

Design Patterns for Low-Level Real-Time Rendering

Nicolas Guillemot

Thanks everyone for coming. My name is Nicolas, and today I'll talk to you about
"Design Patterns for Low-Level Real-Time Rendering"

Context

New Generation of Hardware-Accelerated Graphics APIs

 Vulkan™

DirectX 12



A major reason why this talk exists is because of the new generation of GPU graphics programming APIs that have come out in recent years. These APIs give you a level of flexibility that is much greater than their predecessors.

Context

The rules have changed.

Let's rethink programming with GPUs.

With the new generation of APIs, the rules have changed. We now have access to new primitives, and what I want to do today is to take a step back and rethink how we see things, and how we can use these new primitives to build new high level data structures.

Motivation: Generic Domain-Specific Solutions

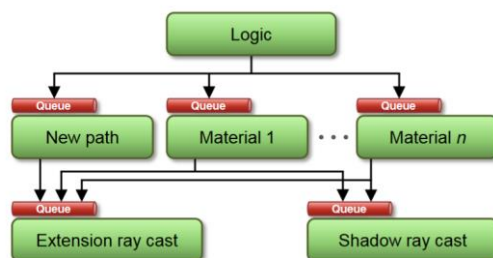
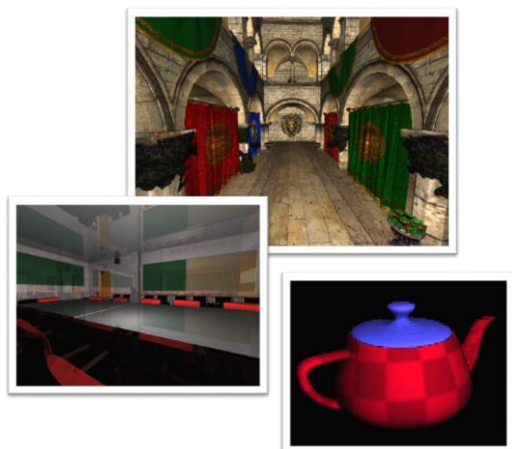


Figure from [Laine et al. '13]

Another motivation for this talk is to use these new low-level primitives to build generic domain specific solutions to domain specific problems. For example, here's a ray tracer I wrote with DirectX 12, based on the ray tracer design from the cited paper by Laine et al. The graph at the right shows the overall design of the ray tracer, where every task node has some dependencies on other nodes, and there exist queues to pass data between the nodes.

When I was implementing this ray tracer, I began by hard-coding each of the nodes with custom code. I quickly realized that I was repeating myself, so I took a step back and made a general purpose task graph interface that automatically managed dependencies between tasks, the execution of the tasks, and the memory for the queues. This greatly simplified my program, and gave me an API where I could cleanly add new task nodes and new material types without having to touch the low level details. I think this highlights the strengths of these new GPU APIs, where it becomes possible to create powerful high level systems using low level features.

Overview

Part 1: GPU Programming Basics

- Memory Management
- Command Lists
- Descriptors

Part 2: Renderer Design

- Ring Buffers
- Parallel Command Recording
- Scheduling GPU Work & Memory

Today my talk is basically broken down into two parts. First, I'll talk about GPU programming "basics", where I'll introduce various primitives for writing GPU API code. Second, I'll talk about some data structures you can build to write a renderer using the primitives introduced in the first section.

By the way, the talk I'm giving today generally introduces many things which make up my personal mental model that I've built up about how all this works. Your mental model may be different, so I invite you to do your own readings of the nitty gritty details and suggest alternate ways to understand these concepts.

Part 1

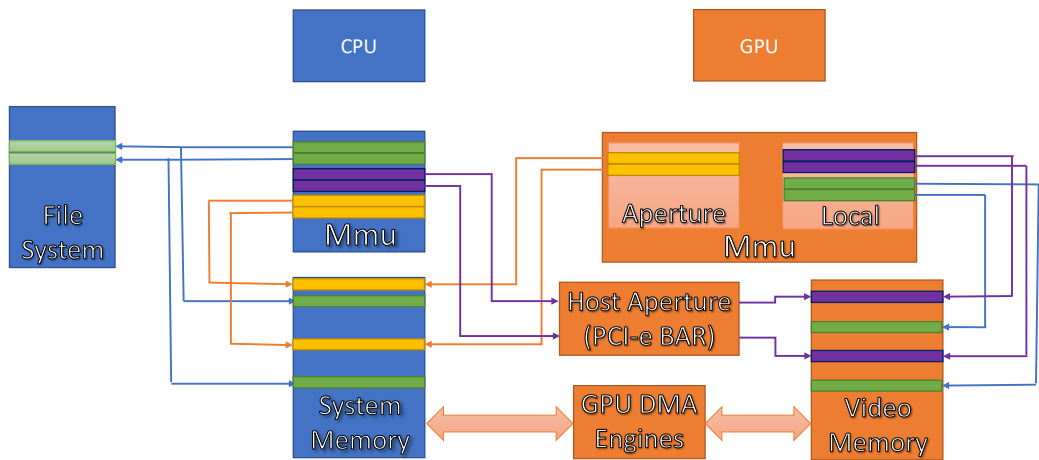
GPU Programming Basics

Let's begin with part 1 of this talk.

Memory Management

I'll first talk about memory management. Memory underlies so much of what we do, so I think it's important that we understand it well.

Discrete Video Memory Management



See: [WDDM2]

This slide introduces a mental model of GPU memory for a discrete GPU, where the CPU and GPU are two separate units.

The CPU has its own region of memory, here denoted “System Memory”. This corresponds to your computer’s RAM. To access this memory, the CPU uses an Mmu (“Memory Management Unit”). The Mmu has many features, and one such feature is virtual memory. For example, if you want to allocate two pages’ worth of memory, it may be that those two pages are physically discontinuous in RAM (green blocks in System Memory segment). However, in the virtual address space implemented by the Mmu, a range of contiguous virtual addresses makes the discontinuous physical memory appears as contiguous virtual memory. Furthermore, the Mmu has the ability to automatically handle memory exhaustion by temporarily storing pages of memory in the filesystem, which conveniently happens automatically with modern OSes.

The GPU also has its own segment of memory, here denoted “Video Memory”. This corresponds to your computer’s VRAM. Similarly to the CPU, the GPU also has its own Mmu, which also implements the abstraction of virtual memory. Again, this allows physically discontinuous pages to appear contiguous in virtual address space. This is

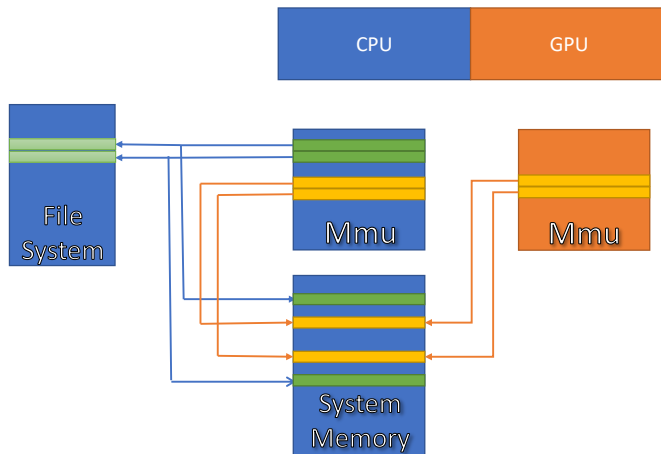
shown by the green squares on the GPU side of the diagram.

Now it gets a bit tricky. In the yellow squares, what's being shown is that some pages of memory are being allocated in the system memory segment, and these physical pages are being accessed through virtual addresses in both the CPU and GPU's Mmu. In other words, both devices can refer to this memory, each through their own virtual address space. This is one way in which CPU and GPU can communicate through memory.

Now, it's actually possible to do it the other way around too. It's possible for the physical pages to be allocated in video memory, and for both the CPU and GPU to map virtual addresses to these physical pages. This is shown as the purple boxes.

Finally, the DMA engines deserve to be mentioned. This is special purpose hardware that is part of the GPU and serves the purpose of doing efficient CPU <-> GPU copies. I deliberately put the DMA engine at the bottom of the diagram and connected to the two memory segments directly. This highlights the fact that DMA engines are often not aware of virtual memory, and thus can only do copies from physical memory to physical memory. Since pages of memory are discontinuous, that means that the copies done by the DMA typically need to be done on a per-page basis.

Integrated Video Memory Management



See: [WDDM2]

Now let's look at the case of the integrated GPU. In this case, the CPU and GPU are roughly merged into one unit. Furthermore, they both share the same region of memory. However, they still both have their own virtual address space. There does exist the possibility of using a shared virtual address space (see OpenCL SVM), but you can't assume that in general. Also, note that the system memory that is accessible to the GPU is not connected to the filesystem. This was also the case in the discrete GPU diagram. This highlights the fact that the GPU is generally not able to make use of the filesystem page faulting mechanism that is so convenient in CPU code. It's not technically impossible to do implement such a page fault system, but it's a generally unsupported use case.

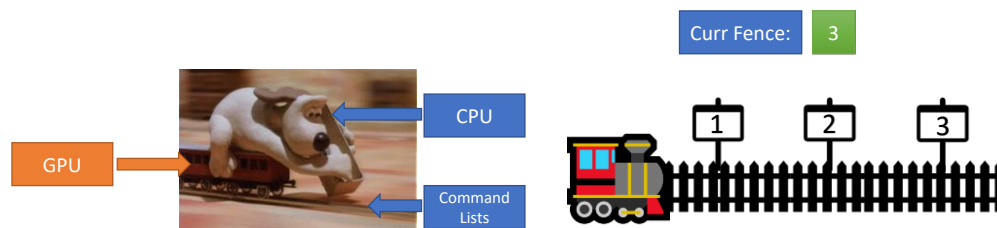
Command Lists

Next, I'll talk about command lists.

Command Lists – Big Picture

CPU – Out of Order Processor – Builds Command Schedule

GPU – In Order Processor – Executes Command Schedule



From a big picture point of view, I see command lists as a way for the CPU to send work to the GPU. Note that CPUs are typically out-of-order processors, and GPUs are typically in-order processors. By the nature of being “in-order”, the GPU executes work in the order in which it was given to it, as opposed to the CPU which can do all sorts of magic (eg. using speculative execution.)

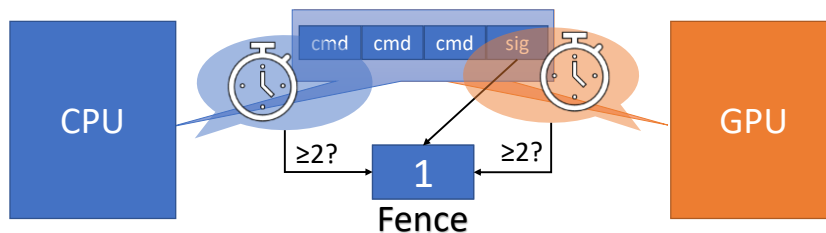
Since the GPU is an in-order processor, and in-order processors execute work in the order it was given, it is important that the work that is given to it is given in an order that is efficient. To do this, the CPU’s strengths can be used to build an efficient schedule to give to the GPU.

If you want an analogy, let’s compare it to this gif of Gromit. The train plays the role of GPU, and Gromit plays the role of the CPU. Gromit is laying down tracks to represent the command lists, and the GPU blindly follows the work it was given.

I’ll actually take this analogy one step further, and introduce you to the concept of “fences”. A fence is an operating system object used to monitor the progress of the GPU as it executes commands. For example, the GPU might start executing a list of commands, with the initial fence value set to 0. As the GPU progresses through the

commands, it eventually reaches the point in the command list where the “1” fencepost exists. At this point, the fence value changes to 1, and this information can be used to detect that the GPU has indeed finished processing the commands before it. As the GPU continues to make progress and reaches fencepost 2, the fence now changes to value 2, and so on. This is how you can keep track of the GPU’s progress.

Command Lists



Elaborating on my previous slide, here is a slightly more concrete example of using a fence. In this scenario, we have the CPU and GPU, and we have a fence used to synchronize between them, which initially has the value 1 (which is mostly arbitrary.)

The CPU records a list of commands that it wants the GPU to execute, and ends that list of commands with a special command: “signal”. This special command will update the value of the fence.

When the GPU receives the command list, it will execute the commands one-by-one. When it reaches the “signal” command, the GPU will change the value of the fence from “1” to “2”. Using this fact, the CPU can now synchronize on the execution of the GPU by checking if the fence has reached the value “2 or greater”. Similarly, the GPU can also synchronize itself on the fence (which is useful when you have multiple independent tracks of commands on the GPU and you want to synchronize between them.)

A Taste of Commands

DMA abstractions

Copy{src, dst}

Executing GPU programs

SetProgram{program}

SetParams{buf, tex, 1337}

Draw{N vertices}

or

Dispatch{N threads}

GPU Renderer Control

SetRenderTarget{rt}

SetGfxPipeline{pipeline}

Memory Model & Object Model

MemoryBarrier{object}

Transition{object, a, b}

Construct{object}

Destruct{object}

See: [D3D12 CL]

To set your expectations for the kinds of commands that exist, I'll go over a few common ones.

One simple example is a command to copy memory. This might be an abstraction over the DMA engine, and is able to copy memory between CPU and GPU efficiently.

Another kind of commands are the ones you need to execute programs on the GPUs. What's being shown at the bottom left here are a very common sequence of commands: Declaring what program you want to execute, passing the parameters for the program, and launching the program using either a Draw call or a Dispatch call. This sequence of commands is actually very similar to a function call. For example, when you make a function call in C or C++, the assembly code might do something like setting the function parameters by setting the registers, then make a branch to the body of the function. It's possible to make invoking a GPU program `_look_` a lot like making a function call.

Another kind of command are the ones that control the GPU's hardware-accelerated renderer. For example, you might have a command that sets the current rendertarget, to decide where the pixels should be drawn. Furthermore, you might have a

command to set the so-called “graphics pipeline state”, which configures the way in which say your triangles should be drawn.

Finally, I’ll show here a few relatively special commands, concerning what I call the memory model and the object model of the GPU. I’m presenting these in a somewhat abstract way, especially “Construct” and “Destruct”, which are in reality implemented by different commands that don’t have a name that looks anything like that.

The memory barrier command is useful for handling data hazards, since you must explicitly handle data hazards in many cases. Furthermore, there exist these so-called “transition” commands, which are a bit complex. The core idea is that some operations require their input object to be in a certain state for the operation to be valid. Therefore, if you want to run such an operation on such an object, and the object is not in the correct state, you must first transition the object from one state to the other. This is like a more general idea of a memory barrier, which does more than just handling read/write transitions.

Finally, I have these “Construct” and “Destruct” commands which, as I mentioned, don’t really exist directly. You can think of them a bit like “placement new” and “placement delete”. They’re useful for something I’ll cover later in this talk.

Note on Indirection

When to read command arguments?

- Command **record** time?
- Command **execute** time?

`Dispatch{N_threads}` vs. `DispatchIndirect{&N_threads}`

Performance vs. Flexibility trade-off

Before moving on from command lists, I want to mention one last concept that comes up a lot. The million dollar question is: “*When* should command arguments be read?”

The two main options are: Either reading the arguments at the time at which the commands are recorded, or reading the arguments at the time at which the command is executed (at the last second.)

For example, if you want to make a dispatch, you might pass the number of threads you want to launch by value. In this case, it might present a performance advantage to the driver and the GPU, since the implementation can better understand what work is coming ahead of time. On the other hand, it’s also possible to pass the number of threads *by reference*, which means that the number of threads to launch will be read at the last second just before executing the dispatch. This can give some very interesting opportunities to write more flexible code, but may come at some performance cost.

This tradeoff happens in many places, not just with dispatch calls, so it’s worth keeping in mind the distinction.

Descriptors

For the last part of this talk about GPU programming, I want to briefly talk about descriptors.

Descriptors

Hardware view of object: address + metadata

- Buffers, Textures...

eg: AMD GCN3 texture descriptor (128 bits)



Pseudo GPU Assembly:

```
dst = image_sample(xy, texture_descriptor, sampler_descriptor);
```

See: [GCN3]

The way I see it, descriptors are the hardware's view of an object. And when I say object, I mean mainly "buffers" and "textures". (And there exist many kinds of buffers and textures.) But in general, a descriptor contains two things: An address to the memory of the object, and some metadata that describes how that object should be accessed.

As a concrete example, here is the specification of a descriptor on AMD's GPUs (note this is not the only type of texture descriptor they support.) Mainly, you can see very precisely how this descriptor contains the address of the memory of the texture in the lower bits, and contains the width and height of the texture in some higher bits.

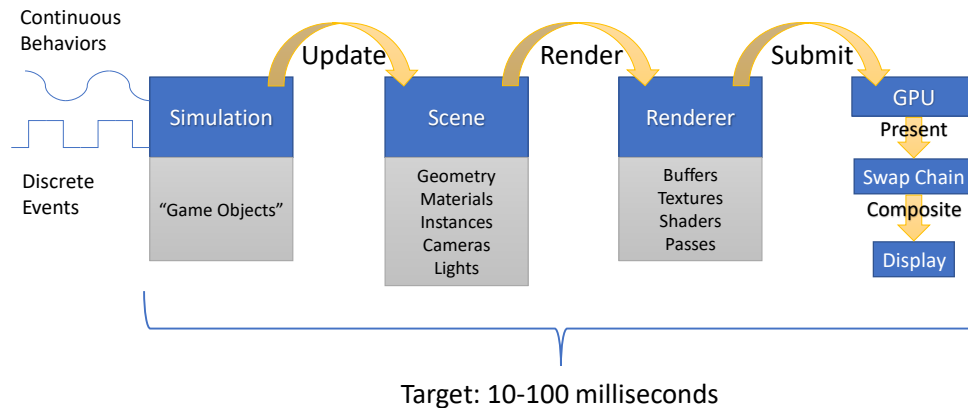
At the assembly level, this descriptor is passed as an argument to functions that relate to textures. Disclaimer: This is a bit of an AMD-centric viewpoint, but the concept of descriptors are supported on all modern GPUs (ie. in all DirectX 12 GPUs.)

Part 2

Renderer Design

Now let's talk about the design of an actual renderer using the primitives introduced in the first part of this talk.

Real-Time Renderer Architecture



Before talking too much about the specifics, I want to give you an idea of the overall design of a real-time rendering architecture.

To start, we have the “Simulation”. This is the place where the actual logic of the application happens. For example, if you’re making a video game, this is where you would implement the game logic. This simulation is usually said to work in terms of “game objects”, as an abstract concept which is implemented in different ways in every game engine. These game objects are updated based on the passages of time, and based on events that arrive through the gameplay, such as messages being sent from one object to another, or through things like key presses.

From there, the simulation code describes the state of the scene it wants to render by updating a data structure that represents high level scene concepts. For example, the Scene might exposes concepts like “Geometry”, “Materials”, “Instances” (of meshes), “Cameras”, “Lights”, and so on.

Once the scene’s organization is described in the abstract terms of the Scene class, the Renderer is now able to reads the contents of the scene, and translate its high level specification into commands that are relevant for actually rendering the scene.

For example, this Renderer deals in objects like Buffers, Textures, Shaders, and Passes. This gives some separation of the description of the Scene vs. how it is displayed, which is a nice way to encapsulate the two. Although in practice, there may be some overlap between the Scene and the Renderer. This separation is mostly conceptual, so feel free to adapt it to your own project however you find works best.

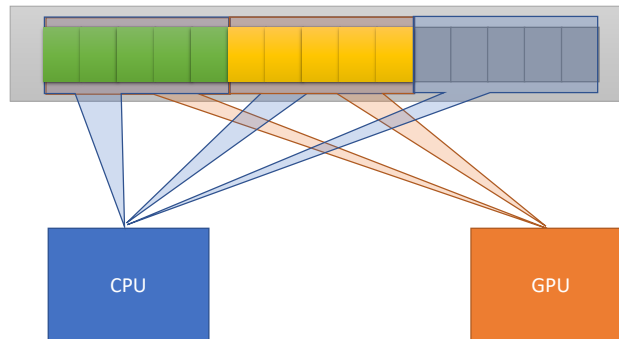
As the renderer translates the scene description into a list of GPU commands, the GPU processes these commands, and usually finishes by “Presenting” a frame of rendering. This frame is passed on to the “Swap Chain”, which represents the operating system’s double-buffering mechanism. The frames in this swap chain will be composited with the rest of your desktop (or this composition step may be skipped if you’re in full-screen mode), and finally the frame is shown on your display.

We aim to have this whole process executing within the order of magnitude of 10-100 milliseconds. If you’re making a shooting game or a virtual reality game, you probably want to be on the lower end of this spectrum. However, you may be making for example a 3D modeling application, where it’s deemed acceptable to increase the latency of rendering in order to give a more accurate visualization. Therefore, the ideal target latency is up to you to determine based on your needs.

Ring Buffers

One of the most useful (in my opinion) data structures for building a real-time renderer is the ring buffer, so I'll talk about that first.

Ring Buffers



A ring buffer is a very general concept. In this case, I'll mostly be talking to you about how it can be used to stream data from the CPU to the GPU, but it's very versatile and can be used in many scenarios. Nevertheless, we'll focus mainly on CPU -> GPU streaming in this talk.

So, here we have a buffer (in grey at the top), which is the ring buffer itself. Then we have the CPU and GPU who will use this memory to communicate with each other.

The CPU begins by reserving a range of memory in the ring buffer, and writing data to it. From there, that memory is passed on to the GPU (which reads the memory), and the CPU begins writing the data that follows it. In this way, the CPU and GPU can both be working together in parallel, with a producer/consumer relationship.

As time goes on, the CPU keeps writing more data ahead of the GPU, and the GPU follows behind it by consuming that data.

Ring Buffer API

```
auto [pCPU, pGPU] = pRing->Alloc<Camera>();  
*pCPU = camera;  
pCmdList->SetDrawParam(CAMERA_PARAM_IDX, pGPU);  
  
auto [pCPU, pGPU] = pDescriptorRing->Alloc(1);  
WriteDescriptor(pCPU, desc);  
pCmdList->SetDrawParam(TEXTURE_PARAM_IDX, pGPU);
```

It's possible to define a very neat API for a ring buffer. What I'm proposing here is to have some kind of ring buffer class which has an "Alloc" function. In typical C++ code, the result of a memory allocation function is a simple CPU virtual address. However, in this case, we don't get just a CPU virtual address. The result of an allocation in the ring buffer is both a CPU virtual address *and* a GPU virtual address.

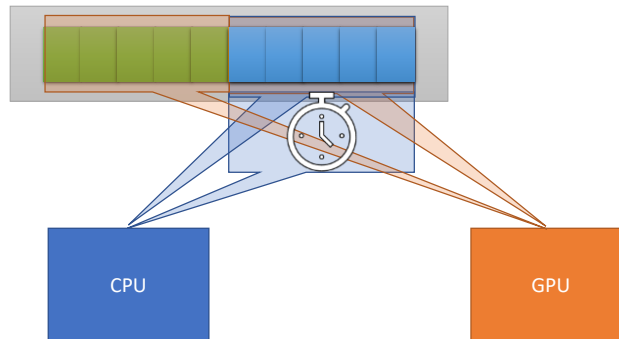
From there, you can use the CPU virtual address to write the data you want to send. Then you can pass the GPU virtual address as a parameter to a draw call. That's why having the two addresses is useful.

By the way, it can be very important to add another step in between, where you copy the memory over to proper video memory. Using system memory as a source of data for rendering will work, and is ideal with integrated GPUs, but may cause performance problems on discrete GPUs depending on what you are doing.

Moving on, another potential use for ring buffers is to use them to allocate descriptors. For example, I might have a ring buffer of descriptor memory, which I can use to freely allocate descriptors. Again, I get both a CPU and GPU virtual address to this descriptor. I can use the CPU virtual address to write the contents of the descriptor (metadata and address), and I can pass the descriptor's GPU virtual

address as a parameter to the draw call.

Ring Buffers: Handling Out-of-Memory



Now, there are lots of tricky cases relating to ring buffers. One such tricky case is handling an out-of-memory scenario.

For example, what's shown here is that the CPU wants to make an allocation in the ring buffer to output some data, but the GPU is currently reading it. In this case, you might decide to force the CPU to block and wait for the GPU to finish reading the data. You could do this with a fence, as was presented earlier in this talk.

Ring Buffers: Handling Wrap-Around

Use a monotonic “virtual” offset

- `data = buffer[virt_offset & (RING_SIZE - 1)];`
- (Simpler fences!)

Consider simply disallowing wrap-around.

- (Fixed memory budget per frame.)

Recommended reading:

- <https://fgiesen.wordpress.com/2010/12/14/ring-buffers-and-queues/>

Another tricky problem is wrap-around. Though there do exist some ways in which you can greatly simplify that problem.

First of all, I recommend using so-called “virtual offsets” in the ring buffer. Rather than keeping track of the actual offset in memory of the ring buffer, you simply imagine that the ring buffer has an infinite size. This means that you keep increasing your offset even if you exceed the size of your ring. When you want to translate a virtual offset into a real offset in the buffer, you can simply use a bit mask (assuming that the ring buffer has a power-of-two size, which is a really good idea for other reasons as well.)

Using monotonically increasing virtual offsets also turns out to make it easier to associate a fence to a ring buffer. Since the ring buffer’s current offset increases monotonically, and the fence’s value also increases monotonically, it becomes easier to associate one to the other.

Another option, which is a bit of a hack, is to simply disallow wrap-around. This can be done by pre-allocating a fixed budget of memory for each frame. You can put an assert for when you exceed that amount, and simply go increase the size of the buffer for your application. It’s a bit simple-minded, but it can get you very far.

If you want to implement something like this, I recommend reading Fabian's writeup on ring buffers. This article has a lot of information about the invariants you can establish to define a ring buffer, and has some proofs about nice properties that they have.

Ring Buffers: Lock-Free Allocation

```
std::atomic<uint64_t> offset;
```

- Structured Data (offset is array index.)

```
uint64_t alloc(uint64_t n) {  
    return offset.fetch_add(n);  
}
```

- Raw Data. Over-aligned for GPU access.

```
#define WORST_ALIGNMENT 512  
uint64_t aligned_alloc(uint64_t sz)  
{  
    uint64_t padded_sz = (sz + WORST_ALIGNMENT - 1) & ~(WORST_ALIGNMENT - 1);  
    uint64_t allocated = offset.fetch_add(padded_sz);  
    assert(allocated + sz <= RING_SIZE);  
    return allocated;  
}
```

Align up to GPU requirements



Another thing we can easily do with ring buffers is to make allocations from them work in a lock-free fashion.

If we make the offset in the buffer be an atomic variable, we can implement allocations from it with a simple “fetch_add”. This function increases the offset by the specified amount, and returns the *old* value of the offset. The old offset represents the start of your allocation, and the offset is moved forward in an atomic way.

If you want to be able to allocate memory for arbitrary structs (which have a size specified in a number of bytes), then you have to worry about alignment. There are many places where the GPU expects the memory addresses you pass to it to be aligned at a specific alignment. In the case of DirectX 12, the worst case you’ll find is 512, used for uploading texture data. You can make a general-purpose allocation function that always allocates to this worst case alignment, which allows you to write code in a somewhat simpler fashion where you don’t have to worry about remembering to align your data. It may be slightly wasteful to have such large padding for every allocation, but so far this hasn’t been a problem for me. “\(\ツ\)”

Ring Buffers: Pros & Cons

Pros

- Simple memory management
 - Dead simple API
 - Avoids fragmentation
- Powerful building block
 - Procedural geometry
 - Texture streaming
 - “Sprite Batch”

Cons

- Memory budget calibration
 - Too small: Bad perf or crash
 - Too big: Wasted memory
- No one-size-fits-all configuration
 - System memory? Video memory?
 - Cache properties? (WC? WB?)

Ring buffers have some pros and cons, which I’ll summarize here.

In the pros, I think it greatly simplifies memory management. Instead of creating lots of small buffer objects all over the place, you can easily allocate objects through a simple API. Furthermore, it avoids the fragmentation that might happen if you tried to make a bunch of small allocations.

Furthermore, I think having a ring buffer gives you the opportunity to easily implement many other kinds of rendering data structures. For example, you might use it to submit procedural geometry data, or use it to stream texture data (upload pixels in a ring buffer & copy into texture), or you could use it to implement a so-called “sprite batch”, which is used to render 2D sprites more efficiently by grouping them together.

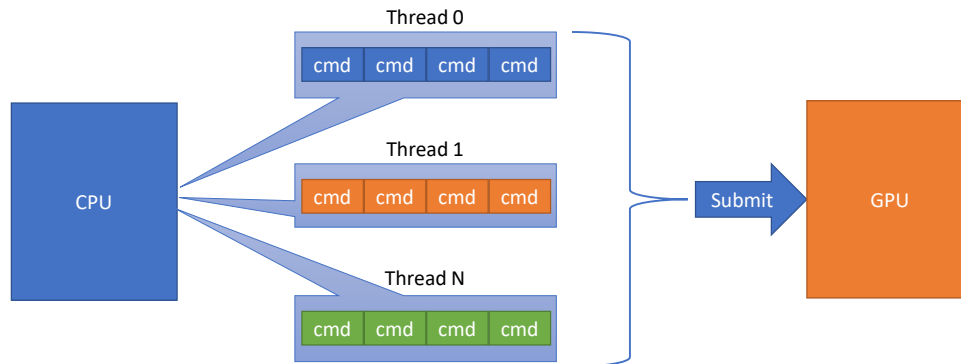
In the cons, the biggest difficulty is the calibration of memory. If you make your ring buffer too small, you’ll either have bad performance (due to requiring lots of synchronization to wait for the GPU to finish), or you’ll crash due to being out of memory (if you used a plain assert.)

Another con is that there's no ideal best configuration of the memory for all cases. For example, should the ring buffer be allocated in system memory, or should it be allocated in video memory? Should the cache properties of the ring buffer be write-combined? Write-back? Write-through even? It's hard to make a general choice, and this choice is very important. For example, if you try to run a procedural geometry algorithm in-place in write-combined memory, you're going to have a bad day.

Parallel Command Recording

The next section of this talk is about parallel command recording, which is one of the major features that Vulkan and DirectX 12 enable.

Parallel Command Recording: Big Picture



The idea for recording commands in parallel is as follows:

Writing GPU commands is a CPU-intensive process. Therefore, when you want to write a large number of commands (like if you want to render many many objects), the command writing itself can easily become a performance problem. To improve the performance of this case, it is possible to record commands in many threads in parallel. For example, the CPU might have many threads each creating its own list of commands, and once those commands are all written, they can all be submitted together to the GPU.

Easy Case: Regular Work

```
CmdList lists[NUM_JOBS];
parallel_for (jobID = 0 .. NUM_JOBS)
{
    lists[jobID].SetRenderTarget(rt);
    foreach (object in job) {
        lists[jobID].SetGeometry(object.geometry);
        lists[jobID].SetMaterial(object.material);
        lists[jobID].Draw(object.num_vertices);
    }
}
Submit(lists);
```

In the simple case where you have many objects that are roughly processed in the same way, it becomes very easy to parallelize the code that writes their commands. You can roughly split the objects of the scene into many subsets, each subset associated to one job. You can run these jobs in parallel, with each job recording the commands necessary for that subset of the scene. After all jobs are done, you can submit them in parallel.

By the way, I'd like to add that you should still not prematurely optimize. It may be not worth using parallelism here if you're writing only a few commands, which may be the case depending on how you architect your renderer.

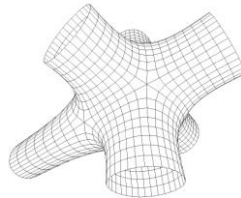
A good rule of thumb is that each command buffer should be at least 50us of GPU time, and each submit should aim for 500us minimum.

Difficult Case: Irregular Work

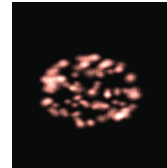
```
void DrawAll(const vector<Drawable*>& drawables);
```



```
class Blob : public Drawable
```



```
class Subdiv : public Drawable
```



```
class Particles : public Drawable
```

CPU work varies per object!

Images: [RTR '08]

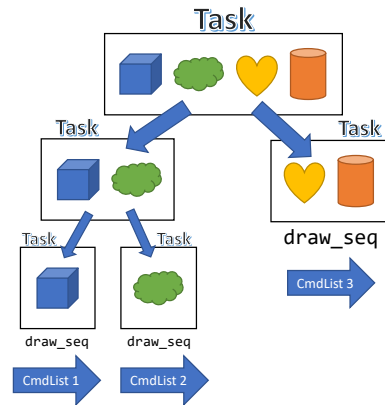
The last slide was the easy case, now let's look at the rough case: When the objects in your scene are relatively heterogeneous, and the amount of CPU time used to prepare their rendering commands varies. This is going to get a little academic, so bear with me.

The amount of work to prepare the rendering of an object may depend on the type of the object. For example, the Blob might use marching cubes to be polygonised, the Subdiv might use some triangle subdivision algorithm, and the Particles might use some binning and sorting. The amount of work may also depend on other factors like how close the object is to the camera or whether the object is occluded or not.

Irregular Work: Basic Fork/Join Solution

```
void draw_par(cmdlist, drawables) {  
    if (drawables.cost() < WORTH_SPLITTING) {  
        draw_seq(cmdlist, drawables);  
    } else {  
        auto [left, right] = drawables.split();  
        spawn draw_par(cmdlist, left );  
        spawn draw_par(new CmdList(), right);  
        sync;  
    }  
}  
draw_par(new CmdList(), all_drawables);
```

See: [CppCon 2015: Pablo Halpern "Work Stealing"]



The standard solution to a case like this is to use fork/join parallelism. The idea is to recursively subdivide the amount of work you have to do until you judge that you have approximately an amount of work suitable for one CPU core. These tasks are then put into a task scheduler, and they can be efficiently allocated to CPU cores.

Illustrated to the right is such a list of objects to draw, which is recursively split until it fits into one CPU task. Each task then has its command lists recorded independently.

I highly recommend watching Pablo Halpern's 2015 CppCon talk on the topic of work stealing. It's a very powerful concept.

Irregular Work: Hyperobject Optimization



- Key Idea: Steal Parent Task & Keep Your List
- See: "Reducers and Other Cilk++ Hyperobjects"
- Keeps draw order intact! (Important!)

See: [Hyper '09]

Going one step further with the idea of handling irregular workloads with fork/join, I'd like to propose an interesting solution to a tricky problem.

Let's look at the case of the blue cube and the green cloud. These two objects were split into their own tasks, so their command lists were recorded independently. However, let's suppose that at runtime, it turns out that both of these task end up running on the same CPU, one after the other. In this case, it was wasteful to create two separate lists. Instead, it would be better if we could just reuse the same command list for the second object.

With the inclusion of the so-called "Hyperobject" language concept, it becomes possible to implement this. As a result, in the scenario where both objects end up being executed on the same CPU one-after-the-other, both objects will be recorded in a single command list. This should reduce the overhead caused by excessively splitting work into command lists.

If you're interested in this idea, I recommend reading the paper cited on this slide. It's a really interesting paper because not only does it explain a lot about how Cilk's scheduler works, but it also shows how Hyperobjects are very cleanly integrated into

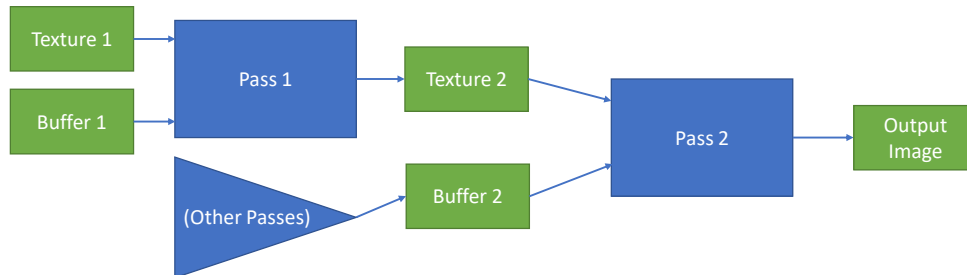
their scheduler.

Finally, an interesting note about applying this problem to rendering, is that the transformation made by the hyperobject does not affect the order of the draw calls. This is important, because drawing objects in a different order can cause flickering (when the final image depends on the order of submission). Drawing objects in a different order can also cause the performance to change between frames, so it can be useful to try maintaining a stable order of draw calls between subsequent frames to avoid sudden latency spikes.

Scheduling GPU Work & Memory

In the final part of this talk, I'm going to be talking about scheduling GPU work and memory through a frame graph style scheduler.

Scheduling: Big Picture



Duties:

- Submit work in valid (& efficient) order
- Manage object allocation, lifetime
- Respect dynamic nature of real-time rendering

The big idea with GPU scheduling comes from looking at the work done during a frame at a global scope, and seeing that it can be decomposed as a series of passes with memory passed between them. From there, we want to build an algorithm that can execute this graph efficiently.

The scheduler should submit the work in a valid order, to mean that it respects the dependencies between nodes. It should ideally also try to submit the work in an order that is efficient, to mean that the code runs on the GPU more efficiently.

Furthermore, there exists the opportunity to handle object allocation and lifetime as part of the execution of this graph. For example, if you look at Texture 2 in the upper middle of this slide, you can see that it's used as an output of pass 1 and an input of pass 2. Beyond this region of the graph, this texture does not need to exist. Therefore, it may be a benefit to allocate and deallocate Texture 2 before and after the range of time in which it is used. That's what I mean by the scheduler managing the allocation and lifetime of objects.

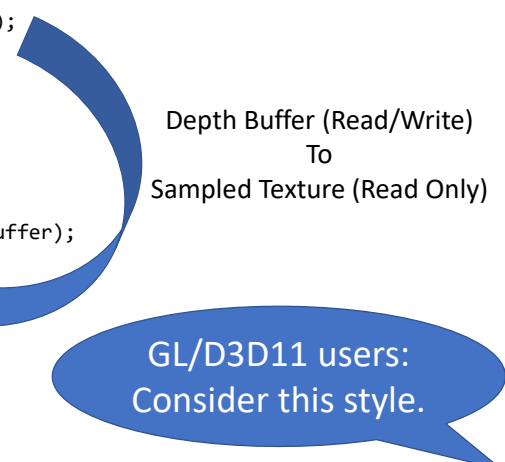
Finally, the scheduler should respect the dynamic nature of real-time rendering. What I mean by that is that the contents of the scene can often change dramatically from

one second to the next. When one looks at a cleanly presented task graph like the one above, it's tempting to make an API where one has to declare the whole state of the graph ahead of time, like at program initialization time. This may work, but it can also be hard to fit this design to a real-time application where everything is changing all the time. In fact, you may want to rebuild the whole graph from scratch every frame.

Scheduling: Classic Multi-Pass Approach

```
SetRenderTargetAndDepth(nullptr, shadowMap);
SetPipeline(shadowPipeline);
foreach (object in scene) {
    SetGeometry(object.geometry);
    Draw(object.num_vertices);
}

SetRenderTargetAndDepth(backbuffer, depthbuffer);
SetPipeline(scenePipeline);
SetParam(SHADOW_MAP_IDX, shadowMap);
foreach (object in scene) {
    SetGeometry(object.geometry);
    SetMaterial(object.material);
    Draw(object.num_vertices);
}
```



Depth Buffer (Read/Write)
To
Sampled Texture (Read Only)

GL/D3D11 users:
Consider this style.

In classic graphics code, this is something like what one would expect to see for handling a multi-pass algorithm.

The code on this slide is split into two parts: The first pass at the top, and the second pass at the bottom.

What's happening is that the first pass renders a depth buffer to the "shadowMap" variable. If you don't know what a shadow map is that's fine, you just need to know that this is the output of this rendering pass.

In the second pass at the bottom of the slide, the shadow map is now used as an input. Therefore, the shadow map went from being a depth buffer to being a depth texture. This represents a change in state from being in a read/write format to being in a read-only format. As a result, there needs to be (at some level) some barriers (and perhaps some compression/decompression) that implement this transition in usage.

If you write this style of code in OpenGL or Direct3D 11, it is likely that the driver automatically handles this for you, since the driver has much more high level knowledge of what's happening. On the other hand, with the new low level APIs, you need to explicitly insert the relevant resource barriers. So if you're still writing

GL/D3D11 code, writing code almost exactly like this slide is not really a problem, but newer APIs require you to do some extra hand-holding.

Scheduling: Previous Work

“FrameGraph: Extensible Rendering Architecture in Frostbite”

Author: Yuriy O'Donnell

- <https://www.ea.com/frostbite/news/framegraph-extensible-rendering-architecture-in-frostbite>

“Render graphs and Vulkan – a deep dive”

Author: Hans-Kristian Arntzen

- <http://themaister.net/blog/2017/08/15/render-graphs-and-vulkan-a-deep-dive/>

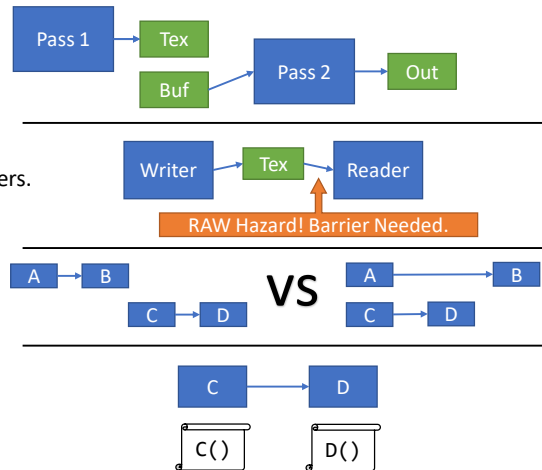
Today: Summary

There is actually a lot of previous work on this topic. I recommend you check out the FrameGraph talk by Yuriy, and the Render graphs article written by Hans-Kristian (especially if you're using Vulkan.) Hans-Kristian's implementation is also open source, so you can go have a look at the internals.

I don't want to just repeat everything they said, so today I'll try to broadly summarize the topics they address.

Work Submission

- Compiler-like optimizations
 - Eg: Dead Code Elimination
- Data hazard handling
 - Insert memory barriers, pipeline barriers.
- Inter-pass latency management
 - Cache coherency vs. latency hiding?
 - Low vs high watermark?
- Invoking command list recording
 - eg. `IRenderPass::Record()`



On the topic of work submission, there are many kinds of optimizations, tradeoffs, and procedures to follow in the implementation of a frame graph.

For example, you have the opportunity to implement “compiler-like optimizations” (as a general term). For example, as in the case at the top right here, Pass 1 is outputting a texture “Tex” which is unused by the rest of the graph. As a result, a good frame scheduler can realize this, and simply drop the whole part of the graph that does this needless work. One might think “Why not just delete the dead code?”, but this can turn out to be useful. For example, a pass may have some optional outputs (such as debug info) which it only wants to generate if there exists another part of the codebase that actually consumes it. This is one case where we can use global knowledge of the frame to our advantage.

Another important task of the scheduler is to automatically insert barriers at points during the frame. For example, if you have one pass which writes to a texture and a subsequent pass that reads the texture, it’s important to insert any necessary memory barriers between these two passes. This can also be used to implement automatic resource transitions.

There are many interesting tradeoffs that a scheduler needs to have, which are not obvious to optimize. An important factor of the scheduler's implementation is to come up with the right heuristics that optimize your workload. One such heuristic is the management of latency between passes.

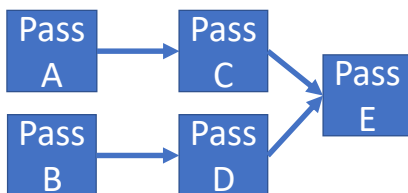
For example, in the example on the left, tasks A and B are run immediately one after the other. This might cause some stalling, since work can't progress until A is done. On the other hand, the memory that was touched by A is still warm when B runs, so this schedule may be better for cache coherency.

On the other hand, with the schedule shown on the right, task A is launched as soon as possible, and task B is put off until later. This means that it's more likely that task A is already done by the time task B arrives, so it won't stall as much. However, it's possible that you lose some cache coherency, and it's possible for the total memory overhead to be greater in this setup (since we're running many tasks in parallel.) In general, these are tough decisions. I'm not sure what the best solution is.

Finally, when you've decided which passes should run when, you can finally visit each node in the graph and invoke some kind of function to record the commands that belong to that node. That's where the tricks from everything previously mentioned in this talk get used.

“List Scheduling” Approach

1. Assign priorities to tasks. (Heuristic)
2. Run highest priority runnable task.
3. Repeat (2) until done.



Schedule:

A C B D E

I wanted to make the idea of scheduling a bit more concrete, and this is the closest well-known algorithm I could find: The so-called “List Scheduling” algorithm.

It’s a simple algorithm at the core. First, assign priorities to the tasks. Then, in a loop, pick the highest priority task *that has its resource requirements satisfied* and execute it.

The difficult part of this algorithm is the heuristics you use to decide the priorities. You can also see this as assigning weights and solving for the constraints they impose, as done by Hans-Kristian in his work.

In the example on this slide, we have a very simple schedule of passes. For the sake of example, each pass is assigned a priority in alphabetical order. Since A is the highest priority task, that gets schedule first. The next highest priority task is B, but let’s suppose that there isn’t enough memory to run B right now. Instead, we run the next highest priority task: C. Finally, now that A and C have executed, it may be that we now have enough memory to run the rest of the graph (B,D,E.)

Memory and Object Lifetime



Potential Schedule:



If T2 can reuse T1:



See: D3D12 CreatePlacedResource

This slide highlights another aspect of scheduling that I wanted to make a little bit more concrete: The allocation of memory and objects throughout the execution of the graph.

Let's suppose that we have the very simple task graph shown above. When we execute this code, we need to make sure that the objects used to store the inputs and outputs of each task are allocated in memory at the time at which they are used.

For example, in this first potential schedule, the textures T1 and T2 are allocated prior to running task A. Once task A finishes, it is now possible to delete T1, since it is no longer used. From there, we allocate the memory for the output of B, run B, and we finish by deleting the now unnecessary T2.

A potential improvement in this schedule can be made if T1's memory can be reused for T2. For example, maybe T1 and T2 actually have the same size and format. In this case, it might be possible to implement the transformation from T1 into T2 in-place, and thus reuse the memory. This is one kind of transformation that could be done by a frame scheduler.

Again, when I say “New” and “Delete” here, I’m talking about something a bit abstract that doesn’t directly map to these GPU APIs. See the “Remarks” section in the documentation for D3D12’s `CreatePlacedResource` if you want to get a better idea of what I mean.

In Summary

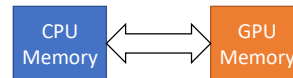
Now I'll just summarize what I talked about today.

In Summary

Part 1

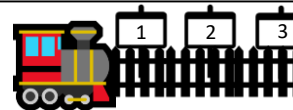
Memory Management

- System Memory, Video Memory
- Virtual Memory between them



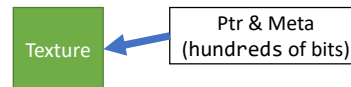
Command Lists

- CPU builds schedule for GPU to consume
- Synchronize with fences



Descriptors

- Address + metadata
- HW-friendly GPU object reference



In part 1 of this talk:

- We first looked at memory management. We talked about the differences between system memory and video memory, and the relationships between them using virtual memory and DMA engines.
- We then looked at command lists, which are a way for the CPU to build a schedule of work for the GPU to do. We also saw that fences can be used to synchronize the CPU and GPU between the execution of work.
- Finally, we briefly looked at descriptors, which are something like a little struct (100s of bits) that stores the address of an object's storage, as well as the metadata used to represent it. These descriptors are hardware friendly view of objects, which can be used in GPU code.

In Summary

Part 2

Ring Buffers

- CPU->GPU streaming primitive



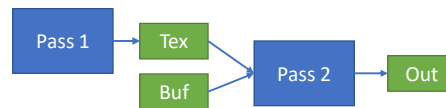
Parallel Command Recording

- Task-parallelizing command recording
- Regular vs. Irregular cases



Scheduling Work & Memory

- Optimize frame globally
- Manage memory & object lifetime



The second part of this talk covered the following topics:

- First, we talked about ring buffers, and described how they can be used as a primitive for streaming data from the CPU to the GPU.
- Next, we talked about recording command lists in parallel, using task-parallelism. We went a bit deep into the different ways in which the cases of regular tasks and irregular tasks can be handled.
- Finally, we talked about the overall scheduling of a frame of rendering. The main goal of this was to be able to apply global optimizations to the rendering passes that make up a frame of rendering. This consists of both optimizing the work and the memory. I have heard suggestions that it's important to leverage this kind of global knowledge to use the new APIs effectively.

Acknowledgements

Thank you for early feedback on this presentation!

- Scott Wardle (EA)
- Mauricio Rovira (GFX @ UVic)

Thank you for answering various questions and sharing inspiring ideas!

- | | |
|-------------------------------|----------------------------------|
| • Dave Oldcorn (AMD) | • Darrel Palke (Intel) |
| • Yuriy O'Donnell (EA) | • Joshua Barczak (Firaxis Games) |
| • Hans-Kristian Arntzen (ARM) | • Matthäus Chajdas (AMD) |
| • Andrew Lauritzen (EA SEED) | • Matt Pettineo (Ready at Dawn) |
| • Jason Ekstrand (Intel) | |

Before I conclude my talk, I'd like to thank the people who helped me put it together.

First I'd like to thank Scott and Mauricio for going over my talk to help me practice it.

Then I'd like to thank all the people here at the bottom of the slide (& maybe more, sorry if I forgot!) for being able to answer various questions I had on the topics of DirectX and GPUs while I was preparing this talk, and for sharing interesting ideas, some of which made their way into this talk.

References

- [Laine et al. '13] <http://research.nvidia.com/publication/megakernels-considered-harmful-wavefront-path-tracing-gpus>
“Megakernels Considered Harmful: Wavefront Path Tracing on GPUs”
Authors: Samuli Laine, Tero Karras, Timo Aila
- [WDDM2] <https://docs.microsoft.com/en-us/windows-hardware/drivers/display/wddm-2-0-and-windows-10>
[D3D12 CL] [https://msdn.microsoft.com/en-us/library/windows/desktop/dn903537\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn903537(v=vs.85).aspx)
[GCN3] <http://gpuopen.com/compute-product/amd-gcn3-isa-architecture-manual/>
- [RingBuffersAndQueues] <https://fgiesen.wordpress.com/2010/12/14/ring-buffers-and-queues/>
Author: Fabian Giesen
- [Work Stealing] <https://www.youtube.com/watch?v=iLHNF7SgVN4>
CppCon 2015: Pablo Halpern “Work Stealing”
- [Hyper '09] <http://www.fft.w.org/~athena/papers/hyper.pdf>
“Reducers and Other Cilk++ Hyperobjects”
Authors: Frigo, Halpern, Leiserson, Lewin-Berlin
- [RTR '08] <http://www.realtimerendering.com/>
“Real-Time Rendering”, Third Edition.
Authors: Akenine-Moller, Haines, and Hoffman.
- [FrameGraph] <http://www.frostbite.com/2017/03/framegraph-extensible-rendering-architecture-in-frostbite/>
“FrameGraph: Extensible Rendering Architecture in Frostbite”
Author: Yuriy O'Donnell
- [RenderGraphs] <http://themaister.net/blog/2017/08/15/render-graphs-and-vulkan-a-deep-dive/>
“Render graphs and Vulkan – a deep dive”
Author: Hans-Kristian Arntzen

If you want to check out the papers and articles I linked in this talk, here are all the links in one place.

The End.

Thank you!

Questions? Comments?

Thank you for your attention! If you have any questions, feel free to ping me on Twitter @nlguillemot or by e-mail (nlguillemot@gmail.com).