Building Big Games on Little Headsets

Ben Hopkins

Principal Graphics Engineer & Mixologist of Realities

Owlchemy Labs

About Me

- Ben Hopkins,
  Principal Graphics Engineer
- 7 years @ Owlchemy Labs
- Worked on all our VR titles
- Including all internal ports; PSVR, Quest
- 25 years in games & interactive coding
- First language: Z80 ASM
- Current languages: C#, HLSL
  - C++, Java, Python (plugins, tooling, R&D)

Owlchemy Labs

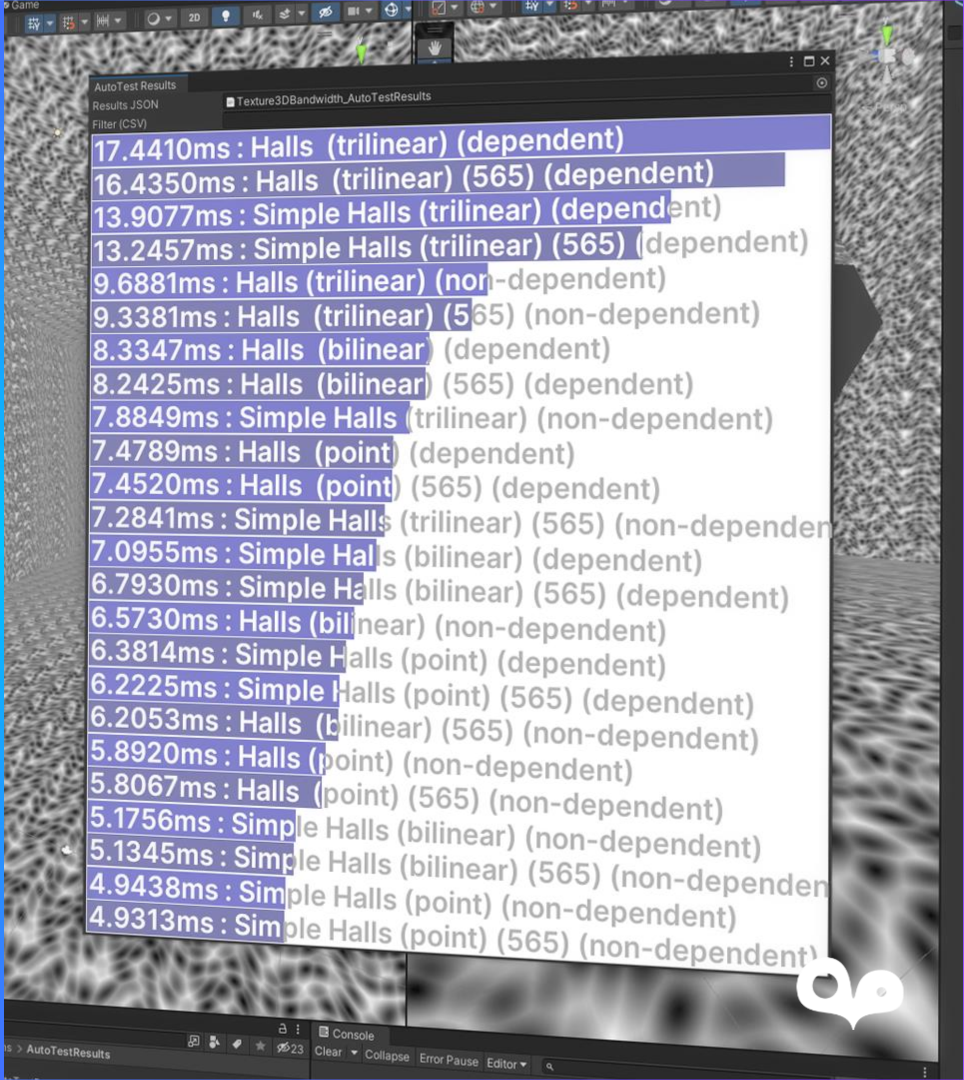
- Founded 2010,
  VR focused since 2016
- Acquired by Google in 2017
- We make absurd & highly polished games!
- Our motto is “VR for everyone”
- Our games:
  - Job Simulator
  - Rick & Morty: Virtual Rick-ality
  - Vacation Simulator
  - Cosmonious High

About this Talk

- An overview of some of our custom optimization tooling and some bite size tricks we used when optimizing Cosmonious High for Quest
- A deeper look at one of the more costly systems in the game, SplatTech. And how we rewrote it for Quest

Optimization Tooling & Bitesize Tricks!

A/B Test Framework



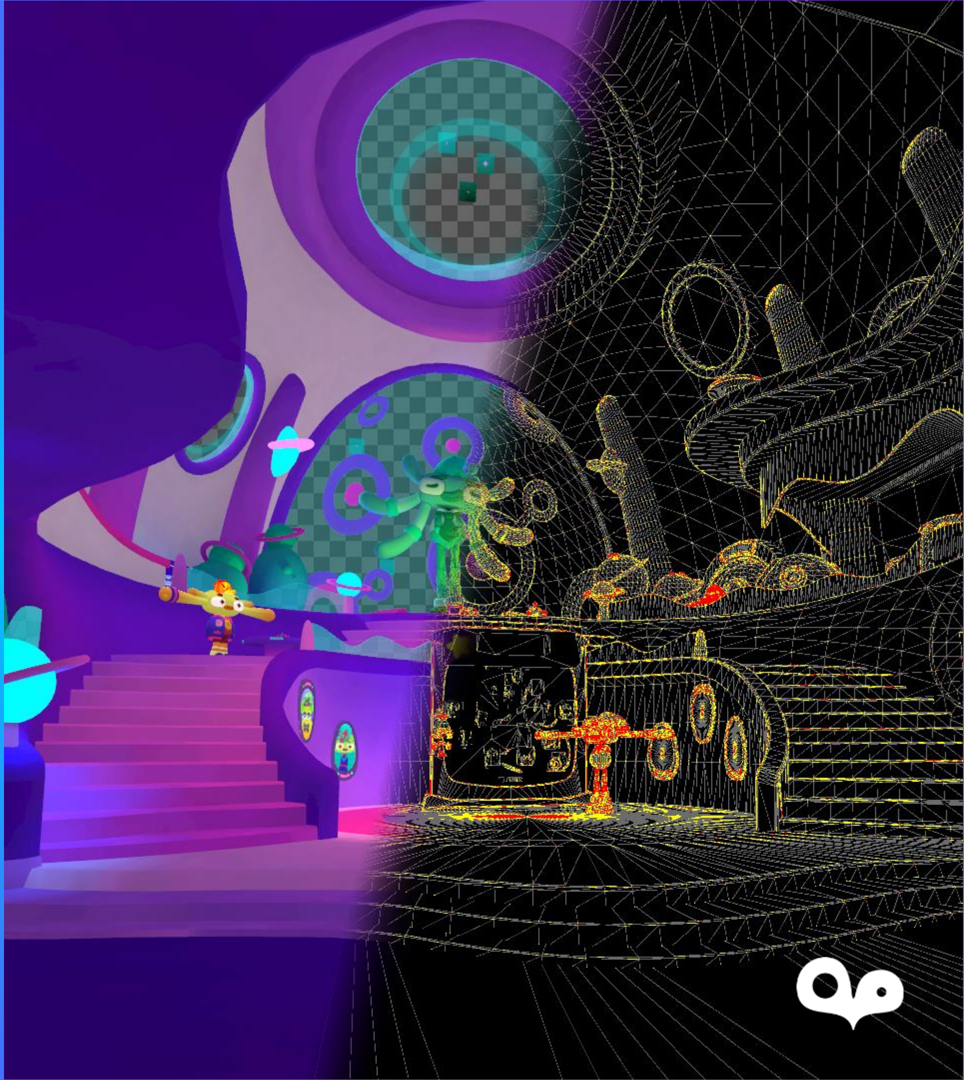
AutoTest Results
Results JSON Texture3DBandwidth_AutoTestResults
Filter (CSV)

```
17.4410ms : Halls (trilinear) (dependent)
16.4350ms : Halls (trilinear) (565) (dependent)
13.9077ms : Simple Halls (trilinear) (dependent)
13.2457ms : Simple Halls (trilinear) (565) (dependent)
9.6881ms : Halls (trilinear) (non-dependent)
9.3381ms : Halls (trilinear) (565) (non-dependent)
8.3347ms : Halls (bilinear) (dependent)
8.2425ms : Halls (bilinear) (565) (dependent)
7.8849ms : Simple Halls (trilinear) (non-dependent)
7.4789ms : Halls (point) (dependent)
7.4520ms : Halls (point) (565) (dependent)
7.2841ms : Simple Halls (trilinear) (565) (non-dependen
7.0955ms : Simple Halls (bilinear) (dependent)
6.7930ms : Simple Halls (bilinear) (565) (dependent)
6.5730ms : Halls (bilinear) (non-dependent)
6.3814ms : Simple Halls (point) (dependent)
6.2225ms : Simple Halls (point) (565) (dependent)
6.2053ms : Halls (bilinear) (565) (non-dependent)
5.8920ms : Halls (point) (non-dependent)
5.8067ms : Halls (point) (565) (non-dependent)
5.1756ms : Simple Halls (bilinear) (non-dependent)
5.1345ms : Simple Halls (bilinear) (565) (non-dependen
4.9438ms : Simple Halls (point) (non-dependent)
4.9313ms : Simple Halls (point) (565) (non-dependent)
```

A/B Test Framework

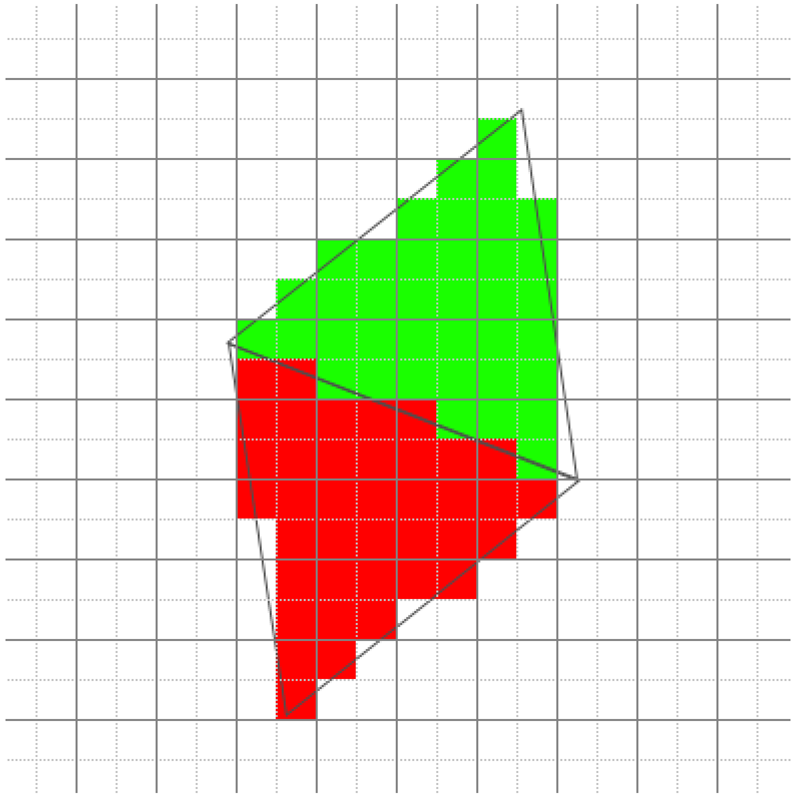
- Wanted to get actual GPU timings for various features/optimizations
- Uses Meta's metrics API to get per frame GPU time (AppT)
- Framework provides a fast way to setup deterministic tests
  - Head tracking and FFR fully disabled
  - Each test is easily duplicated, only altering the aspect we want timings for per test
- Test suite builds are fast and automated
  - Each test runs sequentially for several seconds
  - Per frame GPU time is recorded and averaged at test end
  - Test results are written to a JSON file and saved to device
  - Editor tooling automatically pulls results from device and presents as a filterable graph!

MSAA
Cost Visualization



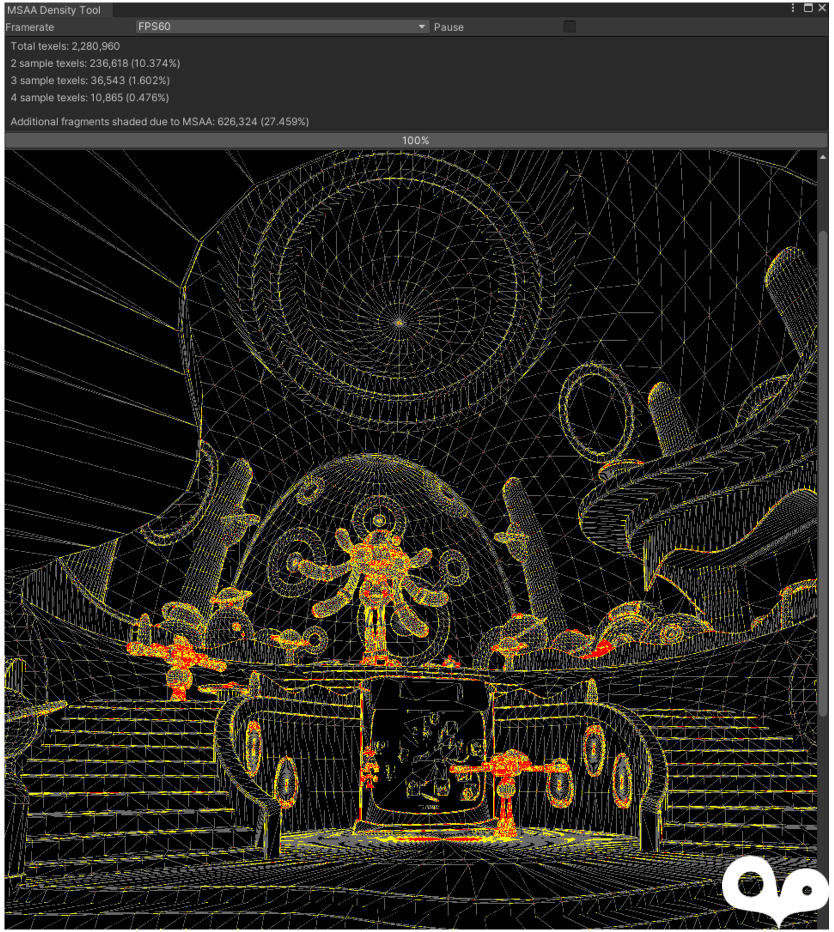
MSAA Cost Visualization

- How MSAA works
  - Can be thought of as optimized super sampling
  - Each screen pixel is subdivided into sub pixels,
  - E.g. MSAA 4x = 4 sub pixels per pixel
  - Depth test performed at this higher resolution
  - Pixel shader only called once per pixel per triangle
  - But since multiple triangles can have coverage within a pixel it is possible for many more pixel shader invocations along triangle edges
  - Is most costly with highly dense meshes, especially as they move farther from the camera
    - One of the less obvious benefits to LODs



MSAA Cost Visualization

- Wrote a tool to help teammates visualize/quantify this
  - Black: 1 pixel shader invocation
  - Gray: 2 pixel shader invocations
  - Yellow: 3 pixel shader invocations
  - Red: 4 pixel shader invocations
- Super useful for validating LODs and identifying hotspots
- Renders the scene view with a custom shader that records subpixel coverage via SV_Coverage
- Quantifies via a compute shader



```
Total texels: 2,280,960
2 sample texels: 236,618 (10.374%)
3 sample texels: 36,543 (1.602%)
4 sample texels: 10,865 (0.476%)

Additional fragments shaded due to MSAA: 626,324 (27.459%)
```

Optimized Texture Atlasing

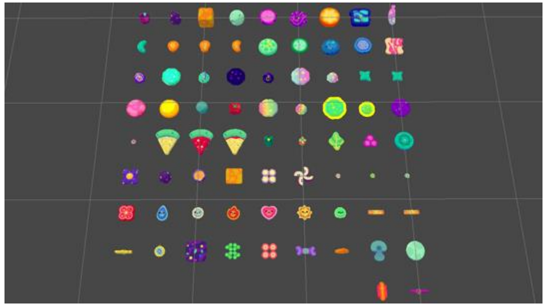
Optimized Texture Atlasing

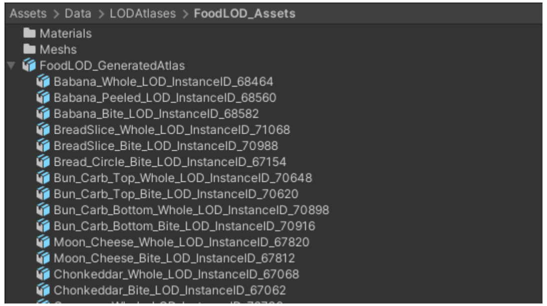
- We have a lot of pickupable objects in Cosmonious High (and a backpack)
  - Player can carry many objects from any scene in the game to any other scene in the game
  - Potentially every object in the game could end up in a single scene
- In the past we've relied on manual texture atlas generation, trying to group objects based on likelihood of them being in a single scene, by hand
- The more objects we can fit in a single atlas, the less chance there is of 2 objects being in different atlases and unbatchable

Optimized Texture Atlasing

- Solution, custom atlas generation pipeline
    - We select all prefabs we'd like in a single atlas
    - Generate a single fbx containing all meshes of all prefabs
    - Import fbx into Blender where a custom python script processes all individual meshes, packing all UV islands of all meshes into a single UV set (UV Packmaster plugin is magic)
    - These new UVs are assigned to the second channel of each mesh
    - Export from Blender and reimport in Unity
    - Run our bake tool which generates a single new texture using whatever we want from UV0 to UV1
    - Can be as simple as baking texture or custom shading
    - Update meshes in all prefabs to use final atlased meshes
    - Whole process is automated to a few button clicks and fully non destructive!

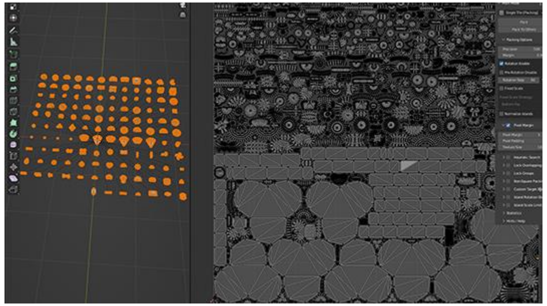
Optimized Texture Atlasing

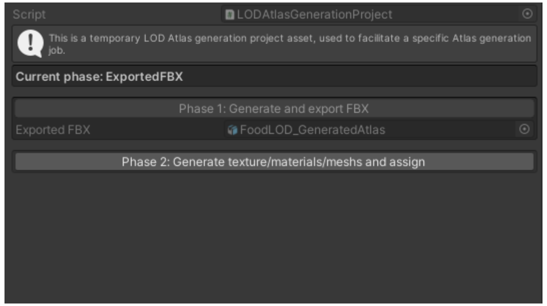

1. Select desired prefabs (74 here)


Assets > Data > LODAtlases > **FoodLOD_Assets**

- Materials
- Meshs
- FoodLOD_GeneratedAtlas
  - Babana_Whole_LOD_InstanceID_68464
  - Babana_Peeled_LOD_InstanceID_68560
  - Babana_Bite_LOD_InstanceID_68582
  - BreadSlice_Whole_LOD_InstanceID_71068
  - BreadSlice_Bite_LOD_InstanceID_70988
  - Bread_Circle_Bite_LOD_InstanceID_67154
  - Bun_Carb_Top_Whole_LOD_InstanceID_70648
  - Bun_Carb_Top_Bite_LOD_InstanceID_70620
  - Bun_Carb_Bottom_Whole_LOD_InstanceID_70898
  - Bun_Carb_Bottom_Bite_LOD_InstanceID_70916
  - Moon_Cheese_Whole_LOD_InstanceID_67820
  - Moon_Cheese_Bite_LOD_InstanceID_67812
  - Chonkeddar_Whole_LOD_InstanceID_67068
  - Chonkeddar_Bite_LOD_InstanceID_67062

2. Automatic FBX export


3. Blender script packs UV islands


Script LODAtlasGenerationProject

This is a temporary LOD Atlas generation project asset, used to facilitate a specific Atlas generation job.

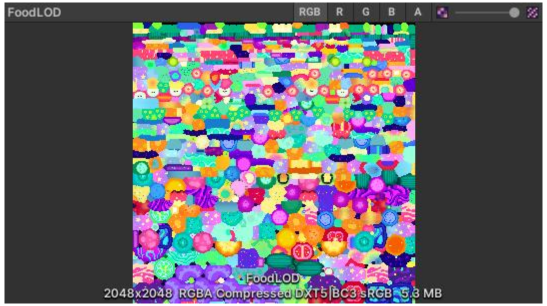
Current phase: ExportedFBX

Phase 1: Generate and export FBX

Exported FBX FoodLOD_GeneratedAtlas

Phase 2: Generate texture/materials/meshs and assign

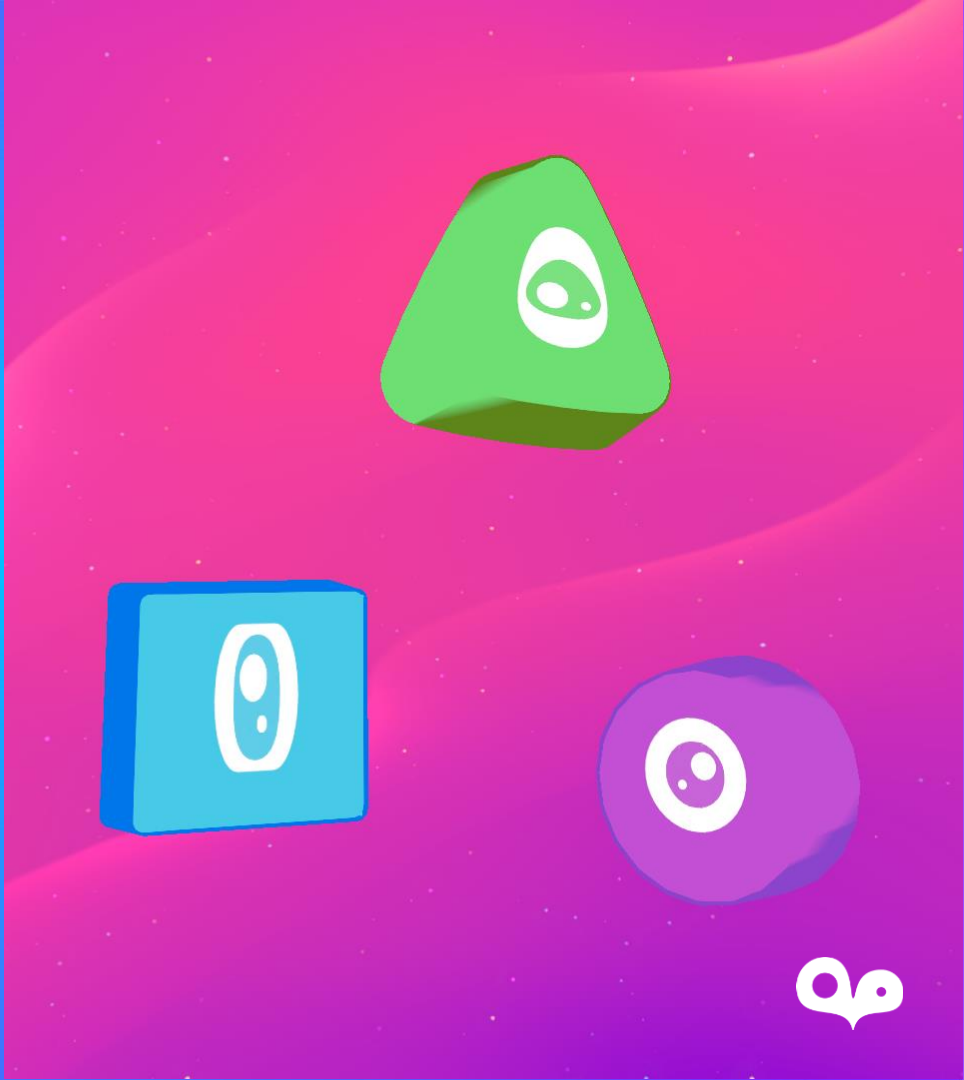
4. Generate texture, meshes & update all prefabs to use them



FoodLOD
2048x2048 RGBA Compressed DXT5|BC3 sRGB 5.3 MB

5. Success! 74 unique objects in a single atlas.

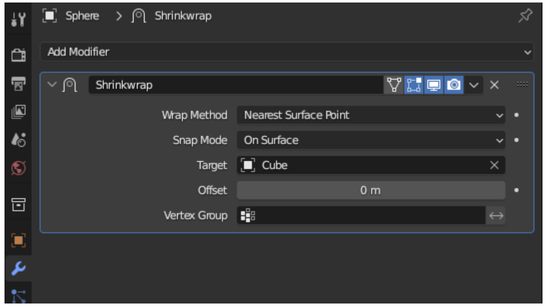
GPU Instancing Multi Meshes



GPU Instancing Multi Meshes

- Pliks are 3D shape creatures that use our special 2D shader
- Several different meshes, potentially a lot of them in a frame
- Pack several meshes into a single mesh, via texture coords
- Interpolate between shapes in vertex shader, similar to blend shapes
- All pliks now render using a single GPU instanced drawcall per frame!

GPU Instancing Multi Meshes



Sphere > Shrinkwrap

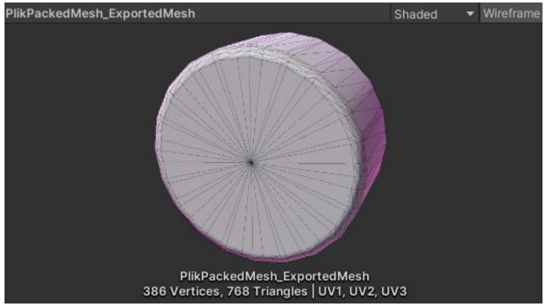
Add Modifier

Shrinkwrap

Wrap Method: Nearest Surface Point
Snap Mode: On Surface
Target: Cube
Offset: 0 m
Vertex Group:

Use Blender's shrinkwrap modifier to morph circle topology to the triangle and square shapes.

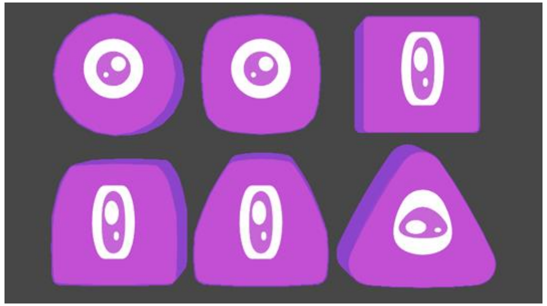
This guarantees same vertex count and order across shape meshes.



PlikPackedMesh_ExportedMesh

PlikPackedMesh_ExportedMesh
386 Vertices, 768 Triangles | UV1, UV2, UV3

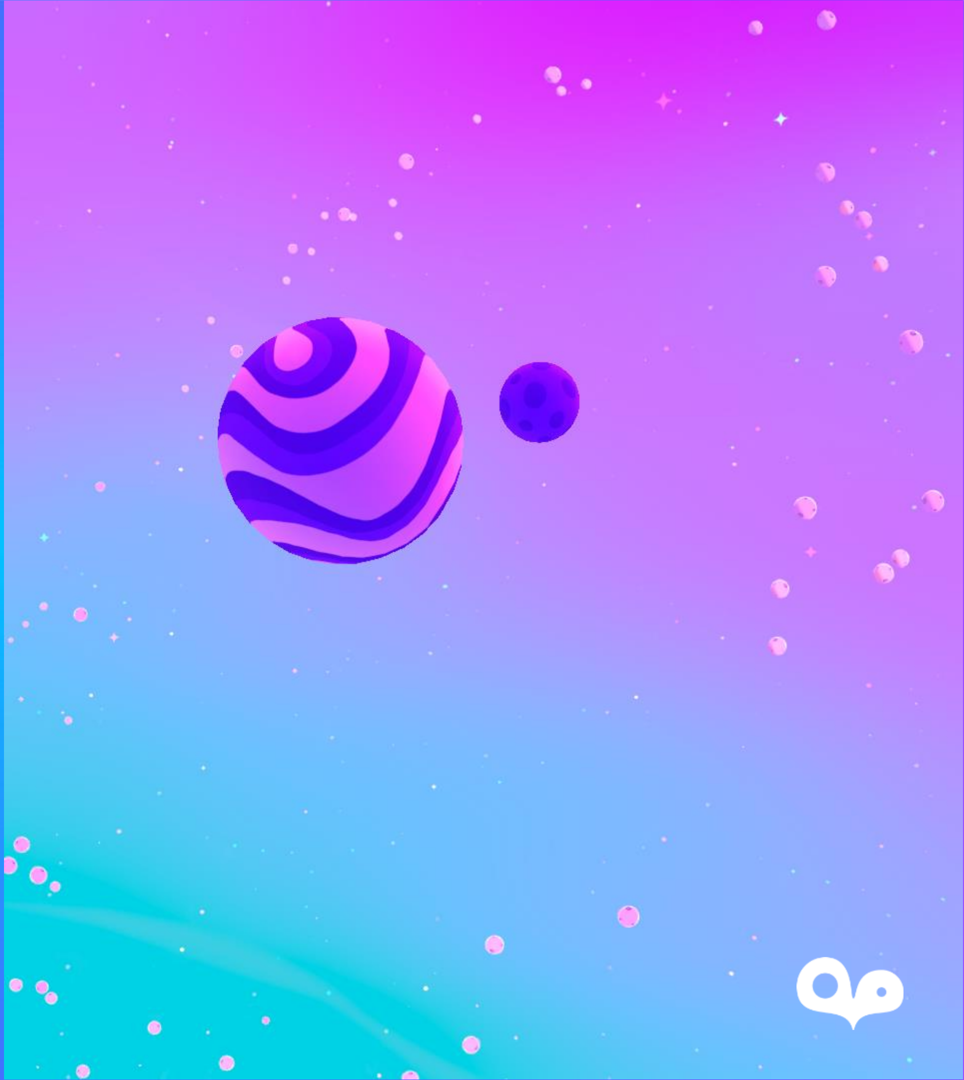
Editor script packs positions and normals into circle mesh's UV0/UV1/UV2 and exports to a new packed mesh asset.



Each instance can smoothly morph between the 3 shapes via a single shader property.

Eye shapes are also packed into RGB channels so only 1 texture read is required.

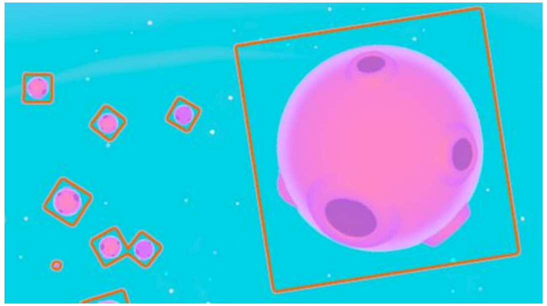
Lighting an Asteroid Belt



Lighting an Asteroid Belt

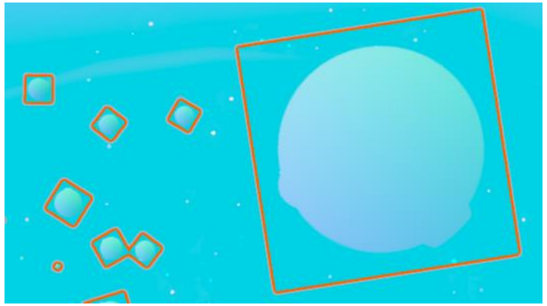
- Large numbers of asteroids orbit planets in our skyboxes
- Don’t want to use mesh particles
- Would rather not pay for a normal map
- Generate spherical normals for billboards based on particle center!

Lighting an Asteroid Belt



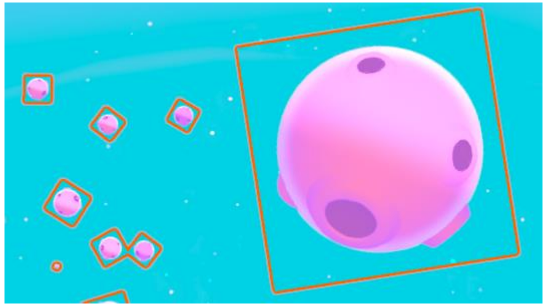
For our asteroid field we use a regular particle system.

Asteroids are basic textured billboards rotating around a planet.



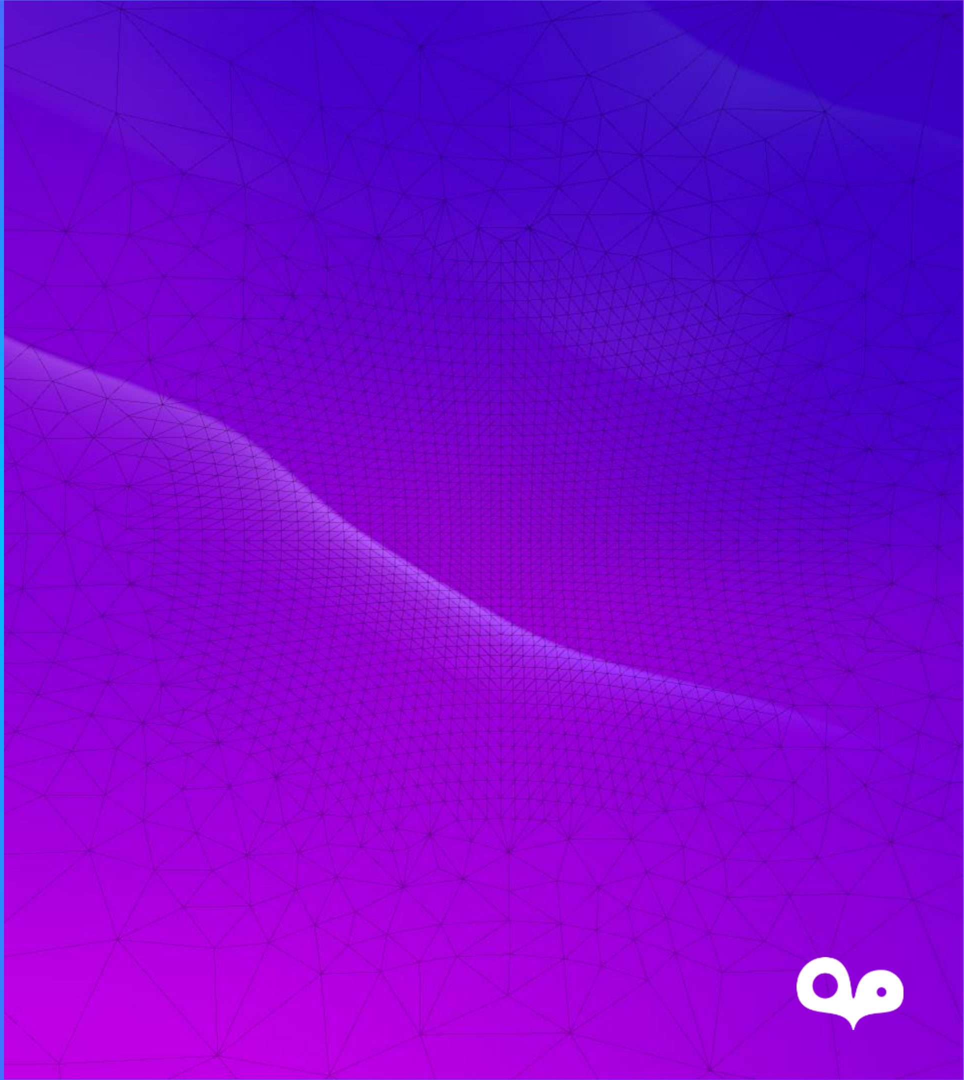
In the vertex shader we calculate the particle's worldspace center.

In the fragment shader we calculate the vector from worldspace center to fragment position and normalize.



Using this calculated normal, we perform lighting as usual!

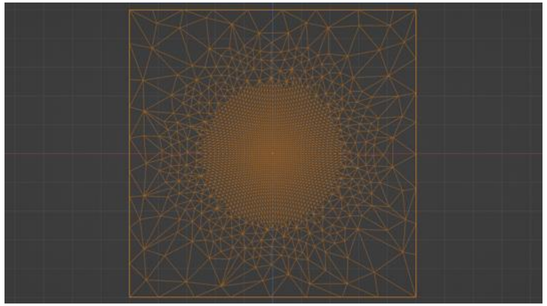
Vertex Shader Based FFR



Vertex Shader Based FFR

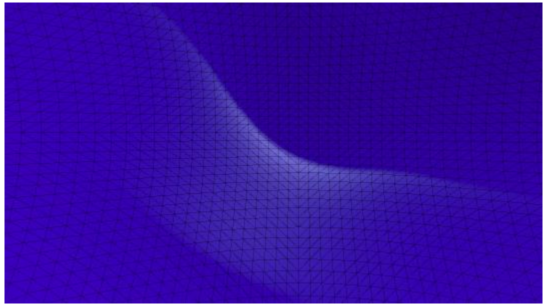
- Our skybox shader produces semi procedural animated nebulas
- Looks awesome but required several texture reads and some math to calculate the underlying animated noise, per fragment
- Sky is highly visible from a lot of vantage points in almost all scenes
- New approach
  - Instead of a sky sphere we generate a custom mesh projected to the far clip plane
  - Mesh is tessellated in such a way that geometric density is highest in the center of view
  - Animated noise calculations are moved to vertex shader
  - Fragment shader uses vertex interpolated noise for shading
  - Essentially a very specific form of FFR

Vertex Shader Bassed FFR



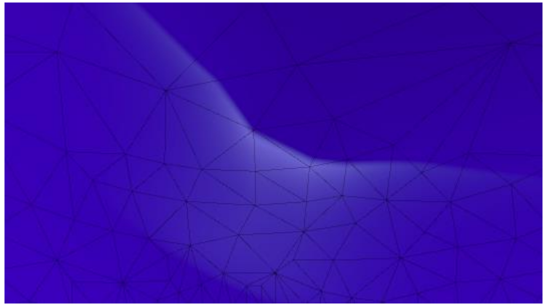
FFR tessellated plane hand authored in Blender.

Positions are already in clip space so we can return them unaltered.



Animated noise is calculated per vertex, shading per fragment.

Nebulas maintain their smooth shapes and motion around the focal point.



Nebula quality decreases towards the periphery.

This is unnoticeable in the HMD!

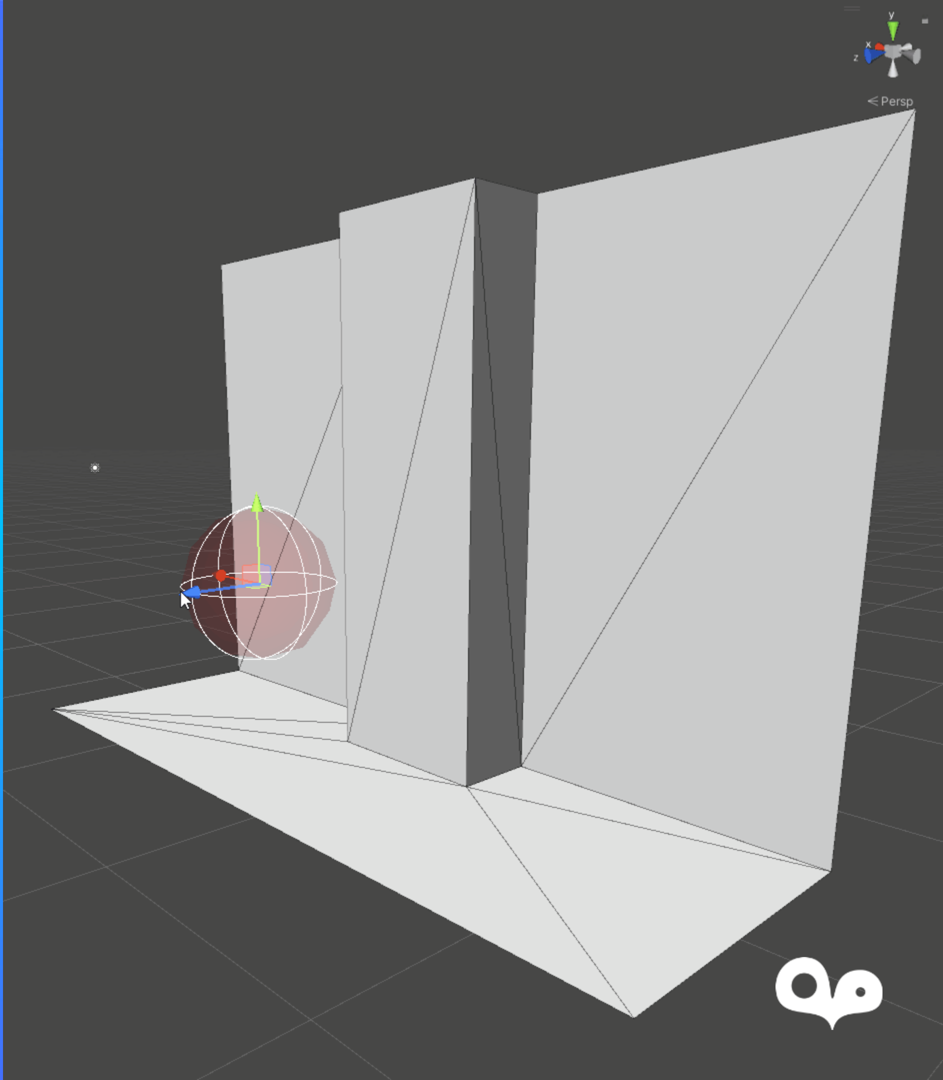
SplatTech
Quest Rewrite

SplatTech Features

SplatTech Features

- Almost every surface in the game splatable
- Multiple substances
- Substances can be removed over time/via player input
- Splats are persistent between scenes

Initial Implementation



Initial Implementation

- All splatable meshes require a unique UV set e.g. lightmap UVs
  - Enables read/write to mesh’s SplatMap
- Splats are added via physics system sphere collisions
- New splats are queued per frame
  - Worldspace position, radius
  - Substance properties; color, wetness etc.
- SplatMap textures are updated per frame using a MRT shader
  - New splat data is passed to shader as float4 arrays
  - SDF is updated via simple sphere distance check to fragment worldspace position
  - Substance properties are interpolated/attenuated
  - Subtracted SDF used to remove/evaporate certain substance
- In-game shaders read SplatMap textures and shade splats inline
  - SDF is eroded with noise for high frequency edge detail
  - Color/substance data displaced with noise to hide low resolution
  - Fake lighting calculated based on substance data

Initial Issues

```
APP/TW/GS T   FOV:      0 EB H:     1584
              BAT:     62 CPU U:       5
              CPU L:    4 GPU U:      99
              GPU L:    3 TW T:     1378
              FPS:      2 GS T:        0
              EARLY:    0 APP T:   65535
              STALE:   90 ASW FPS:     0
              EB W: 1440 C STALE:      0
```

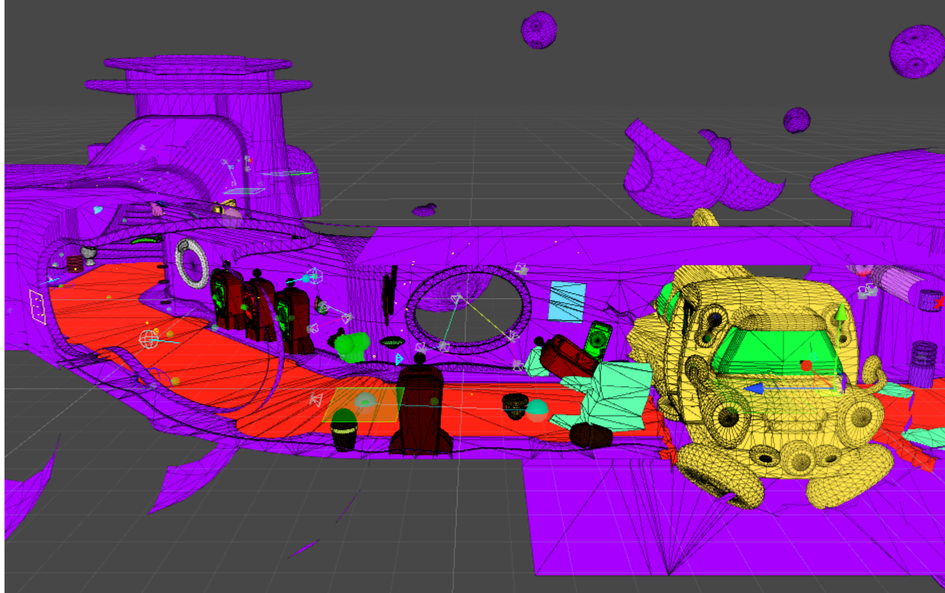
Initial Issues

- Separate SplatMaps for every mesh (including instances)
  - Memory/texture update nightmare
  - Completely breaks batching
- Textures per SplatMap; at start of dev this was 3, managed to reduce to 2
  - This is still 2 textures that need to be updated for every SplatMap and read in every shader that includes Splat shading
- Shading cost added to all splatable surfaces, which is most of them, even if the surface currently has no splats
- Worth noting; this was the implementation we used up to the last 6 months of development!

Quest Implementation

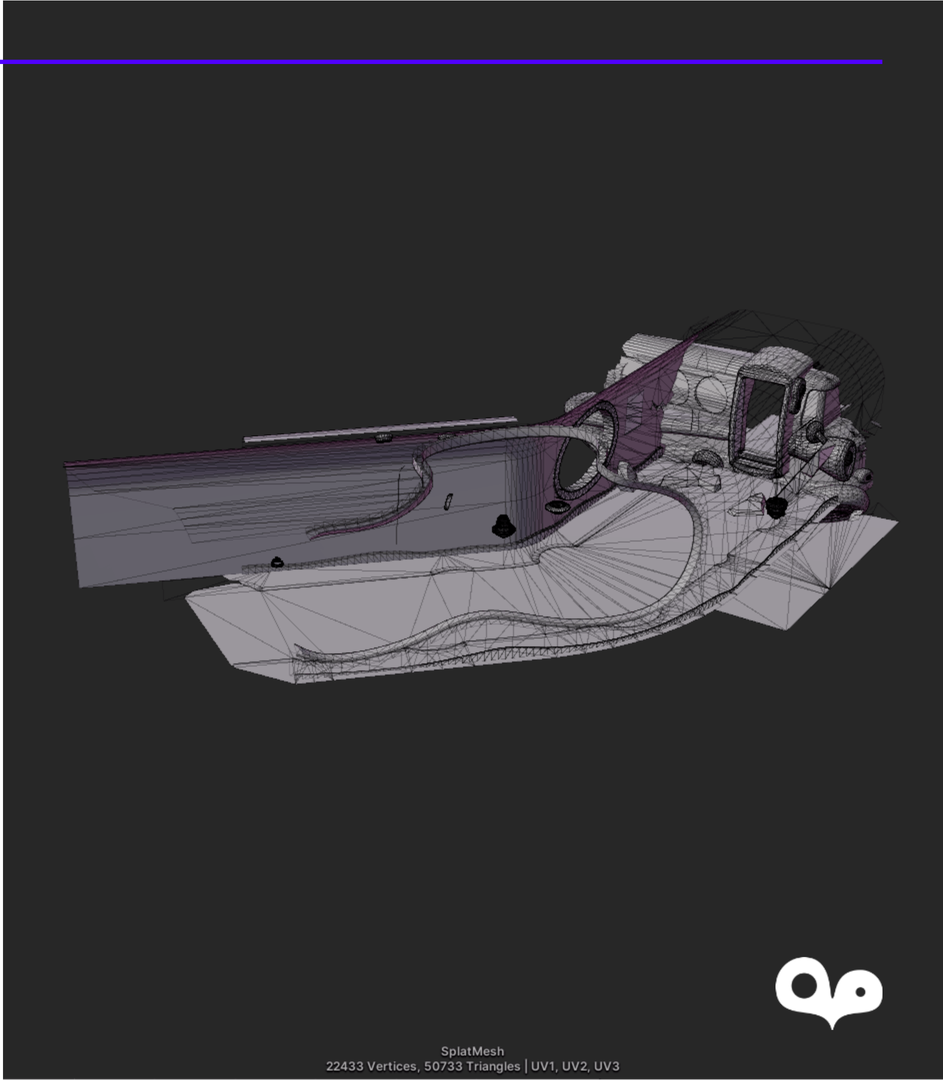
Quest Implementation

- SplatMeshVolume
  - Generate a single mesh around the player
  - 1 update per frame and 1 drawcall per update
  - We control how far from the player we pay for splat shading
  - Decouples splat shading, no more broken batching
  - If we render this mesh last, we essentially get a depth pre-pass. No more splat shading cost from overdraw!
  - We only need to update this mesh when a player teleports



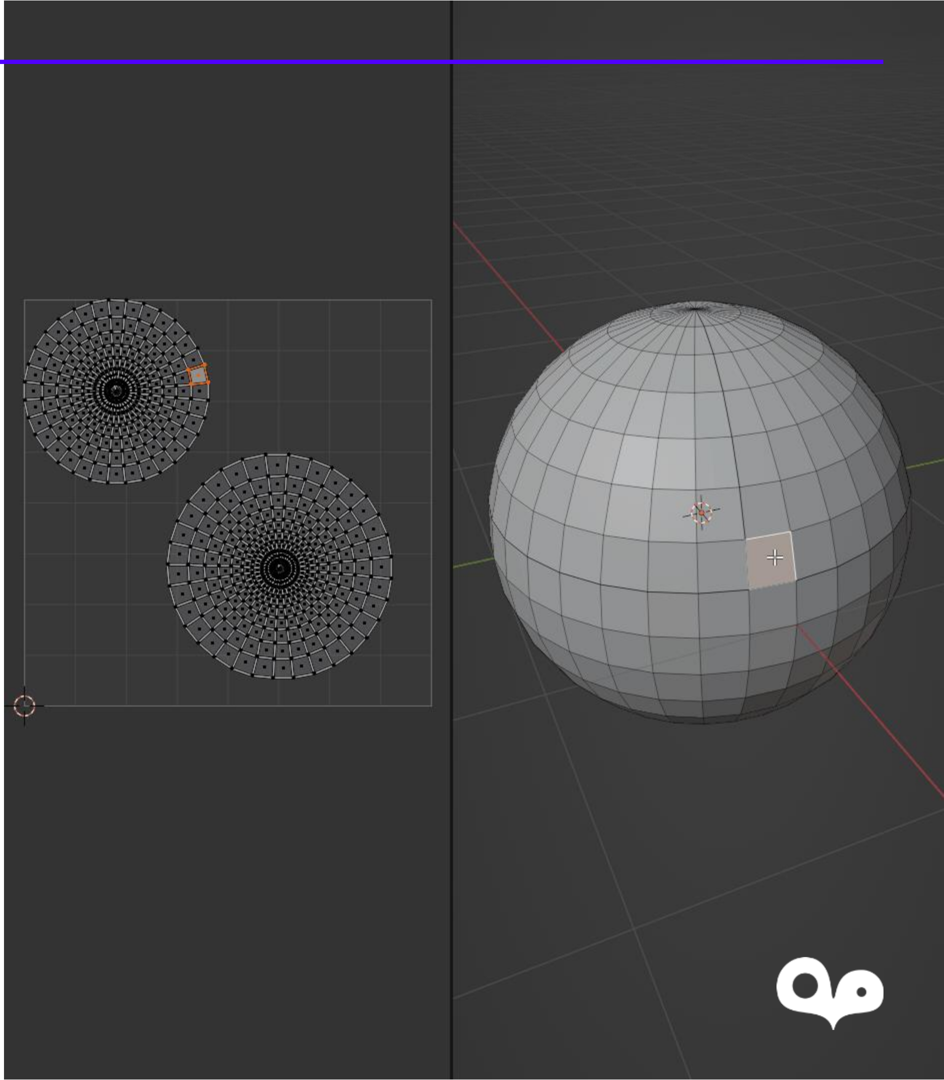
Quest Implementation

- SplatMeshVolume
  - Collect all splatable geometry
  - Cull triangles outside of 10m radius
  - Create single combined mesh
  - Pack UV islands into single texture
    - Remember we're starting with many separate meshes, each individually unwrapped to the full UV space
    - We're culling at the triangle level so simple texture packing would result in a lot of wasted space


SplatMesh
22433 Vertices, 50733 Triangles | UV1, UV2, UV3

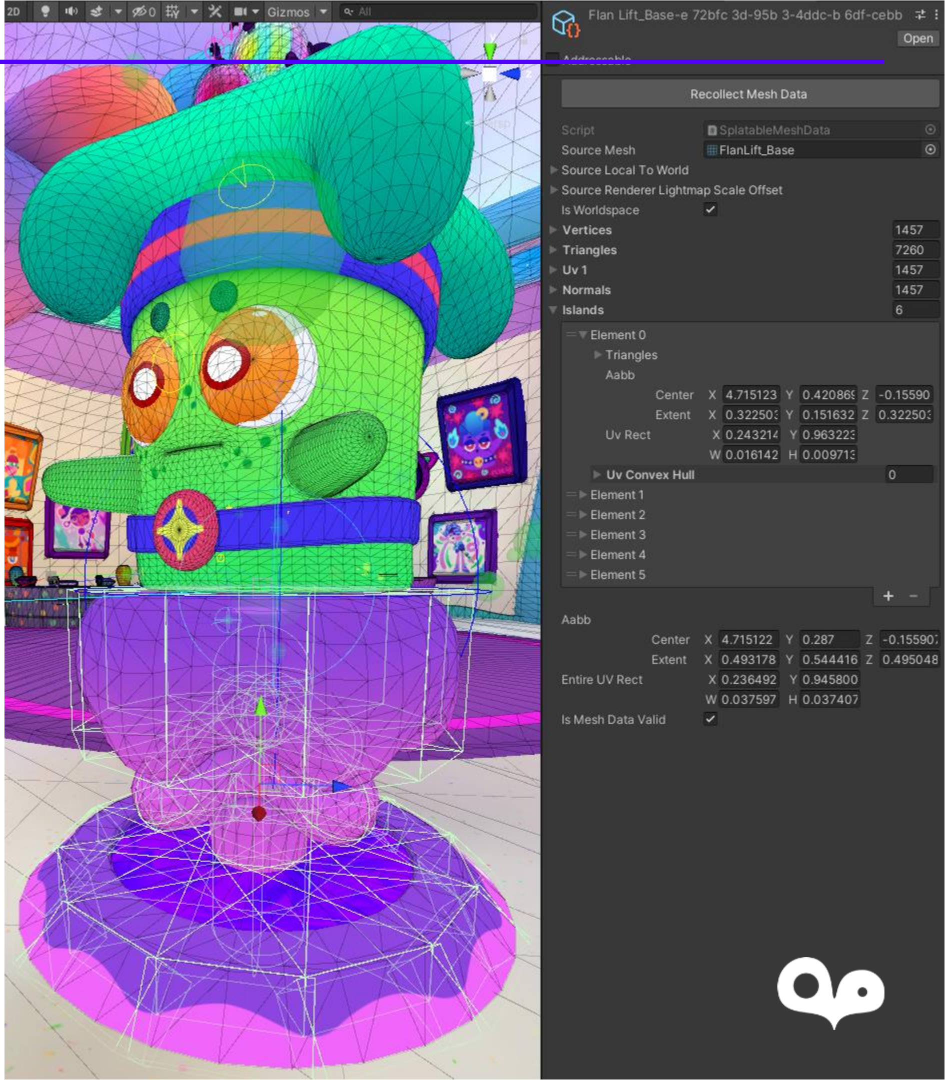
Quest Implementation

- Finding UV islands
  - UV islands split geometry
    - So an island is a sub group of interconnected triangles
  - Mesh assets don't contain adjacency info
    - We have to calculate this ourselves using indices
  - Geometry can be split for other reasons
    - For optimal packing, we have to handle this case too
    - Consider duplicate vertex attributes as equivalent to genuine shared vertices



Quest Implementation

- Offline processing for speed
  - Custom mesh assets with indices broken up into UV islands
  - Pre-transform positions to worldspace
  - Calculate UV and worldspace bounds per island
  - Sort vertex buffers by UV islands so we can fast copy whole islands that are completely within 10m radius

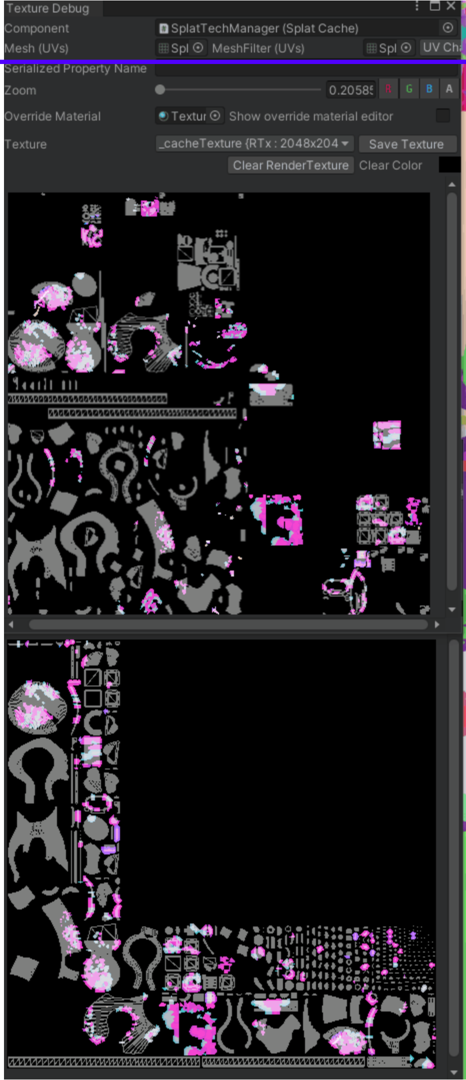


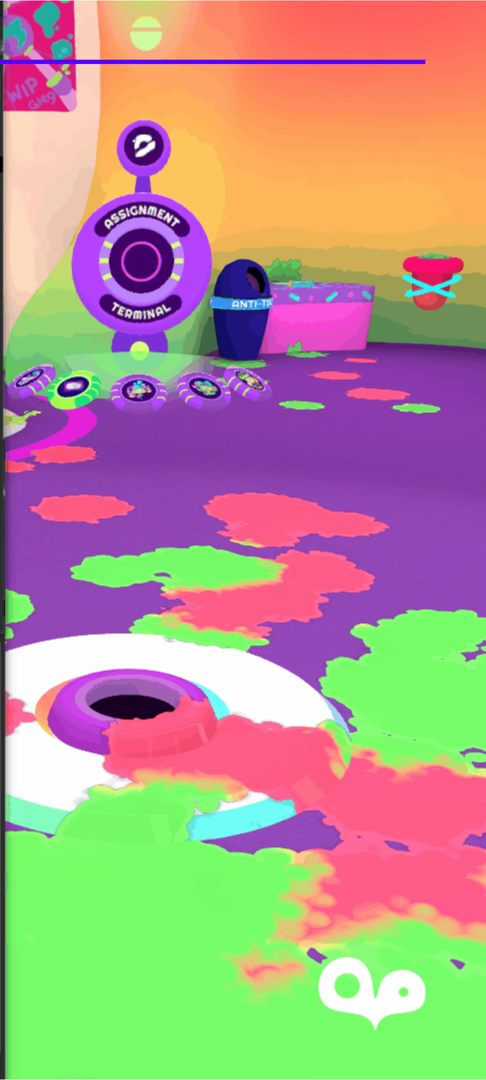
Flan Lift_Base-e 72bfc 3d-95b 3-4ddc-b 6df-cebb

Script	SplatableMeshData		
Source Mesh	FlanLift_Base		
Source Local To World			
Source Renderer Lightmap Scale Offset			
Is Worldspace	✓		
Vertices	1457		
Triangles	7260		
Uv 1	1457		
Normals	1457		
Islands	6		
Element 0			
Triangles			
Aabb			
Center	X 4.715123	Y 0.42086	Z -0.15590
Extent	X 0.32250	Y 0.151632	Z 0.32250
Uv Rect	X 0.243214	Y 0.96322	
	W 0.016142	H 0.00971	
Uv Convex Hull	0		
Element 1			
Element 2			
Element 3			
Element 4			
Element 5			
Aabb			
Center	X 4.715122	Y 0.287	Z -0.15590
Extent	X 0.493178	Y 0.544416	Z 0.495048
Entire UV Rect	X 0.236492	Y 0.945800	
	W 0.037597	H 0.037407	
Is Mesh Data Valid	✓		

Quest Implementation

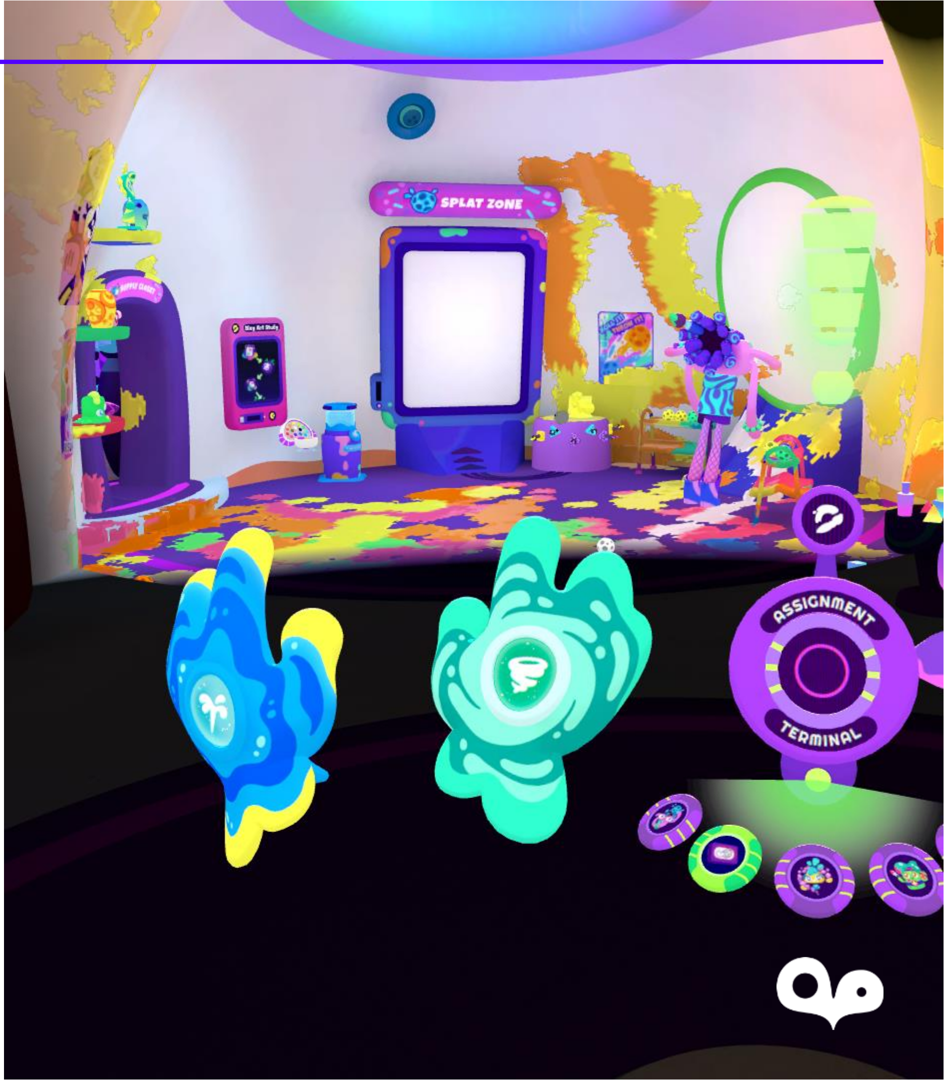
- Persisting splats with SplatCache
  - We have lightmap UVs, let's use them!
  - Store scene's splat data in large textures
  - Before generating new splat mesh, write local splat data to cache
  - After generating new splat mesh, read splat data from the cache





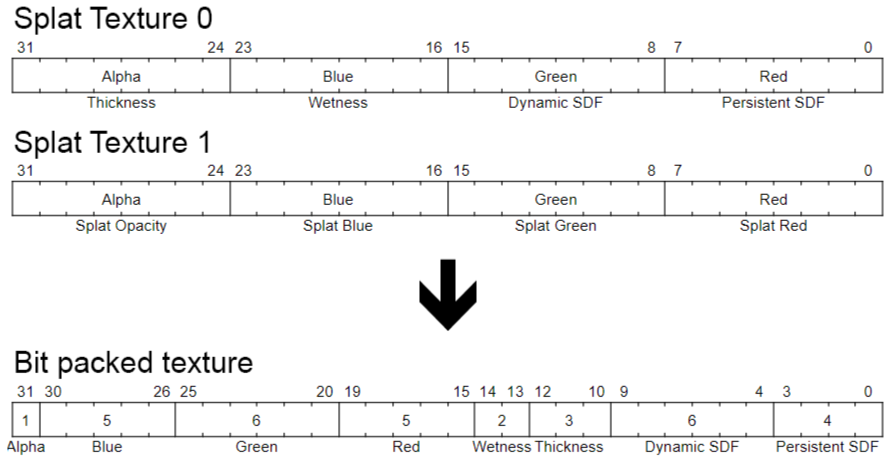
Quest Implementation

- Baking far-field splats
  - 10m is pretty far from the player so resolution is less important
  - Use SplatCache to also bake splat *shading*
  - Shaders beyond 10m blend baked splats with a single texture read
  - SplatMesh shader fades at 10m boundary



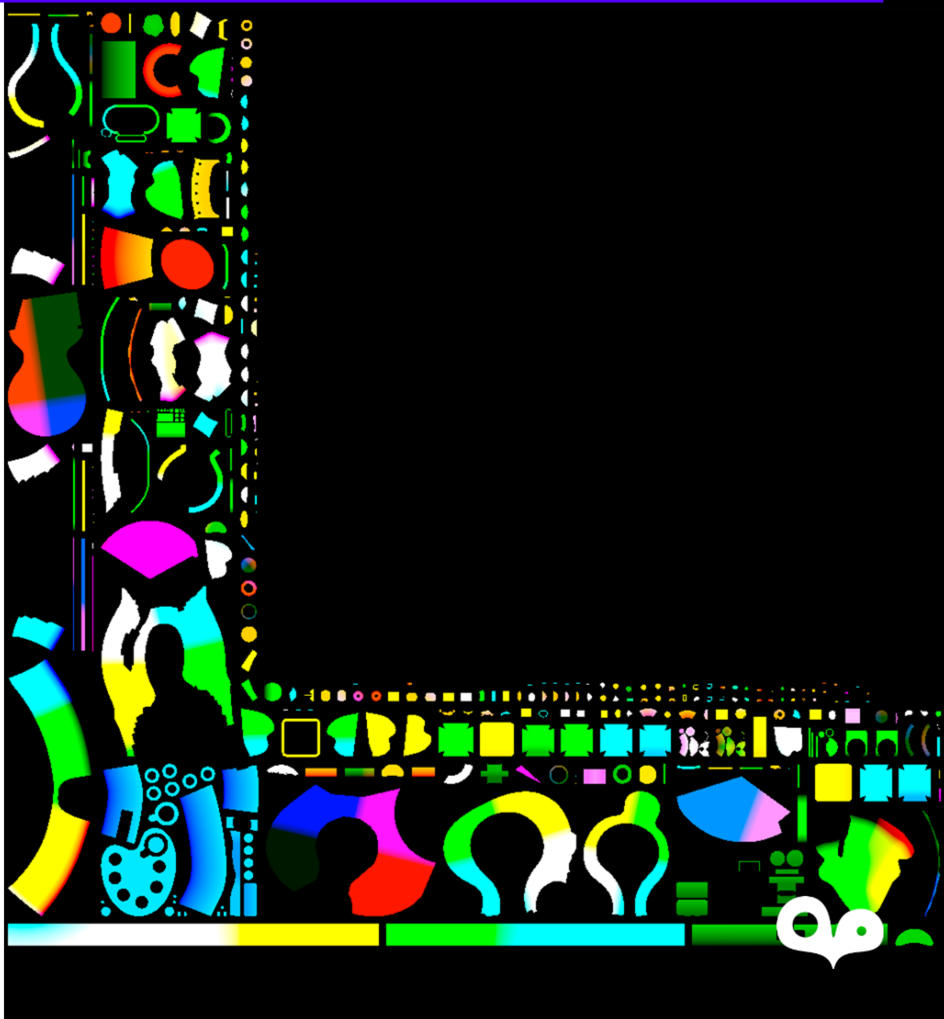
Quest Implementation

- Packing 2 textures into 1
  - Use a single R32_Uint texture
  - Bitwise ops *are* supported in GLES3 but in tests were slower than their arithmetic equivalents
  - For sanity wrote a collection of macros for:
    - Encoding/decoding
    - Each parameter has a single #define BITCOUNT
  - Bit packing means no HW filtering ☹
    - Can use Texture2D:GatherRed() to perform manual filtering *in* the shader, but this was too slow for us
    - We solved this another way…



Quest Implementation

- SplatMap update
  - Update & dilation in a single compute shader!
  - Including dynamic objects
  - Use fast **groupshared** for dilation step
  - Compute doesn't have access to geometry
    - Bake worldspace positions to **ARGBFloat** texture
    - Pack geometry ID + dilation offsets into alpha
  - Decouple update frequency to 30hz
  - Unpack 32bit texture to two 16bit textures
    - Gives us HW filtering and mipmaps!



Quest Implementation

On teleport:

- Write local splats to cache
- Bake local splat shading
- Generate new splat mesh
- Bake worldspace positions
- Read local splats from cache

Update:

- Frame 1: splat update
  Dispatch compute shader to apply new splats, bit pack data and perform dilation
- Frame 2: resolve
  Unpack splat data, combine persistent and dynamic SDFs and output to 16bit MRTs

Rendering:

- Single splat shader
- Splat mesh HW blended at end of opaque drawcalls
- Depth test/write enabled
- In our very specific case **clip()** actually provided a performance boost
- Splats are antialiased using **fwidth(sdf)**

Key Takeaways

Key Takeaways

- No one size fits all solution to optimization
- Use all the tools at your disposal to find areas in *your* game that could be hiding pockets of potential performance. Get creative!
- Use methodical, deterministic testing to keep yourself honest. Prove your optimizations are *actually* making things faster
- If the tools you need don't exist, write them!

Building Big Games on Little Headsets

Thank you!

@kode80
@OwlchemyLabs
owlchemylabs.com

Ben Hopkins
Principal Graphics Engineer
& Mixologist of Realities

Owlchemy Labs