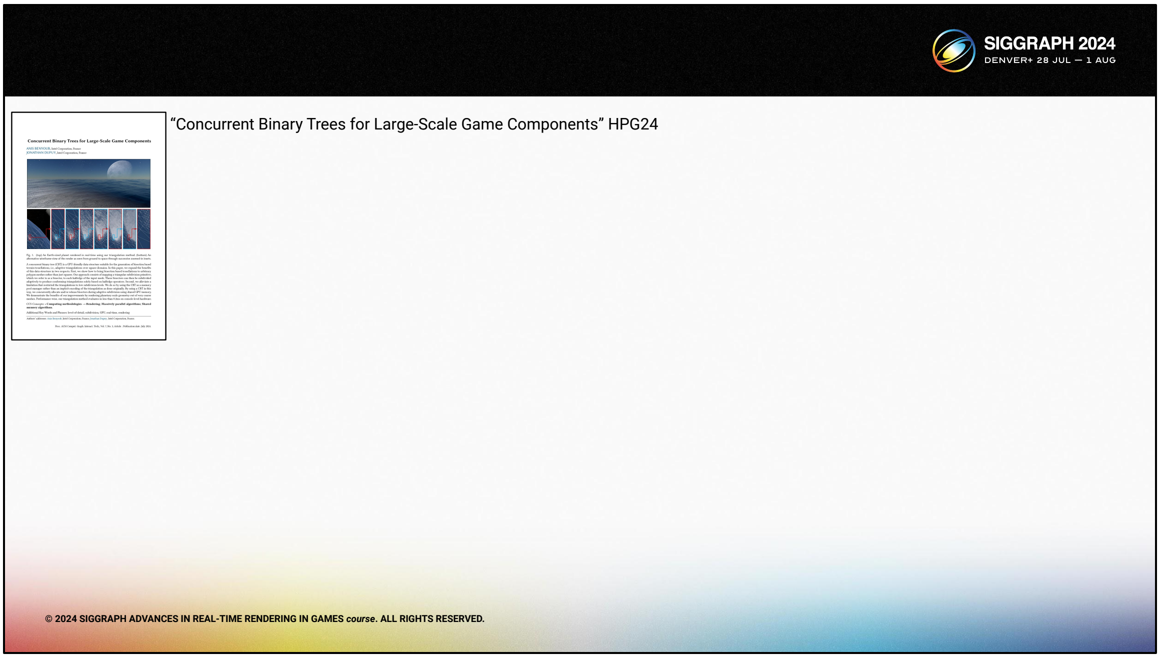
SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

THE PREMIER CONFERENCE
& EXHIBITION ON
COMPUTER GRAPHICS &
INTERACTIVE TECHNIQUES

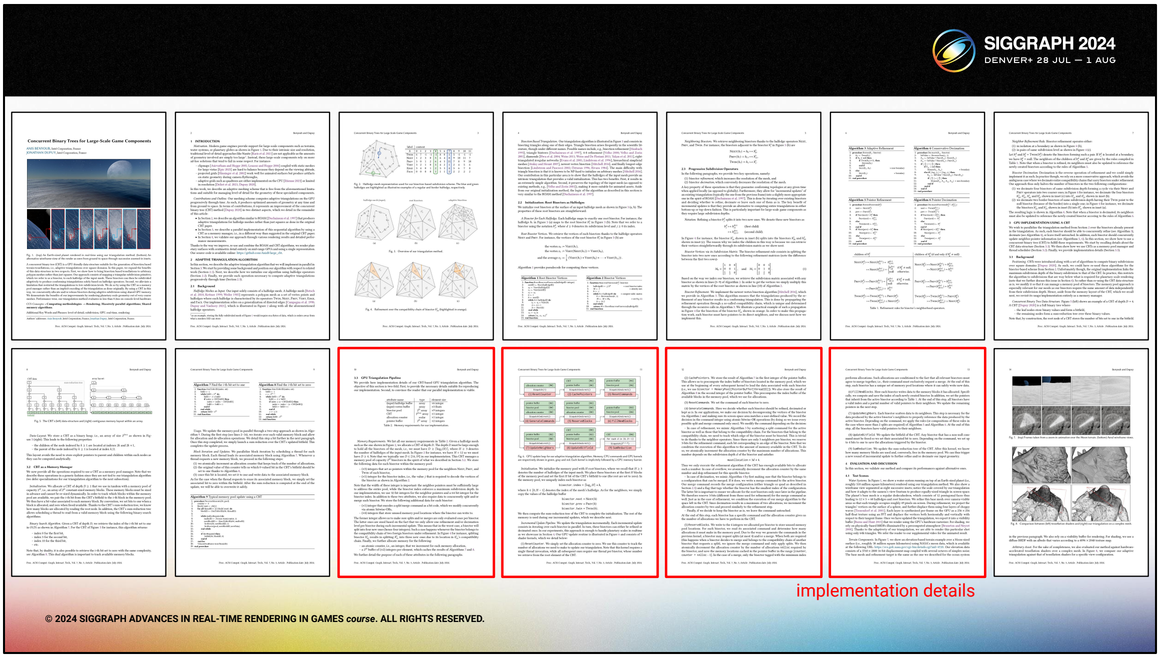
ACHIEVING SCALABLE PERFORMANCES FOR LARGE SCALE GAME COMPONENTS WITH CBTS



This presentation is a follow-up of our HPG paper that we presented just a couple of days ago.

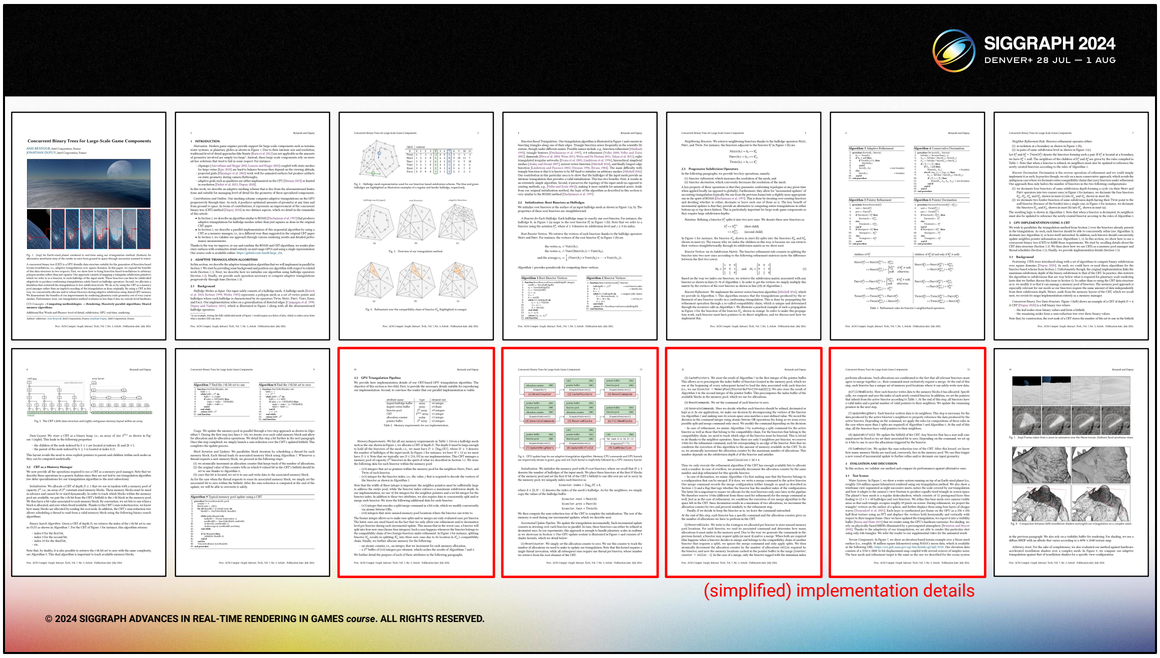
The paper is called “Concurrent Binary Trees for Large-Scale Game Components”...

…and most of the content of the paper is shown here, including…

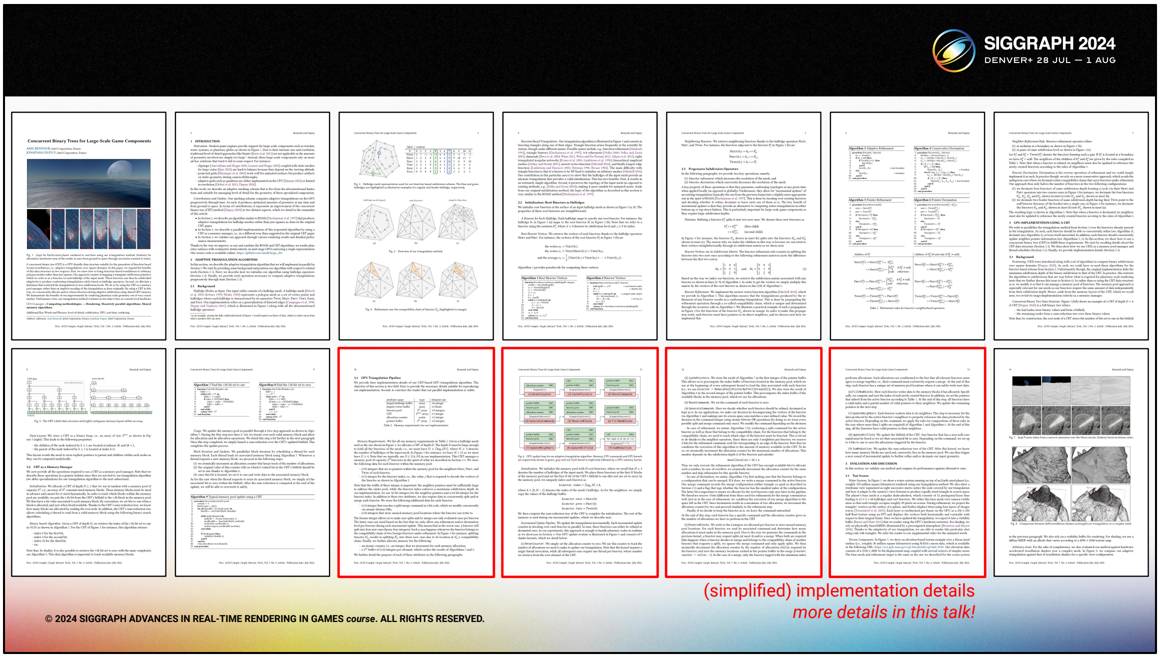


4 pages of implementation details shown in red!

The reason I'm showing these is that it turns out that since we wrote the paper 6 months ago, Anis re-worked the implementation to make it even faster.



So what we describe in the paper is now more or less deprecated already!



And so in this presentation Anis will share all the details of his new implementation.

Before he does so, I will quickly recap what the paper is about to bring everyone up to speed.

Concurrent Binary Trees for Large-Scale Game Components

ANIS BENYOUB, Intel Corporation, France
JONATHAN DUPUY, Intel Corporation, France

Fig. 1. (top) An Earth-sized planet rendered in real-time using our triangulation method. (bottom) An alternative wireframe-view of the render as seen from ground to space through successive zoomed-in insets.

A concurrent binary tree (CBT) is a GPU-friendly data-structure suitable for the generation of bisection based terrain tessellations, i.e., adaptive triangulations over square domains. In this paper, we expand the benefits of this data-structure in two respects. First, we show how to bring bisection based tessellations to arbitrary polygon meshes rather than just squares. Our approach consists of mapping a triangular subdivision primitive, which we refer to as a bisector, to each halfedge of the input mesh. These bisectors can then be subdivided adaptively to produce conforming triangulations solely based on halfedge operators. Second, we alleviate a limitation that restricted the triangulations to low subdivision levels. We do so by using the CBT as a memory pool manager rather than an implicit encoding of the triangulation as done originally. By using a CBT in this way, we concurrently allocate and/or release bisectors during adaptive subdivision using shared GPU memory. We demonstrate the benefits of our improvements by rendering planetary scale geometry out of very coarse meshes. Performance-wise, our triangulation method evaluates in less than 0.2ms on console-level hardware.

CCS Concepts: • **Computing methodologies → Rendering; Massively parallel algorithms; Shared memory algorithms.**

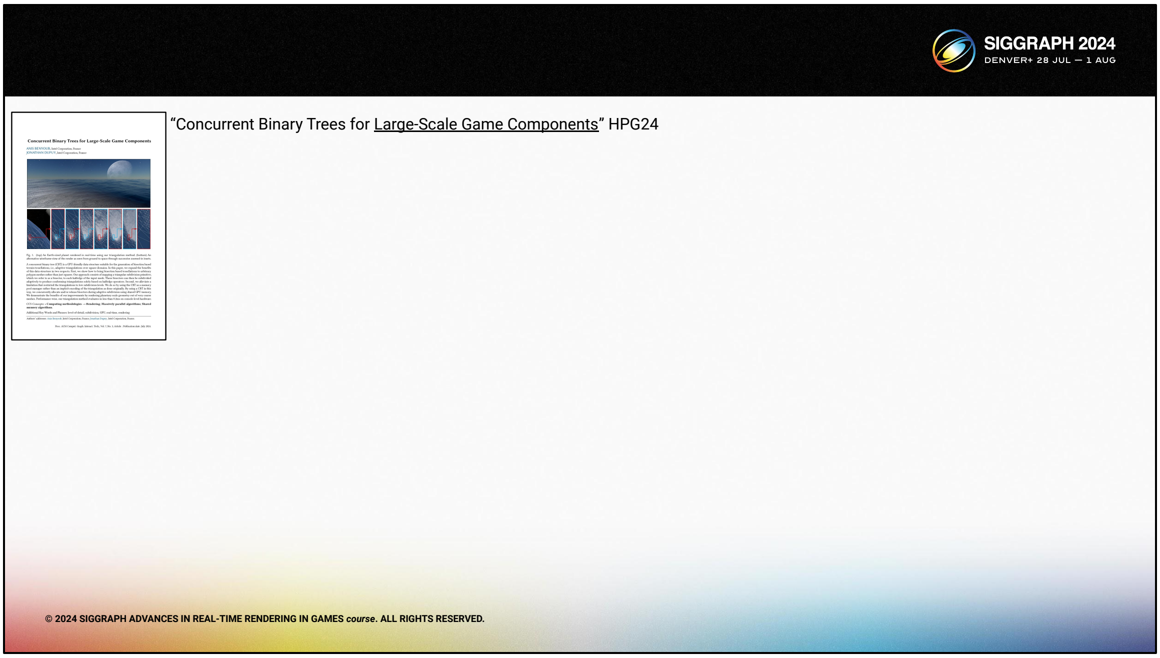
Additional Key Words and Phrases: level-of-detail, subdivision, GPU, real-time, rendering

Authors' addresses: Anis Benyoub, Intel Corporation, France; Jonathan Dupuy, Intel Corporation, France.

Proc. ACM Comput. Graph. Interact. Tech., Vol. 7, No. 3, Article . Publication date: July 2024.

“Concurrent Binary Trees for Large-Scale Game Components” HPG24

The paper introduces a new algorithm to deal with large-scale game components.



A large-scale game component is typically what makes the virtual world of your game look "big".

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

"Concurrent Binary Trees for Large-Scale Game Components" HPG24

terrains

Red Dead Redemption 2
Rockstar

Taking a few video games as example, that would be terrains, …

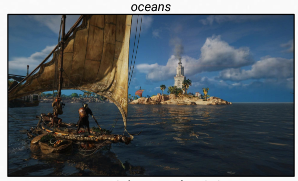
SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

"Concurrent Binary Trees for Large-Scale Game Components" HPG24

terrains

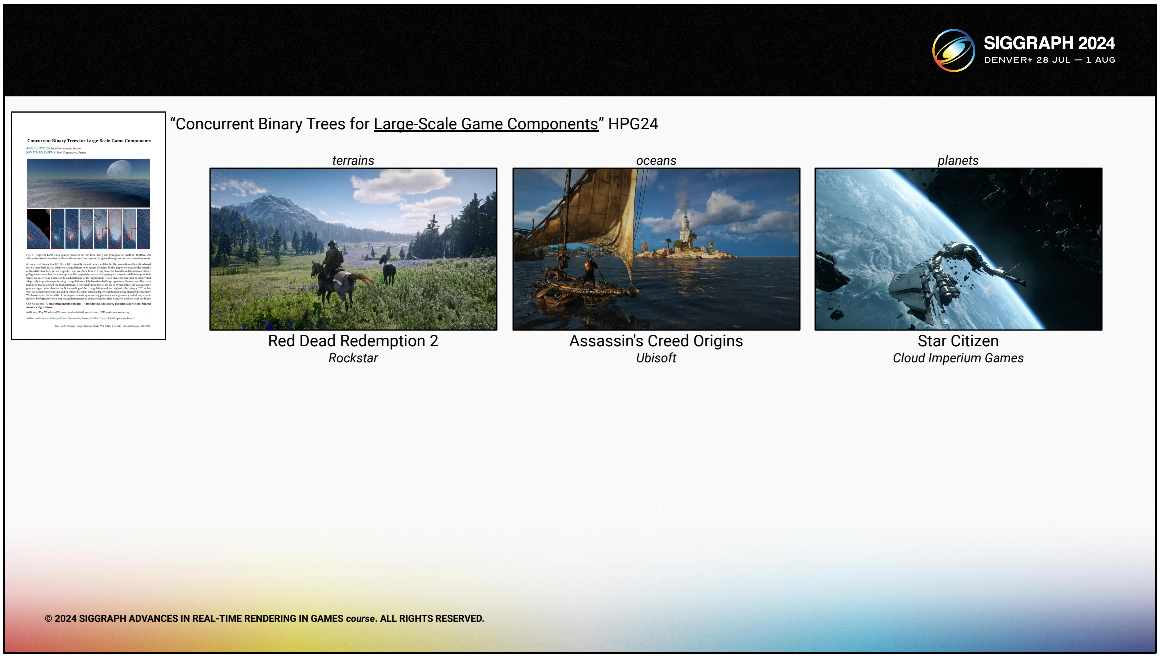
Red Dead Redemption 2
Rockstar

oceans



Assassin's Creed Origins
Ubisoft

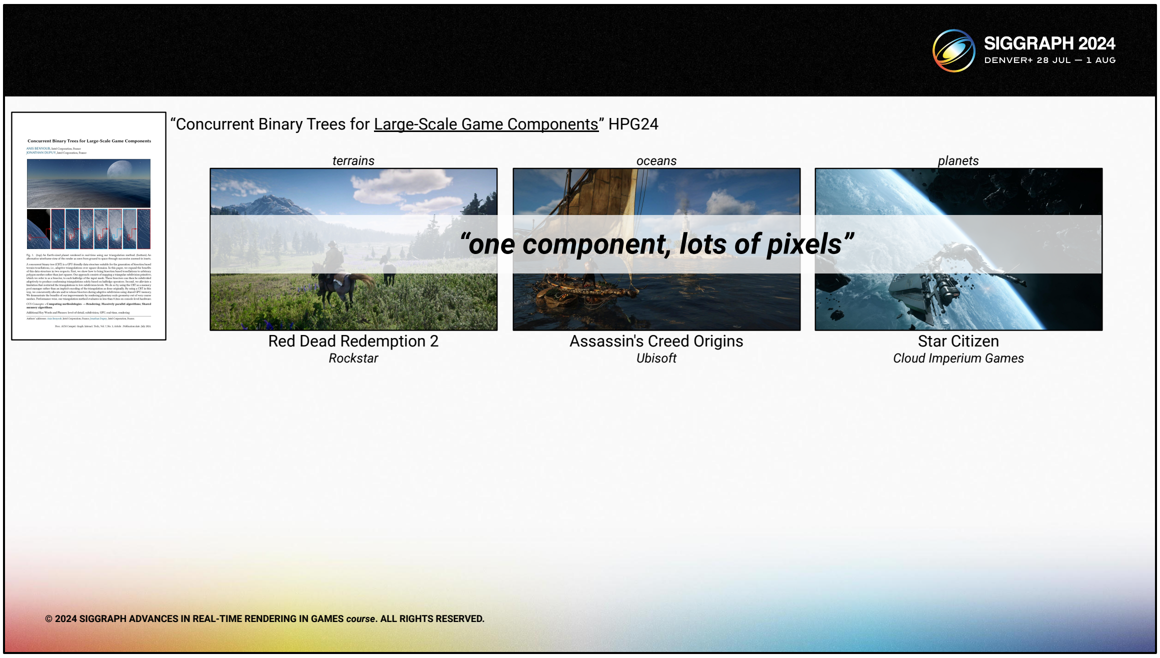
…, oceans, …



… or entire planets.

In each of these screenshots, pretty much everything except the characters in the foreground is rendered using a dedicated system.

This dedicated system is what we refer to a “large-scale game component”.



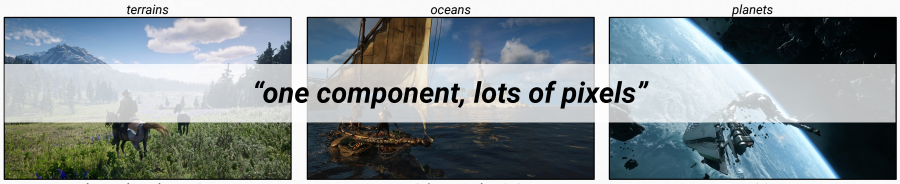
Most of the time, such components occupy a lot of pixels so it’s important to have a set of efficient algorithms to render them as fast as possible.

The goal of our paper is to contribute to this set of efficient algorithms, …

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

“Concurrent Binary Trees for Large-Scale Game Components” HPG24

terrains *oceans* *planets*



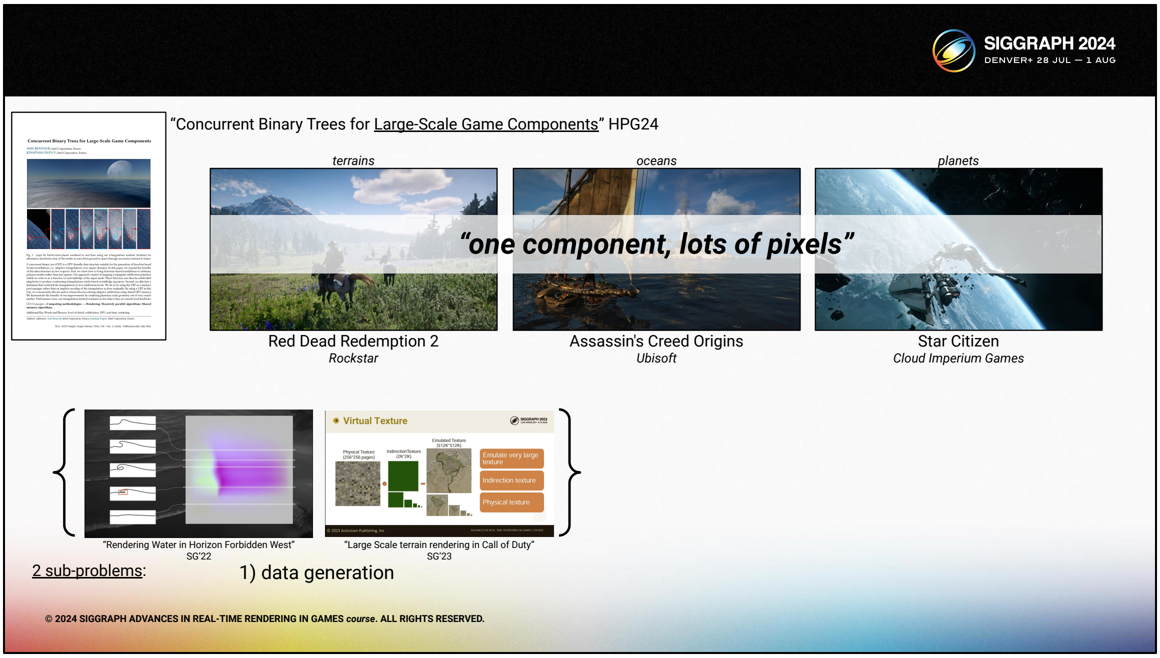
Red Dead Redemption 2
Rockstar

Assassin's Creed Origins
Ubisoft

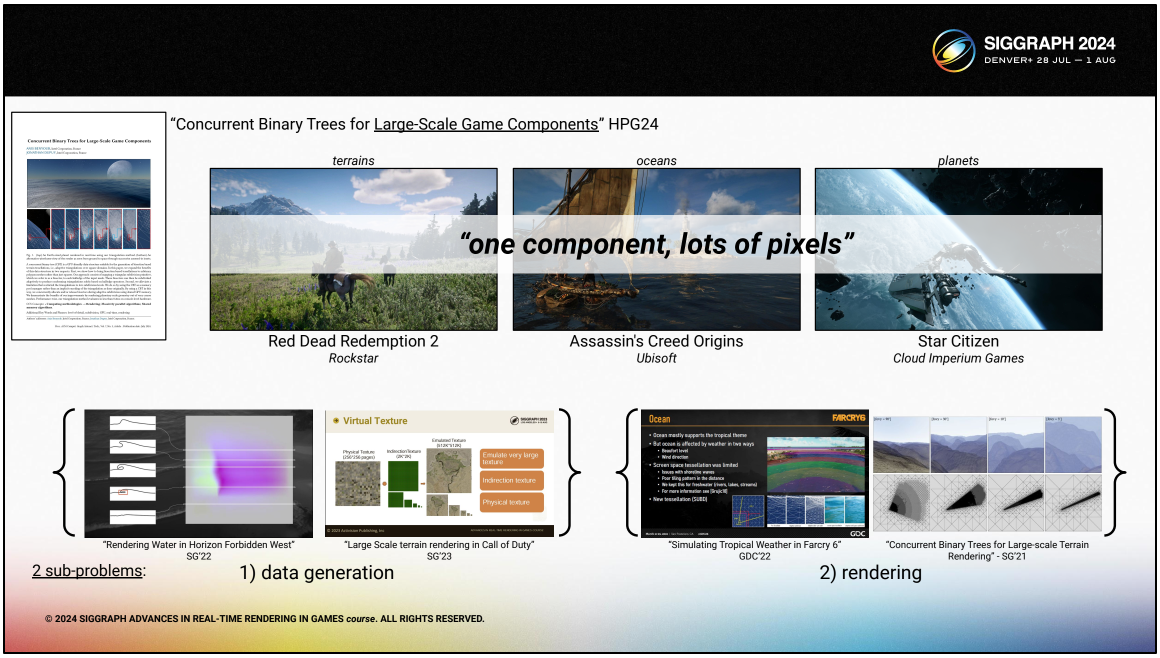
Star Citizen
Cloud Imperium Games

2 sub-problems:

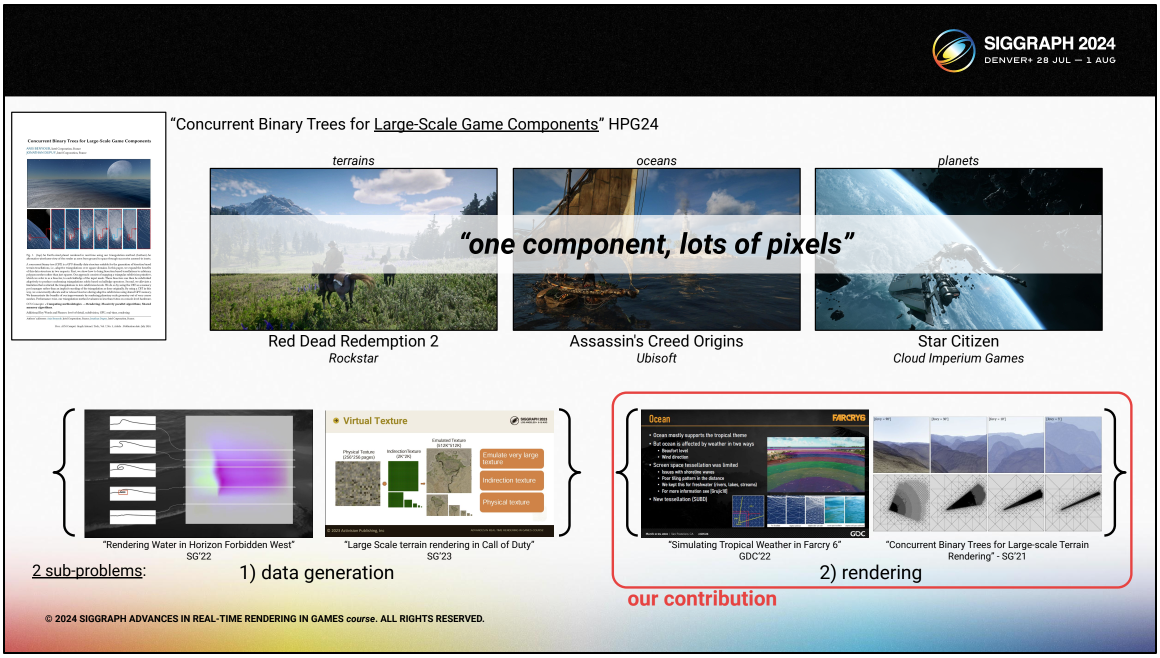
…which can typically be classified into two categories.



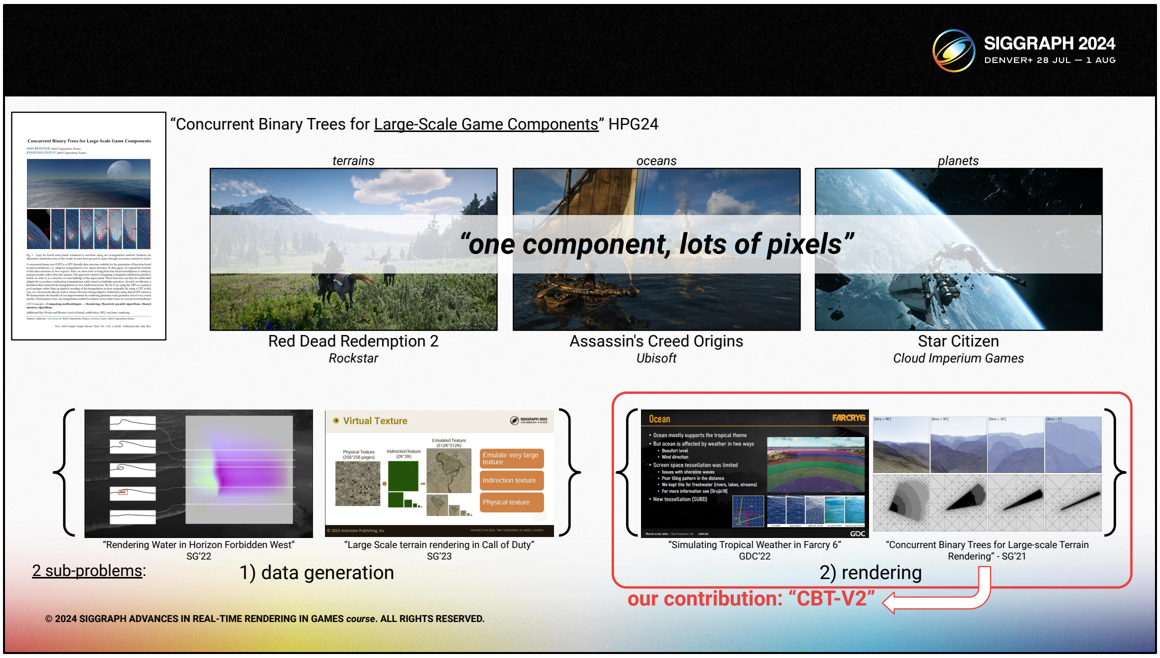
First there’s the data-generation category, which addresses how to generate textures, sprites, instances, etc.



Second the triangulation / rendering category, which focuses on how to render this data.

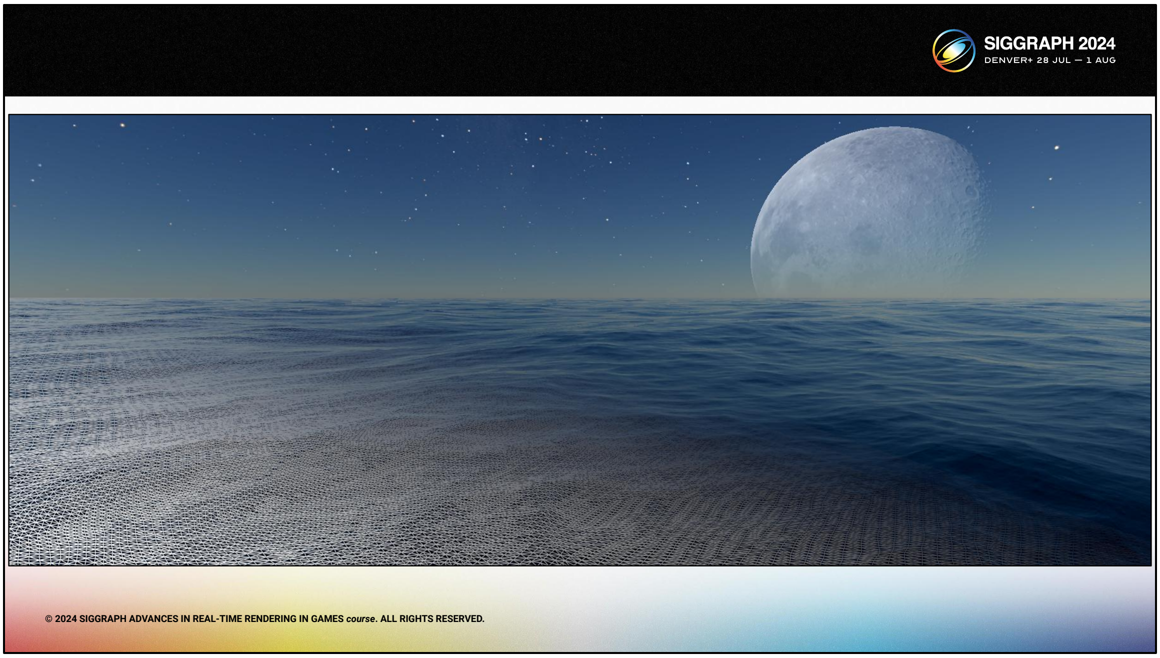


Our paper contributes only to the latter (so we won't be discussing data-generation here) with a triangulation method capable of handling very large environments that can be explored at any different scales.



And actually, our contribution is an improvement over something called "Concurrent Binary Trees" (CBTs), which we presented in the same course 3 years ago.

I like to refer to our improvement as "CBT version 2", or simply CBT-V2. I will explain what CBTs are in just a minute.



Before I do just that, here is a look at what CBT-V2 can render.

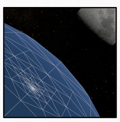
Let's have a look at a video that captures the result of our method, which runs at 250+FPS on a PS5 level hardware at full HD resolution.


Concurrent Binary Trees for Large-Scale Game Components

“Concurrent Binary Trees for Large-Scale Game Components” HPG24

To provide a better sense of scale of what our CBTV2 produces, we are going to have a look at how dense the triangulation of this shot is.

“Concurrent Binary Trees for Large-Scale Game Components” HPG24



Here is an alternative view of the same shot and we are going to zoom into it until we reach the resolution of the mesh

Concurrent Binary Trees for Large-Scale Game Components

ANIS BENYOUB, Intel Corporation, France
JONATHAN DUPUY, Intel Corporation, France

Fig. 1. (top) An Earth-sized planet rendered in real-time using our triangulation method. (bottom) An alternative wireframe-view of the render as seen from ground to space through successive zoomed-in insets.

A concurrent binary tree (CBT) is a GPU-friendly data-structure suitable for the generation of bisection based terrain tessellations, i.e., adaptive triangulations over square domains. In this paper, we expand the benefits of this data-structure in two respects. First, we show how to bring bisection based tessellations to arbitrary polygon meshes rather than just squares. Our approach consists of mapping a triangular subdivision primitive, which we refer to as a bisector, to each halfedge of the input mesh. These bisectors can then be subdivided adaptively to produce conforming triangulations solely based on halfedge operators. Second, we alleviate a limitation that restricted the triangulations to low subdivision levels. We do so by using the CBT as a memory pool manager rather than an implicit encoding of the triangulation as done originally. By using a CBT in this way, we concurrently allocate and/or release bisectors during adaptive subdivision using shared GPU memory. We demonstrate the benefits of our improvements by rendering planetary scale geometry out of very coarse meshes. Performance-wise, our triangulation method evaluates in less than 0.2ms on console-level hardware.

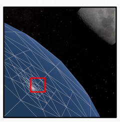
CCS Concepts: • **Computing methodologies → Rendering; Massively parallel algorithms; Shared memory algorithms.**

Additional Key Words and Phrases: level-of-detail, subdivision, GPU, real-time, rendering

Authors' addresses: Anis Benyoub, Intel Corporation, France; Jonathan Dupuy, Intel Corporation, France.

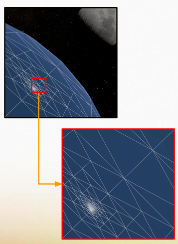
Proc. ACM Comput. Graph. Interact. Tech., Vol. 7, No. 3, Article . Publication date: July 2024.

"Concurrent Binary Trees for Large-Scale Game Components" HPG24



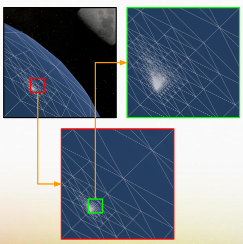

Concurrent Binary Trees for Large-Scale Game Components

“Concurrent Binary Trees for Large-Scale Game Components” HPG24



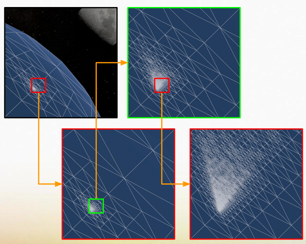

Concurrent Binary Trees for Large-Scale Game Components

“Concurrent Binary Trees for Large-Scale Game Components” HPG24



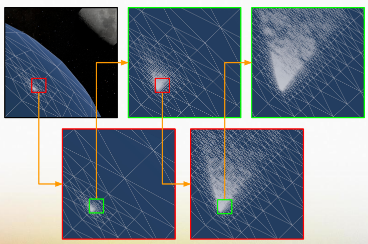

Concurrent Binary Trees for Large-Scale Game Components

"Concurrent Binary Trees for Large-Scale Game Components" HPG24

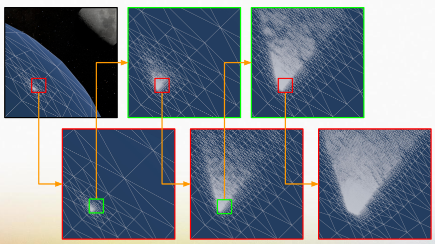



Concurrent Binary Trees for Large-Scale Game Components

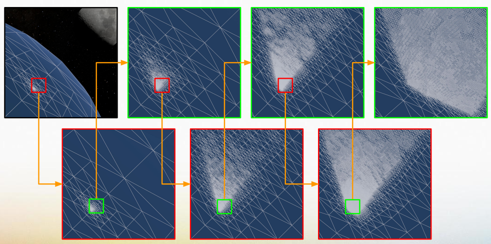
"Concurrent Binary Trees for Large-Scale Game Components" HPG24



"Concurrent Binary Trees for Large-Scale Game Components" HPG24

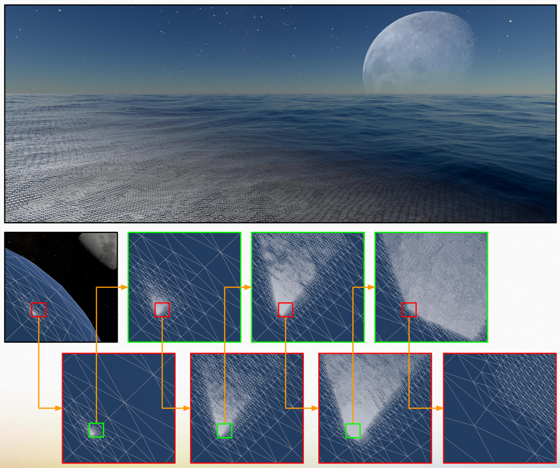


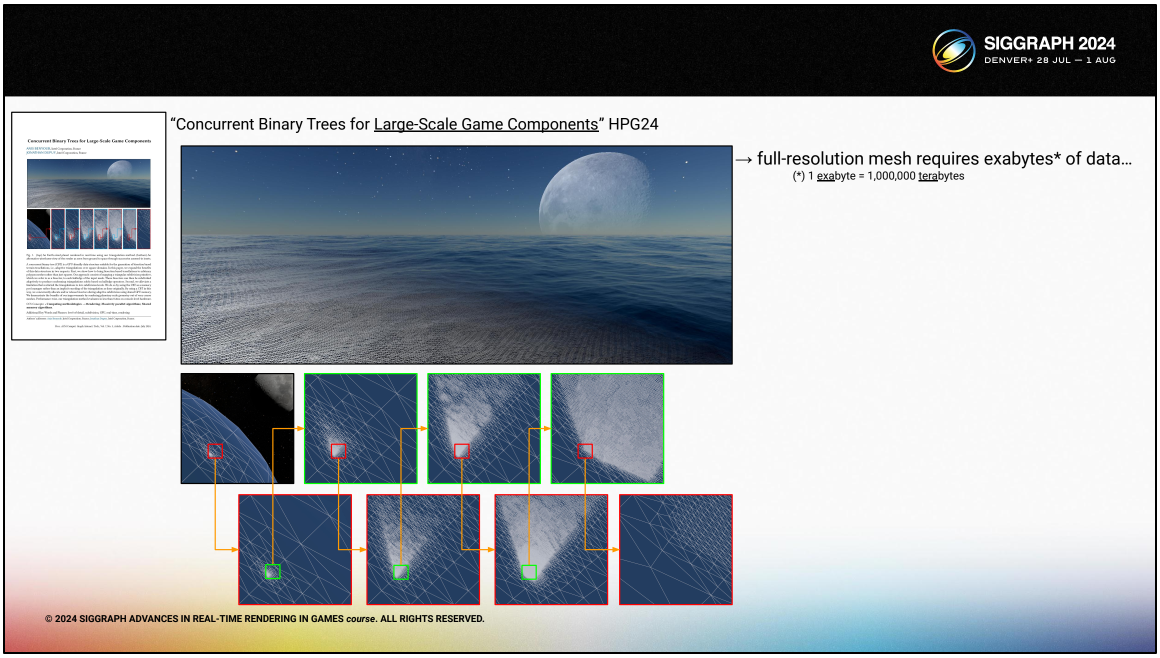
"Concurrent Binary Trees for Large-Scale Game Components" HPG24




Concurrent Binary Trees for Large-Scale Game Components

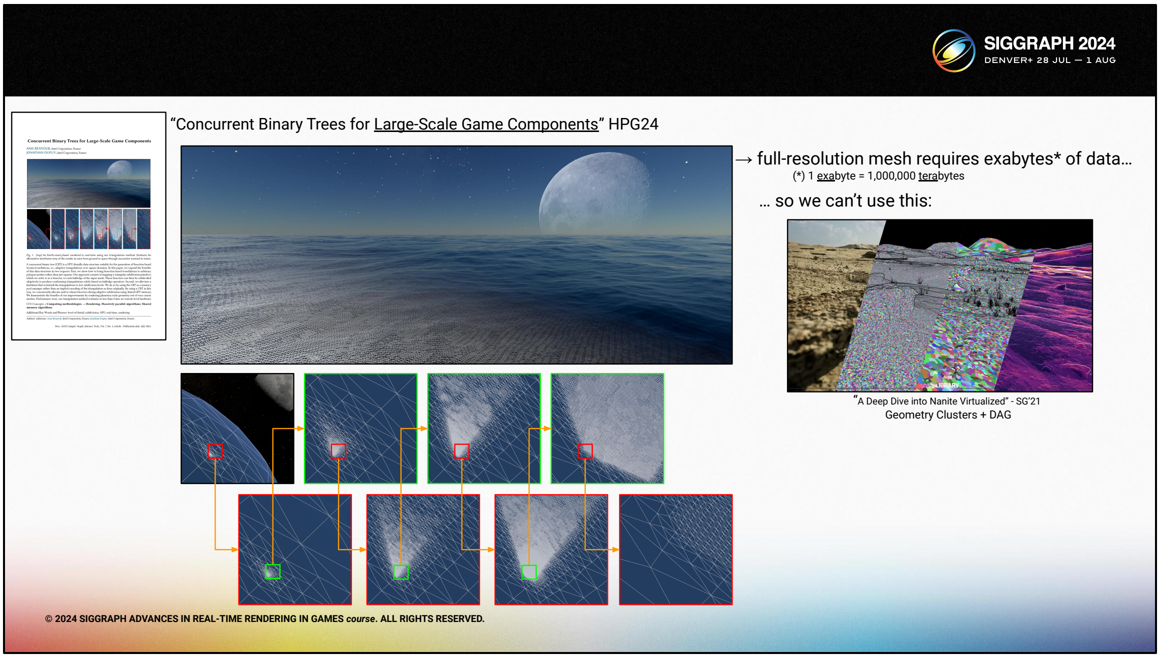
“Concurrent Binary Trees for Large-Scale Game Components” HPG24



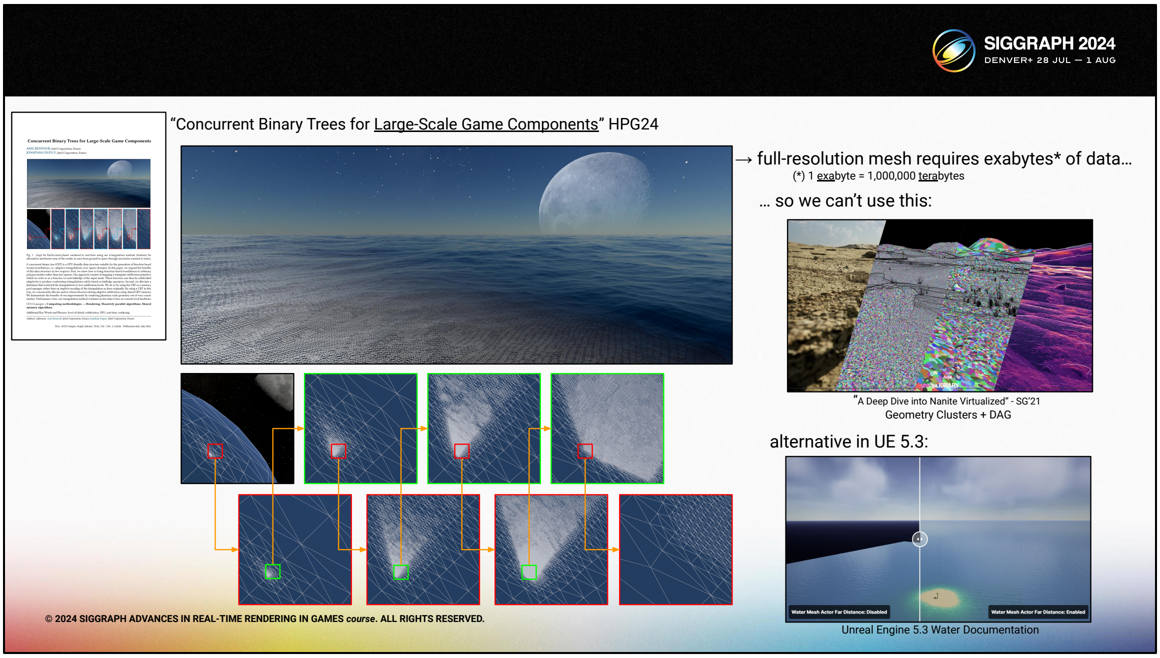


In terms of numbers, it turns out the full resolution mesh of our Earth model, i.e., without any form of level-of-detail as we do here, would require exabytes of data.

An exabyte is a million terabytes, so it would not even fit in a large SSD.



This prevents the use of LOD systems like Nanite, which requires the full resolution mesh as input.



Note that in UE5.3, you could still handle very large meshes using their very own large-scale game component, which consists in coupling two representations: one for close scale, and the other for far away scales.

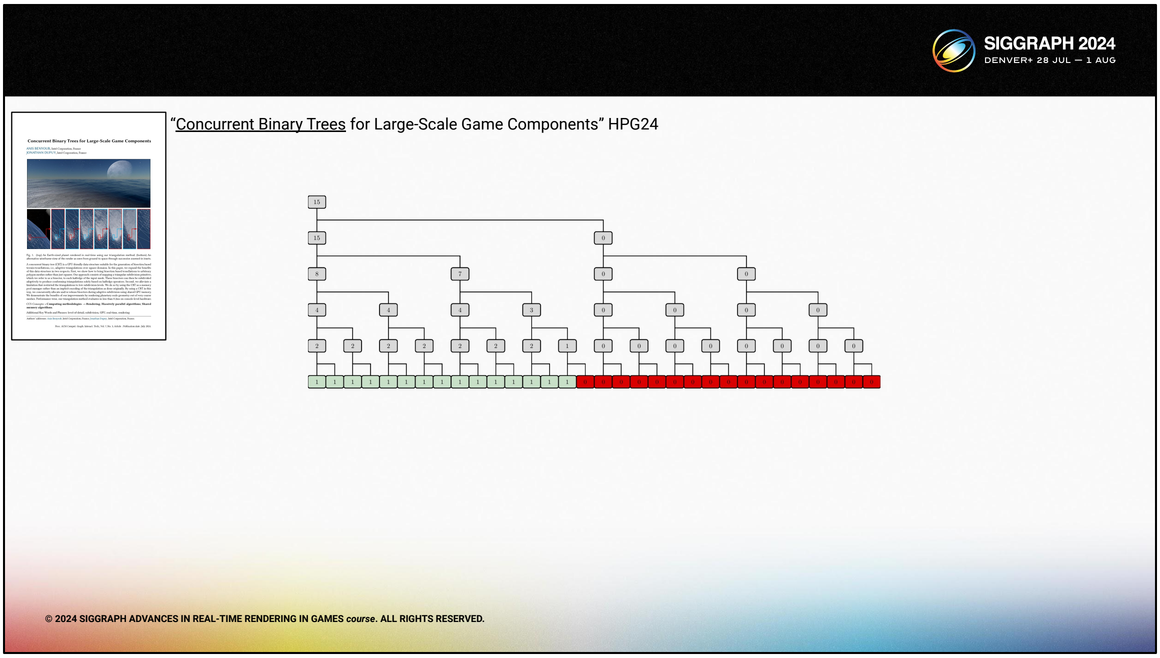
Unfortunately, hybrid representation are hard to use especially with free-flight cameras because it becomes really tricky to set the location of the transition between both representations.

A nice advantage of our method is that it relies on a single representation so you don't need to worry about these issues.

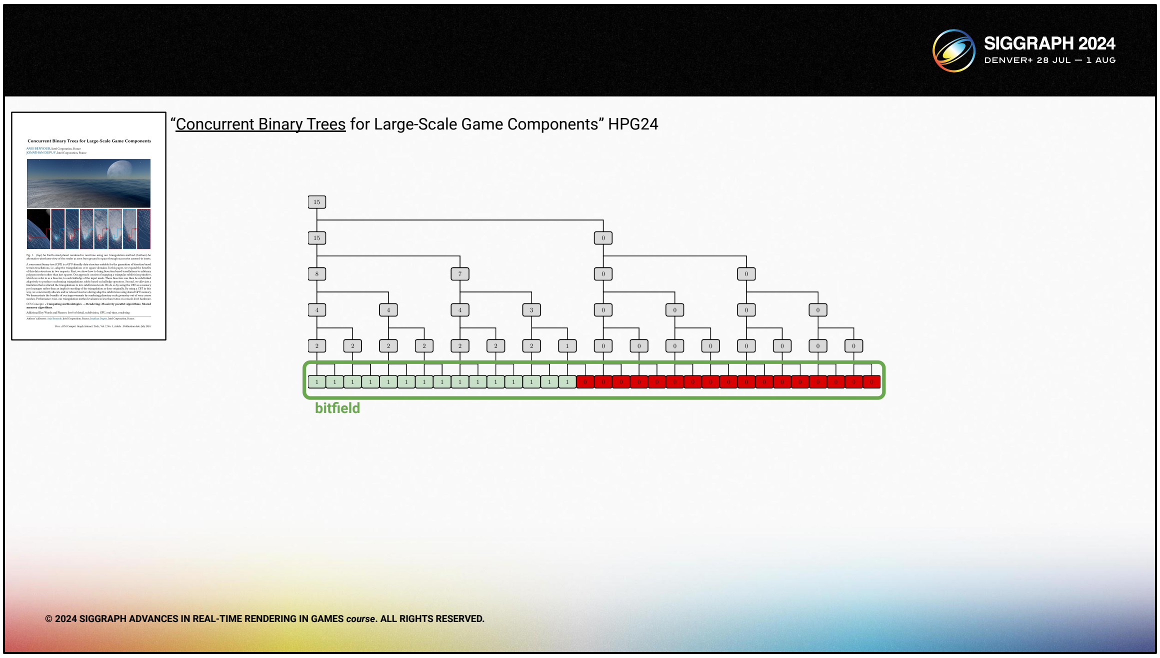
SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

"Concurrent Binary Trees for Large-Scale Game Components" HPG24

Right, now I am going to quickly explain what a concurrent binary tree (CBT) is and how our method works with them, then Anis will dive into the details.

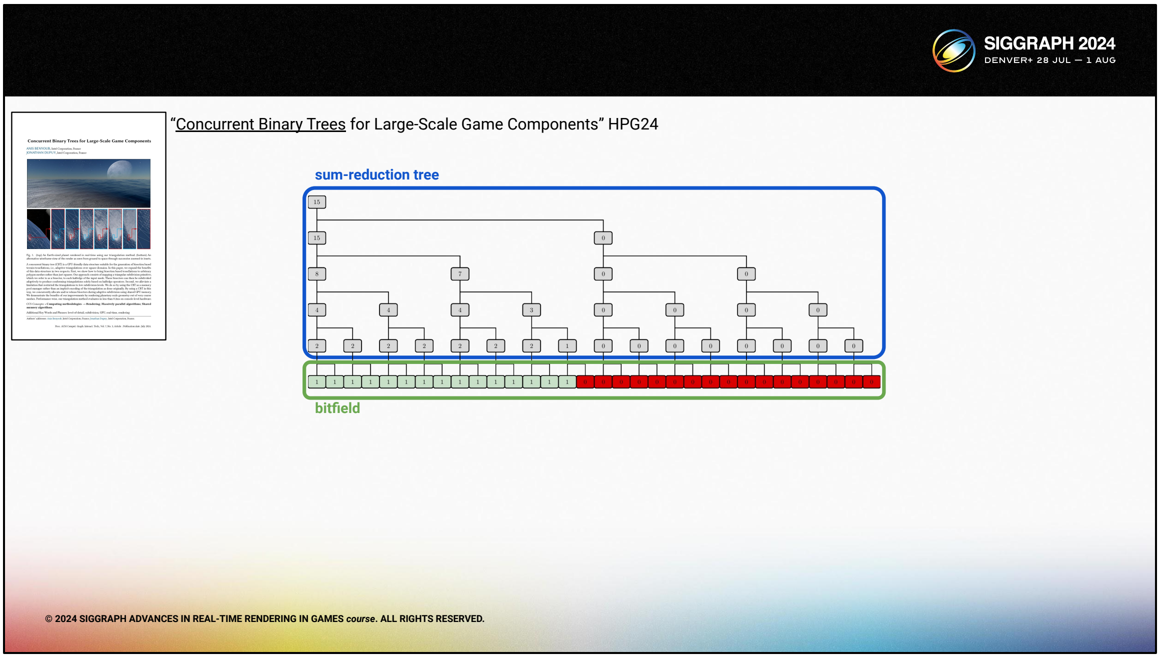


A concurrent binary tree of CBT is a full binary tree (so each node has exactly two children except for the leaves) with two main parts.



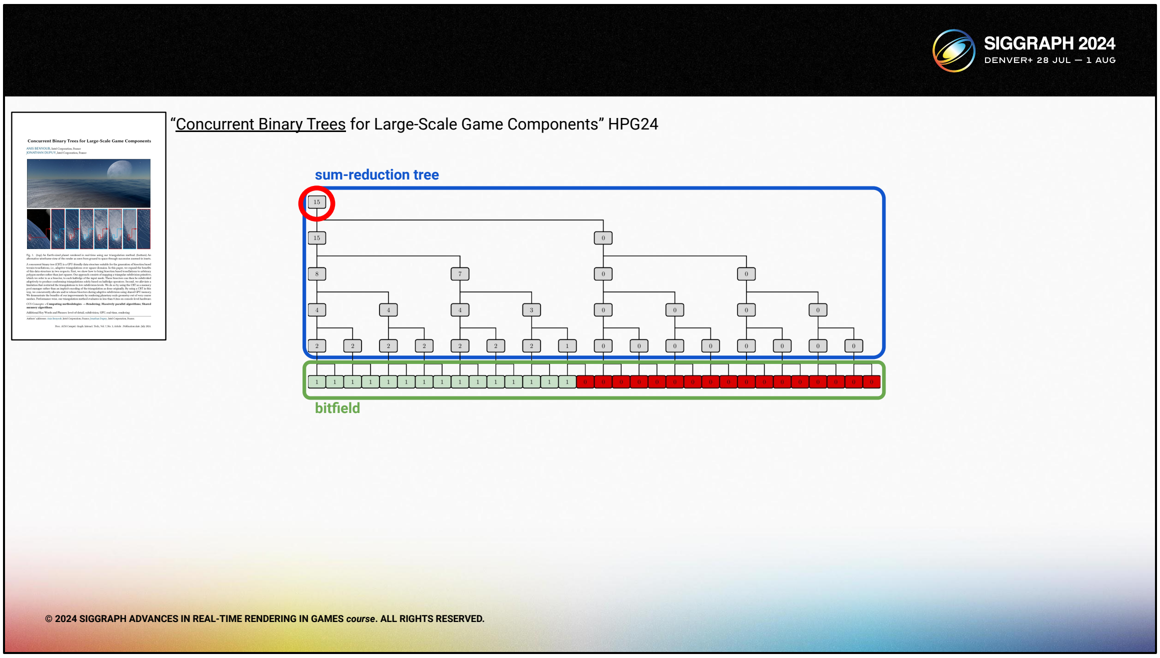
The first part is a bitfield located at the bottom of the tree.

So the leaf nodes only store binary values.



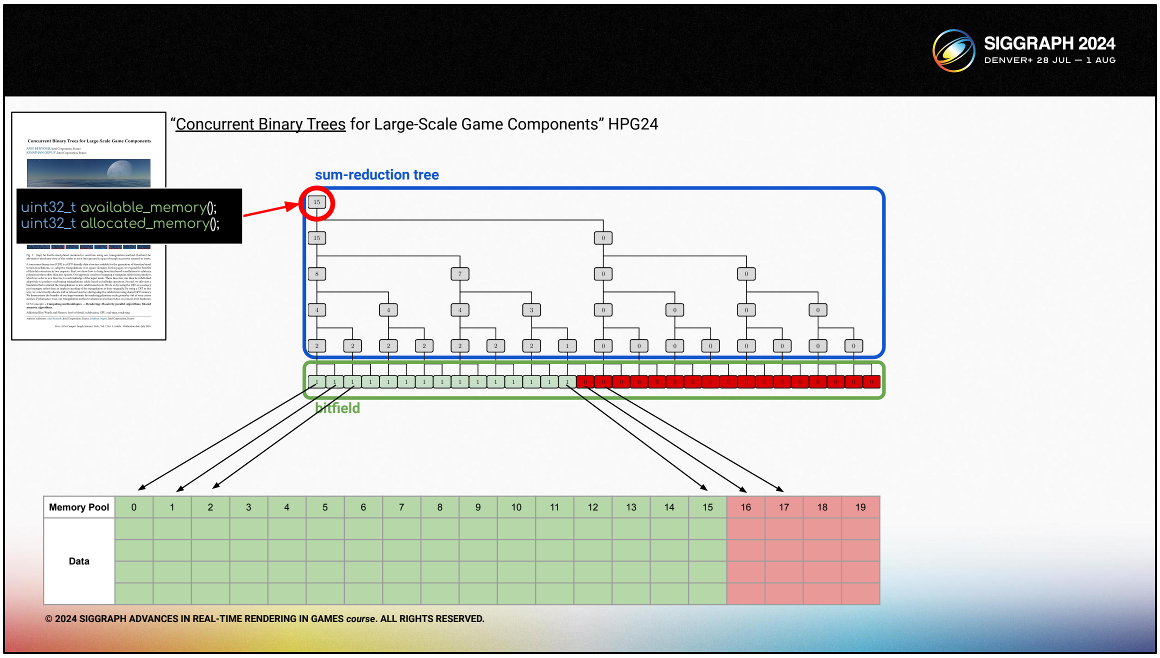
The second part is a sum-reduction tree of this bitfield.

So all remaining nodes store the number of green bits, i.e, bits set to one, in its corresponding subtree.



This means that the root node gives the total number of green bits.

In addition, the sum-reduction tree makes it possible to iterate over the green bits, even if the green bits aren't located sequentially in the bitfield.

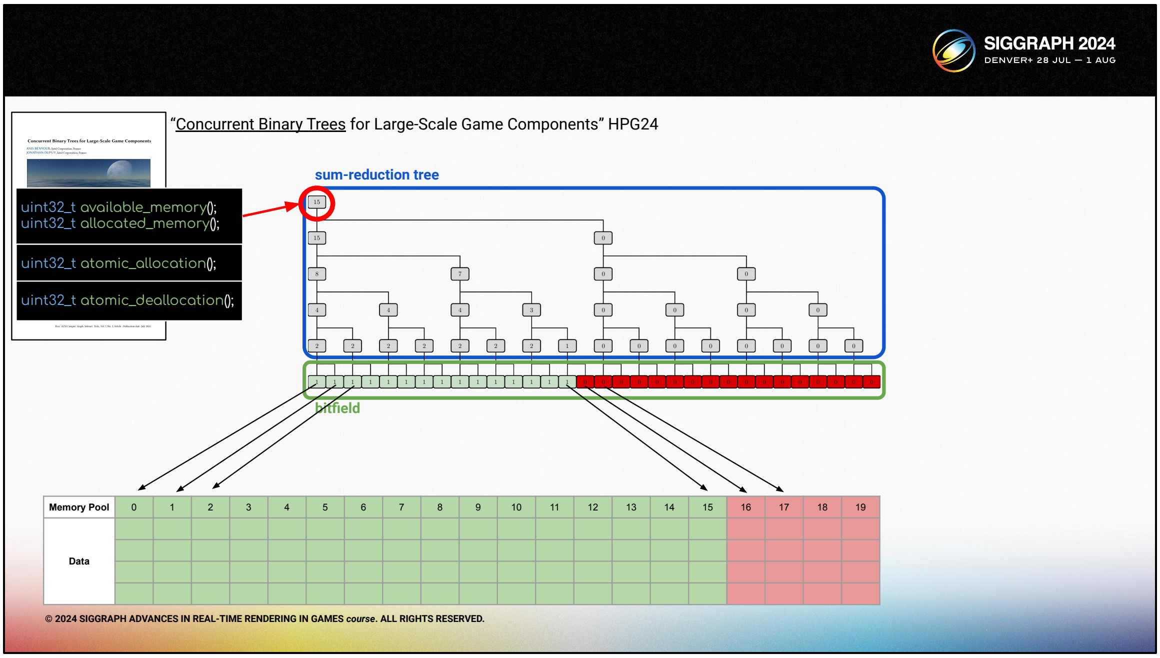


These properties makes it possible to use a CBT as memory pool manager that tracks allocated and available memory.

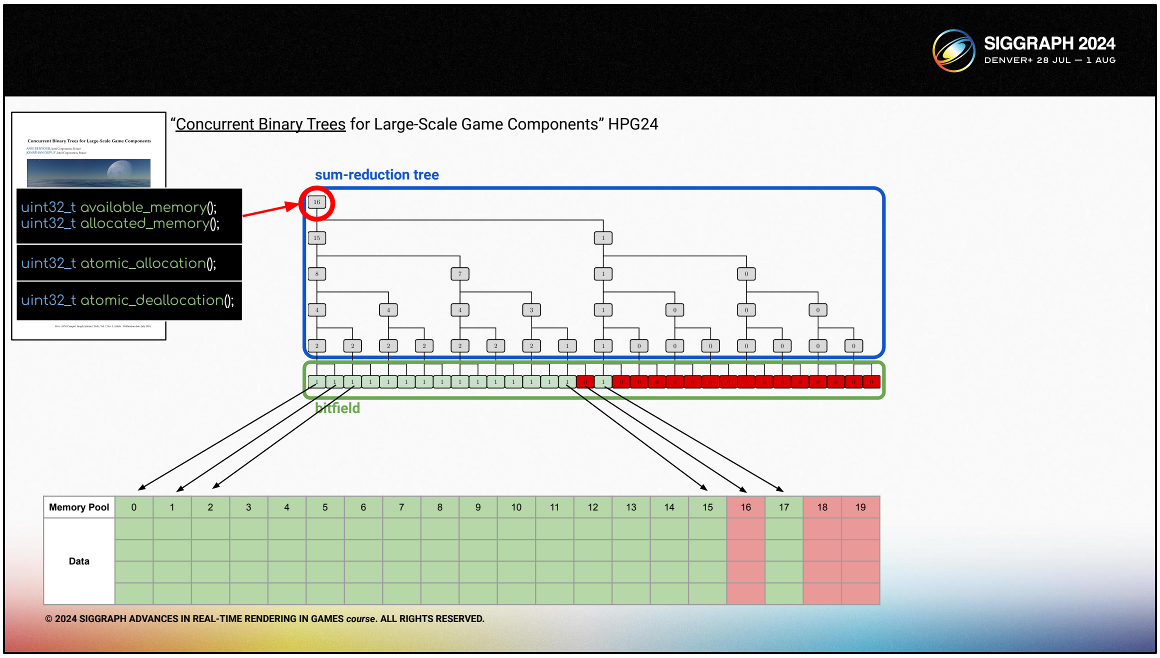
A memory pool is simply an array of whatever data you want to store and we set its capacity to that of the bitfield.

We then track allocated entries using green bits and available memory with red ones.

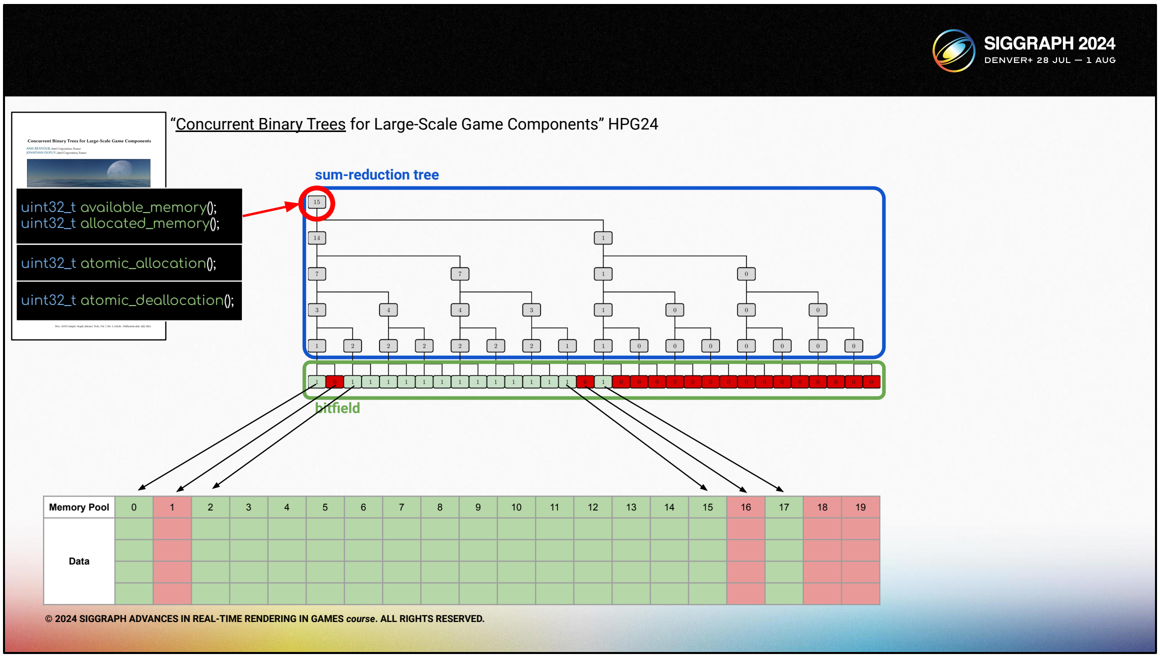
This way the root of the CBT gives us how much memory is available / allocated.



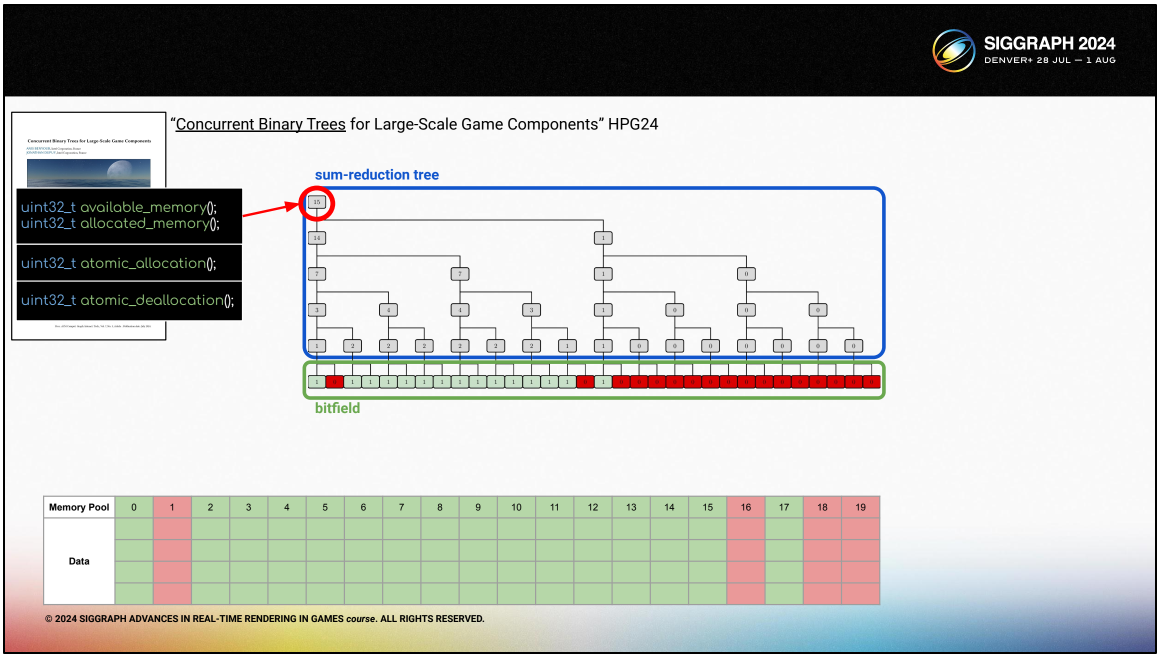
We can then implement a simple allocation and de-allocation operator as follows.



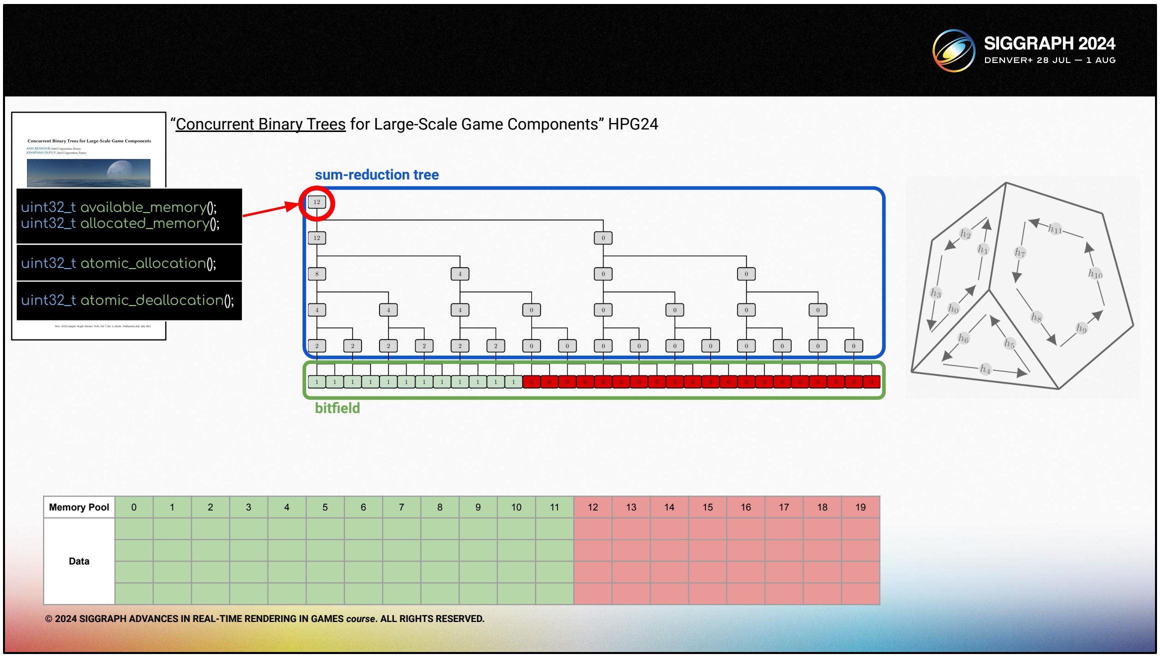
For allocation we set a bit to one and update the sum-reduction tree.



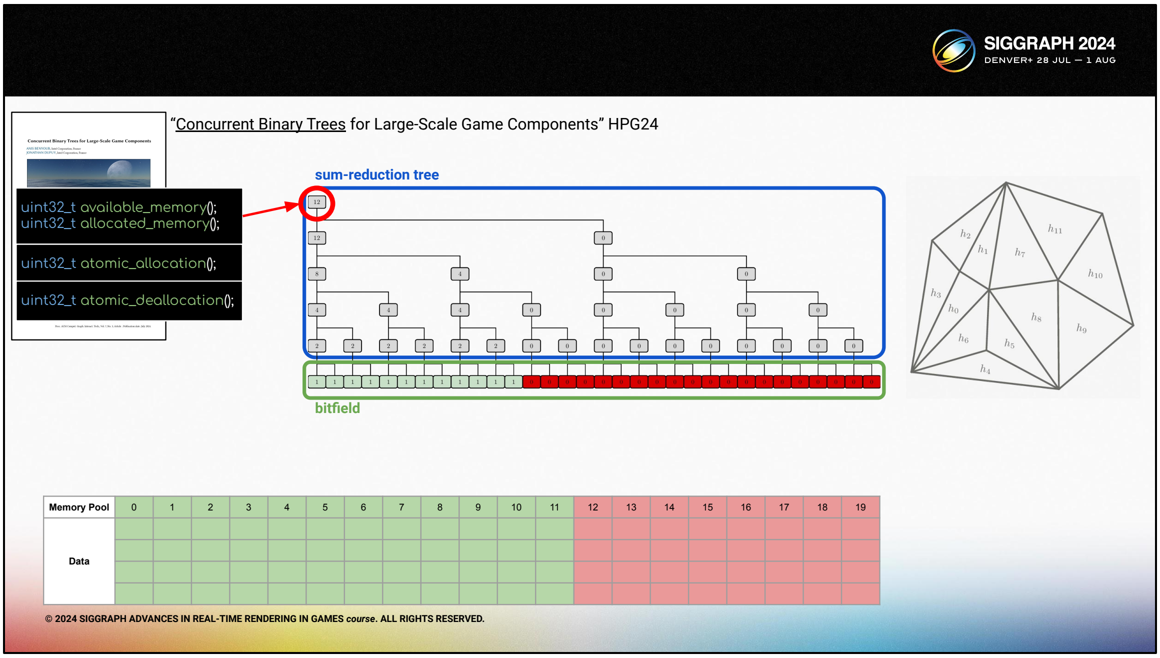
For de-allocation we set a bit to zero and again update the sum-reduction tree.



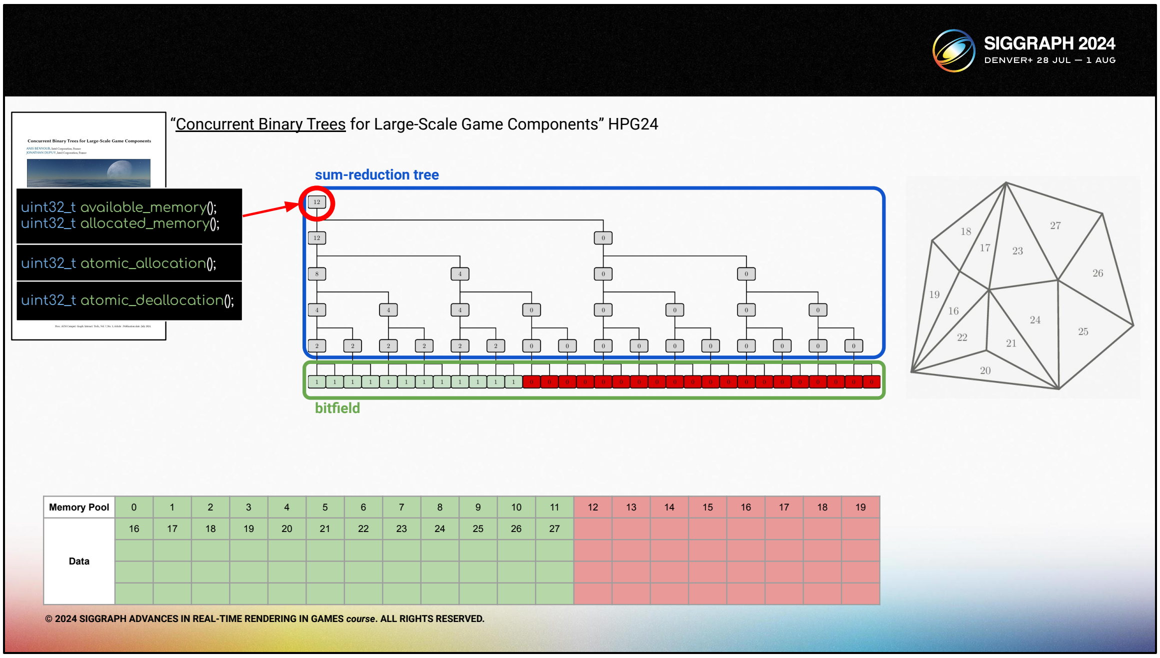
Now here is what we do for our triangulations.



Our method takes a halfedge mesh as input…

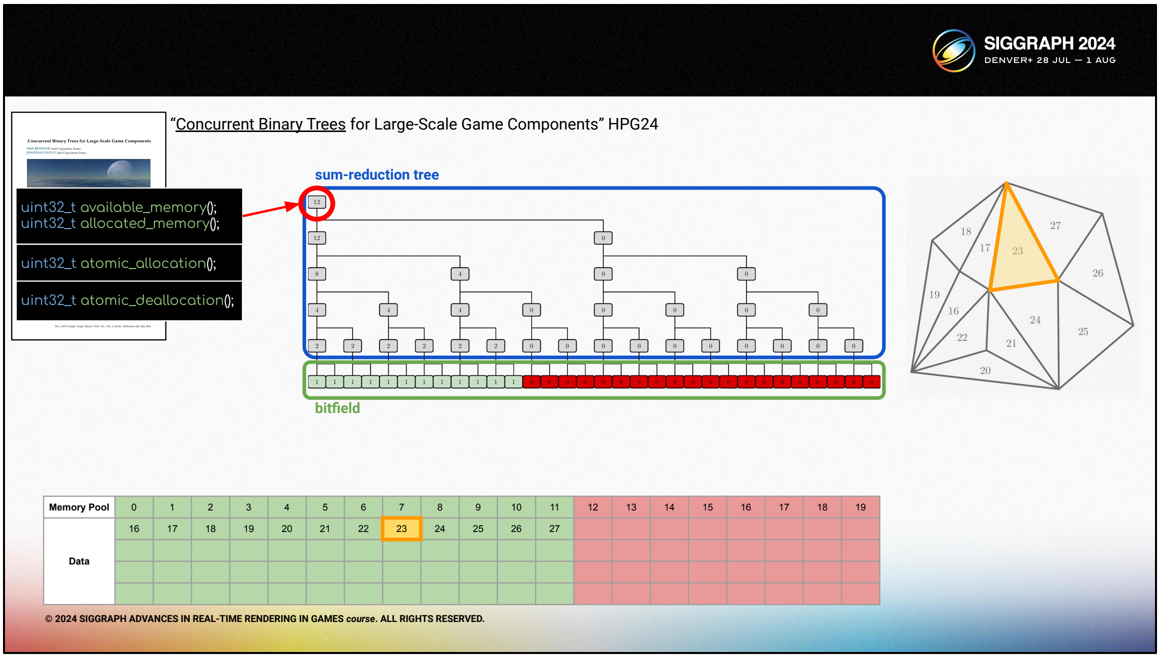


… and produces a triangle for each halfedge (feel free to go back and forth between this slide and the previous one).



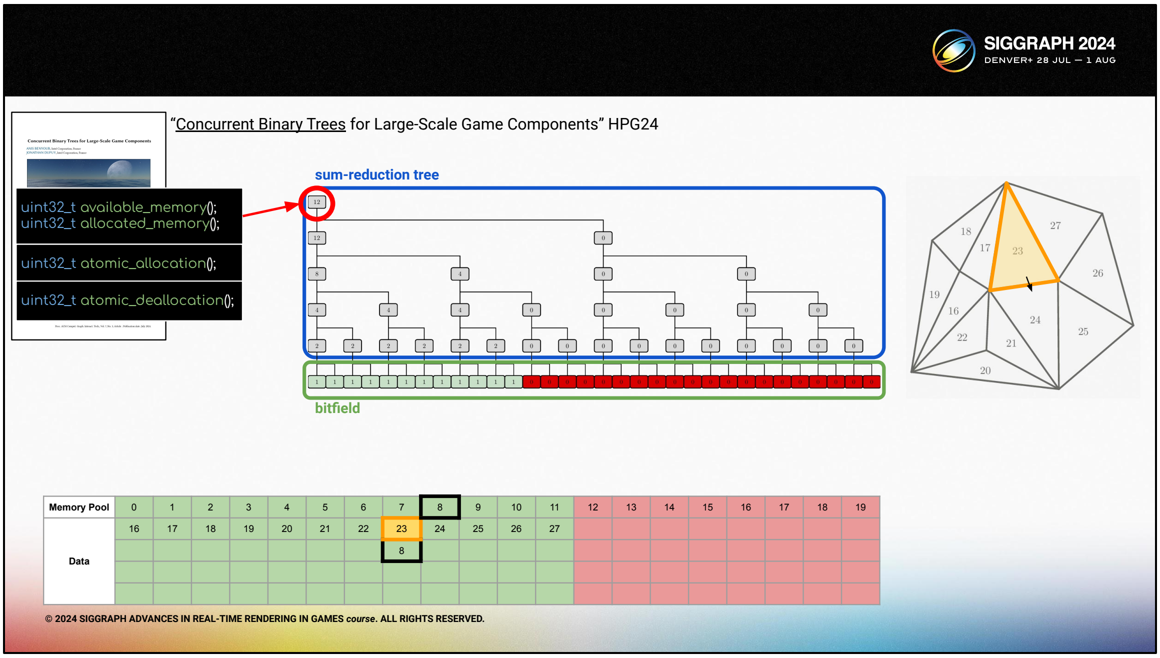
We then compress the 3 vertices of each triangle into a single integer value that we call a heapID and store it in a dedicated entry of the memory pool.

In this example we have 12 triangles so we require 12 slots in the memory pool.



As an example, the triangle 23 is stored in slot 7.

In addition to the heapID, we also store neighborhood information with pointers.

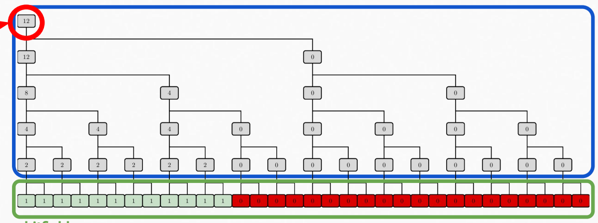


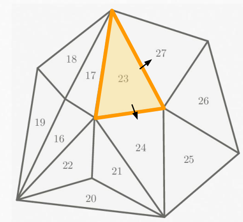
Continuing with triangle 23, the neighbors are triangle 24, which is located at slot 8…

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

“Concurrent Binary Trees for Large-Scale Game Components” HPG24

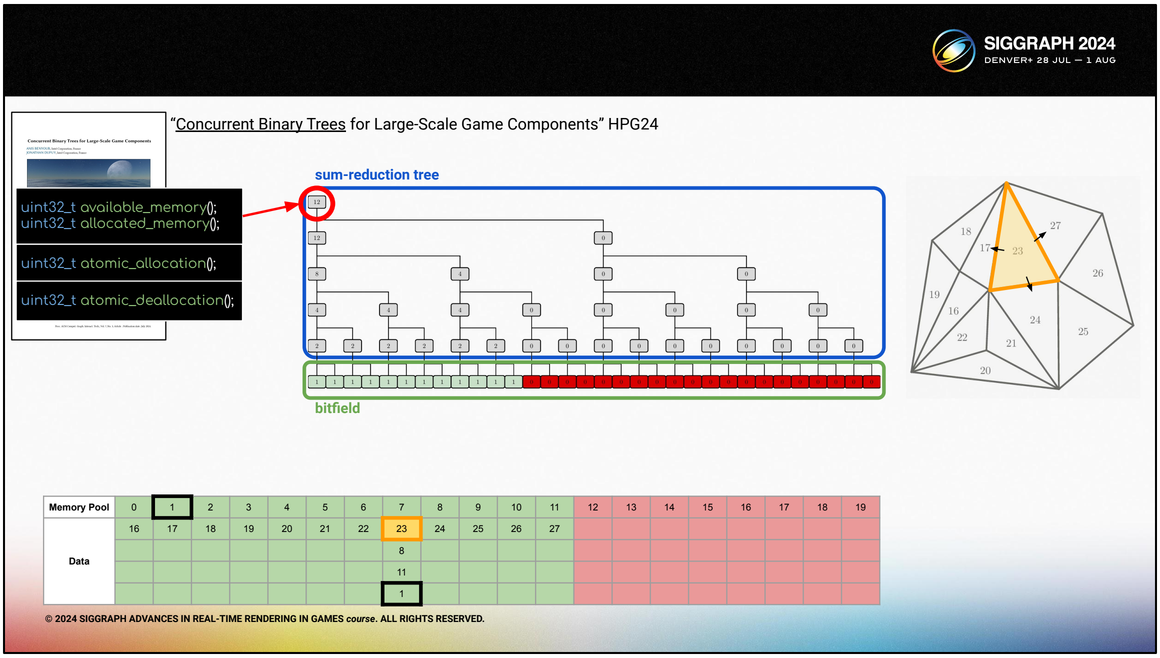
```
uint32_t available_memory();
uint32_t allocated_memory();

uint32_t atomic_allocation();

uint32_t atomic_deallocation();
```





Memory Pool	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Data	16	17	18	19	20	21	22	23	24	25	26	27								
								8												
								11												

… triangle 27, which is located at slot 11 …



… and triangle 17, which is located at slot 1.

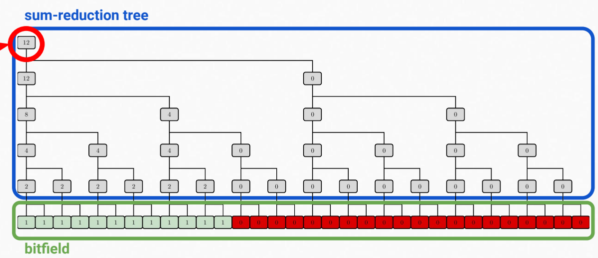
SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

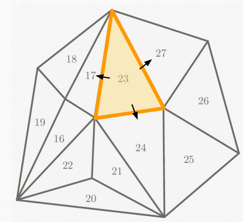
"Concurrent Binary Trees for Large-Scale Game Components" HPG24

```
uint32_t available_memory();
uint32_t allocated_memory();
```

```
uint32_t atomic_allocation();
```

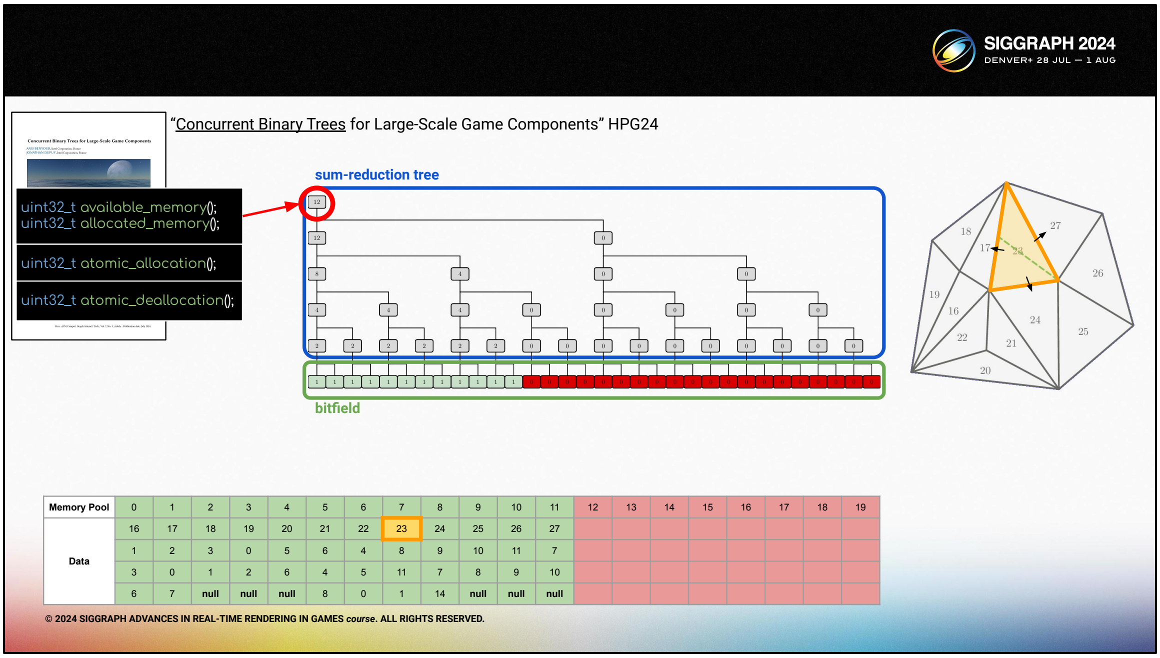
```
uint32_t atomic_deallocation();
```





Memory Pool	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Data	16	17	18	19	20	21	22	23	24	25	26	27								
	1	2	3	0	5	6	4	8	9	10	11	7								
	3	0	1	2	6	4	5	11	7	8	9	10								
	6	7	**null**	**null**	**null**	8	0	1	14	**null**	**null**	**null**								

And naturally we do this for each triangle.

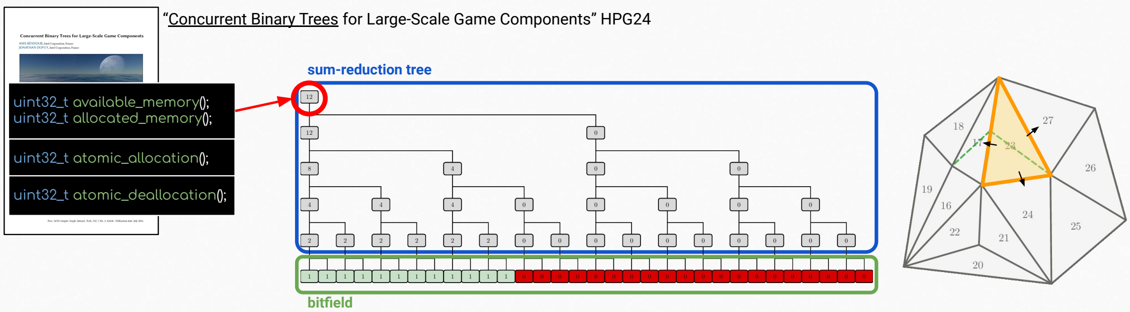


Memory Pool	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Data	16	17	18	19	20	21	22	23	24	25	26	27								
	1	2	3	0	5	6	4	8	9	10	11	7								
	3	0	1	2	6	4	5	11	7	8	9	10								
	6	7	null	null	null	8	0	1	14	null	null	null								

The reason we store this information is because we implement a bisection scheme that can split triangles into two new ones.

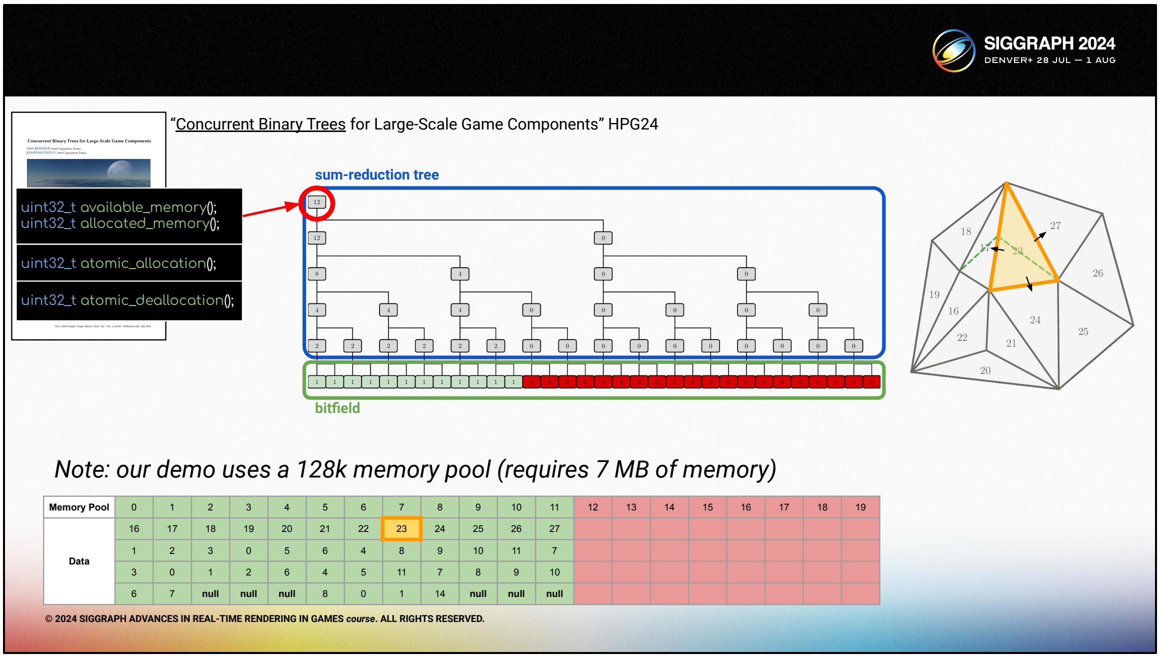
Naively bisecting a triangle would produce a T-junction, which would result in cracks in the final surface.

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG



Memory Pool	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Data	16	17	18	19	20	21	22	23	24	25	26	27								
	1	2	3	0	5	6	4	8	9	10	11	7								
	3	0	1	2	6	4	5	11	7	8	9	10								
	6	7	null	null	null	8	0	1	14	null	null	null								

But thanks to the neighborhood information, we can propagate bisections across multiple triangles to guarantee crack-free surfaces.



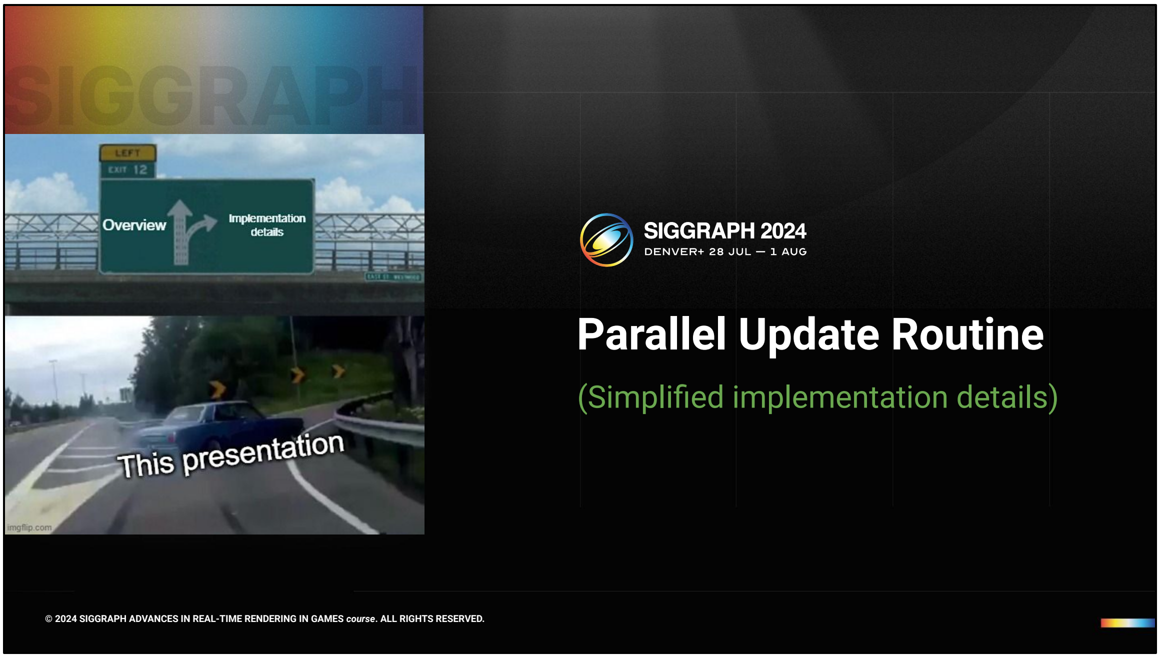
Last thing for me: the memory pool we use for our demo is 128k wide, which requires 7 MB of memory in total.

That concludes my overview of the paper.

The key takeaway here is that CBTs provide a way to allocate, release, and iterate over all the elements of a memory pool.

And as Anis will show, all this can be done efficiently on the GPU.

Your turn Anis!



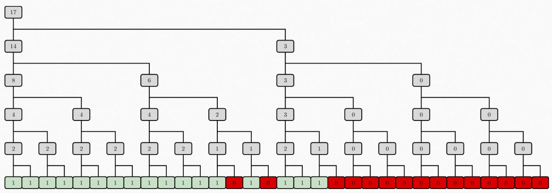
We’re going to dig into some of the implementation details that allowed us to reach reasonable performance numbers with this method.

That said, we’ll not go into code details due to the limited time we have.

The full source code of the demo is released and is available for you to explore and play with it!

Parallel Update Routine (CBT Implementation 128K)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG



```
// Store in shared memory, atomic friendly
groupshared uint32_t gs_cbt[                    ];
```


First, We'll precise the implementation of the CBT.

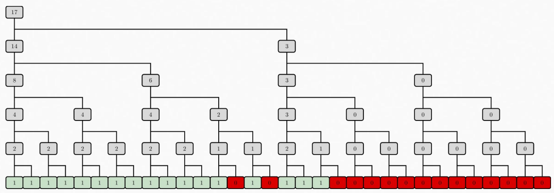
We'd like to store the CBT in the group shared memory to make the access more efficient when reading and writing.

As an example, we'll use the 128k elements CBT in this presentation, but as we mentioned before we can go higher (or lower) depending on the needs of the application

The CBT needs to be readable and writable from worker threads. We could use uint32_t to benefit from the atomic intrinsics, but there is an issue with that:

Parallel Update Routine (CBT Implementation 128K)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

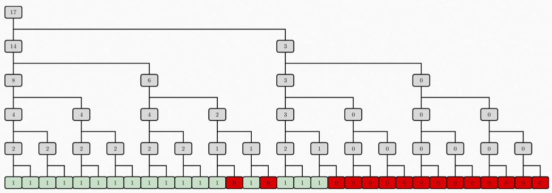


```
// Store in shared memory, atomic friendly
const uint32_t cbt_num_nodes = 2 * (128 * 1024); // 262144
groupshared uint32_t gs_cbt[cbt_num_nodes]; // 1 MB
```


The number of nodes of the CBT is twice the size of the bitfield, which would, if naively stored would be around one 1MB

Parallel Update Routine (CBT Implementation 128K)

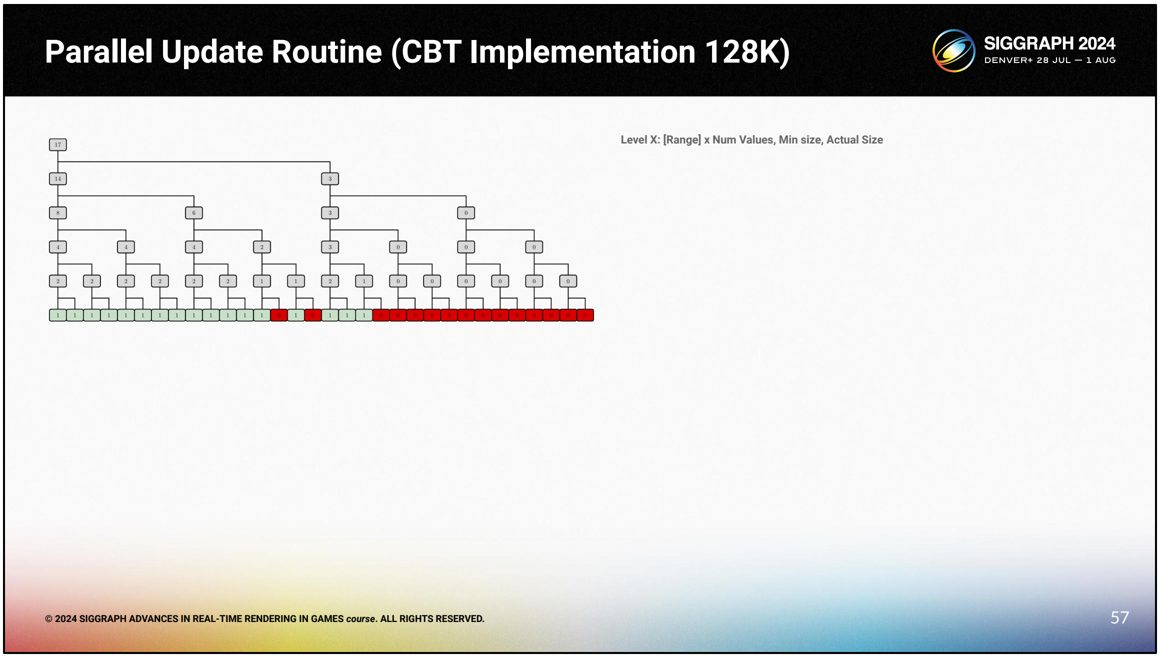
SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG



```
// Store in shared memory, atomic friendly
const uint32_t cbt_num_nodes = 2 * (128 * 1024); // 262144
groupshared uint32_t gs_cbt[cbt_num_nodes]; // 1 MB
```

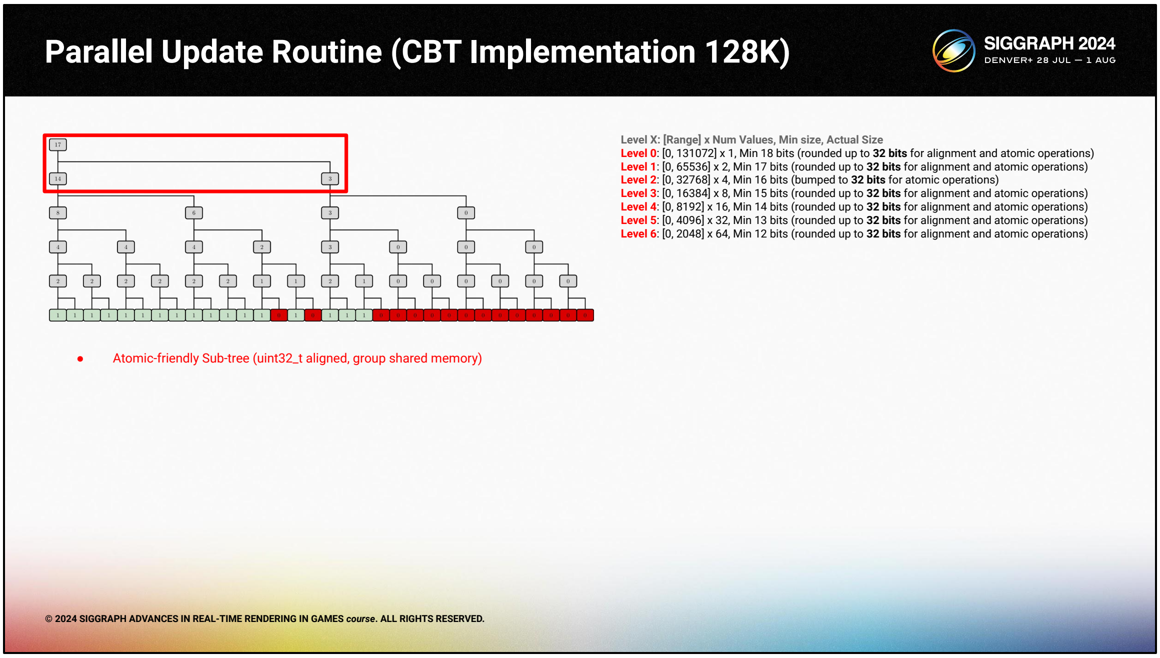
Limited to 32 KB on DX12 !!!

However, the group shared memory storage is limited to 32KB on dx12 so we need a better representation!



Each level of the tree is defined by 4 things:
- The number of nodes within the level
- The range of values each node can represent
- The minimal size of each node to represent that range
- The final size used to represent each node

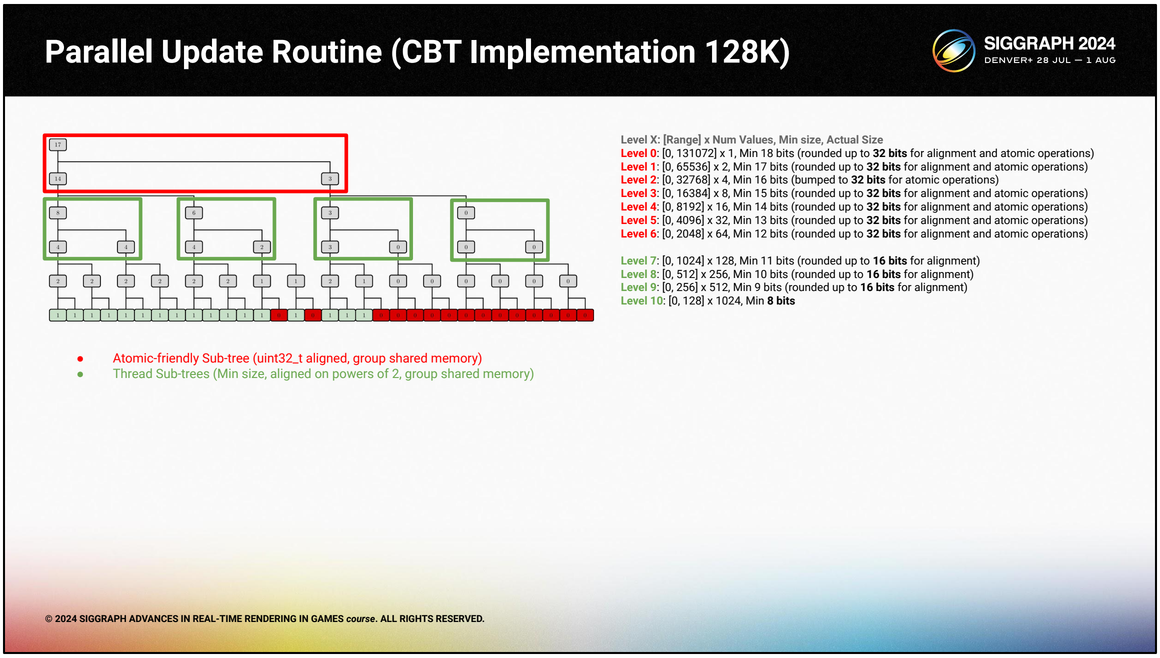
So what we'll do is decompose the tree into multiple parts with different constraints



The first part in red that is an atomic-friendly subtree

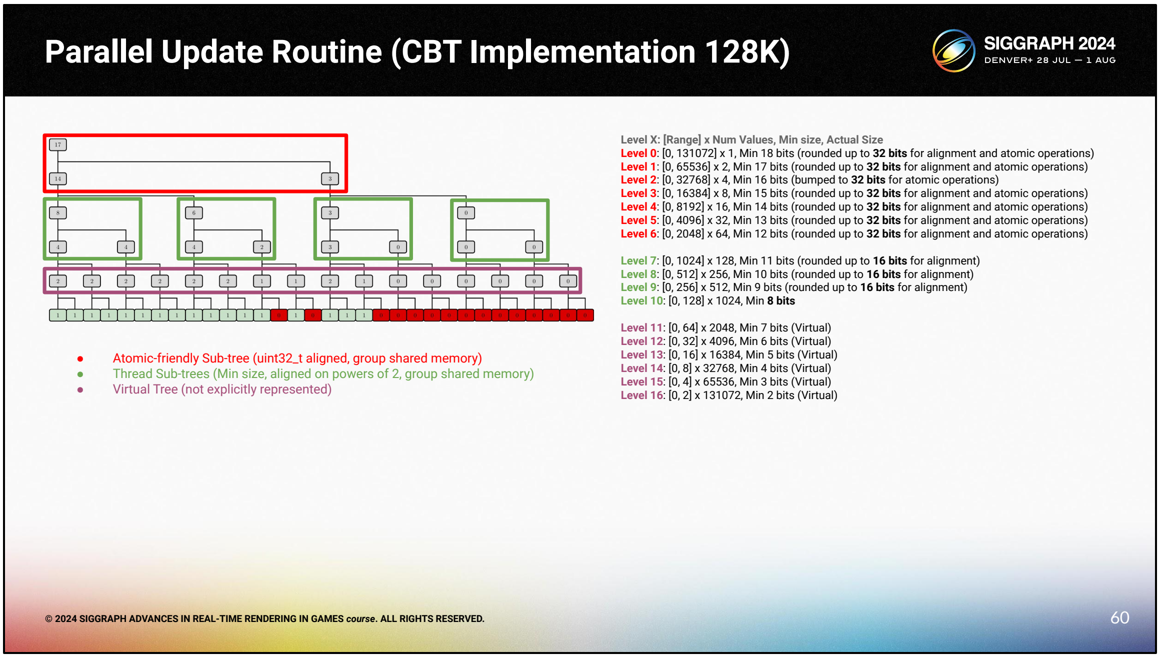
During modification, all threads can write safely to any node of the tree.

Each node is represented by uint32_t to benefit from the atomic intrinsics. This is stored in the group shared memory;



Iin green, we have several subtrees, each subtree will only be modified by one thread at a time.

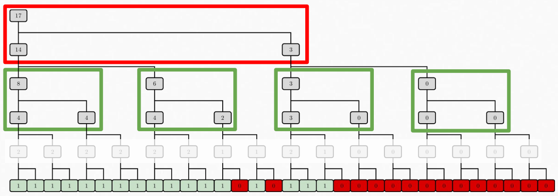
The size of each node is rounded to the closest power of two to represent the data underneath. This is also stored in the group shared memory;



Then there are what we call virtual levels. These are not represented explicitly,

Parallel Update Routine (CBT Implementation 128K)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG



- Atomic-friendly Sub-tree (uint32_t aligned, group shared memory)
- Thread Sub-trees (Min size, aligned on powers of 2, group shared memory)
- Virtual Tree (not explicitly represented)

Level X: [Range] x Num Values, Min size, Actual Size
Level 0: [0, 131072] x 1, Min 18 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 1: [0, 65536] x 2, Min 17 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 2: [0, 32768] x 4, Min 16 bits (bumped to **32 bits** for atomic operations)
Level 3: [0, 16384] x 8, Min 15 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 4: [0, 8192] x 16, Min 14 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 5: [0, 4096] x 32, Min 13 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 6: [0, 2048] x 64, Min 12 bits (rounded up to **32 bits** for alignment and atomic operations)

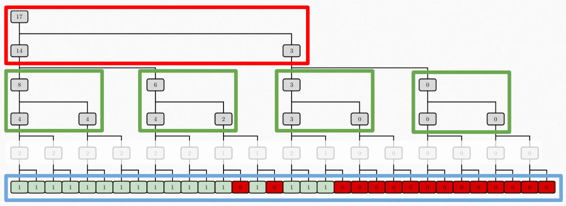
Level 7: [0, 1024] x 128, Min 11 bits (rounded up to **16 bits** for alignment)
Level 8: [0, 512] x 256, Min 10 bits (rounded up to **16 bits** for alignment)
Level 9: [0, 256] x 512, Min 9 bits (rounded up to **16 bits** for alignment)
Level 10: [0, 128] x 1024, Min **8 bits**

Level 11: [0, 64] x 2048, Min 7 bits (Virtual)
Level 12: [0, 32] x 4096, Min 6 bits (Virtual)
Level 13: [0, 16] x 16384, Min 5 bits (Virtual)
Level 14: [0, 8] x 32768, Min 4 bits (Virtual)
Level 15: [0, 4] x 65536, Min 3 bits (Virtual)
Level 16: [0, 2] x 131072, Min 2 bits (Virtual)

but can be deduced from the last level

Parallel Update Routine (CBT Implementation 128K)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG



- Atomic-friendly Sub-tree (uint32_t aligned, group shared memory)
- Thread Sub-trees (Min size, aligned on powers of 2, group shared memory)
- Virtual Tree (not explicitly represented)
- Raw Bitfield Buffer (uint64_t, StructuredBuffer)

Level X: [Range] x Num Values, Min size, Actual Size
Level 0: [0, 131072] x 1, Min 18 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 1: [0, 65536] x 2, Min 17 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 2: [0, 32768] x 4, Min 16 bits (bumped to **32 bits** for atomic operations)
Level 3: [0, 16384] x 8, Min 15 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 4: [0, 8192] x 16, Min 14 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 5: [0, 4096] x 32, Min 13 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 6: [0, 2048] x 64, Min 12 bits (rounded up to **32 bits** for alignment and atomic operations)

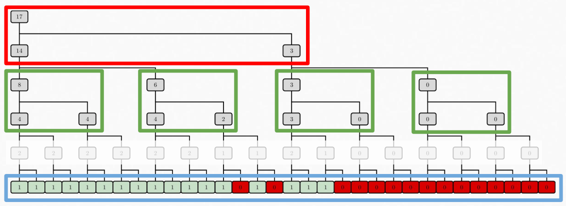
Level 7: [0, 1024] x 128, Min 11 bits (rounded up to **16 bits** for alignment)
Level 8: [0, 512] x 256, Min 10 bits (rounded up to **16 bits** for alignment)
Level 9: [0, 256] x 512, Min 9 bits (rounded up to **16 bits** for alignment)
Level 10: [0, 128] x 1024, Min **8 bits**

Level 11: [0, 64] x 2048, Min 7 bits (Virtual)
Level 12: [0, 32] x 4096, Min 6 bits (Virtual)
Level 13: [0, 16] x 16384, Min 5 bits (Virtual)
Level 14: [0, 8] x 32768, Min 4 bits (Virtual)
Level 15: [0, 4] x 65536, Min 3 bits (Virtual)
Level 16: [0, 2] x 131072, Min 2 bits (Virtual)

Level 17: Raw 64 bits representation

Which is the rawbitfield. This bitfield it is represented using uint64_t to benefit from some intrinsic functions and is stored in a structured buffer outside of the shared memory.

Parallel Update Routine (CBT Implementation 128K)



- Atomic-friendly Sub-tree (uint32_t aligned, group shared memory)
- Thread Sub-trees (Min size, aligned on powers of 2, group shared memory)
- Virtual Tree (not explicitly represented)
- Raw Bitfield Buffer (uint64_t, StructuredBuffer)

Level X: [Range] x Num Values, Min size, Actual Size
Level 0: [0, 131072] x 1, Min 18 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 1: [0, 65536] x 2, Min 17 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 2: [0, 32768] x 4, Min 16 bits (bumped to **32 bits** for atomic operations)
Level 3: [0, 16384] x 8, Min 15 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 4: [0, 8192] x 16, Min 14 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 5: [0, 4096] x 32, Min 13 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 6: [0, 2048] x 64, Min 12 bits (rounded up to **32 bits** for alignment and atomic operations)

Level 7: [0, 1024] x 128, Min 11 bits (rounded up to **16 bits** for alignment)
Level 8: [0, 512] x 256, Min 10 bits (rounded up to **16 bits** for alignment)
Level 9: [0, 256] x 512, Min 9 bits (rounded up to **16 bits** for alignment)
Level 10: [0, 128] x 1024, Min **8 bits**

Level 17: Raw 64 bits representation

And this the memory footprint of a 128k bit CBT. In the implementation, you'll find the layout for the other sizes of the CBT

Parallel Update Routine (CBT Implementation 128K)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

// Graphics Buffers that hold the CBT

Level 0: [0, 131072] x 1, Min 18 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 1: [0, 65536] x 2, Min 17 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 2: [0, 32768] x 4, Min 16 bits (bumped to **32 bits** for atomic operations)
Level 3: [0, 16384] x 8, Min 15 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 4: [0, 8192] x 16, Min 14 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 5: [0, 4096] x 32, Min 13 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 6: [0, 2048] x 64, Min 12 bits (rounded up to **32 bits** for alignment and atomic operations)

Level 7: [0, 1024] x 128, Min 11 bits (rounded up to **16 bits** for alignment)
Level 8: [0, 512] x 256, Min 10 bits (rounded up to **16 bits** for alignment)
Level 9: [0, 256] x 512, Min 9 bits (rounded up to **16 bits** for alignment)
Level 10: [0, 128] x 1024, Min **8 bits**

Level 17: Raw 64 bits representation

Now let’s look in practice what that maps to:

Parallel Update Routine (CBT Implementation 128K)


```
// Graphics Buffers that hold the CBT
RWStructuredBuffer<uint32_t> _TreeBufferRW: register(CBT_BUFFER0_BINDING_SLOT); // RED + GREEN
RWStructuredBuffer<uint64_t> _BitfieldBufferRW: register(CBT_BUFFER1_BINDING_SLOT); // BLUE
```

Level 0: [0, 131072] x 1, Min 18 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 1: [0, 65536] x 2, Min 17 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 2: [0, 32768] x 4, Min 16 bits (bumped to **32 bits** for atomic operations)
Level 3: [0, 16384] x 8, Min 15 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 4: [0, 8192] x 16, Min 14 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 5: [0, 4096] x 32, Min 13 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 6: [0, 2048] x 64, Min 12 bits (rounded up to **32 bits** for alignment and atomic operations)

Level 7: [0, 1024] x 128, Min 11 bits (rounded up to **16 bits** for alignment)
Level 8: [0, 512] x 256, Min 10 bits (rounded up to **16 bits** for alignment)
Level 9: [0, 256] x 512, Min 9 bits (rounded up to **16 bits** for alignment)
Level 10: [0, 128] x 1024, Min **8 bits**

Level 17: Raw 64 bits representation

We have two structured buffers:
- The first one stores uint32_t and contains the red and green parts of the tree
- The second stores uint64_t and contains the bitfield (in blue)

Parallel Update Routine (CBT Implementation 128K)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

```
// Graphics Buffers that hold the CBT
RWStructuredBuffer<uint32_t> _TreeBufferRW: register(CBT_BUFFER0_BINDING_SLOT); // RED + GREEN
RWStructuredBuffer<uint64_t> _BitfieldBufferRW: register(CBT_BUFFER1_BINDING_SLOT); // BLUE
RWStructuredBuffer<uint32_t> _AllocationBufferRW: register(CBT_BUFFER2_BINDING_SLOT);
```

Level 0: [0, 131072] x 1, Min 18 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 1: [0, 65536] x 2, Min 17 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 2: [0, 32768] x 4, Min 16 bits (bumped to **32 bits** for atomic operations)
Level 3: [0, 16384] x 8, Min 15 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 4: [0, 8192] x 16, Min 14 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 5: [0, 4096] x 32, Min 13 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 6: [0, 2048] x 64, Min 12 bits (rounded up to **32 bits** for alignment and atomic operations)

Level 7: [0, 1024] x 128, Min 11 bits (rounded up to **16 bits** for alignment)
Level 8: [0, 512] x 256, Min 10 bits (rounded up to **16 bits** for alignment)
Level 9: [0, 256] x 512, Min 9 bits (rounded up to **16 bits** for alignment)
Level 10: [0, 128] x 1024, Min **8 bits**

Level 17: Raw 64 bits representation

In addition to that we need an allocation buffer which is a uint32_t buffer, we'll see how it is used in a second

Parallel Update Routine (CBT Implementation 128K)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

```
// Graphics Buffers that hold the CBT
RWStructuredBuffer<uint32_t> _TreeBufferRW: register(CBT_BUFFER0_BINDING_SLOT); // RED + GREEN
RWStructuredBuffer<uint64_t> _BitfieldBufferRW: register(CBT_BUFFER1_BINDING_SLOT); // BLUE
RWStructuredBuffer<uint32_t> _AllocationBufferRW: register(CBT_BUFFER2_BINDING_SLOT);

// _TreeBufferRW in the shared memory
const uint32_t tree_num_slots = (1 * 32 + 2 * 32 + ... + 1024 * 8) / 32;
```

Level 0: [0, 131072] x 1, Min 18 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 1: [0, 65536] x 2, Min 17 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 2: [0, 32768] x 4, Min 16 bits (bumped to **32 bits** for atomic operations)
Level 3: [0, 16384] x 8, Min 15 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 4: [0, 8192] x 16, Min 14 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 5: [0, 4096] x 32, Min 13 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 6: [0, 2048] x 64, Min 12 bits (rounded up to **32 bits** for alignment and atomic operations)

Level 7: [0, 1024] x 128, Min 11 bits (rounded up to **16 bits** for alignment)
Level 8: [0, 512] x 256, Min 10 bits (rounded up to **16 bits** for alignment)
Level 9: [0, 256] x 512, Min 9 bits (rounded up to **16 bits** for alignment)
Level 10: [0, 128] x 1024, Min **8 bits**

Level 17: Raw 64 bits representation

Using the memory footprint table on the right, we can deduce the size of the tree

Parallel Update Routine (CBT Implementation 128K)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

```
// Graphics Buffers that hold the CBT
RWStructuredBuffer<uint32_t> _TreeBufferRW: register(CBT_BUFFER0_BINDING_SLOT); // RED + GREEN
RWStructuredBuffer<uint64_t> _BitfieldBufferRW: register(CBT_BUFFER1_BINDING_SLOT); // BLUE
RWStructuredBuffer<uint32_t> _AllocationBufferRW: register(CBT_BUFFER2_BINDING_SLOT);

// _TreeBufferRW in the shared memory
const uint32_t tree_num_slots = (1 * 32 + 2 * 32 + ... + 1024 * 8) / 32;
groupshared uint32_t gs_cbtTree[tree_num_slots]; // ~3KB
```

Level 0: [0, 131072] x 1, Min 18 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 1: [0, 65536] x 2, Min 17 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 2: [0, 32768] x 4, Min 16 bits (bumped to **32 bits** for atomic operations)
Level 3: [0, 16384] x 8, Min 15 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 4: [0, 8192] x 16, Min 14 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 5: [0, 4096] x 32, Min 13 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 6: [0, 2048] x 64, Min 12 bits (rounded up to **32 bits** for alignment and atomic operations)

Level 7: [0, 1024] x 128, Min 11 bits (rounded up to **16 bits** for alignment)
Level 8: [0, 512] x 256, Min 10 bits (rounded up to **16 bits** for alignment)
Level 9: [0, 256] x 512, Min 9 bits (rounded up to **16 bits** for alignment)
Level 10: [0, 128] x 1024, Min **8 bits**

Level 17: Raw 64 bits representation

And that defines the group shared memory space our CBT occupies. In this case the CBT is about 3 KB, which we can consider reasonable.

Parallel Update Routine (CBT Implementation 128K)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

```
// Graphics Buffers that hold the CBT
RWStructuredBuffer<uint32_t> _TreeBufferRW: register(CBT_BUFFER0_BINDING_SLOT); // RED + GREEN
RWStructuredBuffer<uint64_t> _BitfieldBufferRW: register(CBT_BUFFER1_BINDING_SLOT); // BLUE
RWStructuredBuffer<uint32_t> _AllocationBufferRW: register(CBT_BUFFER2_BINDING_SLOT);

// _TreeBufferRW in the shared memory
const uint32_t tree_num_slots = (1 * 32 + 2 * 32 + ... + 1024 * 8) / 32;
groupshared uint32_t gs_cbtTree[tree_num_slots]; // ~3KB
```

Level 0: [0, 131072] x 1, Min 18 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 1: [0, 65536] x 2, Min 17 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 2: [0, 32768] x 4, Min 16 bits (bumped to **32 bits** for atomic operations)
Level 3: [0, 16384] x 8, Min 15 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 4: [0, 8192] x 16, Min 14 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 5: [0, 4096] x 32, Min 13 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 6: [0, 2048] x 64, Min 12 bits (rounded up to **32 bits** for alignment and atomic operations)

Level 7: [0, 1024] x 128, Min 11 bits (rounded up to **16 bits** for alignment)
Level 8: [0, 512] x 256, Min 10 bits (rounded up to **16 bits** for alignment)
Level 9: [0, 256] x 512, Min 9 bits (rounded up to **16 bits** for alignment)
Level 10: [0, 128] x 1024, Min **8 bits**

Level 17: Raw 64 bits representation

Book & Allocate & Cancel

The way the memory manager is used every frame is the following.

- First we're gonna book the worst case memory we need
- Then we're gonna allocate a bunch of bits, that will depend on the subdivision case we're processing,
- We'll finally return to the memory manager memory space what we didn't consume

Parallel Update Routine (CBT Implementation 128K)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

```
// Graphics Buffers that hold the CBT
RWStructuredBuffer<uint32_t> _TreeBufferRW: register(CBT_BUFFER0_BINDING_SLOT); // RED + GREEN
RWStructuredBuffer<uint64_t> _BitfieldBufferRW: register(CBT_BUFFER1_BINDING_SLOT); // BLUE
RWStructuredBuffer<uint32_t> _AllocationBufferRW: register(CBT_BUFFER2_BINDING_SLOT);

// _TreeBufferRW in the shared memory
const uint32_t tree_num_slots = (1 * 32 + 2 * 32 + ... + 1024 * 8) / 32;
groupshared uint32_t gs_cbtTree[tree_num_slots]; // ~3KB

// Memory booking
void cbt_new_frame();
void book_memory_space(uint32_t numSlots);
void cancel_memory_booking(uint32_t numSlots);
```

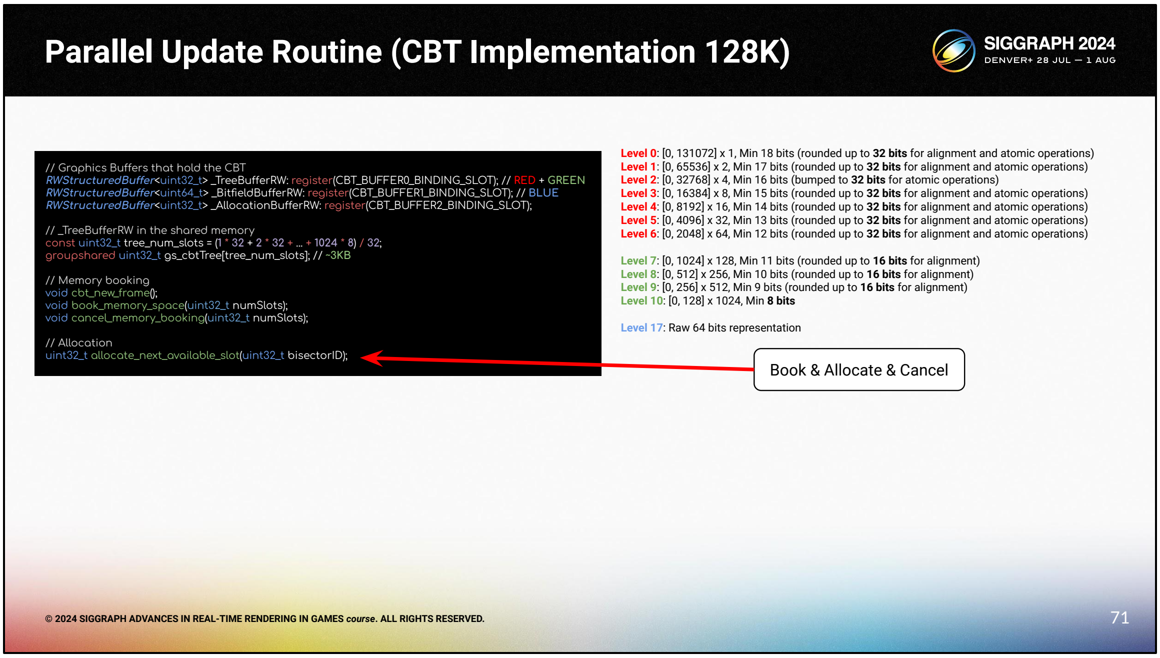
Level 0: [0, 131072] x 1, Min 18 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 1: [0, 65536] x 2, Min 17 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 2: [0, 32768] x 4, Min 16 bits (bumped to **32 bits** for atomic operations)
Level 3: [0, 16384] x 8, Min 15 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 4: [0, 8192] x 16, Min 14 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 5: [0, 4096] x 32, Min 13 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 6: [0, 2048] x 64, Min 12 bits (rounded up to **32 bits** for alignment and atomic operations)

Level 7: [0, 1024] x 128, Min 11 bits (rounded up to **16 bits** for alignment)
Level 8: [0, 512] x 256, Min 10 bits (rounded up to **16 bits** for alignment)
Level 9: [0, 256] x 512, Min 9 bits (rounded up to **16 bits** for alignment)
Level 10: [0, 128] x 1024, Min **8 bits**

Level 17: Raw 64 bits representation

Book & Allocate & Cancel

The booking and cancelling is done using these functions



And this function that allows us to allocate the next available memory slot

Parallel Update Routine (CBT Implementation 128K)

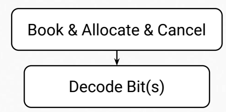
SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

```
// Graphics Buffers that hold the CBT
RWStructuredBuffer<uint32_t> _TreeBufferRW: register(CBT_BUFFER0_BINDING_SLOT); // RED + GREEN
RWStructuredBuffer<uint64_t> _BitfieldBufferRW: register(CBT_BUFFER1_BINDING_SLOT); // BLUE
RWStructuredBuffer<uint32_t> _AllocationBufferRW: register(CBT_BUFFER2_BINDING_SLOT);

// _TreeBufferRW in the shared memory
const uint32_t tree_num_slots = (1 * 32 + 2 * 32 + ... + 1024 * 8) / 32;
groupshared uint32_t gs_cbtTree[tree_num_slots]; // ~3KB

// Memory booking
void cbt_new_frame();
void book_memory_space(uint32_t numSlots);
void cancel_memory_booking(uint32_t numSlots);

// Allocation
uint32_t allocate_next_available_slot(uint32_t bisectorID);
```

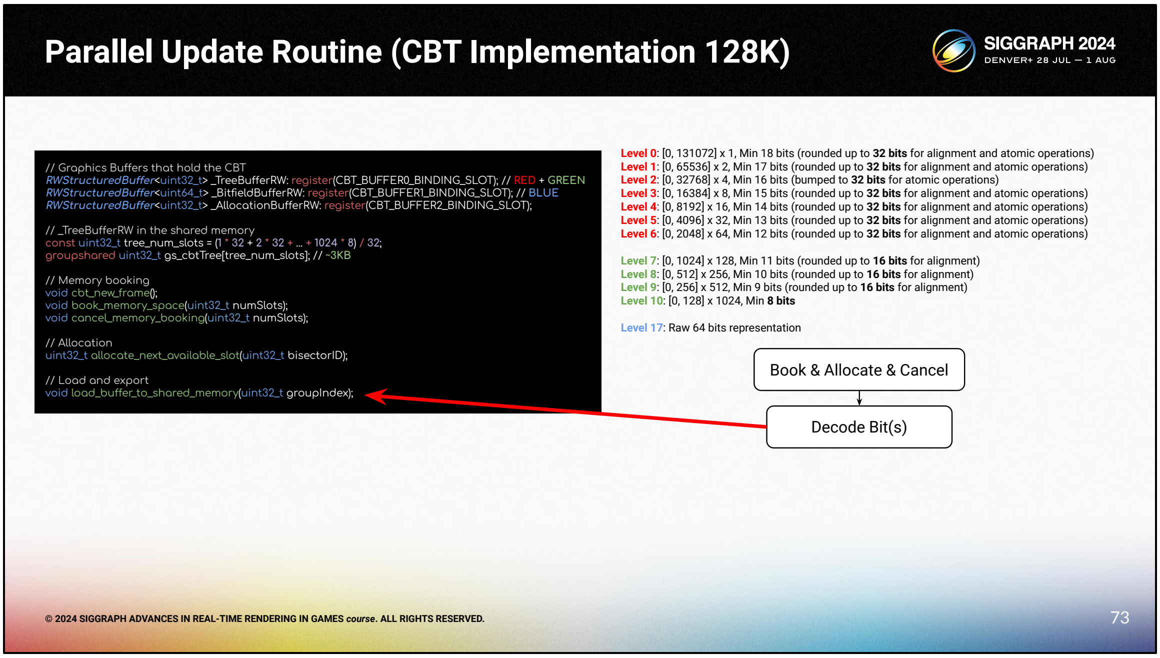
Level 0: [0, 131072] x 1, Min 18 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 1: [0, 65536] x 2, Min 17 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 2: [0, 32768] x 4, Min 16 bits (bumped to **32 bits** for atomic operations)
Level 3: [0, 16384] x 8, Min 15 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 4: [0, 8192] x 16, Min 14 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 5: [0, 4096] x 32, Min 13 bits (rounded up to **32 bits** for alignment and atomic operations)
Level 6: [0, 2048] x 64, Min 12 bits (rounded up to **32 bits** for alignment and atomic operations)

Level 7: [0, 1024] x 128, Min 11 bits (rounded up to **16 bits** for alignment)
Level 8: [0, 512] x 256, Min 10 bits (rounded up to **16 bits** for alignment)
Level 9: [0, 256] x 512, Min 9 bits (rounded up to **16 bits** for alignment)
Level 10: [0, 128] x 1024, Min **8 bits**

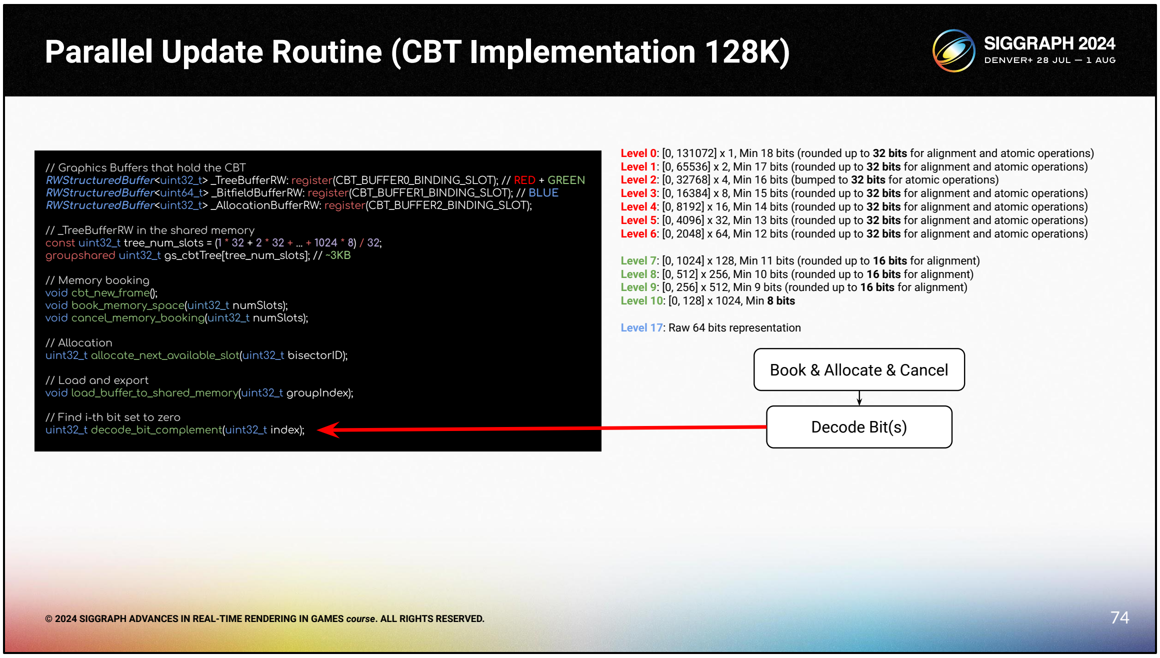
Level 17: Raw 64 bits representation



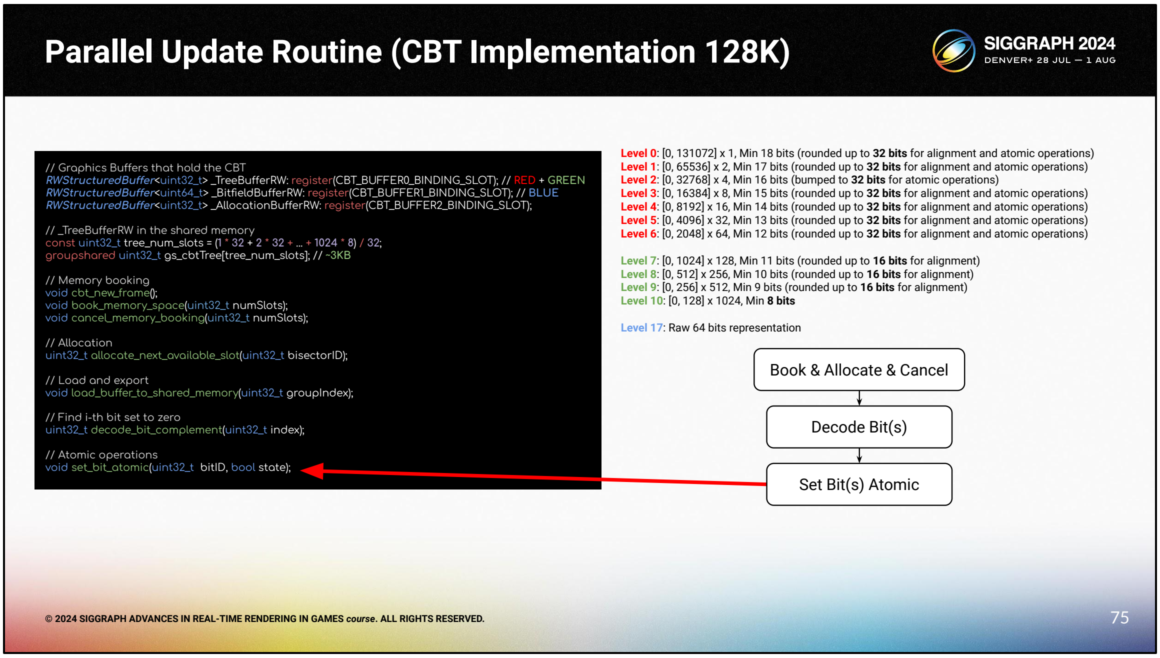
Then once all the allocations have been done, we’re gonna decode the location of the bits that we’ve allocated



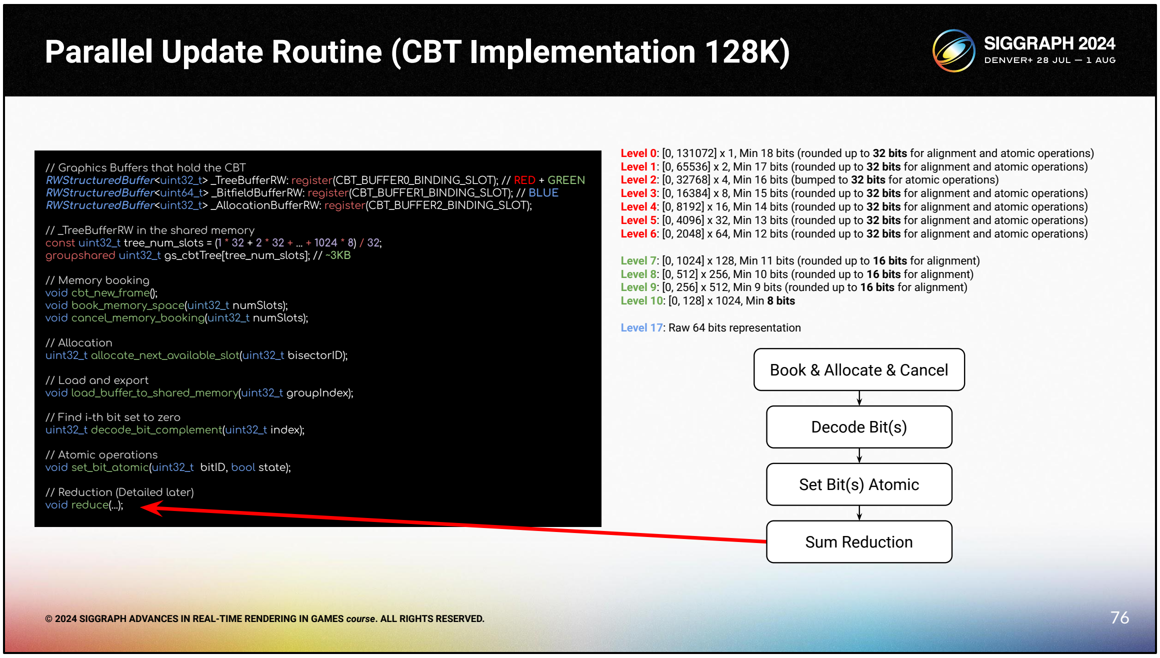
To do so, we first have to load the CBT into shared memory for our searches, we do it using this function, this will copy per workgroup the tree buffer into the group shader memory



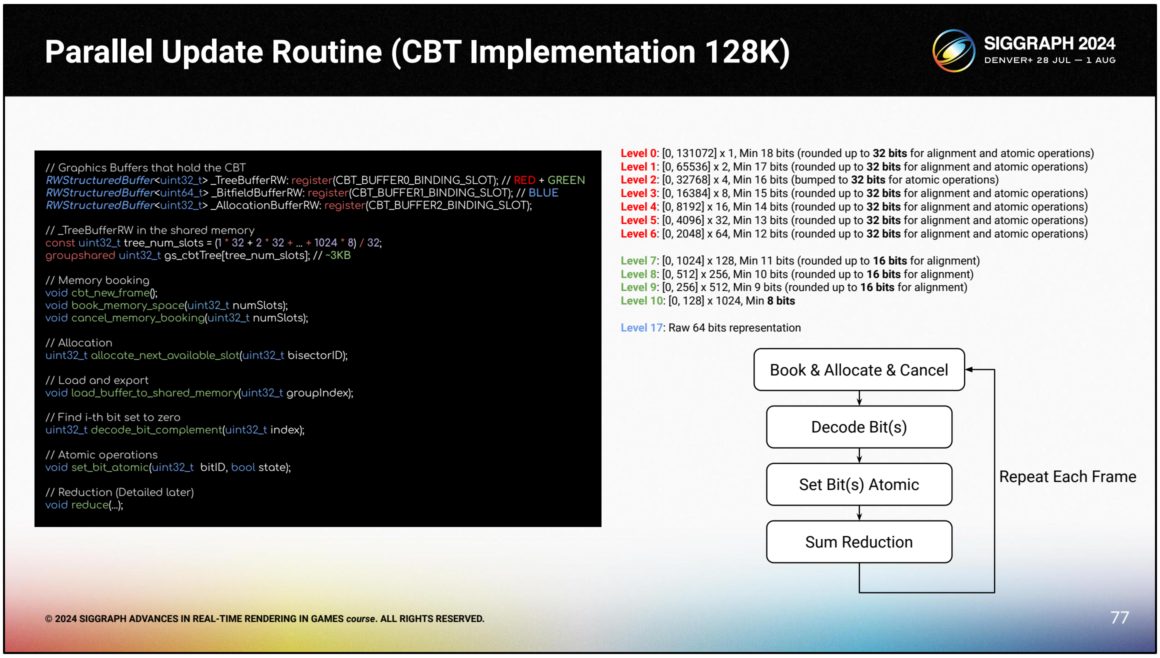
Then we actually operate the tree descents taking advantage of the very specific memory layout of the CBT which helps us to accelerate significantly the operation.



Then we’ll set atomically some bit to 1 (for the allocations that we’ve done) and 0 for the bit’s we’ve deallocated.
It is done using this function that will operate directly on the bitfield buffer.



And finally we'll operate a sum reduction on the tree to be able to return to a valid state. We can then take advantage of the memory layout we defined to get it performant, but i'll cover it a tad later.



And we’ll repeat this operation every frame and that defines how the CBT works as a parallel memory manager

Parallel Update Routine (Memory Pool)

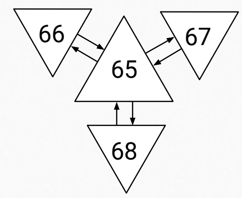
SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

```
// Graphics Buffers that hold the Memory pool
```


On the other side, we need to specify what is the memory pool like and how it’s used

Parallel Update Routine (Memory Pool)

```
// Graphics Buffers that hold the Memory pool
RWStructuredBuffer<uint64_t> _HeapIDBufferRW: register(POOL_BUFFER0_BINDING_SLOT);
RWStructuredBuffer<uint3> _NeighborsBufferRW: register(POOL_BUFFER1_BINDING_SLOT);
```

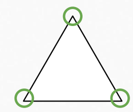


As mentioned before, we need two buffers to store the HeapID and Neighbors

Parallel Update Routine (Memory Pool)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

```
// Graphics Buffers that hold the Memory pool
RWStructuredBuffer<uint64_t> _HeapIDBufferRW: register(POOL_BUFFER0_BINDING_SLOT);
RWStructuredBuffer<uint3> _NeighborsBufferRW: register(POOL_BUFFER1_BINDING_SLOT);
RWStructuredBuffer<float3> _PositionRWSBufferRW: register(POOL_BUFFER2_BINDING_SLOT);
```

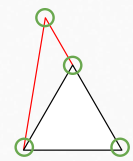


Then we also need to be able to store the camera relative position. Obviously, there are the three positions of the vertices of the triangles

Parallel Update Routine (Memory Pool)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

```
// Graphics Buffers that hold the Memory pool
RWStructuredBuffer<uint64_t> _HeapIDBufferRW: register(POOL_BUFFER0_BINDING_SLOT);
RWStructuredBuffer<uint3> _NeighborsBufferRW: register(POOL_BUFFER1_BINDING_SLOT);
RWStructuredBuffer<float3> _PositionRWSBufferRW: register(POOL_BUFFER2_BINDING_SLOT);
```



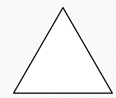
4 positions per bisector

But we actually need to store a 4th position which allows to rebuild the parent triangle and is required for split/merge decisions

Parallel Update Routine (Memory Pool)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

```
// Graphics Buffers that hold the Memory pool
RWStructuredBuffer<uint64_t> _HeapIDBufferRW: register(POOL_BUFFER0_BINDING_SLOT);
RWStructuredBuffer<uint3> _NeighborsBufferRW: register(POOL_BUFFER1_BINDING_SLOT);
RWStructuredBuffer<float3> _PositionRWSBufferRW: register(POOL_BUFFER2_BINDING_SLOT);
RWStructuredBuffer<UpdateData> _UpdateDataBuffer: register(POOL_BUFFER3_BINDING_SLOT);

struct UpdateData
{
};
```

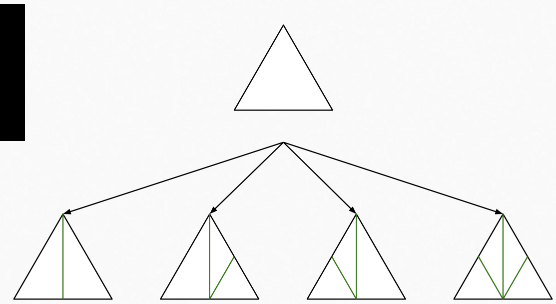


In addition to that, we need some temporary data, that we’ll call those update data

Parallel Update Routine (Memory Pool)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

```
// Graphics Buffers that hold the Memory pool
RWStructuredBuffer<uint64_t> _HeapIDBufferRW: register(POOL_BUFFER0_BINDING_SLOT);
RWStructuredBuffer<uint3> _NeighborsBufferRW: register(POOL_BUFFER1_BINDING_SLOT);
RWStructuredBuffer<float3> _PositionRWSBufferRW: register(POOL_BUFFER2_BINDING_SLOT);
RWStructuredBuffer<UpdateData> _UpdateDataBuffer: register(POOL_BUFFER3_BINDING_SLOT);

struct UpdateData
{
};
```

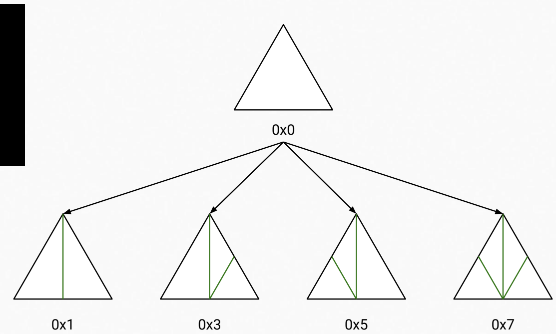


At each update loop, a bisector can be in one of 5 states, unchanged, single split, double splits (two cases) or triple split.

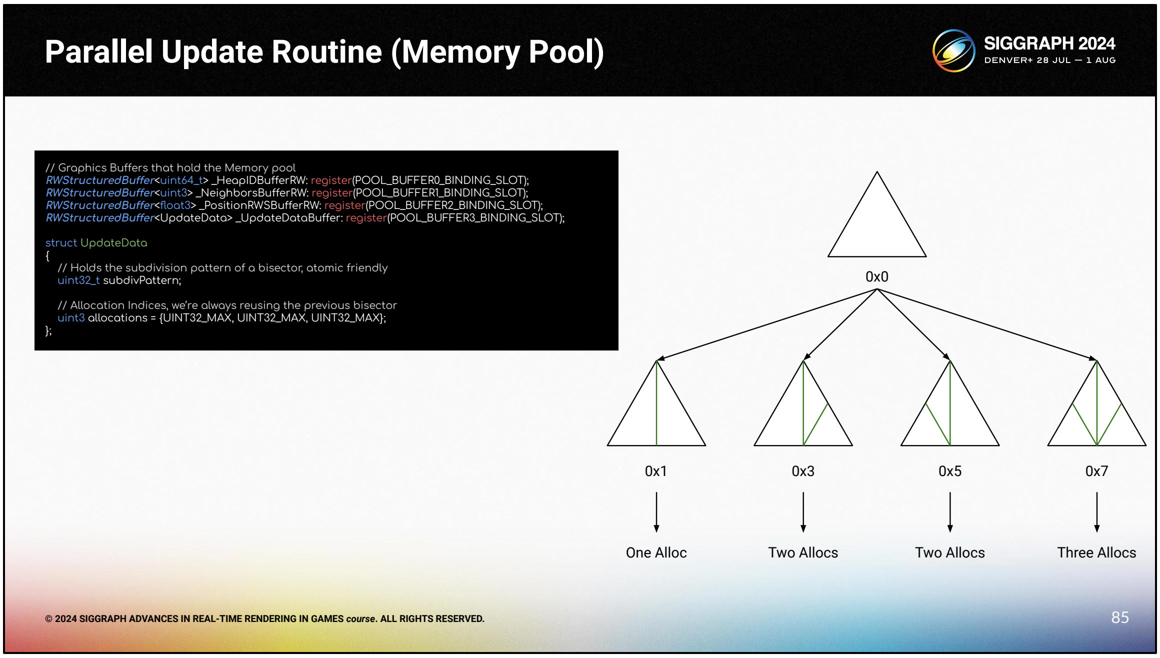
Parallel Update Routine (Memory Pool)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

```
// Graphics Buffers that hold the Memory pool
RWStructuredBuffer<uint64_t> _HeapIDBufferRW: register(POOL_BUFFER0_BINDING_SLOT);
RWStructuredBuffer<uint3> _NeighborsBufferRW: register(POOL_BUFFER1_BINDING_SLOT);
RWStructuredBuffer<float3> _PositionRWSBufferRW: register(POOL_BUFFER2_BINDING_SLOT);
RWStructuredBuffer<UpdateData> _UpdateDataBuffer: register(POOL_BUFFER3_BINDING_SLOT);

struct UpdateData
{
    // Holds the subdivision pattern of a bisector, atomic friendly
    uint32_t subdivPattern;
};
```



We encode that split state into three bits and store it in a unit32_t to use the atomic intrinsics



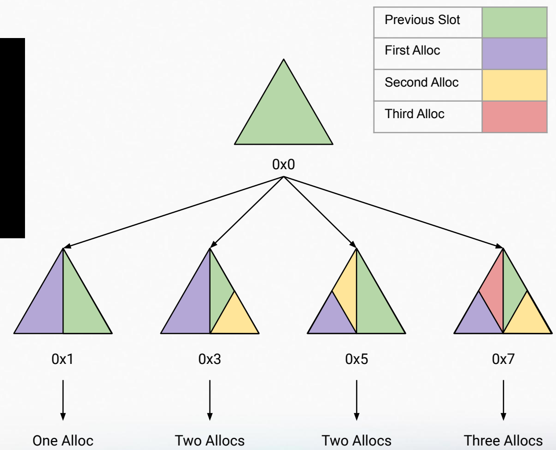
Depending on the split, we’ll need up to 3 news slots to store the location of produced bisectors.

In this implementation, we always reuse the current bisector slot for performance and compactness reasons

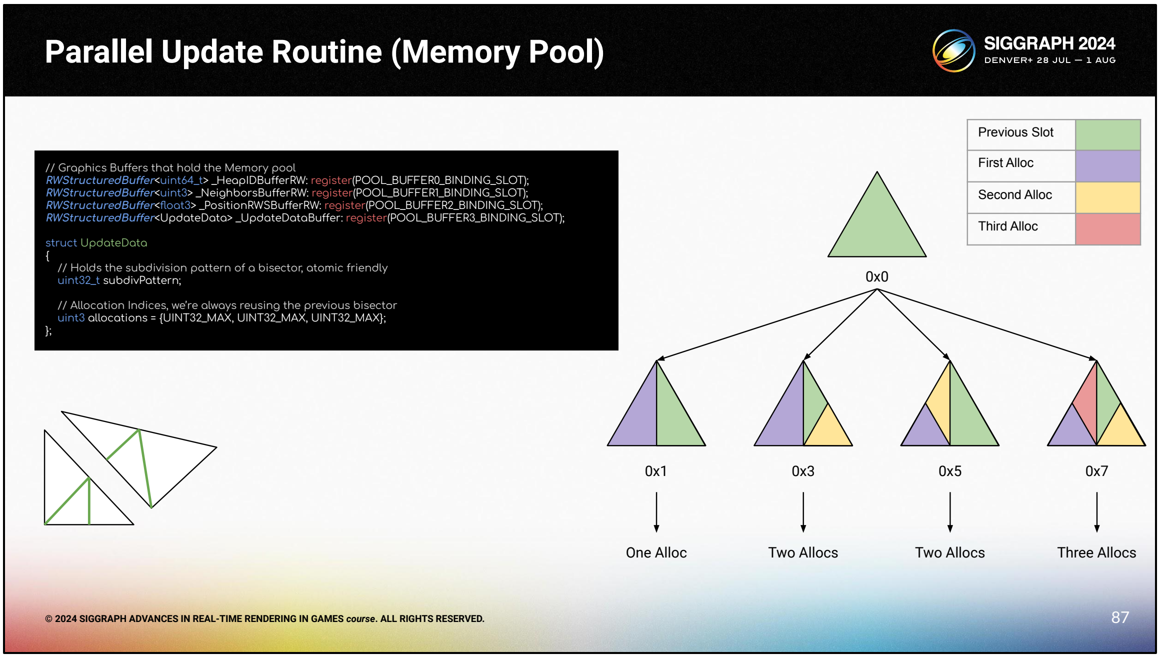
Parallel Update Routine (Memory Pool)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

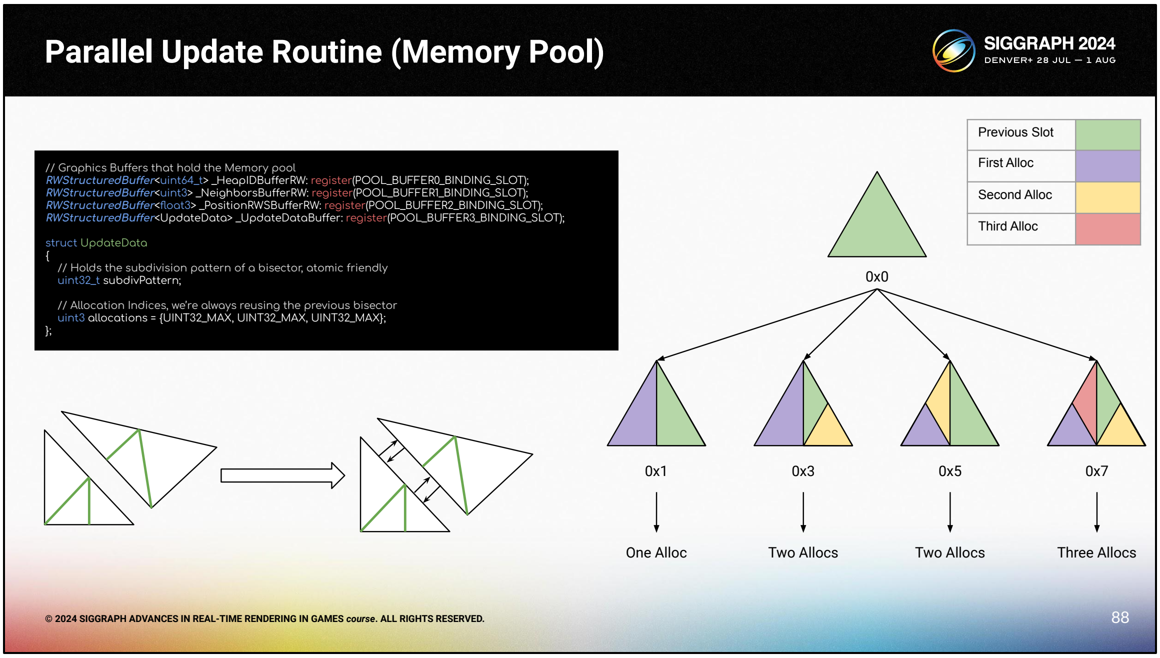
```
// Graphics Buffers that hold the Memory pool
RWStructuredBuffer<uint64_t> _HeapIDBufferRW: register(POOL_BUFFER0_BINDING_SLOT);
RWStructuredBuffer<uint3> _NeighborsBufferRW: register(POOL_BUFFER1_BINDING_SLOT);
RWStructuredBuffer<float3> _PositionRWSBufferRW: register(POOL_BUFFER2_BINDING_SLOT);
RWStructuredBuffer<UpdateData> _UpdateDataBuffer: register(POOL_BUFFER3_BINDING_SLOT);

struct UpdateData
{
    // Holds the subdivision pattern of a bisector, atomic friendly
    uint32_t subdivPattern;

    // Allocation Indices, we're always reusing the previous bisector
    uint3 allocations = {UINT32_MAX, UINT32_MAX, UINT32_MAX};
};
```



The way the indices are assigned has to be consistent



In that way, for any split combination at an interface between two bisectors



We’re able to predict the neighbor data using simply the update and neighbor data of the current and neighbor bisectors.

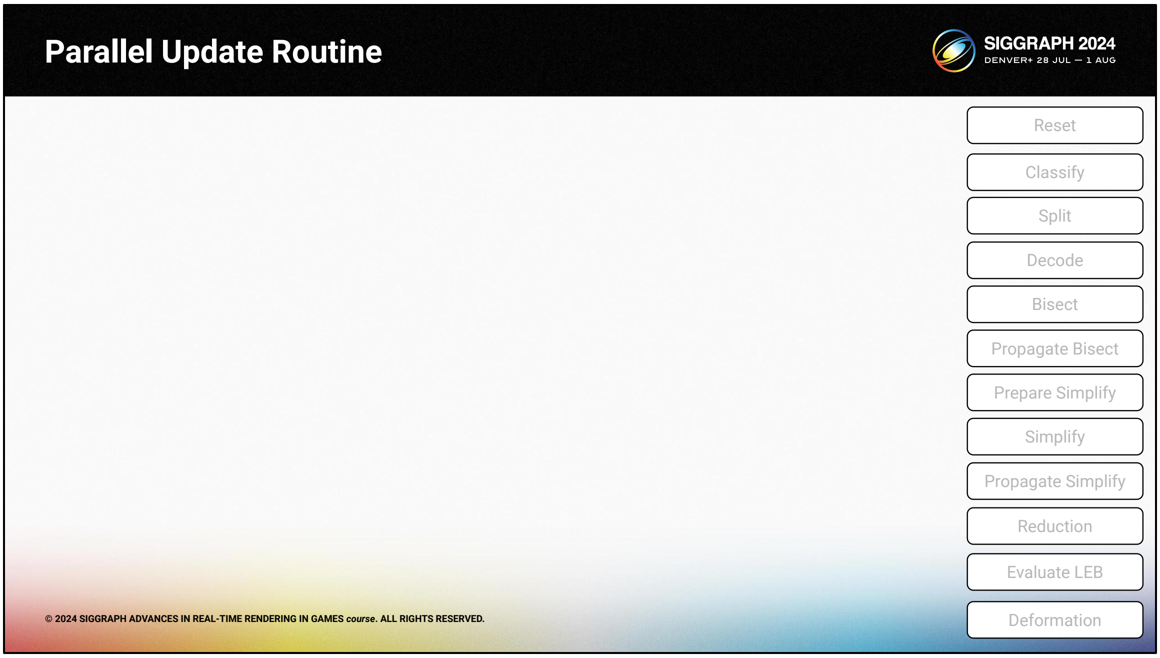
This is important and it is the cornerstone of the sync free parallel implementation

Parallel Update Routine (Memory Pool)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

```
// Graphics Buffers that hold the Memory pool
RWStructuredBuffer<uint64_t> _HeapIDBufferRW: register(POOL_BUFFER0_BINDING_SLOT);
RWStructuredBuffer<uint3> _NeighborsBufferRW: register(POOL_BUFFER1_BINDING_SLOT);
RWStructuredBuffer<float3> _PositionRWSBufferRW: register(POOL_BUFFER2_BINDING_SLOT);
RWStructuredBuffer<UpdateData> _UpdateDataBuffer: register(POOL_BUFFER3_BINDING_SLOT);

struct UpdateData
{
    // Holds the subdivision pattern of a bisector, atomic friendly
    uint32_t subdivPattern;

    // Allocation Indices, we're always reusing the previous bisector
    uint3 allocations = {UINT32_MAX, UINT32_MAX, UINT32_MAX};

    // Visibility and modification flags of a bisector
    uint32_t flags;

    // Used for patching neighbors after modifications
    uint2 propagation;
};
```


In addition to that we need an array to store additional flags and neighbors propagation data

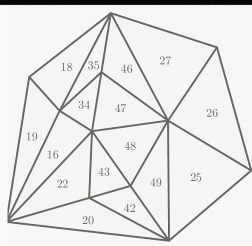


And with that, we've defined the CBT implementation and the Memory pool layout.

Now let's look at the update routine itself, each block on the right maps to one or multiple compute shader

Parallel Update Routine

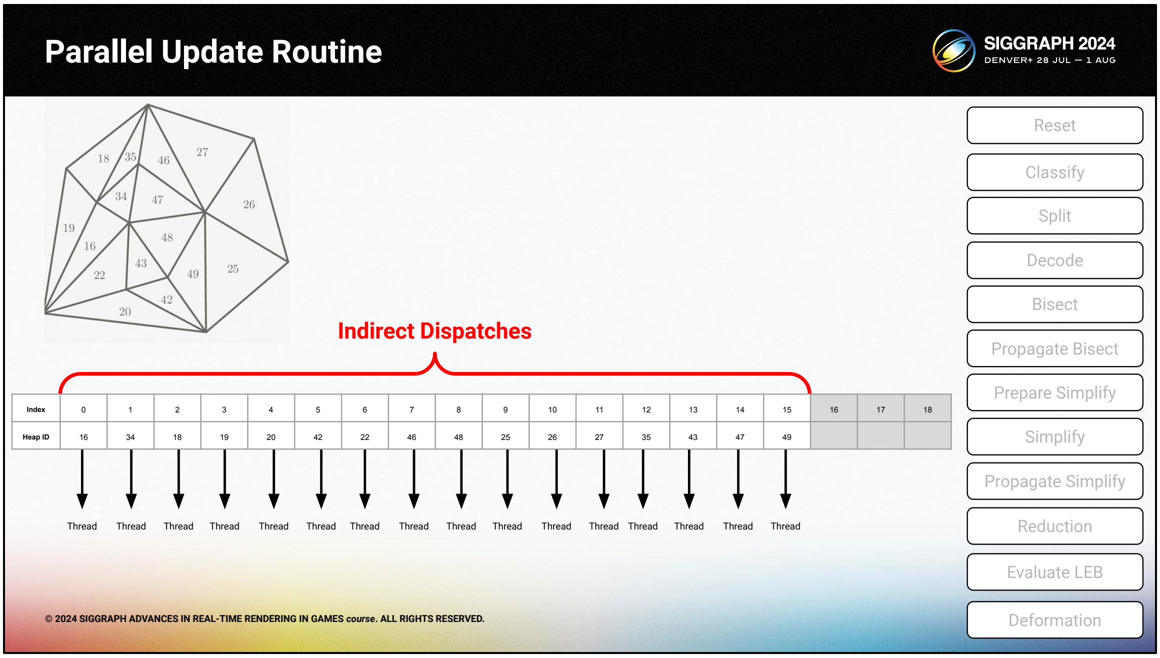
SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG



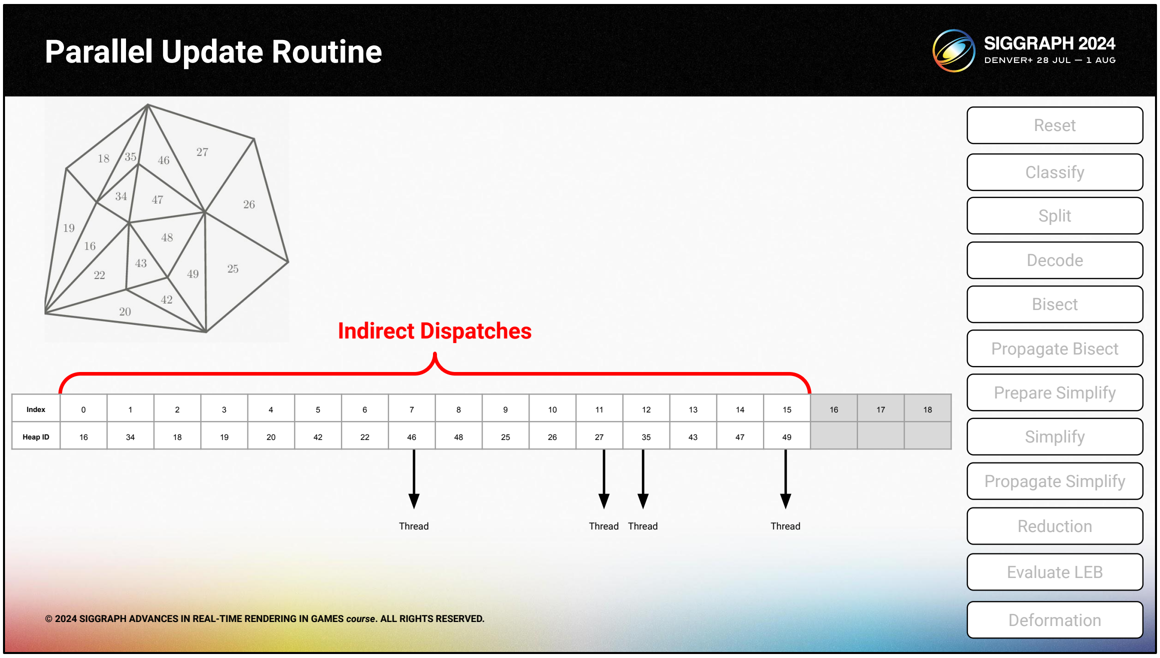
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Heap ID	16	34	18	19	20	42	22	46	48	25	26	27	35	43	47	49			

Reset
Classify
Split
Decode
Bisect
Propagate Bisect
Prepare Simplify
Simplify
Propagate Simplify
Reduction
Evaluate LEB
Deformation

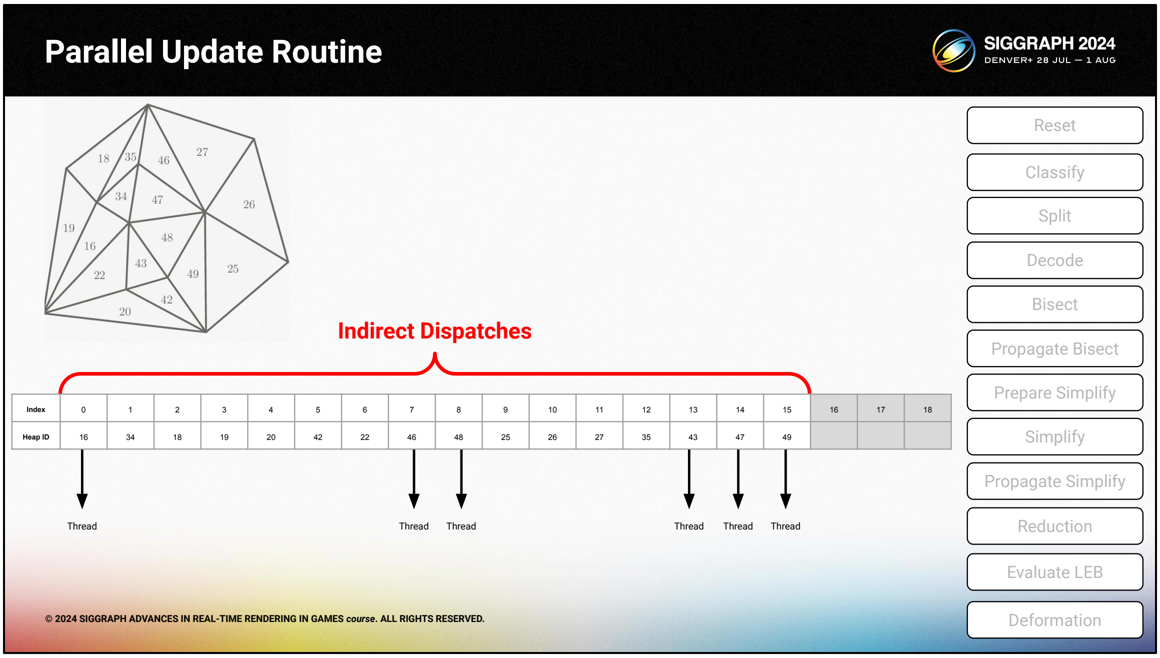
So for a given triangulation, there is a corresponding memory pool



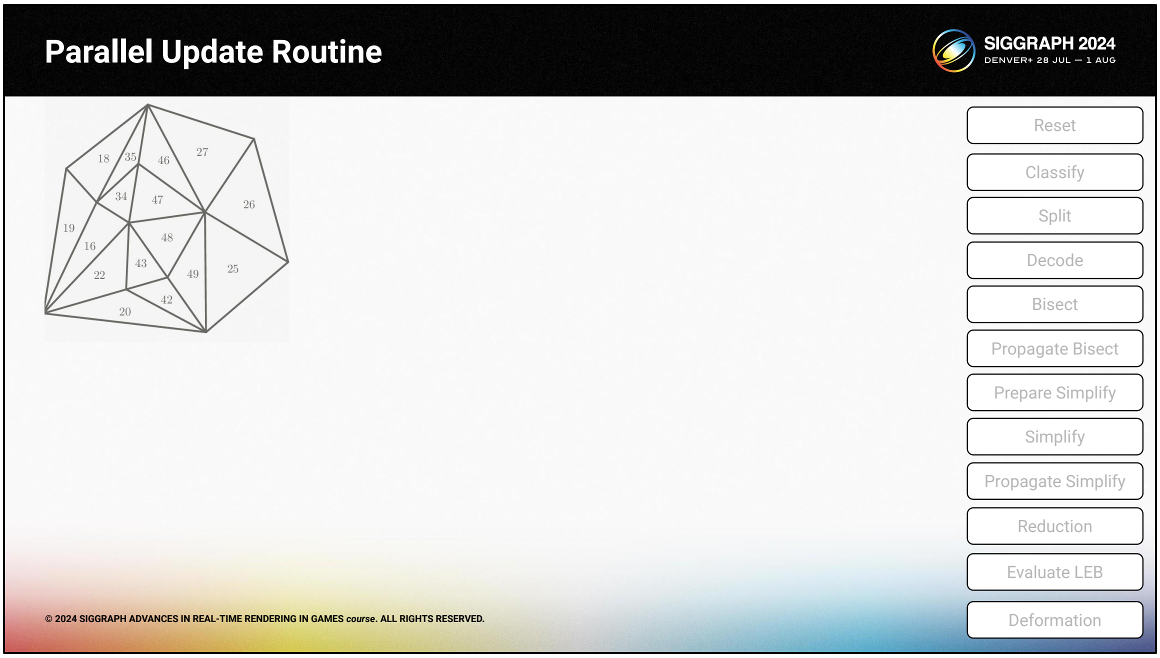
Most of the passes will be an indirect dispatch based on the a given set of bisectors.



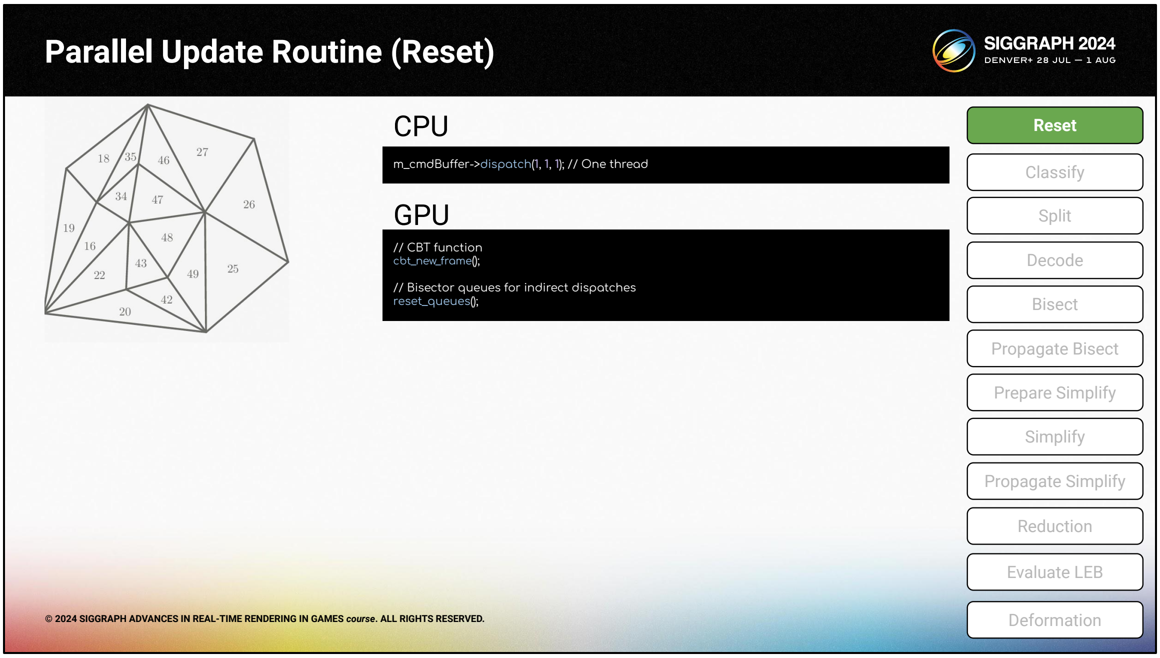
The set of bisectors will change based on the pass



and the subset will be specified at each pass of the pipeline



Alright, the first step of the pipeline is “Reset”

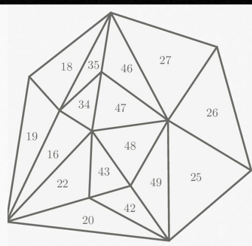


This pass is a single thread dispatch that:

- prepares the allocations for the frame
- resets the bisector queues for indirect dispatches

Parallel Update Routine (Classify)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG



CPU

```
m_cmdBuffer->dispatch_indirect(ACTIVE_BISECTORS); // HeapIDs: 16, 18, 19, .....48, 49
```

GPU

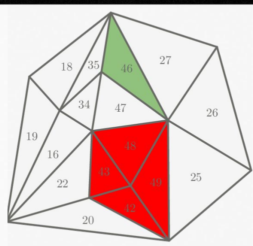
- Reset
- Classify
- Split
- Decode
- Bisect
- Propagate Bisect
- Prepare Simplify
- Simplify
- Propagate Simplify
- Reduction
- Evaluate LEB
- Deformation

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Heap ID	16	34	18	19	20	42	22	46	48	25	26	27	35	43	47	49			
State																			

For the second pass, “Classify”, each thread will run for one active bisector

Parallel Update Routine (Classify)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG



CPU

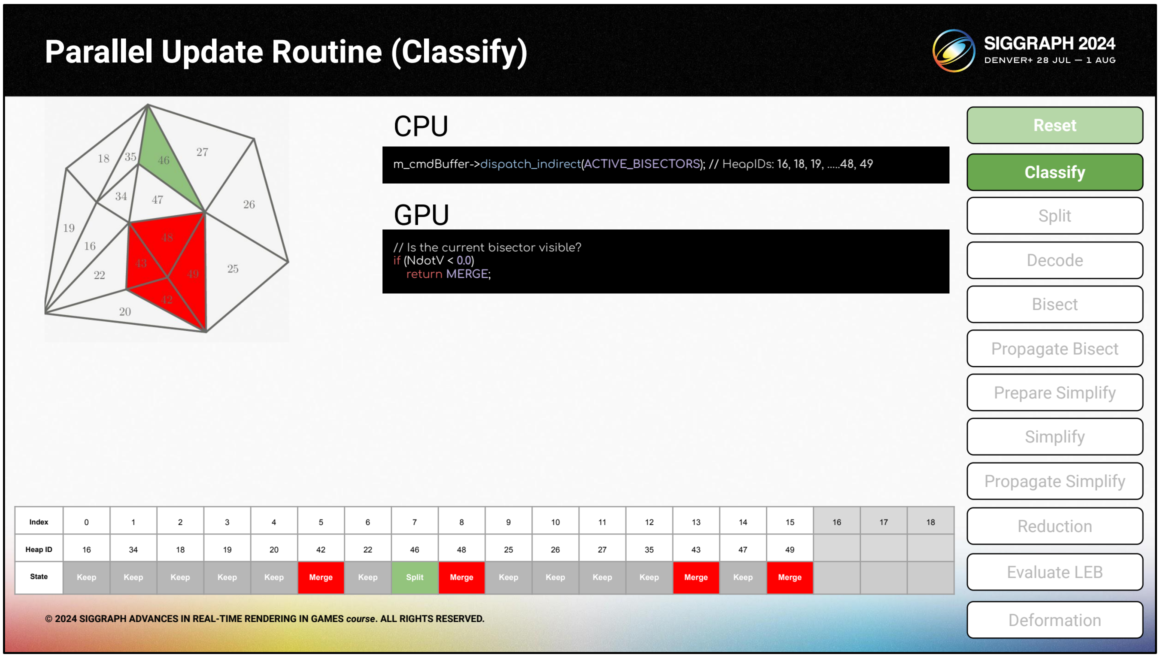
```
m_cmdBuffer->dispatch_indirect(ACTIVE_BISECTORS); // HeapIDs: 16, 18, 19, .....48, 49
```

GPU

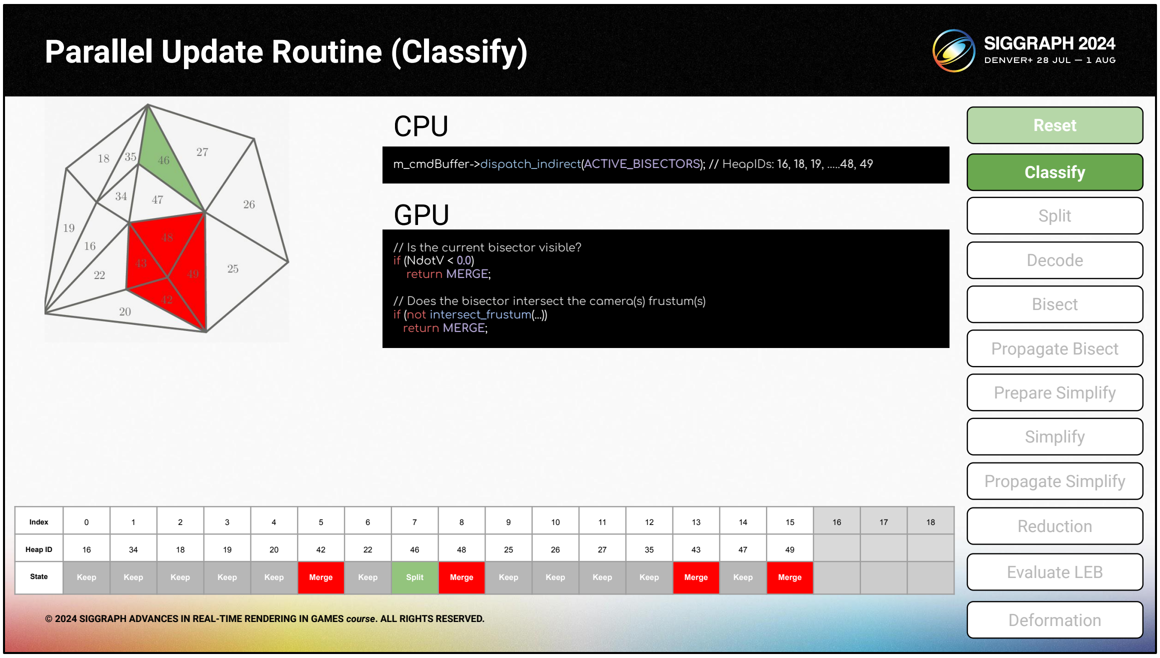
Reset
Classify
Split
Decode
Bisect
Propagate Bisect
Prepare Simplify
Simplify
Propagate Simplify
Reduction
Evaluate LEB
Deformation

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Heap ID	16	34	18	19	20	42	22	46	48	25	26	27	35	43	47	49			
State	Keep	Keep	Keep	Keep	Keep	Merge	Keep	Split	Merge	Keep	Keep	Keep	Keep	Merge	Keep	Merge			

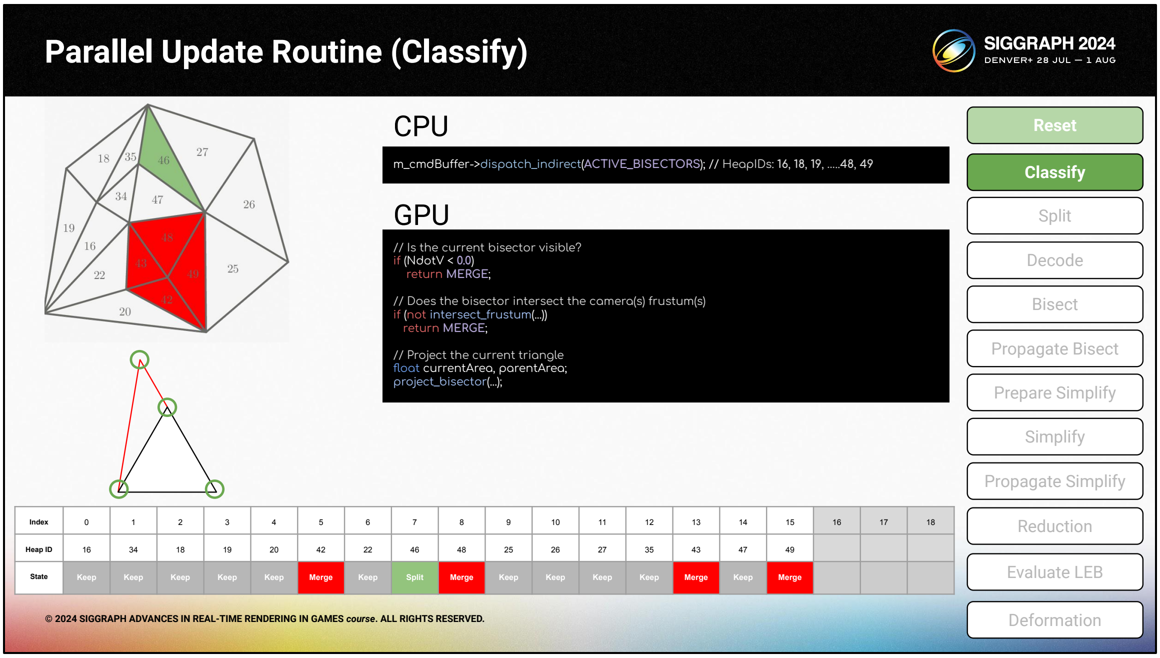
The idea is to classify each bisector in one of 3 states, Keep, Merge or Split. Depending on what we're trying to achieve the criteria for these operations can change, but for the demo here are the ones we used



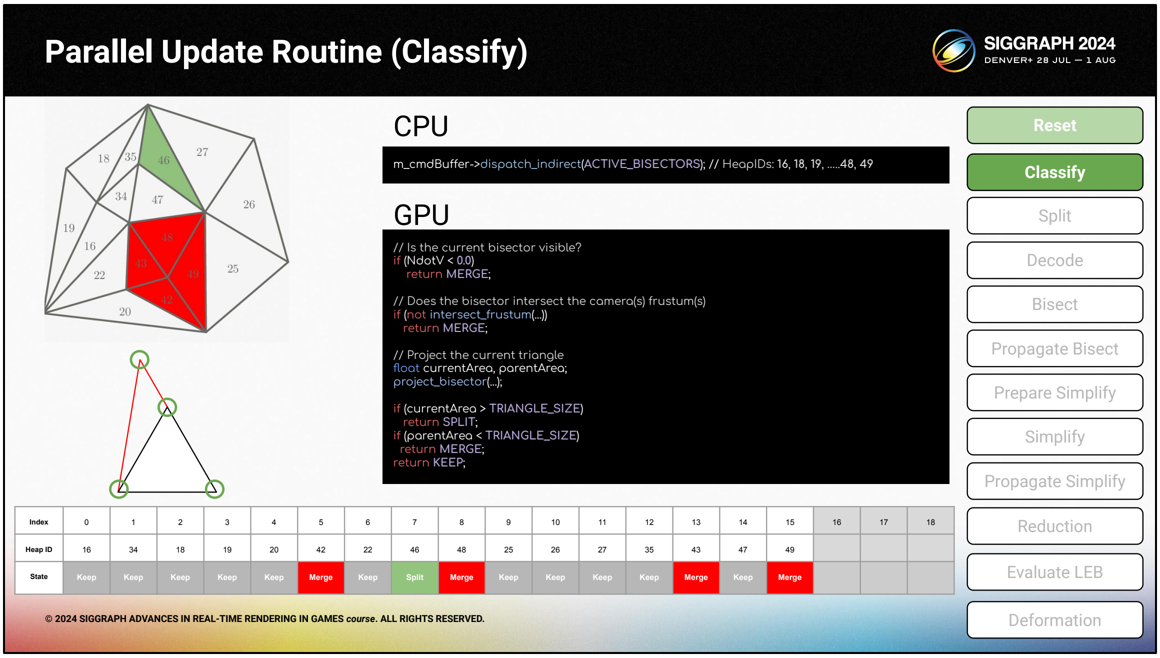
First, we do a NdotV test and flag it for merge if it fails



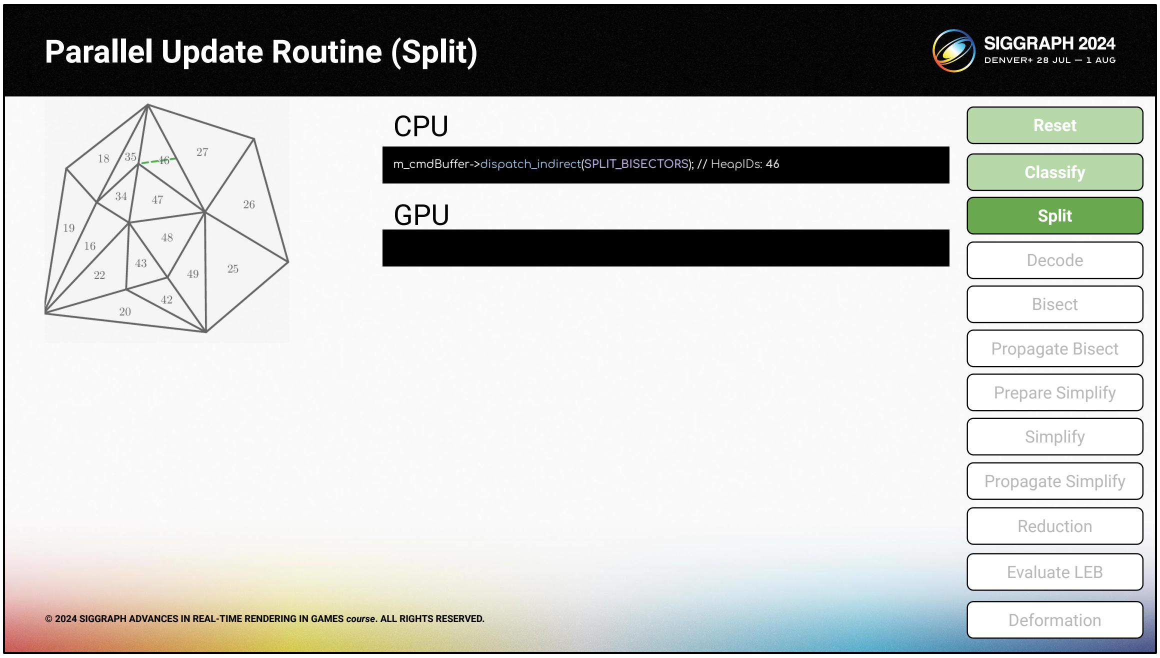
We then intersect one or multiple frustums and flag it for merge if it does not intersect any of them



Then we'll project the 4 vertices and evaluate the projected area of the triangle and the parent triangle

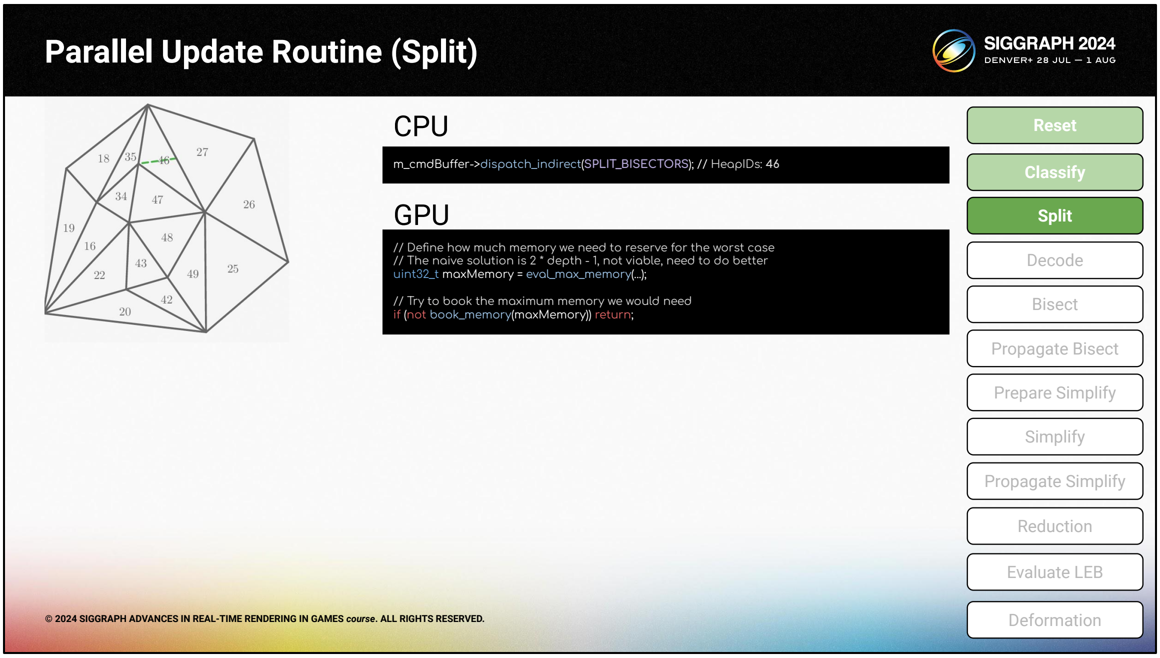


We compare those area to a triangle size that is defined by the application and flag the bisector for merge, split or keep depending on the result



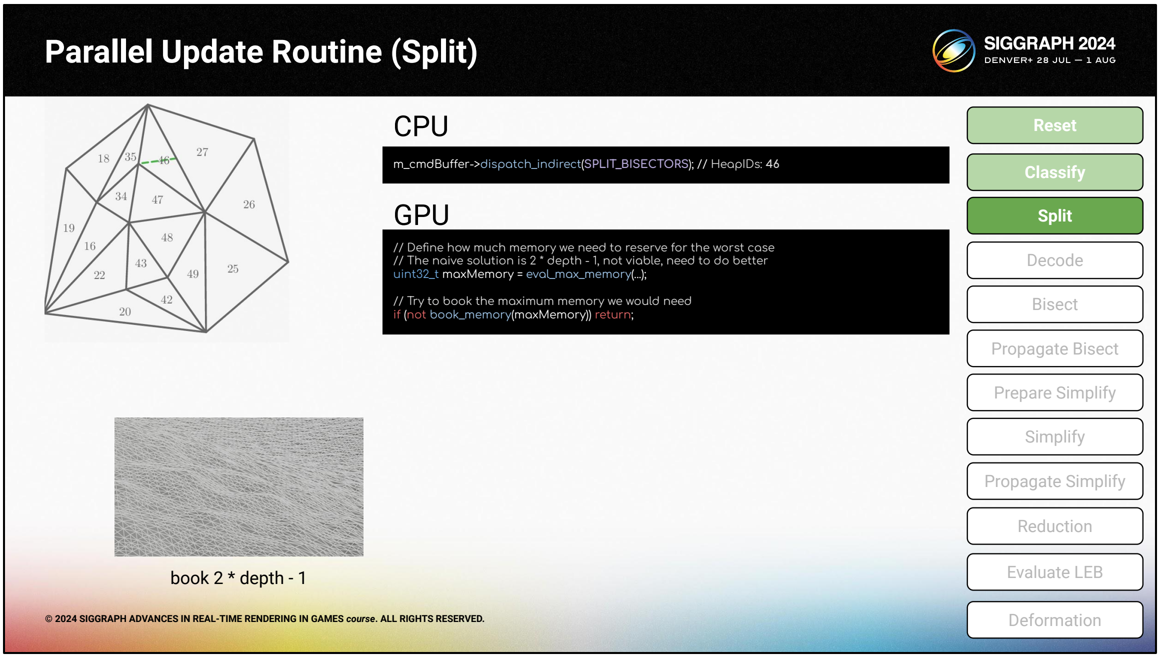
Then we move on to the “Split” pass, this will only run on the bisectors that have been tagged for as requiring a split.

In the example that would be only the bisector with the heapid 46.

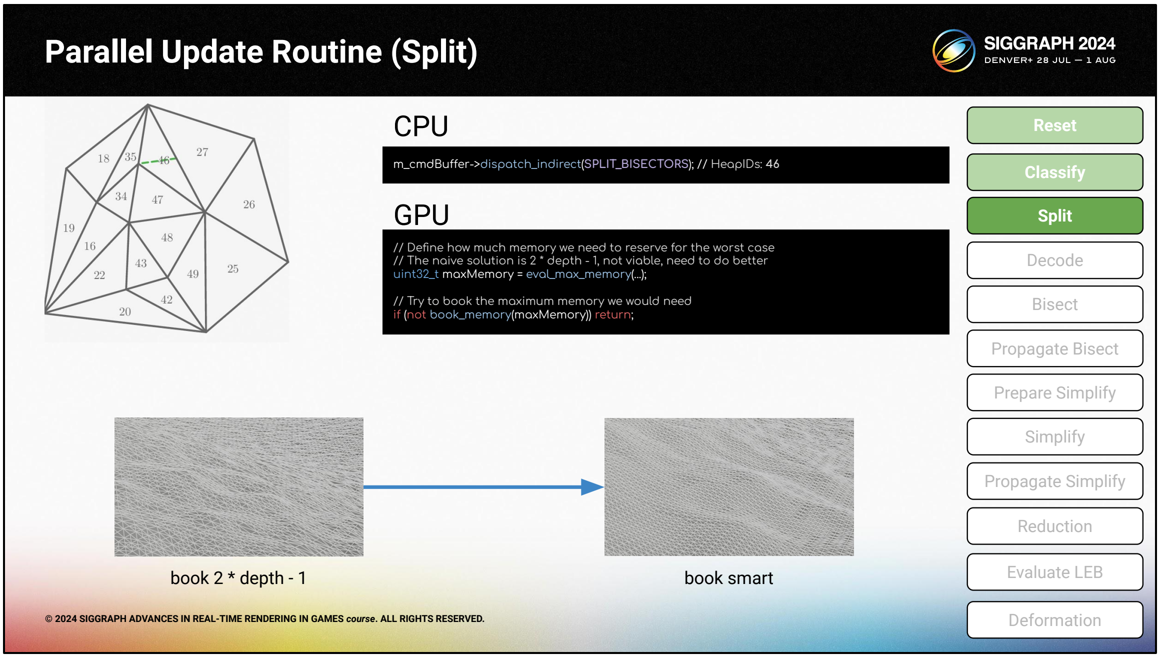


The first thing we need to do is figure out what is the worst case allocation size we need to guarantee the propagation of this split.

If we cannot guarantee the required memory space, the split cannot go through, that would break the LEB scheme

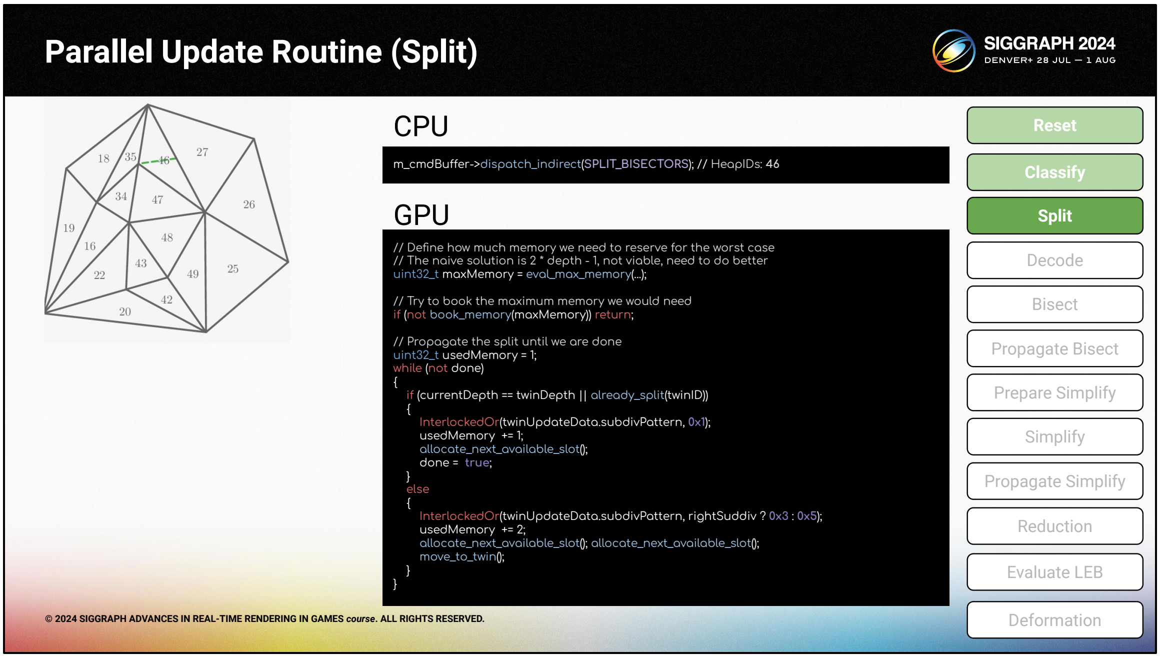


The naive solution that is 2 * depth - 1 ends up causing under tessellation issues when moving at a high speed and this is not viable

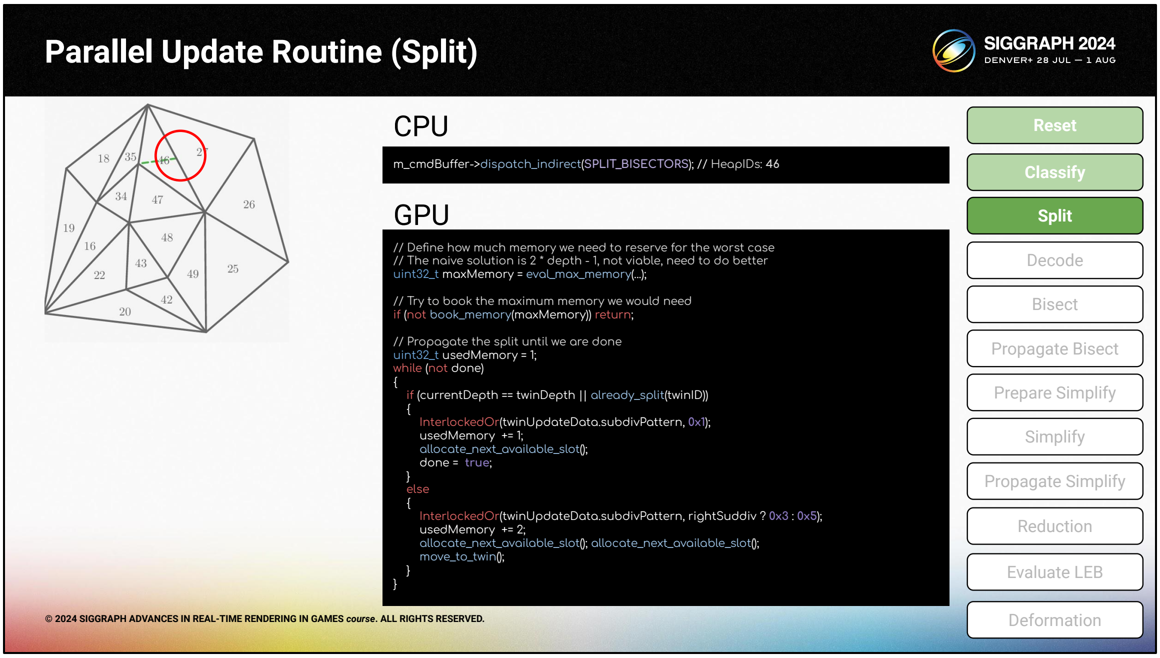


There is a better way of doing it that we'll not detail in the presentation but you can find the details in the source code we share

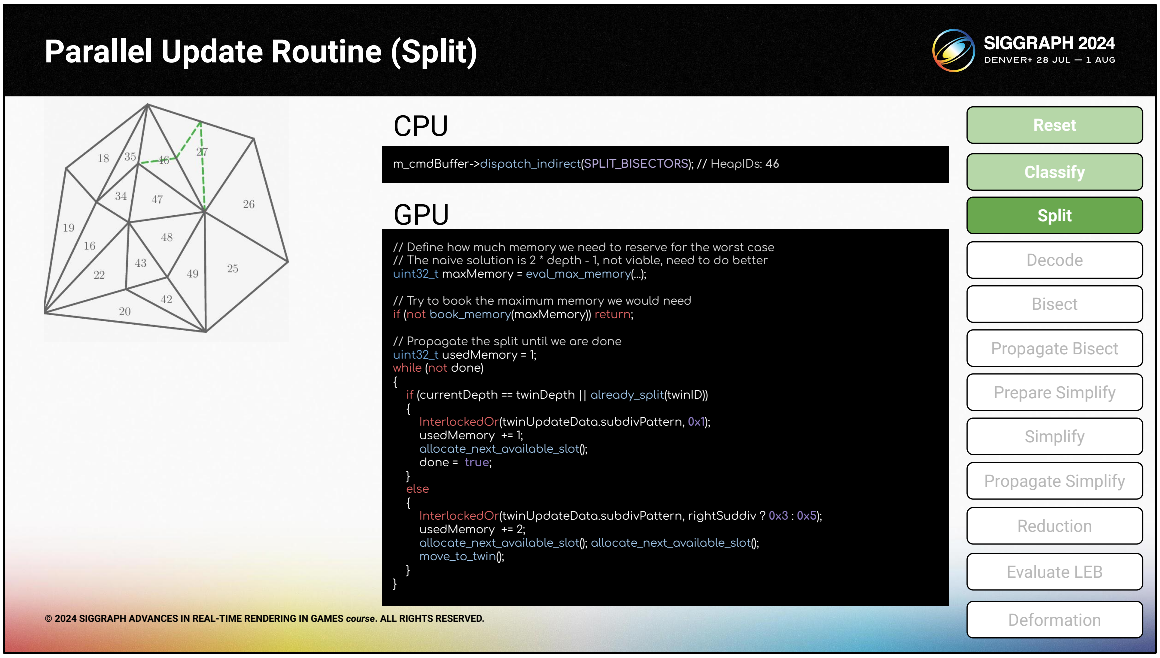
Once we have that estimation, we'll try to book that amount of memory from the CBT. Again we fail to guarantee that amount, this split won't go through as we cannot preserve the LEB scheme



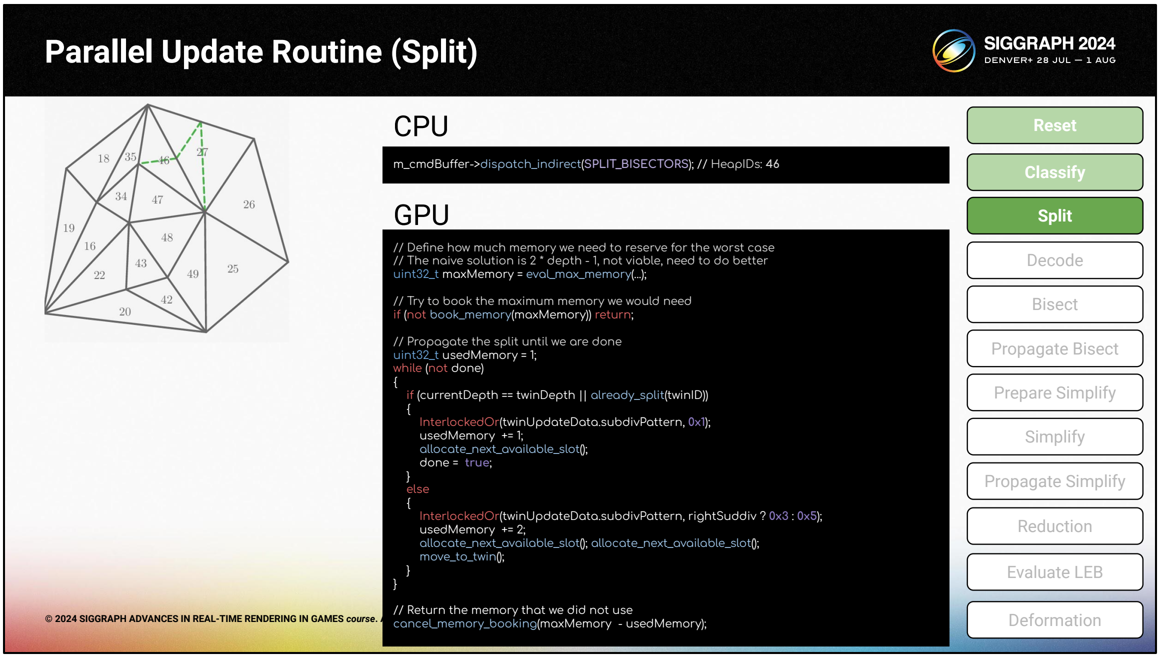
If we manage to book the worst case, we de-recursify the propagation algorithm into a while loop that will split each triangle, allocate one or multiple slots, propagate the subdivision to the twins all the bits we need until we're done.



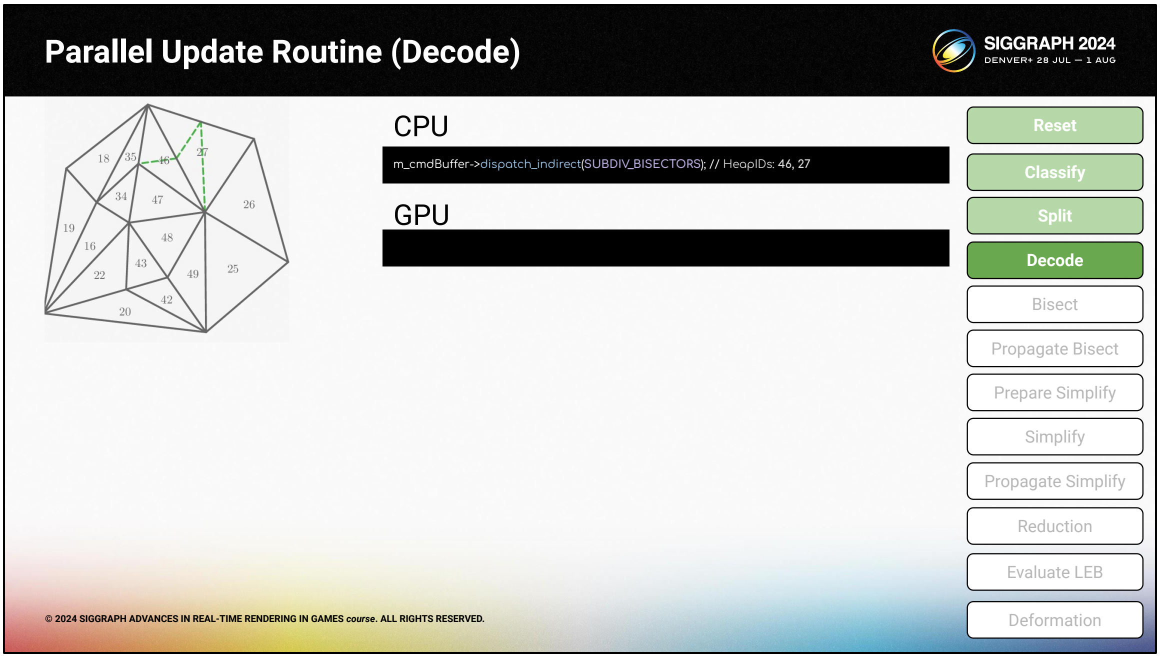
So in this case we do the first allocation, but we have a T junction so we continue on our path



Which leads us to doing two more allocations and we're good to go.

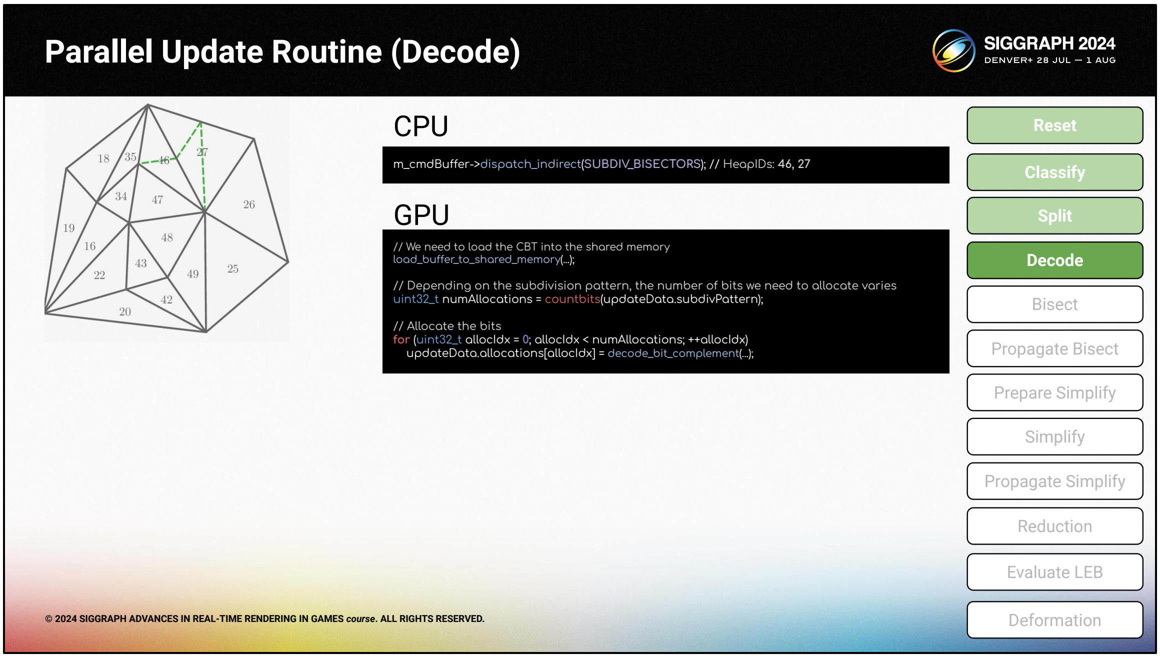


Finally we’re return to the CBT the memory space that we didn't allocate



The next step “Decode”, runs on the bisectors that have been flagged for a subdivision, (46 and 27 in this case),

This will associate each allocated slot with a free bit of the CBT

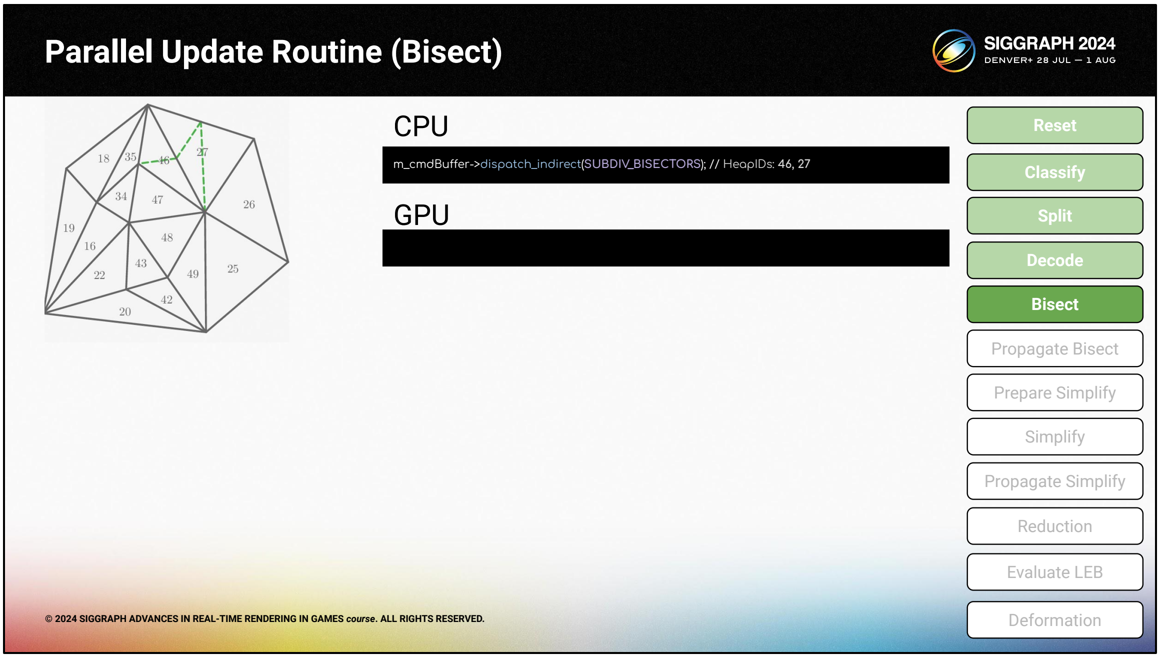


For this pass we need to load the CBT into the shared memory first

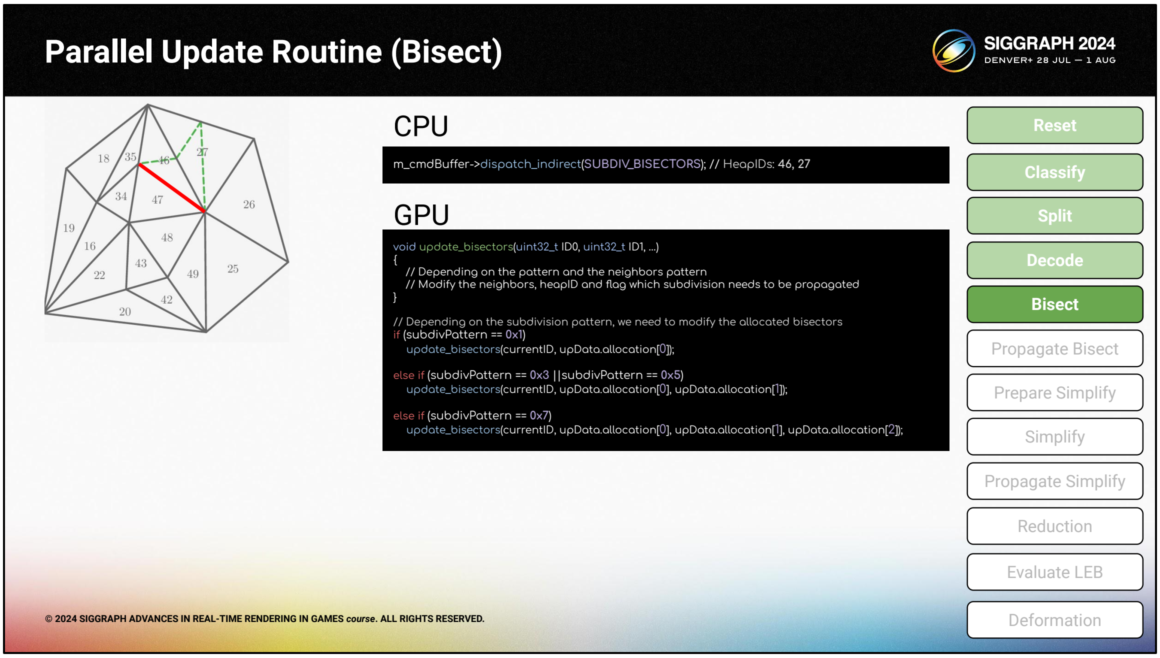
The number of bits set to 1 of the subdivision pattern gives us the number of decodes we need to do.

We do those decodes and store them in the update data

Once this is done, we move on to the next step

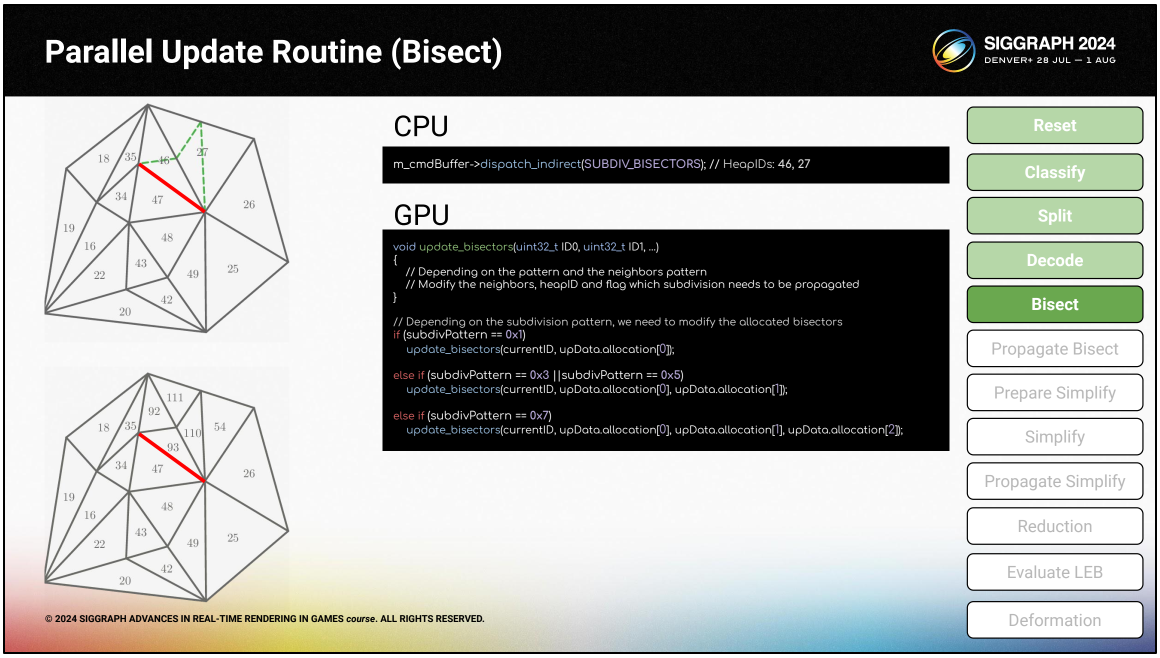


The following pass, “Bisect”, that runs on the same set of bisectors than the decode pass, aka every bisector that will be splitted

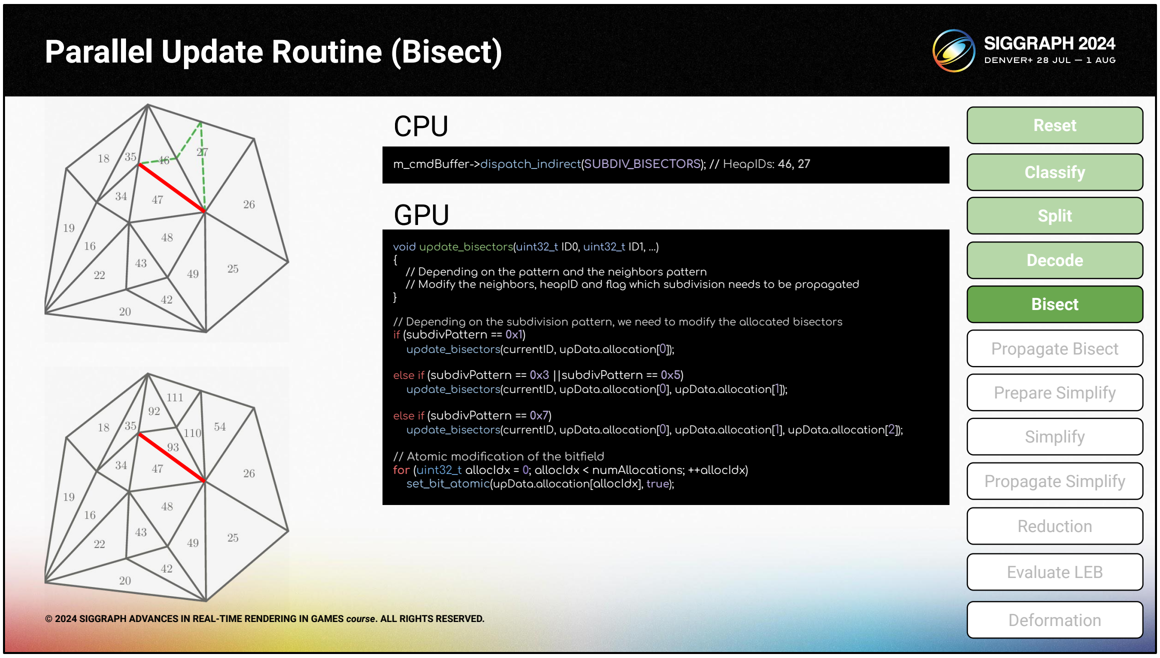


Based on the subdivision pattern of the current bisector and and the one of its neighbors, we'll modify the heap ID and adjust neighbor pointers.

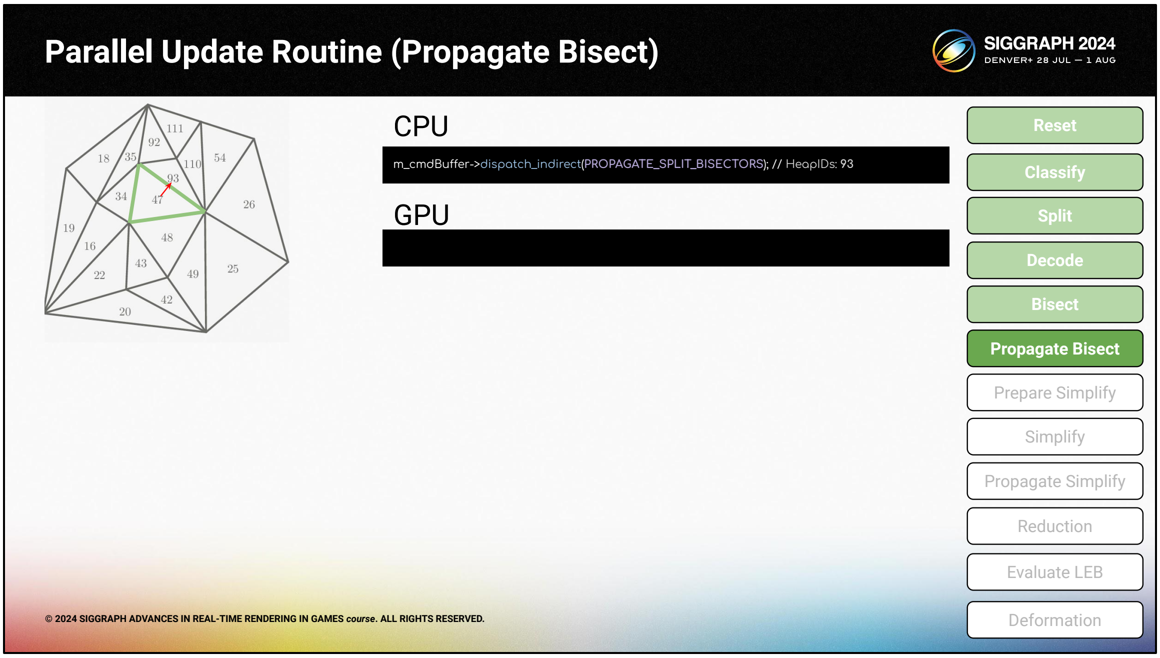
This only partially updates the neighbors and we still have problematic interfaces (marked in red in the example) that will be resolved in the following pass



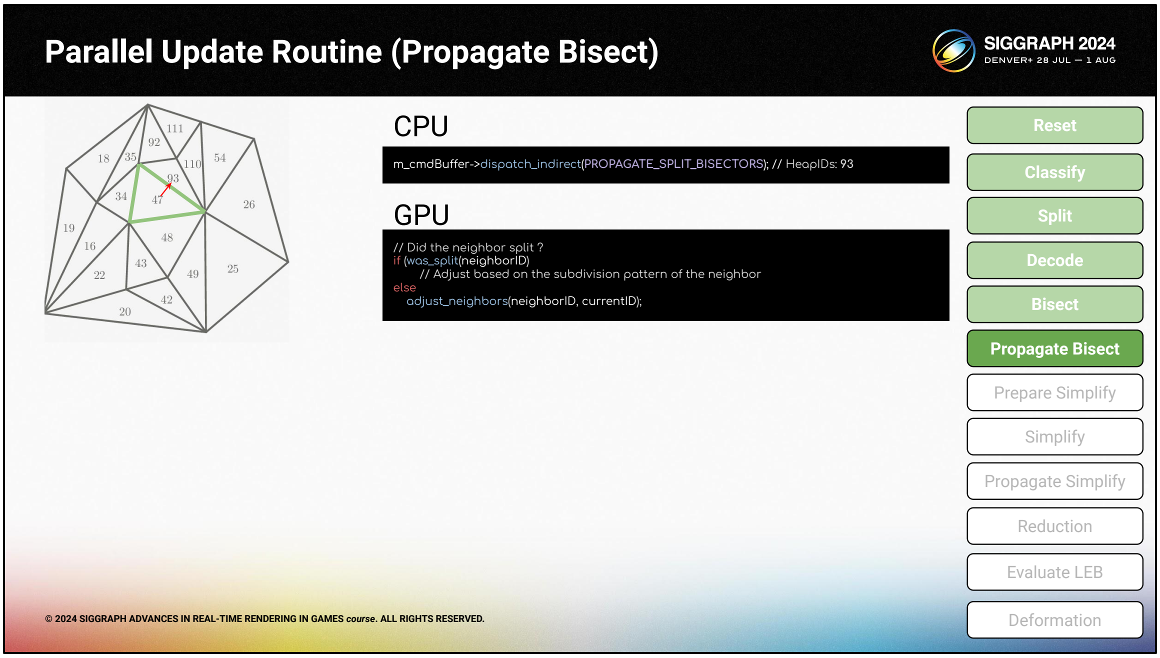
This will produce the new state of the triangulation



And finally we'll update the bits of the CBT based on the number of allocations that were required for this split

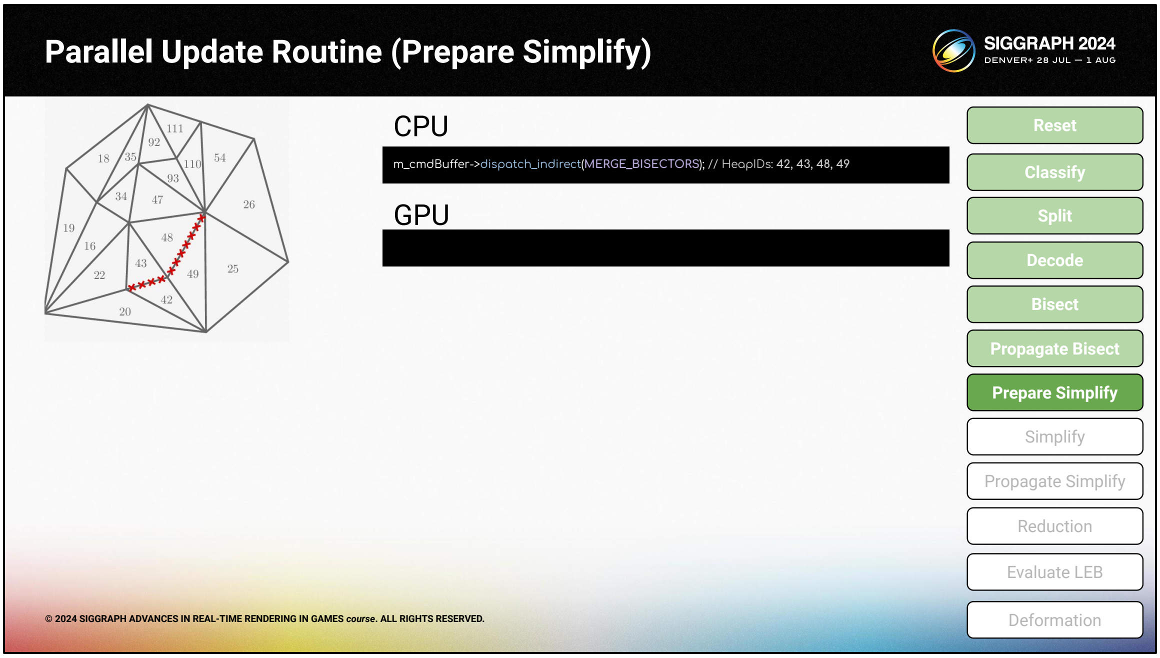


The next step, “Propagate Bisect”, will process these problematic interfaces by changing the neighbors when required

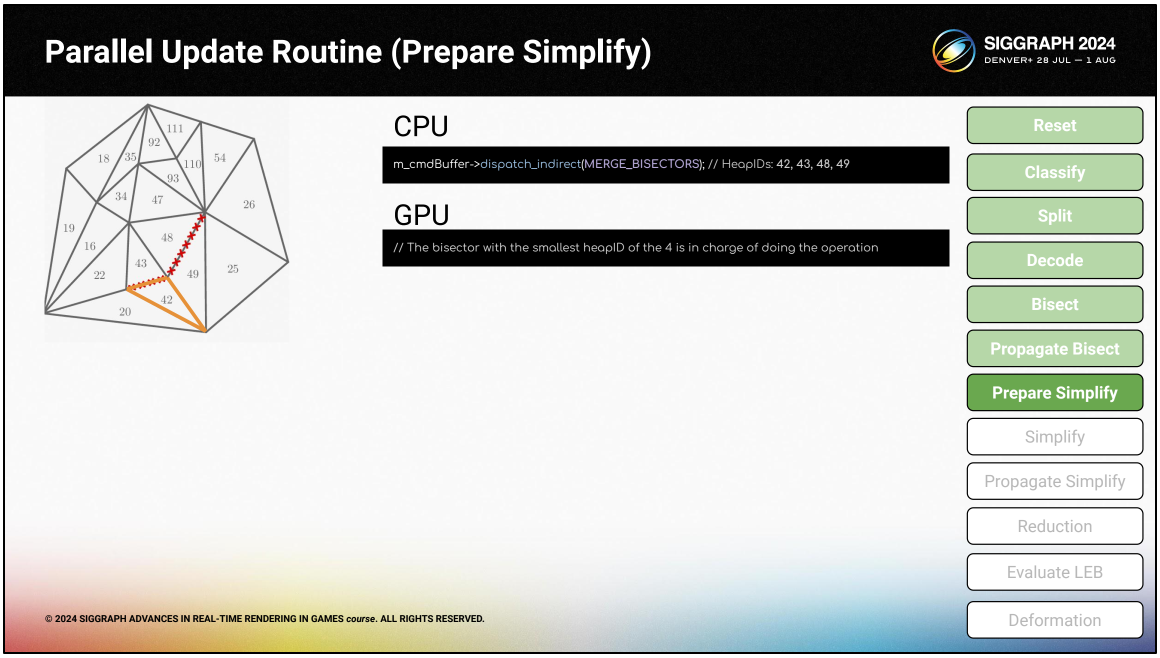


The patching routine has to take into account if the neighbor was split (and how it was split, once, twice or thrice) and adjust the neighbors accordingly

That covers it for the split part of the algorithm. Now we need to process the merges, It is roughly the same approach as the splits, but simpler as there are no propagation to account for.



The first pass is “Prepare simplify”, This dispatch will run on the 4 bisectors that have been flagged for simplification



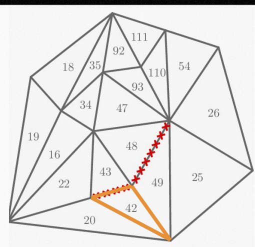
To avoid synchronization and racing conditions, the bisector with the smallest heapID in the four is in charge of checking and registering for the simplification.

In the example, that would be the bisector with the heapID 42

(Actually in practice, we don't enqueue the bisectors with odd heap IDs into the merge queue as an optimization)

Parallel Update Routine (Prepare Simplify)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG



CPU

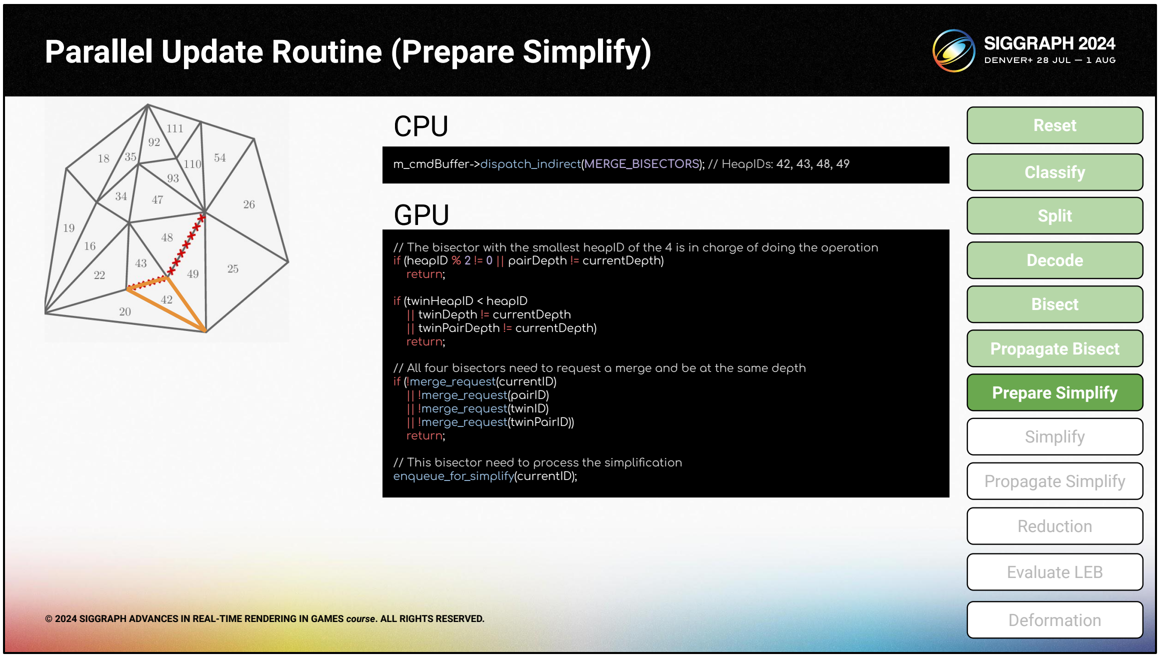
```
m_cmdBuffer->dispatch_indirect(MERGE_BISECTORS); // HeapIDs: 42, 43, 48, 49
```

GPU

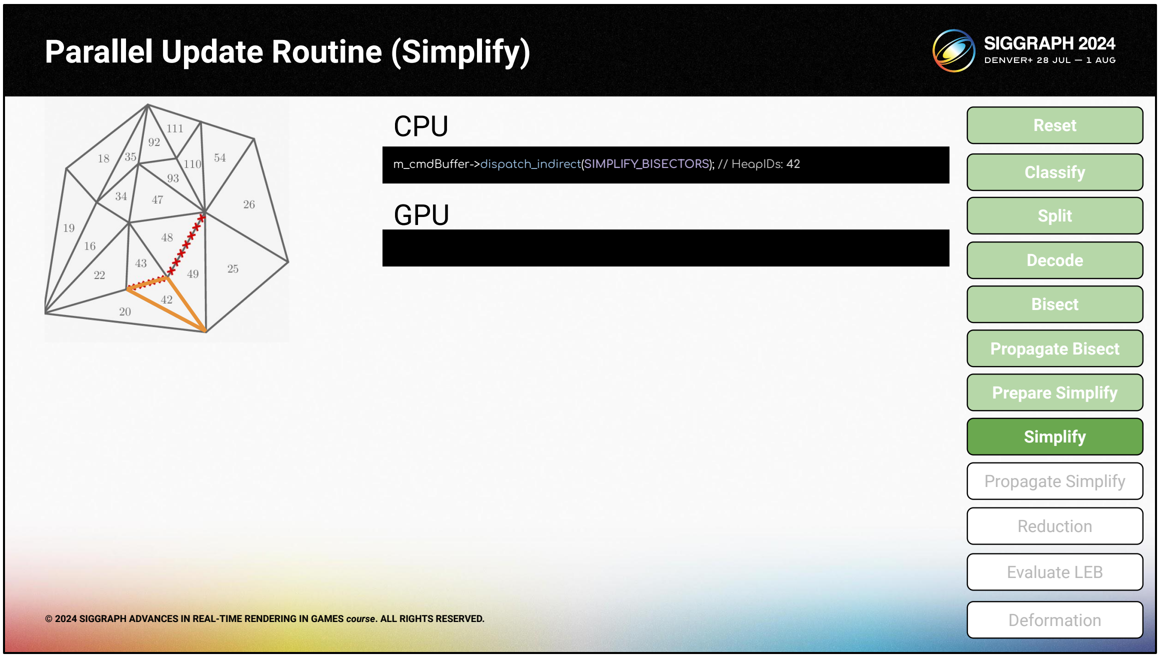
```
// The bisector with the smallest heapID of the 4 is in charge of doing the operation
if (heapID % 2 != 0 || pairDepth != currentDepth)
    return;

if (twinHeapID < heapID
    || twinDepth != currentDepth
    || twinPairDepth != currentDepth)
    return;
```

- Reset
- Classify
- Split
- Decode
- Bisect
- Propagate Bisect
- **Prepare Simplify**
- Simplify
- Propagate Simplify
- Reduction
- Evaluate LEB
- Deformation

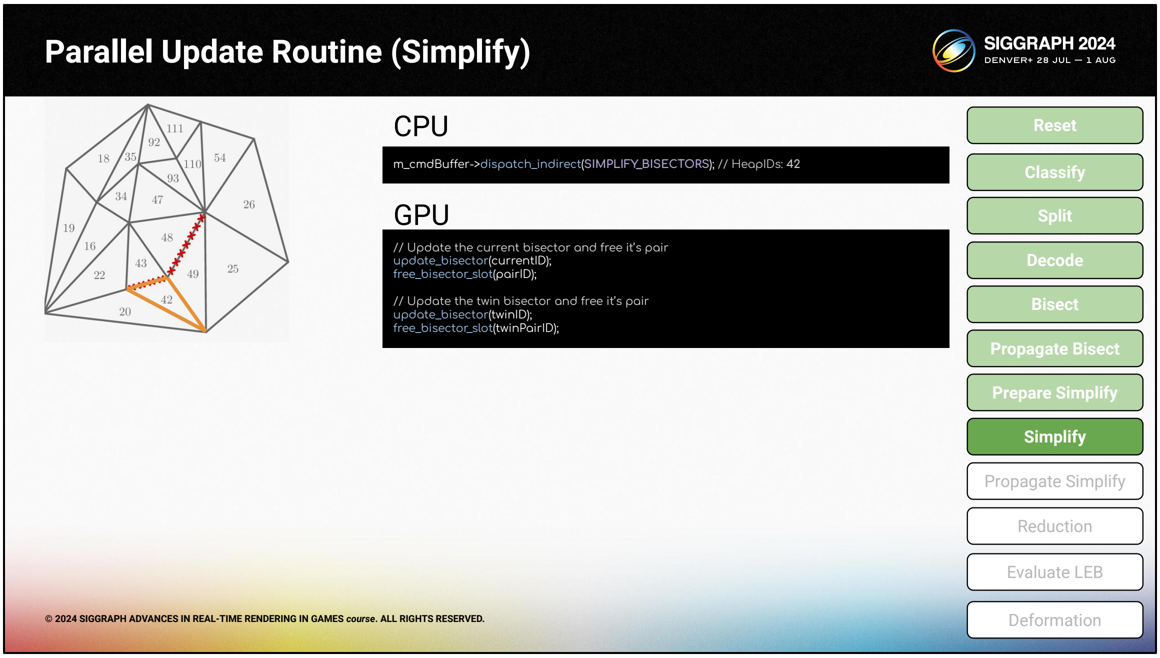
We can easily identify which one it is using the depth of the bisectors and their heapIDs



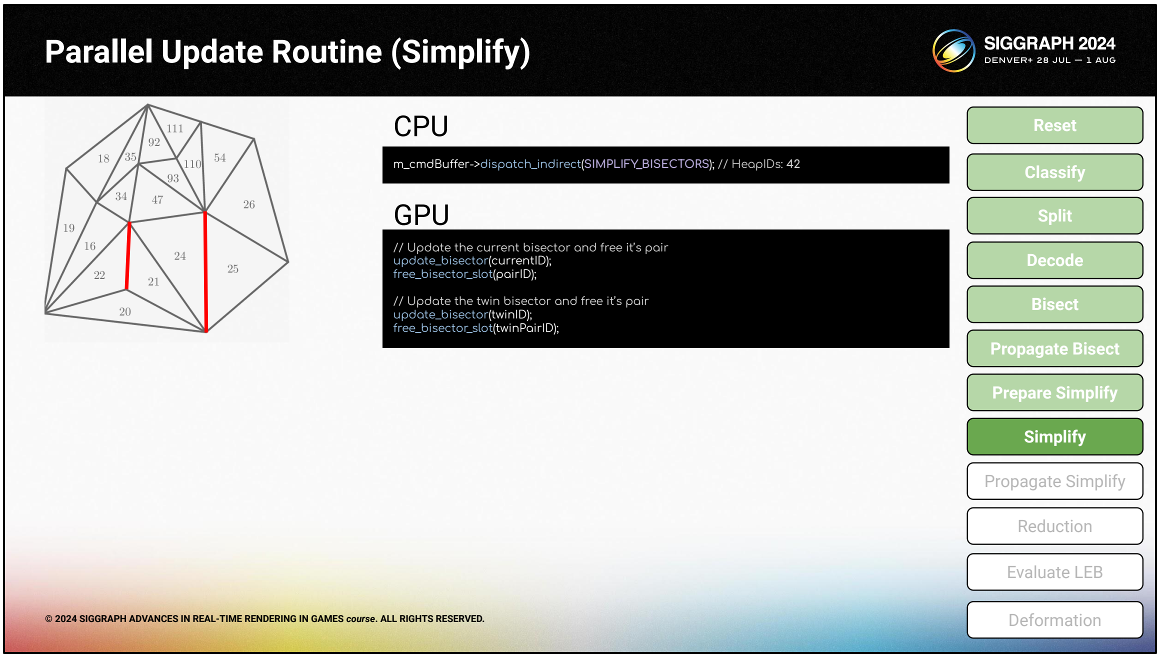
That bisector is then in charge of making sure they all required a merge operation and if it is the case, it will enqueue for the next step



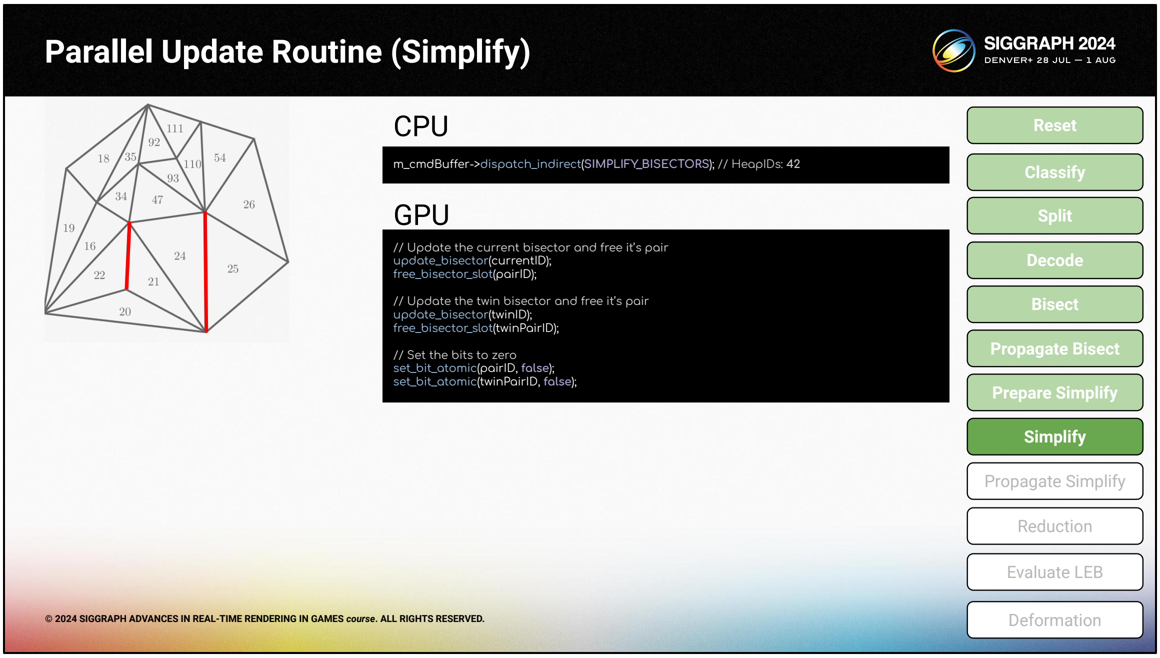
Which is the “Simplify” step, this will only run for the bisector 42 in this case



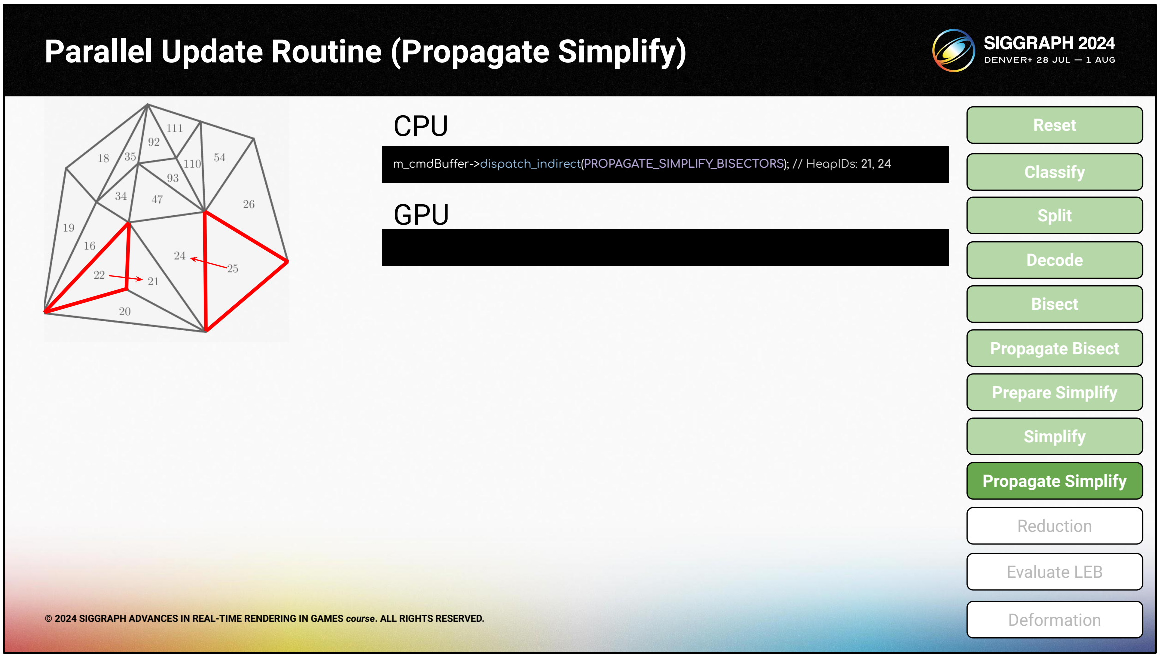
We update the bisectors that will remain (the ones with the smallest heapIDs for each pair, 42 and 48 in this case) and free the one other ones (43 and 49).



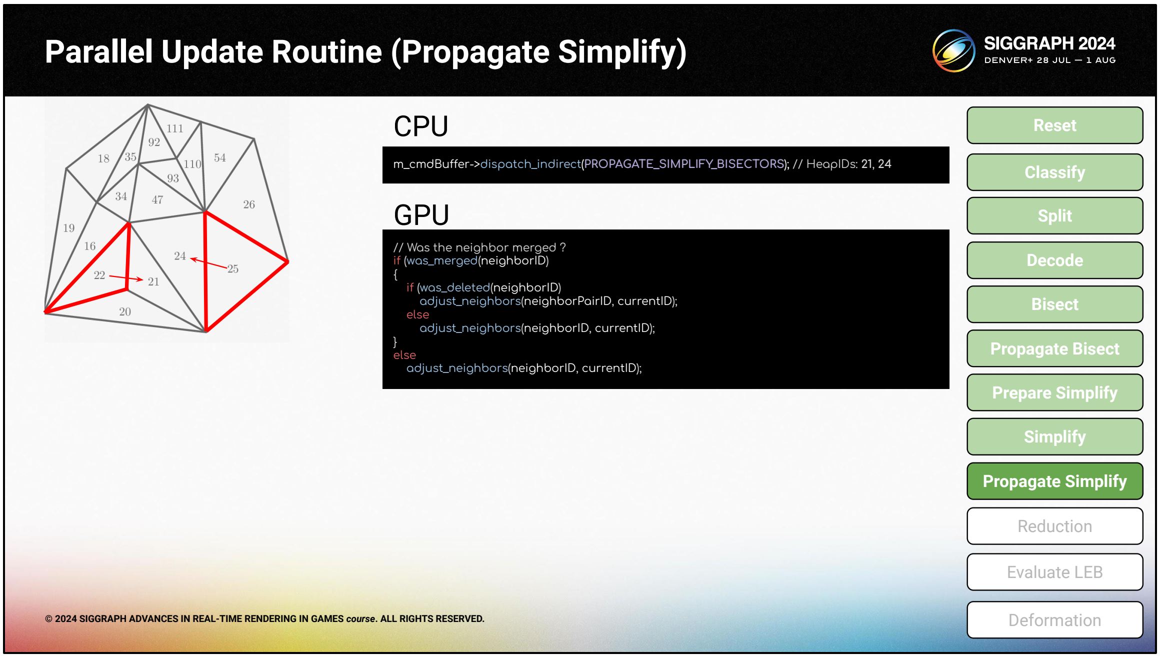
The same way than for split, we update the heapID and only partially the neighbors that will be updated in the following pass. This generates the new triangulation.



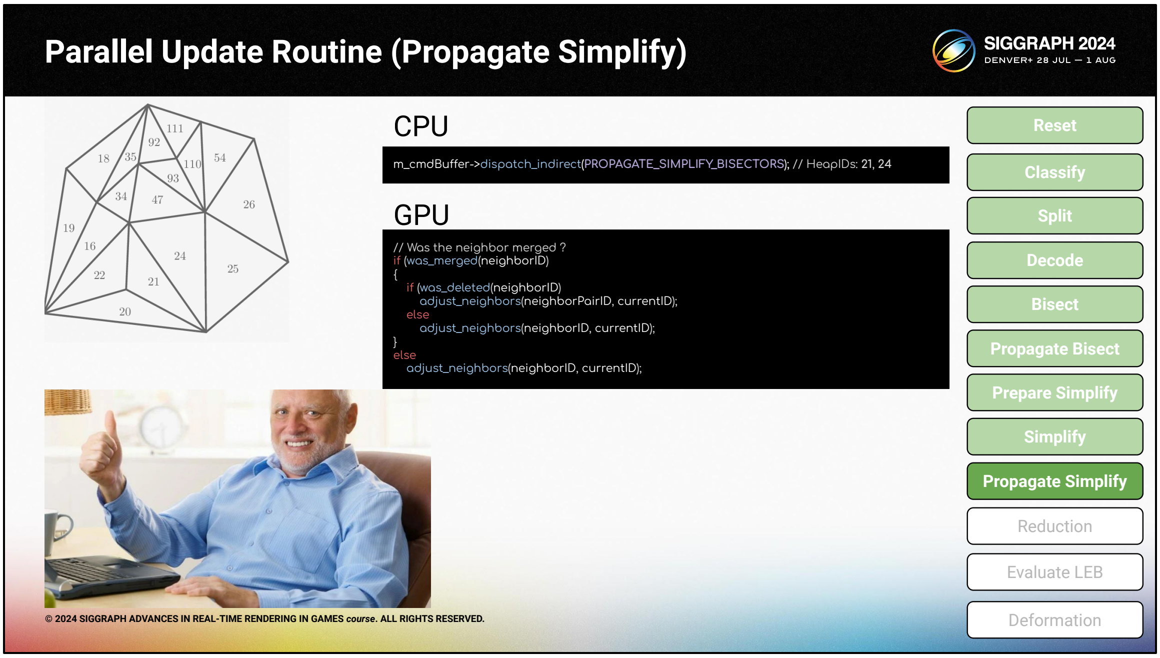
Then we set the bits in the CBT to zero for the unused slots



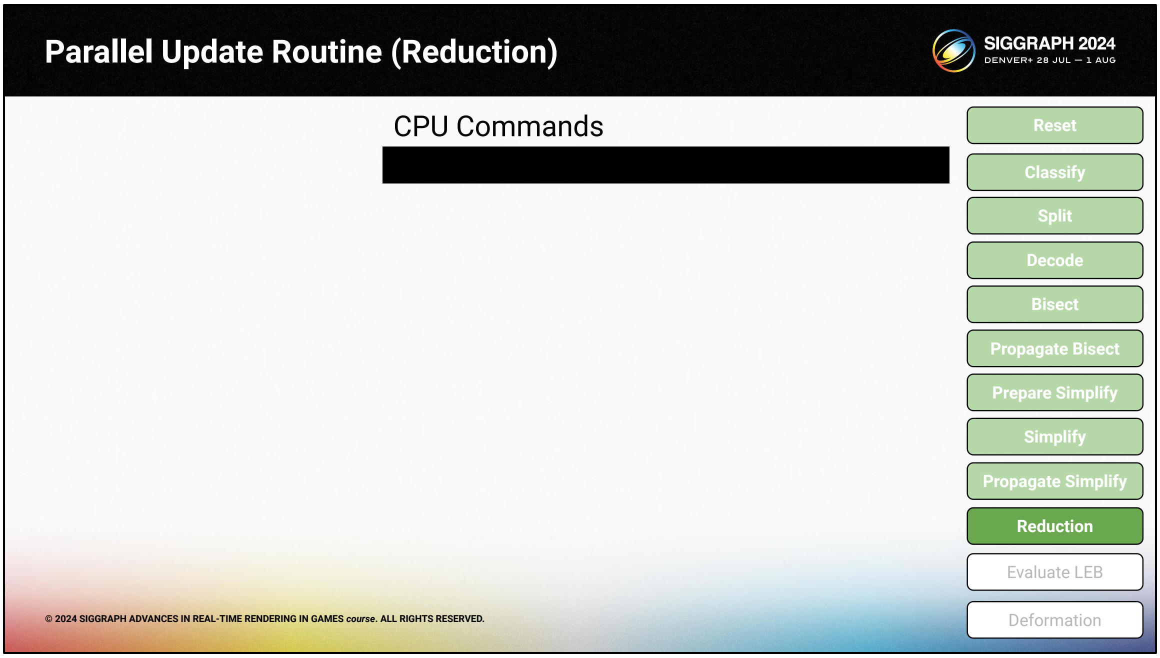
In this last step of the modification “Propagate simplify”, we dispatch on the bisectors that generated an inconsistency in the neighbors



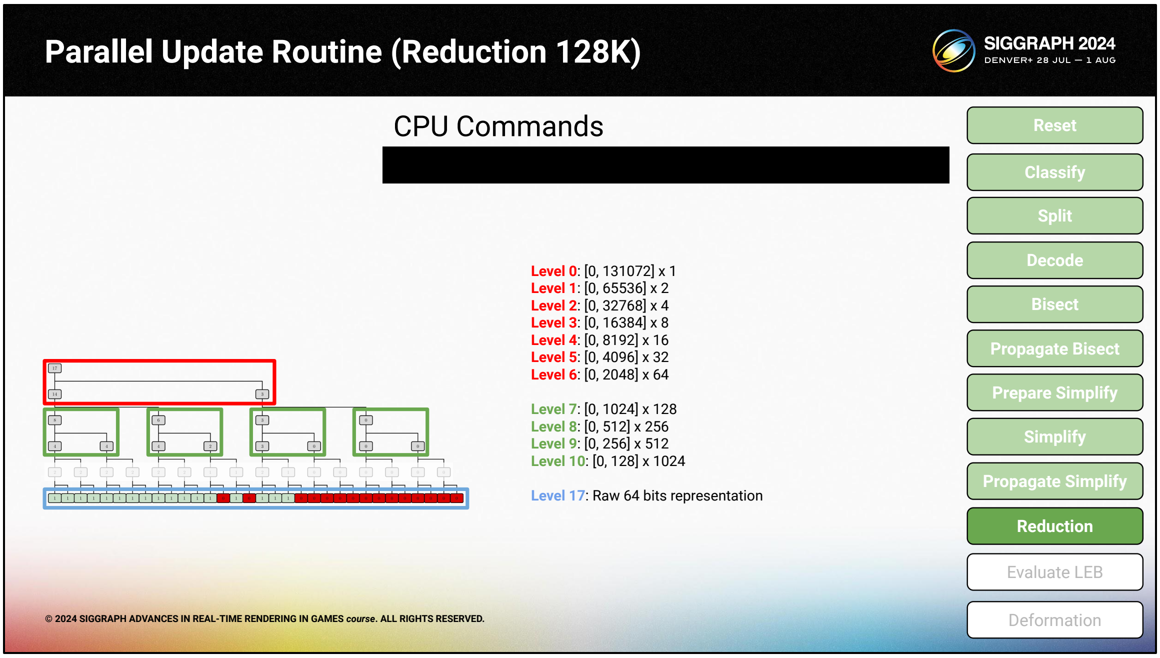
And depending on if the neighbors were merged we or not, the routine needs to take account if the bisector was deleted, etc



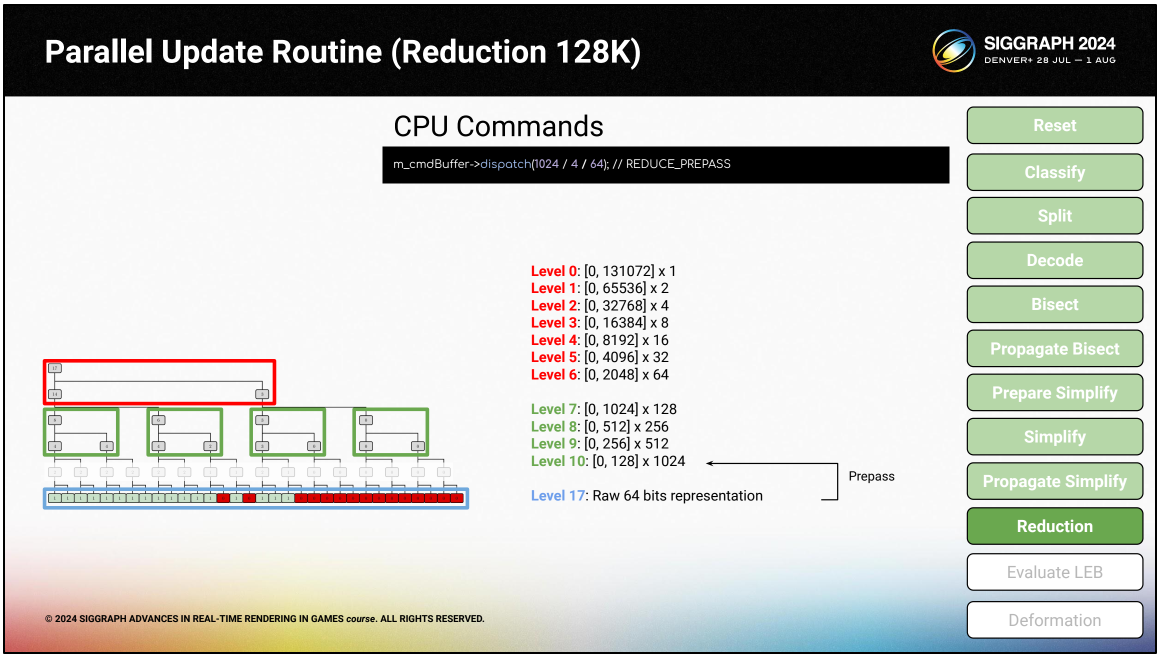
And with that we've processed our split and merge requests and have a coherent and LEB compatible triangulation



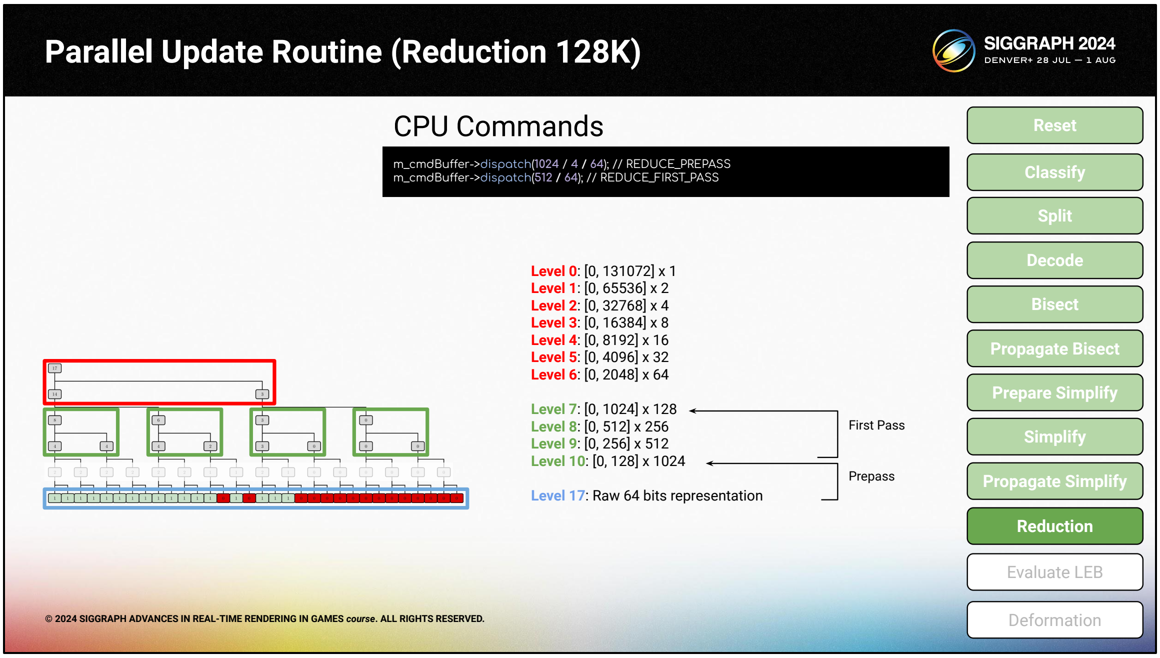
But we’re not done yet, there are couple steps left before we get to the end of our update routine.
We did modify the bit field in the previous steps, but the tree now doesn’t match anymore. The next one is operating a sum reduction on the CBT’s tree



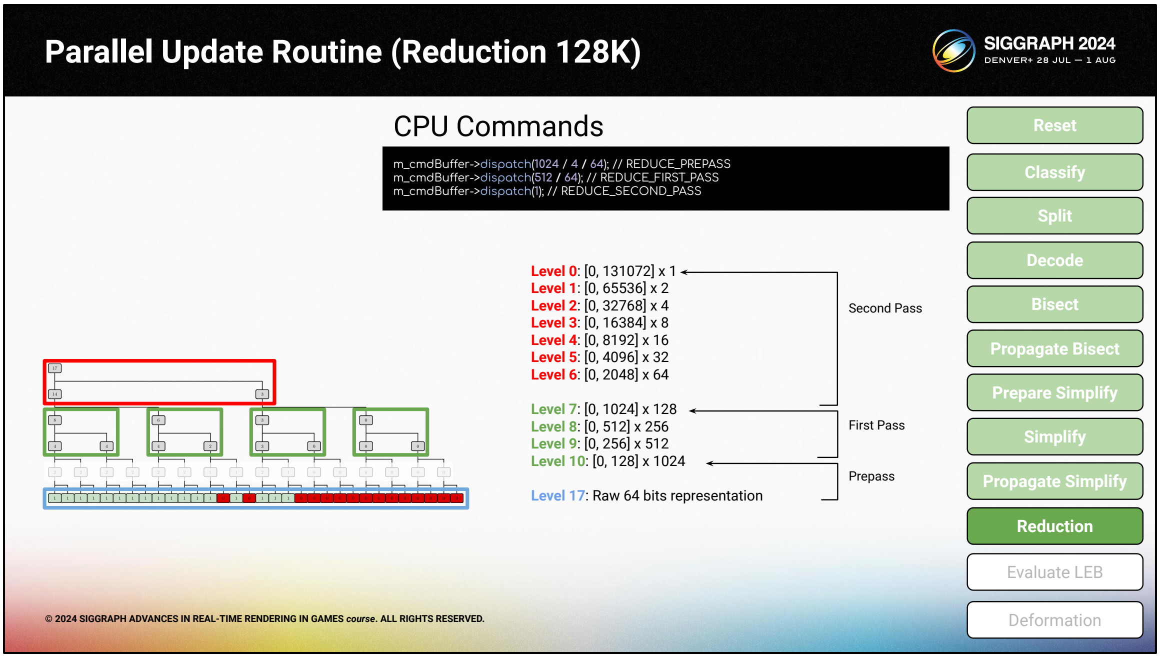
Let’s again take the example of the 128k CBT



First we do a reduction to move from our bitfield to the first explicitly stored level of the tree



Then another pass within the pet-thread atomic part of the tree



Finally one single workgroup will do the full sum reduction on the atomic friendly tree. With that we're able to do the have the full tree updated at a reasonable cost.

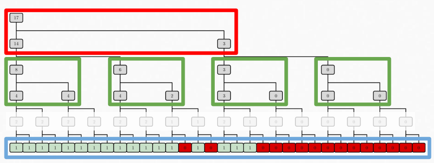
Parallel Update Routine (Reduction 128K)

SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

CPU Commands

```
m_cmdBuffer->dispatch(1024 / 4 / 64); // REDUCE_PREPASS
m_cmdBuffer->dispatch(512 / 64); // REDUCE_FIRST_PASS
m_cmdBuffer->dispatch(1); // REDUCE_SECOND_PASS
```

CBT Size	Intel Arc 770 (ms)	AMD 6650 XT (ms)	Nvidia RTX 4090 (ms)
128k	0.055	0.01	0.008
256k	0.055	0.012	0.01
512k	0.062	0.013	0.01
1m	0.065	0.017	0.011



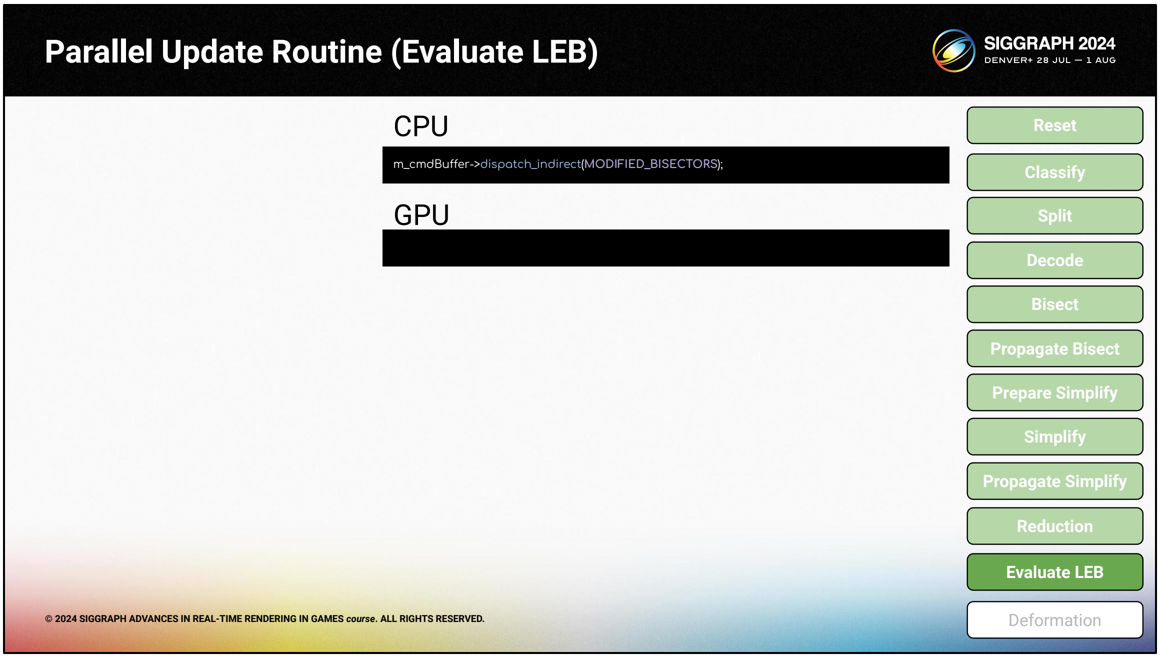
Level 0: [0, 131072] x 1
Level 1: [0, 65536] x 2
Level 2: [0, 32768] x 4
Level 3: [0, 16384] x 8
Level 4: [0, 8192] x 16
Level 5: [0, 4096] x 32
Level 6: [0, 2048] x 64
Second Pass

Level 7: [0, 1024] x 128
Level 8: [0, 512] x 256
Level 9: [0, 256] x 512
Level 10: [0, 128] x 1024
First Pass

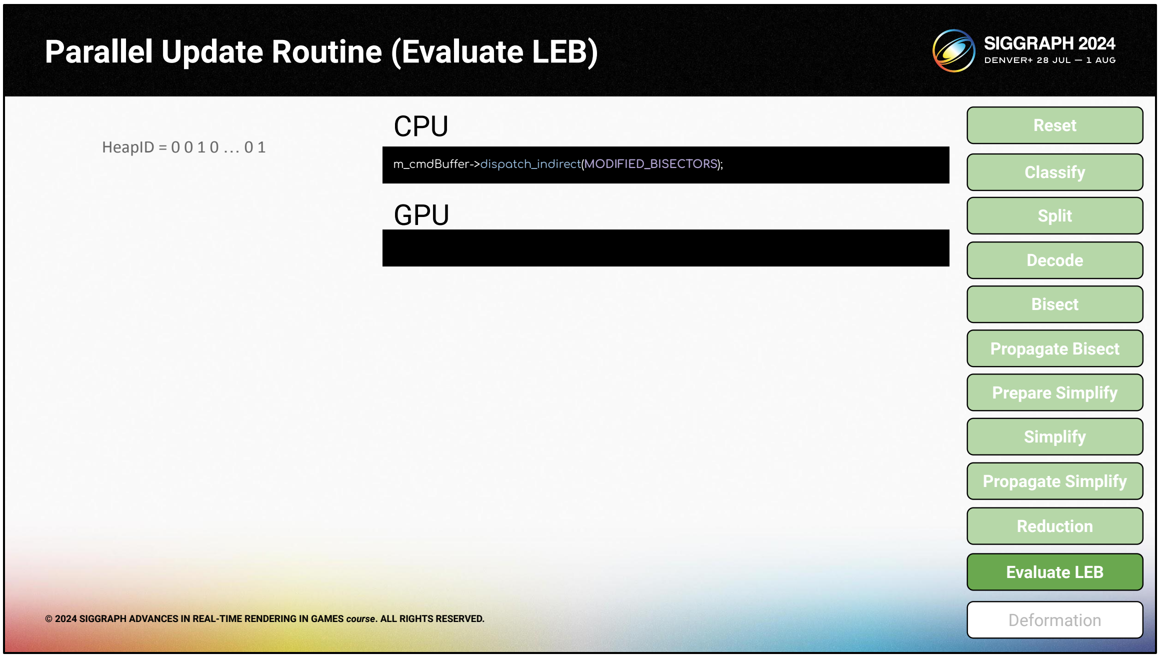
Level 17: Raw 64 bits representation
Prepass

Reset
Classify
Split
Decode
Bisect
Propagate Bisect
Prepare Simplify
Simplify
Propagate Simplify
Reduction
Evaluate LEB
Deformation

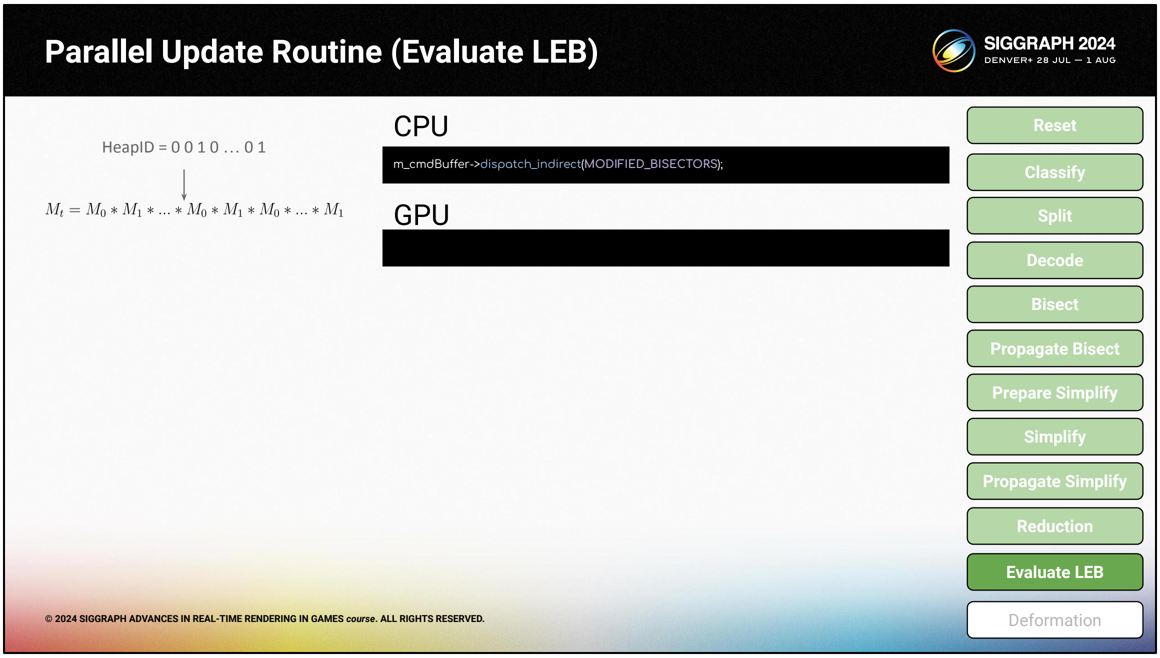
To illustrate what i mean by reasonable, here you can see a table that recaps the reduction time for each CBT size that we provide in our demo and for 3 GPUs (Intel Arc 770, AMD 6650 XT and Nvidia 4090)



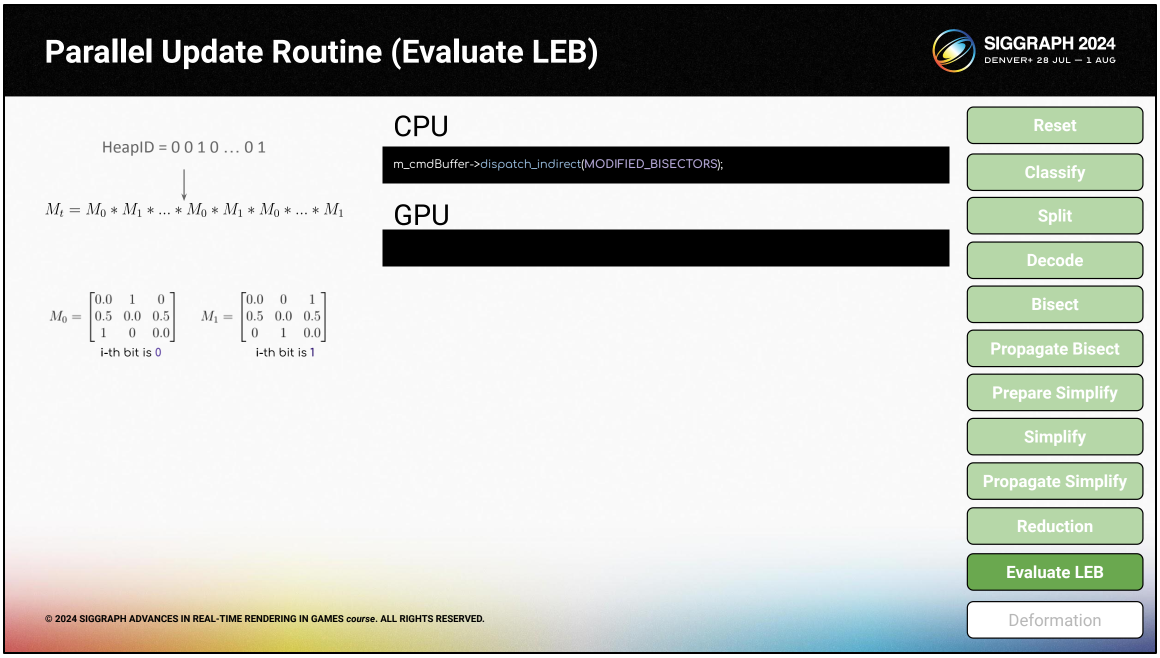
The next step is processing the modified bisectors to ensure that the positions are up to date



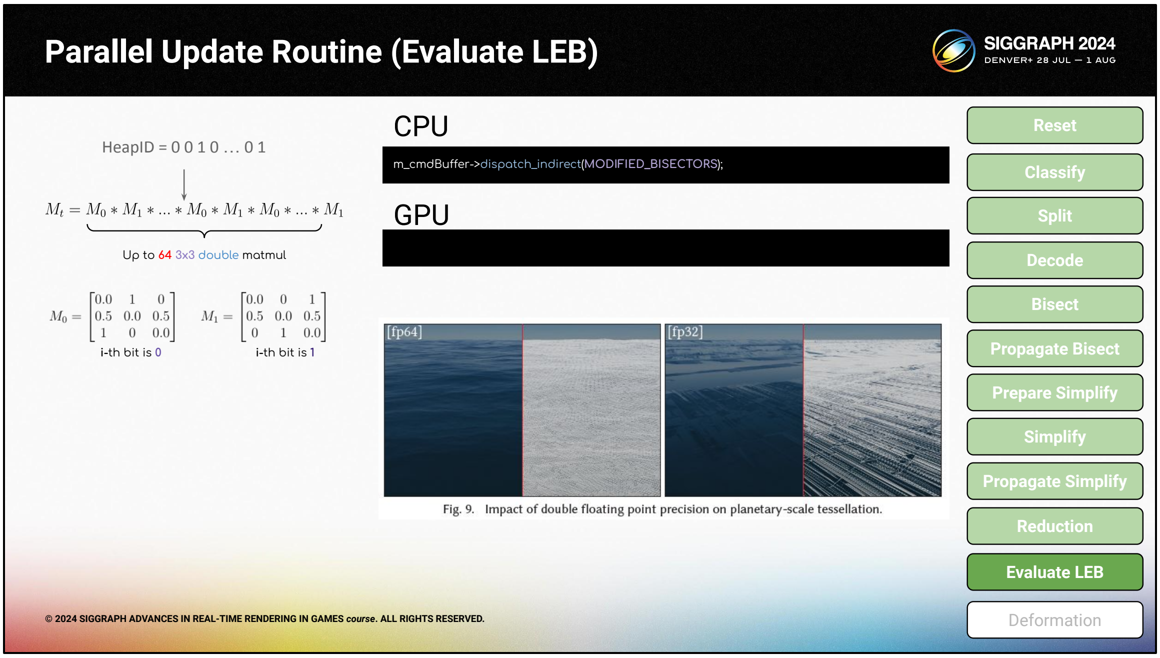
As we mentioned before, the geometry information is encoded using the heapID and if we break it down to a binary representation, it would look something like this



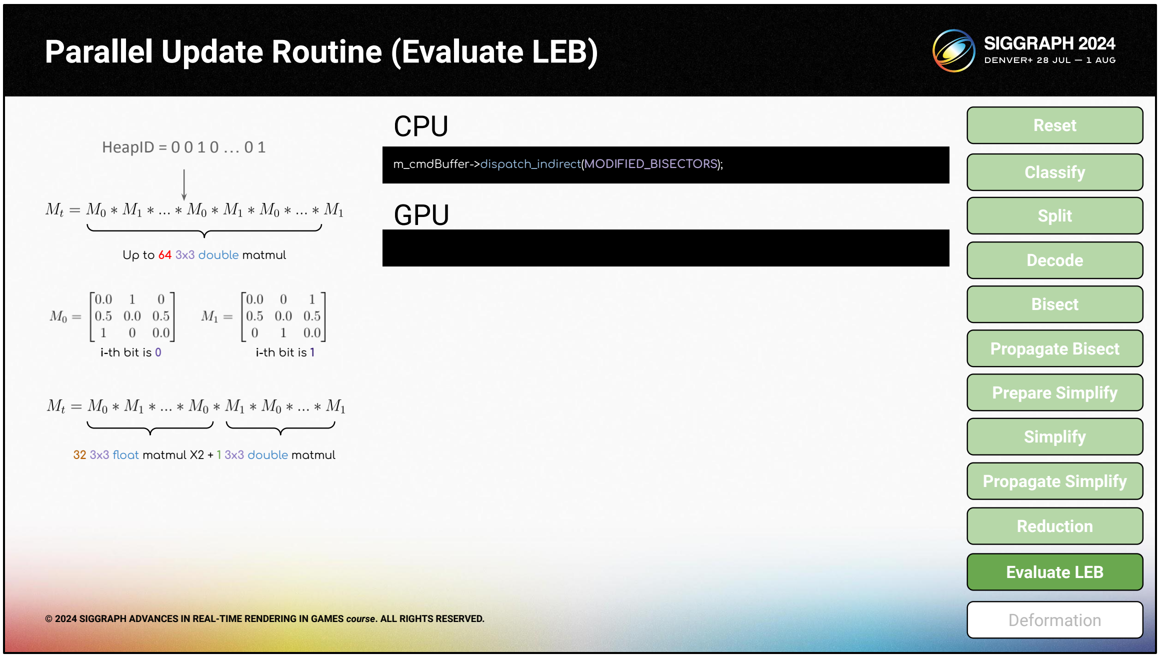
The final position is obtained using a chain of matrix multiplications, each bit of the heapID will correspond to one of two matrices



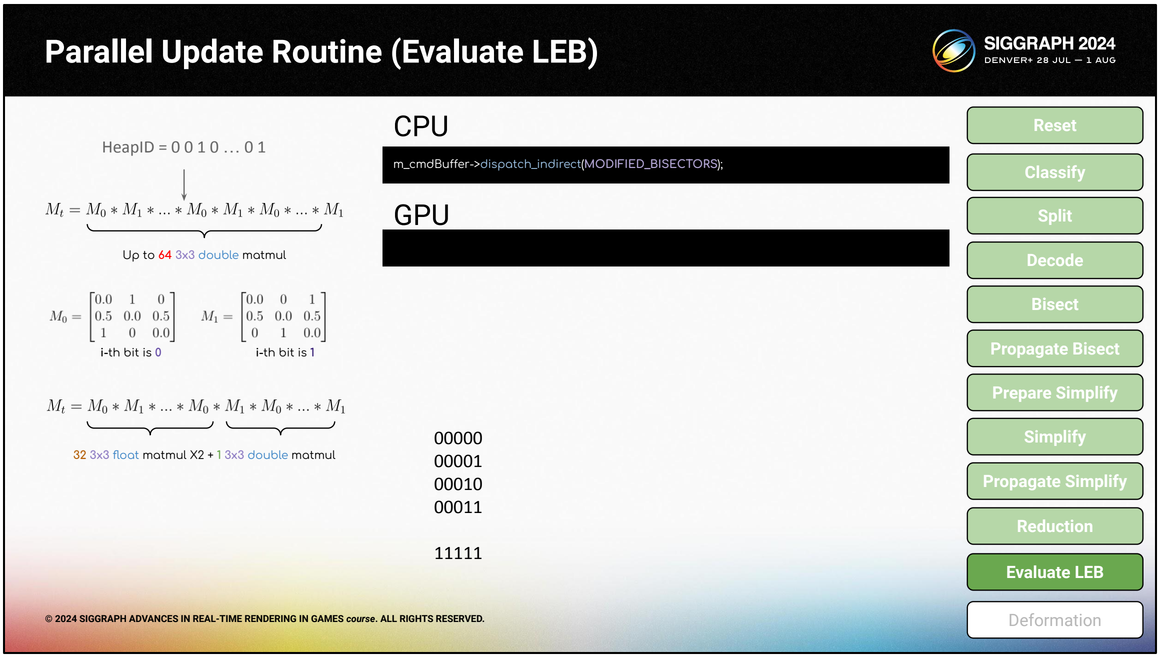
These are the two matrices we use for the demo



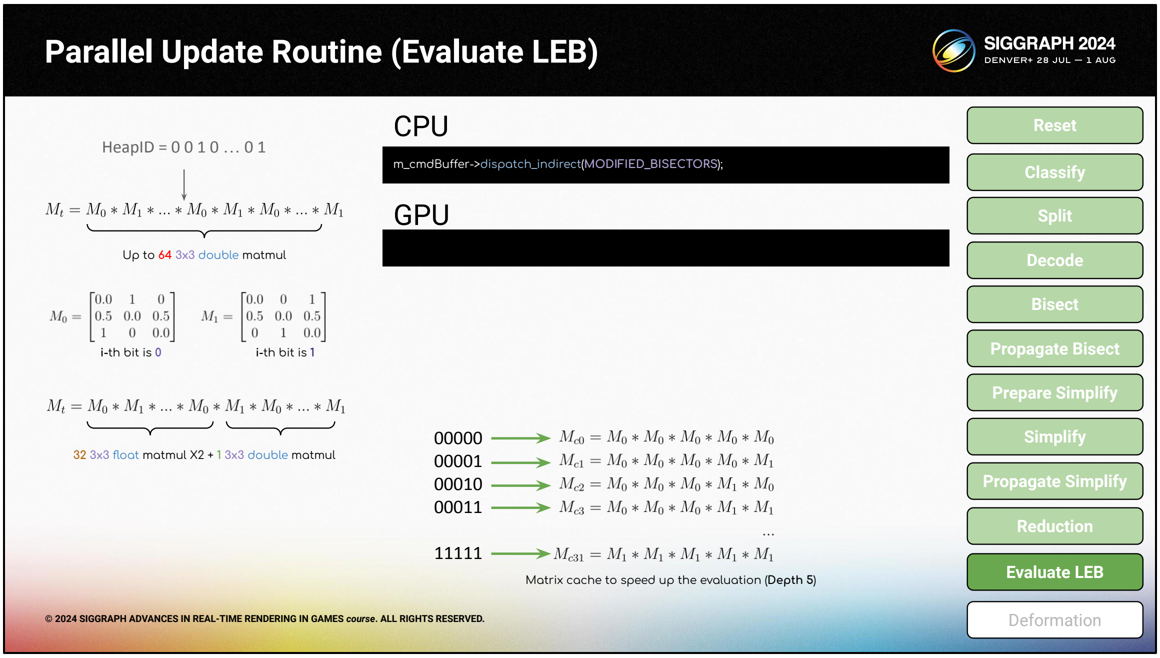
There are two issues here:
- First, even if the output position are stored using simple precision floating point and camera relative, we have to do the multiplications in double space (otherwise we break the floating points due to precisions issues due to to our method)
- Depending on the subdivision level, we can have up to 64 3x3 matrix multiplications to evaluate which is a lot



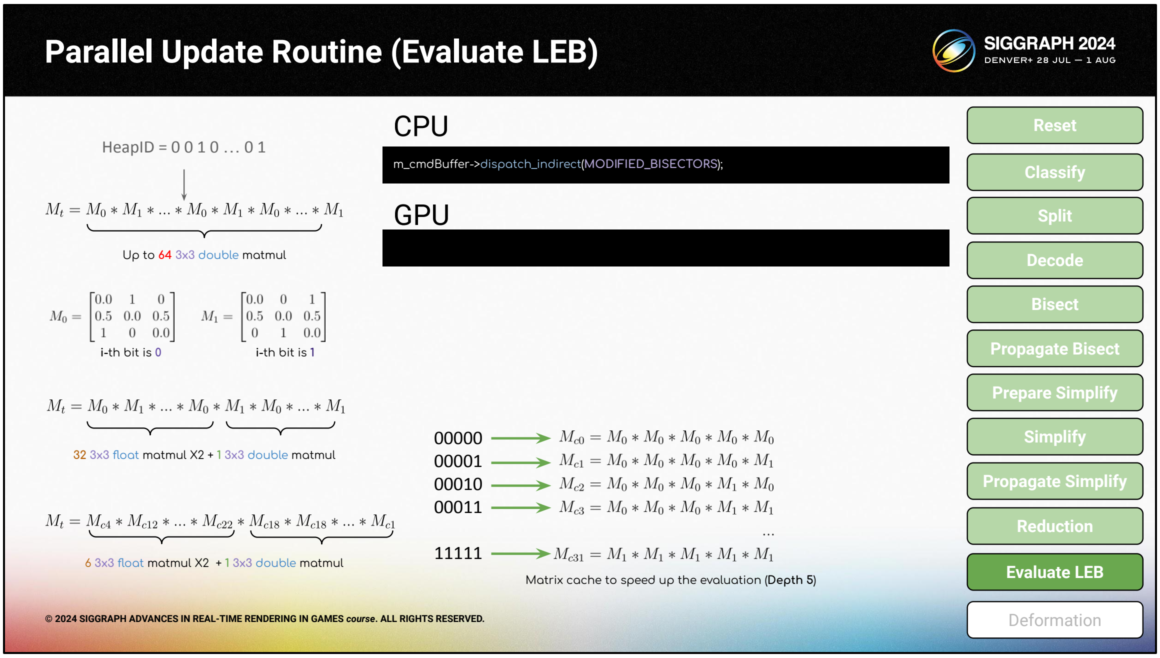
The first thing we can do is split this multiplication into two 32 3x3 float multiplications with a 3x3 double matmul at the middle which reduces the ALU consumption and better uses the FLOPS of the card, but is still not good enough



If we decompose the heap ID into sequences of bits of equal size

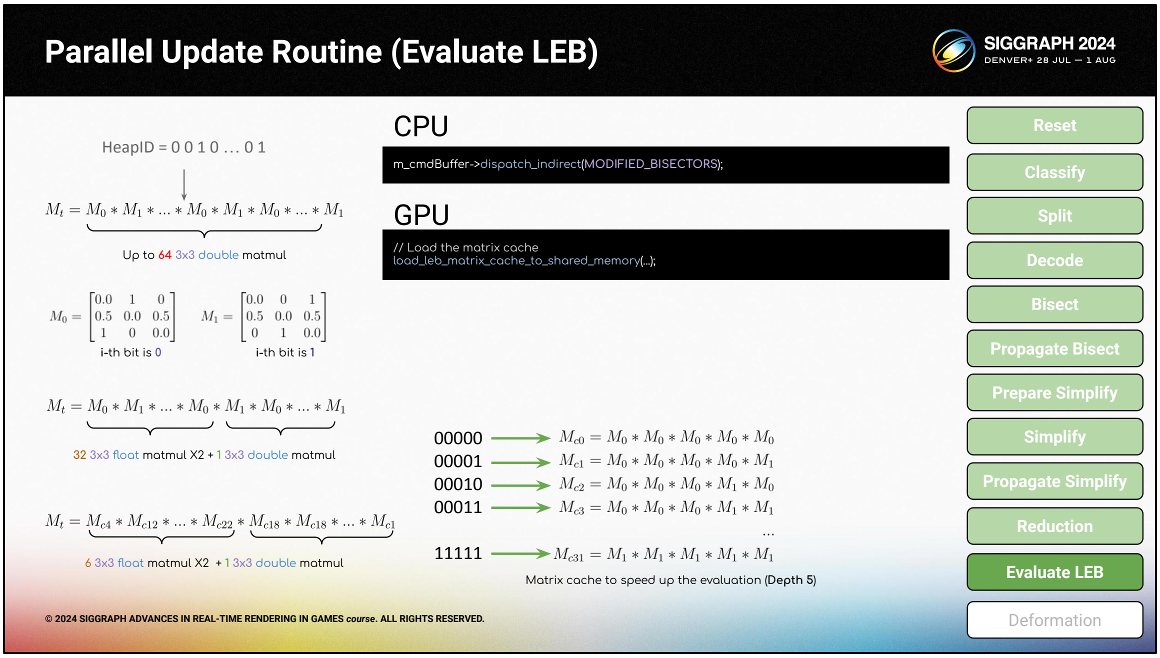


The way we can accelerate this even more is by building what we call a matrix cache, it is a premultiplication of all combinations over a certain number of levels (in practice, we've measure that a depth of 5 is the best speedup/size compromise because we need to load this into shared memory as we'll be accessing it quite a bit).

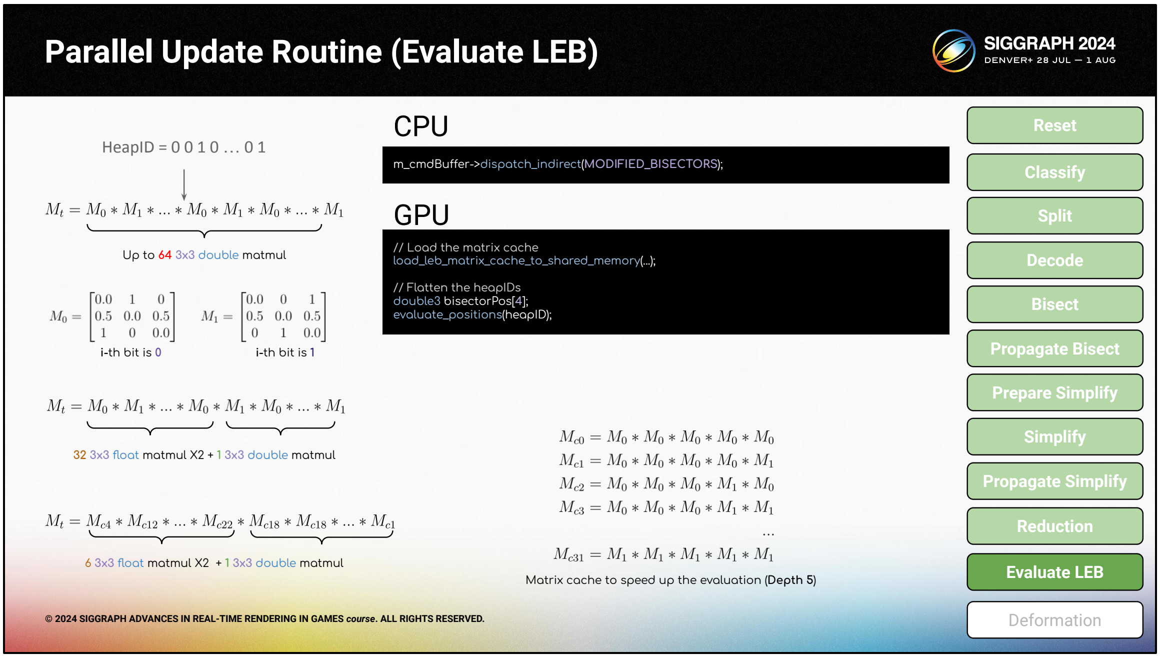


By doing this, we reduce the number of matrices to up to two sets of 6 3x3 mat muls and one double 3x3 matmul which allows us to get very reasonable execution times as you'll see in a second.

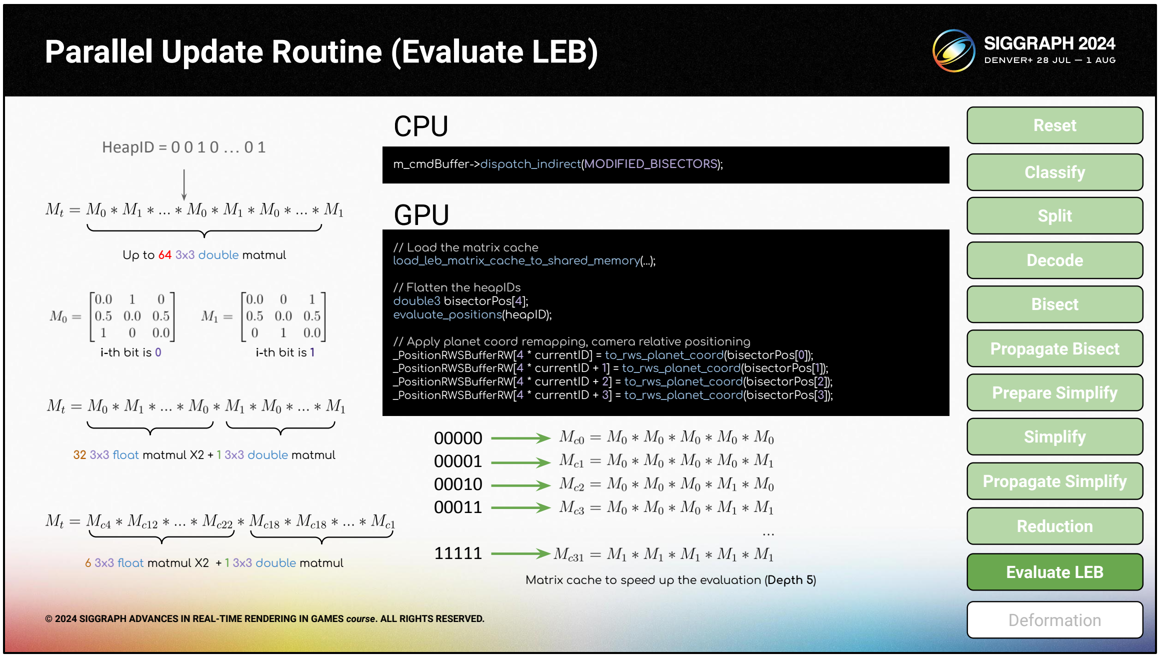
This shader executes only for modified bisectors which is a fraction of the total bisectors



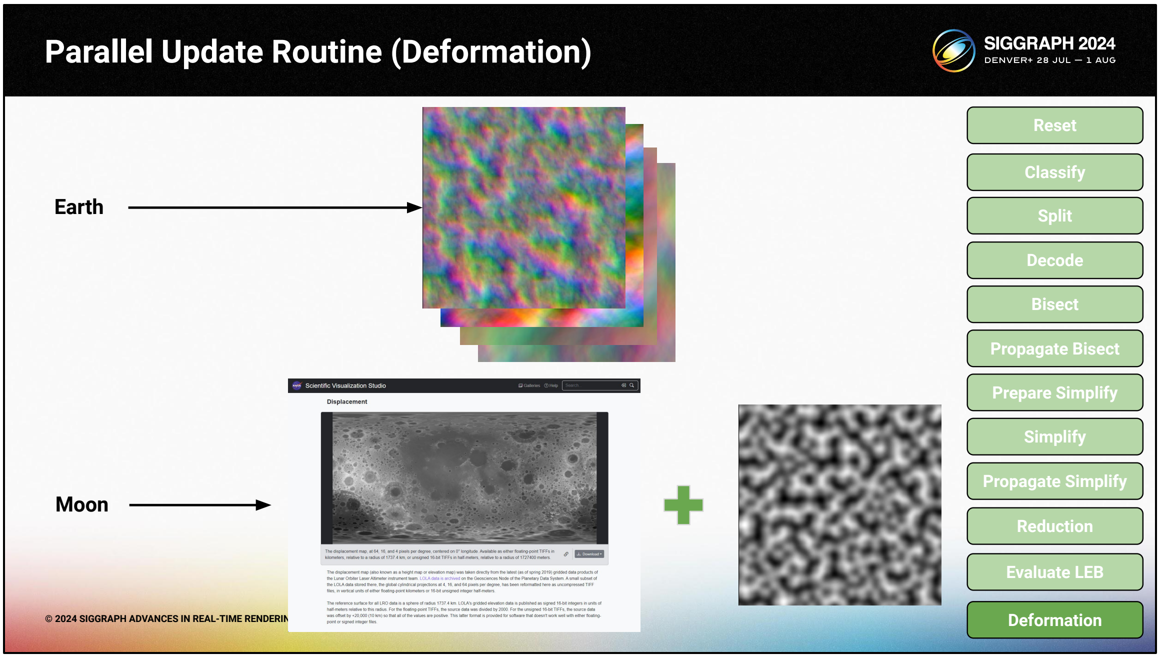
In practice, we would load the matrix cache into shared memory



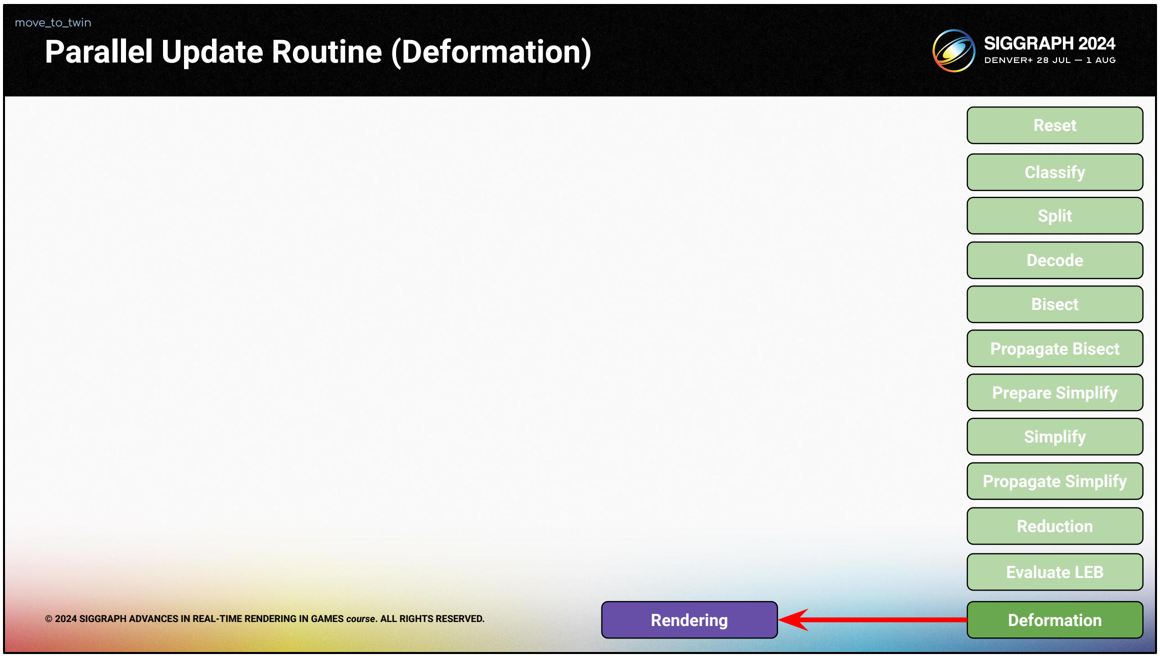
We would flatten the heapID into 4 positions (3 for the current bisector and 1 for the parent)



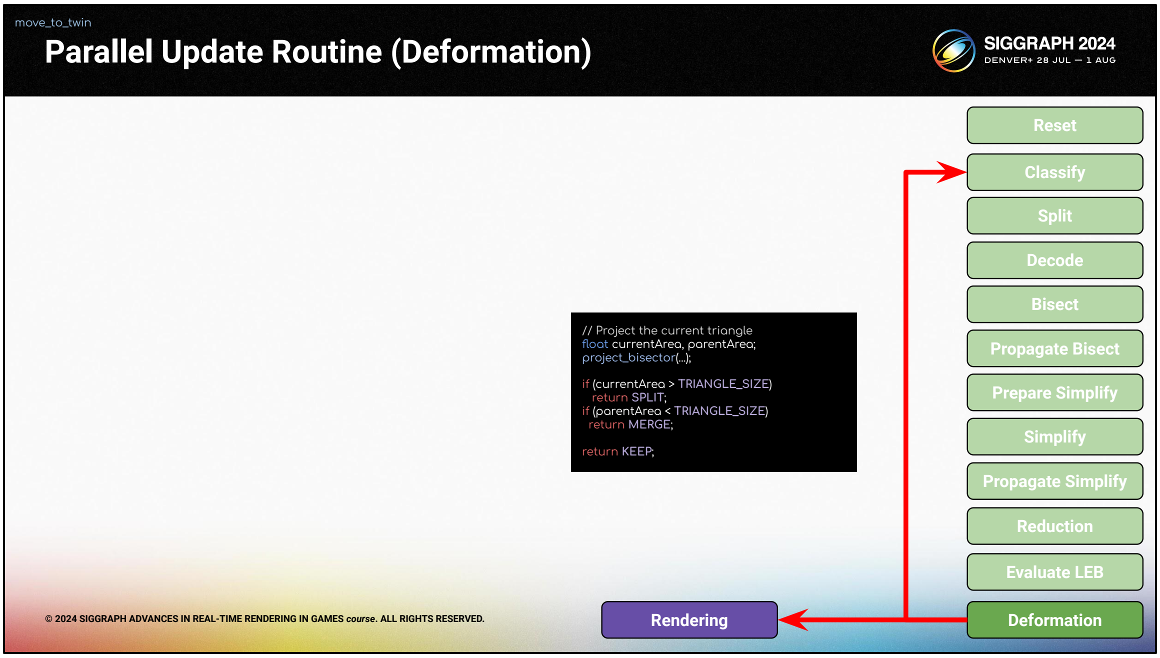
And then we would apply a transformation to map this to a spherical planet and convert it to relative world space



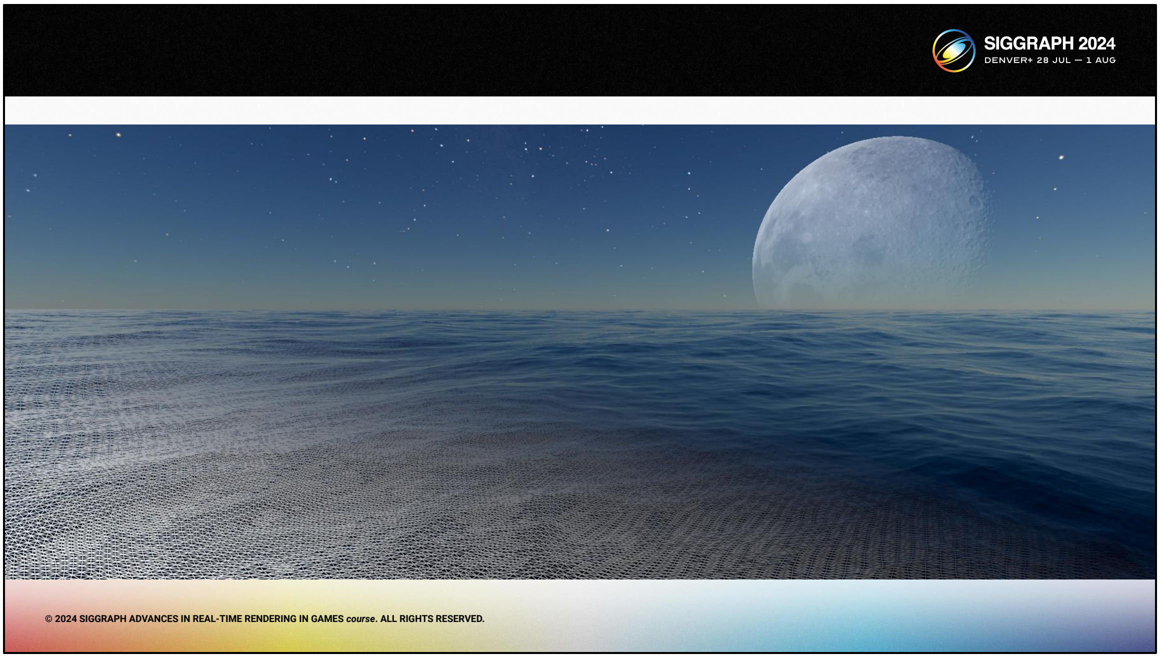
Finally to all existing positions, we would apply the relevant deformation, for the earth it would be the result of a bunch of FFTs and the moon we use the displacement maps that the NASA provides on it's website, plus a bunch of noise functions for high frequency details



These produced positions would then be used for the rendering rasterized or ray traced



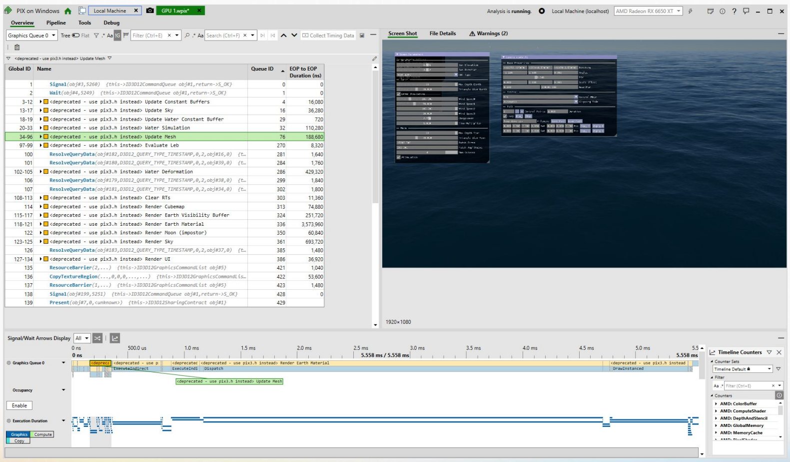
But also for the following frame to operate the classification



And with that, we are ready to render our mesh to screen

Performance

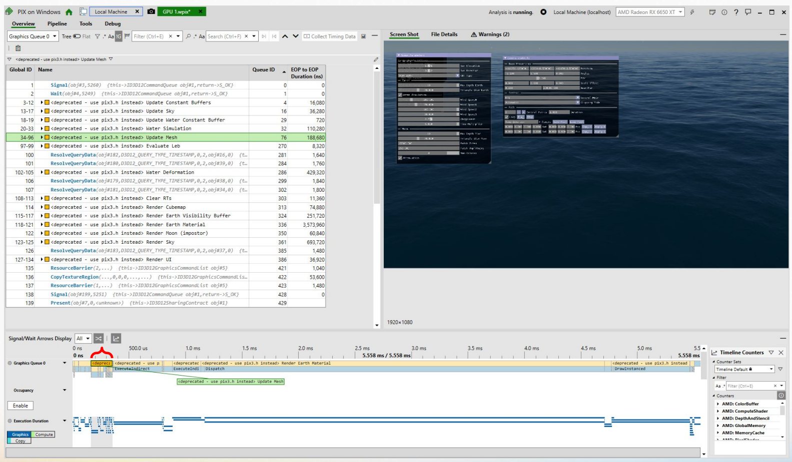
SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG



The last thing we would like to cover is the performance numbers of the method, this is a screenshot of a capture of the demo running at the surface of the earth

Performance

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG



In our case, we're interested by this section, highlighted in red

Performance

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

Scene	Earth Close Up	Earth Close Up	Earth Close Up	Moon Close Up	Moon Close Up	Moon Close Up
GPU	Intel Arc 770	AMD 6650 XT	Nvidia RTX 4090	Intel Arc 770	AMD 6650 XT	Nvidia RTX 4090
Update Proportion	High	High	High	Low	Low	Low
Active Primitives	~64k	~64k	~64k	~95k	~95k	~95k
Reset (ms)	0.014	0.003	0.004	0.013	0.003	0.005
Classify (ms)	0.078	0.058	0.012	0.07	0.072	0.013
Split (ms)	0.042	0.019	0.014	0.023	0.017	0.012
Allocate (ms)	0.024	0.026	0.013	0.031	0.021	0.011
Copy (ms)	0.011	0.017	0.006	0.012	0.017	0.006
Bisect (ms)	0.016	0.005	0.007	0.012	0.006	0.005
Propagate Bisect (ms)	0.016	0.007	0.009	0.02	0.01	0.01
Prepare Simplify (ms)	0.02	0.008	0.007	0.025	0.008	0.007
Simplify (ms)	0.02	0.009	0.011	0.012	0.006	0.012
Propagate Simplify (ms)	0.017	0.006	0.011	0.017	0.006	0.01
Reduce (ms)	0.058	0.01	0.008	0.058	0.01	0.008
Indexation (ms)	0.03	0.014	0.013	0.03	0.014	0.013
Evaluate LEB (ms)	0.025	0.008	0.01	0.019	0.008	0.01
Update Pass (ms)	**0.371**	**0.19**	**0.125**	**0.342**	**0.198**	**0.122**

Rendered at 1920x1080 - CBT 128K

This table recaps the timings for each of the passes that we've explained. We did the profiling on three GPUs:

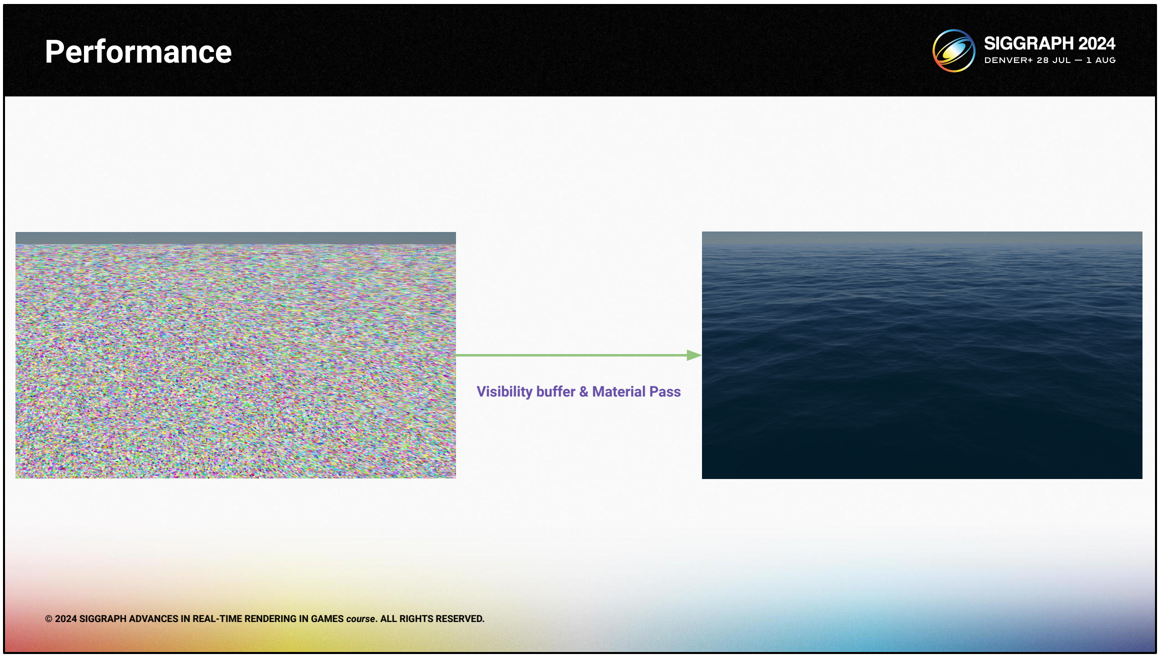
- The Arc 770
- the AMD 6650 XT which is roughly equivalent to a PS5
- A higher end GPU, the Nvidia 4090

Performance

Scene	Earth Close Up	Earth Close Up	Earth Close Up	Moon Close Up	Moon Close Up	Moon Close Up
GPU	Intel Arc 770	AMD 6650 XT	Nvidia RTX 4090	Intel Arc 770	AMD 6650 XT	Nvidia RTX 4090
Update Proportion	High	High	High	Low	Low	Low
Active Primitives	~64k	~64k	~64k	~95k	~95k	~95k
Reset (ms)	0.014	0.003	0.004	0.013	0.003	0.005
Classify (ms)	0.078	0.058	0.012	0.07	0.072	0.013
Split (ms)	0.042	0.019	0.014	0.023	0.017	0.012
Allocate (ms)	0.024	0.026	0.013	0.031	0.021	0.011
Copy (ms)	0.011	0.017	0.006	0.012	0.017	0.006
Bisect (ms)	0.016	0.005	0.007	0.012	0.006	0.005
Propagate Bisect (ms)	0.016	0.007	0.009	0.02	0.01	0.01
Prepare Simplify (ms)	0.02	0.008	0.007	0.025	0.008	0.007
Simplify (ms)	0.02	0.009	0.011	0.012	0.006	0.012
Propagate Simplify (ms)	0.017	0.006	0.011	0.017	0.006	0.01
Reduce (ms)	0.058	0.01	0.008	0.058	0.01	0.008
Indexation (ms)	0.03	0.014	0.013	0.03	0.014	0.013
Evaluate LEB (ms)	0.025	0.008	0.01	0.019	0.008	0.01
Update Pass (ms)	**0.371**	**0.19**	**0.125**	**0.342**	**0.198**	**0.122**

Rendered at 1920x1080 - CBT 128K

The important bit there is that we’re able to stay at reasonable cost at all time and less than 0.2ms on a hardware equivalent to a PS5 which is more than reasonable given the achievements of the method



The rest of the pipeline is dependent on the actual deformation that we'd use and the general structure of the rendering pipeline.
For the demo we're using a visibility buffer followed by a material pass.

Performance

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

Scene	Earth Close Up	Moon Close Up
GPU	AMD 6650 XT	AMD 6650 XT
Update Proportion	High	Low
Active Primitives	~64k	~95k
Reset (ms)	0.003	0.003
Classify (ms)	0.058	0.072
Split (ms)	0.019	0.017
Allocate (ms)	0.026	0.021
Copy (ms)	0.017	0.017
Bisect (ms)	0.005	0.006
Propagate Bisect (ms)	0.007	0.01
Prepare Simplify (ms)	0.008	0.008
Simplify (ms)	0.009	0.006
Propagate Simplify (ms)	0.006	0.006
Reduce (ms)	0.01	0.01
Indexation (ms)	0.014	0.014
Evaluate LEB (ms)	0.008	0.008
Update Pass (ms)	**0.19**	**0.198**
Deformation (ms)	0.43	1
Render Vis Buffer (ms)	0.25	0.27
Render Material (ms)	3.5	3.2
Frame Total	4.18	4.47

These are numbers we got for the AMD 6650 XT, but these are really dependent on how you generate your surface data really.

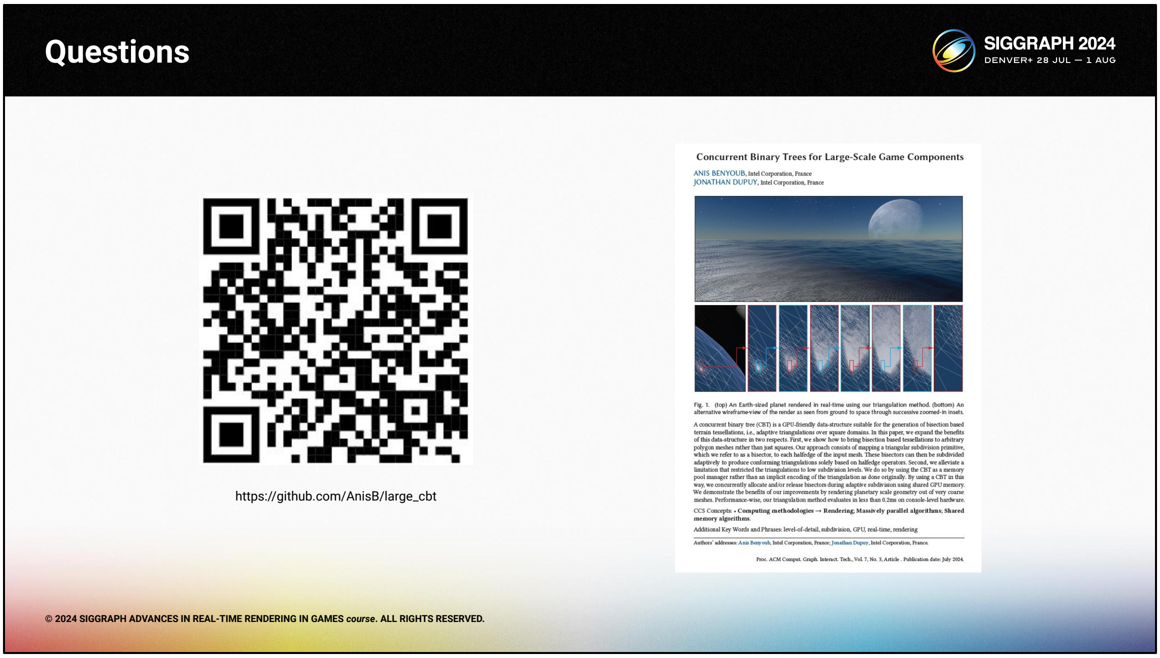
Performance

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

Scene	Earth Close Up	Moon Close Up
GPU	AMD 6650 XT	AMD 6650 XT
Update Proportion	High	Low
Active Primitives	~64k	~95k
Reset (ms)	0.003	0.003
Classify (ms)	0.058	0.072
Split (ms)	0.019	0.017
Allocate (ms)	0.026	0.021
Copy (ms)	0.017	0.017
Bisect (ms)	0.005	0.006
Propagate Bisect (ms)	0.007	0.01
Prepare Simplify (ms)	0.008	0.008
Simplify (ms)	0.009	0.006
Propagate Simplify (ms)	0.006	0.006
Reduce (ms)	0.01	0.01
Indexation (ms)	0.014	0.014
Evaluate LEB (ms)	0.008	0.008
Update Pass (ms)	**0.19**	**0.198**
Deformation (ms)	0.43	1
Render Vis Buffer (ms)	0.25	0.27
Render Material (ms)	3.5	3.2
Frame Total	4.18	4.47

Build RTAS (ms)	3.5	3.6
Trace RTAS (ms)	3.6	2.9

In addition to that i'd like to share some number ray tracing related. The method is compatible with real-time hardware ray tracing and given that the number of primitives remains quite "low", we're able to get something decent even by doing a full rebuild every frame of the tlas and blas



And that's it. The full code of the demo that we've show is open-source and can be found in the github that is linked here.
Also, don't hesitate to check the paper for more detailed explanations about the method, in the meantime, we are available for questions