Intro to Eagle Flight:

- Started in 2014 as an experiment to see what was possible in VR
- Locomotion was the biggest challenge so we decided to focus on that
- Flying mechanic was fun and comfortable
- To really get the freedom of flying, you need a big playground to fly around in

Controls, comfort and performance are all so connected that we'll talk about them all today.

CLICK: Announce trailer – let's get a better feel for the game

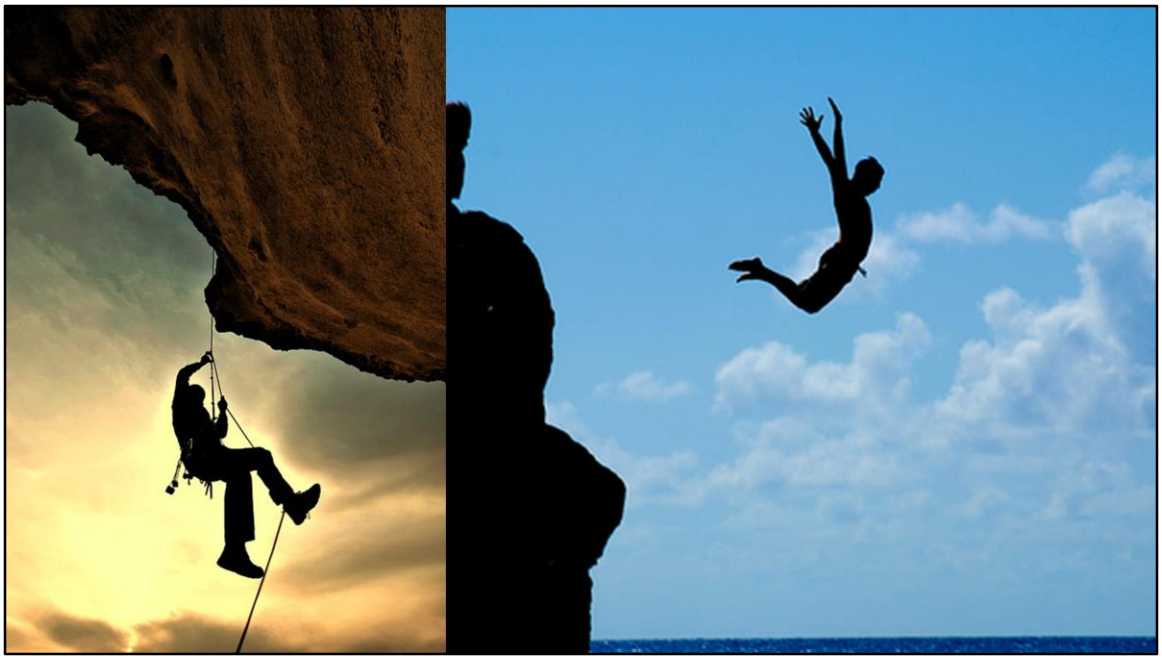
Announcement video

As you can see, there is a lot of city visible when you're flying in our game, and you can visit almost all of it.


•Flying controls
•Comfort
•Performance
•Multiplatform VR dev tips

Overview

Go where you look
NO drift – must be 1 to 1
Continuous forward motion



Despite common advice to avoid acceleration, we found that it felt more natural to slow down when you go up and speed up when you go down.

If you don’t speed up when you fall, you feel like something is “suspending you”, like you’re sitting in a harness and it feels really unnatural. Can actually make you feel more uncomfortable! You don’t get that feeling of freedom we were going for.

Slowing down when you go up rarely is noticeable to the player since you’re mostly looking at sky, but it adds a greater challenge for navigating and a bit more gameplay depth.

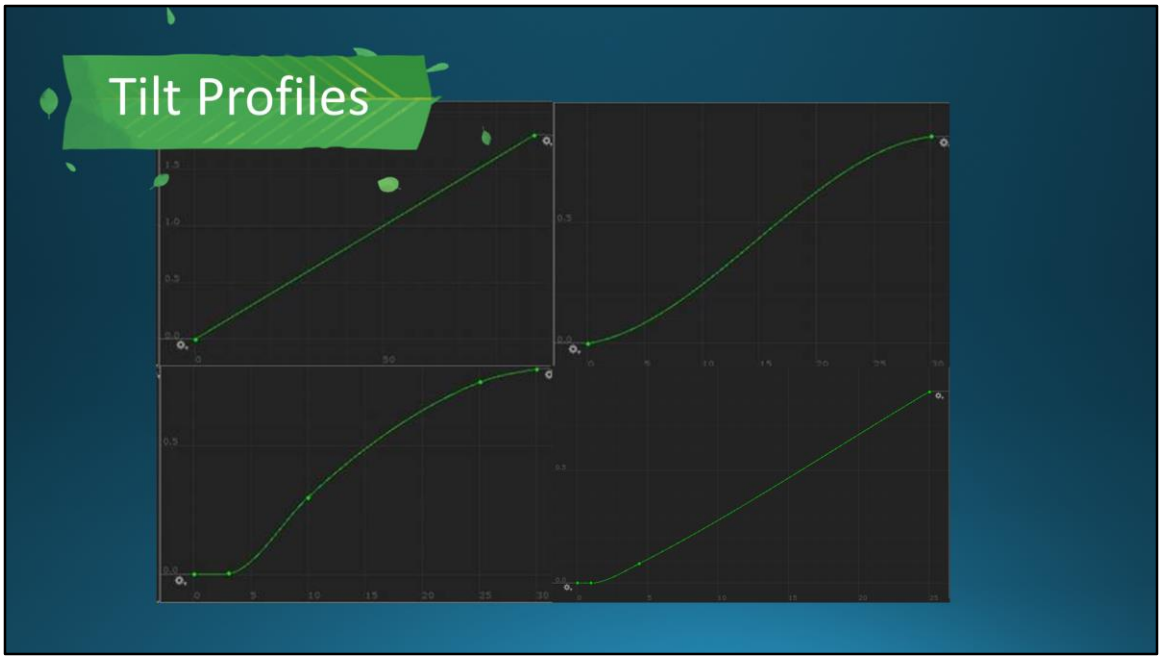
But sometimes you just want to look around at the world around you without flying in that direction and potentially colliding with the landscape.

For this we added a button for free look. You'll continue to fly in the same direction, but for as long as you hold the button, you can look around so your view is decoupled from your flying direction.

The big question is what should happen when you release the button. We don't want to snap the camera back to your moving direction, because that can cause major nausea. Smoothing the camera transition back to that same direction doesn't work either, because we're violating the fundamental tenet that you need to look where you want to go.

So as soon as you release the button, you'll start flying in the new direction you're looking. This has the added advantage of enabling precise, quick turning with the free look button. It's a skill that takes practice but is used frequently by the best players in multiplayer.

Talk about tilt mechanic

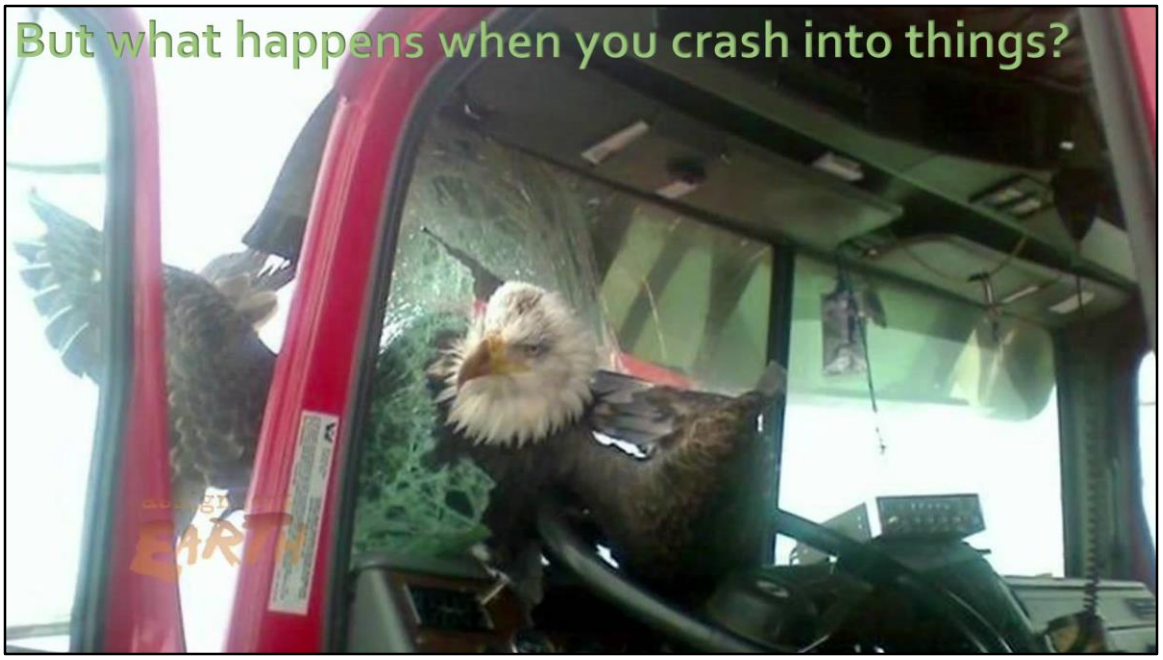


Top Left: original profile
Turn rate depending on tilt of your head is linear
You can tilt your head up to 90 degrees to turn faster
The turn rate doesn't depend on your speed

Top right: new profile
Turn rate depending on tilt of your head grows less quickly at low and high tilt, to smooth it out
Max turn rate is reached at 30 degrees of head tilt
At max speed the turn rate is only 70% of what it is at min speed

Bottom left: Dead zone profile
Turn rate is0 if you don't tilt your head more than 3 degrees
Max turn rate is reached at 30 degrees ofhead tilt
Max speed the turn rate is only 50% of what it is at min speed
Targeted to be the most comfortable

Bottom right is what we ended up with. Small deadzone so you don't start tilting right away (1 degree), but linear turning afterwards.



You can't just stop the player in their tracks. It's like hitting a physical wall, and it goes against our tenet of continuous physical motion.

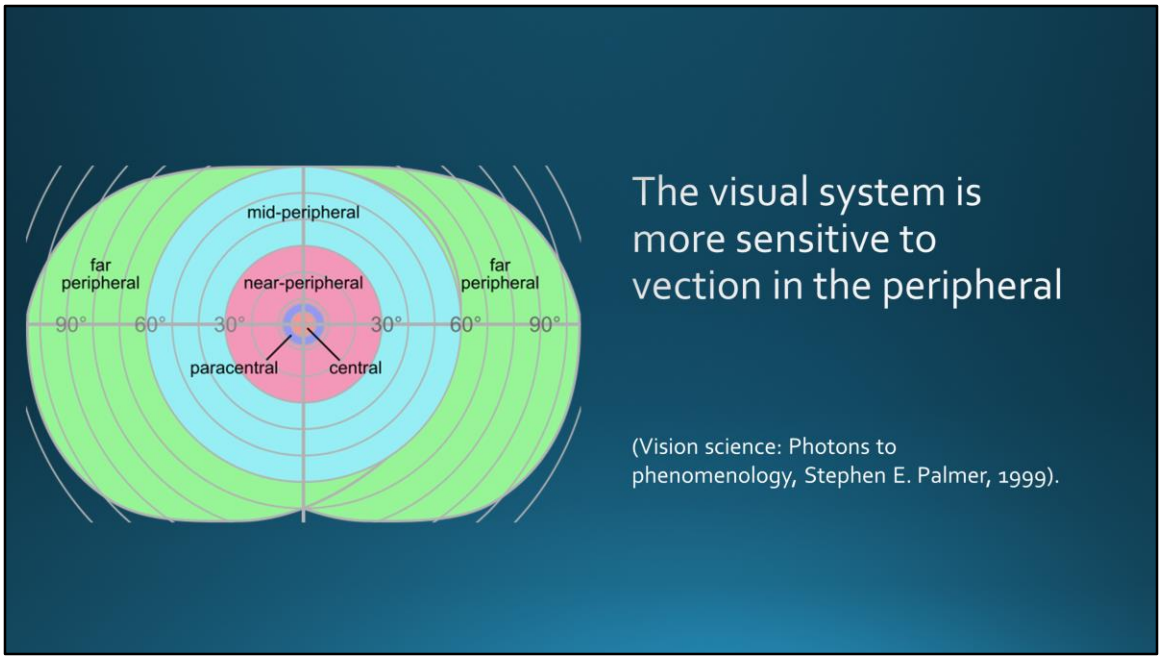
The easiest is to just keep flying, but this is just weird, and it’s not very interesting from a gameplay perspective either.

Fade quickly to black screen, but notice the wind particles that move in the direction of flight keep going.

Avoid feeling like you've hit a brick wall
Feel like you're still moving, just somewhere else

Speaking of comfort, let's talk about vection.

Vection is the feeling you get when a large part of your visual field moves, so you start feeling like you've moved and the world is stationary. This can be a cause of motion sickness, because your visual field disagrees with your inner ear.

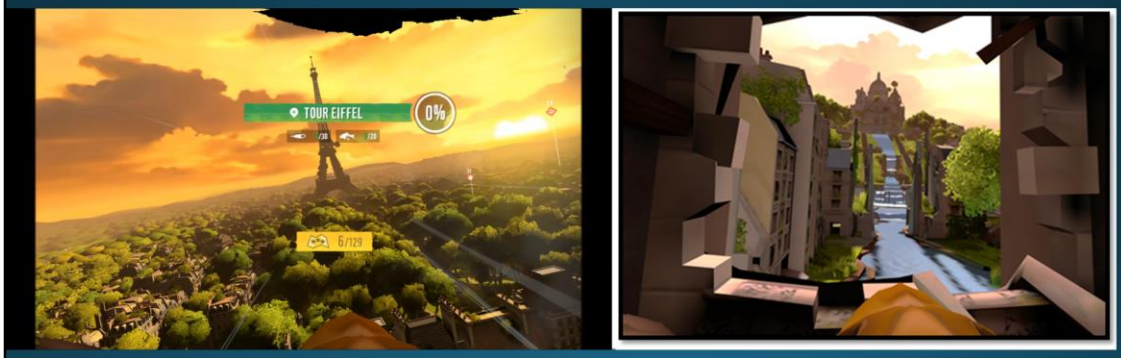


We all know the center of our vision, the fovea, is very sensitive to detail, whereas the peripheral is less so.

However our peripheral vision is really sensitive to motion. TIGER.

The research also shows it's very sensitive to vection, and that vection in the peripheral is the most important.

Two cases of vection in Eagle Flight:



In Eagle Flight there are two scenarios of vection we need to address since they cause cybersickness:

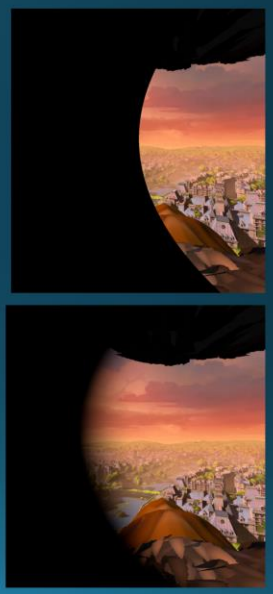
Vection that occurs during illusionary self-rotation of the virtual body (with head tilting)

Vection created by proximity: flying at high velocity close to objects and walls

Parabolic Tunnel Vision

Parabolic Tunnel Vision

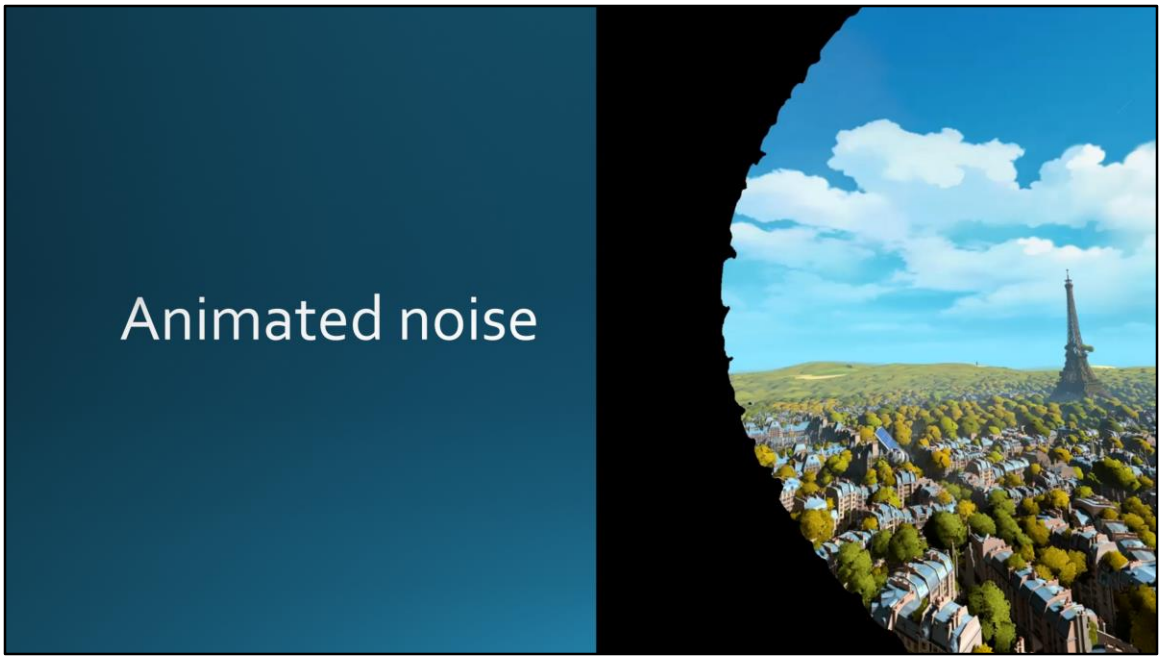
- Alpha-blended full screen effect
- 4 parabolas: top, bottom, left, right
- Vection-based curvature



How do we solve this?

Cover vection in the peripheral dynamically
Add noise to the Optic Flow

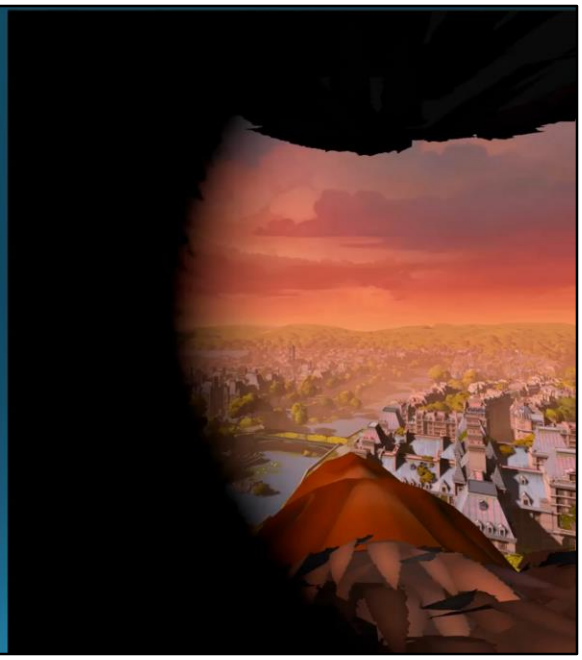
Fade to black a pixel if its screen-space position is on the negative side of the parabola function. The closer is the pixel to the parabola, lesser it will be faded.



Animated noise is applied on the pixel fade. Noise is an input texture and animated with a pseudo-cylindrical texture mapping.

Here's the noise without the alpha blending to get a better idea of what it looks like. Notice the noise is moving in a different direction than the parabola would be.

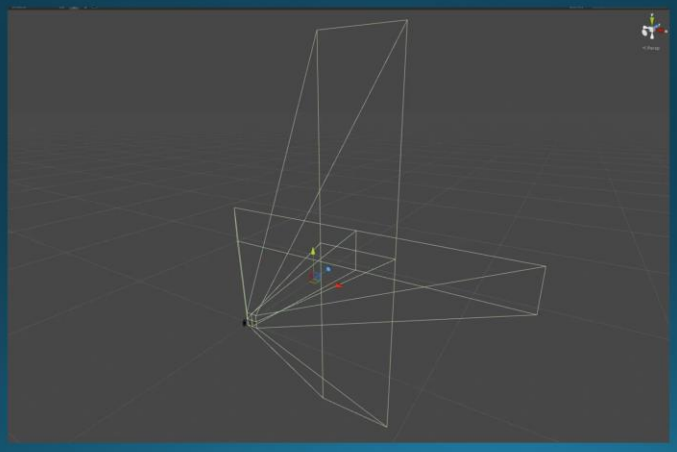
Animated noise



Animated noise is applied on the pixel fade. Noise is an input texture and animated with a pseudo-cylindrical texture mapping.

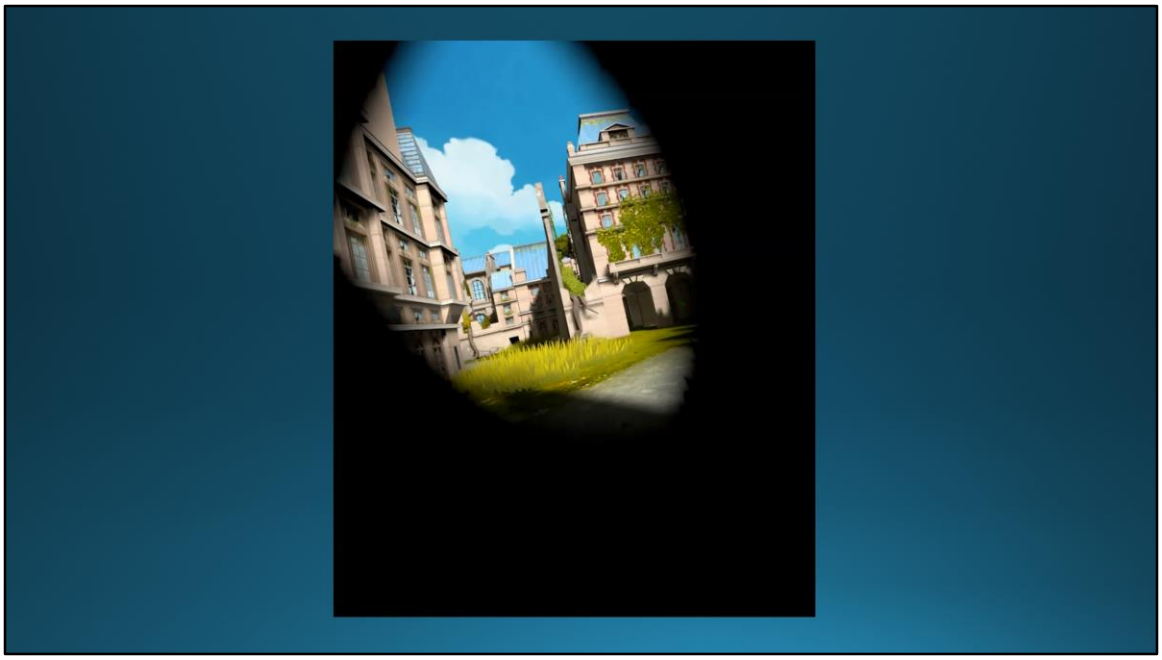
And here’s what it looks like with the fade applied.

Proximity Detector System



4 detectors (top, bottom, left, right) detect objects, walls, ground, etc
Each is mesh collider that intersect with environment colliders
When a collider is hit, activate associated parabola

So now that we have a way to trigger the parabolae, let's take a look at what they look like in action.



Gross!

As the colliders hit things, the parabolas are activated. But this isn't very comfortable or subtle. You need to add some temporal smoothing for a more comfortable experience.

Temporal Smoothing

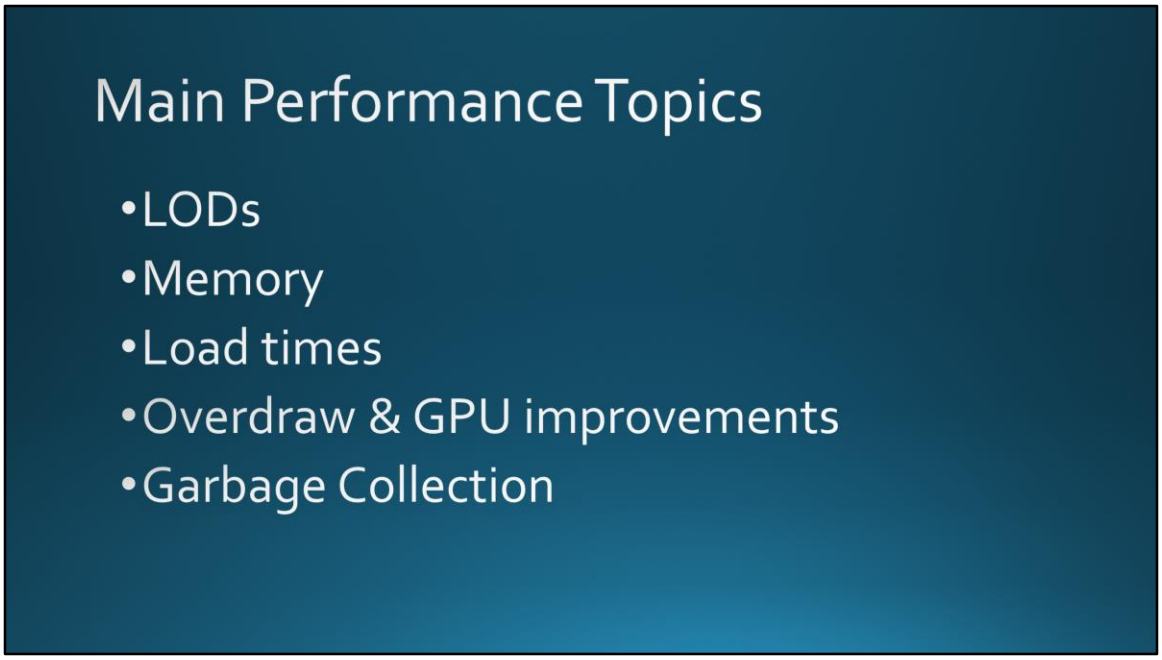
Parabolic parameters are temporally smoothed
Different smoothing applied based on different events

For more on Comfort…

Go see our Game Director Olivier Palmieri's talk, room 2024 West Hall:

- Tuesday at 11:20 am
- Wednesday at 3:30 pm


Performance



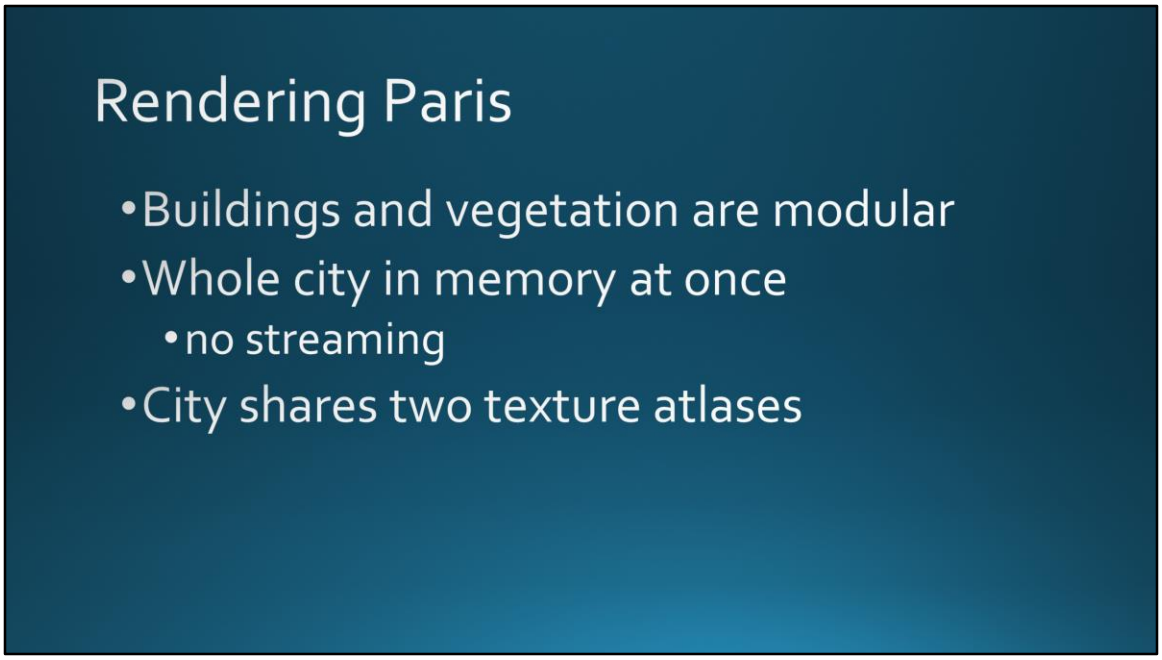
Overview of perf section

VR Performance Considerations

	PC	PS4
Frame rate	90 fps (11.1 ms/frame)	60-120 fps (16.6 ms/frame)
Resolution	1080x1200 per eye	960x1080 per eye

Pixel throughput ~7x higher than 1080p @ 30Hz!

A Refresher on VR performance

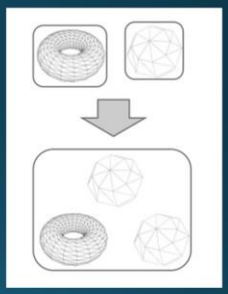


Because you're high above the city and you can turn in any direction with your head, you need to be able to see pretty much the whole world at any given time. We decided to load all of Paris into memory at once to minimize hitching that might occur when loading in other parts of the city. We broke the scene into several smaller subscenes for efficient editing but at runtime it was considered one unit.

And we used two 4K by 4k texture atlases for all of the regular city buildings, so we could batch them as much as possible. We use one for opaque and 1 alpha tested

Performance – Stuff to try…

- Occlusion culling
- Static batching
- GPU Instancing



Here are some optimizations Unity generally recommends and why they weren't sufficient for us.

Occlusion culling
- Yields little benefit when you can see most of the city most of the time
- CPU overhead

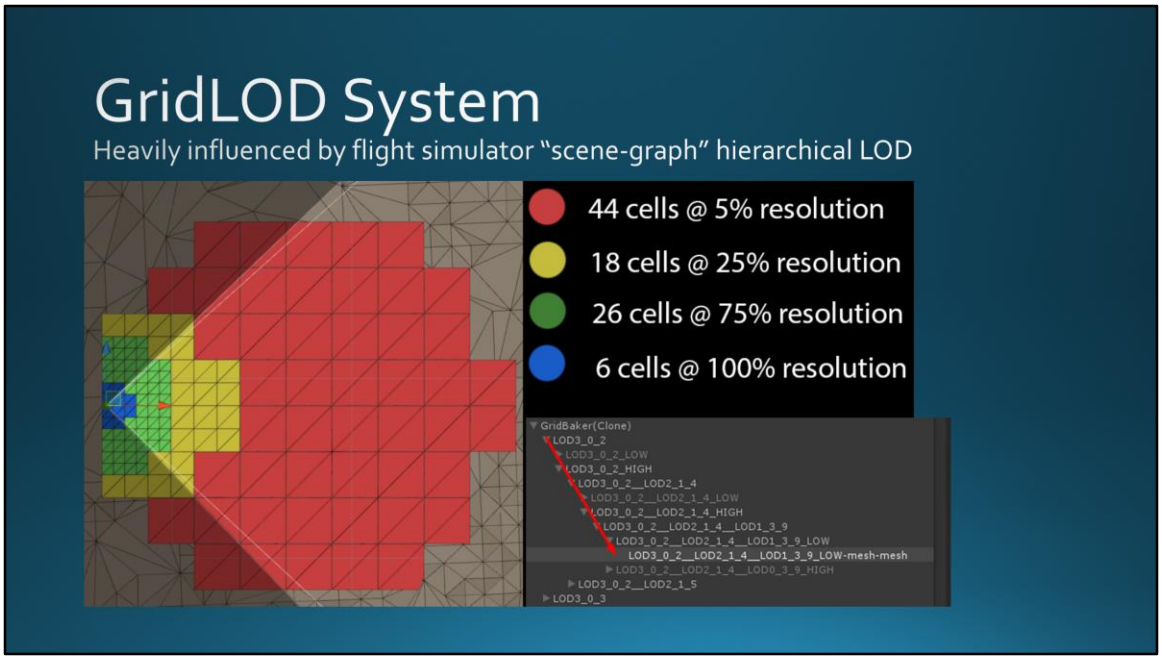
Static batching
- Automatic in Unity, but you don't have any control or much visibility into what is happening
- Batches too small to be useful

GPU Instancing
- Came in late
- Total # of polygons is still too high
- LODs reduces size of GPU instance batches -> less payoff

Bake your own data!

- Mesh Baker Unity plugin bakes LODo offline
- Mesh Baker LOD plugin calculates at runtime only
- Roll your own LOD system…

MeshBaker LOD plugin bakes LOD meshes into combined meshes, but only at runtime



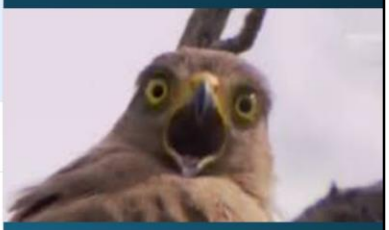
The GridLOD system is fairly simple and heavily influenced by flight simulator "scene-graph" hierarchical levels of details.

It consists in reorganising the scene assets into a grid pattern of 256m x 256m. For each level of detail, we then bake their geometries into clusters of varying dimensions ranging from 64m to 256m. This allows to switch progressively to larger clusters without having too much popping occurring on screen. All these merged objects are then hierarchically organised from low resolution to high resolution. Then, we simply traverse the hierarchy to determine if children should be visible depending on distance to the camera and parent's visibility.

GridLOD Results

Effectively reduces the CPU Render and culling by ~90% without affecting the GPU at all

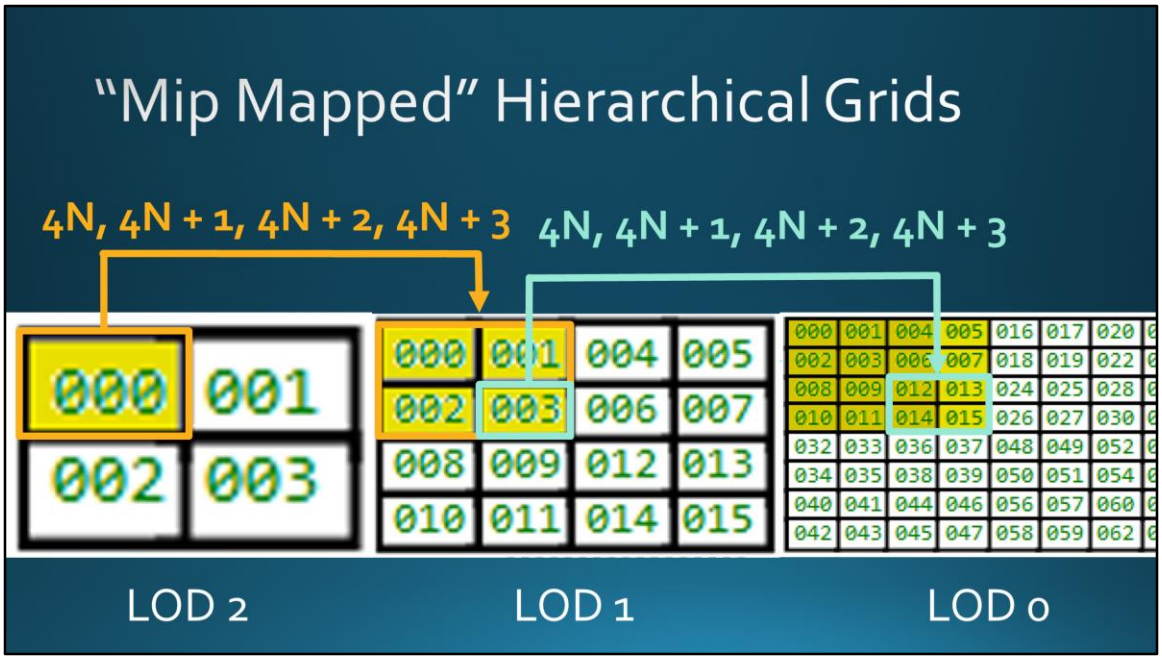
	Draw Calls	Batched VBO	CPU (ms)		GPU (ms)
			Culling	Prepare +Render Opaque	Render Opaque
UnityLOD	4341	1187	0.39	2.91	1.961
GridLOD	189	189	0.03	0.18	1.956



Highlight CPU saving here

GridLOD challenges

- Keep content creation as agile as possible
  - Run as pre-build process to keep separated meshes in Editor
- 20 minutes to bake the geometry
  - Cache unchanged clusters
  - Discard unneeded lightmap UVs. Down to 1 min!
- Enabling/Disabling GameObjects is slow!



Quickly travel through the hierarchy without having to enable/disable GameObjects

Can go up and down hierarchy using simple math

BONUS: can now introduce dither-based smooth transitions

In order to quickly travel through the hierarchy of cells, instead of having a linked structure of cells that know their children and or / parent, we went for a different pattern that aims at arranging all the cells in a particular way, inside their respective LOD specific arrays, so that travelling up & down the hierarchy cells was trivial and super fast.

There's no hierarchical information in there, as we saw above than we can easily go up & down the hierarchy using simple math.

Obviously, to travel from LOD2 indices to LOD1 indices, one just have to use for formula :
4N, 4N+1, 4N+2, 4N+3

Artists add more city…

- We run out of memory.
- Time to start compressing!

Still, the city renderer was costing up to 4ms on the CPU and the geometry memory footprint ended up busting the 1GB PS4 memory budget as content was added to the game.

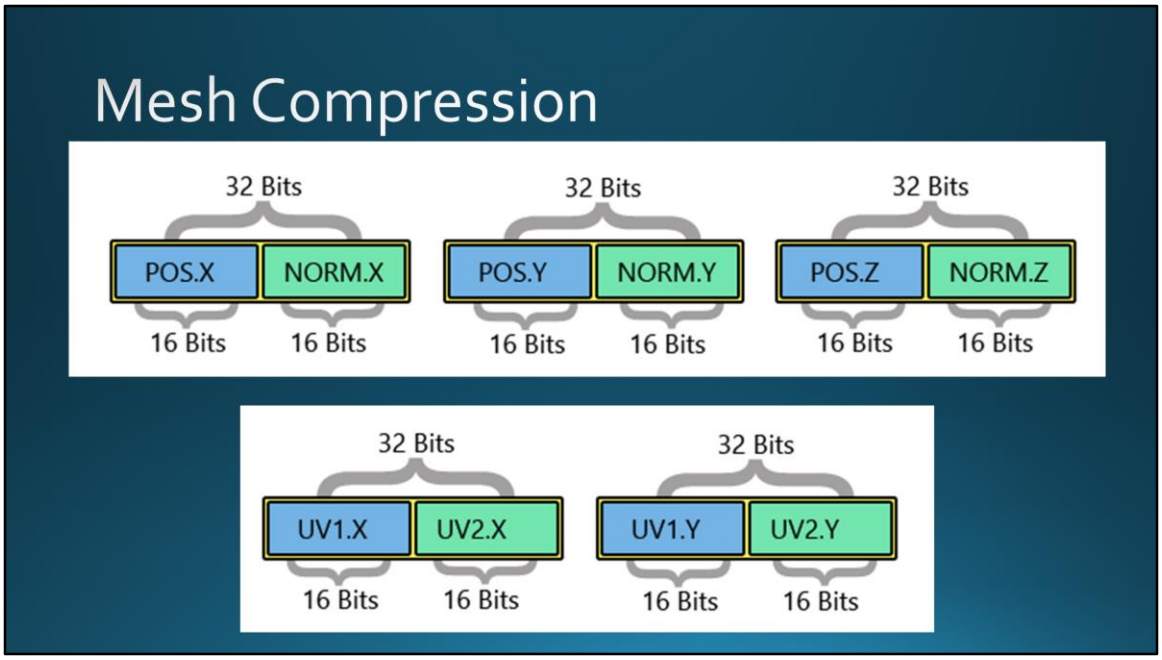
Unity's compression was not going to do the job, because it decompresses on load. So their solution only works to save disk space, not memory

Mesh compression – 1st try

- Custom rendering approach based on DrawProcedural() API and compute buffers?

Unity doesn't allow custom vertex formats for position, normals and UVs
1st try: custom rendering approach based on DrawProcedural() API and compute buffers
Reduced memory footprint to 30% of original budget, but: (CLICK)

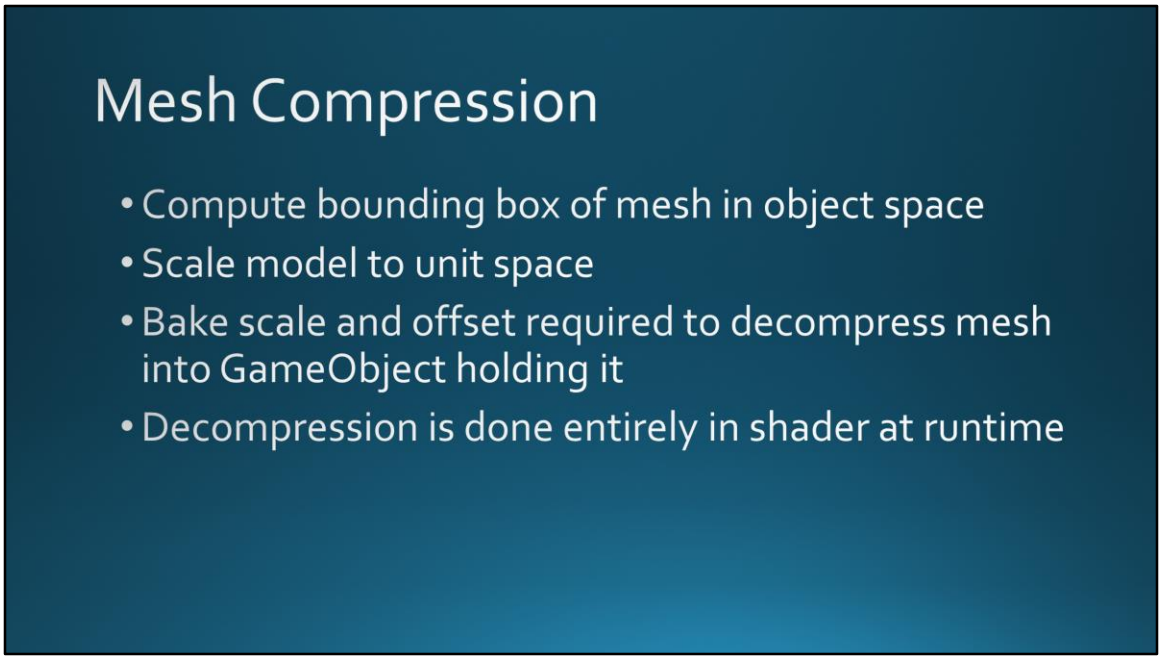
- GPU impact was too large
- Doubled rendering time in critical locations
- Bypassing vertex cache of GPU and processed a lot of extra vertices



Unity being a black box, we needed to find a way to encode multiple channels (POS, NORMAL, UV1, UV2) in the supported Mesh channels provided by Unity.
We use POSITION and UV1 of Unity, but encode all of our channels in there

Pack both Position & Normals into the mesh Position channel as **half**, interlaced to speed up unpacking

Same for UV1 and UV2 into mesh UV channel (also interlaced)



Decompression in shader

First, compute the bounding box of the mesh in object space
Scale down the model to unit space (-1,+1 range on all axes, center at (0,0,0)
Bake the scale and offset required to decompress the mesh into the GameObjects holding it

- Advantage: doesn't add matrix multiply in the shader
- Limitation: can't have another compressed mesh down in the hierarchy
  - Not a concern for us since all compressed meshes were on hierarchy leaves

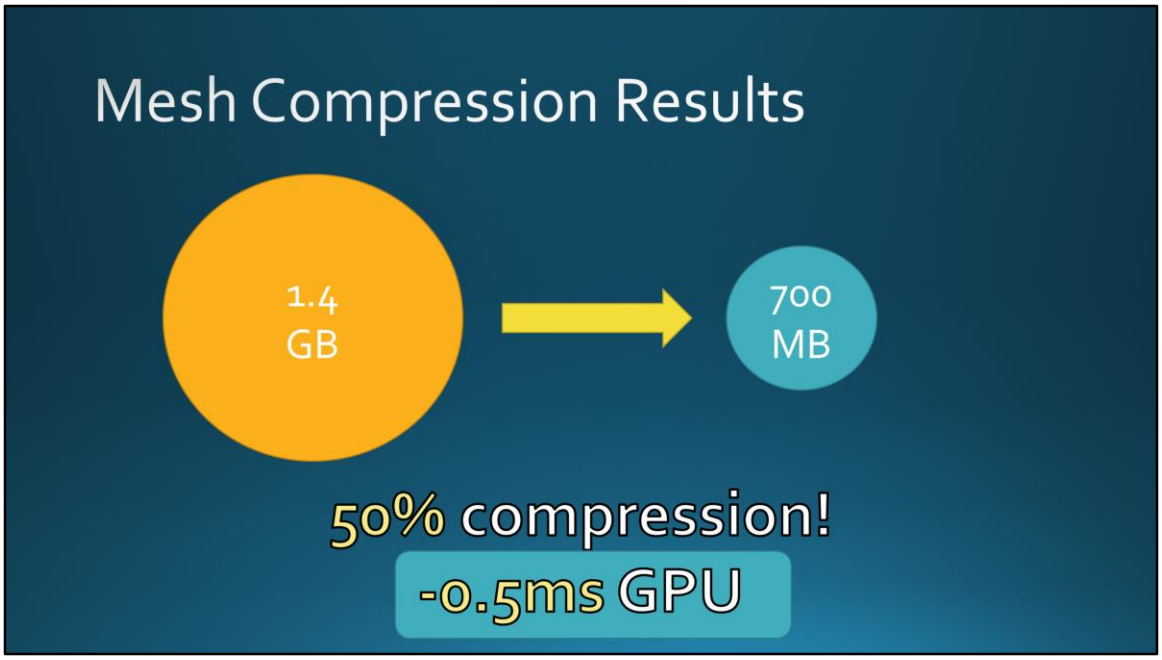
```
// Decompress normal
uint3 pos_normal = asint(v.vertex.xyz);
v.vertex = float4(f16tof32(pos_normal), 1);
v.normal = f16tof32(pos_normal>>16);

uint2 uv1_uv2 = asint(v.texcoord.xy);

// Decompress uv2
#ifndef LIGHTMAP_OFF
v.texcoord1.xy = f16tof32(uv1_uv2>>16);
#endif

// Decompress uv1
v.texcoord.xy = f16tof32(uv1_uv2);
```

Then, to decompress:

We first fetch the fake "input position" as an Integer value. Then we unpack 3 floats at once using the f16tof32 instruction. It takes three 32 bits integers, grab the low 16bits of each, interpret them as f16, and return a float3 vector.

Then, we shift the fake "input position" 16 bits to the right to fill the low 16bits of each component with the normal portion. We decompress the same way the 3 normal floats.

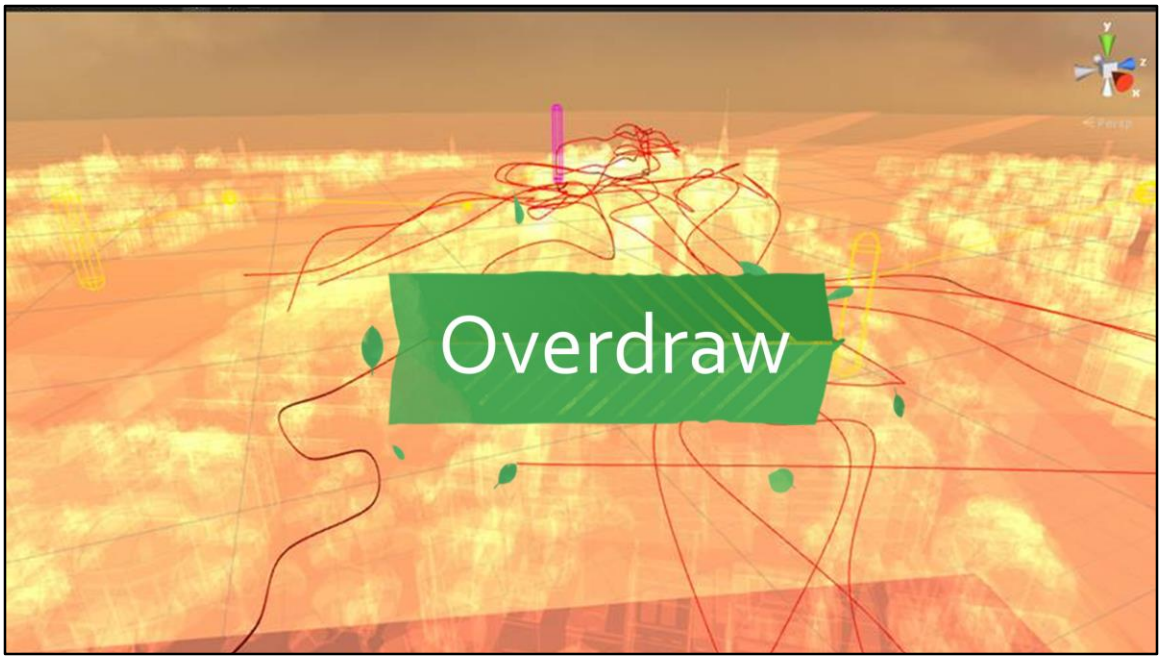
The same process is performed to decompress the UV1 & UV2 channels out of the "fake UV" channel, passed in the shader.



50% hardware compression of the vertex format
Reduces mesh footprint from 1.4GB down to 700MB

(CLICK)
Decompression cost on the GPU on the PS4 offset by lower memory bandwidth usage => actually improved overall performance by up to 0.5ms in worst case!

We fit in memory again, hurray!
BUT…



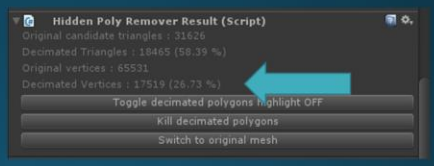
At that point, we still had too many meshes to process on the GPU to be able to run at a solid 60fps. The way we constructed the city with buildings LEGO type buildings pieces meant that, a lot of internal polygons were actually never seen. This is the case of bushes and tree leaves partially included in buildings or grounds, as well as facades and small details. It's particularly true for chunks LODs since they're viewed at a distance. Unfortunately, retouching all the assets manually was not possible because they changed frequently during production, and trying to do it after the data freeze would end up being a very time consuming job.


Persp
LOD 0

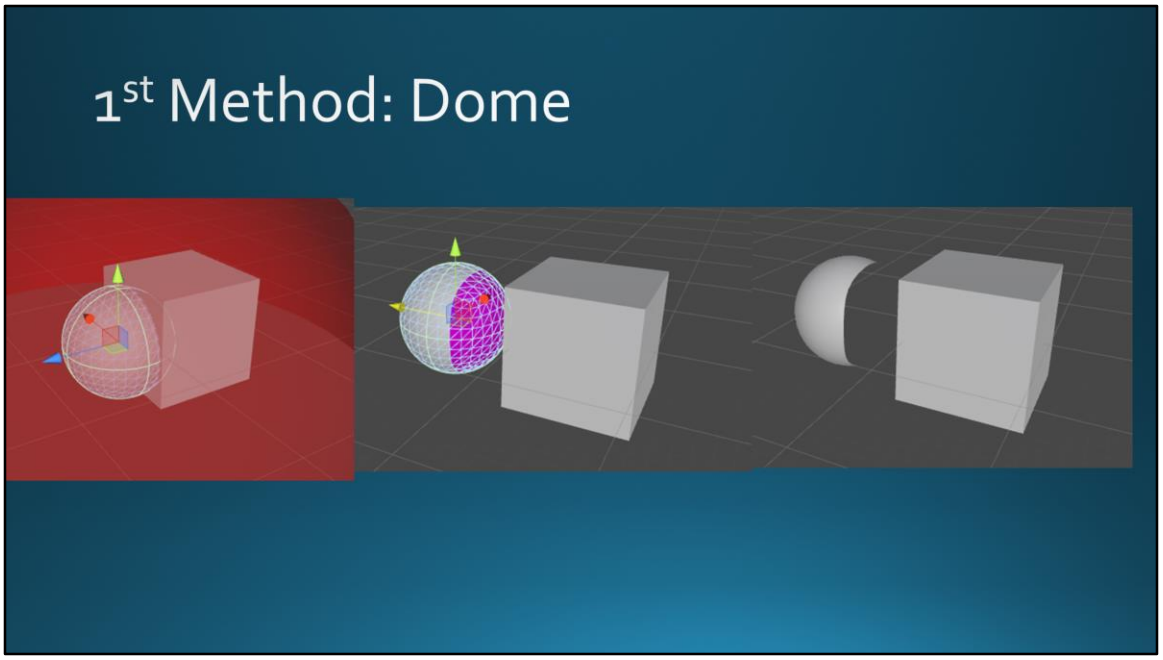
A lot of tree was hidden inside that building!

Polygon Decimation System

- Built a hidden polygon removal tool to remove these in our pipeline
- Runs distributed on the GPU



Combine with previous slide



Separate the city into clusters (like LOD grid)
Fit a dome around each cluster
Take a “snapshot” of the cluster from points sampled from the dome
Keep track of how many pixels were seen from each polygon
Drop polygons with a pixel count lower than a threshold value

Magenta highlight shows where polygons will be cut

Dome - Limitations

A lot of the fun of Eagle Flight happens when you fly around and through buildings, and that means you will be able to see polys inside tunnels that aren't visible from the exterior domes. So unfortunately this method will cull a lot of polys that the player will see if you run it on our LOD0.

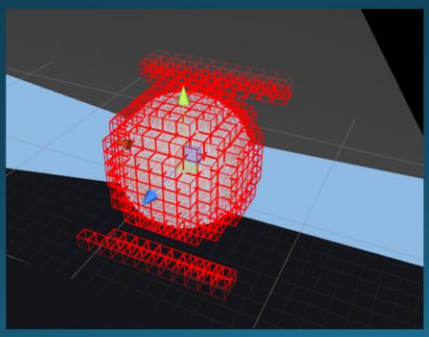
Dome method - results

- Works great far away:
  - **25-40%** polygon reduction
  - **13-15%** vertex reduction
- But highest polygon density is in our LODo
  - Flying inside the clusters revealed a lot of missing polygons

This worked great for LOD1, 2 and 3, since they are always rendered when the player is far away from the clusters. We noticed a ~25% to ~40% of polygons dropped, and around 13-15% of vertices dropped. Unfortunately, the highest polygon density are in our LOD0 clusters, and this approach doesn't work as the player can fly inside the clusters, and would notice a lot of missing polygons.

2nd Method: SDF-based

- Based on Signed Distance Fields (SDF)



Split the LOD grid chunk into 3D bounds into small cells 4m cubed
For each cell,

Render a cube map from the center of the cell – like firing rays in all directions to find closest polygons

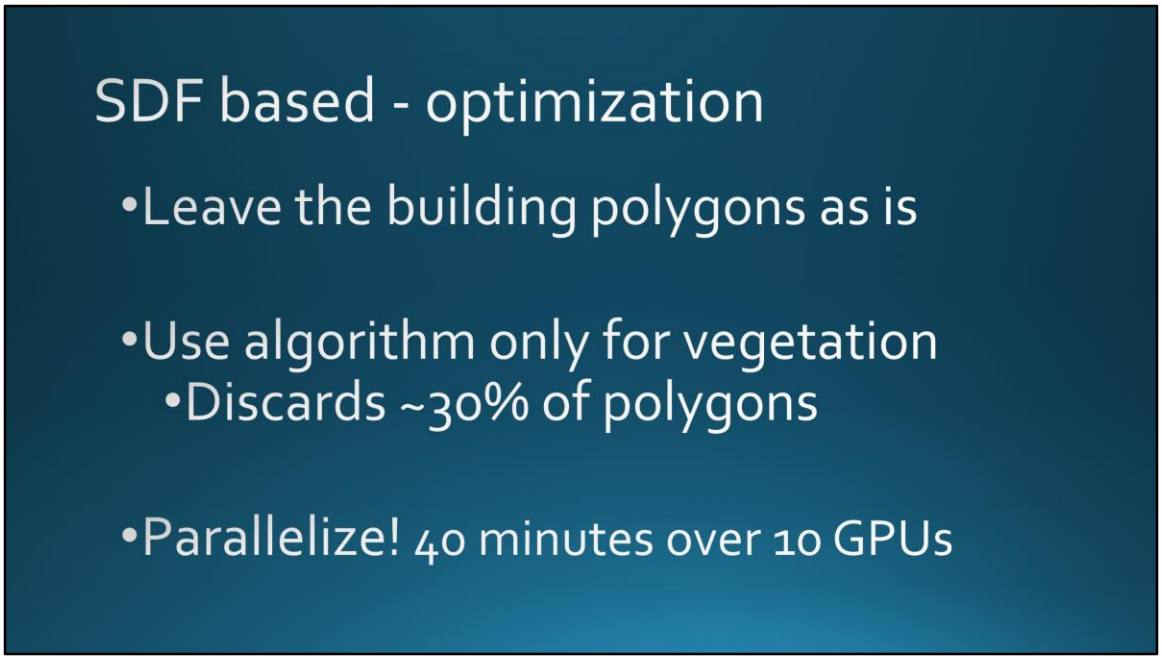
Depending on ratio of backface hits and frontface hits, categorize cell as "interior" or "exterior"

Interior cells give small negative score to front facing polygons
Exterior cells give big positive score (to be conservative)
Discard all cells with negative scores

SDF based - results

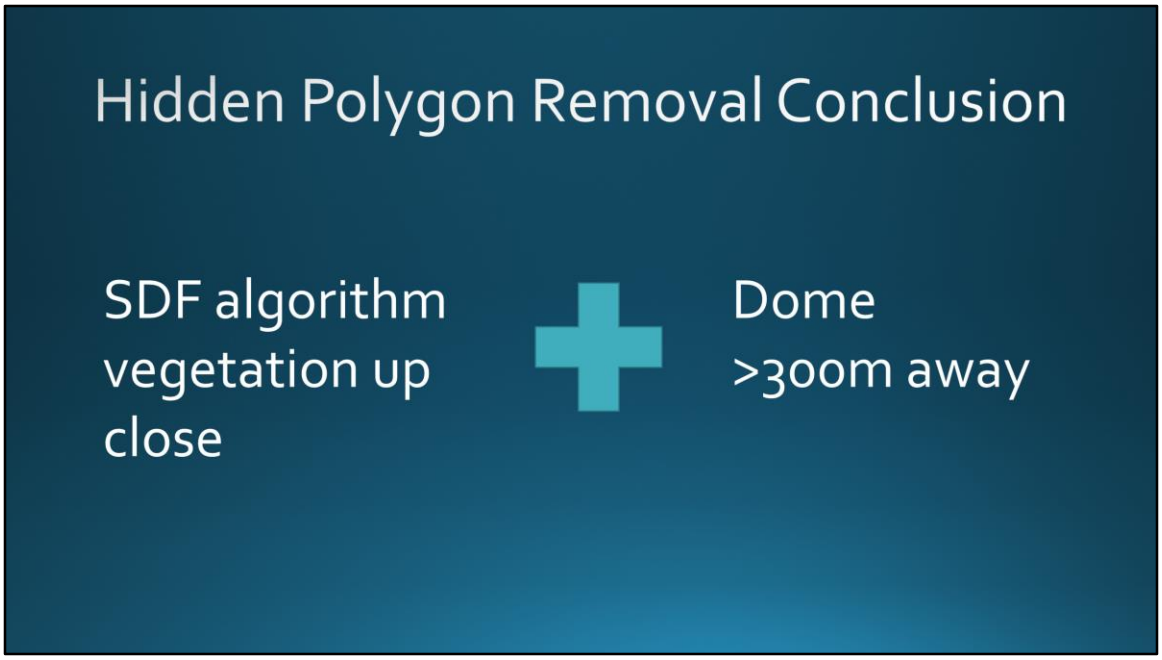
- Still killing important polygons on buildings
  - How they're modelled, holes in other buildings
  - Small sizes could throw off algorithm precision
- Worked wonderfully on vegetation
- BUT…still took ~12 hours on whole city

Although working nicely on portions of the scene, this algorithm was still killing important polygons on buildings, because of the way they were modelled, or holes in nearby buildings, or just because of their small size throwing off the algorithm precision. It was doing a great job on vegetation though. Processing the full city was taking ~12 hours, which was a drastic limiting factor.

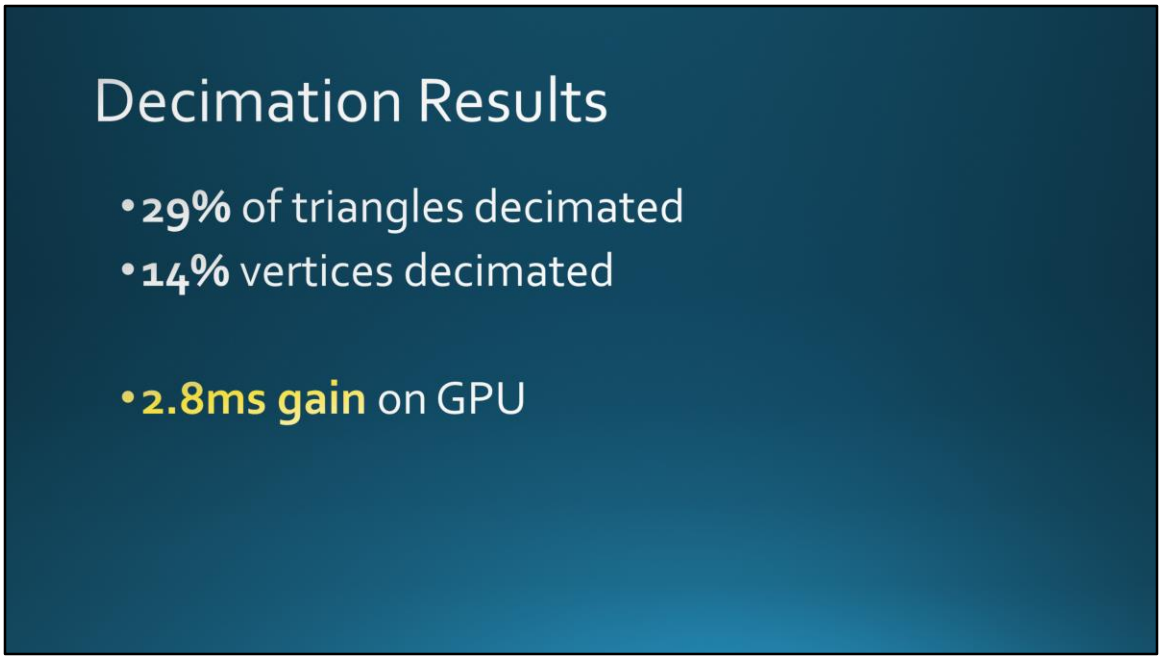


The algorithm was great at discarding vegetation due to the way the trees were authored with a ton of overlapping leaves. It was able to discard 30% of the vegetation polygons. We left the buildings out of the decimation system because we were still seeing too many artifacts. We developed a distributed version of the decimation algorithm and optimized the code to reduce computation time down to 40 minutes over 10 computers.

This was short enough that it could be run nightly or over lunch in a pinch.



In practice, we use the SDF algorithm for vegetation up close, and the Dome algorithm to simplify further things where the player can't reach (LOD 1 & 2, which are always displayed 300m at least from the camera).



1.3 ms gain on GPU

Ensuring correct ordering of polygons for cache friendliness after decimation brought another 1.5 ms

For a total of 2.8ms

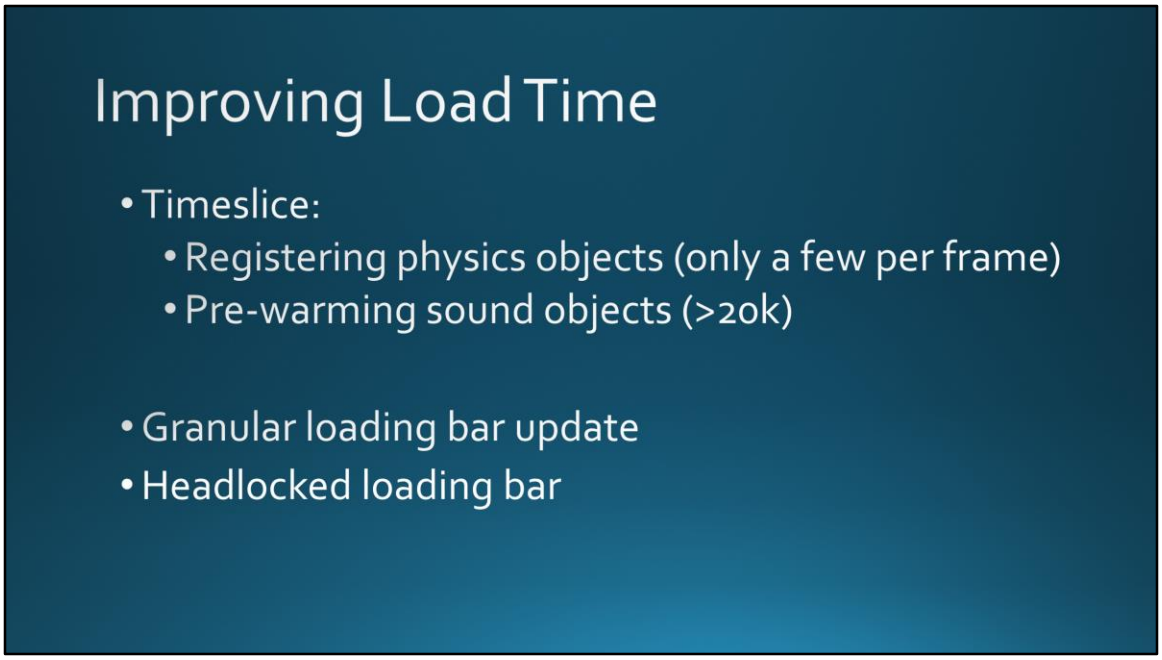
Not bad when total frame is 11 or 16 ms!

Mo' buildings, mo' problems

- Artists and level designers kept adding more and more geometry
- Load times busted TRC limits
- Worse, we had ~20 sec freeze at end of loading screen

Get rid of almost all Awake()
Pre-initialize and serialize everything
We call that data COOKING to distinguish it from mesh BAKING

150K gameObjects in city, 30K with MonoBehaviours



Some spikes in loading you just can't get rid of due to Unity's initialization process
Initial loading screen is head-locked to avoid "hall of mirrors" effect when re-projection fails

At game building time, we extract the meshes from the scene and generate some meta data in the scene that help us streaming back all the assets in the correct gameobjects at load time. Using this approach we can time slice, and update a loading bar in a more granular way.

We noticed that in our case registering the Physics objects in the Physics system was quite time consuming. We do only a few per frame. We timesliced other game systems loading code, like pre-warming the sounds objects, of which we have many.


Runtime Optimizations

Inspired by Alex Vlachos' excellent GDC talk, we wondered: what else could we stencil out?

Stencil out parts of RT occluded by bird beak, eyebrows, and tunnel vision blinders
Extra cost of doing stencil pass was offset by gains we got in critical scenarios, particularly lots of overlapping foliage
Up to 1 ms savings on the GPU
Minor artefact: beak is on post-reprojection layer, city is underneath it => if you shook your head fast enough you could get "undrawn" city peeking out from edges of head locked elements
    Minimized this by adding a buffer area

Avoiding Spikes - Garbage Collection

- Pool smaller objects
- Debug.Log is your enemy
- Watch out for 3rd party libraries eg. Photon, Wwise
- Put garbage collects where you can hide them

Garbage collection will probably cause your biggest spikes – pool smaller objects

Debug.Log is your enemy - use sparingly for debugging only, otherwise set up a logging channel system so you can turn on/off relevant debugging info at will

3rd party libraries like Photon Wwise - often allocate unnecessarily, esp with plugins that are straight ports from C++.

For us we called GC.Collect() in every black screen (~5 min) – fixed cost but spikes are hidden where user can't see them

3 easy ways to improve performance

- Avoid:
  - foreach
  - ToArray()
  - String building

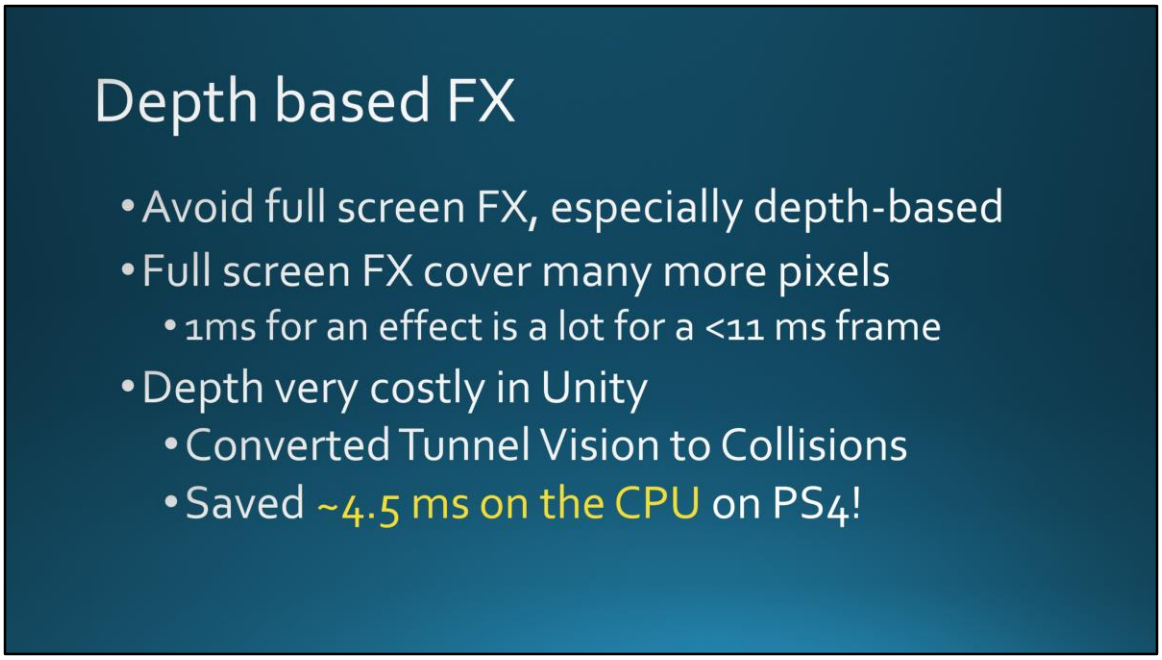
Each of these allocates memory

Canvas optimizations (Unity specific)

Moving canvases or updating them causes a lot of updates throughout the hierarchy – causing massive perf issues

We had an activity starter for every activity in the world, and pressing a button could cause all of these billboards to pop up.
The more canvases you have, the longer the canvas updates take

We ended up doing the billboarding on the GPU to avoid the canvas updates completely.



They also require special consideration to make sure they work properly in VR

We converted our tunnel vision effect to use collisions instead of depth

Part of this is that we were no longer detecting vection per-pixel

Other optimizations

- Sound grid manager
- 2D approximation of volumetric sound
- Simplified sphere-based physics collision system
- Pooling system to pre-spawn and enable/disable game objects

Here are some of the other optimizations we did that we don't have time to cover today.

VR profiling is challenging!

- Perf spikes can kill profiler
- Pair profiling is best

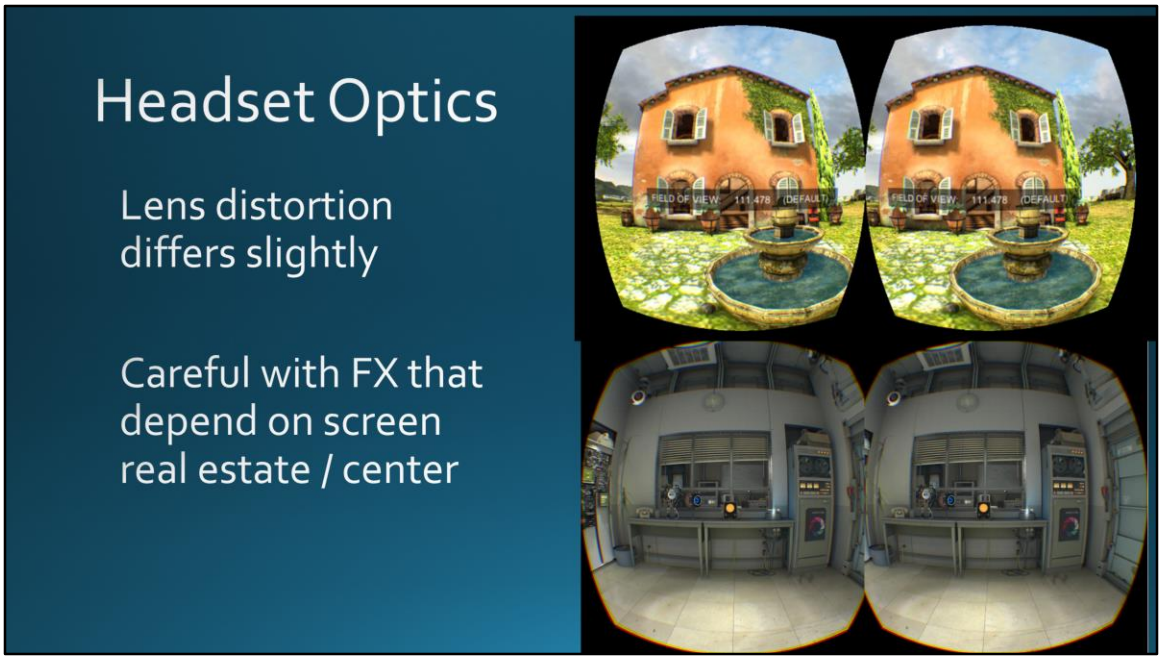
We found it easiest to profile in pairs, with one person playing the game and the other observing the profiler and stopping if necessary. This gives you a better idea of what the profiler looks like under normal conditions and it's easier to recognize abnormal behavior. It also means you don't have to flip in/out of the headset quickly to pause the profiler.

Let’s talk about a few additional tips for anyone considering multiplatform VR development

I won’t talk too much about known headset difference, but tips I haven’t heard elsewhere yet.

Black levels per headset

- Unlit billboard image in simple test app
  - Careful w/ compression settings!
- Adjust contrast/lighting intensities to match



Top is Oculus, Bottom is Vive

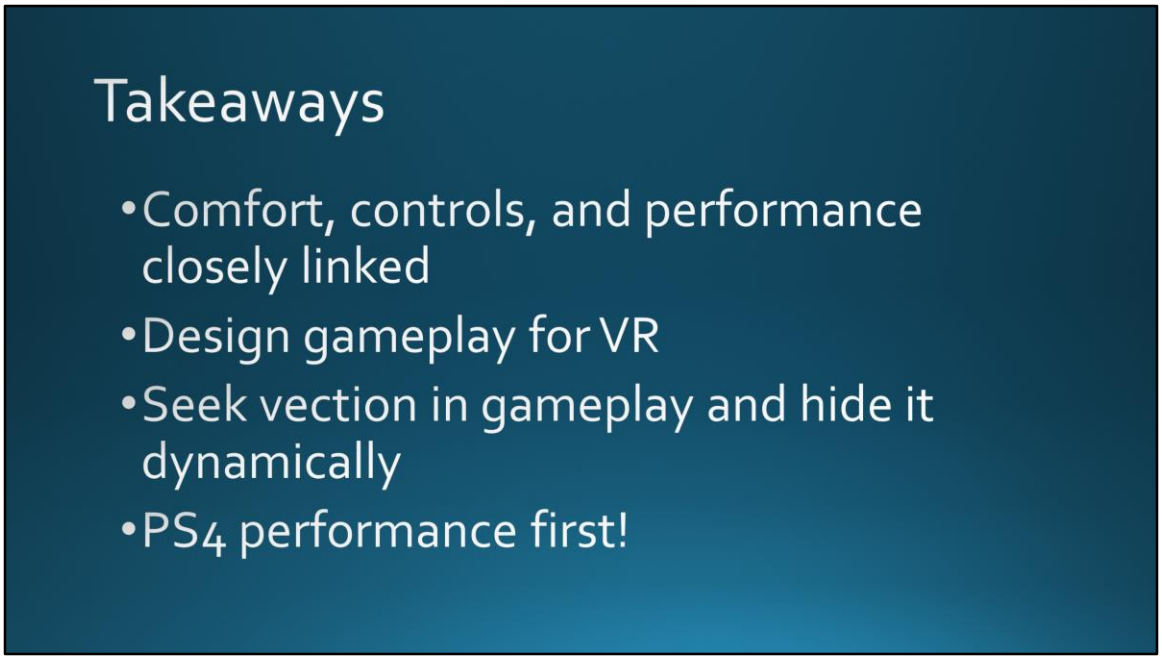
Specifically:
Vive has a larger vertical FOV
“Middle” is not in the same place in the Vive compared to Oculus – initially we had an issue where you could see the black blinder coming in in the middle – quite uncomfortable

Other headset differences

- Headset weight
- Safe zone for text differs. Check your UI!

Add user-adjustable functions for sensitivity if the user is going to be moving their head around a lot

Check your UI on all platforms



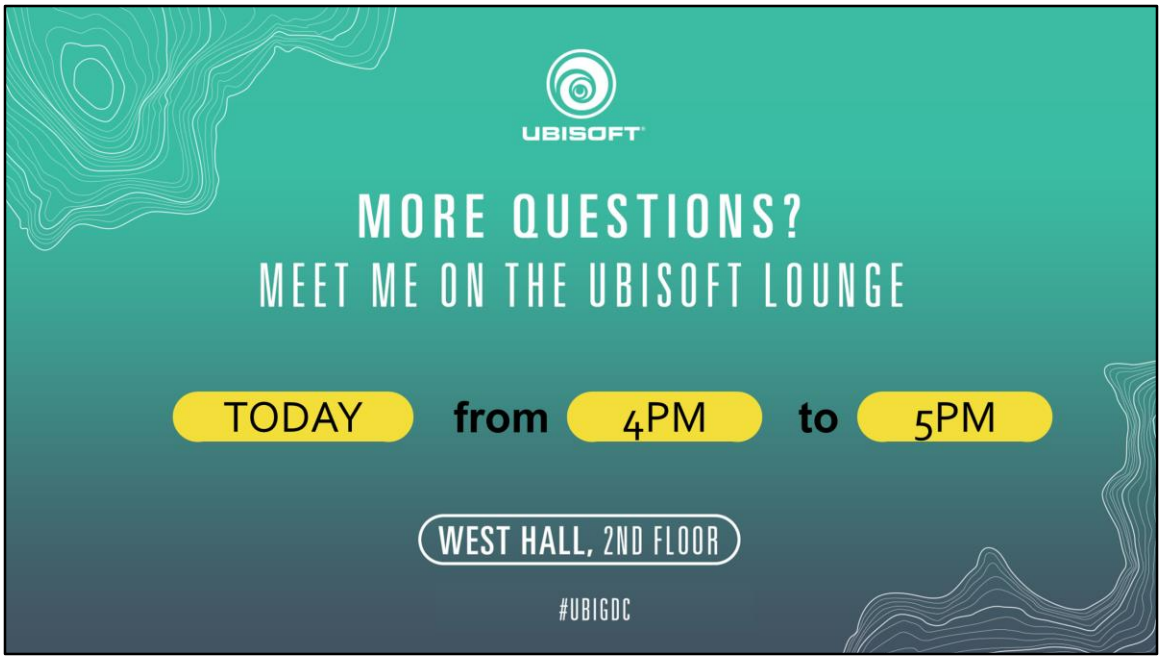
I'll conclude my remarks by reminding you of something I said earlier: comfort, controls, and performance are closely linked. If one changes, you'll probably need to adjust the others.

Think about PS4 performance right from the beginning and KEEP good performance the whole way through

There's no ONE solution for performance or comfort. You'll probably have to do a lot of little things to pull things together.

I want to thank the entire Eagle Flight team, but in particular the following whose work is covered in this presentation


Thank You!
Questions?
Vicki Ferguson
Lead Programmer, Ubisoft
vicki.ferguson@ubisoft.com
@vicki_ferg


UBISOFT
MORE QUESTIONS?
MEET ME ON THE UBISOFT LOUNGE
TODAY
from
4PM
to
5PM
WEST HALL, 2ND FLOOR
#UBIGDC