gcap games connect asia pacific

The Tech Art Journey of creating and optimizing an asset pipeline

#GCAP24

gcap games connect asia pacific

OI,
tudo bem?

#GCAP24

gcap games connect asia pacific

Who am I?

Tech Art Lead @ SMG Studio

Previously:
Lead Gameplay Programmer
Data Scientist
Naval Engineer

But also:
Maths Geek
Coffee + Cake Lover
SoulsBorneKiroRing fan

#GCAP24

gcap games connect asia pacific

What will happen here?

This will be fast

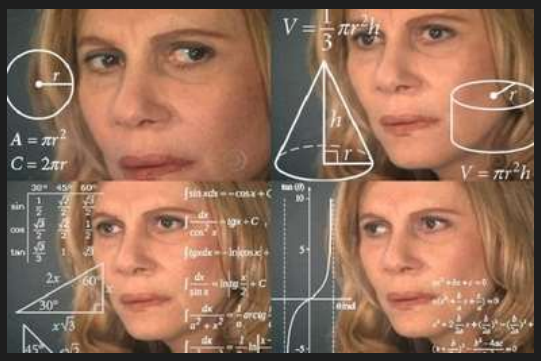
#GCAP24

This talk will be fast as there is a lot of content, but not a lot of time

gcap games connect asia pacific

What will happen here?

This will be fast
This will get technical



#GCAP24

This talk will have technical elements, so be prepared for talking about precision and bytes

gcap games connect asia pacific

What will happen here?

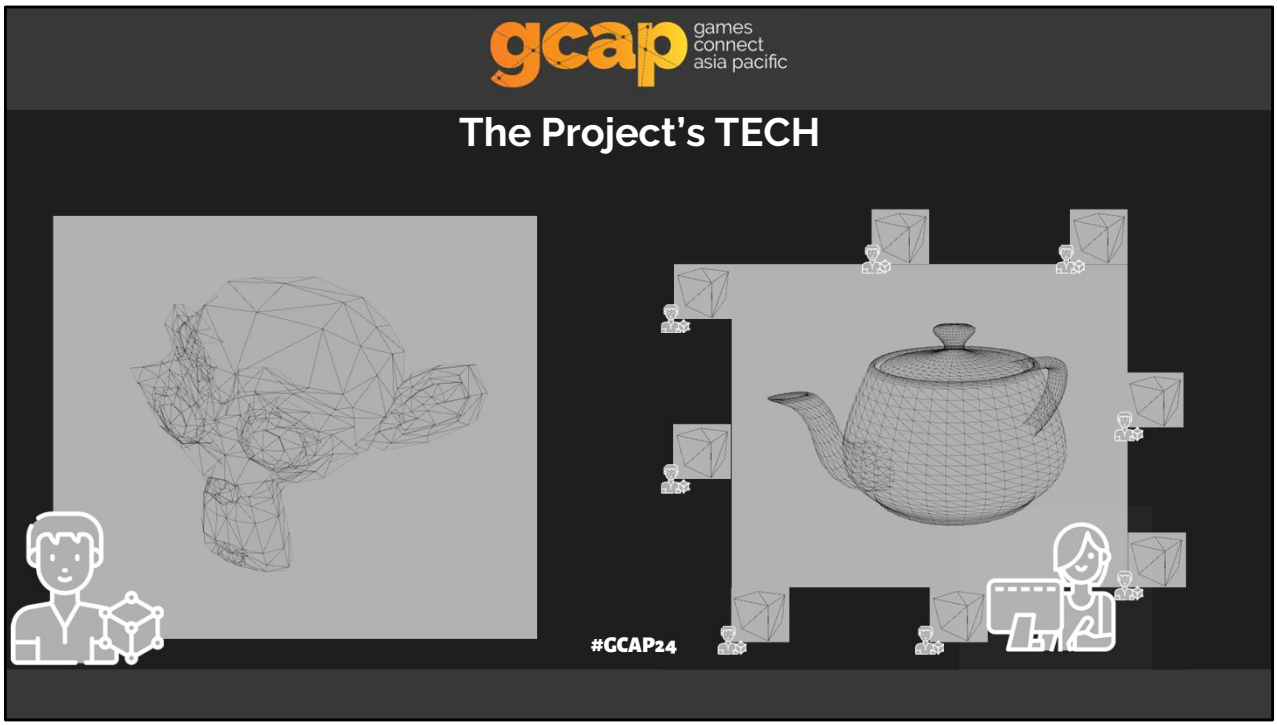
This will be fast
This will get technical
This deck will be available online

#GCAP24

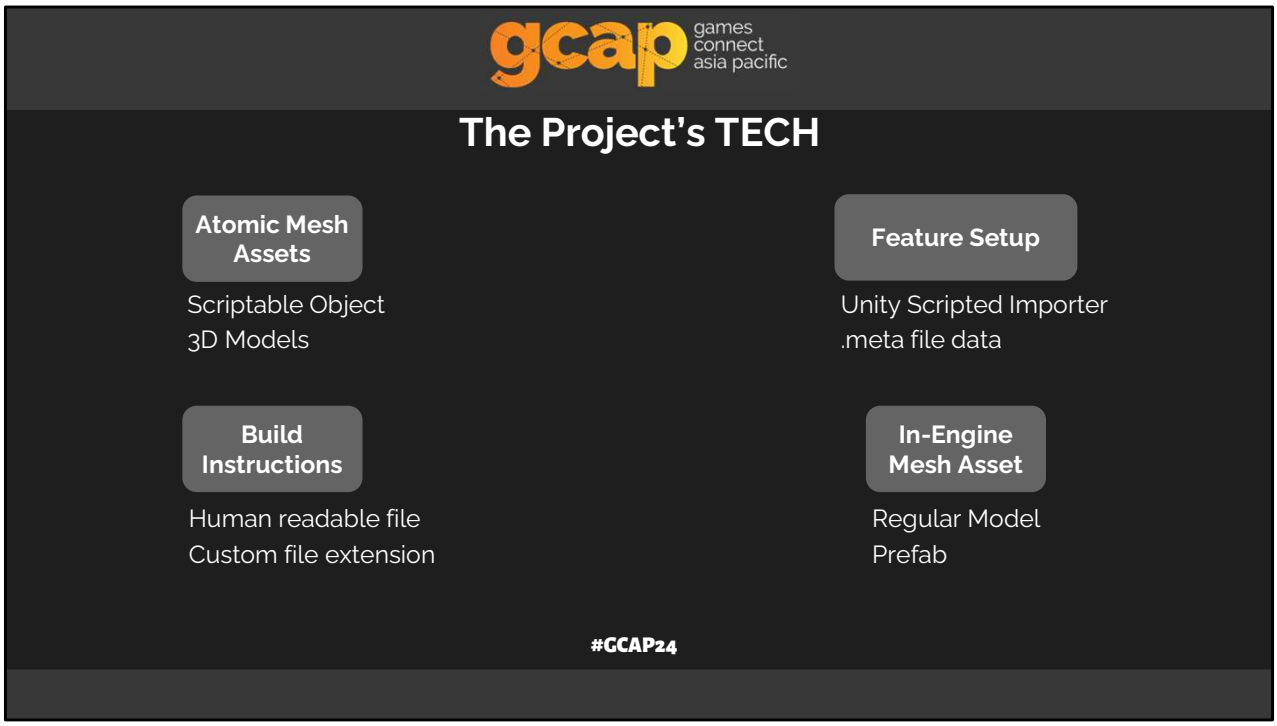
But fret not because this deck will be online and you can get a link at the end! With all that said, get yourself comfortable, keep your hands inside the vehicle at all times and off we go!

This talk will cover the technology implemented on SMG's largest game, a multiplatform party game…

That is still a secret.
The project was intended to be both announced and released by GCAP 2024, but due to release window complications we've delayed until 2025

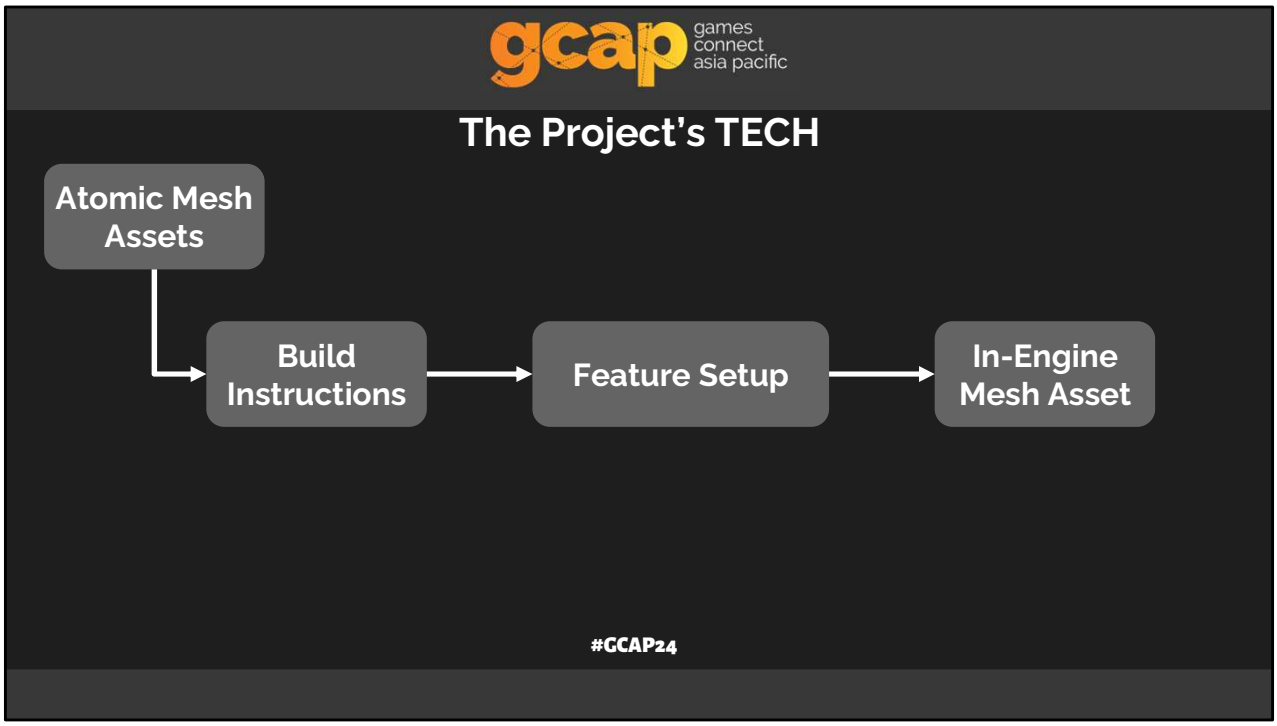


So let's talk about the tech used in the project instead!
While most projects have a team of 3D artists that models entire assets
On our project we took a different route:
We started by building a library of "Atomic mesh assets" these can be anything, from a small wall piece, to a door, or anything!
Then we have a design team that create build instructions for each of our scenes and in some cases even characters!
The Tech art team ingest these instructions into our custom pipeline setting up culling and features such as compression or lightmap unwrapping
Finally the pipeline outputs meshes in-engine that can be used normally by anyone in the team.

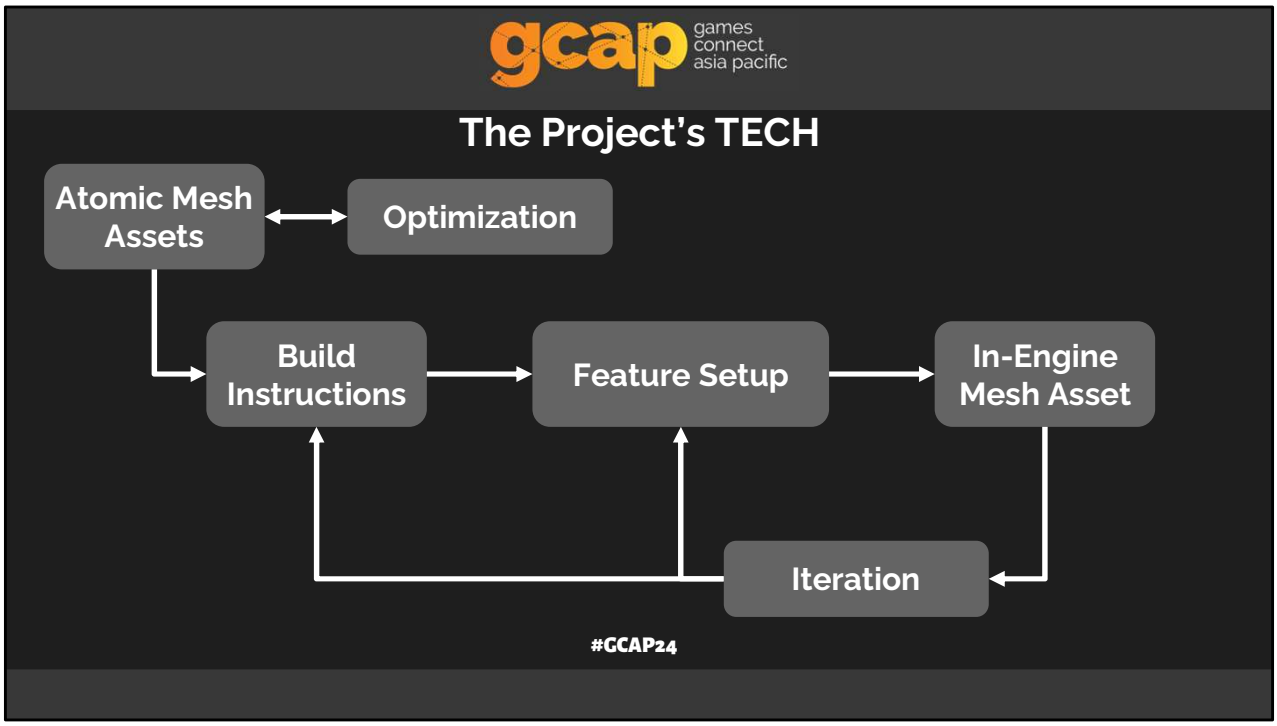


Some details on the different parts:

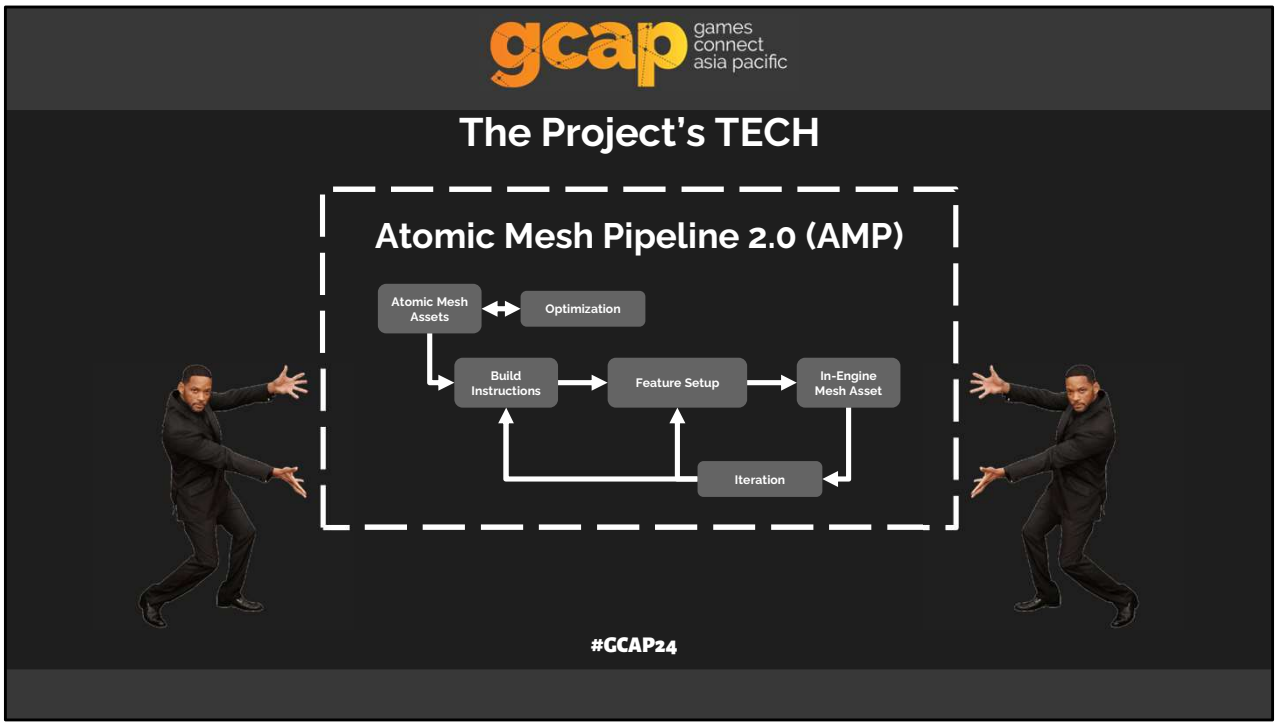
- Atomic Mesh Assets: Scriptable Objects that have 3D models assigned to them
- Build Instructions: human readable files saved with a custom file extension
- Feature Setup: Unity Scripted Importer which saves data to the .meta file (https://docs.unity3d.com/Manual/ScriptedImporters.html)
- In-Engine Mesh Asset: Behaves like a regular model, can be dragged around as a prefab



Putting all these steps together we get the main part of our pipeline.



But the pipeline also needs to be simple and flexible to allow for optimization of the atomic meshes and/or iteration of the final assets due to design changes



This pipeline was put in place before my time and I inherited it.
It has heaps of awesome tech and does the job.
It's based on the Unity's AssetImporter API handling the import of a custom file type that in turn generates an asset in the Unity library similar to a 3D model, which has a Transform hierarchy and associated meshes.

gcap games connect asia pacific

Procedural Meshes

AMP is Great!

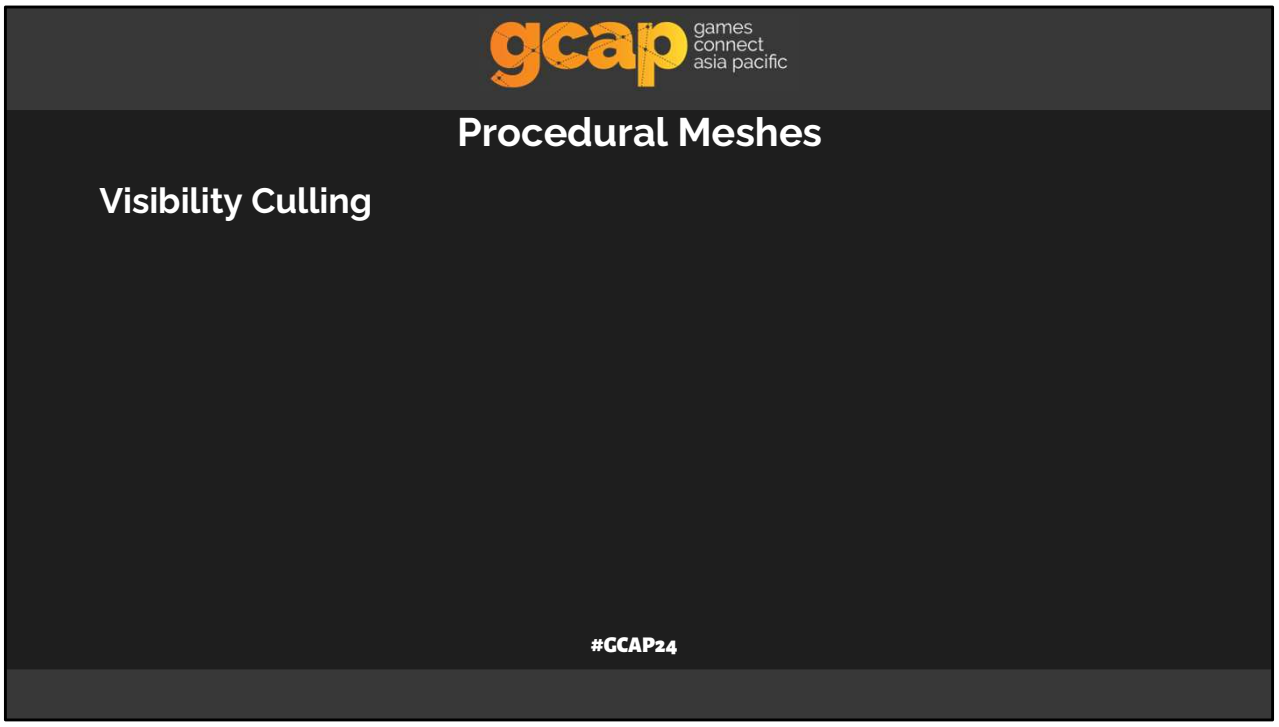
Jamming many meshes together looks great!

But quickly become **expensive due to large number of verts** on the inside and MANY repeated parts.

How can we improve it?

#GCAP24

This idea of procedural meshes is great, flexible and yield quite intricate detail. But the output meshes are insanely high poly, mostly due to internal geometry + geo that will never be seen from our game's mostly static camera setup



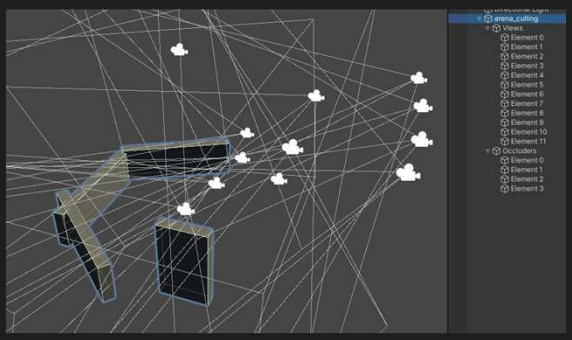
Our solution was an embedded visibility culling solution

gcap games connect asia pacific

Procedural Meshes

Visibility Culling

1) Setup visibility Cameras + Occluders



#GCAP24

It starts by setting up a rig with cameras that mark all the angle the model will be seen in-game.
We can also refine it by adding blocks as visibility occluders if we know that a specific part will be consistently covered by another element, by VFX for example.

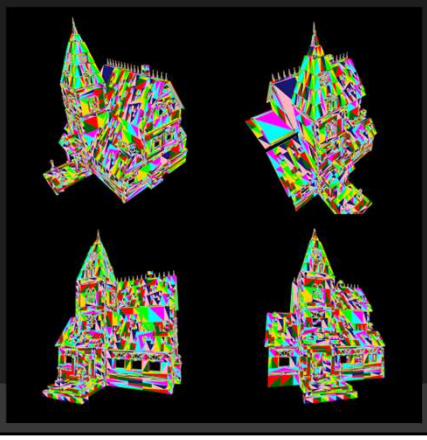
gcap games connect asia pacific

Procedural Meshes

Visibility Culling

1) Setup visibility Cameras + Occluders
2) For each camera
    a) Render triangles indices
    b) Render Occluders on top

#GCAP24



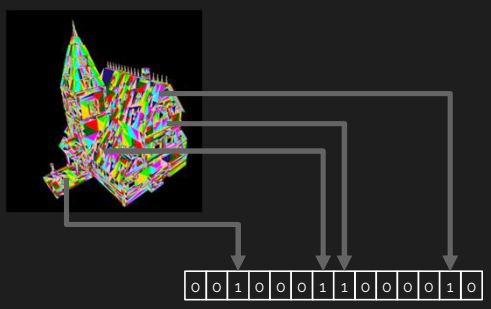
From there we run a process where we render the model from each camera angle, but instead of it's colors we output the triangle index to a uint32 render texture with depth buffer.
We then render the occluders with a null triangle index as to, well, occlude triangles.

gcap games connect asia pacific

Procedural Meshes

Visibility Culling

1) Setup visibility Cameras + Occluders
2) For each camera
    a) Render triangles indices
    b) Render Occluders on top
    c) Collect Data



#GCAP24

Final step is to run a compute shader and aggregate the data on which triangles are visible to a ComputeBuffer.
Rinse and repeat for all cameras.

gcap games connect asia pacific

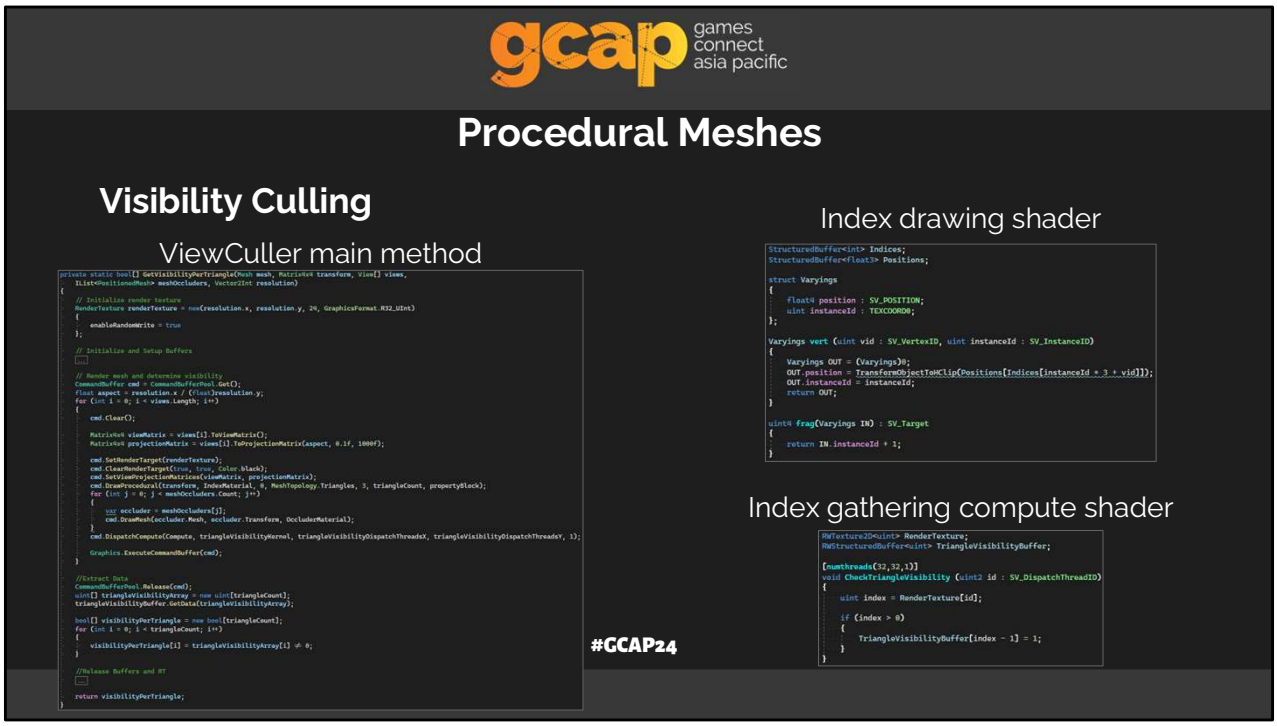
Procedural Meshes

Visibility Culling

1) Setup visibility Cameras + Occluders
2) For each camera
    a) Render triangles indices
    b) Render Occluders on top
    c) Collect Data
3) Aggregate Data
4) Remove Invisible Triangles

#GCAP24

Final step is to just read the data back to the CPU and remove any triangles that are not visible and in turn any vert that is not part of any remaining triangle.



Code Reference

Using a random model as our example
Culling can reduce our meshes up to ~90%!!!
That's right from 6.5 million down to 4.5 hundred thousand verts!
The Nintendo Switch can breath now.

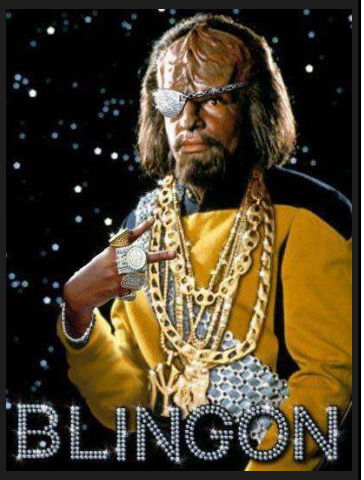
gcap games connect asia pacific

Procedural Meshes

Detail Drawer

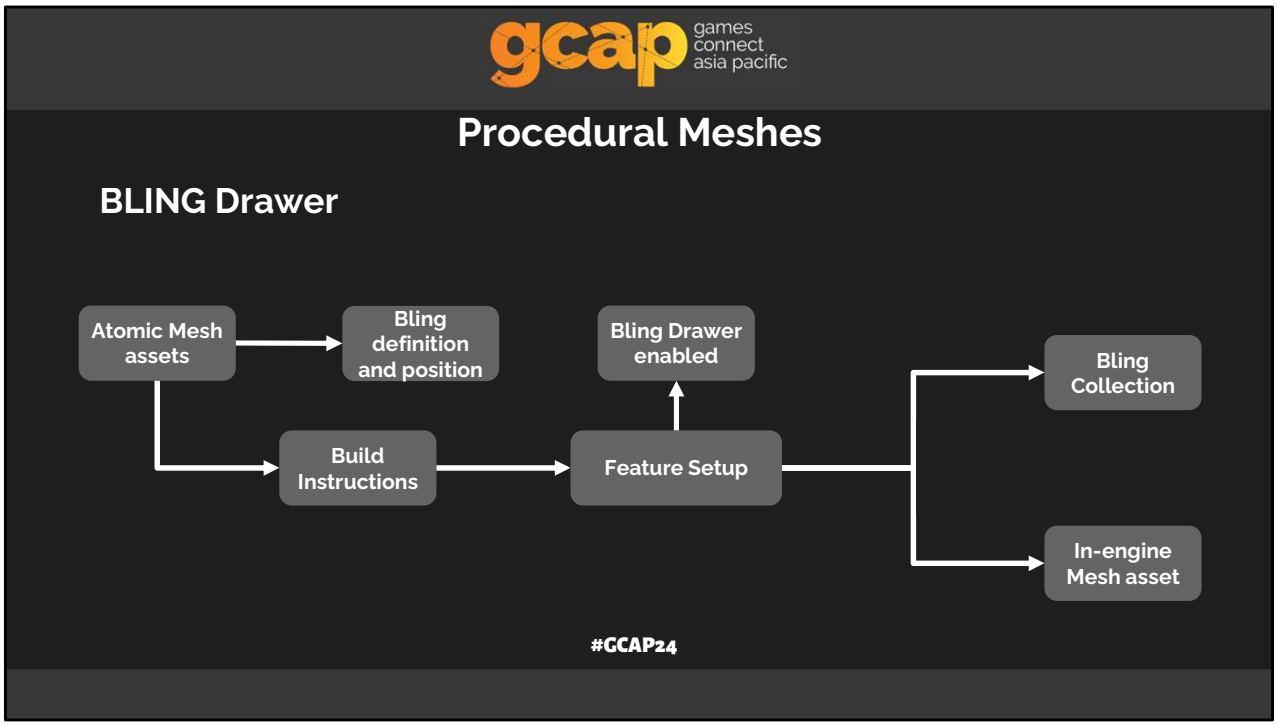
Many of our atomic meshes have what we call "**details**"
And they are small, repetitive geo that are everywhere.

Let's dub them **BLING**

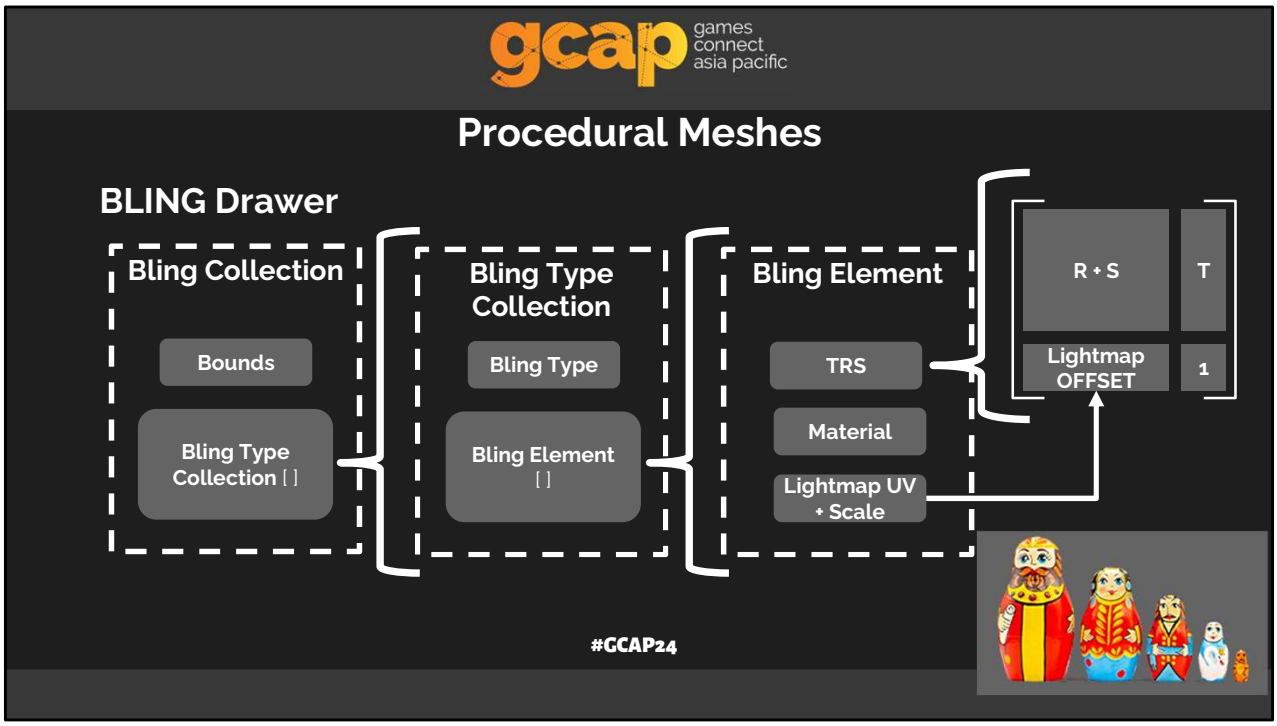


#GCAP24

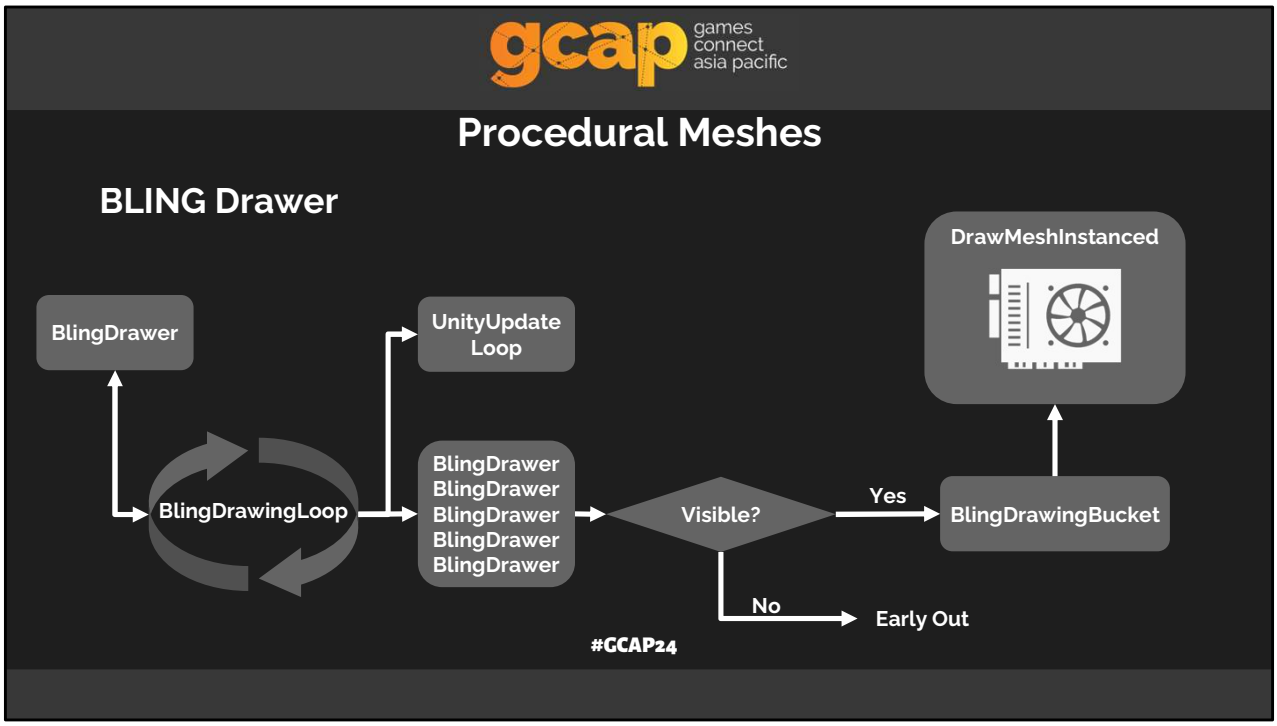
The visibility culling greatly reduced our meshes from millions of verts to hundreds of thousands, but that was still not enough.
Specially on weaker devices the final meshes were still taking too much time to process and consuming too much memory.
A lot of the geometry was spent into small details that were repeated all over the model.
So, to tackle them I created a new feature to render them as instanced meshes.



Going back into the AMP I've implemented a few changes:
For each atomic mesh that had Bling we would store their definition and position as data.
When ingesting the instructions, the AMP would check if the Bling Drawer feature was enabled or not, being able to bake Bling into the geometry or output a scriptable object with a Bling Collection for the model.



This Bling Collection contains the bounds for all Bling in it for visibility evaluation down the line.
It also has an array of Bling Type Collections.
Each of the Type Collections has the bling Type as a string key and another array of Bling Elements
At the bottom of it all the Bling Element contains a TRS Matrix4x4, an uint for compressed material and a Vector4 for Lightmap UV + Scale.
If you're familiar with TRS matrices you know that m30~m32 are set to 0, so we can sneak the Lightmap UV Offset there.



To tie in with the runtime, we have the following parts:

1. A BlingDrawer component that can be attached to GameObjects and has all the settings for drawing a Bling Collection, such as material, cull distance and visibility calculation strategy
2. A BlingDrawingLoop that makes use of the PlayerLoop API to register a custom update method.
3. When a BlingDrawer gets called by the BlingDrawingLoop and is visible, it will call the BlingDrawingBucket which renders the collection via a Graphics.DrawMeshInstanced call

Culling Group API
https://docs.unity3d.com/Manual/CullingGroupAPI.html

Blog on custom update loops
https://medium.com/@thebeardphantom/unity-2018-and-playerloop-5c46a12a677

gcap games connect asia pacific

Procedural Meshes

BLING Drawer

BlingDrawingLoop

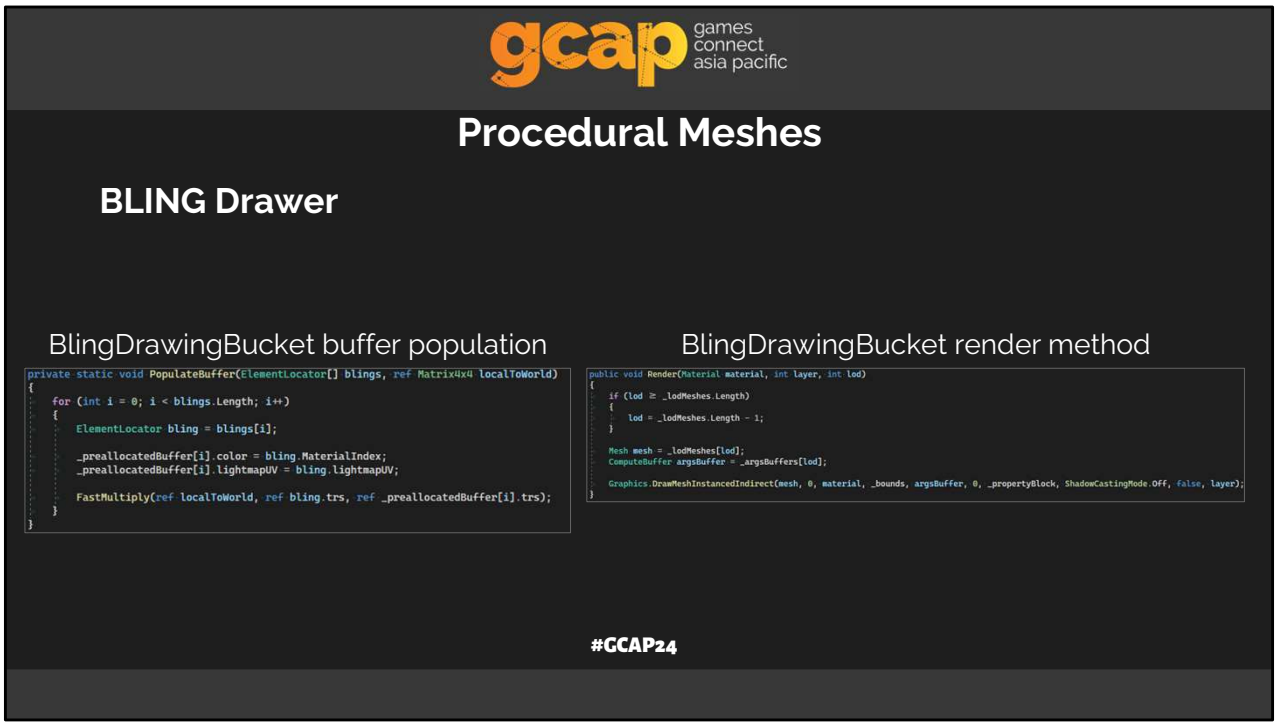
```
public static class BlingDrawerLoop
{
    private static HashSet<BlingDrawer> _activeDrawers = new HashSet<BlingDrawer>();

    static BlingDrawerLoop()
    {
        PlayerLoopUtility.RegisterUpdateMethod(UpdateLoop, typeof(BlingDrawerLoop), PlayerLoopUtility.LoopStage.PostLateUpdate);
    }

    private static void UpdateLoop()
    {
        foreach (BlingDrawer drawer in _activeDrawers)
        {
            drawer.Render();
        }
    }

    public static void Register(BlingDrawer drawer)
    {
        _activeDrawers.Add(drawer);
    }

    public static void Deregister(BlingDrawer drawer)
    {
        _activeDrawers.Remove(drawer);
    }
}
```

#GCAP24

BlingDrawer main methods

```
private async UniTaskVoid InitializeAsync()
{
    Setup References

    Initialize();

    if (HasBakedLightmap())...

    // Set up light map properties for each collection
    for (int i = 0; i < _drawingCollections.Length; i++)...

    BlingDrawerLoop.Register(this);
}

internal void Render()
{
    if (forceRenderingOff)
    {
        return;
    }

    if (ShouldUpdateEveryFrame() && _transform.hasChanged)
    {
        UpdateTransform();
    }

    if (!_isVisible && Application.isPlaying)
    {
        return;
    }

    int layer = gameObject.layer;

    UpdateMaterialsInEditor();

    Material renderMaterial = _lodMaterials[Mathf.Min(_lod, 2)];

    for (int i = 0; i < _drawingCollections.Length; i++)
    {
        _drawingCollections[i].Render(renderMaterial, layer, _lod);
    }
}
```

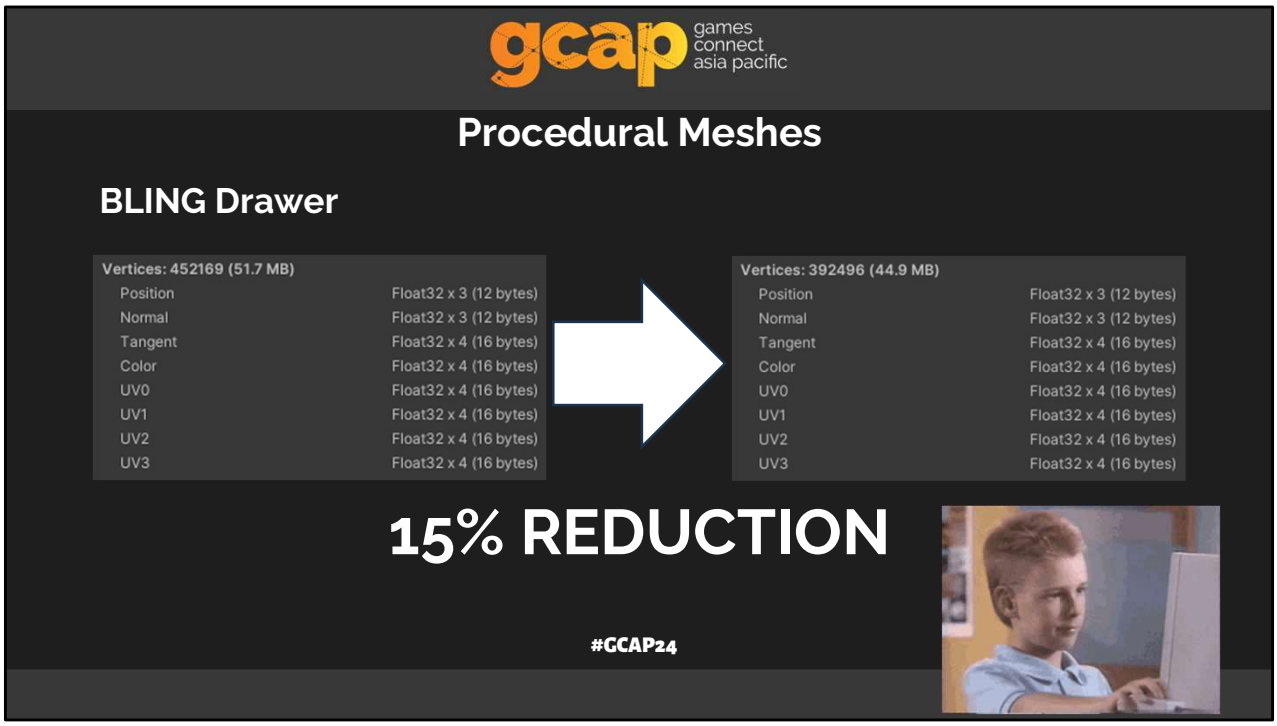
Code Reference




```
private static void PopulateBuffer(ElementLocator[] blings, ref Matrix4x4 localToWorld)
{
    for (int i = 0; i < blings.Length; i++)
    {
        ElementLocator bling = blings[i];

        _preallocatedBuffer[i].color = bling.MaterialIndex;
        _preallocatedBuffer[i].lightmapUV = bling.lightmapUV;

        FastMultiply(ref localToWorld, ref bling.trs, ref _preallocatedBuffer[i].trs);
    }
}
```

```
public void Render(Material material, int layer, int lod)
{
    if (lod ≥ _lodMeshes.Length)
    {
        lod = _lodMeshes.Length - 1;
    }

    Mesh mesh = _lodMeshes[lod];
    ComputeBuffer argsBuffer = _argsBuffers[lod];

    Graphics.DrawMeshInstancedIndirect(mesh, 0, material, _bounds, argsBuffer, 0, _propertyBlock, ShadowCastingMode.Off, false, layer);
}
```

Code Reference



Implementing Bling Drawing on the same mesh as before we get a further 15% reduction on size plus improved resolution on Lightmaps

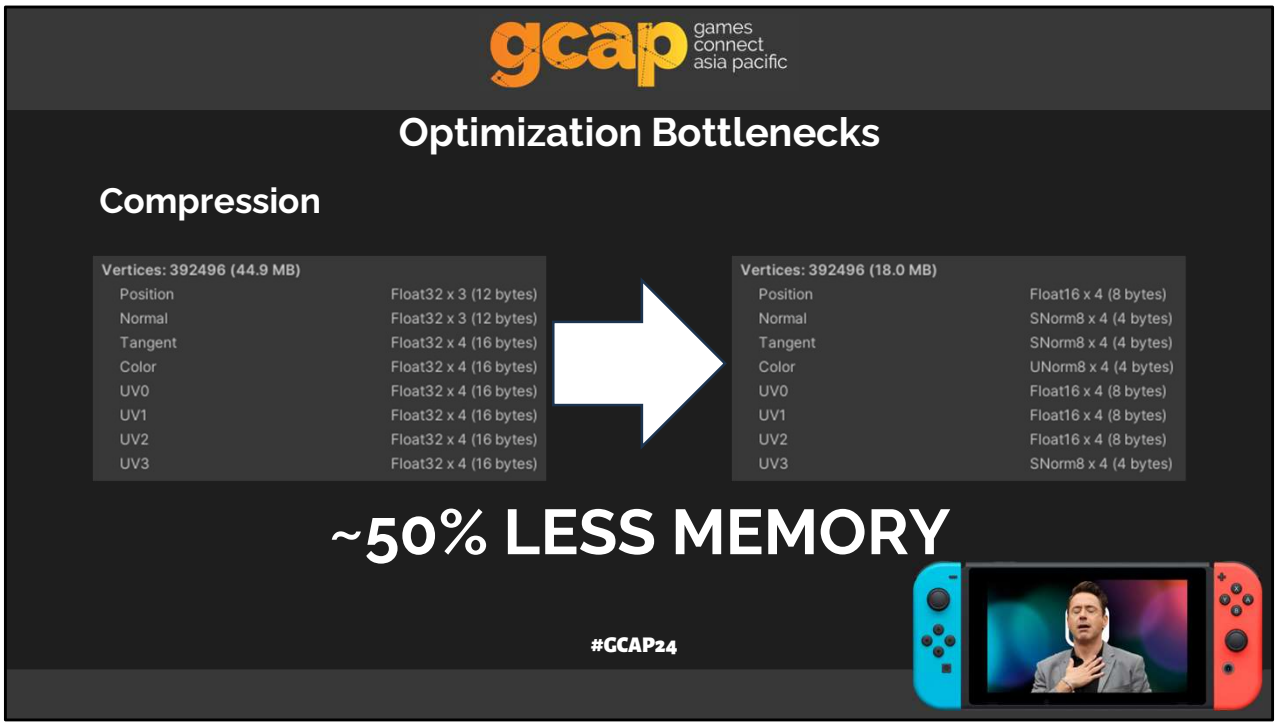
gcap games connect asia pacific

AMP Works!

But is slow, VERY slow
Let's understand why

#GCAP24

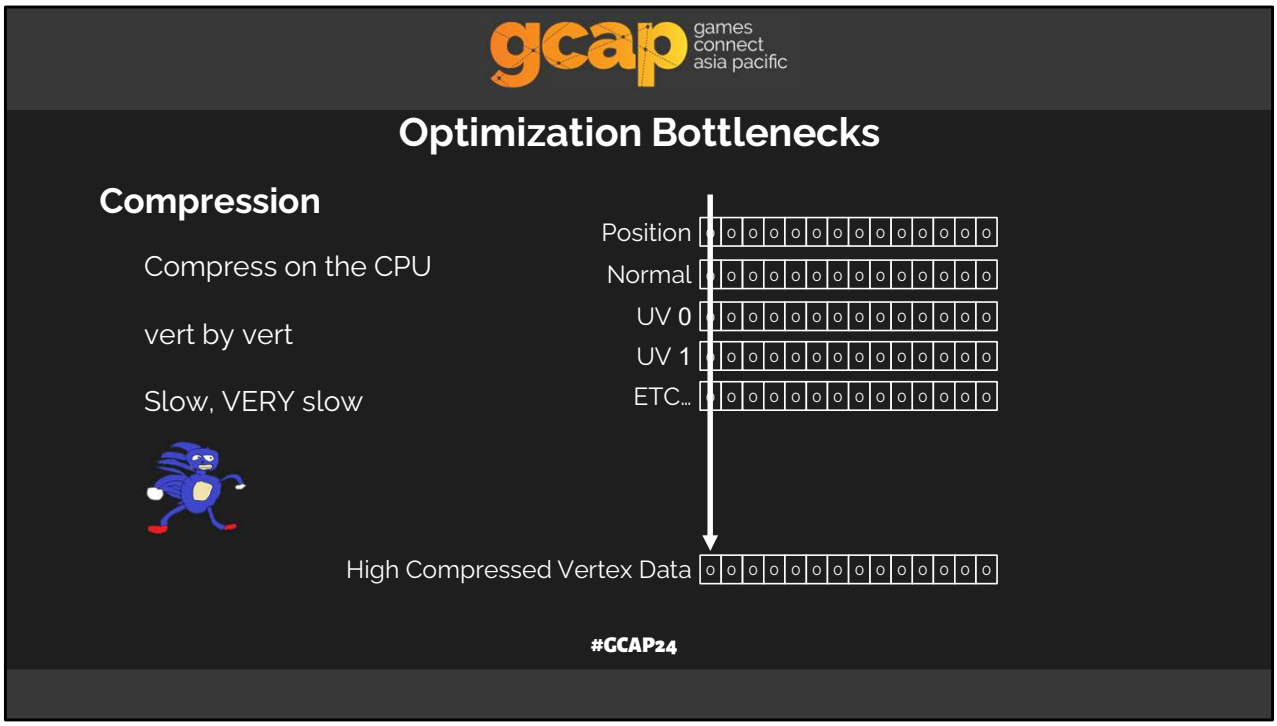
All the tech so far works and is amazing.
But ended up being very slow to process.
Could take upwards to 20min to ingest a model, and sometimes get some computers to run out of memory!!!!
This felt more like Sanic when we needed it to be Sonic!



Why we do it?
Much smaller memory footprint

Update Data formats
Float32 to Float16 or even Norm8
Almost 60% reduction!!!

Our meshes are chunky and don't suffer from compression in most cases



Our first solution was great and functional, which served us well while investing into a vertical slice.
It just used C# to read and convert the data from regular Float32 into smaller Half16 or SByte4, following the format outlined by our HighCompressedVertexData struct
But this process is linear and greatly suffer from very large meshes with a long vert strip.

gcap games connect asia pacific

Optimization Bottlenecks

Compression

Repetitive Task?

Has well-defined contraints?

COMPUTE SHADERS TO THE RESCUE!!!

#GCAP24

Compute shaders works wonders on well defined parallelized tasks such as stride defined vert strips!
20% reduction on total processing time

Optimization Bottlenecks

Compression

```
HighCompressedVertexData[] vertexData = new HighCompressedVertexData[vertexCount];

ComputeBuffer ouputBuffer = new ComputeBuffer(vertexCount, Marshal.SizeOf(typeof(HighCompressedVertexData)));

Compute Setup and Dispatch

ouputBuffer.GetData(vertexData);
```

Setting the data to a defined format is easy to setup, but…

No Normals? Different format

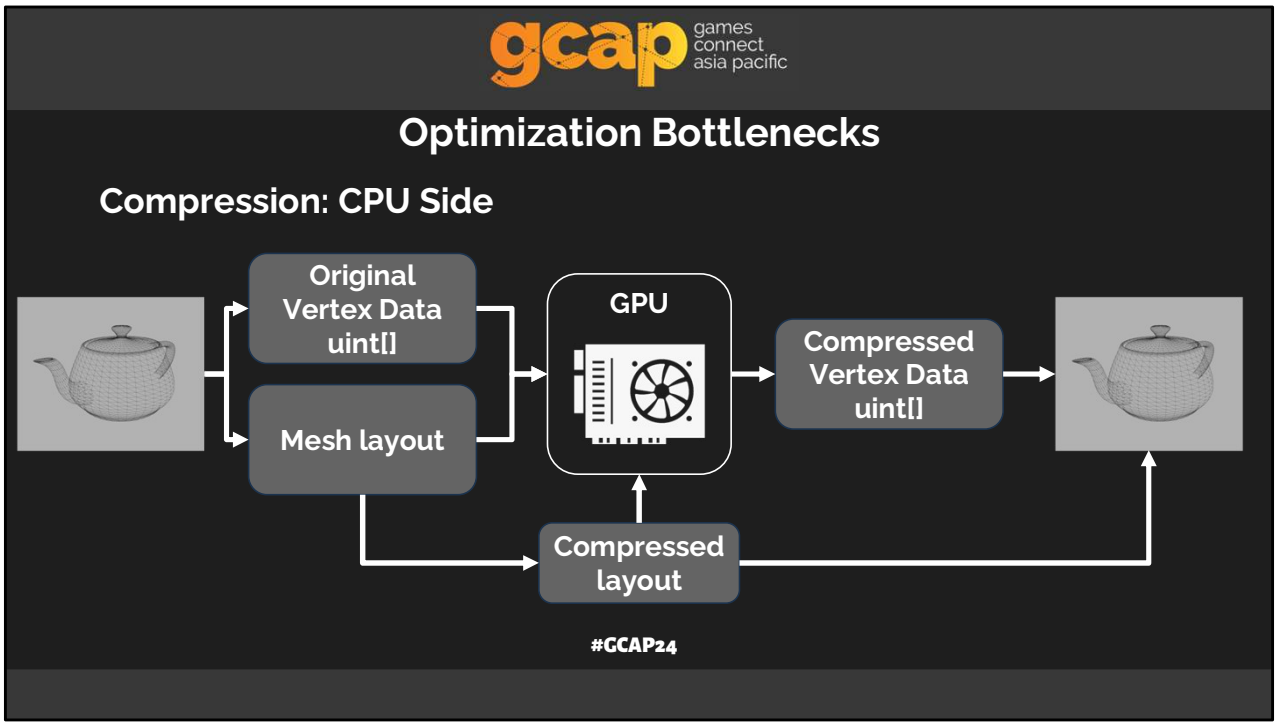
No Colors? Different format

But… Aren't we just moving bytes around?

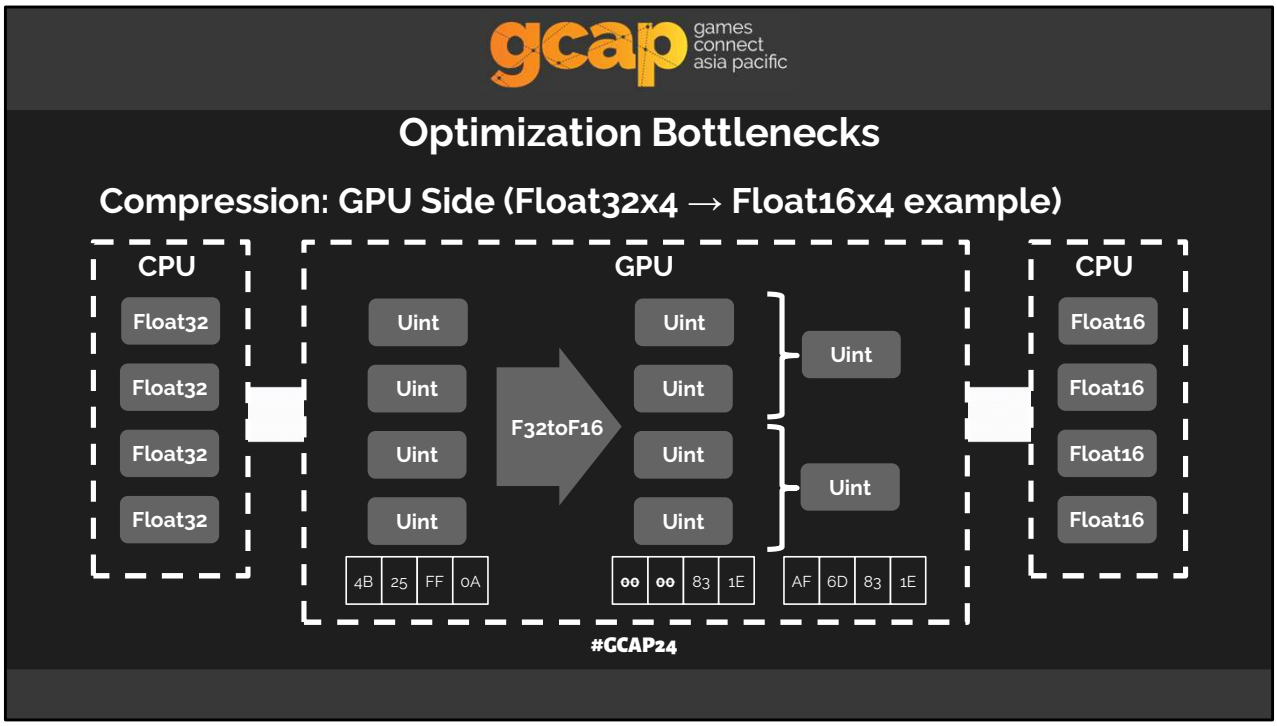
#GCAP24

But this first approach is hardcoded on HighCompressedVertexData struct vertex format, which might need to be cleaned afterwards if the input did not have all attributes.
And despite having meshes with more or less channels, the compression per channel is consistent across the project.

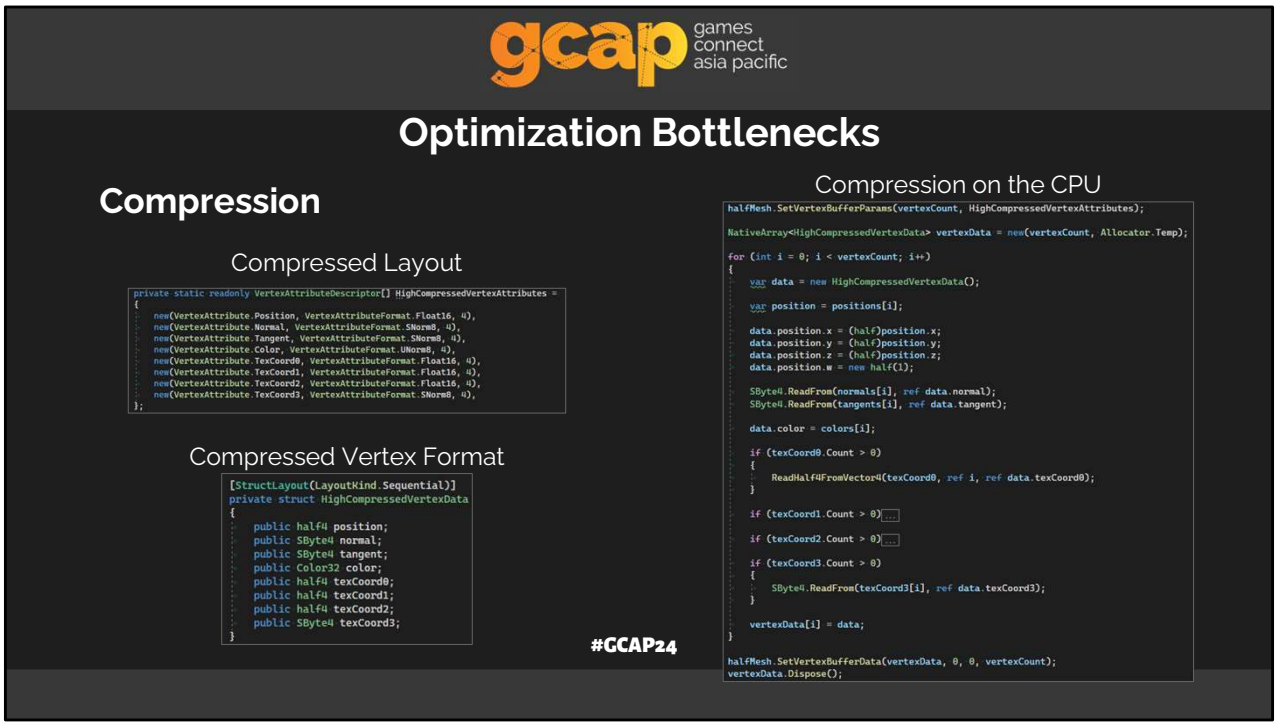
But it's all just bytes…



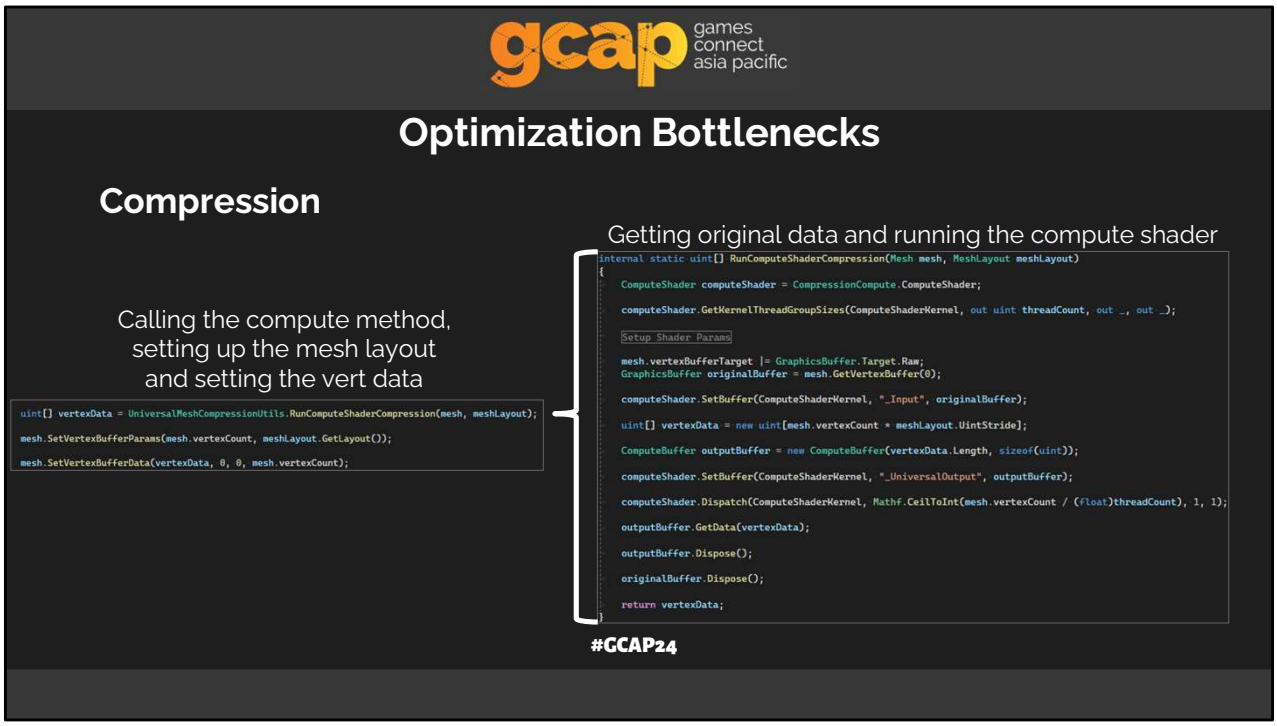
Instead of relying on HighCompressedVertexData struct, we can use a plain uint array, where bytes can be directly read from the GPU and applied to the mesh via the method Mesh.SetVertexBufferData() afterwards.
This is done by extracting the vertex data and mesh layout with Unity's own methods.
The data gets passed as is to the compute shader, while the layout is passed as Booleans, such as _hasNormal, or _Texcoord1Has4Channels.
After compression the data is fetched back into the CPU, the mesh has it's layout updated and finanly it receives the now compressed data.



On the GPU side we just need to read the bytes for each uncompressed value as their uint representation, such as a float3 becomes uint3, given they both have the same bit count (4 bytes * 3).
From there we can compress following any rule we need, as long as each channel fit withing the multiple of 4 byte count, e.g.: float3 is fine (4 bytes * 3 = 12), but half3 isn't (2 bytes * 3 = 6).
It's also important to note that the output will be packed into uints, which are 4 bytes each, so 2 halfs need to be packed into a single uint, or 4 Unorm8 into the same single uint.
Finally all these uint soup is appended to the output buffer sequentially, which we can parallelize as we know the exact stride (or size) of each vert data.



Code Reference



Code Reference

gcap games connect asia pacific

Optimization Bottlenecks

Compression

Kernel Method

```
void UniversalCompressMesh(uint3 id : SV_DispatchThreadID)
{
    int idx = id.x;

    if (idx >= _MeshVertexCount)
    {
        return;
    }

    UncompressedVert uncompressed = ReadUncompressedVert(idx);

    uint outputIndex = idx * _OutputMeshStride;

#define Append(value)   _UniversalOutput[outputIndex] = value; \
                        outputIndex++;

    if (_uncompressedPosition)
    {
        Append(uncompressed.uncompressedPosition.x)
        Append(uncompressed.uncompressedPosition.y)
        Append(uncompressed.uncompressedPosition.z)
    }
    else
    {
        uint2 position = CompressFloat4IntoHalf4(float4(uncompressed.position.xyz, 1));

        Append(position.x)
        Append(position.y)
    }
```

Beginning of the compute shader, it keeps going depending on the mesh layout.
But it's just more of the same from here

Data Reading

```
UncompressedVert uncompressed = (UncompressedVert) 0;

uint inputIndex = idx * _MeshStride;
int offset = 0;

uncompressed.uncompressedPosition = _Input.Load3(inputIndex + offset);
uncompressed.position = asfloat(uncompressed.uncompressedPosition);
offset += 4 * 3; //float3

if (_hasNormal)
{
    uncompressed.normal = asfloat(_Input.Load3(inputIndex + offset));
    offset += 4 * 3; //float3
}

if (_hasTangent)
{
    uncompressed.tangent = asfloat(_Input.Load4(inputIndex + offset));
    offset += 4 * 4; //float4
}
```

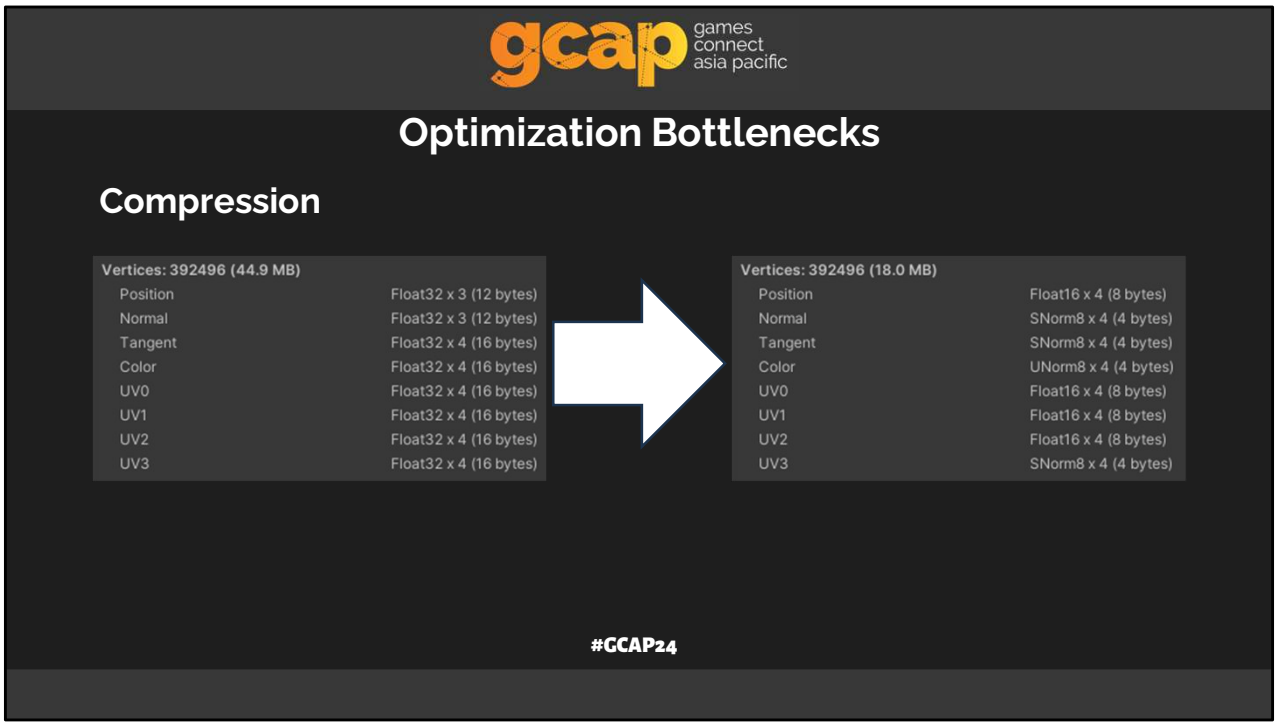
This is the beginning of the ReadUncompressedVert method.
The rest will just read the other mesh data

Compression Methods

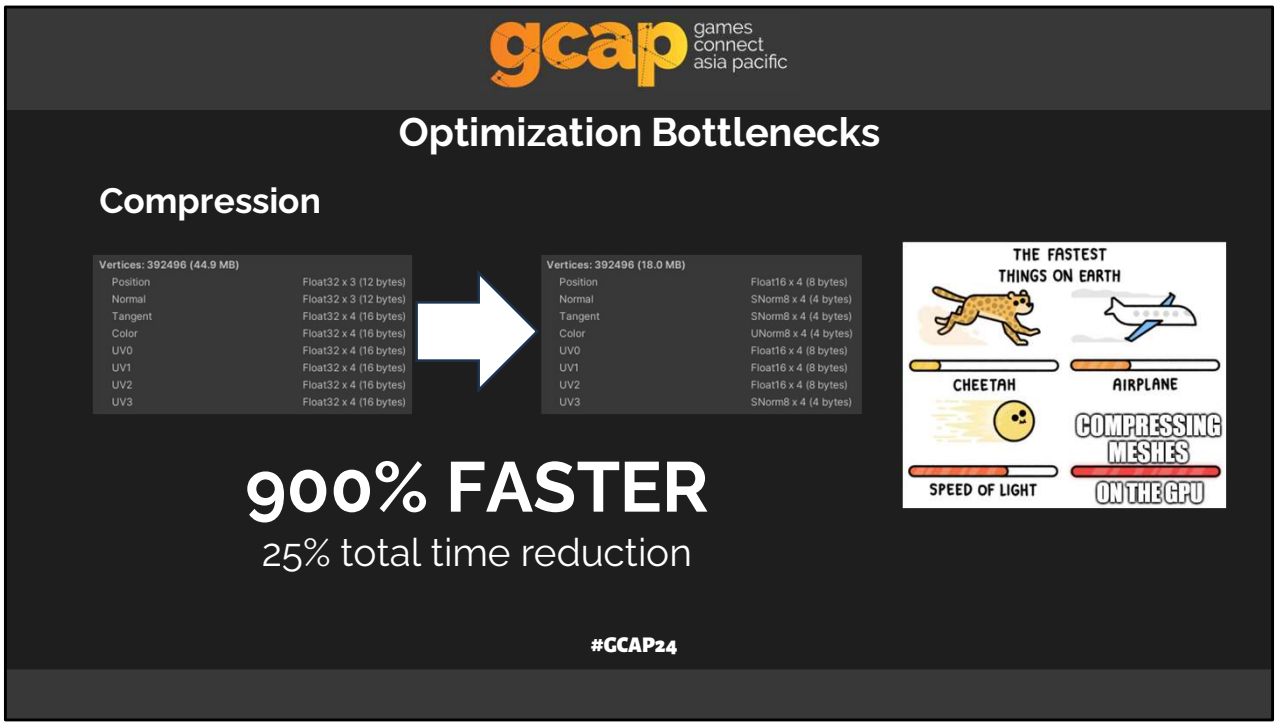
```
uint CompressFloat2IntoHalf2(float2 value)
{
    uint x = f32tof16(value.x);
    uint y = f32tof16(value.y);

    return x | (y << 16);
}

uint CompressFloat4IntoOnebyte4(float4 value)
{
    bool4 signs = value < 0;

    float4 compressed = abs(value) * 127;

    int4 compressedInt = round(compressed);

    compressedInt = signs ? -(compressedInt - 1) : compressedInt;

    compressedInt &= 0xff;

    return compressedInt.x | (compressedInt.y << 8) | (compressedInt.z << 16) | (compressedInt.w << 24);
}

uint CompressFloat4IntoUNorm8(float4 value)
{
    float4 compressed = value * 255;
    compressed = round(compressed);
    int4 compressedInt = compressed;

    return (compressedInt.x & 0xff) | ((compressedInt.y & 0xff) << 8) | ((compressedInt.z & 0xff) << 16) | ((compressedInt.w & 0xff) << 24);
}

float2 CompressFloat4IntoHalf4(float4 value)
{
    return uint2(CompressFloat2IntoHalf2(value.xy), CompressFloat2IntoHalf2(value.zw));
}
```

These are our compression methods to convert Float32 values into lower precision values

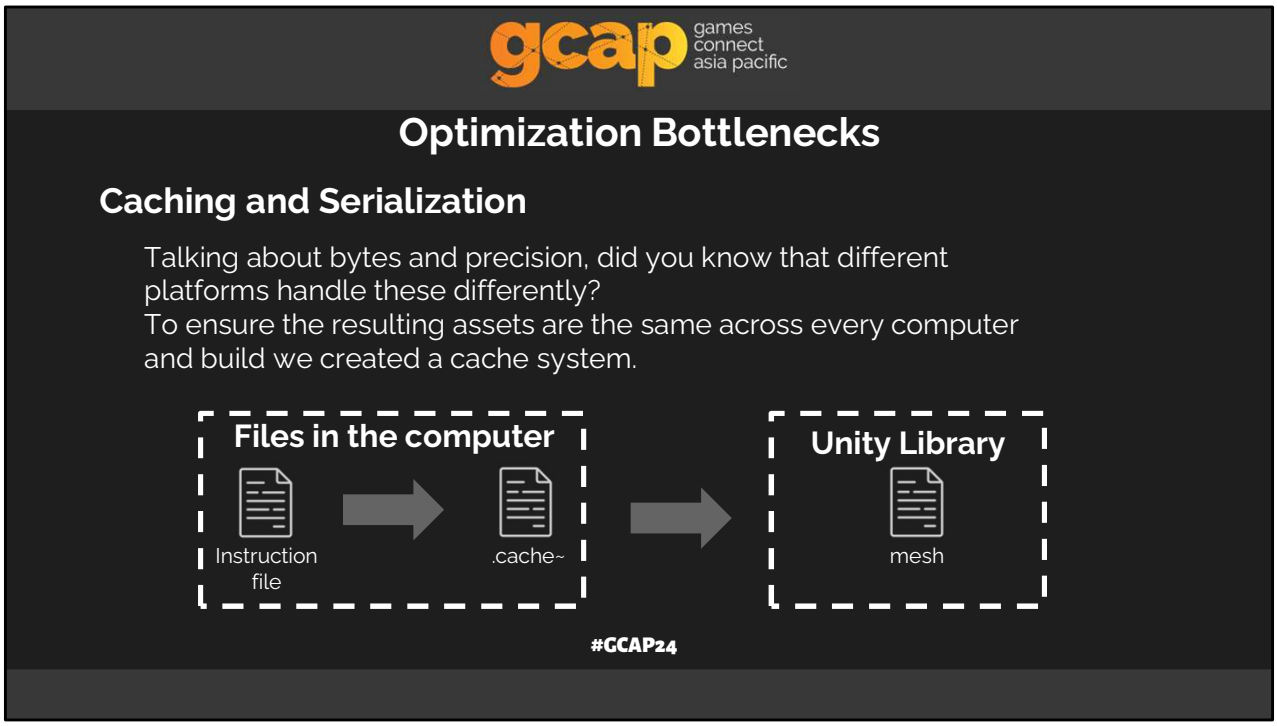
Code Reference



Taking the starting case



The Compute Shader method is ~900% faster!
Which cascades as the entire AMP being 25% faster!



Different platform could generate slightly different results for the mesh generation, specially lightmap unwrapping using xAtlas.
To ensure that the results would be stable and reliable (as well as faster) we created a cache system where the AMP, when importing an instruction file it would check for an adjacent .cache~ file;
If it finds one the .cache~ is read and the result mesh is added to the Unity library, but if it's missing (due to being a new instruction file) it would be generated from scratch and a .cache~ would be then created and committed to the repo.

gcap games connect asia pacific

Optimization Bottlenecks

Caching and Serialization

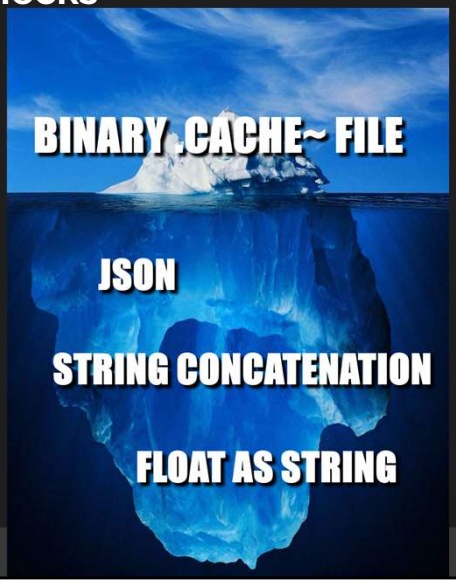
The caching is done by serializing the result meshes to a file as binary.
But how is the binary serialization done?

JSON

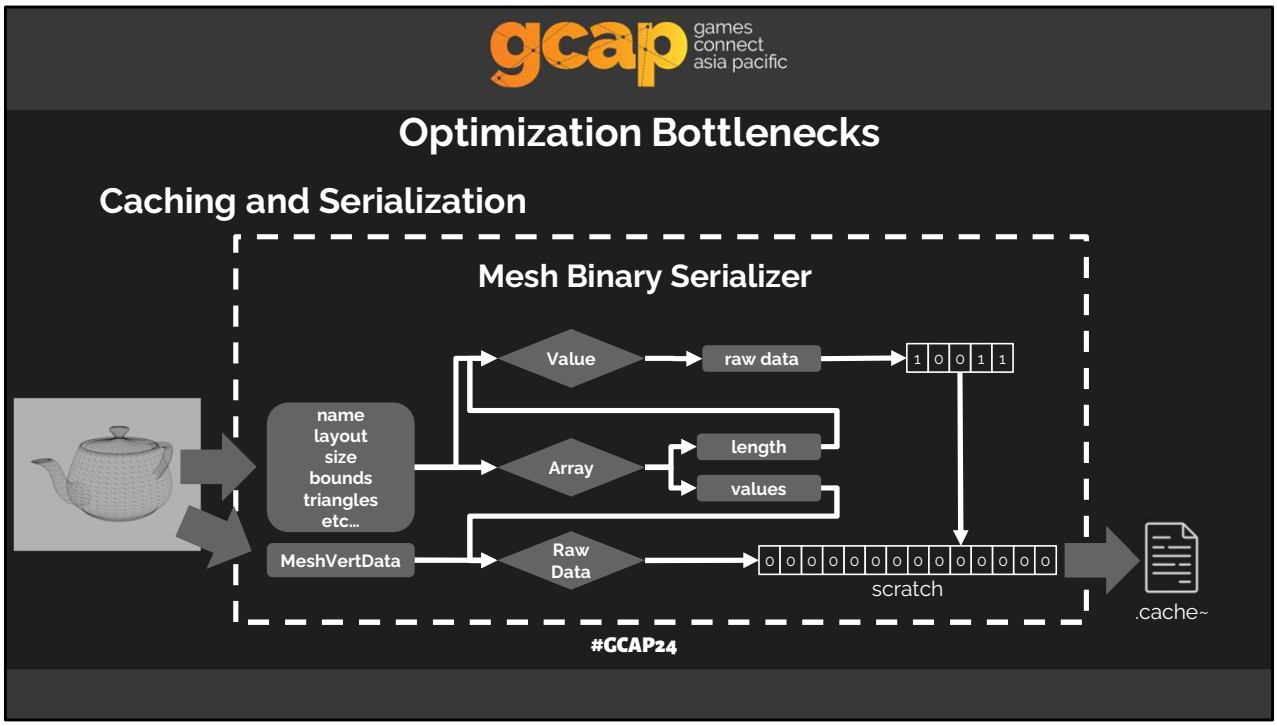
All the data was being interpreted as numbers
Then as text
Then concatenated into JSON
Then converted into bytes
Finally wrote to disk

AND BACK!!!

#GCAP24



The process was sound and seemed great, but how did it work?
Via JSON strings.
Converting data into text and then into binary so it could be written to disk, and the inverse when reading.
This was bad as not only it was quite slow, it also ate up a lot of memory AND lost precision, as different hardware round floating point numbers differently on ToString().
The memory situation was so bad that some instruction files couldn't be ingested as it would require more memory than people's computer had!
Also found out that whenever a mesh was generated, we'd serialize it to disk and then immediately deserialize it back, wasting lots of time when (re)generating a cache.



The improvement was a new serializer, that works in the following way:
We start with a mesh, then create a new Serializer, which contains a large byte array that we'll call the scratch.
First we extract the info from the mesh, such as name, attribute layout, triangles, etc...
That info can be separated into a few different types:
The first type is "Values", which contain 2 parts

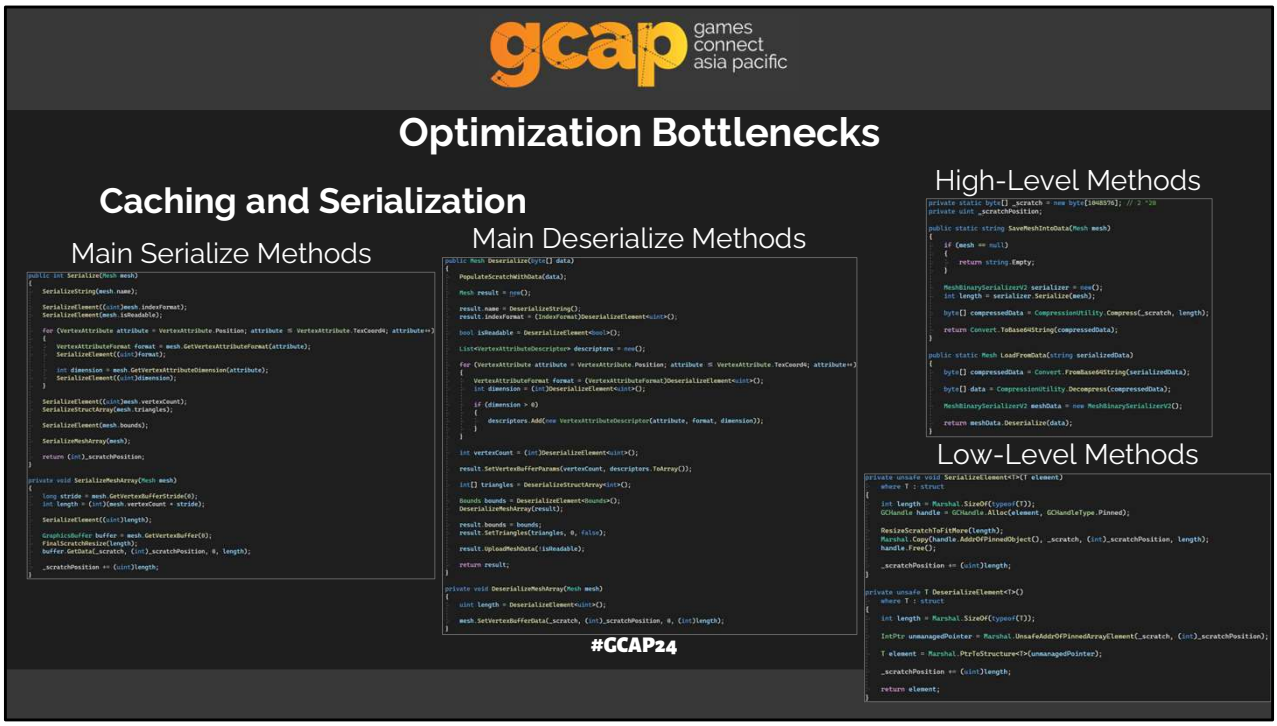
- the type size, for example Bounds is a Vector3, so 3 floats for 4 x 3 bytes
- the raw data, the actual binary representation of the value

The type size ends up being a raw data itself, and all raw data can be copied into the scratch
The second type is "Arrays", which also contain 2 parts

- the length, which can be saved as a Value
- the actual values, which can be copied as raw data

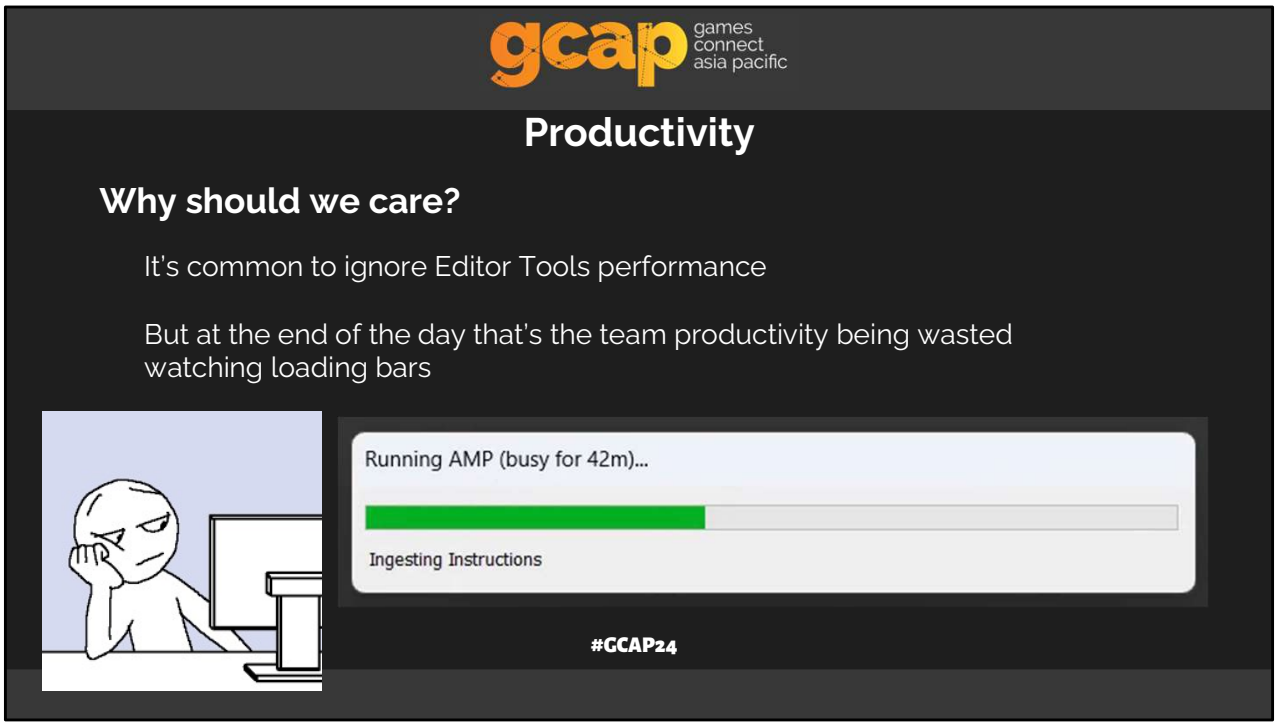
Raw data is highlight as a third type because we can serialize the MeshVertData from the mesh directly.
Then all this get written to disk as a string representation of the scratch

The inverse of this process is done for deserializing and this ended up being 25% faster for deserializing .cache~ files as well as massively reducing the memory usage, allowing the team to handle any instruction that they encountered since without crashes!

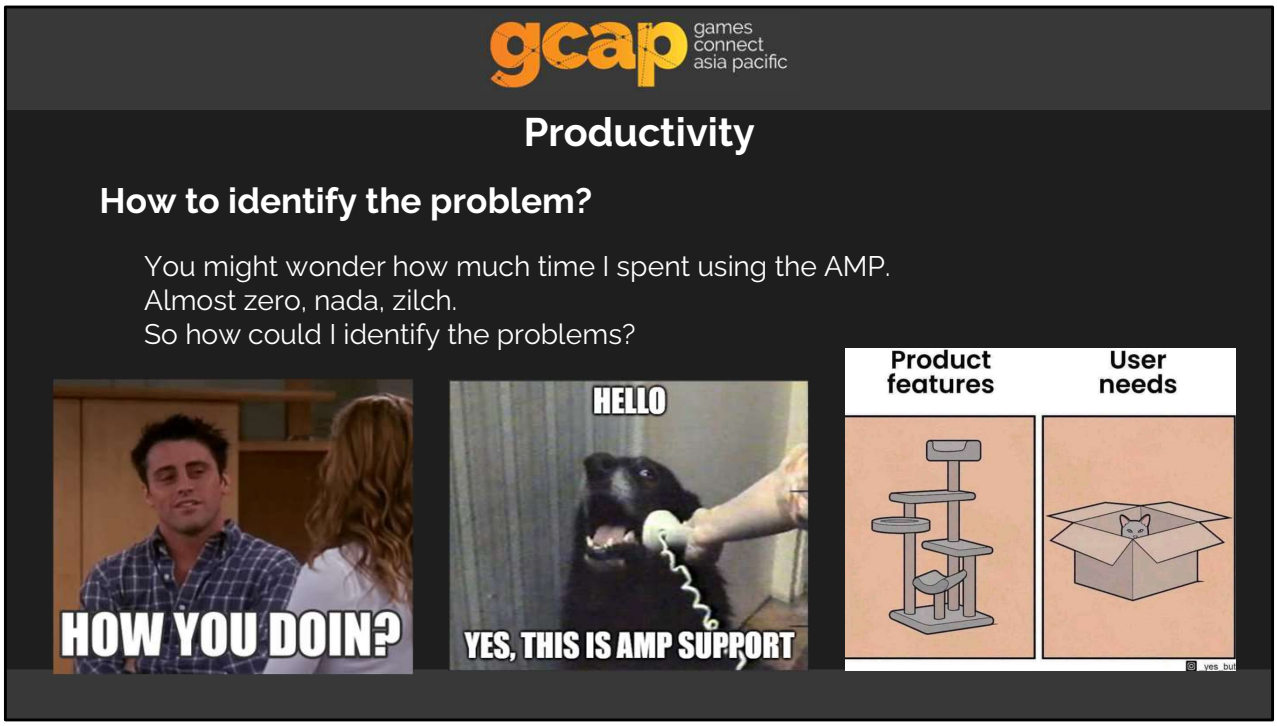


Code Reference

Using the same mesh we've compressed before, the new serializer is 55% faster when writing and 65% faster when reading (which happen more often!)
This led to another speed up of 25% of the AMP!!!
It also prevented memory related crashes



And why do all this just for an Editor tool? No one cares if it's slow.
In reality we all SHOULD care if it's too slow.
Because it's people's time we're talking about, it's not a big deal if a button takes a few seconds.
But if it takes many minutes, you might want to look into it.
There is a balance as some process are done sparingly and can be slow, but in those cases it shouldn't crash after 30min or it will be a huge source of frustration.



Although I put a lot of effort into understanding and improving the tool, I barely used it.
So how could I identify issues?
By doing 3 main things:

1) Asking the team how they were doing, what was bad and annoying
2) Having an open call policy, whenever they wanted to complain about something I'm always a slack call away
3) By lurking on their open huddles and paying attention to how they work so I can spot pain points

gcap games connect asia pacific

Productivity

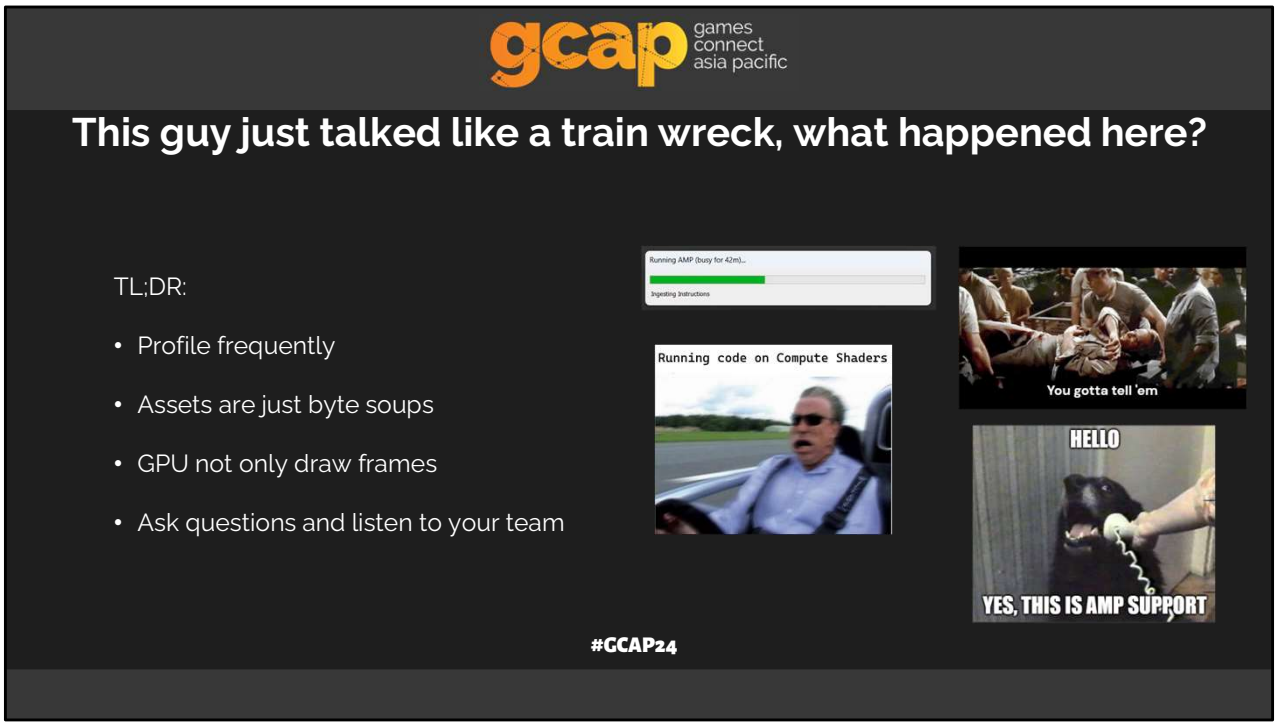
Do you need to do it all yourself?

NO!

The same way that you're doing this for the team, **rely on them for help**

Engage your team in coming up with solutions, even with coding tasks

This helps people to stay motivated and yourself a little bit saner!



- Profile frequently
  - Profile the game and profile the engine, your tools might be killing you
- Assets are just byte soups
  - Don’t be afraid to treat them like so

  - Compute shaders are a powerful

tool for grunt work and even draw calls can be used to “paint” data

- Ask questions and listen to your heart, I mean, your team
  - Talk to people who use your tools, they might not know how it works, but they know how it doesn’t


gcap
games
connect
asia pacific
Questions?
#GCAP24

gcap games connect asia pacific

The Tech Art Journey of creating and optimizing an asset pipeline

Obrigado!

/vicnascimento

SLIDES

#GCAP24


gcap
games
connect
asia pacific