20

A Deferred Decal Rendering Technique

Jan Krassnigg

University of Aachen, Germany

Overview

Rendering decals is a common method used to apply more detail to 3D worlds dynamically. Decals are often used to add bullet holes, blood stains, tire marks, and similar items to a world as events occur in a game, but they can also be used by level designers to enrich the environment with wear and tear textures, dirt textures, signs on walls, etc.

This gem presents a decal rendering technique that uses deferred shading to produce scenes like the one shown in Figure 20.1. The technique is entirely shader based, extremely lightweight on the CPU, does not need to dynamically generate triangles, and is a straightforward addition to most 3D engines.

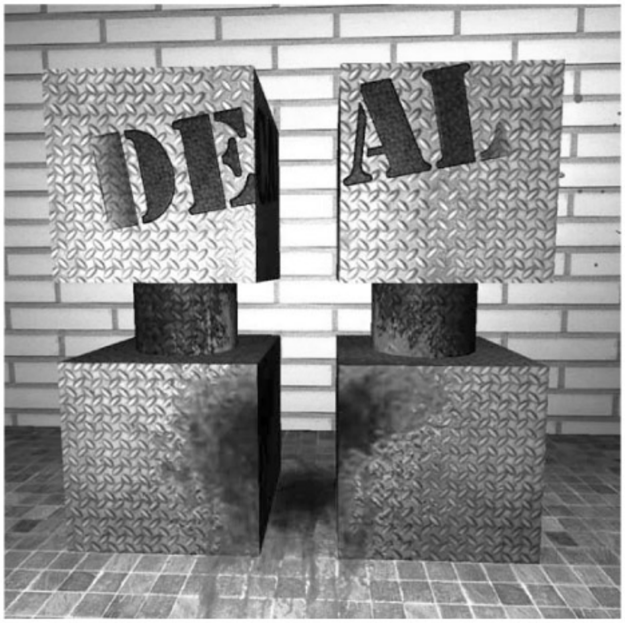

Figure 20.1: Decals applied to complex geometry.

20.1 The Problem

There are several things that a good decal system should solve:

1. The system should integrate well with lighting. Decals should not only change the color, but also change the normal, specular factors, and any other surface parameters such that they become indistinguishable from all other geometry.
2. Decals should work convincingly on all surfaces, static and dynamic.
3. Decals need to clip properly to geometry boundaries, and possibly even wrap around corners.
4. The system needs to work with geometry that might be very different from the geometry used for collision detection.

Item 1 in this list is easily solvable when the graphics engine includes deferred rendering capabilities [2]. For forward renderers, the system in this gem can still be used, but some modifications will be necessary. For a thorough explanation of deferred shading, please refer to [3,4,5,6].

Item 2 can be solved if we can get our hands on a free 8-bit channel in the G-buffer. If not, we can at least make it work on static geometry. This is discussed in Section 20.5.

Item 3 is where the real problems begin. The decal needs to follow the surface to which it is applied, even if that surface is highly tessellated. Some existing decal rendering techniques generate a triangle representation for a decal on such a surface [1], but this can be computationally expensive. Furthermore, the raw mesh is often not available at all on the CPU since the mesh data is loaded into a vertex buffer accessible only by the GPU at a reasonable speed.

Item 4 is an issue because today's games often use very complex meshes for rendering, but a less detailed mesh might be used for collision detection. The difficulty is that the point of intersection that a ray cast returns might differ quite a bit from the location where the decal must be rendered.

20.2 The General Idea

The basic solution afforded by our technique is to project a decal onto a surface using a special fragment shader applied to the decal's bounding volume instead of rendering new decal polygons on top of the scene geometry. The only information required to achieve this consists of the position and orientation of the decal, the decal's size, and a texture containing depth values for the viewport being rendered. This information allows all computation to be done in the vertex and fragment shaders.

When rendering a decal, it is natural for us to work in "decal space", where the *x*-

and *y*-axes lie in the tangent plane to the underlying surface at the decal's center, and the *z*-axis is parallel to the surface's normal vector at the decal's center. In order to move data into the decal's local coordinate system, the shader needs to be supplied with the inverse of the decal's transformation matrix.

The code in Listing 20.1 first fetches the depth from the G-buffer at the fragment position and uses it to reconstruct the 3D world-space position of the fragment. The worldToDecal constant is the inverse of the transformation matrix for the decal that we are currently rendering. With this matrix, we can transform the fragment's position into the local space of the decal. This local position can then be used to compute texture coordinates at which the decal texture is sampled. In this first version, we are using the (*x*,*y*) position only, which simply projects the decal along its local *z*-axis onto the underlying geometry, as shown in Figure 20.2.

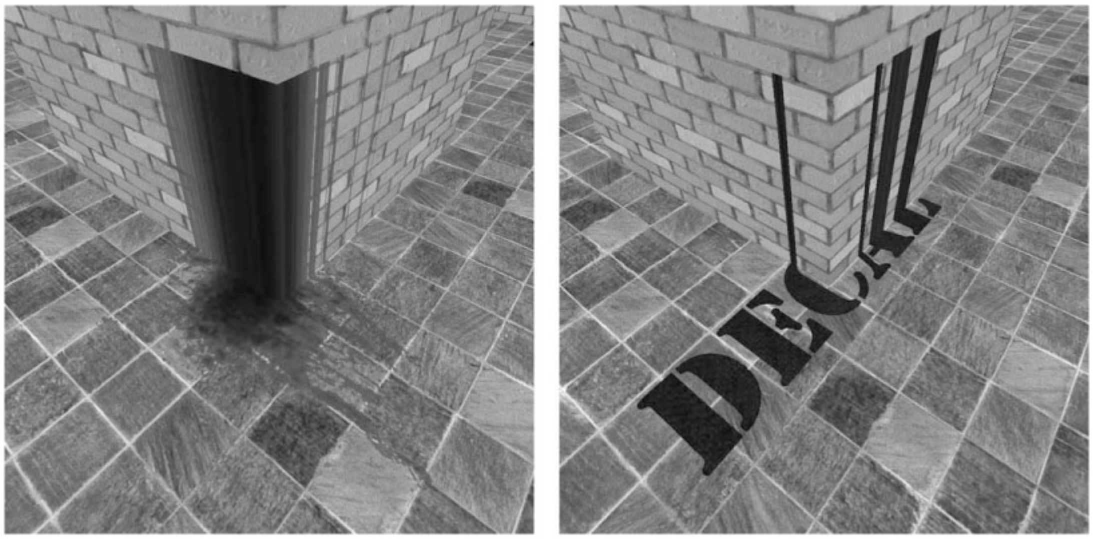

Figure 20.2: Using a simple projection onto the local *x*-*y* plane causes decals to be smeared in the direction of the local *z*-axis.

Listing 20.1: This code determines the decal-space coordinates for the fragment being rendered and transforms them into texture coordinates for the decal. The RT_Depth texture contains depth values for the viewport, the worldToDecal constant is the inverse of the decal's 4×4 matrix transform from decal space to world space, and the recipDecalSize constant is 1/s, where s is the size of the decal in the scene.

```
// Sample the depth at the current fragment
float pixelDepth = texture2DRect(RT_Depth, gl_FragCoord.xy).x;

// Compute the fragment's world-space position
vec3 worldPos = ComputeWorldSpacePosition(gl_FragCoord.xy, pixelDepth);

// Transform into decal space
vec3 decalPos = worldToDecal * worldPos;

// Use the xy position of the position for the texture lookup
vec2 texcoord = decalPos.xy * recipDecalSize * 0.5 + 0.5;

// Fetch the decal texture color
gl_FragColor = texture2D(diffuseDecalTexture, texcoord);
```

20.3 Geometry Rendering

Now that we have some code that calculates basic texture coordinates, we must consider what kind of geometry we actually need to render. Each fragment rendered with the decal shader acts like a "window" through which we can possibly see our decal. So we could just render a surface-aligned quad corresponding to the size and position of the decal. However, we want our decal to have depth, so we instead render a bounding cube as shown in Figure 20.3. This allows us to see the decal from any direction, and it will allow us to add a wrap-around feature later on.

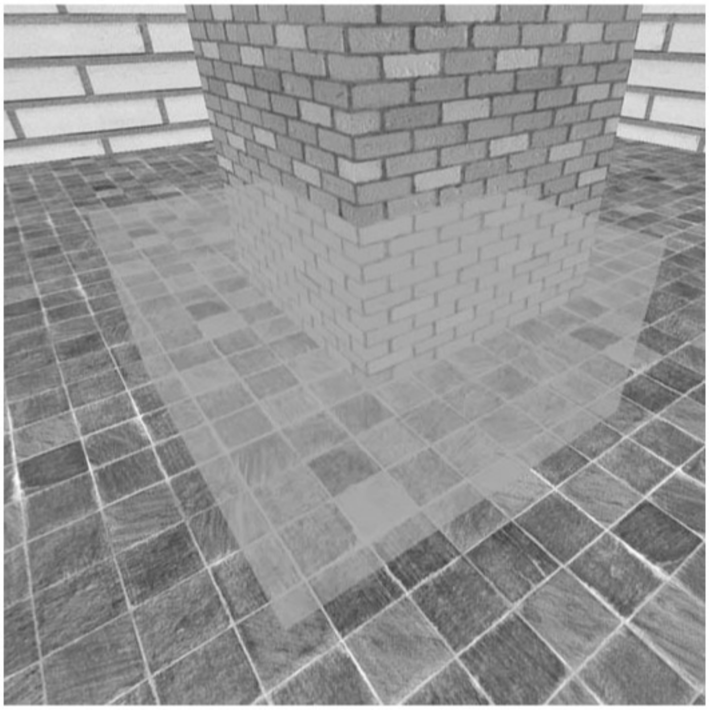

Figure 20.3: A bounding cube centered on a decal can be rendered to ensure that we capture the decal's influence from any viewpoint.

As an optimization for scenes containing a large number of decals, it can be advantageous to render decals through hardware instancing [7]. Therefore, it is a good idea to simply render a unit cube and transform it to the correct size and position in the vertex shader, as demonstrated in Listing 20.2. Note that we can easily extend the uniform constants `decalSize` and `decalToWorld` to arrays and use `gl_InstanceID` to render a batch of decals through instancing.

Listing 20.2: This vertex shader scales a unit cube to the actual size of the decal and translates it to the decal's world-space position.

```
uniform float   decalSize;
uniform mat4    decalToWorld;

// scale the unit cube and position it in world space
vec4 worldPos = decalToWorld *
  vec4(gl_Vertex.xyz * decalSize, gl_Vertex.w);

// output the position in homogeneous clip space
gl_Position = gl_ModelViewProjectionMatrix * worldPos;
```

For small decals, this technique works very well. However, when a cube is rendered with backface culling enabled and the depth test set to `GL_LESS`, the decal disappears the moment the camera enters the cube. One solution to this problem is to cull front faces and set the depth test to `GL_GREATER`. This way, only fragments whose world positions are actually behind surfaces are rendered, but it causes many fragments to be rendered unnecessarily when they are occluded by geometry closer to the camera.

Another solution is to find the corner of the cube that is closest to the camera and render a camera-aligned quad at that depth that is large enough to enclose the entire cube. If the closest corner is in front of the near plane, a full-screen quad should be rendered at the near plane instead.

20.4 Fade Out And Wrap-Around

We have a basic vertex shader and fragment shader in place, but our projection is infinite along the decal's *z*-axis, producing the smearing shown in Figure 20.2. There are two methods we can use to fix this problem. The simpler method is to use the distance from the fragment's position to the decal plane as a fade-out parameter. This distance is already available in `decalPos.z`, and we just have to scale it to the size of the decal as demonstrated in Listing 20.3.

Listing 20.3: The absolute value of the decal-space z-coordinate of the fragment position is scaled to the size of the decal and used as a fade-out parameter for the decal color. As before, the recipDecalSize constant is 1/s, where s is the size of the decal in the scene.

```
// compute the distance of the fragment to the decal's plane
float distance = abs(decalPos.z);

// scale the distance into the [0,1] range
// according to the size of the decal
float scaledDistance = max(distance * recipDecalSize * 2.0, 1.0);

// somehow use that scaled distance to fade out
// here: simple linear fade out
float fadeOut = 1.0 - scaledDistance;

vec4 diffuseColor = texture2D(diffuseDecalTexture, texcoord);

gl_FragColor = vec4(diffuseColor.rgb, diffuseColor.a * fadeOut);
```

This method is useful when a decal is supposed to be flat without wrapping around geometry. The exact formula used to fade out a decal can be varied to produce the best look for different decals, and it is advisable to make this configurable within the engine.

A more interesting method for handling the smearing problem is to make a decal wrap around corners and follow the curvature of complex surfaces. This is especially useful for blood stains and other liquids that splatter because it is much more convincing if such a decal covers an entire surface independently of its curvature.

This is quite easy to achieve. All we need is the surface normal at the position of each fragment in the decal. If we rotate that normal into decal space, its $(x\ y)$ components give us the gradient of the surface relative to the decal plane. We can use this gradient and the fragment's distance to the decal plane to modify the texture coordinates. In areas with no relative slope, the texture lookup remains unchanged, but in areas with a large slope (for example, at corners), the texture coordinates move outward according to the distance to the decal plane. This technique is illustrated in Listing 20.4, and the result is shown in Figure 20.4.

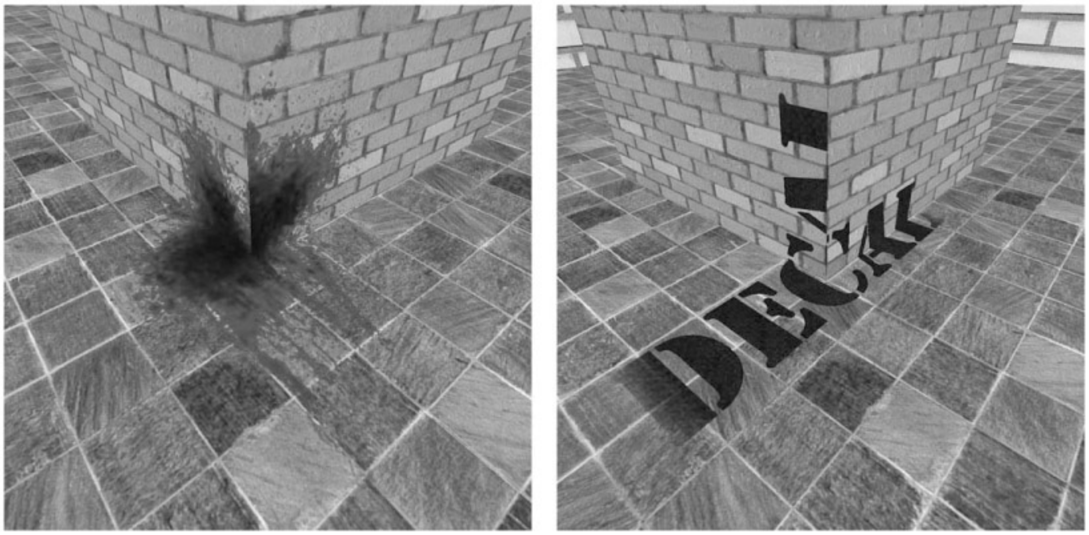

Figure 20.4: The decals wrap around corners based on the surface normals, and they fade out based on the distance from the decal plane.

Listing 20.4: This code demonstrates how the normal of the underlying surface can be used to adjust texture coordinates in such a way that a decal wraps around curves and corners. The RT_Normal texture contains normal vectors for the viewport encoded in the RGB channels.

```
// get the world-space normal at the fragment position
vec3 worldNormal = texture2DRect(RT_Normal, gl_FragCoord.xy).xyz;

// rotate it into the local space of the decal
vec3 decalNormal = (worldToDecal * vec4(worldNormal.xyz, 0.0)).xyz;

vec2 texcoord = decalPos.xy;
```

```
// use the distance and gradient to modify the texture lookup
texcoord -= decalPos.z * decalNormal.xy;

// scale and center the texture coordinates
texcoord += vec2(decalSize);
texcoord *= recipDecalSize * 0.5;

gl_FragColor = texture2D(diffuseDecalTexture, texcoord);
```

For the finishing touch, we add color fading that depends on the distance to the decal plane. We also need to account for the angle between the decal plane's normal and the underlying surface normal, since otherwise, decals appear on the back side of thin walls. A complete example shader with more details can be found on the accompanying CD.

20.5 Surface Clipping

The technique we have described works by applying decals in viewport space and does not differentiate among surfaces onto which it is projected. By using the projection and wrap-around method, it is possible to attach decals to any kind of surface, even to animated characters. However, a decal is never clipped, meaning that a decal attached to a box also projects onto the ground on which the box is resting. If the box moves and carries the decal along with it, then the projection on the ground moves as well.

There are two possible solutions to this problem. One solution is to restrict decal rendering to pixels covered by static geometry only. In this case, we need to render all static geometry first, then apply decals, and only afterward, render dynamic objects.

The second solution requires that an additional channel in the G-buffer be used to hold a "decal ID" for every distinct object that is rendered. In the rendering pass that fills the G-buffer, we write each object's decal ID along with the diffuse color, normal, etc., so that we have a per-pixel mask identifying to which object each pixel belongs. When a decal is applied to an object, we look up the object's ID and associate it with the decal. When the decal is rendered, we pass this ID along as an additional uniform constant and compare it to the ID read from the G-buffer. If the two match, then we know that the pixel belongs to the object to which the decal is attached, and we render normally. Otherwise, we discard the fragment because it would be drawn outside the set of pixels covered by the object.

If no distinction needs to be made among static geometries, then they can all share the same ID, such as zero. All dynamic objects should have different IDs, but those IDs don't need to be unique. All we need to do is make it very improbable that two dynamic objects near each other have the same ID. It then suffices to use an 8-bit channel and simply give the dynamic objects random IDs from the range [1,255].

This kind of object ID management could also be done using the stencil buffer. However, testing stencil values prevents us from using instancing to render the decals due to the fact that we cannot pass the stencil compare value as a uniform constant to the shader and change the stencil test on a per-fragment basis.

20.6 Limitations

There are a few limitations one should be aware of. This technique does not magically create volumetric decals. It only improves a 2D projection so that it looks more realistic in many situations. The wrap-around feature uses the normal of the underlying surface to change the texture coordinates inside a decal. The results shown in Figure 20.4 look much better than in Figure 20.2, but there is still a clearly visible cut at the edge of the vertical column. To prevent such artifacts, one would need to use truly volumetric decals. Figure 20.5 shows a decal applied to a more complex object. In the left image, the object uses face normals, and in the right image, it uses smooth normals.

In both images, no normal mapping is considered. Obviously, the normal of the underlying surface has a big effect on the appearance of the decal. Consequently, surfaces with strong normal mapping tend to distort the decal, sometimes causing visual artifacts. For some kinds of decals, this is less of an issue, though, because a blood stain usually still looks like a blood stain no matter how distorted it becomes when applied to a surface.

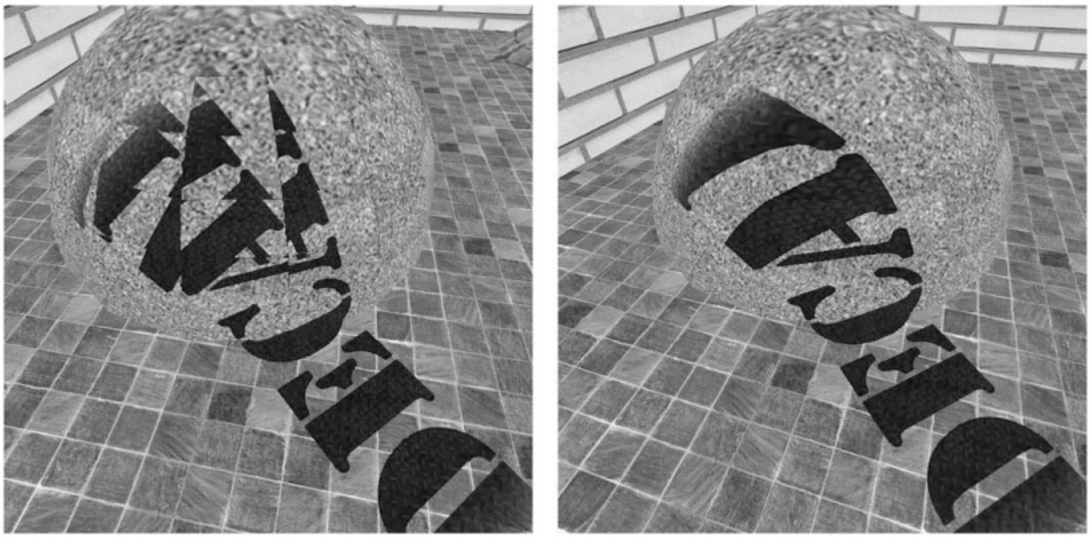

Figure 20.5: (Left) Wrap-around based on face normals. (Right) Wrap-around based on smoothly interpolated vertex normals.

A word about performance: It is easy to cull decals by testing bounding volumes, and the GPU can render decals in large batches through instancing. However, you should be careful not to have many decals layered on top of each other because doing so can easily consume all available fragment-rendering power. You can try to prevent such situations by removing decals after a certain amount of time or by restricting the number of decals on screen at once. The latter approach allows us to keep many decals in the world as long as they are not visible at the same time. Most games simply clean up the scene by removing decals after a fixed amount of time, but lose a lot of immersive quality in the process. The game *Max Payne* places a limit on the number of decals created in a single area, which has the great effect that if the player comes back to a room where he was previously, all the blood stains and bullet holes are still there.

20.7 Additional Features

Since we are already rendering the decals into the G-buffer before any lighting occurs, we can not only change the diffuse color, but can also replace (or modify) the normal, gloss, and other data as we wish. Computing the tangent and bitangent vectors for the decal is very straightforward in the vertex shader, and no additional vertex attributes are needed. For more information, see the example shader on the accompanying CD.

References

[1] Eric Lengyel."Applying Decals to Arbitrary Surfaces". *Game Programming Gems 2*, Charles River Media, 2001.

[2] Joris Mans and Dmitry Andreev."An Advanced Decal System". *Game Programming Gems 7*. Charles River Media, 2008.

[3] Oles Shishkovtsov."Deferred Shading in S.T.A.L.K.E.R.". *GPU Gems 2*. Addison-Wesley, 2005.

[4] Frank Puig Placeres."Fast Per-Pixel Lighting with Many Lights". *Game Programming Gems 6*. Charles River Media, 2006.

[5] Rusty Koonce."Deferred Shading in Tabula Rasa". *GPU Gems 3*. Addison-Wesley, 2008.

[6] Dean Calver."Deferred Lighting on PS 3.0 with High Dynamic Range". *ShaderX3*. Charles River Media, 2005.

[7] Francesco Carucci."Inside Geometry Instancing". *GPU Gems 2*. Addison-Wesley, 2005.