded shadow volume (not normally ever displayed, shown here with colors for shape definition only) and the object with shadow applied.



Three views of stencil shadowing:
simple, stencil volume, and stencil-shadow composite

Shadow Transforms

Regardless of the method you choose for direct shadowing, you will need to determine the incident direction – that is, the vector leading from the shadow source to your shadowed and shadowing objects. FX Composer provides HLSL-standard semantics and annotations for querying the location, orientation, and projection cone of light sources. Here are some typical calls to declare variables that will contain those values:

```
float4x4 LampViewXf : View <string frustum="light0";>;
float4x4 LampProjXf : Projection <string frustum="light0";>;
static float4x4 LampViewProjXf = mul(LampViewXf,LampProjXf);
```

As you can see, the semantics **View** and **Projection**, normally used to query the scene camera, are modified by the **frustum** annotation so that they can be applied to a light source. The light source specified in the example is "light0" but any "light#" designation can be used – further, at runtime the FX Composer user can re-attach

these transforms to any other lightsource the user likes, using the FX Composer properties-panel controls.

The View matrix will express a transform from the model object coordinates into the coordinate system of the shadow source, while the Projection matrix will provide mapping to include the angle of the lamp's spotlight cone. The third, **static** variable declaration concatenates the previous two declarations for a more-complete one-piece matrix.

If your shader includes the header file "shadowMap.fxh" (included with FX Composer), this declaration format is implemented as the macro `DECLARE_SHADOW_XFORMS().` See the "shadowMap.fxh" source for more details.

"shadowMap.fxh" also provides an HLSL function for determining a shadow bias matrix, according to Microsoft's standard formulation. You can call the macro `DECLARE_SHADOW_BIAS` to create a property slider and corresponding matrix automatically, or you can call the `make_bias_mat()` function using your own float bias value.

Using Shadow Maps

NVIDIA hardware provides an accelerated and filtered shadow format that's easy to use and very fast. The DirectX texture format to use is `D24S8_SHADOWMAP` and is a way to use the depth buffer from a shadow render. To declare such a shadow map, we need to declare a render target for both the depth (shadow) map and the (required, but usually un-used) color render target:

```
#define SHADOW_SIZE 256 /* or whatever size you need */
texture CTex : RENDERCOLORTARGET <
      float2 Dimensions = {SHADOW_SIZE,SHADOW_SIZE};
      string Format = "x8b8g8r8" ;
      string UIWidget = "None"; >;
sampler CSamp = sampler_state {
      texture = <CTex>;
      AddressU  = CLAMP; AddressV = CLAMP;
      MipFilter = NONE;
      MinFilter = LINEAR;
      MagFilter = LINEAR;
};
texture DTex : RENDERDEPTHSTENCILTARGET <
      float2 Dimensions = {SHADOW_SIZE,SHADOW_SIZE};
      string format = "D24S8_SHADOWMAP";
      string UIWidget = "None"; >;
sampler DSamp = sampler_state {
      texture = <DTex>;
      AddressU  = CLAMP; AddressV = CLAMP;
      MipFilter = NONE;
      MinFilter = LINEAR;
      MagFilter = LINEAR;
};
```

The shadowmap will be available to pixel shaders in the "DSamp" sampler.

As you might expect, the above construction is available as a "shadowMap.fxh" macro, called `DECLARE_SHADOW_MAPS()`. The "CSamp" declaration here may be gratuitous but is sometimes useful, such as when creating shadows from translucent colored objects.

Sample vertex and pixel shaders to create and call-upon such a shadow map are also provided in the header file. A good complete sample of usage to examine is "shadowSpot2.fx" in the standard FX Composer distribution. This sample uses the header-provided vertex shaders `shadowGenVS()` and `shadowUseVS()` in combination with pixel shaders declared in the .fx file.

Assigning render targets in HLSL requires the use of DX-SAS scripting. Since we are changing the view and clearing buffers, shaders that create and use shadow maps need to be declared as with a "ScriptClass" value of "scene."

Not all HLSL-compatible programs can read SAS scripts, however. For the sake of such programs (e.g., 3DSMax 7.0), it's recommended that your .fx file provide *two* versions of the shader – one that uses shadow maps, and one that does not. The SAS ScriptClass has a special value, "sceneorobject," specifically to accommodate such situations. Using "sceneorobject" will give your shader the widest applicability in multiple applications.

Here is the `STANDARDSGLOBAL` declaration for "shadowSpot2.fx":

```
float Script : STANDARDSGLOBAL <
      string UIWidget = "none";
      string ScriptClass = "sceneorobject";
      string ScriptOrder = "standard";
      string ScriptOutput = "color";
      string Script = "Tech=Technique?Shadowed:Unshadowed;";
> = 0.8; // version #

// The following global variables are values to be used
//   when clearing color and/or depth buffers
float4 ClearColor <
      string UIWidget = "color";
      string UIName = "background";
> = {0,0,0,0.0};

float ClearDepth <string UIWidget = "none";> = 1.0;

float4 ShadowClearColor <
      string UIWidget = "none";
> = {1,1,1,0.0};
```

Note the declaration of two different techniques, which will appear as choices in the FX Composer properties panel.

Following is the declarations of the first, "Shadowed" technique. Note the assignments of color and depth targets, and importantly the assignment of a `RenderPort` – this assignment tells the application (FX Composer) to use a light-style (square) view matrix, clipping planes, etc. On the first pass, we tell the app to use a lamp, and on the second pass we clear these values – this extra step is an important one to avoid confusion for the host.

```
technique Shadowed <
      string Script =    "Pass=MakeShadow;"
                         "Pass=UseShadow;";
> {
      pass MakeShadow <
          string Script =
            "RenderColorTarget0=ColorShadMap;"
            "RenderDepthStencilTarget=ShadDepthTarget;"
            "RenderPort=light0;"
            "ClearSetColor=ShadowClearColor;"
            "ClearSetDepth=ClearDepth;"
            "Clear=Color;"
            "Clear=Depth;"
            "Draw=geometry;";
      > {
            VertexShader = compile vs_2_0
                        shadowGenVS(WorldXf,
                                    WorldITXf,
                                    ShadowViewProjXf);
            ZEnable = true;
            ZWriteEnable = true;
            ZFunc = LessEqual;
            CullMode = None;
            // no pixel shader needed!
      }
      pass UseShadow <
          string Script =
            "RenderColorTarget0=;"
            "RenderDepthStencilTarget=;"
            "RenderPort=;"
            "ClearSetColor=ClearColor;"
            "ClearSetDepth=ClearDepth;"
            "Clear=Color;"
            "Clear=Depth;"
            "Draw=geometry;";
      > {
            VertexShader = compile vs_2_0
                        shadowUseVS(WorldXf,
                                    WorldITXf,
                                    WorldViewProjXf,
                                    ShadowViewProjXf,
                                    ViewIXf,
                                    ShadBiasXf,
                                    SpotLightPos);
            ZEnable = true;
            ZWriteEnable = true;
            ZFunc = LessEqual;
            CullMode = None;
            PixelShader = compile ps_2_a useShadowPS();
      }
}
```

Next is the "Unshadowed" technique. Since this is meant to be applied as an object surface material, rather than as a scene-wide command, it is much simpler. The various render targets and `RenderPort` *are* being explicitly cleared here, but that's

most to avoid potential problems if the app gets confused by the user switching back and forth between techniques.

```
technique Unshadowed <
      string Script = "Pass=NoShadow;";
> {
      pass NoShadow <
           string Script =
             "RenderColorTarget0=;"
             "RenderDepthStencilTarget=;"
             "RenderPort=;"
             "ClearSetColor=ClearColor;"
             "ClearSetDepth=ClearDepth;"
             "Clear=Color;"
             "Clear=Depth;"
             "Draw=geometry;";
      > {
             VertexShader = compile vs_2_0
                          shadowUseVS(WorldXf,
                                      WorldITXf,
                                      WorldViewProjXf,
                                      ShadowViewProjXf,
                                      ViewIXf,
                                      ShadBiasXf,
                                      SpotLightPos);
             ZEnable = true;
             ZWriteEnable = true;
             ZFunc = LessEqual;
             CullMode = None;
             PixelShader = compile ps_2_a unshadowedPS();
      }
}
```

In this case, we're still using the `shadowUseVS()` vertex shader – this is mainly to simplify the coding process. Some additional performace could be gained by writing a slightly simpler vertex shader, but since this works and is fast (there are very few apps that ever get limited by vertex-shader performance), it's a reasonable compromise for most shaders. In this case, we're also using the shadowing projection matrix for a second task – to project a "slide projector" RGB texture ("`SpotSamp`") that we use to shape the spotlight, rather than apply more-expensive math expressions in the shader.

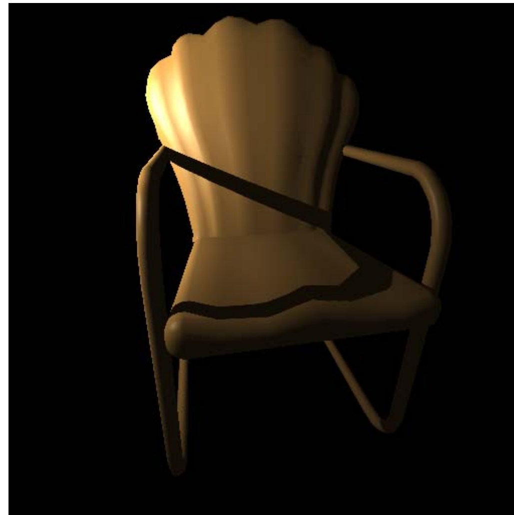
Here are the two different versions of the pixel shader, and a shared function that performs most of the lighting calculation:

```
void lightingCalc(ShadowingVertexOutput IN,
                  out float3 litContrib,
                  out float3 ambiContrib)
{
    float3 Nn = normalize(IN.WNormal);
    float3 Vn = normalize(IN.WView);
    Nn = faceforward(Nn,-Vn,Nn);
    float falloff = 1.0 / dot(IN.LightVec,IN.LightVec);
    float3 Ln = normalize(IN.LightVec);
    float3 Hn = normalize(Vn + Ln);
    float hdn = dot(Hn,Nn);
    float ldn = dot(Ln,Nn);
    float4 litVec = lit(ldn,hdn,SpecExpon);
    ldn = litVec.y * SpotLightIntensity;
    ambiContrib = SurfColor * AmbiLightColor;
    float3 diffContrib = SurfColor*
                         (Kd * ldn * SpotLightColor);
    float3 specContrib = ((ldn * litVec.z * Ks) *
                        SpotLightColor);
    float3 result = diffContrib + specContrib;
    float cone = tex2Dproj(SpotSamp,IN.LProj);
    litContrib =  ((cone*falloff) * result);
}

float4 useShadowPS(ShadowingVertexOutput IN) : COLOR
{
    float3 litPart, ambiPart;
    lightingCalc(IN,litPart,ambiPart);
    float4 shadowed = tex2Dproj(ShadDepthSampler,IN.LProj);
    return float4((shadowed.x*litPart)+ambiPart,1);
}

float4 unshadowedPS(ShadowingVertexOutput IN) : COLOR
{
    float3 litPart, ambiPart;
    lightingCalc(IN,litPart,ambiPart);
    return float4(litPart+ambiPart,1);
}
```

As you can see, the two pixel shaders are almost identical! The value of the shadowing is returned by a single `tex2Dproj()` call to `ShadDepthSampler`, using coordinates pre-calculated by the `shadowUseVS()` vertex shader. All of the BRDF-calculation work is done in the dedicated `lightingCalc()` function.



Simple object with shadow map

Special "Roll Your Own" Maps

It is also possible to use other texture formats for shadow mapping, though your shader will (a) have to render the depth as a color map and (b) will need to do any texture filtering in the shader, rather than through the dedicated hardware channel used by D24S8_SHADOWMAP.

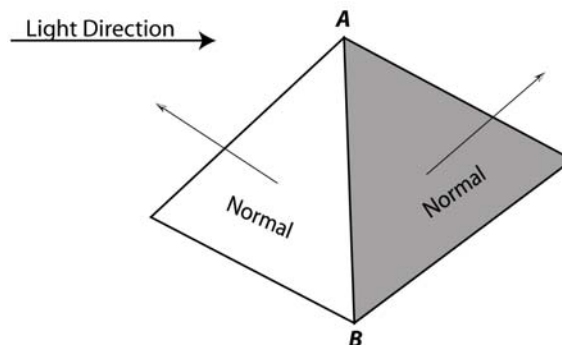
The .fx examples included with the NVIDIA SDK include several such samples, such as shaders that perform user-defined PCF filtering, perspective shadow maps, and blended multi-view soft shadows.

Shadow with wide PCF sampling and floating-point shadow map

Using Shadow Stencil Volumes

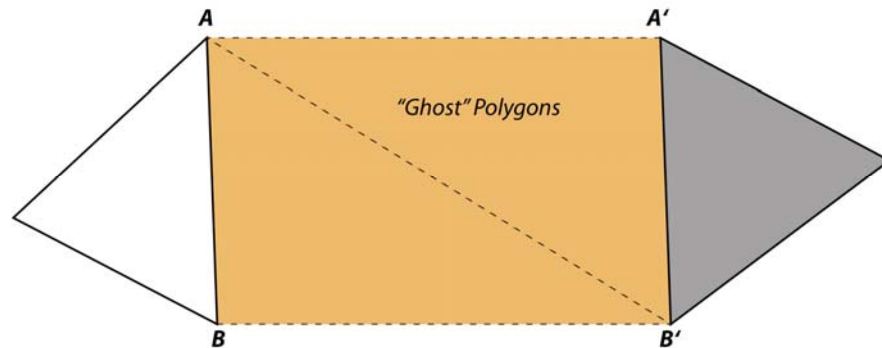
Using extruded shadow volumes and stenciling is a popular method for games, largely because it's fast, simple to implement, and will run on older hardware as well as the latest GPUs. There are some downsides to stenciling, particularly relating to the fact that stencil shadowing is immediate – that is, your program must clear the stencil, render the shadow volume to set the stencil, and then USE the stencil – all in that exact order, without much opportunity for saving the results for re-use later. This means that the shadowing item must be extruded and rendered, then all object receiving that shadow must be rendered, in a fairly strict order of operations (unlike shadow maps, which can be rendered and stored for use later, potentially on multiple objects or even across multiple frames of animation). For many games this is an acceptable compromise.

A second key issue is that the geometry in your scene (at least for shadow-casting objects) must consist of closed volumes, and must be constructed in a special way. Below, consider two polygons, that share a common edge marked AB. The surface normal of the lighter polygon faces the light, while the dark one faces away from the light.



To create a shadow volume, we will extrude the darker polygon away from the lighter one, pushing it away from the light source (to some distance, possibly infinity). For this purpose, the polygons can no longer share edge AB – the edge needs to be split so that the vertices of the each polygon don't distort the other during the shadow-extrusion process.

Further, the sides of the shadow volume need to also be rendered. So two new "ghost" triangles must be added: AA'B' and B'A'A.



For some shadow-extrusion schemes, these new triangles can be inserted by the CPU dynamically for each frame, and the vertex buffer completely refreshed.

Alternatively, without CPU cost but doubling the vertex count, the ghost polygons can be added to the model and always be left in-place! If no shadow extrusion occurs, the A will be coincident with A', B with B', and the area of the ghost triangles will be zero. The triangles will not rasterize so their cost to the pixel shading engine is also zero.

The extra vertices will require extra vertex shader processing, but happily we can use this to our advantage, letting the vertex shader itself do the extrusion step based simply on a light direction (or location of a point light source).

The "stencilVolume.fx" shader effect does just this, so that the CPU impact is minimal. To use this effect, we need to load a model that has already had the ghost polygons inserted.

For models in the .obj format, the SDK tool "splitPolyEdges.pl" is provided to insert such edges. The command

```
perl splitPolyEdges.pl myFile.obj > splitFile.obj
```

will read the moel in "myFile.obj" and write a split-edged copy into "splitFile.obj" containing the new polygons and vertices. Note that only polygons are converted – other .obj entities like patches are ignored since they are not use in shadow generation.

Here's a vertex shader for such a model, which does the extrusion work based on a fixed light position ("LightPos"). The value "GeomInset" defines a slight depth-offset value to avoid z-conflicts in the shadow (like shadow map biasing), added to avoid popping on polygons near the grazing angle.

```
float4 extrudeVS(float4 Position : POSITION,
                 float4 Normal : NORMAL0, // normalized
                 float4 LightPos,
                 float GeomInset,
                 float ShadowExtrudeDist,
                 float4x4 WorldViewProjXf) : POSITION
{
    // Create normalized vector from vertex to light
    float4 Ln = normalize(Position-LightPos);
    float ldn = -dot(Ln.xyz,Normal.xyz);
    float3 inV = Normal.xyz * GeomInset;
    float4 inset_pos = float4(
```

```
                    (Position.xyz - inV),Position.w);
    float4 extrusion_vec = Lvec * ShadowExtrudeDist;
    // if ldn < 0 then the vertex faces away from the light,
    //   so move it away from the light.
    float toggle = (float) (ldn < 0.0); // boolean as float
    float4 new_position = extrusion_vec*toggle + inset_pos;
    float4 HPosition = mul(new_position,WorldViewProjXf);
    return(HPosition);
}
```

The pixel shader is actually arbitrary, and can even be ignored for trivial cases. The stencil will suppress rendering *entirely* for pixels within the shadow, so whatever pixel shader is used need not worry about the shadowing algorithm.

The technique definition performs the complete shadowing effect as multiple passes:

```
technique shadowCW <
            string Script = "Pass=layZ;"
                                    "Pass=back;"
                                    "Pass=front;"
                                    "Pass=lighting;";
      > {
    pass layZ <
            string Script = "Draw=geometry;";
      > {
            VertexShader = compile vs_1_0 backingVS();
            ZEnable = true;
            ZWriteEnable = true;
            CullMode = None;
            StencilEnable = false;
    }
    pass back <
            string Script = "Draw=geometry;";
      > {
            VertexShader = compile vs_1_1 extrudeVS();
            ZEnable = true;
            ZWriteEnable = true;
            ZFunc = lessequal;
            CullMode = CW;
            StencilEnable = True;
            StencilPass = Keep;
            StencilFail = Keep;
            StencilZFail = IncrSat;
            StencilFunc = Always;
            ColorWriteEnable = 0;
    }
    pass front <
            string Script = "Draw=geometry;";
      > {
            VertexShader = compile vs_1_1 extrudeVS();
            ZEnable = true;
            ZWriteEnable = true;
            ZFunc = lessequal;
            CullMode = CCW;
            // TwoSidedStencilMode = false; // needed?
```

```
            StencilEnable = True;
            StencilPass = Keep;
            StencilFail = Keep;
            StencilZFail = DecrSat;
            StencilFunc = Always;
            ColorWriteEnable = 0;
    }
    pass lighting <
            string Script = "Draw=geometry;";
      > {
            VertexShader = compile vs_1_1 simpleVS();
            ZEnable = true;
            // ZWriteEnable = true;
            CullMode = None;
            StencilEnable = True;
            StencilPass = Keep;
            StencilZFail = Keep;
            StencilFail = Keep;
            StencilRef = 0;
            StencilFunc = Equal;
    }
}
```

Soft Stencil Shadows

A recent twist on stencil shadows can be found in the file "softStencilShadow.fx" – in this version, the stencil is created in the usual way but then rendered onto a white field with a black shadow, and saved to a texture.

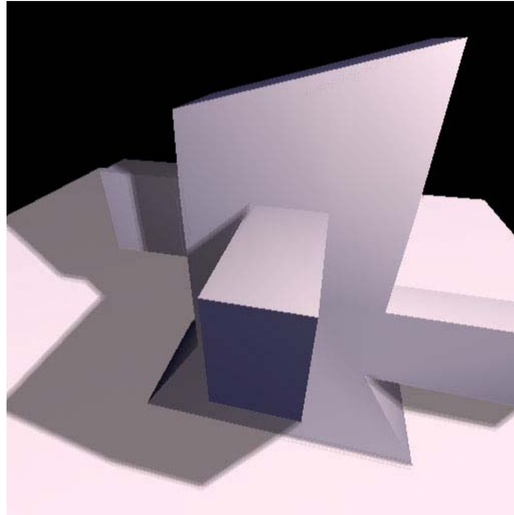
Another texture is rendered, with stored depths across the scene as a grayscale.

A third texture is rendered containing the overall un-shadowed color scene.

A blur pass is then run on the black and white shadow, with this caveat: when sampling neighboring texels for blur, both those texels and the corresponding depth texels are checked. If the depth value is significantly different from the depth of the texel the shader is currently blurring (based on some user-defined threshold), then that value is ignored. The result is that only texels on the same surface are blurred together, while edges between objects remain sharp.

This final "smart-blurred" image is then multiplied with the color-rendered scene to create the final rendered and shadowed picture.

The blur is uniform across screen space, so it is not "physically correct," but is less harsh than typical stencil shadowing, and for many scenes it is entirely adequate.



Soft stencil shadowing

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, and FX Composer 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

© 2005 NVIDIA Corporation. All rights reserved.

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