GPU Work Graphs

SIGGRAPH 2025 Course

Bastian Kuth Max Oberberger Quirin Meyer

1

Welcome to our GPU Work Graphs Course here at SIGGRAPH 2025 in Vancouver.

GPU Work Graphs – Instructors

Bastian Kuth

PhD Student

Coburg University

Max Oberberger

MTS Software Engineer

AMD

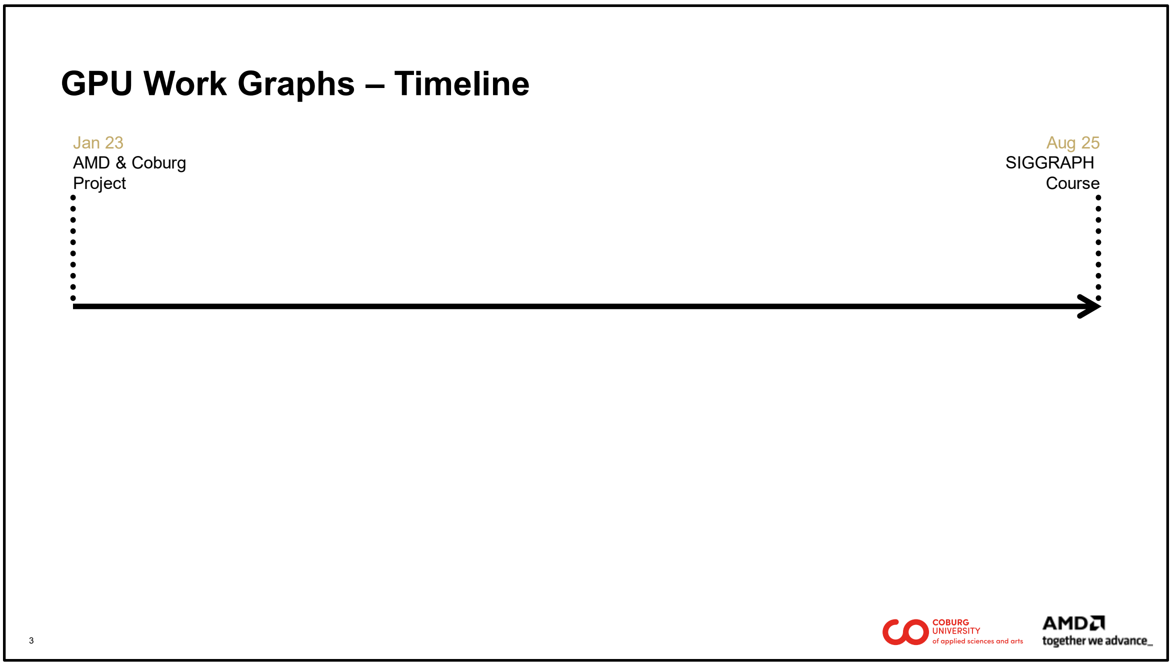
Quirin Meyer

Computer Graphics Professor

Coburg University

2

Before we start, allow us to introduce ourselves. We are Bastian Kuth from Coburg University, Max Oberberger from AMD, and I am Quirin Meyer, also from Coburg University.



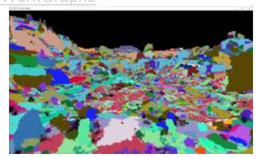
We are teaching a course here today at SIGGRAPH on Work Graphs.

Coburg University and AMD have been jointly focusing on the practical exploration of Work Graphs since January 2023 with funding from the state of Bavaria.

In the last two and a half years, Work Graphs has been dominating our work life, and we would like to tell you briefly what we and others have done so far with this new technology.

GPU Work Graphs – Timeline

- Work Graphs Preview
  https://gpuopen.com/learn/gpu-work-graphs/gpu-work-graphs-intro/
- Sample Code
  https://github.com/GPUOpen-LibrariesAndSDKs/WorkGraphsHelloWorkGraphs
- Vulkan Support
  https://gpuopen.com/gpu-work-graphs-in-vulkan/
- Work Graph Samples
  https://gpuopen.com/learn/rgp-work-graphs/
  https://gpuopen.com/learn/work_graphs_learning_sample/

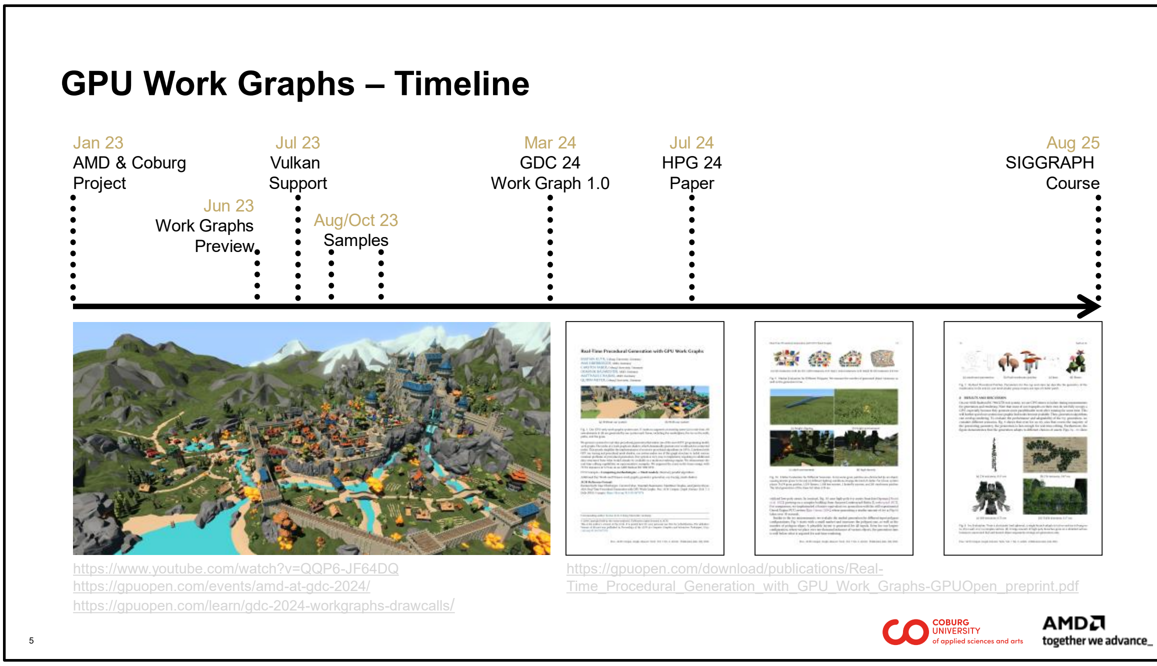


4

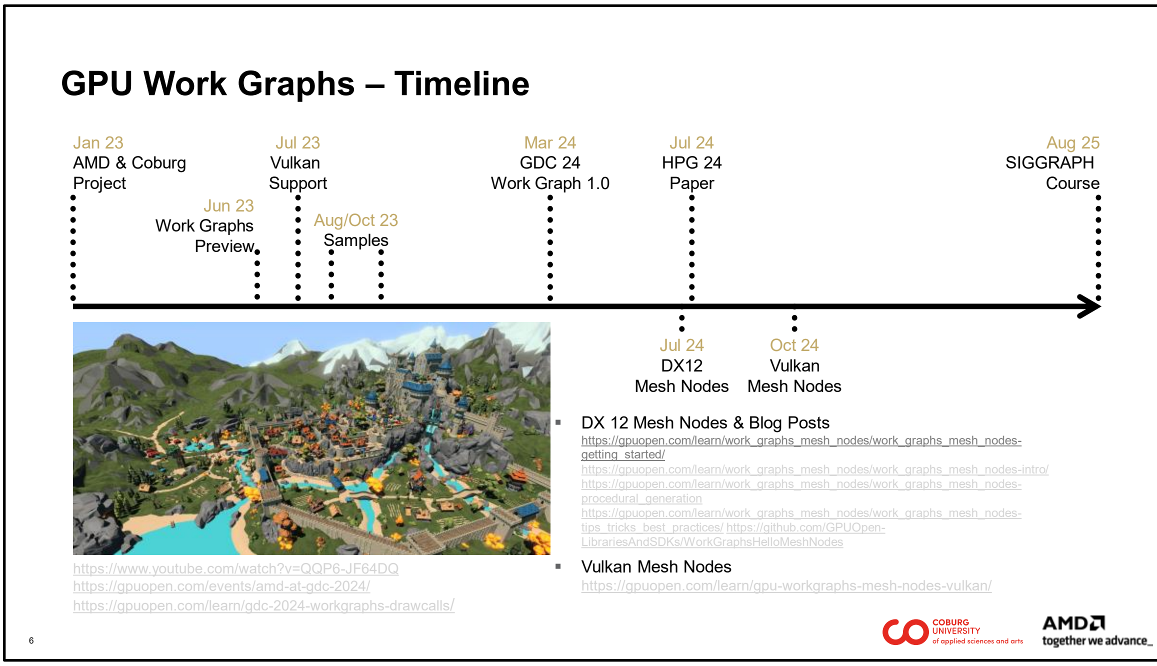
COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

While we were conducting our research, Work Graphs occurred as preview with sample code, Vulkan support was added, and AMD published multiple Work Graphs samples.



At the Game Developer Conference (GDC) 2024, we presented our first demo using Work Graphs. We published our research results at High Performance Graphics (HPG) 2024.



Our research makes use of Mesh Nodes, which were made available in the third quarter of 2024. We also wrote several blog posts teaching about work graphs and mesh nodes.

You can find a video of our demo, which highlights some of the benefits of Work Graphs here: https://gpuopen.com/learn/gdc-2024-workgraphs-drawcalls/

GPU Work Graphs – Timeline

Jan 23
AMD & Coburg Project

Jun 23
Work Graphs Preview

Jul 23
Vulkan Support

Aug/Oct 23
Samples

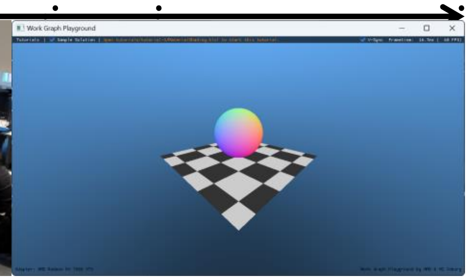
Mar 24
GDC 24 Work Graph 1.0

Jul 24
HPG 24 Paper

Nov 24
GPC Master Class

Aug 25
SIGGRAPH Course

https://gpuopen.com/gpc-2024/



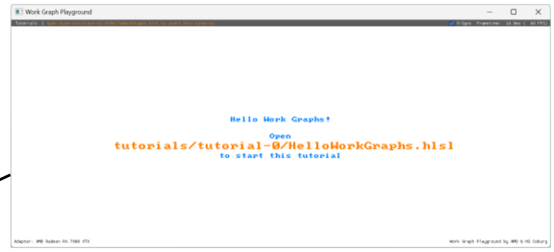
Work Graph Playground App
https://gpuopen.com/learn/work-graph-playground/

7

Our GDC demo and our HPG paper raised some excitement, so we got invited to teach a Master Class at the Graphics Programming Conference in Breda, Netherlands in 2024. This is where we released our Work Graph Playground App for the first time. We are going to use this app in this course, too. In case you brought your laptop, you can join us experimenting with our Work Graph Playground App.

Work Graph Playground App

1. Go to: https://wgpa.short.gy/
   github.com/GPUOpen-LibrariesAndSDKs/WorkGraphPlayground
2. Download the Latest Release
3. Unzip `WorkGraphsPlayground.zip`
4. Open Folder `WorkGraphsPlayground`
5. Run `WorkGraphPlayground.exe`
6. Optional: `DownloadWarpAdapter.bat`
   If you need Software Emulation



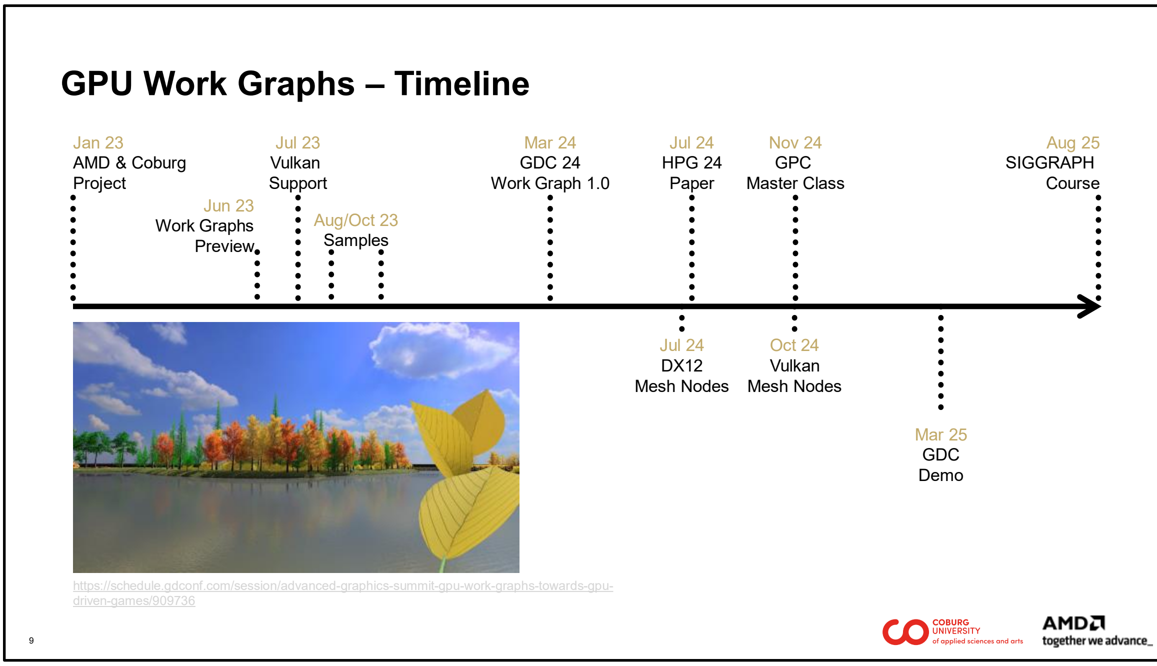

```
Testing adapter "Microsoft Basic Render Driver": Failed
to create D3D12 device.
WARP adapter does not support D3D feature level 12.2
and work graphs.
 See readme.md#running-on-gpus-without-work-graphs-
support for instructions on installing latest WARP
adapter or run DownloadWarpAdapter.bat if you are using
pre-built binaries.
No device with work graphs support was found.
```

8

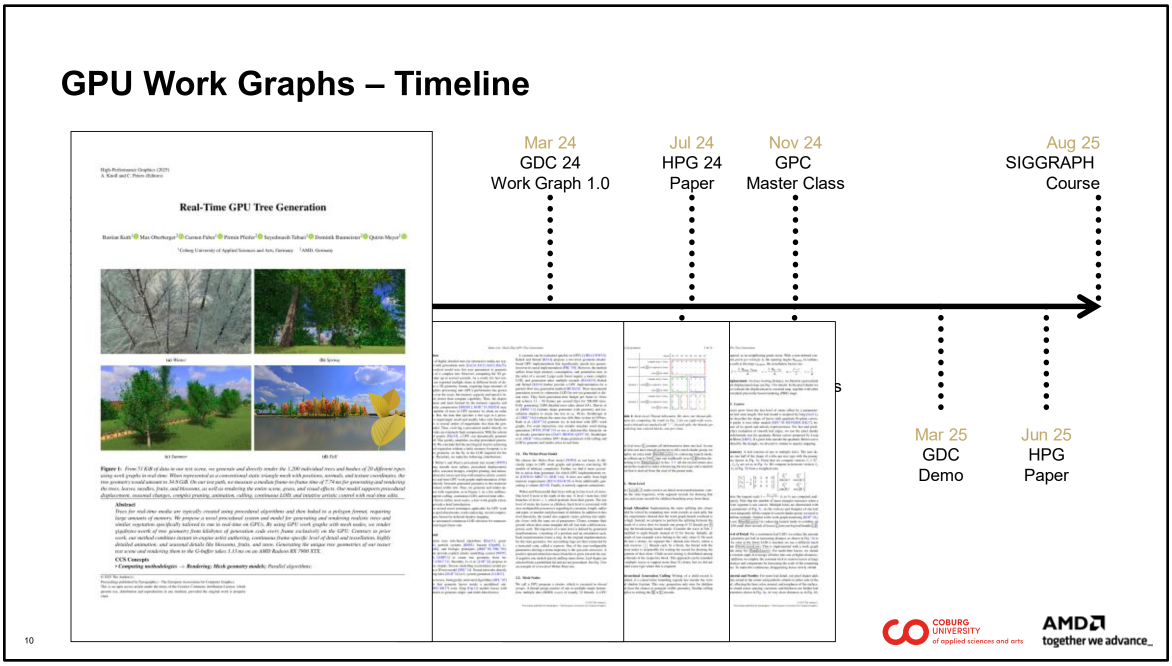
To install the app as a binary, follow these steps. We encourage you to do this right away. In ca. half an hour, you are invited to actively use it.

If your GPU does not support Work Graphs, use the WARP (i.e., software emulation) adapter. Use the DownloadWarpAdapter.bat batch script to download the corresponding DLL.

You can build it from source, too, by following the instruction in the GitHub repository.

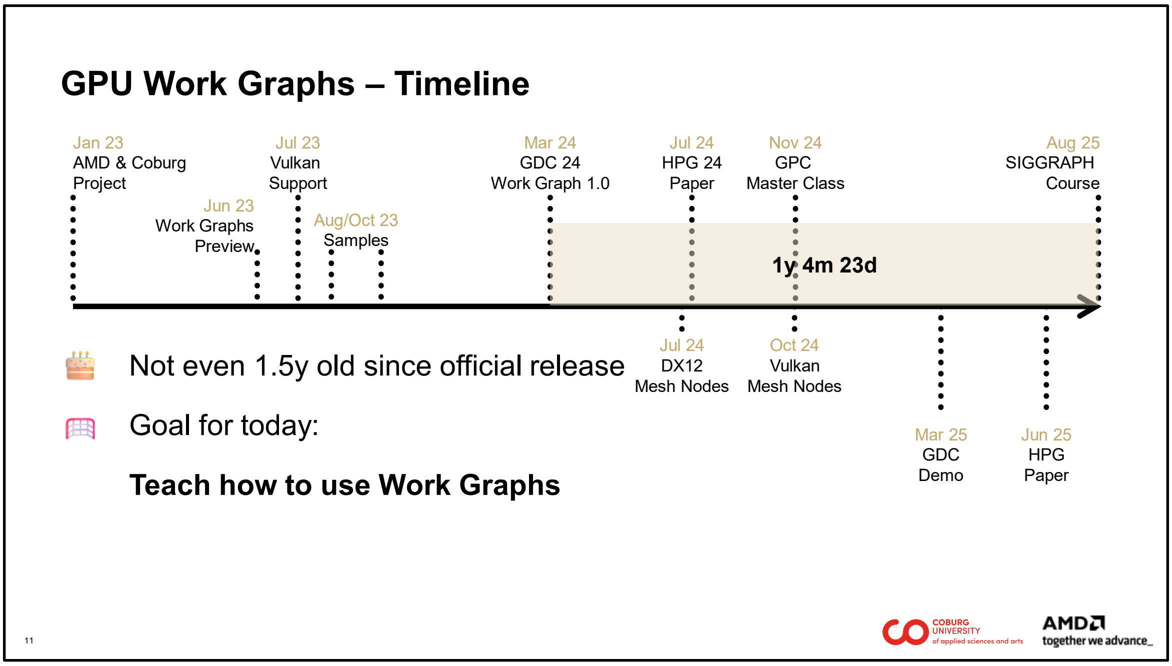


Besides the app, we created a demo for GDC 2025, where we generated vegetation directly on the GPU with Work Graphs.



Our demo is full of research findings that we were able to share at HPG 2025 just a couple of weeks ago.

You can watch a recording of Bastian's talk at HPG here:
https://www.youtube.com/watch?v=SPWDLMc-9h4&t=26050s
The full paper is available here:
https://diglib.eg.org/bitstream/handle/10.2312/hpg20251168/hpg20251168.pdf



In June, we celebrated two years of work graphs when including the preview phase. The official announcement of Work Graphs dates back only less than one and a half year. So, it is a rather new technology.

Our goal in this course is that we teach you how to use Work Graphs and that you can use it for your own applications.

GPU Work Graphs – Course Agenda

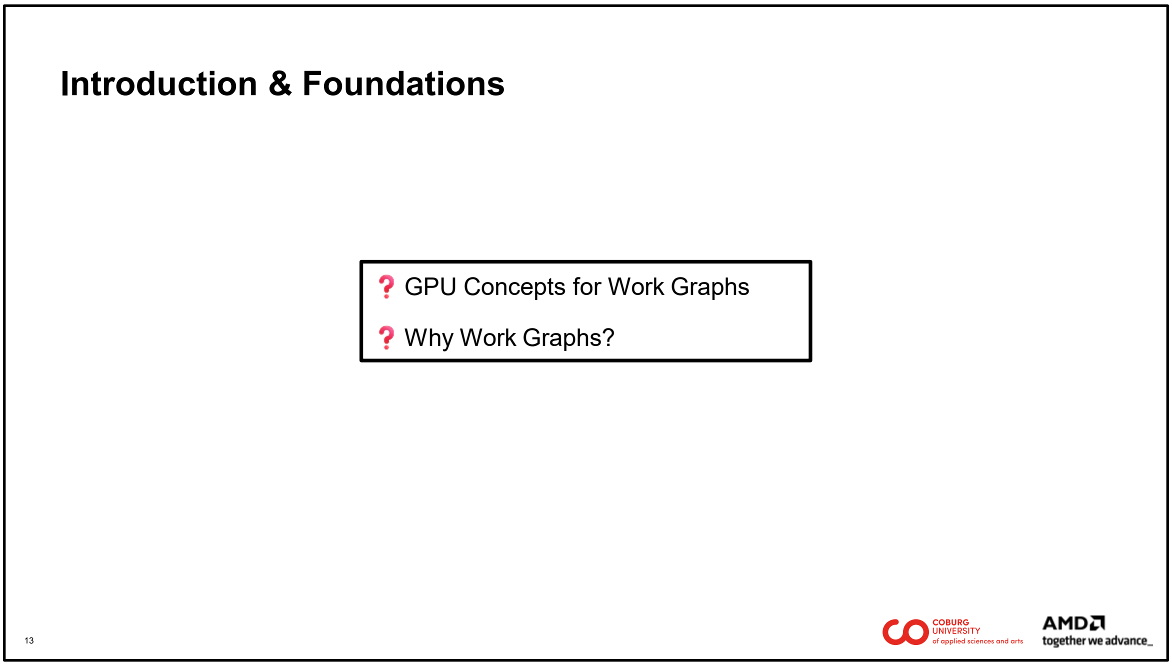
Introduction & Foundations	14:00 – 14:30
Concepts	14:30 – 15:30
Nodes	
Records	
Launches	
Break	15:30 – 15:45
Advanced Work Graphs	15:45 – 16:45
Material Shading	
Recursion & Synchronization	
Procedural Generation	
Under the hood	
Wrap-Up	16:45 – 17:00

12

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

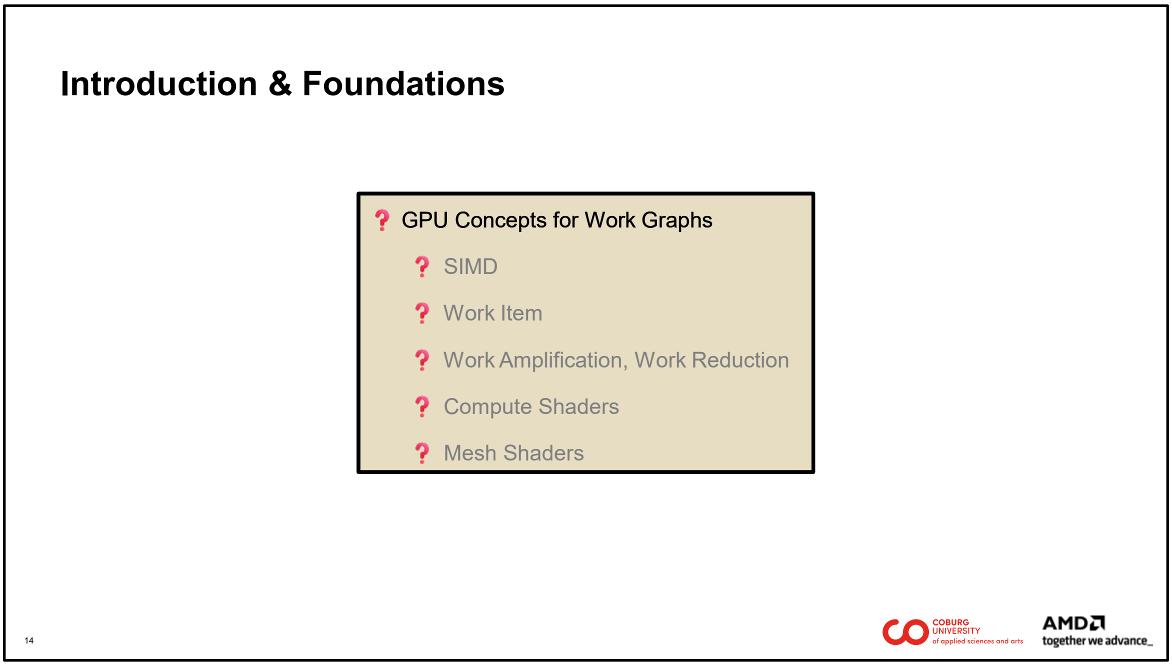
Here is a brief overview of the topics that we will cover today.



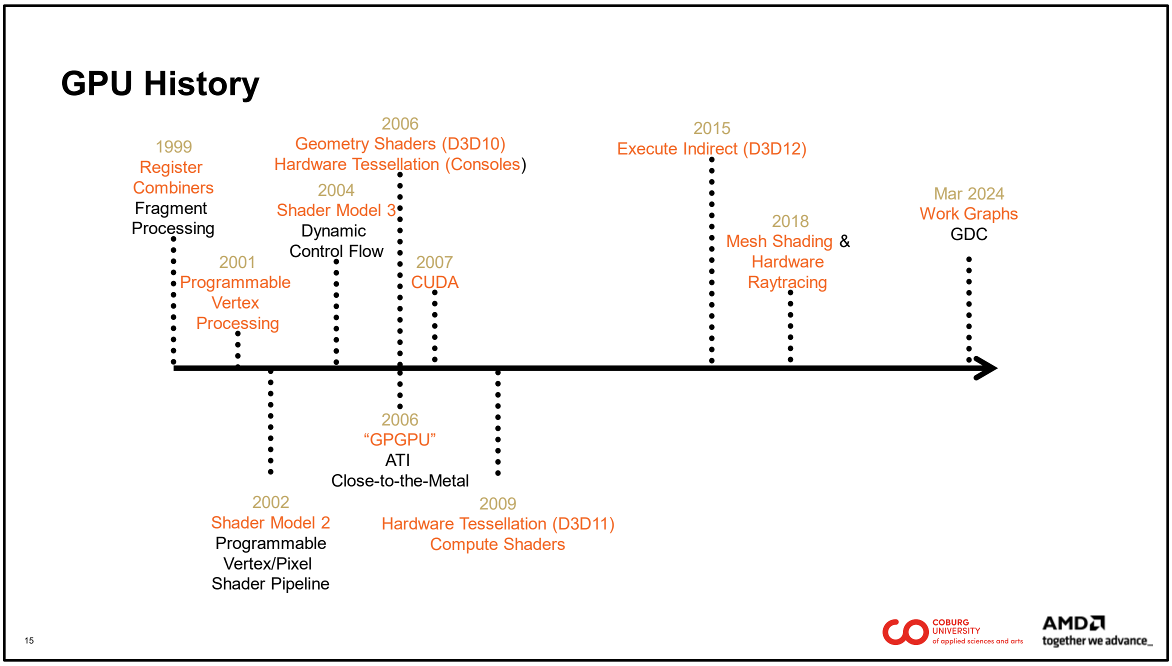
You have just seen some applications demonstrating the power of Work Graphs. Before going into details, we want to first answer the main question:

Why even Work Graphs?

That comes with questions concerning alternative approaches and why you should prefer Work Graphs over them. But before that, we provide a summary of GPU concepts that are important for Work Graphs.



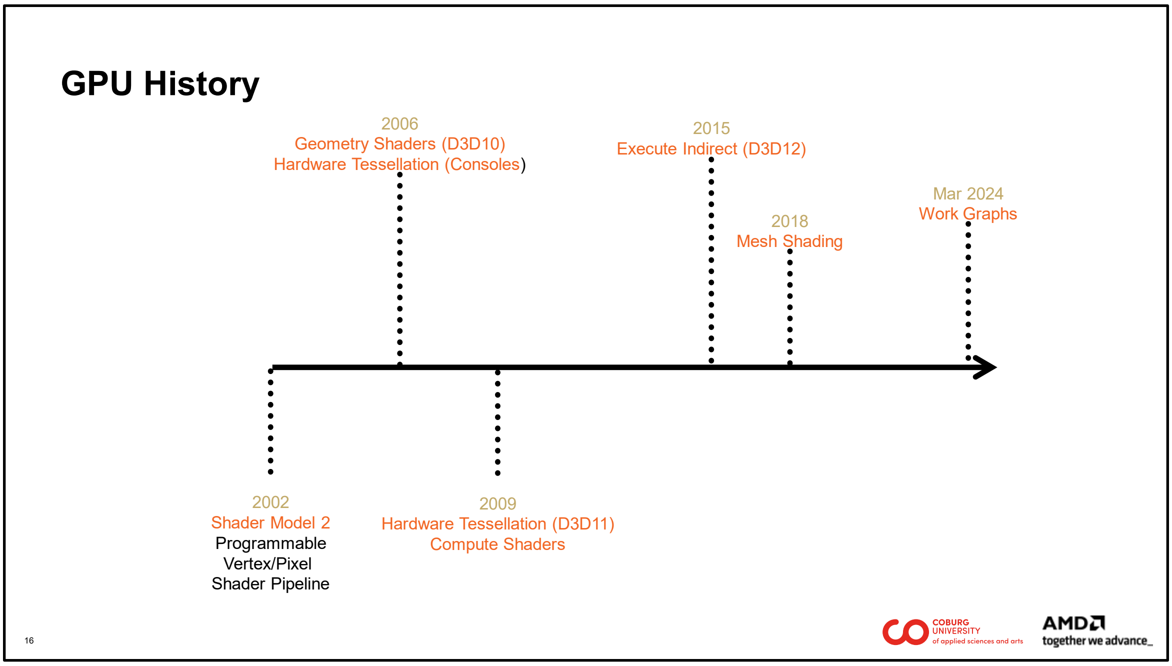
We believe that those concepts are important for Work Graphs.



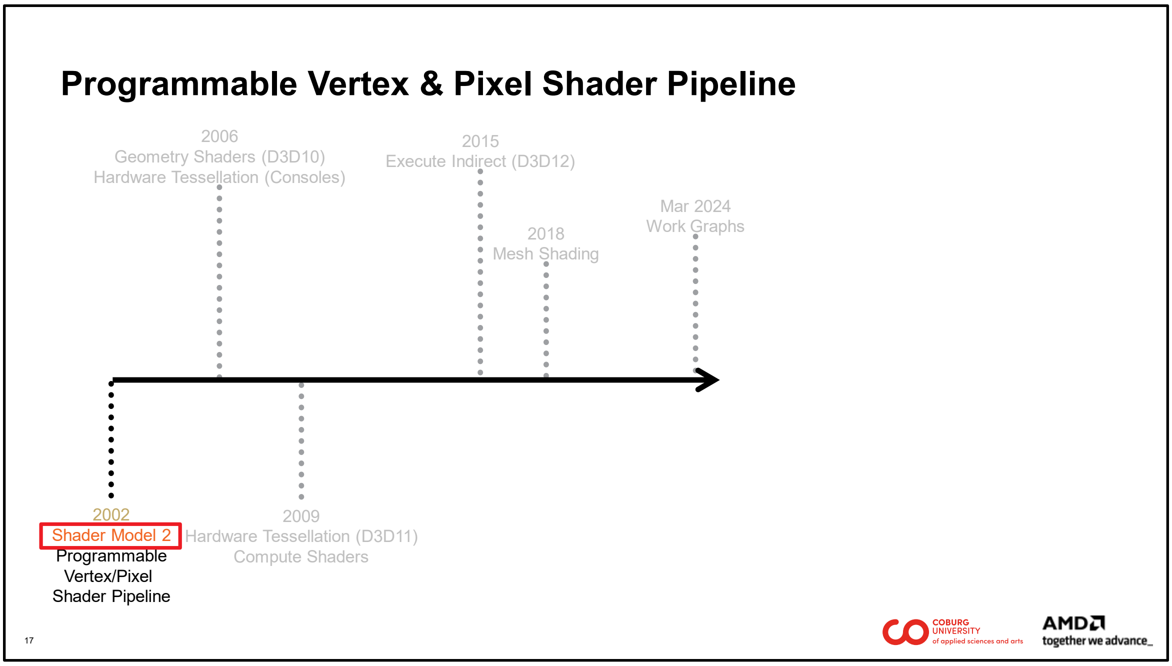
I can best explain these concepts using a brief history of the GPU evolution.

The demand for greater flexibility has driven the evolution of GPU programmability throughout the past decades. Early *register combiners* allowed rudimentary fragment processing [Kilgard 1999], and later *vertex processing* became programmable [Lindholm et al. 2001]. In 2002, the DirectX 9.0 *Shader Model 2.0* is considered to be the first programmable hardware vertex- and pixel-shader pipeline. Two years later, *Shader Model 3.0* added dynamic control flow [Akenine-Möller 2018]. *Geometry shaders* [Blythe 2006] followed with programmable per-primitive processing. *Hardware tessellation* [Andrews and Barker 2006] allowed for fast on-chip geometry amplification [Niessner et al. 2016]. The introduction of *compute-shaders* [Peercy et al., Nvidia 2007] exposed a hardware-oriented programming model – the beginning of GPGPU. It allowed the GPU to execute high-performance graphics and non-graphics applications, as shown for example in the GPU Gems 3 book [Nguyen 2007]. Also, modern *GPU ray tracing* [Haines and Akenine-Möller 2019] on hardware originates back to compute-shader-based ray-tracing implementations [Parker et al. 2010]. With indirect execution or *execute indirect*, the sizes of draw-calls and dispatches are taken from GPU memory, allowing for GPU-driven work creation. Amplification and *mesh shaders* [Kubisch 2018] provide a single-level, non-recursive amplification pipeline for rasterization workloads, following the programming model of compute shaders.

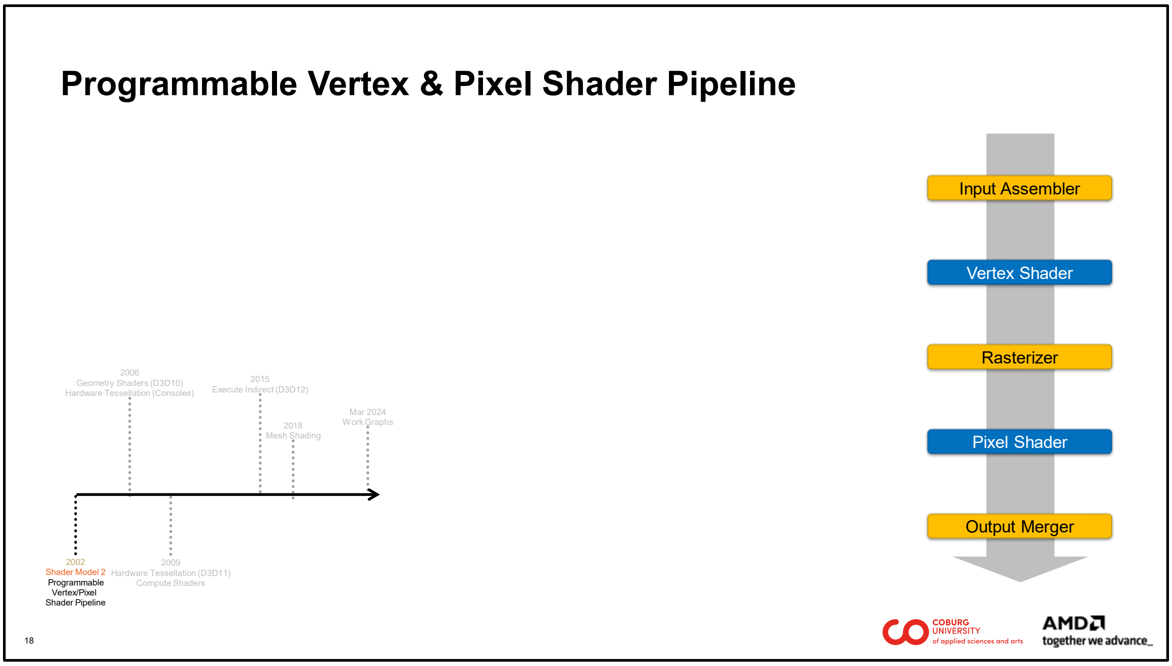
Work Graphs [Microsoft 2024] increase GPU programmability by providing multi-level, self-recursive amplification of both compute and rasterization workloads.



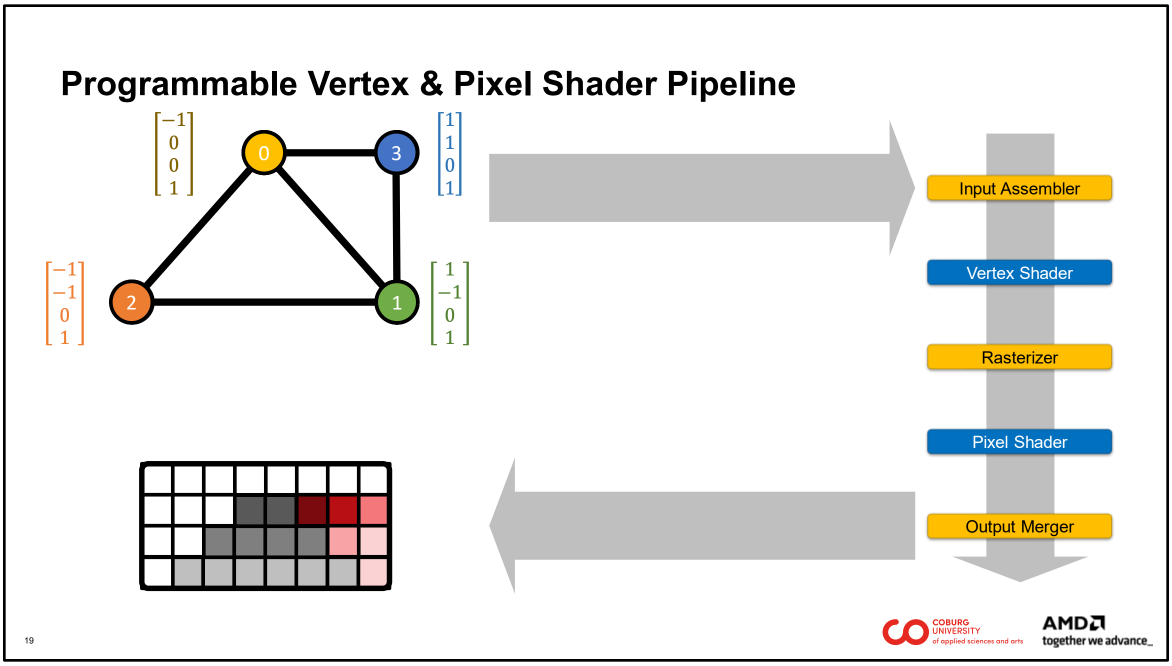
We find those milestones sufficient to explain the basic concepts of GPU pipelines…



… and we start off with *Shader Model 2* which introduced programmable vertex and pixel shading.



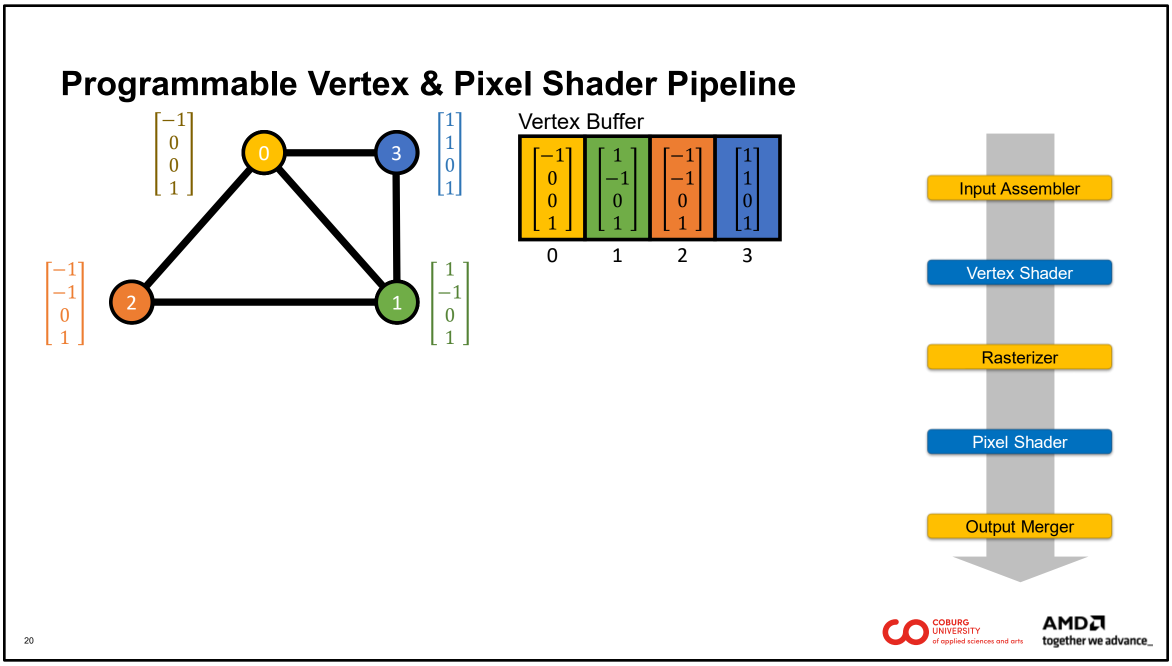
Here is a simplified version of the pipeline. The blue and yellow boxes are the different pipeline stages.



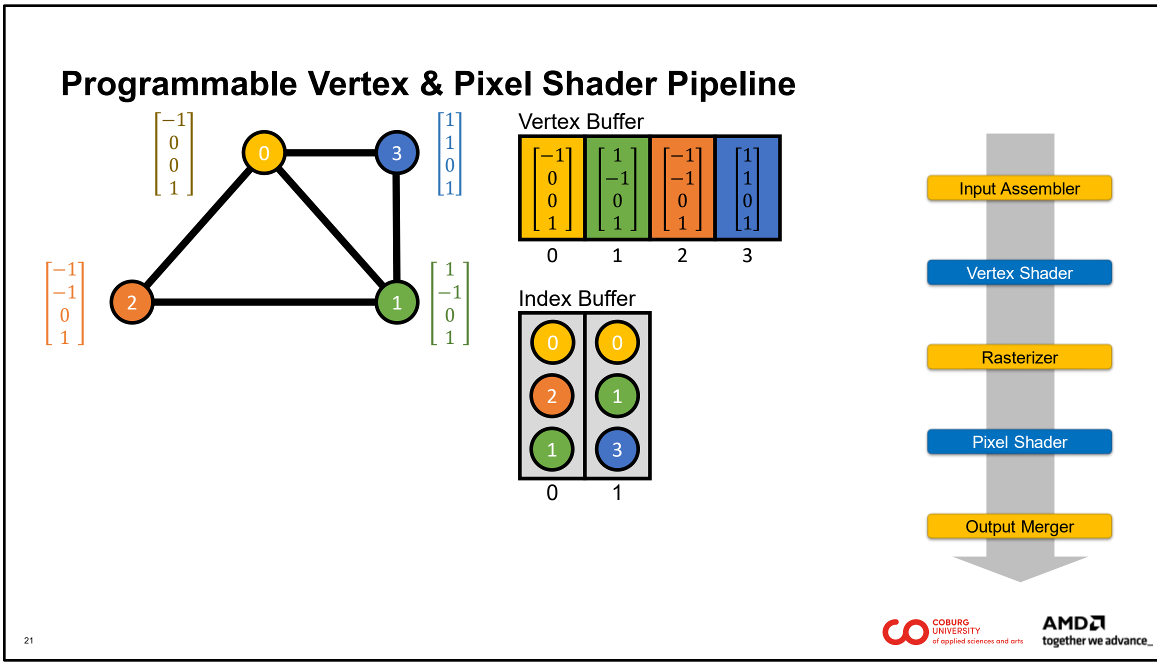
Given a triangle mesh with vertices, shown as circles. The black edges connect the vertices to form triangles. The vertex coordinates are 4D coordinates shown as column vectors.

They are input to the pipeline, shown on the right.

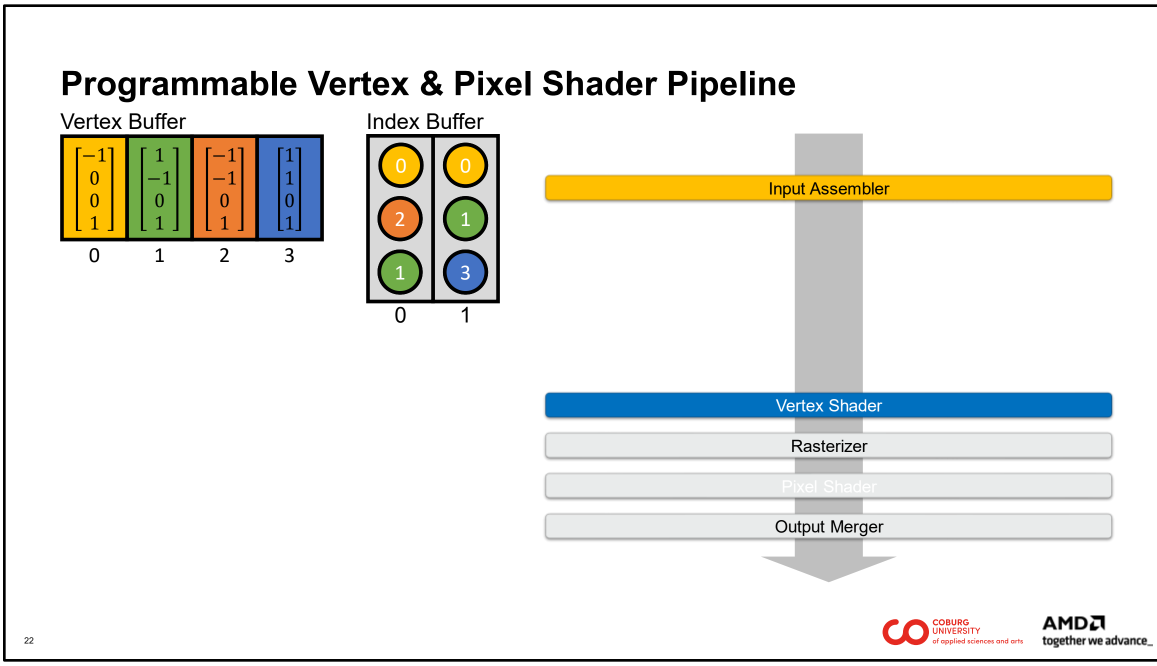
As output, you get pixel graphics, as shown in the pixel grid on the lower left of the slide.



The vertex coordinates are stored in an array called *vertex buffer*.

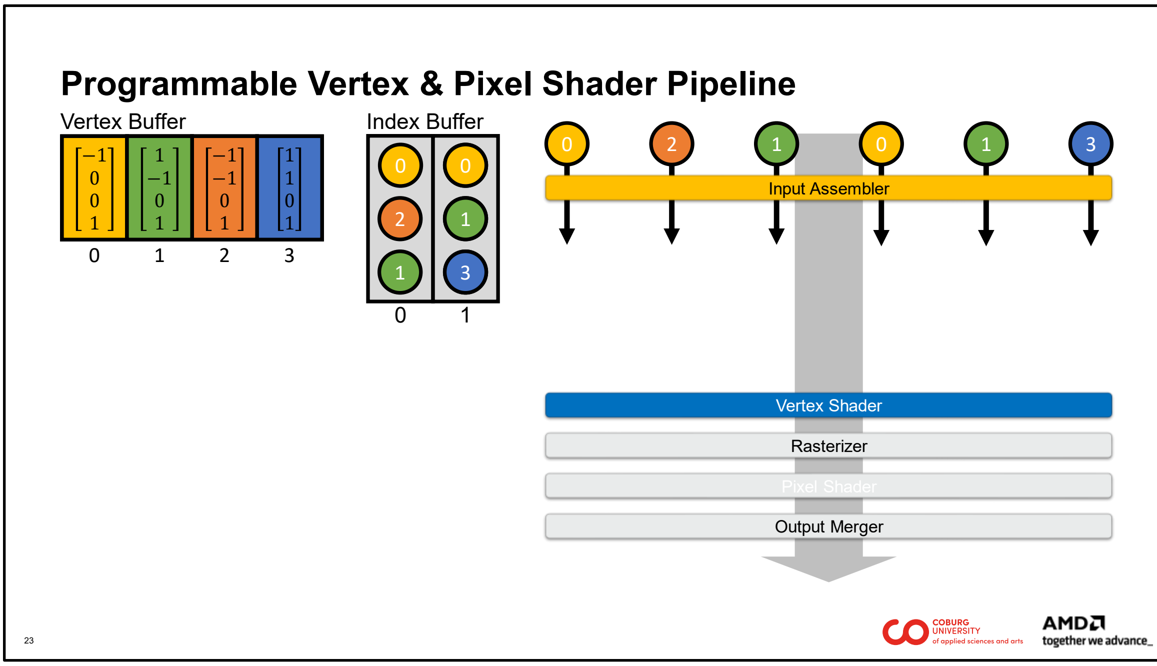


And the vertex indices are stored in what is called an *index buffer*.

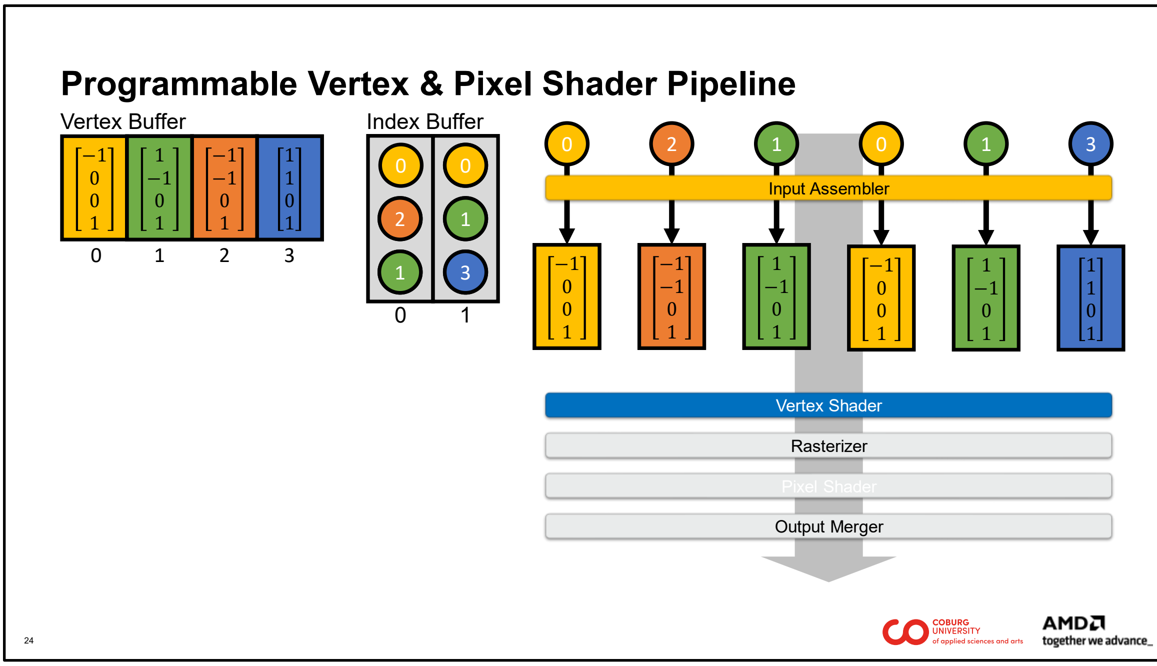


Let's recap what happens when we input a vertex- and an index buffer into the pipeline.

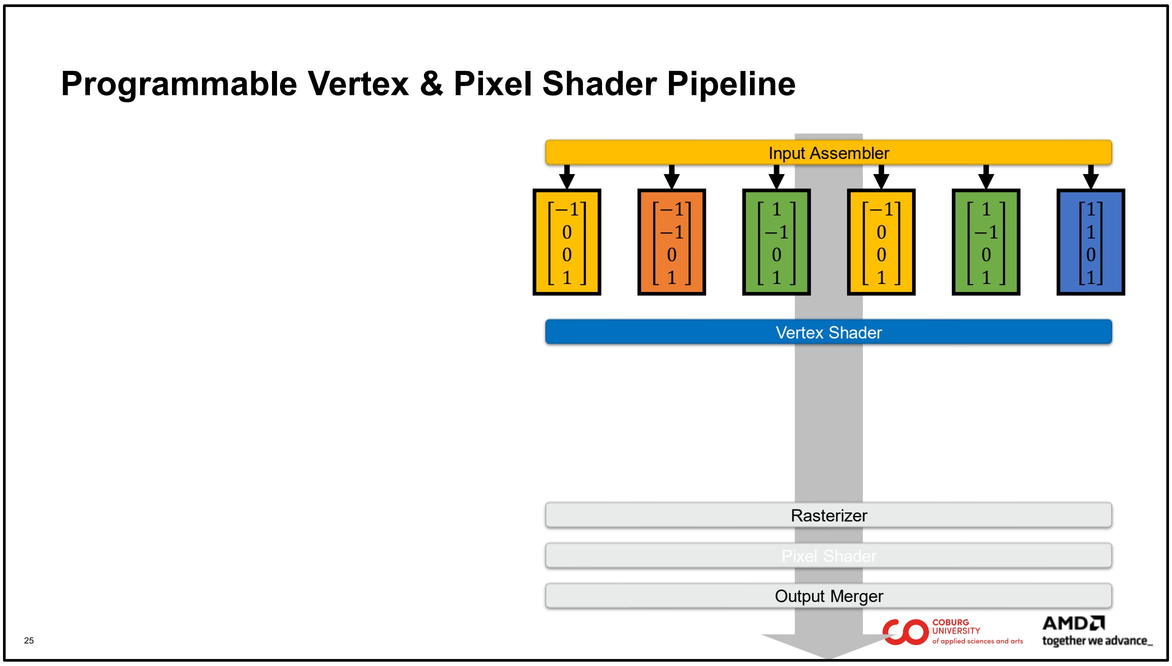
First, each element of the index buffer is fed into the *input assembler*.



The input assembler then gathers the elements from the vertex buffer and makes them available at its outputs.



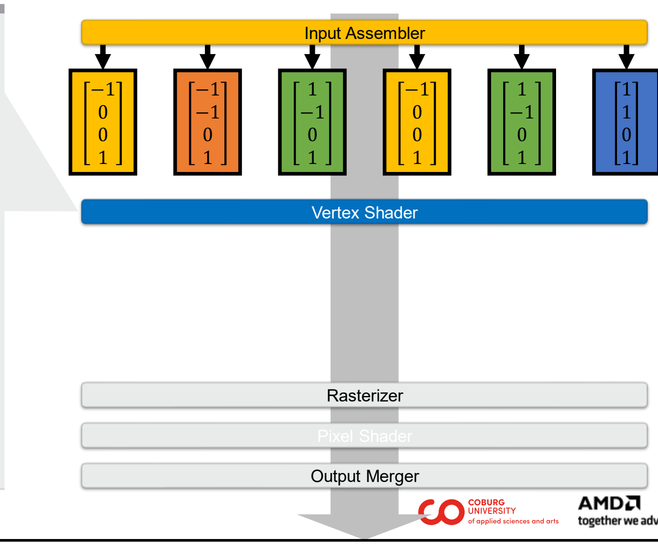
The input assembler can operate on each element independently. This enables GPUs to have a high degree of parallelism.



Then the vertices are fed into the next stage: the *vertex shader*.

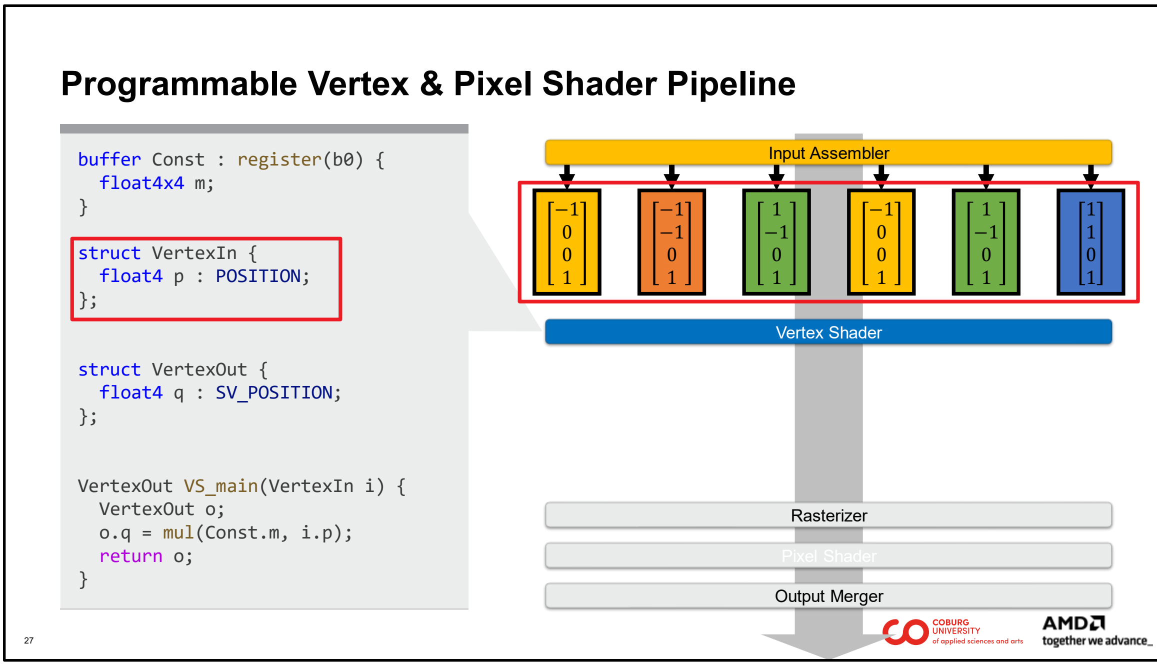
Programmable Vertex & Pixel Shader Pipeline

```
buffer Const : register(b0) {
  float4x4 m;
}

struct VertexIn {
  float4 p : POSITION;
};

struct VertexOut {
  float4 q : SV_POSITION;
};

VertexOut VS_main(VertexIn i) {
  VertexOut o;
  o.q = mul(Const.m, i.p);
  return o;
}
```



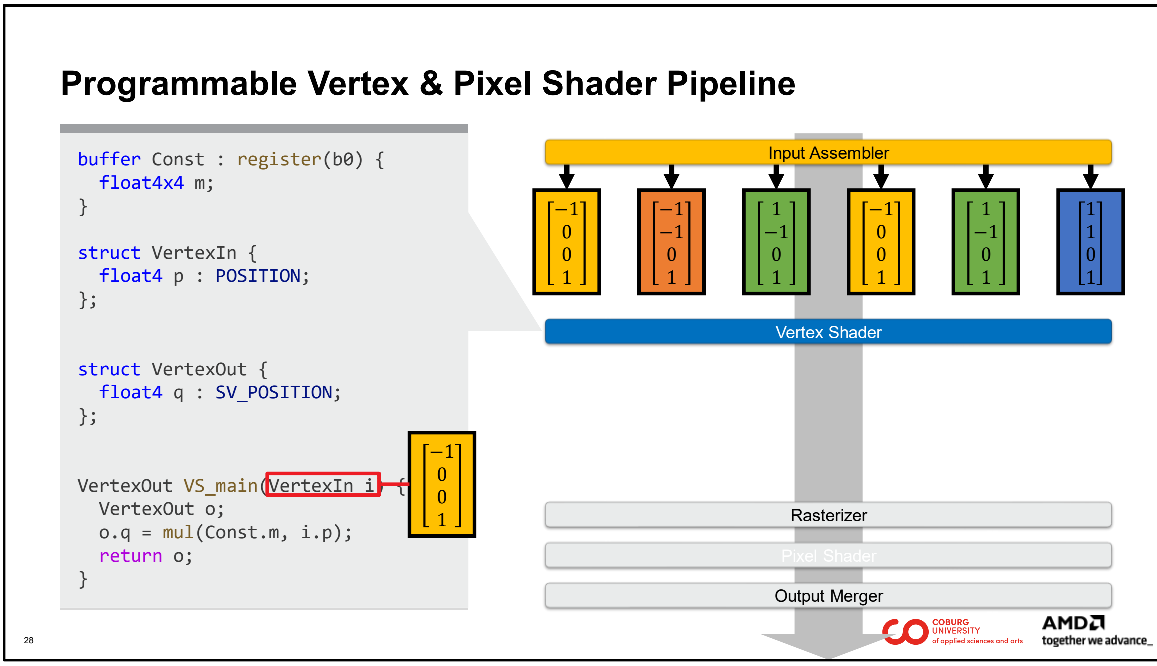
26

In the vertex shader, you as a programmer can write *shader code*, as shown on the left.

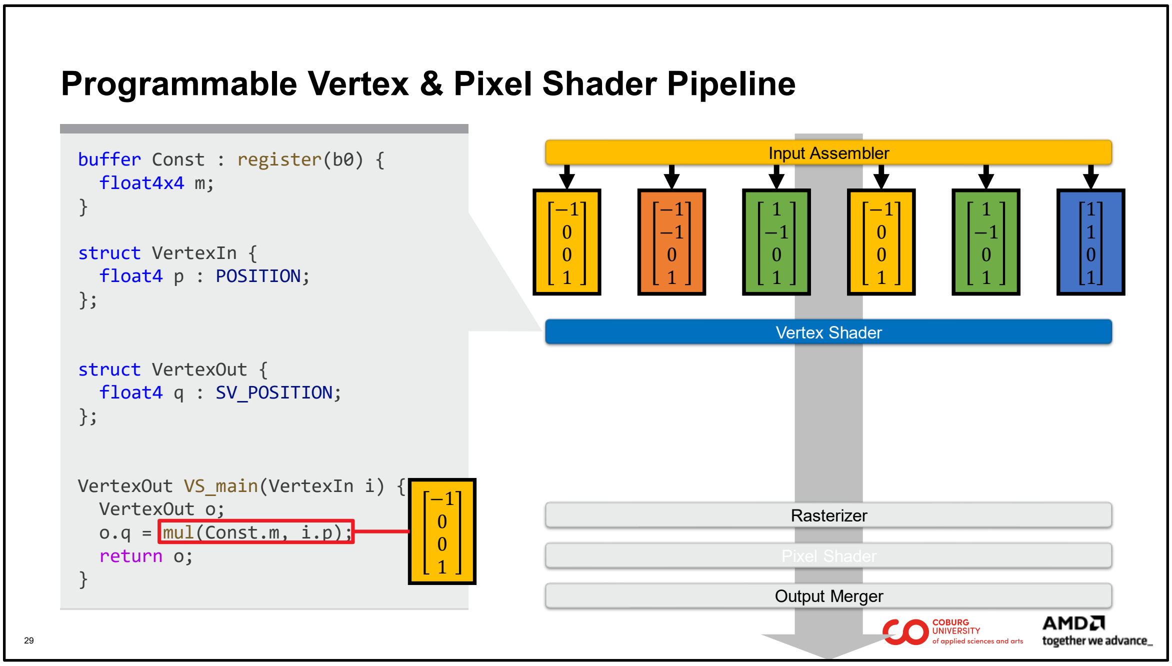


You define a struct, describing the output of the input assembler. At the same time, it serves as input to the vertex shader. For each vertex that the input assembler outputs, the GPU launches one vertex shader thread.

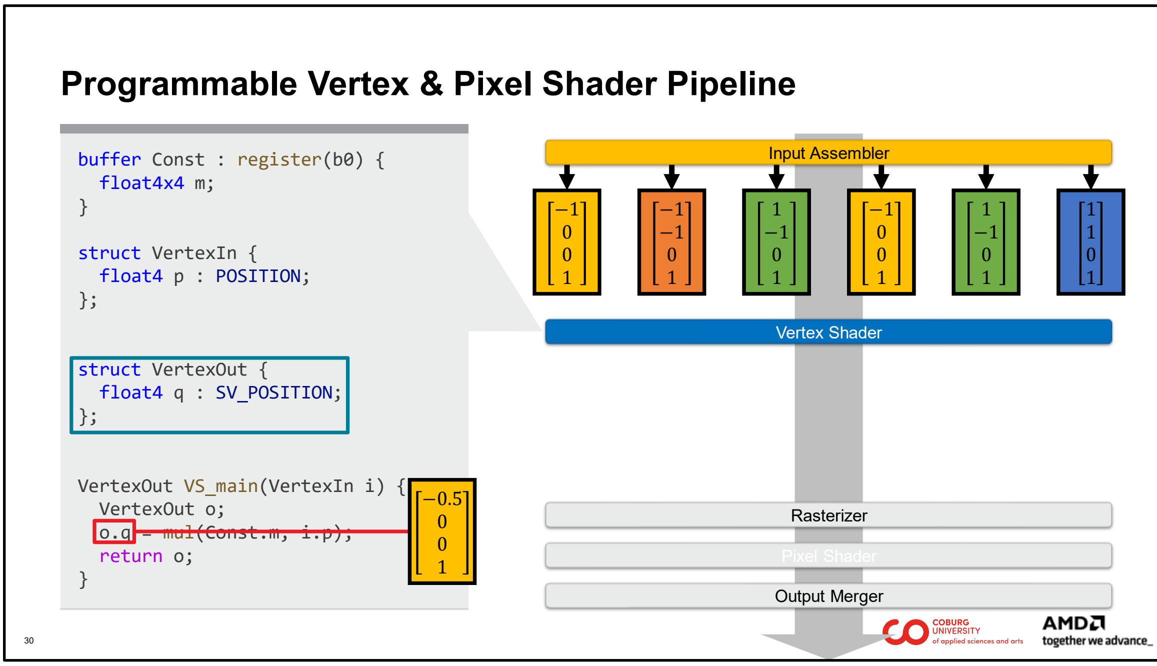
Let’s do an example with the first vertex.



The vertex inside the yellow box serves as input to one vertex shader thread.



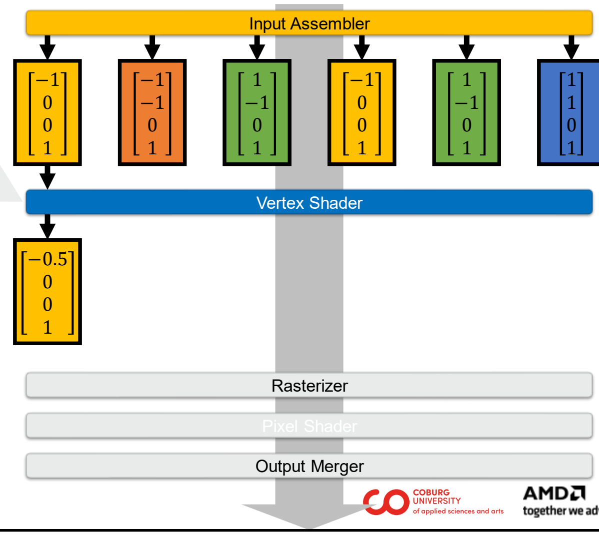
With that input, the vertex shader thread carries out the operations a programmer specified…



… and writes the output, that a programmer defined with the struct shown in the blue box.

Programmable Vertex & Pixel Shader Pipeline

```
buffer Const : register(b0) {
  float4x4 m;
}

struct VertexIn {
  float4 p : POSITION;
};


struct VertexOut {
  float4 q : SV_POSITION;
};


VertexOut VS_main(VertexIn i) {
  VertexOut o;
  o.q = mul(Const.m, i.p);
  return o;
}
```

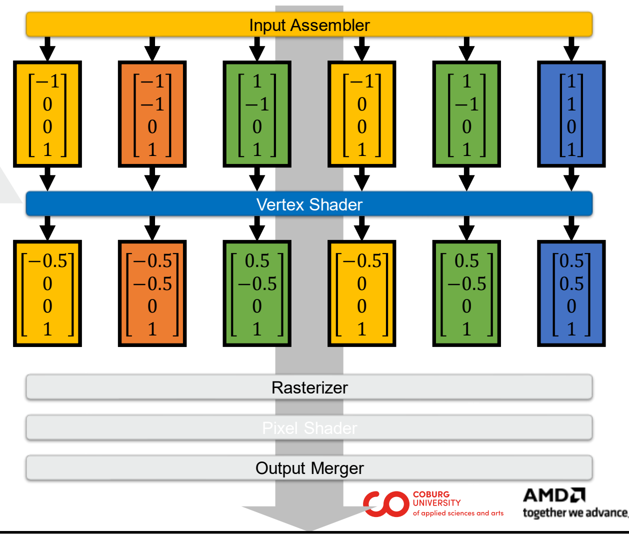


31

The result is then made available at the vertex shader output.

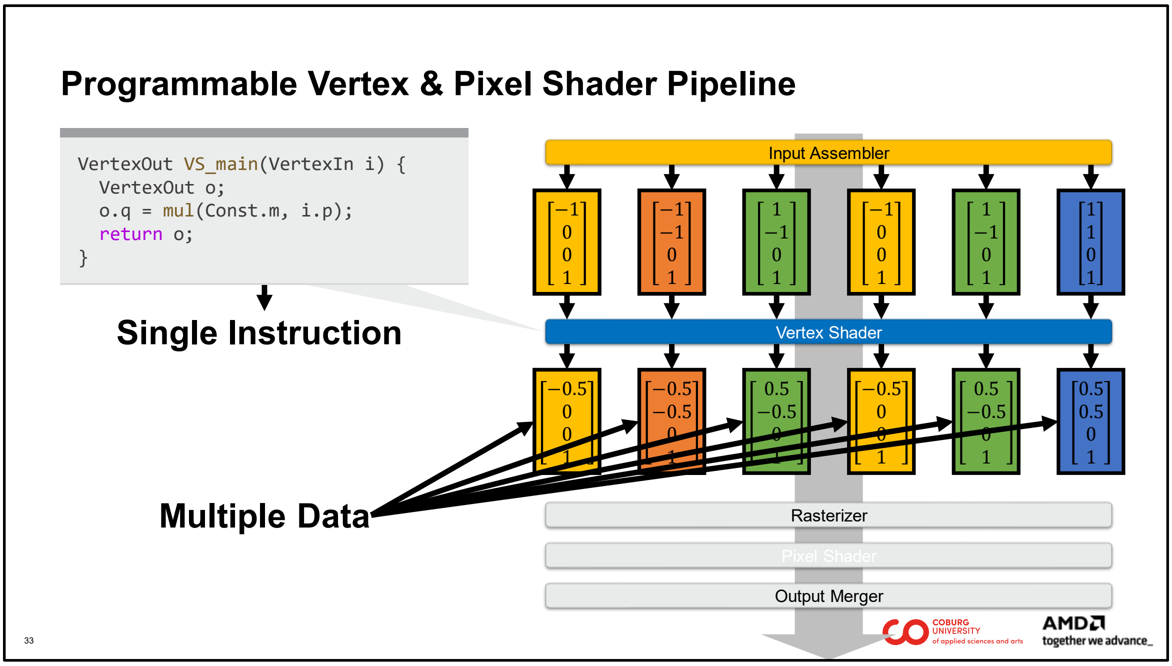
Programmable Vertex & Pixel Shader Pipeline

```
buffer Const : register(b0) {
  float4x4 m;
}

struct VertexIn {
  float4 p : POSITION;
};


struct VertexOut {
  float4 q : SV_POSITION;
};


VertexOut VS_main(VertexIn i) {
  VertexOut o;
  o.q = mul(Const.m, i.p);
  return o;
}
```



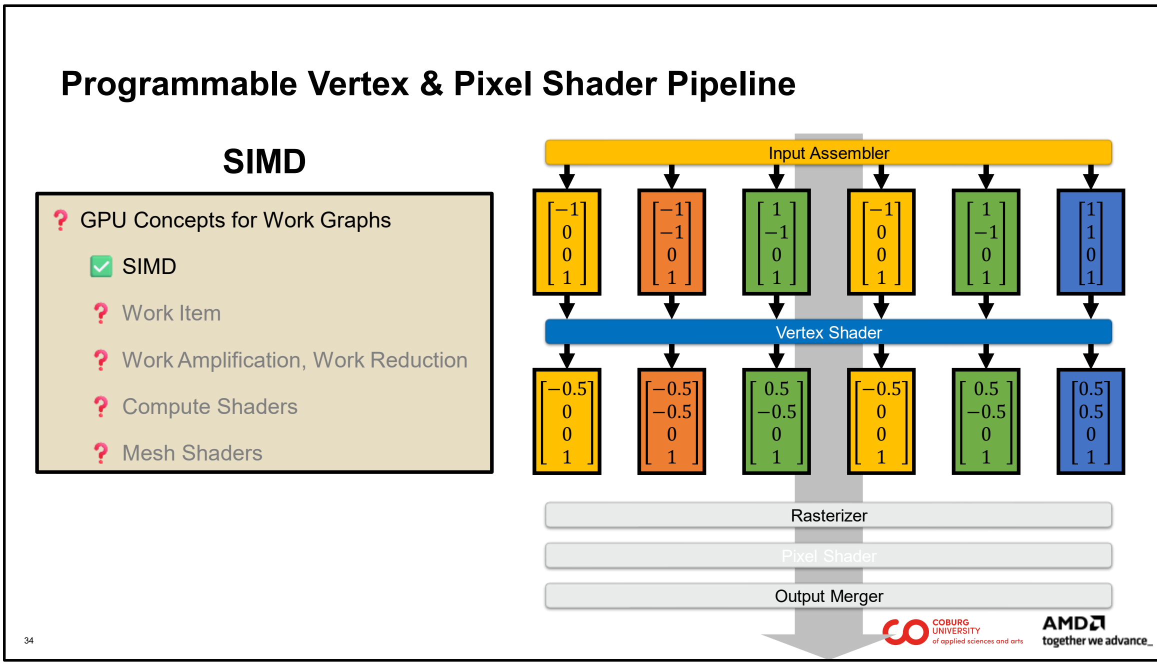
32

All other vertices undergo the same fate: They pass through the same vertex shader code, however, using different inputs.



This is a very important concept. The same piece of code is executed on different data items.

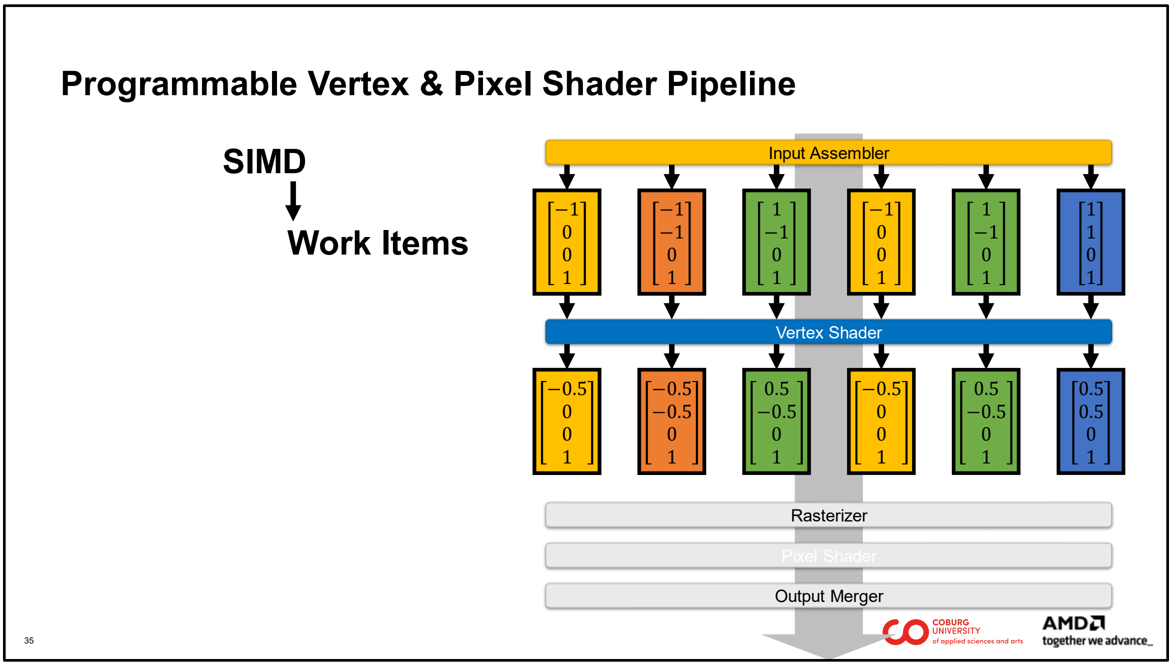
In other words, a single instruction operates on multiple data. Hence the name *Single Instruction, Multiple Data…*



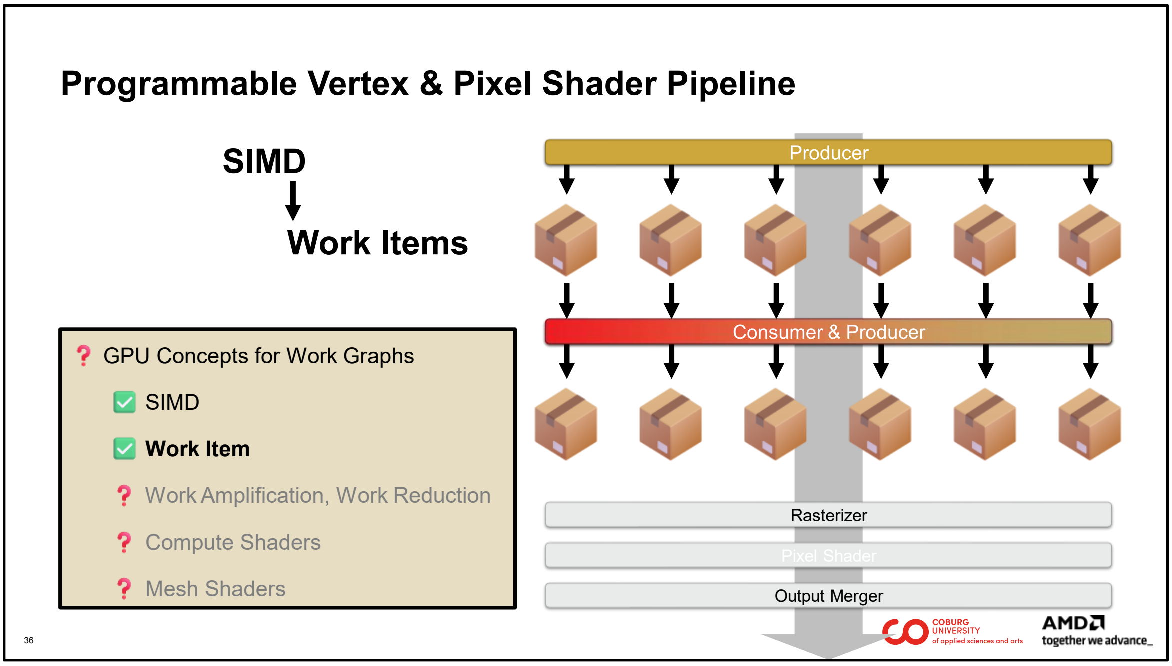
… or short *SIMD*.

SIMD is the underlying parallel computing model of GPUs and it is very important for their performance. Since Work Graphs run on a GPU, they make use of the SIMD model.

Side note: In the context of GPUs, the massively parallel underlying computing model is sometimes also referred to as SIMT (Single Instruction, Multiple Threads).



From an abstract perspective, the vertices attributes, as the **D** (**d**ata) in SIM**D**, are *Work Items*…



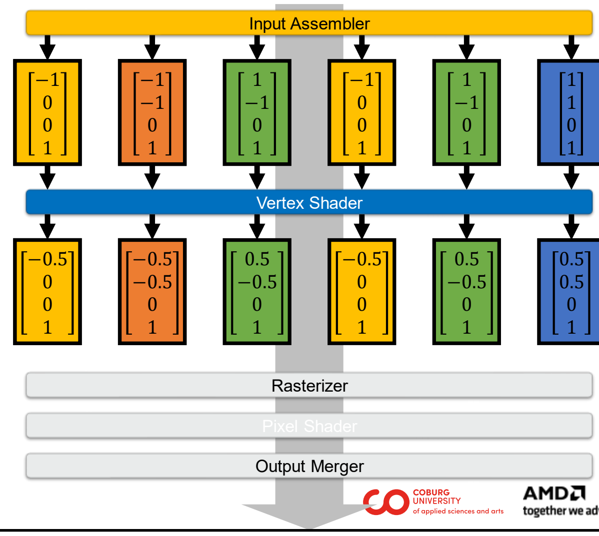
… that flow through a pipeline.

In a pipeline, one stage acts as a producer, and the subsequent stage as a consumer. A stage can consume and produce items at the same time.

The items that flow through the pipeline are called *work items*.

From that point of view, the graphics pipeline is already providing a *data-flow-oriented model* which is also used in Work Graphs, however, in a much more sophisticated way.

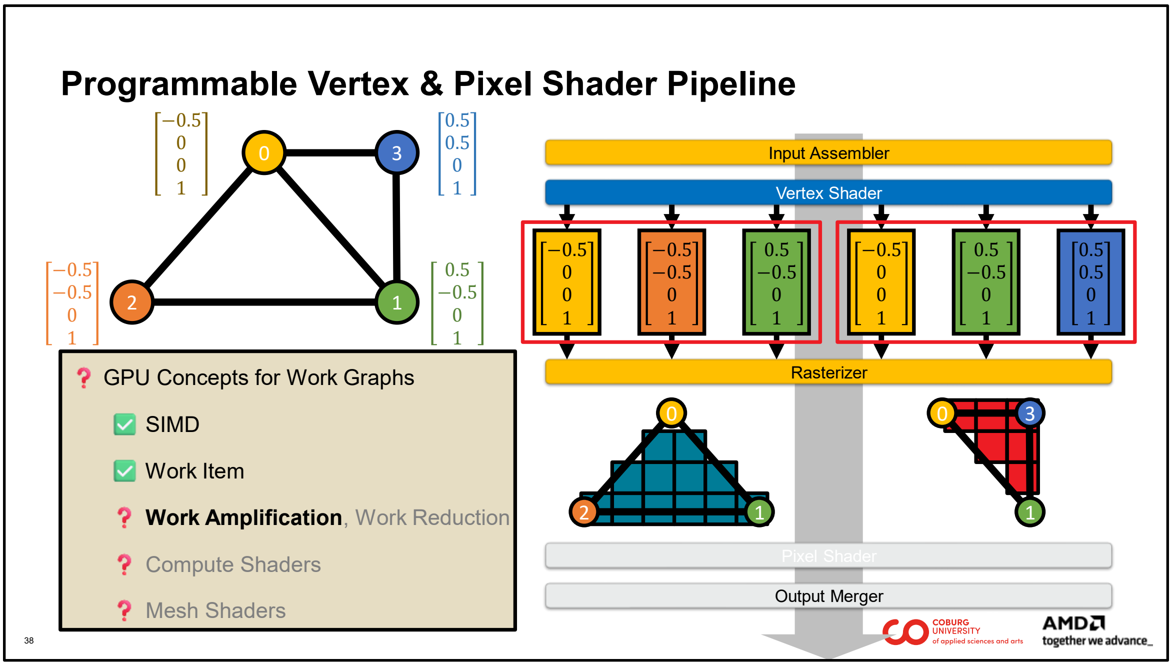
Programmable Vertex & Pixel Shader Pipeline



37

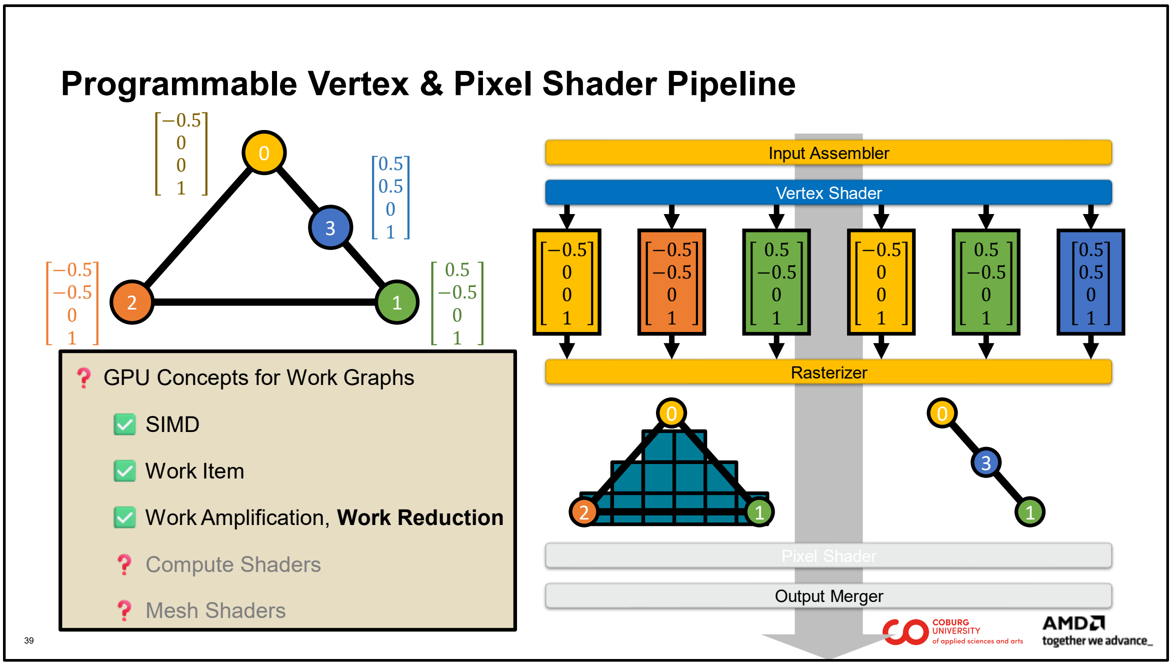
But going back to what you are already familiar with: Our vertex-pixel-shader pipeline.

The vertex shader has just transformed the vertices.

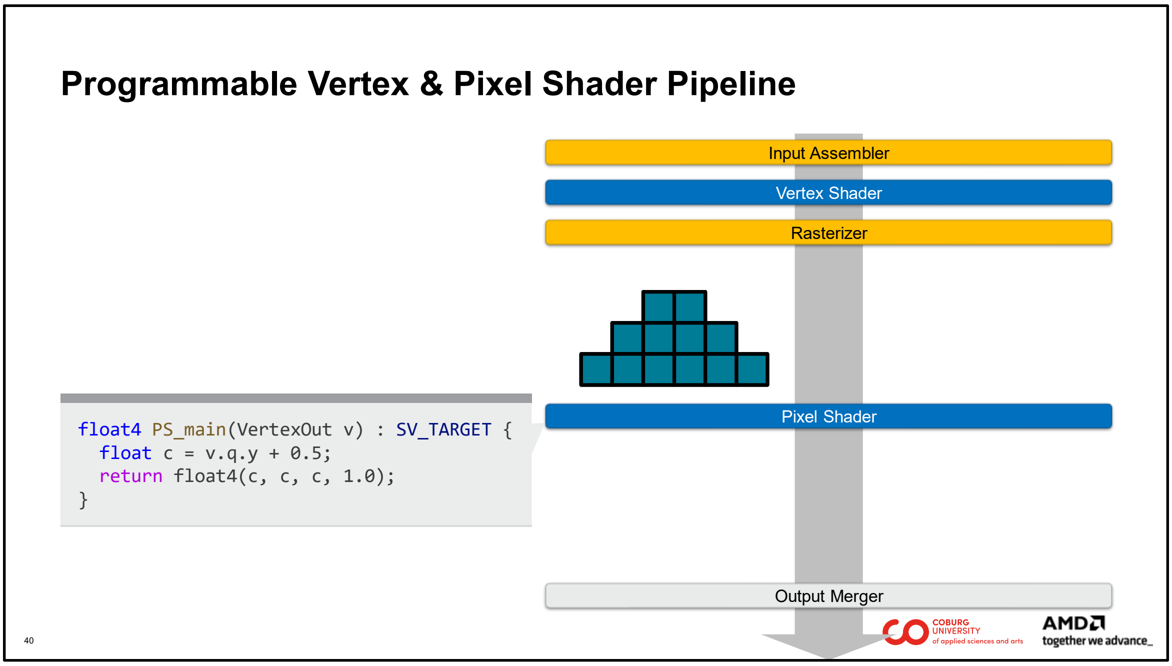


The *rasterizer* then gathers three-tuples of vertices and discretizes the triangles into fragments.

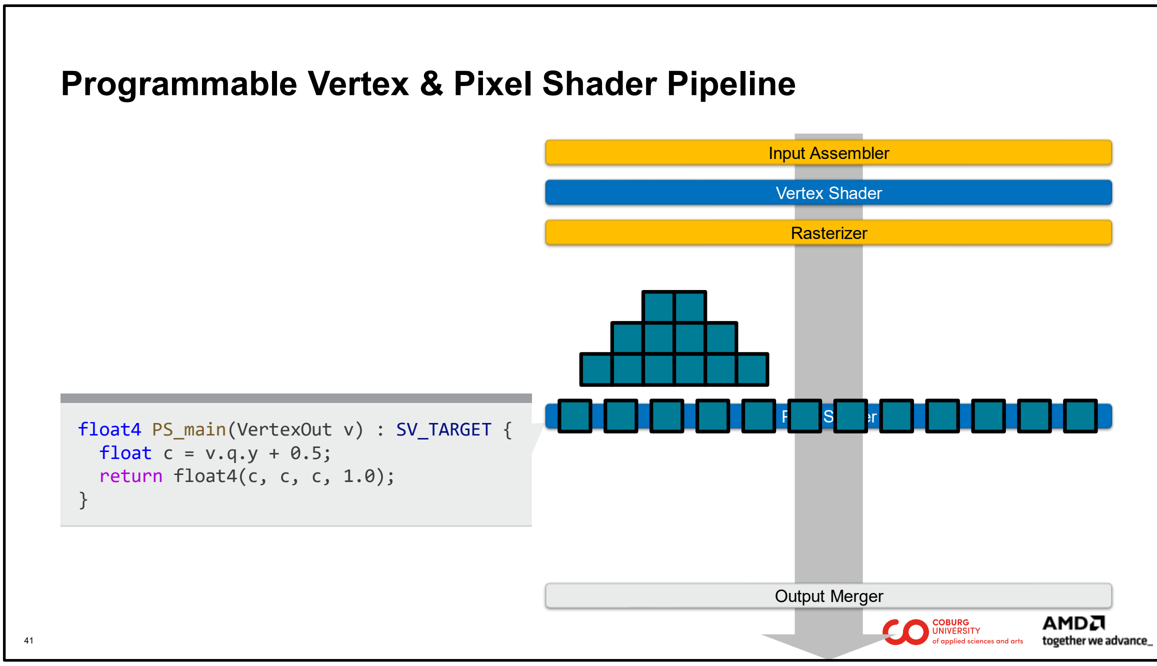
This can be considered *work amplification*. Consider a triangle an input data item. We amplify that input data item to a much larger number of output items, i.e., our fragments.



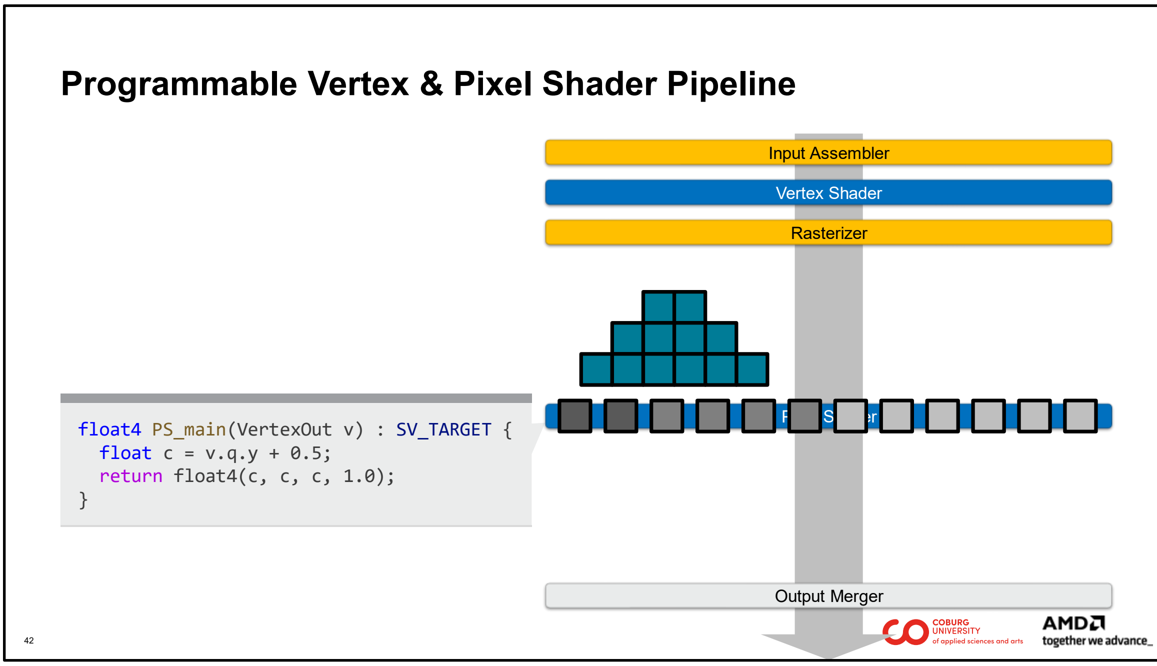
But the rasterizer can also *reduce work* entirely, for example by removing triangles that do not produce fragments.



The *pixel shader* is again a program using the SIMD model. Each fragment is its input work item …

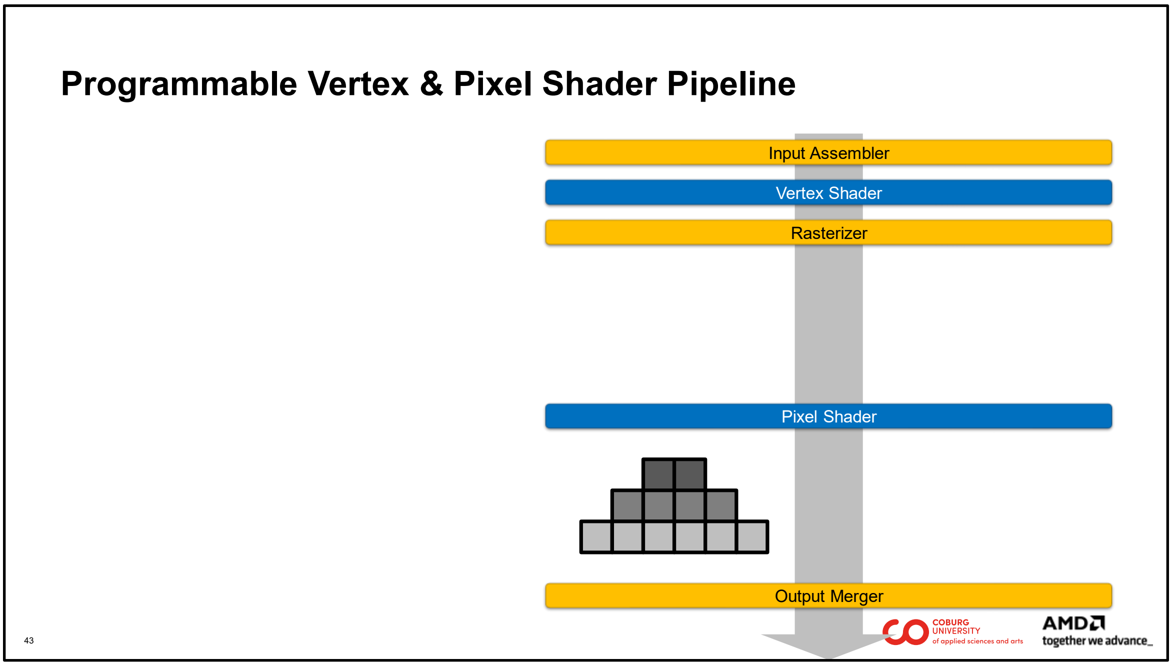


… and gets executed by one thread on the GPU …

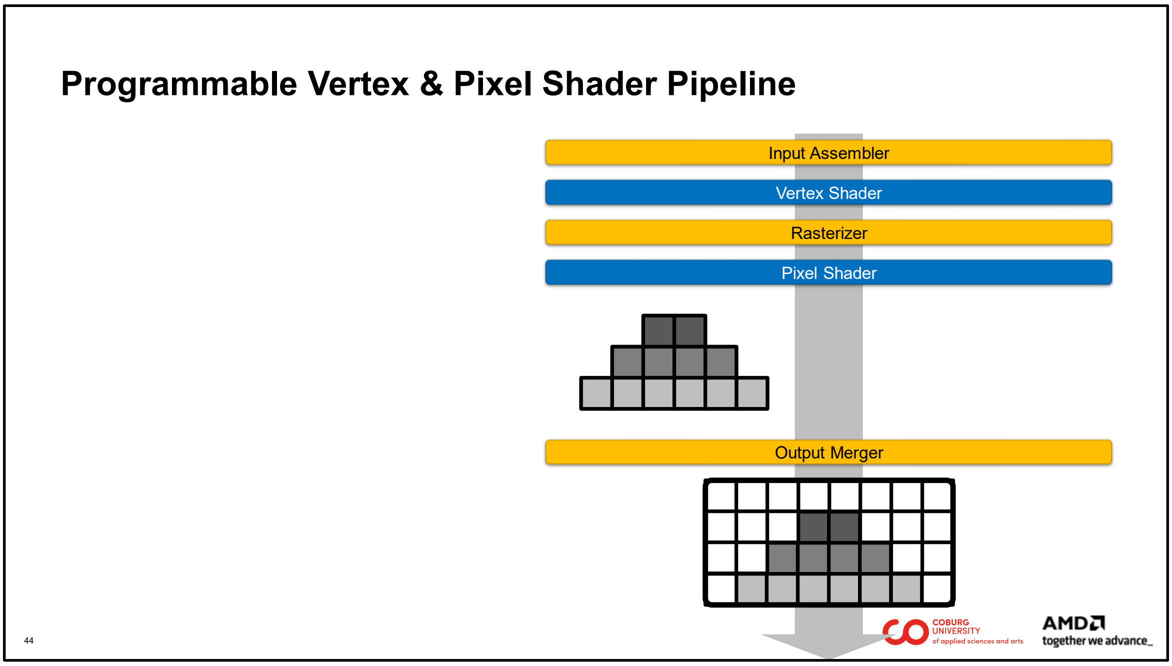



```
float4 PS_main(VertexOut v) : SV_TARGET {
  float c = v.q.y + 0.5;
  return float4(c, c, c, 1.0);
}
```

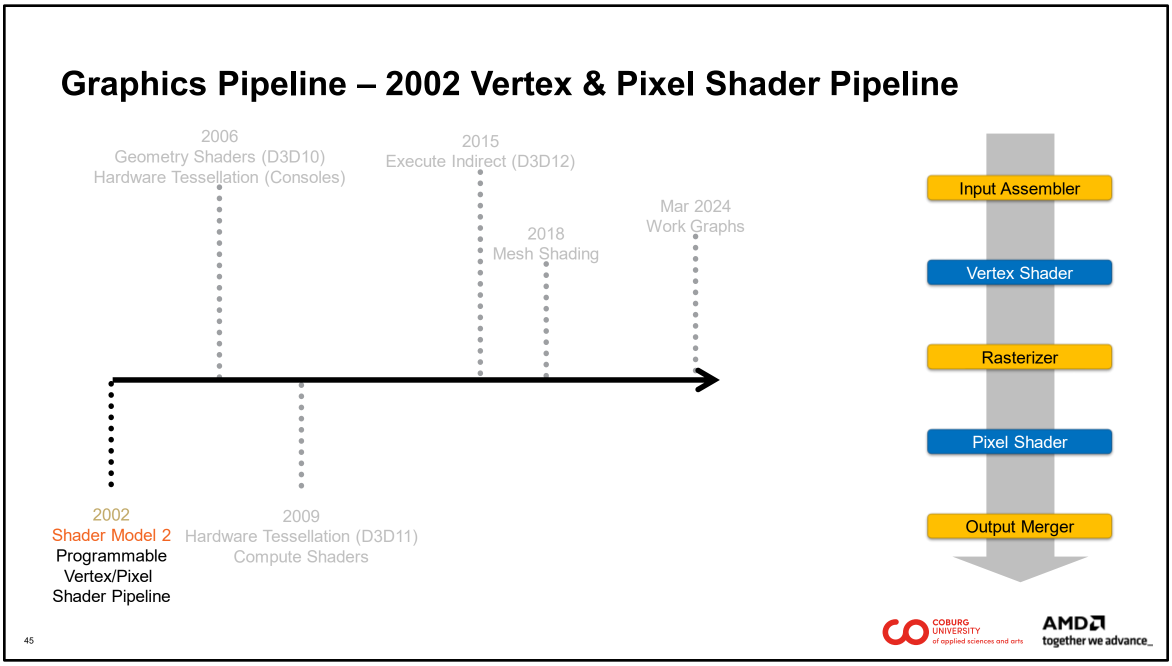
… which computes its output color …



Each fragment shader thread then passes its output data item to the *output merger*.

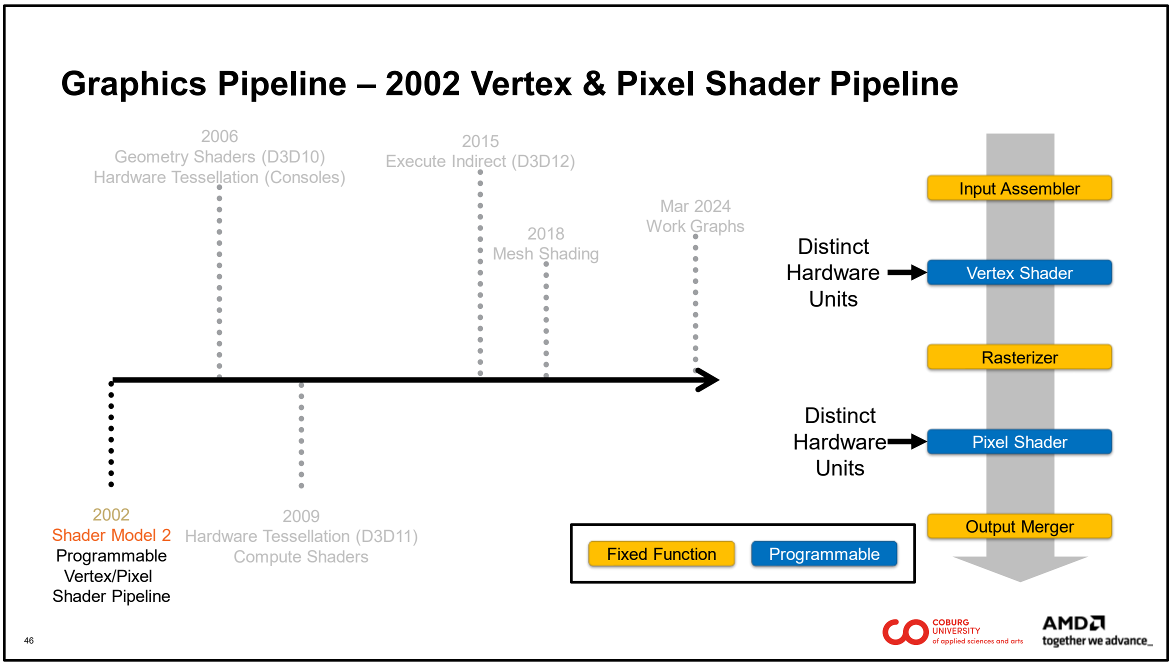


The output merger then merges the fragments with the existing ones to form the final image.

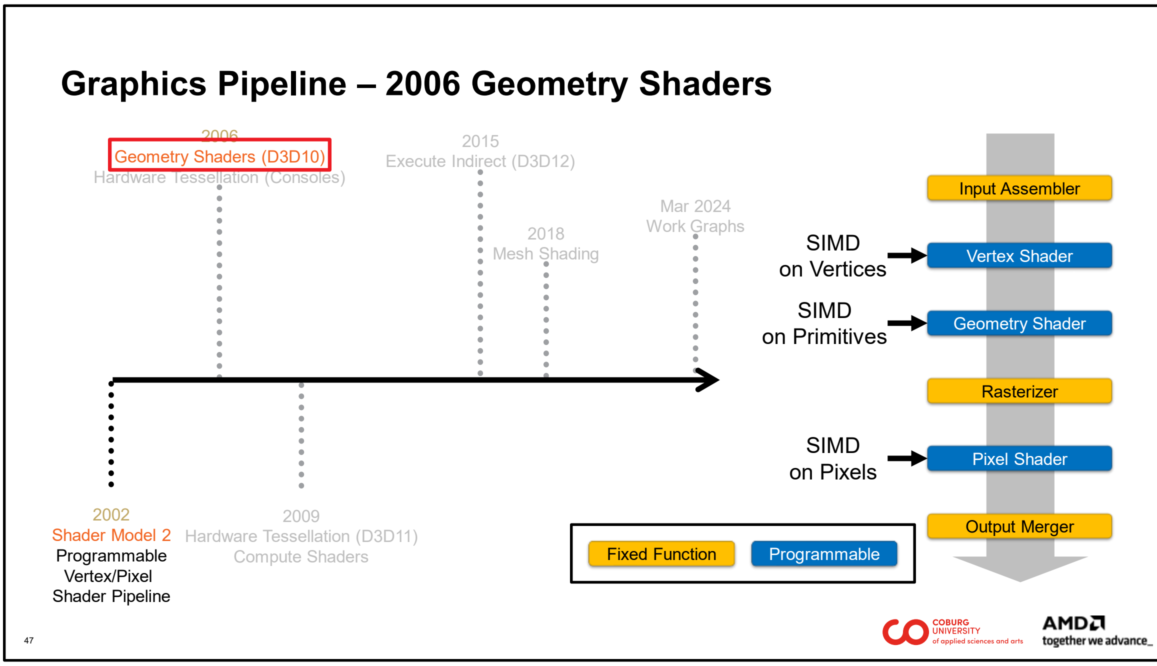


This concludes our talk about the vertex- and pixel-shader pipeline from 2002.

We have seen that some concepts that we will use for Work Graphs already existed back then.

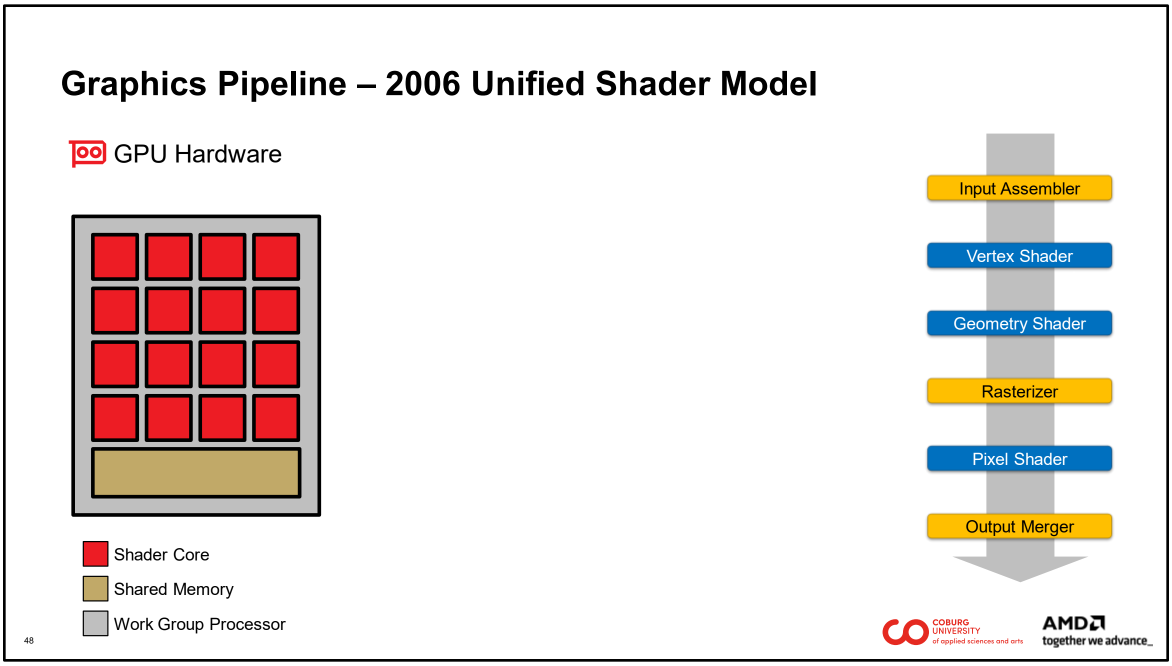


The pipeline has two programmable stages and several configurable fixed-function stages.



The vertex shader is SIMD on vertices; the fragment shader is SIMD on fragments. In 2006, D3D10 introduced *geometry shaders*, another programmable stage. That stage uses SIMD on triangles and other primitives.

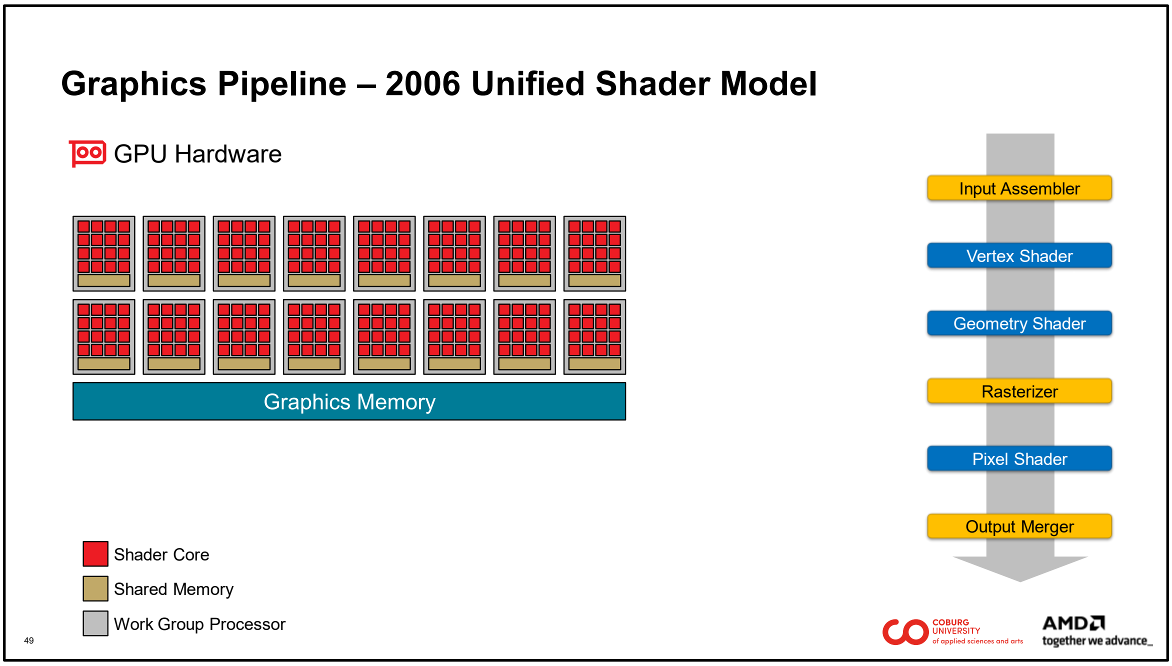
The hardware designers observed that all programmable stages use the same underlying SIMD principle.



To provide a common abstraction, they created the *unified shader model*.

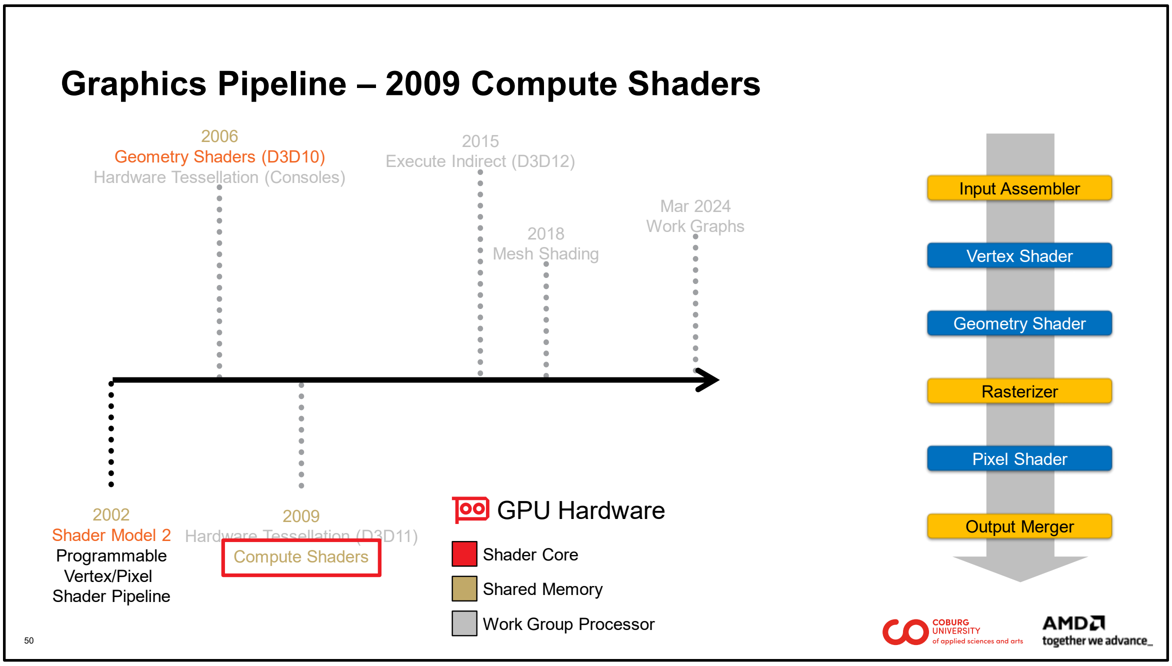
Each thread maps onto a *shader core*. Multiple shader cores are grouped into a *work group processor*. For example, on the AMD RDNA™ 3 architecture, we have 128 shader cores per work group processor.

The shader cores of a work group processor can communicate over a shared memory, which has 128 KiB on AMD RDNA™ 3 GPUs.

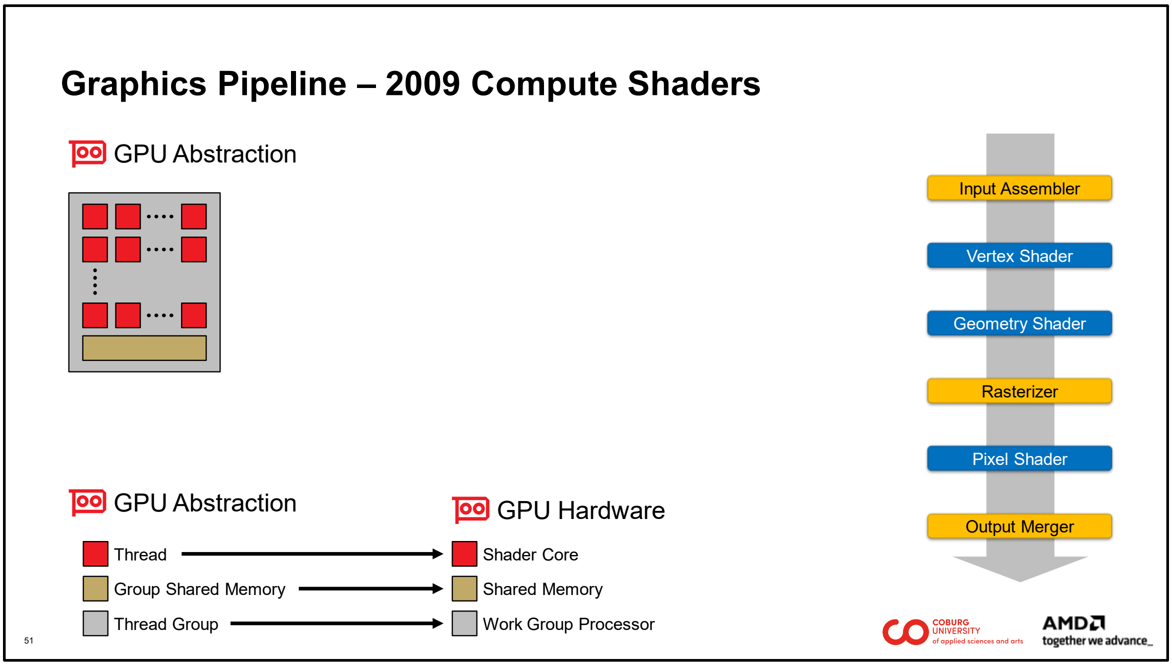


Several such work group processors are on a GPU. On The AMD Radeon™ RX 7800 XT, we have 30 work group processors.

The work group processors share a common Graphics Memory. Today, that is several GiBs large.



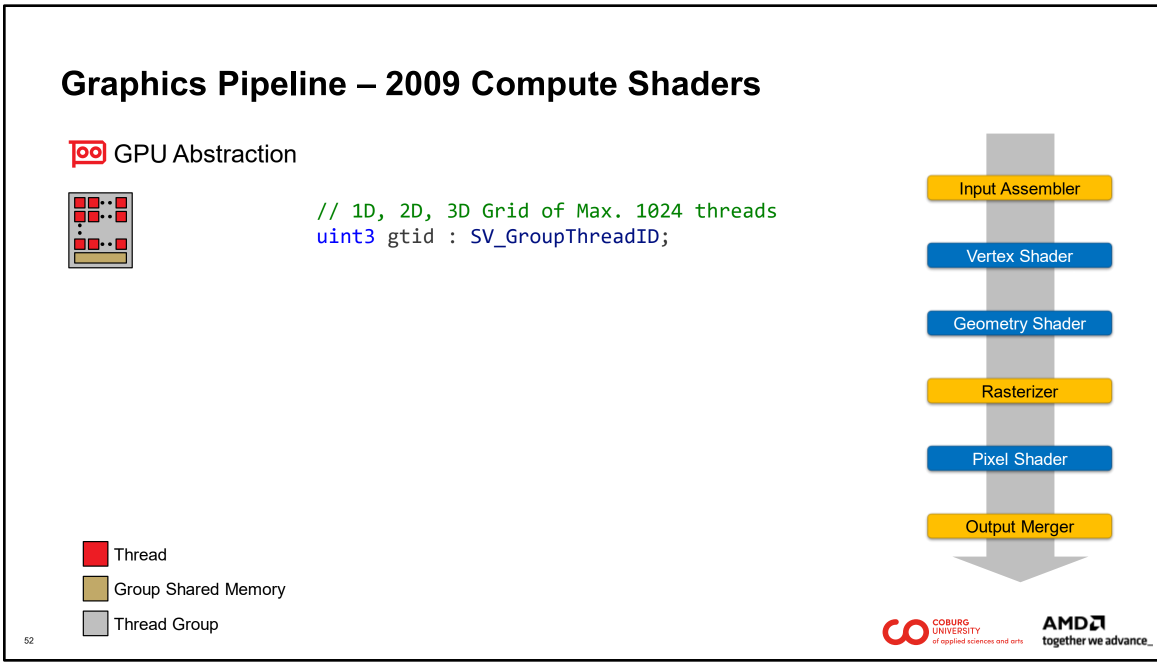
Now with such an abstract model of the GPU, it was just obvious to define new shader types. This gave rise to *compute shaders*.



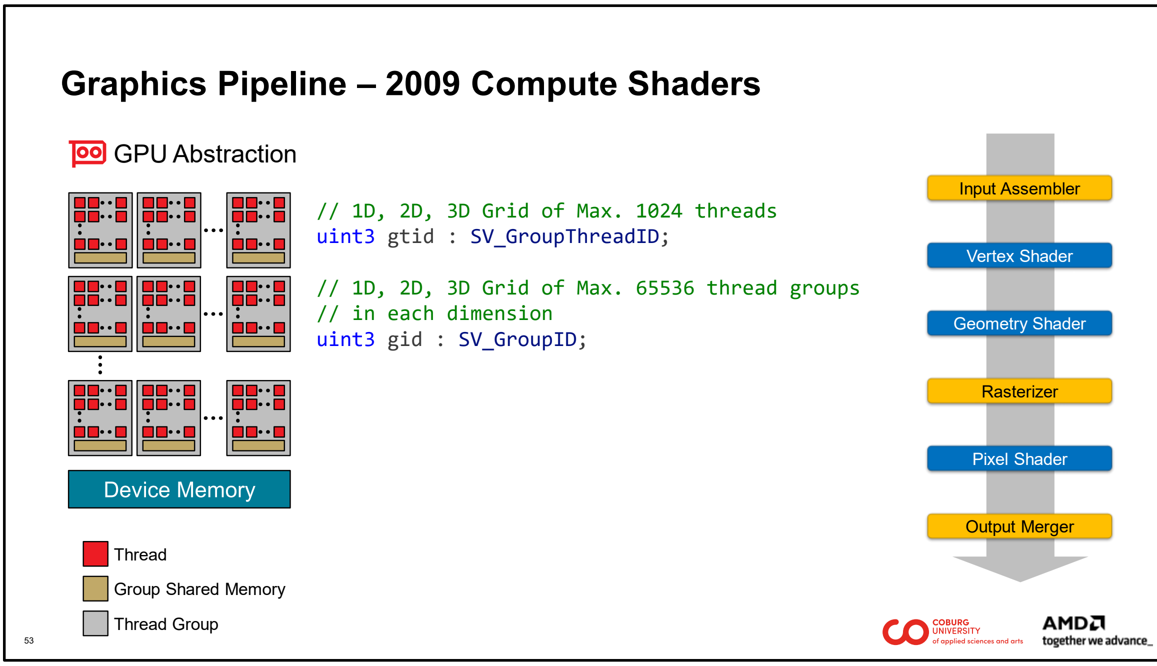
Compute shaders require a *GPU abstraction*.

That contain *threads*, which access a common *group shared memory*. Threads are mapped onto shader cores and *group shared memory* maps to shared memory.

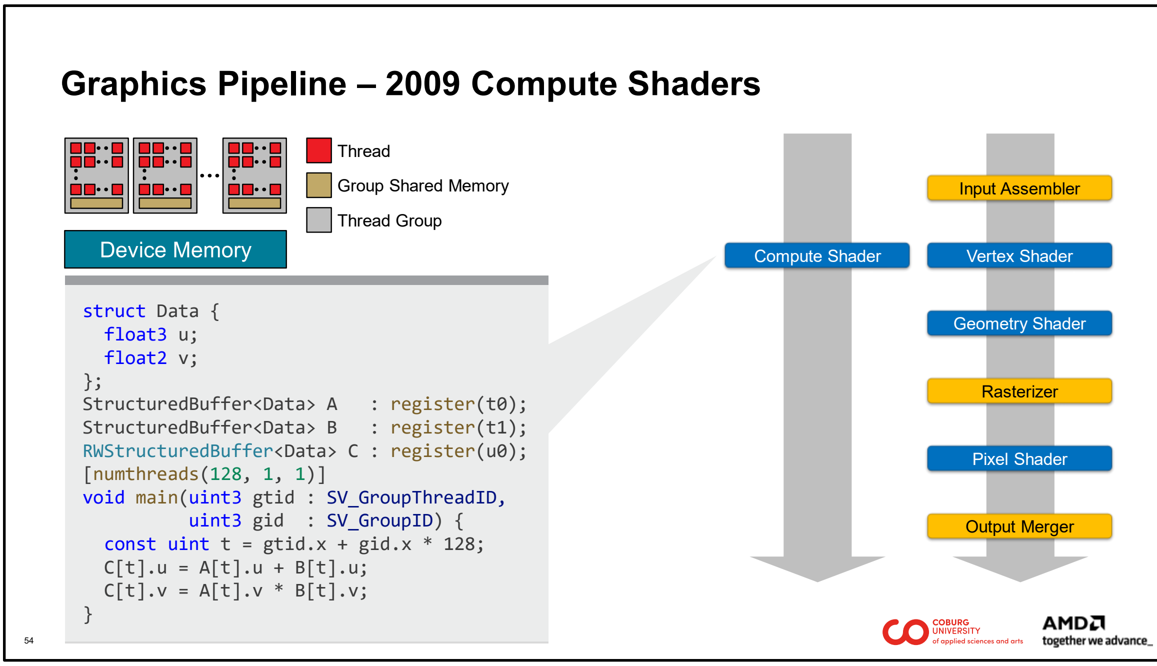
Threads are clustered into *thread groups*. On GPU hardware, a thread group is executed on a work group processor.



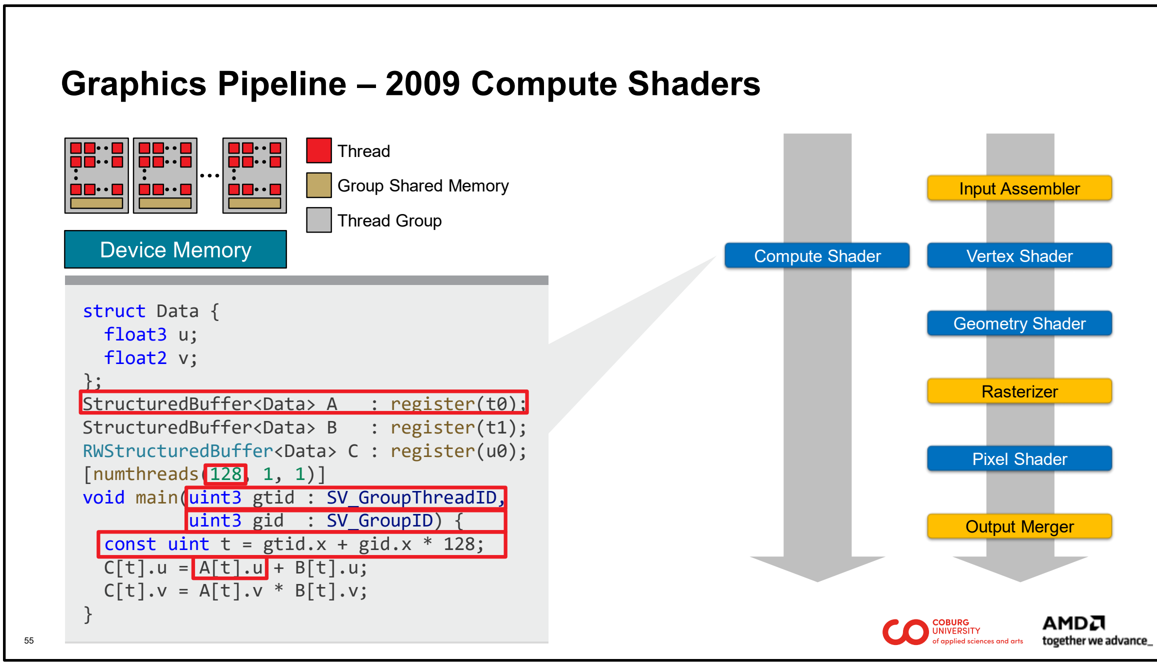
In a compute shader program, the `SV_GroupThreadID` semantic provides a 3D index in a grid of up 1024 threads of a thread group.



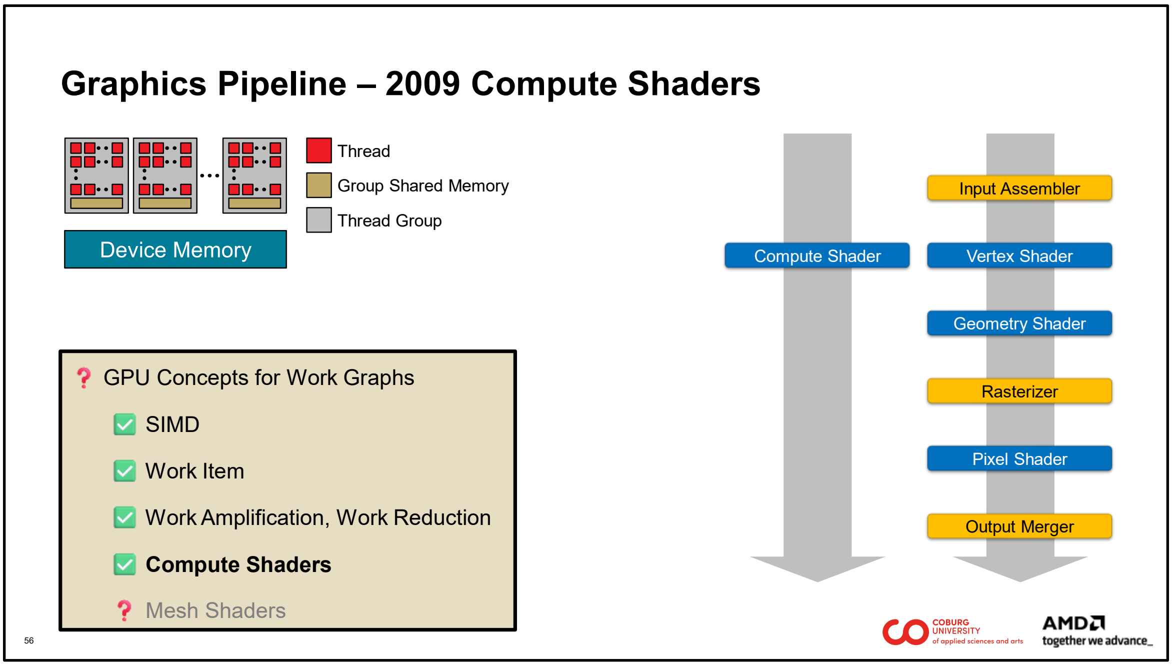
To locate a thread group, the `SV_GroupID` semantic provides the programmer with a 3D index into the grid of thread groups.



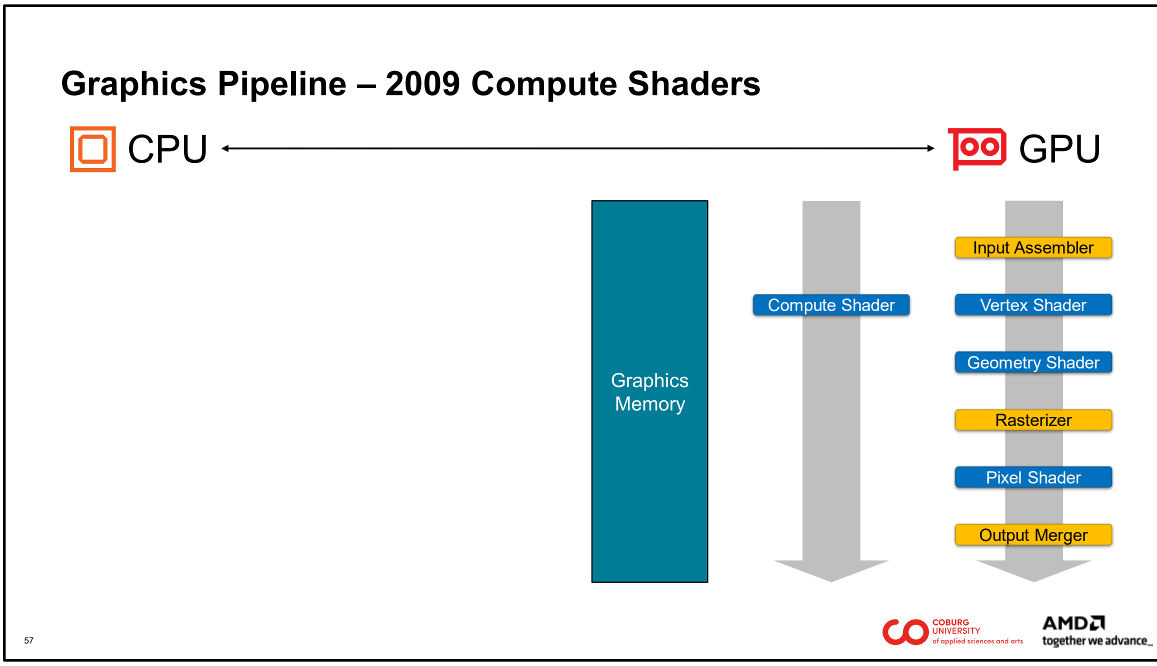
Compute shaders are programmed using shader programs that adhere to the SIMD model.



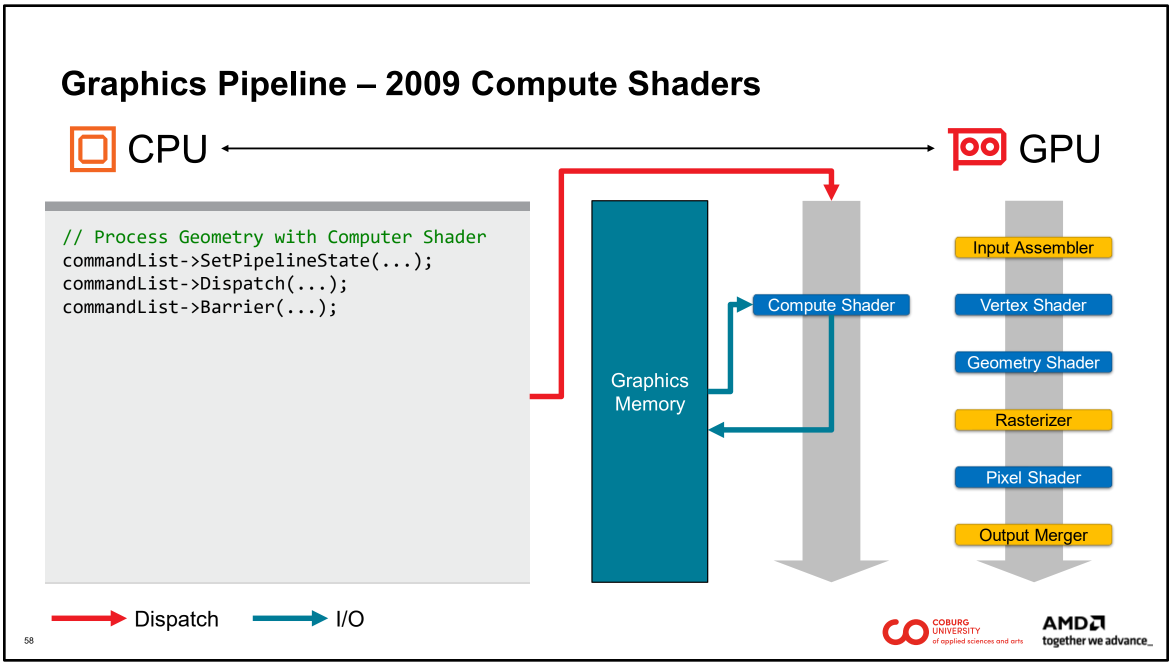
When knowing the thread-group size (in this example 128 threads), `SV_GroupThreadID` and `SV_GroupID` can be used to uniquely identify a thread. We use such a unique ID to index into memory (in this example a `StructuredBuffer`) to perform our computations.



So, we got our fourth concept “Compute Shaders.”

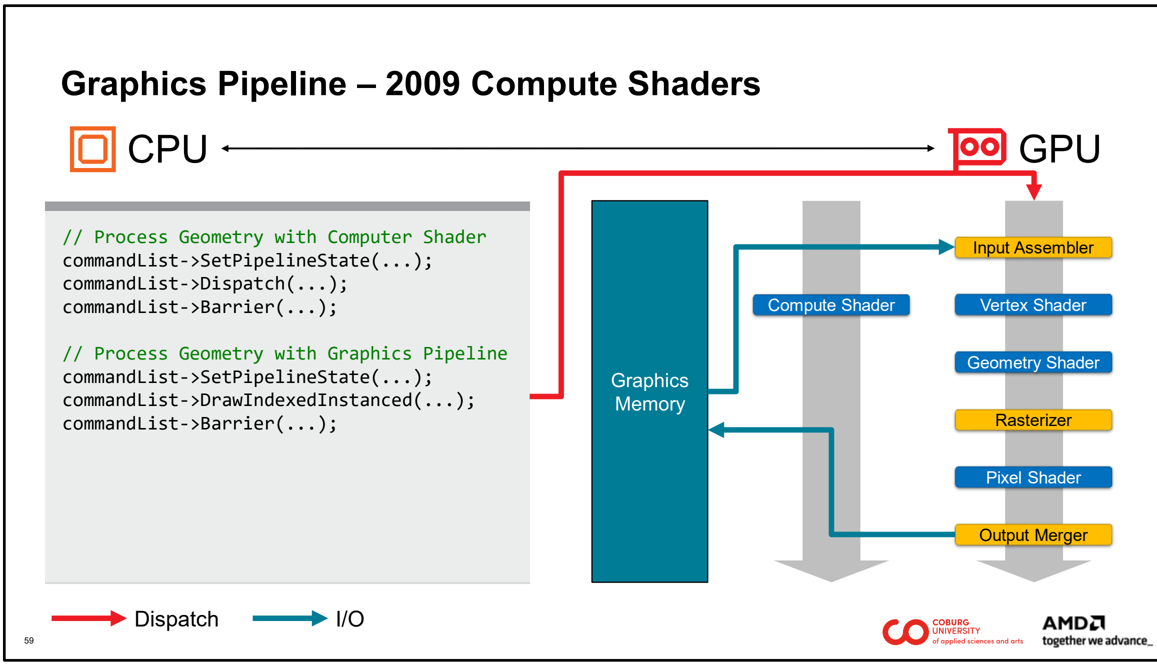


But how do compute shaders interact with the graphics pipeline?

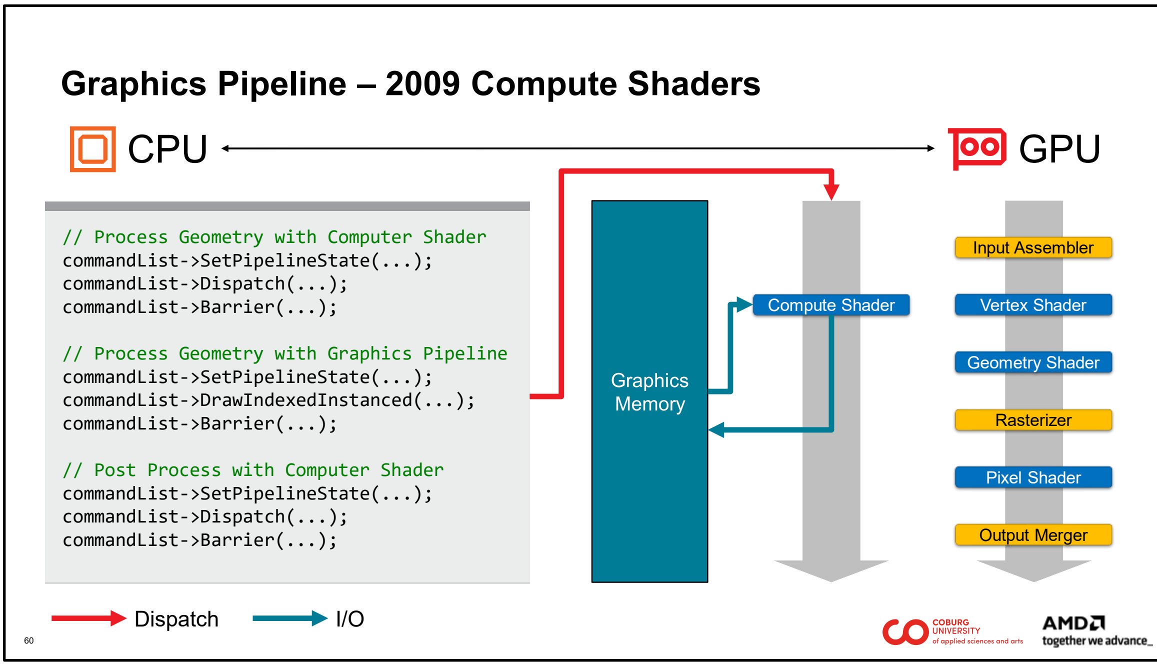


In this example, a compute shader gets dispatched from the CPU. On the GPU, the threads of the compute shader process a list of instances coming from graphics memory.

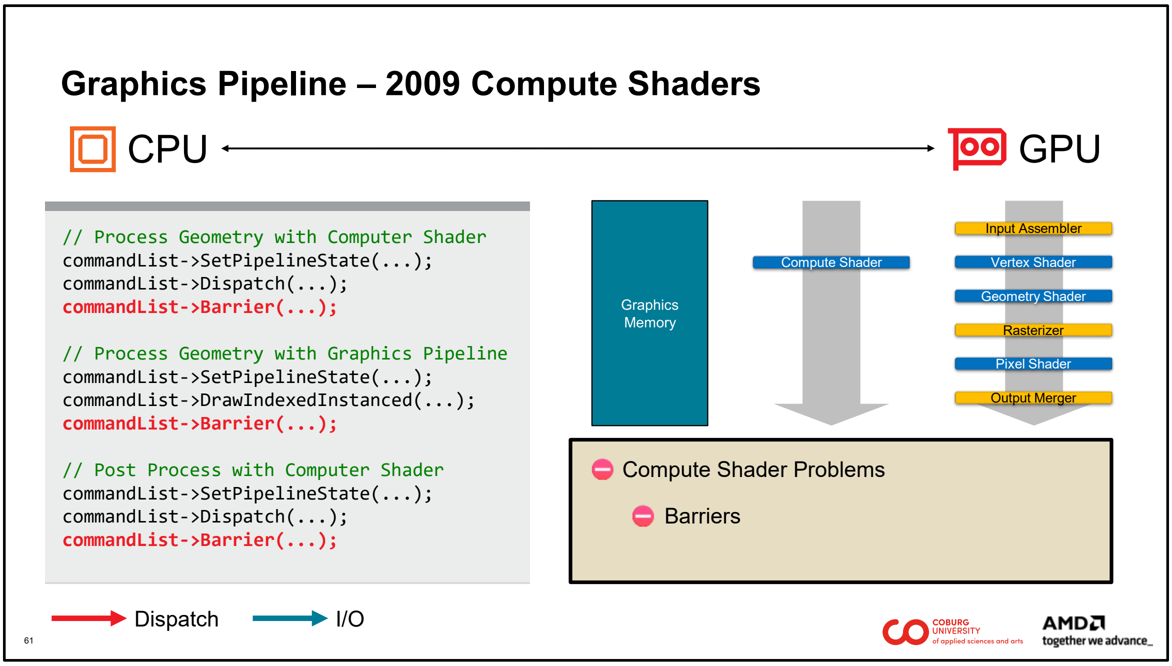
As an example, we assume that the compute shader's task is to cull instances outside the view frustum. The compute shader writes only the visible instances back to graphics memory.



Next, the graphics pipeline renders only the visible instances. It reads them from graphics memory and generates a 2D image. That one is written back to graphics memory.

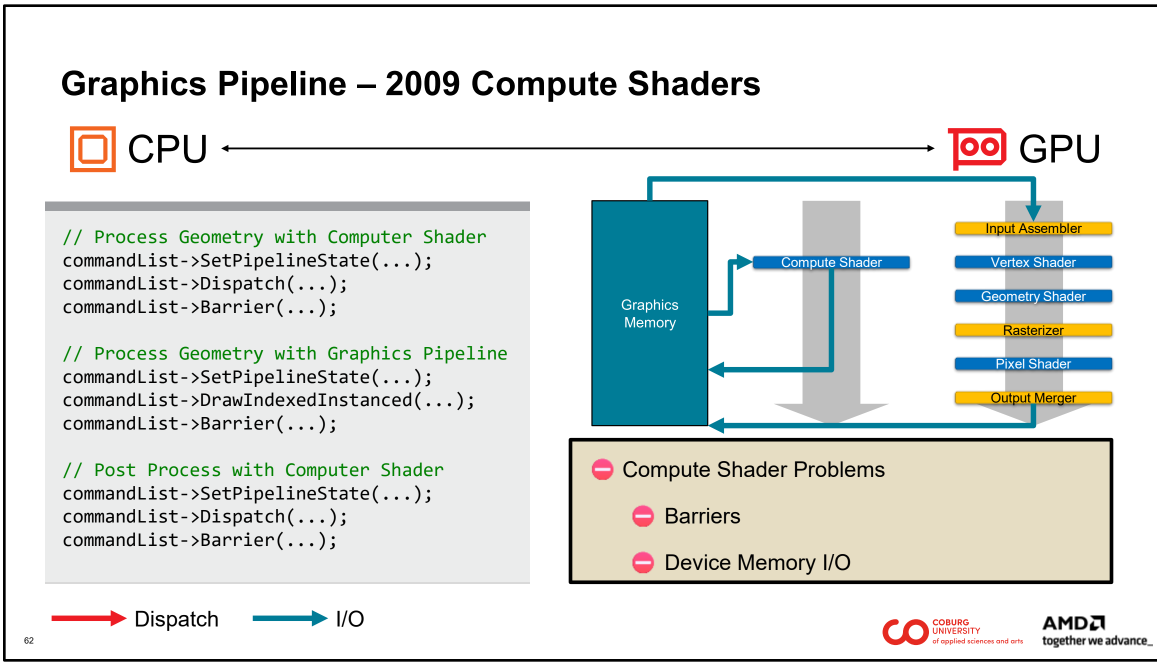


Then, we dispatch another compute shader. This could, for example, do some post-processing on the image. Therefore, we read all the pixels, transform them, and write them back to graphics memory.

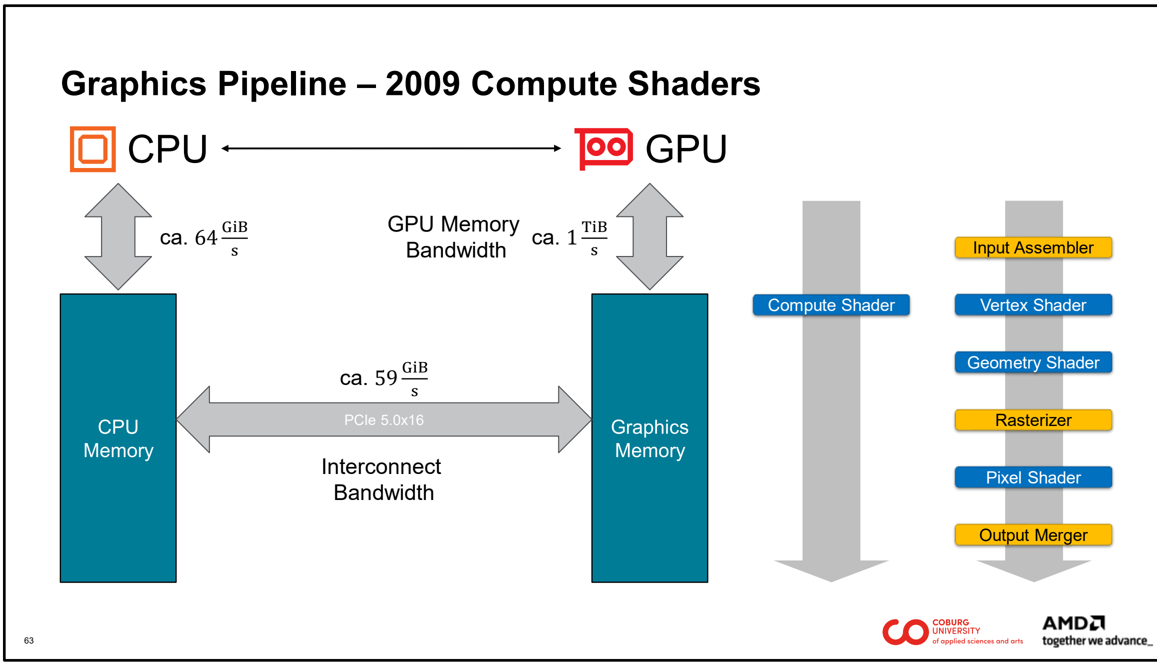


But there are two problems. The first problem is: The *barriers*. They are required to avoid read/write hazards between pipelines. A pipeline must finish its entire computation before any other pipeline can even start. This is assured by barriers.

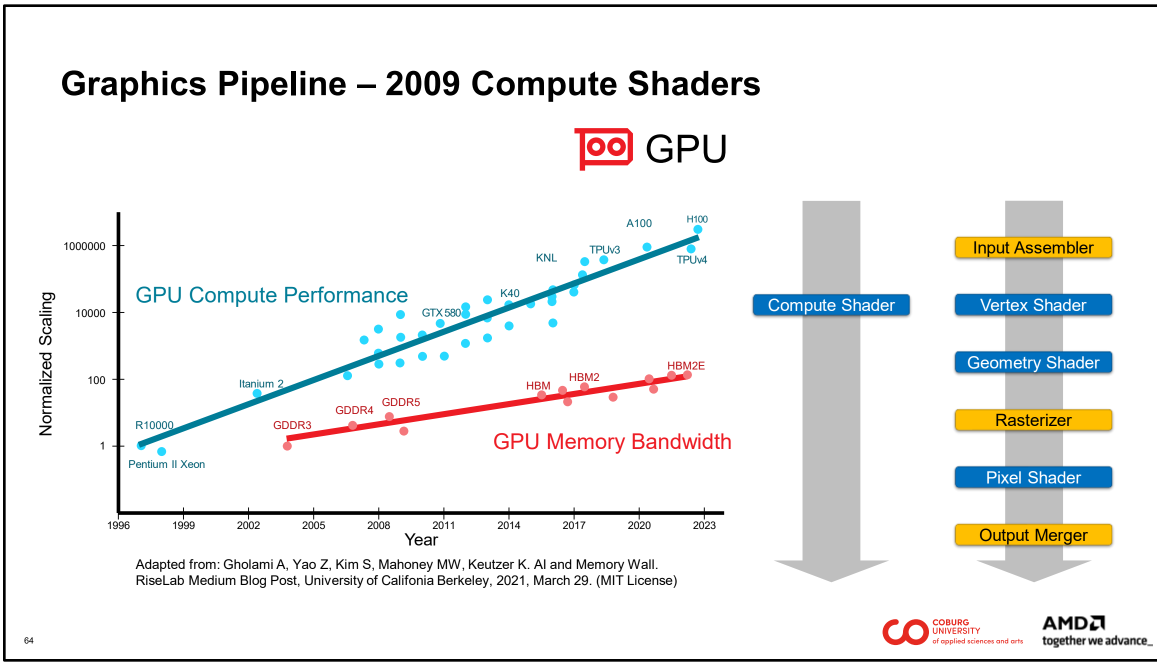
This can leave many work group processors idle, especially when a pipeline computation is about to finish.



The second problem is: The *communication* between pipelines happens over graphics memory, which can become a limiting factor.



Here are some numbers: In comparison to other memory buses we have in our system, 1 TiB/s between the work groups processor of a GPU and graphics memory seems huge.

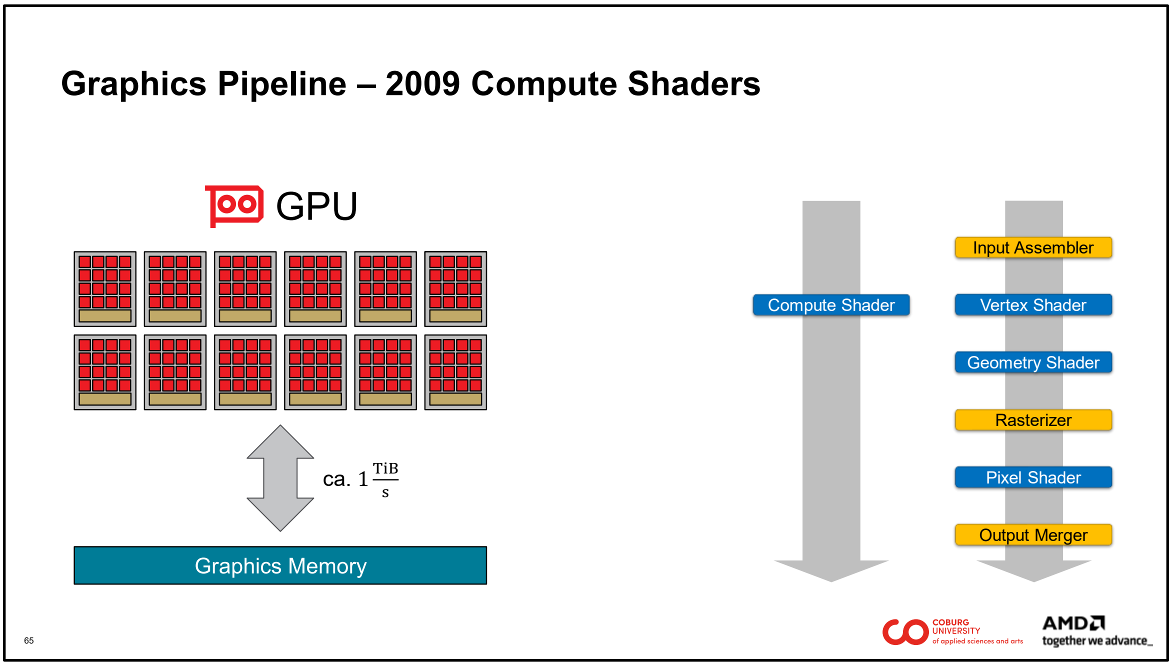


However, as we can see in this plot, over the last years, the growth in *GPU Compute Performance* has outpaced the growth in *GPU Memory Bandwidth*.

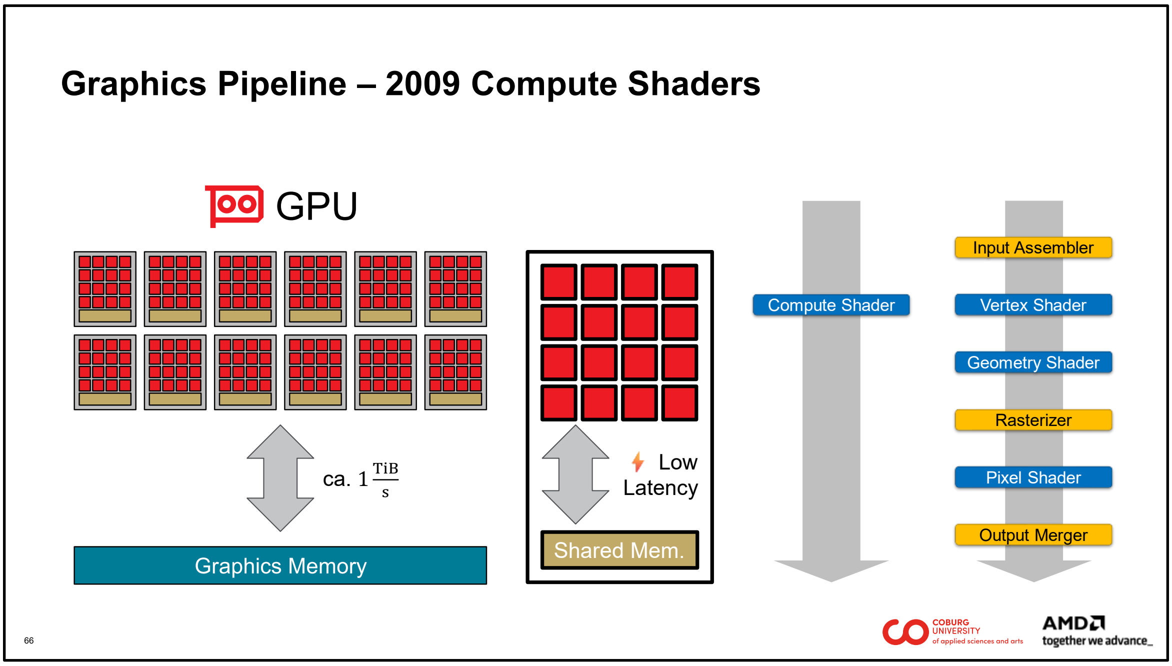
Source: Image adapted from https://github.com/amirgholami/ai_and_memory_wall/blob/main/imgs/pngs/hw_scaling.png

From the paper: Gholami A, Yao Z, Kim S, Mahoney MW, Keutzer K. AI and Memory Wall. RiseLab Medium Blog Post, University of California Berkeley, 2021, March 29.

Available on https://github.com/amirgholami/ai_and_memory_wall/blob/main/README.md

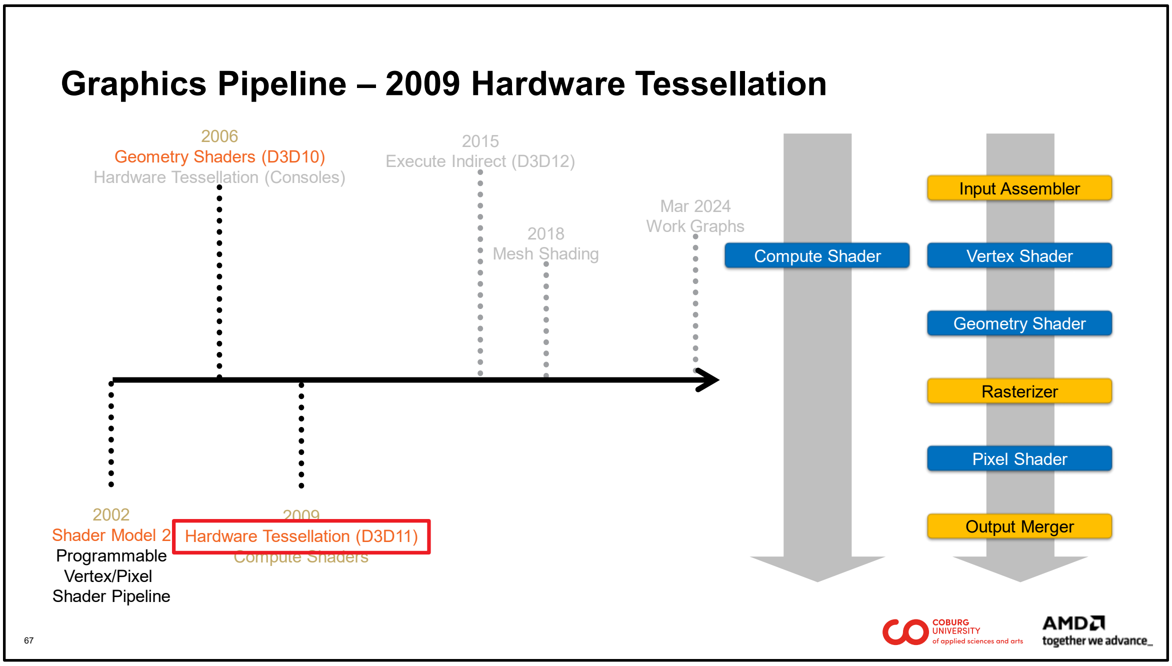


So even with a bandwidth of 1 TiB/s to graphics memory…

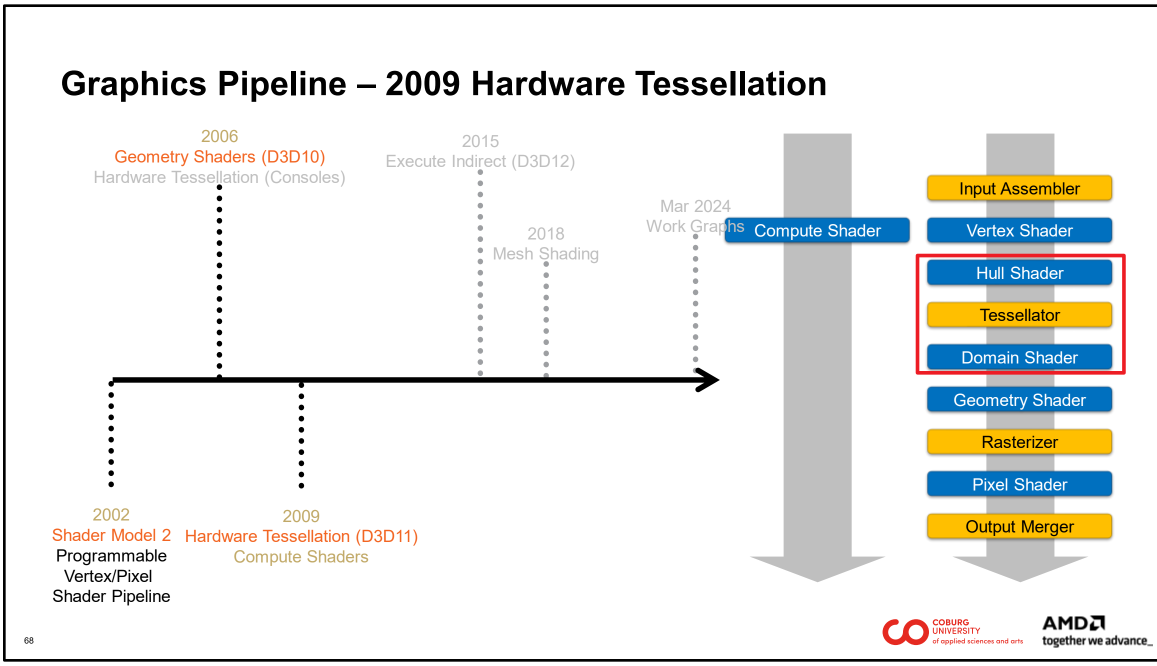


… inside the cores, we have the shared memory which is much faster. In fact, it has a very low latency compared to graphics memory.

However, it is much smaller in memory capacity.

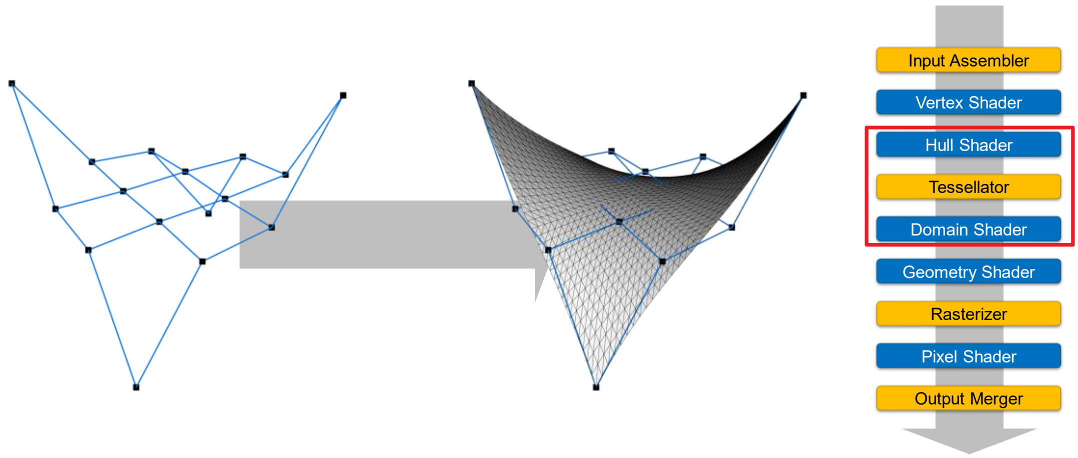


So back in 2009, I/O was, of course, already a problem. To save I/O, *hardware tessellation* was introduced in 2009.



It contains two more programmable stages, *hull shader* and *domain shader*, and a fixed-function hardware *tessellator*.

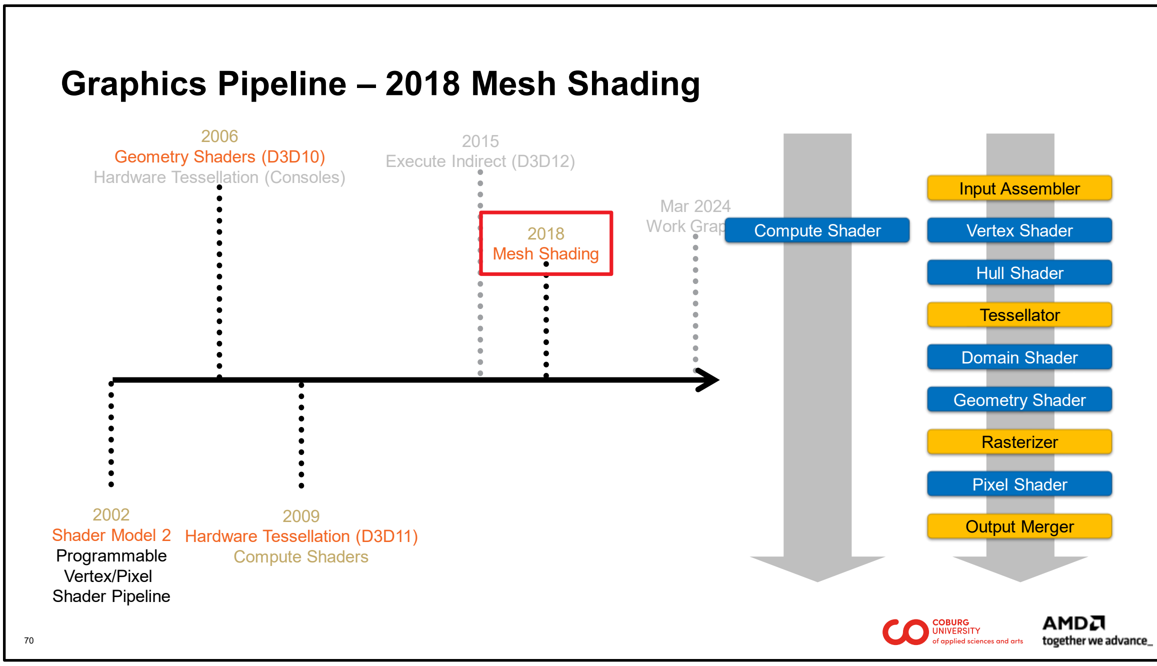
Graphics Pipeline – 2009 Hardware Tessellation



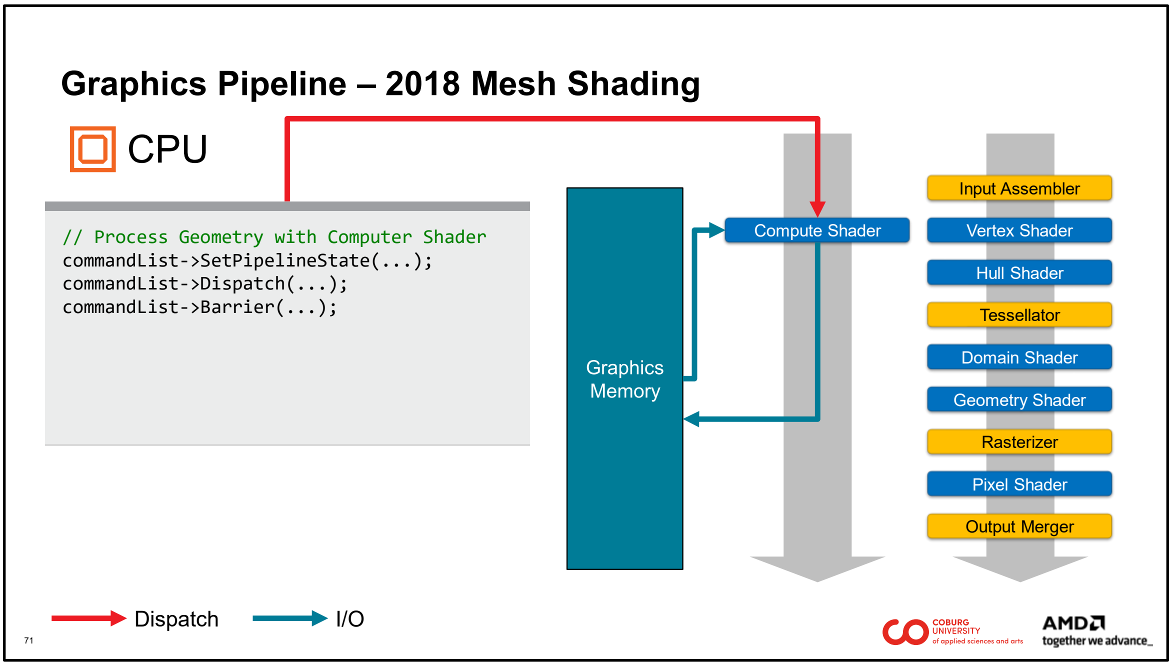
69

COBURG UNIVERSITY of applied sciences and arts AMD together we advance_

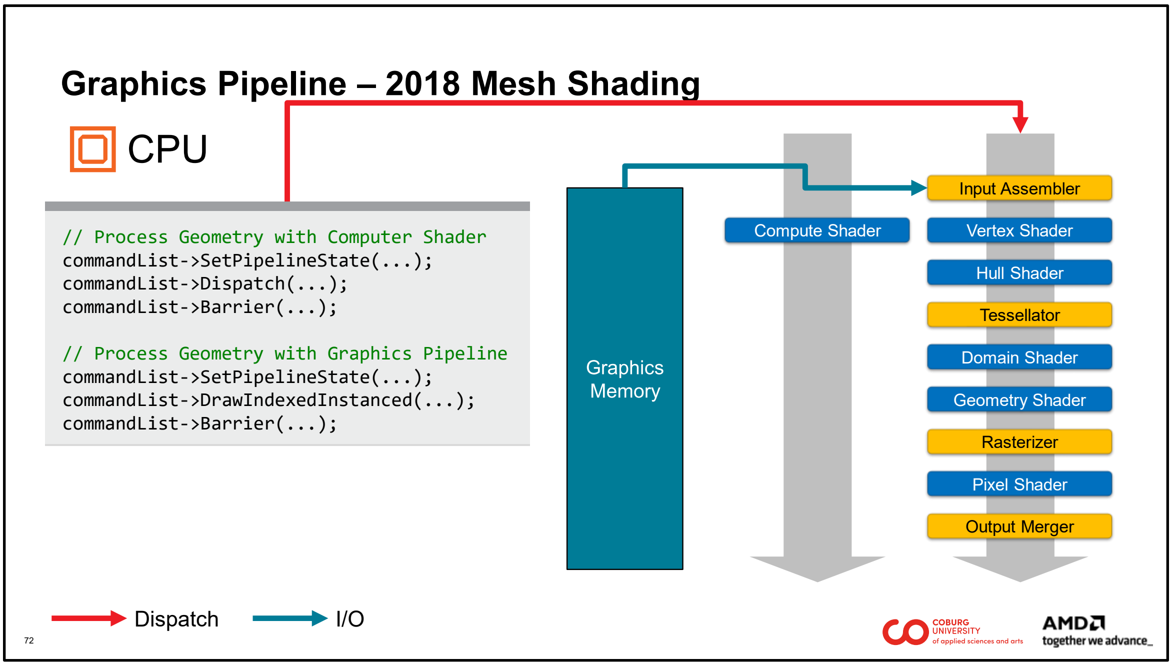
Hardware tessellation allows to amplify geometry from a couple of control points to a larger number of triangles. But the rather rigid tessellation patterns do not offer the desired degree of freedom on topology.



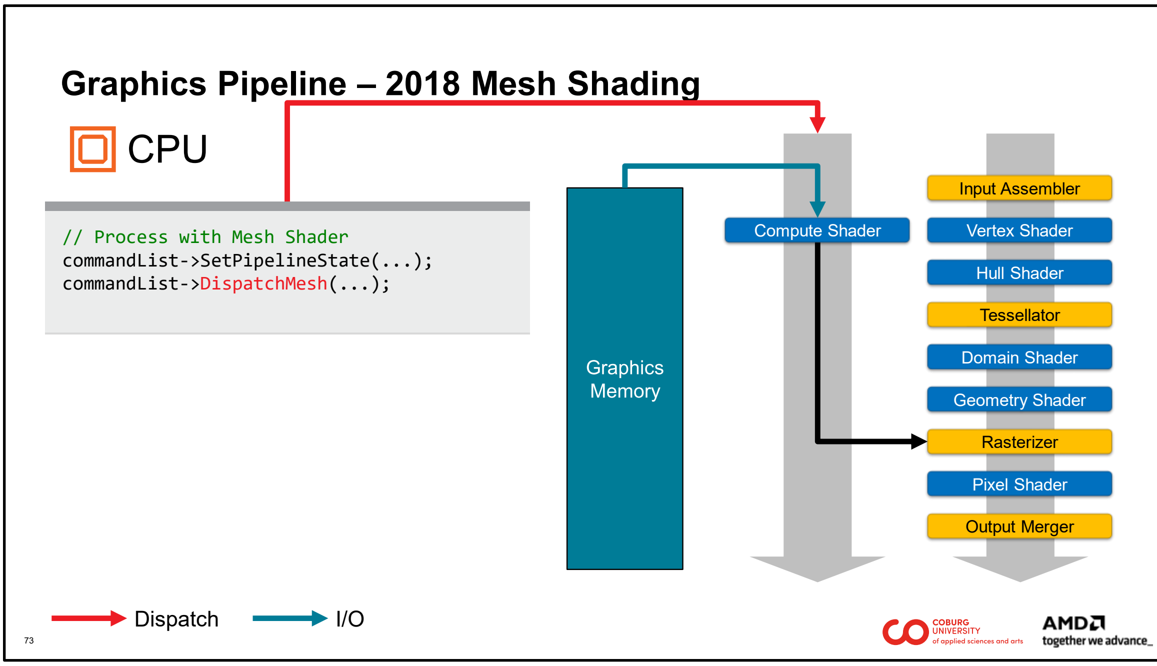
This is why in 2018, *mesh shading* was added to the pipeline. Mesh shading is important for Work Graphs, too.



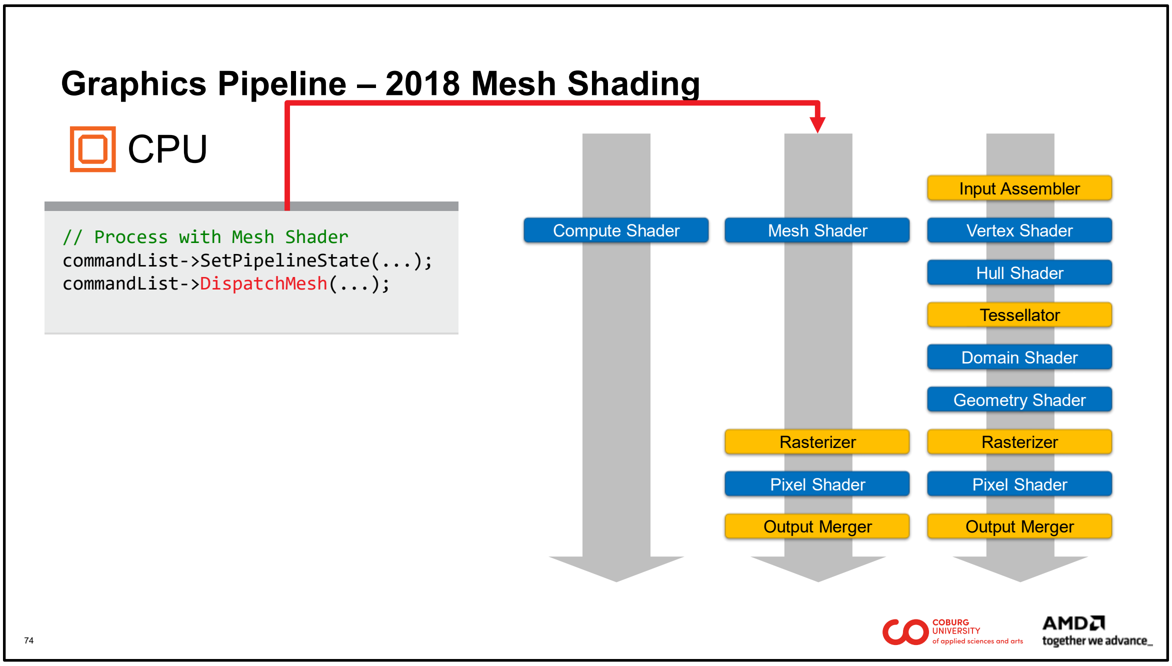
So, what problem does mesh shading solve? Consider a compute shader that creates or transforms geometry. However, the compute shader must write its output to graphics memory.



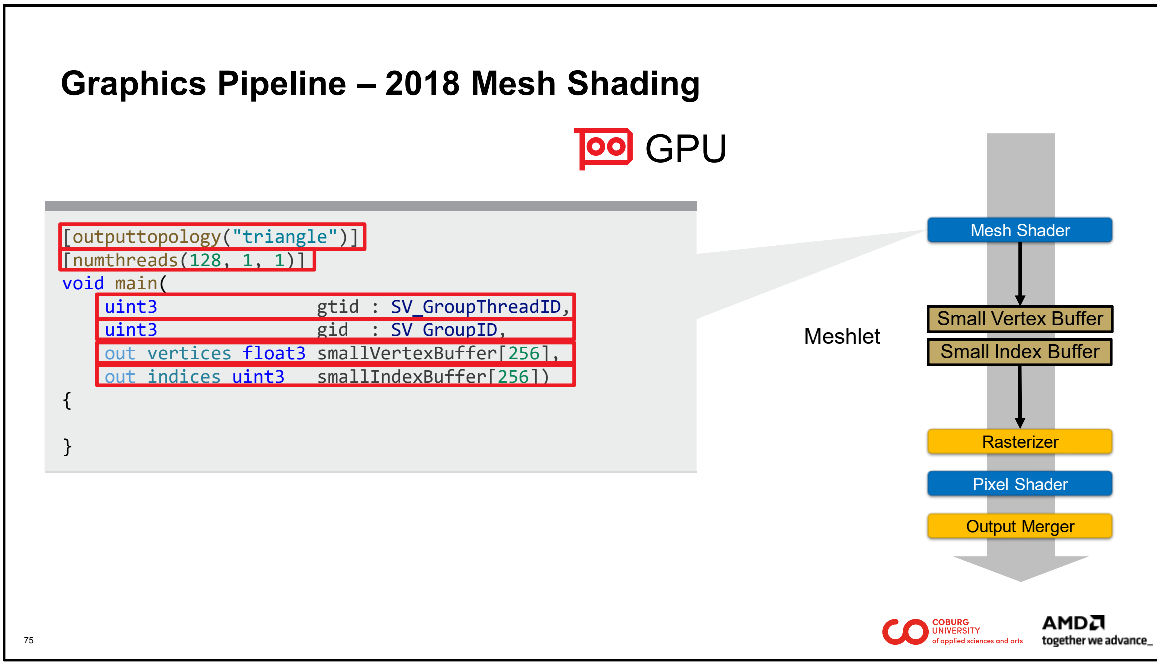
Then, the graphics pipeline can read from graphics memory. Therefore, we have one memory write and one memory read, which we could save. Remember, graphics memory access is rather expensive.



The idea of mesh shading is to directly feed the rasterizer from the compute shader. This saves the extra graphics memory access.



This gives us a third pipeline: the *Mesh Shading Pipeline*.

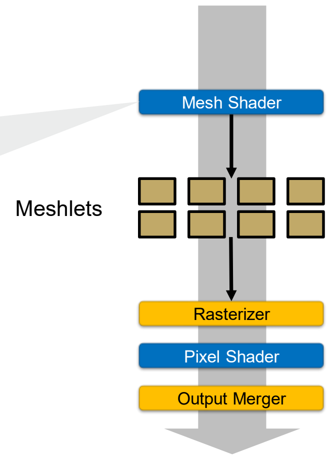


Like a compute shader, you launch a grid of mesh shader thread groups. So, in the code, we have our `SV_GroupThreadID` and `SV_GroupID` semantics. Each mesh shader thread group can have up to 128 threads. We can output triangles (or other primitives) to a small vertex and index buffer with up to 256 vertices and triangles each. A mesh shader output is like a small mesh. Therefore, it is commonly called a *meshlet*.

Graphics Pipeline – 2018 Mesh Shading

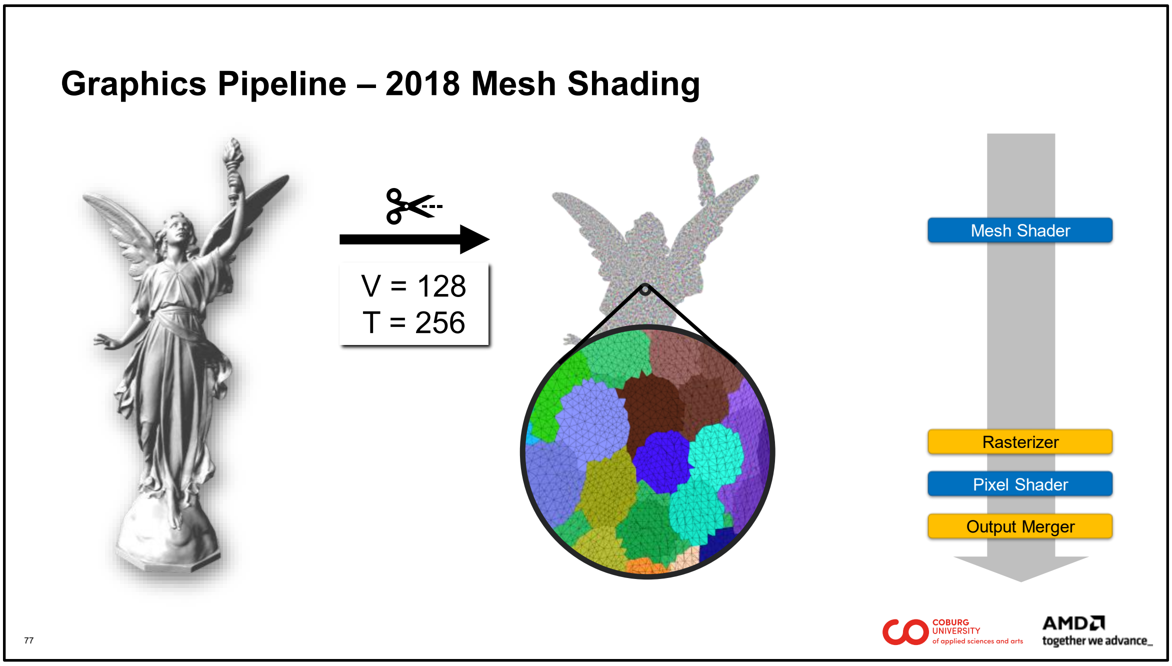
GPU

```
[outputtopology("triangle")]
[numthreads(128, 1, 1)]
void main(
    uint3                gtid : SV_GroupThreadID,
    uint3                gid  : SV_GroupID,
    out vertices float3 smallVertexBuffer[256],
    out indices uint3   smallIndexBuffer[256])
{

}
```



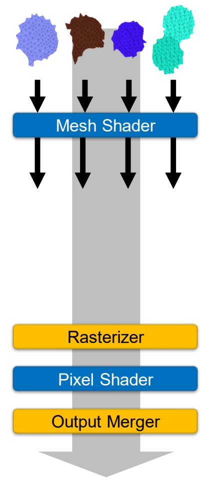
76

With multiple mesh shader thread groups, we can output multiple meshlets.



If you want to render a larger model, you first decompose it into multiple meshlets in a preprocess.

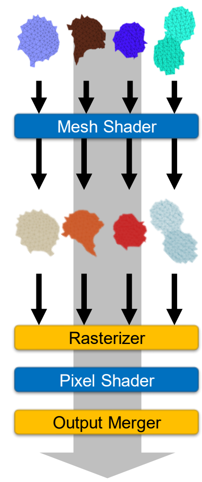
Graphics Pipeline – 2018 Mesh Shading



78

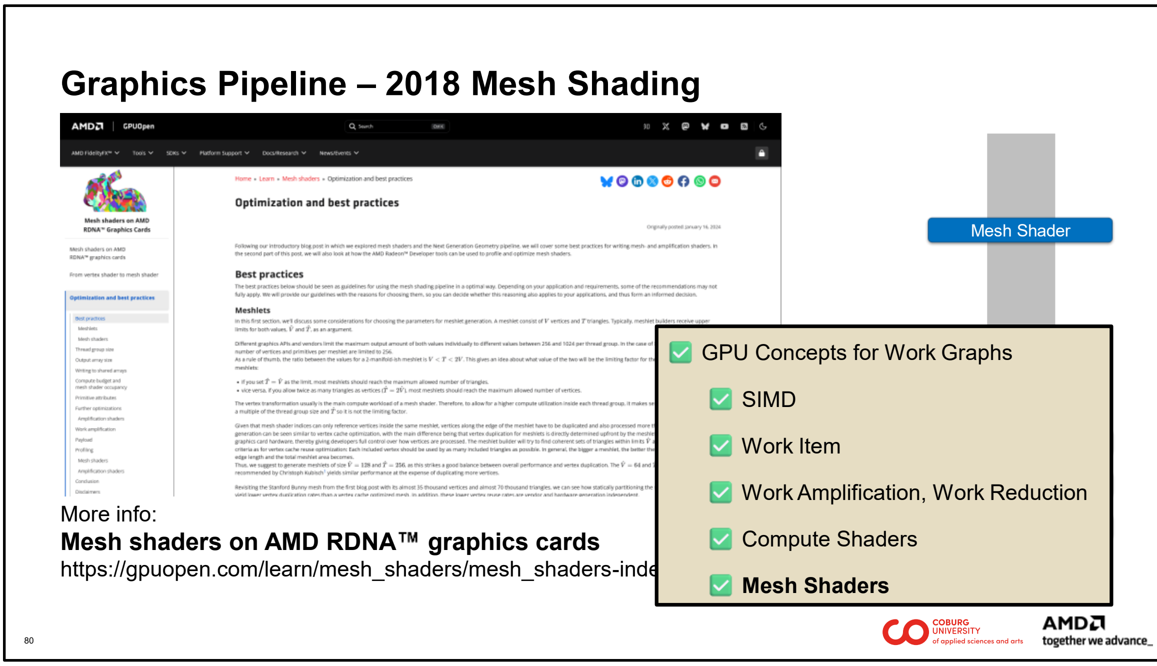
And then run a mesh shader thread group for each meshlet.

Graphics Pipeline – 2018 Mesh Shading



79

The mesh shader thread groups then transform these meshlets and pass them over to the rasterizer.



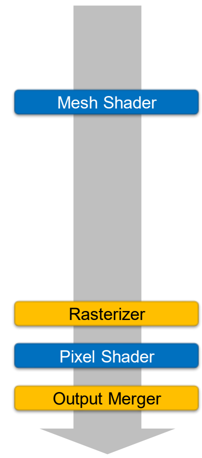
Mesh shading is a super light-weight version of work graphs.

For more information see this blog post series: https://gpuopen.com/learn/mesh_shaders/mesh_shaders-index/

Graphics Pipeline – 2018 Mesh Shading

✅ GPU Concepts for Work Graphs

❓ Why Work Graphs?

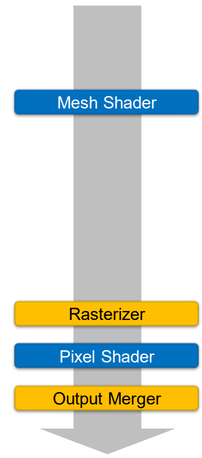


81

But now that we have mesh shading, we have all concepts together that we need for work graphs.

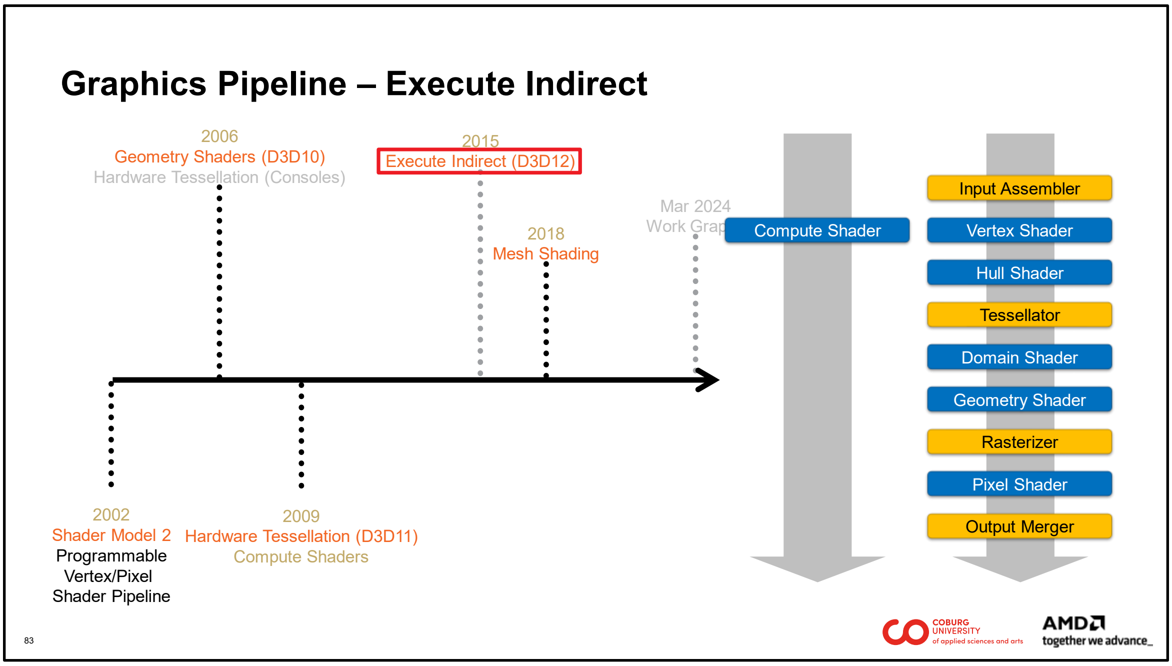
Graphics Pipeline – 2018 Mesh Shading

❓ Why Work Graphs?



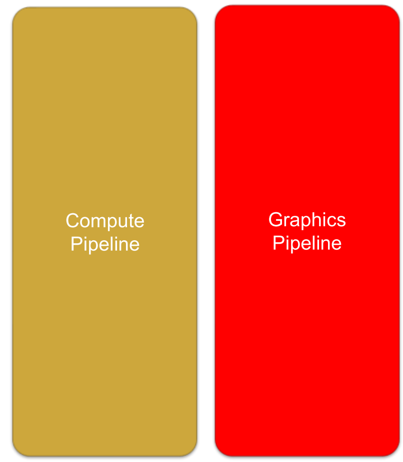
82

The question now is: why do we even need Work Graphs?



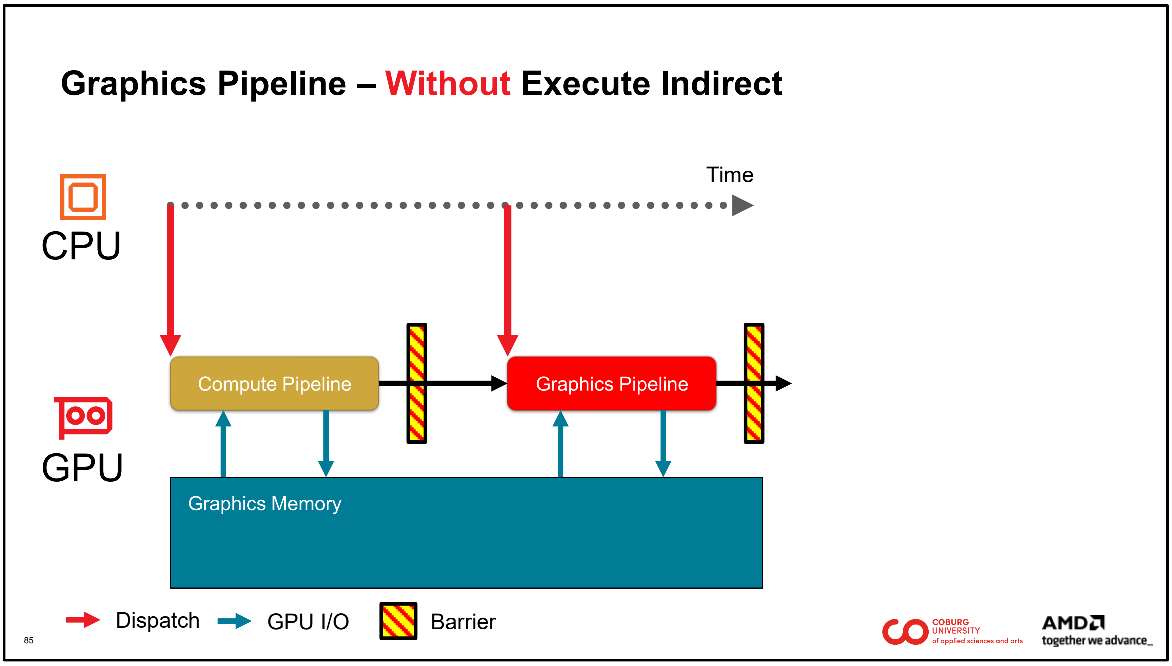
To answer that question, let's first look what another addition to the pipelines attempt to solves: I am speaking of "*execute indirect.*"

Graphics Pipeline – Execute Indirect



84

Suppose you have a compute pipeline and a graphics pipeline.



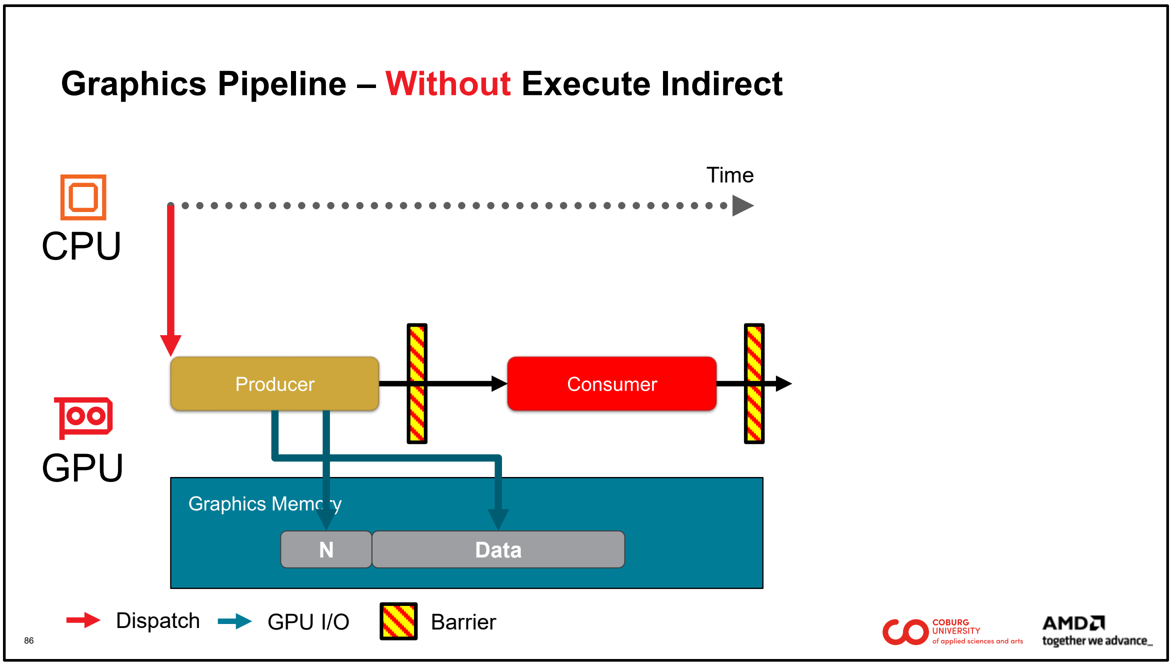
And you kick of your compute pipeline from the CPU.

The GPU then does some computation using the compute pipeline. To that end, it reads data from graphics memory and writes its results back to graphics memory.

To make sure that everything is written into graphics memory, we must include a barrier.

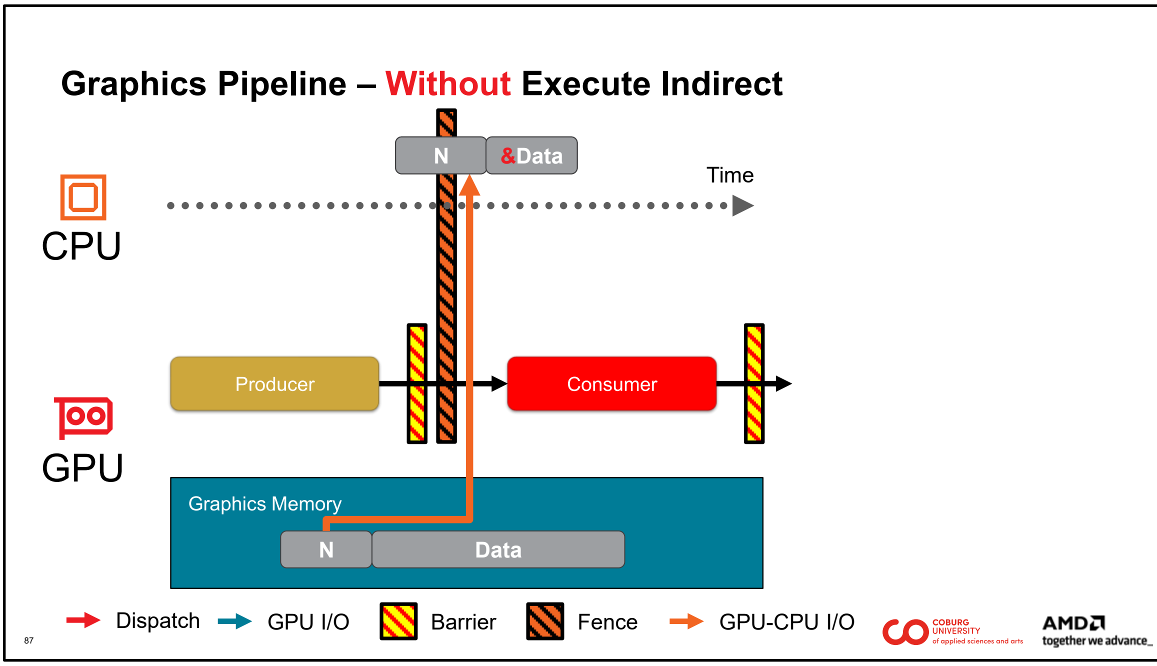
Only after we have reached the barrier, we can kick off the graphics pipeline.
So, we must wait. The graphics pipeline can then read the data from memory and produce the pixels output. After that we need another barrier.

These barriers can become a severe performance problem, because your system must wait actively.



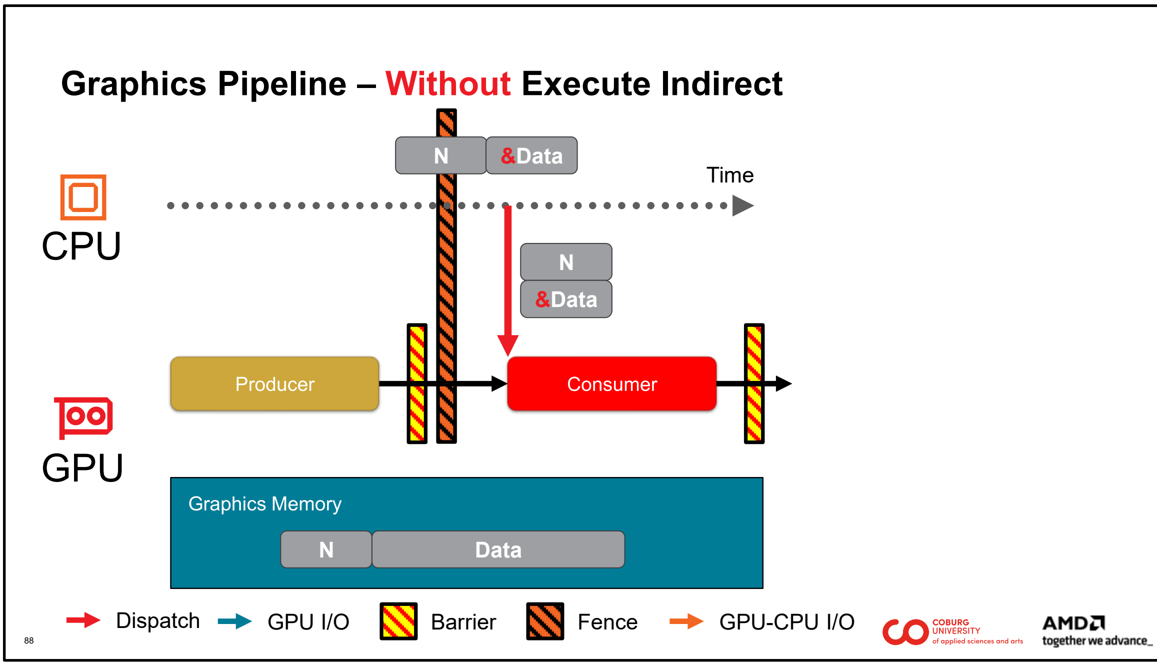
The situation gets even more severe when the producer (i.e., the compute kernel) produces a varying number of data entries.

As an example, imagine a scene with tens of thousands of objects. The task of the producer is to cull invisible objects. After the producer kernel has run, it outputs 5000 visible objects to data. There, it writes N = 5000.



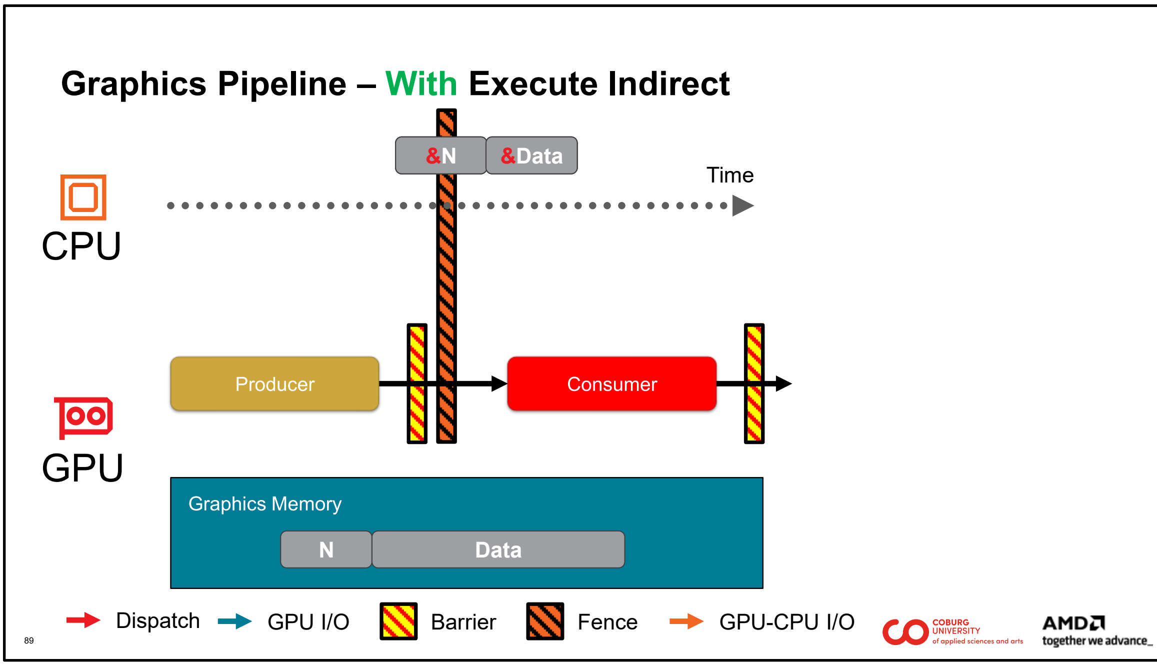
The consumer then renders the 5000 objects. But to do so, the CPU must configure the draw call and it must know that number N.

So, the CPU must read N from the GPU. Therefore, we must include a fence that synchronizes CPU and GPU. Only after that fence can the CPU read the number N and properly configure the draw call.

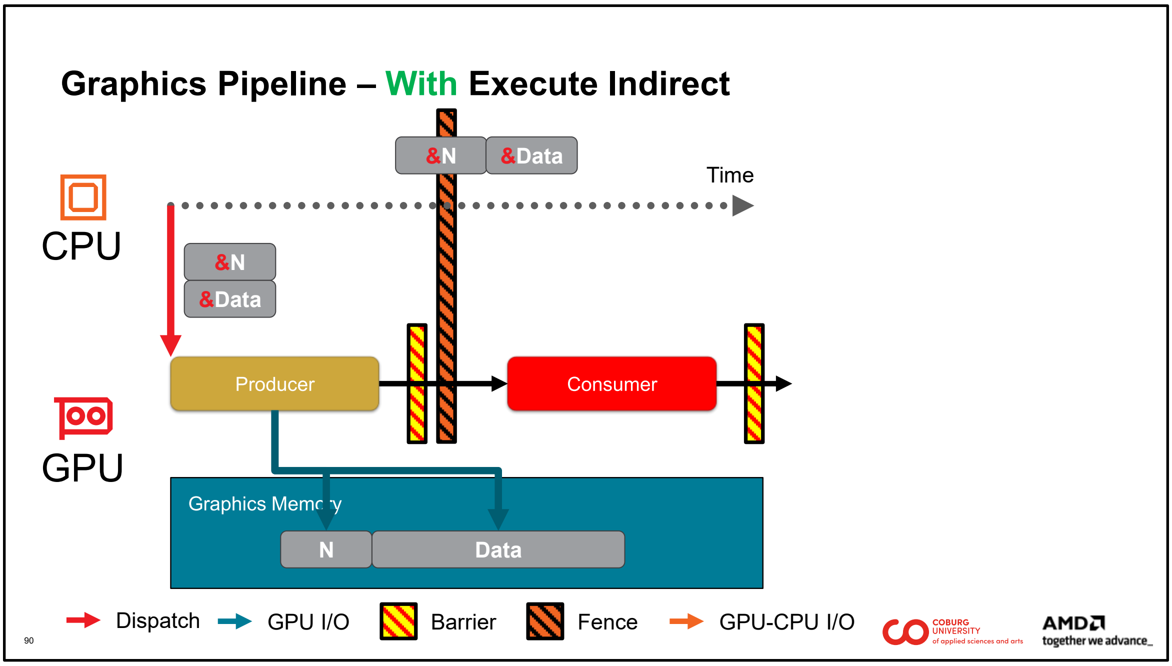


With that handle to the data and the number of objects, the CPU can dispatch 5000 draw calls to the visible objects.

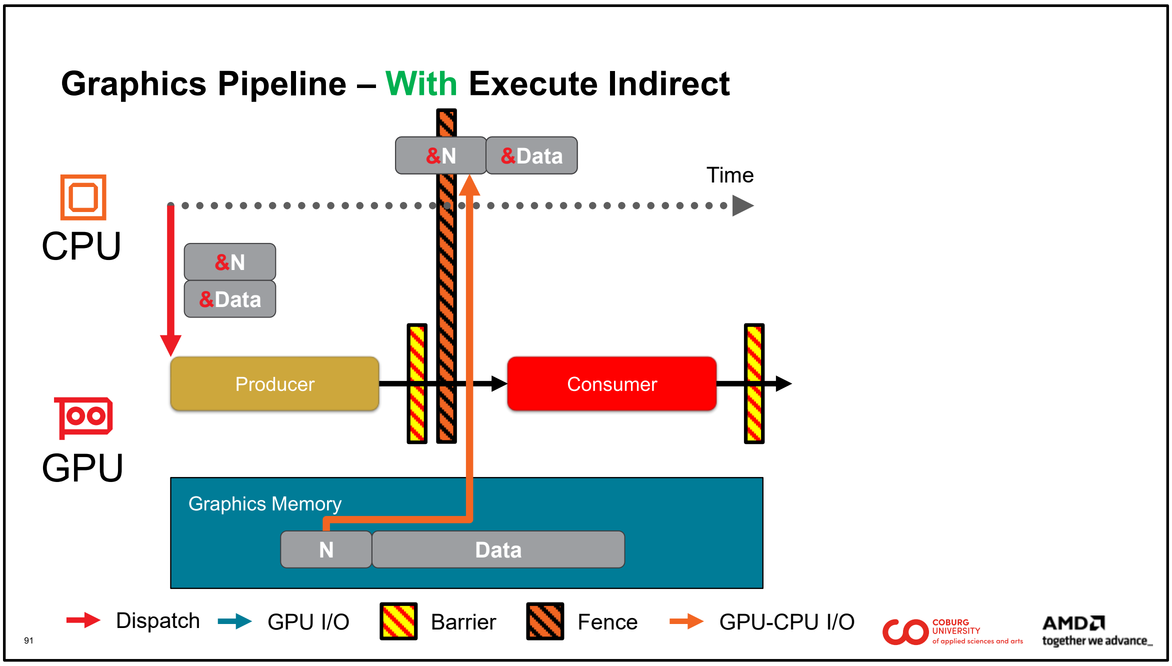
Note that producer and consumer need to agree up-front on the handle to data (&Data).



With “execute indirect”, we also get a handle to where the number N is stored. Let’s see why that can improve things.

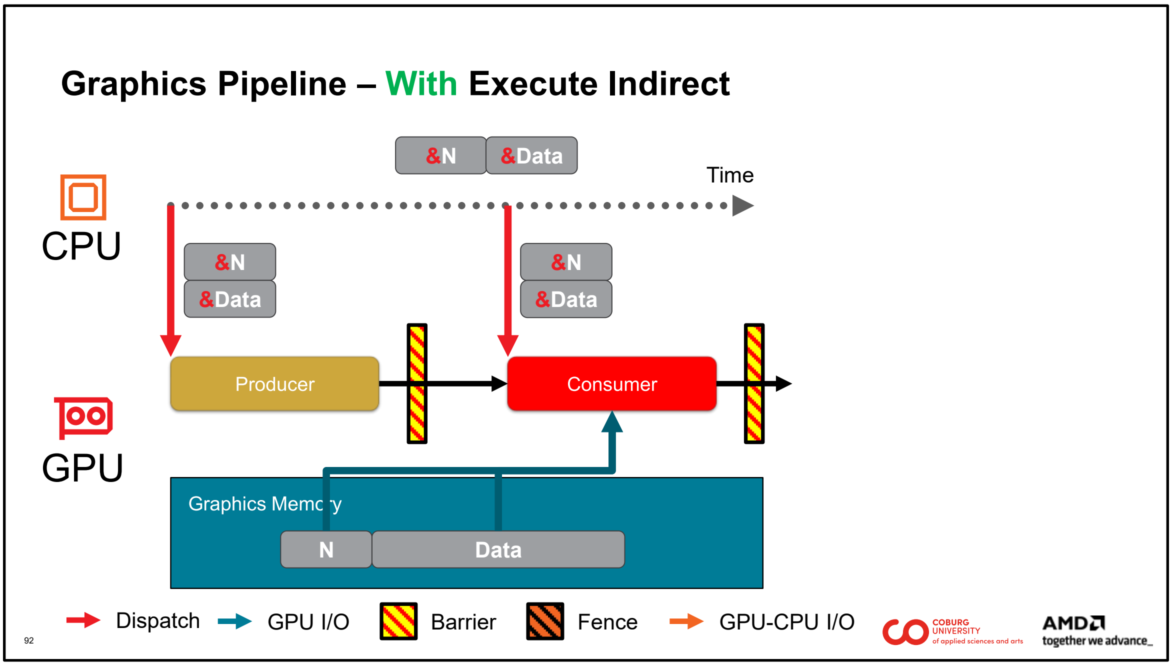


The producer gets both handles: &N and &Data. As before, it writes out the visible objects (Data) and the number of visible objects (N).

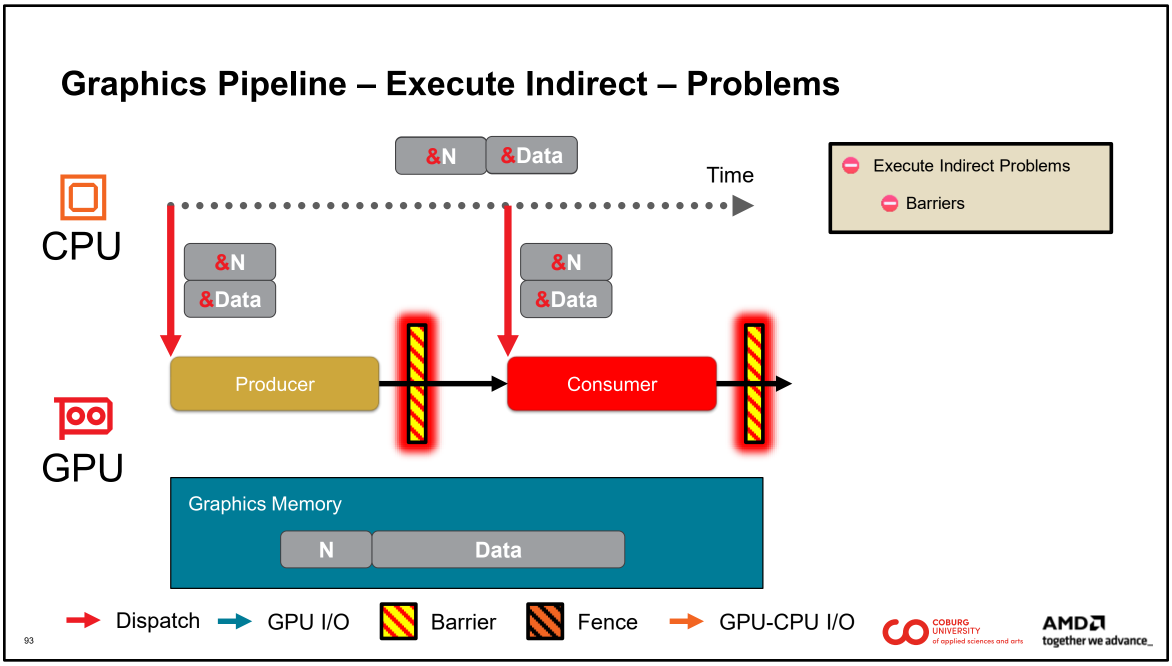


Now the CPU knows the location of the handles on the GPU but not the actual values behind it. So, there is no need to transfer the actual values.

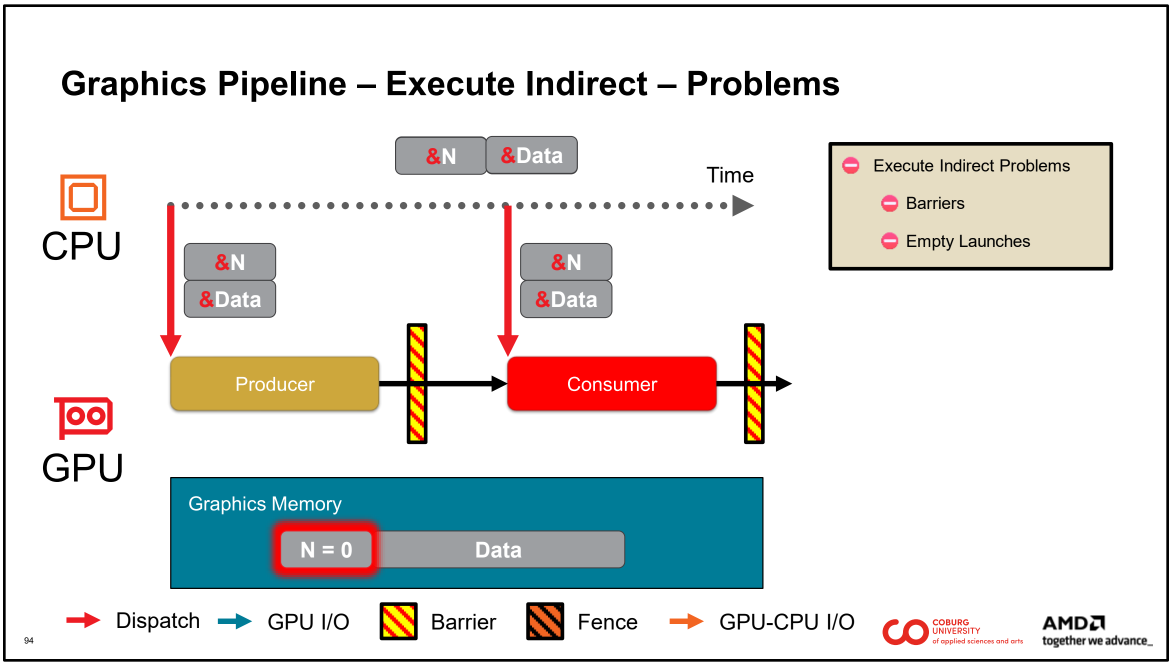
And therefore, no need for fence.



All the CPU needs to do is call the consumer with handles as parameters instead of the actual values.

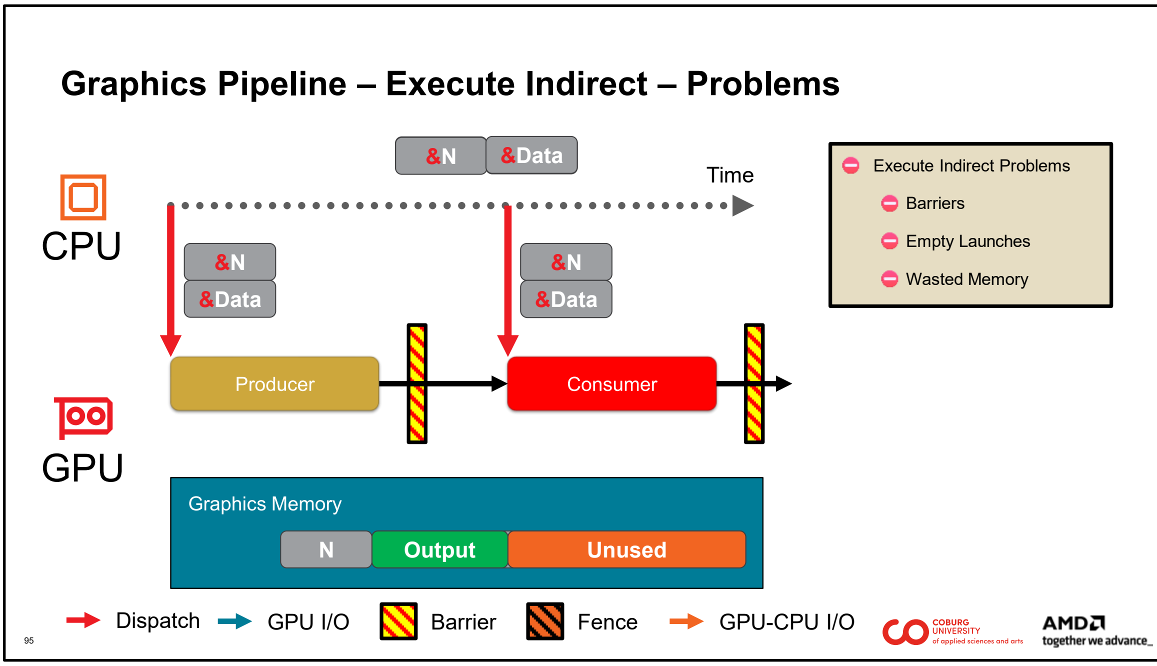


However, we still need the barriers, since the producer and consumer still communicate over graphics memory.

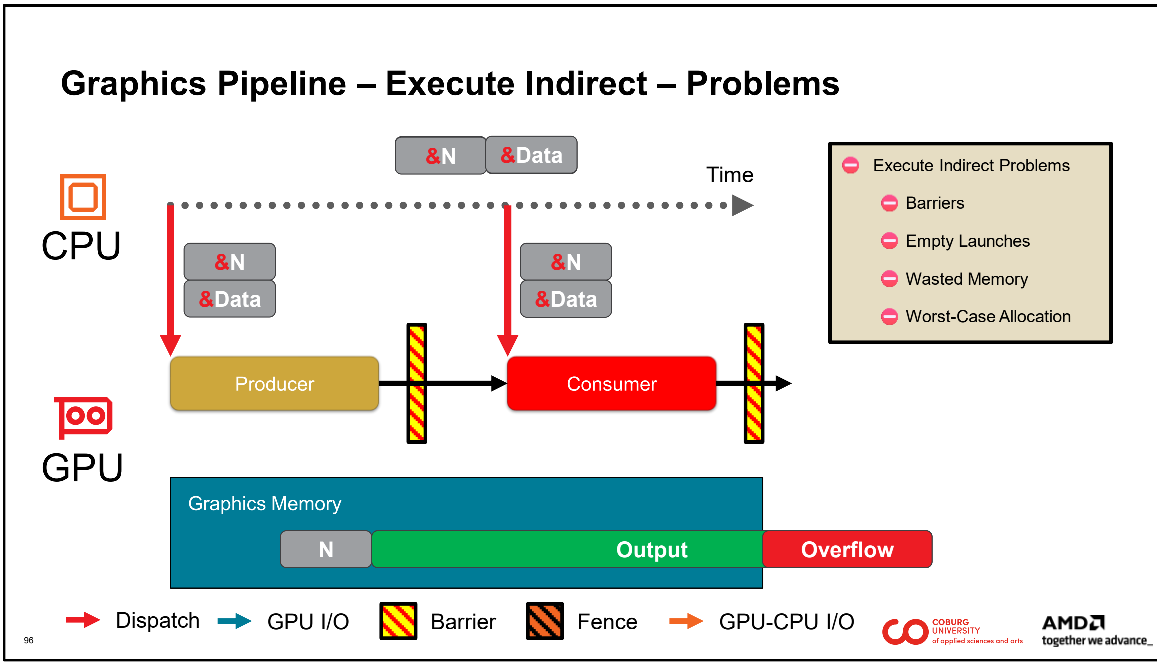


Additionally, if N = 0, there would not be any reason for the CPU to dispatch the Consumer. But the CPU has no idea about N being 0, so it must dispatch the draw call no matter what.

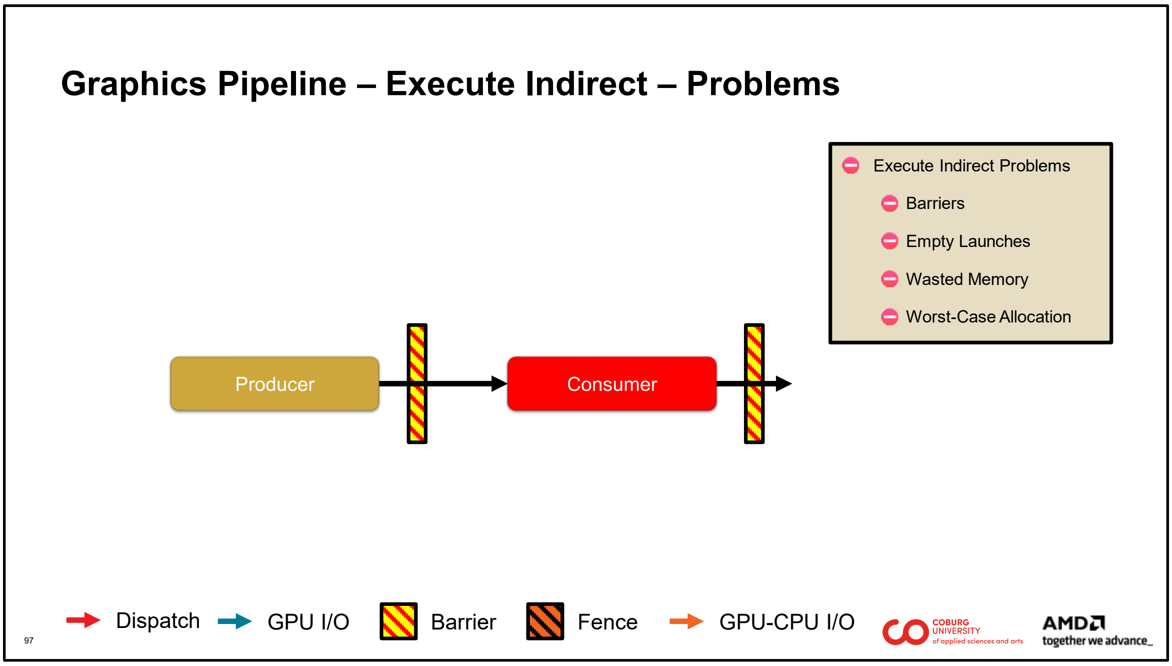
That is not dangerous, but we have the overhead of a launch including the barrier.



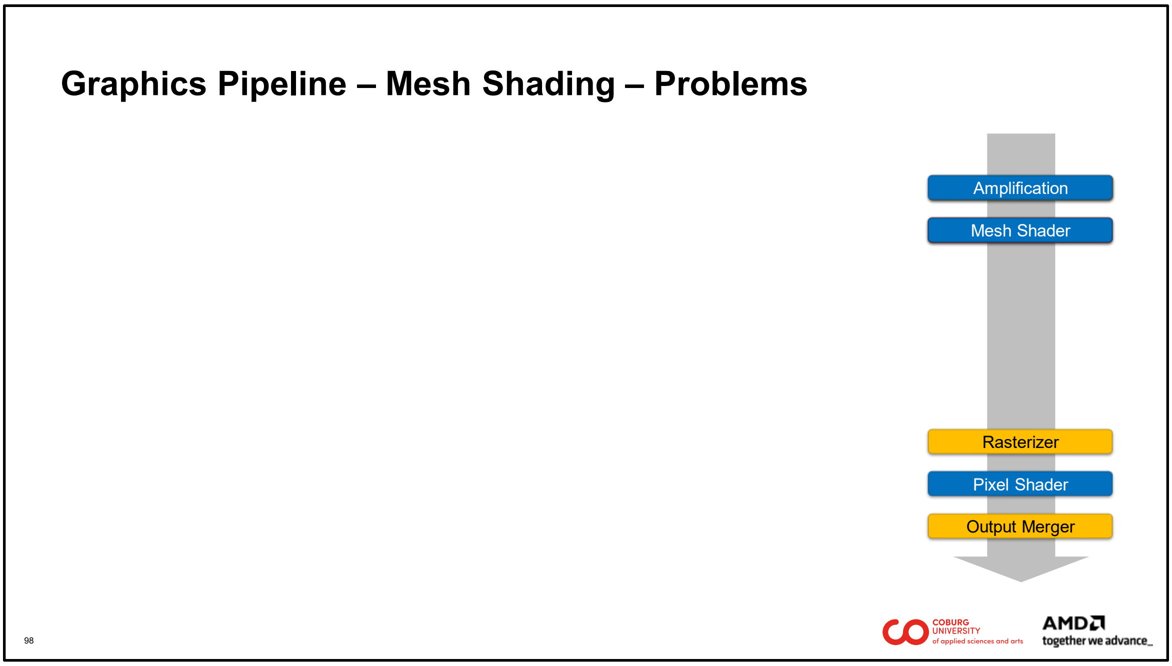
Another problem with execute indirect is, that we have to reserve memory for what could end up in data. Going back to the culling example, we could end up rendering all objects or zero objects. Since we do not know that up front, we must always be prepared for the worst case and thus potentially waste memory.



We must always account for the worst-case scenario. If not, we could run into dangerous memory overflow situations.

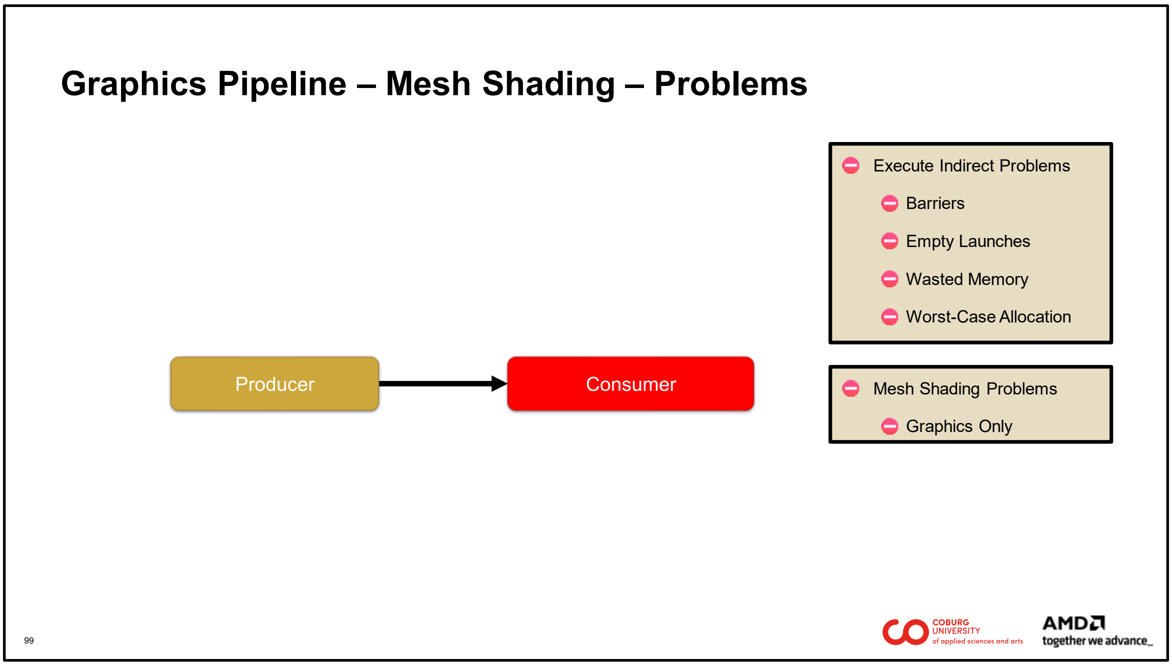


So those are all existing execute-indirect problems.

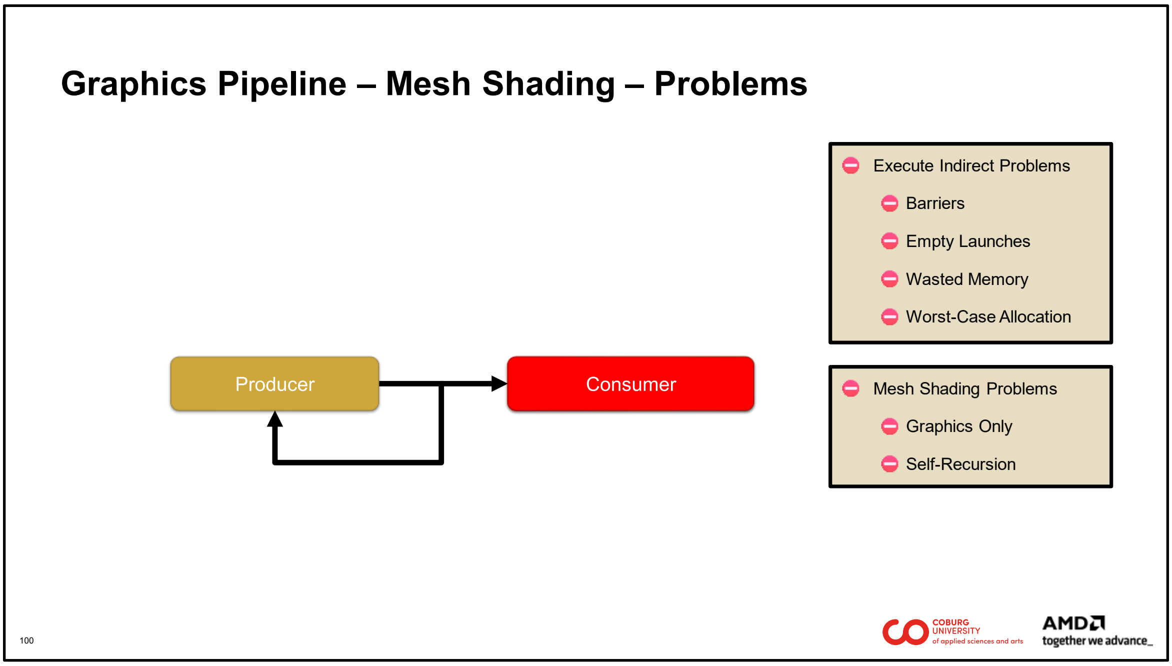


Could we solve those with mesh shaders? I have not yet mentioned the *amplification shader* stage of the mesh-shading pipeline.

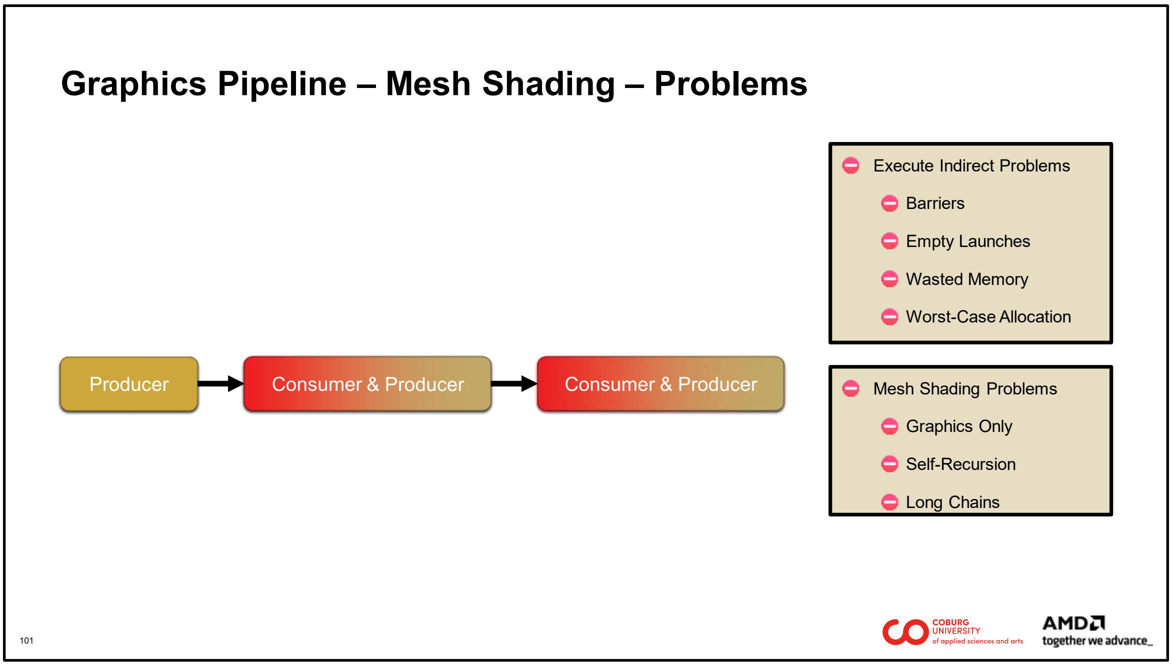
An amplification shader can control the number of mesh shader thread groups to launch directly on the GPU. In essence, this is a little consumer-producer pipeline. So, for very simple scenarios, mesh shading, can solve some of the issues.



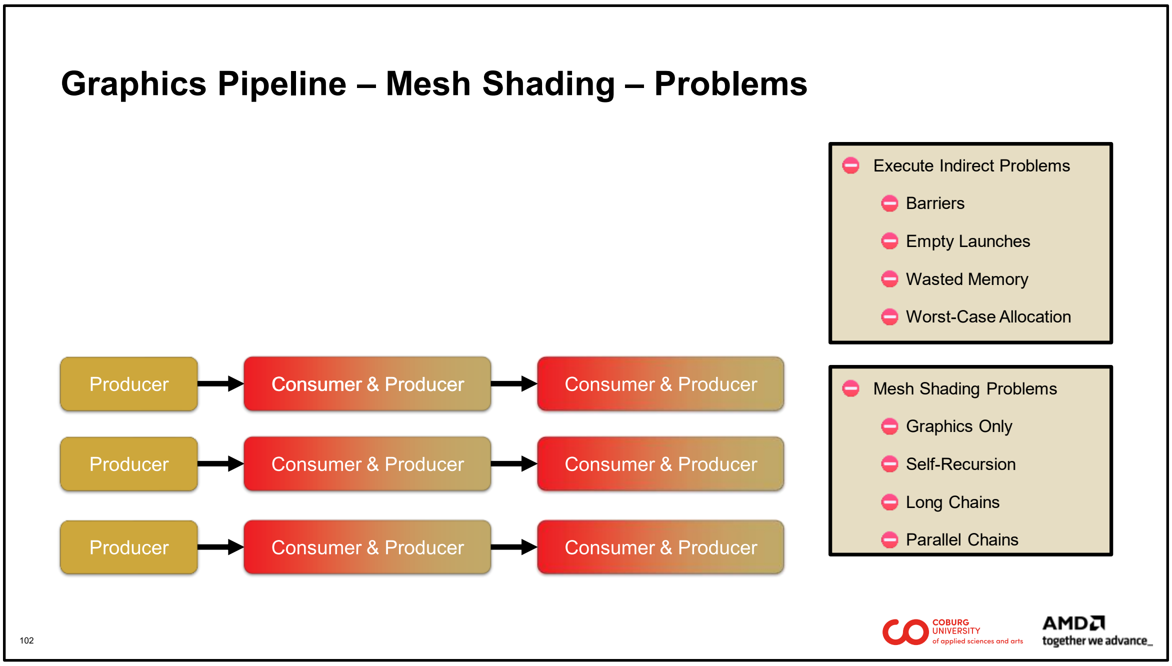
But mesh shading is graphics only. It has no compute support.



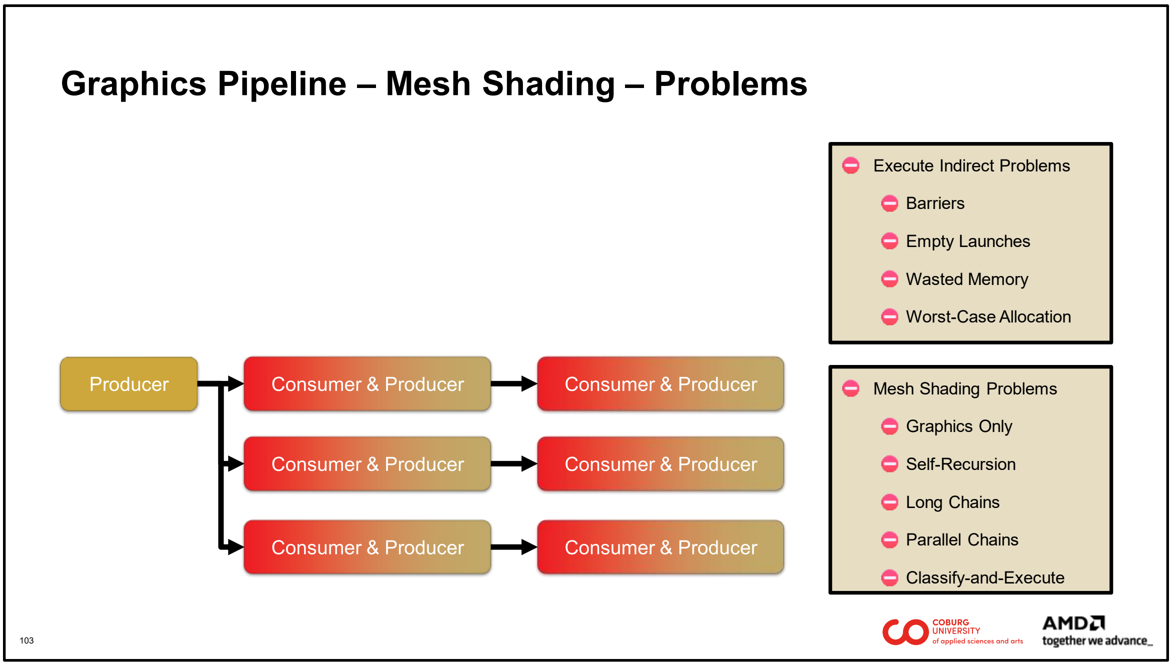
It breaks, if you want something like self-recursions, as for example with recursive subdivision.



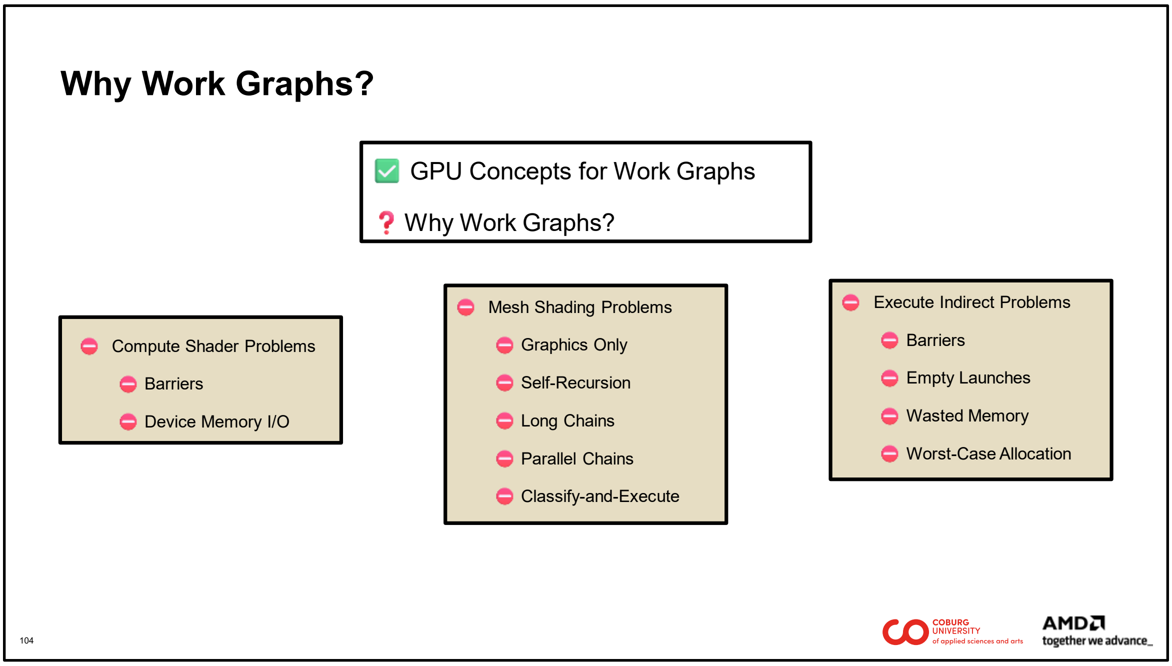
The mesh-shading pipeline has only one or two programmable stages. Long chains are therefore not possible…



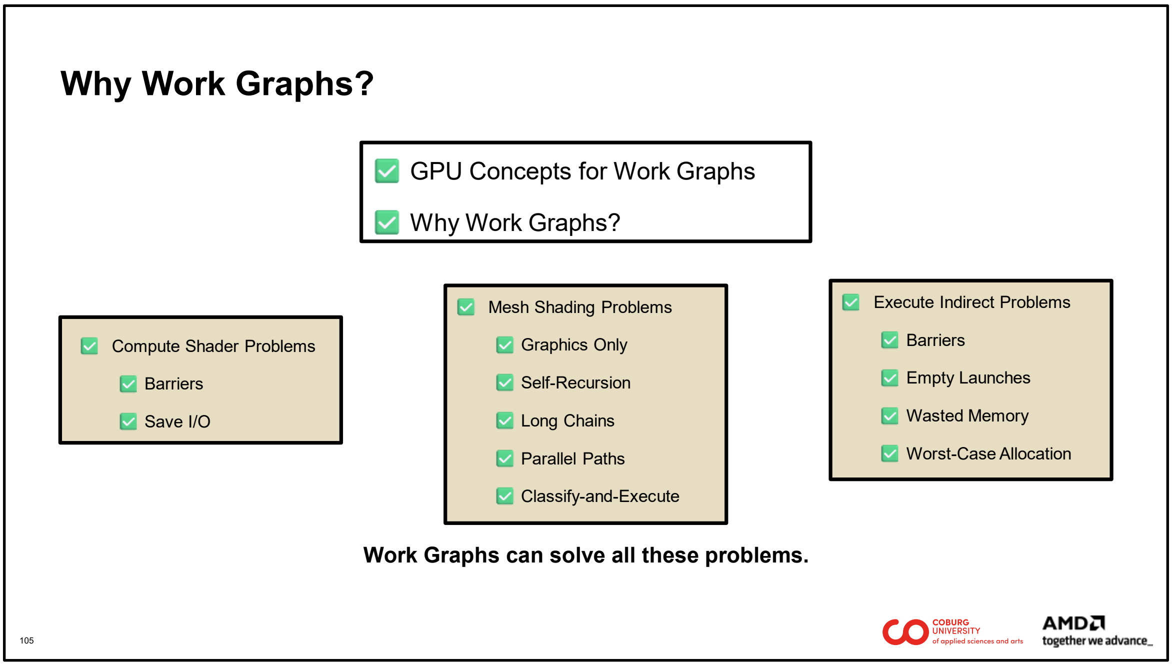
… or even multiple different shader chains.



Diverging branches in a shader chain such as with the classify-and-execute pattern (see later in the Material Shading section of this course) is also not possible.

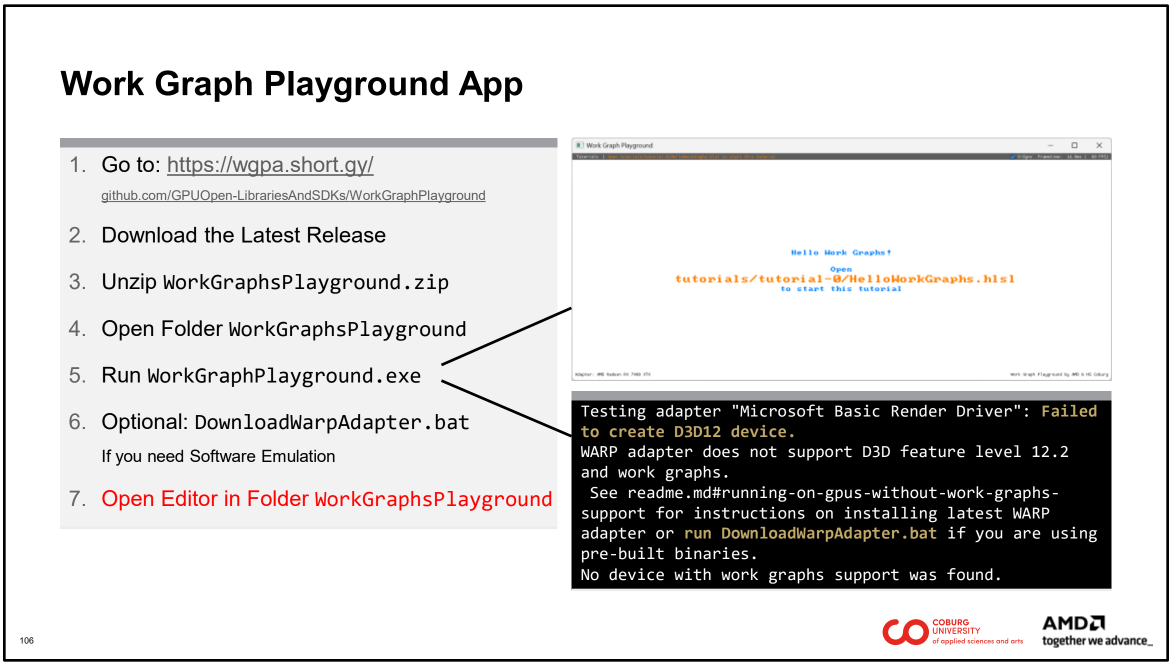


Those problems give us good reasons to define Work Graphs to solve all these problems.



We will show you that Work Graphs can help you solve these problems.

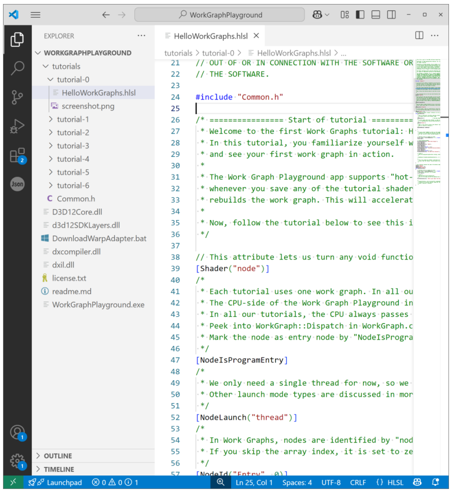
Note: To some extent having, multiple compute queues can deal with these problems, too. Likewise enhanced barriers (https://microsoft.github.io/DirectX-Specs/d3d/D3D12EnhancedBarriers.html) help with better managing barriers.

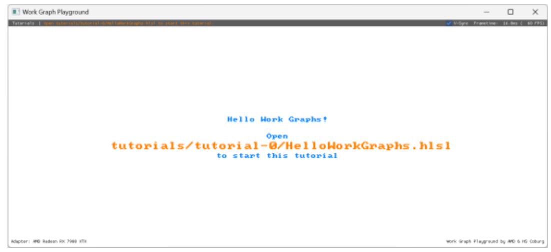


This is now a reminder to download the latest Work Graph Playground App, because in the next section we are going to use it.

So prepare yourself by opening the folder WorkGraphsPlayground in your code editor.

Work Graph Playground App





Useful Command Line Options

1. Software Emulation of the GPU

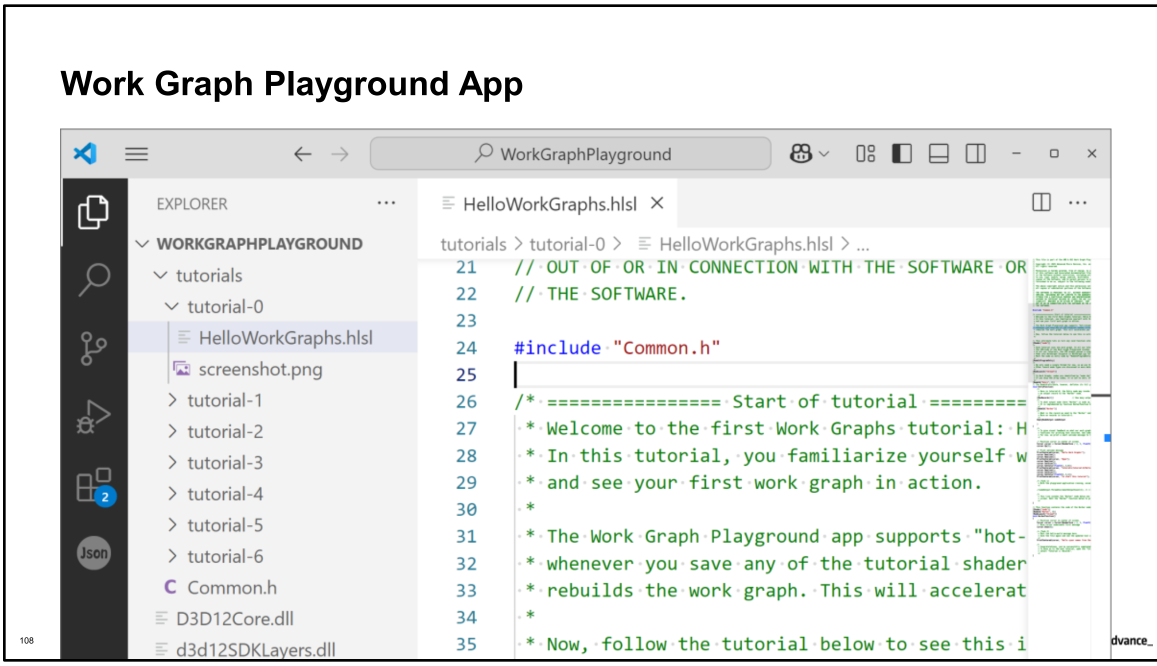
   ǧộsçêẄǻsřAđǻřʧês

2. Print out Debug Information (Requires Graphics Tools)

   êŋǻčľêDêčųĝĽǻỳês

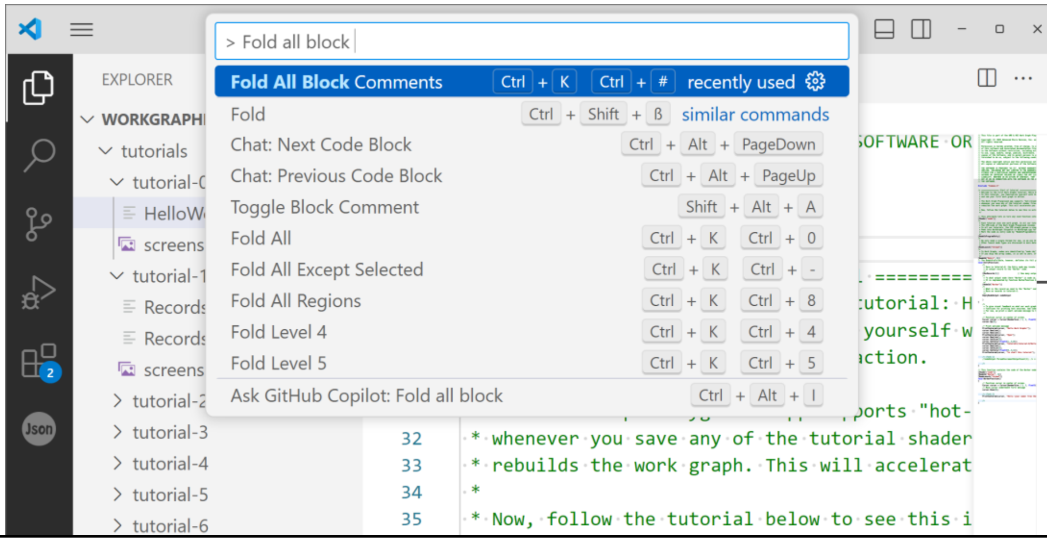
107

All you need to do, is edit the HLSL shader files in your editor.



So please open tutorials/tutorial-0/HelloWorkGraphs.hlsl

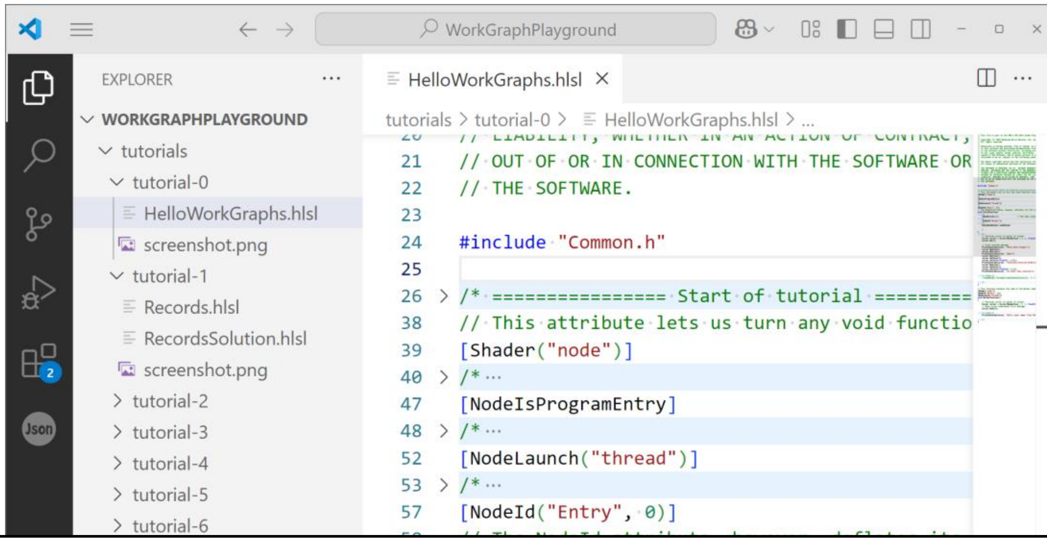
Work Graph Playground App



We provide detailed explanation of the tutorial template and the tutorial tasks. As we will explain a selection of these tasks in this course, you may wish to fold these block comments for easier viewing.

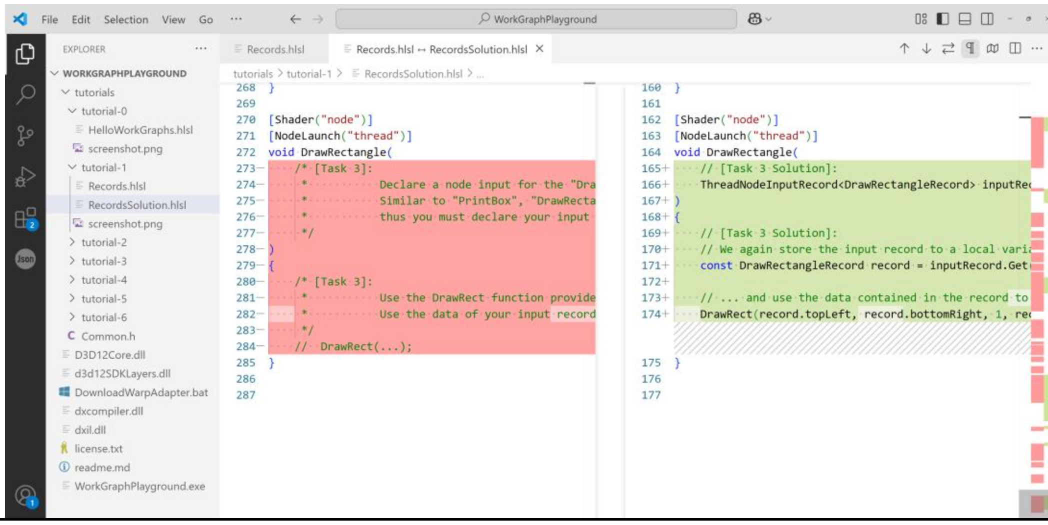
In Visual Studio Code, this can be done with the “Fold All Block Comments” command.

Work Graph Playground App



This should hide the large blocks of comments that might disturb you during the course.

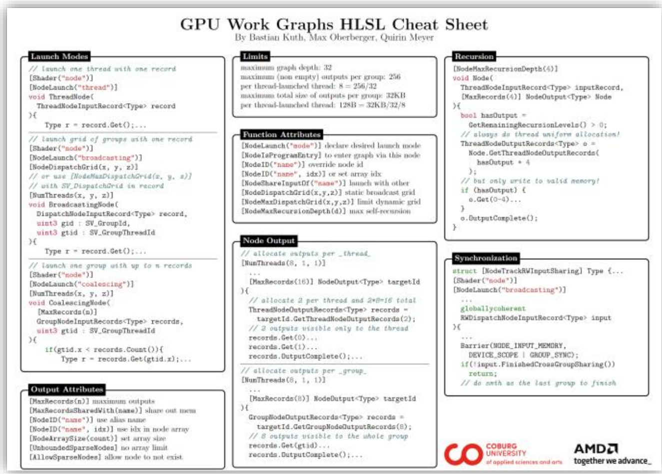
Work Graph Playground App



111

We also provide a sample solution for each tutorial. You can even open both your and our solution in a code-diff editor to compare them.

Work Graphs HLSL Cheat Sheet



GPU Work Graphs HLSL Cheat Sheet

By Bastian Kuth, Max Oberberger, Quirin Meyer

Launch Modes

```
// launch one thread with one record
[Shader("node")]
[NodeLaunch("thread")]
void ThreadNode(
  ThreadNodeInputRecord<Type> record
){
    Type r = record.Get();...
```

```
// launch grid of groups with one record
[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeDispatchGrid(x, y, z)]
// or use [NodeMaxDispatchGrid(x, y, z)]
// with SV_DispatchGrid in record
[NumThreads(x, y, z)]
void BroadcastingNode(
  DispatchNodeInputRecord<Type> record,
  uint3 gid : SV_GroupId,
  uint3 gtid : SV_GroupThreadId
){
    Type r = record.Get();...
```

```
// launch one group with up to n records
[Shader("node")]
[NodeLaunch("coalescing")]
[NumThreads(x, y, z)]
void CoalescingNode(
  [MaxRecords(n)]
  GroupNodeInputRecords<Type> records,
  uint3 gtid : SV_GroupThreadId
){
    if(gtid.x < records.Count()){
        Type r = records.Get(gtid.x);...
```

Output Attributes

[MaxRecords(n)] maximum outputs
[MaxRecordsSharedWith(name)] share out mem
[NodeID("name")] use alias name
[NodeID("name", idx)] use idx in node array
[NodeArraySize(count)] set array size
[UnboundedSparseNodes] no array limit
[AllowSparseNodes] allow node to not exist

Limits

maximum graph depth: 32
maximum (non empty) outputs per group: 256
per thread-launched thread: 8 = 256/32
maximum total size of outputs per group: 32KB
per thread-launched thread: 128B = 32KB/32/8

Function Attributes

[NodeLaunch("mode")] declare desired launch mode
[NodeIsProgramEntry] to enter graph via this node
[NodeID("name")] override node id
[NodeID("name", idx)] or set array idx
[NodeShareInputOf("name")] launch with other
[NodeDispatchGrid(x,y,z)] static broadcast grid
[NodeMaxDispatchGrid(x,y,z)] limit dynamic grid
[NodeMaxRecursionDepth(d)] max self-recursion

Node Output

```
// allocate outputs per _thread_
[NumThreads(8, 1, 1)]
  ...
  [MaxRecords(16)] NodeOutput<Type> targetId
){
  // allocate 2 per thread and 2*8=16 total
  ThreadNodeOutputRecords<Type> records =
    targetId.GetThreadNodeOutputRecords(2);
  // 2 outputs visible only to the thread
  records.Get(0)...
  records.Get(1)...
  records.OutputComplete();...
```

```
// allocate outputs per _group_
[NumThreads(8, 1, 1)]
  ...
  [MaxRecords(8)] NodeOutput<Type> targetId
){
  GroupNodeOutputRecords<Type> records =
    targetId.GetGroupNodeOutputRecords(8);
  // 8 outputs visible to the whole group
  records.Get(gtid)...
  records.OutputComplete();...
```

Recursion

```
[NodeMaxRecursionDepth(4)]
void Node(
  ThreadNodeInputRecord<Type> inputRecord,
  [MaxRecords(4)] NodeOutput<Type> Node
){
  bool hasOutput =
    GetRemainingRecursionLevels() > 0;
  // always do thread uniform allocation!
  ThreadNodeOutputRecords<Type> o =
    Node.GetThreadNodeOutputRecords(
      hasOutput * 4
    );
  // but only write to valid memory!
  if (hasOutput) {
    o.Get(0-4)...
  }
  o.OutputComplete();
}
```

Synchronization

```
struct [NodeTrackRWInputSharing] Type {...
[Shader("node")]
[NodeLaunch("broadcasting")]
  ...
  globallycoherent
  RWDispatchNodeInputRecord<Type> input
){
  ...
  Barrier(NODE_INPUT_MEMORY,
    DEVICE_SCOPE | GROUP_SYNC);
  if(!input.FinishedCrossGroupSharing())
    return;
  // do smth as the last group to finish
```


Additionally, for quick reference, we also provide a cheat sheet for you to look up common Work Graphs syntax.

You can also join the gpu-work-graphs channel on the AMD Developer Community Discord server at https://discord.gg/amd-dev, to connect with the course instructors or other course participants.

GPU Work Graphs – Course Agenda

Introduction & Foundations	14:00 – 14:30
Concepts	14:30 – 15:30
Nodes	
Records	
Launches	
Break	
Advanced Work Graphs	15:45 – 16:45
Material Shading	
Recursion & Synchronization	
Procedural Generation	
Under the hood	
Wrap-Up	16:45 – 17:00

114

Here is a brief overview of the topics that we will cover today.

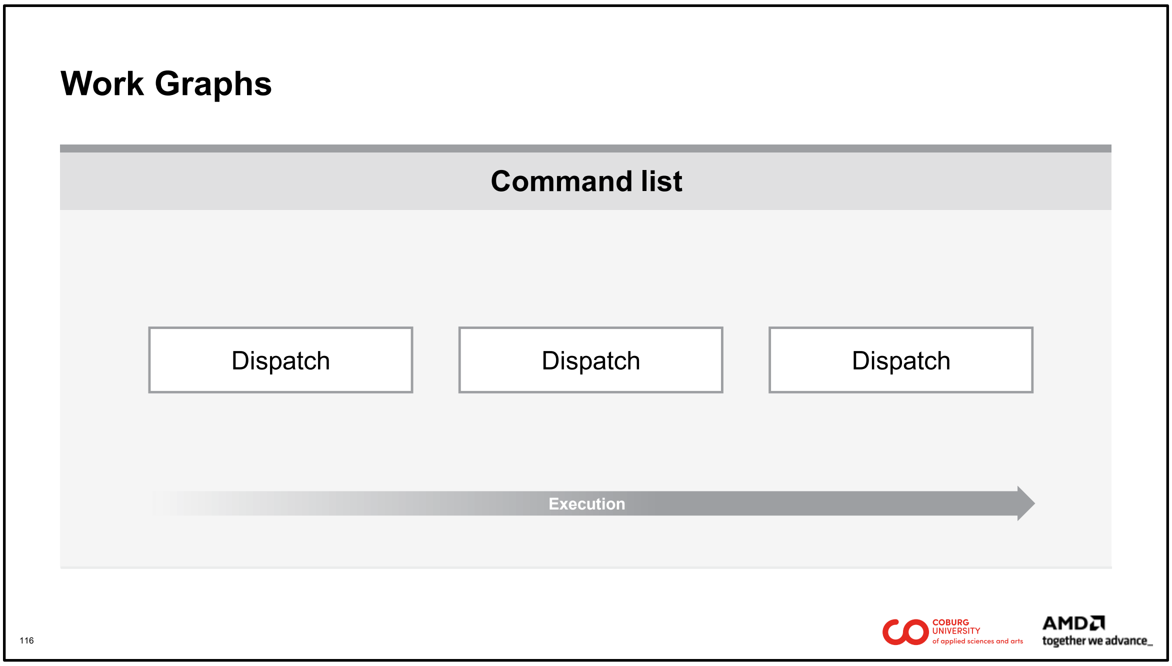
COBURG UNIVERSITY of applied sciences and arts

Work Graph Concepts

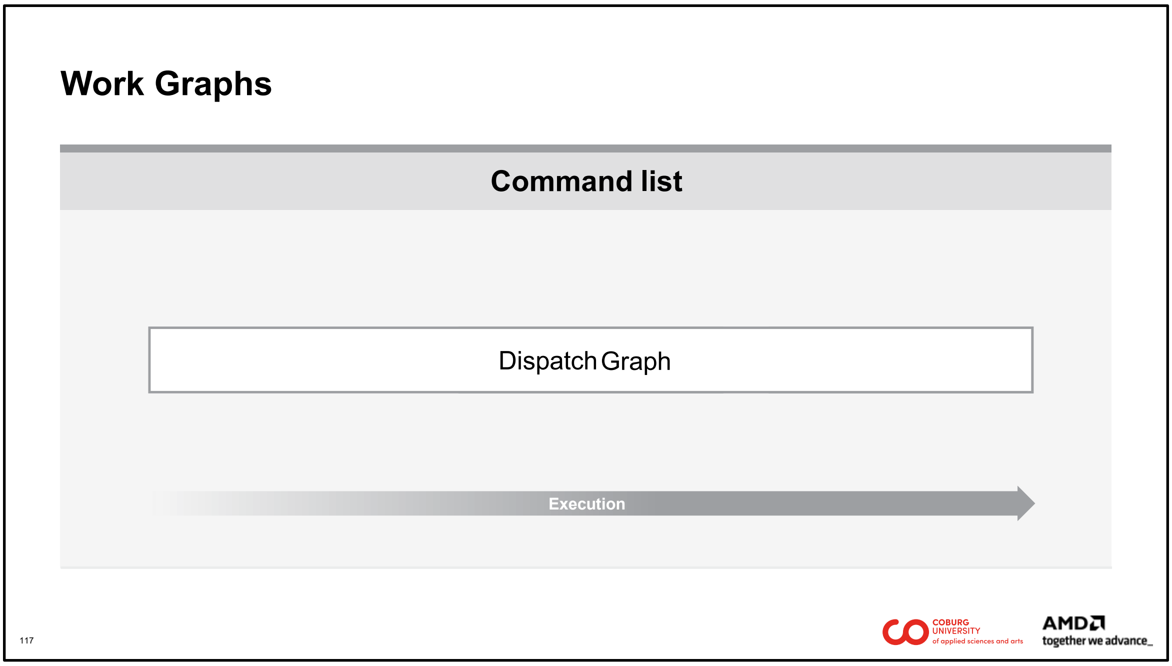
Nodes

115

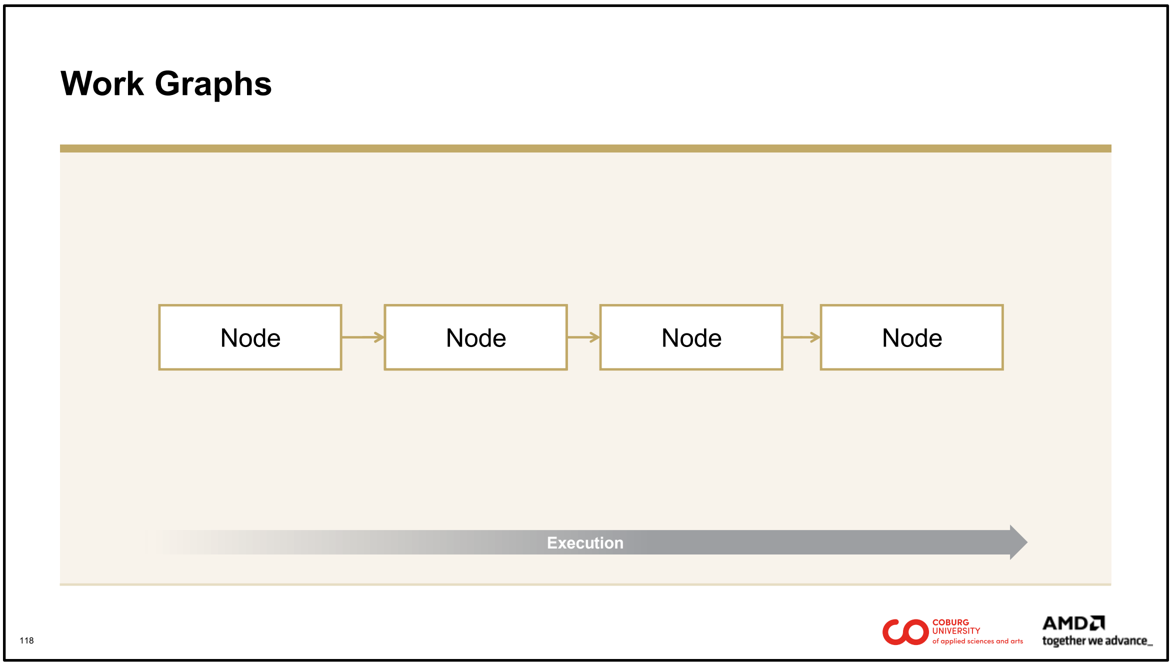
We begin our dive into Work Graphs with the three basic concepts that are key for Work Graphs: nodes, records, and launches. We now start with nodes.



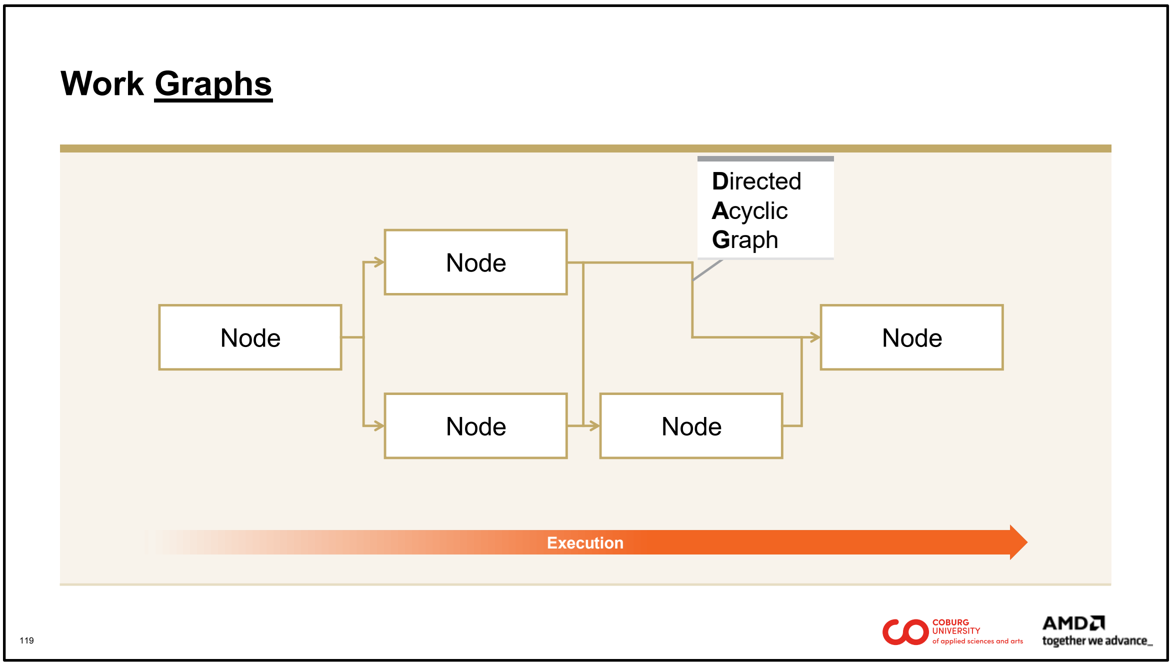
Before work graphs, any work that we wish to carry out on the GPU had to be submitted as individual commands as part of a *command list*. For this course, we focus on compute work loads, thus, the commands shown here are all *dispatches*. The emphasis with these command lists is really on the *list* part, as the GPU would process these command one after the other, thus limiting our options for any type of dynamic decision making on the GPU. In the "Introduction & Foundations" part of this course, we have seen the hassle with fences, barriers, empty launches, and CPU-GPU communication.



With Work Graphs, we can replace these different dispatch commands with a single new command: *DispatchGraph*.



Inside this DispatchGraph command, we no longer have a single compute kernel, but rather a series of connected compute kernel called "*nodes*". These nodes are programmed in a similar way to regular compute kernel/compute shaders using the HLSL programming language and we will dive into the specific syntax in a bit.



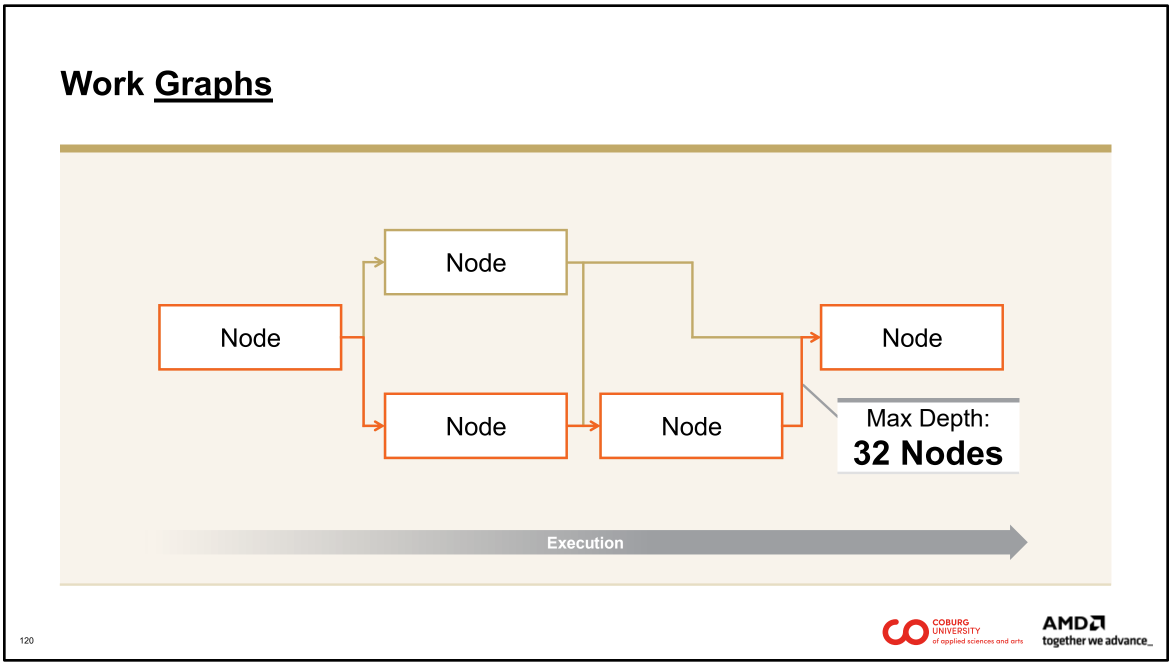
The graph topology of a work graph is, however, not limited to a single long chain of nodes but instead can be classified as a *directed acyclic graph (DAG)*.

As the name “Work Graphs” might suggest, the execution model of this graph is centered around *work* flowing along the edges of the graph from one node to the next. Thus, edges of our graph are directed. Each node can have multiple in- and out-going edges, as shown here.

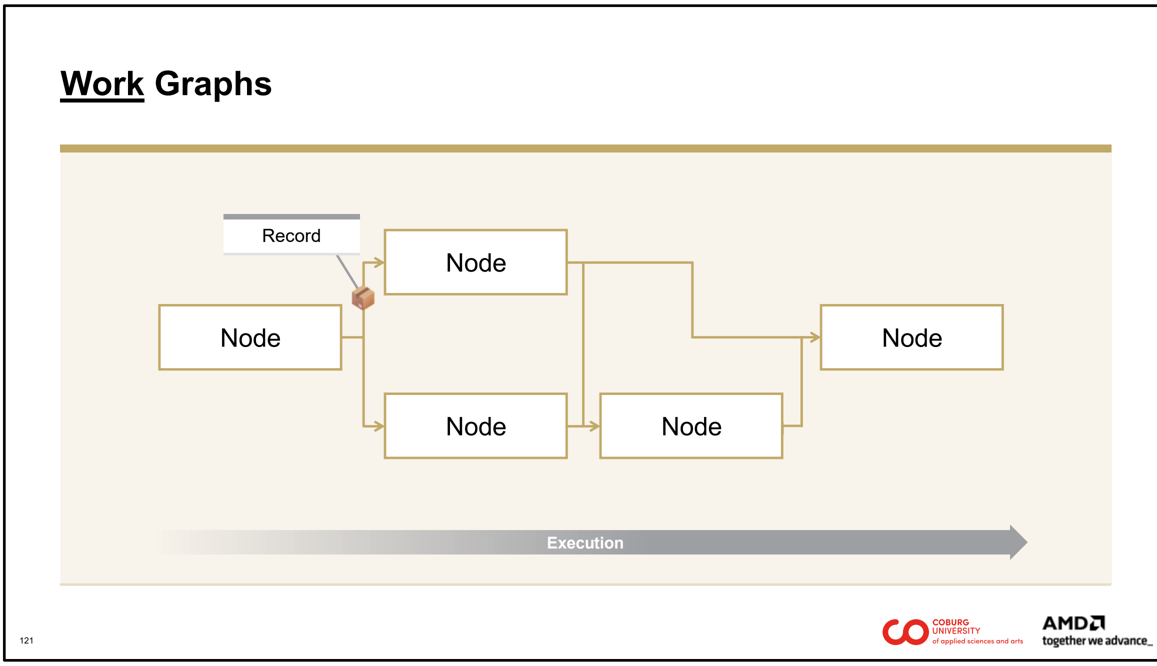
Note that while the graph depicted here has a single root node on the far left, work graphs can have multiple such *root nodes*.

Additionally, cycles* are not allowed in the graph. Therefore, there exists a fixed execution order, shown here going from left to right.

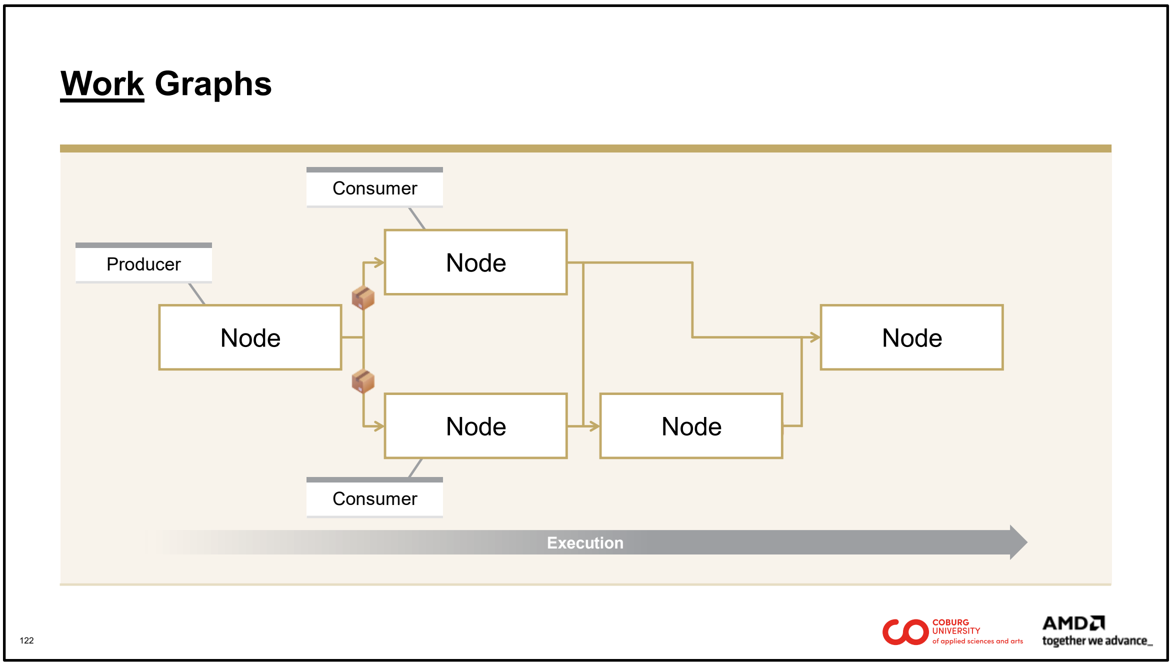
*Note: Work Graphs do allow trivial cycles going from one node to itself. More on this in the “Advanced Work Graphs” section.



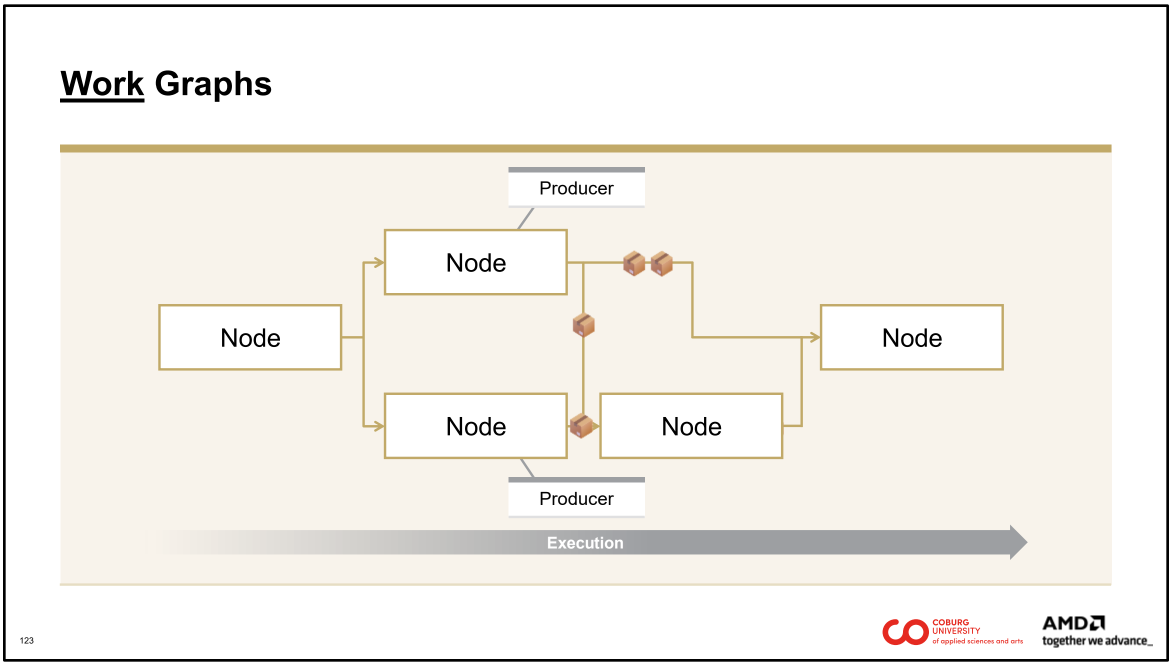
The longest chain of nodes from the first producer node (in graph theory often referred to as *source node*) to the last consumer node (also referred to as *leaf node*), is limited to 32 nodes. The Work Graphs specification refers to this as the maximum graph depth.



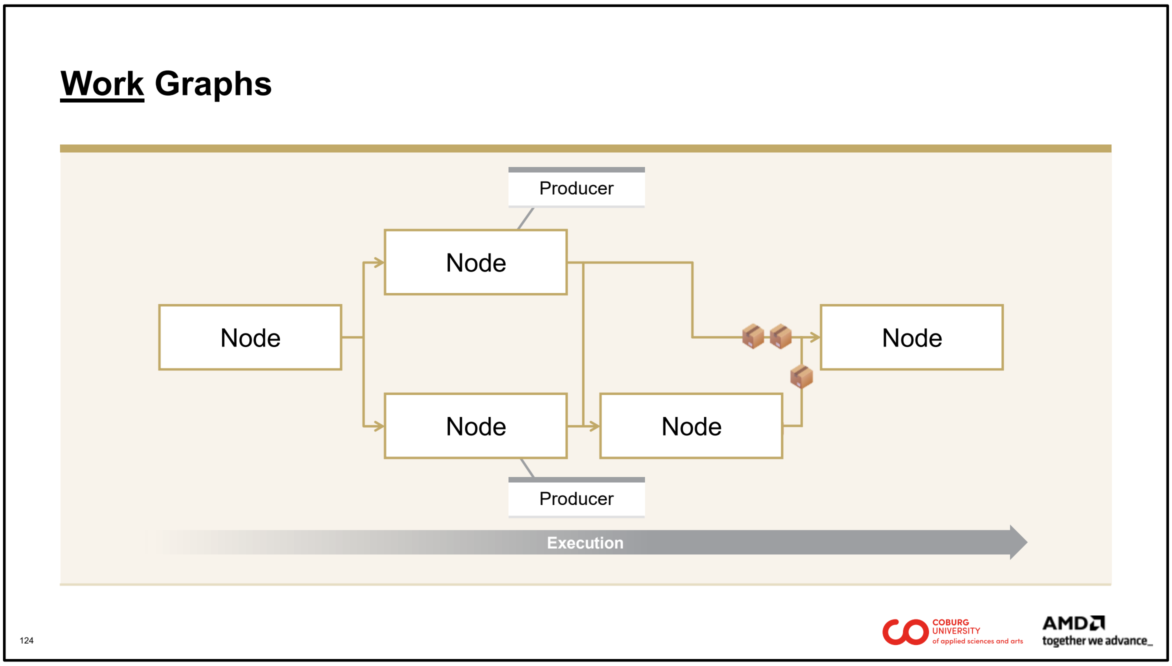
As mentioned before, the execution model is based on work flowing along the edges of the graph. These work items are referred to as *records*.



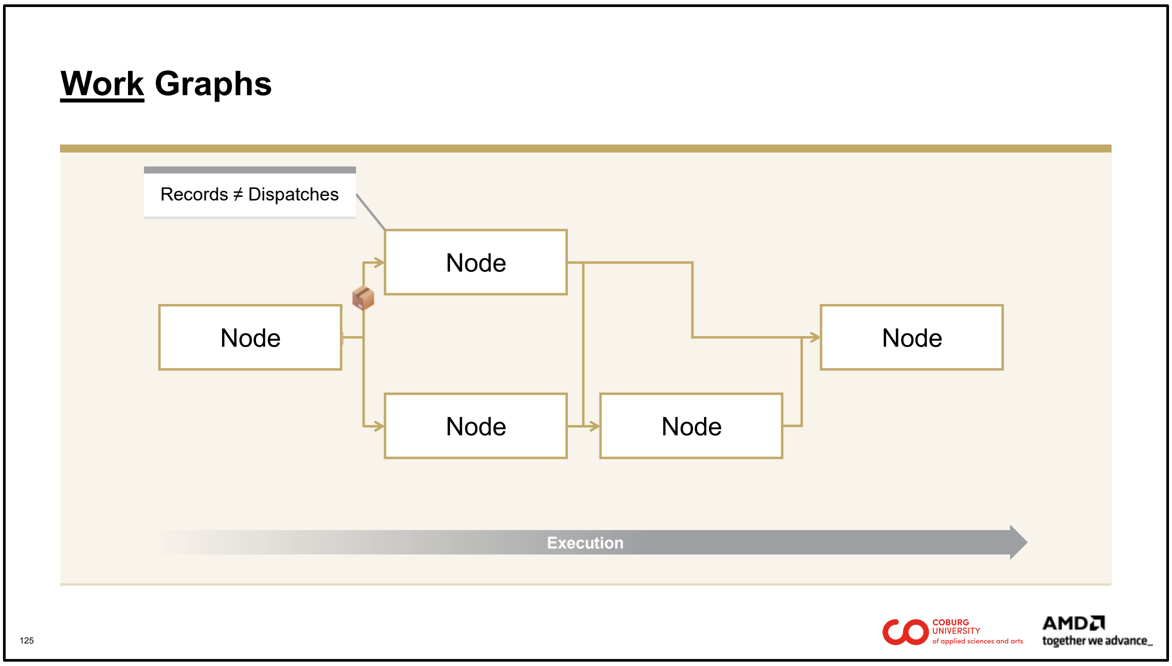
Each node can produce one or more records for one or more other nodes, which then consume these records, thus creating a producer-consumer relationship between nodes.



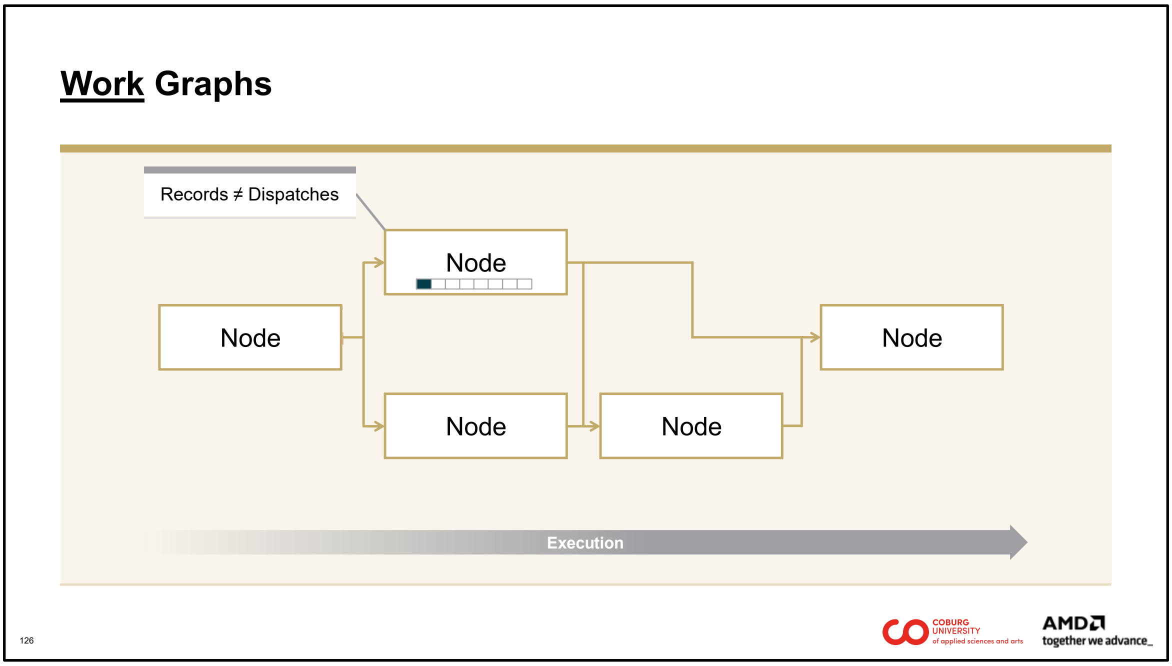
An *inner node*, i.e., with both in- and out-going edges, is both a consumer and a producer at the same time.



These producer-consumer chains repeat until the leaf nodes are reached.

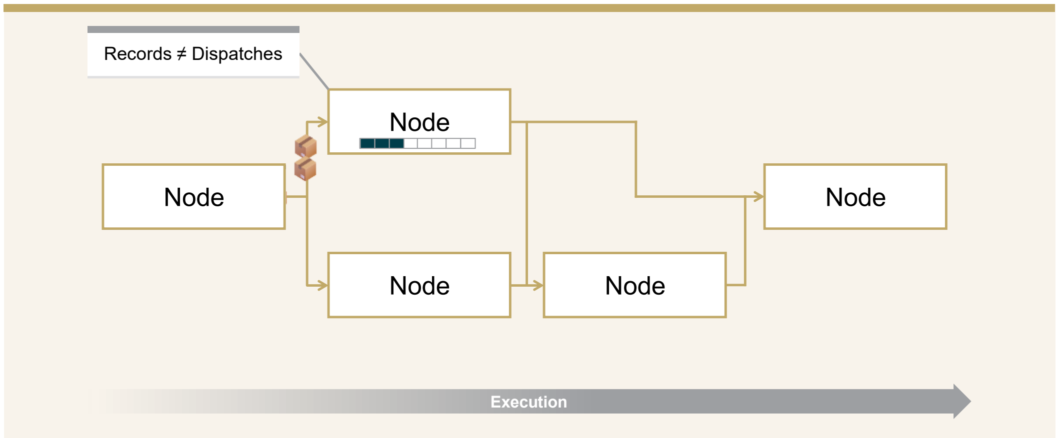


In contrast to other GPU graph programming models, such as CUDA graphs, the records of a work graph are not dispatches to a particular node/compute kernel. Meaning, if a producer node sends a record to a consumer node, the consumer node is not immediately dispatched by the work graph runtime.

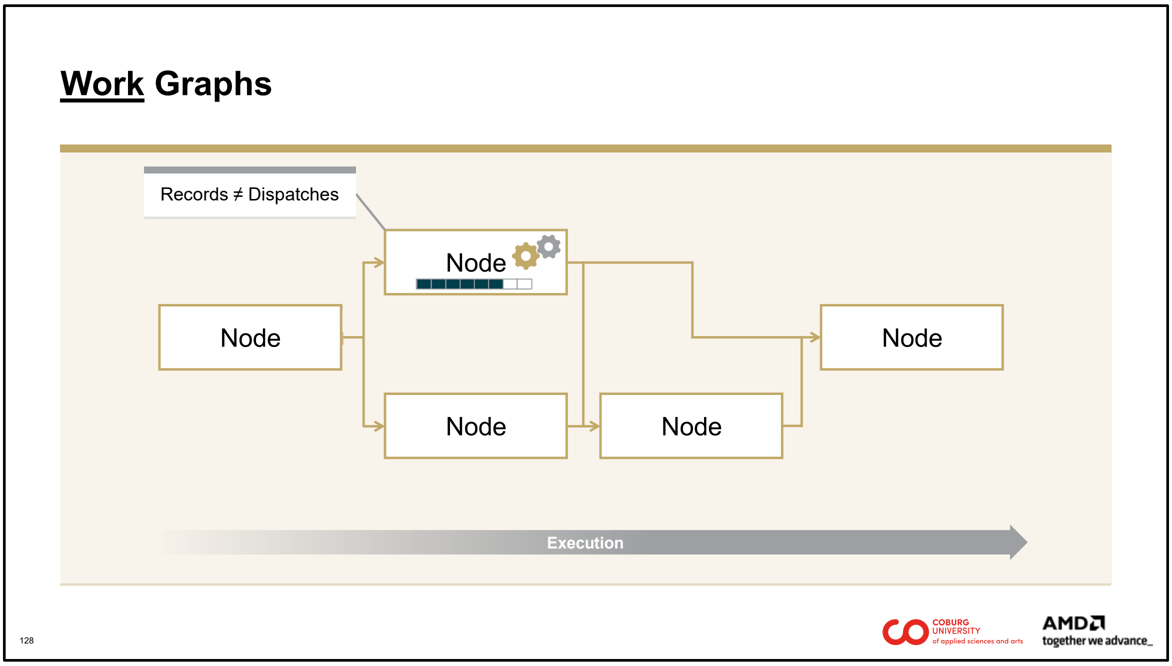


Instead, you can imagine that there is a virtual queue attached to each node. Incoming records are queued up and execution of the node is deferred. It is, however, guaranteed that each incoming record will eventually be processed by the consumer node.

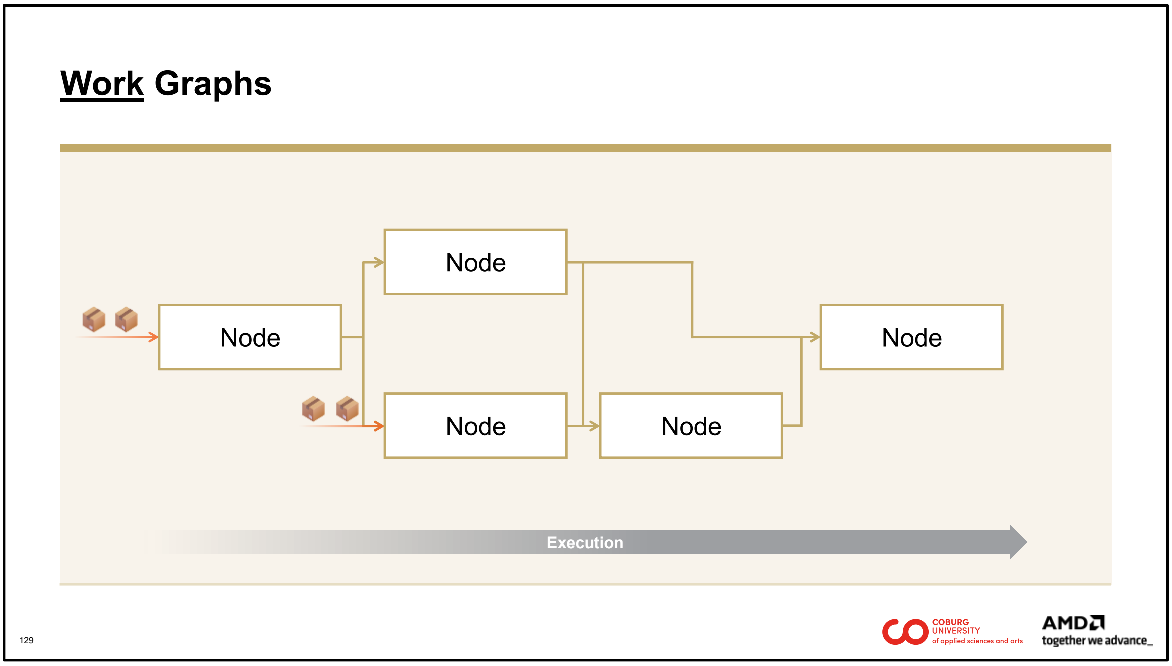
Work Graphs



127



Once the work graph runtime deems enough work is available in the queue, the node is executed. This deferred approach allows the work graph runtime to more efficiently use the available GPU resources.



Dispatching the graph is done by sending records (e.g., from the CPU) to an *entry node*. These initial records are often referred to as *entry work*.

A graph dispatch can contain entry work for multiple nodes. Entry work can also target inner nodes, i.e., nodes that are also targeted by other nodes as well.

Work Graph Concepts – Nodes

tutorial-0/HelloWorkGraphs.hlsl

```
[Shader("node")]
[NodeIsProgramEntry]
[NodeLaunch("thread")]
[NodeId("Entry", 0)]
void EntryFunction() {

}
```

130

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

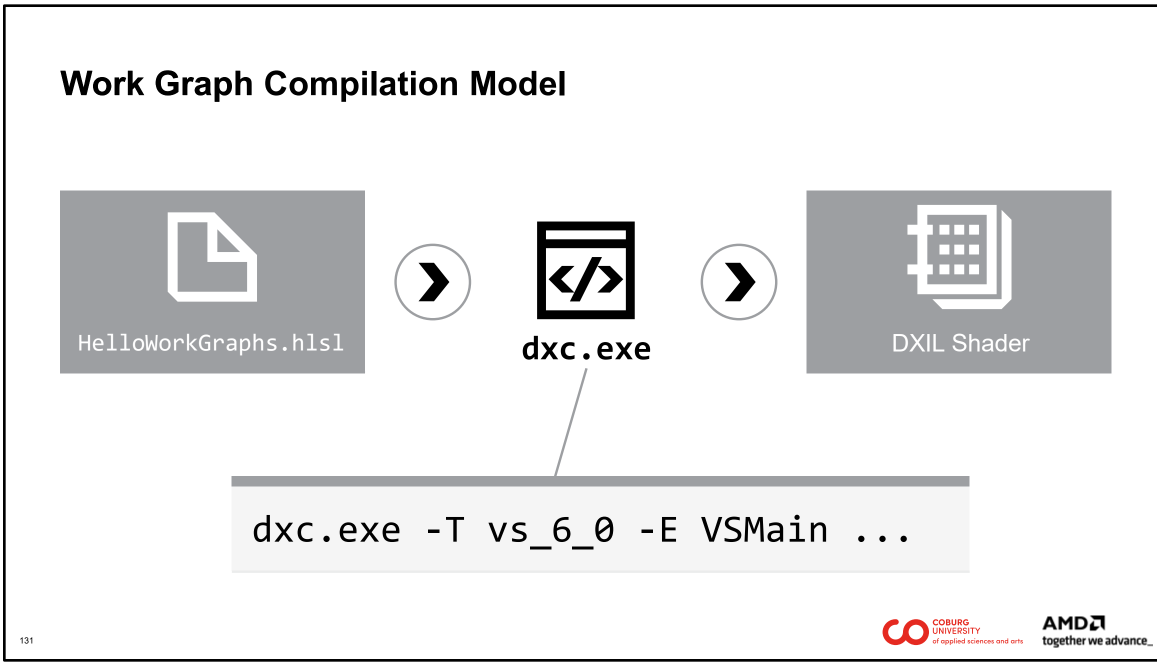
Now, let's have a look at the HLSL syntax for declaring a work graph node.

At its core, a work graph node is an HLSL `void`-function with additional attributes. In our case, we named our function `EntryFunction`, as it will be the entry node to our graph. First, to be able to compile this function as a work graph node, we need to annotate it with a `[Shader("node")]` attribute.

Next, we mark this function as an entry function with the `[NodeIsProgramEntry]` attribute.

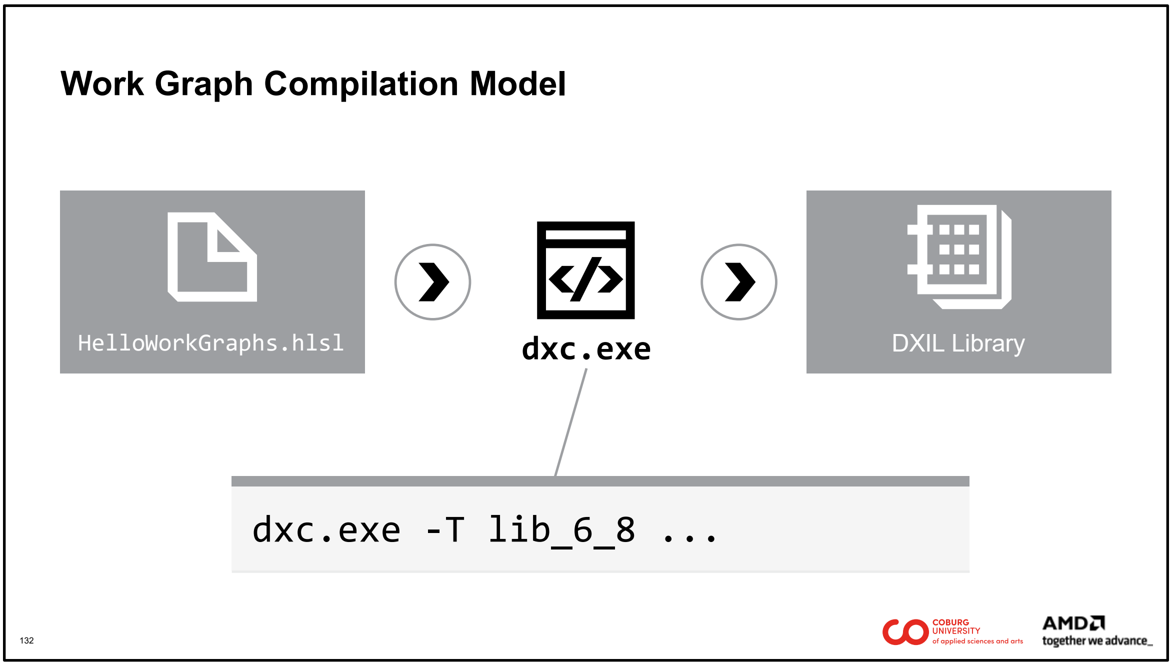
Work Graphs support multiple *launch modes*, which determine how incoming records are processed. We set the launch mode with the `[NodeLaunch(...)]` attribute. We cover the available node launches in greater detail later. For now, we opt for the `"thread"` launch mode. In this launch mode, you write the code of your node function from the perspective of a single thread. The Work Graphs runtime will, of course, attempt to batch multiple threads of the same function together in a thread group to increase SIMD efficiency.

Lastly, we can optionally assign a unique *node id* to our node with the `[NodeId(...)]` attribute. A node id is a pair consisting of an identifier string and an optional index. We uncover what the index is used for, when we discuss Material Shading in the Advanced Work Graphs section.
If we omit the `[NodeId(...)]` attribute, the D3D12 runtime will automatically assign a node id based on the node function name. In our example, this auto-generated node id would be `[NodeId("EntryFunction", 0)]`, as we named our function `EntryFunction`.



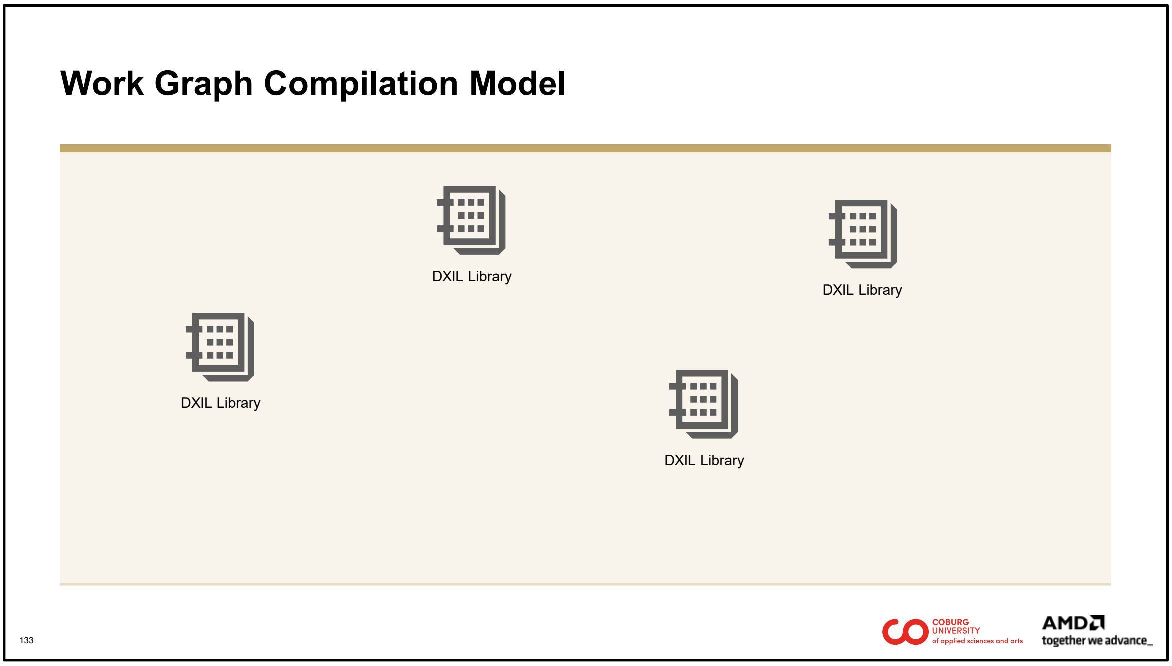
With our shader code complete, we can focus on compiling it for use in a work graph.

When we compile regular shaders, i.e., none Work Graph shaders like compute-, vertex- or pixel-shaders, we compile HLSL files to a single DXIL shader by specifying the shader type (e.g., `vs_...` for vertex shaders or `ps_...` for pixel shaders) and a shader entry point (i.e., the name of e.g., our vertex shader function).

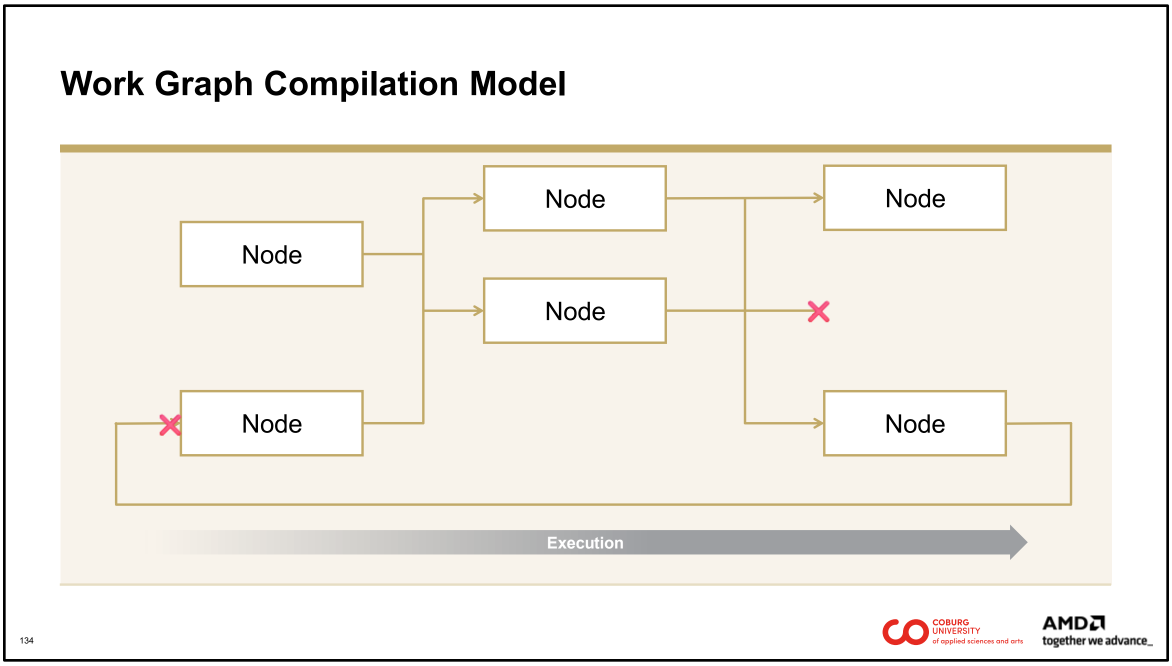


To compile our source file for use with Work Graphs, we need to compile it as a DXIL library, by setting the target to `lib_...`.

DXIL libraries can contain multiple nodes, thus we do not need to specify an entry point. Instead, all functions that we annotated with the `[Shader("node")]` attribute are included in the compiled library.

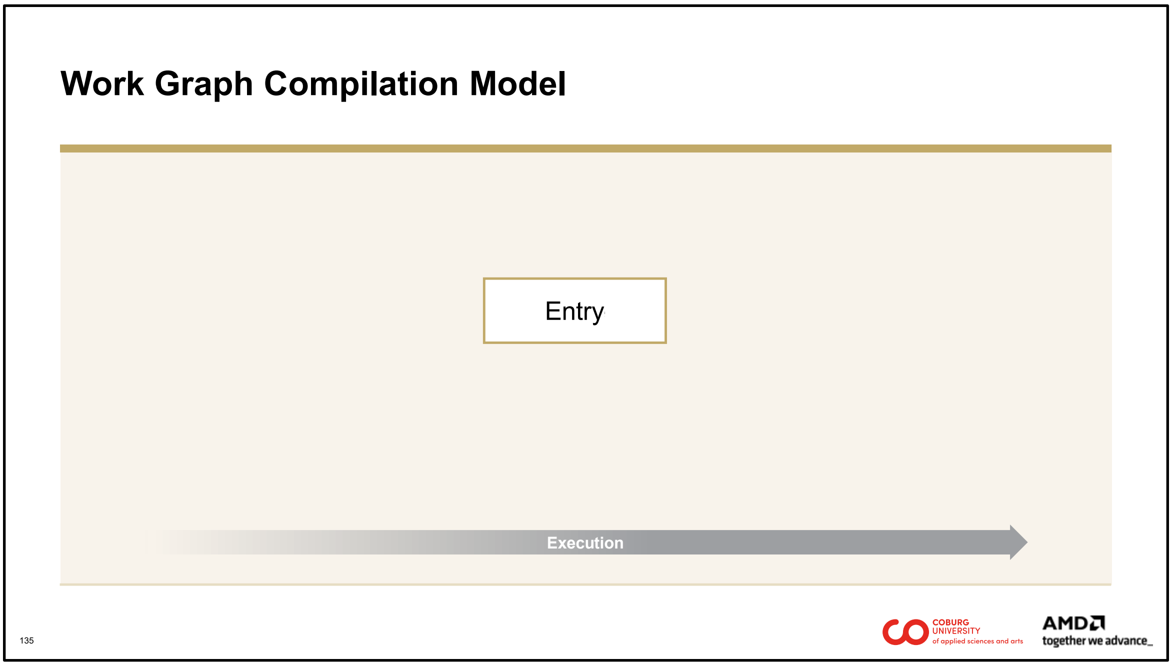


We can then assemble one or more of these DXIL libraries into a work graph.

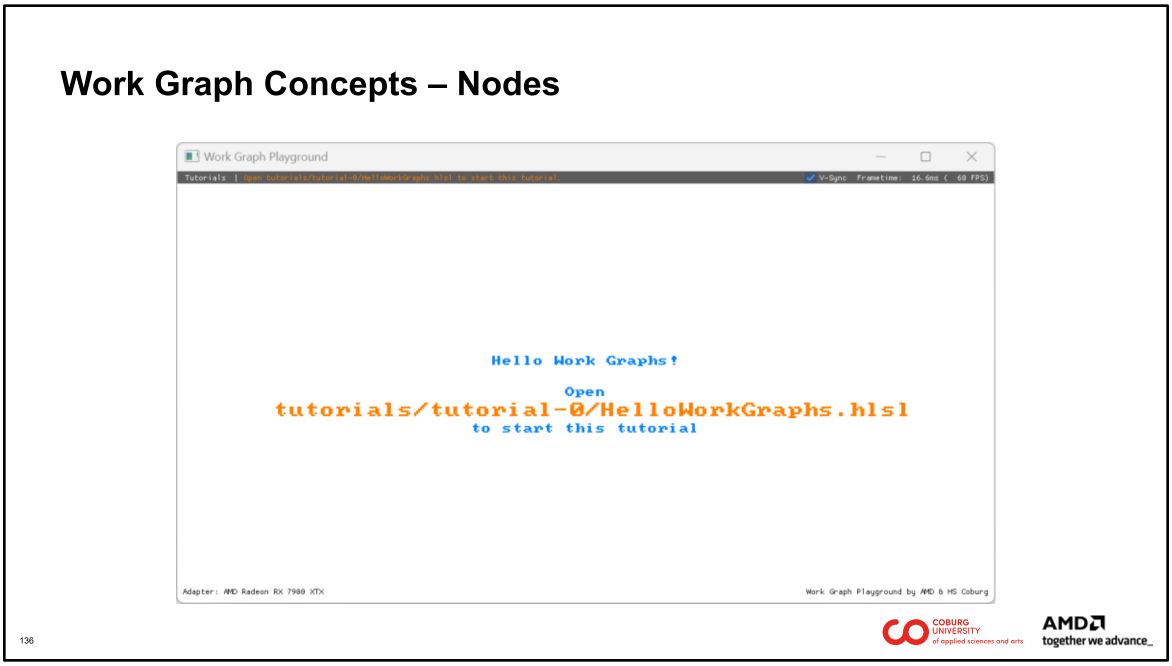


The D3D12 runtime takes the nodes in the DXIL libraries and validates connections between them.

The graph compilation fails if missing nodes (i.e., producers without a matching consumer node) or topological errors (e.g., cycles) are detected.



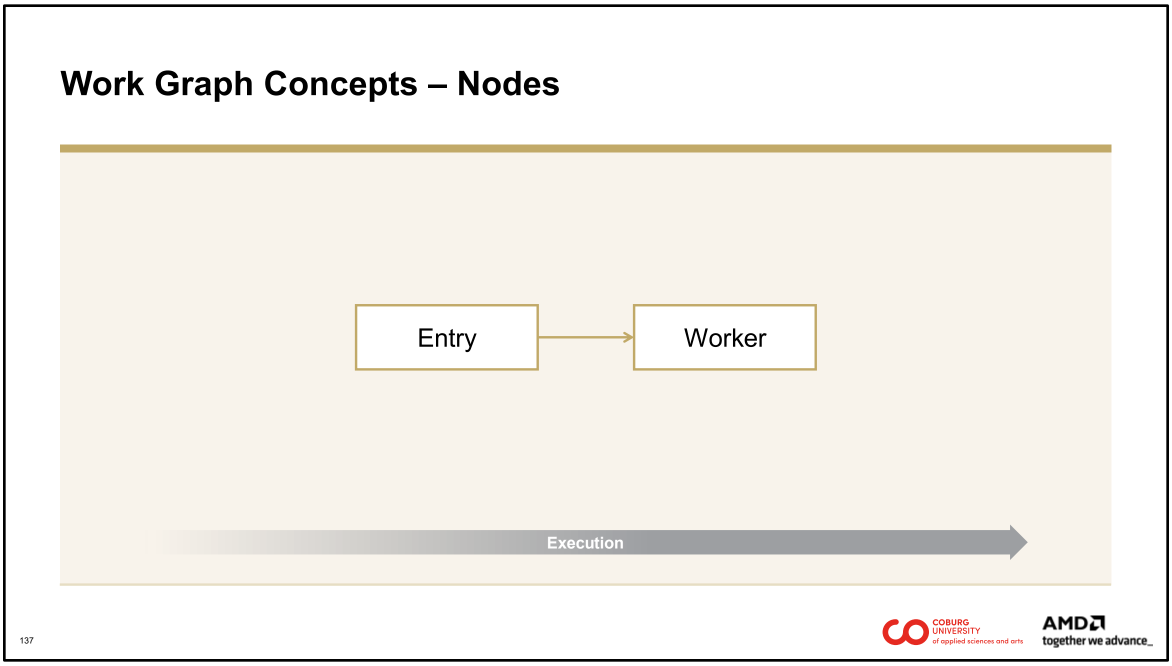
However, in our example from before, we only have a single node, named "Entry".



This node is part of the first tutorial in our Work Graph Playground App.

The "Entry" node prints a "Hello Work Graphs!" message along with instructions for accessing the tutorial.

In the Work Graphs Playground App, you do not have to worry about compilation, as this is fully taken care of by the app. All you need to do to follow along with the tutorial is to run the WorkGraphsPlayground.exe and open `tutorials/tutorial-0/HelloWorkGraphs.hlsl` in an editor of your choice.



With just a single node, however, we cannot show the true capabilities of work graphs, thus we want to create a second node. Here, we opt to call this node “Worker”.

Work Graph Concepts – Nodes

tutorial-0/HelloWorkGraphs.hlsl

```
[Shader("node")]
[NodeLaunch("thread")]
[NodeId("Worker", 0)]
void WorkerFunction() {
    ...
}
```

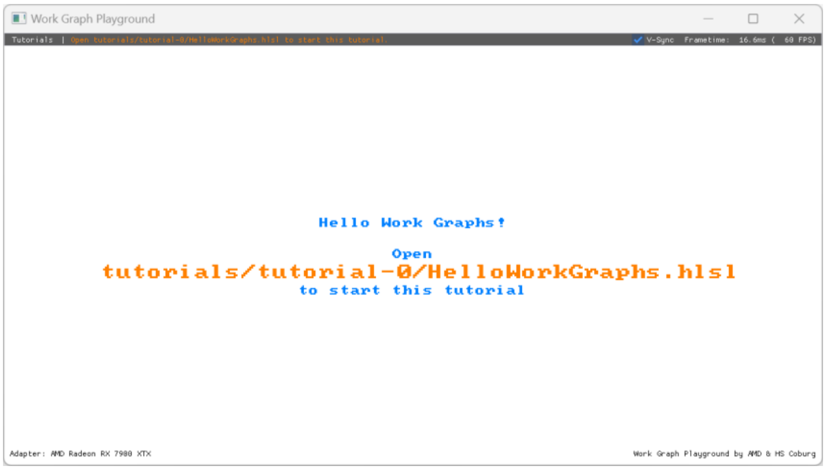
138

To specify the “Worker” node, we write another HLSL function called `WorkerFunction`. We again add the same `[Shader("node")]` and `[NodeLaunch("thread")]` attributes.

To name our node “Worker”, we add a matching `[NodeId("Worker", 0)]` attribute.

You will find this code already in the tutorial file `tutorials/tutorial-0/HelloWorkGraphs.hlsl` on line 114.

Work Graph Concepts – Nodes



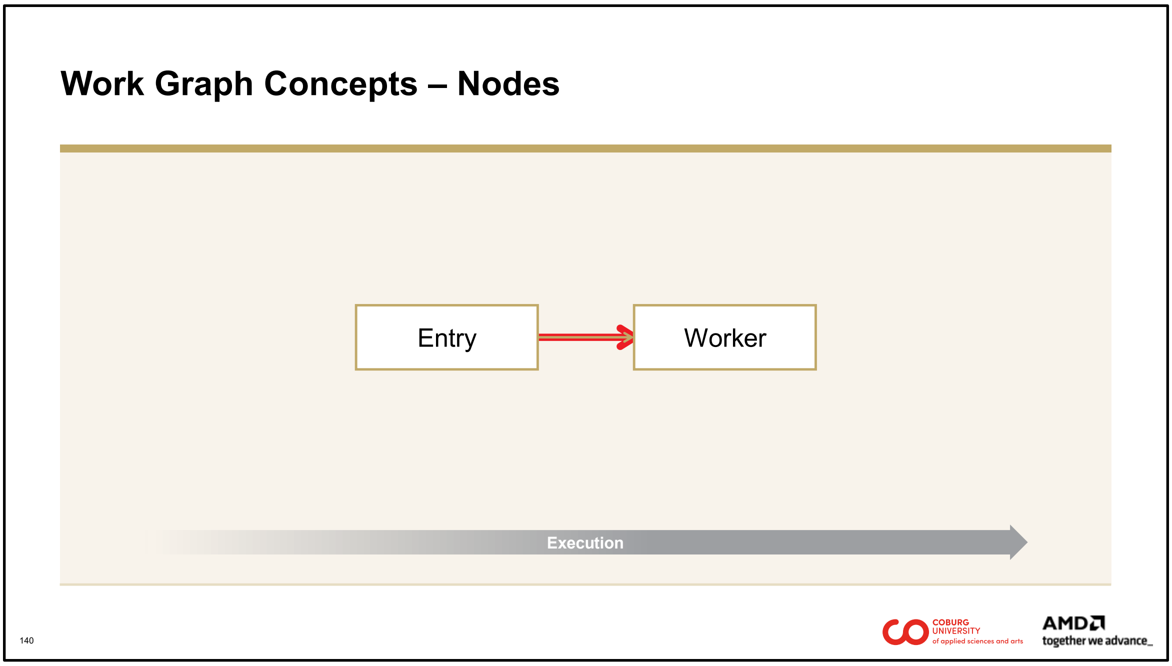
139

Nothing exciting is happening so far. If you look at the code in the `WorkerFunction`, you would expect a

```
Hello <your name> from the "Worker" node!
```

to show up somewhere on screen, but it isn't.

So why is our `WorkerFunction` not yet working?



So far, we have only declared both the “Entry” and “Worker” function, but crucially, we have not set up the connection between them.

Work Graph Concepts – Nodes

```
tutorial-0/HelloWorkGraphs.hlsl
```

```
...
void EntryFunction(

    [MaxRecords(1)]
    [NodeId("Worker")]
    EmptyNodeOutput nodeOutput

) {
}
```

141

To fix this, we need to go back to our `EntryFunction` and declare a *node output*. Node outputs are part of the function signature and form the edges between the nodes in our graph.

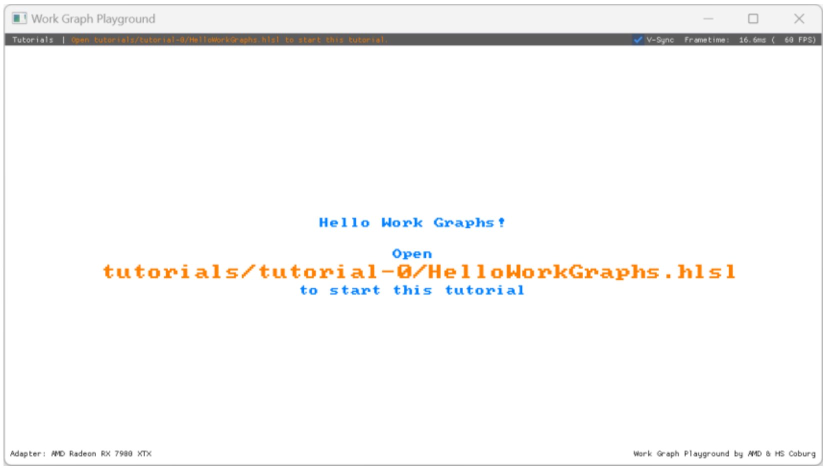
Here, we declare a parameter `nodeOutput` of type `EmtpyNodeOutput`. The type of node output will determine the type of record that we want to send between the nodes, but more on those later. For now, we opt for an empty record, hence the `EmptyNodeOutput` type.

To target our previously created “Worker” node, we can use the `[NodeId(...)]` attribute to specify which node we want to send record(s) to. This attribute is again optional, and if none is present, the node id will be inferred by the name of the node output parameter. Thus, if we want to omit the `[NodeId(...)]` attribute here, we have to write `EmptyNodeOutput Worker`, to target our “Worker” node.

Lastly, we need to declare the maximum number of records that we want to send with the `[MaxRecords(...)]` attribute. In our example, we only send a single record.

You will again find this code in the tutorial file `tutorials/tutorial-0/HelloWorkGraphs.hlsl` on line 64.

Work Graph Concepts – Nodes



142

If we check back with the Work Graph Playground App, we still do not see the message from the “Worker” node.

Work Graph Concepts – Nodes

tutorial-0/HelloWorkGraphs.hlsl

```
...
void EntryFunction(
    [MaxRecords(1)]
    [NodeId("Worker")]
    EmptyNodeOutput nodeOutput
) {
    ...

    // nodeOutput.ThreadIncrementOutputCount(1);
}
```

143

The reason for this is simple: while we have declared an output from “Entry” to “Worker” and thus formed a connection between these two nodes, we have not actually sent any records yet.

In the tutorial file `tutorials/tutorial-0/HelloWorkGraphs.hlsl` on line 106, you’ll find the commented-out code above.

Work Graph Concepts – Nodes

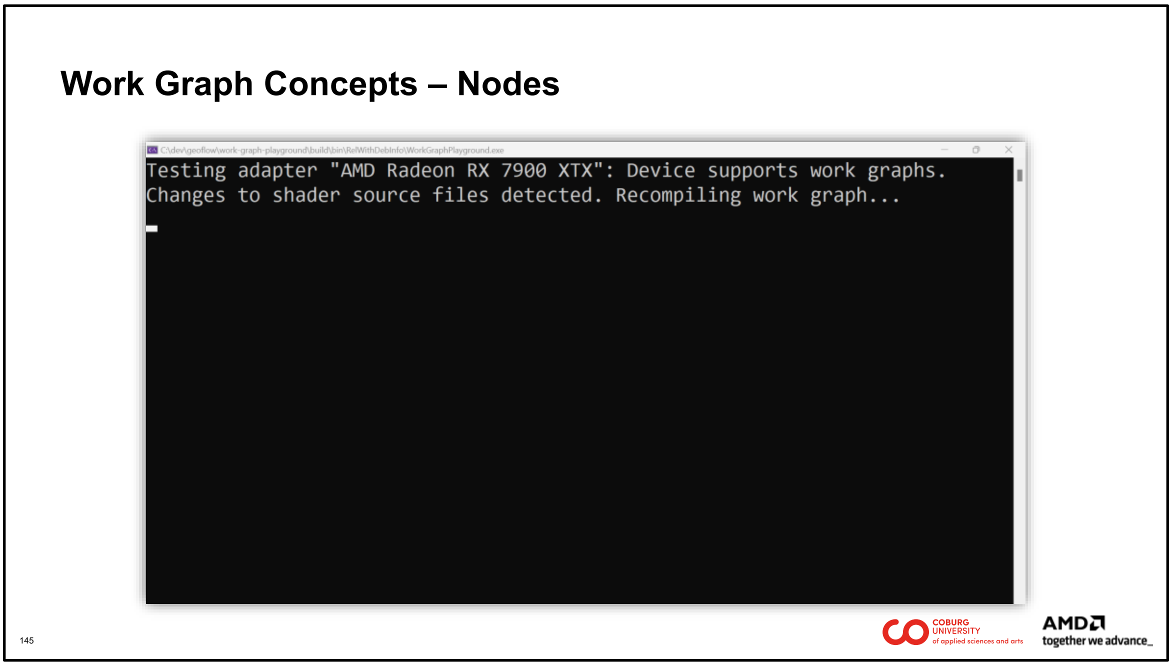
tutorial-0/HelloWorkGraphs.hlsl

```
...
void EntryFunction(
    [MaxRecords(1)]
    [NodeId("Worker")]
    EmptyNodeOutput nodeOutput
) {
    ...

    nodeOutput.ThreadIncrementOutputCount(1);
}
```

144

Uncomment this line!

You can see that we're now using the `nodeOutput` parameter that we declared before and incrementing the output count by one, thus sending a single record to the "Worker" node.



Save your file in the editor and look at the console output of the Work Graphs Playground App. It automatically detects when you change a file and tries to recompile it.

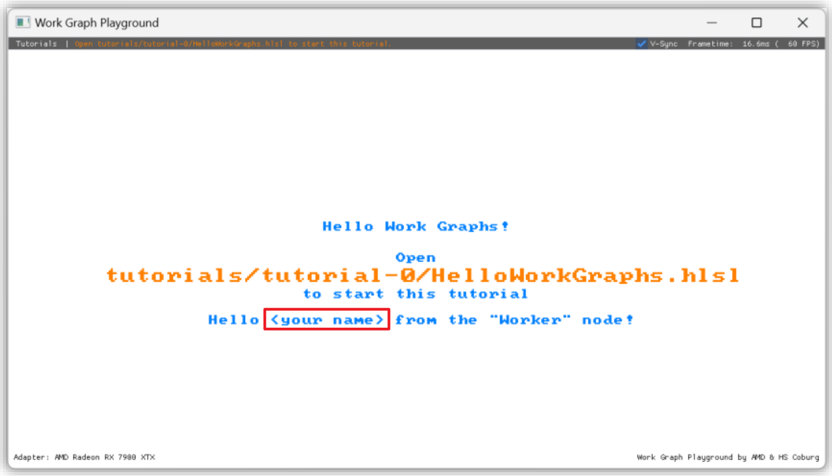
There you will also see error messages. If you run into compile errors, the last successfully compiled work graph continues to execute.

Work Graph Concepts – Nodes

146

Then you should see, that the code of the “Worker” node is executed and the message is printed on screen.

Work Graph Concepts – Nodes



147

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

Your next task is to customize the welcome message with your name.

Warning: Do not copy your answer from your neighbor. We'll find out!

Work Graph Concepts – Nodes

tutorial-0/HelloWorkGraphs.hlsl

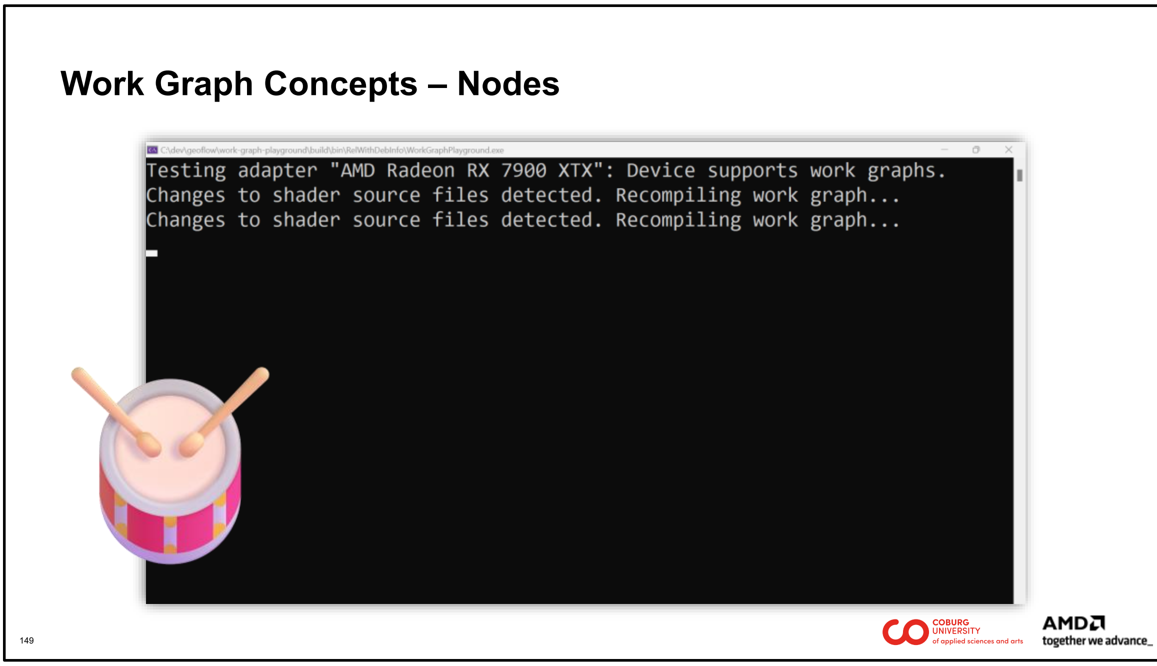
```
...
void WorkerFunction()
{
    ...
    PrintCentered(cursor, "Hello SIGGRAPH 2025..!");
}
```

148

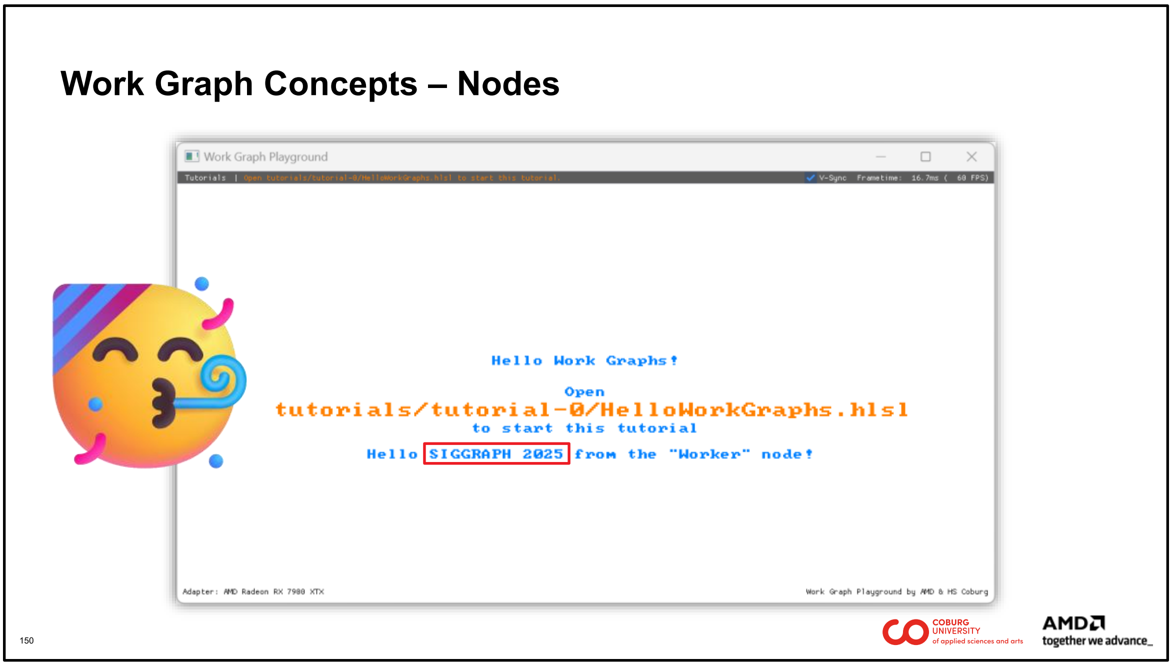
Head back to the tutorial file `tutorials/tutorial-0/HelloWorkGraphs.hlsl` and change the message on line 129.

We instructors send out our greetings to everyone at SIGGRAPH 2025.

Save your file in your code editor…



… look at the console and … wait for it … until the work graph has compiled…



… and congratulations, you just finished your first work graph tutorial 🎉.

Work Graph Concepts – Nodes

tutorial-0/HelloWorkGraphs.hlsl

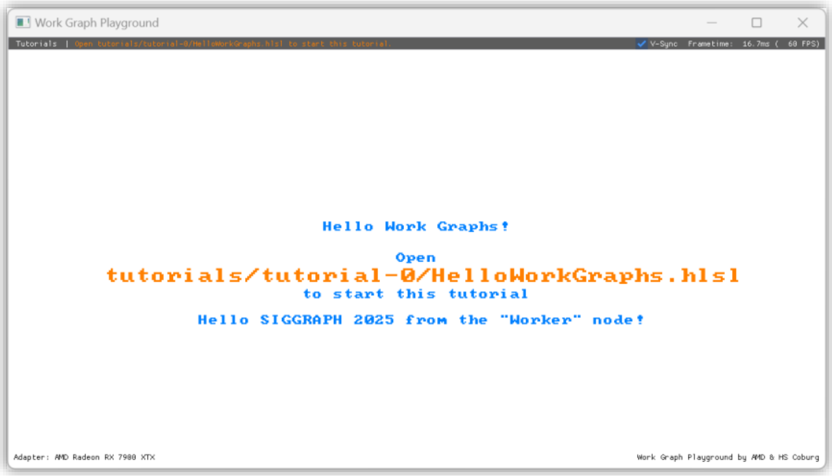
```
...
void EntryFunction(
    [MaxRecords(2)]
    [NodeId("Worker")]
    EmptyNodeOutput nodeOutput
) {
    ...

    nodeOutput.ThreadIncrementOutputCount(2);
}
```

151

Next, let's see what happens when we send two records to the "Worker" node.

First, we increment the `[MaxRecords(...)]` attribute from `1` to `2`. This means, we may now output up to two records. Second, we change the code of the `EntryFunction` itself to increment the output count by two instead of one.

Work Graph Concepts – Nodes

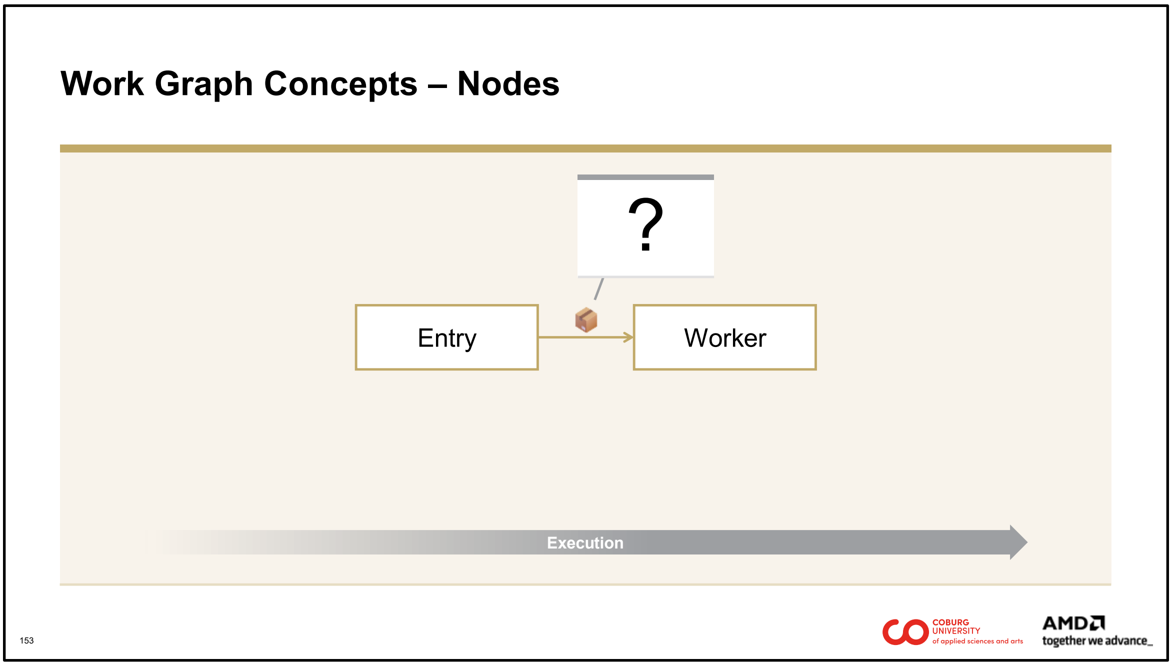


152

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

If we save the file again and go back to the Work Graph Playground App, we see no effect. However, in fact, the “Hello SIGGRAPH 2025 from the “Worker” node!” is written twice.



The reason why we do not see the message twice is simple: we are sending empty records. Thus, while the “Worker” node is executed twice, it is printing the same message at the same location every time.

So next, we are going to see how we can add data to our records, to change the behavior of a consumer node based on data in the record.

COBURG UNIVERSITY of applied sciences and arts

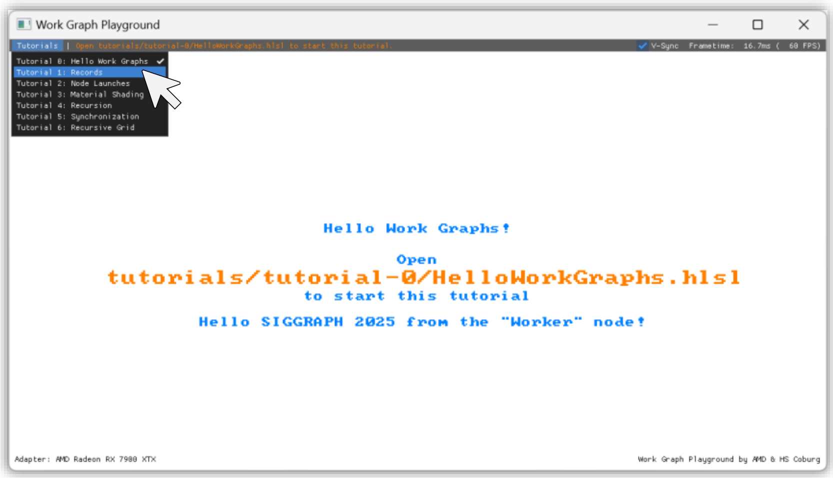
Work Graph Concepts

Records 📦

154

After having learnt about nodes, we learn about *records* as a way to send data between nodes, next.

Work Graph Concepts – Records



155

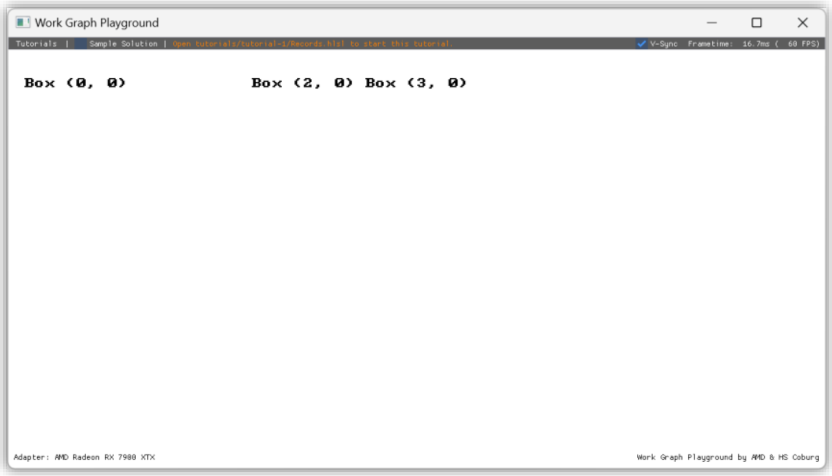
COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

As you have already successfully completed the first tutorial, it is now time to move on to the next one.

Select “Tutorial 1: Records” from the menu on the top-left of the Work Graph Playground App and open `tutorials/tutorial-1/Records.hlsl` in your code editor.

Work Graph Concepts – Records



156

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

Your Work Graph Playground App should now look like this.

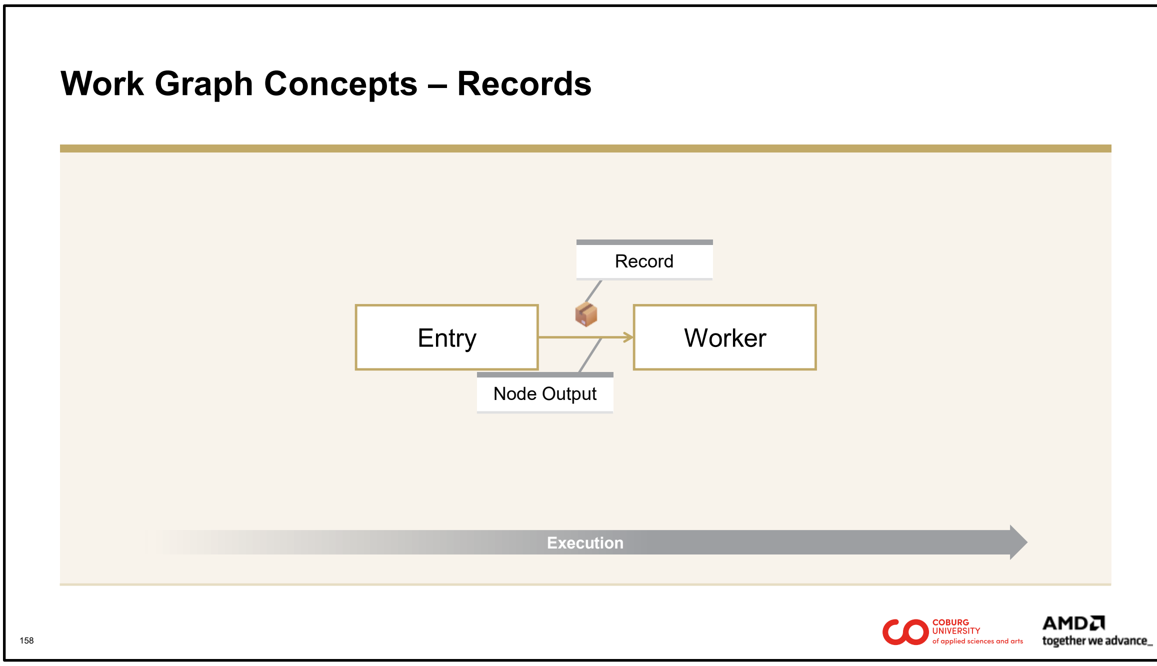
Work Graph Concepts – Records

- ◯ 1. Print “Hello World”
- ◯ 2. Declare `DrawRectangleRecord`
- ◯ 3. Complete `DrawRectangle` node
- ◯ 4. Declare output to `DrawRectangle` node
- ◯ 5. Emit records `DrawRectangle` node
- ◯ 6. Homework: Draw enclosing rectangle

157

COBURG UNIVERSITY of applied sciences and arts AMD together we advance_

In this tutorial, we have six tasks in which we are going to learn how to use records. We complete the first five tasks one at a time and explain the concepts of records along the way.



So far, we have seen how we can declare nodes, how we can add edges between nodes by declaring node outputs, and we have seen how we can send empty records from one node to another.

Up until now, we have only used empty records, meaning we only communicated to the Work Graphs runtime, that we want to launch a particular node, but we have not sent any data.

Work Graph Concepts – Records

tutorial-0/HelloWorkGraphs.hlsl

```
...
void EntryFunction(
    [MaxRecords(1)]
    [NodeId("Worker")]
    EmptyNodeOutput nodeOutput
) {
}
```

```
[Shader("node")]
[NodeLaunch("thread")]
[NodeId("Worker", 0)]
void WorkerFunction() {
    ...
}
```

159

Let us summarize the concept of Work Graphs nodes detailed in the previous tutorial-0.

There, we had a producer node, implemented by the `EntryFunction` that can produce at most a single `EmptyNodeOutput` for the Node `Worker`.

At the consuming node, `Worker`, the `WorkerFunction` executes code once a `nodeOutput` is sent off. Both nodes are connected over the `[NodeId("Worker")]` attribute of `nodeOutput`.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
[Shader("node")]
...
[NumThreads(4, 1, 1)]
void Entry(

    [MaxRecords(1)]
    EmptyNodeOutput PrintHelloWorld
) {
}
```

`[NodeId("PrintHelloWorld")]`

Now, in this tutorial-1, we also start out by sending an `EmptyNodeOutput PrintHelloWorld` to another consuming node `"PrintHelloWorld"`.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
[Shader("node")]
...
[NumThreads(4, 1, 1)]
void Entry(
    [MaxRecords(1)]
    EmptyNodeOutput PrintHelloWorld
) {
    // [Task 1]: Emit a single empty record
    //           to the "PrintHelloWorld" node.

}
```

161

One difference with this tutorial is that the `Entry` node no longer uses the `"thread"` launch mode, but it uses the `"broadcasting"` launch mode, instead. We will cover the specifics of launch modes shortly. For now, the main difference of the broadcasting launch mode over the thread launch mode is that we are programming a thread group instead of a single thread. In our example, our thread group consists of four threads indicated by the `[NumThreads(4, 1, 1)]` attribute. This is very much like you would program a compute shader.

When we are using thread-group launch modes (i.e., not `"thread"`) for our nodes, the `[MaxRecords(...)]` attribute declares the maximum number of records the entire thread group can send to a particular consumer node. In this case, this means that all four threads together can send one single empty record to the `"PrintHelloWorld"` node.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
[Shader("node")]
...
[NumThreads(4, 1, 1)]
void Entry(
    [MaxRecords(1)]
    EmptyNodeOutput PrintHelloWorld
) {
    // [Task 1]: Emit a single empty record
    //           to the "PrintHelloWorld" node.
    PrintHelloWorld.ThreadIncrementOutputCount(1);
}
```

162

Our first task in this tutorial is to send a single record to the `"PrintHelloWorld“` node. However, if we were to use

```
PrintHelloWorld.ThreadIncrementOutputCount(1);
```

as we did in the previous tutorial, every one of our four threads would increment the output count by one, thus sending one empty record per thread.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

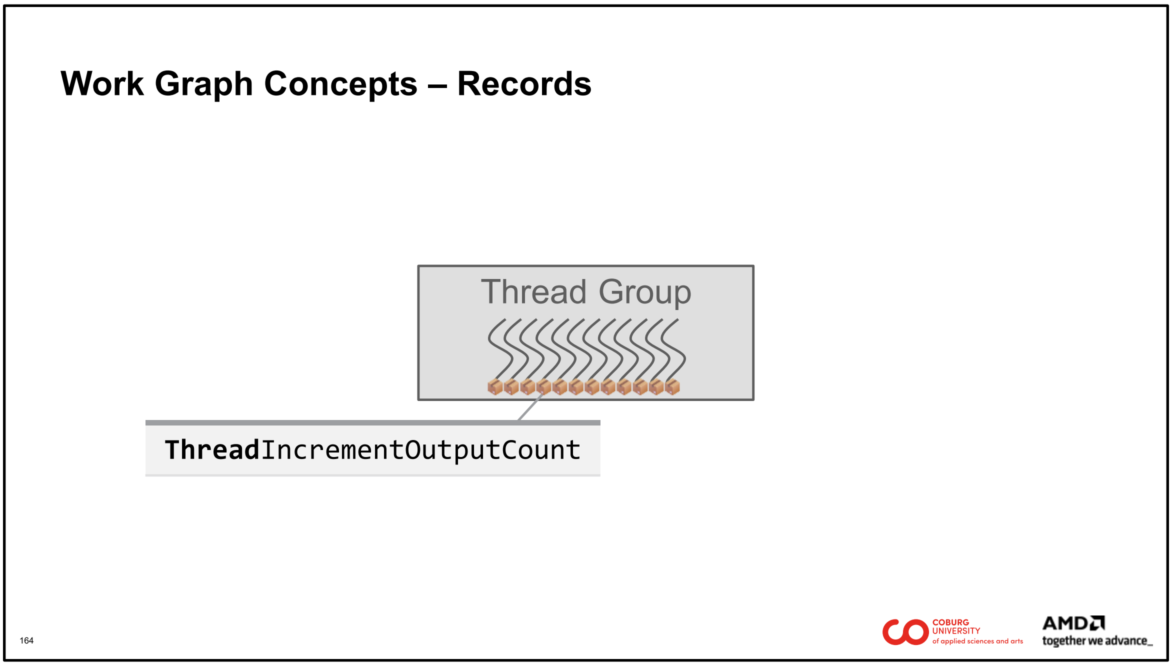
```
[Shader("node")]
...
[NumThreads(4, 1, 1)]
void Entry(
    [MaxRecords(1)]
    EmptyNodeOutput PrintHelloWorld
) {
    // [Task 1]: Emit a single empty record
    //           to the "PrintHelloWorld" node.
    PrintHelloWorld.GroupIncrementOutputCount(1);
}
```

163

To solve this problem, we would either have to change the code to only have a single thread increment the output count, or we can use

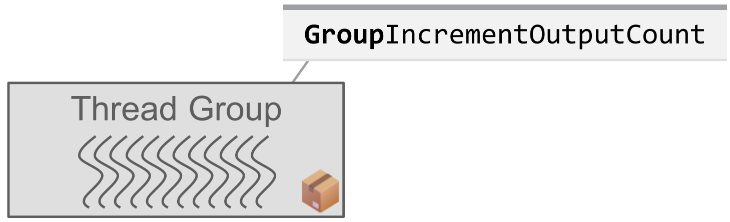
```
PrintHelloWorld.GroupIncrementOutputCount(1);
```

instead. As the name implies, this will increment the output count, i.e., send an empty record once per thread group instead of once per thread.

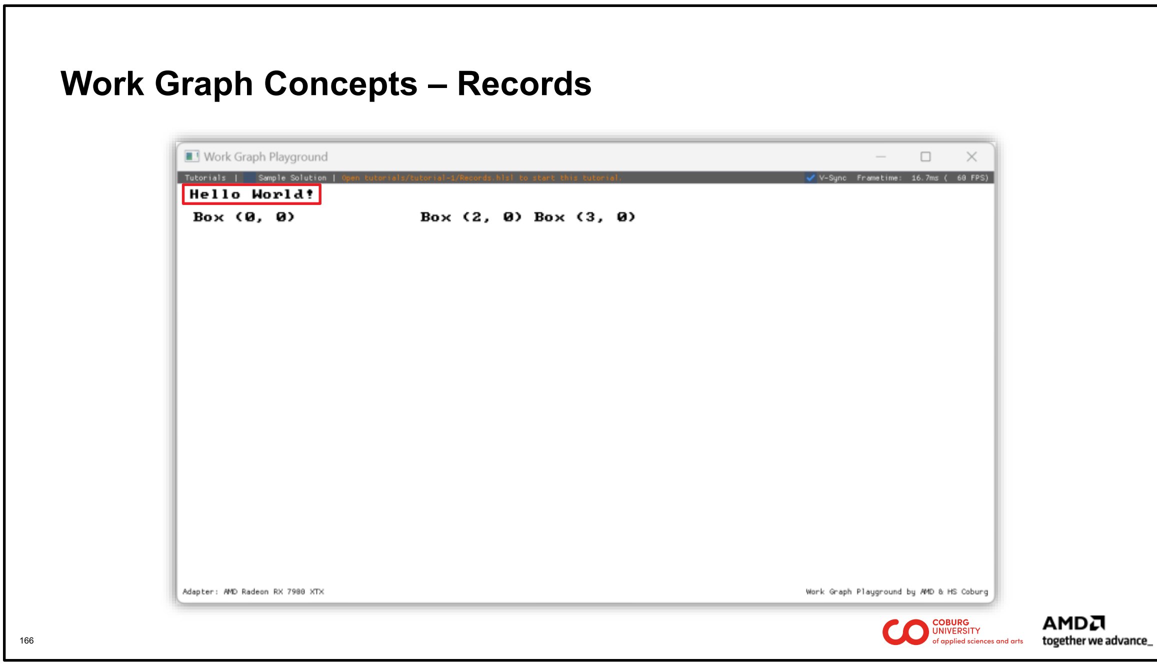


To summarize the difference, consider a thread group: Each wiggly line represents a thread of the thread group. If we call `ThreadIncrementOutputCount`, every single thread emits a single record, indicated by the package at the bottom of each wiggly line.

Work Graph Concepts – Records



If you call `GroupIncrementOutputCount`, instead, the entire group outputs a single record.

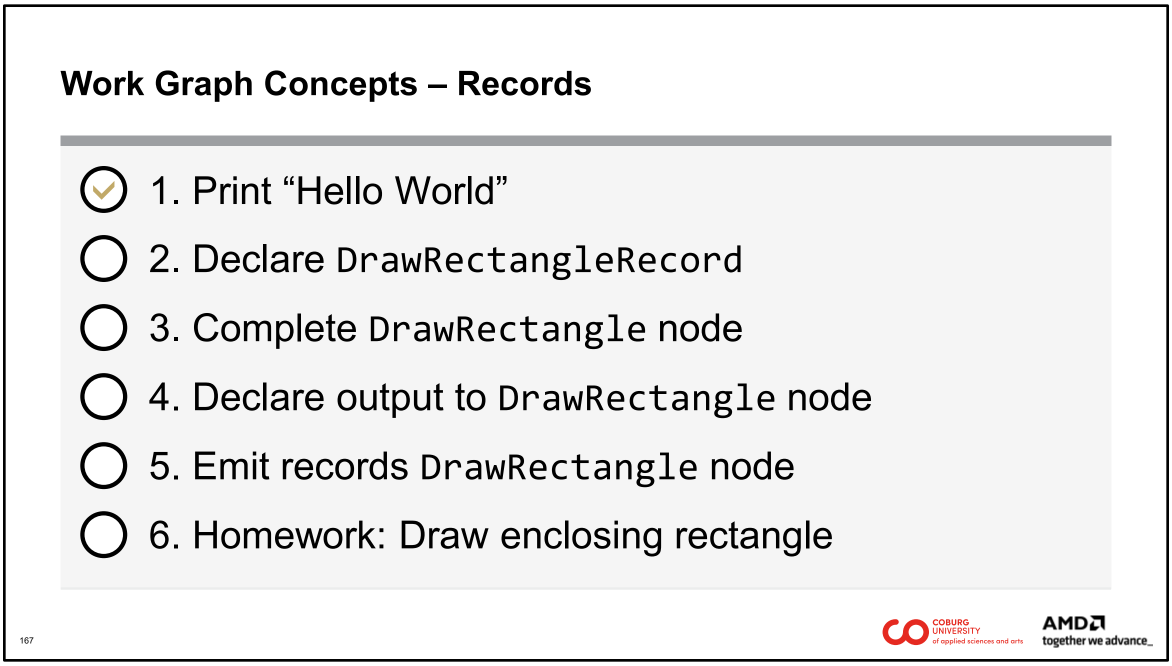


Once you complete Task 1, i.e., by adding the statement

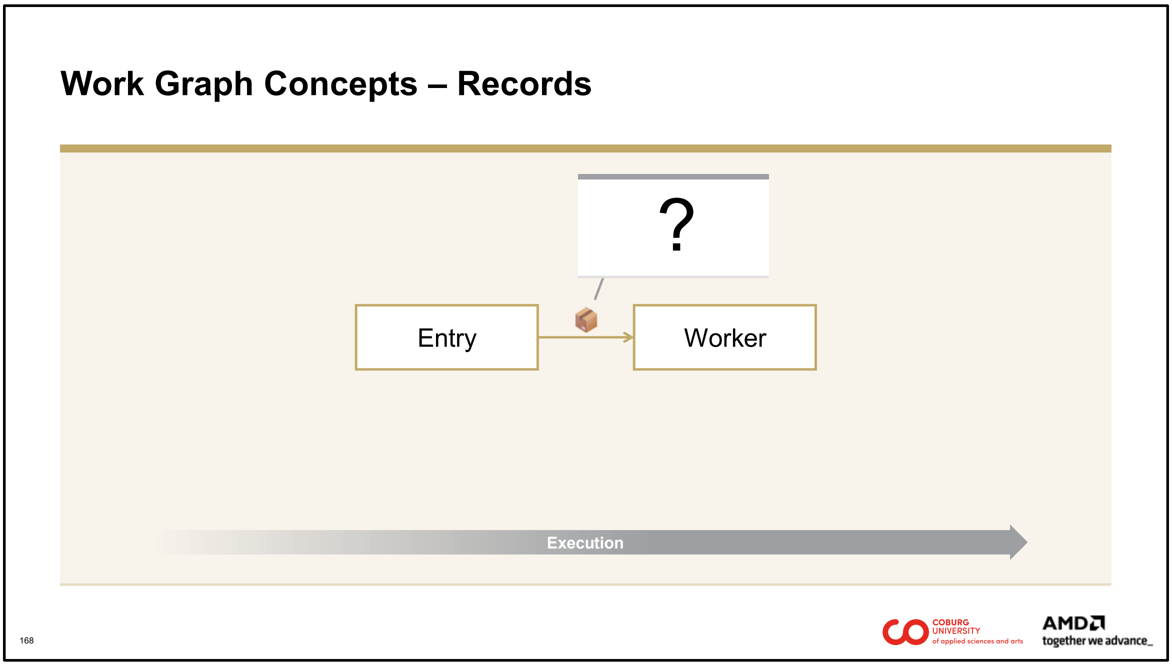
```
PrintHelloWorld.GroupIncrementOutputCount(1);
```

at the appropriate location, you should see a `Hello World` message (without the red box) on your screen.

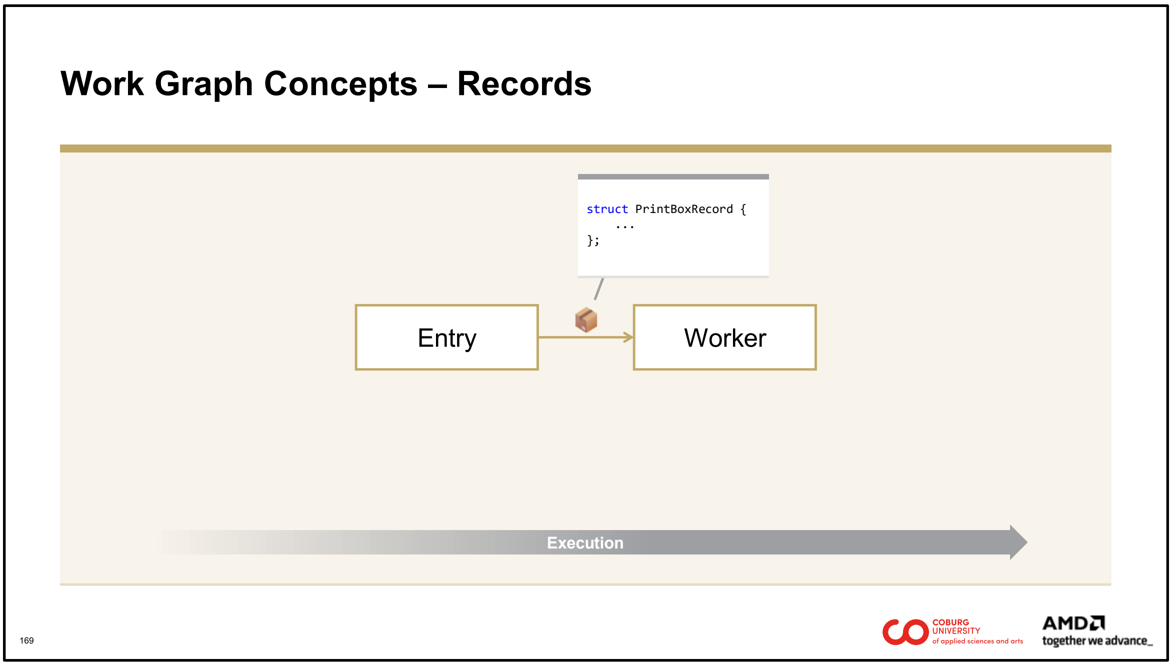
Hint: This will become important again for Task 6 of this tutorial.



This concludes our first task.



Before we go on to the next task, we must finally tell you, how to add data to the record. So far, all the records that we have sent were empty.



Next, we will add some data to it to parameterize a node launch.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
struct PrintBoxRecord {
    // Top-left pixel coordinate for a box.
    int2 topLeft;
    // Index to print inside the box.
    int2 index;
};
```

170

In Work Graphs, we use `struct`s to specify the data layout of the record's payload. Here you seen an example of such a `struct`.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
[Shader("node")]
...
void Entry(
    [MaxRecords(4)]
    [NodeId("PrintBox")]
    NodeOutput<PrintBoxRecord> boxOutput
) {
}
```

Max. 256 Records

171

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

To enable `Entry` node to emit such a record, we must specify three things:

1. We must specify, that the `Entry` node emits records, whose data structure is defined by `struct PrintBoxRecord`. We do this by adding `NodeOutput<PrintBoxRecord> boxOutput` to the node's function parameter list. This is similar to the `EmptyNodeOutput` we were using before, but with `NodeOutput<...>`, we can specify the type of data or payload that we want to send with each record.
2. We must specify which node consumes those records. We do this by adding the attribute `[NodeId("PrintBox")]` to the parameter `boxOutput`. Here, the node `PrintBox` receives those records.
3. Finally, we must provide an upper bound for the number of records the producer may output. This is done by yet another attribute attached to `boxOutput`, i.e. `[MaxRecords(4)]`.

You can send up to 256 records per thread group across all of its `NodeOutput` parameters.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
[Shader("node")]
...
void Entry(
    NodeOutput<PrintBoxRecord> boxOutput,
    NodeOutput<...> ...,
    NodeOutput<...> ...,
    NodeOutput<...> ...
) {
}
```

Max. 1024 Outputs

Max. 32 kiB Output Size

172

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

If you have multiple `NodeOutputs`, make sure that the total number of all `NodeOutputs` of a given node does not exceed 1024 `NodeOutputs` per thread group.

Further, the total amount of memory that all of these `NodeOutputs` combined may produce must not exceed 32 kiB.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
[Shader("node")]
...
void Entry(
    [MaxRecords(4)]
    [NodeId("PrintBox")]
    NodeOutput<PrintBoxRecord> boxOutput
) {
}
```


What we see here is that this node is capable of sending four output records. However, we have not yet seen, how this node does send records.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
[Shader("node")]
...
void Entry(
    ...
    NodeOutput<PrintBoxRecord> boxOutput
) {
    ...
    ThreadNodeOutputRecords<PrintBoxRecord> boxOutputRecord =
        boxOutput.GetThreadNodeOutputRecords(hasBoxOutput ? 1 : 0);

    if (hasBoxOutput) {
        boxOutputRecord.Get(0).topLeft = threadBoxPosition;
        boxOutputRecord[0].index       = dispatchThreadId;
    }

    boxOutputRecord.OutputComplete();
}
```

174

Here is how we actually send out records from our node.

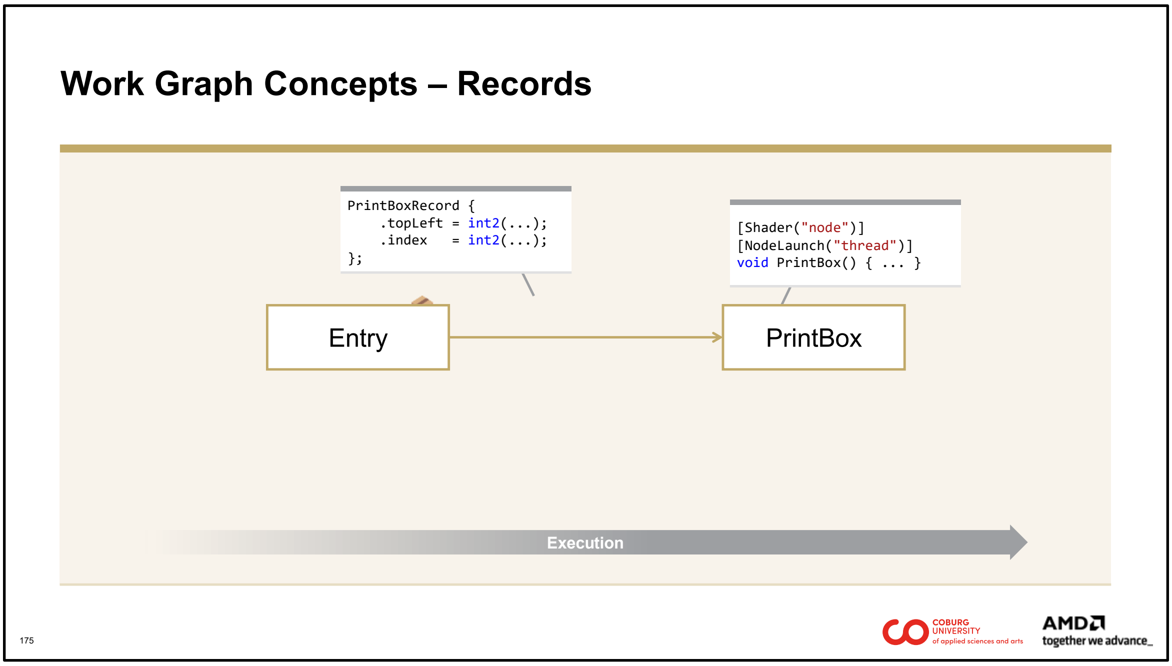
First, we obtain `ThreadNodeOutputRecords` from the `NodeOutput` by calling `GetThreadNodeOutputRecords`. The parameter of that function specifies the number of output records per thread we want to write and send. Here, we want to output either 0 or 1 record per thread. The decision whether a given thread wants to output a record is stored in a per-thread boolean `hasBoxOutput`.

Calling `ThreadNodeOutputRecords` must be thread-group uniform. That means, `ThreadNodeOutputRecords` must be called by all threads in lock-step at the same time by all threads of the thread-group. Otherwise, you can run into undefined behavior, which may result in crashes. With the tertiary operator (i.e., `hasBoxOutput ? 1 : 0`) inside the parameter list of `GetThreadNodeOutputRecords`, we can assure that all threads call this function, even if some threads (i.e., those with `hasBoxOutput = false`) do not with to output a record.

If a given thread needs to send an output, we must fill the record. To get access to the individual `PrintBoxRecord`, you can either use the `Get` function or the `[]`-operator on the `ThreadNodeOutputRecords`. The provided parameter is the index to the record. Here, we only have one output per thread and its index is `0`.

With the access to the record, you can read/write the member variables of the particular record `struct`.

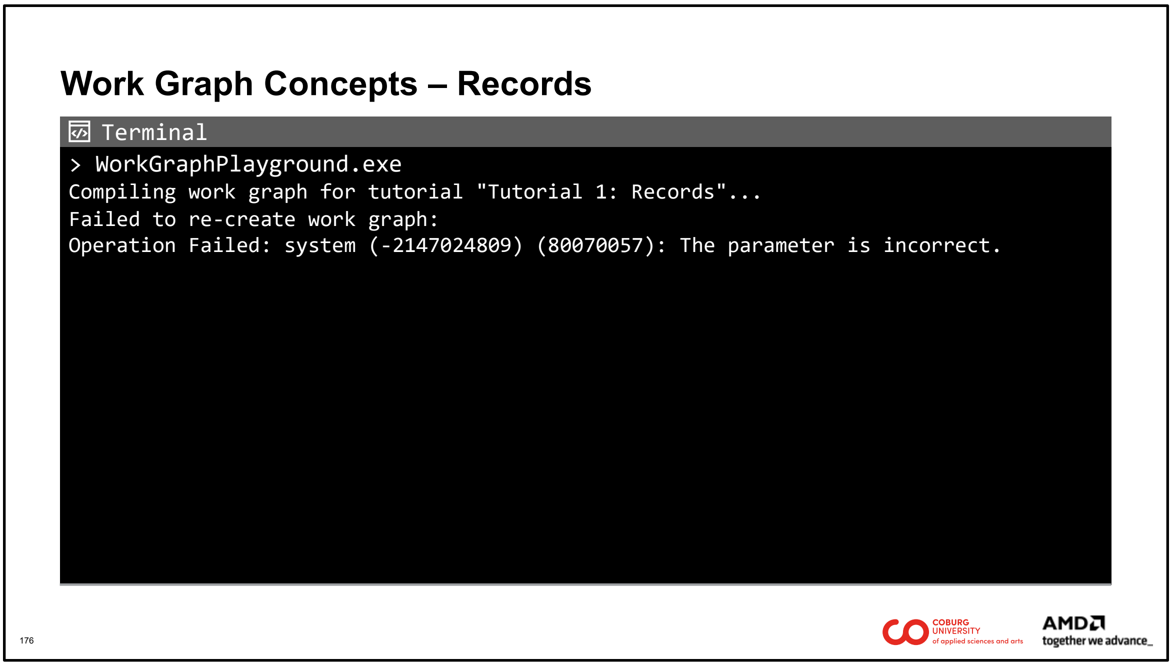
Once all records are filled, you can send it off, by calling `OutputComplete()` on the `ThreadNodeOutputRecords` variable, again in a thread-uniform fashion.



Let’s summarize what has happened so far. We obtained, filled, and send the record to the `PrintBox` node…

… but the `PrintBox` node has no idea that it is supposed to received a record …

… and therefore, our Work Graph Playground App crashes.



Here is what you will probably see as an output. We only see the

```
The parameter is incorrect.
```

error message. This is hinting to us that something about our work graph is not correct.

Work Graph Concepts – Records

```
Terminal
> WorkGraphPlayground.exe --enableDebugLayer
Compiling work graph for tutorial "Tutorial 1: Records"...
[D3D12] ID3D12Device::CreateStateObject: Autopopulated node "Entry" targets output
node PrintBox with an output record size of 16 bytes, but the target node expects an
input record of size 0 bytes.  These must match.
Failed to re-create work graph:
Operation Failed: system (-2147024809) (80070057): The parameter is incorrect.
```

177

To better understand the crash, we encourage you to execute `WorkGraphPlaygroundApp.exe` with the command line parameter shown here*.

Then, you will get meaningful error messages. Here, for example, you see the problem: The producer and consumer node did not agree on the record size. The work graph validation will fail and reports an error.

*Please note that the D3D12 debug layer requires Graphics diagnostic tools to be installed. You can find more information here: https://learn.microsoft.com/en-us/windows/uwp/gaming/use-the-directx-runtime-and-visual-studio-graphics-diagnostic-features

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
struct PrintBoxRecord { ... };

[Shader("node")]
[NodeLaunch("thread")]
void PrintBox(
    ThreadNodeInputRecord<PrintBoxRecord> inputRecord
) {
    ...
}
```

178

To fix this problem, we must specify that the consumer node `PrintBox` accepts an input record. This is by adding

```
ThreadNodeInputRecord<PrintBoxRecord> inputRecord
```

to the parameter list of the corresponding node function `PrintBox`.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
struct PrintBoxRecord { ... };

[Shader("node")]
[NodeLaunch("thread")]
void PrintBox(
    ThreadNodeInputRecord<PrintBoxRecord> inputRecord
) {
    ...
}
```


The template argument is the `struct` that defines the record’s data layout, `PrintBoxRecord`.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
[Shader("node")]
[NodeLaunch("thread")]
void PrintBox(
     ThreadNodeInputRecord<PrintBoxRecord> inputRecord
) {
    const PrintBoxRecord record = inputRecord.Get();

    Cursor cursor = Cursor(record.topLeft + ...);
}
```

180

To get read access to the payload, we call the `.Get()` method on the `inputRecord`…

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
[Shader("node")]
[NodeLaunch("thread")]
void PrintBox(
     ThreadNodeInputRecord<PrintBoxRecord> inputRecord
) {
    const PrintBoxRecord record = inputRecord.Get();

    Cursor cursor = Cursor(record.topLeft + ...);
}
```

181

… and obtain a `const`, i.e., read-only, instance to the `struct`, which we store to a local variable `record` for easier access.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

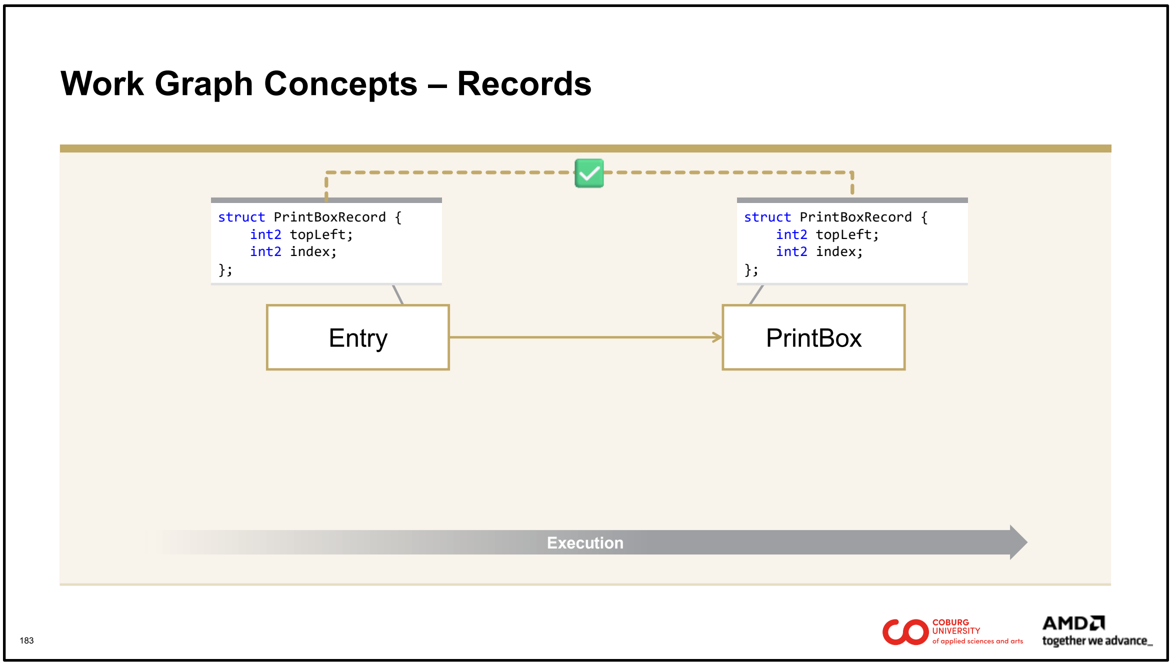
```
[Shader("node")]
[NodeLaunch("thread")]
void PrintBox(
    ThreadNodeInputRecord<PrintBoxRecord> inputRecord
) {
    const PrintBoxRecord record = inputRecord.Get();

    Cursor cursor = Cursor(record.topLeft + ...);
}
```

182

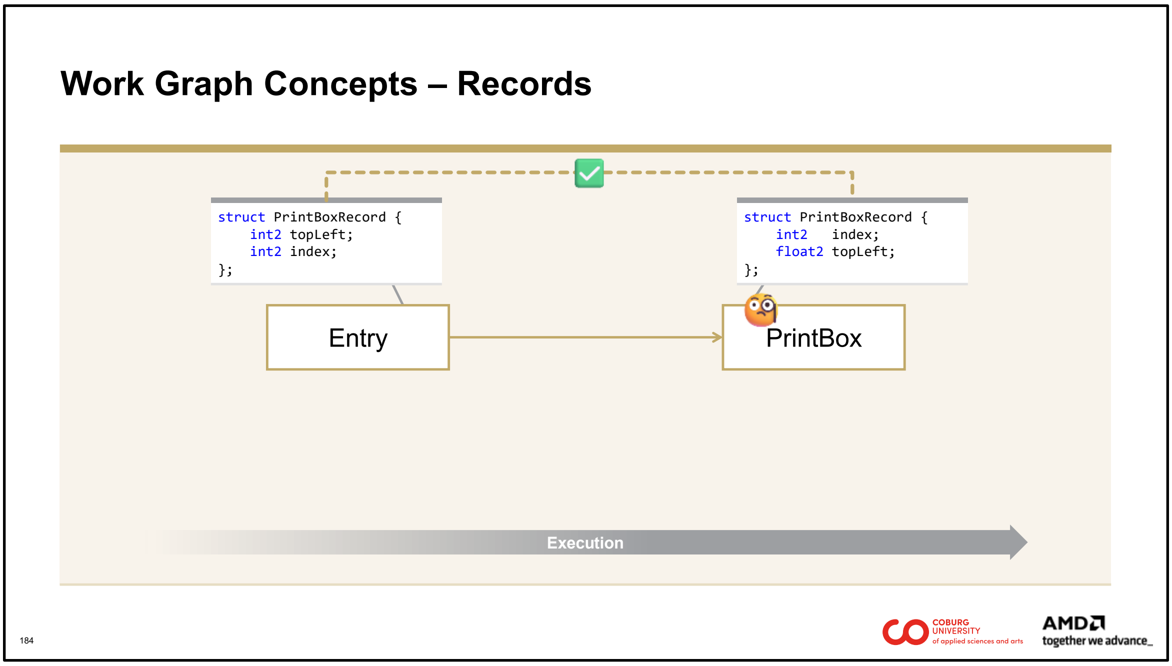
COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

We can now access the `struct`'s members and use it for further processing.



With the producer and consumer now using the same record definition, we have successfully connected the two nodes. The validation errors are now gone.



But beware: the Work Graphs validation only ensures that the size of the output- and input-record match. This example would still be accepted by the validation, even producer and consumer have different definitions of the record's layout. This can cause you hard-to-find errors.

Work Graph Concepts – Records

- ✓ 1. Print “Hello World”
- ○ **2. Declare `DrawRectangleRecord`**
- ○ 3. Complete `DrawRectangle` node
- ○ 4. Declare output to `DrawRectangle` node
- ○ 5. Emit records `DrawRectangle` node
- ○ 6. Homework: Draw enclosing rectangle

185

Now, you are ready to do Task-2, declare a draw rectangle record.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
// [Task 2]: Define a struct for the "DrawRectangle" node
```

Common.h

```
// Draws the outline of a rectangle spanning from "topLeft" to "bottomRight" in window coordinates with a "thickness" in
// pixels. Thickness extends symmetrically to inside and outside of outline. Use "color" to specify the RGB values of
// the rendered rectangle outline pixels.
void DrawRect(in const float2 topLeft,
              in const float2 bottomRight,
              in const float  thickness = 1,
              in const float3 color     = float3(0, 0, 0));
```

186

Task 2: Create the record `struct` to draw a rectangle around all boxes. Take a look at the prepared stub for the "`DrawRectangle`" node to see what data needs to be passed to the record.

Hint: you see that `DrawRect` should be called. The function is defined in `tutorials/Common.h` (line 570) and has the following signature

```
void DrawRect(in const float2 topLeft,
              in const float2 bottomRight,
              in const float  thickness = 1,
              in const float3 color     = float3(0, 0, 0))
```

From this you can infer what your record `struct` should look like.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
// [Task 2]: Define a struct for the "DrawRectangle" node

struct DrawRectangleRecord {
    // Pixel coordinate of top-left corner of rectangle.
    int2   topLeft;
    // Pixel coordinate of bottom-right corner of rectangle.
    int2   bottomRight;
    // Color of the rectangle.
    float3 color;
};
```

187

Here is our suggested solution.

Work Graph Concepts – Records

- [x] 1. Print "Hello World"
- [x] 2. Declare `DrawRectangleRecord`
- [ ] 3. Complete `DrawRectangle` node
- [ ] 4. Declare output to `DrawRectangle` node
- [ ] 5. Emit records `DrawRectangle` node
- [ ] 6. Homework: Draw enclosing rectangle

We got Task-2 done.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
[Shader("node")]
...
void DrawRectangle(
    // [Task 3]: Declare a node input with your new.
) {
    // [Task 3]: Use the DrawRect function to draw a rectangle.
}
```

Common.h

```
// Draws the outline of a rectangle spanning from "topLeft" to "bottomRight" in window coordinates with a "thickness" in
// pixels. Thickness extends symmetrically to inside and outside of outline. Use "color" to specify the RGB values of
// the rendered rectangle outline pixels.
void DrawRect(in const float2 topLeft,
              in const float2 bottomRight,
              in const float  thickness = 1,
              in const float3 color     = float3(0, 0, 0));
```

189

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

Next, we draw the rectangle.

Task 3: Add your record `struct` as an input to the `DrawRectangle` node and complete the code in the node to draw a rectangle on screen.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
[Shader("node")]
...
void DrawRectangle(
    // [Task 3 Solution]:
    ThreadNodeInputRecord<DrawRectangleRecord> inputRecord
) {
    // [Task 3 Solution]:
    const DrawRectangleRecord record = inputRecord.Get();
    DrawRect(
        record.topLeft, record.bottomRight, 1, record.color);
}
```


Here is our suggested solution.

Work Graph Concepts – Records

- ☑ 1. Print “Hello World”
- ☑ 2. Declare `DrawRectangleRecord`
- ☑ 3. Complete `DrawRectangle` node
- ☐ 4. Declare output to `DrawRectangle` node
- ☐ 5. Emit records `DrawRectangle` node
- ☐ 6. Homework: Draw enclosing rectangle

191

Next, we have to declare an output record.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
[Shader("node")]
...
void Entry(
    // [Task 4]: Declare a new "NodeOutput"
    //           to the "DrawRectangle" node.



) {
...
}
```

192

Task 4: Add a node output to the `Entry` node for `DrawRectangle` node with your newly created `record` struct. For now, we only care about the boxes around the already existing text, thus each thread will emit a single record. Set the `[MaxRecords(...)]` attribute for your accordingly.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
[Shader("node")]
...
void Entry(
    // [Task 4 Solution]:
    [MaxRecords(4)]
    [NodeId("DrawRectangle")]
    NodeOutput<DrawRectangleRecord> rectangleOutput

) {
...
}
```

193

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

Here is our suggest solution.

Work Graph Concepts – Records

- 1. Print "Hello World" (done)
- 2. Declare `DrawRectangleRecord` (done)
- 3. Complete `DrawRectangle` node (done)
- 4. Declare output to `DrawRectangle` node (done)
- 5. Emit records `DrawRectangle` node
- 6. Homework: Draw enclosing rectangle

194

We declared our output, next we have to fill and emit it.

Work Graph Concepts – Records

tutorial-1/Records.hlsl

```
[Shader("node")]
...
void Entry(...) {
    // [Task 5]: Emit a record to draw a rectangle.
}
```

195

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

Task 5: Emit the record to the `DrawRectangle` node from the `Entry` node.

Work Graph Concepts – Records

```
tutorial-1/Records.hlsl
[Shader("node")]
...
void Entry(...) {
    // [Task 5 Solution]:
    ThreadNodeOutputRecords<DrawRectangleRecord> threadRectangleRecord =
        rectangleOutput.GetThreadNodeOutputRecords(hasBoxOutput ? 1 : 0);

    if (hasBoxOutput) {
       threadRectangleRecord.Get().topLeft     = threadBoxPosition;
       threadRectangleRecord.Get().bottomRight = threadBoxPosition + BoxSize;
       threadRectangleRecord.Get().color       = float3(0, 0, 0);
    }

    threadRectangleRecord.OutputComplete();
}
```

196

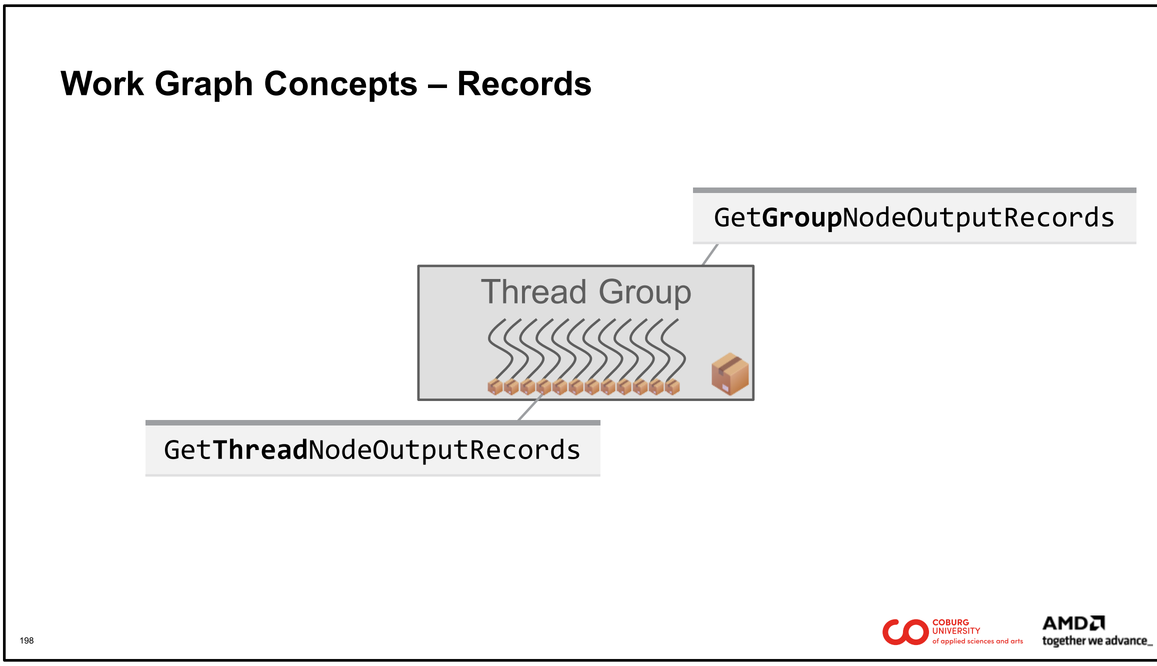
And here is our suggest solution.

Work Graph Concepts – Records

1. Print "Hello World" ✓
2. Declare `DrawRectangleRecord` ✓
3. Complete `DrawRectangle` node ✓
4. Declare output to `DrawRectangle` node ✓
5. Emit records `DrawRectangle` node ✓
6. Homework: Draw enclosing rectangle

197

As a homework, look at the last task.



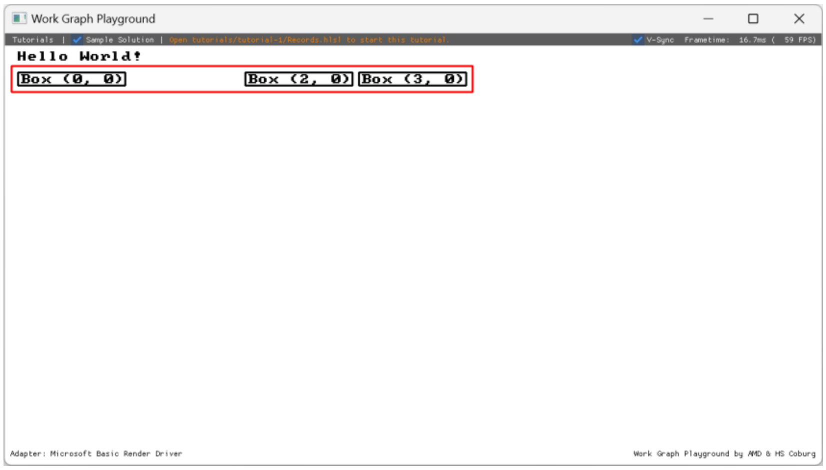
To better give an idea of what awaits you in Task-6, let's look at another way to send out records. Up until now, we have used `ThreadNodeOutputRecords`, i.e., each thread of our thread-group outputs a record.

<Next Animation Slide>

In your homework Task-6, we want that the entire thread-group to output a record. This can be done using `GroupNodeOutputRecords`.

This behavior is similar to **Thread**`IncrementOutputCount` and **Group**`IncrementOutputCount`, but for a non-empty record.

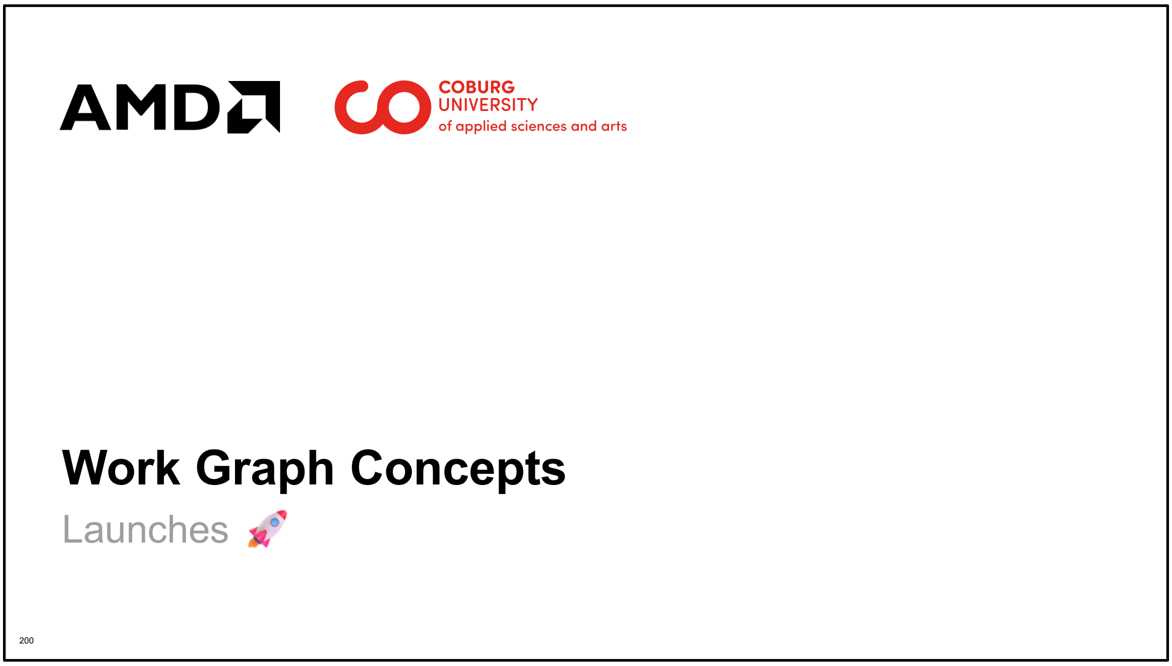
Work Graph Concepts – Records



199

Task 6: Additionally, we now want to draw another rectangle around all of these boxes. Update the `[MaxRecords(...)]` attribute of your node output and follow the instructions below to emit a per-thread-group record.

After completing the task, you should see a box around all boxes you have drawn so far. So good luck and have fun!



In the last section of the Work Graph Concepts block, we will cover "Launches". We slightly touched on launches in the Nodes and Records section, but here we give you the full details.

Work Graph Concepts – Launches 🚀

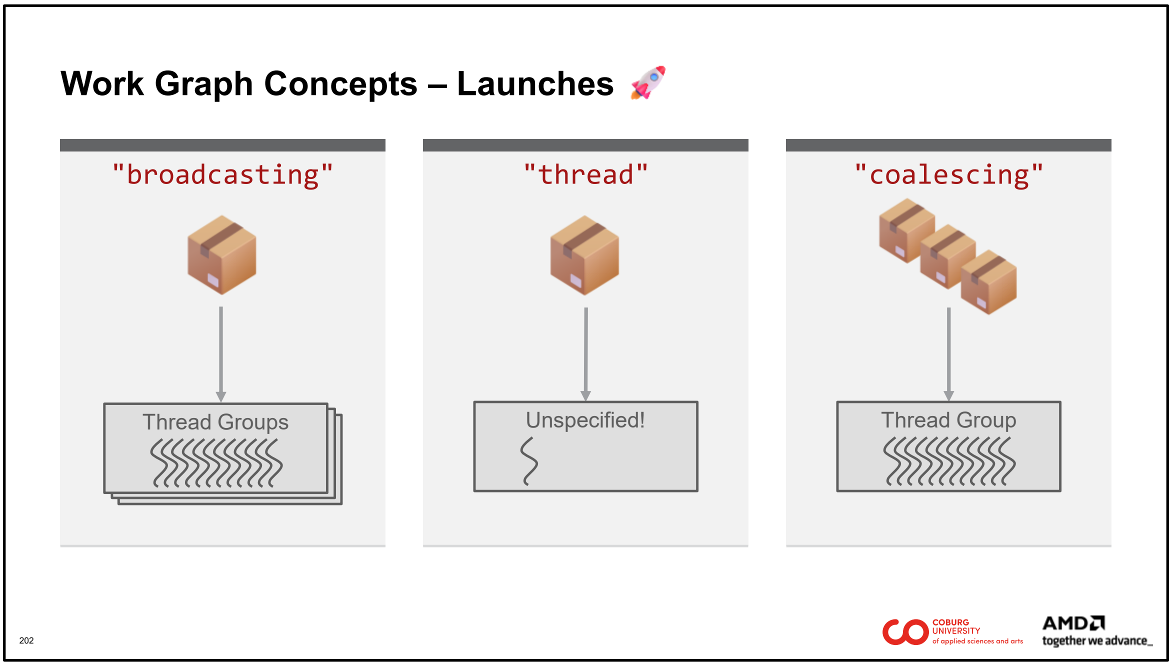
- 1. Change FillRectangle to dynamic dispatch grid
- 2. Implement pass-through coalescing node
- 3. Merge adjacent rectangles
- 4. Non-deterministic coalescers

201

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

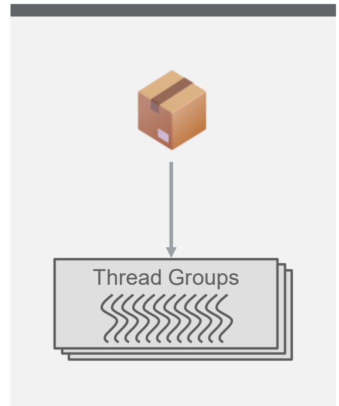
In this tutorial, we have four tasks in which we are going to learn how these different node launch modes work. In the following, we'll highlight each of these tasks and explain the concept of launches and launch modes.



We've seen before that we can specify the "work" in our work graph with records. The launch mode then specifies how each node function is processing the incoming records. In Work Graphs, we have access to three different launch modes: `"broadcasting"`, `"thread"`, and `"coalescing"`.

Work Graph Concepts – "broadcasting" Launches 🚀

```
tutorial-2/NodeLaunches.hlsl
struct RectangleRecord {
  ...
};

[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeDispatchGrid(6, 6, 1)]
[NumThreads(8, 8, 1)]
[NodeId("FillRectangle")]
void FillRectangleNode(
  DispatchNodeInputRecord<RectangleRecord> ir,

  uint2 dispatchThreadId : SV_DispatchThreadID
) { ... }
```



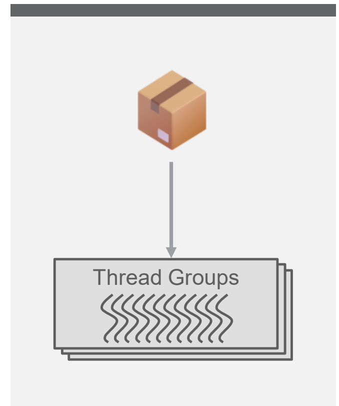
203

Let's start with the "broadcasting" launch mode, since it is the easiest to grasp if you have every worked with compute shaders before. If we use the "broadcasting" launch mode, one record is processed by a grid of thread groups.

Work Graph Concepts – "broadcasting" Launches 🚀

tutorial-2/NodeLaunches.hlsl

```
struct RectangleRecord {
  ...
};

[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeDispatchGrid(6, 6, 1)]
[NumThreads(8, 8, 1)]
[NodeId("FillRectangle")]
void FillRectangleNode(
  DispatchNodeInputRecord<RectangleRecord> ir,

  uint2 dispatchThreadId : SV_DispatchThreadID
) { ... }
```



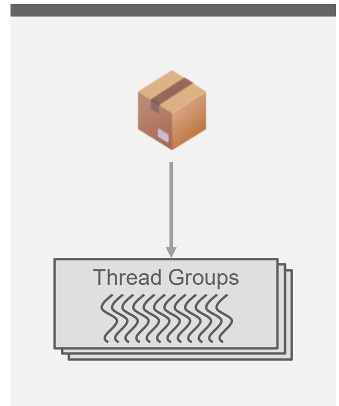
204

Launching a node in `"broadcasting"` launch mode is very similar to dispatching a compute shader kernel. Thus, the input record is declared with type **`Dispatch`**`NodeInputRecord`. This way, thread groups launches for the same records all receive a read-only view to the input record.

Work Graph Concepts – "broadcasting" Launches 🚀

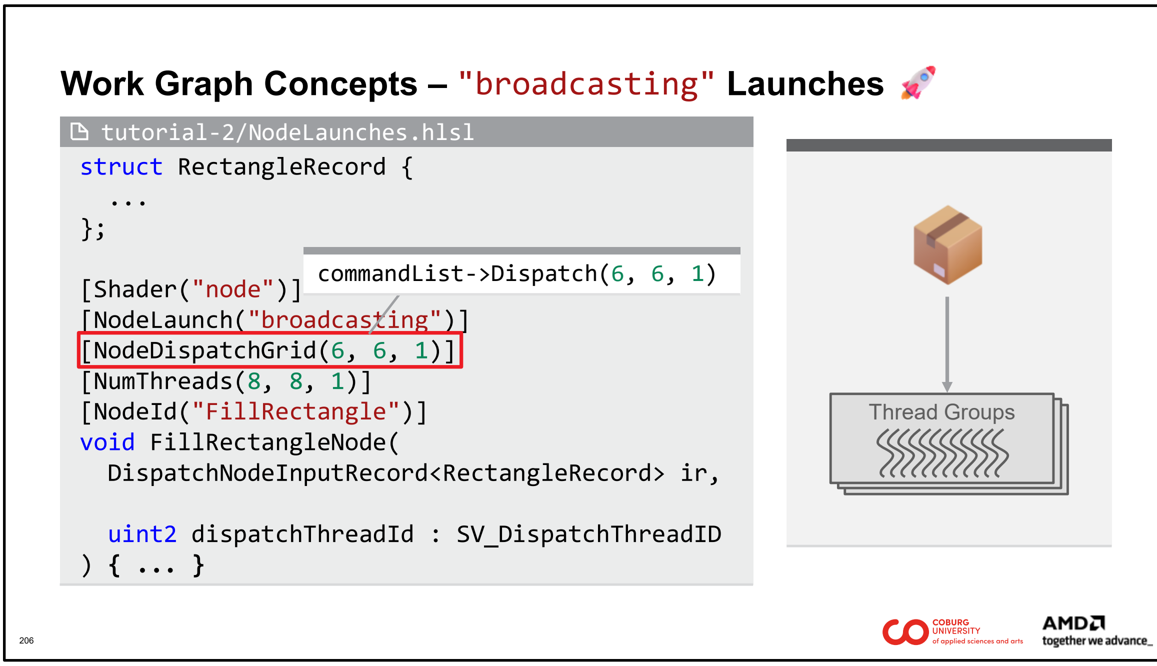
tutorial-2/NodeLaunches.hlsl

```
struct RectangleRecord {
  ...
};

[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeDispatchGrid(6, 6, 1)]
[NumThreads(8, 8, 1)]
[NodeId("FillRectangle")]
void FillRectangleNode(
  DispatchNodeInputRecord<RectangleRecord> ir,

  uint2 dispatchThreadId : SV_DispatchThreadID
) { ... }
```



205

As with any regular compute shader, we define the three-dimensional grid of threads in each thread group with the `[NumThreads(...)]` attribute. In our example, we're using $8 \times 8 \times 1 = 64$ threads.

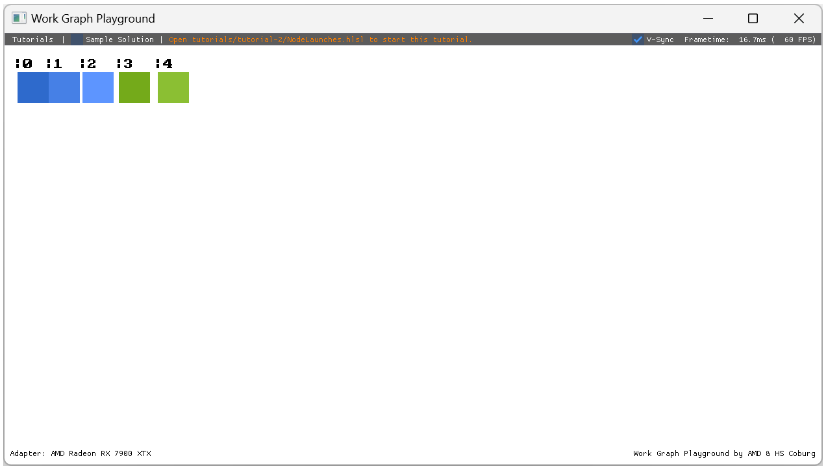


Similarly, the three-dimensional grid of thread groups to launch is defined with the `[NodeDispatchGrid(...)]` attribute.

Here, we a launch a grid of $6 \times 6 \times 1 = 36$ groups. This is similar to launching compute shader from the CPU with the `Dispatch` command.

However, statically setting the dispatch grid through `[NodeDispatchGrid(...)]` means that every incoming record launches the same number of thread groups. In many scenarios (e.g. image filters) we require a dynamic number of thread groups that fits the current problem size.

Work Graph Concepts – "broadcasting" Launches 🚀

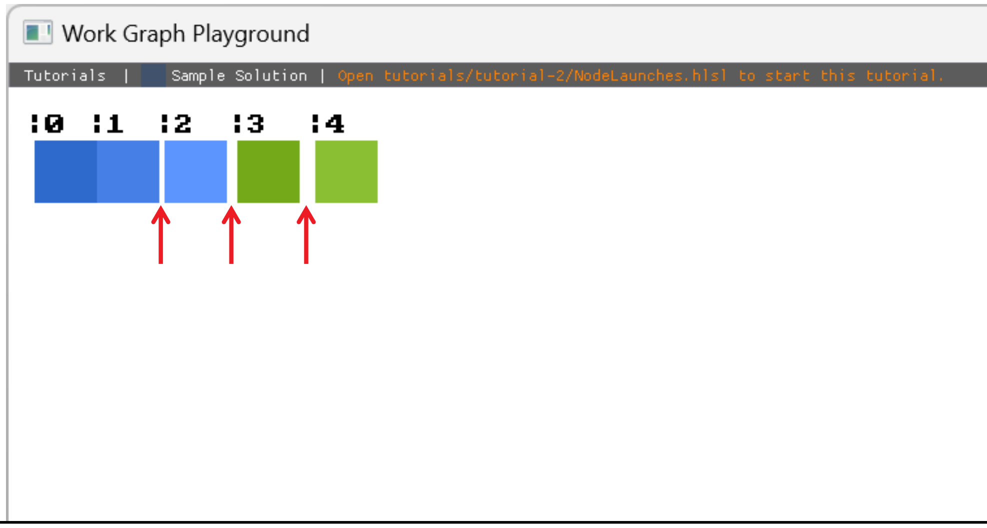


207

COBURG UNIVERSITY of applied sciences and arts
AMD together we advance_

We can see an example of this in the Node Launches tutorial.

Work Graph Concepts – "broadcasting" Launches 🚀



On screen, we see five colored blocks. These blocks are drawn by the `FillRectangle` node. The FillRectangle node uses the `"broadcasting"` launch mode and a fixed dispatch grid of `[NodeDispatchGrid(6, 6, 1)]`.

However, in the node function of the `Entry` node, we can see that each of these rectangles should have a different size, as computed by the `GetRectanglePositionAndSize` helper function.

To then draw each rectangle with the correct size, we must dynamically set the dispatch grid for each rectangle (i.e., each record). Follow the instructions for `[Task 1]` in `tutorials/tutorial-2/NodeLaunches.hlsl`.

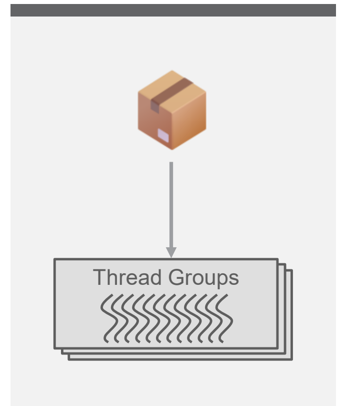
1. Start by adding variables for the dispatch grid and rectangle size in the `"RectangleRecord" struct`.
2. Next, change the `[NodeDispatchGrid(...)]` attribute of the `"FillRectangle“` node to a `[NodeMaxDispatchGrid(...)]` and update the dispatch size limit in the x dimension.
3. Lastly, set the dispatch grid and rectangle size for the rectangle records in the `"Entry"` node.

In the following, we'll discuss the sample solution.

Work Graph Concepts – "broadcasting" Launches 🚀

tutorial-2/NodeLaunches.hlsl

```
struct RectangleRecord {
  uint2 dispatchGrid : SV_DispatchGrid;
};

[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeDispatchGrid(6, 6, 1)]
[NumThreads(8, 8, 1)]
[NodeId("FillRectangle")]
void FillRectangleNode(
  DispatchNodeInputRecord<RectangleRecord> ir,

  uint2 dispatchThreadId : SV_DispatchThreadID
) { ... }
```



209

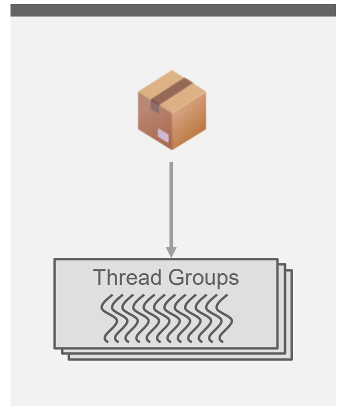
To dynamically set the dispatch grid for each record, we add a variable to the record struct and annotate it with the `SV_DispatchGrid` semantic. This semantic tells the work graph system, that this variable should be used as the dispatch grid for the broadcasting node. The type of this variable can be `uint`, `uint2`, `uint3` or a 16-bit variant of the aforementioned types.

With this, we have completed the first step of Task 1.

Work Graph Concepts – "broadcasting" Launches 🚀

tutorial-2/NodeLaunches.hlsl

```
struct RectangleRecord {
  uint2 dispatchGrid : SV_DispatchGrid;
};

[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeMaxDispatchGrid(16, 6, 1)]
[NumThreads(8, 8, 1)]
[NodeId("FillRectangle")]
void FillRectangleNode(
  DispatchNodeInputRecord<RectangleRecord> ir,

  uint2 dispatchThreadId : SV_DispatchThreadID
) { ... }
```



210

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

Next, we need to change the `[NodeDispatchGrid(...)]` attribute of the `FillRectangle` node to `[NodeMaxDispatchGrid(...)]`. Instead of setting a fixed dispatch grid for all incoming records, we now define an upper limit for the dispatch grid set by each individual record.

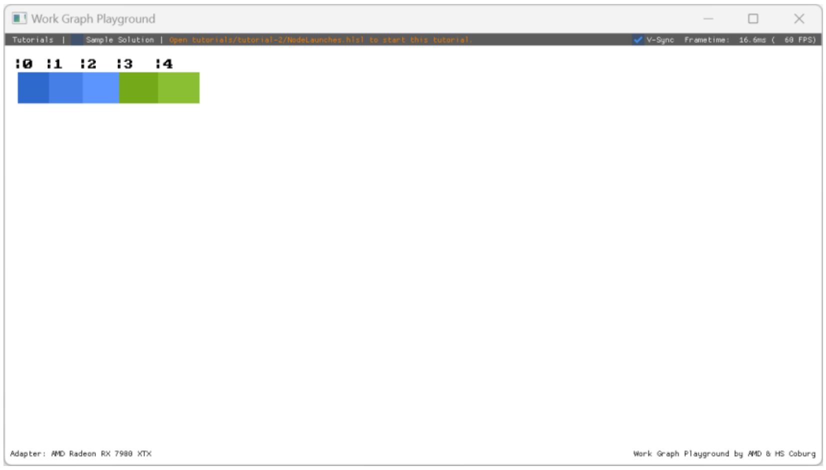
Beside replacing `NodeDispatchGrid` with `NodeMaxDispatchGrid`, we have to determine an upper limit for the grid size. As each thread in the `FillRectangle` node draws a single pixel, we compute the upper limit as follows:

- 6 thread groups for base-size rectangle (48x48)
- 10 thread groups (10x8 = 80 pixels) to cover the size of the 20th thread group (48 + 19 * 4)

Gives us a total of 16 thread groups max.

Finally, we need to set the newly added `dispatchGrid` variable in each of the records that we send to `FillRectangle`. We omitted this step here for simplicity, but you can refer to the sample solution or the previous tutorial on records for more information on writing data to records.

Work Graph Concepts – "broadcasting" Launches 🚀



211

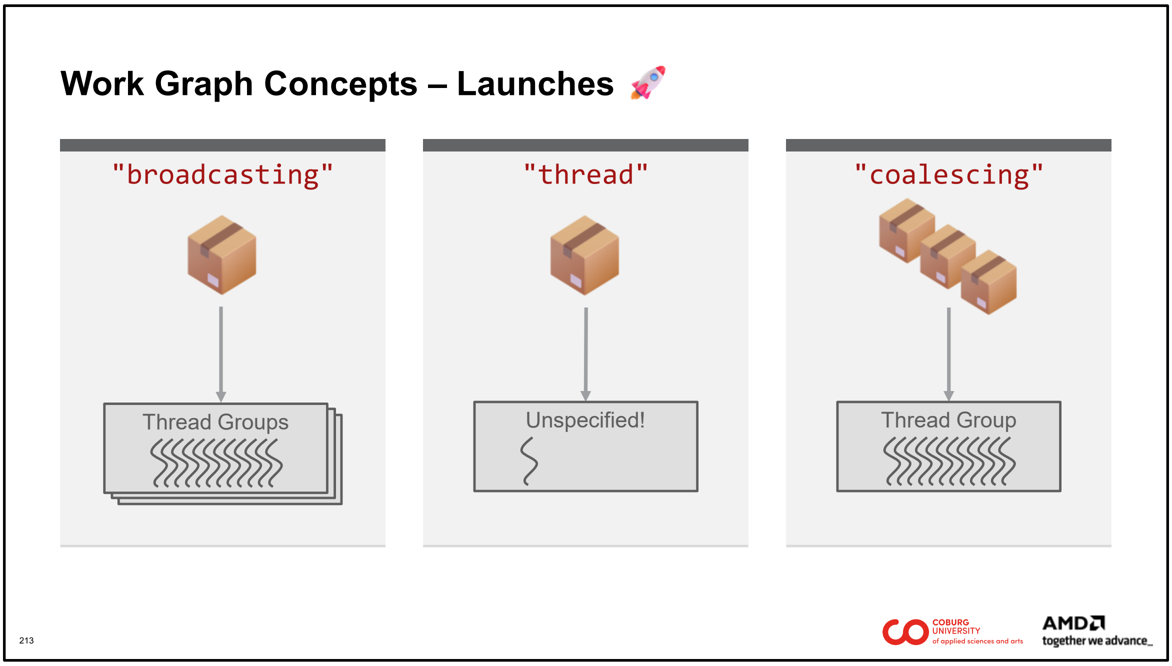
Once you're done with Task 1, the rectangles should now cover a continuous horizontal rectangle.

Work Graph Concepts – `"broadcasting"` Launches 🚀

- ✓ 1. Change FillRectangle to dynamic dispatch grid
- ○ 2. Implement pass-through coalescing node
- ○ 3. Merge adjacent rectangles
- ○ 4. Non-deterministic coalescers

212

This completes our look at Task 1 and the `"broadcasting"` launch mode.

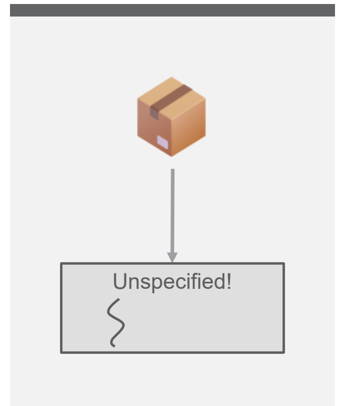


Next, we look at the `"thread"` launch mode.

Work Graph Concepts – "thread" Launches 🚀

tutorial-2/NodeLaunches.hlsl

```
[Shader("node")]
[NodeLaunch("thread")]
[NodeId("PrintLabel")]
void PrintLabelNode(
  ThreadNodeInputRecord<PrintLabelRecord> ir
) { ... }
```



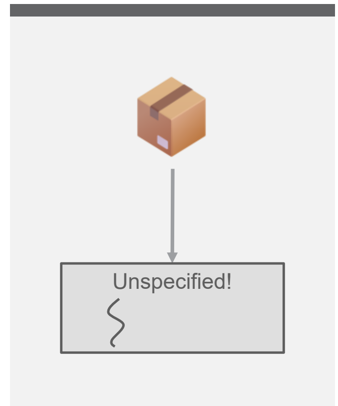
214

We use the `PrintLabelNode` to explain the `"thread"` launch mode. We’ve also seen similar used of the `"thread"` launch mode in the previous tutorials.

Work Graph Concepts – "thread" Launches 🚀

tutorial-2/NodeLaunches.hlsl

```
[Shader("node")]
[NodeLaunch("thread")]
[NodeId("PrintLabel")]
void PrintLabelNode(
  ThreadNodeInputRecord<PrintLabelRecord> ir
) { ... }
```



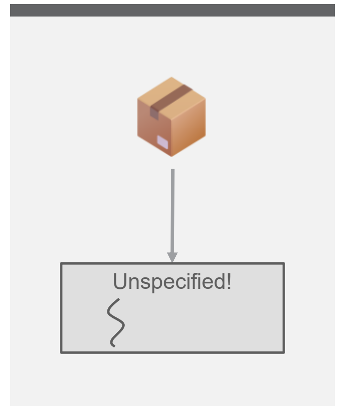
215

Again, we use the `[NodeLaunch(...)]` attribute to provide the launch mode.

Work Graph Concepts – "thread" Launches 🚀

tutorial-2/NodeLaunches.hlsl

```
[Shader("node")]
[NodeLaunch("thread")]
[NodeId("PrintLabel")]
void PrintLabelNode(
  ThreadNodeInputRecord<PrintLabelRecord> ir
) { ... }
```

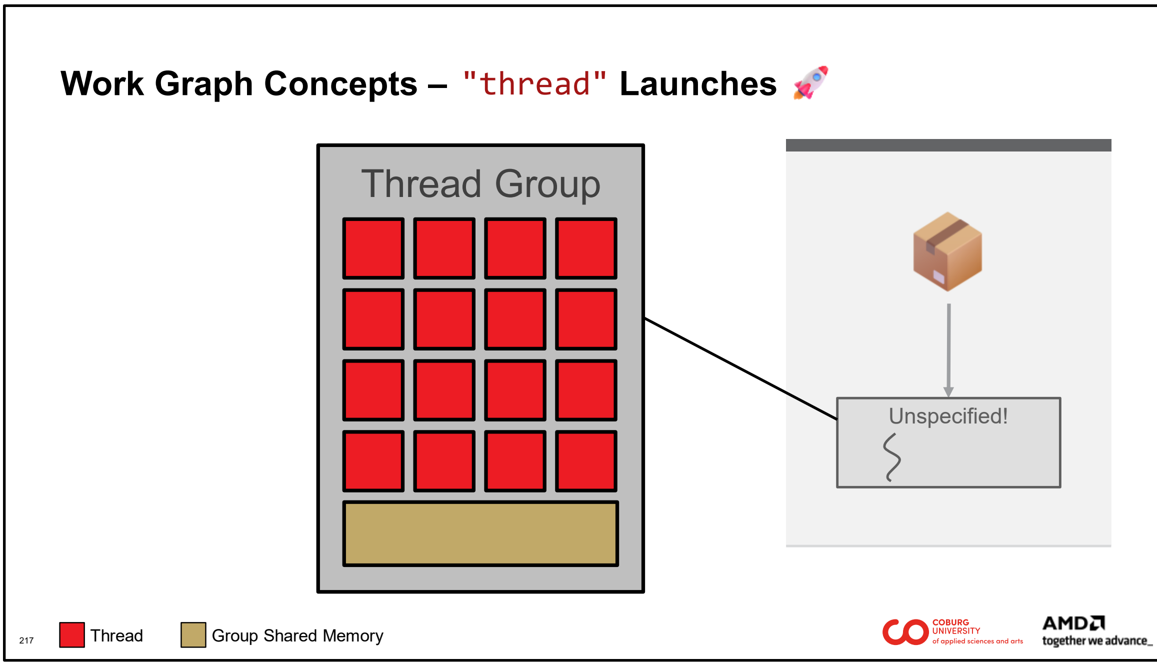


216

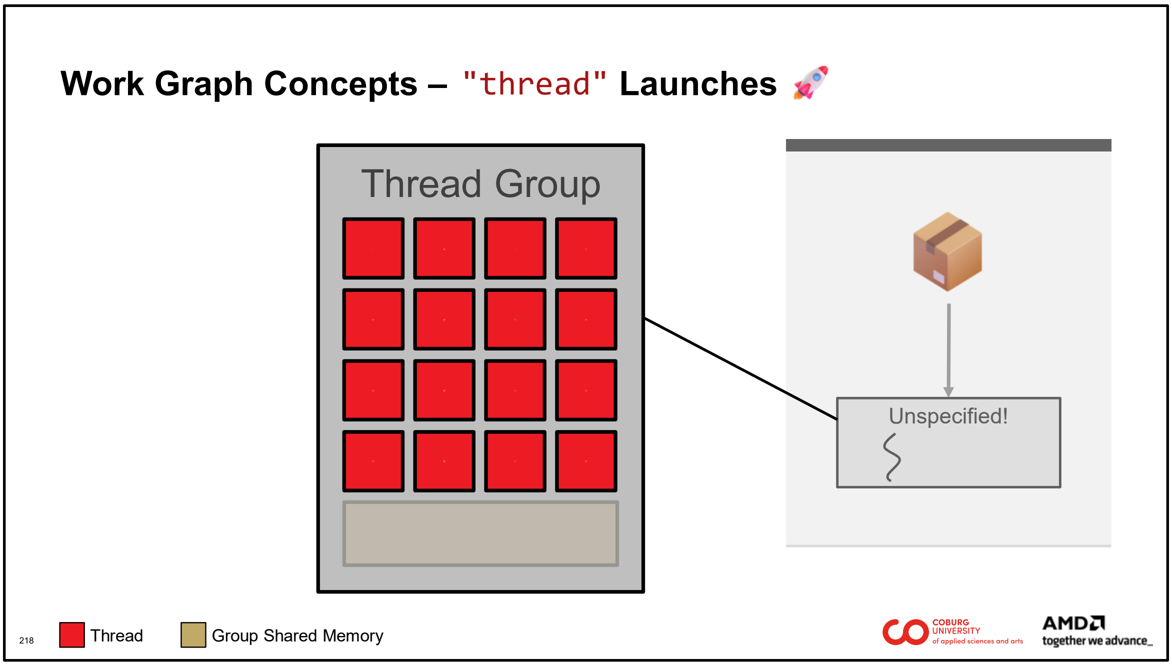
COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

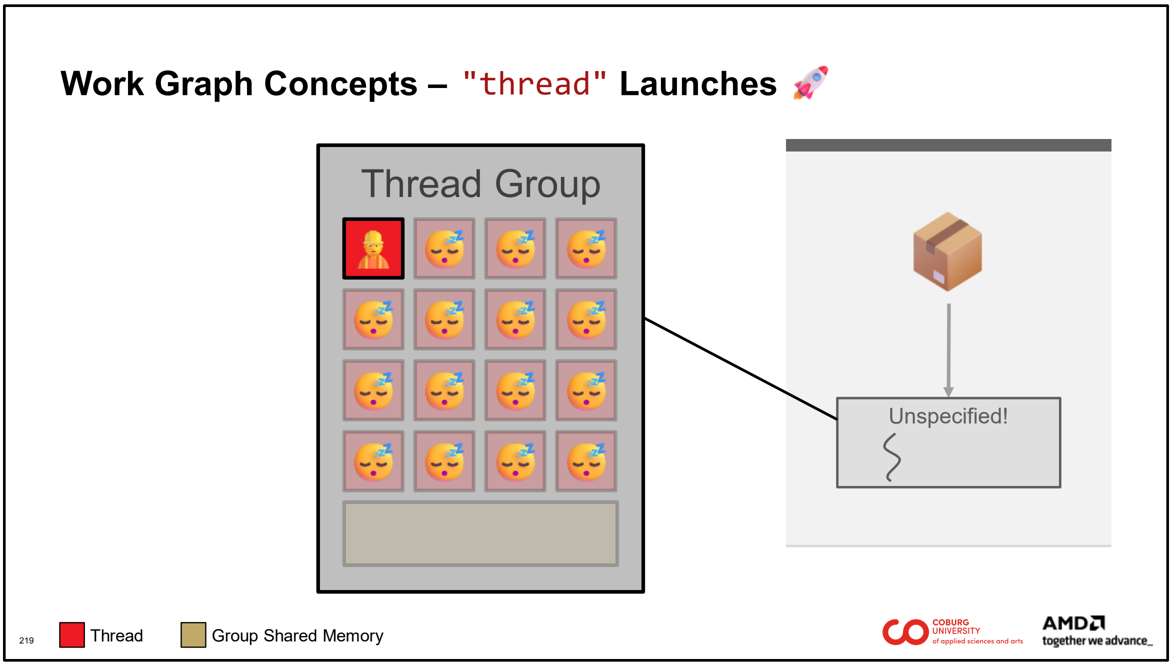
As we are now dealing with a single thread that accesses the incoming record, we use the type `TheadNodeInputRecord` to declare the input.



Even though an execution of "thread"-launch nodes is not defined by the specification, the underlying work-graphs system still uses thread groups to execute these nodes.

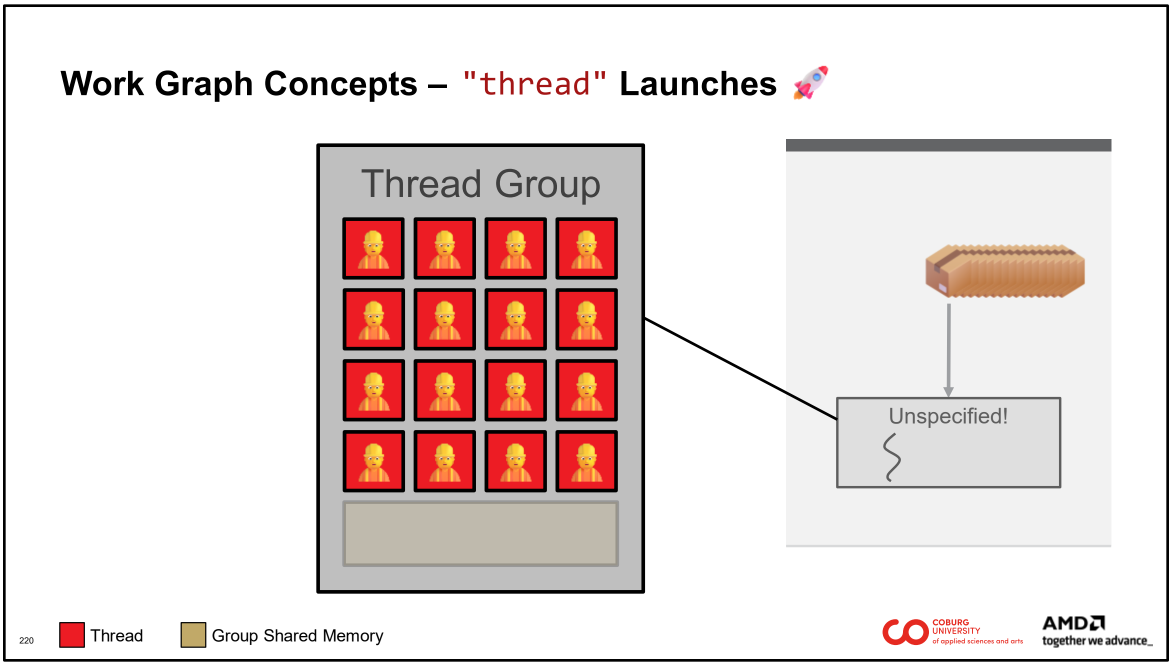


However, access to the group shared memory is not allowed, and…



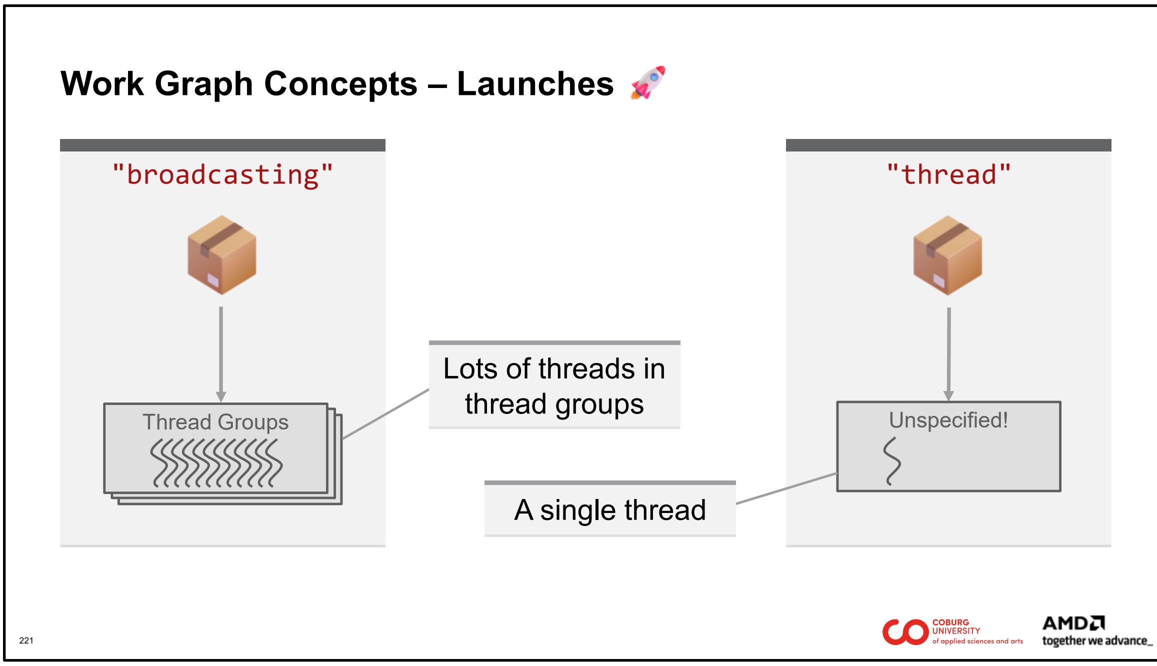
…as we only programmed a single thread, operations, such as wave intrinsics are also not allowed.

However, executing `"thread"`-launch nodes with one thread group per record is very wasteful of GPU resources.



Thus, the Work Graphs scheduler tries to combine multiple `ThreadNodeInputRecords` of the same node into thread groups, thereby increasing the efficiency of `"thread"`-launch nodes.

This is fully transparent to the programmer: we program as if there is just one single thread. With the exception that some work graph limits – like the maximum number of output records – are split up among the invisible group.

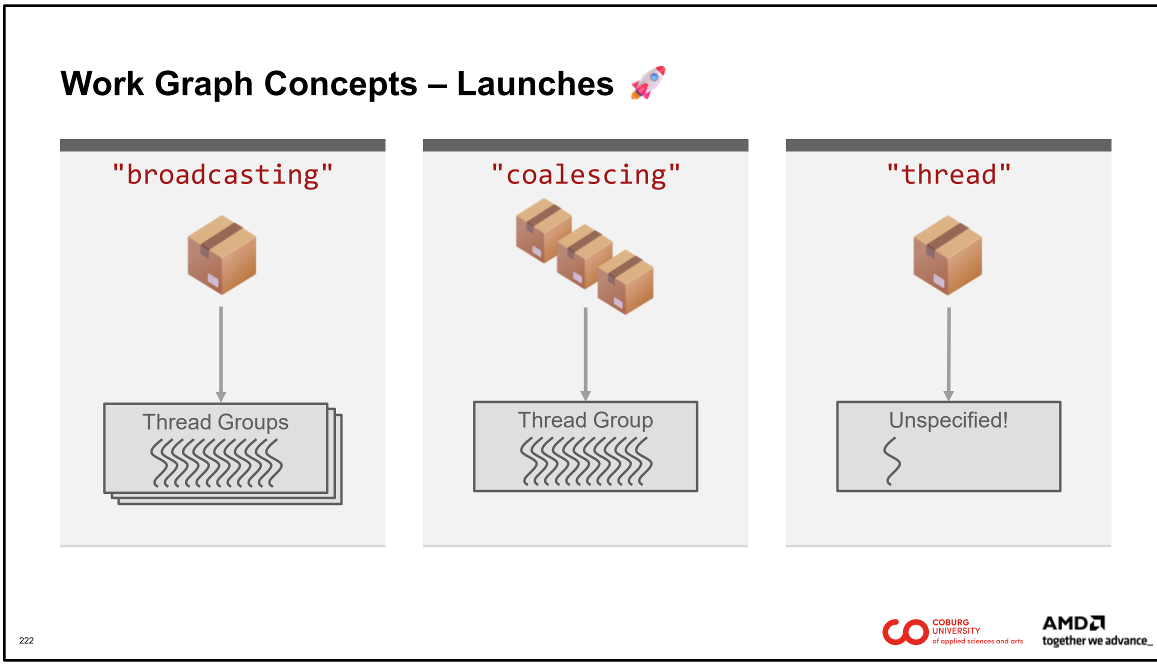


We looked at the two extremes:

- "broadcasting" node launch mode. They resemble compute shaders. There, we program an entire thread group.
- "thread" launch mode that is how we program vertex or pixel shaders. You as a programmer write your code from the perspective of a single thread.

In summary, the "thread" launch mode tries to cluster together records to the same node, but communication between the threads is forbidden. They cannot use shared memory.

What if we take this idea further and allow for communication?

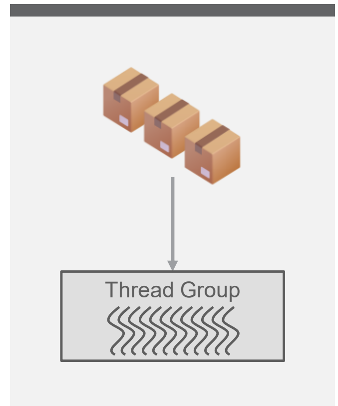


This is where our last launch mode comes in: The "coalescing" launch mode.

Work Graph Concepts – "coalescing" Launches 🚀

CoalescingExample.hlsl

```
[Shader("node")]
[NodeLaunch("coalescing")]
[NumThreads(4, 4, 1)]
void Node(
  [MaxRecords(4)]
  GroupNodeInputRecords<Job> input
){
  ...
}
```

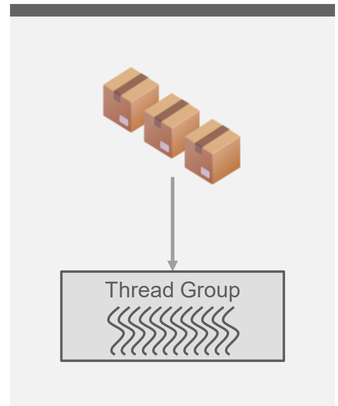


The easiest way to think of the "coalescing" launch mode is as a "thread" launch mode with more flexibility and control: We can specify how many records to the same node should be grouped together at maximum, and how many threads the group that is processing this collection should have.

Work Graph Concepts – "coalescing" Launches 🚀

CoalescingExample.hlsl

```
[Shader("node")]
[NodeLaunch("coalescing")]
[NumThreads(4, 4, 1)]
void Node(
  [MaxRecords(4)]
  GroupNodeInputRecords<Job> input
){
  ...
}
```



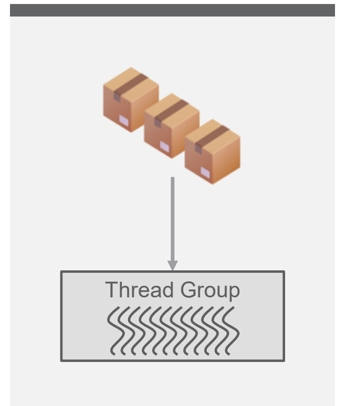
224

So here you see how you define a node in "coalescing" launch mode. We start – as before – by setting the [NodeLaunch(...)] attribute to "coalescing".

Work Graph Concepts – `"coalescing"` Launches 🚀

CoalescingExample.hlsl

```
[Shader("node")]
[NodeLaunch("coalescing")]
[NumThreads(4, 4, 1)]
void Node(
   [MaxRecords(4)]
   GroupNodeInputRecords<Job> input
){
   ...
}
```



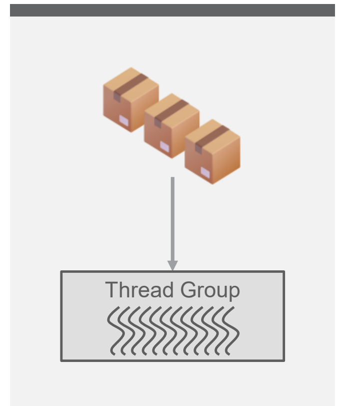
As we now have multiple records that are shared across a single thread group, we use `GroupNodeInputRecords` to declare the node input (Note the plural "s" at the end).

Additionally, we set an upper limit for how many records we want to consume with each thread group of our node. Please note, that this is only an upper limit.

Work Graph Concepts – "coalescing" Launches 🚀

CoalescingExample.hlsl

```
[Shader("node")]
[NodeLaunch("coalescing")]
[NumThreads(4, 4, 1)]
void Node(
  [MaxRecords(4)]
  GroupNodeInputRecords<Job> input
){
  uint recordCount = input.Count();
  ...
}
```



226

COBURG UNIVERSITY of applied sciences and arts

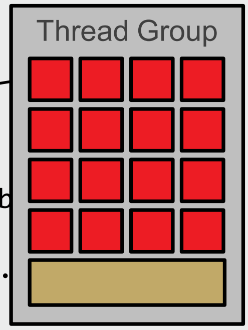
AMD together we advance_

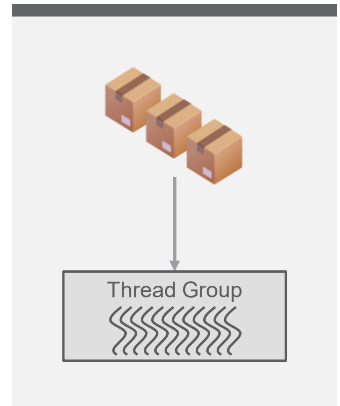
The actual number of available input records can be queried with the `Count()` function in the node input object.

Work Graph Concepts – "coalescing" Launches 🚀

CoalescingExample.hlsl

```
[Shader("node")]
[NodeLaunch("coalescing")]
[NumThreads(4, 4, 1)]
void Node(
  [MaxRecords(4)]
  GroupNodeInputRecords<Job
){
  uint recordCount = input.
  ...
}
```

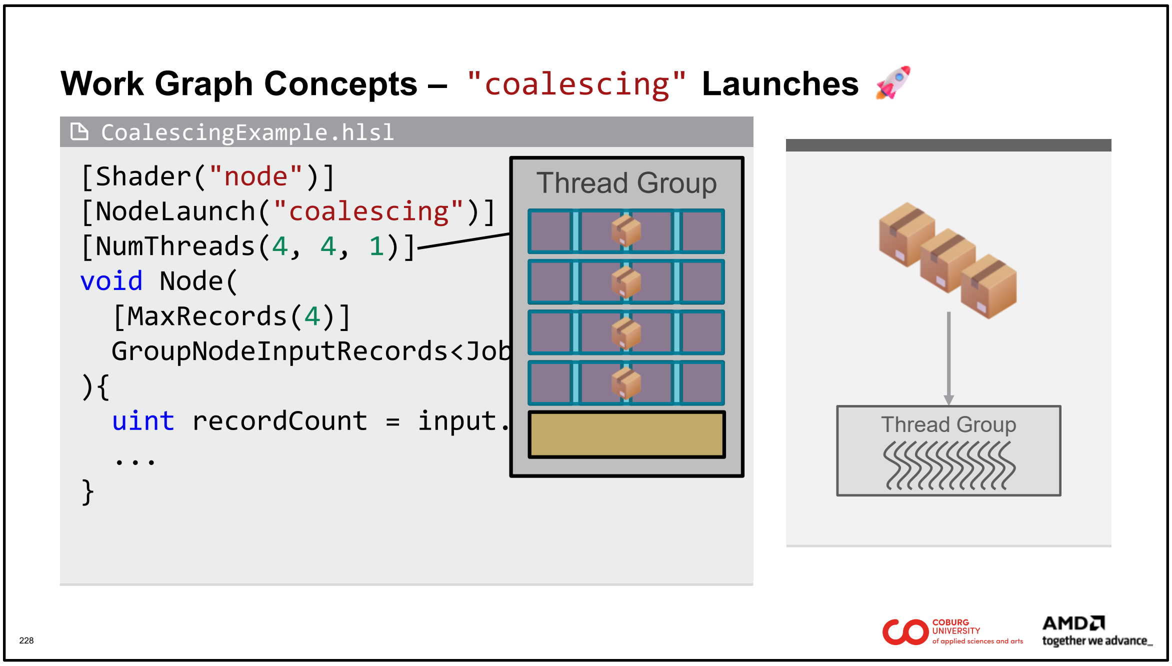




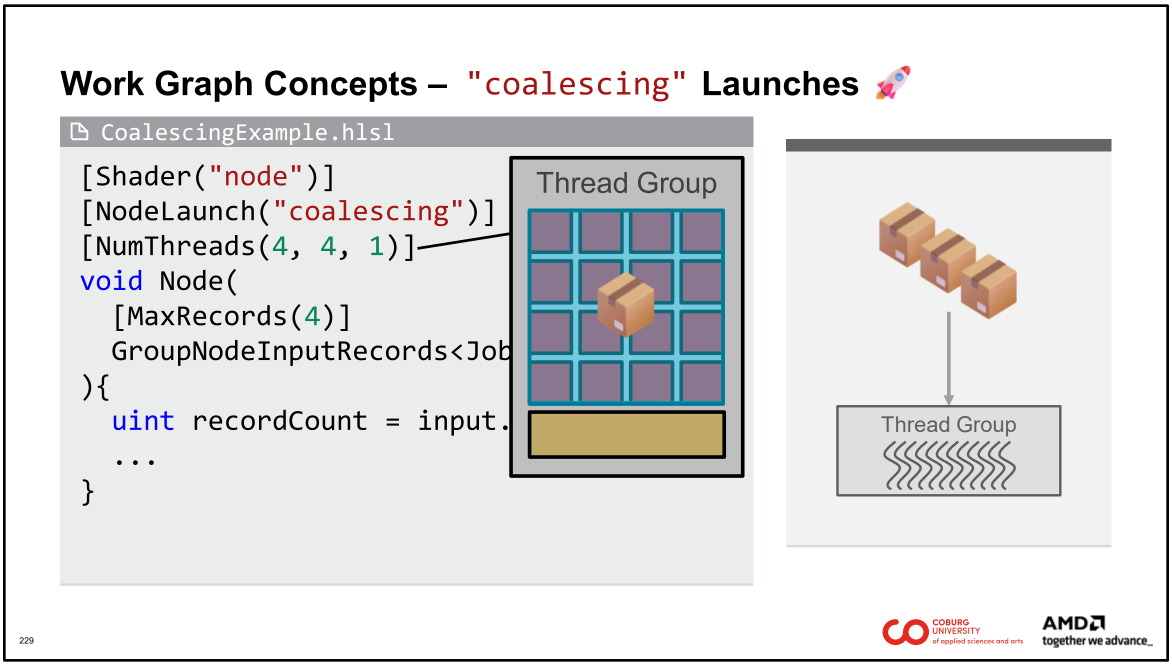
227

As we are programming a thread group, we have full control over how many threads we want per thread group and how these thread should be organized as a three-dimensional grid.

Here we have $4 \times 4 = 16$ threads. We then also have full control over how incoming records are mapped to these threads.



As we have at most four incoming records, we can assign a row of four threads to each of these records. Each of these threads can then process parts of the incoming record. For example, if incoming data are colors with four components (red, green, blue and alpha), each thread can process one color component.

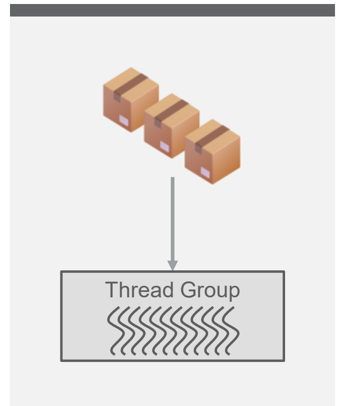


So far, we've seen how we process multiple records separately in parallel with `"coalescing"`-nodes. Additionally, as all threads of our thread group have access to all incoming records, we can also perform operations such as reductions across all incoming records.

Work Graph Concepts – "coalescing" Launches 🚀

tutorial-2/NodeLaunches.hlsl

```
[NodeLaunch("coalescing")]
[NumThreads(1, 1, 1)]
[NodeId("MergeRectangle")]
void MergeRectangleNode(
  [MaxRecords(2)]
  GroupNodeInputRecords<RectangleRecord>
    inputRecords,

  [MaxRecords(2)]
  [NodeId("FillRectangle")]
  NodeOutput<RectangleRecord> output) {
  ...
}
```

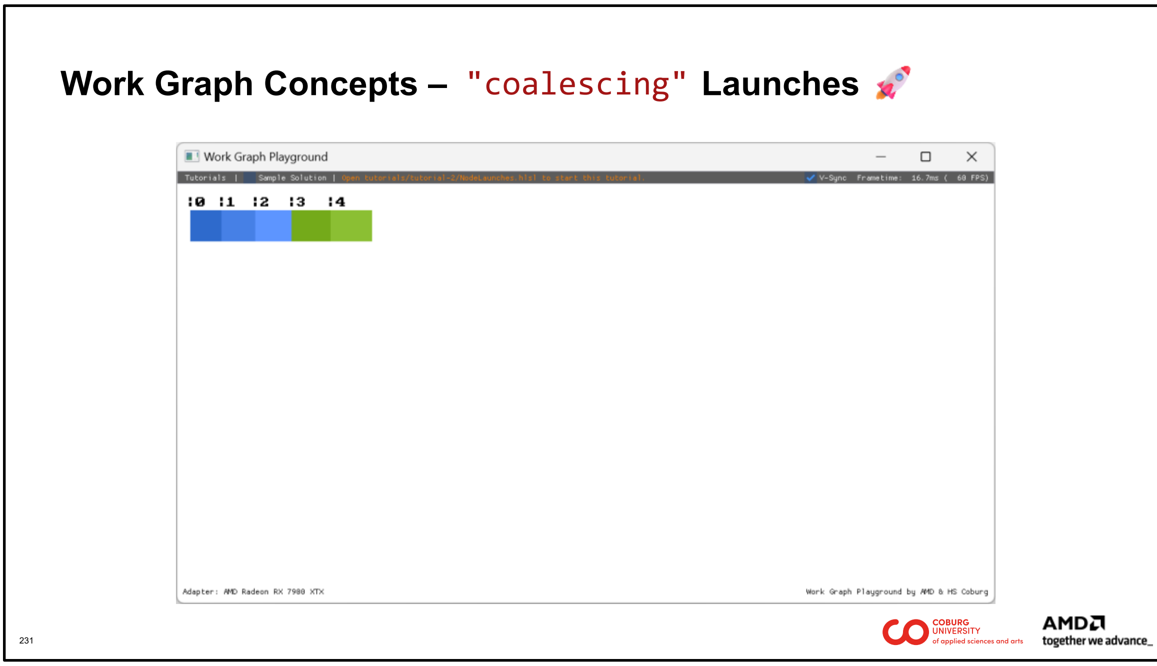


230

Implementing such a reduction is part of Task 2 and Task 3 in the Node Launches tutorial.

Start by opening `tutorials/tutorial-2/NodeLaunches.hlsl` and follow the instructions for `[Task 2]`.

As a first step, implement a `MergeRectangle` node as shown above. This node will take in up to two rectangles and pass them through to the `FillRectangle` node. Later, we will implement the reduction by merging rectangles into a single one if they share an edge.

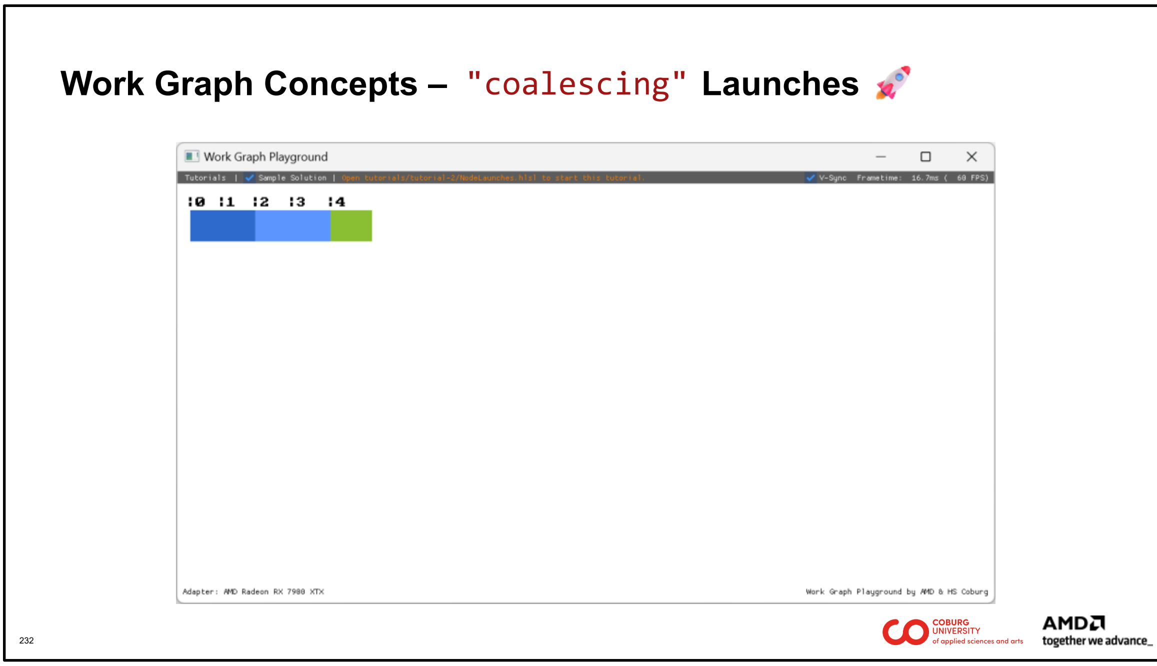


Once you are done with Task 2, the Work Graph Playground App should still look the same.

Continue with instructions for `[Task 3]` to implement the reduction.

Complete the sub-call to the `ComputeCombinedRect` helper method. If this helper returns "`true`", then you must emit a single record to the "`FillRectangle`" node.

Position and size of this rectangle are given by the "`ComputeCombinedRect`" helper. For the color of this rectangle, you can re-use the color from any of the input records (e.g., `record[0]`).



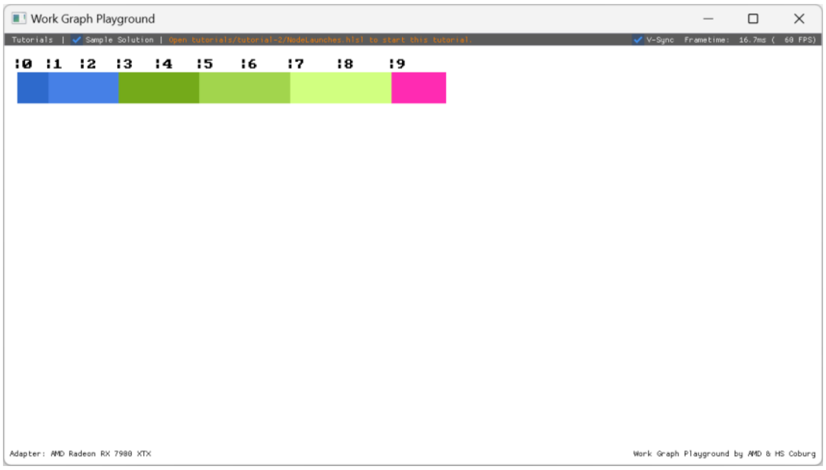
Once you're done, you should now see the same area being filled, but this time with just three instead of five rectangles. As five is not divisible by two, there's also one rectangle which could not be merged and is passed through as-is from the `MergeRectangle` node to the `FillRectangle` node.

Work Graph Concepts – `"broadcasting"` Launches 🚀

- ☑ 1. Change FillRectangle to dynamic dispatch grid
- ☑ 2. Implement pass-through coalescing node
- ☑ 3. Merge adjacent rectangles
- ☐ 4. Non-deterministic coalescers

With Task 2 and Task 3 completed, we can continue to Task 4 and the non-deterministic nature of `"coalescing"`-nodes.

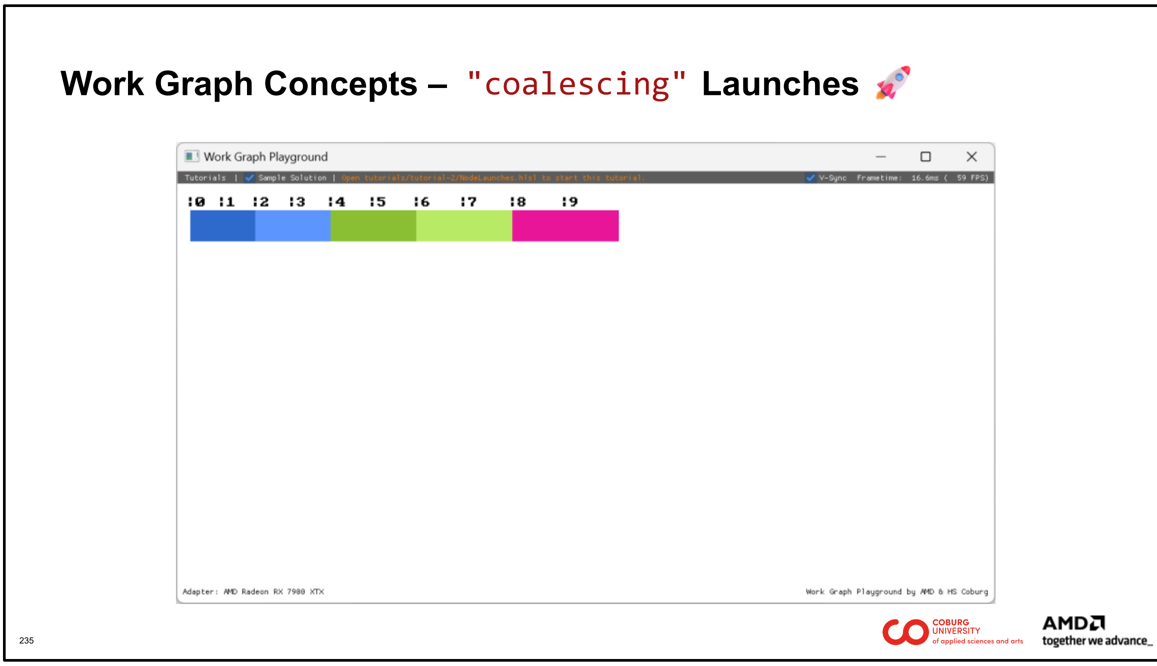
Work Graph Concepts – `"coalescing"` Launches 🚀



234

Increase the dispatch grid of the `Entry` node in `x` dimension to emit more rectangles.

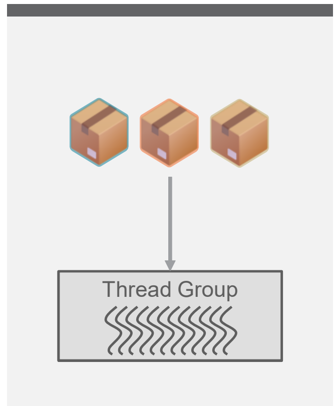
You should now see the merged rectangles flickering…



…between different ways of merging the rectangles. As the input to the coalescer node is non-deterministic and depends on the timing of the different thread groups of the "Entry" node. Thus, every frame different rectangles are merged.

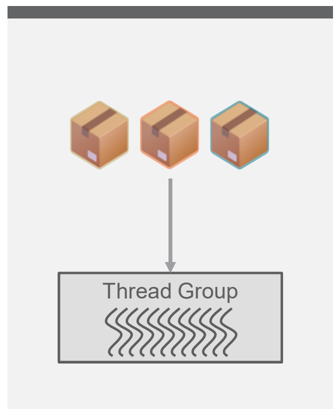
This step is omitted from the sample solution.

Work Graph Concepts – "coalescing" Launches 🚀



Additionally, the order in which the incoming records are passed to the node function is also not deterministic, …

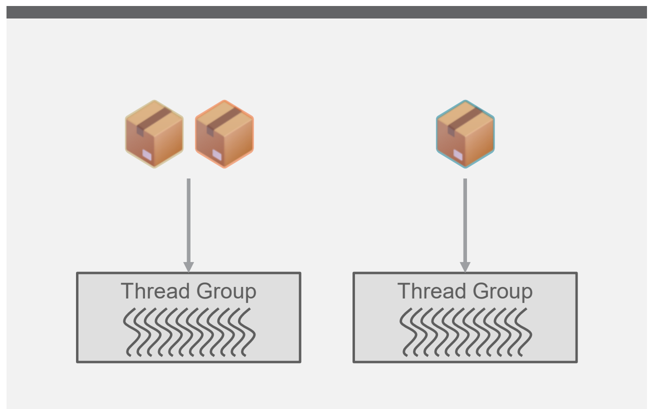
Work Graph Concepts – "coalescing" Launches 🚀



237

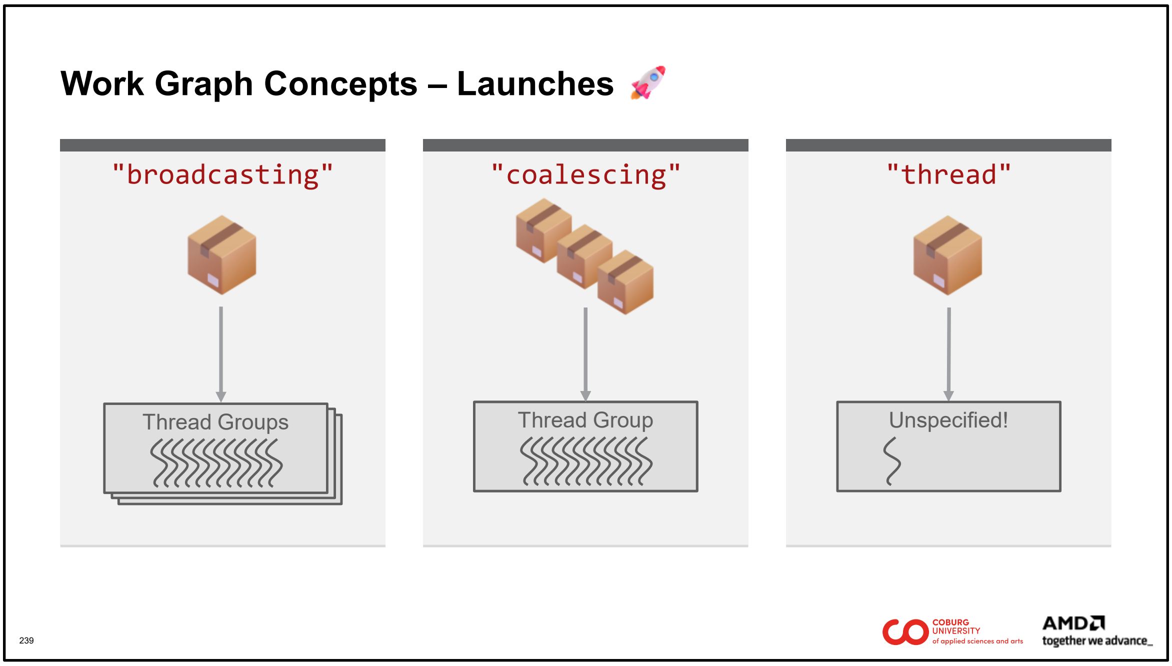
…can change with every execution of the work graph.

Work Graph Concepts – `"coalescing"` Launches 🚀



238

There is also no guarantee that a group always receives the specified number of records. However, all records sent to a `"coalescing"`-node will eventually be processed by it – even if this means invoking the node with just a single record.



In summary, we've now seen the three different launch modes available in Work Graphs: `"broadcasting"`, `"coalescing"` and `"thread"`. Together with nodes and records, these form the three core concepts of Work Graphs, and we are now ready to move on to more advanced uses of Work Graphs.

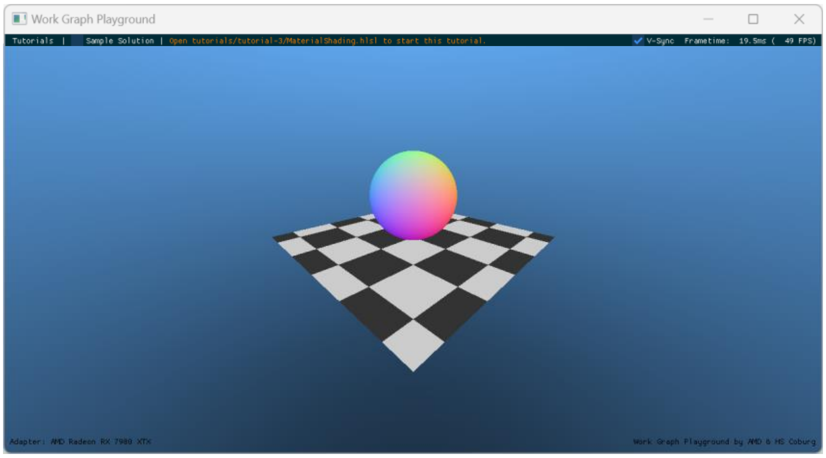
Advanced Work Graphs

Use-case: Material Shading

240

In this section, we put all concepts, i.e., nodes, records, and launches, together to create an advanced use-case for Work-Graphs. Plus, we are going to learn about a powerful Work Graphs feature called “Node Arrays”. We demonstrate this at the practical example of Material Shading.

Problem: Material Shading



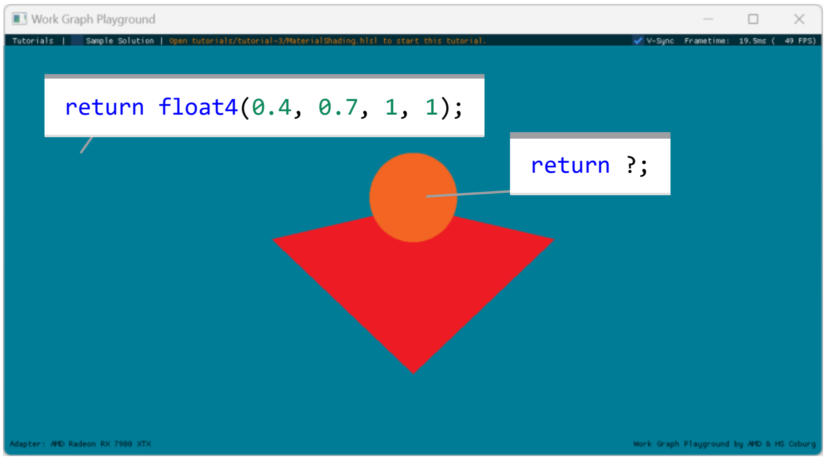
241

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

So, what is the problem of Material Shading? Consider this simple scene with a background, a plane, and a sphere.

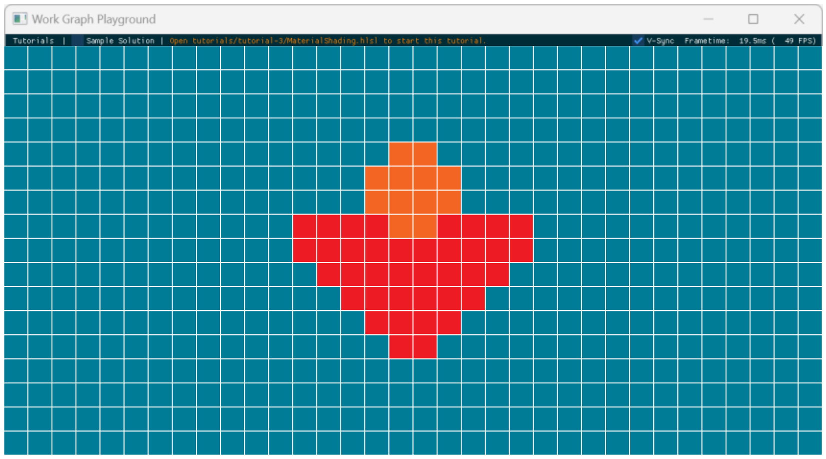
Problem: Material Shading



242

The objects and therefore the rendered pixels have a different material, highlighted with different colors here. For example, the sky could have a very simple material, such as a constant color. But computing the material for the sphere or the plane could be quite involved. They could, in fact, be different materials, requiring different algorithms with different costs.

Problem: Material Shading

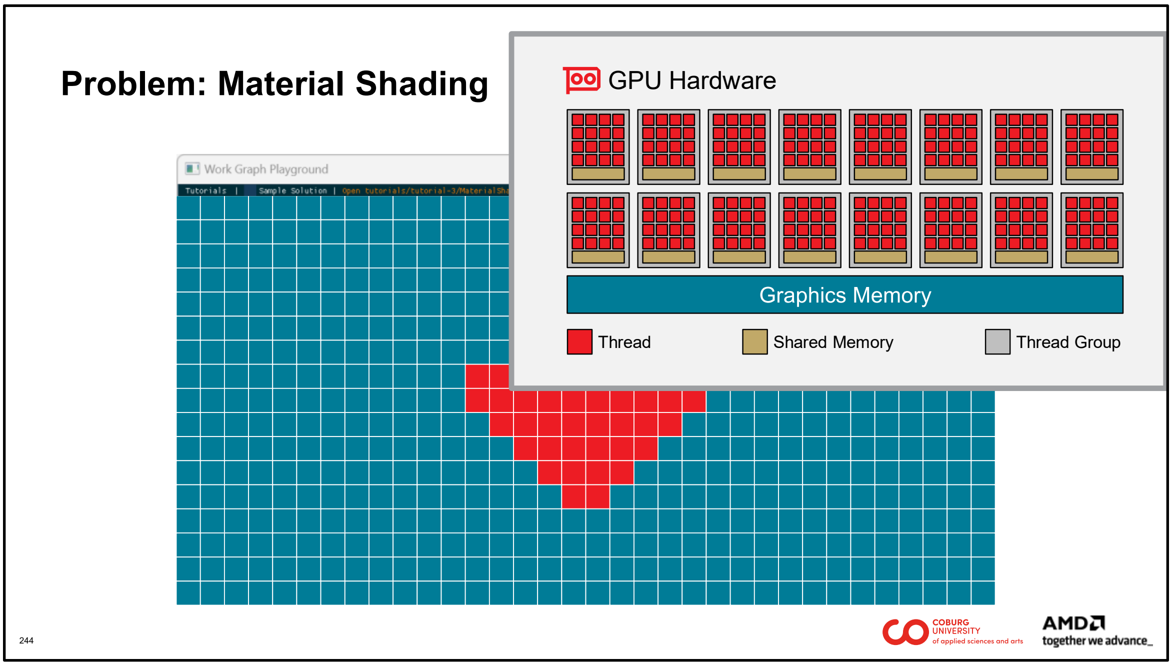


243

COBURG UNIVERSITY of applied sciences and arts

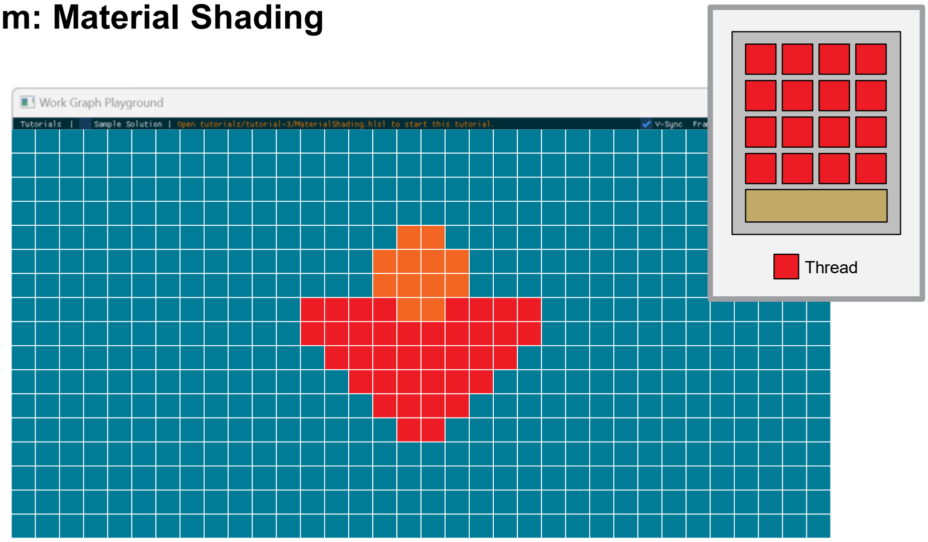
AMD together we advance_

What is the problem then? To explain that, let’s change to a coarser version of that image. You see the individual pixels of the image here using three distinct colors. Each color represents a different material type.



And let's not forget that we are running our computation on a GPU.

Problem: Material Shading

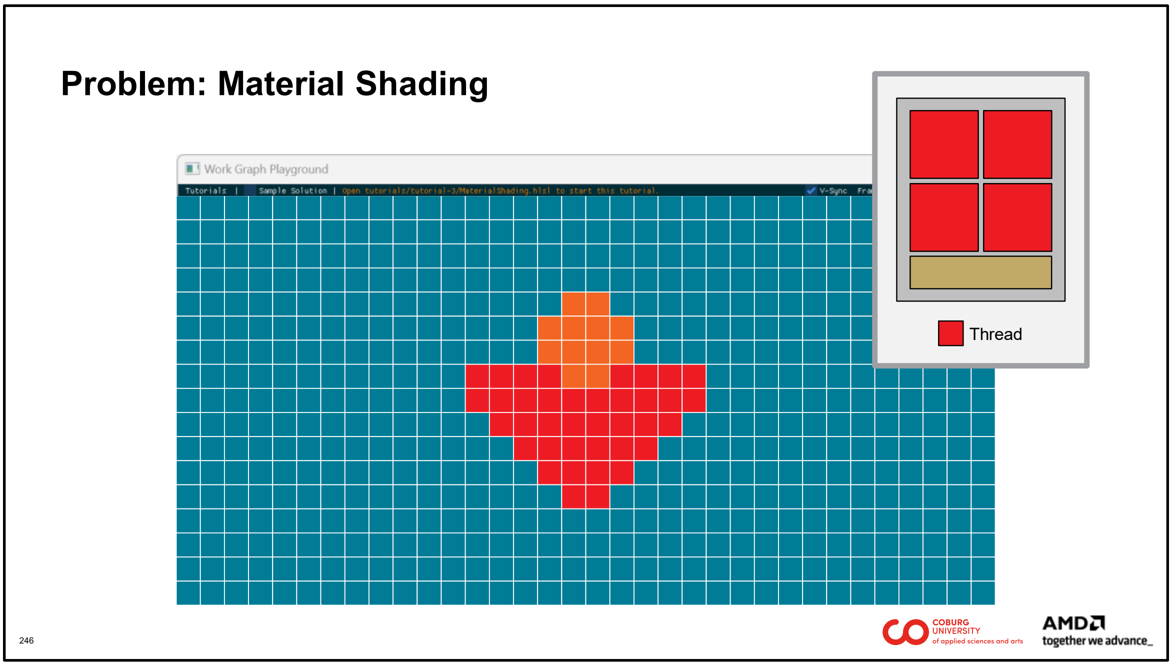


245

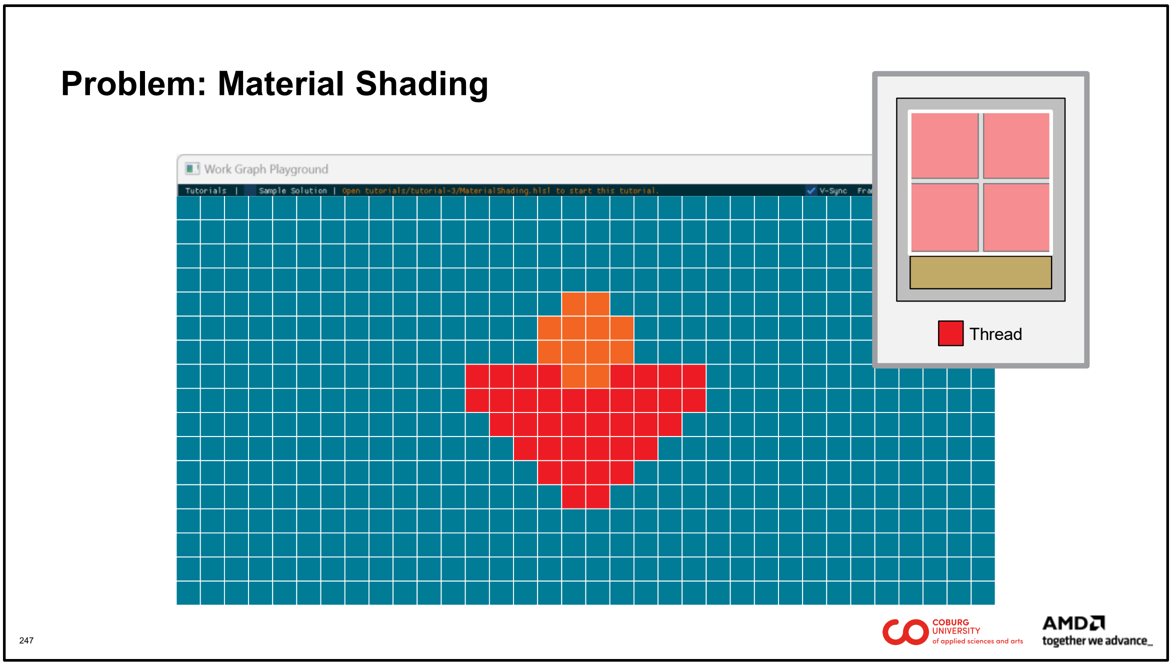
COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

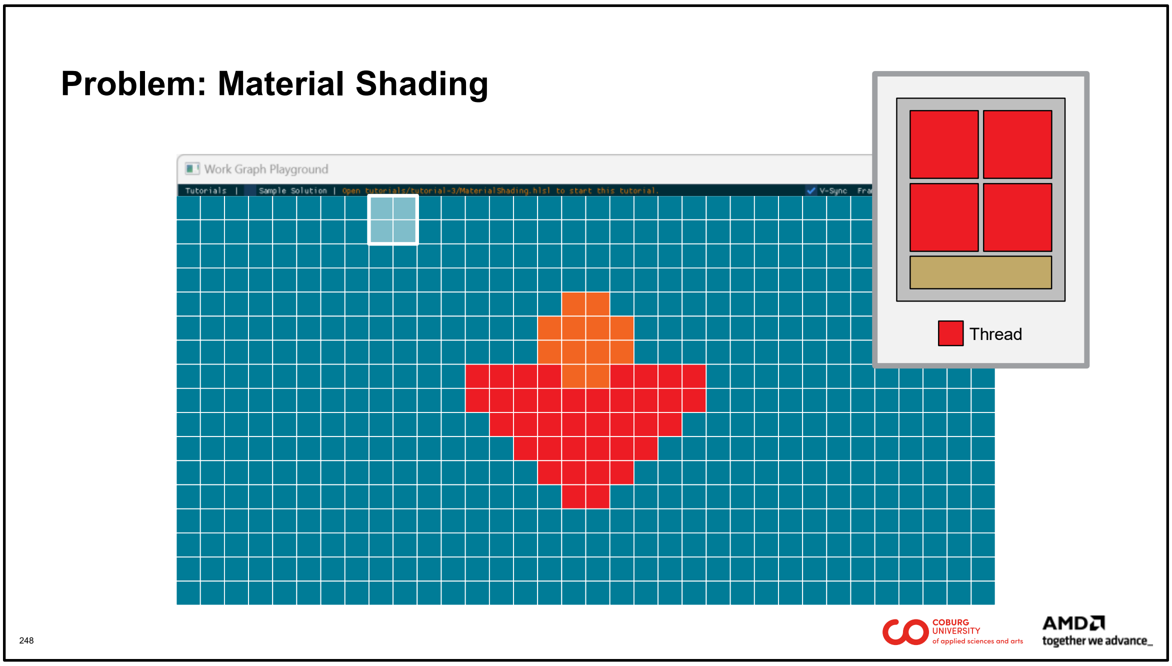
So, let’s see how a GPU thread group would compute that image.



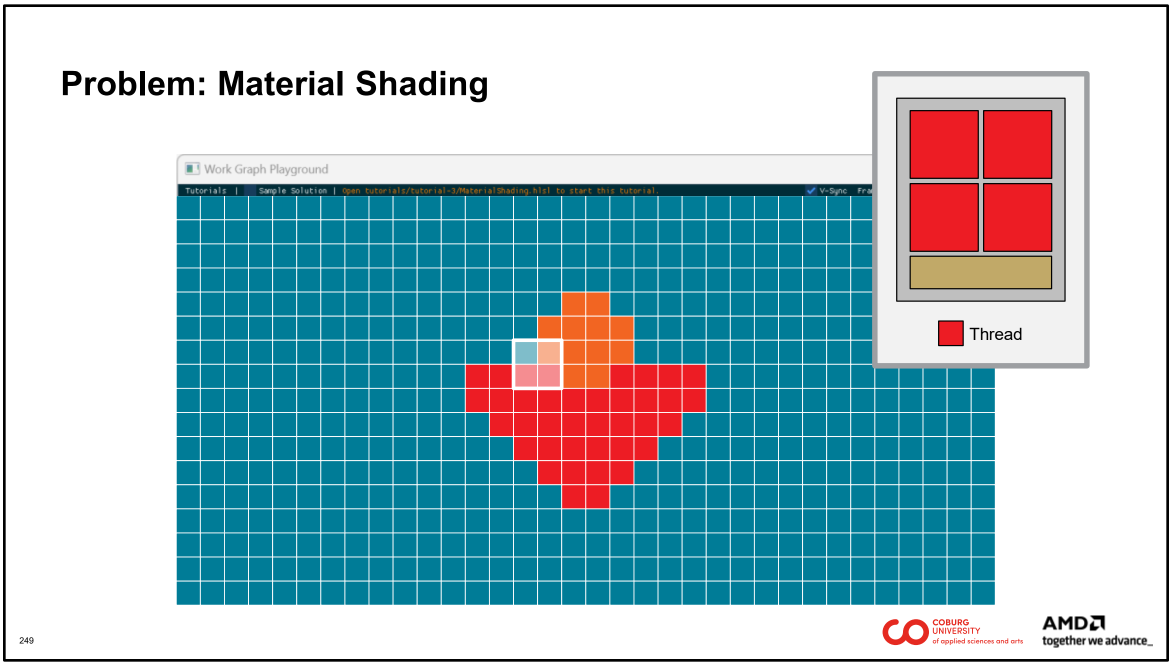
To make it a little more readable, our example thread group only has four threads.



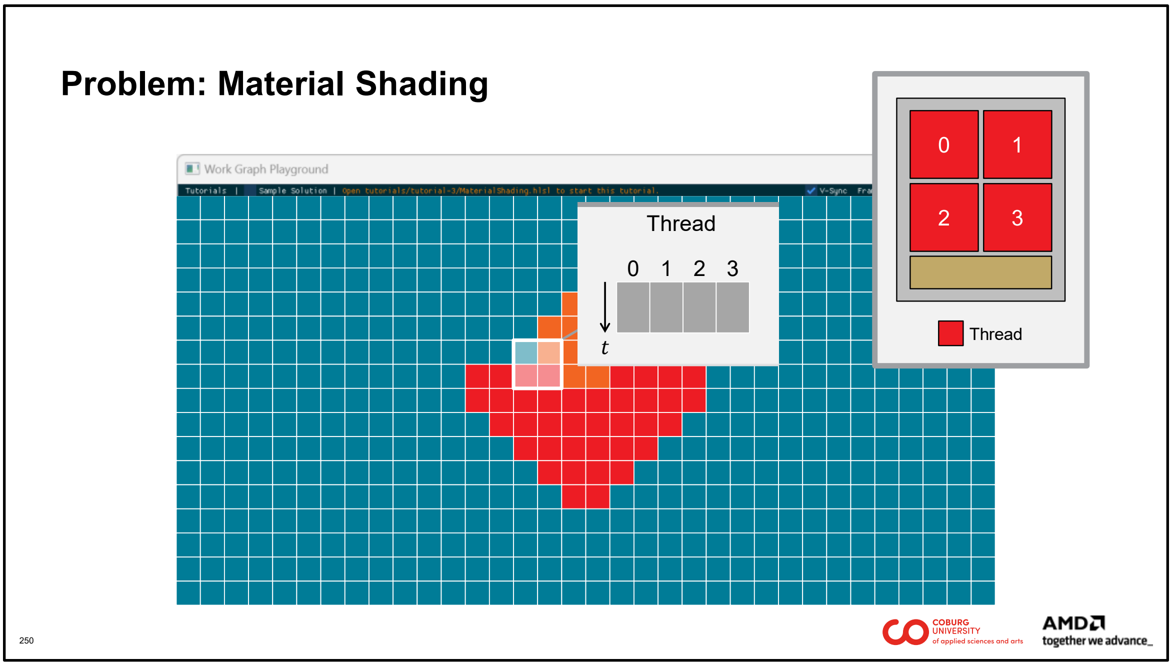
Our thread group can then compute a 2x2 grid of pixels in a SIMD fashion.



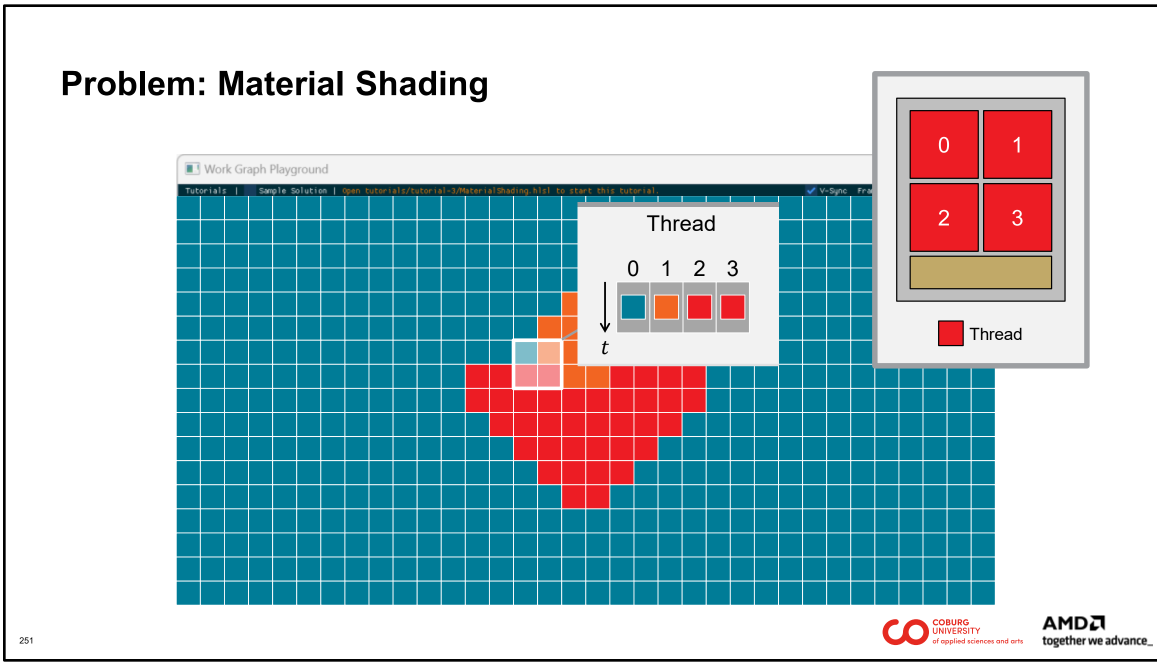
Each thread group computes a subset of those 2x2 blocks.



Things become interesting at the highlighted block here, where different materials need to be evaluated.

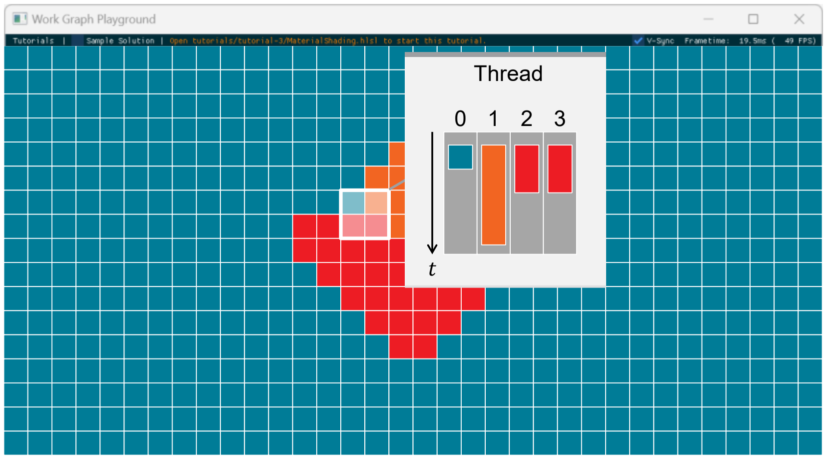


Each of the four pixels is evaluated with one thread in our thread group. Let’s consider how the computation is carried out over time.



The computation of each pixel is scheduled to one thread. Ideally, the four pixels can be executed in parallel and take equally long.

Problem: Material Shading



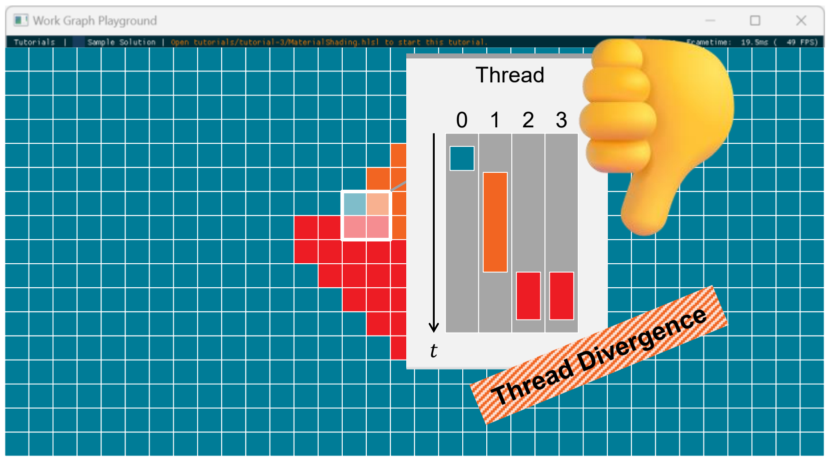
252

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

However, some materials are faster to compute, like the sky (blue), while others take a lot longer. In a thread group, the short code paths must wait for long ones to finish.

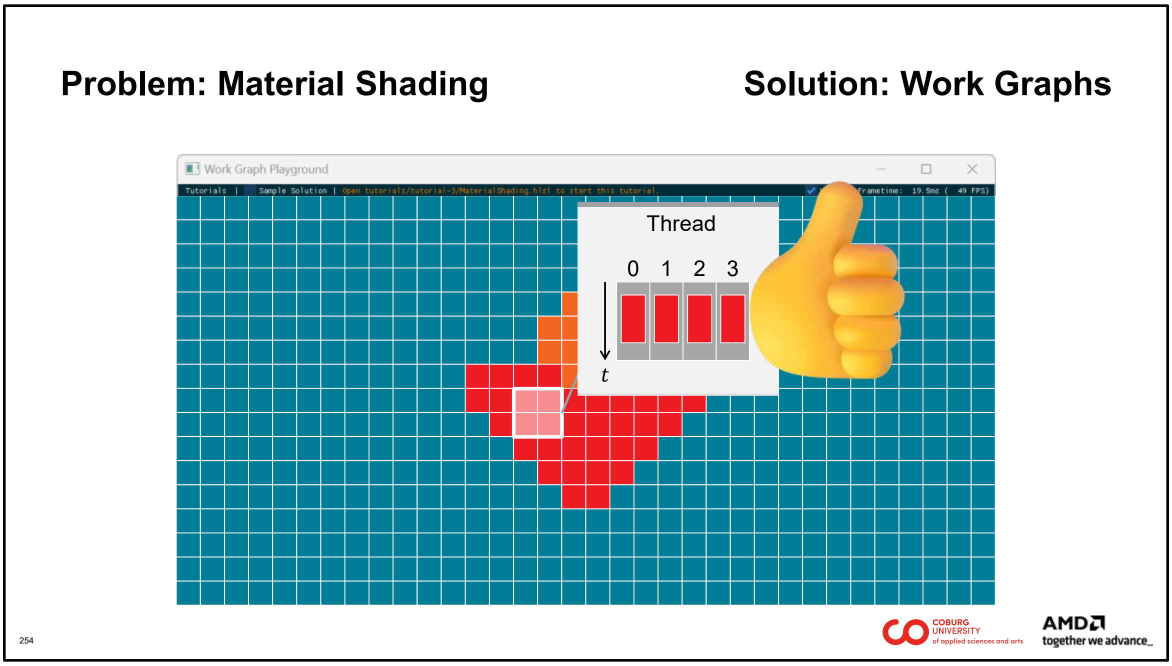
Problem: Material Shading



253

The threads of our thread group are executed on a SIMD core. That means the same instruction must be executed on all SIMD lanes at the same time.

Since the three different materials have different instructions, they cannot be executed in parallel. Instead, only those threads that share the same instructions can physically run in parallel. All other threads must defer their computation to a later point in time. This goes by the name “thread divergence” and can become a huge performance bottleneck on GPUs.



Where SIMD cores can deliver a huge performance boost is when the thread code is coherent, for example here in the group of red pixels.

In the following sections, we'll take a look at how Work Graphs can help us eliminate thread divergence by creating specialized nodes for each material.

Problem: Material Shading

tutorial-3/MaterialShading.hlsl

```
[NumThreads(8, 8, 1)]
void RenderScene(uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```

255

To start with, consider this compute shader example.

Problem: Material Shading

tutorial-3/MaterialShading.hlsl

```
[NumThreads(8, 8, 1)]
void RenderScene(uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```

256

It is called `RenderScene` and we get a unique global thread id `dtid` that gives a 2D integer pixel coordinate for the pixel that we wish to shade.

Problem: Material Shading

tutorial-3/MaterialShading.hlsl

```
[NumThreads(8, 8, 1)]
void RenderScene(uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```

257

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

Each thread group uses an 8x8 grid of threads, so that, each thread group computes 64 pixels.

Problem: Material Shading

tutorial-3/MaterialShading.hlsl

```
[NumThreads(8, 8, 1)]
void RenderScene(uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```

258

For each thread, we trace a ray, to find the closest hit…

Problem: Material Shading

tutorial-3/MaterialShading.hlsl

```
[NumThreads(8, 8, 1)]
void RenderScene(uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```

259

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

… and then carry out the shading, depending on the material that our ray hit.

Problem: Material Shading

tutorial-3/MaterialShading.hlsl

```
[NumThreads(8, 8, 1)]
void RenderScene(uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        case RayHit::Sky:
            color = ShadeSky(ray); break;
        case RayHit::Sphere:
            color = ShadeSphere(ray, hit.distance); break;
        case RayHit::Plane:
            color = ShadePlane(ray, hit.distance); break;
    }
}
```

260

Here, we have the switch statement, which is the root of the thread divergence problem. Depending on the material, we must take a different code path. If those code paths don’t share the same instruction, we effectively serialized the code.

Recap – Work Graph – Nodes

tutorial-3/MaterialShading.hlsl

```
[Shader("node")]
[NumThreads(8, 8, 1)]
void RenderScene(uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```

261

We want to solve this by using Work Graphs.

To turn a compute shader into a Work Graph node, we start by adding a `[Shader("node")]` attribute before the function definition. Nodes are basically compute-shaders with this node attribute.

Recap – Work Graph – Records

tutorial-3/MaterialShading.hlsl

```
struct Record {
    ...
};

[Shader("node")]
[NumThreads(8, 8, 1)]
void RenderScene(uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```

262

As this is no longer a compute shader, but a work graph nodes, we cannot dispatch it with e.g. the Dispatch command. Instead, we must send a record to our newly created node. Thus, we declare a Record struct above with all the data that we want to pass to our node, e.g., a camera view-projection matrix. The actual contents of the struct are omitted here for simplicity.

Recap – Work Graph – Records

tutorial-3/MaterialShading.hlsl

```
struct Record {
    ...
};

[Shader("node")]
[NumThreads(8, 8, 1)]
void RenderScene(        NodeInputRecord<Record> inputRecord,
                 uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```

263

To make our `RenderScene` node a consumer receiving such a record, we must declare a `NodeInputRecord` with our record as template argument.

Recap – Work Graph – Records

tutorial-3/MaterialShading.hlsl

```
struct Record {
    ...
};

[Shader("node")]
[NumThreads(8, 8, 1)]
void RenderScene(          NodeInputRecord<Record> inputRecord,
                  uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```

264

As we've seen before, the specific type of `NodeInputRecord` depends on the launch mode for the node, which we have not yet selected in our example.

Recap – Work Graph – Launches

tutorial-3/MaterialShading.hlsl

```
struct Record { ... };


[Shader("node")]
[NodeLaunch("broadcasting")]
[NumThreads(8, 8, 1)]
void RenderScene(        NodeInputRecord<Record> inputRecord,
                 uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```

265

The compute shader implementation that we started out with was dispatched with multiple thread groups in both x and y direction to cover all the pixels in our render target.

This behavior is mimicked by the `"broadcasting"` node launch, which dispatches a grid of thread groups for each incoming record.

Recap – Work Graph – Launches

tutorial-3/MaterialShading.hlsl

```
struct Record { ... };


[Shader("node")]
[NodeLaunch("broadcasting")]
[NumThreads(8, 8, 1)]
void RenderScene(DispatchNodeInputRecord<Record> inputRecord,
                 uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```

266

For `"broadcasting"` nodes, the input record must be declared as `DispatchNodeInputRecord`. All thread groups of the dispatch have a read-only view on the same `inputRecord`.

Recap – Work Graph – Launches

tutorial-3/MaterialShading.hlsl

```
struct Record { ... };

[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeDispatchGrid(480, 270, 1)]
[NumThreads(8, 8, 1)]
void RenderScene(DispatchNodeInputRecord<Record> inputRecord,
                 uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```

Dispatch(480, 270, 1)

267

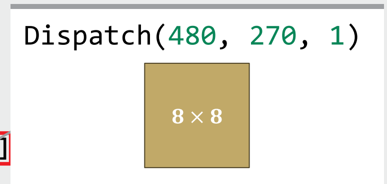
Next, we must specify the dispatch grid for our node, or in other words, how many thread groups we want to launch for each incoming record.

Here, we set it to launch a grid of 480x270x1 thread groups for every record.

Recap – Work Graph – Launches

tutorial-3/MaterialShading.hlsl

```
struct Record { ... };

[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeDispatchGrid(480, 270, 1)]
[NumThreads(8, 8, 1)]
void RenderScene(DispatchNodeInputRecord<Record> inputRecord,
                 uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```



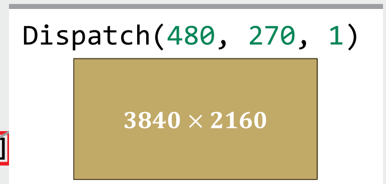
268

As each thread group has 8x8 threads…

Recap – Work Graph – Launches

tutorial-3/MaterialShading.hlsl

```
struct Record { ... };

[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeDispatchGrid(480, 270, 1)]
[NumThreads(8, 8, 1)]
void RenderScene(DispatchNodeInputRecord<Record> inputRecord,
                 uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```



269

… that makes grid of 3840 x 2160 threads in total. That is enough to cover a 4K Ultra HD (UHD) image with one thread per pixel. That is, however, now a fixed grid size. That means, we would always launch 3840 x 2160 threads. But what if we want to keep that size more flexible, for example, if we want to make our window smaller?

Hint: In case you wonder, 8 threads in x direction and 480 blocks in x direction makes $8 \times 480 = 3840$. Likewise, for the y direction we get $8 \times 270 = 2160$.

Recap – Work Graph – Launches

tutorial-3/MaterialShading.hlsl

```
struct Record { ... };

[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeMaxDispatchGrid(480, 270, 1)]
[NumThreads(8, 8, 1)]
void RenderScene(DispatchNodeInputRecord<Record> inputRecord,
                 uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```

270

To get that flexibility, we add a `Max` there. This specifies an upper bound for the number of thread groups.

Recap – Work Graph – Launches

tutorial-3/MaterialShading.hlsl

```
struct Record {
    uint3 dispatchGrid : SV_DispatchGrid;
    ...
};

[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeMaxDispatchGrid(480, 270, 1)]
[NumThreads(8, 8, 1)]
void RenderScene(DispatchNodeInputRecord<Record> inputRecord,
                 uint2 dtid : SV_DispatchThreadId) {
    ...
}
```

271

The producer of the `Record` struct is then tasked with setting the actual number of thread groups. This information is passed to the work graph runtime by annotating a variable with `SV_DispatchGrid`.

Recap – Work Graph – Launches

tutorial-3/MaterialShading.hlsl

```
struct Record {
    uint3 dispatchGrid : SV_DispatchGrid;
    ...
};

[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeMaxDispatchGrid(480, 270, 1)]
[NumThreads(8, 8, 1)]
void RenderScene(DispatchNodeInputRecord<Record> inputRecord,
                 uint2 dtid : SV_DispatchThreadId) {
    ...
}
```

272

Remember, the `Record` struct and thus by extension the variable with `SV_DispatchGrid` semantic are tied to our node through the `DispatchNodeInputRecord` declaration.

Material Shading

tutorial-3/MaterialShading.hlsl

```
struct Record { ... };

[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeMaxDispatchGrid(480, 270, 1)]
[NumThreads(8, 8, 1)]
void RenderScene(DispatchNodeInputRecord<Record> inputRecord,
                 uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        ...
    }
}
```

273

Thus far, we have turned our initial compute shader into a broadcasting node with a dynamic dispatch grid.

Our goal, however, was to solve the issue of thread divergence caused by the switch-case statement for executing the material shaders.

Material Shading

tutorial-3/MaterialShading.hlsl

```
[Shader("node")]
void RenderScene(DispatchNodeInputRecord<Record> inputRecord,
                 uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        case RayHit::Sky:
            color = ShadeSky(ray); break;
        case RayHit::Sphere:
            color = ShadeSphere(ray, hit.distance); break;
        case RayHit::Plane:
            color = ShadePlane(ray, hit.distance); break;
    }
}
```


To reiterate, these shading functions use different instructions and thus cannot run in parallel on the SIMD-architecture of our GPU.

Material Shading

tutorial-3/MaterialShading.hlsl

```
[Shader("node")]
void RenderScene(DispatchNodeInputRecord<Record> inputRecord,
                 uint2 dtid : SV_DispatchThreadId) {
    const RayHit hit = TraceRay(...);
    ...
    switch (hit.material) {
        case RayHit::Sky:
            color = ShadeSky(ray); break;
        case RayHit::Sphere:
            color = ShadeSphere(ray, hit.distance); break;
        case RayHit::Plane:
            color = ShadePlane(ray, hit.distance); break;
    }
}
```

275

The underlying idea is to move these different shading functions into separate nodes and use work graphs to send records to these nodes based on the ray tracing result.

We start by moving the `ShadeSky` function…

Material Shading

tutorial-3/MaterialShading.hlsl

```
struct PixelRecord {
    uint2 pixel;
    Ray   ray;
    float hitDistance;
};

[Shader("node")]
[NodeLaunch("thread")]
void ShadePixel_Sky(ThreadNodeInputRecord<PixelRecord> inputRecord) {
    const PixelRecord record = inputRecord.Get();

    const float4 color = ShadeSky(record.ray);
    WritePixel(record.pixel, color);
}
```


…to a new node named `ShadePixel_Sky`.

Material Shading

tutorial-3/MaterialShading.hlsl

```
struct PixelRecord {
    uint2 pixel;
    Ray   ray;
    float hitDistance;
};

[Shader("node")]
[NodeLaunch("thread")]
void ShadePixel_Sky(ThreadNodeInputRecord<PixelRecord> inputRecord) {
    const PixelRecord record = inputRecord.Get();

    const float4 color = ShadeSky(record.ray);
    WritePixel(record.pixel, color);
}
```

277

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

As this node only processes a single pixel, we can use the `"thread"` launch mode, which assigns a single thread to each incoming record (i.e., each incoming pixel).

Material Shading

tutorial-3/MaterialShading.hlsl

```
struct PixelRecord {
    uint2 pixel;
    Ray   ray;
    float hitDistance;
};

[Shader("node")]
[NodeLaunch("thread")]
void ShadePixel_Sky(ThreadNodeInputRecord<PixelRecord> inputRecord) {
    const PixelRecord record = inputRecord.Get();

    const float4 color = ShadeSky(record.ray);
    WritePixel(record.pixel, color);
}
```

278

As we're using the `"thread"` launch mode, we must declare the node input with `ThreadNodeInputRecord`.

Material Shading

tutorial-3/MaterialShading.hlsl

```
struct PixelRecord {
    uint2 pixel;
    Ray   ray;
    float hitDistance;
};

[Shader("node")]
[NodeLaunch("thread")]
void ShadePixel_Sky(ThreadNodeInputRecord<PixelRecord> inputRecord) {
    const PixelRecord record = inputRecord.Get();

    const float4 color = ShadeSky(record.ray);
    WritePixel(record.pixel, color);
}
```

279

The record data itself is defined in the `PixelRecord` struct above. Here we pass the coordinate of the pixel we wish to shade, the ray that was traced for this pixel along with the ray length.

Material Shading

tutorial-3/MaterialShading.hlsl

```
struct PixelRecord {
    uint2 pixel;
    Ray   ray;
    float hitDistance;
};

[Shader("node")]
[NodeLaunch("thread")]
void ShadePixel_Sky(ThreadNodeInputRecord<PixelRecord> inputRecord) {
    const PixelRecord record = inputRecord.Get();

    const float4 color = ShadeSky(record.ray);
    WritePixel(record.pixel, color);
}
```

280

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

For convenience, we store the incoming record to a local variable called `record`.

Material Shading

tutorial-3/MaterialShading.hlsl

```
struct PixelRecord {
    uint2 pixel;
    Ray   ray;
    float hitDistance;
};

[Shader("node")]
[NodeLaunch("thread")]
void ShadePixel_Sky(ThreadNodeInputRecord<PixelRecord> inputRecord) {
    const PixelRecord record = inputRecord.Get();

    const float4 color = ShadeSky(record.ray);
    WritePixel(record.pixel, color);
}
```

281

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

We can then call the underlying `ShadeSky` function with the data from the record to compute the shaded color and write it to our pixel with the help of the `WritePixel` function.

Material Shading

tutorial-3/MaterialShading.hlsl

```
[Shader("node")]
[NodeLaunch("thread")]
void ShadePixel_Sky(ThreadNodeInputRecord<PixelRecord> inputRecord) {}

[Shader("node")]
[NodeLaunch("thread")]
void ShadePixel_Sphere(ThreadNodeInputRecord<PixelRecord> inputRecord) {}

[Shader("node")]
[NodeLaunch("thread")]
void ShadePixel_Plane(ThreadNodeInputRecord<PixelRecord> inputRecord) {}
```

282

We repeat the same steps for the Sphere and Plane material as well, thus creating a `ShadePixel_Sphere` and `ShadePixel_Plane` node. We can use the same `PixelRecord` struct that we declared earlier for these new nodes as well.

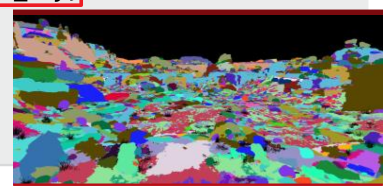
Material Shading

tutorial-3/MaterialShading.hlsl

```
[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeMaxDispatchGrid(480, 270, 1)]
[NumThreads(8, 8, 1)]
void Rende            eInputRecord<Record> i
                   [MaxRecords(8 * 8)]
                   NodeOutput<PixelRecord> ShadePixel_Sky,

                   uint2 dtid : SV_DispatchThreadId)
    ...
}
```

Max. 256 Records

```
[Shader("node")]
void ShadePixel_Sky(...)
```



283

To send records to our newly declared nodes, we must declare a `NodeOutput` in our `RenderScene` node for each material. We show this at the example of the `NodeOuput` for the `ShadePixel_Sky` node.

As all 8x8 pixel in our thread group might have the same material, we must declare all these node outputs with this worst case, i.e. 8 * 8 records.

However, this would mean that we would reach the output limit of 256 records with just four materials (8 * 8 * 4 = 256). Contrast this with the hundreds of materials used by modern AAA games and we can immediately see that this approach of declaring separate node outputs does not scale very well.

We can solve this problem by using a work graph feature specially designed for such use-cases called *node arrays*.

Node Arrays

tutorial-3/MaterialShading.hlsl

```
[Shader("node")]
[NodeLaunch("thread")]
void ShadePixel_Sky(ThreadNodeInputRecord<PixelRecord> inputRecord) {}

[Shader("node")]
[NodeLaunch("thread")]
void ShadePixel_Sphere(ThreadNodeInputRecord<PixelRecord> inputRecord) {}

[Shader("node")]
[NodeLaunch("thread")]
void ShadePixel_Plane(ThreadNodeInputRecord<PixelRecord> inputRecord) {}
```


Consider our different material nodes from before. They all use the same launch mode and input record…

Node Arrays

tutorial-3/MaterialShading.hlsl

```
[Shader("node")]
[NodeId("ShadePixel", 0]
[NodeLaunch("thread")]
void ShadePixel_Sky(ThreadNodeInputRecord<PixelRecord> inputRecord) {}

[Shader("node")]
[NodeId("ShadePixel", 1]
[NodeLaunch("thread")]
void ShadePixel_Sphere(ThreadNodeInputRecord<PixelRecord> inputRecord) {}

[Shader("node")]
[NodeId("ShadePixel", 2]
[NodeLaunch("thread")]
void ShadePixel_Plane(ThreadNodeInputRecord<PixelRecord> inputRecord) {}
```

Node Array Index

285

…thus we can combine them into a single node array named `ShadePixel`. To do this, we add a `[NodeId("ShadePixel", 0]` attribute to each node. The first part (i.e. the node id *name*) is the same for all nodes, but we must assign a different *node array index* to each node.

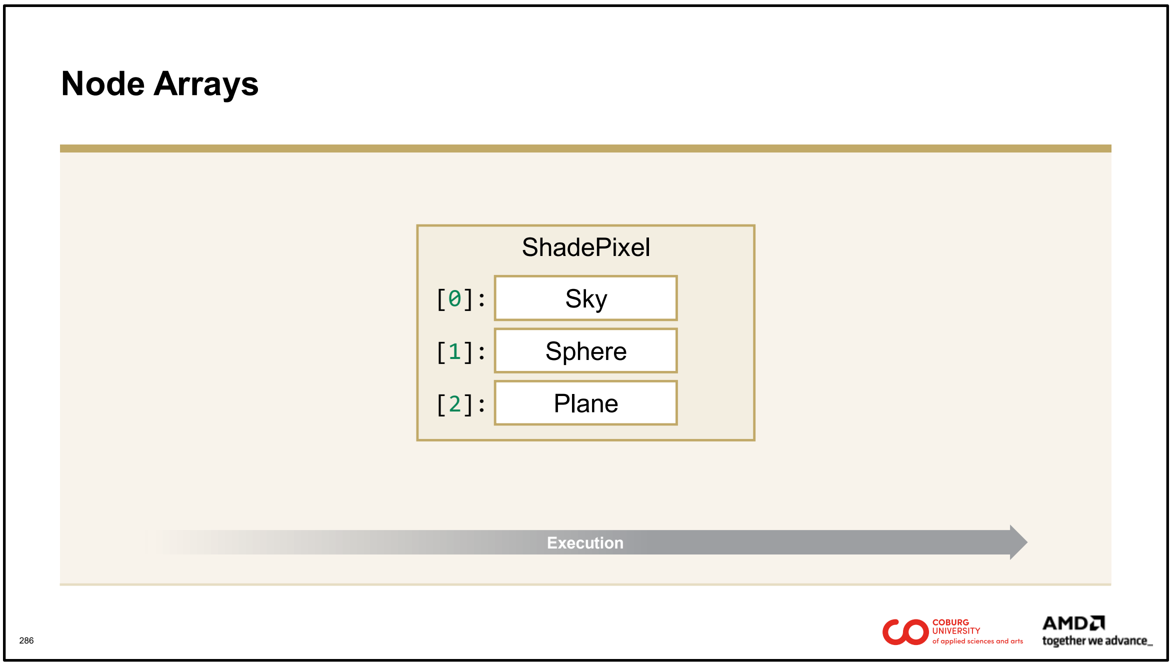
In our example, use the following mapping:

0 – sky material

1 – sphere material

2 – plane material

This mapping aligns with the `RayHit` enum values that we were using for the switch-case statement before.



In our Work Graph, we can then address these nodes as a node array named `ShadePixel`.

Material Shading

tutorial-3/MaterialShading.hlsl

```
[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeMaxDispatchGrid(480, 270, 1)]
[NumThreads(8, 8, 1)]
void RenderScene(DispatchNodeInputRecord<Record> input,

                 [MaxRecords(8 * 8)]
                 NodeOutputArray<PixelRecord> ShadePixel,

                 uint2 dtid : SV_DispatchThreadId) {
    ...
}
```

287

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

We can then target this node array by declaring a `NodeOutputArray`. Note that we do not target any individual node, but rather the whole array at once.

Material Shading

tutorial-3/MaterialShading.hlsl

```
[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeMaxDispatchGrid(480, 270, 1)]
[NumThreads(8, 8, 1)]
void RenderScene(DispatchNodeInputRecord<Record> input,

                 [MaxRecords(8 * 8)]
                 [NodeArraySize(3)]
                 NodeOutputArray<PixelRecord> ShadePixel,

                 uint2 dtid : SV_DispatchThreadId) {
    ...
}
```

288

However, the D3D12 runtime must still be able to validate that all the nodes we expect in this node array are present in the graph. Thus, we must add a `[NodeArraySize(...)]` attribute with the expected number of nodes in the array, which in our case is three.

Material Shading

tutorial-3/MaterialShading.hlsl

```
...
void RenderScene(DispatchNodeInputRecord<Record> input,
                 [MaxRecords(8 * 8)]
                 [NodeArraySize(3)]
                 NodeOutputArray<PixelRecord> ShadePixel,
                 uint2 dtid : SV_DispatchThreadId) {
    ...
    ThreadNodeOutputRecords<PixelRecord> outputRecord =
        ShadePixel[hit.material].GetThreadNodeOutputRecords(1);

    outputRecord.Get().pixel       = dtid;
    outputRecord.Get().ray         = ray;
    outputRecord.Get().hitDistance = hit.distance;

    outputRecord.OutputComplete();
}
```

289

Allocating records to be sent to this node array is very similar to the plain node outputs that we've seen before…

Material Shading

tutorial-3/MaterialShading.hlsl

```
...
void RenderScene(DispatchNodeInputRecord<Record> input,
                 [MaxRecords(8 * 8)]
                 [NodeArraySize(3)]
                 NodeOutputArray<PixelRecord> ShadePixel,
                 uint2 dtid : SV_DispatchThreadId) {
    ...
    ThreadNodeOutputRecords<PixelRecord> outputRecord =
        ShadePixel[hit.material].GetThreadNodeOutputRecords(1);

    outputRecord.Get().pixel
    outputRecord.Get().ray
    outputRecord.Get().hitDistance = hit.distance;

    outputRecord.OutputComplete();
}
```

Node Array Index

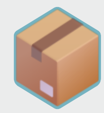
290

…the main difference is the bracket-operator, with which we specify the node array index, to which we want to send the record.

In our case, this index is determined by the ray tracing result.

Material Shading

```
tutorial-3/MaterialShading.hlsl
...
void RenderScene(DispatchNodeInputRecord<Record> input,
                 [MaxRecords(8 * 8)]
                 [NodeArraySize(3)]
                 NodeOutputArray<PixelRecord> ShadePixel,
                 uint2 dtid : SV_DispatchThreadId) {
    ...
    ThreadNodeOutputRecords<PixelRecord> outputRecord =
        ShadePixel[hit.material].GetThreadNodeOutputRecords(1);

    outputRecord.Get().pixel       = dtid;
    outputRecord.Get().ray         = ray;
    outputRecord.Get().hitDistance = hit.distance;

    outputRecord.OutputComplete();
}
```



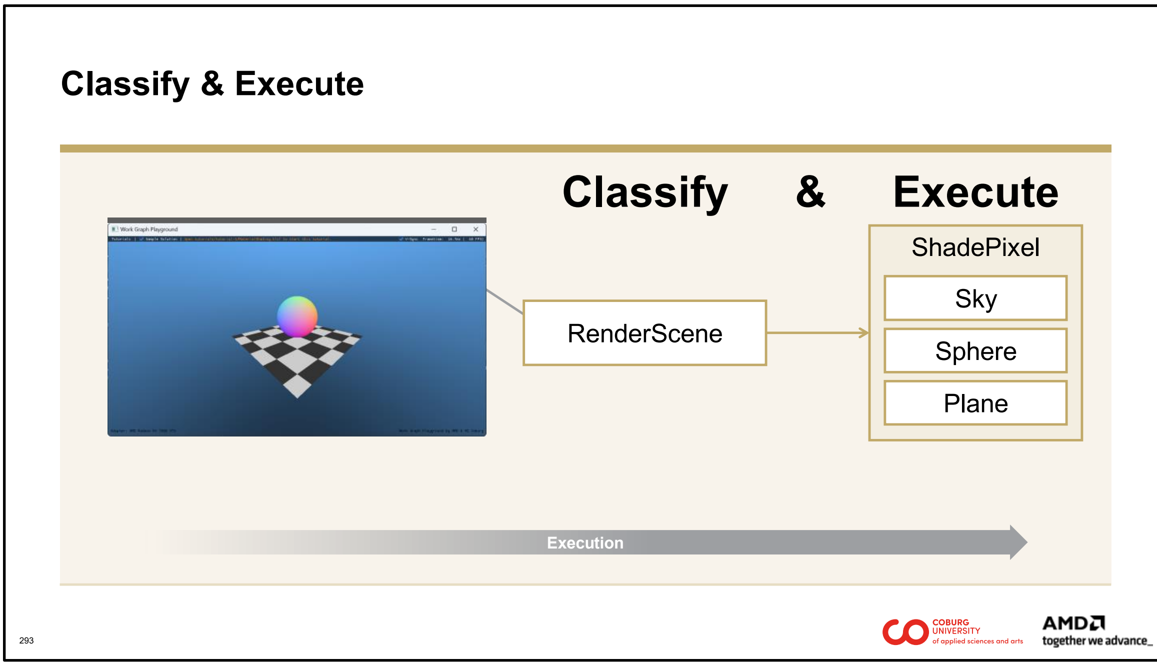
291

Writing data to the record is unchanged…

Material Shading

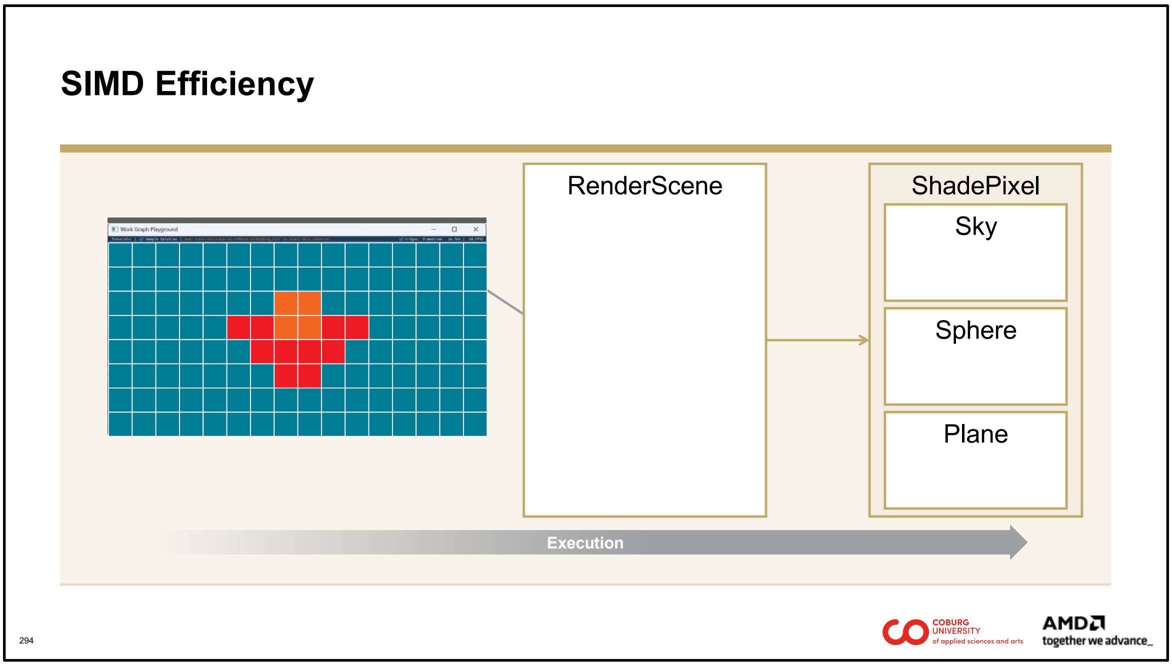
```
🗋 tutorial-3/MaterialShading.hlsl
...
void RenderScene(DispatchNodeInputRecord<Record> input,
                 [MaxRecords(8 * 8)]
                 [NodeArraySize(3)]
                 NodeOutputArray<PixelRecord> ShadePixel,
                 uint2 dtid : SV_DispatchThreadId) {
    ...
    ThreadNodeOutputRecords<PixelRecord> outputRecord =
        ShadePixel[hit.material].GetThreadNodeOutputRecords(1);

    outputRecord.Get().pixel       = dtid;
    outputRecord.Get().ray         = ray;
    outputRecord.Get().hitDistance = hit.distance;

    outputRecord.OutputComplete();
}
```

… and so is sending the record off to the Work Graph runtime for processing.

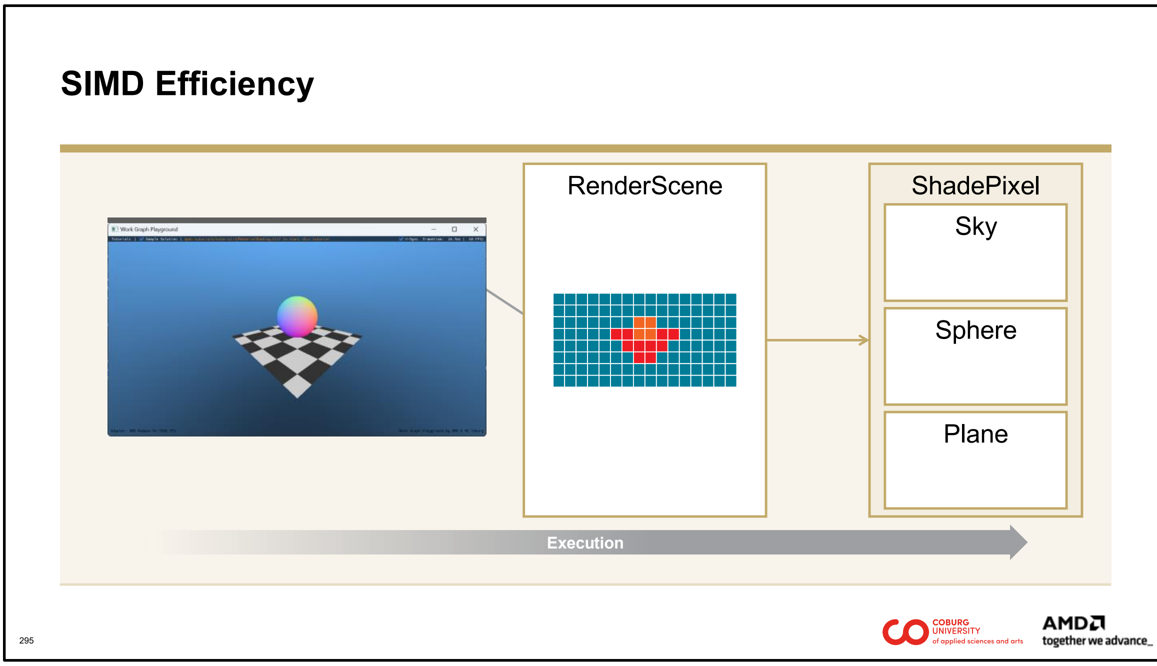


This is our work graph. First, the RenderScene node classifies the pixel and emits a record to the corresponding index in the ShadePixel node array.

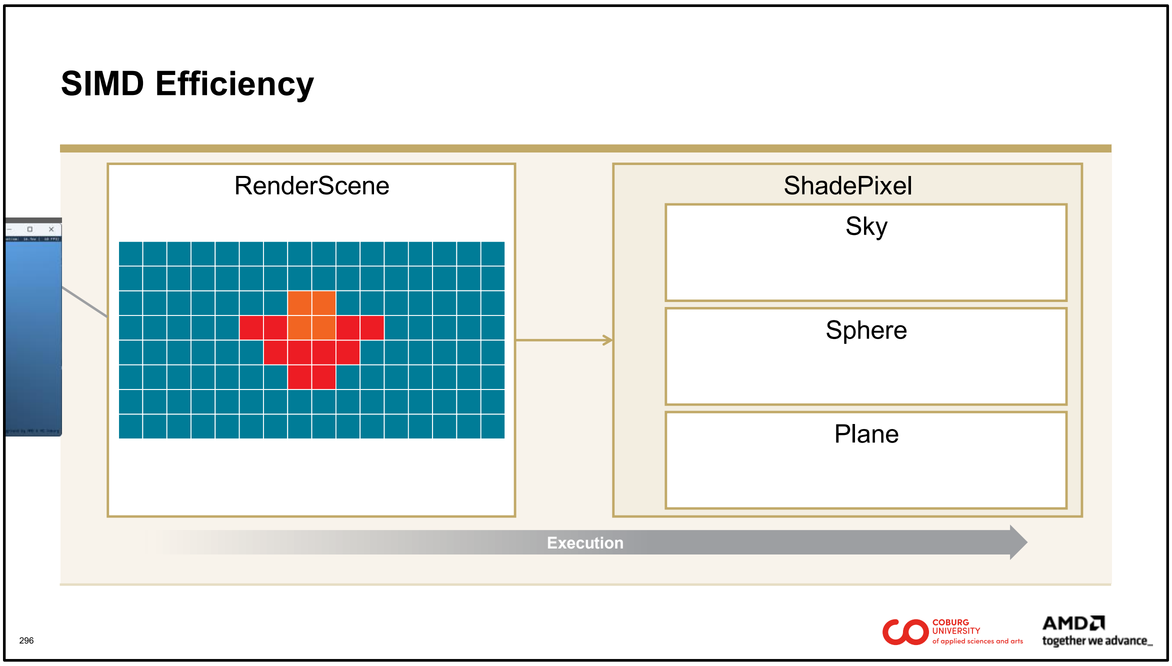
Second, the ShadePixel node array executes the shaders for each pixel in a SIMD friendly way.



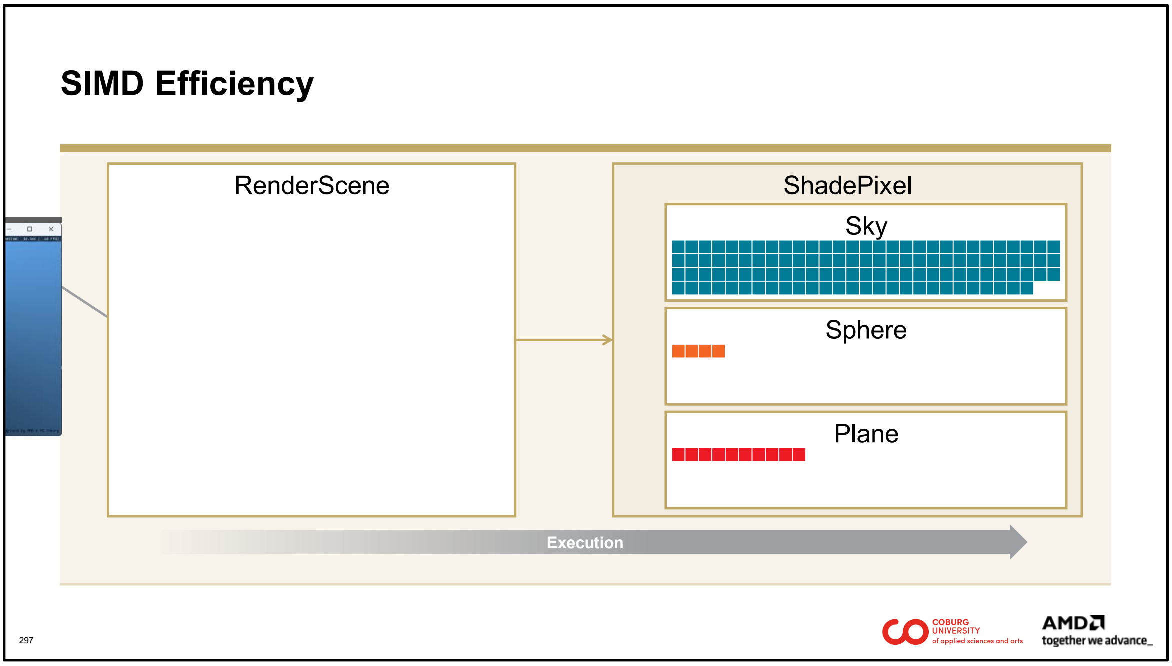
Why are node arrays SIMD friendly? Let’s go back to our coarse pixel grid for demonstration purposes.



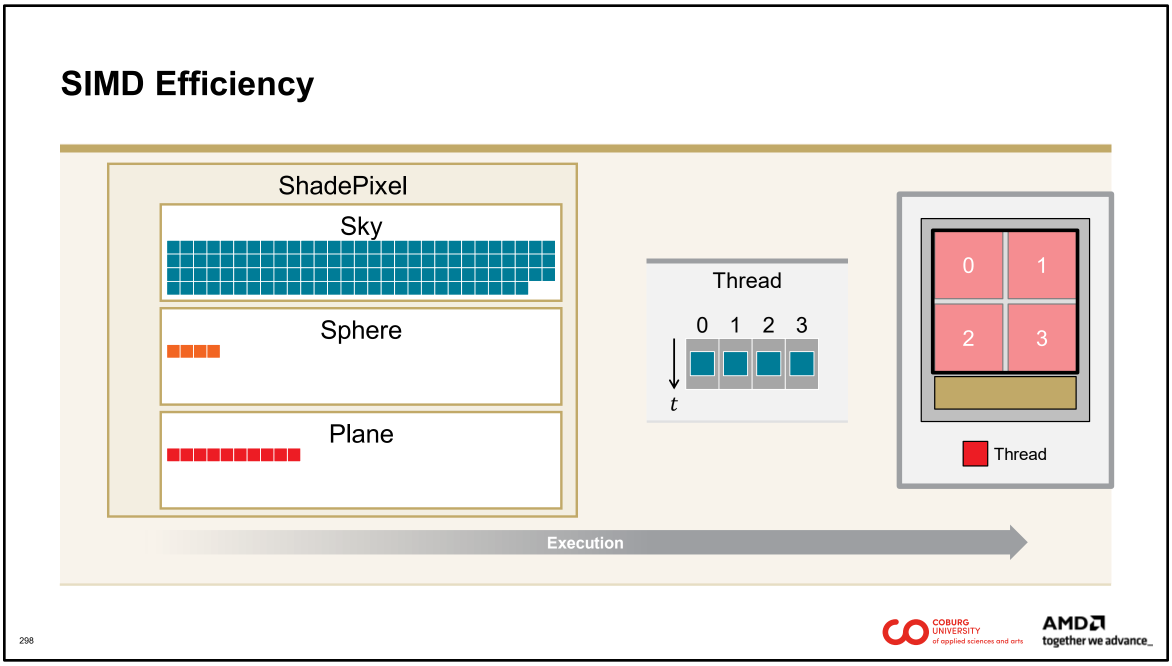
The classifier node “RenderScene” classifies each pixel and…



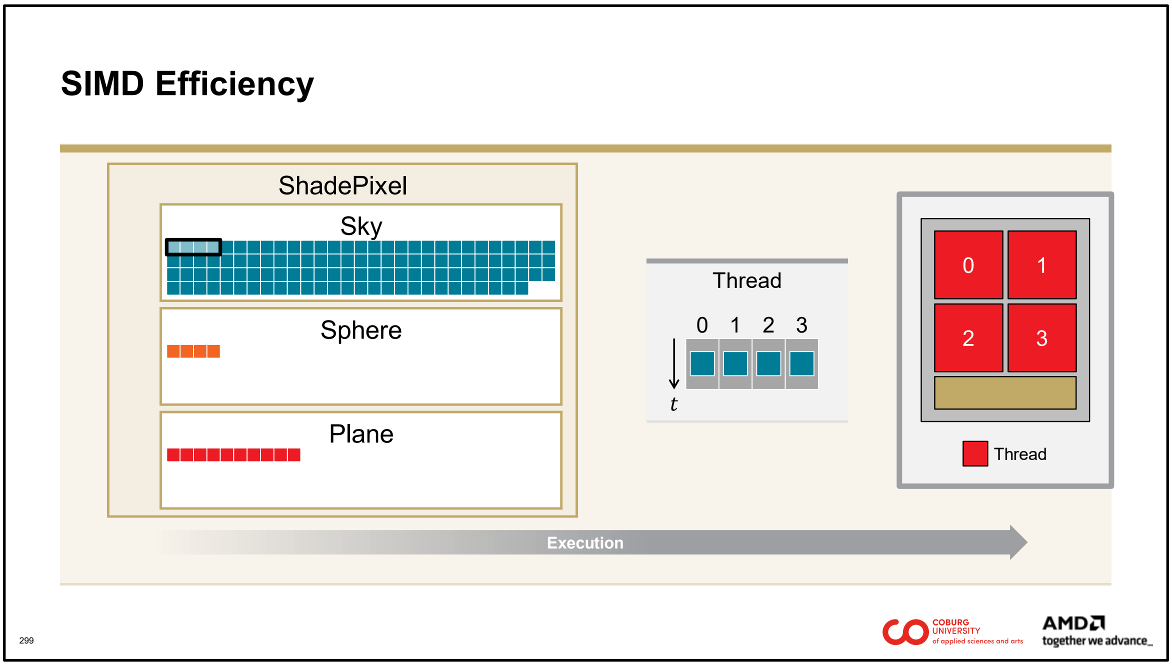
…creates a record for the consumer node in the ShadePixel node array based on the material index.



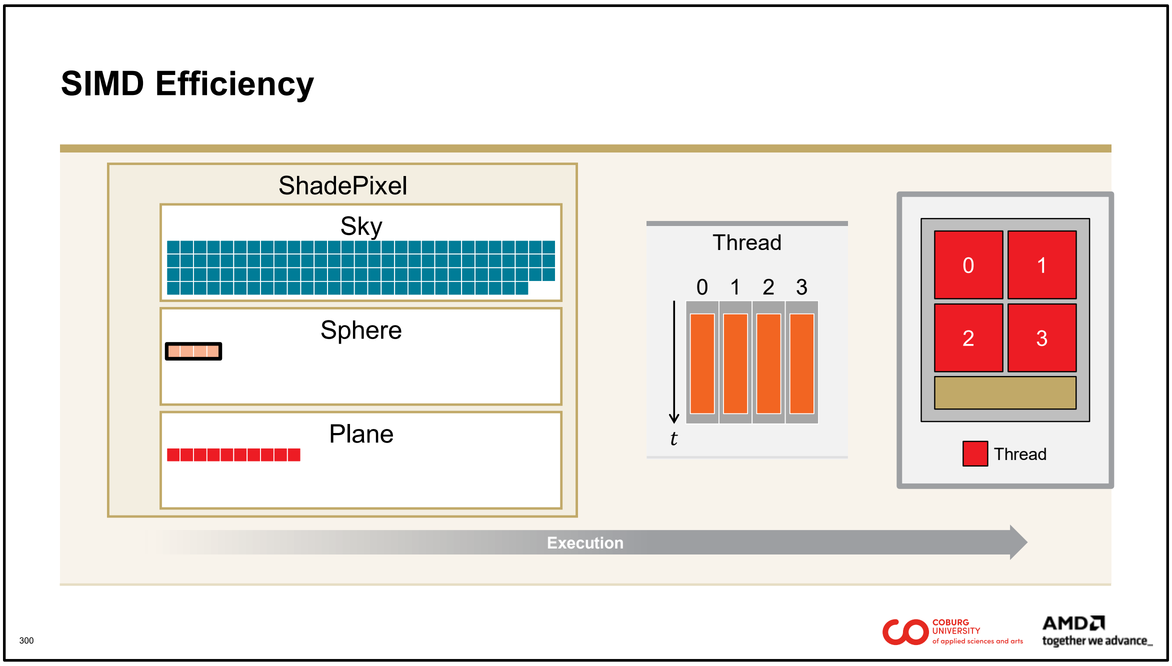
These records are then sent to the individual nodes of the node array.



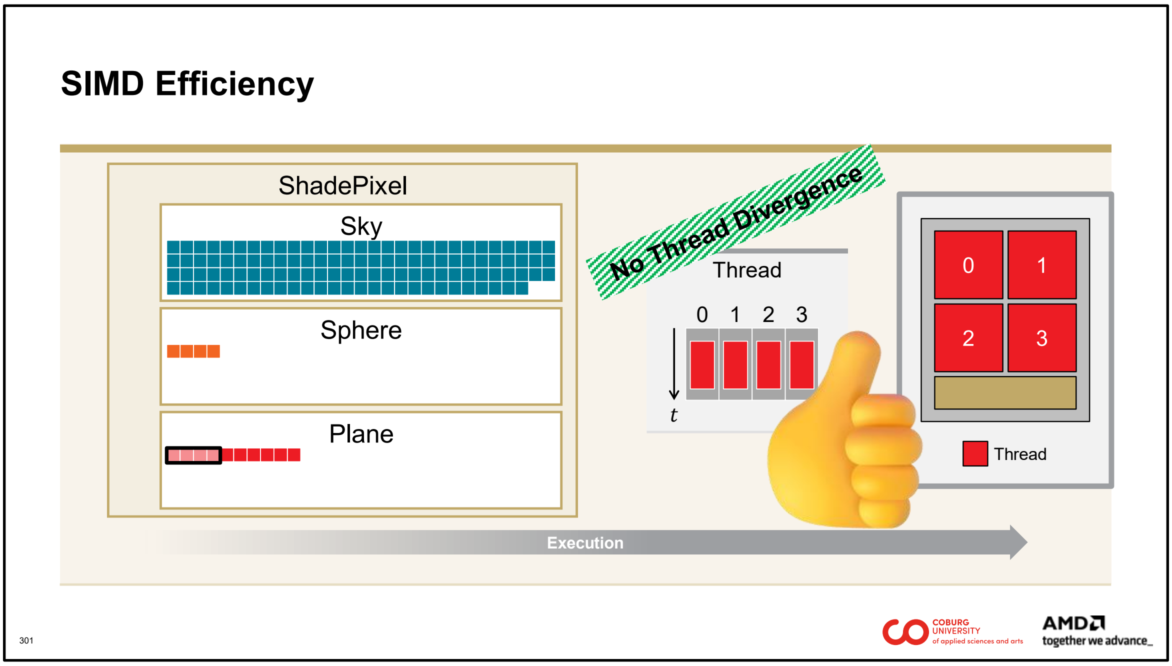
Even though we specified these nodes as “thread” launch nodes, they are still executed in thread groups…



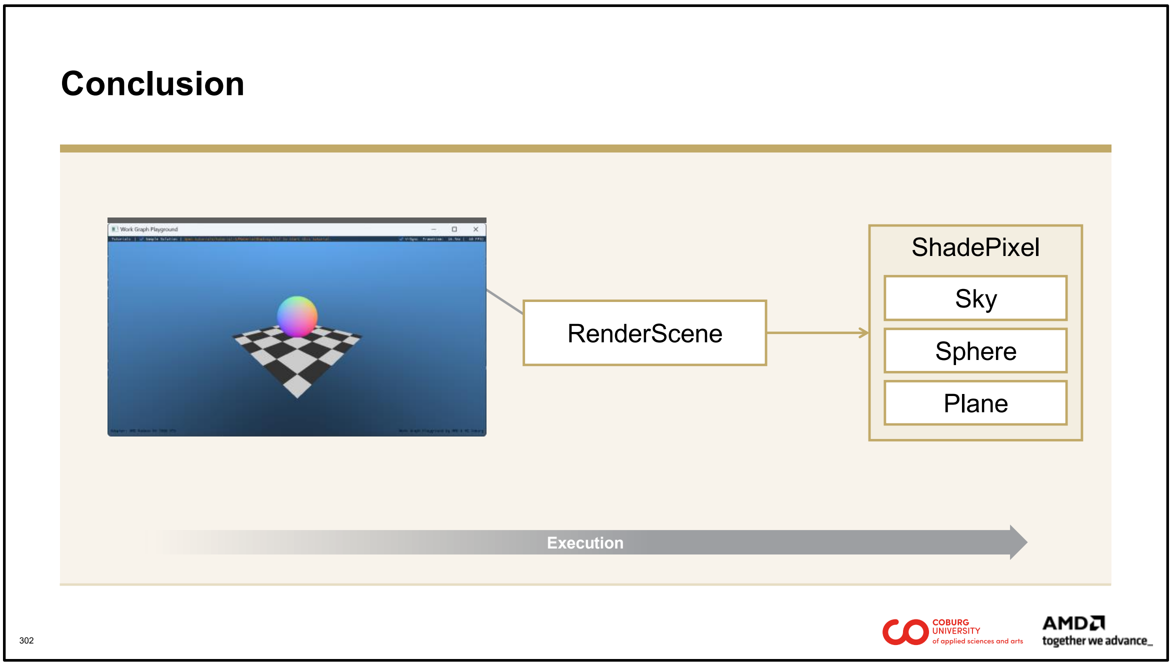
...with one record (i.e., pixel) assigned to each thread. All threads of a thread group now run in SIMD lock step, thereby reducing thread-divergence.



Likewise, for the other materials, too.

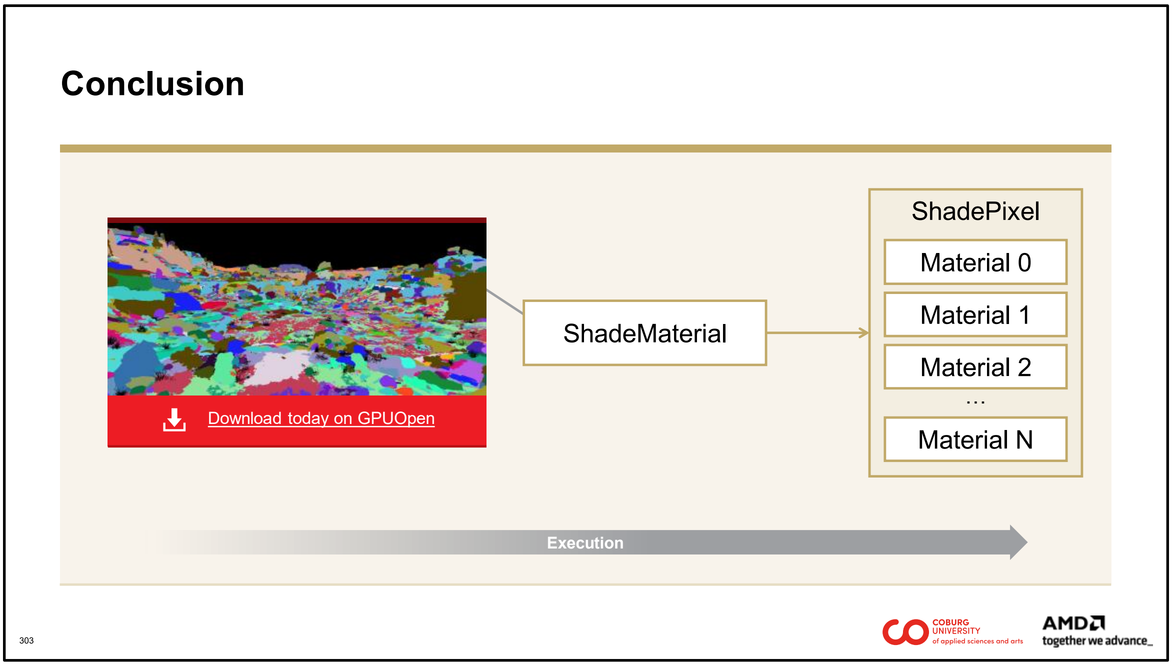


We've seen how work graphs, in combination with node arrays can help us reduce thread-divergence for classify-and-execute applications.

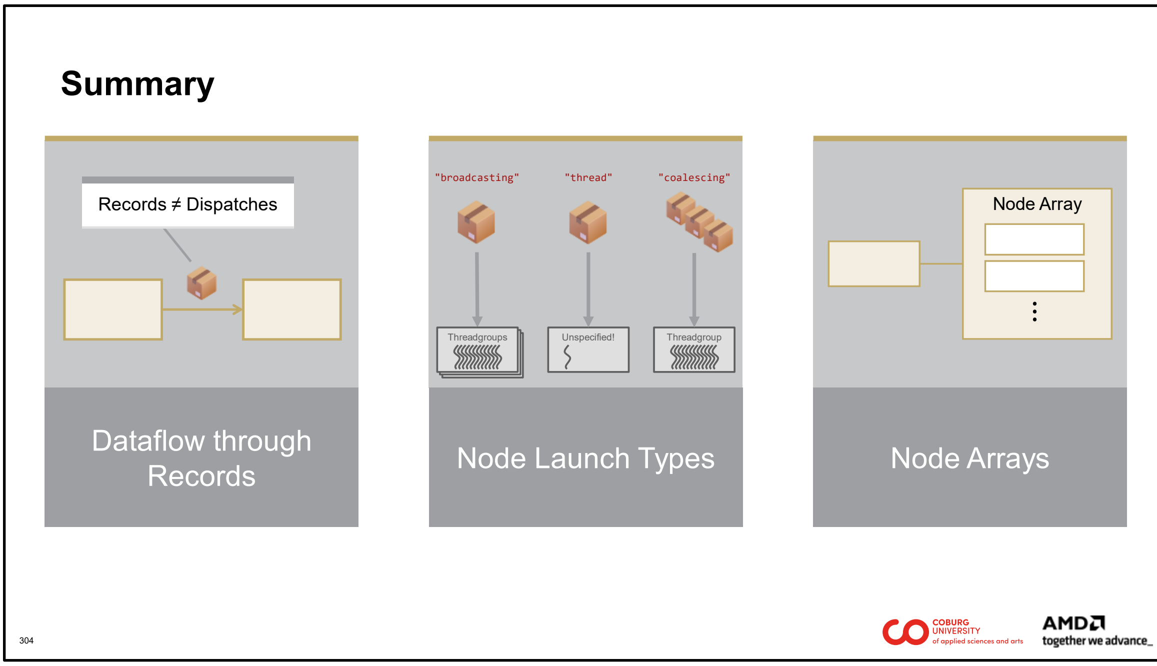


The code in these slides is available in the Work Graph Playground under `tutorials/tutorial-3/MaterialShading.hlsl`. Please follow the instructions there to get a hands-on experience with node arrays.

However, note, that in this materials-example, you will see little to no performance gains. This is because we kept our shader code simple, such that thread-divergence is not an issue. Our goal here is to teach you the principle of how node arrays work.



You can also find a standalone sample of this classify-and-execute work graph on GPUOpen.



In summary, we've seen how work graphs allow for GPU-driven dataflow through records. We've seen how and when to use the different launch modes available in work graphs. And lastly, we've seen how node arrays can help simplify our code and help us manage hundreds or thousands of nodes in a classify-and-execute scenario.

AMD

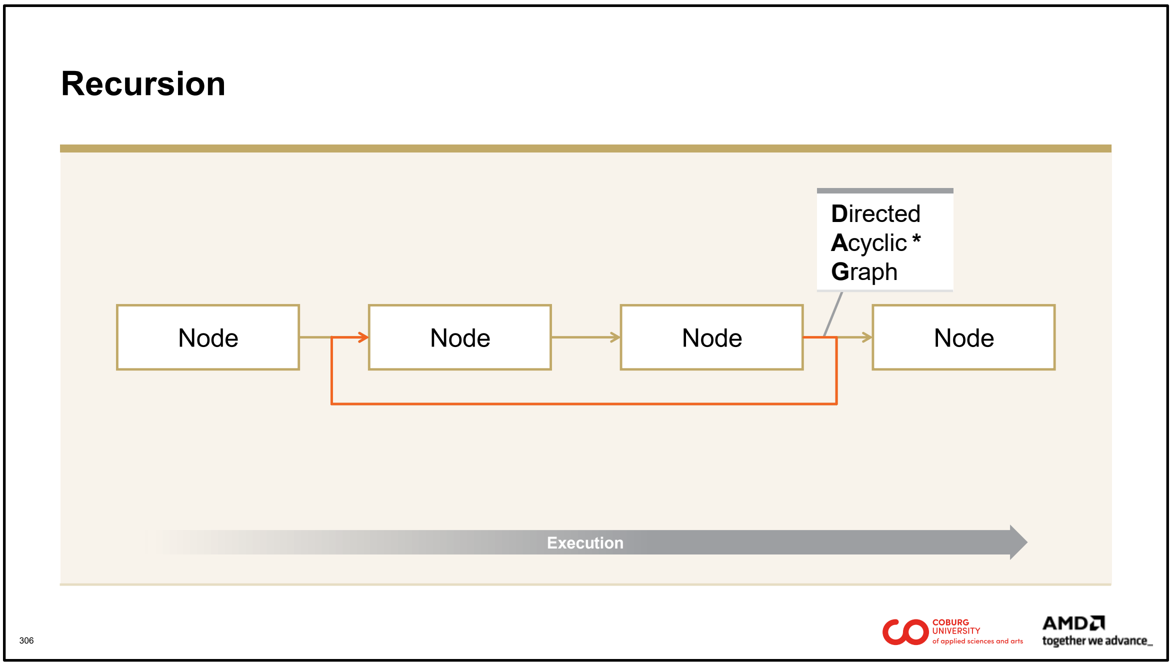
COBURG UNIVERSITY of applied sciences and arts

Advanced Work Graphs

Recursion

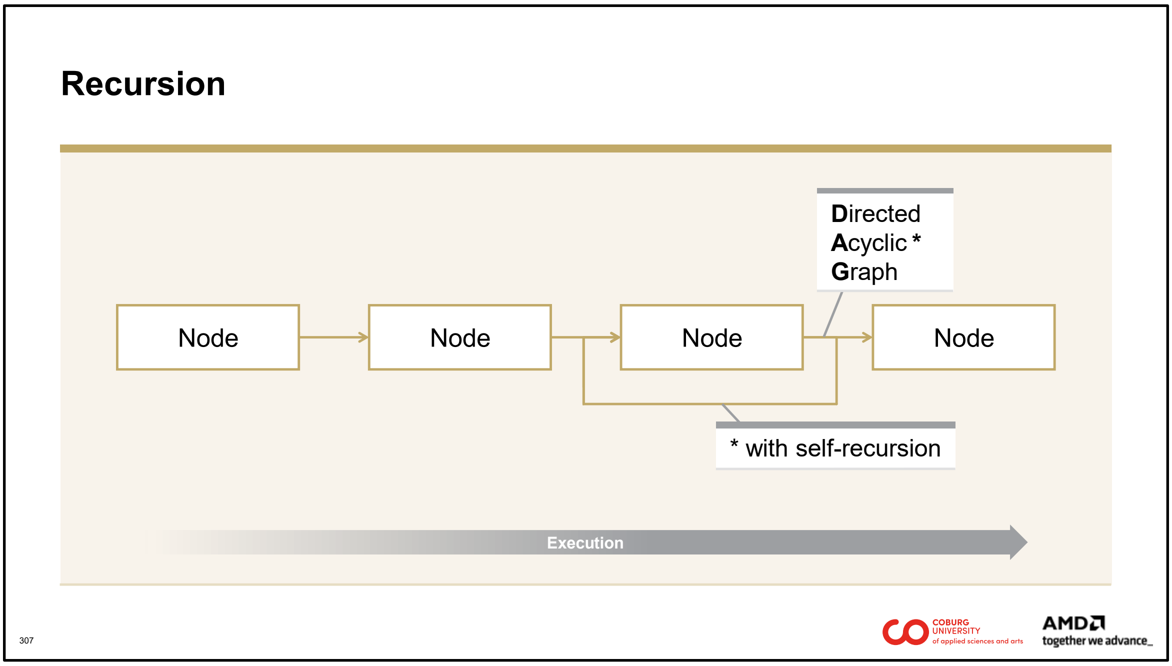
305

Next, we are going to look at how recursion is possible with Work Graphs.



We have seen before that a work graph can be classified as a directed acyclic graph. Thus, a cycle as shown here is not allowed.

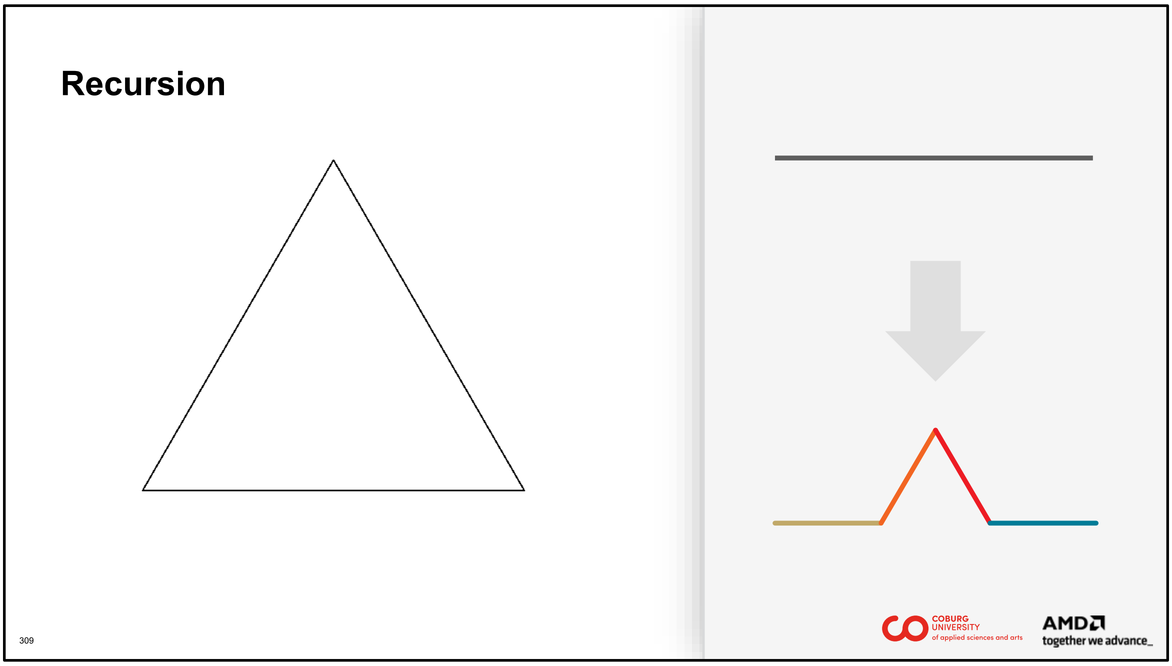
Implementing recursive algorithms with acyclic graphs is difficult, however, the Work Graphs specification allows a small exception to the acyclic constraint.



Self-recursion, or in other words, *trivial cycles* from one node to itself are allowed. These self-recursive cycles can also have a payload amplification, meaning for every incoming record, a node that's part of a self-recursive cycle can emit multiple records to itself.

There are many different applications or algorithms that can be implemented as such self-recursive nodes. These can range from different algorithms for procedural generation or subdivision (e.g., Catmull-Clark subdivision surfaces) to mathematical concepts, such as recursively evaluated fractals.

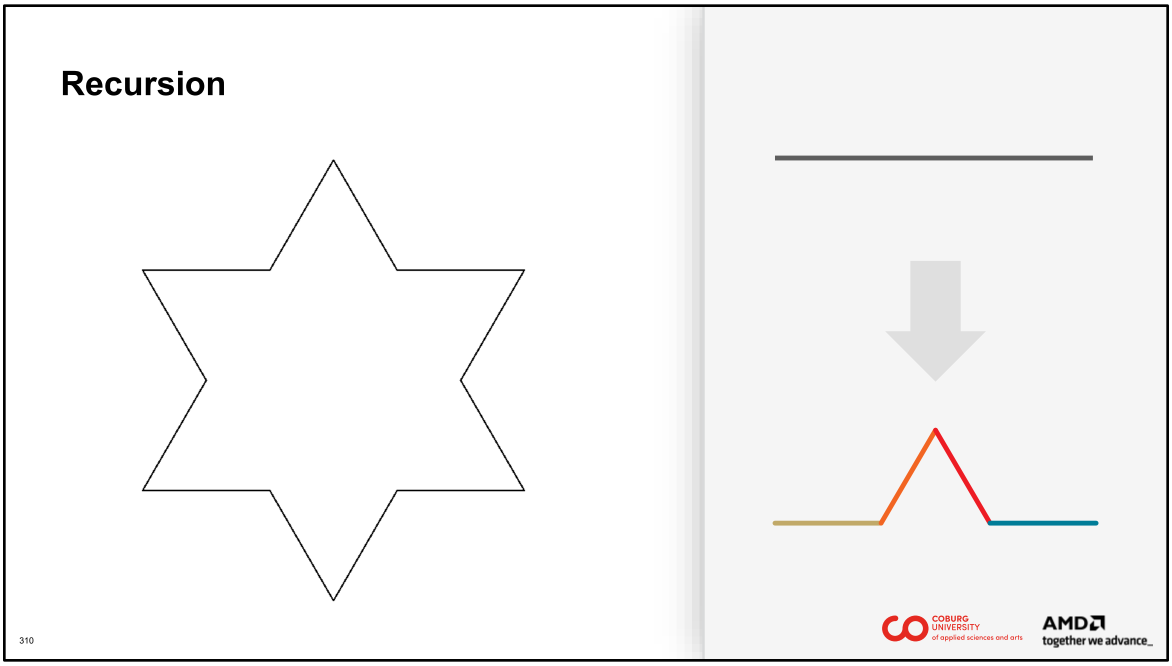
We will take a closer look at self-recursive graphs for procedural generation.



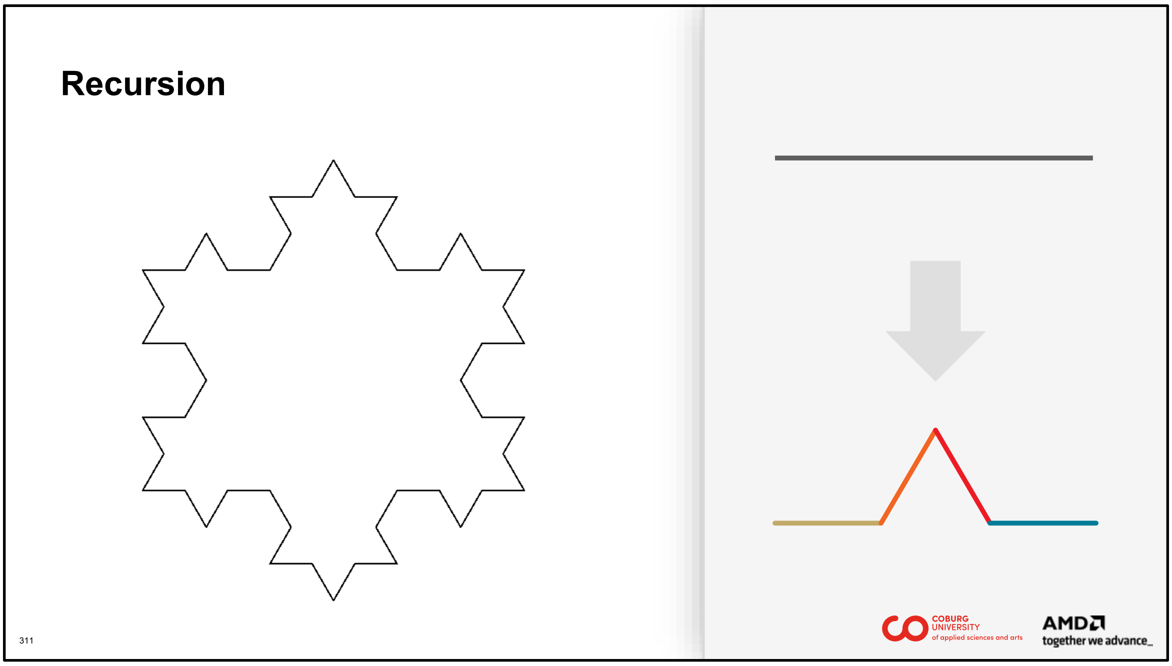
For now, we focus on a simpler example: the Koch Snowflake fractal. This fractal is part of the fourth tutorial in our Work Graph Playground App and you can find the implementation in `tutorials/tutorial-4/Recursion.hlsl`.

In simple terms, the Koch Snowflake recursively subdivides each line segment into four new line segments which form a small triangle in the middle of the original line segment, as you can see on the right part of the slide.

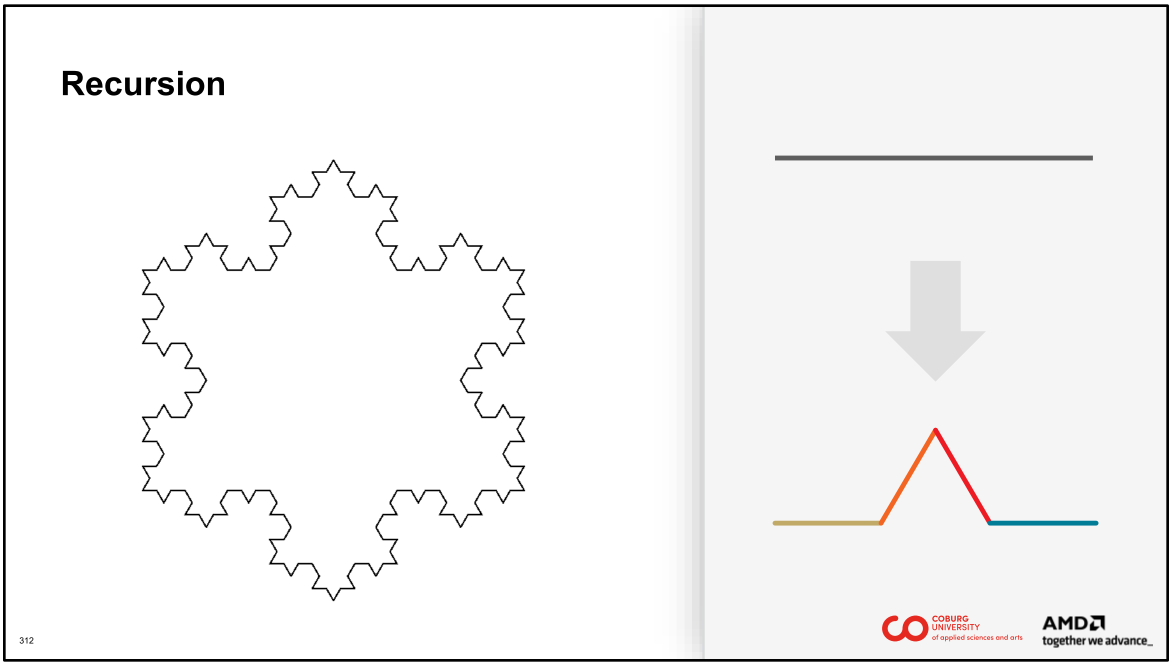
We start with an initial equilateral triangle with three line segments, as shown on the left.



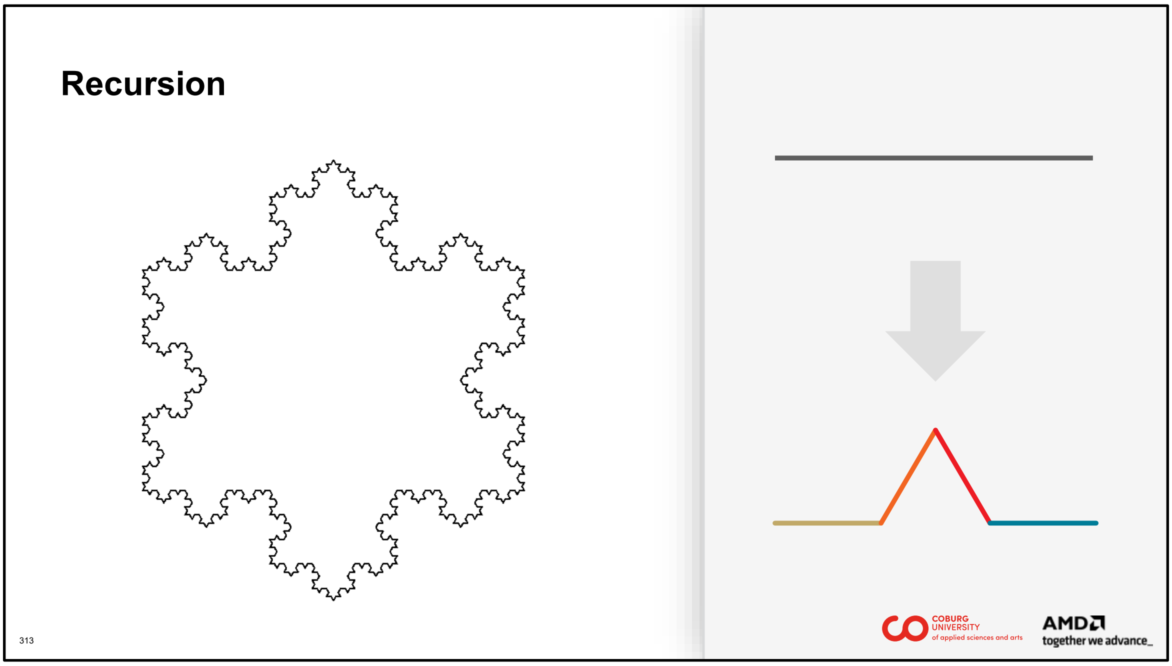
After one iteration, you can see the newly formed triangles on the edges of the initial triangle, thus transforming the initial triangle into a star shape.



After two iteration, we can start to see the snowflake shape forming.



The third…



…and fourth iteration then further refine the snowflake shape.

Recursion

tutorial-4/Recursion.hlsl

```
[Shader("node")]
[NodeLaunch("thread")]
[NodeId("Snowflake")]
void SnowflakeNode(
    ThreadNodeInputRecord<Line> inputRecord
) {
    ...
}
```

314

So how does this self-recursion look like in the shader code? Let's consider this thread node shown in the slide. This is already part of the tutorial, but there will be similar exercise for you as homework.

Recursion

tutorial-4/Recursion.hlsl

```
[Shader("node")]
[NodeLaunch("thread")]
[NodeId("Snowflake")]
void SnowflakeNode(
    ThreadNodeInputRecord<Line> inputRecord,

    [MaxRecords(4)]
    [NodeId("Snowflake")]
    NodeOutput<Line> recursiveOutput
) {
    ...
}
```


COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

Recursive nodes declare a `NodeOutput` to itself.

Recursion

tutorial-4/Recursion.hlsl

```
[Shader("node")]
[NodeLaunch("thread")]
[NodeId("Snowflake")]
void SnowflakeNode(
    ThreadNodeInputRecord<Line> inputRecord,

    [MaxRecords(4)]
    [NodeId("Snowflake")]
    NodeOutput<Line> recursiveOutput
) {
    ...
}
```

316

Note how we use the `[NodeId("Snowflake")]` attribute to both identify the node itself and the `NodeOutput` with the same node id. Thus, the node is recursively outputting records to itself.

Recursion

tutorial-4/Recursion.hlsl

```
[Shader("node")]
[NodeLaunch("thread")]
[NodeMaxRecursionDepth(4)]
[NodeId("Snowflake")]
void SnowflakeNode(
    ThreadNodeInputRecord<Line> inputRecord,

    [MaxRecords(4)]
    [NodeId("Snowflake")]
    NodeOutput<Line> recursiveOutput
) {
    ...
}
```

317

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

Self-recursion is, however, limited to fixed number of iterations, which must be set using the `[NodeMaxRecursionDepth(...)]` attribute.

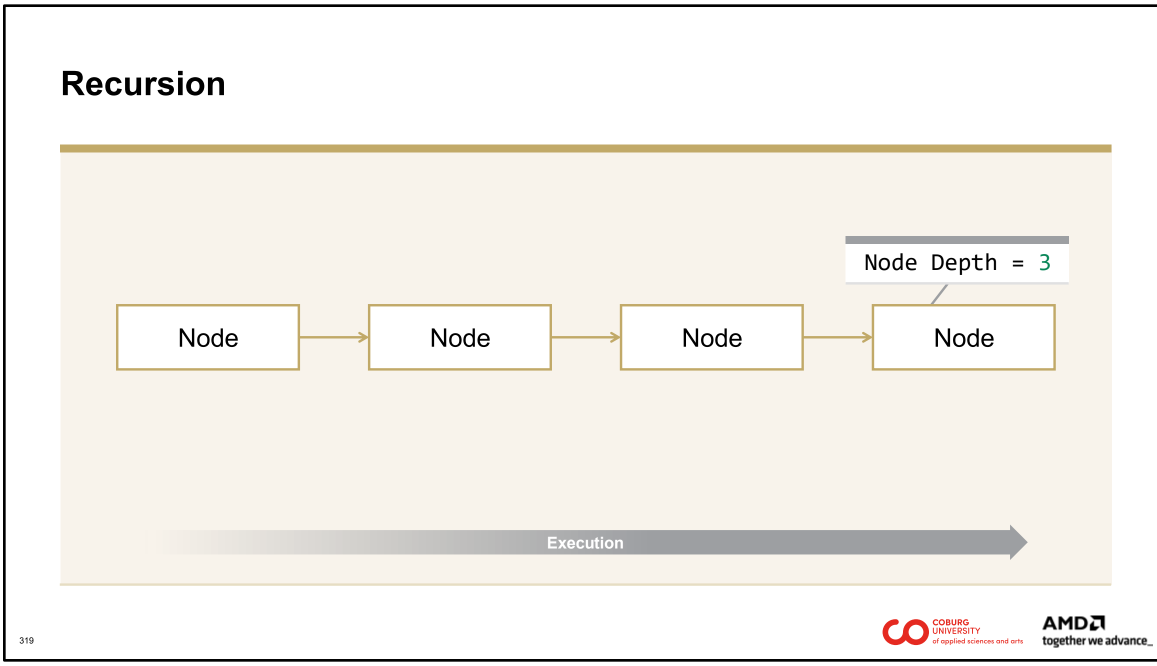
Recursion

tutorial-4/Recursion.hlsl

```
[Shader("node")]
[NodeLaunch("thread")]
[NodeMaxRecursionDepth(4)]
[NodeId("Snowflake")]
void SnowflakeNode(
    ...
) {
    ...
    // Check if we have reached the recursion limit.
    const bool hasOutput = GetRemainingRecursionLevels() != 0;
}
```

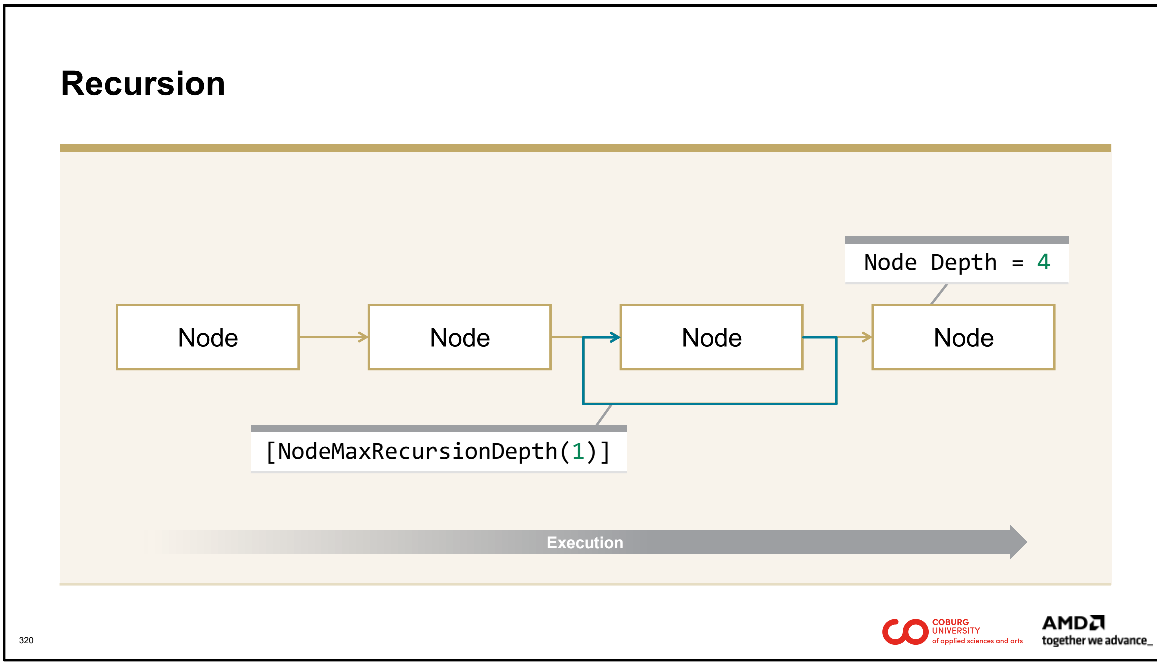
318

In each recursive iteration, we can then query the number of remaining iterations with the `GetRemainingRecursionLevels()` intrinsic. If this intrinsic returns `0`, then the node is no longer allowed to emit self-recursive records.

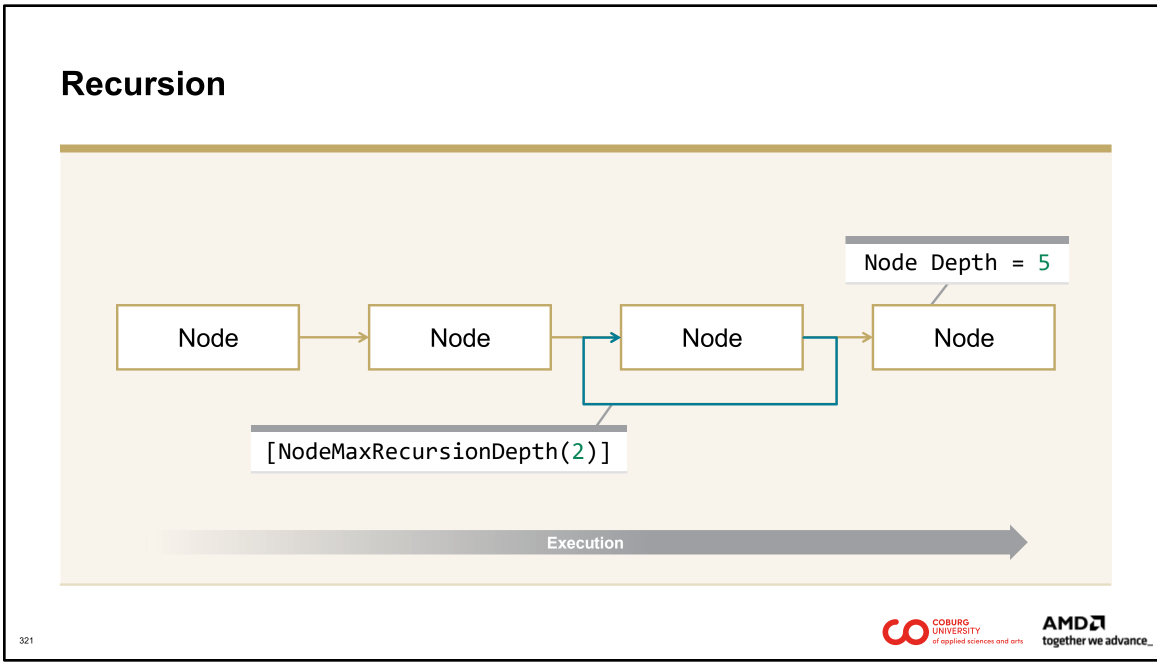


As a reminder, the longest chain of nodes can not exceed the limit of 32 nodes. When computing this longest chain of nodes, the maximum number of recursive iterations (`[NodeMaxRecursionDepth(...)]`) add to the chain length.

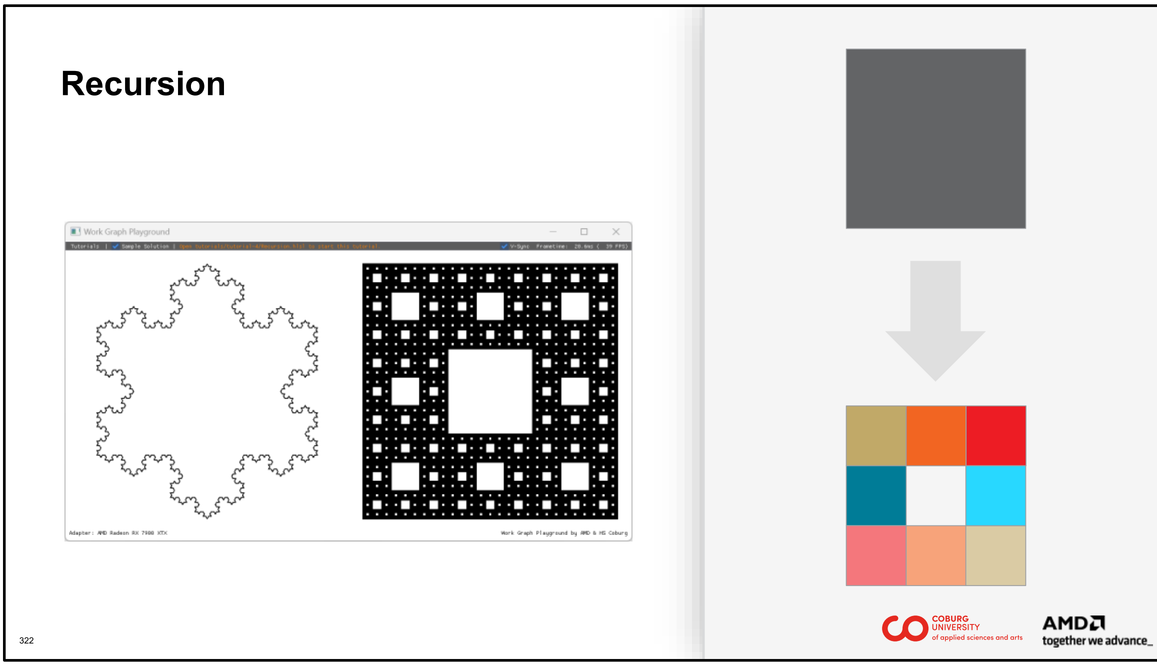
In this example the node on the far-right has a node depth in the graph of three.



If we add a self-recursion loop to the graph, this node depth increases by the value of the `[NodeMaxRecursionDepth(...)]` attribute.



Increasing `[NodeMaxRecursionDepth(...)]` further increases the node depth of the last node.



As a homework assignment, your task is to implement another recursive fractal in the Work Graph Playground App: the Menger sponge.

Follow the instructions in `tutorials/tutorial-4/Recursion.hlsl` and implement the fractal. You can verify your solution by comparing it to the provided sample solution.

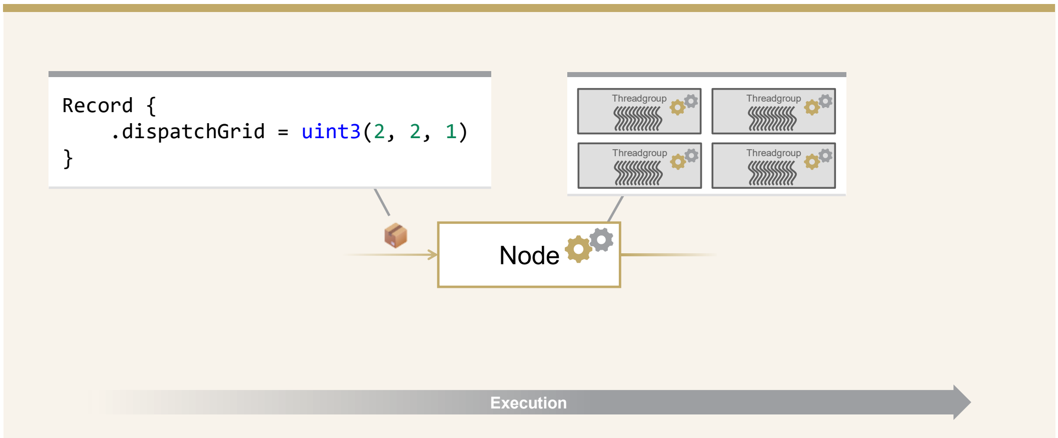
Advanced Work Graphs

Synchronization

323

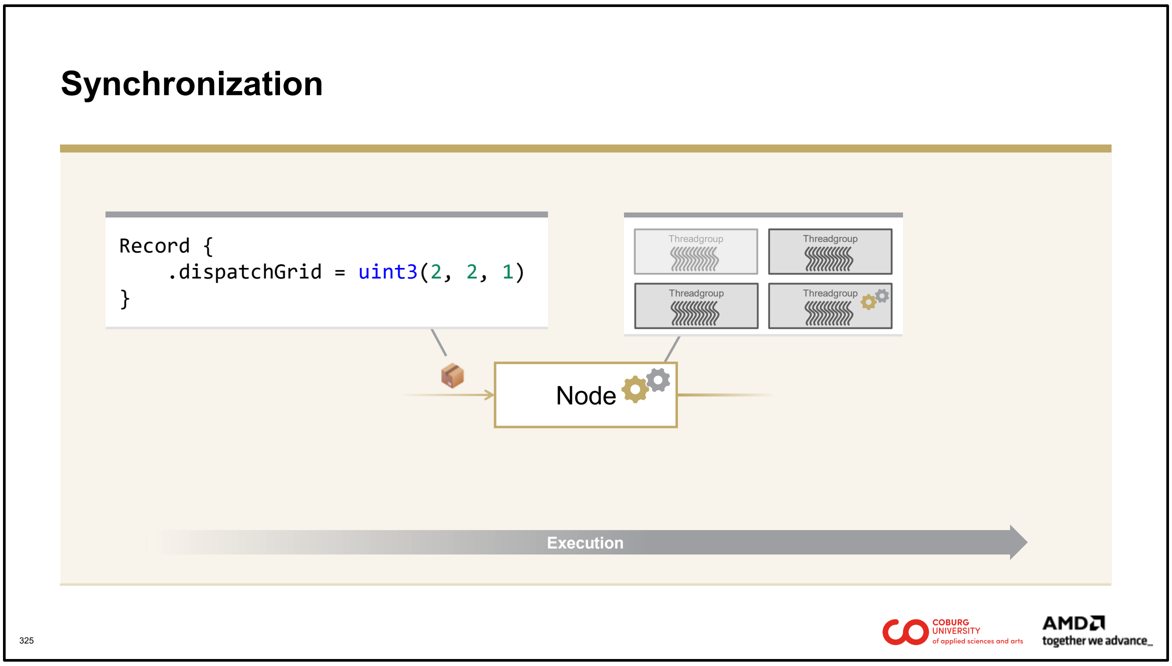
Another aspect for advanced work graphs is synchronization of thread groups in broadcasting launches.

Synchronization

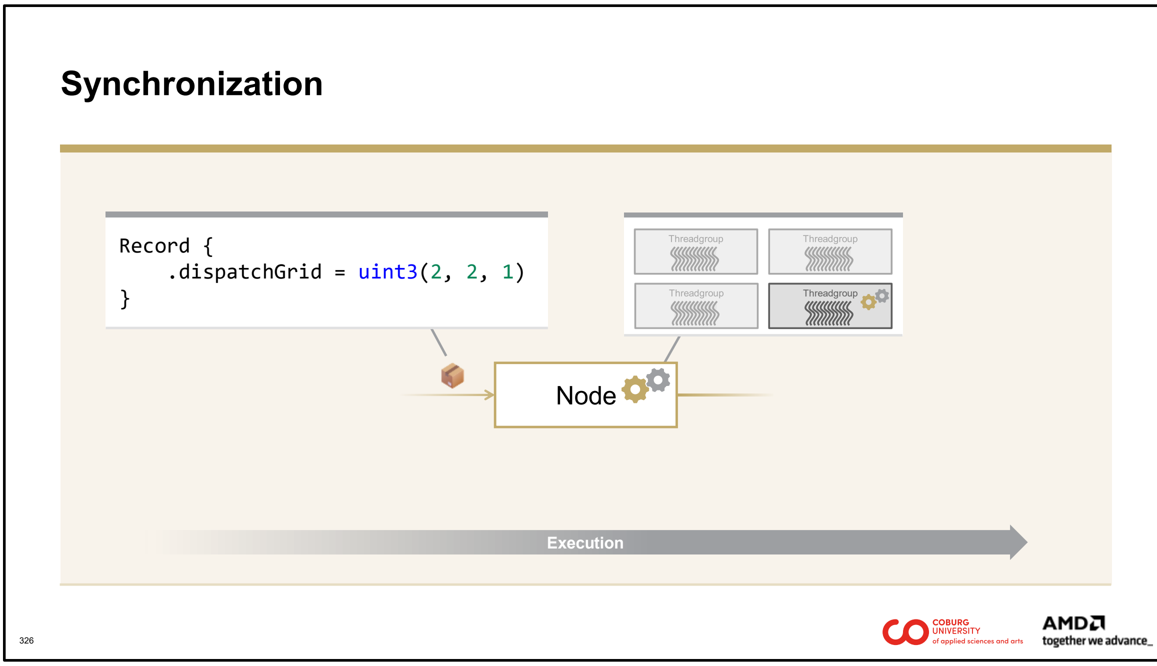


Before, we dive into the code, let's quickly explain what we mean by this. Consider a broadcasting node that is part of a longer chain of nodes, e.g., a chain of image filters. In such a chain, we might have data-dependencies between different nodes in the chain, i.e., we can only launch the next node, if all thread groups of the previous node have finished executing.

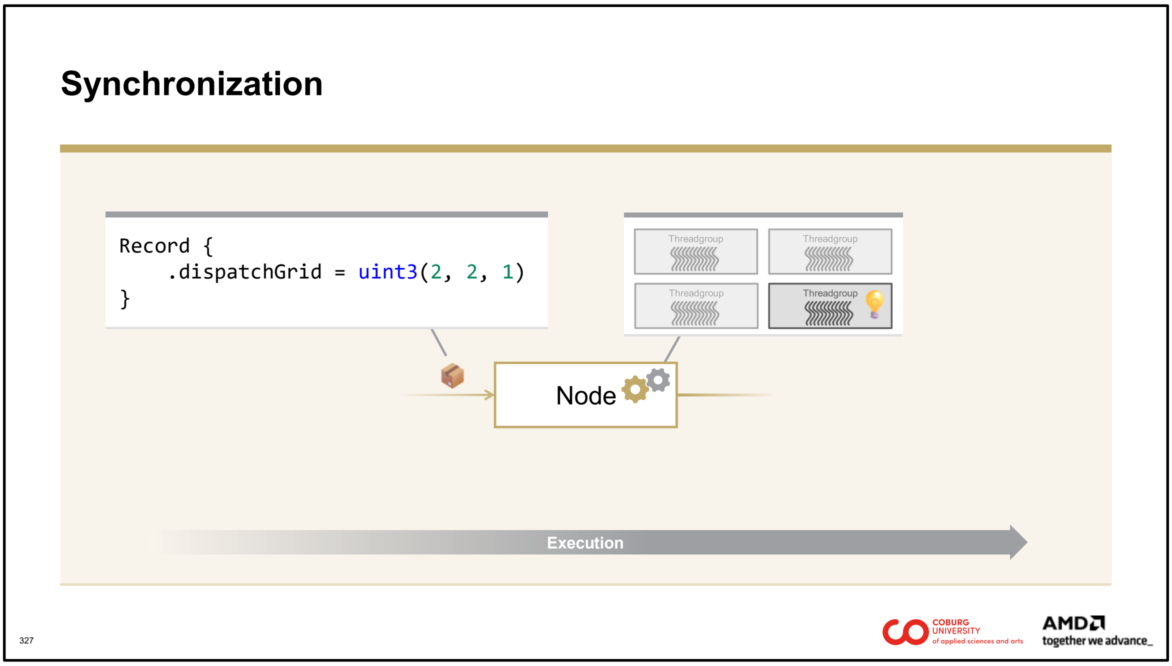
In our example, our node receives an incoming record. Our node is using the broadcasting launch mode. The record sets the dispatch grid of the node to 2x2 thread groups. We assume that these thread groups all run in parallel on our GPU.



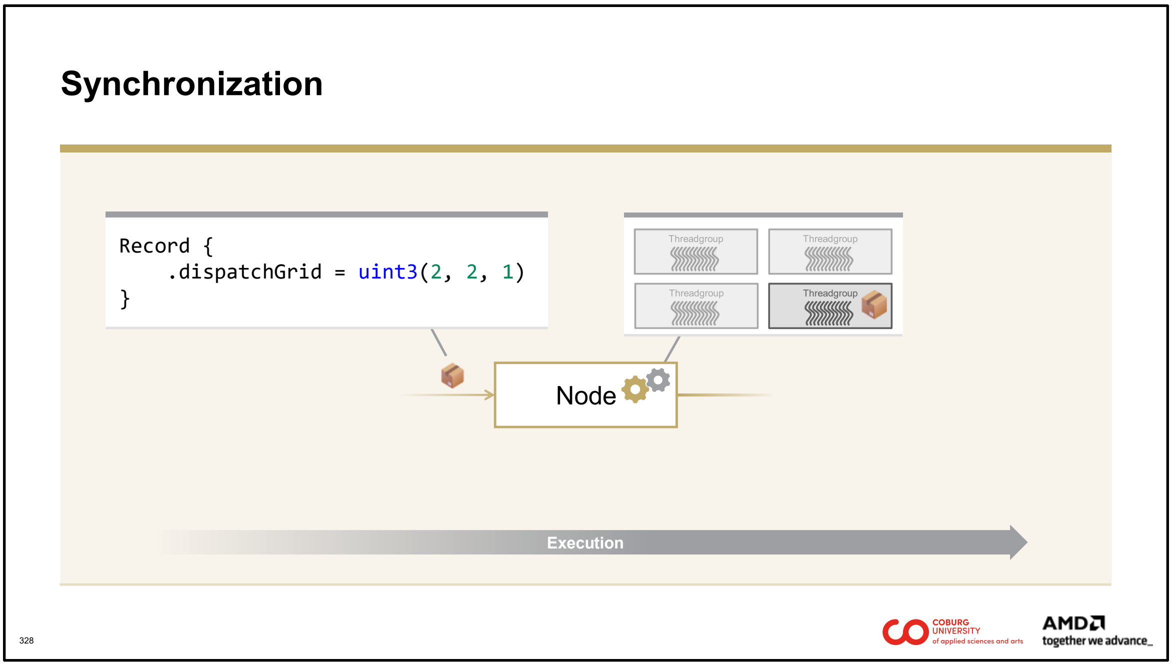
After a while, the thread groups terminate one after the other…



…until only one thread group remains.



Synchronization in broadcasting nodes allows this last thread group to realize that it is in fact the last one.



Thus, it can carry out a final special operation, such as emitting a record for the next node, as we now know that all thread groups in our broadcasting node have finished execution and any data that they might have produced is now ready to be processed by a following node(s).

Synchronization

329

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

In the fifth tutorial of the Work Graphs Playground App, we are going to use such synchronization to draw a bounding box around this dancing trail of circles.

Synchronization

tutorial-5/Synchronization.hlsl

```
[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeDispatchGrid(32, 1, 1)]
[NumThreads(32, 1, 1)]
void ComputeBoundingBox(
    ...
    DispatchNodeInputRecord<Record> inputRecord
) {
    ...
    DrawRect(...);
}
```

330

In the tutorial, we're using a node with `"broadcasting"` launch mode. The node is dispatched with 32 thread groups and 32 threads in each thread group.

Each thread then computes a position and radius of a circle and draws the circle on screen.

We now want to compute the bounding box of all circles. Once all of the thread groups have finished computing the bounding box in parallel, we want to have the last thread group draw the resulting bounding box to the screen.

Synchronization

```
tutorial-5/Synchronization.hlsl
[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeDispatchGrid(32, 1, 1)]
[NumThreads(32, 1, 1)]
void ComputeBoundingBox(
    ...
    DispatchNodeInputRecord<Record> inputRecord
) {
    ...
    if(!inputRecord.FinishedCrossGroupSharing()) return;

    DrawRect(...);
}
```

331

With `FinishedCrossGroupSharing()`, Work Graphs provide a method on the input record, that returns true, if the calling thread group is the last one to call this method.

Synchronization

tutorial-5/Synchronization.hlsl

```
struct [NodeTrackRWInputSharing] Record {
    ...
};

[Shader("node")]
...
void ComputeBoundingBox(
    ...
    DispatchNodeInputRecord<Record> inputRecord
) {
    ...
    if(!inputRecord.FinishedCrossGroupSharing()) return;

    DrawRect(...);
}
```

332

Since this is carried out on the input record, the input record needs to be prepared to support such an operation. Therefore, you must add the `[NodeTrackRWInputSharing]` attribute to the record struct, as shown above.

Synchronization

tutorial-5/Synchronization.hlsl

```
struct [NodeTrackRWInputSharing] Record {
    ...
};

[Shader("node")]
...
void ComputeBoundingBox(
    ...
    RWDispatchNodeInputRecord<Record> inputRecord
) {
    ...
    if(!inputRecord.FinishedCrossGroupSharing()) return;

    DrawRect(...);
}
```

333

As `FinishedCrossGroupSharing` “writes” into the record, you need to adjust the input record declaration to use `RWDispatchnodeInputRecord`.

Note that this adds an even more powerful capability: The `RW` prefix allows you to communicate between thread groups in broadcasting mode.

For `"thread"` and `"coalescing"` node launches, the input node declaration receives the same `RW` prefix, if you want write to your record.

Synchronization

tutorial-5/Synchronization.hlsl

```
[Shader("node")]
...
void ComputeBoundingBox(
    ...
    RWDispatchNodeInputRecord<Record> inputRecord
) {
    ...
    InterlockedMax(inputRecord.Get().aabbmax.y, ...);
    ...
    if(!inputRecord.FinishedCrossGroupSharing()) return;

    DrawRect(...);
}
```

334

We use this ability to write to a shared record in our tutorial: We compute the bounding box with atomic min/max operations, where all threads of our dispatch write to same input record.

Synchronization

tutorial-5/Synchronization.hlsl

```
[Shader("node")]
...
void ComputeBoundingBox(
    ...
    RWDispatchNodeInputRecord<Record> inputRecord
) {
    ...
    InterlockedMax(inputRecord.Get().aabbmax.y, ...);

    Barrier(NODE_INPUT_MEMORY, DEVICE_SCOPE | GROUP_SYNC);
    ...
    if(!inputRecord.FinishedCrossGroupSharing()) return;

    DrawRect(...);
}
```

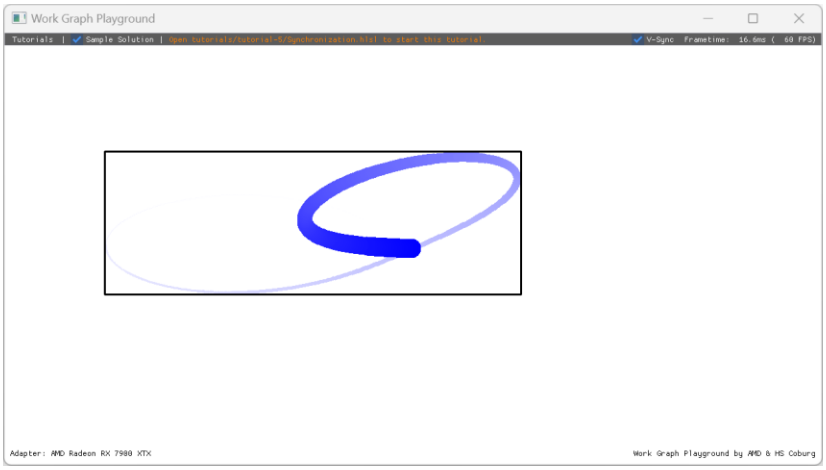
335

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

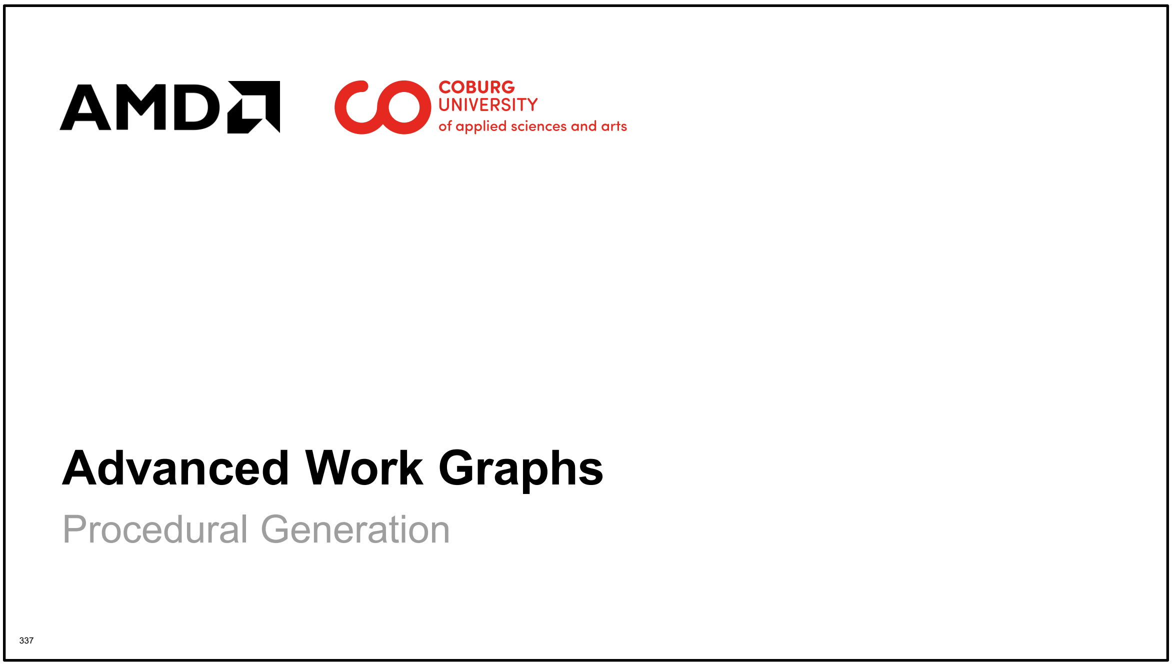
Since we write to record memory from all threads concurrently, we must use a barrier before reading back the resulting bounding box.

Synchronization

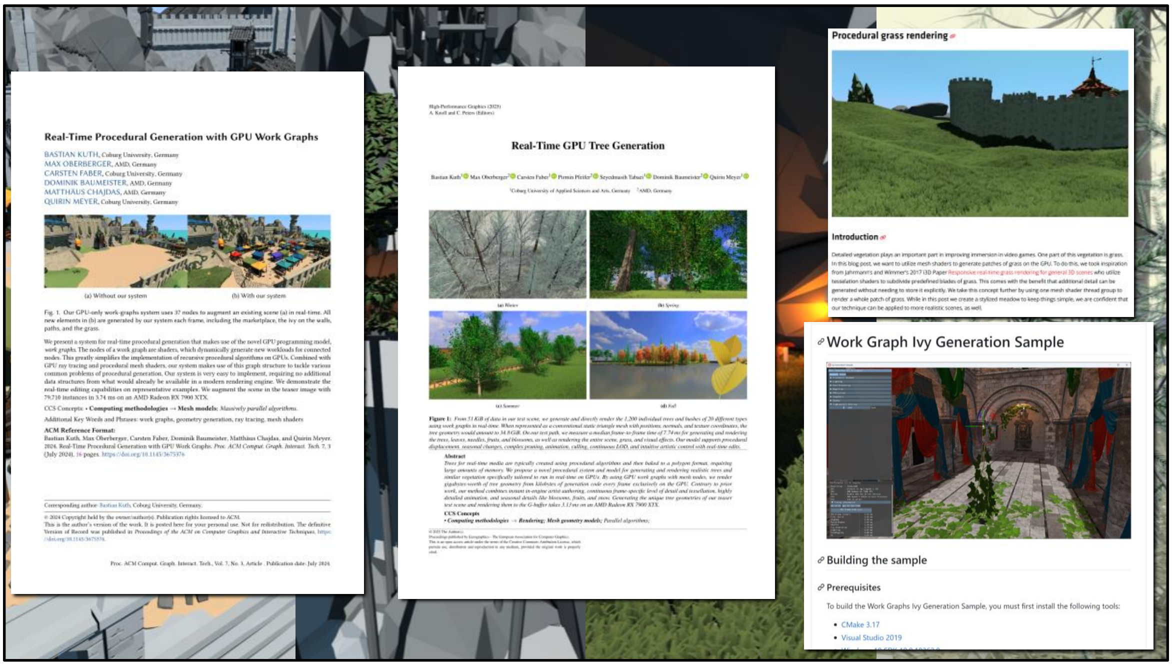


336

Now, we have a nice bounding box!



In this section, we want to show you how Work Graphs can be used for procedural generation.

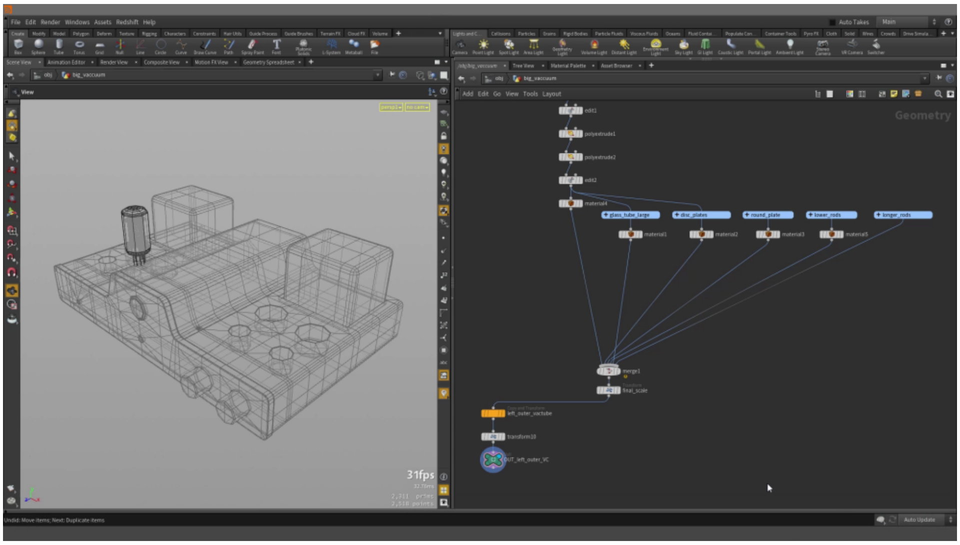


We will present four examples that are based on two papers [Kuth et al. 2024, Kuth et al. 2025], some blog posts, and samples that we have published.

Blog Posts:
https://gpuopen.com/learn/work_graphs_mesh_nodes/work_graphs_mesh_nodes-getting_started/
https://gpuopen.com/learn/work_graphs_mesh_nodes/work_graphs_mesh_nodes-intro/
https://gpuopen.com/learn/work_graphs_mesh_nodes/work_graphs_mesh_nodes-procedural_generation
https://gpuopen.com/learn/work_graphs_mesh_nodes/work_graphs_mesh_nodes-tips_tricks_best_practices/ https://github.com/GPUOpen-LibrariesAndSDKs/WorkGraphsHelloMeshNodes

Samples:
https://gpuopen.com/learn/rgp-work-graphs/
https://gpuopen.com/learn/work_graphs_learning_sample/

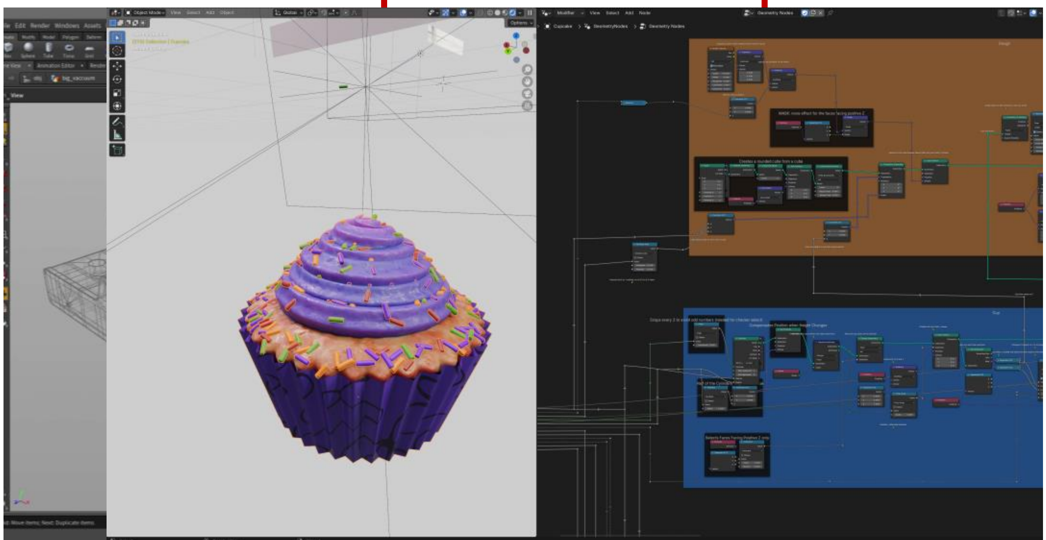
Houdini



339

To get started, let‘s look at existing procedural software. An obvious mention is Houdini.

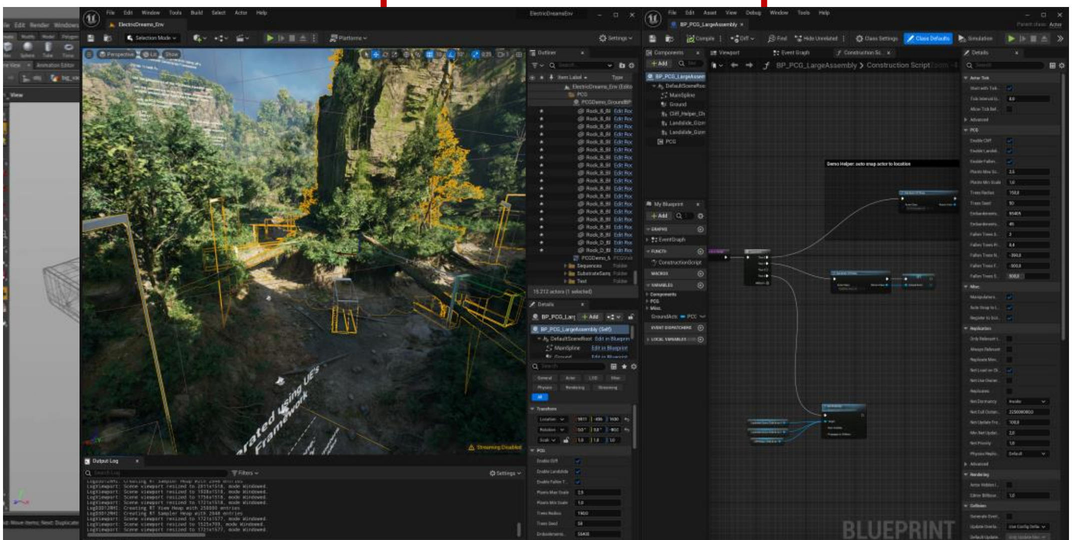
Houdini



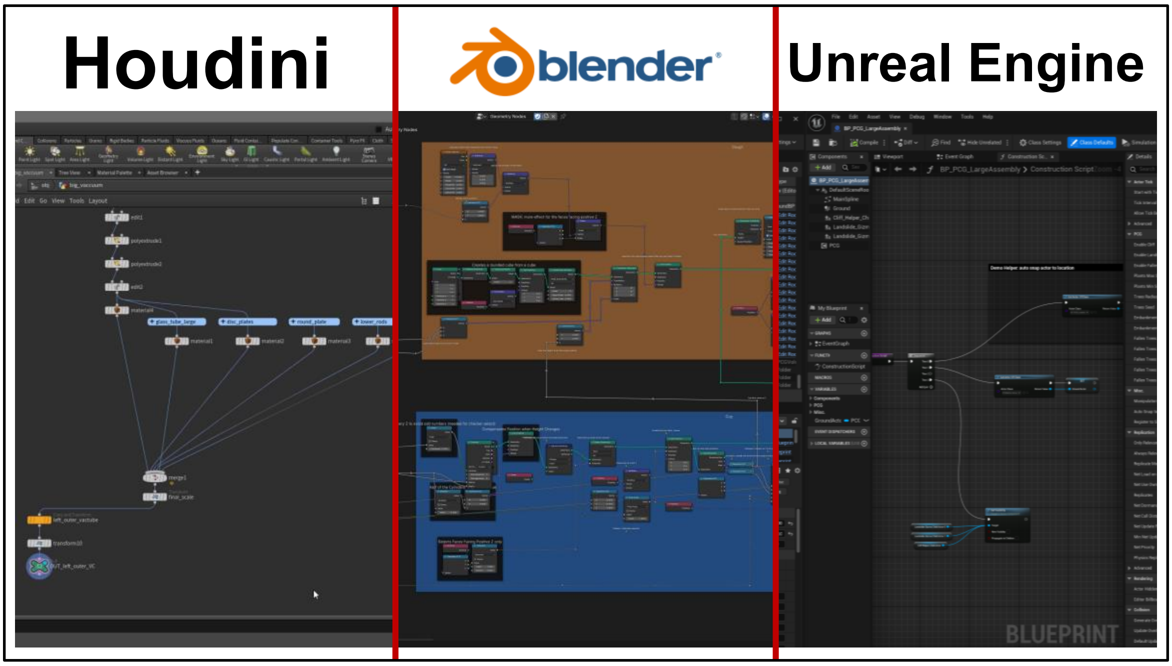
Second one would be Blender with its geometry nodes.

Logo from https://www.blender.org/about/logo/

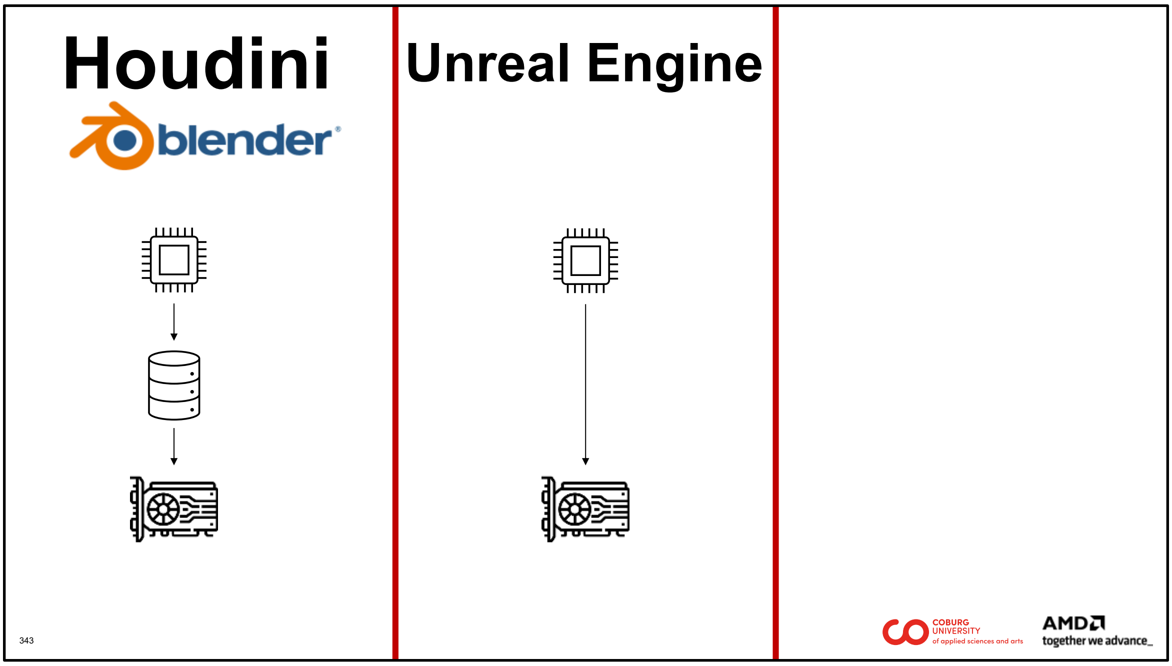
Houdini | blender | Unreal Engine



But also Unreal Engine now has a built-in system named PCG.

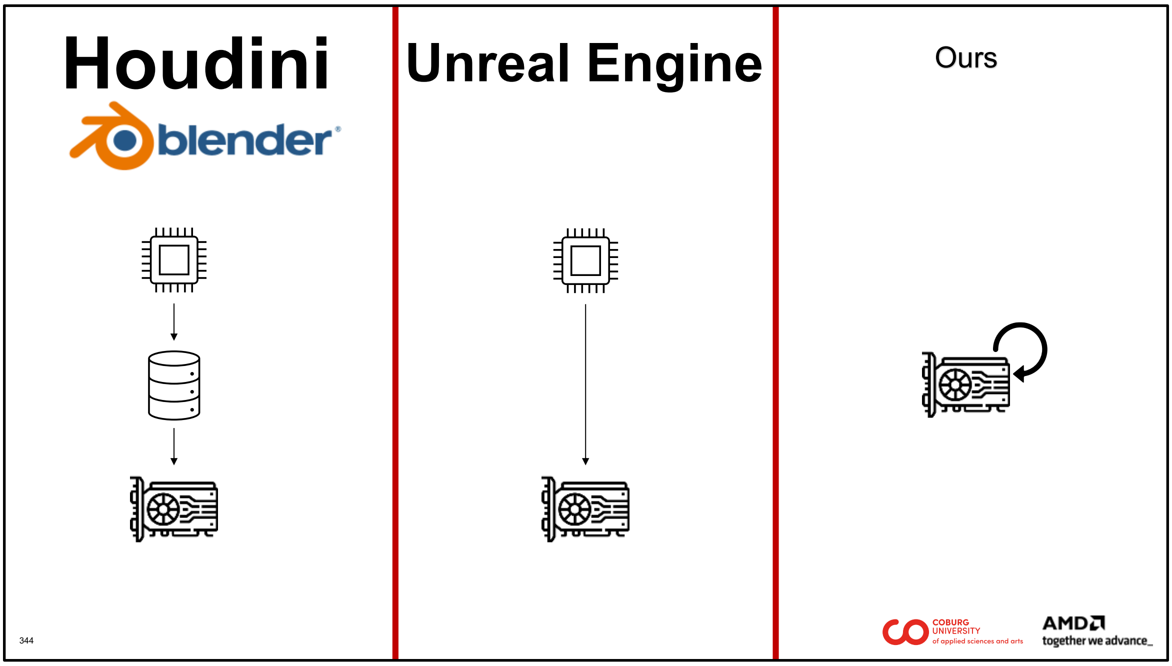


All these tools have one thing in common: The generation is controlled by designing node graphs consisting of reusable nodes.



When we look at how or where these tools generate, we can see that this usually happens on CPU. Then the result gets exported to a polygon format onto disk. Finally, the ready-made model is then uploaded to the GPU for rendering by a game engine.

With the new Unreal Engine PCG system, the export step is skipped: as the generation happens in-engine, there is no need for an export.

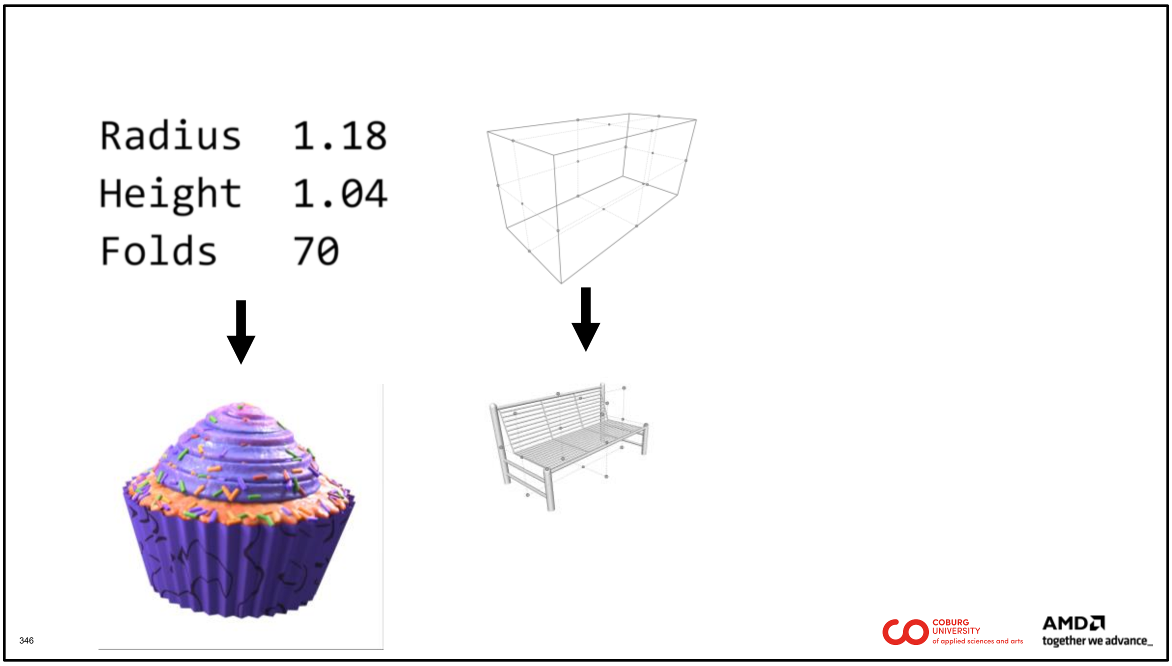


What we want to do with Work Graphs today is to totally skip the CPU part: The GPU generates everything it needs for rendering.

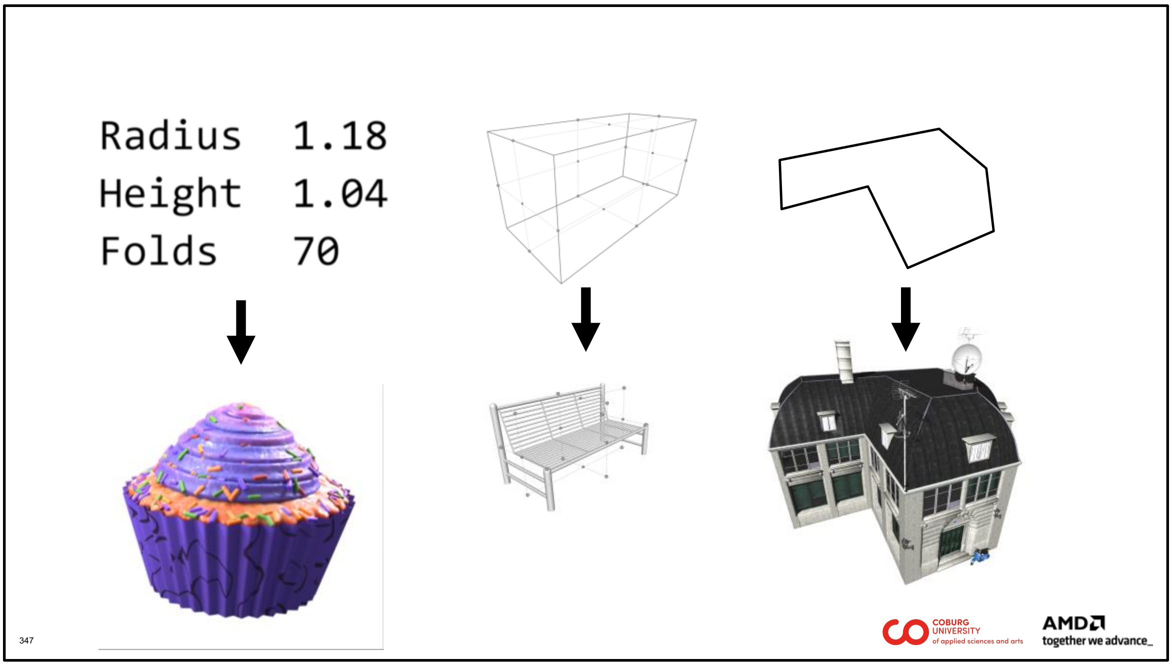
Radius 1.18
Height 1.04
Folds 70

345

We already mentioned the node graphs that control the procedural generation, but what are the edges connecting the nodes? We call the data that flows between the edges *control parameters.* A node receives control parameters and outputs control parameters. A very simple example for this would be generation of this muffin: Three parameters control the shape of it.

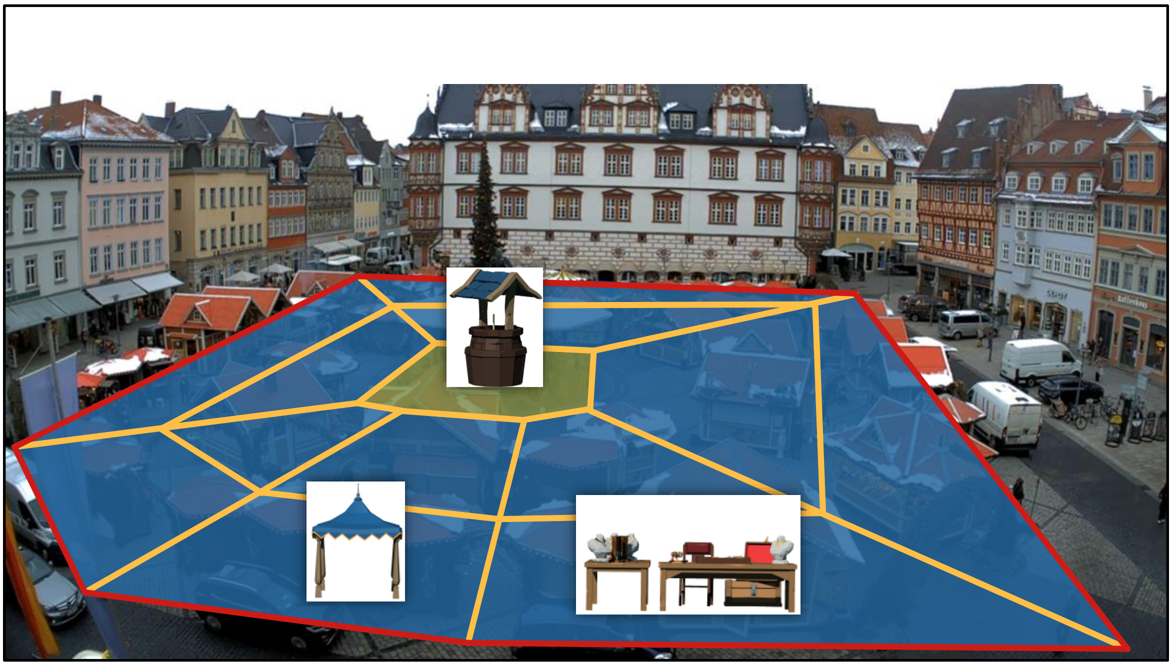


Control parameters do not have to be scalar values: how about a bounding box controlling the generation of a chair. By changing the bounds, we can turn it into a bench or adjust the height of the back support.

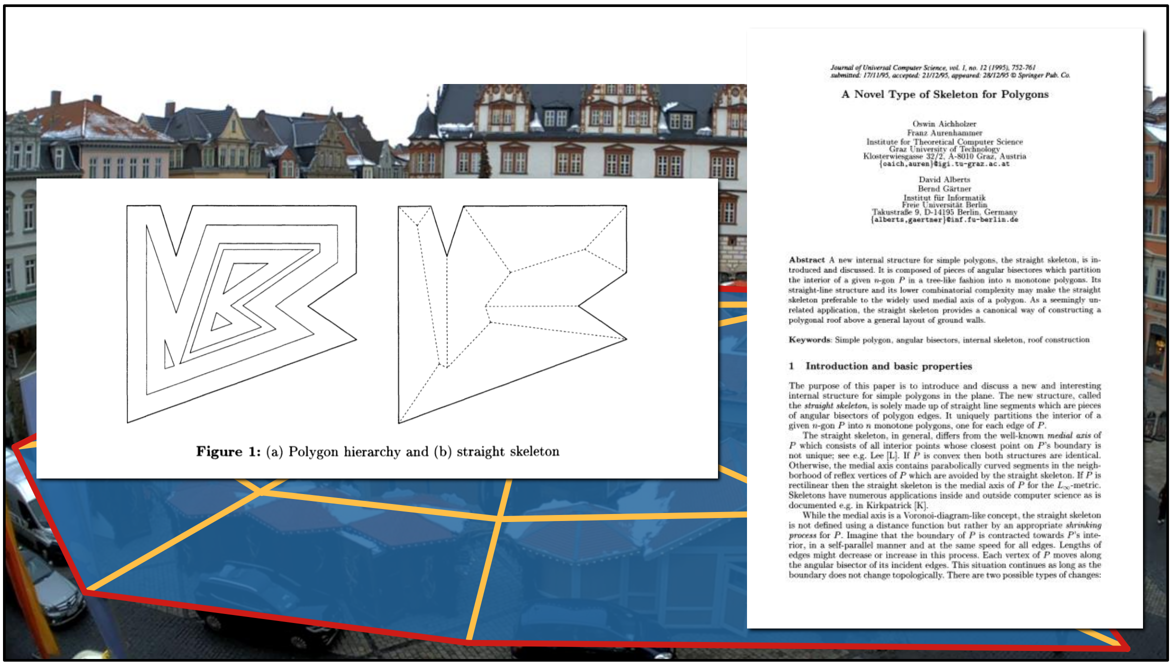


Or what about a polygon controlling the shape of an entire building?

Let‘s start with our first example: a procedural market.



For this, we went on a research trip to the Coburg marketplace and observed the following: the overall shape of it can be described by a polygon. From each corner, a path leads towards the center of the market. These paths are connected by rings of paths. In the regions between the paths, there are the booths. So, we call this the booth islands and should place some fitting assets there like tents or tables. In the center of there market, there is usually a special area with a special asset, like a tree or a well.



This market layout is very close to something called the straight skeleton of a polygon by Aichholzer and coworkers [Aichholzer et al. 1995].

Figure 1(a) and (b) from [Aichholzer et al. 1995].

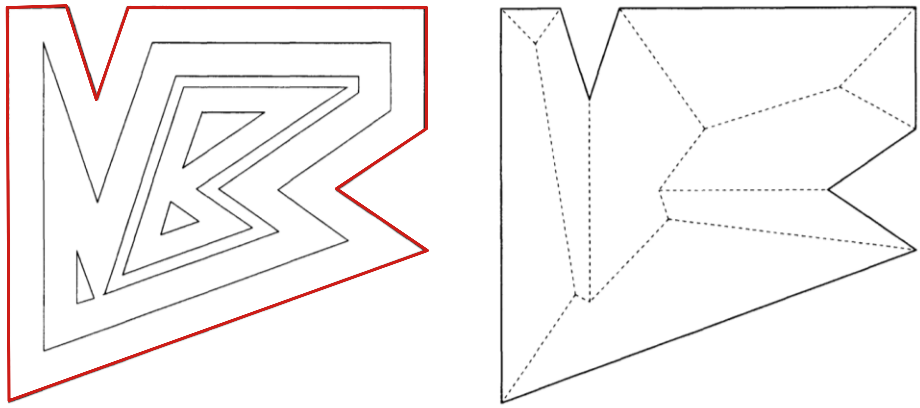
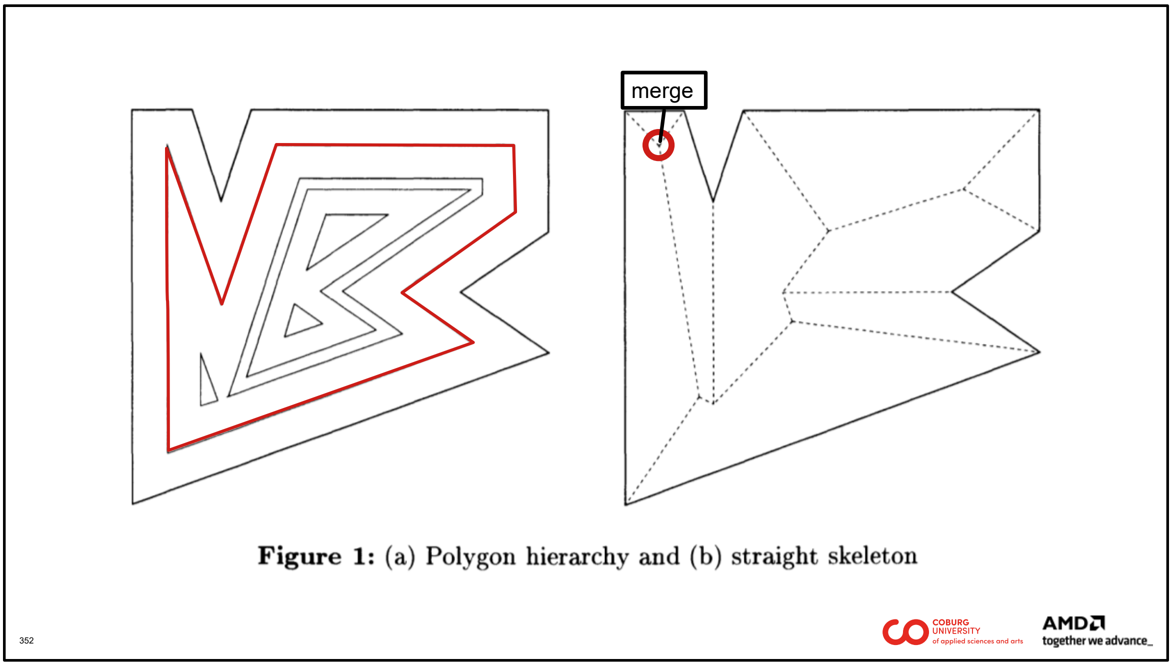


Figure 1: (a) Polygon hierarchy and (b) straight skeleton

For generating it, a polygon is shrunk till one of two possible events occur.



The merge event, where two points of the polygon merge into one.

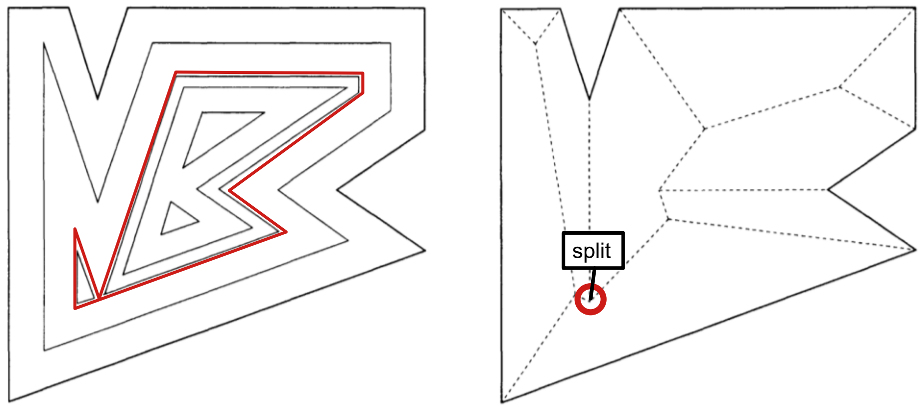
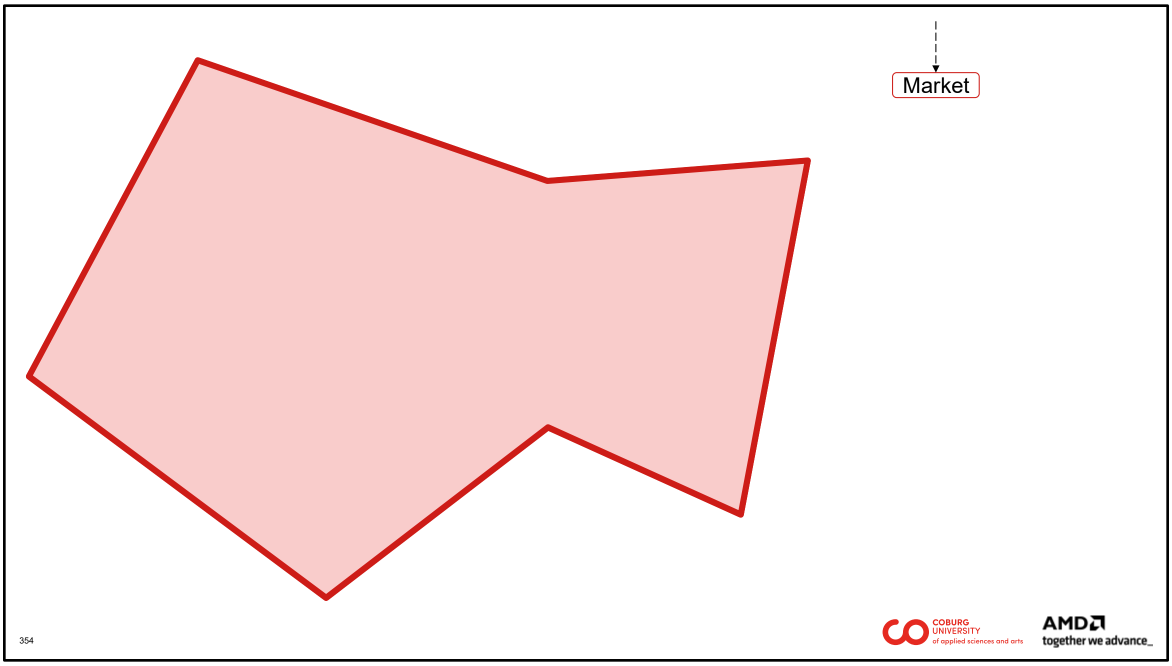


Figure 1: (a) Polygon hierarchy and (b) straight skeleton

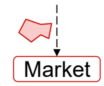
353

And the split event, where the polygon gets split into two.



Now let‘s start with our market generation. A node of a work graph receives a polygon as input.

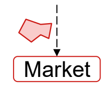
```
static const int maxMarketPoints = 32;
struct MarketRecord {
    float2 points[maxMarketPoints];
    ...
};

[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeDispatchGrid(1, 1, 1)]
[NumThreads(maxMarketPoints, 1, 1)]
...
void Market(
    DispatchNodeInputRecord<MarketRecord> inputRecord,
    uint gtid : SV_GroupThreadId,
    ...

){
```



355

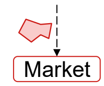
Let‘s look at how this would look like in code:

```
static const int maxMarketPoints = 32;
struct MarketRecord {
    float2 points[maxMarketPoints];
    ...
};

[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeDispatchGrid(1, 1, 1)]
[NumThreads(maxMarketPoints, 1, 1)]
...
void Market(
    DispatchNodeInputRecord<MarketRecord> inputRecord,
    uint gtid : SV_GroupThreadId,
    ...



){
```



356

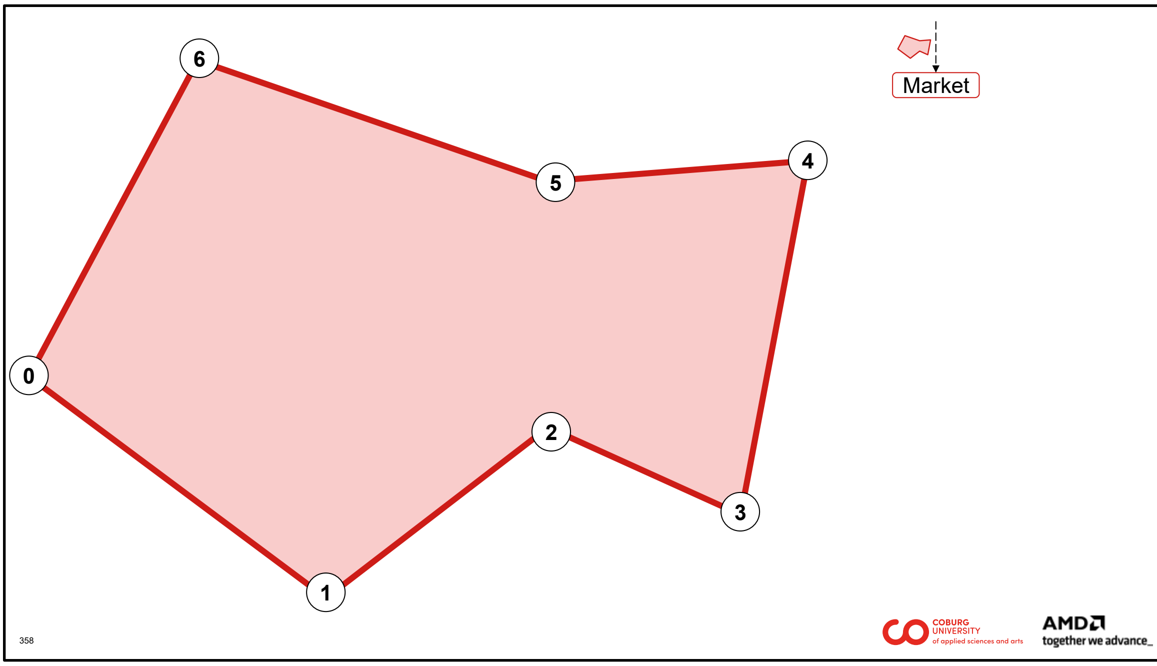
Our market receives a market record as input, consisting of up to 32 points.



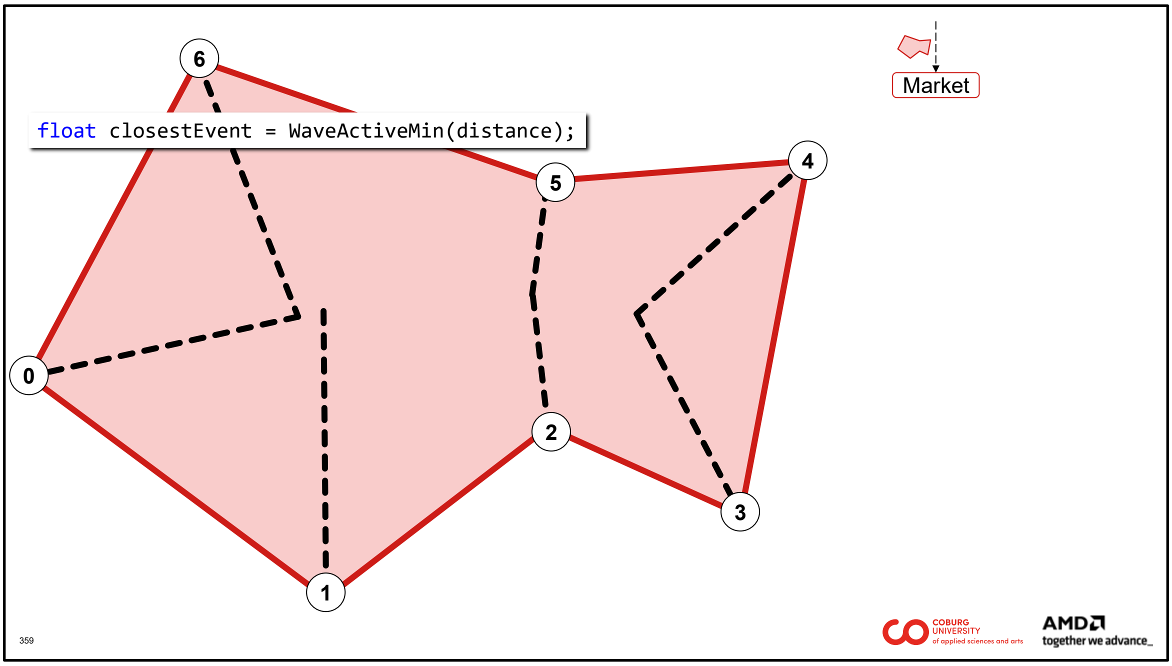

```
static const int maxMarketPoints = 32;
struct MarketRecord {
    float2 points[maxMarketPoints];
    ...
};

[Shader("node")]
[NodeLaunch("broadcasting")]
[NodeDispatchGrid(1, 1, 1)]
[NumThreads(maxMarketPoints, 1, 1)]
...
void Market(
    DispatchNodeInputRecord<MarketRecord> inputRecord,
    uint gtid : SV_GroupThreadId,
    ...





){
```

357

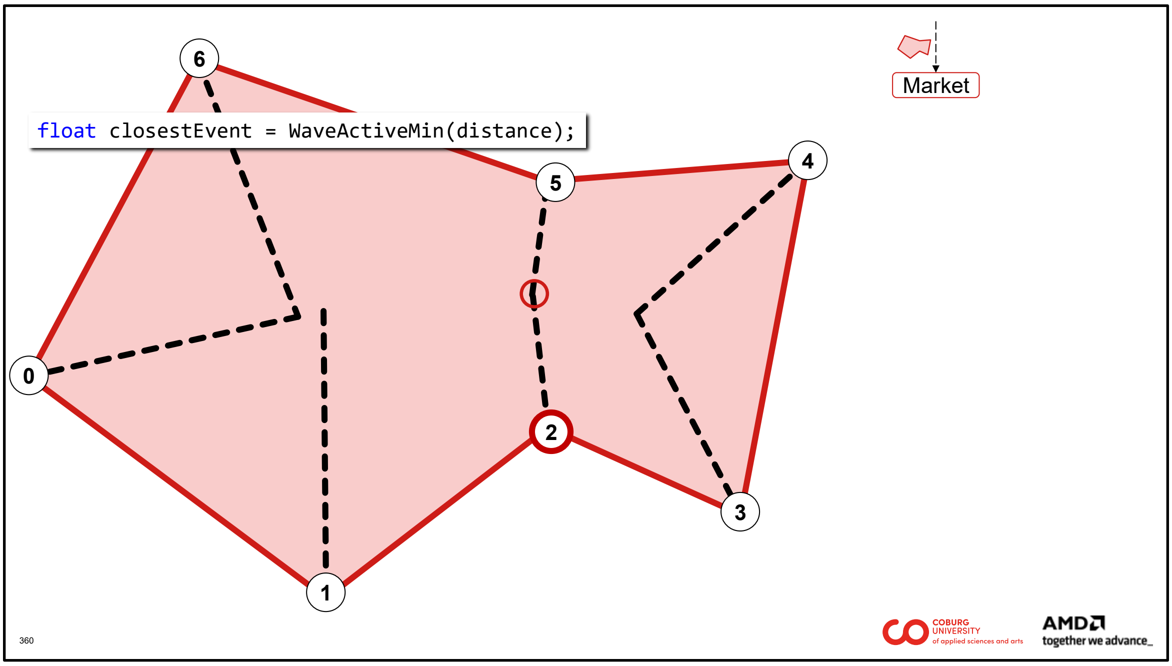
And we launch the market node as one thread group of 32 threads.



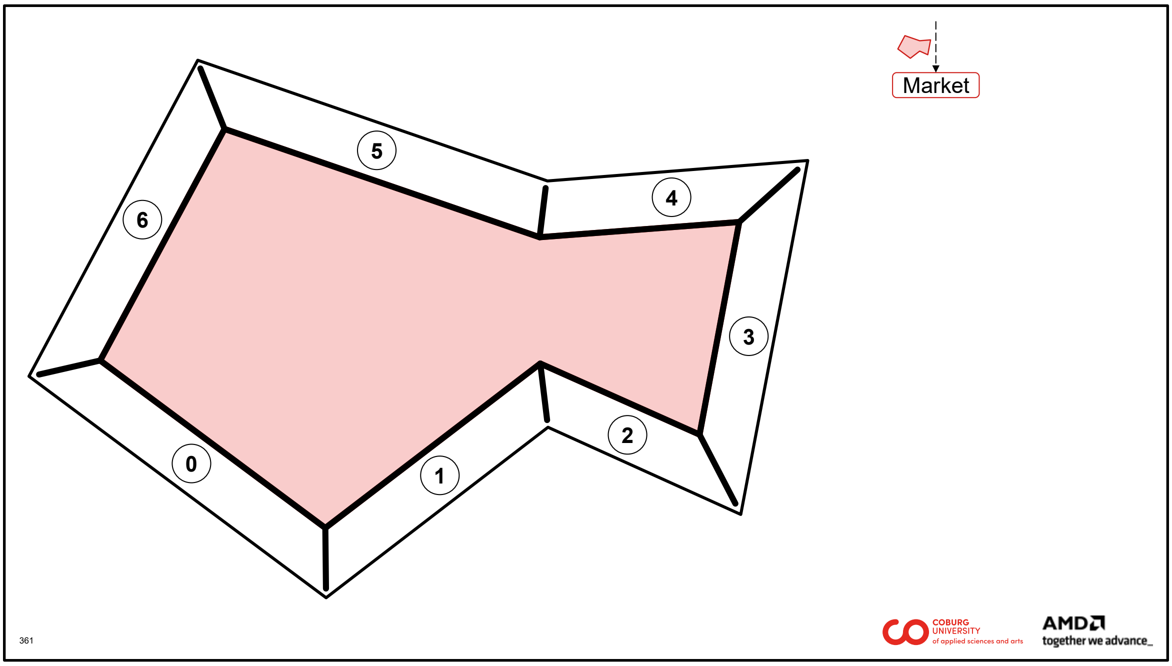
Before shrinking the polygon, we need to check when the next straight skeleton event occurs.



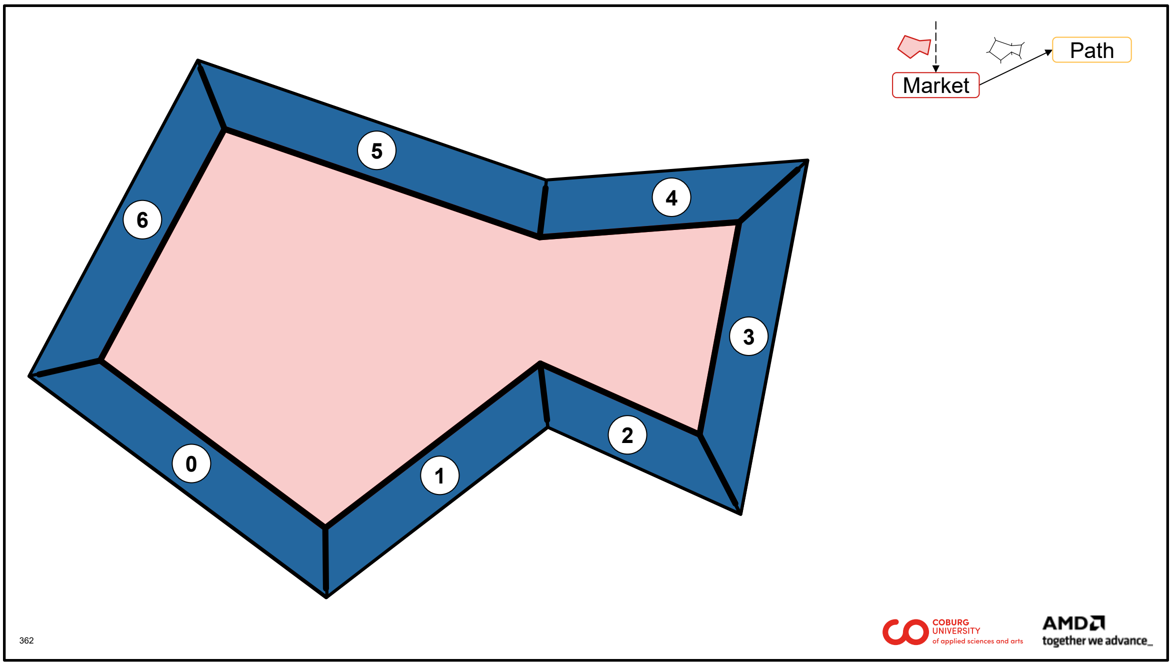
We assign each thread to a corner of the polygon and compute when its event occurs. By using the wave intrinsic `WaveActiveMin`, we can find the closest event.



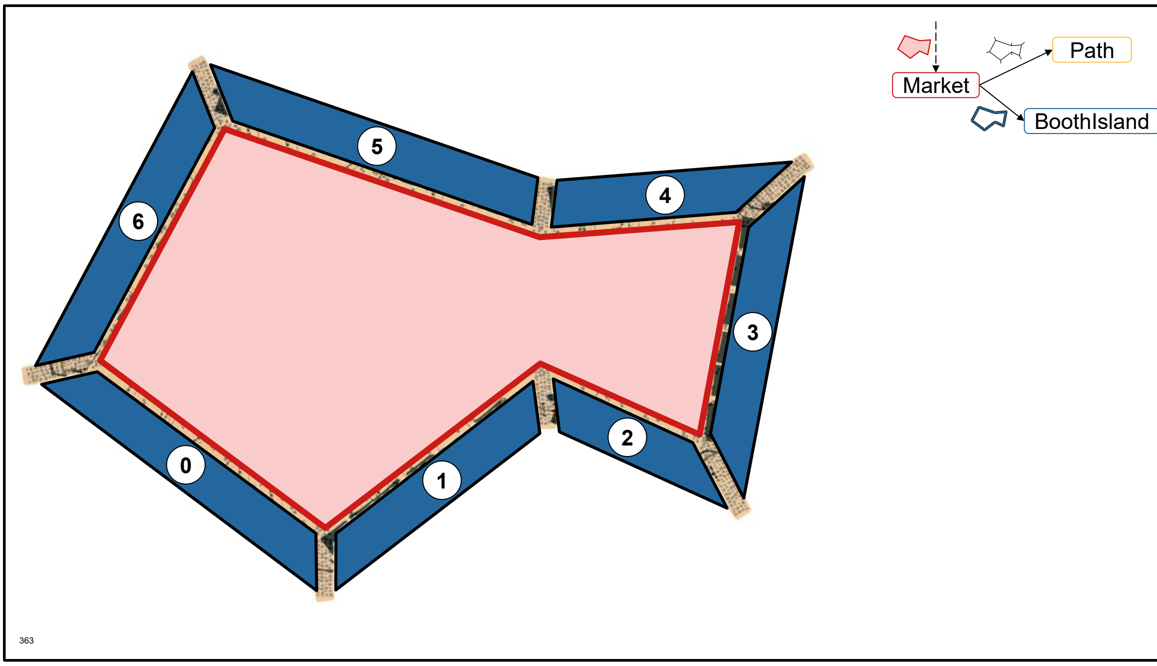
So, in this case it is thread or point 2, but the polygon can still shrink quite a bit before.

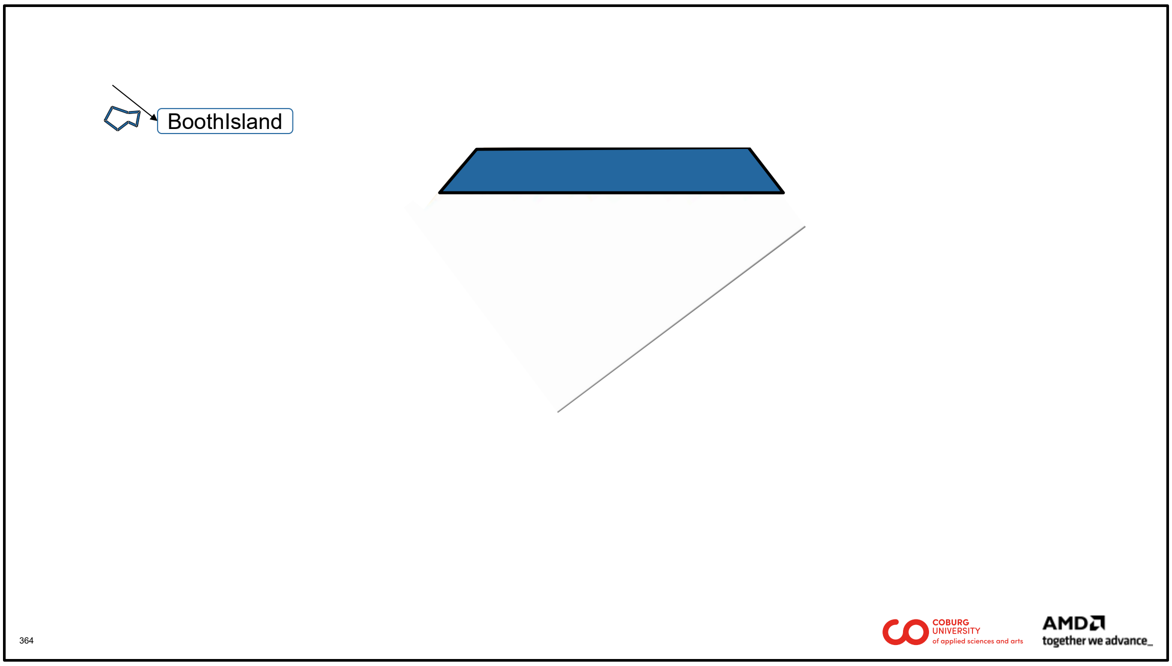


After shrinking, the market node writes output records to a node for drawing paths.

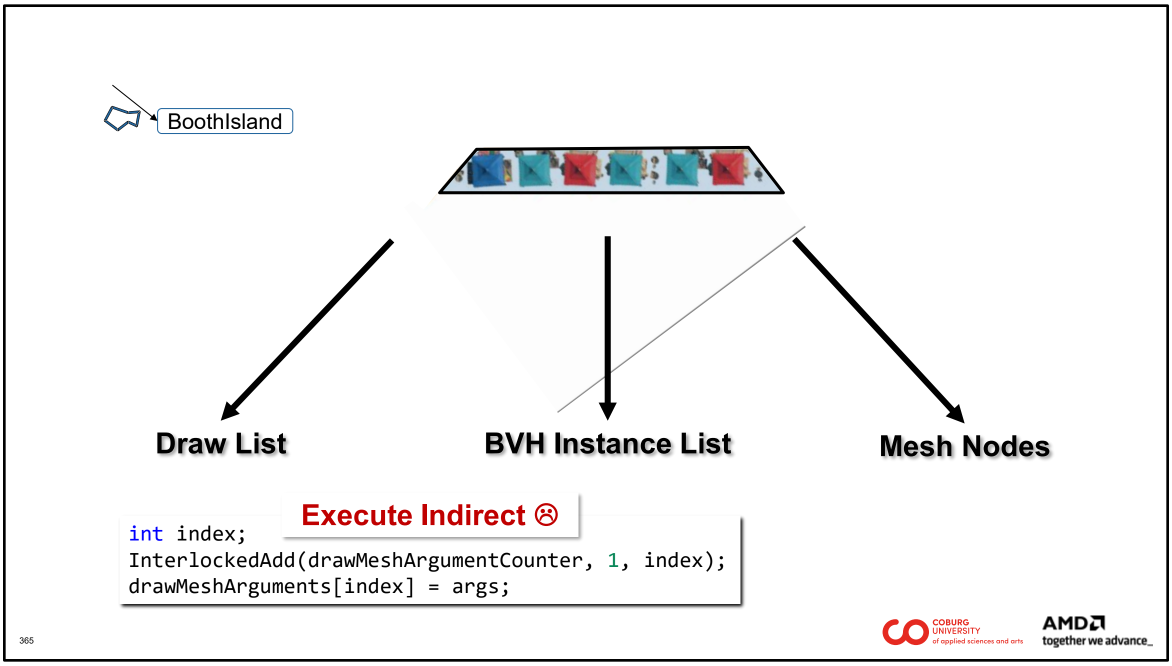


And for the booth islands, we make a little bit of space.


Path
Market
BoothIsland
0
1
2
3
4
5
6
363

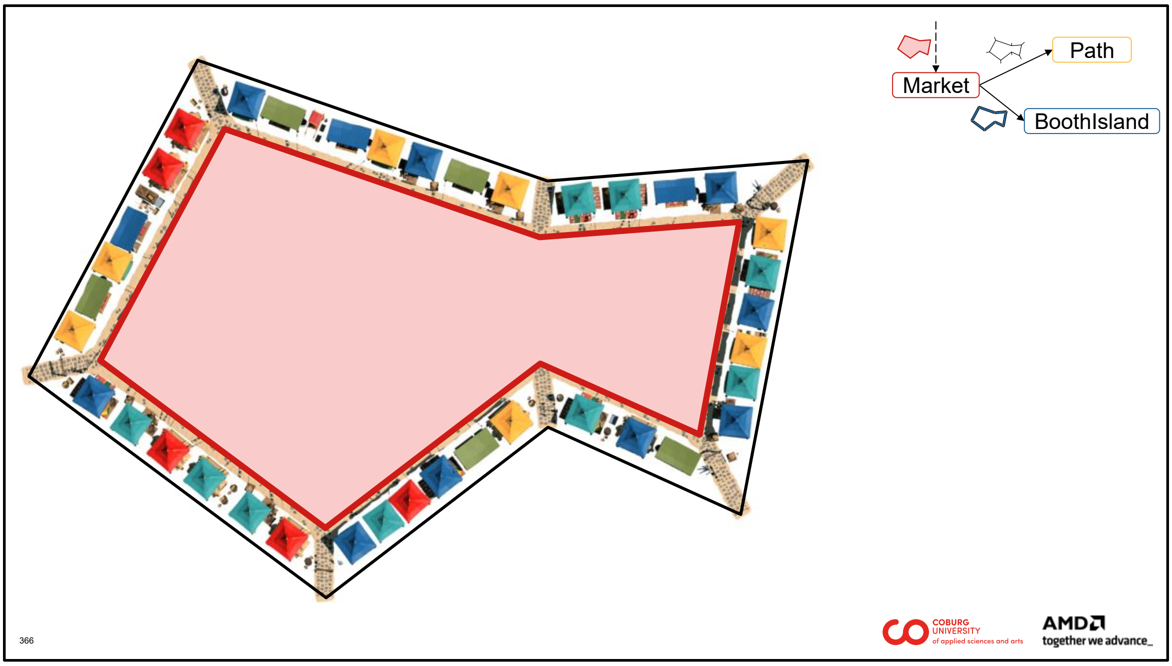


Next, let‘s look at how a work graph can output geometry for drawing.

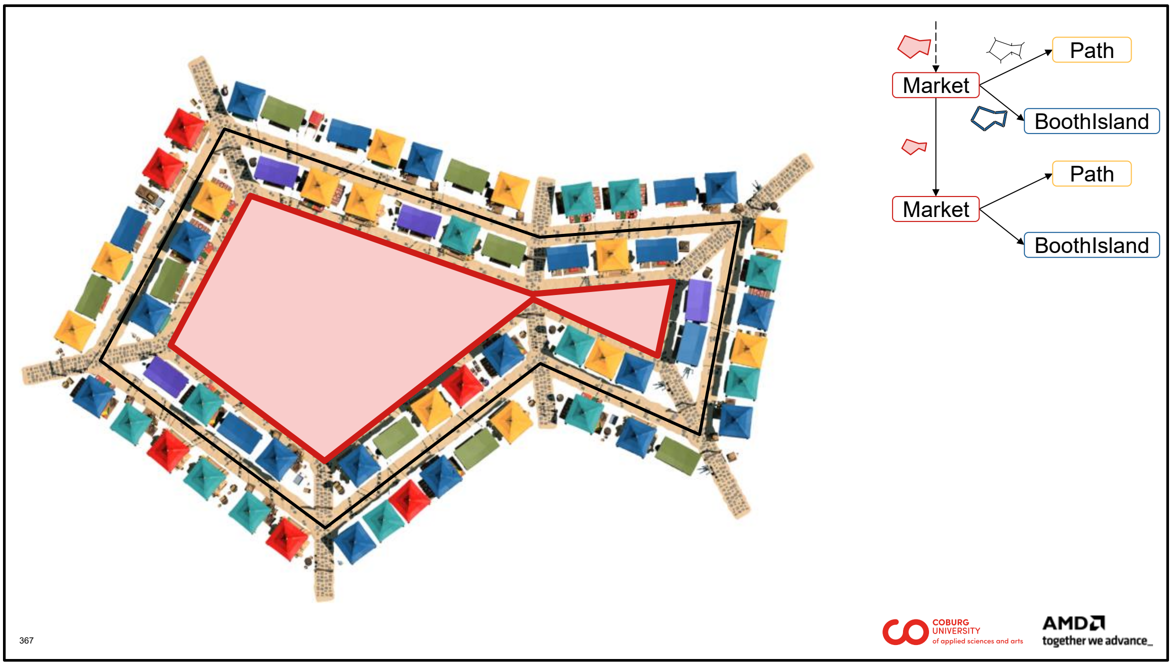


One way would be to append a draw command to a draw list and then dispatch that list after the work graph has finished using execute indirect. To allow for ray-tracing, one can also write to an instance list and then build a TLAS from it after the work graph has finished.

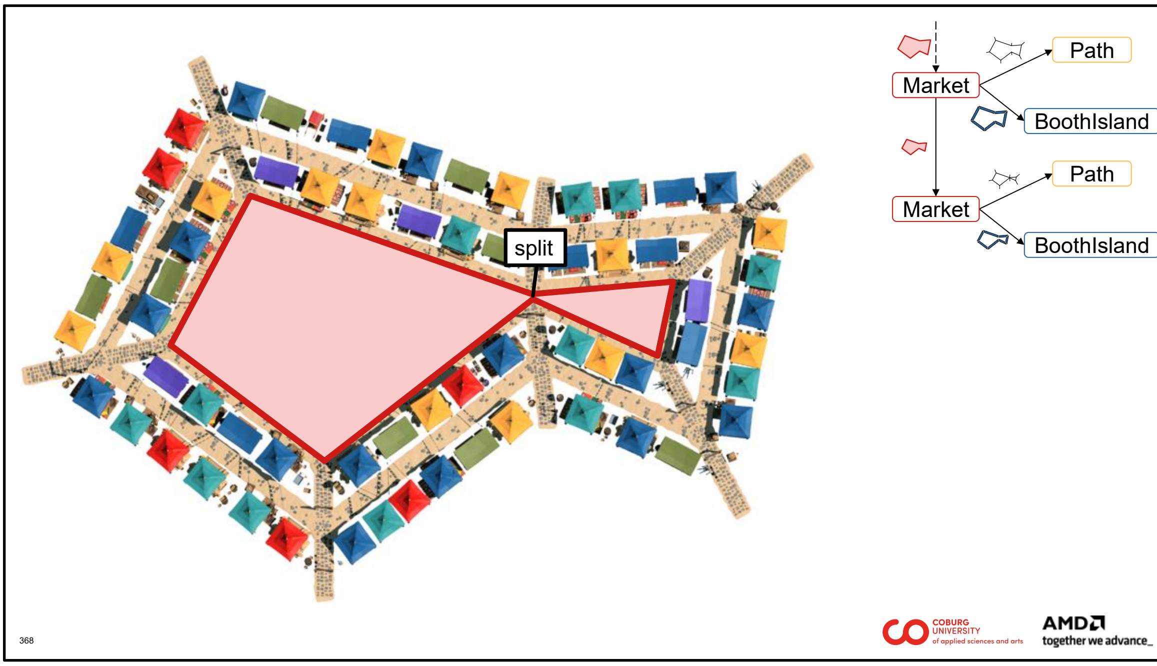
Finally, mesh nodes can draw the generated geometry straight from the work graph to the scene.



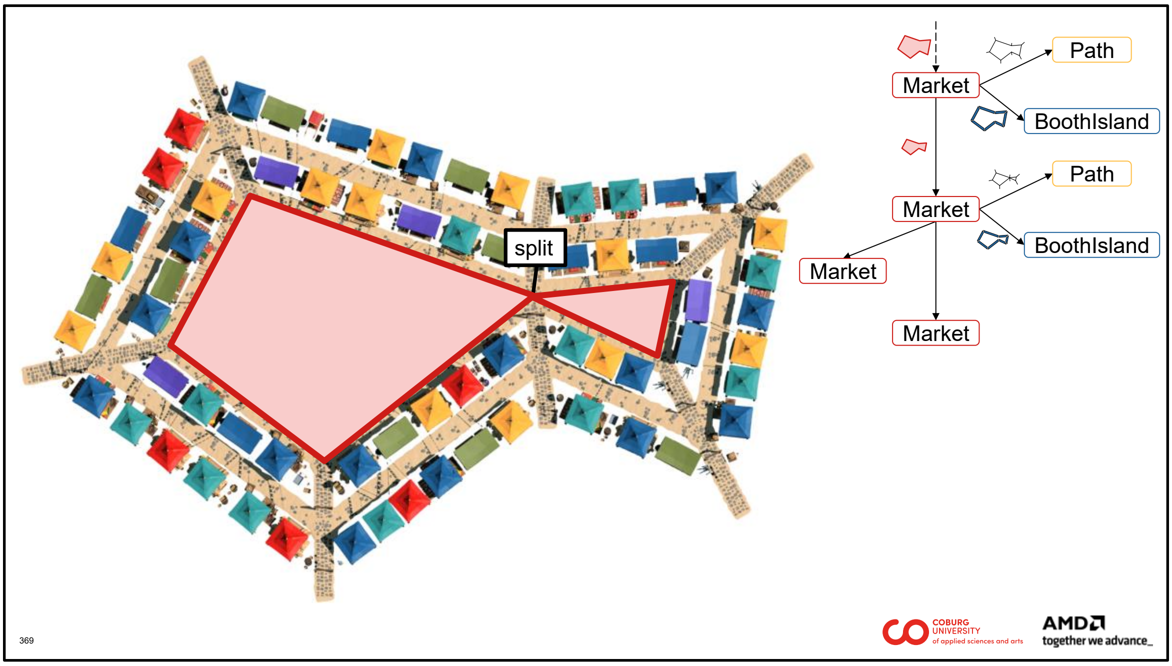
Alright, let‘s get back to our market, where we have just finished one ring. To do the next ring, the market node simply recurses with the new polygon.



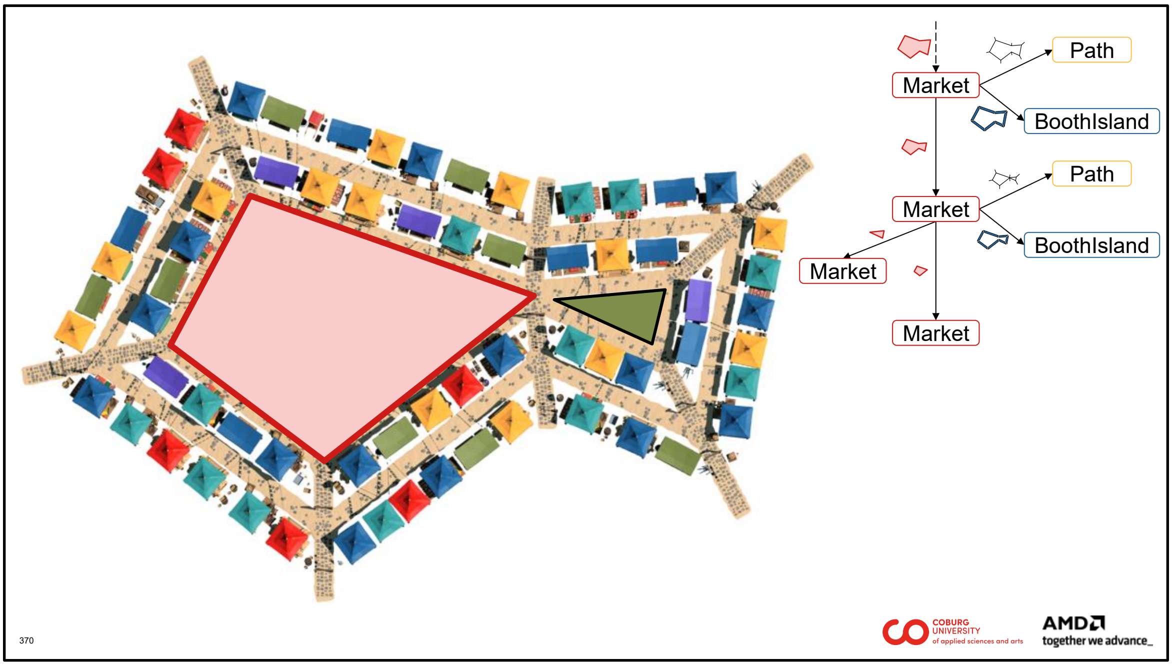
For this ring, we do the same as for the last.



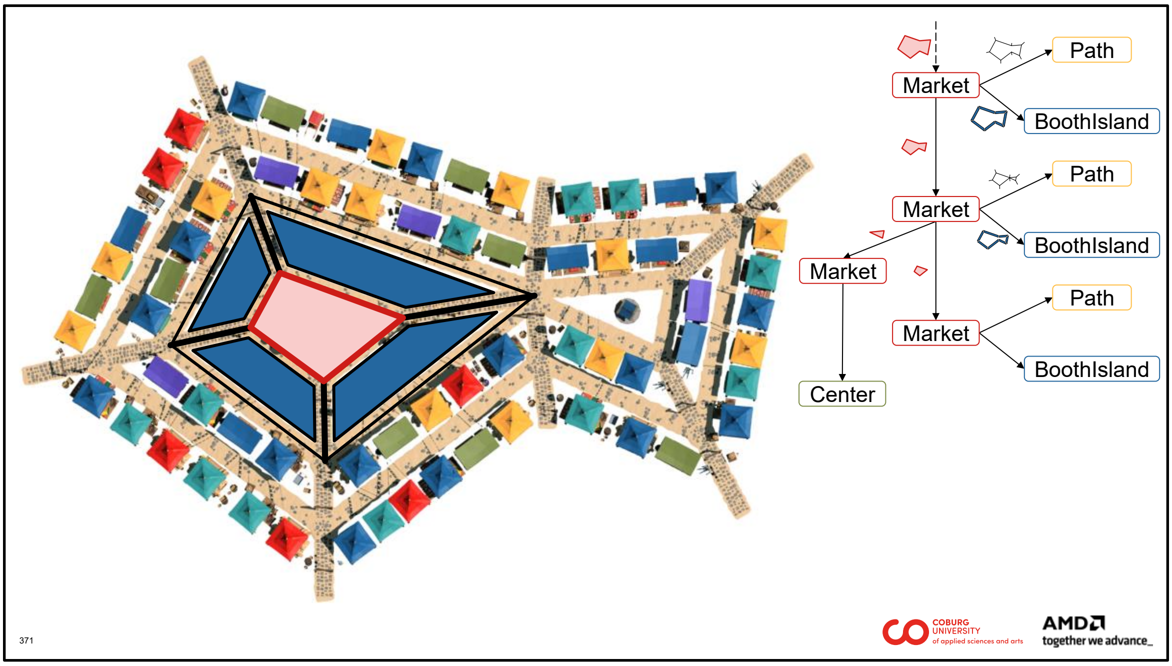
Now we must handle our first event: the polygon splits into two if we continue shrinking.



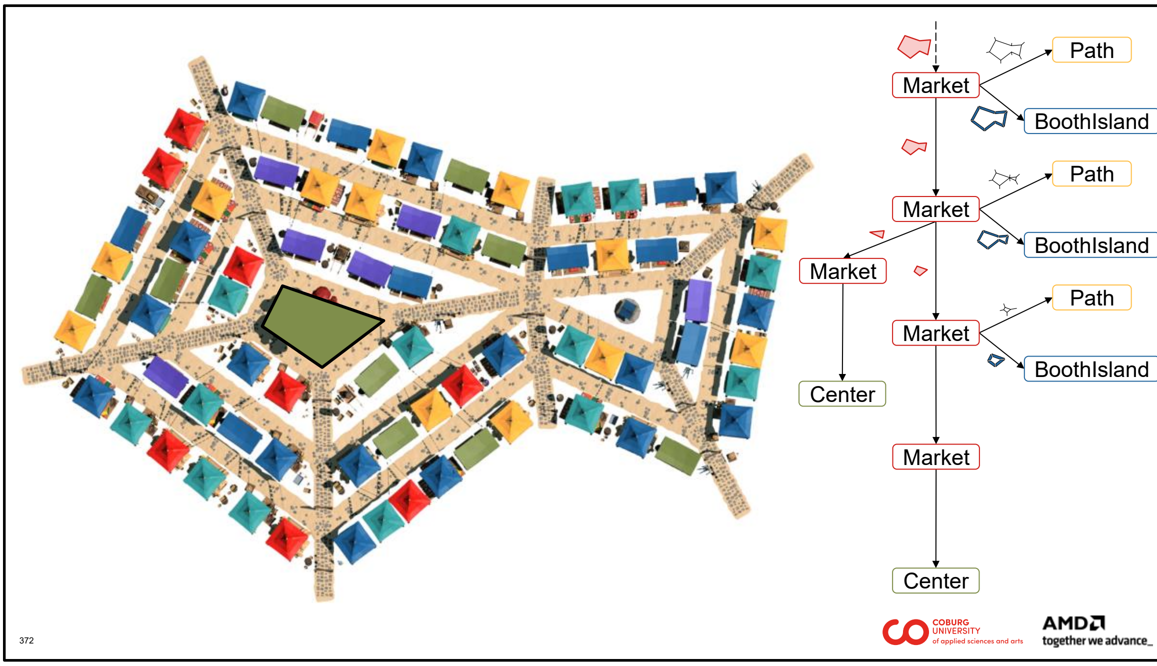
To resolve this, the market node recurses into two markets



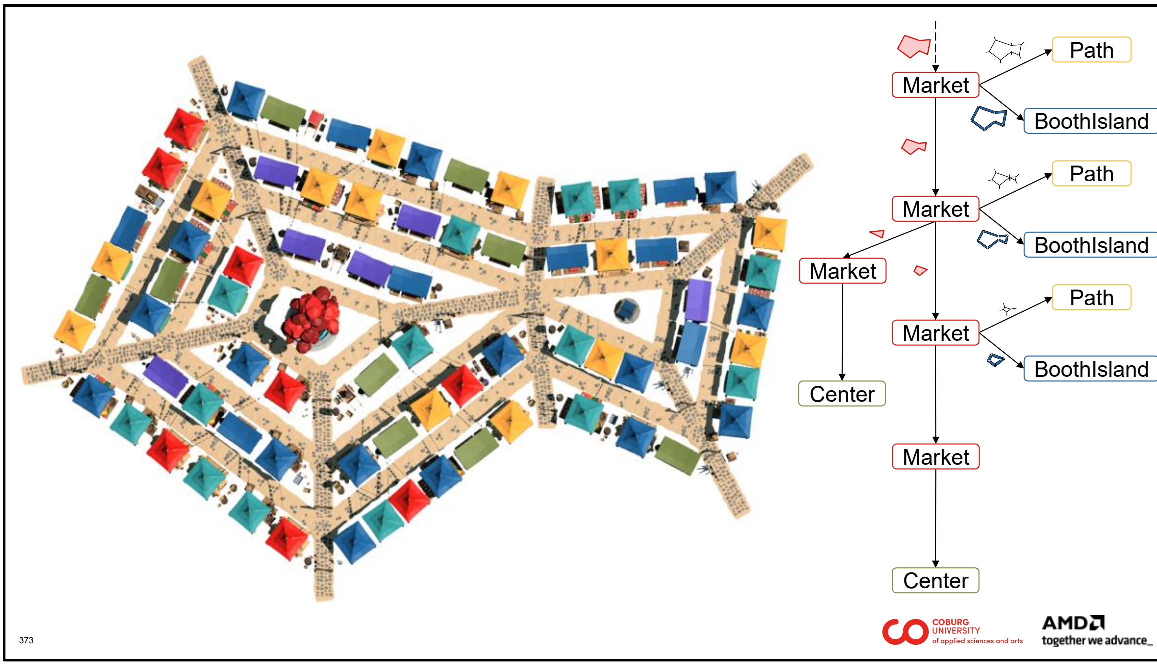
For the smaller side, we do not have enough space for another ring and finish with a market center.



For the other side, we can generate one more ring.

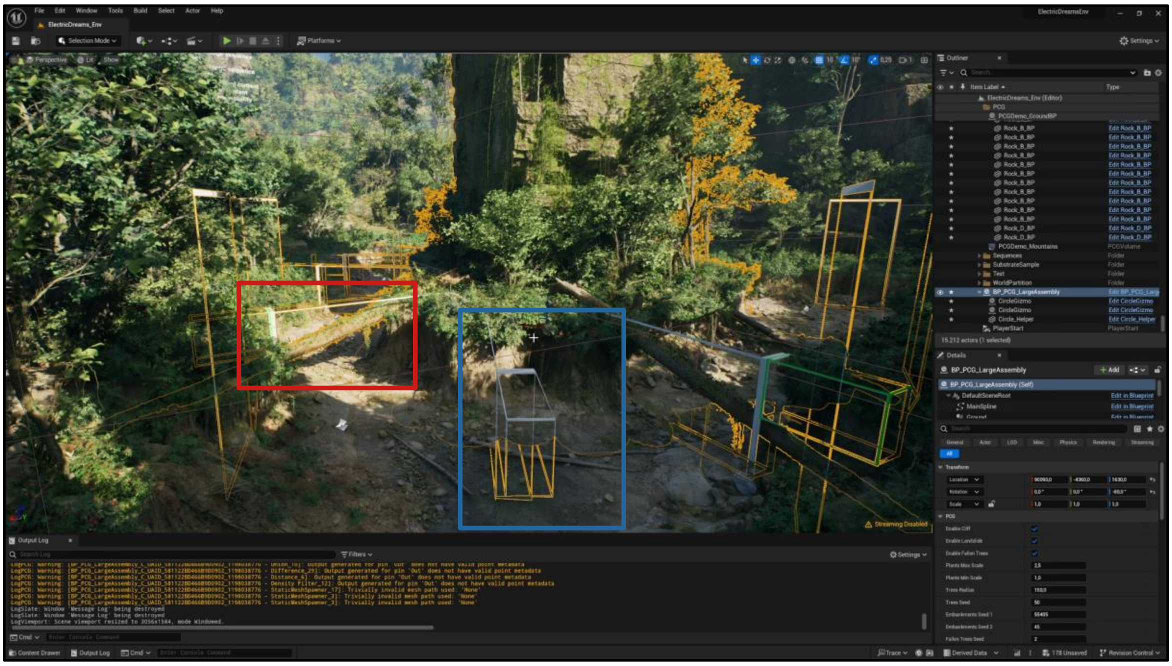


And finish with a market center.

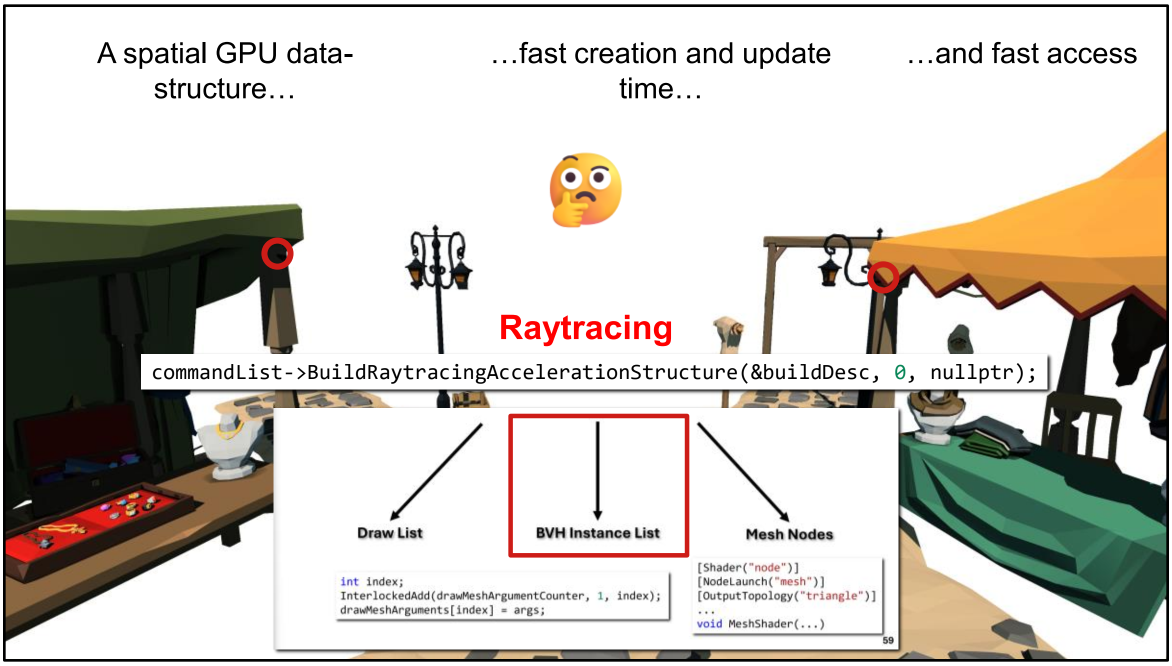

Path
Market
BoothIsland
Path
Market
BoothIsland
Market
Path
Market
BoothIsland
Center
Market
Center
373
COBURG UNIVERSITY of applied sciences and arts
AMD
together we advance_

And with this, we have finished out market generation. Let's see it in action. Because it runs every frame in less than a millisecond, we can see the changes instantly.

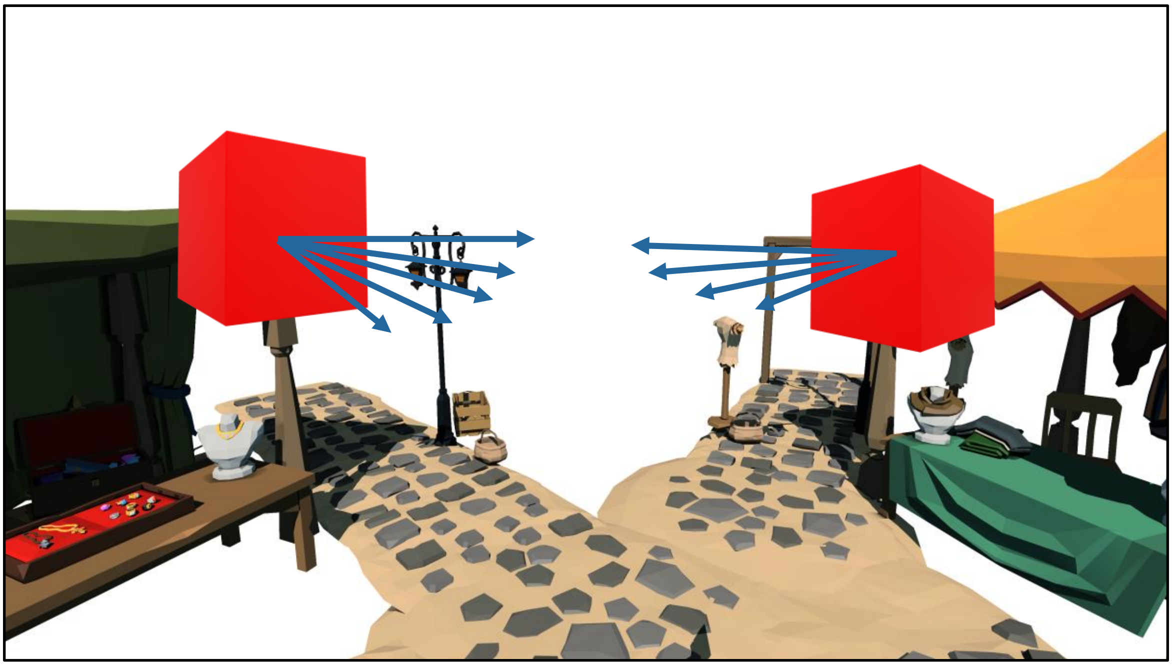
There is one thing, we have not mentioned, yet: You might have spotted these garlands spanning in-between rings. But these are generated independently of each other, so how do we find the connecting points?



This is something we call *dependent generation*. Here is an example of it from Unreal Engine. In the video, you can see the user dragging around the central structure. When the structure marked with the red box instersects with something, a bridge made of a stam is generated towards the center. The structure in the blue box does not intersect with anything and thus no bridge is generated.

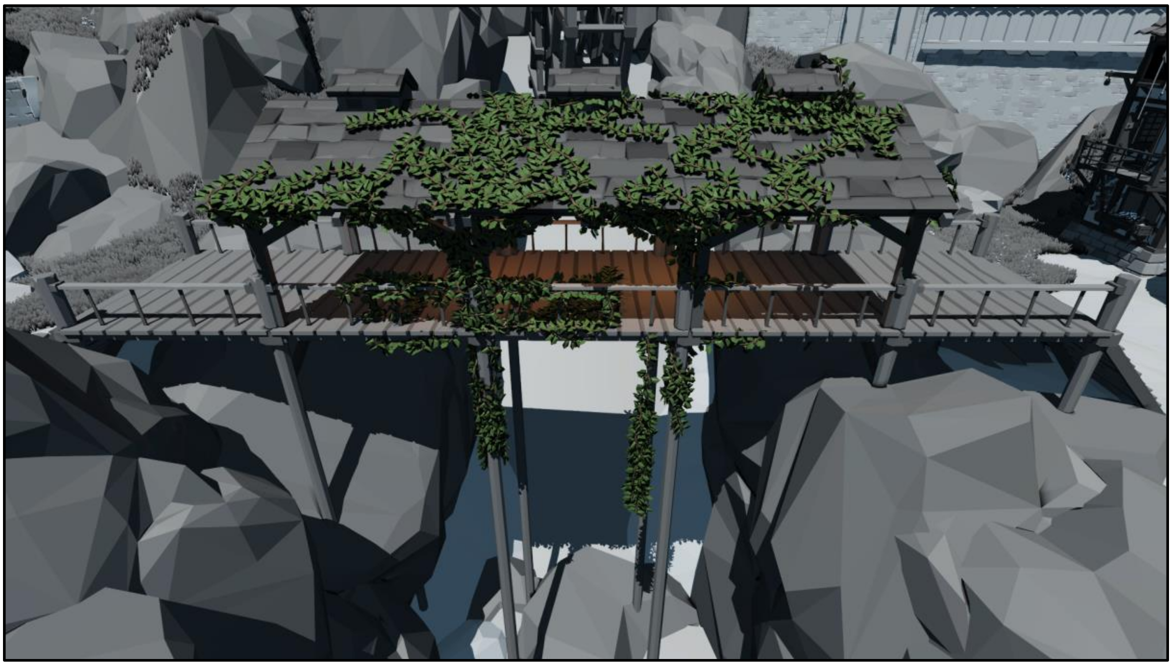


So, for our system, we need a spatial GPU data structure that is fast to create and update and fast to access. This is exactly what a ray tracing BVH is for. Creating and accessing it is just a matter of issuing API calls, and we have already established earlier that we can output for BVH generation.

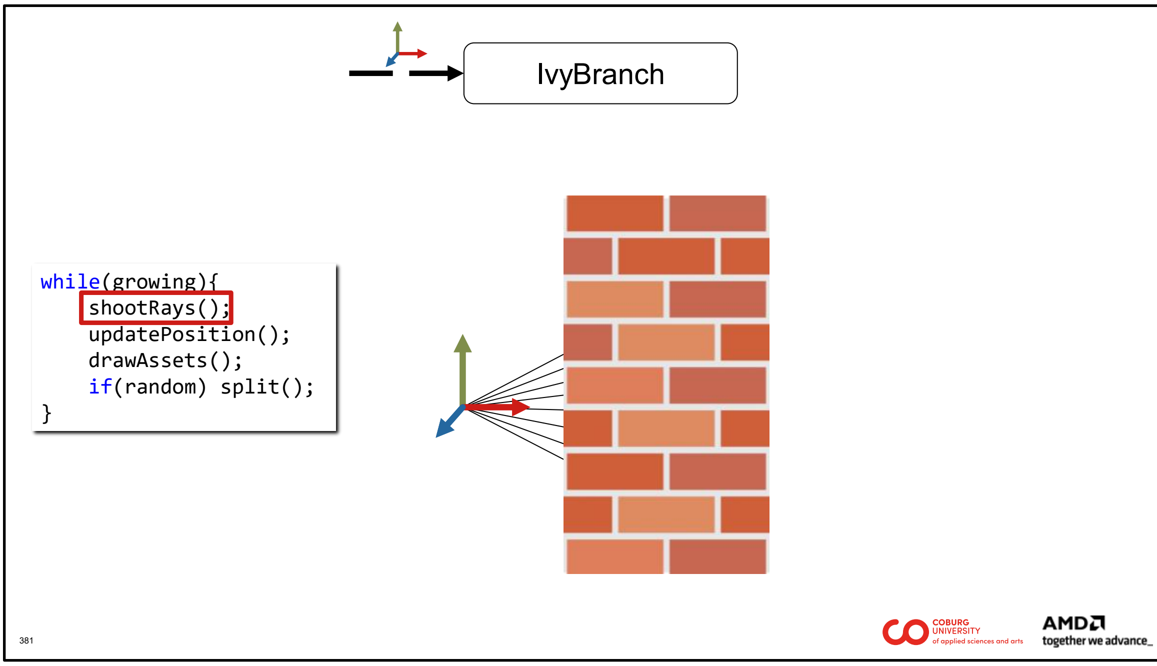


During generation, we add these red bounding boxed to our BVH with a separate instance flag to prevent hitting them when ray-tracing for shading effects. Next a garland starting points shoots rays into its vicinity to find points to connect to.

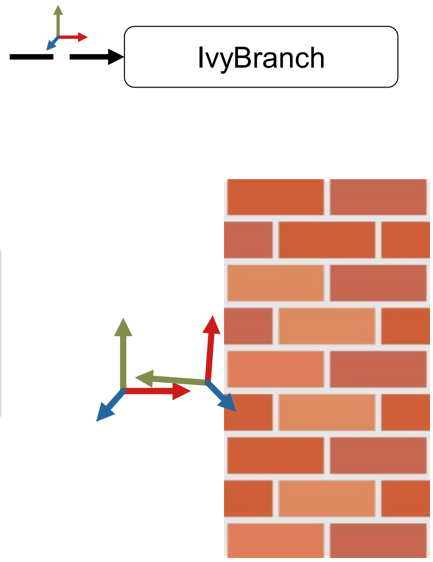
And with this, we can have garlands from market elements generated independently from each other.



Let‘s look at another example for dependent generation: Ivy ontop of existing geometry.

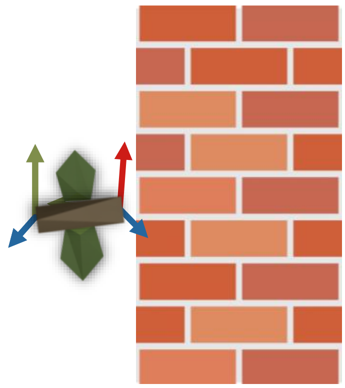


An *IvyBranch* node is given a transformation as the input record. For growing, it runs a loop that shoots rays into its vicinity to find a surface, updates the transformation based on the result, and draws fitting assets like leaves and a stem. Finally, there is a chance that the ivy branches into two which we solve with recursion.

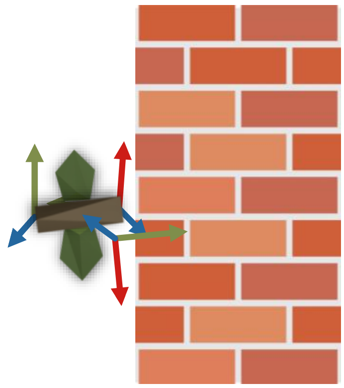



```
while(growing){
    shootRays();
    updatePosition();
    drawAssets();
    if(random) split();
}
```

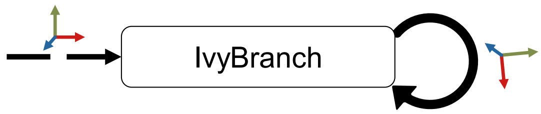

```
while(growing){
    shootRays();
    updatePosition();
    drawAssets();
    if(random) split();
}
```



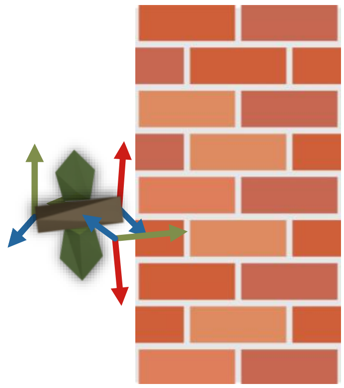

```
while(growing){
    shootRays();
    updatePosition();
    drawAssets();
    if(random) split();
}
```



384

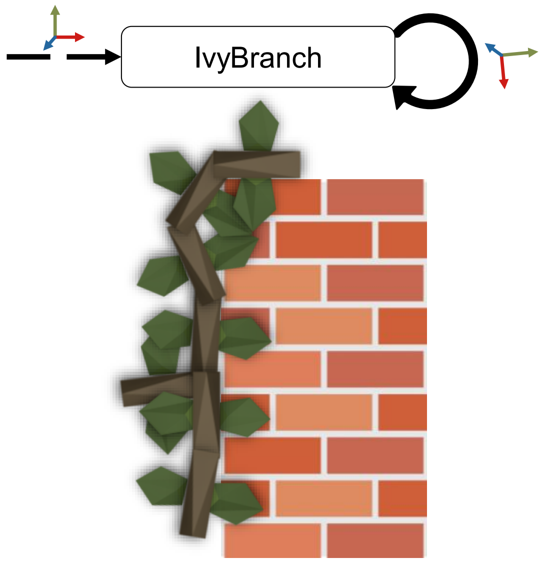



```
while(growing){
    shootRays();
    updatePosition();
    drawAssets();
    if(random) split();
}
```



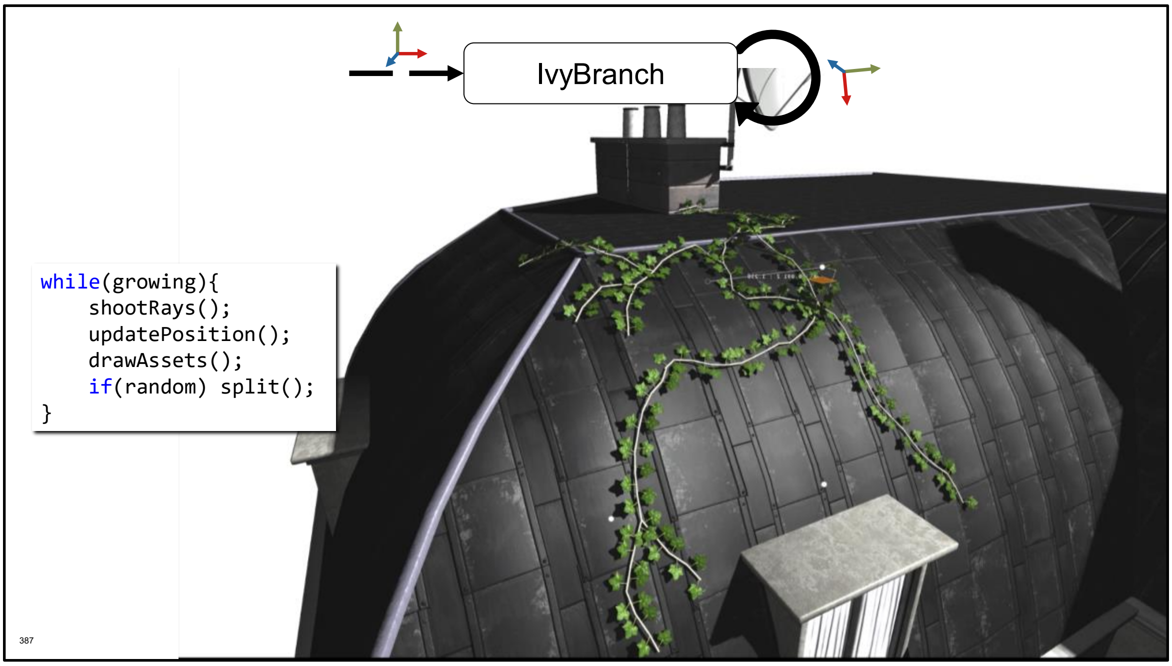
385

COBURG UNIVERSITY of applied sciences and arts
AMD together we advance_

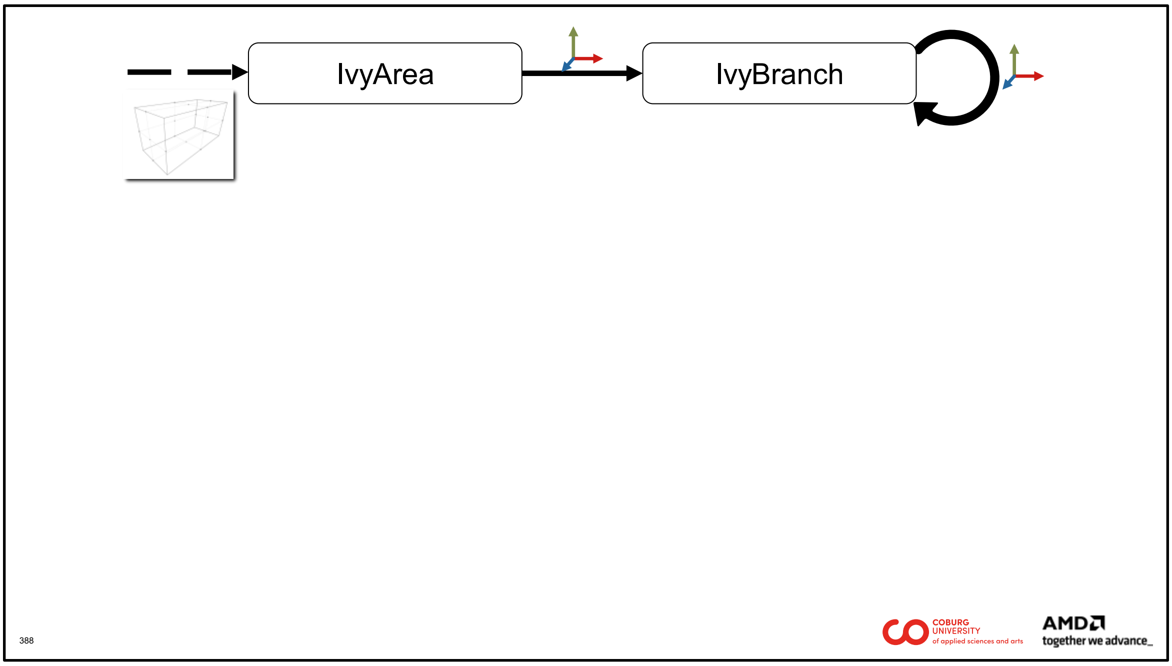



```
while(growing){
    shootRays();
    updatePosition();
    drawAssets();
    if(random) split();
}
```

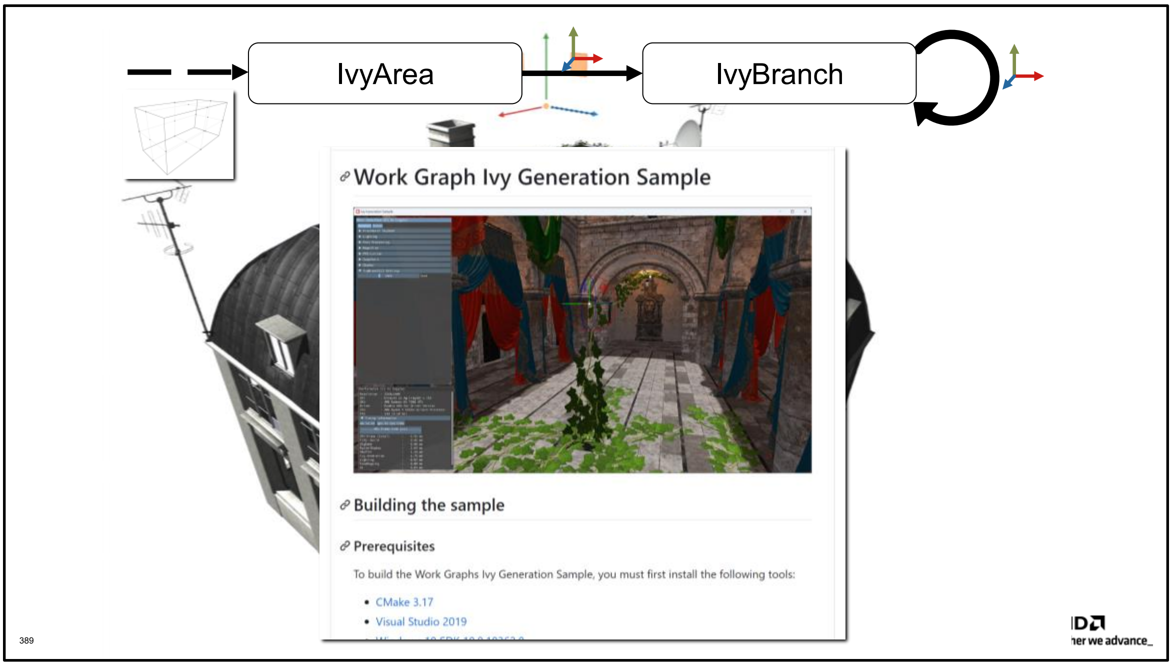
386



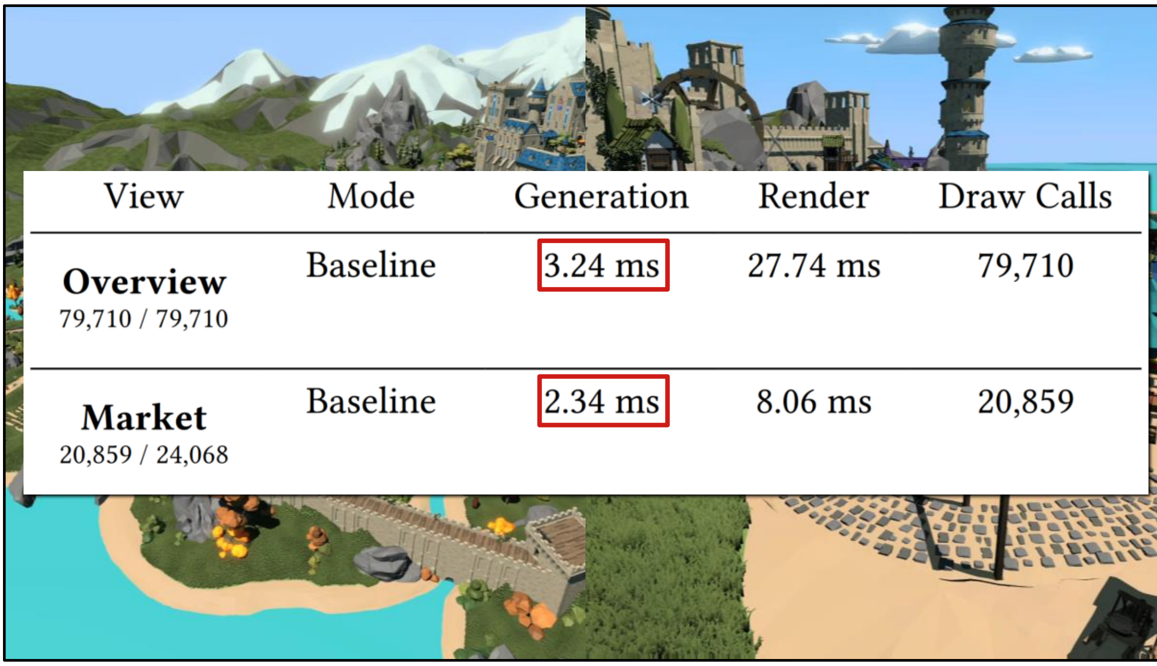
Here you can see an example video of this for more realistic assets.



Let‘s extend on this idea and add a parent to *IvyBranch*, the *IvyArea.* It receives a bounding volume as input, uses rays to find fitting starting locations for ivy to grow and then outputs work records to *IvyBranch*.

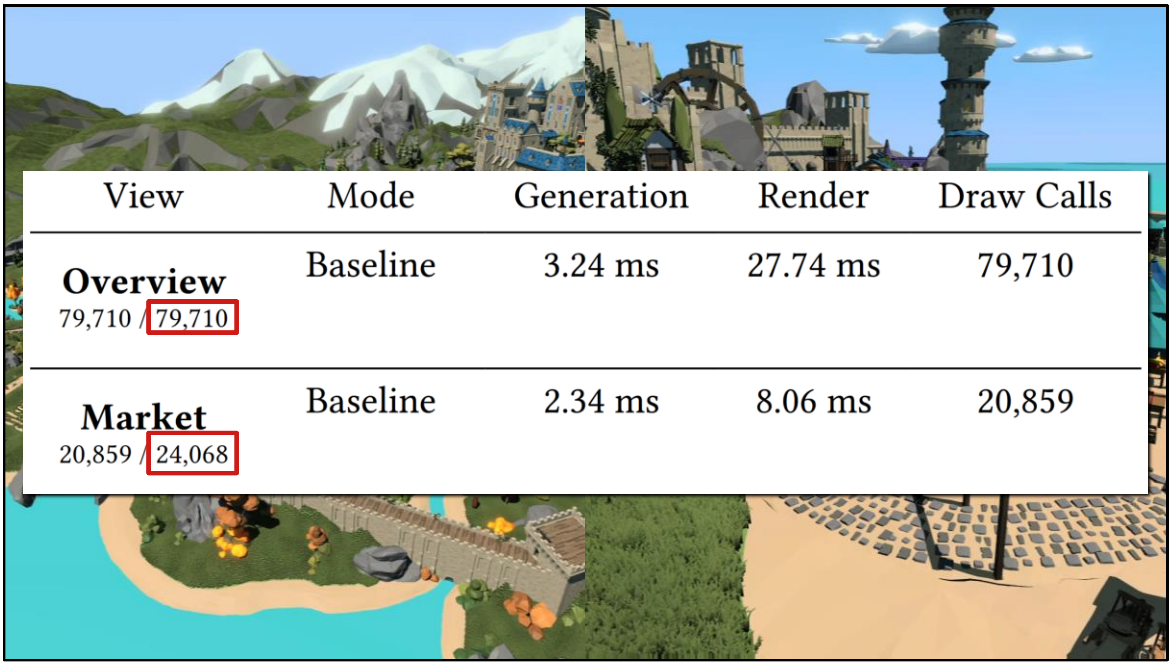


Here you can see an example video of this. We have published a sample of this if you want to play around with the generation yourself.



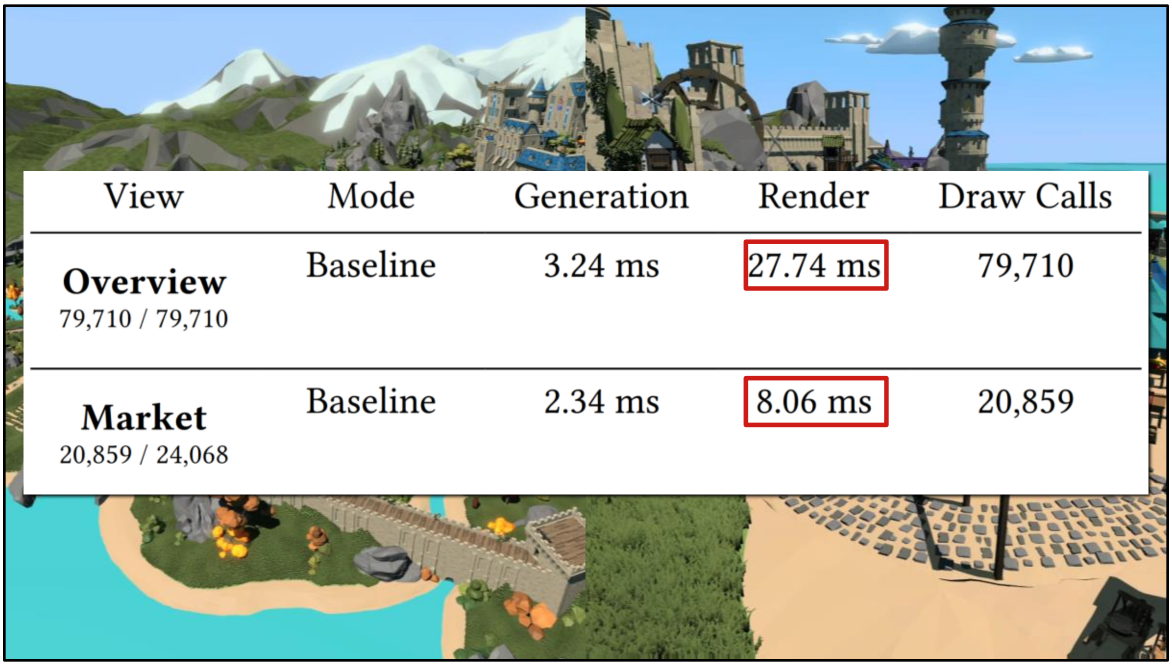
View	Mode	Generation	Render	Draw Calls
Overview 79,710 / 79,710	Baseline	3.24 ms	27.74 ms	79,710
Market 20,859 / 24,068	Baseline	2.34 ms	8.06 ms	20,859

Let‘s talk about timings. We generate our scene from two perspectives, an overview where we generate everything, and a view from the market only, where we can cull some of the generation.



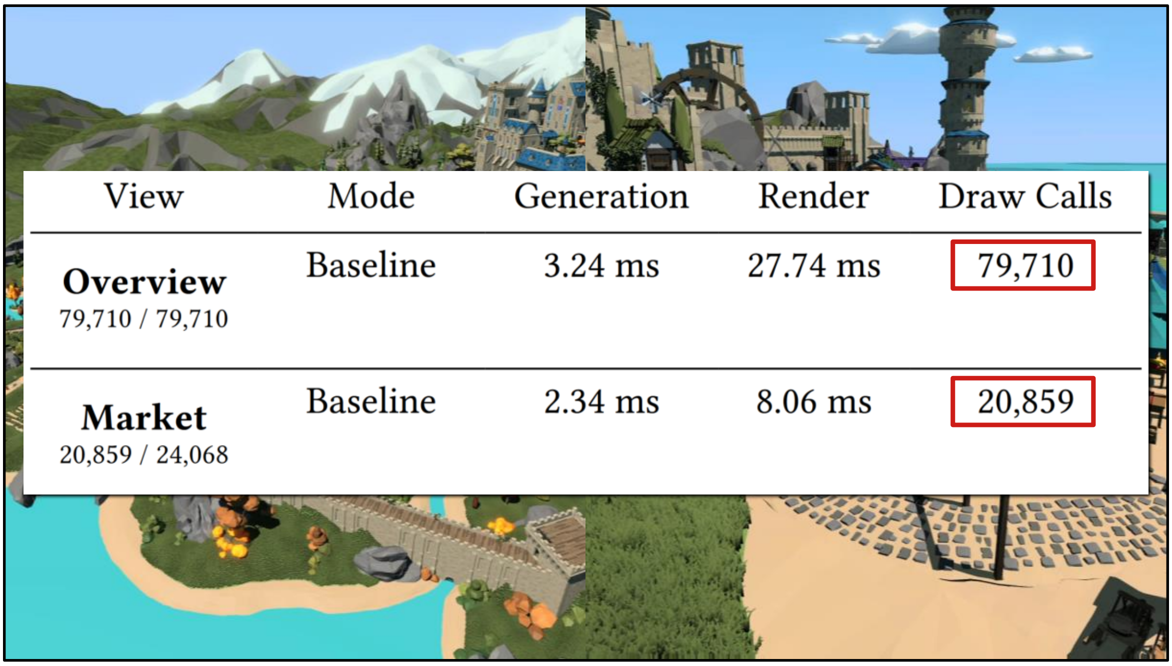
View	Mode	Generation	Render	Draw Calls
Overview 79,710 / 79,710	Baseline	3.24 ms	27.74 ms	79,710
Market 20,859 / 24,068	Baseline	2.34 ms	8.06 ms	20,859

Here you can see the number of instances generated.



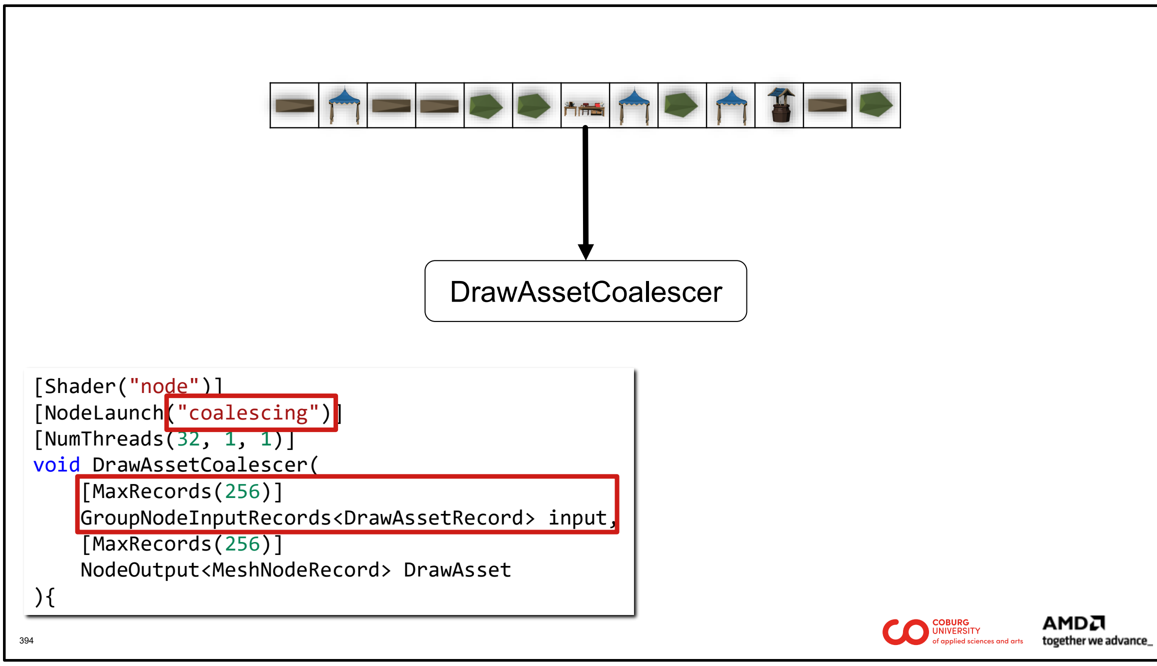
View	Mode	Generation	Render	Draw Calls
Overview 79,710 / 79,710	Baseline	3.24 ms	27.74 ms	79,710
Market 20,859 / 24,068	Baseline	2.34 ms	8.06 ms	20,859

As you can see, right now, the render timings go through the roof for the overview perspective.

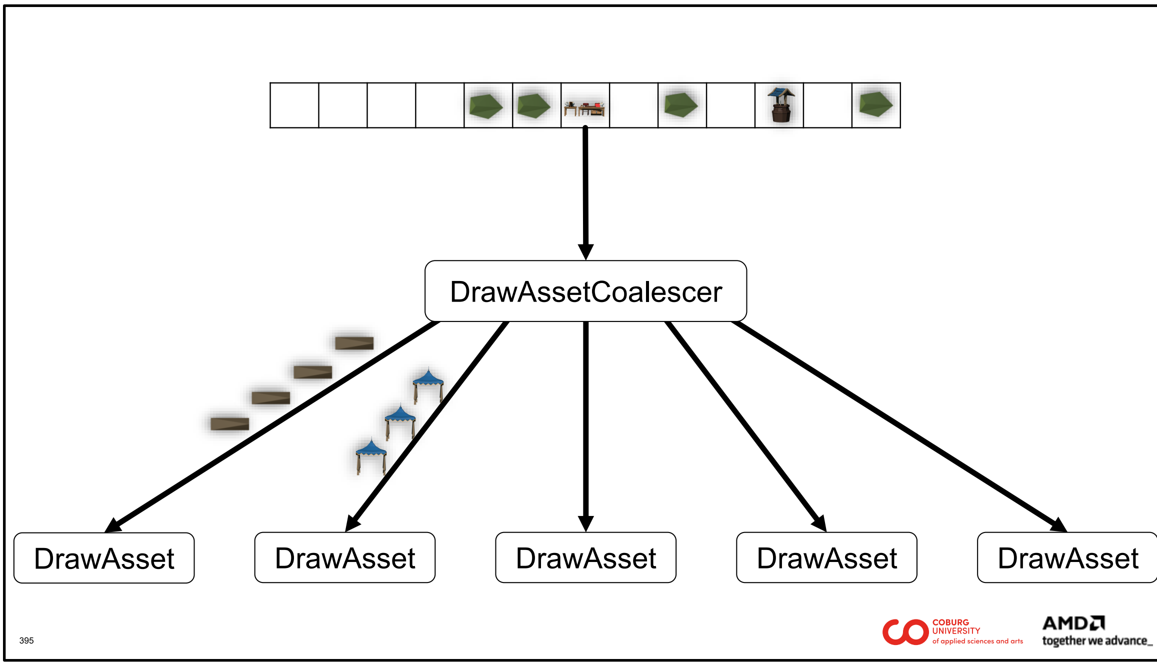


View	Mode	Generation	Render	Draw Calls
Overview 79,710 / 79,710	Baseline	3.24 ms	27.74 ms	79,710
Market 20,859 / 24,068	Baseline	2.34 ms	8.06 ms	20,859

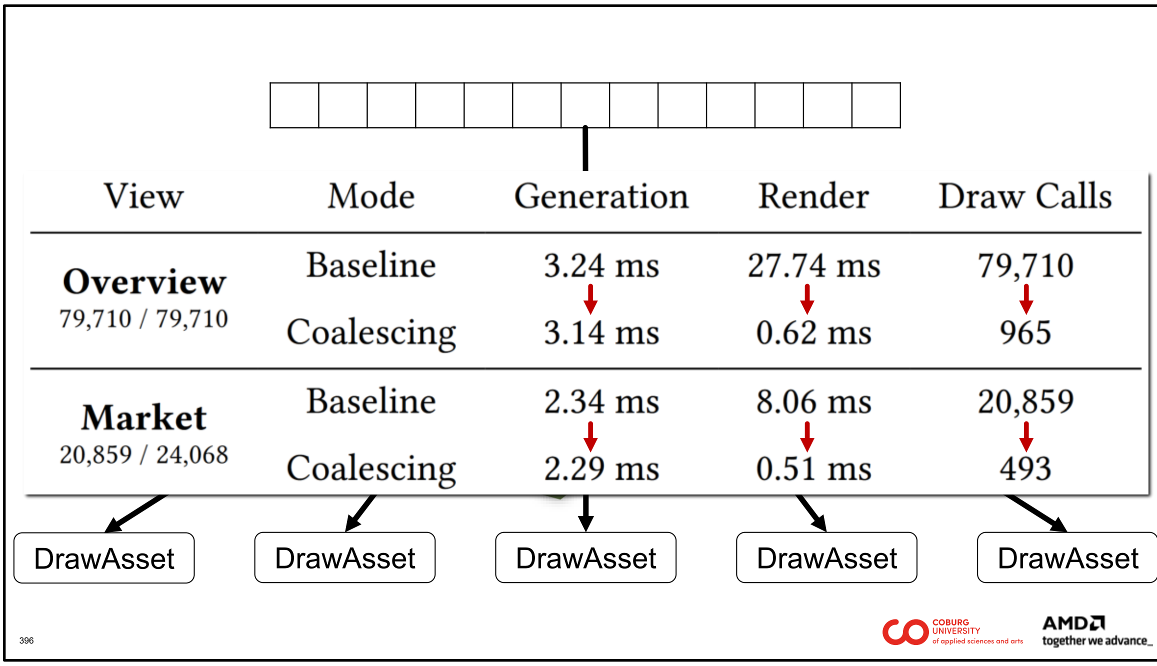
The reason for this can be seen on the right. We have one draw call per instance. We need instancing to optimize this.



For this, we utilize a node in *coalescing* launch mode. It receives up to 256 records for drawing an asset and output up to 256 records. But ideally, we are able to combine some of these using instancing.



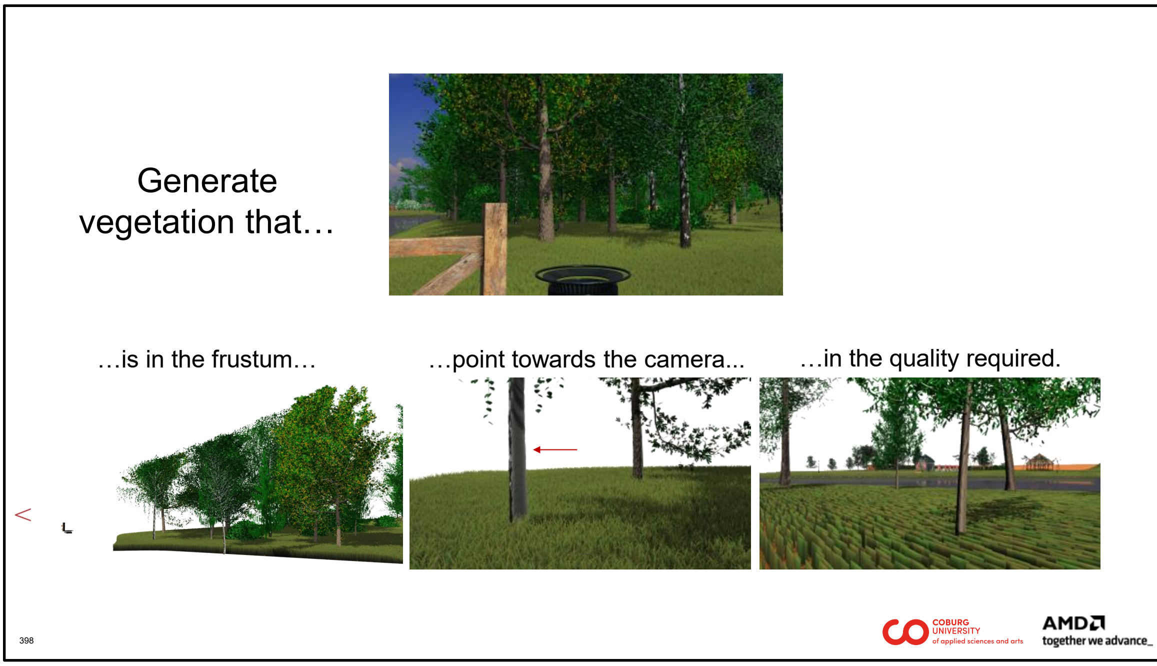
By sorting by asset, we can significantly reduce the number of draw calls.



View	Mode	Generation	Render	Draw Calls
Overview 79,710 / 79,710	Baseline	3.24 ms	27.74 ms	79,710
	Coalescing	3.14 ms	0.62 ms	965
Market 20,859 / 24,068	Baseline	2.34 ms	8.06 ms	20,859
	Coalescing	2.29 ms	0.51 ms	493

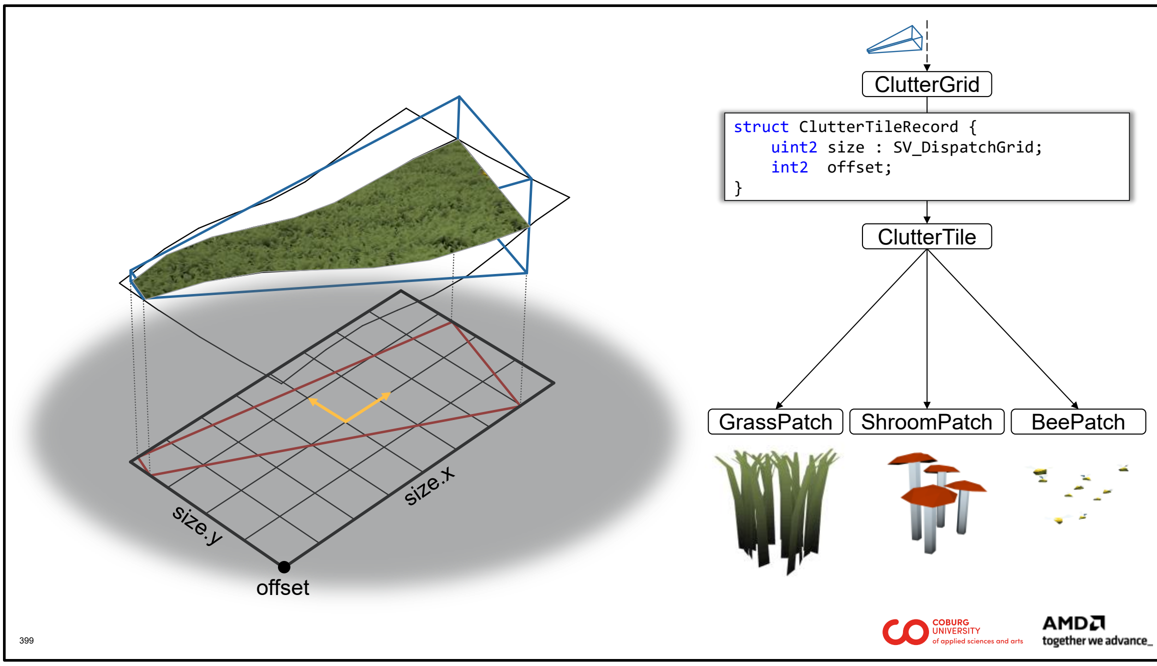
Here you can see the improvement, the number of draw calls was significantly reduced, same with the render timings.

Instead of placing existing assets, for our last two examples, we want our work graph to generate all the geometry from scratch.



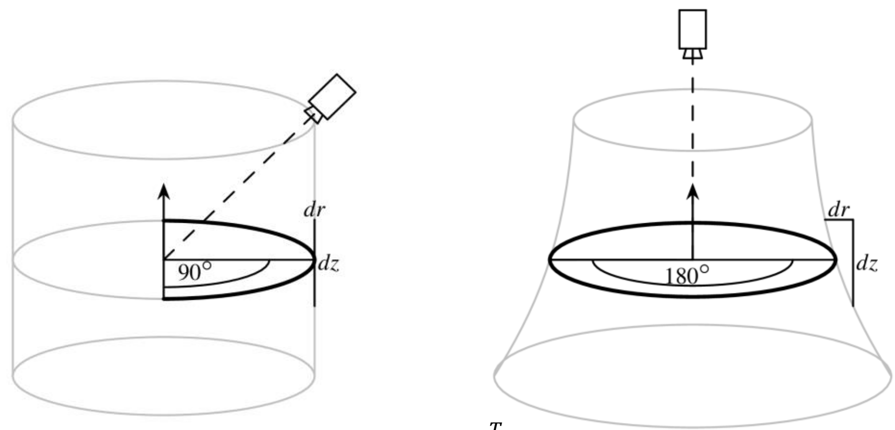
More specifically, for a given camera matrix, we want to only generate everything that is in the camera frustum, faces the camera, and only in the detail required.

The first one is easy: just omit dispatch records for work outside the frustum.



For our ground clutter, we find the 2D grid that encloses our camera frustum. Finer culling is then done inside the individual thread groups. We have a node array of mesh nodes for generating different kinds of clutter like grass, low LOD grass, mushrooms or insects.

For our trees, we omit outputting records of a, e.g., branch of a tree, when its bounding capsule lies outside the frustum.

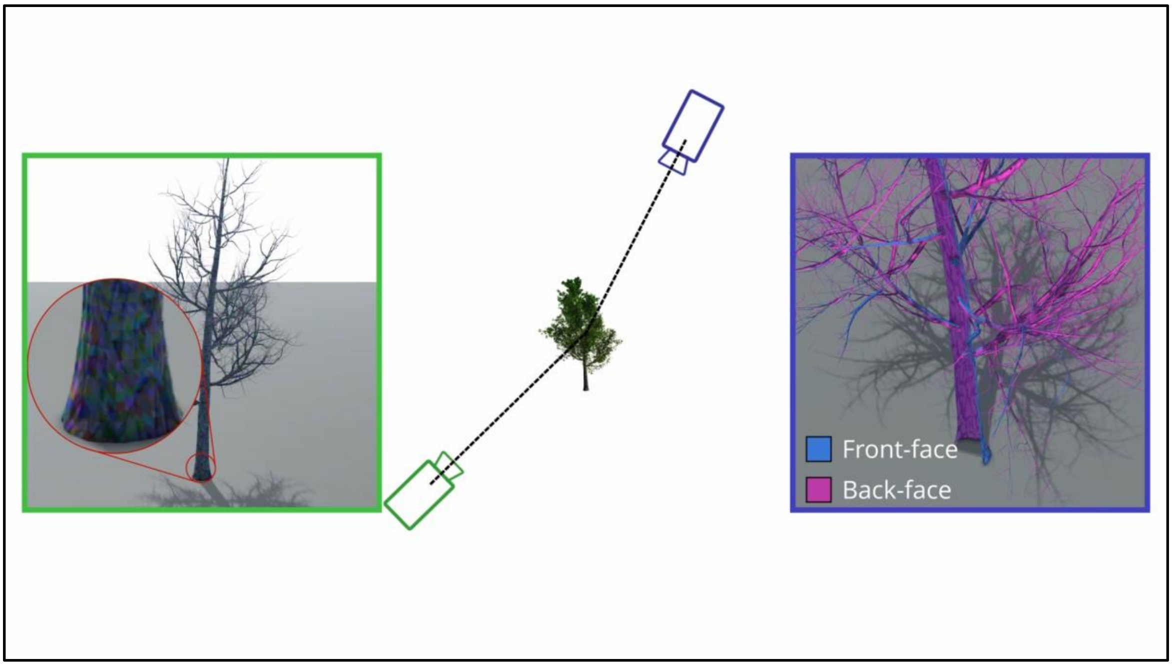


$$\boldsymbol{\theta} = \cos\left(max\left(\frac{dr}{dz} \cdot \frac{x}{\sqrt{1 - x^2}}, -1 \right) \right)$$

$z \cdot v^T$ (→ x)

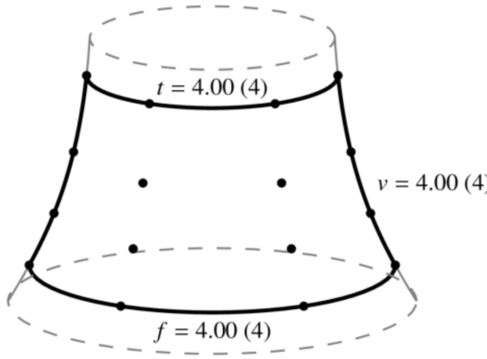
401

To only generate front facing triangles, we analyzed how far around a stem we have to tessellate given the tree growth direction, the change in stem radius, and the camera orientation.



In this video, you can see this in action.

$$t = \frac{2 \cdot \theta_{\text{to}} \cdot r_{\text{to}}}{\Delta}$$



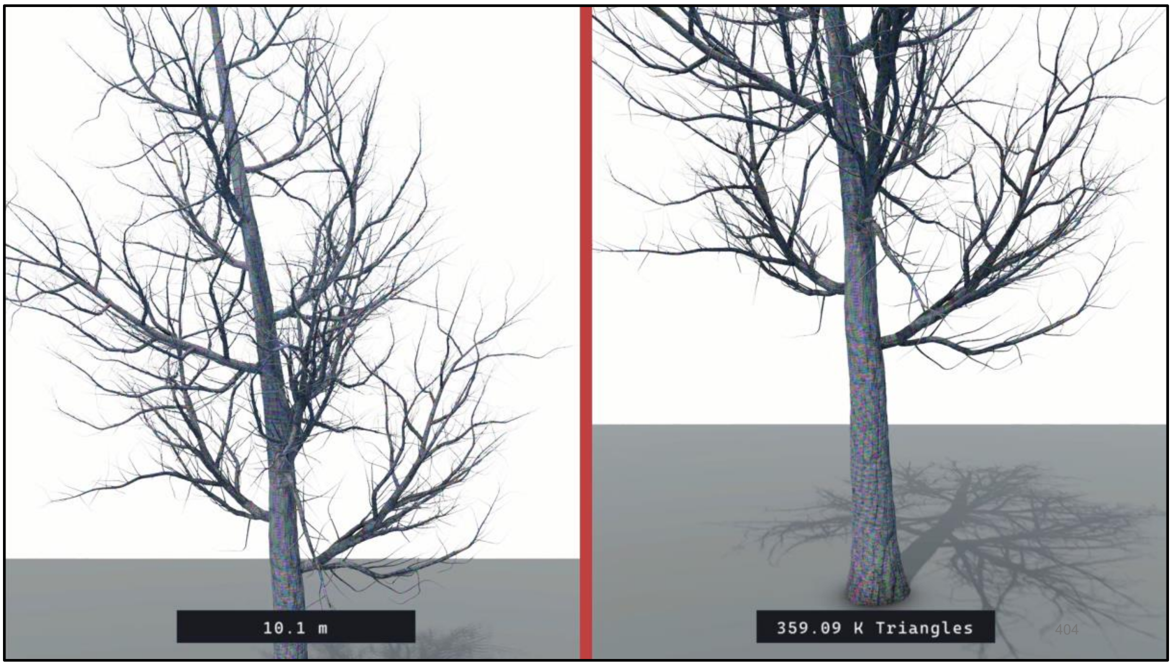
$$v = \frac{l}{\Delta}$$

$$f = \frac{2 \cdot \theta_{\text{from}} \cdot r_{\text{from}}}{\Delta}$$

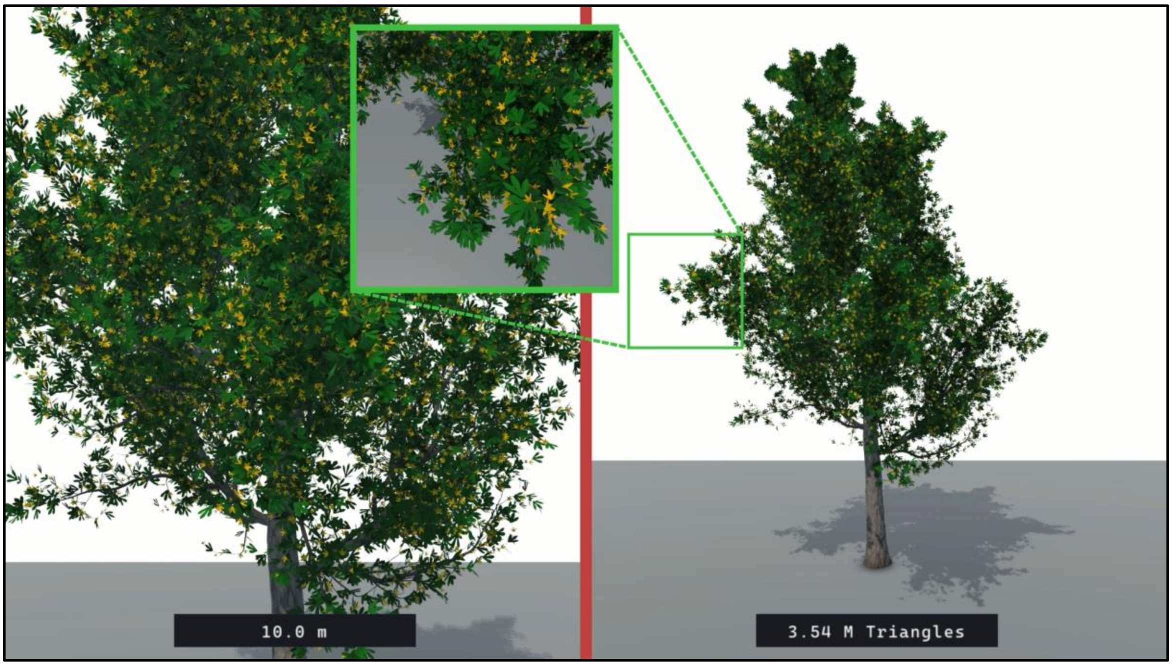
sample distance

403

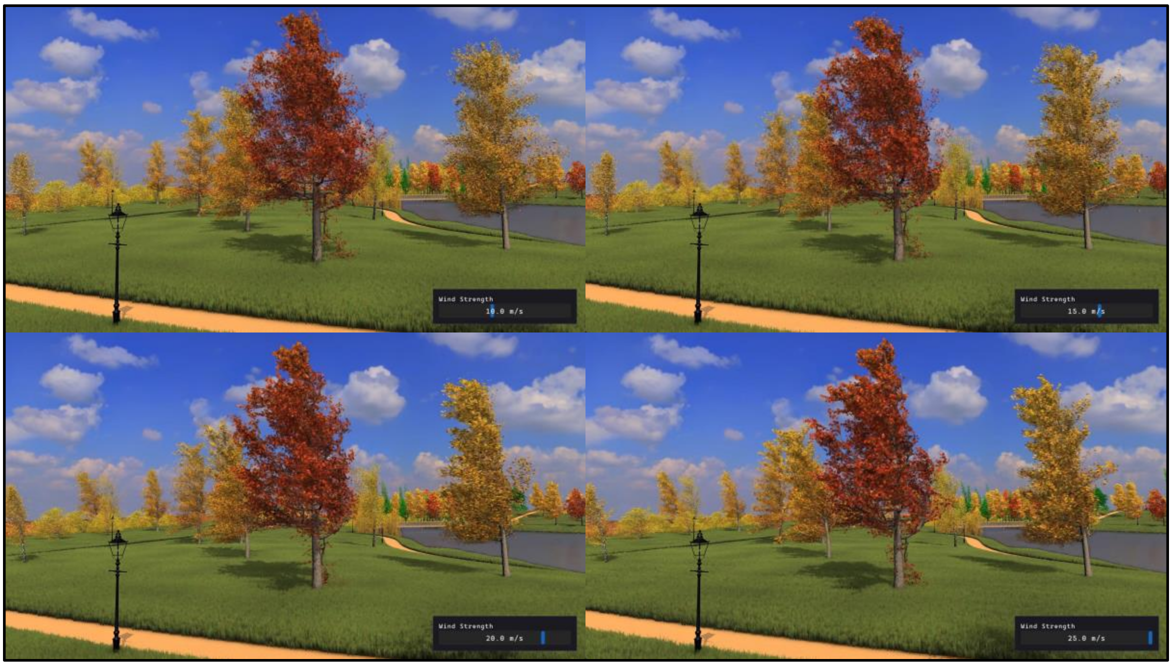
For continuous LOD, we employ fractional tessellation.



And in this video, you can see it in action.



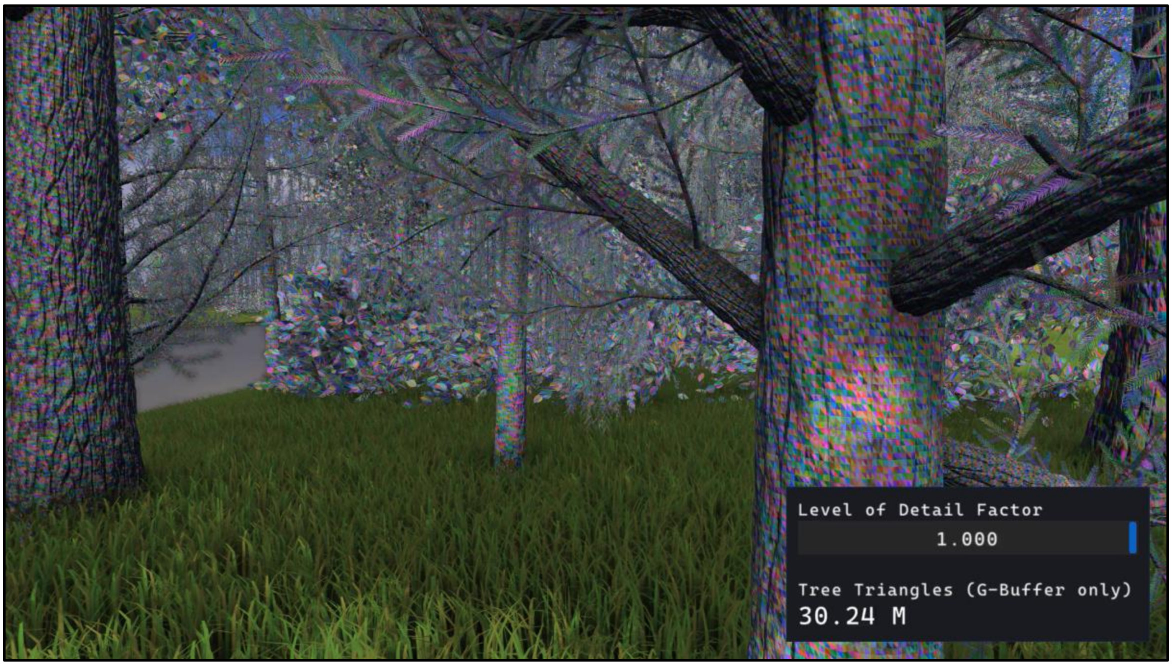
We do something similar with our leaf LOD.



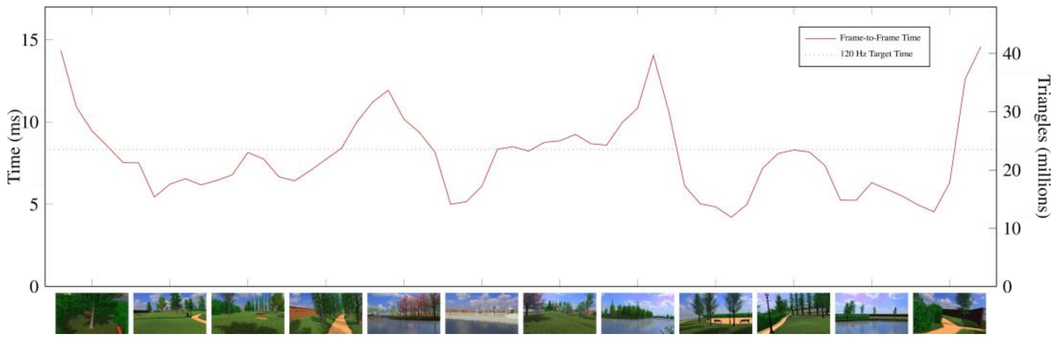
Another thing you can do with real-time generation is animation: simply adjust the generation based on the current timestamp.

Or how about adding seasonal detail based on a real number indicating the time of year.

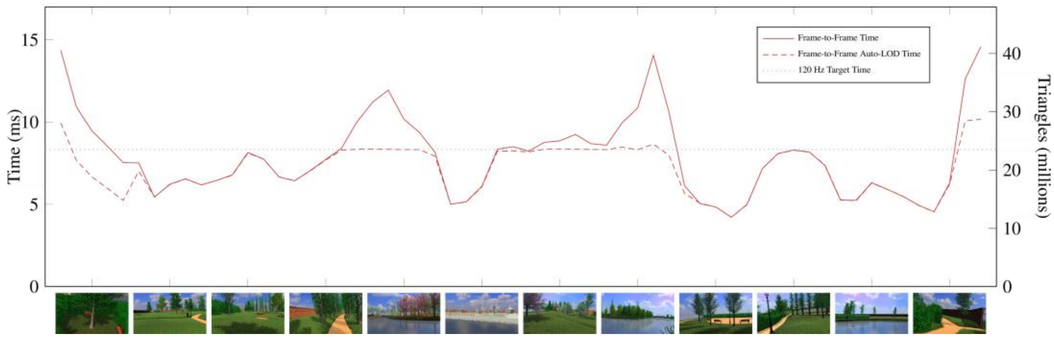
And here you can see real-time edits of our final tree model. Edits effecting an entire forest happen within the next frame.



With a continuous LOD, one can also adjust the image quality smoothly based on the current frame time.



Here is a performance measurement we did on a camera path. Frame-to-frame times vary based on image complexity between 13 – 40ms.

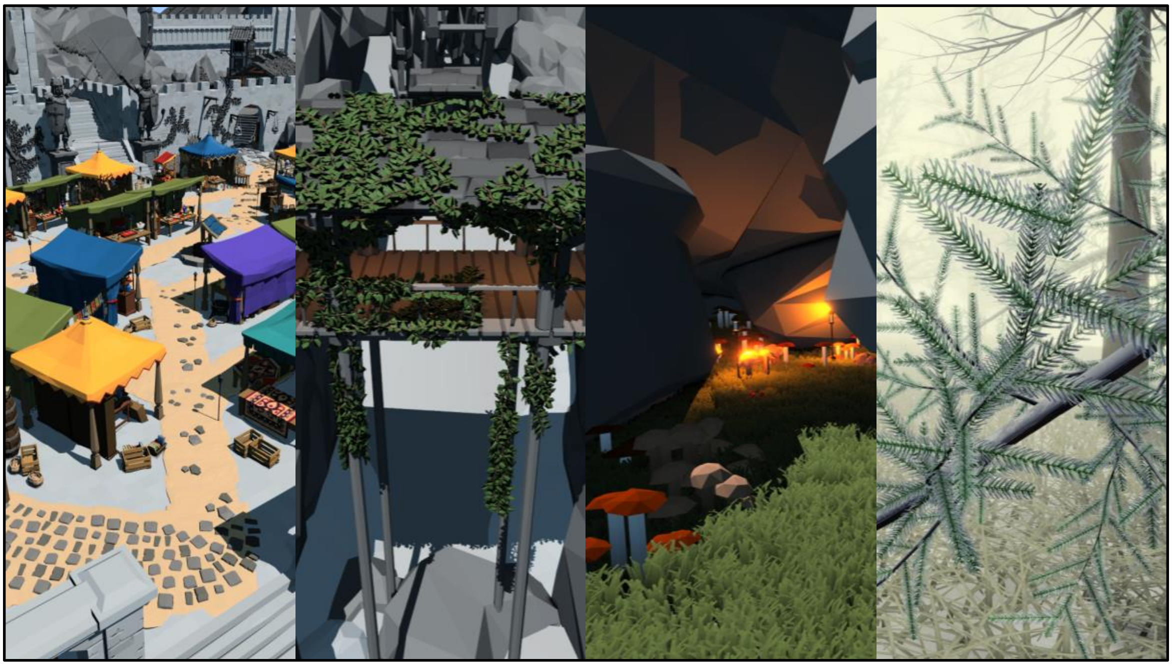


411

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

With an automatic LOD, the performance peaks can be mitigated.



This concludes the procedural generation part of this course.

Advanced Work Graphs

Work Graphs under the Hood

413

As part of the Advanced Work Graphs Section, we would like to present ideas of how Work Graphs might potentially be implemented on a GPU.

How does it work?

414

So, so far, we've seen what Work Graphs is, how it allows us to schedule work directly on the GPU, and how that can help us solving different use-cases.

But how does this "launching work from the GPU" work?

How does it work?

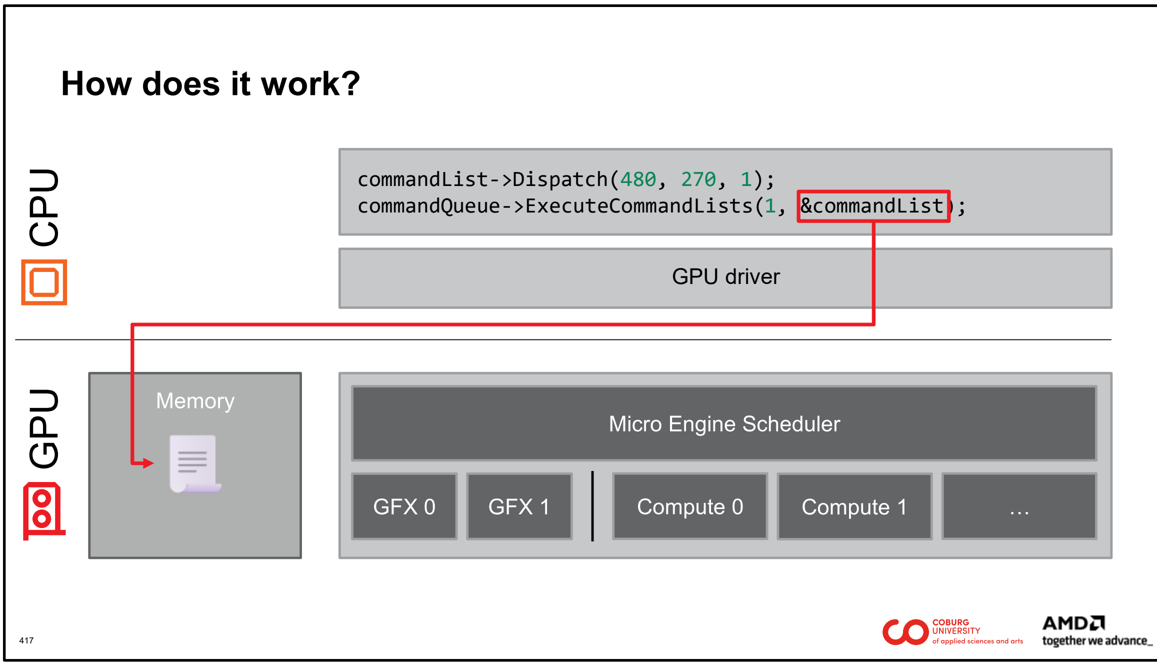
```
commandList->Dispatch(480, 270, 1);
commandQueue->ExecuteCommandLists(1, &commandList);
```


To understand this, we first need to look at how any launch of work on the GPU works. In Direct3D12, we record commands, as for example this `Dispatch`, into a `commandList`.

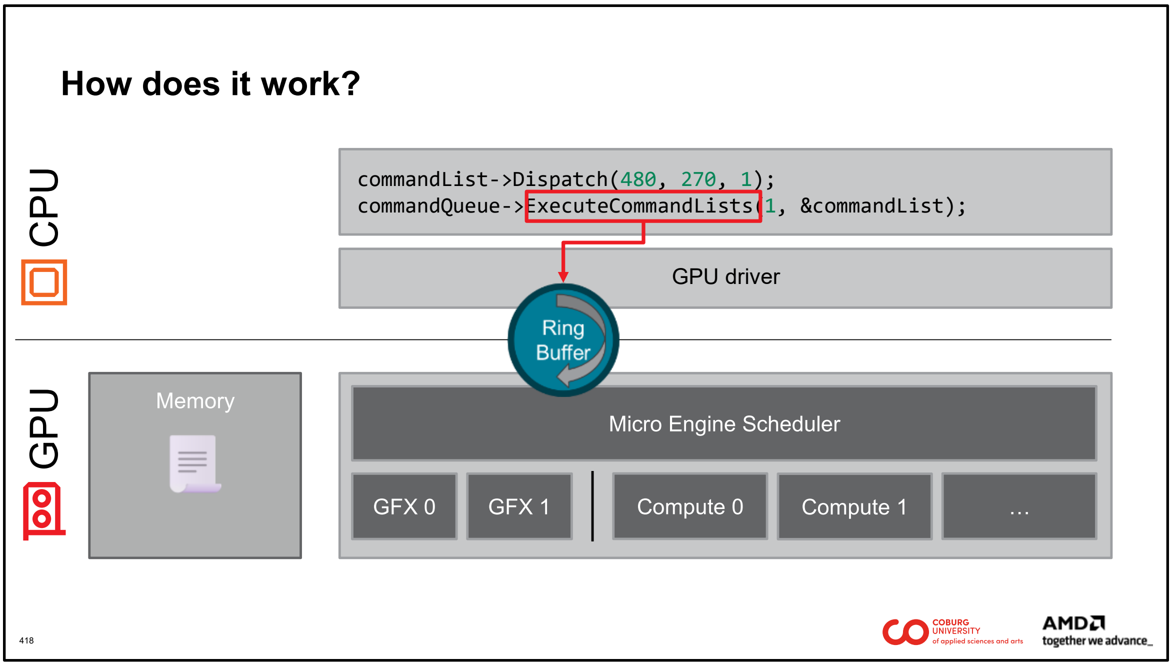
How does it work?

```
commandList->Dispatch(480, 270, 1);
commandQueue->ExecuteCommandLists(1, &commandList);
```

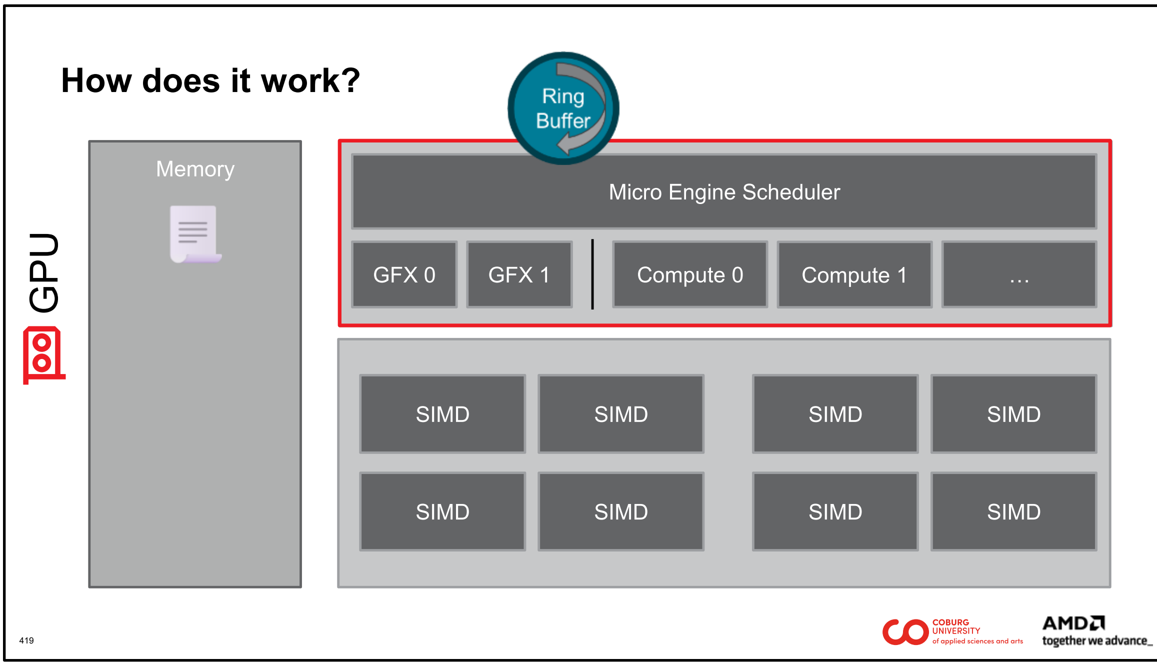

To execute this `commandList`, we chose a `commandQueue` and submit our command list to it.



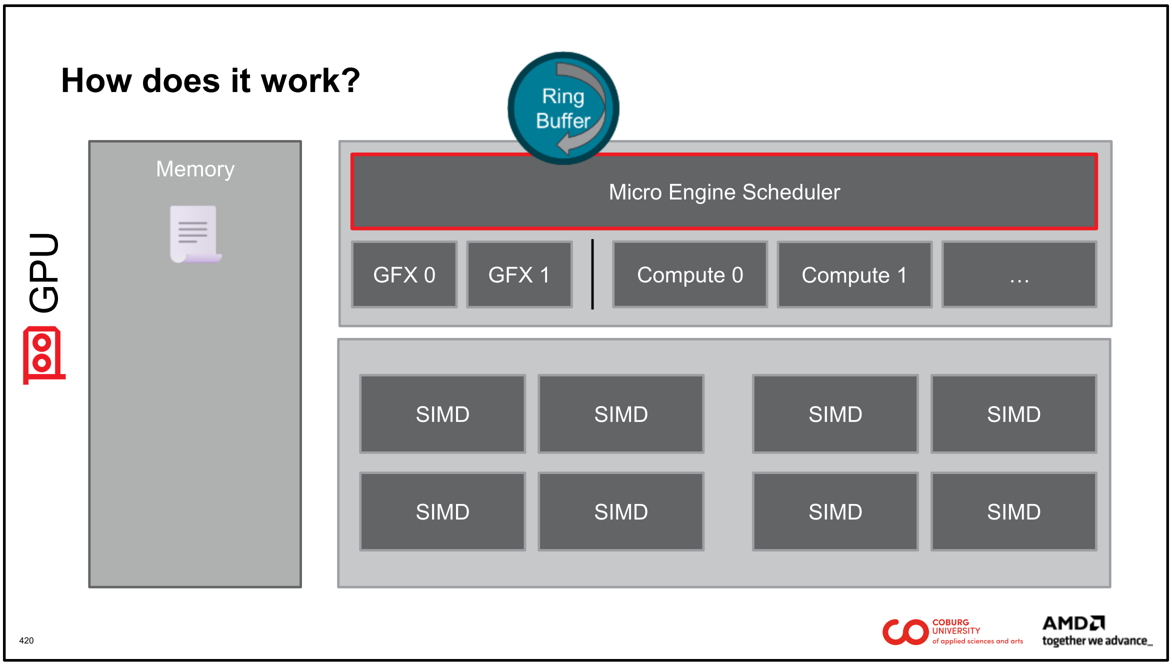
To then actually execute the `commandList`, the GPU driver will copy the command list ( ) into GPU-visible memory…



… and passes an execute-command through a ring buffer to the GPU.

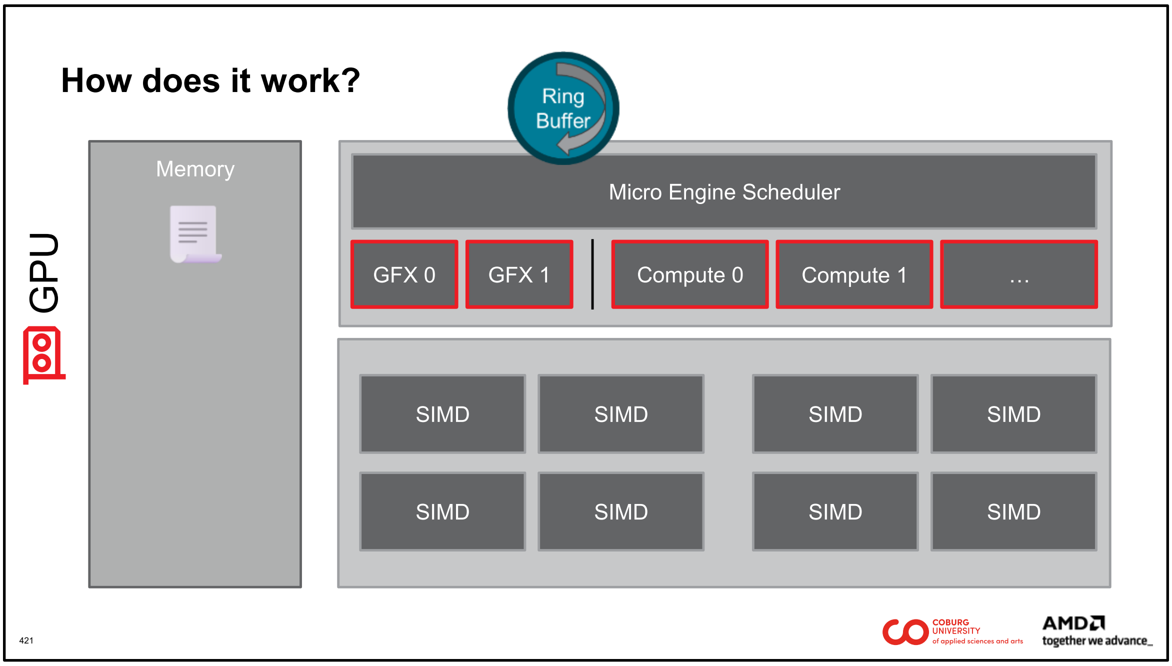


On the GPU, this ring buffer is connected to the command processor (red box)



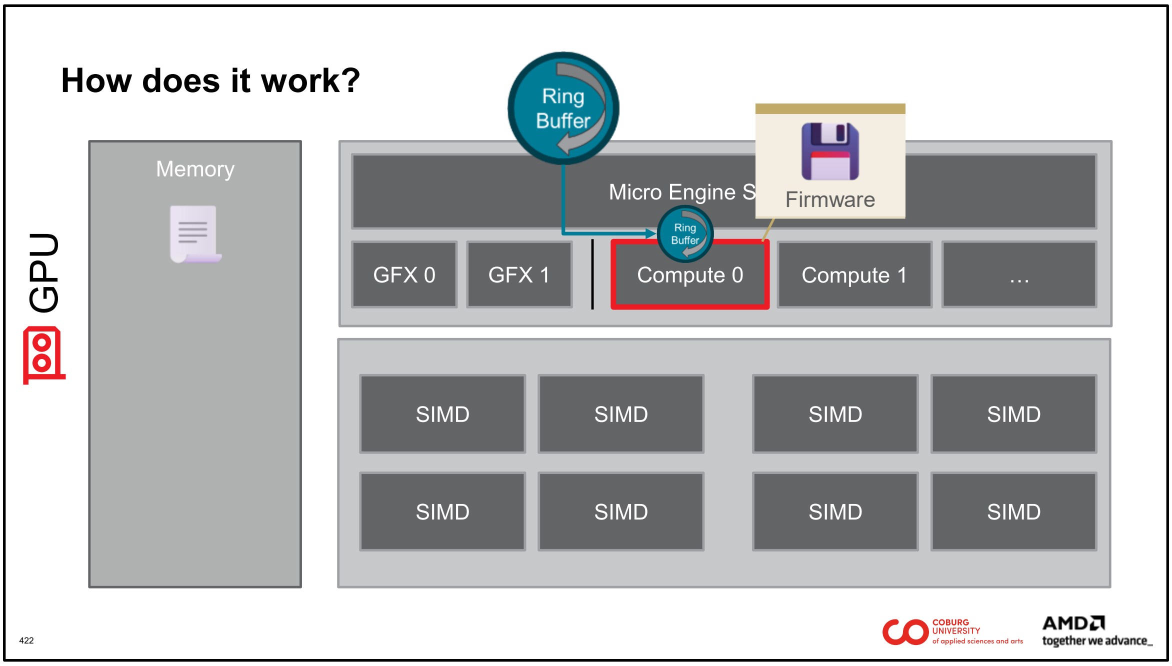
Or more specifically the Micro Engine Scheduler, which is a part of the command processor.

The Micro Engine scheduler is responsible for handling commands, such as the one to execute the command list coming from the CPU.



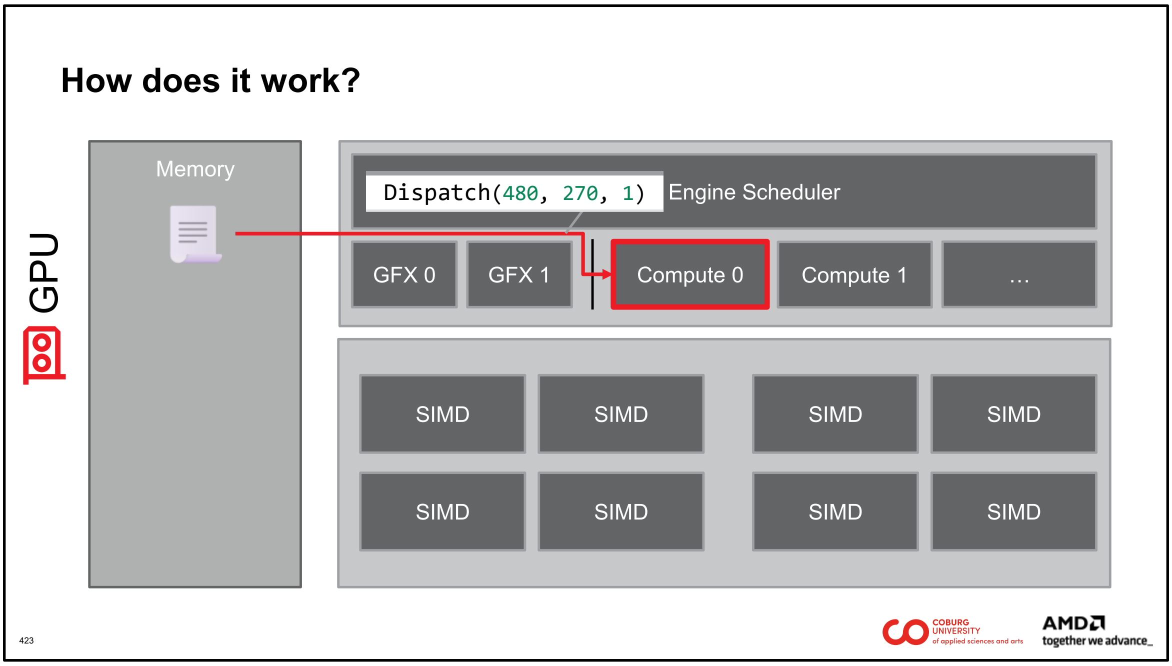
To process any of the incoming commands, the Micro Engine Scheduler has access to different queues.

There are two types of queues: graphics queues (GFX 0, GFX 1 in the slide) and compute queues (Compute 0, Compute 1, …, in the slide).



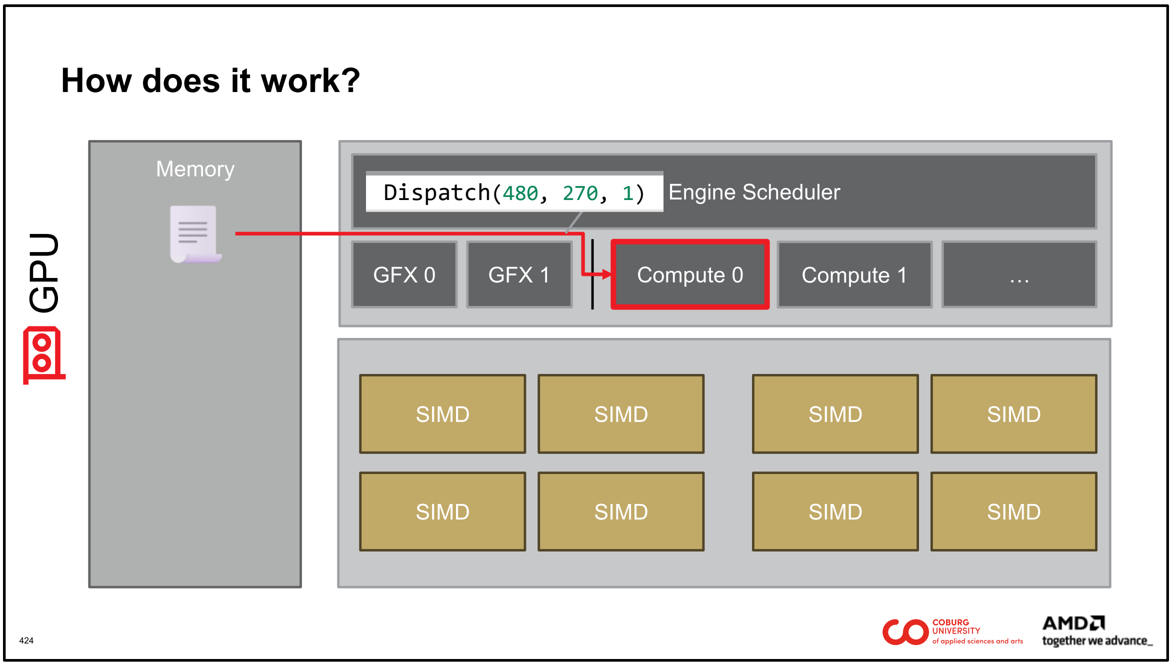
The Micro Engine Scheduler selects one of those queues – in this case Compute 0 – and maps the incoming command to its input ring buffer.

Each of these queues is a small processor which is programmed through the firmware to execute the commands.

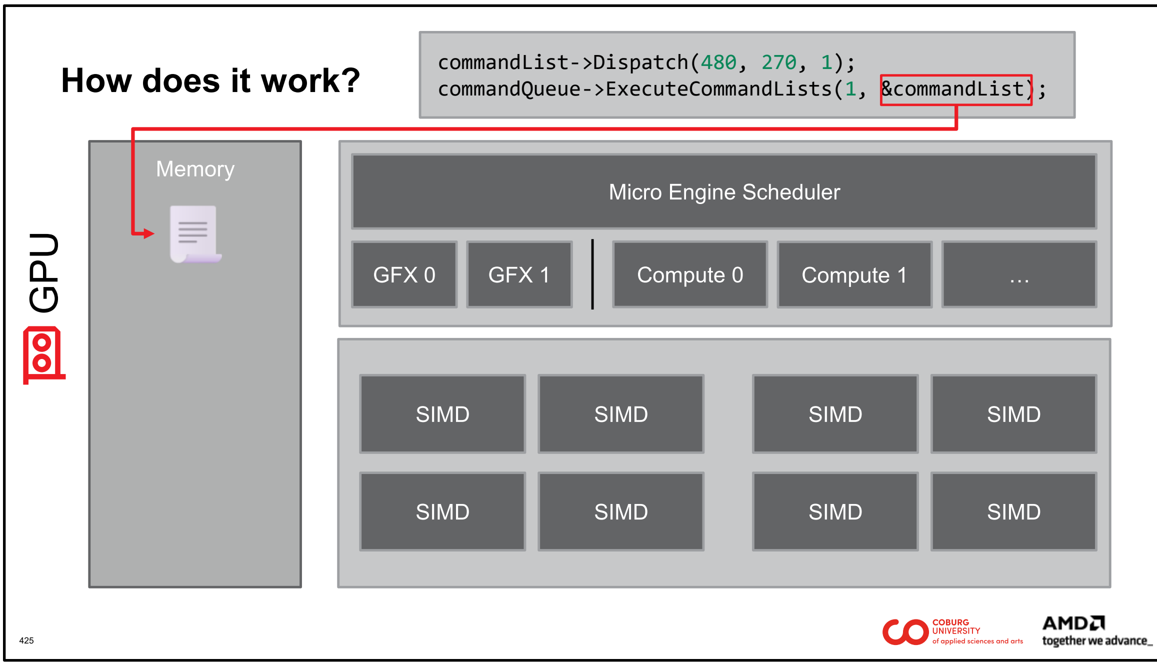


In this case, we want to execute our command list, so the command processor fetches one command after the other from memory, parses, and executes it.

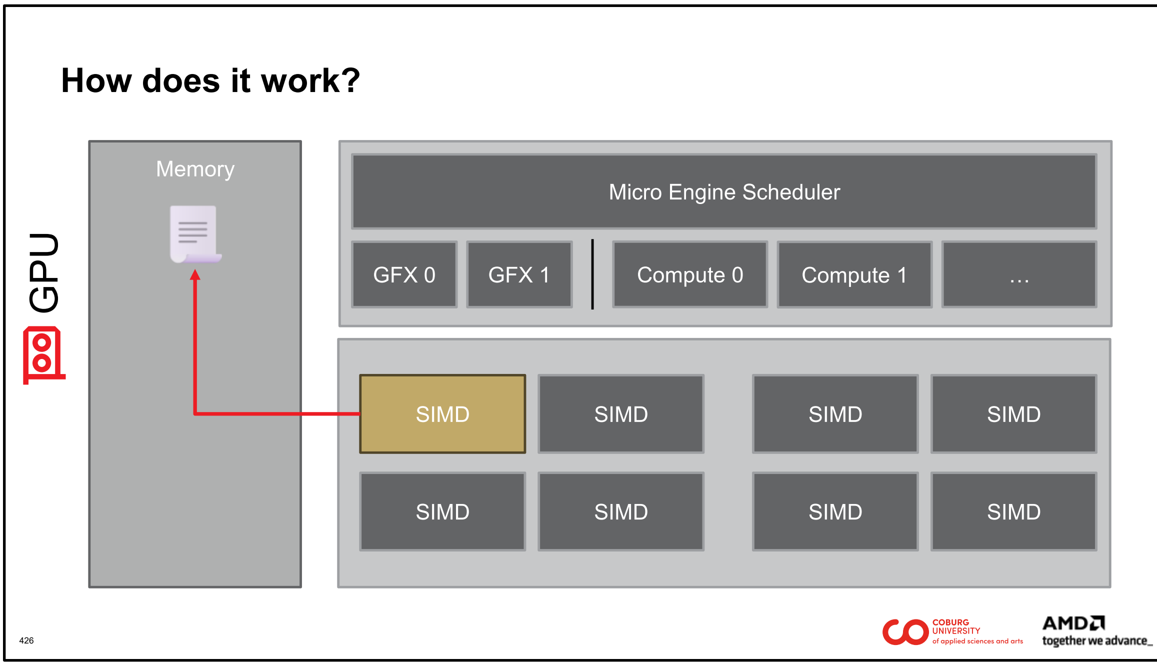
In our example here, we have the dispatch command from before.



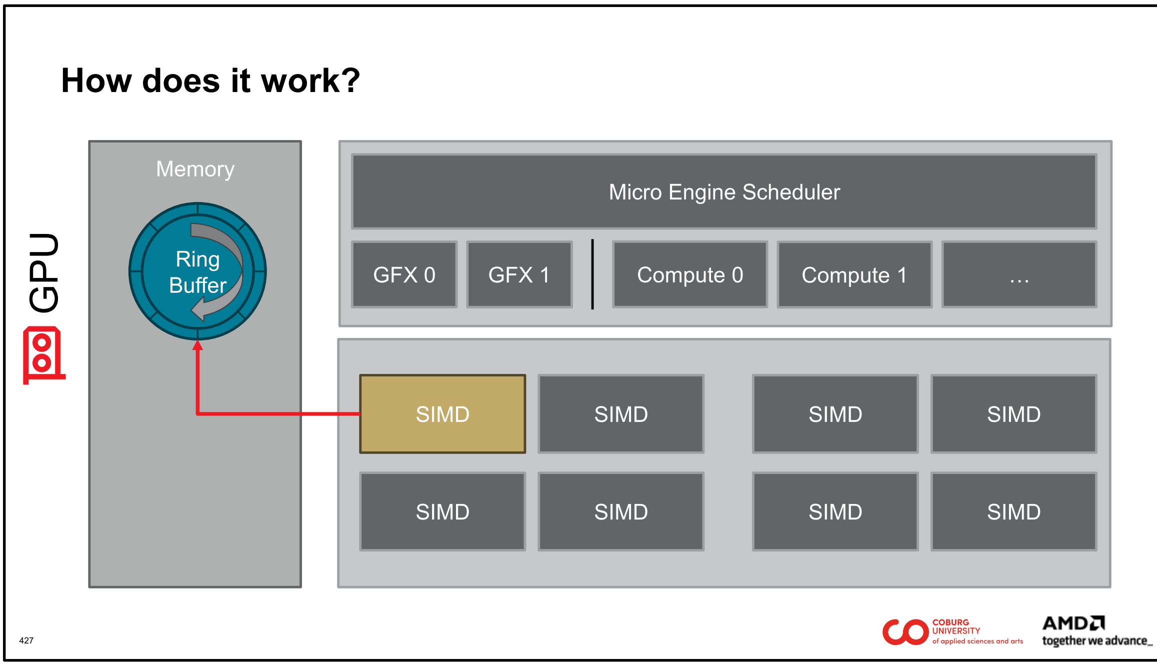
Compute queue 0 then sets up and invokes the SIMDs in order to carry out the dispatch command. This is a very simplified view of the GPU, as we are only interested in how commands such as dispatches are handled and not the specifics of how the actual thread groups of the dispatch are mapped and set up to GPU hardware components.



We've seen that we can place a command buffer in GPU memory and have the command processor execute it.

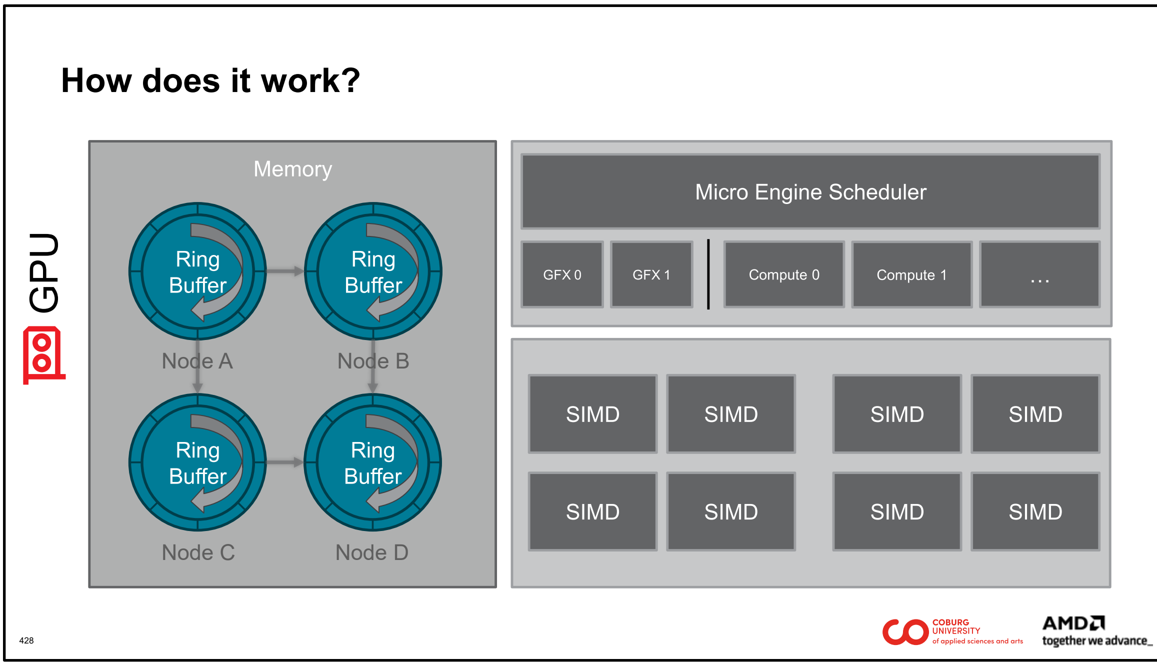


This would mean that if we want to schedule work from the GPU itself, we can just write to such a command buffer in GPU-visible memory and have the command processor execute it.



In the case of Work Graphs, we‘re not writing a command list, but we‘re writing records. To allow for a continuous cycle of writing and launching these records, we can store these records in a ring buffer.

How does it work?

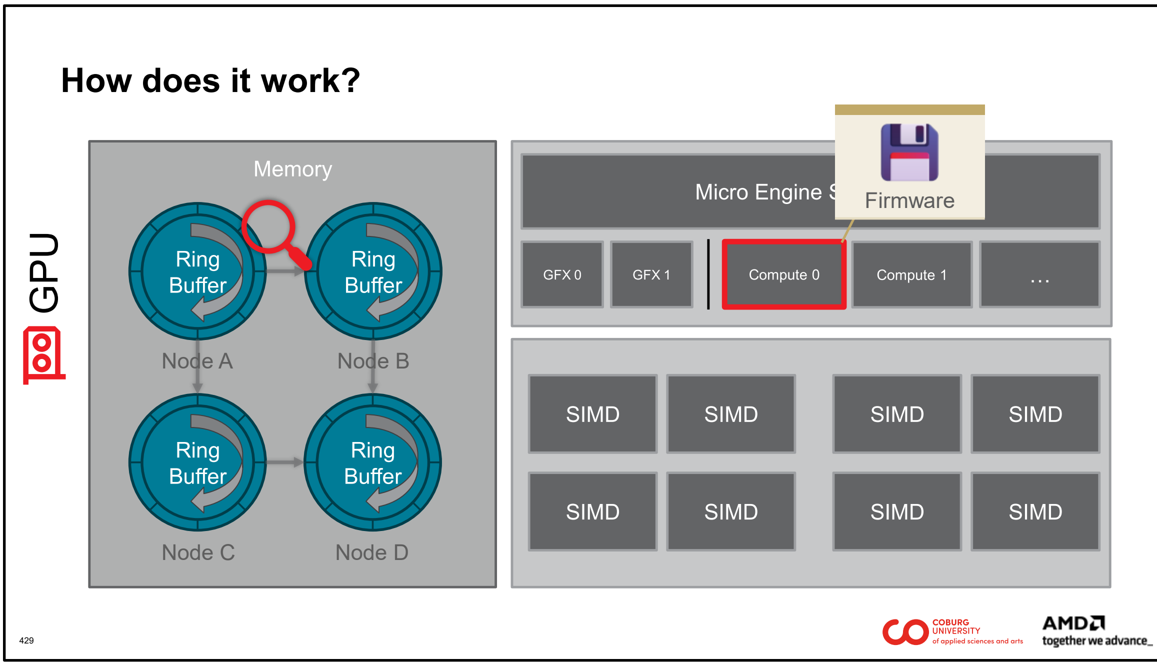


As we have different records for different nodes, we need multiple ring buffers, one for each node.

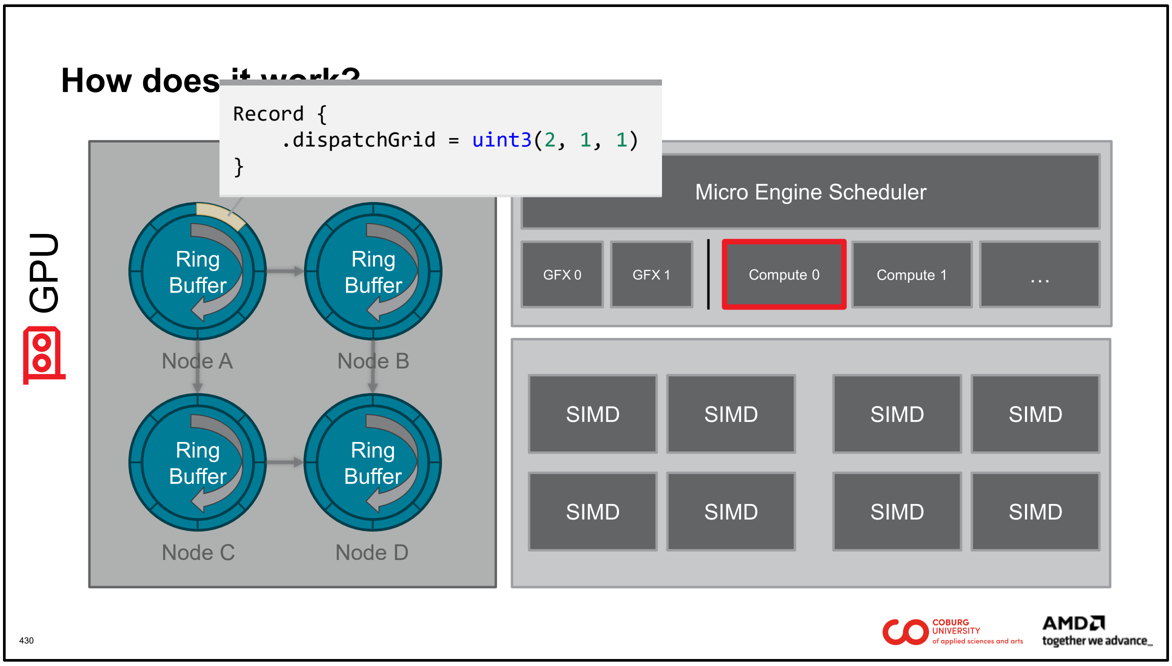
Here we have a very simple Work Graph with nodes A, B, C and D. A can send records to B and C, i.e., thread groups that run code for node A can write records to the ring buffers of node B and C.

Nodes B and C can both send records to node D.

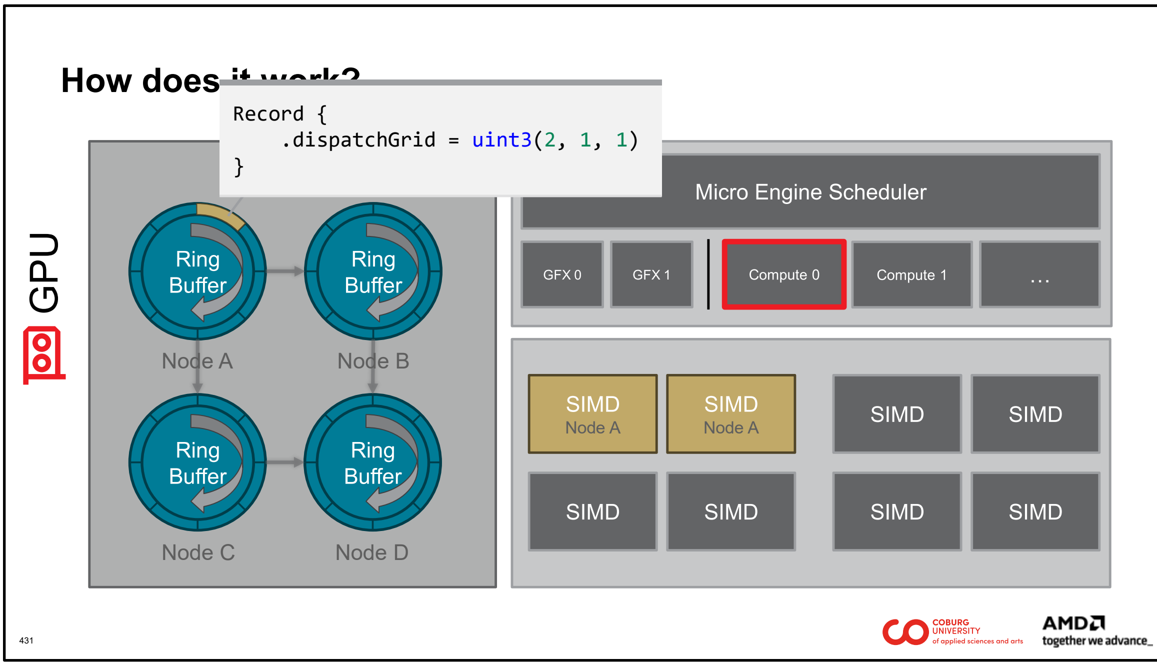
How does it work?



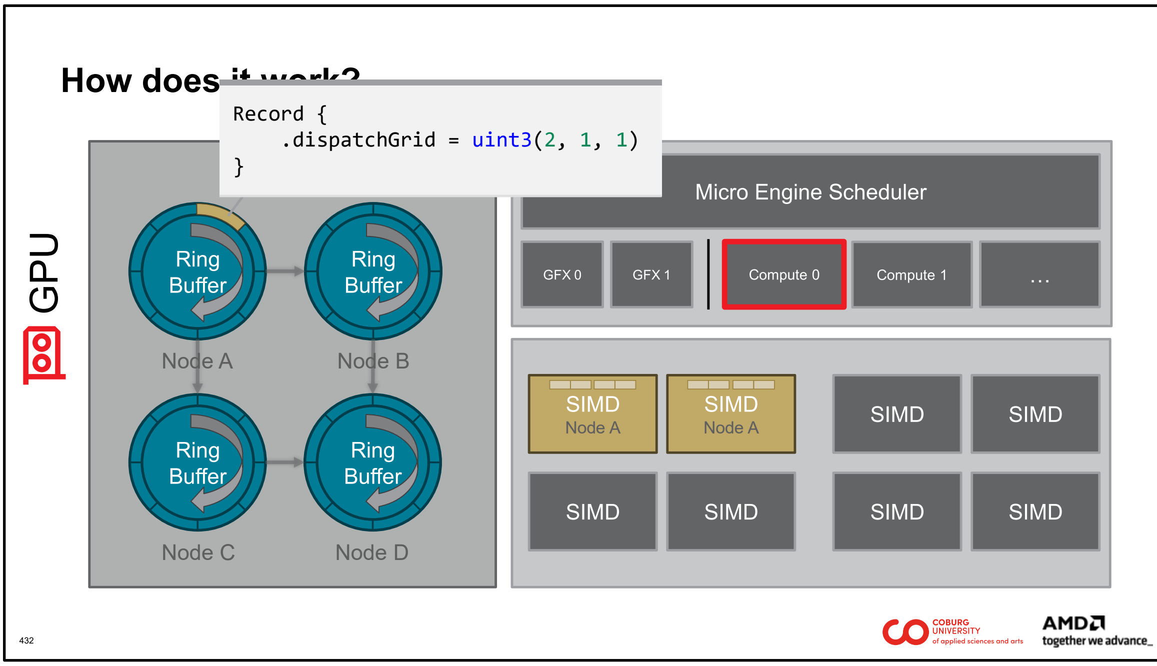
The compute processor can then scan these ring buffers in memory for available records and decide what records to launch and how to launch them.



As an example, we have placed a record in the ring buffer of node A. Node A is using a dynamic dispatch grid and the record specifies that two thread groups should be launched.

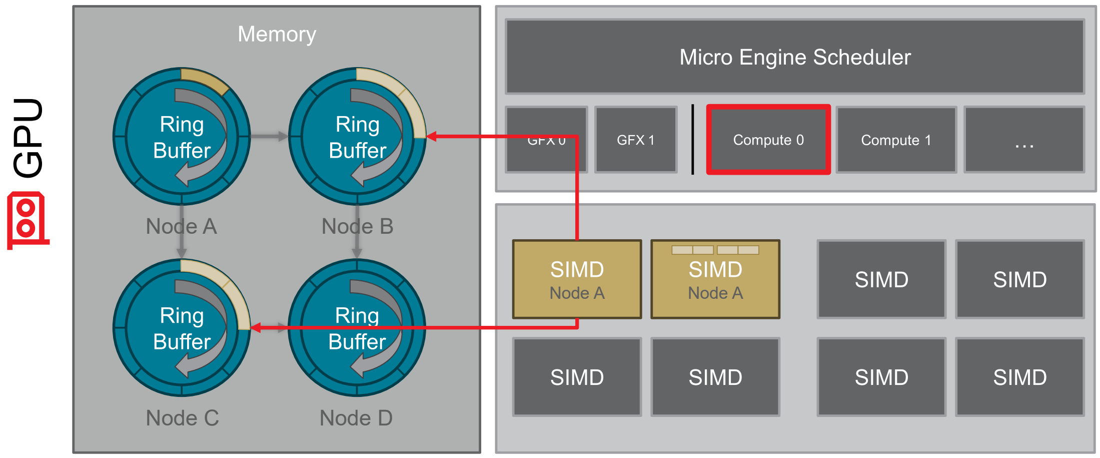


Compute queue 0 can then find this record in the ring buffer and launch two thread groups for it. In our simplified GPU, we've mapped each of these thread groups to on SIMD.



Each of these thread groups then want to send two records to node B and two records to node C. These records are visualized by small yellow boxes at the top of each of the SIMDs.

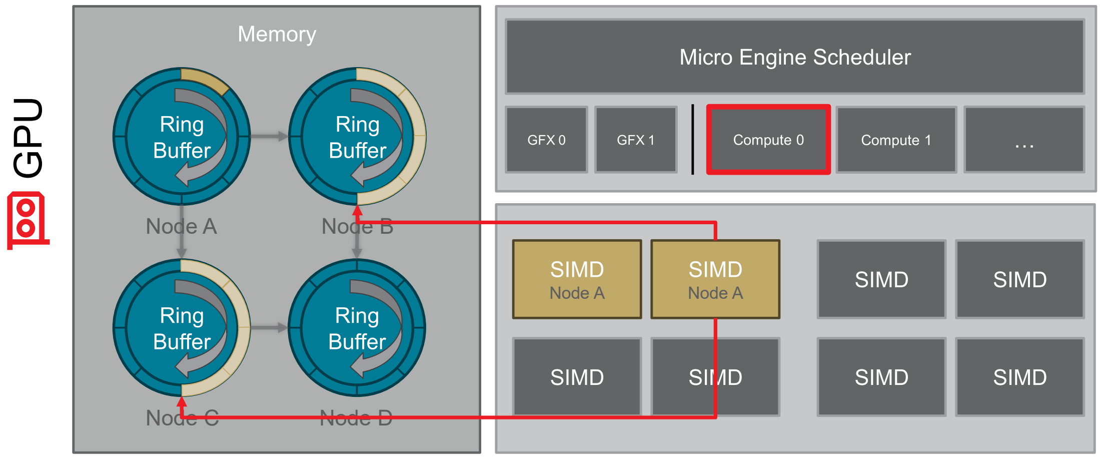
How does it work?



433

Each of these thread groups can then write their outputs to the respective ring buffers of the nodes. Here the first thread group writes its four records…

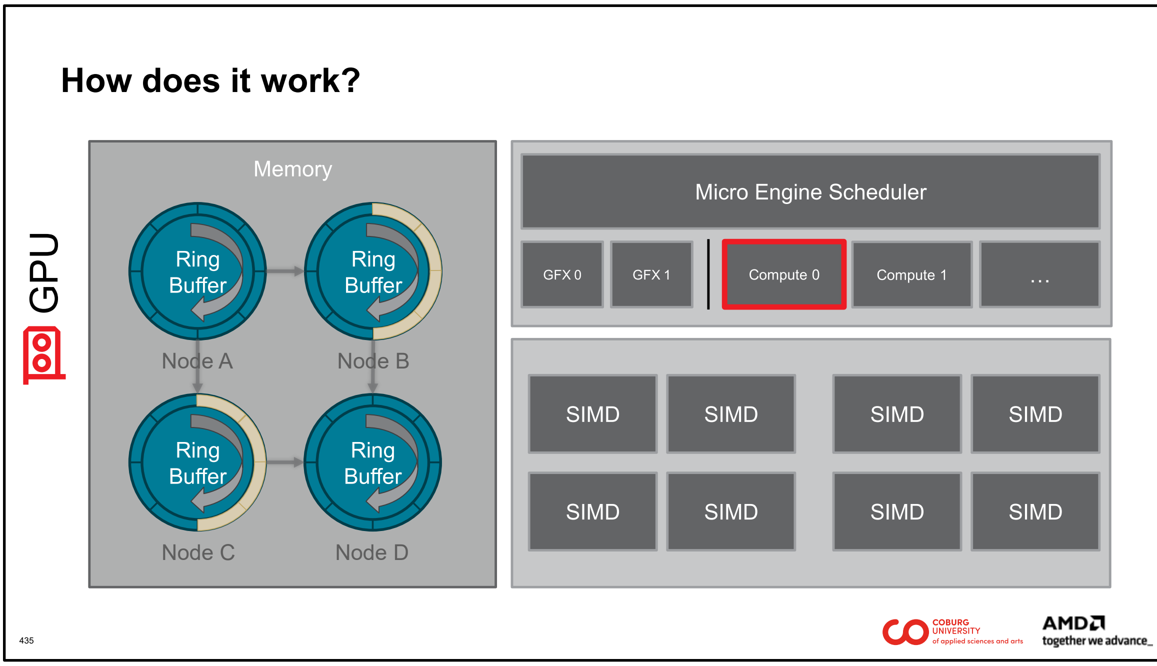
How does it work?



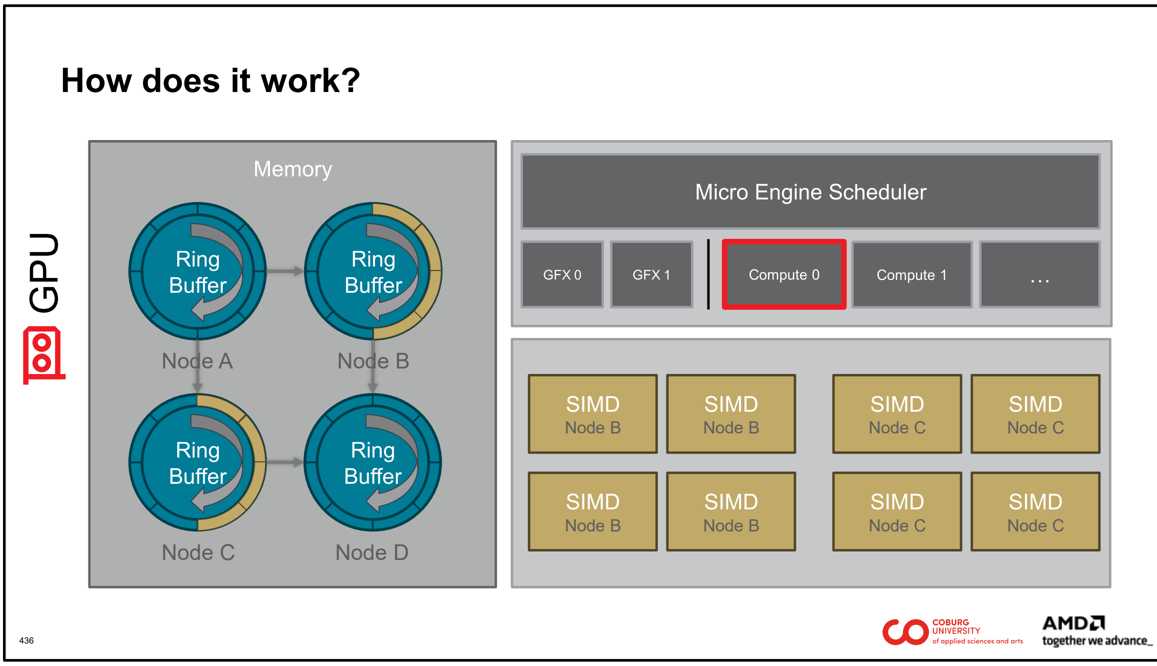
434

COBURG UNIVERSITY of applied sciences and arts
AMD together we advance_

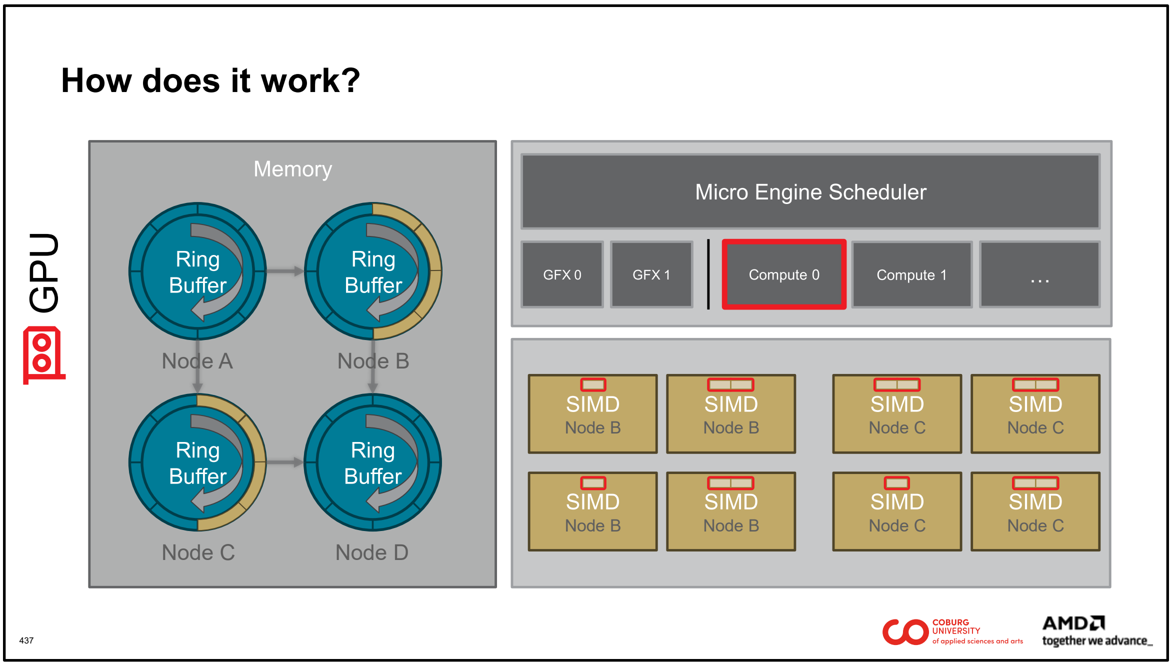
…and so does the second thread group.



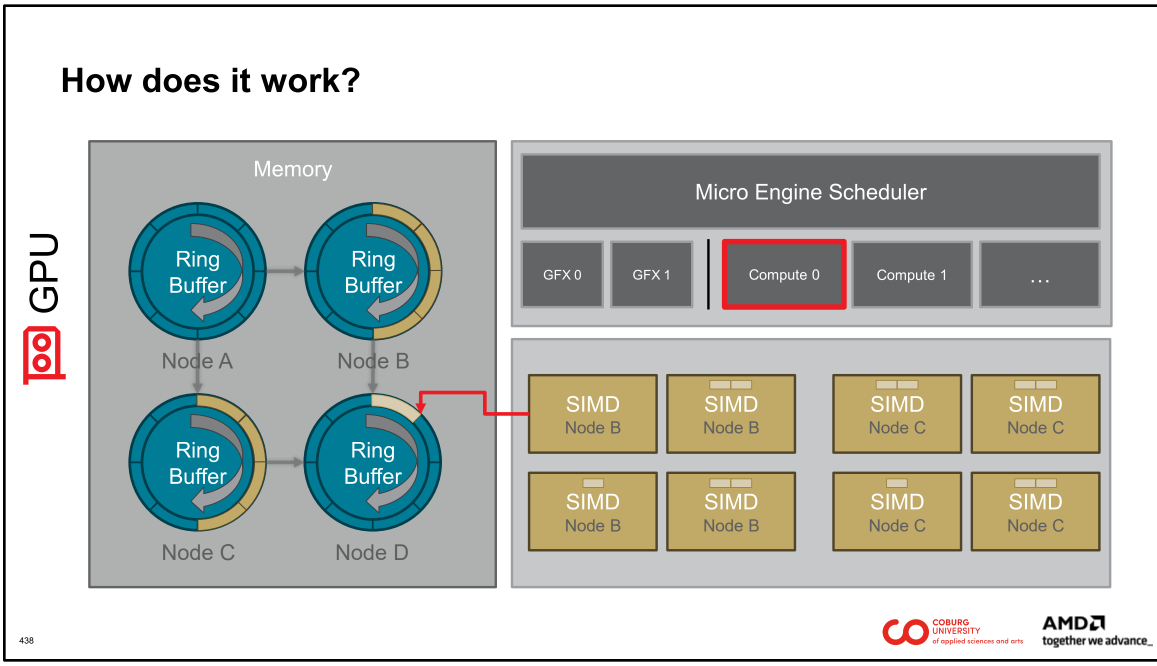
Once writing is complete these records are ready to be picked up by command queue 0 and launched.



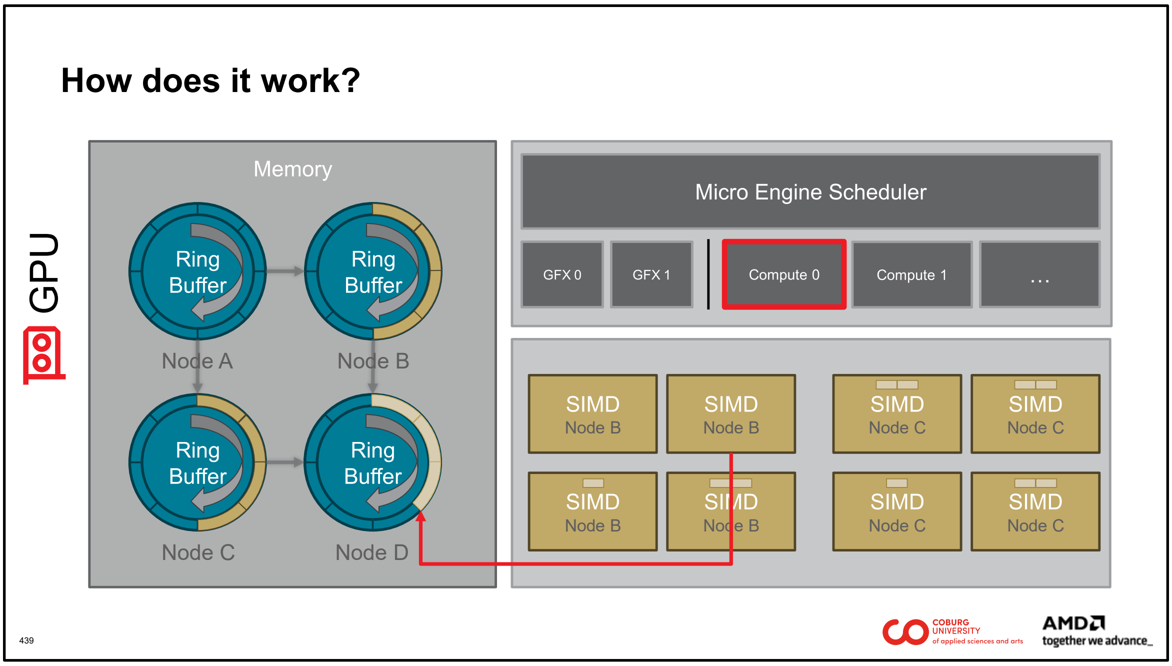
Command queue 0 therefore launches thread groups for each of the records in the ring buffer of node B and node C. In our example, each of these records will launch a single thread group, this yielding four thread groups for running code for node B and four thread groups running code for node C.



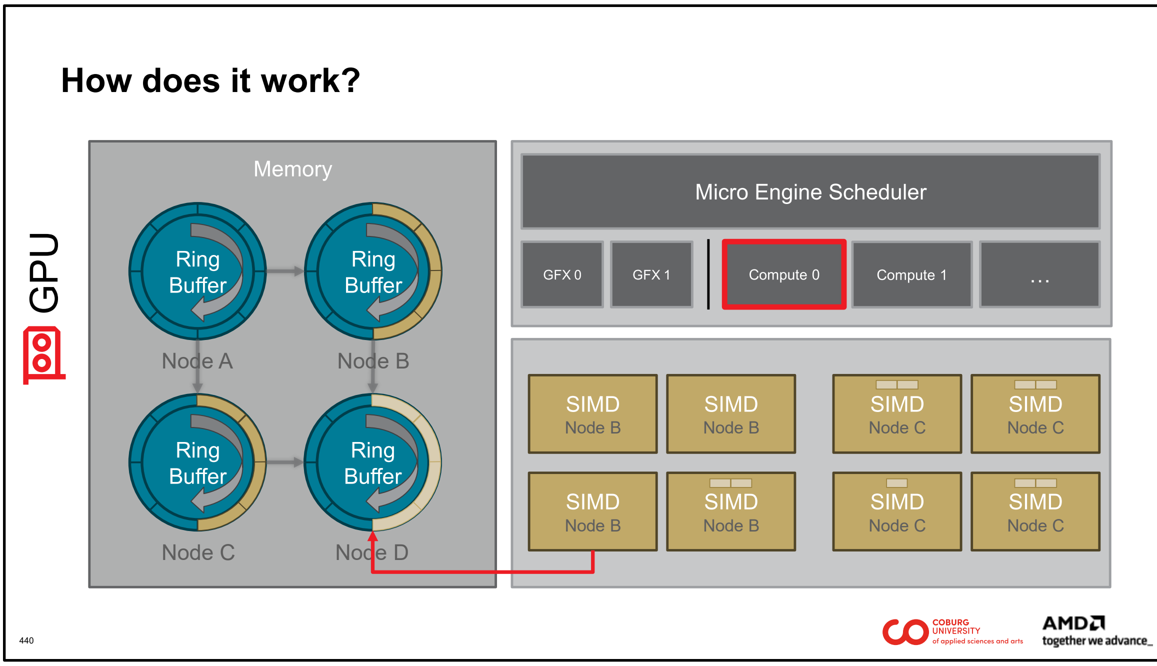
Each of these thread groups then want to send between one and two records to node D. These are again visualized with small yellow boxes in each of the thread groups.



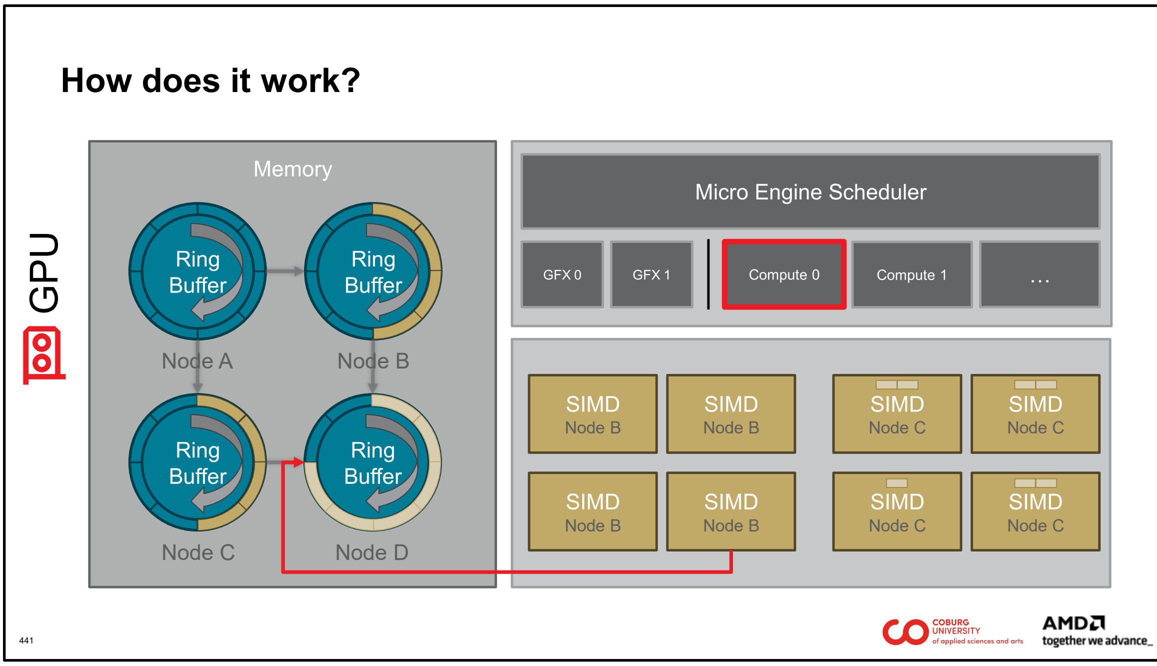
Each of these thread groups then writes their output to the ring buffer of node D. The first thread group of node B writes a single record, …



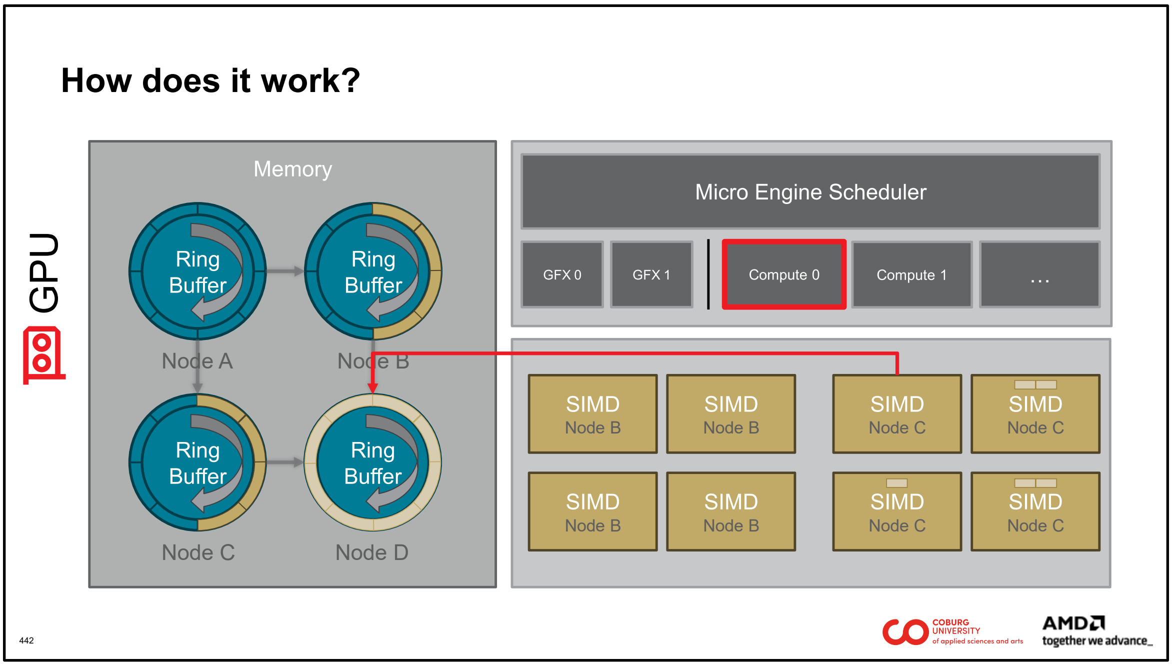
...the third one writes two records, …



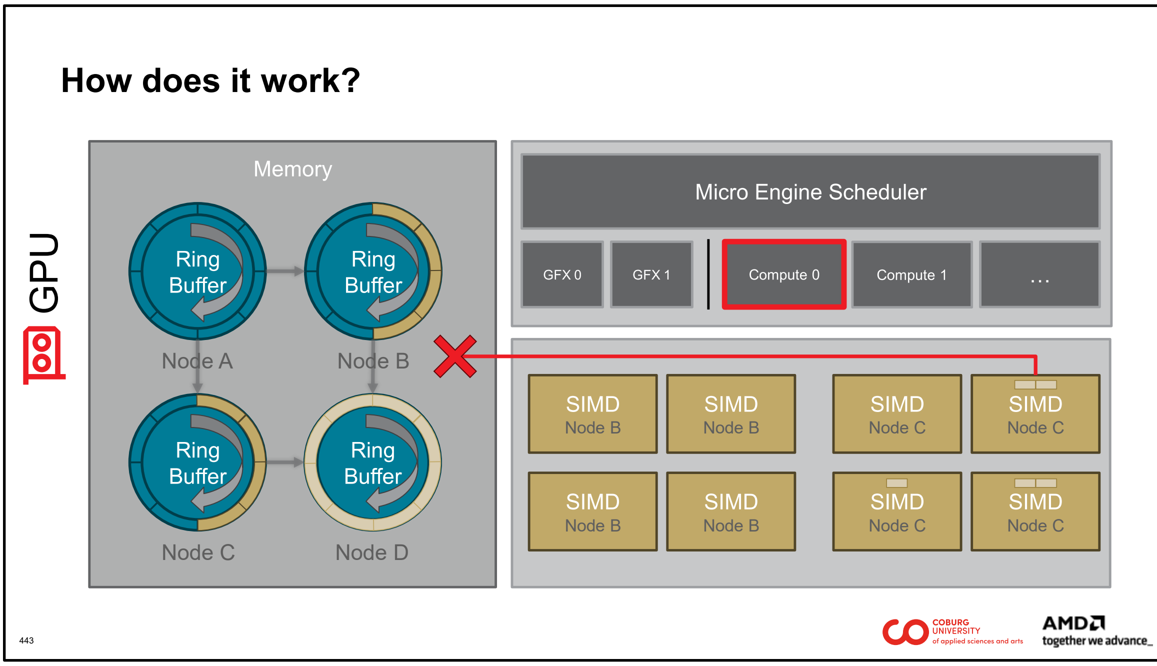
… and the third one writes a single record again.



And finally the last thread group of node B writes two records.



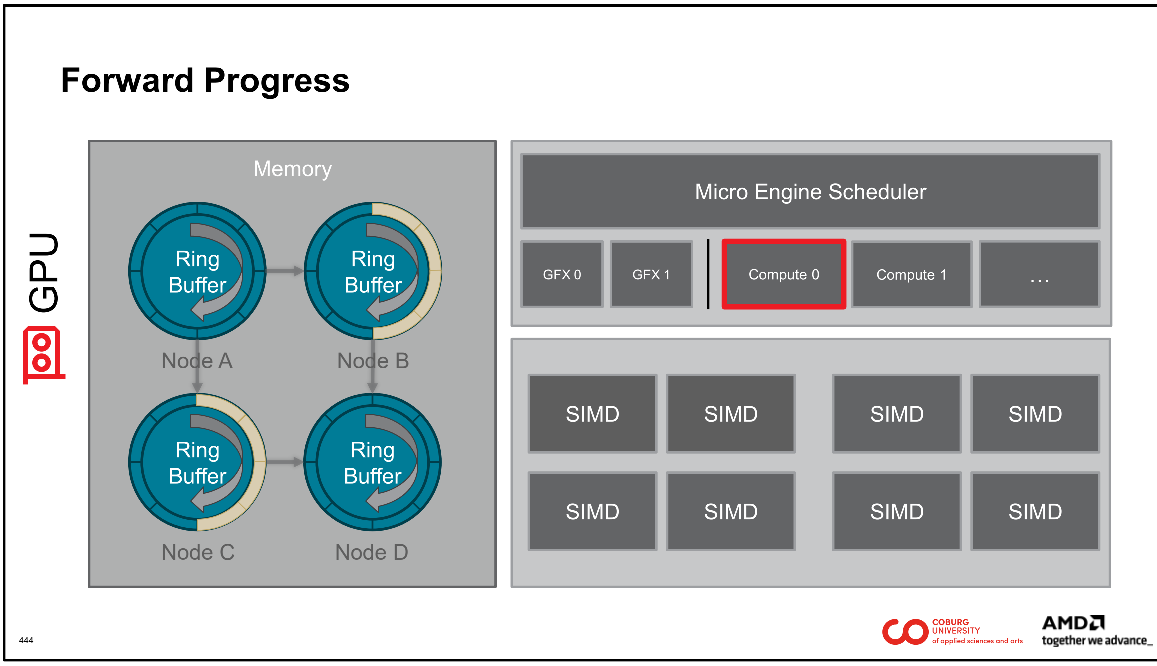
The same process continues for al the thread groups for node C. The first thread group starts by writing two records to the ring buffer of node D.



The second thread group wants to write two records to the ring buffer of node D.

However, the ring buffer for node D is now full, thus no more records can be written to it. Simultaneously, all SIMDs of the GPU are busy, thus the command queue cannot launch any records to free up space in the ring buffer for node D.

This is obviously a problem, since we are now in a deadlock. So maybe this launching work from the GPU is not as simple as initially assumed. Let's go back a few steps to see what we missed.

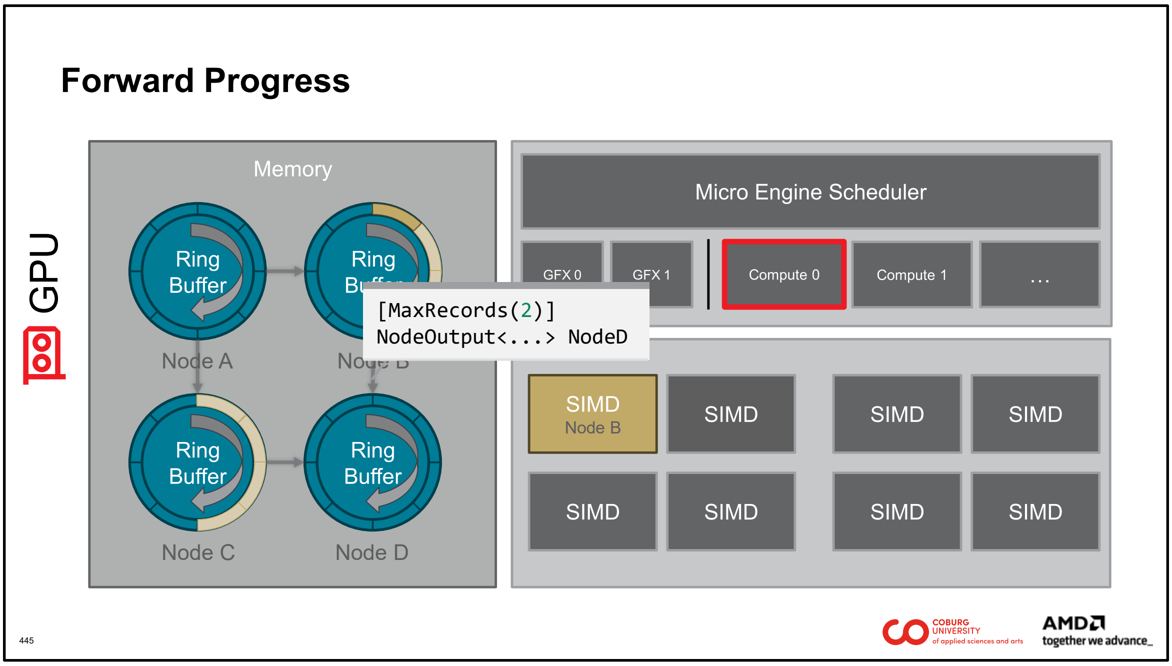


We return to the state just before we started launching the records in the ring buffers of node B and node C. Currently, we have four record in each of these ring buffers.

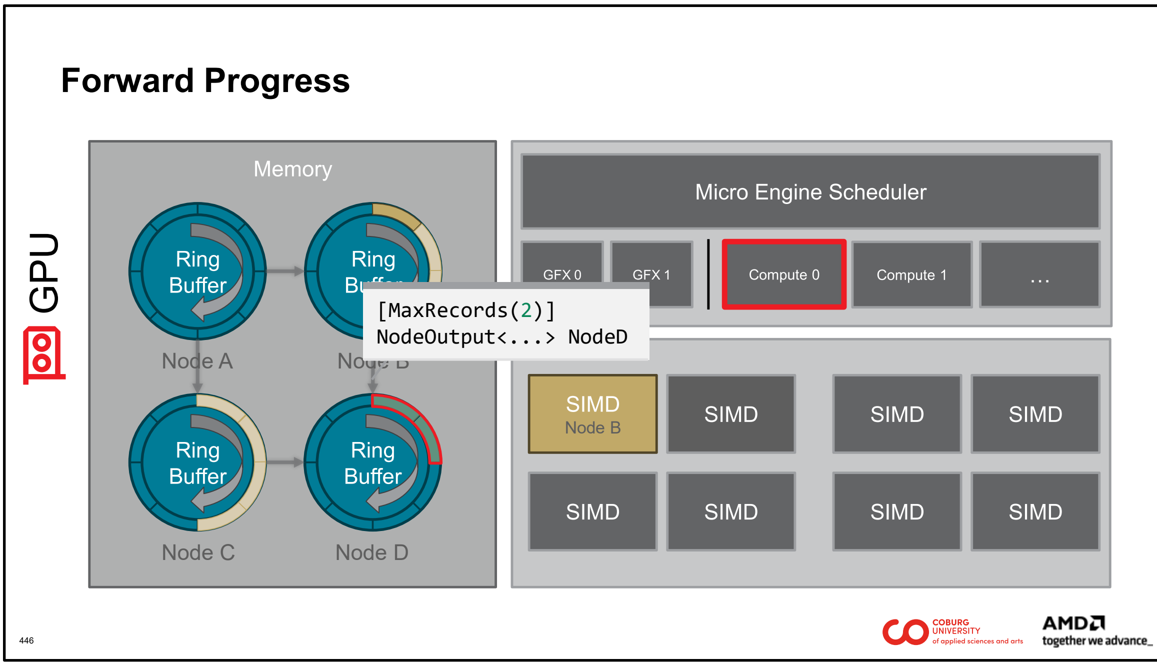
In order to avoid the deadlock from before, the work graphs runtime must ensure a *forward progress guarantee*.

What does the *forward progress guarantee* mean?

Essentially, once the graph is kicked of, it needs to be able to process *all* its records without any deadlocks.

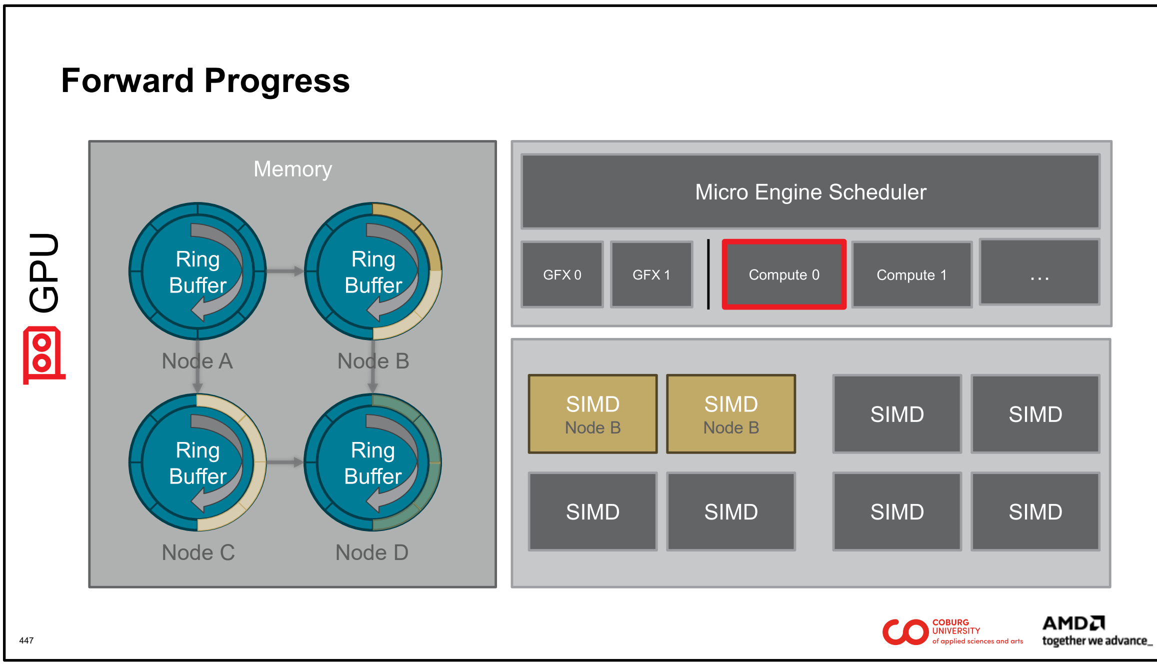


This is ensured with the output limits for each node. We've seen in the beginning of this course, that we need to annotate all outputs of a node with the maximum number of records that we intend to send.

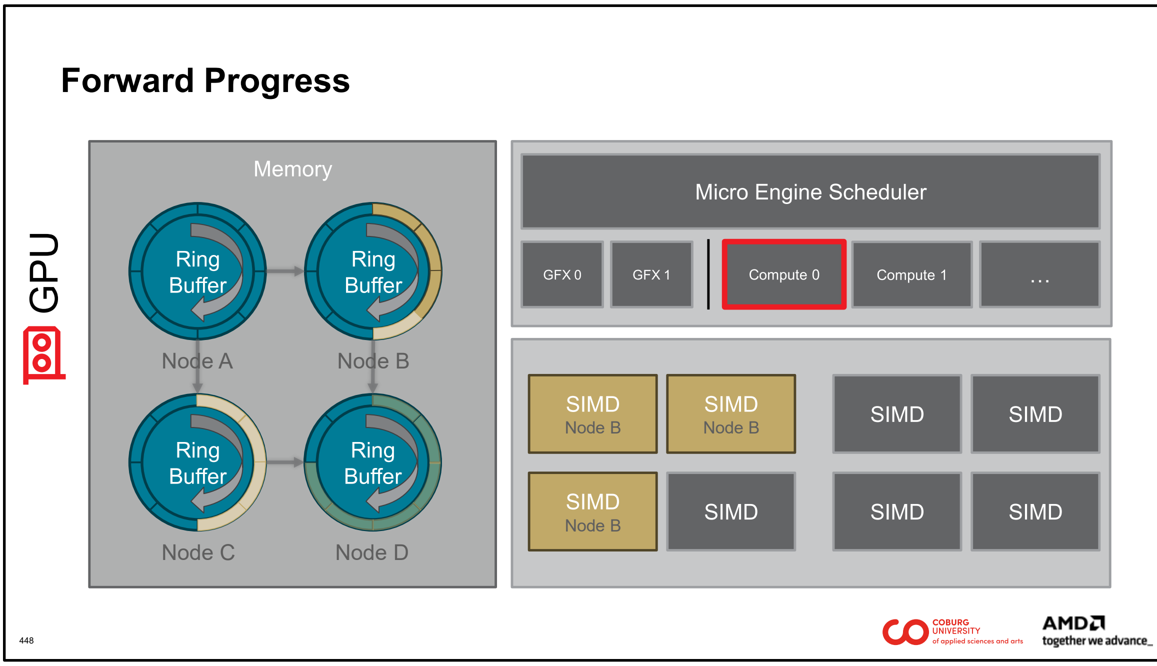


With this limit, the work graph runtime, i.e. the firmware running on the compute queue, can then make a reservation into the ring buffer of node D.

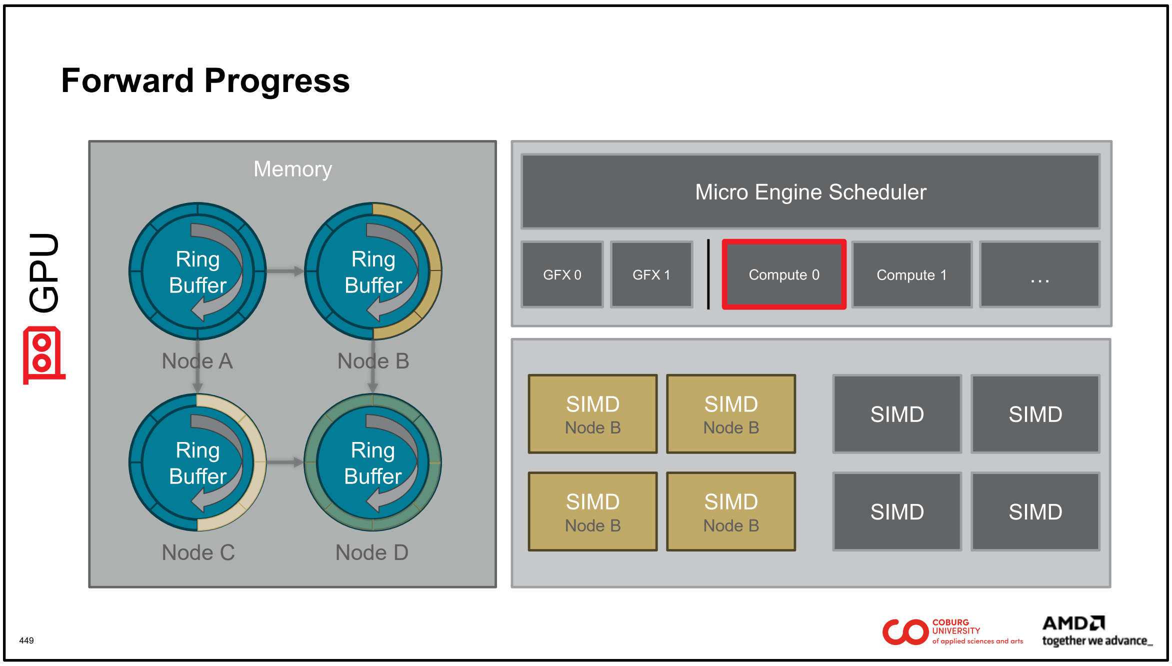
As each thread group of node B can send up to two records to node D, the compute queue reserves the first two slots in the ring buffer of node D.



This continues for the second record in the ring buffer of node B, thus the command queue reserved two more slots in the ring buffer of node D.



The same process happens for the third…

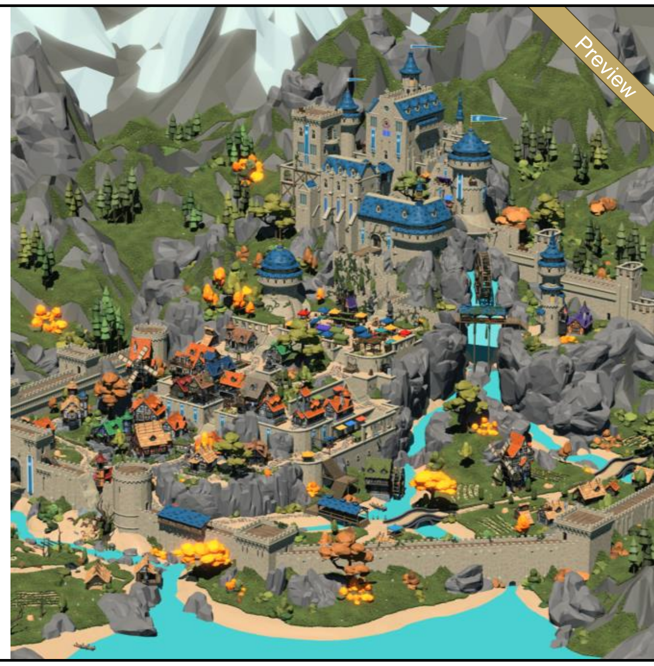


…and fourth record in the ring buffer of node B.

Now, the ring buffer of node D is full with output reservation of all the thread groups of node B. This guarantees that every one of these thread groups can write up to two records into the ring buffer of node D without overflowing the ring buffer.
On the other hand, this also means that we cannot launch any further thread groups that can produce records for node D. In our example, we cannot launch the four records available in the ring buffer of node C.
As you can see, this forward progress guarantee can impact the overall GPU occupancy. This can be solved by choosing appropriate sizes for the ring buffers.

So, now we've seen how launching new work directly from the GPU can work. We've seen the challenges that come along with this and we've seen how the work graph runtime can avoid deadlocks, whilst operating with limited resources.

Mesh Nodes



450

But so far, we’ve only looked at the compute-only node and how the compute queues of the command processor execute the work graph.

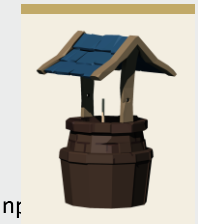
But what about mesh nodes?

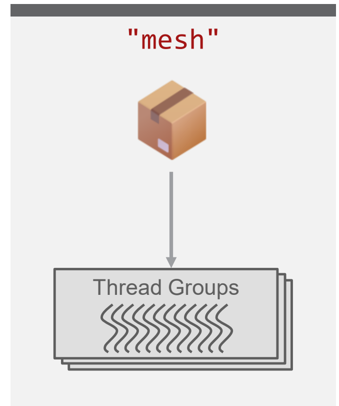
Preview

Mesh Nodes

mesh-node-tutorial-0/HelloMeshNodes.hlsl

```
[Shader("node")]
[NodeLaunch("mesh")]
[NodeId("LineMeshNode", 0)]
[NodeDispatchGrid(1, 1, 1)]
[NumThreads(32, 1, 1)]
[OutputTopology("triangle")]
void LineMeshShader(
    DispatchNodeInputRecord<Line> inp

    out indices  uint3  outputIndices[4],
    out vertices Vertex outputVertices[6])
{
    ...
}
```



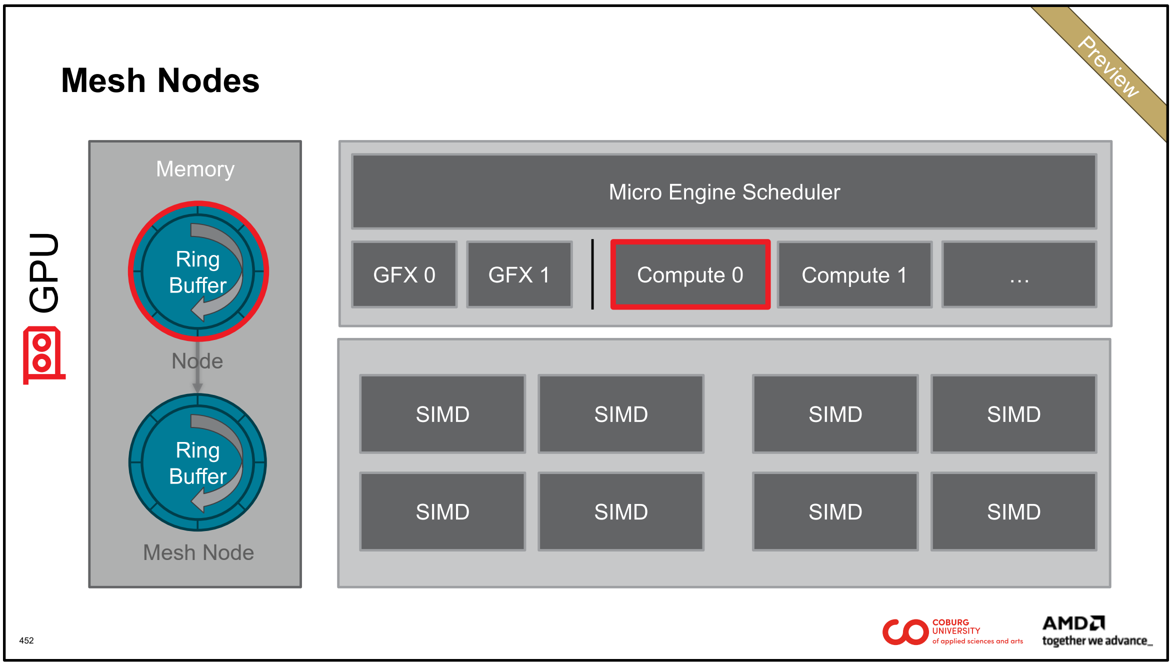


451

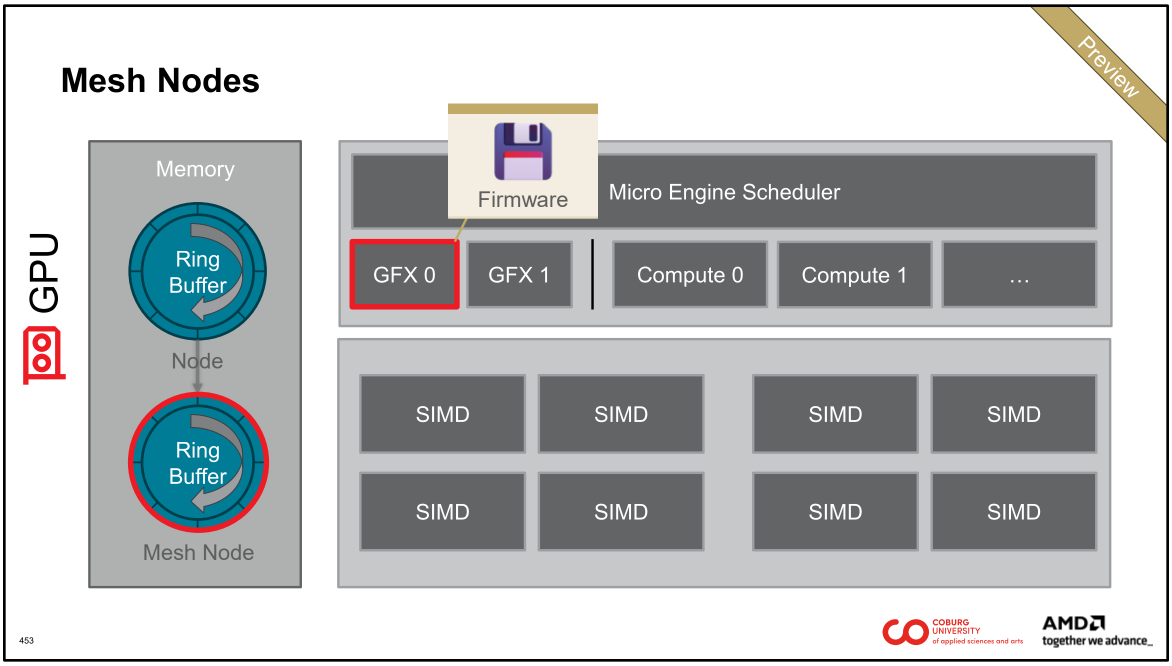
As a reminder, with mesh nodes you can directly output primitives to the rasterizer.

Mesh Nodes consist of a mesh shader, an optional pixel shader, and all other state associated with a pipeline state. The mesh shader is almost identical to the mesh shading pipeline.

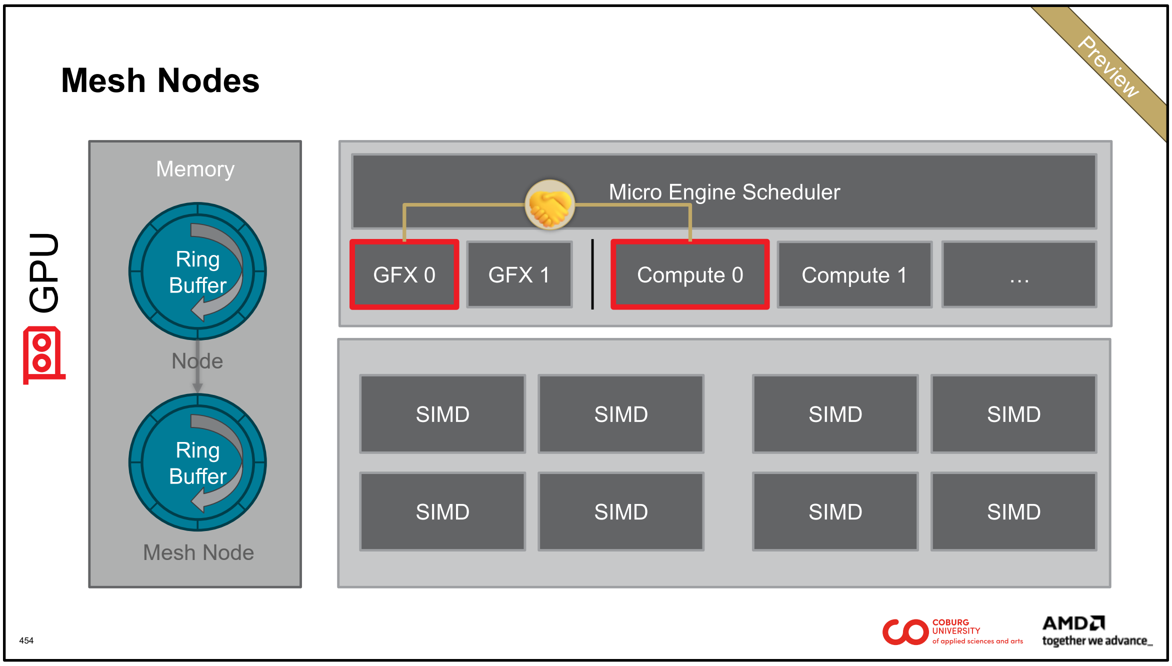
Mesh Nodes come with a new launch mode `"mesh"` that works the same as broadcasting launch mode. That means a grid of thread groups is launched. Each thread group outputs a *meshlet*, i.e., a small mesh consisting of a vertex buffer and an index buffer. This one gets then passed to the rasterizer. The Mesh Node must, however, not output any records. Therefore, a Mesh Node is bound to be a *leaf node* of the Work Graph.



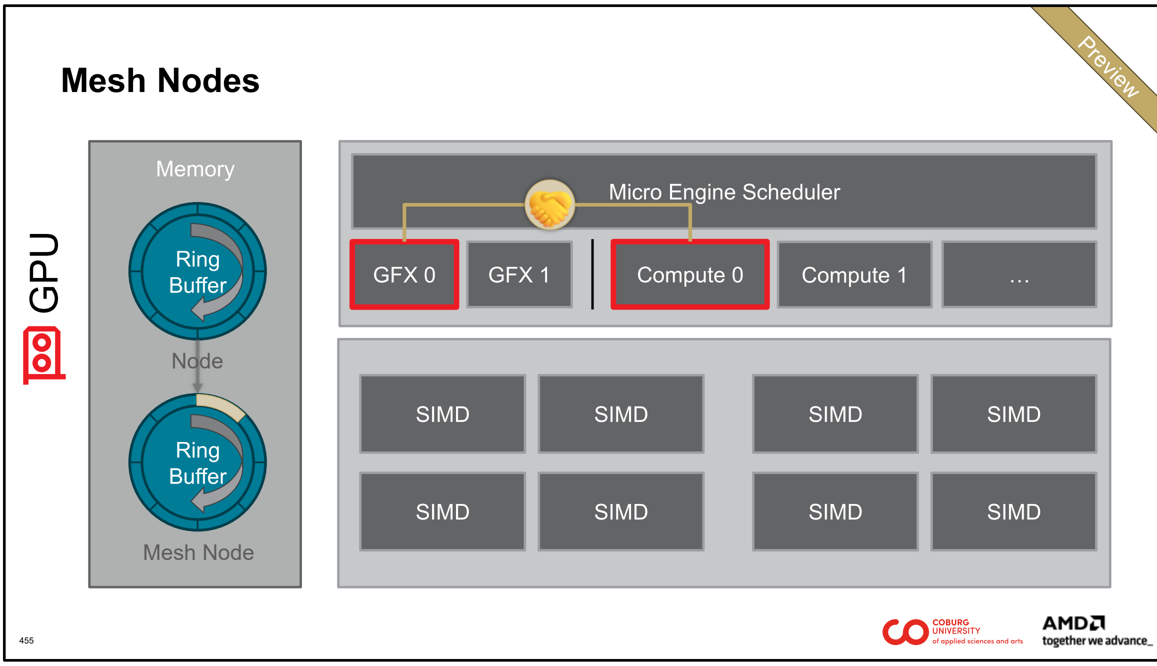
One limitation of the compute queues, which we used before for compute nodes, is that they cannot set up the graphics state, which is required for launching a mesh node.



Therefore, we need to use a graphics queue for mesh nodes. Graphics queues are also programmed by firmware and thus can scan the ring buffers assigned to mesh nodes in GPU memory.

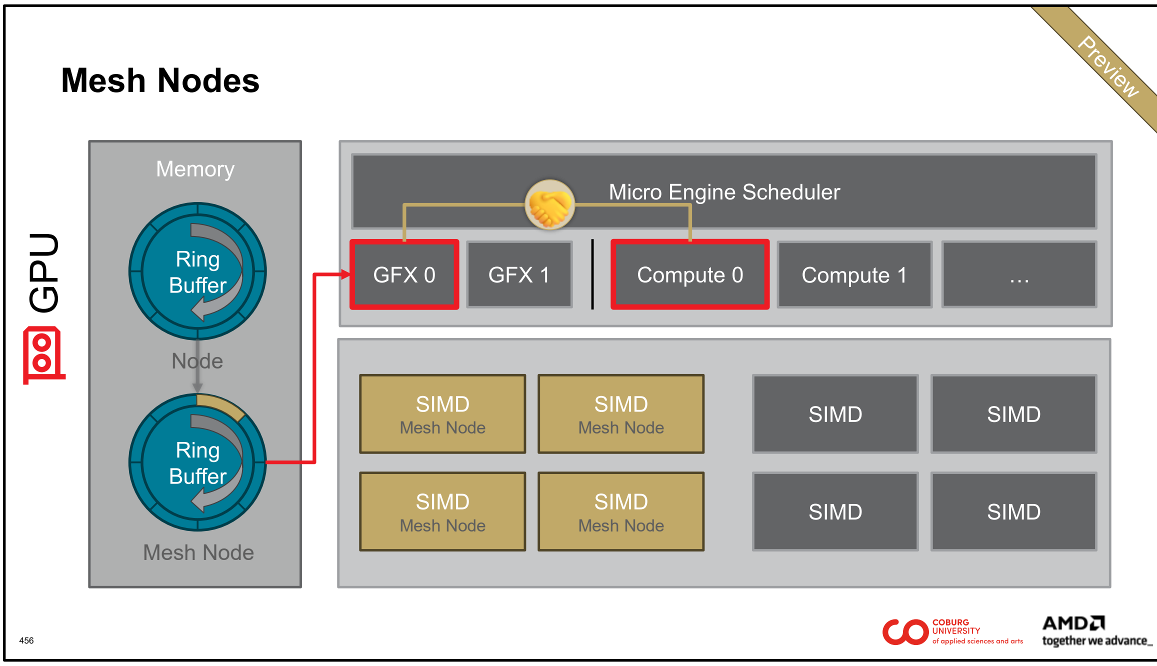


To allow the graphics and compute queue share the work load of a work graph, we can change the Micro Engine Scheduler command to a so called *gang submit*. This joins up a graphics (GFX 0) and compute queue. They can now work together on processing the records.

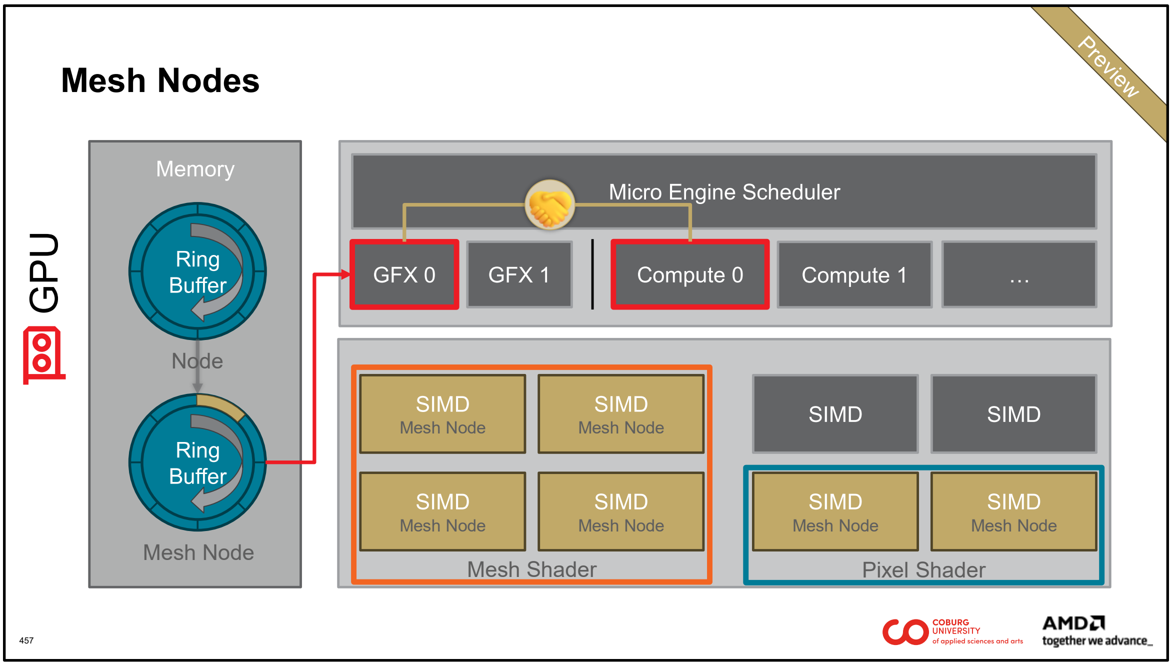


Graphics Queue 0 (GFX 0) can then scan the ring buffer for the mesh node(s) (here shown as the ring buffer on the bottom) and launch mesh shader thread groups.

To launch a mesh node, the graphics queue will also set up the graphics state (e.g. back-face culling or blend state) for each different mesh node. Thus, with a single `DispatchGraph` you can now switch between Pipeline State Objects. This is something that you couldn't do before with a regular draw command.

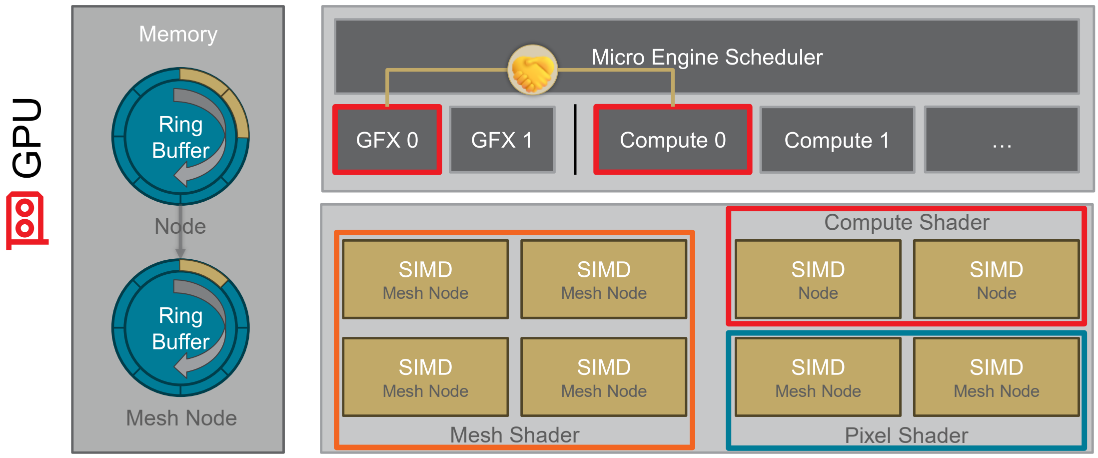


In this example, GFX0 scanned the ring buffer for “Mesh Node”, found one record, and launched four thread groups for the mesh node.

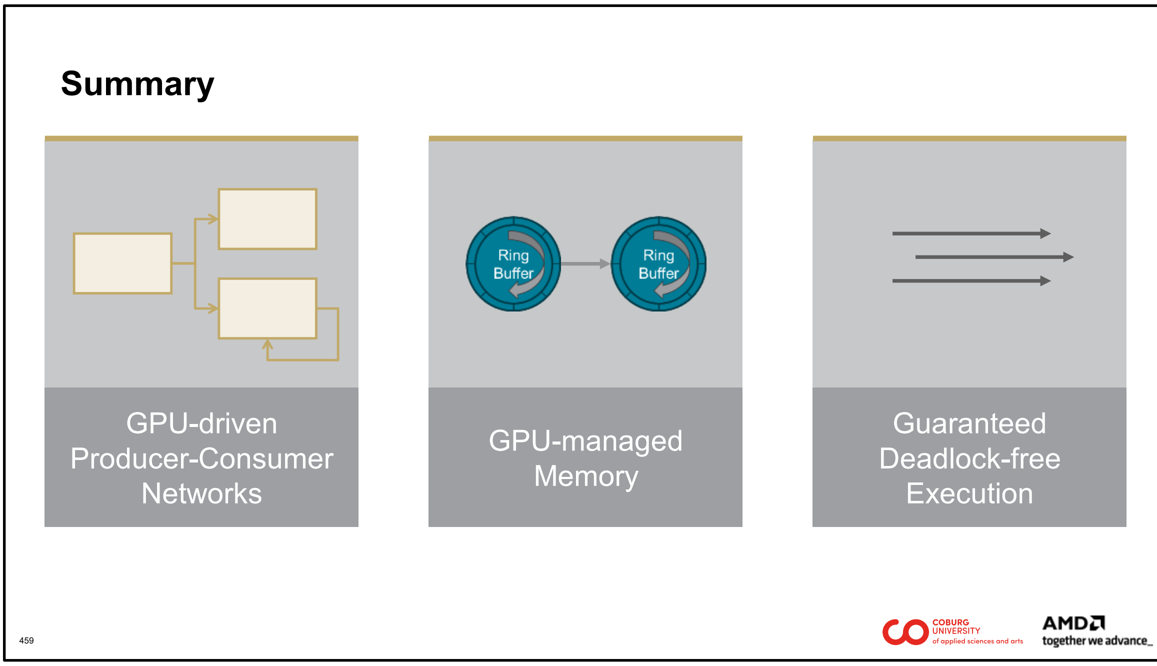


With this, we can have mesh nodes (with their mesh and pixel shaders) and "regular" compute nodes running in parallel.

Mesh Nodes



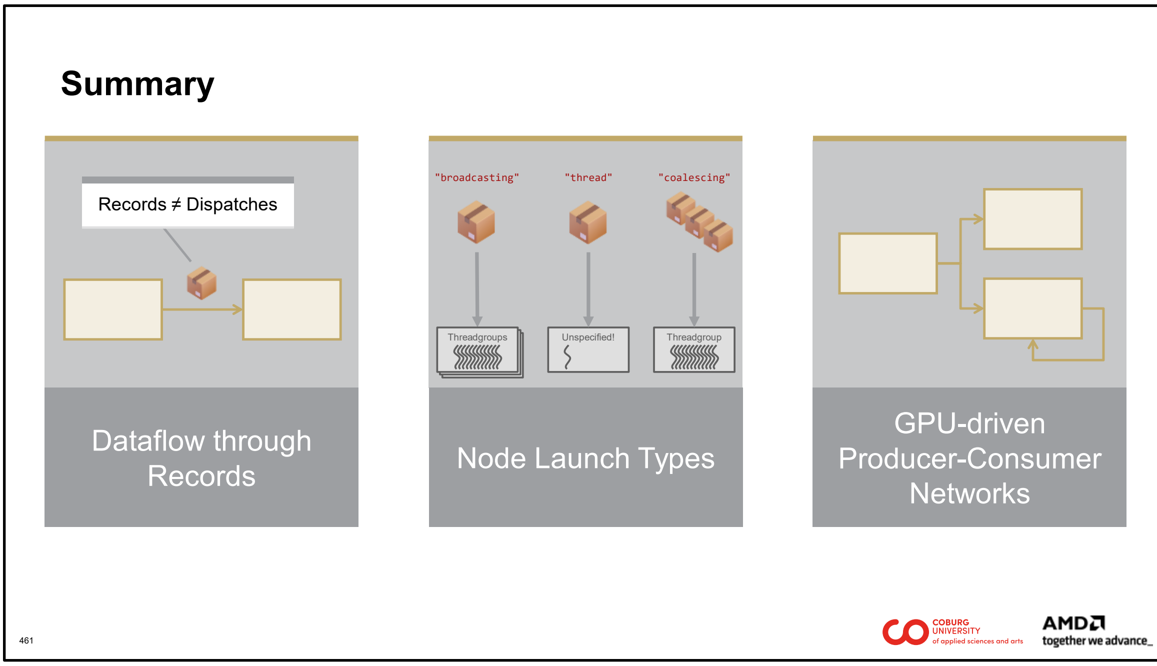
Thus, the GPU can feed itself enough work to completely fill it, all without any *barriers* or other involvement from the CPU.



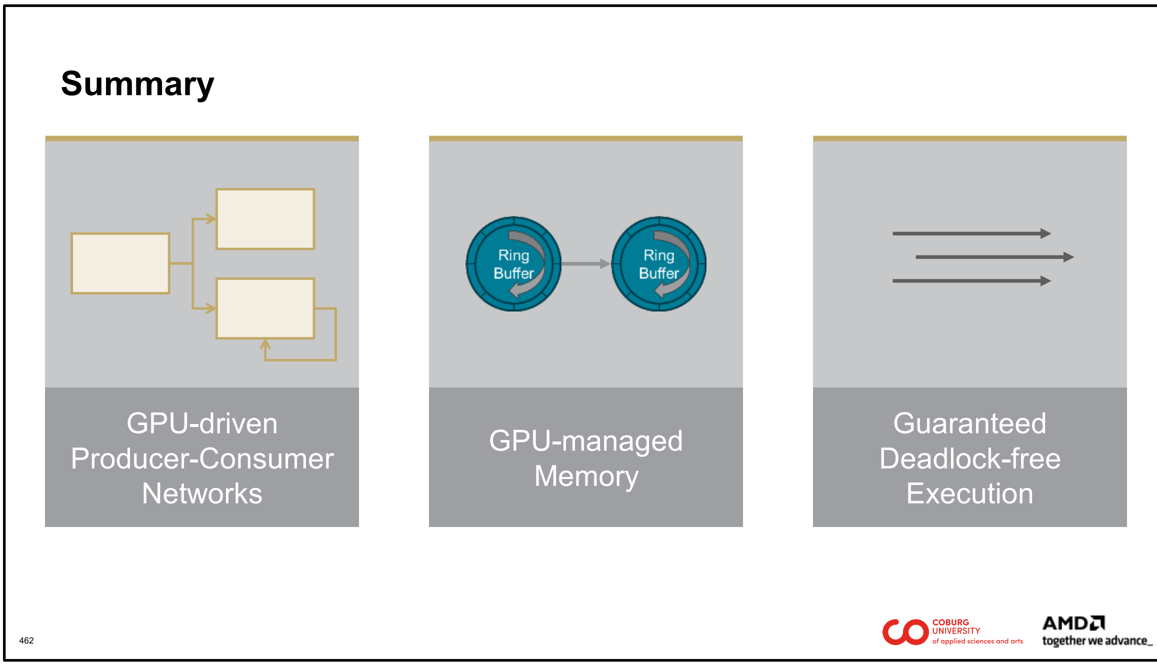
So this concludes our advanced session. With Work Graphs, we have an entirely GPU-driven Producer-Consumer Network that you as programmer can specify using a shading language. The advantage is the memory management is handled by the Work Graphs system, while also guaranteeing you a deadlock-free execution.

Summary

We have seen Work Graphs, its core concepts, and exciting applications.

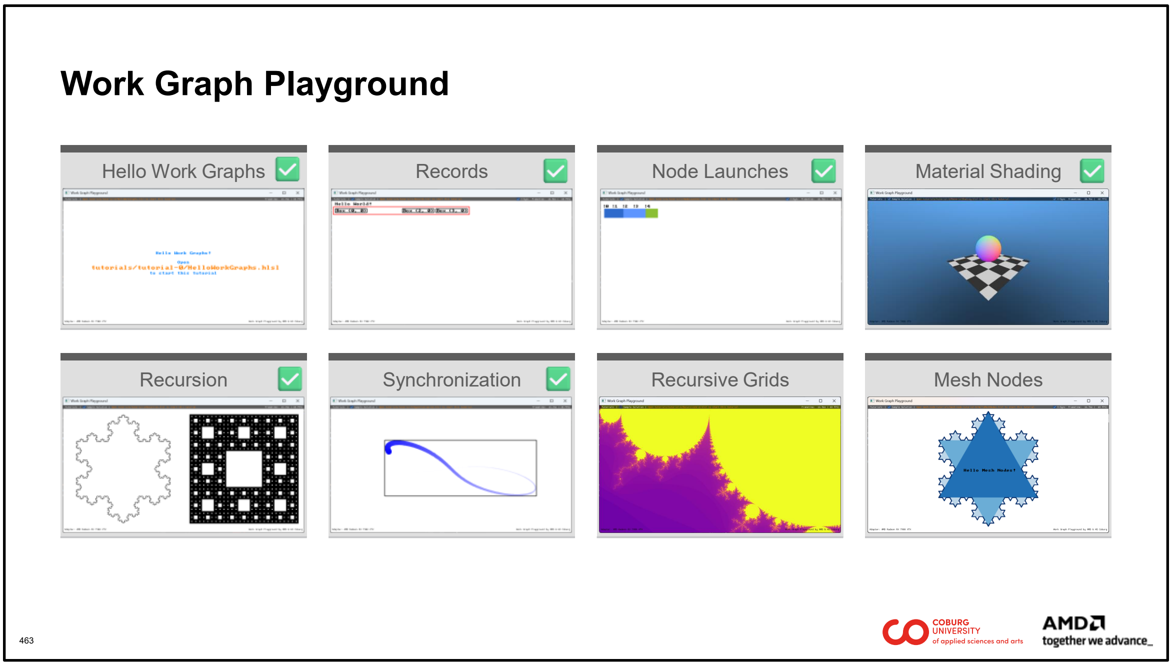


With Work Graphs you model data flow through a directed acyclic graph with trivial self-recursive cycles. The data flow is represented by records that you send from one node to another. Records are not dispatches, but eventually trigger dispatches. The specifics of these dispatches are specified by one of three different launch modes.



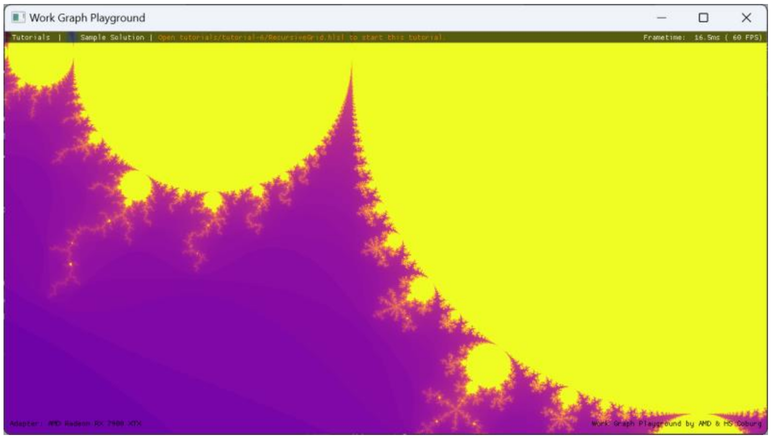
Combining all these concepts gives you a producer-consumer network running entirely on the GPU.

The memory for these records is managed by the Work Graphs system. Further, the Work Graphs system guarantees a deadlock-free execution under limited resources.



In this course, we have walked you through the first six tutorials of our Work Graph Playground App.

Work Graph Playground



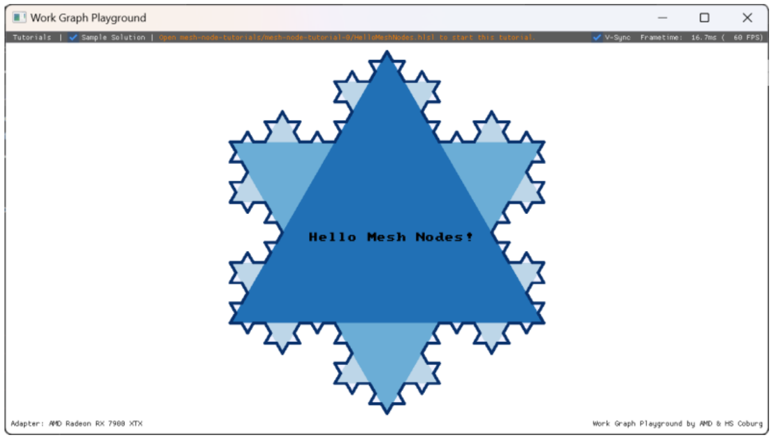
464

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

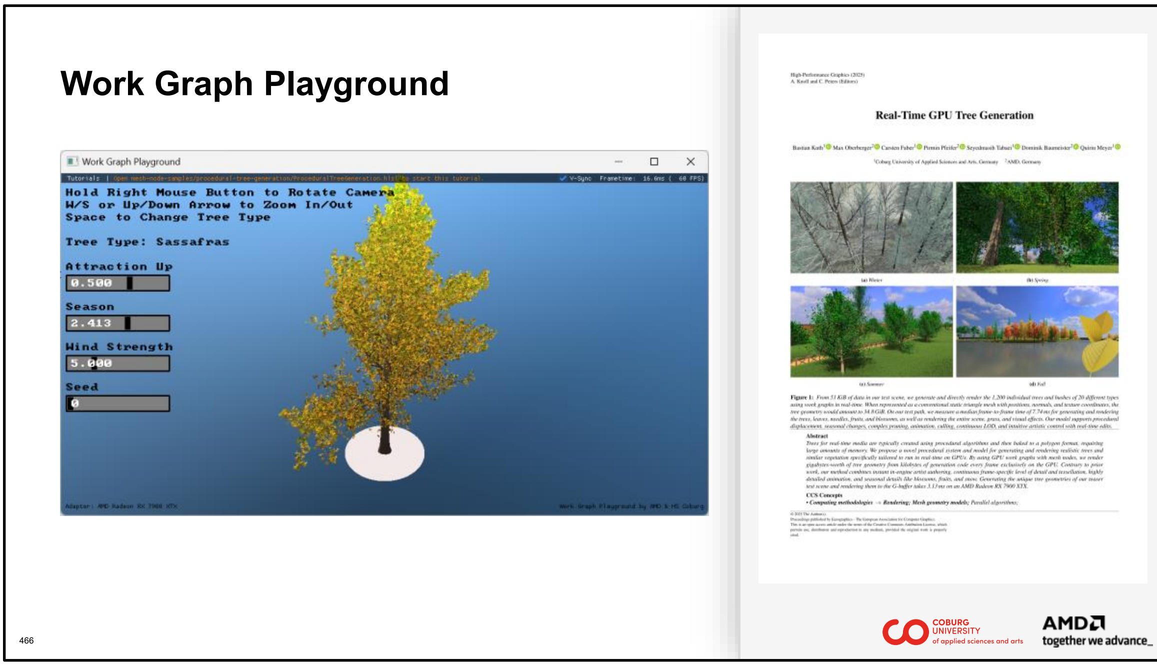
For the Recursive Grids tutorial, you’ll need to combine everything that you’ve learned so far: nodes, records, different launch modes, recursion, and synchronization.

Work Graph Playground



465

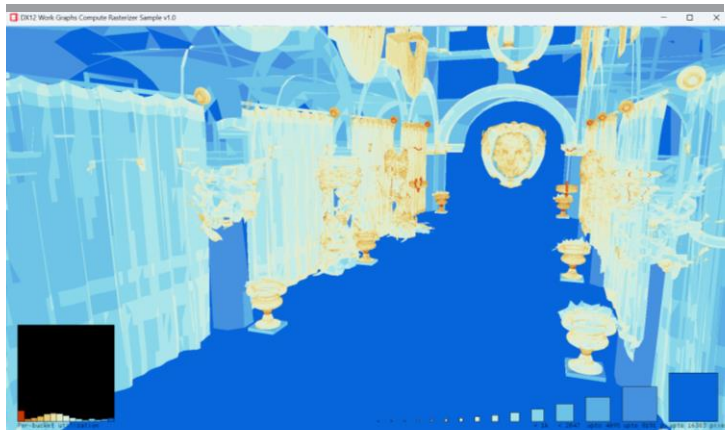
With the latest update, we’ve also added support for mesh nodes and a dedicated mesh nodes tutorial. You can find mesh-nodes enabled versions of the playground in our releases: https://github.com/GPUOpen-LibrariesAndSDKs/WorkGraphPlayground/releases



We've also released a more complex sample for our procedural tree generation. This sample runs in the Work Graph Playground App.

You can find the sample source code here: https://github.com/Bloodwyn/gptree

Samples



467

We also have more standalone samples available on GPUOpen.

For example, you can find this compute rasterizer example here: https://gpuopen.com/learn/work_graphs_learning_sample/

If you're interested in procedural generation with mesh nodes, we have additional samples available here: https://github.com/search?q=topic%3Ameshnodes+org%3AGPUOpen-LibrariesAndSDKs&type=repositories

Join the gpu-work-graphs channel on the AMD Developer Community Discord server at https://discord.gg/amd-dev

Thank you!

Big thanks also go out to:

- Carsten Faber and Seyedmasih Tabaei from the Coburg University
- the whole team at AMD, especially Dominik Baumeister, Niels Fröhling, Pirmin Pfeifer and many more
- Matthäus Chajdas

470

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

This concludes our course today. Big thanks go out to our undergraduate and graduate students at Coburg University, the Work Graphs team at AMD, and Matthäus.

References

- Mark J. Kilgard. 1999. *NV_register_combiners*. Khronos Group
- Lindholm, Mark J. Kilgard, and Henry Moreton. 2001. *A user-programmable vertex engine*. In Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01). ACM, New York, NY, USA, 149–158
- Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. 2018. *Real-Time Rendering*, Fourth Edition (4th ed.). A. K. Peters, Ltd., USA
- David Blythe. 2006. *The Direct3D 10 system*. ACM Trans. Graph. 25, 3 (jul 2006), 724–734.
- Jeff Andrews and Nick Baker. 2006. *Xbox 360 System Architecture*. IEEE Micro 26, 2 (2006), 25–37
- M. Nießner, B. Keinert, M. Fisher, M. Stamminger, C. Loop, and H. Schäfer. 2016. *Real-Time Rendering Techniques with Hardware Tessellation*, Computer Graphics Forum 35, 1 (2016), 113–137
- Mark Peercy, Mark Segal, and Derek Gerstmann. 2006. *A performance-oriented data parallel virtual machine for GPUs*. In ACM SIGGRAPH2006 Sketches (Boston, Massachusetts) (SIGGRAPH '06). ACM, New York, NY, USA, 184–es 2007.
- NVIDIA, *NVIDIA CUDA Compute Unified Device Architecture*, Release 1.0 – Programming Guide. Nvidia
- Hubert Nguyen. 2007. *GPU gems 3 (first ed.)*. Addison-Wesley Professional
- Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. *OptiX: a general purpose ray tracing engine*. ACM Trans. Graph. 29, 4, Article 66 (jul2010)
- Christoph Kubisch, Pat Brown, Jeff Bolz, Daniel Koch, Piers Daniell, and Pierre Boudier. 2018. *VK_NV_mesh_shader*. Khronos Group.
- Microsoft Cooperation. 2024, *D3D12 Work Graphs*, https://microsoft.github.io/DirectX-Specs/d3d/WorkGraphs.html
- Bastian Kuth, Max Oberberger, Carsten Faber, Dominik Baumeister, Matthäus Chajdas, and Quirin Meyer. 2024. *Real-Time Procedural Generation with GPU Work Graphs*. Proc. ACM Comput. Graph. Interact. Tech. 7, 3 (Aug. 2024).
- Bastian Kuth, Max Oberberger, Carsten Faber, Pirmin Pfeifer, Seyedmasih Tabaei, Dominik Baumeister, and Quirin Meyer. 2025. *Real-Time GPU Tree Generation*. In Proceedings of High-Performance Graphics (HPG). ACM, Copenhagen, Denmark
- Oswin Aichholzer, Franz Aurenhammer, David Alberts, and Bernd Gärtner. 1995. *A novel type of skeleton for polygons*. Springer.

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_

Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED 'AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

COBURG UNIVERSITY of applied sciences and arts

AMD together we advance_