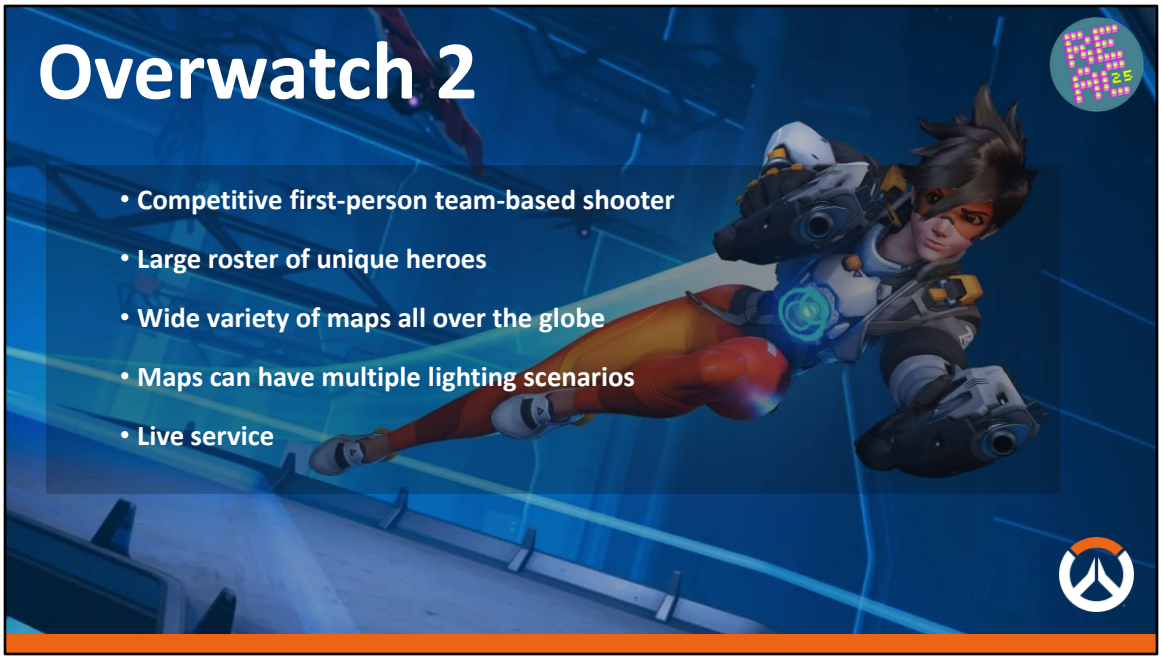
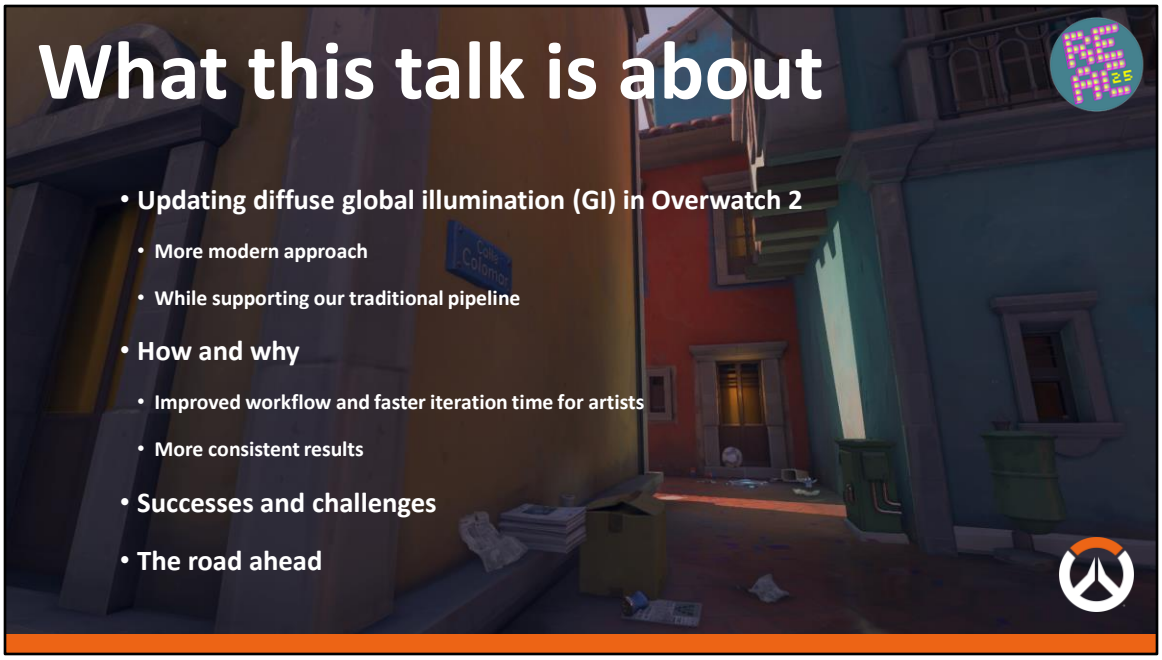
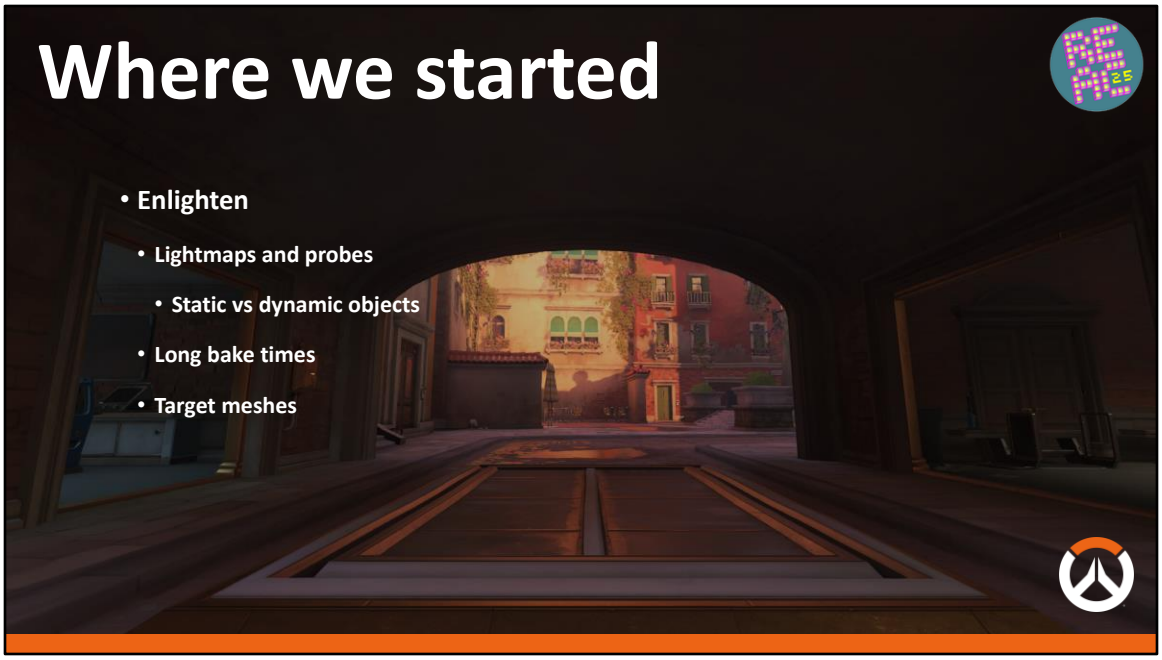
Evolving Global Illumination in Overwatch 2



Overwatch 2 is a free-to-play first-person team-based competitive shooter featuring a large roster of unique heroes, each with their own unique abilities.
We feature a broad range of maps taking place all over the globe.
Many of our maps feature multiple lighting scenarios, for example, a day and nighttime version.
Time of day is set at match start, does not change during a match.

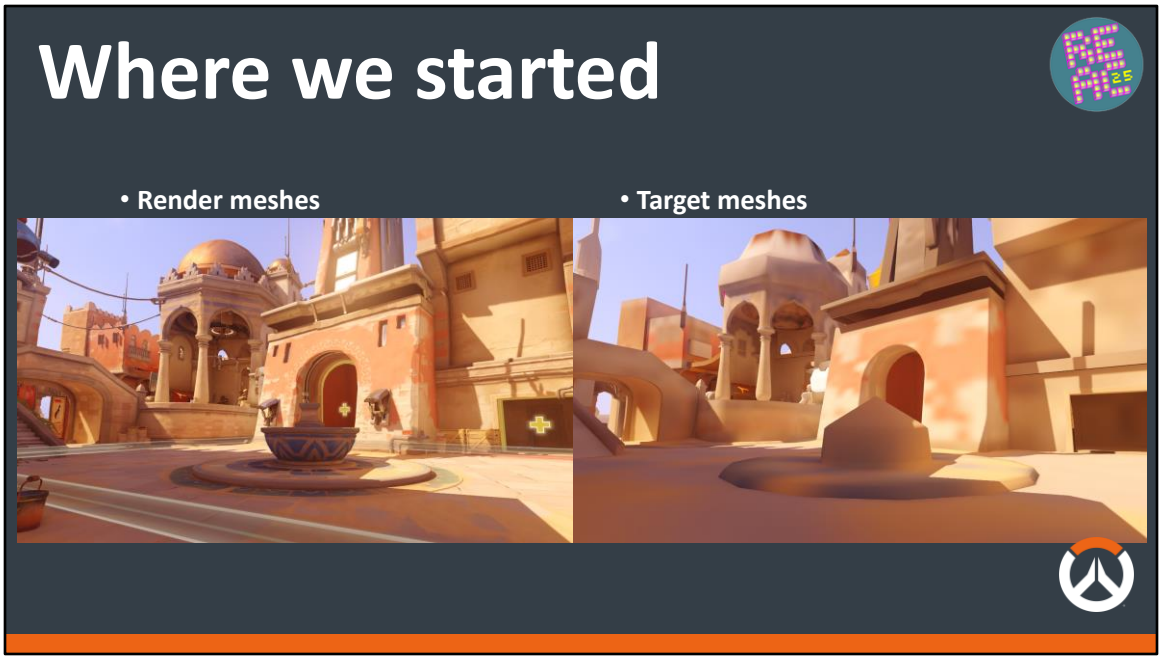


We're going to talk about updating the diffuse global illumination solution in Overwatch 2 to a more modern approach, while maintaining the previous solution in support for existing maps.
Go over the alternatives we evaluated, how we prototyped ideas, gathered meaningful data to help de-risk our major concerns, and finally made it to production.
The hows and whys, as well as the successes and challenges we've have faced along the way.
Why: improve workflow and iteration time for artists
Why: more consistent results
We'll also talk about the road ahead.
You're seeing this in the middle of the journey.
Currently shipping in several maps, but we still have work to do before this is feature complete.



Our current GI solution is using Enlighten.
Enlighten is a middleware for indirect illumination.
This provides a mix between baked lightmaps for static geometry and baked probes for dynamic objects.
Some of the issues we're hoping to address are long bake times and our artists having to create target meshes.

Where we started



On the left here you have a standard render of one of our maps.
On the right you have the target mesh debug view.
Hand-made by artists in Maya.
Additional time and effort burden.
Why use target meshes?

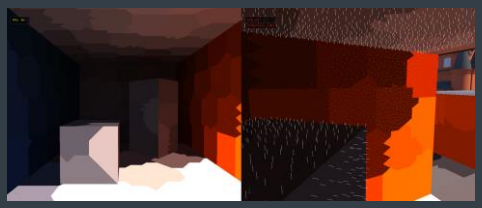
1.Results in fewer lightmap UV islands (aka "charts") than auto-unwrapping solutions. This means fewer seams in the lighting and less memory needed for lightmaps.

2.Gives the artist control over where lightmap seams are placed.

3.Fewer polys sent to the baker = faster turnaround times for bakes.

Where we started

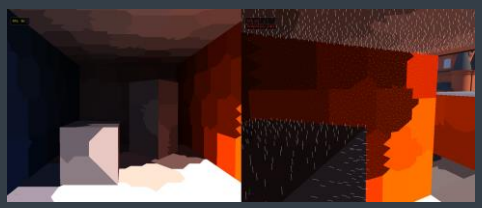
- Enlighten
  - Scene decomposition
    - Divide static scene geometry into small surface patches called “clusters”
  - Evaluate direct light at each cluster
    - Multiple sample locations per cluster called “dusters”



More info on how Enlighten works since we’re currently (temporarily) using some of its data outputs in our new solution.
Enlighten works by decomposing the static geometry of the scene into small surface patches referred to as "clusters".
The target meshes are what we use as the input for this step.
Each cluster contains random points along its surface called dusters that are used to compute direct lighting for the cluster.

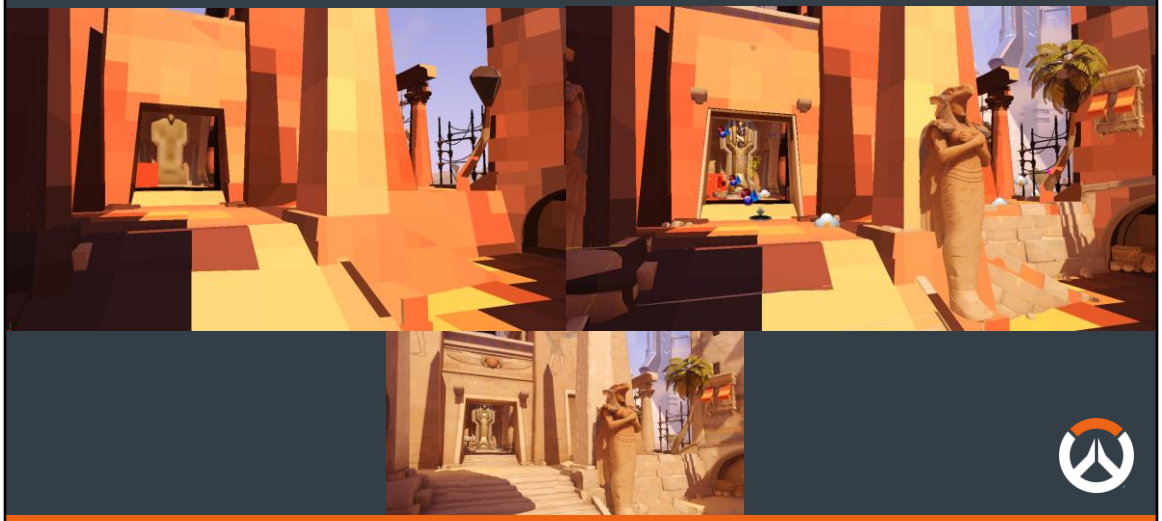
Where we started

- Enlighten
  - Calculate visibility between clusters
    - Determine proportion of energy that can travel between clusters
  - Precompute optimized runtime format using clusters and visibility
    - Must be re-computed any time static geometry is updated
  - Lengthy precompute vs fast runtime evaluation



Enlighten then calculates visibility between each cluster and determines the proportion of energy that can travel directly from one cluster to another.
This data is finally turned into an optimized runtime format.
This is a lengthy process that must be re-done anytime the static geometry is updated.
The tradeoff is that it's fast to evaluate at runtime.

Enlighten Clusters



Debug view showing Enlighten clusters
On the left, the clusters are shown by themselves.
On the right, the clusters are shown combined with the render mesh geometry.
Notice that many objects are either greatly simplified, such as the tree and the entrance path, while other objects are not represented at all, such as the statue.
Bottom image: normal render view

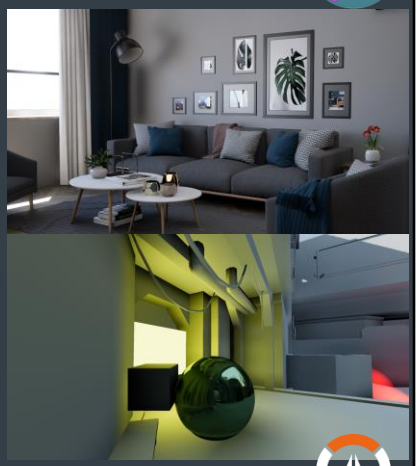
Evaluating alternatives

- Improving the target mesh solution
  - Can we remove this burden from artist workflow?
  - Worked with AI group to see if target meshes could be generated from render meshes
    - Not enough training data at the time to get usable results

First, we looked into seeing if we could automate the target mesh creation pipeline.
Another member of our team worked with the AI group to see if target meshes could be generated from our render meshes.
There was not enough training data at the time to get usable results.

Evaluating alternatives

- SDF-based solutions
  - Lumen (Unreal Engine 5), SDFGI (Godot 4)
    - Complex for time constraints and development resources
      - One engineer
    - Limited window for research



We also investigated signed distance field-based solutions, such as Lumen in Unreal Engine 5 and SDFGI in Godot 4.
We only had one engineer working on this effort and had a limited time to do it. With that limited amount of time, we also didn't have an extended window for research.
Top image source: https://dev.epicgames.com/documentation/en-us/unreal-engine/lumen-global-illumination-and-reflections-in-unreal-engine
Bottom image source: https://docs.godotengine.org/en/stable/tutorials/3d/global_illumination/using_sdfgi.html

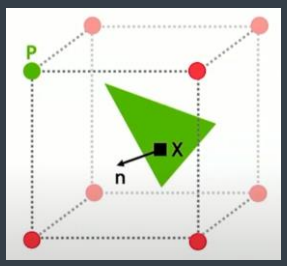
Evaluating alternatives

- Probe-based solutions
  - RTXGI (DDGI)
    - Platform support
    - Hardware ray tracing
    - Compute based fallback

We looked at DDGI as packaged with RTXGI, but it was quickly clear to us that platform support would be an issue.
Our low-end platforms have no hardware ray tracing capabilities and aren't powerful enough to do a compute shader-based implementation.
Even though we couldn't use the out-of-the-box implementation, we did see a lot of promise here and that ultimately led us to our solution.
Image source: https://developer.nvidia.com/blog/rtx-global-illumination-part-i/

Where we are

- **Dynamic Diffuse Global Illumination (DDGI)**
  - **3D grid of regularly spaced probes**
  - **Capture irradiance and visibility data at each probe location**
  - **Interpolate between 8 nearest probes**
  - **https://jcgt.org/published/0008/02/01/**



X: surface position
n: surface normal
P: base probe location

The baseline solution we landed on is based on Dynamic Diffuse Global Illumination, or DDGI.
Uses a 3D grid of regularly spaced probes.
Capture irradiance and visibility data at each probe location.
Solve lighting by interpolating between the 8 nearest probes
See the references for a more thorough treatment of DDGI
https://jcgt.org/published/0008/02/01/
Image source: https://developer.nvidia.com/blog/an-engineers-guide-to-integrating-ddgi/

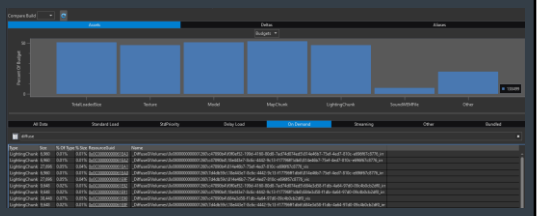
Where we are

- Artist-placed volumes
  - Artist-controlled density
    - Detail where needed (e.g., interiors, tight corridors)
    - Less detail in open spaces
  - Nested
    - Child volumes blend out towards parent volumes

Since our maps are static, we allow artists to manually place GI volumes and set probe density.
Artists are also able to orient the volumes in space.
This provides several levers for controlling where extra detail is needed, offering a tradeoff of memory vs detail.
We also allow artist to nest volumes inside larger volumes.
During our capture process, we blend the last ring of probes in a child volume out towards the nearest probes in the parent volume.

Where we are

- Artist-placed volumes
  - Each volume generates a pair of BC6 textures
    - Irradiance
    - Visibility
  - Keep an eye on overall texture memory usage
    - Existing and new reporting tools
    - Volumes can be removed based on graphics settings



For each volume, we generate a pair of BC6 compressed textures – one for irradiance and one for visibility, each using octahedral encoding to store probes flattened.
We experimented with BC5 for visibility, but BC6 has the same footprint of 16 bytes per block and allows for 16-bit precision floating-point values which simplified and avoided a lot of quantization logic.
Have not noticed any major quality reduction between BC6 and uncompressed.
It is very important to keep an eye on texture memory usage.
We have already built some reporting tools to aid with this and have more planned to make the footprint more obvious for artists.
We also have a basic level of detail system where we can omit volumes entirely based on graphics settings or platform

Where we are

- Leveraging Enlighten
  - We use the existing cluster data for solving irradiance
  - Use Embree for ray tracing irradiance and visibility
  - This won't be the solution forever
    - Got us up and running quickly
- You're seeing this in the middle of the journey!
  - Currently shipping in several maps

As a reminder, you're seeing this in the middle of the journey.
To get things up and running we decided to use data that was already available to us.
Leverage existing data that our Enlighten builds produce.
Specifically, we use the cluster data for solving irradiance.
We use Embree for CPU ray-tracing of irradiance and visibility against the Enlighten cluster meshes
This keeps us from having to evaluate our material and lighting equations on the CPU during our ray trace.
This of course will not be the solution forever, but it moved a significant body of work to later in the pipeline and got us tracing rays and capturing usable data extremely quickly.

Why this solution

- **Scalability**
  - **Extremely wide range of platforms and hardware specs**
  - **Predictable memory usage and runtime performance cost**
- **Artist friendly**
  - **Artists are used to volume-based workflows**
    - **Reflection probes, fog volumes, etc.**

Why this approach versus the other solutions we evaluated?
Tradeoff usually came down to scalability.
We have an extremely wide range of platforms and supported hardware specs.
We needed a solution that had a predictable memory footprint and just as importantly a predictable runtime performance cost.
And remember as mentioned earlier, we do have the ability to LOD-out specific volumes on low end platforms if we need the memory and they aren't adding much important detail.
This solution also provides a workflow that our artists are familiar with.
They are used to placing volumes for things like reflections and local fog.

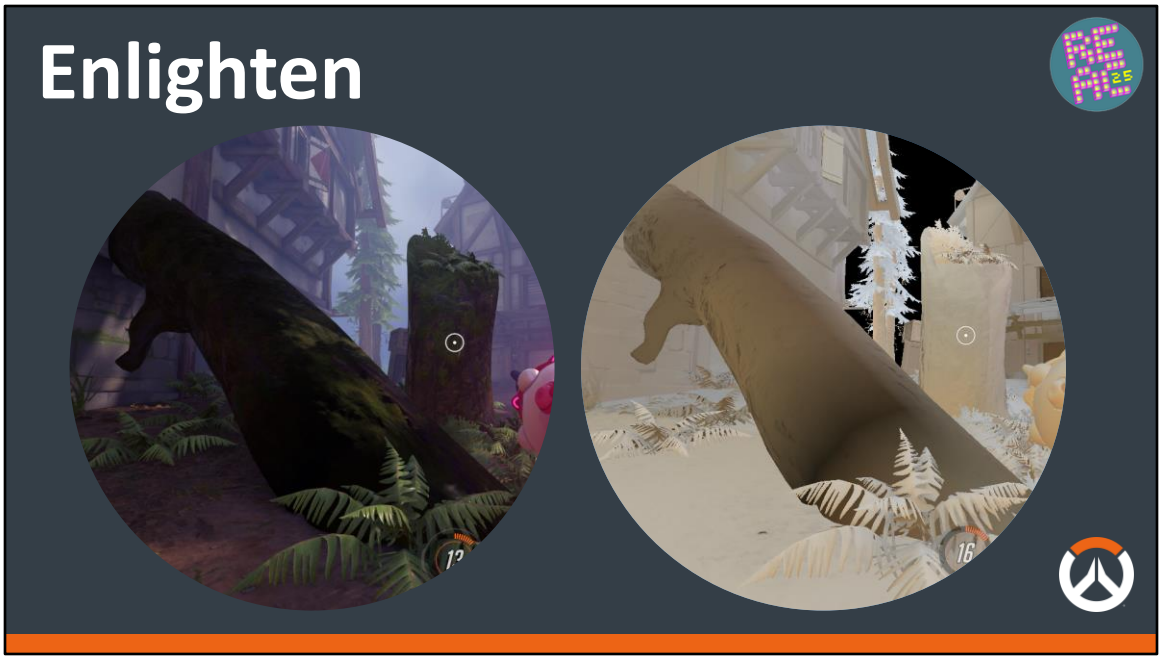
Why this solution

- Static lighting
  - Most of our lighting is static during a match
  - Prebaked GI solution fits this well
    - Prebake multiple lighting scenarios
    - Load just what we need at runtime
    - Just a texture lookup*

A baked solution is still favorable since our maps use mostly static lighting once a match has begun.
We can prebake our data for each of a map's lighting scenarios, then load just the data required for the active time of day at map load.
Don't need super complex evaluation for indirect lighting – just a lookup.
The asterisk here is because the DDGI algorithm is actually 16 texture samples, so a bit heavier than "just a lookup" implies.

Left: Enlighten
Right: DDGI
The barrels are dynamic objects.
They were getting bad lighting data from Enlighten probes.
With DDGI we can evaluate them per pixel and they look coherent with the scene.
Enlighten will bake a scene the same way every time – code change and more memory to use a higher density probe distribution
With DDGI – easy to adjust the volume position or density to improve coverage of a tricky area – rebake is fast (we'll talk about timing later)

Enlighten



This image demonstrates an asset that was giving our artists a lot of trouble with bad target mesh data.
There’s a noticeable lightmap seam near the bottom of the tree trunk.

DDGI

With DDGI everything is evaluated the same.
The lightmap seam in the trunk is gone.

Exploration

- Ideation and experimentation
  - Does it even work for us and our needs?
- Tackling low-end
  - Support for hardware that's over 10 years old
    - E.g., Tegra X1

We'll go a little bit more into the weeds of our process here and talk about ideation and determining if this solution would be a good fit for us.
Also talk about tackling low-end.
We support some hardware that's over 10 years old at the time of writing this presentation.
We knew we would have some serious work ahead of us to get this running on low-end platforms.

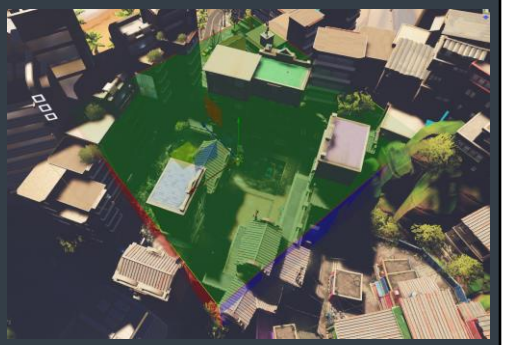
Ideation and experimentation

- Research
- Set limits
  - E.g., we can afford x memory and y frame time for a new technique
- Prototype
- Collect data
  - Memory
  - Performance

These were the main components of our experimentation loop.
We've touched a bit on research already and how and why we landed on a DDGI-based approach.
It's also useful to come up with a set of acceptable limits.
For example, we know our frame budgets are already tight on low end platforms so we must be very careful about any extra time we take up.
Be practical – it's a hard sell to say build a new thing and by the way it can't cost any time whatsoever.
Now we'll talk a bit about the prototyping process, then go a bit into early data collection to ensure we understood memory and performance costs.

Prototype

- Get something going as quickly as possible
  - Understand baseline performance and memory
  - One volume in one map
  - Can we capture what we need?
  - Can we apply that data?
    - And have it look correct?



The goal of our initial prototype was to get something up and running as quickly as possible.
For that we landed on:
One volume in one map
Just covers a small section of the map
Fixed probe density (32x8x32)
Answer the questions:
Can we capture what we need?
Once we have captured data, can we turn it around and apply it as lighting to our scene (and still have it look correct)?

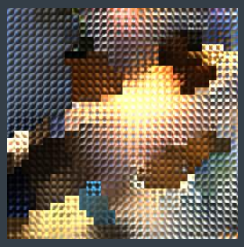
Prototype

- First working implementation
  - Single hard-coded volume
  - Small area of map
  - Stomps all previous light data

Our first implementation didn't even ray trace for irradiance.
We looked up the nearest 8 Enlighten probes and did a trilinear interpolation between them.
Again, we wanted something up quickly so we could prove whether we could use this or not.
The volume only encloses this little courtyard area and the shops.
32x8x32 probes.
We took lots of shortcuts to get something on screen, including stomping all existing lighting data.
Be careful when doing this – the tech debt collector WILL come knocking.
Good for our prototype purposes because it let us start collecting data.

Prototype

- Outputs of first successful DDGI captures



These are the output textures of our first successful DDGI capture.
Irradiance is captured at a resolution of 6x6 and visibility at a resolution of 14x14.
Each probe has a 1-pixel border for bilinear interpolation to work, so the footprint per probe is 8x8 for irradiance and 16x16 for visibility.
We did experiment with reduced resolution for visibility, but as others have seemingly found before us the quality starts to suffer too much.

Data collection

- Memory
  - Per probe (uncompressed)
    - 1024 bytes for visibility
    - 512 bytes for irradiance
  - 32x8x32 probes
    - 8 mb visibility + 4 mb irradiance = 12 mb total

Now that we have our data exporting to textures, we can start verifying assumptions. We were pretty sure we had the memory calculations correct because they're fairly straightforward, but it's always good to check those assumptions.
Uncompressed, each probe should cost 1.5 kilobytes, with 1 kilobyte for visibility and 512 bytes for irradiance.
For a 32x8x32 probe count volume, we had assumed uncompressed memory would be 8 megabytes for visibility and 4 megabytes for irradiance.

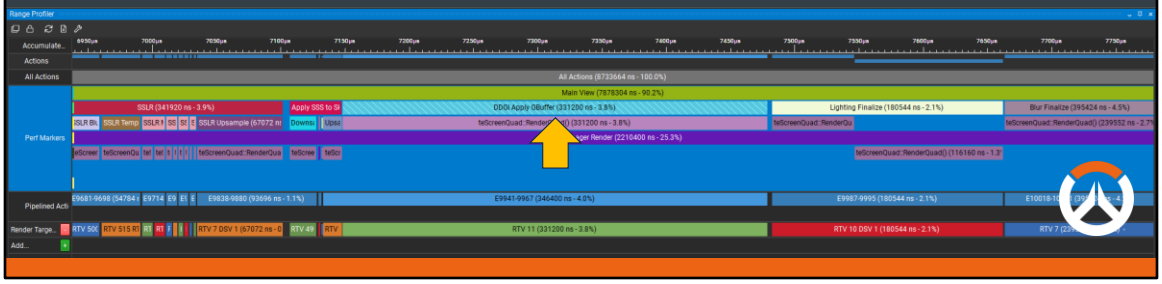
Data collection

- Memory
  - BC6 compression
    - 16 bytes per 4x4 block
    - Visibility: 256 bytes per probe
    - Irradiance: 64 bytes per probe
  - 32x8x32 probes
    - 2 mb visibility + 512 kb irradiance = 2.5 mb total

Once we bring BC6 compression into the mix, the numbers start to look a lot more palatable.
Since BC6 outputs 16 bytes per 4x4 block, we can simply look at how many blocks our probes take up and find our expected results.
Visibility probes: 16x16 -> 16 blocks -> 16 * 16 = 256 bytes per probe.
Irradiance probes: 8x8 -> 4 blocks -> 16 * 4 = 64 bytes per probe.
Our output textures with this test setup are expected to be 2.5 megabytes.
We were able to easily verify that from inspecting the resources on disk.
We feel good about memory utilization at this point as long as we're mindful when setting up volumes in maps.

Data collection

- **Performance – GTX 1080**
  - **~330 microseconds to evaluate at full resolution**
  - **~115 microseconds to evaluate at half resolution**
  - **We get a little time back from our frame since this is a replacement technique**



Now we'll talk a bit about our first performance measurements.
This is an Nsight capture on a GTX 1080.
The DDGI apply pass is taking about 330 microseconds at full resolution.
Not awful. We knew going in that we would likely need to do some massaging for performance numbers, especially if we were going to support multiple volumes of varying densities.

We had also figured early on that for this to run on lower end hardware we would probably need to support a half-resolution version of the effect.
This is the first half-res capture, taking about 116 microseconds on the same GTX 1080 card.

We felt pretty good about the starting point numbers we were getting in our initial tests when gathering data from a discrete desktop GPU.
We get some time back in our lighting pipeline since this is a replacement technique for our previous GI solution.
Felt confident we could pretty much make up any difference.

Data collection

- Performance – Switch and low-end
  - We knew going in low-end hardware would be a challenge
  - Did not expect to see an almost 8 millisecond loss
    - 1/3 of our entire frame time for Switch
  - If there was any point where we doubted viability – this was it

Test Setup	Capture Point A Frame Time (ms)
Baseline	21.544652
DDGI enabled	29.346935

Our low-end platform numbers were not as promising.
The initial performance captures from the Switch were truly disheartening.
We were seeing almost 8 milliseconds of additional frame time, even when accounting for the removal of the existing GI solution.
Sure, we had planned on optimizations to get perf back, but to even get back to the same ballpark as before from that deficit seemed out of reach.
10 milliseconds is a third of our entire frame budget for Switch.
None of this is the Switch's fault, by the way – it has never claimed to be top of the line hardware. We chose to pursue this technique, so it's on us to figure out if it's suitable.
Our low-end PC spec and to an extent previous generation consoles also showed challenging numbers, but none were quite this drastic.

Data collection

- Performance – Switch and low-end
  - Half resolution (or lower) obviously going to be a requirement
  - Half resolution got us down to a deficit of 3 ms
  - Quarter resolution was of course even faster, but the quality started to suffer
    - Was still about a 1 ms deficit
  - What else can we do?

Fortunately, we're stubborn graphics programmers who don't know when to quit.
The first and most obvious thing to do was of course evaluate the data at a lower resolution.
Moving down to half resolution and fully replacing our previous GI code got us down to a deficit of only about 3 milliseconds
We evaluated quarter resolution, as well, but the quality degraded too much to be acceptable.
We need to find ways to claw back more time.
The image shows the location all the performance tests were gathered from.

Performance

- Low-end specific optimizations
  - RGBA16 -> RG11B10 render target format
  - Separate out visibility and weighting loop from irradiance evaluation loops
  - 16-bit data types in shader
    - Limited gains in DDGI shader – mostly texture fetch bound
    - We were able to apply this to other shaders, also
    - Not a silver bullet

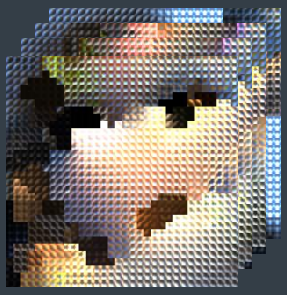
```
// pseudo-code
for (uint c = 0; c < 8; ++c)
{
    sampleVisibility();
    doVisibilityStuff();
    sampleIrradiance();
    doIrradianceStuff();
}
```

```
// pseudo-code
for (uint c = 0; c < 8; ++c)
{
    sampleVisibility();
    doVisibilityStuff();
}
for (uint c = 0; c < 8; ++c)
{
    sampleIrradiance();
    doIrradianceStuff();
}
```

We went through lots of optimization iterations trying to find ways to get time back.
This is a non-exhaustive list of some of our larger wins.
These are listed as “low-end specific” mostly because that was the driving force behind delving into this work.
They of course didn’t make mid- and high-end any worse, but sometimes the impacts were negligible.
Moving from an RGBA16 to an RG11B10 texture format for the evaluation render target was an easy update that helped shave some time off.
A surprising Switch-specific boost we found was that by separating the visibility and irradiance data gathering into separate loops there was a fairly substantial gain.
The texture cache access for Switch specifically seems to be more in tune with this setup.
Other platforms performed about the same regardless of how the loops were setup, so we kept it for all platforms.
We also enabled 16-bit types in our Switch shaders.
We updated this and a handful of other shaders, like a few post-processing effects, to use fp16.
Limited gains in the DDGI shader itself since it is texture fetch bound.
Resulted in a few hundred microseconds back when applied judiciously through

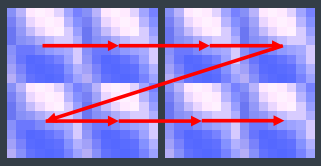
pipeline.
Should be noted this is not a silver bullet – as we've seen for example with our DDGI shader being texture bound, there's not always a meaningful impact to enabling 16-bit types.

Performance



- Low-end specific optimizations
  - Reference implementation loop

```
[unroll]
for (uint i = 0u; i < 8u; ++i)
{
    const uint3 gridOffset = uint3(i, i >> 1u, i >> 2u) & uint3(1u, 1u, 1u);
```

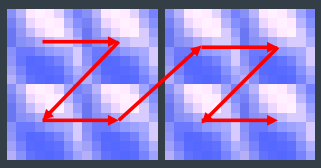


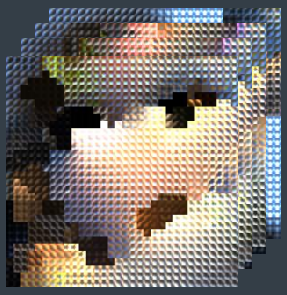
Another texture access optimization that helped get us time back on Switch was to make a slight update to the order probes were read.
The reference implementation processes probe contributions in X, Y, Z order.
This makes total sense, especially from a readability perspective.

Performance

- Low-end specific optimizations
  - Updated to index probes in slice first

```
[unroll]
for (uint i = 0u; i < 8u; ++i)
{
    const uint3 gridOffset = uint3(i, i >> 2u, i >> 1u) & uint3(1u, 1u, 1u);
```





On Switch specifically we found better texture cache hits by reordering the loop to read all probes per slice first.
Since each slice is a slab of probes stacked vertically, we first read neighbor probes on the XZ plane, then bump to the neighboring probes on Y.
For platforms with larger texture caches, this didn’t make much or any difference, but there was a measurable gain on Switch from swapping the access order here.

Data collection

- **Performance – Switch and low-end**
  - **With the previously mentioned optimizations we were back even with our baseline**
  - **For a single volume and a view with no early-outs**
  - **But we need to render several volumes**

With all of the previous optimizations, and likely a few others I've since forgotten, we were finally back on par with our baseline.
In our performance testing setup, we used a single volume and captured a view where the GBuffer was full so we could always understand our worst-case scenario (no early exits from shader).
We need to render several volumes, and it feels like we're going to have a nearly linear cost associated with rendering each additional instance.

Intermission

- Code/hack cleanup
  - The tech debt collector came calling
  - Full tools integration
  - Teach our map load and rendering pipeline about Diffuse GI Volumes
    - Maps consist of Placeables
    - Look for Diffuse GI Volume placeables at map load
    - Set flags so our lighting pass knows which path we're on
    - Maps use either lightmaps or use DDGI

First we needed to take some time to clean up a bunch of our old hacks in code.
Supporting a separate prototype-only path was becoming a pain, and the tech debt collector came calling.
We felt pretty good about the overall approach here and took time to fully integrate creating multiple volumes, nesting, capturing, and blending them into our tools pipeline at this point.
Move away from map-specific hard-coded values to full tools and engine integration.
Basic cleanup we did for our map load and rendering pipeline looks something like this.
Maps are built with things we call Placeables.
Static models, lights, reflection volumes, fog volumes, water, certain sounds and effects, etc.
Placeables contain basic type identifier information.
During map compile placeables of a type are all grouped together.
When we load a map, we search for Diffuse GI Volume placeables.
If found, we know to set flags during scene submission so that our lighting pass knows which technique to use.
All or nothing – maps either use the existing lightmap path or DDGI, but not mixed.

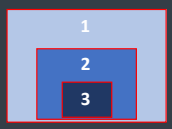
Prototype

- Back to prototyping!
- Hypothesize what we will see
  - Linear cost associated with each additional rendered volume
- Gather ideas on how to address it after verifying

So back to prototyping we went.
Made hypothesis on what we would see when we started putting multiple volumes in map.
Likely to be a near-linear cost associated with each drawn volume (depending on how large the volume is and how many pixels it actually touches – we early out for pixels outside the volume bounds).
It's fine to start thinking on how we would address this shortcoming at this point, but it's important to gather the data and verify assumptions first.

Data collection

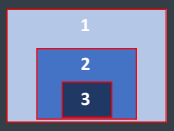
- Able to quickly verify our assumptions on cost per volume
  - Additional small volumes added little overhead
  - Large volumes added worst-case linear overhead
  - Lots of wasted work due to overdraw
  - Naïve (or outright silly) method of drawing volumes
    - Largest to smallest
    - Smaller volumes stomp previous contents
    - We can do better than this

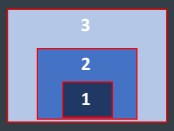


Once we added multiple volumes to a map, we were able to very quickly verify our assumptions on what the performance would look like.
We of course culled volumes that were outside the view frustum altogether but otherwise used a naïve (or outright silly) approach of drawing volumes largest to smallest.
Largest to smallest since child volumes must fit entirely inside parent, so smaller volumes should always be more detailed than the containing parent.
Child volumes just stomp any content written to the temp render target before it, lots of wasted work and overdraw.
Don’t judge too harshly – this was never the long-term plan – but it was very easy to get going for rapid prototyping.

Performance

- What we have
  - Volumes drawn largest to smallest
  - Each smaller volume stomping previous draw
- What we want
  - Volumes drawn smallest to largest
  - Each volume indicates which pixels it has already affected
  - Any pixel only ever gets GI evaluated once

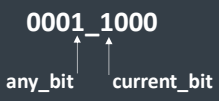




What we have is all our volumes being drawn largest to smallest and the previous contents being stomped.
What we want is our volumes being drawn in the reverse order and indicating which pixels they've touched so we only pay for each pixel once.

Performance

- **Stencil to the rescue**
  - Stencil bits: ‘current’ and ‘any’
  - Draw volume with minimal pixel shader
    - Test against ‘any’ stencil bit – reject if set
    - Discard if pixel is not contained inside volume
    - Set ‘current’ stencil bit for passing pixels
  - Draw volume with GI evaluation pixel shader and stencil
    - Test against ‘current’ stencil bit – pass if set
    - Clear ‘current’ stencil bit, set ‘any’ stencil bit
    - [earlydepthstencil]



We have a few stencil bits that are reserved for special purposes, and it happened that we had enough space to commandeer what we needed since they were not being used at this point in the frame.
We use two bits here, one for marking the currently drawing volume, and one for marking any pixel that has ever been drawn by a volume.
First draw the volume with a very simple pixel shader. Test against the ‘any’ bit – reject volume if set. If the pixel is outside the volume discard it. Otherwise set the ‘current’ bit.
Next draw the volume with the full evaluation pixel shader. Test against the ‘current’ bit. Enforce early depth-stencil test with [earlydepthstencil].

Performance

- Stencil to the rescue (cont'd)
  - We're now evaluating GI for each pixel only once
  - The stencil fill calls are typically in the single-digit microsecond range on low end
    - Couple hundred nanoseconds or less on higher end hardware
  - Now we see the draw time of a single volume spread across multiple volumes
    - But not exceeding that time
  - But we can do a little better still…

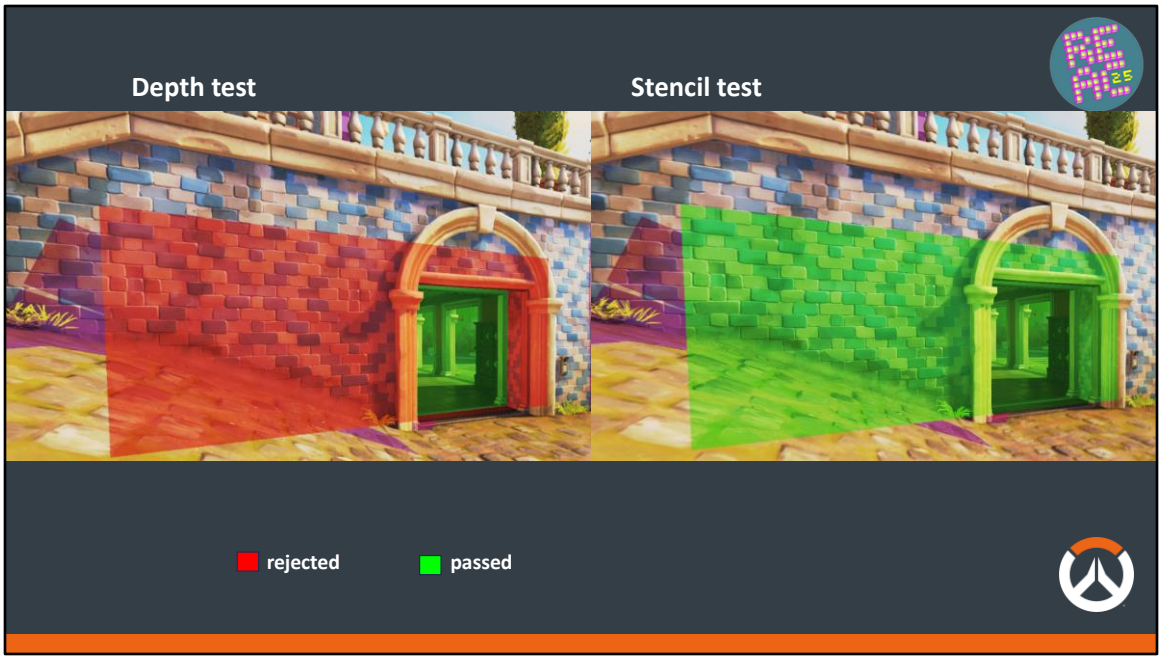
Now each pixel will only receive full GI evaluation once.
The stencil fill draws are very cheap – single digit microsecond range on low end, and even faster on higher end hardware (couple hundred nanoseconds).
Can observe the draw time of a single volume spread out across multiple volumes, but not exceeding the previous single volume time.
And there's actually still one more way we can get a little bit of time back.

Performance

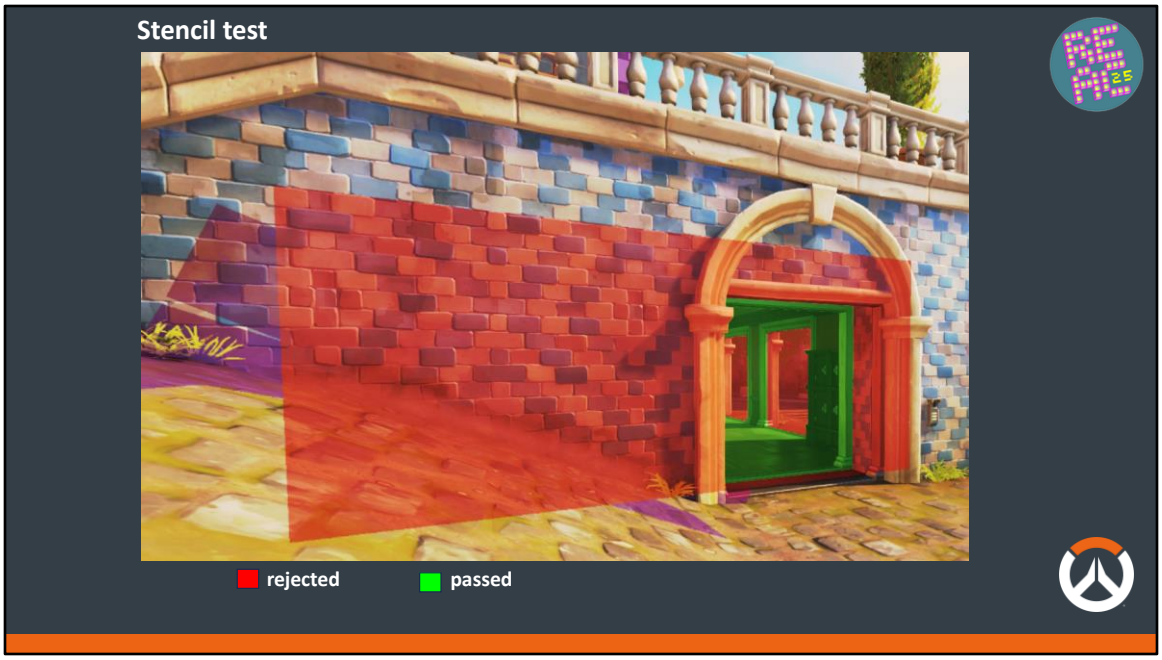
- Depth testing
  - Draw volumes differently based on if the camera is inside or outside
    - We use reverse-z depth
    - Outside: Depth compare less or equal
    - Inside: Depth compare greater
  - If inside, depth test will reject values not contained in the volume
    - Skip the stencil fill step
    - Reject if 'any' bit is set during drawing
    - Just write the 'any' stencil bit during evaluation

Even though it's very cheap to draw the stencil volumes, we don't want to waste work if we can help it.
We use different depth tests for the volume depending on if the camera is inside or outside while drawing.
When inside, the depth test will reject geometry outside the box for us.
Skip the stencil fill pass in this case.
Draw the volume with the evaluation shader.
Instead of looking for the 'current' bit, reject if the 'any' bit is set – just like we do in the stencil fill pass.
Continue to write the 'any' bit out for pixels that are not rejected.
If you're in an interior, you might end up filling your entire stencil buffer on the very first volume and every other volume just gets rejected downstream.

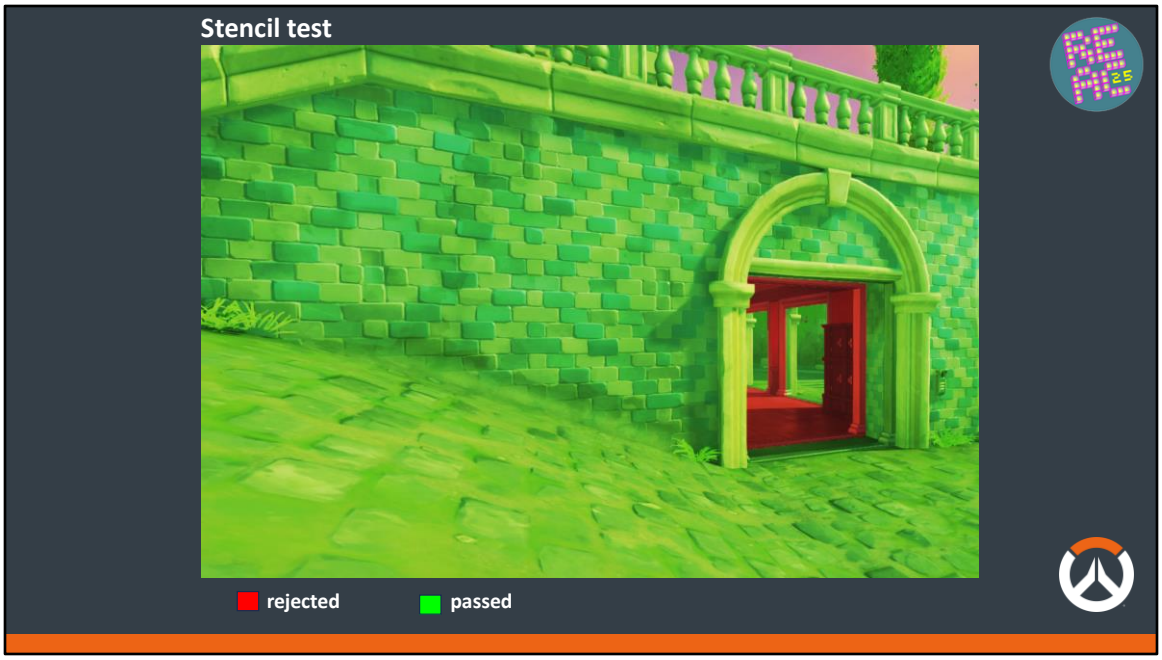
Let’s look an example of how this works.
Starting with this scene, there are three GI volumes.
There is a small volume inside the room in front of the camera, a second medium sized volume encompassing most of the playable space, then a massive low density volume encompassing the rest of the map.



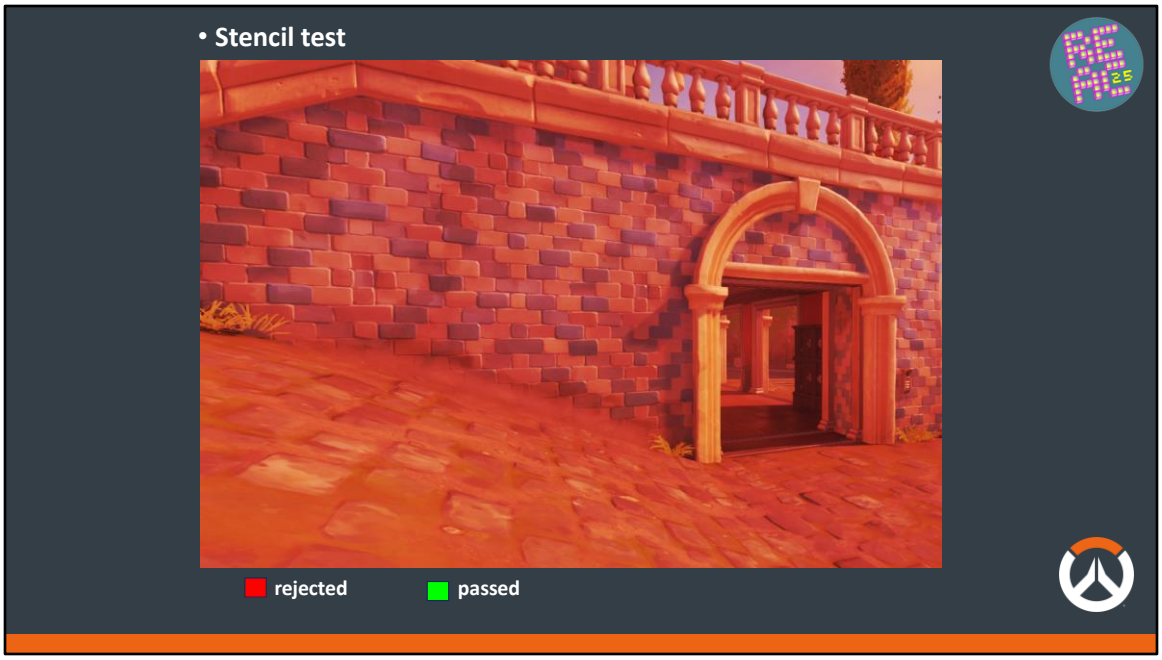
Since we're outside of the smallest volume, we will do the stencil fill pass.
Left: results of depth test
Right: results of stencil test – at this point the GI stencil bits are clear –remember we do a manual discard in the shader if we detect that a pixel is not contained inside the volume.
This pass will set the 'current' bit for non-discarded pixels that pass depth and stencil.
Ignore the purple areas – just leftover from this texture being used earlier in the frame – will not be included in final result



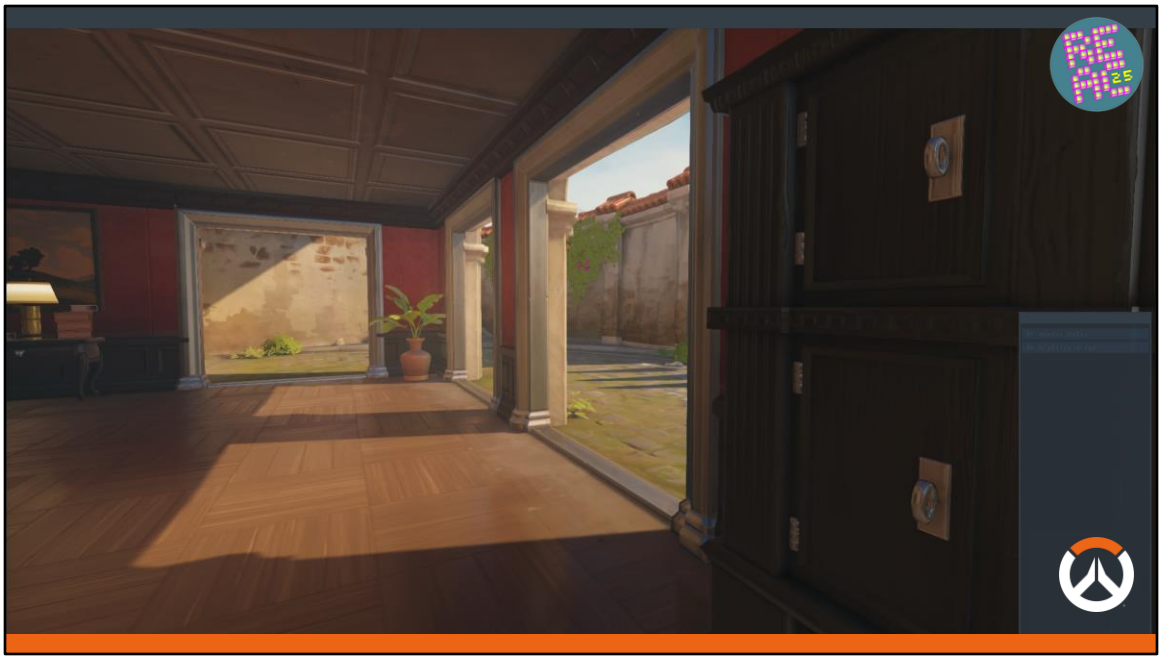
Here are the pixels that pass stencil when we do the actual evaluation of the GI inside the room.
Notice the back doorways were rejected.
This pass reads the ‘current’ stencil bit and only draws where it has been set.
Then this pass clears the ‘current’ stencil bit and sets the ‘any’ stencil bit.



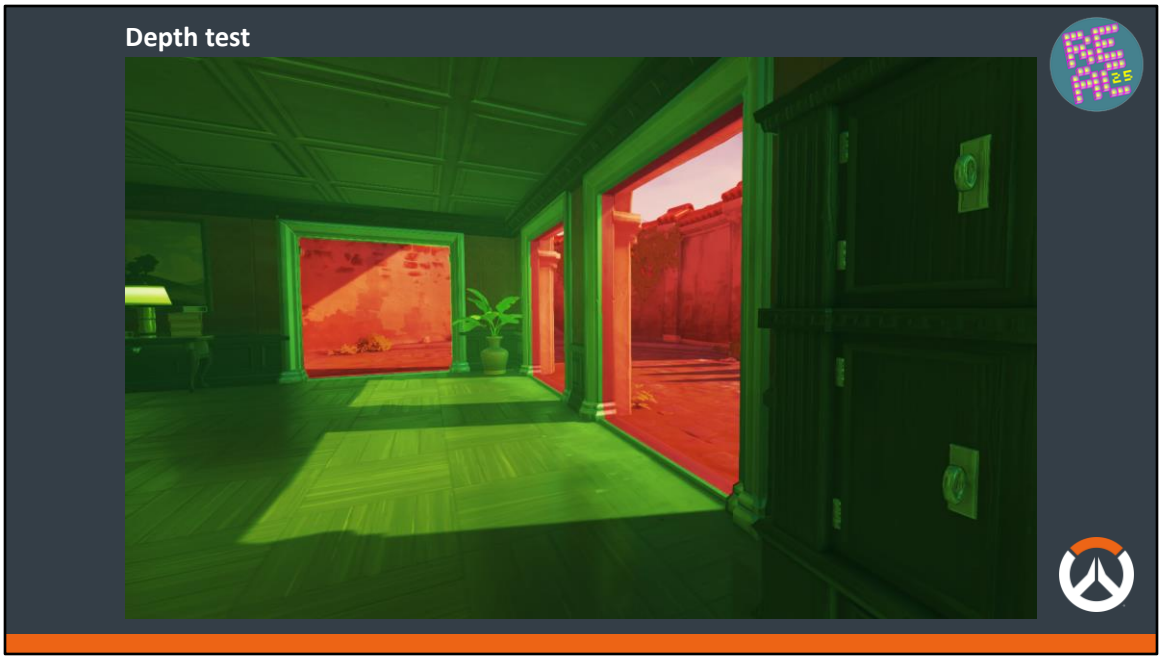
The next volume is large enough that the camera is inside it, so we skip the stencil fill pass.
Notice that the interior of the room is stenciled out, but now the area behind the room will receive GI.
This pass will set the ‘any’ stencil bit for all relevant pixels.



And finally our global volume has no actual work to do since everything has been stenciled out at this point.

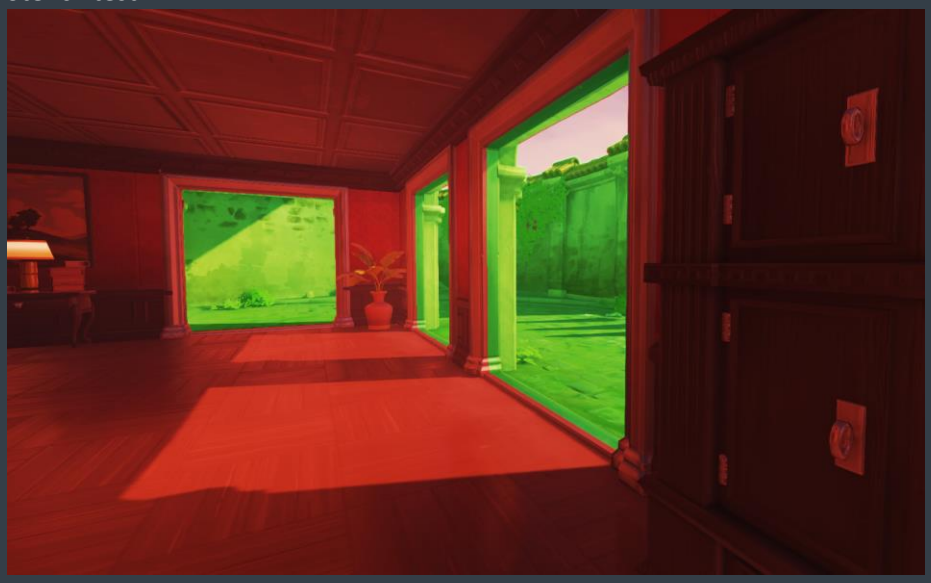


We’ll go through one more quick example to show what happens if you start inside a volume.
This is the same view from before, we’ve just moved the camera into the room.

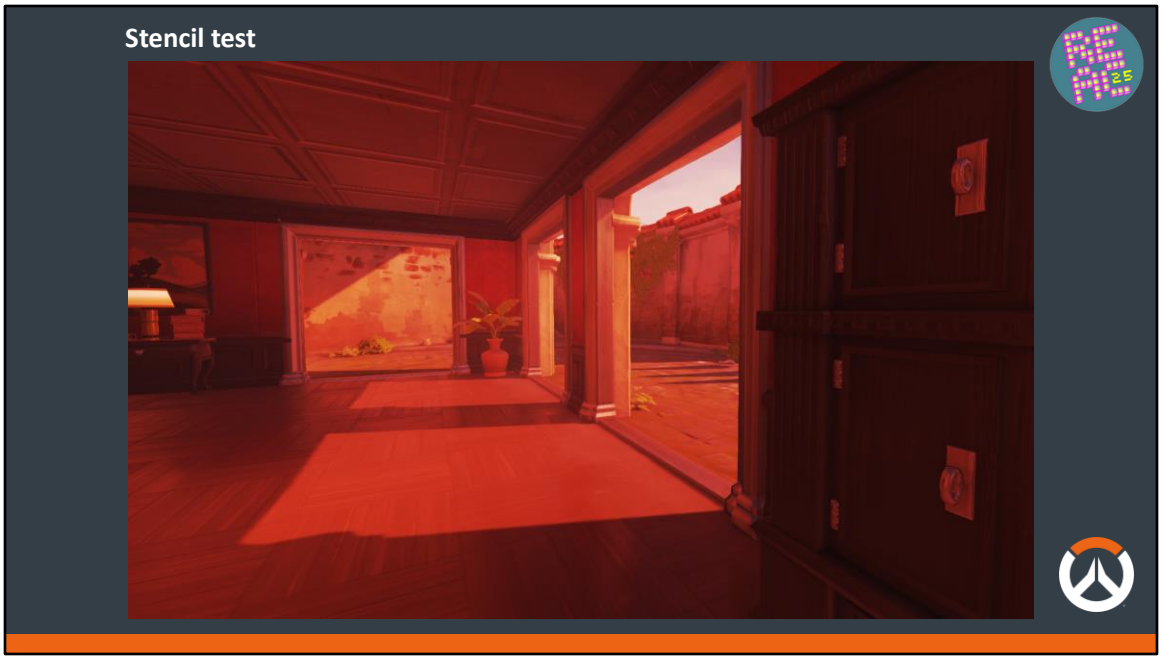


Notice this time that the depth test is rejecting geometry outside of the volume, so everything beyond the large doorways are being ignored.

Stencil test



When we go to the next volume up, there’s very little work left to do.



And again we see that our global volume has no work to do at all.

Quality

- At full resolution, we can evaluate GI per pixel
  - Great quality, little to complain about
- At half resolution, we will need some work
  - We started with just a naïve bilinear upsample
  - Knew we would need something better for shipping

With the performance looking favorable, it's time to move on to quality.
For higher end hardware, we can evaluate GI per pixel at full resolution.
This yields great quality with very little to complain about.
For mid and low end hardware we want to evaluate GI at half resolution then upsample.
We started with just a naïve bilinear upsample to get things on screen, but knew we would need something nicer for shipping.
Any time we say half resolution we mean half on each axis (so ¼ the total pixels of full resolution)

Quality

When that low resolution texture is naïvely upsampled during lighting, you get this hot mess.
It gets worse the longer your stare at it.
And makes you nauseous when you see it in motion.

Quality

- **Experimented with a few options for half res upsample**
  - **Bilinear – obviously out**
  - **Bilateral – improvement but still not the quality we were going for**
  - **Multi-tap – similar to bilateral, more expensive, quality wasn't there**

We experimented with a few simpler upsample techniques first.
Bilinear is where we started but was always intended to be replaced.
A bilateral upsample offered a noticeable improvement, but there were still too many objectionable artifacts to be shippable.
We also experimented with taking multiple samples around the pixel we were reconstructing and reasoning about what the result should be, but it was more expensive and at the end of the day the results were still close to bilateral.

Quality

- Temporal upsample
  - Good fit because diffuse GI is low frequency
    - Hard to spot any ghosting
  - Keeps all the expensive stuff at low resolution
  - Reproject pixel to last frame's location and sample
  - Weight using depth deltas, disocclusion, luma, distance moved, etc.

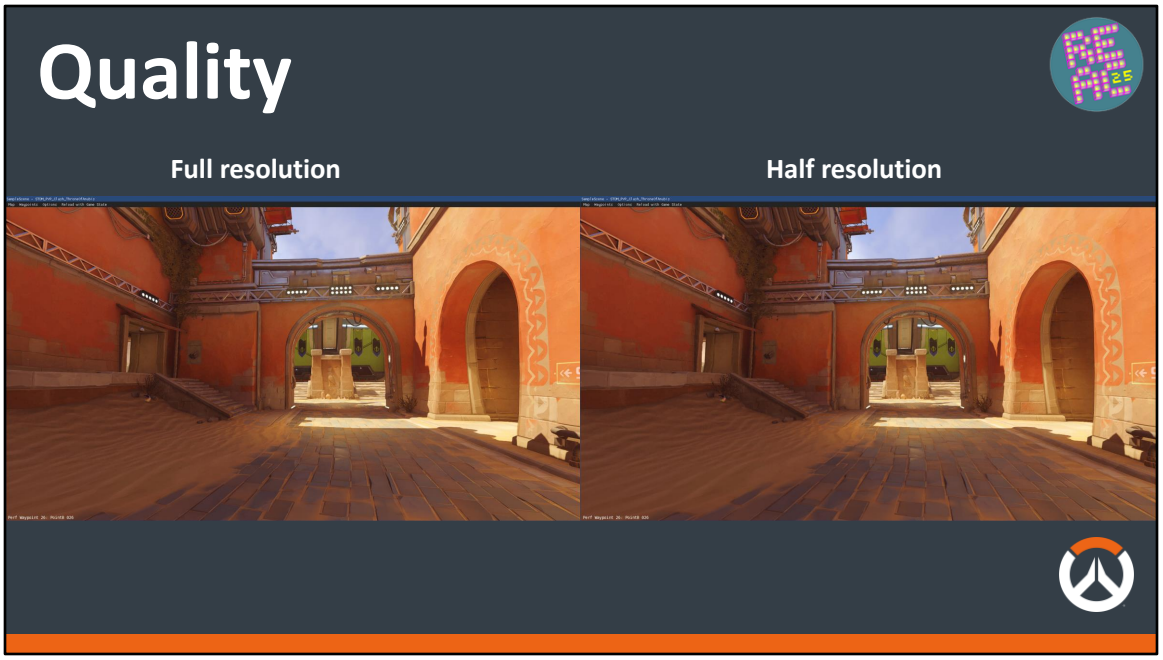
We started digging into a temporal solution where we could accumulate a full resolution image over time and reuse existing data.
We felt a temporal solution would be a good fit due to the low frequency nature of diffuse GI.
Even in traditionally tough cases for temporal techniques, it's difficult to spot artifacts like ghosting.
This allows us to keep all the expensive evaluation stuff at low resolution and then combine those results with a full resolution temporal texture.
We weight new samples based on several criteria such as depth deltas, disocclusion, luma, and distance moved in screen space.

Quality

- **Temporal upsample (cont'd.)**
  - Our lowest-end targets don't have motion vectors
  - Reproject static objects only
    - Calculate on the fly for static objects based on camera motion
  - Dynamic objects use spherical harmonics (SH) evaluated at the center of the object
    - Still uses DDGI input to evaluate SH coefficients, so lighting remains coherent
    - Additional early-out since we know which pixels are for dynamic objects – extra perf back
  - Fire dynamic object SH calculation early in the frame, ready to be read by time we get to lighting

The main issue we needed to overcome with a temporal solution is that the platforms that most need it don't generate motion vectors.
To tackle this, we decided to evaluate dynamic objects using spherical harmonics.
This is something we had already planned on doing for things like particles and effects, so wasn't a huge shift to add support.
We have data stored on the entity representation CPU side to differentiate, and we have a flag in one of our GBuffer fields to indicate dynamic objects on the GPU.
We fire off a compute shader to calculate coefficients for dynamic objects early in the frame and the results are ready by time we get to the lighting pass.
We use the DDGI inputs to evaluate SH, so the lighting remains coherent.
This also gives us an additional early out in the evaluation shader since we know which pixels are dynamic objects.
Then for static objects, we can use just the camera motion to calculate motion vectors on the fly.
Special thanks to our graphics lead, Bruce, who stepped in and helped with a ton of the temporal stuff!

Quality



Here’s a side by side of the same scene with our full resolution evaluation on the left and our half resolution evaluation on the right.
Notice how even in corners and near depth discontinuities the half resolution upsample holds up well.
The difference in the curtains in the background comes from them being dynamic objects.
They are evaluated per pixel at full resolution.
They each use a single set spherical harmonic coefficients at half resolution.

Here’s a larger version of the full resolution result

And the half resolution with temporal upsample result.

Moving to production

- Crawl -> Walk -> Run
  - Start small with limited scope and impact
  - Slowly move up
  - React to failures and shortcomings before moving ahead
- Release Schedule

Season 13	Season 14	Season 15	Season 16
2024/10/15	2024/12/10	2025/02/18	2025/04/22

Once we were confident with our overall performance and quality across all platforms, it was time to start looking at getting this into production.
Remember that all along this journey, we've also needed to maintain our existing GI solution, and that has not changed.
We don't want to ship a new feature of this size and suddenly have everything come crashing down.
We need to right-size our approach and be pragmatic about rollout.
The seasonal content model we have lends itself well to this.
Our first map to receive any of the new GI solution was our Hero Gallery map for reasons we'll discuss shortly.
We rolled it out in Season 13. Initially planned to only do dynamic objects, but we pushed ourselves a little and enabled it for static objects on high-end platforms (current generation consoles, PC high graphics).
In Season 14 we turned it on for everything (static and dynamic) across all platforms.
We also updated more maps in Season 14, which we'll look at shortly.

Moving to production

- Memory
  - We know what to expect per volume based on probe density
  - How does this translate to real maps?

We're updating real maps to use the new solution so let's look at actual numbers.
Memory-wise we know what to expect based on the probe density per volume.
But how does that translate to production maps?

By the numbers

- Memory
  - Hero Gallery
    - Probe memory: 3.46 MB -> 1.43 MB

Hero Gallery Lighting Memory (MB)			
Before		**After**	
Atlas	0.66	Atlas	0.00
UVs	0.88	UVs	0.44
ProbeSets	3.46	ProbeSets	0.01
Baked	3.98	Baked	3.98
ReflProbes	3.00	ReflProbes	3.00
Shadows	0.93	Shadows	0.93
		DDGIProbes	1.42
Total	12.91	Total	9.78

The first map we shipped the new GI solution in was our Hero Gallery map in Season 13.
We wanted to do a controlled and limited rollout since this was a brand new system.
We saw Hero Gallery as a good candidate:
Small map
Lots of use – this is the map that's loaded when inspecting hero cosmetics like skins and emotes
Limited stuff going on – low surface area for unexpected performance or quality issues
The ProbeSets value is the footprint of our existing Enlighten probes, and the new DDGIProbes value is the footprint of the new solution.
We do keep a very small number of Enlighten probes around as a fallback in case we miss updating something in the pipeline – this way they'll still get some (very low resolution) ambient, which we felt was better than no ambient at all.
We'll also clean this up even more to forego loading the Atlas textures, which are the lightmaps.
We will also only load half of the UV set data shown above. We can fully remove lightmap UVs, but will keep baked AO UVs.

By the numbers

- Memory
  - Chateau
    - Probe memory: 3.78 MB -> 2.99 MB

Chateau Lighting Memory (MB)			
Before		**After**	
Atlas	4.39	Atlas	0.00
UVs	1.61	UVs	0.81
ProbeSets	3.78	ProbeSets	0.06
Baked	5.31	Baked	5.31
ReflProbes	10.00	ReflProbes	10.00
Shadows	1.65	Shadows	1.65
		DDGIProbes	2.93
Total	26.74	Total	20.76

The next map we updated to the new GI solution for Season 14 was one our Free-For-All maps called Chateau.
This map is a bit larger than Hero Gallery, has a more extensive interior as well as a basement section.
Most importantly, players can play real matches in this map, so we can monitor performance as chaos ensues.
You can see that the DDGIProbes memory has grown in comparison to the smaller map, but is still coming in under the previous ProbeSets memory, even before we discard the Atlas and UV memory.

Pushing ourselves

- More Free-For-All maps
  - Malevento, Black Forest, Kanezaka, Castillo, Antarctica, Necropolis
  - Similar observations across all maps
    - Usually at or under the previous probe set memory
    - A few cases were slightly over (~0.5 – 1 MB)
    - But always well under once accounting for Atlas and UV data

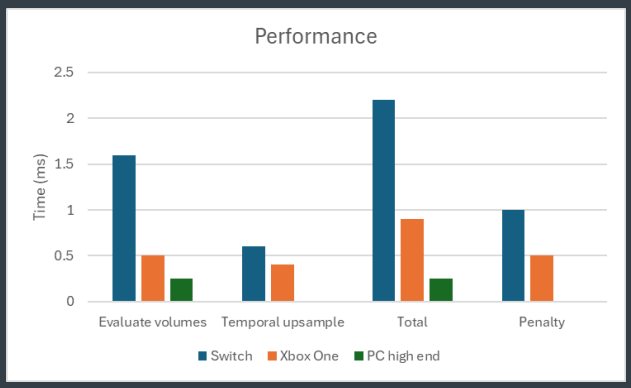
From our initial observations, we felt pretty good about how things were shaping up and decided instead of just releasing an updated Chateau in Season 14, we would push ourselves and test the new solution in even more of our FFA maps.
Same reasoning applies – they're smaller than our typical PvP maps but still have interesting mixes of interiors and exteriors.
Memory gains held up.
Usually at or under the previous probe set memory, save for a few cases.
Always under previous allocation once accounting for Atlas and UV data.
This is probably when we moved from "crawl" to "walk" as we were gaining more confidence.

Going further

- Larger maps
  - Hanaoka
    - Probe memory: 12.6 MB -> 16.5 MB
    - Atlas: 2.5 MB, UVs: 16.63 MB
  - Throne of Anubis
    - Probe memory: 8.3 MB -> 12.1 MB
    - Atlas: 2.07 MB, UVs: 11.62 MB

For Season 15 we wanted to finally test out the new GI solution in larger PvP maps. We decided to update both of our Clash game mode maps, Hanaoka and Throne of Anubis.
For both maps, by time we had added sufficient volumes and density we were over the initial probe memory by a few megabytes.
But once the memory from Atlas plus half the UVs is reclaimed, we're back to being comfortably under our starting point in both cases.

Performance in production



We start by looking at Xbox One and Switch since they are our lowest-powered consoles.
PS4 numbers are very similar to Xbox One.
The Switch numbers aren't great at first glance, but we have a 30 fps target there so we're comfortable with where we landed.
Those number are also not just tacked on to our old frame time. Remember we optimized other parts of our pipeline to get time back, as well as replaced the existing GI evaluation code.
The true "penalty" value for Switch is closer to 1 millisecond, for Xbox One closer to 0.5 milliseconds, and on our high end platforms it's small enough to be considered noise.
Our lowest-end supported PC hardware typically falls somewhere between the Switch and Xbox One, so numbers are similar there.
Of note for our high-end PC as well as current generation consoles (Xbox Series and PS5), since we evaluate GI at full resolution we entirely skip the temporal upsample so only pay the cost of rendering the volumes themselves.
Numbers presented are all for worst-case scenario – entire GBuffer full of pixels that need to be evaluated.
They start to drop off fast when you have more sky and background in view.

Ready to run?

- **Stadium**
  - **Season 16**
  - **Brand new game mode**
  - **All new maps to use new GI solution**
  - **Feature multiple times of day at release**

Stadium – our new game mode starting in Season 16.
Nine new maps (some based on existing locations).
Every map to feature multiple times of day at release.
Want this to be the inflection point where we fully use our new solution for all new maps going forward.
Desire for faster iteration times for artists – we'll talk about this in a bit. Despite being confident in the runtime portion of the technique, artist workflow is an area where we are still in the "walk" phase.
We were able to meet this goal and ship all Stadium maps with the DDGI-based solution.

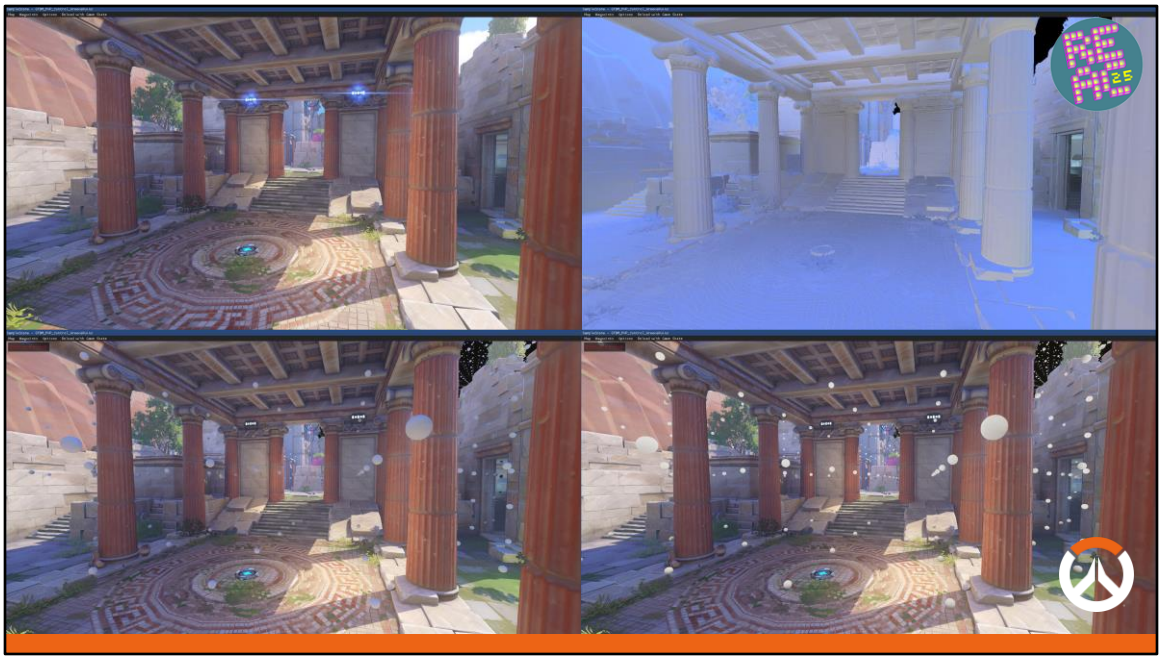
A small collection of images showing some of our new Stadium maps during different times of day.

A small collection of images showing some of our new Stadium maps during different times of day.

Debugging

- Any large system needs some type of debugging support
  - Debug views
  - Metric reporting
    - Memory, performance, etc.
  - Value tracking and observation
  - Whatever is useful!
- Build debugging tools early and alongside everything else
  - Have them before you need them!

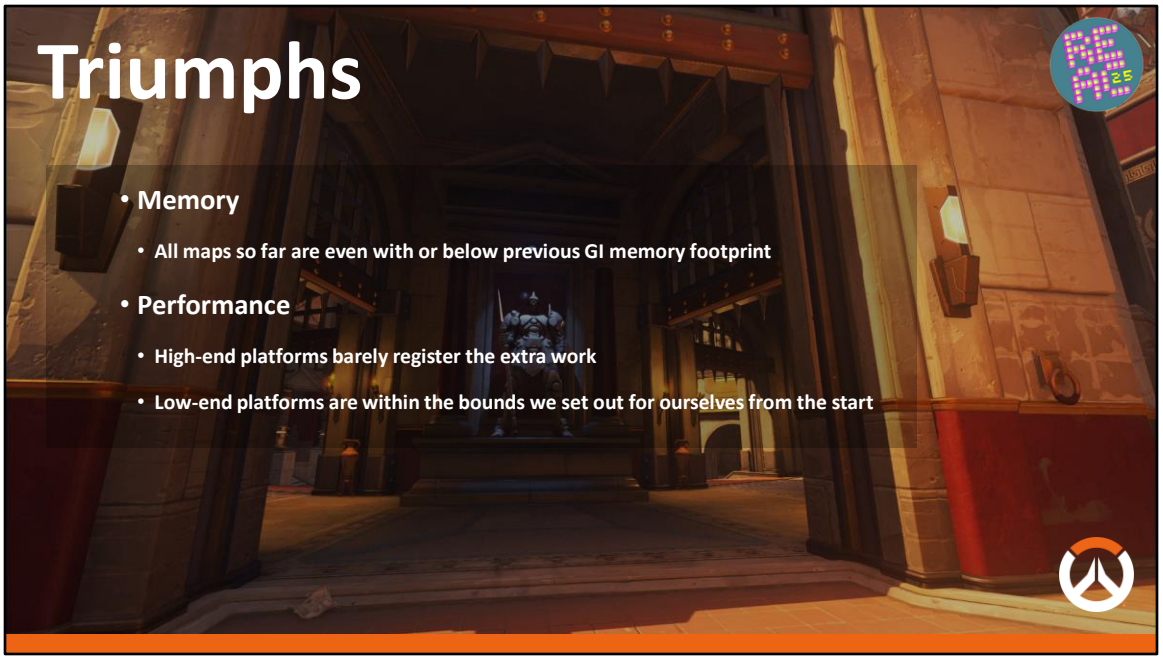
Take a brief detour to talk about debugging utilities.
Incredibly important when building a new system of any size and complexity to have some form of debugging available.
Can be any combination of debug views, metric reporting, value observation, or anything else that's useful.
Build these early and alongside everything else.
You want to have them before you need them.



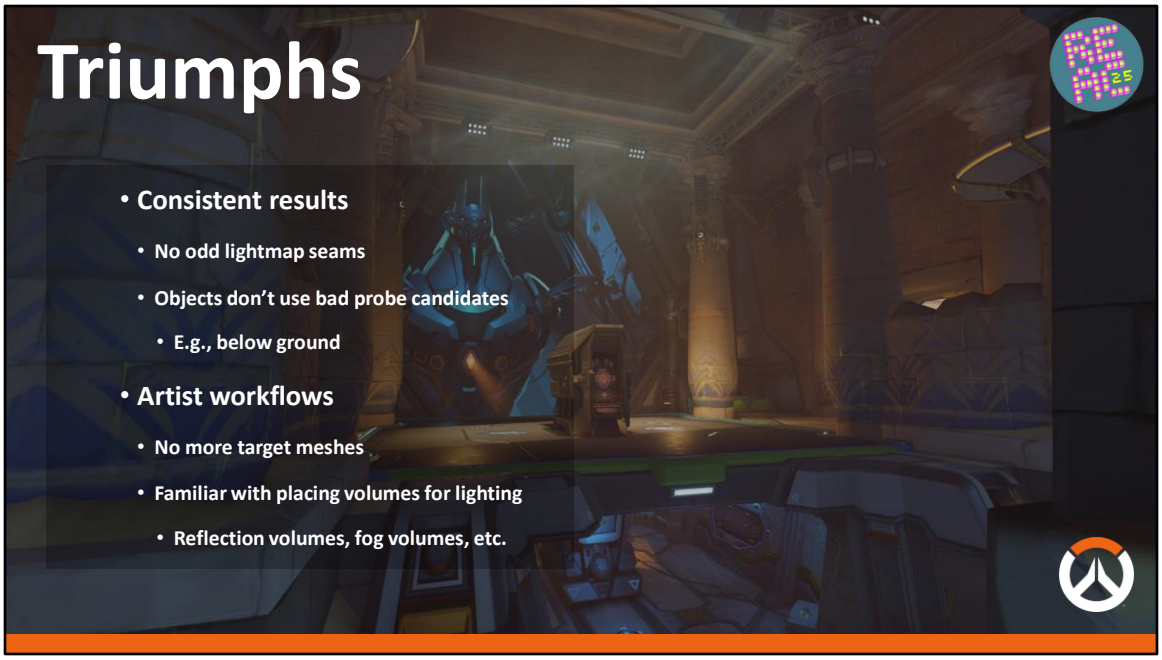
Here are a few of the debug views we built early on.
Top left is the scene with default rendering.
Top right is the ambient light view.
Bottom left shows the probes in space, with each probe showing irradiance.
Bottom right shows the probes again, but this time showing visibility information, where a brighter probe means better visibility and a darker probe means it has worse visibility.
Entirely black probe means it sees too many backfaces – e.g., inside a wall

Debugging

This is another super useful tool that another one of our engineers, Marco, worked on as part of some ray tracing work he was doing.
The little window shows the world we ray trace against with Embree.
Red means front-face hit, blue means back-face hit.
Turned out to be invaluable when trying to figure out why a probe has poor visibility.
We have found several cases of geometry erroneously poking into rooms and wreaking havoc with probe visibility using this tool.

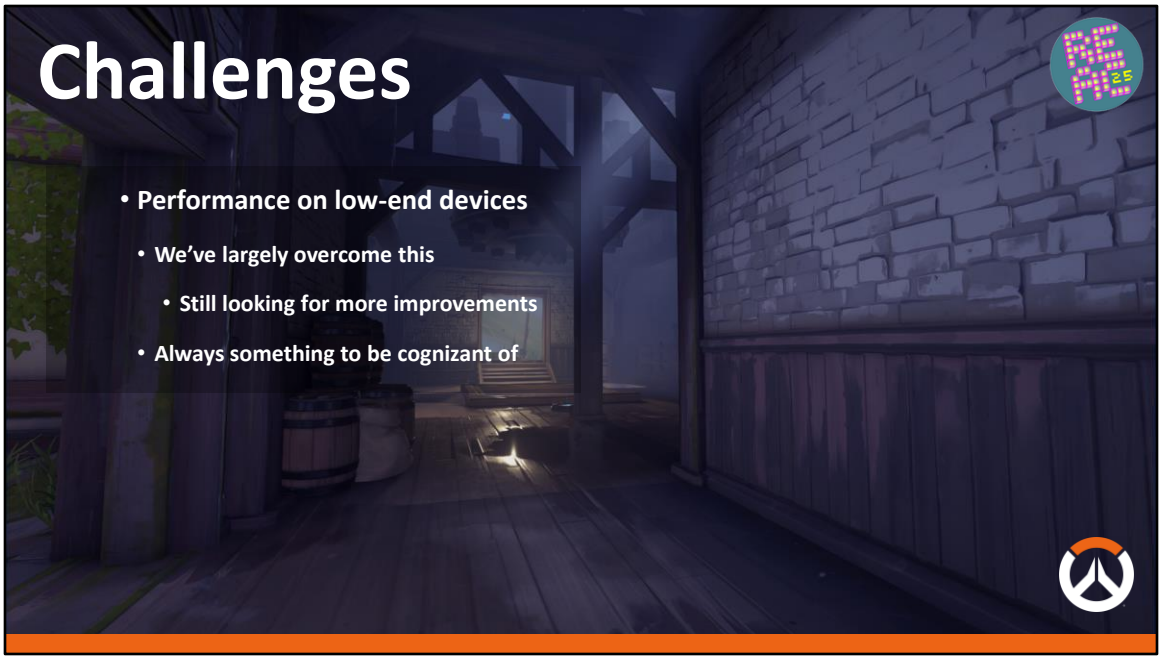


Recap what's gone well.
We met our goal of shipping all new Stadium maps with our new GI solution.
Maps are even with or below their previous memory footprints for GI.
Our high-end performance numbers look excellent, and our low-end platforms are within the bounds we set for ourselves at the start.

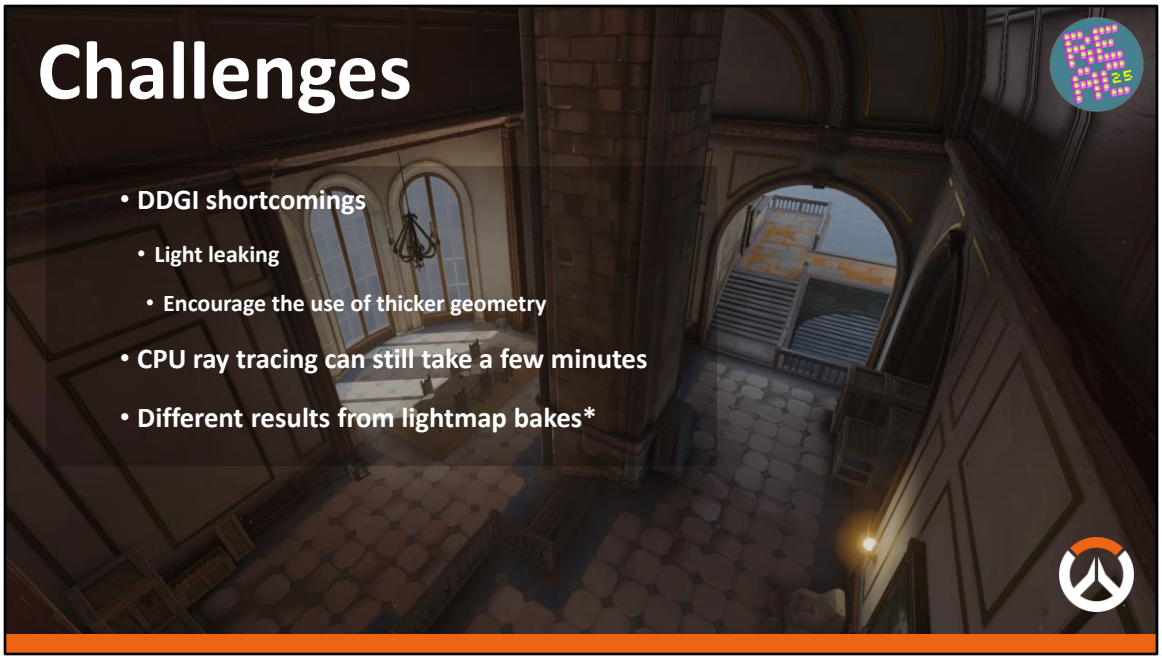


We're now getting consistent results with GI, regardless of if the object being drawn is static or dynamic.
We've done away with some objects getting bad lightmap seams and other objects using bad probe candidates for lighting.
We've improved the artist workflow by giving them tools they're already familiar with in placing volumes.
Removed the need for artists to create lighting target meshes in additional to the scene geometry meshes.

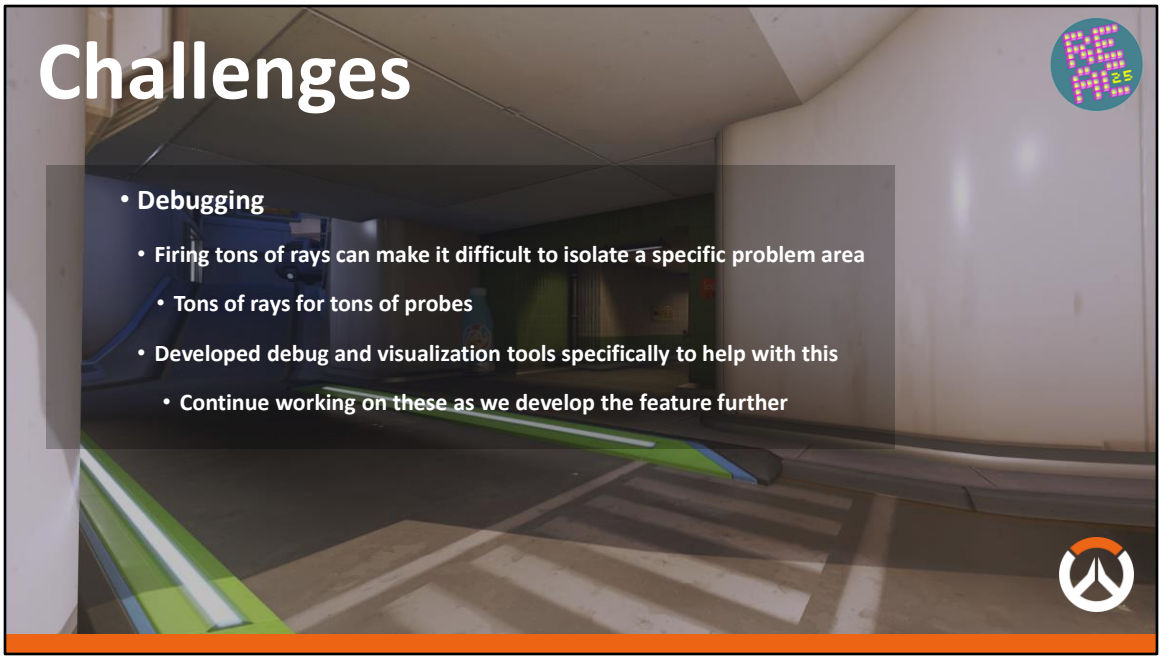
Our bake times with Enlighten could take between 30 minutes to well over an hour depending on the map size, complexity, and the number of lighting scenarios.
Run distributed across several machines.
Bake times with DDGI generally top out at a handful of minutes even for complex maps.
Fast enough to run locally.
Could still speed up a bit by, for example, capturing one lighting scenario locally and another distributed.
And we have plans in the future to speed this up even more.



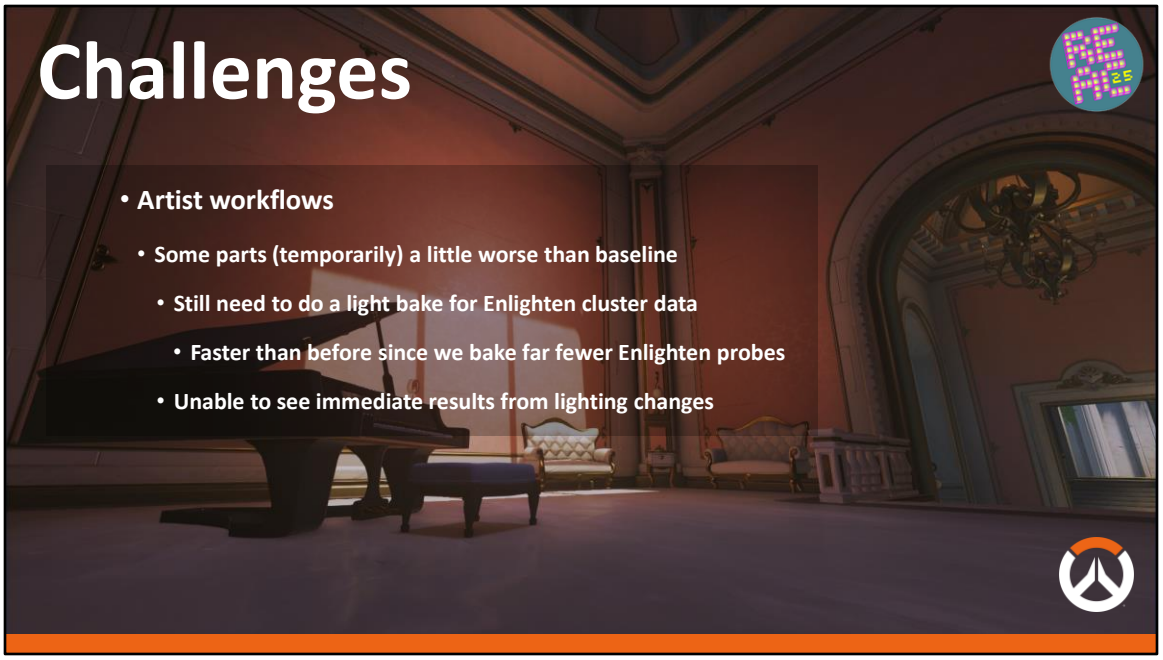
We do still have some challenges ahead of us.
We are still actively investigating performance especially on low-end devices to see if there's additional time we can get back.
We're not unhappy with the current numbers, but we also feel they could be a little better.



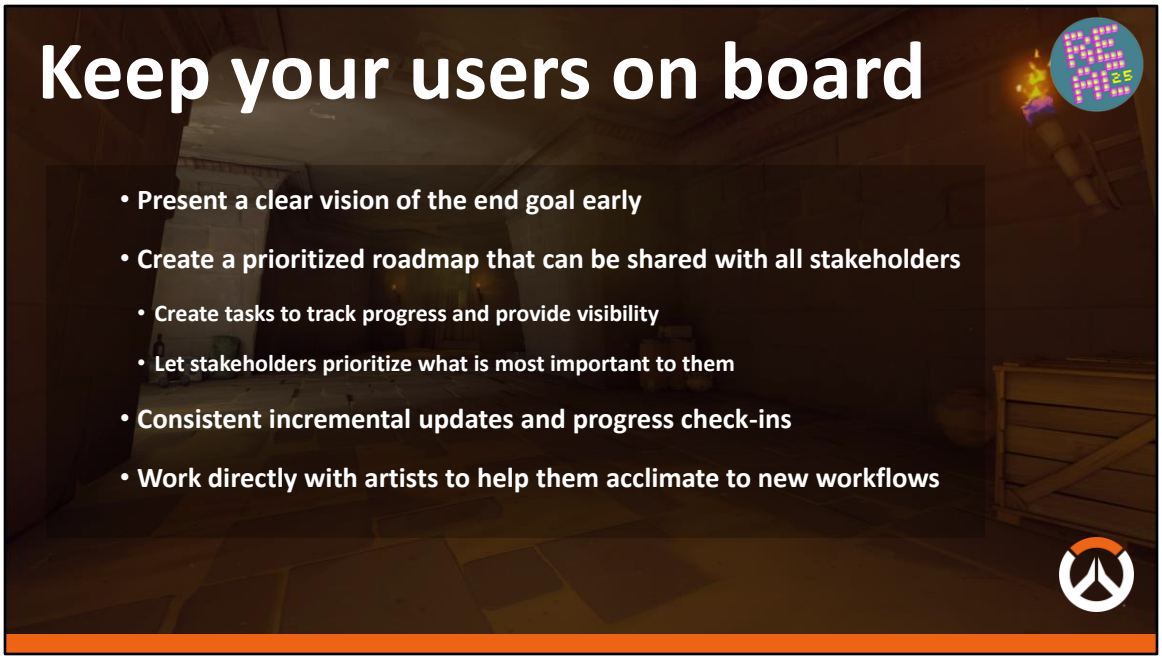
DDGI as a technique has its own shortcomings.
There can still be light leaking in some areas.
Mainly due to probe spacing and thin geometry.
Balance between solving every case and using more memory with child volumes to cover tricky areas.
Visibility data stored in first two channels of BC6 texture – can we use the free channel for something useful to help alleviate more light leaking?
CPU baking and ray tracing can still take a few minutes, which is something we'd like to see improved.
Results from baking DDGI probes can also be different from our lightmap bake results.
We're solving GI in space around a surface rather than directly along the surface.
This one gets an asterisk because sometimes the lightmap bake gets things wrong, as well, so it might be a correct result that looks different from what were previously outputting.



Debugging can still be a challenge in some cases.
Firing thousands of rays from thousands of probes can make isolating a specific problem area difficult.
We've shown and talked about a few of the debug tools we have already built, and we're continuing to work on building useful visualizers and other tools as we continue to develop the feature.



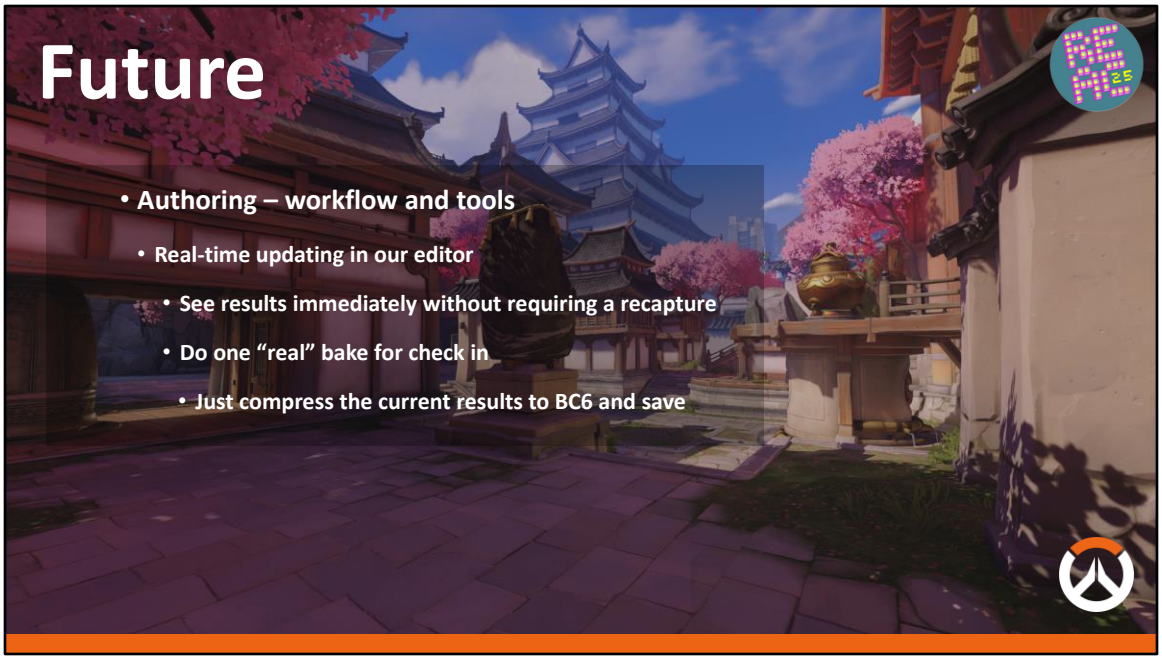
And our current biggest hurdle.
Remember I said you were seeing this partway through the full development and implementation cycle.
To meet deadlines for shipping we had to focus on getting the runtime portion as good as possible before focusing as much effort on the authoring side.
This has unfortunately resulted in the artists having a temporarily worse workflow in some regards.
Since we're still using Enlighten cluster data, we still need to do an Enlighten bake to generate that data.
Those bakes are faster than they used to be since we bake a significantly smaller set of Enlighten probes – typically just used as fallback but eventually can remove them altogether.
This means that bake times in practice haven't improved as dramatically as we'd like them to yet.
Artists are also currently unable to see immediate results from lighting changes until they re-capture the DDGI textures.



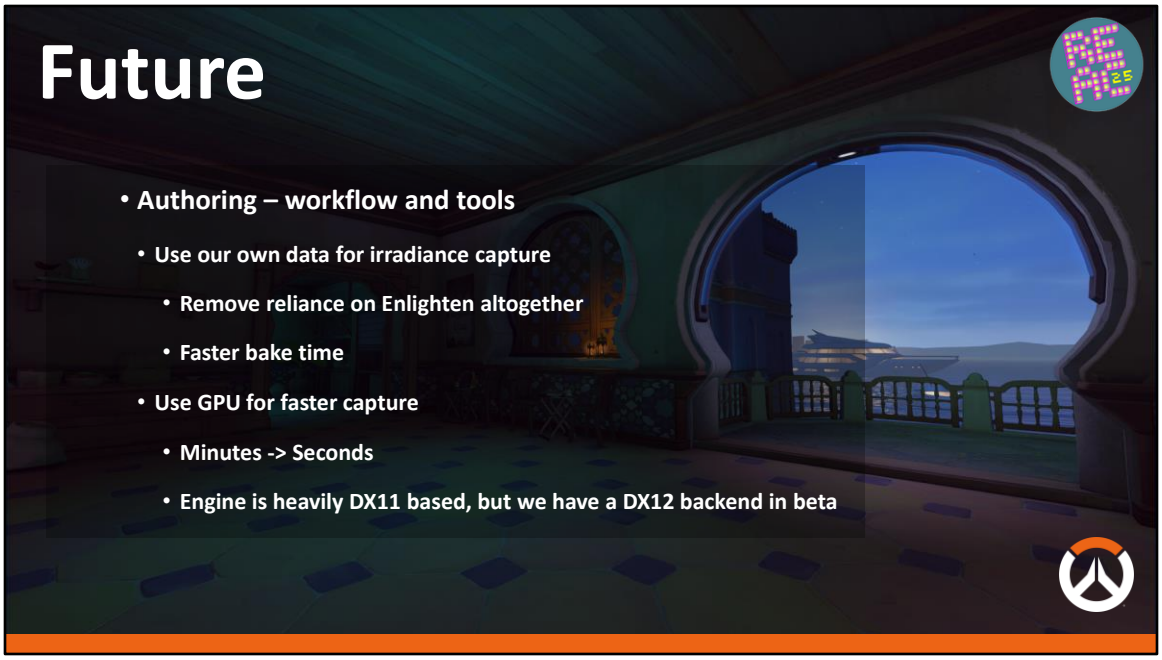
Discussing challenges, especially the ones we're currently imposing on artist, dovetails nicely into mentioning this.
It's important to keep your users on board from the start.
As soon as we were confident in our prototype, we had a big meeting with art to introduce them to the new system.
Showed a few maps that we had updated ourselves to use DDGI.
Of course we talked about all the ways it could improve things, but we were also upfront about the drawbacks, even if temporary.
This was also a great opportunity to get a first round of feedback.
We took the feedback from this meeting and created a prioritized roadmap that anyone can review to see what's been done and what's left.
Make tasks/tickets out of everything!
Since that first meeting, we've provided regular incremental updates and progress check-ins when a new feature is added.
We also take time to work directly alongside the artists as they're acclimating to the new workflow:

a) To help them understand the current state of tooling
b) To get valuable feedback for things that we hadn't thought of that they think could be better

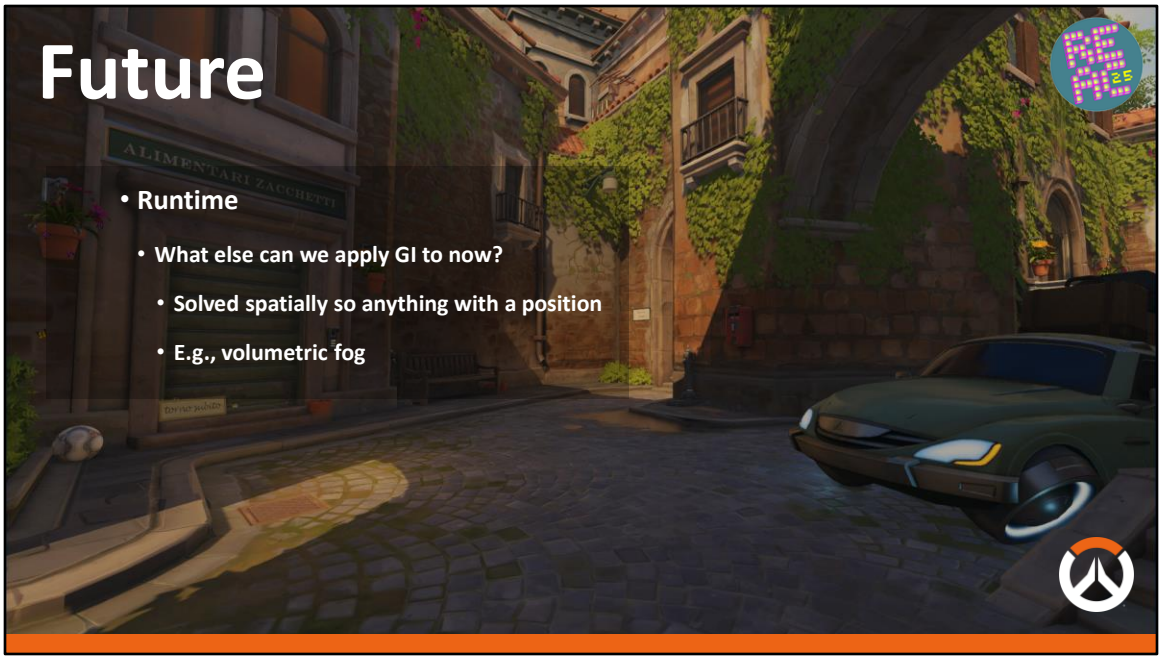
Artists are being patient with us with the temporarily worsened workflow, the least we can do is make sure we're building the final product into what will help them do their job best.



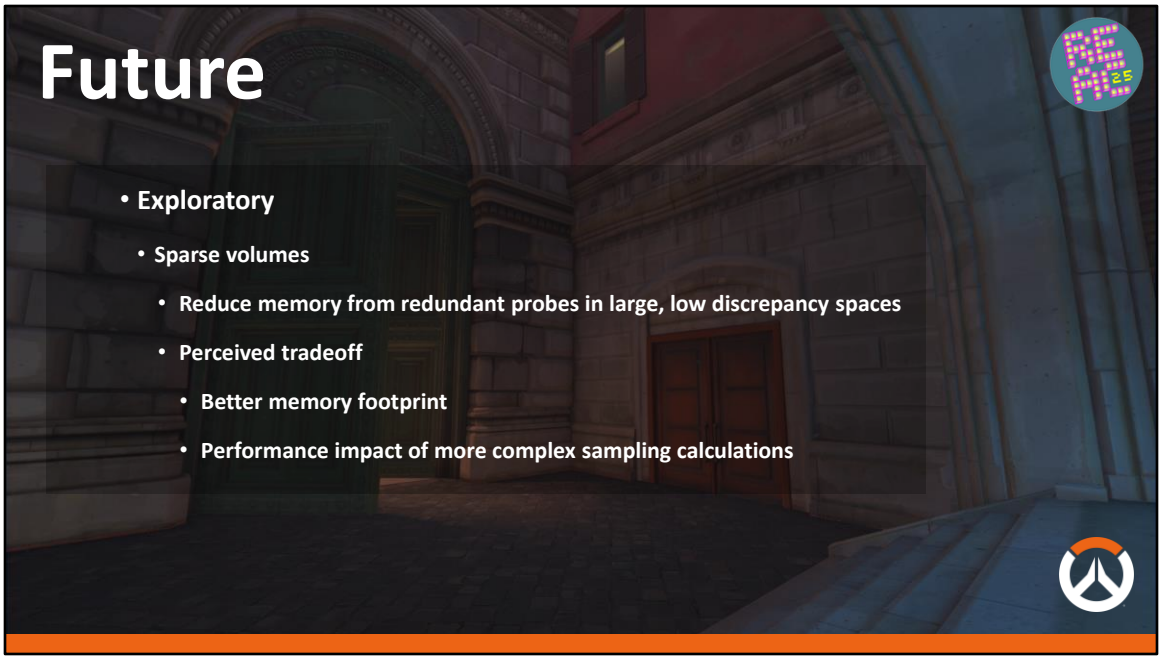
Now that we’re shipping, we can focus back on our patient artists and start easing their workflow woes.
High in the list of priorities is to re-enable real-time lighting updates in the editor.



We also want to fully move off using Enlighten's data and use our own materials, etc., for lighting.
This will remove the need to do the full lighting bake and get us down to just baking what DDGI needs.
We already use our own geometry for tracing visibility since we just care about hit or not hit in that case.
Also want to experiment more with using the GPU to make capture times even faster. Running the out-of-the-box DDGI algorithm on the GPU could take our bake times down from minutes to seconds.



Now that we have this spatial GI data, we want to see where else might be interesting to apply it on the runtime side of things.
An example is our volumetric fog.
Currently uses a single ambient value.
Using DDGI data could give nice spatial variation.



There is also some exploratory work we'd like to do, time permitting.
One example of this is looking into sparse volume representations.
Aim of reducing memory from redundant probes in large, low discrepancy spaces (e.g., the sky).
Perceived tradeoff between better memory footprint and a performance impact due to more complex sampling calculations.
Like we've discussed already – don't rely on assumptions – make your hypothesis, measure before, measure after

Timeline

- From concept to production
  - Summer 2021 – initial talk with other graphics programmers
    - "wouldn't it be neat if…"
  - Unrelated features, tasks, and support
  - October 2022 – Overwatch 2 launch
  - Unrelated features, tasks, and support
  - Early 2023 – initial prototype and data collection
  - Mid 2023 – unrelated features, tasks, and support
  - Late 2023 – deep dive into getting performance back and figuring out if this was actually tenable
  - Early 2024 – unrelated features, tasks and support
  - Mid 2024 – integrated fully into engine, demo for art, green light, get into first production map
  - October 2024 – first map shipping with DDGI enabled

The timeline for this feature has been interesting and a bit all over the place – a direct result of supporting a live game while also building a new feature.
It started as a conversation I had with some of the other graphics programmers during the summer of 2021.
But we had too much to do to get Overwatch 2 shipped at this point so couldn't really start pursuing it.
Overwatch 2 shipped in October, and even for a good while afterwards, everyone was mostly focused on reacting quickly to bugs and other issues.
In early 2023, finally had some time to put together the initial prototype.
But then had to pivot onto feature requests and additional support tasks for a while.
Late 2023, picked the prototype back up and went through a performance deep dive to get things to a point where we could feel confident in our ability to ship.
Early 2024 saw more feature requests (including a new BRDF!) and support tasks.
Mid 2024 was the hunker-down time. This is when we fully integrated the prototype into the engine and finished up the temporal upscale.
October 2024 we shipped the first DDGI-enabled map as part of Season 13.
If you're wondering where the research phase fit into all of this, pretty much any downtime I had between the initial conversation in 2021 and starting on the first prototype in 2023 I was trying to find and read as many papers, posts, etc., that I

could get my hands on to try and determine what our approach should be.

Acknowledgements

- **Engine & Graphics**
  - Bruce Wilkie
  - Marco Alamia
  - David Givone
  - Oleksander Polischuk
  - Will Reynard
  - Ian Tran
  - Phil Teschner
  - Josh Braendel
  - Matt Hines
  - Otmar Schlunk
  - James Touton
- **Tools**
  - Julie Anne Brame
  - Paul Skibitzke
  - Steven Garcia
- **Art**
  - Peter Tran
  - Mike Hardison
  - Marie Lazar
  - Phil Wang
- **QA**
  - David Butterfield
  - Jessica Castillo
  - Meko Morgan
  - Jennifer Powell
- **Production**
  - Calen Stivers

Thank you to our engineers, artists, and QA members who have been critical in helping get this technique from concept to production.

References

- Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields
  - https://jcgt.org/published/0008/02/01/
- Scaling Probe-Based Real-Time Dynamic Global Illumination for Production
  - https://jcgt.org/published/0010/02/01/
- Integrating Dynamic Diffuse Global Illumination
  - https://developer.nvidia.com/blog/an-engineers-guide-to-integrating-ddgi/

Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields:
https://jcgt.org/published/0008/02/01/
Scaling Probe-Based Real-Time Dynamic Global Illumination for Production:
https://jcgt.org/published/0010/02/01/
Integrating Dynamic Diffuse Global Ilumination:
https://developer.nvidia.com/blog/an-engineers-guide-to-integrating-ddgi/


Thank you!

Bonus slides

Bonus slides start here.

Screenshots

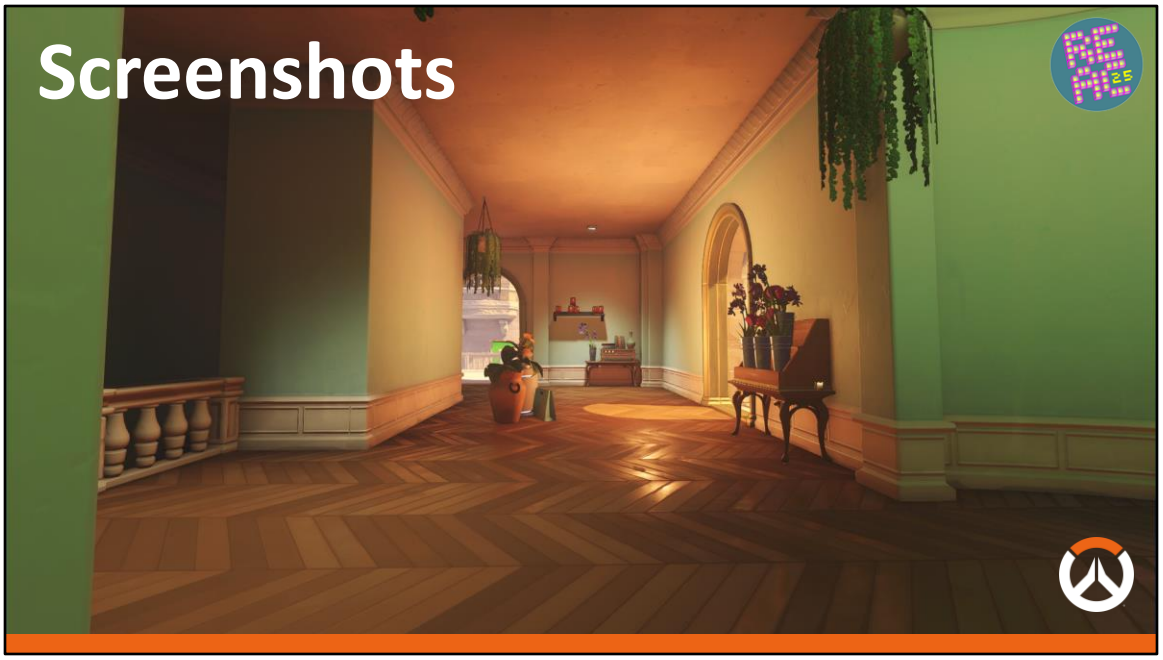
Screenshots

Screenshots

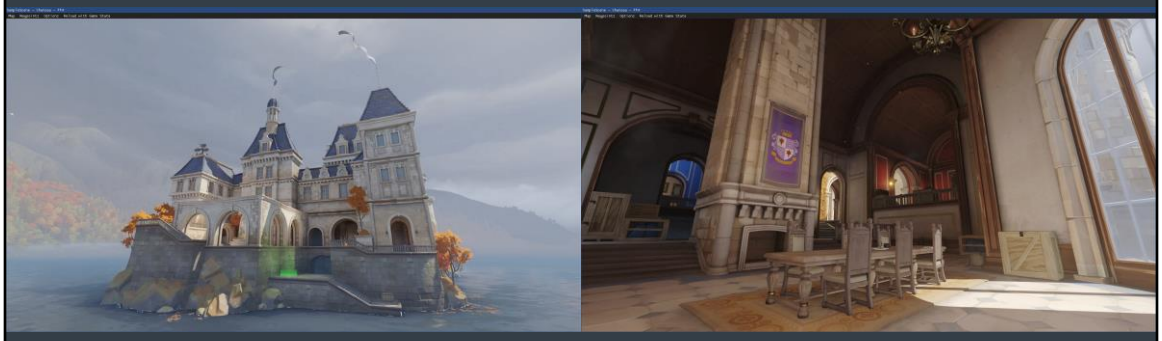
Screenshots

Screenshots

Screenshots

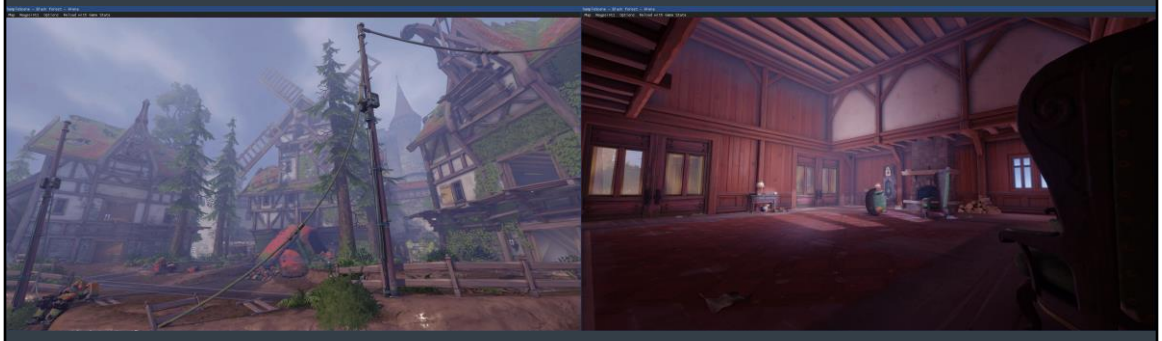


Screenshots



Screenshots

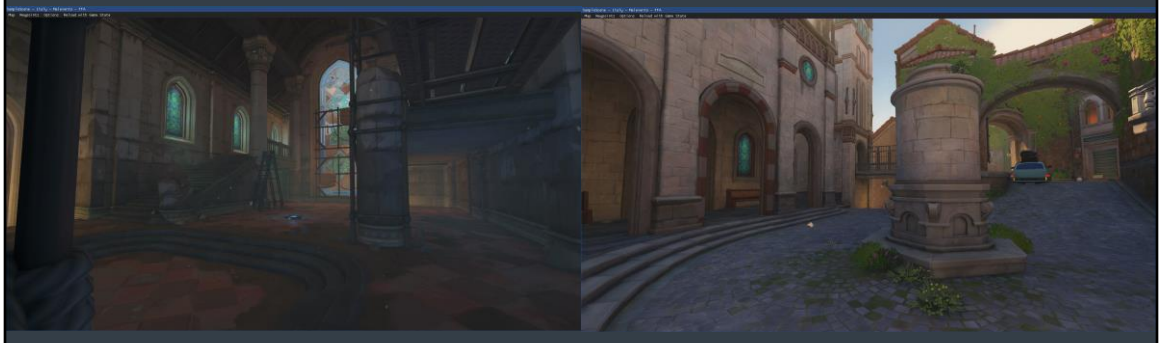
Screenshots



Screenshots

Screenshots

Screenshots



Screenshots

Screenshots