# 10

# Camera-Centric Engine Design for Multithreaded Rendering

*Colt McAnlis*

*Blizzard Entertainment*

## Overview

Modern graphics APIs grant the ability to render in parallel by allowing the creation of drawing commands on separate threads. As such, an advanced engine design must have the ability to scale practically in order to take advantage of increased core count for rendering. In order to accomplish this, an engine must solve the problem of how to properly break up rendering tasks to be computed in parallel, but in order to do that, it must first consider what the rendering subsystem is actually doing.

Modern graphics engines require multiple renderings of the same scene to aid in the overall appearance of the final image. For instance, separate renders of a scene are required to compute shadow mapping, run-time reflections, screen tiling (for antialiasing on some game consoles), imposter generation, and offscreen particles. As such, it makes logical sense that the 'camera' is the coarsest form of scene organization directly tied to the visibility of an environment, given that in order to visualize any of the concepts above, you must first define a view position, view direction, and eventually a render target in which to store the results. As most views of the scene can be rendered independently of each other, we submit that grouping rendering workload by camera view offers the best rendering workload organization to take advantage of multi-core rendering.

In this gem, we present an engine design that scales to multi-core systems by using the concept of a camera to separate parallel rendering jobs. To accomplish this, we present two concepts. The first is an API-independent method of creating recordable command buffers, a process that enables us to use parallel rendering on any device. The second is an observation of how to distribute the creation of command buffers to separate threads based upon the camera with which it will be used. Using these two simple, yet powerful concepts can enable your engine to take advantage of multiple cores for rendering with the least amount of pain possible.

## 10.1 Uses of Multi-Core in Video Games

With the boom of multi-core system availability, there's a mad dash to fill extra cores with proper amounts of work to enhance the look of our products. The downside, though, is that for developers on platforms where the hardware configuration can change between two users (or even the same user), decisions must be made about how to accurately scale out visual features and workloads on lower-end processors.

Typically, we run in parallel things such as particles, animations, physics, etc., which can have level-of-detail (LOD) built into them for low-end machines. This concept, however, doesn't directly translate to the act of submitting commands to the graphics API, a process which can easily incur a great deal of performance overhead. APIs such as DirectX 9 and DirectX 10 suffer greatly from this, as it's not uncommon for frequent calls of basic API functions to become a performance burden. This typically causes the rendering phase to delay the processing of other systems in the architecture, which if highly dependent on refresh rate, could cause problems. The cause of this issue is the fact that we are required to submit commands to the device on the thread that owns the device, which for most games, is the same thread on which the simulation logic is run. For an in-depth examination of this process, please refer to [2], which provides many significant and properly documented examples.

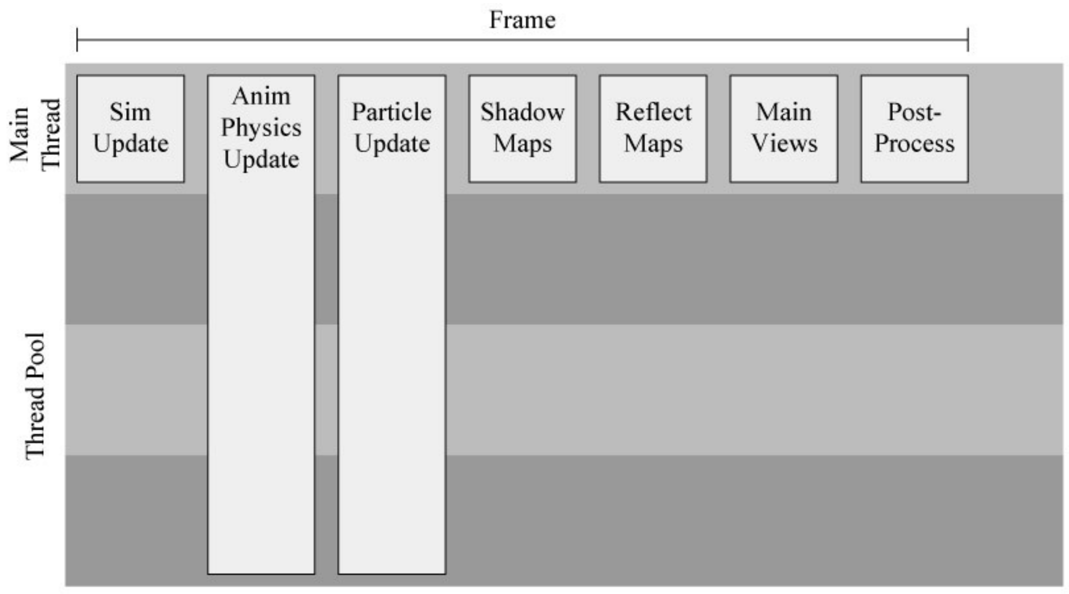

Figure 10.1: A standard game usage of parallel processing. Notice that only the update information tends to be multithreaded.

Figure 10.1 displays an example frame setup for a given game engine. In this example, the rendering device is owned by the primary thread, and as such, rendering a scene blocks the frequency of simulation updates. As described, the only two systems that use the thread pool are particles and animations, namely because of their ability to be updated without memory contention, and the concept of scaling back accuracy or LOD for these systems is trivial. The end of our frame goes through the process of rendering shadow maps, reflections, the main view, and finally post-processing.

Because of the fact that we must submit commands to the device on the thread that owns it, most of the rendering commands are interwoven with logic-based operations such as scene traversals, update logic, and the like. This type of rendering architecture can make it difficult to maintain your engine and port it to other platforms.

To be fair, modern engines are not architected as poorly as shown in Figure 10.1, and on average make much better use of thread pool availability. Some excellent alternative threading architectures for various genres of video games, most of which operate by offloading the render device ownership to a separate thread, are listed in [2]. This follows the observation that the simulation code is not required to update at the same frequency as the graphics rendering, and offloading the rendering to a different thread allows the simulation to continue on to process the next frame after submitting a draw request.

Although this architecture is more common now, it is still far from ideal. Even with the device submission being offloaded to a different thread, it would be beneficial to have the ability to modify the drawing part of a frame such that it could make better use of thread pool availability, thus balancing out your thread utilization more evenly.

## 10.2 Multithreaded Command Buffers

Modern rendering APIs operate internally with a data structure known as a *command buffer* containing rendering information needed to process draw commands at the device level. Typically, these command buffers are filled on your behalf by the exposed rendering API and inserted into a command queue that is executed at a later time. Because of this, the thread on which the rendering APIs are called incurs the overhead of filling the command buffer, often absorbing CPU cycles in a way that has historically been a common problem for modern games.

At the time of the writing of this gem, a few APIs allow the ability to create dynamic command buffers or command lists on a thread separate from the one that owns the device, a process that allows us to distribute the overhead of filling the command buffers across the entire system. This represents an important step forward for rendering engines, as we now possess the ability to more finely control the workload of our rendering engine across our threading systems. The negative side, however, is that older APIs do not have the same abilities, causing issues for engines that aim to be compatible with multiple hardware levels.

To address this issue, Scheib [1] provided research showing a manner in which to create an overloaded DX9 device that would mimic the same recording ability as the newer APIs. To accomplish this, they overload the device APIs for the standard DX9 interface, and reroute them to their own system to composite a custom data structure that resembles an internal command buffer. They then submit this command buffer to the API on the primary thread owning the device using standard DX9 calls. On the primary thread, we are still incurring the overhead of submitting API calls, however due to the batched state, the overhead of API calls on the primary thread is reduced significantly, providing additional performance increases. Scheib shows that even with the overhead of submitting the buffer to traditional APIs, they still gain a significant performance increase by compositing the rendering command data on a separate thread.

## 10.3 Device-Independent Command Buffers

For legacy titles that would like to reduce the amount of API performance overhead without too much code rework, the method of [1] works quite well, allowing a drop-in solution that can benefit performance. For those titles with the luxury of analysis and not being bound by crunch deadlines, it's worth pointing out that the presented method can be considered overkill in terms of complexity, and short-sighted in terms of API differentiation. Analysis of your rendering systems will prove that a subset of device APIs are often used, and more so that most of them occur in frequently predictable patterns. As such, when generating an API wrapper to the device, you waste time by offering support for functions that are not used by your title.

Logically, API calls should be hidden behind a wrapper interface, anyway, in order to minimize the amount of code that would need to be reworked to support multiple platforms and rendering APIs; as such, it makes sense that API calls for dynamic command buffer recording should also be hidden. In this context, the presented method of [1] is lacking, and a custom API-independent solution is required.

## RenderCommand Structure

To this end, we present the concept of a `RenderCommand`, which contains the least amount of information required to submit a draw call to the API in a device-independent fashion. In effect, our `RenderCommand` structure is designed to mimic the draw commands that modern APIs use internally, containing information about which vertex buffer to use, the shading states, and how many polygons to draw. Depending on the needs of your graphics engine, the layout and members of this structure can vary greatly; however, in the interest of memory throughput, it's a good idea to keep this structure as small as possible by quantizing as much state data as you can. The code in Listing 10.1 shows an example of this process in that, rather than listing out explicit states for blending, a single enumerated state is used.

Listing 10.1: A simple RenderCommand structure that contains basic information for a draw call.

```
struct RenderCommand
{
  ResourceHandle  VertexBufferHandle;
  uint32         VertexDeclEnumIndex;
  uint32         NumTriangles;
  uint32        NumVerts;

  enum PrimType
  {
    kTriList = 0,
    kTriStrip
  };

  PrimType    PrimitiveType;

  enum BlendType
  {
    kBlend_None = 0,
    kBlend_Standard,
    kBlend_Additive,
    kBlend_Subtractive
  };

  BlendType   BlendType;
  // and so on...
}
```

The code in Listing 10.1 is a very stripped down, simplified version of what a full production `RenderCommand` structure would look like. For bonus points in the memory category, you could make a dynamically resizable version of this structure that holds only delta states that change between this draw call and the previous one. In effect, this matches closer to what the APIs use internally, but for the complexity involved, the simple version presented above will suffice for the descriptive purposes of this article.

### Device-Independent Resource Handles

By design, the `RenderCommand` structure must contain handles to device resources that it will reference when executing the draw command. This exhibits a problem in that directly exposing device handles to the rest of the systems makes it difficult to port the engine to other platforms, often causing rendering code to be spread out across the entire project. As such, it's often useful to create managers that hold the actual device-specific resource handles and offer a wrapped handle to the rest of the systems. These `ResourceHandle` objects can help the process of porting the code to other platforms and also grant you a buffer interface for doing asynchronous asset loading. A full discussion regarding the issues of multithreaded resource creation and management is beyond the scope of this gem. For more information, we refer the reader to [1] as an introduction to the topic.

### Filling a RenderCommand Structure

A given `RenderCommand` structure simply references device information that it needs to execute. It does not, in contrast, actually load or own the device resources itself. As such, another class, which we will call a `RenderObject`, is responsible for managing the lifetime of a given resource (the actual resource itself should be owned by the manager responsible for it at a lower level, as described previously). Before filling in a `RenderCommand` structure, we assume that a given `RenderObject` structure has already been loaded into your engine and has communicated with the device in such a way to acquire proper rendering resources (such as vertex buffers).

Listing 10.2 describes the straightforward process of filling in a `RenderCommand` structure with the information contained in a `RenderObject` structure. As described, the `RenderObject` structure must contain information about how it needs to render, and in our simple example, it copies its state over to the `RenderCommand` structure based upon internal types and logic.

Listing 10.2: Filling a command buffer using generic handles. This is a great place to do additional logic related to rendering setup, since it will be executed on the thread pool.

```
void RenderObject::fillCommandBuffer(RenderCommand *RC)
{
  // make sure we're running on the threadpool
  ThreadAssert(ThreadPoolThread);

  if (ObjectType == kTypeOpaqueMesh)
  {
    RC->VertexBufferHandle = mVBHandle;
    RC->VertexDeclEnumIndex = kVD_Mesh;
    RC->PrimitiveType = kTriList;
    RC->BlendType = kBlend_None;
    RC->NumTriangles = numTrisFromPrimType();
    RC->NumVerts = mNumVerts;
  }
  else if (ObjectType == kTypeTransparentMesh)
  {
    // and so on...
  }
}
```

It's worth noting here that we're not computing new state for the `RenderObject` structure at this point; we're simply copying over state information relative to rendering and assigning it into the structure. This is a highly critical point when discussing the filling of these buffers on multiple threads, as this assignment pattern lends itself to being free of memory contention. That being said, the `fillCommandBuffer()` function is typically where your `RenderObject` structure would contain logic deciding *how* to fill in a `RenderCommand` structure properly. This includes things like checking the material to determine if we need to render with alpha blending. These types of logic and data access patterns can get complex at times, which is why moving them off to a separate thread frees up cycles on your device thread.

### Submitting a RenderCommand to the API

The act of submitting our custom `RenderCommand` structure to the API is overly verbose, yet direct in execution. Since we are filling in our commands on other threads, we must eventually resign ourselves to submitting the commands on the thread that owns the device. In practice, this relates to converting the data in our `RenderCommand` structure to information that we pass on to the rendering device APIs for execution. For more recent APIs that support command buffer creation, the act of submitting to the device requires a conversion from our custom command buffer to the desired device's format before submission; whereas for older APIs that do not directly support command buffer creation, the data must be directly fed to the device through API calls. Listing 10.3 describes the `executeDrawCommand*()` function, which at this point is the only function we've described that actually has direct access to the low-level rendering device. We present here the DirectX 9 version of the command, where we must call the API directly with our abstracted data types. It's worth noting that proper logic to determine which version of `executeDrawCommand*()` to call is a higher-level engine concept that is beyond the scope of this article; simple versions include a pointer to the proper function to call, while more advanced versions overload the `RenderControl` class entirely.

Listing 10.3: Submitting a RenderCommand structure to the API. This snippet of code is the only function that can actually communicate directly with the device.

```
void renderControl::executeDrawCommandDX9(const RenderCommand *params)
{
  ThreadAssert(DeviceOwningThread);
  // Set vertex stream
  const VertexBufferContainer *vbc =
    mManagedVBs.getElement(params->vbHandle);
  DX9Dev->SetStreamSource(0,
    (IDirect3DVertexBuffer9 *) vbc->devicehandle, 0,
    vbc->perVertSizeInBytes);

  SetShaderData(params);
  SetRenderStates(params);

  // We use lookup tables for these mappings because it's faster.
  DX9Dev->SetVertexDeclaration(StaticVDeclHandles[params->vDecl]);

  D3DPRIMITIVETYPE type = PrimTypeMappingLUT[params->PrimitiveType];

  // do draw
  DX9Dev->DrawPrimitive(type, 0, params->NumTriangles);
}
```

Because our `RenderCommand` structure wraps up data and information in an abstracted fashion, submitting that data to the API has to include redirection from the `ResourceHandle` types to API handles that can be sent to the device. Listing 10.3 below shows this process directly, where the vertex buffer manager must be given a `ResourceHandle` object in order to receive the proper device handle.

## 10.4 A Camera-Centric Design

As we've discussed, a camera is the coarsest form of batching container used to gather work for rendering. So far, we've described a system that allows us to fill in single command buffers in a device-independent manner and submit them to the render device at a later time. Now, we need to describe the proper manner of creating container classes to aid with this process of batching `RenderCommand` objects as well as a larger engine design to scale easily with multiple threads.

## Balancing Rendering Across Multiple Threads: Everything's a Camera

With the ability to render across multiple threads, the next step is properly utilizing this feature and applying it to your threading system. This means you need to figure out how to create job packets that represent rendering work to be done by filling in the `RenderCommand` structures. At the most direct level, it makes sense to simply create one job packet for each draw call that would occur in your scene, thus creating one `RenderCommand` object per job. There are some issues with this, the foremost being that modern engines typically group many draws together into a single command buffer so that it minimizes API submission overhead and also takes advantage of things like redundant render state filtering. Creating the draw commands independently of each other robs us of the ability to take advantage of this optimization as the commands are being created.

As such, it makes sense that we still need to create draw commands in such a way to take advantage of state filtering by batching them sequentially. The difficult part about this process is determining how to properly batch up your `RenderCommand` objects to take advantage of this. Typically, most engine designs embrace command buffer generation for objects being rendered into the primary view, but not for subsequent things like rendering shadow maps. This results in an unbalanced throughput with your rendering pipeline, being that submission of some draw commands in one section are significantly faster than in others.

As an alternate view on the rendering process, a given frame of rendering in a game can be described as a grouping of cameras that all contribute their view data to the final scene. GPU-based shadow mapping is a great example of this concept in action, as it defines the same camera and render-target system that your primary view does, dealing with the same culling, redundant state overhead, and LOD. Reflective render targets, offscreen particle buffers, and tiled antialiasing systems all exhibit the same characteristics as well and can be described using the same concepts.

This effectively solves a problem for us. Because our scene can be described in terms of cameras, we can use that idea to batch our `RenderCommand` generation, since objects that render to the same target typically share some sort of similar state data (for instance, shadow mapping requires all objects to use a different set of shaders).

Dividing our recording work by camera also provides us with a simple heuristic for scaling back workflow based upon hardware features. For instance, if we know the hardware is not fast enough to handle real-time reflections, cube-map cameras can be avoided and not processed. Another example is tiled antialiasing, a process in which subregions of the primary camera are used to generate larger images for portions of the screen, which are then downsampled and combined to generate the final image. By viewing each subregion as another camera render target, changes between antialiasing levels simply result in addition or removal of cameras from the system.

Figure 10.2 shows how a given rendering pipeline can change by breaking up command buffer generation by grouping them across cameras. Notice that the amount of work done across multiple threads increases (decreasing our overall processing time on the primary thread), but we add an additional API submit phase on the thread that owns the device.

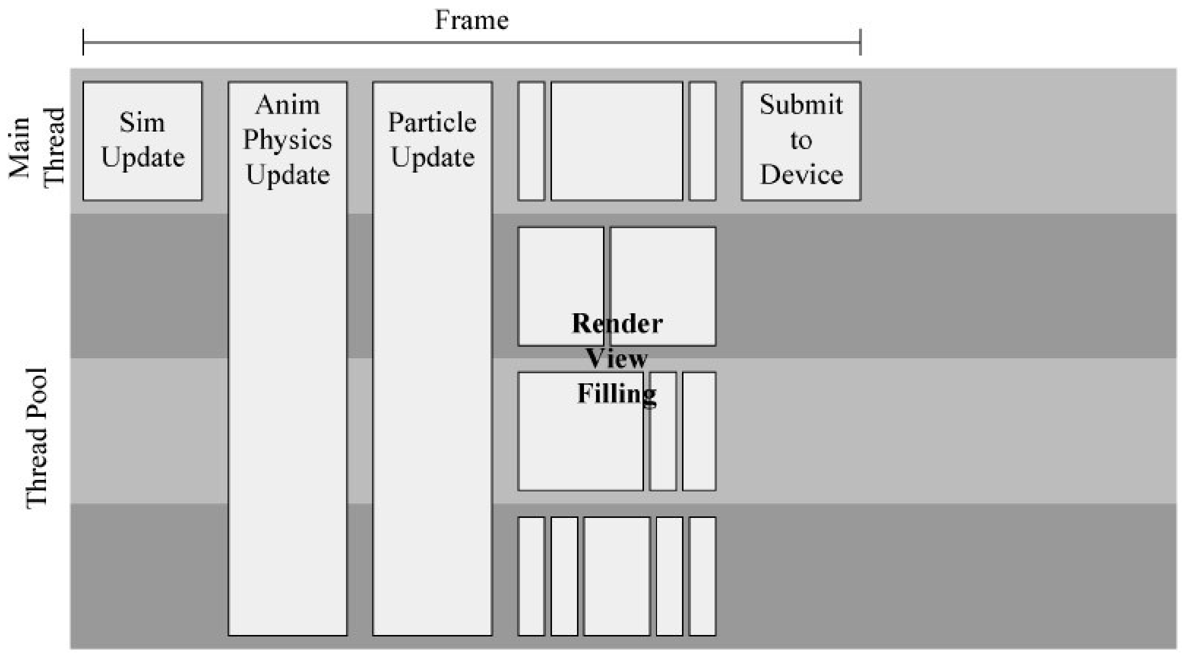

Figure 10.2: A camera-centric design. We make better usage of the thread pool for additional processing of rendering jobs.

For modern APIs, it's worth pointing out that the `RenderCommand` submit phase in Figure 10.2 is significantly smaller because the command buffers are simply copied from main memory to the device. This is possible due to the fact these APIs have their own command buffer formats, and a translation from our custom `RenderCommand` structure to the APIs version can be trivial. For older APIs that do not support command buffers, you cannot simply translate the command buffer data to the device format. Instead, you must communicate with the API through standard function calls as you normally would, using the data in the `RenderCommand` structure, as shown in Listing 10.3.

## RenderView Structure

At this point, we seek to fill `RenderCommand` structures by logically dividing them into rendering bins based upon the cameras to which they are visible. This is beneficial to us, because it will allow us to group together these draw commands, which typically will share some sort of state data (at the very least, they will all share the same camera transform matrix).

A `RenderView` object defines a structure that contains a linkage between a `camera` and its render target. More importantly, a RenderView object describes how a given render target is filled, meaning it must contain a list of all objects that are to be rendered to that given render target.

Listing 10.4 shows the comparison between a `Camera` structure and our `RenderView` structure. Notice that in general, our render view is a superset of a camera, taking into account additional graphics-related properties. In addition, note that we still contain wrapped `ResourceHandle` objects to represent our destination render targets. The `RenderView` structure also contains a list of RenderCommand pointers, which are filled in by the `RenderObject` objects that are visible to this view.

Listing 10.4: Comparison between a Camera structure that the simulation would use, and a RenderView structure.

```
struct Camera
{
  Float3   at, up, right;
  float    aspectRatio;
};

struct RenderView
{
  Camera              ViewCamera;
  Frustum              Frust;
  RenderTargetHandle    DestColorRTT;
  RenderTargetHandle    DestDepthRTT;

  List<RenderCommand *> RenderCommands;

  // this enumeration is very important as it defines the
  // order in which we submit render views to the API
  enum ViewType
  {
    kVT_ShadowMap = 0,
    kVT_ReflectionMap,
    kVT_MainCamera,
    kVT_PostProcessing,
    kVT_Count
  };

  viewType    ViewType;
}
```

It's worth taking a moment to discuss the `ViewType` member of the `RenderView` structure. Although we're compositing view information on multiple threads, we still require a specific resource dependency as we're submitting primitives to the API. For instance, we need to composite all the shadow map data before the primary view so that objects that use those resources can rest assured that they are available for use. To accomplish this, we must tag each `RenderView` object with a type so that later on, we can submit the render views to the API in a proper, serial manner. We cover the implementation of this process in more detail below.

## Filling a Render View Structure

Creating and filling the render views is a very simple process, but still requires a bit of understanding in the realm of threading, or at least an understanding of thread pools. We refer back to a concept that a large portion of your environment can be represented as a camera view, and as such, we must iterate over those object containers to create our render views.

Listing 10.5 below covers the process of creating a new `RenderView` structure for each camera type in your scene. This includes primary cameras, shadow maps, reflection maps, etc. Once the views have been created, we create jobs for the thread pool that fill in a given render view. At this point, the order in which the `RenderView` objects are assembled is trivial since the data access patterns should already be thread safe.

Listing 10.5: Creating all the render views for a given frame.

```
void renderControl::CreateRenderViews()
{
  List<RenderView *>    currentViews;

  // for each primary camera (this includes portal cameras)
  for (int i = 0; i < mCameras.size(); i++)
  {
    currentViews.add(new RenderView(mCameras[i], kVT_MainCamera));
  }

  // for each shadow map!
  for (int i = 0; i < mLights.size(); i++)
  {
    if (mLights[i].IsShadowCasting())
    {
      currentViews.add(new RenderView(mLights[i].getShadowCamera(),
        kVT_ShadowMap));
    }
  }
  // for each reflective target, etc...

  // now fill our render views with visible objects in a
  // threaded environment.
  for (int i = 0; i < currentViews.size(); i++)
  {
    Thread pool.QueueWork(procThreadedFillRenderView,
      currentViews[i]);
  }

  Thread pool.waitForWorkToFinish();
}
```

Once we've created our render views, we need to move forward with determining what objects to render. To do this, we must first cull the environment against the frustum of the camera owned by the render view, and then create a `RenderCommand` structure for each object that's visible to this camera. These `RenderCommand` objects can reside in a list structure that can be submitted sequentially to the API at a later time. Listing 10.6 covers this process by describing the internals of a thread procedure that creates the `RenderCommand` objects for a given render view. We assume that your object manager has some ability to perform frustum culling, from which you then gather the visible objects and have each one create a new `RenderCommand` structure.

Listing 10.6: Filling in RenderCommand structures should occur in a thread-safe manner, on a separate thread.

```
void renderControl::procThreadedFillRenderView(void *DataPacket)
{
  RenderView *currView = (RenderView *) DataPacket;
  List<RenderObject *> objects =
    gObjectManager.giveFrustumCollision(currView->frustum);
  for (int q = 0; q < objects.size(); q++)
  {
    RenderCommand *RC = new RenderCommand();
    Objects[q]->fillCommandBuffer(RC);
    currentViews[i].RenderCommands.add(RC);
  }
}
```

It's once again important to point out that your data access model at this point needs to be a read-only system that is thread safe. This means that while traversing your object hierarchy to cull against a camera frustum, you should be doing so in a manner that does not corrupt memory for code accessing the same data in other threads.

For the sake of thread safety and memory coherence, we allocate a new `RenderCommand` structure for each instance of an object for each camera to which it is visible. Our particular example uses this allocation metric for a few reasons, of which one is the assumption that you will submit your `RenderCommand` objects in a thread separate from the one they are allocated in, requiring frame-coherent memory. This can be a very performance-heavy process, and as such, you should keep an eye on it in case you need to implement a custom container that has faster allocation/deallocation speed. If this does not match your particular threading system, then you may be able to get by with a less dynamic model of `RenderCommand` allocation.

### Submitting a Render View to the API

Submitting a render view is a fairly straightforward process. We must simply bind the render target we're drawing to and then submit each of the commands that we have in our list to the API for drawing. Listing 10.7 shows this process, and it adds an additional set of data to indicate whether the render target should be cleared.

Listing 10.7: Serializing render views requires us to resolve them in a manner that satisfies dependencies.

```
void renderControl::serializeRenderViews(List<RenderView *> Views)
{
  for (int viewType = 0; viewType < Count; viewType++)
  {
    for (int i = 0; i < views.size(); i++)
    {
      if (Views[i].mViewType != viewType) continue;

      BindRenderTarget(Views[i]->renderTarget,
        Views[i]->DepthTarget);

      if (Views[i]->clearTargets)
      {
        ClearTarget(Views[i]->clearFlags,
          Views[i]->clearColor, Views[i]->clearDepths);
      }

      for (int k = 0; k < Views[i]->commands.size(); k++)
        executeDrawCommand(Views[i]->commands[k]);
    }
  }
}
```

In addition, Listing 10.7 highlights a very important aspect of serialization of `RenderView` objects, namely that some `RenderView` objects are dependent on others, so even though we're compositing them in multiple threads, we must still submit them in a particular order to the API. For instance, you need to composite shadow maps before you can use them in subsequent draw commands.

## 10.5 Future Work

We've presented in this article a means in which to offload the overhead of creating command buffers to separate threads in a device API agnostic manner and use a camera-centric design to ensure proper load balancing across multiple threads. As we continue to take greater advantage of our rendering APIs for general purpose computing, the ability to properly break up work based upon packets of rendering data will continue to be important.

There are many project- and system-specific issues that are related to this process that are beyond the scope of this article. For completeness however, we briefly describe them here and leave the nitty-gritty details as an exercise for the reader.

### Sorting and Instancing

`RenderCommand` objects provide excellent dynamic primitives for instancing evaluation. Once all the objects for a render view have been culled and added to a `RenderCommand` buffer, it is trivial at that point to define multiple types of sorting operations that can organize the `RenderCommand` objects in the buffer in various ways. Sorting by material index, vertex data, and object type often lends itself to a sorting process that combines objects in the command buffer in such a manner that multiple draw commands can be removed and a single command that uses instancing can be used. And if you're generating your `RenderCommand` structures dynamically, you can easily remove the original commands and insert the new instanced version. On some hardware, this may be a more performance friendly method of sorting by data, as your `RenderCommand` can take better usage of processor caches and reduce memory traversal that would normally be required when walking your object list.

### Better Load Balancing

For those of you who are more proficient at threaded architectures, it's worth noting that there's a modification to this process that can take greater advantage of your threading architecture. The practical observation is that most camera-specific `RenderCommand` generation processes exhibit uneven processing times. For instance, smaller viewports or different types of API commands cause less work to be done. As such, you can often wind up wasting processing time waiting for these unbalanced threads to finish.

A more advanced solution is to modify your thread procedure to create one thread job for each draw call, allowing each to be created on a separate thread, yet filling in the camera data structure atomically. This usually means that your visibility culling for each camera would need to occur on the primary thread so that you can ensure proper allocation of space in your `RenderCommand` list. In order to take advantage of render-state filtering, sorting, or instancing, you must perform these processes once all the individual `RenderCommand` objects have been properly created.

The benefit of this modification is that you've now created smaller job packets that can maximize your thread utilization over the course of your frame. This is very useful on platforms where your thread operations can be interrupted by OS events. For a further discussion on issues related to small-job packets in video games, please refer to [3].

## References

[1] Vincent Scheib."Practical Parallel Rendering with DirectX9 and 10". GameFest 2008.

[2] Lindberg, et al."Studies of threading success in popular PC games". Game Developers Conference, 2008.

[3] Randall Turner."Saints Row Scheduler". Game Developers Conference, 2007.