Practical Thread Rendering for DirectX 9

David Pangerl

4.1 Abstract

This article describes a practical and efficient solution for multi-core processor DirectX 9 rendering. The solution significantly accelerates the rendering of complex scenes on multi-core processors by using double buffered command buffers and dedicated rendering thread to execute the command buffers. Its main advantages over similar implementations are the following:

- It is extremely easy to add into any DirectX 9 application.
- It supports all DirectX functions.
- It supports dynamic buffers.
- It supports D3DX functions (e.g., font rendering).
- It supports miscellaneous functions (e.g., win pix).

The implementation requires the developer only to replace the main device class `IDirect3DDevice9` with the one supplied with this solution (`TDxDevice`).

4.2 Introduction

DirectX 9 is implemented so that all functions are DLL function calls. As such they can not be in-lined and thus require some additional instruction to execute. Additionally, they internally tend to frequently synchronize with hardware and can suffer unpredicted time delays (specific to the application rendering pipeline,

DirectX release and device driver version). On the other hand, the DirectX design itself requires issuing multiple functions for rendering a single primitive (set render state, render target, source streams, vertex shader, pixel shader, textures, and, finally, render), even further worsening the situation. In the end, all this function calls and inherent synchronization constitutes in a significant overhead that our solution eliminates.

In the last decade the processor industry has changed the road from developing faster processor into developing multi-core processors.

Our solution takes advantage of the parallelism offered by multi-core processors by creating a dedicate thread for rendering and transferring all the DirectX work to that thread.

4.3 Solution

The solution itself is very plain and simple; all DirectX functions calls are efficiently queued into the command buffer (together with the copy of all function volatile data), and executed on a dedicated rendering thread.

Functions that require immediate return (e.g., CreateDevice) are buffered into the immediate buffer, synchronized with rendering thread, and executed immediately so that the result can be returned.

The trick is to store the function and function parameters directly into our command buffer. In order to do that we make a virtual class with the `Execute` function that we template into a class which can save parameters. In the main buffer class we use `placement new` to create that template directly into the command buffer in specific offset (see Listing 4.1).

4.3.1 Why is it Faster?

The usual game cycle is divided into two steps: an *update* followed by a *render*. In the update step, all system and game objects are updated and then visualized in the render step. Due to the nature of game objects, the update step is usually internally split into several worker threads to optimize execution. However, the rendering step can not be split since DirectX itself does not support that. Consequently, all rendering commands are executed from a single thread.

This is what we do much faster! In fact, more than 100 times faster. We replace the DirectX 9 device with our own, which buffers all the commands but does so really quickly. We used template functions that are in-lined, producing a very fast buffering. Our buffering functions take in average 17 instructions when parameters are not copied and in average 129 instruction when parameters are copied. Just for comparison; an average DirectX function takes 15,231 instructions.

```
class TExecute {
public:
   virtual void                    Execute() const=0;
};

template <typename TR, typename T0>
class TBufferedFunction1 : public TExecute {
protected:
   typedef TR                      (*TFunction)(T0);
   int                             size;
   TFunction                       Function;
   T0                              Param0;
public:
   TBufferedFunction1(TFunction funct, T0 param0)
   {
      size     = sizeof( *this );
      Function = funct;
      Param0   = param0;
   }
public:
   void                            Execute() const {Function( Param0 );}
};

class TBufferFunction {
protected:
   int                             offset;
   char                            *aBuffer;
public:
   TBufferFunction()
   {
      offset=0;
      aBuffer=0;
   }
public:
   template <typename TR, typename T0>
   void                    Buffer(TR (*funct)(T0), T0 param0)
   {
      typedef TBufferedFunction1<TR,T0>T;
      new( &aBuffer[ offset ] ) T ( funct , param0 );
      offset+=sizeof( T );
   }
};
```

Listing 4.1. The solution.

Once rendering is done (signaled with the `IDirect3DDevice9::Present` function), we check if previous rendering has already finished; we do not want to buffer more than one frame ahead (since DirectX already buffers multiple frames ahead). We then synchronize, exchange buffers, and go to the next cycle (see Table 4.1).

Function	Calls	DX cycles	Out cycles	x times faster
SetRenderState	956678	49	16	3.0
SetVertexShaderConstantF	718149	7910	223	35.4
DrawIndexedPrimitive	624299	894	17	52.1
SetSamplerState	359649	66	16	4.1
SetTexture	334216	72	18	4.1
SetStreamSource	328894	113	17	6.6
SetVertexDeclaration	328846	84	16	5.3
SetPixelShader	261353	141	15	9.1
SetVertexShader	261339	250	21	11.7
SetIndices	134897	133	16	8.4
SetPixelShaderConstantF	120313	118	53	2.2
DrawPrimitive	98162	923	16	57.1
UnlockVertexBuffer	98162	88	15	5.8
LockVertexBuffer	98161	234963	23	10056.2
SetRenderTarget	37398	482	20	24.4
SetDepthStencilSurface	28407	254	17	15.0
Write (write text)	22871	15436	263	58.8
EndScene	22656	160	16	10.1
BeginScene	22656	151	16	9.4
WriteUni (write unicode text)	19542	85354	154	553.0
Clear	16540	1106	33	33.1
LockRect	1438	17743	57	308.9
UnlockRect	1438	13117	16	845.4
SetViewport	1438	992	19	51.5
SetPixelShaderConstantI	1438	180	69	2.6

Table 4.1. Comparison of the most used DirectX 9 functions.

4.3.2 Command Buffer

The command buffer is actually a memory buffer that stores functions and their parameters. We chose to use double buffering since it is a simple synchronization technique that effectively avoids data races by synchronizing threads on one point only.

The command buffer is implemented with the use of templates, which enables easy usage and high optimization level (all buffering code is in-lined).

4.3.3 Immediate Commands

Immediate commands are actually a bit slower than direct DirectX calls since they require synchronization with rendering thread. Such implementation is required

since DirectX 9 requires all function calls to be issued from one thread (the thread that actually created the `IDirect3DDevice9`).

However, the use of these commands is reserved mainly for application startup when all objects are created. It does not influence the main application performance.

It is important to note that a DirectX 9 device can be created in a multi-threaded way, but the implementation is much slower than the one presented in this article.

Here is a list of all immediate functions:

- `CreateDevice`
- `GetAvailableTextureMem`
- `EvictManagedResources`
- `GetDirect3D`
- `GetDeviceCaps`
- `CreateAdditionalSwapChain`
- `Reset`
- `CreateTexture`
- `CreateVolumeTexture`
- `CreateCubeTexture`
- `CreateVertexBuffer`
- `CreateIndexBuffer`
- `CreateRenderTarget`
- `CreateDepthStencilSurface`
- `GetRenderTargetData`
- `GetFrontBufferData`
- `CreateOffscreenPlainSurface`
- `GetRenderTarget`
- `GetDepthStencilSurface`
- `CreateVertexDeclaration`
- `CreateVertexShader`
- `CreatePixelShader`

4.4 Conclusion

The solution described in this article can provide a very easy way to optimize your application. Our test applications showed the optimization up to 200%. However, in a real-time game we achieved only a 15% optimization since the rendering thread was already CPU-bound.

4.5 Future Work

The main focus of our further research will be to separate rendering into several threads (e.g., main scene render, shadows, user interface). The idea is to create sub-command buffers that will able to split the rendering step into several threads.