Experimenting with Concurrent Binary Trees: Large Scale Terrain Rendering

SIGGRAPH 2021

Thomas Deliot, Xiaoling Yao, Jonathan Dupuy, Kees Rijnen
Unity Technologies

(Xiaoling)
Hi I'm Xiaoling Yao. I'm a graphics engineer in the Unity terrain team.

My team is currently developing a next generation terrain renderer, and the purpose of this talk is to share our progress and some insights using the concurrent binary tree for rendering large scale terrain.

Disclaimer

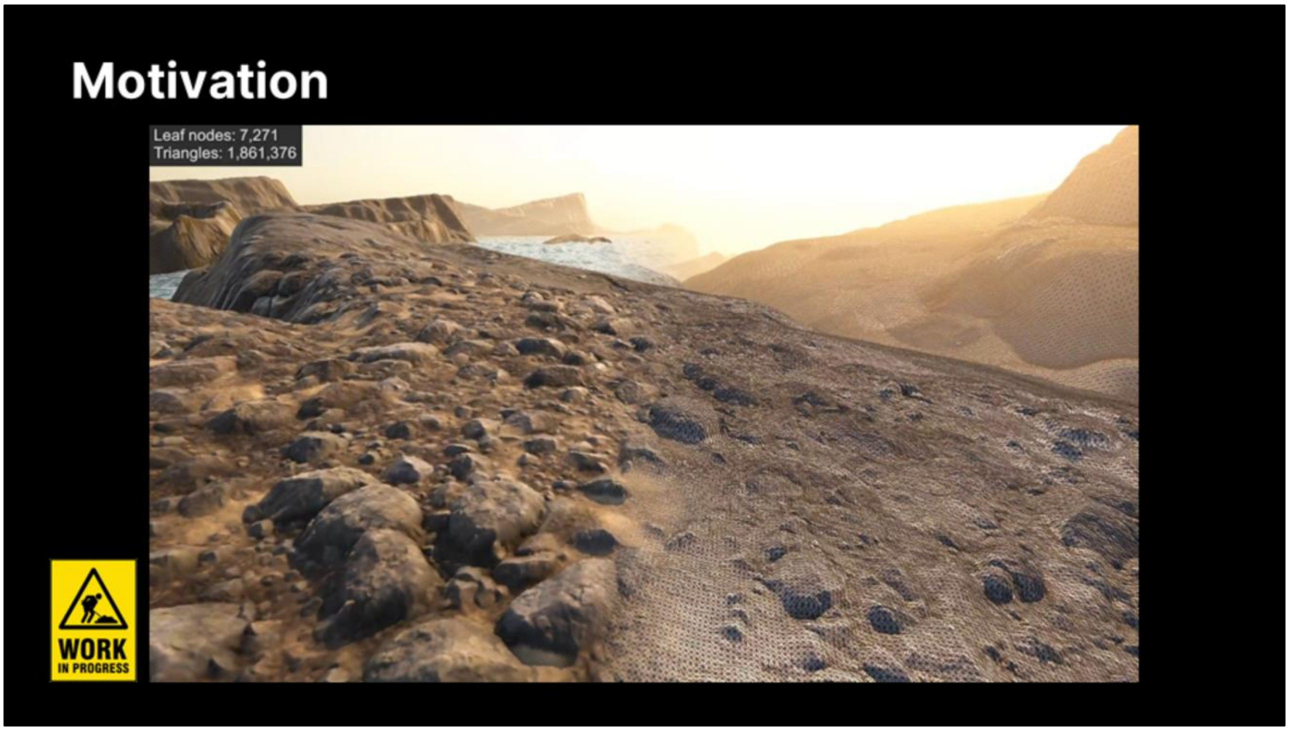
(still Xiaoling)
As a disclaimer, the content we are about to share is work in progress, and it is not a final product.

Motivation

[Xiaoling reads text below while getting his mouse on top of the video to press play]

With that being said, here’s a video that shows the current state of our terrain renderer.

[presses play, then silences while the video plays. Move on to next slide once animation is finished]

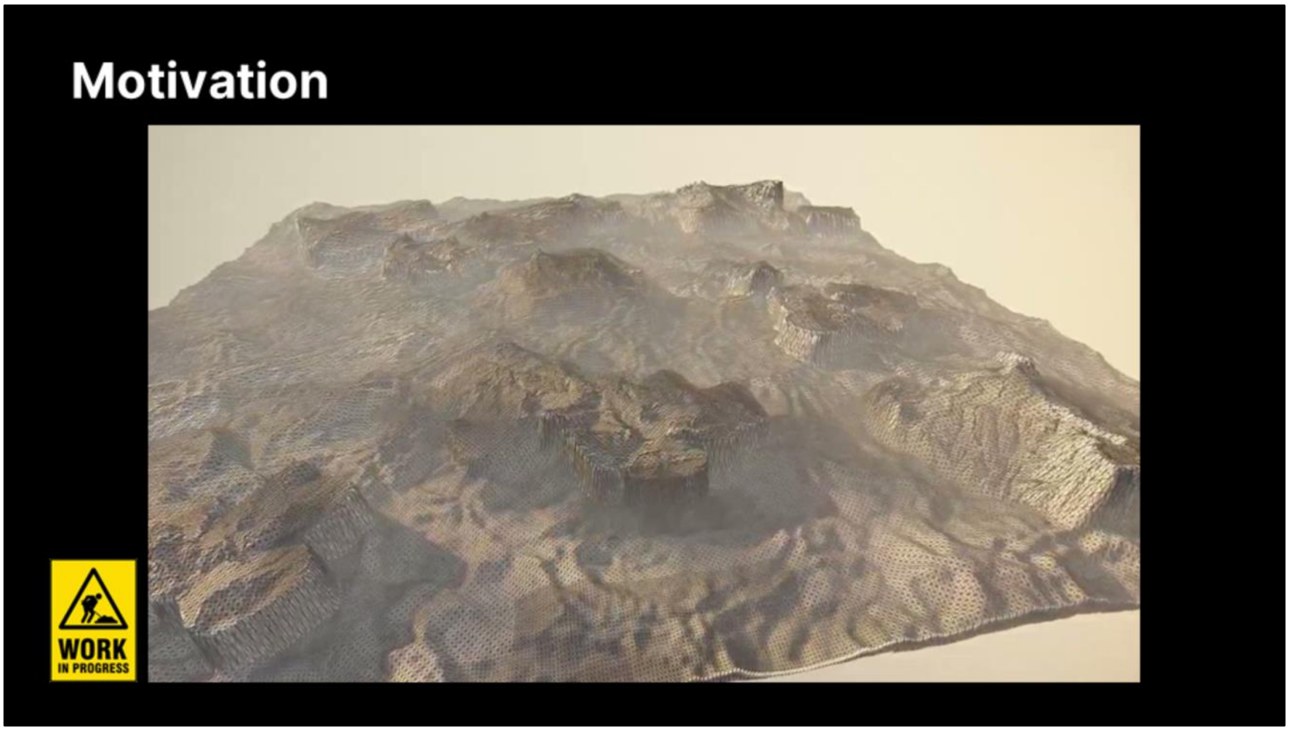


Motivation

(still Xiaoling -- shows shaded terrain animation video)
The video showed a 2 by 2 km terrain rendered in real time in Unity.

We rely on multiple components such as a virtual texture system, a biome system, and the adaptive terrain geometry.

In this presentation we will only focus on the geometry.

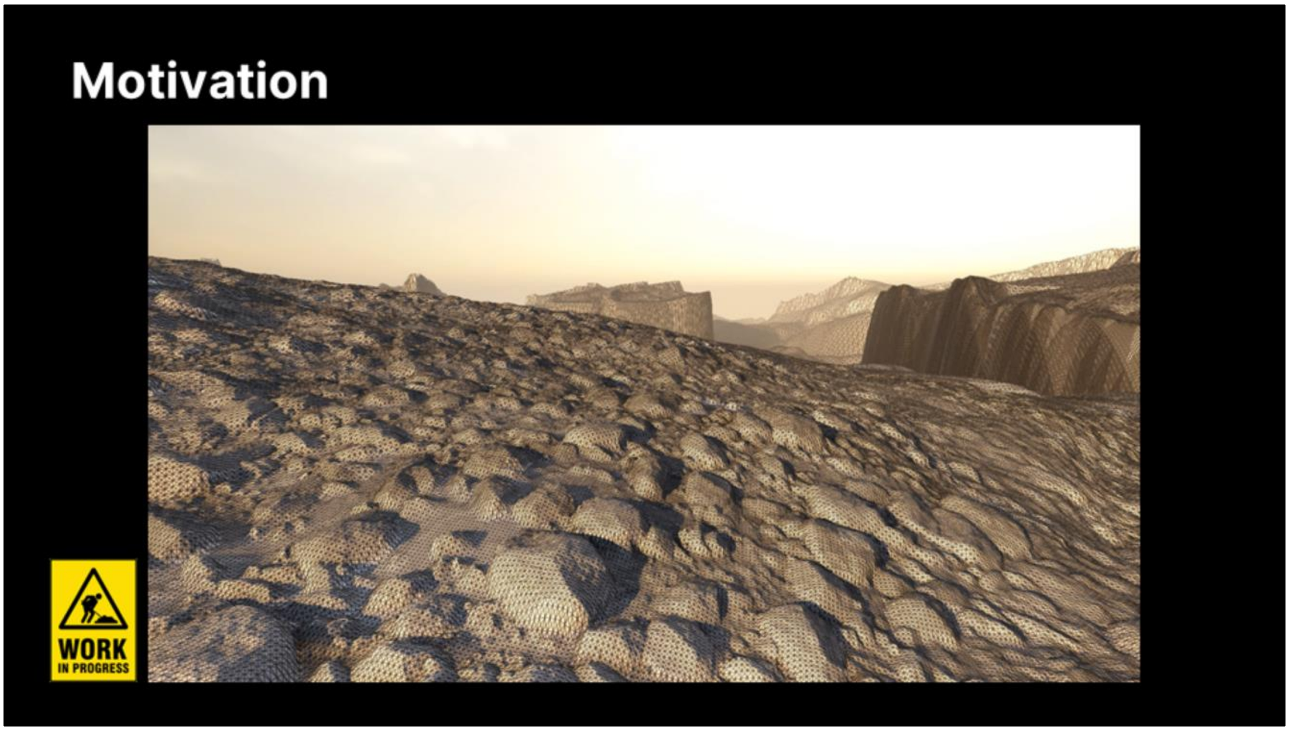


Motivation

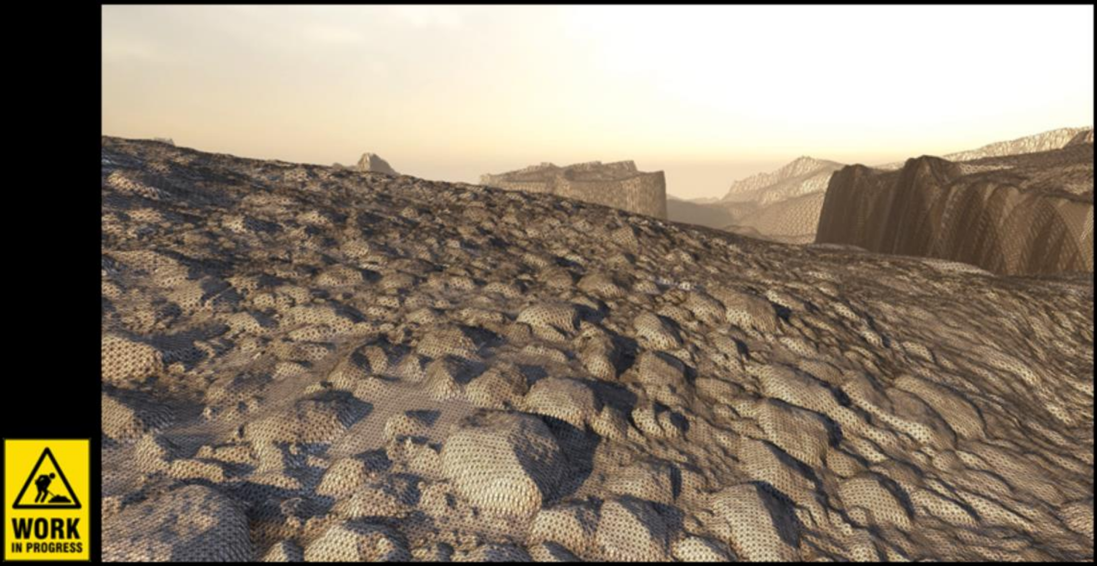
[Xiaoling reads text below while getting his mouse on top of the video to press play]

Let’s play the video again but this time with the geometry highlighted in wireframe.

[presses play, then silences while the video plays. Move on to next slide once animation is finished]



Motivation



(still Xiaoling)

As you can see, the wireframe is pretty dense: our current implementation produces triangles as small as 1.5cm over this terrain.

We produce these triangles thanks to a subdivision algorithm published in a recent paper from our research team at Unity Labs Grenoble. My colleague Thomas will now talk about the paper in more detail.

Concurrent Binary Trees for subdivision

Concurrent Binary Trees
(with application to longest edge bisection)

JONATHAN DUPUY, Unity Technologies

Fig. 1. A crack-free, 54x54 km terrain produced and rendered entirely on the GPU using the Unity game-engine. Internally, the engine leverages our new concurrent binary-tree data-structure to compute adaptive triangle tessellations on the GPU. Our concurrent data structure is lightweight, easy to implement and can be used to accelerate various computer graphics applications such as this adaptive terrain renderer, which renders in less than five milliseconds at full-HD resolution on an NVIDIA RTX 2080 GPU.

We introduce the concurrent binary tree (CBT), a novel concurrent representation to build and update arbitrary binary trees in parallel. Fundamentally, our representation consists of a binary heap, i.e., a 1D array, that explicitly stores the sum-reduction tree of a bitfield. In this bitfield, each one-valued bit represents a leaf node of the binary tree encoded by the CBT, which we locate algorithmically using a binary-search over the sum-reduction. We show that this construction allows to dispatch down to one thread per leaf node and that, in turn, these threads can safely split and/or remove nodes concurrently via simple bitwise operations over the bitfield. The practical benefit of CBTs lies in their ability to accelerate binary-tree-based algorithms with parallel processors. To support this claim, we leverage our representation to accelerate a longest edge-bisection-based algorithm that computes and renders adaptive geometry for large-scale terrains entirely on the GPU. For this specific algorithm, the CBT accelerates processing speed linearly with the number of processors.

21

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

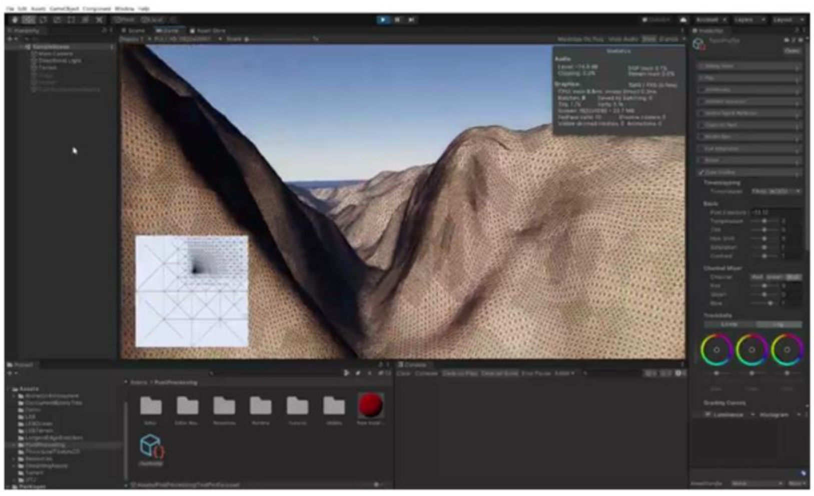
Additional Key Words and Phrases: binary tree, concurrent, parallel, binary heap, longest edge bisection, GPU, real-time

ACM Reference Format:
Jonathan Dupuy. 2020. Concurrent Binary Trees (with application to longest edge bisection). *Proc. ACM Comput. Graph. Interact. Tech.* 3, 2, Article 21 (August 2020), 20 pages. https://doi.org/10.1145/3406186

Author's address: Jonathan Dupuy, Unity Technologies, jonathan.dupuy@outlook.com.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, https://doi.org/10.1145/3406186.

Proc. ACM Comput. Graph. Interact. Tech., Vol. 3, No. 2, Article 21. Publication date: August 2020.



(Thomas)
Hi, I'm Thomas Deliot, I'm a research engineer at Unity Labs, and I work with the team that wrote this research paper here, published last year at HPG (high performance graphics)

The key contribution of this paper is to provide an adaptive tessellation scheme that is computed entirely on the GPU thanks to a novel data structure called the Concurrent Binary Tree.

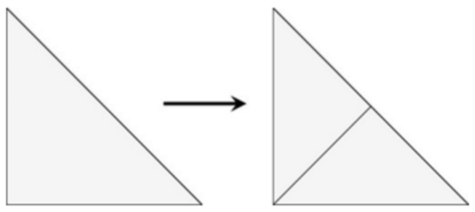
Here on the right you can see the video of the HPG paper and you can see that it's the same geometry as that Xiaoling just showed.

An important thing to note is that we don't rely on tessellation shaders at all. Rather, this geometry is only generated with compute and vertex shaders using an algorithm called Longest Edge Bisection, which I'm going to introduce first.

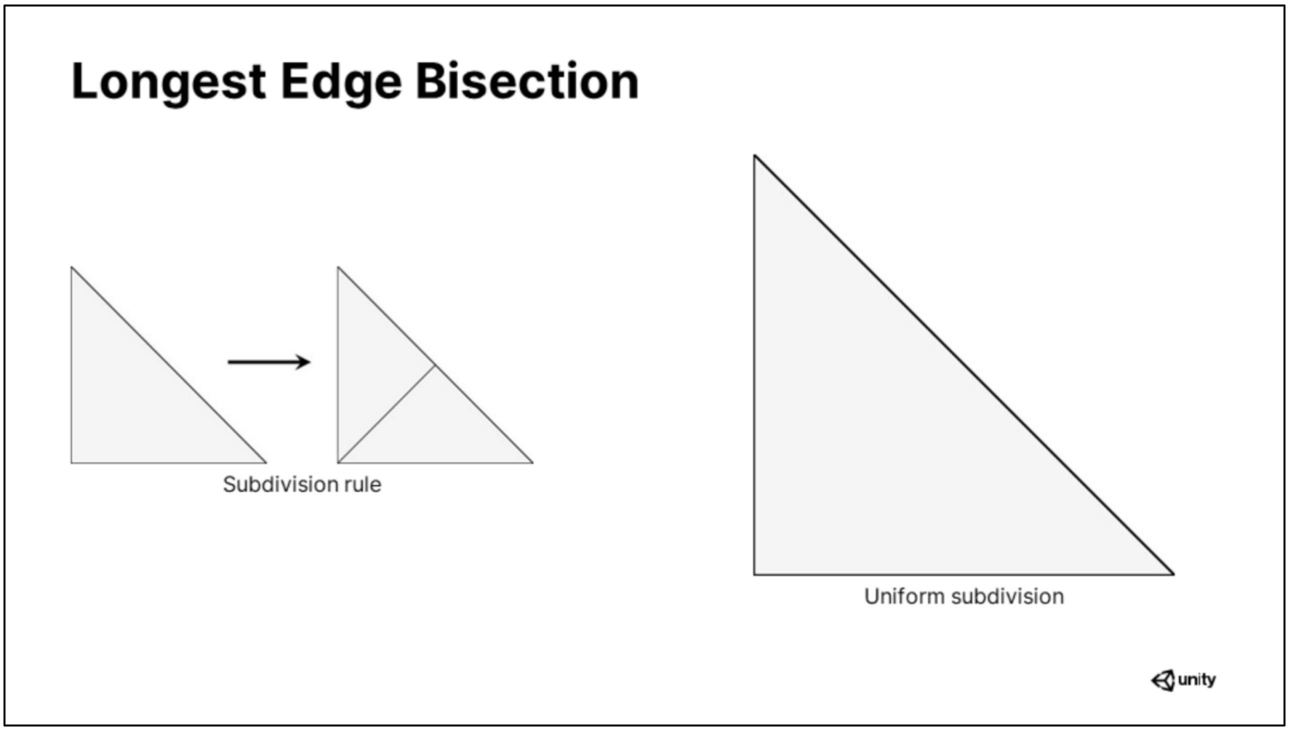
Longest Edge Bisection

Longest Edge Bisection is a subdivision scheme, and specifically it is the simplest form of subdivision scheme for 2D shapes

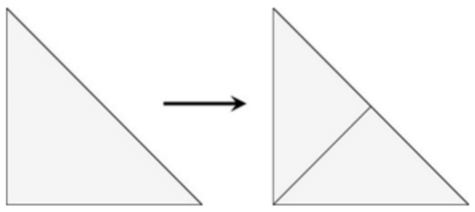
Longest Edge Bisection

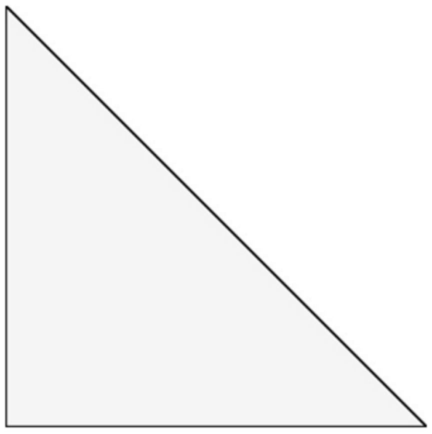

Subdivision rule

It has a simple rule : given a triangle, divide it into two triangles along it’s longest edge.



Longest Edge Bisection

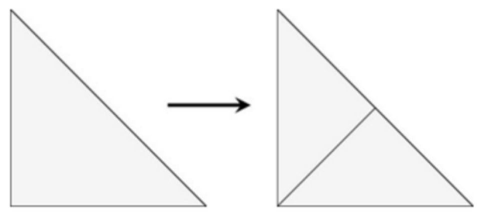


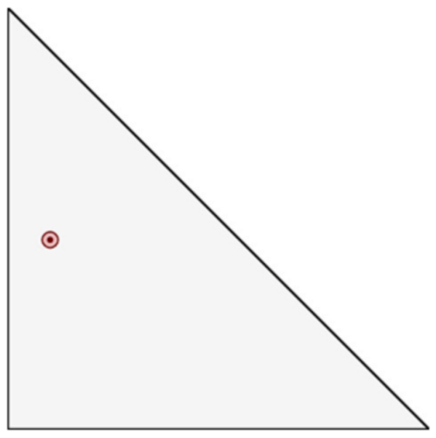


This can then be applied recursively on the two new triangles, and so on

This uniform subdivision gives us a regular tessellation of the original triangle

Longest Edge Bisection


Subdivision rule

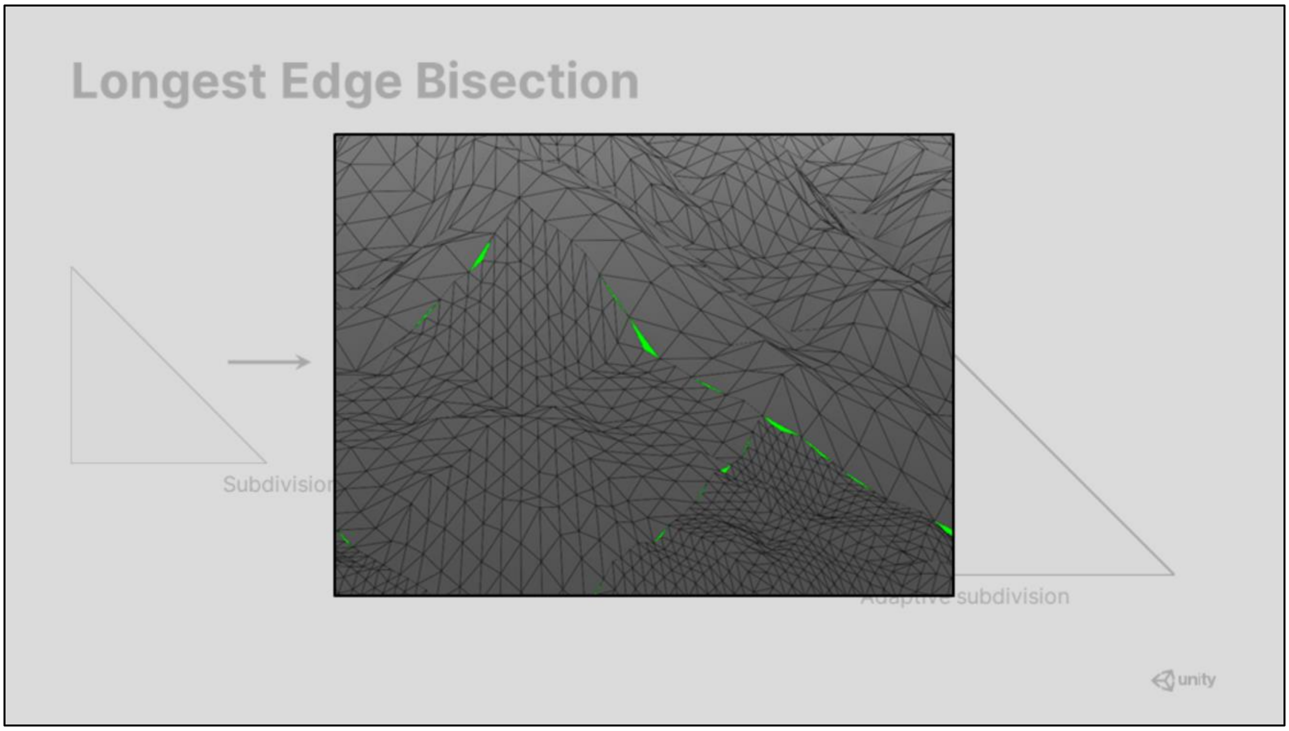

Adaptive subdivision

unity

Of course this can also be done non-uniformly to produce adaptive tessellations

For example, here we are targeting a finer subdivision around the red target

One important thing to notice is that the adaptive subdivision is free of t-junctions by construction



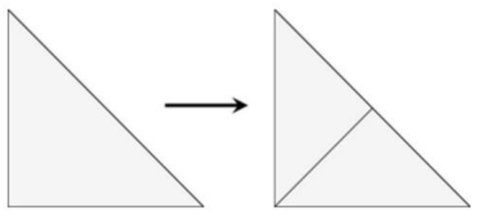
Longest Edge Bisection

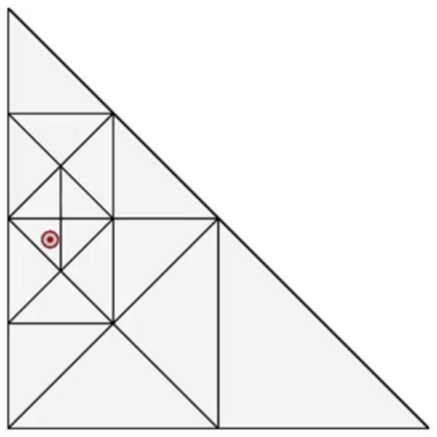
And avoiding T-junctions is a crucial property for terrains as it prevents cracks and holes in the final geometry.

Which is something that quad trees for example cannot achieve without further post processing.

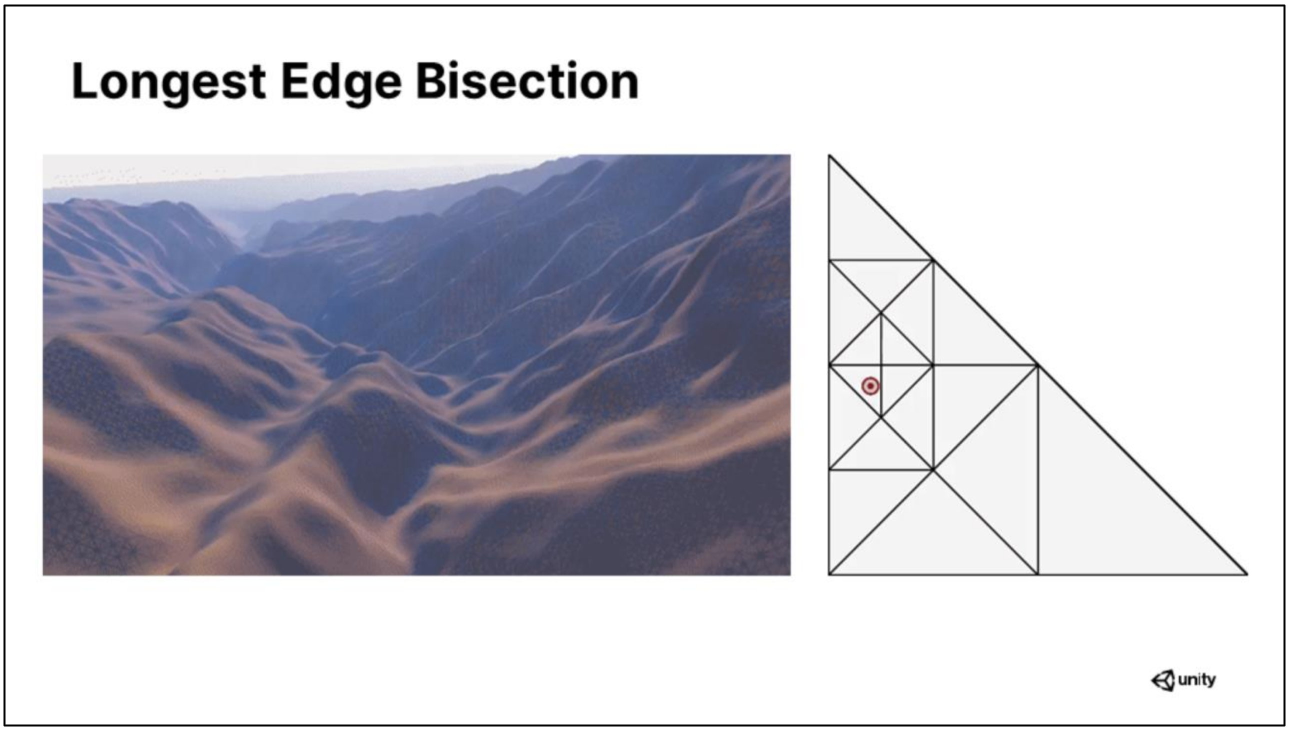
Image source : https://victorbush.com/2015/01/tessellated-terrain/

Longest Edge Bisection

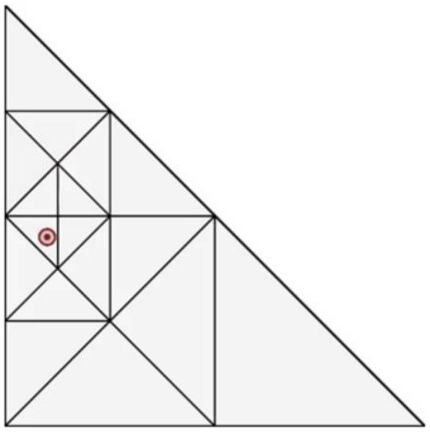

Subdivision rule



Finally if we apply this adaptive subdivision to a moving target, we get a level-of-detail system



Longest Edge Bisection



And if we apply this subdivision algorithm to a triangular mesh with a heightmap, we get an adaptive terrain mesh.

Since its adaptive, for every triangle of the subdivision we are deciding each frame whether to bisect it or not, based on an arbitrary level-of-detail criteria

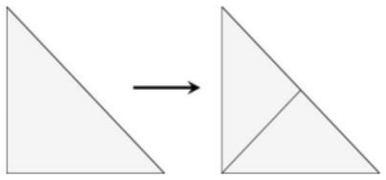
But how can we do this in real-time ?

There are way too many triangles in this example to iterate through each of them sequentially

And ideally we'd want to be able to dispatch this work to the GPU which is much more suited to handle such a workload, with thousands of parallel threads.

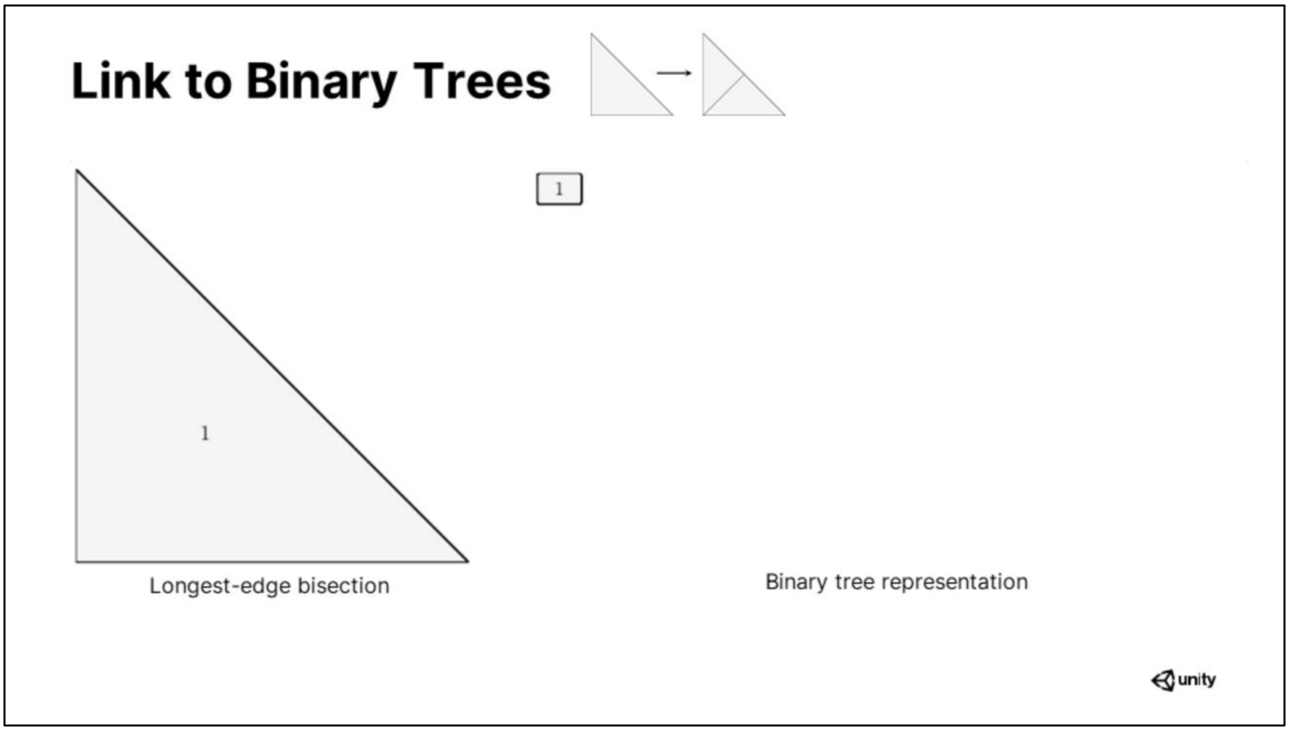
This is precisely what Concurrent Binary Trees bring to the table.

Link to Binary Trees



Subdivision rule

So if we look at that longest edge bisection scheme again, we see that it’s a binary subdivision rule : each triangle subdivides into two new triangles



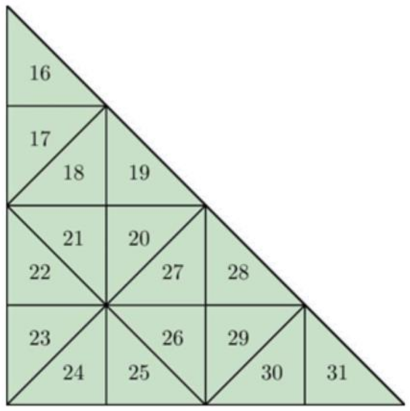
Link to Binary Trees

What that means is that we can represent the entire subdivision process as a binary tree

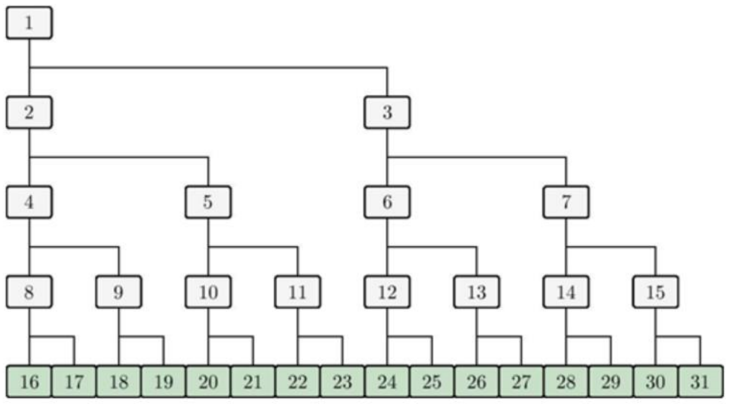
where each bisection creates two children nodes out of a parent node

Here in this uniform subdivision, each new level in the binary tree represents a new subdivision depth,

Link to Binary Trees



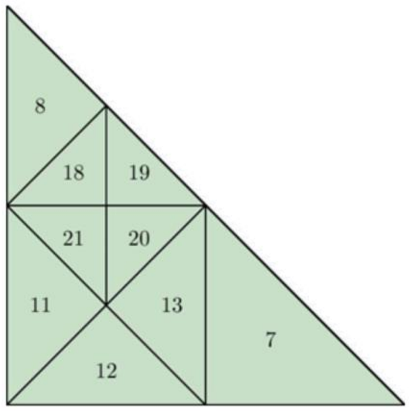
Longest-edge bisection



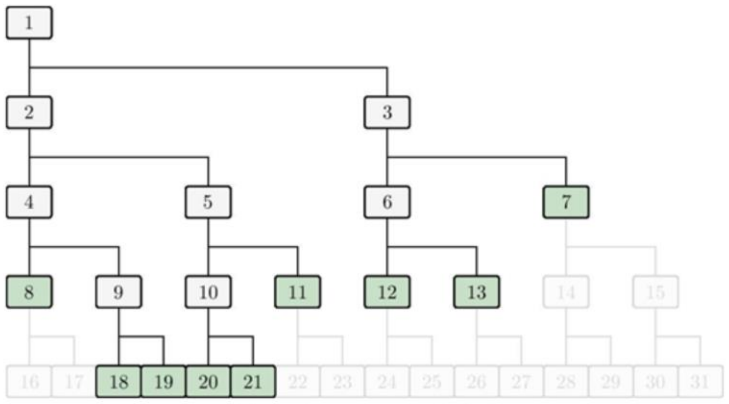
Binary tree representation

Which means that the leaf nodes of the binary tree representation (in green here) correspond to the triangles we want to display for the subdivision scheme

Link to Binary Trees

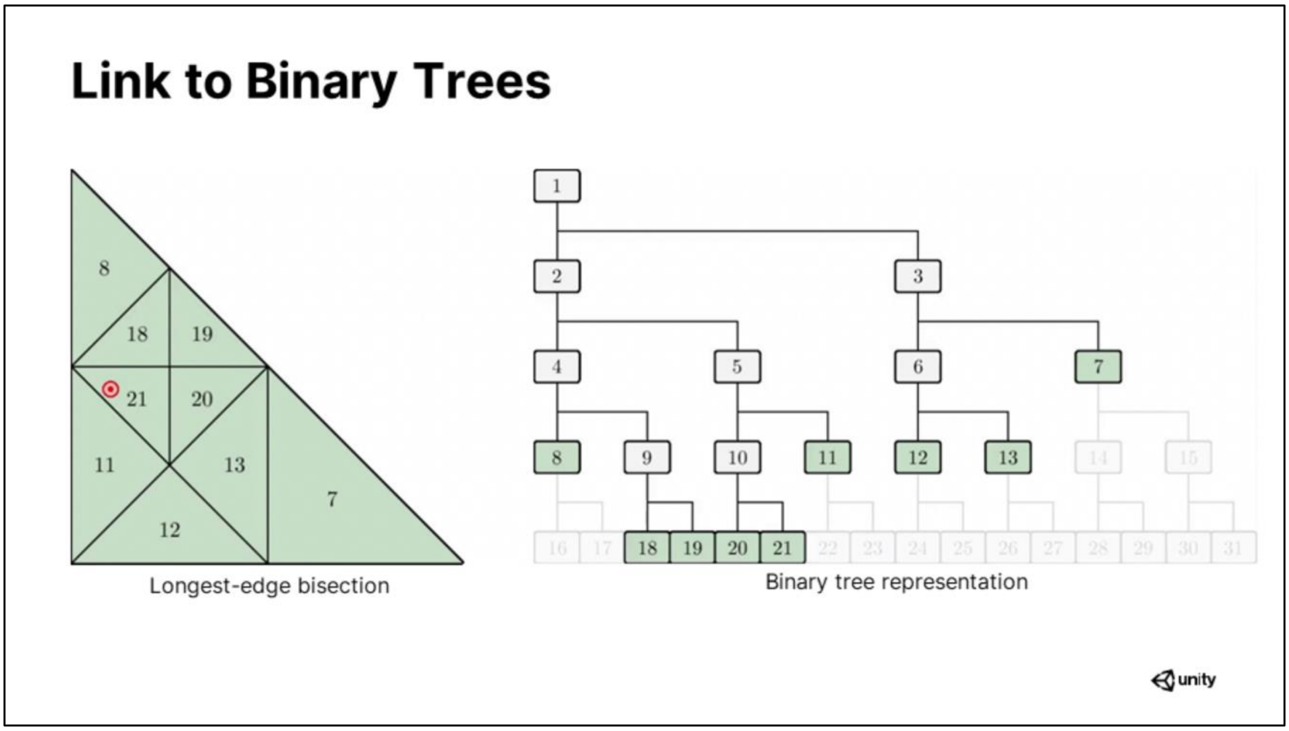


Longest-edge bisection

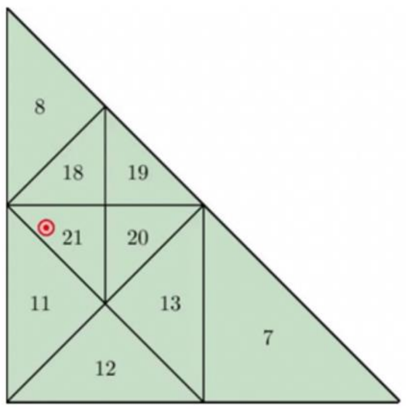


Binary tree representation

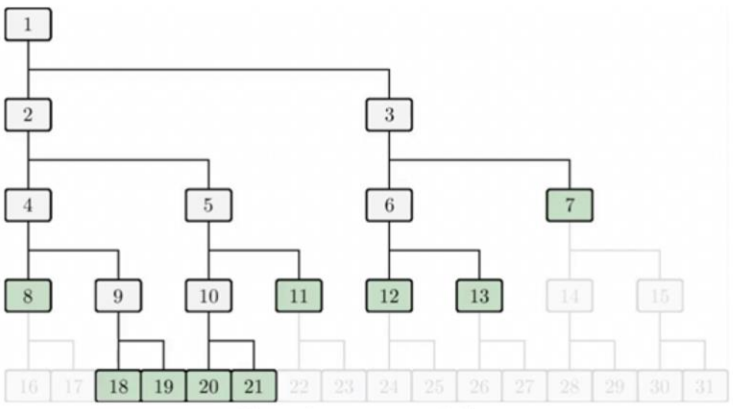
And this is also true in the case of an adaptive subdivision, where the smaller and bigger triangles displayed on the left correspond to a binary tree with leaf nodes at varying depth.



Link to Binary Trees



Longest-edge bisection



Binary tree representation

Basically this means that any state of this subdivision scheme can be represented by a specific binary tree.

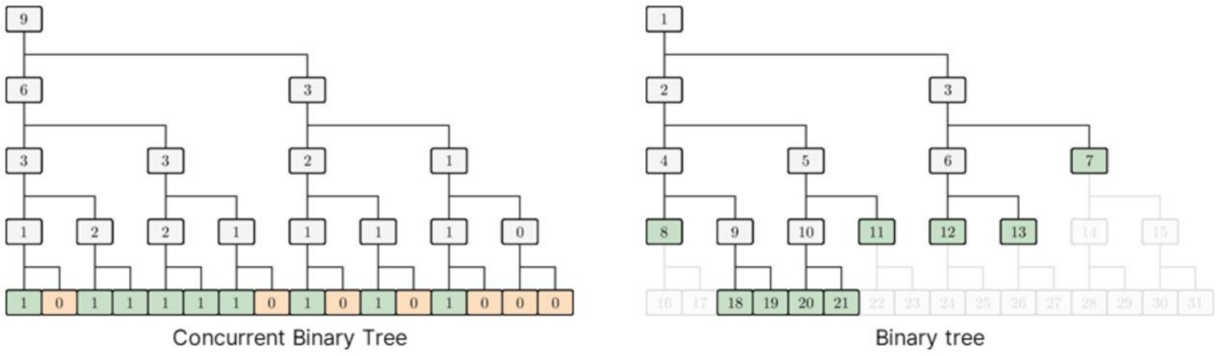
Enumerating the leaf nodes of the binary tree means looping over the current triangles of the subdivision scheme.

Bisecting a triangle in two means creating two new leaf nodes in the binary tree,

And vice-versa merging two triangles means deleting two sibling leaf nodes in the binary tree.

What Concurrent Binary Trees offer is a way to do these three operations concurrently, in parallel, by introducing a novel data structure.

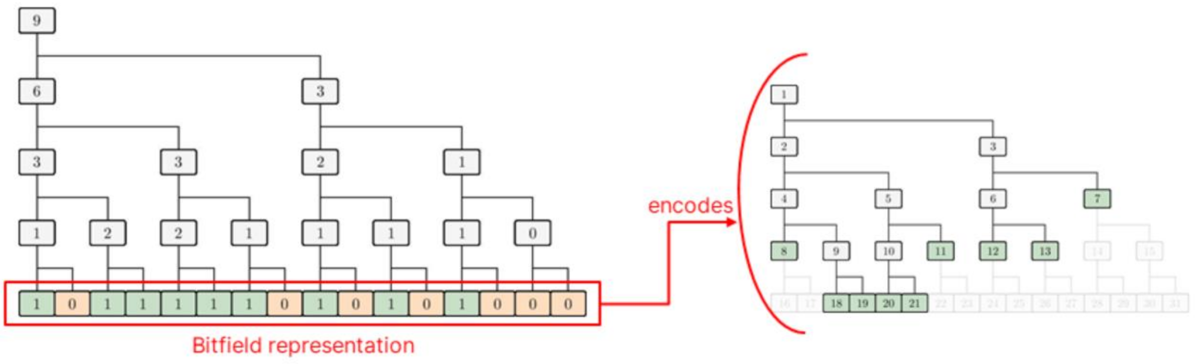
Concurrent Binary Trees



So here’s a Concurrent Binary Tree on the left. It is an alternative representation of the binary tree on the right.

And it’s different from a regular binary tree in a couple significant ways:

Concurrent Binary Trees

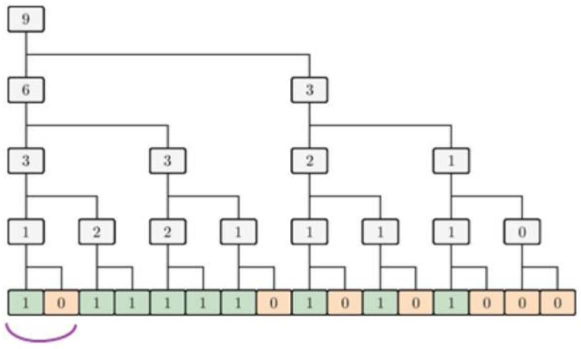


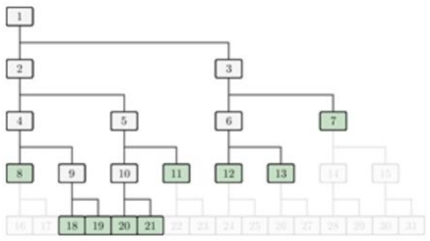
First, the last level of the Concurrent Binary Tree is a bitfield of ones and zeroes, that in itself, encodes another binary tree fully.

It does so by using ones to represent the presence of a leaf node

And a number of zeros after it to represent its level in the binary tree

Concurrent Binary Trees

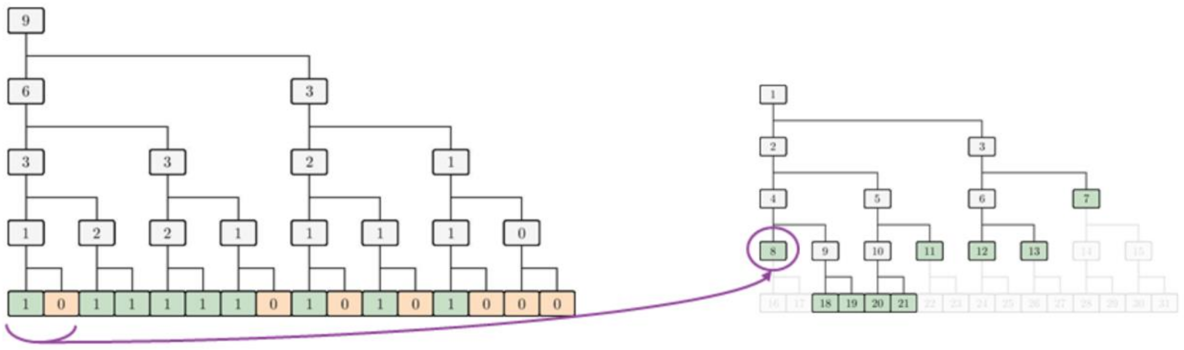




A few quick examples :

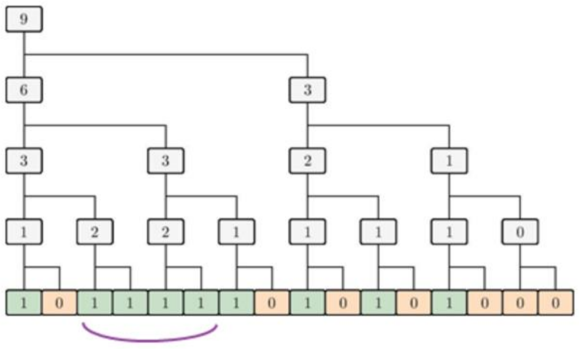
If we look at the first two bits in the bitfield, we have a one followed by a zero

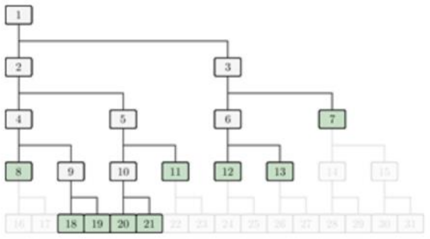
Concurrent Binary Trees



This means that we have a leaf node which is one level higher than the max depth of the binary tree

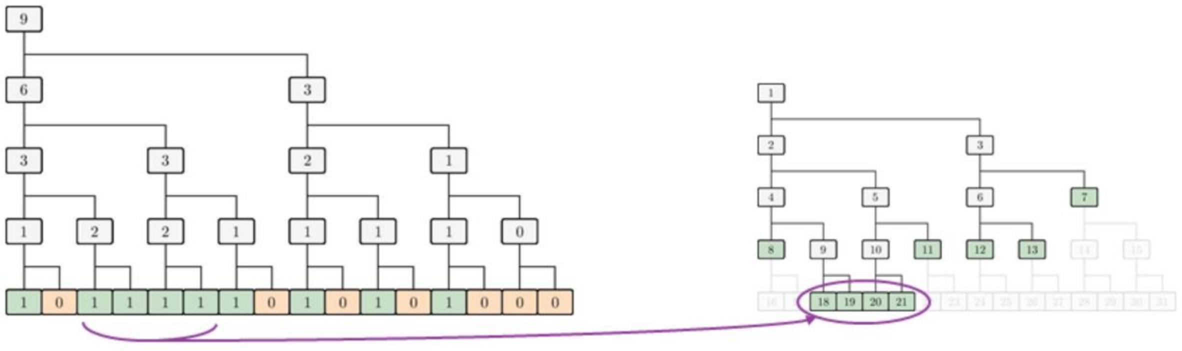
Concurrent Binary Trees





Now if we look at the next four bits, they are all ones without any zero behind them

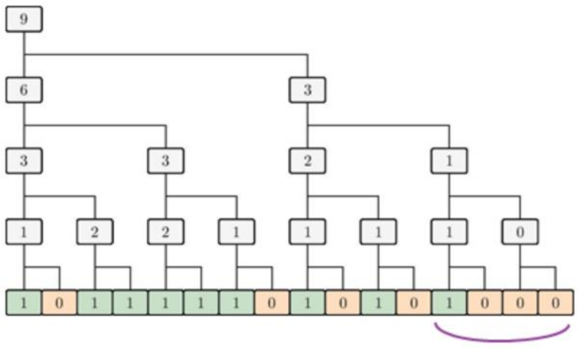
Concurrent Binary Trees

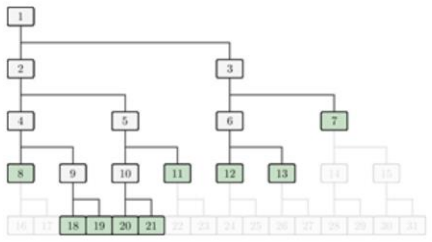


unity

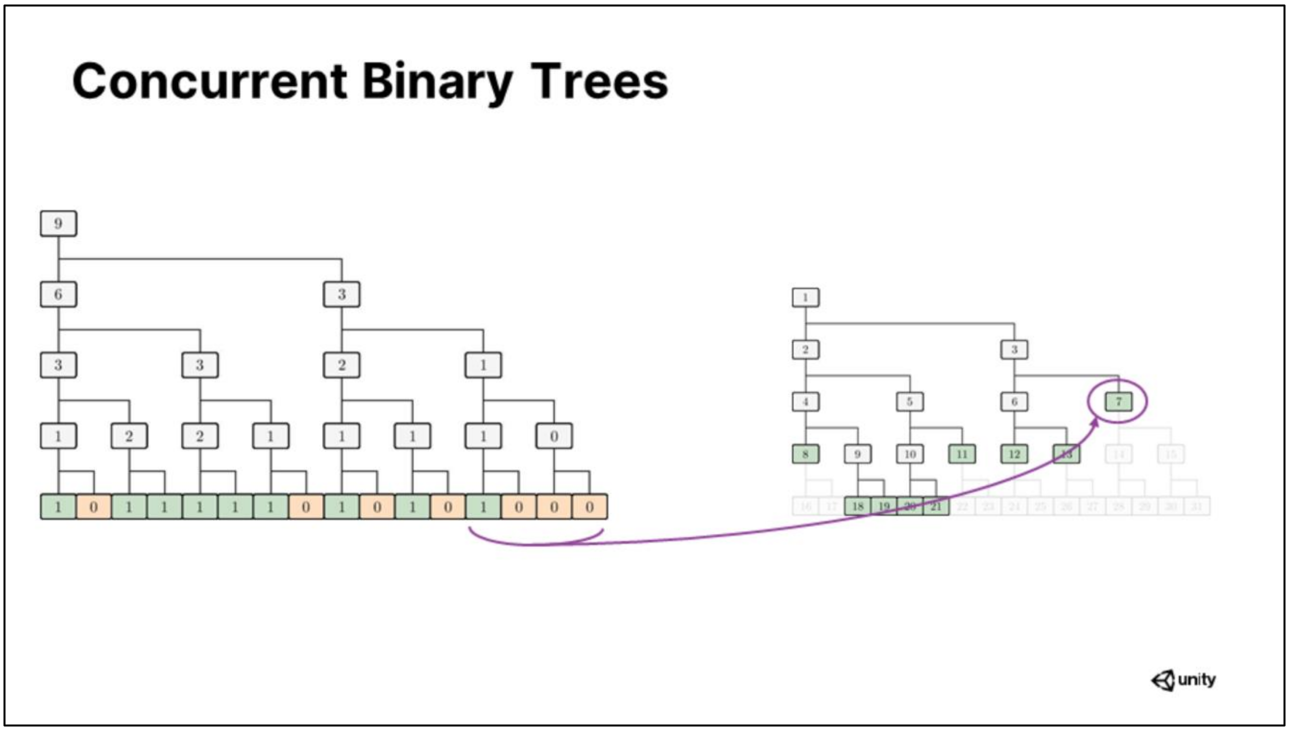
This means that we have four consecutive leaf nodes at the binary tree’s max depth

Concurrent Binary Trees

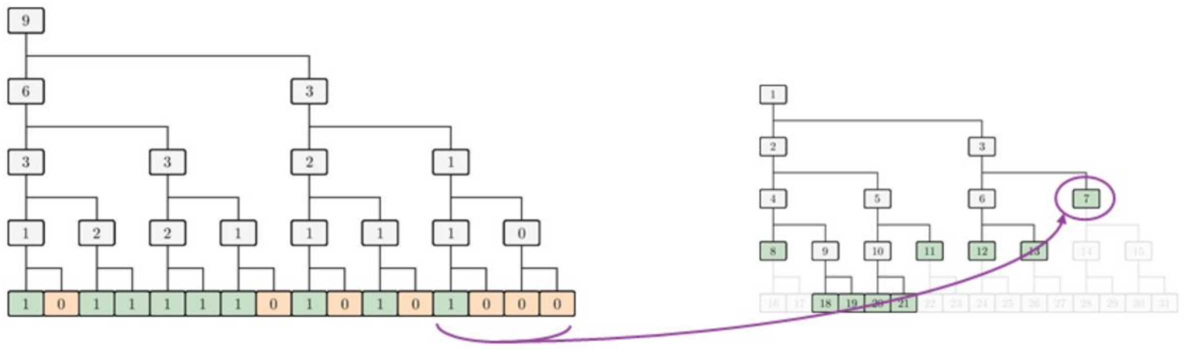




Finally if we look at the last bits in the bitfield, we have a one followed by three zeros



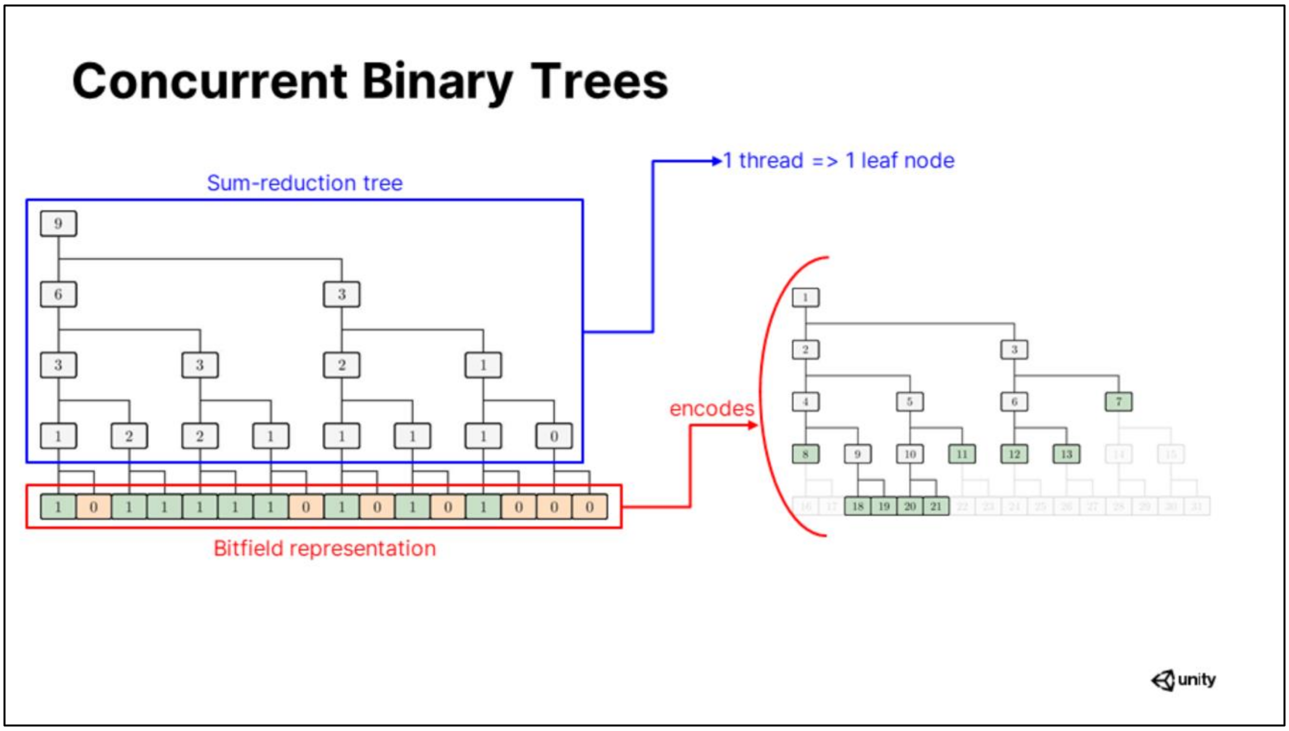
Concurrent Binary Trees



And this tells us that we have a leaf node which is two levels higher than the max depth of the tree. And so on.

This encoding of a binary tree in a flat bitfield is explained in much more detail in Jonathan’s paper and his HPG talk, which I recommend.

And it’s exploited by the other part of the Concurrent Binary Tree...



Concurrent Binary Trees

Which is the Sum-Reduction Tree, stored in all the other levels of the Concurrent Binary Tree.

It progressively sums the number of bits set to one (or leaf nodes) in the bitfield,

until the root node, which gives us the total amount of leaf nodes in the binary tree encoded by the bitfield

Then by traversing down this sum-reduction tree, it gives us a way to loop over the bits set to 1 while ignoring the zeros,

or in other words to loop over the leaf nodes.

Going back to our practical example where the encoded binary tree represents a Longest Edge Bisection,

we use this to assign one thread per active triangle in the subdivision.

Using these two components, Concurrent Binary Trees are able to process the binary tree associated with a longest edge bisection scheme, completely in parallel.

It dispatches individual threads to all active triangles, and uses bitwise atomic operations for splitting and merging leaf nodes.

And again, Jonathan’s presentation talks at length about how these operators are implemented in a concurrent manner, if you are interested.

For now I will focus on what it takes to implement and use them.

Rendering terrains with Concurrent Binary Trees

So while Jonathan was working on his paper he asked me to work on implementing his code in Unity to accompany the paper with a practical video and demo.

So what does it take to do this? Well there's a couple of systems to implement.

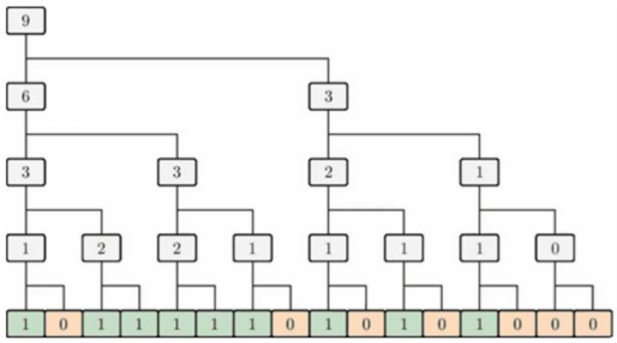
Libraries

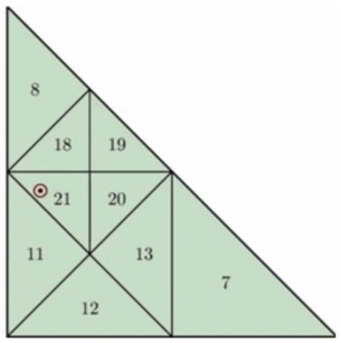
CBT Library

- cbt.h
- cbt.glsl

LEB Library

- leb.h
- leb.glsl





The first step is importing the two function libraries that Jonathan provides with the paper : the CBT library and the LEB library

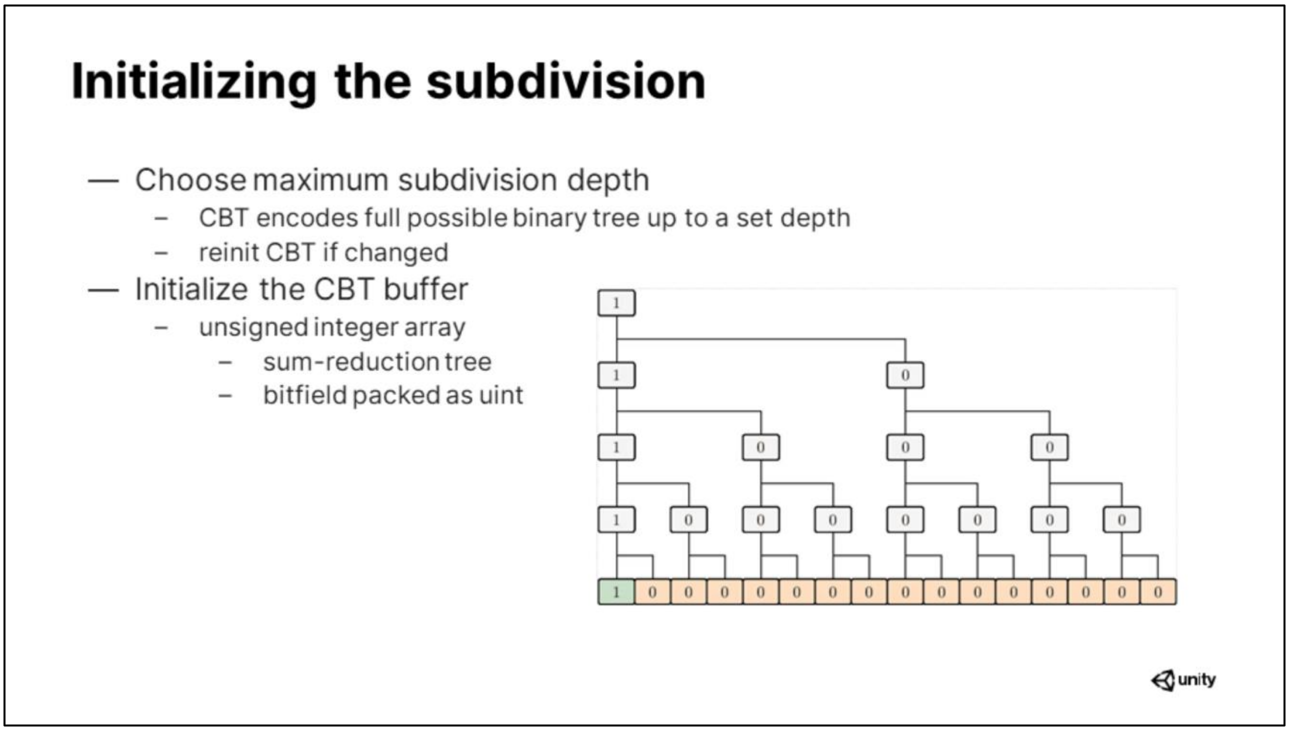
They are single file libraries with both a cpu and gpu version.

The first library provides all the tools to manipulate the Concurrent Binary Tree : fetching data, splitting and merging nodes, and so on.

The other implements all the code required to compute Longest Edge Bisection schemes in parallel using a Concurrent Binary Tree.

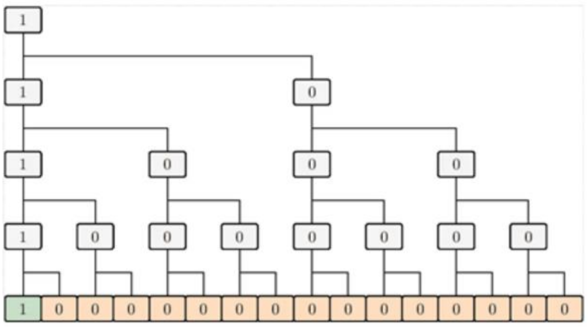
So, getting triangle vertices, subdivision rules for splitting and merging, getting sibling and parent nodes, etc.

Of course we are mostly interested in their gpu versions, but having the cpu code at hand is useful for testing.



Initializing the subdivision

- Choose maximum subdivision depth
  - CBT encodes full possible binary tree up to a set depth
  - reinit CBT if changed
- Initialize the CBT buffer
  - unsigned integer array
    - sum-reduction tree
    - bitfield packed as uint



Then the next thing to do is initializing a concurrent binary tree to its base state.

To do that we first need to decide on a maximum subdivision depth, which corresponds to the maximum binary tree depth

This dictates the size of the bitfield we want to allocate and the max depth of the binary tree it can fully encode.

And in practise this will determine the maximum terrain resolution for a terrain of a certain size.

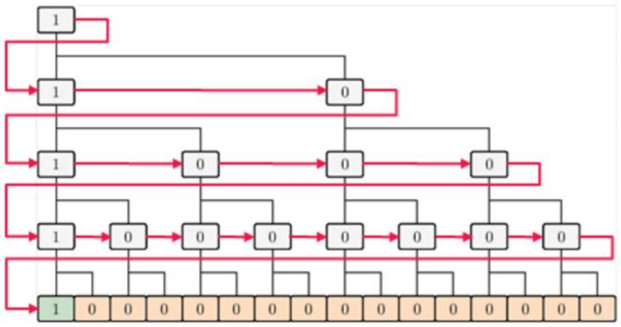
As a quick sidenote, the concurrent binary tree is a pointer-free data structure

In the memory, it is simply a flat integer array

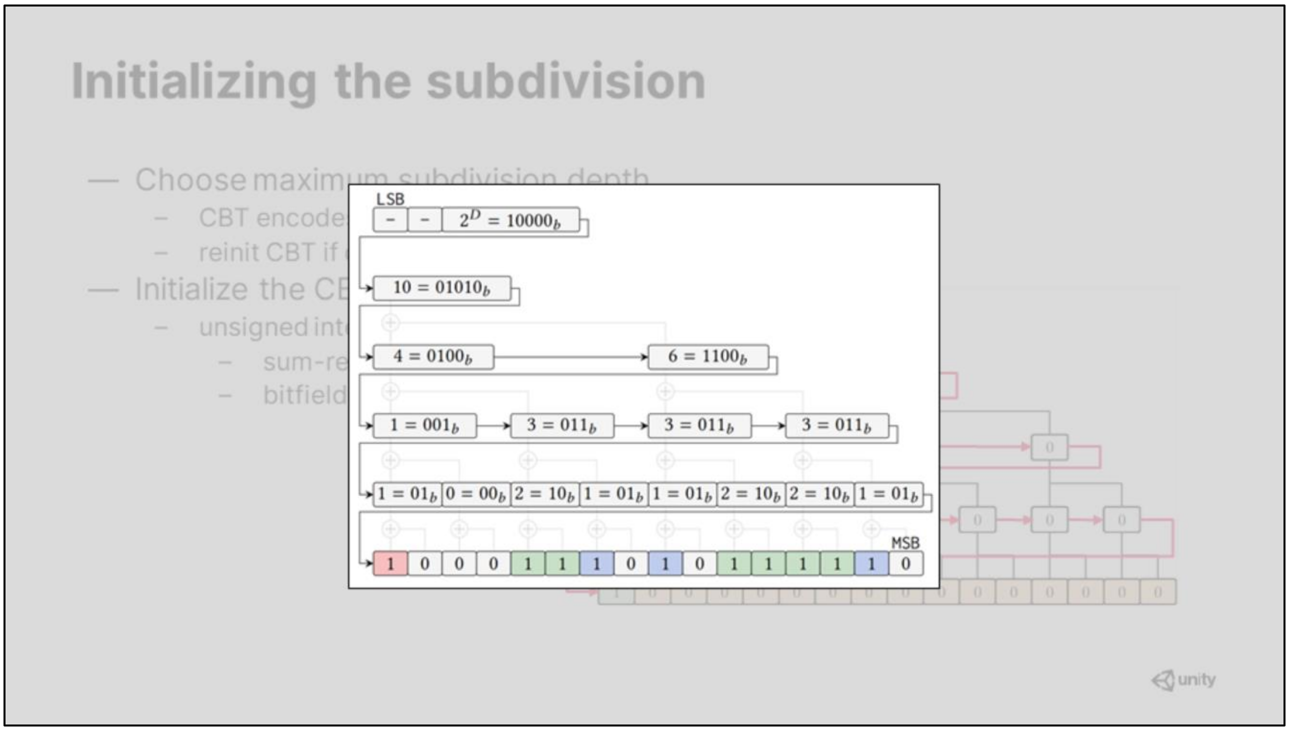
It contains first the sum-reduction tree in sequential order like this

Initializing the subdivision

- Choose maximum subdivision depth
  - CBT encodes full possible binary tree up to a set depth
  - reinit CBT if changed
- Initialize the CBT buffer
  - unsigned integer array
    - sum-reduction tree
    - bitfield packed as uint



And then the rest is the bitfield at the end



Initializing the subdivision

- Choose maximum subdivision depth
  - CBT encode
  - reinit CBT if
- Initialize the CB
  - unsigned int
    - sum-re
    - bitfield

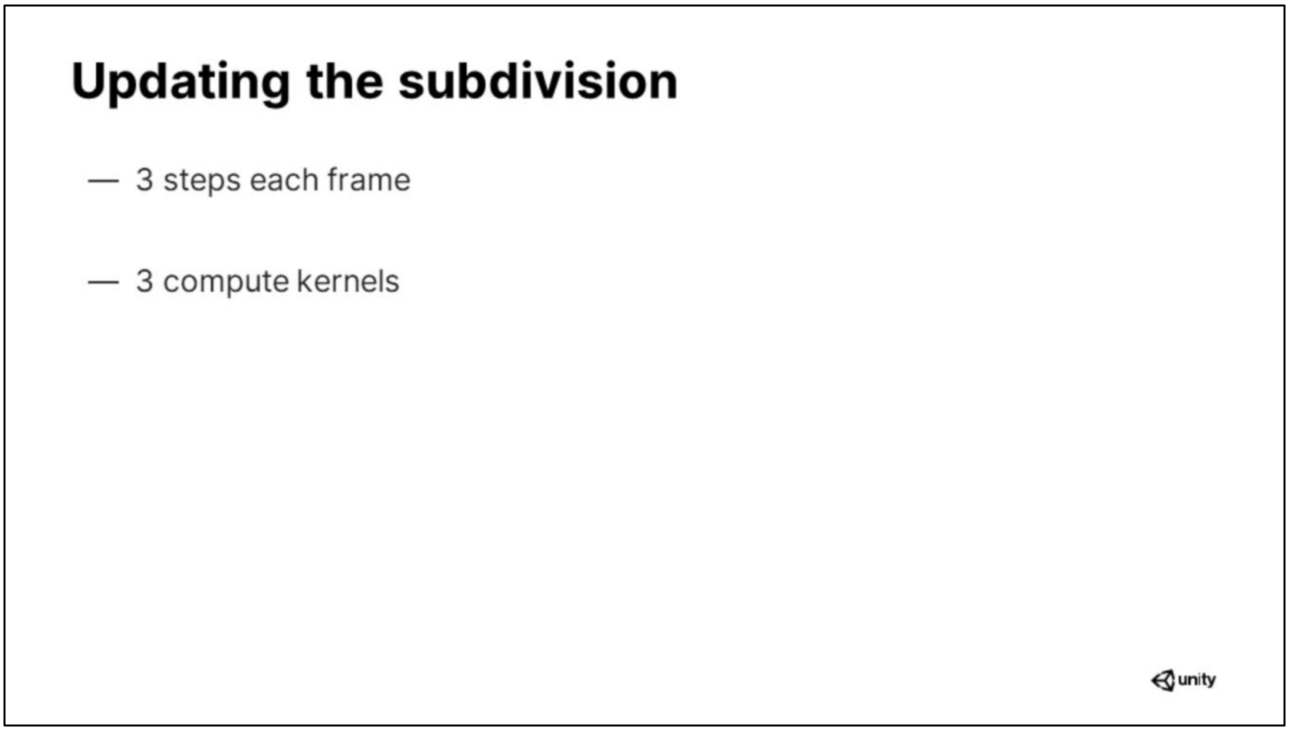
And another quick sidenote, I need to add that in practise,

concurrent binary trees use an optimized memory layout which allows packing all this data into the least possible amount of bytes

The bitfield at the bottom packs 32 nodes together into a single integer

Then because the deepest level of the sum-reduction tree can only have a value of 0, 1 or 2, it’s represented by two bit nodes packed in integers, and so on.

Jonathan explains this optimized memory layout in detail in his paper.



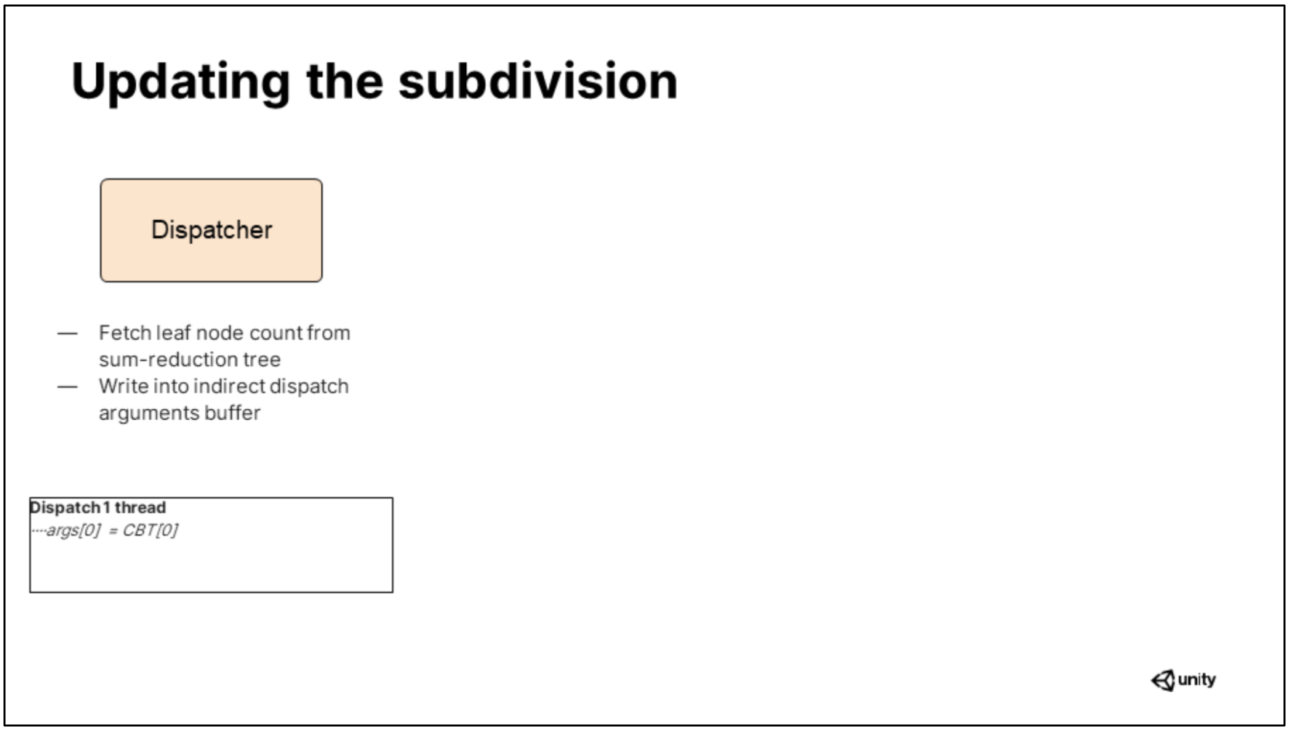
Updating the subdivision

- 3 steps each frame
- 3 compute kernels

Ok let’s get back to our implementation

There are three different steps now that we need to perform each frame during the update loop, to update the longest edge bisection

And in practise they correspond to three different compute shaders



Updating the subdivision

Dispatcher

- Fetch leaf node count from sum-reduction tree
- Write into indirect dispatch arguments buffer

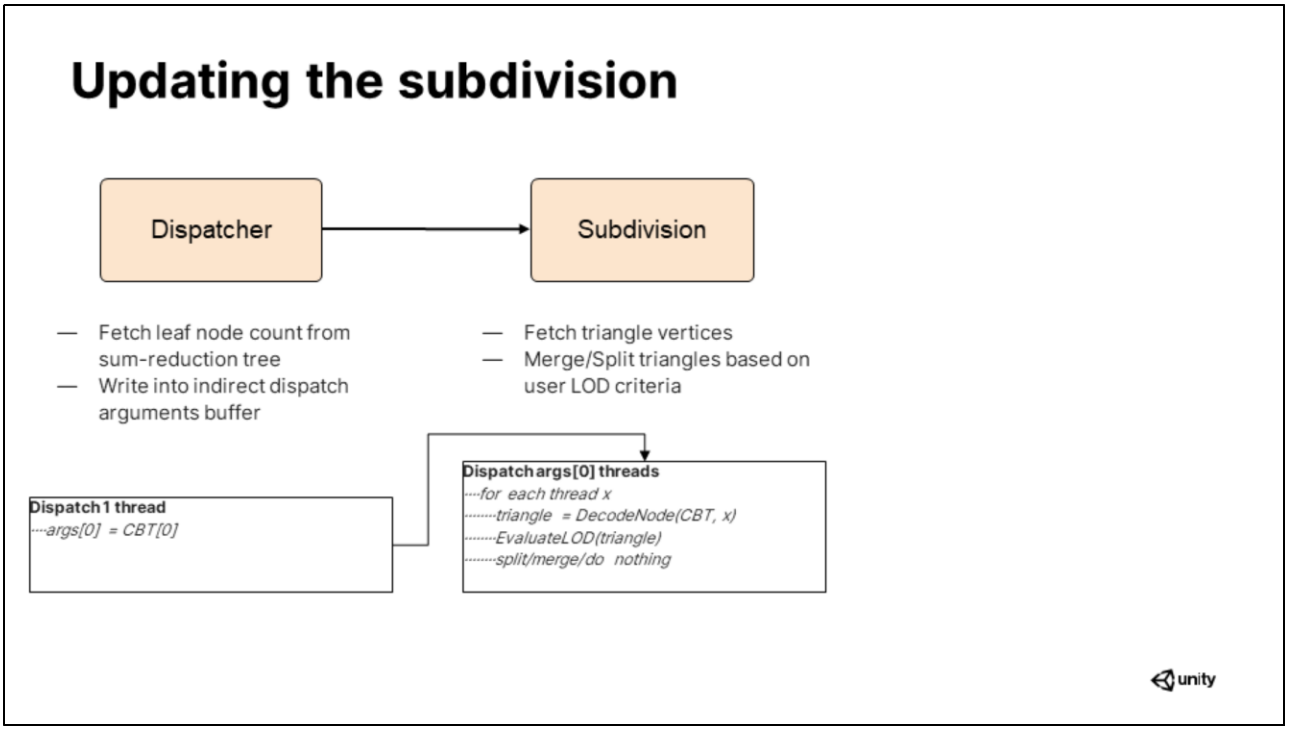
```
Dispatch 1 thread
····args[0] = CBT[0]
```

First, Because our concurrent binary tree lives in GPU memory,

the initial step is to dispatch one thread on the GPU to read the root node of the sum-reduction tree,

which gives us the amount of leaf nodes in the subdivision, or active triangles.

We write this value into a indirect dispatch argument buffer...



Updating the subdivision

- Fetch leaf node count from sum-reduction tree
- Write into indirect dispatch arguments buffer

- Fetch triangle vertices
- Merge/Split triangles based on user LOD criteria

```
Dispatch 1 thread
----args[0] = CBT[0]
```

```
Dispatch args[0] threads
----for each thread x
--------triangle = DecodeNode(CBT, x)
--------EvaluateLOD(triangle)
--------split/merge/do nothing
```

Which we use for the next step to dispatch on the GPU one thread per triangle for the subdivision update.

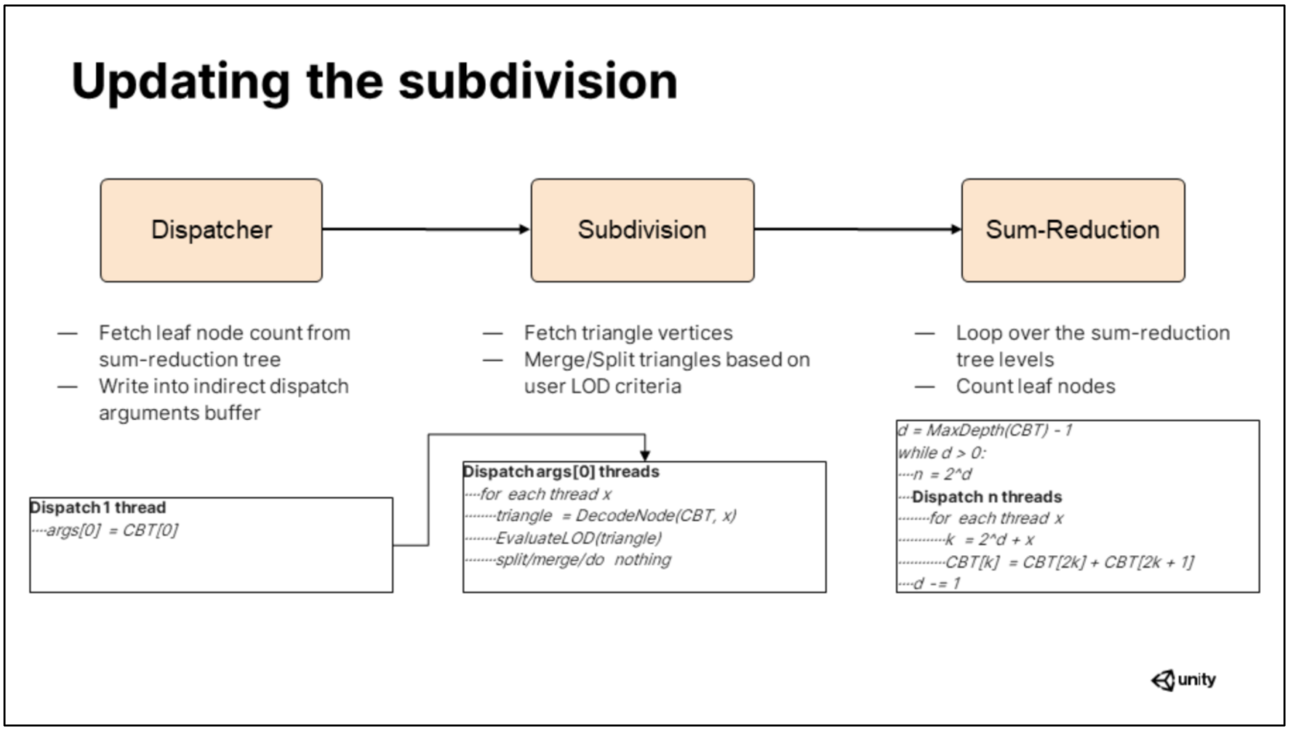
Each thread gets assigned to its triangle using the sum-reduction tree.

And once we've fetched the triangle vertices, we can decide whether or not we want to split it or merge it.

This depends entirely on the chosen level of detail criteria, which I'll talk about a bit later.

If we want to split or merge a triangle, we use the LEB library to do so while preventing cracks in the geometry

And the CBT library to update the binary tree using atomic operations to avoid race conditions.



Updating the subdivision

Dispatcher → Subdivision → Sum-Reduction

Dispatcher

- Fetch leaf node count from sum-reduction tree
- Write into indirect dispatch arguments buffer

```
Dispatch 1 thread
    args[0] = CBT[0]
```

Subdivision

- Fetch triangle vertices
- Merge/Split triangles based on user LOD criteria

```
Dispatch args[0] threads
    for each thread x
        triangle = DecodeNode(CBT, x)
        EvaluateLOD(triangle)
        split/merge/do nothing
```

Sum-Reduction

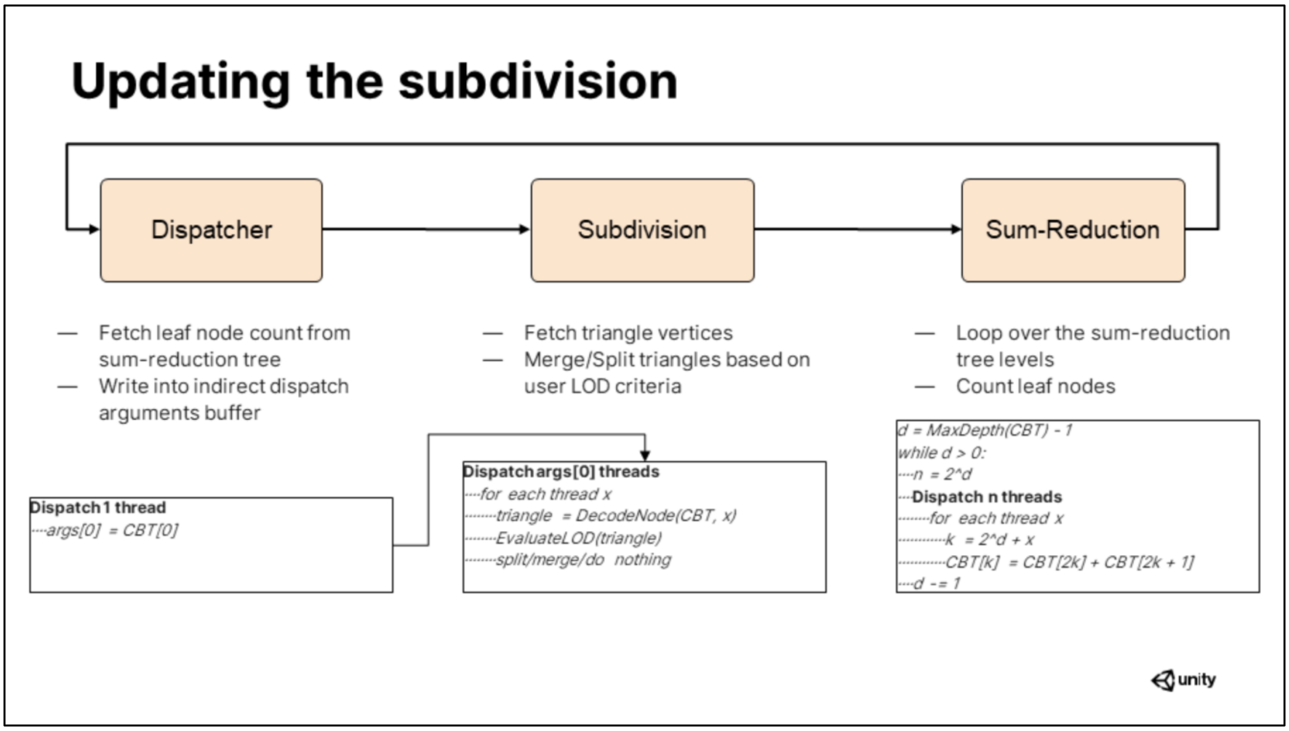
- Loop over the sum-reduction tree levels
- Count leaf nodes

```
d = MaxDepth(CBT) - 1
while d > 0:
    n = 2^d
    Dispatch n threads
        for each thread x
            k = 2^d + x
            CBT[k] = CBT[2k] + CBT[2k + 1]
    d -= 1
```

The last step, since during the Subdivision dispatch we might've created new leaf nodes or deleted some,

is to update the sum-reduction tree.

We use another compute kernel repeatedly for this, iterating over each level in the sum-reduction tree while summing the leaf node count.



Updating the subdivision

Dispatcher → Subdivision → Sum-Reduction → Dispatcher

Dispatcher
- Fetch leaf node count from sum-reduction tree
- Write into indirect dispatch arguments buffer

```
Dispatch 1 thread
    args[0] = CBT[0]
```

Subdivision
- Fetch triangle vertices
- Merge/Split triangles based on user LOD criteria

```
Dispatch args[0] threads
    for each thread x
        triangle = DecodeNode(CBT, x)
        EvaluateLOD(triangle)
        split/merge/do nothing
```

Sum-Reduction
- Loop over the sum-reduction tree levels
- Count leaf nodes

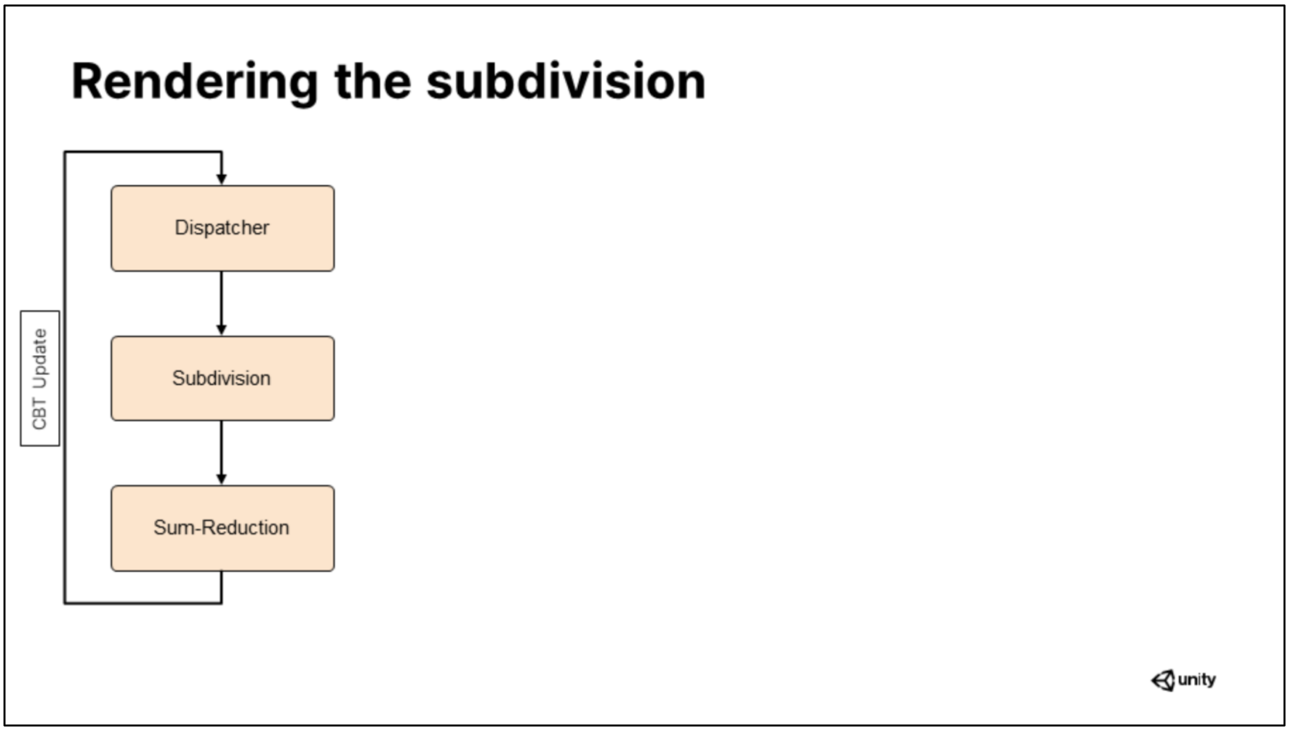
```
d = MaxDepth(CBT) - 1
while d > 0:
    n = 2^d
    Dispatch n threads
        for each thread x
            k = 2^d + x
            CBT[k] = CBT[2k] + CBT[2k + 1]
    d -= 1
```

unity

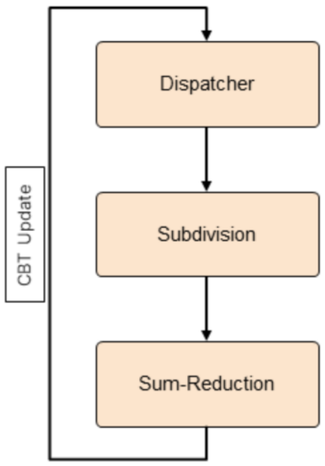
And finally, we go to the next frame and start again with the Dispatcher kernel which will read the new leaf node count value,

to prepare for the next subdivision step.

And that’s it for updating the longest edge bisection.



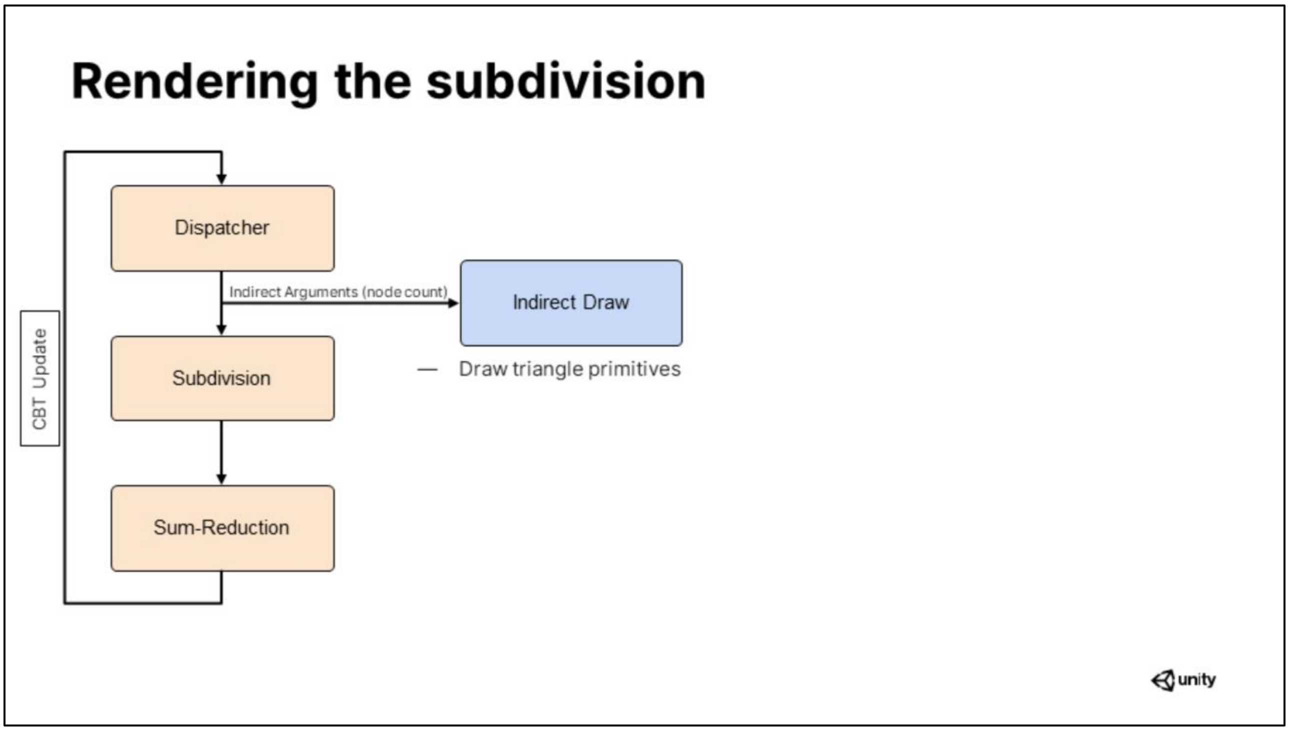
Rendering the subdivision



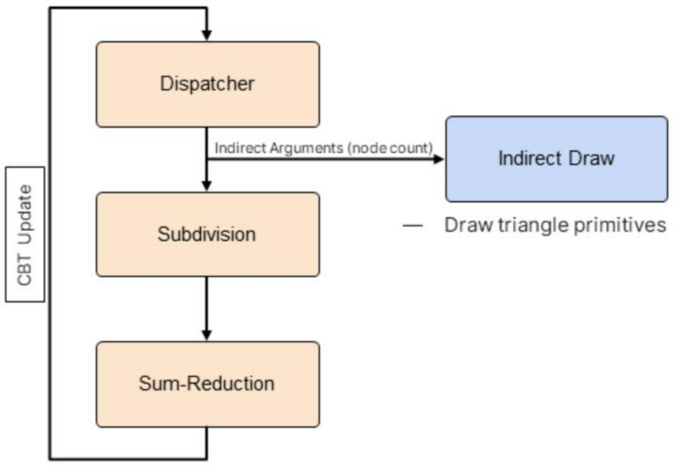
Now that we know how to update our subdivision, we need to also render it to the screen.

We want to render all the active triangles of the subdivision, which correspond to the leaf nodes of our binary tree.

How do we do this ? It’s pretty simple :

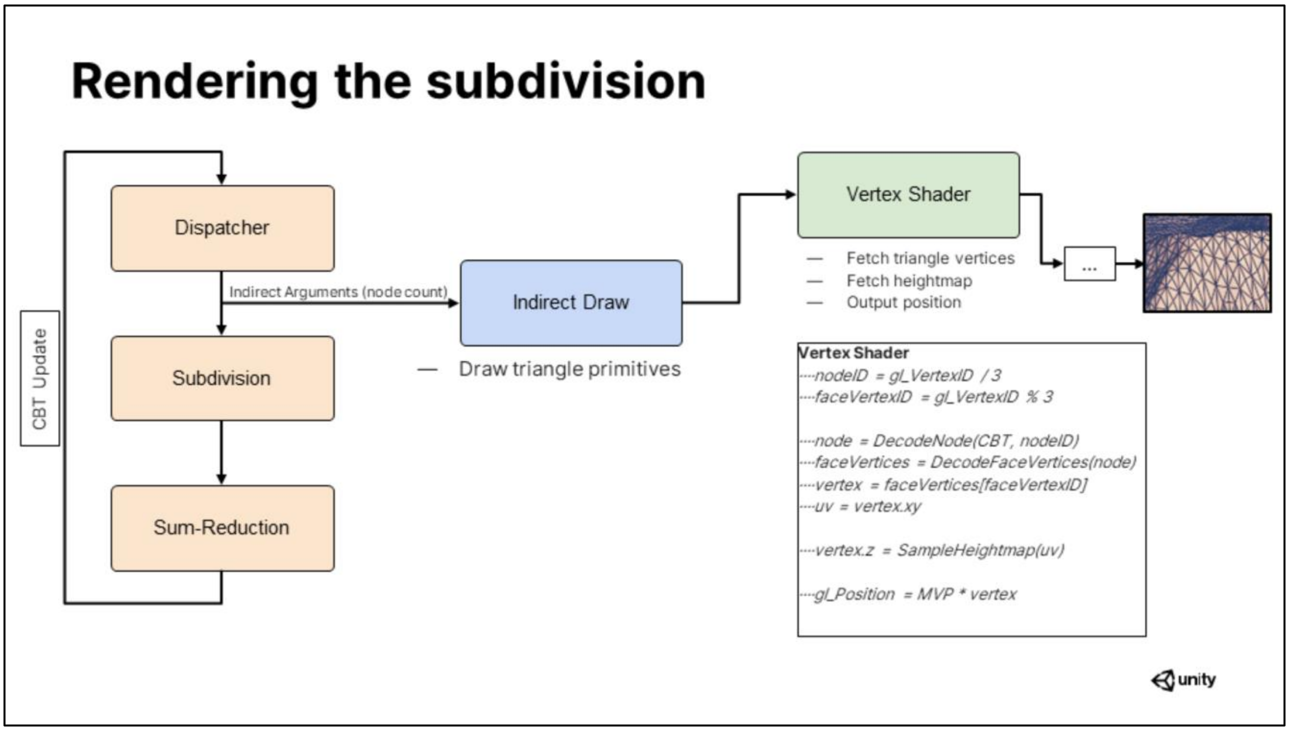


Rendering the subdivision



we use the dispatcher kernel again to read the amount of leaf nodes, and output that to a indirect draw arguments buffer,

That we use to draw triangle primitives procedurally.



Rendering the subdivision

```
Vertex Shader
----nodeID = gl_VertexID / 3
----faceVertexID = gl_VertexID % 3

----node = DecodeNode(CBT, nodeID)
----faceVertices = DecodeFaceVertices(node)
----vertex = faceVertices[faceVertexID]
----uv = vertex.xy

----vertex.z = SampleHeightmap(uv)

----gl_Position = MVP * vertex
```

This will spawn three vertex shader threads per triangle,

and in that vertex shader we use the LEB library to fetch the triangle vertices associated to the current leaf node,

And choose the correct one.

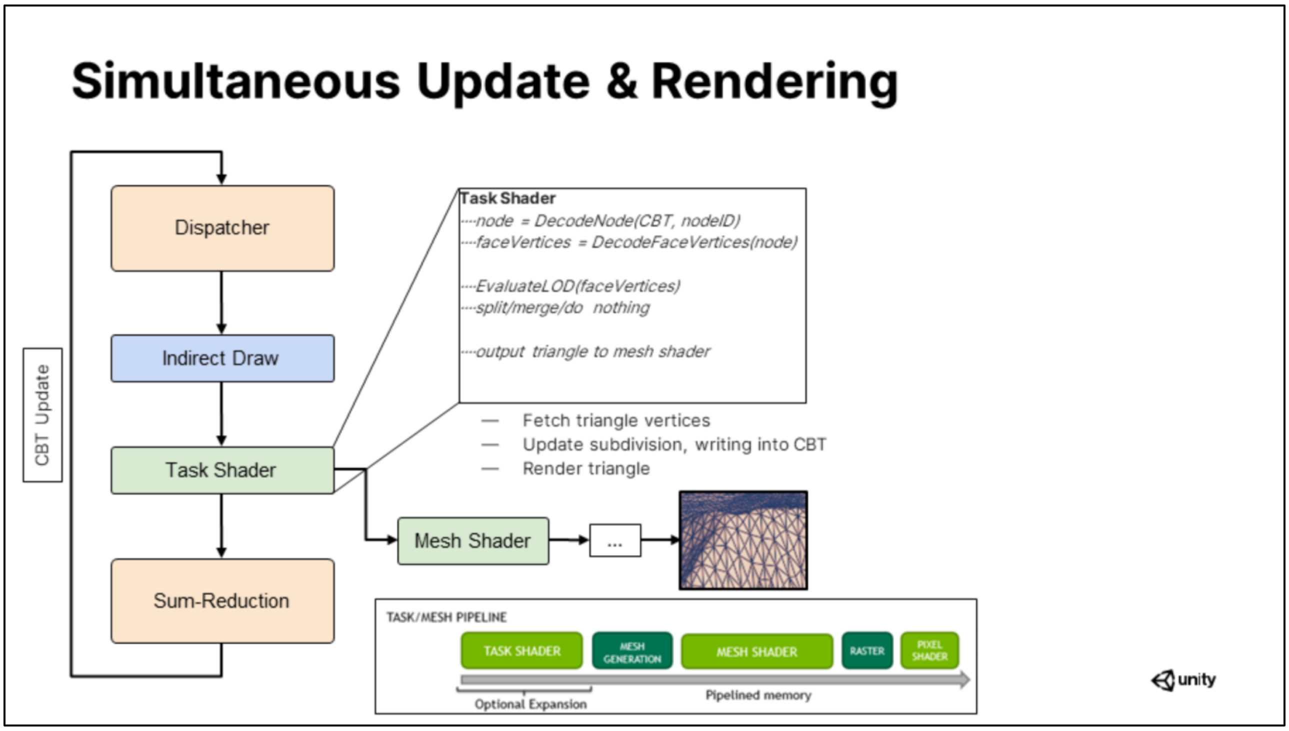
And right before projecting it into clip space,

We can use its 2D position as its uv texture coordinate,

since our subdivision happens on a flat unit square.

With this we can sample the heightmap and apply it to our terrain vertices.

Then we output the transformed vertex to the rest of the pipeline and shade our terrain how we want



Simultaneous Update & Rendering

Task Shader

```
····node = DecodeNode(CBT, nodeID)
····faceVertices = DecodeFaceVertices(node)

····EvaluateLOD(faceVertices)
····split/merge/do  nothing

····output  triangle to mesh shader
```

- Fetch triangle vertices
- Update subdivision, writing into CBT
- Render triangle

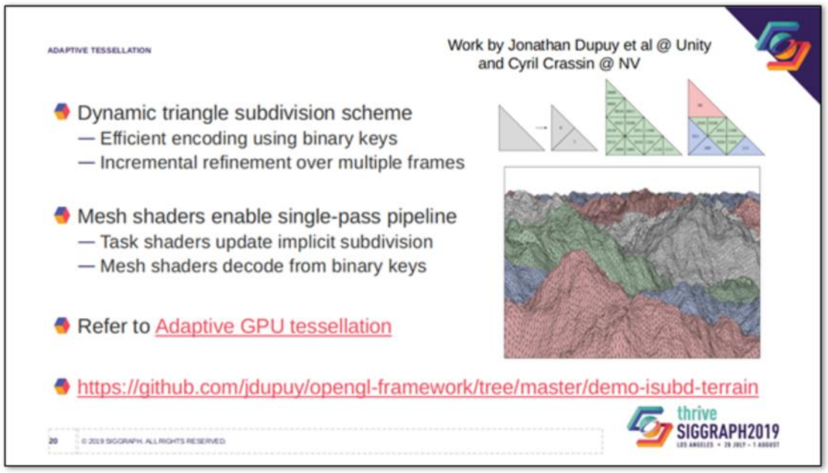
On an another subject, it would be possible to update the subdivision and render it simultaneously, with the newly introduced Mesh Shader API.

Mesh shaders basically introduce a “compute-like” capability to the first stage of traditional rendering shaders, and replaces vertex shaders.

Here we would simply update the subdivision during the Task Shader, and let the Mesh Shader read from it right afterwards for immediate rendering.

Simultaneous Update & Rendering

Mesh Shading: Towards Greater Efficiency of Geometry Processing
Yury Uralsky, SIGGRAPH 2019


ADAPTIVE TESSELLATION

Work by Jonathan Dupuy et al @ Unity and Cyril Crassin @ NV

- Dynamic triangle subdivision scheme
  - Efficient encoding using binary keys
  - Incremental refinement over multiple frames
- Mesh shaders enable single-pass pipeline
  - Task shaders update implicit subdivision
  - Mesh shaders decode from binary keys
- Refer to Adaptive GPU tessellation
- https://github.com/jdupuy/opengl-framework/tree/master/demo-isubd-terrain

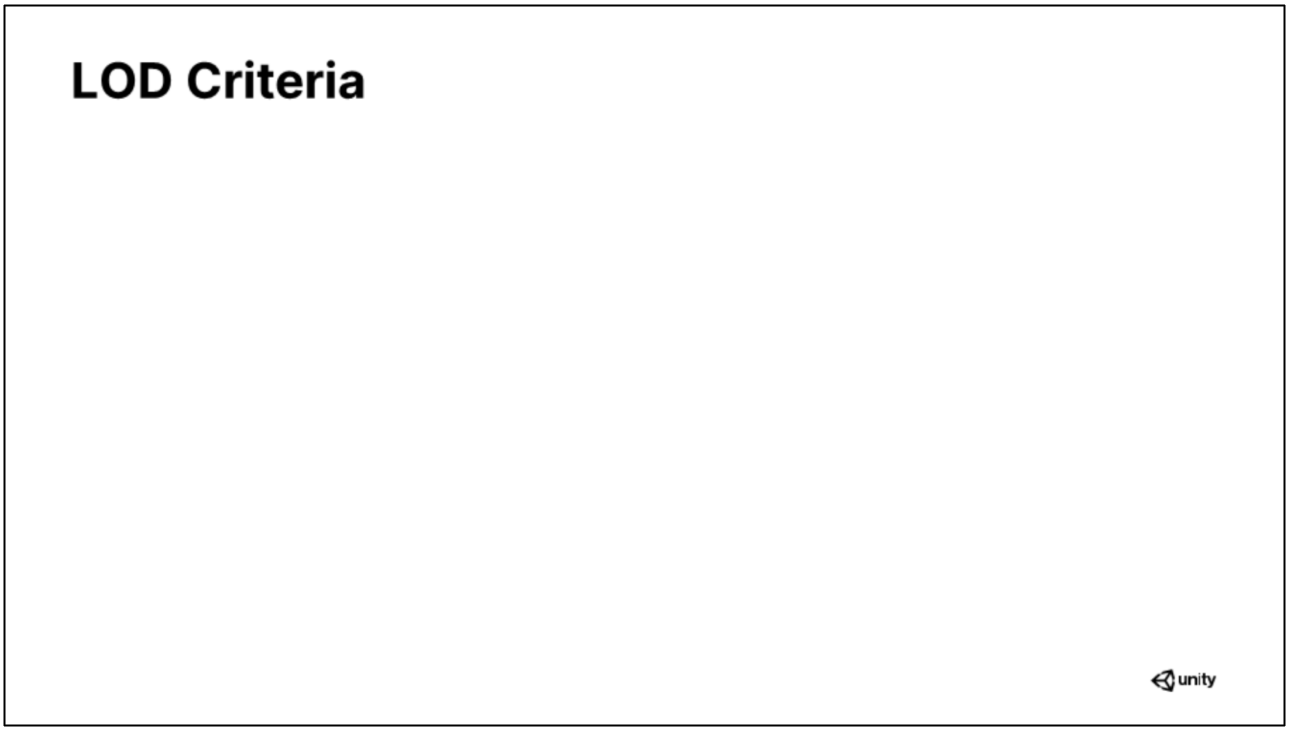
20 © 2019 SIGGRAPH. ALL RIGHTS RESERVED.

thrive SIGGRAPH2019 LOS ANGELES • 28 JULY - 1 AUGUST

unity

It's worth mentioning that combining Jonathan's work on GPU binary trees with Mesh Shaders was mentioned two years ago by Yury Uralsky during his siggraph presentation on this new API,

So it's definitely a very interesting avenue for future research.



LOD Criteria

Finally, the last thing I’ll talk about is the level of detail criteria that we use in our demo.

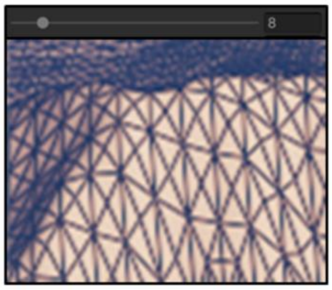
Anything can be used here and it’s pretty easy to plug in whichever criteria you want in the subdivision update kernel.

In our case we use two separate ones :

LOD Criteria

Target edge length

- Measure longest edge length in pixels
- User criteria



First we read the current triangle’s vertices and read the heightmap to get final displaced triangle.

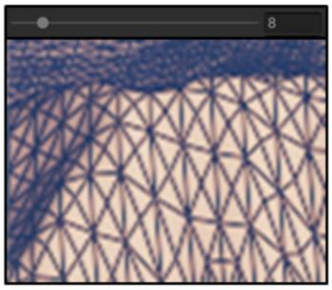
Then we measure the length of the longest edge in pixels on the screen.

And we compare this length against a set target length to decide whether to split or merge the triangle.

LOD Criteria

Target edge length

— Measure longest edge length in pixels
— User criteria

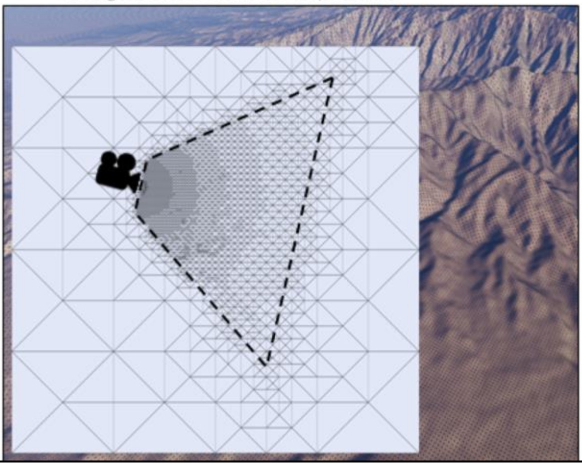


Frustum Culling

— Compute axis-aligned triangle bounding box

```
float3 bmin = min(min(faceVertices[0], faceVertices[1]), faceVertices[2]);
float3 bmax = max(max(faceVertices[0], faceVertices[1]), faceVertices[2]);
```

— Test against camera frustum planes
— Only split if inside frustum
— Merge outside as much as possible



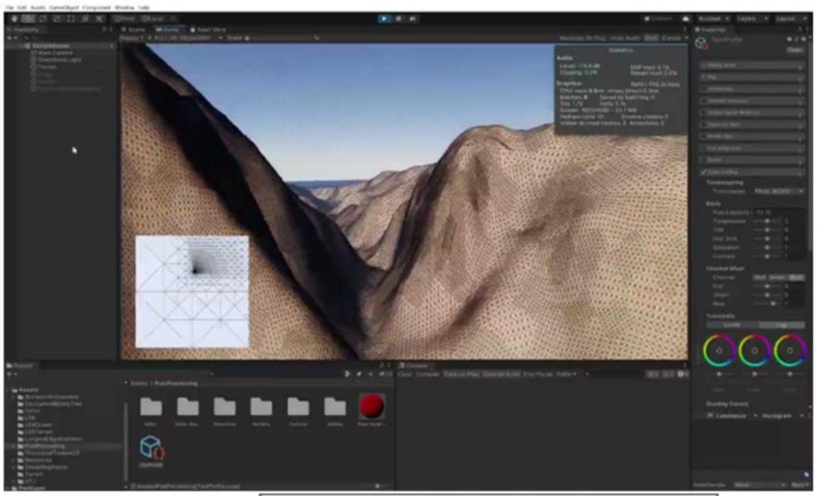
Our second level of detail criteria is frustum culling.

To avoid drawing many small triangles that are outside of the camera view,

We test the axis aligned bounding box of the triangle against the camera’s frustum.

If they don’t intersect, the triangle is not visible, and we don’t allow it to split.

Results



- 51*51 km terrain
- Depth = 27
- => max resolution ≈ 6m
  (0.1m with level 6 meshlet)
- => VRAM: 64mb
- Sum-reduction is bottleneck

	CBT update (D = 27)		
Kernel	dispatch	subdivision	sum-reduction
Timing (ms)	0.017	0.035	1.484

unity

So with all this implemented, this is the result we got for the paper's publication.

We used a real world heightmap at its real scale, here we see the Kauai island at 51 by 51 kilometers.

And setting our maximum CBT depth to 27, we get the following numbers.

Memory usage is 64mb. It's equal to two to the power of depth minus one, so it doubles with each new depth level.

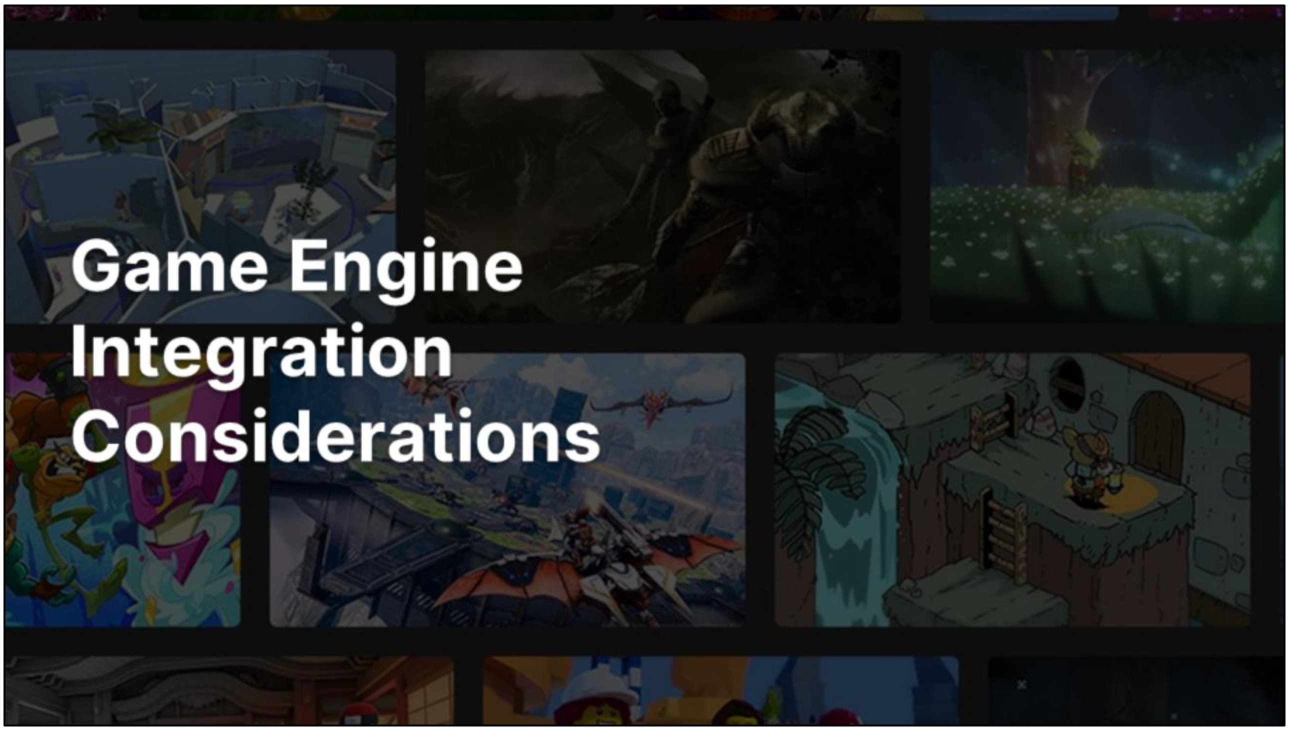
The maximum resolution we get on the ground for the subdivision is about 6 meters. This isn't detailed enough for a game, but it increases dramatically when replacing triangle leaf nodes with meshlets, which I will let my colleague Xiaoling explain right after.

Finally, we can clearly see that the sum-reduction quickly becomes the bottleneck with deep subdivisions, while the subdivision itself has a negligible cost. This is expected because of the nature of these reduction operations.

Nethertheless, this is really good for a demo of a gpu driven, completely adaptive mesh tessellation, so we shipped the paper with this video.

And then we started transferring this from our research team to Xiaoling's team, with an immediate focus on optimization for their needs.

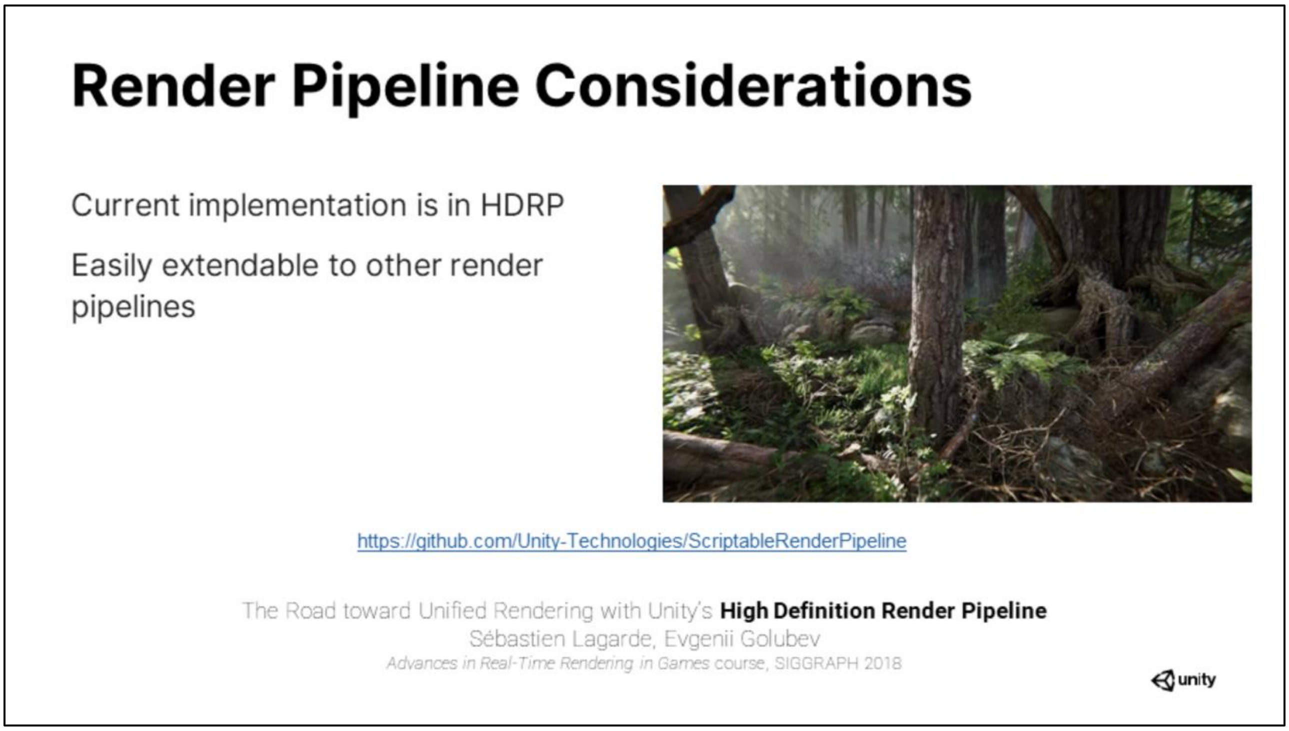
And I’ll let Xiaoling take over for this.



Game Engine Integration Considerations

Thank you Thomas.

In this half of the presentation, we will talk about how to integrate this technique into a production game engine and some of the practical considerations we've learned through this process. In our case, we have integrated this method into the Unity engine, which is a real-time engine designed for games and variety of other applications, such as film, architecture and construction, automotive, design and so much more.



Render Pipeline Considerations

Current implementation is in HDRP

Easily extendable to other render pipelines

https://github.com/Unity-Technologies/ScriptableRenderPipeline

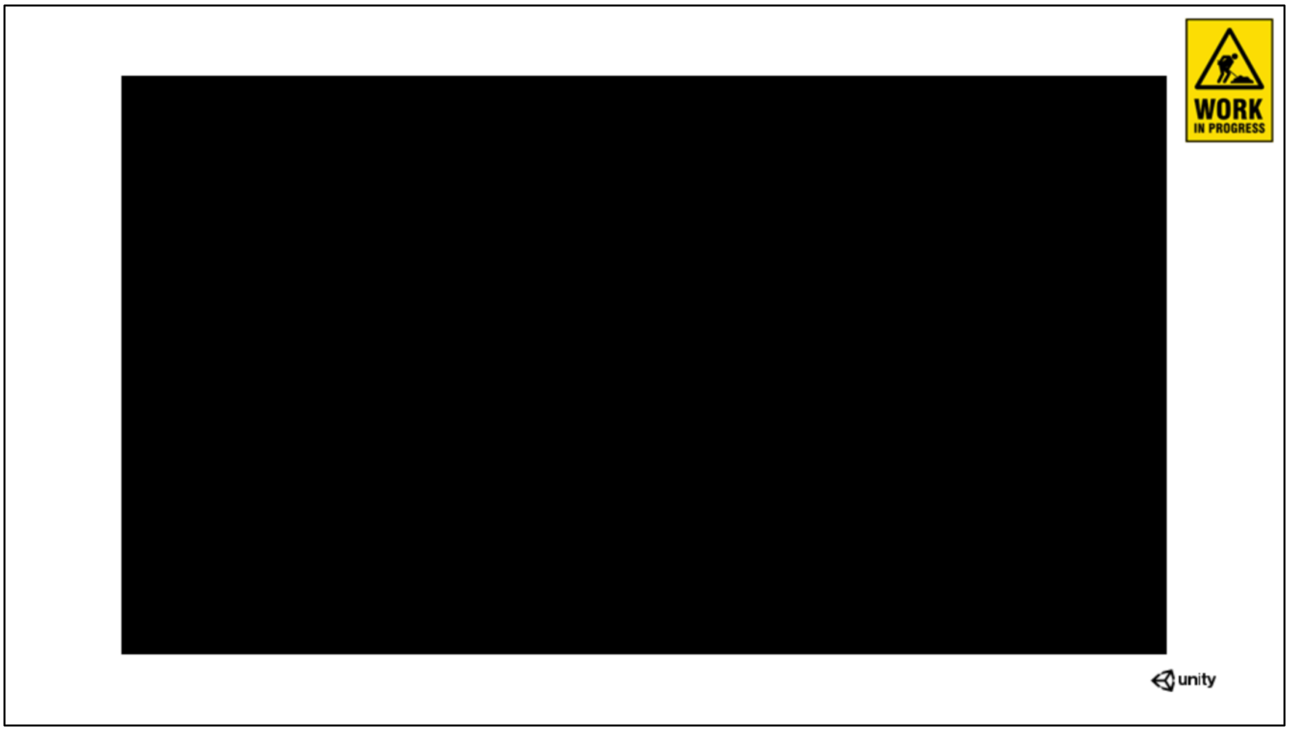
The Road toward Unified Rendering with Unity's **High Definition Render Pipeline**
Sébastien Lagarde, Evgenii Golubev
Advances in Real-Time Rendering in Games course, SIGGRAPH 2018

Unity enables you to customize the render pipeline for your specific project needs. For our prototype implementation we use Unity's own High-Definition Render Pipeline.

If you are interested in additional details about it, feel free to check out our github repository or the slides of the previous talk.

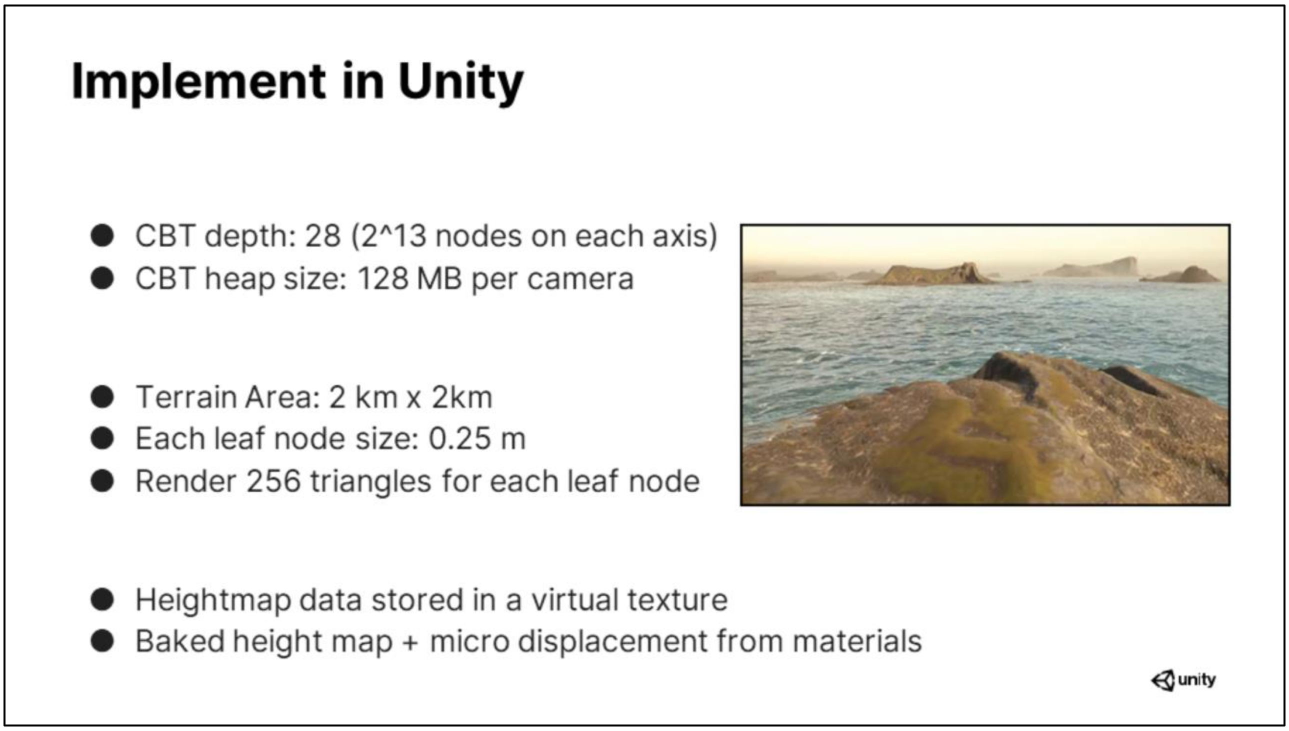
However our implementation doesn't really rely on HDRP features. It should be easily extendable to other render pipelines as long as they have compute shader and indirect draw capabilities.

https://github.com/Unity-Technologies/ScriptableRenderPipeline
http://advances.realtimerendering.com/s2018/index.htm



Before I dive into all the details, let’s play the video again that shows our current result.

(Plays the video)



Implement in Unity

- CBT depth: 28 (2^13 nodes on each axis)
- CBT heap size: 128 MB per camera

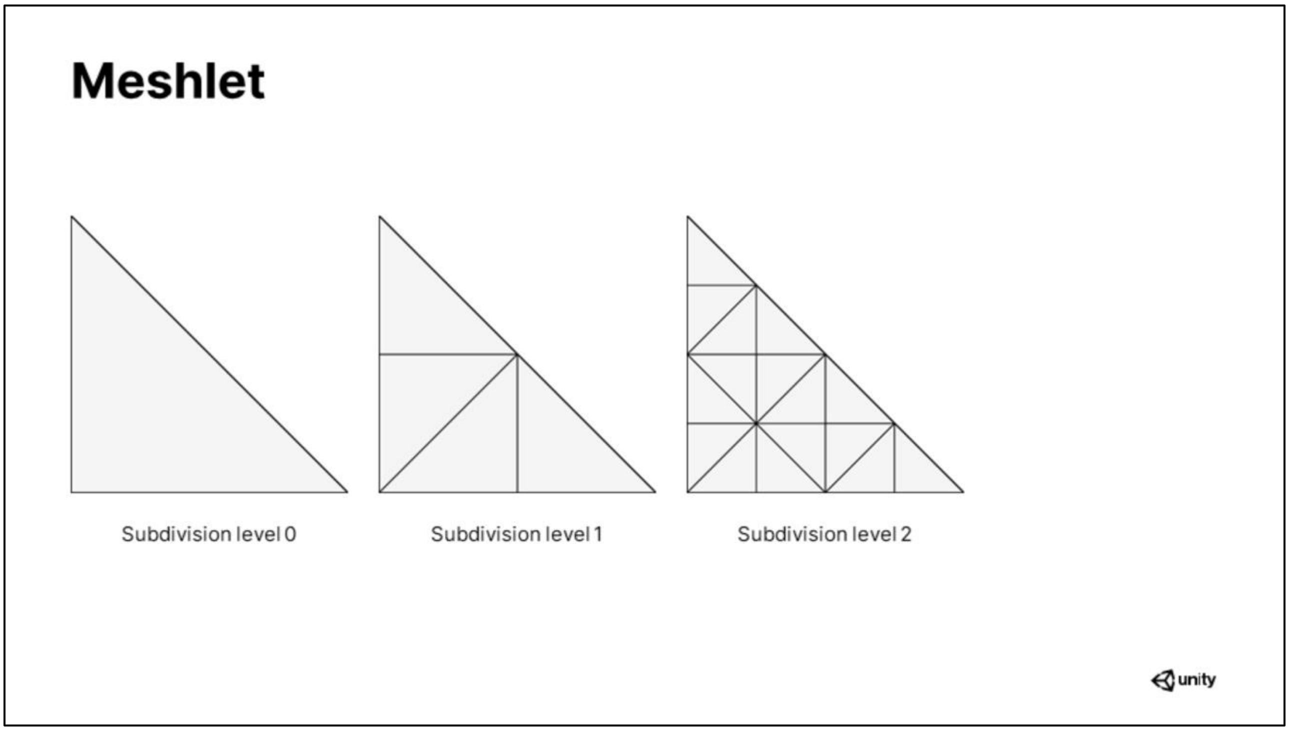
- Terrain Area: 2 km x 2km
- Each leaf node size: 0.25 m
- Render 256 triangles for each leaf node

- Heightmap data stored in a virtual texture
- Baked height map + micro displacement from materials

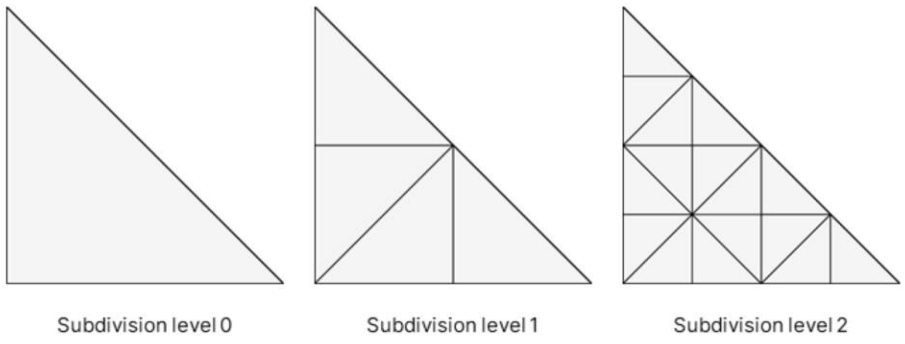
The terrain you have just seen in the video uses a CBT with a maximum depth of 28. This means that over the entire 2 by 2 kilometers area, each leaf node represents an area of a quarter by a quarter meter.

This is less than the resolution we want to achieve. So instead of rendering just one triangle for every leaf node, we further render 256 triangles, which we refer to as meshlets, to reach the resolution we wanted. I'll explain in detail in the next slide, but eventually in the video the terrain triangles can be as small as 1.5 centimeters.

It is also worth mentioning that we put the heightmap data in a virtual texture so that the draw call has access to the entire terrain heightmap. The heightmap data has a resolution of an eighth of a meter. On top of that we blend micro-displacement textures from the terrain materials to provide details as fine as 1.5 centimeter resolution.



Meshlet



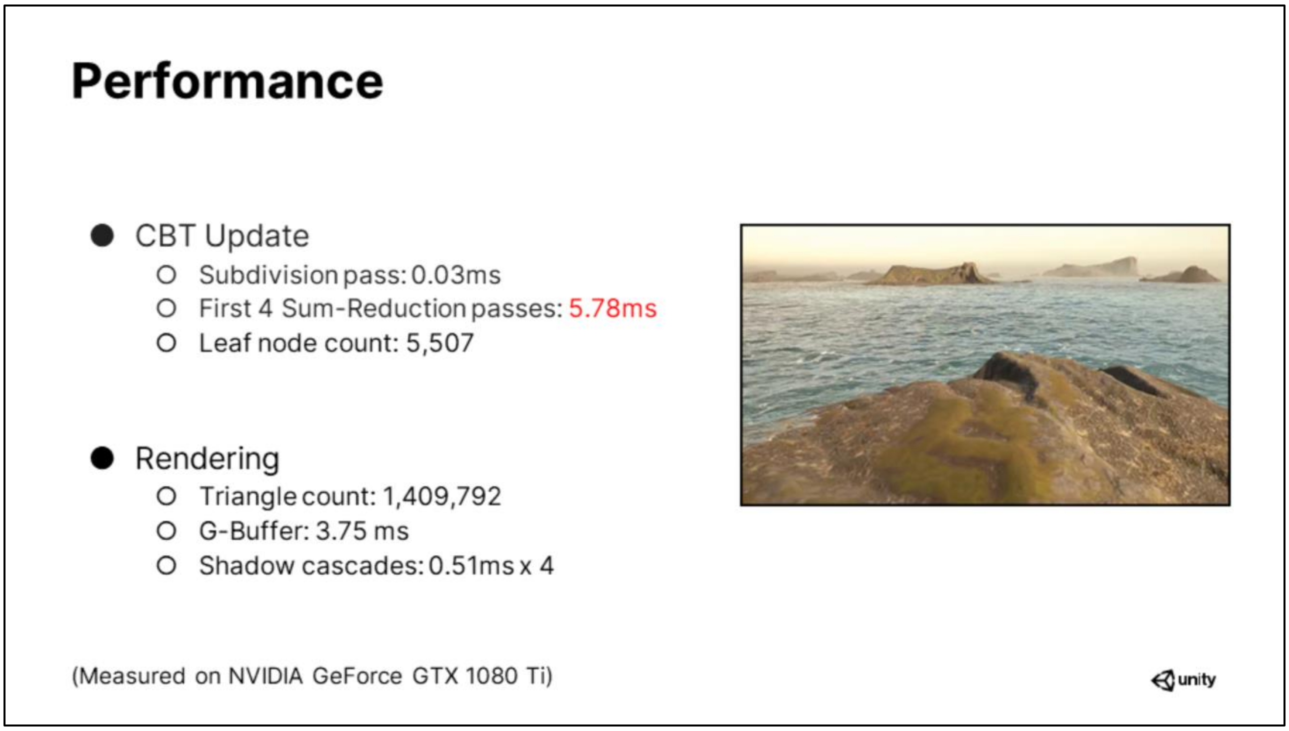
Now let's get back to the details of the technique called meshlet.

To achieve higher triangle density under the current limitations, one quick way is to render more triangles for each leaf node. Jonathan implemented this meshlet idea in his original paper.

This illustration shows what these pre-subdivided triangles look like at level 0, 1 and 2. The triangles are constructed exactly the same way as regular CBT subdivision, making them visually seamless when incorporated into the terrain CBT geometry.

We construct the vertex and index buffers on the CPU and simply send them to the Unity indirect draw as the mesh argument.

This approach basically gives us more resolution over the existing CBT for free. The downside is that the meshlets are not adaptive, so the final mesh is not as adaptive as what the CBT can produce.



Performance

- CBT Update
  - Subdivision pass: 0.03ms
  - First 4 Sum-Reduction passes: 5.78ms
  - Leaf node count: 5,507
- Rendering
  - Triangle count: 1,409,792
  - G-Buffer: 3.75 ms
  - Shadow cascades: 0.51ms x 4

(Measured on NVIDIA GeForce GTX 1080 Ti)

unity

With such huge amount of subdivision data, the performance soon became the number one focus in our implementation. Please notice that we've only done our profiling on PC environment as we are in early prototyping stage.

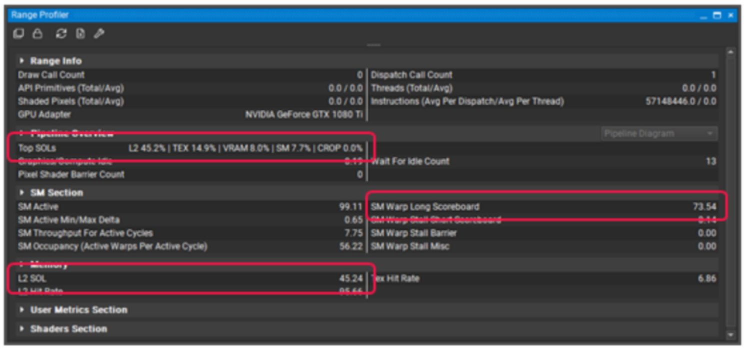
The subdivision pass is pretty fast because we don't need lots of leaf nodes to get to a satisfying subdivision of the terrain geometry. In the demo video most of the time there are fewer than 10 thousand leaf nodes.

The sum-reduction passes however are the slowest among all the CBT update passes. On a middle tier hardware we measured 5.78 milliseconds for the first 4 passes, which are the most expensive.

You may notice that the rendering time of the terrain geometry isn't ideal either. I'll list some of the ideas later in the talk that the team will be experimenting in the near future.

For now we'll take a deep look at what's causing the slowness in the sum-reduction passes.

Optimizations: Sum-Reduction Passes



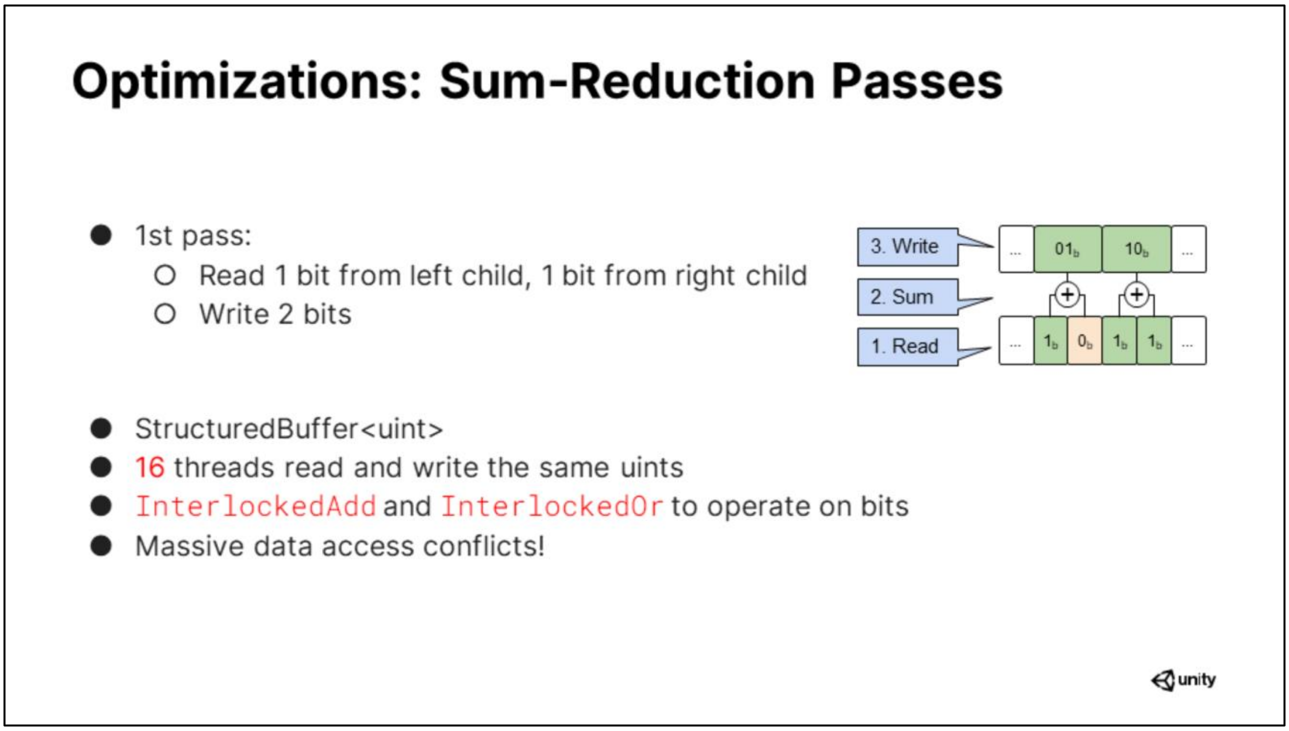
```
Range Profiler

Range Info
Draw Call Count                                  0 | Dispatch Call Count                                             1
API Primitives (Total/Avg)               0.0 / 0.0 | Threads (Total/Avg)                                     0.0 / 0.0
Shaded Pixels (Total/Avg)                0.0 / 0.0 | Instructions (Avg Per Dispatch/Avg Per Thread)  57148446.0 / 0.0
GPU Adapter              NVIDIA GeForce GTX 1080 Ti |
Pipeline Overview                                                                                 Pipeline Diagram
Top SOLs  L2 45.2% | TEX 14.9% | VRAM 8.0% | SM 7.7% | CROP 0.0% |
Graphics/Compute Idle                         0.19 | Wait For Idle Count                                            13
Pixel Shader Barrier Count                       0 |
SM Section
SM Active                                    99.11 | SM Warp Long Scoreboard                                     73.54
SM Active Min/Max Delta                       0.65 | SM Warp Stall Short Scoreboard                               0.14
SM Throughput For Active Cycles               7.75 | SM Warp Stall Barrier                                        0.00
SM Occupancy (Active Warps Per Active Cycle) 56.22 | SM Warp Stall Misc                                           0.00
Memory
L2 SOL                                       45.24 | Tex Hit Rate                                                 6.86
L2 Hit Rate                                  95.66 |
User Metrics Section
Shaders Section
```

- Cause: Memory access
  - Top SOL falls in L2 and is not efficient
  - Hit rate is pretty good
  - High "Long Scoreboard" : Stall due to memory access

Let's look at this profiling results for the first sum-reduction pass, the most expensive one.

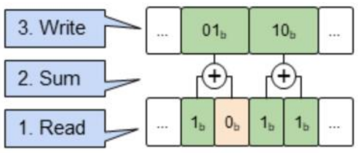
We quickly noticed that the Top "speed of light" is L2 memory, followed by the texture unit, and the efficiency hasn't even reached 50%. Specifically NSight shows the warp stalls mostly waiting for "Long Scoreboard". From NVidia's website we know it means the GPU is waiting for buffer access operations to finish.

Improving the way we read and write the CBT buffer in the sum-reduction passes became our main goal for the optimization.



Optimizations: Sum-Reduction Passes

- 1st pass:
  - Read 1 bit from left child, 1 bit from right child
  - Write 2 bits



- StructuredBuffer<uint>
- 16 threads read and write the same uints
- `InterlockedAdd` and `InterlockedOr` to operate on bits
- Massive data access conflicts!

unity

Now let's take a deeper look at the code. What exactly does the compute shader do in this pass?

The first sum-reduction pass by its definition counts the two bits from the left and right child nodes of each level-2 parent node, sums them up to either 0, 1, or 2, and writes this 2 bit integer sequentially back in the buffer.

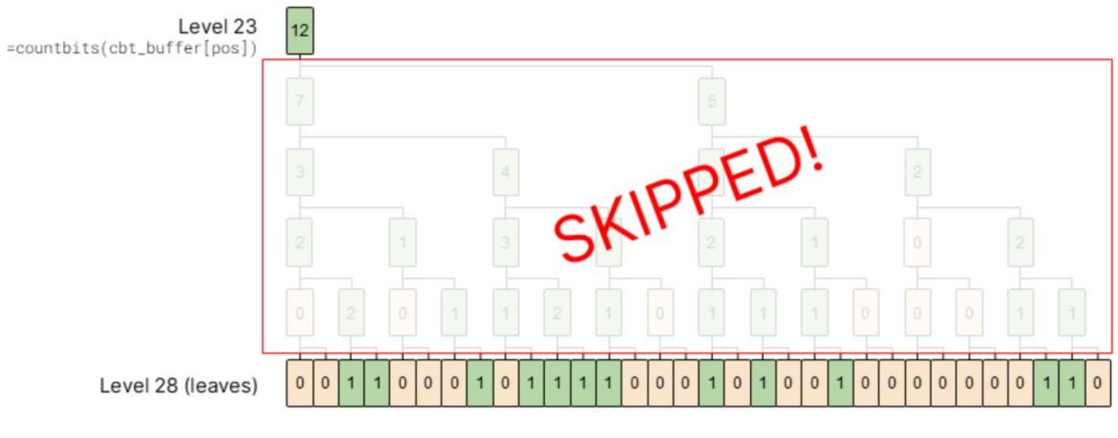
The subsequent passes do exactly the same, only they operate on one bit wider data at each level from bottom up.

So, first of all, there are simply lots of data to write. For a 28 depth CBT we have 2 to the 27th level-2 parent nodes, which is about 127 million such nodes. Dispatching 1 thread for 1 such node is not only unviable, but also feeds way too little workload to the GPU to consume efficiently.

Secondly, our buffer for the CBT data is 4 bytes wide. It means that if we did naively implement this code, dispatching 1 thread for 1 level-2 parent node, we would end up having 16 threads write to the same uint in the buffer. And because of that, we have to use atomic operations, which are InterlockedAdd and InterlockedOr intrinsics, to operate on the bits to make sure the concurrent write is done correctly. This for sure leads to massive data access conflicts between threads.

Let's see how we can tackle these two problems.

Optimizations: Sum-Reduction Passes

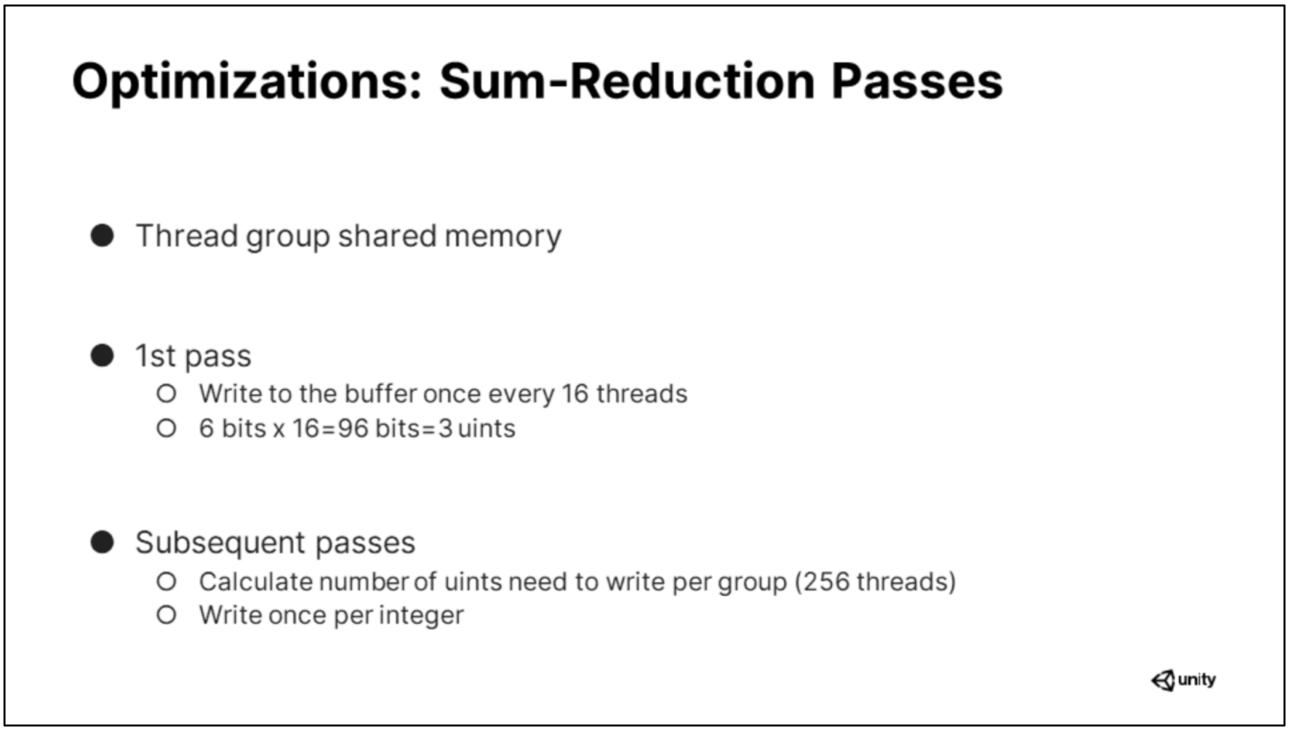


So writing 2 bits per thread is not optimal. How about combining 32 of them in one go?

The fact that leaf nodes are 1 bit wide gives us the opportunity to treat 32 of them as a 32 bit wide integer, and summing the 1s basically translates to a call to the countbits intrinsic. This way we quickly get to the parent nodes of 6th level from the bottom.

As a result we completely skipped processing the sum-reduction info for the levels 2 to 5 from the bottom. (Click)

They are not necessary anyway because for tracing any node deeper than the max depth level minus 5 correctly we can read 32 bits from the bottom level and trace the bits within the integer directly.



Optimizations: Sum-Reduction Passes

- Thread group shared memory
- 1st pass
  - Write to the buffer once every 16 threads
  - 6 bits x 16=96 bits=3 uints
- Subsequent passes
  - Calculate number of uints need to write per group (256 threads)
  - Write once per integer

Still writing the 6th level node from the bottom is not memory friendly because each node is 6 bit wide and the memory access conflicts still exist.

To solve that, we relied on shared memory. Specifically, we allocate an array of integers in the thread group shared memory. Each element in the array is dedicated to one thread, holding the 6 bit wide value that would need to be written.

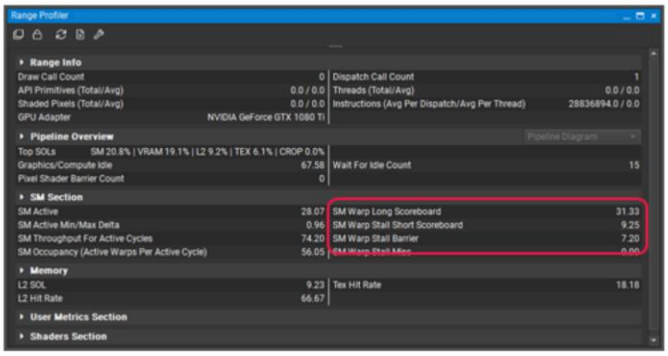
For every 16 threads we take 16 such 6 bit wide integers out of the array, combine them into 3 full integers and write out to buffer with regular non-atomic writes, because we don't have concurrent writes from different threads any more. Problem is solved for the first pass!

For subsequent sum-reduction passes we employed a similar optimization: first, calculate the number of integers we need to write every 256 threads, which is our group size of choice. Then after all the threads have written to their allocated slot in the shared memory array, we again combine the values into full integers, and use the first few threads to write one integer per thread to the buffer.

Notice that we only cache the memory writes to eliminate the write conflicts and atomic operations. It's not beneficial to cache the buffer reads according to our profiling results, even though there are many redundant reads. We think the L1 and L2 cache is working well for this type of access pattern.

Results

- Before: 5.78ms
- After: 0.40ms
- 14.5x faster



```
Range Profiler
▸ Range Info
Draw Call Count                                   0 | Dispatch Call Count                                   1
API Primitives (Total/Avg)                0.0 / 0.0 | Threads (Total/Avg)                           0.0 / 0.0
Shaded Pixels (Total/Avg)                 0.0 / 0.0 | Instructions (Avg Per Dispatch/Avg Per Thread)  28836894.0 / 0.0
GPU Adapter              NVIDIA GeForce GTX 1080 Ti |
▸ Pipeline Overview                                                                  Pipeline Diagram
Top SOLs  SM 20.8% | VRAM 19.1% | L2 9.2% | TEX 6.1% | CROP 0.0% |
Graphics/Compute Idle                         67.58 | Wait For Idle Count                                  15
Pixel Shader Barrier Count                        0 |
▸ SM Section
SM Active                                     28.07 | SM Warp Long Scoreboard                            31.33
SM Active Min/Max Delta                        0.96 | SM Warp Stall Short Scoreboard                      9.25
SM Throughput For Active Cycles               74.20 | SM Warp Stall Barrier                               7.20
SM Occupancy (Active Warps Per Active Cycle)  56.05 | SM Warp Stall Misc                                  0.00
▸ Memory
L2 SOL                                         9.23 | Tex Hit Rate                                       18.18
L2 Hit Rate                                   66.67 |
▸ User Metrics Section
▸ Shaders Section
```

Bonus

Avoid using buffer.GetDimensions()

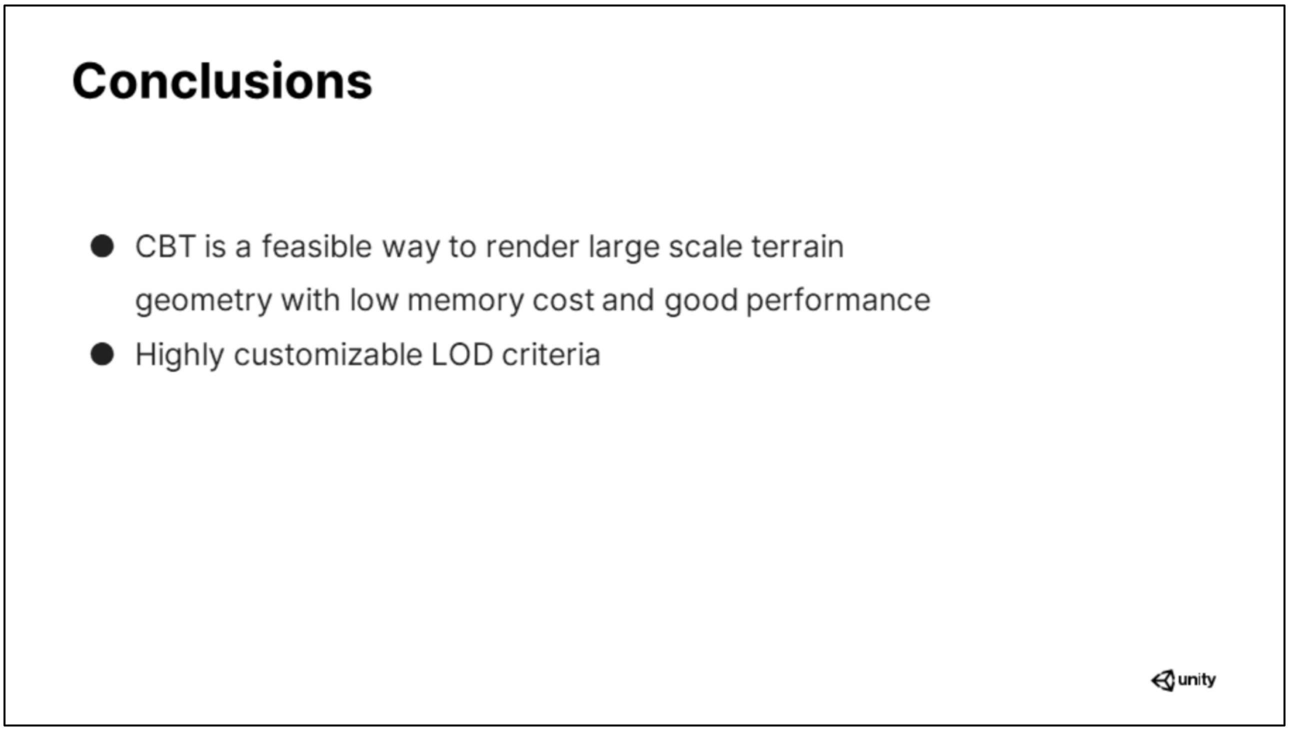
The resulting performances are pretty encouraging. We measured a 14.5 times boost on the same hardware. Now it takes less than half a millisecond on a 4 year old graphics card, making the technique much more viable.

You can see in the screenshot that the Long Scoreboard dependency is much reduced. Instead the Short scoreboard increases as it represents the stall due to the shared memory operations, which is expected.

There is also a tiny increase in Stall Barrier dependency, because the thread group needs to be synchronized via a barrier before reading the shared memory array.

(Click)

One fun thing we noticed is that using GetDimensions on the buffer object is actually very slow. We used it to get the total depth of the CBT. Replacing it with a uniform variable we get roughly 1.6x faster, which was a happy surprise to the team.



Conclusions

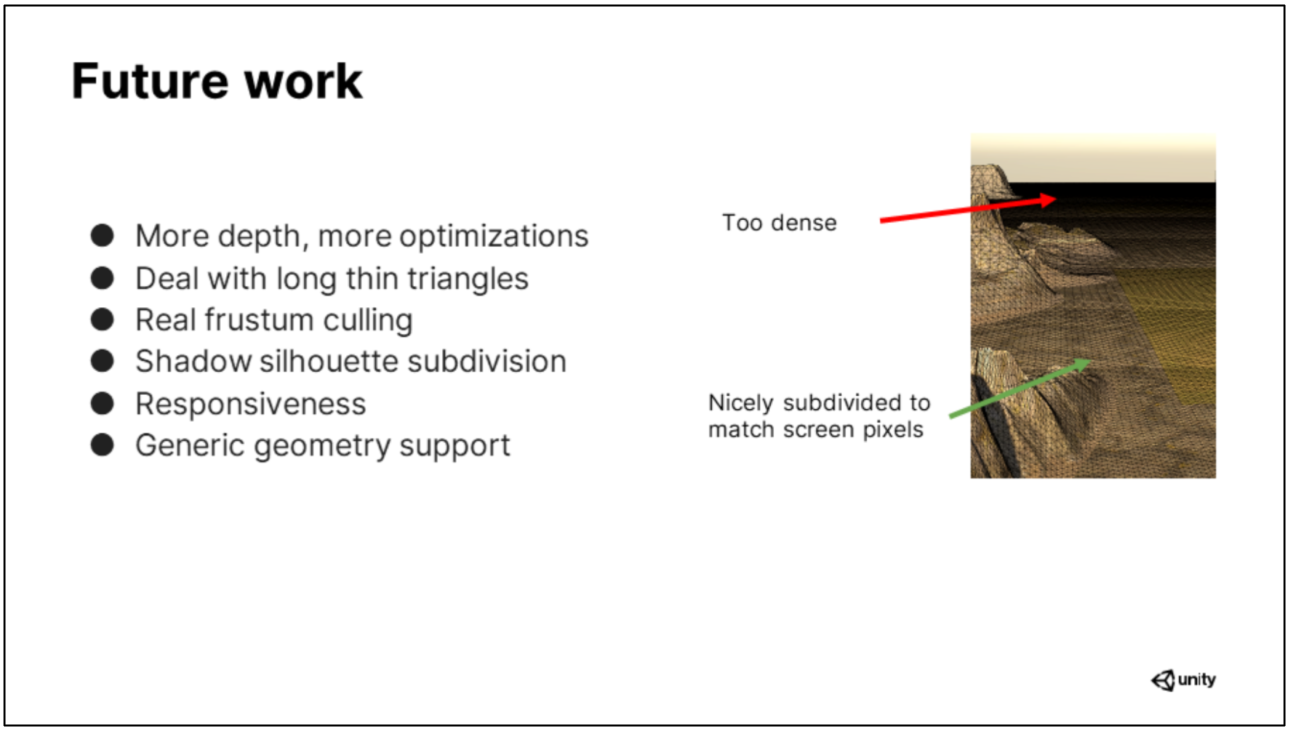
- CBT is a feasible way to render large scale terrain geometry with low memory cost and good performance
- Highly customizable LOD criteria

unity

Conclusions!

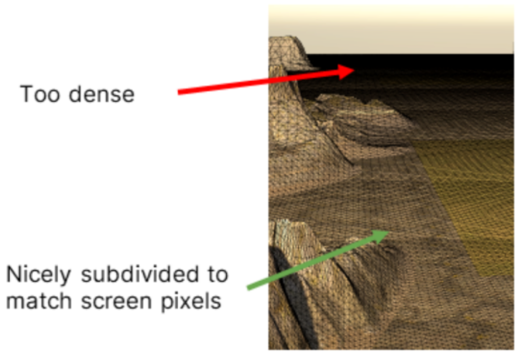
With CBT subdivision the team is able to achieve the size and resolution we want for the terrain with relatively low memory cost and good runtime performance.

The team especially likes that the algorithm allows customizing the LOD criteria arbitrarily to meet the project needs.



Future work

- More depth, more optimizations
- Deal with long thin triangles
- Real frustum culling
- Shadow silhouette subdivision
- Responsiveness
- Generic geometry support



The team is eager to explore the technique more in the near future. Here lists a few things that we will be focusing on.

First the team is aiming for getting more depth in the tree hierarchy, so we rely less on the meshlet to reach our target triangle density. In order to do that we need to optimize the system more because each additional depth would double the memory and time cost.

We want to find a way of avoiding the subdivision of flat triangles. (Click) We will extend our LOD criteria to account for terrain curvature and screen-space projected area to help mitigate the situation.

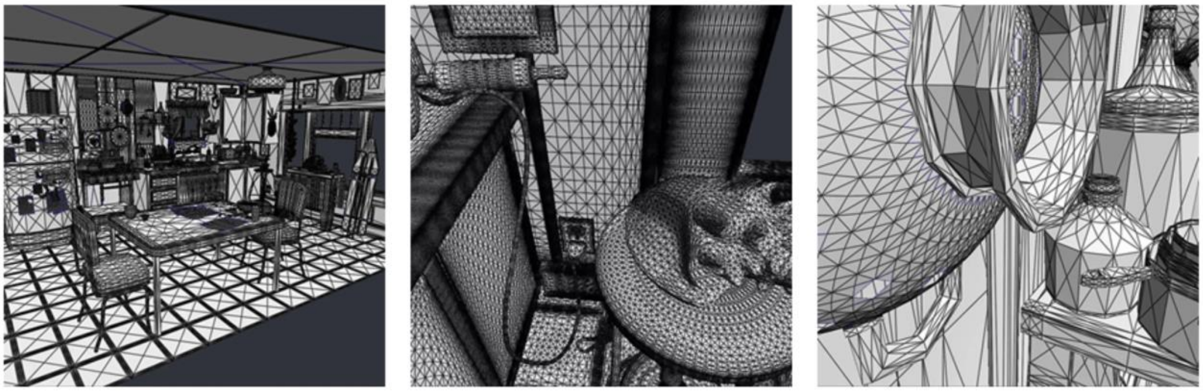
We also need to implement real view frustum culling for the geometry especially for rendering to cascaded shadow maps. We believe it can be done relatively easily by leveraging the inherent tree hierarchy.

Again for shadow map rendering we want to customize the LOD criteria so we can subdivide more at silhouette triangles for better preservation of the silhouette shape. Again, having a curvature information should help here.

We want to improve the overall responsiveness of the system when the camera moves due to the limitation of doing only one level of subdivision every frame.

Additionally our lab team continues their exploration of the idea and is looking to extend the application to generic geometries other than terrain. (next slide)

Future work



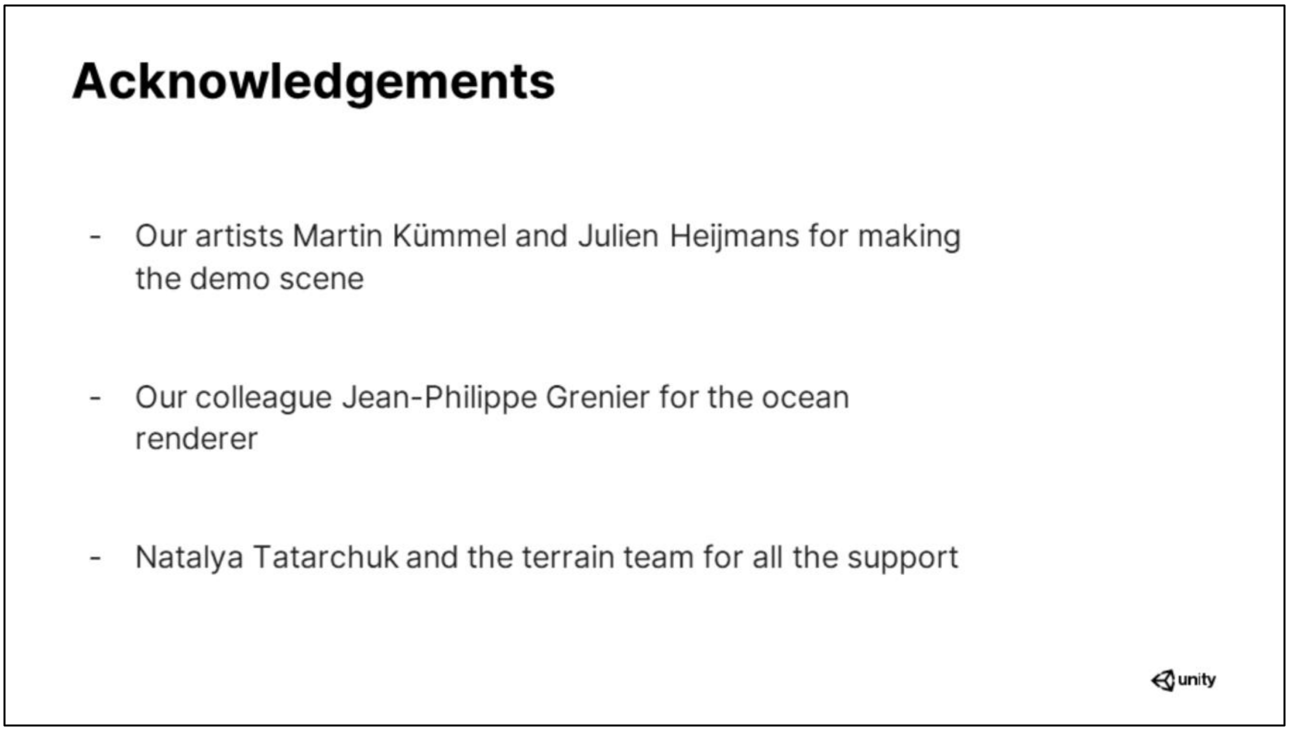
A Halfedge Refinement Rule for Parallel Catmull-Clark Subdivision
Jonathan Dupuy, Kenneth Vanhoey, HPG 2021

For example, an advantage of CBTs is that they can also allow tessellating arbitrary subdivision surfaces.

Here's a work in progress demonstration.

It shows Catmull-Clark subdivision surfaces adaptively tessellated on the GPU, in real-time in Unity, thanks to CBTs.

You can check out their latest HPG talk for more information on this.



Acknowledgements

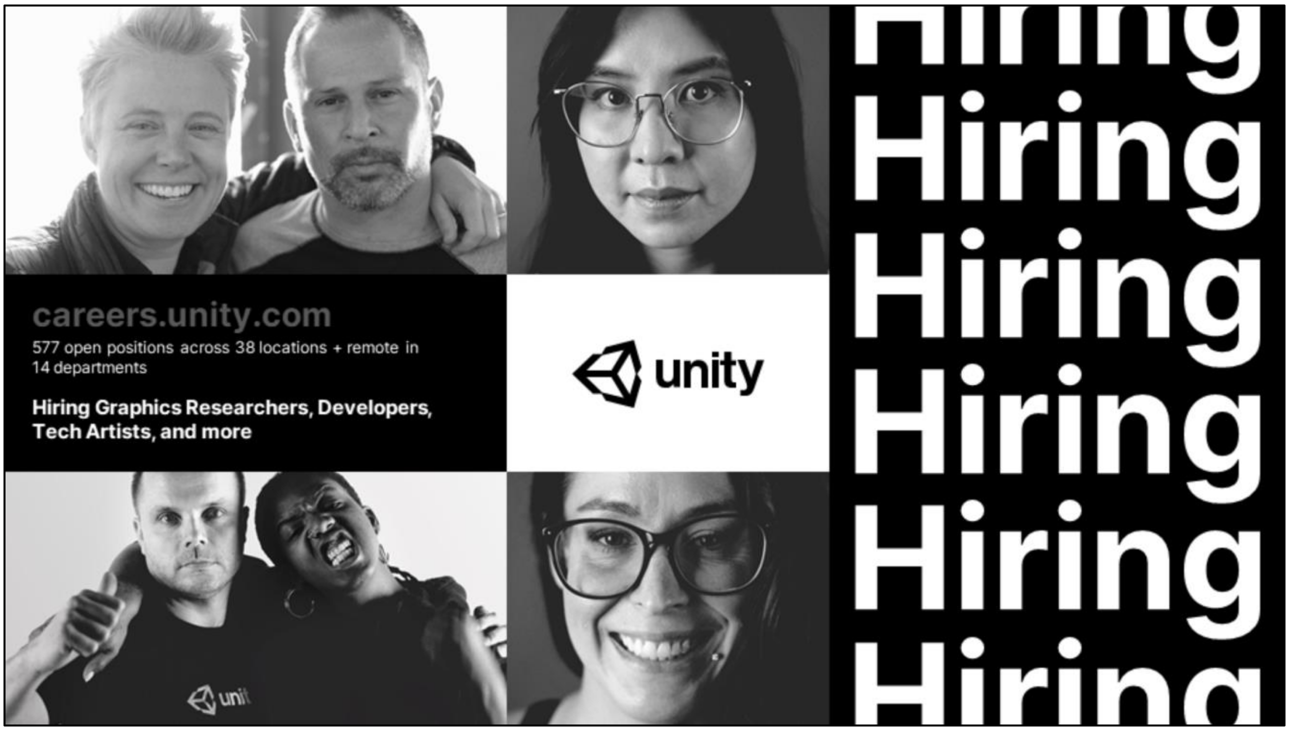
- Our artists Martin Kümmel and Julien Heijmans for making the demo scene
- Our colleague Jean-Philippe Grenier for the ocean renderer
- Natalya Tatarchuk and the terrain team for all the support

Okay this is it!

I want to thank our artist colleagues Martin and Julien for making the demo scene in less than 1 week.

Our colleague Jean-Philippe for the beautiful ocean rendering.

And Natalya and my team for all your support.


careers.unity.com
577 open positions across 38 locations + remote in 14 departments
Hiring Graphics Researchers, Developers, Tech Artists, and more
unity
Hiring
Hiring
Hiring
Hiring
Hiring
Hiring

Last but not least, Unity continues hiring graphics researchers, developers, tech artists and more world wide. Please visit careers dot unity dot com if you are interested in joining us.

Thank you!

Thank you for listening! Good bye!