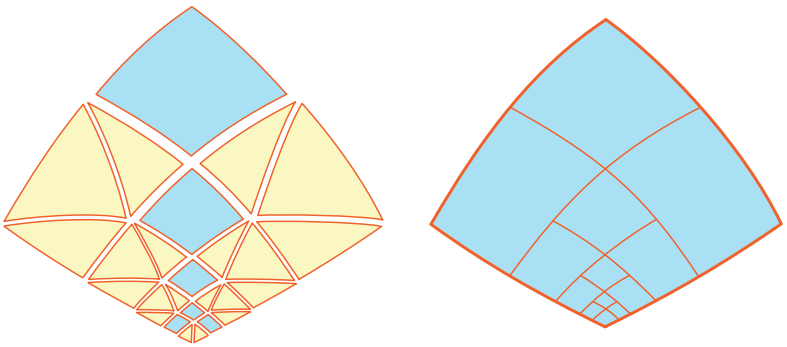

Figure 17.69. Left: hierarchical subdivision according to feature adaptive subdivision (FAS), where each triangle and quad is rendered as a separate tessellated primitive. Right: hierarchical subdivision using adaptive quadtrees, where the entire quad is rendered as a single tessellated primitive. (*Illustration after Brainerd et al. [190].*)

and handles creases and other topological features. An additional advantage of using adaptive quadtrees over FAS is illustrated in Figure 17.69 and further shown in Figure 17.70. Adaptive quadtrees also provide more uniform tessellations since there is a one-to-one mapping between each submitted quad and tessellated primitive.

Figure 17.70. Subdivision patches using adaptive quadtrees. Each patch, corresponding to a base mesh face, is surrounded by black curves on the surface and the subdivision steps are illustrated hierarchically inside each patch. As can be seen, there is one patch with uniform color in the center. This implies that it was rendered as a bicubic B-spline patch, while others (with extraordinary vertices) clearly show their underlying adaptive quadtrees. (*Image courtesy of Wade Brainerd.*)

Further Reading and Resources

The topic of curves and surfaces is huge, and for more information, it is best to consult the books that focus solely on this topic. Mortenson's book [1242] serves as a good general introduction to geometric modeling. Books by Farin [458, 460], and by Hoschek and Lasser [777] are general and treat many aspects of *Computer Aided Geometric Design* (CAGD). For implicit surfaces, consult the book by Gomes et al. [558] and the more recent article by de Araújo et al. [67]. For much more information on subdivision surfaces, consult Warren and Heimer's book [1847] and the SIGGRAPH course notes on "Subdivision for Modeling and Animation" by Zorin et al. [1977]. The course on substitutes for subdivision surfaces by Ni et al. [1275] is a useful resource here as well. The survey by Nießner et al. [1283] and Nießner's PhD thesis [1282] are great for information on real-time rendering of subdivision surfaces using the GPU.

For spline interpolation, we refer the interested reader to the Killer B's book [111] in addition to the books above by Farin [458] and Hoschek and Lasser [777]. Many properties of Bernstein polynomials, both for curves and surfaces, are given by Goldman [554]. Almost everything you need to know about triangular Bézier surfaces can be found in Farin's article [457]. Another class of rational curves and surfaces is the nonuniform rational B-spline (NURBS) [459, 1416, 1506], often used in CAD.

Chapter 18
Pipeline Optimization

> *"We should forget about small efficiencies, say about 97% of the time: Premature optimization is the root of all evil."*
> —Donald Knuth

Throughout this volume, algorithms have been presented within a context of quality, memory, and performance trade-offs. In this chapter we will discuss performance problems and opportunities that are not associated with particular algorithms. Bottleneck detection and optimization are the focus, starting with making small, localized changes, and ending with techniques for structuring an application as a whole to take advantage of multiprocessing capabilities.

As we saw in Chapter 2, the process of rendering an image is based on a pipelined architecture with four conceptual stages: *application*, *geometry processing*, *rasterization*, and *pixel processing*. There is always one stage that is the bottleneck—the slowest process in the pipeline. This implies that this bottleneck stage sets the limit for the throughput, i.e., the total rendering performance, and so is a prime candidate for *optimization*.

Optimizing the performance of the rendering pipeline resembles the procedure of optimizing a pipelined processor (CPU) [715] in that it consists mainly of two steps. First, the bottleneck of the pipeline is located. Second, that stage is optimized in some way; and after that, step one is repeated if the performance goals have not been met. Note that the bottleneck may or may not be located at the same place after the optimization step. It is a good idea to put only enough effort into optimizing the bottleneck stage so that the bottleneck moves to another stage. Several other stages may have to be optimized before this stage becomes the bottleneck again. For this reason, effort should not be wasted on over-optimizing a stage.

The location of the bottleneck may change within a frame, or even within a draw call. At one moment the geometry stage may be the bottleneck because many tiny triangles are rendered. Later in the frame pixel processing could be the bottleneck because a heavyweight procedural shader is evaluated at each pixel. In a pixel shader execution may stall because the texture queue is full, or take more time as a particular loop or branch is reached. So, when we talk about, say, the application stage being the

bottleneck, we mean it is the bottleneck most of the time during that frame. There is rarely only one bottleneck.

Another way to capitalize on the pipelined construction is to recognize that when the slowest stage cannot be optimized further, the other stages can be made to work just as much as the slowest stage. This will not change performance, since the speed of the slowest stage will not be altered, but the extra processing can be used to improve image quality [1824]. For example, say that the bottleneck is in the application stage, which takes 50 milliseconds (ms) to produce a frame, while the others each take 25 ms. This means that without changing the speed of the rendering pipeline (50 ms equals 20 frames per second), the geometry and the rasterizer stages could also do their work in 50 ms. For example, we could use a more sophisticated lighting model or increase the level of realism with shadows and reflections, assuming that this does not increase the workload on the application stage.

Compute shaders also change the way we think about bottlenecks and unused resources. For example, if a shadow map is being rendered, vertex and pixel shaders are simple and the GPU computational resources might be underutilized if fixed-function stages such as the rasterizer or the pixel merger become the bottleneck. Overlapping such draws with asynchronous compute shaders can keep the shader units busy when these conditions arise [1884]. Task-based multiprocessing is discussed in the final section of this chapter.

Pipeline optimization is a process in which we first maximize the rendering speed, then allow the stages that are not bottlenecks to consume as much time as the bottleneck. That said, it is not always a straightforward process, as GPUs and drivers can have their own peculiarities and fast paths. When reading this chapter, the dictum

KNOW YOUR ARCHITECTURE

should always be in the back of your mind, since optimization techniques vary greatly for different architectures. That said, be wary of optimizing based on a specific GPU's implementation of a feature, as hardware can and will change over time [530]. A related dictum is, simply,

MEASURE, MEASURE, MEASURE.

18.1 Profiling and Debugging Tools

Profiling and debugging tools can be invaluable in finding performance problems in your code. Capabilities vary and can include:

- Frame capture and visualization. Usually step-by-step frame replay is available, with the state and resources in use displayed.
- Profiling of time spent across the CPU and GPU, including time spent calling the graphics API.

- Shader debugging, and possibly hot editing to see the effects of changing code.
- Use of debug markers set in the application, to help identify areas of code.

Profiling and debugging tools vary with the operating system, the graphics API, and often the GPU vendor. There are tools for most combinations, and that's why the gods created Google. That said, we will mention a few package names specifically for interactive graphics to get you started on your quest:

- *RenderDoc* is a high-quality Windows debugger for DirectX, OpenGL, and Vulkan, originally developed by Crytek and now open source.
- *GPU PerfStudio* is AMD's suite of tools for their graphics hardware offerings, working on Windows and Linux. One notable tool provided is a static shader analyzer that gives performance estimates without needing to run the application. AMD's Radeon GPU Profiler is a separate, related tool.
- *NVIDIA Nsight* is a performance and debugging system with a wide range of features. It integrates with Visual Studio on Windows and Eclipse on Mac OS and Linux.
- Microsoft's *PIX* has long been used by Xbox developers and has been brought back for DirectX 12 on Windows. Visual Studio's *Graphics Diagnostics* can be used with earlier versions of DirectX.
- *GPUView* from Microsoft uses Event Tracing for Windows (ETW), an efficient event logging system. GPUView is one of several programs that are consumers of ETW sessions. It focuses on the interaction between CPU and GPU, showing which is the bottleneck [783].
- *Graphics Performance Analyzers* (GPA) is a suite from Intel, not specific to their graphics chips, that focuses on performance and frame analysis.
- Xcode on OSX provides *Instruments*, which has several tools for timing, performance, networking, memory leaks, and more. Worth mentioning are *OpenGL ES Analysis*, which detects performance and correctness problems and proposes solutions, and *Metal System Trace*, which provides tracing information from the application, driver, and GPU.

These are the major tools that have existed for a few years. That said, sometimes no tool will do the job. *Timer query* calls are built into most APIs to help profile a GPU's performance. Some vendors provide libraries to access GPU counters and thread traces as well.

18.2 Locating the Bottleneck

The first step in optimizing a pipeline is to locate the largest bottleneck [1679]. One way of finding bottlenecks is to set up several tests, where each test decreases the amount of work a particular stage performs. If one of these tests causes the frames per second to increase, the bottleneck stage has been found. A related way of testing a stage is to reduce the workload on the other stages without reducing the workload on the stage being tested. If performance does not change, the bottleneck is the stage where the workload was not altered. Performance tools can provide detailed information on which API calls are expensive, but do not necessarily pinpoint exactly what stage in the pipeline is slowing down the rest. Even when they do, it is useful to understand the idea behind each test.

What follows is a brief discussion of some of the ideas used to test the various stages, to give a flavor of how such testing is done. A perfect example of the importance of understanding the underlying hardware comes with the advent of the *unified shader architecture*. It forms the basis of many GPUs from the end of 2006 on. The idea is that vertex, pixel, and other shaders all use the same functional units. The GPU takes care of load balancing, changing the proportion of units assigned to vertex versus pixel shading. As an example, if a large quadrilateral is rendered, only a few shader units could be assigned to vertex transformation, while the bulk are given the task of fragment processing. Pinpointing whether the bottleneck is in the vertex or pixel shader stage is less obvious [1961]. Either shader processing as a whole or another stage will still be the bottleneck, however, so we discuss each possibility in turn.

18.2.1 Testing the Application Stage

If the platform being used is supplied with a utility for measuring the workload on the processor(s), that utility can be used to see if your program uses 100% (or near that) of the CPU processing power. If the CPU is in constant use, your program is likely to be *CPU-limited*. This is not always foolproof, since the application may at times be waiting for the GPU to complete a frame. We talk about a program being CPU- or GPU-limited, but the bottleneck can change over the lifetime of a frame.

A smarter way to test for CPU limits is to send down data that causes the GPU to do little or no work. For some systems this can be accomplished by simply using a null driver (a driver that accepts calls but does nothing) instead of a real driver. This effectively sets an upper limit on how fast you can get the entire program to run, because you do not use the graphics hardware nor call the driver, and thus, the application on the CPU is always the bottleneck. By doing this test, you get an idea on how much room for improvement there is for the GPU-based stages not run in the application stage. That said, be aware that using a null driver can also hide any bottleneck due to driver processing itself and communication between CPU and GPU. The driver can often be the cause of a CPU-side bottleneck, a topic we discuss in depth later on.

Another more direct method is to underclock the CPU, if possible [240]. If performance drops in direct proportion to the CPU rate, the application is CPU-bound to at least some extent. This same underclocking approach can be done for GPUs. If the GPU is slowed down and performance decreases, then at least some of the time the application is GPU-bound. These underclocking methods can help identify a bottleneck, but can sometimes cause a stage that was not a bottleneck before to become one. The other option is to overclock, but you did not read that here.

18.2.2 Testing the Geometry Processing Stage

The geometry stage is the most difficult stage to test. This is because if the workload on this stage is changed, then the workload on one or both of the other stages is often changed as well. To avoid this problem, Cebenoyan [240] gives a series of tests working from the rasterizer stages back up the pipeline.

There are two main areas where a bottleneck can occur in the geometry stage: vertex fetching and processing. To see if the bottleneck is due to object data transfer, increase the size of the vertex format. This can be done by sending several extra texture coordinates per vertex, for example. If performance falls, this area is the bottleneck.

Vertex processing is done by the vertex shader. For the vertex shader bottleneck, testing consists of making the shader program longer. Some care has to be taken to make sure the compiler is not optimizing away these additional instructions.

If your pipeline also uses geometry shaders, their performance is a function of output size and program length. If you are using tessellation shaders, again program length affects performance, as well as the tessellation factor. Varying any of these elements, while avoiding changes in the work other stages perform, can help determine whether any are the bottleneck.

18.2.3 Testing the Rasterization Stage

This stage consists of triangle setup and triangle traversal. Shadow map generation, which uses extremely simple pixel shaders, can bottleneck in the rasterizer or merging stages. Though normally rare [1961], it is possible for triangle setup and rasterization to be the bottleneck for small triangles from tessellation or objects such as grass or foliage. However, small triangles can also increase the use of both vertex and pixel shaders. More vertices in a given area clearly increases vertex shader load. Pixel shader load also increases because each triangle is rasterized by a set of 2×2 quads, so the number of pixels outside of each triangle increases [59]. This is sometimes called *quad overshading* (Section 23.1). To find if rasterization is truly the bottleneck, increase the execution time of both the vertex and pixel shaders by increasing their program sizes. If the render time per frame does not increase, then the bottleneck is in the rasterization stage.

18.2.4 Testing the Pixel Processing Stage

The pixel shader program's effect can be tested by changing the screen resolution. If a lower screen resolution causes the frame rate to rise appreciably, the pixel shader is likely to be the bottleneck, at least some of the time. Care has to be taken if a level of detail system is in place. A smaller screen is likely to also simplify the models displayed, lessening the load on the geometry stage.

Lowering the display resolution can also affect costs from triangle traversal, depth testing and blending, and texture access, among other factors. To avoid these factors and isolate the bottleneck, one approach is the same as that taken with vertex shader programs, to add more instructions to see the effect on execution speed. Again, it is important to determine that these additional instructions are not optimized away by the compiler. If frame rendering time increases, the pixel shader is the bottleneck (or at least has become the bottleneck at some point as its execution cost increased). Alternately, the pixel shader could be simplified to a minimum number of instructions, something often difficult to do in a vertex shader. If overall rendering time decreases, a bottleneck has been found. Texture cache misses can also be costly. If replacing a texture with a 1×1 resolution version gives considerably faster performance, then texture memory access is a bottleneck.

Shaders are separate programs that have their own optimization techniques. Persson [1383, 1385] presents several low-level shader optimizations, as well as specifics about how graphics hardware has evolved and how best practices have changed.

18.2.5 Testing the Merging Stage

In this stage depth and stencil tests are made, blending is done, and surviving results are written to buffers. Changing the output bit depth for these buffers is one way to vary the bandwidth costs for this stage and see if it could be the bottleneck. Turning alpha blending on for opaque objects or using other blending modes also affects the amount of memory access and processing performed by raster operations.

This stage can be the bottleneck with post-processing passes, shadows, particle system rendering, and, to a lesser extent, rendering hair and grass, where the vertex and pixel shaders are simple and so do little work.

18.3 Performance Measurements

To optimize we need to measure. Here we discuss different measures of GPU speed. Graphics hardware manufacturers used to present peak rates such as *vertices per second* and *pixels per second*, which were at best hard to reach. Also, since we are dealing with a pipelined system, true performance is not as simple as listing these kinds of numbers. This is because the location of the bottleneck may move from one time to another, and the different pipeline stages interact in different ways during execution.

Because of this complexity, GPUs are marketed in part on their physical properties, such as the number and clock rate of cores, memory size, speed, and bandwidth.

All that said, GPU counters and thread traces, if available, are important diagnostic tools when used well. If the peak performance of some given part is known and the count is lower, then this area is unlikely to be the bottleneck. Some vendors present counter data as a utilization percentage for each stage. These values are over a given time period during which the bottleneck can move, and so are not perfect, but help considerably in finding the bottleneck.

More is better, but even seemingly simple physical measurements can be difficult to compare precisely. For example, the clock rate for the same GPU can vary among IHV partners, as each has its own cooling solution and so overclocks its GPUs to what it considers safe. Even FPS benchmark comparisons on a single system are not always as simple as they sound. NVIDIA's *GPU Boost* [1666] and AMD's *PowerTune* [31] technology are good examples of our dictum "know your architecture." NVIDIA's GPU Boost arose in part because some synthetic benchmarks worked many parts of the GPU's pipeline simultaneously and so pushed power usage to the limit, meaning that NVIDIA had to lower its base clock rate to keep the chip from overheating. Many applications do not exercise all parts of the pipeline to such an extent, so can safely be run at a higher clock rate. The GPU Boost technology tracks GPU power and temperature characteristics and adjusts the clock rate accordingly. AMD and Intel have similar power/performance optimizations with their GPUs. This variability can cause the same benchmark to run at different speeds, depending on the initial temperature of the GPU. To avoid this problem, Microsoft provides a method in DirectX 12 to lock the GPU core clock frequency in order to get stable timings [121]. Examining power states is possible for other APIs, but is more complex [354].

When it comes to measuring performance for CPUs, the trend has been to avoid IPS (*instructions per second*), FLOPS (*floating point operations per second*), gigahertz, and simple short benchmarks. Instead, the preferred method is to measure wall clock times for a range of different, real programs [715], and then compare the running times for these. Following this trend, most independent graphics benchmarks measure the actual frame rate in FPS for several given scenes, and for a variety of different screen resolutions, along with antialiasing and quality settings. Many graphics-heavy games include a benchmarking mode or have one created by a third party, and these benchmarks are commonly used in comparing GPUs.

While FPS is useful shorthand for comparing GPUs running benchmarks, it should be avoided when analyzing a series of frame rates. The problem with FPS is that it is a reciprocal measure, not linear, and so can lead to analysis errors. For example, imagine you find the frame rate of your application at different times is 50, 50, and 20 FPS. If you average these values you get 40 FPS. That value is misleading at best. These frame rates translate to 20, 20, and 50 milliseconds, so the average frame time is 30 ms, which is 33.3 FPS. Similarly, milliseconds are pretty much required when measuring the performance of individual algorithms. For a specific benchmarking situation with

a given test and a given machine, it is possible to say that some particular shadow algorithm or post-process effect "costs" 7 FPS, and that the benchmark ran this much slower. However, it is meaningless to generalize this statement, since this value also depends on how much time it takes to process everything else in the frame and because you cannot add together the FPS of different techniques (but you can add times) [1378].

To be able to see the potential effects of pipeline optimization, it is important to measure the total rendering time per frame with double buffering disabled, i.e., in single-buffer mode by turning vertical synchronization off. This is because with double buffering turned on, swapping of the buffers occurs only in synchronization with the frequency of the monitor, as explained in the example in Section 2.1. De Smedt [331] discusses analyzing frame times to find and fix frame stutter problems from spikes in the CPU workload, as well as other useful tips for optimizing performance. Using statistical analysis is usually necessary. It is also possible to use GPU timestamps to learn what is happening within a frame [1167, 1422].

Raw speed is important, but for mobile devices another goal is optimizing power consumption. Purposely lowering the frame rate but keeping the application interactive can significantly extend battery life and have little effect on the user's experience [1200]. Akenine-Möller and Johnsson [25, 840] note that performance per watt is like frames per second, with the same drawbacks as FPS. They argue a more useful measure is joules per task, e.g., joules per pixel.

18.4 Optimization

Once a bottleneck has been located, we want to optimize that stage to boost the performance. In this section we present optimization techniques for the application, geometry, rasterization, and pixel processing stages.

18.4.1 Application Stage

The application stage is optimized by making the code faster and the memory accesses of the program faster or fewer. Here we touch upon some of the key elements of code optimization that apply to CPUs in general.

For code optimization, it is crucial to locate the place in the code where most of the time is spent. A good code profiler is critical in finding these code hot spots, where most time is spent. Optimization efforts are then made in these places. Such locations in the program are often *inner loops*, pieces of the code that are executed many times each frame.

The basic rule of optimization is to try a variety of tactics: Reexamine algorithms, assumptions, and code syntax, trying variants as possible. CPU architecture and compiler performance often limit the user's ability to form an intuition about how to write the fastest code, so question your assumptions and keep an open mind.

One of the first steps is to experiment with the optimization flags for the compiler. There are usually a number of different flags to try. Make few, if any, assumptions about what optimization options to use. For example, setting the compiler to use more aggressive loop optimizations could result in slower code. Also, if possible, try different compilers, as these are optimized in different ways, and some are markedly superior. Your profiler can tell you what effect any change has.

Memory Issues

Years ago the number of arithmetic instructions was the key measure of an algorithm's efficiency; now the key is memory access patterns. Processor speed has increased much more rapidly than the data transfer rate for DRAM, which is limited by the pin count. Between 1980 and 2005, CPU performance doubled about every two years, and DRAM performance doubled about every six [1060]. This problem is known as the *Von Neumann bottleneck* or the *memory wall*. *Data-oriented design* focuses on cache coherency as a means of optimization.[1]

On modern GPUs, what matters is the distance traveled by data. Speed and power costs are proportional to this distance. Cache access patterns can make up to an orders-of-magnitude performance difference [1206]. A cache is a small fast-memory area that exists because there is usually much coherence in a program, which the cache can exploit. That is, nearby locations in memory tend to be accessed one after another (spatial locality), and code is often accessed sequentially. Also, memory locations tend to be accessed repeatedly (temporal locality), which the cache also exploits [389]. Processor caches are fast to access, second only to registers for speed. Many fast algorithms work to access data as locally (and as little) as possible.

Registers and local caches form one end of the *memory hierarchy*, which extends next to dynamic random access memory (DRAM), then to storage on SSDs and hard disks. At the top are small amounts of fast, expensive memory, at the bottom are large amounts of slow and inexpensive storage. Between each level of the hierarchy the speed drops by some noticeable factor. See Figure 18.1. For example, processor registers are usually accessed in one clock cycle, while L1 cache memory is accessed in a few cycles. Each change in level has an increase in latency in this way. As discussed in Section 3.10, sometimes latency can be hidden by the architecture, but it is always a factor that must be kept in mind.

Bad memory access patterns are difficult to directly detect in a profiler. Good patterns need to be built into the design from the start [1060]. Below is a list of pointers that should be kept in consideration when programming.

- Data that is accessed sequentially in the code should also be stored sequentially in memory. For example, when rendering a triangle mesh, store texture coordinate #0, normal #0, color #0, vertex #0, texture coordinate #1, and normal #1, sequentially in memory if they are accessed in that order. This can also be

[1] This area of study should not be confused with *data-driven design*, which can mean any number of things, from the AWK programming language to A/B testing.

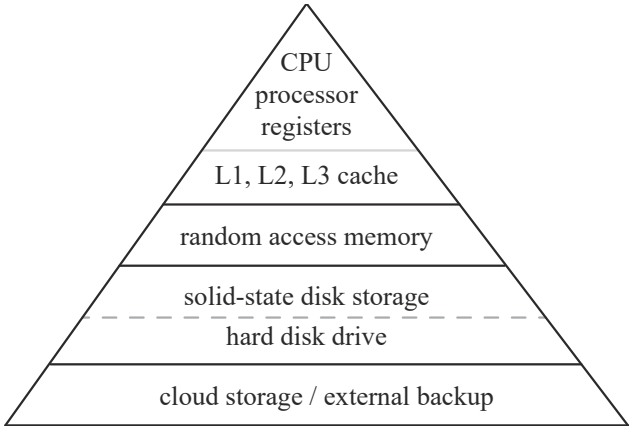

Figure 18.1. The memory hierarchy. Speed and cost decrease as we descend the pyramid.

important on the GPU, as with the post-transform vertex cache (Section 16.4.4). Also see Section 16.4.5 for why storing separate streams of data can be beneficial.

- Avoid pointer indirection, jumps, and function calls (in critical parts of the code), as these may significantly decrease CPU performance. You get pointer indirection when you follow a pointer to another pointer and so on. Modern CPUs try to speculatively execute instructions (branch prediction) and fetch memory (cache prefetching) to keep all their functional units busy running code. These techniques are highly effective when the code flow is consistent in a loop, but fail with branching data structures such as binary trees, linked lists, and graphs; use arrays instead, as possible. McVoy and Staelin [1194] show a code example that follows a linked list through pointers. This causes cache misses for data both before and after, and their example stalls the CPU more than 100 times longer than it takes to follow the pointer (if the cache could provide the address of the pointer). Smits [1668] notes how flattening a pointer-based tree into a list with skip pointers considerably improves hierarchy traversal. Using a van Emde Boas layout is another way to help avoid cache misses—see Section 19.1.4. High-branching trees are often preferable to binary trees because they reduce the tree depth and so reduce the amount of indirection.

- Aligning frequently used data structures to multiples of the cache line size can significantly improve overall performance. For example, 64 byte cache lines are common on Intel and AMD processors [1206]. Compiler options can help, but it is wise to design your data structures with alignment, called *padding*, in mind. Tools such as *VTune* and *CodeAnalyst* for Windows and Linux, *Instruments* for the Mac, and the open-source *Valgrind* for Linux can help identify caching bottlenecks. Alignment can also affect GPU shader performance [331].

- Try different organizations of data structures. For example, Hecker [698] shows how a surprisingly large amount of time was saved by testing a variety of matrix structures for a simple matrix multiplier. An array of structures,

```
struct Vertex {float x,y,z;};
Vertex myvertices[1000];
```

  or a structure of arrays,

```
struct VertexChunk {float x[1000],y[1000],z[1000];};
VertexChunk myvertices;
```

  may work better for a given architecture. This second structure is better for using SIMD commands, but as the number of vertices goes up, the chance of a cache miss increases. As the array size increases, a hybrid scheme,

```
struct Vertex4 {float x[4],y[4],z[4];};
Vertex4 myvertices[250];
```

  may be the best choice.

- It is often better to allocate a large pool of memory at start-up for objects of the same size, and then use your own allocation and free routines for handling the memory of that pool [113, 736]. Libraries such as *Boost* provide pool allocation. A set of contiguous records is more likely to be cache coherent than those created by separate allocations. That said, for languages with garbage collection, such as C# and Java, pools can actually reduce performance.

While not directly related to memory access patterns, it is worthwhile to avoid allocating or freeing memory within the rendering loop. Use pools and allocate scratch space once, and have stacks, arrays, and other structures only grow (using a variable or flags to note which elements should be treated as deleted).

18.4.2 API Calls

Throughout this book we have given advice based on general trends in hardware. For example, indexed vertex buffers objects are usually the fastest way to provide the accelerator with geometric data (Section 16.4.5). This section is about how to best call the graphics API itself. Most graphics APIs have similar architectures, and there are well-established ways of using them efficiently.

Understanding object buffer allocation and storage is basic to efficient rendering [1679]. For a desktop system with a CPU and a separate, discrete GPU, each normally has its own memory. The graphics driver is usually in control of where objects reside, but it can be given hints of where best to store them. A common classification is static versus dynamic buffers. If the buffer's data are changing each frame, using a dynamic buffer, which requires no permanent storage space on the GPU, is preferable. Consoles, laptops with low-power integrated GPUs, and mobile

devices usually have unified memory, where the GPU and CPU share the same physical memory. Even in these setups, allocating a resource in the right pool matters. Correctly tagging a resource as CPU-only or GPU-only can still yield benefits. In general, if a memory area has to be accessed by both chips, when one writes to it the other has to invalidate its caches—an expensive operation—to be sure not to get stale data.

If an object is not deforming, or the deformations can be carried out entirely by shader programs (e.g., skinning), then it is profitable to store the data for the object in GPU memory. The unchanging nature of this object can be signaled by storing it as a static buffer. In this way, it does not have to be sent across the bus for every frame rendered, thus avoiding any bottleneck at this stage of the pipeline. The internal memory bandwidth on a GPU is normally much higher than the bus between CPU and GPU.

State Changes

Calling the API has several costs associated with it. On the application side, more calls mean more application time spent, regardless of what the calls actually do. This cost can be minimal or noticeable, and a null driver can help identify it. Query functions that depend on values from the GPU can potentially halve the frame rate due to stalls from synchronization with the CPU [1167]. Here we will delve into optimizing a common graphics operation, preparing the pipeline to draw a mesh. This operation may involve changing the state, e.g., setting the shaders and their uniforms, attaching textures, changing the blend state or the color buffer used, and so on.

A major way for the application to improve performance is to minimize state changes by grouping objects with a similar rendering state. Because the GPU is an extremely complex state machine, perhaps the most complex in computer science, changing the state can be expensive. While a little of the cost can involve the GPU, most of the expense is from the driver's execution on the CPU. If the GPU maps well to the API, the state change cost tends to be predictable, though still significant. If the GPU has a tight power constraint or limited silicon footprint, such as with some mobile devices, or has a hardware bug to work around, the driver may have to perform heroics that cause unexpectedly high costs. State change costs are mostly on the CPU side, in the driver.

One concrete example is how the PowerVR architecture supports blending. In older APIs blending is specified using a fixed-function type of interface. PowerVR's blending is programmable, which means that their driver has to patch the current blend state into the pixel shader [699]. In this case a more advanced design does not map well to the API and so incurs a significant setup cost in the driver. While throughout this chapter we note that hardware architecture and the software running it can affect the importance of various optimizations, this is particularly true for state change costs. Even the specific GPU type and driver release may have an effect. While reading, please imagine the phrase "your mileage may vary" stamped in large red letters over every page of this section.

Everitt and McDonald [451] note that different types of state changes vary considerably in cost, and give some rough idea as to how many times a second a few could be performed on an NVIDIA OpenGL driver. Here is their order, from most expensive to least, as of 2014:

- Render target (framebuffer object), ∼60k/sec.
- Shader program, ∼300k/sec.
- Blend mode (ROP), such as for transparency.
- Texture bindings, ∼1.5M/sec.
- Vertex format.
- Uniform buffer object (UBO) bindings.
- Vertex bindings.
- Uniform updates, ∼10M/sec.

This approximate cost order is borne out by others [488, 511, 741]. One even more expensive change is switching between the GPU's rendering mode and its compute shader mode [1971]. Avoiding state changes can be achieved by sorting the objects to be displayed by grouping them by shader, then by textures used, and so on down the cost order. Sorting by state is sometimes called *batching*.

Another strategy is to restructure the objects' data so that more sharing occurs. A common way to minimize texture binding changes is to put several texture images into one large texture or, better yet, a texture array. If the API supports it, bindless textures are another option to avoid state changes (Section 6.2.5). Changing the shader program is usually relatively expensive compared to updating uniforms, so variations within a class of materials may be better represented by a single shader that uses "if" statements. You might also be able to make larger batches by sharing a shader [1609]. Making shaders more complex can also lower performance on the GPU, however. Measuring to see what is effective is the only foolproof way to know.

Making fewer, more effective calls to the graphics API can yield some additional savings. For example, often several uniforms can be defined and set as a group, so binding a single uniform buffer object is considerably more efficient [944]. In DirectX these are called *constant buffers*. Using these properly saves both time per function and time spent error-checking inside each individual API call [331, 613].

Modern drivers often defer setting state until the first draw call encountered. If redundant API calls are made before then, the driver will filter these out, thus avoiding the need to perform a state change. Often a dirty flag is used to note that a state change is needed, so going back to a base state after each draw call may become costly. For example, you may want to assume state X is off by default when you are about

to draw an object. One way to achieve this is "Enable(X); Draw(M_1); Disable(X);" then "Enable(X); Draw(M_2); Disable(X);" thus restoring the state after each draw. However, it is also likely to waste significant time setting the state again between the two draw calls, even though no actual state change occurs between them.

Usually the application has higher-level knowledge of when a state change is needed. For example, changing from a "replace" blending mode for opaque surfaces to an "over" mode for transparent ones normally needs to be done once during the frame. Issuing the blend mode before rendering each object can easily be avoided. Galeano [511] shows how ignoring such filtering and issuing unneeded state calls would have cost their WebGL application up to nearly 2 ms/frame. However, if the driver already does such redundancy filtering efficiently, performing this same testing per call in the application can be a waste. How much effort to spend filtering out API calls primarily depends on the underlying driver [443, 488, 741].

Consolidating and Instancing

Using the API efficiently avoids having the CPU be the bottleneck. One other concern with the API is the small batch problem. If ignored, this can be a significant factor affecting performance in modern APIs. Simply put, a few triangle-filled meshes are much more efficient to render than many small, simple ones. This is because there is a fixed-cost overhead associated with each draw call, a cost paid for processing a primitive, regardless of size.

Back in 2003, Wloka [1897] showed that drawing two (relatively small) triangles per batch was a factor of 375 away from the maximum throughput for the GPU tested.[2] Instead of 150 million triangles per second, the rate was 0.4 million, for a 2.7 GHz CPU. For a scene rendered consisting of many small and simple objects, each with only a few triangles, performance is entirely CPU-bound by the API; the GPU has no ability to increase it. That is, the processing time on the CPU for the draw call is greater than the amount of time the GPU takes to actually draw the mesh, so the GPU is starved.

Wloka's rule of thumb is that "You get X batches per frame." This is a maximum number of draw calls you can make per frame, purely due to the CPU being the limiting factor. In 2003, the breakpoint where the API was the bottleneck was about 130 triangles per object. Figure 18.2 shows how the breakpoint rose in 2006 to 510 triangles per mesh. Times have changed. Much work was done to ameliorate this draw call problem, and CPUs became faster. The recommendation back in 2003 was 300 draw calls per frame; in 2012, 16,000 draw calls per frame was one team's ceiling [1381]. That said, even this number is not enough for some complicated scenes. With modern APIs such as DirectX 12, Vulkan, and Metal, the driver cost may itself be minimized—this is one of their major advantages [946]. However, the GPU can have its own fixed costs per mesh.

[2] Wloka used *batch* to mean a single mesh rendered with a draw call. This term has widened out over the years, now sometimes meaning a group of separate objects to be rendered that have the same state, as the API overhead can then be reduced.

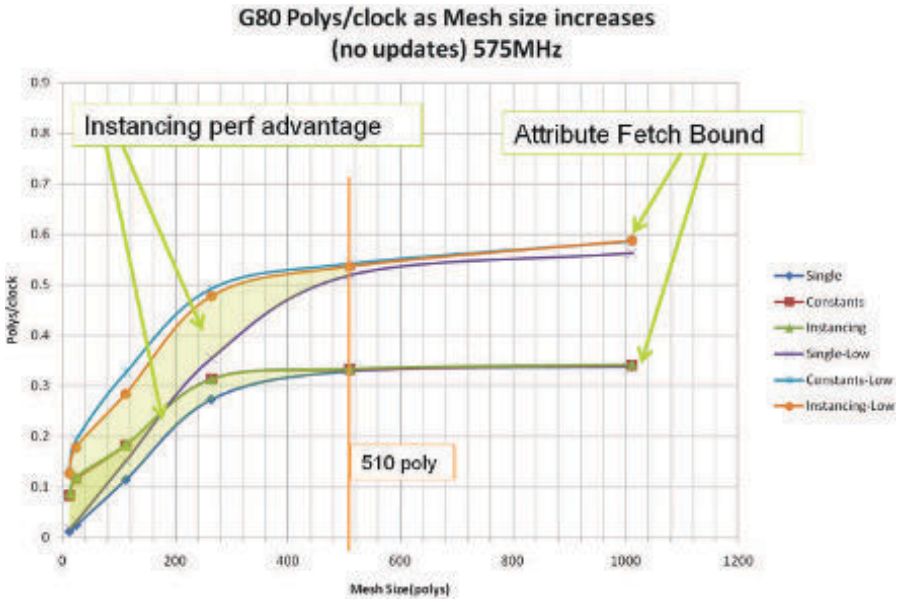

Figure 18.2. Batch performance benchmarks for an Intel Core 2 Duo 2.66 GHz CPU using an NVIDIA G80 GPU, running DirectX 10. Batches of varying size were run and timed under different conditions. The "Low" conditions are for triangles with just the position and a constant-color pixel shader; the other set of tests is for reasonable meshes and shading. "Single" is rendering a single batch many times. "Instancing" reuses the mesh data and puts the per-instance data in a separate stream. "Constants" is a DirectX 10 method where instance data are put in constant memory. As can be seen, small batches hurt all methods, but instancing gives proportionally much faster performance. At a few hundred triangles, performance levels out, as the bottleneck becomes how fast vertices are retrieved from the vertex buffer and caches. *(Graph courtesy of NVIDIA Corporation.)*

One way to reduce the number of draw calls is to consolidate several objects into a single mesh, which needs only one draw call to render the set. For sets of objects that use the same state and are static, at least with respect to one another, consolidation can be done once and the batch can be reused each frame [741, 1322]. Being able to consolidate meshes is another reason to consider avoiding state changes by using a common shader and texture-sharing techniques. The cost savings from consolidation are not just from avoiding API draw calls. There are also savings from the application itself handling fewer objects. However, having batches that are considerably larger than needed can make other algorithms, such as frustum culling, be less effective [1381]. One practice is to use a bounding volume hierarchy to help find and group static objects that are near each other. Another concern with consolidation is selection, since all the static objects are undifferentiated, in one mesh. A typical solution is to store an object identifier at each vertex in the mesh.

The other approach to minimize application processing and API costs is to use some form of *instancing* [232, 741, 1382]. Most APIs support the idea of having an

object and drawing it several times in a single call. This is typically done by specifying a base model and providing a separate data structure that holds information about each specific instance desired. Beyond position and orientation, other attributes could be specified per instance, such as leaf colors or curvature due to the wind, or anything else that could be used by shader programs to affect the model. Lush jungle scenes can be created by liberal use of instancing. See Figure 18.3. Crowd scenes are a good fit for instancing, with each character appearing unique by selecting different body parts from a set of choices. Further variation can be added by random coloring and decals. Instancing can also be combined with level of detail techniques [122, 1107, 1108]. See Figure 18.4 for an example.

A concept that combines consolidation and instancing is called *merge-instancing*, where a consolidated mesh contains objects that may in turn be instanced [146, 1382].

In theory, the geometry shader can be used for instancing, as it can create duplicate data of an incoming mesh. In practice, if many instances are needed, this method can be slower than using instancing API commands. The intent of the geometry shader is to perform local, small-scale amplification of data [1827]. In addition, for some architectures, such as Mali's tile-based renderer, the geometry shader is implemented in software. To quote Mali's best practices guide [69], "Find a better solution to your problem. Geometry shaders are not your solution."

18.4.3 Geometry Stage

The geometry stage is responsible for transforms, per-vertex lighting, clipping, projection, and screen mapping. Other chapters discuss ways to reduce the amount of data flowing through the pipeline. Efficient triangle mesh storage, model simplification, and vertex data compression (Chapter 16) all save both processing time and memory. Techniques such as frustum and occlusion culling (Chapter 19) avoid sending the full primitive itself down the pipeline. Adding such large-scale techniques on the CPU can entirely change performance characteristics of the application and so are worth trying early on in development. On the GPU such techniques are less common. One notable example is how the compute shader can be used to perform various types of culling [1883, 1884].

The effect of elements of lighting can be computed per vertex, per pixel (in the pixel processing stage), or both. Lighting computations can be optimized in several ways. First, the types of light sources being used should be considered. Is lighting needed for all triangles? Sometimes a model only requires texturing, texturing with colors at the vertices, or simply colors at the vertices.

If light sources are static with respect to geometry, then the diffuse and ambient lighting can be precomputed and stored as colors at the vertices. Doing so is often referred to as "baking" on the lighting. A more elaborate form of prelighting is to precompute the diffuse global illumination in a scene (Section 11.5.1). Such illumination can be stored as colors or intensities at the vertices or as light maps.

Figure 18.3. Vegetation instancing. All objects the same color in the lower image are rendered in a single draw call [1869]. *(Image from CryEngine1, courtesy of Crytek.)*

Figure 18.4. Crowd scene. Using instancing minimizes the number of draw calls needed. Level of detail techniques are also used, such as rendering impostors for distant models [1107, 1108]. *(Image courtesy of Jonathan Maïm, Barbara Yersin, Mireille Clavien, and Daniel Thalmann.)*

For forward rendering systems the number of light sources influences the performance of the geometry stage. More light sources means more computation. A common way to lessen work is to disable or trim down local lighting and instead use an environment map (Section 10.5).

18.4.4 Rasterization Stage

Rasterization can be optimized in a few ways. For closed (solid) objects and for objects that will never show their backfaces (for example, the back side of a wall in a room), backface culling should be turned on (Section 19.3). This reduces the number of triangles to be rasterized by about half and so reduces the load on triangle traversal. In addition, this can be particularly beneficial when the pixel shading computation is expensive, as backfaces are then never shaded.

18.4.5 Pixel Processing Stage

Optimizing pixel processing is often profitable, since usually there are many more pixels to shade than vertices. There are notable exceptions. Vertices always have to be processed, even if a draw ends up not generating any visible pixels. Ineffective culling in the rendering engine might make the vertex shading cost exceed pixel shading. Too small a triangle not only causes more vertex shading evaluation than may be

needed, but also can create more partial-covered quads that cause additional work. More important, textured meshes that cover only a few pixels often have low thread occupancy rates. As discussed in Section 3.10, there is a large time cost in sampling a texture, which the GPU hides by switching to execute shader programs on other fragments, returning later when the texture data has been fetched. Low occupancy can result in poor latency hiding. Complex shaders that use a high number of registers can also lead to low occupancy by allowing fewer threads to be available at one time (Section 23.3). This condition is referred to as high *register pressure*. There are other subtleties, e.g., frequent switching to other warps may cause more cache misses. Wronski [1911, 1914] discusses various occupancy problems and solutions.

To begin, use native texture and pixel formats, i.e., use the formats that the graphics accelerator uses internally, to avoid a possible expensive transform from one format to another [278]. Two other texture-related techniques are loading only the mipmap levels needed (Section 19.10.1) and using texture compression (Section 6.2.6). As usual, smaller and fewer textures mean less memory used, which in turn means lower transfer and access times. Texture compression also can improve cache performance, since the same amount of cache memory is occupied by more pixels.

One level of detail technique is to use different pixel shader programs, depending on the distance of the object from the viewer. For example, with three flying saucer models in a scene, the closest might have an elaborate bump map for surface details that the two farther away do not need. In addition, the farthest saucer might have specular highlighting simplified or removed altogether, both to simplify computations and to reduce "fireflies," i.e., sparkle artifacts from undersampling. Using a color per vertex on simplified models can give the additional benefit that no state change is needed due to the texture changing.

The pixel shader is invoked only if the fragment is visible at the time the triangle is rasterized. The GPU's early-z test (Section 23.7) checks the z-depth of the fragment against the z-buffer. If not visible, the fragment is discarded without any pixel shader evaluation, saving considerable time. While the z-depth can be modified by the pixel shader, doing so means that early-z testing cannot be performed.

To understand the behavior of a program, and especially the load on the pixel processing stage, it is useful to visualize the depth complexity, which is the number of surfaces that cover a pixel. Figure 18.5 shows an example. One simple method of generating a depth complexity image is to use a call like OpenGL's `glBlendFunc(GL_ONE,GL_ONE)`, with z-buffering disabled. First, the image is cleared to black. All objects in the scene are then rendered with the color $(1/255, 1/255, 1/255)$. The effect of the blend function setting is that for each primitive rendered, the values of the written pixels will increase by one intensity level. A pixel with a depth complexity of 0 is then black and a pixel of depth complexity 255 is full white, $(255, 255, 255)$.

The amount of *pixel overdraw* is related to how many surfaces actually were rendered. The number of times the pixel shader is evaluated can be found by rendering the scene again, but with the z-buffer enabled. Overdraw is the amount of effort

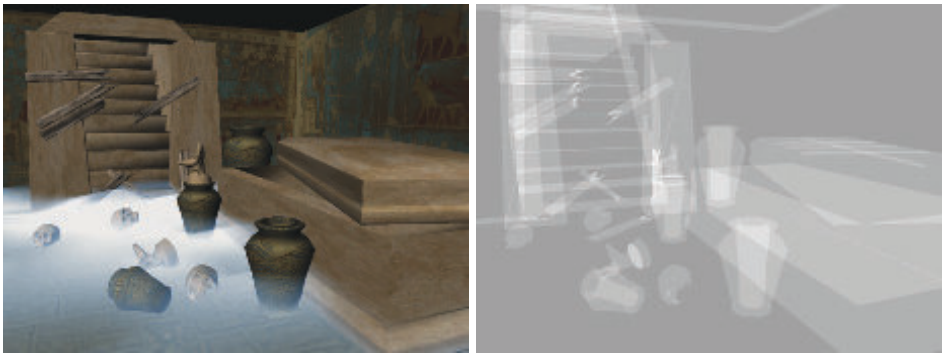

Figure 18.5. The depth complexity of the scene on the left is shown on the right. *(Images created using NVPerfHUD from NVIDIA Corporation.)*

wasted computing a shade for a surface that is then hidden by a later pixel shader invocation. An advantage of deferred rendering (Section 20.1), and ray tracing for that matter, is that shading is performed after all visibility computations are performed.

Say two triangles cover a pixel, so the depth complexity is two. If the farther triangle is drawn first, the nearer triangle overdraws it, and the amount of overdraw is one. If the nearer is drawn first, the farther triangle fails the depth test and is not drawn, so there is no overdraw. For a random set of opaque triangles covering a pixel, the average number of draws is the *harmonic series* [296]:

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}. \tag{18.1}$$

The logic behind this is that the first triangle rendered is one draw. The second triangle is either in front or behind the first, a 50/50 chance. The third triangle can have one of three positions compared to the first two, giving one chance in three of it being frontmost. As n goes to infinity,

$$\lim_{n\to\infty} H(n) = \ln(n) + \gamma, \tag{18.2}$$

where $\gamma = 0.57721\ldots$ is the Euler-Mascheroni constant. Overdraw rises rapidly when depth complexity is low, but quickly tapers off. For example, a depth complexity of 4 gives an average of 2.08 draws, 11 gives 3.02 draws, but it takes a depth complexity of 12,367 to reach an average of 10.00 draws.

So, overdraw is not necessarily as bad as it seems, but we would still like to minimize it, without costing too much CPU time. Roughly sorting and then drawing the opaque objects in a scene in an approximate front-to-back order (near to far) is a common way to reduce overdraw [240, 443, 488, 511]. Occluded objects that are drawn later will not write to the color or z-buffers (i.e., overdraw is reduced). Also, the pixel fragment can be rejected by occlusion culling hardware before even reaching

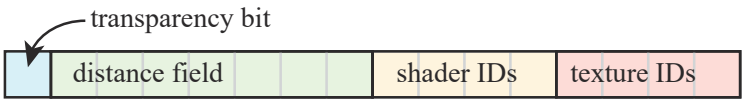

Figure 18.6. Example sort key for draw order. Keys are sorted from low to high. Setting the transparency bit means that the object is transparent, as transparent objects are to be rendered after all opaque objects. The object's distance from the camera is stored as an integer with low precision. For transparent objects the distance is reversed or negated, since we want objects in a back-to-front order. Shaders are each given a unique identification number, as are textures.

the pixel shader program (Section 23.5). Sorting can be accomplished by any number of methods. An explicit sort based on the distance along the view direction of the centroids of all opaque objects is one simple technique. If a bounding volume hierarchy or other spatial structure is already in use for frustum culling, we can choose the closer child to be traversed first, on down the hierarchy.

Another technique can be useful for surfaces with complex pixel shader programs. Performing a z-prepass renders the geometry to just the z-buffer first, then the whole scene is rendered normally [643]. This eliminates all overdraw shader evaluations, but at the cost of an entire separate run through all the geometry. Pettineo [1405] writes that the primary reason his team used a depth prepass in their video game was to avoid overdraw. However, drawing in a rough front-to-back order may provide much of the same benefit without the need for this extra work. A hybrid approach is to identify and first draw just a few large, simple occluders likely to give the most benefit [1768]. As McGuire [1177] notes, a full-draw prepass did not help performance for his particular system. Measuring is the only way to know which technique, if any, is most effective for your application.

Earlier we recommended grouping by shader and texture to minimize state changes; here we talk about rendering objects sorted by distance. These two goals usually give different object draw orders and so conflict with each other. There is always some ideal draw order for a given scene and viewpoint, but this is difficult to find in advance. Hybrid schemes are possible, e.g., sorting nearby objects by depth and sorting everything else by material [1433]. A common, flexible solution [438, 488, 511, 1434, 1882] is to create a sorting key for each object that encapsulates all the relevant criteria by assigning each a set of bits. See Figure 18.6.

We can choose to favor sorting by distance, but by limiting the number of bits storing the depth, we can allow grouping by shader to become relevant for objects in a given range of distances. It is not uncommon to sort draws into even as few as two or three depth partitions. If some objects have the same depth and use the same shader, then the texture identifier is used to sort the objects, which then groups objects with the same texture together.

This is a simple example and is situational, e.g., the rendering engine may itself keep opaque and transparent objects separate so that the transparency bit is not necessary. The number of bits for the other fields certainly varies with the maximum

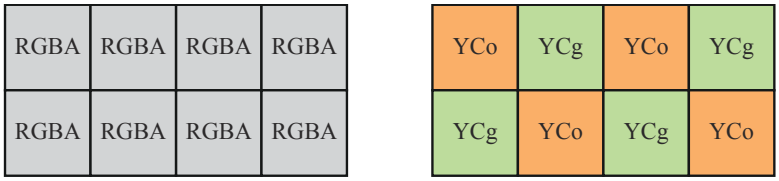

Figure 18.7. Left: 4×2 pixels, each storing four color components (RGBA). Right: an alternative representation where each pixel stores the luminance, Y, and either the first (C_o) or the second (C_g) chrominance component, in a checkerboard pattern.

number of shaders and textures expected. Other fields may be added or substituted in, such as one for blend state and another for z-buffer read and write. Most important of all is the architecture. For example, some tile-based GPU renderers on mobile devices do not gain anything from sorting front to back, so state sorting is the only important element to optimize [1609]. The main idea here is that putting all attributes into a single integer key lets you perform an efficient sort, thus minimizing overdraw and state changes as possible.

18.4.6 Framebuffer Techniques

Rendering a scene often incurs a vast amount of accesses to the framebuffer and many pixel shader executions. To reduce the pressure on the cache hierarchy, a common piece of advice is to reduce the storage size of each pixel of the framebuffer. While a 16-bit floating point value per color channel provides more accuracy, an 8-bit value is half the size, which means faster accesses assuming that the accuracy is sufficient. The chrominance is often subsampled in many image and video compression schemes, such as JPEG and MPEG. This can often be done with negligible visual effect due to fact that the human visual system is more sensitive to luminance than to chrominance. For example, the Frostbite game engine [1877] uses this idea of *chroma subsampling* to reduce bandwidth costs for post-processing their 16-bits-per-channel images.

Mavridis and Papaioannou [1144] propose that the lossy YCoCg transform, described on page 197, is used to achieve a similar effect for the color buffer during rasterization. Their pixel layout is shown in Figure 18.7. Compared to RGBA, this halves the color buffer storage requirements (assuming A is not needed) and often increases performance, depending on architecture. Since each pixel has only one of the chrominance components, a reconstruction filter is needed to infer a full YCoCg per pixel before converting back to RGB before display. For a pixel missing the C_o-value, for example, the average of the four closest C_o-values can be used. However, this does not reconstruct edges as well as desired. Therefore, a simple edge-aware filter is used instead, which is implemented as

$$C_o = \sum_{i=0}^{3} w_i C_{o,i}, \quad \text{where } w_i = 1.0 - \text{step}(t - |L_i - L|), \tag{18.3}$$

for a pixel that does not have C_o, where $C_{o,i}$ and L_i are the values to the left, right, top, and bottom of the current pixel, L is the luminance of the current pixel, and t is a threshold value for edge detection. Mavridis and Papaioannou used $t = 30/255$. The step(x) function is 0 if $x < 0$, and 1 otherwise. Hence, the filter weights w_i are either 0 or 1, where they are zero if the luminance gradient, $|L_i - L|$, is greater than t. A WebGL demo with source code is available online [1144].

Because of the continuing increase in display resolutions and the shader execution cost savings, using a checkerboard pattern for rendering has been used in several systems [231, 415, 836, 1885]. For virtual reality applications, Vlachos [1824] uses a checkerboard pattern for pixels around the periphery of the view, and Answer [59] reduces each 2×2 quad by one to three samples.

18.4.7 Merging Stage

Make sure to enable blend modes only when useful. In theory "over" compositing could be set for every triangle, opaque or transparent, since opaque surfaces using "over" will fully overwrite the value in the pixel. However, this is more costly than a simple "replace" raster operation, so tracking objects with cutout texturing and materials with transparency is worthwhile. Alternately, there are some raster operations that cost nothing extra. For example, when the z-buffer is being used, on some systems it costs no additional time to also access the stencil buffer. This is because the 8-bit stencil buffer value is stored in the same word as the 24-bit z-depth value [890].

Thinking through when various buffers need to be used or cleared is worthwhile. Since GPUs have fast clear mechanisms (Section 23.5), the recommendation is to always clear both color and depth buffers since that increases the efficiency of memory transfers for these buffers.

You should normally avoid reading back render targets from the GPU to the CPU if you can help it. Any framebuffer access by the CPU causes the entire GPU pipeline to be flushed before the rendering is returned, losing all parallelism there [1167, 1609].

If you do find that the merging stage is your bottleneck, you may need to rethink your approach. Can you use lower-precision output targets, perhaps through compression? Is there any way to reorder your algorithm to mitigate the stress on this stage? For shadows, are there ways to cache and reuse parts where nothing has moved?

In this section we have discussed ways of using each stage well by searching for bottlenecks and tuning performance. That said, be aware of the dangers of repeatedly optimizing an algorithm when you may be better served by using an entirely different technique.

18.5 Multiprocessing

Traditional APIs have evolved toward issuing fewer calls that each do more [443, 451]. The new generation of APIs—DirectX 12, Vulkan, Metal—take a different strategy.

For these APIs the drivers are streamlined and minimal, with much of the complexity and responsibility for validating the state shifted to the calling application, as well as memory allocation and other functions [249, 1438, 1826]. This redesign in good part was done to minimize draw call and state change overhead, which comes from having to map older APIs to modern GPUs. The other element these new APIs encourage is using multiple CPU processors to call the API.

Around 2003 the trend of ever-rising clock speeds for CPUs flattened out at around 3.4 GHz, due to several physical issues such as heat dissipation and power consumption [1725]. These limits gave rise to multiprocessing CPUs, where instead of higher clock rates, more CPUs were put in a single chip. In fact, many small cores provide the best performance per unit area [75], which is the major reason why GPUs themselves are so effective. Creating efficient and reliable programs that exploit concurrency has been the challenge ever since. In this section we will cover the basic concepts of efficient multiprocessing on CPU cores, at the end discussing how graphics APIs have evolved to enable more concurrency within the driver itself.

Multiprocessor computers can be broadly classified into *message-passing* architectures and *shared memory multiprocessors*. In message-passing designs, each processor has its own memory area, and messages are sent between the processors to communicate results. These are not common in real-time rendering. Shared memory multiprocessors are just as they sound; all processors share a logical address space of memory among themselves. Most popular multiprocessor systems use shared memory, and most of these have a *symmetric multiprocessing* (SMP) design. SMP means that all the processors are identical. A multicore PC system is an example of a symmetric multiprocessing architecture.

Here, we will present two general methods for using multiple processors for real-time graphics. The first method—*multiprocessor pipelining*, also called temporal parallelism—will be covered in more detail than the second—*parallel processing*, also called spatial parallelism. These two methods are illustrated in Figure 18.8. These two types of parallelism are then brought together with *task-based multiprocessing*, where the application creates jobs that can each be picked up and processed by an individual core.

18.5.1 Multiprocessor Pipelining

As we have seen, pipelining is a method for speeding up execution by dividing a job into certain pipeline stages that are executed in parallel. The result from one pipeline stage is passed on to the next. The ideal speedup is n times for n pipeline stages, and the slowest stage (the bottleneck) determines the actual speedup. Up to this point, we have seen pipelining used with a single CPU core and a GPU to run the application, geometry processing, rasterization, and pixel processing in parallel. Pipelining can also be used when multiple processors are available on the host, and in these cases, it is called *multiprocess pipelining* or *software pipelining*.

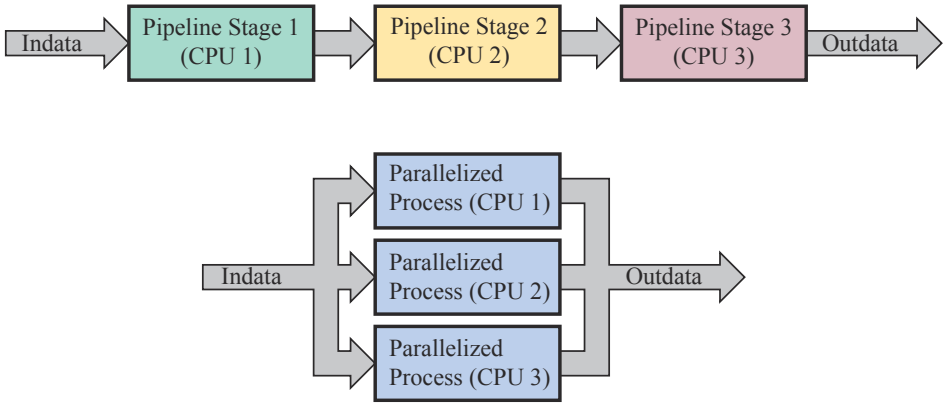

Figure 18.8. Two different ways of using multiple processors. At the top we show how three processors (CPUs) are used in a *multiprocessor pipeline*, and at the bottom we show *parallel* execution on three CPUs. One of the differences between these two implementations is that lower latency can be achieved if the configuration at the bottom is used. On the other hand, it may be easier to use a multiprocessor pipeline. The ideal speedup for both of these configurations is linear, i.e., using n CPUs would give a speedup of n times.

Here we describe one type of software pipelining. Endless variations are possible and the method should be adapted to the particular application. In this example, the application stage is divided into three stages [1508]: APP, CULL, and DRAW. This is coarse-grained pipelining, which means that each stage is relatively long. The APP stage is the first stage in the pipeline and therefore controls the others. It is in this stage that the application programmer can put in additional code that does, for example, collision detection. This stage also updates the viewpoint. The CULL stage can perform:

- Traversal and hierarchical view frustum culling on a scene graph (Section 19.4).
- Level of detail selection (Section 19.9).
- State sorting, as discussed in Section 18.4.5.
- Finally (and always performed), generation of a simple list of all objects that should be rendered.

The DRAW stage takes the list from the CULL stage and issues all graphics calls in this list. This means that it simply walks through the list and feeds the GPU. Figure 18.9 shows some examples of how this pipeline can be used.

If one processor core is available, then all three stages are run on that core. If two CPU cores are available, then APP and CULL can be executed on one core and DRAW on the other. Another configuration would be to execute APP on one core and CULL

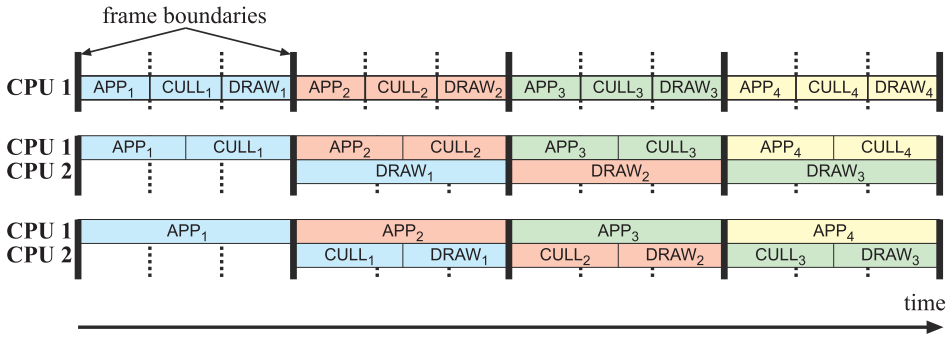

Figure 18.9. Different configurations for a multiprocessor pipeline. The thick lines represent synchronization between the stages, and the subscripts represent the frame number. At the top, a single CPU pipeline is shown. In the middle and at the bottom are shown two different pipeline subdivisions using two CPUs. The middle has one pipeline stage for APP and CULL and one pipeline stage for DRAW. This is a suitable subdivision if DRAW has much more work to do than the others. At the bottom, APP has one pipeline stage and the other two have another. This is suitable if APP has much more work than the others. Note that the bottom two configurations have more time for the APP, CULL, and DRAW stages.

and DRAW on the other. Which is the best depends on the workloads for the different stages. Finally, if the host has three cores available, then each stage can be executed on a separate core. This possibility is shown in Figure 18.10.

The advantage of this technique is that the throughput, i.e., the rendering speed, increases. The downside is that, compared to parallel processing, the latency is greater. Latency, or temporal delay, is the time it takes from the polling of the user's actions to the final image [1849]. This should not be confused with frame rate, which is the number of frames displayed per second. For example, say the user is using an unteth-

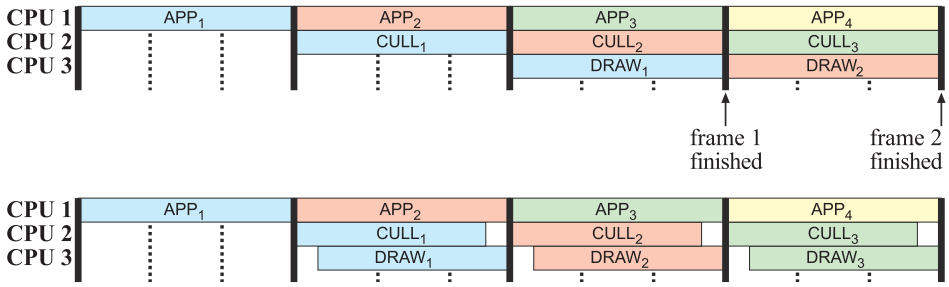

Figure 18.10. At the top, a three-stage pipeline is shown. In comparison to the configurations in Figure 18.9, this configuration has more time for each pipeline stage. The bottom illustration shows a way to reduce the latency: The CULL and the DRAW are overlapped with FIFO buffering in between.

ered head-mounted display. The determination of the head's position may take 10 milliseconds to reach the CPU, then it takes 15 milliseconds to render the frame. The latency is then 25 milliseconds from initial input to display. Even though the frame rate is 66.7 Hz (1/0.015 seconds), if no location prediction or other compensation is performed, interactivity can feel sluggish because of the delay in sending the position changes to the CPU. Ignoring any delay due to user interaction (which is a constant under both systems), multiprocessing has more latency than parallel processing because it uses a pipeline. As is discussed in detail in the next section, parallel processing breaks up the frame's work into pieces that are run concurrently.

In comparison to using a single CPU on the host, multiprocessor pipelining gives a higher frame rate and the latency is about the same or a little greater due to the cost of synchronization. The latency increases with the number of stages in the pipeline. For a well-balanced application the speedup is n times for n CPUs.

One technique for reducing the latency is to update the viewpoint and other latency-critical parameters at the end of the APP stage [1508]. This reduces the latency by (approximately) one frame. Another way to reduce latency is to execute CULL and DRAW overlapped. This means that the result from CULL is sent over to DRAW as soon as anything is ready for rendering. For this to work, there has to be some buffering, typically a FIFO, between those stages. The stages are stalled on empty and full conditions; i.e., when the buffer is full, then CULL has to stall, and when the buffer is empty, DRAW has to starve. The disadvantage is that techniques such as state sorting cannot be used to the same extent, since primitives have to be rendered as soon as they have been processed by CULL. This latency reduction technique is visualized in Figure 18.10.

The pipeline in this figure uses a maximum of three CPUs, and the stages have certain tasks. However, this technique is in no way limited to this configuration—rather, you can use any number of CPUs and divide the work in any way you want. The key is to make a smart division of the entire job to be done so that the pipeline tends to be balanced. The multiprocessor pipelining technique requires a minimum of synchronization in that it needs to synchronize only when switching frames. Additional processors can also be used for parallel processing, which needs more frequent synchronization.

18.5.2 Parallel Processing

A major disadvantage of using a multiprocessor pipeline technique is that the latency tends to increase. For some applications, such as flight simulators, first person shooters, and virtual reality rendering, this is not acceptable. When moving the viewpoint, you usually want instant (next-frame) response but when the latency is long this will not happen. That said, it all depends. If multiprocessing raised the frame rate from 30 FPS with 1 frame latency to 60 FPS with 2 frames latency, the extra frame delay would have no perceptible difference.

If multiple processors are available, one can also try to run sections of the code concurrently, which may result in shorter latency. To do this, the program's tasks must possess the characteristics of *parallelism*. There are several different methods for parallelizing an algorithm. Assume that n processors are available. Using static assignment [313], the total work package, such as the traversal of an acceleration structure, is divided into n work packages. Each processor then takes care of a work package, and all processors execute their work packages in parallel. When all processors have completed their work packages, it may be necessary to merge the results from the processors. For this to work, the workload must be highly predictable.

When this is not the case, dynamic assignment algorithms that adapt to different workloads may be used [313]. These use one or more work pools. When jobs are generated, they are put into the work pools. CPUs can then fetch one or more jobs from the queue when they have finished their current job. Care must be taken so that only one CPU can fetch a particular job, and so that the overhead in maintaining the queue does not damage performance. Larger jobs mean that the overhead for maintaining the queue becomes less of a problem, but, on the other hand, if the jobs are too large, then performance may degrade due to imbalance in the system—i.e., one or more CPUs may starve.

As for the multiprocessor pipeline, the ideal speedup for a parallel program running on n processors would be n times. This is called *linear speedup*. Even though linear speedup rarely happens, actual results can sometimes be close to it.

In Figure 18.8 on page 807, both a multiprocessor pipeline and a parallel processing system with three CPUs are shown. Temporarily assume that these should do the same amount of work for each frame and that both configurations achieve linear speedup. This means that the execution will run three times faster in comparison to serial execution (i.e., on a single CPU). Furthermore, we assume that the total amount of work per frame takes 30 ms, which would mean that the maximum frame rate on a single CPU would be $1/0.03 \approx 33$ frames per second.

The multiprocessor pipeline would (ideally) divide the work into three equal-sized work packages and let each of the CPUs be responsible for one work package. Each work package should then take 10 ms to complete. If we follow the work flow through the pipeline, we will see that the first CPU in the pipeline does work for 10 ms (i.e., one third of the job) and then sends it on to the next CPU. The first CPU then starts working on the first part of the next frame. When a frame is finally finished, it has taken 30 ms for it to complete, but since the work has been done in parallel in the pipeline, one frame will be finished every 10 ms. So, the latency is 30 ms, and the speedup is a factor of three (30/10), resulting in 100 frames per second.

A parallel version of the same program would also divide the jobs into three work packages, but these three packages will execute at the same time on the three CPUs. This means that the latency will be 10 ms, and the work for one frame will also take 10 ms. The conclusion is that the latency is much shorter when using parallel processing than when using a multiprocessor pipeline.

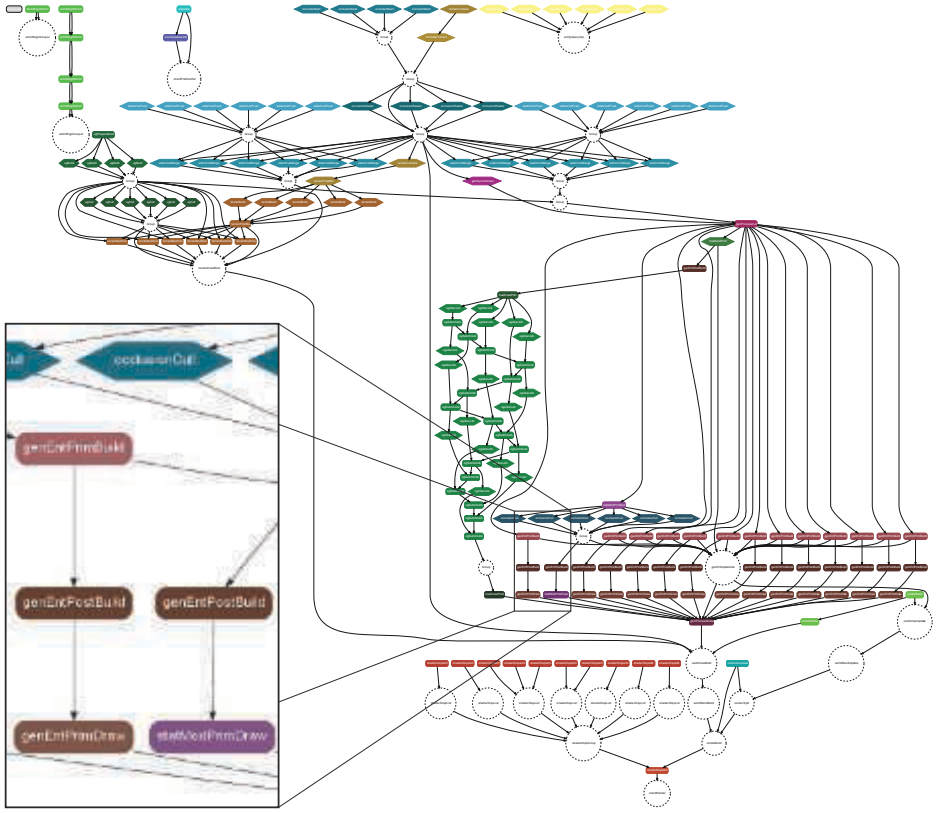

Figure 18.11. Frostbite CPU job graph, with one small zoomed-in part inset [45]. *(Figure courtesy of Johan Andersson—Electronic Arts.)*

18.5.3 Task-Based Multiprocessing

Knowing about pipelining and parallel processing techniques, it is natural to combine both in a single system. If there are only a few processors available, it might make sense to have a simple system of explicitly assigning systems to a particular core. However, given the large number of cores on many CPUs, the trend has been to use task-based multiprocessing. Just as one can create several tasks (also called *jobs*) for a process that can be parallelized, this idea can be broadened to include pipelining. Any task generated by any core is put into the work pool as it is generated. Any free processor gets a task to work on.

One way to convert to multiprocessing is to take an application's workflow and determine which systems are dependent on others. See Figure 18.11.

Having a processor stall while waiting for synchronization means a task-based version of the application could even become slower due to this cost and the overhead for task management [1854]. However, many programs and algorithms do have a large number of tasks that can be performed at the same time and can therefore benefit.

The next step is to determine what parts of each system can be decomposed into tasks. Characteristics of a piece of code that is a good candidate to become a task include [45, 1060, 1854]:

- The task has a well-defined input and output.
- The task is independent and stateless when run, and always completes.
- It is not so large a task that it often becomes the only process running.

Languages such as C++11 have facilities built into them for multithreading [1445]. On Intel-compatible systems, Intel's open-source *Threading Building Blocks* (TBB) is an efficient library that simplifies task generation, pooling, and synchronization [92].

Having the application create its own sets of tasks that are multiprocessed, such as simulation, collision detection, occlusion testing, and path planning, is a given when performance is critical [45, 92, 1445, 1477, 1854]. We note here again that there are also times when the GPU cores tend to be idle. For example, these are usually underused during shadow map generation or a depth prepass. During such idle times, compute shaders can be applied to other tasks [1313, 1884]. Depending on the architecture, API, and content, it is sometimes the case that the rendering pipeline cannot keep all the shaders busy, meaning that there is always some pool available for compute shading. We will not tackle the topic of optimizing these, as Lauritzen makes a convincing argument that writing fast and portable compute shaders is not possible, due to hardware differences and language limitations [993]. How to optimize the core rendering pipeline itself is the subject of the next section.

18.5.4 Graphics API Multiprocessing Support

Parallel processing often does not map to hardware constraints. For example, DirectX 10 and earlier allow only one thread to access the graphics driver at a time, so parallel processing for the actual draw stage is more difficult [1477].

There are two operations in a graphics driver that can potentially use multiple processors: resource creation and render-related calls. Creating resources such as textures and buffers can be purely CPU-side operations and so are naturally parallelizable. That said, creation and deletion can also be blocking tasks, as they might trigger operations on the GPU or need a particular device context. In any case, older APIs were created before consumer-level multiprocessing CPUs existed, so needed to be rewritten to support such concurrency.

A key construct used is the *command buffer* or *command list*, which harks back to an old OpenGL concept called the *display list*. A command buffer (CB) is a list of API state change and draw calls. Such lists can be created, stored, and replayed as

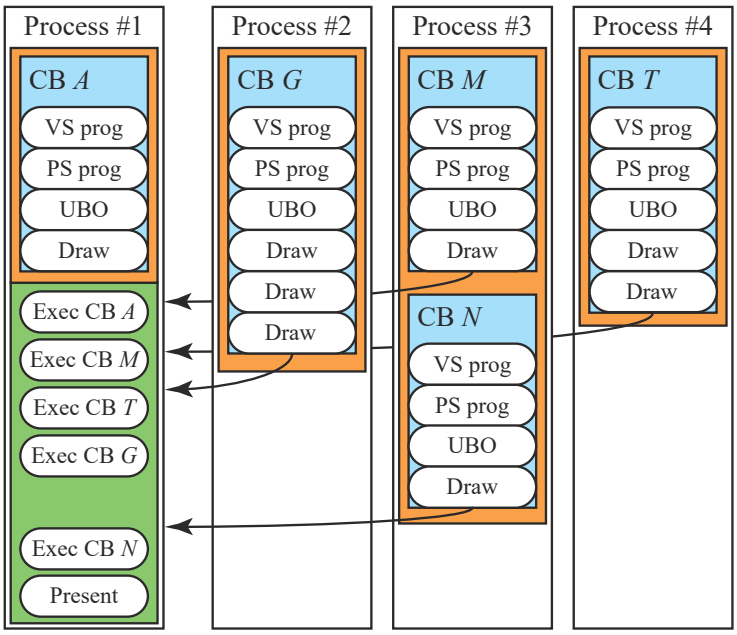

Figure 18.12. Command buffers. Each processor uses its deferred context, shown in orange, to create and populate one or more command buffers, shown in blue. Each command buffer is sent to Process #1, which executes these as desired, using its immediate context, shown in green. Process #1 can do other operations while waiting for command buffer N from Process #3. *(After Zink et al. [1971].)*

desired. They may also be combined to form longer command buffers. Only a single CPU processor communicates with the GPU via the driver and so can send it a CB for execution. However, every processor (including this single processor) can create or concatenate stored command buffers in parallel.

In DirectX 11, for example, the processor that communicates with the driver sends its render calls to what is called the *immediate context*. The other processors each use a *deferred context* to generate command buffers. As the name implies, these are not directly sent to the driver. Instead, these are sent to the immediate context for rendering. See Figure 18.12. Alternately, a command buffer can be sent to another deferred context, which inserts it into its own CB. Beyond sending a command buffer to the driver for execution, the main operations that the immediate context can perform that the deferred cannot are GPU queries and readbacks. Otherwise, command buffer management looks the same from either type of context.

An advantage of command buffers, and their predecessor, display lists, is that they can be stored and replayed. Command buffers are not fully bound when created, which aids in their reuse. For example, say a CB contains a view matrix. The camera moves, so the view matrix changes. However, the view matrix is stored in a constant

buffer. The constant buffer's contents are not stored in the CB, only the reference to them. The contents of the constant buffer can be changed without having to rebuild the CB. Determining how best to maximize parallelism involves choosing a suitable granularity—per view, per object, per material—to create, store, and combine command buffers [1971].

Such multithreading draw systems existed for years before command buffers were made a part of modern APIs [1152, 1349, 1552, 1554]. API support makes the process simpler and lets more tools work with the system created. However, command lists do have creation and memory costs associated with them. Also, the expense of mapping an API's state settings to the underlying GPU is still a costly operation with DirectX 11 and OpenGL, as discussed in Section 18.4.2. Within these systems command buffers can help when the application is the bottleneck, but can be detrimental when the driver is.

Certain semantics in these earlier APIs did not allow the driver to parallelize various operations, which helped motivate the development of Vulkan, DirectX 12, and Metal. A thin draw submission interface that maps well to modern GPUs minimizes the driver costs of these newer APIs. Command buffer management, memory allocation, and synchronization decisions become the responsibility of the application instead of the driver. In addition, command buffers with these newer APIs are validated once when formed, so repeated playback has less overhead than those used with earlier APIs such as DirectX 11. All these elements combine to improve API efficiency, allow multiprocessing, and lessen the chances that the driver is the bottleneck.

Further Reading and Resources

Mobile devices can have a different balance of where time is spent, especially if they use a tile-based architecture. Merry [1200] discusses these costs and how to use this type of GPU effectively. Pranckevičius and Zioma [1433] provide an in-depth presentation on many aspects of optimizing for mobile devices. McCaffrey [1156] compares mobile versus desktop architectures and performance characteristics. Pixel shading is often the largest cost on mobile GPUs. Sathe [1545] and Etuaho [443] discuss shader precision issues and optimization on mobile devices.

For the desktop, Wiesendanger [1882] gives a thorough walkthrough of a modern game engine's architecture. O'Donnell [1313] presents the benefits of a graph-based rendering system. Zink et al. [1971] discuss DirectX 11 in depth. De Smedt [331] provides guidance as to the common hotspots found in video games, including optimizations for DirectX 11 and 12, for multiple-GPU configurations, and for virtual reality. Coombes [291] gives a rundown of DirectX 12 best practices, and Kubisch [946] provides a guide for when to use Vulkan. There are numerous presentations about porting from older APIs to DirectX 12 and Vulkan [249, 536, 699, 1438]. By the time you read this, there will undoubtedly be more. Check IHV developer sites, such as NVIDIA, AMD, and Intel; the Khronos Group; and the web at large, as well as this book's website.

Though a little dated, Cebenoyan's article [240] is still relevant. It gives an overview of how to find the bottleneck and techniques to improve efficiency. Some popular optimization guides for C++ are Fog's [476] and Isensee's [801], free on the web. Hughes et al. [783] provide a modern, in-depth discussion of how to use trace tools and GPUView to analyze where bottlenecks occur. Though focused on virtual reality systems, the techniques discussed are applicable to any Windows-based machine.

Sutter [1725] discusses how CPU clock rates leveled out and multiprocessor chipsets arose. For more on why this change occurred and for information on how chips are designed, see the in-depth report by Asanovic et al. [75]. Foley [478] discusses various forms of parallelism in the context of graphics application development. *Game Engine Gems 2* [1024] has several articles on programming multithreaded elements for game engines. Preshing [1445] explains how Ubisoft uses multithreading and gives specifics on using C++11's threading support. Tatarchuk [1749, 1750] gives two detailed presentations on the multithreaded architecture and shading pipeline used for the game *Destiny*.

Chapter 19
Acceleration Algorithms

"Now here, you see, it takes all the running you can do to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"
—Lewis Carroll

One of the great myths concerning computers is that one day we will have enough processing power. Even in a relatively simple application such as word processing, we find that additional power can be applied to all sorts of features, such as on-the-fly spell and grammar checking, antialiased text display, and dictation.

In real-time rendering, we have at least four performance goals: more frames per second, higher resolution and sampling rates, more realistic materials and lighting, and increased geometrical complexity. A speed of 60–90 frames per second is generally considered fast enough. Even with motion blurring, which can lower the frame rate needed for image quality, a fast rate is still needed to minimize latency when interacting with a scene [1849].

Today, we have 4k displays with 3840×2160 resolution; 8k displays with 7680×4320 resolution exist, but are not common yet. A 4k display typically has around 140–150 dots per inch (DPI), sometimes called pixels per inch (PPI). Mobile phone displays have values ranging on up to around 400 DPI. A resolution of 1200 DPI, 64 times the number of pixels of a 4k display, is offered by many printer companies today. Even with a limit on screen resolution, antialiasing increases the number of samples needed for generating high-quality images. As discussed in Section 23.6, the number of bits per color channel can also be increased, which drives the need for higher-precision (and therefore more costly) computations.

As previous chapters have shown, describing and evaluating an object's material can be computationally complex. Modeling the interplay of light and surface can soak up an arbitrarily high amount of computing power. This is true because an image should ultimately be formed by the contributions of light traveling down a limitless number of paths from an illumination source to the eye.

Frame rate, resolution, and shading can always be made more complex, but there is some sense of diminishing returns to increasing any of these. However, there is no

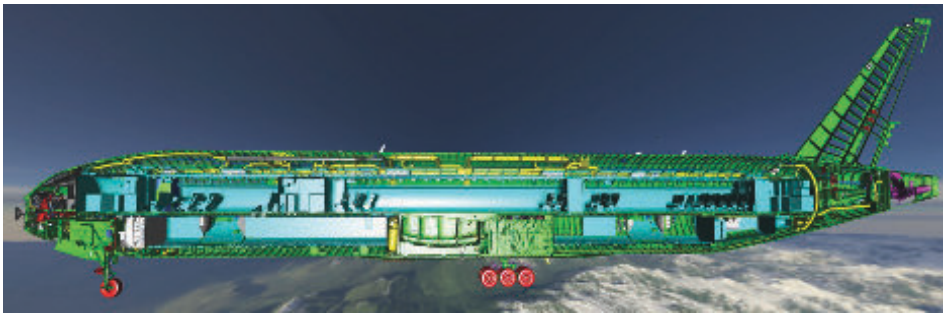

Figure 19.1. A "reduced" Boeing model with a mere 350 million triangles rendered with ray tracing. Sectioning is performed by using a user-defined clipping plane. *(Image courtesy of Computer Graphics Group, Saarland University. Source 3D data provided by and used with permission of the Boeing Company.)*

real upper limit on scene complexity. The rendering of a Boeing 777 includes 132,500 unique parts and over 3,000,000 fasteners, which yields a polygonal model with over 500,000,000 polygons [310]. See Figure 19.1. Even if most of those objects are not seen due to their small size or position, some work must be done to determine that this is the case. Neither z-buffering nor ray tracing can handle such models without the use of techniques to reduce the sheer number of computations needed. Our conclusion: Acceleration algorithms will always be needed.

In this chapter we offer a smörgåsbord of algorithms for accelerating computer graphics rendering, in particular the rendering of large amounts of geometry. The core of many such algorithms is based on *spatial data structures*, which are described in the next section. Based on that knowledge, we then continue with *culling techniques*. These are algorithms that try to rapidly determine which objects are visible and need to be treated further. *Level of detail* techniques reduce the complexity of rendering the remaining objects. To close the chapter, we discuss systems for rendering huge models, including virtual texturing, streaming, transcoding, and terrain rendering.

19.1 Spatial Data Structures

A spatial data structure is one that organizes geometry in some n-dimensional space. Only two- and three-dimensional structures are used in this book, but the concepts can often easily be extended to higher dimensions. These data structures can be used to accelerate queries about whether geometric entities overlap. Such queries are used in a wide variety of operations, such as culling algorithms, during intersection testing and ray tracing, and for collision detection.

The organization of spatial data structures is usually hierarchical. This means, loosely speaking, that the topmost level contains some children, each defining its own

volume of space and which in turn contains its own children. Thus, the structure is nested and of a recursive nature. Geometry is referenced by some of the elements in this hierarchy. The main reason for using a hierarchy is that different types of queries get significantly faster, typically an improvement from $O(n)$ to $O(\log n)$. That is, instead of searching through all n objects, we visit a small subset when performing operations such as finding the closest object in a given direction. Construction time of a spatial data structure can be expensive, and depends on both the amount of geometry inside it and the desired quality of the data structure. However, major advances in this field have reduced construction times considerably, and in some situations it can be done in real time. With lazy evaluation and incremental updates, the construction time can be reduced further still.

Some common types of spatial data structures are *bounding volume hierarchies*, variants of *binary space partitioning* (BSP) trees, quadtrees, and octrees. BSP trees and octrees are data structures based on *space subdivision*. This means that the entire space of the scene is subdivided and encoded in the data structure. For example, the union of the space of all the leaf nodes is equal to the entire space of the scene. Normally the leaf nodes' volumes do not overlap, with the exception of less common structures such as loose octrees. Most variants of BSP trees are *irregular*, which means that the space can be subdivided more arbitrarily. The octree is *regular*, meaning that space is split in a uniform fashion. Though more restrictive, this uniformity can often be a source of efficiency. A bounding volume hierarchy, on the other hand, is not a space subdivision structure. Rather, it encloses the regions of the space surrounding geometrical objects, and thus the BVH need not enclose all space at each level.

BVHs, BSP trees, and octrees are all described in the following sections, along with the *scene graph*, which is a data structure more concerned with model relationships than with efficient rendering.

19.1.1 Bounding Volume Hierarchies

A *bounding volume* (BV) is a volume that encloses a set of objects. The idea of a BV is that it should be a much simpler geometrical shape than the contained objects, so that tests using a BV can be done much faster than using the objects themselves. Examples of BVs are spheres, *axis-aligned bounding boxes* (AABBs), *oriented bounding boxes* (OBBs), and k-DOPs. See Section 22.2 for definitions. A BV does not contribute visually to the rendered image. Instead, it is used as a *proxy* in place of the bounded objects, to speed up rendering, selection, queries, and other computations.

For real-time rendering of three-dimensional scenes, the bounding volume hierarchy is often used for hierarchical view frustum culling (Section 19.4). The scene is organized in a hierarchical tree structure, consisting of a set of connected nodes. The topmost node is the *root*, which has no parents. An *internal node* has pointers to its children, which are other nodes. The root is thus an internal node, unless it is the only node in the tree. A *leaf node* holds the actual geometry to be rendered, and it

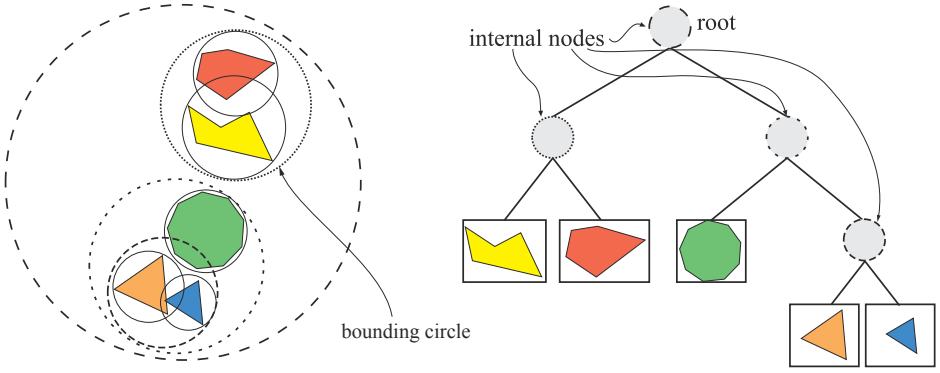

Figure 19.2. The left part shows a simple scene with five objects, with bounding circles that are used in the bounding volume hierarchy to the right. A single circle encloses all objects, and then smaller circles are inside the large circle, in a recursive manner. The right part shows the bounding volume hierarchy (tree) that is used to represent the object hierarchy on the left.

does not have any children nodes. Each node, including leaf nodes, in the tree has a bounding volume that encloses the geometry in its entire subtree. One may also decide to exclude BVs from the leaf nodes, and instead include them in an internal node just above each leaf node. This setup is where the name *bounding volume hierarchy* stems from. Each node's BV encloses the geometry of all the leaf nodes in its subtree. This means that the root has a BV that contains the entire scene. An example of a BVH is shown in Figure 19.2. Note that some of the larger bounding circles could be made tighter, as each node needs to contain only the geometry in its subtree, not the BVs of the descendant nodes. For bounding circles (or spheres), forming such tighter nodes can be expensive, as all geometry in its subtree would have to be examined by each node. In practice, a node's BV is often formed "bottom up" through the tree, by making a BV that contains the BVs of its children.

The underlying structure of a BVH is a tree, and in the field of computer science the literature on tree data structures is vast. Here, only a few important results will be mentioned. For more information, see, for example, the book *Introduction to Algorithms* by Cormen et al. [292].

Consider a k-ary tree, that is, a tree where each internal node has k children. A tree with only one node (the root) is said to be of height 0. A leaf node of the root is at height 1, and so on. A balanced tree is a tree in which all leaf nodes either are at height h or $h-1$. In general, the height, h, of a balanced tree is $\lfloor \log_k n \rfloor$, where n is the total number of nodes (internal and leaves) in the tree. Note that a higher k gives a tree with a lower height, which means that it takes fewer steps to traverse the tree, but it also requires more work at each node. The binary tree is often the simplest choice, and one that gives reasonable performance. However, there

is evidence that a higher k (e.g., $k = 4$ or $k = 8$) gives better performance for some applications [980, 1829]. Using $k = 2$, $k = 4$, or $k = 8$ makes it simple to construct trees; just subdivide along the longest axis for $k = 2$, and for the two longest axes for $k = 4$, and for all axes for $k = 8$. It is more difficult to form good trees for other values of k. Trees with a higher number, e.g., $k = 8$, of children per node are often preferred from a performance perspective, since they reduce average tree depth and the number of indirections (pointers from parent to child) to follow.

BVHs are excellent for performing various queries. For example, assume that a ray should be intersected with a scene, and the first intersection found should be returned, as would be the case for a shadow ray. To use a BVH for this, testing starts at the root. If the ray misses its BV, then the ray misses all geometry contained in the BVH. Otherwise, testing continues recursively, that is, the BVs of the children of the root are tested. As soon as a BV is missed by the ray, testing can terminate on that subtree of the BVH. If the ray hits a leaf node's BV, the ray is tested against the geometry at this node. The performance gains come partly from the fact that testing the ray with the BV is fast. This is why simple objects such as spheres and boxes are used as BVs. The other reason is the nesting of BVs, which allows us to avoid testing large regions of space due to early termination in the tree.

Often the closest intersection, not the first found, is what is desired. The only additional data needed are the distance and identity of the closest object found while traversing the tree. The current closest distance is also used to cull the tree during traversal. If a BV is intersected, but its distance is beyond the closest distance found so far, then the BV can be discarded. When examining a parent box, we intersect all children BVs and find the closest. If an intersection is found in this BV's descendants, this new closest distance is used to cull out whether the other children need to be traversed. As will be seen, a BSP tree has an advantage over normal BVHs in that it can guarantee front-to-back ordering, versus this rough sort that BVHs provide.

BVHs can be used for dynamic scenes as well [1465]. When an object contained in a BV has moved, simply check whether it is still contained in its parent's BV. If it is, then the BVH is still valid. Otherwise, the object node is removed and the parent's BV recomputed. The node is then recursively inserted back into the tree from the root. Another method is to grow the parent's BV to hold the child recursively up the tree as needed. With either method, the tree can become unbalanced and inefficient as more and more edits are performed. Another approach is to put a BV around the limits of movement of the object over some period of time. This is called a *temporal bounding volume* [13]. For example, a pendulum could have a bounding box that enclosed the entire volume swept out by its motion. One can also perform a bottom-up refit [136] or select parts of the tree to refit or rebuild [928, 981, 1950].

To create a BVH, one must first be able to compute a tight BV around a set of objects. This topic is treated in Section 22.3. Then, the actual hierarchy of BVs must be created. See the collision detection chapter at realtimerendering.com for more on BV building strategies.

19.1.2 BSP Trees

Binary space partitioning trees, or BSP trees for short, exist as two noticeably different variants in computer graphics, which we call *axis-aligned* and *polygon-aligned*. The trees are created by using a plane to divide the space in two, and then sorting the geometry into these two spaces. This division is done recursively. One worthwhile property is that if a BSP tree is traversed in a certain way, the geometrical contents of the tree can be sorted front to back from any point of view. This sorting is approximate for axis-aligned and exact for polygon-aligned BSPs. Note that the axis-aligned BSP tree is also called a k-d tree

Axis-Aligned BSP Trees (k-D Trees)

An axis-aligned BSP tree is created as follows. First, the whole scene is enclosed in an *axis-aligned bounding box* (AABB). The idea is then to recursively subdivide this box into smaller boxes. Now, consider a box at any recursion level. One axis of the box is chosen, and a perpendicular plane is generated that divides the space into two boxes. Some schemes fix this partitioning plane so that it divides the box exactly in half; others allow the plane to vary in position. With varying plane position, called nonuniform subdivision, the resulting tree can become more balanced. With a fixed plane position, called uniform subdivision, the location in memory of a node is implicitly given by its position in the tree.

An object intersecting the plane can be treated in any number of ways. For example, it could be stored at this level of the tree, or made a member of both child boxes, or truly split by the plane into two separate objects. Storing at the tree level has the advantage that there is only one copy of the object in the tree, making object deletion straightforward. However, small objects intersected by the splitting plane become lodged in the upper levels of the tree, which tends to be inefficient. Placing intersected objects into both children can give tighter bounds to larger objects, as all objects percolate down to one or more leaf nodes, but only those they overlap. Each child box contains some number of objects, and this plane-splitting procedure is repeated, subdividing each AABB recursively until some criterion is fulfilled to halt the process. See Figure 19.3 for an example of an axis-aligned BSP tree.

Rough front-to-back sorting is an example of how axis-aligned BSP trees can be used. This is useful for occlusion culling algorithms (Sections 19.7 and 23.7), as well as for generally reducing pixel shader costs by minimizing pixel overdraw. Assume that a node called N is currently traversed. Here, N is the root at the start of traversal. The splitting plane of N is examined, and tree traversal continues recursively on the side of the plane where the viewer is located. Thus, it is only when the entire half of the tree has been traversed that we start to traverse the other side. This traversal does not give exact front-to-back sorting, since the contents of the leaf nodes are not sorted, and because objects may be in many nodes of the tree. However, it gives a rough sorting, which often is useful. By starting traversal on the other side of a node's plane when compared to the viewer's position, rough back-to-front sorting can

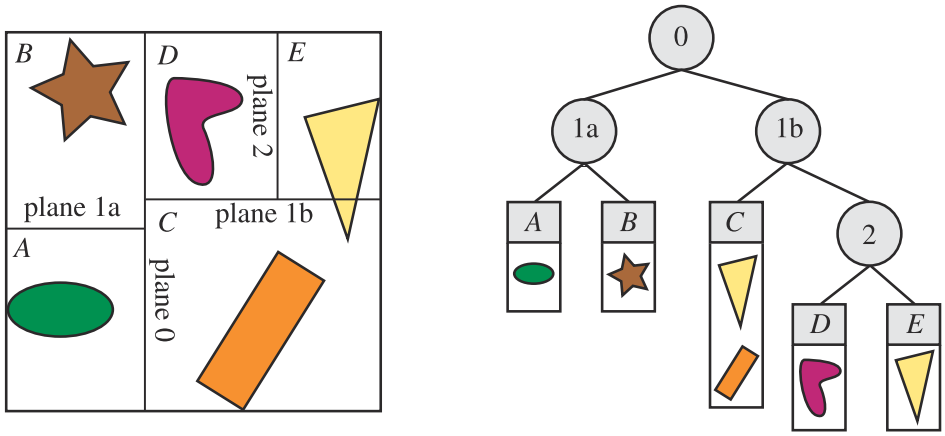

Figure 19.3. Axis-aligned BSP tree. In this example, the space partitions are allowed to be anywhere along the axis, not just at its midpoint. The spatial volumes formed are labeled A through E. The tree on the right shows the underlying BSP data structure. Each leaf node represents an area, with that area's contents shown beneath it. Note that the triangle is in the object list for two areas, C and E, because it overlaps both.

be obtained. This is useful for transparency sorting. BSP traversal can also be used to test a ray against the scene geometry. The ray's origin is simply exchanged for the viewer's location.

Polygon-Aligned BSP Trees

The other type of BSP tree is the polygon-aligned form [4, 500, 501]. This data structure is particularly useful for rendering static or rigid geometry in an exact sorted order. This algorithm was popular for games like *DOOM* (2016), back when there was no hardware z-buffer. It still has occasional use, such as for collision detection and intersection testing.

In this scheme, a polygon is chosen as the divider, splitting space into two halves. That is, at the root, a polygon is selected. The plane in which the polygon lies is used to divide the rest of the polygons in the scene into two sets. Any polygon that is intersected by the dividing plane is broken into two separate pieces along the intersection line. Now in each half-space of the dividing plane, another polygon is chosen as a divider, which divides only the polygons in its half-space. This is done recursively until all polygons are in the BSP tree. Creating an efficient polygon-aligned BSP tree is a time-consuming process, and such trees are normally computed once and stored for reuse. This type of BSP tree is shown in Figure 19.4. It is generally best to form a balanced tree, i.e., one where the depth of each leaf node is the same, or at most off by one.

The polygon-aligned BSP tree has some useful properties. One is that, for a given view, the structure can be traversed strictly from back to front (or front to back).

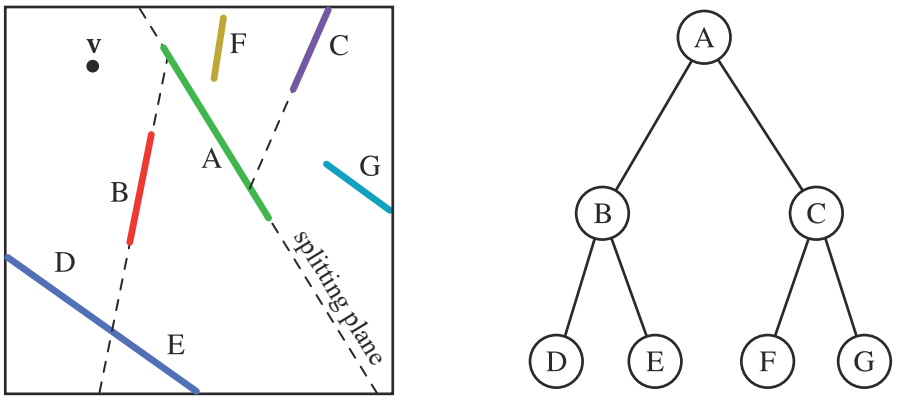

Figure 19.4. Polygon-aligned BSP tree. Polygons A through G are shown from above. Space is first split by polygon A, then each half-space is split separately by B and C. The splitting plane formed by polygon B intersects the polygon in the lower left corner, splitting it into separate polygons D and E. The BSP tree formed is shown on the right.

This is in comparison to the axis-aligned BSP tree, which normally gives only a rough sorted order. Determine on which side of the root plane the camera is located. The set of polygons on the far side of this plane is then beyond the near side's set. Now with the far side's set, take the next level's dividing plane and determine which side the camera is on. The subset on the far side is again the subset farthest away from the camera. By continuing recursively, this process establishes a strict back-to-front order, and a *painter's algorithm* can be used to render the scene. The painter's algorithm does not need a z-buffer. If all objects are drawn in a back-to-front order, each closer object is drawn in front of whatever is behind it, and so no z-depth comparisons are required.

For example, consider what is seen by a viewer $\mathbf{v}$ in Figure 19.4. Regardless of the viewing direction and frustum, $\mathbf{v}$ is to the left of the splitting plane formed by A, so C, F, and G are behind B, D, and E. Comparing $\mathbf{v}$ to the splitting plane of C, we find G to be on the opposite side of this plane, so it is displayed first. A test of B's plane determines that E should be displayed before D. The back-to-front order is then G, C, F, A, E, B, D. Note that this order does not guarantee that one object is closer to the viewer than another. Rather, it provides a strict occlusion order, a subtle difference. For example, polygon F is closer to $\mathbf{v}$ than polygon E, even though it is farther back in occlusion order.

19.1.3 Octrees

The octree is similar to the axis-aligned BSP tree. A box is split simultaneously along all three axes, and the split point must be the center of the box. This creates eight new boxes—hence the name *octree*. This makes the structure regular, which can make some queries more efficient.

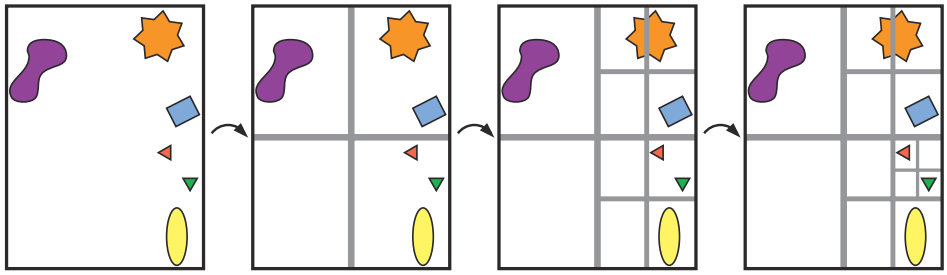

Figure 19.5. The construction of a quadtree. The construction starts from the left by enclosing all objects in a bounding box. Then the boxes are recursively divided into four equal-sized boxes until each box (in this case) is empty or contains one object.

An octree is constructed by enclosing the entire scene in a minimal axis-aligned box. The rest of the procedure is recursive in nature and ends when a stopping criterion is fulfilled. As with axis-aligned BSP trees, these criteria can include reaching a maximum recursion depth, or obtaining a certain number of primitives in a box [1535, 1536]. If a criterion is met, the algorithm binds the primitives to the box and terminates the recursion. Otherwise, it subdivides the box along its main axes using three planes, thereby forming eight equal-sized boxes. Each new box is tested and possibly subdivided again into $2 \times 2 \times 2$ smaller boxes. This is illustrated in two dimensions, where the data structure is called a *quadtree*, in Figure 19.5. Quadtrees are the two-dimensional equivalent of octrees, with a third axis being ignored. They can be useful in situations where there is little advantage to categorizing the data along all three axes.

Octrees can be used in the same manner as axis-aligned BSP trees, and thus, can handle the same types of queries. A BSP tree can, in fact, give the same partitioning of space as an octree. If a cell is first split along the middle of, say, the x-axis, then the two children are split along the middle of, say, y, and finally those children are split in the middle along z, eight equal-sized cells are formed that are the same as those created by one application of an octree division. One source of efficiency for the octree is that it does not need to store information needed by more flexible BSP tree structures. For example, the splitting plane locations are known and so do not have to be described explicitly. This more compact storage scheme also saves time by accessing fewer memory locations during traversal. Axis-aligned BSP trees can still be more efficient, as the additional memory cost and traversal time due to the need for retrieving the splitting plane's location can be outweighed by the savings from better plane placement. There is no overall best efficiency scheme; it depends on the nature of the underlying geometry, the usage pattern of how the structure is accessed, and the architecture of the hardware running the code, to name a few factors. Often the locality and level of cache-friendliness of the memory layout is the most important factor. This is the focus of the next section.

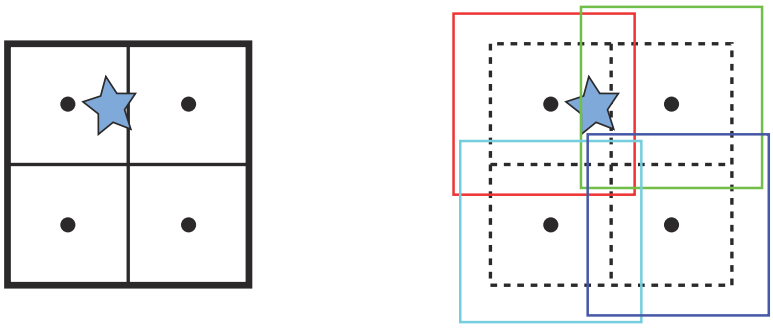

Figure 19.6. An ordinary octree compared to a loose octree. The dots indicate the center points of the boxes (in the first subdivision). To the left, the star pierces through one splitting plane of the ordinary octree. Thus, one choice is to put the star in the largest box (that of the root). To the right, a loose octree with $k = 1.5$ (that is, boxes are 50% larger) is shown. The boxes are slightly displaced, so that they can be discerned. The star can now be placed fully in the red box to the upper left.

In the above description, objects are always stored in leaf nodes. Therefore, certain objects have to be stored in more than one leaf node. Another option is to place the object in the box that is the smallest that contains the entire object. For example, the star-shaped object in the figure should be placed in the upper right box in the second illustration from the left. This has a significant disadvantage in that, for example, a (small) object that is located at the center of the octree will be placed in the topmost (largest) node. This is not efficient, since a tiny object is then bounded by the box that encloses the entire scene. One solution is to split the objects, but that introduces more primitives. Another is to put a pointer to the object in each leaf box it is in, losing efficiency and making octree editing more difficult.

Ulrich presents a third solution, *loose octrees* [1796]. The basic idea of loose octrees is the same as for ordinary octrees, but the choice of the size of each box is relaxed. If the side length of an ordinary box is l, then kl is used instead, where $k > 1$. This is illustrated for $k = 1.5$, and compared to an ordinary octree, in Figure 19.6. Note that the boxes' center points are the same. By using larger boxes, the number of objects that cross a splitting plane is reduced, so that the object can be placed deeper down in the octree. An object is always inserted into only one octree node, so deletion from the octree is trivial. Some advantages accrue by using $k = 2$. First, insertion and deletion of objects is $O(1)$. Knowing the object's size means immediately knowing the level of the octree it can successfully be inserted in, fully fitting into one loose box. In practice, it is sometimes possible to push the object to a deeper box in the octree. Also, if $k < 2$, the object may have to be pushed up the tree if it does not fit.

The object's centroid determines into which loose octree box it is put. Because of these properties, this structure lends itself well to bounding dynamic objects, at the expense of some BV efficiency, and the loss of a strong sort order when traversing the structure. Also, often an object moves only slightly from frame to frame, so that the previous box still is valid in the next frame. Therefore, only a fraction of animated

objects in the loose octree need updating each frame. Cozzi [302] notes that after each object/primitive has been assigned to the loose octree, one may compute a minimal AABB around the objects in each node, which essentially becomes a BVH at that point. This approach avoids splitting objects across nodes.

19.1.4 Cache-Oblivious and Cache-Aware Representations

Since the gap between the bandwidth of the memory system and the computing power of CPUs increases every year, it is critical to design algorithms and spatial data structure representations with caching in mind. In this section, we will give an introduction to cache-aware (or cache-conscious) and cache-oblivious spatial data structures. A cache-aware representation assumes that the size of cache blocks is known, and hence we optimize for a particular architecture. In contrast, a cache-oblivious algorithm is designed to work well for all types of cache sizes, and are hence platform-independent.

To create a cache-aware data structure, you must first find out what the size of a cache block is for your architecture. This may be 64 bytes, for example. Then try to minimize the size of your data structure. For example, Ericson [435] shows how it is sufficient to use only 32 bits for a k-d tree node. This is done in part by appropriating the two least significant bits of the node's 32-bit value. These 2 bits can represent four types: a leaf node, or the internal node split on one of the three axes. For leaf nodes, the upper 30 bits hold a pointer to a list of objects; for internal nodes, these represent a (slightly lower-precision) floating point split value. Hence, it is possible to store a four-level deep binary tree of 15 nodes in a single cache block of 64 bytes. The sixteenth node indicates which children exist and where they are located. See his book for details. The key concept is that data access is considerably improved by ensuring that structures pack cleanly to cache boundaries.

One popular and simple cache-oblivious ordering for trees is the van Emde Boas layout [68, 422, 435]. Assume we have a tree, $\mathcal{T}$, with height h. The goal is to compute a cache-oblivious layout, or ordering, of the nodes in the tree. The key idea is that, by recursively breaking a hierarchy into smaller and smaller chunks, at some level a set of chunks will fit in the cache. These chunks are near each other in the tree, so the cached data will be valid for a longer time than if, for example, we simply listed all nodes from the top level on down. A naive listing such as that would lead to large jumps between memory locations.

Let us denote the van Emde Boas layout of $\mathcal{T}$ as $v(\mathcal{T})$. This structure is defined recursively, and the layout of a single node in a tree is just the node itself. If there are more than one node in $\mathcal{T}$, the tree is split at half the height, $\lfloor h/2 \rfloor$. The topmost $\lfloor h/2 \rfloor$ levels are put in a tree denoted $\mathcal{T}_0$, and the children subtree starting at the leaf nodes of $\mathcal{T}_0$ are denoted $\mathcal{T}_1, \ldots, \mathcal{T}_n$. The recursive nature of the tree is described as follows:

$$v(\mathcal{T}) = \begin{cases} \{\mathcal{T}\}, & \text{if there is single node in } \mathcal{T}, \\ \{\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_n\}, & \text{else.} \end{cases} \tag{19.1}$$

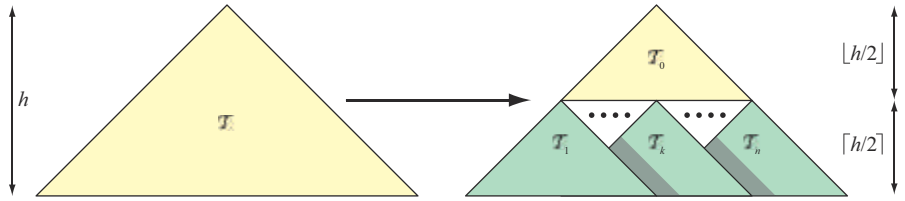

Figure 19.7. The van Emde Boas layout of a tree, $\mathcal{T}$, is created by splitting the height, h, of the tree in two. This creates the subtrees, $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_n$, and each subtree is split recursively in the same manner until only one node per subtree remains.

Note that all the subtrees $\mathcal{T}_i$, $0 \leq i \leq n$, are also defined by the recursion above. This means, for example, that $\mathcal{T}_1$ has to be split at half its height, and so on. See Figure 19.7 for an example.

In general, creating a cache-oblivious layout consists of two steps: clustering and ordering of the clusters. For the van Emde Boas layout, the clustering is given by the subtrees, and the ordering is implicit in the creation order. Yoon et al. [1948, 1949] develop techniques that are specifically designed for efficient bounding volume hierarchies and BSP trees. They develop a probabilistic model that takes into account both the locality between a parent and its children, and the spatial locality. The idea is to minimize cache misses when a parent has been accessed, by making sure that the children are inexpensive to access. Furthermore, nodes that are close to each other are grouped closer together in the ordering. A greedy algorithm is developed that clusters nodes with the highest probabilities. Generous increases in performance are obtained without altering the underlying algorithm—it is only the ordering of the nodes in the BVH that is different.

19.1.5 Scene Graphs

BVHs, BSP trees, and octrees all use some sort of tree as their basic data structure. It is in how they partition the space and store the geometry that they differ. They also store geometrical objects, and nothing else, in a hierarchical fashion. However, rendering a three-dimensional scene is about so much more than just geometry. Control of animation, visibility, and other elements are usually performed using a scene graph, which is called a *node hierarchy* in glTF. This is a user-oriented tree structure that is augmented with textures, transforms, levels of detail, render states (material properties, for example), light sources, and whatever else is found suitable. It is represented by a tree, and this tree is traversed in some order to render the scene. For example, a light source can be put at an internal node, which affects only the contents of its subtree. Another example is when a material is encountered in the tree. The material can be applied to all the geometry in that node's subtree, or possibly be overridden by a child's settings. See also Figure 19.34 on page 861 on how different levels of detail can be supported in a scene graph. In a sense, every graphics application uses some

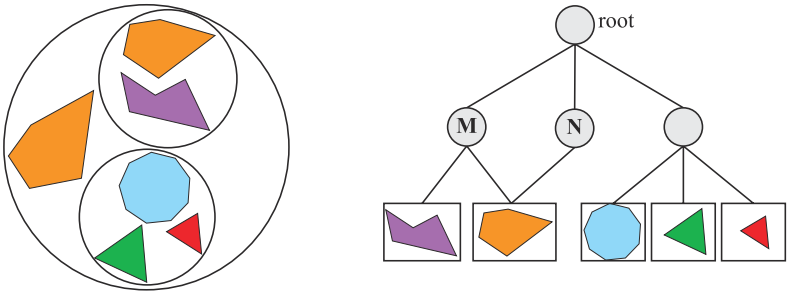

Figure 19.8. A scene graph with different transforms **M** and **N** applied to internal nodes, and their respective subtrees. Note that these two internal nodes also point to the same object, but since they have different transforms, two different objects appear (one is rotated and scaled).

form of scene graph, even if the graph is just a root node with a list of children to display.

One way of animating objects is to vary transforms of internal nodes in the tree. Scene graph implementations then transform the entire contents of that node's subtree. Since a transform can be put in any internal node, hierarchical animation can be done. For example, the wheels of a car can spin, and the car as a whole can move forward.

When several nodes may point to the same child node, the tree structure is called a *directed acyclic graph* (DAG) [292]. The term *acyclic* means that it must not contain any loops or cycles. By *directed*, we mean that as two nodes are connected by an edge, they are also connected in a certain order, e.g., from parent to child. Scene graphs are often DAGs because they allow for instantiation, i.e., when we want to make several copies (instances) of an object without replicating its geometry. An example is shown in Figure 19.8, where two internal nodes have different transforms applied to their subtrees. Using instances saves memory, and GPUs can render multiple copies of an instance rapidly via API calls (Section 18.4.2).

When objects are to move in the scene, the scene graph has to be updated. This can be done with a recursive call on the tree structure. Transforms are updated on the way from the root toward the leaves. The matrices are multiplied in this traversal and stored in relevant nodes. However, when transforms have been updated, any BVs attached are obsolete. Therefore, the BVs are updated on the way back from the leaves toward the root. A too relaxed tree structure complicates these tasks enormously, so DAGs are often avoided, or a limited form of DAGs is used, where only the leaf nodes are shared. See Eberly's book [404] for more information on this topic. Note also that when JavaScript-based APIs, such as WebGL, are used, then it is of extreme importance to move over as much work as possible to the GPU with as little feedback to the CPU as possible [876].

Scene graphs themselves can be used to provide some computational efficiency. A node in the scene graph often has a bounding volume, and is thus quite similar to