



Basics of optimizing VRChat worlds

By MyroP

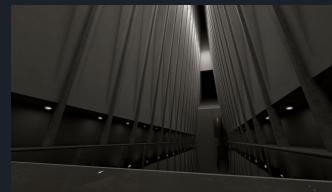
Download the presentation/benchmarks on my GitHub

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



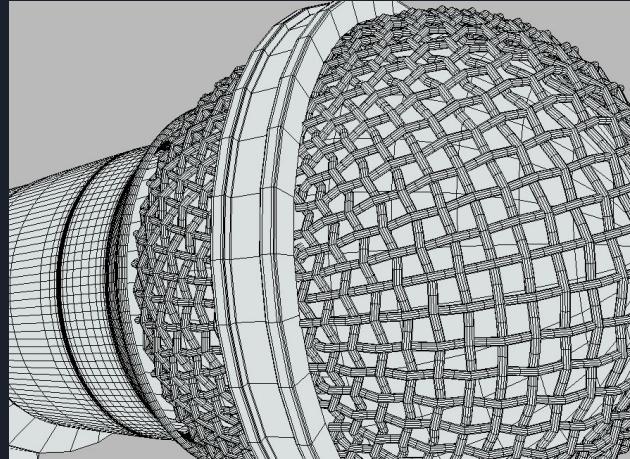
Who am I ?

- 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=1jHUY3goBu8>



Learn Low Poly Modeling in Blender 2.9 / 2.8

Imphenzia
289 k abonnés

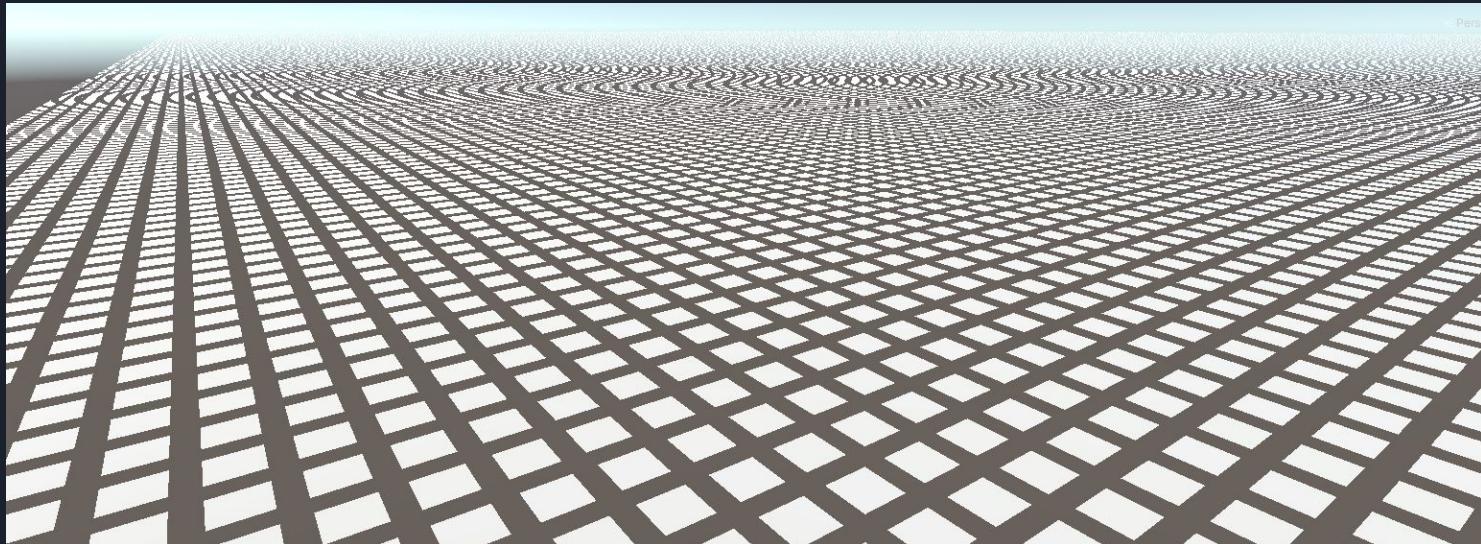
S'abonner

75 k | 440

Partager

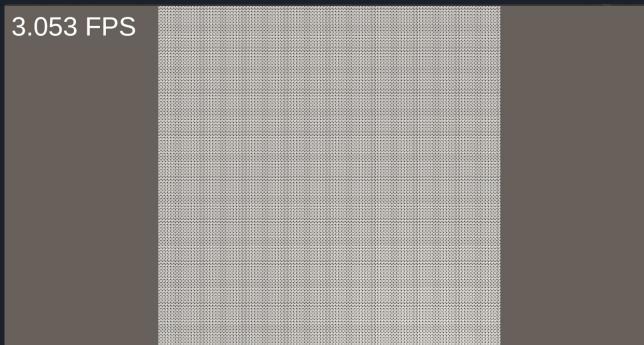
Let's start with a very basic example...

~1 million separate quads, ~2 million triangles



Let's compare the performance of two setups

Each square is a separate mesh



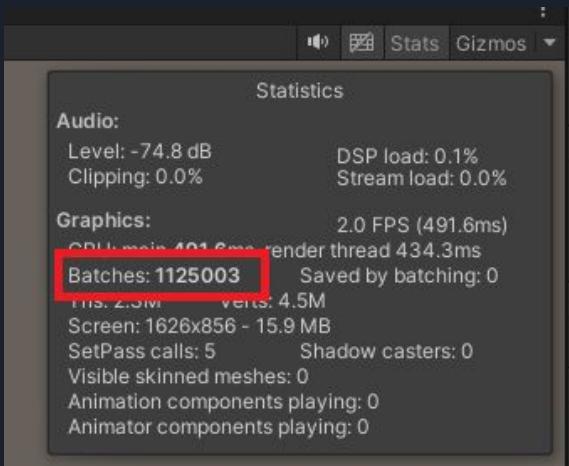
One single mesh



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 !



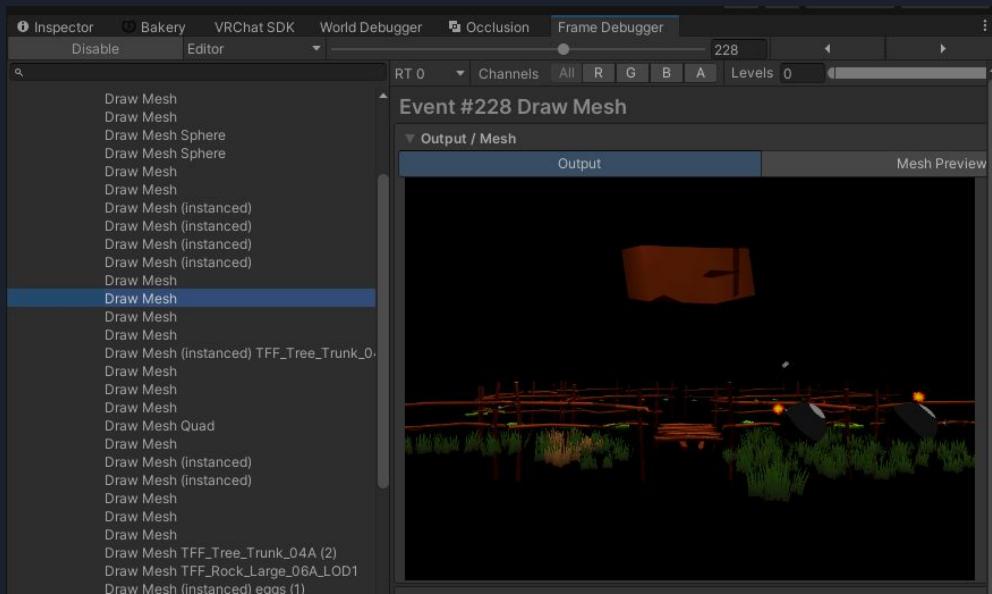
Draw calls

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!



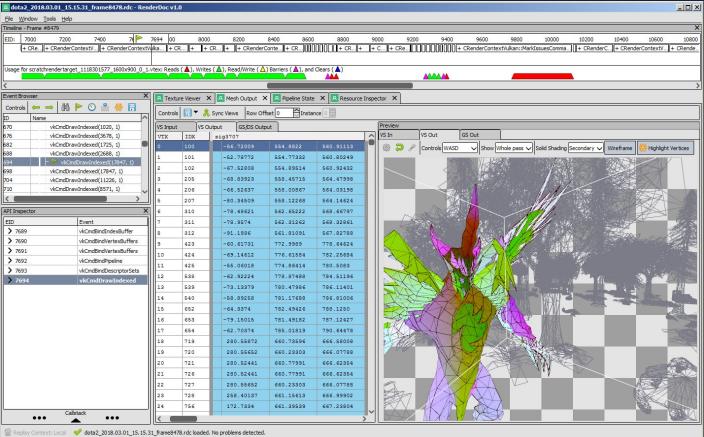
FpsVR

- Great to test performance inside VRChat.
- Shows CPU and GPU frame time
 - CPU frame time too high? Maybe too many draw calls!
- To perfectly test the performance of your world :
 - Use a very optimized avatar!
 - Be alone in the instance



RenderDoc

- Great to profile the GPU
 - Has a lot of features (analyses overdraw, detect “micro triangles” of a scene etc.)
 - Free and open source (MIT)
- Drawbacks
 - Injects DLL into the app, so it cannot be used when EAC is running
 - But it can be used in the Unity editor
 - Very technical, not easy to use





Passes and materials

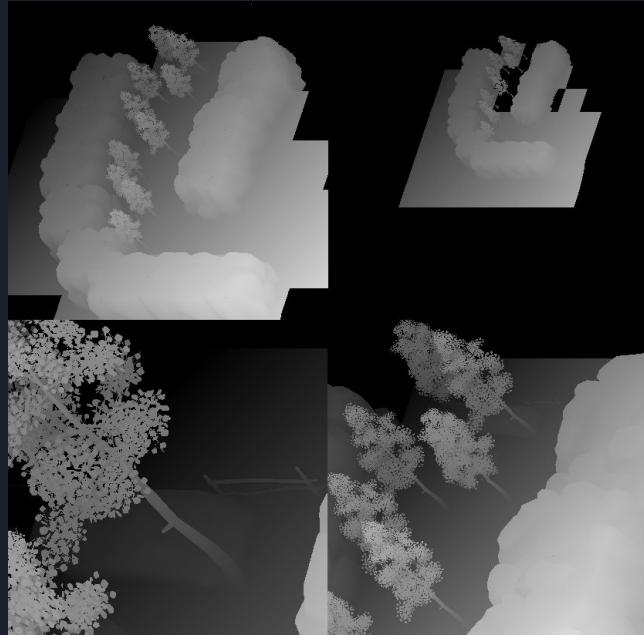
- A render pass is “one round of drawing something onto the screen”, each pass = 1 draw call.
- Shaders can define multiple passes:
 - Example : A Classical PBR shader
 - Base pass : handles base color, directional light, lightmap etc.
 - “ForwardAdd” pass : Handle RT per-pixel lights (point/spot lights) 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 materials, 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!

To reduce draw calls:

- Turn off shadow casting on some objects
 - For instance : grass doesn't always need to cast a shadow!
- Bake your lighting



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

A: With batching!



Batching

What is batching?

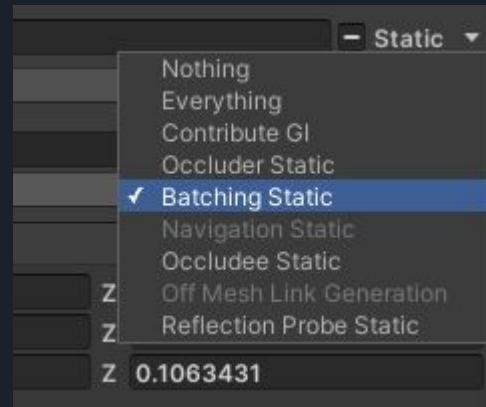
- Batching = combining multiple draw calls into a single batch.
- Goal: Improve performance.

Unity supports several batching techniques:

- Static batching
- GPU instancing
- Dynamic batching

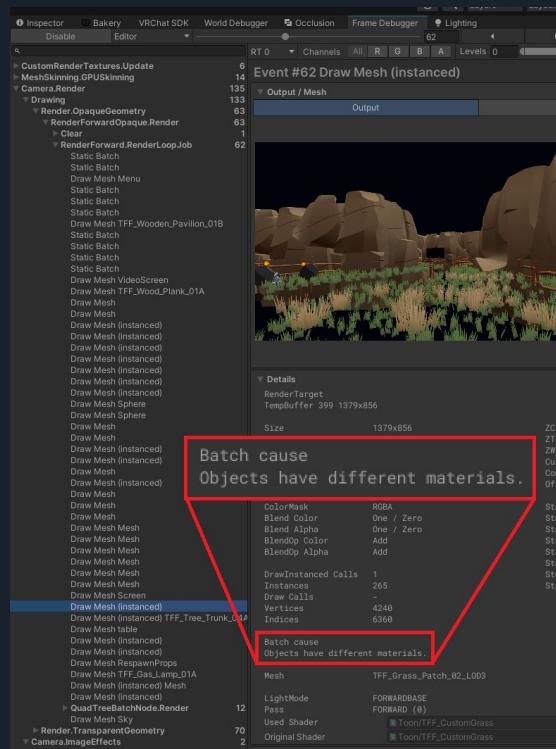
Static Batching

- 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
 - Limited to 64k vertices per batch



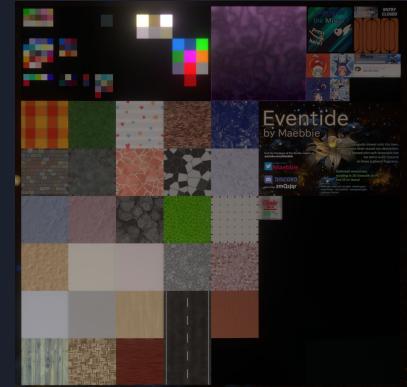
Static Batching

- 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 Reflection probes
 - etc.



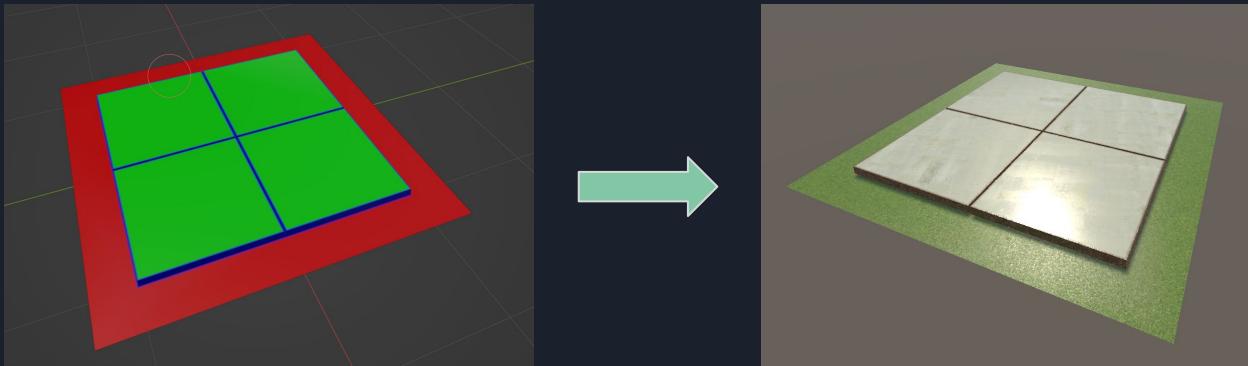
Atlasing/ Trimsheets

- 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/maebbie-s-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
- 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, but that makes lightmaps more pixelated
 - "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



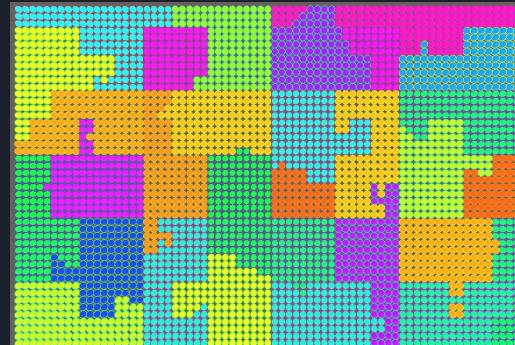
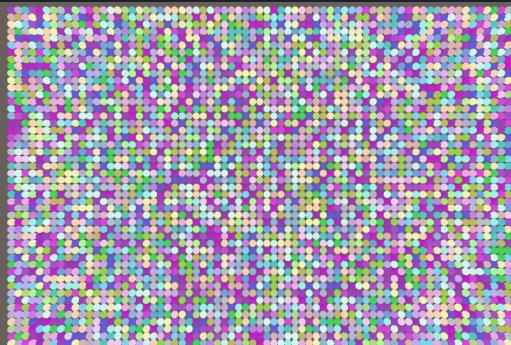
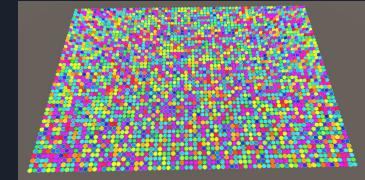
Anchor override

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



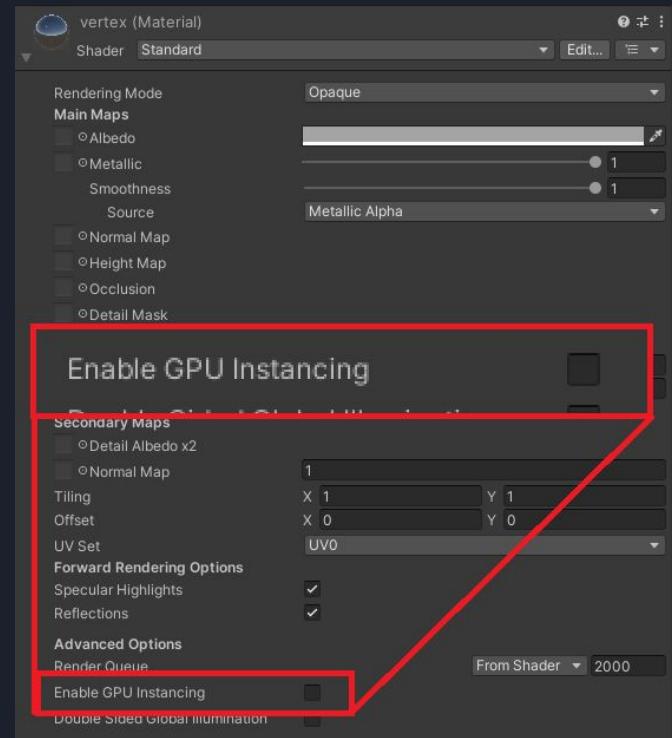
Static Batching

- 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
 - Can reduce the number of batches by reducing the number of batches in a certain area of the scene



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 !
 - But objects need to be the same!

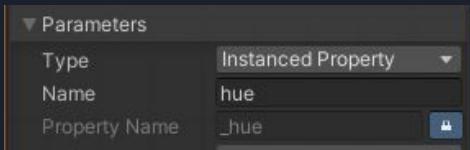


**Q : Can I GPU Instance
meshes with slight
material variations?**

A: Yes, with Instanced
properties!

Instanced properties

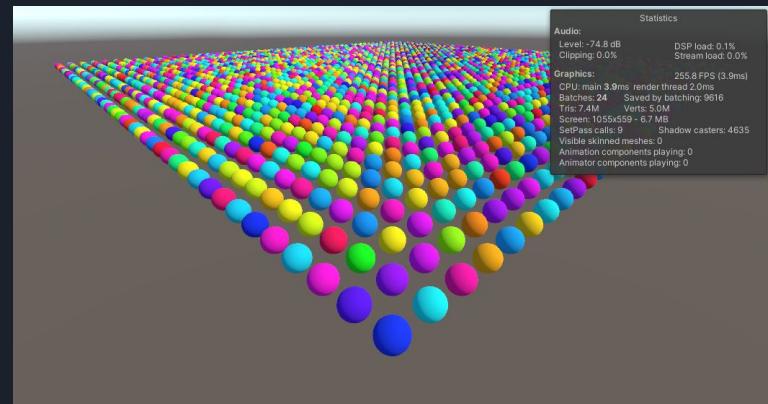
- Step 1 : Setup your shader to use Instanced properties (ASE example)



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

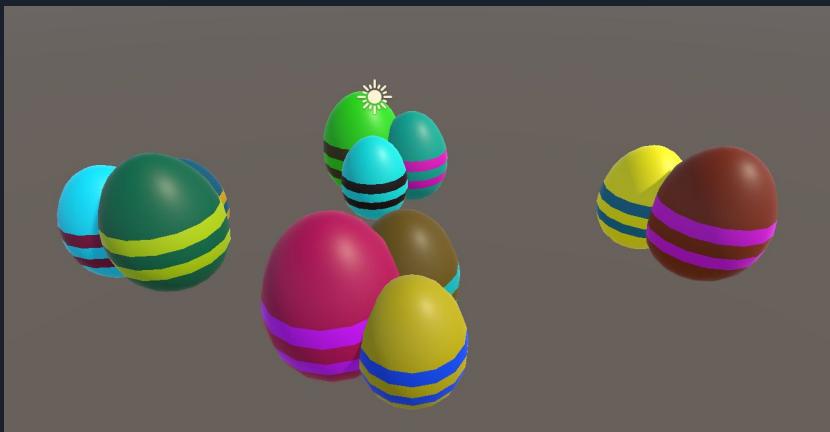
```
MaterialPropertyBlock mpb = new MaterialPropertyBlock();
mpb.SetFloat("_hue", Random.Range(0.0f, 1));
meshRenderer SetPropertyBlock(mpb);

//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!

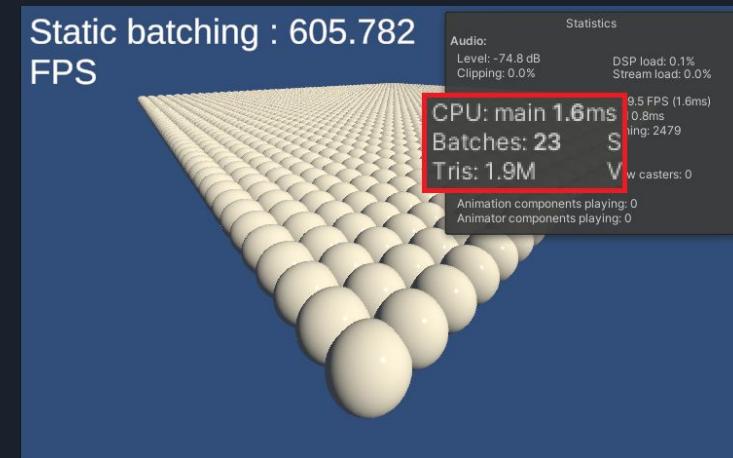
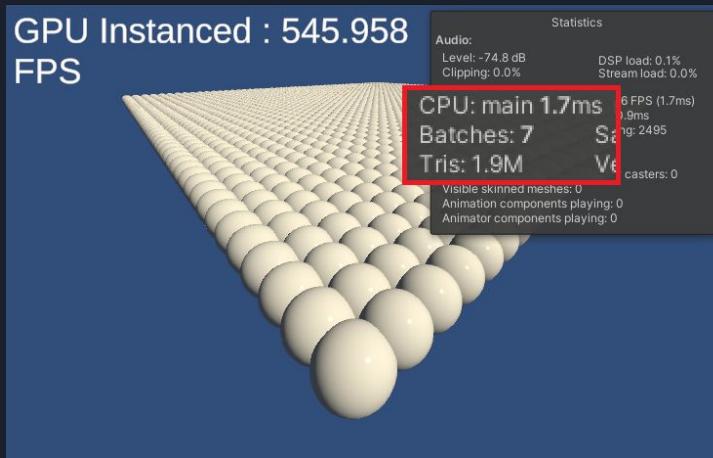


**Q : What is more
performant, GPU
instancing or Static
Batching?**

A: It depends!

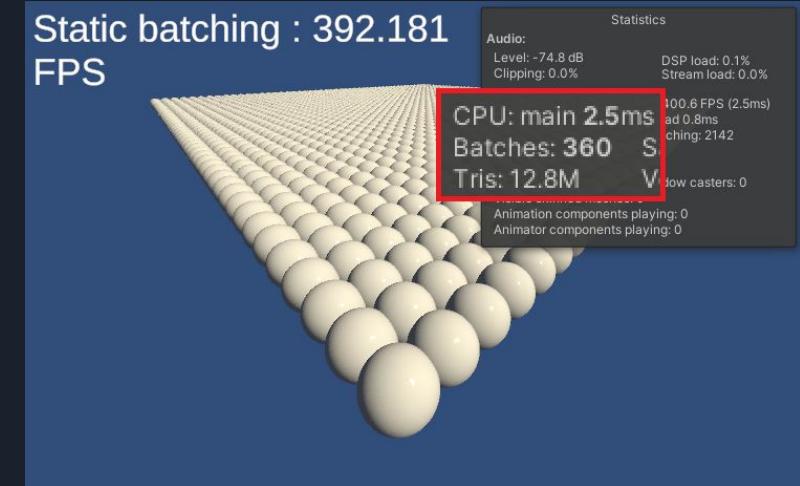
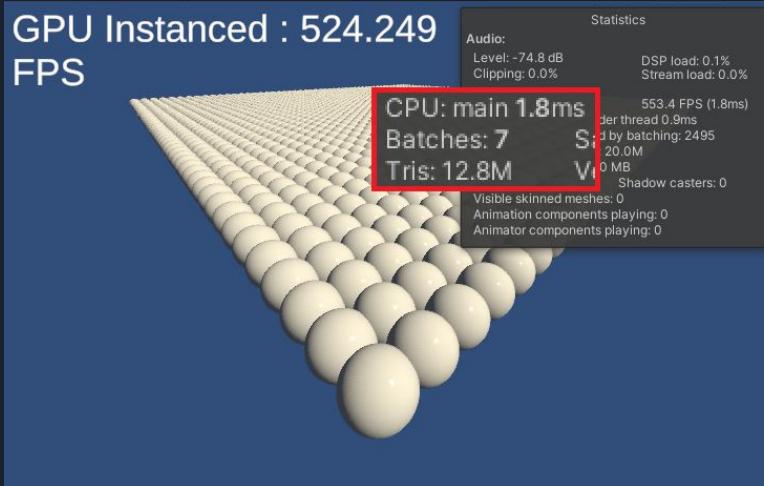
Static batching vs GPU Instancing

- Low polygon count.



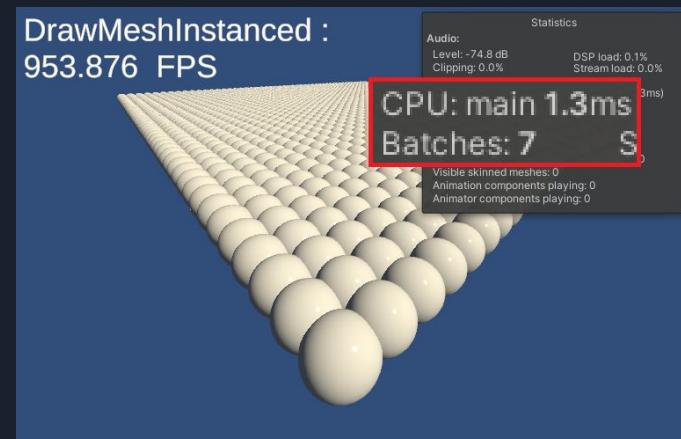
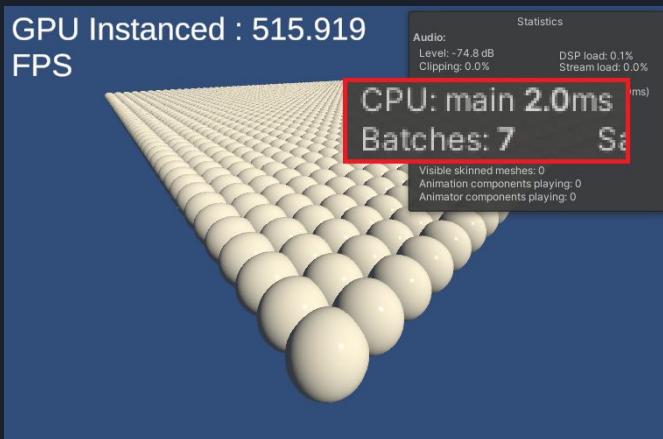
Static batching vs GPU Instancing

- 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 :

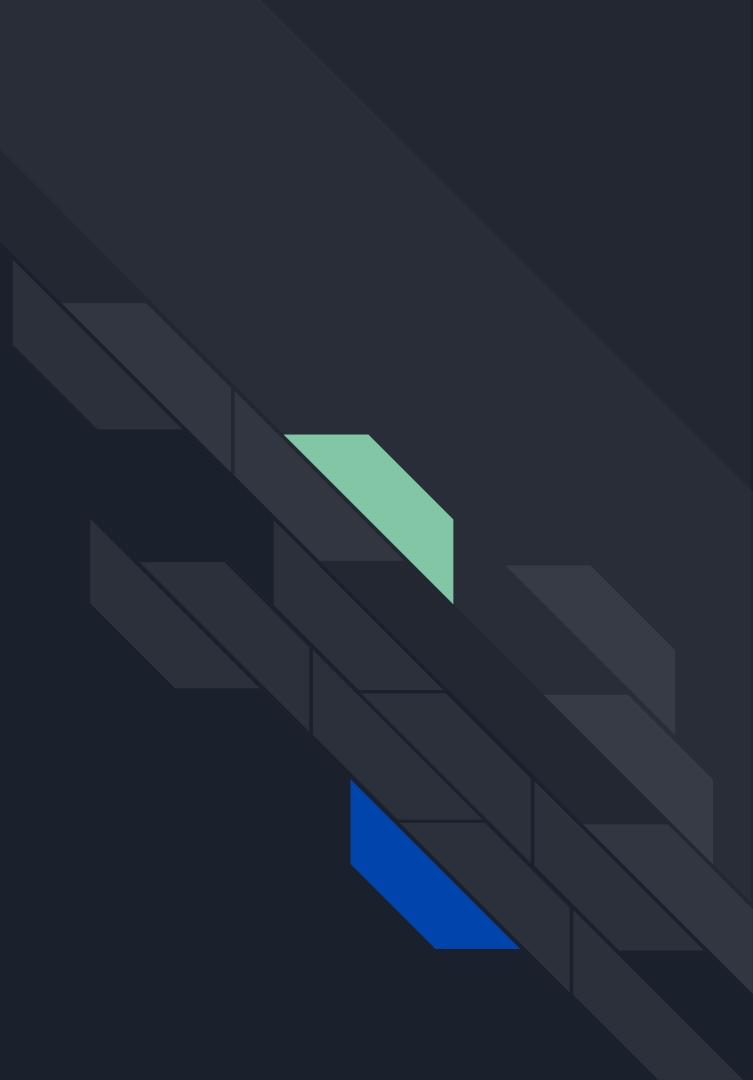
- GPU Instancing :
 - Great to render hundreds of similar objects, even better for high poly meshes
 - Great for meshes that use vertex shaders
 - Great for objects that can move
 - ~1000 Instances/batch
 - Less overdraw (see “Overdraw” chapter)
- Static batching :
 - Great to batch static objects that 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 worse than the 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

- We still need to avoid rendering what we can't see

Culling: Skip what doesn't need rendering

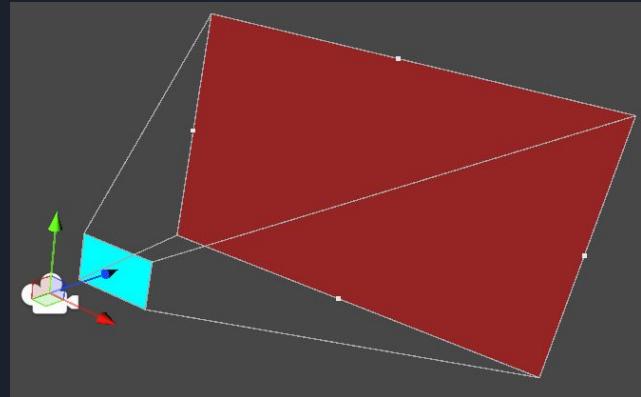
- Culling reduces draw calls by skipping objects not visible on screen
- Three main culling methods:
 - Frustum Culling
 - Occlusion Culling
 - Manual Culling

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 areas, like a VIP area
- Per-layer culling distances possible via "Camera.layerCullDistances"



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

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

Occlusion culling

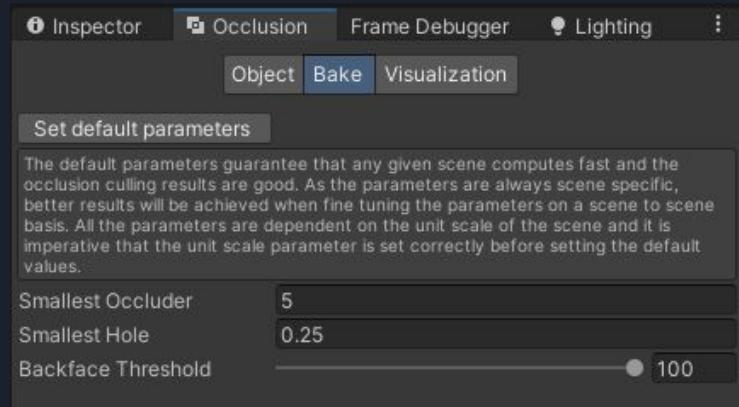
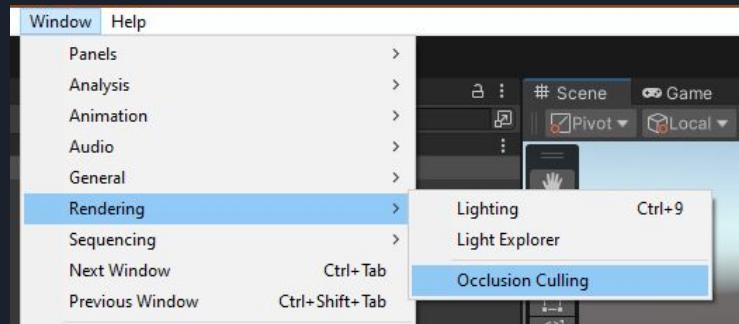
- Skips rendering objects hidden behind other objects
 - Reduces GPU load and draw calls, especially in dense scenes
- Unlike Frustum Culling, it checks visibility
- Best for static environments with lots of walls

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

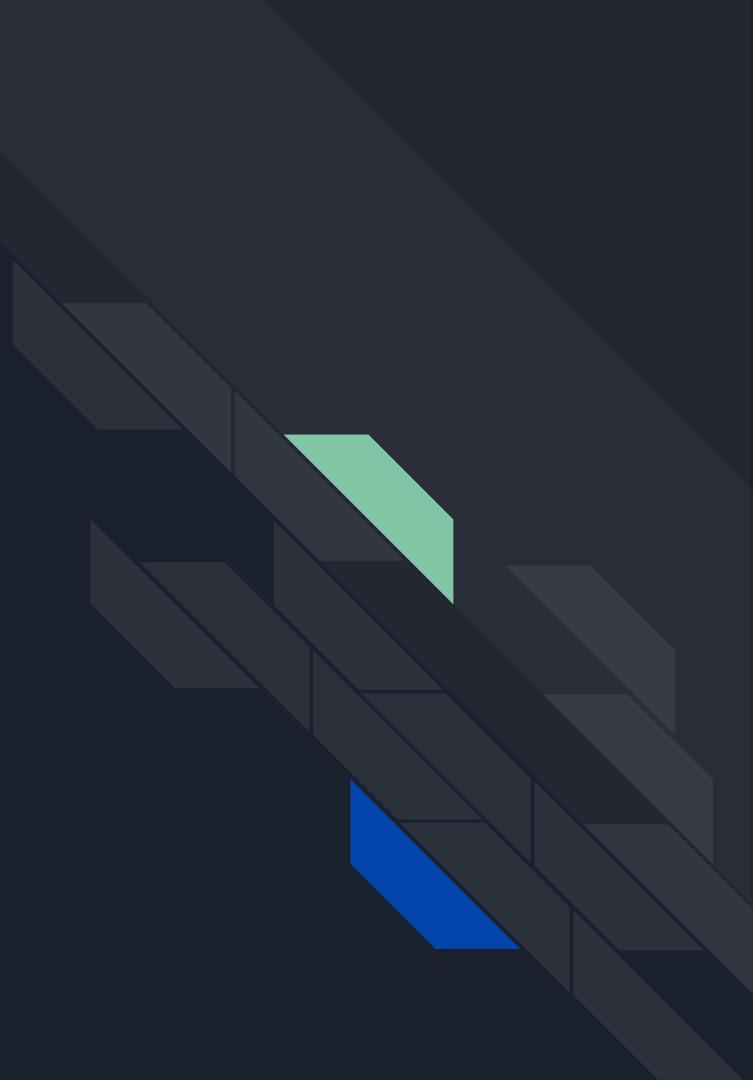




Manual culling

- 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
- Workarounds :
 - Distance-based object toggle
 - Less performant
 - Implementing custom player colliders
 - Requires attaching a collider to the player → less performant
 - LOD (explained later in the presentation)
 - Easier, more performant

Overdraw





Overdraw

- Happens when the same pixel is processed 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)

Overdraw simulation

Let's render this scene in two different ways

- Wall first
- Avatar first

Note : Simulation applies to fully opaque PBR shaders, it may work differently with other shaders



Overdraw simulation - Wall rendered first



First, the wall gets
textured



Then the avatar, some
parts of the wall get
retextured



In total, some pixels got
retextured twice!

We have some overdraw

Overdraw simulation - Avatar rendered first



First, the avatar gets textured



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
- Reminder : This applies to fully opaque geometries, like Unity Standard shader

Closer pixels should be rendered first





Render order matters!

Issue :

- The GPU doesn't sort pixels on its own
- → Unity needs to handle the sorting (CPU), before sending those draw calls to the GPU.

Unity's solution:

- Batches are first sorted based on the materials render queue
- If multiple material have the same render queue, then batches are sorted based on the distance between the bounding box center and the camera.

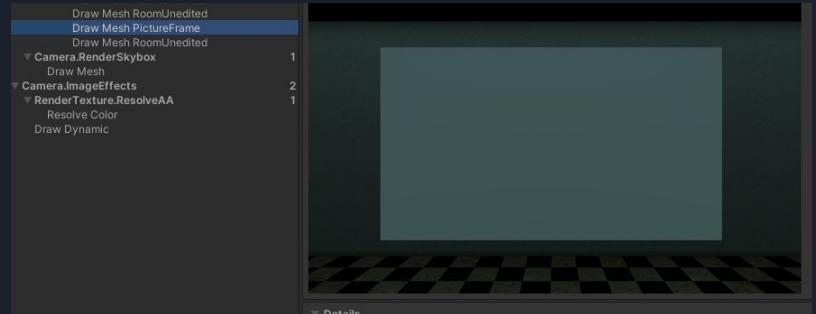
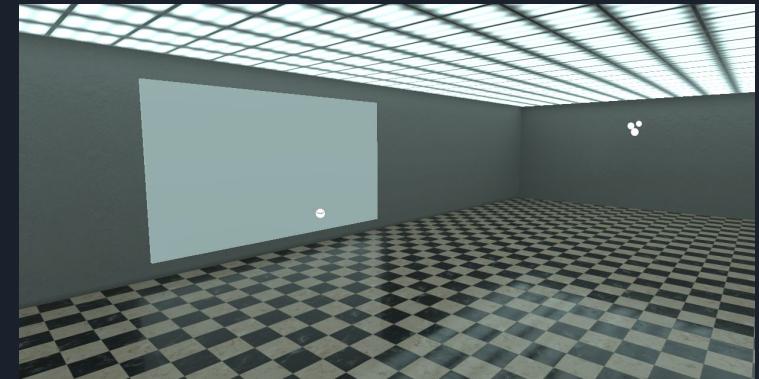
Rendering order :

- Render queue < 2499 : Opaque geometry, batches are sorted front-to-back
- Render queue 2500 : Skybox
- Render queue > 2500 : Transparent geometry, batches are sorted back-to-front
 - So transparent materials overdraw

Overdraw example

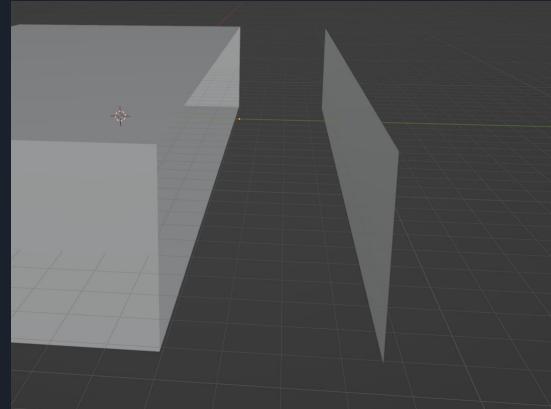
- A room with a picture frame
- Issue : The picture frame renders over the wall (Overdraw),
 - The camera is close to the bounding box center of the room
 - Front-to-back sorting : Room is rendered first, then the picture frame (first and second picture)

The Unity frame debugger is a great tool to check the render order of each batch



Fixing overdraw : Solution 1

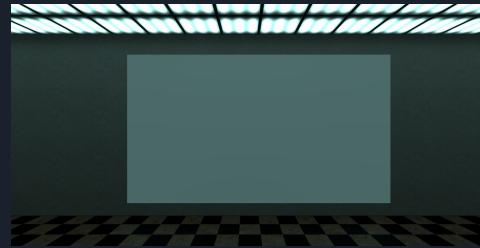
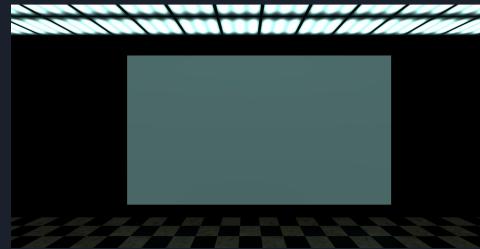
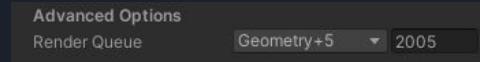
- Separating the wall from the rest of the room
 - 3 separate objects
- Turning off batching on the wall
 - Ensures the wall doesn't get merged back with the room
- Now we have 3 separate bounding boxes:
 - Main room
 - Back wall
 - Picture frame



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



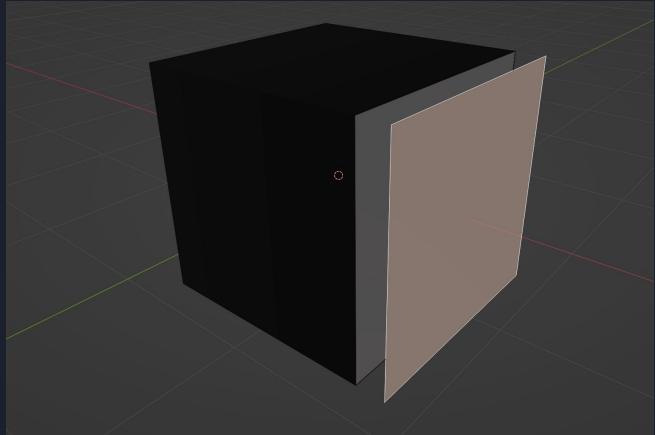
Hidden geometry

To test the performance of hidden geometry:

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

I made two versions :

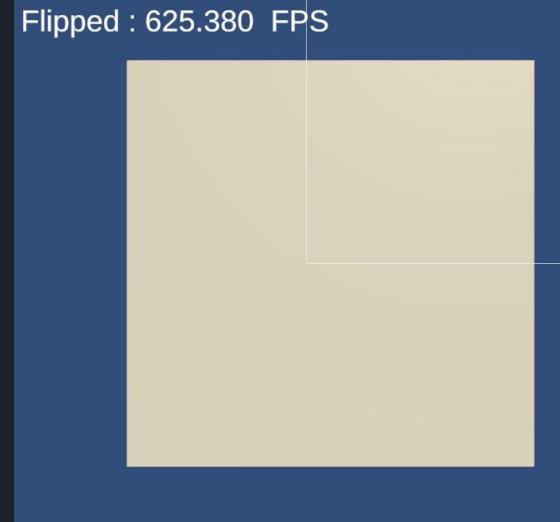
- One with flipped normals
- One without



Hidden geometry



Original mesh



Mesh with flipped normals

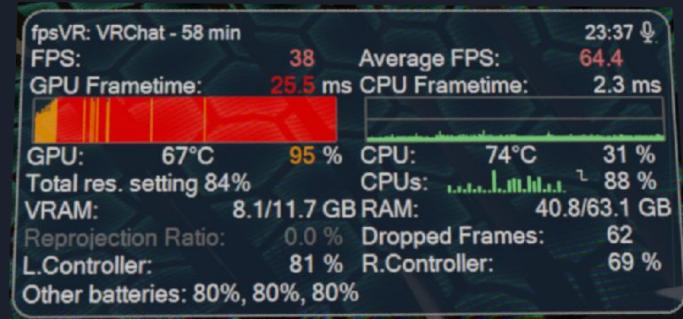
Hidden geometry - Explanation

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:

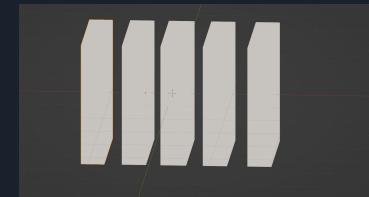
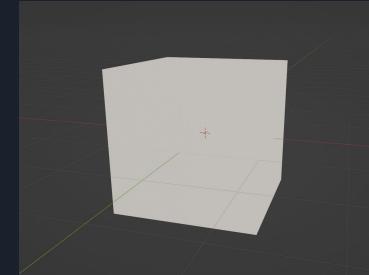
- Modify the mesh in Blender to delete hidden faces
- Avoid using cutout shaders to hide parts of a mesh, as it won't fix the problem
 - Instead, you can hide geometry before the pixel gets shaded, so in the vertex shader (example : Poiyomi's UV-Tile Discard system)
- Another solution explained the next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



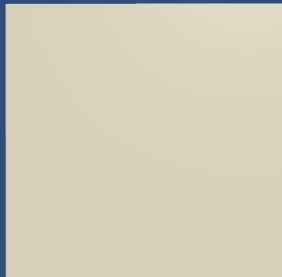
Hidden geometry

If we separate the “overdraw cube” into multiple meshes:

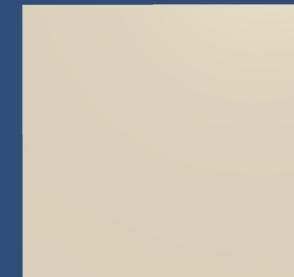
- Unity will sort each sub-mesh front-to-back (If static batching is disabled)
- Polygons would still be rendered back-to-front, but it won't be as bad
- Only the first mesh would cause some overdraw issues (if we look at the mesh from the front)
- Example on the right : Mesh separated into 5 meshes



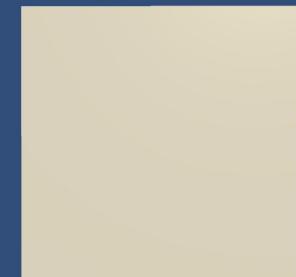
Separated mesh : 169.632 FPS



Separated mesh : GPU Inst : 172.124 FPS



Separated mesh and Static batched : 100.749 FPS



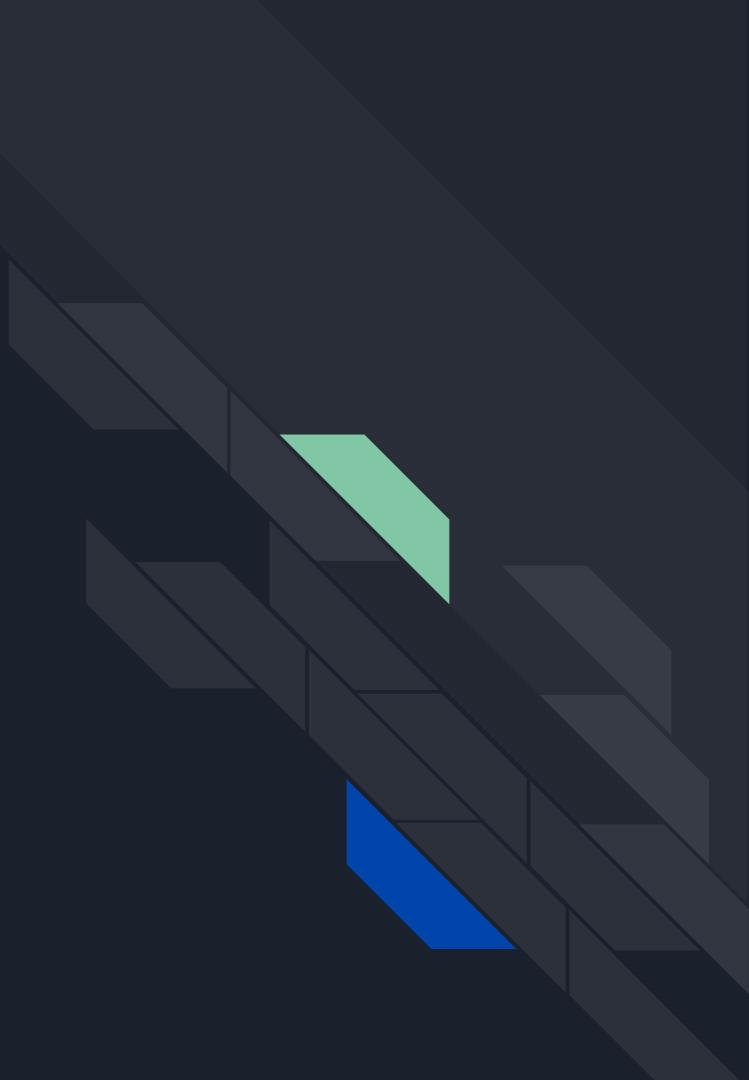
Hidden geometry and static batching

Possible use-case : Player inside a train/airplane looking at seats

- Huge amount of overdraw if all seats were a single mesh
- Static batching vs GPU instancing :
 - Static batching merges meshes, no front-to-back sorting
 - GPU Instanced batches are sorted front-to-back : Less overdraw
- I would :
 - Make each seat a separate mesh
 - Each seat needs to use the same seat mesh
 - Use GPU instancing



LOD (Level of details)

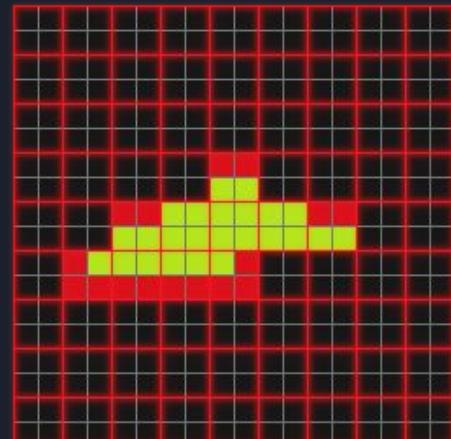
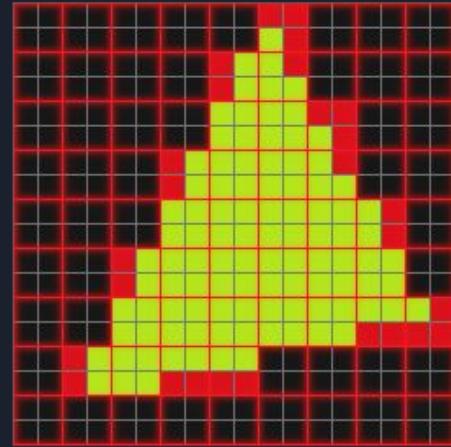


LOD (Level of details)

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

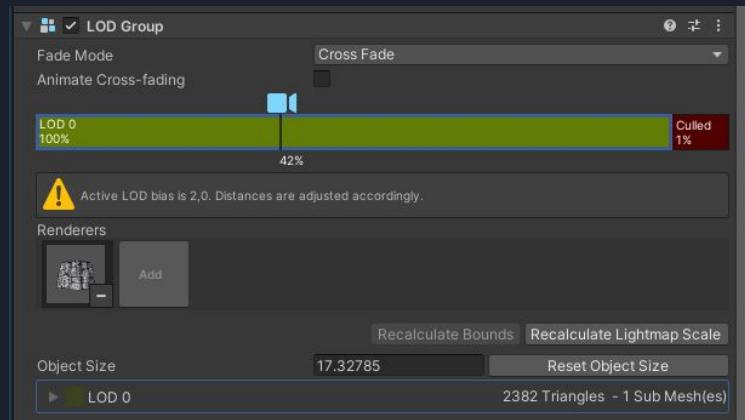
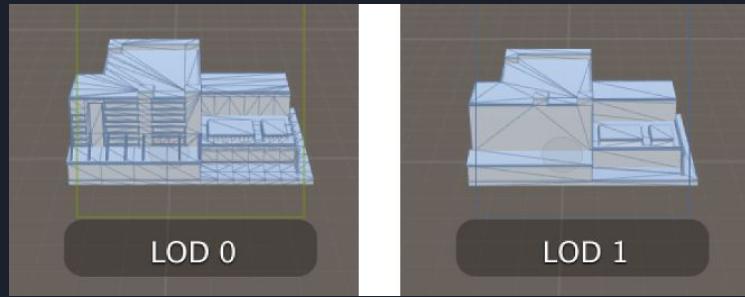
Polygon size matters!



LOD (Level of details)

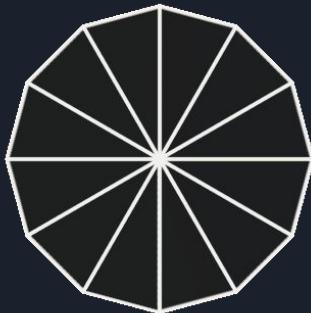
Solution : LODs

- 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
- There's an LOD graphic setting : Test your LODS with different settings
- Ideas:
 - Disabling shader features on further away objects
 - Culling smaller objects sooner
 - Transparent to opaque
 - Reduce poly count on tree leaves (cutout)
 - Billboards

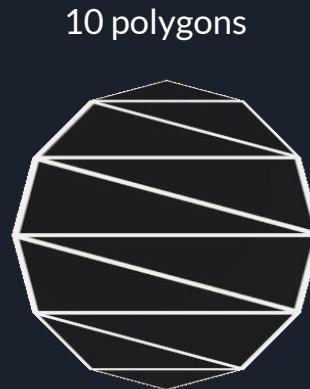


Efficient LOD topology

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



681 FPS

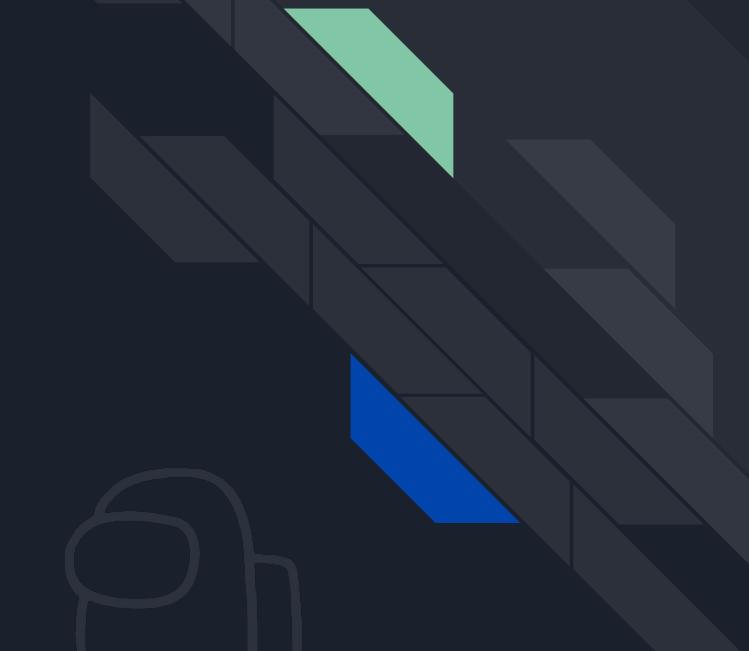


696 FPS



714 FPS

Impostors



Impostors

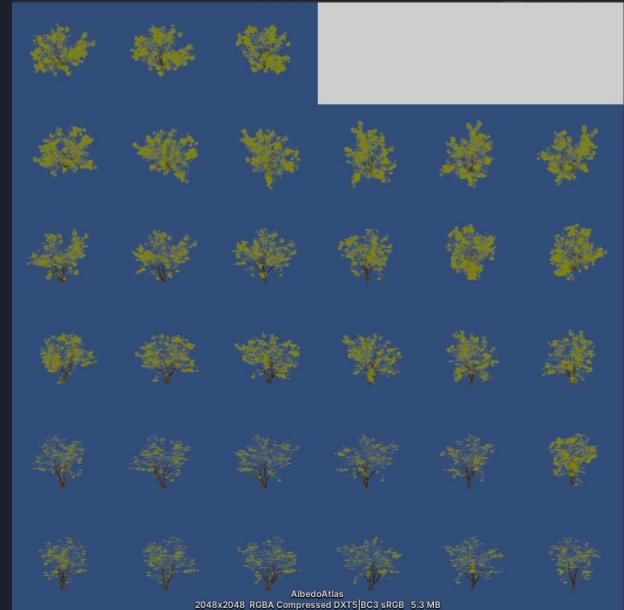
- Basically “fancier billboards”
- Great to render far-away objects

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
- Texture size (can be higher than regular billboards)



Impostor result





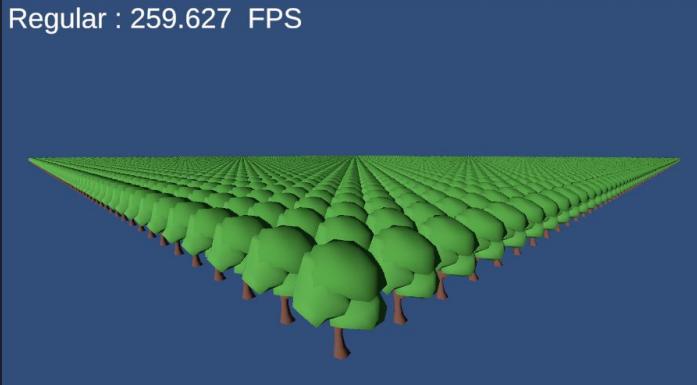
Impostor result from far away



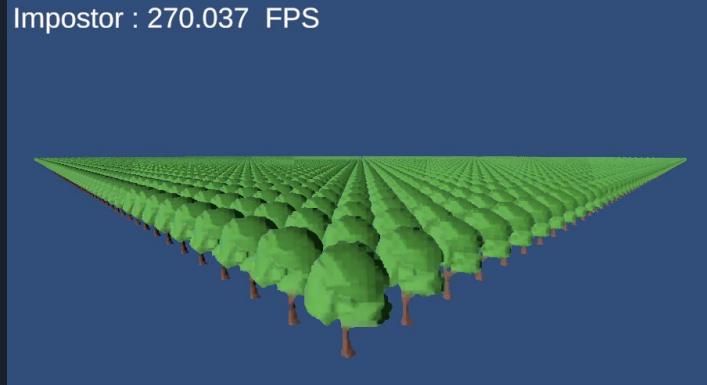
Impostor benchmarks

Original mesh has 455 Triangles, benchmark done with 10000 instances

Regular : 259.627 FPS

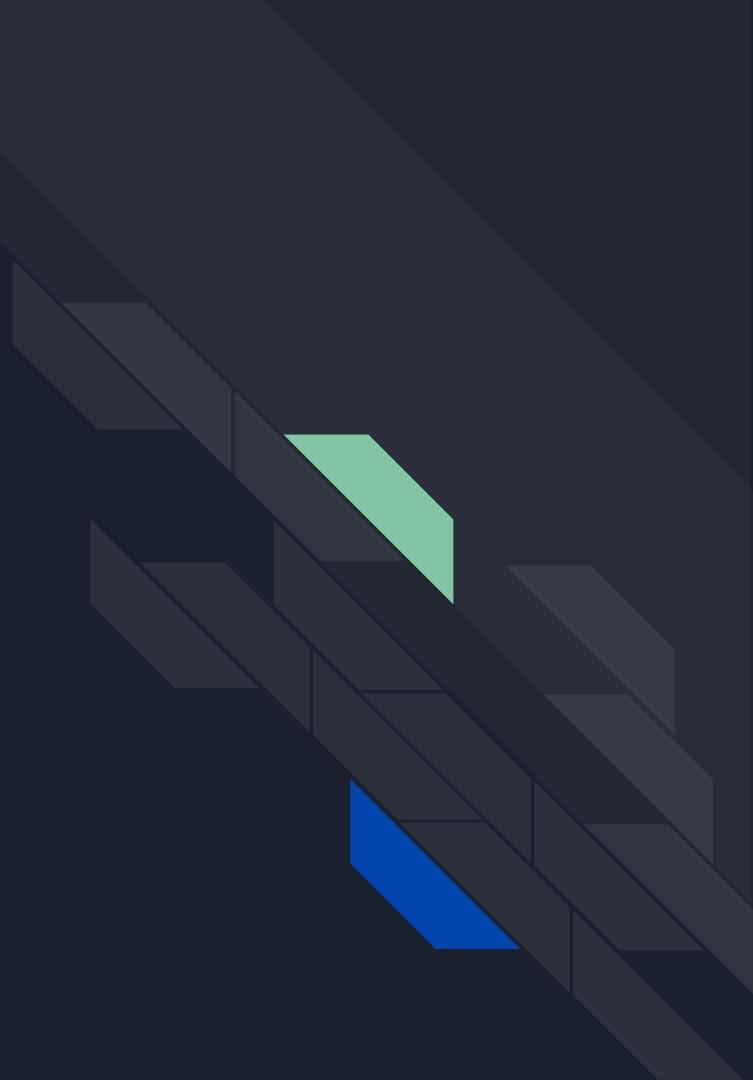


Impostor : 270.037 FPS



- Impostors are great if you want to work with high-poly meshes
- Be careful : Cutout shaders can be expensive on mobile devices

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.
 - Disable scripts that don't always need to run
 - Billboard effects should be done inside a shader, not Udon if possible.
- 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;

❸ 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);
}
```

How I optimized my NPCs in my VRtual Park world

Issue :

- Update loops are slow : I couldn't move my NPCs in that loop
- NavMesh baking not available at runtime

Solution: Rigidbody-based NPC movement

- Each NPC is a Rigidbody (no drag, no gravity)
 - Once pushed, the NPC keeps moving without stopping
- Movement logic:
 - Push the NPC toward the next road intersection
 - When reaching an intersection, push toward the next
 - Udon code is only executed once per NPC at each intersection

Pseudo-code example on the right, no update-loop required!

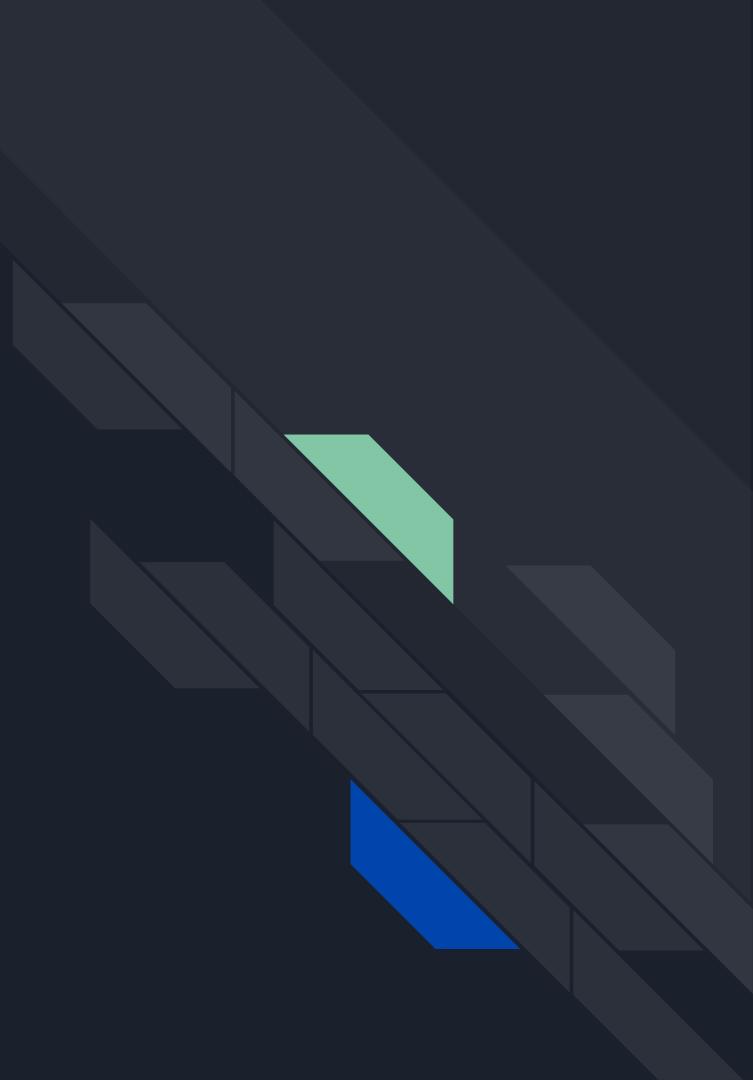
```
const float NPC_VELOCITY = 3.0f;
1 reference
public void NPCMovementSystem()
{
    // Finding the next intersection
    Vector3 destinationToReach = FindNextDestination();
    Vector3 currentNPCPosition = npcRigidbody.transform.position;

    // Direction the NPC needs to be pushed
    Vector3 direction = destinationToReach - currentNPCPosition;
    float distanceToDestination = direction.magnitude;

    // And we push the NPC
    npcRigidbody.velocity = direction.normalized * NPC_VELOCITY;

    // Destination will be reached in "time" seconds
    // then we push the NPC again
    float time = distanceToDestination / NPC_VELOCITY;
    SendCustomEventDelayedSeconds(
        nameof(NPCMovementSystem), time
    );
}
```

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

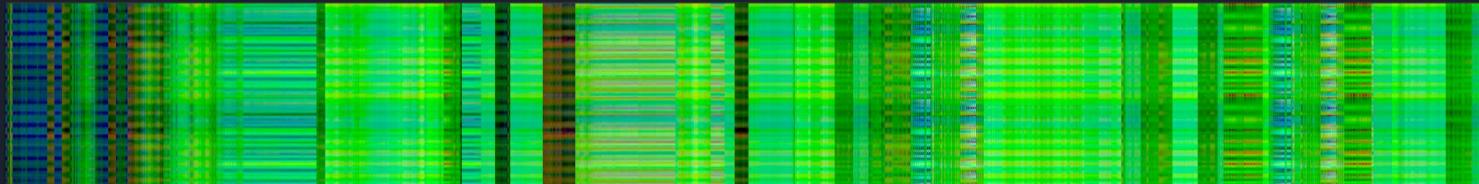
Drawbacks :

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



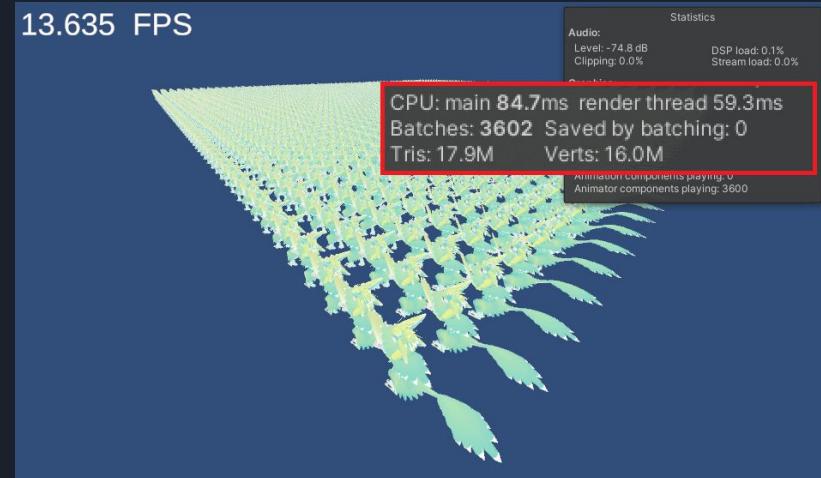
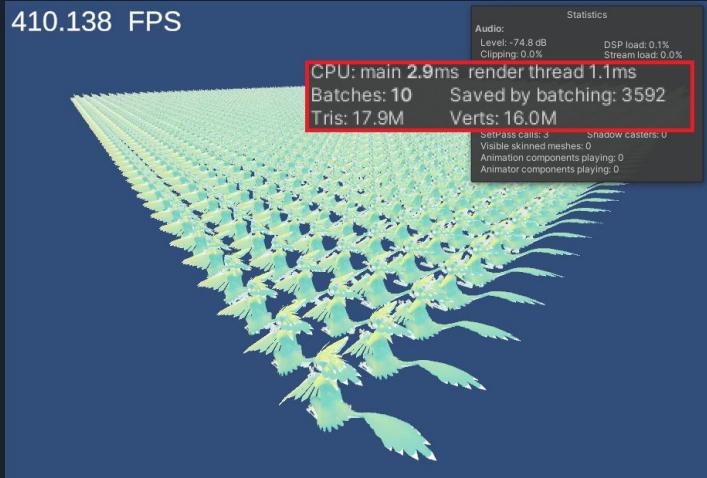
VAT Baker

- <https://github.com/fuqunaga/VatBaker>
- Converts animation files into a texture (example below)

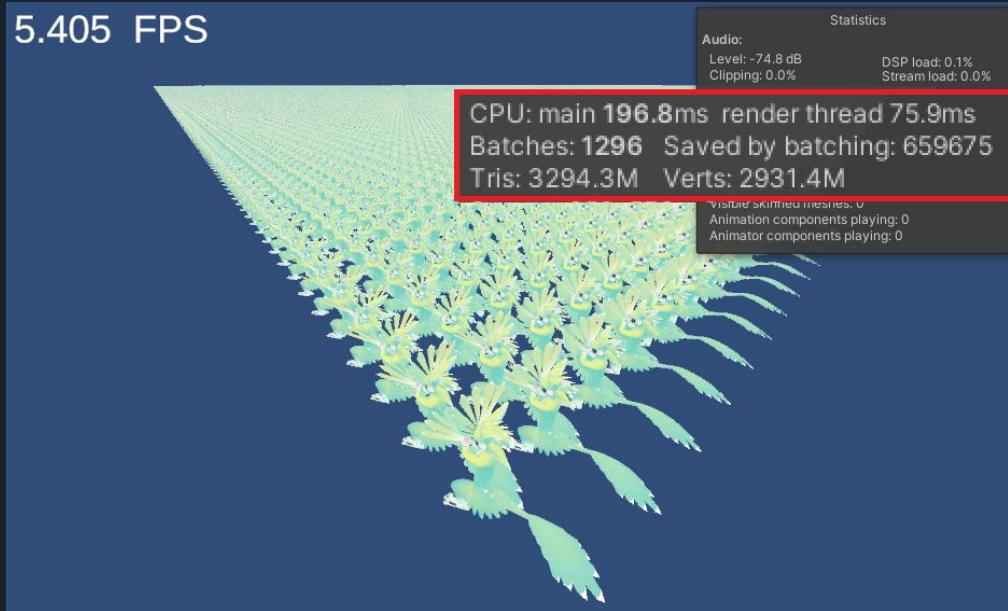


VAT vs. Skinned mesh renderers

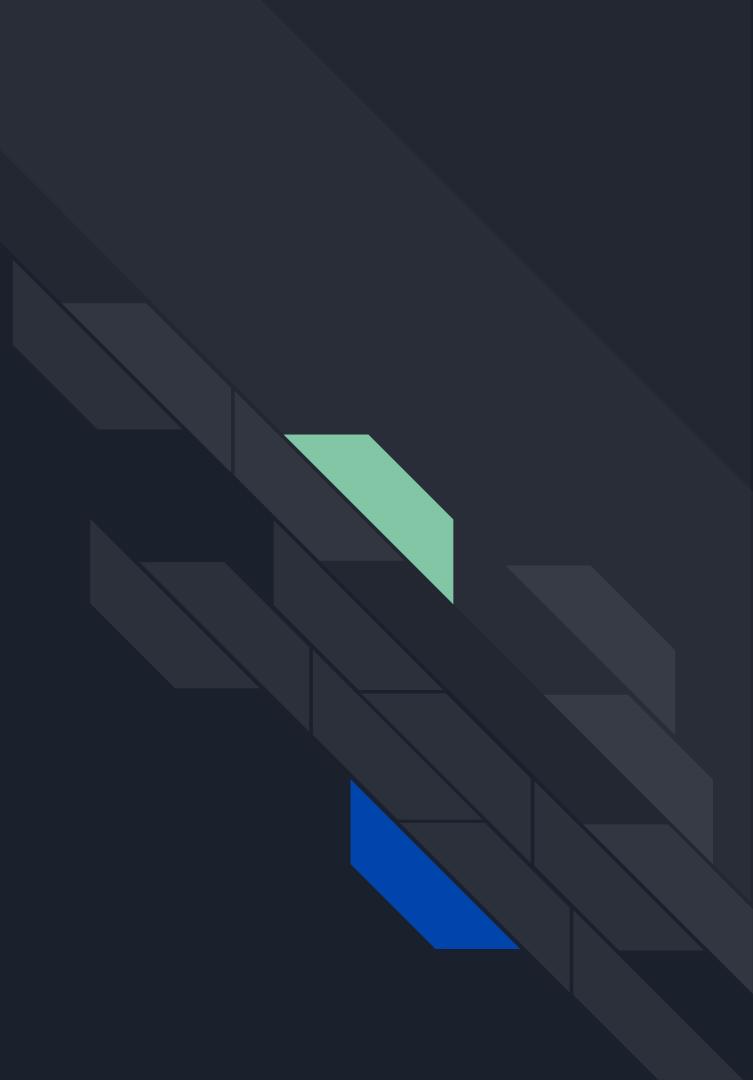
3600 Spir'i'valis



And just for fun, 660 000 Spiri'valis



Neat optimisation techniques

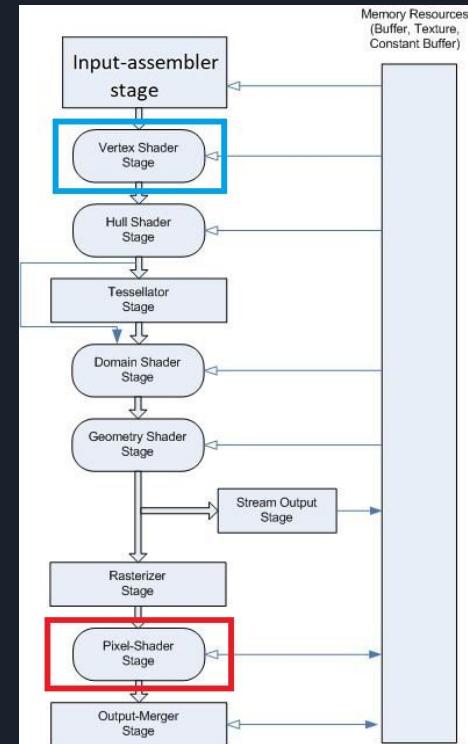


UV Tile Discard/UDIM Discard

Right Image: DirectX graphic pipeline.

Question : How can we efficiently hide polygons without deleting them in Blender?

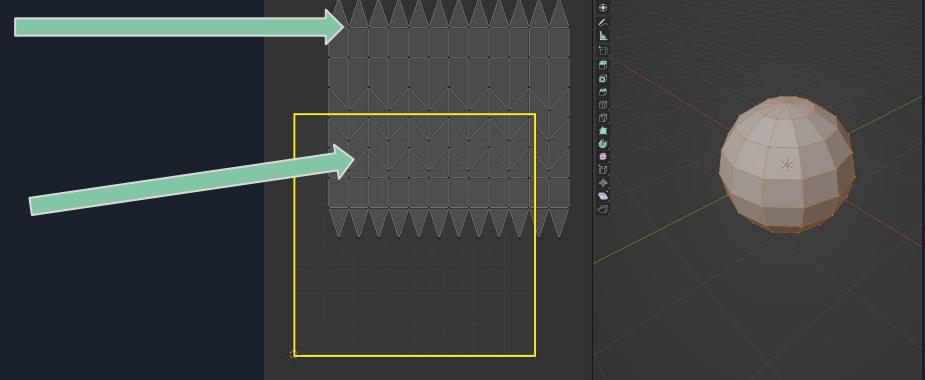
- Option A: Cutout shader
 - Discards pixels in the pixel shader
 - Not efficient, happens late in the pipeline (marked in red)
- Option B: “Disable” vertices in the vertex shader
 - Skip or push vertices off-screen in the vertex shader stage
 - Very performant, happen very early in the pipeline (marked in blue)
 - Drawback : Destroys whole polygons, not just a part of it.



UV Tile Discard/UDIM Discard

The Poiyomi shader supports that feature, but to understand how it works, let's create a very basic shader that destroys all polygons and pixels placed outside of the UV map, and to keep it even more simple, only if the Y-coordinate on the UV map > 1

Polygons/pixels I want to hide

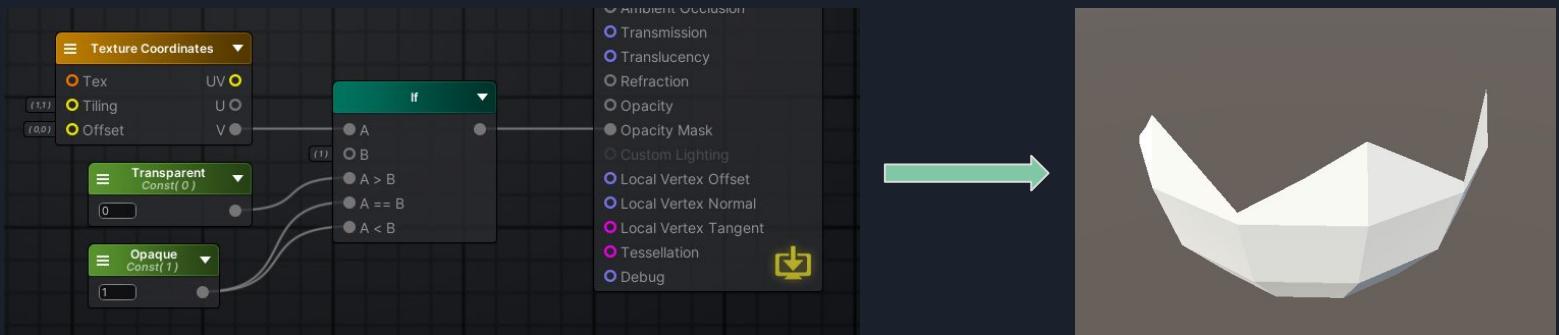


Polygons/pixels I want to keep

UV Tile Discard/UDIM Discard

Using a cutout shader :

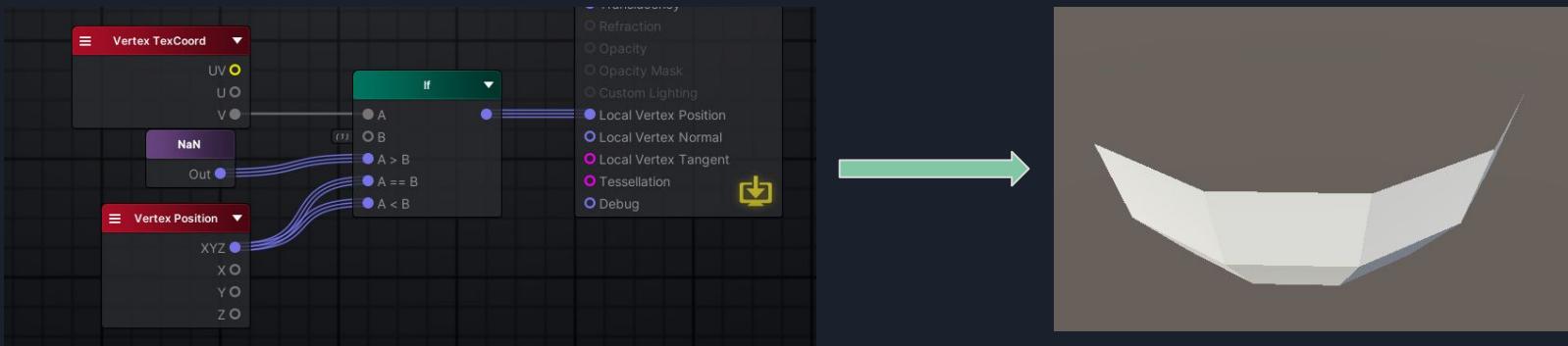
- “If the UV coordinates of the pixel are outside of the UV map, make the pixel transparent, otherwise make it opaque”
- Setup below made with the Amplify shader editor



UV Tile Discard/UDIM Discard

Using a vertex shader :

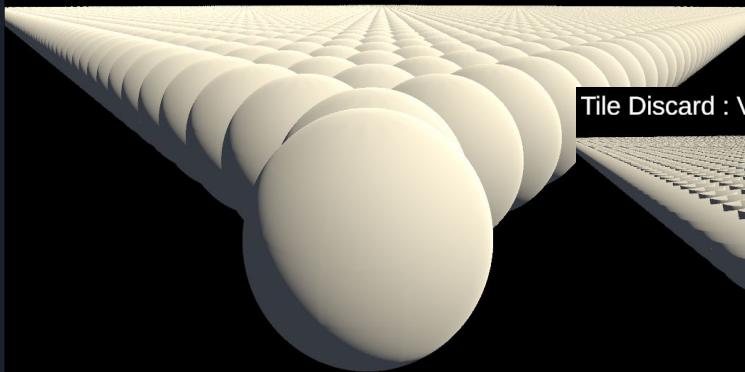
- “If the UV coordinates of the vertex are outside of the UV map, set its position to “NaN” (not a number), otherwise do not change its position”
- Setting the vertex position to “NaN”, like “float3(0.0f/0.0f, 0.0f/0.0f, 0.0f/0.0f)” pretty much destroys it.
- Destroys whole polygons



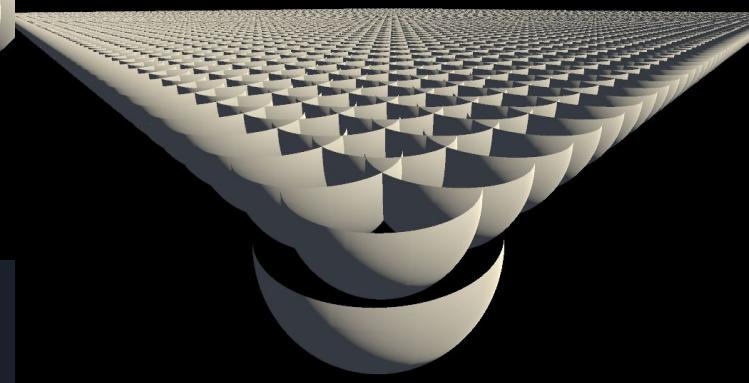
UV Tile Discard/UDIM Discar

Performance benchmark : Better performance by discarding tiles in the vertex shader

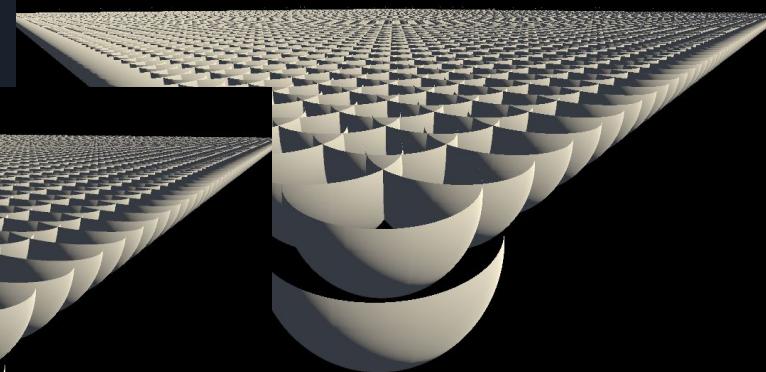
Regular standard shader : 16.340 FPS



Tile Discard : Vertex : 23.749 FPS



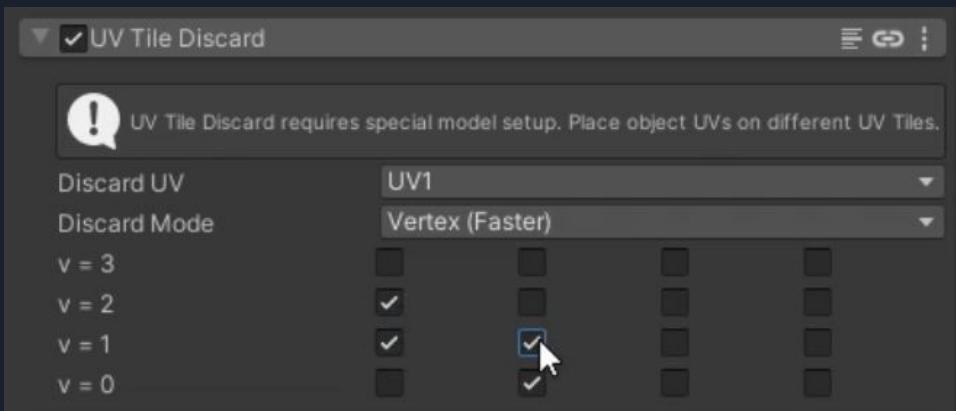
Tile Discard : Cutout : 15.574 FPS



UV Tile Discard/UDIM Discard

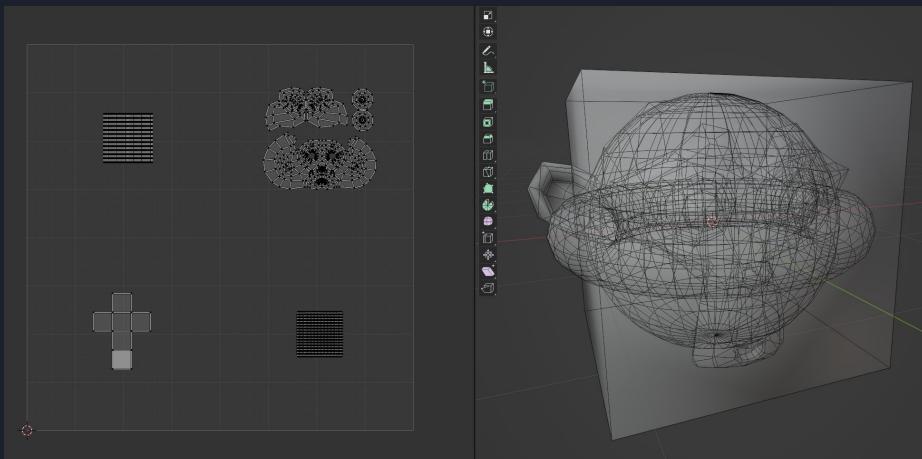
Now, let's go back to the Poiyomi shader:

- Similar system : Can discard vertices
- Discard vertices based on a grid system
 - UV map divided in 4x4 grid
 - We can tell the shader “*Discard all vertices that are in row 3, column 2 of the UV map*”
 - A different UV map can be selected, allowing you to keep the original UV map intact



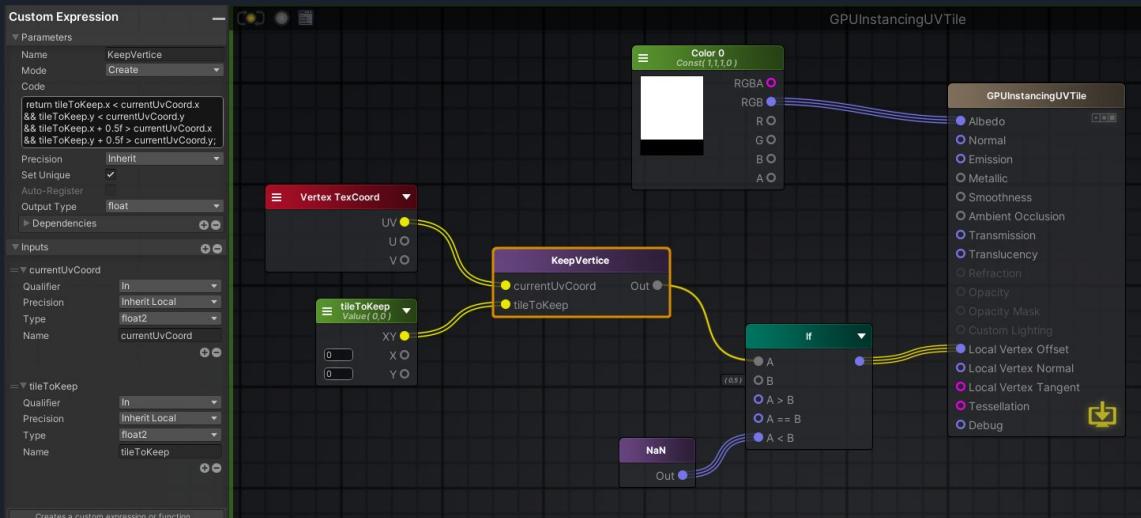
GPU instancing “different meshes” using UV time discard

Step 1 : Create a mesh that combines multiple meshes into one, and put each mesh on a separate UV island, this way we can discard parts of the meshe based on their position on the UV map



GPU instancing “different meshes” using UV time discard

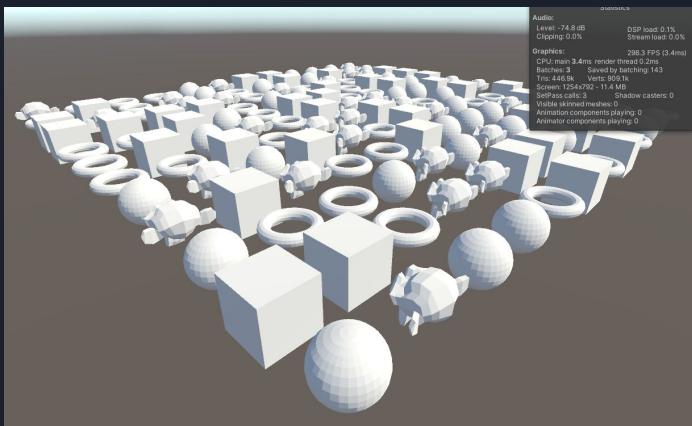
Step 2 : Make a shader that uses the parameter “tileToKeep” as an instanced property



GPU instancing “different meshes” using UV time discard

Step 3 : Use MaterialPropertyBlock to set the “tileToKeep” property, this way some parts of the original mesh can be destroyed and still be GPU instanced

It looks like every mesh is different, but it is still the same mesh! So they can be GPU Instanced



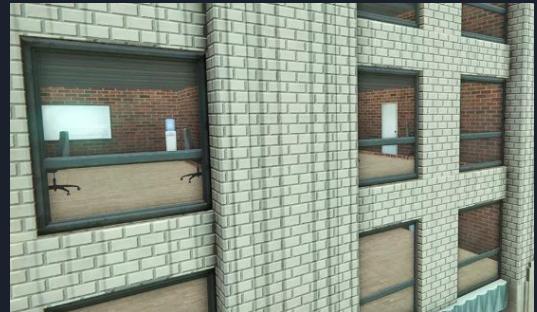
Render apartments

Issues:

- Rendering fake apartments as JPEGs look ugly (top image)
- Rendering an interior is too expensive.

Solution: Fake apartment shader

- Simulates depth and detail without actual geometry
- Used for instance in the game “Spider-man” (PS4/PS5)
- Example : “Fake Interiors” shader (Free, by “Amplify Creations”)



Render apartments : Another idea!

Switching apartment detail based on player position!

When Outside the Building:

- Use a Reflection Probe to fake the apartment interior on windows (see image)
- Hide the actual apartment geometry
- Use an opaque window shader to prevent seeing inside

When Inside the Building:

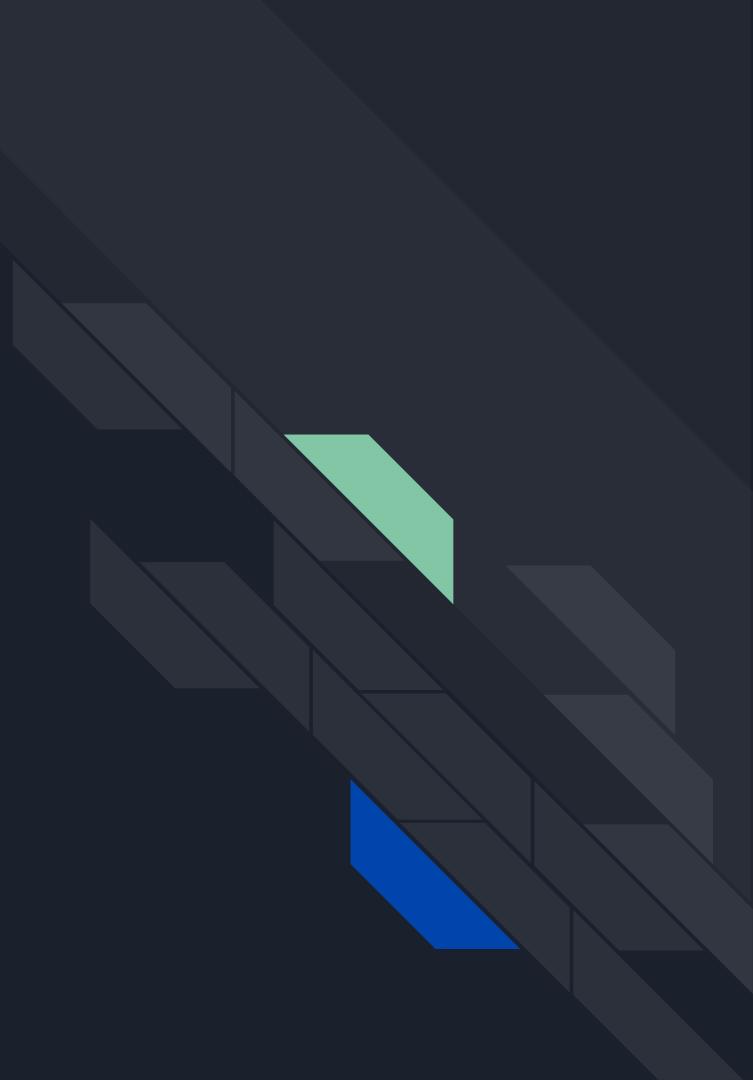
- Toggle on the real apartment geometry

Example of a VRChat world using this system: "Thad Home of Cuddles" by "Thad Gyther"

- Window shader can be downloaded on his GitHub
<https://github.com/thad-gyther/Thad-Unity-Vrc-Sandbox/blob/main/Thad/Shader/EnvironmentProbe.shader>



Summary





Summary

- Drawcalls affect the performance on the CPU
- To reduce draw calls :
 - Static batching, GPU Instancing
 - Bake your lighting to reduce number of RT lights and shadows
 - Reduce material count
- Overdraw affects the performance on the GPU
 - Make sure draw calls are rendered front-to-back for opaque shaders! You can use the Unity frame debugger to check.
 - 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



Some 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!