A Novel Multithreaded Rendering System based on a Deferred Approach

Jorge Alejandro Lorenzon
Universidad Austral, Argentina
jorgelorenzon@gmail.com

Esteban Walter Gonzalez Clua
Media Lab – UFF, Brazil
esteban@ic.uff.br

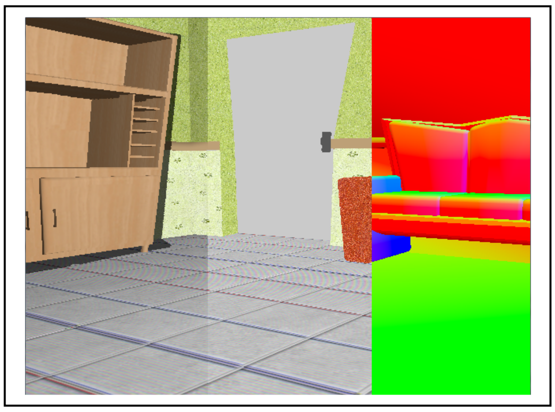

Figure 1: Mix of the final illuminated picture, the diffuse color buffer and the normal buffer

Abstract

This paper presents the architecture of a rendering system designed for multithreaded rendering. The implementation of the architecture following a deferred rendering approach shows gains of 65% on a dual core machine.

Keywords: multithreaded rendering, deferred rendering, DirectX 11, command buffer, thread pool

1. Introduction

Game engines and 3D software are constantly changing as the underlying hardware and low level APIs evolve. The main driving force of change is the pursuit of greater performance for 3D software, which means, pushing more polygons with more realistic models of illumination and shading techniques to the screen. The problem then becomes how to design 3D software in order to use the hardware to its maximum potential.

Central processing unit (CPU) manufactures are evolving the hardware to multi-core solutions. Currently dual core CPUs have become the common denominator while quad cores, like the recently released Intel Core i7 Extreme Edition processor, with the capability of running 8 processing threads, are slowly filling the high end market. In order for software to use all the capabilities and potential of the hardware it is now imperative that it divides its execution tasks among the different cores.

Therefore, the architecture of newer game engines must include fine-grained multithreaded algorithms and systems. Fortunately for some systems like physics and AI this can be done. However, when it comes to rendering there is one big issue: All draw and state calls must go to the graphics processing unit (GPU) in a serialized manner[1]. This limits game engines as only one thread can actually execute draw calls to the graphics card. Adding to the problem, draw calls and state management of the graphics pipeline are expensive for the CPU as there is a considerable overhead created by the API and driver. For this reason, most games and 3D applications are CPU bound and rely on batching 3D models to feed the GPU.

Microsoft, aware of this problem, is pushing forward a new multithreaded graphics API for the PC, Direct3D11. Direct3D11 was designed to remove the current restriction of single threaded rendering. It allows this by:

- Providing the ability to record command buffers in different threads
- Free threaded creation of resources and states

Command buffers are basically lists of functions to execute. There are two types of command buffers:

[1] This holds true for multi-GPU solutions, such as those that use SLI. The actual speedup with these setups (using alternate frame rendering) is accomplished by having the graphics API buffer commands for multiple frames so that each GPU can work on one. For this to happen effectively the application must not be limited by the CPU.

those that are part of an API and those that are not. The common benefit that both provide is that by deferring the communication with the GPU to a later stage, they allow the application to simulate multiple devices and divide its rendering work across multiple threads Natively supported command buffers can provide an additional performance benefit: Usually part of the graphic APIs' functions have a part that needs to be executed in the CPU like the validation of the data passed to them. So if the command buffers are designed to only hold commands ready to be executed by the GPU, the CPU load of the graphic APIs' functions can be processed at the time of their building, thus benefiting from the use of all the CPU cores. The execution of these buffers is then easier on the CPU leading to increased performance in CPU bound applications.

This paper proposes a new architecture of a multithreaded rendering system and shows the performance gains in different scenarios with an implementation based on DirectX11.

2. Related Work

Games traditionally use a game loop that execute update and render logic serially. The first approach in game engines to increase performance in multi-core hardware was to execute natural independent systems in parallel. The problem with this approach is that very few systems are independent from each other. For example: a particle system is independent from the AI system, however, the AI system is not independent from the physics engine as it needs to have the latest state of the world objects to compute the behavior of AI driven entities. The rendering and sound system need to have all the final data for the frame to present to the user so they depend on all of the systems. Thus, just multithreading independent systems is not an adequate enough solution for current hardware.

An engine has to be designed from the ground up with multiprocessing in mind to fully utilize multiprocessor hardware. There are two classic ways to approach this task: multiprocessor pipelining and parallel processing [Akenine-Moller et al. 2008]. Multiprocessor pipelining consists in dividing the execution in different stages so that each processor works on a different pipeline stage. For example: if the pipeline is divided into stages APP, CULL, and DRAW. For a given frame N, Core 0 would work on APP on frame N + 2, Core 1 would work on CULL on frame N + 1 and Core 2 would work on frame N. This architecture increases the throughput with the negative effect of increased latency. Parallel processing, on the other hand, consists in dividing up the work into small independent packages. This approach provides a theoretical linear speedup but requires for the algorithm to be naturally parallel.

Multithreaded engines have adopted different combinations of the techniques of multiprocessor pipelining and parallel processing. One approach has been to let each system of an engine run in a thread of its own [Gabb and Lake 2005]. In this solution systems use the latest available data for them, many times, like in multiprocessor pipelining, the data has been processed in a previous frame by a different system. The data independent systems benefit from the parallel processing speedup. Data sharing between systems is the biggest challenge in this type of architecture. Usage of synch primitives around shared data can be very expensive, so a buffering scheme is usually used to make it possible to for a system to write to a buffer while another system reads from a previously written buffer, thus avoiding heavy use of synch primitives. However, there are two problems with buffering schemes; firstly they utilize more memory, a scarce resource in some platforms. Secondly, copying of memory between buffers[2] might be expensive in terms of processing time [Lewis 2007]. In conclusion, this type of architecture is adequate while the number of systems is greater or equal to the number of cores. However, to scale further the game engine systems need to be designed to be internally multithreaded.

The system that this paper focuses on is the graphics system. The internal multithreaded architecture relies on the use of command buffers. The next paragraph gives an overview of the current support of command buffers under different platforms.

Microsoft's XBOX 360 DirectX and DirectX 11 support native command buffers [Lee 2008]. For other platforms, non-native command buffers can be used. The development team of Gamebryo has created an open source command buffer recording library for older versions of DirectX [Scheib 2008]. Their implementation currently only supports DirectX 9 but they are currently working on the implementation for DirectX 10. The multithreaded architecture of the rendering system is discussed in section 3, but before jumping to that section, it is important to read about the pipeline that will follow the rendering system.

A deferred rendering pipeline is used in the graphics system treated in this paper. This type of pipeline was chosen as many modern games, like S.T.A.L.K.E.R [Pharr 2005] and Tabula Rasa [Nguyen 2007] among others, have adopted this technique. Its main benefit is the decoupling of the geometry stage from the light stage [Deering et al. 1988]. This decoupling allows rendering to have a linear N + L complexity instead of the greater N * L complexity of the forward rendering approach, where N is the total number of objects and L is the total number of lights.

[2] Memory copying is necessary so that a system that reads, processes, and writes to buffer 1 is able to utilize the data written in buffer 1 in the next frame when it will need to do the same process using buffer 2.

Older games did not use this technique, as it requires the support of multiple render targets, extra video memory, and a higher memory bandwidth. However, with current hardware these requirements are no longer prohibitive.

3. Multithreaded Rendering System

Games usually have a 25% to 40% of the frame time used by the D3D runtime and driver [Davies 2008]. The overhead would not be such a problem if the engine could work on something else while the scene is being rendered. However, because the graphics system needs for the shared data to remain unmodified while it is doing its work the other engine systems become stalled. So if the graphics system is single threaded, the application wastes much of the CPU power. Therefore, the system architecture discussed in this paper is designed to utilize all of the CPU cores to create command buffers to give back to the other systems the ownership of the shared data as soon as possible. Once the buffers are created the "update" systems can run again while only one graphics thread remains submitting the command buffers to GPU. Figure 2 illustrates the flow described.

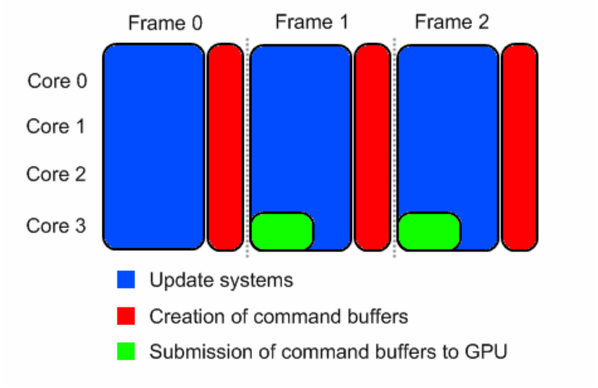

Figure 2: Top level application flow

The next sub section explains the approach and design used to build such a rendering system.

3.1 API abstraction layer

The first step in the design of the system is to abstract it from the API that will be used. This allows the software products that will later be constructed on top of the engine to be able to target more platforms.

Engines commonly grouped the creation of state and resources objects with the draw and state calls under a single rendering interface. The first design decision was to divide the responsibilities in two different interfaces: Device and Context. Device is in charge of the creation of resources and state objects while Context is responsible for the actual rendering. The Device is expected to be only instantiated once, while from Context many instances may be created; one for each thread that will submit rendering work. Having a mapping of one to one between contexts and threads is a necessary limitation to avoid the performance penalty that would appear from the need of synch primitives to maintain a rendering consistency. The Device, on the other hand, may be called from any thread. The best alternative, though, is to have an independent thread that manages the creation of resources. This could allow an application to go through a continuous world without needing to stop with loading screens.

The second step in creating the low level abstraction was to abstract resource and state objects. An abstract data type was declared for each of these. Adapter classes were created to extend from these abstract data types to make the adaptation necessary to communicate with the various graphic APIs. Figure 3 shows the UML class diagram for the depth stencil state.

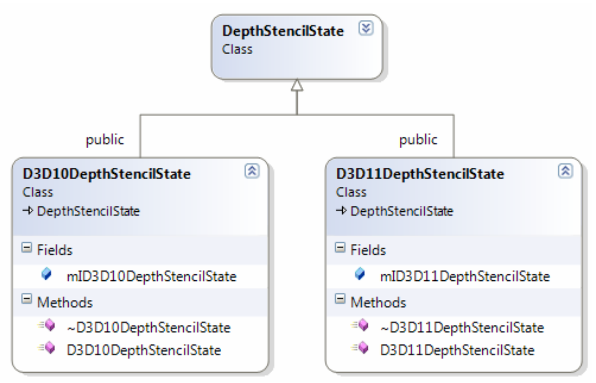

Figure 3: Depth stencil state abstraction

3.2 Rendering in different contexts

The instances of Context are the "renderers". As such, they are the ones that receive the messages to draw or change states. Having a number of them allows the application to submit geometry in parallel.

However, not all of them are equal, since there is **one** context with a special privilege: The Immediate Context. This is the only one that can effectively communicate with the GPU. The other contexts called Deferred Contexts submit state and draw calls to command buffers. Each of the deferred contexts then contains a command buffer ready to be executed by either another deferred context or by the immediate context. Therefore, the creation of command buffers becomes multithreaded while the submission of them is single threaded. The distinction of contexts is made because the nature of the CPU-GPU hardware allows only one CPU thread to send content to the GPU. The benefit in speed comes reducing the time that other systems remain stalled and from using idle CPU cores to help in the task of converting the state/draw calls from the API into a list of low level calls ready to be executed by the GPU. The immediate context can then execute the command buffers at a faster rate than the equivalent content via direct calls.

3.3 Graphics manager and render processes

In the previous section the low level abstraction layer was presented. Its designed was influenced greatly by the Direct3D11 API. In this section the next layer of the architecture will be presented. This layer will be responsible for the load balancing of rendering work.

Context and thread creation is not a lightweight task and so creating them to render every frame is not an option if we want to keep a high performance. The knowledge of how many contexts to create is part of the application design. Therefore, when initializing this layer that information will need to be communicated to it so it can allocate the resources needed at startup.

The Graphics Manager, represented in figure 4, is the central class of this layer. It is responsible for initializing a pool of threads and subsequently feeding them with the work that comes from the application. For each of the threads created a context is instanced and assigned to it. This ownership extends throughout the thread's life. Changing the ownership of the context is not possible because different threads may never make calls to the same context.

These threads remain asleep as long as no work is assigned to them. This prevents the Graphics Manager from consuming CPU cycles when the application is not rendering.

By making the number of possible worker threads variable the application developer has the freedom to choose as many threads as cores are available or any other number that the developer feels that it will provide a better performance.

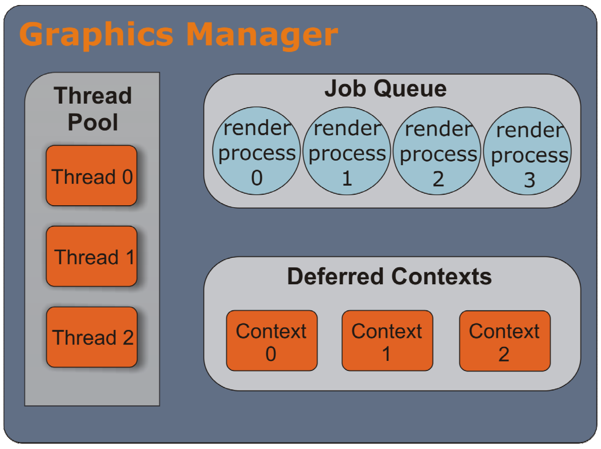

Figure 4: Graphics Manager Diagram

Because the developer should be limited as little as possible by the engine, the type of work that can be submitted to the pool can be as fine grained as executing a single draw call or as coarse as rendering a shadow map. The only limitation is that all work tasks need to be of the type Render Process.

The Render Process abstract class is the second most important class of this layer. Its importance is due to the ability that this class provides to application specific rendering work to be executed by the Graphics Manager through polymorphism.

The applications can define their rendering pipelines by creating and implementing subclasses of Render Process. The Graphics Manager commences the execution of the rendering work by calling the virtual method `Execute()` of Render Process from one of the pool threads.

The Render Process and the derived classes receive the context in which they can submit work as a parameter of `Execute()`. This frees the Graphics Manager from needing to keep a record of which context was used for each render process and allows it to do an optimal load balancing.

4. Multithreaded Deferred rendering

In the previous section we have discussed the architecture of the rendering system. In this section we will discuss how a deferred rendering pipeline fits on top and benefits from the multithreading rendering. A pipeline of this nature includes: a geometry buffer, lighting, transparency and post processing effects stages [Policarpo and Fonseca 2005]. The latter two were not implemented for the testing of the architecture. The following sub sections describe what the stages do and how they were encapsulated to be multithreaded rendered.

4.1 Geometry buffer

The geometry buffer (G-Buffer) creation is the first step in a deferred rendering pipeline. The purpose of the creation of the G-Buffer is to store the information necessary for the shading of each pixel during the lighting stage [Policarpo and Fonseca 2005]. The values that get stored depend on what illumination model the application will use. For the purpose of testing the performance of the architecture by keeping the application CPU bound a simple G-Buffer was used. The values stored are depths, normals and diffuse color.

The flow of data to the G-Buffer for every pixel is what consumes a lot of bandwidth. This is why before starting to create the G-Buffer itself it is better to have a rendering pass that just calculates the vertices' positions in order to set the Z-buffer. This pass, however, was not implemented for the test setup.

The steps to create the G-Buffer were all encapsulated in a class called Geometry Buffer Creator, which, extends from Render Process. The steps that it goes through are:

```
1. Set viewport
2. Set render targets
```

3. Set depth stencil state (read and write depth enabled)
4. Set rasterizer state (cull back, solid fill)
5. Clear the depth texture
6. Render each of the objects
7. Finish the command buffer

4.2 Lighting

Next is the lighting stage, where the application sends to the pipeline the lights that affect the scene. The effect that each light has on the pixels is calculated with the stored information in the geometry buffer and added to the final frame buffer.

4.2.1 Non-shadow casting lights

There are different alternatives to render non-shadow casting lights. The most efficient is to use geometry to represent lights [Calver 2003]. With this approach a spotlight is represented as a cone, a point light as a sphere and a directional light as a screen aligned quad. The benefit of using geometry is that the Z-Buffer rejects pixels more effectively than using scissors rectangles.

For the pixels that are not rejected by the scissor test or Z-Buffer a pixel shader that calculates how the light is influencing it is executed.

The process of rendering the non-shadow casting lights is isolated in another render process called Non Shadow Casting Lighting. The steps that this process goes through are:

1. Set viewport
2. Set final buffer as the render target
3. Set depth stencil state (depth test enabled, write disabled)
4. Set rasterizer state (fill solid)
5. Set the G buffer as shader resources
6. Render each light
7. Finish the command buffer

4.2.2 Shadow casting lights

The difference between non-shadow and shadow casting lights is that the latter have to calculate the obstruction of light due to the scene’s geometry. One effective way to calculate this obstruction is through a technique called shadow mapping [Akenine-Moller et al. 2008]. Rendering the scene from the light’s point of view while having writes and reads on the depth buffer enabled creates a shadow map. The shadow map contains depth information where the light is obstructed by geometry.

The shadowing lights’ pixel shader, before shading, checks if the pixel is affected by the light or if it is in shadow.

The test application uses one shadowing light. It is a hemispherical light and the shadow map algorithm used was parabolic mapping. The process to create the shadow map and light the pixels was encapsulated in the Shadow Renderer class. The steps that it goes through are:

1. Set viewport
2. Set depth buffer
3. Set depth stencil state (read and write depth enabled)
4. Set rasterizer state (cull back, solid fill)
5. Clear depth buffer
6. Render all objects that casts shadows from the point of view of the light
7. Set the shadow map (depth buffer recently set) as a shader resource
8. Render light
9. Finish command buffer

5. Discussion

When designing the graphics system it was assumed that the application would give to it the total ownership of the CPU and data. With this in mind it, the architecture was built to fulfill two major objectives: stall the other application systems the less time possible, and provide transparent scalability throughout different platforms, current and future ones.

To make the system flexible enough it was designed to follow a producer-consumer model. The work products (encapsulated in render processes) are produced by the application and are consumed by the available threads. The means of distribution of the render processes is the Graphics Manager’s responsibility. This decision was made to encapsulate the necessary platform dependent code that creates and manages threads. Clean encapsulation of platform dependent code makes not only porting, but also optimizations for different platforms easier.

It is important to note that if the number of cores increases beyond the number of stages that the rendering pipeline has, a further splitting will be needed. The split can occur at the application data level. For example, the Geometry Buffer Creator can be instanced twice so that each instance works on a subset of the application data. The split would be best done by object’s material, this way the shader swapping in the GPU is kept to a minimum. Another possible split is to calculate the shadow maps for different shadow casting lights in parallel.

With a further increase in cores it will become harder to divide rendering work in an efficient way, as there are a limited number of materials used by objects or shadow casting lights at a given frame. So an option for the job-based architecture could be to start handling some real time raytracing for some effects like reflections and refractions.

6. Results

The tests were made on a Core 2 Duo E7200 CPU with an ATI Radeon HD4750 GPU. Microsoft's DirectX March 2009 SDK was used. Note that the DirectX11 version in this SDK is a tech preview. The hardware and drivers used are DirectX 10 level. DirectX 11 level GPUs are not yet available in the market.

The test application did not do any update to the objects in the scene so that the frame time was completely used by the rendering system. Even though, the objects did not move or update they were treated as dynamic. The application sent to the rendering system the objects that compose the scene shown in Figure 5.

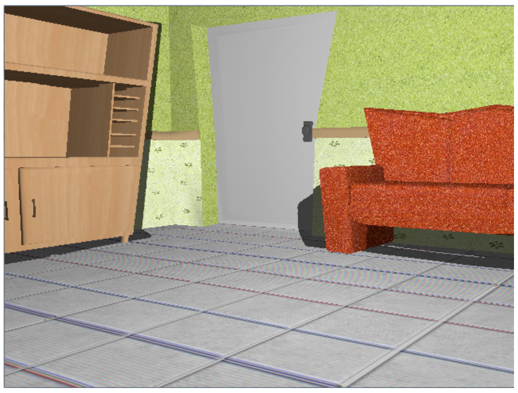

Figure 5: Test scene

Two scenarios were created to compare the system to the traditional ones. The first scenario consisted of using a single thread while the second one consisted in using the multithreaded solution.

In the first scenario the render processes that represent the deferred shading pipeline were executed in a sequential way in the main rendering thread with straight communication to the immediate context. This way is how traditionally games submit rendering work to the GPU.

In the second scenario, the Graphics Manager was initialized to work with 3 threads. At the start of each frame the render processes were queued in the Graphics Manager, which in this particular case used a Win32 thread pool as part of its implementation. When all of them were finished building their respective command buffers the main thread would submit these to the GPU through the immediate context.

The experiment was repeated with five variations. These variations were related to scene complexity. By scene complexity we mean the number of objects drawn. The tests were done with: 7, 16, 106, 1006 and 2006 objects. It is worth mentioning that the final image of the scene did not vary in the different tests as the added objects had the same position and mesh that the original ones. This way the Z-rejection hardware of the GPU would cull the objects before they reached the more processing intensive pixel shader stage. Doing this allowed our application to be CPU bound in the tests 1006 and 2006 objects.

The following chart (figure 6) shows the frames per second obtained by the rendering engine with the use of the multithreaded graphics manager versus the common single threaded solution. The single threaded results are shown by the green bar titled ST. The multithreaded ones are represented by the red bar titled MT.

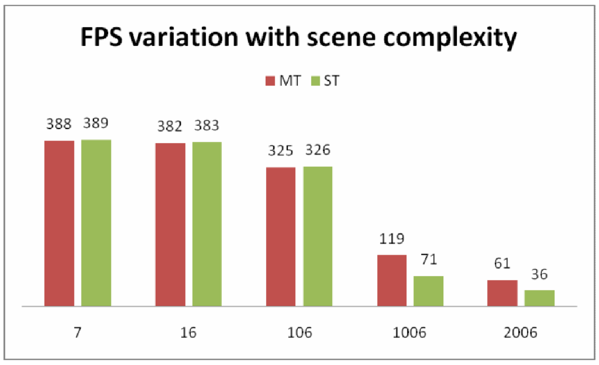

Figure 6: FPS variation with scene complexity.

The first 3 scenarios with 7, 16 and 106 objects show that the multithreaded design does not provide an improvement over the common single threaded one. The multithreaded solution was slightly worse than the traditional one. This is because the multithreaded solution does add a little overhead. Profiling showed that the CPU only used 15% of its capacity. The bottleneck in these tests, therefore, was created by the inability of the GPU to render the polygons faster.

The CPU starts to work harder when the number of objects increases, as it has to issue more draw calls, which take up CPU time. With 1006 objects, the CPU work created by the draw calls becomes sufficiently heavy to shift the bottleneck from the GPU to the CPU. Profiling of the single threaded scenario showed that one the cores was working 4 times more than the other, which was only running other program's processes in the back. The frames per second were in this scenario were 71 in average.

When switching to the multithreaded solution, the profiler showed a more even load among the cores and the frames per second rose around 65% to 119. The explanation to this is that, because the command buffers used were native to the graphics API, the load that each API call adds to the CPU was now being distributed along two cores.

With 2006 objects the ratio of frames per second between the two models only rose 2%. This evidenced that the two cores had hit their limit.

7. Conclusion

In a low scene complexity scenario the benefits of distributing the load of graphic API calls among multiple cores is very low compared to the added overhead of running a more complex multithreaded system. Also, without other systems running there is no stall caused to them by the single threaded graphics system. So in this scenario the multithreaded solution has a clear disadvantage. With the test results showing a very slight decrease in performance for the multithreaded system. It is promising that with other systems running the application will have a better performance using a multithreaded graphics system than single threaded one.

In the high scene complexity scenario the results show the multithreaded design as clear winner with a 65% increase in speed. This lead would certainly increase with in an application that utilized other systems. The speedup shown by the results is significant considering that the API is still immature and that hardware and drivers were not the optimal for the multithreaded solution.

The speedup of the proposed graphics system will never be linear as there is still a part of the process that is single threaded, however, it is clear by analyzing the results that it is faster and more scalable than traditional ones.

In conclusion the multithreaded rendering solution based on a deferred rendering approach provides a promising solution for applications that need high performance and quality graphics.

Acknowledgements

The authors would like to thank André Luiz Brandão for providing scene geometry to test the architecture.

References

DEERING MICHAEL, STEPHANIE WINNER, BIC SCHEDIWY, CHRIS DUFFY, NEIL HUNT. "The triangle processor and normal vector shader: a VLSI system for high performance graphics". *ACM SIGGRAPH Computer Graphics* (ACM Press) **22** (4): 21–30. 1988.

PHARR MATT, FERNANDO RANDIMA. *GPU Gems 2*. Addison-Wesley Professional. 2005.

NGUYEN HUBERT. *GPU Gems 3*. Addison-Wesley Professional. 2007.

SCHEIB VINCENT. *Practical Parallel rendering with DirectX 9 and 10*. GameFest 2008. [online]. Available from: http://www.emergent.net/Global/Downloads/GameFest2008-ParallelRendering.pdf [Accessed July 23, 2009].

POLICARPO FABIO, FONSECA FRANCISCO. *Deferred Shading Tutorial*. SBGAMES 2005. [online]. http://www710.univ-lyon1.fr/~jciehl/Public/educ/GAMA/2007/Deferred_Shading_Tutorial_SBGAMES2005.pdf [Accessed July 23, 2009]

AKENINE-MOLLER TOMAS, HAINES ERIC, HOFFMAN NATY. *Real-Time Rendering*. Third edition. AK Peters. 2008.

HEIRICH ALAN, BAVOIL LOUIS. Deferred Pixel Shading on the PLAYSTATION 3. [online] http://research.scea.com/ps3_deferred_shading.pdf [Accessed July 23, 2009]

BRABEC STEFAN, ANNEN THOMAS, SEIDEL HANS-PETER. *Shadow Mapping for Hemispherical and Omnidirectional Light Sources*. [online] http://www.mpi-inf.mpg.de/~brabec/doc/brabec_cgi02.pdf [Accessed July 23, 2009]

GABB HENRY, LAKE ADAM. *Threading 3D Game Engine Basics*. November 17, 2005. [online] http://www.gamasutra.com/features/20051117/gabb_01.shtml [Accessed 3 September 2009]

ANDREWS JEFF. *Designing the Framework of a Parallel Game Engine.* February 25, 2009. [online] http://software.intel.com/en-us/articles/designing-the-framework-of-a-parallel-game-engine/ [Accessed 3 September 2009]

LEE MATT. *Multi-Threaded Rendering for Games.* GameFest 2008. [online] http://www.microsoft.com/downloads/details.aspx?FamilyID=DA8816C2-CFBE-4208-8DD8-9DEEA0C2E2B5&displaylang=en [Accessed 3 September 2009]

LEWIS IAN. *Multicore Programming Two Years Later.* GameFest 2007. [online] http://www.microsoft.com/downloads/details.aspx?FamilyID=DA8816C2-CFBE-4208-8DD8-9DEEA0C2E2B5&displaylang=en [Accessed 3 September 2009]

DAVIES LEIGH. *Optimizing DirectX on Multi-core architectures.* Game Developers Conference 2008. [online] http://software.intel.com/en-us/videos/optimizing-directx-on-multi-core-architecture-part-1/ [Accessed 4 September 2009]

CALVER DEAN. *Photo-realistic Deferred Lighting.* July 31, 2003. [online] http://www.beyond3d.com/content/articles/19/1 [Accessed 5 September 2009]