Threading the OGRE3D Render System

by Jeff Andrews

Threading the OGRE3D Render System

Introduction

OGRE3D (a.k.a. Ogre) is one of the most popular open source 3D engines available. It is a complete and modern general purpose 3D engine that is being used in several commercial products from games to scientific simulations. It has been in development for the past five years contributed to by hundreds of individuals from the open source community. For complete information on Ogre you can visit their website www.ogre3d.org.

However, as great as Ogre is, the one drawback that it has is that it does not take advantage of multiple processors in a system. Intel now has several dual core

products on the market and Hyper-Threading Technology has been available on the Pentium 4 for several years now. Threading Ogre can give it the performance benefit that a second processor offers.

Three different alternatives to threading the Ogre render system are presented in this paper. However, only one of them was chosen for a full implementation given the threading goals described in the following section.

Threading Goals

There are several goals that the threading of Ogre should accomplish. They are to provide an implementation that would:

- Be readily acceptable by the Ogre community by minimizing changes to Ogre's source code,
- Achieve a minimum 25% speed increase in frames-per-second (FPS) over the non-threaded version of Ogre on a dual processor system for applications that have a moderately equal or better balance of CPU and GPU utilization, and
- Be a feasible functional alternative for a majority of Ogre applications measured by no user noticeable on-screen defects for the majority of the demos packaged with the Ogre SDK.

Assumptions

This paper makes mention of several classes and uses several code snippets from within Ogre. It is assumed that the reader is familiar with the Ogre source code or has access to the code to do a quick reference so no attempt is made to describe these Ogre specific items. Knowledge of the various threading concepts by the reader is also assumed so there is very little discussion of those concepts.

Limitations

The implementations discussed here will not scale in performance with more than two processors. The maximum amount of performance benefit will be achieved by a dual processor or dual core system. A future article will discuss how to create a threading queue within Ogre that can be used both internally by Ogre as well as by an application using Ogre to take advantage of more than two processors in a system.

Techniques for Threading the Ogre Render System

There are several areas in Ogre that can be threaded but the area that would see the most benefit from a multi-processor machine is the render system. The render system comprises a huge chunk within Ogre and is somewhat isolated from direct access by the application as Ogre abstracts it as much as possible. There are several locations where threading of the rendering system can be accomplished and are as follows:

1. The call within Ogre main to start the rendering can be placed on its own thread, or

2. A threaded layer can be placed in between OgreMain and the render system plug-in, or
3. The plug-in can thread calls to the graphics APIs.

All three of these methods are viable alternatives with their own pros and cons and so are discussed in this paper.

Threading Won't Benefit Every App

Keep in mind that threading the render system is only beneficial to applications that have a somewhat equal time spent in the application and in the graphics API modules. Being either application heavy or graphics system heavy would not see much benefit from threading.

Technique 1: Threading OgreMain (High-Level Threading)

Threading within OgreMain provides the highest level of threading and also presents the highest level of potential performance gains. This is because there are several things that Ogre needs to do once rendering begins, and it's not just a matter of passing on commands and data to the graphics card. Some examples of this are: determining which camera is the active camera, iterating through all visible objects in the scene, informing all visible objects to render themselves, and many others.

The following diagram (Figure 1) shows the majority of Ogre's render path (some items have been left out for simplicity). Threading from the high-level will cause this entire path to be placed on its own thread.

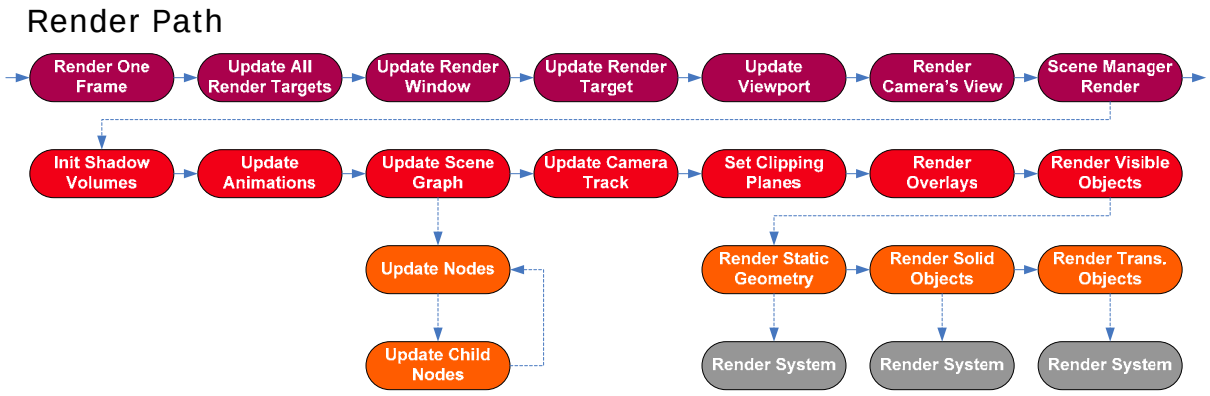

Figure 1: Ogre Render Path

Threading Issues

A major threading issue for this method is that there are several areas in the code where overlap between the two threads will occur. For instance, both the main thread and the render thread would need to access the same member items in each of the Ogre objects in the scene. The main thread could be updating the location and orientation of an object while the render thread is reading that information or it could

happen before the render thread reads it then the rendering would be a frame ahead for that object. This is particularly unsafe when deleting objects where the main thread removes an object that the render system is still using.

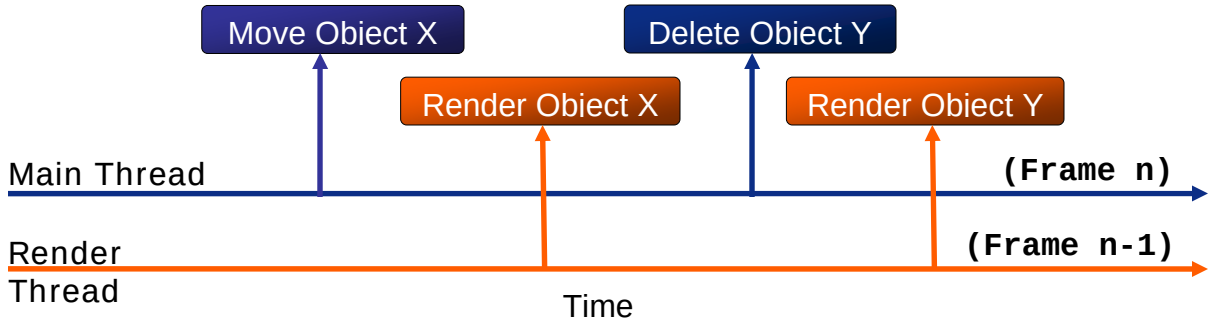

Figure 2: Object Modification Threading Issue

Aside from the mentioned threading inconsistencies, there is also the processor issue of *false sharing*. False sharing arises when a variable being updated by one thread resides on the same cache line of a different variable being updated by a second thread. Since they share the cache line, the one processor needs to evict the entire cache line whenever the other processor makes an update to it. This is a problem for this high-level threading technique because both the main thread and the render thread use the same instance of a class. Since class variables are allocated sequentially in memory they will share the same cache line.

For more information on false sharing and other processor related threading issues, check the Additional Resources section which has links to various threading specific papers.

Avoiding the Issues

In order to avoid the threading issues just mentioned, here are two methods that can be used to do thread safe access and updates to the objects:

1. Use an update queue, or
2. Make duplicate objects.

The *update queue* method would prevent overlapped access to objects by keeping a queue of all updates to all the objects (see Figure 3). The updates would only occur once the main thread was ready to have the rendering thread start rendering again. Of course you would need to wait for the rendering thread to complete first if it hadn't already done so. One drawback to this method is that on single-processor systems the microprocessor would incur the extra overhead of processing a queue without receiving any of the benefits of threading. The other drawback to this method is changes to the hardware resources (i.e. index/vertex buffer, etc.). Queuing changes to these resources would be difficult to accomplish since the task usually involves large amounts of data.

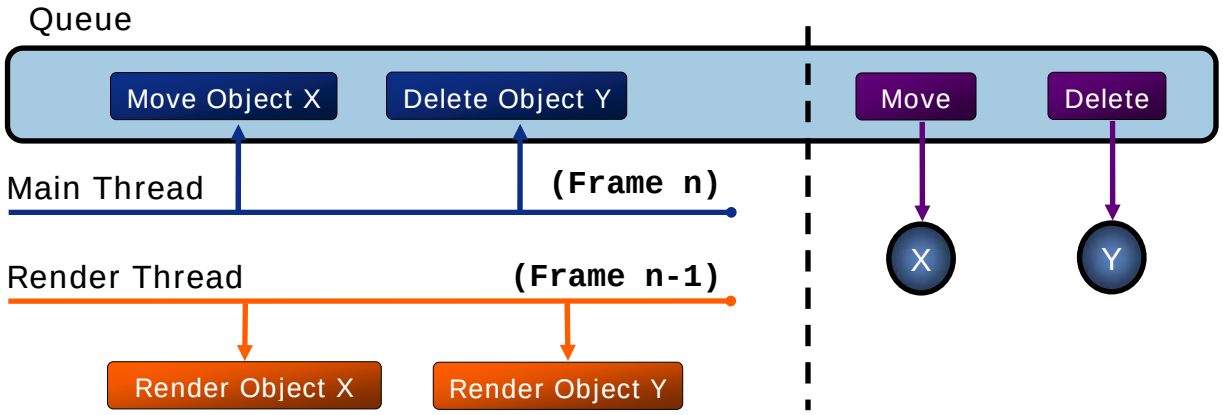

Figure 3: Object Update Queue

The *duplicate objects* method essentially makes a duplicate copy of all objects that are frequently updated. With this technique, the application using Ogre would have to request a duplicate of the object since it is the one that knows which objects are frequently updated. The application would also have to be written in such a way as to know that the object being dealt with is a frame behind what is being displayed. This usually isn't a problem unless the application doesn't update the object every frame then, in that case, the object being accessed may be several frames behind and lead to inconsistencies. Some optimizations can be done so that only the shared items of an object are duplicated reducing the overhead. In this case the object would no longer be a duplicate but would have items that are *double-buffered*.

In Figure 4, notice that the object X will maintain consistency (assuming the update occurs around 30 times a second or higher) but object Y will not be because of the scale that occurred in the previous frame which was not captured in the duplicate object.

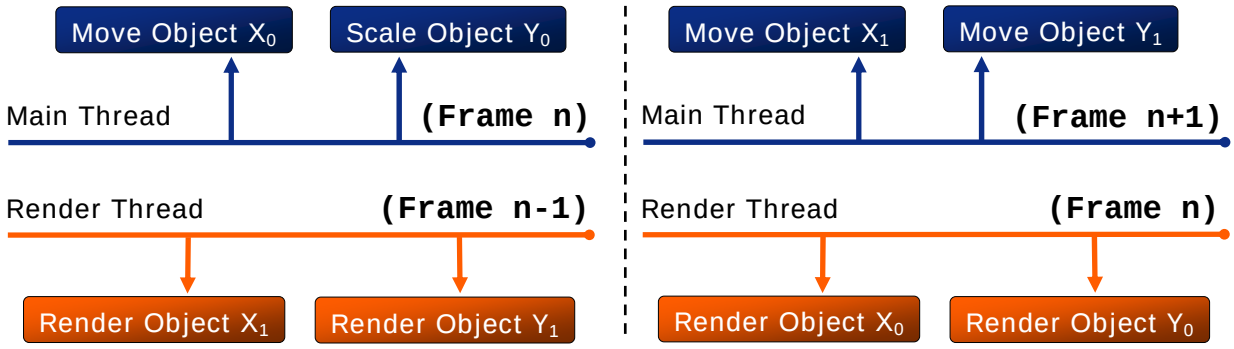

Figure 4: Duplicate Objects

There are some steps I have omitted from both of these techniques to fully synchronize everything. The general concepts of the implementations, however, are covered.

Neither technique would be readily acceptable to the Ogre community because they both require significant changes to the Ogre code and was therefore not pursued. An example of a couple of the items that would need major modifications are the `Frustum::updateView` function and the `_update` function, for all classes that implement it, as these are called in the rendering as well as in other locations within Ogre.

Where to Thread

The most ideal location to place the threading of OgreMain is in `Root::renderOneFrame`. This function makes a call into the main rendering call `Root::_updateAllRenderTargets` which can be easily wrapped around a thread.

Here is some example code of how this could be implemented:

```
	...
	_beginthreadex(0, 0, Root::renderThreadFunc, 0, 0, 0);
	...

/*static*/ unsigned int Root::renderThreadFunc(void)
{
	while(1)
	{
		_waitForStartRendering();
		_setStartRendering(false);

		_updateAllRenderTargets();

		_setRenderingComplete(true);
	}
}

bool Root::renderOneFrame(void)
{
	if(!_fireFrameStarted())
		return false;

	if(mThreadedRendering)
	{
		_waitForRenderingComplete();
		_setRenderingComplete(false);
		_setStartRendering(true);
	}
	else
	{
		_updateAllRenderTargets();
	}

	return _fireFrameEnded();
}
```

Code Example 1: High-Level Threading

The _wait… and _set… functions in this code example are calls to OS dependent synchronization functions. For example, the Windows version of _waitForRenderingComplete would contain a call to WaitForSingleObject. Notice that the _fireFrameEnded function gets called before the actual completion of the rendering when threaded.

Technique 2: Creating a Threaded Render System Layer (Mid-Level Threading)

Creating a threaded render system layer is the most unobtrusive method for threading the render system of Ogre but it is also the most difficult approach due to the complexity of Ogre's render system. The render layer is essentially a wrapper around the render system plug-ins (i.e. Direct3D and OpenGL). Only minor modifications would need to be made within Ogre to incorporate this additional render system, but creating the threaded render system layer is another story.

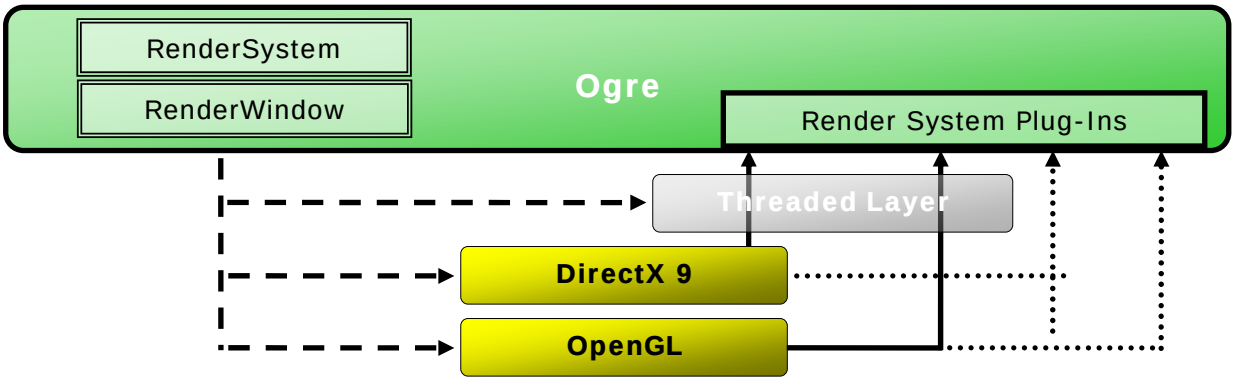

Figure 5: Threaded Layer

In order to create a threaded render system, you would need to implement, at minimum, both the Ogre::RenderSystem class and the Ogre::RenderWindow class located in OgreMain. The two classes would just be a layer in between Ogre and the actual plug-in. This layer would not just be a simple layer that only passes the function calls along to the wrapped plug-in. Several decisions need to be made as to what to do to call the plug-in since the point of this method is to place the rendering on a separate thread. Here are some things to consider when implementing the class functions:

- Initialization routines (i.e. all functions called only prior to the start of rendering) can more than likely call directly into the plug-in. It may be necessary to do some initialization within the wrapper class as well.

- Creation calls will need to wait for the previous frame's rendering to complete before calling the plug-in and may need a wrapper class around the created class that the plug-in returns (provided one exists). A good example of a class needing a wrapper is the RenderWindow class returned by the plug-in. This class is returned by a call to RenderSystem::createRenderWindow.

- Certain functions should not call into the wrapped plug-in. The wrapper class needs to keep track of some items on its own and calling into the plug-in would cause duplicate tracking and may lead to some strange behavior with Ogre since you would want Ogre to access the wrapper's object and not the plug-in's object. The RenderTarget functions are an example of this.
- Certain functions will also need to call the base class. The RenderSystem and RenderWindow classes are required to be called for some of the functions as they do some internal housekeeping for the class. Not doing so will result in erroneous behavior within Ogre.
- Functions used for rendering will need to be queued in order to take advantage of threading. The rendering thread will traverse the queue and call each of the functions in order. The swapBuffers function within the RenderWindow class is a special case. Aside from having its wrapped function be queued it is also the location where the render thread is signaled to start executing the functions in the queue.

While the items mentioned above give a thorough description of what needs to be done to implement this method, there are still several other minor items to consider and corner cases that need to be handled to complete this implementation.

Threading Issues

This implementation shares the same threading issues as the high-level threading method, except in this case the best resolution would be to make copies of the items since this middle layer does not own the classes passed into it from Ogre. There are also some other methods that will need to be borrowed from the low-level threading technique to complete this implementation in order to provide thread safe access to the video card's resources.

Where to Thread

As mentioned earlier, this would be the most unobtrusive method for threading Ogre's rendering system. The only significant changes to Ogre would be adding the threaded render system to Ogre's list of available render systems, as such:

```
void Root::addRenderSystem(RenderSystem *newRend)
{
      mRenderers.push_back(newRend);

#ifdef __THREADRENDERSYSTEM__
      if (mThreadRenderSystem)
      {
            RenderSystem *newTRend =
                  new ThreadedRenderSystem(newRend);
            if (newTRend != 0)
                  mRenderers.push_back(newTRend);
      }
#endif
}
```

Code Example 2: Mid-Level Threading

Technique 3: Threading the Render Plug-In (Low-Level Threading)

Threading a specific render system plug-in yields the lowest flexibility because it is tied to a specific technology (e.g. Direct3D9, OpenGL) but it also lets you have the greatest control over the hardware resources.

The cleanest way to implement this method is to create a layer between the API and the plug-in that handles the threading (see Figure 6). In this way it would just be a matter of replacing the API interface with your layer and then each and every call from within the plug-in would be thread safe since it's now your layer's interface handling all the calls. Doing this for DirectX is just a matter of replacing all the `IDirect3D…` interfaces used with your own classes and methods that match those of DirectX. For OpenGL it would involve removing the OpenGL header files and replacing them with you wrapper functions' header file. It would then require wrapping the OpenGL include files around a namespace within your module to avoid the function name conflicts.

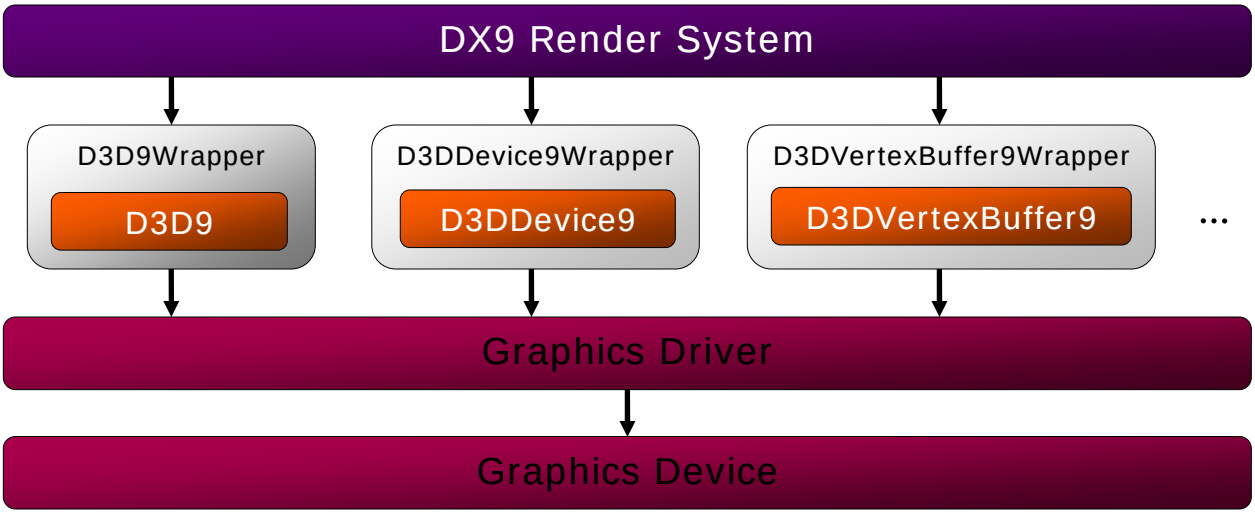

Figure 6: Threaded API Wrappers

This method follows the same threading considerations as the mid-level method with the exception of wrapping the Ogre specific classes as follows:

- Initialization will not need to be threaded
- Creation functions and some get functions will need to wait until the previous frame's rendering is complete before making the calls
- The render commands, and accompanying parameters, will need to be queued in order to take advantage of threading..

Locking Index and Vertex Buffers

As with all the methods, use of hardware resources poses a potential threading issue. The locking of index and vertex buffers, however, is especially challenging because certain applications may repeatedly lock these buffers throughout the course of execution. These buffers, known as dynamic buffers, typically change their content per

frame to alter the image being displayed. Some applications even re-use buffers so that multiple objects will share the same index and vertex buffer space per frame. In order to solve this we will need to "buffer" the buffers. While this can be done is many ways, here are two buffering methods that can be used:

1. Partially buffered locks, is a buffering technique that works somewhat like double buffering. Instead of one buffer being created, two buffers are created in video memory. In this way, while the application is writing to one buffer the render threading is using the other buffer for rendering. This method has the benefit of being fast but also has the drawback of consuming more video memory and not having the ability to handle applications that use the buffer for double-duty (i.e. writing to the buffer more than once per frame), or

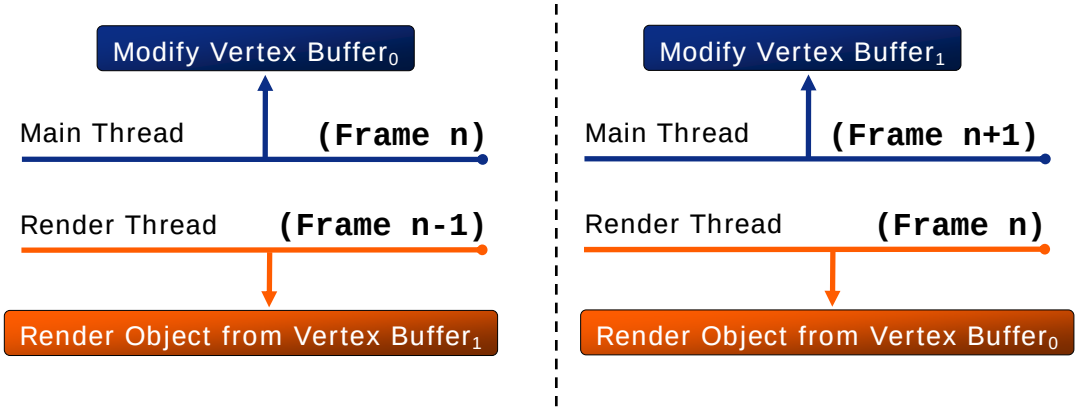

Figure 7: Partially Buffered Locks

2. Fully buffered locks, is a technique that keeps a copy of all locks to the buffer. A local buffer is maintained and all modifications are done on the local buffer. On unlocking the buffer, the data are placed in a parameter queue so that it maintains a unique copy for each lock/unlock pair. While this has the benefit of maintaining accuracy for the buffer, it could perform poorly for applications that lock large amounts of data.

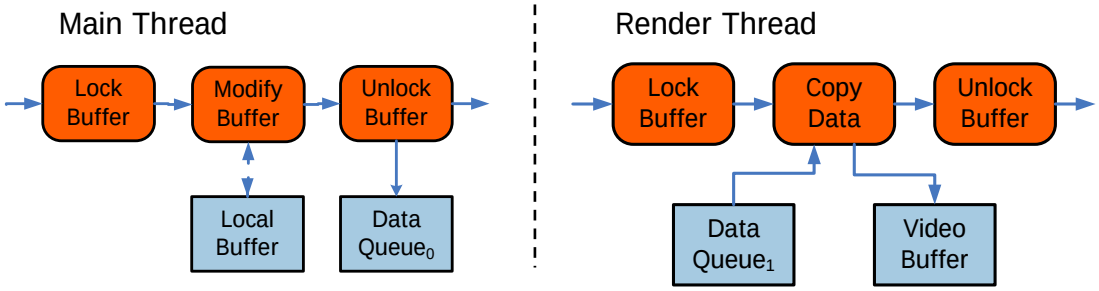

Figure 8: Fully Buffered Locks

Why Aren't Surface Locks Buffered

Surfaces (whether front and back buffers or textures) typically consume a large amount of memory so buffering locks to a surface would be impractical and also unnecessary. This would be impractical because, in the case of the partially buffered locks technique, a video card would quickly consume available memory since two

surfaces would now need to be created instead of just one. For example a 1024x1024 32bpp texture would normally consume 4MB of video memory but if implemented using the partially buffered locks technique would double that to 8MB. The technique would essentially be cutting in half the amount of memory available for textures. As for the fully buffered locks technique, the implementation would consume a large amount of system memory and spend most its time copying data. Because this technique maintains a local copy of data in system memory it first would have to allocate enough system memory for the surface. On an unlock it would then have to copy the data onto the parameter queue and allocate more memory for the queue if required. The render thread would then need to copy the data from the parameter queue into the actual surface. All these copies would significantly slow down any lock-unlock pairs.

Buffering surfaces locks is unnecessary because applications have no need to read/write directly into a surface (except for textures) as much better performance can be attained from using the video card to manipulate the surface. Textures, on the other hand, are usually only locked once by the application on surface creation to load in the texture information. So in the case of a texture being locked (and the unlikely scenario of the front and back buffer being locked) it would be best to just have the main thread wait until the rendering thread is complete before performing the lock instead of buffering it. Not waiting for the rendering to complete could lead to erroneous data within the surface especially if the surface is being written to by the video card. Keep in mind that the constant locking of surfaces would lose any benefit from threading because of this mandatory wait for the rendering thread to complete.

Where to Thread

Threading for this method would occur only within the wrapper classes that wrap the graphics APIs. Each function would be implemented differently as they would ether call the API directly, wait for the previous frame's rendering to complete before calling the API, or be queued for the rendering thread to call the API. Some optimizations may also be possible so that calls that would normally have a wait would be cached in the wrapper class. You would also need to modify the render system plug-in's code to replace all references to the graphics APIs to that of your implementation.

The following code is an example of how to wrap the IUnknown interface specific to this DirectX implementation.

```
D3D9WrapperUnknown::D3D9WrapperUnknown(
    D3D9WrapperDevice* pD3DDevice
    )
    : m_pD3DDevice(pD3DDevice)
    , m_RefCount(1)
{
};

D3D9WrapperUnknown::~D3D9WrapperUnknown()
{
    if (mRefCount > 0)
    {
```

```
        // Log an error for this.
    }
}

ULONG D3D9WrapperUnknown::AddRef(void)
{
      return ++m_RefCount;
}

ULONG D3D9WrapperUnknown::Release(void)
{
    ULONG   RefCount = --m_RefCount;

    if (!RefCount)
    {
        // Schedule the IUnknown for release and
        //  delete this class.
        if (m_pUnknown != NULL)
        {
            m_pD3DDevice->SubmitRelease(m_pUnknown);
        }
        delete this;
    }

    return RefCount;
}
```

Code Example 3: Wrapping an Interface

This technique met all the threading goals listed in the introduction and was therefore chosen for implementation. The Direct3D9 render system was chosen for the threading implementation and the source code accompanies this paper as a download from the Intel developer website.

Ogre Analysis and Results

This section discusses Ogre's execution profile and results achieved from the low-level threading implementation using Ogre's supplied demos.

The following numbers are for a 2.4GHz Core 2 Duo processor with an Nvidia 7800 video card. Numbers shown are for the 2 different buffing locking methods and 1 for showing the overhead of having a wrapper around the DirectX interfaces but without threading enabled (for single processor systems). A green background indicates scaling of 1.1x or better, yellow indicates scaling slightly below 1.0x, and red indicates scaling below 0.95x. Numbers in red indicate visual artifacts for the partially buffered locks method.

	Speedup vs. Non-Threaded		
	Threaded (No)	Threaded (Partial)	Threaded (Full)
Bezier	0.963	1.196	1.194
BSP	0.957	**1.616**	0.223
BSPCollision	0.967	**1.042**	1.183
CelShading	0.961	1.235	1.235
Compositor	1.000	1.000	1.000
CubeMapping	1.029	1.037	1.038
DeferredShading	1.000	1.000	1.000
Dot3Bump	1.000	1.058	1.113
Dyntex	1.001	1.090	1.248
EnvMapping	0.970	1.195	1.195
FacialAnimation	0.983	1.330	1.330
Fresnel	1.003	1.004	1.006
Grass	1.000	1.000	1.000
Lighting	0.995	1.033	1.054
Ocean	1.000	1.000	1.000
ParticleFX	0.998	1.308	0.232
RenderToTexture	0.994	1.086	1.031
Shadows	1.000	**1.004**	1.001
SkeletalAnimation	1.000	1.000	1.000
Smoke	1.000	0.999	1.000
Terrain	0.998	1.098	1.098
TextureFX	0.999	1.002	1.002
Transparency	1.000	1.061	1.061
VolumeTex	1.000	1.000	1.000
Water	0.979	1.233	1.357

Notice how most items are close to 1.0x scaling and there are also several slightly below but not significantly below. There are several items in green that are showing good scaling but keep in mind that these are just Ogre's demos and the majority of them are not particularly CPU intensive. Only 2 items are in red, both in the *fully buffered locks method*, because these demos are constantly updating the vertex buffers with large amounts of data negating the benefits of threading with all the data copying that needs to be done.

Something that needs pointing out is the Shadows demo. There are 2 shadow techniques, stencil and texture. Only the stencil shadow technique is an issue with the partially buffered locks technique. The texture technique works correctly.

Also note that some of the demos run better on the *fully buffered locks* technique than on the *partially buffered locks* technique. In these cases the application gets back a buffer that it can both read and write. This causes an overhead within the main thread for copying the data from the video memory to system memory. The *fully buffered locks* technique hides this overhead by having the lock/unlock execute in the render thread.

Conclusion

Threading a 3D engine can be challenging but it is most certainly doable. As was shown, threading of Ogre's render system was successfully accomplished and proved to be beneficial to several of the demos. Three different threading techniques were presented as viable alternatives for threading Ogre but only the low-level technique was chosen for implementation to keep within the threading goals described in this paper's introduction. With the combined efforts of the Ogre community, Ogre can be threaded for performance giving applications that are both CPU and GPU intensive additional FPS for a smoother display.