# 4

# Screen-Space Classification for Efficient Deferred Shading

*Balor Knight*
*Matthew Ritchie*
*George Parrish*
*Black Rock Studio*

## 4.1 Introduction

Deferred shading is an increasingly popular technique used in video game rendering. Geometry components such as depth, normal, material color, etc., are rendered into a geometry buffer (commonly referred to as a G-buffer), and then deferred passes are applied in screen space using the G-buffer components as inputs.

A particularly common and beneficial use of deferred shading is for faster lighting. By detaching lighting from scene rendering, lights no longer affect scene complexity, shader complexity, batch count, etc. Another significant benefit of deferred lighting is that only relevant and visible pixels are lit by each light, leading to less pixel overdraw and better performance.

The traditional deferred lighting model usually includes a fullscreen lighting pass where global light properties, such as sun light and sun shadows, are applied. However, this lighting pass can be very expensive due to the number of onscreen pixels and the complexity of the lighting shader required.

A more efficient approach would be to take different shader paths for different parts of the scene according to which lighting calculations are actually required. A good example is the expensive filtering techniques needed for soft shadow edges. It would improve performance

significantly if we only performed this filter on the areas of the screen that we know are at the edges of shadows. This can be done using dynamic shader branches, but that can lead to poor performance on current game console hardware.

Swoboda [2009] describes a technique that uses the PlayStation 3 SPUs to analyze the depth buffer and classify screen areas for improved performance in post-processing effects, such as depth of field. Moore and Jefferies [2009] describe a technique that uses low-resolution screen-space shadow masks to classify screen areas as in shadow, not in shadow, or on the shadow edge for improved soft shadow rendering performance. They also describe a fast multisample antialiasing (MSAA) edge detection technique that improves deferred lighting performance.

These works provided the background and inspiration for this chapter, which extends things further by classifying screen areas according to the global light properties they require, thus minimizing shader complexity for each area. This work has been successfully implemented with good results in Split/Second, a racing game developed by Disney's Black Rock Studio. It is this implementation that we cover in this chapter because it gives a practical real-world example of how this technique can be applied.

## 4.2 Overview of Method

The screen is divided into $4 \times 4$ pixel tiles. For every frame, each tile is classified according to the minimum global light properties it requires. The seven global light properties used on Split/Second are the following:

1. *Sky*. These are the fastest pixels because they don't require any lighting calculations at all. The sky color is simply copied directly from the G-buffer.

4. *Sun light*. Pixels facing the sun require sun and specular lighting calculations (unless they're fully in shadow).

5. *Solid shadow*. Pixels fully in shadow don't require any shadow or sun light calculations.

6. *Soft shadow*. Pixels at the edge of shadows require expensive eight-tap percentage closer filtering (PCF) unless they face away from the sun.

7. *Shadow fade*. Pixels near the end of the dynamic shadow draw distance fade from full shadow to no shadow to avoid pops as geometry moves out of the shadow range.

8. *Light scattering*. All but the nearest pixels have a light scattering calculation applied.

9. *Antialiasing*. Pixels at the edges of polygons require lighting calculations for both 2X MSAA fragments.

We calculate which light properties are required for each 4 × 4 pixel tile and store the result in a 7-bit classification ID. Some of these properties are mutually exclusive for a single pixel, such as sky and sunlight, but they can exist together when properties are combined into 4 × 4 pixel tiles.

Once we've generated a classification ID for every tile, we then create an index buffer for each ID that points to the tiles with that ID and render it using a shader with the minimum lighting code required for those light properties.

We found that a 4 × 4 tile size gave the best balance between classification computation time and shader complexity, leading to best overall performance. Smaller tiles meant spending too much time classifying the tiles, and larger tiles meant more lighting properties affecting each tile, leading to more complex shaders. A size of 4 × 4 pixels also conveniently matches the resolution of our existing screen-space shadow mask [Moore and Jefferies 2009], which simplifies the classification code, as explained later. For *Split/Second*, the use of 4 × 4 tiles adds up to 57,600 tiles at a resolution of 1280 × 720. Figures 4.1 and 4.2 show screenshots from the *Split/Second* tutorial mode with different global light properties highlighted.


Figure 4.1. A screenshot from *Split/Second* with soft shadow edge pixels highlighted in green.

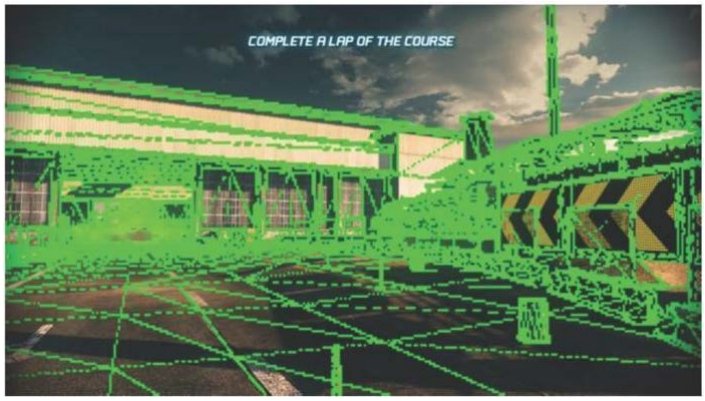


Figure 4.2. A screenshot from *Split/Second* with MSAA edge pixels highlighted in green.

## 4.3 Depth-Related Classification

Tile classification in *Split/Second* is broken into two parts. We classify four of the seven light properties during our screen-space shadow mask generation, and we classify the other three in a per-pixel pass. The reason for this is that the screen-space shadow code is already generating a one-quarter resolution (320 × 180) texture, which perfectly matches our tile resolution of pixels, and it is also reading depths, meaning that we can minimize texture reads and shader complexity in the per-pixel pass by extending the screen-space shadow mask code to perform all depth-related classification.

Moore and Jefferies [2009] explain how we generate a one-quarter resolution screen-space shadow mask texture that contains three shadow types per pixel: pixels in shadow, pixels not in shadow, and pixels near the shadow edge. This work results in a texture containing zeros for pixels in shadow, ones for pixels not in shadow, and all other values for pixels near a shadow edge. By looking at this texture for each screen-space position, we can avoid expensive PCF for all areas except those near the edges of shadows that we want to be soft.

For tile classification, we extend this code to also classify light scattering and shadow fade since they're both calculated from depth alone, and we're already reading depth in these shaders to reconstruct world position for the shadow projections.

Listing 4.1. Classifying light scattering and shadow fade in the first-pass shadow mask shader.

```
float shadowTpe = CalcShadowType(worldPos, depth);
float lightScattering = (depth > scatteringStartDist) ? 1.0 : 0.0;
float shadowFade = (depth > shadowFadeStartDist) ? 1.0 : 0.0;
output.color = float4(shadowTpe, lightScattering, shadowFade, 0.0);
```

Recall that the shadow mask is generated in two passes. The first pass calculates the shadow type per pixel at one-half resolution (640 × 360) and the second pass conservatively expands the pixels marked as near shadow edge by downsampling to one-quarter resolution. Listing 4.1 shows how we add a simple light scattering and shadow fade classification test to the first-pass shader.

Listing 4.2 shows how we extend the second expand pass to pack the classification results together into four bits so they can easily be combined with the per-pixel classification results later on.

Listing 4.2. Packing classification results together in the second-pass shadow mask shader. Note that this code could be simplified under shader model 4 or higher because they natively support integers and bitwise operators.

```
// Read 4 texels from 1st pass with sample offsets of 1 texel.
#define  OFFSET_X(1.0 / 640.0)
#define  OFFSET_Y(1.0 / 360.0)

float3 rgb = tex2D(tex, uv + float2(-OFFSET_X, -OFFSET_Y)).rgb;
rgb += tex2D(tex, uv + float2(OFFSET_X, -OFFSET_Y)).rgb;
rgb += tex2D(tex, uv + float2(-OFFSET_X, OFFSET_Y)).rgb;
rgb += tex2D(tex, uv + float2(OFFSET_X, OFFSET_Y)).rgb;

// Pack classification bits together.
#define  RAW_SHADOW_SOLID          (1 << 0)
#define  RAW_SHADOW_SOFT           (1 << 1)
#define  RAW_SHADOW_FADE           (1 << 2)
#define  RAW_LIGHT_SCATTERING       (1 << 3)

float bits = 0.0;

if (rgb.r ==0.0)
  bits += RAW_SHADOW_SOLID / 255.0;
else if (rgb.r < 4.0)
  bits += RAW_SHADOW_SOFT / 255.0;
```

```
if (rgb.b ! = 0.0)
   bits += RAW_SHADOW_FADE / 255.0;

if (rgb.g ! = 0.0)
  bits += RAW_LIGHT_SCATTERING / 255.0;

// Write results to red channel.
output.color = float4(bits, 0.0, 0.0 ,0.0);
```

## 4.4 Pixel Classification

It helps to explain how this pass works by describing the *Split/Second* G-buffer format (see Table 4.1). Moore and Jefferies [2009] explain how we calculate a per-pixel MSAA edge bit by comparing the results of centroid sampling against linear sampling. We pack this into the high bit of our motion ID byte in the G-buffer. For classifying MSAA edges, we extract this MSAA edge bit from both MSAA fragments and also compare the normals of each of the fragments to catch situations in which there are no polygon edges (e.g., polygon intersections).

Table 4.1. The *Split/Second* G-buffer format. Note that each component has an entry for both 2X MSAA fragments.

| Buffer | Red | Green | Blue | Alpha |
|---|---|---|---|---|
| Buffer 0 | Albedo red | Albedo green | Albedo blue | Specular amount |
| Buffer 1 | Normal *x* | Normal *y* | Normal *z* | Motion ID + MSAA edge |
| Buffer 2 | Prelit red | Prelit green | Prelit blue | Specluar power |

The motion ID is used for per-pixel motion blur in a later pass, and each object type has its own ID. For the sky, this ID is always zero, and we use this value to classify sky pixels.

For sun light classification, we test normals against the sun direction (unless it's a sky pixel). Listing 4.3 shows how we classify MSAA edge, sky, and sunlight from both G-buffer 1 fragments.

Listing 4.3. Classifying MSAA edge, sky, and sun light. This code could also be simplified in shader model 4 or higher.

```
// Separate motion IDs and MSAA edge fragments from normals.
float2 edgeAndID_frags = float2(gbuffer1_frag0.w, gbuffer1_frag1.w);

// Classify MSAA edge (marked in high bit).
float2 msaaEdge_frags = (edgeAndID_frags > (128.0 / 255.0));
float mssaEdge = any(msaaEdge_frags);

float3 normalDiff = gbuffer1_frag0.xyz -gbuffer1_frag1.xyz;
mssaEdge += any(normalDiff);

// Classify sky (marked with motion ID of 0 - MSAA edge bit
// will also be 0).
float2 sky_frags = (edgeAndID_frags == 0.0);
float sky = any(sky_frags);

// Classify sunlight (except in sky).
float2 sunlight_frags;
sunlight_frags.x = sky_frags.x ? 0.0 : -dot(normal_frag0, sunDir);
sunlight_frags.y = sky_frags.y ? 0.0 : -dot(normal_frag1, sunDir);
float sunlight = any(sunlight_frags);

// Pack classification bits together.
#define  RAW_MSAA_EDGE       (1 << 4)
#define  RAW_SKY            (1 << 5)
#define  RAW_SUN_LIGHT       (1 << 6)

float bits = msaaEdge ? (RAW_MSAA_EDGE / 255/0) : 0.0;
bits += sky ? (RAW_SKY / 255/0) : 0.0;
bits += sunlight ? (RAW_SUN_LIGHT / 255/0) : 0.0;
```

## 4.5 Combining Classification Results

We now have per-pixel classification results for MSAA edge, sky, and sunlight, but we need to downsample each $4 \times 4$ pixel area to get a per-tile classification ID. This is as simple as ORing each $4 \times 4$ pixel area of the pixel classification results together. We also need to combine these results with the depth-related classification results to get a final classification ID per tile. Both these jobs are done in a very different way on each platform in order to make the most of their particular strengths and weaknesses, as explained in Section 4.9.

## 4.6 Index Buffer Generation

Once both sets of classification results are ready, a GPU callback triggers index buffer generation for each classification ID. There is one preallocated index buffer containing exactly enough indices for all tiles. On the Xbox 360, we use the `RECT` primitive type, which requires three indices per tile, and on the PlayStation 3, we use the the `QUAD` primitive type, which requires four indices per tile. The index buffer references a prebuilt vertex buffer containing a vertex for each tile corner. At a tile resolution of $4 \times 4$ pixels, this equates to $321 \times 181$ vertices at a screen resolution of $1280 \times 720$.

Index buffer generation is performed in three passes. The first pass iterates over every tile and builds a table containing the number of tiles using each classification ID, as shown in Listing 4.4. The second pass iterates over this table and builds a table of offsets into the index buffer for each classification ID, as shown in Listing 4.5. The third pass fills in the index buffer by iterating over every tile, getting the current index buffer offset for the tile's classification ID, writing new indices for that tile to the index buffer, and advancing the index buffer pointer. An example using the `QUAD` primitive is shown in Listing 4.6. We now have a final index buffer containing indices for all tiles and a table of starting indices for each classification ID. We're ready to render!

Listing 4.4. This code counts the number of tiles using each classification ID.

```
#define  SHADER_COUNT      128
#define  TILE_COUNT        (320 * 180)

unsigned int shaderTileCounts[SHADER_COUNT];
for (int shader = 0; shader < SHADER_COUNT; shader++)
{
  shaderTileCounts[shader] = 0;
}

for (int tile = 0; tile < TILE_COUNT; tile++)
{
  unsigned char id = classificationData[tile];
  shaderTileCounts[id] ++;
}
```

Listing 4.5. This code builds the index buffer offsets. We store a pointer per shader for index buffer generation and an index per shader for tile rendering.

```
unsigned int  *indexBufferPtrs[SHADER_COUNT];
int           indexBufferOffsets{SHADER_COUNT};
int           currentIndexBufferOffset = 0;

for (int shader = 0; shader < SHADER_COUNT; shader++)
{
  // Store shader index buffer ptr.
  indexBufferPtrs[shader] = indexBufferStart + currentIndexBufferOffset;

  // Store shader index buffer offset.
  indexBufferOffsets[shader] = currentIndexBufferOffset;

  // Update current offset.
  currentIndexBufferOffset += shaderTileCounts[shader] * INDICES_PER_PRIM;
}
```

Listing 4.6. This code builds the index buffer using the QUAD primitive.

```
#define  TILE_WIDTH   320
#define  TILE_HEIGHT  180

for (int  y = 0;  y < TILE_HEIGHT;  y++)
{
  for (int x = 0; x < TILE_WIDTH; x++)
  {
    int tileIndex = y * TILE_WIDTH + x;
    unsigned char id = classificationData[tileIndex];
    unsigned int index0 = y * (TILE_WIDTH + 1) + x;

    *indexBufferPtrs[id]++ = index0;
    *indexBufferPtrs[id]++ = index0 + 1;
    *indexBufferPtrs[id]++ = index0 + TILE_WIDTH + 2;
    *indexBufferPtrs[id]++ = index0 + TILE_WIDTH + 1;
  }
}
```

## 4.7 Tile Rendering

To render the tiles, we'd like to simply loop over each classification ID, activate the shaders,

then issue a draw call to render the part of the index buffer we calculated earlier. However, it's not that simple because we want to submit the index buffer draw calls before we've received the classification results and built the index buffers. This is because draw calls are submitted on the CPU during the render submit phase, but the classification is done later on the GPU. We solve this by submitting each shader activate, then inserting the draw calls between each shader activate later on when we've built the index buffers and know their starting indices and counts. This is done in a very platform-specific way and is explained in Section 4.9.

## 4.8 Shader Management

Rather than trying to manage 128 separate shaders, we opted for a single ubershader with all lighting properties included, and we used conditional compilation to remove the code we didn't need in each case. This is achieved by prefixing the uber-shader with a fragment defining just the properties needed for each shader. Listing 4.7 shows an example for a shader only requiring sunlight and soft shadow. The code itself is not important and just illustrates how we conditionally compile out the code we don't need.

Listing 4.7. This example shader code illustrates how we generate a shader for sunlight and soft shadow only.

```
// Fragment definging light properties.
#define  SUN_LIGHT
#define  SOFT_SHADOW

//Uber-shader starts here.
//...

// Output color stats wiht prelit.
float3 oColor = preLit;

// Get sun shadow contribution.
#if defined(SOFT_SHADOW) && defined(SUN_LIGHT)

float sunShadow = CascadeShadowMap_8Taps(worldPos, depth);

#elif defined(SUN_LIGHT) && !defined(SOFT_SHADOW)

float sunShadow = 1.0;
```

```
#else

float sunShadow = 0.0;

#endif

// Fade sun shadow.
#if (defined(SOFT_SHADOW) || defined(SOFT_SHADOW)) && \
  defined(SHADOW_FADE) && defined(SUN_LIGHT)

sunShadow = lerp(sunShadow, 1.0, satureate(depth * shadowFadeScale +
shadowFadeOffset));

#endif

// Apply sunlight.
#if defined(SUN_LIGHT) && !defined(SOLID_SHADOW)

float3 sunDiff, sunSpec;

Global_CalcDirectLighting(normal, view, sunShadow, specIntAndPow,
     sunDiff, sunSpec);

oColor += (albedo * sunDiff) + sunSpec;

#endif

// Apply ligth scattering.
#ifdef LIGHT_SCATTERING

float3 colExtinction, colInscattering;
lightScattering(view, depth, lightDir, colExtinction, colInscattering);
oColor = oColor * colExtinction + colInscattering;

#endif
```

## 4.9 Platform Specifics

### Xbox 360

On the Xbox 360, downsampling pixel classification results and combining with the depth-

related classification results is performed inside the pixel classification shader, and the final 7-bit classification IDs are written to a one-quarter resolution buffer in main memory using the `memexport` API. We use `memexport` rather than rendering to texture so we can output the IDs as nonlinear blocks, as shown in Figure 4.3. This block layout allows us to speed up index buffer generation by coalescing neighboring tiles with the same ID, as explained in Section 4.10. Another benefit of using `memexport` is that it avoids a resolve to texture. Once we've written out all final IDs to CPU memory, a GPU callback wakes up a CPU thread to perform the index buffer generation.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 16 | 17 | 18 | 19 |
| 4 | 5 | 6 | 7 | 20 | 21 | 22 | 23 |
| 8 | 9 | 10 | 11 | 24 | 25 | 26 | 27 |
| 12 | 13 | 14 | 15 | 28 | 29 | 30 | 31 |

Figure 4.3. Xbox 360 tile classification IDs are arranged in blocks of 4 × 4 tiles, giving us 80 × 45 blocks in total. The numbers show the memory offsets, not the classification IDs.

Before we can allow tile rendering to begin, we must make sure that the CPU index buffer generation has finished. This is done by inserting a GPU block that waits for a signal from the CPU thread (using asynchronous resource locks). We insert other GPU jobs before the block to avoid any stalls.

We use Xbox Procedural Synthesis (XPS) callbacks for tile rendering as they allow us to dynamically generate draw calls inside the callback. We insert an XPS callback after each shader activate during the CPU render submit, then submit each draw call in the XPS callback using the index buffer offsets and counts we calculated during index buffer generation.

Figure 4.4 shows how it all fits together, particularly the classification flow between GPU and CPU. The dotted arrow represents other work that we do to keep the GPU busy while the CPU generates the index buffer.

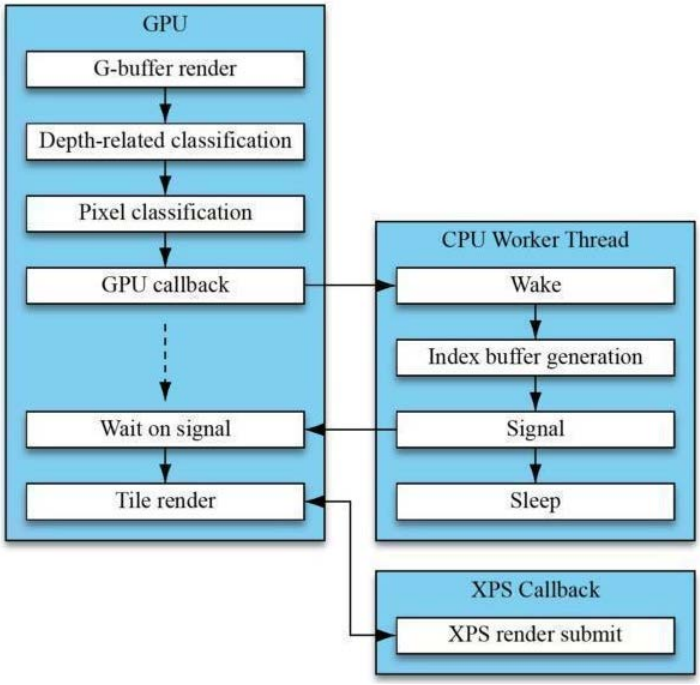


Figure 4.4. Xbox 360 classification flow.

### PlayStation 3

On the PlayStation 3, the pixel classification pass is piggybacked on top of an existing depth and normal restore pass as an optimization to avoid needing a specific pass. This pass creates non-antialiased, full-width depth and normal buffers for later non-antialiased passes, such as local lights, particles, post-processing, etc., and we write the classification results to the unused *w* component of the normal buffer.

Once we've rendered the normals and pixel classification to a full-resolution texture, we then trigger a series of SPU downsample jobs to convert this texture into a one-quarter resolution buffer containing only the pixel classification results. Combination with the depth-related classification results is performed later on during the index buffer generation because those results aren't ready yet. This is due to the fact that we start the depth-related classification work on the GPU at the same time as these SPU downsample jobs to maximize parallelization between the two.

We spread the work across four SPUs. Each SPU job takes 64 × 64 pixels of classification

data (one main memory frame buffer tile), ORs each 4 × 4 pixel area together to create a block of classification IDs, and streams them back to main memory. Figure 4.5 shows how output IDs are arranged in main memory. We take advantage of this block layout to speed up index buffer generation by coalescing neighboring tiles with the same ID, as explained in Section 4.10. Using 16 × 16 tile blocks also allows us to send the results back to main memory in a single DMA call. Once this SPU work and the depth related classification work have both finished, a GPU callback triggers SPU jobs to combine both sets of classification results together and perform the index buffer generation and draw call patching.

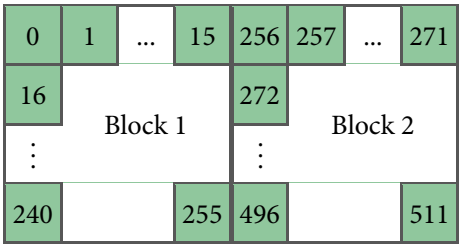


Figure 4.5. PlayStation 3 tile classification IDs are arranged in blocks of 16 × 16 tiles, giving us 20 × 12 blocks in total. The numbers show the memory offsets, not the classification IDs.

The first part of tile rendering is to fill the command buffer with a series of shader activates interleaved with enough padding for the draw calls to be inserted later on, once we know their starting indices and counts. This is done on the CPU during the render submit phase.

Index buffer generation and tile rendering is spread across four SPUs, where each SPU runs a single job on a quarter of the screen. The first thing we do is combine the depth-related classification with the pixel classification. Remember that we couldn't do it earlier because the depth-related classification is rendered on the GPU at the same time as the pixel classification downsample jobs are running on the SPUs. Once we have final 7-bit IDs, we can create the final draw calls. Listings 4.5 and 4.6 show how we calculate starting indices and counts for each shader, and we use these results to patch the command buffer with each draw call.

Figure 4.6 shows how it all fits together, particularly the classification flow between GPU and SPU jobs. The dotted arrow represents other work that we do to keep the GPU busy while the SPU generates the index buffer.

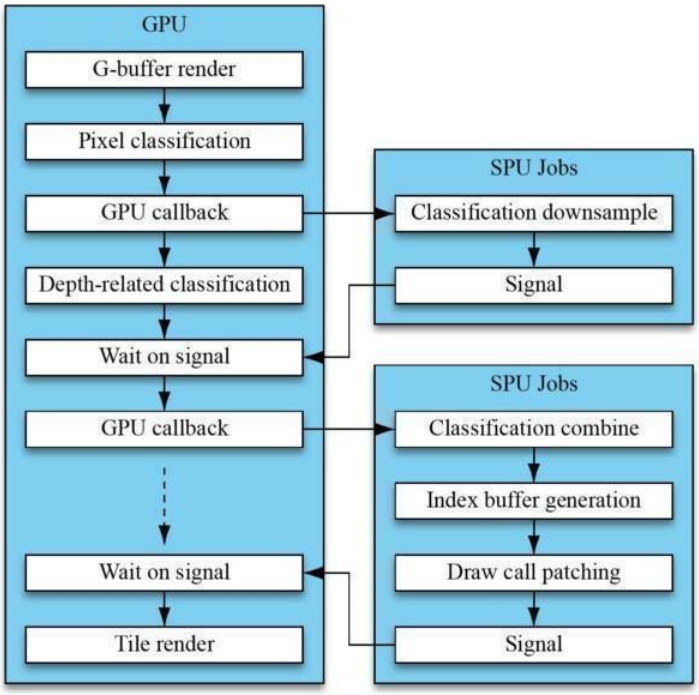


Figure 4.6. PlayStation 3 classification flow.

## 4.10 Optimizations

### Reducing Shader Count

We realized that some of the 7-bit classification combinations are impossible, such as sun light and solid shadow together, no sunlight and soft shadow together, etc., and we were able to optimize these seven bits down to five by collapsing the four sun and shadow bits into two. This reduced the number of shaders from 128 to 32 and turned out to be a very worthwhile optimization.

We do this by prebuilding a lookup table that converts a 7-bit raw classification ID into a 5-bit optimized classification ID, as shown in Listing 4.8. Each ID is passed through this lookup table before being used for index buffer generation.

These two bits give us four possible combined properties, which is enough to represent all possible combinations in the scene as follows:

- 00 = solid shadow.
- 01 = solid shadow + shadow fade + sunlight.
- 10 = sun light (with no shadows).
- 11 = soft shadow + shadow fade + sunlight.

The only caveat to collapsing these bits is that we're now always calculating shadow fade when soft shadows are enabled. However, this extra cost is negligible and is far outweighed by the benefits of reducing the shader count to 32.

Listing 4.8. This builds a lookup table to convert from raw to optimized material IDs.

```
#define  LIGHT_SCATTERING     (1 << 0)
#define  MSAA_EDGE            (1 << 1)
#define  SKY                  (1 << 2)
#define  SUN_0                (1 << 3)
#define  SUN_1                (1 << 4)

unsigned char output[32];

for (int iCombo = 0; iCombo < 128; iCombo++)
{
  //Clear output bits.
  unsigned char bits = 0;

  // Most combos are directly copied.
  if (iCombo & RAW_LIGHT_SCATTERING) bits |= LIGHT_SCATTERING;
  if (iCombo & RAW_MSAA_EDGE) bits |= MSAA_EDGE;
  if (iCombo & RAW_SKY) bits |= SKY;

  // If in solid shadow.
  if (iCombo & RAW_SHADOW_SOLID)
  {
    // Set bit 0 if fading to sun.
    if ((iCombo & RAW_SHADOW_FADE) &&
      (iCombo & RAW_SUN_LIGHT))
      bits |= SUN_0;
  }
  else if (iCombo & RAW_SUN_LIGHT)  //else if in sun
  {
    // Set bit 1.
    bits |= SUN_1;
```

```
    // Set bit 0 if in soft shadow.
    if (iCombo & RAW_SHADOW_SOFT)
      bits |= SUN_0;
  }

  // Write output.
  output[iCombo] = bits;
}
```

### Tile Coalescing

We mentioned earlier that we take advantage of the tile block layout to speed up index buffer generation by coalescing adjacent tiles. This is done by comparing each entry in a block row, which is very fast because they're contiguous in memory. If all entries in a row are the same, we join the tiles together to make a single quad. Figure 4.7 shows an example for the Xbox 360 platform. By coalescing a row of tiles, we only have to generate one primitive instead of four. On the PlayStation 3, the savings are even greater because a block size is 16 × 16 tiles. We extend this optimization even further on the Xbox 360 and coalesce an entire block into one primitive if all 16 IDs in the block are the same.

| | | | |
|---|---|---|---|
| 10 | 10 | 18 | 10 |
| 34 | 34 | 34 | 34 |
| 34 | 18 | 10 | 10 |
| 10 | 10 | 10 | 10 |

Figure 4.7. Coalescing two rows of tiles within a single block on the Xbox 360.

## 4.11 Performance Comparison

Using the scene shown in Figure 4.8, we compare performance between a naive implementation that calculates all light properties for all pixels and our tile classification method. The results are shown in Table 4.2. The pixel classification cost on the PlayStation 3 is very small because we're piggybacking onto an existing pass, and the extra shader cost is minimal.

Figure 4.8. The Split/Second scene used for our performance comparisons.

Table 4.2. Performance comparisons for a naive implementation versus our tile classification method. All numbers are times measured in milliseconds.

| Task | PlayStation 3 Naive | PlayStation 3 Classification | Xbox 360 Naive | Xbox 360 Classification |
|---|---|---|---|---|
| Depth-related classification | 0.00 | 1.29 | 0.00 | 0.70 |
| Pixel classification | 0.00 | ~ 0.00 | 0.00 | 0.69 |
| Global light pass | 13.93 | 4.82 | 8.30 | 2.69 |
| Total | 13.93 | 6.11 | 8.30 | 4.08 |

## References

[Swoboda 2009] Matt Swoboda. "Deferred Lighting and Post Processing on PlayStation 3." *Game Developers Conference*, 2009.

[Moore and Jefferies 2009] Jeremy Moore and David Jefferies. "Rendering Techniques in Split/Second." *Advanced Real-Time Rendering in 3D Graphics and Games*, ACM SIGGRAPH 2009 course notes.