9

A Multithreaded 3D Renderer

Sebastien Schertenleib

Sony Computer Entertainment Europe

Overview

The 3D renderer remains one of the main components in most modern video games. Usually, 3D rendering engines handle both the software and hardware pipelines that are being exposed through 3D graphics APIs such as DirectX, OpenGL, or libgcm. Nowadays, multi-core CPUs are widely available through game consoles and PCs. In order to ensure that the GPU is continuously fed with data to process, it is critical that 3D renderers take full advantage of this new programming scheme by utilizing the available processing cores. The workflow involved in creating a 3D picture on the screen relies on preparing a list of commands that the GPU can interpret and execute.

Different approaches can be taken to decouple the CPU from the GPU. For instance, one common technique consists of using a double buffer or triple buffer scheme where the CPU builds the commands in frame *N* and where the GPU consumes them in frame *N*+1, as illustrated in Figure 9.1. Alternatively, the GPU can consume the data within the same frame in order to reduce the latency, as shown in Figure 9.2. One potential drawback is that it might be difficult to avoid some GPU stalls early in the frame. On the other hand, the memory footprint is much smaller compared to a double buffering method, and this is particularly important on embedded systems with

restricted amounts of memory.

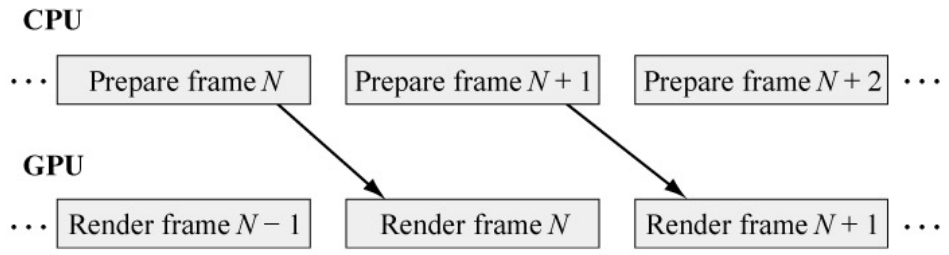

Figure 9.1: In a double-buffering scheme, the GPU consumes the data a frame later than it is generated by the CPU.

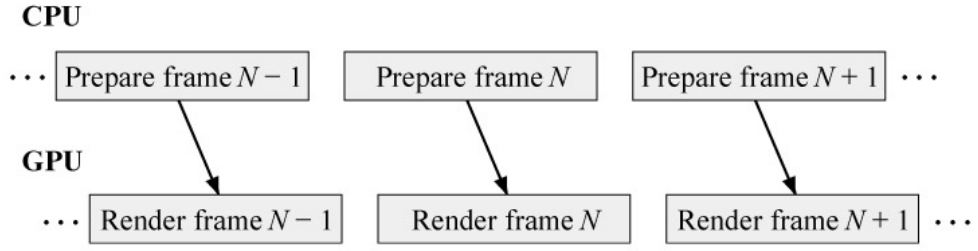

Figure 9.2: In this scheme, the GPU consumes the data soon after it is generated by the CPU.

9.1 The Memory Model

Regardless of how the display lists get created, a renderer might ultimately be restricted by memory access speed, particularly when the graphics commands are generated through a single thread. In recent years, the increasing performance gap between GPU processing power and memory latencies has made it harder to feed the GPU due to expensive data accesses in system memory, as shown in Figure 9.3. A typical frame is subdivided into passes which handle, for instance, rendering shadow maps, rendering the main scene, and rendering fullscreen post-processing effects. A unit of

work during rendering is often referred to as a *batch* and combines a set of render states, shaders, and geometry elements as exemplified by the following listing:

```
//setting up a batch
setRenderStates(...);
bindTextures(...);
setShaders(...);
setShaderConstants(...);
setVertexBuffer(...);
setIndexBuffer(...);
drawCall(...);
```

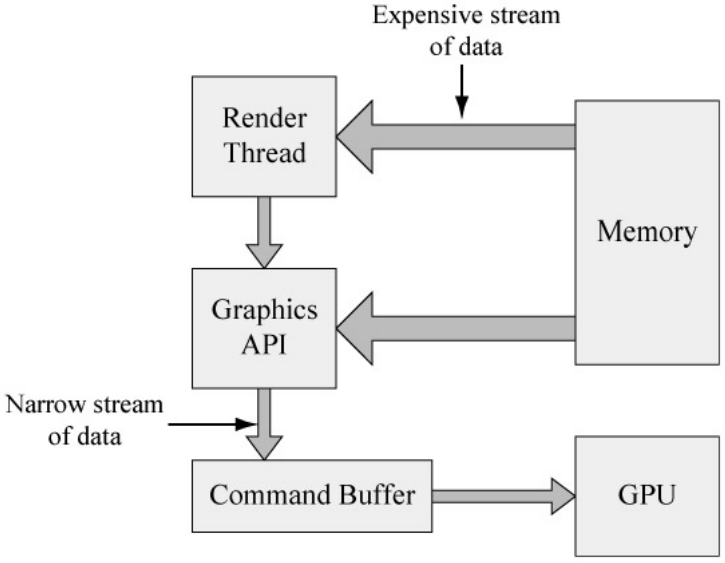

Figure 9.3: Both the render thread and the graphics API are likely to access data in various locations in memory.

Setting up a batch consists mostly of setting the addresses of various resources needed to render an object. In the process of creating batches, the rendering code has

to traverse the scene graph and is likely to access data in various locations throughout main memory. This leads to a large number of cache misses. With inorder CPUs such as the PowerPC chips found in the Xbox 360 and PlayStation 3 game consoles, this could stall the processor for hundreds of cycles each time a cache miss occurs, greatly impacting performance. This problem can be mitigated with out-of-order CPUs because other instructions can potentially be executed while previous instructions are waiting for data to be ready.

One may argue that it might be possible to reorganize the data structures to avoid many of the cache-miss penalties. Indeed, adopting cache-aware or cache-oblivious algorithms [2] helps, especially for the scene graph management, but unfortunately, the graphics library also needs to access and manipulate some data on its own. Large structures such as vertex buffers and index buffers are generally stored in the GPU's local memory and so do not cause a problem, but many types of rendering state, such as shader constants and texture configurations, need to be copied to the command buffer each frame. Therefore, alternative solutions are needed to overcome the cache limitations.

9.2 Building the Display Lists in Parallel

To ensure that the display lists are created in the shortest amount of time and in a way that minimizes memory bandwidth limitations, our solution uses multiple threads, and each is responsible for creating a subset of the rendering commands. However, this is only possible when the 3D graphics library exposes such a level of granularity. Thankfully, this is the case for Xbox 360, PlayStation 3, and DirectX 11 developers. In the PlayStation 3 case, the Synergistic Processing Units (SPUs) of the Cell Broadband Engine, being purely vector processors, excel with geometric processing. This means they can easily perform graphical operations that help offload work from the GPU when

necessary. To create the display lists in parallel, a commonly found paradigm is to have the primary command buffer reference display lists created inside secondary command buffers. Those display lists are created in parallel on different execution units as shown in Figure 9.4. Often, those secondary buffers handle a subset of the frame, and the granularity could be anything from a single draw call to an entire pass.

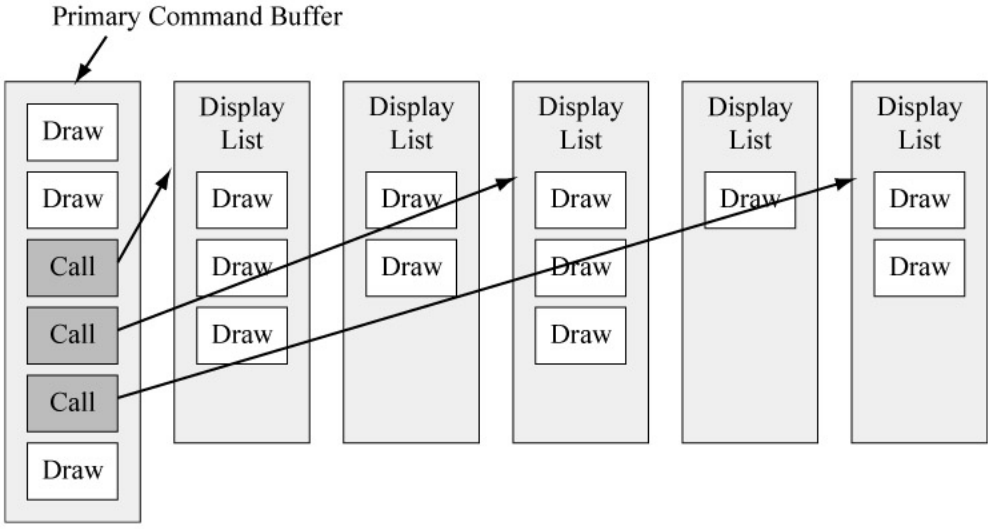

Figure 9.4: The primary command buffer references multiple secondary command buffers, each of which handles a subset of the frame.

By distributing the creation of the draw calls and their graphics states to multiple command buffers in parallel, the overall latency for creating the display lists is considerably reduced, which is particularly useful when using the scheme shown in Figure 9.2. Moreover, this makes the 3D renderer more scalable to various configurations of multi-core architectures and helps generate the thousands of draw calls commonly found in modern video games.

9.3 Parallel Models

3D renderers are strong candidates for parallelization because each draw call can usually be treated as a standalone unit of work. Each of these tasks pairs a chunk of data with some logic to operate on that data as shown in the following code listing.

```
int TaskMain(...)
{
  const int sourceAddr = ...;  // source address of data
  const int count = ...;       // number of elements
  const int dataSize = count * sizeof(gfxObject);

  // On some platforms like the PS3,
  // you may have to keep the data local to the executing unit
  gfxObject *buffer = (gfxObject *) Allocate(dataSize);
  DmaGet(buffer, sourceAddr, dataSize);
  DmaWait(...);   // barrier to wait for the data

  // let's do some work
  for (int i = 0; i < count; ++i)
  {
    buffer[i]->update();
  }

  // On some platforms like the PS3,
  // you may have to store back the data to system memory
  DmaPut(buffer, sourceAddr, dataSize);
  DmaWait(....);  // barrier to wait for the data
}
```

This model provides an efficient parallel paradigm, where each task can be queued and consumed by available processing units, as shown in Figure 9.5. This also avoids

some of the issues of a more standard multithreaded approach, as the tasks provide a more fine-grained subdivision and can cope more easily with uneven computation, while a multithreaded architecture might end up waiting on a particular subsystem to complete its tasks.

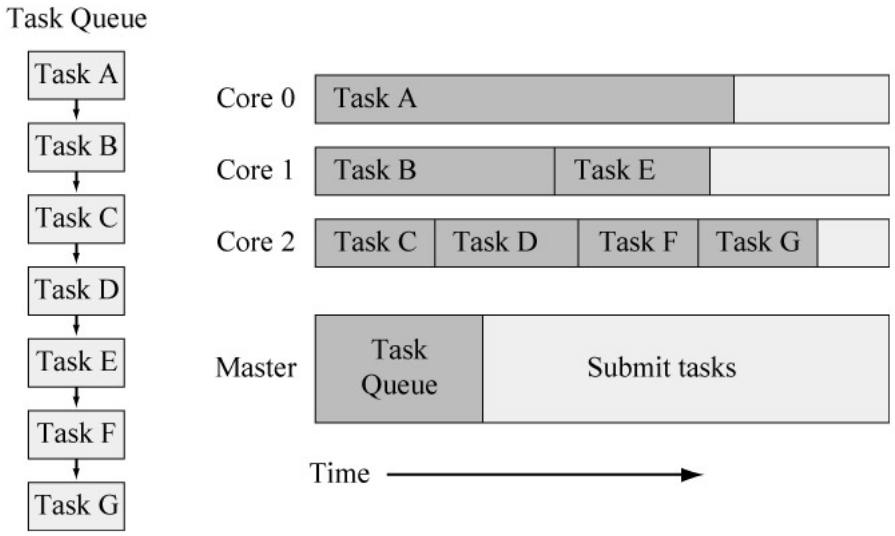

Figure 9.5: Tasks are stored in a queue and are consumed by available processing units.

9.4 Synchronizing the GPU and CPU

Synchronizing the GPU and CPU is more difficult to handle in a multithreaded environment since multiple processing units might potentially want to access the same data. To avoid any inconsistencies and race conditions, there is a need to employ synchronization primitives such as mutexes or atomics. Usually, the GPU can report its progress in a specific memory area. For instance, when a particular command has been finished on the GPU, it can write a specified value, or "report", to a CPU-accessible

location to indicate completion. Depending on the architecture, the CPU can either poll for the report value or receive a system callback of some kind. Table 9.1 presents some possible configurations.

Table 9.1: Synchronizing the GPU and the CPU.

CPU	CPU	Report	Comments
…	WaitReport(22)	0	GPU waits for report to be set
SetReport(22)	WaitReport(22)	22	GPU is now unlocked
…	…	22	
WaitReport(33)			CPU wait for report to be set
WaitReport(33)	SetReport(33)	33	GPU set the report & unlock CPU
…	…	33	
SetReport(12)	SetReport(15)	?	Race condition
WaitReport(33)	WaitReport(22)	?	Deadlock

As with any multithreaded environment, special care is needed to avoid potential race conditions or deadlocks since both the CPU and the GPU can generate and consume data with unpredictable timing patterns.

9.5 Using Additional Processing Resources

In many games, the processing load is not fully balanced among the available processing units. In these cases, it can be worthwhile to move some operations that are ordinarily performed on the main CPU to other units that may have idle time each frame. For instance, less intensive graphics applications can employ GPGPU (general purpose GPU) code to offload physics or AI simulations from the CPU using technologies such as CUDA. On the other hand, games willing to push the boundaries

of real-time 3D graphics might want to use spare CPU cores to perform some graphical operations. If it so happens that those cores provide an interesting ISA (instruction set architecture) with SIMD instructions such as the SPUs on the Cell processor, then it becomes possible to offload the GPU for several kinds of operations such as the following:

- Geometric processing, including procedural algorithms for creating terrain, trees, decals, or subdivision surfaces.
- Physics and particle system updates.
- View frustum object culling or occlusion culling.
- Software rendering, in particular for occlusion queries and post-processing effects.

9.6 Reducing the Pressure on the Memory Bandwidth

The performance of both the CPU and GPU is improving at a very fast pace, but memory speed is not following the same curve, and consequently, memory bandwidth becomes more and more of a significant bottleneck. Therefore, it is important to consider any technique that would help minimize the requirements on the memory systems, even if it means using more CPU or GPU cycles. Some techniques that can be used include packing the shader input and output attributes or using the tessellation units of the GPU when available. On the CPU side, special geometry culling can avoid sending up to 70% of primitives that end up being discarded by the GPU (backfacing, off-screen, zero-size, and degenerate primitives). Another technique is to generate a coarse depth buffer in software to perform occlusion queries instead of relying on the GPU [1], as the latter can involve reading back from video memory, which is often a slow path.

9.7 Performing Graphical Operations in Parallel

So far, we have discussed some techniques that help to improve overall performance, but we can go a step further and allow different processing units to work in parallel to create the final image for a frame. Modern games use multiple render targets within a single frame, and some of them are not accessed simultaneously by the GPU. For instance, it is often possible to access a back buffer with another processing unit such as an SPU to perform some post-processing effects while the GPU starts rendering the next frame. This gives up to a full frame to render the effects, while keeping the same frame rate, but at the expense of an additional frame of latency (see Figure 9.6).

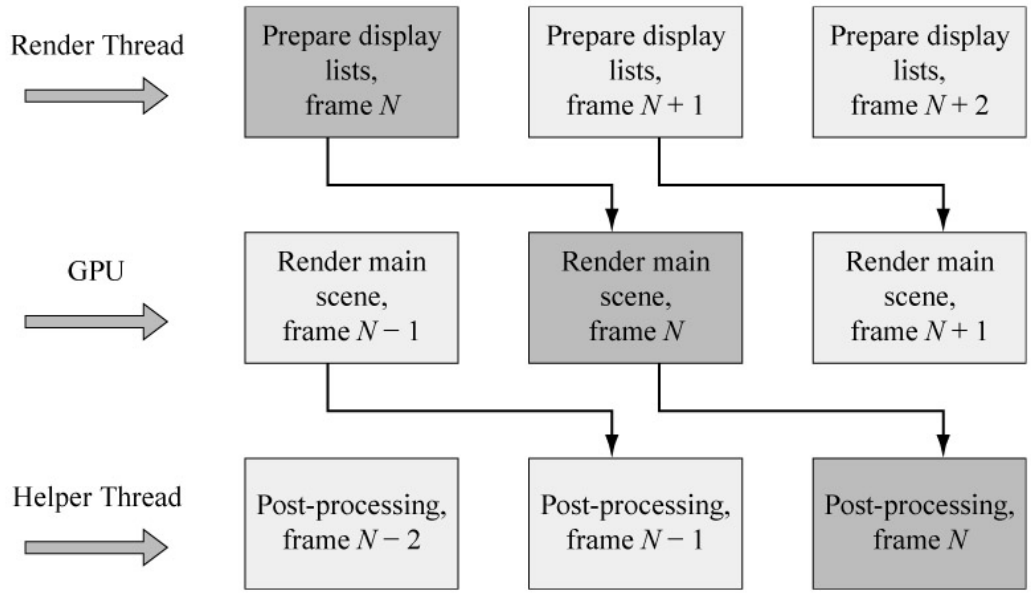

Figure 9.6: While the helper thread computes the post-effects, the GPU starts rendering the next frame.

References

[1] Johan Andersson."The Intersection of Game Engines and GPUs: Current & Future". *Graphics Hardware 2008.*

[2] Sebastien Schertenleib."An Effective Cache-Oblivious Implementation of the ABT Tree". *Game Programming Gems 5*, Charles River Media, 2005.