Introduction to Multithreaded rendering and the usage of Deferred Contexts in DirectX 11

Rodrigo B. Pinheiro, Alexandre Valdetaro, Gustavo B. Nunes, Bruno Feijo, Alberto Raposo
Pontificia Universidade Catolica - PUC-Rio

Abstract

In this tutorial we intend to cover one of the innovations brought by DirectX11. Previous versions of DirectX didn't support native multithreading and most part of the API was not thread-safe. The user needed to add mutexes in the code to avoid race conditions in order to support a multi-threaded renderer. Moreover, the lack of native support wouldn't properly manage the swap of render states by multiple threads. This kind of guarantee can also be an application requirement.

With the support of Deferred Contexts in DirectX11 the game engine can properly avoid the overhead on the submission thread being the bottleneck of the application. One may now queue API calls through command lists and multiple threads to be executed later. The API is now responsible for inter-thread synchronization for final submission to the GPU. The main goal behind multithreading is to use every cycle of CPU and GPU without making the GPU wait, which impacts the game frame rate.

The full improvements set of DirectX 11 will be showed briefly, then the explanation of why states of the API must be synchronized for a proper rendering will follow. A considerable part of the course will explain how to use the Deferred Contexts and how to properly build command lists for later submission. The focus will be in the comparison to previous APIs, highlighting the issues of the previous versions that the new DirectX11 improvements had arisen from.

Then we present some samples of code and cases that would have good performance improvements with the adoption of Deferred Contexts. During the samples exhibition, the important parts of the code should be discussed briefly at a high level of abstraction in order to give some consistency to the knowledge of the audience.

This tutorial is a sequence of SBGames 2010 course entitled: "Understanding Shader Model 5.0 with DirectX11" [Valdetaro et al. 2010]. In that tutorial we presented other set of innovations brought by DirectX11, which is the Tessellator pipeline.

Keywords:: DirectX 11, Multithreaded, Deferred Context, Immediate Context, Shader Model 5

Author's Contact:

{rodrigo, alexandre, gustavo}@xtunt.com
bfeijo@inf.puc-rio.br
abraposo@tecgraf.puc-rio.br

1 Introduction

One of the most important capabilities introduced in the DirectX 11 API is around multithreading. The number of cores in PCs have been increasing significantly in the past few years. Developers started to seek solutions for spreading the computation of a game among the available cores. Tasks such as physics or AI already could use parallel paradigm to take advantage of multiple cores, but mainly, rendering tasks were only done in a single-thread. Although one could implement a multi-threaded with past rendering APIs(DirectX9 and DirectX10), there were lots of syncronization work that must be guaranteed by the application in order to function properly. DirectX11 API was specifically designed to handle the syncronization issues for a multi-threaded application. The concept of a deferred context was created. With this new concept one may call many API functions in a thread-safe environment. In this tutorial our intent is to explain this new DirectX11 feature and enlighten others with some examples and cases where a multi-threaded approach may improve rendering and loading performance for games and 3D applications.

2 DirectX11 Improvements

This section shows a quick review of the main improvements brought by DirectX11. Later we will focus on DirectX11 deferred contexts which is the main purpose of this tutorial

2.1 Compute Shader

The Compute Shader technology is also known as the DirectCompute technology. It is the DirectX11 solution for GPGPU, one mayfind it easir to use this solutions instead of others(ex. CUDA, OpenCL) due to its tight integration with DirectX11 API and not being necessary to add more dependencies to the project. With this technology programmers are able to use the GPU as a general processor. This provides more control than the regular shader stages for GPGPU purposes such as global shared memory. With the full parallel processing power of modern graphics at hand, programmers can create new techniques that may assist existing rendering algorithms. For example, one may render output an image from the bound render target to a compute shader for a post-processing effect.

2.2 Tessellation

The new graphics pipeline provides a way to adaptively tessellate a mesh on the GPU. This capability implies that we will be trading a lot of CPU-GPU bus bandwidth for GPU ALU operations, which is a fair trade as moderns GPUs have a massive processing power, and the bandwidth is constantly a bottleneck.

Aside from this straightforward advantage in performance, the Tessellator also enables a faster dynamic computations such as: skinning animation, collision detection, morphing and any per vertex transform on a model. These computations are now faster because they use the pre-tessellated mesh which is going to be a device object contexts tessellated into a highly detailed mesh later on. Another advantage of the Tessellator usage, is the possibility of applying continuous Level-of-detail to a model, which has always been a crucial issue to be addressed in any rendering engine. For a detailed introduction to the Tessellation stage please refer to [Valdetaro et al. 2010].

2.3 Multithreading

When older Direct3D versions had been released, there was no real focus on supporting multithreading, as multi-core CPUs were not so popular back then. However, with the recent growth on CPU cores, there is an increasing need for a better way to control the GPU from a multithreaded scenario. DirectX11 addressed this matter with great concern.

Asynchronous graphics device access is now possible in the DirectX11 device object. Now programmers are able make API calls from multiple threads. This feature is possible because of the improvements in synchronization between the device object and the graphics driver in DirectX11.

DirectX11 device object has now the possibility of extra rendering contexts. The main immediate context that controls data flow to the GPU continues, but there is now additional deferred contexts, that can be created as needed. Deferred contexts can be created on separate threads and issues commands to the GPU that will be processed when the immediate context is ready to send a new task to the GPU.

3 Process and Thread

We will briefly introduce the concept involved in process and threads before starting with DirectX11 API.

3.1 Process

Process is the structure responsable for the maintenance of all the needed information for the exeution of a program. A process stores the information about hardware context, software context and addressing space. Those information are important inside a multitask environment were many processes are being executed concurrently. In that manner, it is needed to know how to alternate between them without losing of data. However, the swap between process is costly, so the concept of multiple threads for a single process is introduced. Each process is created with at least 1 execution thread, although more threads may be created for the same process.

3.2 Thread

Thread is an execution line inside a process. Although they have different hardware context, each execution line inside a process has the same software context and shares the same memory space. In that way the cost generated by the information exchance between the threads is much less than the information exchange between processes.

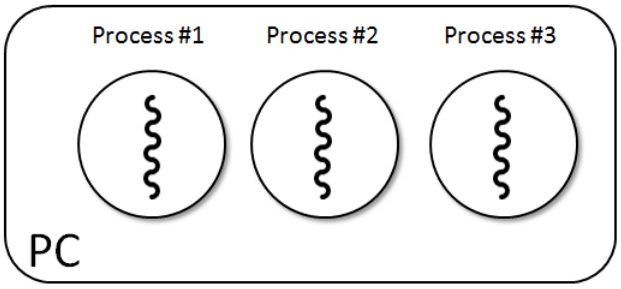

Figure 1: *Multiples processes with single thread*

3.3 Multithreaded

The process, in a multithreaded environment, has at least one execution thread. It may share the address space with other threads that may be fastly concurrently executed in the case of multiple processors. With this approach, computers with many cores are capable to have a performance increase, executting tasks in parallel. However, it is needed to consider how the access of shared resources is made among the threads. This kind of control is necessary to avoid that a thread change data of a shared resource while another thread is still using old data. This kind of guaranteed is called thread safe.

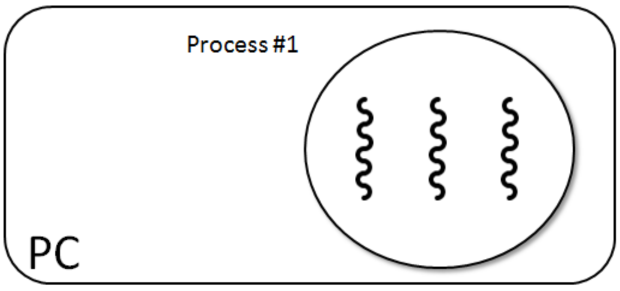

Figure 2: *Single process with multiples threads*

3.4 Thread-safe

Thread-safe is a commonly used technique when we are in a multithreaded application. This technique is used to ensure that a particular snippet of code of your program, when executed by a thread #1, does not interfers in the shared data of another thread #2. In other words, multiple threads can run concurrently with the assurance that they will not modify the shared data that they have in common. Moreover, if we are in an environment with multiple processors, these threads can be executed simultaneously and not only concurrently.

4 Threading Differences between DirectX Versions

In DirectX9 and DirectX10 it was possible to set one multithreading flag making some API methods thread-safe. However, when they becomed thread safe, some syncronization issues needed to be respected by the application and it was necessary to use synchronization solutions (such as mutexes) to turning some critical code sections thread-safe and prevent it from being acessed from more than a thread on a given time. Sometimes this syncronization overhead was so significant that the usage of multiple threads in the previous rendering APIs were completely avoided.

DirectX11 API has a buil-in syncronization system that is not dependent on the application. The runtime is responsable for syncronizing threads for the application allowing them to run concurrently. This improvement turned the DirectX11 syncronization solution much more efficient than previous DirectX thread-safe flags.

5 Multithreading in DirectX11

In DirectX11 the use of the ID3D11Device interface is thread-safe. This interface may be called by any number of threads concurrently. Its mainly purpose is the creation of resources, like vertex buffers, index buffers, constant buffers, shaders, render targets, textures and more. With the ID3D11Device the application is also able to create a ID3D11DeviceContext which is NOT thread safe and one device context should be created for each core. There is only one ID3D11DeviceContext which is called the Immediate Context, this context is the main rendering thread, it is this thread that submits the renderization call to the pipeline. The others ID3D11DeviceContext that might be created are called Deferred Contexts, they work by saving command lists that will be later called by the main thread (Immediate Context). Please see Figure 3.

There are two main improvements that might be used with multiple threads in DirectX11: Parallel Resources creation and Command Lists recording. The first is achieved with the usage of the ID3D11Device by multiple threads. The later and most important is achieved with the usage of Deferred Contexts.

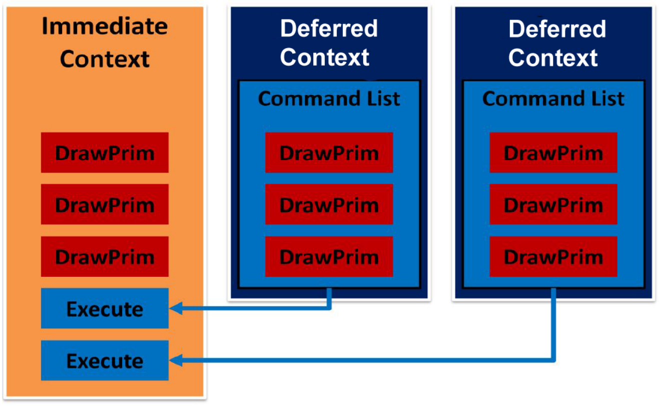

Figure 3: *Deferred Contexts recording command lists that are executed by the immediate context thread after.[Lee 2008]*

5.1 Resources Creation

Being the ID3D11Device a thread-safe interface, the resources for the application may now be loaded in parallel. The main advantage of this feature is for static and dynamic loading.

5.1.1 Static Loading

With the evolution of 3D graphics rendering systems, many games might have to load lots of resources before properly beginning the rendering. This leads to the (many times) annoying loading periods. With DirectX11 the application is able to split the resource workload among the available cores.

One must remember that concurrent loading resources does not always lead to a performance improvement. For example, loading big textures from file has a heavy bottleneck in the memory bandwith and the CPU is many times idle during this process. If an application split all its heavy texture creation work among many cores it may experience no improvement in speed at all when compared to a single thread solution. However, splitting shader creation would probably give a great increase in performance because such operations are CPU intensive. This is specially true in modern game engines that have a high permutation of shaders(uber-shaders) that leads to, in some cases, thousands of shaders resources that needs to be created.

5.1.2 Dynamic Loading

Many games such as flight simulators, RPGs or sandbox-style games have a big outdoor environment, lost of geometry and textures. Loading all the necessary resources to run the entire game at the beginning of the application might not be an option. Such games need a fast dynamic loading of resources when players change between areas. A slow area transition may frustate the immersive experience of the gamer.

DirectX11 API can help with dynamic loading of resources in multiple threads. As stated above, the ID3D11Device is thread safe, and when the player is entering a new area the application may start loading textures, shader and all the resources of that given area in parallel. That feature may be a powerful tool to avoid lower FPS while changing between areas.

5.2 Recording Command Lists

The DirectX11 is basically divided in the following pipeline stages: Input Assembler, Vertex Shader, Hull Shader, Tessellator, Domain Shader, Geometry Shader, Stream Output, Rasterizer and Output Merger. Besides the Tessellator which is configured basically by the Hull Shader, every pipeline stage must have their resources and configurations set by API CPU calls. Making many different CPU API calls may become a bottleneck for the CPU and thus the GPU will get idle frequently. With the introduction of Deferred Contexts and Command Lists, the API calls may be recorded in parallel to be executed later by the main thread.

Usage of Command List is the true hidden treasure of the DirectX11 API. Basically, the application should create one deferred context for each thread besides the immediate context thread. Those deferred contexts may NOT directly invoke the API, it can only store commands in a Command List to be later executed by the thread with the immediate context. Commands stored in a Command List are not executed promptly, they will be only executed when the immediate context calls to execute the command list.

One may ask, what is the advantage of just recording the commands and executing later in the main thread, instead of just executing them later in the main thread anyway. The difference is that the within the command lists, the API calls are heavily optimized and calling them from a command list is much faster then calling separate commands from the main thread.

Figure 4 shows how a single thread approach would be. The APIs states are sequentially set and then drawed by the single thread. Figure 5 shows the approach using a deferred context. All the API states of each drawcall are independently recorded by each core, after all cores are done recording commands the main thread may execute using the ExecuteCommandList() function.

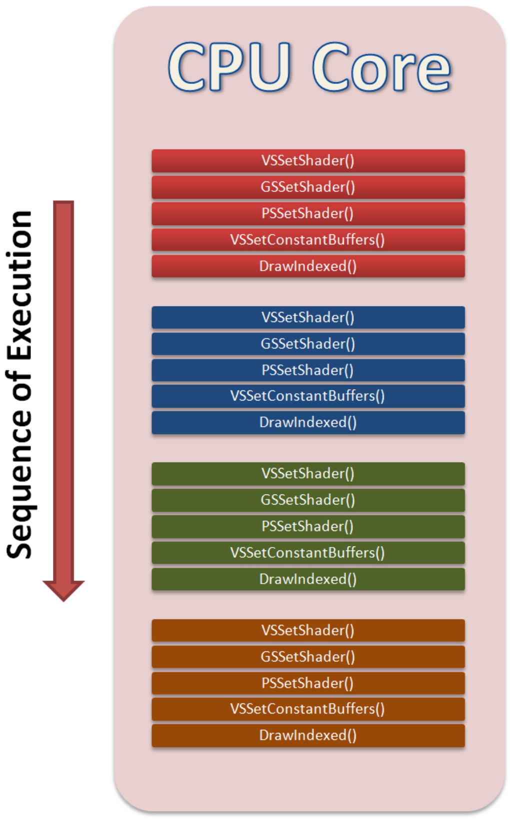

Figure 4: *Draw submission (single thread).[Jason Zink 2011]*

With Command Lists the programmer is able to set many device states such as shaders, textures and rendertargets in parallel saving a lot of CPU time.

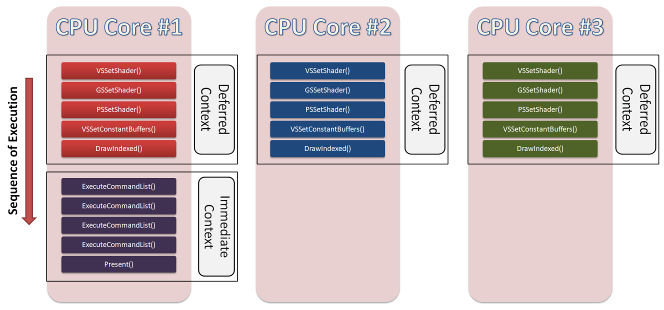

Figure 5: *Threads and device contexts.[Jason Zink 2011]*

Figure 6 shows two deferred contexts issuing commands and finishing them. Then the runtime does the inter-thread sync and the main threads executes the command lists that set the pipeline states.

6 Rendering with DirectX11

As explained in previous sections, DirectX11 has some new major features that helps the user to make their game engines multithreaded such as free threaded asynchronous resource loading. Besides new features, there are also lots of changes in the Graphics Device API if compared to previous DX versions. All these changes have been made in order to facilitate the usage of the device in a multithreaded environment.

In this section, we will walk through this new API, explaining step by step how to create a DirectX11 application that uses deferred contexts.

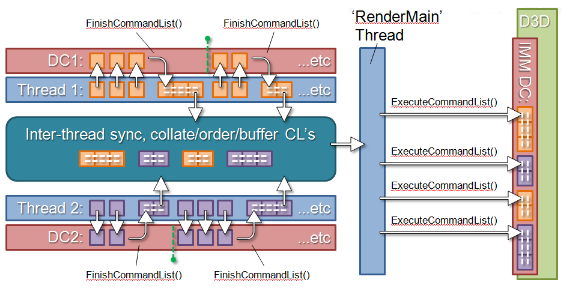

Figure 6: *Sequence of execution[Jansen 2011]*

6.1 Device Contexts

Prior DirectX versions kept all the rendering functionality inside the D3D device. DirectX11 separates out much of the core rendering functionality into a new interface called the D3D device context. As already mentioned in previous sections, D3D device contexts can be one of two types: immediate or deferred. The actual context type is completely transparent from the user's point of view. The rendering code will be called from both identically.

There is only one immediate context for a device, and it represents exactly the rendering API that has been separated from the device, it is responsible for submitting commands directly to the device driver, as in traditional rendering. There can be any amount of deferred contexts, however that amount is generally less or equal than the number of logical cores. These contexts can receive rendering commands as well, but instead of executing them on the device, they batch up commands for inclusion in a command list; the command list can be executed by the immediate context at any time, possibly running on a different thread.

6.2 Command Lists

The command lists are logical arrays containing recorded rendering commands. These commands can be played back just for simplicity and reduction of runtime overhead, so you could pre-record complex rendering ahead of time, while loading a level for example.

Although interesting, this pre-recording scheme is hardly ever useful. The actual usefulness of the command lists lie in multithreading, where the rendering commands are recorded in different threads and then played back in the submission thread. Moreover, the complex rendering tasks get scaled across multiple threads.

7 Using the DirectX11 API

7.1 Creating Device and Checking Multithreading Support

The interface for immediate and deferred contexts is ID3D11DeviceContext. So lets start by creating our immediate context. In order to create our imediate context, the DirectX11 device abstraction ID3D11Device must be created. So we call D3D11CreateDevice.

```
HRESULT D3D11CreateDevice(
  __in   IDXGIAdapter *pAdapter,
  __in   D3D_DRIVER_TYPE DriverType,
  __in   HMODULE Software,
  __in   UINT Flags,
  __in   const D3D_FEATURE_LEVEL *pFeatureLevels,
  __in   UINT FeatureLevels,
  __in   UINT SDKVersion,
  __out  ID3D11Device **ppDevice,
  __out  D3D_FEATURE_LEVEL *pFeatureLevel,
  __out  ID3D11DeviceContext **ppImmediateContext
);
```

As we can see, D3D11CreateDevice already gives us the the immediate context as well. We can also access the immediate context through the ID3D11Device::GetImmediateContext function. As the function shows, the device has one and only one immediate context , which can retrieve data from the GPU. However, in order to use device contexts and asynchronous thread free resource loading, there is the need to check if there is driver support for it available. so we use the following code after creating the device:

```
D3D11_FEATURE_DATA_THREADING threadingFeature;
device->CheckFeatureSupport( D3D11_FEATURE_THREADING, &↩
     threadingFeature, sizeof( threadingFeature ) );
if( threadingFeature.DriverConcurrentCreates && ↩
     threadingFeature.DriverCommandLists )
    // Application code
```

7.2 Multithreaded Multi-Viewport Scene

A good example of a very simple multithreaded scheme for a game is a local 4 player first person shooter. Every player should have his own viewport, and some games allow up to 4 viewports as seen on Figure 7. In this kind of setup, the resources for each viewport can greatly vary. Consequently, assigning the rendering of each viewport to a different worker thread can greatly speed up the submission pipeline, specially because with DirectX11, the loading of resources is thread free, and can be executed asynchronously. Thus, we can instantiate all the buffers and assets from different threads, and create as many worker threads as desired.

Figure 7: *Call of Duty 2 [InfinityWard and Activision 2005]*

The following pseudocode of the setup of a generic multiple viewport application demonstrate a bit of the basic API of DirectX11.

We start by creating a deferred context for every desired thread (or viewport)

```
ID3D11DeviceContext* deferredContexts[NUM_PLAYERS] = {↩
     NULL}
for( int i = 0; i < NUM_PLAYERS; i++ )
{
    device->CreateDeferredContext( 0, &deferredContexts↩
         [i] );
}
```

We should also create a command list for each thread.

```
ID3D11CommandList* commandLists[NUM_PLAYERS] = {NULL}
```

The command lists have no creation method, the deferred context will handle their creation later on .

Now, with every worker thread set up the rendering can start. Every viewport of the scene to be rendered will have its rendering code executed normally on its own thread using the deferredContext designated to the thread. Remembering that the ENTIRE state of the renderer must be set up for every command list to be executed because it will be reset everytime after an execution. This reset is needed to make sure there is no temporal dependency between different command lists, else it could create unpredictable states of the renderer depending on the application set up. So for example we could give the following commands to a deferred context:

```
deferredContexts[threadNumber]->IASetInputLayout( ↩
    vertexLayout );
deferredContexts[threadNumber]->IASetPrimitiveTopology↩
    ( D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST );
deferredContexts[threadNumber]->IASetVertexBuffers( 0,↩
     1, vertexBuffer, stride, 0 );
deferredContexts[threadNumber]->VSSetShader( ↩
    vertexShader, NULL, 0 );
deferredContexts[threadNumber]->PSSetShader( ↩
    pixelShader, NULL, 0 );
deferredContexts[threadNumber]->Draw( count, 0 );
```

After finishing, it is time to put the rendering code into the command list:

```
deferredContexts[threadNumber]->FinishCommand(
        needRestore, &commandLists[threadNumber] );
```

The first parameter tells the deferred context if it should reset its state or not. If set to TRUE then the context will save its state and restore its state. Keep it set to FALSE unless there is going to be a very similar state afterwards, or it will just cause unnecessary state transitions.

After making sure that all the command lists are available, the submission thread has to execute them all in the immediate context:

```
for( int i = 0; i < NUM_PLAYERS; i++ )
{
    immediateContext->ExecuteCommandList( commandLists[↩
        i] , 0 );
}
```

Just remember to make sure the threads are synchronized, so all the viewports of the scene get rendered every frame correctly.

7.3 Effecient Multi-Pass

The usage of deferred contexts is not limited to multithreading. A very good example of usage is when there is a scene with a spatial structure containing a large amount of objects to be rendered and this scene requires a multiple-pass rendering. In a traditional approach, the spatial structure has to be traversed for every pass, which can be expensive. However, with a deferred context approach, there is the possiblity to make 1 traverse only.

This single traversal implementation is very simple and can be very efficient. First, a deferred context and a command list must be created for every desired pass. Then, the rendertarget of every deferred context must be an input texture for the deferred context responsible for the following pass. For example, DC1 is responsible for pass1 has a rendertarget texture1, DC2 is responsible for pass2 and therefore has texture1 as a shader resource. Preferentially, all the deferred contexts should have the same rendertarget for efficiency. This way, we guarantee that the rendertarget of every pass get passed on to the next pass with only one traversal through the scene.

The following pseudocode shows how to render a scene with shadows with only one pass:

```
shadowImmCtx->OMSetRenderTarget( 1, &↩
    shadowMapTexRTView, NULL );
actorsDefCtx->OMSetRenderTarget( 1, &outputTexRTView, ↩
    NULL );
actorsDefCtx->PSSetShaderResources( 0, 1, &↩
    shadowMapTexSRView );
AODefCtx->OMSetRenderTarget( 1, &outputTexRTView, NULL↩
     );
AODefCtx->PSSetShaderResources( 0, 1, &outputTexSRView↩
     );

// Begin the traversal through the spatial structure
        /* Pass 0: Renders the scene from light`s point↩
            of view onto the shadow map */
        shadowImmCtx->Draw();

         /* Pass 1: Renders the scene from camera`s ↩
            point of view onto the outputTexture */
        actorsDefCtx->Draw();

        /* Pass 2: Renders the AO mask and blend with ↩
            the outputTexture */
        AODefCtx->Draw();
// End the traversal

  // Execute Pass 1
  actorsDefCtx->FinishCommandList( 0, &actorCommandList↩
      );
  shadowImmCtx->ExecuteCommandList( actorCommandList );

  // Execute Pass 2
  AODefCtx->FinishCommandList( 0, &AOCommandList );
  shadowImmCtx->ExecuteCommandList( AOCommandList );
```

8 Conclusion

Since rendering code existed, it has been usually a monolithic flux of states. However, The new API architechture of DirectX11 enables the programmer to switch to a new paradigm of rendering programming. The ability to load resources asynchronously from different threads, during startup or rendering, and the parallelization of the rendering calls greatly simplifies the creation of an efficient rendering engine.

And as the CPU cores number steeply increase, the penalty of having a single thread submitting all the rendering code to the GPU increases as well. However, the cost to implement a effective multithreading system is still steep even with this new set of tools, and not every application is a candidate to benefit from the usage of deferred contexts. To port an exisiting system to DirectX11 API can be very simple if it is a DirectX10 system and a lot harder if it is DirectX9, and the decision to do it should made only if the rendering is CPU bound, specially with work submission.

References

INFINITYWARD, AND ACTIVISION. 2005. Call of duty 2.

JANSEN, J. 2011. Programming directx11 performance gems. *Game Development Conference 2011*.

JASON ZINK, MATT PETTINEO, J. H. 2011. Practical rendering and computation with direct3d 11.

LEE, M. 2008. Multi-threaded rendering for games. *Gamefest*.

VALDETARO, A., NUNES, G., RAPOSO, A., FEIJO, B., AND DE TOLEDO, R. 2010. Understanding shader model 5.0 with directx11. *IX Brazilian symposium on computer games and digital entertainment*.