Rendering Techniques in Split/Second

Jeremy Moore\textsuperscript{1}
David Jefferies\textsuperscript{2}

\textbf{Figure 1.} A still frame from Split/Second.

\textsuperscript{1} jeremy.moore@disney.com
\textsuperscript{2} david.jefferies@disney.com
1. Introduction

*Split/Second* is *Black Rock Studio's* second game, due for release in early 2010 on Microsoft XBox® 360, Sony Playstation® 3 and PC. It is an action-packed street racing game with the emphasis heavily on action. Like racing through an action movie, the track around the players dynamically changes while they race through it. Massive environmental destruction opens new routes and creates dynamic obstacles to be navigated. Triggered by the drivers in the race, the city is the player’s weapon as they aim to take out opponents with remotely detonated explosions, bringing down bridges, toppling buildings, even crash landing jumbo jets.

The art direction behind *Split/Second* demands that layered and choreographed visual effects feature heavily and combine well with cinematic dynamic lighting to tell the game’s story.

In this chapter we outline some of the rendering approaches used by *Black Rock Studio* during the making of *Split/Second*. The work presented here is based upon techniques developed in the wider graphics community. We will cover the specific use of these techniques, the choices and trade-offs that were made during development, and describe some of our own innovations.

2. Deferred Shading in Split/Second

The art direction for *Split/Second* demands layered and choreographed visual effects to create an interactive graphical look similar to a summer blockbuster movie. This requires many lights, high detail geometry and post processing effects such as depth-of-field and motion blur. These requirements led us to believe that a deferred shading architecture was the right choice for its graphics pipeline.

For *Split/Second* the benefits of using deferred shading are:
- decoupling lighting cost from geometry complexity
- decoupling the material and lighting code
- advanced post-processing using the G-Buffer
- only lighting each pixel sample once per light

2.1. Previous Work

The use of a deferred shading architecture is increasingly common in games. A number of papers have been written describing implementation details for real-time engines such as [HARGREAVES04], [SHISHKOVTSOV05] and [KOONE07].
More recently [ENGEL09] introduced the light pre-pass renderer which only stores depth and normal values in the G-Buffer before performing a lighting pass to store light properties into a light buffer.

For Split/Second we considered the use of light pre-pass renderer but decided that, due to the high geometric complexity of our scenes, the overhead of two geometry passes outweighed the storage and memory bandwidth benefits.

## 2.2. The Split/Second G-Buffer

A deferred shading architecture decouples the lighting of the scene from the rendering of the geometry. Initially, the scene’s geometry is rendered into a deep frame buffer or G-Buffer. In the case of Split/Second the G-Buffer consists of three 32-bit color render targets and a depth target. The structure of the Split/Second G-Buffer is described in Table 1 and Table 2.

In Split/Second the G-Buffer is also used to support advanced post processing effects. For example the per-pixel motion vectors are used for motion blur and the surface normals are used to calculate screen space ambient occlusion.

<table>
<thead>
<tr>
<th>Render Target</th>
<th>Channel 1</th>
<th>Channel 2</th>
<th>Channel 3</th>
<th>Channel 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Albedo.r</td>
<td>Albedo.b</td>
<td>Albedo.g</td>
<td>Unlit.r</td>
</tr>
<tr>
<td>2</td>
<td>Normal.x</td>
<td>Normal.y</td>
<td>Normal.z</td>
<td>Unlit.b &amp; Edge</td>
</tr>
<tr>
<td>3</td>
<td>Unlit.g</td>
<td>Specular</td>
<td>Motion.x</td>
<td>Motion.y</td>
</tr>
</tbody>
</table>

*Table 1. The default G-Buffer layout for Split/Second*

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albedo</td>
<td>RGB albedo color at the pixel</td>
</tr>
<tr>
<td>Unlit</td>
<td>RGB color that won’t be lit in the lighting pass</td>
</tr>
<tr>
<td>Normal</td>
<td>Surface normal with 8bits per component</td>
</tr>
<tr>
<td>Specular</td>
<td>Specular power packed in 2bits with Specular intensity packed in 6bits</td>
</tr>
<tr>
<td>Motion</td>
<td>Screen space motion vector</td>
</tr>
<tr>
<td>Edge</td>
<td>1bit value representing whether the pixel is on a triangle edge</td>
</tr>
</tbody>
</table>

*Table 2. Attributes used in the G-Buffer for Split/Second*
2.3. Deferred Shading with MSAA

A conceptually simple form of screen-based anti-aliasing is full screen anti-aliasing (FSAA). With this technique the scene is rendered at a high resolution and then sampled down to the target resolution. Take for example a typical console resolution of 1280x720. The scene could be initially rendered at a resolution of 2560x1440 and each pixel, or sample, would be averaged with its neighbors to produce the final image. In a simple renderer each sample would consist of a color and depth component. In a deferred shading renderer each sample also contains the G-Buffer information.

Multisample anti-aliasing is a variation of full screen anti-aliasing that is supported by most current consumer graphics hardware. It reduces processing time by only running the pixel shader once per pixel, while still supersampling the depth.

Early graphics hardware that supported multisample anti-aliasing only provided a fixed function resolve operation. This is unsuitable for deferred shading architectures since the linear blend operation of the fixed function resolve is not applicable for many quantities such as the surface normal or material ID.

Current generation consoles and Microsoft DirectX® 10 level PC hardware give access to the individual samples generated by multisample anti-aliasing. This enables us to consider all samples individually during our lighting pass before blending the results.

This introduces a performance cost when using deferred shading with multisample anti-aliasing since the number of calculations in the deferred lighting pass will increase linearly with the number of samples per pixel.

However, when using multisample anti-aliasing, a significant proportion of pixels on the screen are likely to hold precisely the same values for each of their samples. For example, a typical scene in Split/Second has only 15% of the final rendered pixels at polygon edges, meaning that the other 85% of pixels do not require individual lighting contributions from each sample.

We make the observation that if we can determine and isolate the pixels which require us to consider both available samples, then we can substantially reduce the amount of work required in the deferred lighting pass.

2.4. Centroid Sampling

During multisample anti-aliasing vertex attributes are interpolated to the pixel center by default. If the pixel center itself is not covered, the vertex attributes are extrapolated beyond the edges of the rendered primitive. This can lead to visual artifacts when a texture is sampled at texels outside of the primitive’s UV boundaries. These texturing
artifacts are visible when using point filtering with multisampling because multisampling extends the rasterized area of the triangle to include all pixels in which sampling points are covered, even when the pixel center is not.

The solution to this problem is to use centroid sampling, which adjusts the position used for determining polygon color to be the center of all the sampling points covered by the polygon. This guarantees that a centroid-sampled location will always lie within the polygon being rendered. In Figure 2 it can be seen that the red sampled location is adjusted to lie within the polygon by centroid sampling.

![Figure 2. Multisampling (left) versus centroid multisampling (right).](image)

Centroid sampling is available in HLSL pixel shader model 2.0, and is activated for an interpolator using the `__CENTROID` semantic.

### 2.5. Approach on Split/Second

On *Split/Second* we use two techniques to determine whether a pixel has unique samples to consider in the deferred lighting pass.

#### 2.5.1. Edge Detection from Centroid Sampling

We can determine which pixels are affected by multisample anti-aliasing by interrogating the value of the centroid sample from the pixel shader. If a triangle covers only a single sample in a pixel then the centroid sample value will be offset from zero.

We store whether the centroid sample value is non-zero as a single bit in the G-Buffer. We use the top bit of the blue channel of the Unlit color which reduces the resolution of this channel to 7 bits. The blue channel was chosen because this is the color that the human eye is least sensitive to. *Listing 1* gives example shader code for this operation. We later use this value to determine whether to consider one or more samples in the deferred lighting pass. *Figure 3* shows a screenshot with the tagged edge pixels highlighted.
2.5.2. Edge Detection from G-Buffer Normal

There is a further less common condition for which the value of individual samples need to be considered. That is when two or more polygons intersect at a pixel. We can determine whether this is the case by comparing the normal at the individual samples.

```
struct PSInput
{
    float4 vPos : TEXCOORD0;
    float4 vPosCentroid : TEXCOORD1_CENTROID;
};

float4 main( PSInput In ) : COLOR
{
    float2 vEdge = In.vPosCentroid.xy - In.vPos.xy;
    float fEdge = (vEdge.x + vEdge.y == 0.0f) ? 0.0f : 1.0f;
    // For deferred shading we would usually pack this value into
    // one bit in the G-Buffer. (Maybe along with material ID?)
    return float4( fEdge );
}
```

Listing 1. An example pixel shader that uses centroid sampling to mark the pixels affected by multisample anti-aliasing

Figure 3. Screenshot showing pixels in red that have been detected as requiring individual lighting from all MSAA samples.
2.5.3. Depth/Stencil Repopulate Pass

On Split/Second we repopulate the depth/stencil buffer and z-cull memory after the multisample anti-aliasing geometry pass is complete. During repopulation we mark our edge pixels in the stencil buffer. They can then be masked out for some or all of the deferred lighting passes. Figure 4 shows the state of the stencil buffer containing the marked edge pixels.

On Split/Second we also determine whether a pixel is part of the sky by testing if the value of the normal in the G-Buffer is zero. We mark sky pixels in the stencil buffer so that they can be rejected in the deferred lighting pass.

![Figure 4](image.png)

*Figure 4.* Screenshot showing the contents of the stencil buffer used during the deferred lighting pass

2.5.4. Deferred Lighting Pass

The deferred lighting pass treats each set of G-Buffer samples individually. During lighting accumulation we carry out one set of lighting passes using the first sample. We then use the stencil mask to only calculate and blend the results of the lighting passes for other samples where necessary.

2.5.5. Performance

Sample performance timings for the Xbox® 360 are given in Table 3.
3. Deferred Shadowing in Split/Second

In *Split/Second* we use parallel-split shadow maps [ZSXL06] to generate sunlight shadows. We also support local shadow maps for a fixed number of scene spot-lights.

We use a deferred shadowing implementation similar to that described in [MITTRING07]. Using the shadow maps along with the depth buffer generated in the geometry pass, a deferred shadowing pass creates a screen space shadow mask which is used for attenuation in the deferred lighting pass.

We highlight here some optimizations which we employ when rendering the screen space shadow mask.

### 3.1. Shadow Edge Detection

To avoid aliasing artifacts commonly seen in shadow map rendering we use hardware accelerated percentage closer filtering. This takes a number of samples from a shadow map, performs a depth test for each of them against a shadow receiver, and averages the results [Rsc87].

Percentage closer filtering is a costly operation due to the additional texture samples required. When using deferred shadowing we have the benefit that we only pay this cost once per screen space pixel. However, we observe further that we only want to pay the cost of percentage closer filtering in the areas of the screen that border shadowed and non-shadowed areas.

We can see that if we can divide the screen into 3 areas:

- areas definitely in shadow
- areas definitely not in shadow
- areas that may be in or out of shadow

Then we need to only apply percentage closer filtering in the last of these areas.

---

**Table 3.** Performance timings for deferred lighting pass

<table>
<thead>
<tr>
<th>Operation</th>
<th>Without Centroid Optimization</th>
<th>With Centroid Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build stencil mask</td>
<td>-</td>
<td>0.3 ms</td>
</tr>
<tr>
<td>Light fragment 0</td>
<td>2.3 ms</td>
<td>2.3 ms</td>
</tr>
<tr>
<td>Light fragment 1 and blend into frame buffer with AA</td>
<td>2.3 ms</td>
<td>0.7 ms</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>4.6 ms</strong></td>
<td><strong>3.3 ms</strong></td>
</tr>
</tbody>
</table>
In *Split/Second* we use a multi-pass approach. We first determine which pixels fall into each of the above three classifications. We then use this information to reduce the number of pixels that require percentage closer filtering when creating our screen space shadow mask. The order of the render passes is shown in *Figure 5*.

![Diagram showing the order of render passes when creating the screen space shadow mask.](image)

### 3.1.1. Shadow Edge Passes

We first render a one quarter screen sized shadow edge mask. This is rendered with the same approach as for a standard deferred shadow mask, except that we write only values of 0, 0.5 or 1.0 to represent no shadow, possible shadow and in-shadow states. Example shader code to do this is given in *Listing 2*.

We then downsize this buffer to a one sixteenth sized shadow edge mask using shader code like that given in *Listing 3*. This pass thickens the "edge" areas of the mask so as to give a conservative estimate of the screen areas that require percentage closer filtering. It also reduces the size of the texture used as input to the final shadow mask pass which reduces texture bandwidth. *Figure 6* shows the two shadow edge masks.
float ShadowEdgeValue(
    in float3 vShadowCoord,  // pixel depth in light space
    in sampler2D shadowMapTexture,  // shadow map
    in float fFilterScale )  // size of edge filter
{
    // calculate sample offsets using based on fixed values in SampleOffsets
    float2 vOffset0 = vShadowCoord.xy + SampleOffsets[0].xy * fFilterScale;
    float2 vOffset1 = vShadowCoord.xy + SampleOffsets[0].zw * fFilterScale;
    float2 vOffset2 = vShadowCoord.xy + SampleOffsets[1].xy * fFilterScale;
    float2 vOffset3 = vShadowCoord.xy + SampleOffsets[1].zw * fFilterScale;

    // sample shadow map (ReadDepth defined elsewhere)
    float4 Depths0;
    Depths0.x = ReadDepth( shadowMapTexture, vOffset0.xy );
    Depths0.y = ReadDepth( shadowMapTexture, vOffset1.xy );
    Depths0.z = ReadDepth( shadowMapTexture, vOffset2.xy );
    Depths0.w = ReadDepth( shadowMapTexture, vOffset3.xy );

    // sum boolean results of the 4 samples
    float4 Attenuation0 = step( vShadowCoord.z, Depths0 );
    float fEdge = Attenuation0.x + Attenuation0.y + Attenuation0.z + Attenuation0.w;

    if( fEdge == 4.0f )
        fEdge = 1.0f;
    else if( fEdge > 0.0f )
        fEdge = 0.5f;
    return fEdge;
}

Listing 2. Example (unoptimised) pixel shader code which is used to determine whether a pixel is on a shadow edge.

sampler shadowMaskTexture : register(s0);

struct PSInput
{
    float2 uvScreen;
};

float4 main( PSInput In ) : COLOR
{
    float2 uv = In.uvScreen;
    float fEdge = 0.0f;
    fEdge += tex2D( shadowMaskTexture, uv + SampleOffsets[0].xy ).r;
    fEdge += tex2D( shadowMaskTexture, uv + SampleOffsets[0].zw ).r;
    fEdge += tex2D( shadowMaskTexture, uv + SampleOffsets[1].xy ).r;
    fEdge += tex2D( shadowMaskTexture, uv + SampleOffsets[1].zw ).r;

    if( fEdge == 4.0f )
        fEdge = 1.0f;
    else if( fEdge > 0.0f )
        fEdge = 0.5f;
    return float4( fEdge, 1.0f, 1.0f, 1.0f );
}

Listing 3. Example (unoptimized) pixel shader code which expands the shadow edge mask for a single color channel.
Figure 6. The initial quarter sized shadow mask (top) and the edge expanded shadow mask (below).

3.1.2. Shadow Mask Pass

The final shadow mask pass is rendered using the downsized screen space shadow edge mask as input. The pixels marked as fully shadowed, and fully non-shadowed are rendered without the need for further shadow map lookups. The pixels marked as shadow edges are rendered as normal using percentage closer filtering.
The resulting final screen space shadow mask is shown in *Figure 7*.

![Figure 7](image)

*Figure 7. The final deferred shadow buffer. Yellow wireframe has been added to aid visualisation.*

### 3.1.3. Depth Bound Test

The depth bound test is a rendering optimization supported by some, but not all, modern consumer graphics hardware. For our target platforms the feature is supported by the Playstation® 3 but is not supported by the XBox® 360. The test compares the depth value at the screen space coordinates of the incoming fragment and discards fragments that are not within a user-defined range.

When rendering any form of cascaded shadow maps a common approach is to use the depth of the current fragment to determine which shadow cascade to sample. This is done either by using dynamic branching in the shader or by gathering all of the shadow cascades in a single texture and calculating texture sampling coordinates based on the cascade we are interested in.

The depth bound test suggests an alternative multi-pass approach. When creating our deferred shadow map we can use the depth bound test to carry out one pass for each shadow cascade. This early rejects all screen pixels which are not in the depth range matching the current cascade. The depth bounds test happens early in the graphics pipeline, which ensures there are no side effects from writes to the stencil or depth buffers and makes the operation very efficient.
Using this approach we avoid all shader predication based on depth when rendering our shadow masks. An added advantage is that we can easily turn off (or reduce the kernel size for) percentage closer filtering on all pixels belonging to the furthest cascade, and not pay any dynamic branching cost in the shader for doing so.

### 3.1.4. Performance

Sample performance timings for the Xbox® 360 are given in Table 4.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Without edge mask</th>
<th>With edge mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculate edge mask</td>
<td>-</td>
<td>0.6ms</td>
</tr>
<tr>
<td>Project shadows with PCF onto fragment 0 using edge mask</td>
<td>3.2ms</td>
<td>1.2ms</td>
</tr>
<tr>
<td>Project shadows with PCF onto fragment 1 using edge mask</td>
<td>3.2ms</td>
<td>0.5ms</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>6.4ms</strong></td>
<td><strong>2.3ms</strong></td>
</tr>
</tbody>
</table>

*Table 4. Performance timings for the deferred shadow pass*

### 4. Irradiance Volumes in Split/Second

For *Split/Second* we want the ability to render global illumination effects in a way that integrates well with our deferred shading pipeline. The environment in *Split/Second* can dramatically change and animate so we need a form of global illumination that can respond to those changes. For *Split/Second* irradiance volumes provide an acceptable approximation to global illumination within satisfactory performance parameters.

#### 4.1. Background

##### 4.1.1. Irradiance Environment Map

Irradiance is a measure of light flowing into a surface. It is quantified by measuring light energy crossing a unit area plane in one second.

We can derive irradiance for a given point from the radiance distribution function. In real time rendering we often store a discrete approximation of radiance in an environment cube map. To calculate irradiance for a point on a diffuse surface we simply perform a convolution of this radiance distribution function with a cosine kernel. The results can be stored in another cube map called the irradiance environment map.
4.1.2. Irradiance Volume

An irradiance volume is a 3D map of diffuse lighting samples. It can be created by storing irradiance environment maps for a grid of points in a static scene.

Irradiance volumes provide a way to approximate the irradiance distribution function in 3D space. This is a 5D function with 3 spatial dimensions and 2 directional dimensions. We can sample irradiance volumes to calculate global illumination for points in our scene. By evaluating the irradiance distribution function in the direction of a surface normal we get the irradiance at that surface location.

The irradiance distribution function at a sample point in the irradiance volume can be stored as a cube map, but other representations are also possible. Since the function tends to be low frequency, other more compact representations can be used without sacrificing accuracy.

4.2. Previous Work

The concept of an irradiance volume was pioneered by Greger et al. in [GSHG98]. Oat [OAT07] outlines how irradiance volumes can be integrated into real-time games, with comprehensive details on accelerating pre-calculation and minimizing storage overhead.

Ramamoorthi et al. [RAMAMOORTHIHANRAHAN01] showed that irradiance environment maps could be compactly represented in terms of Spherical Harmonics. Mitchell et al. [MMG06] define an alternative representation called the ambient cube.

4.3. Irradiance Volumes and Deferred Shading

When using irradiance volumes with a forward renderer, a common approach is to upload one or more sets of spherical harmonic coefficients describing the local indirect lighting to the shader constant registers before submitting a render batch. The indirect light contribution is then calculated in either the vertex or pixel shader. Shader code for this operation is provided in [SLOAN04].

When using deferred shading we have the option of calculating some part of the indirect lighting contribution and storing it in the G-Buffer. This can have the disadvantage that it complicates the shader code used in our geometry pass and uses limited G-Buffer space.

On Split/Second we apply the irradiance contribution in the deferred lighting pass. This has the advantage typical of deferred shading systems that we only pay the calculation
cost once per screen pixel. To do this we need to be able to determine the correct irradiance sample for each pixel in the G-Buffer.

Our solution is to store the irradiance volume information in a set of volume textures. This has the advantages that it maps naturally to GPU usage and the GPU carries out tri-linear interpolation of the coefficients to give us a smooth representation of indirect lighting across 3D space.

4.4. Irradiance Volumes in Split/Second

4.4.1. Irradiance Volume Creation

Irradiance volume data is pre-calculated by rendering cubic environment maps at our scene sample points. We then project this into our chose representation of irradiance.

We use an octree structure to adaptively subdivide the world. The maximum cell size for our octree data matches the size of our world streaming sectors. We adaptively subdivide cells further by testing if a cell contains scene geometry and subdividing if it does. For this process we use the depth map from our generated cube maps as described in [OAT07] to determine if a cell contains any scene geometry.

4.4.2. Rendering the Irradiance Volume

In Split/Second we divide the game world into sectors for which we stream geometry and texture data into memory as the camera moves around the world. It is straightforward to extend this system to stream our irradiance volume data in as required.

We chose a 2 meter world resolution for the volume texture texels and use volume textures of dimension 64x64x16. The volume textures are world-space aligned and snapped to a world-space grid to prevent shimmering artifacts when moving the camera.
**Figure 8.** Scene rendered with different colors denoting the area of influence of each irradiance volume.

**Figure 9.** Example scene rendered with a constant ambient color.
Figure 10. Example scene rendered with only the lighting from the irradiance volumes

Figure 11. Example scene rendered with only the direct lighting
The number of volume textures required for rendering depends on our choice of irradiance representation. The options are shown in Table 5. There is a significant benefit in choosing the more compact representations both in terms of GPU memory bandwidth when reading the textures, and CPU memory bandwidth when filling them. The ambient cube representation gives a good trade-off between size and quality for current hardware performance.

<table>
<thead>
<tr>
<th>Representation</th>
<th>Coefficients Per Color Channel</th>
<th>Volume Textures Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd order spherical harmonics</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>1st order spherical harmonics</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Ambient cube</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

*Table 5. Number of volume textures required for our possible irradiance representations*

We fill the volume textures once per frame. To minimize the cost of octree traversal we maintain a cache of fully calculated blocks that have been filled from our octree structure. We fill the volume texture memory on a background thread/processor using asynchronous resource locking techniques available in current generation consoles. Because the volume textures are statically positioned, one optimization is to only fill one volume texture per frame and use fixed offsets in the volume texture lookups to account for the case when volume textures span slightly offset regions.

The cost of rendering the irradiance volume is dominated by the texture lookups. *Table 6* shows the cost of rendering the irradiance volumes on a typical console GPU. At full
resolution, even the 1\textsuperscript{st} order spherical harmonics would usually be prohibitively expensive for a real time application. One typical optimization is to render the volume texture quarter sized making a trade-off between quality and speed of render.

<table>
<thead>
<tr>
<th>Representation</th>
<th>Full Size</th>
<th>Quarter Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>2\textsuperscript{nd} order</td>
<td>8.8ms</td>
<td>2.2ms</td>
</tr>
<tr>
<td>spherical harmonics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1\textsuperscript{st} order</td>
<td>4.6ms</td>
<td>1.2ms</td>
</tr>
<tr>
<td>spherical harmonics</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6. Performance timings for rendering with irradiance volumes

4.4.3. Volume Texture Draw Distance

The volume texture dimensions give us a maximum draw distance of 128 meters for the irradiance volume lighting contribution. This is close enough for the player to be able to detect a popping in of the effect when moving around the game world. We considered a number of possible solutions to this issue.

The simplest solution is to increase the size of the volume textures. This has a negative impact on memory usage and CPU cost when filling the volume textures. Another solution is to decrease the resolution of the volume textures. This has a negative impact on lighting quality. We found a resolution of 2 meters to be the sweet spot for the Split Second game world where performance and visual fidelity were balanced.

We also considered using a non-uniform volume texture with resolution decreasing as distance from the camera increases. This solution can give good visual results, but isn’t directly compatible with the use of an optimal cache to fill the volume texture.

The chosen solution was to interpolate the irradiance representation towards a fixed value when above a set draw distance. This is straight forward to do in the pixel shader and gives acceptable results. It also allows us to use a depth bound test on platforms that support it (as in section 3.1.3) which allows us to pay the cost of calculating indirect lighting only on those pixels within a set draw distance.

4.4.4. Dynamic Lighting Effects

Our initial hope when adding irradiance volumes to the \textit{Split/Second} graphics engine was that we would be able to update the irradiance volume samples in real-time as the world’s lighting and geometry were updated.

We considered a number of approaches to limit the storage and processing cost involved in doing this. Among them were:

- Only calculating and storing the irradiance samples in a fixed region around the camera.
• Only updating a fixed number of sampling points per frame, using heuristics to determine which are the most important.
• Using a simplified representation of the scene to generate the environment maps at the sampling points.
• Rendering the environment maps at the sampling points using deferred shading and caching the G-Buffers, so that changes in the lighting are relatively quick to recalculate.

After trying these approaches we were still unable to achieve the necessary level of performance and settled for an irradiance volume system based on pre-calculated irradiance samples. Even with this system there are a number of approaches which allow for some dynamic lighting.

Since irradiance is additive, lighting effects can simply be added into our irradiance representation at the point where we fill the volume texture. This provides a cheap way to add dynamic localized lighting.

During set-piece destruction of large scale geometry in a constrained region of the world it is possible to switch between two sets of irradiance volumes that describe the irradiance environments before and after the set-piece event. If the destruction is accompanied by strong direct lighting from an explosion it is generally possible to conceal the switch from the player.

5. Conclusion

We have shown a number of the techniques used in the rendering of Split/Second. With the combination of these and other techniques we expect to achieve high performance and visual quality in the final released game.

More generally as the quality and scope of real-time graphics continues to improve we hope to see increasing use of cinematography, lighting and color to create fantastic, visceral game experiences.

6. Acknowledgments

We would like to thank all of those who are contributing to the making of Split/Second. Specifically we would like to acknowledge the graphics programmers Matt Ritchie, Neil Hutchinson and Balor Knight for their contributions to the techniques described here.
7. References


Chapter N: Rendering Techniques in Split/Second