Basics of optimizing VRChat worlds

By MyroP

Download the presentation/benchmarks on my GitHub

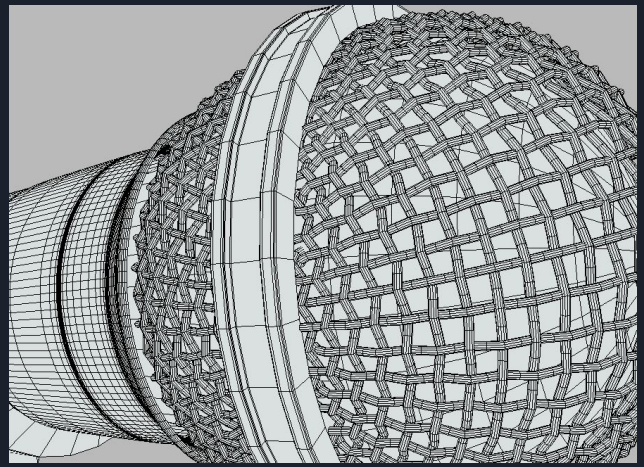
https://github.com/MyroG/Optimizing-VRChat-worlds

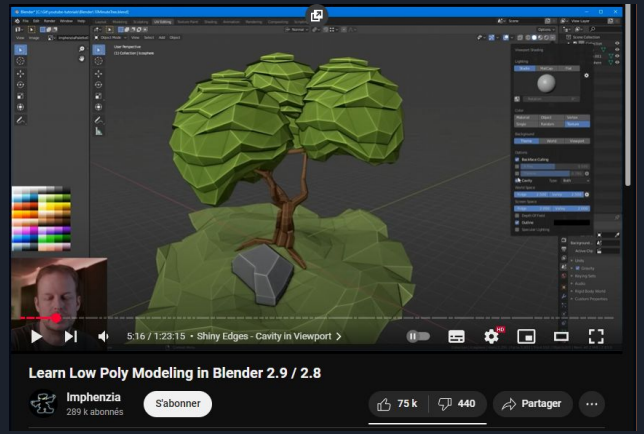
Who am I ?

- 32 years-old French software developer
- Creating VRChat worlds since 2021, as a hobby
- Some worlds I created : VRbomber, VRtual park, Nucleus etc.

You should learn Blender!

- Many assets we can download aren't « game ready »:
  - Too many polygons, materials, meshes etc.
  - TurboSquid, Sketchfab, even on the Unity asset store !
- « Learn Low Poly Modeling in Blender 2.9 / 2.8 » by Imphenzia
  - Great video to learn the basics (shortcuts, low-poly modeling, modifiers, UVs)
  - https://www.youtube.com/watch?v=1jHUY3qoBu8



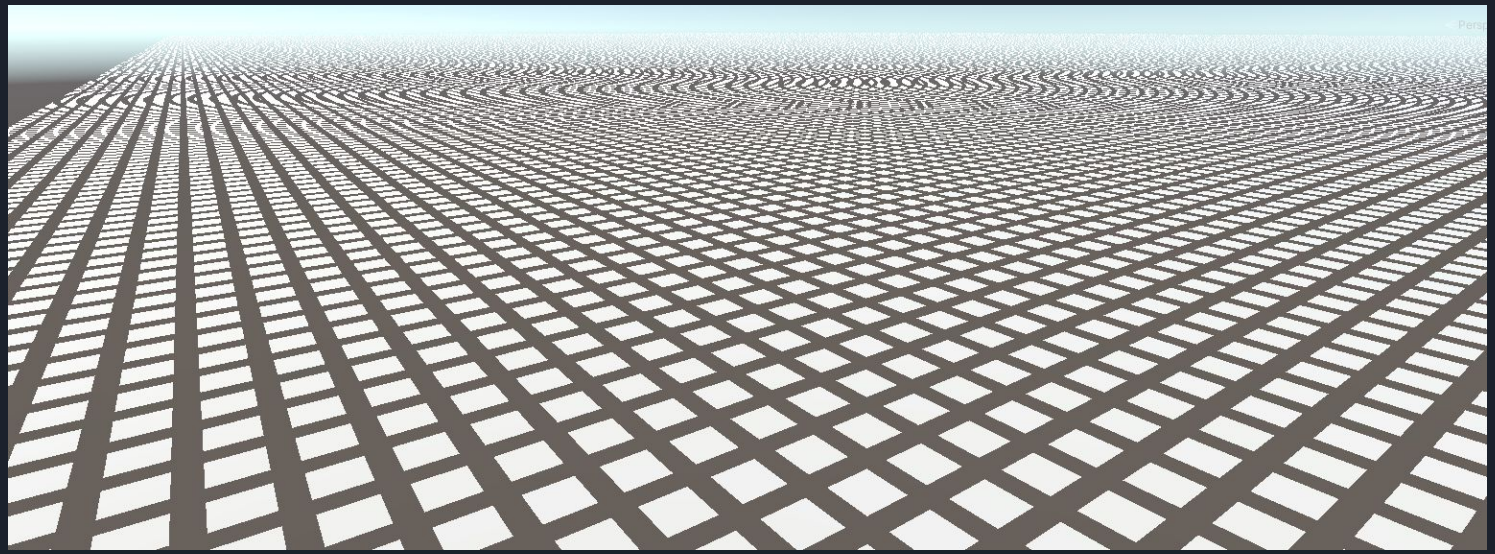


FpsVR

- Great to test performance inside VRChat.
- To perfectly test the performance of your world :
  - Use a very optimized avatar!
  - Be alone in the instance

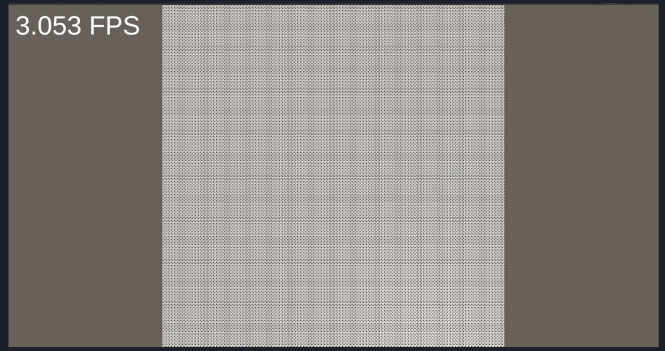
Let’s start with a very basic example…

~1 million separate quads, ~2 million triangles

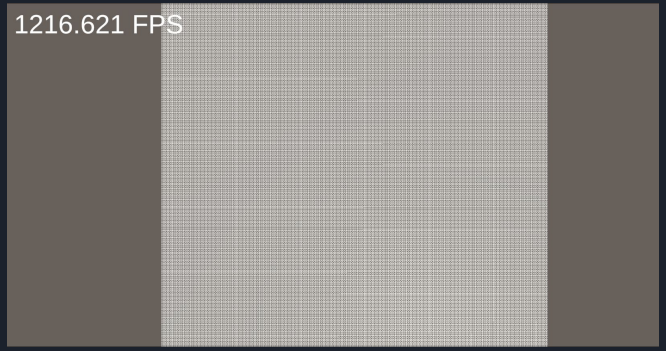


Let’s compare the performance of two setups

Each square is a separate mesh



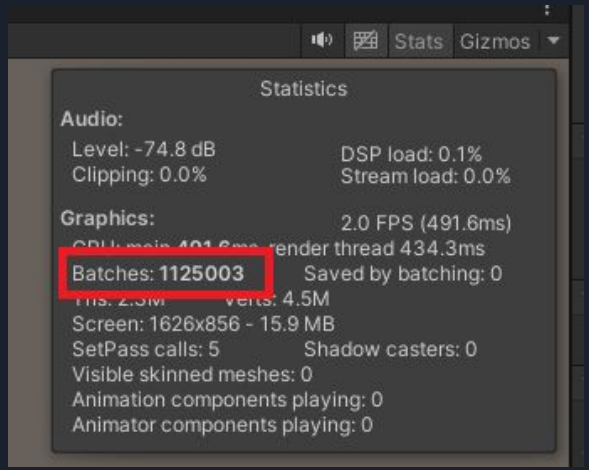
One single mesh

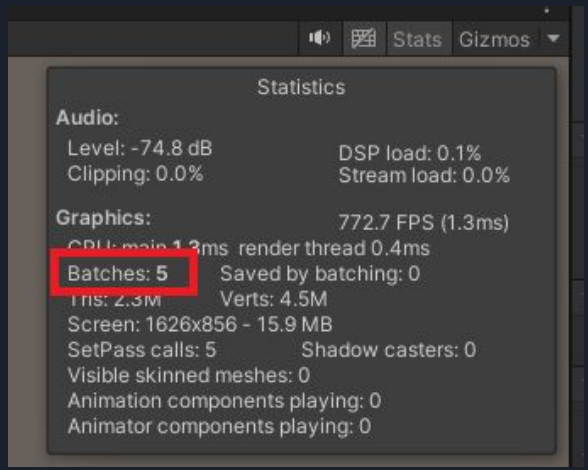


Much better performance with the scene on the right !

Draw calls

In the « Stats » window, we can see why our left scene performed so poorly





Way too many batches !

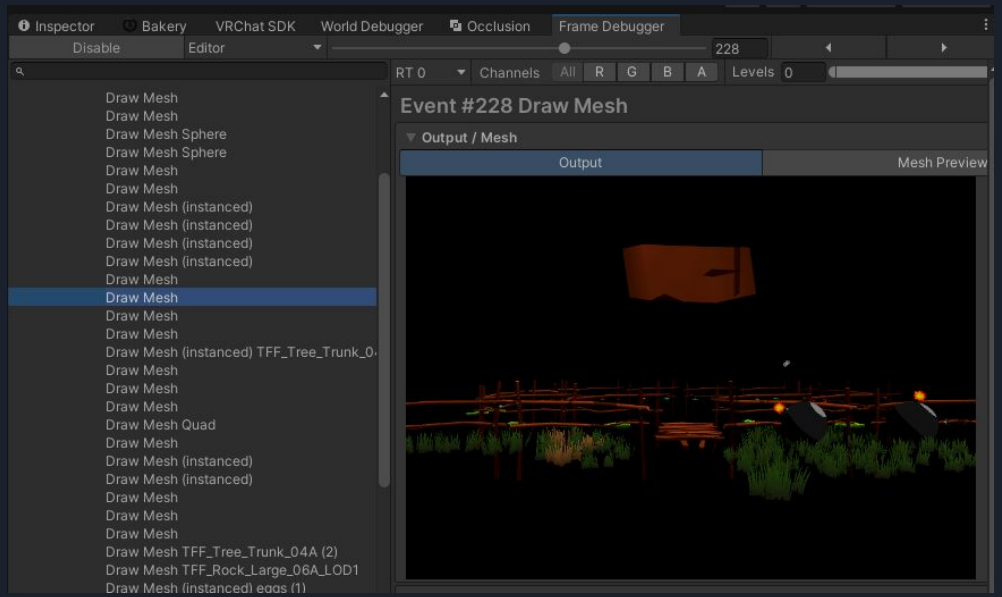
Draw calls

A draw call tells the graphics API what to draw and how to draw it. Each draw call contains all the information the graphics API needs to draw on the screen.

- Draw calls are CPU bound : The more draw calls, the worse the performance on the CPU side.
  - Less draw calls improves the performance on the CPU side.
- Many things can increase the number of draw calls :
  - Number of objects and materials in the scene
  - Realtime lights casting a shadow
  - Cameras (including the VRChat mirror)
  - Etc.

Unity Frame Debugger

- Allows us to analyse each draw call
- Windows > Analysis > Frame Debugger
- Great tool to find the source of our draw calls!



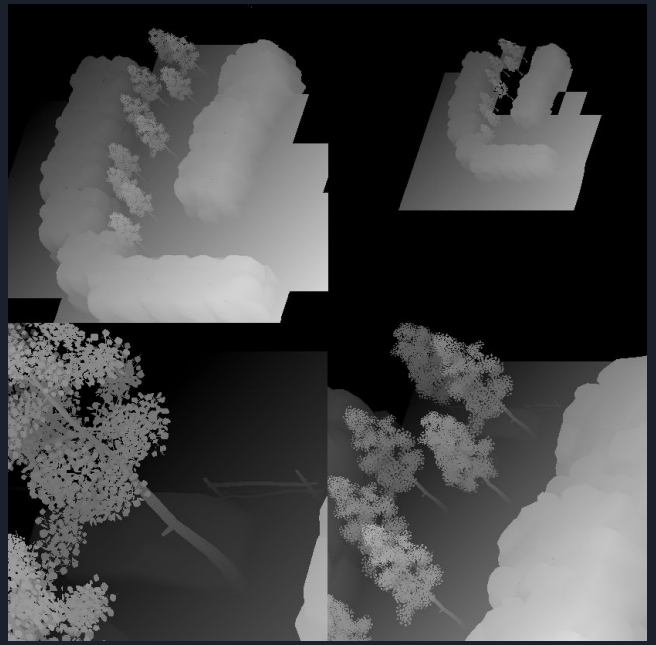
Passes and materials

- A pass is a single rendering operation, each pass = 1 draw call.
- Shaders can define multiple passes:
  - Example : A Classical PBR shader
    - Base pass : handles base color, directional light, lightmap etc.
    - “ForwardAdd” pass : Handle RT per-pixel lights (point/spot lights) in the scene
  - A PBR shader affected by a point light:
    - 1 draw call for the base pass
    - 1 additional draw call per light affecting the object
- Each material requires at least one draw call
  - Example: A mesh with 3 materials = 3 draw calls minimum

- Scenario : An avatar with 10 separate meshes, each using 2 PBR shaders, and a point light in the scene:
  - 10 meshes x 2 materials x 2 draw calls (1 base pass, and 1 “ForwardAdd” pass) = 40 draw calls

Real-time shadows

- Unity uses shadow mapping for real-time shadows
- Generates a shadow map from the light's perspective, similar to a camera
- Requires re-rendering the scene from the light's view, increasing draw calls and GPU workload
    - Each separate object can be a separate draw call!



Q : How can we reduce the number of draw calls?

A: With batching!

Batching

What is batching?

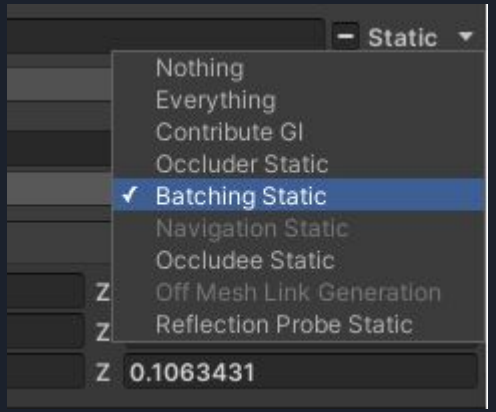
- Batching = combining multiple draw calls into one.
- Goal: Reduce draw calls → improve performance.

Unity supports several batching techniques:

- Static batching
- GPU instancing
- Dynamic batching

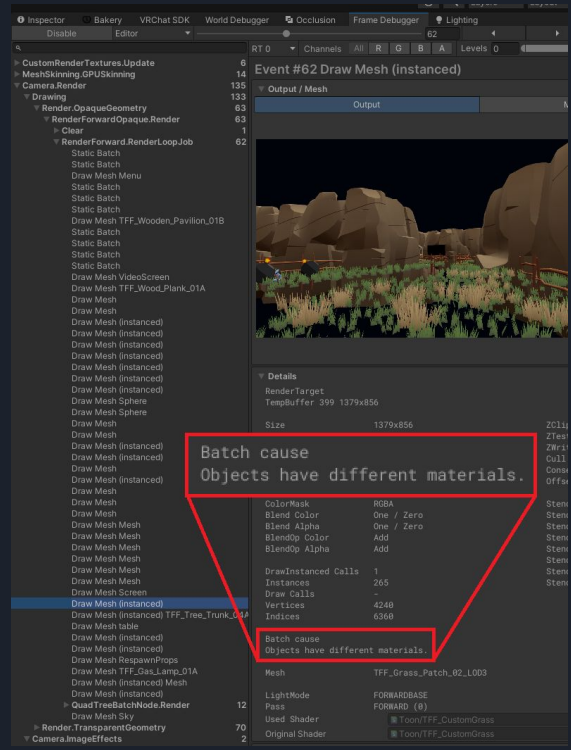
Static Batching

- Static batching merges meshes during the build time.
- To turn on static batching, the checkbox « Batching static » needs to be checked
- Downside:
  - Creates an additional mesh
  - Increases build size
  - Increases memory usage
  - Can cause issues if the mesh uses a vertex shader, like a billboard shader



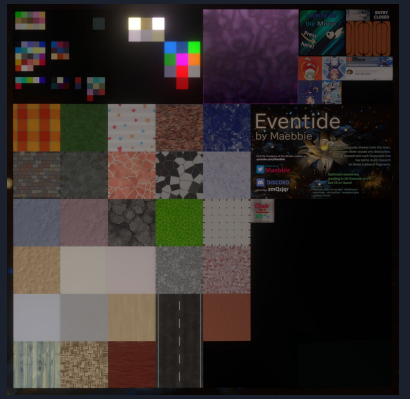
Static Batching

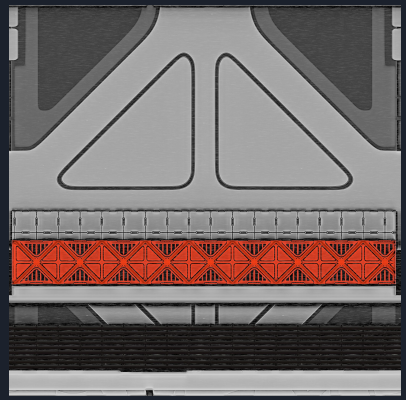
- Unity is not always able to static batch meshes.
- Use the Frame Debugger to inspect which draw calls were batched, and which weren't
- Multiple possible reasons why static batching can fail :
  - Objects affected by different materials
  - Objects affected by different lightmaps
  - Objects affected by multiple forward lights
  - Objects affected by multiple Reflection/Light probes
  - Static batching is limited to 64k vertices per batch
  - etc.



Atlasing/ Trimsheets

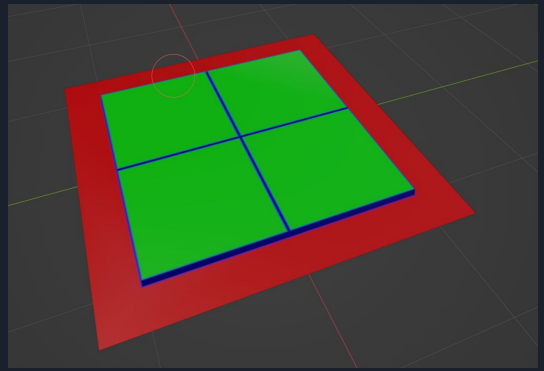
- Great way to “combine” materials
  - Can fix the issue “Objects affected by different materials”
- Top-right Atlas made by Maebbie (VRChat world creator), used in the world ”Eventide”
- Requires to update the UVs of the mesh
- Doesn’t work well with some shader features, like Stochastic Texturing
- https://vrclibrary.com/wiki/books/maebbies-precise-solutions/page/how-to-atlas-everything-in-your-world-literally

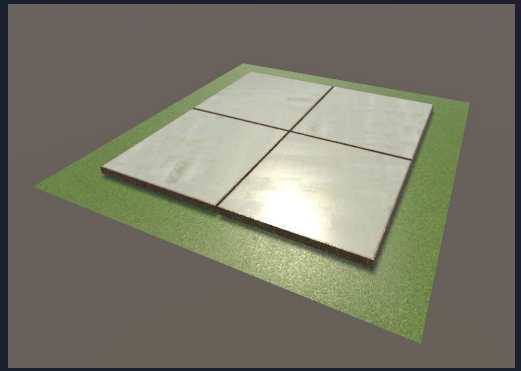




Another way to reduce material counts...

- Working with custom shaders
- Example of a shader that uses vertex colors to blend between different textures
  - Single material to render 3 textures
  - Easier to UV, “Smart UV Project” can be good enough
  - But requires to vertex paint each vertice

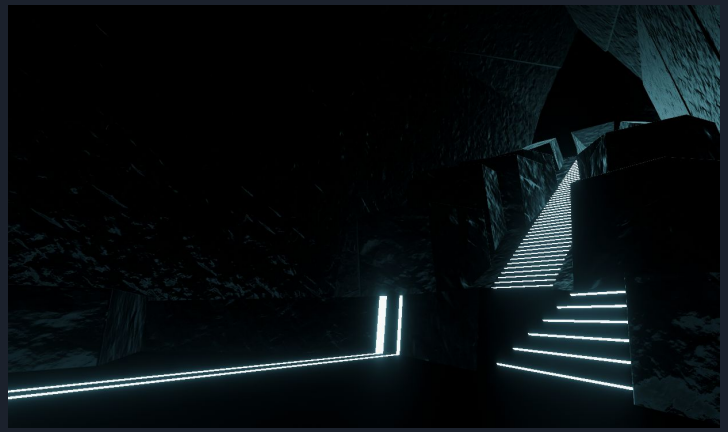




Light baking

- Generates a lightmap containing all the lighting infos of a scene
- Can fix the issue “Objects affected by multiple forward lights”
- Reduces real-time lights → fewer draw calls :
  - No need to generate shadow maps
  - No extra ForwardAdd passes for baked lighting
- And it usually looks so much better!

That doesn’t mean that RT light should be banned, RT lights can look really good also!



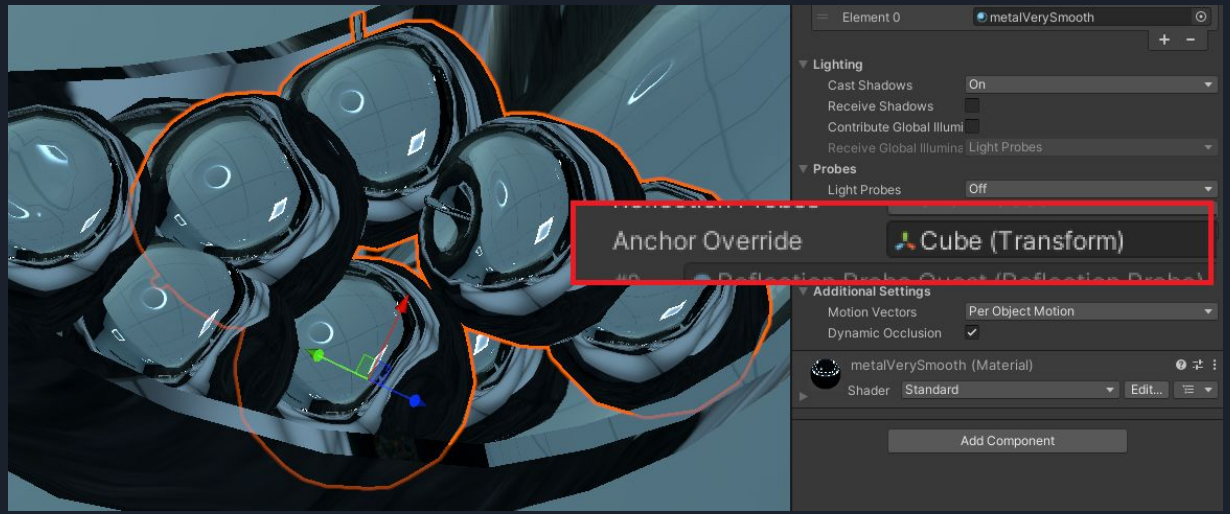
Light baking

Too many lightmaps can also break batching, which shows as “Objects affected by different lightmaps”, a few ways to fix that issue :

- Higher resolution lightmaps
  - Requires a really good GPU with the Unity lightmapper, not really required with Bakery.
- Lower texel density
  - "Bicubic Lightmap Sampling" can help improving the look of some pixelated lightmaps (see images below), supported by many shaders like Mochie Standard
- Grouping areas on the same lightmap using the « Bakery Lightmap group » component
  - Forces Bakery to put the entire group on a single lightmap
  - Resolution is dynamically adjusted
  - Improves lightmap organisation

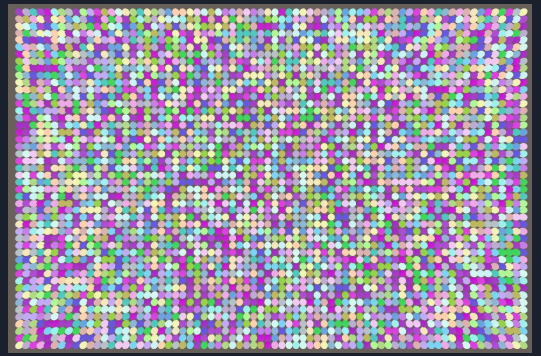
Anchor override

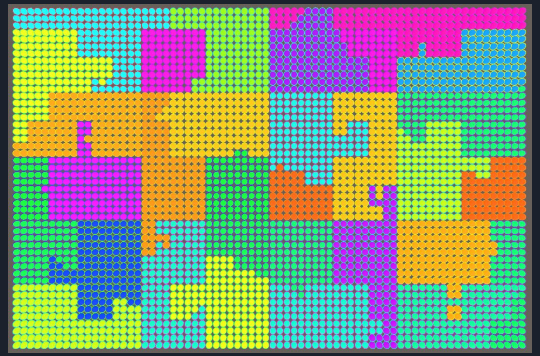
- Can fix the issue “Objects affected by multiple Reflection/Light probes”
- Forces objects to share the same probe data (light probe, reflection probe)



Static Batching

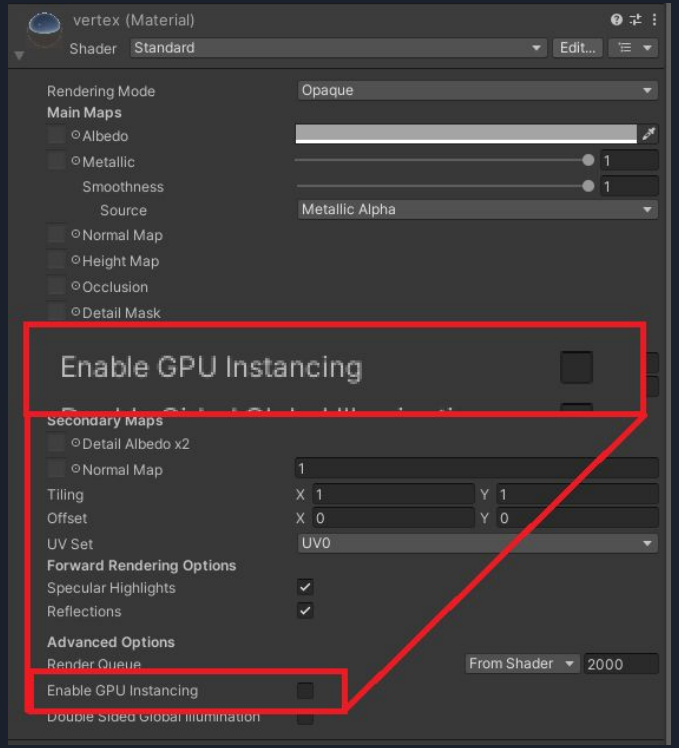
- Unity doesn’t always do a good job at static batching objects
  - Editor tools can improve it !
- https://github.com/StressLevelZero/CustomStaticBatching
- Pictures below show the tool in action
  - Before (first picture): Unity batches groups of object from seemingly random positions
  - After (second picture): Distance-based static batching : Meshes are nicely grouped





GPU Instancing

- Great if we want to render the same object many times
- Basically tells the GPU "*Render this object at position a, b, c…*" in a single draw call!
- Can be turned on in the shader
- Compared to static Batching :
  - Doesn't generate a new mesh
  - Works well with vertex shaders
  - No real polygon limit, but up to 1023 objects can be rendered in a single batch
  - Objects can move !

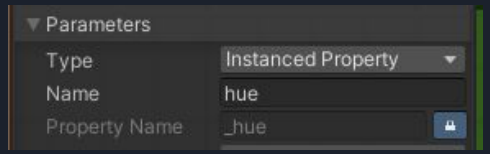


Q : Can I GPU Instance meshes with slight material variations?

A: Yes, with Instanced properties!

Instanced properties

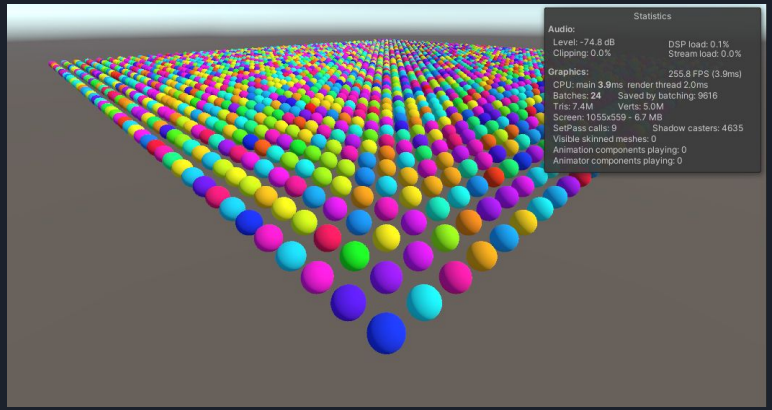
- Step 1 : Setup your shader to use Instanced properties (ASE example)



- Step 2 : Set the property via script using a material property block

```
MaterialPropertyBlock mpb = new MaterialPropertyBlock();
mpb.SetFloat("_hue", Random.Range(0.0f, 1));
meshRenderer.SetPropertyBlock(mpb);

//This line doesn't work, it breaks batching!!
meshRenderer.material.SetFloat("_hue", Random.Range(0.0f, 1));
```



Another idea to GPU instance variations....

- A shader that changes the color and the rotation of the mesh based on it's position
- No need to work with MaterialPropertyBlocks here!

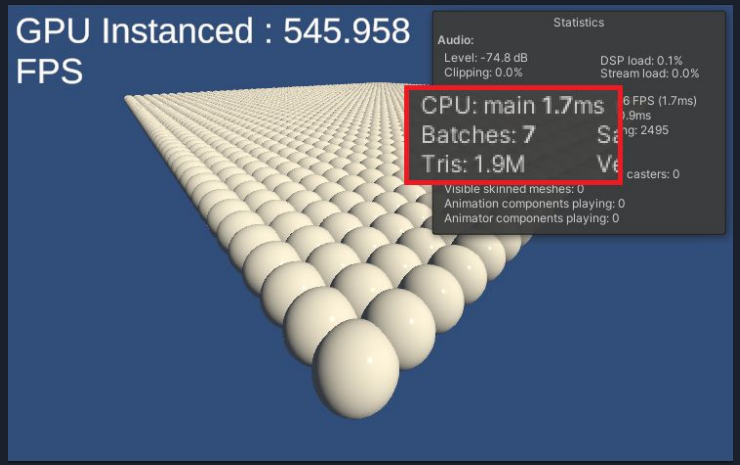
Q : What is more performant, GPU instancing or Static Batching?

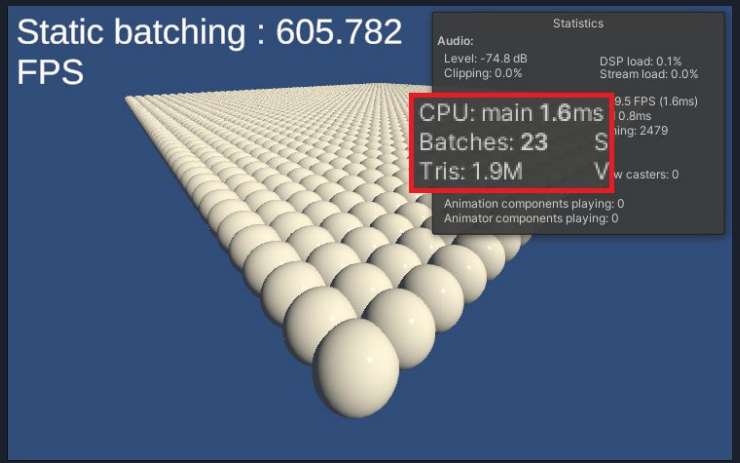
A: It depends!

And I think it's not really the right question to ask

Static batching vs GPU Instancing

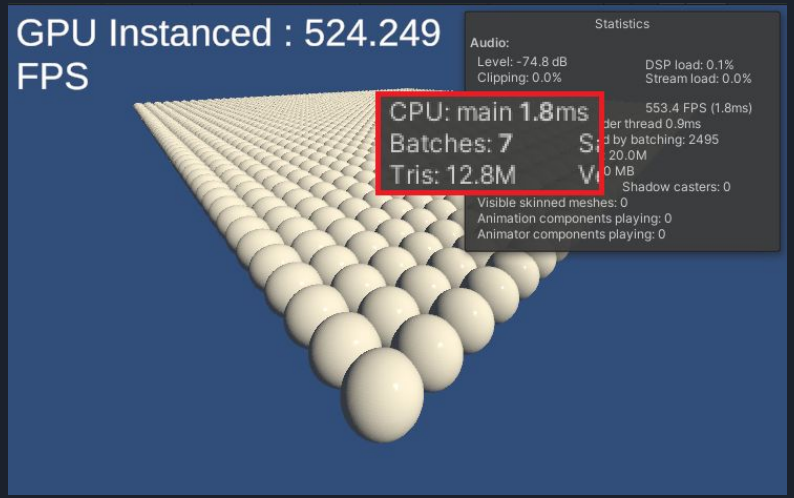
- Low polygon count.

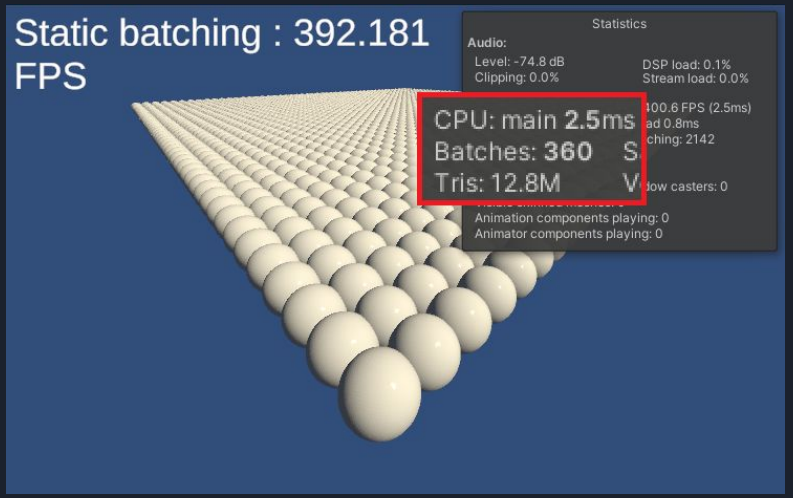




Static batching vs GPU Instancing

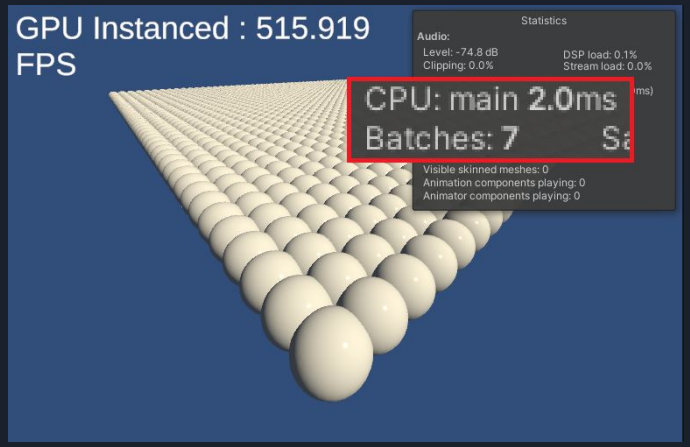
- higher polygon count.

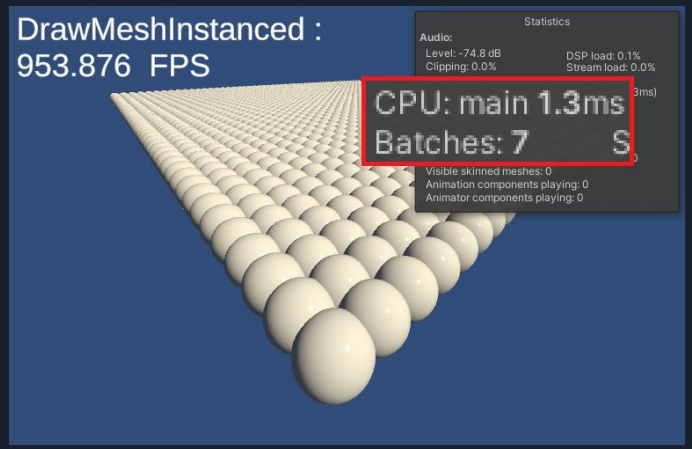




Graphics.DrawMeshInstanced

- Lets you manually perform GPU Instancing via script
- Can offer better performance than automatic instancing
- But it's painful to setup
- Not really worth it inside VRChat, because of the Udon overhead





Graphics.DrawMeshInstanced

```
public Mesh MeshToRender;
public Material MaterialToApply;
public int AreaSize;

private List<Matrix4x4> _matrices = new List<Matrix4x4>();
private const int INSTANCE_LIMIT = 1020;

Unity Message | 0 references
void Start() {
    // We are setting here a list of positions, rotations and scales (as TRS matrices) where we
    // want to render our GPU Instanced meshes
    for (int x = 0; x < AreaSize; x++) {
        for (int z = 0; z < AreaSize; z++) {
            Matrix4x4 matrix = Matrix4x4.TRS(
                new Vector3(x * 0.7f, 0, z * 0.7f), //Position
                Quaternion.identity, //Rotation
                Vector3.one //Scale
            );
            _matrices.Add(matrix);
        }
    }
}

Unity Message | 0 references
void Update() {
    // Rendering our meshes
    for (int i = 0; i < _matrices.Count; i += INSTANCE_LIMIT) {
        int count = Mathf.Min(INSTANCE_LIMIT, _matrices.Count - i);
        Graphics.DrawMeshInstanced(MeshToRender, 0, MaterialToApply, _matrices.GetRange(i, count));
    }
}
```

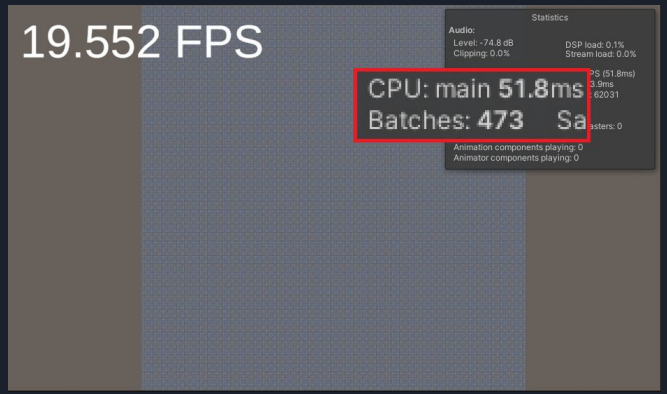
Static batching vs GPU Instancing

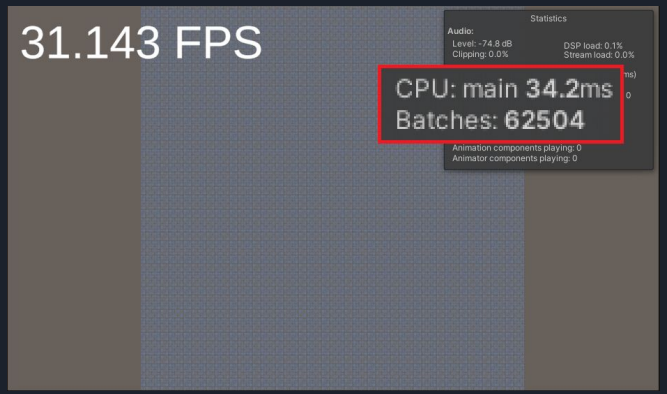
Summary :

- GPU Instancing :
  - Great to render hundreds of similar objects, even better for high poly meshes
  - Great for meshes that use vertex shaders
  - Great for objects that can move
  - ~1000 Instances/batch
  - Less overdraw (see “Overdraw” chapter)
- Static batching :
  - Great to batch static objects that share the same material
  - A bit more performant than GPU instancing on lower poly meshes
  - ~64,000 vertices per combined mesh

Dynamic Batching

- Similar to static batching, but it is done at runtime
- Not recommended by Unity : On modern systems, Dynamic Batching overhead > Draw call overhead
- That overhead gets worse when the poly count per mesh is higher
- Use GPU Instancing or Static batching instead!





Culling

Culling

Batching helps, but it's not always enough

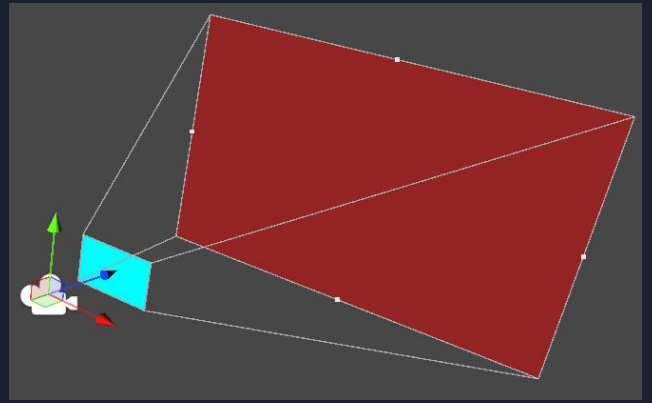
- Batching reduces draw calls by combining objects
- But we still need to avoid rendering what we can’t see

Culling: Skip what doesn’t need rendering

- Culling reduces draw calls by skipping objects not visible on screen
- Three main culling methods:
  - Frustum Culling
  - Occlusion Culling
  - Manual Culling

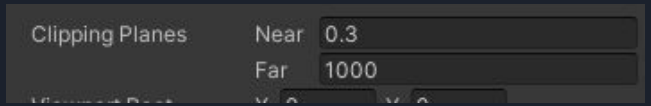
Frustum Culling

- Automatically done by Unity
- Automatically skips objects outside the camera's view
- Does not cull objects hidden behind other objects



far-clipping plane (Red) and the near-clipping plane (cyan):

- Set in the Camera settings
- Lowering the far clipping plane can reduce draw calls
  - Nice to cull far-away VIP areas
- Per-layer culling distances possible via "Camera.layerCullDistances"




```
float[] distances = new float[32];
distances[4] = WaterCullingDistance;
distances[5] = UICullingDistance;
CameraInstance.layerCullDistances = distances;

//On VRChat
VRCCameraSettings.ScreenCamera.layerCullDistances = distances;
```

Occlusion culling

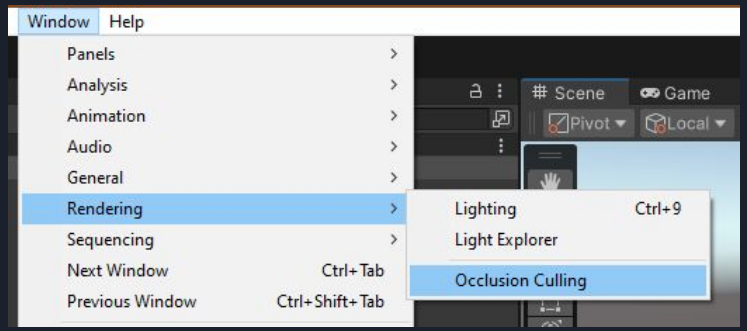
- Skips rendering objects hidden behind other objects
  - Reduces GPU load and draw calls, especially in dense scenes
- Unlike Frustum Culling, it checks visibility
- Best for static environments with lots of walls

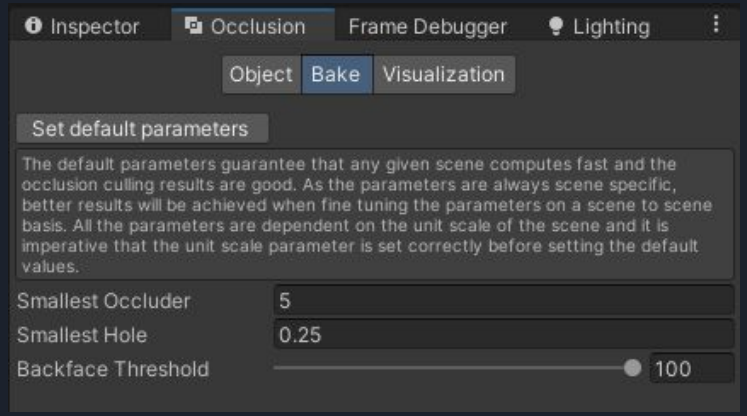
Setup :

- Windows > Rendering > Occlusion Culling
- Mark Occluders (objects that block others) as Occluder Static
- Mark Occludees (objects that can be hidden) as Occludee Static
- Bake!

Limitations :

- Increases build size
- Not efficient in open worlds, where objects don't usually hide other objects





Manual culling

- We can toggle on and off areas manually via script
- Be careful with stations, including avatar stations !
  - Stations turn off player colliders, meaning that certain events (OnPlayerTriggerEnter/Exit) won't be triggered
  - Stations allow players to bypass trigger events
  - Avatar stations can break a lot of game worlds !
  - Workarounds :
    - Distance-based object toggle
    - Implementing custom player colliders

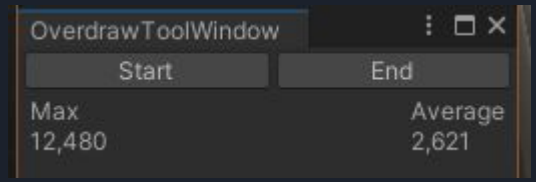
Overdraw

Overdraw

- Happens when the same pixel is drawn multiple times per frame
  - Even with fully opaque shaders
- Common in scenes with overlapping transparent or stacked objects
- Not always avoidable, but should be minimized where possible
- Impacts GPU performance, unlike draw calls (which hit the CPU)

Nordeus's OverdrawMonitor:

- Presented at Unite 2017 https://www.youtube.com/watch?v=vJZcbscZ4-o
- https://github.com/Nordeus/Unite2017/tree/master/OverdrawMonitor
- Great tool to check for overdraw
- DirectX only



Overdraw simulation

Let’s render this scene in two different ways

- Wall first
- Avatar first

Overdraw simulation - Wall rendered first

First, the wall gets rendered

Then the avatar, some parts of the wall get re-drawn

In total, some pixels got re-rendered twice!

We have some overdraw

Overdraw simulation - Avatar rendered first

First, the avatar gets rendered

Then the wall, but not the entire wall! Since an avatar is in front of it.
Less pixels get textured.

In total, less pixels got textured

Less overdraw

Overdraw simulation - Summary

Summary :

- When a pixel is about to be rendered
  - The GPU checks if another pixel is already in front (closer to the camera)
  - If so : the pixel is thrown away
  - If not : the pixel gets shaded (in the pixel/fragment shader)
- Key Point: This check happens before the expensive shading process
- Closer pixels should be rendered first
  - Helps reduce overdraw and improve performance

Note : This check (called Early Z-Test) doesn't always happen, for instance Cutout shaders (in that case, the Z-Test happens after the shading process)

Render order matters!

How to reduce overdraw?

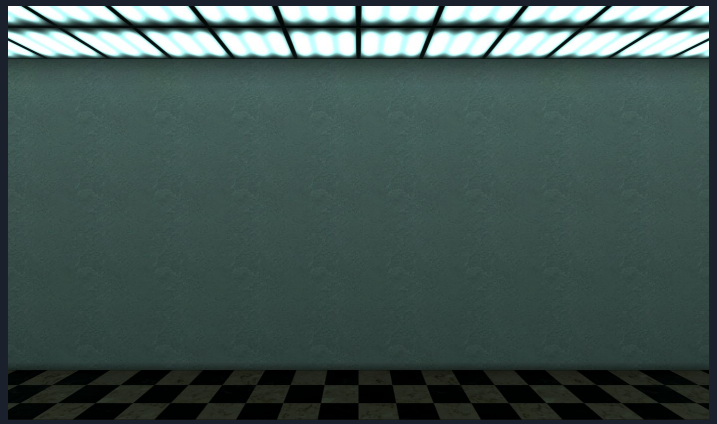
- By sorting objects front-to-back. Thankfully, Unity already does it!
  - Ensures further pixels get discarded early, before the shading process
  - Skybox should be rendered at the end (render queue 2500)
  - Then, transparent materials (render queue >2500)
    - So transparent materials overdraw
- The GPU doesn't sort pixels on its own. So, Unity handles sorting on the CPU side, before sending those draw calls to the GPU.

How Unity sorts draw calls :

- Unity roughly sorts draw calls front-to-back, based on the distance between the camera and the bounding box center
- Not perfect! But it's fast and usually good enough
- Example of overdraw caused by bad sorting: A player standing in the middle of the room, close to the bounding box center
  - Unity : "The room-object is the closest to the player, so it needs to be rendered first!"
  - We have overdraw here

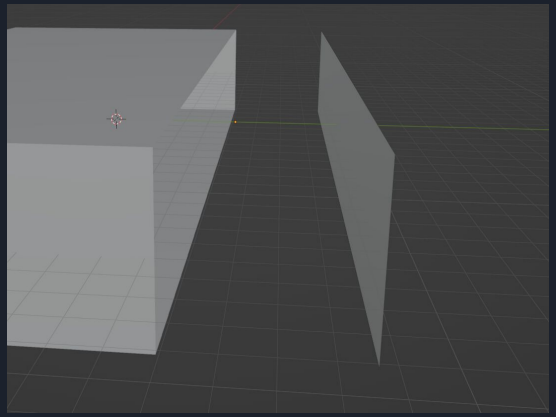
Overdraw example

- A room with a picture frame
- Issue : The picture frame renders over the wall (Overdraw),
  - The camera is close to the bounding box center of the room
  - Front-to-back sorting
    - Room is rendered first, then the picture frame (first and second picture)



Fixing overdraw : Solution 1

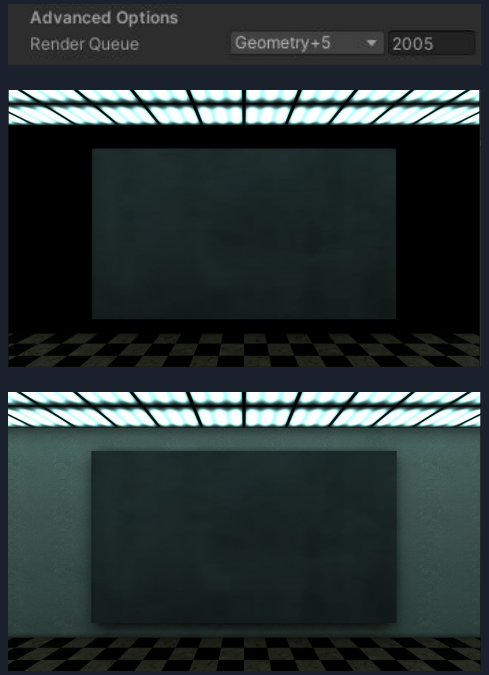
- Separating the wall from the rest of the room
  - 3 separate objects
- Turning off batching on the wall
  - Ensures the wall doesn't get merged back with the room
- Now we have 3 separate bounding boxes:
  - Main room
  - Back wall
  - Picture frame



Easier for Unity to sort!

Fixing overdraw : Solution 2

- Way easier solution in this case!
- Increasing render queue of the wall material (2000~2499) to ensure it gets render later
- Done!
  - First picture : Picture frame gets rendered first
  - Second picture : Wall gets rendered later



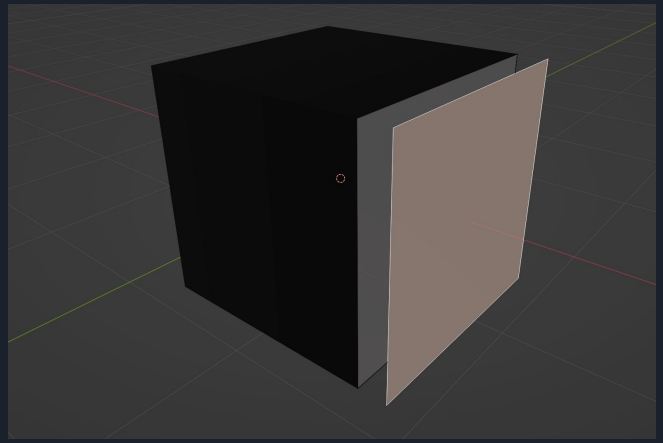
Hidden geometry

To test the performance of hidden geometry:

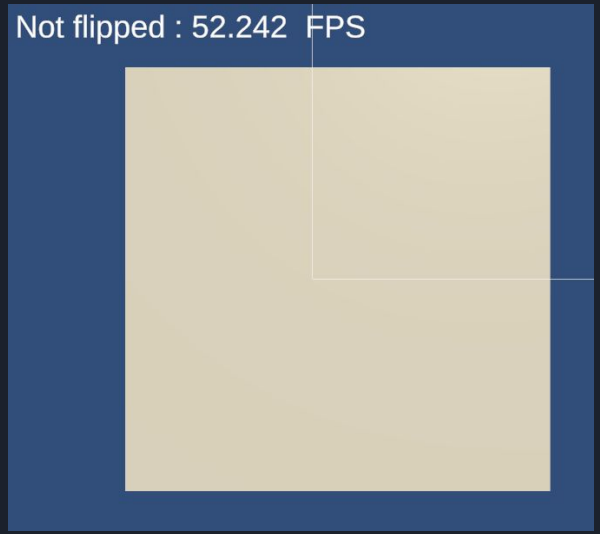
- “Array” modifier applied on a plane
- 10000 overlapping planes

I made two versions :

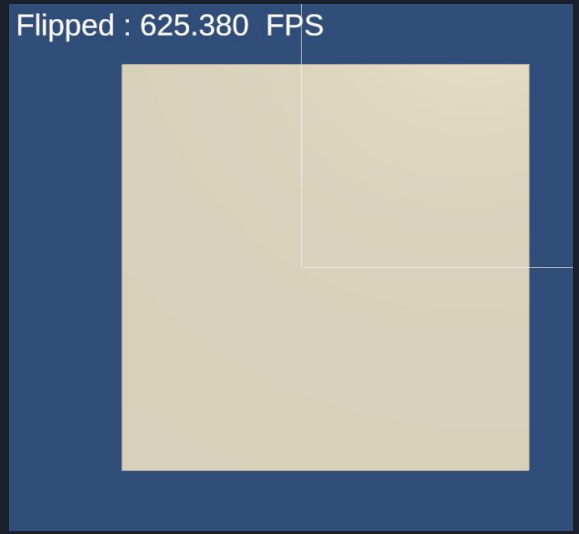
- One with flipped normals
- One without



Hidden geometry



Original mesh

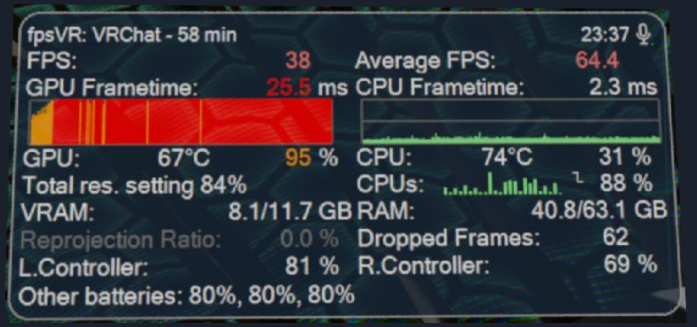


Mesh with flipped normals

Hidden geometry - Explanation

In the demo:

- The first mesh rendered polygons back-to-front
- The second mesh rendered polygons front-to-back
- So the second mesh performed much better.
- We cannot really control the polygon rendering order.
- hidden geometry costs performance!



How to avoid hidden geometry:

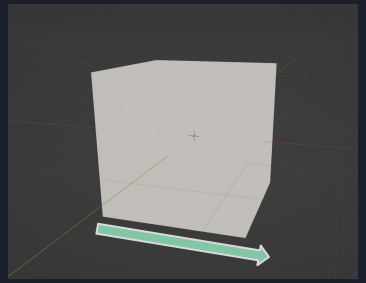
- Modify the mesh in Blender to delete hidden faces
- Avoid using cutout shaders to hide parts of a mesh, as it won't fix the problem
  - Instead, hide geometry before the pixel gets shaded, so in the vertex shader (example : Poiyomi's UV-Tile Discard system)
- Another solution explained the next slide…

Hidden geometry

If we separate the “overdraw test object” into multiple meshes:

- Unity will sort each sub-mesh front-to-back (If static batching is disabled)
- Polygons would still be rendered back-to-front, but it won’t be as bad
- Only the first mesh would cause some overdraw issues (if we look at the mesh from the front)

- Example on the right : Mesh separated into 5 meshes
- Possible use-case : Player inside a train or an airplane looking at seats
  - Huge amount of overdraw if all seats were a single mesh



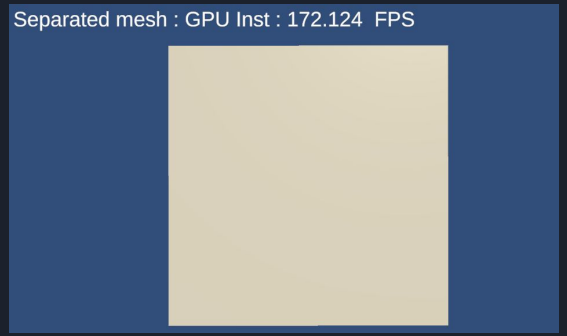
Hidden geometry and static batching

Static batching “merges” meshes :

- front-to-back sorting order is not always respected during the merge
- Static batching can cause overdraw

GPU Instancing is more intelligent : Front-to-back sorting is respected, because Unity knows at runtime where the player is.

- Example : Train seats, each seat could be GPU Instanced to minimize overdraw.



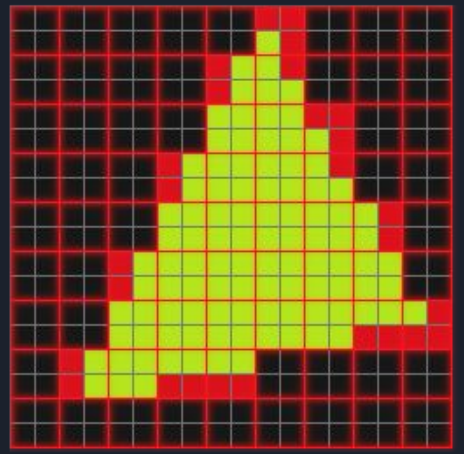
LOD (Level of details)

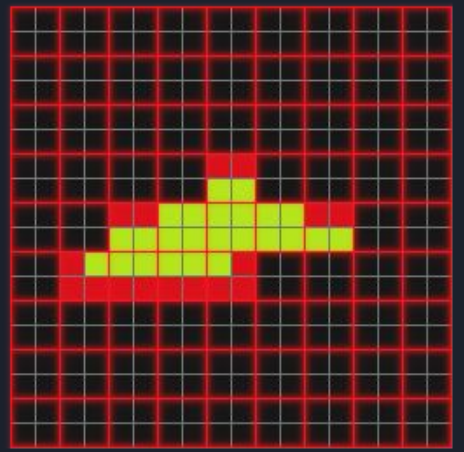
LOD (Level of details)

Issue:

- GPUs render polygons in 2×2 pixel quads (blocks of 4 pixels)
- If a polygon (green) only covers part of a quad (red), the GPU still processes the full quad
- Pixels outside the polygon (green) are discarded (red), but the work is still done
- Visualization:
  - Red = rendered quads
  - Green = pixels actually kept
- The smaller the polygon, the worse the pixel kept / pixel discarded ratio

Polygon size matters!

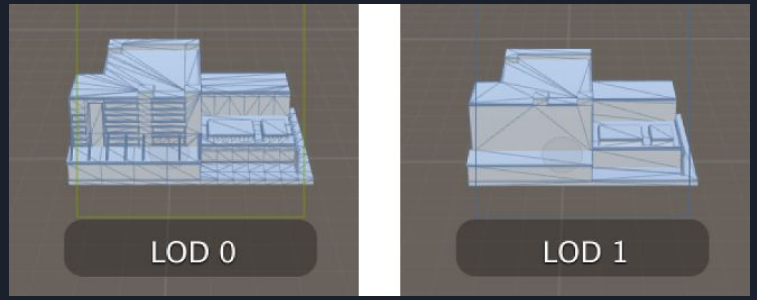


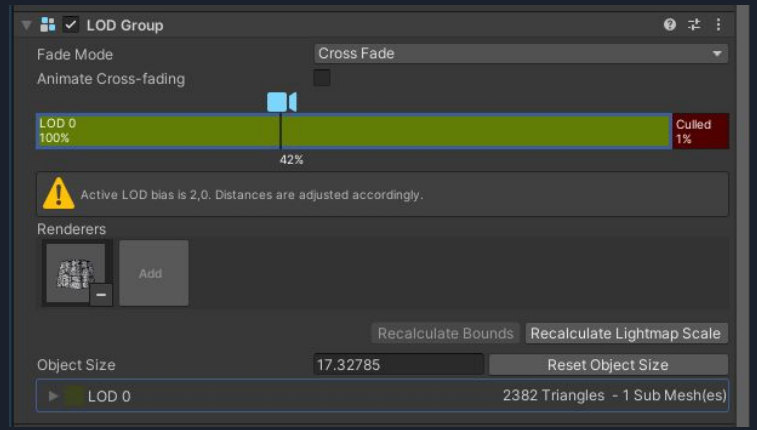


LOD (Level of details)

Solution : LODs

- Reducing polygon count on further away objects
- Maximize polygon size
- LODs can be generated using editor tools
- Can also cull objects
  - Easy to setup, no tool required
- Ideas:
  - Disabling shader features on further away objects
  - Culling smaller objects sooner
  - Transparent to opaque
  - Reduce poly count on tree leaves (cutout)
  - Billboards

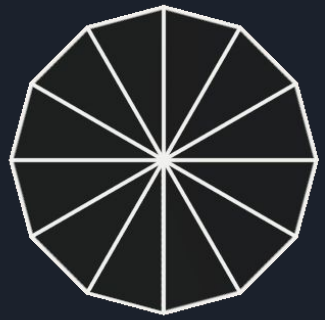




Efficient LOD topology

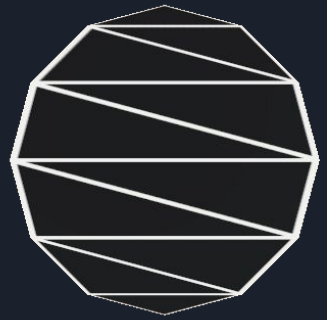
- Avoid small polygons
- Different triangulations can affect performance
- Benchmark below done with 360 copies of that mesh
- How a mesh gets triangulated affect performance!

12 polygons



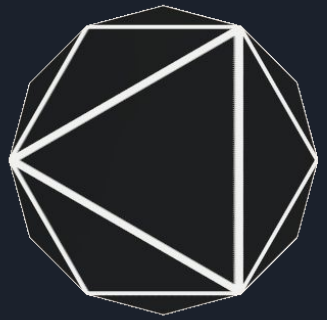
681 FPS

10 polygons



696 FPS

10 polygons



714 FPS

Impostors

Impostors

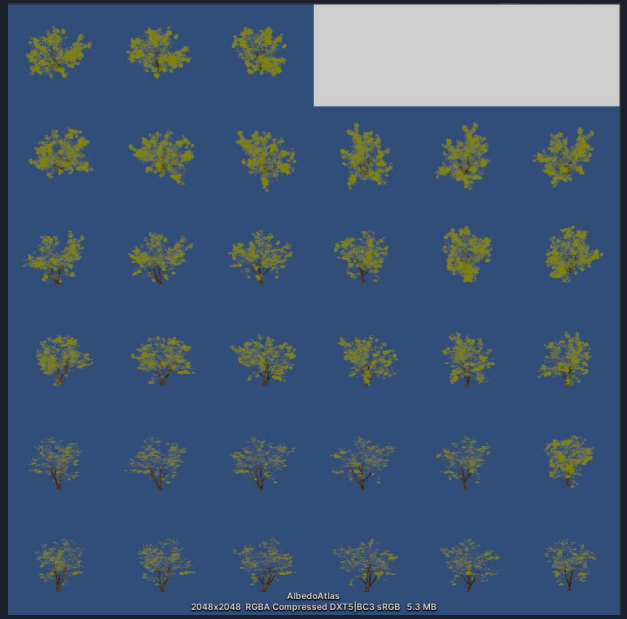
- Basically “fancier billboards”
- Great to render far-away objects

How it works :

- The original mesh is scanned from multiple angles
- Generates an atlas showing the object from different views
- Custom shader picking the right texture based on camera angle

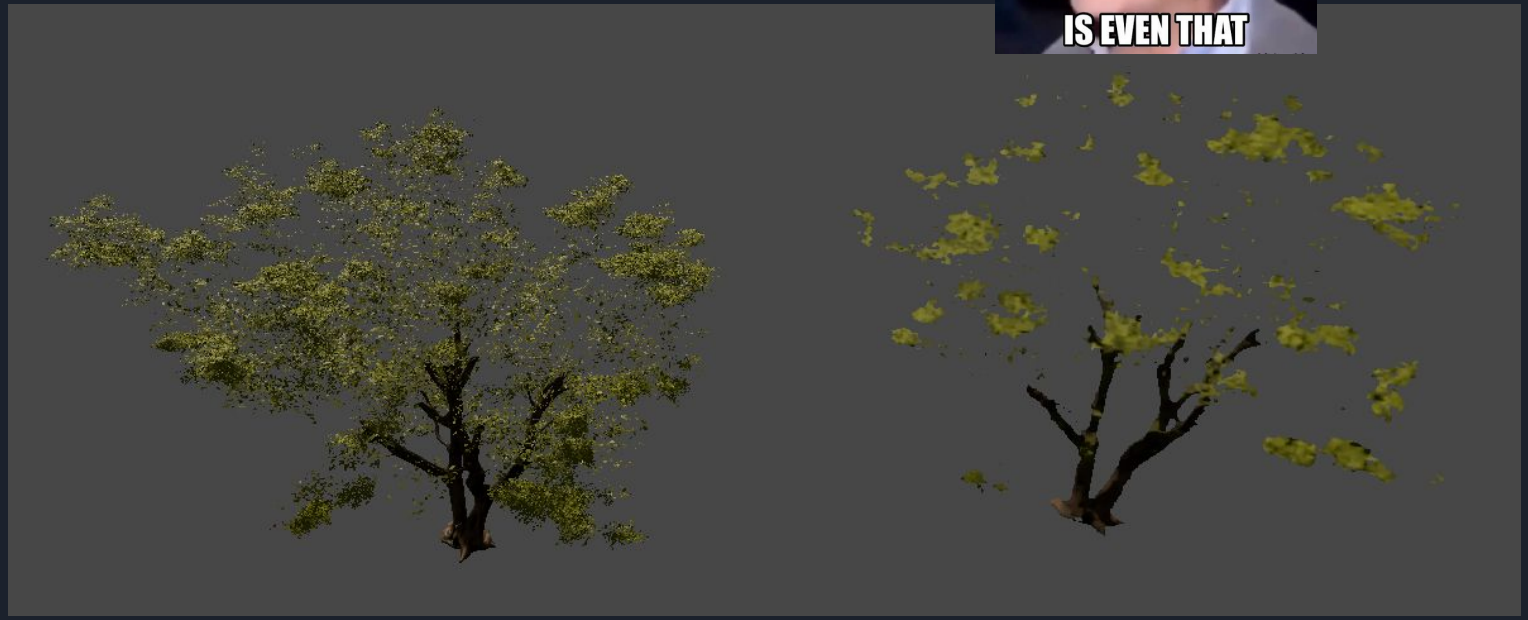
Drawback :

- Requires an editor tool to generate them



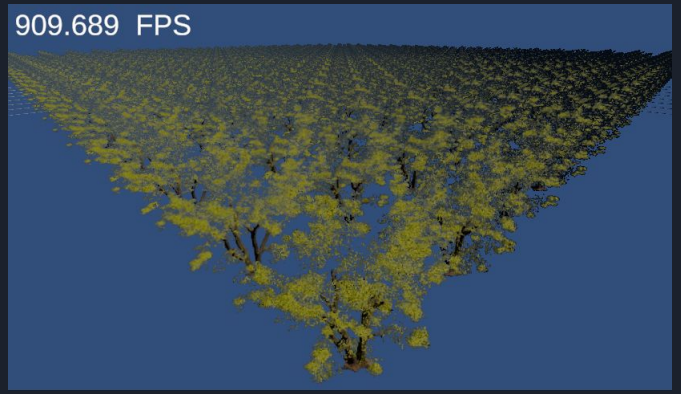
Impostor result

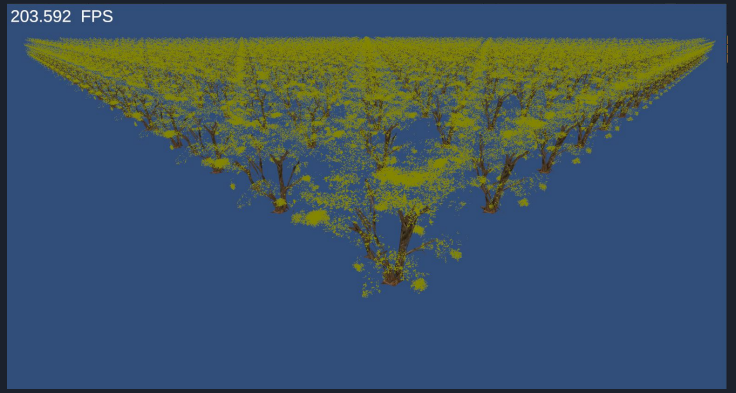
Impostor result

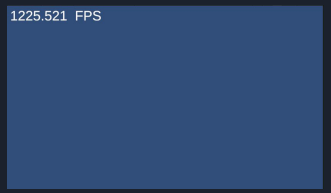


Impostor benchmarks with 400 instances

Impostors on the left, regular mesh on the right







Optimizing Udon code

Optimizing Udon code

- Udon basics : Excellent talk by Happyrobot33 : “Furality Umbra - Udon Development: From First Steps to Optimization” https://www.youtube.com/watch?v=qjFZ6ggEkJc
- Udon is very slow
  - About 500~1000 times slower than regular C#

Optimizing Udon code means :

- Run less Udon code per frame.
  - Avoid heavy use of Update() loops, even worse FixedUpdate(), prioritize custom event loops instead
  - Sometimes, heavy tasks only need to be executed by the instance master.
- Minimize networking overhead
  - Less networked objects in the scene
    - Use “None” as the sync method for non-networked objects
  - Reduce synchronization frequency

Cameras

- Cameras re-render the scene, even if the player isn’t directly looking at the rendered image
  - More draw calls, more work on the GPU
- Idea :
  - Decrease framerate of additional cameras.
  - disable cameras when they aren’t used

```
/// Renders a camera view at a given framerate
public Camera CameraInstance;
public float Framerate;

Unity Message | 0 references
void Start()
{
    CameraInstance.enabled = false;
    CustomLoop();
}

2 references
private void CustomLoop()
{
    CameraInstance.Render();
    SendCustomEventDelayedSeconds(
        nameof(CustomLoop), 1.0f / Framerate
    ); // Use "Invoke" outside of VRC
}
```

```
/// Can be used on the screen of a surveillance camera. the script
/// disables the camera when the player isn't directly looking at a screen.
/// Attach this script on a screen.
/// Outside of VRChat, use OnBecameVisible and OnBecameInvisible instead
public Camera CameraInstance;

0 references
public void _onBecameVisible()
{
    CameraInstance.gameObject.SetActive(true);
}

0 references
public void _onBecameInvisible()
{
    CameraInstance.gameObject.SetActive(false);
}
```

VAT (Vertex animated textures)

How can we render thousands of animated meshes?

Example : A huge crowd

Issues :

- Skinned mesh renderers cannot be batched
- Animated on the CPU (Unity) with an animator
  - Expensive CPU tasks

Solution :

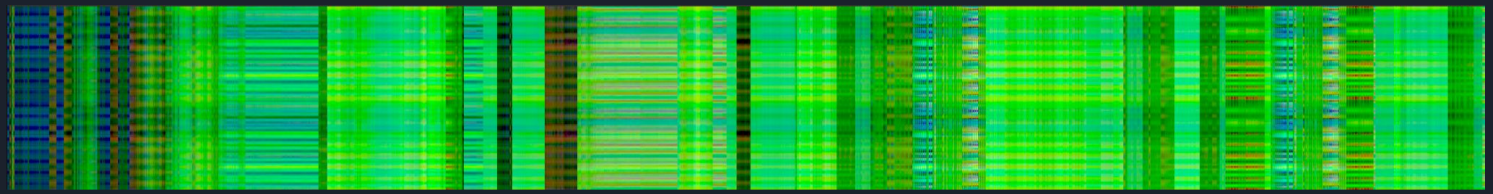
- Using mesh renderers instead
- Animating on the GPU using a vertex shader

Drawback :

- Requires an editor to convert all animation files into a VAT
- Less flexible, harder to customize the animations

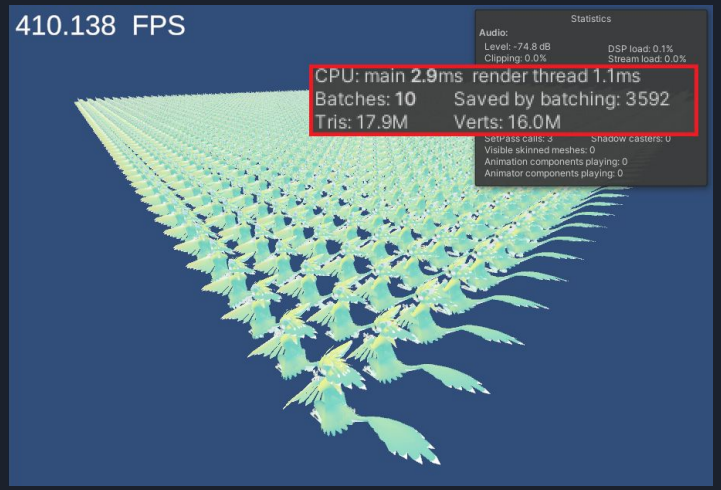
VAT Baker

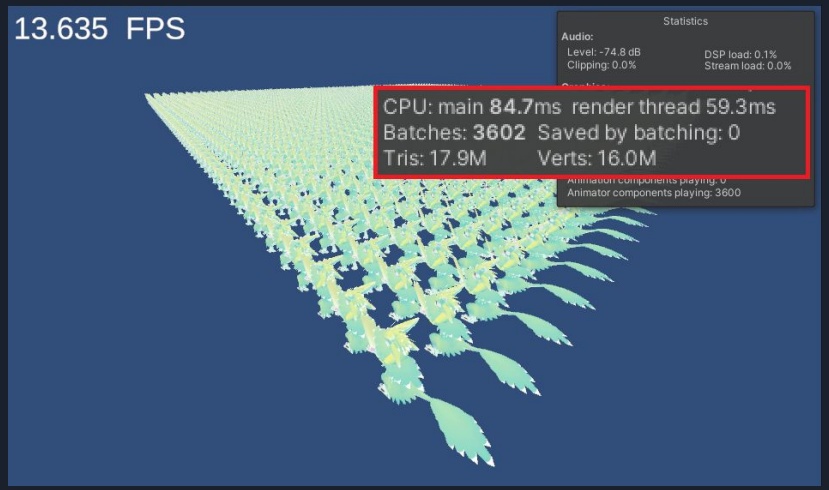
- https://github.com/fuqunaga/VatBaker
- Converts animation files into a texture (example below)
- Usual limitations
  - Polygon count limit
  - Texture cannot be compressed
    - Increases build size



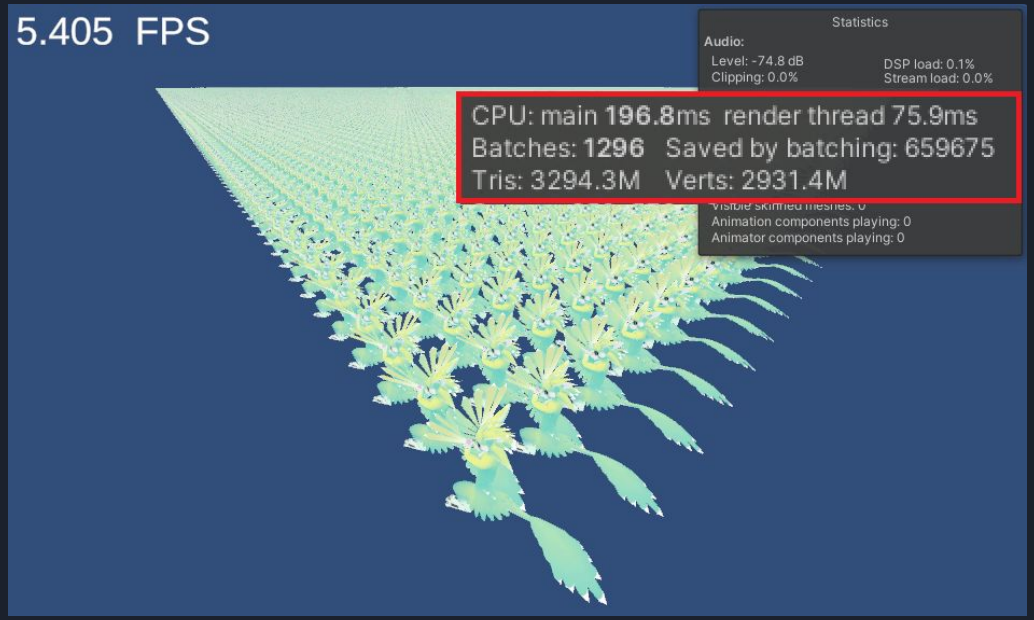
VAT vs. Skinned mesh renderers

3600 Spiri'valis





And just for fun, 660 000 Spiri’valis



Summary

Summary

- Drawcalls affect the performance on the CPU
  - Batching reduces them (Static batching, GPU Instancing)
- Bake your lighting to reduce number of RT lights and shadows
  - Which also reduces draw calls and GPU load
- Overdraw affects the performance on the GPU
  - Make sure pixels are rendered front-to-back for opaque shaders!
  - Avoid hidden geometry
- Use LODs to reduce polygon count on further away objects
- Use Occlusion culling in more enclosed spaces, doesn’t work well in more open areas
- Avoid executing Udon code every frame

Ressources

- Basics of video game graphics : https://www.youtube.com/watch?v=C8YtdC8mxTU
- SimonDev YT channel https://www.youtube.com/@simondev758/videos
- Overdraw : https://thegamedev.guru/unity-gpu-performance/overdraw-optimization/
- To Early-Z, or Not To Early-Z : https://therealmjp.github.io/posts/to-earlyz-or-not-to-earlyz/
- Atlasing everything : https://vrclibrary.com/wiki/books/maebbies-precise-solutions/page/how-to-atlas-everything-in-your-world-literally
- Some popular shaders : z3y Lit, Mochie Standard, Silent Filamented, Orels shaders
    - Once you use one of those shaders, you'll never got back to Unity standard!

Thanks for watching!

- BlueSky : @myrodev.bsky.social
- X : @MyroDev

The entire presentation can be downloaded below

SCAN ME