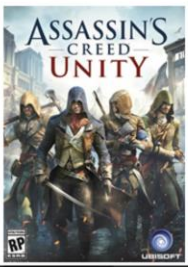

GDC
Moving to DirectX 12:
Lessons Learned
Tiago Rodrigues
3D Programmer, Ubisoft Montreal
GAME DEVELOPERS CONFERENCE® | FEB 27–MAR 3, 2017 | EXPO: MAR 1–3, 2017 #GDC17
UBM

Talk goals:

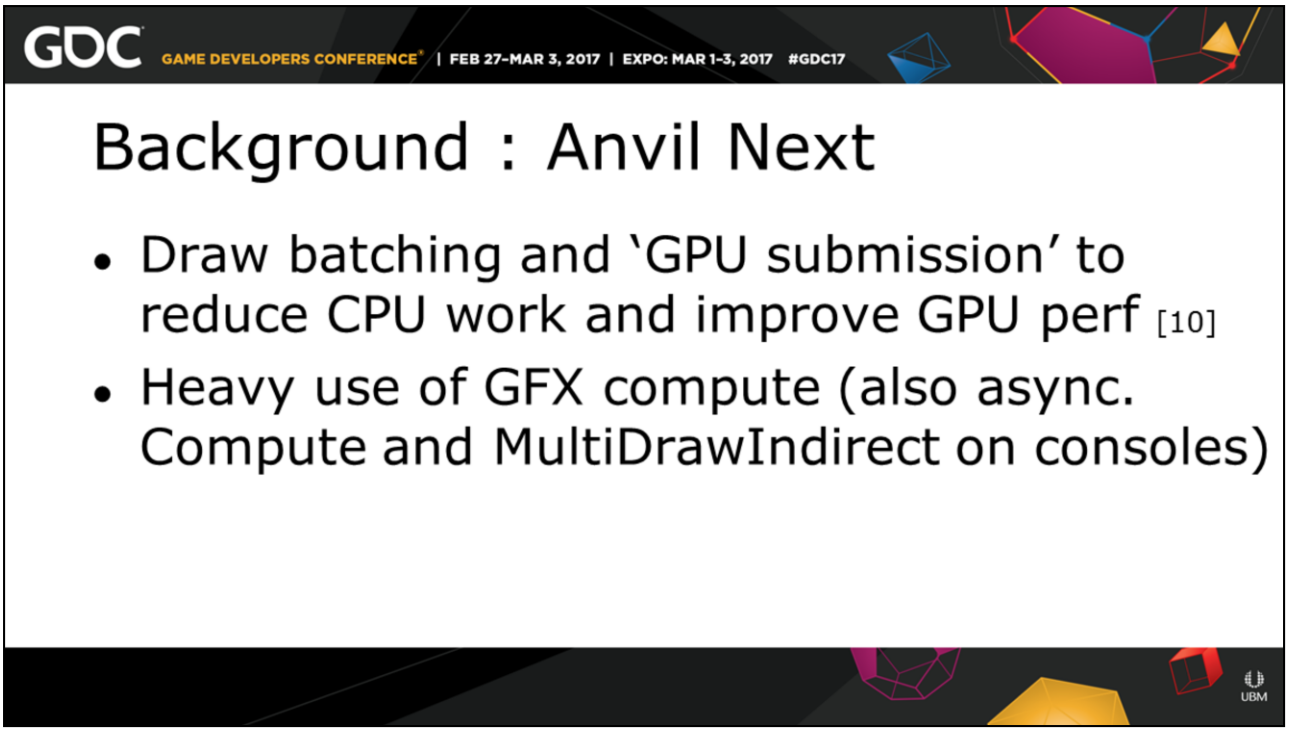
- Many other talks have already focused on more direct API usage advice [1]-[8]
- I'll focus on some higher level strategies to implement some of this advice

Background : Anvil Next

- Anvil (in-house engine), has shipped 8 AC tiles in the last 10 years. Anvil Next was a major upgrade we developed for ACU.

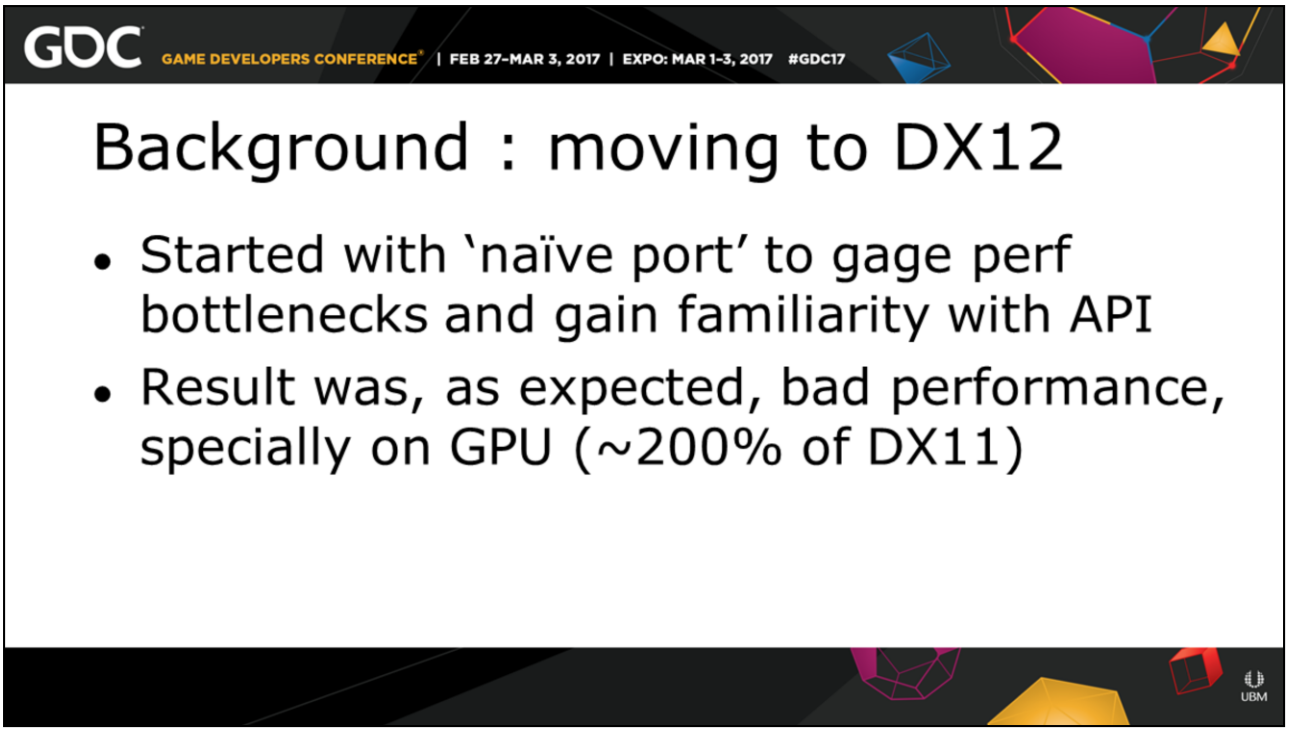


- For ACU we did a significant upgrade, which we call “Anvil Next”. You can see examples of the most recent ACs that have shipped with this version.



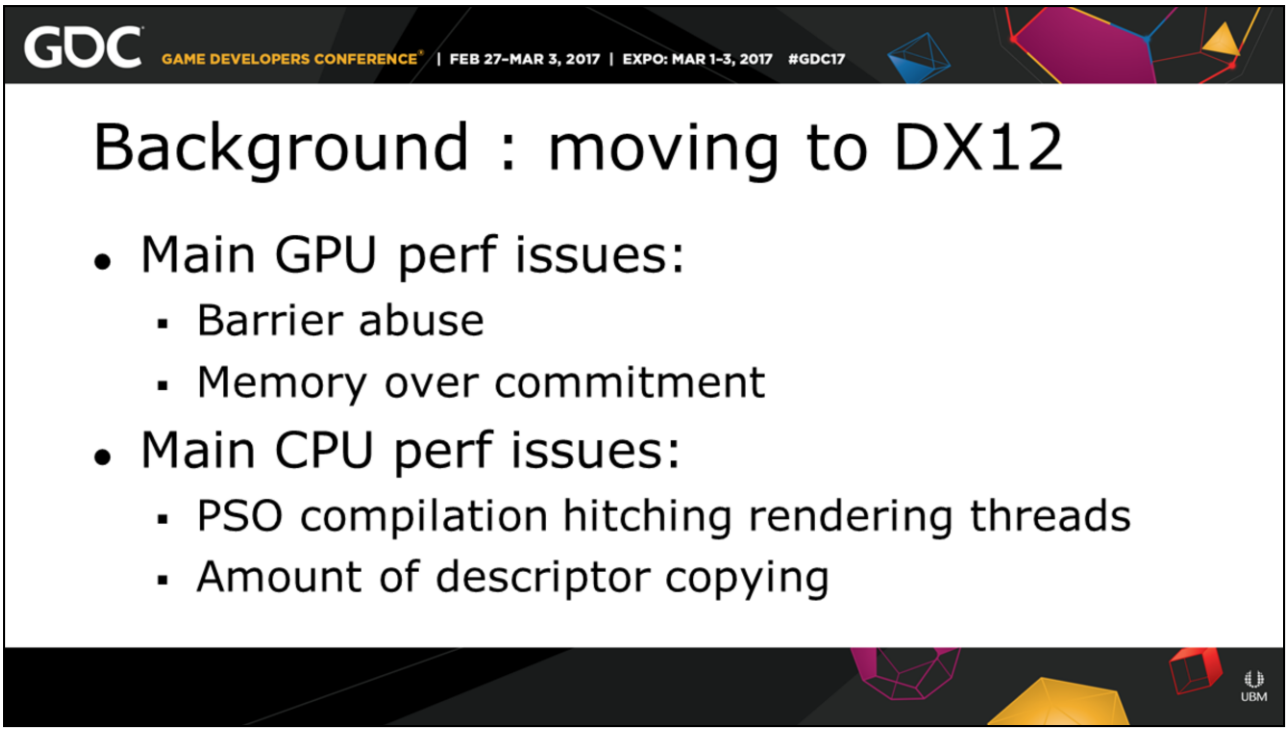
- We did a lot of work for AC:Unity related to draw call batching and 'GPU submission' (using the GPU to cull draws/instances/triangle clusters/triangles), you can get some more details on this topic from Ulrich's GDC15 presentation on the subject.

- We also had console specific optimizations on the rendering pipeline, since certain features weren't available on PC (like using async. compute for the GPU culling dispatches and multidrawindirect), DX12 enables the use of these features on PC.



- Started with a very basic port, where the DX12 API details were hidden behind the old renderer interfaces

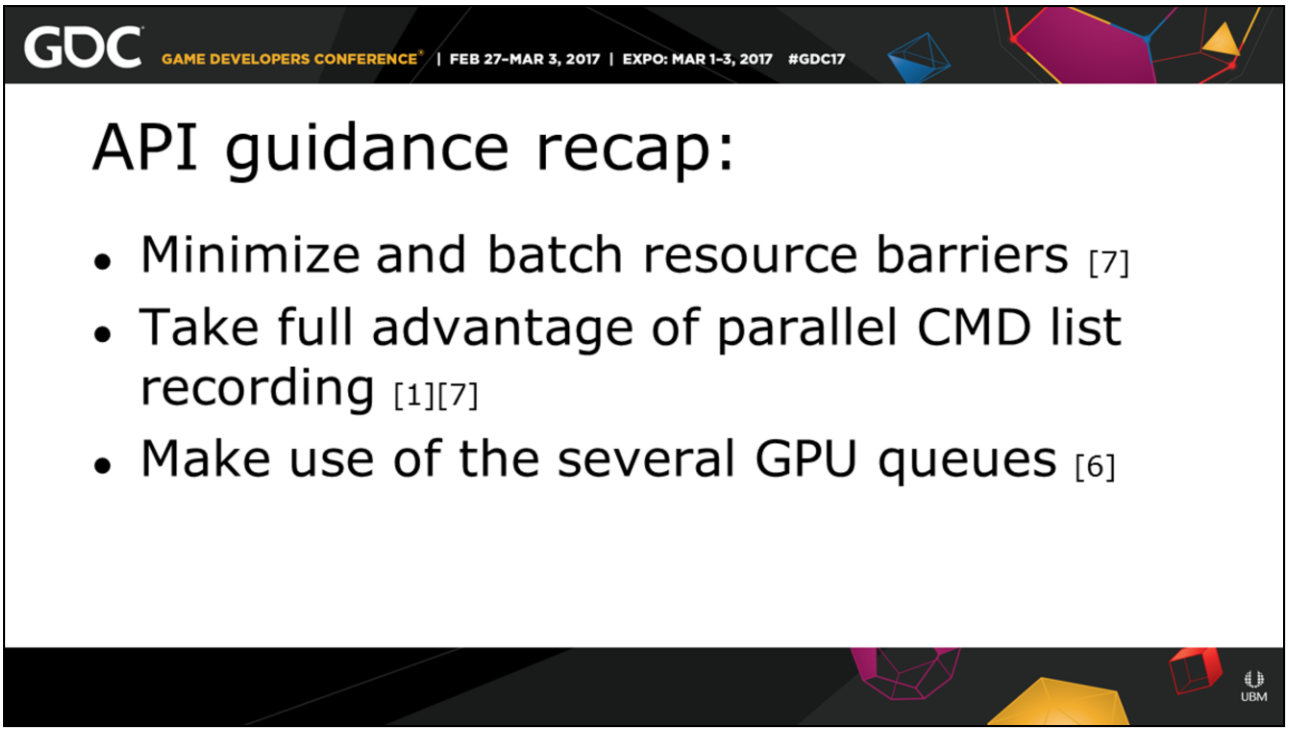
- As expected we had extreme bad performance (GPU time was ~200% of DX11). It was simply too hard to implement some of common API usage advice, mainly due to very low level and narrow view of resource states in the hardware abstraction layer



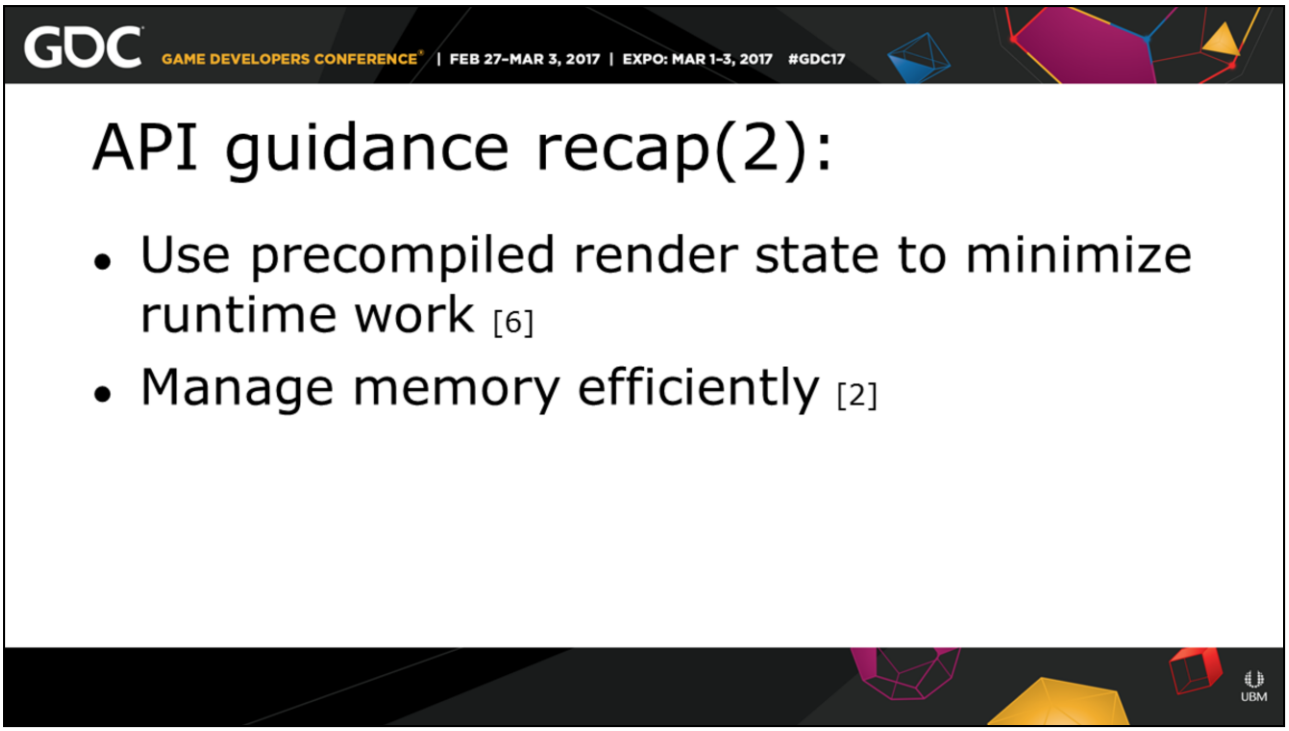
- Barrier abuse:
  - We did a lot of individual barrier calls, even when batching between commits
  - I was quite hard to manage barriers with multithreaded CMD list recording, without spreading forced initial states everywhere (leading us to do a lot of intermediate unnecessary transitions)
- Regarding Memory:
  - We were significantly overcommitted (due to the lack of aliasing, tiled resources/mip streaming)
  - We also had issues with CMD list management and reuse
- PSO and descriptor management:
  - Hitching in rendering threads due to PSO compilations
  - Due to using a common root signature to handle all use cases, we had a lot of descriptor copying

Background : moving to DX12

- Started planning renderer redesign based on:
  - experience from initial port
  - other teams at Ubisoft
  - advice from various talks on DX12



- Minimizing and batching resource barriers is very important:
  - Otherwise you'll end up hurting GPU performance by serializing your work or flushing caching unnecessarily.
  - DX11 drivers have had many years to optimize this under the hood. DX12 still provides you with a bit of help here: if you batch barriers in a single call, it will produce the minimal set of barrier actions for all transitions in that call
- DX12 also provides fully functional, low overhead parallel CMD list recording, using it is essential to obtain good CPU perf.
  - There are some caveats to have in mind, like not having too many tiny CMD lists, or calling ExecuteCommandlLists too often, else you'll hurt CPU perf
- We also now have access to copy engines and async. compute queues, if you want to match the DX11 driver you'll probably have to use them
  - Also, for console developers, feature set wise, we are now pretty close to feature parity on PC, which is great from an engine design perspective



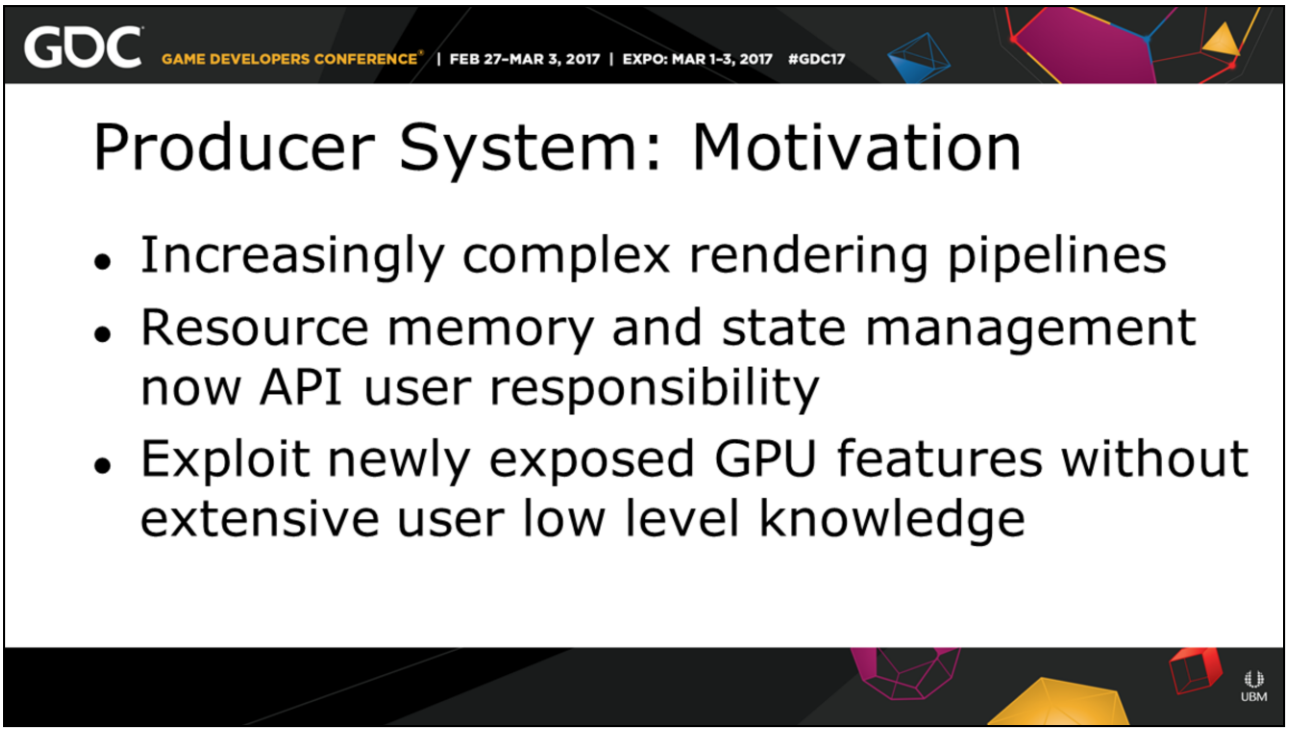
- Using precompiled render state effectively to minimize runtime work is also quite important for CPU perf.
  - The API has a heavy focus on precompiled state blobs, allowing us to do some expensive work once, cache it, and then have minimal overhead at bind time

- Memory management is a large topic, I'll only touch on some aspects of it that relate to the systems I'll introduce
  - The API now enables users much control over how memory is allocated and managed (enabling for example implementing memory aliasing)

Systems

- Producer System
- Shader Input Groups
- Pipeline State management

Producer System



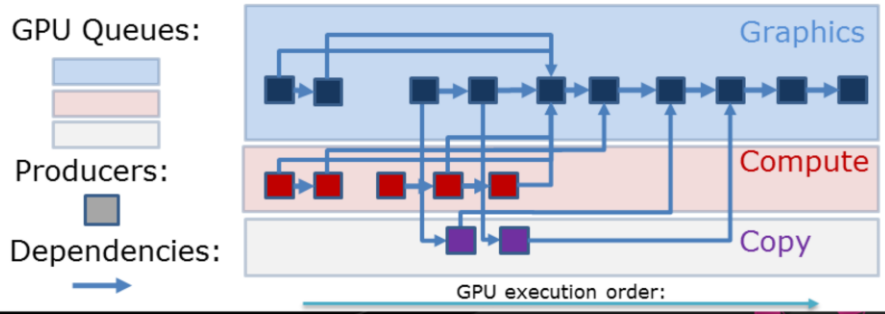
- At the end of the development of AC:Unity, we started to really feel the limitations of our rendering pass architecture as the complexity of the rendering pipeline continued to grow.

- At the same time, we felt that the new graphic APIs coming in the near future, would really allow a different level of control on all platforms that should be addressed on a architectural and cross-platform level to get the full range of benefits.

- This new level of control does mean that managing of resource memory and state is now an API user concern. We want to efficiently drive the API, while at the same time avoiding imposing a large burden on the engine users.

Complex Rendering Pipelines

- Producers are GPU resource writers/readers
- Resource dependencies determine execution order on GPU

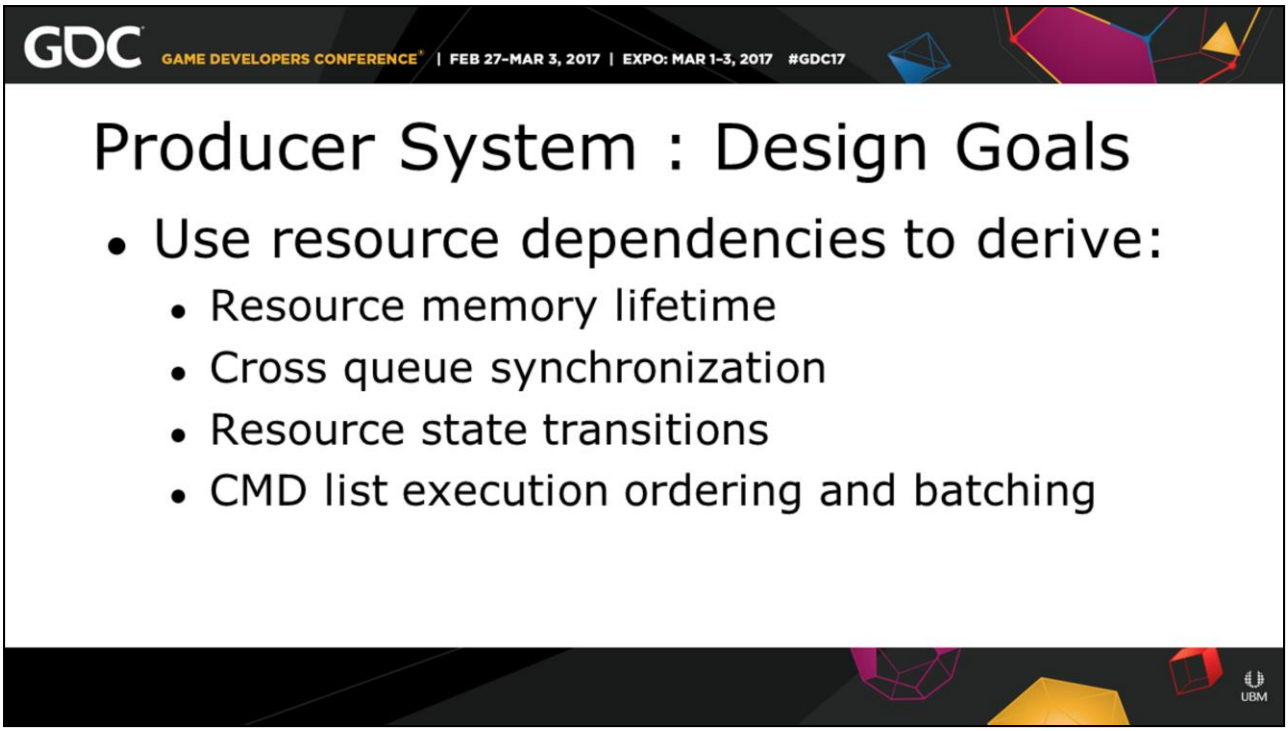


- Example diagram of a rendering pipeline:
  - You can see a number Producers (squares in the diagram), these are essentially writers and readers of GPU resources
  - Then you have dependencies between these that define the GPU execution order
  - you also have several GPU queues, which add another layer of complexity in terms of synchronization, resource lifetime management, etc.

- Just looking at the static configuration of this graph it's already quite complex, however you'll have to add another layer of permutations to support different sets of enabled features (for example, for performance scaling on lower end machines)

- Reconfiguring the pipeline to the several use cases, has many knock-on effects which make manual/explicit scheduling somewhat impractical.

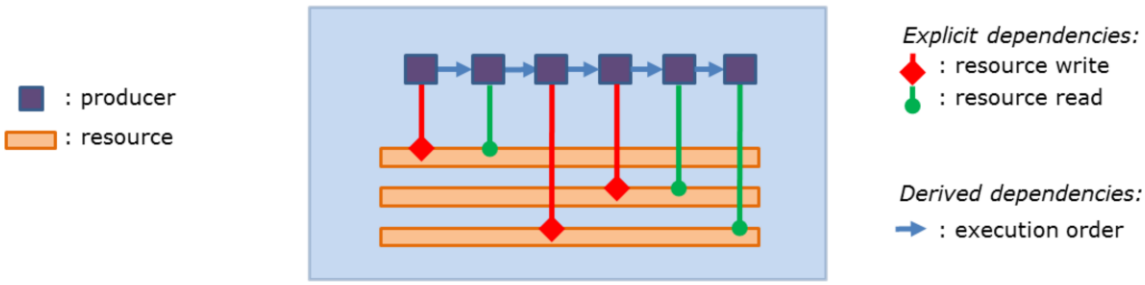
- We needed some level of automation that would not sacrifice performance, but allow us to capture enough high level information to then drive the low level API efficiently



- One of the key design aspects of this system was to make resource dependencies explicit

Resource Dependency Tracking

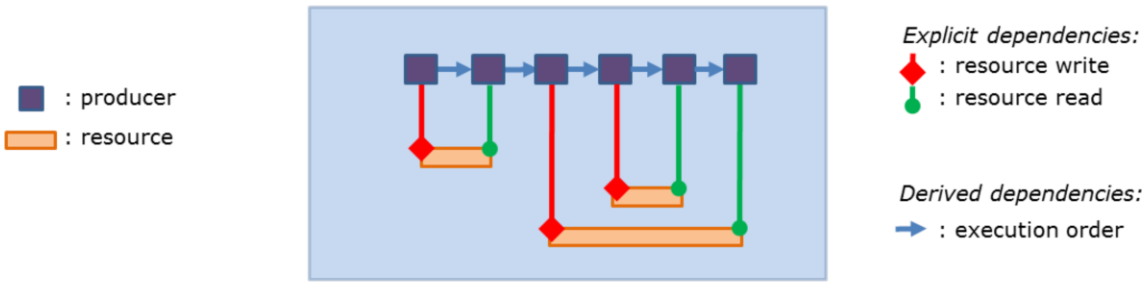
- Explicit resource dependencies specified for each producer



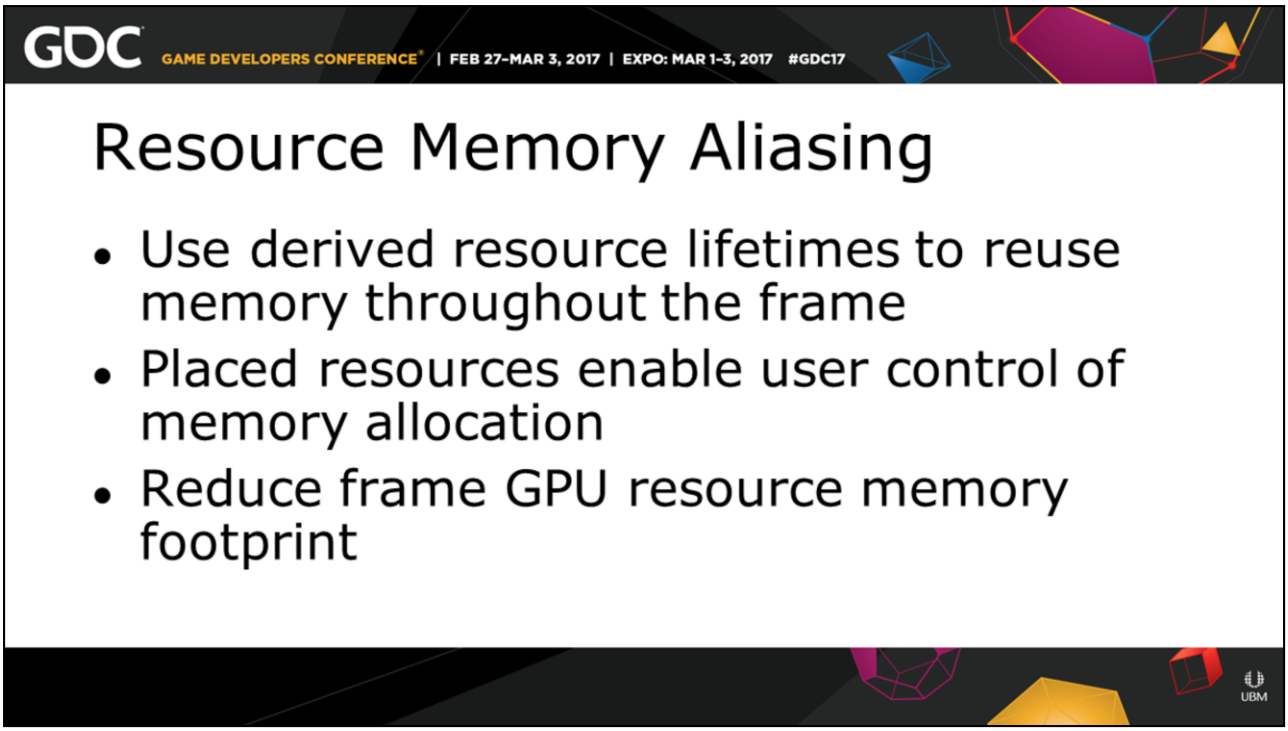
- In this example you can see a set of producers and how they specify their relationship to resources when executing on the GPU

Resource Dependency Tracking

- Explicit dependencies allow us to automatically determine GPU resource lifetime



- Based on these dependencies we can then automatically derive the lifetime of each GPU resource:

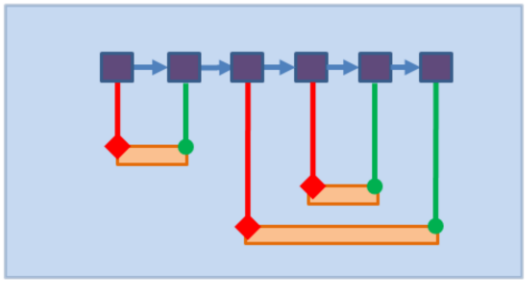


- Using placed resources and the derived resource lifetimes we can reuse memory throughout the frame and reduce GPU memory footprint

Resource Memory Aliasing

- Use resource lifetime to derive memory reuse

: producer
: resource

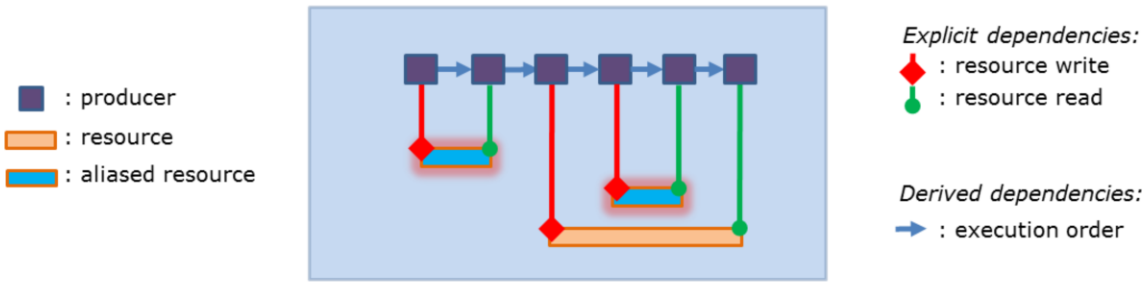


Explicit dependencies:
: resource write
: resource read

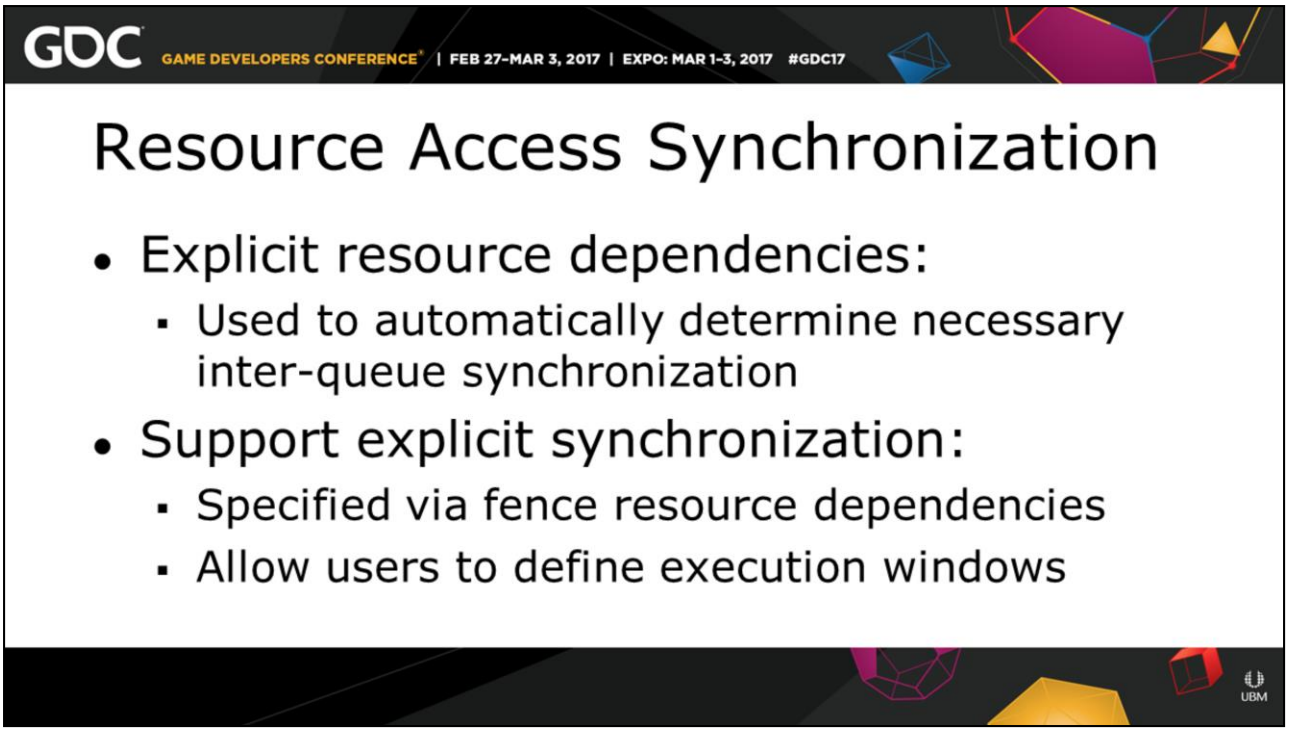
Derived dependencies:
: execution order

Resource Memory Aliasing

- Use resource lifetime to derive memory reuse



- Memory aliasing is automatically derived based on resource lifetime and following cross-queue synch points. In the diagram, you can see an example of two GPU resources that share the same memory since their usage does not overlap in the GPU execution lifetime



- One aspect the producer system automates is: Resource Access synchronization

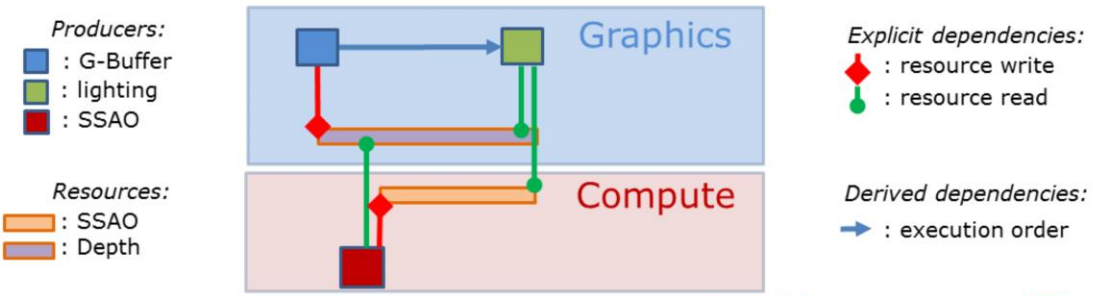
- Every producer specifies a GPU queue that it wants to execute it's command lists on: graphics/async. compute/copy (which can change at runtime for debugging or other configuration purposes).

- Based on the explicitly specified resource dependencies for each producer, we can then derive necessary cross-queue synchronization (fencing) to guarantee the correct execution order of CMDs lists on the GPU

- We still support explicit synchronization via fence resources so users have control over execution windows (for example to better match GPU workloads running in different queues)

Resource Access Synchronization

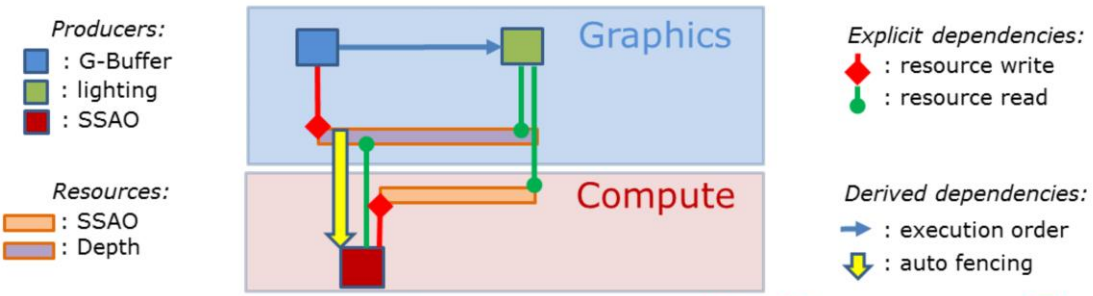
- SSAO buffer produced in compute, consumed in GFX queue



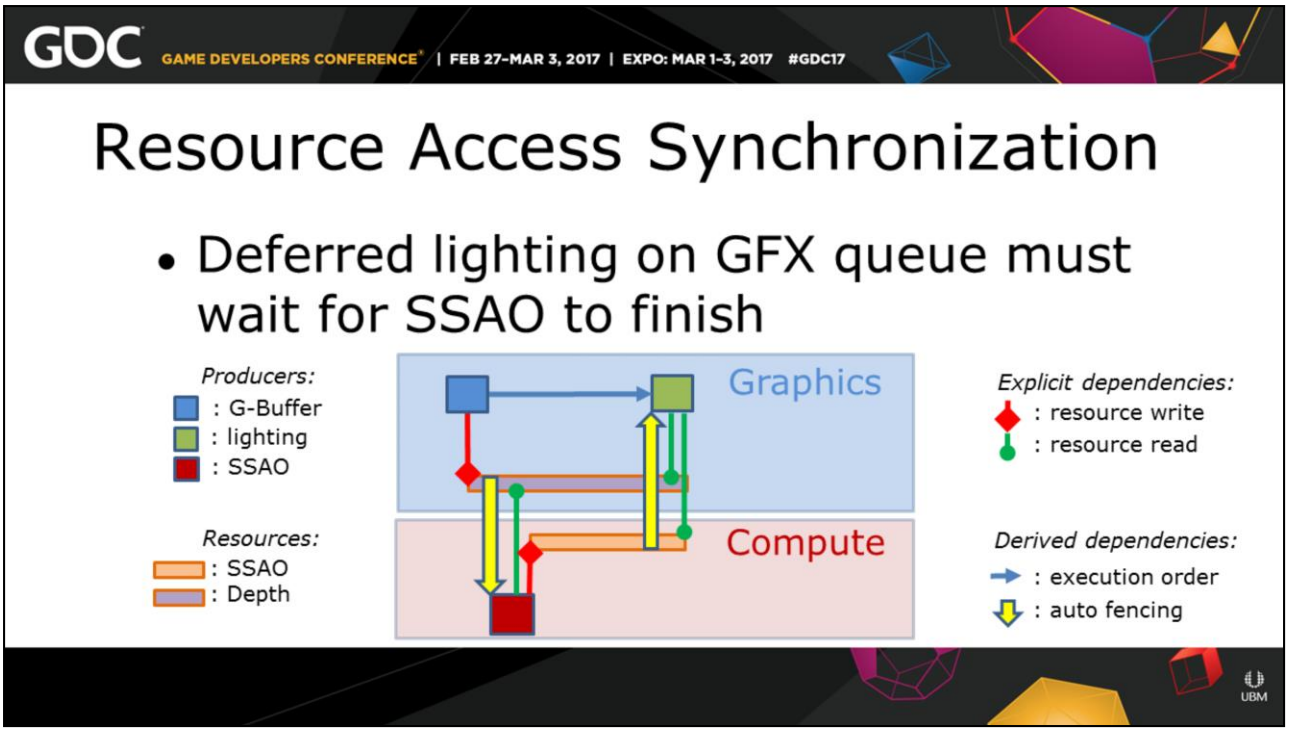
- Here is an example of a simple necessary cross queue synchronization:
  - You can see the SSAO producer (in red) running in the compute queue, it depends on the G-Buffer depth, produced in the GFX by the GBuffer producer (in blue)

Resource Access Synchronization

- SSAO on compute queue must wait for G-Buffer rendering to finish

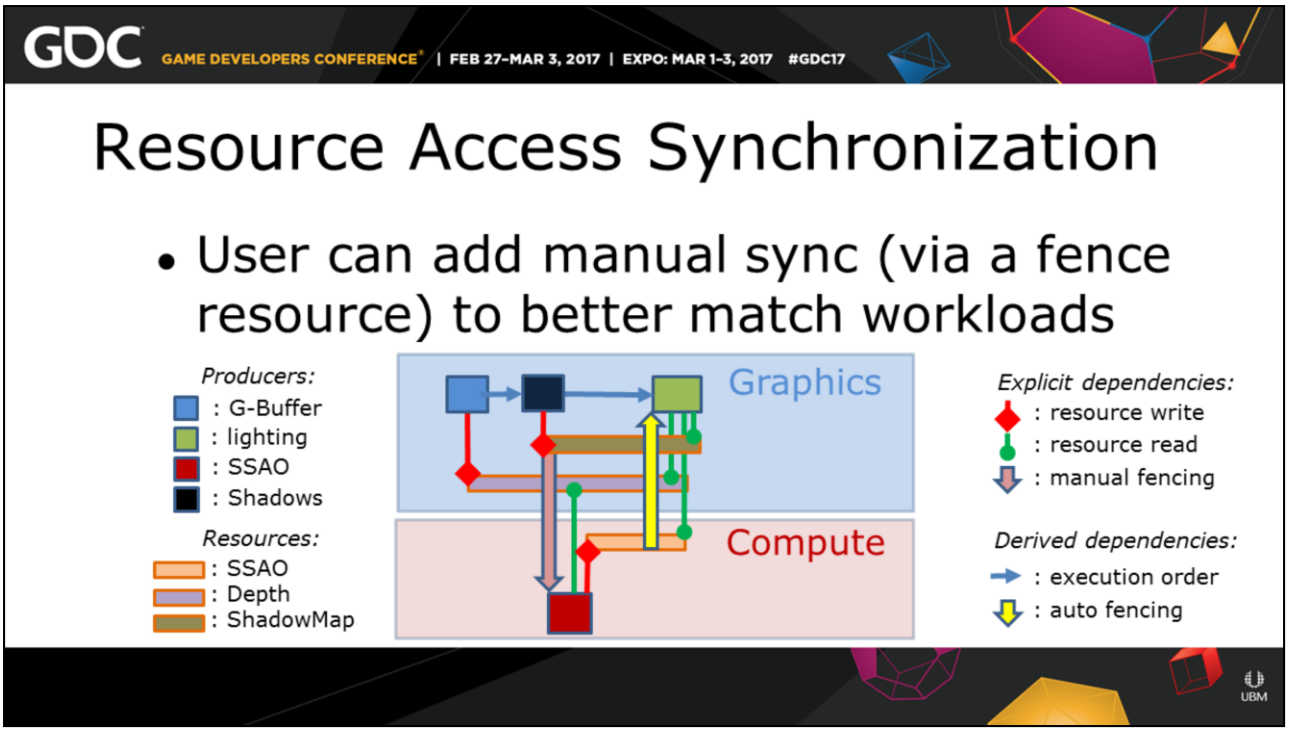


- The first necessary synchronization is before the SSAO producer as it cannot start before the GBuffer pass has finished writing to the scene depth.

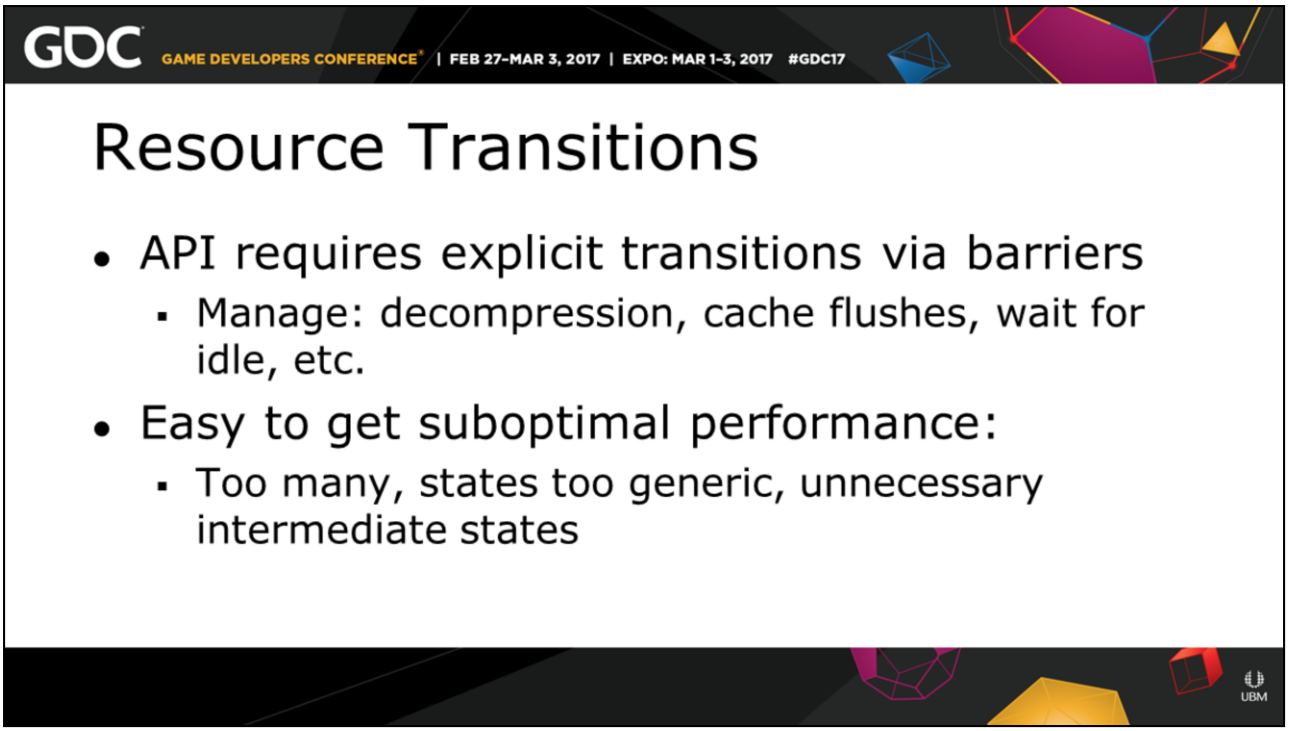


- Conversely, we also need fencing between the SSAO producer in the compute queue and the first consumer of the SSAO mask, which in this case is the deferred lighting producer in the GFX queue (which you can see in green in the diagram)

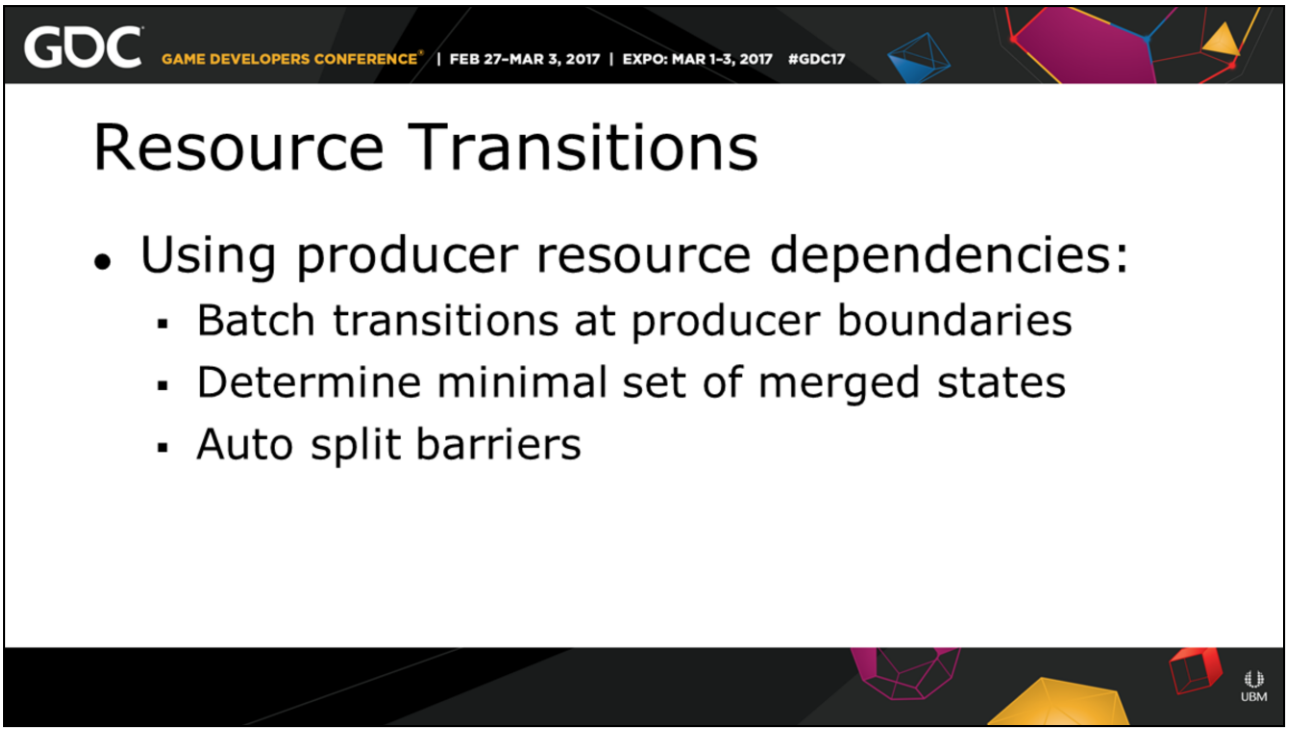
- This example is quite simple, in a real frame schedule there could be many more producers that could, for example, write to the scene depth after the G-Buffer. If one of these producers where to move, and synchronization was done manually by the user, it could introduce timing dependent glitches that are sometimes hard to spot and debug



- To better match GPU workloads on different queues the user might want to explicitly define a window in which the async. compute workload will run
  - For example, over the vertex heavy shadow map producer (in black in the diagram)
  - A manual sync is simply done by depending on a fence resource (fences are a producer resource just like any other)
  - Because explicit fencing is a resource, we can easily cope with alternative configurations (for example load time/lazy updated static shadow maps on some levels, where the system automatically adds automatic syncing when the shadow producer isn't scheduled every frame)
- Since the remaining fencing was automatically derived, it can take into account the user fencing and eliminate any extra synching due to the GFX workload execution order (as you can see in the diagram by the absence of automatic sync between the GBuffer producer and the SSAO)
- One alternative to manual fencing for this purpose, would be to tag producers at a high level to indicate if they are Vertex heavy/Bandwidth heavy/ALU heavy and then let the producer scheduling attempt to match them automatically



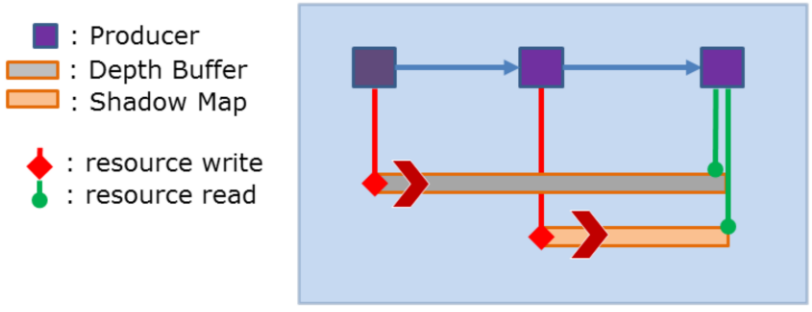
- Resource Transitions are another aspect our system automatic manages
- In the API these are specified via barriers, which manage operations like resource decompression, cache flushes, waits for idle, etc.
- As I mentioned before, it's easy to get suboptimal GPU performance by having a very narrow view of resources, which can lead to doing too many individual barrier calls, transitioning to generic or unnecessary intermediate states, etc.



- Using resource dependencies:
  - Batch barriers at producer boundaries to minimize the work that is actually performed by the driver
  - We avoid doing unneeded intermediate state changes, because knowing the resource dependency graph, we can know upfront the best set of state(s) to transition to
  - Having knowledge of when you finish producing a resource and when it's actually needed to be used for the first time, allows us to split the barriers and potentially hide some internal driver work

Resource State Transitions

- Barriers at producer boundaries

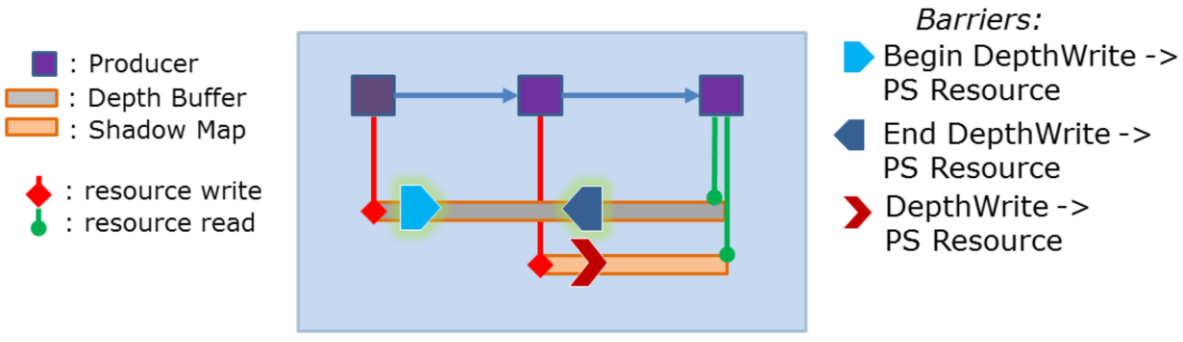


- In this example you can see a number of producers (in purple), and a couple of "depth write" to "pixel shader resource" barriers (the red arrow heads), which can trigger a depth decompression

- As I mentioned before, we issue barriers at the end producers, and since we know what is the next required state in the graph, we can make the transition to the next required state early

Resource State Transitions

- Auto split barriers



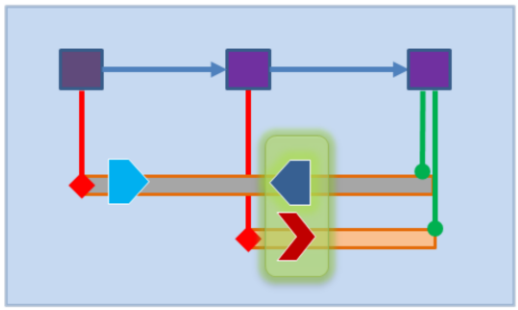
- Instead of an instant barrier, we can use split barriers which define a window where we guarantee we won't access that resource, to hint to the driver it can 'hide' some potentially expensive internal operations in that window

Resource State Transitions

- Group barriers

: Producer
: Depth Buffer
: Shadow Map

: resource write
: resource read



Barriers:

Begin DepthWrite -> PS Resource

End DepthWrite -> PS Resource

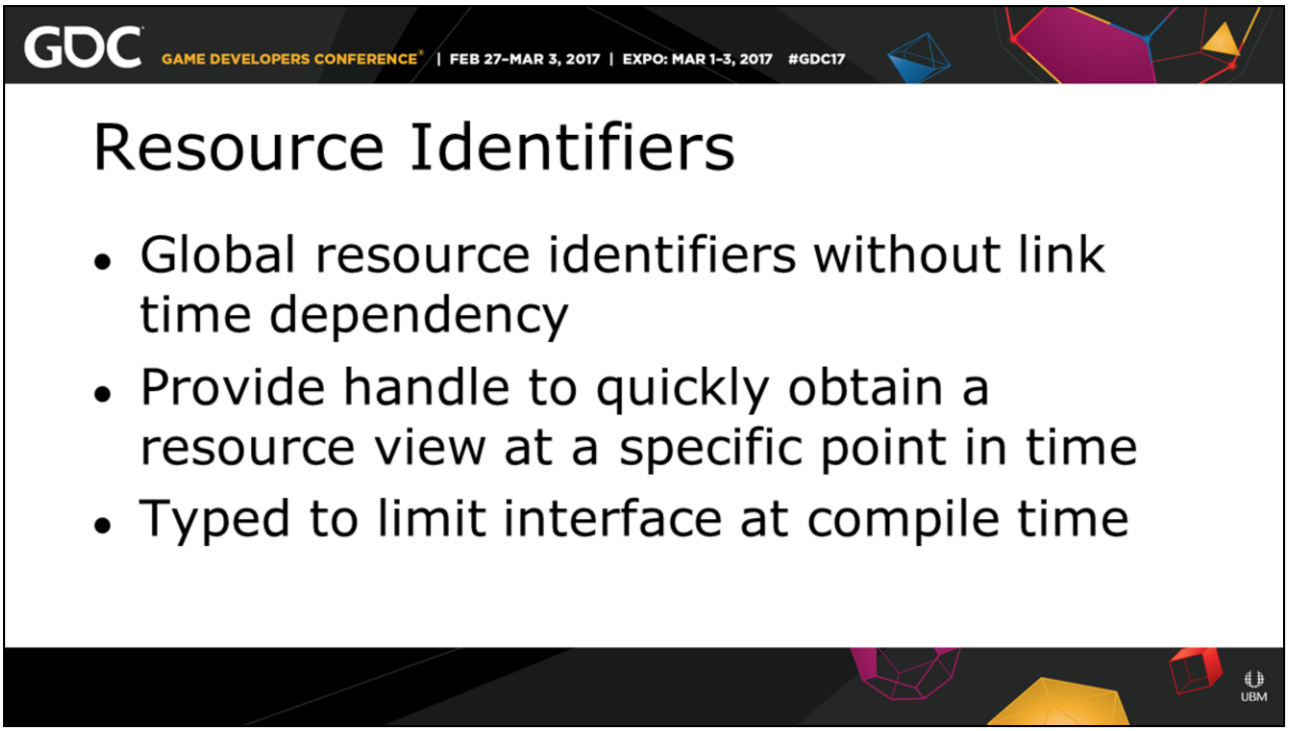
DepthWrite -> PS Resource

Single call to ResourceBarrier()

- As mentioned before, we have a list of transitions so we batch them in a single call to reduce the number of internal driver side-effects to the minimum set

Producer System: Implementation

- Resource Identifiers
- Two Steps: Gather resources, Record CMD buffers
- Scheduling



- Identifiers are used to logically address a specific resource from different producers

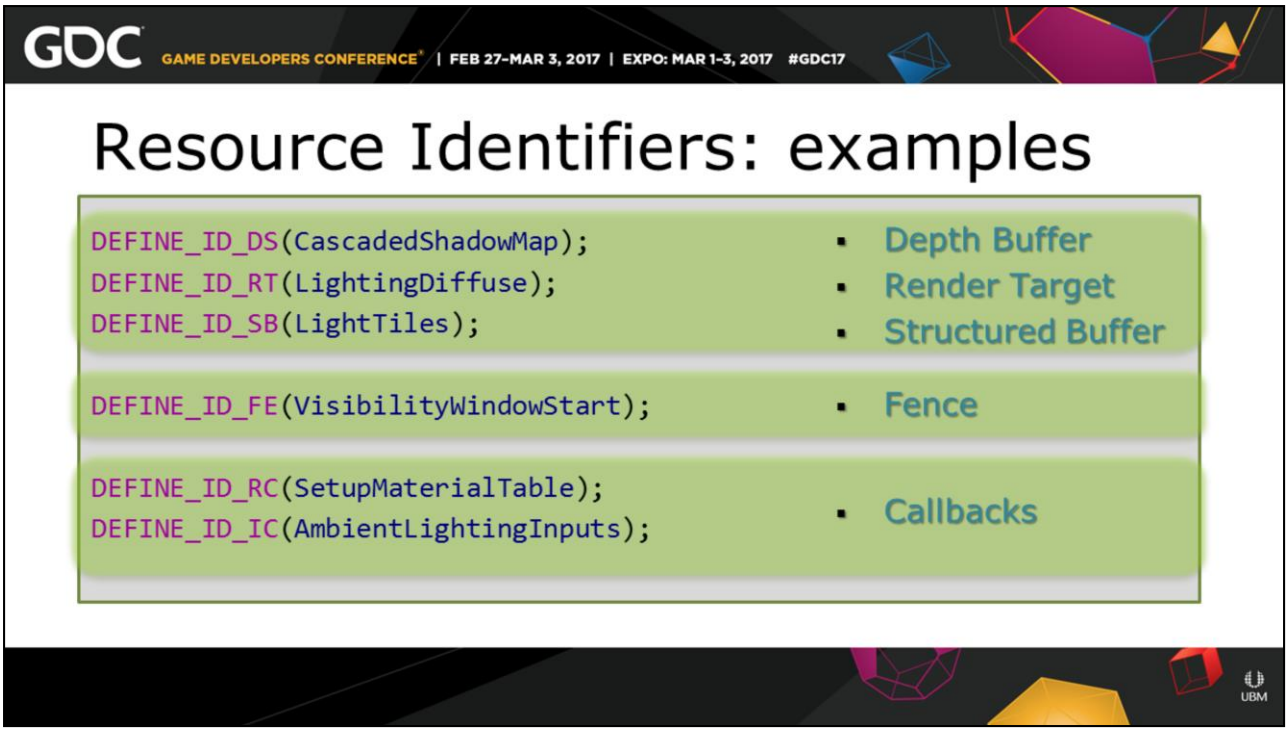
- Due to using resource IDs, producers don't need to depend on each other at compile time allowing us to implement resource memory aliasing, as I showed before, where different IDs point to the same memory at different times during the GPU execution timeline

- At static initialization time, we 'bake' these identifiers into an index that we can them use in the producer system to efficiently obtain it's associated resource view

- These IDs are also strongly typed in order to provide meaningful compile time error checking and allow function overloading

Resource Identifiers: examples

```
DEFINE_ID_DS(CascadedShadowMap);
DEFINE_ID_RT(LightingDiffuse);
DEFINE_ID_SB(LightTiles);

DEFINE_ID_FE(VisibilityWindowStart);

DEFINE_ID_RC(SetupMaterialTable);
DEFINE_ID_IC(AmbientLightingInputs);
```



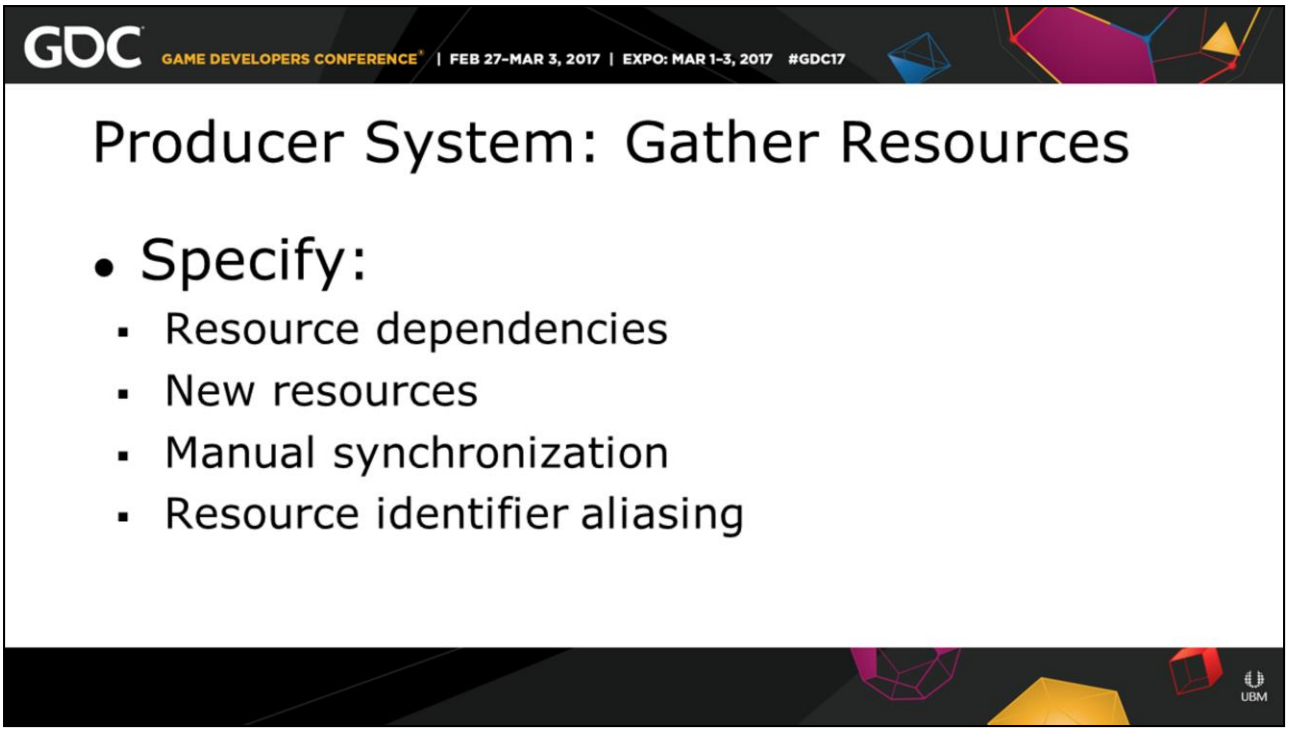
- Examples of resource IDs:

1) Depth surfaces
   Render targets
   Structured buffers
2) Explicit Fence resources
3) Callbacks:
   Render callbacks (that we can use to call functions form other producers
   Input callbacks (that group several other input dependencies together)

Producer system: interface

```
class GfxProducer
{
public:
    virtual void GetInputOuput(GfxScheduleContext& context);
    virtual void Record(GfxRenderContext& context);
};
```

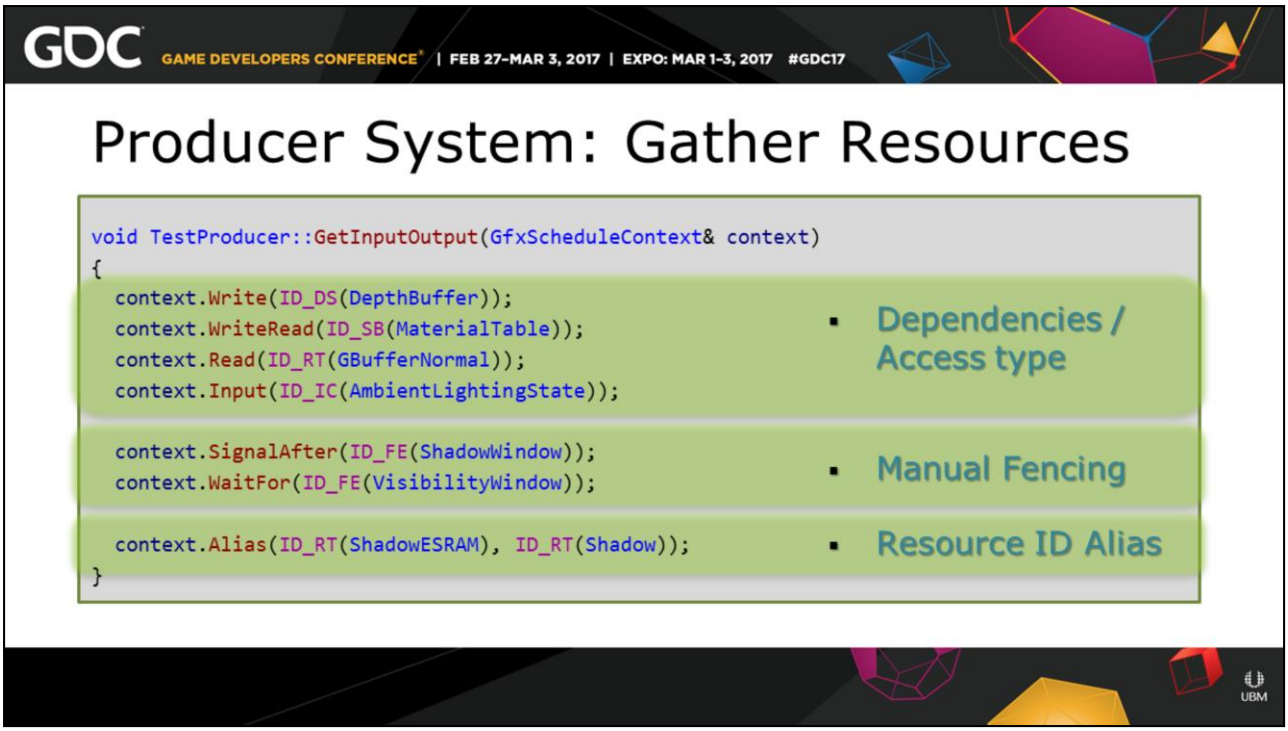

- At its core the producer interface is quite simple, only two main entry points:
  - Gathering of input/outputs
  - Recording of commands



- In GatherResources():
  - The user specifies resource dependencies (and the required access type: read, write, depth test, etc.)
  - Also, this is where new resources are specified (defining initial state parameters)
  - Any required manual synchronization
  - Resource identifier aliasing (essentially pointing a resource ID to another after a producer is scheduled)

Producer System: Gather Resources

```
void TestProducer::GetInputOutput(GfxScheduleContext& context)
{
  context.Write(ID_DS(DepthBuffer));
  context.WriteRead(ID_SB(MaterialTable));
  context.Read(ID_RT(GBufferNormal));
  context.Input(ID_IC(AmbientLightingState));

  context.SignalAfter(ID_FE(ShadowWindow));
  context.WaitFor(ID_FE(VisibilityWindow));

  context.Alias(ID_RT(ShadowESRAM), ID_RT(Shadow));
}
```

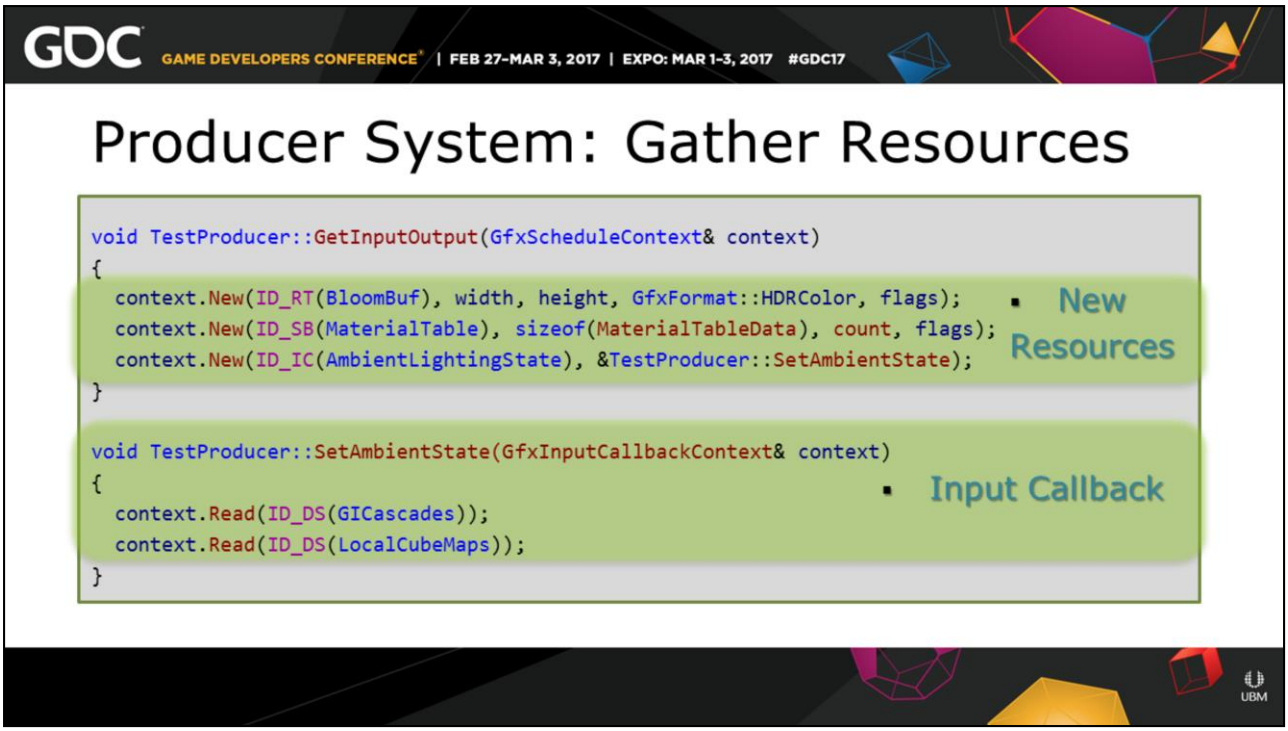


- Examples:

1) At the top you can see examples of specifying resource dependencies and the required access type
2) Manual synchronization (in the middle)
3) Resource identifier aliasing at the bottom

Producer System: Gather Resources

```
void TestProducer::GetInputOutput(GfxScheduleContext& context)
{
  context.New(ID_RT(BloomBuf), width, height, GfxFormat::HDRColor, flags);
  context.New(ID_SB(MaterialTable), sizeof(MaterialTableData), count, flags);
  context.New(ID_IC(AmbientLightingState), &TestProducer::SetAmbientState);
}

void TestProducer::SetAmbientState(GfxInputCallbackContext& context)
{
  context.Read(ID_DS(GICascades));
  context.Read(ID_DS(LocalCubeMaps));
}
```



- Examples:

1) Creating new resources:

  - Where we pass the initial configuration

2) Input callback example: where we can provide a shortcut to depend on a number of resources at the same time

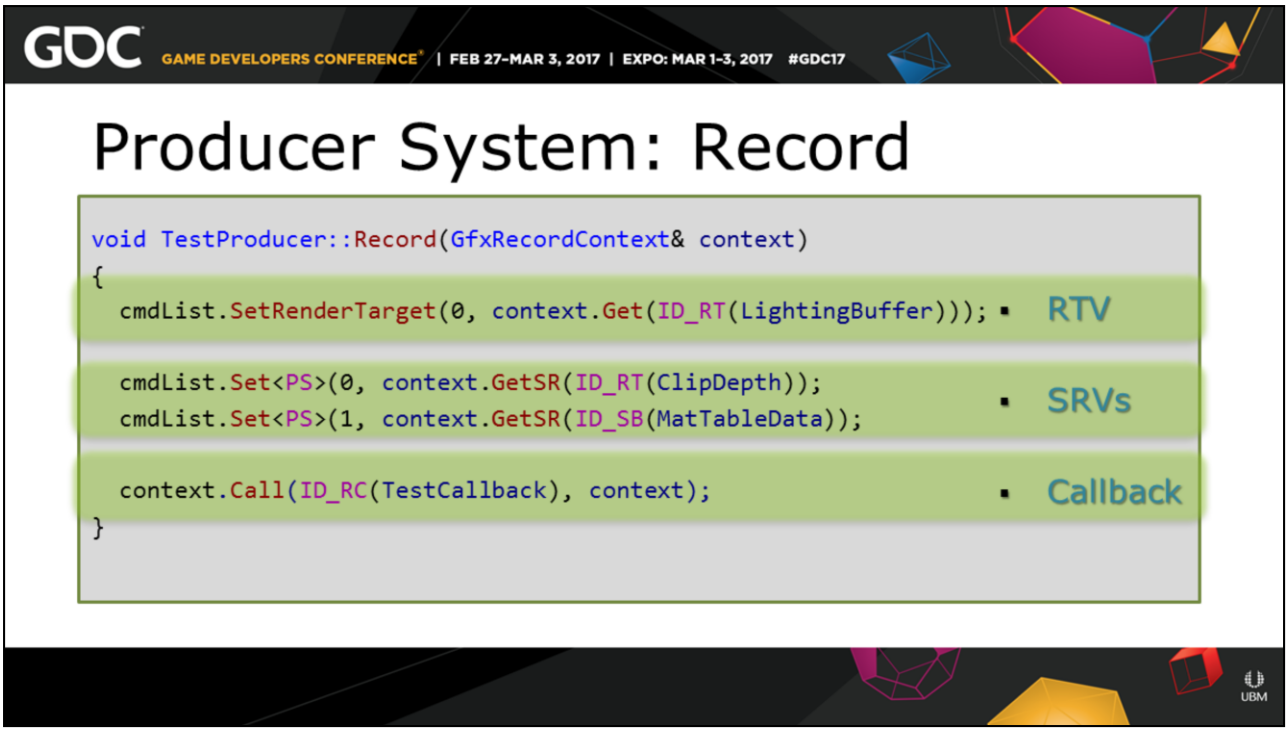
Producer System: Record

- Each producer is assigned a unique resource context
- Producers index into this context to get resources views to use in a CMD list

- During the Record step, the requested resources can then be accessed via a producer specific resource context using the resource IDs.
- This resource context is unique to every producer, and contains the assigned resource views for the point in time where this producer's CMD lists execute on the GPU

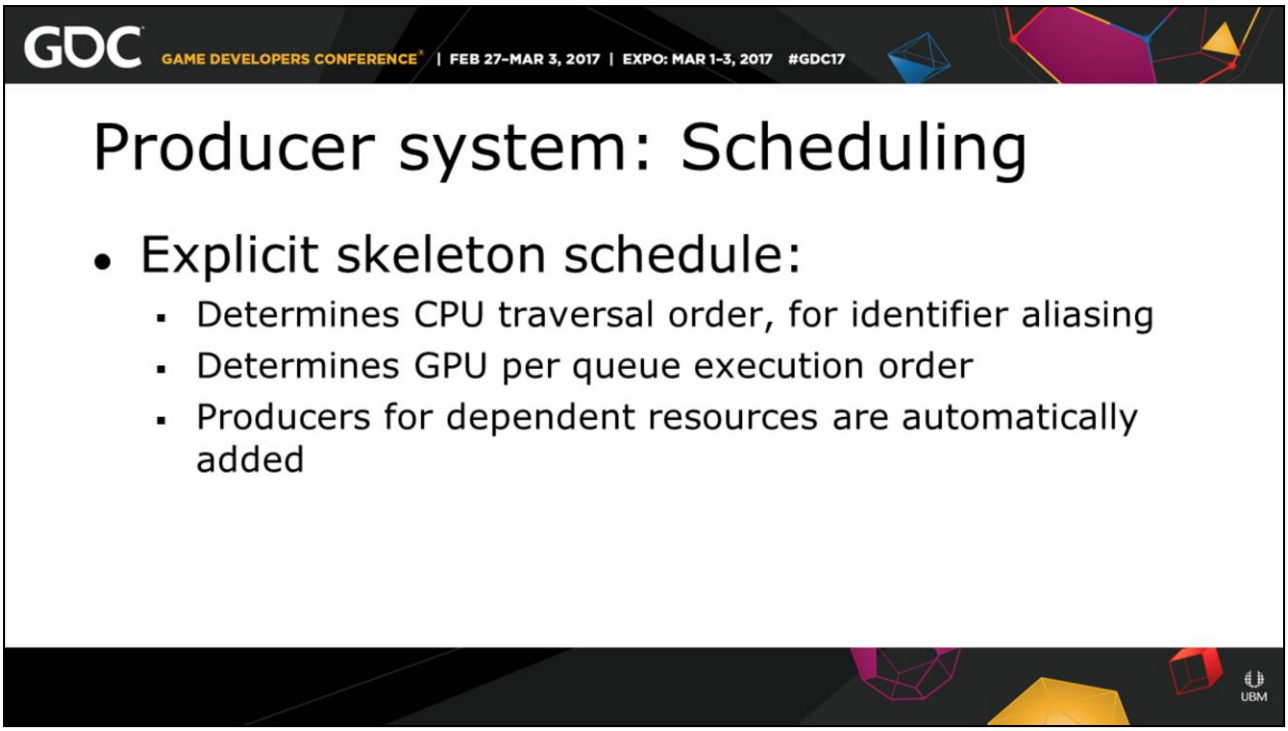
Producer System: Record

```
void TestProducer::Record(GfxRecordContext& context)
{
  cmdList.SetRenderTarget(0, context.Get(ID_RT(LightingBuffer)));

  cmdList.Set<PS>(0, context.GetSR(ID_RT(ClipDepth));
  cmdList.Set<PS>(1, context.GetSR(ID_SB(MatTableData));

  context.Call(ID_RC(TestCallback), context);
}
```



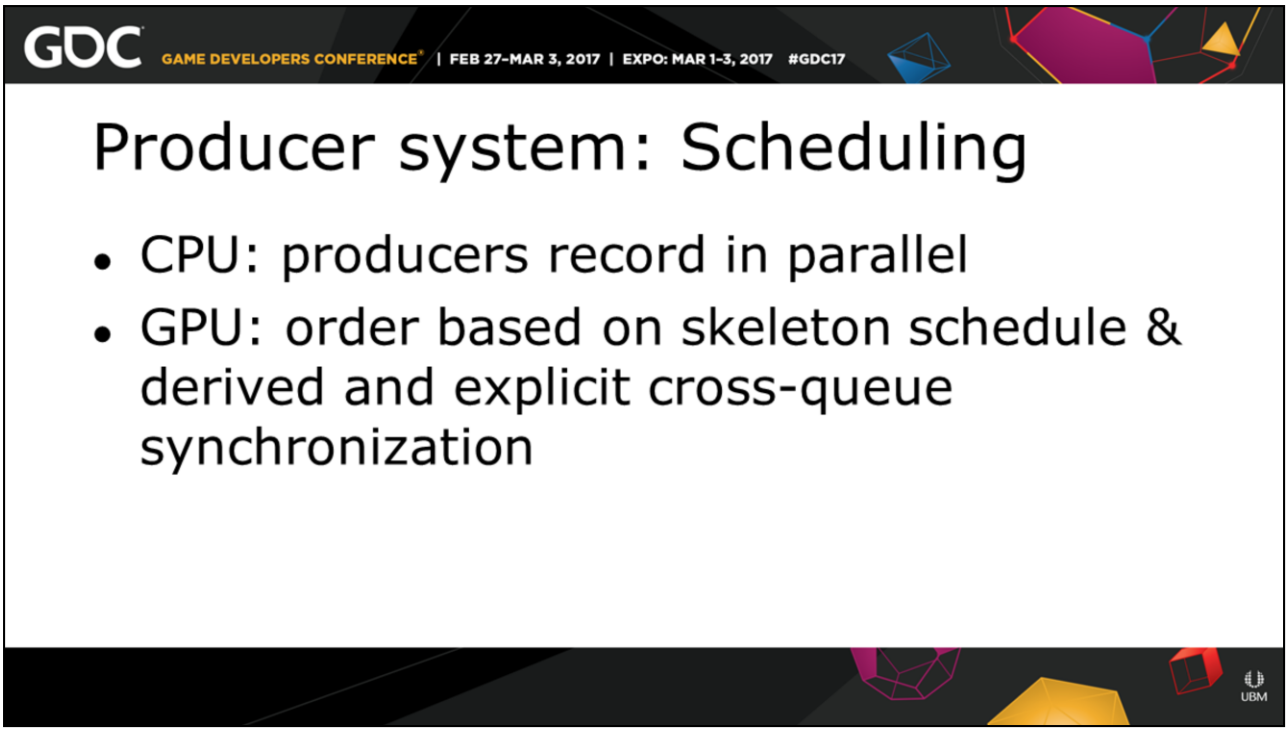
- Here are some examples of obtaining resource views using the IDs…

1) Setting a render target view
2) Shader resource views
3) A callback into another producer (that might issue it's own set of commands)



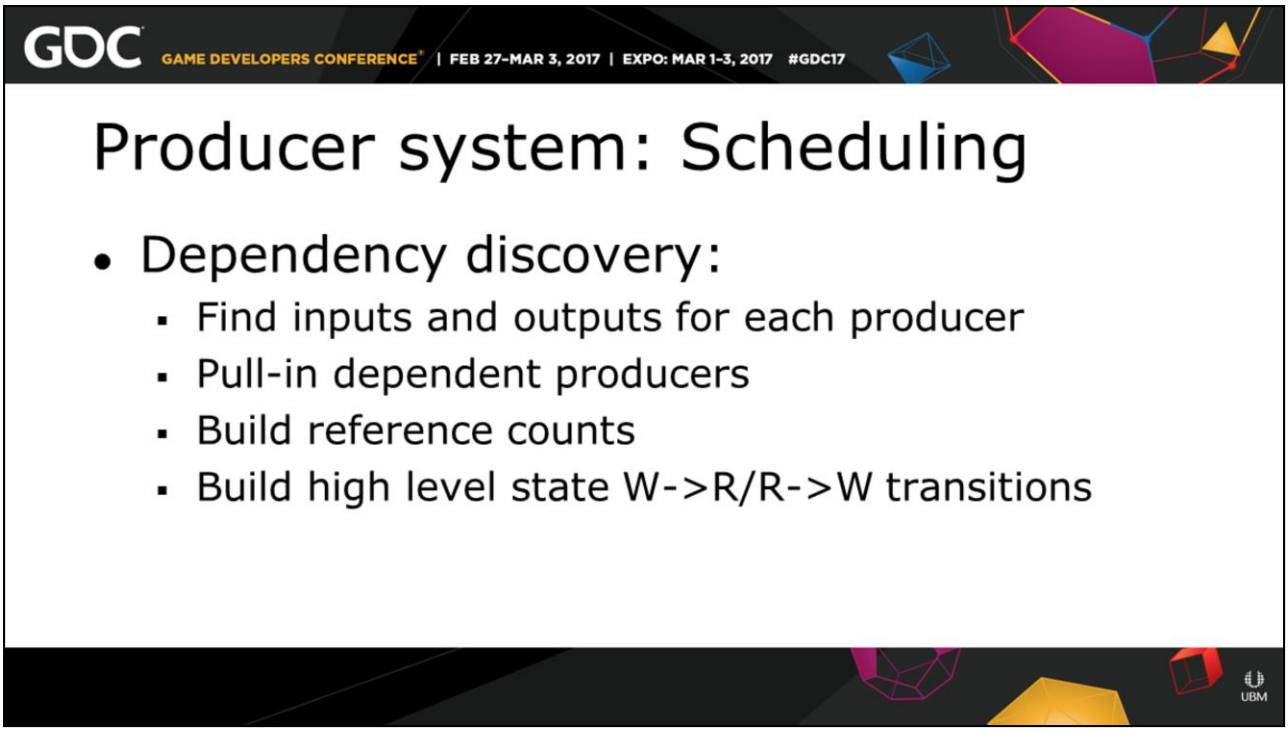
- We could theoretically construct the whole GPU schedule just based on the desired final outputs and producer dependencies

- To have more control on the actual execution order we decided to support a partial explicit skeleton
  - Here we add at least a few key producers (where relative execution order needs to be set more precisely)
  - It's also easier to add platform/configuration specific producers (ex: explicit transfer memory between memory pools [ESRAM/DRAM] on console
  - This is also where we would generally add producers that define the bounds of executing windows to match workload across queues, etc.
  - Producers of required resources, that are not specified in this skeleton are pulled in automatically by resource dependencies from other producers



- On the CPU, producers record their CMD lists in parallel, maximizing CPU utilization across available work threads (this also forces us to remove CPU dependencies between producers. We have a shader parameter system, that allows a producer to fill a set of parameters to be used by other producers)

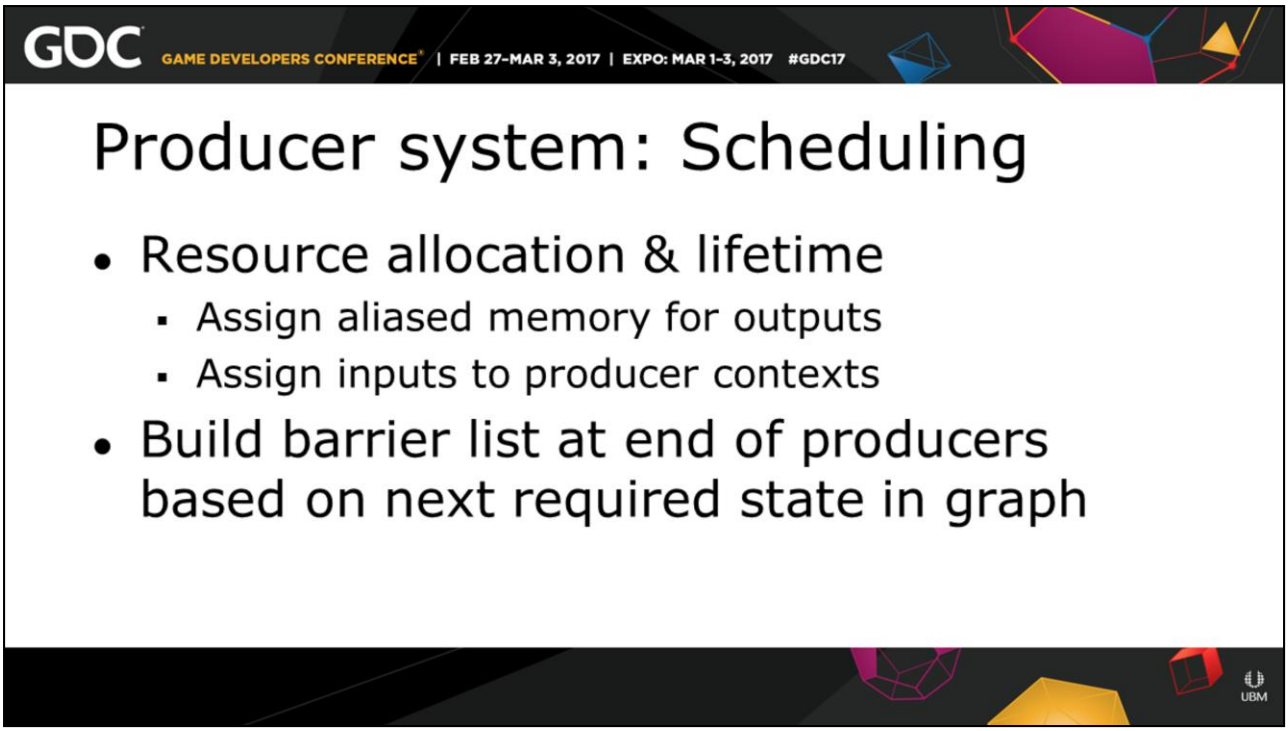
- The order of execution on the GPU is based on the skeleton schedule ordering and cross-queue derived and explicit synchronization



- The first step in building the schedule is to perform dependency discovery (This is essentially obtaining all of the inputs and outputs for each producer)

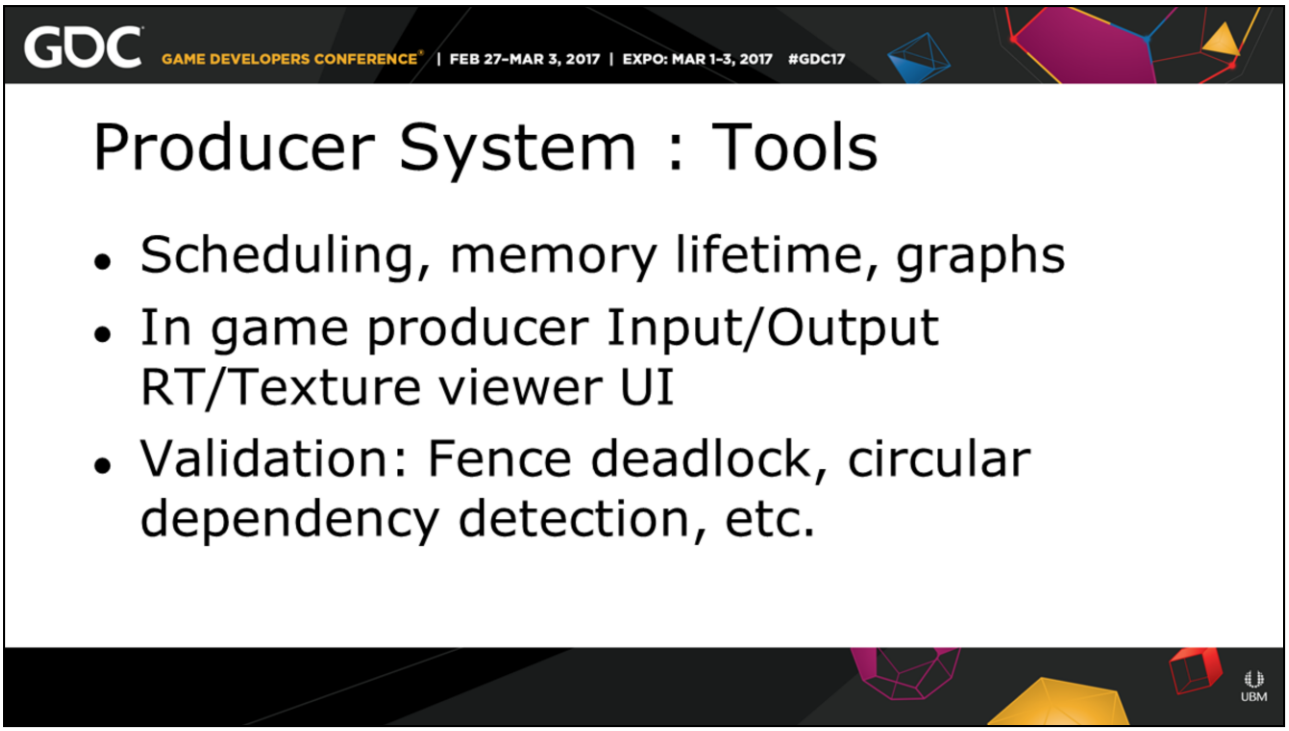
- During this process we also pull any producers for resources that have not yet been scheduled and build reference counts for each resource to later drive resource memory allocations

- We also keep track of resource, write->read and read->write transitions across producer boundaries and produce a list of necessary GPU synchronization across queues, as resources output from a producer running on one queue, are requested as inputs to a producer on another queue



- We then have enough information to do another pass and allocate and free the resources at the appropriate producer, filling its resource context with the correct views.

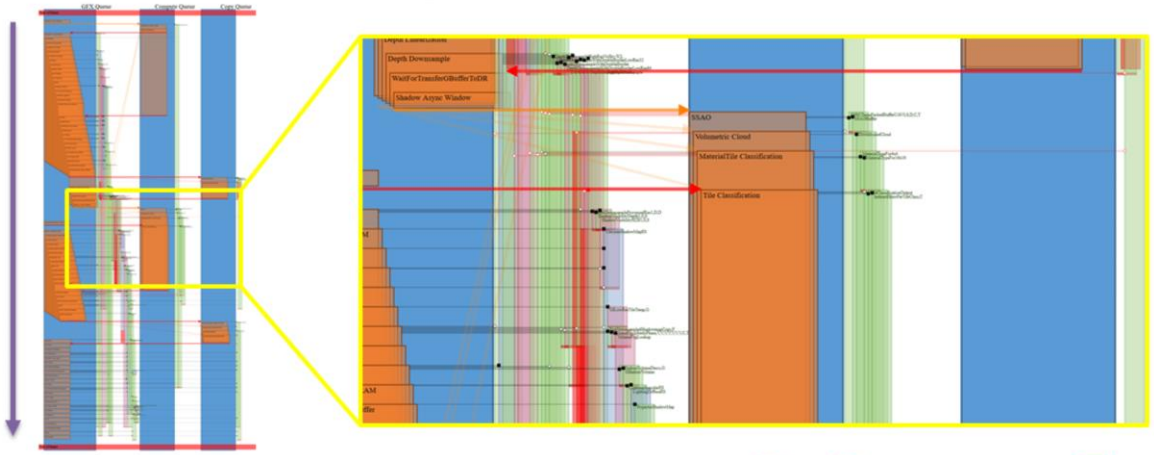
- Finally, we can generate the barrier list based on resource types and the high level resource transitions



- One of the key advantages of this kind of system is that it enables us to develop very rich debugging tools, that exploit the data we get from the explicitly defined dependencies and automatic scheduling:
  - The first tool we developed is a graph of the generated schedule including a lot useful information (you can see an example of this graph on the next slide)
  - We also generate a number of other graphs for easy visualization of memory aliasing
  - Another tool we have is an in game resource viewer, where we can inspect the contents of a resource at producer boundaries (by injecting a resource copy in the schedule to capture the resource state)

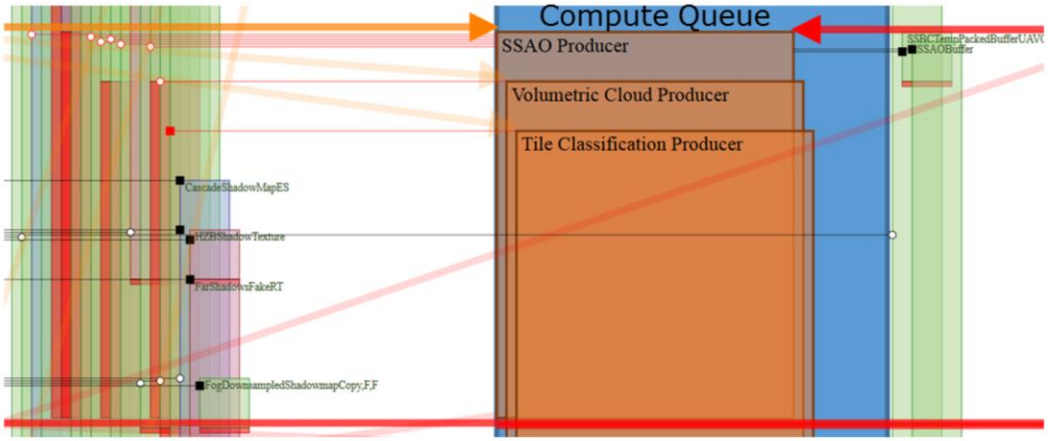
- Finally, we have a number of validation tools that can help detect user induced deadlocks or circular dependencies

Schedule Graph (autogenerated)

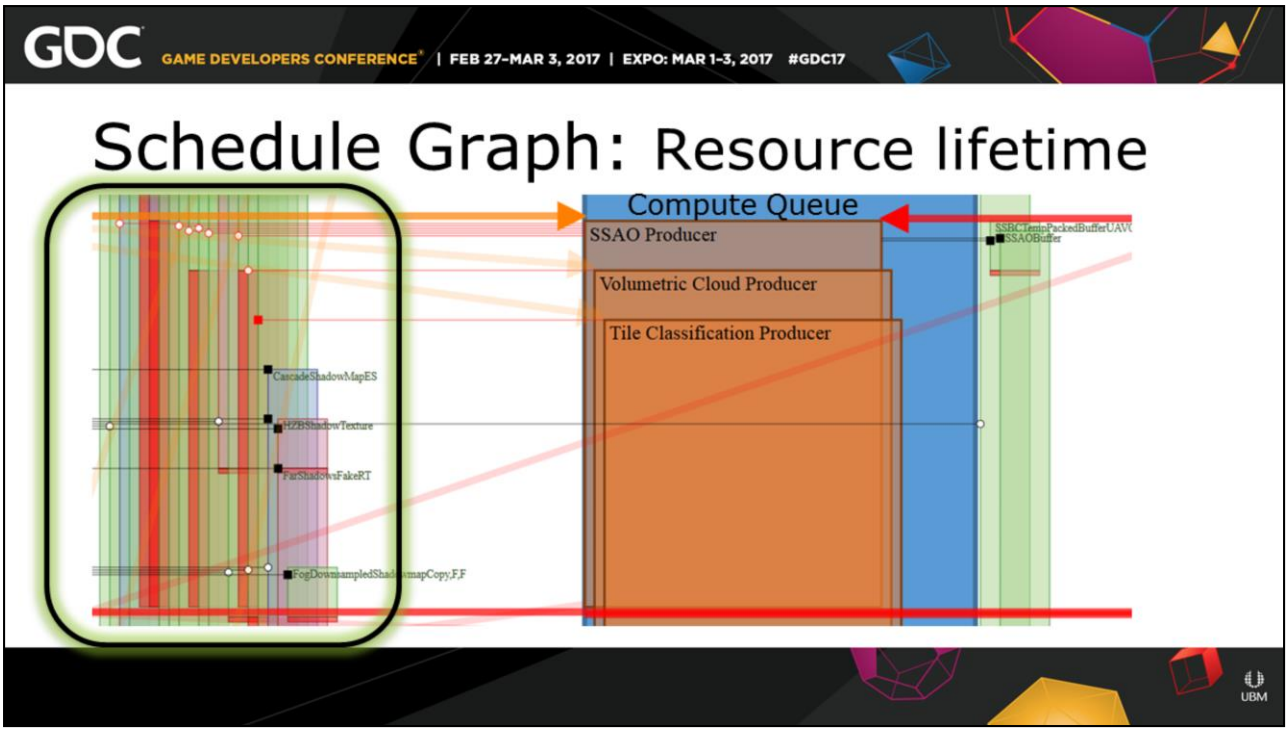


- Here you can see a diagram of one configuration of our rendering pipeline
  - GPU execution order is from top to bottom
  - GPU queues are represented in Blue (graphics on the left, compute in the middle, and copy on the right)
  - Producers are shown in orange
- The graph is quite information dense since it exposes a lot of details
- On the right side you can see a window of the schedule, centered around some async. compute work

Schedule Graph

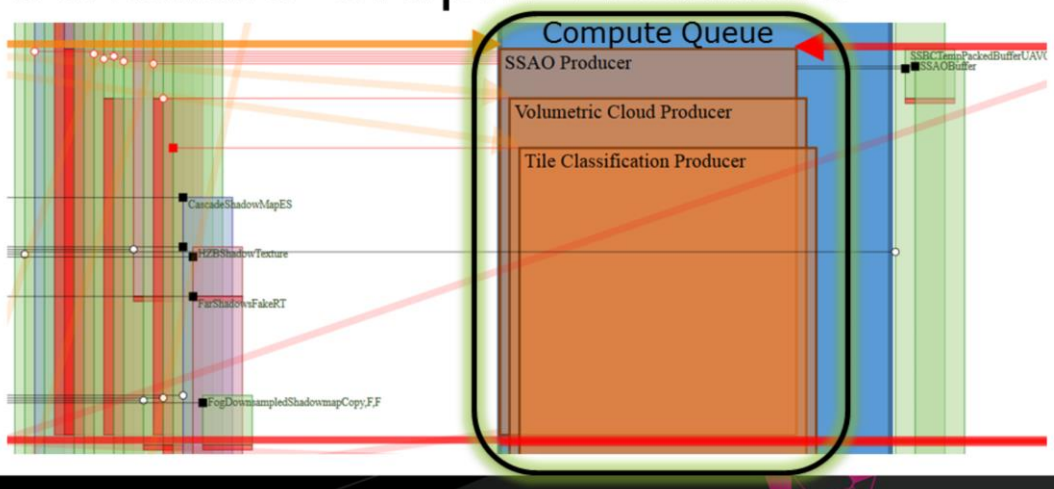


- Zooming further…



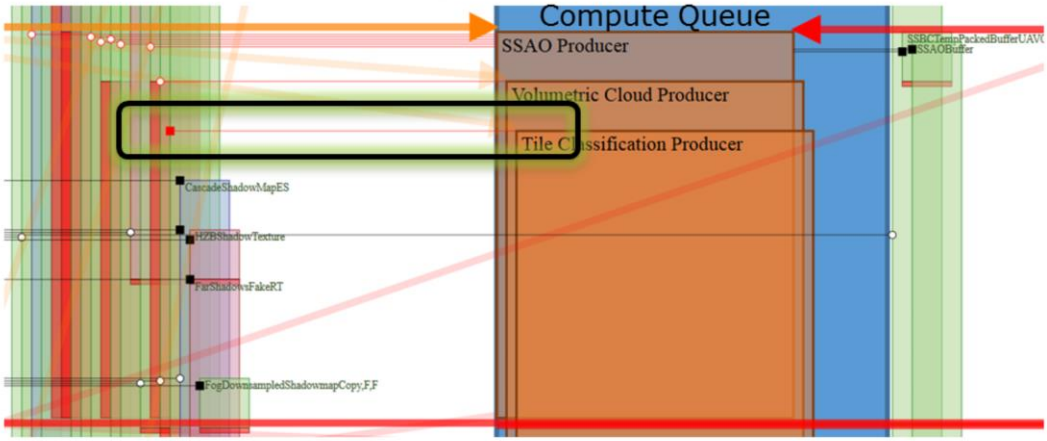
- On the left, you can see resource lifetimes bars (from top to bottom), color coded by memory pool

Schedule Graph: Producers



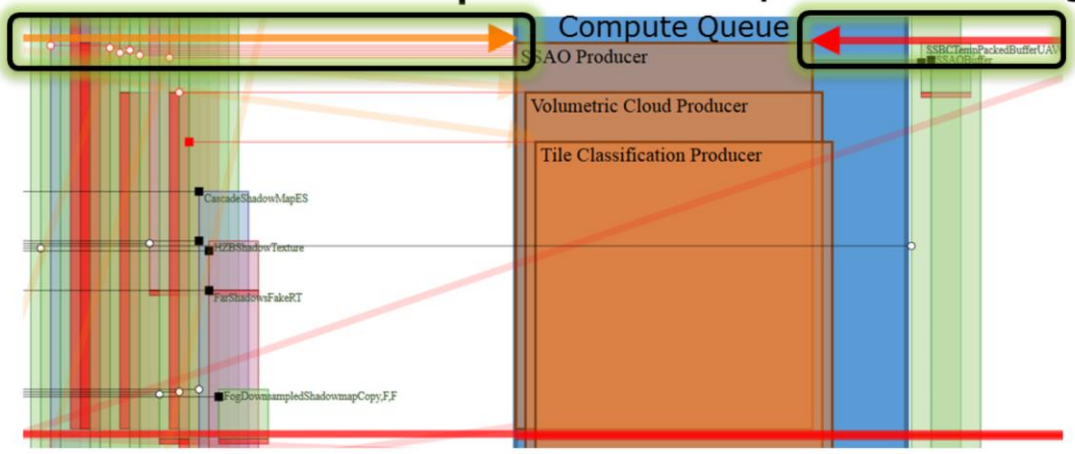
- Some producers running on the async. compute queue in orange…

Schedule Graph: Resource dependencies



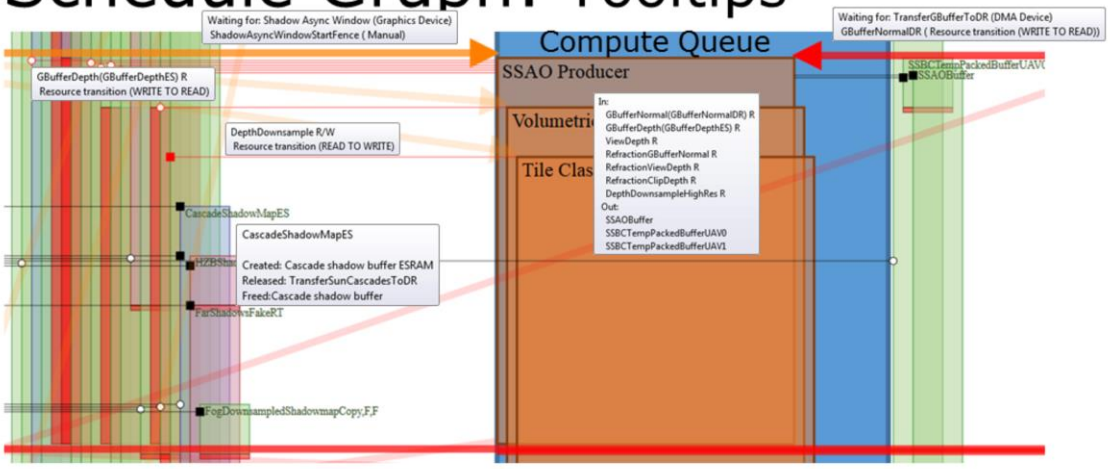
- Dependencies from producers to resources, represented by the many horizontal lines linking them to the resource lifetime bars on the left

Schedule Graph: Cross queue fencing



- Cross-queue fencing, represented by the thick arrows

Schedule Graph: Tooltips

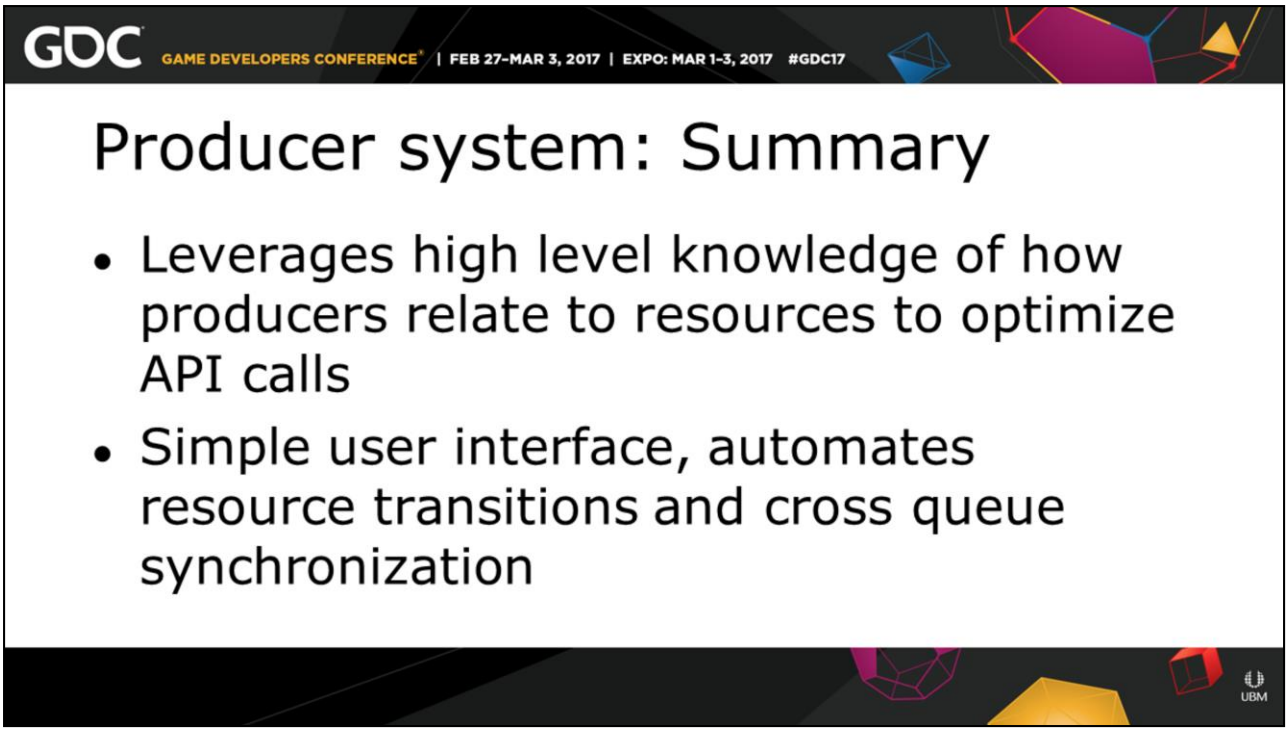


- Each of these elements has tooltips that show useful information when we hover over them, for example:
  - List of resources a producer creates and depends on
  - Parameters of resource state transitions
  - Fencing parameters
  - etc.

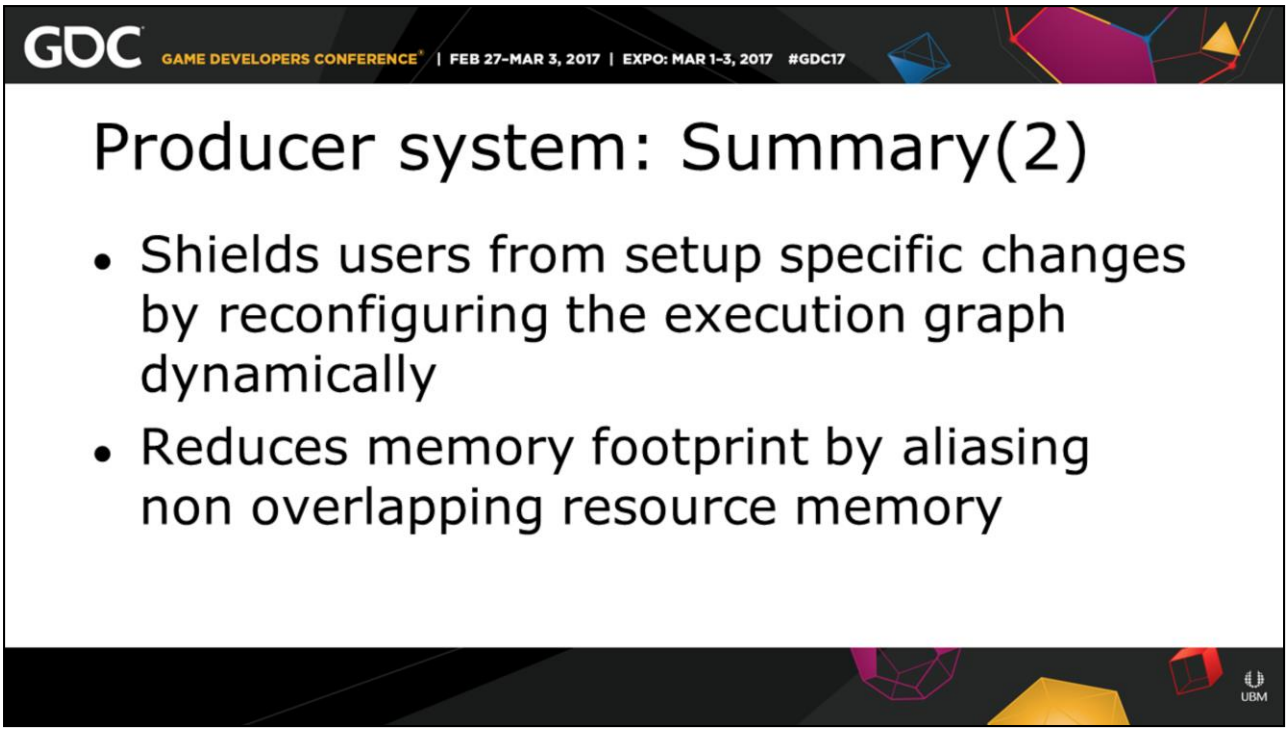
Producer system: Stats

	Average per frame	Notes
Producer count	50	Have a lot of small ones that get their CMD lists merged
ResourceBarrier() calls	150	Still high, mainly intra-producer UAV/for indirect buffers
Fence count	5	Most resources are frame fenced
ExecuteCommandLists()	15	Due to batching
Producer Resource count	130	Still have a few resources outside producers
Resource Memory Footprint	200Mb (With aliasing)	375Mb (With no aliasing)

- We have a lot of producers (~50 on an average frame), but a lot of them record a small amount of work so we 'merge' their CMD lists (that is, we end up record them into the same DX12 command list)
- The ratio of barrier to producers is still high, the reason behind this is mainly UAV barriers within producers for our GPU culling and indirect parameter buffer filling dispatches
  - We tackled these on console with async. compute, but on PC, fence granularity is not good enough for this kind of synching, we are doing some further work to reduce these, at the expense of some higher memory usage
- Fence count is quite small since most of the memory related lifetime management is done with frame fencing
- For a 1080p frame (not counting persistent or tiled resources) we have almost 50% memory saving vs not using memory aliasing (memory aliasing still has a few caveats, tier1 heap restrictions relating to resource type segregation and some rather high alignment requirements for some resource types)
  - As you would expected, the bulk of our resource memory footprint is for RO textures (we do not track these in the producer system) which we tackle with mip streaming.



- Simple user interface: producers, which queues they execute on, which inputs they depend on, and which new outputs they produce. This interface relieves users from API responsibilities like issuing barriers and doing cross-queue synchronization, by doing them automatically



- Shields users from configuration specific features (like disabling certain passes for performance scaling on lower end machines) by reconfiguring the execution graph dynamically, taking care of all the side effects that it might entail in terms of synchronization, barriers, etc.

Producer system: Summary(3)

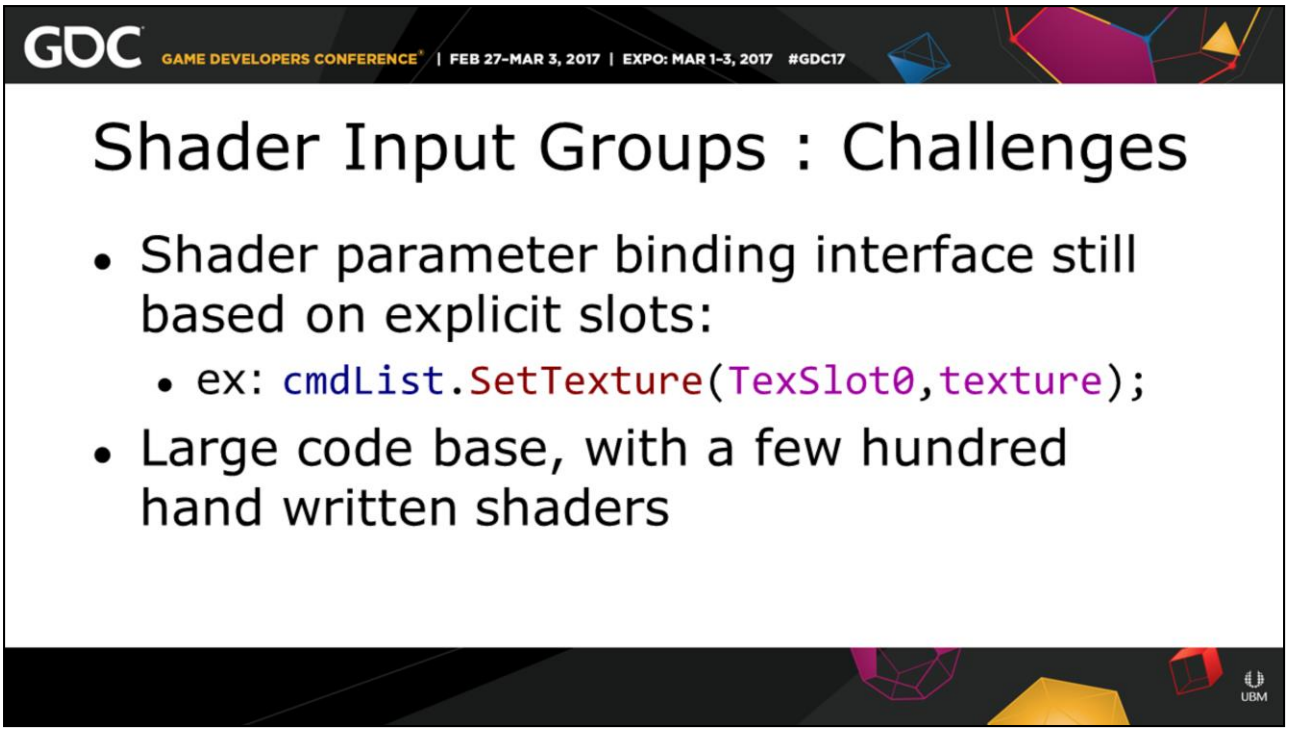
- Maximize CPU utilization by recording CMD lists in parallel
- Implements small producer and CMD list execution batching to reduce API call overhead

- Enables us to maximize CPU utilization by distributing CMD recording over the available worker threads, while at the same time hiding some details like cost of frequent exec calls, by batching and coalescing small producers into the same CMD list

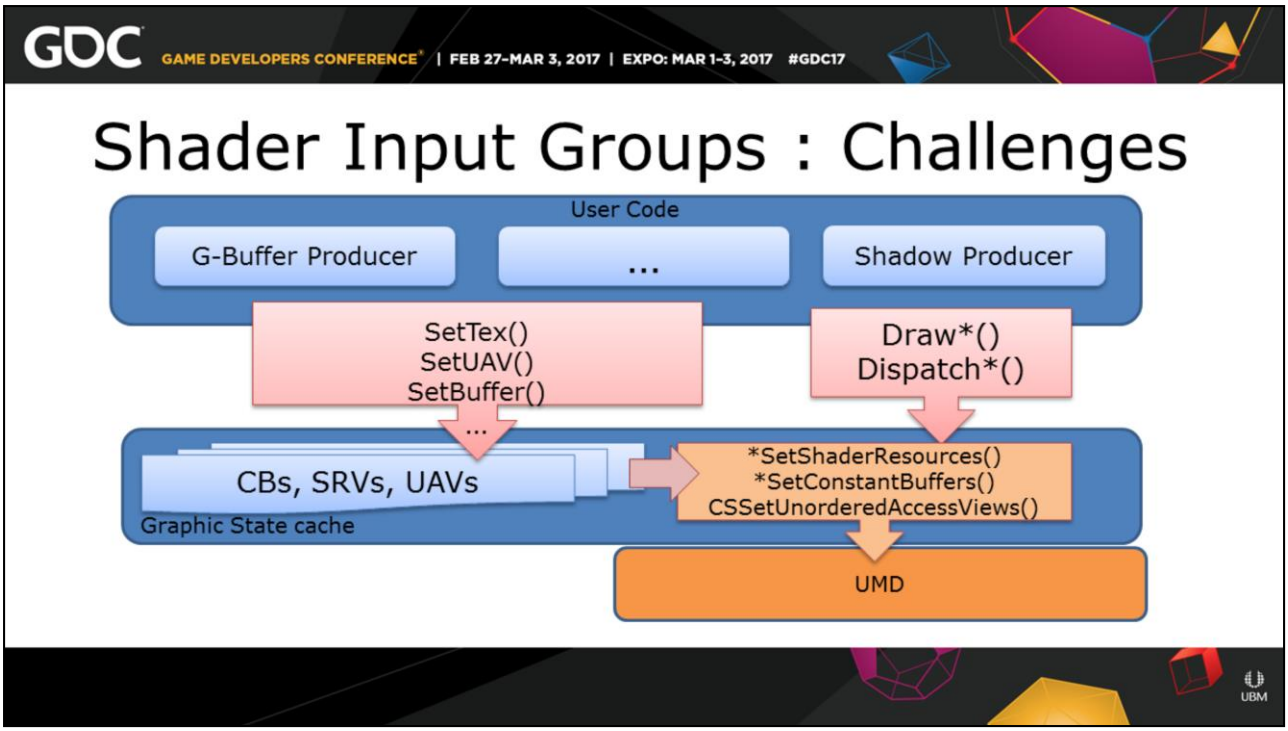
Shader Input Groups

Shader Input Groups

- A system to setup shader parameters to match the DX12 binding model
- Expose this binding model to users via a simple abstraction



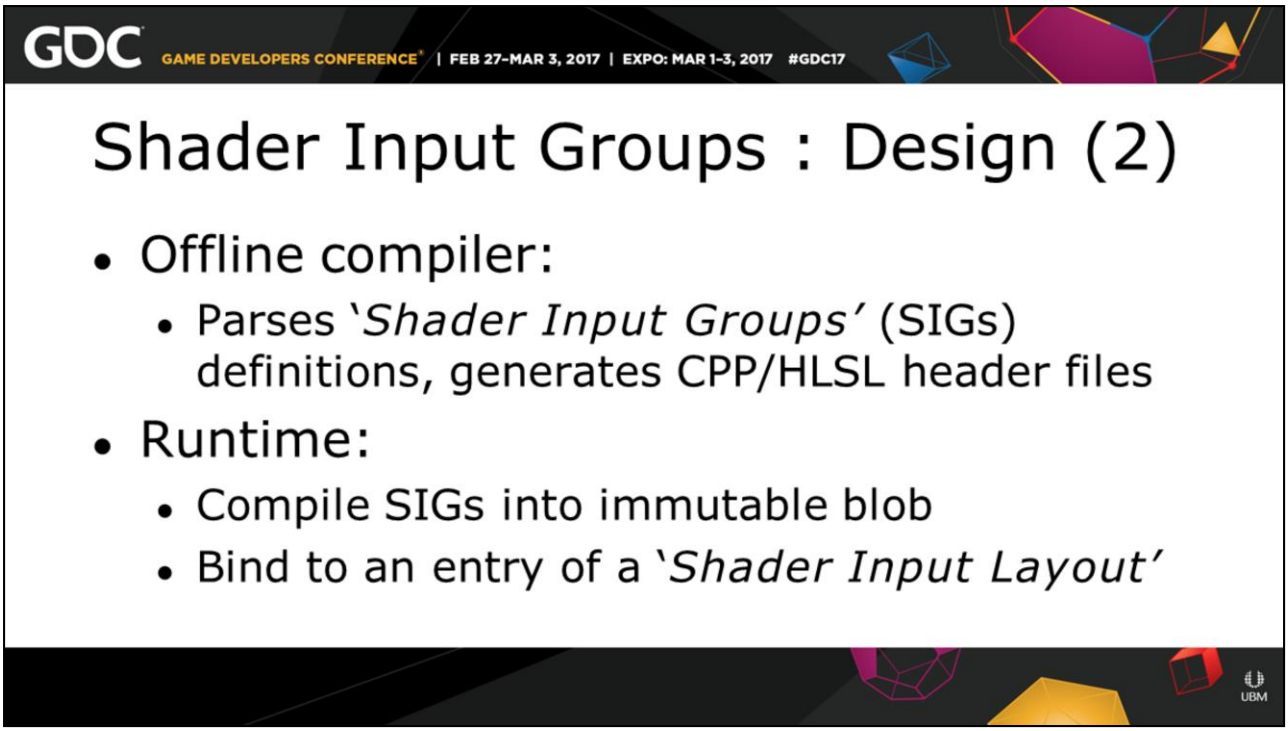
- We had some challenges going in, mainly due to:
  - Our existing shader input binding interface was very granular, still based on explicit slots
  - We also had a significant amount of hand-crafted shaders that would have to be ported (we had many more generated from shader graphs, but for these we could easily patch the code generator)



- Here you can see a diagram exemplifying setting shader parameters with the previous granular interfaces:
  - Producers record command lists in parallel and issue a lot of very granular input resource changes
  - Work is done at Draw/Dispatch time, converting the cached data into the required shader inputs (a lot of this work is also repeated if any of the parameters change due to the slot based interface)

Shader Input Groups : Design

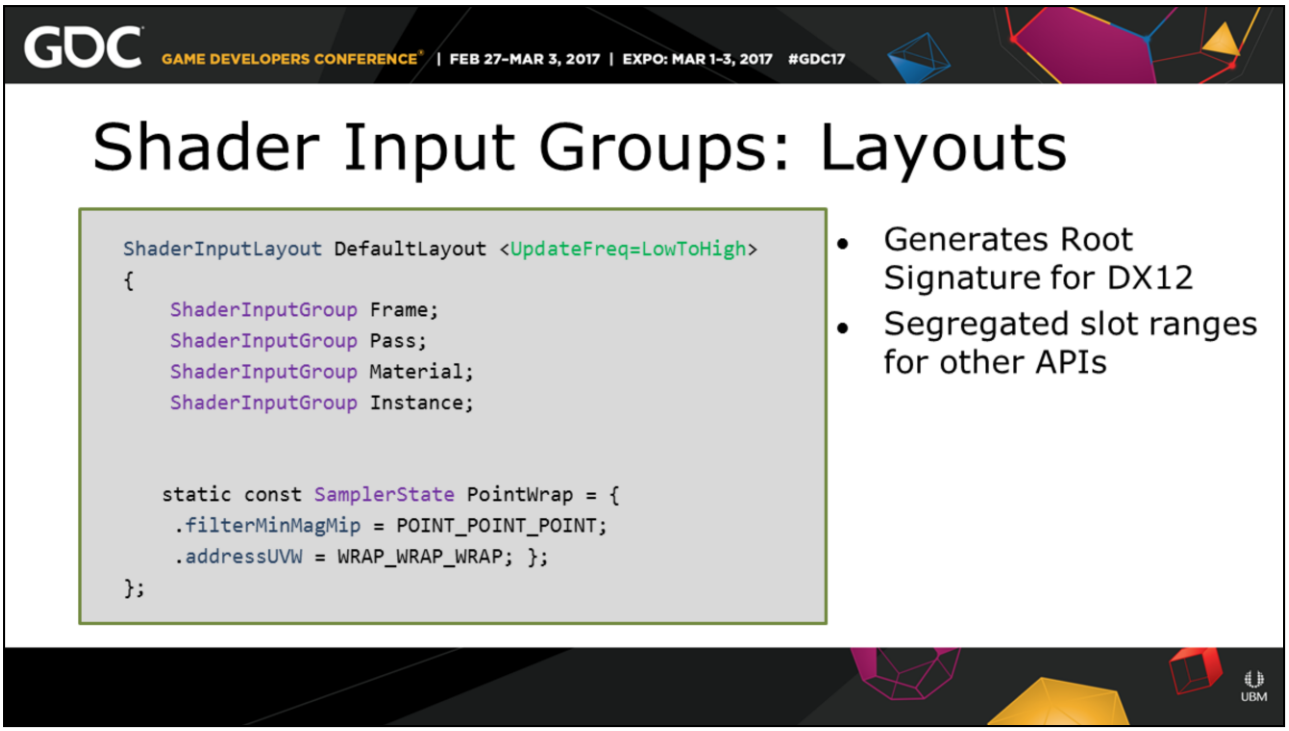
- Bind parameters as groups, precompiled at the rate of change
- Uniform Set/Get interface across all APIs via auto-generated headers



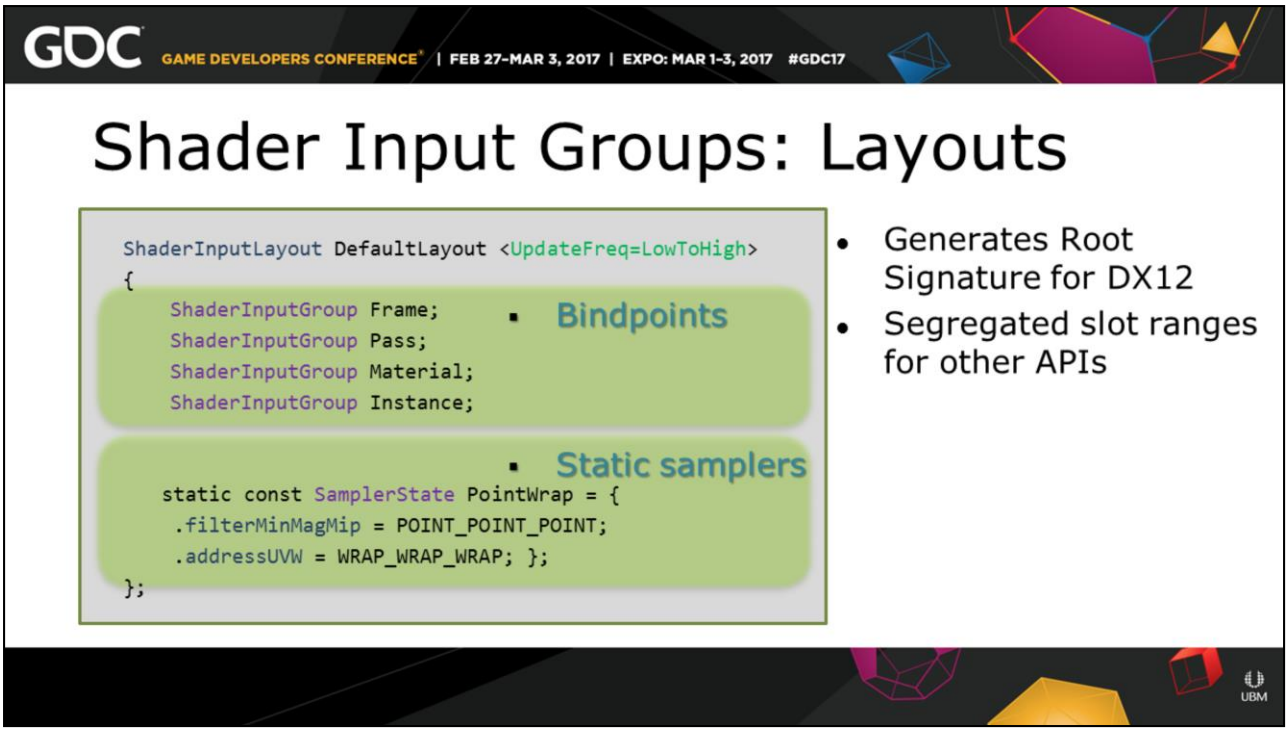
- In order to transition to a precompiled blob of parameters approach we developed an offline compiler that parses what we call "Shader Input Group" definitions and generates headers files that abstract the setting/getting of parameters in CPP and HLSL respectively

- At blob compile time we can (for DX12):
  - Fill descriptor heaps and build descriptor tables
  - We can also hide how constant memory is copied, for example by using the upload heap directly or batch updates to reduce copy queue related transitions

- Parameter binding points are represented in what we call a "Shader Input Layout", which you can see an example of in the next slide



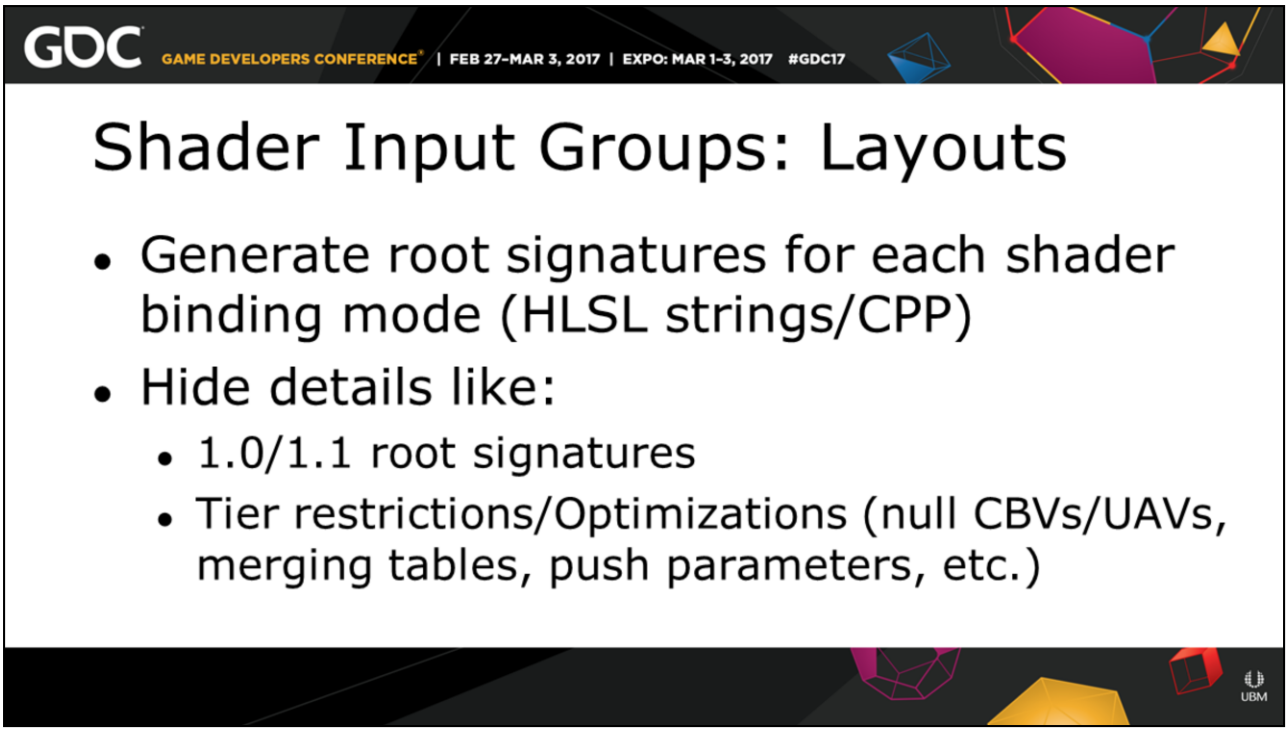
- Here is an example of a layout definition
- We already grouped constants by frequency of update with segregated constant buffers, now we extend this concept to all shader input parameters
- For DX12, this layout will generate a root signature(s), based on the max required resource counts from the Shader Input Groups that bind to the same entry



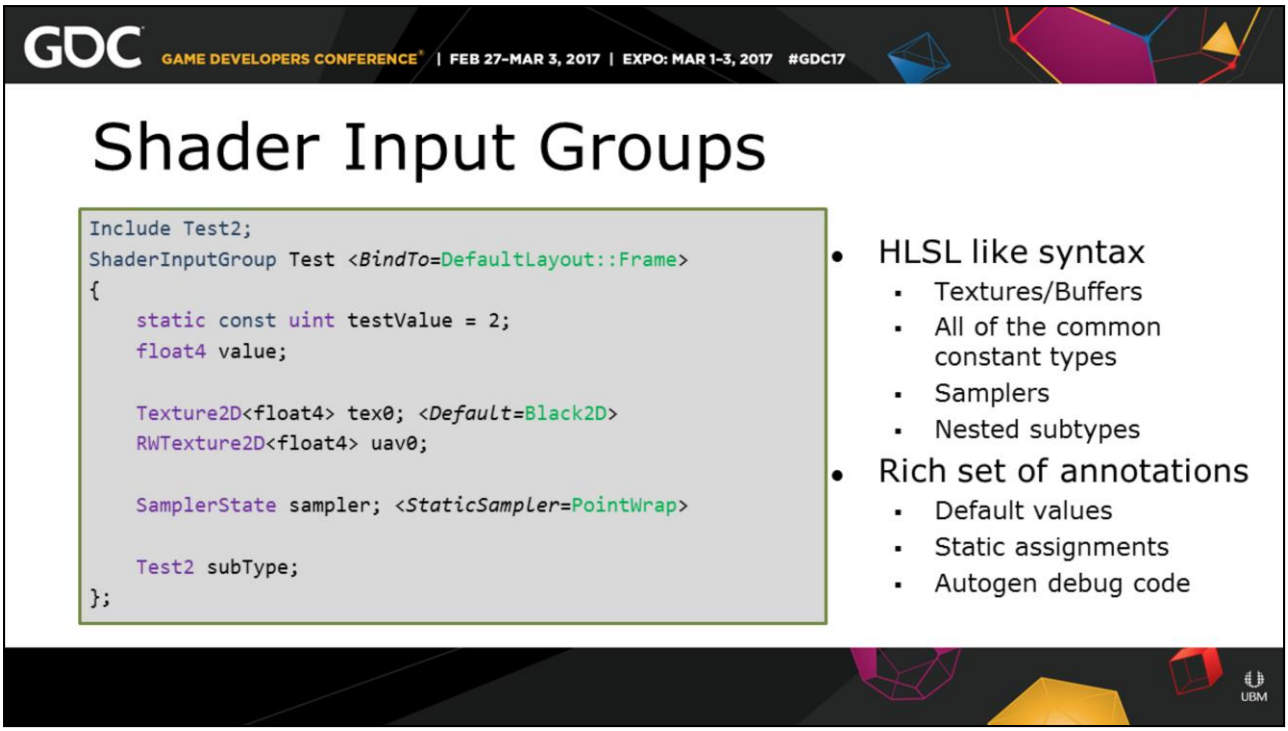
- Here you can see examples of those bind points (segregated by update frequency)

- In DX12, each layout bindpoint entry, essential defines a set of descriptor tables (ex; a CBV/SRT/UAV table and a SAMPLER one if needed)

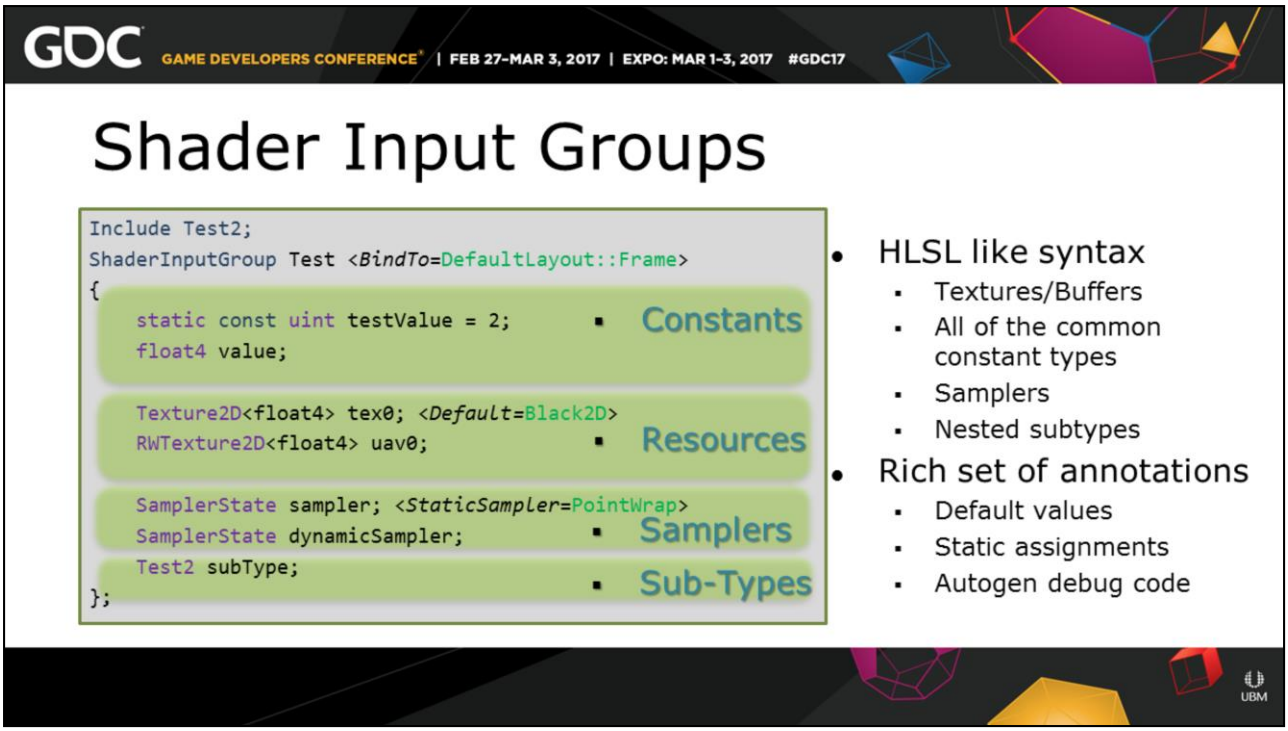
- We also support handling of static samplers automatically across APIs by generating the required CPP side code for platforms that don't support it



- For a layout, the SIG compiler:
  - Generates several versions of the root signature for each stage combination with the required visibility flags
  - Generates rootsig 1.1 versions with the appropriate static flags for runtime hinting to the driver so it can optimize parameter management
  - Handles tier restrictions and implementing other optimizations to reduce root table count, placing parameters directly at the root, etc.



- "Shader Input group" definitions have pretty much HLSL syntax, plus some annotation support



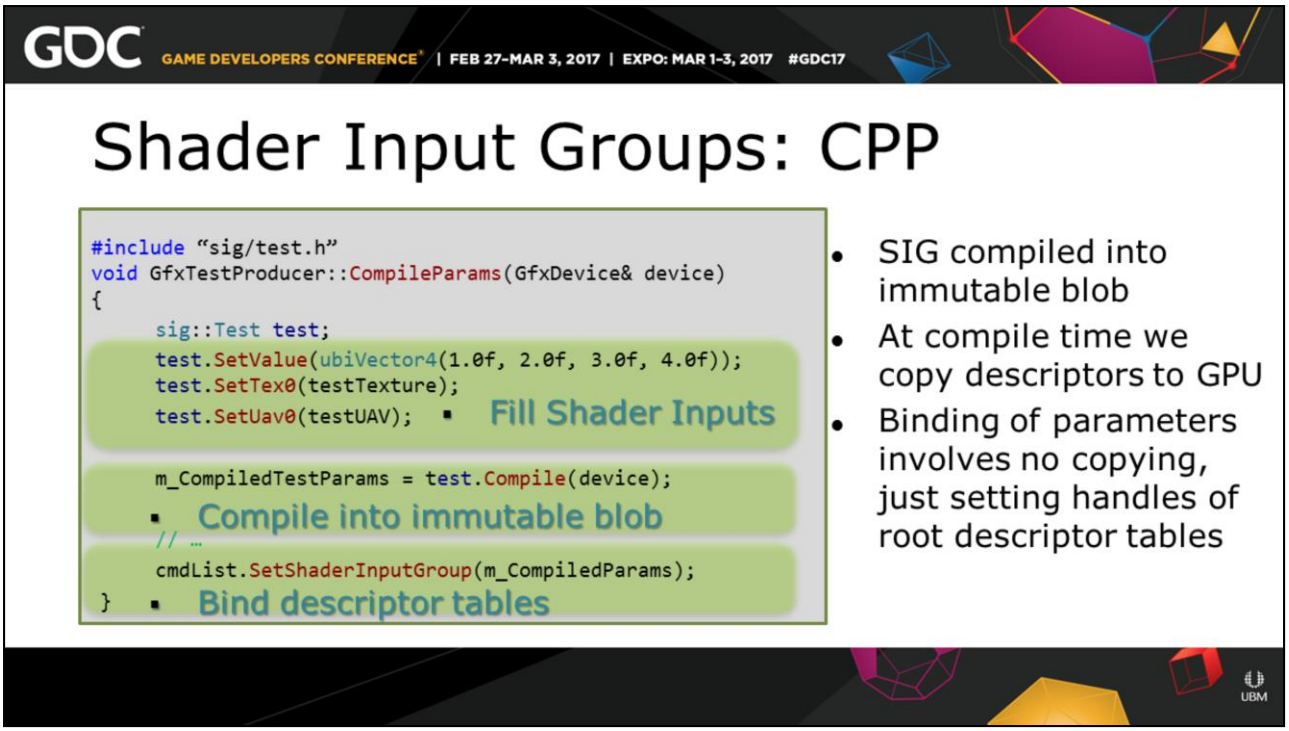
- Here we can specify constants and a full set of resources and samplers
- We support a wide range of annotations (for example: for specific bind points [allow building of the descriptor tables], defining resource defaults, generating debug code [ex: GPU debug print code], etc.)
- We can also specify nested types (ex: Test2), in order to manage segregated groups of parameters more easily

Shader Input Groups: CPP

```
#include "sig/test.h"
void GfxTestProducer::CompileParams(GfxDevice& device)
{
      sig::Test test;
      test.SetValue(ubiVector4(1.0f, 2.0f, 3.0f, 4.0f));
      test.SetTex0(testTexture);
      test.SetUav0(testUAV);

      m_CompiledTestParams = test.Compile(device);

      // …
      cmdList.SetShaderInputGroup(m_CompiledPassParams);
 }
```

- SIG compiled into immutable blob
- At compile time we copy descriptors to GPU
- Binding of parameters involves no copying, just setting handles of root descriptor tables

- At runtime, we use the auto generated setter interface (these provide provide a layer of validation based on type information and annotations)



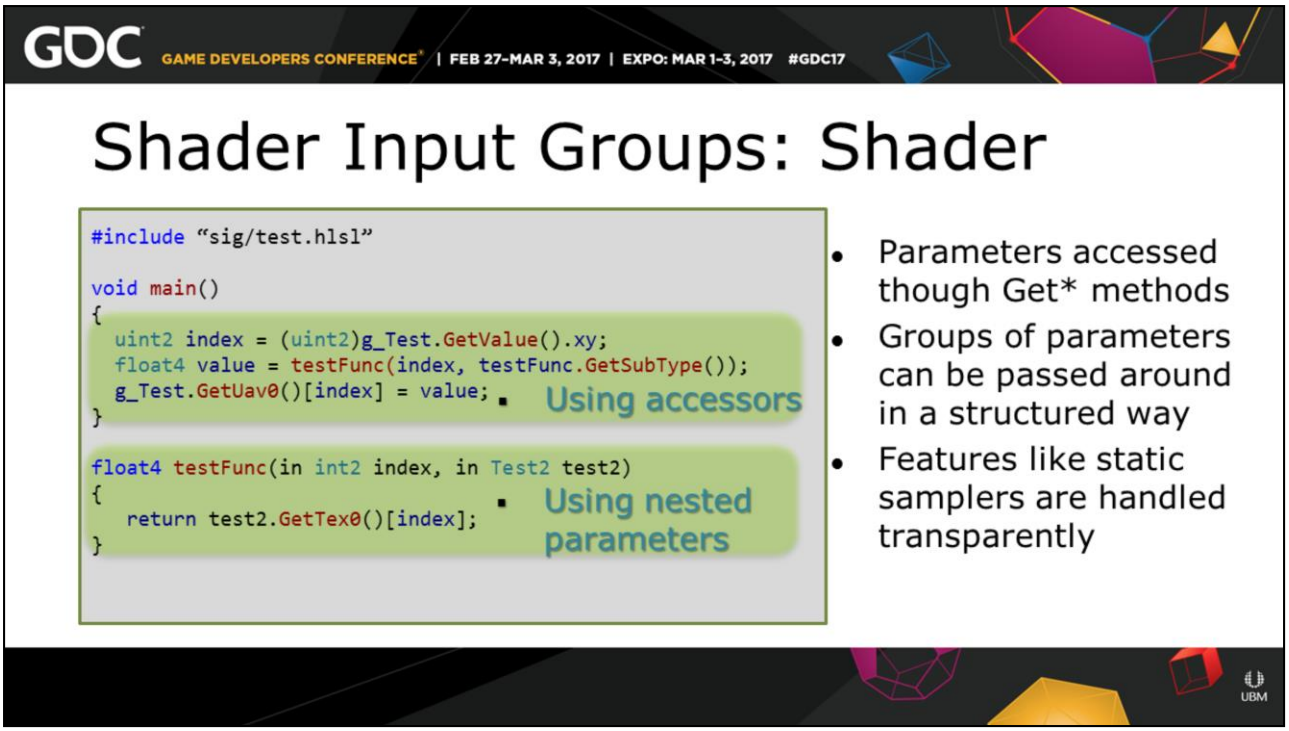
- We use the Set methods to fill the required shader parameters

- Before being able to bind these, we require compiling them into an immutable blob (it is at this point that we issue descriptor copying)
    - Once we have a blob, we can then reuse it multiple times, effectively just passing around descriptor table handles

Shader Input Groups: Shader

```
#include "sig/test.hlsl"

void main()
{
  uint2 index = (uint2)g_Test.GetValue().xy;
  float4 value = testFunc(index, testFunc.GetSubType());
  g_Test.GetUav0()[index] = value;
}

float4 testFunc(in int2 index, in Test2 test2)
{
   return test2.GetTex0()[index];
}
```

- Parameters accessed though Get* methods
- Groups of parameters can be passed around in a structured way
- Features like static samplers are handled transparently

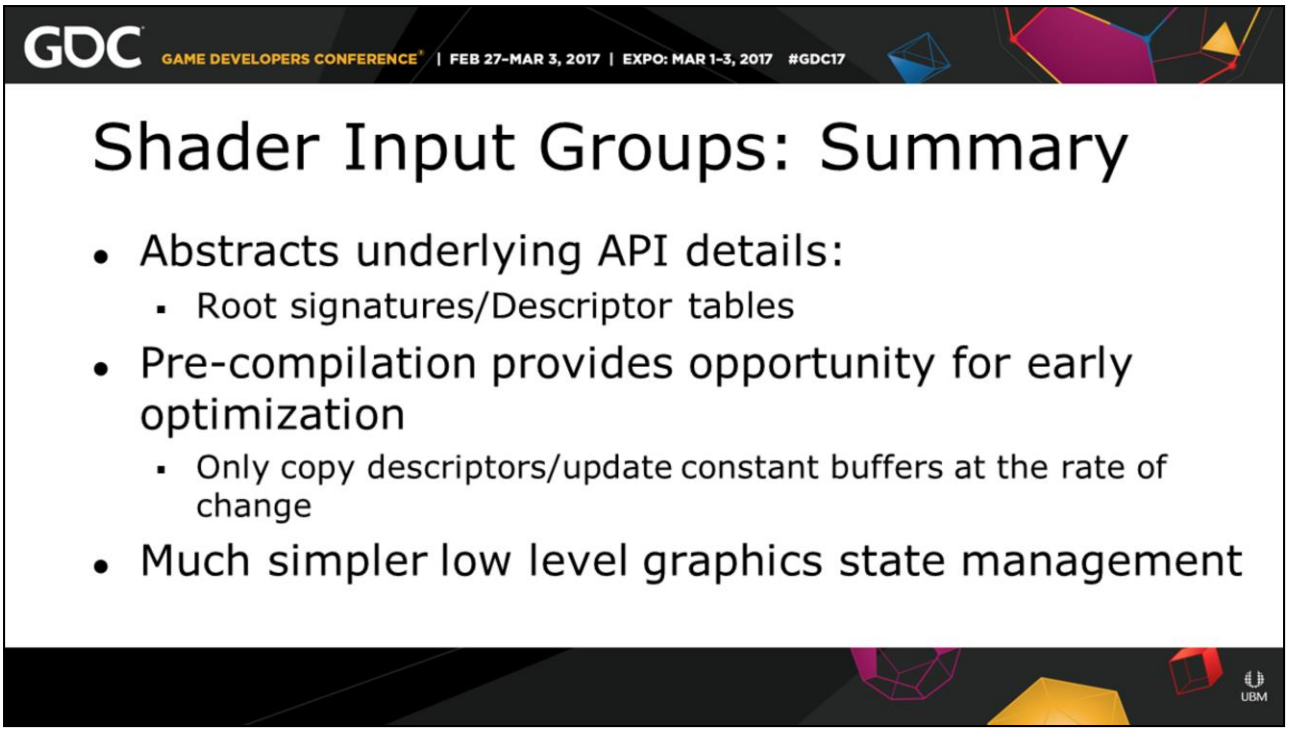


- In terms of shader interface, it's also quite simple:
  - We use the auto generated get methods to access resources and constants
  - The use of nested SIG types, allows us to write shader code headers that can be easily reused without depending on global parameters or passing a long large lists of individual parameters
  - We also support auto-generating loader functions for loading structures of parameters from buffers, which we use in draw instancing code
  - We can also have auto generated GPU debug tracing code based on annotations

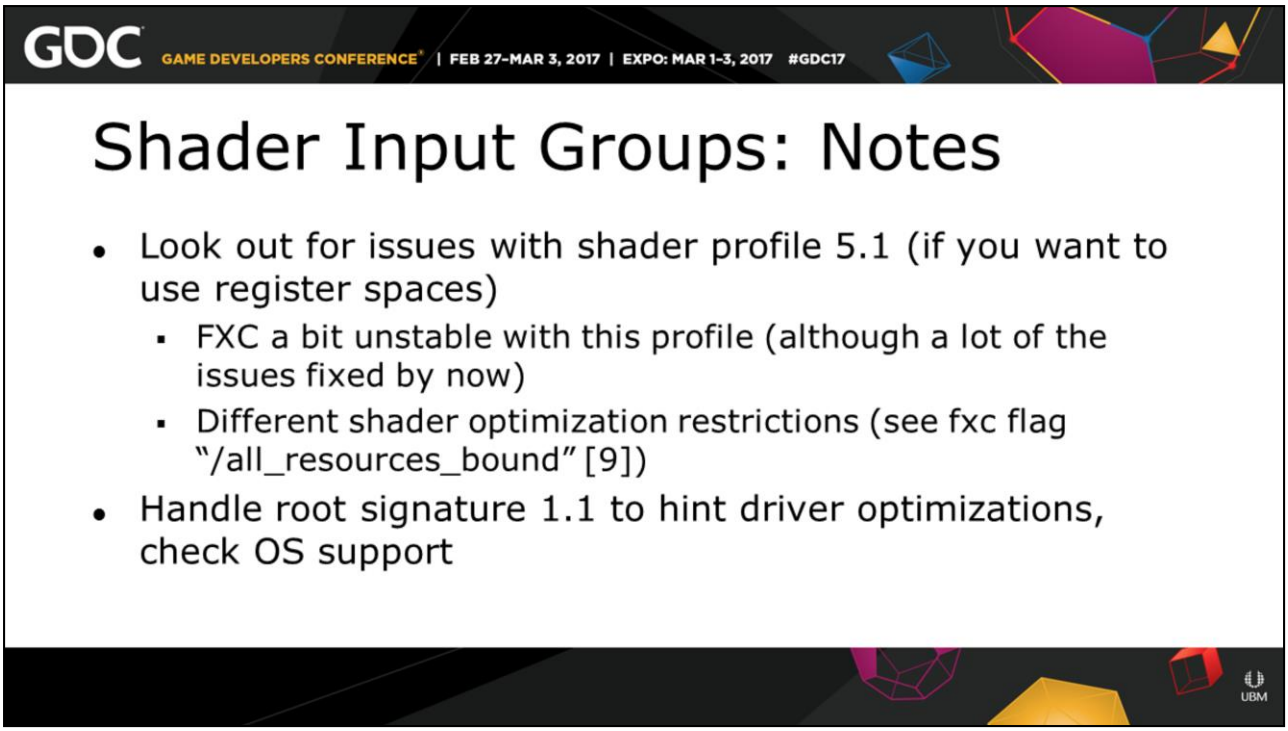
Shader input groups: Stats

	Average value	Notes
Static descriptors loaded #	15000	Copied once at asset load time
Transient descriptors #	5000	Copied every frame (mostly for pass SIGs)
Unique SIG Layouts	10	
Unique SIG definitions	300	

- Here are some stats related to this system:
  - As you can see, most of the shader input parameters are static (these come from material SIGs, whose associated descriptors and constants are copied once during asset load)
  - Most of the transient descriptor copying is not between individual draw calls, but at the start of producers
  - The unique SIG instances actually end up matching up roughly between themselves in term of descriptor table sizes, so we don't have many unique layouts, mostly for very specific rendering like terrain, water, etc.



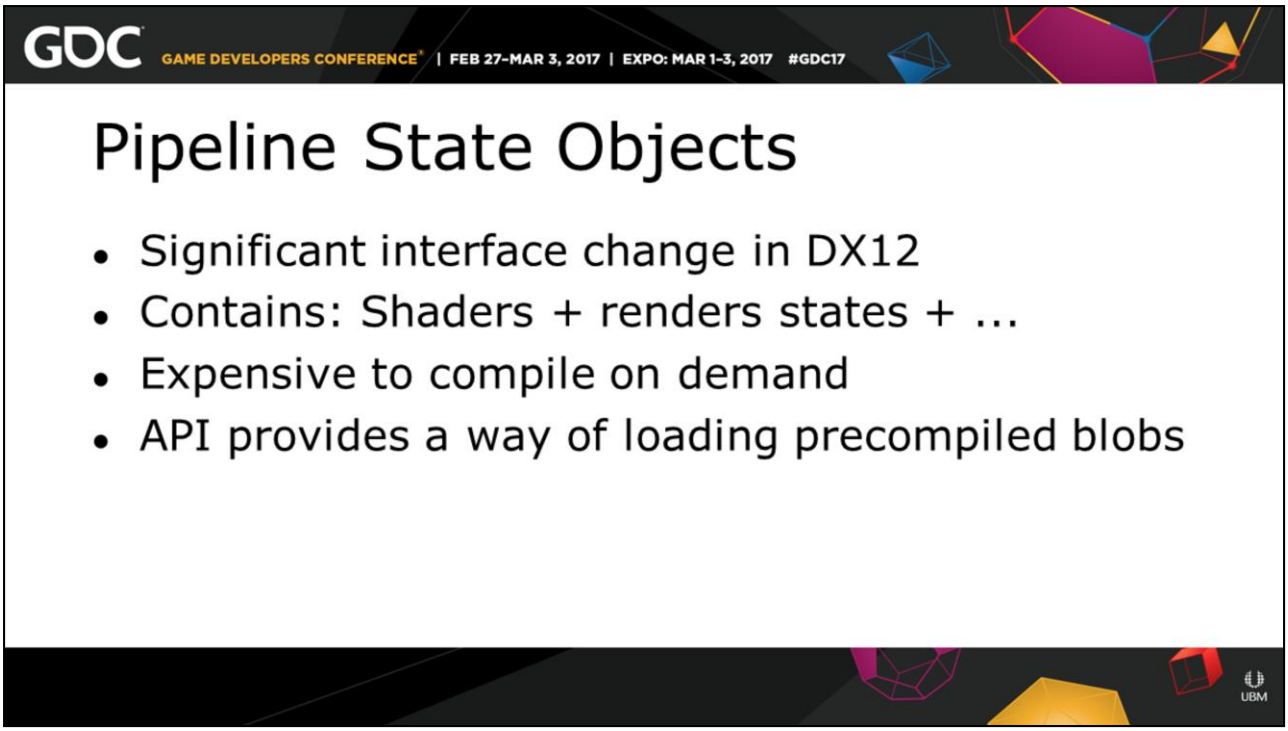
- Summary:
  - Abstracts underlying API details (like root signatures and descriptor tables) but only in a very thin fashion
  - Descriptors are updated at the approximate frequency they change
  - Minimal overhead since the interfaces are closer to how a lot of the HW behaves with pointers to tables instead of individual slots, we also end up with a small amount of bind points (5-6), which means very simple low level graphic state management code
  - Internal knowledge of how descriptors are updated allows us to generate more optimal root sig 1.1 to hint optimizations to the driver and implement internal Tier limitations like null CBV/UAVs



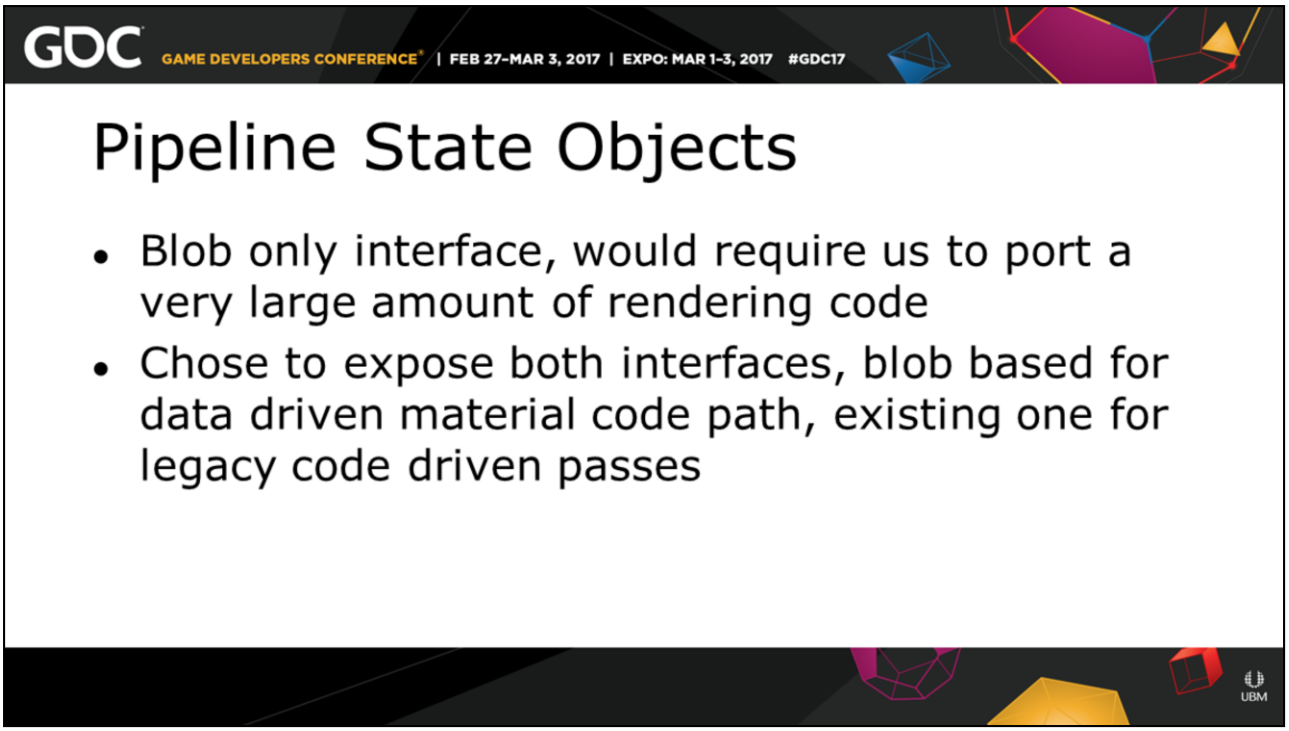
- Transition to shader profile 5.1 was more 'painful' than was anticipated by us (we did it mostly to use register spaces for ease of management of slots in root descriptor tables)
  - We had many compiler issues and crashes (most have been resolved by now), some still remain (like indexing arrays with resources, for example)
  - The different binding model also changes FXC optimization rules, so look into using the "/all_resources_bound" options (see Marcelo's blog about this) to get very similar DXBC to profile 5.0

- Root signature management a bit messy since 1.1 is only supported from win10 RS1, so be careful with managing it if you need to support systems with older Win10 versions

Pipeline State Management



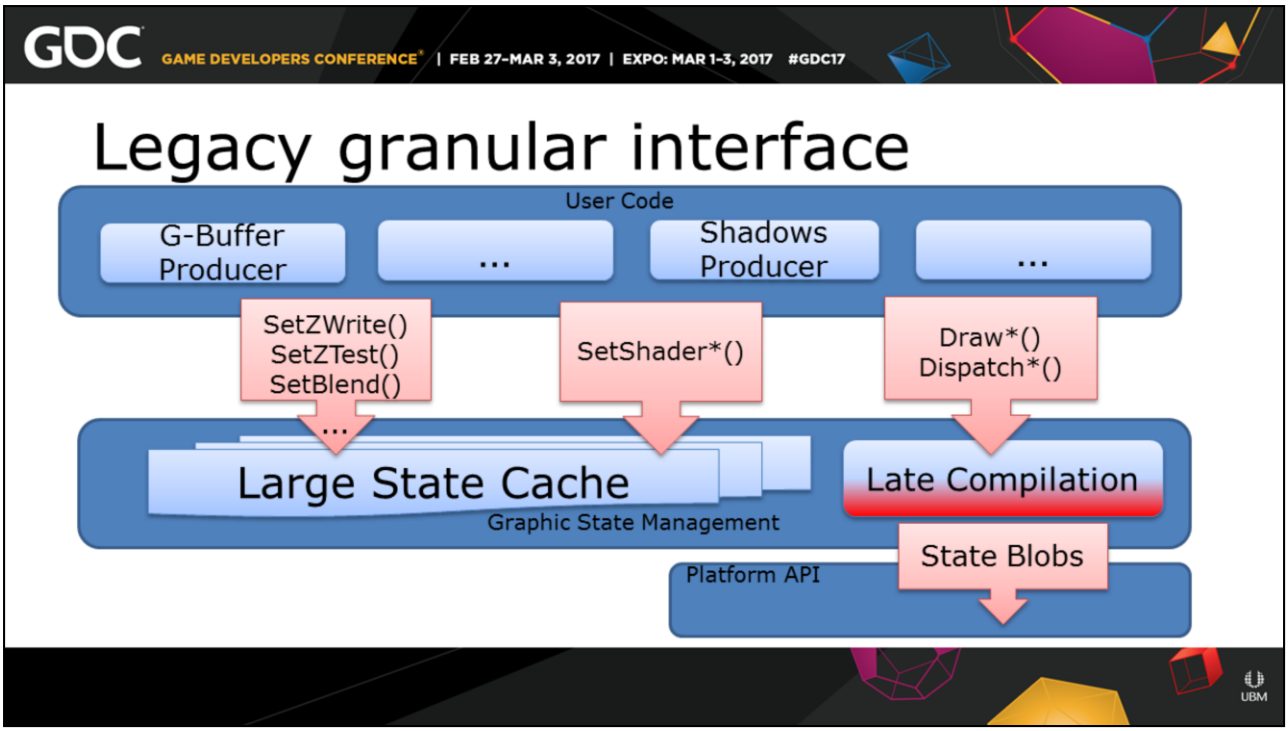
- PSO's are a significant interface change in DX12, most state is now bound via a single blob
- Expensive to compile on demand (could take 100's of ms) since this is were shader compilation can happen
- We can load serialized blobs and drivers implements some internal caching and derived state optimizations
- For us, the render state part of the PSOs was the most problematic one due to the varied sources where they can come from in our engine:
    - CPP code, materials, artist driven FX scripts, etc.



- Precompiled render state + shaders is great when thinking about performance and generating optimized shaders
  - In practice most engines have had a pretty relaxed approach to were render state changes come from, even allowing micro RS changes in artist facing interfaces

- We do however have two very distinct rendering code paths in our engine:
  - Material based rendering, which is data driven, based on precompiled shader permutations (which covers >90% of our draws)
  - Handcrafted HLSL based code for features like deferred lighting, GPU culling, most post effects, etc.

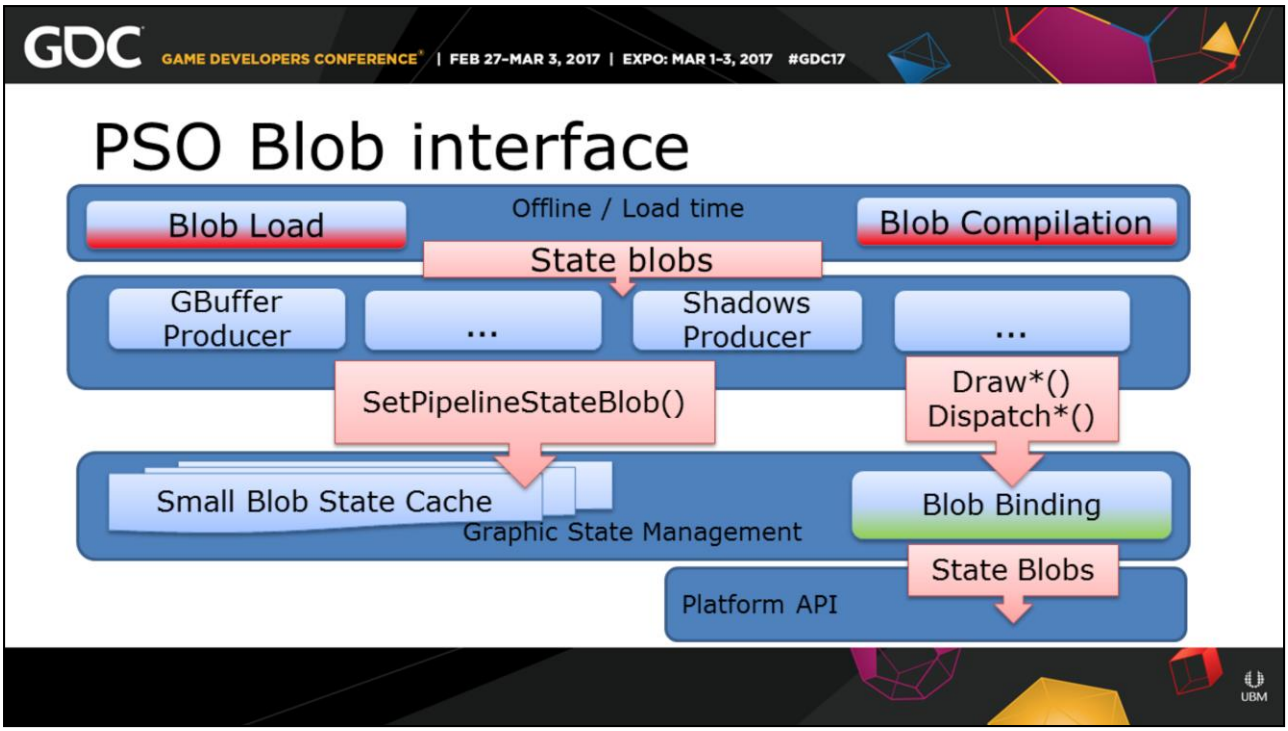
- We chose to maintain the old interface along side the new blob based one, in order to avoid having to port all of the code in one go



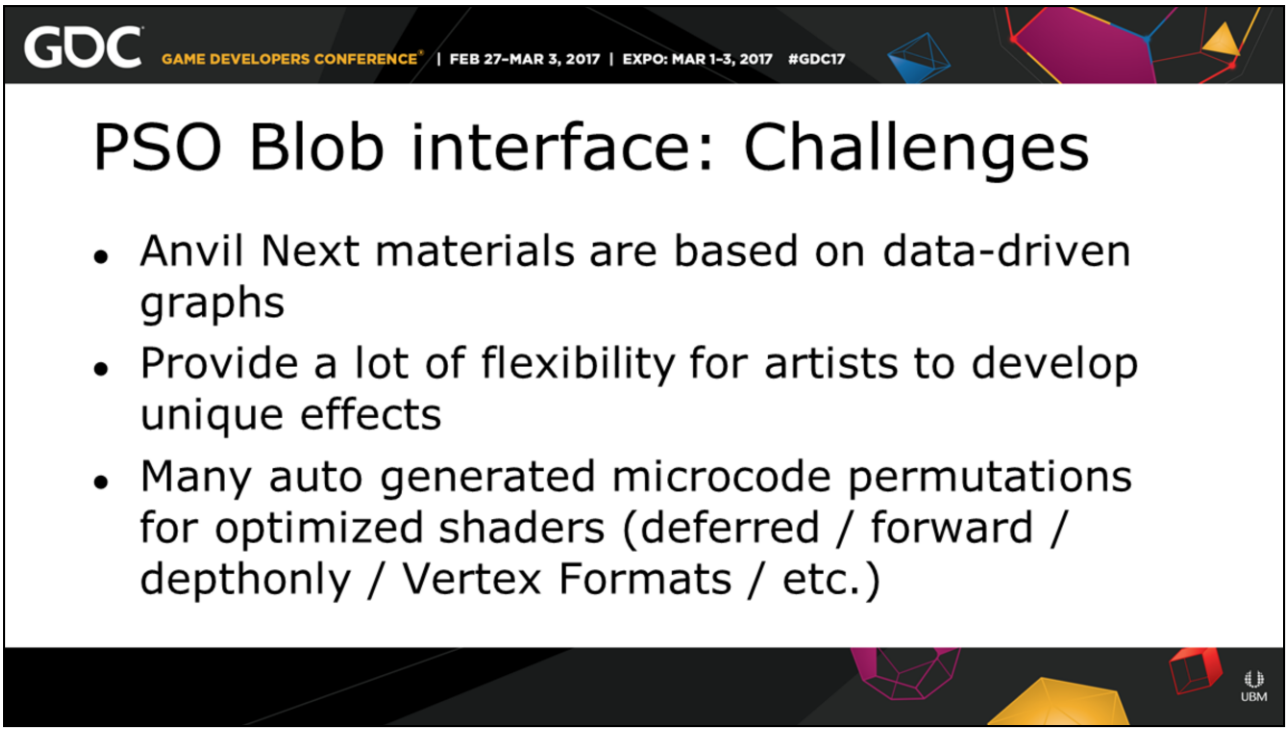
- In a similar fashion to shader parameters, producers issued a lot of very granular state changes
- Internal state compilation was deferred until the last minute when we issued the actual draws/dispatches.
  - It was at this point that we searched for the appropriate pipeline state blobs based on the hashes of the many render states, shaders, etc. or created a new one, if needed, which would cause hitches that would impact rendering threads and thus potentially framerate

PSO Blob interface

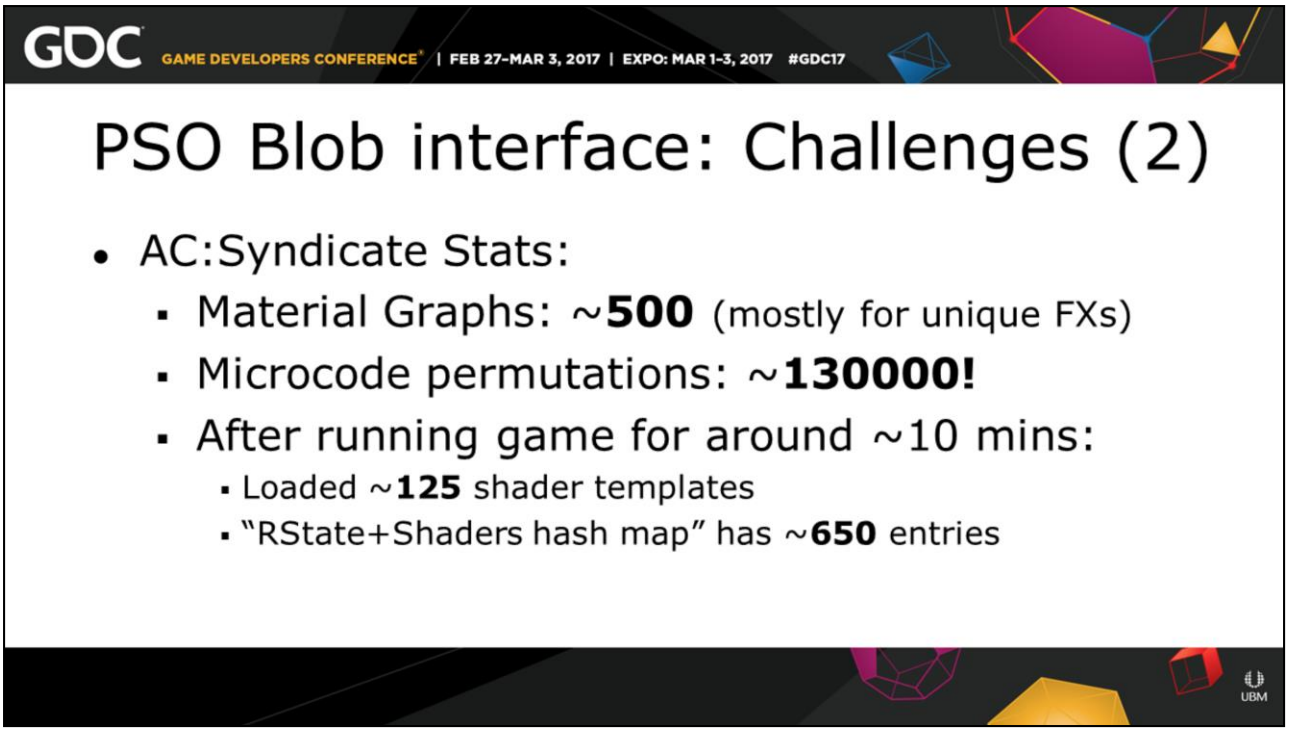
- Require binding of precompiled groups of state
- Restrict independent state changes that the users can do by using state presets
- Open the opportunity for load time/Offline blob compile time optimizations



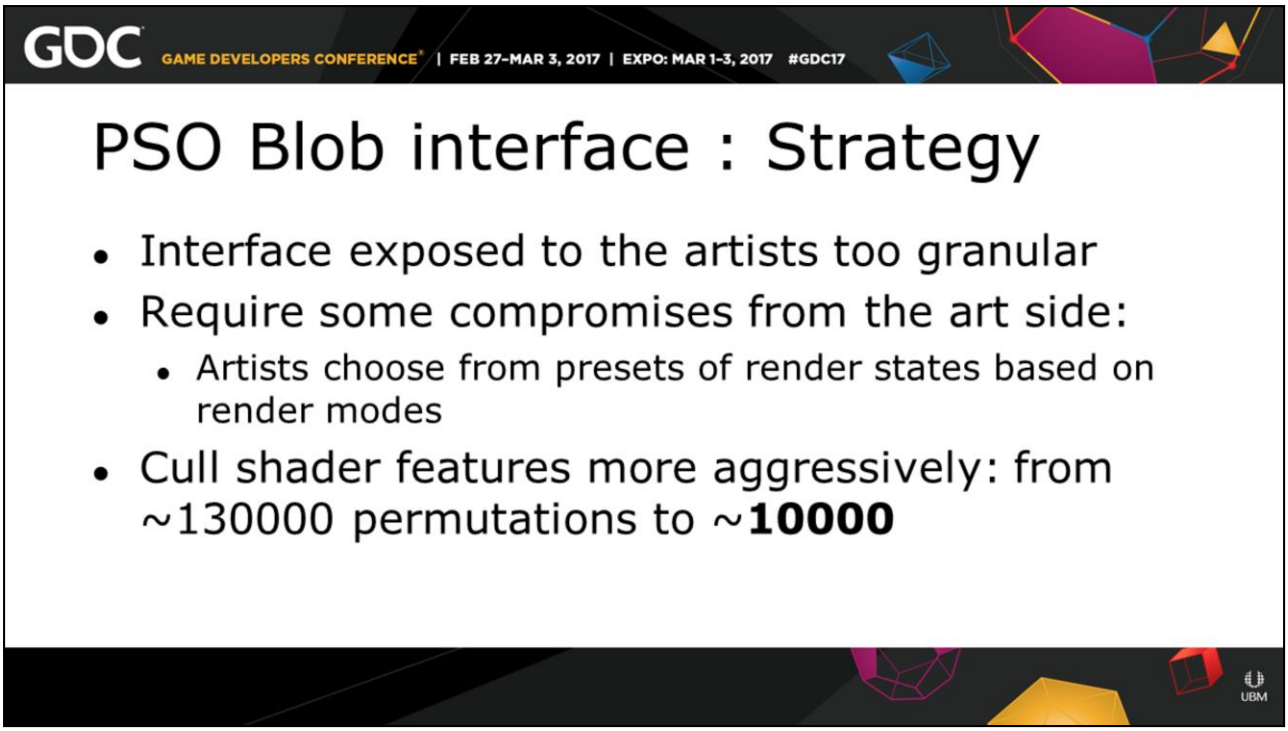
- With the blob approach:
  - blobs of state are either compiled offline (material rendering code path case), or at load time (in loading threads)
  - we have a much simpler state setting interface: SetPipelineStateBlob (+render target/viewport, other misc)
  - this results into much simpler state caches (essentially managing a couple of pointers to these blobs) and much lower overhead at Draw/Dispatch time (no hitching in CMD list recording threads)



- Here we also had some initial challenges:
  - Data shaders are based on data graphs that generate a lot of permutations, to support vertex formats, mesh options (instancing/clustering modes), optimized shaders for the several render passes (GBuffer, Forward, depth only, etc.)
  - We precompile all shader permutations upfront, at data baking time



- To give some perspective (stats from AC:Syndicate)
  - we had ~500 material shader graphs
  - most of these materials are for FXs shaders, used in very specific cut scenes/missions
- We can see we had a lot of permutations that are very rarely used (or actually never used during a game session)
- We rely on microcode pre-compilation:
  - So we get a well defined set of data, no corner cases of needing a permutation that wasn't generated
  - However as we can see, if left unchecked, the permutation count can become difficult to manage …



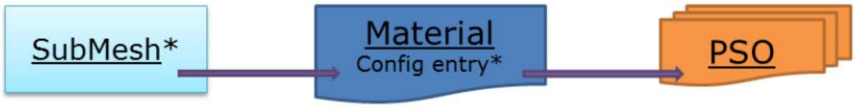
- The very first step we took, was to tackle the number of shader permutations
- Traditionally in Anvil Next artists Could:
  - specify individual render states per material
  - toggle render states on materials at runtime via the FXs system

- Introduced restrictions:
  - only allow use of preset groups of render states and permutation features
  - remove the ability to toggle individual render states at runtime, allow them to swap materials instead

- This helped reduce the number of permutations per material template significantly

PSO Blob interface : strategy (2)

- At runtime we already have the ideal place to point to our PSOs: Materials



- Associate PSO in loading thread (disk load/compiled derived)
- Rendering just needs to index PSOs based on rendering mode

- After getting the shader permutation side a bit more under control we moved to precompiling the several pipeline states in a similar fashion to how we did shader microcode

- In Material instances we previously stored references to shader microcode, lists of render states, etc. which were applied individually via the old granular interfaces, now we just need PSO references

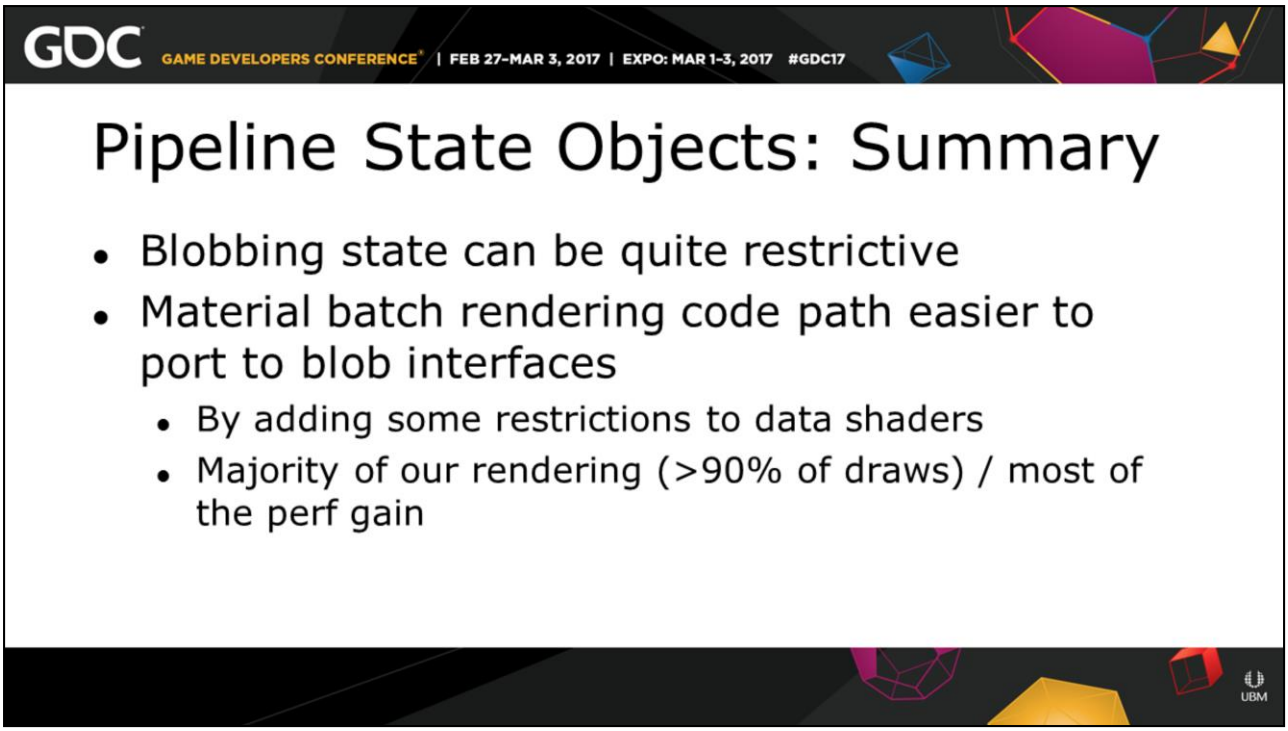
- Because all of these permutations are known upfront we can pre-associate them to the meshes at load time and at render time, do a simple indirection based on the current render mode

- This effectively means we have a number of PSO databases, one for each material graph, whose entries are indexed and cached at load time

Pipeline state objects: Stats

	Average per frame value	Notes
Material PSO bind (fast path)	2000	Blobs cached in loading thread
Non native PSO bind (legacy path)	200	Slowly porting remaining code to native PSO path
CPU saving in Material::Bind()	~40%	Most of the code here was doing granular render state setting

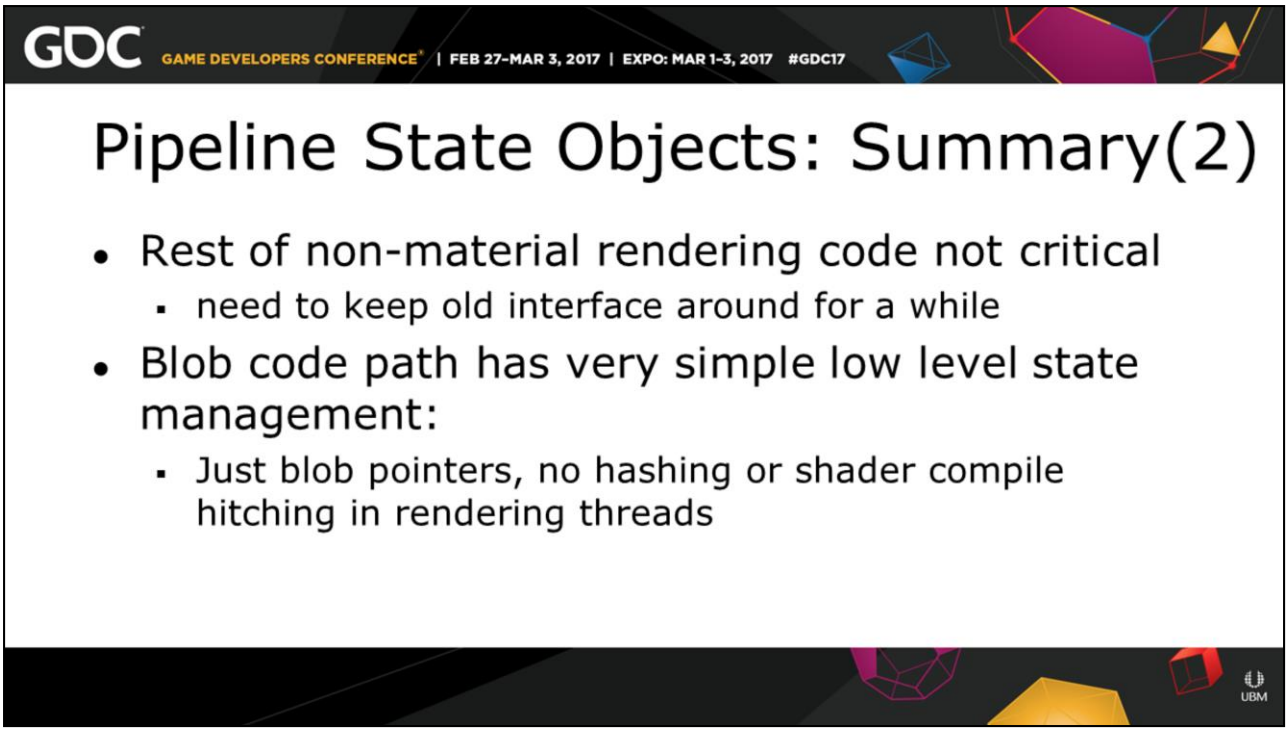
- Here are some stats related to PSOs:
  - As you can see the bulk of our rendering uses the precompiled PSO code path
  - Still have a bit of use of the old code path, but it also uses a PSO cache that can be warmed up to prevent hitching. We are slowly porting some of these passes to using the blob interfaces
- Finally you can see some good savings on material apply, since the majority of work there was related to setting render state



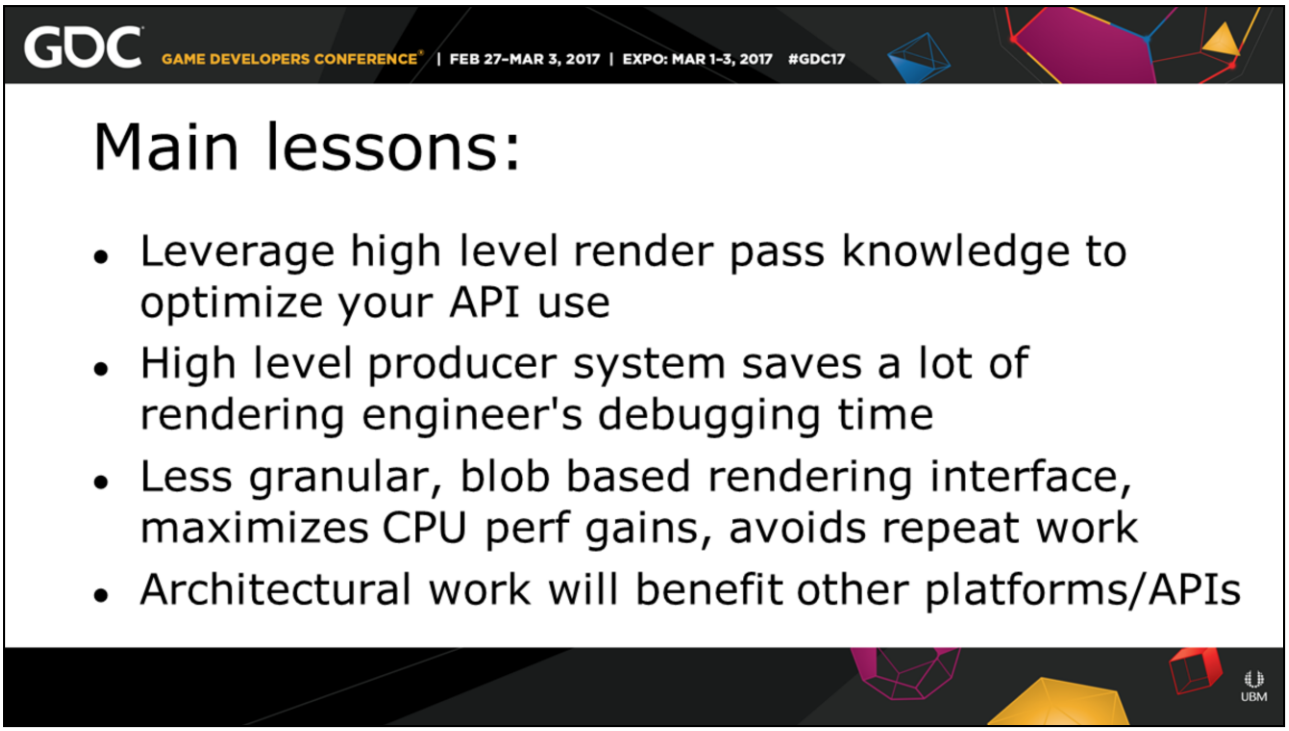
- Blobbing can be restrictive. Some of the more problematic cases:
  - Using custom depth bias (in cut scenes there was a lot of tweaking being done to get around self shadowing issues for example) – chose to move it to being done in the testing shaders instead
  - No more runtime toggle of render states by effects, they now have to switch materials instead
  - Debug modes (wireframe, picking, etc.) can also create a lot of PSO permutations, but this is only a concern for non release builds

- The bulk of our CPU rendering related cost is in the Material+Mesh code path, also the majority of the perf gains, the remaining code can be more progressively ported to the new blob interfaces

- By having knowledge of which permutations are available we can even compiled them offline in the users machines (at install time and when we detect driver changes, etc. and regenerate our PSO caches)



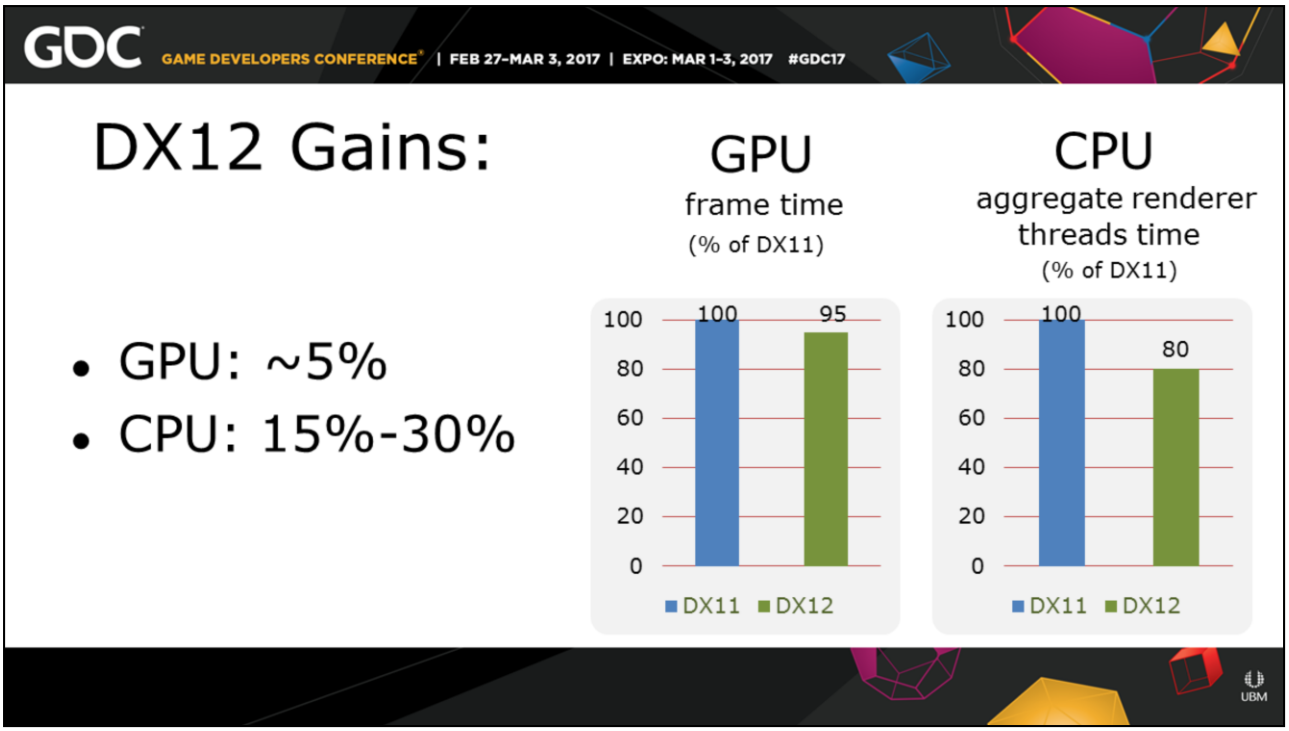
- We end up with very simple low level state management, very close to the ideal that DX12 interface requires
- However, imposing restrictions on users who have used the engine for many years is generally not very popular



- Main lessons from our experience:
  - Invest in implementing systems that can leverage high level render pass knowledge to optimize your API use
  - These systems pay off in terms of rendering engineer's time:
    - by facilitating debugging with the aid of rich visualization tools
    - decoupling your render passes and thus making them more modular and making code cleaner, easier to understand, debug and reuse
  - A coarser rendering user interface goes in the direction of what the new graphic APIs expect, forces pre-batching of state and results in much more simplified runtime graphic state management

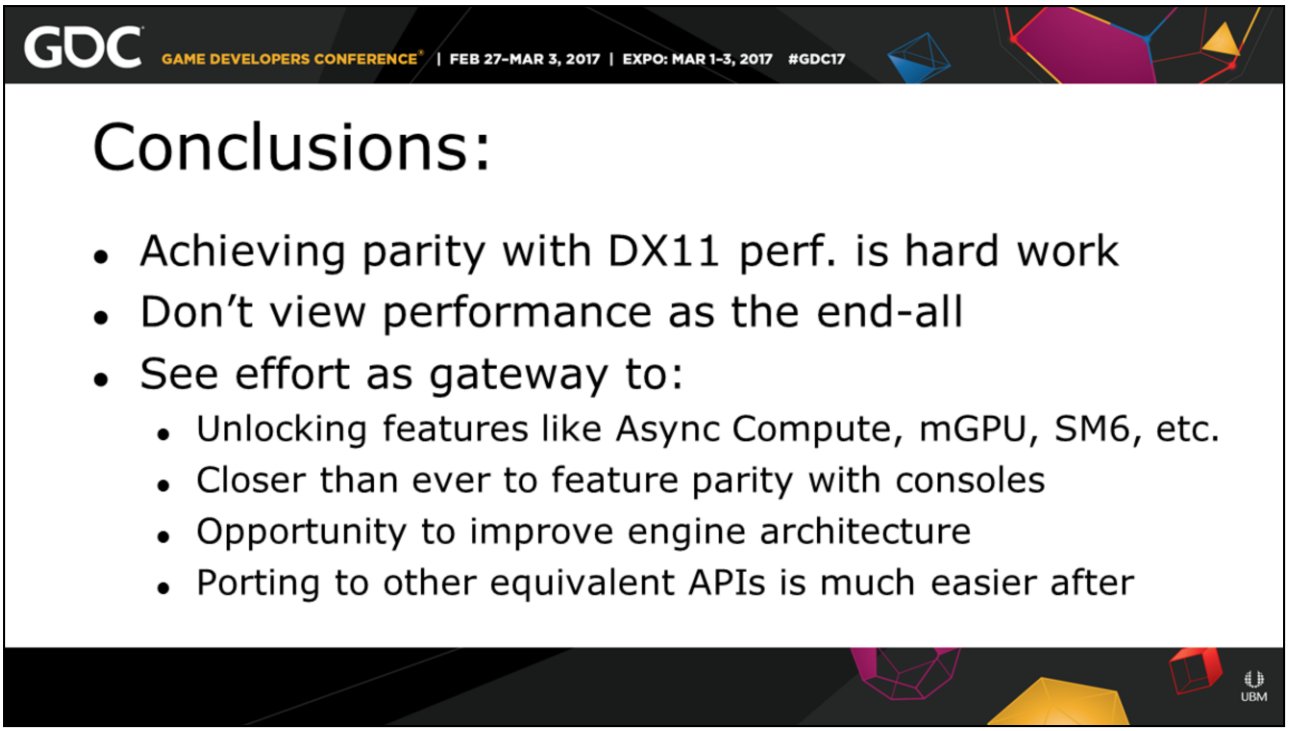
- Pre-compilation of states allows for early (offline/load time) optimizations and avoid repeat work, also allows us to shift potentially expensive work to loading threads

- Finally, a lot of this high level work will make it much easier to port to other similar graphic APIs, and even benefit old ones, mainly because it forced us to think about ways of minimizing and group state changes



- Here are some high level stats from our current DX12 version vs the DX11 one:
  - Current GPU gains are small (~5% faster), matching the DX11 driver was actually quite a bit of work, most gains here were mainly due to async. compute on some IHVs. (I have to note that some parts of our async. code done on consoles, like async. compute instance and triangle culling, don't port to PC very well due to the lack of granular low overhead cross-queue synchronization primitives. This results in still larger than ideal number of barriers within producers of the graphics queue. We are doing some work to try to overcome this, by refactoring the code so it's less dependent on granular synchronization at the expense of using more memory).

- Regarding CPU, here is where DX12 shows better gains from: 15%-30% on our render tasks
  - these vary quite a bit depending on the ratio of the work we do in the runtime draw pre-batching vs the number of API calls (the more batching we do on our side, the less gains we see)
  - I haven't included here the performance we gain by not having other UMD threads we have in DX11



- In conclusion:
  - If you take the narrow view of only caring about raw performance, you probably won't be satisfied with the amount of resources and effort it takes to get to even just performance parity with DX11

- I think you should look at it from a broader perspective, see it as gateway to:
  - Unlock access the new exposed features (async. compute, multi GPU, shader model 6, etc.)
  - More or less unify the feature set with consoles
  - Opportunity to do some positive architectural changes in your engine
  - Do the bulk of the ground work to port to other APIs like Vulkan

Acknowledgements

- Michel Bouchard (Ubisoft Montreal)
- Ulrich Haar (Ubisoft Montreal)
- Vincent Veilleux Gaboury (Ubisoft Montreal)
- Andrei Tatarinov (NVIDIA)
- The whole 3D team at Ubi Montreal and the many others that helped from the several Ubi studios (KIEV, QUE, SIN, SOF)

I would like to thank the following people who contributed to this presentation or helped in the design and implementation of the systems I presented.

Questions?

- Contact: tiago.rodrigues@ubisoft.com

Misc tools notes:

- It pays to add your own tools for high level analysis (graphs to visualize dependencies, resource transitions, etc.)
- The DX12 Debug layer is also quite helpful for tracking transition issues, etc. GPU validation mode also very useful.
- If you loved PIX for XB1, you now have **PIX for windows** which is already great and maturing very fast!
- **RenderDoc** works great, open source, help make it better yourself!
- IHV tools getting better on DX12 support
- **GPUView**: we all love to hate it, but probably the only tool that can help you with certain issues (ex: page fault tracking). Inject your own ETW for extra help.

References:

- [1] GDC2016 – "Practical DirectX 12" – Gareth Thomas & Alex Dunn
- [2] GDC2016 – "Right on Queue" – Stephan Hodes, Dan Baker, Dave Oldcorn
- [3] GDC2016 – "D3D12 and Vulkan: Lessons learned – Matthaus G. Chajdas
- [4] GDC2015 – "DirectX 12: Improving Performance in your game" – Bennett Sorbo
- [5] GDC2015 – "DirectX 12: Advanced Graphics and Performance" – Max McMullen
- [6] GDC2015 – "Getting the best out of D3D12" – Evan Hart, Dave Oldcorn
- [7] "D3D 12 – A new meaning for efficiency and performance – Dace Oldcorn, Stephan Hodes, Max Mcmullen, Dan Baker
- [8] https://developer.nvidia.com/dx12-dos-and-donts
- [9] https://blogs.msdn.microsoft.com/marcelolr/2016/08/19/understanding-all_resources_bound-in-hlsl/
- [10] GDC2015 - GPU-Driven Rendering Pipelines - Ulrich Haar, Sebastian Aaltonen