Chapter 20
Efficient Shading

"Never put off till to-morrow what you can do the day after to-morrow just as well."
—Mark Twain

For simple scenes—relatively little geometry, basic materials, a few lights—we can use the standard GPU pipeline to render images without concerns about maintaining frame rate. Only when one or more elements become expensive do we need to use more involved techniques to rein in costs. In the previous chapter we focused on culling triangles and meshes from processing further downstream. Here we concentrate on techniques for reducing costs when evaluating materials and lights. For many of these methods, there is an additional processing cost, with the hope being that this expense is made up by the savings obtained. Others trade off between bandwidth and computation, often shifting the bottleneck. As with all such schemes, which is best depends on your hardware, scene structure, and many other factors.

Evaluating a pixel shader for a material can be expensive. This cost can be reduced by various shader level of detail simplification techniques, as noted in Section 19.9. When there are several light sources affecting a surface, two different strategies can be used. One is to build a shader that supports multiple light sources, so that only a single pass is needed. Another is *multi-pass shading*, where we create a simple one-light pixel shader for a light and evaluate it, adding each result to the framebuffer. So, for three lights, we would draw the primitive three times, changing the light for each evaluation. This second method might be more efficient overall than a single-pass system, because each shader used is simpler and faster. If a renderer has many different types of lights, a one-pass pixel shader must include them all and test for whether each is used, making for a complex shader.

In Section 18.4.5 we discussed avoiding unnecessary pixel shader evaluations by minimizing or eliminating overdraw. If we can efficiently determine that a surface does not contribute to the final image, then we can save the time spent shading it. One technique performs a z-prepass, where the opaque geometry is rendered and only z-depths are written. The geometry is then rendered again fully shaded, and the z-buffer from the first pass culls away all fragments that are not visible. This type of

pass is an attempt to decouple the process of finding what geometry is visible from the operation of subsequently shading that geometry. This idea of separating these two processes is an important concept used throughout this chapter, and is employed by several alternative rendering schemes.

For example, a problem with using a z-prepass is that you have to render the geometry twice. This is an additional expense compared to standard rendering, and could cost more time than it saves. If the meshes are formed via tessellation, skinning, or some other involved process, the cost of this added pass can be considerable [992, 1177]. Objects with cutout alpha values need to have their texture's alpha retrieved each pass, adding to the expense, or must be ignored altogether and rendered only in the second pass, risking wasted pixel shader evaluations. For these reasons, sometimes only large occluders (in screen or world space) are drawn in this initial pass. Performing a full prepass may also be needed by other screen-space effects, such as ambient occlusion or reflection [1393]. Some of the acceleration techniques presented in this chapter require an accurate z-prepass that is then used to help cull lists of lights.

Even with no overdraw, a significant expense can arise from a large number of dynamic lights evaluated for a visible surface. Say you have 50 light sources in a scene. A multi-pass system can render the scene successfully, but at the cost of 50 vertex and shader passes per object. One technique to reduce costs is to limit the effect of each local light to a sphere of some radius, cone of some height, or other limited shape [668, 669, 1762, 1809]. The assumption is that each light's contribution becomes insignificant past a certain distance. For the rest of this chapter, we will refer to lights' volumes as spheres, with the understanding that other shapes can be used. Often the light's intensity is used as the sole factor determining its radius. Karis [860] discusses how the presence of glossy specular materials will increase this radius, since such surfaces are more noticeably affected by lights. For extremely smooth surfaces this distance may go to infinity, such that environment maps or other techniques may need to be used instead.

A simple preprocess is to create for each mesh a list of lights that affects it. We can think of this process as performing collision detection between a mesh and the lights, finding those that may overlap [992]. When shading the mesh, we use this list of lights, thus reducing the number of lights applied. There are problems with this type of approach. If objects or lights move, these changes affect the composition of the lists. For performance, geometry sharing the same material is often consolidated into larger meshes (Section 18.4.2), which could cause a single mesh to have some or all of the lights in a scene in its list [1327, 1330]. That said, meshes can be consolidated and then split spatially to provide shorter lists [1393].

Another approach is to bake static lights into world-space data structures. For example, in the lighting system for *Just Cause 2*, a world-space top-down grid stores light information for a scene. A grid cell represents a 4 meter $\times$ 4 meter area. Each cell is stored as a texel in an RGBα texture, thus holding a list of up to four lights. When a pixel is rendered, the list in its area is retrieved and the relevant lights are applied [1379]. A drawback is that there is a fixed storage limit to the number of

Figure 20.1. A complicated lighting situation. Note that the small light on the shoulder and every bright dot on the building structure are light sources. The lights in the far distance in the upper right are light sources, which are rendered as point sprites at that distance. *(Image from "Just Cause 3," courtesy of Avalanche Studios [1387].)*

lights affecting a given area. While potentially useful for carefully designed outdoor scenes, buildings with a number of stories can quickly overwhelm this storage scheme.

Our goal is to handle dynamic meshes and lights in an efficient way. Also important is predictable performance, where a small change in the view or scene does not cause a large change in the cost of rendering it. Some levels in *DOOM* (2016) have 300 visible lights [1682]; some scenes in *Ashes of the Singularity* have 10,000. See Figure 20.1 and Figure 20.15 on page 913. In some renderers a large number of particles can each be treated as small light sources. Other techniques use light probes (Section 11.5.4) to illuminate nearby surfaces, which can be thought of as short-range light sources.

20.1 Deferred Shading

So far throughout this book we have described *forward shading*, where each triangle is sent down the pipeline and, at the end of its travels, the image on the screen is updated with its shaded values. The idea behind *deferred shading* is to perform all visibility testing and surface property evaluations before performing any material lighting computations. The concept was first introduced in a hardware architecture in 1988 [339], later included as a part of the experimental PixelFlow system [1235], and used as an offline software solution to help produce non-photoreal styles via image processing [1528]. Calver's extensive article [222] in mid-2003 lays out the basic

ideas of using deferred shading on the GPU. Hargreaves and Harris [668, 669] and Thibieroz [1762] promoted its use the following year, at a time when the ability to write to multiple render targets was becoming more widely available.

In forward shading we perform a single pass using a shader and a mesh representing the object to compute the final image. The pass fetches material properties—constants, interpolated parameters, or values in textures—then applies a set of lights to these values. The z-prepass method for forward rendering can be seen as a mild decoupling of geometry rendering and shading, in that a first geometry pass aims at only determining visibility, while all shading work, including material parameter retrieval, is deferred to a second geometry pass performed to shade all visible pixels. For interactive rendering, deferred shading specifically means that all material parameters associated with the visible objects are generated and stored by an initial geometry pass, then lights are applied to these stored surface values using a post-process. Values saved in this first pass include the position (stored as z-depth), normal, texture coordinates, and various material parameters. This pass establishes all geometry and material information for the pixel, so the objects are no longer needed, i.e., the contribution of the models' geometry has been fully decoupled from lighting computations. Note that overdraw can happen in this initial pass, the difference being that the shader's execution is considerably less—transferring values to buffers—than that of evaluating the effect of a set of lights on the material. There is also less of the additional cost found in forward shading, where sometimes not all the pixels in a 2×2 quad are inside a triangle's boundaries but all must be fully shaded [1393] (Section 23.8). This sounds like a minor effect, but imagine a mesh where each triangle covers a single pixel. Four fully shaded samples will be generated and three of these discarded with forward shading. Using deferred shading, each shader invocation is less expensive, so discarded samples have a considerably lower impact.

The buffers used to store surface properties are commonly called *G-buffers* [1528], short for "geometric buffers." Such buffers are also occasionally called *deep buffers*, though this term can also mean a buffer storing multiple surfaces (fragments) per pixel, so we avoid it here. Figure 20.2 shows the typical contents of some G-buffers. A G-buffer can store anything a programmer wants it to contain, i.e., whatever is needed to complete the required subsequent lighting computations. Each G-buffer is a separate render target. Typically three to five render targets are used as G-buffers, but systems have gone as high as eight [134]. Having more targets uses more bandwidth, which increases the chance that this buffer is the bottleneck.

After the pass creating the G-buffers, a separate process is used to compute the effect of illumination. One method is to apply each light one by one, using the G-buffers to compute its effect. For each light we draw a screen-filling quadrilateral (Section 12.1) and access the G-buffers as textures [222, 1762]. At each pixel we can determine the location of the closest surface and whether it is in range of the light. If it is, we compute the effect of the light and place the result in an output buffer. We do this for each light in turn, adding its contribution via blending. At the end, we have all lights' contributions applied.

Figure 20.2. Geometric buffers for deferred shading, in some cases converted to colors for visualization. Left column, top to bottom: depth map, normal buffer, roughness buffer, and sunlight occlusion. Right column: texture color (a.k.a. albedo texture), light intensity, specular intensity, and near-final image (without motion blur). *(Images from "Killzone 2," courtesy of Guerrilla BV [1809].)*

This process is about the most inefficient way to use G-buffers, since every stored pixel is accessed for every light, similar to how basic forward rendering applies all lights to all surface fragments. Such an approach can end up being slower than forward shading, due to the extra cost of writing and reading the G-buffers [471]. As a start

on improving performance, we could determine the screen bounds of a light volume (a sphere) and use them to draw a screen-space quadrilateral that covers a smaller part of the image [222, 1420, 1766]. In this way, pixel processing is reduced, often significantly. Drawing an ellipse representing the sphere can further trim pixel processing that is outside the light's volume [1122]. We can also use the third screen dimension, z-depth. By drawing a rough sphere mesh encompassing the volume, we can trim the sphere's area of effect further still [222]. For example, if the sphere is hidden by the depth buffer, the light's volume is behind the closest surface and so has no effect. To generalize, if a sphere's minimum and maximum depths at a pixel do not overlap the closest surface, the light cannot affect this pixel. Hargreaves [668] and Valient [1809] discuss various options and caveats for efficiently and correctly determining this overlap, along with other optimizations. We will see this idea of testing for depth overlap between the surface and the light used in several of the algorithms ahead. Which is most efficient depends on the circumstances.

For traditional forward rendering, the vertex and pixel shader programs retrieve each light's and material's parameters and compute the effect of one on the other. Forward shading needs either one complex vertex and pixel shader that covers all possible combinations of materials and lights, or shorter, specialized shaders that handle specific combinations. Long shaders with dynamic branches often run considerably more slowly [414], so a large number of smaller shaders can be more efficient, but also require more work to generate and manage. Since all shading functions are done in a single pass with forward shading, it is more likely that the shader will need to change when the next object is rendered, leading to inefficiency from swapping shaders (Section 18.4.2).

The deferred shading method of rendering allows a strong separation between lighting and material definition. Each shader is focused on parameter extraction or lighting, but not both. Shorter shaders run faster, both due to length and the ability to optimize them. The number of registers used in a shader determines occupancy (Section 23.3), a key factor in how many shader instances can be run in parallel. This decoupling of lighting and material also simplifies shader system management. For example, this split makes experimentation easy, as only one new shader needs to be added to the system for a new light or material type, instead of one for each combination [222, 927]. This is possible since material evaluations are done in the first pass, and lighting is then applied to this stored set of surface parameters in the second pass.

For single-pass forward rendering, all shadow maps usually must be available at the same time, since all lights are evaluated at once. With each light handled fully in a single pass, deferred shading permits having only one shadow map in memory at a time [1809]. However, this advantage disappears with the more complex light assignment schemes we cover later, as lights are evaluated in groups [1332, 1387].

Basic deferred shading supports just a single material shader with a fixed set of parameters, which constrains what material models can be portrayed. One way to support different material descriptions is to store a material ID or mask per pixel in

	R8	G8	B8	A8
RT0	world normal (RGB10)			GI
RT1	base color (sRGB8)			config (A8)
RT2	metalness (R8)	glossiness (G8)	cavity (B8)	aliased value (A8)
RT3	velocity.xy (RGB8)			velocity.z (A8)

Figure 20.3. An example of a possible G-buffer layout, used in *Rainbow Six Siege*. In addition to depth and stencil buffers, four render targets (RTs) are used as well. As can be seen, anything can be put into these buffers. The "GI" field in RT0 is "GI normal bias (A2)." *(Illustration after El Mansouri [415].)*

some given field. The shader can then perform different computations based on the G-buffer contents. This approach could also modify what is stored in the G-buffers, based on this ID or mask value [414, 667, 992, 1064]. For example, one material might use 32 bits to store a second layer color and blend factor in a G-buffer, while another may use these same bits to store two tangent vectors that it requires. These schemes entail using more complex shaders, which can have performance implications.

Basic deferred shading has some other drawbacks. G-buffer video memory requirements can be significant, as can the related bandwidth costs in repeatedly accessing these buffers [856, 927, 1766]. We can mitigate these costs by storing lower-precision values or compressing the data [1680, 1809]. An example is shown in Figure 20.3. In Section 16.6 we discussed compression of world-space data for meshes. G-buffers can contain values that are in world-space or screen-space coordinates, depending on the needs of the rendering engine. Pesce [1394] discusses the trade-offs in compressing screen-space versus world-space normals for G-buffers and provides pointers to related resources. A world-space octahedral mapping for normals is a common solution, due to its high precision and quick encoding and decoding times.

Two important technical limitations of deferred shading involve transparency and antialiasing. Transparency is not supported in a basic deferred shading system, since we can store only one surface per pixel. One solution is to use forward rendering for transparent objects after the opaque surfaces are rendered with deferred shading. For early deferred systems this meant that all lights in a scene had to be applied to each transparent object, a costly process, or other simplifications had to be performed. As we will explore in the sections ahead, improved GPU capabilities have led to the development of methods that cull lights for both deferred and forward shading. While it is possible to now store lists of transparent surfaces for pixels [1575] and use a pure deferred approach, the norm is to mix deferred and forward shading as desired for transparency and other effects [1680].

An advantage of forward methods is that antialiasing schemes such as MSAA are easily supported. Forward techniques need to store only N depth and color samples per pixel for $N\times$ MSAA. Deferred shading could store all N samples per element in the G-buffers to perform antialiasing, but the increases in memory cost, fill rate, and computation make this approach expensive [1420]. To overcome this limitation,

Shishkovtsov [1631] uses an edge detection method for approximating edge coverage computations. Other morphological post-processing methods for antialiasing (Section 5.4.2) can also be used [1387], as well as temporal antialiasing. Several deferred MSAA methods avoid computing the shade for every sample by detecting which pixels or tiles have edges in them [43, 990, 1064, 1299, 1764]. Only those with edges need to have multiple samples evaluated. Sousa [1681] builds on this type of approach, using stenciling to identify pixels with multiple samples that need more complex processing. Pettineo [1407] describes a newer way to track such pixels, using a compute shader to move edge pixels to a list in thread group memory for efficient stream processing.

Antialiasing research by Crassin et al. [309] focuses on high-quality results and summarizes other research in this area. Their technique performs a depth and normal geometry prepass and groups similar subsamples together. They then generate G-buffers and perform a statistical analysis of the best value to use for each group of subsamples. These depth-bounds values are then used to shade each group and the results are blended together. While as of this writing such processing at interactive rates is impractical for most applications, this approach gives a sense of the amount of computational power that can and will be brought to bear on improving image quality.

Even with these limitations, deferred shading is a practical rendering method used in commercial programs. It naturally separates geometry from shading, and lighting from materials, meaning that each element can be optimized on its own. One area of particular interest is decal rendering, which has implications for any rendering pipeline.

20.2 Decal Rendering

A decal is some design element, such as a picture or other texture, applied on top of a surface. Decals are often seen in video games in such forms as tire marks, bullet holes, or player tags sprayed onto surfaces. Decals are used in other applications for applying logos, annotations, or other content. For terrain systems or cities, for example, decals can allow artists to avoid obvious repetition by layering on detailed textures, or by recombining various patterns in different ways.

A decal can blend with the underlying material in a variety of ways. It might modify the underlying color but not the bump map, like a tattoo. Alternately, it might replace just the bump mapping, such as an embossed logo does. It could define a different material entirely, for example, placing a sticker on a car window. Multiple decals might be applied to the same geometry, such as footprints on a path. A single decal might span multiple models, such as graffiti on a subway car's surfaces. These variations have implications for how forward and deferred shading systems store and process decals.

To begin, the decal must be mapped to the surface, like any other texture. Since multiple texture coordinates can be stored at each vertex, it is possible to bind a few decals to a single surface. This approach is limited, since the number of values that

can be saved per vertex is relatively low. Each decal needs its own set of texture coordinates. A large number of small decals applied to a surface would mean saving these texture coordinates at every vertex, even though each decal affects only a few triangles in the mesh.

To render decals attached to a mesh, one approach is to have the pixel shader sample every decal and blend one atop the next. This complicates the shader, and if the number of decals varies over time, frequent recompilation or other measures may be required. Another approach that keeps the shader independent from the decal system is to render the mesh again for each decal, layering and blending each pass over the previous one. If a decal spans just a few triangles, a separate, shorter index buffer can be created to render just this decal's sub-mesh. One other decal method is to modify the material's texture. If used on just one mesh, as in a terrain system, modifying this texture provides a simple "set it and forget it" solution [447]. If the material texture is used on a few objects, we need to create a new texture with the material and decal composited together. This baked solution avoids shader complexity and wasted overdraw, but at the cost of texture management and memory use [893, 1393]. Rendering the decals separately is the norm, as different resolutions can then be applied to the same surface, and the base texture can be reused and repeated without needing additional modified copies in memory.

These solutions can be reasonable for a computer-aided design package, where the user may add a single logo and little else. They are also used for decals applied to animated models, where the decal needs to be projected before the deformation so it stretches as the object does. However, such techniques become inefficient and cumbersome for more than a few decals.

A popular solution for static or rigid objects is to treat the decal as a texture orthographically projected through a limited volume [447, 893, 936, 1391, 1920]. An oriented box is placed in the scene, with the decal projected from one of the box faces to its opposite face, like a film projector. See Figure 20.4. The faces of the box are rasterized, as a way to drive the pixel shader's execution. Any geometry found inside this volume has the decal applied over its material. This is done by converting the surface's depth and screen position into a location in the volume, which then gives a (u, v) texture coordinate for the decal. Alternately, the decal could be a true volume texture [888, 1380]. Decals can affect only certain objects in the volume by assigning IDs [900], assigning a stencil bit [1778], or relying on the rendering order. They are also often faded or clamped to the angle of the surface and the projection direction, to avoid having a decal stretch or distort wherever the surface becomes more edge-on [893].

Deferred shading excels at rendering such decals. Instead of needing to illuminate and shade each decal, as with standard forward shading, the decal's effect can be applied to the G-buffers. For example, if a decal of a tire's tread mark replaces the shading normals on a surface, these changes are made directly to the appropriate G-buffer. Each pixel is later shaded by lights with only the data found in the G-buffers, so avoiding the shading overdraw that occurs with forward shading [1680]. Since

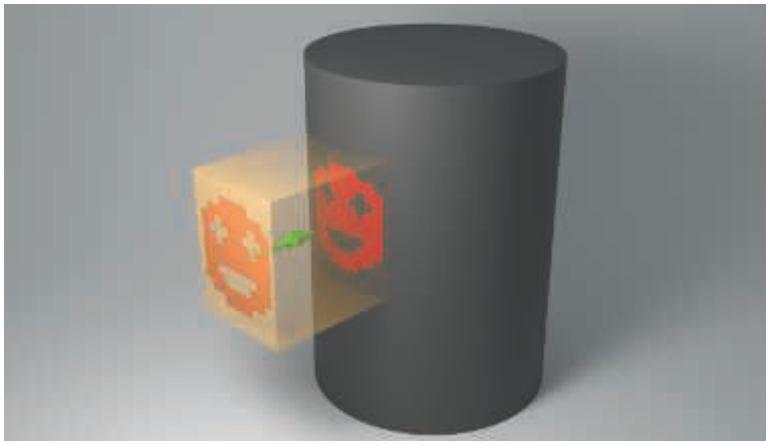

Figure 20.4. A box defines a decal projection, and a surface inside that box has the decal applied to it. The box is shown with exaggerated thickness to display the projector and its effect. In practice the box is made as thin and tight to the surface as possible, to minimize the number of pixels that are tested during decal application.

the decal's effect can be captured entirely by G-buffer storage, the decal is then not needed during shading. This integration also avoids a problem with multi-pass forward shading, that the surface parameters of one pass may need to affect the lighting or shading of another pass [1380]. This simplicity was a major factor in the decision to switch from forward to deferred shading for the Frostbite 2 engine, for example [43]. Decals can be thought of as the same as lights, in that both are applied by rendering a volume of space to determine its effect on the surfaces enclosed. As we will see in Section 20.4, by using this fact, a modified form of forward shading can capitalize on similar efficiencies, along with other advantages.

Lagarde and de Rousiers [960] describe several problems with decals in a deferred setting. Blending is limited to what operations are available during the merge stage in the pipeline [1680]. If the material and decal both have normal maps, achieving a properly blended result can be difficult, and harder still if some bump texture filtering technique is in use [106, 888]. Black or white fringing artifacts can occur, as described in Section 6.5. Techniques such as signed distance fields can be used to sharply divide such materials [263, 580], though doing so can cause aliasing problems. Another concern is fringing along silhouette edges of decals, caused by gradient errors due to using screen-space information projected back into world space. One solution is to restrict or ignore mipmapping for such decals; more elaborate solutions are discussed by Wronski [1920].

Decals can be used for dynamic elements, such as skid marks or bullet holes, but are also useful for giving different locations some variation. Figure 20.5 shows a scene with decals applied to building walls and elsewhere. The wall textures can be reused, while the decals provide customized details that give each building a unique character.

Figure 20.5. In the top image the areas where color and bump decals are overlaid are shown with checkerboards. The middle shows the building with no decals applied. The bottom image shows the scene with about 200 decals applied. *(Images courtesy of IO Interactive.)*

20.3 Tiled Shading

In basic deferred shading, each light is evaluated separately and the result is added to the output buffer. This was a feature for early GPUs, where evaluating more than a few lights could be impossible, due to limitations on shader complexity. Deferred shading could handle any number of lights, at the cost of accessing the G-buffers each time. With hundreds or thousands of lights, basic deferred shading becomes expensive, since all lights need to be processed for each overlapped pixel, and each light evaluated at a pixel involves a separate shader invocation. Evaluating several lights in a single shader invocation is more efficient. In the sections that follow, we discuss several algorithms for rapidly processing large numbers of lights at interactive rates, for both deferred and forward shading.

Various hybrid G-buffer systems have been developed over the years, balancing between material and light storage. For example, imagine a simple shading model with diffuse and specular terms, where the material's texture affects only the diffuse term. Instead of retrieving the texture's color from a G-buffer for each light, we could first compute each light's diffuse and specular terms separately and store these results. These accumulated terms are added together in light-based G-buffers, sometimes called L-buffers. At the end, we retrieve the texture's color once, multiply it by the diffuse term, and then add in the specular. The texture's effect is factored out of the equation, as it is used only a single time for all lights. In this way, fewer G-buffer data points are accessed per light, saving on bandwidth. A typical storage scheme is to accumulate the diffuse color and the specular intensity, meaning four values can be output via additive blending to a single buffer. Engel [431, 432] discusses several of these early *deferred lighting* techniques, also known as *pre-lighting* or *light prepass* methods. Kaplanyan [856] compares different approaches, aiming to minimize G-buffer storage and access. Thibieroz [1766] also stresses shallower G-buffers, contrasting several algorithms' pros and cons. Kircher [900] describes using lower-resolution G-buffers and L-buffers for lighting, which are upsampled and bilateral-filtered during a final forward shading pass. This approach works well for some materials, but can cause artifacts if the lighting's effect changes rapidly, e.g., a roughness or normal map is applied to a reflective surface. Sousa et al. [1681] use the idea of subsampling alongside Y'CbCr color encoding of the albedo texture to help reduce storage costs. Albedo affects the diffuse component, which is less prone to high-frequency changes.

There are many more such schemes [892, 1011, 1351, 1747], each varying such elements as which components are stored and factored, what passes are performed, and how shadows, transparency, antialiasing, and other phenomena are rendered. A major goal of all these is the same—the efficient rendering of light sources—and such techniques are still in use today [539]. One limitation of some schemes is that they can require even more restricted material and lighting models [1332]. For example, Shulz [1589] notes that moving to a physically based material model meant that the specular reflectance then needed to be stored to compute the Fresnel term from the lighting. This increase in the light prepass requirements helped push his group to move from a light prepass to a fully deferred shading system.

Accessing even a small number of G-buffers per light can have significant bandwidth costs. Faster still would be to evaluate only the lights that affect each pixel, in a single pass. Zioma [1973] was one of the first to explore creating lists of lights for forward shading. In his scheme, light volumes are rendered and the light's relative location, color, and attenuation factor are stored for each pixel overlapped. Depth peeling is used to handle storing the information for light sources that overlap the same pixels. The scene's geometry is then rendered, using the stored light representation. While viable, this scheme is limited by how many lights can overlap any pixel. Trebilco [1785] takes the idea of creating light lists per pixel further. He performs a z-prepass to avoid overdraw and to cull hidden light sources. The light volumes are rendered and stored as ID values per pixel, which are then accessed during the forward rendering pass. He gives several methods for storing multiple lights in a single buffer, including a bit-shifting and blending technique that allows four lights to be stored without the need for multiple depth-peeling passes.

Tiled shading was first presented by Balestra and Engstad [97] in 2008 for the game *Uncharted: Drake's Fortune*, soon followed by presentations about its use in the Frostbite engine [42] and PhyreEngine [1727], among others. The core idea of tiled shading is to assign light sources to tiles of pixels, so limiting both the number of lights that need to be evaluated at each surface, and the amount of work and storage required. These per-tile light lists are then accessed in a single shader invocation, instead of deferred shading's method of calling a shader for each light [990].

A tile for light classification is a square set of pixels on the screen, for example, 32×32 pixels in size. Note that there are other ways tiling the screen is used for interactive rendering; for instance, mobile processors render an image by processing tiles [145], and GPU architectures use screen tiles for a variety of tasks (Chapter 23). Here the tiles are a construct chosen by the developer and often have relatively little to do with the underlying hardware. A tiled rendering of the light volumes is something like a low-resolution rendering of the scene, a task that can be performed on the CPU or in, say, a compute shader on the GPU [42, 43, 139, 140, 1589].

Lights that potentially affect a tile are recorded in a list. When rendering is performed, a pixel shader in a given tile uses the tile's corresponding list of lights to shade the surface. This is illustrated on the left side of Figure 20.6. As can be seen, not all lights overlap with every tile. The screen-space bounds of the tile form an asymmetrical frustum, which is used to determine overlap. Each light's spherical volume of effect can quickly be tested on the CPU or in a compute shader for overlap with each tile's frustum. Only if there is an overlap do we need to process that light further for the pixels in the tile. By storing light lists per tile instead of per pixel, we err on the side of being conservative—a light's volume may not overlap the whole tile—in exchange for much reduced processing, storage, and bandwidth costs [1332].

To determine whether a light overlaps with a tile, we can use frustum testing against a sphere, which is described in Section 22.14. The test there assumes a large, wide frustum and relatively small spheres. However, since the frustum here originates from a screen-space tile, it is often long, thin, and asymmetrical. This decreases the

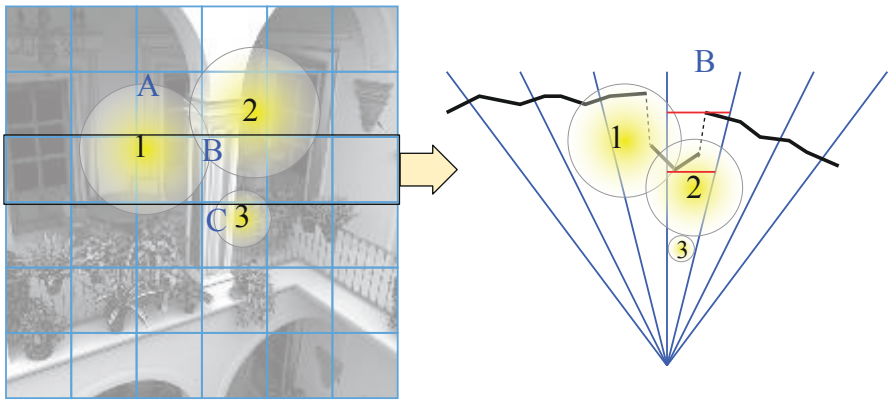

Figure 20.6. Illustration of tiling. Left: the screen has been divided into 6×6 tiles, and three lights sources, 1–3, are lighting this scene. Looking at tiles A–C, we see that tile A is potentially affected by lights 1 and 2, tile B by lights 1–3, and tile C by light 3. Right: the black outlined row of tiles on the left is visualized as seen from above. For tile B, the depth bounds are indicated by red lines. On the screen, tile B appears to be overlapped by all lights, but only lights 1 and 2 overlap with the depth bounds as well.

efficiency of the cull, since the number of reported intersections can increase (i.e., false positives). See the left part of Figure 20.7. Instead one can add a sphere/box test (Section 22.13.2) after testing against the planes of the frustum [1701, 1768], which is illustrated on the right in Figure 20.7. Mara and McGuire [1122] run through alternative tests for a projected sphere, including their own GPU-efficient version. Zhdan [1968] notes that this approach does not work well for spotlights, and discusses optimization techniques using hierarchical culling, rasterization, and proxy geometry.

This light classification process can be used with deferred shading or forward rendering, and is described in detail by Olsson and Assarsson [1327]. For *tiled deferred shading*, the G-buffers are established as usual, the volume of each light is recorded in the tiles it overlaps, and then these lists are applied to the G-buffers to compute the final result. In basic deferred shading each light is applied by rendering a proxy object such as a quadrilateral to force the pixel shader to be evaluated for that light. With tiled shading, a compute shader or a quad rendered for the screen or per tile is used to drive shader evaluation for each pixel. When a fragment is then evaluated, all lights in the list for that tile are applied. Applying lists of lights has several advantages, including:

- For each pixel the G-buffers are read at most one time total, instead of one time per overlapping light.
- The output image buffer is written to only one time, instead of accumulating the results of each light.

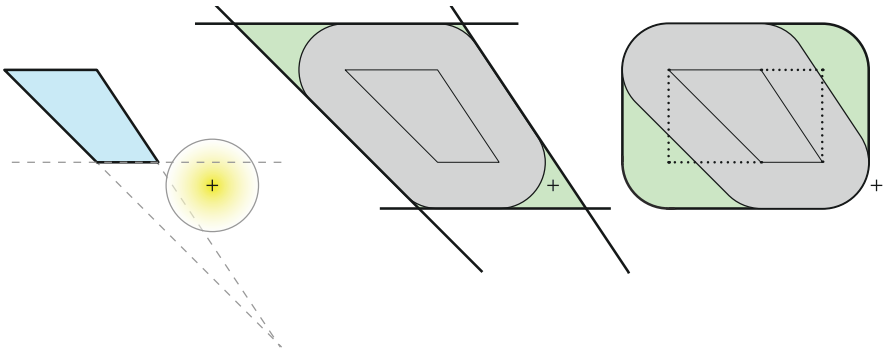

Figure 20.7. Left: with a naive sphere/frustum test, this circle would be reported as intersecting since it overlaps with the bottom and right planes of the frustum. Middle: an illustration of the test to the left, where the frustum has grown and the origin (plus sign) of the circle is tested against just the thick black planes. False intersections are reported in the green regions. Right: a box, shown with dotted lines, is placed around the frustum, and a sphere/box test is added after the plane test in the middle, forming the thick-outlined shape shown. Note how this test would produce other false intersections in its green areas, but with both tests applied these areas are reduced. Since the origin of the sphere is outside the shape, the sphere is correctly reported as not overlapping the frustum.

- Shader code can factor out any common terms in the rendering equation and compute these once, instead of per light [990].
- Each fragment in a tile evaluates the same list of lights, ensuring coherent execution of the GPU warps.
- After all opaque objects are rendered, transparent objects can be handled using forward shading, using the same light lists.
- Since the effects of all lights are computed in a single pass, framebuffer precision can be low, if desired.

This last item, framebuffer precision, can be important in a traditional deferred shading engine [1680]. Each light is applied in a separate pass, so the final result can suffer from banding and other artifacts if the results are accumulated in a framebuffer with only 8 bits per color channel. That said, being able to use a lower precision is not relevant for many modern rendering systems, since these need higher-precision output for performing tone mapping and other operations.

Tiled light classification can also be used with forward rendering. This type of system is called *tiled forward shading* [144, 1327] or *forward+* [665, 667]. First a z-prepass of the geometry is performed, both to avoid overdraw in the final pass and to permit further light culling. A compute shader classifies the lights by tiles. A second geometry pass then performs forward shading, with each shader accessing the light lists based on the fragment's screen-space location.

Tiled forward shading has been used in such games as *The Order: 1886* [1267, 1405]. Pettineo [1401] provides an open-source test suite to compare implementations of deferred [990] and forward classification of tiled shading. For antialiasing each sample was stored when using deferred shading. Results were mixed, with each scheme outperforming the other under various test conditions. With no antialiasing, deferred tended to win out on many GPUs as the number of lights increased up to 1024, and forward did better as the antialiasing level was increased. Stewart and Thomas [1700] analyze one GPU model with a wider range of tests, finding similar results.

The z-prepass can also be used for another purpose, culling lights by depth. The idea is shown on the right in Figure 20.6. The first step is finding the minimum and maximum z-depths of objects in a tile, $z_{\min}$ and $z_{\max}$. Each of these is determined by performing a *reduce* operation, in which a shader is applied to the tile's data and the $z_{\min}$ and $z_{\max}$ values are computed by sampling in one or more passes [43, 1701, 1768]. As an example, Harada et al. [667] use a compute shader and unordered access views to efficiently perform frustum culling and reduction of the tiles. These values can then be used to quickly cull any lights that do not overlap this range in the tile. Empty tiles, e.g., where only the sky is visible, can also be ignored [1877]. The type of scene and the application affect whether it is worthwhile to compute and use the minimums, maximums, or both [144]. Culling in this way can also be applied to tiled deferred shading, since the depths are present in the G-buffers.

Since the depth bounds are found from opaque surfaces, transparency must be considered separately. To handle transparent surfaces, Neubelt and Pettineo [1267] render an additional set of passes to create per-tile lights, used to light and shade only transparent surfaces. First, the transparent surfaces are rendered on top of the opaque geometry's z-prepass buffer. The $z_{\min}$ of the transparent surfaces are kept, while the $z_{\max}$ of the opaque surfaces are used to cap the far end of the frustum. The second pass performs a separate light classification pass, where new per-tile light lists are generated. The third pass sends only the transparent surfaces through the renderer, in a similar fashion as tiled forward shading. All such surfaces are shaded and lit with the new light lists.

For scenes with a large number of lights, a range of valid z-values is critical in culling most of these out from further processing. However, this optimization provides little benefit for a common case, depth discontinuities. Say a tile contains a nearby character framed against a distant mountain. The z range between the two is enormous, so is mostly useless for culling lights. This depth range problem can affect a large percentage of a scene, as illustrated in Figure 20.8. This example is not an extreme case. A scene in a forest or with tall grass or other vegetation can contain discontinuities in a higher percentage of tiles [1387].

One solution is to make a single split, halfway between $z_{\min}$ and $z_{\max}$. Called *bimodal clusters* [992] or *HalfZ* [1701, 1768], this test categorizes an intersected light as overlapping the closer, farther, or full range, compared to the midpoint. Doing so directly attacks the case of two objects in a tile, one near and one far. It does not address all concerns, e.g., the case of a light volume overlapping neither object, or

Figure 20.8. A visualization of tiles where large depth discontinuities exist. *(Image from "Just Cause 3," courtesy of Avalanche Studios [1387].)*

more than two objects overlapping at different depths. Nonetheless, it can provide a noticeable reduction in lighting calculations overall.

Harada et al. [666, 667] present a more elaborate algorithm called *2.5D culling*, where each tile's depth range, z_{min} and z_{max}, is split into n cells along the depth direction. This process is illustrated in Figure 20.9. A geometry bitmask of n bits is created, and each bit is set to 1 where there is geometry. For efficiency, they use $n = 32$. Iteration over all lights follows, and a light bitmask is created for every light that overlaps the tile frustum. The light bitmask indicates in which cells the light is located. The geometry bitmask is `AND`:ed with the light mask. If the result is zero, then that light does not affect any geometry in that tile. This is shown on the right in Figure 20.9. Otherwise, the light is appended to the tile's light list. For one GPU architecture, Stewart and Thomas [1700] found that when the number of lights rose to be over 512, HalfZ began outperforming basic tiled deferred, and when the number rose beyond 2300, 2.5D culling began to dominate, though not significantly so.

Mikkelsen [1210] prunes the light lists further by using the pixel locations of the opaque objects. A list for each 16×16 pixel tile is generated with a screen-space bounding rectangle for each light, along with the z_{min} and z_{max} geometry bounds for culling. This list is then culled further by having each of 64 compute-shader threads compare four pixels in the tile against each light. If none of the pixels' world-space locations in a tile are found to be inside a light's volume, the light is culled from the list. The resulting set of lights can be quite accurate, since only those lights guaranteed to affect at least one pixel are saved. Mikkelsen found that, for his scenes, further culling procedures using the z-axis decreased overall performance.

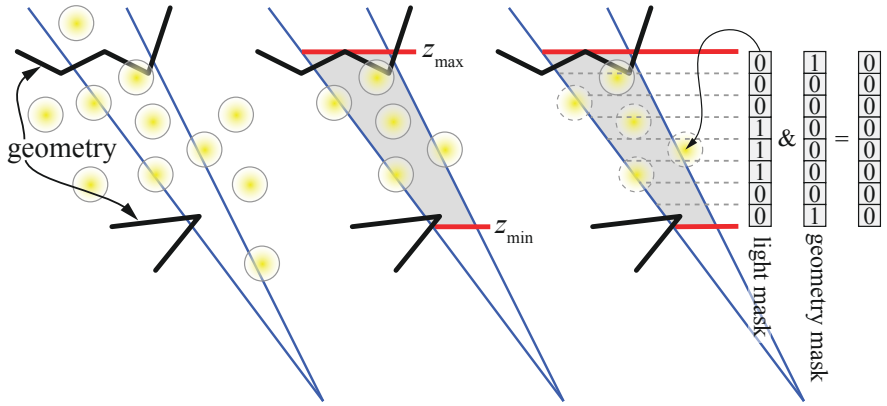

Figure 20.9. Left: a frustum for a tile in blue, some geometry in black, and set of circular yellow light sources. Middle: with tiled culling the $z_{\min}$ and $z_{\max}$ values in red are used to cull light sources that do not overlap with the gray area. Right: with clustered culling, the region between $z_{\min}$ and $z_{\max}$ is split into n cells, where $n = 8$ in this example. A geometry bitmask (`10000001`) is computed using the depths of the pixels, and a light bitmask is computed for each light. If the bitwise `AND` between these is `0`, then that light is not considered further for that tile. The topmost light has `11000000` and so is the only light that will be processed for lighting computations, since `11000000 AND 10000001` gives `10000000`, which is nonzero.

With lights placed into lists and evaluated as a set, shader complexity for a deferred system can become quite complex. A single shader must be able to handle all materials and all light types. Tiles can help reduce this complexity. The idea is to store a bitmask in every pixel, with each bit associated with a shader feature the material uses in that pixel. For each tile, these bitmasks are `OR`:ed together to determine the smallest number of features used in that tile. The bitmasks can also be `AND`:ed together to find features that are used by all pixels, meaning that the shader does not need an "if" test to check whether to execute this code. A shader fulfilling these requirements is then used for all pixels in the tile [273, 414, 1877]. This shader specialization is important not only because less instructions need to be executed, but also because the resulting shaders might achieve higher occupancy (Section 23.3), as otherwise the shader has to allocate registers for the worst-case code path. Attributes other than materials and lights can be tracked and used to affect the shader. For example, for the game *Split/Second*, Knight et al. [911] classify 4×4 tiles by whether they are fully or partially in shadow, if they contain polygon edges that need antialiasing, and other tests.

20.4 Clustered Shading

Tiled light classification uses the two-dimensional spatial extents of a tile and, optionally, the depth bounds of the geometry. *Clustered shading* divides the view frustum

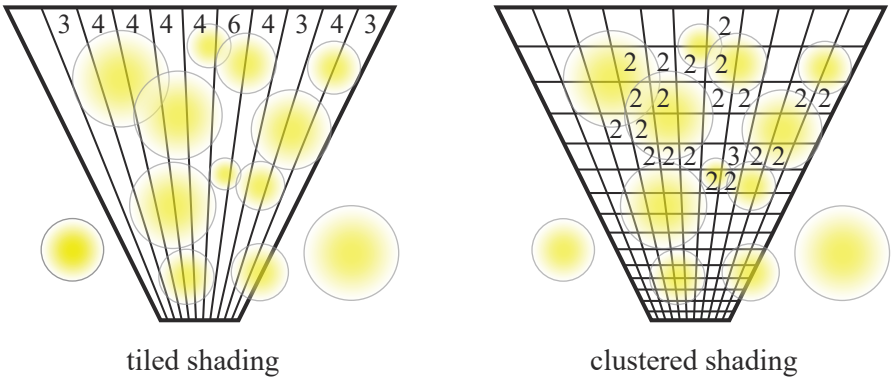

Figure 20.10. Tiled and clustered shading, shown in two dimensions. The view frustum is subdivided and a scene's light volumes are categorized by which regions they overlap. Tiled shading subdivides in screen space, while clustered also divides by z-depth slices. Each volume contains a list of lights; values are shown for lists of length two or more. If $z_{\min}$ and $z_{\max}$ are not computed for tiled shading from the scene's geometry (not shown), light lists could contain large numbers of unneeded lights. Clustered shading does not need rendered geometry to cull its lists, though such a pass can help. *(Figure after Persson [1387].)*

into a set of three-dimensional cells, called clusters. Unlike the z-depth approaches for tiled shading, this subdivision is performed on the whole view frustum, independent of the geometry in the scene. The resulting algorithms have less performance variability with camera position [1328], and behave better when a tile contains depth discontinuities [1387]. Clustered shading can be applied to both forward and deferred shading systems.

Due to perspective, a tile's cross-section area increases with distance from the camera. Uniform subdivision schemes will create squashed or long and thin voxels for the tile's frustum, which are not optimal. To compensate, Olsson et al. [1328, 1329] cluster geometry exponentially in view space, without any dependency on geometry's $z_{\min}$ and $z_{\max}$, to make clusters be more cubical. As an example, the developers for *Just Cause 3* use 64×64 pixel tiles with 16 depth slices, and have experimented with larger resolutions along each axis, as well as using a fixed number of screen tiles regardless of resolution [1387]. The Unreal Engine uses the same size tiles and typically 32 depth slices [38]. See Figure 20.10.

Lights are categorized by the clusters they overlap, and lists are formed. By not depending on the scene geometry's z-depths, the clusters can be computed from just the view and the set of lights [1387]. Each surface, opaque or transparent, then uses its location to retrieve the relevant light list. Clustering provides an efficient, unified lighting solution that works for all objects in the scene, including transparent and volumetric ones.

Like tiled approaches, algorithms using clustering can be combined with forward or deferred shading. For example, *Forza Horizon 2* computes its clusters on the GPU,

and then uses forward shading because this provides MSAA support without any additional work [344, 1002, 1387]. While overdraw is possible when forward shading in a single pass, other methods, such as rough front-to-back sorting [892, 1766] or performing a prepass for only a subset of objects [145, 1768], can avoid much overdraw without a second full geometry pass. That said, Pettineo [1407] finds that, even using such optimizations, using a separate z-prepass is faster. Alternately, deferred shading can be performed for opaque surfaces, with the same light list structure then used for forward shading on transparent surfaces. This approach is used in *Just Cause 3*, which creates the light lists on the CPU [1387]. Dufresne [390] also generates cluster light lists in parallel on the CPU, since this process has no dependence on the geometry in the scene.

Clustered light assignment gives fewer lights per list, and has less view dependence than tiled methods [1328, 1332]. The long, thin frusta defined by tiles can have considerable changes in their contents from just small movements of the camera. A straight line of streetlights, for example, can align to fill one tile [1387]. Even with z-depth subdivision methods, the near and far distances found from surfaces in each tile can radically shift due to a single pixel change. Clustering is less susceptible to such problems.

Several optimizations for clustered shading are explored by Olsson et al. [1328, 1329] and others, as noted. One technique is to form a BVH for the lights, which is then used to rapidly determine which light volumes overlap a given cluster. This BVH needs to be rebuilt as soon as at least one light moves. One option, usable with deferred shading, is to cull using quantized normal directions for the surfaces in a cluster. Olsson et al. categorize surface normals by direction into a structure holding 3×3 direction sets per face on a cube, 54 locations in total, in order to form a normal cone (Section 19.3). This structure can then be used to further cull out light sources when creating the cluster's list, i.e., those behind all surfaces in the cluster. Sorting can become expensive for a large number of lights, and van Oosten [1334] explores various strategies and optimizations.

When the visible geometry locations are available, as with deferred shading or from a z-prepass, other optimizations are possible. Clusters containing no geometry can be eliminated from processing, giving a sparse grid that requires less processing and storage. Doing so means that the scene must first be processed to find which clusters are occupied. Because this requires access to the depth buffer data, cluster formation must then be performed on the GPU. The geometry overlapping a cluster may have a small extent compared to the cluster's volume. More lights may be culled by using these samples to form a tight AABB to test against [1332]. An optimized system can handle upward of a million light sources and scales well as this number increases, while also being efficient for just a few lights.

There is no requirement to subdivide the screen z-axis using an exponential function, and such a subdivision may have a negative effect for scenes with many distant lights. With an exponential distribution, cluster volume increases with depth, which can result in a distant cluster's light list being excessively long. Limiting the cluster

set's maximum distance, the "far plane" for light clustering, is one solution, with more distant lights faded out, represented as particles or glares, or baked in [293, 432, 1768]. Simpler shaders, lightcuts [1832], or other level of detail techniques can also be used. Conversely, the volume closest to the viewer may be relatively unpopulated but heavily subdivided. One approach is to force the classifying frustum's "near plane" to some reasonable distance and categorize lights closer than this depth to fall into the first depth slice [1387].

In *DOOM* (2016), the developers [294, 1682] implemented their forward shading system using a combination of clustering methods from Olsson et al. [1328] and Persson [1387]. They first perform a z-prepass, which takes about 0.5 ms. Their list-building scheme can be thought of as clip-space voxelization. Light sources, environment light probes, and decals are inserted by testing each for intersection with an AABB representing each cell. The addition of decals is a significant improvement, as the clustered forward system gains the advantages that deferred shading has for these entities. During forward shading the engine loops through all decals found in a cell. If a decal overlaps the surface location, its texture values are retrieved and blended in. Decals can be blended with the underlying surface in any way desired, instead of being limited to only the operations available in the blending stage, as done with deferred shading. With clustered forward shading decals can also be rendered on transparent surfaces. All relevant lights in the cell are then applied.

The CPU can be used to build light lists because the scene's geometry is not necessary, and because analytically testing light volume spheres and cluster boxes for overlap is inexpensive. However, if a spotlight or other light volume shape is involved, using a spherical bounding volume around it can result in adding such a light to many clusters where it has no effect, and the precise analytic intersection test can be expensive. Along these lines, Persson [1387] provides a rapid method for voxelizing a sphere into a set of clusters.

The GPU's rasterization pipeline can be used to categorize light volumes to avoid these problems. Örtegren and Persson [1340] describe a two-pass process to build the light lists. In the shell pass, each light is represented by a low-resolution mesh that encompasses it. Conservative rasterization (Section 23.1.2) is used to render each of these shells into the cluster grid, recording the minimum and maximum cluster each overlaps. In the fill pass, a compute shader adds the light to a linked list for each cluster between these bounds. Using meshes instead of bounding spheres gives tighter bounds for spotlights, and geometry can directly occlude light visibility, culling the lists further still. When conservative rasterization is not available, Pettineo [1407] describes a method that employs surface gradients to conservatively estimate the z bounds of a triangle at each pixel. For example, if the farthest distance is needed at a pixel, the x- and y-depth gradients are used to select which corner of the pixel is farthest away and to compute the depth at that point. Because such points may be off the triangle, he also clamps to the z-depth range of the light as a whole, to avoid having a triangle that is nearly edge-on throw the estimated z-depth far off. Wronski [1922] explores a variety of solutions, landing on the idea of putting a bounding sphere around

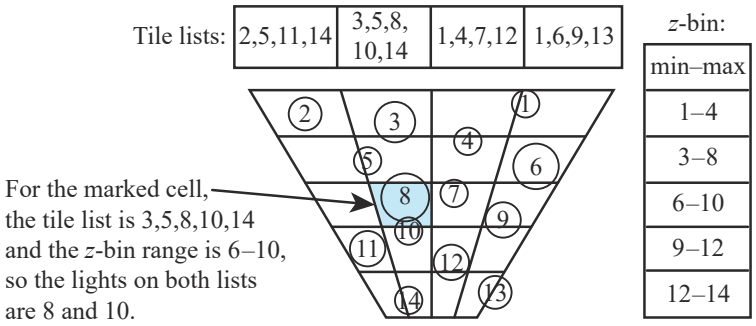

Figure 20.11. Using z-binning, each light is given an ID based on its z-depth. A list for each tile is generated. Each z-bin stores a minimum and maximum ID, a conservative range of lights that may overlap the slice. For any pixels in the marked cell, we retrieve both lists and find the overlap.

a grid cell and performing an intersection test against the cone. This test is quick to evaluate and works well when the cells are nearly cubic, less so when elongated.

Drobot [385] describes how meshes were used for inserting lights in *Call of Duty: Infinite Warfare*. Think of a static spotlight. It forms a volume in space, such as a cone. Without further processing, that cone could extend some considerable distance, either to the extents of the scene or to some maximum distance defined for the light. Now imagine a shadow map for this spotlight, generated using static geometry in the scene. This map defines a maximum distance for each direction in which the light shines. In a baking process, this shadow map is turned into a low-resolution mesh that then serves as the light's effective volume. The mesh is conservative, formed using the maximum depths in each shadow map area, so that it fully encompasses the volume of space that is illuminated by the light. This representation of the spotlight will likely overlap fewer clusters than the original cone's volume.

Independent of this process, the light list storage and access method, called *z-binning*, takes considerably less memory than clustered shading. In it, the lights are sorted by screen z-depth and given IDs based on these depths. A set of z-slices, each of the same depth thickness instead of exponential, is then used to classify these lights. Each z-slice stores just the minimum and maximum ID of the lights overlapping it. See Figure 20.11. Tiled shading lists are also generated, with geometry culling being optional. Each surface location then accesses this two-dimensional tiling structure and the one-dimensional z-bin ID range per slice. The tiling list gives all lights in a tile that may affect the pixel. The pixel's depth retrieves the range of IDs that may overlap that z-slice. The overlap of these two is computed on the fly and gives the valid light list for the cluster.

Instead of creating and storing a list for every cluster in a three-dimensional grid, this algorithm needs only a list per two-dimensional tile and a small fixed-sized array for the set of z-slices. Less storage, bandwidth use, and precomputing are needed, at the cost of a bit more work to determine the relevant lights at each pixel. Using

z-binning may lead to some lights being miscategorized, but Drobot found that for man-made environments there was often little overlap between lights in both the xy screen coordinates and the z-depths. Using pixel and compute shaders, this scheme is able to give near-perfect culling in tiles with depth discontinuities.

Three-dimensional data structures for accessing objects often can be categorized as volume-related, where a grid or octree is imposed on the space; object-related, where a bounding volume hierarchy is formed; or hybrid, such as using a bounding volume around the contents in a grid cell. Bezrati [139, 140] performs tiled shading in a compute shader to form an enhanced light list, where each light includes its minimum and maximum z-depth. In this way, a fragment can quickly reject any lights that do not overlap it. O'Donnell and Chajdas [1312] present *tiled light trees*, which they form on the CPU side. They use tiled light lists with depth bounds for each light and form a bounding interval hierarchy. That is, instead of forming a separate three-dimensional hierarchy of all lights, as done by Olsson et al. [1328], they create a simpler one-dimensional hierarchy from the z extents of each light in a tile. This structure maps well to the GPU's architecture, and is better able to handle cases where a large number of lights fall into a single tile. They also provide a hybrid algorithm that chooses between dividing the tile into cells—the normal clustered shading approach—or using light trees. Light trees work best in situations where the average overlap between a cell and its lights is low.

The idea of local light lists can be used on mobile devices, but there are different limitations and opportunities. For example, rendering one light at a time in a traditional deferred fashion can be the most efficient method on mobile, because of the unique property that mobile keeps the G-buffer in local memory. Tiled forward shading can be implemented on devices that support OpenGL ES 2.0, which is almost a given on mobile GPUs. With OpenGL ES 3.0 and an extension called *pixel local storage*, the tile-based rendering system available in ARM GPUs can be used to efficiently generate and apply light lists. See Billeter's presentation [145] for more information. Nummelin [1292] discusses conversion of the Frostbite engine from the desktop to mobile, including trade-offs with light classification schemes because compute shaders have less support on mobile hardware. Due to mobile devices using tile-based rendering, G-buffer data generated for deferred shading can be maintained in local memory. Smith and Einig [1664] describe using *framebuffer fetch* and pixel local storage to do so, finding that these mechanisms reduce overall bandwidth costs by more than half.

In summary, tiled, clustered, or other light-list culling techniques can be used with deferred or forward shading, and each can also be applied to decals. Algorithms for light-volume culling focus on minimizing the number of lights that are evaluated for each fragment, while the idea of decoupling geometry and shading can be used to balance processing and bandwidth costs to maximize efficiency. Just as frustum culling will cost additional time with no benefit if all objects are always in view, so will some techniques provide little benefit under various conditions. If the sun is the only light source, a light culling preprocess is not necessary. If there is little surface

	Trad. forward	Trad. deferred	Tiled/clust. defer.	Tiled/clust. fwd.
Geometry passes	1	1	1	1–2
Light passes	0	1 per light	1	1
Light culling	per mesh	per pixel	per volume	per volume
Transparency	easy	no	w/fwd→	easy
MSAA	built-in	hard	hard	built-in
Bandwidth	low	high	medium	low
Vary shading models	simple	hard	involved	simple
Small triangles	slow	fast	fast	slow
Register pressure	poss. high	low	poss. low	poss. high
Shadow map reuse	no	yes	no	no
Decals	expensive	cheap	cheap	expensive

Table 20.1. For a typical desktop GPU, comparison of traditional single-pass forward, deferred, and tiled/clustered light classification using deferred and forward shading. *(After Olsson [1332].)*

overdraw and few lights, deferred shading may cost more time overall. For scenes with many limited-effect sources of illumination, spending time to create localized light lists is worth the effort, whether using forward or deferred shading. When geometry is complex to process or surfaces are expensive to render, deferred shading provides a way to avoid overdraw, minimize driver costs such as program and state switches, and use fewer calls to render larger consolidated meshes. Keep in mind that several of these methods can be used in rendering a single frame. What makes for the best mix of techniques is not only dependent on the scene, but can also vary on a per-object or per-light basis [1589].

To conclude this section, we summarize the main differences among approaches in Table 20.1. The right arrow in the "Transparency" row means that deferred shading is applied to opaque surfaces, with forward shading needed for transparent ones. "Small triangles" notes an advantage of deferred shading, that quad shading (Section 23.1) can be inefficient when forward rendering, due to all four samples being fully evaluated. "Register pressure" refers to the overall complexity of the shaders involved. Using many registers in a shader can mean fewer threads formed, leading to the GPU's warps being underused [134]. It can become low for tiled and clustered deferred techniques if shader streamlining methods are employed [273, 414, 1877]. Shadow map reuse is often not as critical as it once was, when GPU memory was more limited [1589].

Shadows are a challenge when a large number of light sources are present. One response is to ignore shadow computations for all but the closest and brightest lights, and the sun, at the risk of light leaks from the lesser sources. Harada et al. [667] discuss how they use ray casting in a tiled forward system, spawning a ray for each visible surface pixel to each nearby light source. Olsson et al. [1330, 1331] discuss generating shadow maps using occupied grid cells as proxies for geometry, with samples created as needed. They also present a hybrid system, combining these limited shadow maps with ray casting.

Using world space instead of screen space for generating light lists is another way to structure space for clustered shading. This approach can be reasonable in some situations, though might be worth avoiding for large scenes because of memory con-

straints [385] and because distant clusters will be pixel-sized, hurting performance. Persson [1386] provides code for a basic clustered forward system where static lights are stored in a three-dimensional world-space grid.

20.5 Deferred Texturing

Deferred shading avoids overdraw and its costs of computing a fragment's shade and then having these results discarded. However, when forming G-buffers, overdraw still occurs. One object is rasterized and all its parameters are retrieved, performing several texture accesses along the way. If another object is drawn later that occludes these stored samples, all the bandwidth spent for rendering the first object was wasted. Some deferred shading systems perform a partial or full z-prepass to avoid texture access for a surface that is later drawn over by another object [38, 892, 1401]. However, an additional geometry pass is something many systems avoid if possible. Bandwidth is used by texture fetches, but also by vertex data access and other data. For detailed geometry, an extra pass can use more bandwidth than it might save in texture access costs.

The higher the number of G-buffers formed and accessed, the higher the memory and bandwidth costs. In some systems bandwidth may not be a concern, as the bottleneck could be predominantly within the GPU's processor. As discussed at length in Chapter 18, there is always a bottleneck, and it can and will change from moment to moment. A major reason why there are so many efficiency schemes is that each is developed for a given platform and type of scene. Other factors, such as how hard it is to implement and optimize a system, the ease of authoring content, and a wide variety of other human factors, can also determine what is built.

While both computational and bandwidth capabilities of GPUs have risen over time, they have increased at different rates, with compute rising faster. This trend, combined with new functionality on the GPU, has meant that a way to future-proof your system is by aiming for the bottleneck to be GPU computation instead of buffer access [217, 1332].

A few different schemes have been developed that use a single geometry pass and avoid retrieving textures until needed. Haar and Aaltonen [625] describe how virtual *deferred texturing* is used in *Assassin's Creed Unity*. Their system manages a local 8192×8192 texture atlas of visible textures, each with 128×128 resolution, selected from a much larger set. This atlas size permits storing (u, v) texture coordinates that can be used to access any texel in the atlas. There are 16 bits used to store a coordinate; with 13 bits needed for 8192 locations, this leaves 3 bits, i.e., 8 levels, for sub-texel precision. A 32-bit tangent basis is also stored, encoded as a quaternion [498] (Section 16.6). Doing so means only a single 64-bit G-buffer is needed. With no texture accesses performed in the geometry pass, overdraw can be extremely inexpensive. After this G-buffer is established, the virtual texture is then accessed during shading. Gradients are needed for mipmapping, but are not stored. Rather, each pixel's neighbors are examined and those with the closest (u, v) values are used

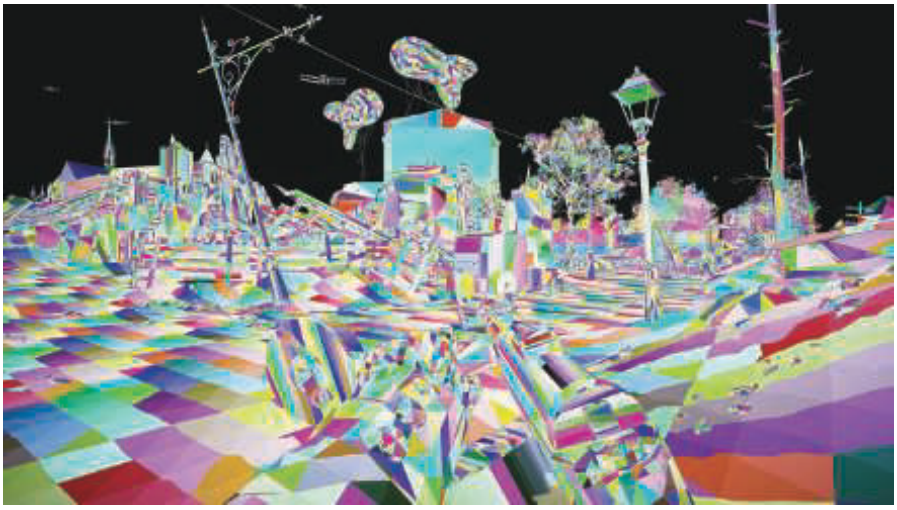

Figure 20.12. In the first pass of the visibility buffer [217], just triangle and instance IDs are rendered and stored in a single G-buffer, here visualized with a different color per triangle. *(Image courtesy of Graham Wihlidal—Electronic Arts [1885].)*

to compute the gradients on the fly. The material ID is also derived from determining which texture atlas tile is accessed, done by dividing the texture coordinate values by 128, i.e., the texture resolution.

Another technique used in this game to reduce shading costs is to render at a quarter resolution and use a special form of MSAA. On consoles using AMD GCN, or systems using OpenGL 4.5, OpenGL ES 3.2, or other extensions [2, 1406], the MSAA sampling pattern can be set as desired. Haar and Aaltonen set a grid pattern for 4× MSAA, so that each grid sample corresponds directly to the center of a full-screen pixel. By rendering at a quarter resolution, they can take advantage of the multisampling nature of MSAA. The (u, v) and tangent bases can be interpolated across the surface with no loss, and 8× MSAA (equivalent to 2× MSAA per pixel) is also possible. When rendering scenes with considerable overdraw, such as foliage and trees, their technique significantly reduces the number of shader invocations and bandwidth costs for the G-buffers.

Storing just the texture coordinates and basis is pretty minimal, but other schemes are possible. Burns and Hunt [217] describe what they call the *visibility buffer*, in which they store two pieces of data, a triangle ID and an instance ID. See Figure 20.12. The geometry pass shader is extremely quick, having no texture accesses and needing to store just these two ID values. All triangle and vertex data—positions, normals, color, material, and so on—are stored in global buffers. During the deferred shading pass, the triangle and instance IDs stored for each pixel are used to retrieve these data. The view ray for the pixel is intersected against the triangle to find the barycentric coordinates, which are used to interpolate among the triangle's vertex data. Other

computations that are normally done less frequently must also be performed per pixel, such as vertex shader calculations. Texture gradient values are also computed from scratch for each pixel, instead of being interpolated. All these data are then used to shade the pixel, applying lights using any classification scheme desired.

While this all sounds expensive, remember that computational power is growing faster than bandwidth capabilities. This research favors a compute-heavy pipeline that minimizes bandwidth losses due to overdraw. If there are less than 64k meshes in the scene, and each mesh has less than 64k triangles, each ID is then 16 bits in length and the G-buffer can be as small as 32 bits per pixel. Larger scenes push this to 48 or 64 bits.

Stachowiak [1685] describes a variant of the visibility buffer that uses some capabilities available on the GCN architecture. During the initial pass, the barycentric coordinates for the location on the triangle are also computed and stored per pixel. A GCN fragment (i.e., pixel) shader can compute the barycentric coordinates inexpensively, compared to later performing an individual ray/triangle intersection per pixel. While costing additional storage, this approach has an important advantage. With animated meshes the original visibility buffer scheme needs to have any modified mesh data streamed out to a buffer, so that the modified vertex locations can be retrieved during deferred shading. Saving the transformed mesh coordinates consumes additional bandwidth. By storing the barycentric coordinates in the first pass, we are done with the vertex positions, which do not have to be fetched again, a disadvantage of the original visibility buffer. However, if the distance from the camera is needed, this value must also be stored in the first pass, since it cannot later be reconstructed.

This pipeline lends itself to decoupling geometry and shading frequency, similar to previous schemes. Aaltonen [2] notes that the MSAA grid sampling method can be applied to each, leading to further reductions in the average amount of memory required. He also discusses variations in storage layout and differences in compute costs and capabilities for these three schemes. Schied and Dachsbacher [1561, 1562] go the other direction, building on the visibility buffer and using MSAA functionality to reduce memory consumption and shading computations for high-quality antialiasing.

Pettineo [1407] notes that the availability of bindless texture functionality (Section 6.2.5) makes implementing deferred texturing simpler still. His deferred texturing system creates a larger G-buffer, storing the depth, a separate material ID, and depth gradients. Rendering the Sponza model, this system's performance was compared against a clustered forward approach, with and without z-prepass. Deferred texturing was always faster than forward shading when MSAA was off, slowing when MSAA was applied. As noted in Section 5.4.2, most video games have moved away from MSAA as screen resolutions have increased, instead relying on temporal antialiasing, so in practical terms such support is not all that important.

Engel [433] notes that the visibility buffer concept has become more attractive due to API features exposed in DirectX 12 and Vulkan. Culling sets of triangles (Section 19.8) and other removal techniques performed using compute shaders reduce the number of triangles rasterized. DirectX 12's `ExecuteIndirect` command can be

used to create the equivalent of an optimized index buffer that displays only those triangles that were not culled. When used with an advanced culling system [1883, 1884], his analysis determined that the visibility buffer outperformed deferred shading at all resolutions and antialiasing settings on the San Miguel scene. As the screen resolution rose, the performance gap increased. Future changes to the GPU's API and capabilities are likely to further improve performance. Lauritzen [993] discusses the visibility buffer and how there is a need to evolve the GPU to improve the way material shaders are accessed and processed in a deferred setting.

Doghramachi and Bucci [363] discuss their deferred texturing system in detail, which they call *deferred+*. Their system integrates aggressive culling techniques early on. For example, the previous frame's depth buffer is downsampled and reprojected in a way that provides a conservative culling depth for each pixel in the current scene. These depths help test occlusion for rendering bounding volumes of all meshes visible in the frustum, as briefly discussed in Section 19.7.2. They note that the alpha cutout texture, if present, must be accessed in any initial pass (or any z-prepass, for that matter), so that objects behind cutouts are not hidden. The result of their culling and rasterization process is a set of G-buffers that include the depth, texture coordinates, tangent space, gradients, and material ID, which are used to shade the pixels. While its number of G-buffers is higher than in other deferred texturing schemes, it does avoid unneeded texture accesses. For two simplified scene models from *Deus Ex: Mankind Divided*, they found that deferred+ ran faster than clustered forward shading, and believe that more complex materials and lighting would further widen the gap. They also noted that warp usage was significantly better, meaning that tiny triangles caused fewer problems, so GPU tessellation performed better. Their implementation of deferred texturing has several other advantages over deferred shading, such as being able to handle a wider range of materials more efficiently. The main drawbacks are those common to most deferred schemes, relating to transparency and antialiasing.

20.6 Object- and Texture-Space Shading

The idea of decoupling the rate at which the geometry is sampled from the rate at which shading values are computed is a recurring theme in this chapter. Here we cover several alternate approaches that do not easily fit in the categories covered so far. In particular, we discuss hybrids that draw upon concepts first seen in the *Reyes*[1] batch renderer [289], used for many years by Pixar and others to make their films. Now studios primarily use some form of ray or path tracing for rendering, but for its day, Reyes solved several rendering problems in an innovative and efficient way.

The key concept of Reyes is the idea of *micropolygons*. Every surface is diced into an extremely fine mesh of quadrilaterals. In the original system, dicing is done

[1]The name "Reyes" was inspired by Point Reyes peninsula, and is sometimes capitalized as "REYES," an acronym meaning "Renders Everything You Ever Saw."

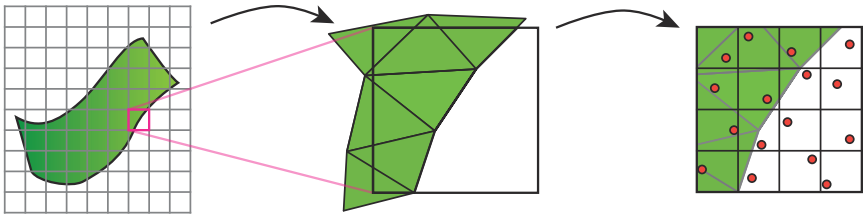

Figure 20.13. Reyes rendering pipeline. Each object is tessellated into micropolygons, which are then individually shaded. A set of jittered samples for each pixel (in red) are compared to the micropolygons and the results are used to render the image.

with respect to the eye, with the goal of having each micropolygon be about half the width and height of a pixel, so that the Nyquist limit (Section 5.4.1) is maintained. Quadrilaterals outside the frustum or facing away from the eye are culled. In this system, the micropolygon was shaded and assigned a single color. This technique evolved to shading the vertices in the micropolygon grid [63]. Our discussion here focuses on the original system, for the ideas it explored.

Each micropolygon is inserted into a jittered 4×4 sample grid in a pixel—a supersampled z-buffer. Jittering is done to avoid aliasing by producing noise instead. Because shading happens with respect to a micropolygon's coverage, before rasterization, this type of technique is called *object-based shading*. Compare this to forward shading, where shading takes place in screen space during rasterization, and deferred shading, where it takes place after. See Figure 20.13.

One advantage of shading in object space is that material textures are often directly related to their micropolygons. That is, the geometric object can be subdivided such that there is a power-of-two number of texels in each micropolygon. During shading the exact filtered mipmap sample can then be retrieved for the micropolygon, since it directly correlates to the surface area shaded. The original Reyes system also meant that cache coherent access of a texture occurs, since micropolygons are accessed in order. This advantage does not hold for all textures, e.g., environment textures used as reflection maps must be sampled and filtered in traditional ways.

Motion blur and depth-of-field effects can also work well with this type of arrangement. For motion blur each micropolygon is assigned a position along its path at a jittered time during the frame interval. So, each micropolygon will have a different location along the direction of movement, giving a blur. Depth of field is achieved in a similar fashion, distributing the micropolygons based on the circle of confusion.

There are some disadvantages to the Reyes algorithm. All objects must be able to be tessellated, and must be diced to a fine degree. Shading occurs before occlusion testing in the z-buffer, so can be wasted due to overdraw. Sampling at the Nyquist limit does not mean that high-frequency phenomena such as sharp specular highlights are captured, but rather that sampling is sufficient to reconstruct lower frequencies.

Generally, every object must be "chartable," in other words, must have (u, v) texture values for the vertices that give a unique texel for each different area on the

model. See Figure 2.9 on page 23 and Figure 6.6 on page 173 for examples. Object-based shading can be thought of as first baking in the shading, with the camera used to determine view-dependent effects and possibly limit the amount of effort expended for each surface area. One simple method of performing object-based shading on the GPU is to tessellate the object to a fine subpixel level, then shade each vertex on the mesh. Doing so can be costly because the setup costs for each triangle are not amortized over a number of pixels. The expense is made worse still because a single-pixel triangle generates four pixel shader invocations, due to quad rendering (Section 23.1). GPUs are optimized to render triangles that cover a fair number of pixels, e.g., 16 or more (Section 23.10.3).

Burns et al. [216] explore object-space shading by performing it after establishing which object locations are visible. They determine these with a "polygon grid" for an object that is diced, culled as possible, and then rasterized. An independent object-space "shading grid" is then used to shade the visible areas, with each texel corresponding to an area of the surface. The shading grid can be a different resolution than the polygon grid. They found that a finely tessellated geometric surface provided little benefit, so decoupling the two led to more efficient use of the resources. They implemented their work only in a simulator, but their techniques have influenced newer research and development.

Considerable research that draws inspiration from Reyes has examined faster shading methods on the GPU for various phenomena. Ragan-Kelley et al. [1455] propose a hardware extension based on decoupled sampling, applying their idea to motion blur and depth of field. Samples have five dimensions: two for the subpixel location, two for the lens location, and one for time. Visibility and shading are sampled separately. A "decoupling mapping" determines the shading sample needed for a given visibility sample. Liktor and Dachsbacher [1042, 1043] present a deferred shading system in a similar vein, where shading samples are cached when computed and used during stochastic rasterization. Effects such as motion blur and depth of field do not require high sampling rates, so shading computations can be reused. Clarberg et al. [271] present hardware extensions for computing shading in texture space. These eliminate the quad overshading problem and hence allow for smaller triangles. Since shading is computed in texture space, the pixel shader can use a bilinear filter or a more complex one, when looking up the shading from the texture. This allows reducing shading costs by turning down the texture resolution. For low-frequency terms, this technique usually works well, since filtering can be used.

Andersson et al. [48] take a different approach, called *texture-space shading*. Each triangle is tested for frustum and backface culling, then its charted surface is applied to a corresponding area of an output target, shading this triangle based on its (u, v) parameterization. At the same time, using a geometry shader, each visible triangle's size in the camera's view is computed. This size value is used to determine in which mipmap-like level the triangle is inserted. In this way, the amount of shading performed for an object is related to its screen coverage. See Figure 20.14. They use stochastic rasterization to render the final image. Each fragment generated looks up

Figure 20.14. Object-space texture shading. On the left is the final rendering, including motion blur. In the middle the visible triangles in the chart are shown. On the right each triangle is inserted at its proper mipmap level based on screen coverage of the triangle, for use during the final camera-based rasterization pass. *(Reprinted by permission of M. Andersson [48] and Intel Corporation, copyright Intel Corporation, 2014.)*

its shaded color from the texture. Again, computed shading values can be reused for motion blur and depth-of-field effects.

Hillesland and Yang [747, 748] build upon the texture-space shading concept, along with caching concepts similar to Liktor and Dachsbacher's. They draw geometry to the final view, use a compute shader to populate a mipmap-like structure of object-based shading results, and render the geometry again to access this texture and display the final shade. A triangle ID visibility buffer is also saved in the first pass, so that their compute shader can later access vertex attributes for interpolation. Their system includes coherence over time. Since the shading is in object space, the same area is associated with the same output texture location for each frame. If a surface area's shade at a given mipmap level was computed previously and is not too old, it is reused instead of being recomputed. Results will vary with the material, lighting, and other factors, but they found that reusing a shading sample every other frame at 60 FPS led to negligible errors. They also determined that the mipmap level could be selected not only by screen size, but also by variation in other factors, such as the change in normal direction over an area. A higher mipmap level means that less shading is computed per screen fragment, which they found could lead to considerable savings.

Baker [94] describes Oxide Games' renderer for the game *Ashes of the Singularity*. It is inspired by Reyes, though the implementation details are considerably different, and uses texture-space shading for each model as a whole. Objects may have any number of materials covering their surfaces, which are differentiated by using masks. Their process is:

- Several large—4k × 4k, 16 bits per channel—"master" textures are allocated for shading.
- All objects are evaluated. If in view, an object's estimated area on the screen is computed.

- This area is used to assign a proportion of a master texture to each object. If the total requested area is larger than the texture space, proportions are scaled down to make the requests fit.
- Texture-based shading is performed in a compute shader, with each material attached to the model applied in turn. Results from each material are accumulated in the assigned master texture.
- Mipmap levels are computed for the master textures as needed.
- Objects are then rasterized, with the master textures used to shade them.

Using multiple materials per object allows such effects as having a single terrain model include dirt, roads, ground cover, water, snow, and ice, each with its own material BRDF. Antialiasing works on both a pixel level and a shader level, if desired, as the full information about an object's surface area and its relationship to the master texture is accessible during shading. This ability allows the system to stably handle models with extremely high specular powers, for example. Because shaded results are attached to objects as a whole, regardless of visibility, shading can also be computed at a different frame rate than rasterization. Shading at 30 FPS was found to be adequate, with rasterization occurring at 60 FPS, or 90 FPS for virtual reality systems. Having asynchronous shading means that the frame rate for the geometry can be maintained even if the shader load becomes too high.

There are several challenges in implementing such a system. About twice as many batches are sent overall, compared to a typical game engine, since each object's "material quadrilateral" is processed with a compute shader during the object-shading step, then the object is drawn during rasterization. Most batches are simple, however, and APIs such as DirectX 12 and Vulkan help remove overhead. How the master textures are allocated to objects based on their size makes a significant difference in image quality. Objects that are large on the screen, or otherwise vary in texel density, such as terrain, can have issues. Performing an additional stitching process is used to maintain a smooth transition among terrain tiles of different resolutions in the master texture. Screen-space techniques, such as ambient occlusion, are a challenge to implement. Like the original visibility buffer, animation affecting object shape must be done twice, for shading and for rasterization. Objects are shaded, then occluded, which is a source of waste. For an application with low depth complexity, such as a real-time strategy game, this cost can be relatively low. Each material is simple to evaluate, unlike complex deferred shaders, and shading is done on a chart for the whole object. Objects with simple shaders, such as particles and trees, receive little benefit from this technique. For performance, these effects can instead be rendered with forward shading. As can be seen in Figure 20.15, being able to handle many lights gives a richness to the rendered scene.

We end our discussion of efficient shading here. We have only touched upon a panoply of specialized techniques to improve speed and quality of results used in

Figure 20.15. A scene from *Ashes of the Singularity* lit by approximately a thousand lights. These include at least one light source for each vehicle and each bullet. *(Image courtesy of Oxide Games and Stardock Entertainment.)*

different applications. Our goal here is to present popular algorithms used to accelerate shading and explain how and why they arose. As graphics hardware capabilities and APIs evolve, and as screen resolutions, art workflows, and other elements change over time, efficient shading techniques will continue to be researched and developed in new and, likely, unanticipated ways.

If you have read the book through to this point, you now have working knowledge of the major algorithms that go into a modern interactive rendering engine. One of our goals is to get you up to speed so that you can comprehend current articles and presentations in the field. If you wish to see how these elements work together, we highly recommend you read the excellent articles on different commercial renderers by Courrèges [293, 294] and Anagnostou [38]. After this point, the chapters that follow delve deeper into several fields, such as rendering for virtual and augmented reality, algorithms for intersection and collision detection, and architectural features of graphics hardware.

Further Reading and Resources

So, of the various mixes of these approaches—deferred, forward, tiled, clustered, visibility—which is better? Each has its own strengths, and the answer is "it depends." Factors such as platform, scene characteristics, illumination model, and design goals can all play a part. As a starting spot, we recommend Pesce's extensive discussions [1393, 1397] about the effectiveness and trade-offs of various schemes.

The SIGGRAPH course "Real-Time Many-Light Management and Shadows with Clustered Shading" [145, 1331, 1332, 1387] presents a thorough run-through of tiled and clustered shading techniques and their use with deferred and forward shading, along with related topics such as shadow mapping and implementing light classification on mobile devices. An earlier presentation by Stewart and Thomas [1700] explains tiled shading and presents copious timing results showing how various factors affect performance. Pettineo's open-source framework [1401] compares tiled forward and deferred systems, and includes results for a wide range of GPUs.

For implementation details, the book on DirectX 11 by Zink et al. [1971] has about 50 pages on the subject of deferred shading and includes numerous code samples. The NVIDIA GameWorks code samples [1299] include an implementation of MSAA for deferred shading. The articles by Mikkelsen [1210] and Örtegren and Persson [1340] in the book *GPU Pro 7* describe modern GPU-based systems for tiled and clustered shading. Billeter et al. [144] give coding details on implementing tiled forward shading, and Stewart [1701] walks through code for performing tiled culling in a compute shader. Lauritzen [990] provides a full implementation for tiled deferred shading, and Pettineo [1401] builds a framework to compare it to tiled forward. Dufresne [390] has demo code for clustered forward shading. Persson [1386] provides code for a basic world-space clustered forward rendering solution. Finally, van Oosten [1334] discusses various optimizations and gives a demo system with code that implements different forms of clustered, tiled, and vanilla forward rendering, showing the performance differences.

Chapter 21
Virtual and Augmented Reality

"Reality is that which, when you stop believing in it, doesn't go away."
—Philip K. Dick

Virtual reality (VR) and augmented reality (AR) are technologies that attempt to stimulate your senses in the same way the real world does. Within the field of computer graphics, augmented reality integrates synthetic objects with the world around us; virtual reality replaces the world completely. See Figure 21.1. This chapter focuses on rendering techniques specific to these two technologies, which are sometimes grouped together using the umbrella term "XR," where the X can stand for any letter. Much of the focus here will be on virtual reality techniques, since this technology is more widespread as of this writing.

Rendering is but a small part of these fields. From a hardware standpoint, some type of GPU is used, which is a well-understood piece of the system. Creating accurate and comfortable head-tracking sensors [994, 995], effective input devices (possibly with haptic feedback or eye-tracking control), and comfortable headgear and optics, along with convincing audio, are among the challenges system creators face. Balancing performance, comfort, freedom of movement, price, and other factors make this a demanding design space.

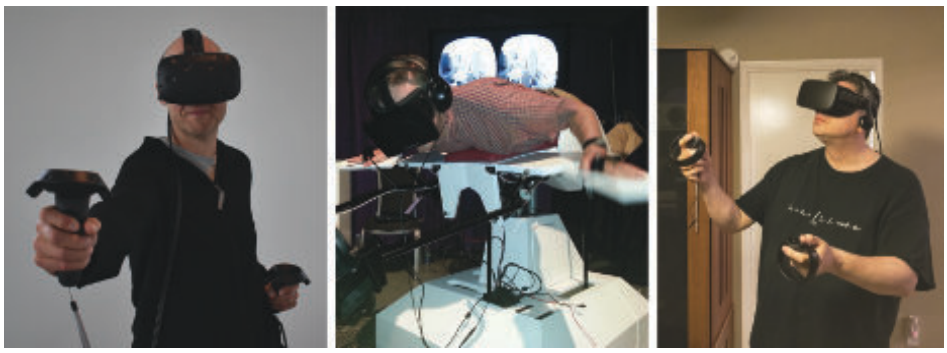

Figure 21.1. The first three authors using various VR systems. Tomas using an HTC Vive; Eric in the Birdly fly-like-a-bird simulator; Naty using an Oculus Rift.

We concentrate on interactive rendering and the ways these technologies influence how images are generated, starting with a brief survey of various virtual and augmented reality systems currently available. The capabilities and goals of some systems' SDKs and APIs are then discussed. We end with specific computer graphics techniques that should be avoided or modified to give the best user experience.

21.1 Equipment and Systems Overview

Aside from CPUs and GPUs, virtual and augmented reality equipment for graphics can be categorized as either sensors or displays. Sensors include trackers that detect the rotation and position of the user, along with a myriad of input methods and devices. For display, some systems rely on using a mobile phone's screen, which is logically split into two halves. Dedicated systems often have two separate displays. The display is all the user sees in a virtual reality system. For augmented reality, the virtual is combined with a view of the real world by use of specially designed optics.

Virtual and augmented reality are old fields that have undergone a recent explosion in new, lower-cost systems, in good part directly or indirectly due to the availability of various mobile and console technologies [995]. Phones can be used for immersive experiences, sometimes surprisingly well. A mobile phone can be placed inside a *head-mounted display* (HMD), ranging from a simple viewer, such as Google Cardboard, to those that are hands-free and provide additional input devices, such as GearVR. The phone's orientation sensors for gravity, magnetic north, and other mechanisms allow the orientation of the display to be determined. The orientation, also called *attitude*, has three degrees of freedom, e.g., yaw, pitch, and roll, as discussed in Section 4.2.1.[1] APIs can return the orientation as a set of Euler angles, a rotation matrix, or a quaternion. Real-world content such as fixed-view panoramas and videos can work well with these devices, as the costs of presenting the correct two-dimensional view for the user's orientation are reasonably low.

Mobile devices' relatively modest computational capabilities, as well as the power requirements for extended use of GPU and CPU hardware, limit what can be done with them. Tethered virtual reality devices, in which the user's headset is connected by a set of wires to a stationary computer, limit mobility, but allow more powerful processors to be used.

We will briefly describe the sensors for just two systems, the Oculus Rift and the HTC Vive. Both provide six degrees of freedom (6-DOF) tracking: orientation and position. The Rift tracks the location of the HMD and controllers by up to three separate infrared cameras. When the headset's position is determined by stationary external sensors, this is called *outside-in tracking*. An array of infrared LEDs on the outside of the headset allow it to be tracked. The Vive uses a pair of "lighthouses" that shine non-visible light into a room at rapid intervals, which sensors in the headset and

[1] Many phones' inertial measurement units have six degrees of freedom, but positional tracking errors can accumulate rapidly.

controllers detect in order to triangulate their positions. This is a form of *inside-out tracking*, where the sensors are part of the HMD.

Hand controllers are a standard piece of equipment, being trackable and able to move with the user, unlike mice and keyboards. Many other types of input devices have been developed for VR, based on a wide range of technologies. Devices include gloves or other limb or body tracking devices, eye tracking, and those simulating in-place movement, such as pressure pads, single- or omni-directional treadmills, stationary bicycles, and human-sized hamster balls, to name but a few. Aside from optical systems, tracking methods based on magnetic, inertia, mechanical, depth detection, and acoustic phenomena have been explored.

Augmented reality is defined as computer-generated content combined with a user's real-world view. Any application providing a heads-up display (HUD) with text data overlaid on an image is a basic form of augmented reality. Yelp Monocle, introduced in 2009, overlays business user ratings and distances on the camera's view. The mobile version of Google Translate can replace signs with translated equivalents. Games such as *P*okémon GO overlay imaginary creatures in real environments. Snapchat can detect facial features and add costume elements or animations.

Of more interest for synthetic rendering, *mixed reality* (MR) is a subset of augmented reality in which real-world and three-dimensional virtual content blend and interact in real time [1570]. A classic use case for mixed reality is in surgery, where scanned data for a patient's organs are merged with the camera view of the external body. This scenario assumes a tethered system with considerable computing power and precision. Another example is playing "tag" with a virtual kangaroo, where the real-world walls of the house can hide your opponent. In this case, mobility is more important, with registration or other factors affecting quality being less critical.

One technology used in this field is to mount a video camera on the front of an HMD. For example, every HTC Vive has a front-mounted camera that the developer can access. This view of the world is sent to the eyes, and synthetic imagery can be composited with it. This is sometimes called *pass-through* AR or VR, or *mediated reality* [489], in which the user is not directly viewing the environment. One advantage of using such a video stream is that it allows more control of merging the virtual objects with the real. The downside is that the real world is perceived with some lag. Vrvana's Totem and Occipital's Bridge are examples of AR systems using a head-mounted display with this type of arrangement.

Microsoft's HoloLens is the most well-known mixed-reality system as of this book's writing. It is an untethered system, with CPU, GPU, and what Microsoft terms an HPU (holographic processing unit) all built into the headset. The HPU is a custom chip consisting of 24 digital signal processing cores that draws less than 10 watts. These cores are used to process world data from a Kinect-like camera that views the environment. This view, along with other sensors such as accelerometers, perform inside-out tracking, with the additional advantage that no lighthouses, QR codes (a.k.a. fiducials), or other external elements are needed. The HPU is used to identify a limited set of hand gestures, meaning that no additional input device is necessary

for basic interactions. While scanning the environment, the HPU also extracts depths and derives geometry data, such as planes and polygons representing surfaces in the world. This geometry can then be used for collision detection, e.g., having virtual objects sit on a real-world tabletop.

Tracking using the HPU allows a wider range of motion, effectively anywhere in the world, by creating real-world waypoints, called *spatial anchors*. A virtual object's position is then set relative to a particular spatial anchor [1207]. The device's estimates of these anchor positions can also improve over time. Such data can be shared, meaning that a few users can see the same content in the same location. Anchors can also be defined so that users at different locations can collaborate on the same model.

A pair of transparent screens allow the user to see the world along with whatever is projected onto these screens. Note that this is unlike a phone's use of augmented reality, where the view of the world is captured by camera. One advantage of using transparent screens is that the world itself never has latency or display problems, and consumes no processing power. A disadvantage of this type of display system is that virtual content can only add luminance to the user's view of the world. For example, a dark virtual object will not occlude brighter real world objects behind it, since light can only be increased. This can give virtual objects a translucent feel. The HoloLens also has an LCD dimmer that can help avoid this effect. With proper adjustment, the system can be effective in showing three-dimensional virtual objects merged with reality.

Apple's ARKit and Google's ARCore help developers create augmented reality apps for phones and tablets. The norm is to display a single (not stereoscopic) view, with the device held some distance from the eyes. Objects can be fully opaque, since they are overlaid on the video camera's view of the world. See Figure 21.2. For ARKit, inside-out tracking is performed by using the device's motion-sensing hardware along with a set of notable features visible to the camera. Tracking these feature points from frame to frame helps precisely determine the device's current position and orientation. Like the HoloLens, horizontal and vertical surfaces are discovered and the extents determined, with this information then made available to the developer [65].

Intel's Project Alloy is an untethered head-mounted display that, like the HoloLens, has a sensor array to detect large objects and walls in the room. Unlike the HoloLens, the HMD does not let the user directly view the world. However, its ability to sense its surroundings gives what Intel calls "merged reality," where real-world objects can have a convincing presence in a virtual world. For example, the user could reach out to a control console in the virtual world and touch a table in the real world.

Virtual and augmented reality sensors and controllers are undergoing rapid evolution, with fascinating technologies arising at a breakneck pace. These offer the promise of less-intrusive headsets, more mobility, and better experiences. For example, Google's Daydream VR and Qualcomm's Snapdragon VR headsets are untethered and use inside-out positional tracking that does not need external sensors or devices. Systems from HP, Zotac, and MSI, where the computer is mounted on your back, make for untethered systems that provide more compute power. Intel's WiGig wire-

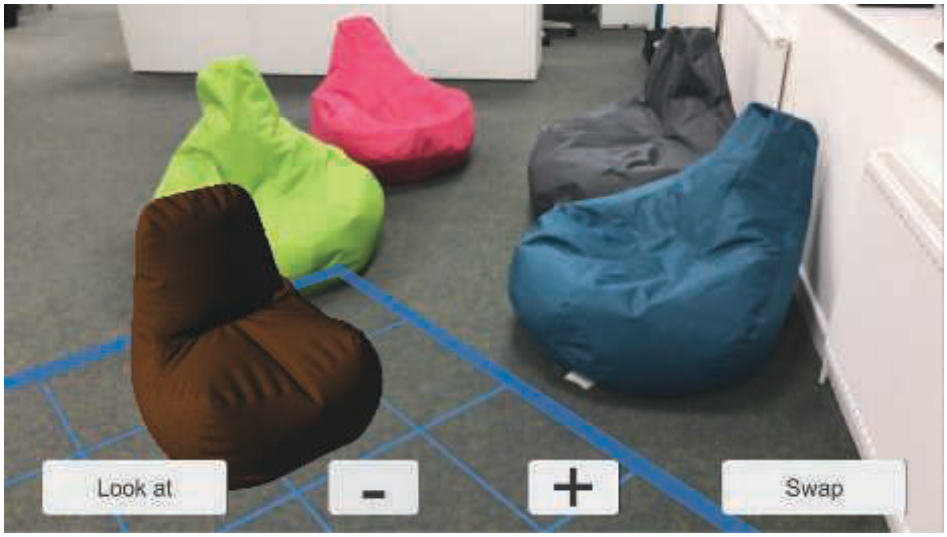

Figure 21.2. An image from ARKit. The ground plane is detected and displayed as a blue grid. The closest beanbag chair is a virtual object added on the ground plane. It is missing shadows, though these could be added for the object and blended over the scene. *(Image courtesy of Autodesk, Inc.)*

less networking technology uses a short-range 90 GHz radio to send images from a PC to a headset. Another approach is to compute expensive lighting computations on the cloud, then send this compressed information to be rendered by a lighter, less powerful GPU in a headset [1187]. Software methods such as acquiring point clouds, voxelizing these, and rendering the voxelized representations at interactive rates [930] open up new ways for the virtual and real to merge.

Our focus for most of this chapter is on the display and its use in VR and AR. We first run through some of the physical mechanics of how images are displayed on the screen and some of the issues involved. The chapter continues with what the SDKs and hardware systems provide to simplify programming and enhance the user's perception of the scene. This section is followed by information about how these various factors affect image generation, with a discussion of how some graphical techniques need to be modified or possibly avoided altogether. We end with a discussion of rendering methods and hardware enhancements that improve efficiency and the participant's experience.

21.2 Physical Elements

This section is about the various components and characteristics of modern VR and AR systems, in particular those related to image display. This information gives a framework for understanding the logic behind the tools provided by vendors.

21.2.1 Latency

Mitigating the effects of latency is particularly important in VR and AR systems, often the most critical concern [5, 228]. We discussed how the GPU hides memory latency in Chapter 3. That type of latency, caused by operations such as texture fetches, is specific to a small portion of the entire system. What we mean here is the "motion-to-photon" latency of the system as a whole. That is, say you begin to turn your head to the left. How much time elapses between your head coming to face in a particular direction and the view generated from that direction being displayed? Processing and communication costs for each piece of hardware in the chain, from the detection of a user input (e.g., your head orientation) to the response (the new image being displayed), all add up to tens of milliseconds of latency.

Latency in a system with a regular display monitor (i.e., one not attached to your face) is annoying at worst, breaking the sense of interactivity and connection. For augmented and mixed reality applications, lower latency will help increase "pixel stick," or how well the virtual objects in the scene stay affixed to the real world. The more latency in the system, the more the virtual objects will appear to swim or float relative to their real-world counterparts. With immersive virtual reality, where the display is the only visual input, latency can create a much more drastic set of effects. Though not a true illness or disease, it is called *simulation sickness* and can cause sweating, dizziness, nausea, and worse. If you begin to feel unwell, immediately take off the HMD—you are not able to "power through" this discomfort, and will just become more ill [1183]. To quote Carmack [650], "Don't push it. We don't need to be cleaning up sick in the demo room." In reality, actual vomiting is rare, but the effects can nonetheless be severe and debilitating, and can be felt for up to a day.

Simulation sickness in VR arises when the display images do not match the user's expectations or perceptions through other senses, such as the inner ears' vestibular system for balance and motion. The lower the lag between head motion and the proper matching displayed image, the better. Some research points to 15 ms being imperceptible. A lag of more than 20 ms can definitely be perceived and has a deleterious effect [5, 994, 1311]. As a comparison, from mouse move to display, video games generally have a latency of 50 ms or more, 30 ms with vsync off (Section 23.6.2). A display rate of 90 FPS is common among VR systems, which gives a frame time of 11.1 ms. On a typical desktop system it takes about 11 ms to scan the frame over a cable to the display, so even if you could render in 1 ms, you would still have 12 ms of latency.

There are a wide variety of application-based techniques that can prevent or ameliorate discomfort [1089, 1183, 1311, 1802]. These can range from minimizing visual flow, such as not tempting the user to look sideways while traveling forward and avoiding going up staircases, to more psychological approaches, such as playing ambient music or rendering a virtual object representing the user's nose [1880]. More muted colors and dimmer lighting can also help avoid simulator sickness. Making the system's response match the user's actions and expectations is the key to providing an

enjoyable VR experience. Have all objects respond to head movements, do not zoom the camera or otherwise change the field of view, properly scale the virtual world, and do not take control of the camera away from the user, to name a few guidelines. Having a fixed visual reference around the user, such as a car or airplane cockpit, can also diminish simulator sickness. Visual accelerations applied to a user can cause discomfort, so using a constant velocity is preferable. Hardware solutions may also prove useful. For example, Samsung's Entrim 4D headphones emit tiny electrical impulses that affect the vestibular system, making it possible to match what the user sees to what their sense of balance tells them. Time will tell as to the efficacy of this technique, but it is a sign of how much research and development is being done to mitigate the effects of simulator sickness.

The *tracking pose*, or simply the *pose*, is the orientation and, if available, position of the viewer's head in the real world. The pose is used to form the camera matrices needed for rendering. A rough prediction of the pose may be used at the start of a frame to perform simulation, such as collision detection of a character and elements in the environment. When rendering is about to start, a newer pose prediction can be retrieved at that moment and used to update the camera's view. This prediction will be more accurate, since it is retrieved later and is for a shorter duration. When the image is about to be displayed, another pose prediction that is more accurate still can be retrieved and used to warp this image to better match the user's position. Each later prediction cannot fully compensate for computations based on an earlier, inaccurate prediction, but using them as possible can considerably improve the overall experience. Hardware enhancements in various rigs provide the ability to rapidly query and obtain updated head poses at the moment they are needed.

There are elements other than visuals that make interaction with a virtual environment convincing, but getting the graphics wrong dooms the user to an unpleasant experience at best. Minimizing latency and improving realism in an application can help achieve *immersion* or *presence*, where the interface falls away and the participant feels physically a part of the virtual world.

21.2.2 Optics

Designing precise physical optics that map a head-mounted display's contents to the corresponding locations on the retina is an expensive proposition. What makes virtual reality display systems affordable is that the images produced by the GPU are then distorted in a separate post-process so that they properly reach our eyes.

A VR system's lenses present the user with a wide field-of-view image that has pincushion distortion, where the edges of the image appear to curve inward. This effect is canceled out by warping each generated image using barrel distortion, as seen on the right in Figure 21.3. Optical systems usually also suffer from *chromatic aberration*, where the lenses cause the colors to separate, like a prism does. This problem can also be compensated for by the vendor's software, by generating images that have an inverted chromatic separation. It is chromatic aberration "in the other

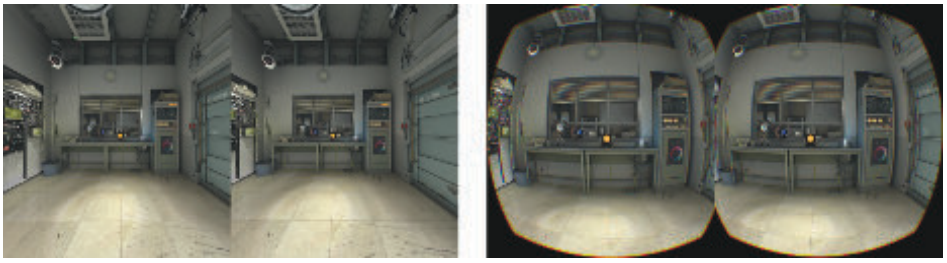

Figure 21.3. The original rendered targets (left) and their distorted versions (right) for display on an HTC Vive [1823]. *(Images courtesy of Valve.)*

direction." These separate colors combine properly when displayed through the VR system's optics. This correction can be seen in the orange fringing around the edges of the images in the distorted pair.

There are two types of displays, rolling and global [6]. For both types of display, the image is sent in a serial stream. In a *rolling display*, this stream is immediately displayed as received, scanline by scanline. In a *global display*, once the whole image is received, it is then displayed in a single, short burst. Each type of display is used in virtual reality systems, and each has its own advantages. In comparison to a global display, which must wait for the entire image to be present before display, a rolling display can minimize latency, in that the results are shown as soon as available. For example, if images are generated in strips, each strip could be sent as rendered, just before display, "racing the beam" [1104]. A drawback is that different pixels are illuminated at different times, so images can be perceived as wobbly, depending on the relative movement between the retinas and the display. Such mismatches can be particularly disconcerting for augmented reality systems. The good news is that the compositor usually compensates by interpolating the predicted head poses across a block of scan lines. This mostly addresses wobble or shearing that would otherwise happen with fast head rotation, though cannot correct for objects moving in the scene.

Global displays do not have this type of timing problem, as the image must be fully formed before it is shown. Instead, the challenge is technological, as a single short timed burst rules out several display options. Organic light-emitting diode (OLED) displays are currently the best option for global displays, as they are fast enough to keep up with the 90 FPS display rates popular for VR use.

21.2.3 Stereopsis

As can be seen in Figure 21.3, two images are offset, with a different view for each eye. Doing so stimulates *stereopsis*, the perception of depth from having two eyes.While an important effect, stereopsis weakens with distance, and is not our only way of perceiving depth.We do not use it at all, for example, when looking at an image on a standard monitor. Object size, texture pattern changes, shadows, relative movement (parallax), and other visual depth cues work with just one eye.

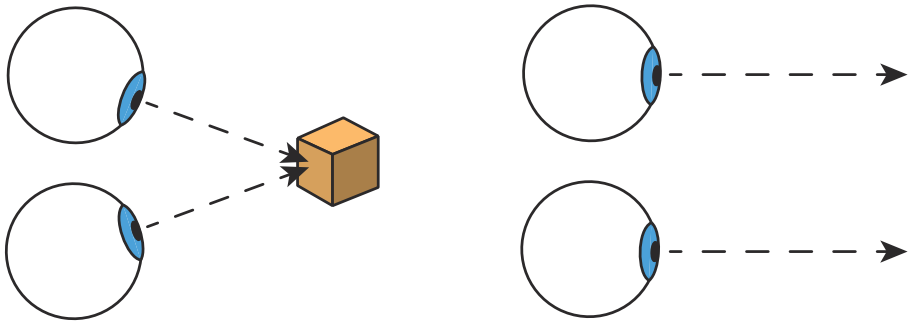

Figure 21.4. How much two eyes rotate to see an object is the vergence. Convergence is the motion of the eyes inward to focus on an object, as on the left. Divergence is the outward motion when they change to look at objects in the distance, off the right edge of the page. The lines of sight for viewing distant objects are effectively parallel.

How much the eyes must adjust shape to bring something into focus is known as *accommodative demand*. For example, the Oculus Rift's optics are equivalent to looking at a screen located about 1.3 meters from the user. How much the eyes need to turn inward to focus on an object is called *vergence demand*. See Figure 21.4. In the real world the eyes change lens shape and turn inward in unison, a phenomenon known as the *accommodation-convergence reflex*. With a display, the accommodative demand is a constant, but the vergence demand changes as the eyes focus on objects at different perceived depths. This mismatch can cause eye strain, so Oculus recommends that any objects the user is going to see for an extended period of time be placed about 0.75 to 3.5 meters away [1311, 1802]. This mismatch can also have perceptual effects in some AR systems, for example, where the user may focus on a distant object in the real world, but then must refocus on an associated virtual billboard that is at a fixed depth near the eye. Hardware that can adjust the perceived focal distance based on the user's eye movements, sometimes called an *adaptive focus* or *varifocal* display, is under research and development by a number of groups [976, 1186, 1875].

The rules for generating stereo pairs for VR and AR are different than those for single-display systems where some technology (polarized lens, shutter glasses, multi-view display optics) presents separate images to each eye from the same screen. In VR each eye has a separate display, meaning that each must be positioned in a way that the images projected onto the retinas will closely match reality. The distance from eye to eye is called the *interpupillary distance* (IPD). In one study of 4000 U.S. Army soldiers, the IPD was found to range from 52 mm to 78 mm, with an average of 63.5 mm [1311]. VR and AR systems have calibration methods to determine and adjust to the user's IPD, thus improving image quality and comfort. The system's API controls a camera model that includes this IPD. It is best to avoid modifying a user's perceived IPD to achieve an effect. For example, increasing the eye-separation distance could enhance the perception of depth, but can also lead to eye strain.

Stereo rendering for head-mounted displays is challenging to perform properly from scratch. The good news is that much of the process of setting up and using the proper camera transform for each eye is handled by the API, the subject of the next section.

21.3 APIs and Hardware

Let us say this from the start: Always use the VR software development kit (SDK) and application programming interface (API) provided by the system provider, unless you have an excellent reason to do otherwise. For example, you might believe your own distortion shader is faster and looks about right. In practice, however, it may well cause serious user discomfort—you will not necessarily know whether this is true without extensive testing. For this and other reasons, application-controlled distortion has been removed from all major APIs; getting VR display right is a system-level task. There is much careful engineering done on your behalf to optimize performance and maintain quality. This section discusses what support various vendors' SDKs and APIs provide.

The process for sending rendered images of a three-dimensional scene to a headset is straightforward. Here we will talk about it using elements common to most virtual and augmented reality APIs, noting vendor-specific functionality along the way. First, the time when the frame about to be rendered will be displayed is determined. There is usually support for helping you estimate this time delay. This value is needed so that the SDK can compute an estimate of where and in which direction the eyes will be located at the moment the frame is seen. Given this estimated latency, the API is queried for the pose, which contains information about the camera settings for each eye. At a minimum this consists of the head's orientation, along with the position, if sensors also track this information. The OpenVR API also needs to know if the user is standing or seated, which can affect what location is used as the origin, e.g., the center of the tracked area or the position of the user's head. If the prediction is perfect, then the rendered image will be displayed at the moment the head reaches the predicted location and orientation. In this way, the effect of latency can be minimized.

Given the predicted pose for each eye, you generally render the scene to two separate targets.[2] These targets are sent as textures to the SDK's *compositor*. The compositor takes care of converting these images into a form best viewed on the headset. The compositor can also composite various layers together. For example, if a monoscopic heads-up display is needed, one where the view is the same for both eyes, a single texture containing this element can be provided as a separate layer that is composited atop each eye's view. Textures can be different resolutions and formats, with the compositor taking care of conversion to the final eye buffers. Doing so can allow optimizations such as dynamically lowering the resolution of the three-dimensional scene's layer to save time on rendering [619, 1357, 1805], while maintaining high resolution and quality for the other layers [1311]. Once images are composed for each

[2] Some APIs instead accept a single target split into two views.

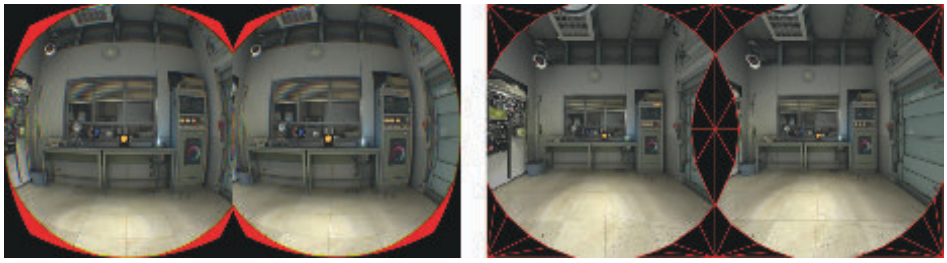

Figure 21.5. On the left, the red areas in the display image show pixels that are rendered and then warped, but that are not visible to the HMD user. Note that the black areas are outside the bounds of the transformed rendered image. On the right, these red areas are instead masked out in advance with the red-edged mesh at the start of rendering, resulting in this rendered (pre-warped) image needing fewer pixels to be shaded [1823]. Compare the right image with the original, on the left in Figure 21.3. *(Images courtesy of Valve.)*

eye, distortion, chromatic aberration, and any other processes needed are performed by the SDK and the results are then displayed.

If you rely on the API, you do not need to fully understand the algorithms behind some of these steps, since the vendor does much of the work for you. However, knowing a bit about this area is still worthwhile, if only to realize that the most obvious solution is not always the best one. To start, consider compositing. The most efficient way is to first composite all the layers together, and then to apply the various corrective measures on this single image. Instead, Oculus first performs these corrections separately to each layer, then composites these distorted layers to form the final, displayed image. One advantage is that each layer's image is warped at its own resolution, which can improve text quality, for example, because treating the text separately means that resampling and filtering during the distortion process is focused on just the text's content [1311].

The field of view a user perceives is approximately circular. What this means is that we do not need to render some of the pixels on the periphery of each image, near the corners. While these pixels will appear on the display, they are nearly undetectable by the viewer. To avoid wasting time generating these, we can first render a mesh to hide these pixels in the original images we generate. This mesh is rendered into the stencil buffer as a mask, or into the z-buffer at the front. Subsequent rendered fragments in these areas are then discarded before being evaluated. Vlachos [1823] reports that this reduces the fill rate by about 17% on the HTC Vive. See Figure 21.5. Valve's OpenVR API calls this pre-render mask the "hidden area mesh."

Once we have our rendered image, it needs to be warped to compensate for the distortion from the system's optics. The concept is to define a remapping of the original image to the desired shape for the display, as shown in Figure 21.3. In other words, given a pixel sample on the incoming rendered image, to where does this sample move in the displayed image? A ray casting approach can give the precise answer and adjust by wavelength [1423], but is impractical for most hardware. One method is

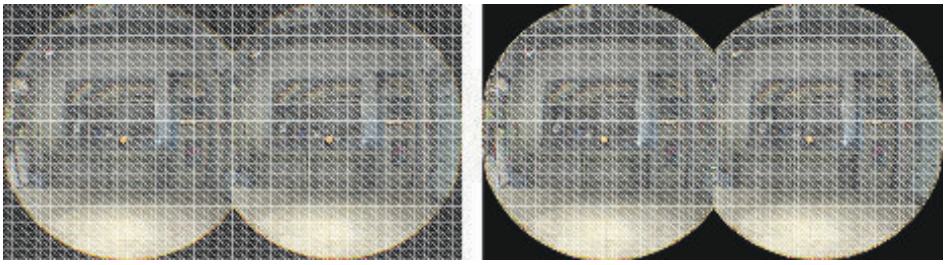

Figure 21.6. On the left, the mesh for the final, displayed image is shown. In practice, this mesh can be trimmed back to the culled version on the right, since drawing black triangles adds nothing to the final image [1823]. *(Images courtesy of Valve.)*

to treat the rendered image as a texture and draw a screen-filling quadrilateral to run a post-process. The pixel shader computes the exact location on this texture that corresponds to the output display pixel [1430]. However, this method can be expensive, as this shader has to evaluate distortion equations at every pixel.

Applying the texture to a mesh of triangles is more efficient. This mesh's shape can be modified by the distortion equation and rendered. Warping the mesh just once will not correct for chromatic aberration. Three separate sets of (u, v)-coordinates are used to distort the image, one for each color channel [1423, 1823]. That is, each triangle in the mesh is rendered once, but for each pixel the rendered image is sampled three times in slightly different locations. These red, green, and blue channel values then form the output pixel's color.

We can apply a regularly spaced mesh to the rendered image and warp to the displayed image, or vice versa. An advantage of applying the gridded mesh to the displayed image and warping back to the rendered image is that fewer 2×2 quads are likely to be generated, as no thin triangles will be displayed. In this case the mesh locations are not warped but rendered as a grid, and only the vertices' texture coordinates are adjusted in order to distort the image applied to the mesh. A typical mesh is 48×48 quadrilaterals per eye. See Figure 21.6. The texture coordinates are computed once for this mesh by using per-channel display-to-render image transforms. By storing these values in the mesh, no complex transforms are needed during shader execution. GPU support for anisotropic sampling and filtering of a texture can be used to produce a sharp displayable image.

The rendered stereo pair on the right in Figure 21.5 gets distorted by the display mesh. The slice removed in the center of this image corresponds to how the warping transform generates the displayable images—note how this slice is missing from where the images meet in the displayed version on the left in Figure 21.5. By trimming back the displayed warping mesh to only visible areas, as shown on the right in Figure 21.6, we can reduce the cost for the final distortion pass by about 15%.

To sum up the optimizations described, we first draw a hidden area mesh to avoid evaluating fragments in areas we know will be undetectable or unused (such as the

middle slice). We render the scene for both eyes. We then apply this rendered image to a gridded mesh that has been trimmed to encompass only the relevant rendered areas. Rendering this mesh to a new target gives us the image to display. Some or all of these optimizations are built in to virtual and augmented reality systems' API support.

21.3.1 Stereo Rendering

Rendering two separate views seems like it would be twice the work of rendering a single view. However, as Wilson notes [1891], this is not true for even a naive implementation. Shadow map generation, simulation and animation, and other elements are view-independent. The number of pixel shader invocations does not double, because the display itself is split in half between the two views. Similarly, post-processing effects are resolution-dependent, so those costs do not change either. View-dependent vertex processing is doubled, however, and so many have explored ways to reduce this cost.

Frustum culling is often performed before any meshes are sent down the GPU's pipeline. A single frustum can be used to encompass both eye frusta [453, 684, 1453]. Since culling happens before rendering, the exact rendered views to use may be retrieved after culling occurs. However, this means that a safety margin is needed during culling, since this retrieved pair of views could otherwise view models removed by the frustum. Vlachos [1823] recommends adding about 5 degrees to the field of view for predictive culling. Johansson [838] discusses how frustum culling and other strategies, such as instancing and occlusion cull queries, can be combined for VR display of large building models.

One method of rendering the two stereo views is to do so in a series, rendering one view completely and then the other. Trivial to implement, this has the decided disadvantage that state changes are also then doubled, something to avoid (Section 18.4.2). For tile-based renderers, changing your view and render target (or scissor rectangle) frequently will result in terrible performance. A better alternative is to render each object twice as you go, switching the camera transform in between. However, the number of API draw calls is still doubled, causing additional work. One approach that comes to mind is using the geometry shader to duplicate the geometry, creating triangles for each view. DirectX 11, for example, has support for the geometry shader sending its generated triangles to separate targets. Unfortunately, this technique has been found to lower geometry throughput by a factor of three or more, and so is not used in practice. A better solution is to use instancing, where each object's geometry is drawn twice by a single draw call [838, 1453]. User-defined clip planes are set to keep each eye's view separate. Using instancing is much faster than using geometry shaders, and is a good solution barring any additional GPU support [1823, 1891]. Another approach is to form a command list (Section 18.5.4) when rendering one eye's image, shift referenced constant buffers to the other eye's transform, and then replay this list to render the second eye's image [453, 1473].

There are several extensions that avoid sending geometry twice (or more) down the pipeline. On some mobile phones, an OpenGL ES 3.0 extension called *multi-view* adds support for sending the geometry only once and rendering it to two or more views, making adjustments to screen vertex positions and any view-dependent variables [453, 1311]. The extension gives more much freedom in implementing a stereo renderer. For example, the simplest extension is likely to use instancing in the driver, issuing the geometry twice, while an implementation requiring GPU support could send each triangle to each of the views. Different implementations have various advantages, but since API costs always are reduced, any of these methods can help CPU-bound applications. The more complex implementations can increase texture cache efficiency [678] and perform vertex shading of view-independent attributes only once, for example. Ideally, the entire matrix can be set for each view and any per-vertex attributes can also be shaded for each view. To make a hardware implementation use less transistors, a GPU can implement a subset of these features.

Multi-GPU solutions tuned for VR stereo rendering are available from AMD and NVIDIA. For two GPUs, each renders a separate eye's view. Using an *affinity mask*, the CPU sets a bit for all GPUs that are to receive particular API calls. In this way, calls can be sent to one or more GPUs [1104, 1453, 1473, 1495]. With affinity masks the API still needs to be called twice if a call differs between the right and left eye's view.

Another style of rendering provided by vendors is what NVIDIA calls *broadcasting*, where rendering to both eyes is provided using a single draw call, i.e., it is broadcast to all GPUs. Constant buffers are used to send different data, e.g., eye positions, to the different GPUs. Broadcasting creates both eyes' images with hardly any more CPU overhead than a single view, as the only cost is setting a second constant buffer.

Separate GPUs mean separate targets, but the compositor often needs a single rendered image. There is a special sub-rectangle transfer command that shifts render target data from one GPU to the other in a millisecond or less [1471]. It is asynchronous, meaning that the transfer can happen while the GPU does other work. With two GPUs running in parallel, both may also separately create the shadow buffer needed for rendering. This is duplicated effort, but is simpler and usually faster than attempting to parallelize the process and transfer between GPUs. This entire two-GPU setup results in about a 30 to 35% rendering speedup [1824]. For applications that are already tuned for single GPUs, multiple GPUs can instead apply their extra compute on additional samples for a better antialiased result.

Parallax from stereo viewing is important for nearby models, but is negligible for distant objects. Palandri and Green [1346] take advantage of this fact on the mobile GearVR platform by using a separating plane perpendicular to the view direction. They found a plane distance of about 10 meters was a good default. Opaque objects closer than this are rendered in stereo, and those beyond with a monoscopic camera placed between the two stereo cameras. To minimize overdraw, the stereo views are drawn first, then the intersection of their depth buffers is used to initialize the z-buffer for the single monoscopic render. This image of distant objects is then composited

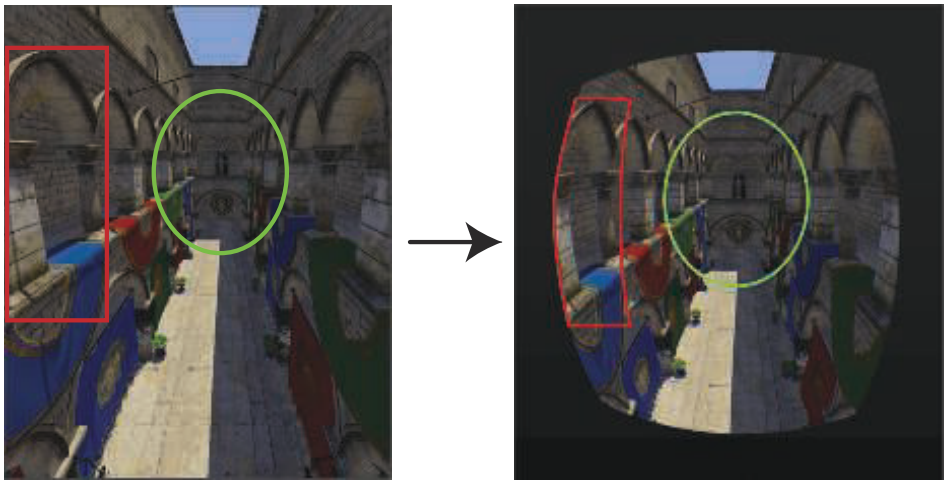

Figure 21.7. On the left is the rendered image for one eye. On the right is the warped image for display. Note how the green oval in the center maintains about the same area. On the periphery, a larger area (red outline) in the rendered image is associated with a smaller displayed area [1473]. *(Images courtesy of NVIDIA Corporation.)*

with each stereo view. Transparent content is rendered last for each view. While more involved, and with objects spanning the separating plane needing an additional pass, this method produced consistent overall savings of about 25%, with no loss in quality or depth perception.

As can be seen in Figure 21.7, a higher density of pixels is generated in the periphery of each eye's image, due to distortion needed by the optics. In addition, the periphery is usually less important, as the user looks toward the center of the screen a considerable amount of the time. For these reasons, various techniques have been developed for applying less effort to pixels on the periphery of each eye's view.

One method to lower the resolution along the periphery is called *multi-resolution shading* by NVIDIA and *variable rate shading* by AMD. The idea is to divide the screen into, e.g., 3×3 sections and render areas around the periphery at lower resolutions [1473], as shown in Figure 21.8. NVIDIA has had support for this partitioning scheme since their Maxwell architecture, but with Pascal on, a more general type of projection is supported. This is called *simultaneous multi-projection* (SMP). Geometry can be processed by up to 16 individual projections times 2 separate eye locations, allowing a mesh to be replicated up to 32 times without additional cost on the application side. The second eye location must be equal to the first eye location offset along the x-axis. Each projection can be independently tilted or rotated around an axis [1297].

Using SMP, one can implement *lens matched shading*, where the goal is to better match the rendered resolution to what is displayed. See Figure 21.7. Four frusta