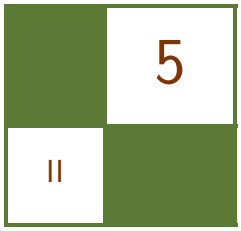

Volume Decals

Emil Persson

5.1 Introduction

Decals are often implemented as textured quads that are placed on top of the scene geometry. While this implementation works well enough in many cases, it can also provide some challenges. Using decals as textured quads can cause Z-fighting problems. The underlying geometry may not be flat, causing the decal to cut into the geometry below it. The decal may also overhang an edge, completely ruining its effect. Dealing with this problem often involves clipping the decal to the geometry or discarding it entirely upon detecting the issue. Alternatively, very complex code is needed to properly wrap the decal around arbitrary meshes, and access to vertex data is required. On a PC this could mean that system-memory copies of geometry are needed to maintain good performance. Furthermore, disturbing discontinuities can occur, as in the typical case of shooting a rocket into a corner and finding that only one of the walls got a decal or that the decals do not match up across the corner. This article proposes a technique that overcomes all of these challenges by projecting a decal volume onto the underlying scene geometry, using the depth buffer.

5.2 Decals as Volumes

5.2.1 Finding the Scene Position

The idea behind this technique is to render the decal as a volume around the selected area. Any convex volume shape can be used, but typical cases would be spheres and boxes. The fragment shader computes the position of the underlying geometry by sampling the depth buffer. This can be accomplished as follows:

```
// texCoord is the pixel's normalized screen position
float depth = DepthTex.Sample(Filter, texCoord);
float4 scrPos = float4(texCoord, depth, 1.0f);
float4 wPos = mul(scrPos, ScreenToWorld);
```

Figure 5.1. Example decal rendering.

```
float3 pos = wPos.xyz / wPos.w;
// pos now contains pixel position in world space
```

The `ScreenToWorld` matrix is a composite matrix of two transformations: namely the transformation from screen coordinates to clip space and then from clip space to world space. Transforming from world space to clip space is done with the regular `ViewProjection` matrix, so transforming in the other direction is done with the inverse of this matrix. Clip space ranges from -1 to 1 in x and y, whereas the provided texture coordinates are in the range of 0 to 1, so we also need an initial scale-bias operation baked into the matrix. The matrix construction code could look something like this:

```
float4 ScaleToWorld = Scale(2, -2, 1) *
    Translate(-1, 1, 0) * Inverse(ViewProj);
```

What we are really interested in, though, is the local position relative to the decal volume. The local position is used as a texture coordinate used to sample a volume texture containing a volumetric decal (see Figure 5.1). Since the decal is a volumetric texture, it properly wraps around nontrivial geometry with no discontinuities (see Figure 5.2). To give each decal a unique appearance, a random rotation can also be baked into the matrix for each decal. Since we do a matrix transformation we do not need to change the shader code other than to name the matrix more appropriately as `ScreenToLocal`, which is then constructed as follows:

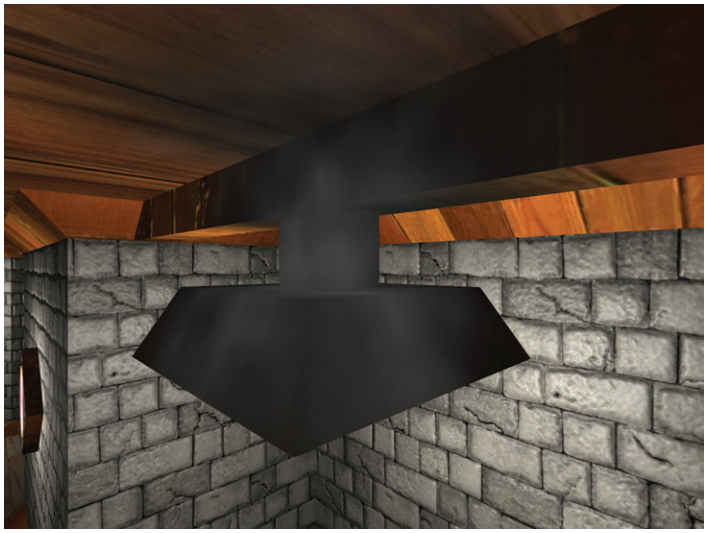

Figure 5.2. Proper decal wrapping around nontrivial geometry.

```
float4 ScreenToLocal = Scale(2, -2, 1) *
    Translate(-1, 1, 0) * Inverse(ViewProj) *
    DecalTranslation * DecalScale * DecalRotation;
```

The full fragment shader for this technique is listed below and a sample with full source code is available in the web materials.

```
Texture2D <float> DepthTex;
SamplerState DepthFilter;

Texture3D <float4> DecalTex;
SamplerState DecalFilter;

cbuffer Constants
{
    float4x4 ScreenToLocal;
    float2 PixelSize;
};

float4 main(PsIn In) : SV_Target
{

    // Compute normalized screen position
    float2 texCoord = In.Position.xy * PixelSize;
```

```
    // Compute local position of scene geometry
    float depth = DepthTex.Sample(DepthFilter, texCoord);
    float4 scrPos = float4(texCoord, depth, 1.0f);
    float4 wPos = mul(scrPos, ScreenToLocal);

    // Sample decal
    float3 coord = wPos.xyz / wPos.w;
    return DecalTex.Sample(DecalFilter, coord);
}
```

Listing 5.1. The full fragment shader.

5.2.2 Implementation and Issues

In a deferred-rendering system [Thibieroz 04] this technique fits perfectly. The decals can be applied after the *geometry buffer* (G-buffer) pass and the relevant attributes, such as diffuse color and specularity, can simply be updated, and then lighting can be applied as usual. This technique also works well with a light pre-pass renderer [Engel 09], in which case lighting information is readily available for use in the decal pass.

In a forward rendering system the decals will be applied after lighting. In many cases this is effective also, for instance, for burn marks after explosions, in which case the decals can simply be modulated with the destination buffer. With more complicated situations, such as blending with alpha, as is typically the case for bullet holes, for instance, the decal application may have to take lighting into account. One solution is to store the overall lighting brightness into alpha while rendering the scene; the decal can then pre-multiply source color with alpha in the shader and multiply with destination alpha in the blender to get reasonable lighting. This will not take light color into account, but may look reasonable if lighting generally is fairly white. Another solution is to simply go by the attenuation of the closest light and not take any normal into account. Alternatively, a normal can be computed from the depth buffer, although this is typically slow and has issues of robustness [Persson 09].

One issue with this technique is that it applies the decal on everything within the decal volume. This is not a problem for static objects, but if you have a large decal volume and dynamic objects move into it they will get the decal smeared onto them, for instance, if you previously blew a bomb in the middle of the road and a car is passing through at a later time. This problem can be solved by drawing dynamic objects after the decal pass. A more elaborate solution is to render decals and dynamic objects in chronological order so that objects that are moved after the decal is added to the scene will not be affected by the decal. This will allow dynamic objects to be affected by decals as well. Another solution is to use object IDs. The decal can store the IDs of objects that intersected the decal

volume at the time it was added to the scene and cull for discarded pixels that do not belong to any of those objects.

5.2.3 Optimizations

On platforms where the depth-bounds test is supported, the depth-bounds test can be used to improve performance. On other platforms, dynamic branching can be used to emulate this functionality by comparing the sample depth to the depth bounds. However, given that the shader is relatively short and typically a fairly large number of fragments survive the test, it is recommended to benchmark to verify that it actually improves performance. In some cases it may in fact be faster to not attempt to cull anything.

5.2.4 Variations

In some cases it is desirable to use a two-dimensional texture instead of a volume decal. Volume textures are difficult to author and consume more memory. Not all cases translate well from a two-dimensional case to three dimensions. A bullet hole decal can be swept around to a spherical shape in the three-dimensional case and can then be used in any orientation, but this is not possible for many kinds of decals; an obvious example is a decal containing text, such as a logo or graffiti tag.

An alternate technique is to sample a two-dimensional texture using just the x, y components of the final coordinates. The z component can be used for fading. When a volume texture is used, you can get an automatic fade in all directions by letting the texture alpha fade to zero toward the edges and using a border color with an alpha of zero. In the 2D case you will have to handle the z direction yourself.

Two-dimensional decals are not rotation invariant so when placing them in the scene they must be oriented such that they are projected sensibly over the underlying geometry. The simplest approach would be to just align the decal plane with the normal of the geometry at the decal's center point. Some problematic cases exist though, such as when wrapping over a corner of a wall. If it is placed flat against the wall you will get a perpendicular projection on the other side of the corner with undesirable texture-stretching as a result.

An interesting use of the two-dimensional case is to simulate a blast in a certain direction. This can be accomplished by using a pyramid or frustum shape from the point of the blast. When the game hero shoots a monster you place a frustum from the bullet-impact point on the monster to the wall behind it in the direction of the bullet and you will get the effect of blood and slime smearing onto the wall. The projection matrix of this frustum will have to be baked into the `ScreenToLocal` matrix to get the proper projection of the texture coordinates.

The blast technique can also be varied for a cube decal scenario. This would better simulate the effect of a grenade blast. In this case a cube or sphere would be rendered around the site of the blast and a cubemap lookup is performed with the final coordinates. Fading can be effected using the length of the coordinate vector.

To improve the blast effect you can use the normals of underlying geometry to eliminate the decal on back-facing geometry. For the best results, a shadowmapesque technique can be used to make sure only the surfaces closest to the front get smeared with the decal. This "blast-shadow map" typically has to be generated only once at the time of the blast and can then be used for the rest of the life of the decal. Using the blast-shadow map can ensure splatter happens only in the blast shadow of monsters and other explodable figures, whereas areas in the blast-shadow map that contain static geometry only get scorched. This requires storing a tag in the shadow buffer for pixels belonging to monsters, however. Creative use of the shadow map information also can be used to vary the blood-splatter intensity over the distance from the blast to the monster and from the monster to the smeared wall.

5.3 Conclusions

An alternate approach for decal rendering has been shown that suggests solutions to many problems of traditional decal-rendering techniques. Using volumes instead of flat decal geometry allows for continal decals across nontrivial geometry. It also eliminates potentially expensive buffer locks or the need for system-memory buffer copies.

Bibliography

[Thibieroz 04] Nicolas Thibieroz. "Deferred Shading with Multiple Render Targets." In *ShaderX^2: Shader Programming Tips and Tricks with DirectX 9*, edited by Wolfgang Engel, pp. 251–269. Plano, TX: Wordware Publishing, 2004.

[Engel 09] Wolfgang Engel. "Designing a Renderer for Multiple Lights - The Light Pre-Pass Renderer." In *ShaderX^7: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 655–666. Hingham, MA: Charles River Media, 2009.

[Persson 09] Emil Persson. "Making It Large, Beautiful, Fast and Consistent: Lessons Learned Developing Just Cause 2." in *GPU Pro: Advanced Rendering Techniques*, pp. 571–596. Natick, MA: A K Peters, 2010.