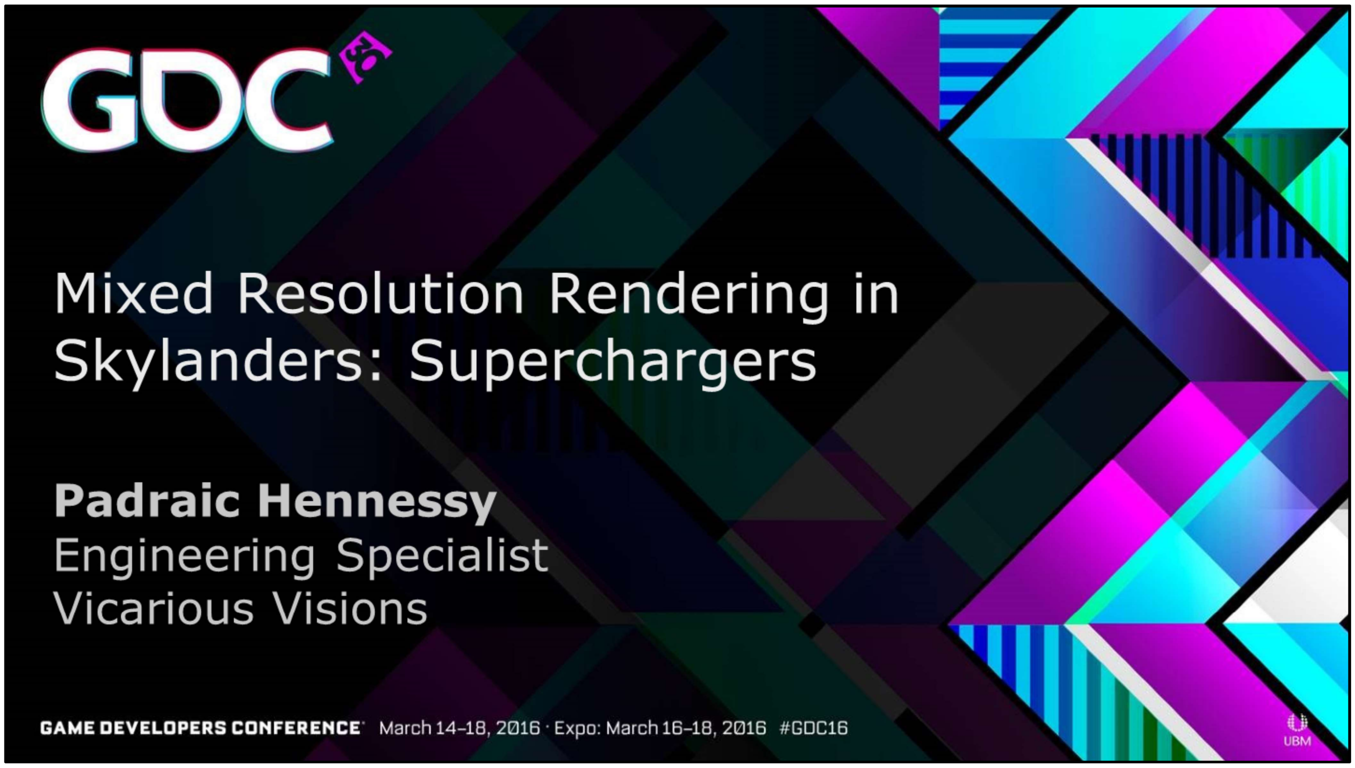


GDC

Mixed Resolution Rendering in Skylanders: Superchargers

Padraic Hennessy
Engineering Specialist
Vicarious Visions

GAME DEVELOPERS CONFERENCE® March 14–18, 2016 · Expo: March 16–18, 2016 #GDC16

UBM

If you are unfamiliar with the skylanders franchise, it’s a fantasy action adventure game that takes players to the skylands which is a magical place suspended in the air.

Players travels through out the skylands completing quests and challenges unlocking powers and abilities for over 200 playable characters.

In Superchargers we introduced vehicle gameplay where the player can take to land sea or air interesting vehicles.

These are some concepts and screenshots to show the amount of variety we are going for.

If you are unfamiliar with the skylanders franchise, it’s a fantasy action adventure game that takes players to the skylands which is a magical place suspended in the air.

Players travels through out the skylands completing quests and challenges unlocking powers and abilities for over 200 playable characters.

In Superchargers we introduced vehicle gameplay where the player can take to land sea or air interesting vehicles.

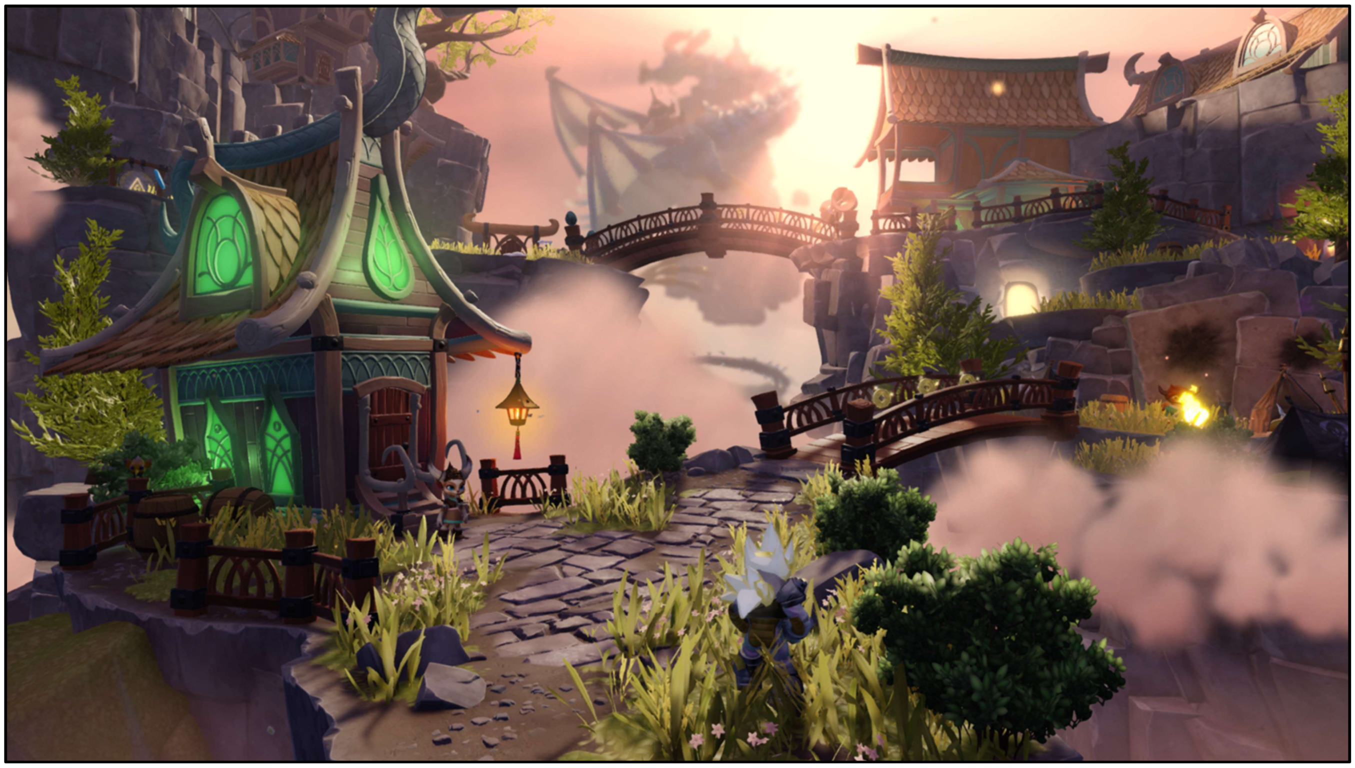
These are some concepts and screenshots to show the amount of variety we are going for.

If you are unfamiliar with the skylanders franchise, it’s a fantasy action adventure game that takes players to the skylands which is a magical place suspended in the air.

Players travels through out the skylands completing quests and challenges unlocking powers and abilities for over 200 playable characters.

In Superchargers we introduced vehicle gameplay where the player can take to land sea or air interesting vehicles.

These are some concepts and screenshots to show the amount of variety we are going for.

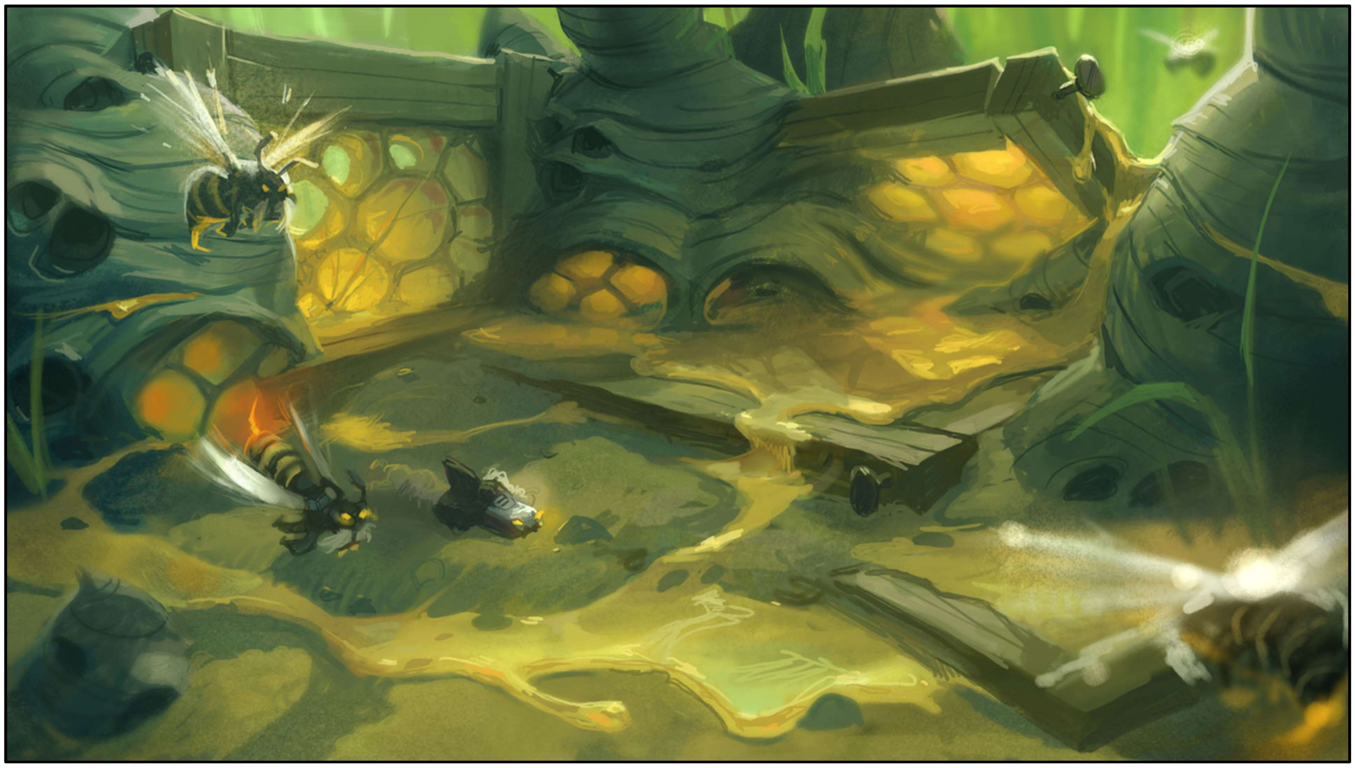


If you are unfamiliar with the skylanders franchise, it's a fantasy action adventure game that takes players to the skylands which is a magical place suspended in the air.

Players travels through out the skylands completing quests and challenges unlocking powers and abilities for over 200 playable characters.

In Superchargers we introduced vehicle gameplay where the player can take to land sea or air interesting vehicles.

These are some concepts and screenshots to show the amount of variety we are going for.



If you are unfamiliar with the skylanders franchise, it’s a fantasy action adventure game that takes players to the skylands which is a magical place suspended in the air.

Players travels through out the skylands completing quests and challenges unlocking powers and abilities for over 200 playable characters.

In Superchargers we introduced vehicle gameplay where the player can take to land sea or air interesting vehicles.

These are some concepts and screenshots to show the amount of variety we are going for.

If you are unfamiliar with the skylanders franchise, it’s a fantasy action adventure game that takes players to the skylands which is a magical place suspended in the air.

Players travels through out the skylands completing quests and challenges unlocking powers and abilities for over 200 playable characters.

In Superchargers we introduced vehicle gameplay where the player can take to land sea or air interesting vehicles.

These are some concepts and screenshots to show the amount of variety we are going for.

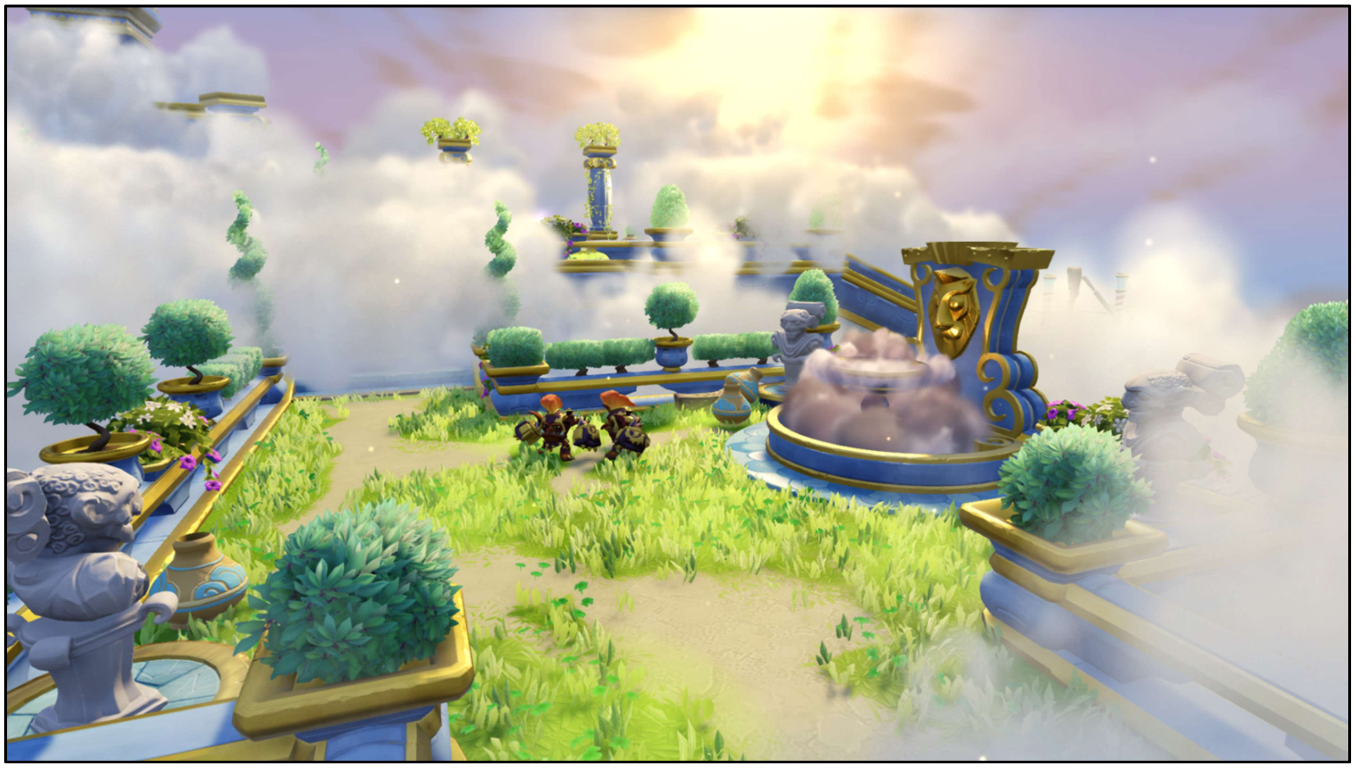
MIXED RESOLUTION RENDERING

So why did we look into mixed resolution rendering?

Being in the skylands we really wanted to sell the feeling of being suspended in the air.

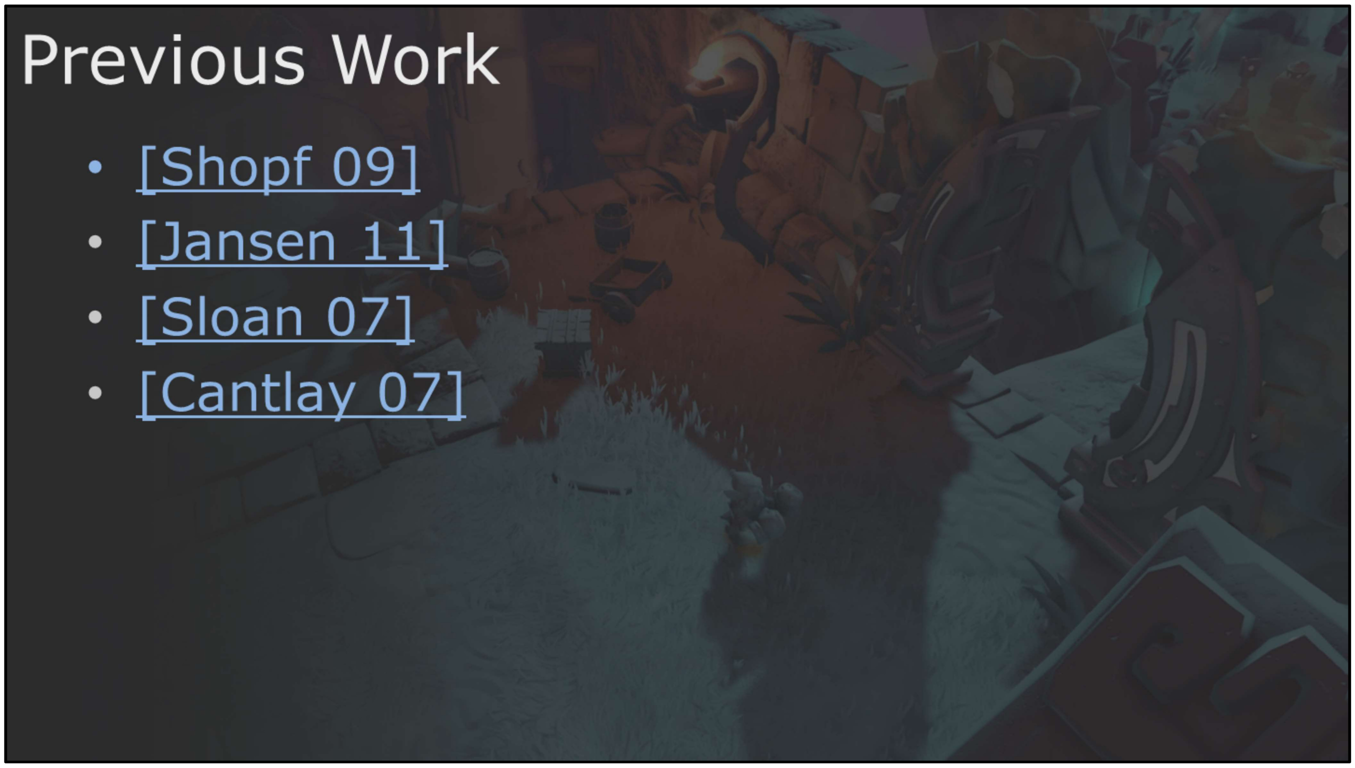
With the addition of air vehicles it became even more important to have interesting atmosphere to travel with in.

With these goals we started looking at cloud rendering tech, like real clouds not internet cloud….



After prototyping a bunch of different approaches we decided to go forward with a sprite based approach so we could have it working on 360 and ps3.

While it was capable of achieving the aesthetic we were targeting we still had to address the fill rate performance of having these overlapping sprites.



Previous Work

- [Shopf 09]
- [Jansen 11]
- [Sloan 07]
- [Cantlay 07]

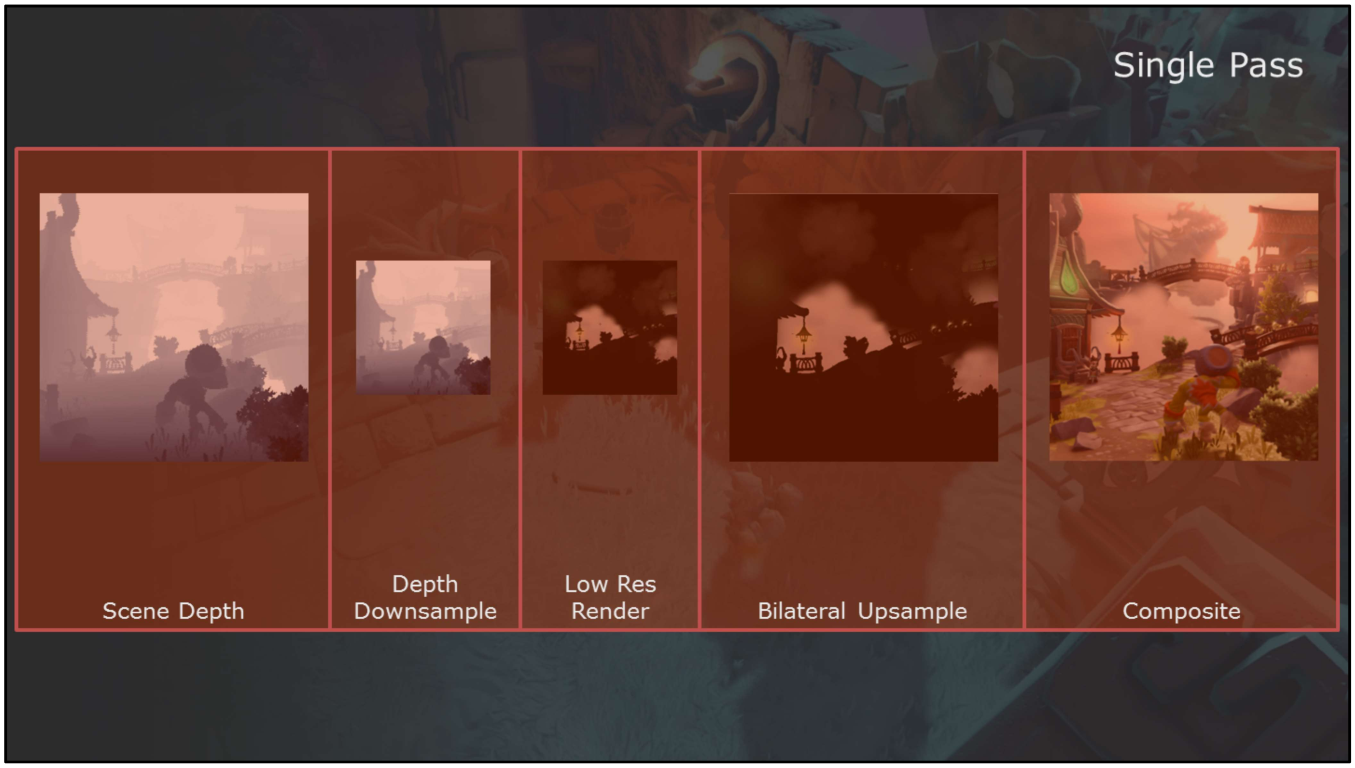
At this point we began our process of investigating what previous work was done in this area. Turns out there was a lot of it, there are a few of the references that we started our implementation from.

[Shopf 09] Jeremy Shopf, “Mixed Resolution Rendering”, GDC 2009
http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/ShopfMixedResolutionRendering.pdf

[Jansen 11] Jon Jansen, Louis Bavoil, “Fast rendering of opacity-mapped particles using DirectX 11 tessellation and mixed resolutions”, Whitepaper
http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/OpacityMappingSDKWhitePaper.pdf

[Sloan 07] Peter-Pike Sloan, Naga K. Govindaraju, Derek Nowrouzezahrai, John Snyder, “Image-Based Proxy Accumulation for Real-Time Soft Global Illumination”, Pacific Graphics 2007
http://www.ppsloan.org/publications/ProxyPG.pdf

[Cantlay 07] Iain Cantlay, “High-Speed, Off-Screen Particles”, GPU Gems 3
http://http.developer.nvidia.com/GPUGems3/gpugems3_ch23.html



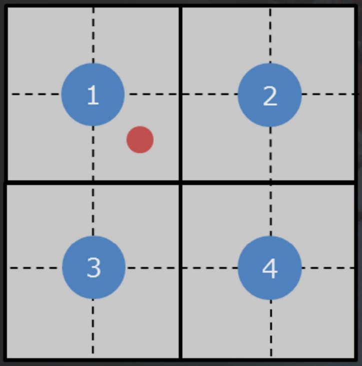
Single Pass

The initial implementation of mixed resolution rendering that we used is what I'm calling a single pass.

The scene depth is downsampled then we rasterize against that lower resolution buffer

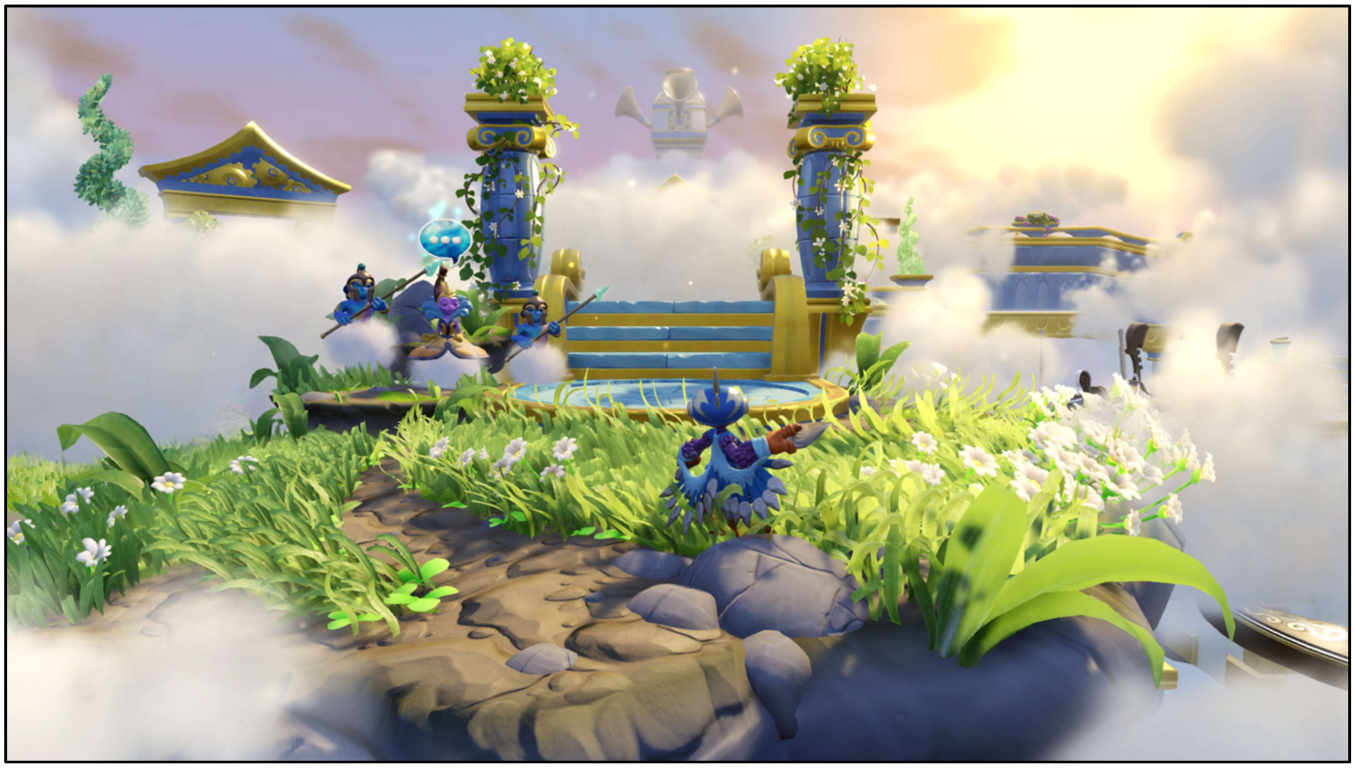
Then that low resolution render is upsampled using a bilateral upsample and then finally composited on to the scene buffer.

Bilateral Upsample



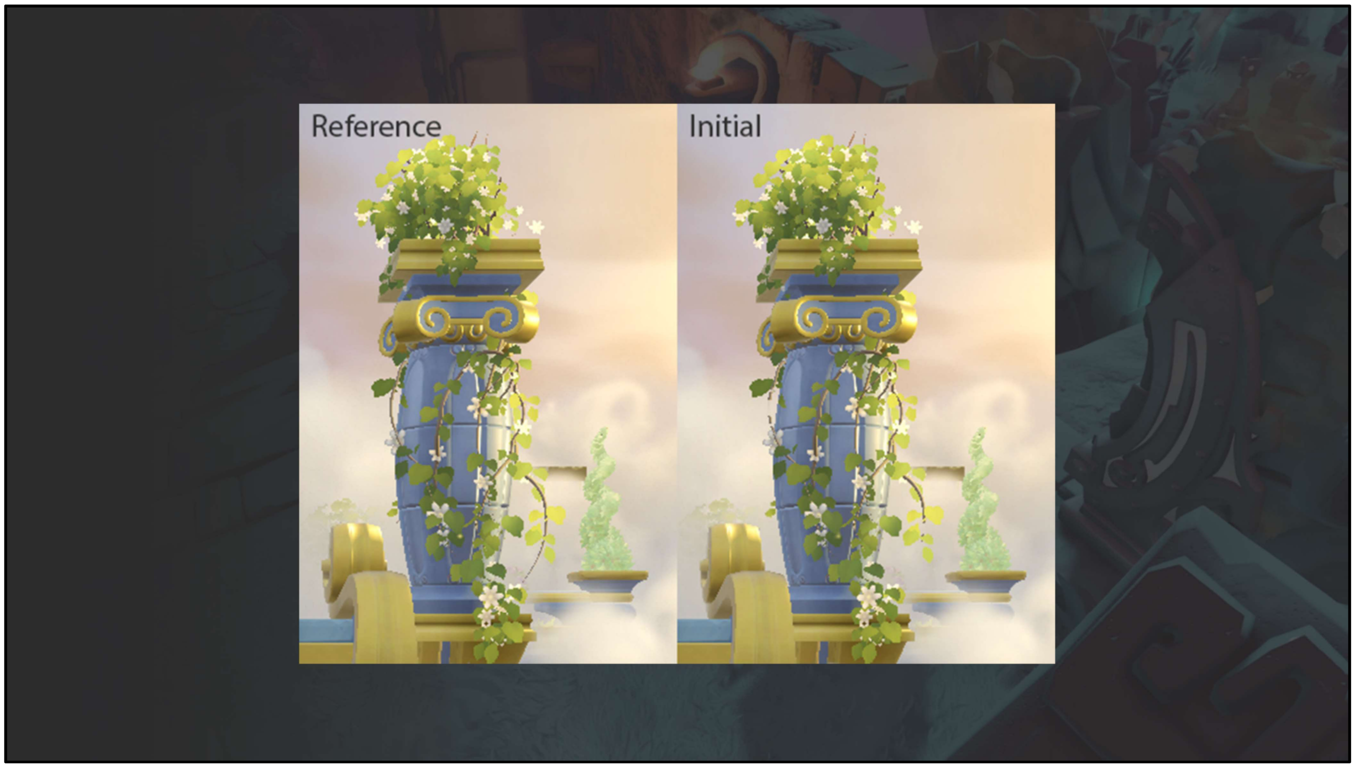
```
float4 vBilinearWeights = GetBilinearWeights(vTexCoord);

float4 vSampleDepths = GetLowResolutionDepths(vTexcoord);
float vPixelDepth = GetHighResolutionDepth(vTexCoord);
float4 vDepthWeights = GetDepthSimilarity(vPixelDepth, vSampleDepths);

return vDepthWeights * vBilateralWeights;
```

The bilateral upsample essentially takes a standard bilinear filter and augments sample weights based on the similarity of the high resolution depth with the depths associated at each low resolution sample.



This is the opening shot for one of our levels where the clouds act as a fog of war for the player, obscuring the path forward and hiding secrets.

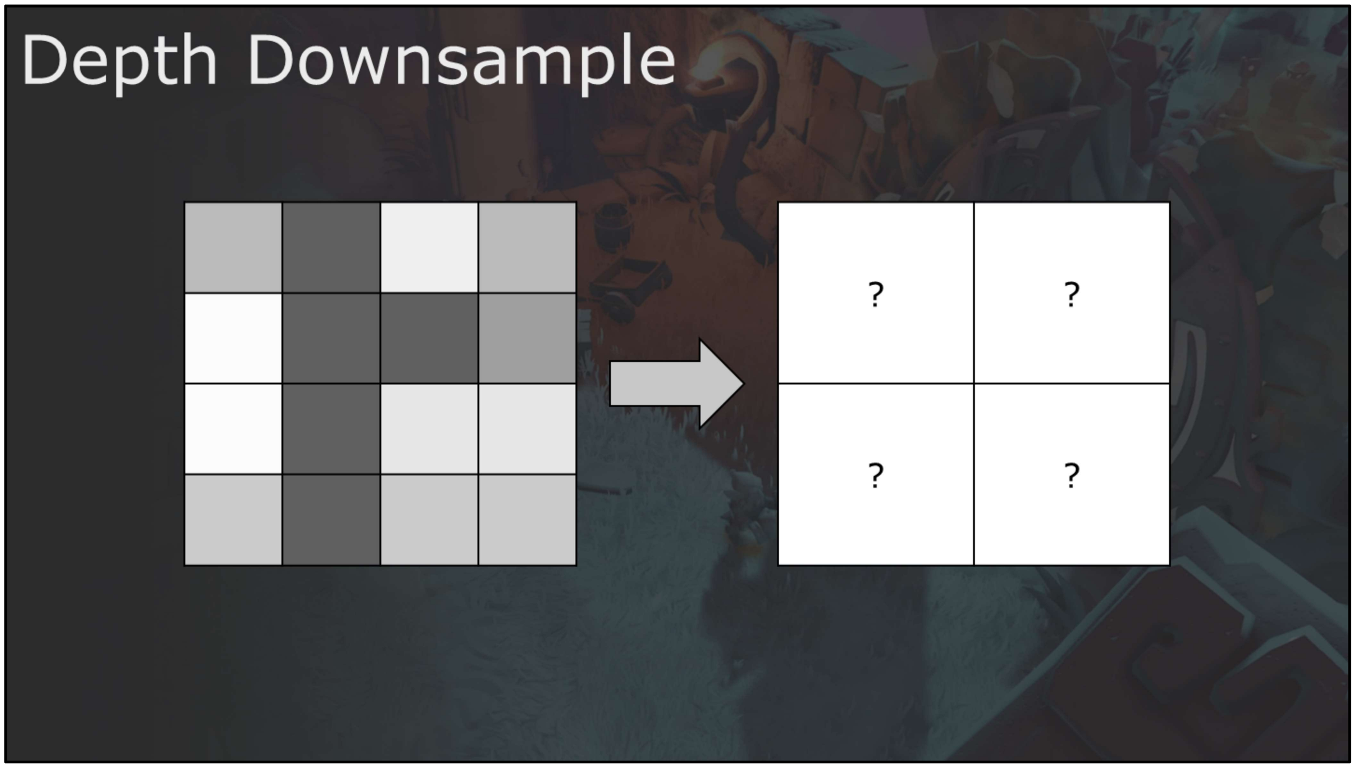
As you can see there is a lot of clouds in the shot and there is a decent sense of atmosphere because of that.



So with the initial implementation that I described you can see in this image that we are getting some artifacts

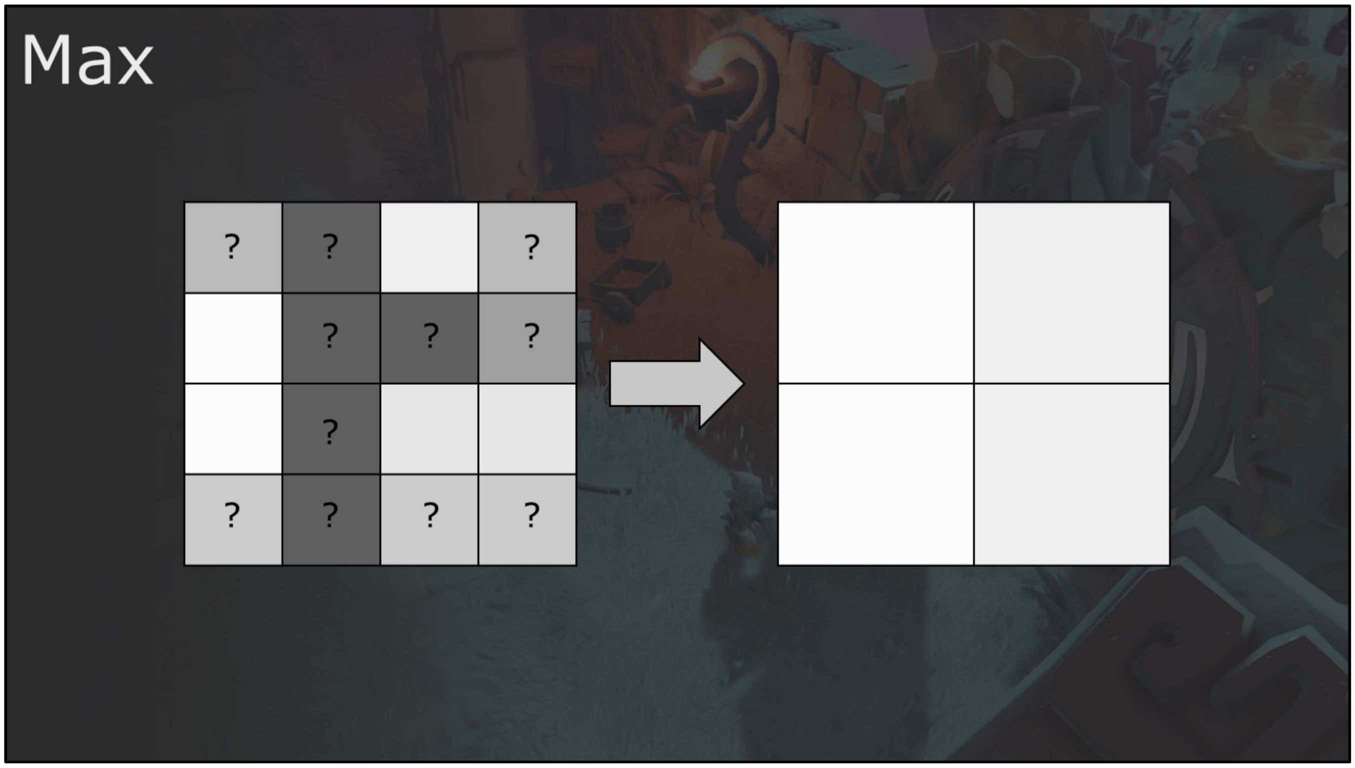
For instance we are loosing some of the think features in this model

So why is that?



Depth Downsample

Well the first part of this has to do with the depth downsample. A few different methods are recommended in the various posts and papers so we decided that we would look at them to see which worked best.

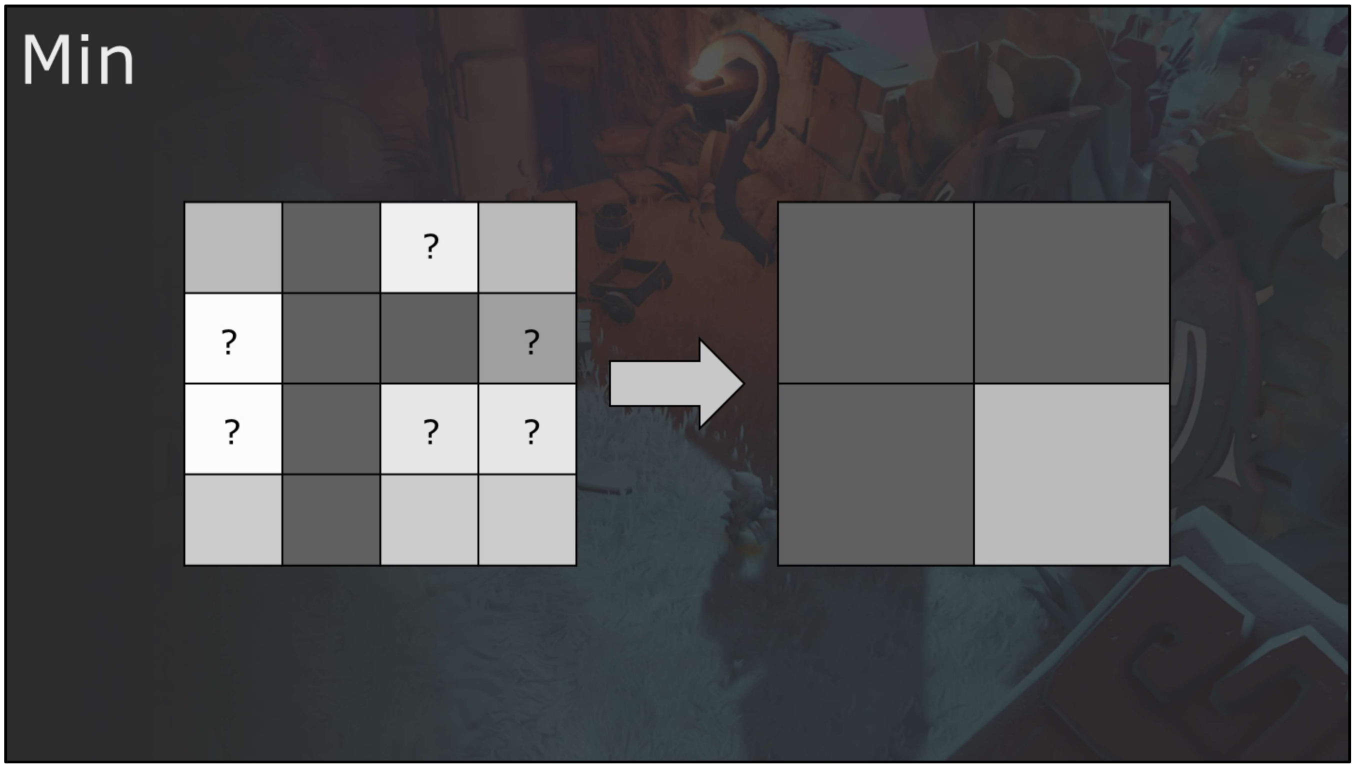


Max

Taking the furthest depth was the first thing that we started with, this shows what the result would be on the right hand side of the screen and what pixels would be left with low quality data on the left.

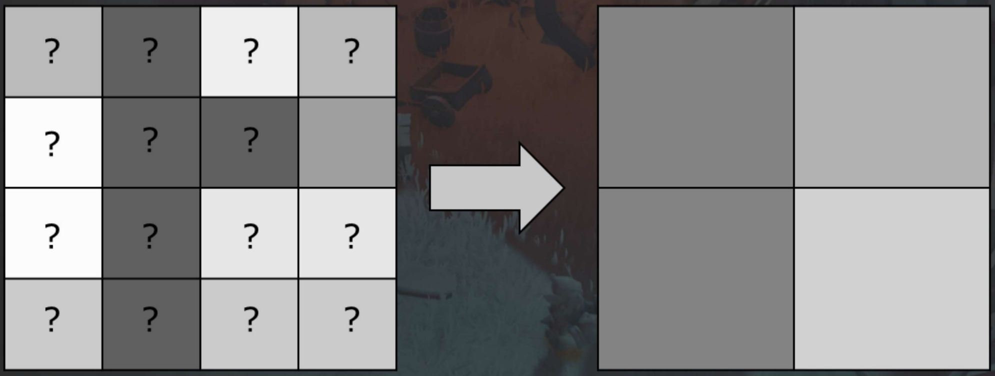
You can see that many of the depth values for the high resolution pixels are not well represented.

Min

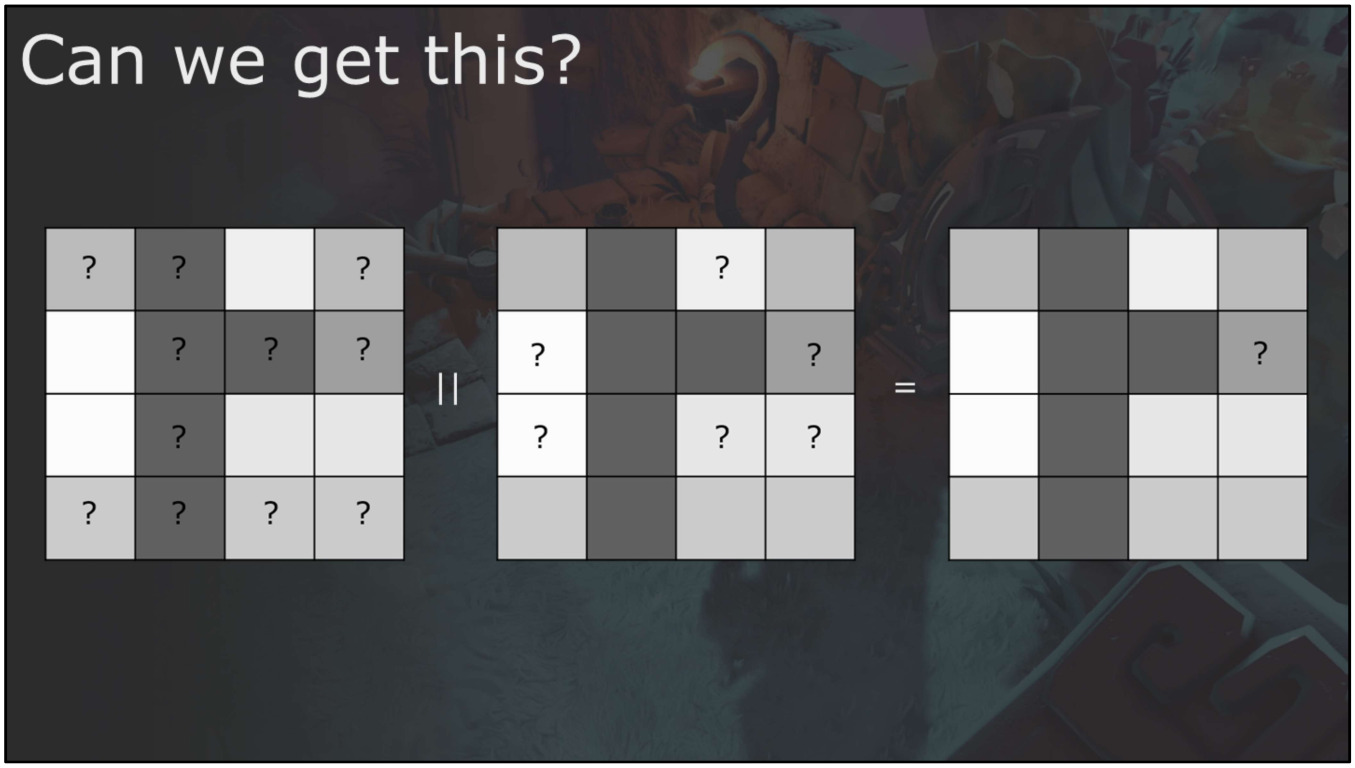


Similarly with the min depth we have a decent amount of pixels that are not represented

Average

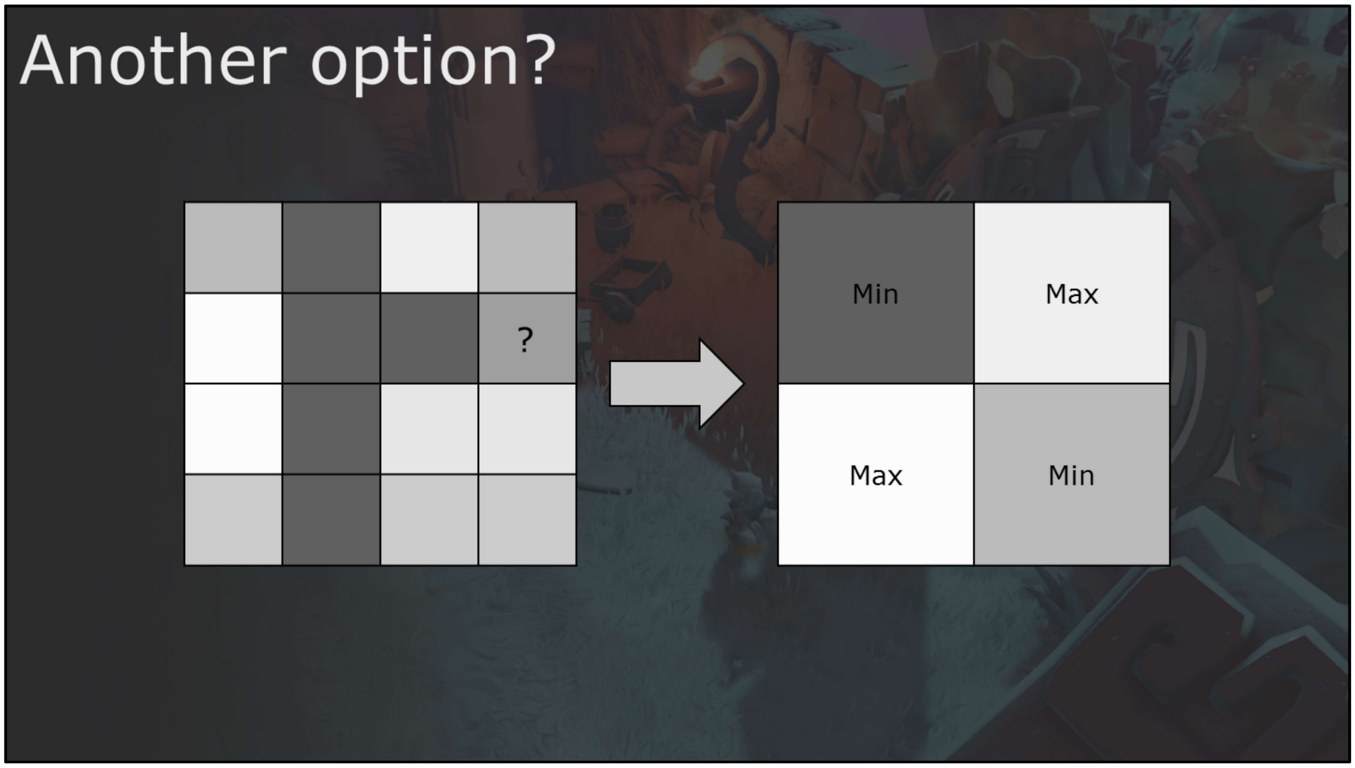


Turns out that average is actually the worst performing method, at depth discontinuities you end up creating new depth values that don’t represent any high resolution pixels well.



Can we get this?

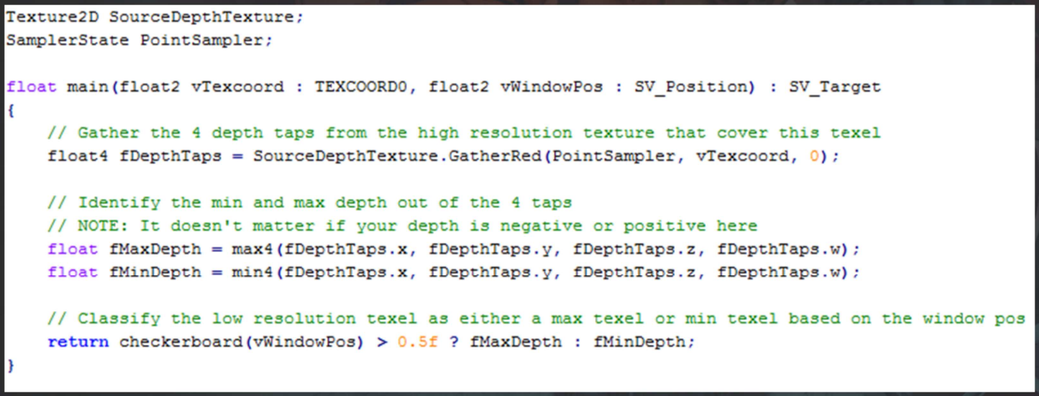
So looking back it would be great if we could combine min and max because the pixels they handled were nearly mutually exclusive.



Another option?

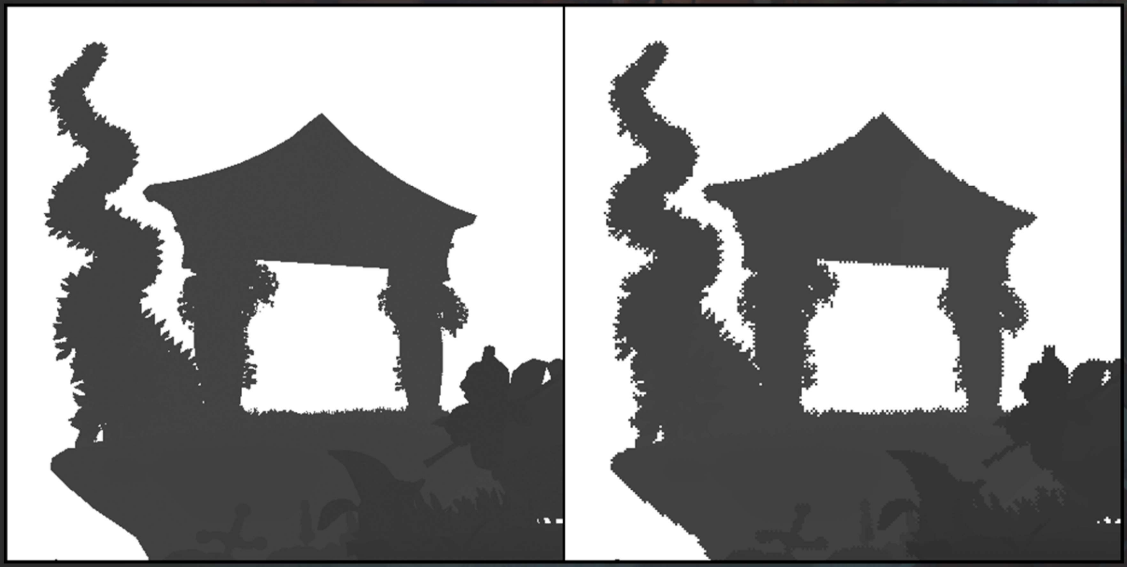
Turns out the simple thing can work some times. Here we just alternate the min and the max in a checkerboard pattern, this gives the total 4x4 blocks of pixels good depth representation.

Code reference



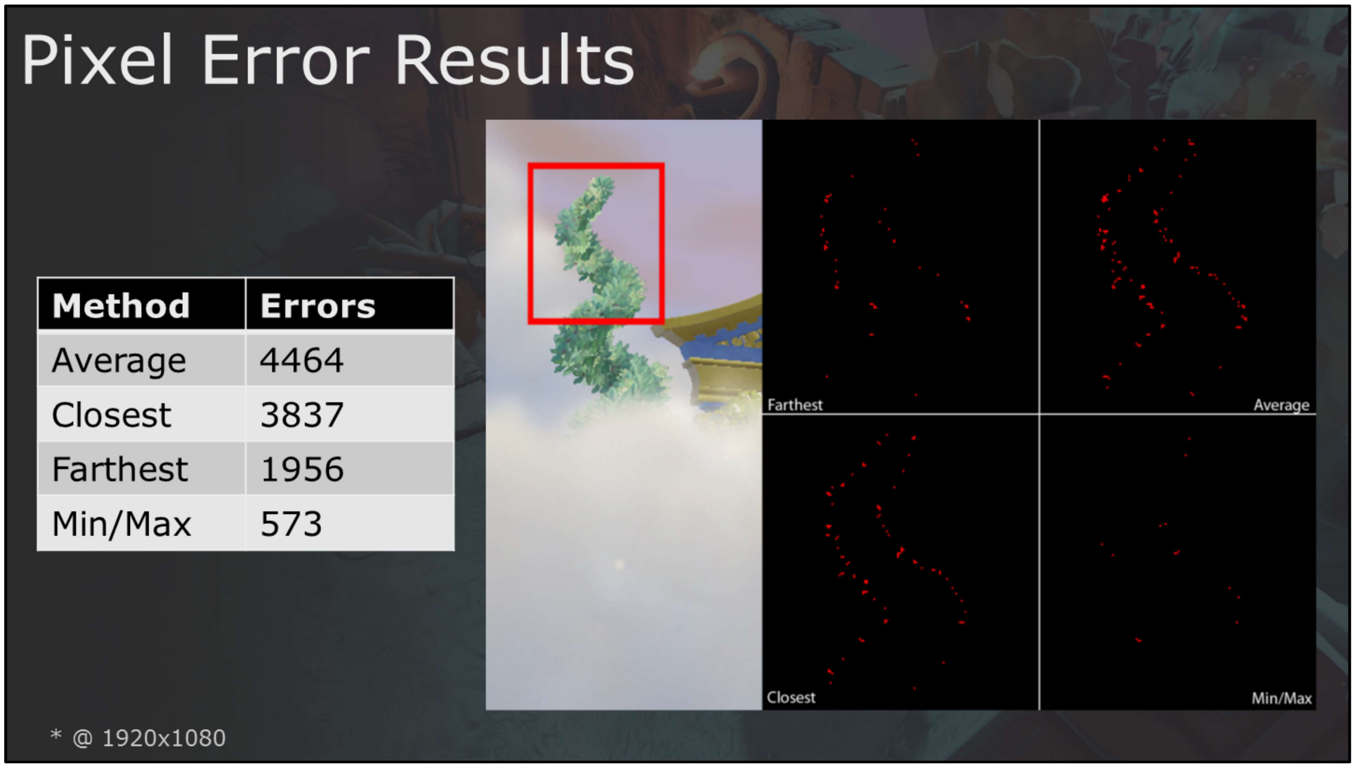
```
Texture2D SourceDepthTexture;
SamplerState PointSampler;

float main(float2 vTexcoord : TEXCOORD0, float2 vWindowPos : SV_Position) : SV_Target
{
    // Gather the 4 depth taps from the high resolution texture that cover this texel
    float4 fDepthTaps = SourceDepthTexture.GatherRed(PointSampler, vTexcoord, 0);

    // Identify the min and max depth out of the 4 taps
    // NOTE: It doesn't matter if your depth is negative or positive here
    float fMaxDepth = max4(fDepthTaps.x, fDepthTaps.y, fDepthTaps.z, fDepthTaps.w);
    float fMinDepth = min4(fDepthTaps.x, fDepthTaps.y, fDepthTaps.z, fDepthTaps.w);

    // Classify the low resolution texel as either a max texel or min texel based on the window pos
    return checkerboard(vWindowPos) > 0.5f ? fMaxDepth : fMinDepth;
}
```

Min/Max in 2x2 block



The resulting depth buffer from this turns high frequency depth discontinuities into a checkerboard pattern as you can see from the grass blades under the archway.

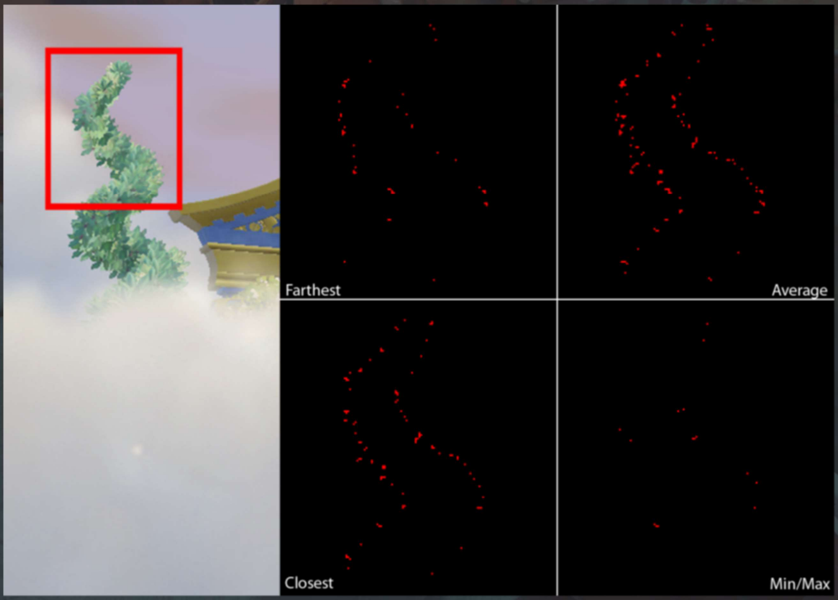
So with this implemented we needed a way of evaluating them. We looked a lot at just back and forth but wanted to formalize it a little more.



Pixel Error Results

Method	Errors
Average	4464
Closest	3837
Farthest	1956
Min/Max	573

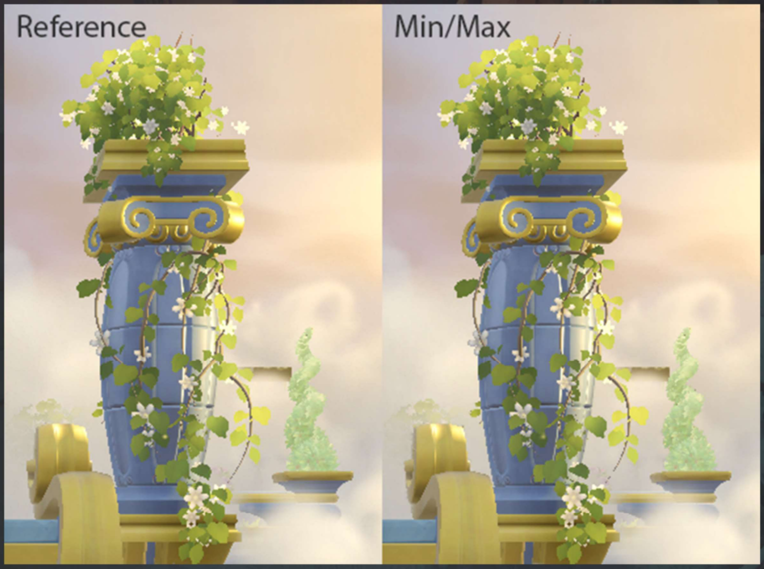
* @ 1920x1080



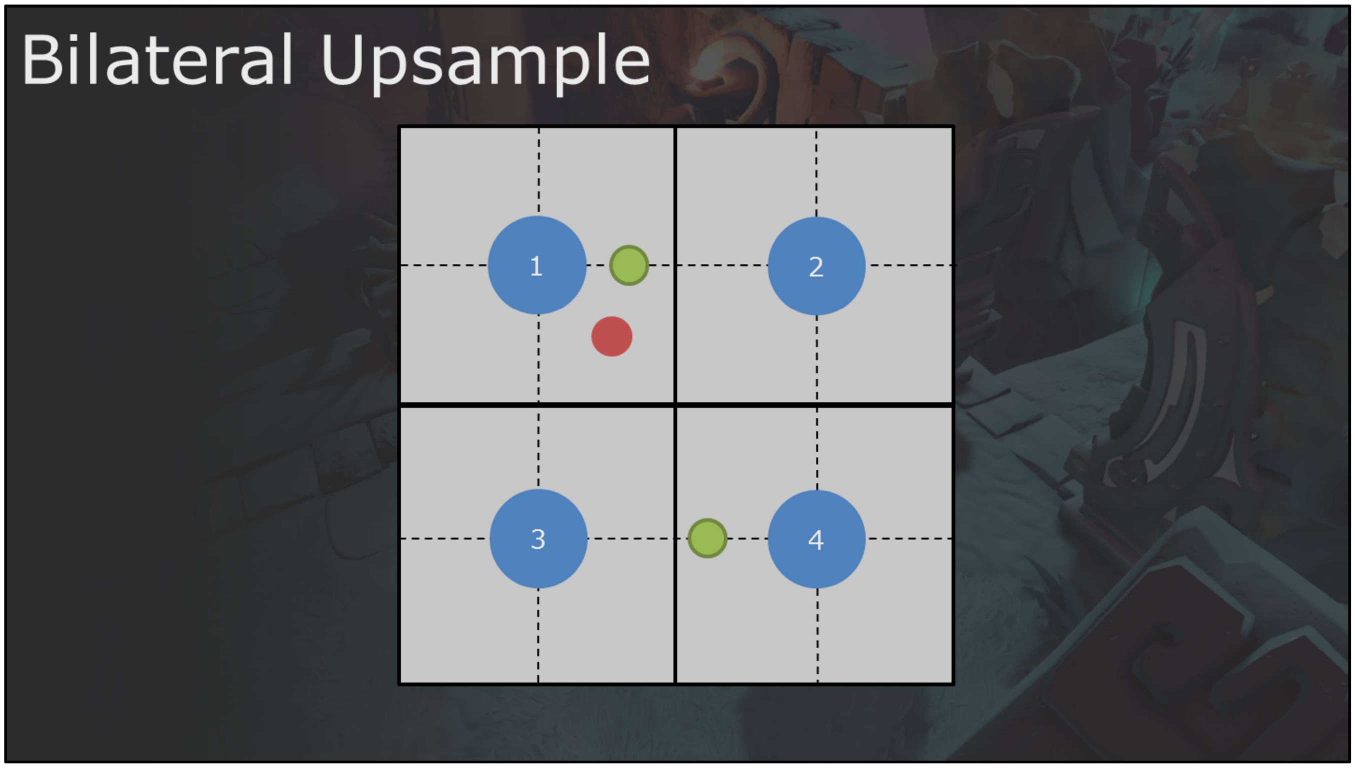
We did a quick debug visualization where we highlighted all the pixels that had low total weights as red.

We scraped this data from captures and accumulated the number of pixels that are considered to be errors into the table on the left.

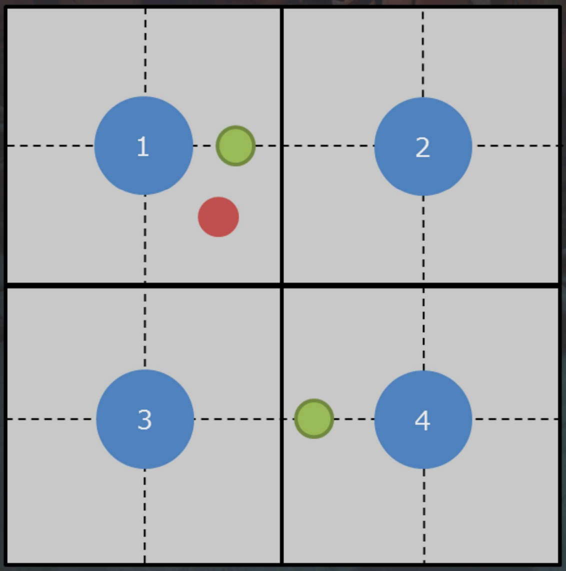
As you can see the min/max method produced the fewest error pixels in this scenario.



Here is the final comparison between the full resolution solution on the left and the min/max half resolution on the right.



Bilateral Upsample



We investigated the upsample for quite a while but we didn't find anything that had better performance characteristics for the quality.

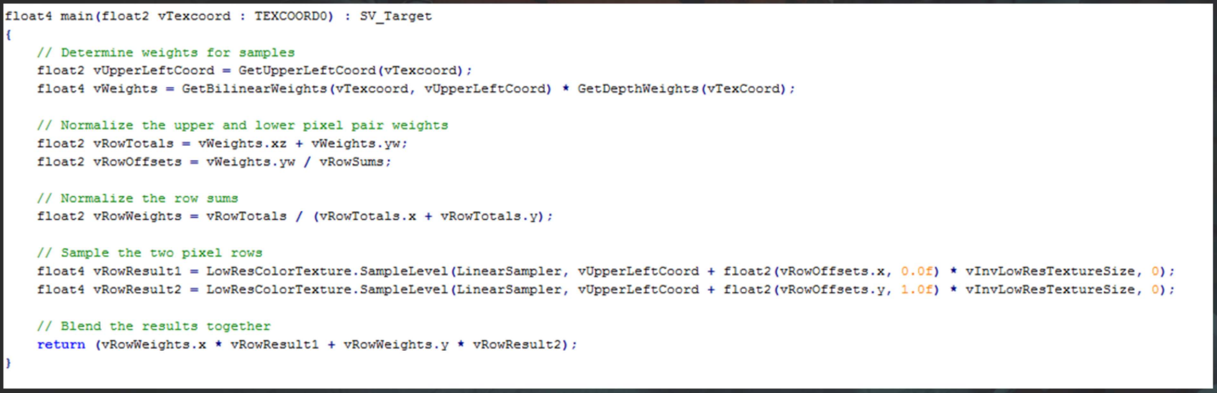
We did re-structure the code a bit which saved us a few texture fetches in the upsample shader.

The basic idea is that instead of using 4 point filtered taps of the low resolution depth texture we took two linearly sampled taps between 1 and 2 and 3 and 4 respectively.

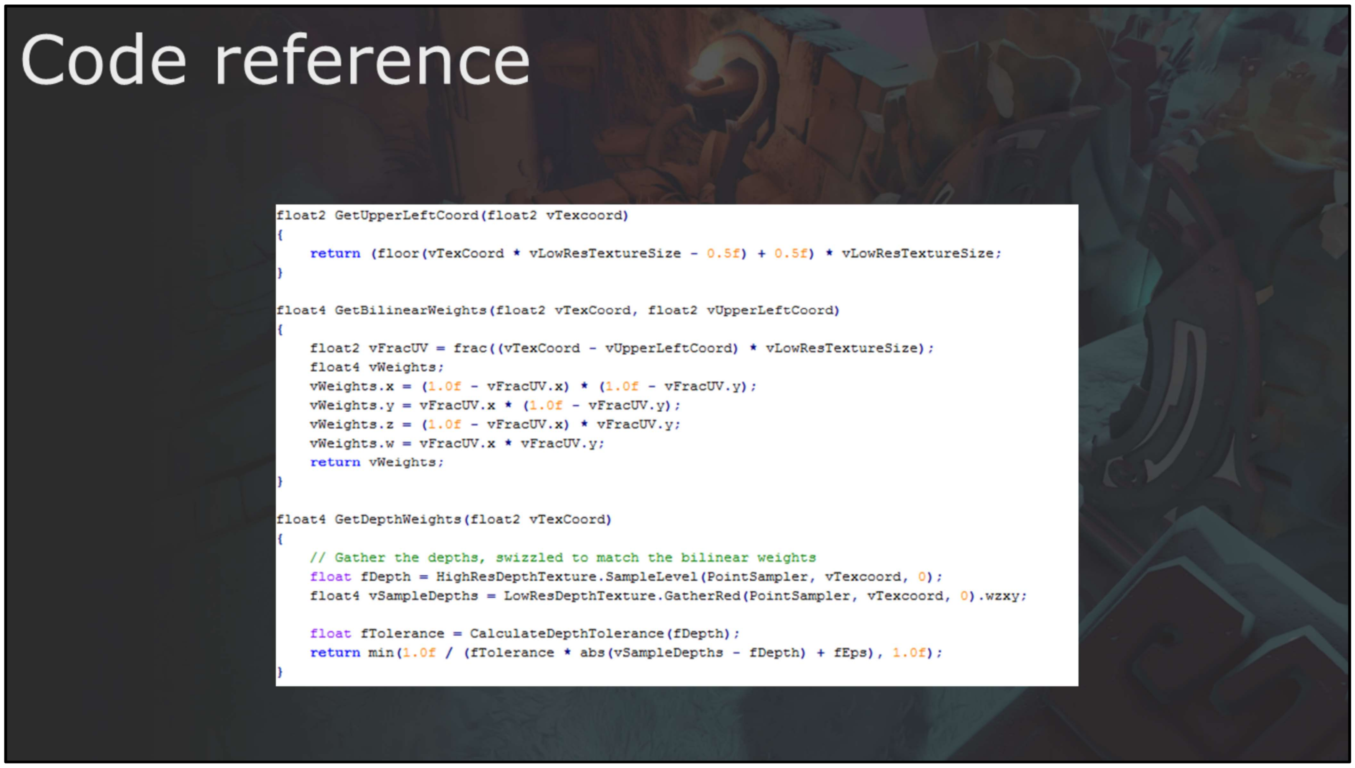
Then these samples are blended together, this lets us represent any combination of weights like the full bilateral upsample but with fewer texture fetch instructions.

So we do all of this and we check it in…. And then we get the first wave of feedback.

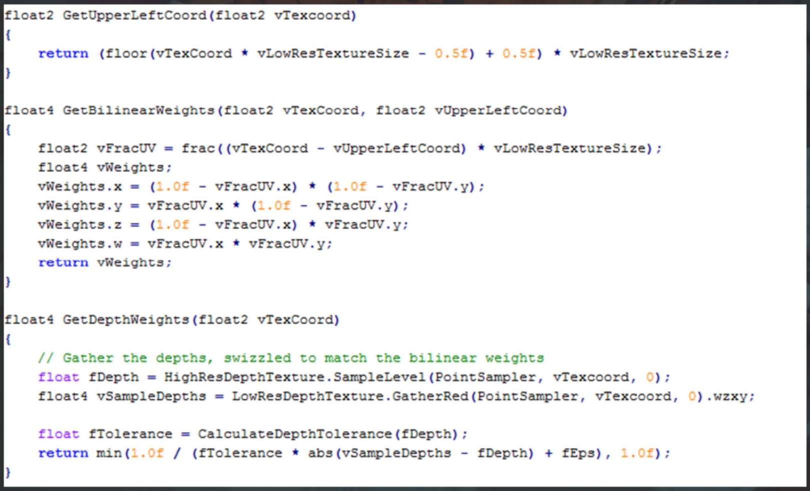
Code reference



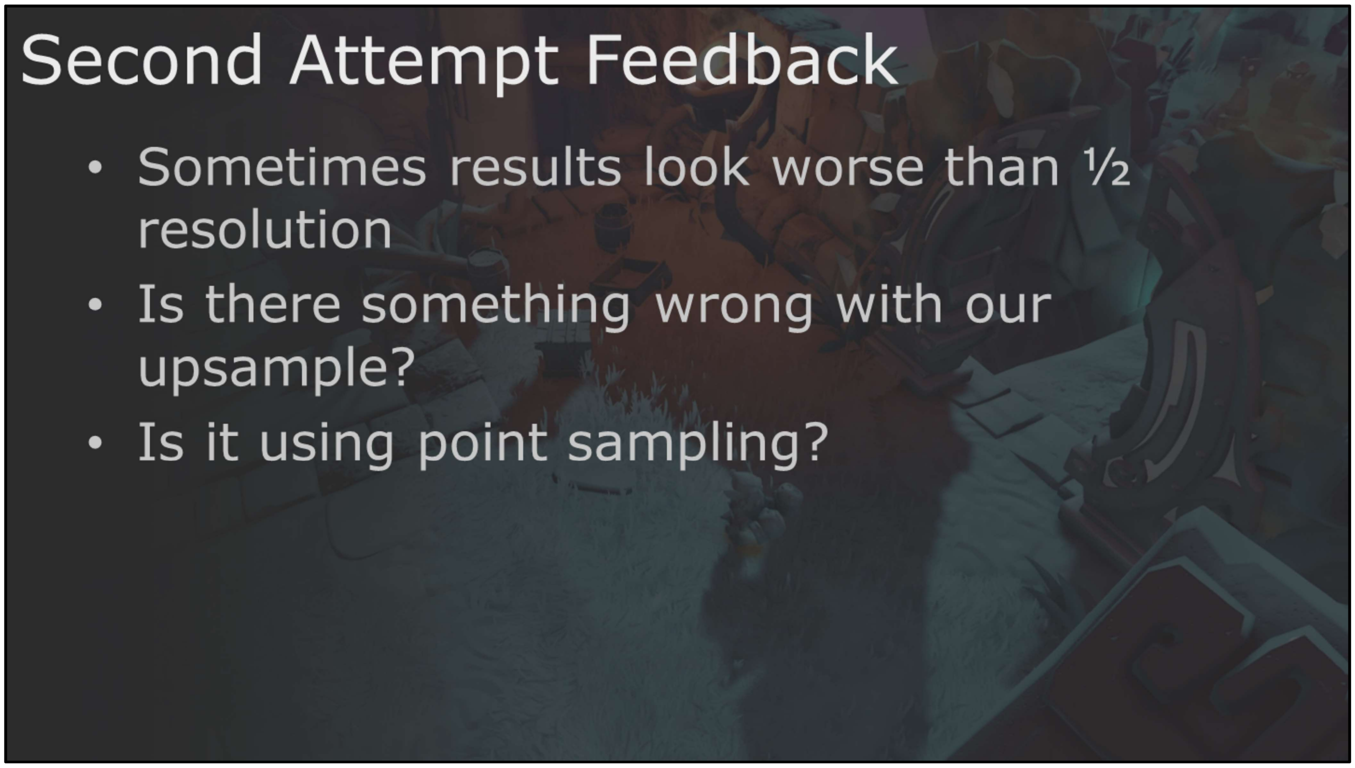
```
float4 main(float2 vTexcoord : TEXCOORD0) : SV_Target
{
    // Determine weights for samples
    float2 vUpperLeftCoord = GetUpperLeftCoord(vTexcoord);
    float4 vWeights = GetBilinearWeights(vTexcoord, vUpperLeftCoord) * GetDepthWeights(vTexCoord);

    // Normalize the upper and lower pixel pair weights
    float2 vRowTotals = vWeights.xz + vWeights.yw;
    float2 vRowOffsets = vWeights.yw / vRowSums;

    // Normalize the row sums
    float2 vRowWeights = vRowTotals / (vRowTotals.x + vRowTotals.y);

    // Sample the two pixel rows
    float4 vRowResult1 = LowResColorTexture.SampleLevel(LinearSampler, vUpperLeftCoord + float2(vRowOffsets.x, 0.0f) * vInvLowResTextureSize, 0);
    float4 vRowResult2 = LowResColorTexture.SampleLevel(LinearSampler, vUpperLeftCoord + float2(vRowOffsets.y, 1.0f) * vInvLowResTextureSize, 0);

    // Blend the results together
    return (vRowWeights.x * vRowResult1 + vRowWeights.y * vRowResult2);
}
```



Code reference

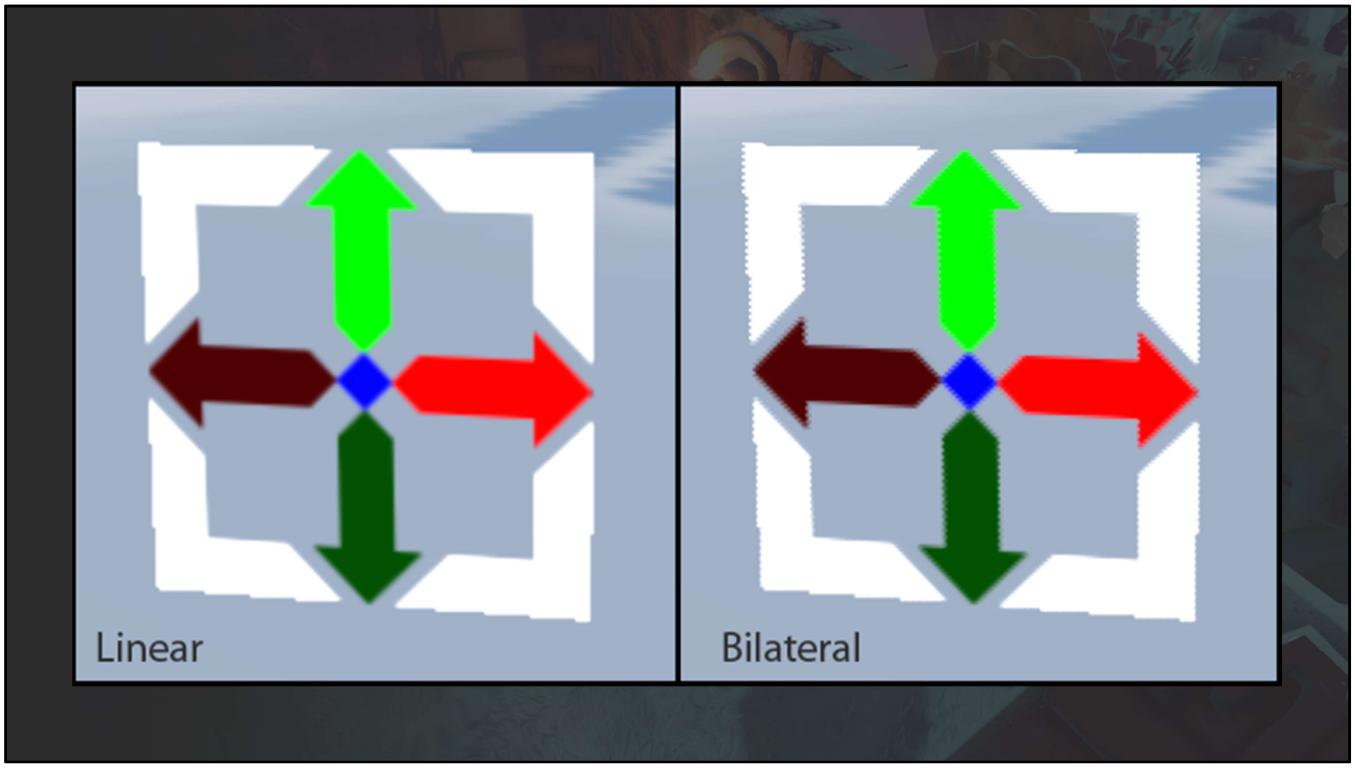


```
float2 GetUpperLeftCoord(float2 vTexcoord)
{
    return (floor(vTexCoord * vLowResTextureSize - 0.5f) + 0.5f) * vLowResTextureSize;
}

float4 GetBilinearWeights(float2 vTexCoord, float2 vUpperLeftCoord)
{
    float2 vFracUV = frac((vTexCoord - vUpperLeftCoord) * vLowResTextureSize);
    float4 vWeights;
    vWeights.x = (1.0f - vFracUV.x) * (1.0f - vFracUV.y);
    vWeights.y = vFracUV.x * (1.0f - vFracUV.y);
    vWeights.z = (1.0f - vFracUV.x) * vFracUV.y;
    vWeights.w = vFracUV.x * vFracUV.y;
    return vWeights;
}

float4 GetDepthWeights(float2 vTexCoord)
{
    // Gather the depths, swizzled to match the bilinear weights
    float fDepth = HighResDepthTexture.SampleLevel(PointSampler, vTexcoord, 0);
    float4 vSampleDepths = LowResDepthTexture.GatherRed(PointSampler, vTexcoord, 0).wzxy;

    float fTolerance = CalculateDepthTolerance(fDepth);
    return min(1.0f / (fTolerance * abs(vSampleDepths - fDepth) + fEps), 1.0f);
}
```



Second Attempt Feedback

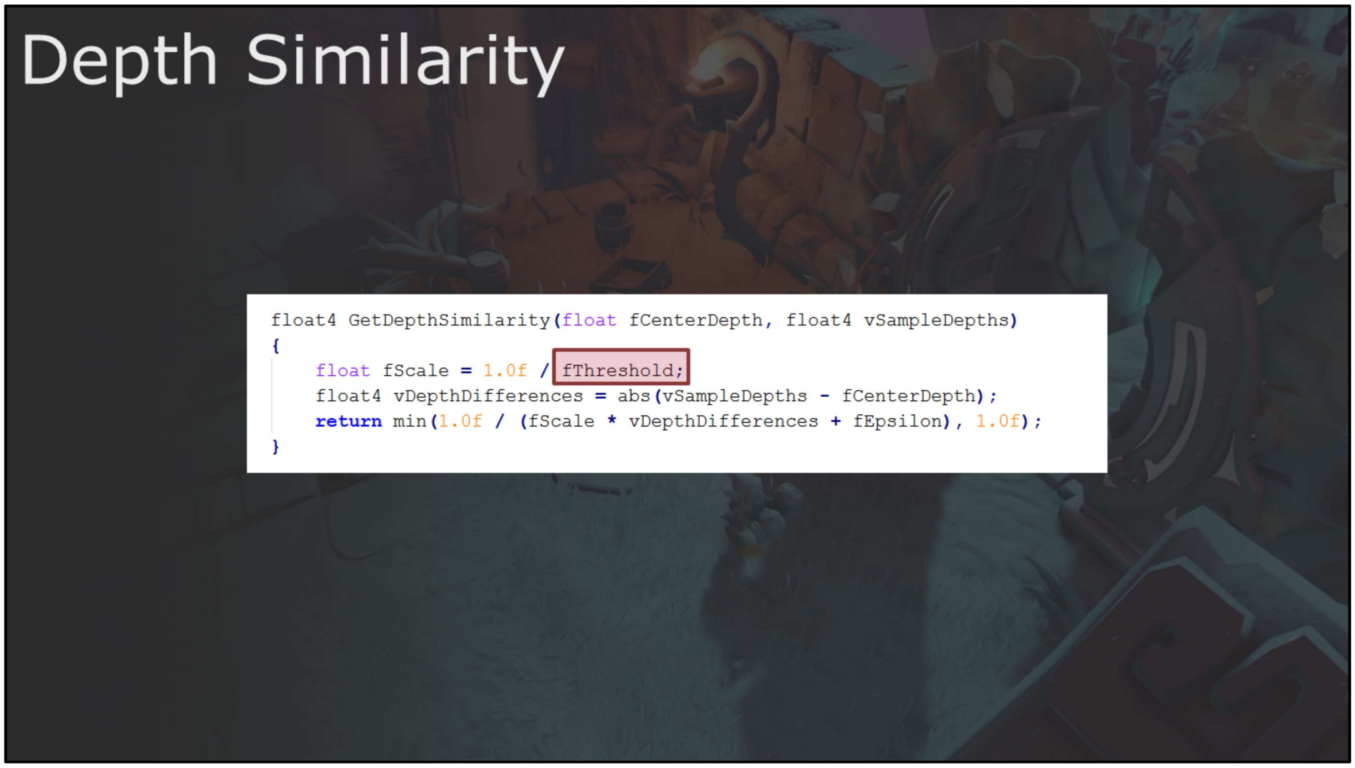
- Sometimes results look worse than ½ resolution
- Is there something wrong with our upsample?
- Is it using point sampling?



Turns out there were some cases where the result had some funky artifacts, when you compared it against a simple linear filtering with out the depth weights you could tell something was going on.

So why is this happening?

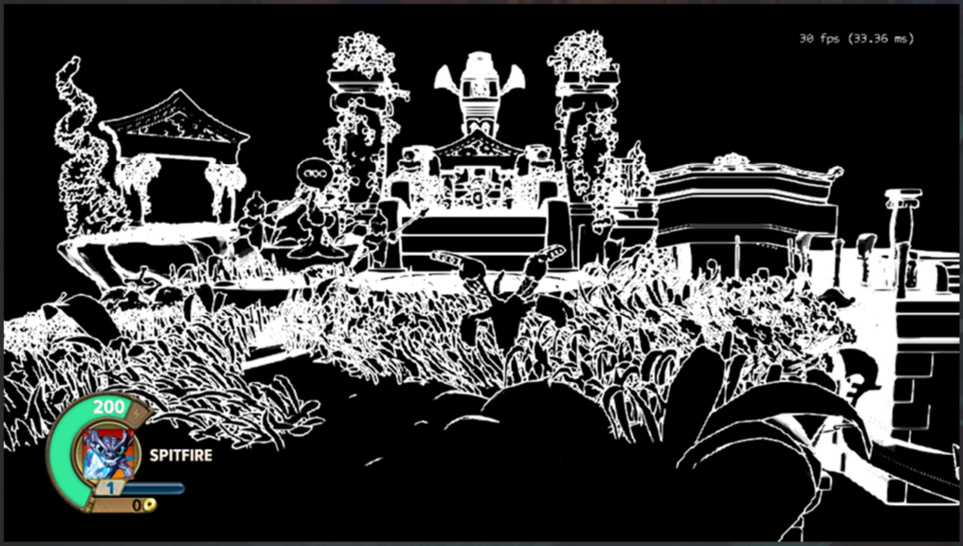
It has to do with the bilateral upsample and how depth similarity is computed.



Depth Similarity

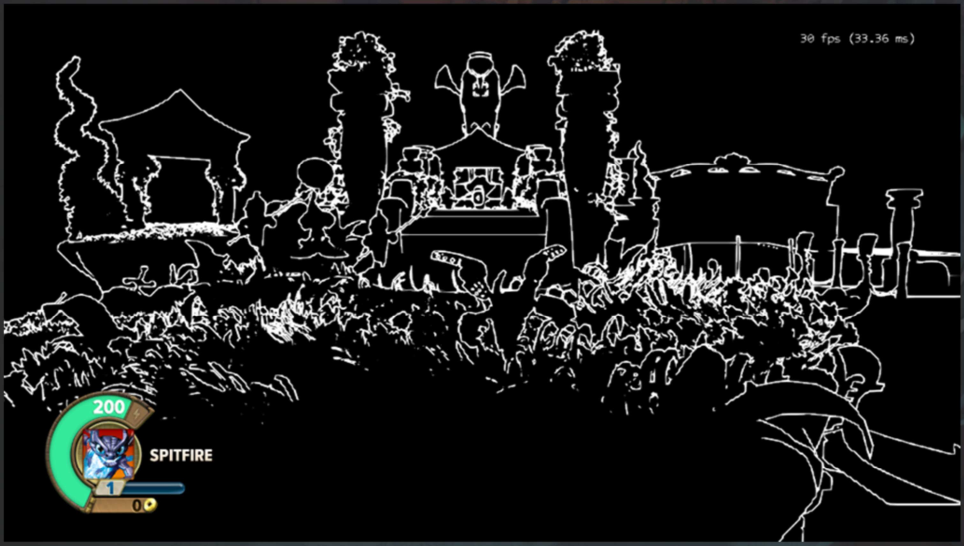
```
float4 GetDepthSimilarity(float fCenterDepth, float4 vSampleDepths)
{
    float fScale = 1.0f / fThreshold;
    float4 vDepthDifferences = abs(vSampleDepths - fCenterDepth);
    return min(1.0f / (fScale * vDepthDifferences + fEpsilon), 1.0f);
}
```

Small threshold

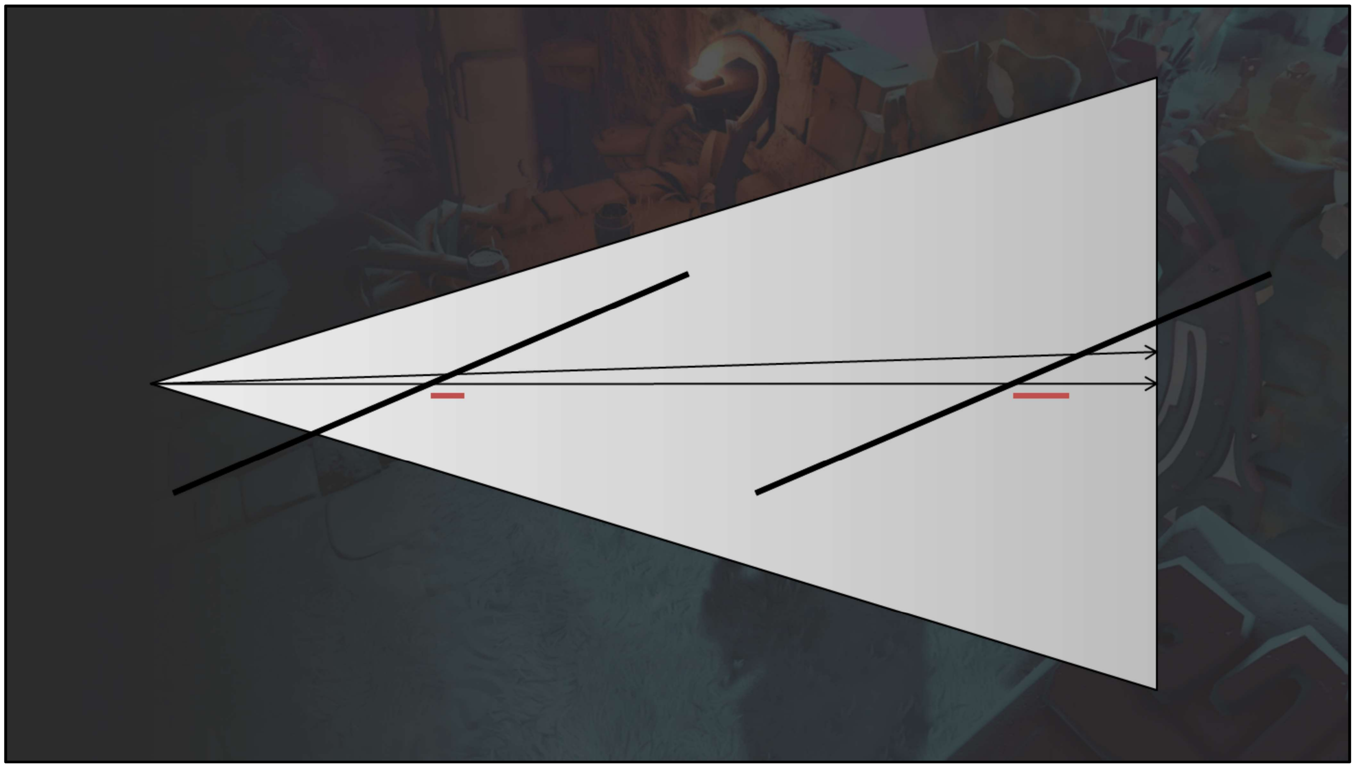


Typically the depth similarity is computed by determining the difference in depth from the high resolution pixel and the low resolution pixels. That number is used along with a threshold to determine if the depth is similar. So if you pick a small threshold you end up detecting false edges.

Large threshold



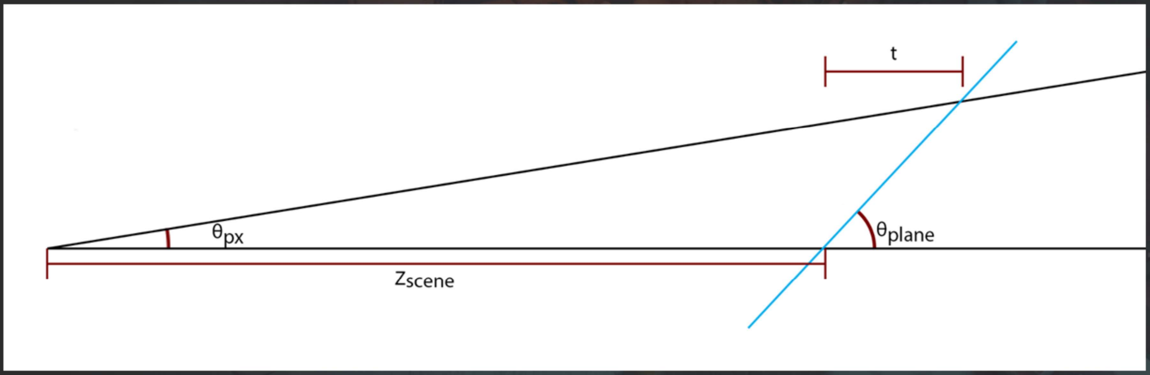
Similarly if your threshold is too large you start to lose edges that are close together. Like the blades of grass in the front of this image.



This was due to the fact that we were using a fixed depth bias value for the bilateral weighting. The problem was that as the surface receded into the screen the number of units that a single pixel step represented increased.

We decided to have a threshold value that was based on the depth because of this.

Target slope threshold



$$g(z) = z \cdot \boxed{2} \cdot \left(\left(\frac{\cos(\theta_{px}) \cdot \sin(\theta_{plane})}{\sin(\theta_{plane} - \theta_{px})} \right) - 1 \right)$$

Max pixel distance in neighborhood

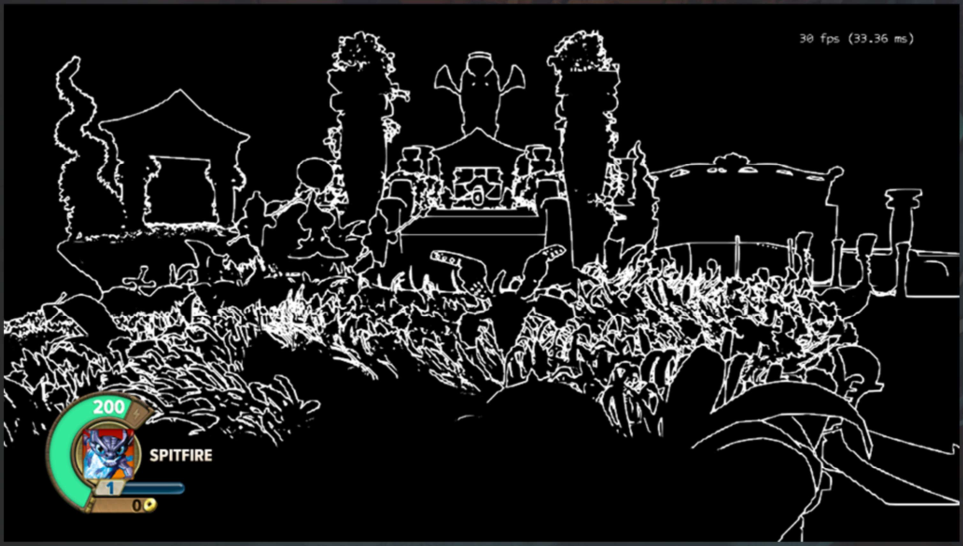
We actually used the model of a plane being pushed back into the scene as our basis for determining the threshold.

As an input to this we give the angular difference of a single pixel and the slope of the plane that we want to ensure that we maintain linear blending in front of.

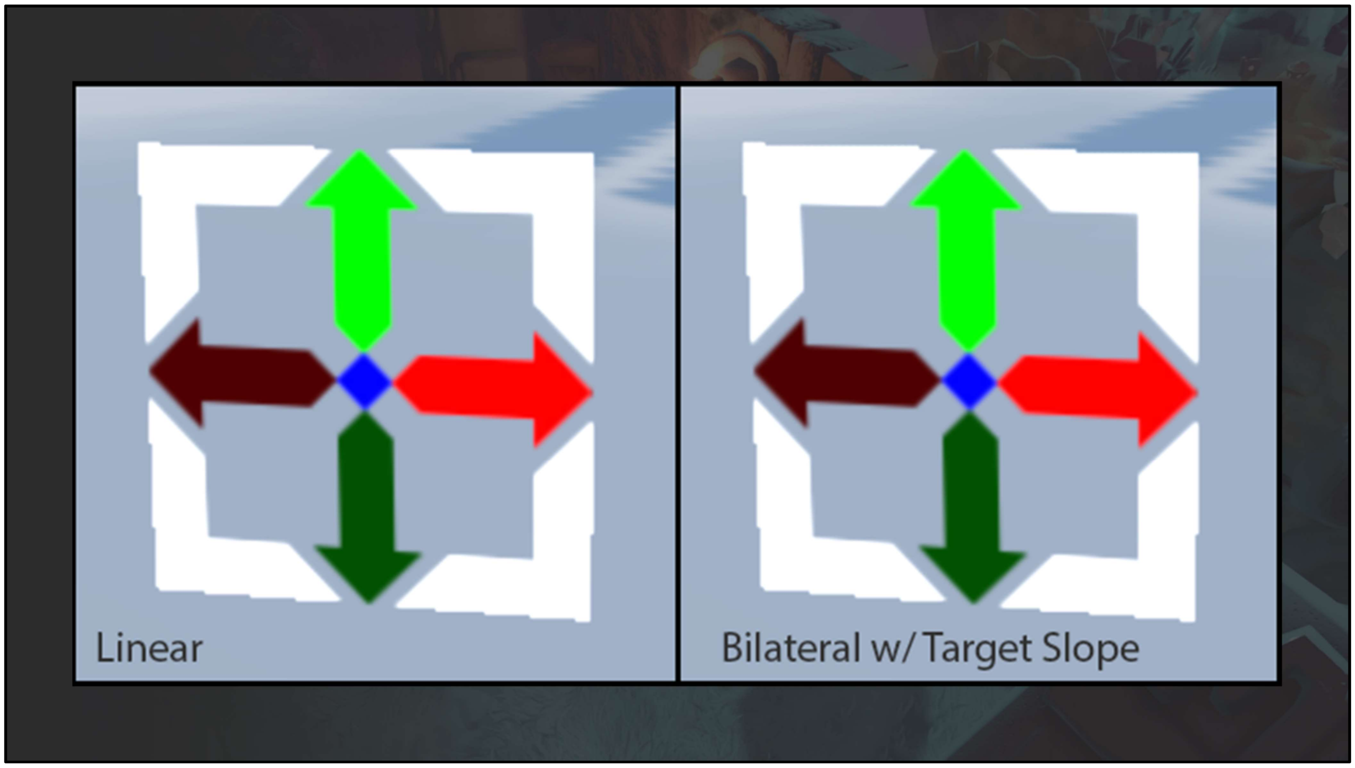
This turns it into a simple trig problem and you get this result.

We scale that value to compensate for the fact that when we downsample the depth value may come from up to 2 pixels away.

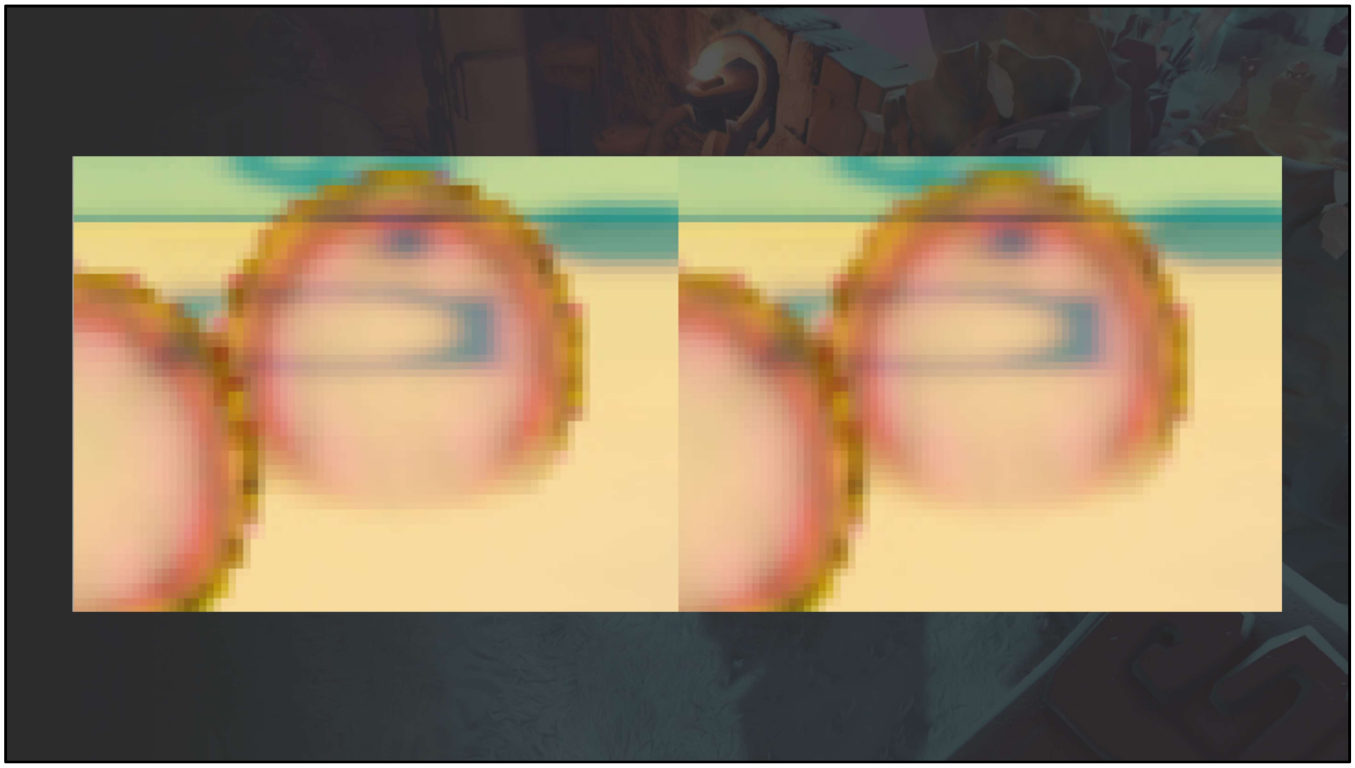
Target slope threshold



Here is the scene using the target slope threshold

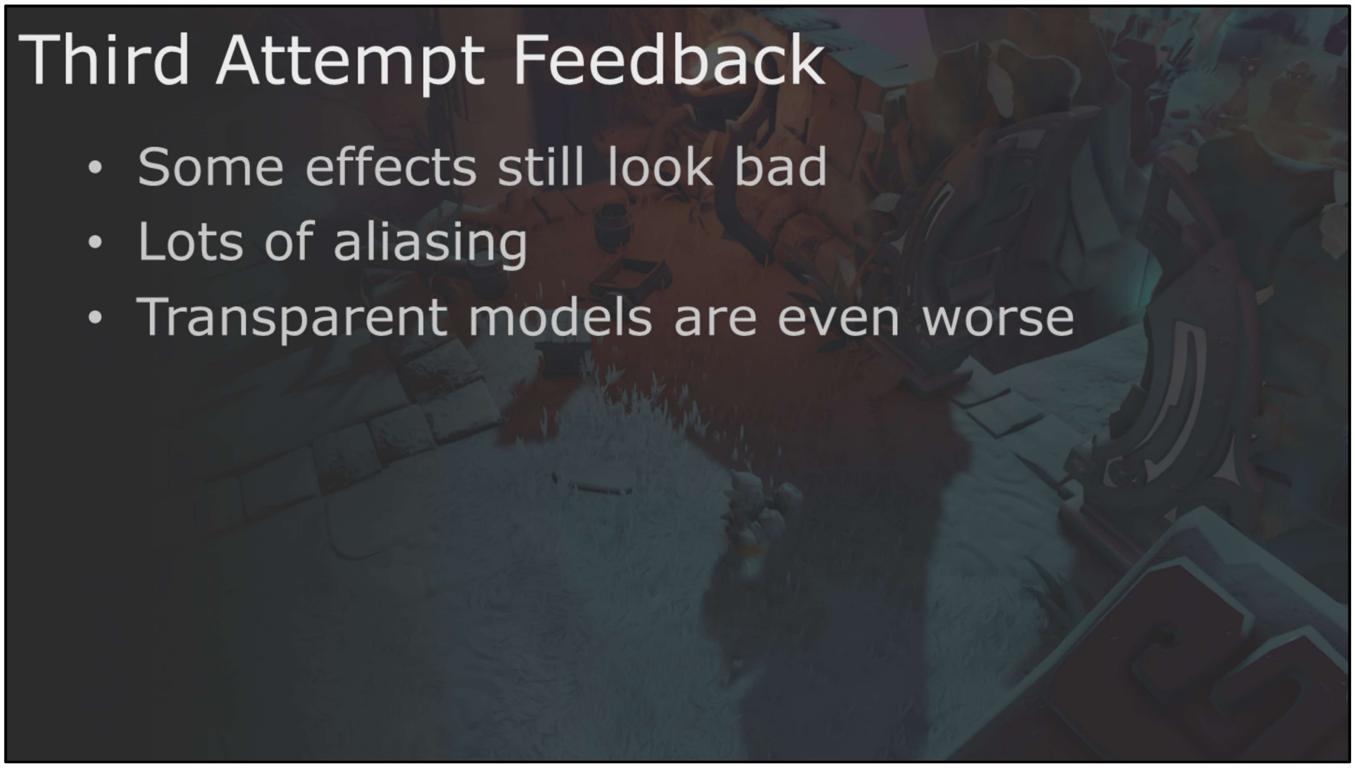


Going back to the original example you can see that it resolved the serrated edge problem we were seeing before.



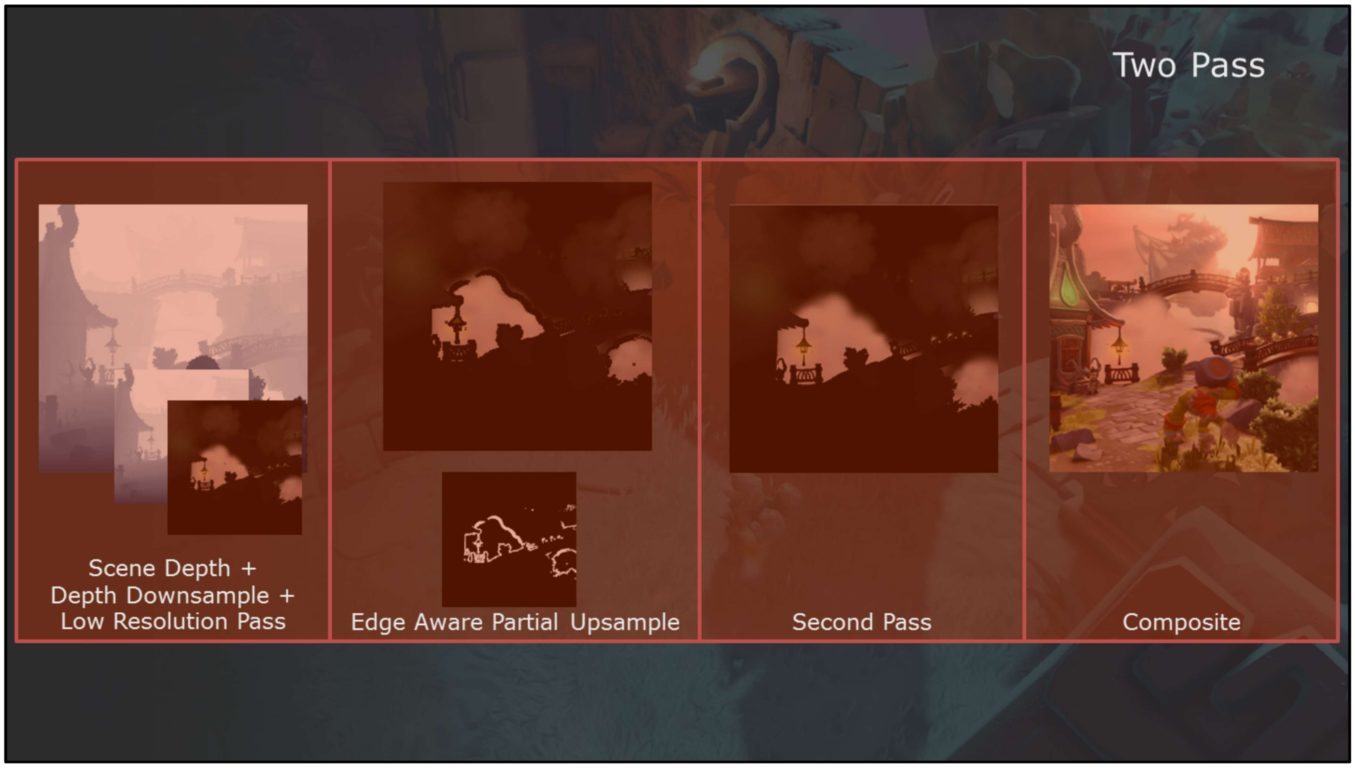
Here is another example of this problem before and after using the target slope threshold

You can see on the left that we are getting filtering horizontally but not vertically.



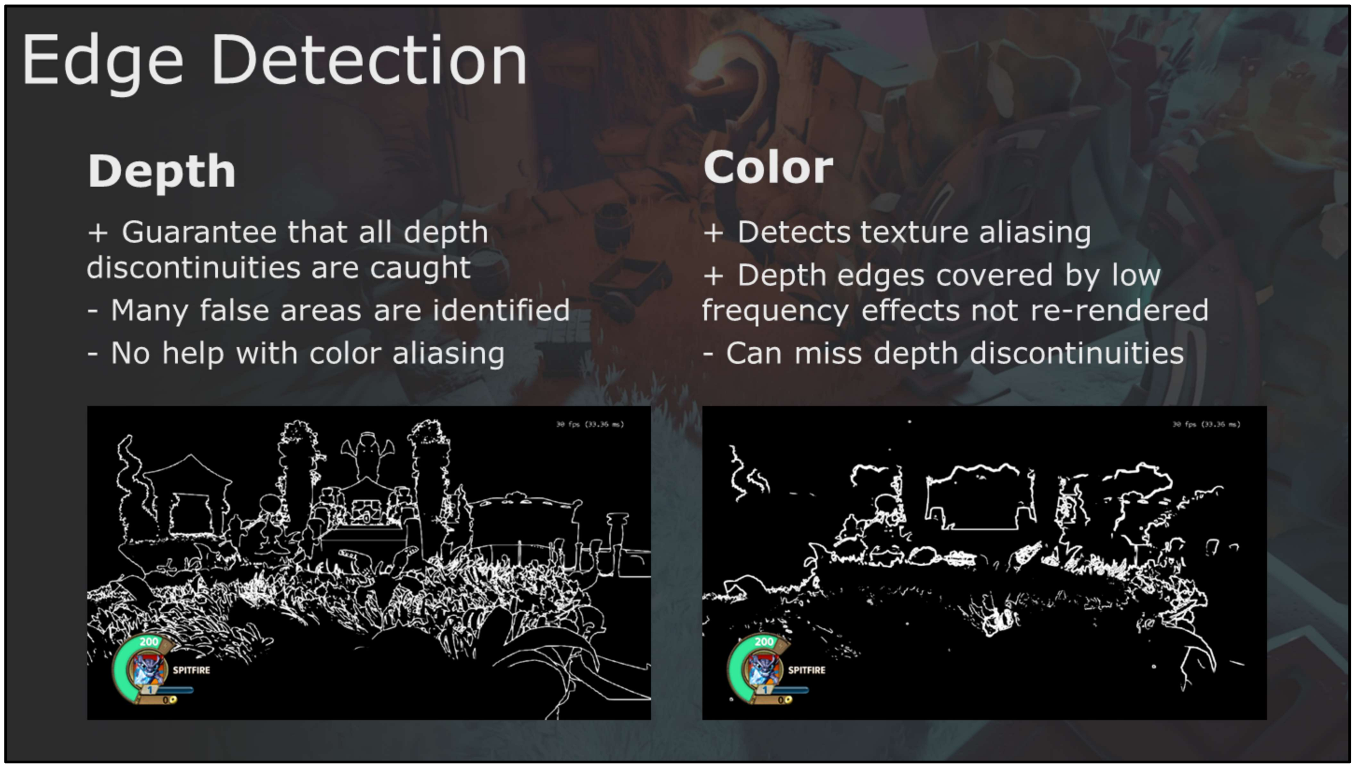
Third Attempt Feedback

- Some effects still look bad
- Lots of aliasing
- Transparent models are even worse



Two Pass

To mitigate edge aliasing we moved to a two pass approach



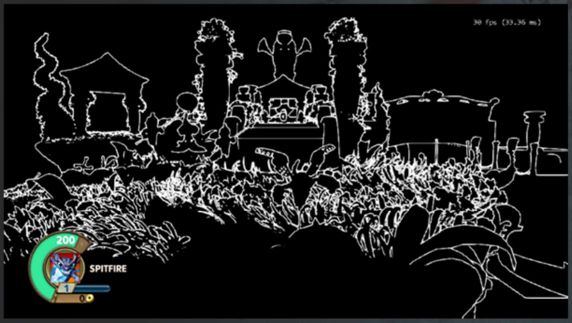
Edge Detection

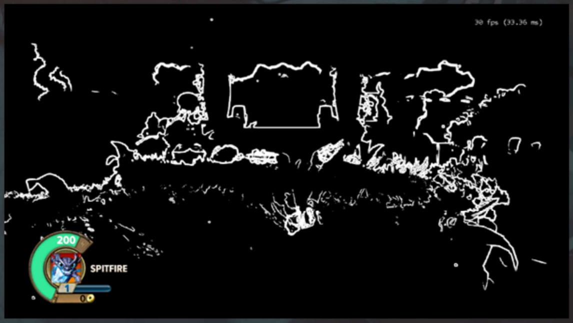
Depth

+ Guarantee that all depth discontinuities are caught
- Many false areas are identified
- No help with color aliasing

Color

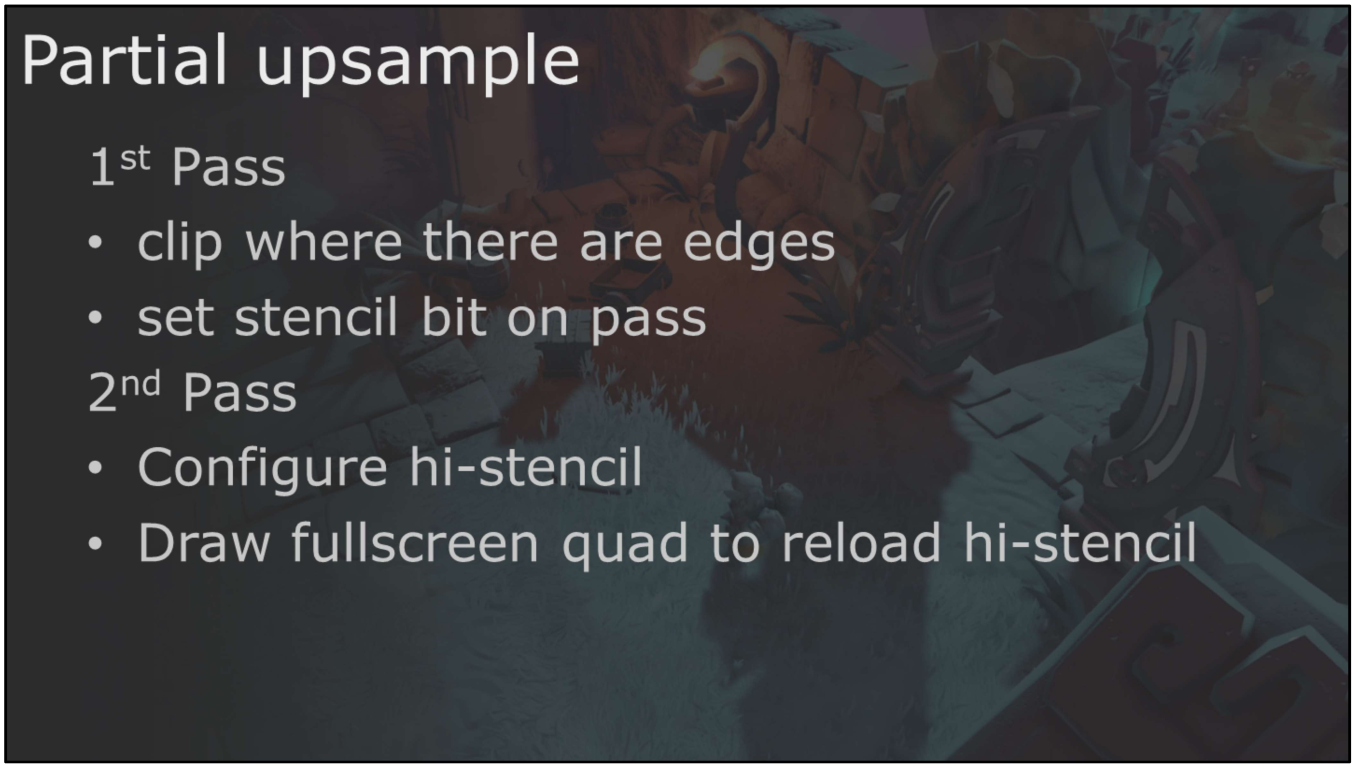
+ Detects texture aliasing
+ Depth edges covered by low frequency effects not re-rendered
- Can miss depth discontinuities





In order to benefit from the fact that we are rendering low frequency effects like clouds into the offscreen buffer we use a color buffer to detect edges as suggested in the references.

This results in fewer edges than testing depth and addresses texture and raster aliasing in the image.



Partial upsample

1st Pass

- clip where there are edges
- set stencil bit on pass

2nd Pass

- Configure hi-stencil
- Draw fullscreen quad to reload hi-stencil

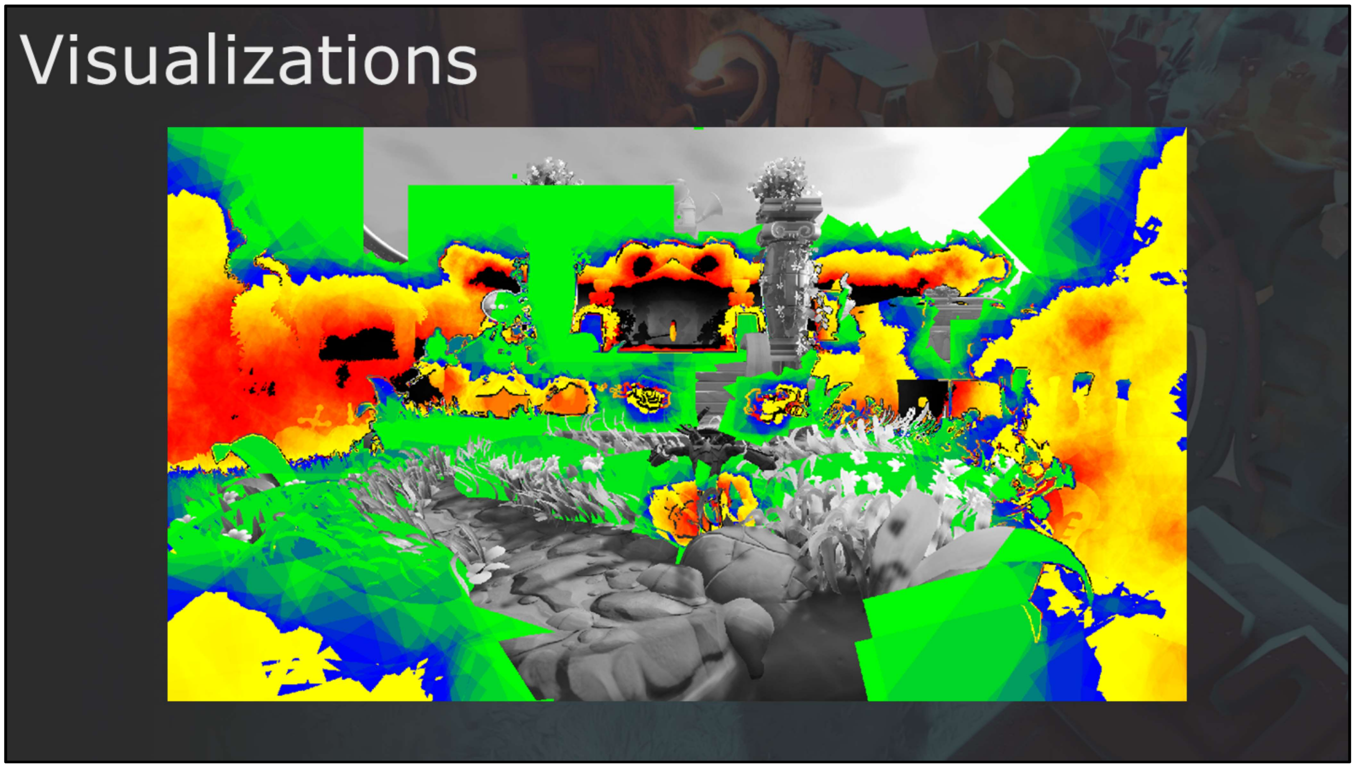
Fourth attempt feedback

- Quality is shippable
- Does it still save us time?

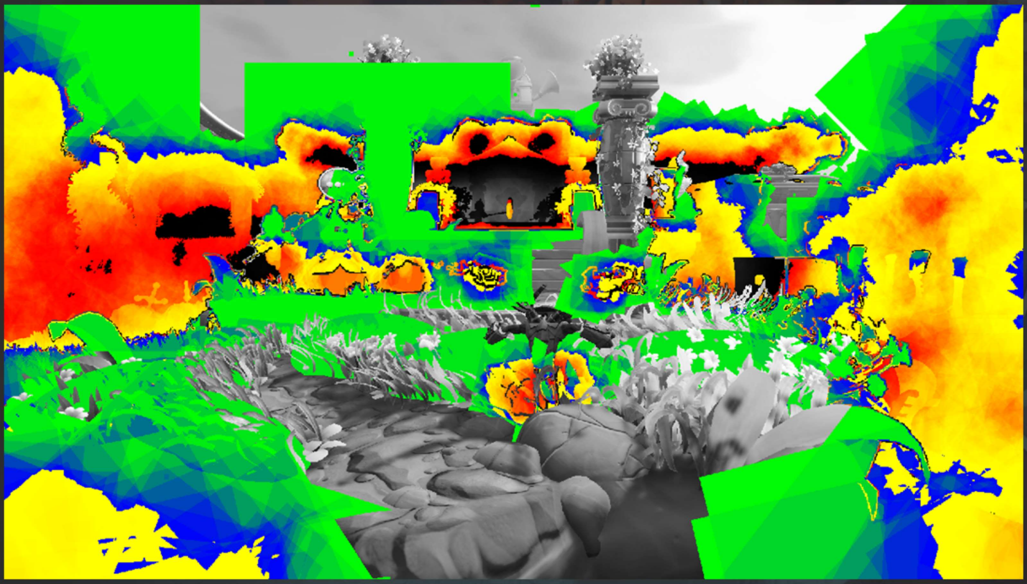
RESULTS

360 Timing

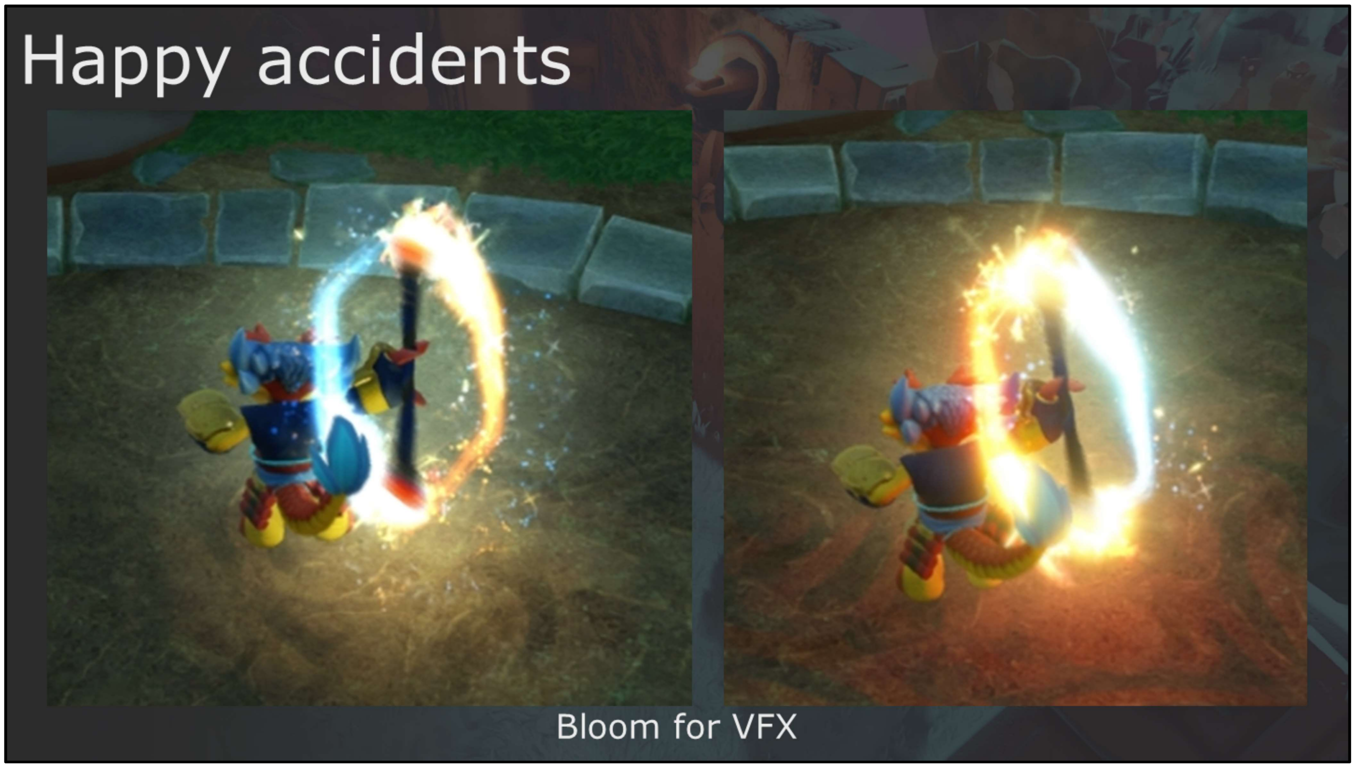
Pass	Mixed	Full
Downsample	0.2	-
Low Res Pass	1.8	-
Edge Detect	0.3	-
Upsample	1.0	-
Hi-S Reload	0.2	-
High Res Pass	1.6	6.9
Total	4.9	6.9



Visualizations



Visualizations were still important to ensure that the artists knew how this two pass approach was adding to the cost of their scenes.

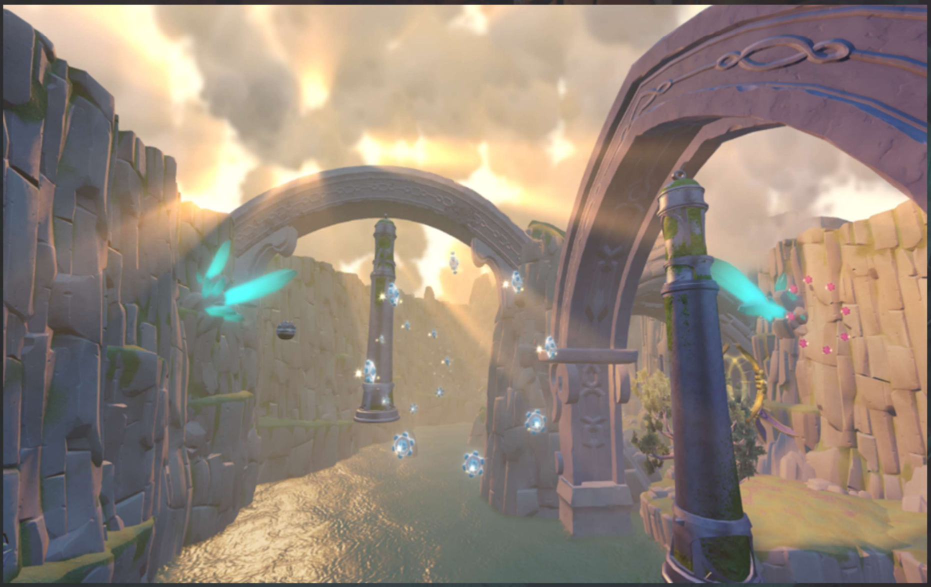


Happy accidents

Bloom for VFX

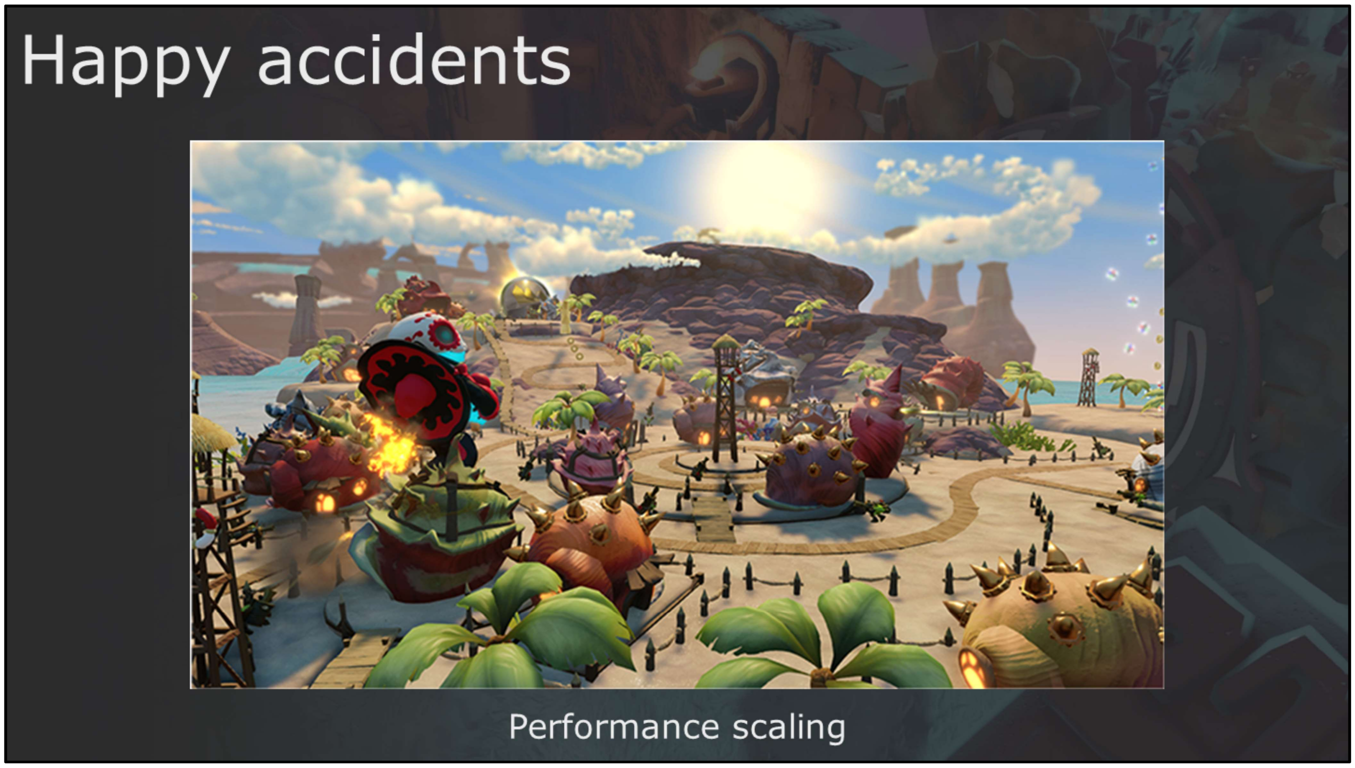
Because we had the vfx in a separate buffer we were able to apply an alternate thresholding to it to generate blooming vfx

Happy accidents



Mask for post effects

The alpha channel in a pre-multiplied buffer is simply a mask, you can use this to inform other post effects like sun-shafts and bloom.

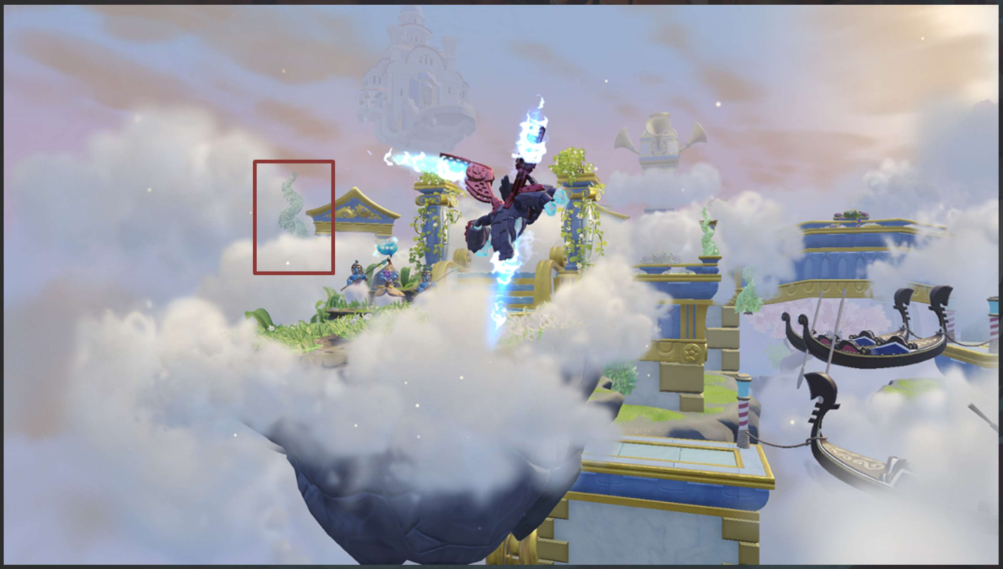


Happy accidents

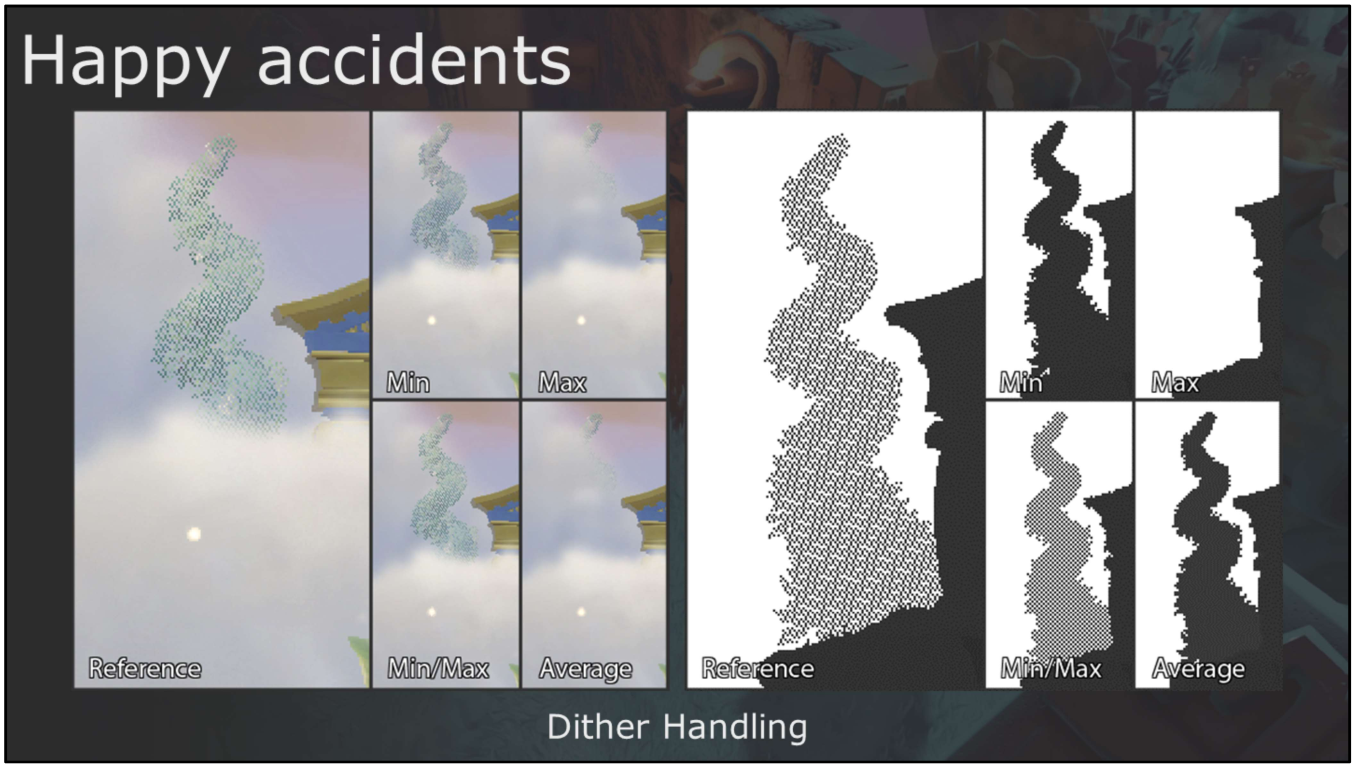
Performance scaling

Because the single pass approach is inherently part of the two pass approach we can selectively disable the second pass entirely. This converts it back to the single pass in scenarios where we really need the perf.

Happy accidents



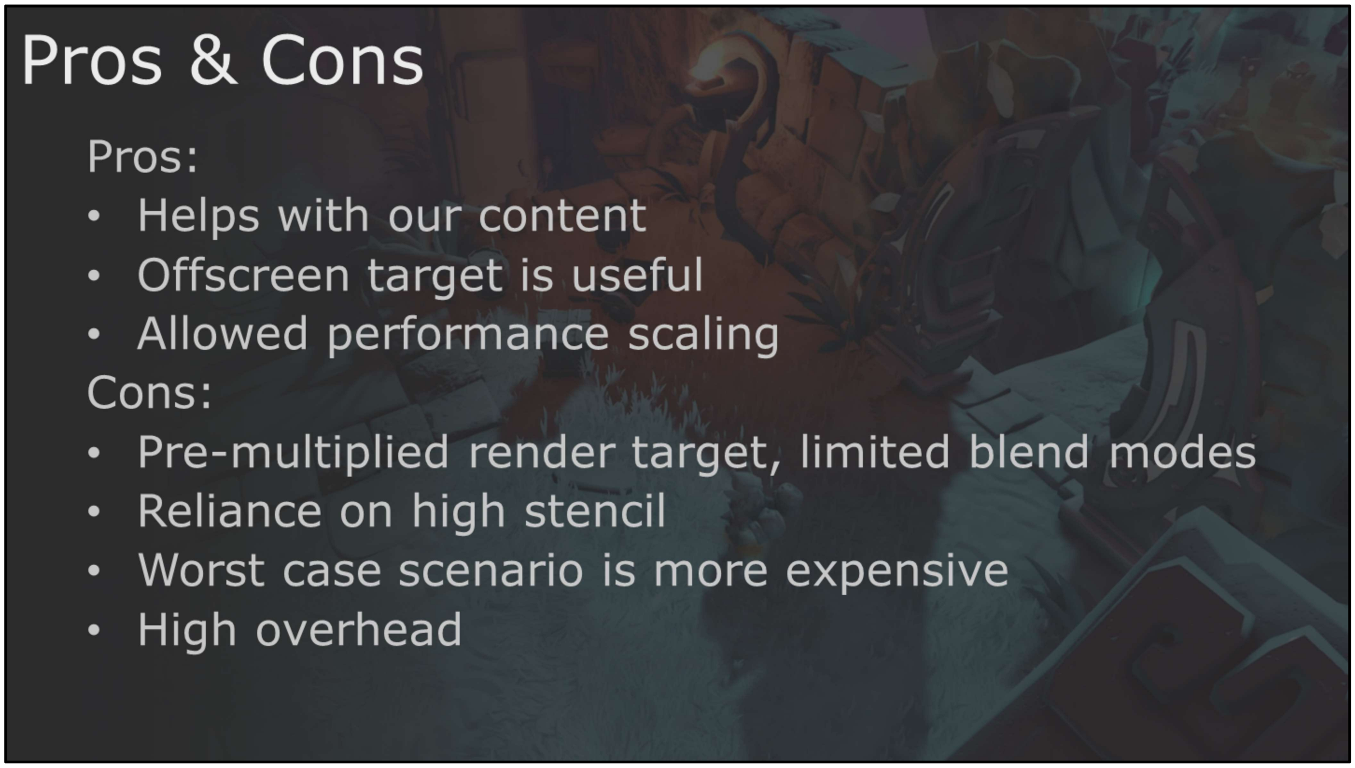
Dither Handling



Happy accidents

Because we used dither fading in our game it was important to have a downsample that respected that. Here you can see that the min/max buffer clearly maintained the dithered nature of the tree that is fading out.

PROS & CONS



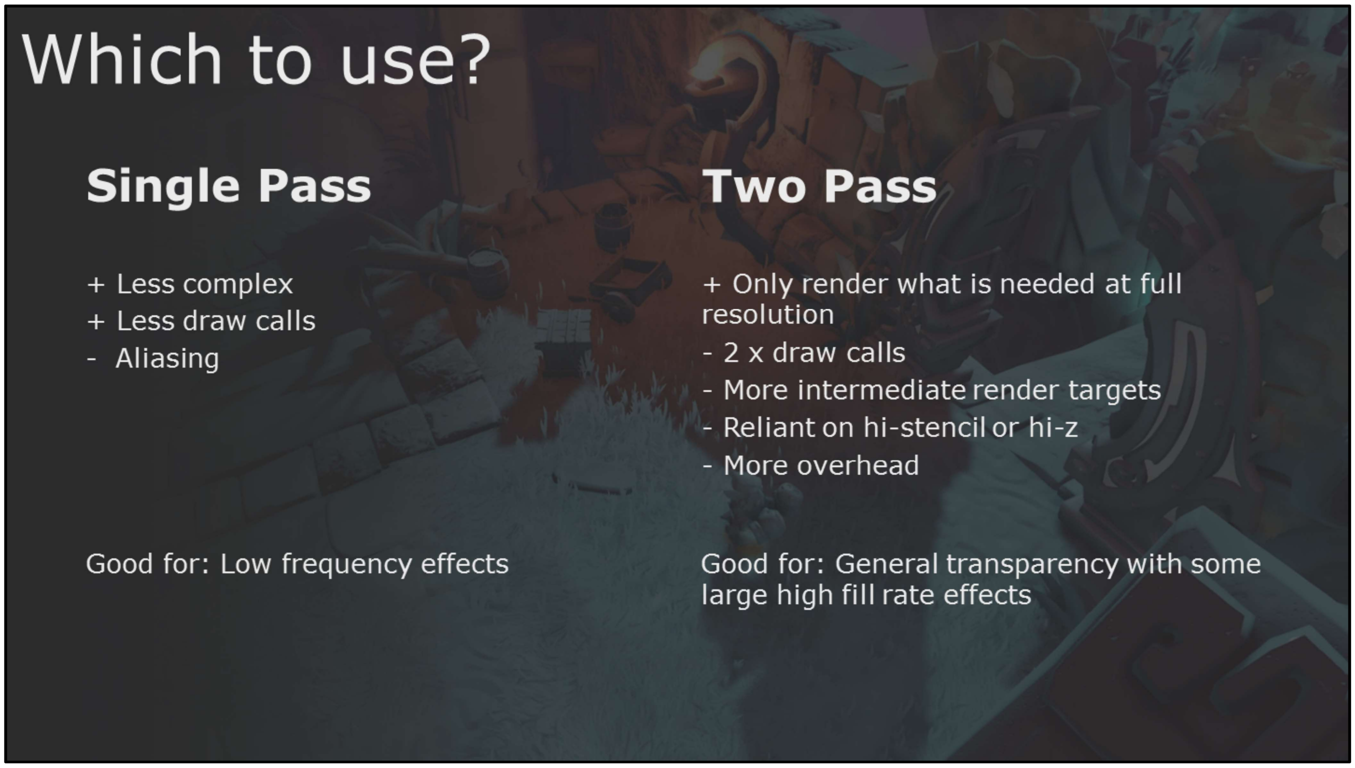
Pros & Cons

Pros:

- Helps with our content
- Offscreen target is useful
- Allowed performance scaling

Cons:

- Pre-multiplied render target, limited blend modes
- Reliance on high stencil
- Worst case scenario is more expensive
- High overhead



Which to use?

Single Pass

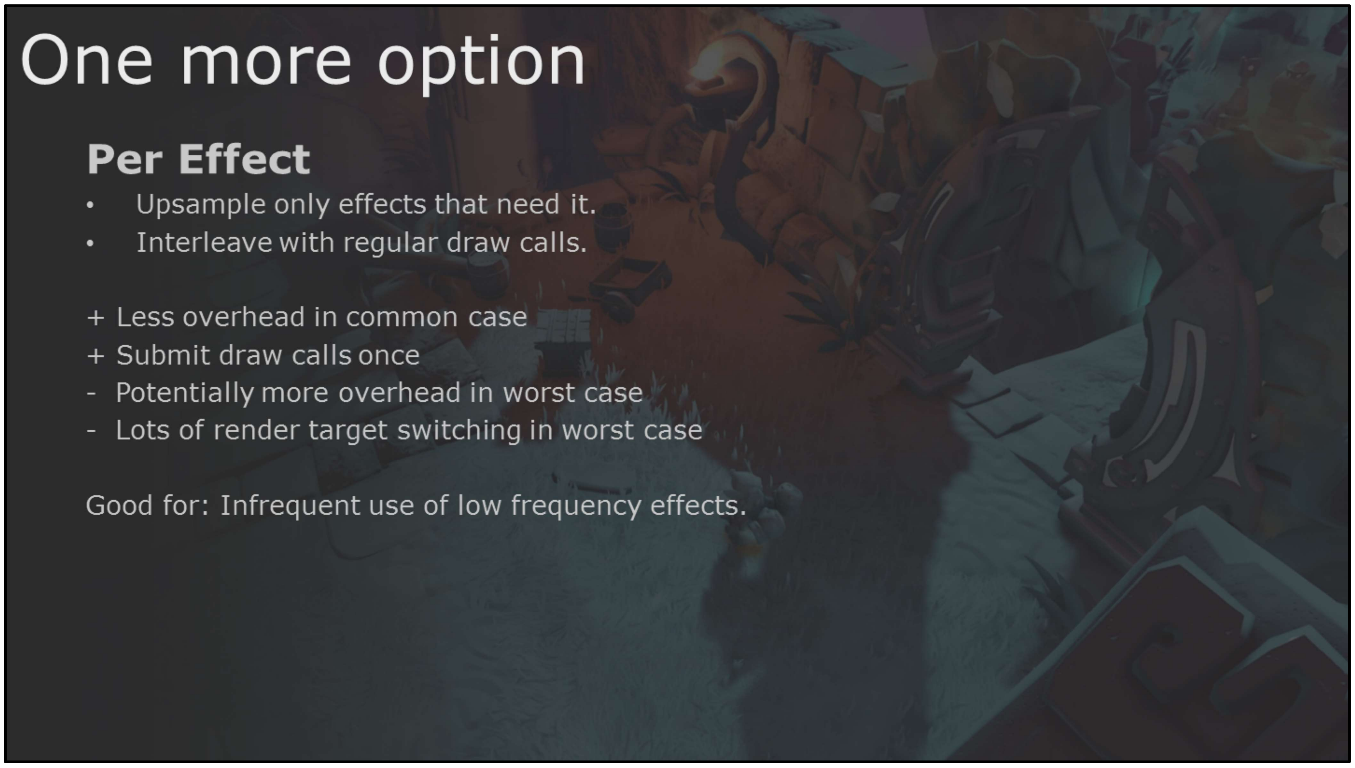
+ Less complex
+ Less draw calls
- Aliasing

Good for: Low frequency effects

Two Pass

+ Only render what is needed at full resolution
- 2 x draw calls
- More intermediate render targets
- Reliant on hi-stencil or hi-z
- More overhead

Good for: General transparency with some large high fill rate effects



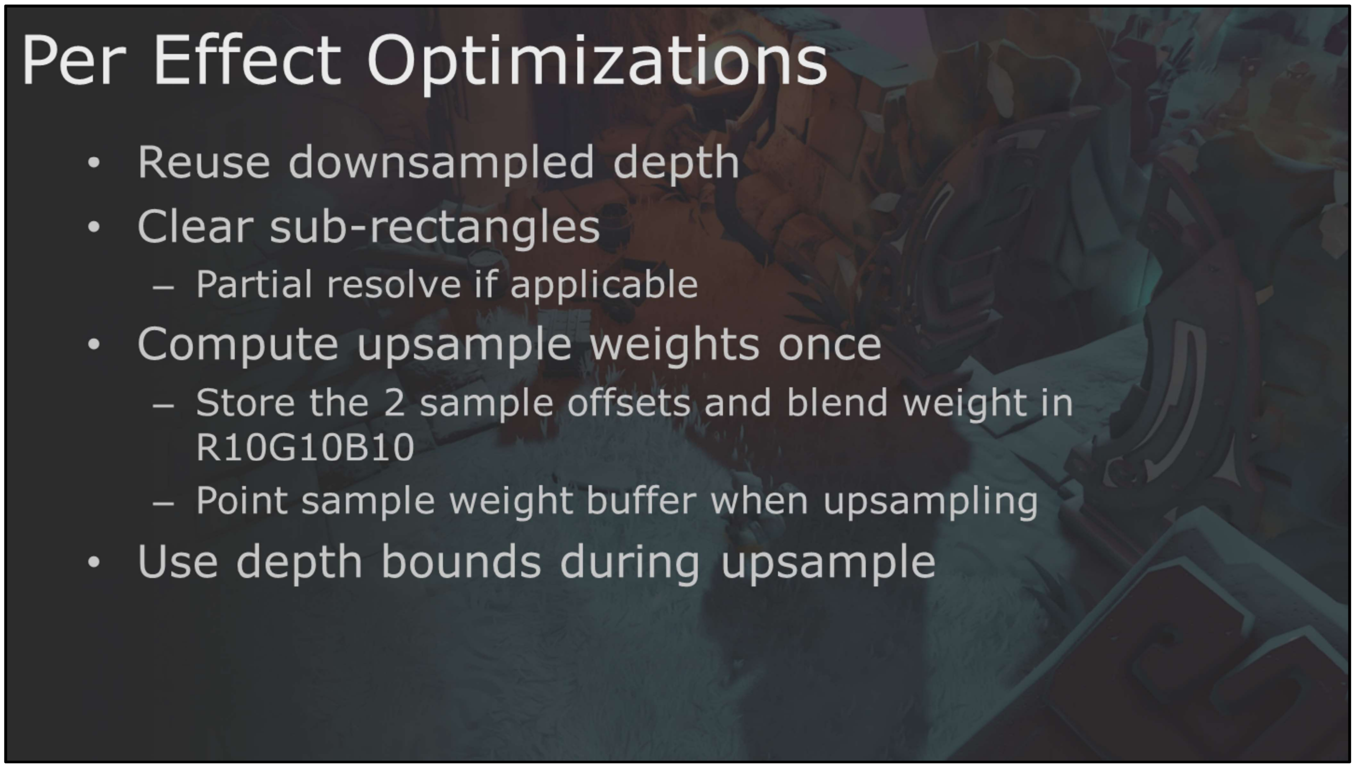
One more option

Per Effect

- Upsample only effects that need it.
- Interleave with regular draw calls.

+ Less overhead in common case
+ Submit draw calls once
- Potentially more overhead in worst case
- Lots of render target switching in worst case

Good for: Infrequent use of low frequency effects.



Per Effect Optimizations

- Reuse downsampled depth
- Clear sub-rectangles
  - Partial resolve if applicable
- Compute upsample weights once
  - Store the 2 sample offsets and blend weight in R10G10B10
  - Point sample weight buffer when upsampling
- Use depth bounds during upsample

Future Work

- Improve target slope threshold
- Better blend mode support
- Increased depth variance
- Implement per-effect
- Improved edge detection
- Cheaper upsample w/ two pass

QUESTIONS?