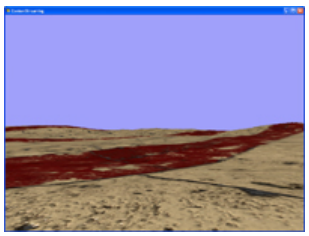
ContentStreaming Sample

Collapse All

The ContentStreaming sample demonstrates streaming content in the background for applications that need to display more data than can fit in video or system RAM at any given time.



Path

Source	*SDK root*\Samples\C++\Direct3D10\ContentStreaming
Executable	*SDK root*\Samples\C++\Direct3D10\Bin*platform*\ContentStreaming.exe

Sample Overview

With games growing more complex and gamers demanding more expansive worlds, more content — and just more game in general — the ability to render a large and seamless world, while staying within fixed memory limits, has become essential. When game-level resources do not fit into GPU memory, or when game data does not fit into system memory, one solution is to reduce the size of these assets. However, this is not always the best solution. Sometimes, we want high-quality *and* high-*quantity*. The ContentStreaming sample attempts to show how best to load only the assets that are visible or will be visible in the near future, while causing the main render thread to stall as little as possible.

Before getting into the details of loading data, this topic describes some underlying concepts that are used by the various streaming methods. These concepts are the *resource reuse cache*, *memory-mapped I/O*, and *placeholder meshes*. After these are explained, various methods of streaming data in on-demand are covered.

Resource Reuse Cache

To limit the number of GPU resources created by the application while loading in new data, the sample uses a *resource reuse cache*. The resource reuse cache tracks a fixed number of GPU resources.

When the application needs to load a resource, it simply looks for a compatible resource in the cache that isn't currently in use and uses that instead. If no resources of the appropriate type can be found, the application can do one of two things: It can either attempt to create a new resource of that type and add it to the resource cache, or it can wait to load the resource until a compatible resource frees up. The advantages of this approach are that it keeps the render thread from stalling while trying to create a new texture, and that the application can keep its memory use to a known value. The main disadvantage is that it puts restrictions on artists to only use textures of a certain size to gain maximum texture reuse. While it isn't strictly necessary to reuse textures, it does cause fewer stalls in the main render thread. The cost of creating a texture increases with the size of the texture.

Memory-Mapped I/O

Memory-mapped I/O differs from conventional I/O in that **ReadFile** is never called to move data into memory. Instead, the file is mapped into the virtual address space of the application and paged in as needed. To use memory-mapped I/O, you open a file as you normally would, by using **CreateFile**. Instead of reading data from this file directly, you then call **CreateFileMapping** on the file handle. This function creates a file-mapping object, but it does not map any data on the disk into the virtual address space of the application. To to this, the application needs to call **MapViewOfFile**, which returns an actual pointer to the virtual address that represents the start of the file.

There are some caveats with using memory-mapped I/O. The first is that the application can run out of virtual address space by specifying ranges that are too large when calling **MapViewOfFile**. For example, consider the following scenario. An application has a 4-GB packed file that contains all of its assets. The application tries to map the entire contents of this file to its virtual address space by calling **MapViewOfFile** and passing in 4 GB as the number of bytes to map. A 32-bit application would fail, as Windows cannot find enough virtual address space to handle a mapping this large. However, a 64-bit application would not have this problem.

To combat the lack of address space on 32-bit platforms, the application should use multiple file mappings when accessing large files. For example, the 4-GB file could be mapped with a sliding window that only maps 500 MB at a time. When the player moves on to the next area, the file mapping would slide that area in the packed file. The ContentStreaming sample uses multiple file mappings within the same file based upon a preset chunk size. Care must be taken to ensure that the mappings do not eat up all of the virtual address space. An LRU scheme is used to determine which chunk to remove from mapping when a new chunk needs to be mapped.

Another caveat is that **MapViewOfFile** can only map data that is aligned on the memory allocation granularity boundaries. On most systems this is 64 K. This has many implications when packing data into a packed file. For a 4-GB packed file with all files tightly packed together, mapping a 500-MB chunk may not work if the start of that chunk isn't on a 64-k boundary. To combat this, the packed file used in this sample creates predefined areas to be mapped called *chunks*. Each chunk contains a grouping of resource files. The chunks are located on 64-k boundaries so that they can be easily mapped using **MapViewOfFile**.

Packed Files

To ensure easy installation, consistent disk access, and compression, game developers often pack resources into a single packed file. The nature of the packed file changes depending on the game. For this sample, we forego compression, and create a simple packed file that consists mainly of assets in the media folder. The packed file is created the first time that the sample is run. It is stored in a per-user directory that is compatible with User Account Control; in this case, in CSIDL_LOCAL_APPDATA + "\ContentStreaming". The packed file contains an index which gives the locations and sizes of all of the files in the pack. Following the index, the resources files are packed together into chunks. The chunk size is set to ensure that the application does not run out of virtual address space when using memory-mapped I/O. As noted above, these chunks must be 64-k aligned to be mapped with **MapViewOfFile**.

At run time, the index of the packed file is loaded into memory by calling **ReadFile**. This is done regardless of whether we're using memory-mapped I/O. When retrieving a packed file, the CPackedFile class first consults the index to see if the file is resident in the packed file. If it is, the CPackedFile class either calls **ReadFile** to fill the supplied buffer with the file data, in the case of normal I/O, or returns a memory-mapped

pointer, in the case of memory-mapped I/O.

This packed file can easily be deleted from within the sample by using the *Delete Packfile* button in the startup UI.

On-Demand Single-Threaded Loading

Single-threaded loading is basically serial loading. When the resource is needed, the application immediately tries to load that resource. No other processing can be done while the resource is loading. To make this type of loading as efficient and painless as possible, the application uses the resource reuse cache. The resource reuse cache allows the application to lock and fill an existing resource instead of creating a new resource from scratch. Locking and filling an existing resource is less costly than creating an entirely new resource.

On-Demand Multithreaded Loading

Although the resource reuse cache does improve the performance of on-demand single-threaded loading, there will still be hiccups in rendering as the application pauses to load the data from disk (or read from memory-mapped I/O) and do the actual copying to the device object. A more optimal solution would cause the rendering thread to pause as little as possible. On Direct3D 9 and Direct3D 10, it is possible for the rendering thread to only stop for two things: locking a resource and unlocking a resource.

There is a second option that exists on Direct3D 10 only. On Direct3D 10, we can ensure that the graphics thread only has to pause for one thing: calling **UpdateSubresource** on an object. This can be enabled by commenting out the USE_D3D10_STAGING_RESOURCES definition at the top of ResourceReuseCache.h. Currently, the application uses staging resources to mimic D3D9 behavior. The acts of loading the resource from disk and copying data to the device objects can be done in separate threads.

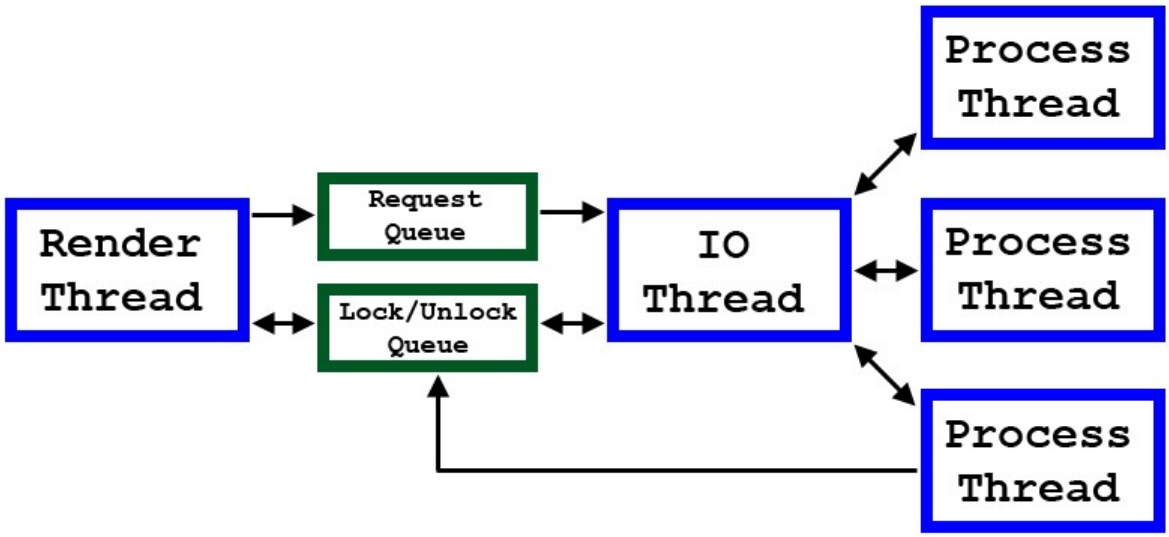
Three types of threads come into play. The first is the *render thread*. The render thread's main job is to draw the scene without stalling. Lesser jobs of the rendering thread include sending I/O requests to the I/O thread by way of the request queue, and handling lock and unlock requests from the I/O thread. To keep the render thread from stalling, both of these lesser jobs are handled when it is convenient for the render thread.

The second type of thread is the *I/O thread*. The I/O thread handles requests from the render thread. It also either loads files from the packed file or gets memory-mapped pointers to them. The I/O thread handles communications with the various process threads and also places lock/unlock requests into the lock/unlock queue for consumption by the render thread. In addition, on Direct3D 9, the I/O thread does the actual copying of bits into the device object. There is generally one I/O thread for each storage device that will be used on the system. Because the I/O thread may potentially spend a lot of time waiting for I/O requests to come back from various storage devices, it may be acceptable to put this thread to share a hardware thread with one of the process threads.

The last type of thread is the *process thread*. Process threads handle the dirty work of processing the data after it has been loaded from disk by the I/O thread. Process threads can decompress data, prefetch data when using memory-mapped I/O, or handle other operations, such as swizzling, that may need to be performed on the data before it is copied into the device object. There should generally be a single process thread for each available hardware thread.

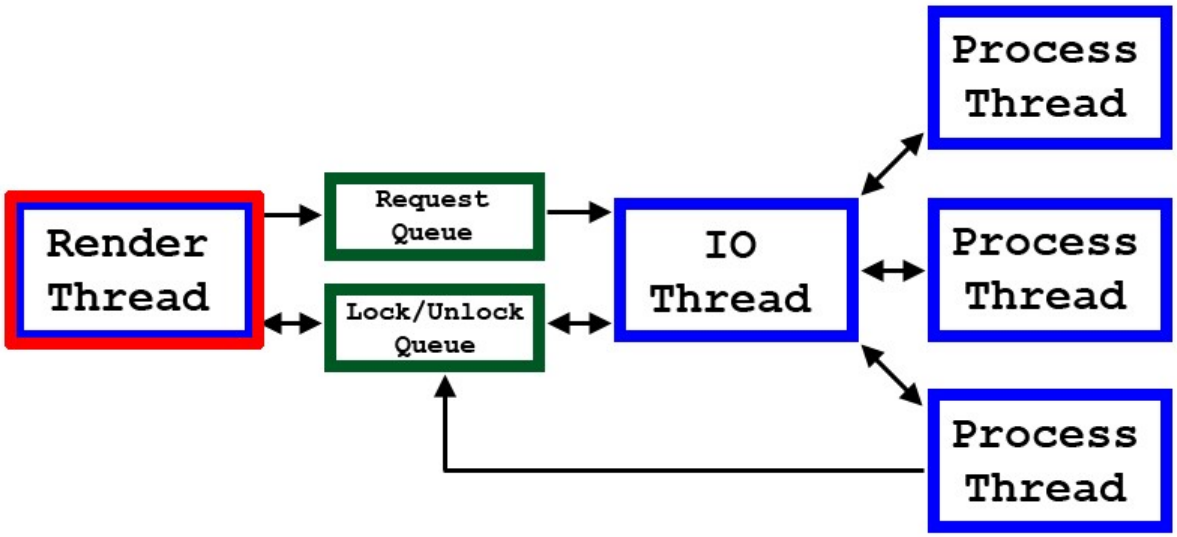
The following diagram illustrates all of the threads used in multithreaded loading.

Figure 1. Threads in Multithreaded Loading

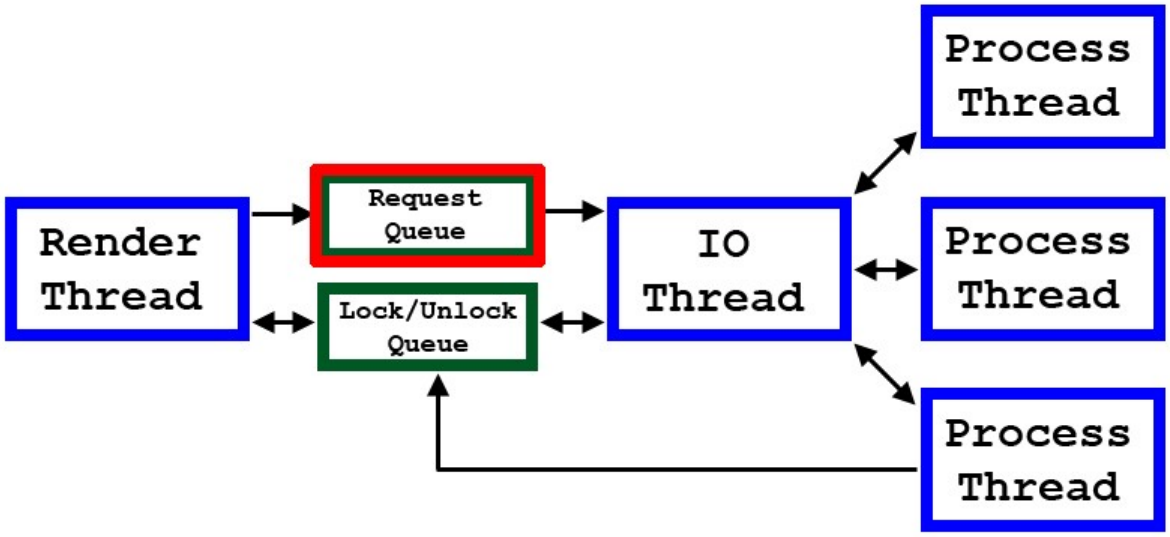


The typical chain of events is as follows:

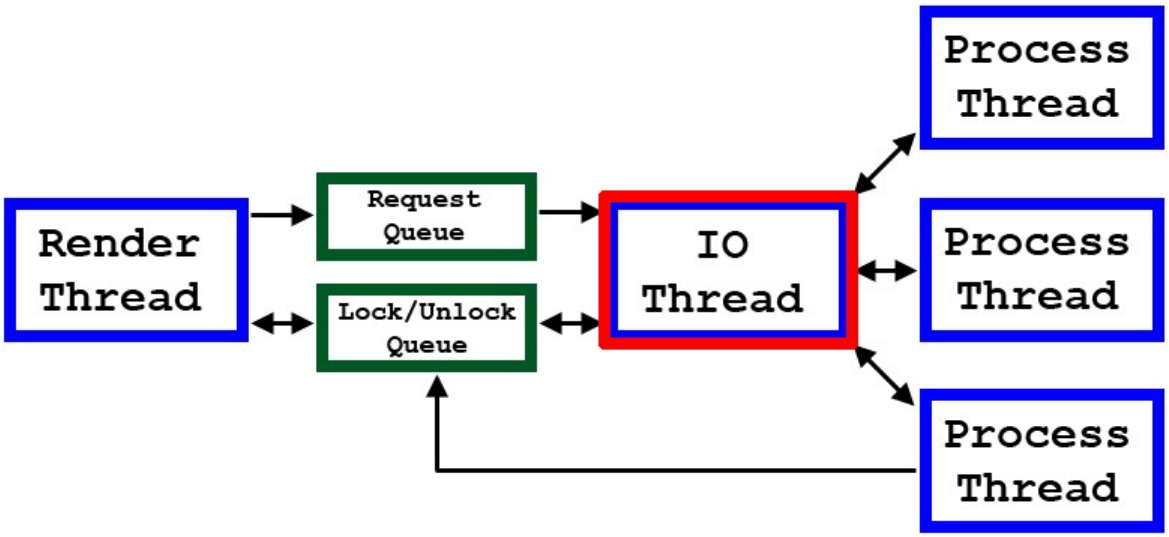
1. The I/O thread and the process threads are sleeping, waiting for something to do. The graphics thread is rendering away.



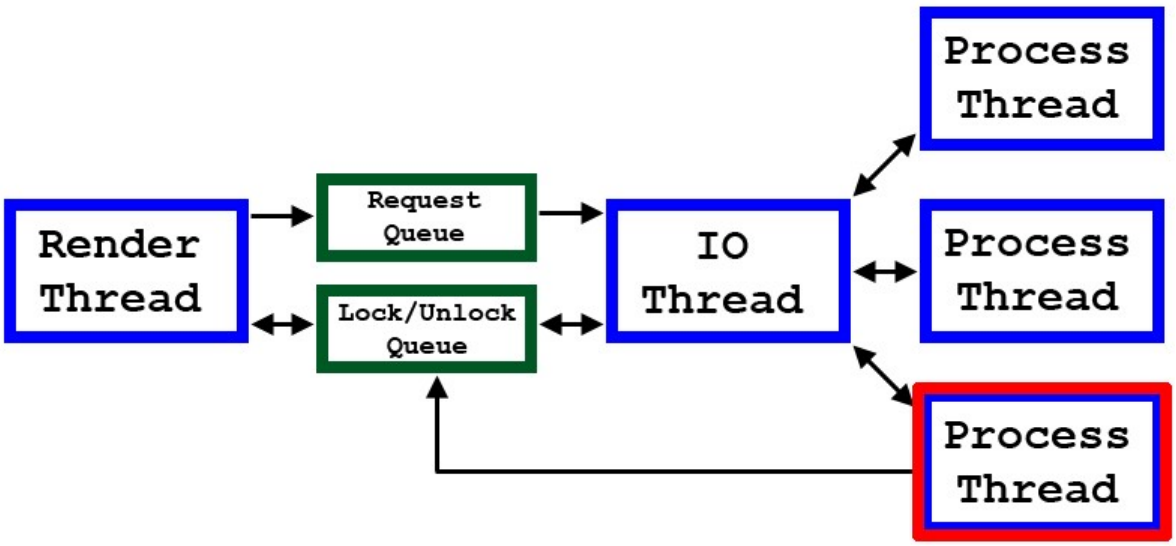
2. The graphics thread determines that a resource is going to be visible sometime in the near future. It places a resource request on the request queue.



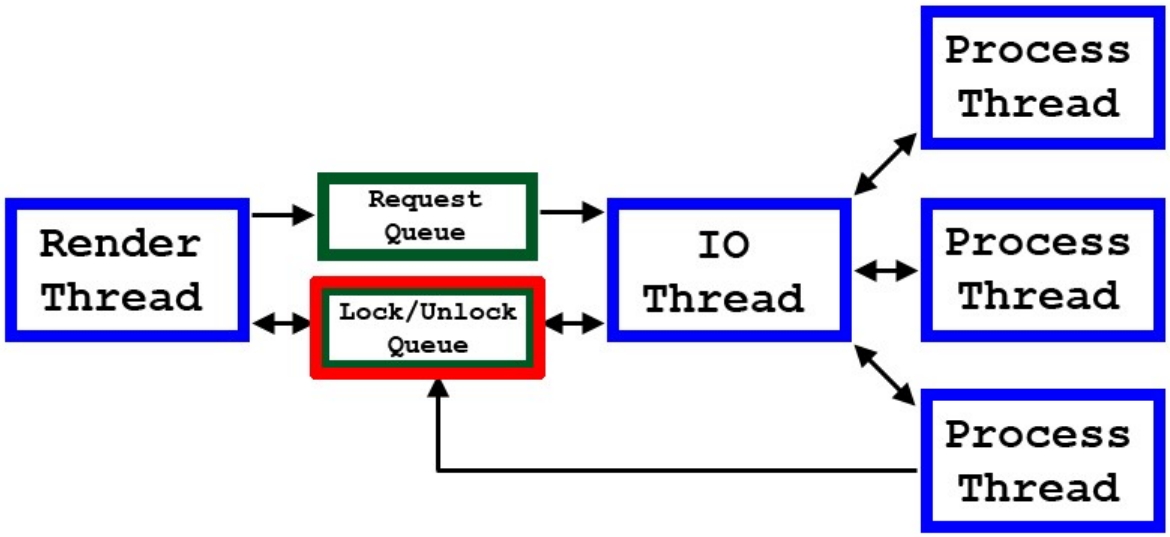
3. The I/O thread picks this request up and loads the data from the packed file. If using memory-mapped I/O, a memory-mapped pointer is returned.



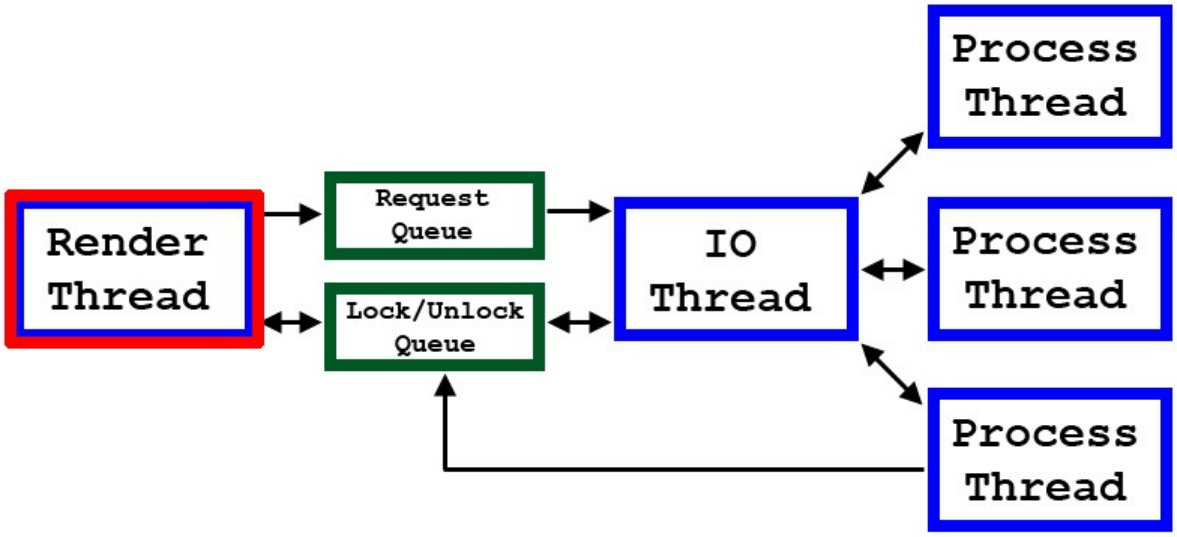
4. The I/O thread then sends a process request to one of the waiting process threads. The processing thread performs some computation on the data. If using memory-mapped I/O, and pre-fetching data is enabled, the process thread will touch 1 byte in every 4 K of the data being processed, to ensure that the resource is fully paged into memory before being used.



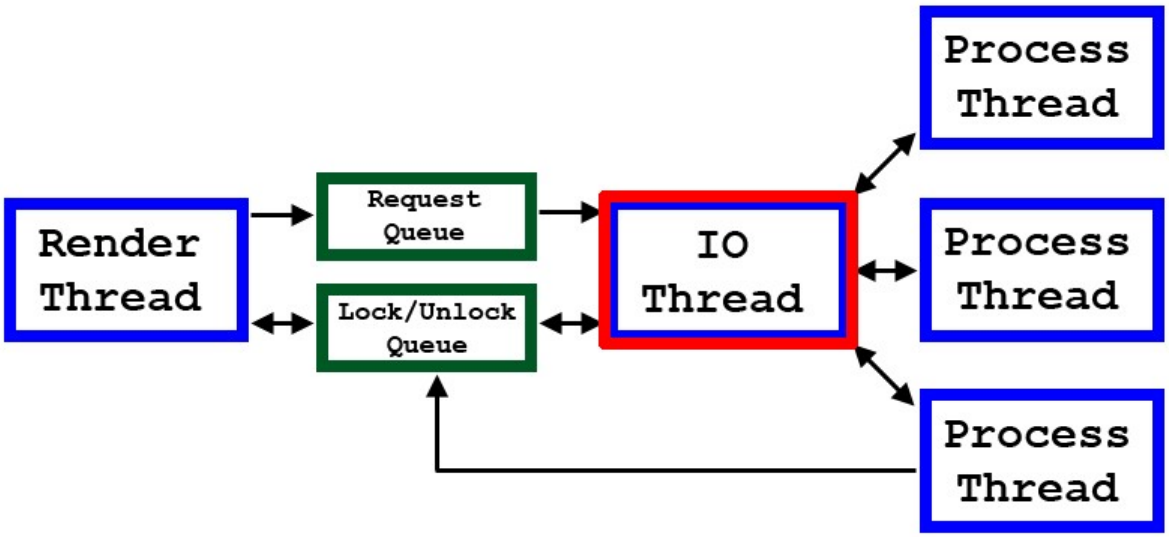
5. When the process thread is done processing, it puts a lock request into the lock/unlock queue.



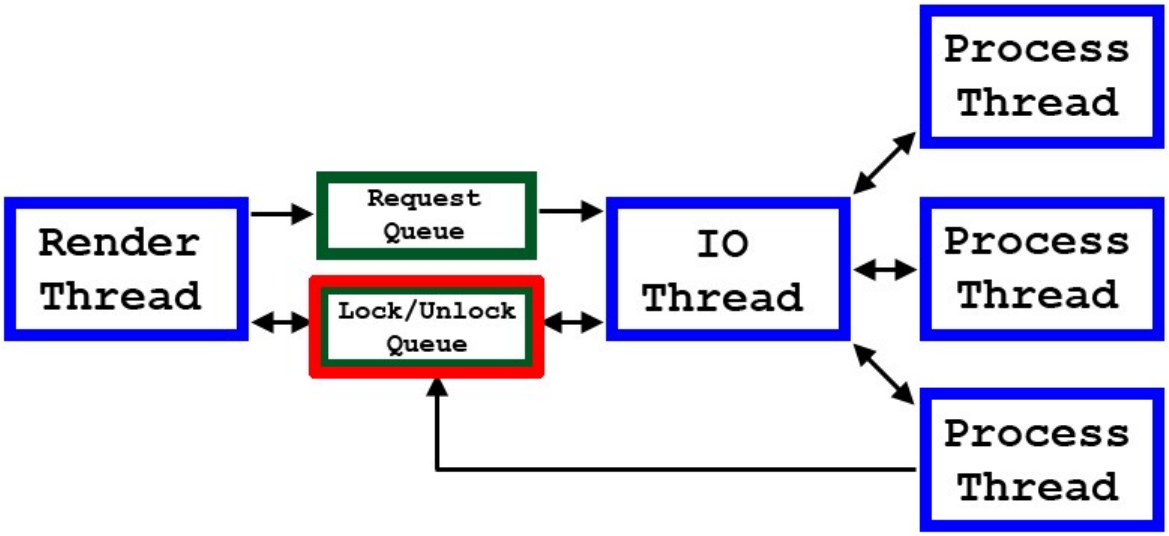
6. When the render thread feels like it, it picks up the lock request, finds an available resource from the resource reuse cache, and locks the resource.



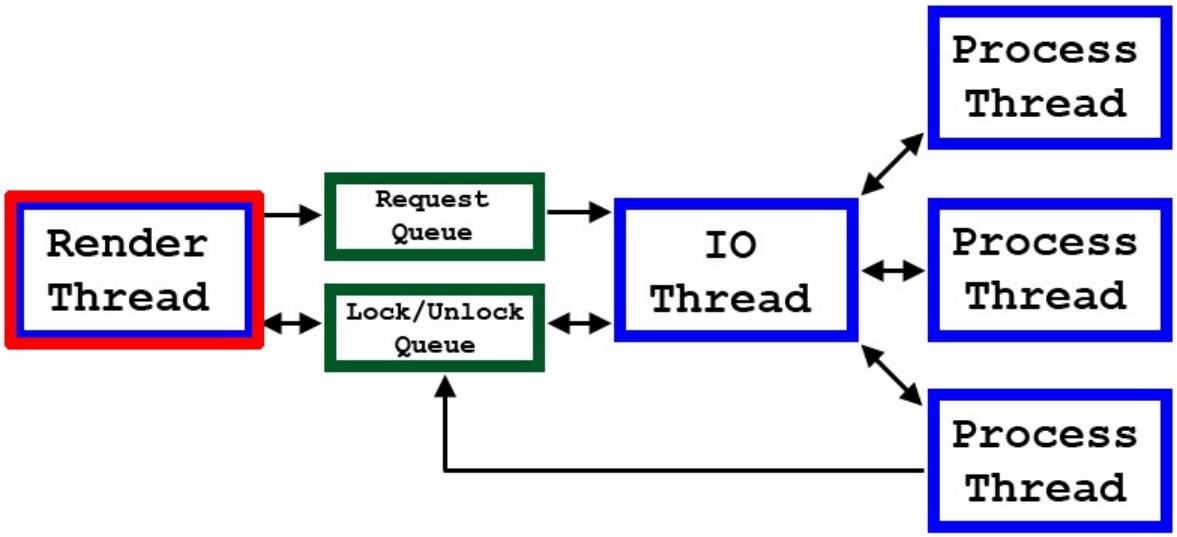
7. The render thread then sends the request back to the I/O thread. On Direct3D 9, the I/O thread copies the data into the device object.



8. Then the I/O thread puts an unlock request in the lock/unlock queue.



9. When the render thread feels like it, it picks up the unlock request and unlocks the resource. The resource is now ready to use.



© 2010 Microsoft Corporation. All rights reserved.
Send feedback to DxSdkDoc@microsoft.com.
Version: 1962.00