Hello everyone.

My name is Charlie Birtwistle, and I'm a Senior Principal Engineer at Activision Central Tech.

I'm presenting today with my colleague Francois Durand, Technical Director at Beenox.

We'll be talking about the task graph renderer, which we developed for use in our technology stack at Activision.

AGENDA

- Background
- Task graph API & backend
- Initial results
- Evolution
- Future work

First a quick agenda of what we are covering today.

We'll start with some background and historical context of how we came to develop task graph, particularly looking at the impact of modern graphics APIs.

Next, we'll cover the task graph API, and discuss some of the backend implementation details.

We'll show some initial results of the first task graph integration.

After that we'll discuss some of the improvements we've made in the years since.

Finally, we'll present a few ideas on future work.

BACKGROUND – MODERN APIs

- Cross generation titles, 2013-2014 era
  - Moderate number of passes
  - Limited compute
  - RTs global allocs
  - Very little memory reuse
  - PS4 low level API
    - Manual synchronization
  - Xbox One DX11
    - Automatic synchronization
    - Artisanal RT placement in ESRAM

Our first exposure to a modern, low-level API was over a decade ago, during development of our first titles for the 8th generation of consoles.

At this time DirectX 12 was not yet available. However, the low-level API on PlayStation 4 had many of the same concepts.

By today's standards these were simple renderers, with only a moderate number of passes. Compute usage was limited, typically just ports of work that was running on SPU on PlayStation 3, such as skinning. We had few render targets which were all setup as global allocations.

The PlayStation 4 version introduced us to manual synchronization between GPU workloads. However, this could be handled in a trivial fashion, for example when unbinding a render target, we would flush the caches and wait, so work was visible to any subsequent rendering.

Furthermore, the Xbox One version at this time could only be deployed on DirectX 11, so we didn’t need to worry about synchronization there at all. The only complication was the limited ESRAM for render targets. But with the low number of RTs, we simply

did some artisanal placement at specific addresses based on known lifetimes.

In short, though we were introduced to some of the challenges around synchronization and transient memory allocation, they weren't really a major problem.

BACKGROUND – MODERN APIs

- 8th generation exclusive titles, 2015-2018 era
  - Increasing number of passes [1]
  - Significant compute
  - RTs dynamic allocs
  - PS4 API
    - Async compute
    - Low level synchronization
  - Xbox One DX12
    - High level manual resource barriers
    - Lots of validation errors
  - PC DX11

Fast forward a few years and we are now developing titles that are exclusively targeting PlayStation 4 and Xbox One.

Because of the power these machines offered, the number of rendering passes exploded. Among other features, F+ or deferred rendering, particle lighting, volumetrics, and screen space reflections were added. See presentation here for the kind of details.

All these new features had significant memory costs, so a new dynamic allocation system was added for RTs, allowing some memory reuse over the course of the frame. Although better than no aliasing, the variable memory costs would contend with streaming. Additionally, the allocations were on the CPU timeline, requiring complex handling for our multi-threaded rendering.

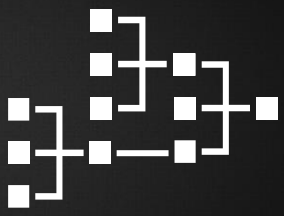
On PS4 the major addition was async compute. We would place certain passes on the async queue, and carefully code the cross-queue synchronizations. Under this environment converting a pass from graphics to async queue could be a significant undertaking.

Meanwhile on Xbox One, we were transitioning to the newly released DirectX 12. This required adding resource barrier calls at a high level across the rendering code. Few people ran the DX12 version, preferring the PC DX11 or PS4 versions, so these transitions were often broken with associated visual errors. Every week someone would have to spend a whole day patching up all the barriers.

It was clear these modern rendering requirements had started to put serious strain on our rendering architecture.

BACKGROUND – MOVING FORWARD

- GDC 2017
  - Several render graph presentations [2] [3]
  - Seemed to be a solution to our problems



As we started to plan for our second generation of renderers on these consoles, it was clear that we needed a better solution.

Fortunately, these problems were not unique to us, and the topic was covered at GDC 2017. There were a couple of presentations covering render graphs, and how they could solve some of the problems around synchronization and RT memory usage.

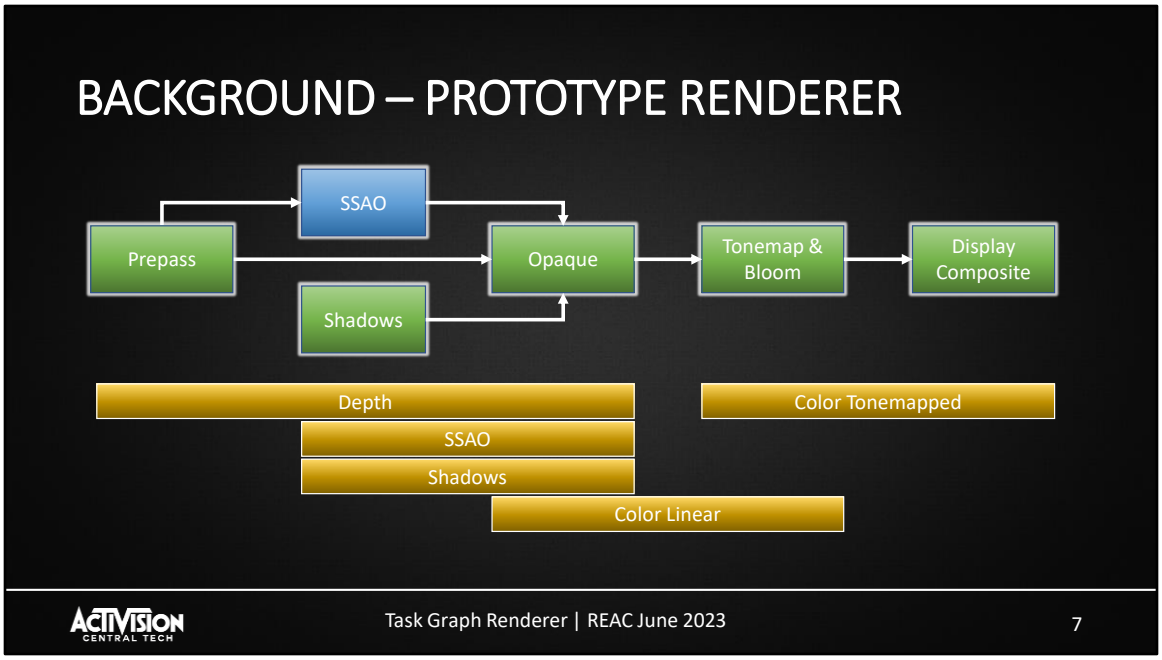
BACKGROUND – R&D

- Central Tech Research 2018
  - Several months R&D
  - Targeting future titles
- Prototyping
  - Standalone app DX12 / Vulkan
  - Simple level viewer
    - Prepass
    - Shadows
    - SSAO (optional async)
    - Tonemapping and Bloom

Rather than jump right in and attempt to program a render graph system into one of our renderers, we first wanted to experiment and iterate on the idea from a research and development perspective.

Starting in early 2018, we began to iterate in a standalone “test bed” application. We developed a simple level viewer, using a minimal platform layer targeting DirectX 12 and Vulkan.

Initially we wrote the prototype in the traditional immediate mode fashion, with a prepass, cascaded shadow maps, screen space AO, and some minimal post FX.



A rough overview of the dependencies between passes can be seen here, with graphics work as green boxes, and compute work as blue. The gold bars underneath show the lifetime of various render targets used across the frame.

Although very simple compared to our production renderers, this setup could test the key areas of synchronization & transitions, aliased render targets, and a flexible GPU schedule. The SSAO in particular could be parallelized with shadows, either using implicit pipelined compute, or explicitly executed on async queue.

BACKGROUND – C++ CODE DRIVEN PROTOTYPE

- Code-driven approach
- Per task
  - Static setup func
  - Virtual render
  - Member data
- Setup renderer func
  - C++ based

```
Opaque& Opaque::SetupTask( TgBuilder& builder )
{
      TgImageDesc colorDesc = { “color”, 1920, 1080, TG_R11G11B10F };
      TgImageDesc depthDesc = { “depth”, 1920, 1080, TG_D32F };
      Opaque& opaque = builder.AddGfxTask<Opaque>( “opaque” );
      opaque.m_color = builder.CreateImage( colorDesc, TgAccess::RenderTarget );
      opaque.m_depth = builder.CreateImage( depthDesc TgAccess::DepthTarget );
      return opaque;
}

Tonemap& Tonemap::SetupTask( TgBuilder& builder, TgResource dest, TgResource source )
{
      Tonemap& tonemap = builder.AddGfxTask<Tonemap>( “tonemap” );
      tonemap.m_dest = builder.Write( dest, TgAccess::RenderTarget );
      tonemap.m_source = builder.Read( source, TgAccess::FragmentShader );
      return tonemap;
}

void SetupRenderer( TgBuilder& builder )
{
      TgResource backBuffer = builder.GetBackBuffer();
      Opaque& opaque = Opaque::SetupTask( builder );
      Tonemap& tonemap = Tonemap::SetupTask( builder, backBuffer, opaque.m_color );
}
```


Initially we followed the code-driven approach that had been well documented. In this setup you have a unique class per task type. This could be completely handwritten or partially automated using lambdas. The class would have a virtual render function, and some members for at least storing handles to resources required by the render function. There would also be a setup function, taking a builder class, and optional handles to resources created elsewhere in the frame.

In the code snippet here, we can see what an Opaque and Tonemap might look like. In this case you can see the opaque task creates 2 new images, with color and depth access. The tonemap writes directly to the backbuffer, so it specifies write access there, and read access to a color buffer written by the Opaque pass.

Having made a version of our prototype renderer using this approach, we were able to take a step back and make a couple of observations.

The first observation is that the setup function looks like a table, where each line of code is making an entry in a vector. Whilst populating this vector, often the only variables that are unique to the task instance are the resource handles.

The second observation is we have duplication. First we need to declare the tasks member data in the class, for example the color and depth handles for the opaque task. Then we need to assign them via the setup function, with the additional access or creation parameters.

It might not look too bad in this contrived example, but we know equivalent passes in our production renderers can be accessing tens of transient resources each. And the whole rendering pipeline is made up of 100s of passes.

One thing we want to avoid is excessive boiler plate code, and this looks like it will end up with a lot of typing.

BACKGROUND – C++ DECLARATIVE PROTOTYPE

- Declarative approach
- Data-driven setup
- Per task
  - Macro definition
  - Render function
- Setup renderer func
  - C++ based DSL

```
// macro based definitions

BEGIN_TASK( Opaque )
      COLOR_CREATE( color, 1920, 1080, TG_R11G11B10F )
      DEPTH_CREATE( depth, 1920, 1080, TG_D32F )
END_TASK()

BEGIN_TASK( Tonemap )
      COLOR_TARGET( dst )
      TEXTURE_READ( src )
END_TASK()

// setup renderer

void SetupRenderer( TgBuilder& builder )
{
      TgResource backBuffer = builder.GetBackBuffer();

      TgResource color, depth;
      cOpaque( builder, color, depth );

      cTonemap( builder, backBuffer, color );
}
```


Taking into account the observations we made from our first implementation, we tried another approach.

This time, we data-drive the setup function. Rather than having a uniquely coded setup function per task type, we have a generic version, which uses a task definition to perform the equivalent process of adding tasks and resource access properties to a graph builder object.

To do this, we will use macros, which will allow us to replace the class declarations entirely.

I will cover how these macros work exactly a little later, but for now you can see we are able to encode the same information as before, and we end up with a similar looking renderer setup function. Everything is a lot more concise, and overall much more readable than the first approach.

BACKGROUND – LUA PROTOTYPE

- Lua based DSL
  - Runtime render bind
  - Hot reloading
  - Offline analysis
- Other DSL languages
  - C++/HLSL like

```
-- definitions

begin_task( “Opaque” )
      color_create( “color”, 1920, 1080, TG_R11G11B10F )
      depth_create( “depth”, 1920, 1080, TG_D32F )
end_task()

begin_task( “Tonemap” )
      color_target( “dst” )
      texture_read( “src” )
end_task()

-- setup renderer

backBuffer = TgGetBackBuffer()

color, depth = Opaque()

Tonemap( backBuffer, color )
```


After experimenting with the C++ DSL, we attempted to move the definitions and renderer setup function into a scripting language.

Using essentially the same approach, we were able to replicate the behavior in Lua.

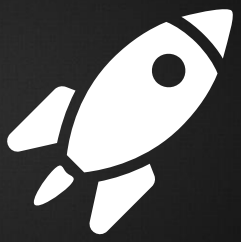
One of the positives of this was we were now able to hot-reload the renderer. Being able to quickly comment out passes or rearrange them while the app was still running felt very powerful.

We were also able to load the script into an entirely separate analysis tool, which could estimate memory usage, detect logic errors, and so on.

We also thought about whether there would be more suitable languages. Perhaps a custom C++ or HLSL like language we could either compile directly into the game or optionally interpret with hot reloading.

BACKGROUND – GREENLIGHT

- Production greenlight mid-2018
- C++ declarative approach
  - No time for custom language
  - Concerns around lua
- 2nd gen renderers for 8th gen consoles
  - Geometry pipeline [4]
  - VRS renderer [5]
  - Terrain virtual texturing
  - Raytracing [6]
  - Streaming rewrite
  - Big map tech [7]



Inevitably though, as with all research projects, there comes a time when you need to apply your findings.

After some demonstrations of the prototype to key stakeholders, we were given the green light to begin integrating task graph into our production technology.

The approach we decided on was the C++ declarative approach, with the Lua or custom language DSL considered a little too ambitious for a first step.

This was a period of great upheaval for our renderers. We had shipped our first set of 8th gen titles and now understood much better what we could achieve. Our rendering pipelines were getting a lot more sophisticated, with extensive changes coming to the geometry pipeline, and we were adding tech like variable rate shading, terrain virtual texturing, and raytracing as well as rewriting our streaming and developing techniques to allow us to handle much bigger worlds.

In retrospect this was not the ideal time to change the renderer to a new paradigm.

TASK GRAPH API – REQUIREMENTS

- Easy to understand, scalable interface
- No boiler plate
- Handle common use-cases
  - Temporal resources
  - Dynamic scene resolution
  - Split-screen
- Conditional render paths
- Automatic barriers
- Automatic resource aliasing/overlaps
- Scheduled render order
  - Optimize performance - barrier batching
  - Optimize memory – free resources sooner
  - Program order - debugging

Before starting the integration, we wanted to double check our requirements. It's all very well having a simple prototype, but when dealing with a full production renderer, there are likely complications and edge cases that need to be handled in the API. We therefore audited the renderer and made sure we could handle everything.

We also needed to ensure our API was going to be easily used by the many rendering engineers we have scattered across various time zones, without extensive training.

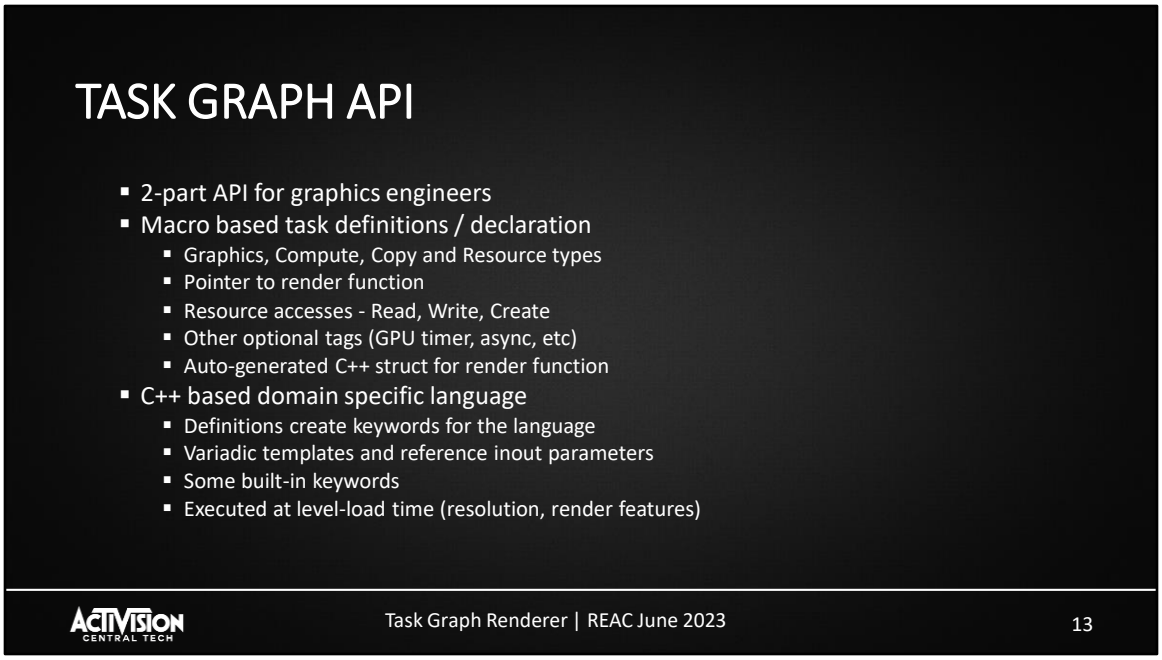
It had to be easy to understand, with a scalable interface and a minimum of boiler plate.

We wanted to handle use cases that we never actually tested in our prototype. This included temporal techniques, optimizations like dynamic scene resolution, and of course split-screen.

Unlike the prototype, we also needed some conditionality from frame to frame, such as whether to use an expensive depth of field algorithm or not.

And of course, we wanted to get the standard benefits of a render graph system, and

free our engineers from worrying about barriers, resource memory aliasing, or even the final rendered order of the frame on GPU.



As we illustrated in the prototype section, the API has 2 major components.

The first part is the macro-based task definitions. These are available in Graphics, Compute, Copy and Resource types. At the minimum here there's a pointer to the render function, and a list of resource accesses. These being either read/write operations, or creation of new resources. There are also additional tags which don't indicate resource access but some other property, such as preferring to run this task on async queue for compute tasks.

The second part is the SetupRenderer function, where the actual rendering pipeline is laid out, one task at a time. This is a C++ based domain specific language, using keywords created by the definitions, and specifies how resources and tasks are combined to create the overall rendering pipeline. This setup function is typically invoked just once per level during loading. More detail on that a bit later.

Let's cover each of these in a little more detail with an example.

TASK GRAPH API – DEFINITION DETAIL

```
BEGIN_TASK( GenerateSunVisibility )
      COLOR_TARGET_CREATE( shadowVisibility, RELATIVE_PARAM(), RELATIVE_PARAM(), FORMAT_SUN_VISIBILITY )
      COLOR_TARGET_CREATE( shadowTransmittance, RELATIVE_PARAM(), RELATIVE_PARAM(), FORMAT_SUN_TRANSMITTANCE )
      DEPTH_TARGET_READ( sceneDepth )
      TEXTURE_READ( floatZfull )
      TEXTURE_READ( sceneTangentFrame )
      TEXTURE_READ( sunshadowCascades )
      TEXTURE_READ( sunshadowTranslucentExtinction )
      CONDITION_FUNC( ShouldRenderShadows )
END_TASK()
```

- Function pointer “GenerateSunVisibility“, list of resources, and some additional tags.
- 2 x color targets, created for this task. Size derived from other resources in this case
- Slots for depth, floatz, tangent space, shadow cascades, shadow translucent extinction

Here we have a task definition for calculating sun shadow visibility in screen space.

As can be seen in this snippet, we have begin/end tags, and a list of the resources created or used by the task. Under the hood, this macro serves 3 key purposes.

Firstly, it fills out an array of resource access or creation data. This is the task definition data.
Secondly, by processing the macro again, it fills out a C-struct of resource handles. This will be used directly by the rendering function.
And thirdly, it creates a variadic template that allows the task to be used like a function call in the DSL.

In this example we are creating 2 new render targets. The CREATE variant of the COLOR_TARGET tag means we can specify the resource description directly in the tag. Rather than use a hard coded size, in this case we are specifying the width and height with the special values RELATIVE_PARAM. This means the dimensions will be determined by the size of the resources passed to the task.

Next, we have 4 texture reads, of resources created elsewhere in the frame.

Finally, there’s a conditional function which means on a per-frame basis this task might be enabled or disabled. More on that later.

TASK GRAPH API – DSL USAGE DETAIL

```
void RendererSetup( TgBuilder& builder, GraphConstants constants )
{
     TgHandle nullTexture = builder.GetNullTexture();
     ...
     TgHandle sunVisMask, sunTransMask;  // invalid handles
     if ( constants.sunshadows )
          cGenerateSunVisibility(   builder, sunVisMask, sunTransMask, sceneDepth,
                                    floatZfull, sceneTangentFrame, shadowCascades, shadowTrans );
     else
          sunVisMask = sunTransMask = nullTexture;
     ...
}
```

- sunVisMask, sunTransMask passed via reference and initialized by func
- Other handles are from elsewhere in the frame

In order to use this task in the renderer, we need to instantiate it. We do that using the C++ domain specific language we showed earlier.

Here you can see we are using GenerateSunVisibility like a function call, with a list of parameters. If you look carefully, you can see the small “c” added before Generate. This is the variadic template, created by the macro definition. Internally it builds a stack of parameters by stripping them off, then passes the result to a data-driven setup function, along with the corresponding task definition data. These parameters are all passed by reference, so we can modify them if necessary. In this case, sunVisMask and sunTransMask are both invalid handles, and are initialized by the task call.

The other key thing to note here is that the usage of this task is conditional, based on the GraphConstants that are passed to the function. If sun shadows are disabled for this level, we skip the task call completely. Instead, we assign the handles with a null resource.

TASK GRAPH API – RENDER CALLBACK DETAIL

```
void GenerateSunVisibility( GfxRenderContext gfxContext, const GfxTaskInfo *taskInfo )
{
     GenerateSunVisibility_Struct params( taskInfo );

     // params.shadowVisibility, params.shadowTransmittance and params.sceneDepth bound as RTs by system
     GenerateSunVisibility_Leaf( gfxContext, params.floatZfull, params.sceneTangentFrame,
                                             params.sunshadowCascades, params.sunshadowTranslucentExtinction );
}
```

- Function name is the task name
- <TaskName>_Struct autogenerated type
- Can call a leaf function shared with traditional renderer

Let's take a look at the rendering call-back function used by this definition.

Here we can see the function has the same name as the task, so the name specified in the BEGIN_TASK macro is the rendering function itself.

The other interesting point is we can use the GenerateSunVisibility_Struct for accessing the parameters passed to the task.

In this example, we call a separate internal function, passing the parameters from the struct. During initial integration of the task graph, this allowed us to share most of the rendering code between the traditional and task graph renderer.

TASK GRAPH API – EXTENDED DSL EXAMPLE

```
void DOF_Apply( TgBuilder& builder, TgHandle sceneColor, TgHandle floatZ, TgHandle velocity, TgHandle scope )
{
    cConditionBegin( builder, DOF_GetEnabled );
    TgHandle dofTile1;
    {
        TgHandle dofPingpong;
        cDOF_DownsampleTile_Horizontal( builder, dofPingpong, sceneColor, floatZ );
        TgHandle dofTile0;
        cDOF_DownsampleTile_Vertical( builder, dofTile0, sceneColor, floatZ, dofPingpong );
        cDOF_TileNeighor( builder, dofTile1, dofTile0 );
    }

    TgHandle dofHalfColorPingPong, dofHalfPrepass;
    cDOF_CreateTemporal( builder, dofHalfColorPingPong, dofHalfPrepass, sceneColor );

    TgHandle dofHalfColor;
    cDOF_Prepass( builder, dofHalfColor, dofHalfPrepass, dofHalfPrepass, dofHalfColorPingPong, floatZ, dofTile1, sceneColor, velocity );

    TgHandle dofHalfAlphaPingPong;
    cDOF_CircularFilter( builder, dofHalfColorPingPong, dofHalfAlphaPingPong, dofHalfPrepass, dofHalfColor, dofTile1, sceneColor );

    TgHandle dofHalfAlpha;
    cDOF_Median( builder, dofHalfColor, dofHalfAlpha, dofHalfColorPingPong, dofHalfAlphaPingPong );

    cDOF_Sharpen( builder, dofHalfColorPingPong, dofHalfColor );

    cDOF_Upsample( builder, sceneColor, floatZ, dofTile1, dofHalfColorPingPong, dofHalfAlpha, dofHalfPrepass, scope, sceneColor );
    cConditionEnd( builder );
}
```


Here we have an extended example showing a complete depth of field algorithm implemented in the DSL.

For clarity we implement this in its own function, which helps show that the algorithm as a whole takes these 4 render targets. All other resources are internal to this algorithm and are not used elsewhere in the frame.

I won’t describe what all these passes do, the point is this is a common approach where many individual techniques are themselves implemented by a large number of tasks and intermediate resources.

TASK GRAPH API – TEMPORAL RESOURCES

```
BEGIN_TASK( DOF_CreateTemporal )
     COLOR_TARGET_CREATE_TEMPORAL( halfColor, RELATIVE_DIV2(), RELATIVE_DIV2(), FORMAT_DOF_COLOR, 2 )
     COLOR_TARGET_CREATE_TEMPORAL( halfPrepass, RELATIVE_DIV2(), RELATIVE_DIV2(), FORMAT_DOF_PREPASS )
     TEXTURE_READ( sceneColor )
END_TASK()


void DOF_CreateTemporal( GfxRenderContext gfxContext, const GfxTaskInfo *taskInfo )
{
     DOF_CreateTemporal_Struct data( taskInfo );

     // data.halfColor[0] and data.halfPrepass already bound as RTs
     // data.halfColor[1] is previous frame's RT
     DOF_CreateTemporal_Leaf( gfxContext, data.halfColor[1], data.sceneColor );
}
```

- Temporal resources are first class concepts in the API
- Auto-rotation per frame, nulling out previous layers after reset

Now we'll cover a few specific features of the API that allow us to handle common use cases.

Temporal resources are extremely common in modern renderers, so we added support for these as first-class objects in the API.

The system itself will handle the per-frame rotation, and null out previous frame handles in the case of a camera cut, for example. This vastly simplifies what used to be a very manual process.

TASK GRAPH API – SUBRESOURCES & PARAMS

```
BEGIN_RESOURCE_TASK( CreateMipChain )
      CREATE_TEXTURE_EX( mips, RELATIVE_DIV2(), RELATIVE_DIV2(), 1, RELATIVE(), TASK_PARAM(), 1, 1, Flags::None )
      // create( name, width, height, depth, format, mips, arraySlices, multisamples, flags )
      TEXTURE_REF( src )
END_RESOURCE_TASK()

BEGIN_TASK( DownSamplePS )
     COLOR_TARGET( dst )
     TEXTURE_READ( src )
END_TASK()

TgHandle sceneColor; // valid from earlier
const uint mipCount = 4;
TgHandle downsampledSceneColor;
cCreateMipChain( builder, downsampledSceneColor, sceneColor, mipCount );

DownsamplePS( builder, mipChain.Mip( 0 ), sceneColor );
for ( uint mip = 1; mip < mipCount; mip++ )
      DownsamplePS( builder, mipChain.Mip( mip ), mipChain.Mip( mip - 1 ) );
```

- DownsamplePS used to populate a mip chain of specified size
- TgHandle.Mip(X) to dereference specific subresouces

Another couple of important features here are being able to pass arbitrary values from the setup function into the definitions. In this case, this allows creating a mip chain with the specified number of mip levels. Often there are resource setup parameters that are data driven at run time so its not always suitable to hard code the definition. Another example here would be a shadow map size. On PC there are likely many graphics options setting for the sizing of the shadow cascades.

The other important feature demonstrated here is being able to dereference sub-resources from the resource handle. In this case we are down sampling the scene color RT into a mip chain RT.

TASK GRAPH API – NULL RESOURCES

```
BEGIN_TASK( DrawOpaque )
       COLOR_TARGET( color )
       COLOR_TARGET( albedoSSS )
       COLOR_TARGET( auxSSS )
       // lots of other parameters
END_TASK()

TgHandle sceneColor, albedoSSS, auxSSS;
if ( constants.subSurfaceScatter )
{
       cCreateSceneColorSSS( builder, sceneColor, albedoSSS, auxSSS );
}
else
{
       cCreateSceneColor( builder, sceneColor );
       // set optional RTs to null
       albedoSSS = auxSSS = builder.GetNullTexture();
}

cDrawOpaque( builder, sceneColor, albedoSSS, auxSSS, ... );
```

- Null resources used extensively to disable features
- Allows a single uber-definition to have optional features

Lastly on the front-end API, another key feature is being able to pass around null resources.

Depending on graphics settings, passes or entire pipelines might be completely disabled. In this case we set any associated resource handles to null. This mean we can use uber definitions and just pass null handles for the optional resources.

In this case we have a task that can use up to 3 render targets at the same time. Depending on a subsurface scatter option, we either pass 1 or 3 valid RTs to the task. This means we don't duplicate task definitions to handle permutations of enabled options.

TASK GRAPH BACKEND – BUILD STEP 1

- Run “RendererSetup” DSL front-end
  - With level and platform specific graph constants
  - Serialized task list in program order
- Build dependency graph
  - Recursive based on resource access
  - Starting with external resources (backbuffer and others)
  - Implicitly culls tasks that have no impact (debug passes)
- Schedule graph into GPU order
  - Barrier batching strategy
  - Render pass compatibility strategy
- Create heaps and resources
  - All dynamic resolution steps
  - Efficiency depends on the platform capability

Heavyweight
Level load only*
10-100+ ms

I’ll now cover a few details of the backend implementation and how we go from the render pipeline code written in our DSL to recording command buffers during rendering.

We have a build process consisting of 2 main parts. A heavyweight part that happens on level load, and a lightweight part that happens per-frame.

At level load time, we execute the RendererSetup function, with graph constants from that level’s config file combined with per-platform settings. These constants are for graphics options that are fixed for the duration of the level. This could be something like whether we have sun shadows or the water system enabled. You could imagine an underground bunker style level might not need sun shadows. These parameters could also be specific quality settings, such as boosting the capacity of a GPU particle system for a snowy level that heavily relies on particles.

After we’ve executed this function, we have a list of tasks and resources. From this we build a dependency graph from the resource accesses. This process starts from the external resources such as the back buffer, and works backwards pulling in dependent tasks. In this way, any tasks that don’t contribute are discarded.

Next, we schedule the graph into a GPU render order using strategies to reduce barriers or number of render passes. Graphics engineers generally don't care too much about the actual final render order on GPU, though we may use artificial dependencies between tasks to influence the render order for optimization purposes. This could be to reduce memory requirements by limiting the lifetime of certain resources.

Finally, we create memory heaps and place all needed resources into those heaps. This would include all the overlapping resolution steps for resources with dynamic resolution scaling.

As shown on the slide this whole process is rather expensive, taking at least 10 milliseconds but potentially over 100ms on platforms where memory allocation and resource creation can be particularly expensive.

I should note that although this step only usually occurs at level load for the final game, during development this can happen any time the graph constants are changed using debug tools.

TASK GRAPH BACKEND – BUILD STEP 2

- Evaluate conditions
  - Based on tasks tagged with conditional behavior
- Build permutation
  - Unique list of enabled tasks
  - Barriers / synchronization / render passes
  - No resource allocation here
- Cache & reuse results
  - Small cost for uncached permutation

Lightweight
Per frame
0-1 ms

The second step is some lightweight per-frame processing.

Earlier we showed that tasks can be tagged with a conditional function. This is to allow changes to the renderer during gameplay without doing the full build process we discussed on the previous slide. We call these graph permutations.

The way this works is rather straightforward. At the start of rendering we evaluate all the conditions that we found in the graph, and create a bitfield key. This key will enable or disable tasks based on their conditional settings. We iterate this enabled task list, and build barriers, synchronization such as cross-queue syncs, and render passes if needed.

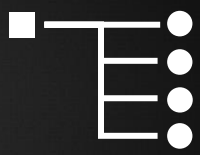
We don’t do any resource allocation so this process is quite fast. We also cache the results, so switching quickly between several commonly used permutations won’t introduce additional processing.

It would be common for us to have around 5-10 unique per frame conditions. This could work out to over 1000 possible permutations of enabled tasks.

As is clear from this setup, this means we pay the memory cost for the worst possible case – everything enabled – rather than trying to adjust memory usage dynamically.

TASK GRAPH BACKEND – RENDER

- Iterate permutation task list
  - Issue barriers
  - Add debug markers / GPU timers
  - Optionally flush cmd buf to queue
  - Call user render function
- Initially single threaded
  - Artisanal multi-threading
    - Task internal (DrawPrepass / DrawOpaque) [8]
- Async tasks thread
- Full multi-threading
  - More detail later



Per frame
5-10 ms

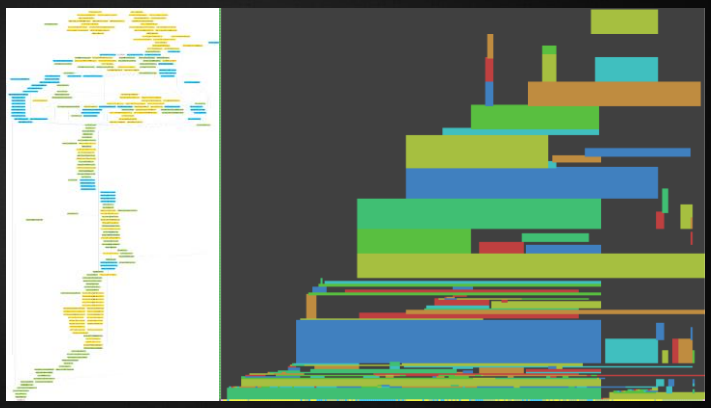
After we've completed the 2 steps above, we are ready to render. This is now a fairly simple process of iterating the task list, issuing barriers, debug markers, and then calling the user render functions. We also handle queue submission here; we'll flush command buffers to queues a few times during rendering to ensure we keep the GPU well fed.

In our initial implementation this code was single threaded, with some artisanal multi-threading handled inside the task render function themselves. This is discussed in a previous presentation.

We then moved to also recording async queue tasks in a separate thread, before later moving to a full multi-threading system where we split the tasks between all available cores. More detail on that later.

INITIAL RESULTS – 2019

- 236 tasks (185 GFX, 52 Async)
- 258 resources
- 92 barriers, 328 elements
- 5 conditions, 32 permutations
- 576mb heap memory PS4

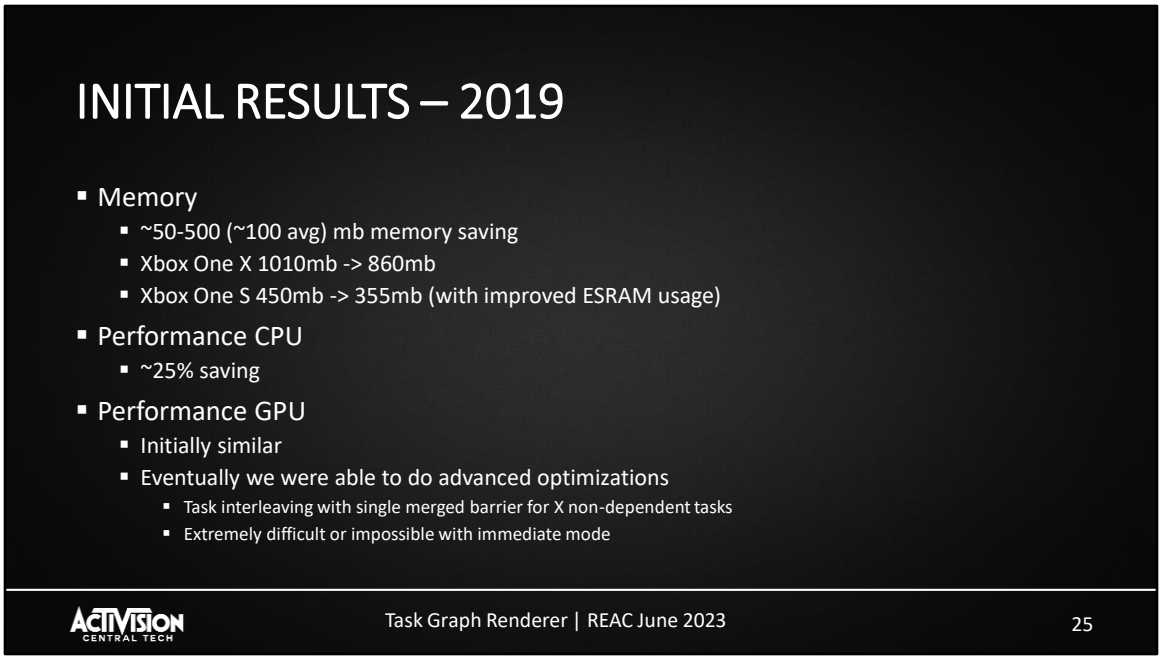


I'll now discuss some initial results when we first deployed this system around 2019.

At this time while we assessed our results, we were able to switch between the traditional immediate mode renderer and the new task graph renderer dynamically at run time.

The graphic here shows a dependency graph, and a heap view of the transient resource memory over the frame. The dependency graph is showing graphics work in green, compute in yellow, and async compute in blue.

From the heap view you can see we are getting some reasonable memory reuse over the frame.



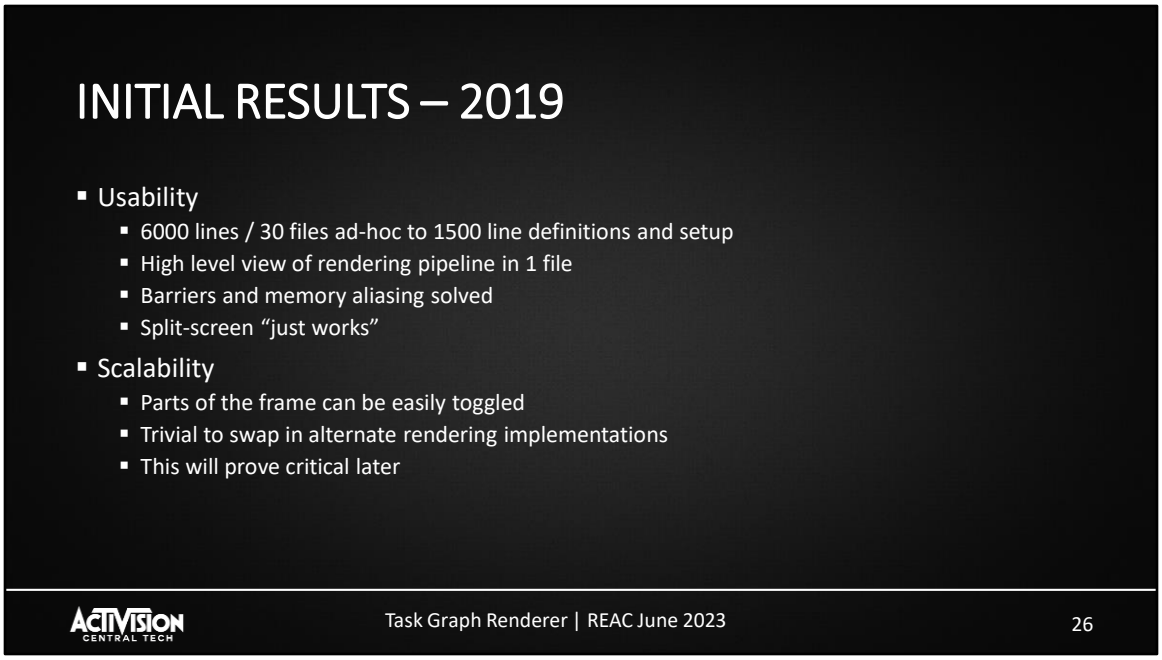
Improved memory usage was the most obvious benefit, with an average saving of 100mb but up to 500mb saved depending on platforms and settings.

On Xbox we were able to improve our ESRAM usage, partially because we now supported tiled allocations and split ESRAM / DDR allocs, which were not supported by the traditional renderer.

Performance results were more modest, with a small but welcome 25% CPU render time improvement.

On the GPU side, performance was initially similar, with reduced barriers giving us only a minor win.

Later however, we were able to do some advanced optimizations, such as scheduling 2 independent GPU pipelines in an interleaved fashion. This resulted in fewer total barriers and improved overlap of GPU workloads.



Besides memory and performance improvements, the most important benefit was graphics engineer productivity by improving usability and scalability.

The traditional renderer was around 6000 lines of glue code scattered around 30 files in an ad-hoc fashion. This didn't include actual drawing and dispatch code but just the setup code in between. This was replaced by around 1500 lines of task graph definitions and the renderer setup function.

This gave a high-level view of the entire rendering pipeline in a single place for the first time. You could read through the frame top to bottom and see how resources were flowing from task to task. Previously, it was difficult to follow the renderer as resources were often stored in globals and accessed randomly.

This also solved the long-standing problem of split-screen. Many graphics techniques would be written without considering split-screen, such as storing a previous frame RT in a single global. Obviously when we are rendering 2 views per frame, the "previous" RT would be from the other players view, causing visual errors.

Task graph was able to understand split-screen at a system level and increase the

number of temporal resources, so every view gets their own unique copy. Graphics engineers no longer needed to worry about it.

In terms of scalability, one problem we used to have was managing the myriad graphics options on PC. In the traditional renderer this resulted in cyclomatic complexity, and it was almost impossible to test every possible path. Often certain combinations of options would result in obscure errors, particularly around memory allocation and barriers.

In the new task graph renderer, the compiler would always generate the correct memory layout and barriers for any setup.

EVOLUTION – 2019 - 2023

- Iterative optimizations
  - Memory
- New platforms / APIs
  - Gen 9 consoles
  - Mobile
- New tools
  - Debugging and introspection
- Multi-threading

I'll now cover a few improvements we've made to task graph since its initial deployment in 2019.

The improvements have been in 4 main areas:

Iterative optimizations, particularly around resource memory allocation have been made constantly.

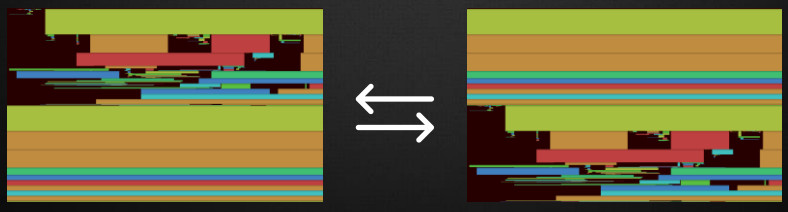
We've also added new platforms and APIs. We initially supported PC DX12 and Gen 8 consoles, we've since added PC Vulkan, Gen 9 and mobile.

We've developed tools to allow real time debugging and introspection of the task graph.

And as we mentioned earlier, we've added full-multithreaded rendering at a system level, replacing the user-level multi-threading we started with.

EVOLUTION – MEMORY OPTIMIZATION

- Temporal resource aliasing
  - Previously unaliased
  - Current temporal layer unused for portion of the frame
  - Place transients into these gaps and rotate each frame



Frame 0, 2, 4, 6, …

Frame 1, 3, 5, 7, …

For iterative optimizations, we've made hundreds of these though one which particularly helped us was temporal resource aliasing.

As mentioned previously we treat temporal resources as a first-class concept in task graph. So along with width, height, format and so on you would also specify the number of temporal layers of a resource.

The idea with temporal resource aliasing is we can place frame transient resources into the unused part of the current temporal layer.

If you think of something like SMAA, you need a temporal resource with 2 layers. We can take advantage of the fact that the current frame's layer is not needed until late in the frame, and place other resources into this gap.

Of course, the current will become the previous on the next frame, so we need to rotate the transients each frame into the current gap.

The 2 graphics here show the heap layout of temporal resources and transients, and how we ping-pong between 2 layouts. If we had resources with 3, 4 or more layers

we’d equally have 3, 4 or more layouts that we cycle between.

This saved us many 10s of megabytes and was especially effective on ultra-high-resolution platforms.

EVOLUTION – NEW PLATFORMS

- Mobile support
  - Metal on iOS including tile shader support
  - Vulkan on Android
  - New scheduling algorithms
    - Attempt to batch render pass compatible tasks
    - Many tasks can share the same render pass
  - Uses same renderer setup function as high-end
    - Parameterized differently
    - Alternate implementations
    - Fragment shader multi-pass vs compute shader single-pass

In terms of new platforms, we've of course added Gen 9 console, but this isn't especially interesting as we already had a high-spec PC version.

More interesting is mobile, as this represents a quite different architecture and vastly reduced performance levels even compared to our minimum spec PC.

One major difference to PC & console is we change the scheduling algorithm to prefer placing render pass compatible tasks consecutively. This allows the renderer to batch many tasks into a single render pass at the API level.

The mobile renderer uses the same renderer setup function as all other platforms, just parametrized differently.

Whole expensive GPU pipelines such as water or volumetrics are disabled entirely, and many other parts of the pipeline have alternate implementations.

This might even be the same algorithm, but implemented differently, for example using fragment shader multi-pass rather than compute shader single-pass.

EVOLUTION – CONSOLE VS MOBILE EXAMPLE

```
TgHandle sceneDepth, linearFloatZ;

if ( options.floatZsinglePass )
{
      cResolveFloatZ_AllMipsCS( builder, linearFloatZ, sceneDepth );
}
else
{
      cResolveFloatZ_SingleMipFS( builder, linearFloatZ.Mip( 0 ), sceneDepth );
      cResolveFloatZ_DownsampleFS( builder, linearFloatZ.Mip( 1 ), linearFloatZ.Mip( 0 ) );
      cResolveFloatZ_DownsampleFS( builder, linearFloatZ.Mip( 2 ), linearFloatZ.Mip( 1 ) );
      cResolveFloatZ_DownsampleFS( builder, linearFloatZ.Mip( 3 ), linearFloatZ.Mip( 2 ) );
}
```

- Single pass compute shader console optimal
- Multi pass fragment shader mobile optimal

A quick illustration of that here.

On console and PC, we spent a lot of time combining multi-pass techniques into single pass. An example is generating 4 mips of a depth downsample in a single dispatch using groupshared memory or wave intrinsics.

Though this approach is functional on mobile, the performance can be extremely poor, so we optionally use a multi-pass fragment shader version as shown in the code snippet above.

With this unified renderer approach, we can progressively enable additional features for mobile as the hardware becomes more capable.

EVOLUTION – DEBUGGING TOOLS

- Debug UI
  - Using Dear ImGui
  - In-game and remote viewable
  - Heap layouts
  - Event viewer
    - Capture draws / dispatches
  - Resource viewer
    - Capture resources before / after tasks
    - Helped track some nasty bugs
  - Dependency graph viewer
    - Follow a resource through the graph

The last topic I'll cover briefly before handing over is the debugging tools we've integrated in-game.

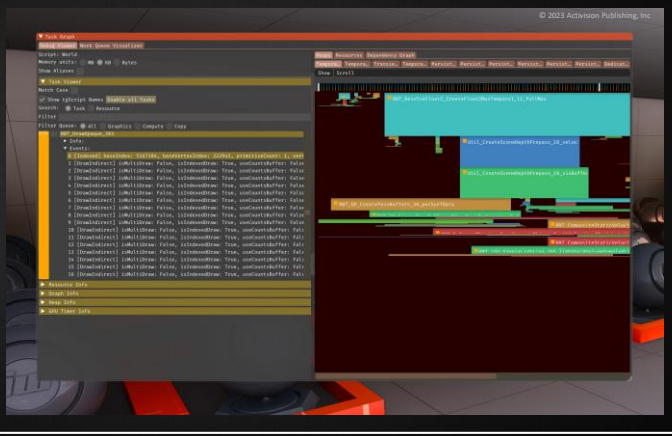
One of the advantages of a system like this is the huge amount of meta-data the system provides, as well as simple hook points in the renderer so we can create rich debugging tools.

The main tool we have here is the Task Graph Debug UI. This is a tool created using Dear ImGui, roughly based on RenderDoc and PIX but critically allowing for real time inspection while the game is running.

The tool supports viewing heap memory layouts, viewing draws and dispatches, and capturing render targets and buffers during rendering. We also have a visualization of the dependency graph we generated during the compilation process.

EVOLUTION – DEBUGGING TOOLS

- Task viewer
  - Tasks with events
  - Draws
  - Dispatches
- Heap tab
  - Resource allocs

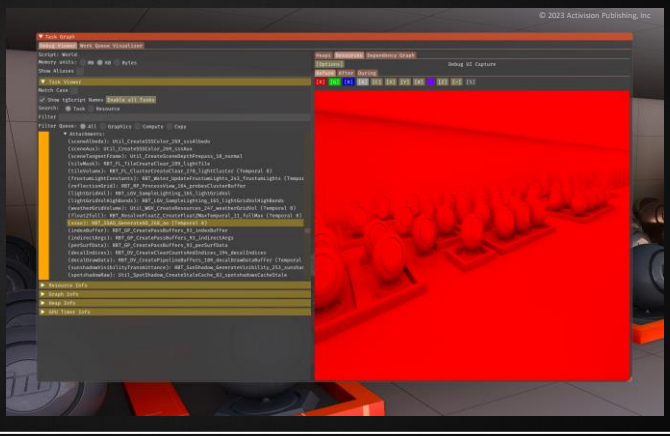


This screen here shows the task viewer panel on the left, in this case we have a DrawOpaque task selected, and we are inspecting the draw calls contained in that task.

On the right-hand side, we have a heap layout, showing how the task graph has allocated memory for transient, temporal and persistent resources.

EVOLUTION – DEBUGGING TOOLS

- Resource tab
  - Capture resources
  - Render targets
  - Buffers

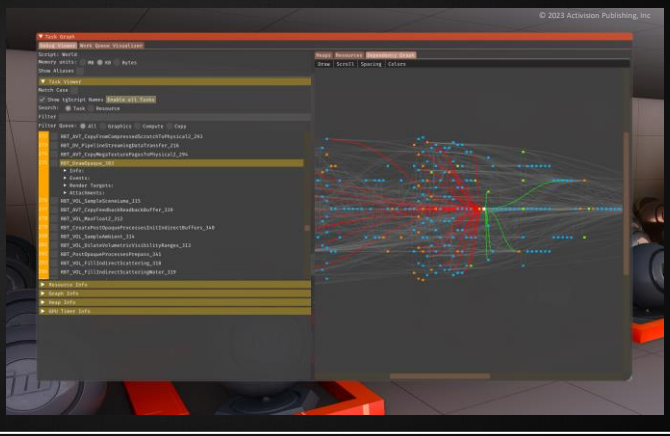


This screen shot shows how we're inspecting the DrawOpaque task, and now we're capturing one of the resources we passed to that task, an SSAO render target.

We can capture these resources before and after the task, and even per-draw call or dispatch.

EVOLUTION – DEBUGGING TOOLS

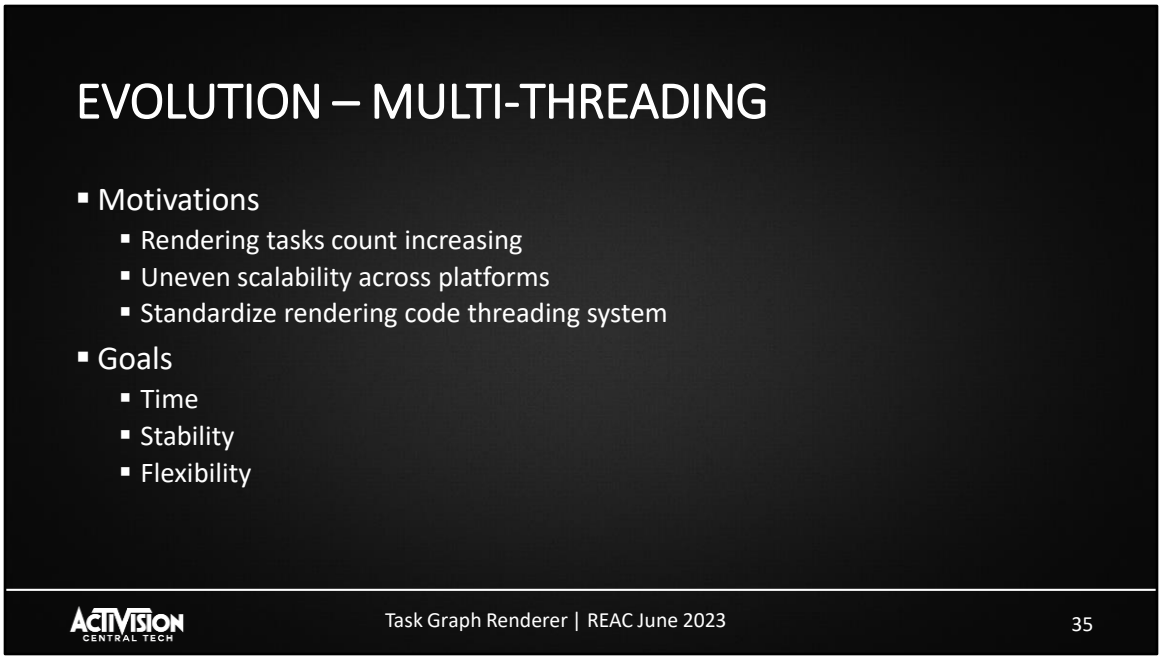
- Dependency graph
  - Input/outputs
  - Follow resources



Lastly, we have a dependency graph viewer. Again, we have a DrawOpaque task selected, and we can see the dependencies from previous tasks, as well as future tasks that are themselves dependent.

This allows us to follow resources through the graph and make more sense of how the frame was scheduled.

And with that, I'll hand over to Francois who will discuss the multi-threading improvements we have made.



Thanks Charlie,

The task graph really simplified how we manage and code rendering tasks. After using it for a couple years, the complexity of our render graph fairly grew.

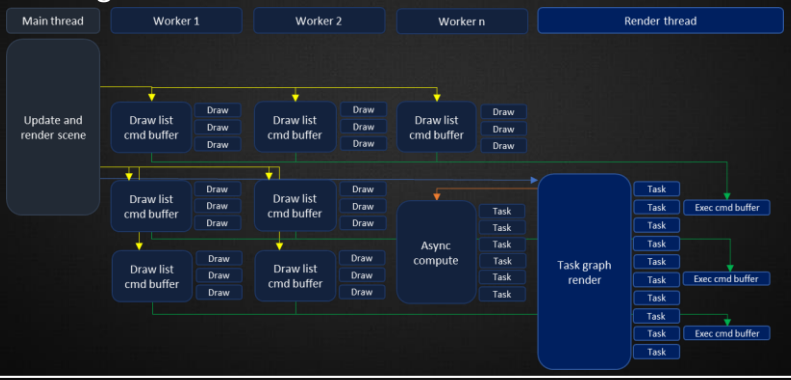
- At the time we started reworking our threading approach, around 350 tasks could be executed to render a frame and this number was constantly going up as new features were coming in. It was foreseeable that it could become a bottleneck and needed some way to scale.
- We already had some workloads threaded like draw lists and async compute command buffers, but these didn't always translated well to all platforms. For instance all async compute tasks wouldn't be threaded if there was no hardware support for it.
- As new hardware was being released, it was getting harder to feed GPUs fast enough just because CPU single threaded performance hardly improves nowadays.
- It would also be great if we could find a unified approach to standardize how all rendering code was threaded.

We had several goals for this to be a success.

- We ship games with this renderer every year so we needed a system that could be rolled out progressively. The amount of code to rework was likely to be too large to do within a single title but still wanted to see some benefits.
- We wanted the system to be robust. Threading issues can be a pain to debug, so revisiting core rendering systems and making sure they were thread-safe would be key.
- Finally, as we target multiple platforms, some of which have vastly different hardware, we needed a solution that would be adaptable and configurable.

EVOLUTION – MULTI-THREADING

- Threads background



This was presented and largely discussed by Michael Vance two years ago, but to give a bit of context this is a simplified view of how our rendering code was threaded.

- Main thread (commonly called frontend)
  - Is responsible of updating player, do scene traversal, coarse culling and prepare most rendering data required by the render thread. Most of these tasks are assisted by worker threads but main thread acts as an orchestrator.
  - As we gather the data we need to render our draw lists, the main thread will fire jobs to workers so they prepare command buffers. These are represented by yellow arrows. The draw lists effectively contains all the information we need to issue draw calls for our prepass, shadows, opaque pass and so on.
  - Worker threads then come into action, by building those command buffers and forward the results to the render thread for submission which is highlighted by green arrows.
  - Our main thread will finalize its frame while the workers are processing draw lists and make sure all data is ready for the render thread before unblocking it which is depicted by the blue arrow.

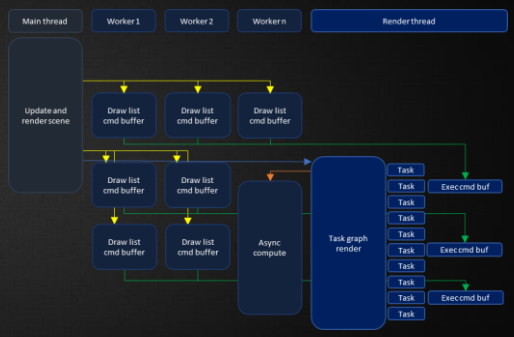
- Render thread (commonly called backend)
    - Is responsible of submitting graphics queue command buffers.
    - For the most part, it is only iterating over all of task graph tasks. Some of those tasks might require to execute draw list command buffers which were prepared by workers.
        - It can sometime happens that a worker did not finished its work by the time the render thread needs the command buffers. In that case, the render thread can take over the remaining work and finalize the draw list command buffer.
            - This mechanism helps ensure the backend is always feeding the GPU fast enough and it also frees the worker to do additional work without having the render thread stall. We will get back to this concept later in the presentation.
    - Ther render thread also fires an async compute job when support is available (showed by the orange arrow), async command buffer submission is handled directly by the worker processing the job.
        - when there is no async compute support, the render thread has the responsibility to execute these tasks.

- That should be a good summary on how rendering flows through the different threads.

- We should note that our worker thread count varies depending on platforms and processors
    - Their responsibilities are to assist critical threads in their work, often in a fork-join model although there are some exceptions
    - They also don't have clear a ownership, they can assist main thread, render thread or other critical threads

EVOLUTION – MULTI-THREADING

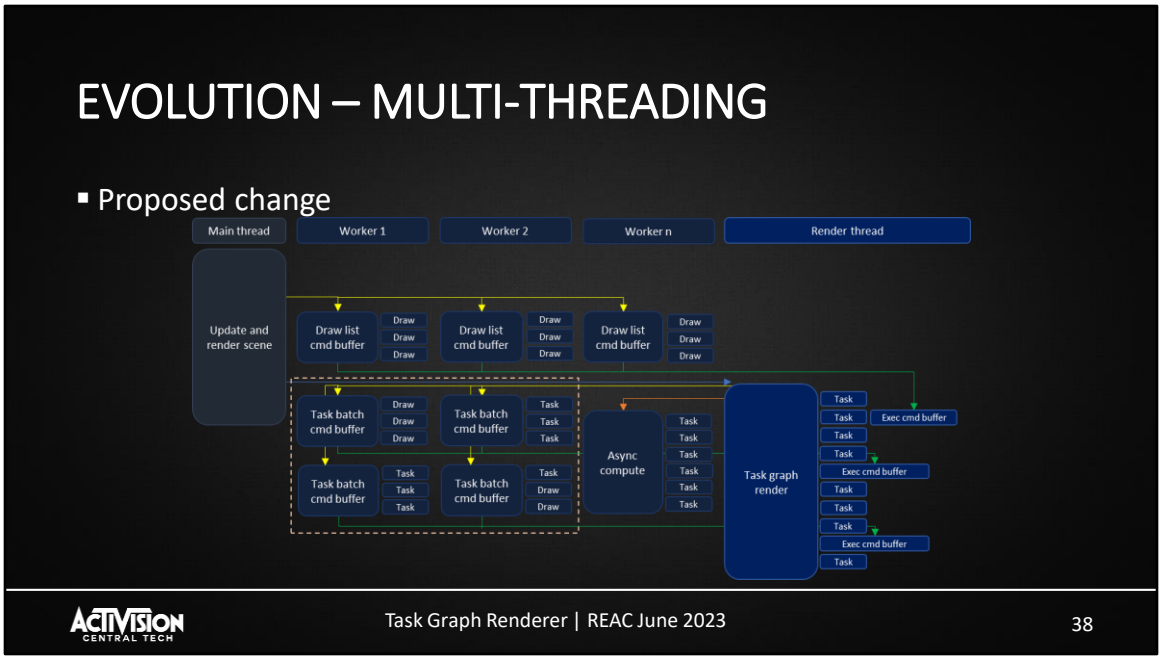
- Shortcomings
  - System mostly scaled with draw lists
  - Draw workers could compete with other frontend workloads
  - Draw workers not always working on backend critical tasks
  - System hardly compatible with TBDR HW



There were a few shortcomings with this system.

- It mostly scaled with draw lists
  - Draw lists were still a major part of the rendering workload, but as said earlier, task graph's tasks count was increasing and becoming a new bottleneck
  - They also weren't aware of the full task graph context, so small command buffers could be needed before or after these tasks to do resource state transitions or flush caches.
  - We could also have draw lists containing few draw calls which could spin a new command buffer that had a minimal amount of work.
- Draw list jobs could sometime compete with other frontend workloads
  - The graph shown here is a very simplified example, in reality there are many other tasks needs to run on workers, some more critical than other. When workers would work on draw lists too soon, they could potentially prevent the frontend from finishing early, delaying next frame work.
- Similar to the frontend workloads issue, the backend might need the command buffers which workers are working on a lot later down the frame
  - So workers may not be assisting the render thread where it would

need help, for instance if there was other heavy tasks where render thread could have benefited to submit work to GPU faster.
    - It wasn't a common cause of gpu bubbles but on lower spec hardware it could still occur.
- Finally, to have more predictable granularity, longer draw lists like prepass and opaque pass would rely on splitting their workloads across multiple jobs generating several command buffers. This approach wasn't viable on Tiled Based Deferred Rendering hardware as it would incur a performance penalty.

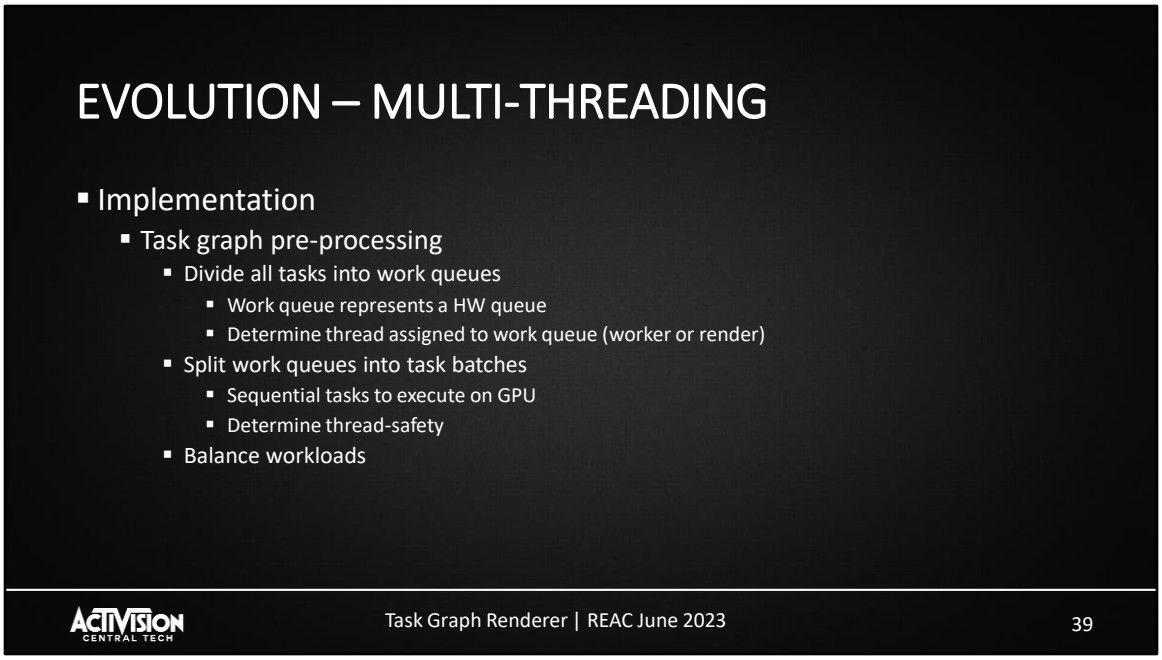


- The proposed change was to move back responsibility to the render thread to issue worker jobs that would allow to execute task graph tasks as well.
    - Render thread would issue these jobs in the GPU's expected order to make sure workers would work on upcoming needed render commands.
    - This would also prevent jobs from hogging on precious worker thread cpu time at inopportune moments.
- Task batch job responsibility would be to record command buffers for a batch of tasks.
    - Their command buffers would be forwarded to the render thread for gpu submission just like draw lists command buffers (as shown with green arrows)
    - The draw lists could still be processed like they did before but they could also be processed as a part of a task batch.
        - This would also be useful to do A/B testing or profiling and roll out the new system progressively.
- The granularity of a batch would be controlled by backend, scaling on hardware needs, worker count or hardware architecture.

So in the end, the render thread would simply be offloading part of its work to

workers and they would progress in tandem to render a frame.
Such a change meant that we would need to go over all rendering code and annotate tasks which are thread-safe or modify them so they are.

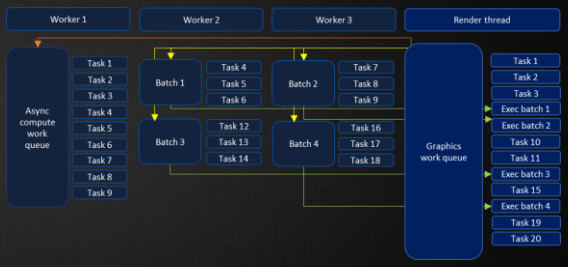
Prior to that, tasks could just assume they would be executed in a specific order on the cpu timeline caused by dependencies between them. This wouldn't the case anymore so data dependencies needed to be reworked and moved out of render thread timeline.
That would make tasks thread-safe and they could afterward be processed in any order on the cpu timeline.



- To implement this, we added a pre-processing step which would build work queues.
    - A work queue simply represents a hardware queue and the list of tasks it needs to execute.
    - We then assign the thread responsible of overseeing the workqueue tasks and execution. For instance our graphics queue is owned by our render thread, but async compute queue by a worker thread. These are the threads that will actually be responsible of submitting command buffers for those queues.
- Work queues are then split further into batches of tasks, these batches are tagged as thread-safe or not depending on the tasks they contains. This determines whether or not a worker thread can assist the one owning the workqueue.
- A simple balancing pass is then done to make sure we are issuing similar workloads between batches to worker threads.

EVOLUTION – MULTI-THREADING

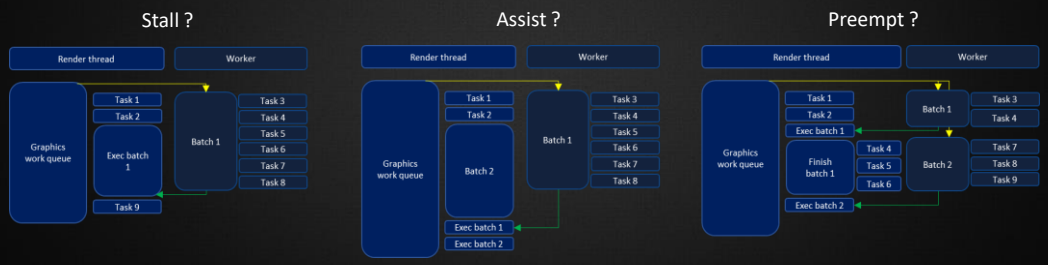
- Implementation
  - Task graph runtime
    - Work queues are processed
      - Graphics queue runs on render thread
      - Async compute queue runs on worker
    - Workers process task batches
    - Render thread
      - Execute non-thread safe tasks
        - or draw lists
      - Submits completed batches



- Once we sorted out all of our batches and the thread they are safe to run on, the render thread issues all jobs to workers. The jobs are issued in the order the gpu expects them and therefore ensure we are working on what is needed next.
- While the workers are executing tasks, the render thread will execute those which aren't deemed thread safe ensuring that data dependencies between tasks can still be fulfilled.
    - As we removed those dependencies we would annotate the task to mark it as thread safe and it could be processed and batched on a worker.
- Finally as the workers finish the jobs, the queue owner submits the command buffers prepared by workers once the time is right.

EVOLUTION – MULTI-THREADING

- This all sounds good in theory but what if
  - Workers are too busy to process batches in time ?



- So this all sounds good in theory but what happens if workers are too busy to process batches in time ?
- This could certainly happen if a batch takes too long to finish, or if workers are already too busy to even be able to start processing a batch.
- Even more predictably, the render thread can submit command buffers pretty fast, likely faster than workers can process task batches. This means the render thread will likely catch up on workers at some point, so what should we do ?
  - We could stall render thread and wait on the worker to finish ?
    - But this would lose precious cpu cycles we certainly don't want to waste, many platforms are always searching for more free cycles.
  - Maybe we could have the render thread assist workers ?
    - We would at least make forward progress, but still, there is always the question of how long can we afford to do additional work before the GPU ends up bubbling up ?
  - A trickier approach would be to have the render thread pre-empt a worker thread and take over the remaining work ?
    - This certainly has many implications on when we can pre-empt, like right in the middle of a task or just between tasks ?

- It certainly would increase overall code complexity depending on when we want pre-emption to occur

EVOLUTION – MULTI-THREADING

- This all sounds good in theory but what if
  - Workers run out of rendering resources ?
    - Constant / Vertices / Indices buffers
    - Command buffer
    - Descriptors

- There is also another case that can happen, our workers can run out of rendering resources.
  - By rendering resources I mean all the data we need to allocate and setup to issue our rendering calls.
  - It can be constant buffers, descriptors, the command buffer itself or any other kind of data which is filled as we need to issue draw calls or dispatches.

- To understand why this is a concern, we should refer once again to Michael Vance's prior presentation of our renderer.

EVOLUTION – MULTI-THREADING

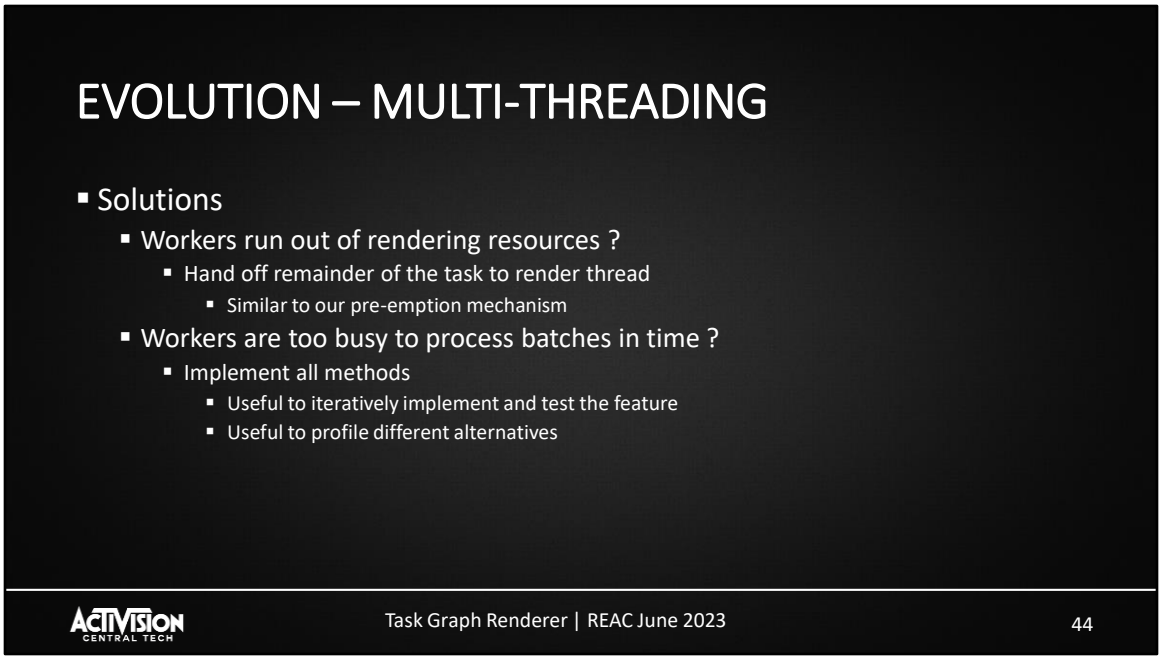
- How are our rendering resources allocated ?

- Ring allocator
  - Ownership solely on render thread
  - Segments are periodically fenced
  - Effectively allows to issue draws endlessly
- Linear allocator
  - Lock-free, fast allocator
  - Pages allocated from worker threads
  - Can be exhausted
    - Handoff remaining work to render thread

- Our rendering resources allocations are static and will not be reallocated if we run out of them, we have a finite amount of memory to work with.
- We have two different allocators for that kind of data.
  - The ring allocator is solely used by the render thread, as it needs to allocate data, it will store it in a ring buffer. As we submit command buffers, used chunks are fenced with the gpu so we can reuse segments that are finished with.
    - This allocator allows the render thread to submit as many draw calls as needed, the worst that can happen is that it will submit outstanding commands and wait for the gpu to be finished with a chunk of memory so it can progress.
    - This is something we make sure and hope will never happens on a live game, but at least it is resilient so we don't crash.
  - The linear allocator is used by worker threads, each thread will allocate pages within that segment as they need memory.
    - That allocator doesn't have endless memory and there is a risk of eventually running out of it when we encounter an heavy scene with a ton of draw calls. So workers could need more space than there actually is.

- Still we do not want to crash out of memory and need to submit all of our draw calls. So when this occurs, draw lists workers can hand off their remaining work to the render thread. Since the render thread uses a ring buffer we have a guarantee that, as we submit commands, we will always recover more space once the gpu is done with it.

- So what does that means for our use case ?

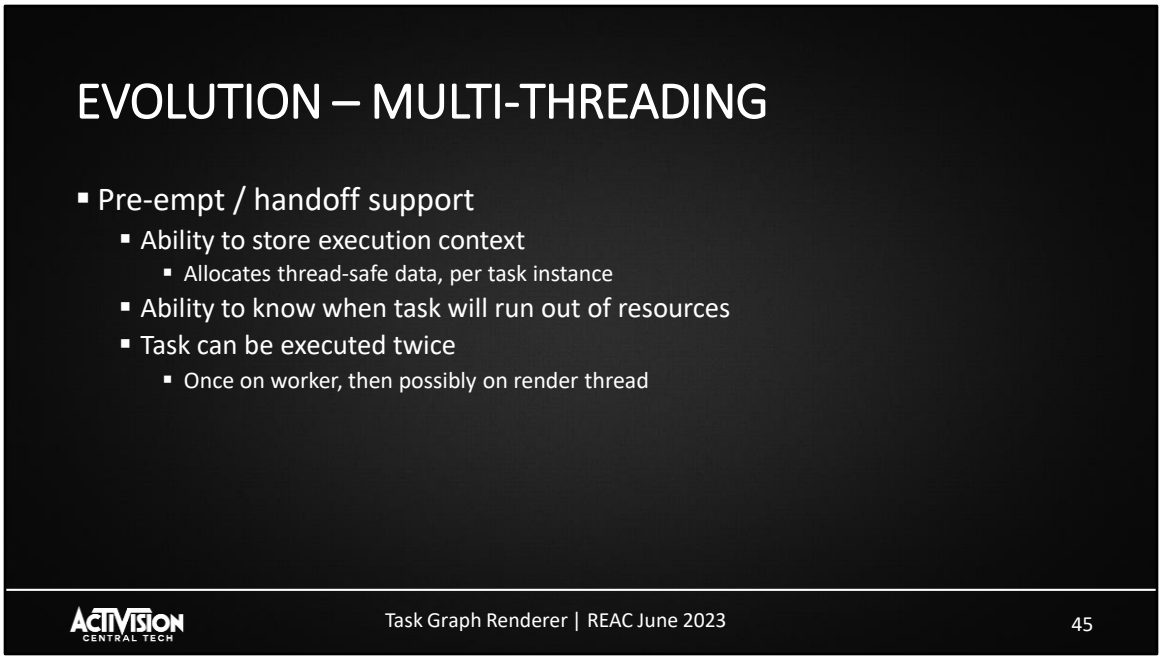


- This means that in the event we run out of resources, we need to have the ability to recover from it. Our existing way to do so is to hand off worker thread's work back to render thread. There is no need to trash the work which is already done and start over. If we can store a task context we can resume our work where we were at, on a different thread.
- We already had such a system in place with draw lists and was proven to work well. We adapted it so not only draw lists can hand off their work but any task as well.
- This would effectively become the tool required so we can pre-empt worker threads which cannot complete their batch in time for the render thread to submit them.

- In the end we did implement all solutions to test them out and it has been valuable to fine tune system.
    - Naturally, having the render thread able to stall waiting for a worker wasn't really used in the end.
    - Our render thread then had the ability to assist workers, but could also to pre-empt them and make sure we never starve the gpu.
        - At the same time this became our defence mechanism when we ran

out of resources.

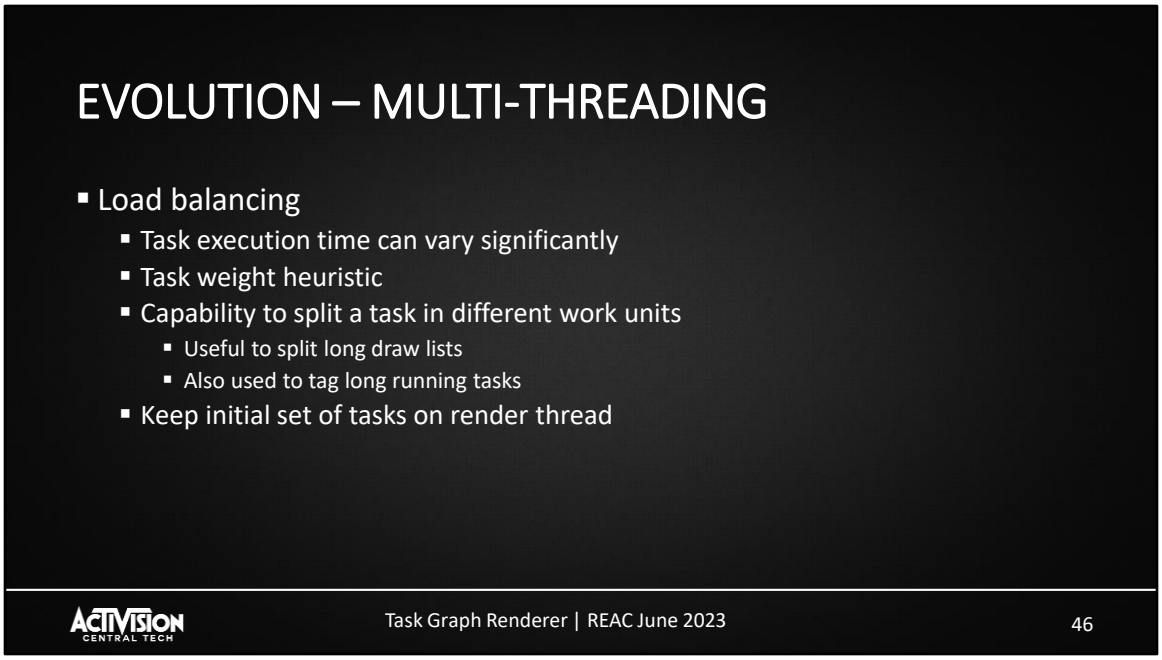
- We could also have the render thread assist on one batch, then pre-empt a worker if it still wasn't finished.



- To support preemption or handoff
  - We needed the ability to store a task execution context
    - We added necessary utilities so coders can add metadata in task definition to easily allocate and access thread-safe data to store execution context.
    - This data is allocated on a per-instance basis as a task can be re-used multiple times within a frame graph.
    - User can then with the help of utility functions retrieve that data to store and load context.
  - We also needed the ability to know when task will run out of resources
    - We didn't wanted to increase complexity every time we issued a draw call or a dispatch because we knew this would become cumbersome for engineers to handle these cases and wanted this to be automatic most of the time.
    - As the task graph executes tasks, it bears the responsibility to ensure we have enough space to progress forward. We reserve enough space in our linear allocator for a good amount of draw calls or dispatches. It really depends on how much constant buffer, command buffer and descriptors space are required for those commands.

- If the task really needs to issue significantly more commands than what task graph reserved upfront, then it becomes the task responsibility to poll and ensure it has enough space to progress forward. In the event it cannot, it must store its execution context and stop processing. The render thread will be responsible of executing this task a second time with ring allocators to finalize the tasks.
- We added defensive mechanisms to ensure we trapped cases where it wasn't handled.

- In our renderer it wasn't a common occurrence that we needed to manually poll resource availability, so we got away with this and it allowed us to progressively roll out the system.
- This certainly is something that will be improved in the future but it implied larger changes that we didn't had the time to do.



- Tasks processing long draw list can take milliseconds to complete whereas other small tasks finish in a few microseconds. Batching some tasks together could leave us with largely different workloads between worker jobs.

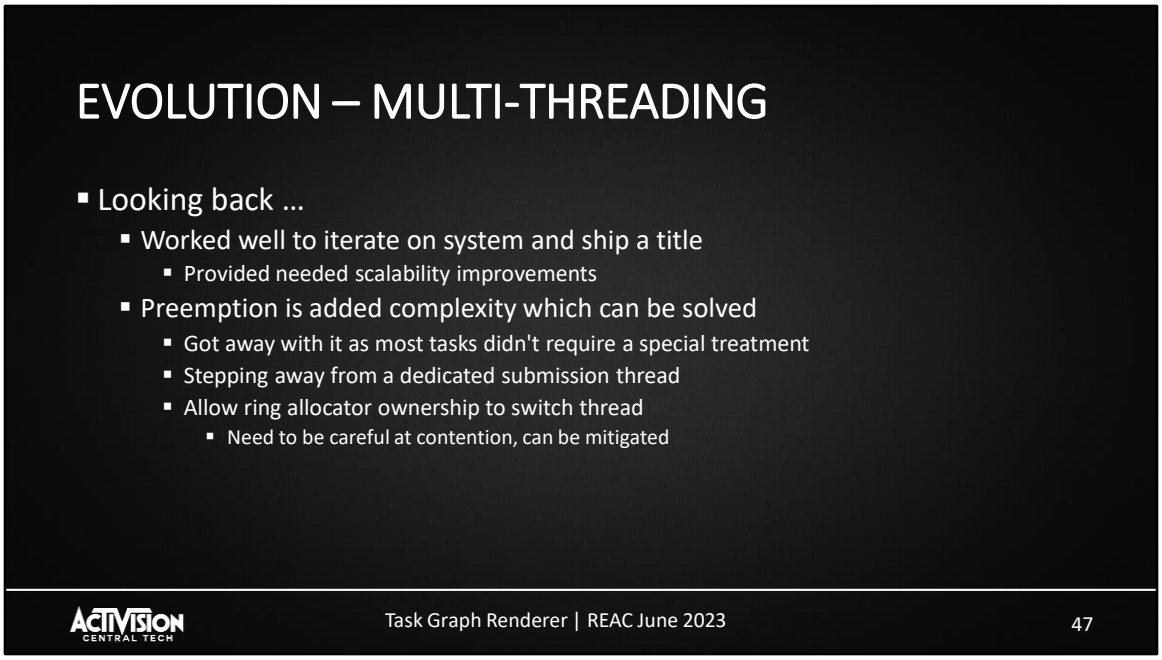
- A simple yet efficient way to address this was to add support for task weight. This is fairly opaque to engineers and mostly automated. As we pre-process tasks to be executed in a batch, given a small set of rules, we determine if task is lightweight or heavy.
- We split task batches when their weight exceed certain threshold and we balance them out. This way we can get a more predictable runtime for our batches and it also helped not creating very small command buffers which could add overhead.
- We have lots of ideas to improve heuristics and improve this, but simple things worked out well.
    - Still, it would be nice to keep some history of tasks execution time and help even out things further.

- We also added a way to split long standing tasks into different jobs so they can be threaded further.
    - Tasks would determine at creation time the maximum amount of work units

they can have. Then at runtime during task setup phase they could specify how many they need for a given frame, this amount would tell how many worker jobs to issue in order to further parallelize this task.

- Tasks processing draw lists for instance would benefit from this to behave similarly to the prior draw list jobs we had which could split a draw list in multiple segments. As this is easily configurable, TBD platforms can simply issue one job so their render pass can be executed in one go.
- This allows to re-balance tasks weight and provide a better workload split between workers.

- We also kept the first set of tasks on the render thread. It helped feeding the gpu more quickly and also gave some head start to workers so render thread can play catch up game later preventing it to try to pre-empt workers too often.



- Looking back on this evolution while it worked great to ship previous title, there is a lot room for improvements and we are still actively working on this.
  - It required a mind shift on how we coded more complex tasks as they cannot be written in a fire and forget manner. We got away with this because not many tasks required this hand-off special treatment.
    - It can be error prone even with validation and debug functionalities to make sure everything was properly handled.
  - We are working on alleviating this issue by changing some mechanisms.
    - Ring buffer resources can be shared with workers allowing them not to run out of resources. If workers can submit their own command buffers, they will be able to reuse ring buffer segments.
      - This could add a new source of contention between workers but we can improve our linear allocator to prevent this from happening.
    - By doing so, tasks wouldn't need to be re-executed by the render thread a second time and we wouldn't need to store the execution context anymore. This would greatly simplify the process and remove the need to handle resource exhaustion manually.
    - It would also help us take the step away from needing a dedicated

render thread and have renderer more job driven.

- That being said, the core functionality worked great for our needs. It ensured our workers were always processing meaningful tasks and that render thread constantly progressed forward. It was a good stepping stone for our needs and it opened new doors to have our renderer scale better in the future.

FUTURE WORK

- C++ based DSL
  - Generally successful
  - Still a desire for a data-driven format
  - Tighter integration with materials
    - Inject available bindings
    - Validate shaders against tasks at compile time
  - Share pipeline snippets
    - Tools or prototypes
- Render Pipeline Shaders SDK
  - Presented at GDC 2022 [9] and recently open sourced [10]
  - HLSL based render graph specification
  - Providing a more unified environment for engineers
  - Offline analysis

Aside from further improvements to multi-threading, the biggest change we are keen to pursue is to the front-end DSL.

Generally, we feel like the C++ based approach worked well, but we'd like to move to more of a data-driven format we can leverage earlier in our pipeline.

This would allow a couple of improvements such as tighter integration with materials and PSOs. We could compile shaders in context of a node's definition, allowing injection of available bindings and improved validation.

We could also share snippets of rendering pipeline with tools or minimal apps for prototyping.

One option here would be to leverage a new render graph SDK. This was presented last year at GDC and open sourced.

The HLSL based DSL is attractive and could help provide a more unified programming environment for graphics engineers.

We could potentially also leverage offline analysis tools, which could give useful feedback about memory requirements or performance without needing to launch into the game.

REFERENCES

[1] Michal Drobot, “Rendering of Call of Duty: Infinite Warfare”, https://research.activision.com/publications/archives/rendering-of-call-of-dutyinfinite-warfare

[2] Yuriy O’Donnell, “FrameGraph: Extensible Rendering Architecture in Frostbite”, GDC 2017

[3] Tiago Rodrigues, “Moving to DirectX 12: Lessons Learned”, GDC 2017

[4] Michal Drobot, “Geometry Rendering Pipeline Architecture”, REAC 2021

[5] Michal Drobot, “Software-based Variable Rate Shading in Call of Duty: Modern Warfare”, https://research.activision.com/publications/2020-09/software-based-variable-rate-shading-in-call-of-duty--modern-war

[6] Michal Olejnik, Pawel Kozlowski, “Raytraced Shadows in Call of Duty: Modern Warfare”, Digital Dragons 2020

[7] Rulon Raymond, “Lima Oscar Delta!: Scaling Content in Call of Duty: Modern Warfare”, GDC 2020

[8] Michael Vance, “Rendering Engine Architecture at Activision”, REAC 2021

[9] Zhuo Chen, “Render Pipeline Shaders”, GDC 2022,

[10] Zhuo Chen, Florian Herick, Noah Cabral, “Introducing AMD Render Pipeline Shaders SDK”, https://gpuopen.com/learn/rps_1_0/

That’s it, thanks for listening!