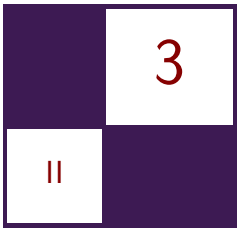


Deferred Attribute Interpolation Shading

Christoph Schied and Carsten Dachsbacher

3.1 Introduction

Deferred shading is a popular technique in real-time rendering. In contrast to a traditional rendering pipeline, deferred shading is split into two phases. First, the geometry is sampled and stored in the geometry buffer, which serves as input for the second phase where the shading is computed. Thereby, the complexity for shading is decoupled from the geometric complexity, and furthermore advanced geometry-aware screen-space techniques may be employed. However, deferred shading does not play well with multi-sample antialiasing. Multi-sample antialiasing samples the visibility of a primitive at several subpixel positions, however the shading is only evaluated once inside a pixel per primitive. Deferred shading samples the geometric attributes, and the shading is deferred into a second phase where the correspondence between primitives and visibility samples is lost, which makes it hard to avoid redundant shading. Furthermore, the geometry buffer can become prohibitively large in case of high screen resolutions and high visibility sampling because each sample needs to store all attributes.

In this chapter based on our publication [Schied and Dachsbacher 15], we present a technique to dramatically reduce the memory consumption of deferred shading in the aforementioned setting. Unlike deferred shading, our method samples solely visibility in the geometry phase and defers the attribute interpolation and material evaluation to the shading phase. This allows us to store the data needed for shading at per-triangle instead of per-sample frequency. Compared to a G-buffer sample, storing a triangle uses more memory, but since in practice most triangles will cover several pixels, the cost is amortized across several visibility samples, which leads to a significant reduction in the overall memory cost. Visible triangles are identified in the geometry phase and stored in the triangle buffer. The geometry buffer is replaced by a visibility buffer [Burns and Hunt 13],

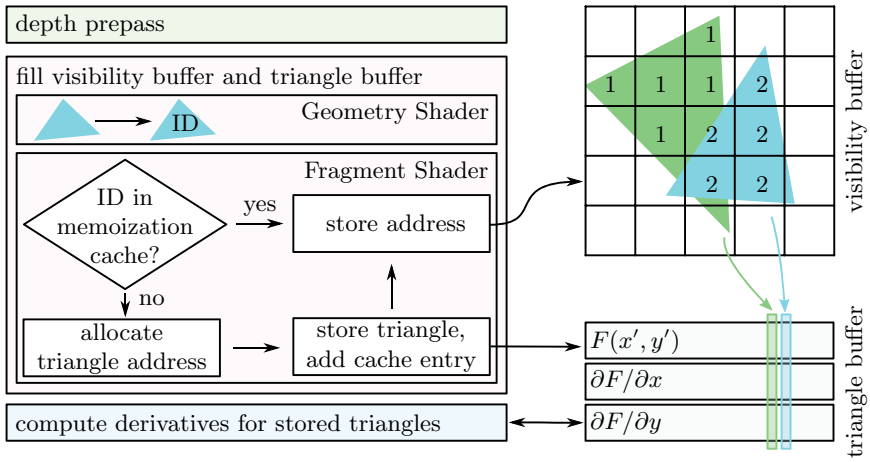


Figure 3.1. The first phase of our algorithm that samples the visibility of triangles. A depth prepass is performed to ensure that in the second geometry pass only visible triangles will generate fragment shader invocations. In the second pass, each triangle is first assigned an ID that is used in the fragment shader as a lookup into the memoization cache that stores mappings between triangle IDs and physical addresses. In case the cache does not contain the mapping yet, a new entry is allocated in the triangle buffer, the triangle is stored, and the new address is added to the cache. Finally, for each triangle the screen-space partial derivatives, needed for attribute interpolation, are computed in a separate pass.

which stores references in the triangle buffer. To enable efficient attribute interpolation during shading, triangles are represented using partial derivatives of the attributes.

3.2 Algorithm

Similar to deferred shading, the drawing of a frame is split into two phases. In the first phase all visible triangles are determined and stored in the triangle buffer. (See Figure 3.1.) Furthermore the visibility buffer is populated with references to these triangles. In the second phase the triangle attributes are interpolated and the shading is computed. Our method stores visible triangles after vertex transformations have been applied. Therefore, vertex transformations do not need to be carried out during the shading phase, and furthermore this makes our method compatible with the use of tessellation shaders. Compared to deferred shading, we introduce the cut in the pipeline at an earlier stage, i.e., before attribute interpolation and material evaluation. The following sections describe the attribute interpolation as well as the two rendering phases in more detail.

3.2.1 Attribute Interpolation Using Partial Derivatives

Interpolation of vertex attributes a_i with respect to a triangle is commonly done by barycentric weighting of all attributes. The barycentric coordinates λ_i of a point (x, y) with respect to a triangle with points $p_i = (u_i, v_i)$ can be computed as a ratio of areas by

$$\begin{aligned}\lambda_1(x,y) &= \frac{(v_2 - v_3)(x - u_3) + (u_3 - u_2)(y - v_3)}{D},\\ \lambda_2(x,y) &= \frac{(v_3 - v_1)(x - u_3) + (u_1 - u_3)(y - v_3)}{D},\\ \lambda_3(x,y) &= 1 - \lambda_1(x,y) - \lambda_2(x,y),\end{aligned} \tag{3.1}$$

where $D = \det(p_3 - p_2, p_1 - p_2)$. The interpolated attribute is then determined as

$$\tilde{a}(x,y) = \sum_{i=1}^{3} \lambda_i(x,y) \cdot a_i \ . \tag{3.2}$$

Because $\lambda_i(x, y)$ is linear in the x- and y-directions, the partial derivatives with respect to x, y are constant, and Equation (3.2) can be reformulated as

$$\begin{aligned}\tilde{a}(x,y) &= a_{x'y'} + (x - x') \sum_i \frac{\partial \lambda_i}{\partial x} \cdot a_i + (y - y') \sum_i \frac{\partial \lambda_i}{\partial y} \cdot a_i\\ &= a_{x'y'} + (x - x') \frac{\partial a}{\partial x} + (y - y') \frac{\partial a}{\partial y},\end{aligned} \tag{3.3}$$

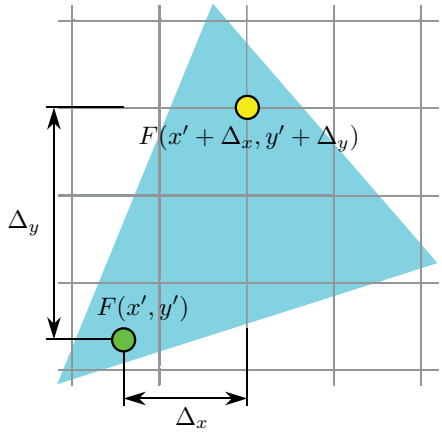
assuming that the attribute $a_{x'y'}$ is known for an arbitrary sample point (x', y'). (See Figure 3.2.)

For projected triangles defined in four-dimensional homogeneous coordinates with the vertices (x_i, y_i, z_i, w_i), a perspective correction needs to be applied when interpolating attributes. This correction is done by interpolating a_i/w_i and $1/w_i$ linearly in the screen space and dividing the interpolants afterward. This leads to an interpolation scheme defined as $a(x, y) = (\sum \lambda_i a_i/w_i)/(\sum \lambda_i/w_i)$. Reformulating this expression similar to Equation (3.3) leads to

$$a(x,y) = \frac{\frac{a_{x'y'}}{w_{x'y'}} + (x - x') \frac{\partial a/w}{\partial x} + (y - y') \frac{\partial a/w}{\partial y}}{\frac{1}{w_{x'y'}} + (x - x') \frac{\partial 1/w}{\partial x} + (y - y') \frac{\partial 1/w}{\partial y}}. \tag{3.4}$$

Assuming that the triangle has been clipped and projected to the screen, the partial derivatives of the attributes can be computed as

$$\frac{\partial a/w}{\partial x} = \sum_i \frac{\partial \lambda_i}{\partial x} \cdot \frac{a_i}{w_i}, \quad \frac{\partial a/w}{\partial y} = \sum_i \frac{\partial \lambda_i}{\partial y} \cdot \frac{a_i}{w_i}, \tag{3.5}$$



$$F(x' + \Delta_x, y' + \Delta_y) = F(x', y') + \Delta_x \tfrac{\partial F}{\partial x} + \Delta_y \tfrac{\partial F}{\partial y}$$

Figure 3.2. A sample of the attribute is stored at a sample point (green). The attribute can be interpolated at an arbitrary position (yellow) by weighting the partial derivatives in the x- and y-directions by their respective screen-space distances Δ_x and Δ_y.

whereas the partial derivatives of the barycentric coordinates are derived from Equation (3.1) as

$$\begin{aligned} \frac{\partial \lambda_1}{\partial x} &= \frac{y_2 - y_3}{D}, & \frac{\partial \lambda_2}{\partial x} &= \frac{y_3 - y_1}{D}, & \frac{\partial \lambda_3}{\partial x} &= \frac{y_1 - y_2}{D}, \\ \frac{\partial \lambda_1}{\partial y} &= \frac{x_3 - x_2}{D}, & \frac{\partial \lambda_2}{\partial y} &= \frac{x_1 - x_3}{D}, & \frac{\partial \lambda_3}{\partial y} &= \frac{x_2 - x_1}{D}. \end{aligned} \tag{3.6}$$

3.2.2 Visibility Sampling Phase

The first phase employs two geometry passes to identify and store visible triangles. A depth prepass is performed that constrains the fragment shader execution in the second pass to visible surfaces. Therefore, we can use the fragment shader in the second geometry pass to store visible triangles. Since it is our goal to share the triangle data across several visibility samples, we need to ensure that triangles are uniquely stored. Additionally, the address of the stored triangle needs to be communicated to all fragment shader invocations, which store the address in the visibility buffer. We assign each triangle a unique ID and use a memoization cache that stores mappings between triangle IDs and physical addresses. This allows each fragment shader invocation to query the cache if the triangle is already stored, and thereby get the physical address of the triangle. If a requested triangle ID is not found in the cache, one invocation is selected to allocate space in the triangle buffer and to store the ID-to-address mapping in

the cache. Finally, the triangle is stored by the same invocation in the triangle buffer. All invocations store the physical address in the visibility buffer.

3.2.3 Shading Phase

Because the geometry pass only samples visibility and does not capture the geometric attributes per visibility sample, these attributes need to be interpolated in the shading phase. A compute pass is used to determine the partial derivatives needed to interpolate attributes, as described in Section 3.2.1 for each visible triangle.

During shading the visible triangles can be determined per pixel using a lookup into the visibility buffer. The precomputed data needed for interpolation is loaded, the attributes are interpolated according to Equation (3.4), and finally the materials are evaluated.

3.2.4 Multi-rate Shading

Shading contributes a large part to the computational costs for rendering a frame and becomes increasingly expensive with growing screen resolutions. Since not all components of the shading signal are high frequency (such as, for example, indirect illumination, which is particularly costly to evaluate), such components can be sampled at reduced frequency. Our pipeline allows us to create shading samples that reference a primitive and store a screen-space position. These shading samples are referenced by an additional render target in the visibility buffer and are evaluated in a compute pass prior to shading. In the shading phase the results of the evaluated shading samples are combined with the full shading-rate signal. While it would be possible to achieve arbitrary shading-rates using a similar approach as proposed by Liktor et al. [Liktor and Dachsbacher 12], we use a simplified approach that relies on inter-thread communication inside of a shading quad.

3.3 Implementation

The following section explains our implementation that makes use of the OpenGL 4.5 API.

3.3.1 Visibility Sampling Phase

At first the depth buffer is populated with front-most surfaces by performing a depth prepass. Setting the depth test in the second geometry pass to `GL_EQUAL` allows us to perform alpha-clipping in the depth prepass and thus to ignore alpha in the second geometry pass. In the second pass the geometry shader is used to pass all vertices of the triangle through to the fragment shader. When using tessellation shaders, each triangle needs to be assigned a unique ID; otherwise, the

language built-in variable `gl_PrimitiveID` may be used. To assign a unique ID, an atomic counter is incremented and passed through to the fragment shader. We use frustum culling in the geometry shader, which can be implemented efficiently using bit operations to reduce the number of atomic counter operations.

Early depth testing has to be enabled in the fragment shader to ensure that the fragment shader is executed for visible fragments only:

```
layout(early_fragment_tests) in;
```

In the fragment shader a lookup into the memoization cache is performed to get the physical address of the stored triangle. The return value of the lookup function tells if the triangle needs to be stored by the current invocation. Our implementation of the memoization cache closely follows the implementation by Liktor et al. [Liktor and Dachsbacher 12]. It is explained in depth in Section 3.3.2.

The fragment shader stores all vertices of the triangle in the triangle buffer, whereas in a later pass the vertices are overwritten by their partial derivatives, since the original vertex data is not needed anymore during shading. To reduce storage costs when storing the vertices, normal vectors are encoded to 32 Bit using a octahedral encoding [Cigolle et al. 14]. In the beginning of the triangle struct we store a material ID what enables the use of multiple storage formats.

3.3.2 Memoization cache

Our implementation of the memoization cache (refer to Listing 3.1) closely follows the implementation by Liktor et al. [Liktor and Dachsbacher 12]. The image buffer `locks` stores a lock for each cache bucket, where each entry can be either `LOCKED` or `UNLOCKED`. Furthermore, the `cache` image buffer stores two cache entries, each represented by a triangle ID and the corresponding address. Invalid addresses are represented by negative values. When an ID is found in a cache bucket, the found address is stored in the address variable. The return value of the function denotes if a new slot was allocated and therefore the data has to be stored by the current invocation. In the case that the cache does not contain the desired entry, a `imageAtomicExchange` operation is issued to gain exclusive access to the cache bucket. When exclusive access is granted, a new address is allocated and stored alongside the ID in the cache bucket. Older entries are removed in a FIFO manner. This strategy is reasonable because it is to be expected that fragment shader invocations are scheduled according to the rasterization order. For the same reason, a simple modulus hash-function works well with monotonically increasing triangle IDs. When the fragment shader invocation fails to gain access to the cache bucket, it waits a limited amount of time for the bucket to be unlocked and reloads the entry.

Graphics cards execute several threads in lock-step whereby diverging branches are always taken by all threads and the results are masked out accordingly af-

```
1  layout(rgba32ui) coherent volatile restrict uimageBuffer cache;
2  layout(r32ui)    coherent volatile restrict uimageBuffer locks;
3
4  bool lookup_memoization_cache(
5    int id, int hash_mod, int triangle_size, out int address)
6  {
7    bool store_sample = false;
8    int hash = id & hash_mod;
9    uvec4 b = imageLoad(cache, hash);
10   address = get_address_from_bucket(id, b);
11   for(int k = 0; address < 0 && k < 1024; k++) {
12     // ID not found in cache, make several attempts.
13     uint lock = imageAtomicExchange(locks, hash, LOCKED);
14     if(lock == UNLOCKED) {
15       // Gain exclusive access to the bucket.
16       b = imageLoad(cache, hash);
17       address = get_address_from_bucket(id, b);
18       if(address < 0) {
19         // Allocate new storage.
20         address = int(atomicAdd(ctr_ssid[1], triangle_size));
21         b.zw = b.xy; // Update bucket FIFO.
22         b.xy = uvec2(id, address);
23         imageStore(cache, hash, b);
24         store_sample = true;
25       }
26       imageStore(locks, hash, uvec4(UNLOCKED));
27     }
28     // Use if(expr){} if(!expr){} construct to explicitly
29     // sequence the branches.
30     if(lock == LOCKED) {
31       for(int i = 0; i < 128 && lock == LOCKED; i++)
32         lock = imageLoad(locks, hash).r;
33       b = imageLoad(cache, hash);
34       address = get_address_from_bucket(id, b);
35     }
36   }
37   if(address < 0) { // Cache lookup failed, store redundantly.
38     address = int(atomicAdd(ctr_ssid[1], triangle_size));
39     store_sample = true;
40   }
41   return store_sample;
42 }
```

Listing 3.1. The memoization cache uses several `imageBuffers` to store locks as well as cache entries. An access to the cache bucket determines if the cache contains the requested ID. If it is not found, all invocations concurrently try to acquire exclusive access to the cache where the winner is allowed to allocate memory. All other invocations repeatedly poll the cache to retrieve the updated cache entry.

terward. Since an `if-else` statement does not carry any notions about the first executed branch in case of divergence, this statement must be explicitly sequenced by dividing it into two disjunct statements when it contains side effects that require explicit ordering. This is important when implementing the memoization cache because invocations should be waiting for the updated cache buckets strictly following the update step; otherwise, deadlocks might occur.

```
1  void compute_attribute_derivatives(
2    in Triangle triangle, out TriangleDerivatives d)
3  {
4    mat3x4 pos; mat3x2 tex_coord; mat3 normal;
5    for(int i = 0; i < 3; i++) {
6      pos[i]       = P * vec4(triangle.positions[i], 1.0);
7      normal[i]    = triangle.normals[i];
8      tex_coord[i] = triangle.tex_coords[i];
9    }
10   // Clip triangle against all frustum planes.
11   for(int i = 0; i < 3; i++) {
12     shrink_triangle(pos, tex_coord, normal, i, true);
13     shrink_triangle(pos, tex_coord, normal, i, false);
14   }
15   vec3 one_over_w = 1.0 / vec3(pos[0].w, pos[1].w, pos[2].w);
16   vec2 pos_scr[3]; // projected vertices
17   for(int i = 0; i < 3; i++) {
18     pos_scr[i]    = pos[i].xy * one_over_w[i];
19     tex_coord[i] *= one_over_w[i];
20     normal[i]    *= one_over_w[i];
21   }
22   vec3 db_dx, db_dy; // Gradient barycentric coordinates x/y
23   compute_barycentric_derivatives(pos_scr, db_dx, db_dy);
24   // Compute derivatives in x/y for all attributes.
25   d.d_normal_dx = normal * db_dx;
26   d.d_normal_dy = normal * db_dy;
27   d.d_tex_dx    = tex_coord * db_dx;
28   d.d_tex_dy    = tex_coord * db_dy;
29   d.d_w_dx      = dot(one_over_w, db_dx);
30   d.d_w_dy      = dot(one_over_w, db_dy);
31   // Compute attributes shifted to (0,0).
32   vec2 o = -pos_scr[0];
33   d.one_by_w_fixed = one_over_w[0]
34                    + o.x * d.d_w_dx + o.y * d.d_w_dy;
35   d.tex_coord_fixed = tex_coord[0]
36                     + o.x * d.d_tex_dx + o.y * d.d_tex_dy;
37   d.normal_fixed = normal[0];
38                  + o.x * d.d_normal_dx + o.y * d.d_normal_dy;
39 }
```

Listing 3.2. Derivatives are computed according to Equation (3.5). First, the stored triangles are transformed into clip space and consecutively clipped against all view planes, which allows us to project them to the screen. The derivatives of the barycentric coordinates are computed according to Equation (3.1) to compute the partial derivatives for all attributes. Finally, the sample point of the attribute is extrapolated to the center of the screen to make the storage of the sample point's coordinate redundant.

3.3.3 Computing Partial Derivatives of Triangle Attributes

For the attribute interpolation, the partial derivatives need to be computed for each triangle. (Refer to Listing 3.2.) In theory it would be possible to compute the derivatives using the fragment shader built-in `dFdx,dFdy` functions. However, the numerical precision is not sufficient, and therefore the derivatives need to be

```
void shrink_triangle(inout mat3x4 pos, // Positions in clip space
          inout mat3x2 tex,            // Texture coordinates
          inout mat3   normal,         // Normals
          const int axis, const bool is_min) // Clip plane
{
  const int V0 = 1, V1 = 2, V2 = 4;
  uint clipmask = 0;
  if(is_min) {
    clipmask |= pos[0][axis] < -pos[0].w ? V0 : 0;
    clipmask |= pos[1][axis] < -pos[1].w ? V1 : 0;
    clipmask |= pos[2][axis] < -pos[2].w ? V2 : 0;
  } else {
    clipmask |= pos[0][axis] >  pos[0].w ? V0 : 0;
    clipmask |= pos[1][axis] >  pos[1].w ? V1 : 0;
    clipmask |= pos[2][axis] >  pos[2].w ? V2 : 0;
  }
  float a, b1, b2;
// Push the vertex on the edge from->to.
#define PUSH_VERTEX(from, to)                                  \
    b1 = is_min ? pos[to  ][axis] : -pos[to  ][axis]; \
    b2 = is_min ? pos[from][axis] : -pos[from][axis]; \
    a = (pos[to].w + b1)                                       \
      / (pos[to].w - pos[from].w + b1 - b2);                   \
    pos[from]    = mix(pos[to],    pos[from],    a);  \
    tex[from]    = mix(tex[to],    tex[from],    a);  \
    normal[from] = mix(normal[to], normal[from], a);

  // Only two vertices may be outside; otherwise,
  // the triangle would not be visible.
  switch(clipmask) {
  case V2|V0: PUSH_VERTEX(2, 1);
  case V0:    PUSH_VERTEX(0, 1); break;
  case V0|V1: PUSH_VERTEX(0, 2);
  case V1:    PUSH_VERTEX(1, 2); break;
  case V1|V2: PUSH_VERTEX(1, 0);
  case V2:    PUSH_VERTEX(2, 0); break;
  }
}
```

Listing 3.3. Shrinking a triangle to make it fit into the frustum. First, a bitmask is computed that indicates for each vertex if it is outside with respect to the current clip plane. This bitmask is used to determine which of the edges alongside the respective vertices are pushed.

computed manually in a separate pass after visibility of the triangles has been established.

For computing the partial derivatives of the attributes as described in Section 3.2.1, the triangles need to be projected to the screen, which necessitates clipping against the view frustum. Our implementation uses homogeneous clipping, however we do not create additional triangles since the derivatives are identical for all resulting clipped triangles.

A bitmask is computed (refer to Listing 3.3) that stores per vertex if it is outside with respect to the current clip plane. Since this computation considers

only visible triangles, at most two vertices may be outside with respect to a single clip plane. The bitmask is used to determine which vertices need to be pushed, and furthermore the corresponding triangle edge to the vertex lying inside the frustum is found. The intersection of the edge with the clip plane is computed, and the vertex is moved to the intersection point.

3.3.4 Shading

Attributes are interpolated according to Equation (3.3). By storing a material ID in the first field of the triangle struct, different materials and triangle storage formats can be identified. Akin to deferred shading, the world-space position could be reconstructed from the depth buffer; however, we reconstruct the world-space position from the $1/w$ attribute, allowing us to reconstruct the attribute precisely for arbitrary screen positions. First, the position $p_{\text{NDC}} = (x, y, z, 1)^T_{\text{NDC}}$ is computed from the screen-space coordinate; z_{NDC} is computed from the w-component as $z_{\text{NDC}} = P_{34}/w - P_{33}$, where P is the projection matrix. The world-space position p thus can be computed as $p = w \cdot (PV)^{-1} p_{\text{NDC}}$ with V as the view matrix.

For the shading the materials need to be evaluated. We use OpenGLs bindless texture mechanism for random access to the resources needed by the materials. A gradient is needed for the texture access to ensure proper texture filtering, which can be computed by interpolating the attribute offset to the neighboring pixel and by computing the difference to the attribute at the shaded pixel. We do not store the tangent space as an additional attribute but rather compute the tangent using screen-space derivatives [Schueler 07].

3.3.5 Linked List of Visibility Samples

Since, for most pixels, only a small number of different triangles is referenced, it is more memory efficient to dynamically allocate memory for a linked list and to reference that linked list using a per-pixel head pointer. Each linked list element stores a pointer to the next list element as well as a reference to the triangle. The coverage mask reported in the fragment shader is used to determine if a pixel is fully covered by the current triangle. The depth prepass ensures that correct coverage information is determined; however, the coverage information needs to take the depth test into account. The following code fragment shows how to determine the correct number of covered samples:

```
#extension GL_ARB_post_depth_coverage : require
layout(post_depth_coverage) in;
uint num_samples_covered = bitCount(gl_SampleMaskIn[0]);
```

In the case of full coverage, the head pointer is used to directly encode the triangle reference, which is indicated using a special bit in the head pointer. Otherwise,

```
#extension GL_NV_shader_thread_group: enable
uint sid = 0; // Shading sample address
if((gl_ThreadInWarpNV & 3) == 0) // One thread allocates memory.
  sid = atomicCounterIncrement(ctr_shading_samples);
// Communicate to all invocations.
uint sid_sw = floatBitsToUint(
      quadSwizzle0NV(uintBitsToFloat(sid)));
if(sid_sw == 0) { // Fails when there are helper-invocations.
  if(sid == 0) // Allocate shading samples for all invocations.
    sid = atomicCounterIncrement(ctr_shading_samples);
  store_shading_sample(sid);
} else if((gl_ThreadInWarpNV & 0x03) == 0) {
  sid = sid_sw;
  store_shading_sample(sid);
} else { // Communication worked, do not need to store.
  sid = sid_sw;
}
```

Listing 3.4. Multi-rate shading samples are created in the fragment shader by a specific invocation that then tries to broadcast this address to all invocations in the shading quad. If the broadcast fails, each invocation creates a shading sample, which might happen if there are helper-invocations.

the linked list is build similar to *order independent transparency* techniques by allocating samples using an atomic counter and performing an atomic exchange operation on the list head pointer. Alongside the triangle address, the number of covered samples is stored to allow for correct weighting of the samples in the shading phase.

3.3.6 Multi-rate Shading

Our multi-rate shading approach requires shading samples to be spawned that are referenced by the visibility buffer. Each of the shading samples stores a reference to the triangle to be shaded, as well as the screen-space coordinate to enable attribute interpolation. Our approach uses inter-shading-quad communication to determine which fragment shader invocation creates the shading sample and to communicate the address of the sample to all four invocations. Listing 3.4 shows our approach to communicate the shading sample address. First, one invocation allocates memory and tries to communicate the address to all other invocations in the quad. Next, all invocations check if the communication succeeded, as it might fail in case there are helper-invocations inside the shading quad. If the communication of the sample failed, each invocation creates a separate shading sample.

We issue a compute pass for all samples and compress the computed shading into the LogLuv [Larson 98] representation. The compressed result replaces the input needed for shading in-place. In the final shading phase, these samples are read by looking up the visibility buffer and are combined with the full shading.

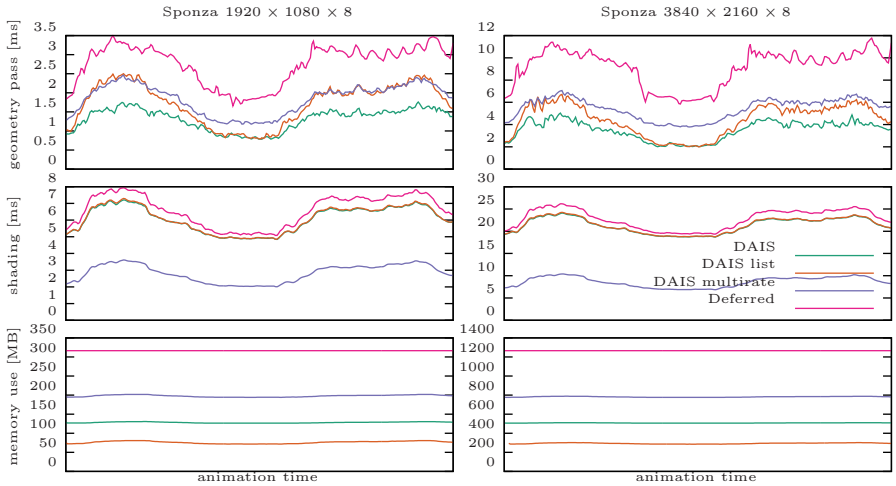


Figure 3.3. Timings and total memory usage for an animation in the Sponza scene. The geometry pass corresponds to the visibility sampling phase and includes the depth prepass. The memory consumption includes all buffers needed for shading. Note the varying y-axis scaling.

3.4 Results

We compare the performance characteristics and memory consumption of three versions of our algorithm to a standard deferred shading implementation. *DAIS* uses a standard multi-sample render target, whereas *DAIS list* employs a per-pixel linked list of visibility samples. Furthermore, we test our multi-rate shading implementation that reduces the shading rate for indirect illumination evaluated using reflective shadow maps [Dachsbacher and Stamminger 05] to 50%. Our deferred shading implementation uses a G-buffer format of 20 bytes per visibility sample.

Figure 3.3 shows our measurements for a camera animation in the Sponza scene, which has 262,267 triangles. Furthermore, we performed the measurements (refer to Figure 3.4) using the San Miguel scene, which has 8,145,860 triangles. On the Sponza scene our method is consistently able to outperform deferred shading while at the same time significantly reducing the storage consumption. Due to the large number of triangles, the San Miguel scene stresses our method, which is not able to meet the performance of deferred shading at a resolution of $1920 \times 1080 \times 8$; however, our method is able to outperform deferred shading at the higher screen resolution.

We furthermore evaluated our method using tessellation shaders (refer to Figure 3.5) to spawn equally-sized triangles in screen space on the Sponza scene. The performance characteristics are similar to the San Miguel scene as shown in Figure 3.4.

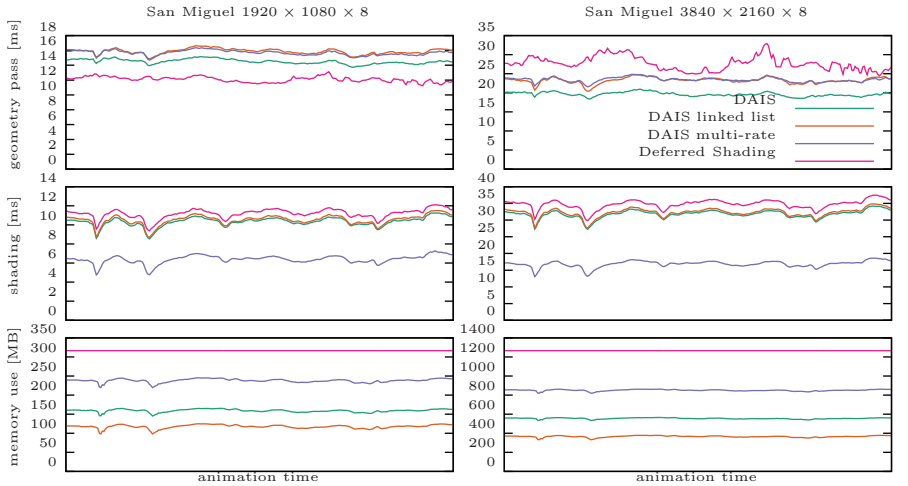


Figure 3.4. Timings and total memory usage for an animation in the San Miguel scene. The geometry pass corresponds to the visibility sampling phase and includes the depth prepass. The memory consumption includes all buffers needed for shading. Note the varying y-axis scaling.

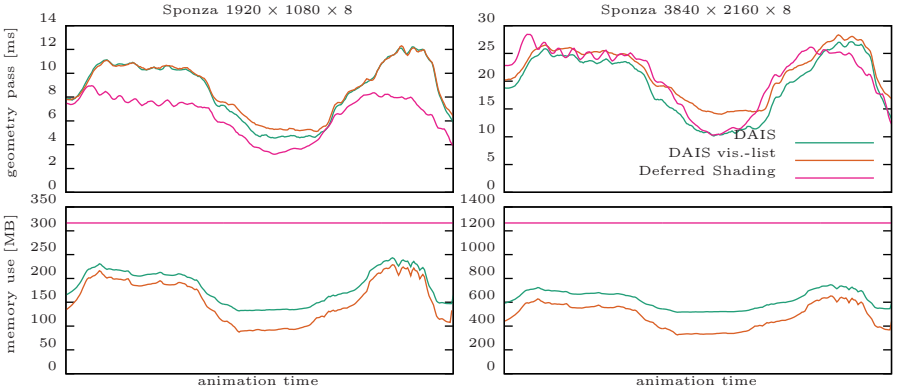


Figure 3.5. Timings and total memory usage for an animation in the Sponza scene. A tessellation shader was used to create triangles with approximately equal screen-space size, generating up to 13 and 26 million triangles for the resolutions of $1920 \times 1080 \times 8$ and $3840 \times 2160 \times 8$, respectively.

3.5 Conclusion

In this chapter we presented a memory-efficient deferred shading algorithm that makes the usage of multi-sample antialiasing in conjunction with high screen resolutions viable. Storing data per triangle instead of per visibility sample sig-

nificantly reduces the memory usage and allows us to employ caches efficiently, which makes the method faster and more memory efficient compared to deferred shading. The visibility buffer is of low entropy since many visibility samples store the same reference, which allows the GPU to effectively apply transparent memory compression to further reduce the memory bandwidth usage.

Bibliography

[Burns and Hunt 13] Christopher A. Burns and Warren A. Hunt. "The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading." *Journal of Computer Graphics Techniques (JCGT)* 2:2 (2013), 55–69. Available online (http://jcgt.org/published/0002/02/04/).

[Cigolle et al. 14] Zina H. Cigolle, Sam Donow, Daniel Evangelakos, Michael Mara, Morgan McGuire, and Quirin Meyer. "A Survey of Efficient Representations for Independent Unit Vectors." *Journal of Computer Graphics Techniques (JCGT)* 3:2 (2014), 1–30. Available online (http://jcgt.org/published/0003/02/01/).

[Dachsbacher and Stamminger 05] Carsten Dachsbacher and Marc Stamminger. "Reflective Shadow Maps." In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, *I3D '05*, pp. 203–231. New York: ACM, 2005.

[Larson 98] Gregory Ward Larson. "LogLuv Encoding for Full-Gamut, High-Dynamic Range Images." *Journal of Graphics Tools* 3:1 (1998), 15–31.

[Liktor and Dachsbacher 12] Gábor Liktor and Carsten Dachsbacher. "Decoupled Deferred Shading for Hardware Rasterization." In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 143–150. New York: ACM, 2012.

[Schied and Dachsbacher 15] Christoph Schied and Carsten Dachsbacher. "Deferred Attribute Interpolation for Memory-Efficient Deferred Shading." In *Proceedings of the 7th Conference on High-Performance Graphics*, pp. 1–5. New York: ACM, 2015.

[Schueler 07] Christian Schueler. "Normal Mapping without Pre-Computed Tangents." In *ShaderX5: Advanced Rendering Techniques*, edited by Wolfgang F Engel. Boston: Charles River Media, 2007.