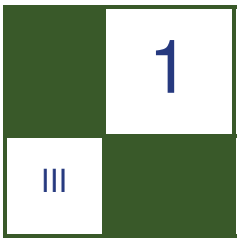

Deferred+: Next-Gen Culling and Rendering for the Dawn Engine

Hawar Doghramachi and Jean-Normand Bucci

1.1 Introduction

We present a comprehensive system for culling and rendering complex scenes in real time that we developed internally for future use in the Dawn Engine. This was one of several research projects done by the Eidos-Montréal R&D team, named Labs, but the system is not used in the game *Deus Ex: Mankind Divided*. The main goal of our research was to develop a system that could satisfy the growing demands on visual fidelity and runtime performance, but one that would remain compatible with traditional game assets and allow for fast iteration times during game production.

Our culling system combines the low latency and low overhead of a hierarchical depth buffer-based approach [Hill and Collin 2011] with the pixel accuracy of conventional GPU hardware occlusion queries. It efficiently culls highly dynamic, complex environments while maintaining compatibility with standard mesh assets. Our rendering system uses a practical approach to the idea of deferred texturing [Reed 2014] and efficiently supports highly diverse and complex materials while using conventional texture assets. Both parts of the proposed system make use of new graphics capabilities available with DirectX 12, most notably enhanced indirect rendering and the new shader resource binding model.

1.2 Overview

For real-time rendering applications like video games, it is crucial to employ an efficient occlusion culling system in order to be able to render large environments while ensuring high interactive frame rates. Traditional CPU-based culling sys-

tems, for example portal culling, fail to support dynamic, complex, alpha-tested occluders, and they are, therefore, not well-suited for such environments. Our culling system is partially based on the ideas presented by [Haar and Aaltonen 2015], where the depth buffer from the previous frame is used to acquire an initial visibility, and potential false negatives are retested with the updated depth buffer from the current frame. In this way, we avoid rendering dedicated occlusion geometry, which may be difficult to generate; for example, natural environments. However, instead of using a hierarchical depth buffer-based approach, a concept is utilized in the spirit of [Kubisch and Tavenrath 2014] that relies on the early depth-stencil testing capability of modern consumer graphics hardware. The main idea is to rasterize the boundaries of the occludees with the dedicated graphics hardware using a down-sampled and re-projected depth buffer from the prior frame. By forcing the associated pixel-shader to use early depth-stencil testing, only visible fragments mark in a common GPU buffer, at a location unique for each mesh instance, that the corresponding instance is visible. In comparison to a hierarchical depth buffer-based approach, this provides significantly higher culling efficiency with non-clustered, standard mesh assets. A subsequent compute shader generates, from the acquired visibility information, the data which is used later on for indirect rendering. As proposed by [Haar and Aaltonen 2015], occluded objects are retested with the updated depth buffer from the current frame to avoid missing false negatives.

For modern games, it is also important to utilize a rendering system that can handle increasingly complex mesh geometry and realistic surface materials. Forward rendering systems support high material diversity, but they either suffer from overdraw or require a depth pre-pass, which can be expensive for meshes with high triangle counts, GPU hardware tessellation, alpha-testing, or vertex-shader skinning. Deferred rendering systems manage to run efficiently without a depth pre-pass, but only support a limited range of materials and, therefore, often require additional forward rendering for more diverse materials. Our practical approach to deferred texturing combines the strengths of both rendering systems by supporting a high material diversity while only performing a single geometry pass. We go one step further than traditional deferred rendering and completely decouple geometry from materials and lighting. In an initial geometry pass, all mesh instances that pass the GPU culling stage are rendered indirectly, and their vertex attributes are written, compressed, into a set of geometry buffers. No material-specific operations and texture fetches are done (except for alpha testing and certain kinds of GPU hardware-tessellation techniques). A subsequent full-screen pass transfers a material ID from the geometry buffers into a 16-bit depth buffer. Finally, in the shading pass for each material, a screen-space rectangle is rendered that encloses the boundaries of all visible meshes. The depth of the rectangle vertices is set to a value that corresponds to the currently processed material ID and uses early depth-stencil testing to reject pixels from other materials. All standard materials that use the same shader and resource-binding layout are rendered in a single pass via dynamically

indexed textures. At this point, material-specific rendering and lighting (e.g., tiled [M. Billeter and Assarsson 2013] or clustered [Olsson et al. 2012]) are done simultaneously.

1.3 Implementation

In the following sections, each step of the system will be described in detail. The explanations are assuming the use of DirectX 12, but the system could be also implemented with OpenGL or Vulkan. No graphics hardware specific assumptions are made and the supplied shader code is hardware agnostic.

1.3.1 Culling

The culling part of the system can be subdivided into two passes, each consisting of several distinct steps (Figure 1.1). We decided to use instancing instead of a flattened mesh list. Even though draw calls will be generated on the GPU, internal tests showed that instancing, paired with a compacted indirect draw buffer, reduced GPU times for rendering geometry into the geometry buffers by approximately 45%.

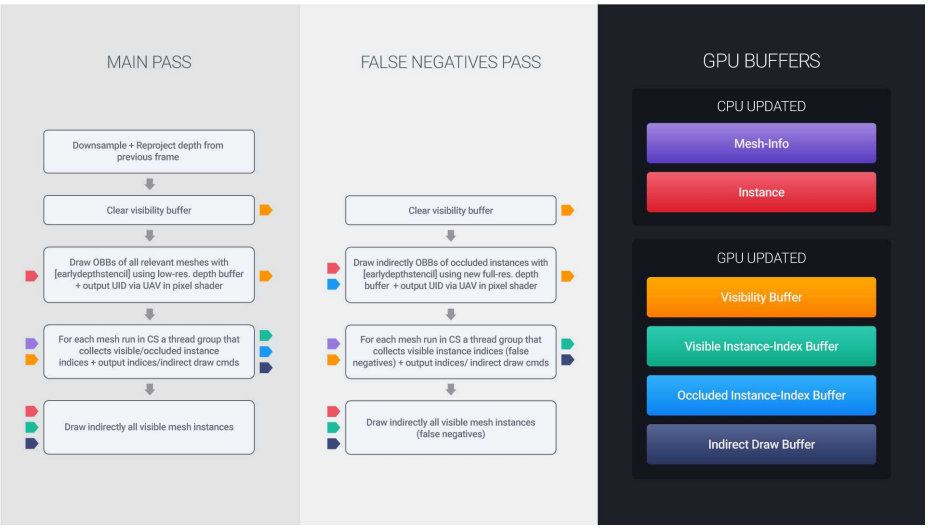

Figure 1.1. Overview of each step involved in the culling system. Grey blocks on the left side represent each culling step, blocks on the right side illustrate the required GPU resources. Arrows on the left side of each culling step represent input data, arrows on the right side output data. The colors match with those of the corresponding GPU buffers.

Main pass. The first pass acquires an initial visibility and can be subdivided into 5 distinct steps.

Depth buffer down-sampling and re-projection. In order to accelerate the algorithm, the depth buffer from the previous frame is first down-sampled to quarter resolution by taking, conservatively, the maximum value of 4×4 pixel areas. We found further down-sampling of the depth buffer did not provide any notable performance improvement and deteriorated culling efficiency. Since large planar objects can cause false self-occlusion, the down-sampled depth buffer is re-projected to the current frame. This is done by scattering the re-projected depth values to new pixel locations via atomic max operations on an unordered access view (UAV). Unfortunately, at high camera motion this produces holes that significantly lower culling efficiency. To close such holes, the re-projected depth values are additionally written to the currently processed pixel locations. Large re-projected depth values behind the camera can be an additional source of holes; in that case we use the non-re-projected depth from the last frame. In this way, high culling efficiency can be achieved even under high camera motion while false self-occlusion is significantly reduced (Figure 1.2).

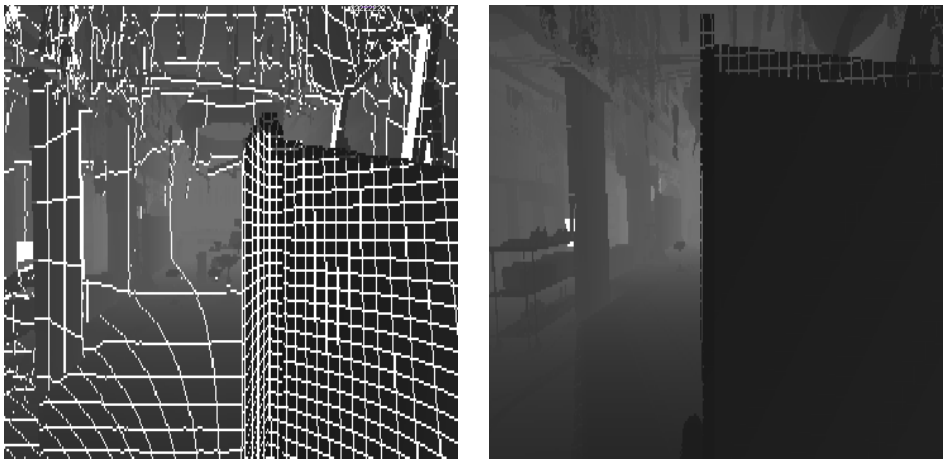

Figure 1.2. The left image shows the down-sampled, re-projected depth buffer from the last frame with many holes, significantly deteriorating culling efficiency. The right image shows how the proposed measures close such holes.

Listing 1.1 shows the HLSL code for a compute shader that down-samples and re-projects the depth buffer from the last frame. Since UAVs are not supported on depth buffers, the down-sampled and re-projected depth values are first written into a color buffer and then copied into the final depth buffer.

```
[numthreads(REPROJECT_THREAD_GROUP_SIZE, REPROJECT_THREAD_GROUP_SIZE, 1)]
void main(uint3 dispatchThreadID : SV_DispatchThreadID)
{
  if((dispatchThreadID.x < (uint(SCREEN_WIDTH)/4)) && (dispatchThreadID.y <
     (uint(SCREEN_HEIGHT)/4)))
  {
    const float2 screenSize = float2(SCREEN_WIDTH/4.0f, SCREEN_HEIGHT/4.0f);
    float2 texCoords = (float2(dispatchThreadID.xy) + float2(0.5f, 0.5f))/screenSize;

    const float offsetX = 1.0f/SCREEN_WIDTH;
    const float offsetY = 1.0f/SCREEN_HEIGHT;

    // down-sample depth (gather and max operations can be replaced by using a max
    // sampler if device supports MinMaxFiltering)
    float4 depthValues00 = depthMap.GatherRed(depthMapSampler, texCoords +
                           float2(-offsetX, -offsetY));
    float depth = max( max( max(depthValues00.x, depthValues00.y), depthValues00.z ),
                       depthValues00.w);

    float4 depthValues10 = depthMap.GatherRed(depthMapSampler, texCoords +
                           float2(offsetX, -offsetY));
    depth = max( max( max( max(depthValues10.x, depthValues10.y), depthValues10.z ),
                depthValues10.w), depth);

    float4 depthValues01 = depthMap.GatherRed(depthMapSampler, texCoords +
                           float2(-offsetX, offsetY));
    depth = max( max( max( max(depthValues01.x, depthValues01.y), depthValues01.z ),
                depthValues01.w), depth);

    float4 depthValues11 = depthMap.GatherRed(depthMapSampler, texCoords +
                           float2(offsetX, offsetY));
    depth = max( max( max( max(depthValues11.x, depthValues11.y), depthValues11.z ),
                depthValues11.w), depth);

    // reconstruct world-space position of last frame from down-sampled depth
    float4 lastProjPosition = float4(texCoords, depth, 1.0f);
    lastProjPosition.xy = (lastProjPosition.xy * 2.0f) - 1.0f;
    lastProjPosition.y = -lastProjPosition.y;
    float4 position = mul(cameraCB.lastInvViewProjMatrix, lastProjPosition);
    position /= position.w;

    // calculate projected position of current frame
    float4 projPosition = mul(cameraCB.viewProjMatrix, position);
    projPosition.xyz /= projPosition.w;
    projPosition.y = -projPosition.y;
    projPosition.xy = (projPosition.xy*0.5f) + 0.5f;
    int2 outputPos = int2(saturate(projPosition.xy) * screenSize);

    // prevent output of large depth values behind camera
    float depthF = (projPosition.w < 0.0f) ? depth : projPosition.z;

    // Convert depth into UINT for atomic max operation. Since bound color buffer is
    // initialized to zero, first invert depth, perform atomic max, and then
    // invert depth back when copied into final depth buffer.
    uint invDepth = asuint(saturate(1.0f - depthF));

    // write re-projected depth to new location
    InterlockedMax(depthTexture[outputPos], invDepth);

    // write re-projected depth to current location to handle holes from re-projection
    InterlockedMax(depthTexture[dispatchThreadID.xy], invDepth);
  }
}
```

Listing 1.1. Compute shader for down-sampling and re-projecting depth buffer from last frame.

Clear visibility buffer. The visibility buffer is a global structured buffer that is used to keep track of the visibility of all processed mesh instances. We avoid atomic memory operations in the next step by using a 32-bit UINT for each mesh instance in the visibility buffer. This leaves 31 bits per entry unused; however, even a visibility buffer that can handle one million mesh instances will consume only 4 MB of GPU memory, which results in a very good performance-memory trade-off. The entries of the visibility buffer are cleared to zero either by using the dedicated UAV clear API of DirectX 12 or by manually running a simple compute shader over the buffer.

Fill visibility buffer. In this step the oriented bounding boxes (OBBs) of all mesh instances, that passed frustum culling on the CPU, are rendered in a single indexed instanced draw call using the down-sampled, re-projected depth buffer from the previous step for depth testing. The associated pixel shader is flagged with [earlydepthstencil] so that only passed fragments will write, with the help of an UAV, to the unique mesh instance location in the visibility buffer. By avoiding atomic operations and ROPs, fast execution times can be achieved. Listing 1.2 shows the vertex and pixel shader for filling the visibility buffer.

```
// vertex shader
VS_Output main(uint vertexID: SV_VertexID, uint instanceID: SV_InstanceID)
{
  VS_Output output;

  output.occludeeID = instanceID;

  // generate unit cube position
  float3 position = float3(((vertexID & 0x4) == 0) ? -1.0f : 1.0f,
                           ((vertexID & 0x2) == 0) ? -1.0f : 1.0f,
                           ((vertexID & 0x1) == 0) ? -1.0f : 1.0f);

  matrix instanceMatrix = instanceBuffer[output.occludeeID];
  float4 positionWS = mul(instanceMatrix, float4(position, 1.0f));
  output.position = mul(cameraCB.viewProjMatrix, positionWS);

  // When camera is inside the bounding box, it is possible that the bounding box is
  // fully occluded even when the object itself is visible. Therefore, bounding box
  // vertices behind the near plane are clamped in front of the near plane to avoid
  // culling such objects.
  output.position = (output.position.w < 0.0f) ? float4(clamp(output.position.xy,
    float2(-0.999f, -0.999f), float2(0.999f, 0.999f)), 0.0001f, 1.0f)
     : output.position;

  return output;
}

// pixel shader
[earlydepthstencil]
void main(VS_Output input)
{
  visBuffer[input.occludeeID] = 1;
}
```

Listing 1.2. Vertex and pixel shader for filling visibility buffer.

OBBs are rendered using a second instance buffer, where each instance matrix is pre-combined with a scale/ bias matrix that transforms a unit axis-aligned bounding box (AABB) into the mesh AABB before the actual instance transform is applied. Since rendering instanced meshes with a low triangle count used to be suboptimal [Bilodeau 2014], we tried to render all OBBs in a single draw call without instancing. At an overall bounding-box count of approximately 7000 it turned out that this approach is slightly slower than the instanced approach (tested on a NVIDIA Geforce GTX 970). Therefore, we decided to render bounding boxes with instancing.

As with conventional hardware occlusion queries, it is possible that when the camera is inside the bounding box of an occludee that the corresponding mesh occludes its own bounding box, leading to false occlusion (Figure 1.3). Therefore, bounding box vertices behind the camera near plane are clamped directly in front of the camera near plane, in order to ensure that such objects are always marked as visible.

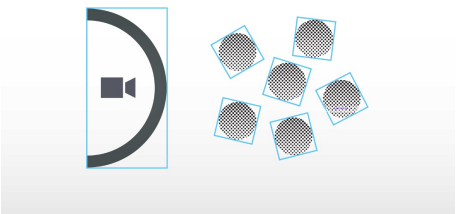

Figure 1.3. Since the object to the left occludes its own bounding box, the corresponding mesh is not rendered and is therefore missing in the updated depth buffer. Because the updated depth buffer doesn't contain this major occluder, almost all occluded objects to the right will be marked as visible and rendered later on.

Instead of using auto-computed bounding boxes, with the help of `Execute Indirect` it is also possible to optionally use artist-created meshes with a low triangle count to better approximate the shape of the meshes to be culled.

Generating indirect draw information. For each relevant mesh (at least one instance-passed frustum culling on the CPU), a compute shader thread group is dispatched. Each thread in the group checks the previously generated visibility buffer to see if the corresponding mesh instance is visible. Indices for all visible instances are written into the visible instance-index buffer, and for all occluded instances, into the occluded instance-index buffer. Additionally, if at least one instance of a mesh is found to be visible, a new indirect draw command is written into the indirect draw buffer. In order to support multiple mesh types (opaque, alpha-tested, tessellated, skinned, etc.), that require a different shader to efficiently fill the geometry buffers, indirect draw commands are written into the

indirect draw buffer from a mesh type specific offset. The number of indirect draw commands for each mesh type is accumulated into separate entries at the beginning of the visible instance-index buffer, and the number of occluded instances is written into the `InstanceCount` member of the first indirect draw command. This is used later on to render all occluded bounding boxes indirectly at once, to test for false negatives. These buffer entries have to be cleared before the compute shader is dispatched that generates the indirect draw information (Listing 1.3).

```
#define THREAD_GROUP_SIZE 64

groupshared uint visibleInstanceIndexCounter;

[numthreads(THREAD_GROUP_SIZE, 1, 1)]
void main(uint3 groupID: SV_GroupID, uint groupIndex: SV_GroupIndex,
          uint3 dispatchThreadID : SV_DispatchThreadID)
{
  if(groupIndex == 0)
  {
    visibleInstanceIndexCounter = 0;
  }
  GroupMemoryBarrierWithGroupSync();


  MeshInfo meshInfo = meshInfoBuffer[groupID.x];
  for(uint i=0; i<meshInfo.numInstances; i+=THREAD_GROUP_SIZE)
  {
    uint elementIndex = groupIndex + i;
    if(elementIndex < meshInfo.numInstances)
    {
      uint instanceIndex = meshInfo.instanceOffset + elementIndex;
      if(visBuffer[instanceIndex] > 0)
      {
        uint index;
        InterlockedAdd(visibleInstanceIndexCounter, 1, index);
        visibleInstanceIndexBuffer[meshInfo.instanceOffset+index+NUM_FILL_PASS_TYPES] =
          instanceIndex;
      }
      else
      {
        // Occluded instances will be rendered as occludees to determine false
        // negatives.
        uint index;
        InterlockedAdd(drawIndirectBuffer[0].instanceCount, 1, index);
        occludedInstanceIndexBuffer[index] = instanceIndex;
      }
    }
  }
  GroupMemoryBarrierWithGroupSync();


  if(groupIndex == 0)
  {
    if(visibleInstanceIndexCounter > 0)
    {
      // Increment counter of visible meshes.
      uint cmdIndex;
      InterlockedAdd(visibleInstanceIndexBuffer[meshInfo.meshType], 1, cmdIndex);
      cmdIndex += meshInfo.meshTypeOffset + 1;

```

```
52        // Visible instances will be rendered directly as meshes into GBuffers.
53        DrawIndirectCmd cmd;
54        cmd.instanceOffset = meshInfo.instanceOffset;
55        cmd.materialID = meshInfo.materialID;
56        cmd.indexCountPerInstance = meshInfo.numIndices;
57        cmd.instanceCount = visibleInstanceIndexCounter;
58        cmd.startIndexLocation = meshInfo.firstIndex;
59        cmd.baseVertexLocation = 0;
60        cmd.startInstanceLocation = 0;
61        drawIndirectBuffer[cmdIndex] = cmd;
62      }
63    }
64  }
```

Listing 1.3. Compute shader for generating indirect draw information.

Render indirectly visible meshes. In this step, all visible mesh instances are rendered into the geometry buffers, used later on for material rendering and lighting. For each mesh type (opaque, alpha-tested, tessellated, skinned, etc.), a separate `ExecuteIndirect` command is issued, using the previously generated indirect draw buffer as source for indirect draw calls. For each `ExecuteIndirect` command, the corresponding mesh type specific offset from the previous step is used as an offset into the indirect draw buffer, and the corresponding entry at the beginning of the visible instance-index buffer is used to specify the number of indirect calls. With the help of the visible instance-index buffer for each mesh instance, the transformation matrix can be retrieved from the instance buffer. It is necessary that each vertex knows at which offset to start reading from the visible instance-index buffer. This information is provided by a root constant parameter, specified at the beginning of each indirect draw command. A second root constant parameter provides the material ID, which is required to be written into the geometry buffers for subsequent rendering. Listing 1.4 shows the vertex shader that is used for rendering.

```
1  VS_Output main(VS_Input input, uint instanceID: SV_InstanceID)
2  {
3    VS_Output output;
4
5    uint instanceIndex = instanceInfoCB.instanceOffset + instanceID;
6
7    // first members of buffer are counters for visible indirect draw commands
8    instanceIndex = visibleInstanceIndexBuffer[instanceIndex + NUM_MESH_TYPES];
9
10   matrix transformMatrix = instanceBuffer[instanceIndex].transformMatrix;
11   ...
12 }
```

Listing 1.4. Vertex shader for rendering indirectly visible meshes.

False negatives pass. Since, in the main pass, visibility culling is performed using the down-sampled, re-projected depth buffer from the last frame, false negatives can occur. The first obvious reason for this are errors introduced by re-projecting information obtained from the previous frame. The second

reason is that, due to performance considerations, bounding boxes are rendered in the first occlusion pass at quarter resolution. Thus, it is possible that very small bounding boxes in screen space fail to be rasterized and, therefore, the corresponding objects will be marked as occluded (Figure 1.4).

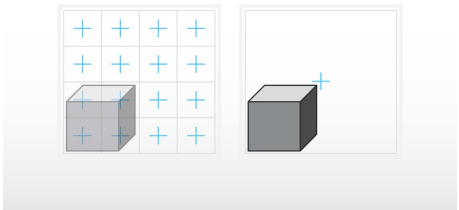

Figure 1.4. Very small bounding boxes in screen space can fail to be rasterized at quarter resolution.

To prevent such objects being incorrectly culled, potential false negatives have to be checked against a full resolution depth buffer. In theory, with conservative rasterization, the second occlusion pass could use a quarter resolution depth buffer too. However, we tested this approach and encountered multiple problems:

- The graphics card we used for testing (NVIDIA GeForce GTX 970, second generation Maxwell architecture) only supports tier 1 of conservative rasterization. Unfortunately, tier 1 culls very small triangles that get degenerated due to sub-pixel grid snapping, thus potentially culling visible objects. Moreover, tier 1 has a very large uncertainty region (half of the size of a pixel), leading to over-conservativeness and significantly lower culling efficiency. Though these problems are addressed by tier 2, at the time of writing, Intel's Skylake architecture was the only one supporting this tier.

- In our tests at 1920×1080 screen resolution, the time taken to downsample the current depth buffer to quarter resolution was higher than the benefit obtained from using a quarter resolution depth buffer with conservative rasterization for the second occlusion pass. Therefore, the second occlusion pass uses the depth buffer from the current frame at full resolution, obtained after rendering the visible meshes in the first occlusion pass, to detect false negatives and can be subdivided into 4 distinct steps.

Clear visibility buffer. As was done in the main occlusion pass, the entries of the visibility buffer are cleared to zero.

Fill visibility buffer. Analog to the main occlusion pass, the OBBs of all occluded mesh instances are rendered in a single indexed instanced draw call. But this

time, rendering is done indirectly using the first entry of the indirect draw buffer that was reserved for occluded objects. With the help of the occluded instance-index buffer, the transformation matrices can be retrieved from the instance buffer in order to transform the bounding boxes. As in the main occlusion pass, the pixel shader marks each instance as visible in the visibility buffer if at least one fragment passes the early depth-stencil test. As described above, the depth buffer of the current frame is used at full resolution for depth testing. It is not necessary to execute the clamping code for bounding-box vertices behind the camera near plane, since all problematic objects were rendered anyway in the first pass.

Generate indirect draw information. Indirect draw information is generated as in the main occlusion pass. However, this time only visible instance draw information is generated, as no further processing is required for occluded instances.

Render indirectly false negatives. This step doesn't differ from the corresponding step in the main occlusion pass and ensures that only occluded objects are culled by the proposed system.

1.3.2 Rendering

The rendering part of the system can be subdivided into 3 distinct steps (Figure 1.5).

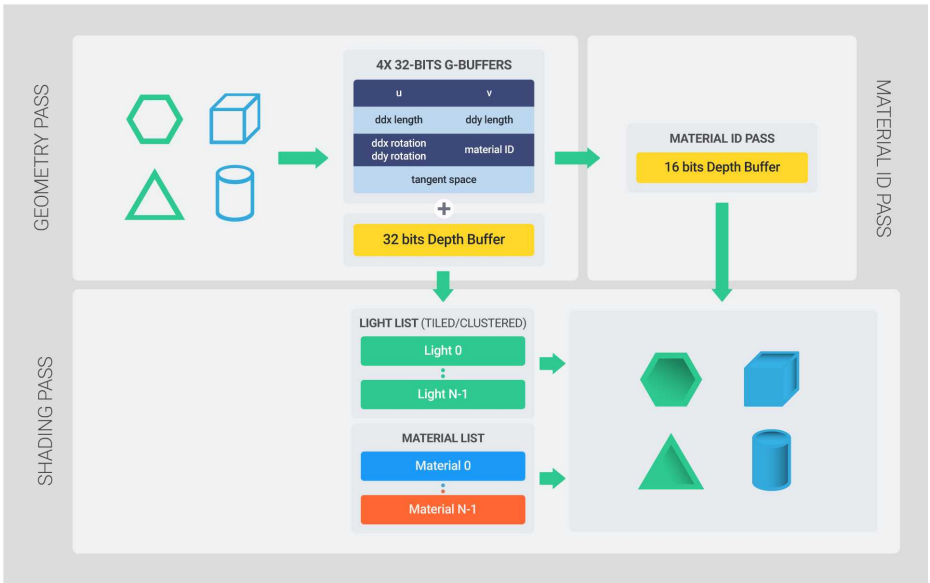

Figure 1.5. Overview of the steps involved in the rendering system.

Geometry pass. All mesh instances that pass the culling system are rendered indirectly into a set of compressed geometry buffers (4×32 bits per pixel). In contrast to traditional deferred rendering, no material specific code is executed or textures fetched except for some special cases, such as alpha testing with an alpha-mask texture or GPU hardware tessellation with displacement mapping that samples a height map. By outputting vertex attributes into the geometry buffers that are required for subsequent material rendering and lighting, it is possible to completely separate geometry processing from materials and lighting.

Geometry buffer layout. Since, in general, memory bandwidth of current consumer graphics hardware is much more limited than computational power, it is important to keep the size of the geometry buffers as low as possible. In the following section we explain how the required vertex attributes are compressed into the geometry buffers.

Texture coordinates. To avoid precision artifacts, texture coordinates need to be stored with at least 2×16 bits per pixel. As soon as texture coordinates are wrapped, i.e., exceed the $[0, 1]$ range, 16 bits per component is generally not enough anymore. Fortunately, there is an easy way to store wrapped texture coordinates with enough precision in 2×16 bits per pixel, by storing only their fractional part in the pixel shader after interpolation. Since the derivatives of the original texture coordinates are stored alongside, no seams will be visible later on. Texture coordinates are stored as `DXGI_FORMAT_R16G16_SNORM` in the first geometry buffer.

Texture coordinate derivatives. In theory, derivatives can be reconstructed in the shading pass by using the neighbor texture coordinates. However, in the case of geometry edges, where appropriate neighbor texture coordinates can't always be obtained, artifacts will be visible. This is especially noticeable under camera motion with dense alpha-tested foliage, where temporal artifacts can be observed. Therefore, we decided to store the texture coordinates along with their derivatives.

Texture coordinate derivatives require at least 4×16 bits per pixel for enough precision. However, it is possible to do an artifact-free compression from 64 bits to 48 bits by treating the derivatives in the X- and Y-direction as 2D vectors. By decoupling the vector length from the orientation, the vector length can be stored as 2×16 bits, and the orientation as 2×8 bits, which still gives enough precision for anisotropic texture filtering. Listing 1.5 shows how derivatives can be en/decoded in HLSL. The lengths of the derivatives are stored as `DXGI_FORMAT_R16G16_FLOAT` in the second geometry buffer, and the orientations in the red channel of the third geometry buffer `(DXGI_FORMAT_R16G16_UINT)`.

```
1 void EncodeDerivatives(in float4 derivatives, out float2 derivativesLength, out uint
2                        encodedDerivativesRot)
3 {
```

```
4    derivativesLength = float2(length(derivatives.xy), length(derivatives.zw));
5    float2 derivativesRot = float2(derivatives.x/derivativesLength.x,
6                                   derivatives.z/derivativesLength.y) * 0.5f + 0.5f;
7    uint signX = (derivatives.y < 0) ? 1 : 0;
8    uint signY = (derivatives.w < 0) ? 1 : 0;
9    encodedDerivativesRot = (signY << 15u) | (uint(derivativesRot.y * 127.0f) << 8u) |
10                           (signX << 7u) | uint(derivativesRot.x*127.0f));
11 }
12
13 float4 DecodeDerivatives(in uint encodedDerivativesRot, in float2 derivativesLength)
14 {
15   float2 derivativesRot;
16   derivativesRot.x = float(encodedDerivativesRot.r & 0x7f) / 127.0f;
17   derivativesRot.y = float((encodedDerivativesRot >> 8) & 0x7f) / 127.0f;
18   derivativesRot = derivativesRot * 2.0f - 1.0f;
19   float signX = (((encodedDerivativesRot.r >> 7u) & 0x1) == 0) ? 1.0f : -1.0f;
20   float signY = (((encodedDerivativesRot.r >> 15u) & 0x1) == 0) ? 1.0f : -1.0f;
21   float4 derivatives;
22   derivatives.x = derivativesRot.x;
23   derivatives.y = sqrt(1.0f - derivativesRot.x * derivativesRot.x)*signX;
24   derivatives.z = derivativesRot.y;
25   derivatives.w = sqrt(1.0f - derivativesRot.y * derivativesRot.y)*signY;
26   derivatives.xy *= derivativesLength.x;
27   derivatives.zw *= derivativesLength.y;
28   return derivatives;
29 }
```

Listing 1.5. HLSL code for en/de-coding texture coordinate derivatives.

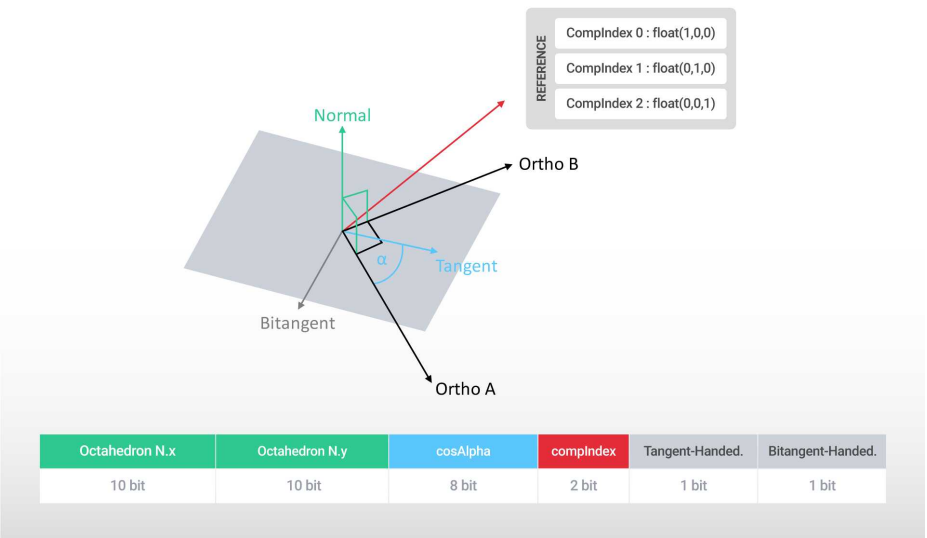

Figure 1.6. Relationship of the vectors involved in en/de-coding the tangent space and storage layout of the corresponding geometry buffer.

Tangent space. For normal mapping, parallax occlusion mapping, etc. it is required to store the tangent space, consisting of a tangent, bitangent and normal. We looked into storing the tangent space as a quaternion in 32 bits per pixel according to [McAuley 2015], but dismissed this approach due to visible faceting on smooth shiny surfaces. Instead, we store the tangent space as an axis-angle representation, where the normal is stored as an axis and the tangent as an angle. Normals are stored in 2×10 bits by using octahedron normal vector encoding [Meyer et al. 2010]. For tangents, first a vector has to be found that can be easily reconstructed for decoding and that is guaranteed to be orthonormal to the normal. Choosing an arbitrary orthonormal vector produces severe artifacts due to singularity issues and noise at different frequencies. To overcome these issues, we first select a reference vector by taking the largest component of the tangent and store this vector into two bits for decoding. With the help of this vector, we calculate an orthonormal vector and store the angle between this vector and the tangent into eight bits. The handedness of the tangent and bitangent are both stored in one bit. In this way, we can store the entire tangent space as `DXGI_FORMAT_R10G10B10A2_UINT` in the fourth geometry buffer (Figure 1.6, Listing 1.6). It should be noted that this method requires about half the instruction count to encode the tangent space into 32 bits as when converting a TBN matrix into a quaternion in a mathematically stable, precise manner and packing it into 32 bits.

```
uint4 EncodeTBN(in float3 normal, in float3 tangent, in uint bitangentHandedness)
{
  // octahedron normal vector encoding
  uint2 encodedNormal = uint2((EncodeNormal(normal) * 0.5f + 0.5f) * 1023.0f);

  // find largest component of tangent
  float3 tangentAbs = abs(tangent);
  float maxComp = max(max(tangentAbs.x, tangentAbs.y), tangentAbs.z);
  float3 refVector;
  uint compIndex;
  if(maxComp == tangentAbs.x)
  {
    refVector = float3(1.0f, 0.0f, 0.0f);
    compIndex = 0;

  else if(maxComp == tangentAbs.y)
  {
    refVector = float3(0.0f, 1.0f, 0.0f);
    compIndex = 1;
  }
  else
  {
    refVector = float3(0.0f, 0.0f, 1.0f);
    compIndex = 2;
  }

  // compute cosAngle and handedness of tangent
  float3 orthoA = normalize(cross(normal, refVector));
  float3 orthoB = cross(normal, orthoA);
  uint cosAngle = uint((dot(tangent, orthoA) * 0.5f + 0.5f) * 255.0f);
  uint tangentHandedness = (dot(tangent, orthoB) > 0.0001f) ? 2 : 0;

```

```
33    return uint4(encodedNormal, (cosAngle<<2u) | compIndex, tangentHandedness |
34                 bitangentHandedness);
35  }
36
37  void DecodeTBN(in uint4 encodedTBN, out float3 normal, out float3 tangent, out float3
38                 bitangent)
39  {
40    // octahedron normal vector decoding
41    normal = DecodeNormal((encodedTBN.xy / 1023.0f) * 2.0f - 1.0f);
42
43    // get reference vector
44    float3 refVector;
45    uint compIndex = (encodedTBN.z & 0x3);
46    if(compIndex == 0)
47    {
48      refVector = float3(1.0f, 0.0f, 0.0f);
49    }
50    else if(compIndex == 1)
51    {
52      refVector = float3(0.0f, 1.0f, 0.0f);
53    }
54    else
55    {
56      refVector = float3(0.0f, 0.0f, 1.0f);
57    }
58
59    // decode tangent
60    uint cosAngleUint = ((encodedTBN.z >> 2u) & 0xFF);
61    float cosAngle = (cosAngleUint / 255.0f) * 2.0f - 1.0f;
62    float sinAngle = sqrt(saturate(1.0f - (cosAngle * cosAngle)));
63    sinAngle = ((encodedTBN.w & 0x2) == 0) ? -sinAngle : sinAngle;
64    float3 orthoA = normalize(cross(normal, refVector));
65    float3 orthoB = cross(normal, orthoA);
66    tangent = (cosAngle * orthoA) + (sinAngle * orthoB);
67
68    // decode bitangent
69    bitangent = cross(normal, tangent);
70    bitangent = ((encodedTBN.w & 0x1) == 0) ? bitangent : -bitangent;
71  }
```

Listing 1.6. HLSL code for en/de-coding the tangent space.

The quality that can be achieved with this compression method is nearly equivalent to storing tangent, bitangent and normal uncompressed in 3×30 bits per pixel (Figure 1.7).

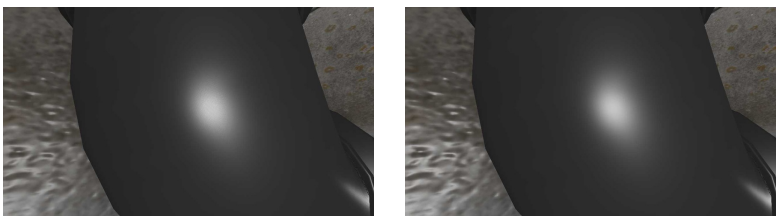

Figure 1.7. Both images show a smooth shiny surface illuminated by a spherical area light source with GGX as the specular lighting term. For the left image, tangent space was stored uncompressed in 3×30 bits per pixel and for the right image, tangent space was store with the proposed compression scheme in 32 bits per pixel.

Material ID To apply deferred materials, each material must store a unique ID. This material ID is stored in 16 bits as the green channel of the third geometry buffer (`DXGI_FORMAT_R16G16_UINT`), and supports up to 65,536 individual materials, which should be enough even for large-scale projects.

Additional vertex attributes. In practice, it can be possible that additional vertex attributes are required, such as multiple texture coordinate sets and vertex colors. As long as the texture coordinates that are stored in the geometry buffers are not wrapped (in [0,1] range), it should be possible to reconstruct additional texture coordinate sets per material that are just scaled and biased relative to the stored texture coordinates. Proper derivatives can then be obtained by scaling the stored derivatives.

For all other cases, such meshes can be treated for culling as separate mesh types, which allows them to write additional vertex attributes into further geometry buffers. In the shading pass, the corresponding materials are rendered separately as described later, which allows efficient fetching of additional geometry buffers. Obviously this is only a feasible solution as long as the number of additional vertex attributes is low or such materials are not used frequently.

Material ID pass. After filling the geometry buffers, the stored material ID is transferred into a 16-bit depth buffer in a simple full-screen pass. This is done by dividing the material ID with the maximum number of supported materials and outputting this value via `SV_Depth` in the pixel shader. For standard materials, that use the same shader and are rendered together in the shading pass, a special reserved depth value (e.g., 0) is output.

Shading pass. In this pass materials and lighting are applied. For all standard materials that use the same shader and resource binding layout, a single full-screen pass is executed and required textures are fetched by dynamically indexing into a common shader-resource descriptor table with the help of the material ID from the geometry buffers.

However, processing materials together that require different shader paths in an Übershader is a bad idea, since adjacent screen pixels can use completely different shader paths. This results in divergent dynamic branching with poor performance characteristics. Instead, for each non-standard material, a screen-space rectangle is rendered that encloses the boundaries of all visible meshes that use this material. The depth of the rectangle vertices is set to the same depth that was output for this material in the material ID pass. By setting depth comparison to equal, early depth-stencil testing will prevent pixels with a different material ID from being processed further. Since the same binary depth value is used as was output in the material ID pass, equal depth testing doesn't produce any precision issues and incorrect material classification is avoided. The same rejection technique is used for commonly rendered standard materials to

prevent pixels with non-standard materials from being processed by using the same special, reserved depth value as was output in the material ID pass.

The calculation of the screen-space rectangle boundaries is done on the GPU. For this, the indices of all mesh instances that were frustum-culled on the CPU, are stored per material in a contiguous GPU buffer. A second GPU buffer stores, per material, the first and last index into the former buffer. Additionally, separate visibility buffers have to be used for the first and second occlusion pass, in order to be able to check each instance for visibility. A compute shader is dispatched that runs a thread group per material, where each thread processes one mesh instance. If a mesh instance is visible, its OBB is projected into screen space and a bounding rectangle is inflated in the shared thread group memory. Finally, each thread group outputs the inflated bounding rectangle, in a structured buffer, at a position specific for each material (Listing 1.7). Consequently, this structured buffer is manually fetched in a vertex shader with `SV_VertexID` to construct the corners of the bounding rectangle in clip space for each processed material.

```
#define GROUP_SIZE 1024

groupshared uint2 iBoundsMins, iBoundsMaxes;

[numthreads(GROUP_SIZE, 1, 1)]
void main(uint3 groupID: SV_GroupID, uint groupIndex: SV_GroupIndex)
{
  if(groupIndex == 0)
  {
    iBoundsMins = uint2(0xffffffff, 0xffffffff);
    iBoundsMaxes = uint2(0, 0);
  }
  GroupMemoryBarrierWithGroupSync();

  MaterialBoundsInfo materialBoundsInfo = materialBoundsInfoBuffer[groupID.x];
  for(uint i=materialBoundsInfo.startMaterialIndex;
      i<materialBoundsInfo.endMaterialIndex; i+=GROUP_SIZE)
  {
    uint materialIndex = groupIndex + i;
    if(materialIndex < materialBoundsInfo.endMaterialIndex)
    {
      uint instanceIndex = materialIndexBuffer[materialIndex];

      // check if mesh is visible in visibility buffer of main or false negatives pass
      if((visBuffer0[instanceIndex] > 0) || (visBuffer1[instanceIndex] > 0))
      {
        // calculate AABB in view space
        const matrix modelViewMatrix = mul(cameraCB.viewMatrix,
                                           instanceBuffer[instanceIndex]);

        float3 boxMins = float3(999999.0f, 999999.0f, 999999.0f);
        float3 boxMaxes = float3(-999999.0f, -999999.0f, -999999.0f);

        [unroll]
        for(uint j=0; j<8; j++)
        {
          // unit AABB position
          float3 position = float3(((j & 0x4) == 0) ? -1.0f : 1.0f,
                                   ((j & 0x2) == 0) ? -1.0f : 1.0f,
```

```
                                    ((j & 0x1) == 0) ? -1.0f : 1.0f);

          float3 positionVS = mul(modelViewMatrix, float4(position, 1.0f)).xyz;
          boxMins = min(boxMins, positionVS);
          boxMaxes = max(boxMaxes, positionVS);
        }


        // clip view space AABB against camera near plane and calculate screen space
        // bounding rectangle
        float2 mins = float2(1.0f, 1.0f);
        float2 maxes = float2(0.0f, 0.0f);

        [unroll]
        for(uint k=0; k<8; k++)
        {
          float3 positionVS = float3(((k & 0x4) == 0) ? boxMins.x : boxMaxes.x,
                                     ((k & 0x2) == 0) ? boxMins.y : boxMaxes.y,
                                     ((k & 0x1) == 0) ? boxMins.z : boxMaxes.z);
          positionVS.z = (positionVS.z > 0.0f) ? 0.0f : positionVS.z;

          float4 positionCS = mul(cameraCB.projMatrix, float4(positionVS, 1.0f));
          positionCS.xy /= positionCS.w;
          float2 positionSS = saturate(positionCS.xy * 0.5f + 0.5f);
          mins = min(mins, positionSS);
          maxes = max(maxes, positionSS);
        }

        // inflate bounding rectangle in screen space
        uint2 iMins = asuint(mins);
        uint2 iMaxes = asuint(maxes);
        InterlockedMin(iBoundsMins.x, iMins.x);
        InterlockedMin(iBoundsMins.y, iMins.y);
        InterlockedMax(iBoundsMaxes.x, iMaxes.x);
        InterlockedMax(iBoundsMaxes.y, iMaxes.y);
      }
    }
  }
  GroupMemoryBarrierWithGroupSync();

  // store bounding rectangle of material in clip space
  if(groupIndex == 0)
  {
    float4 bounds = (iBoundsMins.x == 0xffffffff) ?
                    float4(0.0f, 0.0f, 0.0f, 0.0f) :
                    (float4(asfloat(iBoundsMins), asfloat(iBoundsMaxes)) * 2.0f - 1.0f);
    materialBoundsBuffer[groupID.x] = bounds;
  }
}
```

Listing 1.7. Compute shader that generates boundaries for materials.

Though early depth-stencil testing rejects pixels with different materials, current consumer GPUs run in warps of 32 or 64 threads and use helper pixels to ensure that pixels are processed in 2×2 quads for derivative calculations. Thus, even when irrelevant pixels are rejected, there could be a large amount of processed helper pixels and inactive GPU threads. However, we also implemented a rendering prototype in OpenGL and measured the number of active (memory-outputting), helper and inactive GPU threads with the help of the OpenGL extension `GL_NV_shader_thread_group` (Listing 1.8).

```
uint activeThreadsMask = activeThreadsNV();
uint helperThreadMask = ballotThreadNV(gl_HelperThreadNV);
uint outputThread;
for(uint i=0; i<32; i++)
{
  uint bit = 1 << i;
  if(((activeThreadsMask & bit) != 0) && ((helperThreadMask & bit) == 0))
  {
    outputThread = i;
    break;
  }
}
if(gl_ThreadInWarpNV == outputThread)
{
  uint numActiveThreads = bitCount(activeThreadsMask);
  uint numInactiveThreads = 32 - numActiveThreads;
  atomicAdd(threadCounterBuffer.counters.numActiveThreads, numActiveThreads);
  atomicAdd(threadCounterBuffer.counters.numInactiveThreads, numInactiveThreads);
  uint numHelperThreads = bitCount(helperThreadMask);
  atomicAdd(threadCounterBuffer.counters.numHelperThreads, numHelperThreads);
}
```

Listing 1.8. GLSL pixel shader code to count number of active, inactive and helper GPU threads. Only active GPU threads output to memory.

It turned out that the number of helper and inactive GPU threads for material rendering and lighting was significantly lower in comparison to a forward renderer and drastically lower with the use of GPU hardware tessellation techniques. This was even true for situations where each material was processed in a separate pass and the amount of alpha-tested foliage was high.

The reason that GPU hardware tessellation performs much better with deferred+ is that, even with the use of adaptive tessellation techniques, the number of small triangles drastically increases. As the number of small triangles increases, so does the number of helper and inactive GPU threads for pixel shading. This makes per pixel operations inefficient. Since, with deferred+, most of the per-pixel operations are deferred from the geometry pass to subsequent screen passes with high warp utilization, the negative impact of small triangles is far less noticeable.

We also tried an alternative rendering strategy. After the initial geometry pass, with the help of atomic counters, the number of pixels with the same material ID is counted and corresponding chunks are reserved in a common GPU buffer. Each pixel's screen location is then recorded into the chunk with the corresponding material ID. After that, the number of thread groups necessary to process the pixels for each material is written into a GPU buffer by dividing the pixel count by the number of threads per group. Finally, for each material, an indirect dispatch command is issued, sourcing the number of thread groups from the aforementioned GPU buffer. In this step, materials and lighting are applied. Unfortunately, this system performed significantly slower than the early depth-stencil approach. On the one hand, there was an additional overhead for binning the screen pixels per material. On the other hand, shading the pixels was also

slower than in the early depth-stencil approach. We thought the reason for this was that binned pixels did not have a spatial locality that matches the texture memory swizzling pattern of the fetched geometry buffer textures. Therefore, we also tried to bin pixels in screen tiles to achieve better texture cache efficiency, but couldn't achieve any notable performance improvement.

As mentioned previously, lighting can be done with a tiled [M. Billeter and Assarsson 2013] or clustered [Olsson et al. 2012] approach.

1.4 Comparison with Hierarchical Depth Buffer-based Culling

Similar GPU-based culling systems have already been proposed and used in games [Hill and Collin 2011, Haar and Aaltonen 2015]. However, these systems are based on a hierarchical depth buffer that is generated by conservative down-sampling. Occludee bounding boxes are manually projected into screen space and tested against the hierarchical depth buffer to determine visibility. This approach results in several problems that can be avoided with the proposed culling system:

- To determine visibility, occludee bounding boxes are projected in screen space into rectangular regions, which in some situations can significantly reduce culling efficiency (Figure 1.8).

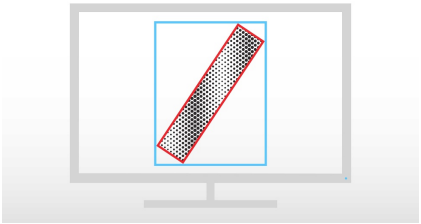

Figure 1.8. Viewport, showing an oblique object in screen space. With hierarchical depth buffer-based culling, the blue boundaries are used for visibility determination instead of the red boundaries that are used in our culling approach, resulting in lower culling efficiency.

- To determine visibility, a single conservative depth value is used for each occludee, which leads to false positives in some situations (Figure 1.9).
- In order to be able to test each projected bounding box against the hierarchical depth buffer with a fixed amount of texture samples, higher mip-map levels have to be used for larger occludees. The problem with this approach is that mip-maps were generated by conservative down-sampling, i.e., always taking the maximum of the depth values from the previous mip-map

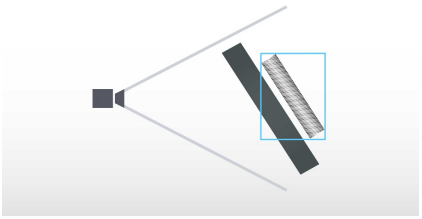

Figure 1.9. Top-down view of the camera frustum, where the object to the right is fully occluded, but since conservatively only one depth value is used, the object will still pass as visible.

levels. In consequence, large depth values (e.g., from the sky) propagate into higher mip-map levels thus making culling of larger objects in screen space inefficient (Figure 1.10).

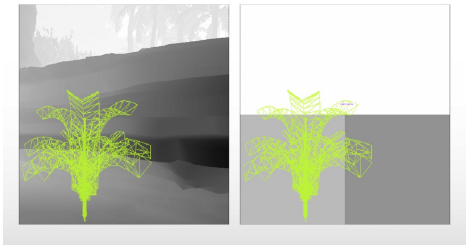

Figure 1.10. The object shown in wireframe is fully occluded but still passes as visible because a high mip-map level is used from the hierarchical depth buffer.

- If we would use a hierarchical depth buffer-based approach for the previously described two-pass culling system, first a hierarchical depth buffer would need to be constructed from the depth buffer of the last frame and then, after the main occlusion pass, reconstructed with the updated depth buffer of the current frame. On a PC, where Hi-Z tiles of the depth buffer can't be accessed, this has a high performance overhead compared to the remaining steps of the culling system.

The first three issues can be mitigated by subdividing mesh assets into small clusters [Haar and Aaltonen 2015]. However, in our case we wanted a culling system that does not rely on mesh clustering and is compatible with a traditional asset pipeline. Since scenes that were built with the current asset pipeline of the Dawn Engine consisted of relatively small modular blocks, with the proposed culling system we could avoid introducing a system for subdividing meshes into small clusters.

We compared how well the proposed culling system performs, in comparison to a hierarchical depth buffer-based culling approach, inside a natural jungle environment without using mesh clustering. On average we could achieve 2.3× higher culling rates and 1.6× faster frame times with the proposed culling system.

1.5 Pros and Cons

The following subsections give an overview of the pros and cons of the culling and rendering parts of the proposed system.

1.5.1 Culling

Pros

- Achieves the same pixel accuracy as conventional hardware occlusion queries while eliminating latency issues (popping).
- Highly dynamic, complex, alpha-tested occluders are supported without needing to author and render dedicated occluder geometry.
- High culling efficiency can be achieved without mesh clustering for modular composited scenes, thus maintaining full compatibility with standard mesh assets and asset pipelines.
- The system is compatible with CPU-based culling systems (such a frustum culling, portal culling, etc.) that can be considered as coarse pre-filtering.
- The culling system itself has a low performance overhead.
- The number of draw calls is massively reduced, which gives a performance benefit even with low-overhead graphics APIs such as DirectX 12 and Vulkan.

Cons

- Since the indirect draw buffer is generated in parallel in a compute shader, draw commands are no longer in a deterministic order. Thus, nearly coplanar surfaces are more likely to cause Z-fighting and should be avoided. Furthermore, depth sorting of draw calls is not given anymore, causing higher overdraw. However, since with deferred+ the geometry pass is light-weight and culling efficiency is high, this will have much less negative impact on performance than with traditional rendering systems.
- In situations where the camera changes drastically from one frame to the other (e.g., with teleportation), culling efficiency can significantly drop for a short amount of time.

1.5.2 Rendering

Pros

- Due to a light-weight geometry pass, a depth pre-pass, which can be expensive for meshes with high triangle counts, GPU hardware tessellation, alpha-testing or vertex-shader skinning, is no longer required,
- Warp utilization for applying materials and lighting is significantly better than with clustered forward shading [Olsson et al. 2012]. Thus small triangles are far less problematic, and GPU hardware tessellation performs much better.
- Deferred+ is a unified rendering system that, in contrast to deferred rendering, can handle a highly diverse range of materials efficiently.
- Geometry processing is completely decoupled from material rendering and lighting, resulting in fewer shader permutations and faster iteration times in game production.
- By decoupling geometry processing from shading, switching of GPU resources is significantly reduced.
- In contrast to a system where vertex attribute fetching is deferred [Burns and Hunt 2013], geometry information is only fetched once per frame in a cache-friendly, coherent manner.
- Compressed texture data does not need to be decompressed into GPU memory as with deferred rendering, thus texture memory bandwidth is significantly reduced.
- HDR textures are not a problem anymore as with deferred rendering.
- Modified geometry buffers contain useful information not available with deferred rendering:
  - Texture coordinate derivatives can be used to fix mip-mapping issues with deferred decals.
  - Vertex normals can be used to enhance screen-space ambient occlusion techniques.
  - Vertex tangents can be used for anisotropic lighting.
- The proposed rendering system does not depend on vendor-specific graphics features and is compatible with the entire range of DirectX 12-capable graphics hardware. When the supported range of dynamically indexed textures is too low, applications can still fall back to rendering common materials separately.

Cons

- Vertex attributes are much more limited in comparison to traditional rendering techniques.
- Transparent objects have to be handled separately.
- Antialiasing is still difficult to handle.

1.6 Results

To capture the results, we used two scenes from the game *Deus Ex: Mankind Divided* that we converted into a format we could load and render in an experimental framework that is based on DirectX 12. The first scene is illuminated by 1024, and the second by 256, moving spherical area lights using clustered lighting. To simulate dynamic objects for the first scene, the source of each area light is rendered as an emissive sphere. Each material uses a diffuse texture with an optional alpha mask, a normal texture, a specular texture, and a roughness texture. For diffuse lighting, a simple Lambert term is used, while specular lighting uses the GGX microfacet model; both terms were adapted for spherical area lights. Indirect lighting uses a simple constant ambient term. Frustum culling is performed on the CPU prior to GPU culling. The test machine used an NVIDIA GeForce GTX 970 graphics card and the screen resolution was set to 1920×1080.

For capturing the profiling results, the first scene was rendered from a point of view where 23,116 instances, distributed over 4073 meshes, 5,556,614 triangles, and 316 materials, are in the view frustum and processed (Figure 1.11). To be able to compare our rendering system with a reference clustered forward renderer while using GPU culling, all materials, except the emissive sphere material, use the same shading code. Tables 1.1 and 1.2 compare timings of deferred+ with a reference clustered forward renderer; Tables 1.3 and 1.4 show detailed GPU timings and efficiency of the culling system.

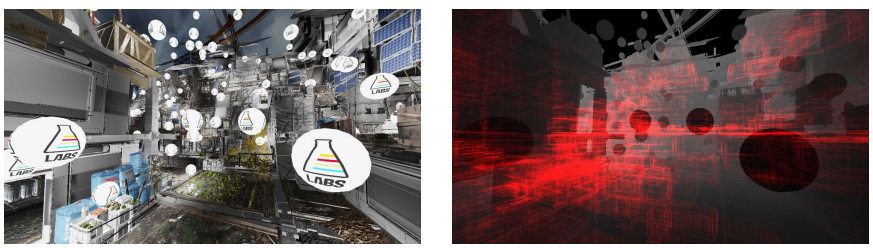

Figure 1.11. Scene from the game *Deus Ex: Mankind Divided*, culled and rendered with our system. The left image shows the final rendering result while the right image illustrates the boundaries of culled objects as red boxes.

	Deferred+ (single pass)		Deferred+ (multi-pass)		Clustered forward renderer	
	Culling on	Culling off	Culling on	Culling off	Culling on	Culling off
Frame time	5.86	6.92	6.50	8.29	9.33	12.47
Depth pre-pass	-	-	-	-	0.94	2.41
Light culling pass	0.34	0.34	0.34	0.34	0.34	0.34
Geometry pass	1.72	3.69	1.72	3.69	6.70	9.30
Material ID pass	0.13	0.13	0.13	0.13	-	-
Material bounds pass	0.02	0.03	0.13	0.51	-	-
Deferred pass	2.34	2.35	2.86	3.21	-	-

Table 1.1. Frame time and GPU times in ms for each rendering step. For deferred+ in single pass mode, all materials are rendered in a single fullscreen pass with the help of dynamically indexed textures. In multi pass mode, each material is rendered separately with the proposed depth-stencil reject method. The multi pass mode is only slightly slower than the single pass method.

	Deferred+ (single pass)		Deferred+ (multi-pass)		Clustered forward renderer	
	Culling on	Culling off	Culling on	Culling off	Culling on	Culling off
Frame time	10.00	21.43	10.77	22.78	17.96	36.15
Depth pre-pass	-	-	-	-	3.00	8.18
Light culling pass	0.34	0.34	0.34	0.34	0.34	0.34
Geometry pass	5.87	18.13	5.94	18.12	13.28	27.25
Material ID pass	0.13	0.13	0.13	0.13	-	-
Material bounds pass	0.02	0.03	0.13	0.51	-	-
Deferred pass	2.35	2.37	2.88	3.25	-	-

Table 1.2. Same as Table 1 with the exception that all mesh instances are rendered with adaptive GPU hardware tessellation, using a maximum tessellation factor of 5.

Culling overhead	0.90
Main pass	0.46
Downsample depth	0.0699
Clear visibility buffer	0.0050
Fill visibility buffer	0.3175
Generate draw list	0.0656
False negatives pass	0.44
Clear visibility buffer	0.0051
Fill visibility buffer	0.3774
Generate draw list	0.0531

Table 1.3. GPU times in ms for each step involved in the proposed culling system.

Main pass — number of visible instances	4163
Main pass — number of occluded instances	18953
False negatives pass — number of visible instances	275
Percentage of culled instances	80.80

Table 1.4. Culling statistics, acquired with the help of atomic GPU counters.

Figure 1.12 compares, in a second scene, the visual rendering results of deferred+ to that of a reference clustered forward rendering system. We used 8× anisotropic texture filtering to ensure that the compression of texture coordinate derivatives in deferred+ was working properly.

Figure 1.12. The image on the left was rendered with deferred+ and the image on the right with clustered forward shading. Quality-wise both rendering techniques are almost indistinguishable.

The captured results show that deferred+ runs faster than clustered forward shading, especially when GPU hardware tessellation is used, while producing almost equivalent results quality-wise. Under realistic game conditions with more complex materials (using more than 4 textures) and more complex lighting (different light types, shadow mapping), the performance benefit of deferred+ should be even more prominent.

1.7 Conclusion

We presented a comprehensive system for culling and rendering complex, highly dynamic scenes, which makes use of new graphics capabilities available with DirectX 12. The culling system provides high culling efficiency even for non-clustered, traditional mesh assets while having a low overhead. The rendering system outperforms a quality-wise comparable clustered forward rendering system, even more so when GPU hardware tessellation techniques are employed. It is fully compatible with conventional texture assets, doesn't depend on vendor-specific graphics features and can run on the entire range of DirectX 12 capable graphics hardware. By combining the proposed culling and rendering system it is possible to render an entire complex scene in just a few draw calls.

Acknowledgment

We would like to thank Francis Maheux for providing us with the assets for the prototype and Samuel Delmont and Uriel Doyon for their valuable input on the implementation itself.

Bibliography

Bilodeau, B. 2014. Vertex shader tricks. In *Game Developer Conference 2014 Talks*. URL: http://www.slideshare.net/DevCentralAMD/vertex-shader-tricks-bill-bilodeau.

Burns, C. A., and Hunt, W. A. 2013. The visibility buffer: A cache-friendly approach to deferred shading. *Journal of Computer Graphics Techniques 2*, 2, 55–69. URL: http://jcgt.org/published/0002/02/04/.

Haar, U., and Aaltonen, S. 2015. Gpu-driven rendering pipelines. In *Advances in Real-Time Rendering in Games, ACM SIGGRAPH Course*, ACM, New York. URL: http://advances.realtimerendering.com/s2015/index.html.

Hill, S., and Collin, D. 2011. Practical dynamic visibility for games. In *GPU Pro 2*, W. Engel, Ed. A K Peters, Natick, MA, 329–347.

Kubisch, C., and Tavenrath, M. 2014. OpenGL 4.4 scene rendering techniques. In *GPU Technology Conference 2014*. URL: http://on-demand.gputechconf.com/gtc/2014/presentations/S4379-opengl-44-scene-rendering-techniques.pdf.

M. Billeter, O. O., and Assarsson, U. 2013. Tiled forward shading. In *GPU Pro 4: Advanced Rendering Techniques*, W. Engel, Ed. A K Peters/CRC Press, Boca Raton, FL, 99–114.

McAuley, S., 2015. Rendering the world of Far Cry 4. URL: http://www.gdcvault.com/play/1022235/Rendering-the-World-of-Far.

Meyer, Q., Süssmuth, J., Sussner, G., Stamminger, M., and Greiner, G. 2010. On floating-point normal vectors. In *Proceedings of the 21st Eurographics Conference on Rendering*, Eurographics Association, Aire-la-Ville, Switzerland, EGSR'10, 1405–1409.

Olsson, O., Billeter, M., and Assarsson, U. 2012. Clustered deferred and forward shading. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, EGGH-HPG'12, 87–96.

Reed, N., 2014. Deferred texturing. Blog post. URL: http://www.reedbeta.com/blog/2014/03/25/deferred-texturing.