THE RENDERING OF INSIDE

HIGH FIDELITY, LOW COMPLEXITY

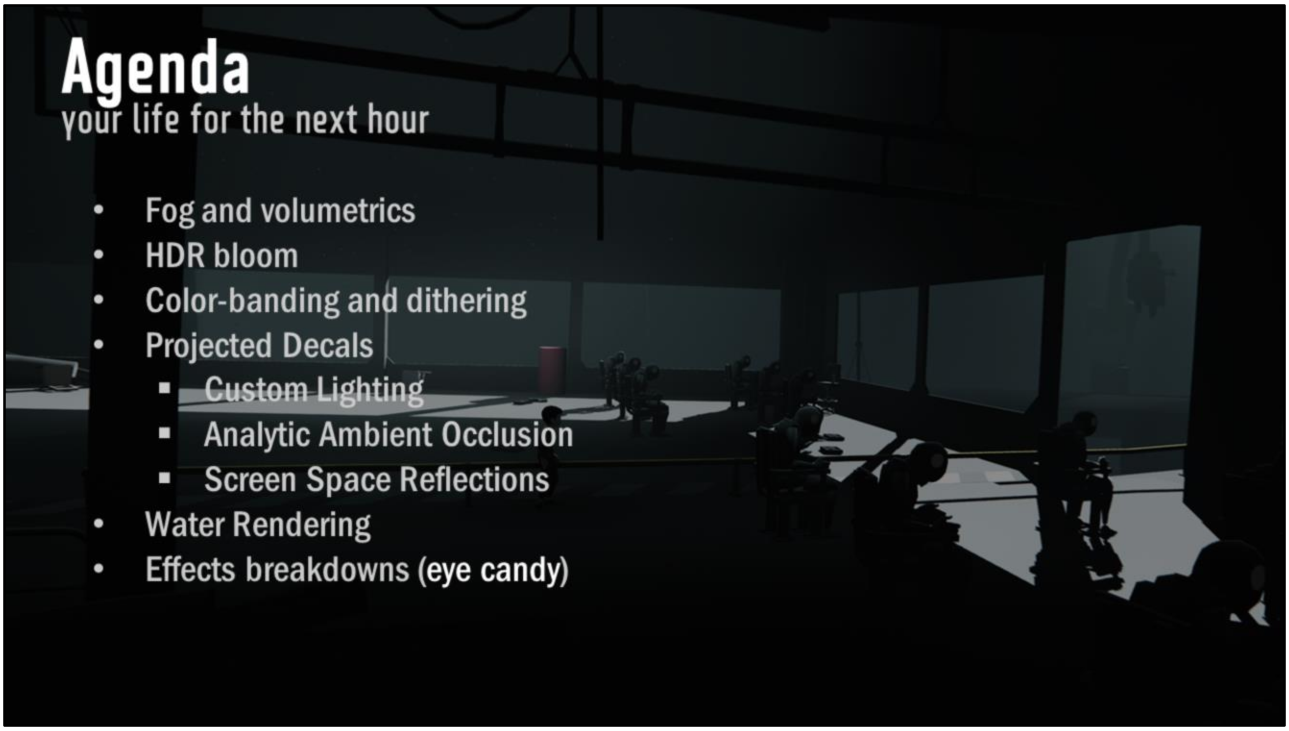
PLAYDEAD@GDC2016

Mikkel Gjøl & Mikkel Svendsen

url: github.com/playdeadgames/publications

@pixelmager
@ikarosav

Trailer: http://playdead.com/inside/

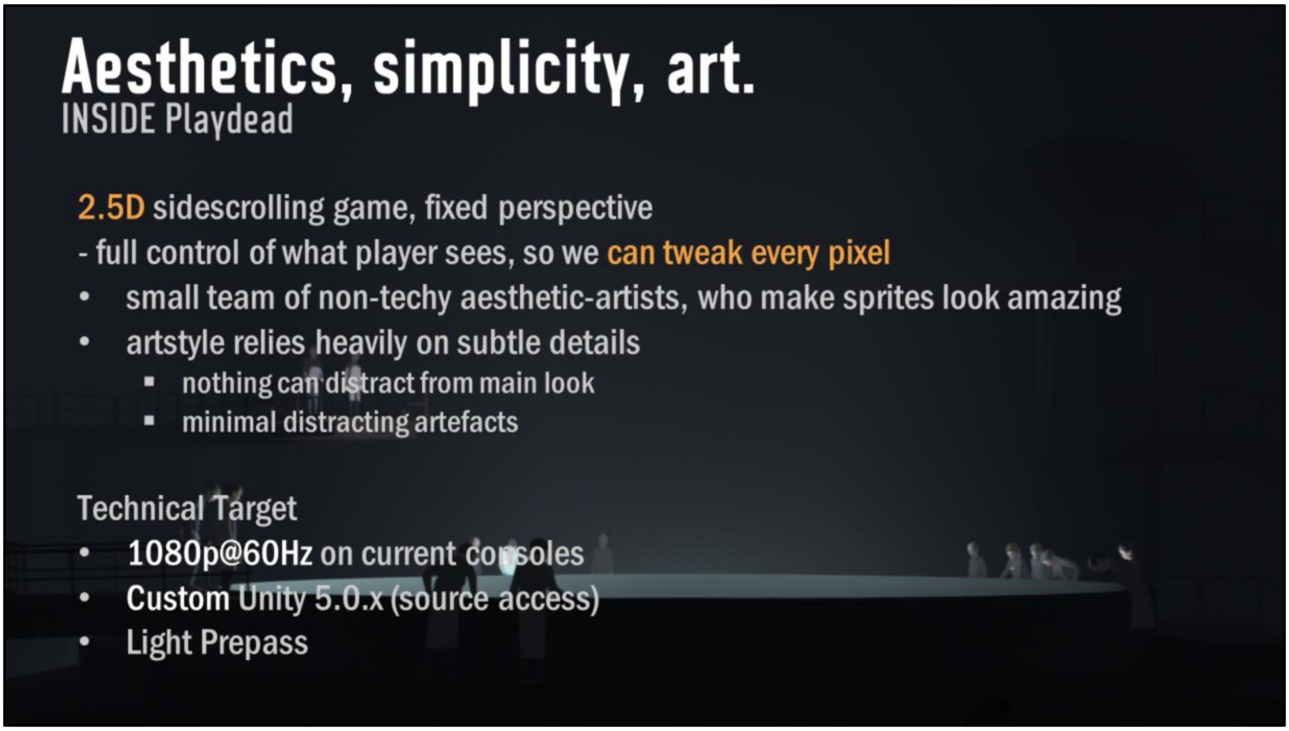


Agenda

your life for the next hour

- Fog and volumetrics
- HDR bloom
- Color-banding and dithering
- Projected Decals
  - Custom Lighting
  - Analytic Ambient Occlusion
  - Screen Space Reflections
- Water Rendering
- Effects breakdowns (**eye candy**)

Specifically rendering wise that might be useful to others



Aesthetics, simplicity, art.

INSIDE Playdead

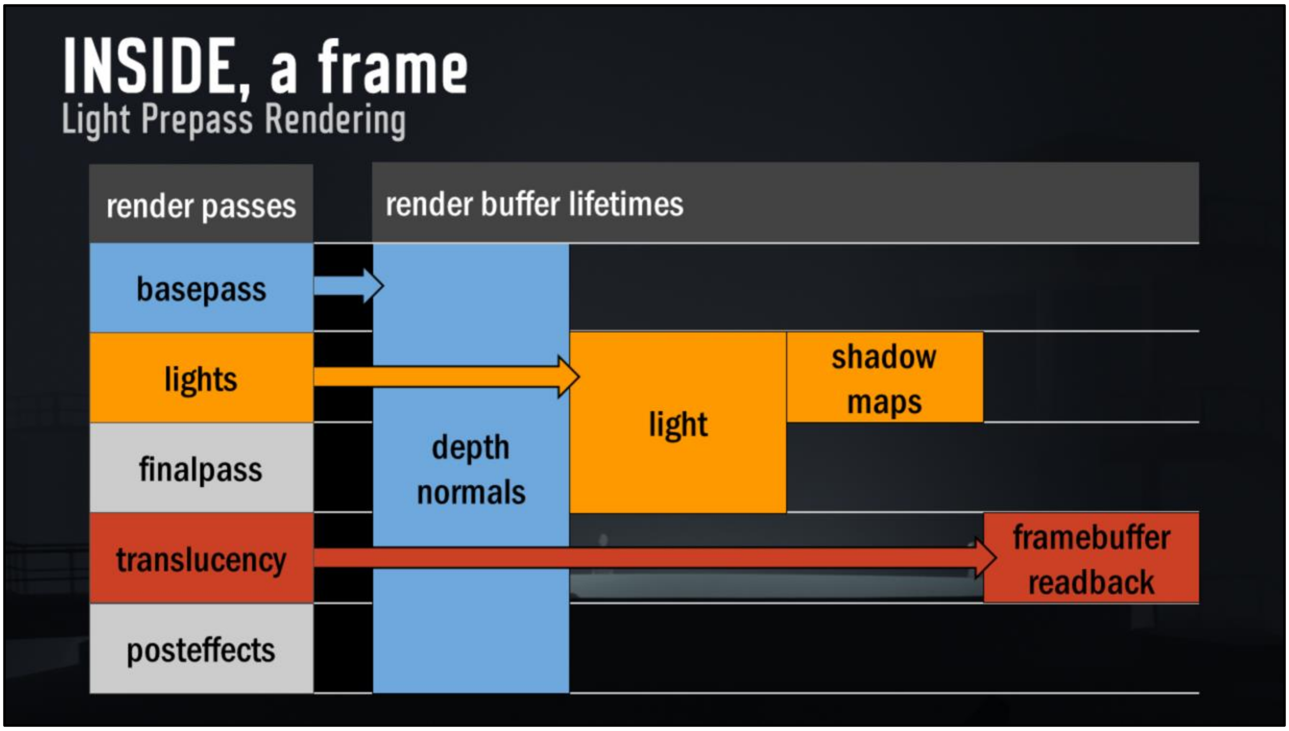
2.5D sidescrolling game, fixed perspective

- full control of what player sees, so we can tweak every pixel

- small team of non-techy aesthetic-artists, who make sprites look amazing
- artstyle relies heavily on subtle details
  - nothing can distract from main look
  - minimal distracting artefacts

Technical Target

- 1080p@60Hz on current consoles
- Custom Unity 5.0.x (source access)
- Light Prepass



INSIDE, a frame

Light Prepass Rendering

render passes

render buffer lifetimes

basepass

lights

finalpass

translucency

posteffects

depth normals

light

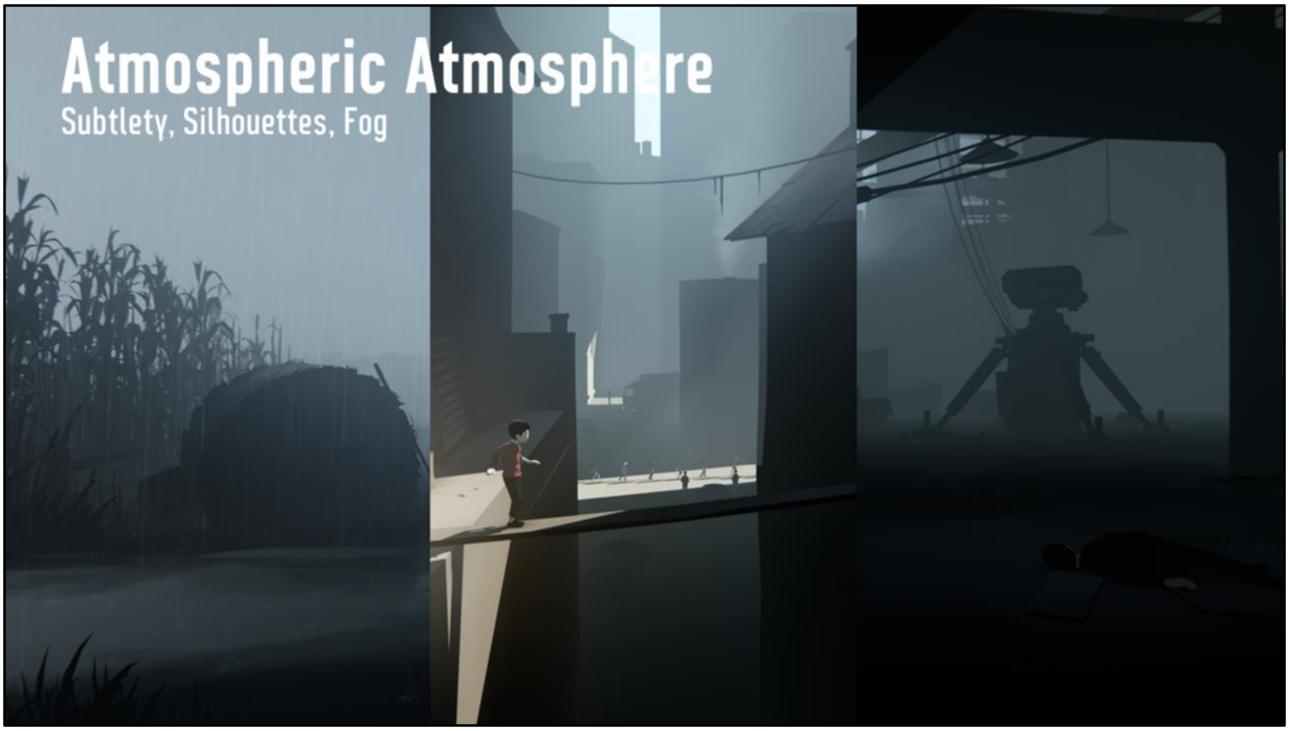
shadow maps

framebuffer readback

Essentially use vanilla light-prepass rendering (unity: “Legacy deferred”)

Note: single “grabpass” (copy of backbuffer) at first effect that uses it in translucency

note: shadowmaps for “static” lights actually span several frames
note: reflection-textures are rendered before the frame

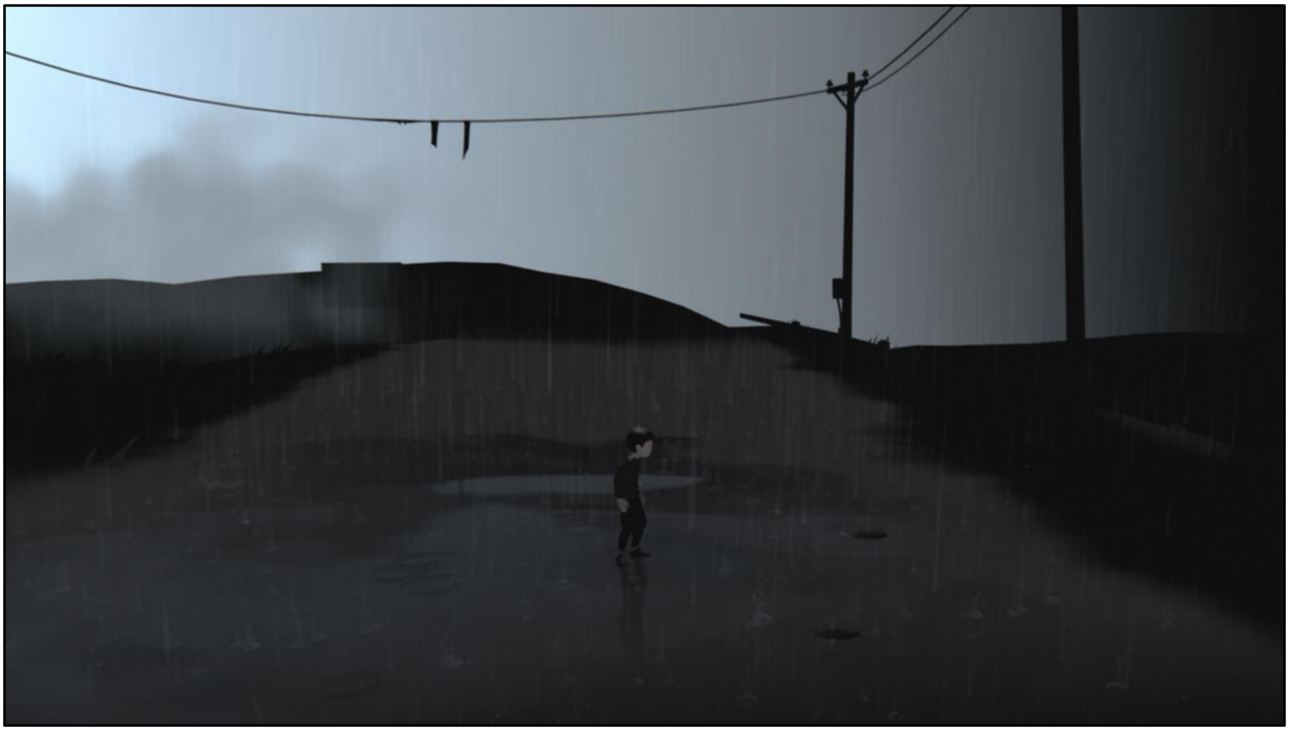


Atmospheric Atmosphere

Subtlety, Silhouettes, Fog

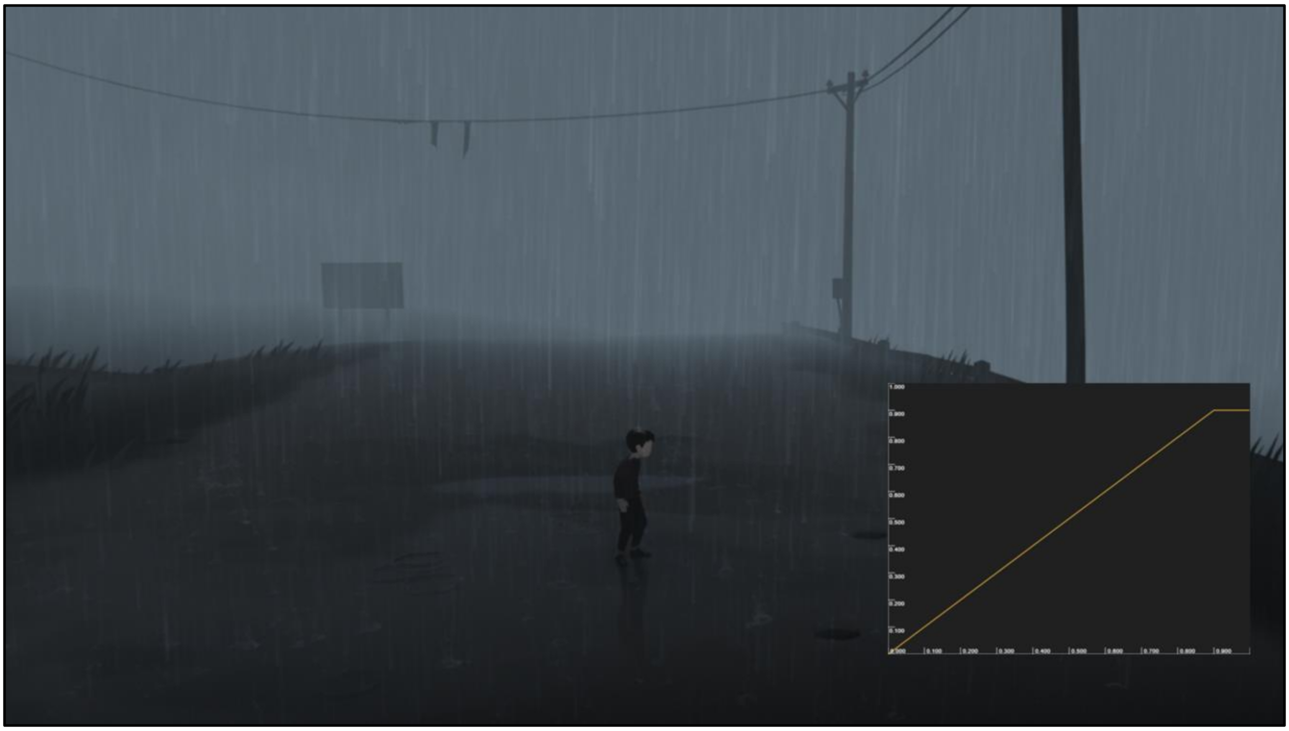
Fog turned out to be very central to our artstyle

Many of the initial scenes in the game were literally just fog + silhouettes



Quick set of images to show how much mileage our artists get out of the fog

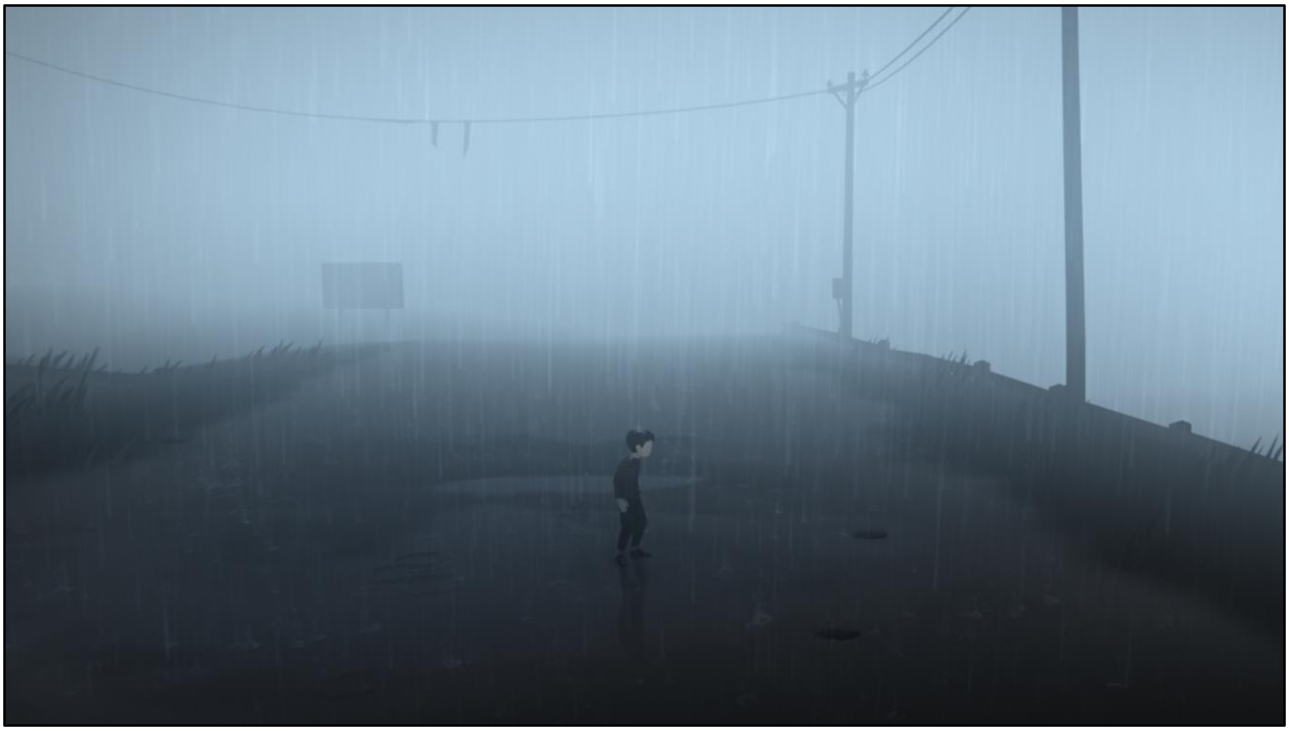
Here without no fog or “scattering”
...we are relying on simple fog A LOT for our expression, very bare without it.



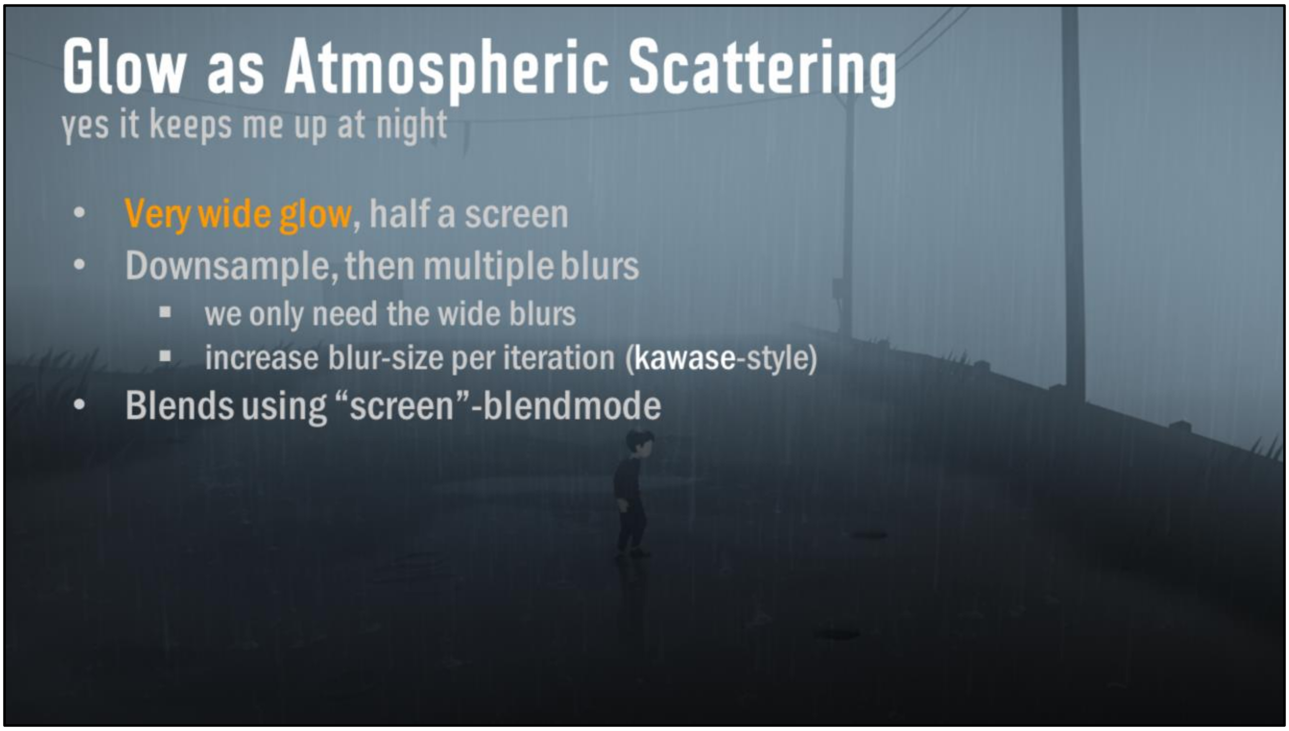
Fog, no “scattering”

Really uses fog to set the mood
Simple linear fog!

Only interesting tidbit: We clamp the fog to a max-value to enable bright objects to “bleed through”
(not shown here, used for headlights, spotlights etc.).



fog+glare
the main effect of glare here is atmospheric scattering
(adds a bit of vignetting as well)



Glow as Atmospheric Scattering

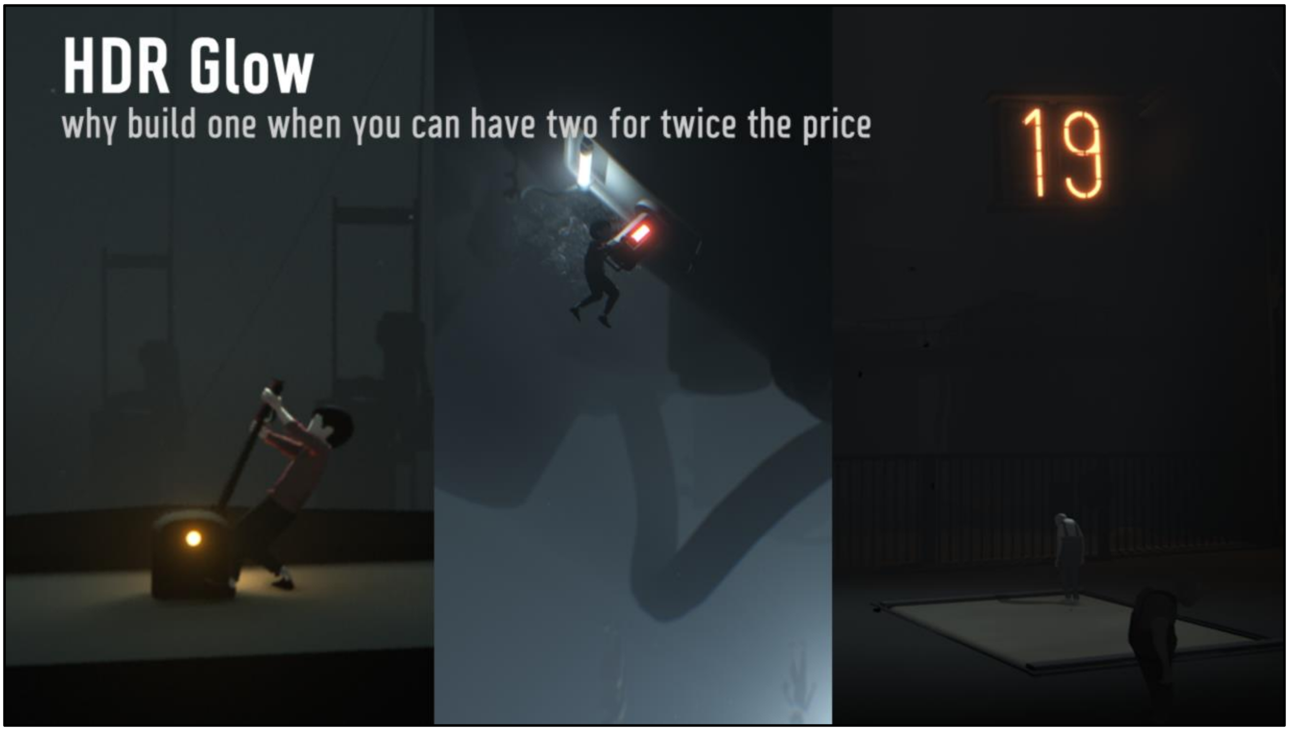
yes it keeps me up at night

- Very wide glow, half a screen
- Downsample, then multiple blurs
  - we only need the wide blurs
  - increase blur-size per iteration (**kawase**-style)
- Blends using “screen”-blendmode

Artists added early on.
The old-school low-threshold “Buttery lens” LDR glow-effect that makes everything look unintentionally misty…
Unsurprisingly, works well at intentionally creating a misty look :)

http://www.chrisoat.com/papers/Oat-SteerableStreakFilter.pdf

www.daionet.gr.jp/~masa/archives/GDC2003_DSTEAL.ppt



HDR Glow

why build one when you can have two for twice the price

There was a need for a tight glow-pass.
We tried just using a combined glow-pass with weights for each layer, but the fog-glow and the narrow glows interfered too much with each other... so added two separate passes (and tweaked them for their respective uses).

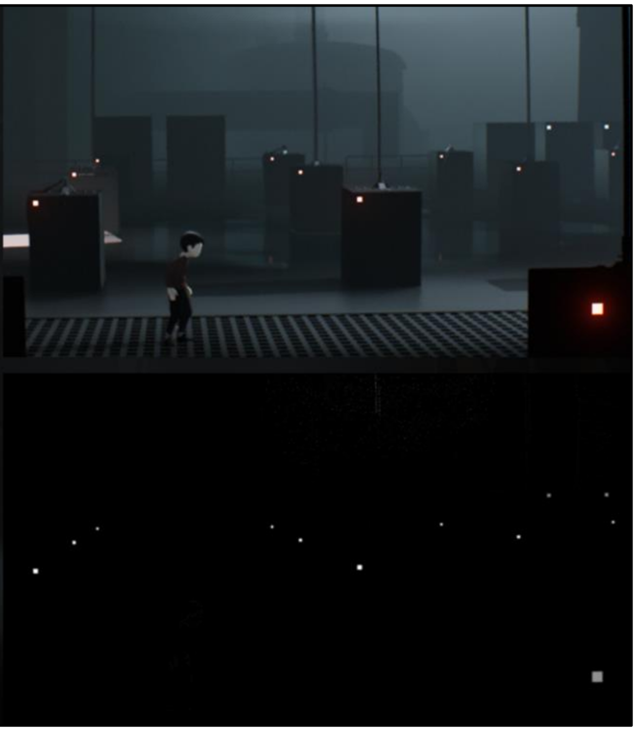
HDR Glow

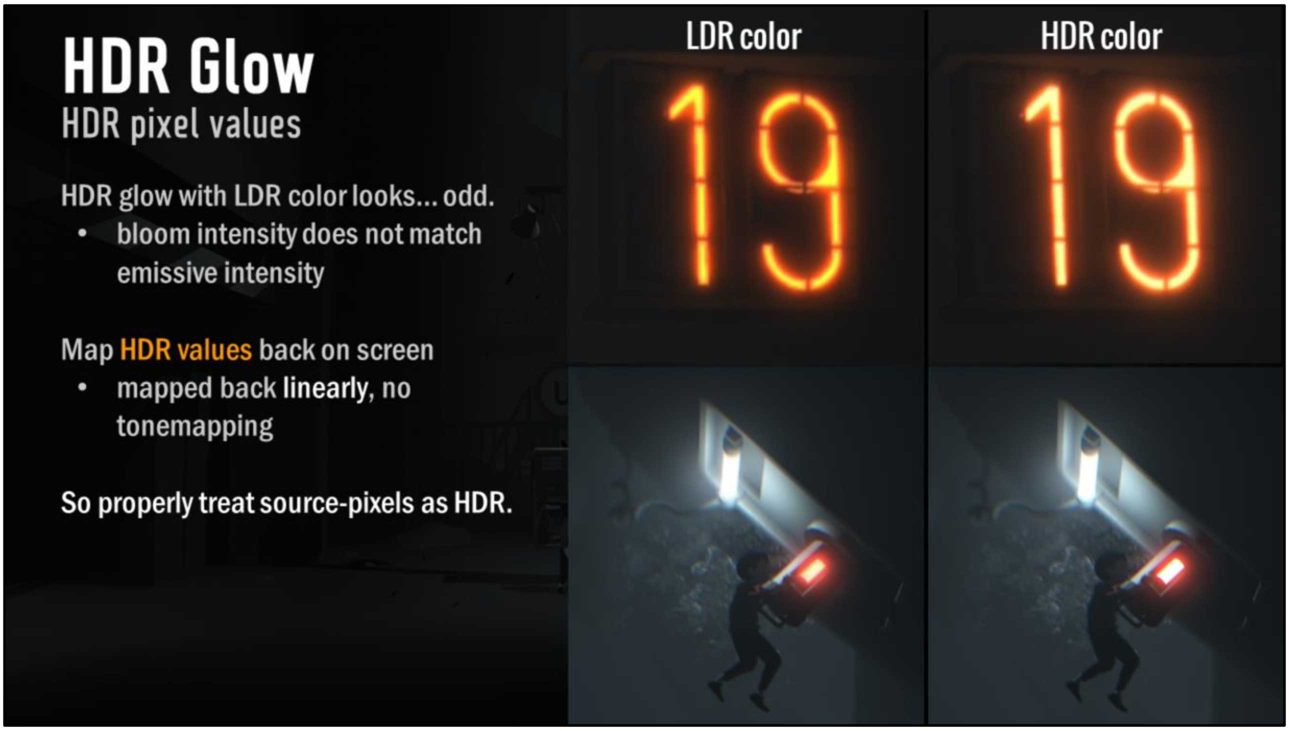
second glow pass, bright glowy objects

Narrow glow from **masked** objects

- **Emissive** materials only (written to alpha-channel)
- Mask-values remap RGB to non-linear intensity [1;7ish]

Intermediate HDR-values are encoded to [0;1]-fixedpoint as $x/(x+1)$





HDR Glow

HDR pixel values

HDR glow with LDR color looks... odd.

- bloom intensity does not match emissive intensity

Map **HDR values** back on screen

- mapped back **linearly**, no tonemapping

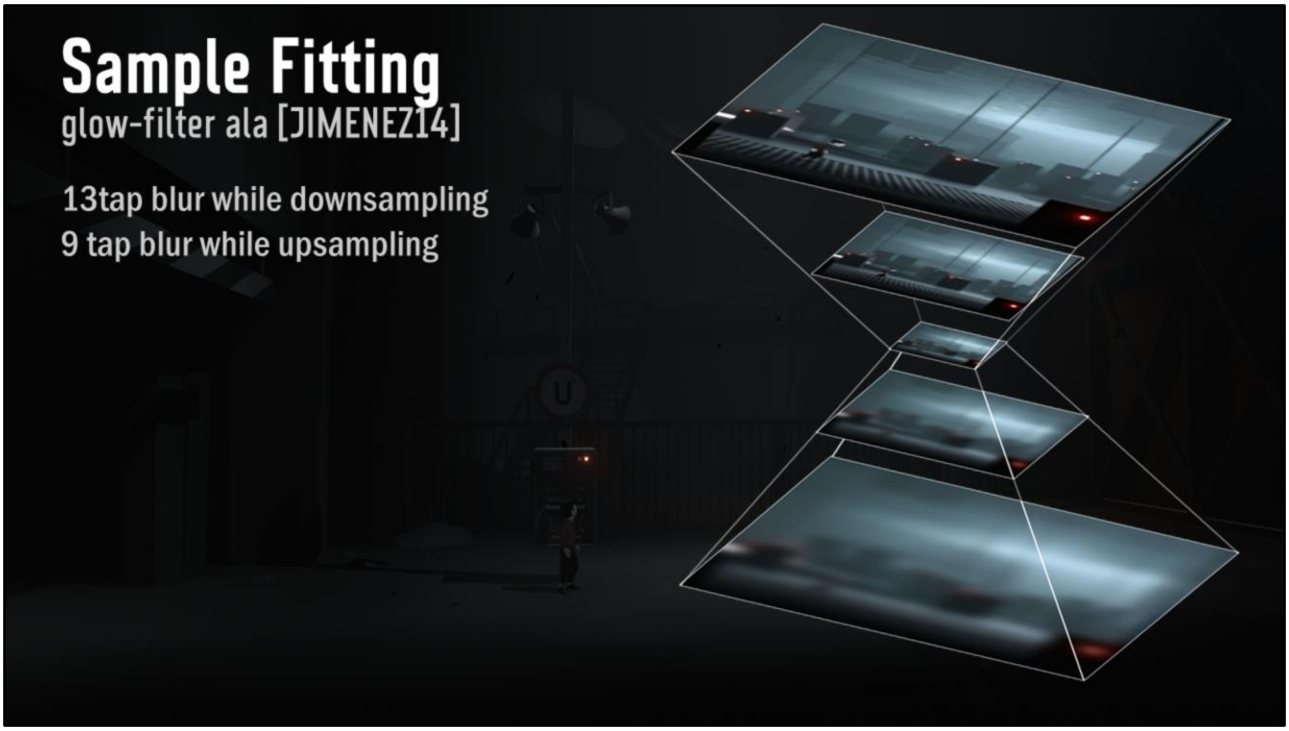
So properly treat source-pixels as HDR.

LDR color

HDR color

If bloom calculated with a certain intensity, we need to show source pixel with the same intensity

- obvious in hindsight, but very easy to just add a glow-intensity slider and not think any further of it – made a big visual difference to do it right!



Sample Fitting

glow-filter ala [JIMENEZ14]

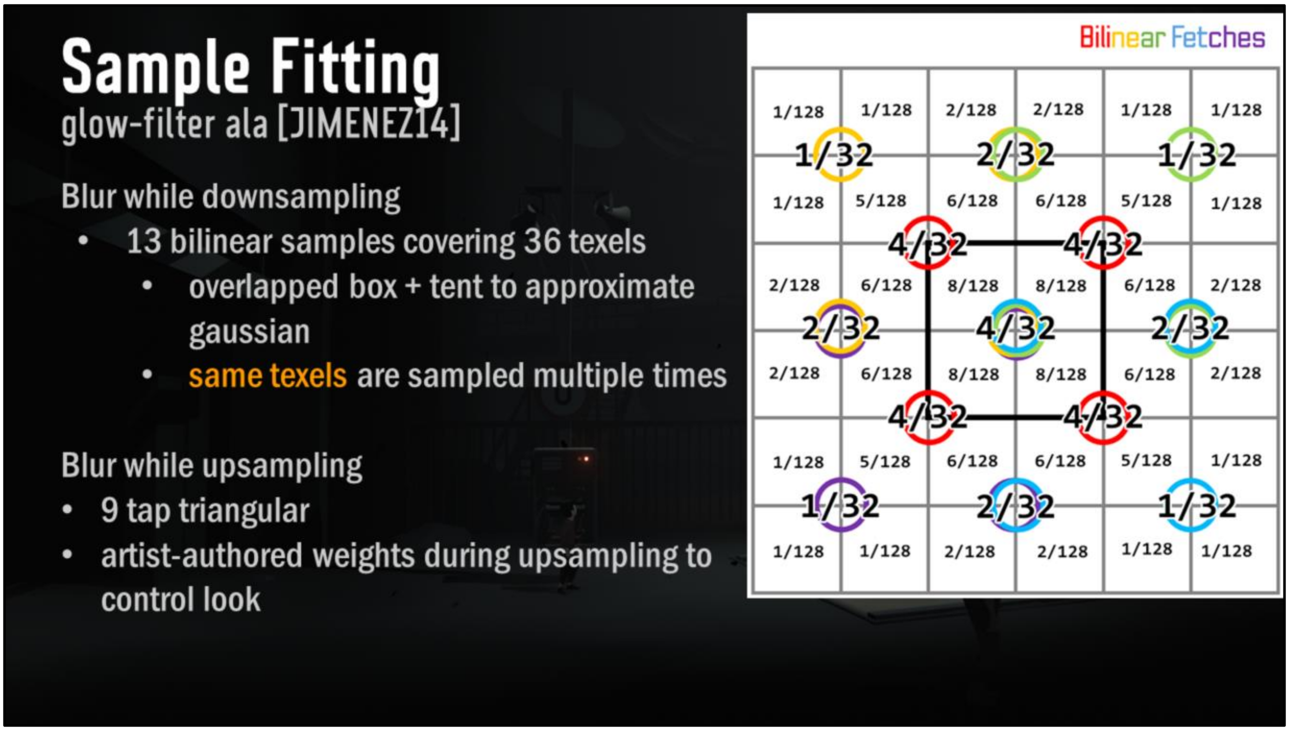
13tap blur while downsampling

9 tap blur while upsampling

Jimenez, Mittring (samaritan)

(yeayea the source-image already has glow, you pedant, you!)

http://www.iryoku.com/next-generation-post-processing-in-call-of-duty-advanced-warfare
https://de45xmedrsdbp.cloudfront.net/Resources/files/The_Technology_Behind_the_Elemental_Demo_16x9-1248544805.pdf



Sample Fitting

glow-filter ala [JIMENEZ14]

Blur while downsampling

- 13 bilinear samples covering 36 texels
  - overlapped box + tent to approximate gaussian
  - same texels are sampled multiple times

Blur while upsampling

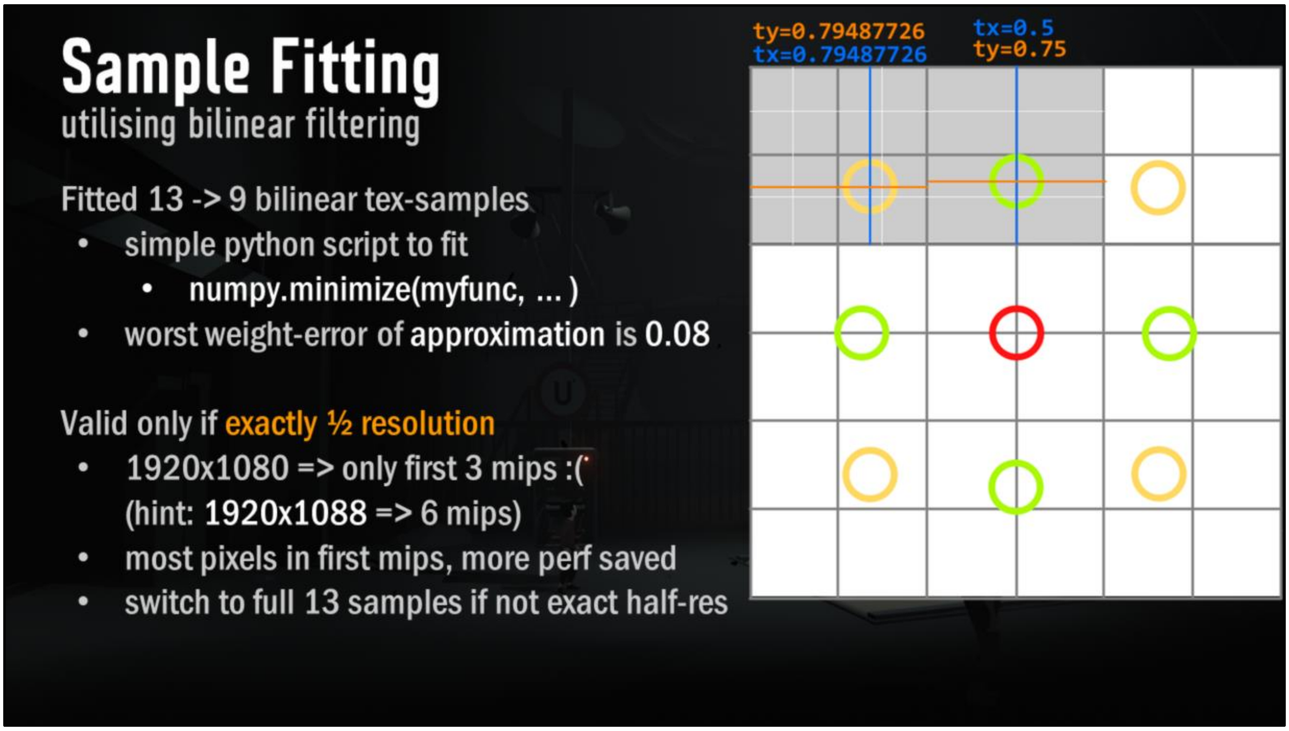
- 9 tap triangular
- artist-authored weights during upsampling to control look

Bilinear Fetches

1/128	1/128	2/128	2/128	1/128	1/128
1/128	5/128	6/128	6/128	5/128	1/128
2/128	6/128	8/128	8/128	6/128	2/128
2/128	6/128	8/128	8/128	6/128	2/128
1/128	5/128	6/128	6/128	5/128	1/128
1/128	1/128	2/128	2/128	1/128	1/128

Observation is that the same texels are sampled multiple times.

We use 9 samples instead of 13 the original samples, by fitting the samplepoint to utilise bilinear filtering to only sample each texel once.
(really, this should be done to a gaussian distribution, instead of this already approximated sampling... next time)



Sample Fitting

utilising bilinear filtering

Fitted 13 -> 9 bilinear tex-samples

- simple python script to fit
  - numpy.minimize(myfunc, ...)
- worst weight-error of approximation is 0.08

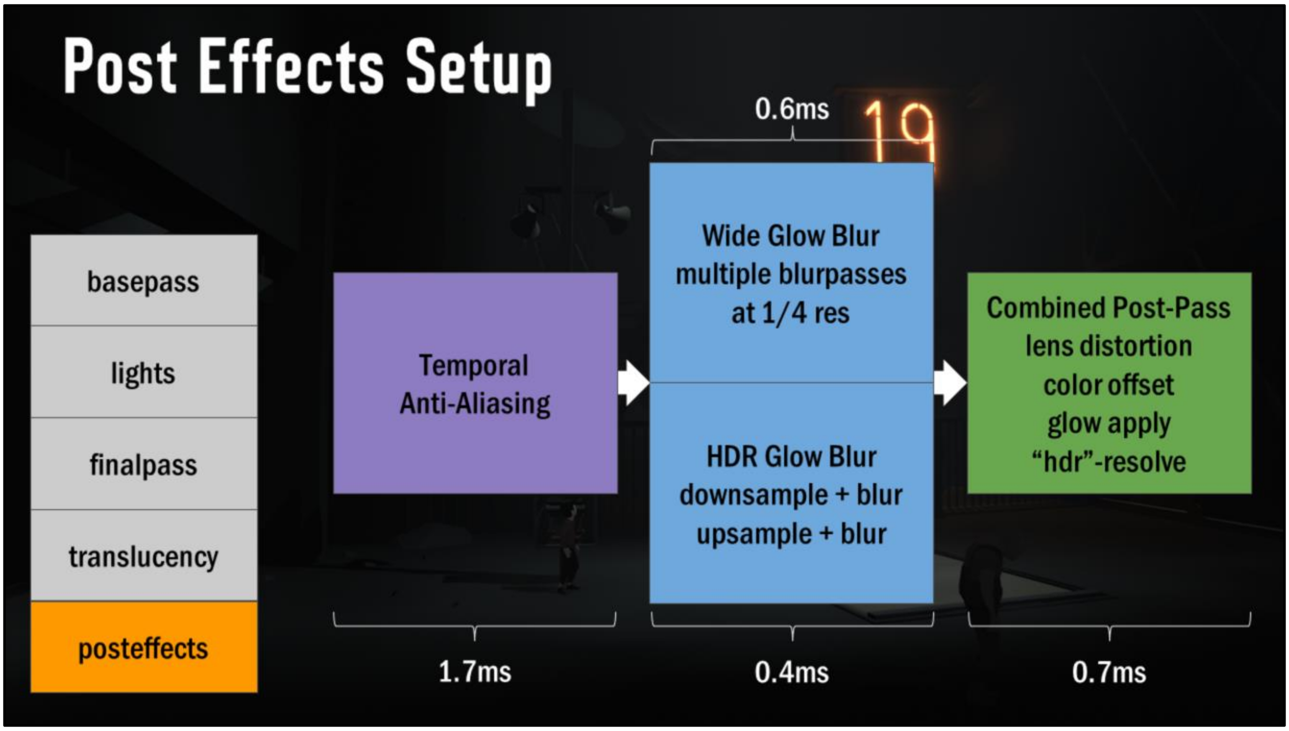
Valid only if exactly ½ resolution

- 1920x1080 => only first 3 mips :(
  (hint: 1920x1088 => 6 mips)
- most pixels in first mips, more perf saved
- switch to full 13 samples if not exact half-res

Corner sample has 8% error on bilinear weights, top sample is exact. Visually good enough.
Because we rely so much on the bilinear filtering, if we are not downsampling to exactly half resolution, we get severe issues... so use full 13 taps in that case.

(iirc saved ~0.1ms)

http://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.optimize.minimize.html
https://twitter.com/adamjmiles/status/683041184915263489



Post Effects Setup

basepass

lights

finalpass

translucency

posteffects

Temporal Anti-Aliasing

1.7ms

0.6ms

Wide Glow Blur
multiple blurpasses
at 1/4 res

HDR Glow Blur
downsample + blur
upsample + blur

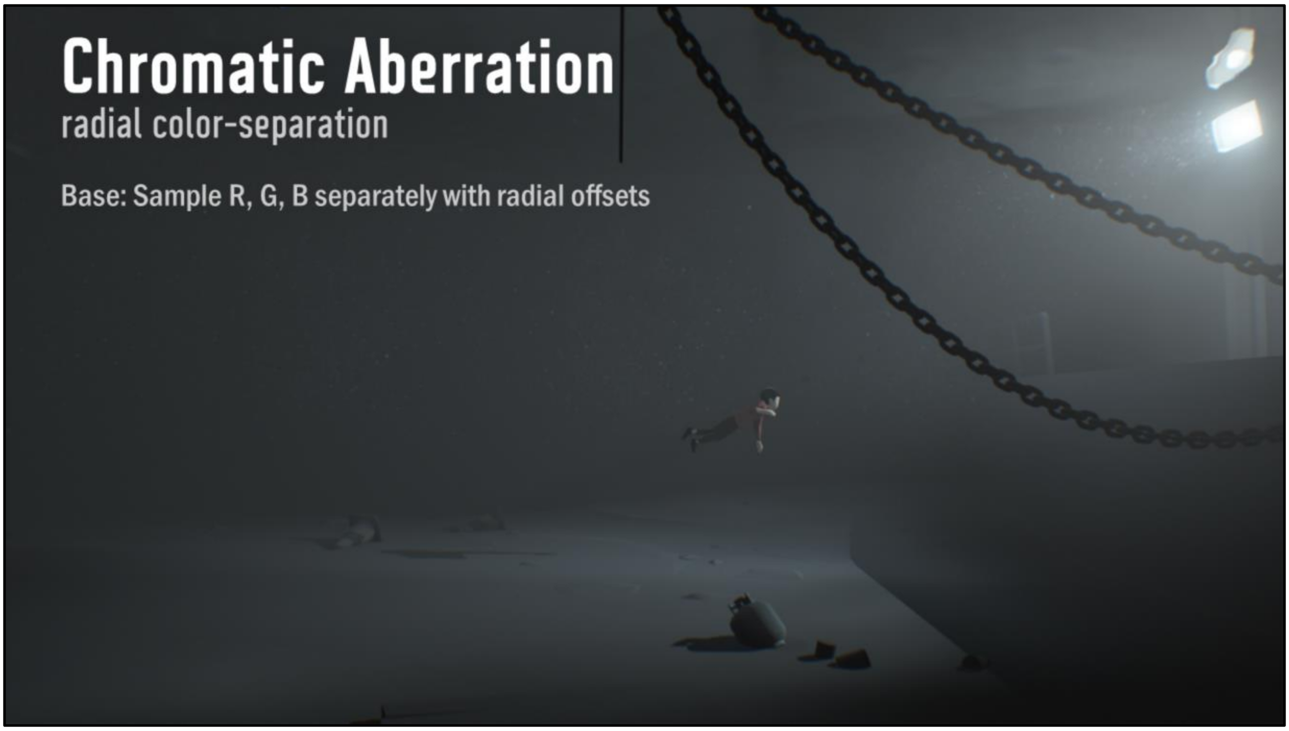
0.4ms

Combined Post-Pass
lens distortion
color offset
glow apply
“hdr”-resolve

0.7ms

TAA feeds into the bloom (important, as it also AAs the bloom-mask - very hard to do decent HDR-glow with aliasing input”)
Independent glow passes (but interleaved for performance reasons)

“HDR resolve” from glow mask



Chromatic Aberration

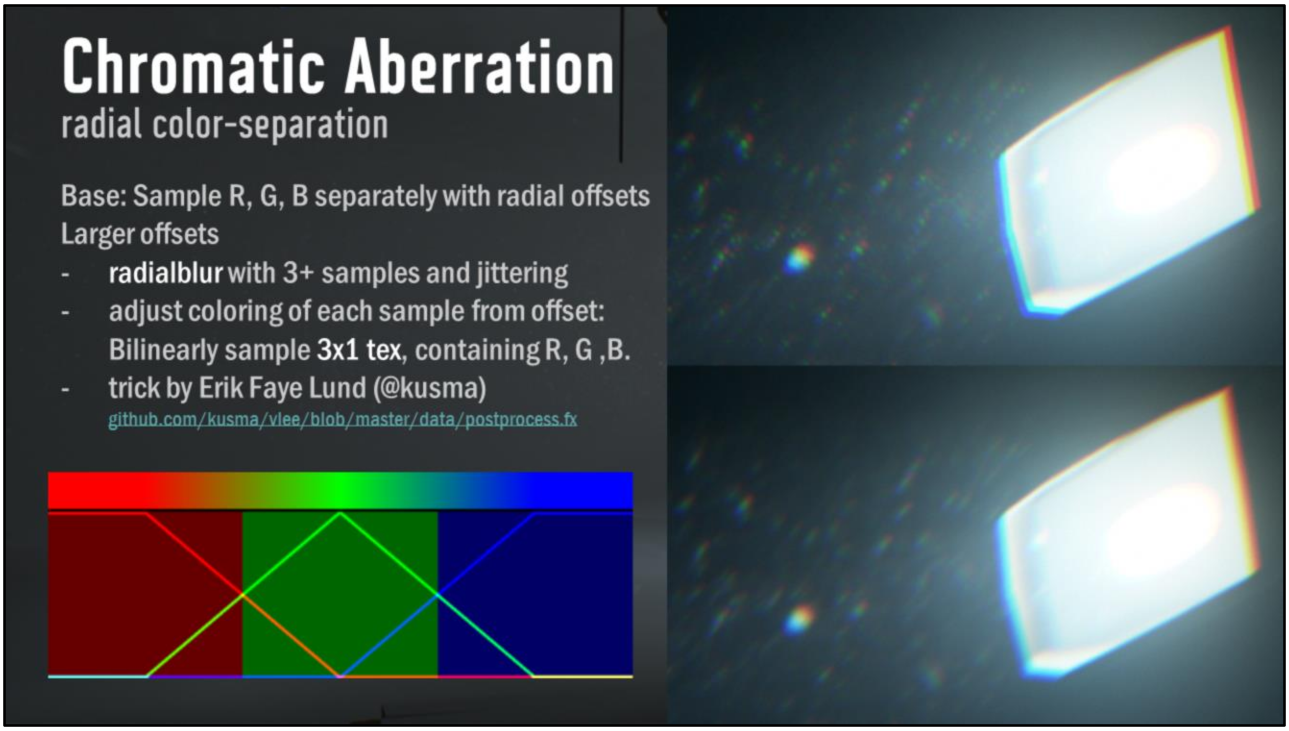
radial color-separation

Base: Sample R, G, B separately with radial offsets

Radial offsets

Three samples @1080p, 8 samples at 4k, jittering 8 sample above 4k. Jittering is hard on T$ but scales well.

We desaturate a bit afterwards, so it's in essense a very small radialblur



Chromatic Aberration

radial color-separation

Base: Sample R, G, B separately with radial offsets

Larger offsets

- **radialblur** with 3+ samples and jittering
- adjust coloring of each sample from offset: Bilinearly sample **3x1 tex**, containing R, G ,B.
- trick by Erik Faye Lund (@kusma)
  github.com/kusma/vlee/blob/master/data/postprocess.fx

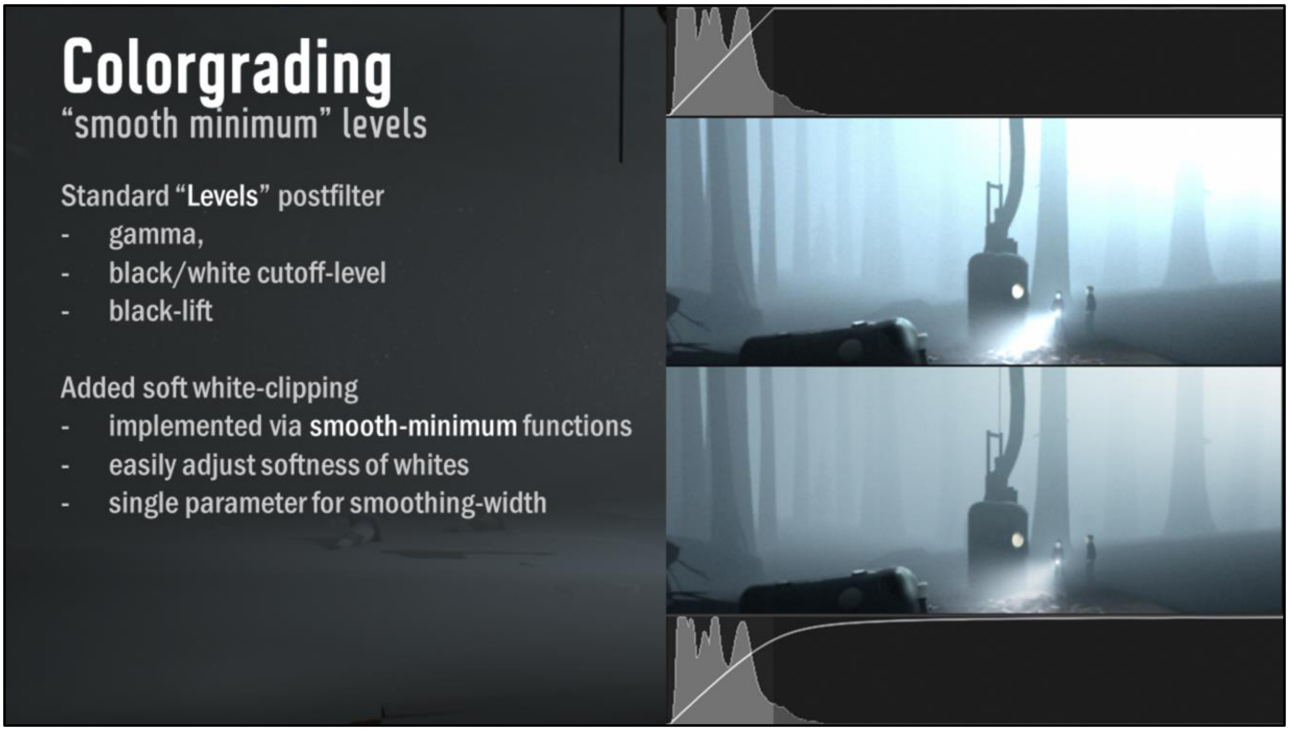
Larger offsets, underwater or highe res, radialblur with coloring

-

Done on both main image and glowtexture (faster than to add the extra pass to the glow-texture)

Want different colors? Hue-shift RGB-texture. Or design your own, just make sure:
At any point, RGB-weights must sum to 1 (for luminance to not vary)
Pixels must sum to (1,1,1) (for white to remain white)

Shadertoy implementation of jittering, not using the RGB-lookup trick.
https://www.shadertoy.com/view/XssGz8



Colorgrading

“smooth minimum” levels

Standard “**Levels**” postfilter

- gamma,
- black/white cutoff-level
- black-lift

Added soft white-clipping

- implemented via **smooth-minimum** functions
- easily adjust softness of whites
- single parameter for smoothing-width

Brightness adjustment

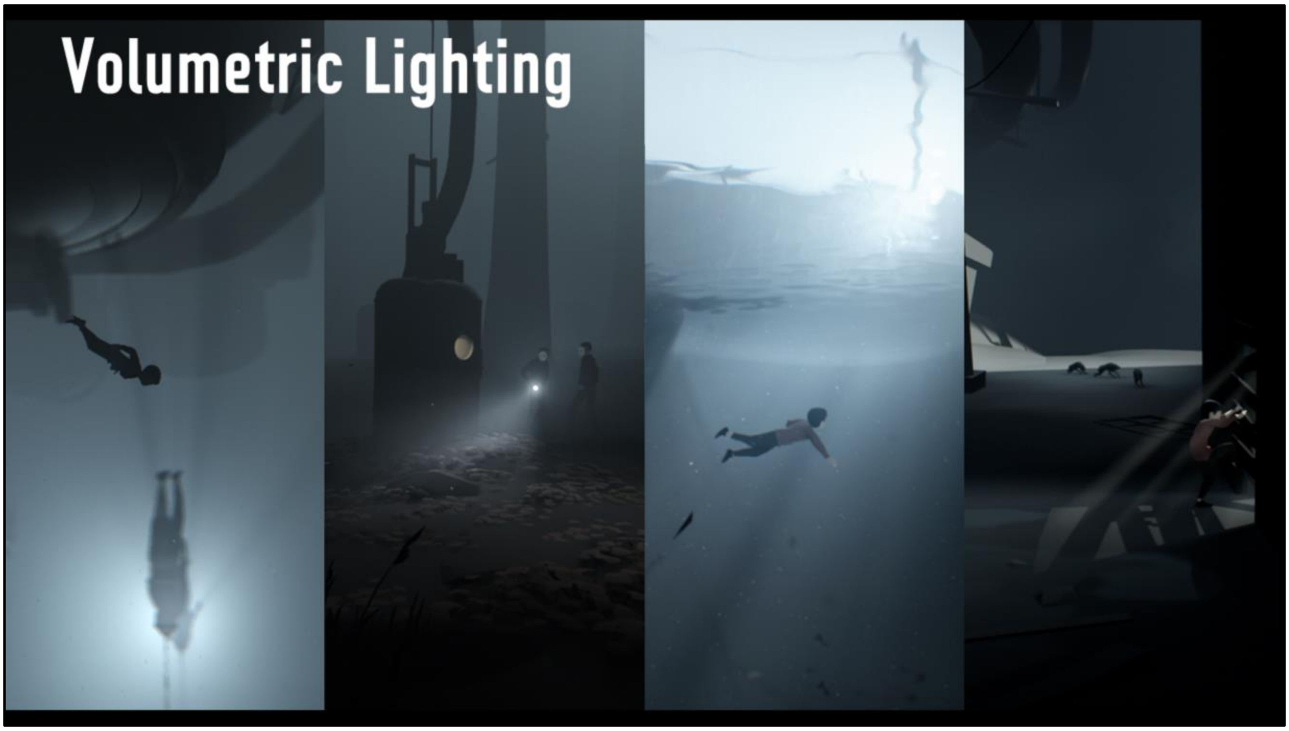
Smooth minimum of 1 and the brightness curve

(…we also have saturation-adjustment)

--
Our brightness implementation
https://www.shadertoy.com/view/Ml3GWs
- comparison of various methods:
https://www.shadertoy.com/view/MtjGWm

Shader implementation on smooth-min / smooth-max
https://www.shadertoy.com/view/ltf3W2

Some theory
http://iquilezles.org/www/articles/smin/smin.htm

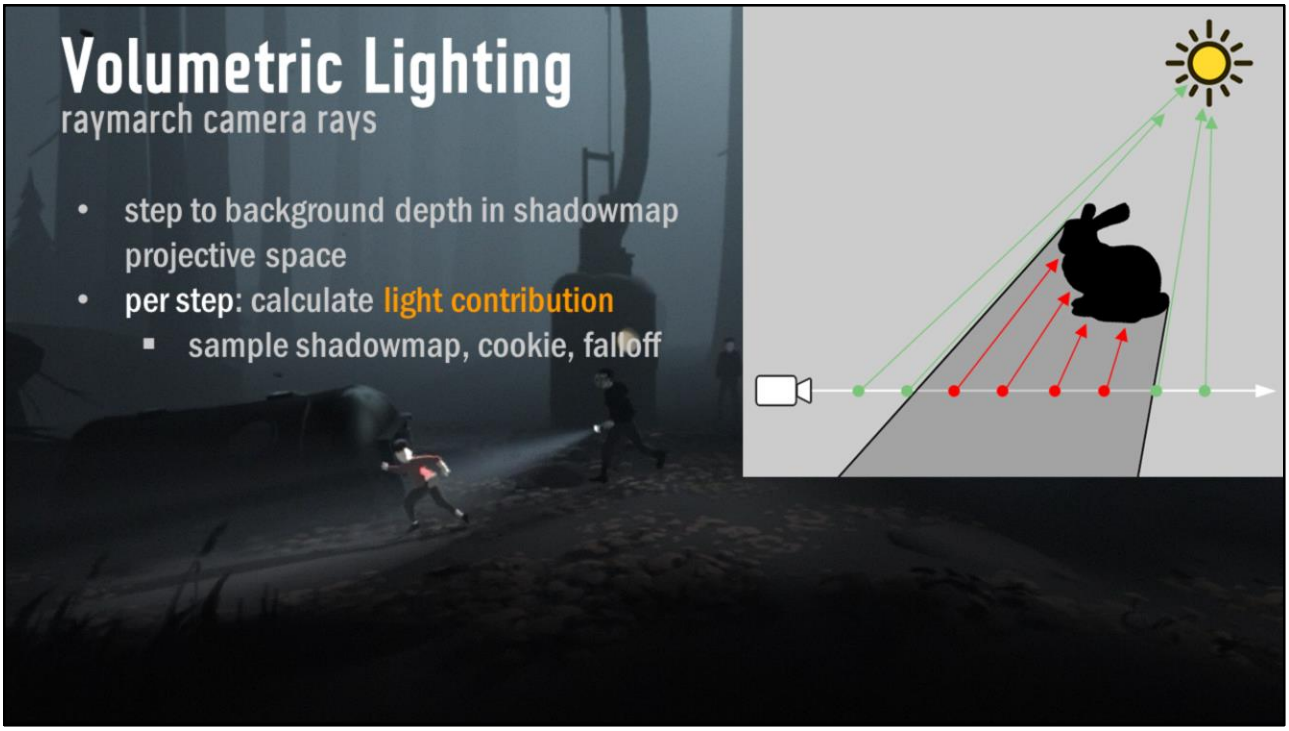


Volumetric Lighting

…back to fog!

Turned out we needed way more local control over fog that just the global linear fog we just showed.

Effects
- Underwater
- Flash lights
- Dusty air



Volumetric Lighting

raymarch camera rays

- step to background depth in shadowmap projective space
- **per step**: calculate light contribution
  - sample shadowmap, cookie, falloff

Uniform Sampling

128 samples per pixel, 22ms@1080p

```
vec3 stepvec=(p1-p0)/STEPS;
vec3 p = p0;

vec3 sum = 0;
for(int i=0; i<128; ++i)
{
    sum += sampleLight(p);
    p += stepvec;
}
sum /= STEPS;
```

Naïve brute force sampling
prohivitively large number of samples (around 128 required in this scene)

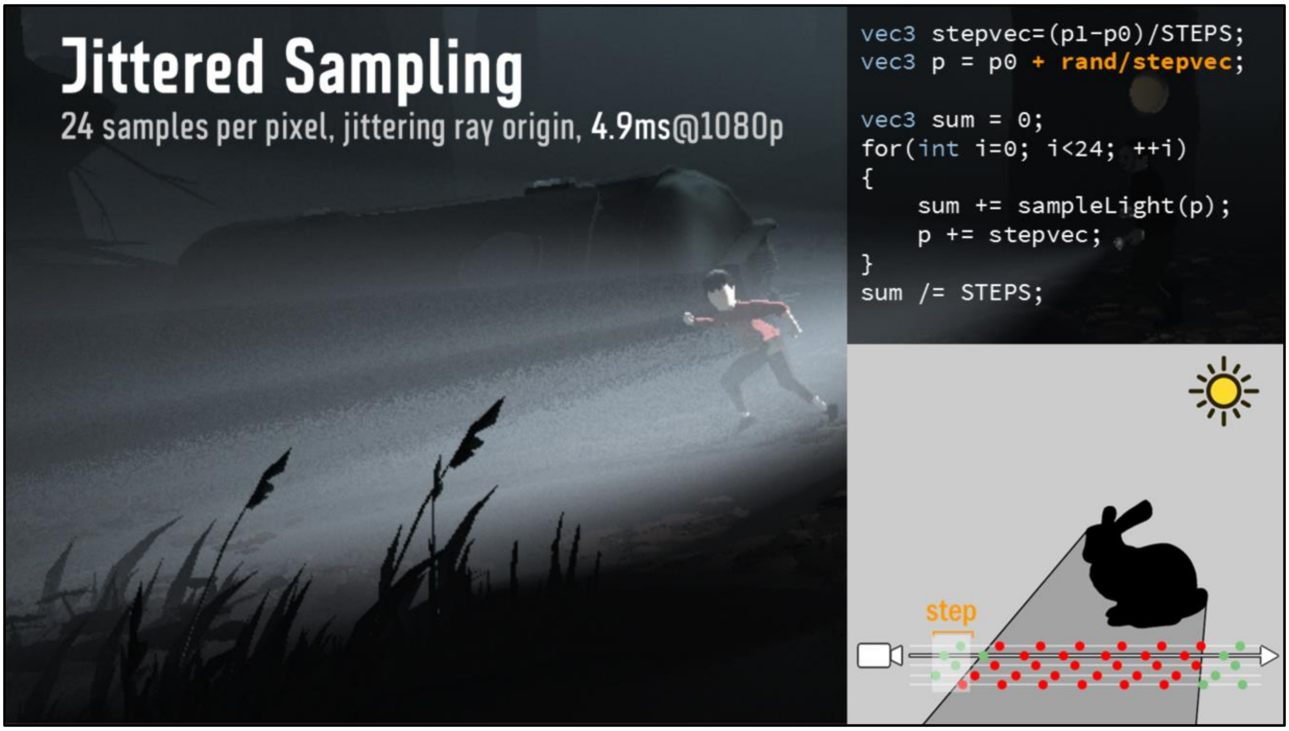
Uniform Sampling

24 samples per pixel, 3.6ms@1080p

```
vec3 stepvec=(p1-p0)/STEPS;
vec3 p = p0;

vec3 sum = 0;
for(int i=0; i<24; ++i)
{
    sum += sampleLight(p);
    p += stepvec;
}
sum /= STEPS;
```

Down to 3.2ms which is still quite slow…
…and artefact case is severe stepping artefacts

Ugly AND slow…



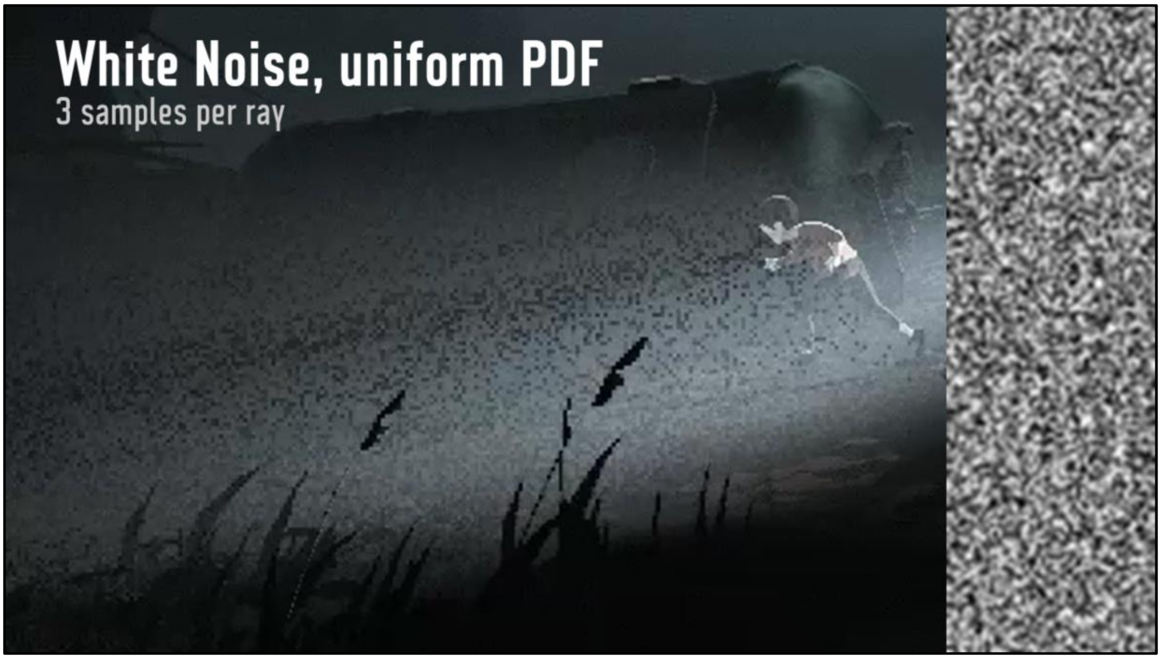
Jittered Sampling

24 samples per pixel, jittering ray origin, 4.9ms@1080p

```
vec3 stepvec=(p1-p0)/STEPS;
vec3 p = p0 + rand/stepvec;

vec3 sum = 0;
for(int i=0; i<24; ++i)
{
    sum += sampleLight(p);
    p += stepvec;
}
sum /= STEPS;
```

slower due to worse T$ than uniform sampling
...still needs an impractical number of samples, making it too slow…

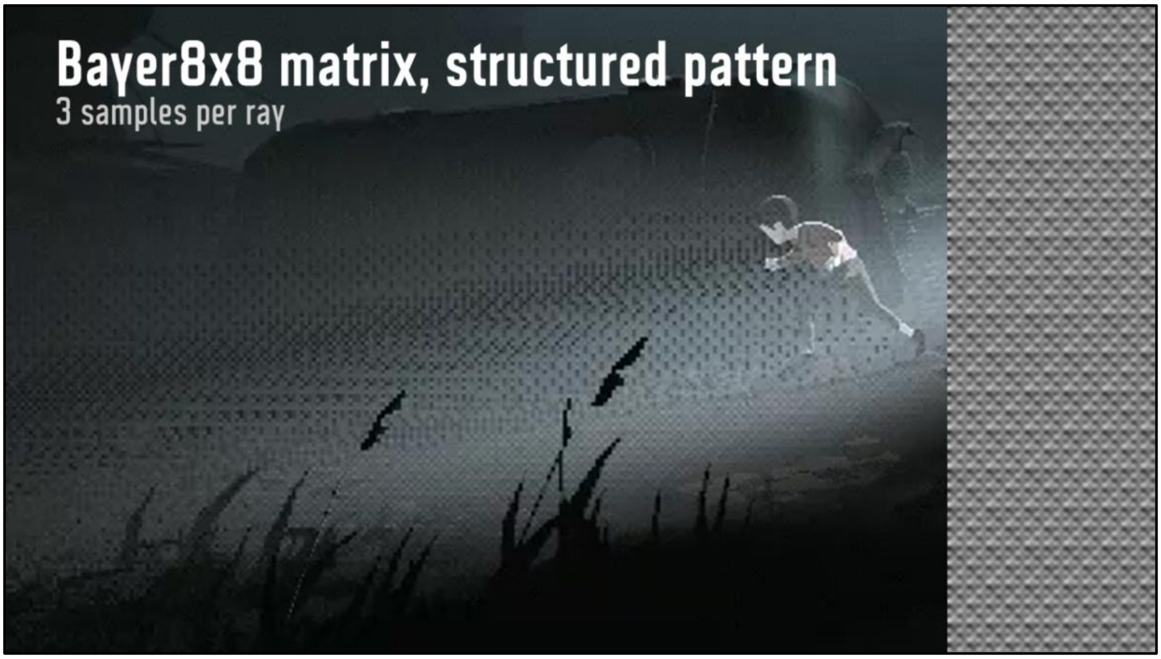
But artefacts are better, human eye quite forgiving towards noise… **interesting property**! Let us explore that a bit further…



White Noise, uniform PDF

3 samples per ray

same image as before, but with 3 samples instead of 32… shadow effect barely recognizable
massive undersampling, loads of noise

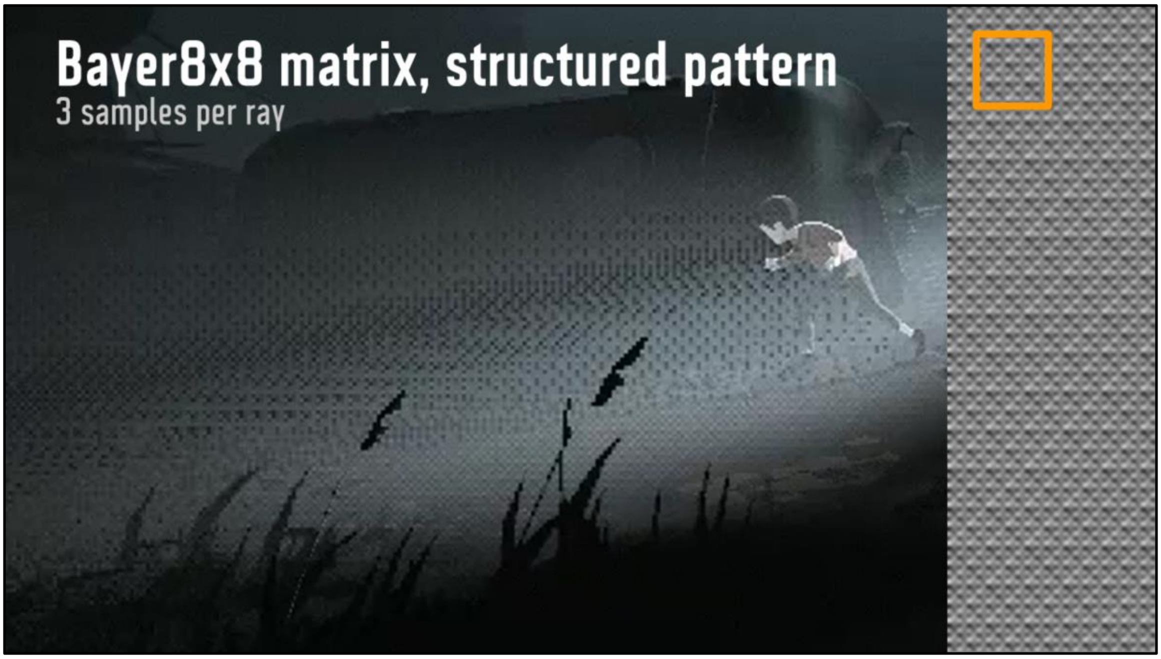


Bayer8x8 matrix, structured pattern

3 samples per ray

Less noise because entirely homogenous pattern - no random!

(we actually used Interleaved Gradient Noise for a while as very fast to compute, but very hard to filter out moiré-like artefacts caused by pattern... Same with bayer)



Bayer8x8 matrix, structured pattern

3 samples per ray

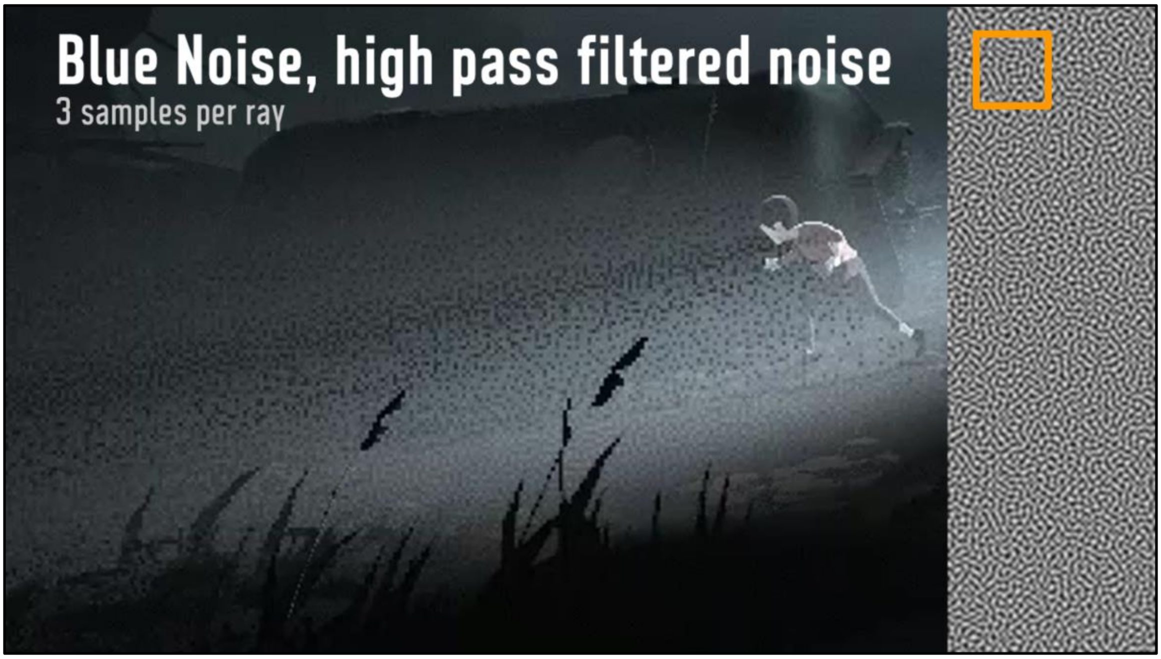
Better sampling - all values represented in small region
BUT - eye NOT forgiving to structure
SO want
- No structure
- Quick changes within small region => a high pass filter

--
Also, within a local neighborhood, all values are represented
(with noise, maybe they were, maybe they weren’t)

It is a goal to have as **much sampling-diversity within as local a region** as possible.

A **Bayer-matrix is optimal** in this regard, but is structured…
causes moiré-like patterns, which are very noticeable
very hard to filter the structure out of the image again.
So what we need is something without structure, but with all values represented in a small area. So a high-frequency noise.

Bayer matrix property from wiki:
”average distance between two successive numbers in the map is as large as possible,”



Blue Noise, high pass filtered noise

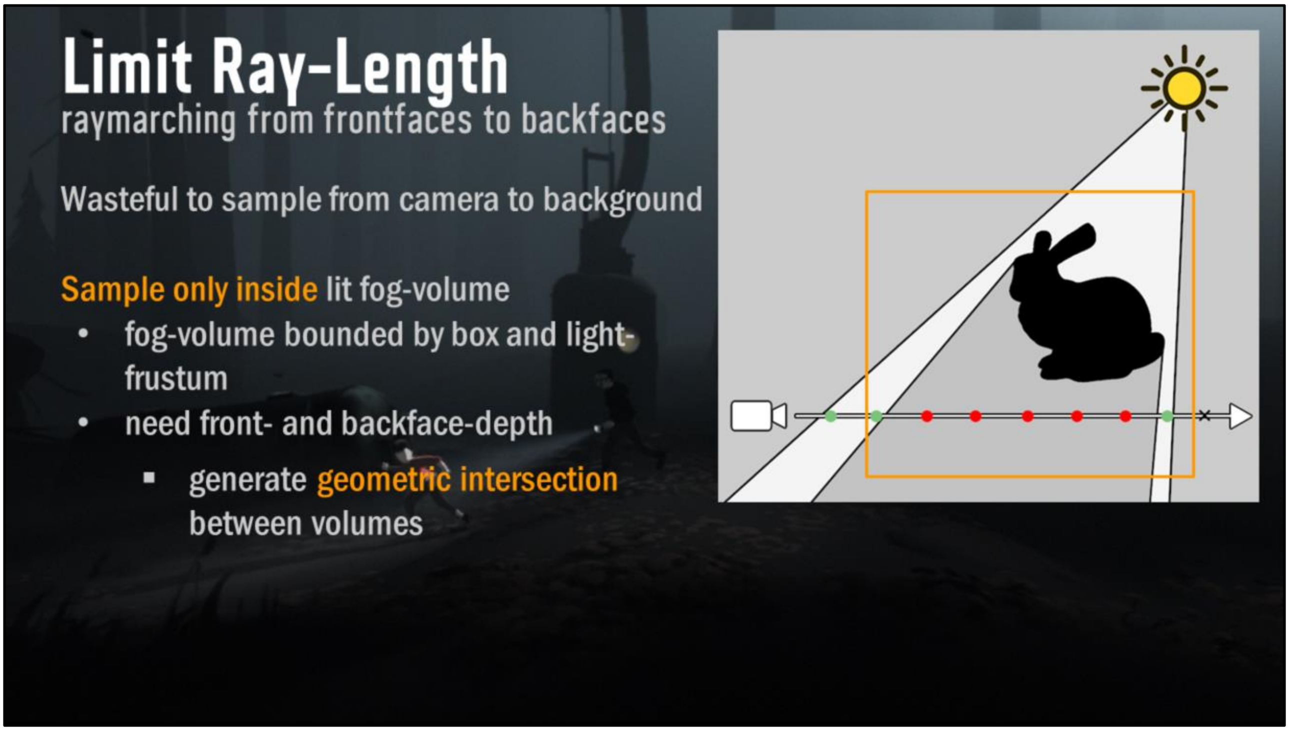
3 samples per ray

high-pass filtered uniform-PDF noise (blue noise)
no structure
more values represented in a local neighborhood

--
Artefacts occur in undersampled areas, low-frequency areas ok
Uniform noise has no patterns, but heavy noise
Bayer pattern is the most uniform, best sampling, but causes noticeable patterns in undersampled areas (when signal aligns with pattern). Moiré patterns ensue.
(...about 5% slower than Interleaved Gradient Noise, but we can get away with way less samples without artefacts.

Blue noise gives decent sampling, while falling back to noise for artefacts in undersampled regions Can get away with much fewer samples \o/

http://excedrin.media.mit.edu/wp-content/uploads/sites/10/2013/07/spie97newbern.pdf



Limit Ray-Length

raymarching from frontfaces to backfaces

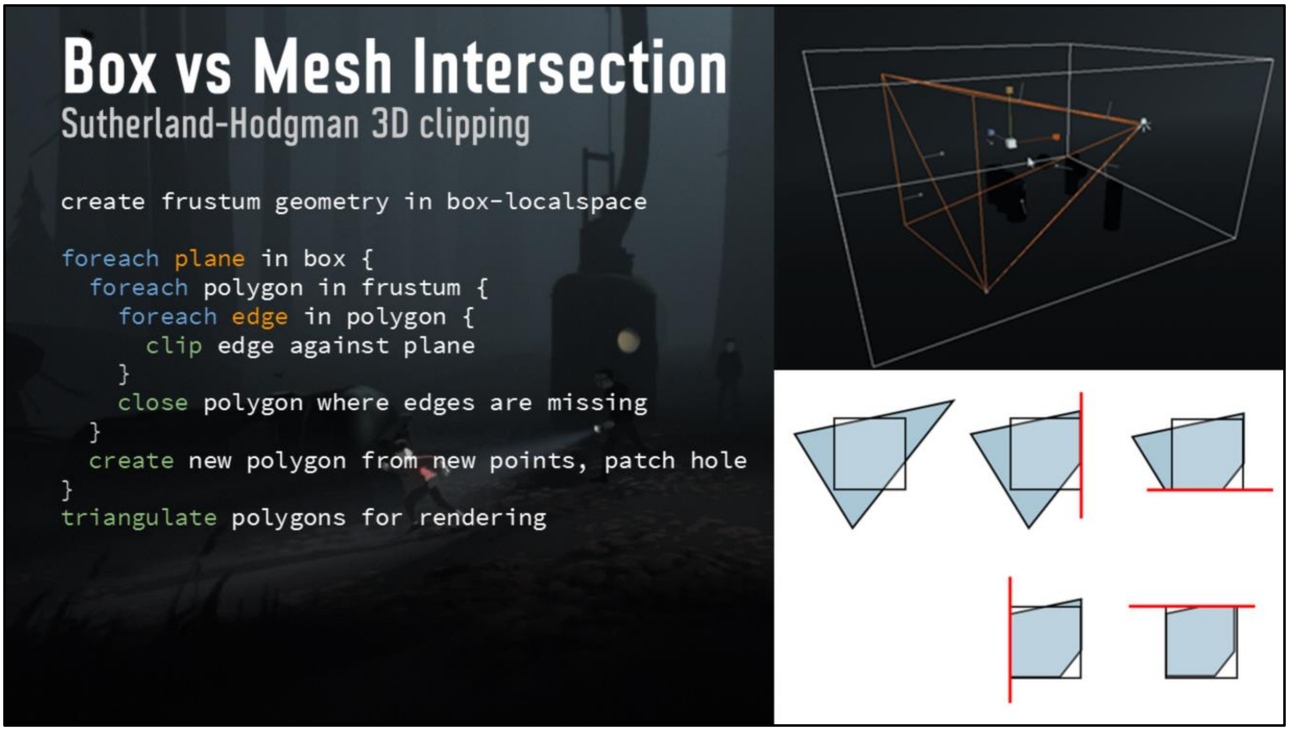
Wasteful to sample from camera to background

Sample only inside lit fog-volume

- fog-volume bounded by box and light-frustum
- need front- and backface-depth
  - generate **geometric intersection** between volumes

How we distribute our samples
Local fog-volumes are defined in the editor by a box.

-
additive effect - we add the light caught by fog in the room, we do not add fog locally.
Global fog should match (but we leave it up to artists)



Box vs Mesh Intersection

Sutherland-Hodgman 3D clipping

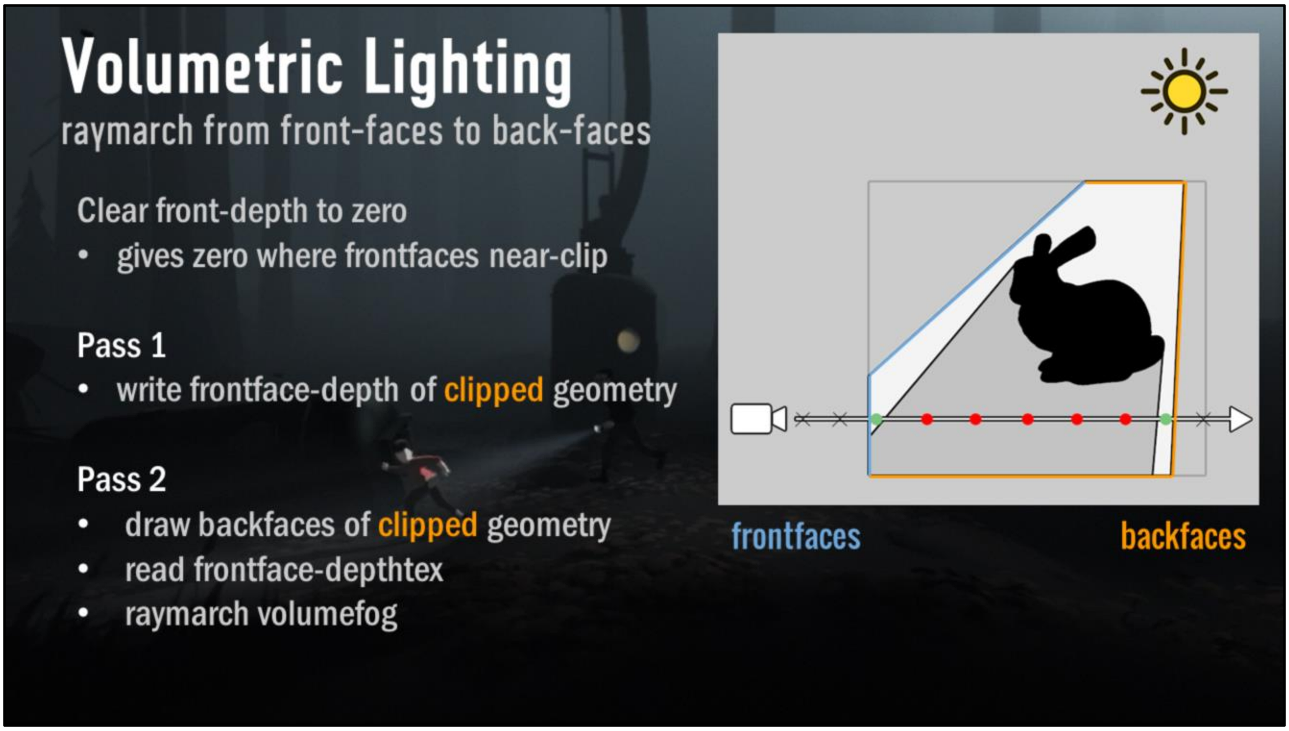
```
create frustum geometry in box-localspace

foreach plane in box {
  foreach polygon in frustum {
    foreach edge in polygon {
      clip edge against plane
    }
    close polygon where edges are missing
  }
  create new polygon from new points, patch hole
}
triangulate polygons for rendering
```

Clipping with all planes and patching up the holes.

Recalculate only when relative transform between fog-volume and light changes (ie mostly static)

Built this algorithm, then realised it was identical to 3D Sutherland-Hodgman… not overly well described online.
Implementation uses static lists of polygons. Plane-intersections boil down to a 1D “InverseLerp”, so very fast.

Still, only do calculation if light/box transformation changes.



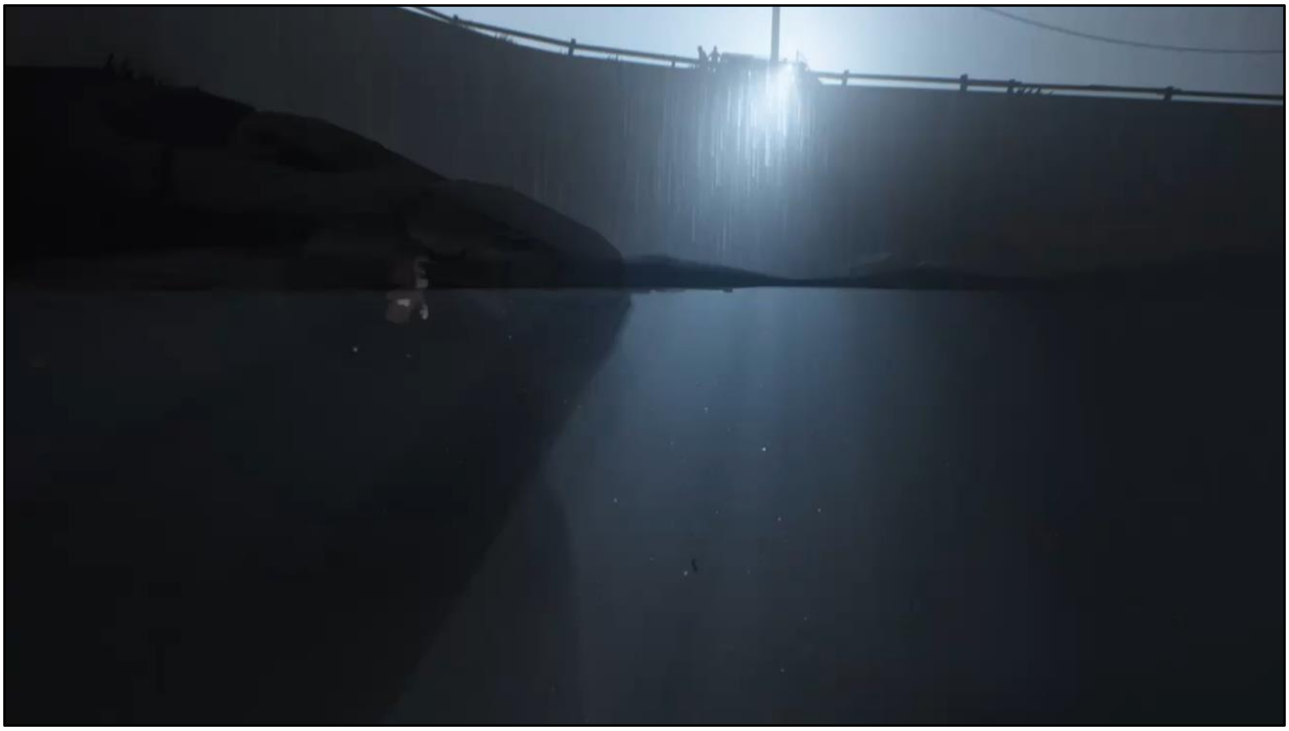
Volumetric Lighting

raymarch from front-faces to back-faces

Clear front-depth to zero
- gives zero where frontfaces near-clip

Pass 1
- write frontface-depth of clipped geometry

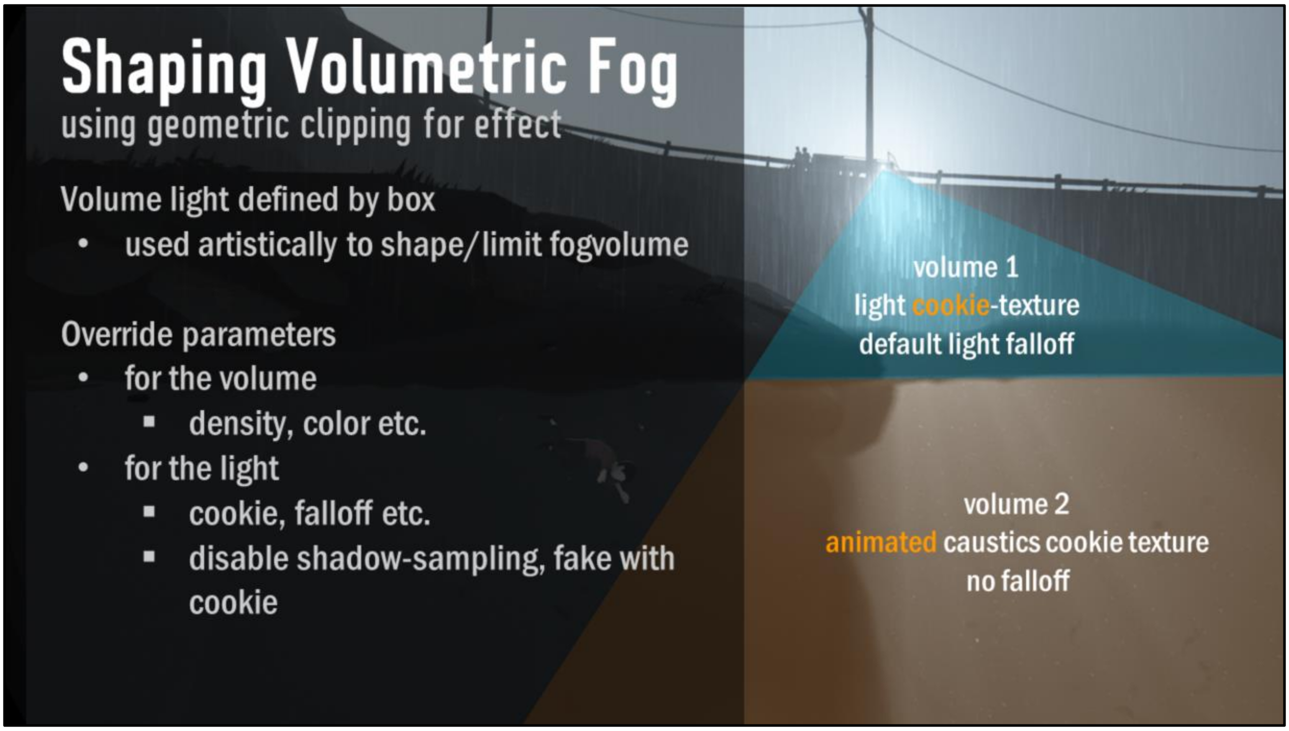
Pass 2
- draw backfaces of clipped geometry
- read frontface-depthtex
- raymarch volumefog



using geometric clipping for effect

volume 1
light cookie-texture
default light falloff

volume 2
animated caustics cookie texture
no falloff



Shaping Volumetric Fog

using geometric clipping for effect

Volume light defined by box

- used artistically to shape/limit fogvolume

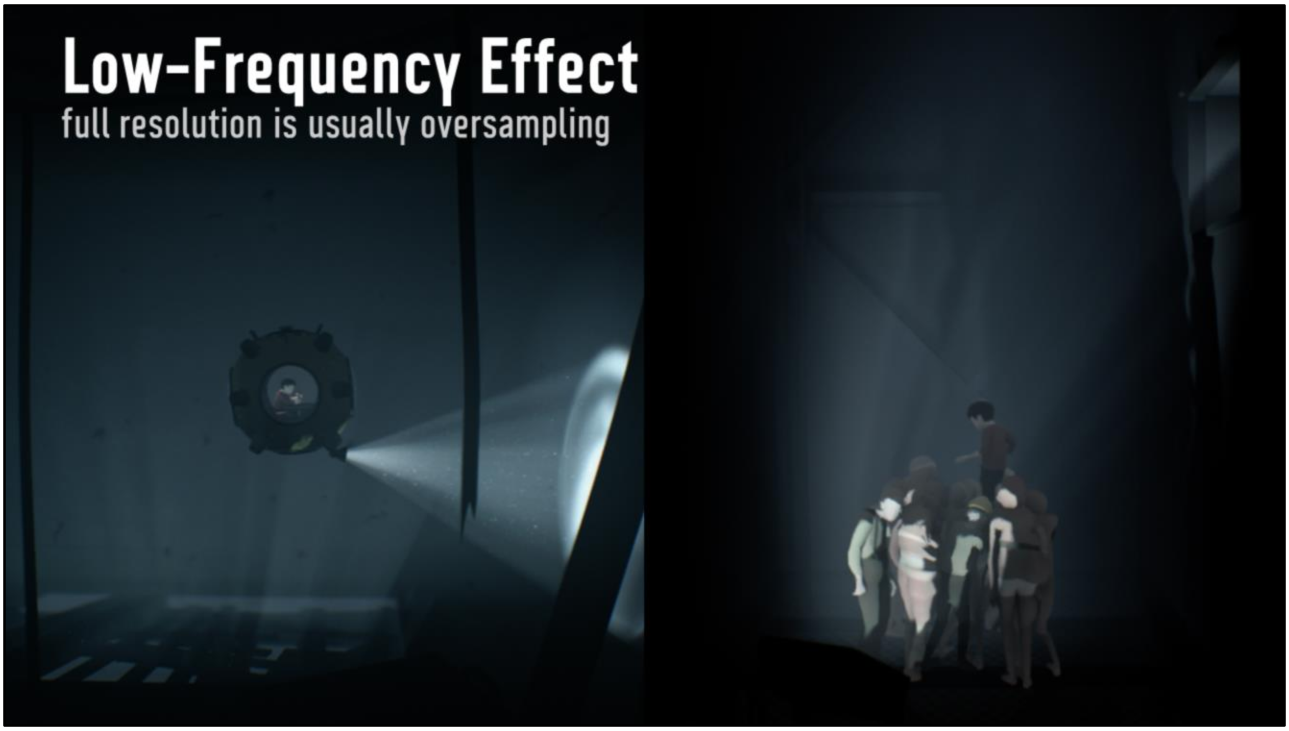
Override parameters

- for the volume
  - density, color etc.
- for the light
  - cookie, falloff etc.
  - disable shadow-sampling, fake with cookie

using geometric clipping for effect

volume 1
light cookie-texture
default light falloff

volume 2
animated caustics cookie texture
no falloff

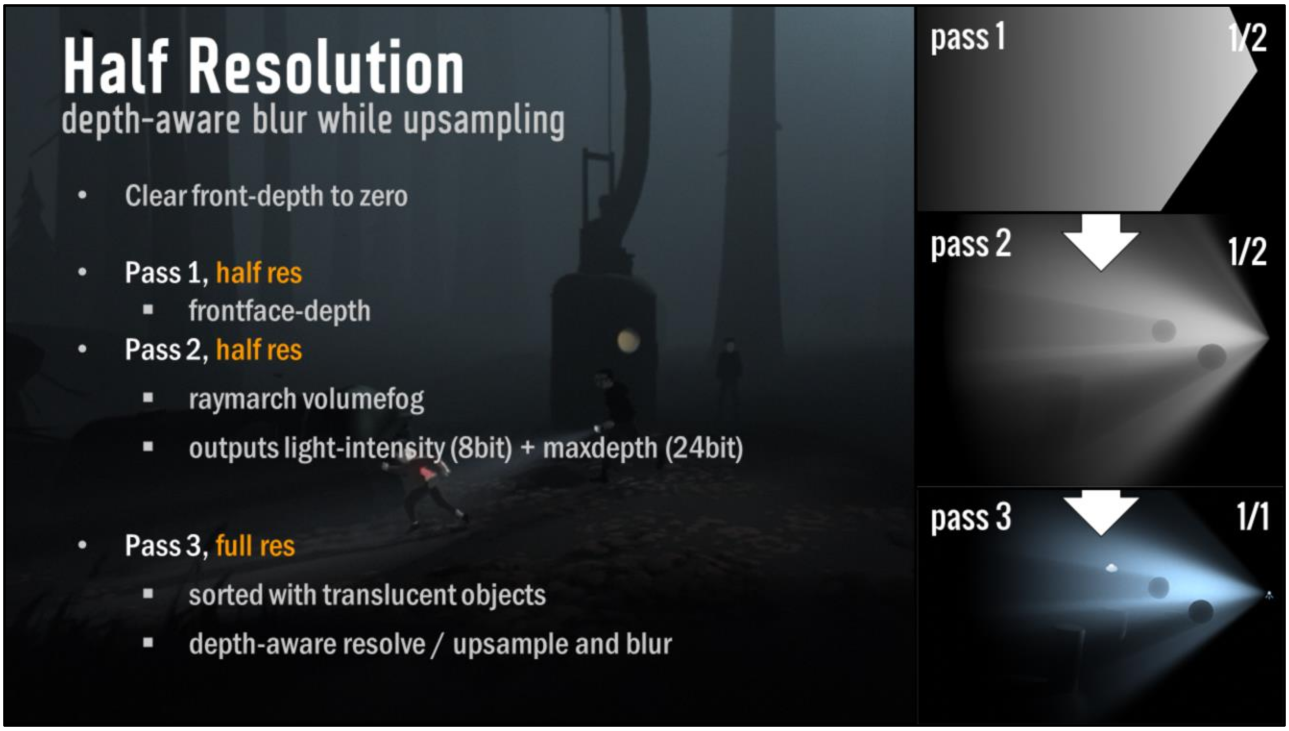


Low-Frequency Effect

full resolution is usually oversampling

Smooth effect, low frequency

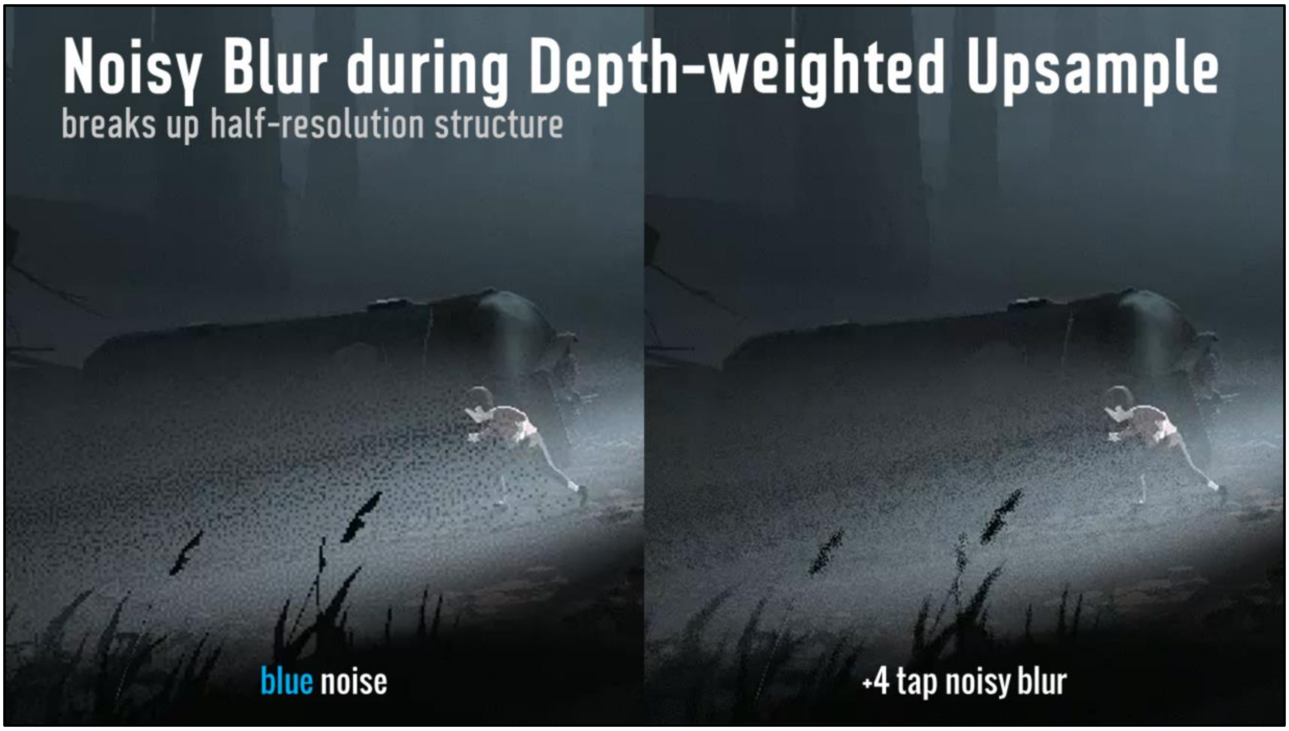
Not required to sample at full res



Half Resolution

depth-aware blur while upsampling

- Clear front-depth to zero
- Pass 1, half res
  - frontface-depth
- Pass 2, half res
  - raymarch volumefog
  - outputs light-intensity (8bit) + maxdepth (24bit)
- Pass 3, full res
  - sorted with translucent objects
  - depth-aware resolve / upsample and blur

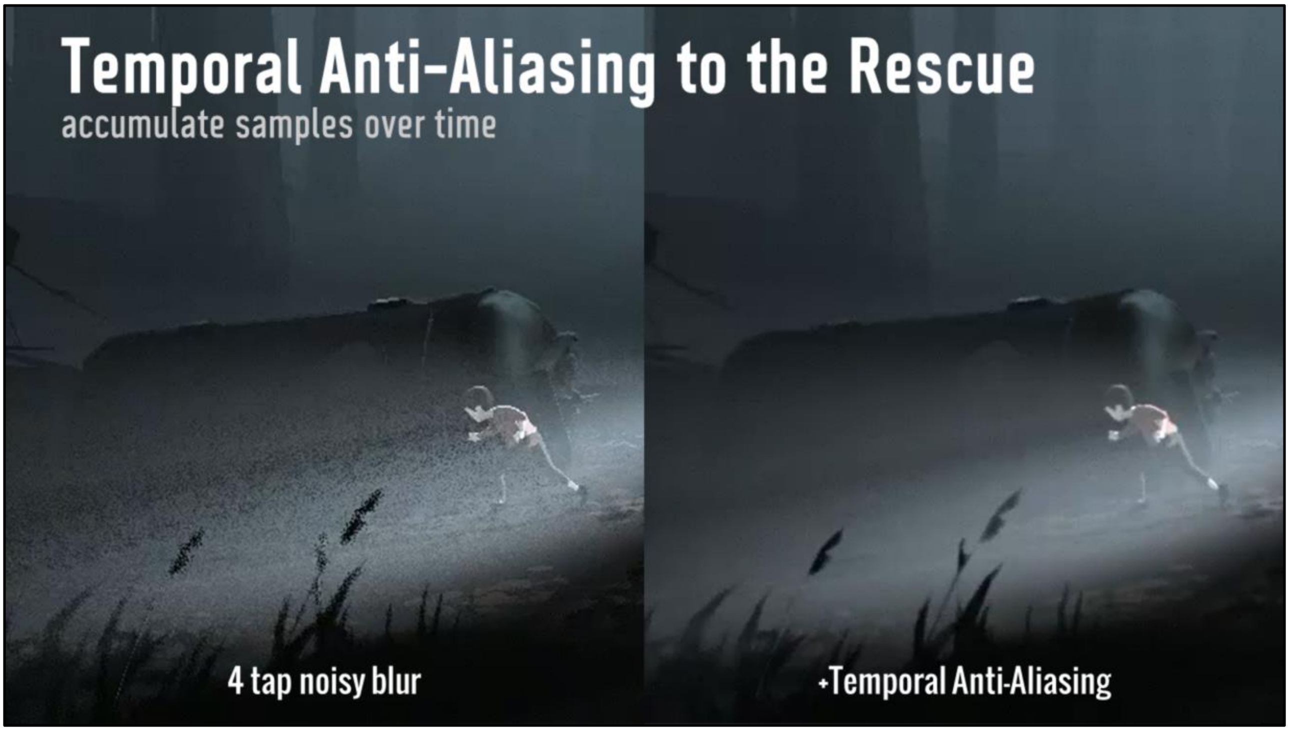


Noisy Blur during Depth-weighted Upsample

breaks up half-resolution structure

break up pattern using noisy blur
(depth-discard per sample)

insight from ssrt-presentation by DICE
http://www.frostbite.com/2015/08/stochastic-screen-space-reflections/

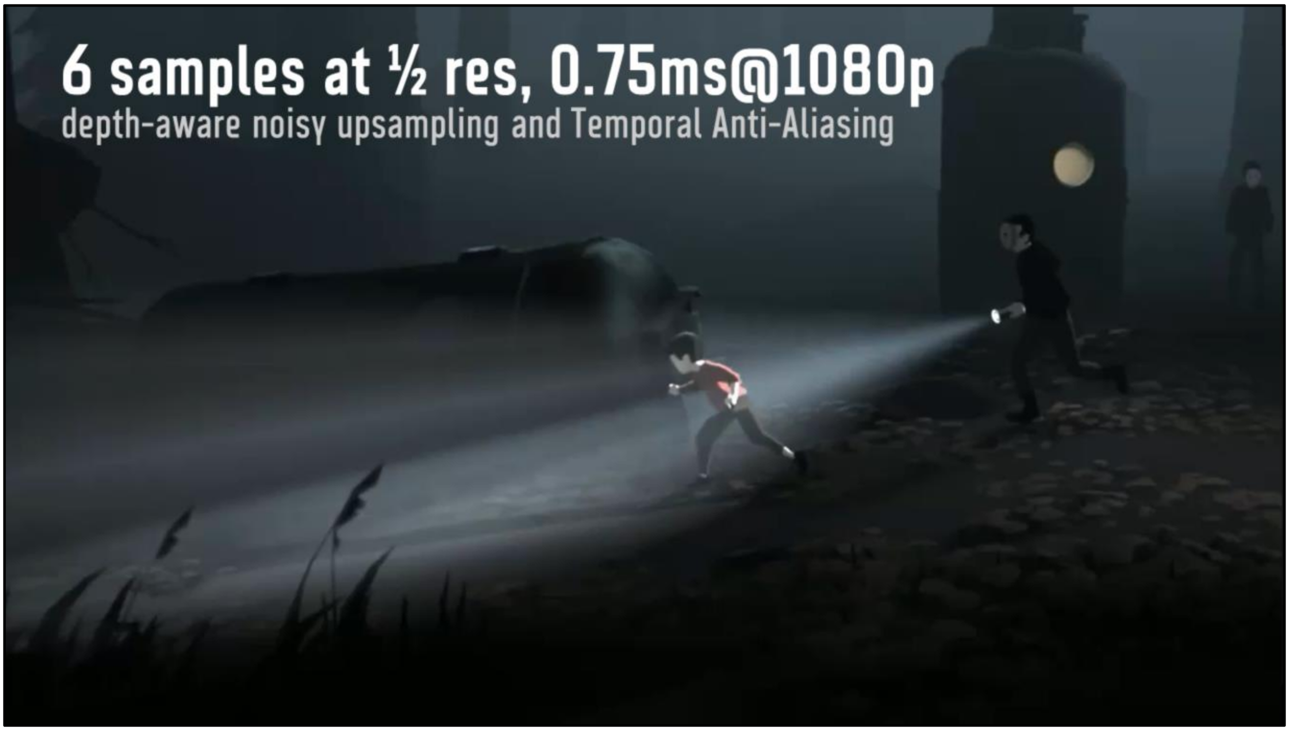


Temporal Anti-Aliasing to the Rescue

accumulate samples over time

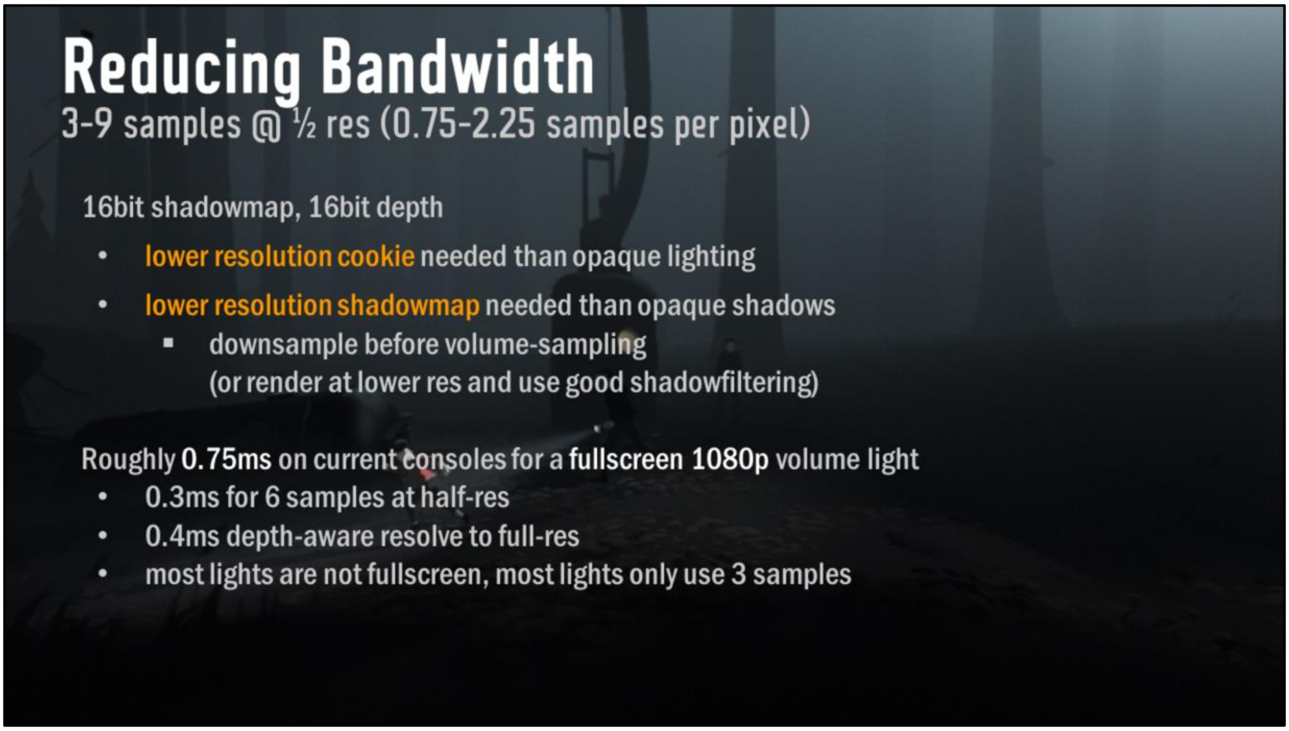
The real reason we are breaking up the structure, is because the Temporal Anti-Aliasing looks at the neighborhood, and half-res details confuse it.

This is just our regular post-effect temporal anti-aliasing working it’s magic, nothing special added for this.



6 samples at ½ res, 0.75ms@1080p

depth-aware noisy upsampling and Temporal Anti-Aliasing



Reducing Bandwidth

3-9 samples @ ½ res (0.75-2.25 samples per pixel)

16bit shadowmap, 16bit depth

- lower resolution cookie needed than opaque lighting
- lower resolution shadowmap needed than opaque shadows
  - downsample before volume-sampling
    (or render at lower res and use good shadowfiltering)

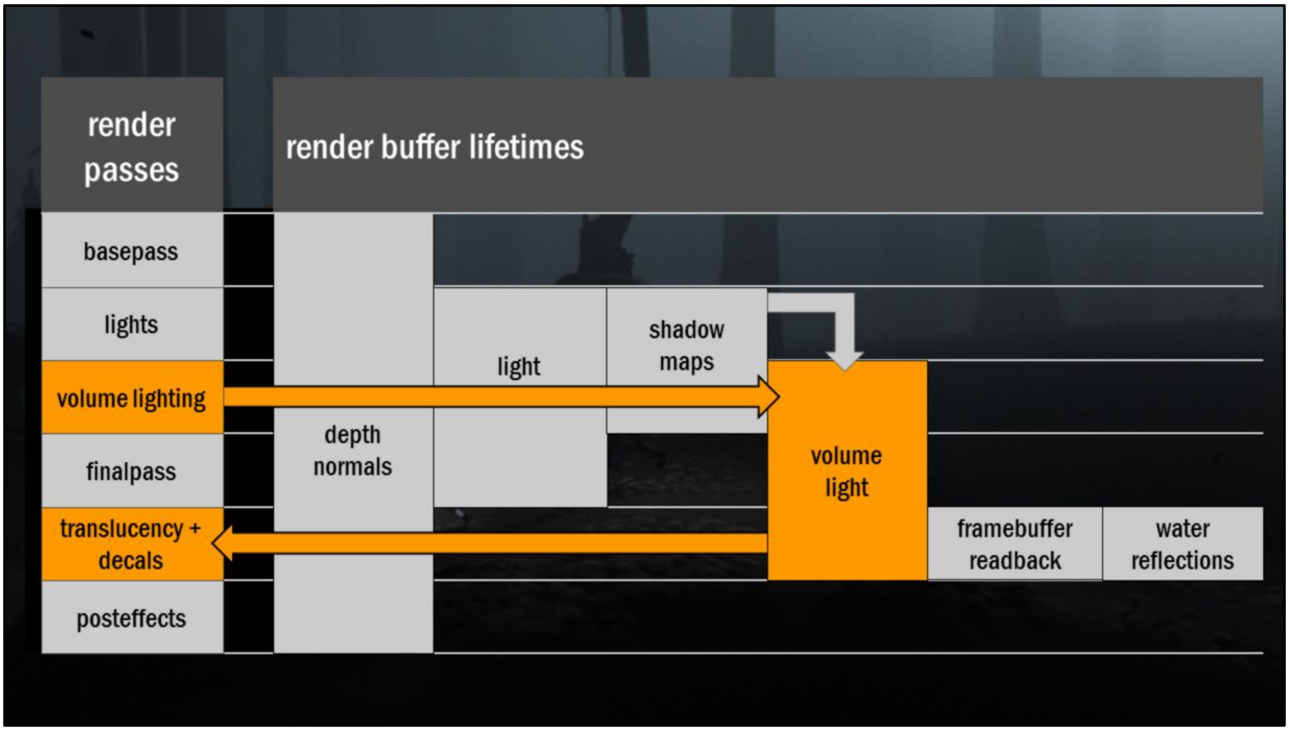
Roughly **0.75ms** on current consoles for a **fullscreen 1080p** volume light

- 0.3ms for 6 samples at half-res
- 0.4ms depth-aware resolve to full-res
- most lights are not fullscreen, most lights only use 3 samples

The reason why we have been **obsessing** about samples is that this technique is primarily bandwidth bound from the potentially many texture-samples.

https://developer.nvidia.com/sites/default/files/akamai/gameworks/samples/DeinterleavedTexturing.pdf

Future work: cookie importance sampling (e.g. equi-angular sampling for spotlights)
Future work: 3D blue-noise to improve variation in sampling-distribution over time



render passes

render buffer lifetimes

basepass

lights

volume lighting

finalpass

translucency + decals

posteffects

depth normals

light

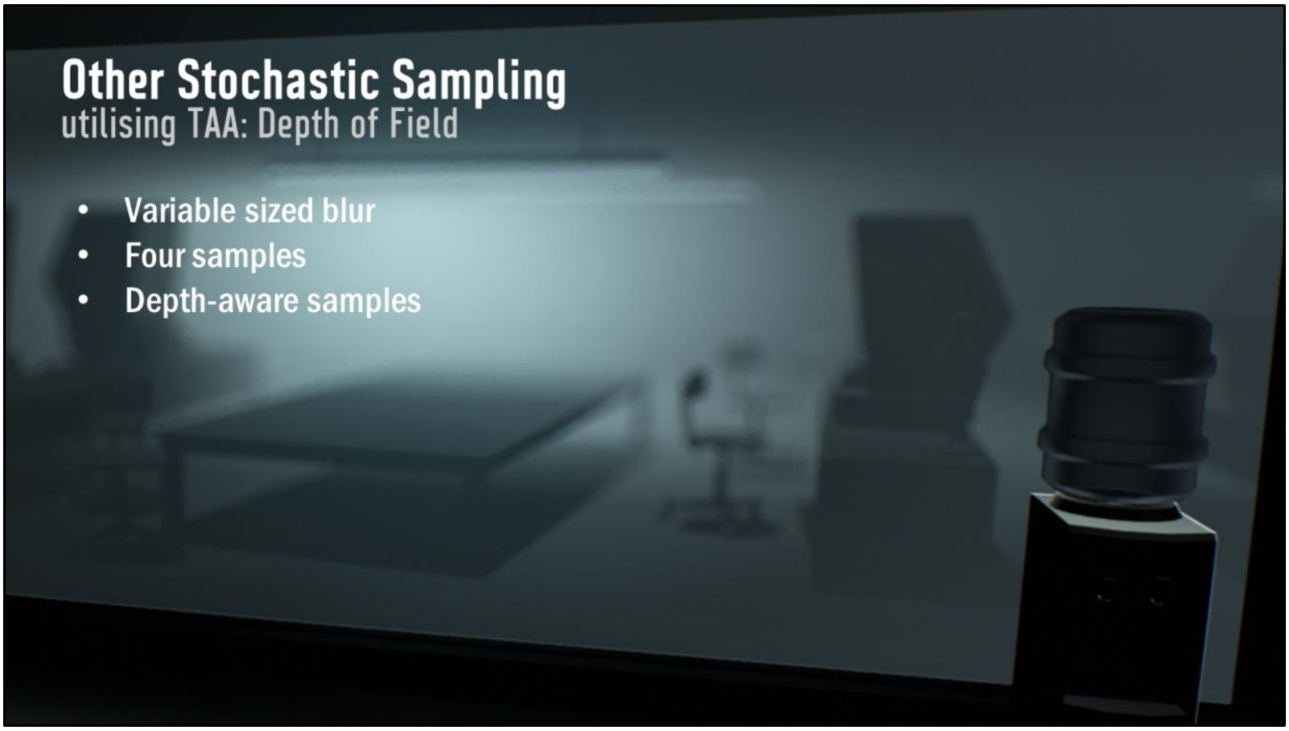
shadow maps

volume light

framebuffer readback

water reflections

Select shadowmaps are kept around after light-pass.
Half-res volumelight buffer is saved for use in translucency pass, where the fog-geometry is re-rendered and sorted with other translucencies.



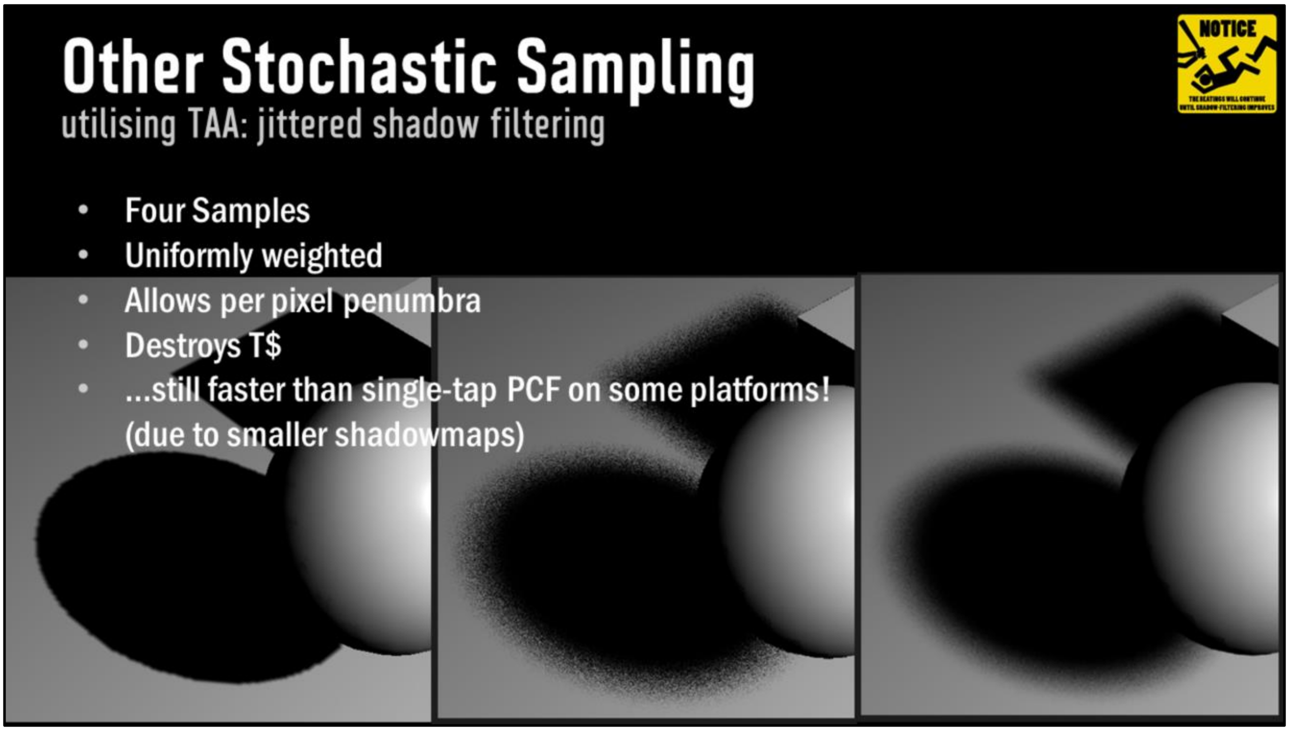
Other Stochastic Sampling
utilising TAA: Depth of Field

- Variable sized blur
- Four samples
- Depth-aware samples

We have several effects that play into the hand of TAA, undersampling the effect and letting the TAA do integration over time.

Other Stochastic Sampling

utilising TAA: jittered shadow filtering



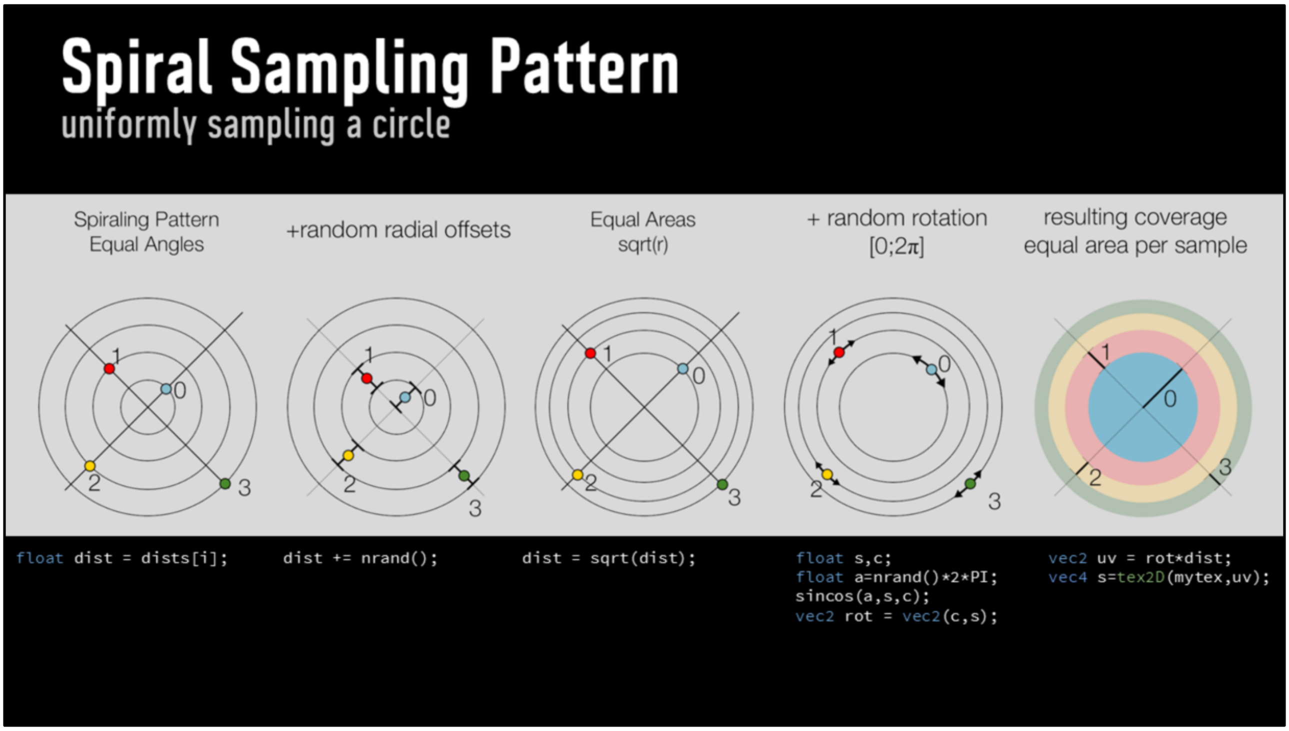
Other Stochastic Sampling

utilising TAA: jittered shadow filtering

- Four Samples
- Uniformly weighted
- Allows per pixel penumbra
- Destroys T$
- ...still faster than single-tap PCF on some platforms! (due to smaller shadowmaps)

speed? Faster than single sample PCF, because the shadowmap can be smaller with a better filter.

Future work: Test blue-noise



Spiral Sampling Pattern

uniformly sampling a circle

Spiraling Pattern
Equal Angles

+random radial offsets

Equal Areas
sqrt(r)

+ random rotation
[0;2π]

resulting coverage
equal area per sample

```
float dist = dists[i];
```

```
dist += nrand();
```

```
dist = sqrt(dist);
```

```
float s,c;
float a=nrand()*2*PI;
sincos(a,s,c);
vec2 rot = vec2(c,s);
```

```
vec2 uv = rot*dist;
vec4 s=tex2D(mytex,uv);
```

Goto-sampling pattern, same sampling for gcube resolve, depth and shadows...

Boils down to covering an equal area with each sample

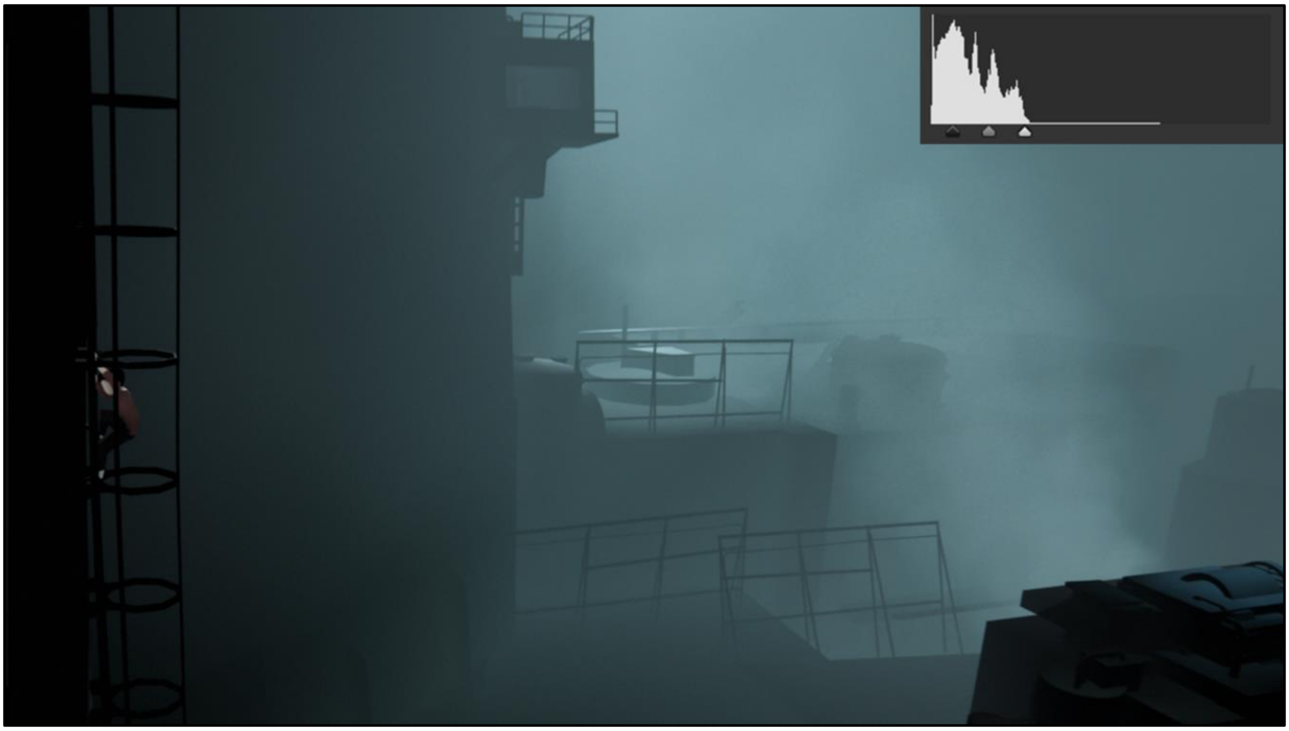
-

If you want a different filter, e.g. gaussian, you can adjust the sizes of the areas to achieve this, rather than assigning weights.
(to help your google along, the proper term for this would be *importance sampling*)

Dithering

see also http://www.loopit.dk/banding_in_games.pdf

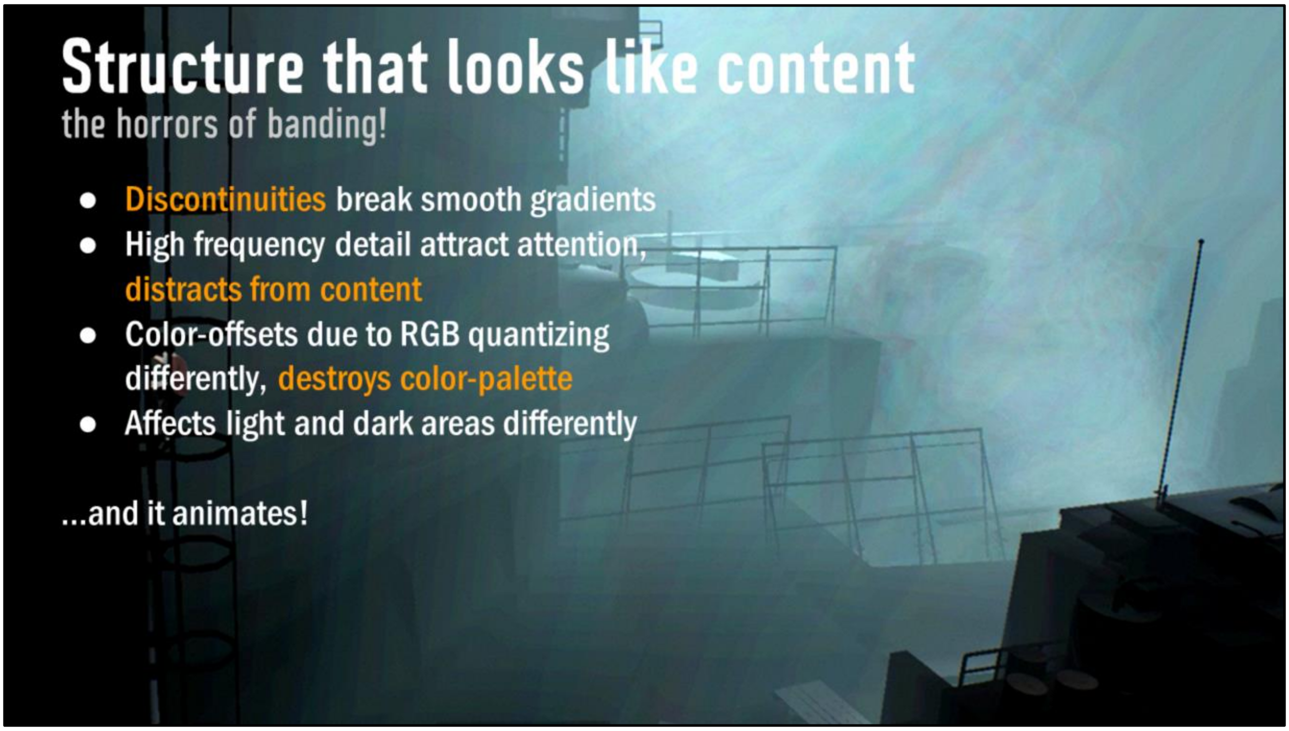
http://www.loopit.dk/banding_in_games.pdf



very soft image, lots of subtle color-tones

…this image has a lot of noise though – not really visible

(note: this image is dithered using a TPDF noise, not bluenoise)



Structure that looks like content

the horrors of banding!

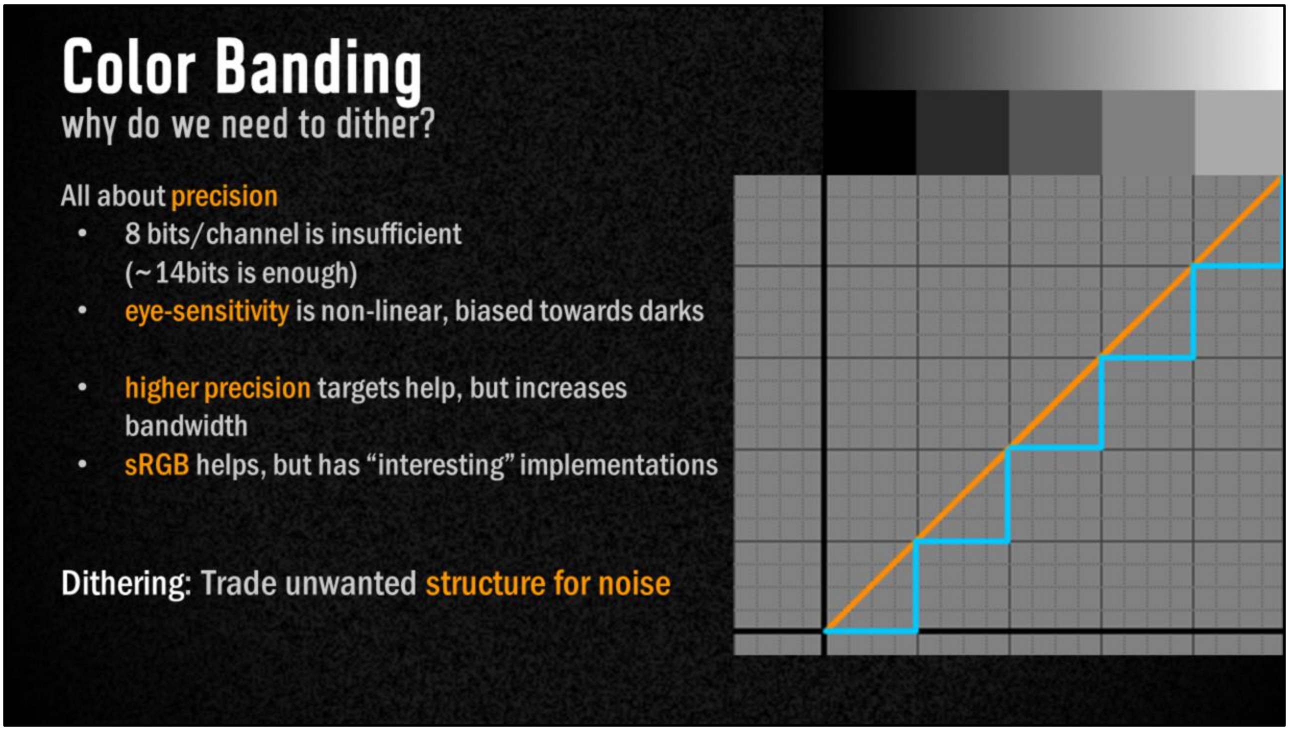
- Discontinuities break smooth gradients
- High frequency detail attract attention, distracts from content
- Color-offsets due to RGB quantizing differently, destroys color-palette
- Affects light and dark areas differently

...and it animates!

No dithering.

Note: tweaked colors even more. Not cutting off any values. On a decent enough monitor, the game would look like this without dithering.

Note: TAA also broken here…



Color Banding

why do we need to dither?

All about precision

- 8 bits/channel is insufficient (~14bits is enough)
- eye-sensitivity is non-linear, biased towards darks
- higher precision targets help, but increases bandwidth
- sRGB helps, but has “interesting” implementations

Dithering: Trade unwanted structure for noise

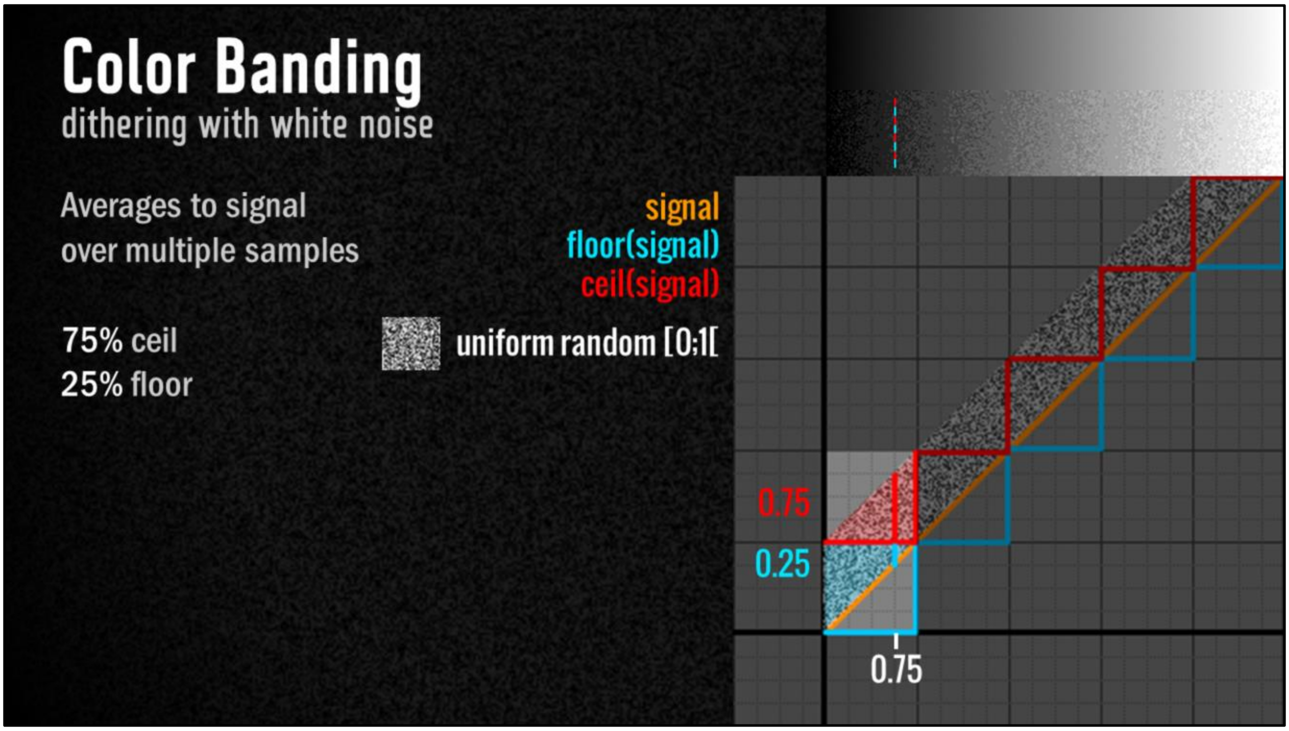
Stairstepping => banding
-
we represent values as discrete values.. eye notices high frequency changes. Also great at dark colors.

Use highest precision possible
Use sRGB or similar biasing towards darks on colors
If banding still visible, dither.
If full 16bit HDR pipeline: Dither when tonemapping to 8bit colortarget.
(almost always dither any 8bit write...)

See Charles Poynton’s Digital Video And HDTV Algorithms and Interfaces, p 269

moar details on banding:
http://loopit.dk/banding_in_games.pdf

moar details on srgb
http://download.microsoft.com/download/b/5/5/b55d67ff-f1cb-4174-836a-bbf8f84fb7e1/Picture%20Perfect%20-%20Gamma%20Through%20the%20Rendering%20Pipeline.zip



Color Banding

dithering with white noise

Averages to signal
over multiple samples

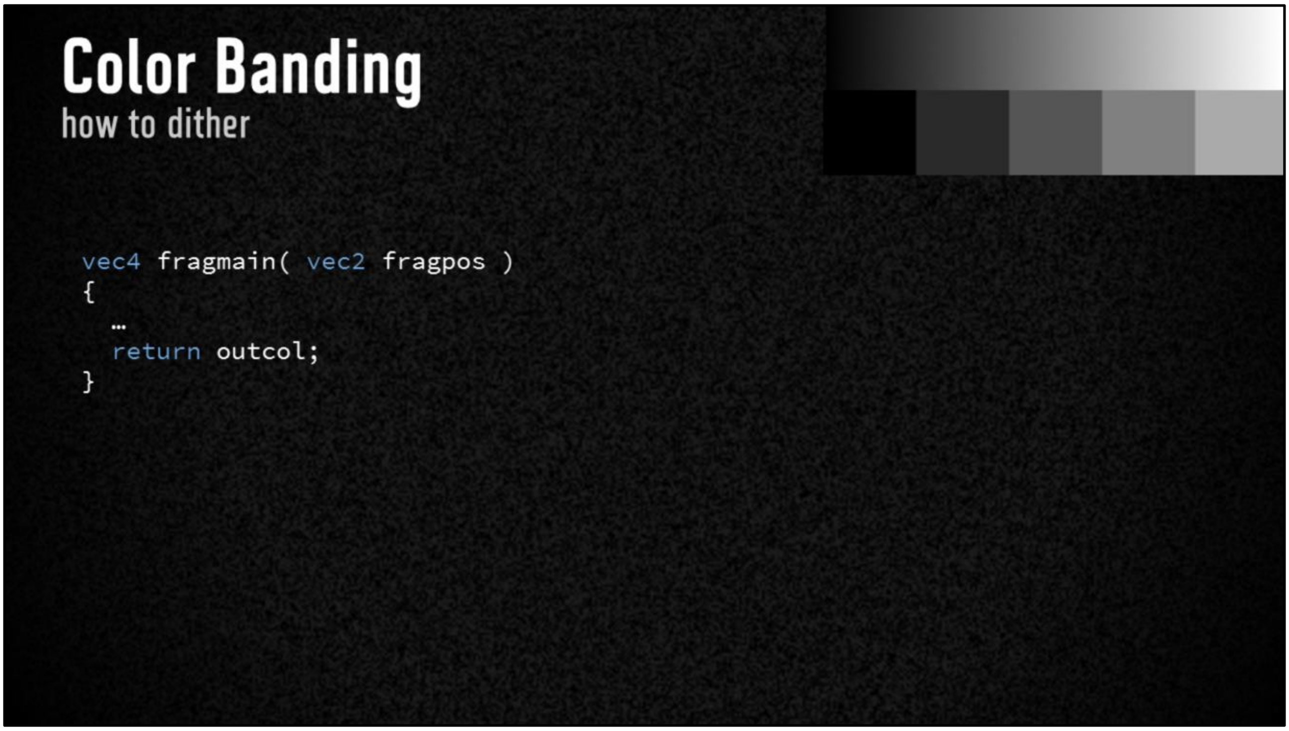
75% ceil
25% floor

Example of dithering
Add 1 bit of noise to signal before **quantisation**
Look at value 0.75

-

If we look at the accumulated error for a single value (integrating over the red/blue stippled line), the error will now cancel out as well as for entire signal, resulting in the original signal for any arbitrary single value

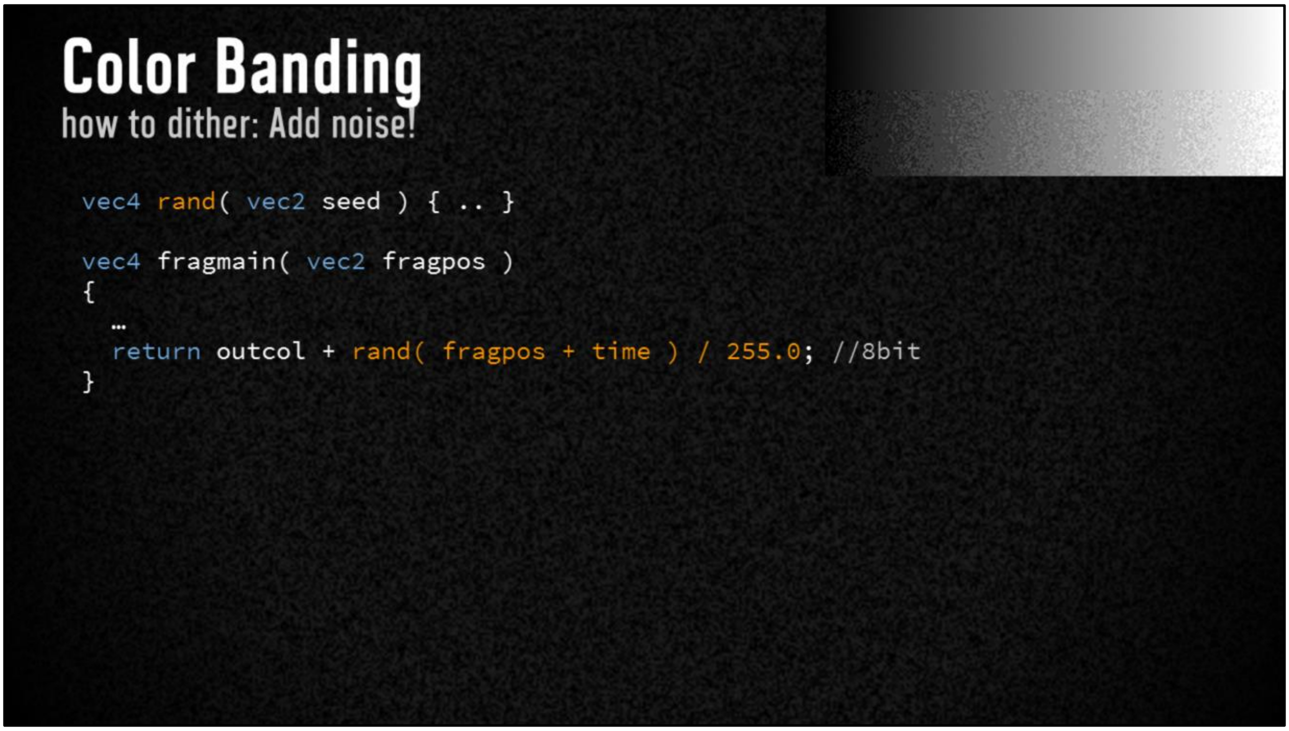
intuitively, since the noise-distribution is uniform, when integrating across the line shown the length of the line corresponds to the probability that the value will either round up or down
...integrating floor(f) / ceil(f) across this line, we’ll end up at the signal



Color Banding

how to dither

```
vec4 fragmain( vec2 fragpos )
{
  …
  return outcol;
}
```



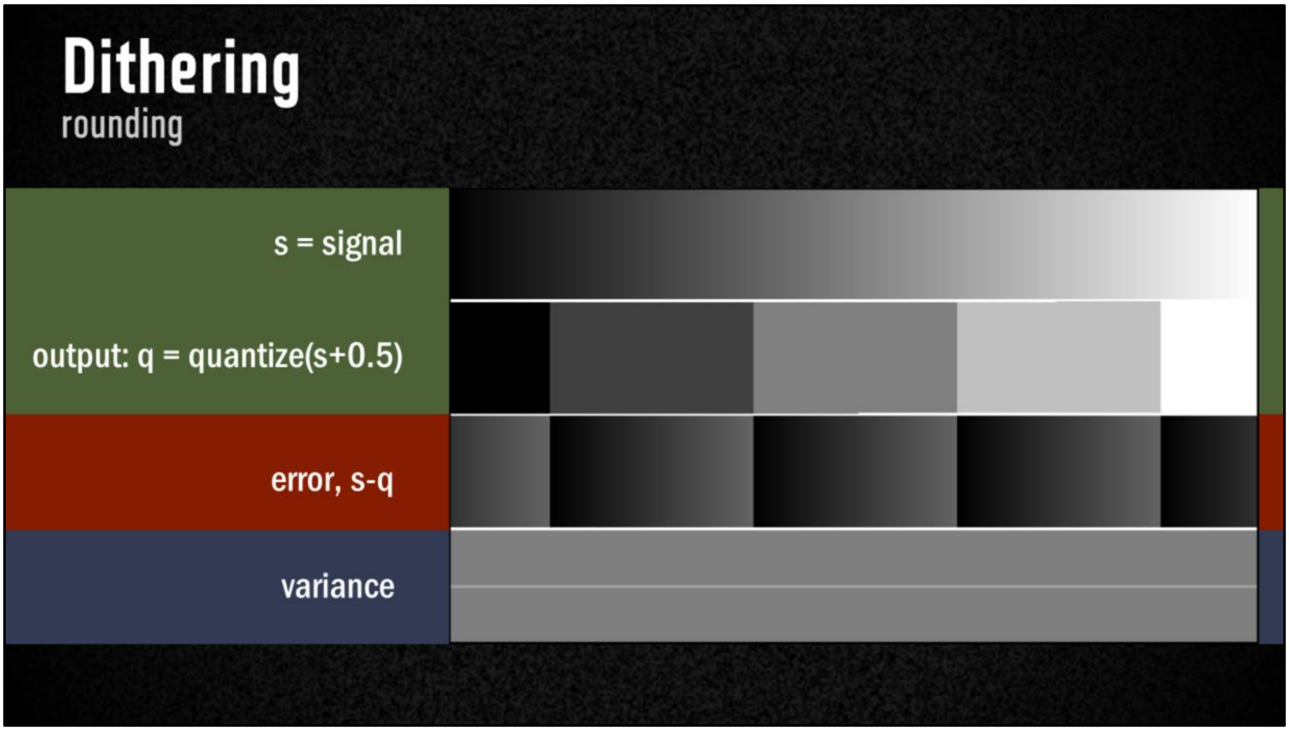
Color Banding

how to dither: Add noise!

```
vec4 rand( vec2 seed ) { .. }

vec4 fragmain( vec2 fragpos )
{
  …
  return outcol + rand( fragpos + time ) / 255.0; //8bit
}
```

divide by 1 LSB

Spectacularly easy to add
Potentially make your game look at lot better



Dithering

rounding

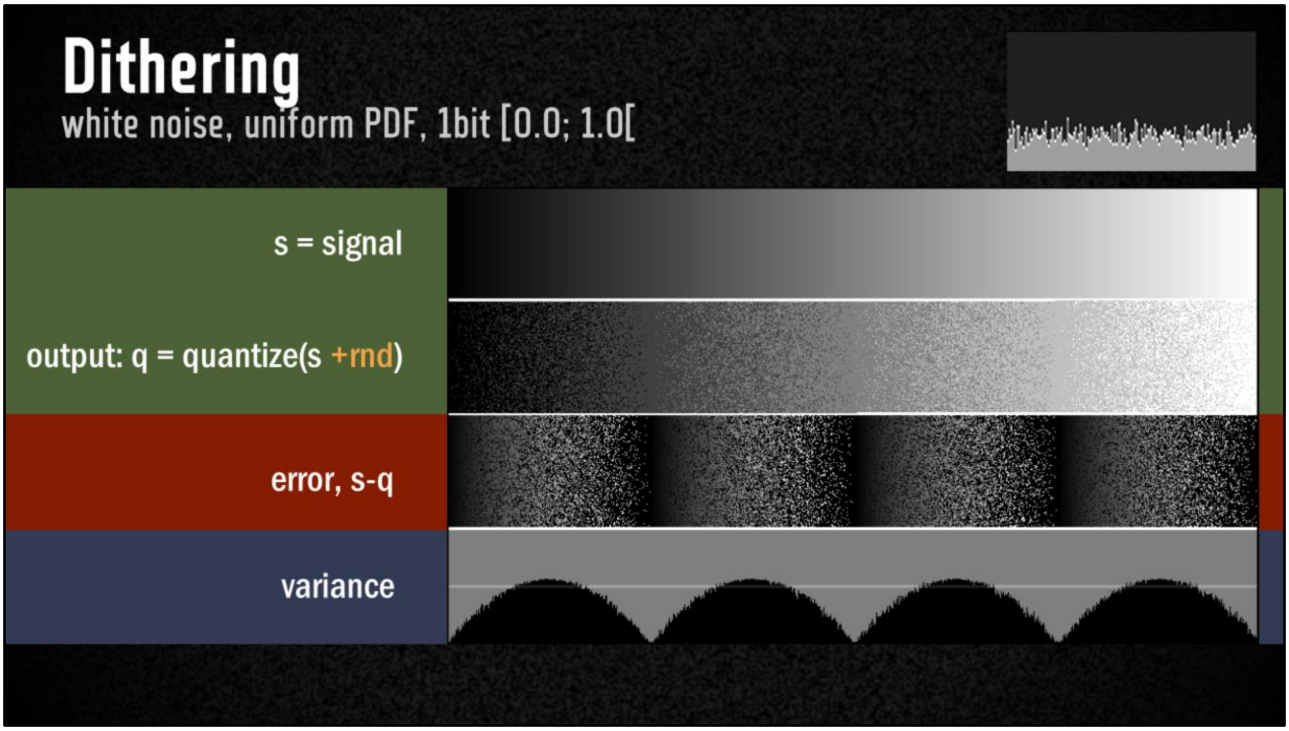
s = signal

output: q = quantize(s+0.5)

error, s-q

variance

https://www.shadertoy.com/view/ltBSRG



Dithering

white noise, uniform PDF, 1bit [0.0; 1.0[

s = signal

output: q = quantize(s +rnd)

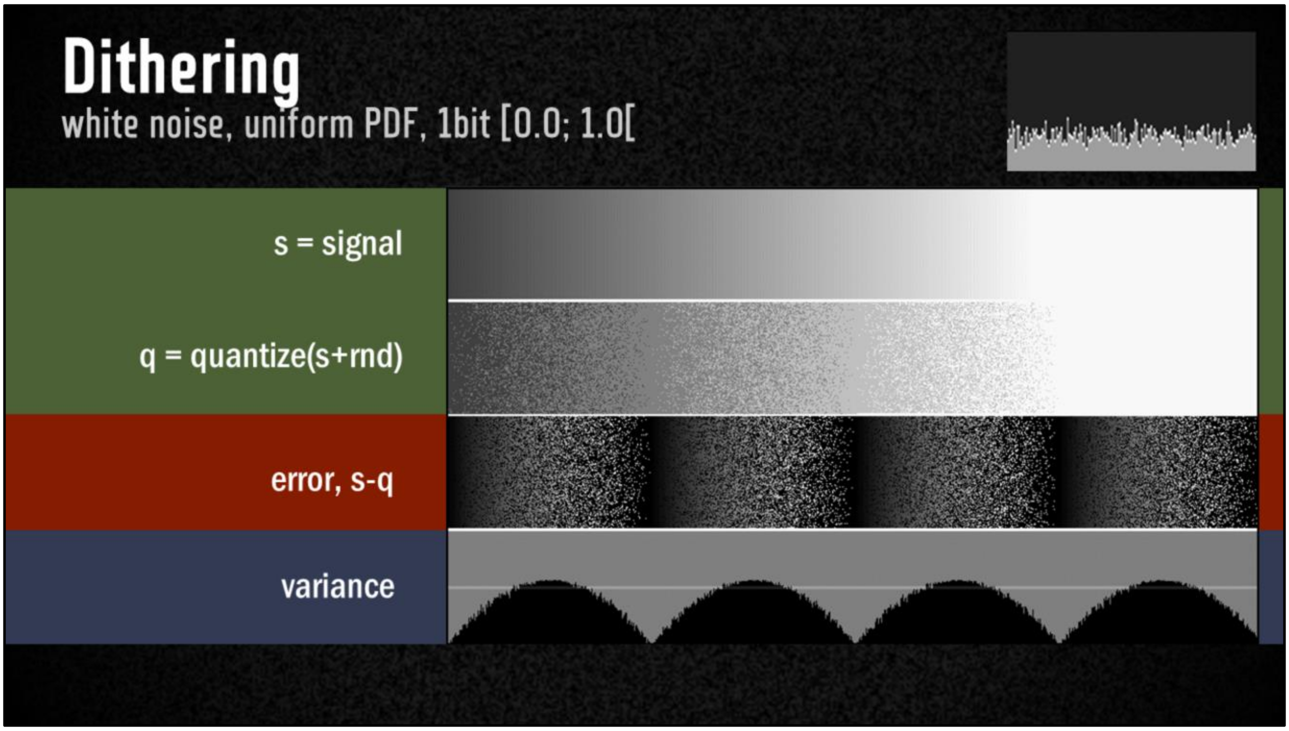
error, s-q

variance

...noise is uniformly distributed, but the resulting visual noise is NOT uniformly distributed - almost no noise near “correct” values (where the value crosses bit-borders and is truncated to value itself).

Variance: average of (q-mean)^2

https://www.shadertoy.com/view/ltBSRG

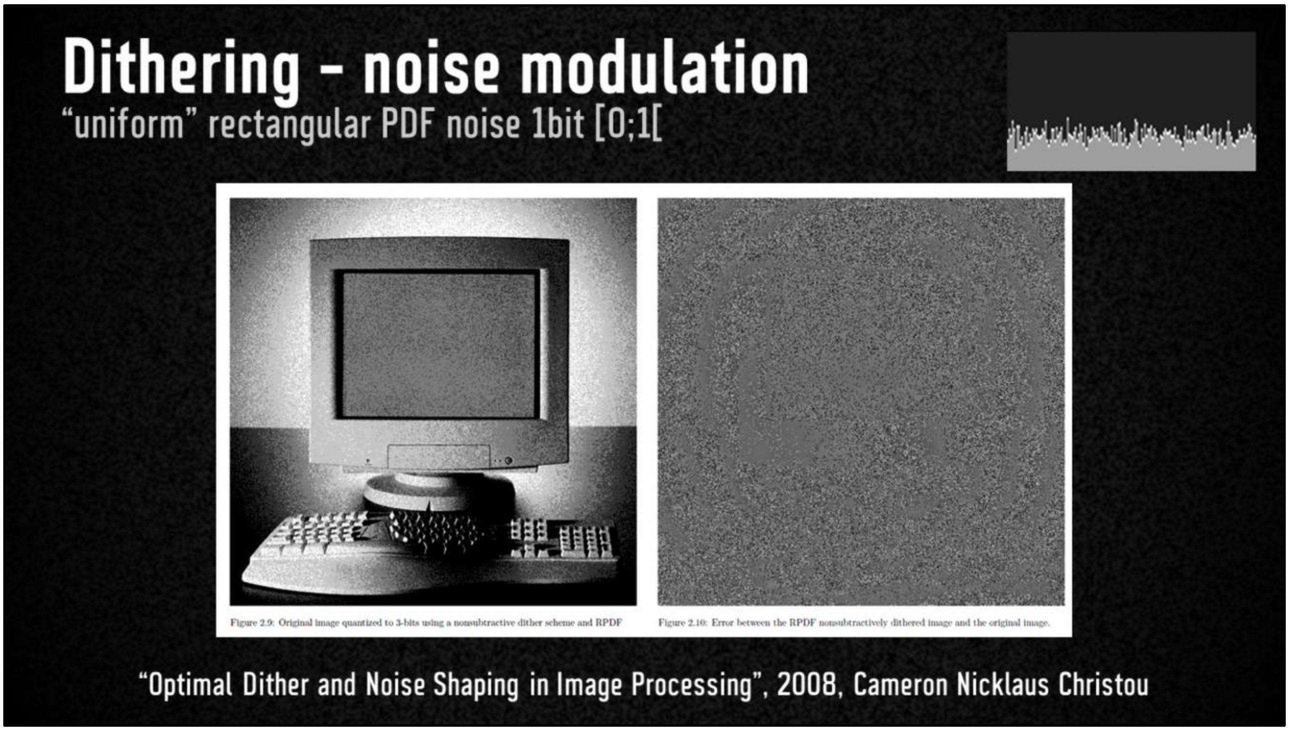


Dithering

white noise, uniform PDF, 1bit [0.0; 1.0[

quantisation banding is gone, but the amount of noise vary across the signal, causing bands with visibly less noise to appear

https://www.shadertoy.com/view/ltBSRG



Dithering - noise modulation

"uniform" rectangular PDF noise 1bit [0;1[

Figure 2.9: Original image quantized to 3-bits using a nonsubtractive dither scheme and RPDF

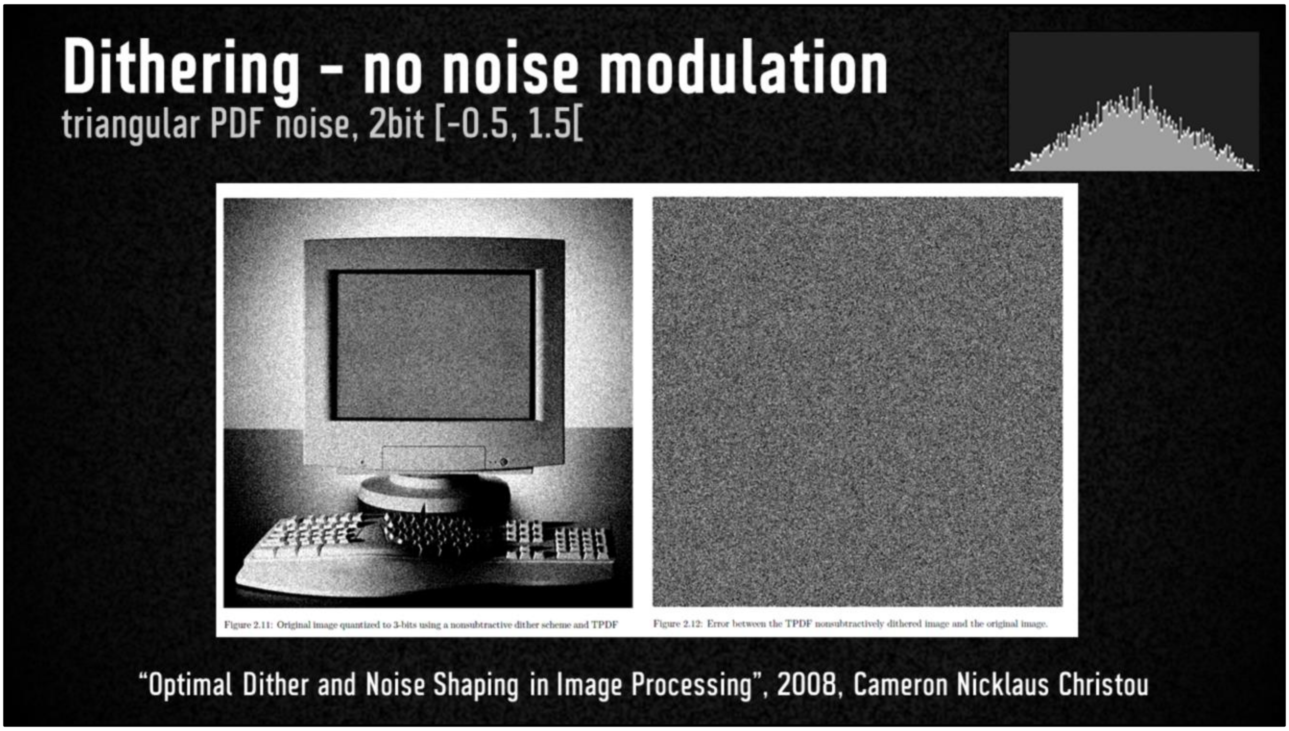
Figure 2.10: Error between the RPDF nonsubtractively dithered image and the original image.

"Optimal Dither and Noise Shaping in Image Processing", 2008, Cameron Nicklaus Christou

As it turns out, this is a property of the noise we are using - or rather, a statisticaly property of the distribution of the noise.
The phaenomenon is known as **noise-modulation**, which is the effect of the resulting error, after quantisation, being dependant on the signal

We would much prefer it being entirely uniform (as eye does not notice it then)

https://uwspace.uwaterloo.ca/bitstream/handle/10012/3867/thesis.pdf;jsessionid=74681FAF2CA22E754C673E9A1E6957EC?sequence=1



Dithering - no noise modulation

triangular PDF noise, 2bit [-0.5, 1.5[

Figure 2.11: Original image quantized to 3-bits using a nonsubtractive dither scheme and TPDF

Figure 2.12: Error between the TPDF nonsubtractively dithered image and the original image.

“Optimal Dither and Noise Shaping in Image Processing”, 2008, Cameron Nicklaus Christou

Luckily, the solution is to just use different distribution for the noise

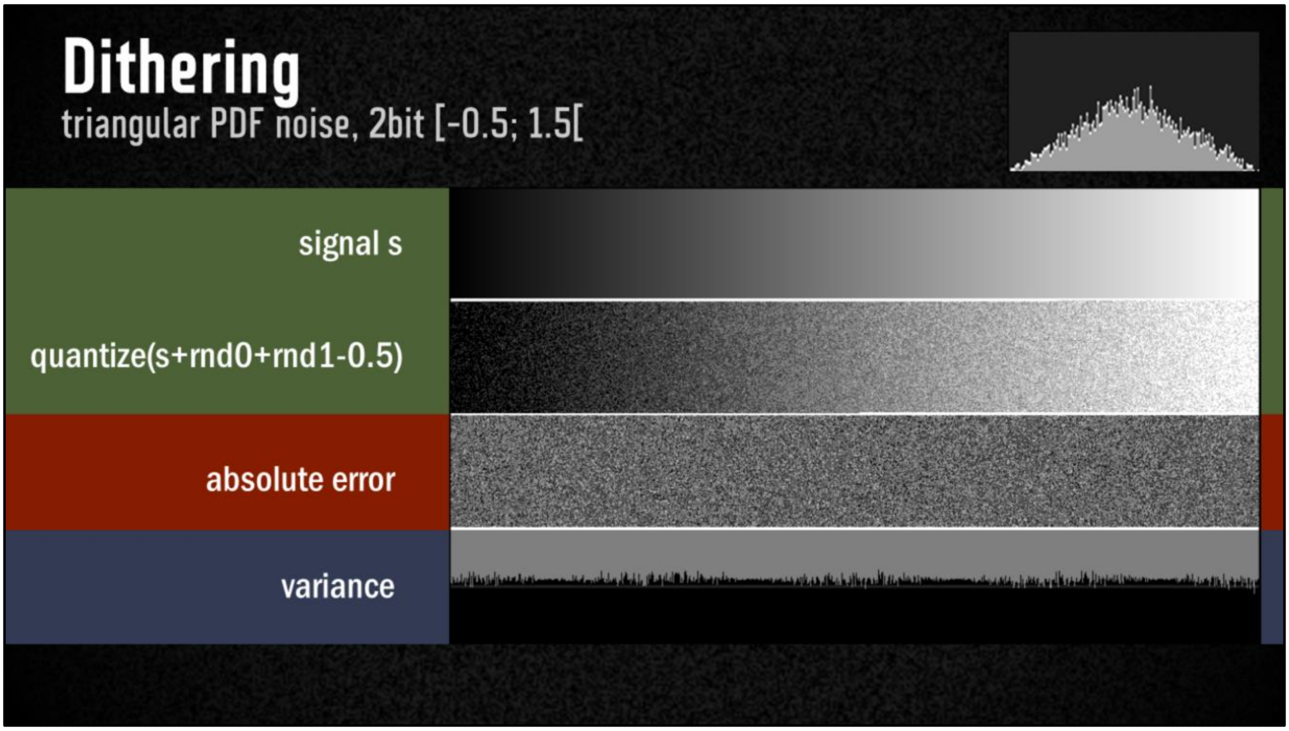
The error resulting from a triangularly distributed noise is independent of the signal.

TPDF (Triangular Probability Density Function) is the simplest distribution of noise that has this property… a Gaussian distribution does too, but is more complex to calculate.

Two easy ways to generate a TPDF:

1. Add to uniform distribtions (literally just (rand(seed0)+rand(seed1))/2)
2. Reshape a single uniform distribution (see e.g. https://www.shadertoy.com/view/4t2SDh)

https://uwspace.uwaterloo.ca/bitstream/handle/10012/3867/thesis.pdf;jsessionid=74681FAF2CA22E754C673E9A1E6957EC?sequence=1



Dithering

triangular PDF noise, 2bit [-0.5; 1.5[

signal s

quantize(s+rnd0+rnd1-0.5)

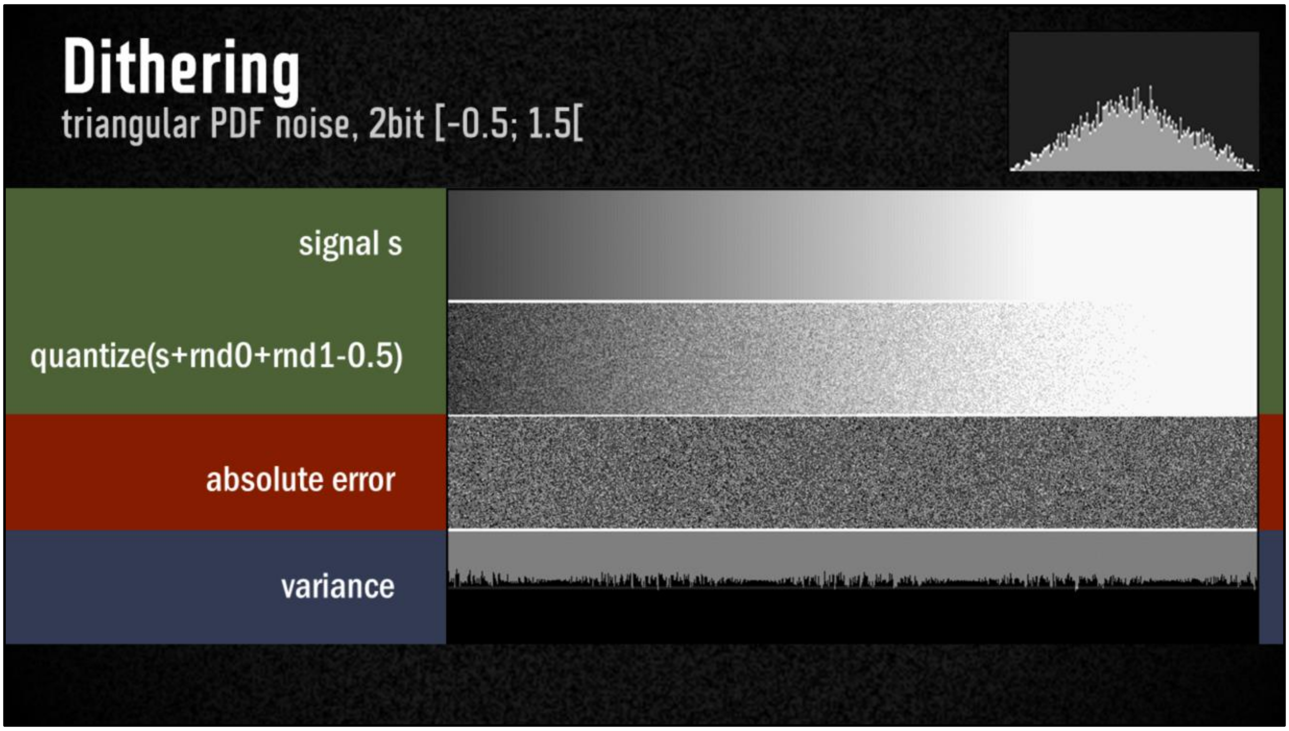
absolute error

variance

Much better! Can’t see any “bands” of no noise

Also now **2bit** wide ("overlaps" the dither-noise to compensate for the magnitude of the noise being non-uniform).

https://www.shadertoy.com/view/ltBSRG



Dithering

triangular PDF noise, 2bit [-0.5; 1.5[

signal s

quantize(s+rnd0+rnd1-0.5)

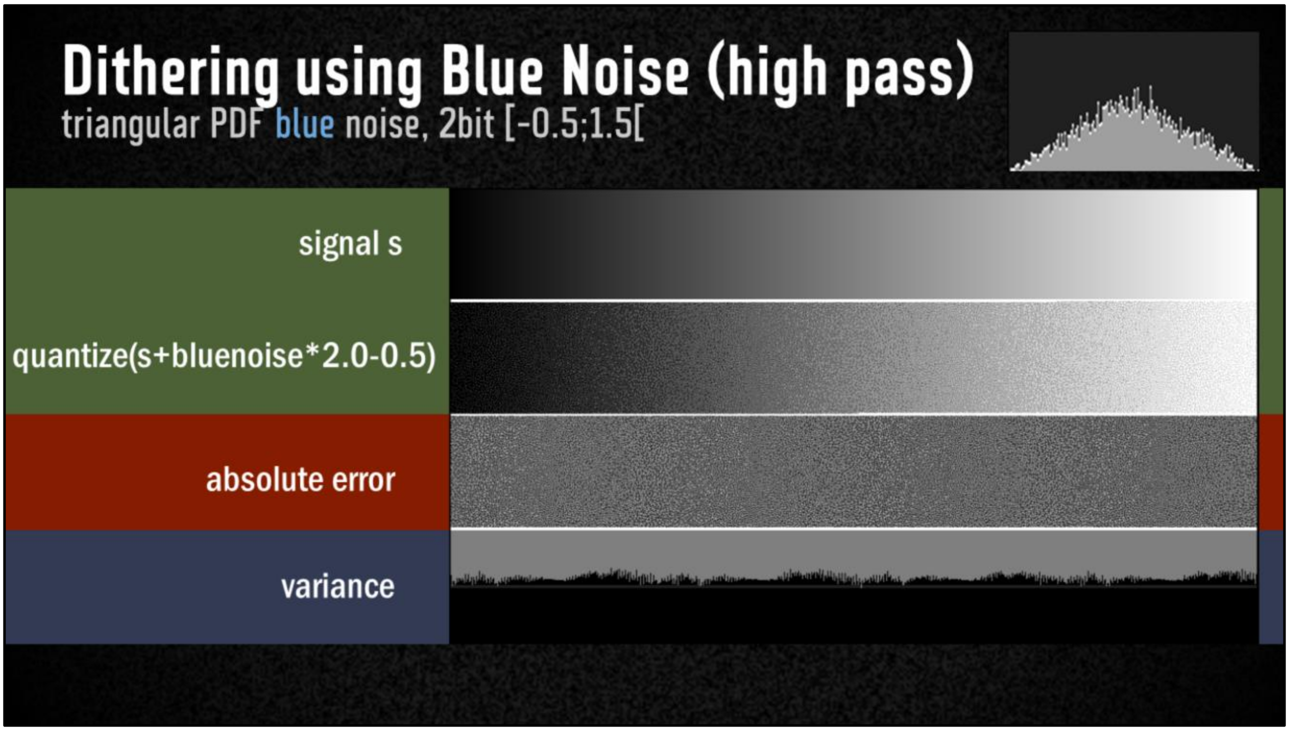
absolute error

variance

much better, can’t see any “bands” of no noise

- but we have added **quite a lot of noise**, what do do about that?

https://www.shadertoy.com/view/ltBSRG



Dithering using Blue Noise (high pass)

triangular PDF blue noise, 2bit [-0.5;1.5[

signal s

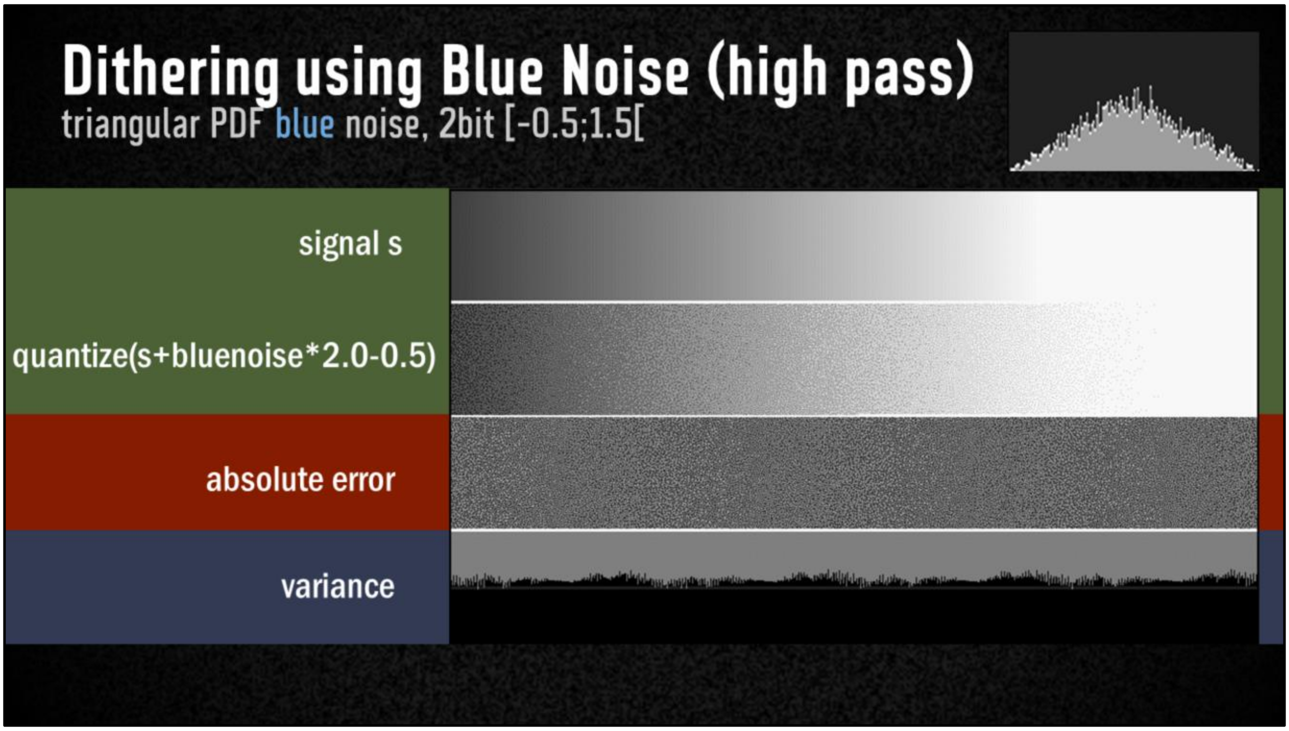
quantize(s+bluenoise*2.0-0.5)

absolute error

variance

blue noise to the rescue again

https://www.shadertoy.com/view/ltBSRG



Dithering using Blue Noise (high pass)

triangular PDF blue noise, 2bit [-0.5;1.5[

signal s

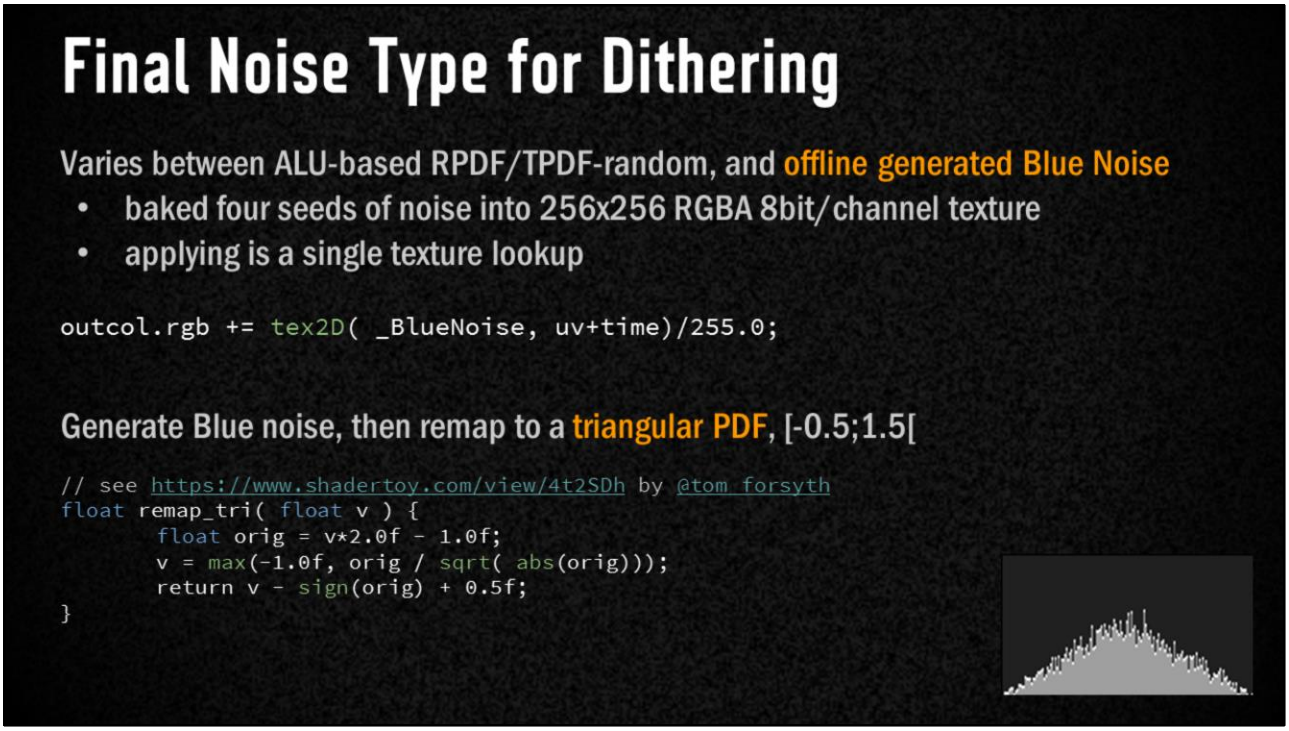
quantize(s+bluenoise*2.0-0.5)

absolute error

variance

Perceptually less noise

https://www.shadertoy.com/view/ltBSRG



Final Noise Type for Dithering

Varies between ALU-based RPDF/TPDF-random, and **offline generated Blue Noise**

- baked four seeds of noise into 256x256 RGBA 8bit/channel texture
- applying is a single texture lookup

```
outcol.rgb += tex2D( _BlueNoise, uv+time)/255.0;
```

Generate Blue noise, then remap to a **triangular PDF**, [-0.5;1.5[

```
// see https://www.shadertoy.com/view/4t2SDh by @tom_forsyth
float remap_tri( float v ) {
        float orig = v*2.0f - 1.0f;
        v = max(-1.0f, orig / sqrt( abs(orig)));
        return v - sign(orig) + 0.5f;
}
```

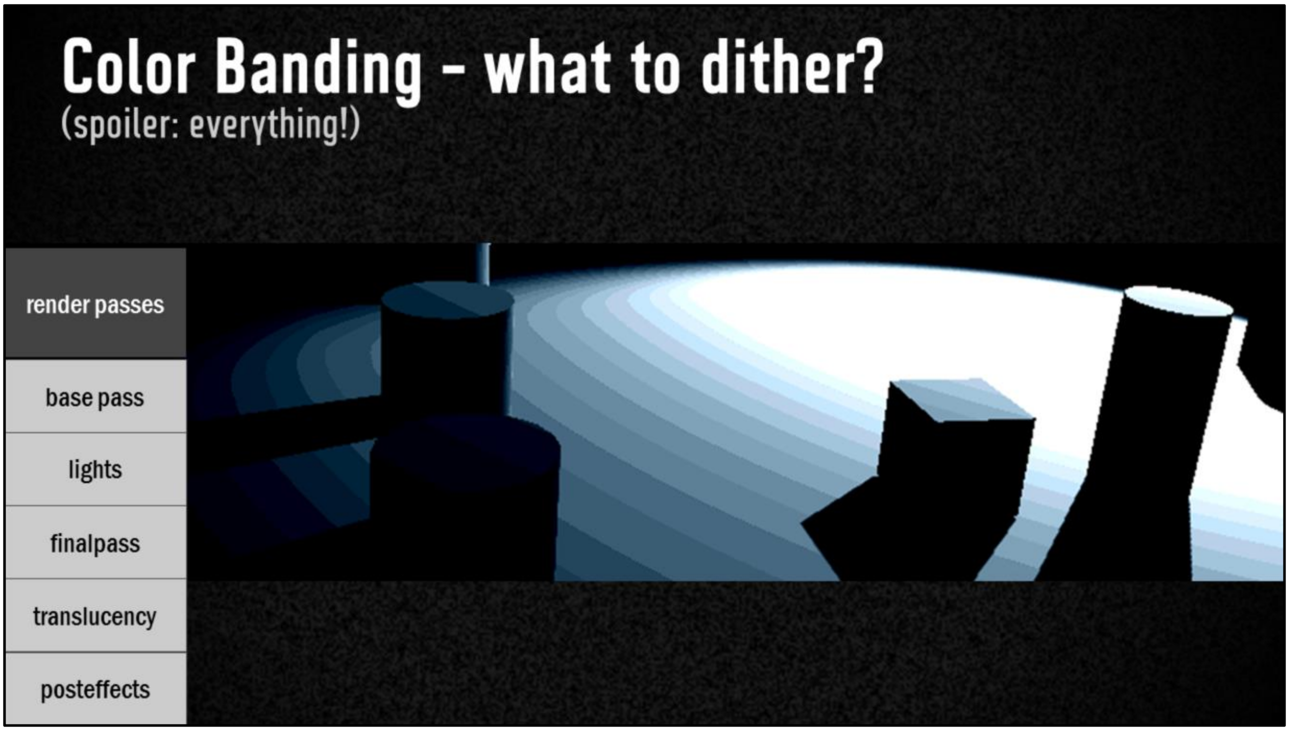
Texture with bluenoise
Sometimes ALU-based TPDF noise
Depending on bottleneck

-

optimised remapping by forsyth
https://www.shadertoy.com/view/4t2SDh

blue noise by Timothy Lottes
https://www.shadertoy.com/view/4sBSDW

blue noise array using generational algorithms
http://excedrin.media.mit.edu/wp-content/uploads/sites/10/2013/07/spie97newbern.pdf

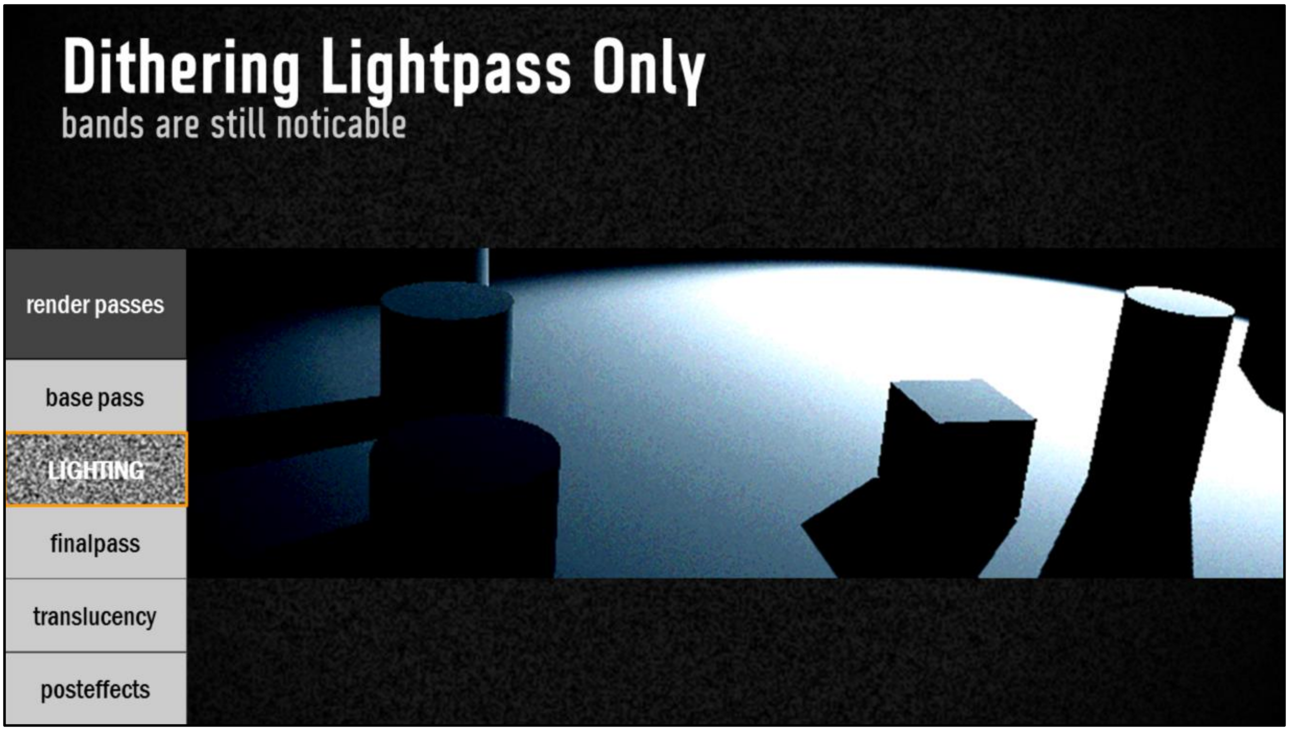


Color Banding - what to dither?

(spoiler: everything!)

- render passes
- base pass
- lights
- finalpass
- translucency
- posteffects

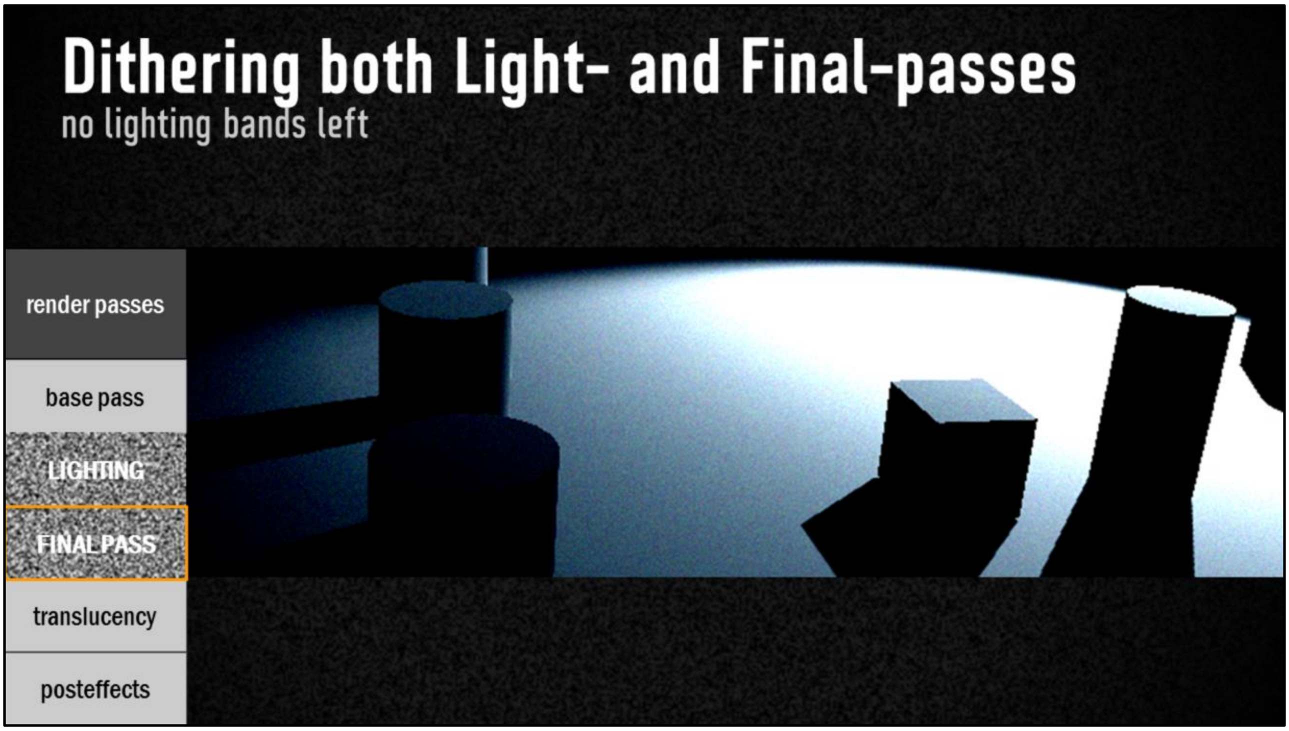
Banding in the lighting… so let us dither the lighting pass



Dithering Lightpass Only

bands are still noticable

Noise not entirely uniform
Finalpass writing into low-precision buffer as well



Dithering both Light- and Final-passes

no lighting bands left

render passes

base pass

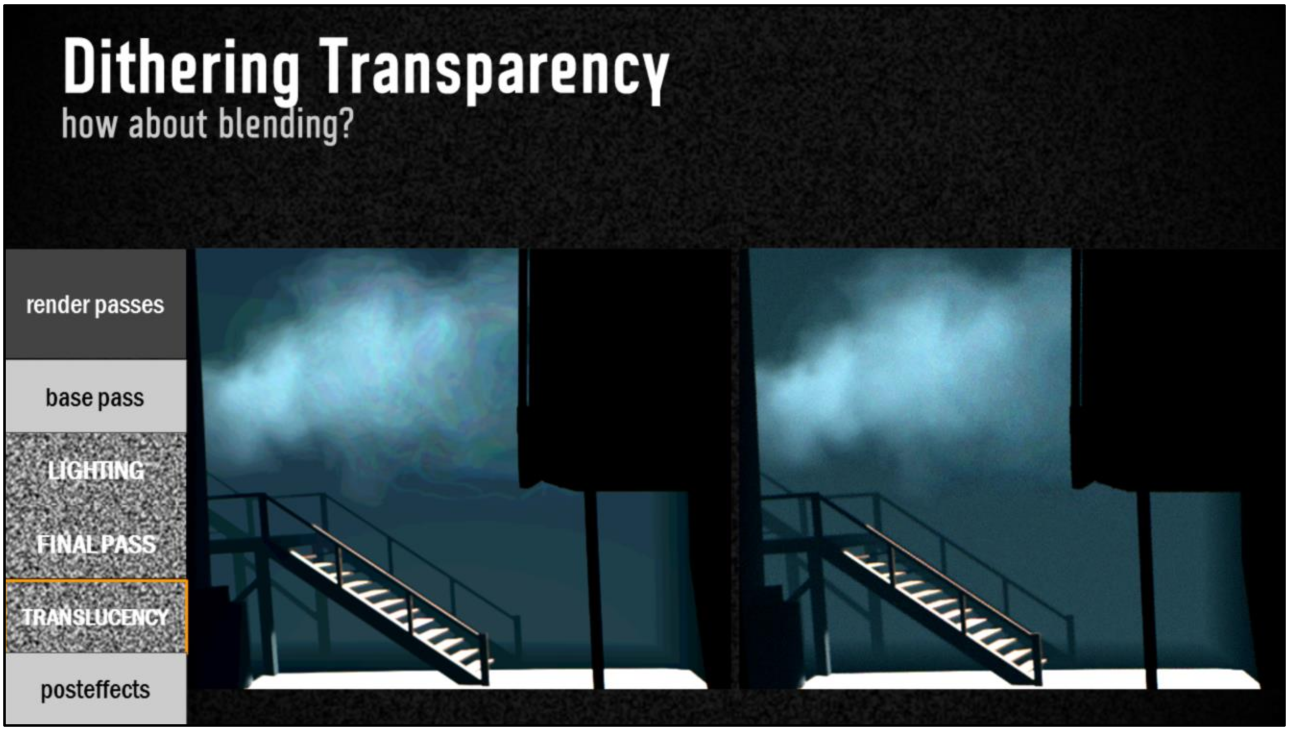
LIGHTING

FINAL PASS

translucency

posteffects

Entirely uniform noise across image



Dithering Transparency

how about blending?

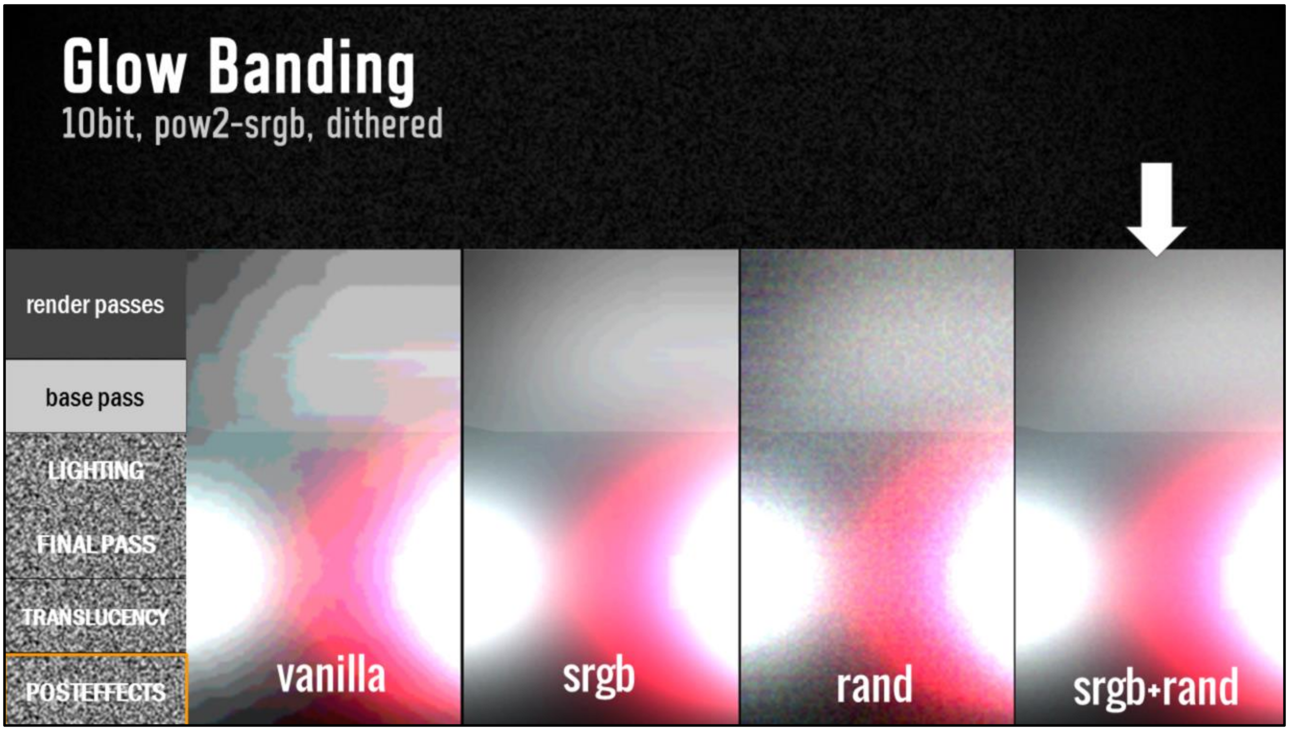
More so as every pixel written and read multiple times for blending

-

Note on blending:
Can not determine amount of noise needed for e.g. multiplicative blending - add artist-controlled amount

additive/subtractive blending ok
multiplicative blending is not
lerp, modulate etc.
needed amount of noise depends on unknown target
add artist-controlled amount of noise
(TAA soaks it up)
additive lights with exp(-i) encoding :(

See http://loopit.dk/banding_in_games.pdf for more on this.



Glow Banding

10bit, pow2-srgb, dithered

render passes

base pass

LIGHTING

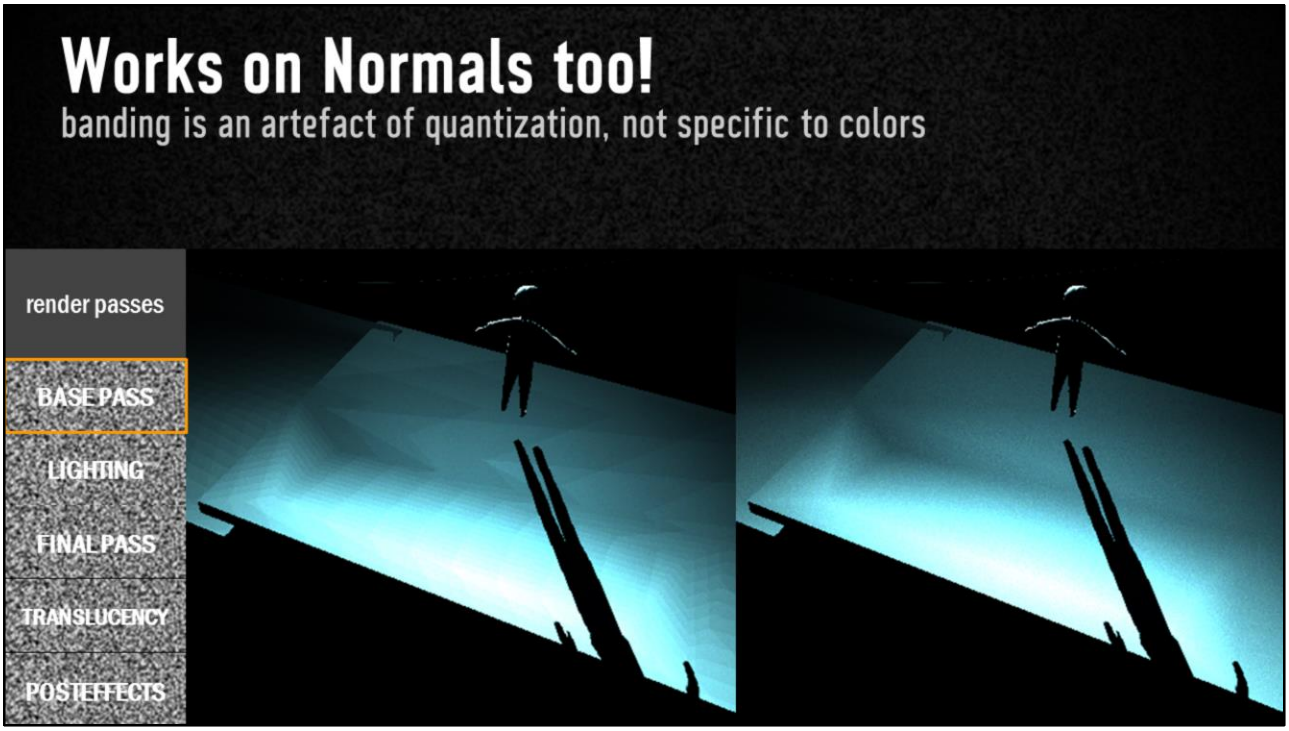
FINAL PASS

TRANSLUCENCY

POSTEFFECTS

our solution dithers after every pass
manually converts all intermediate rendertargets to srgb (pow2, nothing fancy)

Dithering at lower resolutions (as used for blurring) results in larger than 1pixel noise, which had us worried but turned out to not be visible.
sRGB goes a long way! adding noise was necessary for us though (srgb approximated by pow2)



Works on Normals too!

banding is an artefact of quantization, not specific to colors

...dither gbuffers too! Normals are written to 8bit rendertargets

Fixes unsightly patterns in specular highlights
(the lines that are still noticeable are due to linear interpolation across triangles)

OKOK, we will dither everything (gees!)

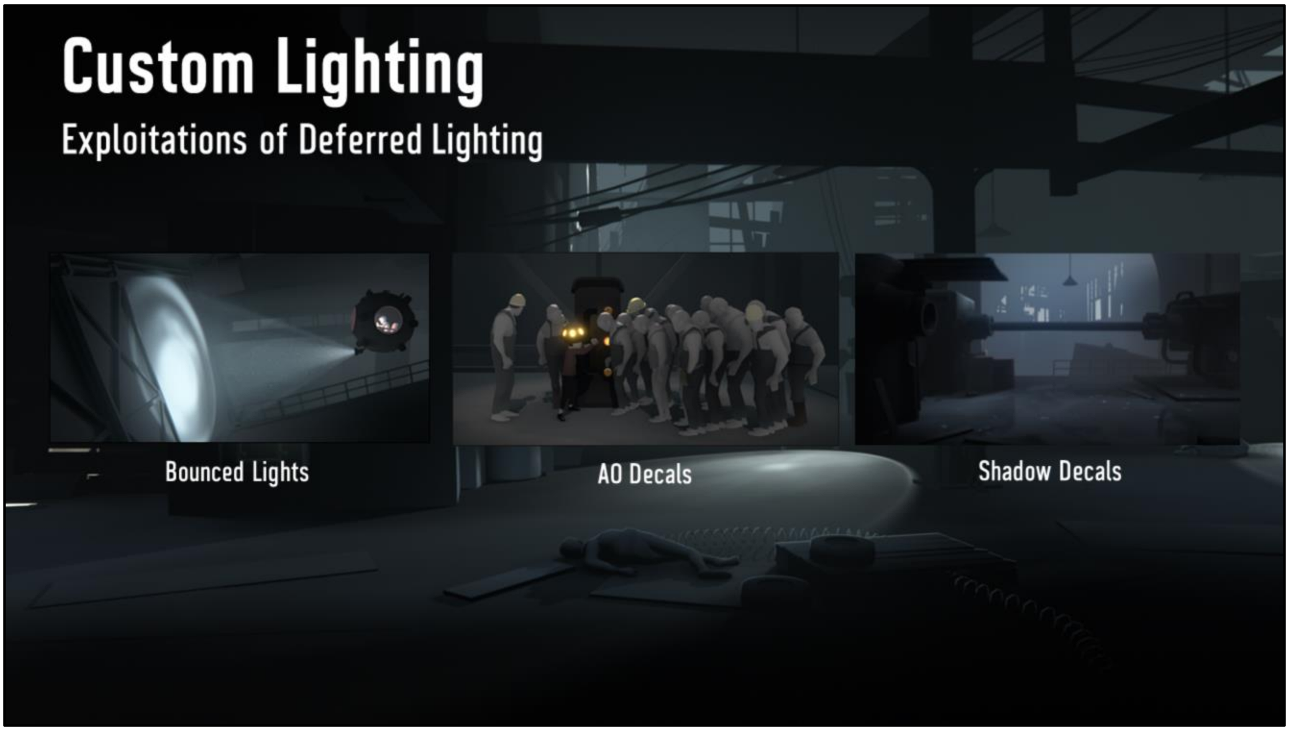
...anything else?

Animate the noise! For the love of all that is good!

- Screenspace static patterns are noticeable
- Animating it also means Temporal Anti-Aliasing will soak up the noise...

Match TV input-range yourself, so output-hardware will not do lossy conversion to Limited-range RGB.

PS ...also dither UI!



Custom Lighting

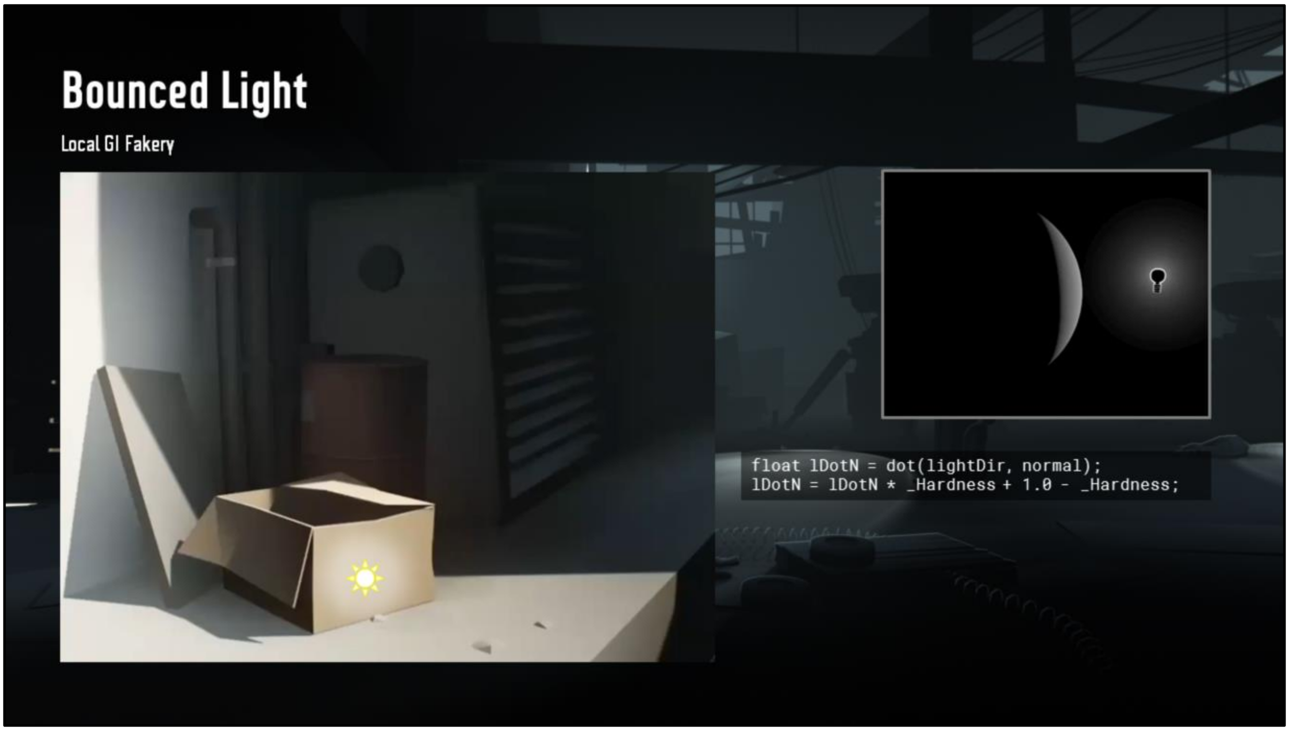
Exploitations of Deferred Lighting

Bounced Lights

AO Decals

Shadow Decals

Now let’s talk about custom lighting
Like we said, we use light prepass deferred rendering, which allows us to do a few tricks
We don’t need to limit ourselves to render just point lights, spotlights and direct lights
We exposed the ability to render out any object with any shader, because we can!
So with this we get custom lights, we’ve got a bunch, let’s look at 3 of these



Bounced Light

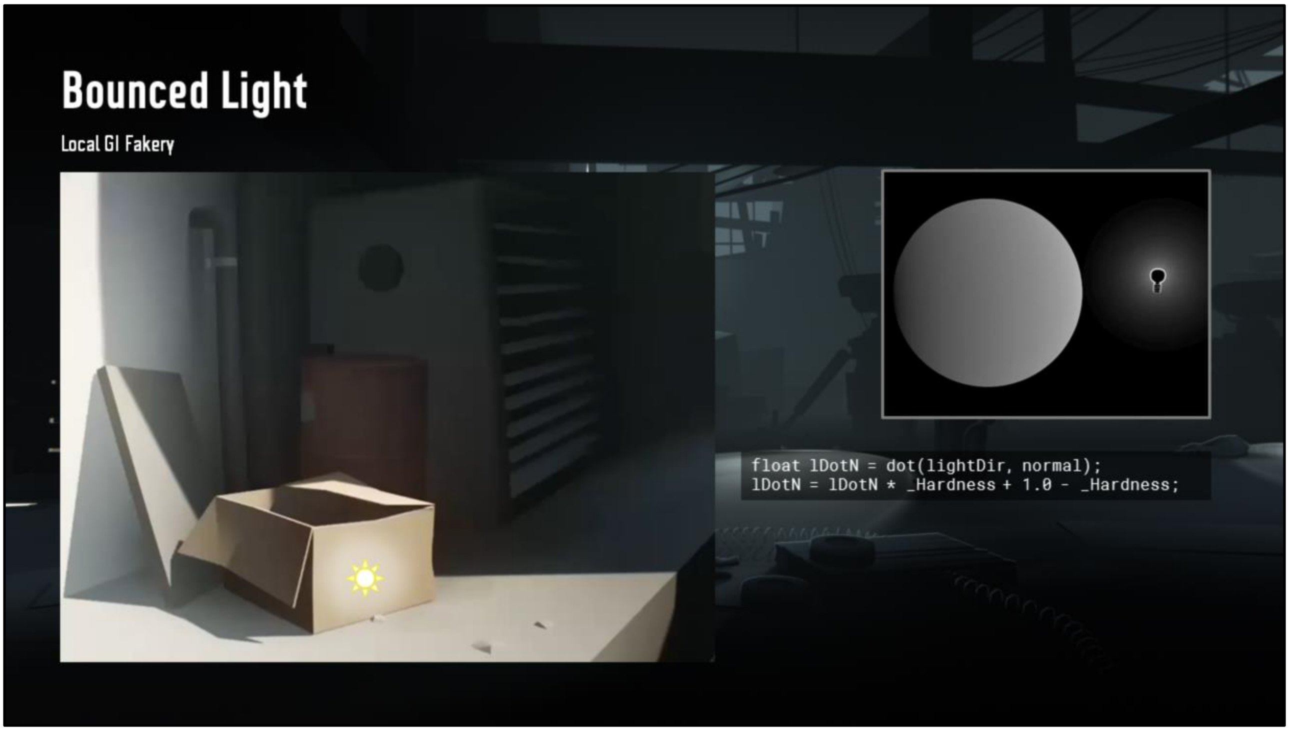
Local GI Fakery

```
float lDotN = dot(lightDir, normal);
lDotN = lDotN * _Hardness + 1.0 - _Hardness;
```

The bounced light is used for, you guessed it, global illumination

It’s pretty much just a regular lambert point light, except…

Rather than using a vanilla dot product, we fade that down using a slider
It’s called lambert wrap or half lambert
Gives a less directional, more smooth result

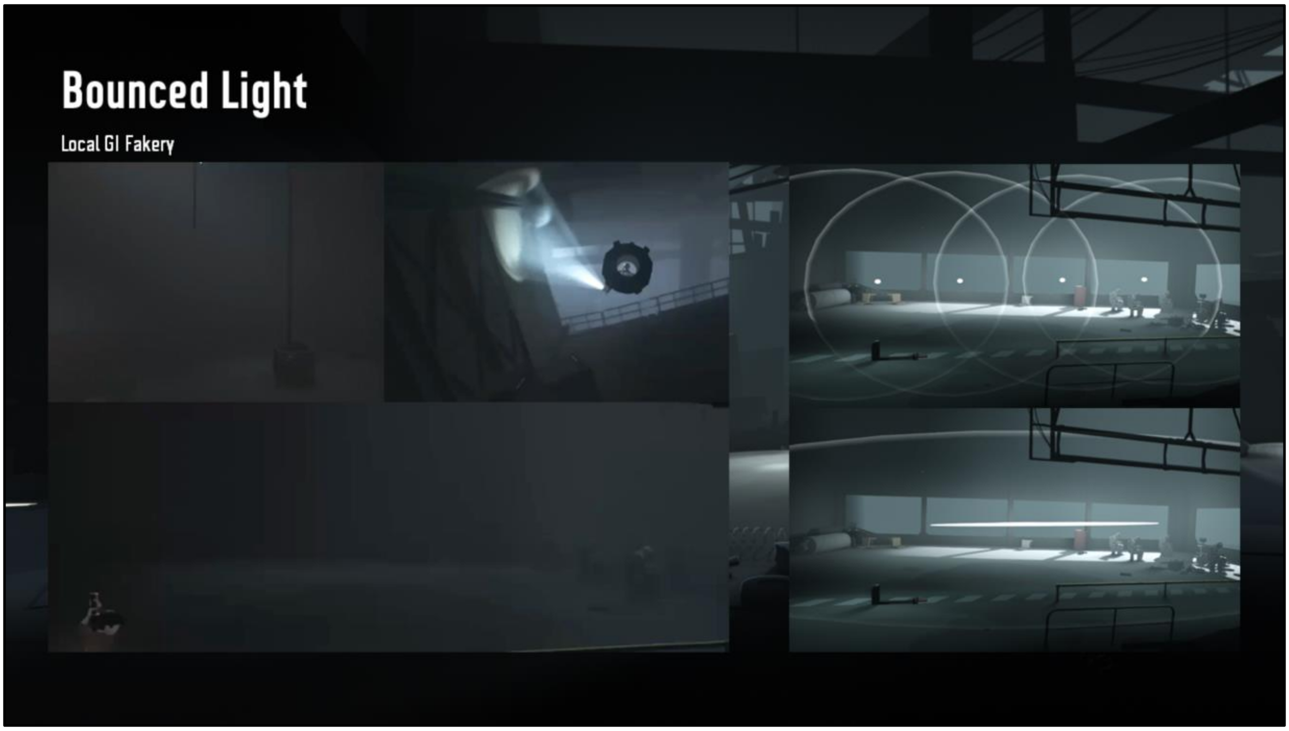


Bounced Light

Local GI Fakery

```
float lDotN = dot(lightDir, normal);
lDotN = lDotN * _Hardness + 1.0 - _Hardness;
```

We can fade off the normal completely if we’d like, giving us a more ambient light



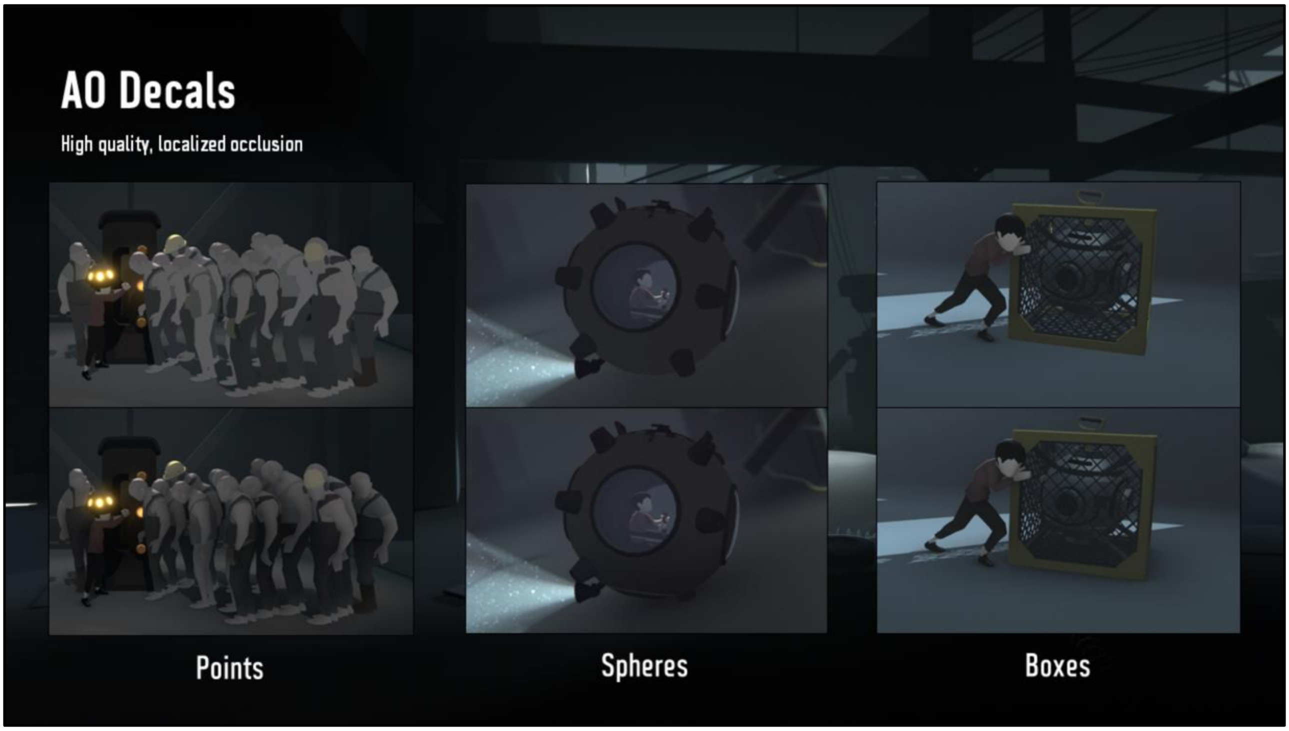
Bounced Light

Local GI Fakery

Since they’re not static, we often use them for opening windows and moving flashlights

Unlike regular points where you’d have to make a sausage of points to cover a corridor, or an array to fill a room

We use the full transform matrix, to give us non-uniform shapes,
Fits more cases with stretched pills and squashed buttons, and it’s cheaper because we get less overdraw and overlap



AO Decals

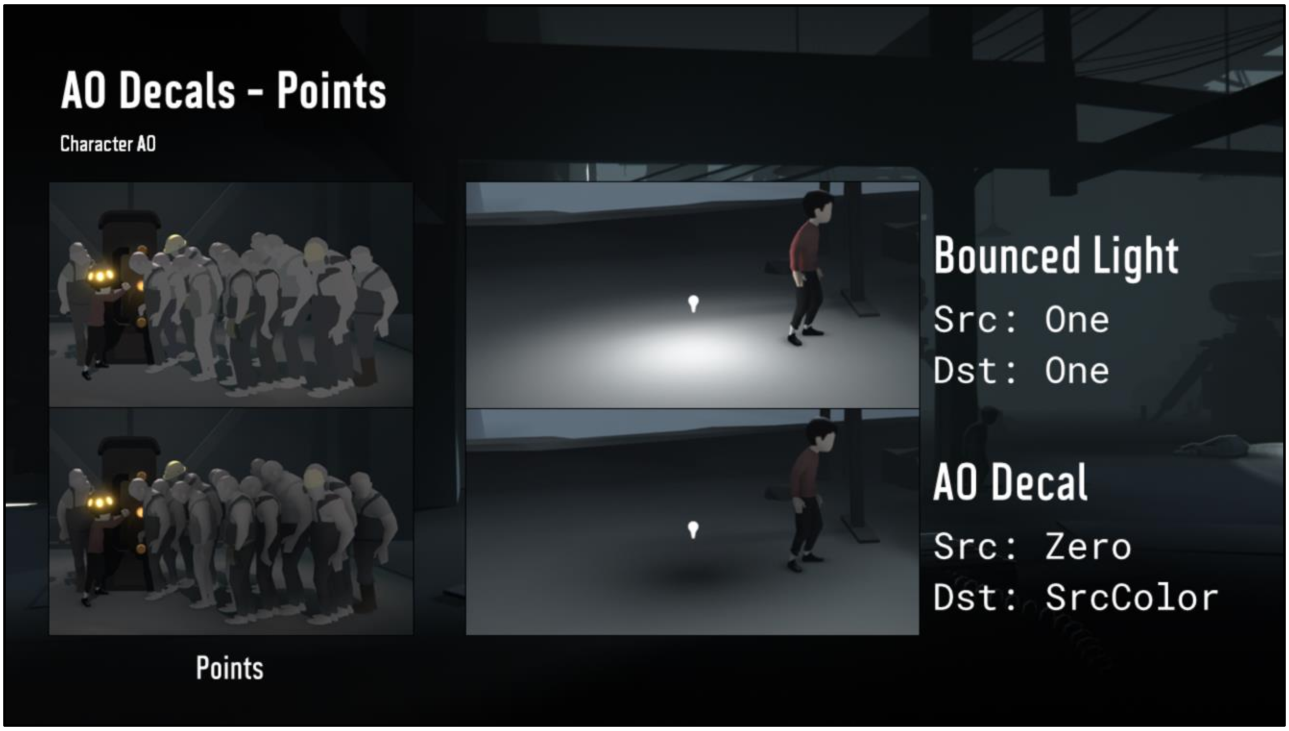
High quality, localized occlusion

Points

Spheres

Boxes

Now for AO Decals
Since we can draw whatever, we can also blend however, we aren’t limited to additive lights, so we made multiplicative ones.
This is our only AO solution, we don’t wanna use SSAO due to it’s lack of local control and artefacts associated with screen space effects.
We have three of these, so let’s take a look first at…



AO Decals - Points

Character AO

Bounced Light
Src: One
Dst: One

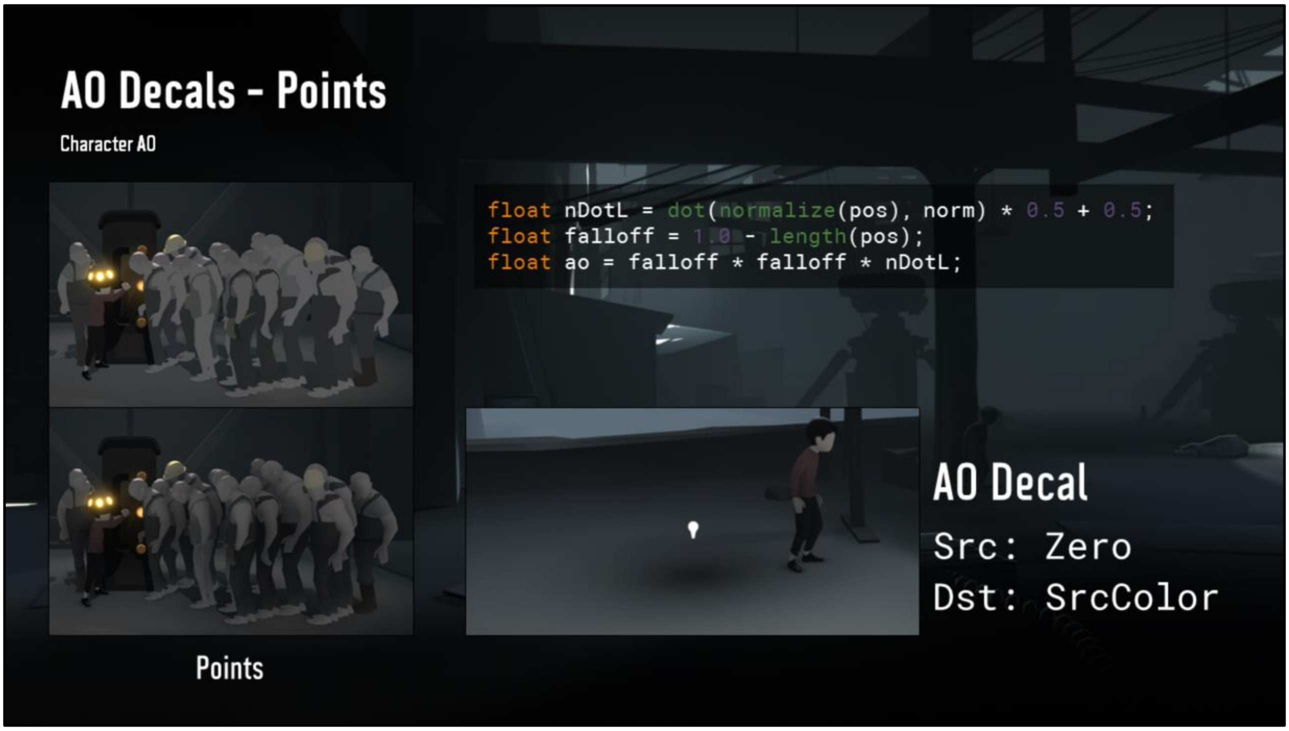
AO Decal
Src: Zero
Dst: SrcColor

Points

The point.
We use these mostly for characters, it ground them to the world, other characters and to themselves.
We set them up one per bone, stretched to fit the volume of that limb.
It makes it easy to see the contact of arms against sides, and the head on the shoulder and so on.
In this example we've probably got like 256 of them, I see 16 characters and I count 16 bones each.

Their implementation is easy, if you start with a regular half lambert point light aka bounced light, which is additive.

And just make it multiplicative rather than additive, you're done.



AO Decals - Points

Character AO

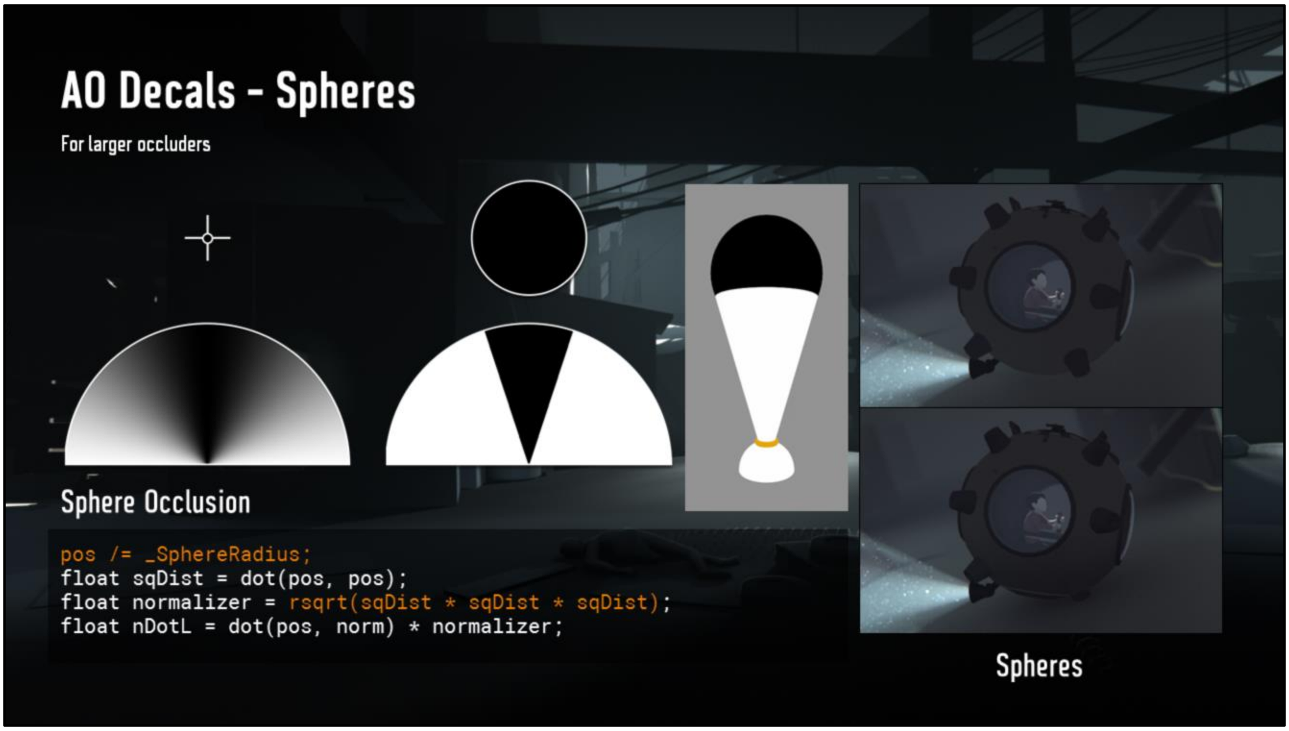
```
float nDotL = dot(normalize(pos), norm) * 0.5 + 0.5;
float falloff = 1.0 - length(pos);
float ao = falloff * falloff * nDotL;
```

AO Decal

```
Src: Zero
Dst: SrcColor
```

Points

This entity has no wrap parameter, we fixed it to half lambert, and gave it a fixed falloff.
Since we put so many, it’s important to have controls be easy, so all we got is the transformation (position/rotation/scaling) and an intensity slider.



AO Decals - Spheres

For larger occluders

Sphere Occlusion

```
pos /= _SphereRadius;
float sqDist = dot(pos, pos);
float normalizer = rsqrt(sqDist * sqDist * sqDist);
float nDotL = dot(pos, norm) * normalizer;
```

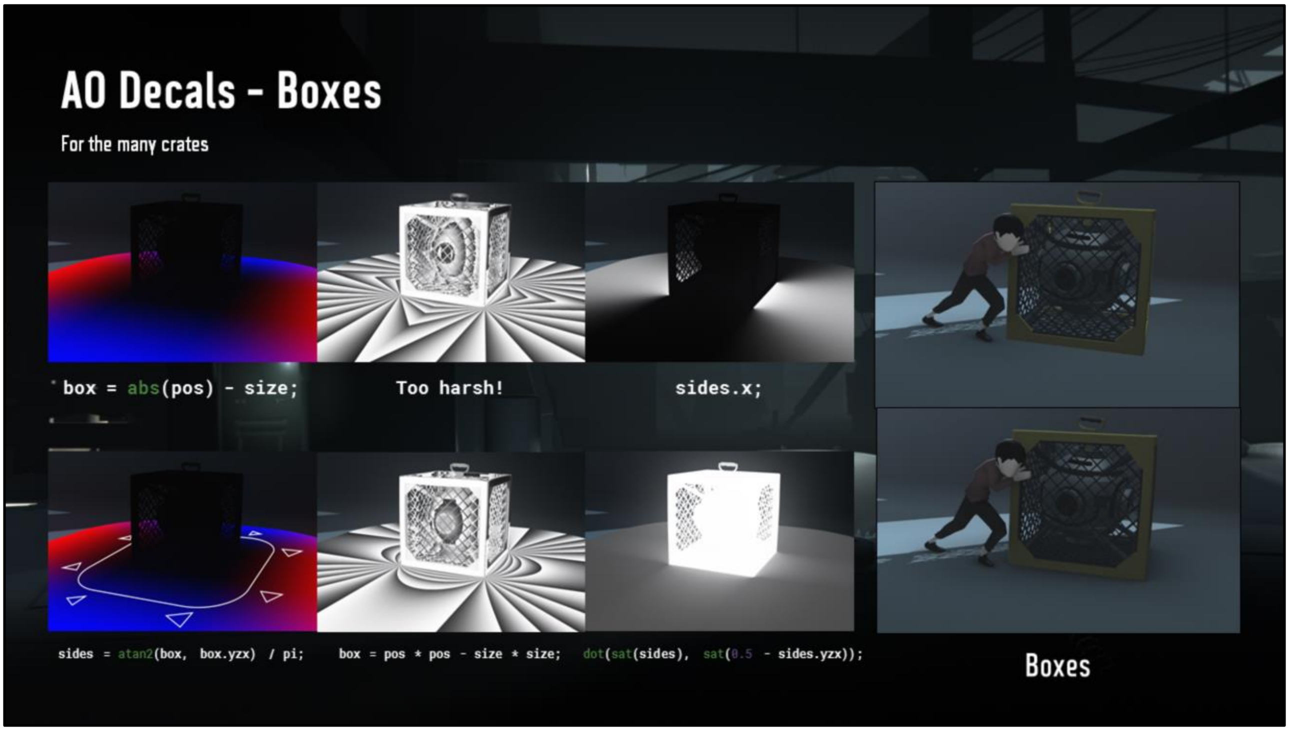
Spheres

Next up, spheres.
These are for our larger round occluders, like this submarine. In these cases we also need the contact to be apparent to judge distances better.

With implementation, we start with a regular lambert point, again

But in this case, we don't want the angle between the occluder and the surface
We want the solid angle coverage between occluder and surface
So to implement this, instead of doing our normal normalizer term, we use…

This, which actually gives you that result accurately, the covered sum.
Assuming the occluder doesn't intersect the surface, which we do assume.
This implementation comes from Íñigo Quílez [Quílez06]
http://www.iquilezles.org/www/articles/sphereao/sphereao.htm



AO Decals - Boxes

For the many crates

```
box = abs(pos) - size;
```

Too harsh!

```
sides.x;
```

```
sides = atan2(box, box.yzx) / pi;
```

```
box = pos * pos - size * size;
```

```
dot(sat(sides), sat(0.5 - sides.yzx));
```

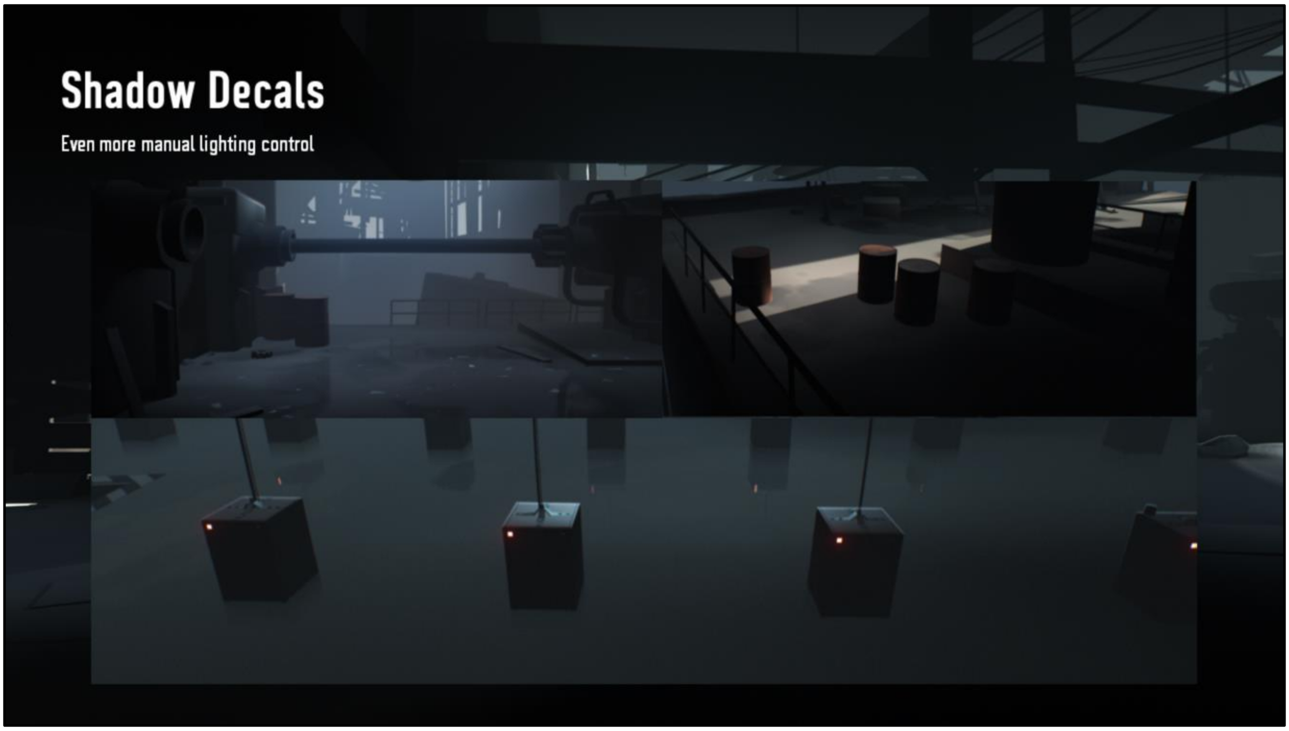
Boxes

Next up, boxes.
For the many boxes and crates that are used to solve puzzles.
Obviously the requirements are different here, we want sharp falloffs along corners.

For implementing this one, we need the unsigned distance vector
Then get the angle around that vector

But that’s too harsh, especially when looking at an exaggerated fractional.
So we use the squared distance vector instead, smoothing out the first derivative completely.

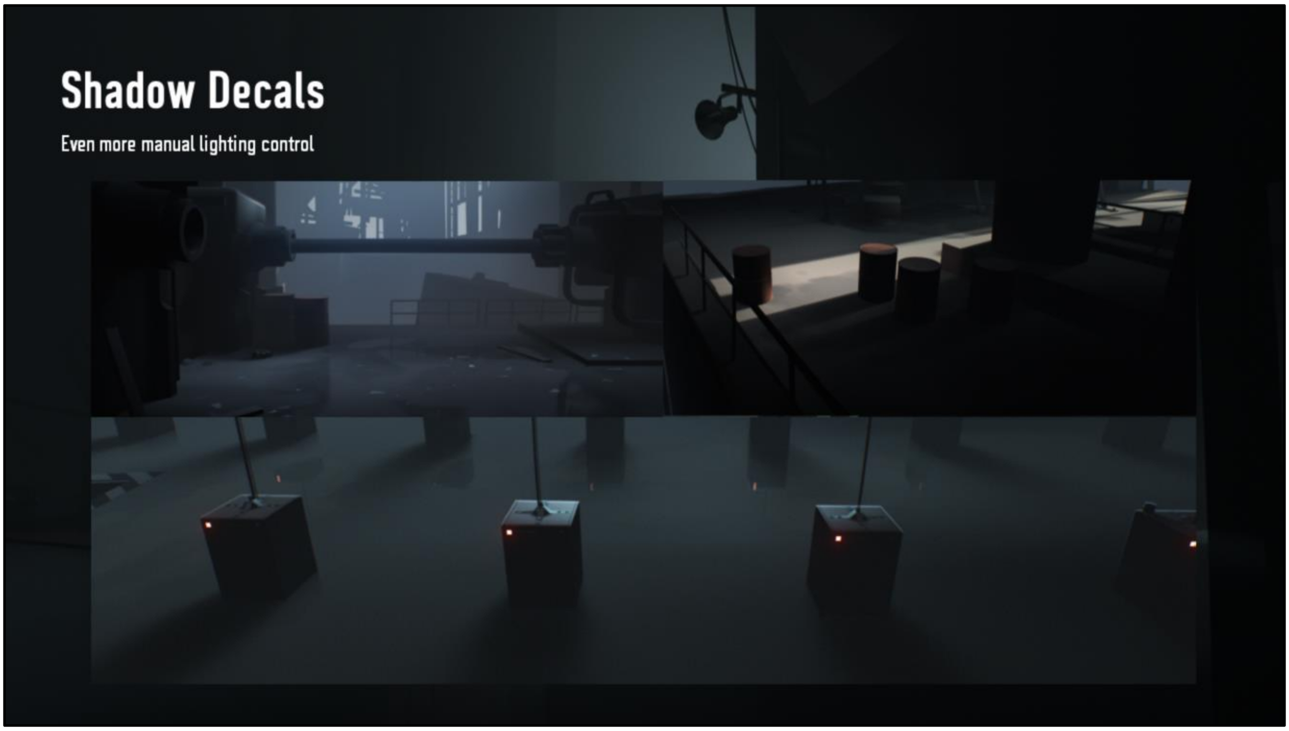
Each side should look like this, kind of a planar area ligh.t
Composed like below, now we get those corners.
This solution is not physically accurate in any way, it’s totally empirically based. But it is inspired by https://www.shadertoy.com/view/4djXDy [Quílez14]



Shadow Decals

Even more manual lighting control

Finally, shadow decals. This is our simplest custom lighting effect.
We use it for grounding in scenes with unshadowed lights and ambient lighting.
Like this one, let’s add shadows!...



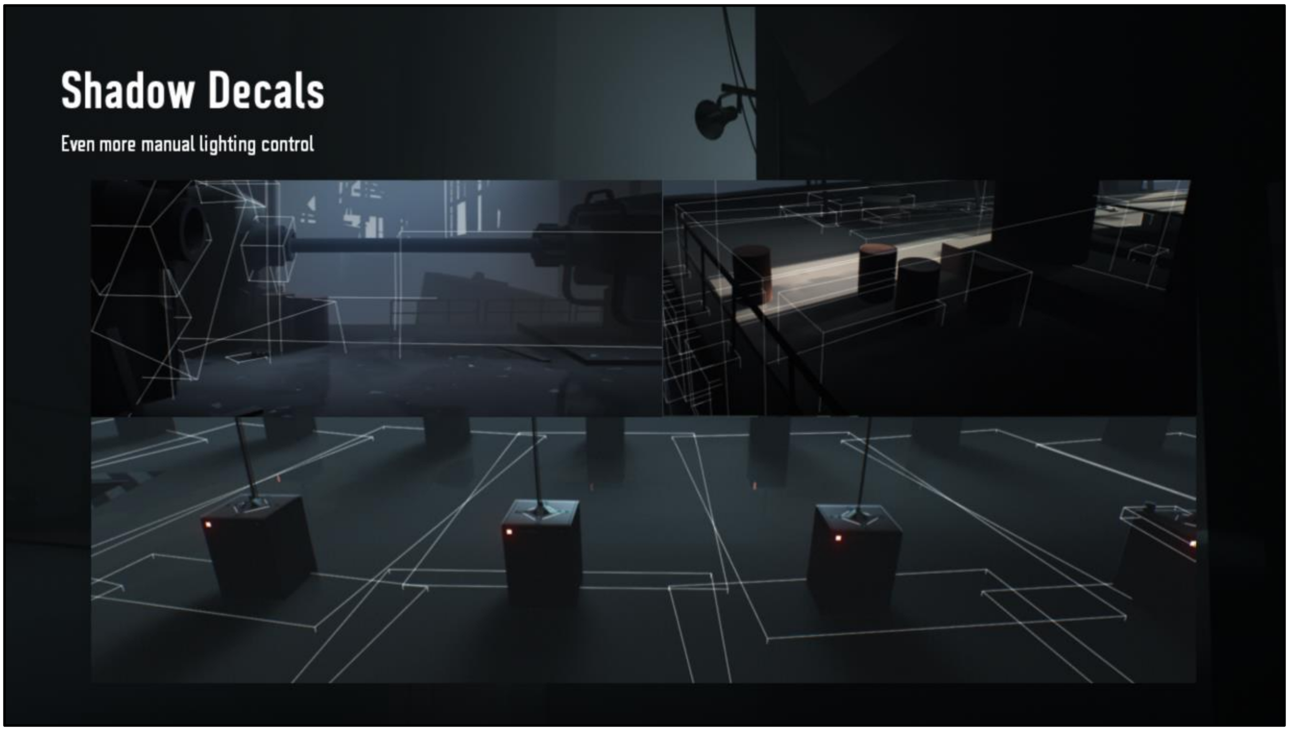
Shadow Decals

Even more manual lighting control

There!

Implementation wise, this is just a box-shaped decal with a projected alpha texture that multiplies the color of the light buffer. Also a falloff gradient.

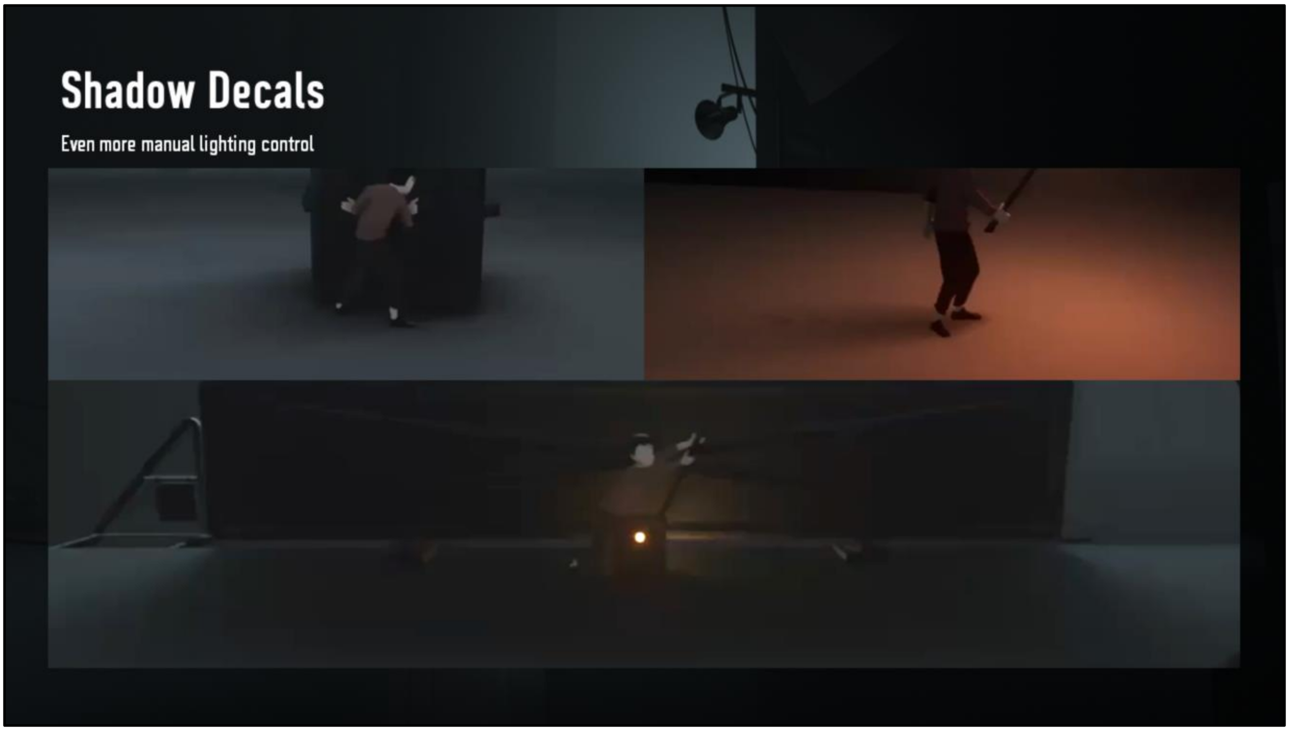
Typically we end up with very soft, gradieted shadows, they lack the detail you'd get with shadowmaps, on purpose to simplify.



Shadow Decals

Even more manual lighting control

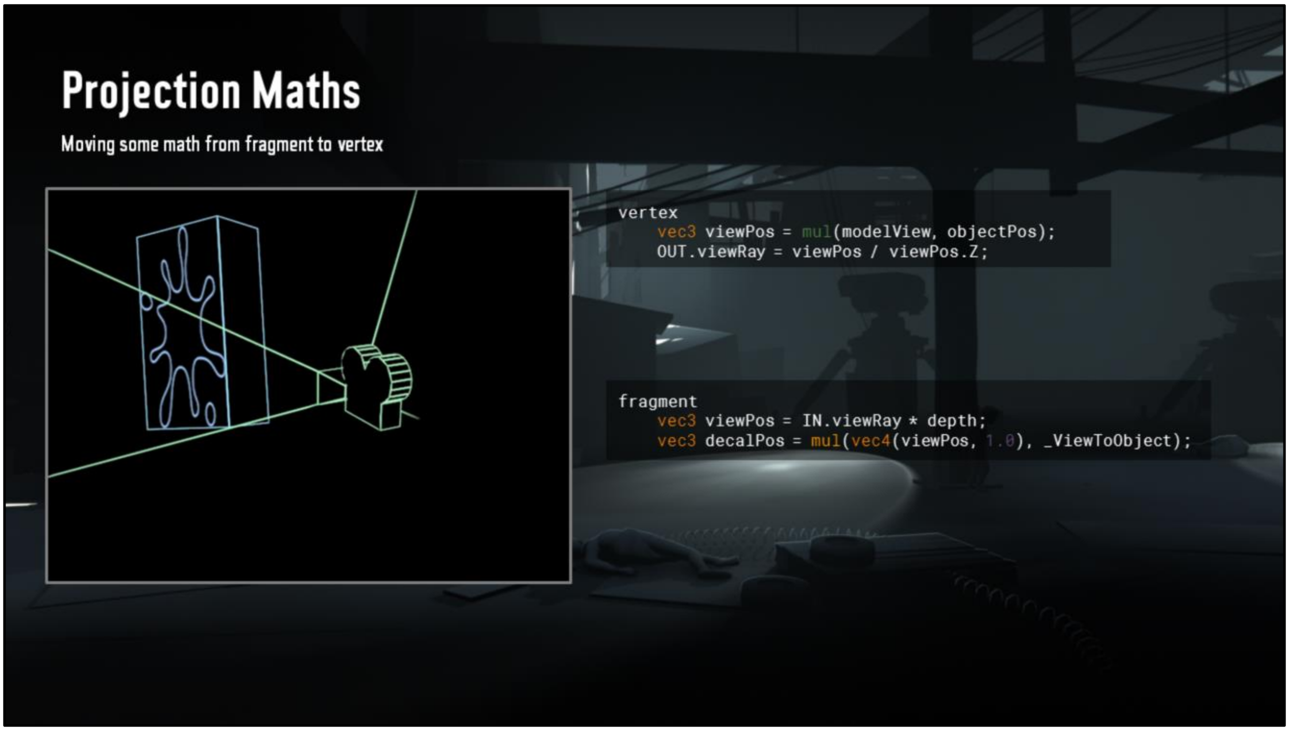
And we use a lot of these!



Shadow Decals

Even more manual lighting control

Since they’re not static, we can use them in clever ways.
On a draggable box.
Anchored to boy’s feet, to avoid rendering point shadows from a torch, giving a very OOT blob shadow look.
And on this big platform to get a nicely fading shadow, that you couldn’t otherwise get.



Projection Maths

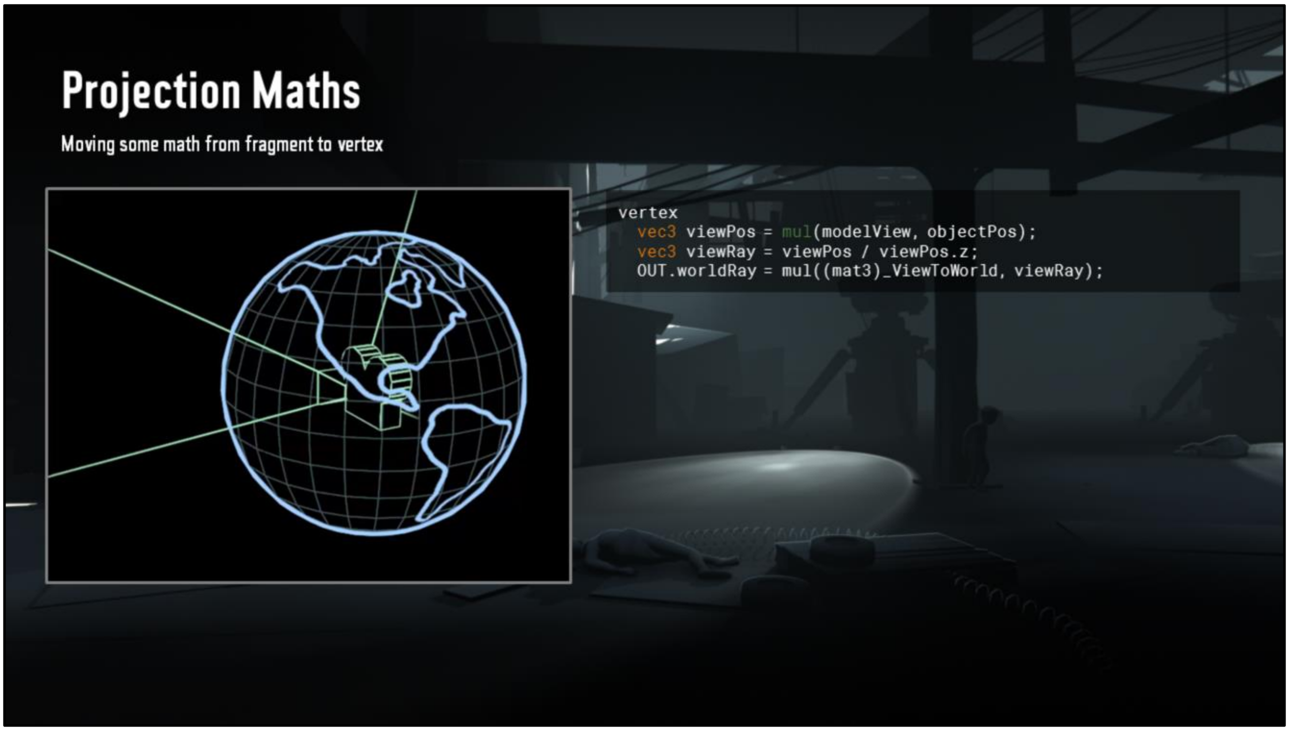
Moving some math from fragment to vertex

```
vertex
    vec3 viewPos = mul(modelView, objectPos);
    OUT.viewRay = viewPos / viewPos.Z;
```

```
fragment
    vec3 viewPos = IN.viewRay * depth;
    vec3 decalPos = mul(vec4(viewPos, 1.0), _ViewToObject);
```

Like I said we do a lot of these, a lot of pixels covered, so we wanna do minimal work in fragment shader.
But we can do almost anything we like in vertex, since there are only 8 of those.
Normally you’d make a view ray in vertex…

Then in fragment multiply depth, transform it to decal or object space here.
At the cost of a matrix multiplication
Luckily, fixing this is simple, especially if…



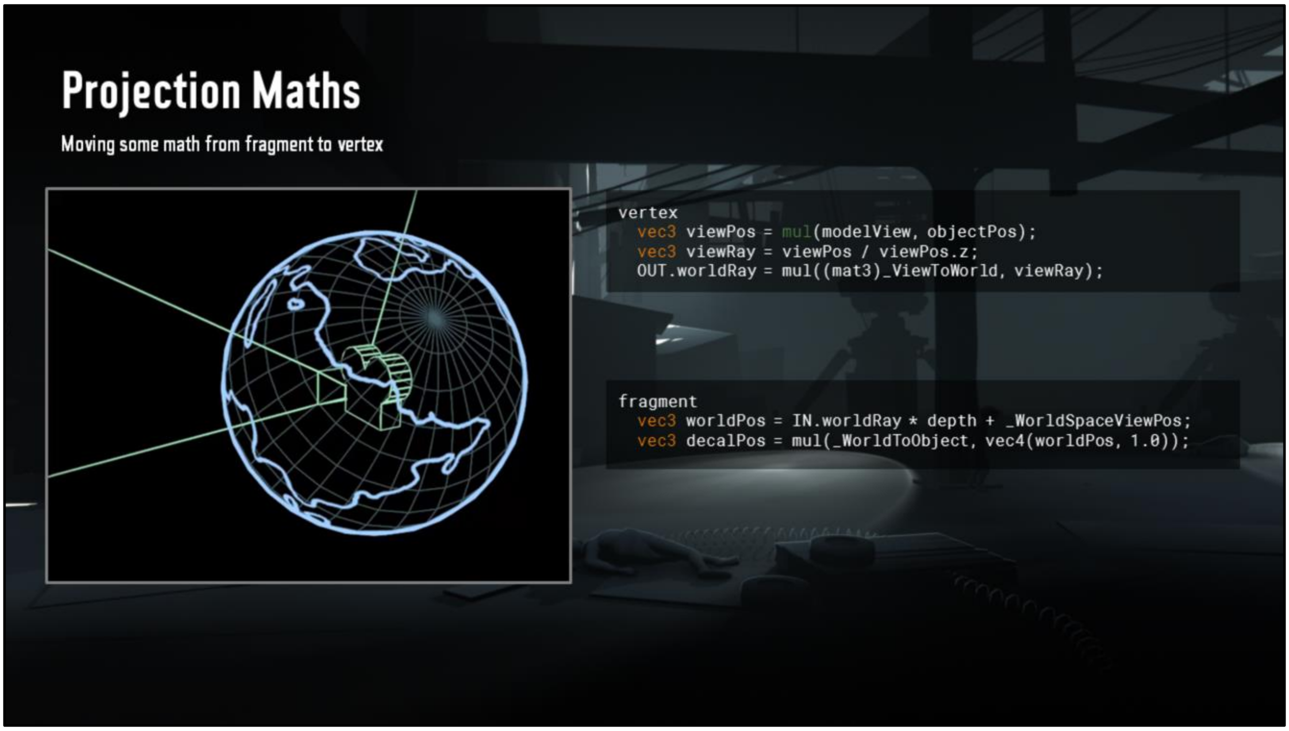
Projection Maths

Moving some math from fragment to vertex

```
vertex
  vec3 viewPos = mul(modelView, objectPos);
  vec3 viewRay = viewPos / viewPos.z;
  OUT.worldRay = mul((mat3)_ViewToWorld, viewRay);
```

You’ve done world space interpolated rays for world space reconstruction before!
In this case, you make a view ray in vertex

Transform it to world, while in vertex



Projection Maths

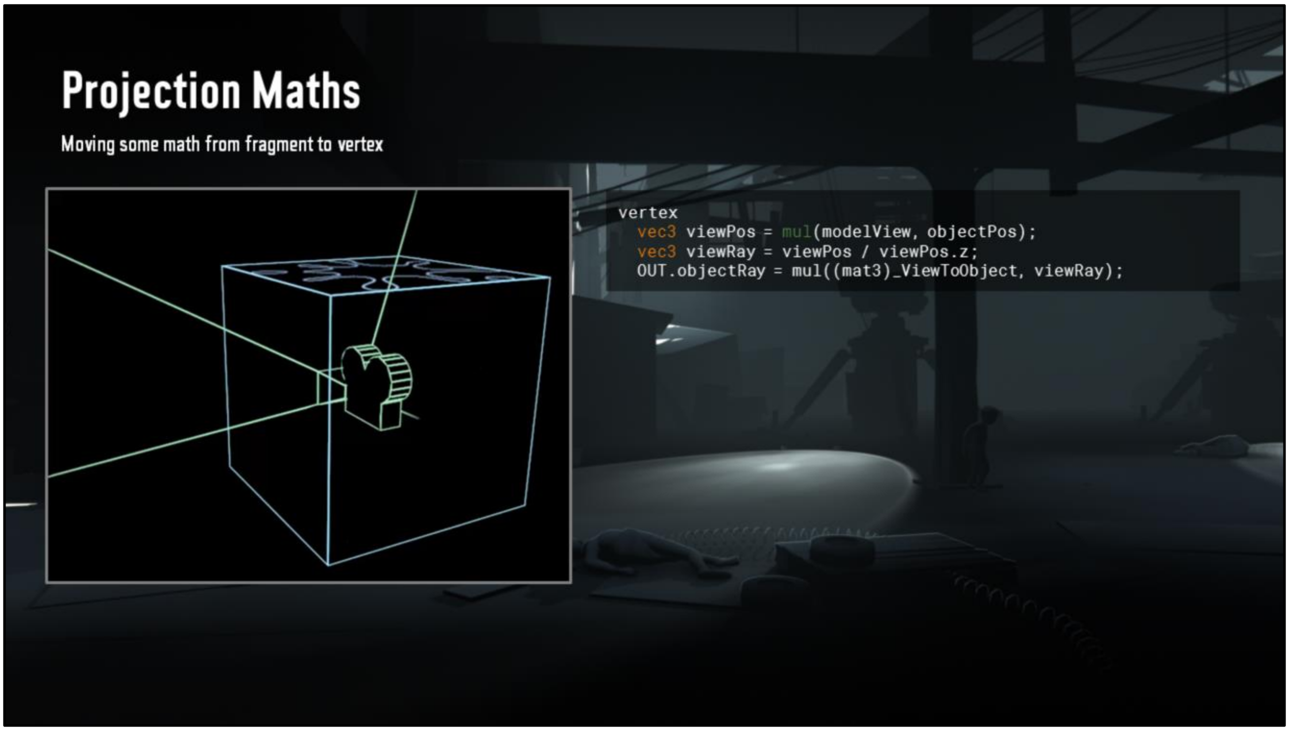
Moving some math from fragment to vertex

```
vertex
  vec3 viewPos = mul(modelView, objectPos);
  vec3 viewRay = viewPos / viewPos.z;
  OUT.worldRay = mul((mat3)_ViewToWorld, viewRay);
```

```
fragment
  vec3 worldPos = IN.worldRay * depth + _WorldSpaceViewPos;
  vec3 decalPos = mul(_WorldToObject, vec4(worldPos, 1.0));
```

Then finish it off in fragment by multiplying by depth and adding world offset
This world space position uses only a single MAD or FMA, so that’s neat.

But here for decals we’d still need to use a matrix in the fragment shader
So…



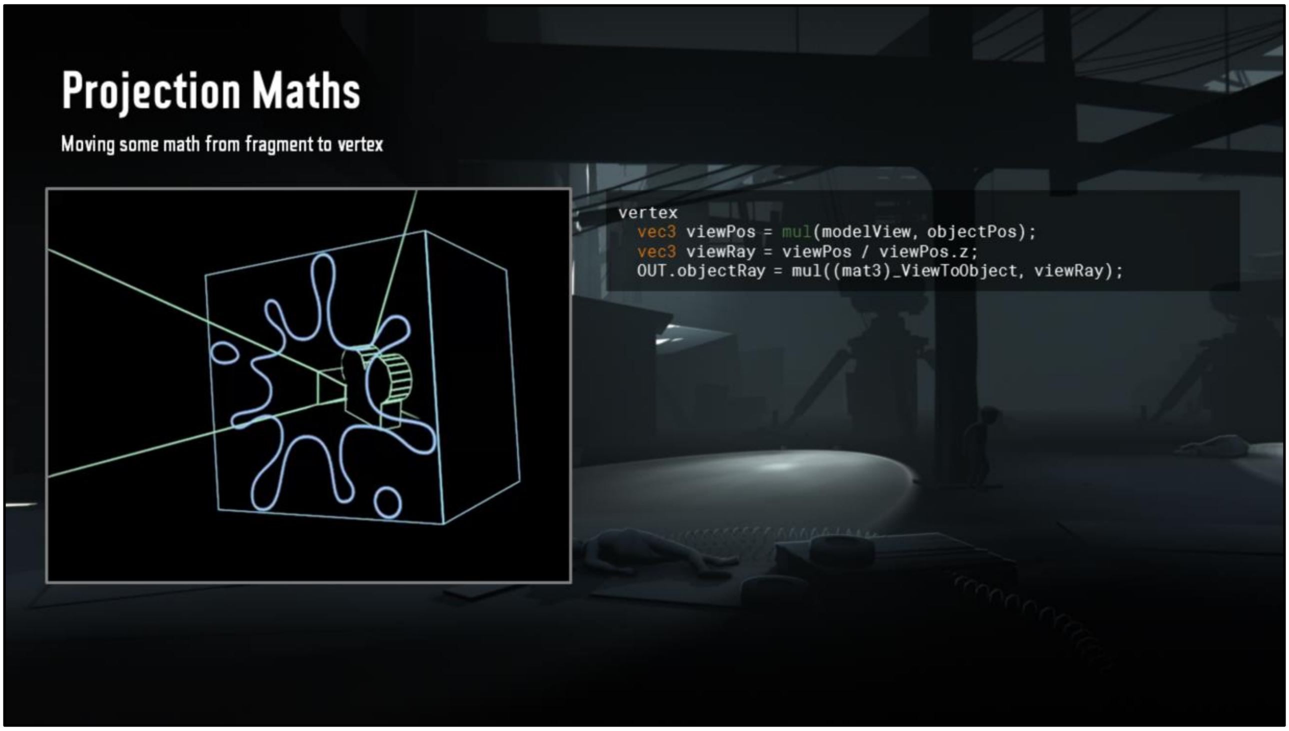
Projection Maths

Moving some math from fragment to vertex

```
vertex
  vec3 viewPos = mul(modelView, objectPos);
  vec3 viewRay = viewPos / viewPos.z;
  OUT.objectRay = mul((mat3)_ViewToObject, viewRay);
```

For the decal version, we do the same view ray in vertex

Transform it into object space for our rotation

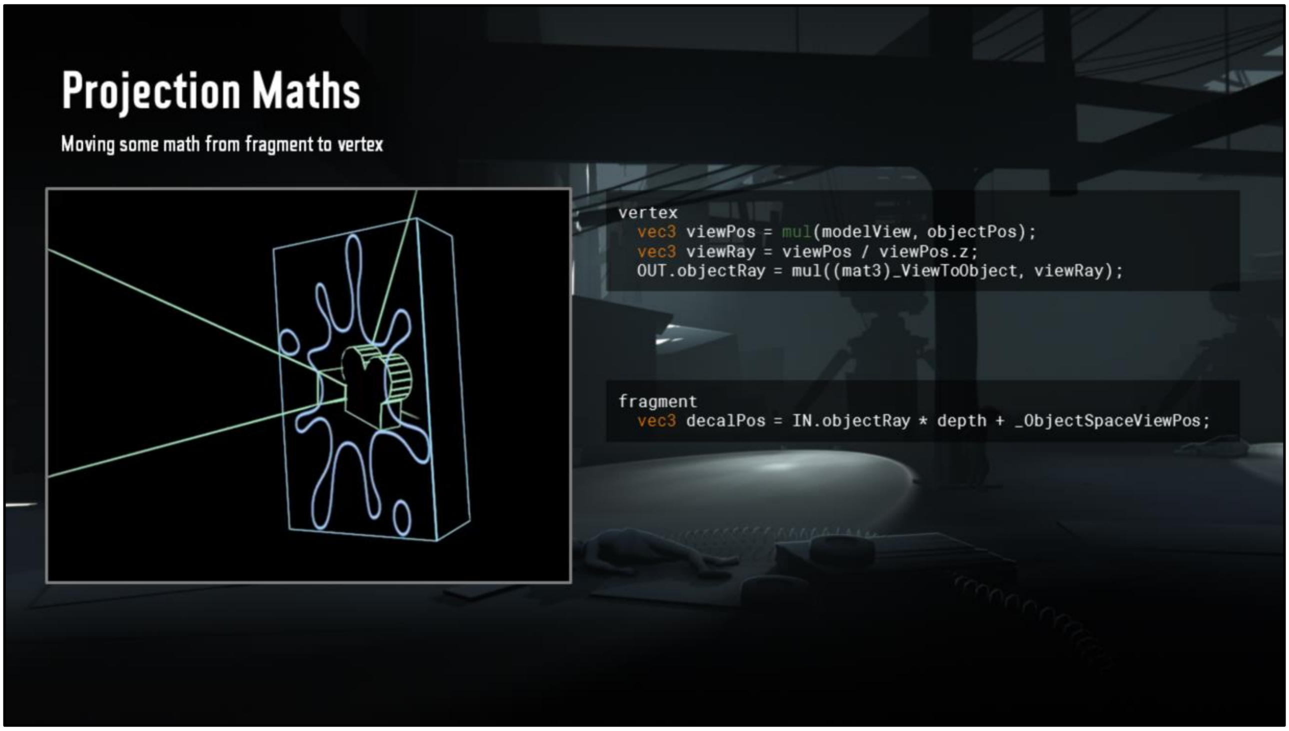


Projection Maths

Moving some math from fragment to vertex

```
vertex
  vec3 viewPos = mul(modelView, objectPos);
  vec3 viewRay = viewPos / viewPos.z;
  OUT.objectRay = mul((mat3)_ViewToObject, viewRay);
```

And our scale



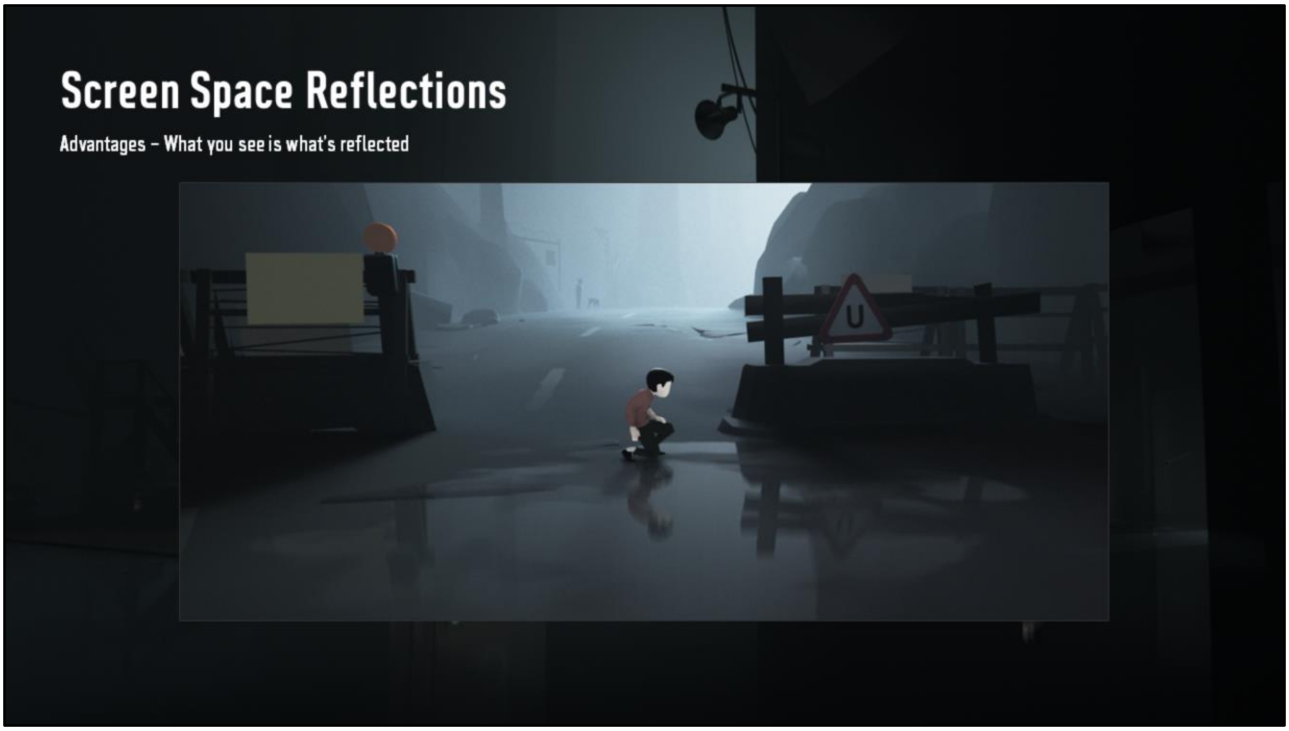
Projection Maths

Moving some math from fragment to vertex

```
vertex
  vec3 viewPos = mul(modelView, objectPos);
  vec3 viewRay = viewPos / viewPos.z;
  OUT.objectRay = mul((mat3)_ViewToObject, viewRay);
```

```
fragment
  vec3 decalPos = IN.objectRay * depth + _ObjectSpaceViewPos;
```

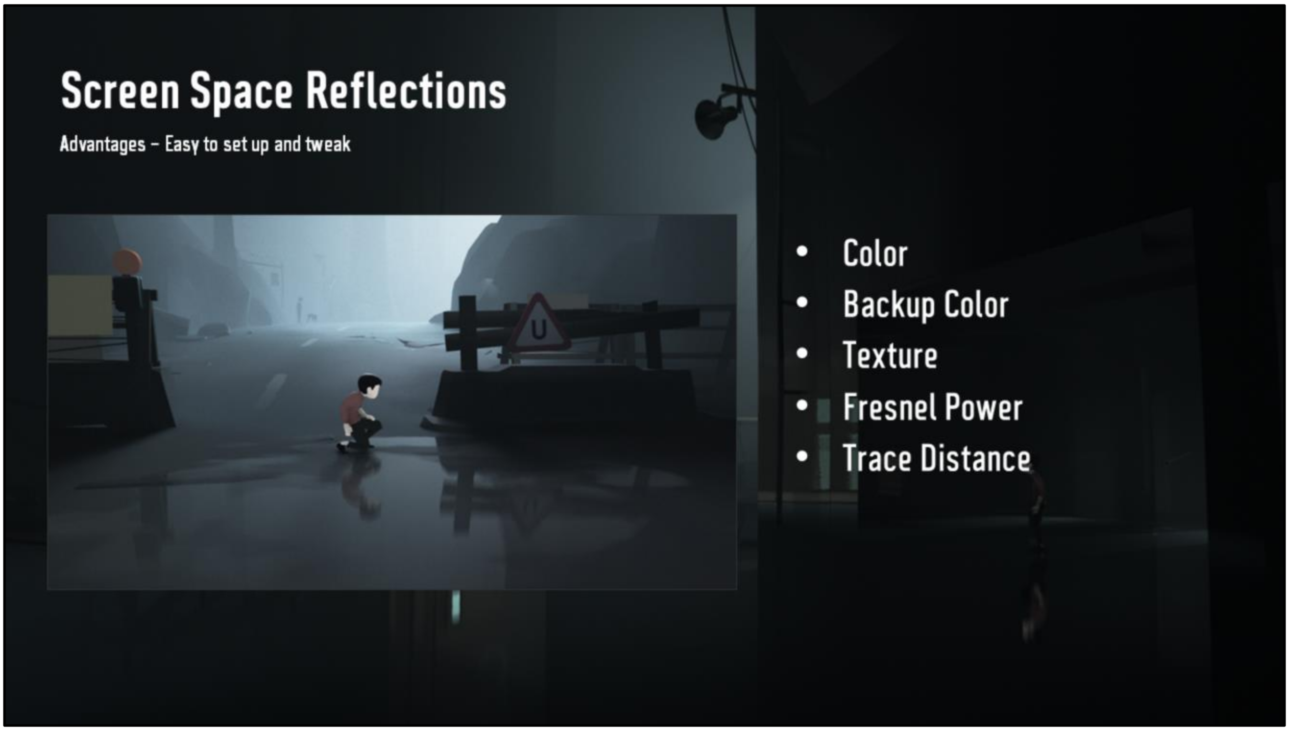
And then now we can get away with an FMA or MAD in the fragment shader for our decal position!



Screen Space Reflections

Advantages – What you see is what's reflected

Another decal type we have are SSR or Screen Space Reflection.
We don't do it as a post effect, but rather locally, as puddles.
We prefare this over planar oblique clipping camera reflections, since what you see is what's reflected, no tagging of reflected objects or imposter modelling required, and also the lighting and shading doesn't need to be replicated in forward.

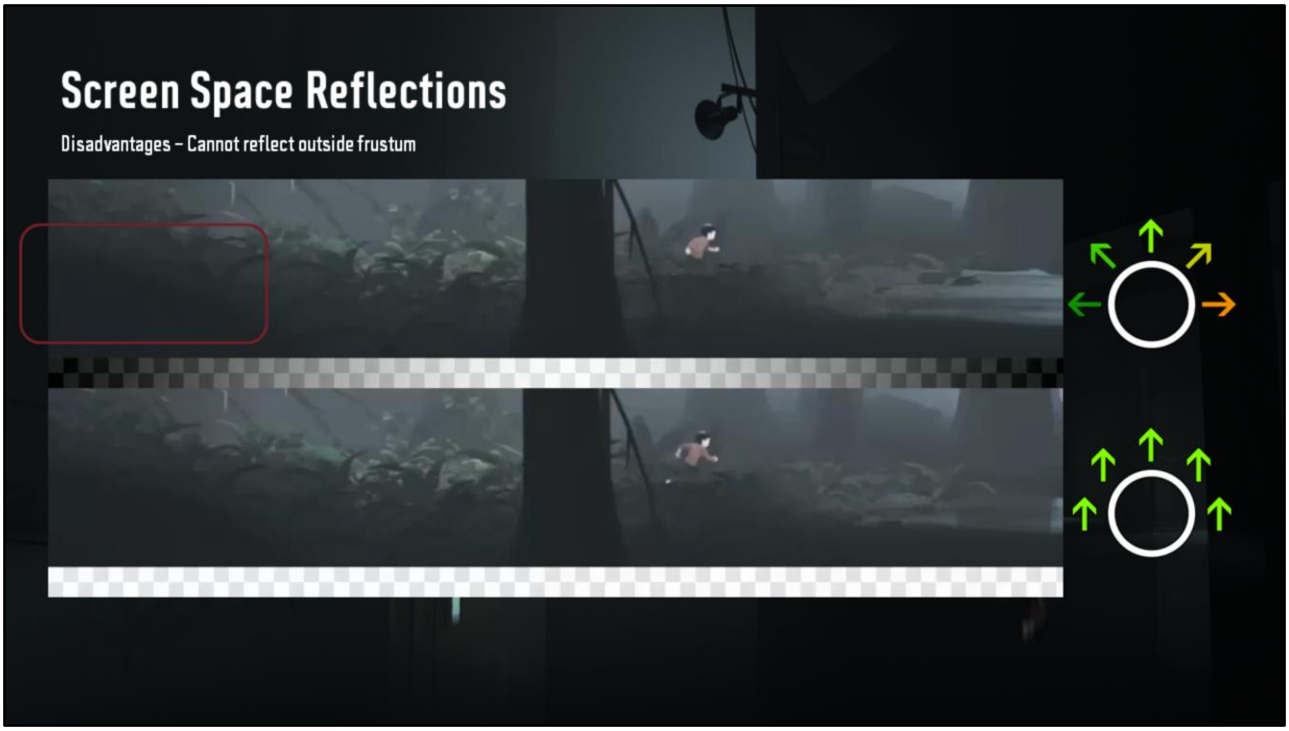


Screen Space Reflections

Advantages – Easy to set up and tweak

- Color
- Backup Color
- Texture
- Fresnel Power
- Trace Distance

Setup is almost just drag and drop, we just need to define a color.
Backup color, which is what we fade to in case our ray misses.
Texture that represents the puddle shape.
Fresnel power to control opacity of the reflection.
And finally trace distance, which is the only trace-affecting parameter, it simply says how far the ray needs to go, from 0 which travels nowhere to 1 which traverses entire screen.



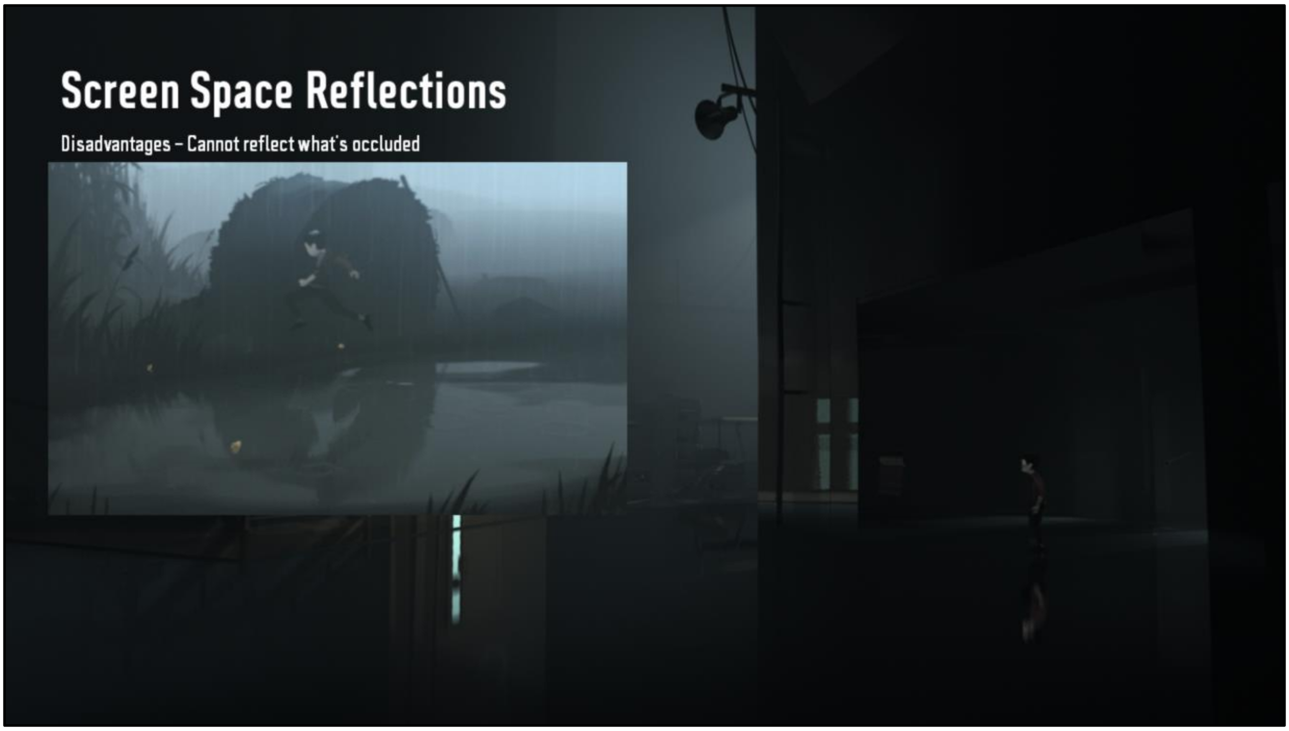
Screen Space Reflections

Disadvantages – Cannot reflect outside frustum

Disadvantages are of course, what you cannot see is not reflected.
Like in this case, where we fade out along the edges of the screen.

This we do, since a ray might fall outside the screen if it points into that direction, but we've got an advantage…

We're a 2.5D sidescroller, our camera typically points inwards to the screen and only scrolls.
So if this fade is causing issues in a scene, we simply force all rays to have no X movement, alá sDir.x *= 0.0;
Now a ray will never go out those boundaries, so we disable the fading.

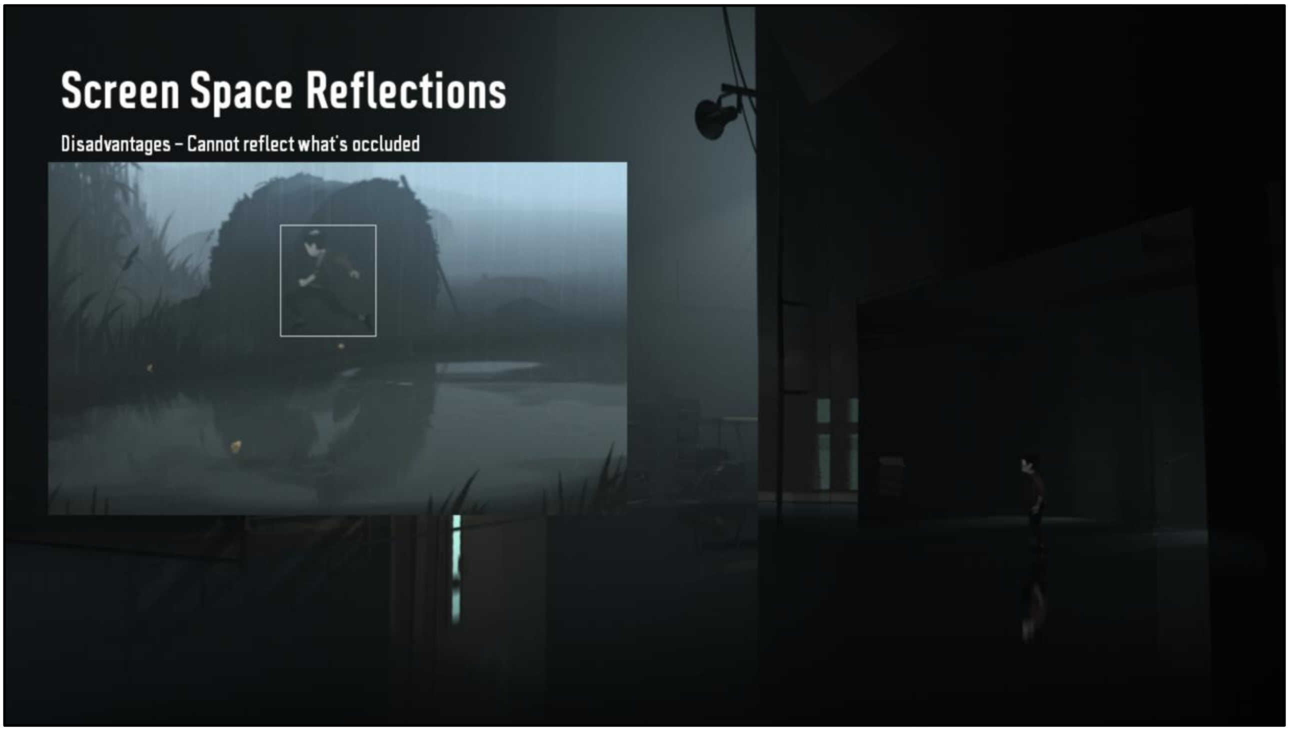


Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

Other case of what you cannot see is not reflected is occlusion.

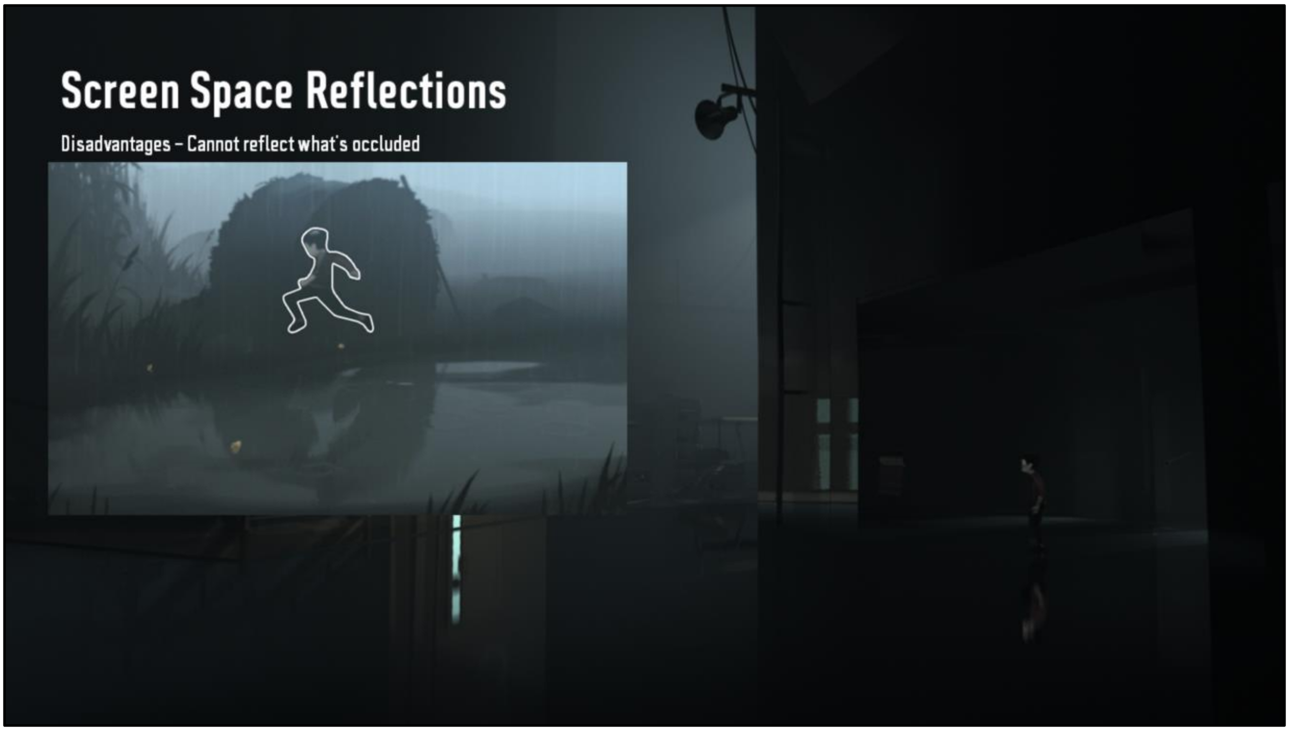
Particularly the boy, you're pretty much bound to notice that you can jump in puddles in front of walls to make ghosts of yourself, if for no other reason just to annoy me. We need not solve all occlusion and it'd be pretty hard, but we can try and solve this one in a couple of ways. Our solution is to stitch surrounding details, sometimes horizontally, sometimes vertically along the line of reflection. Let's start with horizontal.



Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

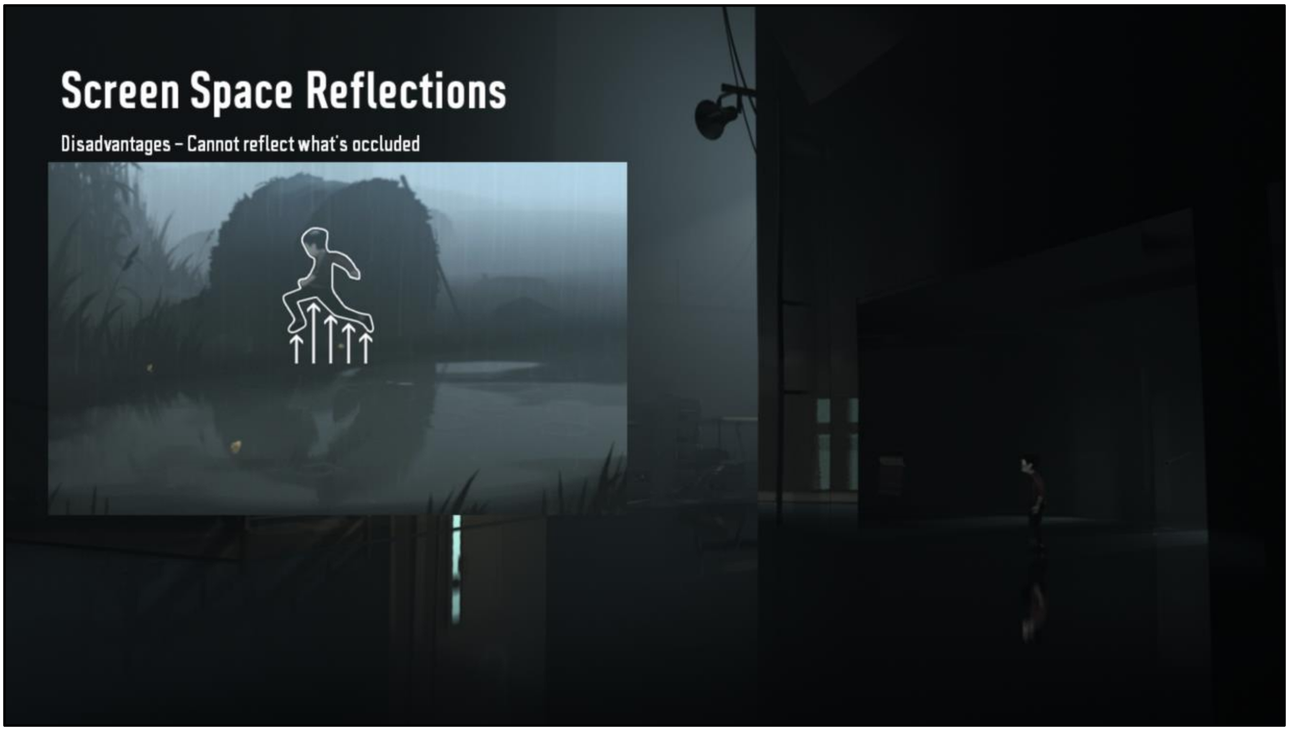
This solution starts with us making a screen space bounding box around our boy. If our ray gets inside, we're close to being boy occluded.



Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

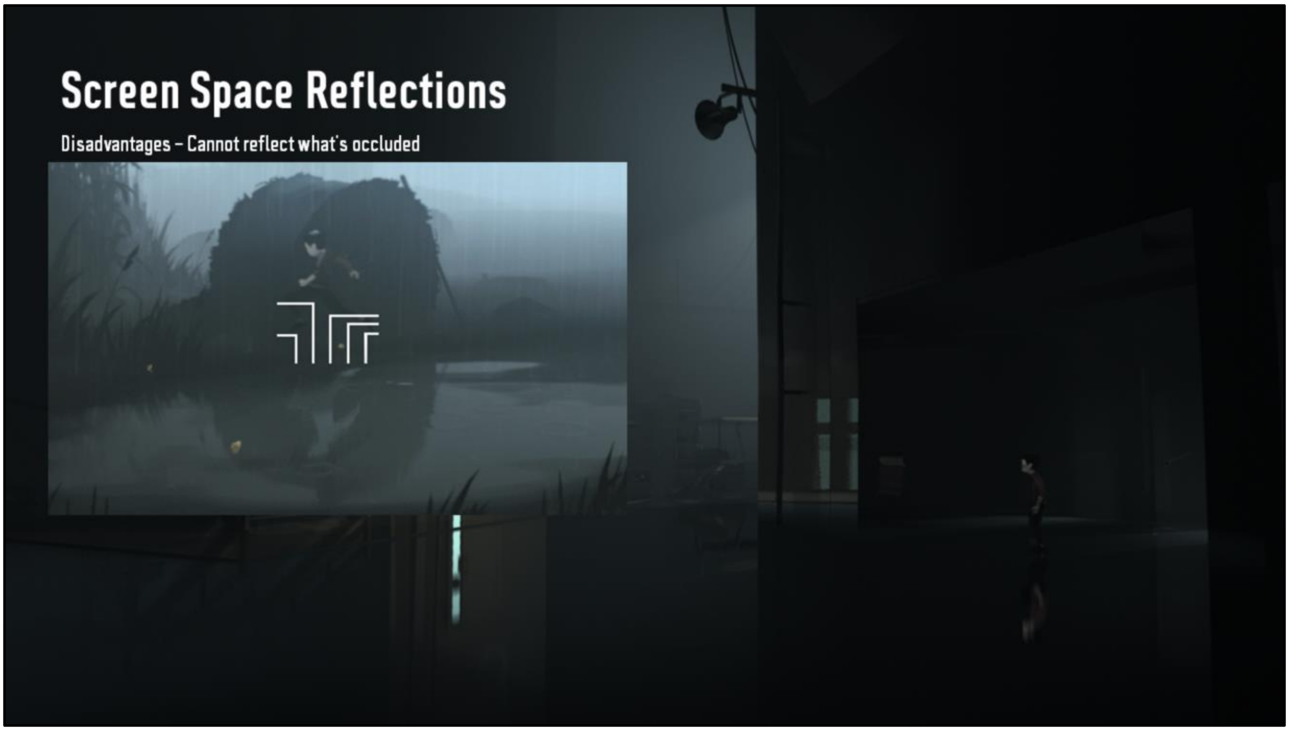
If that, plus our ray is behind the depth buffer without hitting anything, we feel confident that we're not boy occluded.



Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

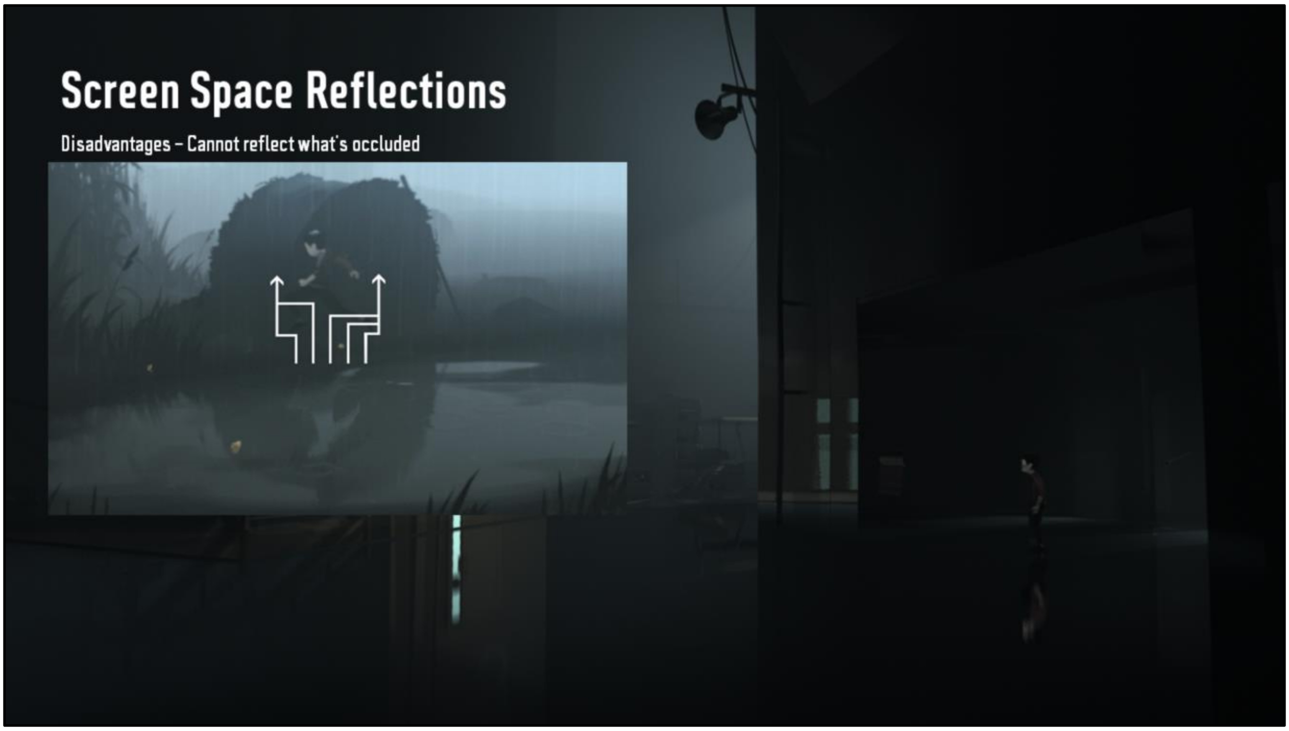
So in the case of a ray tracing into the scene and hitting this condition, what we do, is…



Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

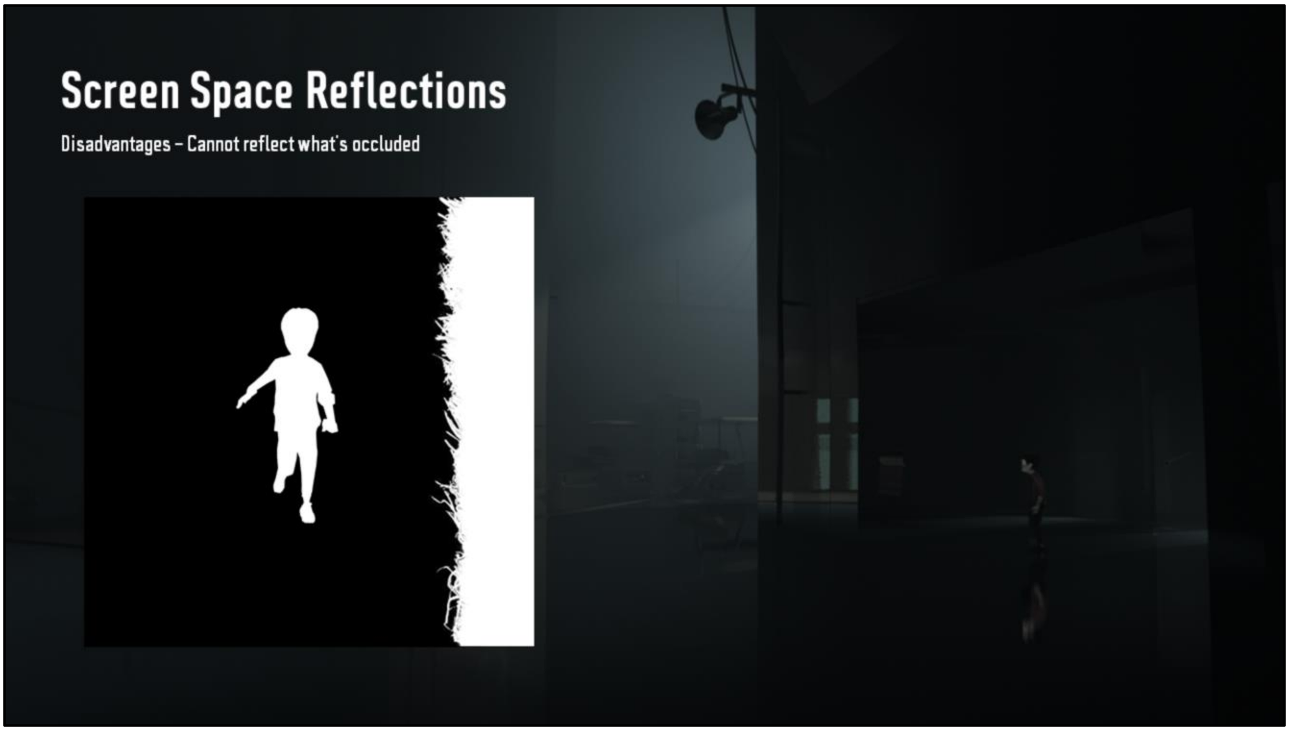
Tell the ray to move to the nearest horizontal exit of the bounding box…



Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

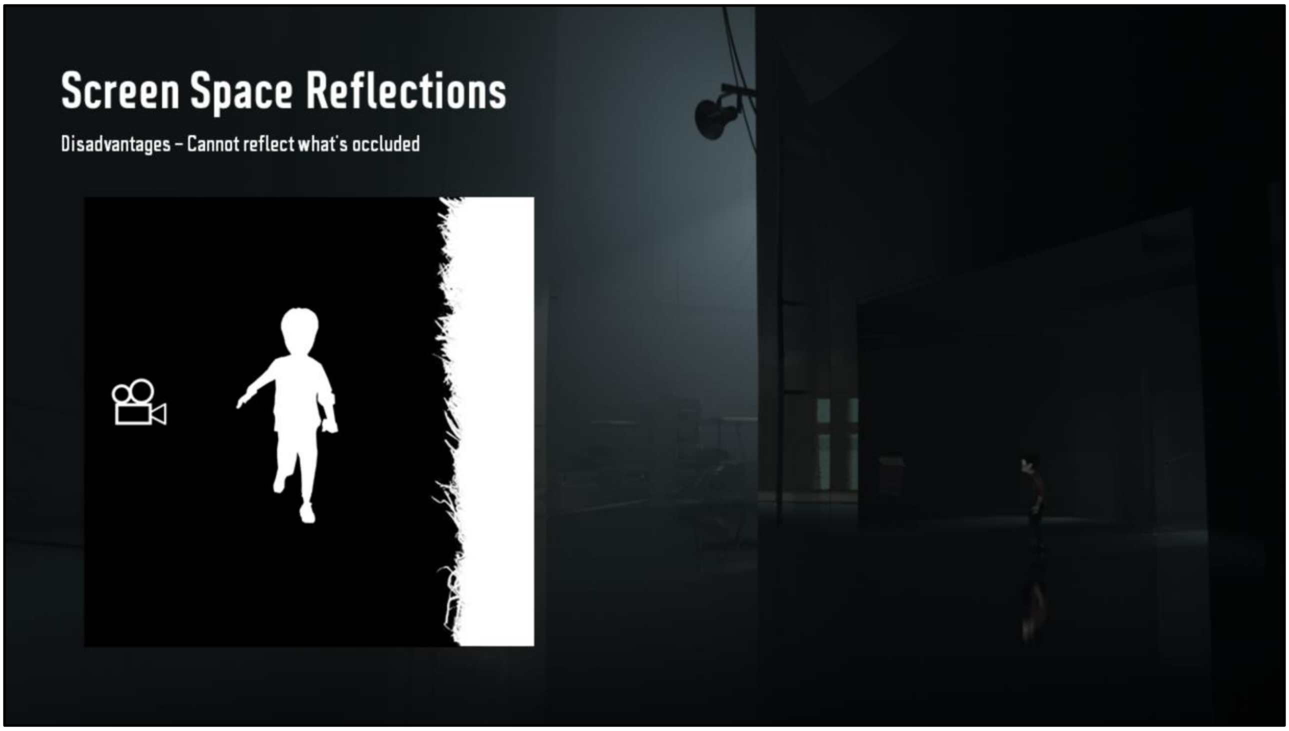
And continue tracing from there.



Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

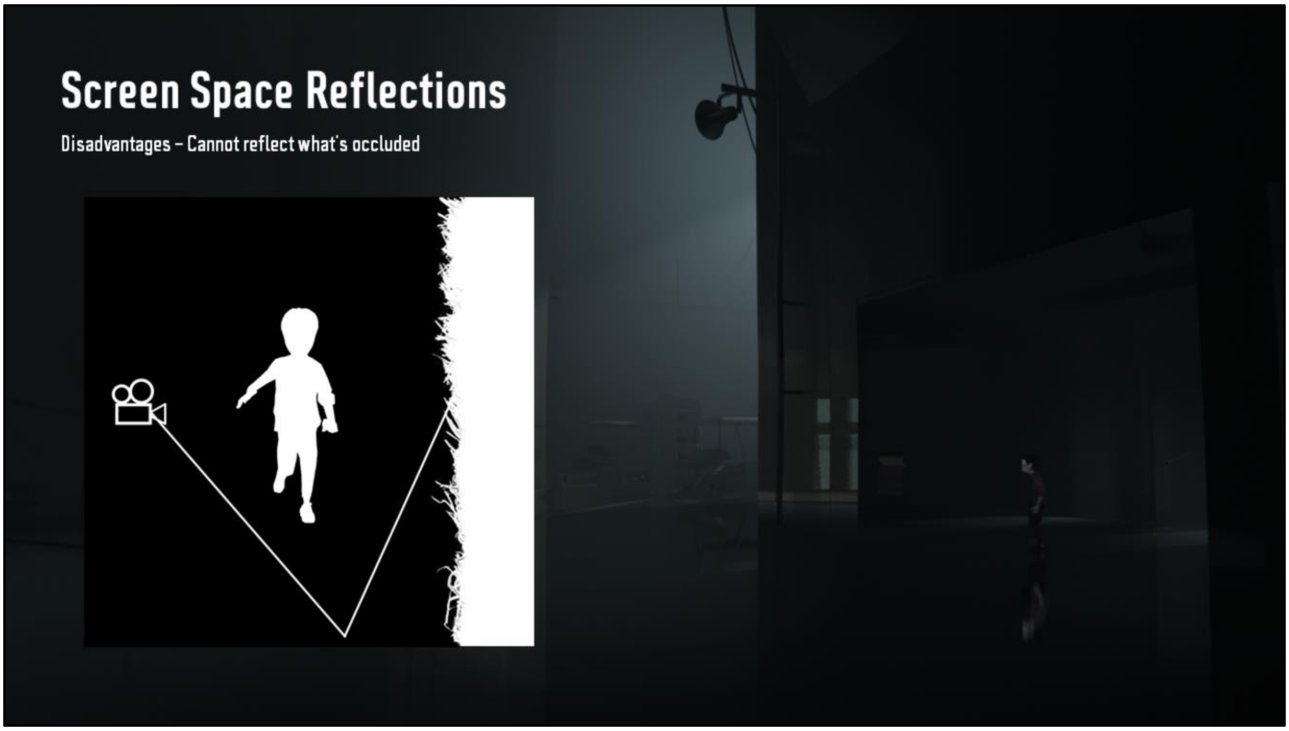
The second method doesn't need a boy bounding box, and it's much more general. Where as the previous method ends up stitching horizontally, this one stitches vertically, or in the direction of ray travel.



Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

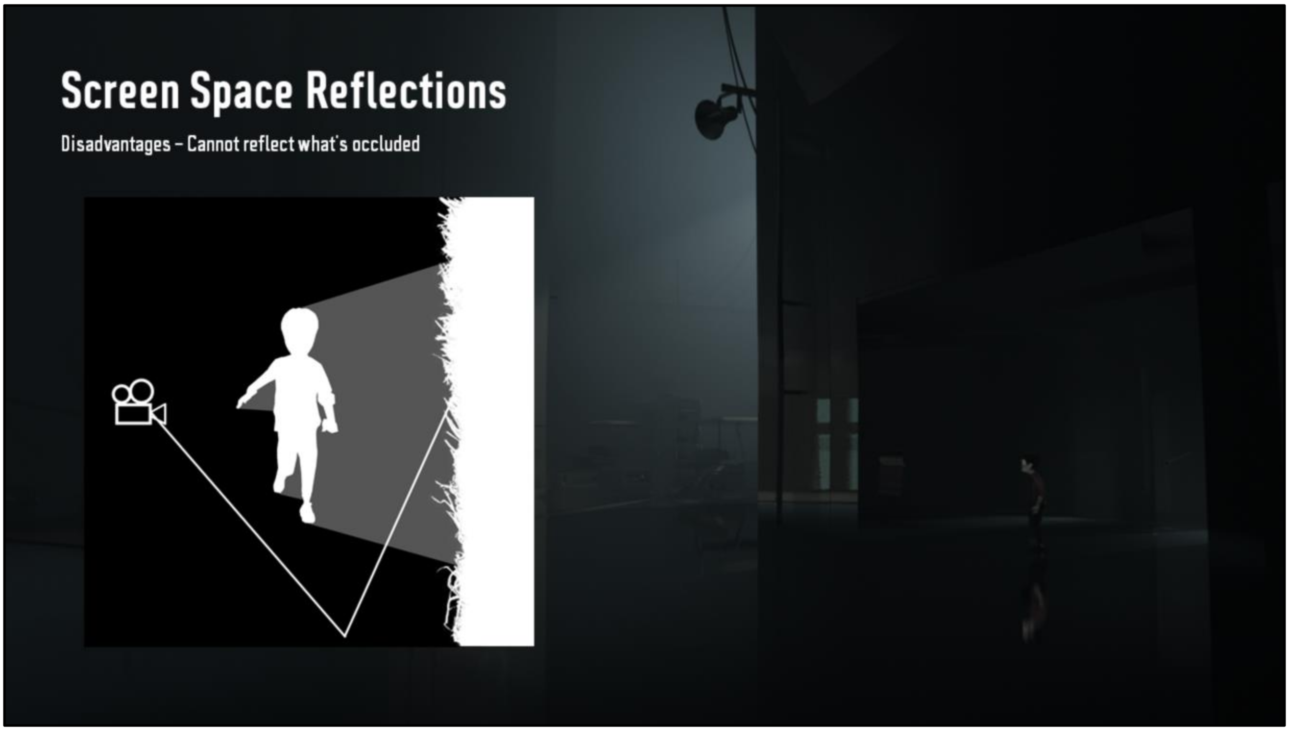
We if we're seeing the boy in front of a haystack here.



Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

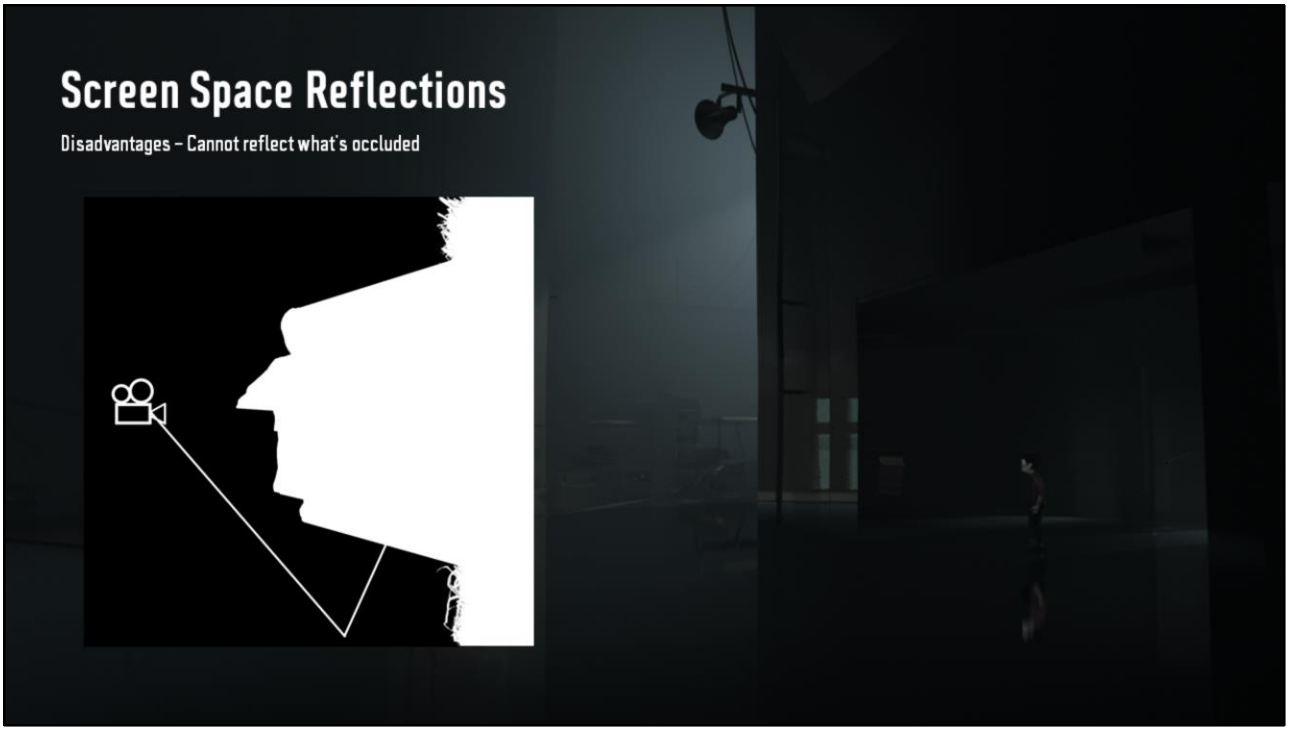
Then cast out a ray.



Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

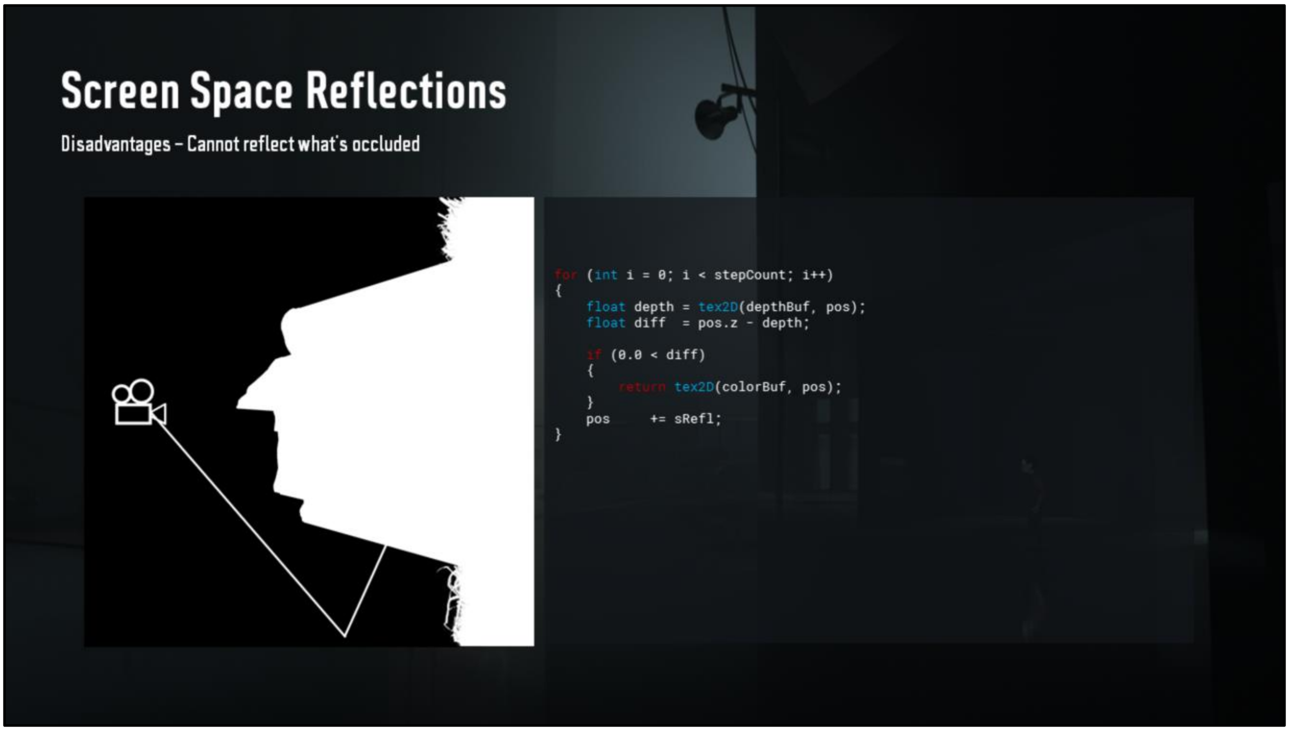
This is what we cannot see or reflect, this is our problem space.



Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

In fact this is how it looks as a height map, we'd just reflect an elongated boy

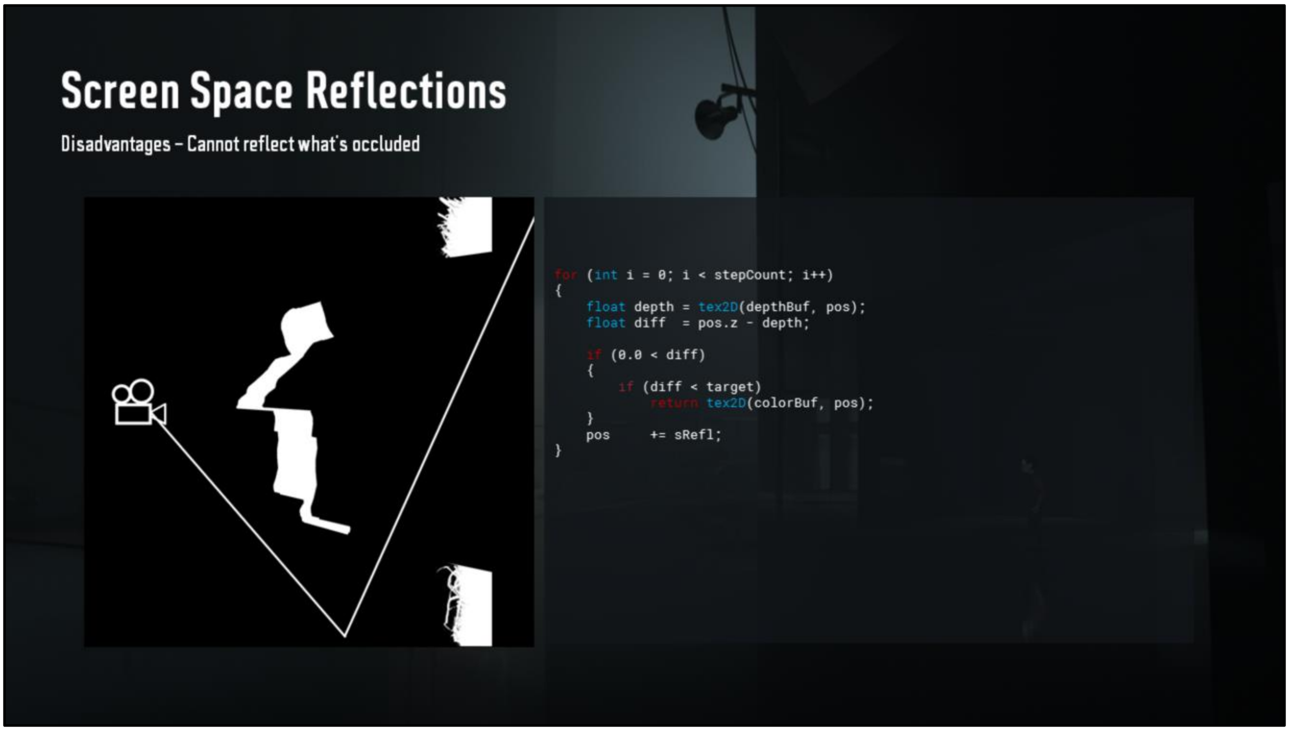


Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

```
for (int i = 0; i < stepCount; i++)
{
    float depth = tex2D(depthBuf, pos);
    float diff  = pos.z - depth;

    if (0.0 < diff)
    {
        return tex2D(colorBuf, pos);
    }
    pos     += sRefl;
}
```

Let's get the code in there.

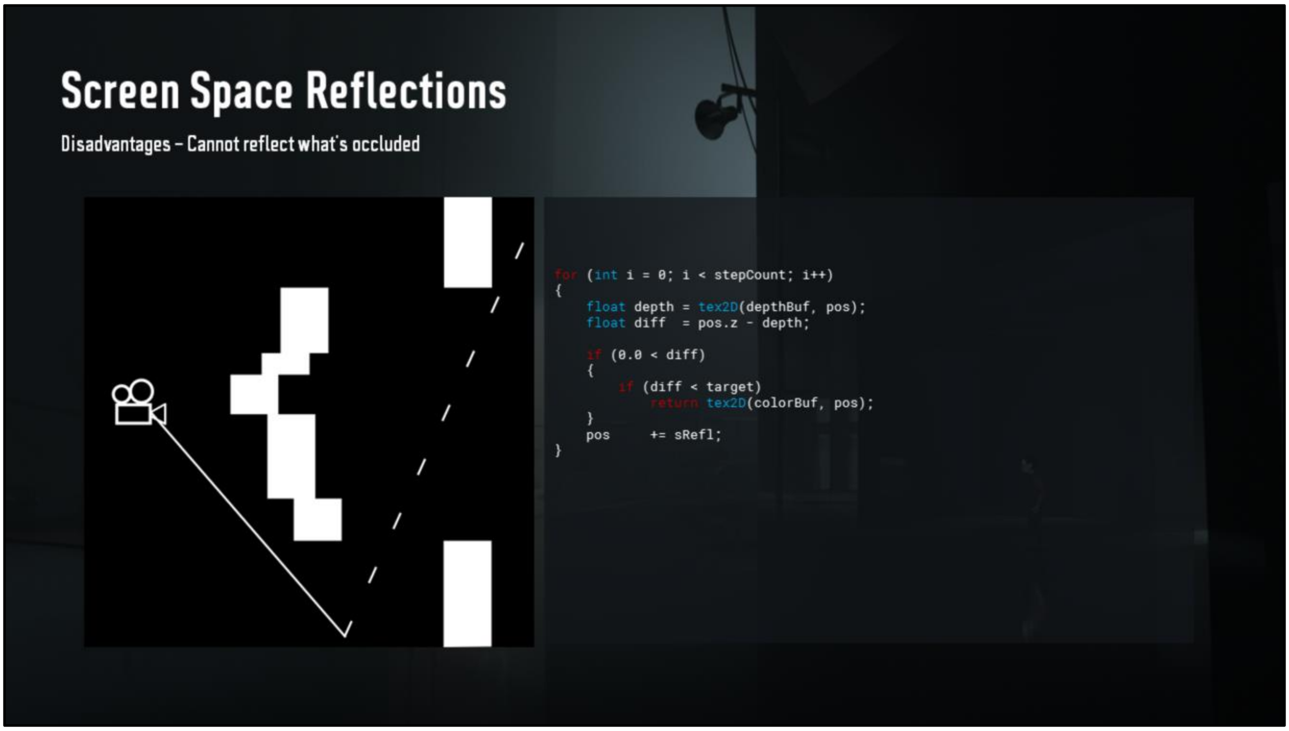


Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

```
for (int i = 0; i < stepCount; i++)
{
    float depth = tex2D(depthBuf, pos);
    float diff  = pos.z - depth;

    if (0.0 < diff)
    {
        if (diff < target)
            return tex2D(colorBuf, pos);
    }
    pos     += sRefl;
}
```

So, we don't want that elongated boy, so we assume some wall thickness with a target depth, more on that later.

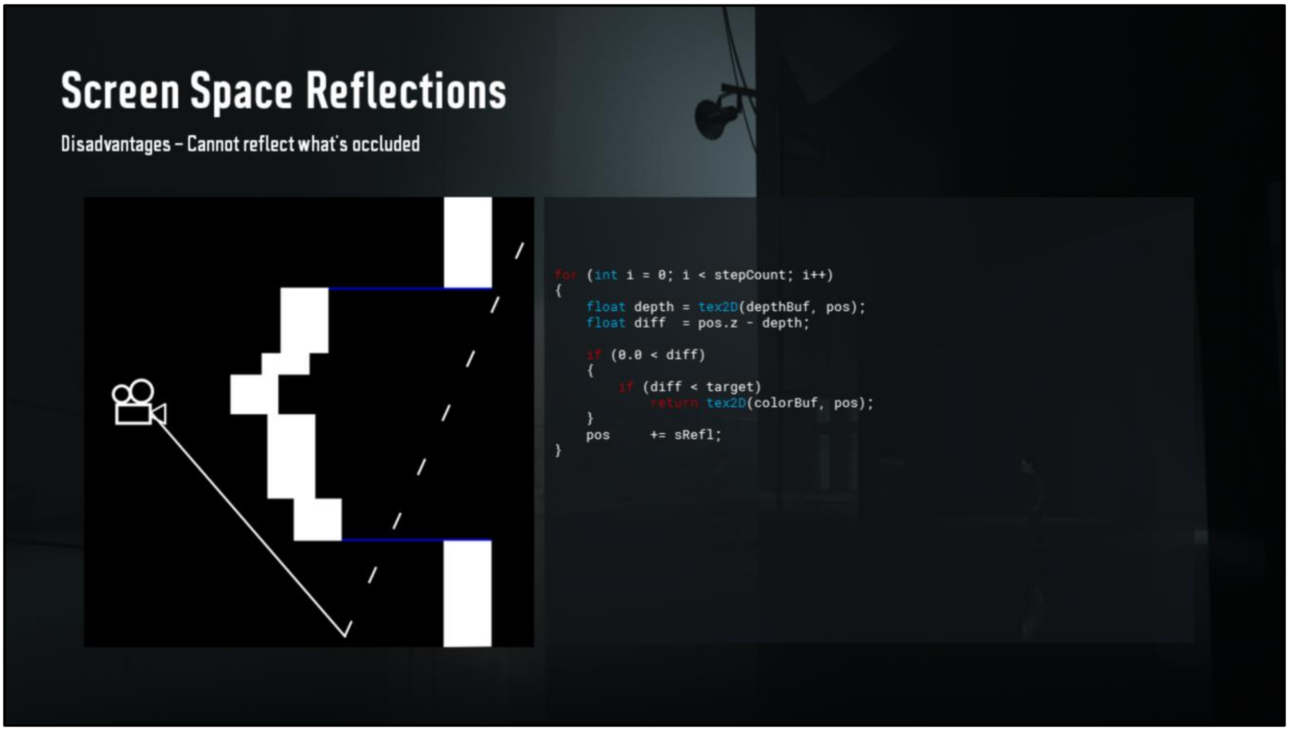


Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

```
for (int i = 0; i < stepCount; i++)
{
    float depth = tex2D(depthBuf, pos);
    float diff  = pos.z - depth;

    if (0.0 < diff)
    {
        if (diff < target)
            return tex2D(colorBuf, pos);
    }
    pos     += sRefl;
}
```

And of course it's not that nice, cause we trace in steps. Quantizing our scene a bit.

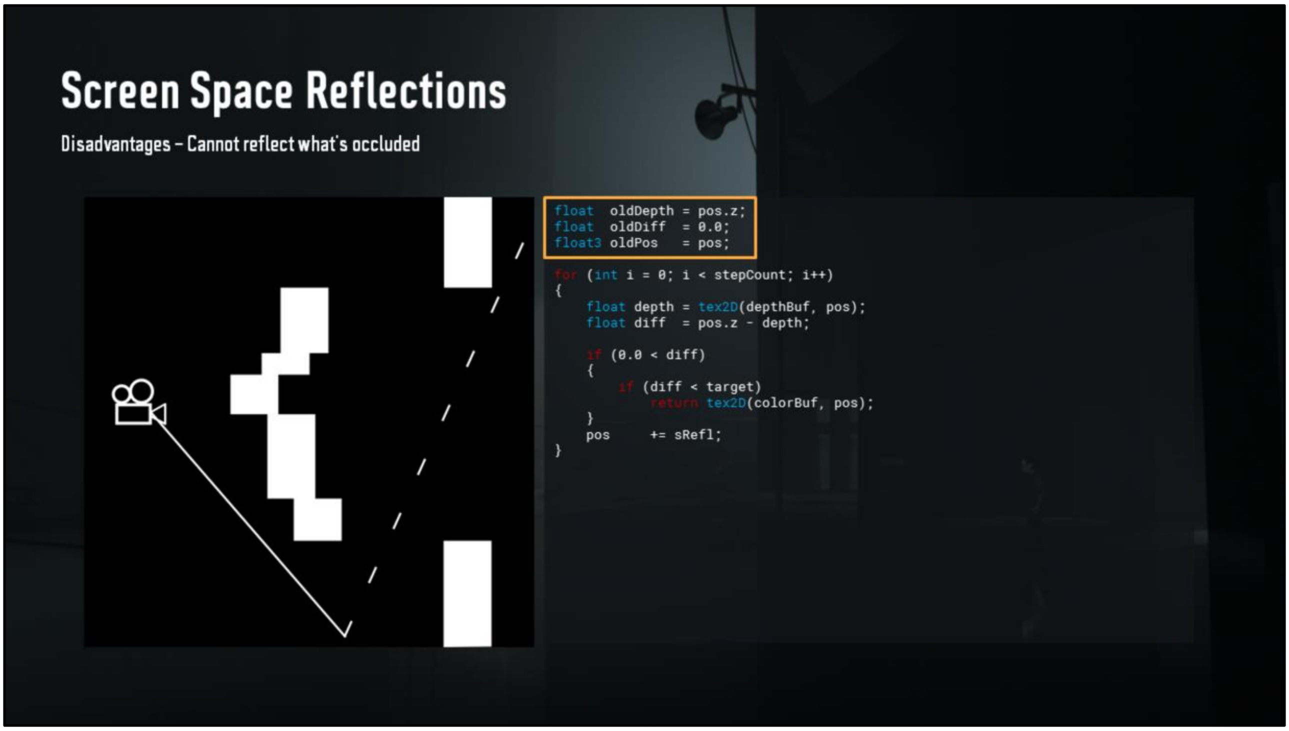


Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

```
for (int i = 0; i < stepCount; i++)
{
    float depth = tex2D(depthBuf, pos);
    float diff  = pos.z - depth;

    if (0.0 < diff)
    {
        if (diff < target)
            return tex2D(colorBuf, pos);
    }
    pos     += sRefl;
}
```

So the solution to this stitching is in these two places, we need the results before and after these blue lines to stitch out the occlusion.

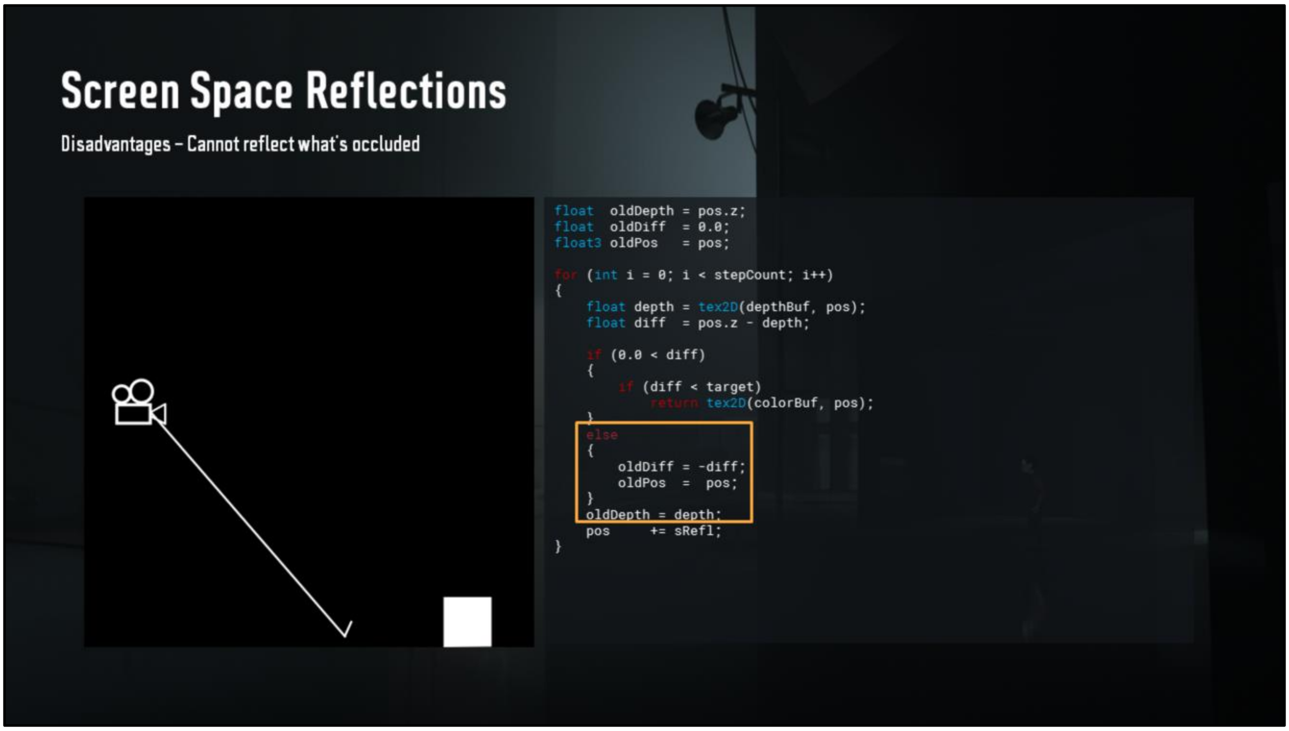


Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

```
float  oldDepth = pos.z;
float  oldDiff  = 0.0;
float3 oldPos   = pos;

for (int i = 0; i < stepCount; i++)
{
    float depth = tex2D(depthBuf, pos);
    float diff  = pos.z - depth;

    if (0.0 < diff)
    {
        if (diff < target)
            return tex2D(colorBuf, pos);
    }
    pos     += sRefl;
}
```

To do so, we're gonna need some state.

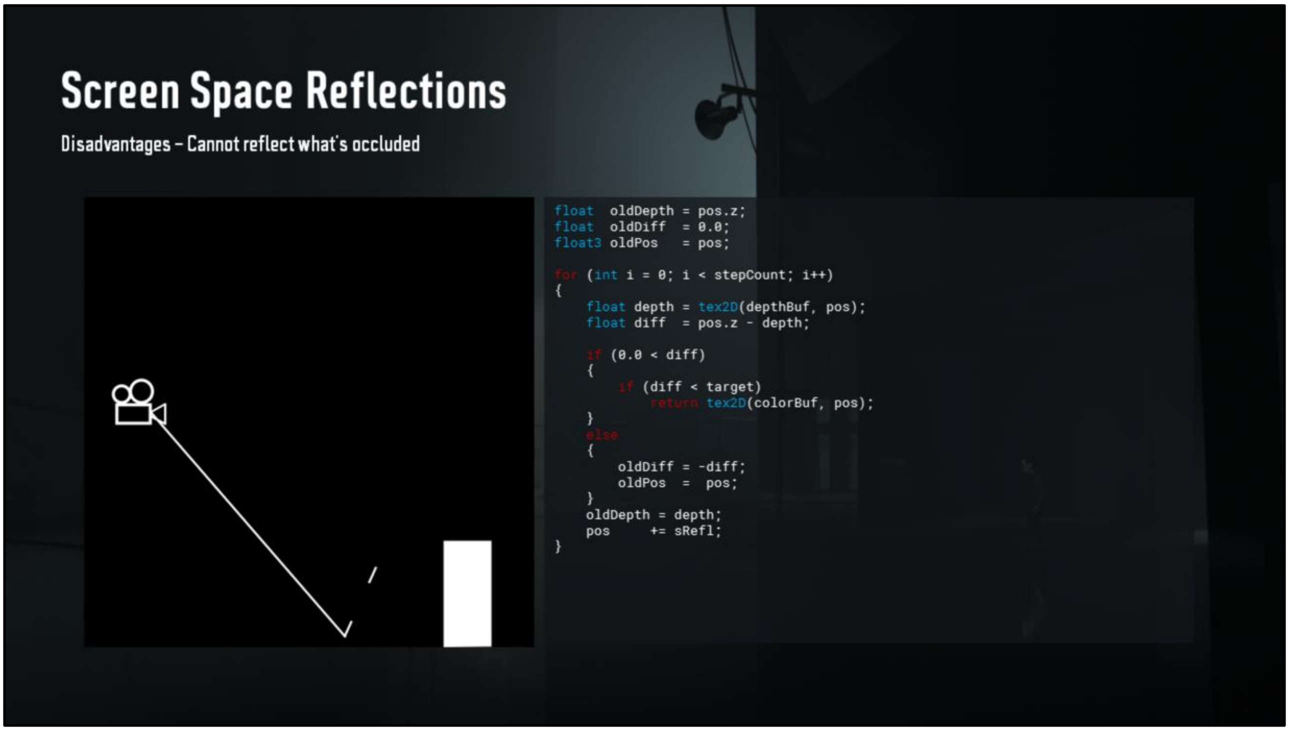


Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

```
float  oldDepth = pos.z;
float  oldDiff  = 0.0;
float3 oldPos   = pos;

for (int i = 0; i < stepCount; i++)
{
    float depth = tex2D(depthBuf, pos);
    float diff  = pos.z - depth;

    if (0.0 < diff)
    {
        if (diff < target)
            return tex2D(colorBuf, pos);
    }
    else
    {
        oldDiff = -diff;
        oldPos  =  pos;
    }
    oldDepth = depth;
    pos     += sRefl;
}
```

Let's start tracing, first step is in the clear, so we save the last distance to the depth, as well as the position for this sample. And in any case we save the depth from this sample.

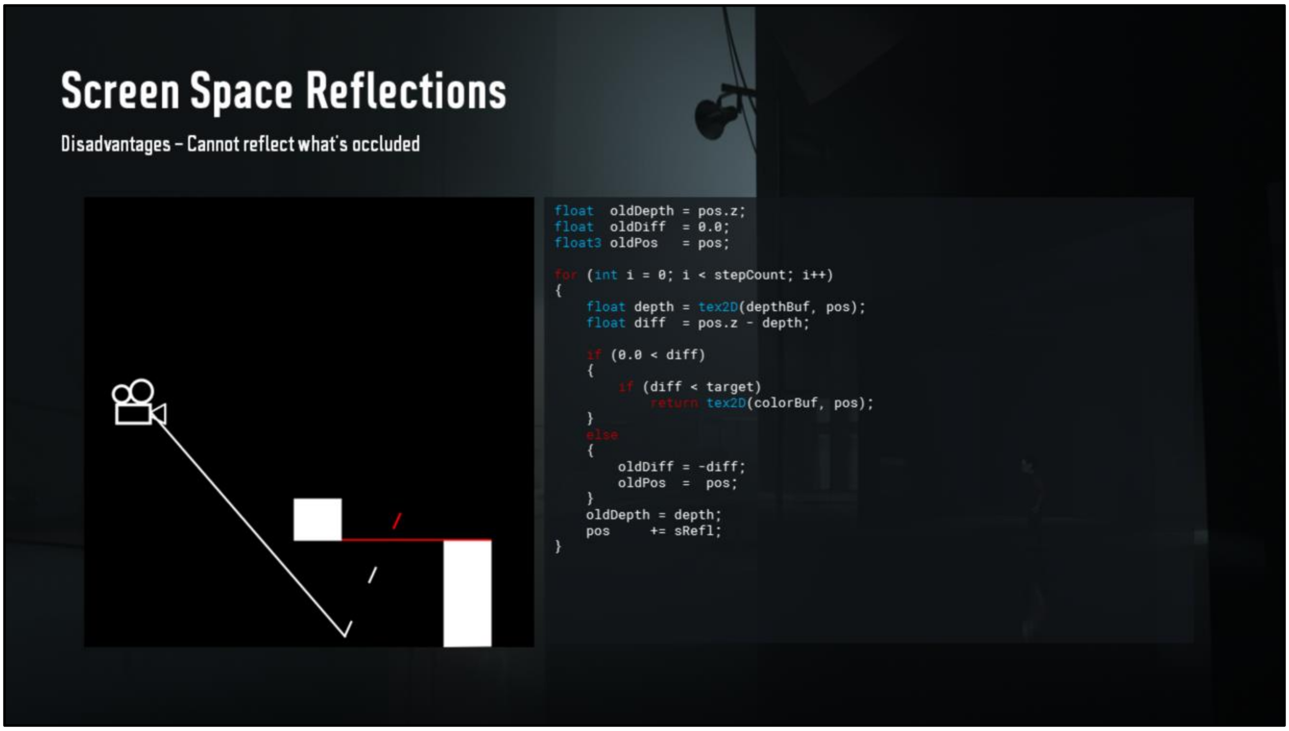


Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

```
float  oldDepth = pos.z;
float  oldDiff  = 0.0;
float3 oldPos   = pos;

for (int i = 0; i < stepCount; i++)
{
    float depth = tex2D(depthBuf, pos);
    float diff  = pos.z - depth;

    if (0.0 < diff)
    {
        if (diff < target)
            return tex2D(colorBuf, pos);
    }
    else
    {
        oldDiff = -diff;
        oldPos  =  pos;
    }
    oldDepth = depth;
    pos     += sRefl;
}
```

This step is also in the clear, so same goes.

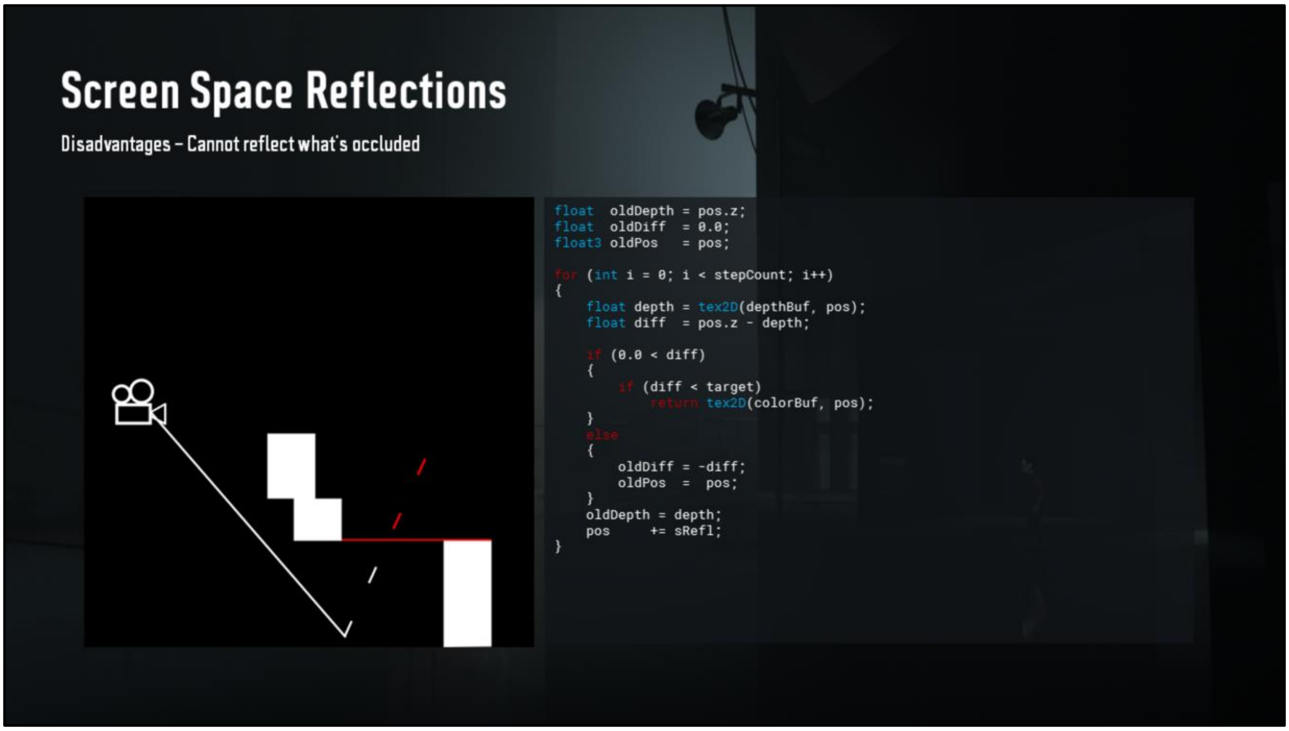


Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

```
float  oldDepth = pos.z;
float  oldDiff  = 0.0;
float3 oldPos   = pos;

for (int i = 0; i < stepCount; i++)
{
    float depth = tex2D(depthBuf, pos);
    float diff  = pos.z - depth;

    if (0.0 < diff)
    {
        if (diff < target)
            return tex2D(colorBuf, pos);
    }
    else
    {
        oldDiff = -diff;
        oldPos  =  pos;
    }
    oldDepth = depth;
    pos     += sRefl;
}
```

Whoops, we found a wall, we no longer save the distance to the depth, or the pre-occluded position.

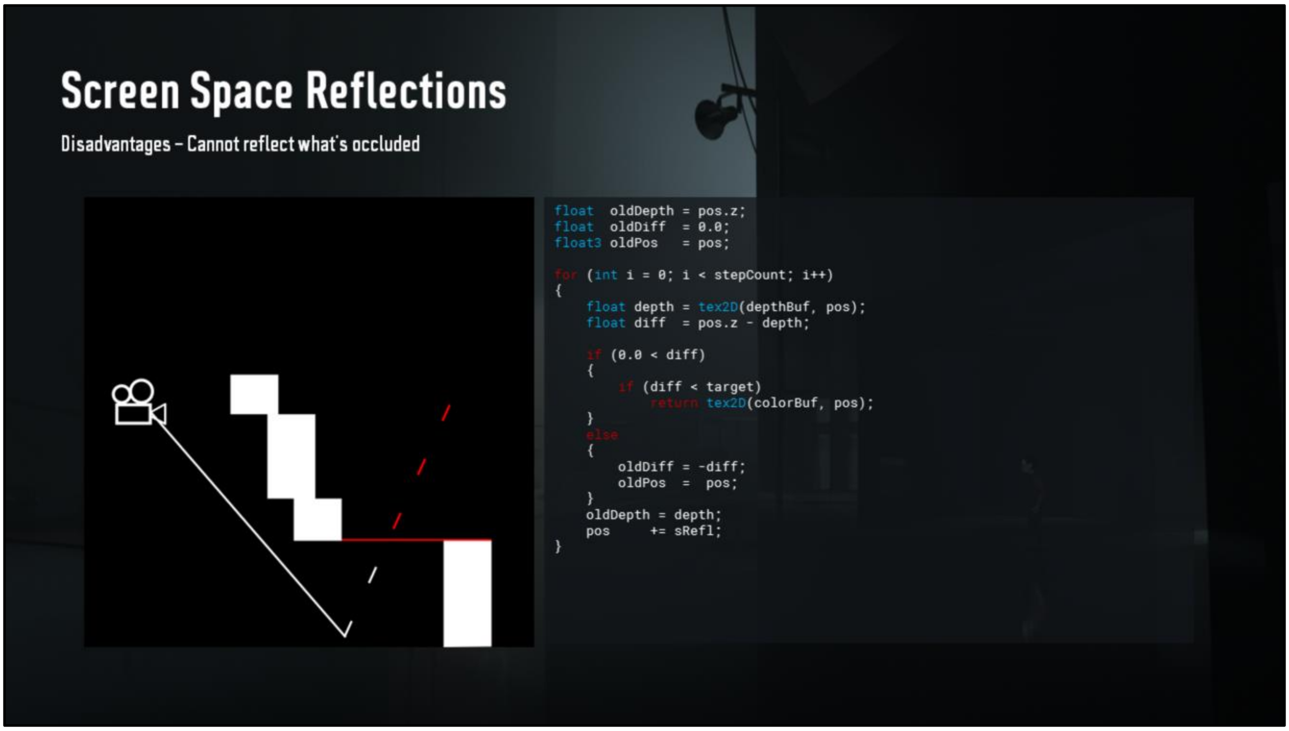


Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

```
float  oldDepth = pos.z;
float  oldDiff  = 0.0;
float3 oldPos   = pos;

for (int i = 0; i < stepCount; i++)
{
    float depth = tex2D(depthBuf, pos);
    float diff  = pos.z - depth;

    if (0.0 < diff)
    {
        if (diff < target)
            return tex2D(colorBuf, pos);
    }
    else
    {
        oldDiff = -diff;
        oldPos  =  pos;
    }
    oldDepth = depth;
    pos     += sRefl;
}
```

Let's keep stepping...

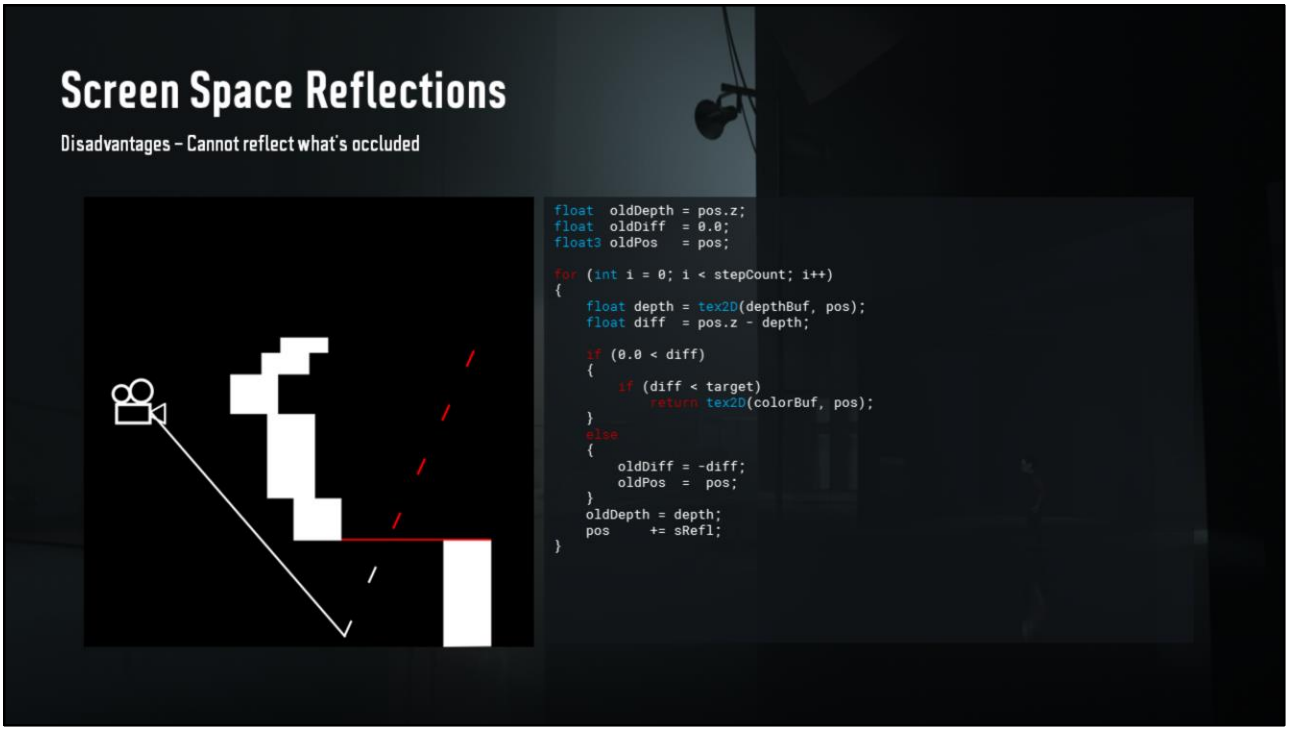


Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

```
float  oldDepth = pos.z;
float  oldDiff  = 0.0;
float3 oldPos   = pos;

for (int i = 0; i < stepCount; i++)
{
    float depth = tex2D(depthBuf, pos);
    float diff  = pos.z - depth;

    if (0.0 < diff)
    {
        if (diff < target)
            return tex2D(colorBuf, pos);
    }
    else
    {
        oldDiff = -diff;
        oldPos  =  pos;
    }
    oldDepth = depth;
    pos     += sRefl;
}
```

…

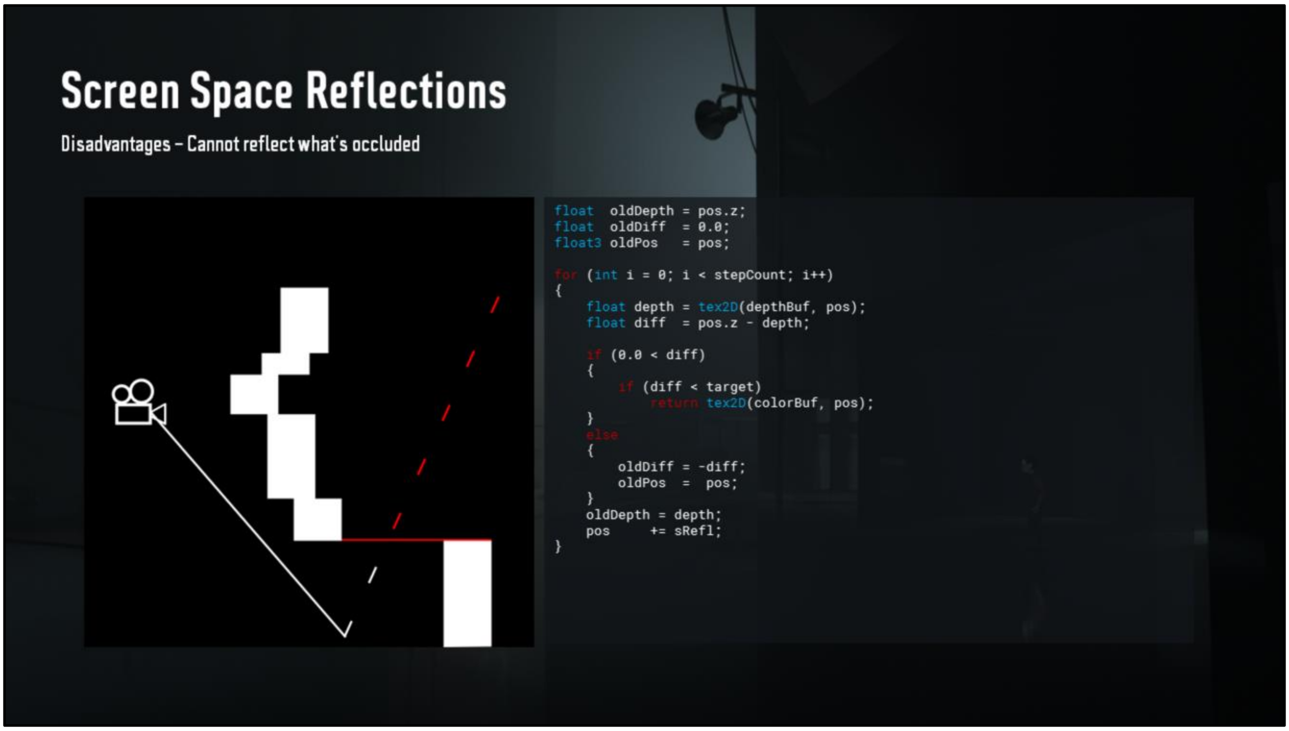


Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

```
float  oldDepth = pos.z;
float  oldDiff  = 0.0;
float3 oldPos   = pos;

for (int i = 0; i < stepCount; i++)
{
    float depth = tex2D(depthBuf, pos);
    float diff  = pos.z - depth;

    if (0.0 < diff)
    {
        if (diff < target)
            return tex2D(colorBuf, pos);
    }
    else
    {
        oldDiff = -diff;
        oldPos  =  pos;
    }
    oldDepth = depth;
    pos     += sRefl;
}
```

…

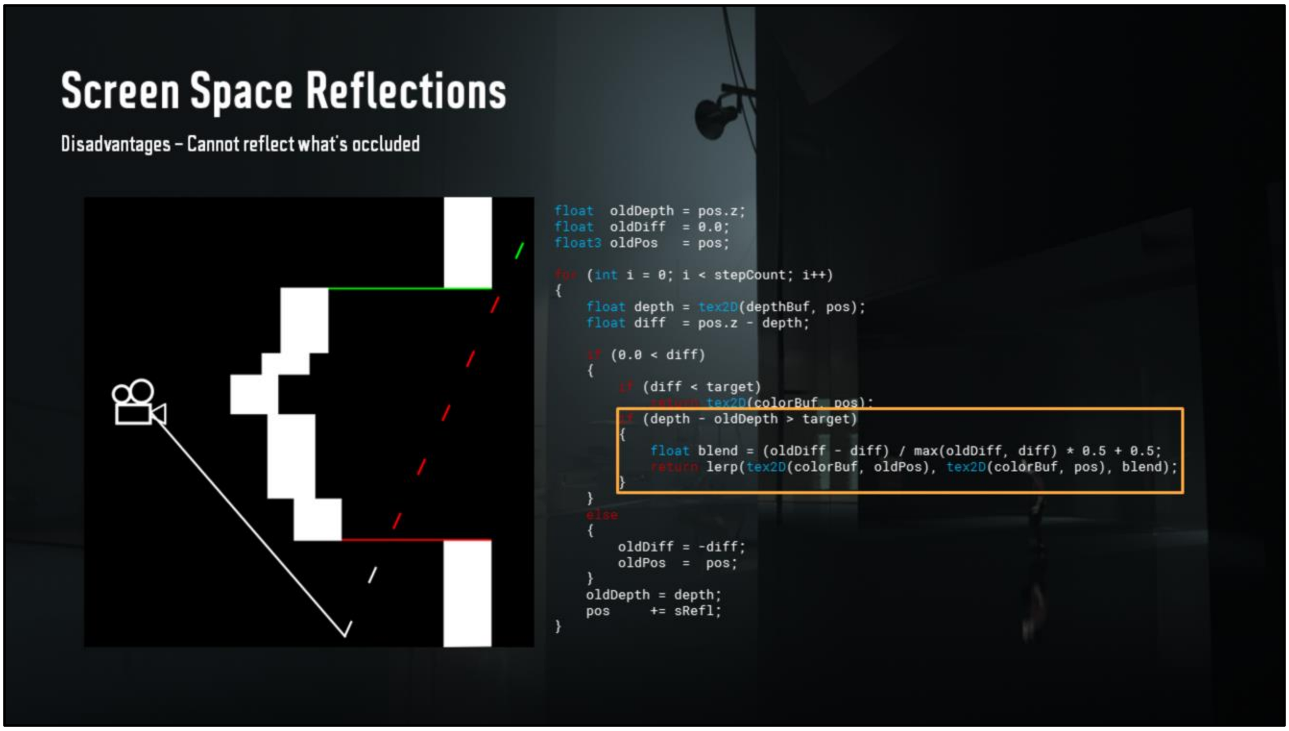


Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

```
float  oldDepth = pos.z;
float  oldDiff  = 0.0;
float3 oldPos   = pos;

for (int i = 0; i < stepCount; i++)
{
    float depth = tex2D(depthBuf, pos);
    float diff  = pos.z - depth;

    if (0.0 < diff)
    {
        if (diff < target)
            return tex2D(colorBuf, pos);
    }
    else
    {
        oldDiff = -diff;
        oldPos  =  pos;
    }
    oldDepth = depth;
    pos     += sRefl;
}
```

…

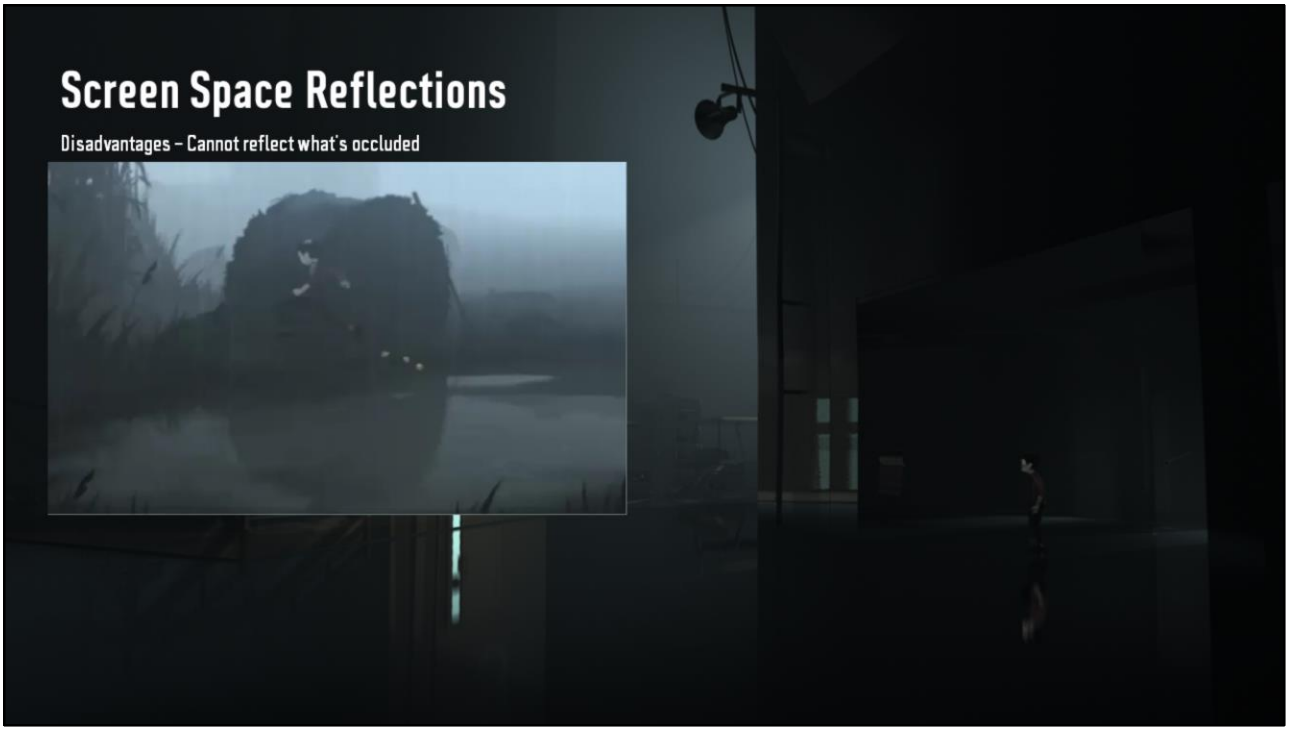


Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

```
float  oldDepth = pos.z;
float  oldDiff  = 0.0;
float3 oldPos   = pos;

for (int i = 0; i < stepCount; i++)
{
    float depth = tex2D(depthBuf, pos);
    float diff  = pos.z - depth;

    if (0.0 < diff)
    {
        if (diff < target)
            return tex2D(colorBuf, pos);
        if (depth - oldDepth > target)
        {
            float blend = (oldDiff - diff) / max(oldDiff, diff) * 0.5 + 0.5;
            return lerp(tex2D(colorBuf, oldPos), tex2D(colorBuf, pos), blend);
        }
    }
    else
    {
        oldDiff = -diff;
        oldPos  =  pos;
    }
    oldDepth = depth;
    pos     += sRefl;
}
```

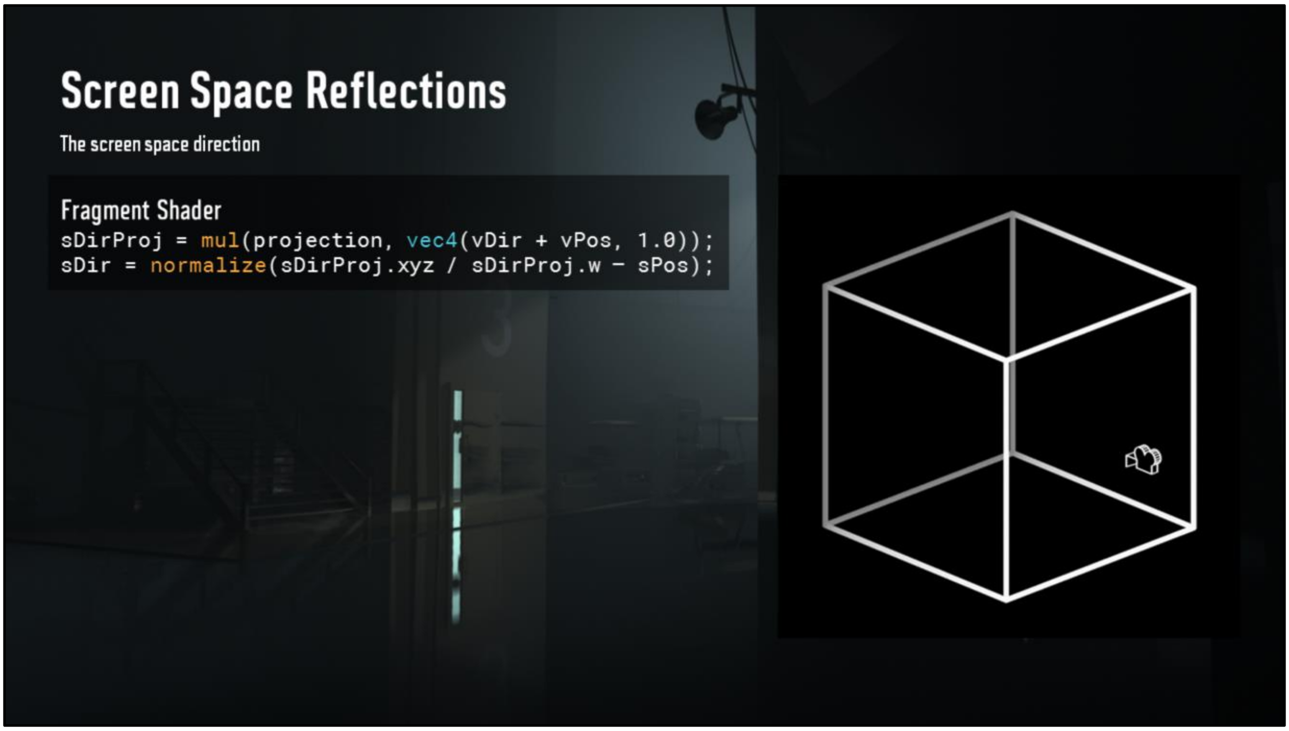
There! So we found this other wall by checking the difference between last depth sample and this one, if it's bigger than the wall thickness, that's a gap!
Now we sample a weighted mix of these two places, and return that.



Screen Space Reflections

Disadvantages – Cannot reflect what's occluded

These are hacks, but it makes SSR usable for us, and artefacts are nearly imperceptible since our game is so low detail.



Screen Space Reflections

The screen space direction

Fragment Shader

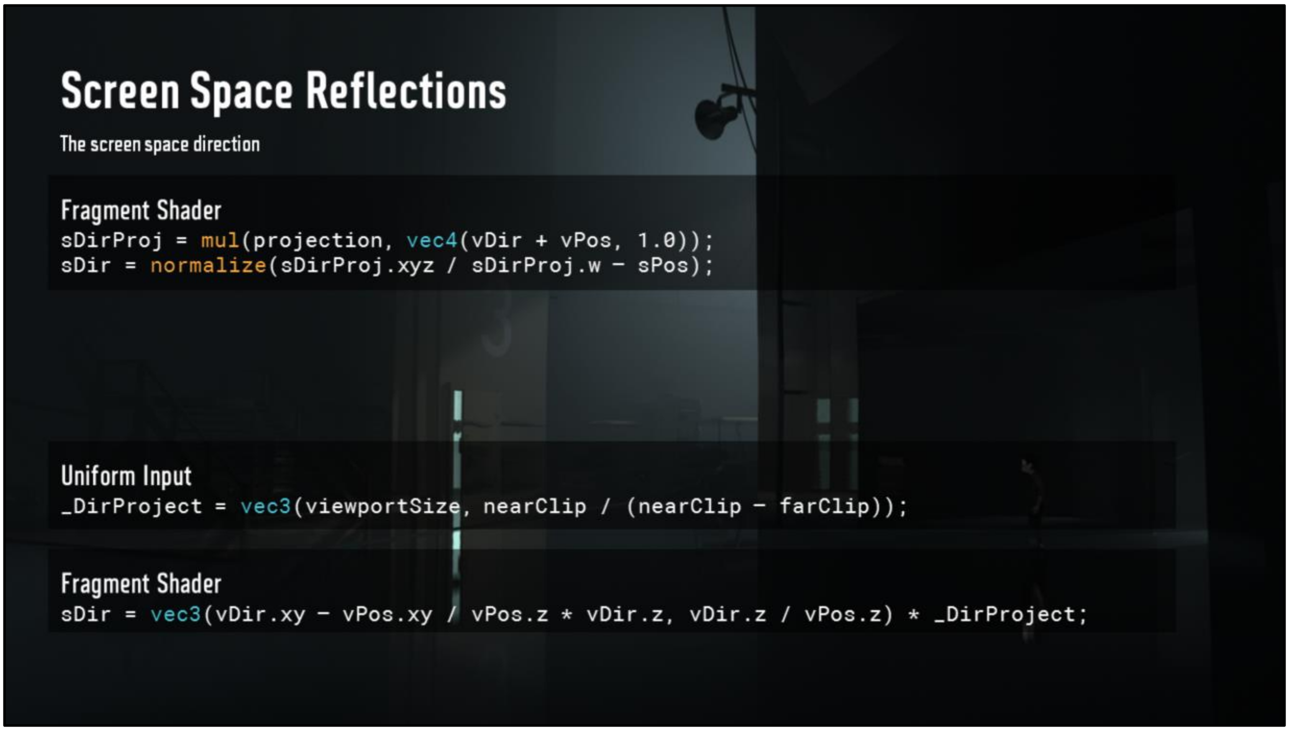
```
sDirProj = mul(projection, vec4(vDir + vPos, 1.0));
sDir = normalize(sDirProj.xyz / sDirProj.w - sPos);
```

So, for our reflections we need some directions, but not view space, rather we need screen space reflection directions.

So we need to project our directions, not positions, into frustum space.

The easy way to do this is to project a position, our reflected position.
So take the starting position and the reflection direction.
Project them using matrix, devide by w and subtract the screen position.

We don’t like this due to the matrix, normalize and divisions required.
Plus it doesn’t handle positions clipped by near plane, even though we don’t encounter that scenario, it’s neat to solve.



Screen Space Reflections

The screen space direction

Fragment Shader

```
sDirProj = mul(projection, vec4(vDir + vPos, 1.0));
sDir = normalize(sDirProj.xyz / sDirProj.w − sPos);
```

Uniform Input

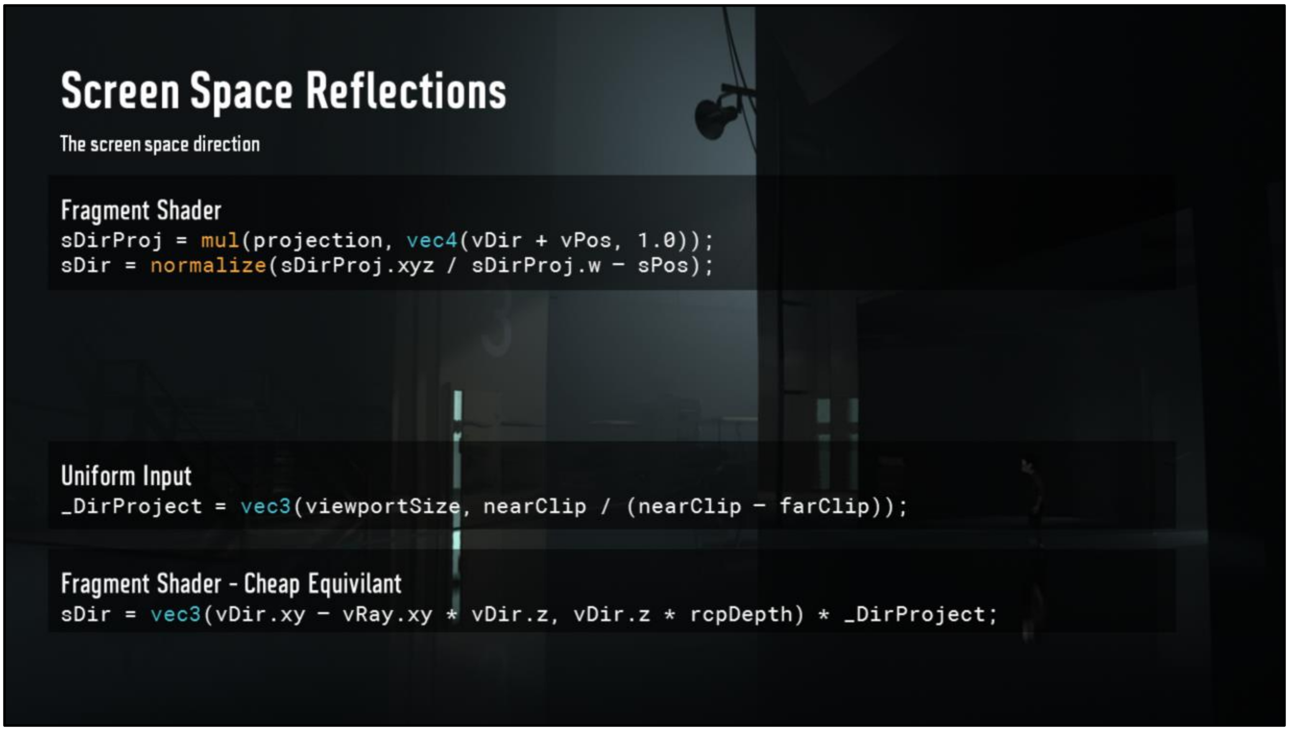
```
_DirProject = vec3(viewportSize, nearClip / (nearClip − farClip));
```

Fragment Shader

```
sDir = vec3(vDir.xy − vPos.xy / vPos.z * vDir.z, vDir.z / vPos.z) * _DirProject;
```

So our solution starts by generating a bit of data on the CPU, this vector is essentially the size of the viewport that we're using.

Then in fragment we can do this using the same data from before and this new number.
This version doesn't need normalization, but it does have a couple of divides.
Luckily we can kill those…



Screen Space Reflections

The screen space direction

Fragment Shader

```
sDirProj = mul(projection, vec4(vDir + vPos, 1.0));
sDir = normalize(sDirProj.xyz / sDirProj.w - sPos);
```

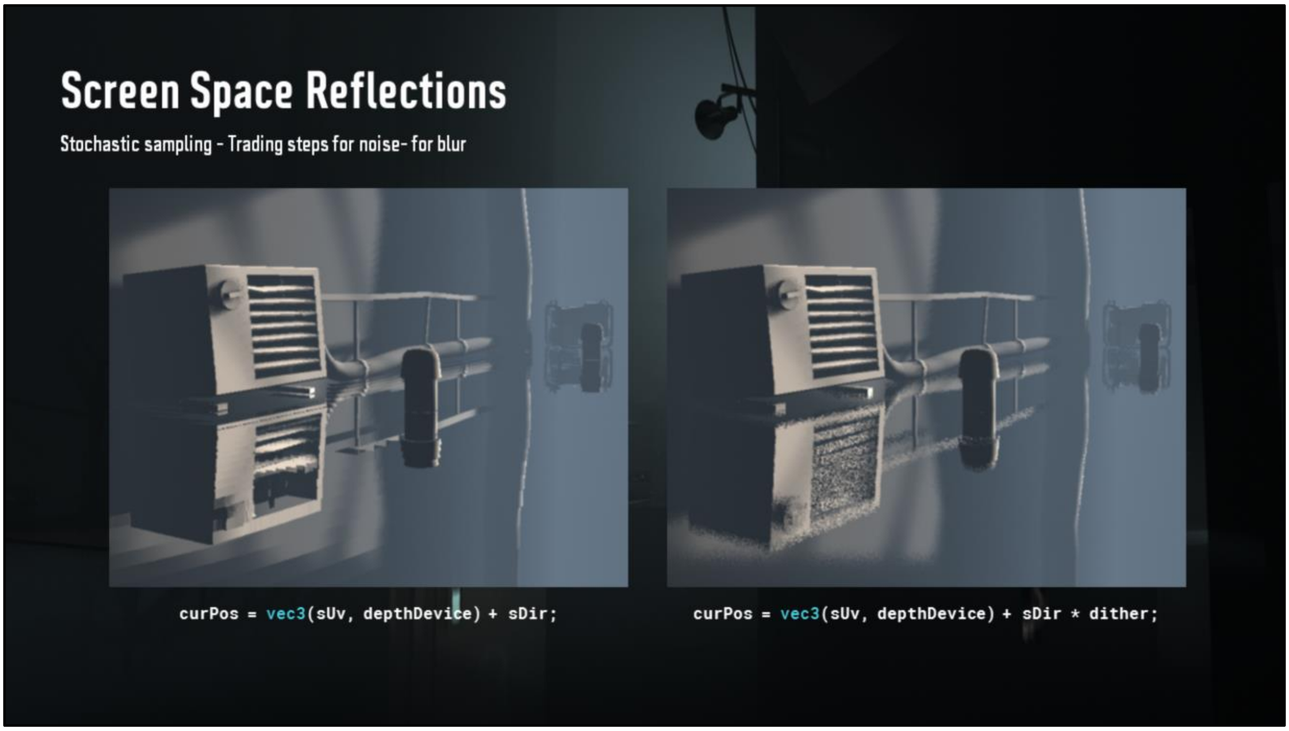
Uniform Input

```
_DirProject = vec3(viewportSize, nearClip / (nearClip - farClip));
```

Fragment Shader - Cheap Equivilant

```
sDir = vec3(vDir.xy - vRay.xy * vDir.z, vDir.z * rcpDepth) * _DirProject;
```

This, there. The new vRay.xy is the same as the vPos.xy / vPos.z from before.
Reciprocal depth is just something that’s there while converting the depth to linear.
So now we’re down to a MAD, a MOV and a couple of MUL.
This method is naturally usable with other things, not just reflections, it can be useful for SSAO vectors or anything screen space.



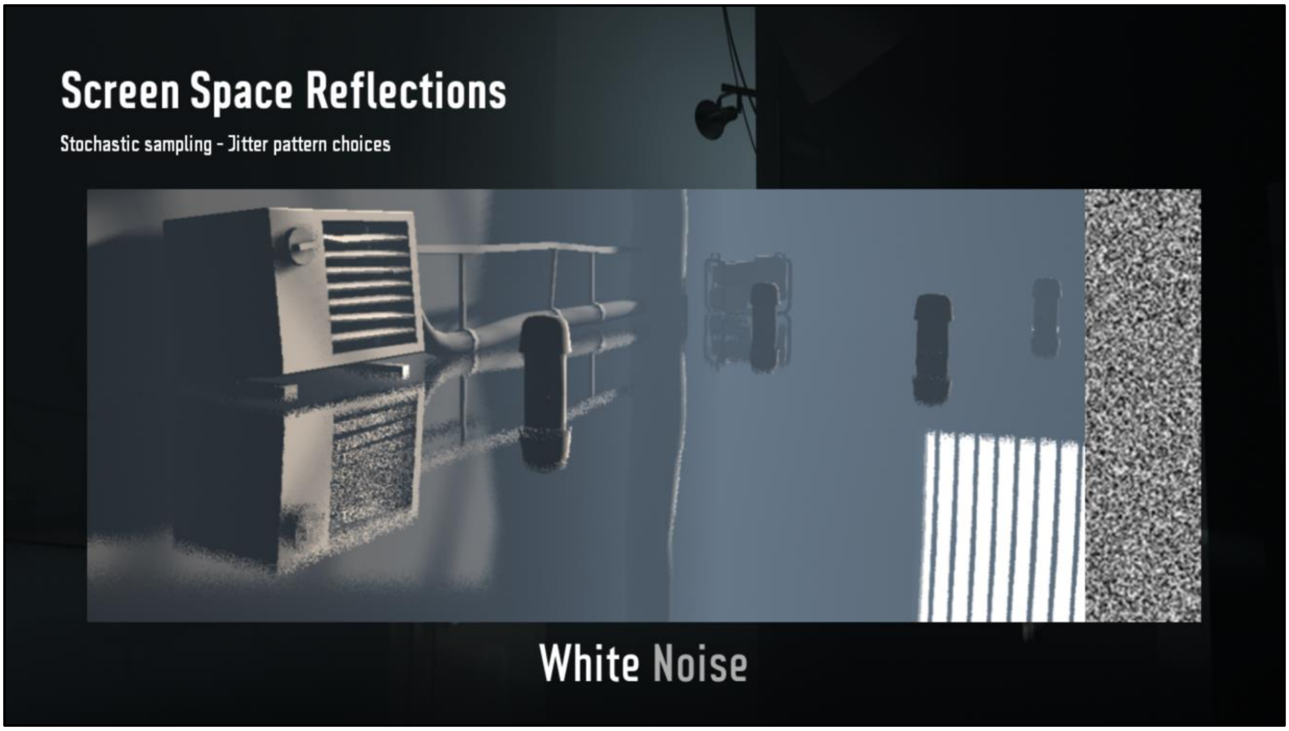
Screen Space Reflections

Stochastic sampling - Trading steps for noise- for blur

```
curPos = vec3(sUv, depthDevice) + sDir;
```

```
curPos = vec3(sUv, depthDevice) + sDir * dither;
```

Now, SSR is a stochastic sampling effect, just like with volume light. So to avoid stepping artifacts, we just need to add some dither or jitter to the first step, giving us an offset at the length of a single step.

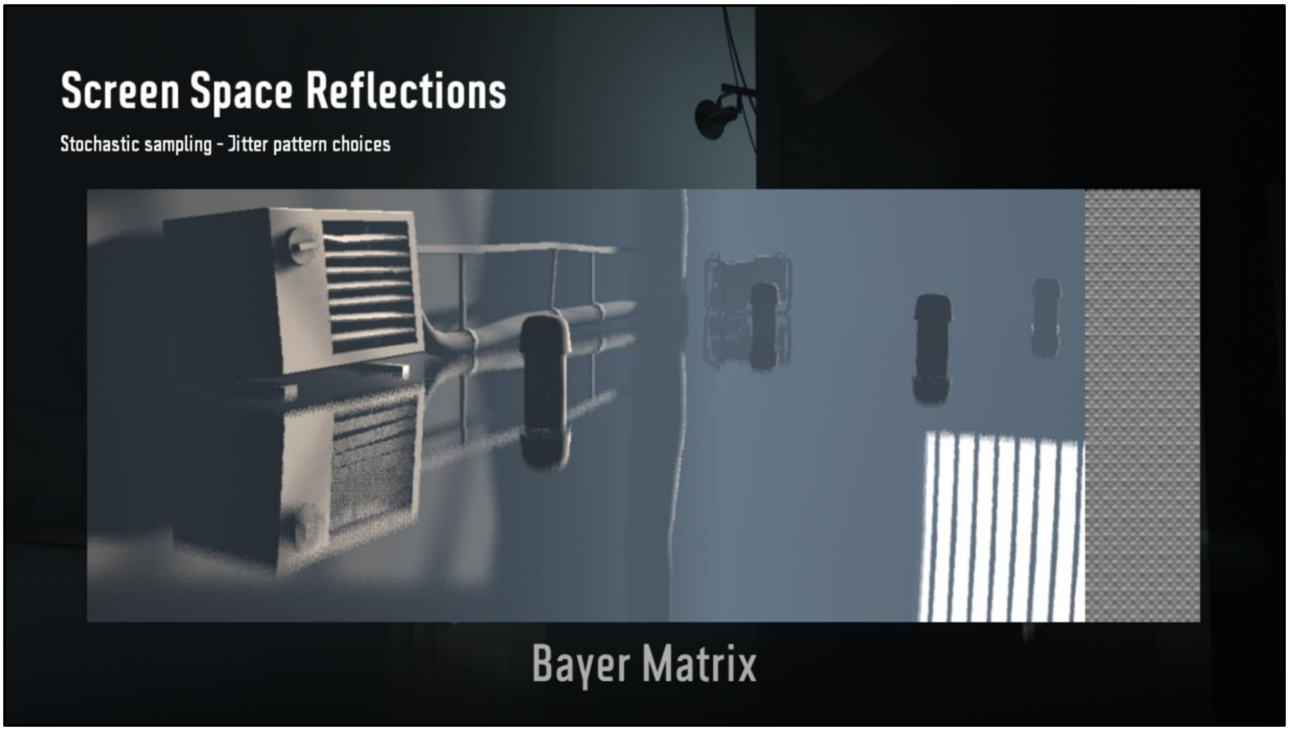


Screen Space Reflections

Stochastic sampling - Jitter pattern choices

White Noise

And just like with any stochastic, picking jitter pattern is important. White noise gets the job done, but has a varying local neighbourhood histogram.

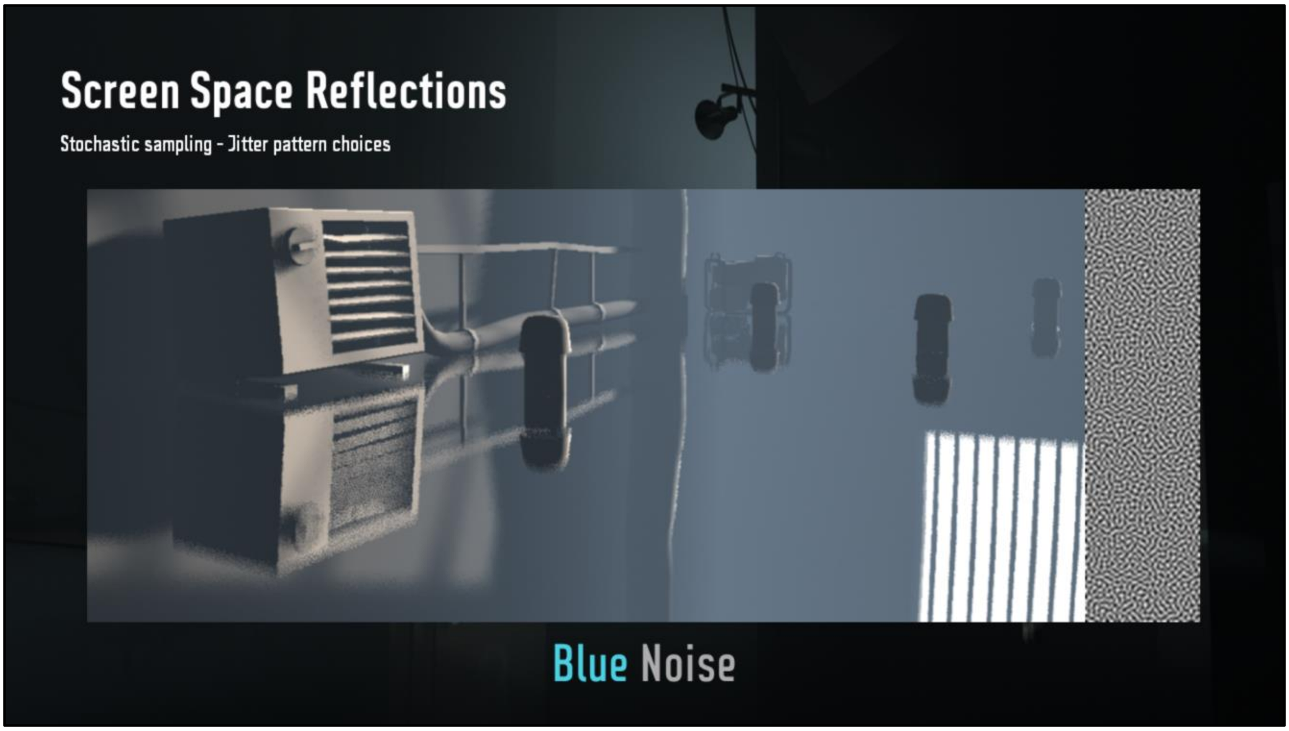


Screen Space Reflections

Stochastic sampling - Jitter pattern choices

Bayer Matrix

Bayer matrix has perfect neighbourhood information, an even histogram for every local region, but too patterny on the eyes.

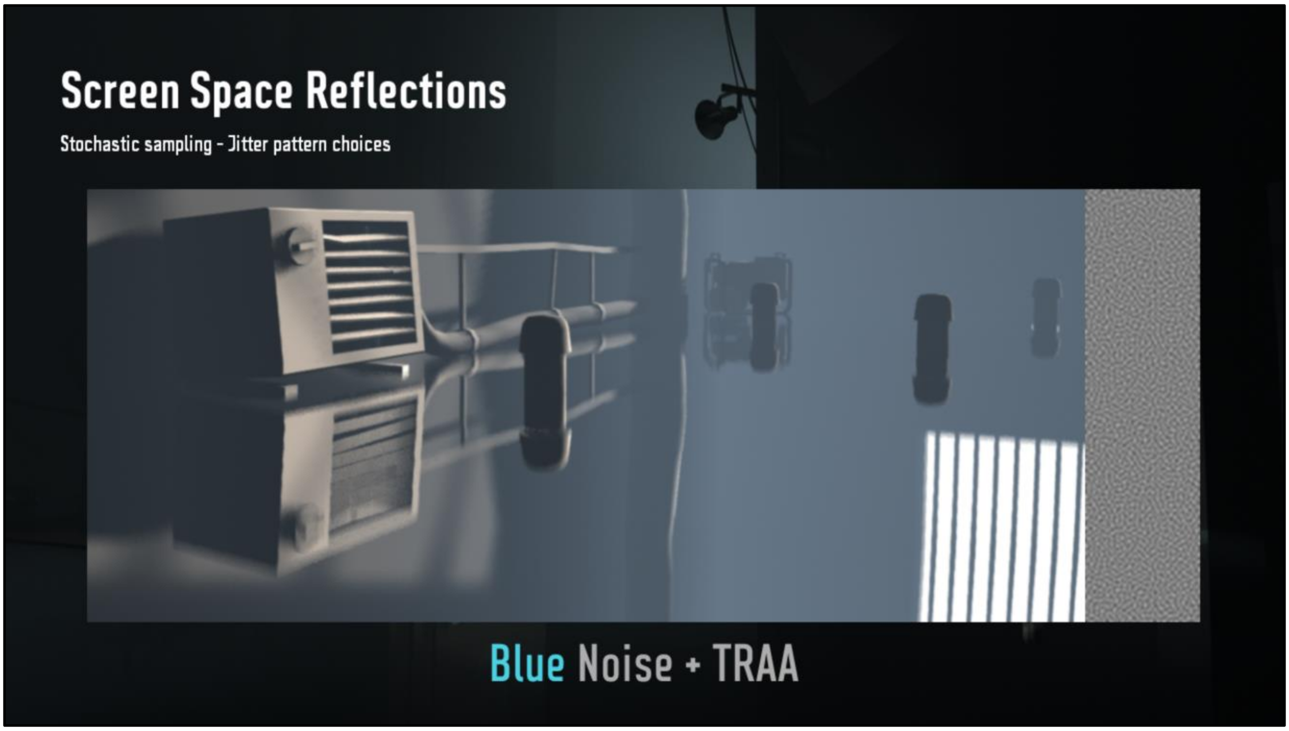


Screen Space Reflections

Stochastic sampling - Jitter pattern choices

Blue Noise

Finally blue noise again has the best of both worlds, a near-perfect neighbourhood histogram, but no visually jarring patterns.

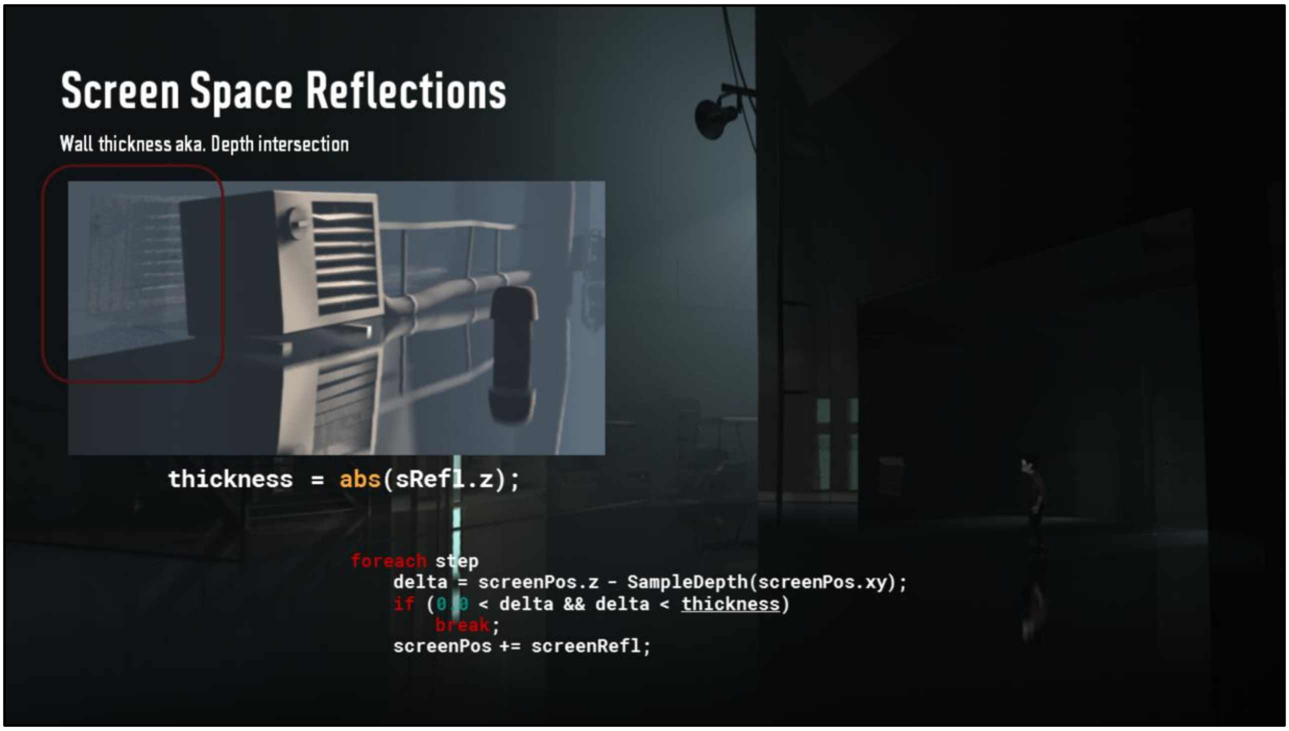


Screen Space Reflections

Stochastic sampling - Jitter pattern choices

Blue Noise + TRAA

Why is it important to have a well covered local neighbourhood? Because It’s used by temporal anti-aliasing. If a pixel has pretty much all possible value within a 3x3 region that TRAA uses, it’ll be less likely clamped or clipped in TRAA, and the history will be used a lot more, giving us temporal upsampling.



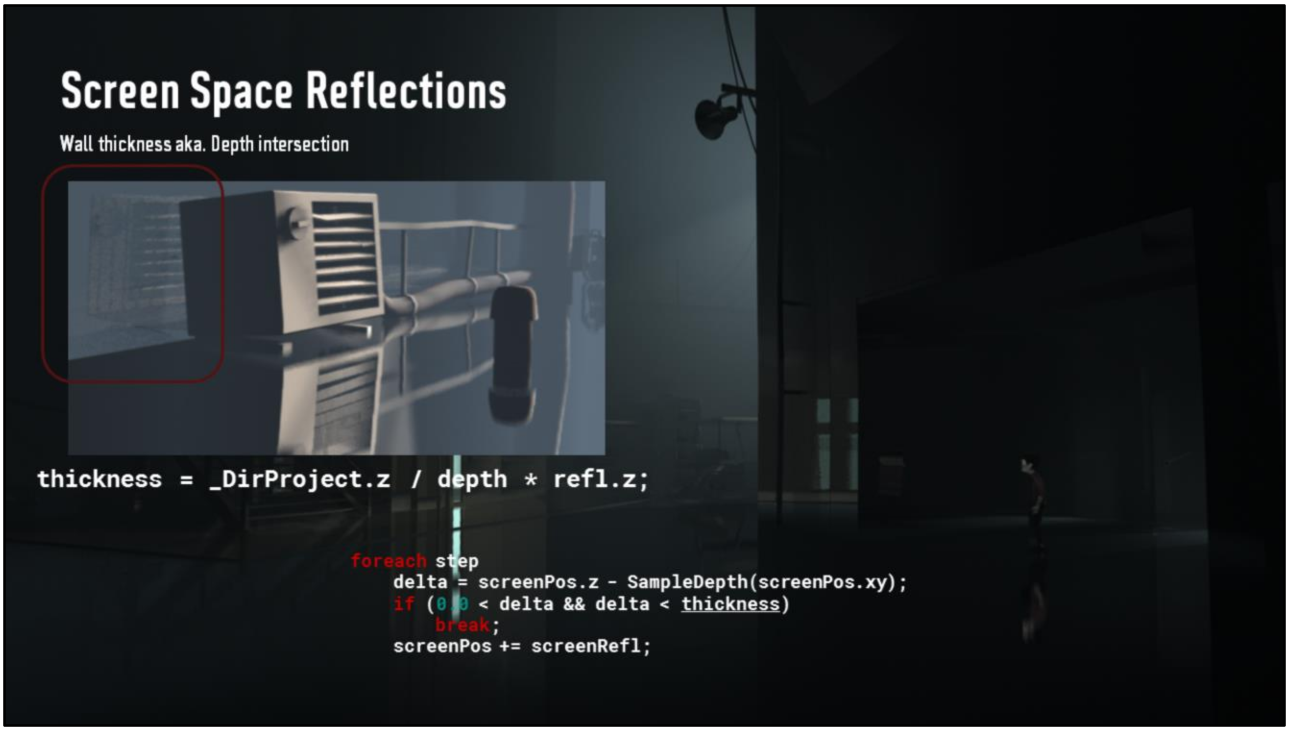
Screen Space Reflections

Wall thickness aka. Depth intersection

```
thickness = abs(sRefl.z);

foreach step
    delta = screenPos.z - SampleDepth(screenPos.xy);
    if (0.0 < delta && delta < thickness)
        break;
    screenPos += screenRefl;
```

Finally, there’s the issue of wall thickness. We lack volume information when ray marching, all we have is our depth buffer which is essentially an empty shell of surfaces, so we need to know how thick we simulate this shell to be.

Our first implementation as a simple constant and a slider for it, but it was hard to use and the results were unpleasant.
Second round involved using the delta of the screen reflection ray’s Z, so if we’ve come into a wall between last step and this, we’re happy.
But this had problems when looking at walls that are 45 degrees to the viewer generating reflection rays that are perpendicular to the view direction. In that case, we have a 0 Z delta, and only move in X and Y.



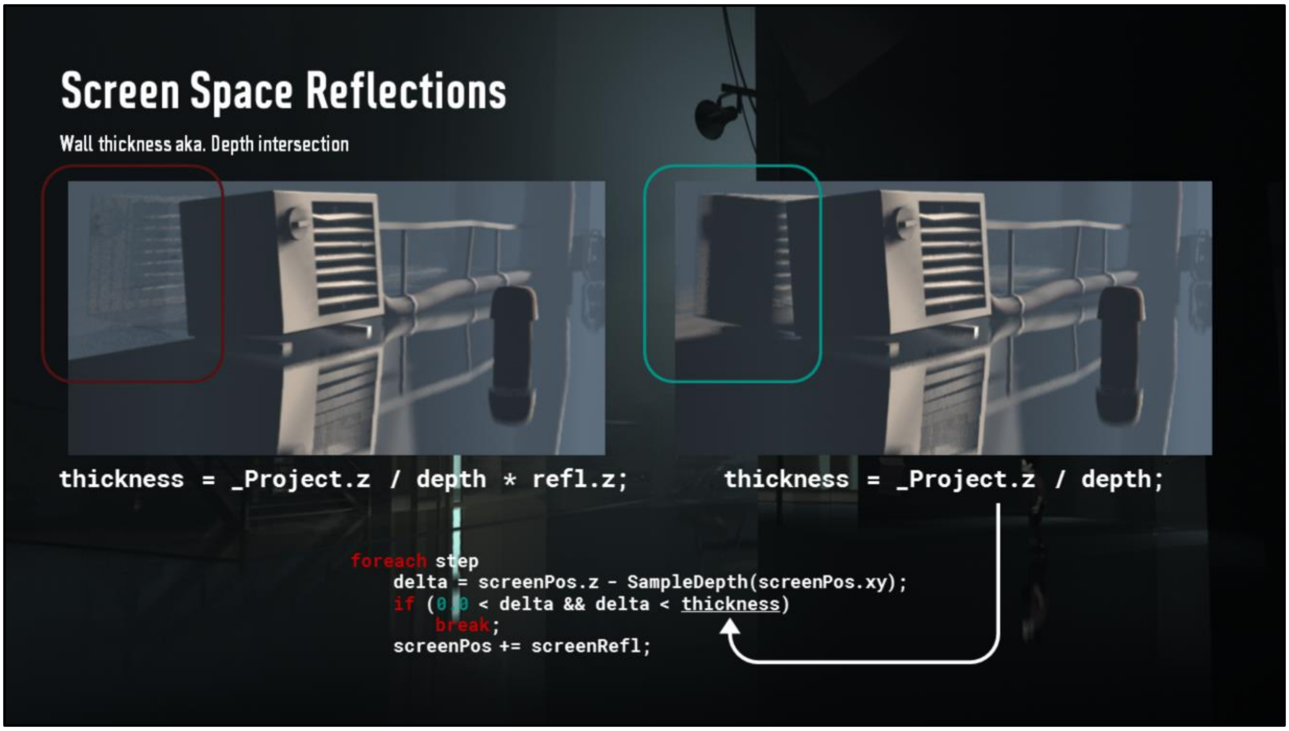
Screen Space Reflections

Wall thickness aka. Depth intersection

```
thickness = _DirProject.z / depth * refl.z;
```

```
foreach step
    delta = screenPos.z - SampleDepth(screenPos.xy);
    if (0.0 < delta && delta < thickness)
        break;
    screenPos += screenRefl;
```

To solve this, we can try to unwrap what was actually done in that abs(sRefl.z) term, and here, it turns out the sinner is simply the reflection direction itself!



Screen Space Reflections

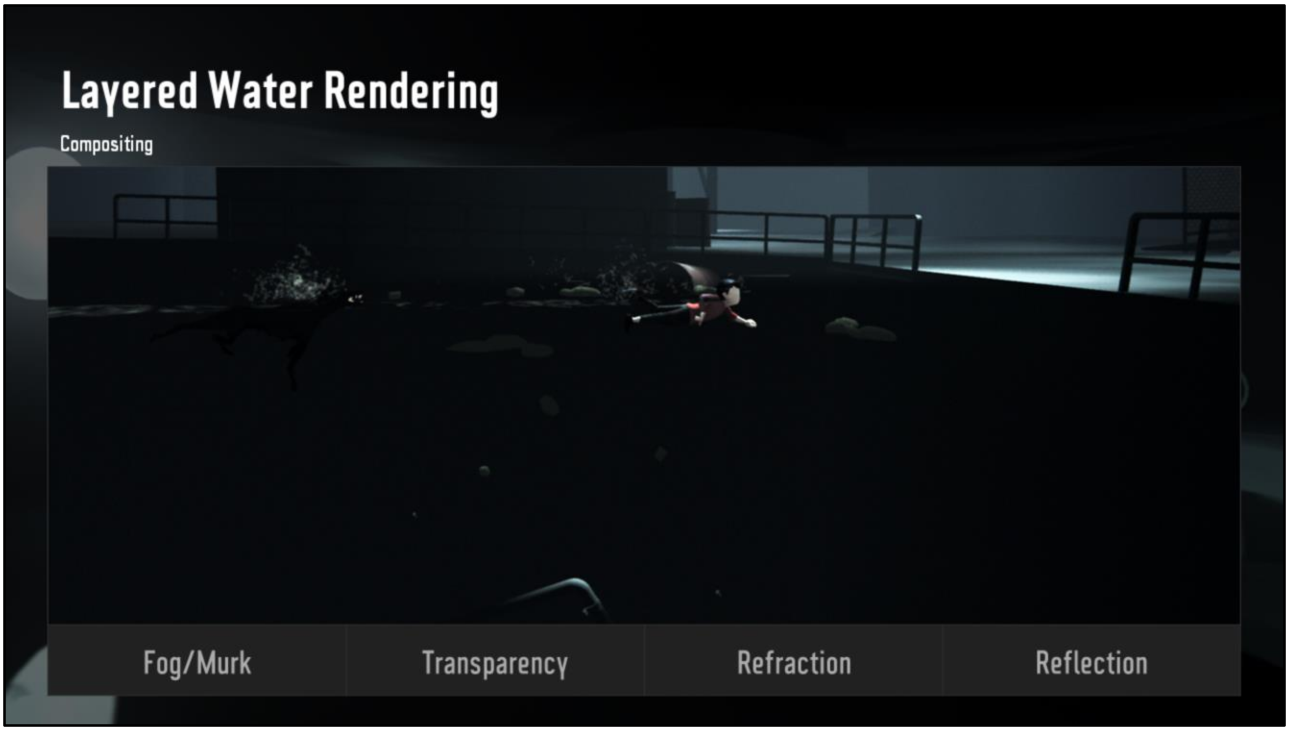
Wall thickness aka. Depth intersection

```
thickness = _Project.z / depth * refl.z;
```

```
thickness = _Project.z / depth;
```

```
foreach step
    delta = screenPos.z - SampleDepth(screenPos.xy);
    if (0.0 < delta && delta < thickness)
        break;
    screenPos += screenRefl;
```

So if we remove it and simply use this instead, we remove our edge case, and are almost guaranteed to hit something as long as it’s within view!



Layered Water Rendering

Compositing

Fog/Murk

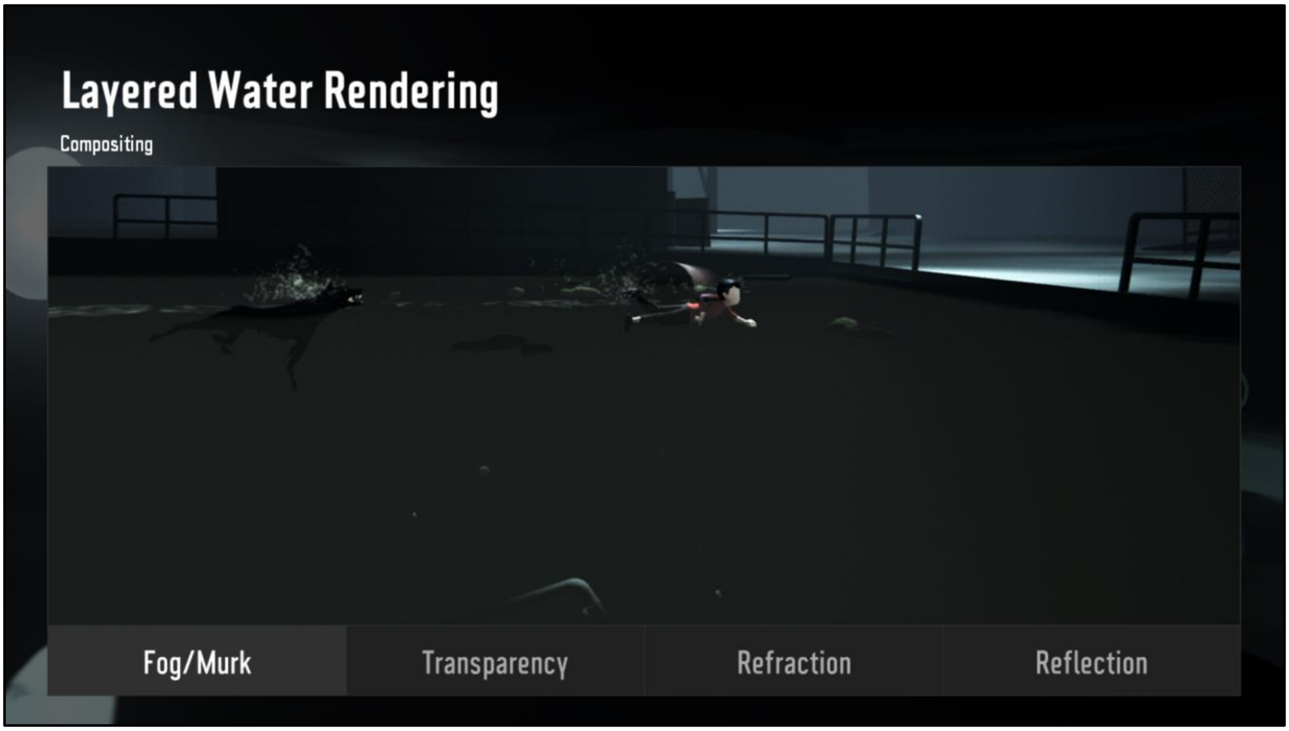
Transparency

Refraction

Reflection

Another place we use reflections is water, but in this case, we don’t utilize SSR, for something that fills the screen as much as water, we’d get too many out of bounds issues. So we are using planar oblique clipping camera reflections.

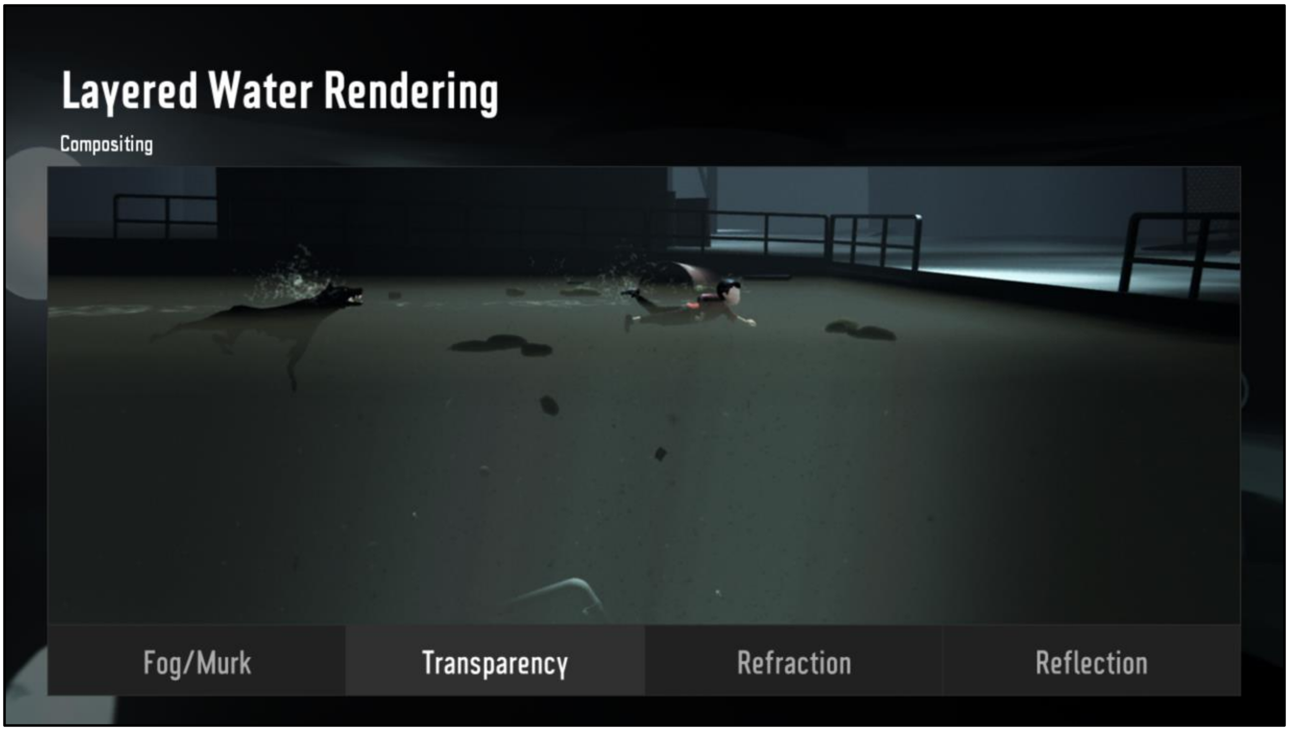
Now when we think of water visually, we abstract it as layers, first a layer of fogginess, murkiness or dirt, then refraction and finally reflection.
It turned out to be useful to render it out in practice as we think about it in abstract, so first…



Layered Water Rendering

Compositing

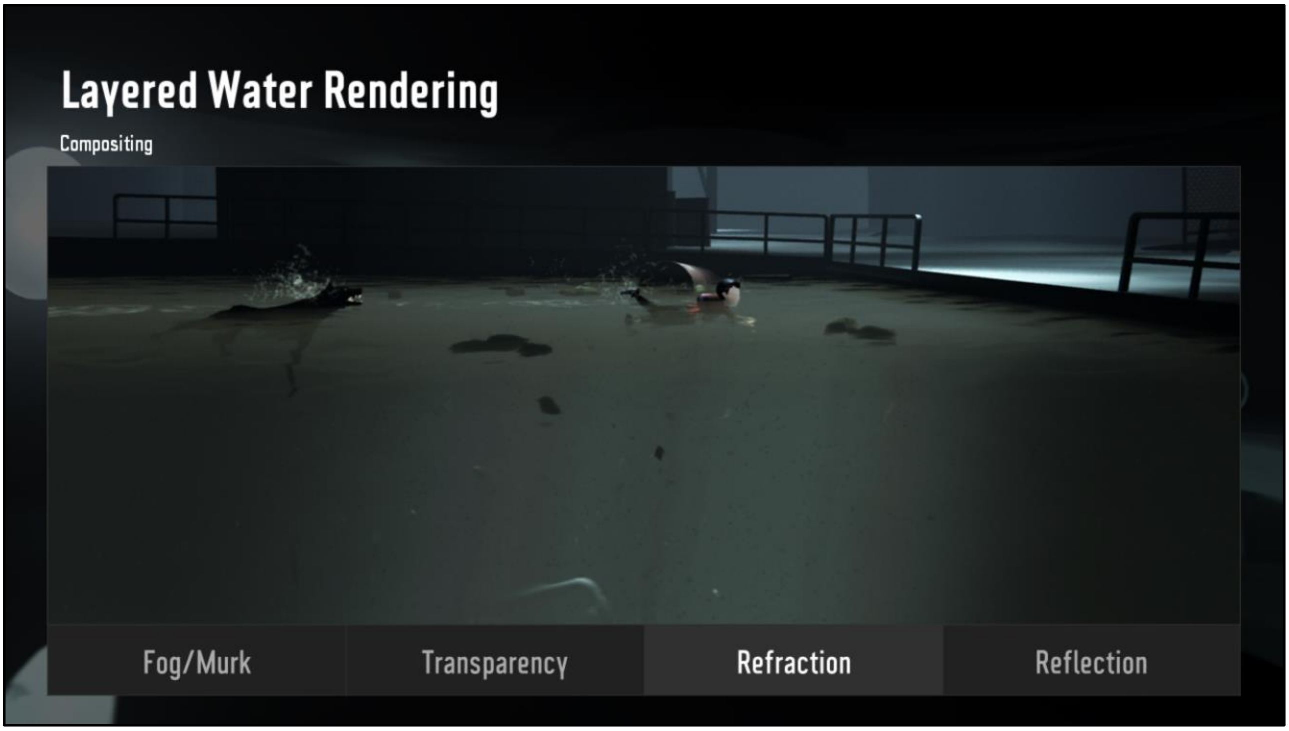
We render the fog, this is from surface to background geometry, we just have a color and a distance aka how fast it fades in.



Layered Water Rendering

Compositing

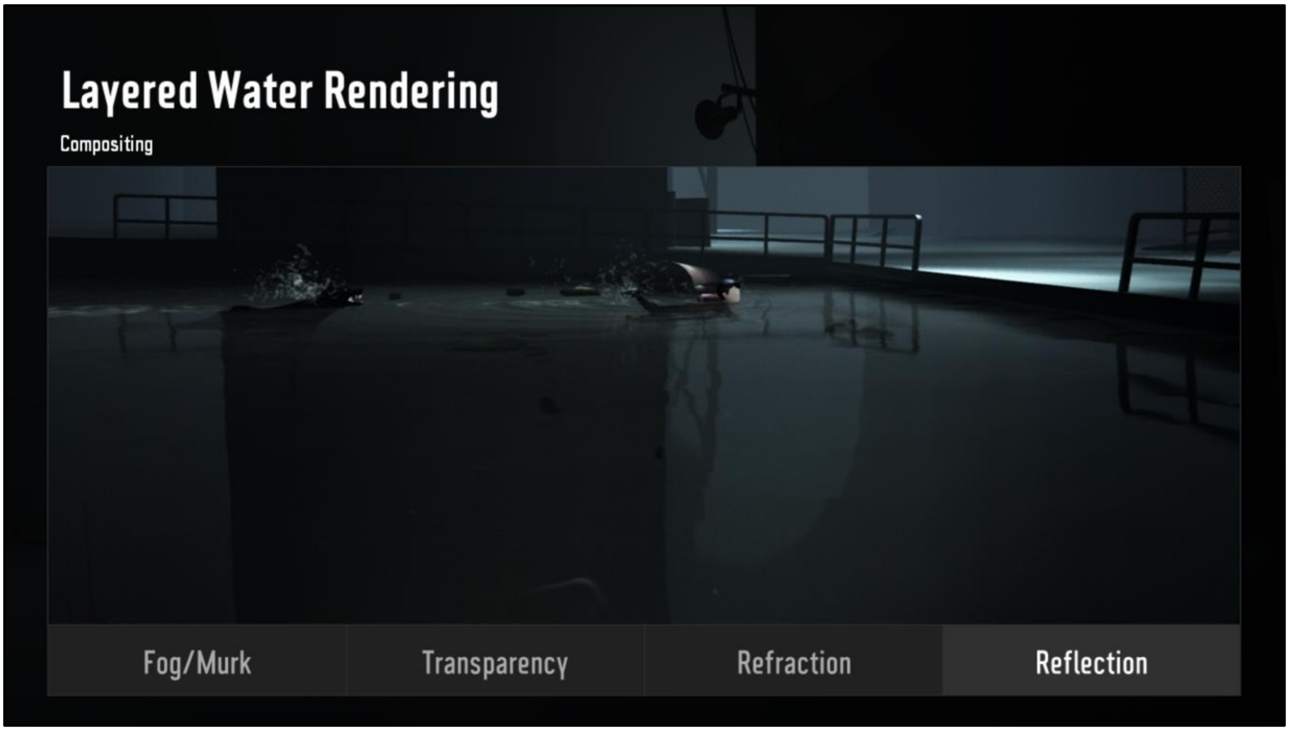
Then we render any transparent objects that have been tagged to be under water, we do this because soon a screen shot will be taken, so that refraction can distort it and put it back, and for that we need all that we want visible in that picture to be rendered out now.



Layered Water Rendering

Compositing

So after that we render the refraction layer. It uses a single stochastic sample along the normal of the surface using blue noise to generate the distortion.

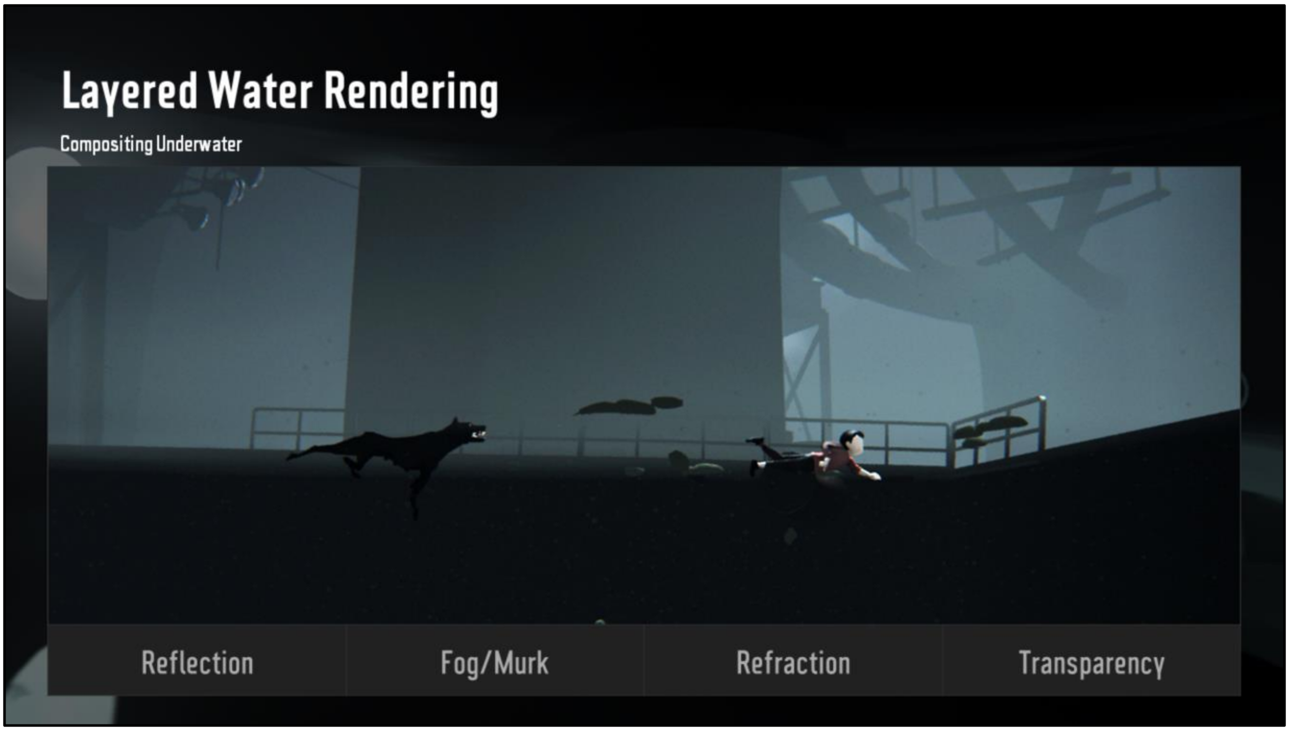


Layered Water Rendering

Compositing

Fog/Murk | Transparency | Refraction | **Reflection**

And lastly reflection, using Fresnel to control the opacity of the surface, and distortion from normal to add motion.

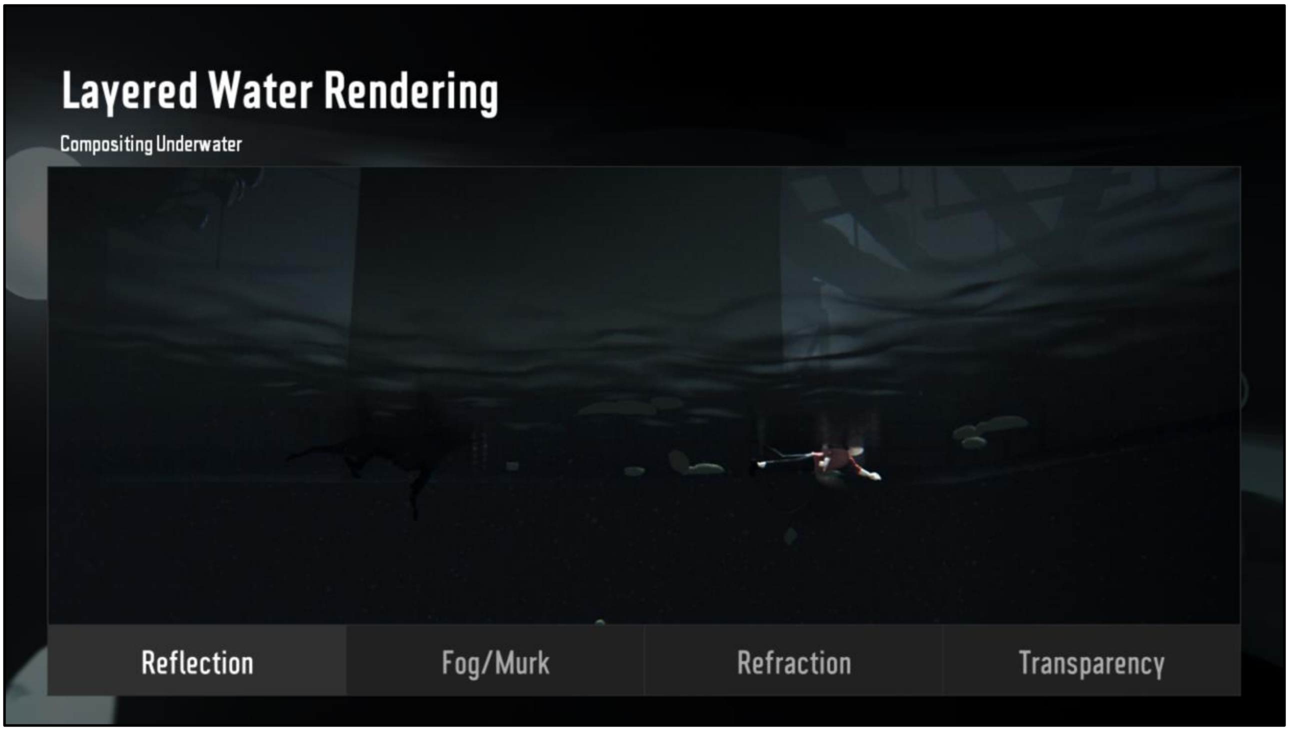


Layered Water Rendering

Compositing Underwater

Reflection | Fog/Murk | Refraction | Transparency

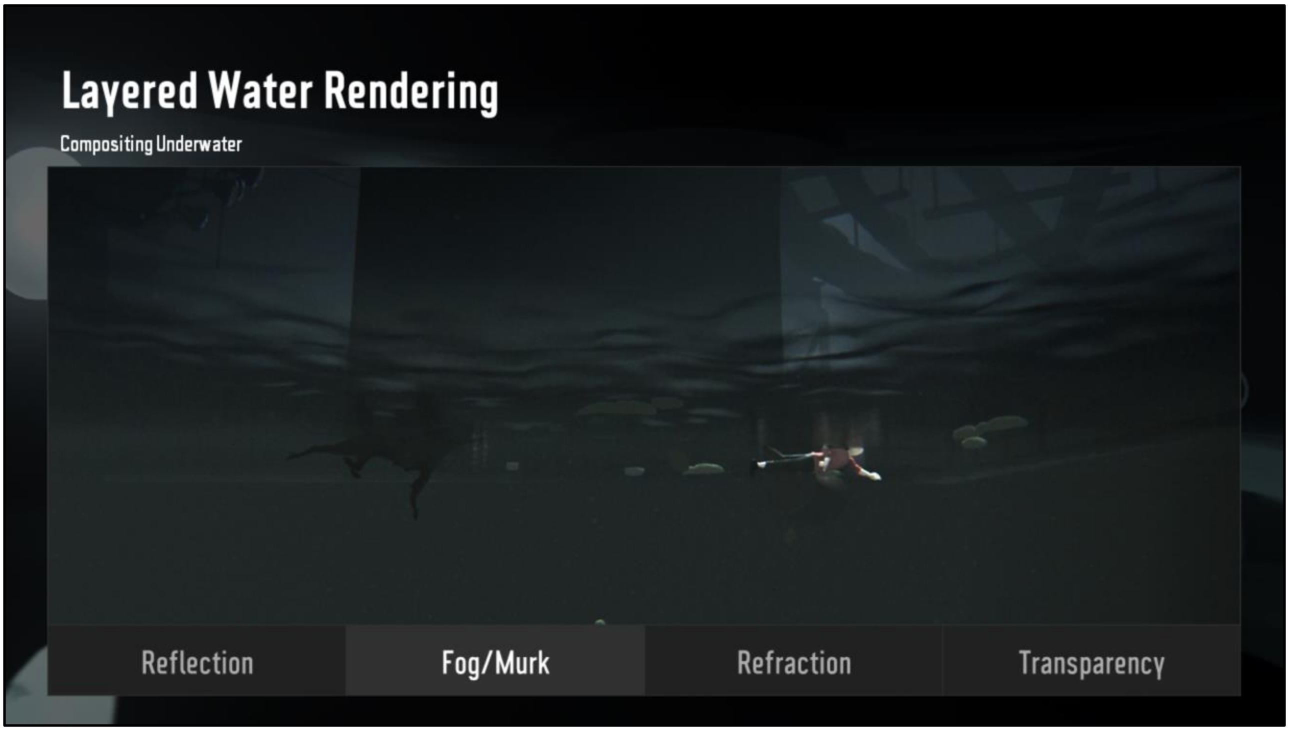
Now, in case the camera is under water, the order of the objects have been shuffled since the order we see things in is now different, and the shading is a bit different as well.



Layered Water Rendering

Compositing Underwater

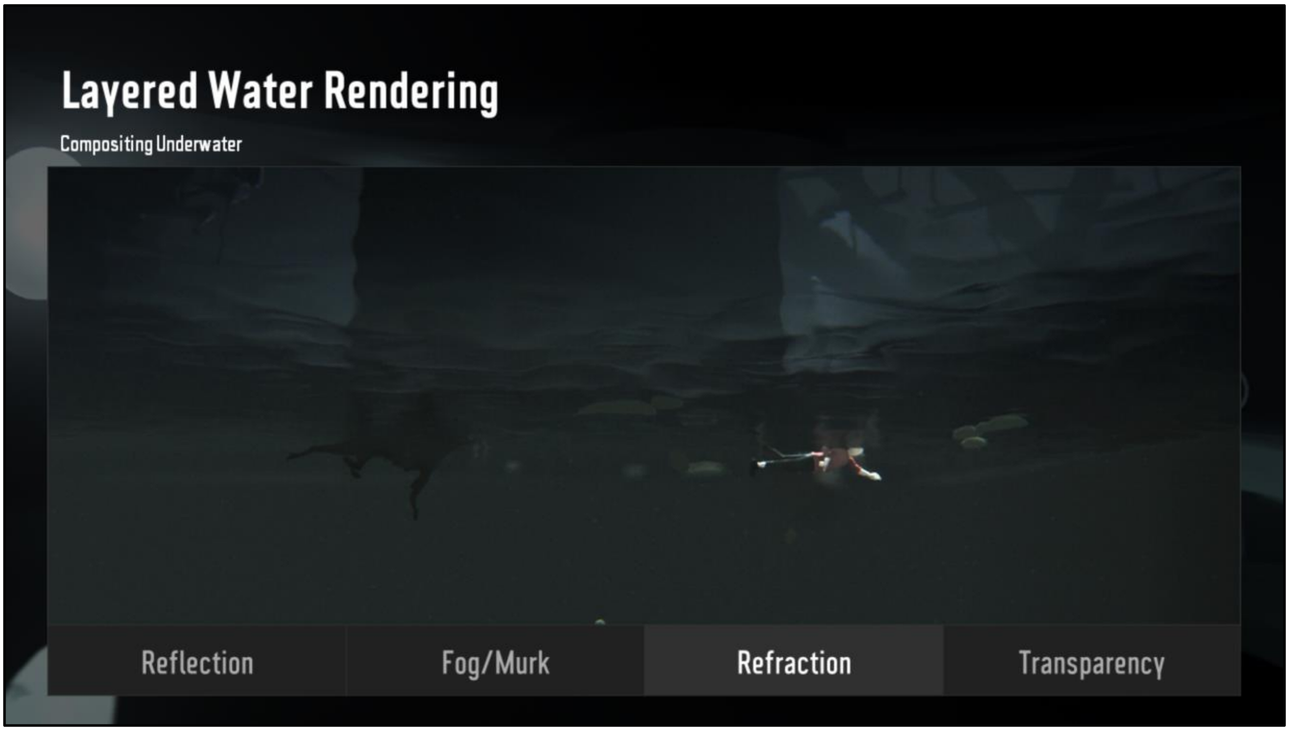
This time we start with the reflection, and what’s different is that instead of a regular power Fresnel, we use a smoothstep between two very close numbers to get a quick transition to total reflection and back.



Layered Water Rendering

Compositing Underwater

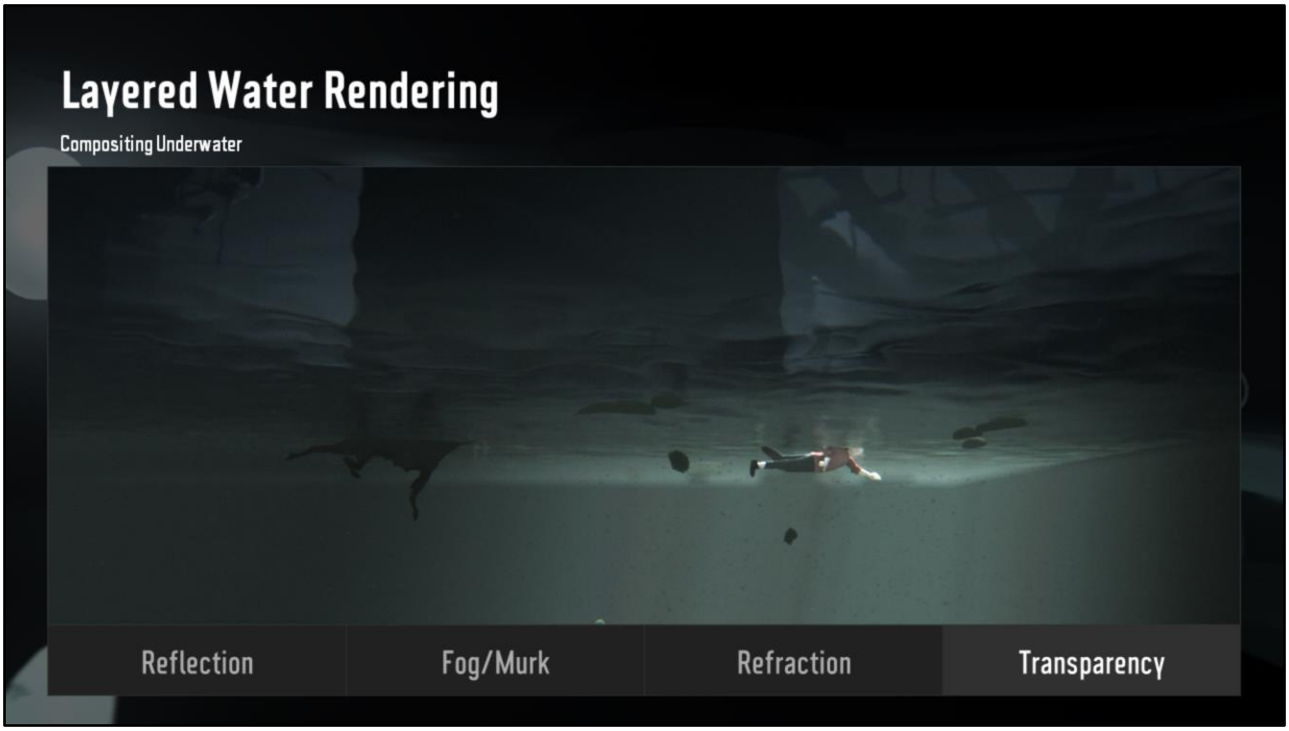
Then we layer fog on that, this time from view to the surface or scene depth, whatever comes first.



Layered Water Rendering

Compositing Underwater

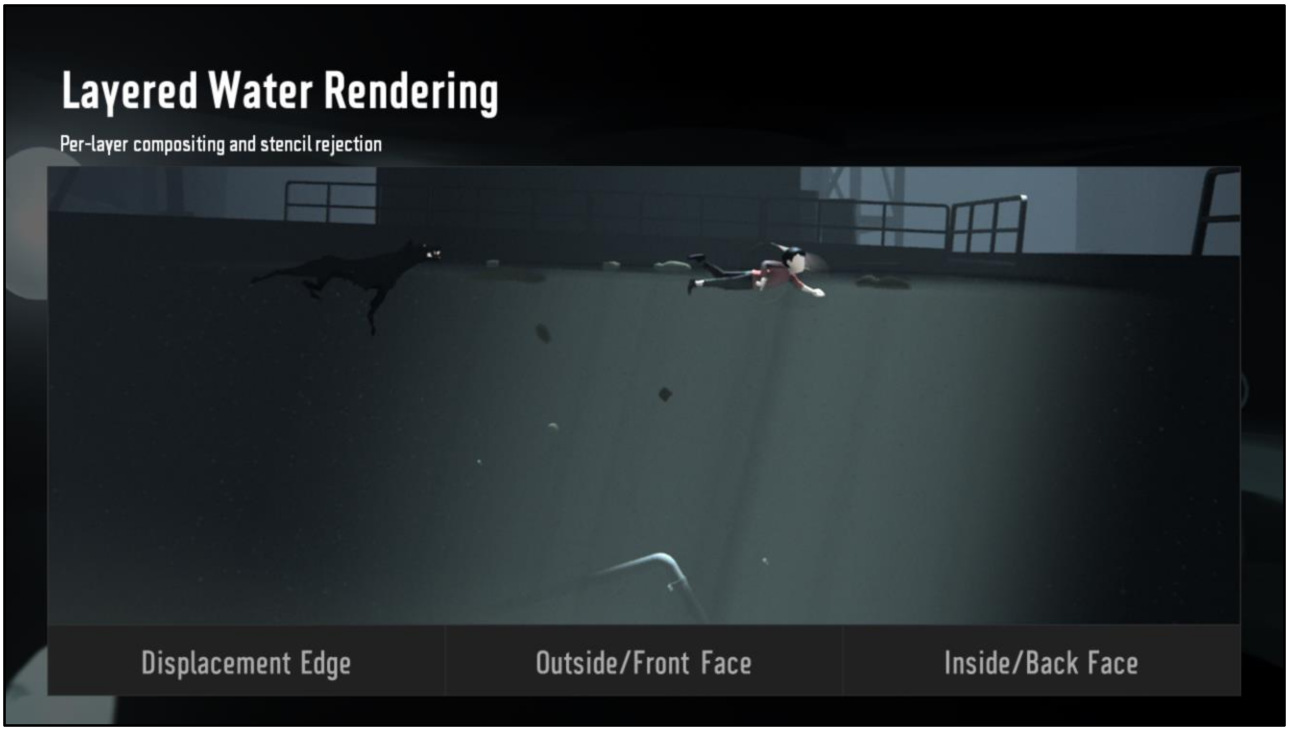
Now we refract, and we add fog before refraction because this time we also add depth of field to things behind the gameplay plane and we want that fog to blend well with that.



Layered Water Rendering

Compositing Underwater

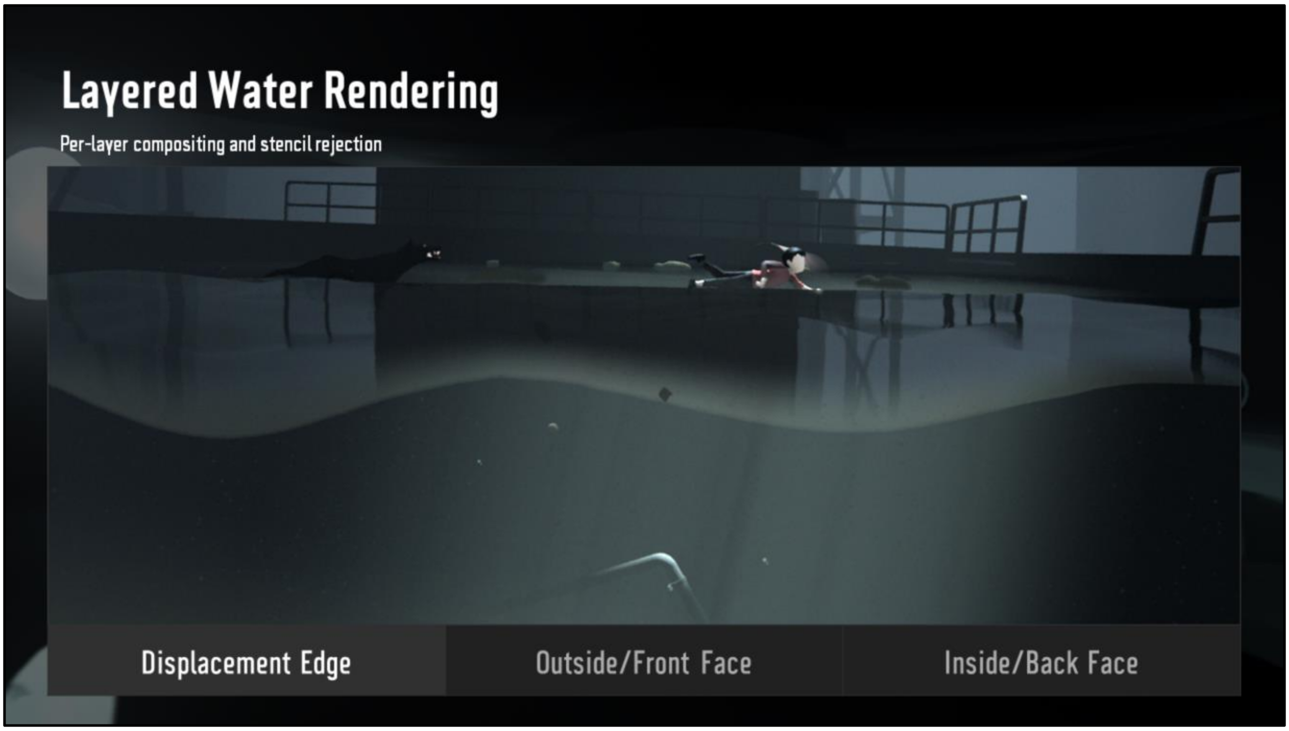
Finally the transparencies tagged as under water, cause we don’t wanna blur or distort that. The transparencies that are not tagged were naturally rendered before any water was



Layered Water Rendering

Per-layer compositing and stencil rejection

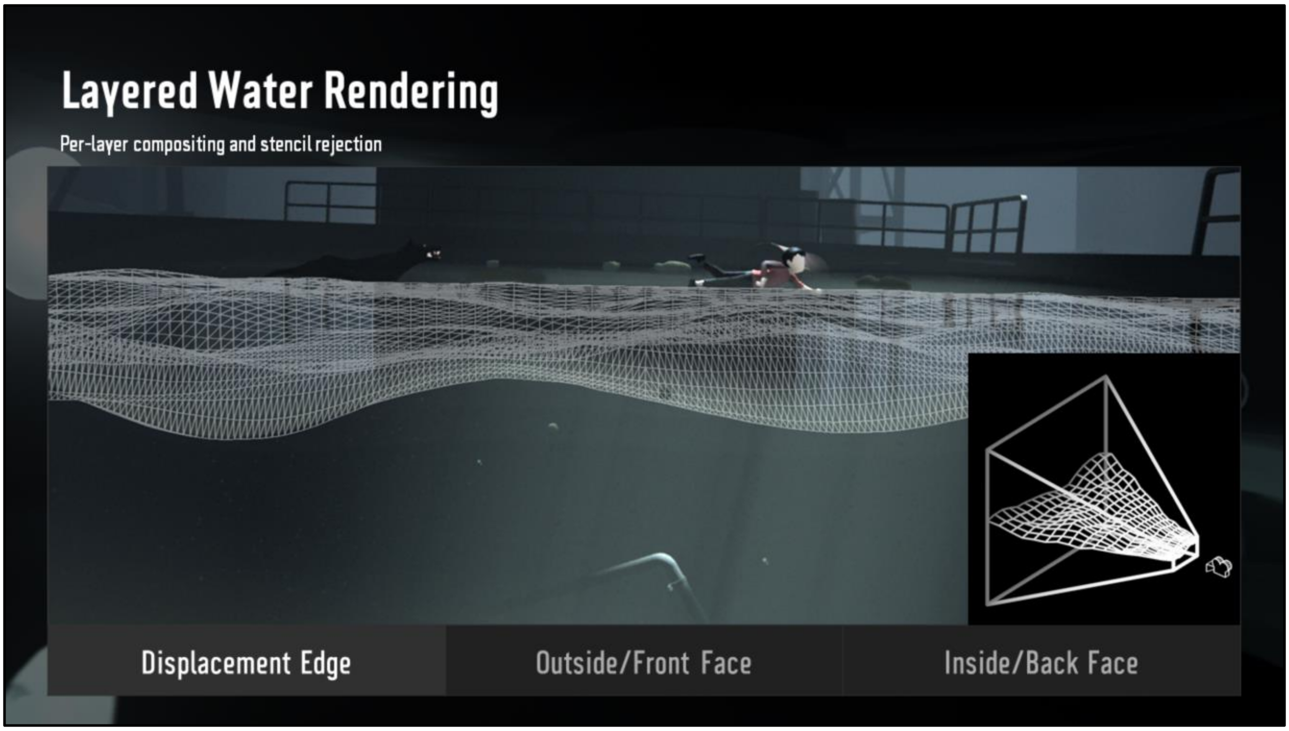
Finally, for each layer we render out 3 surfaces to make the volume. We don’t have any pops or clips when passing through, the water is supposed to take care of the transition using shading alone.



Layered Water Rendering

Per-layer compositing and stencil rejection

We start with the closest, a displacement edge that spans anywhere from 6-12 meters from the camera near clip plane into the water.
This is needed otherwise we get a paper thin surface as we transition in and out of the water. The limited distance is just because you hardly notice the parallax or depth after a certain point.

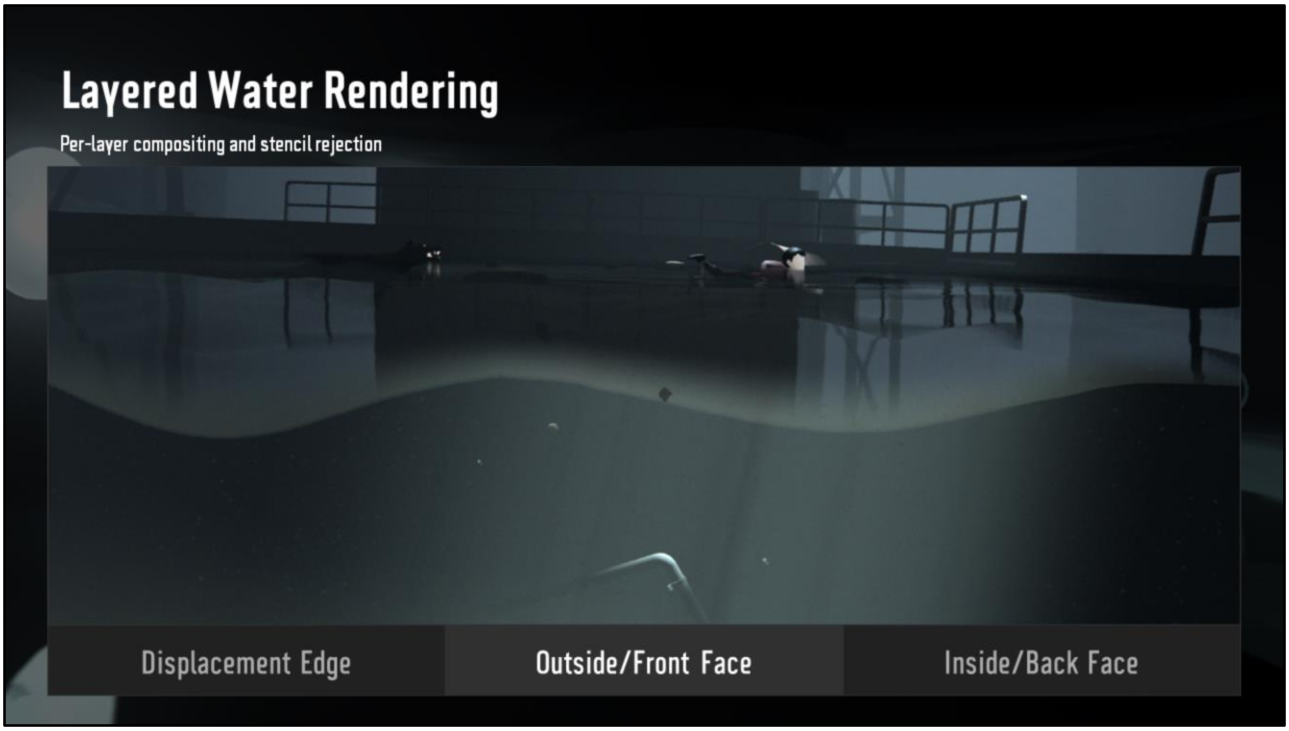


Layered Water Rendering

Per-layer compositing and stencil rejection

Displacement Edge | Outside/Front Face | Inside/Back Face

It's stuck to the camera and vertices are distributed linearly in screen space rather than view space, for best distribution.



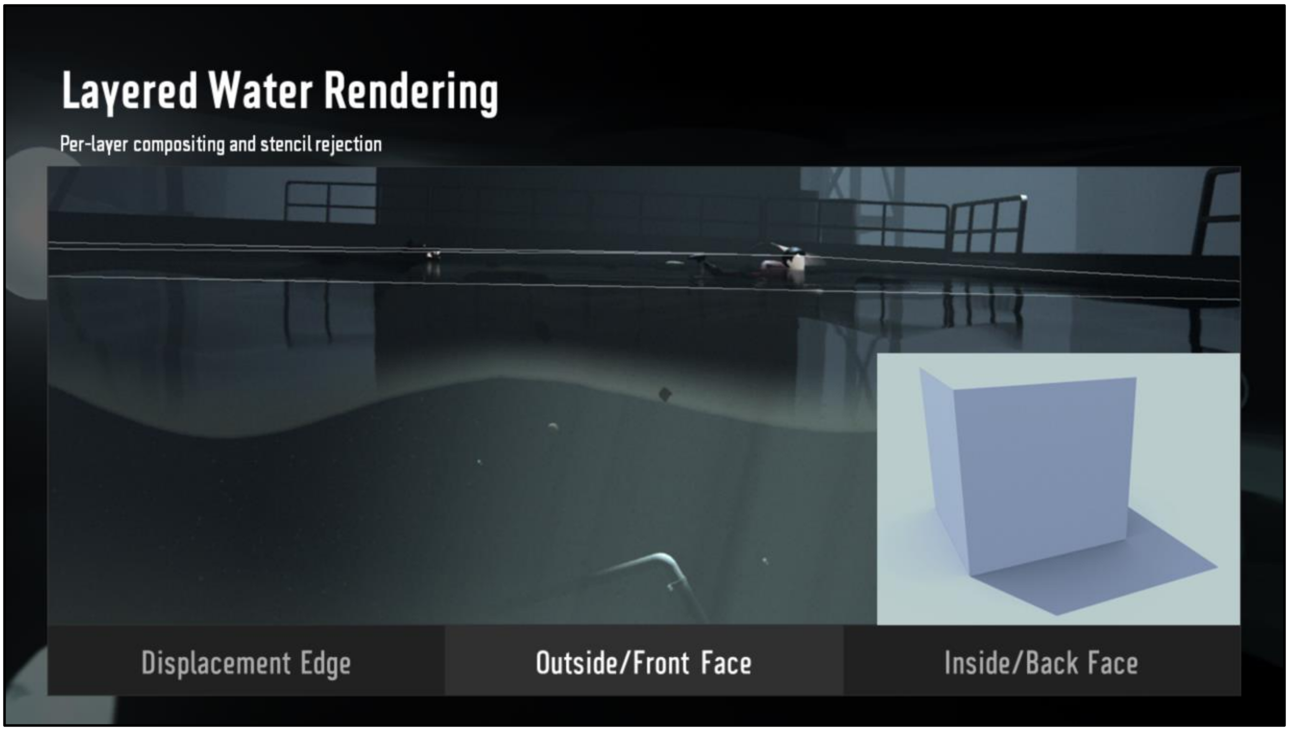
Layered Water Rendering

Per-layer compositing and stencil rejection

Displacement Edge | Outside/Front Face | Inside/Back Face

Secondly we render the outter sides of the volume, it takes care of whatever the displacement edge didn’t.

It doesn’t draw on top of it, though. We render them in order of visibility front to back, and use stencil rejection to make sure only the first surface is visible.



Layered Water Rendering

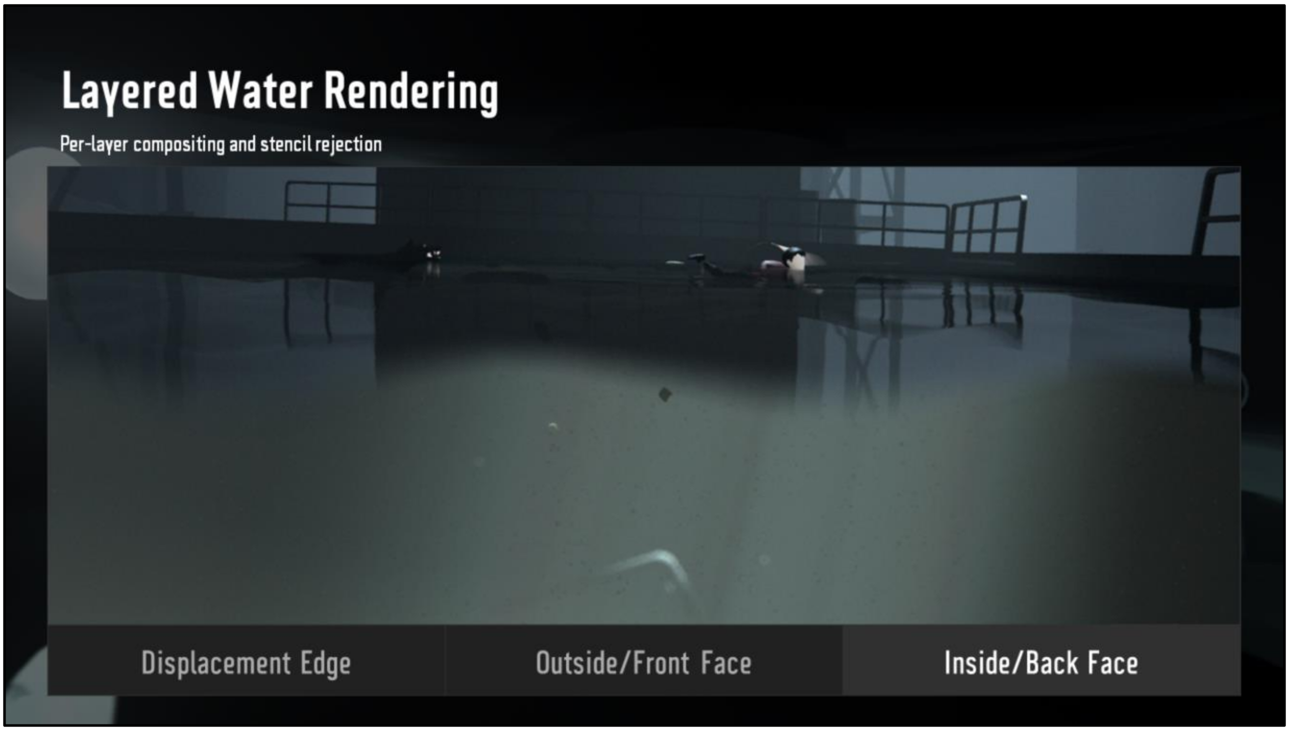
Per-layer compositing and stencil rejection

Displacement Edge

Outside/Front Face

Inside/Back Face

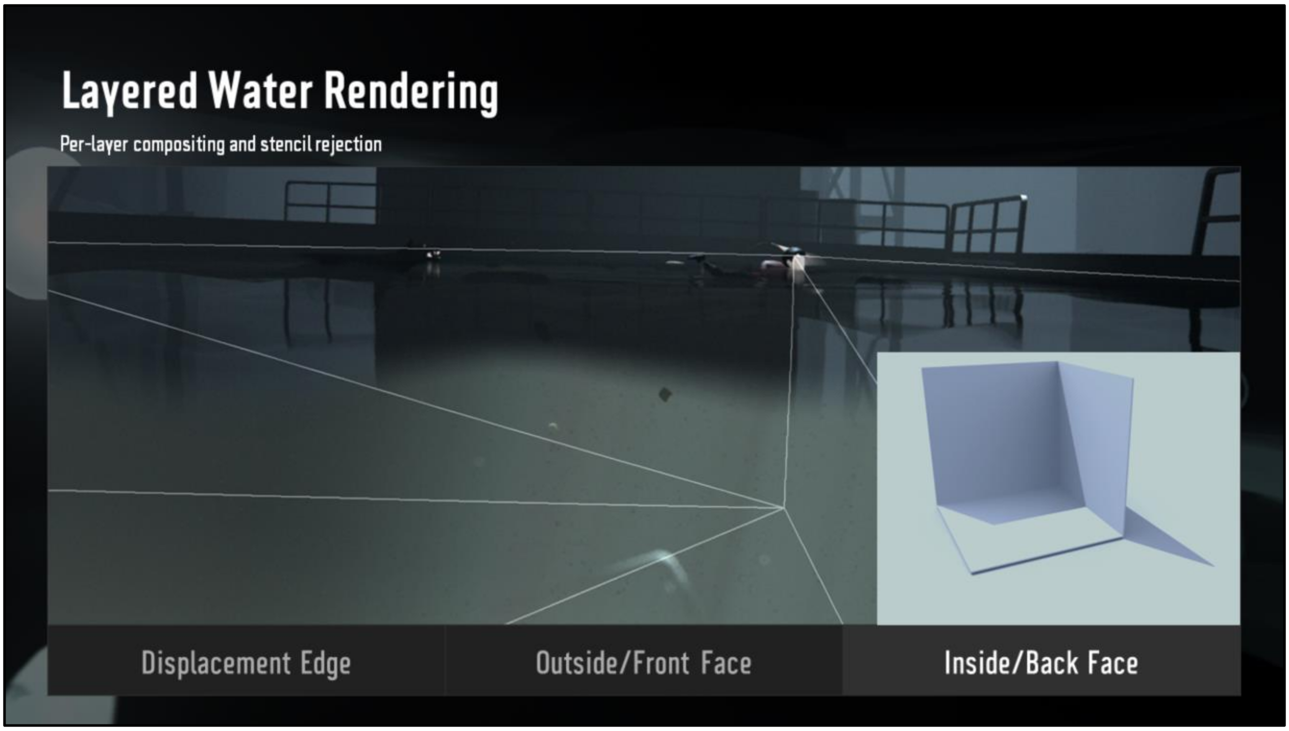
This is what is meant by outter sides, in this case a box. It’s simply the front faces.



Layered Water Rendering

Per-layer compositing and stencil rejection

Finally we render the inner sides, these use that underwater variant of the shaders, they skip on the ZTesting and also reject on stencil so the outter sides are preserved.



Layered Water Rendering

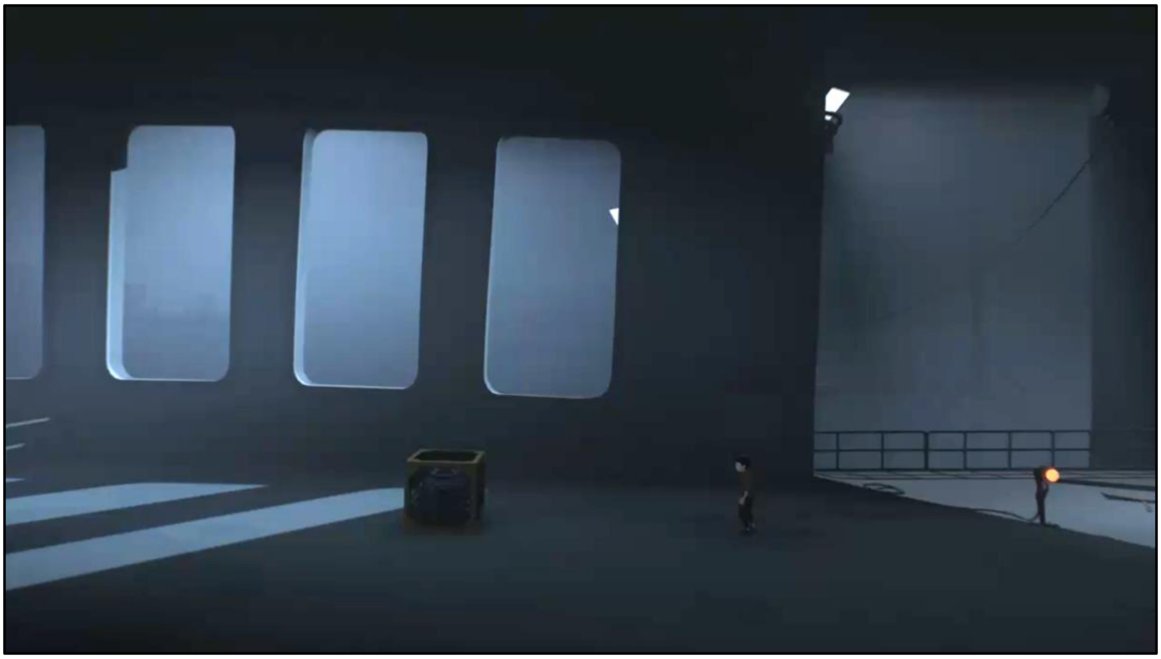
Per-layer compositing and stencil rejection

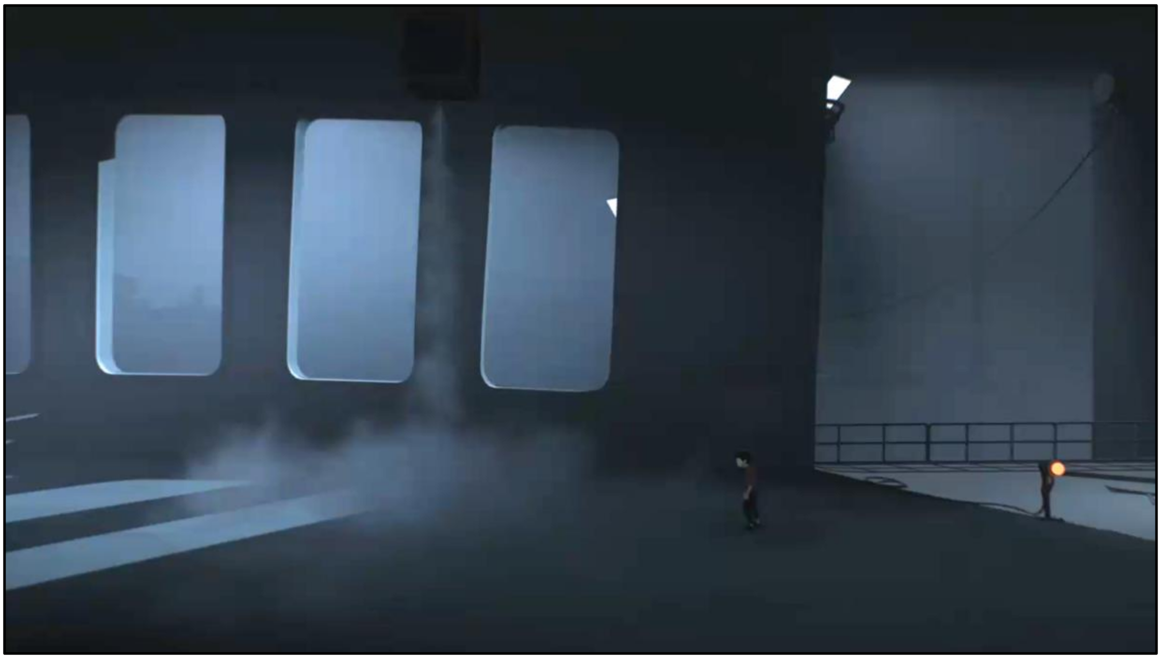
Displacement Edge

Outside/Front Face

Inside/Back Face

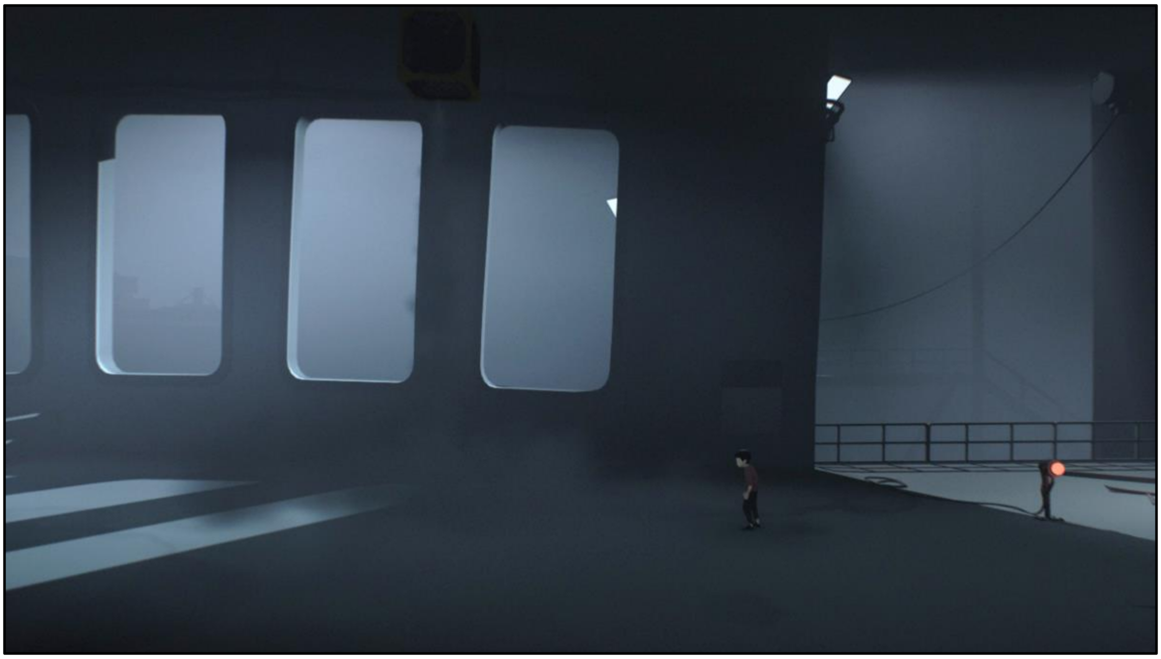
SMOKE



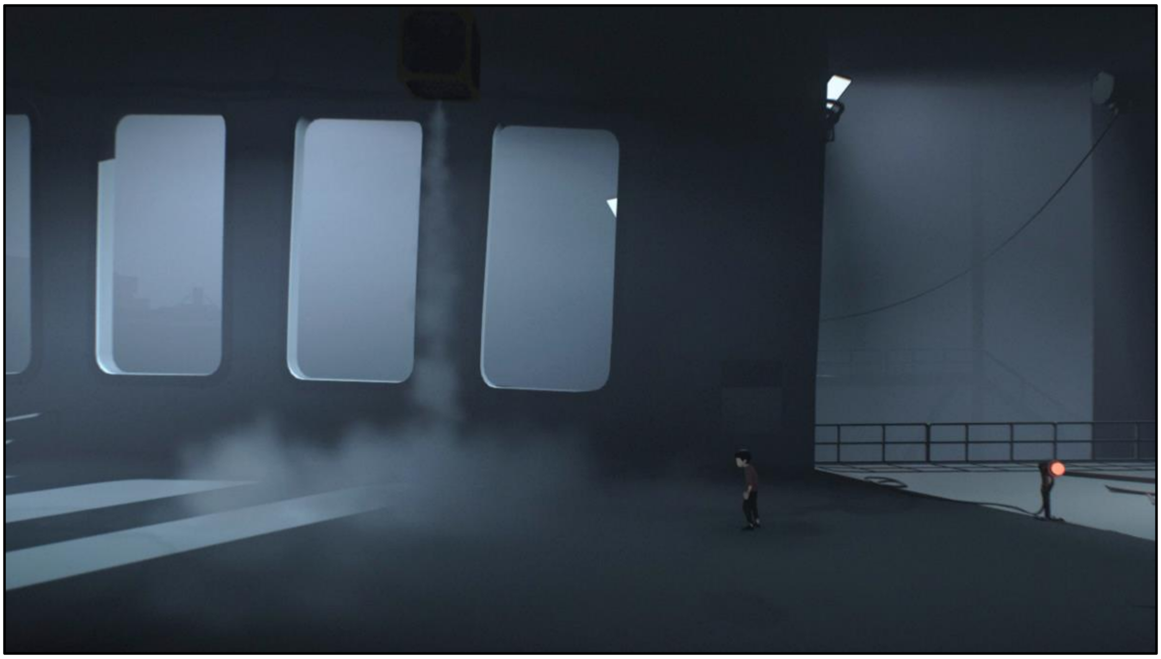


Now let's jump out and dive into some VFX, starting with smoke.

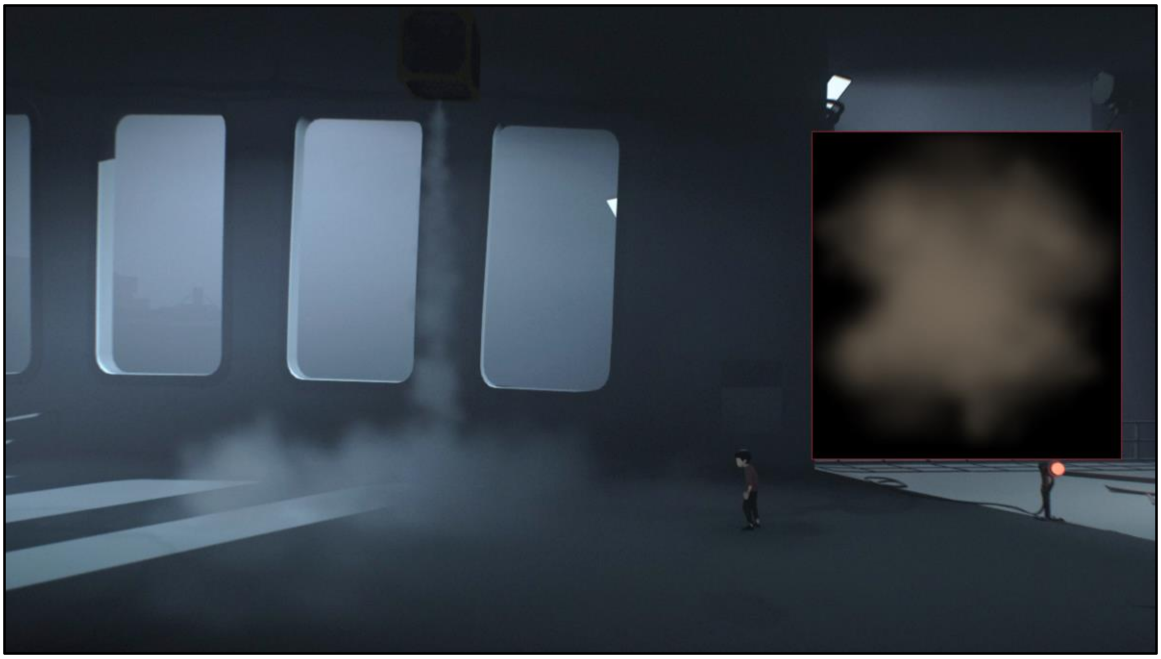
So you'll notice that even when paused, there's motion in the smoke column.
There's also some gradiated lighting from the ground and more.
If we disable them they look like…



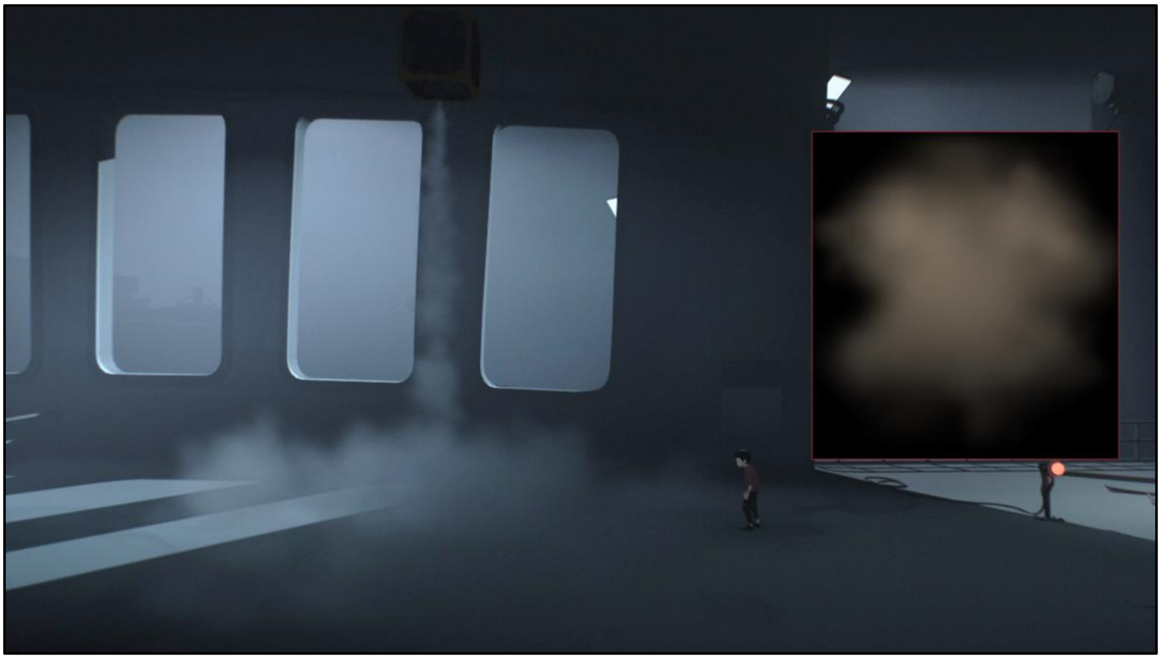
This! It’s hard to see now since the color is the same ambient color of the room. So now let’s put them back one by one.



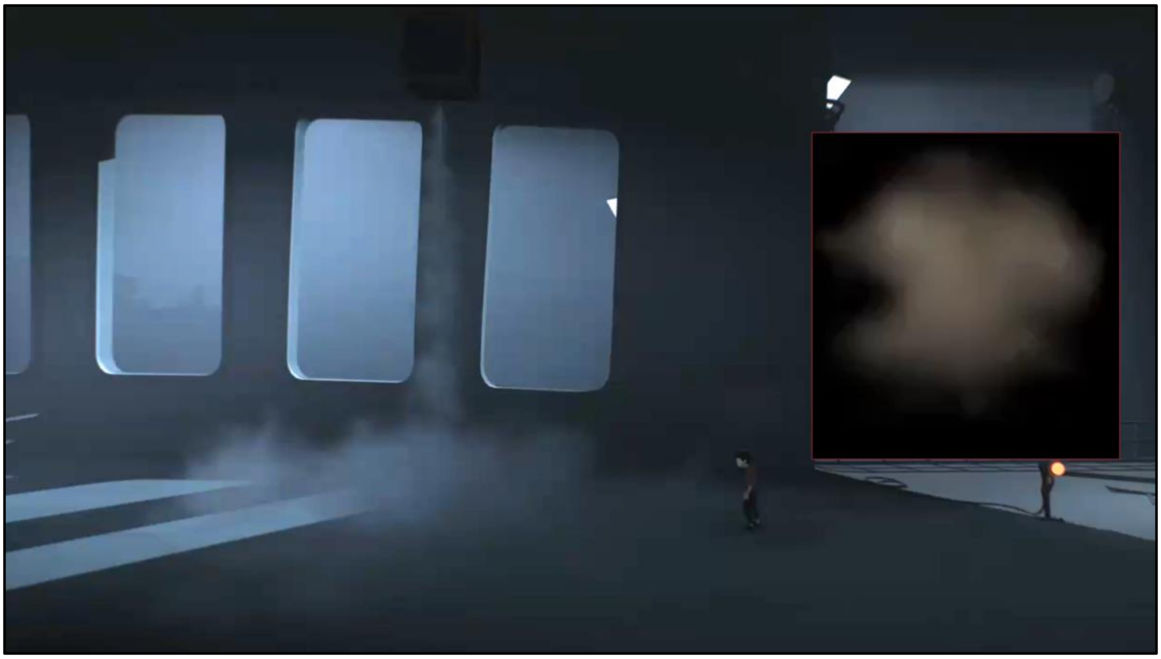
First the light from the ground, simulating GI. We just set up point lights specifically for our particles



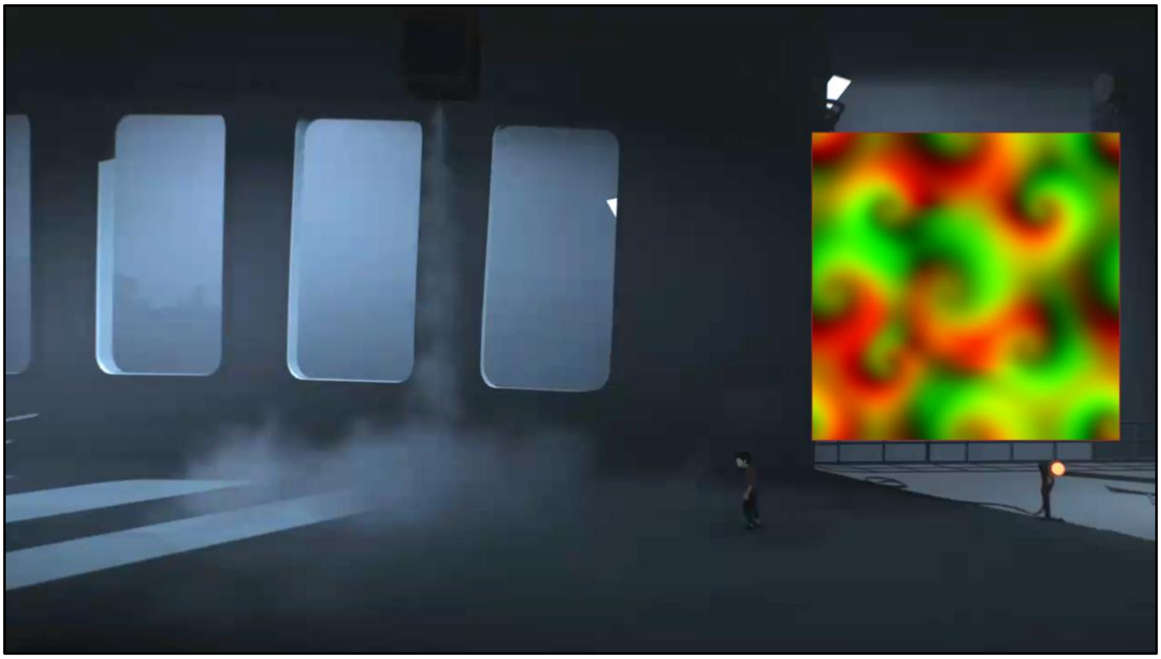
Up next we’re gonna add a subtle vertical gradient, we want the top to be lighter than the bottom to simulated occlusion. So let’s enable that…



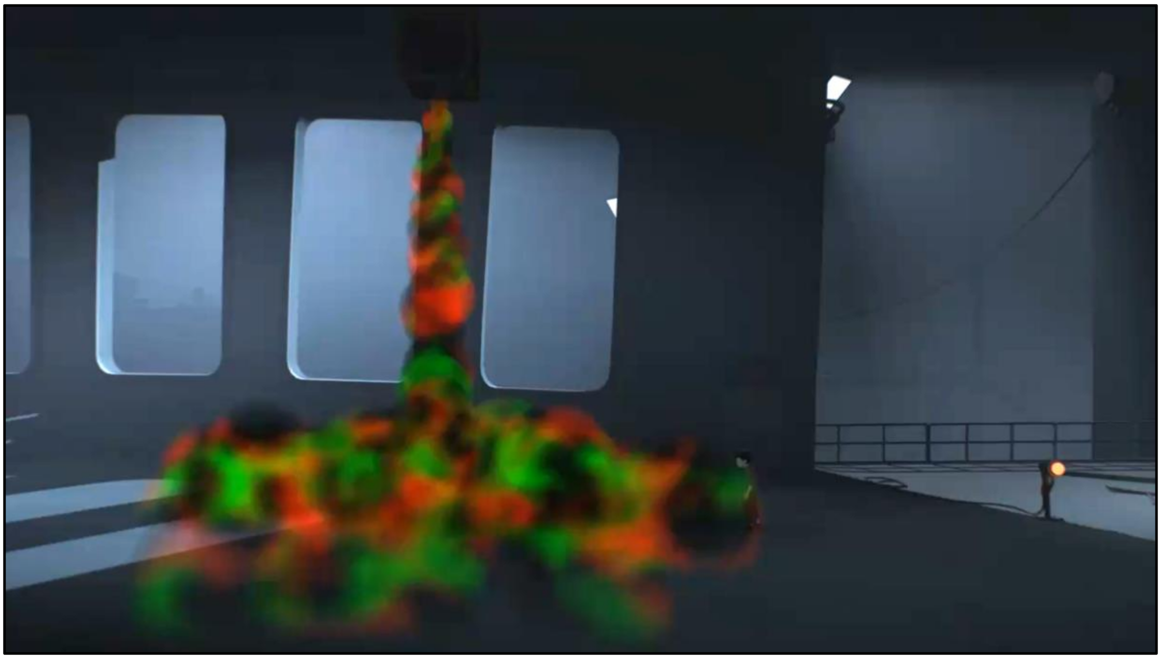
Now. So this just helps ground it a little bit. Finally, we add…



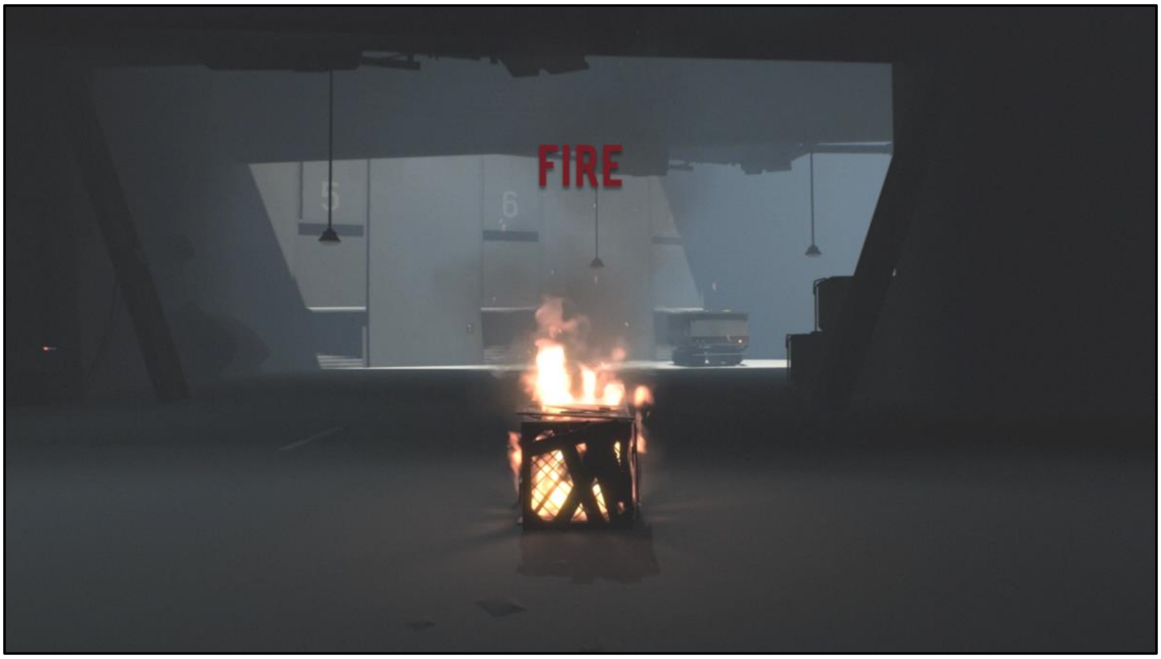
Distortion back! And now we’re back to where we started.



The map we use for diction is this swirl noise as we call it. It’s made by using the twirl filter on a UV gradient in photoshop and pasting it around a bit in a tillable fashion.

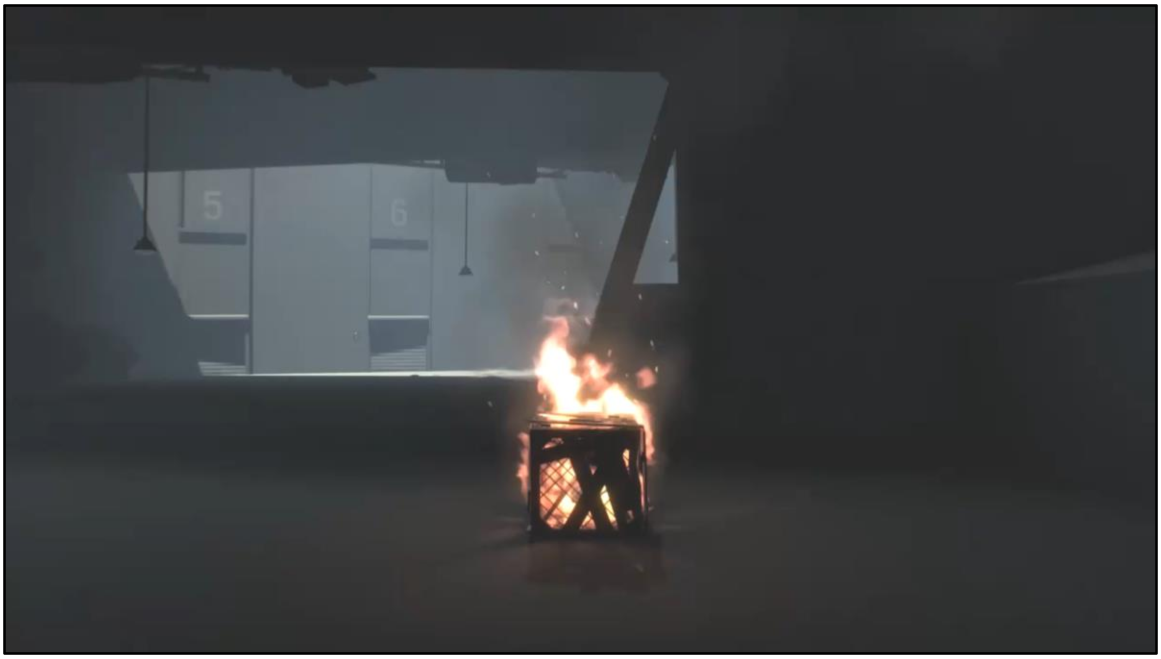


We project this in worldspace onto the particles, with a random offset on each, and scroll them in a single direction.
This time it’s down since that’s the direction it came out of the box.

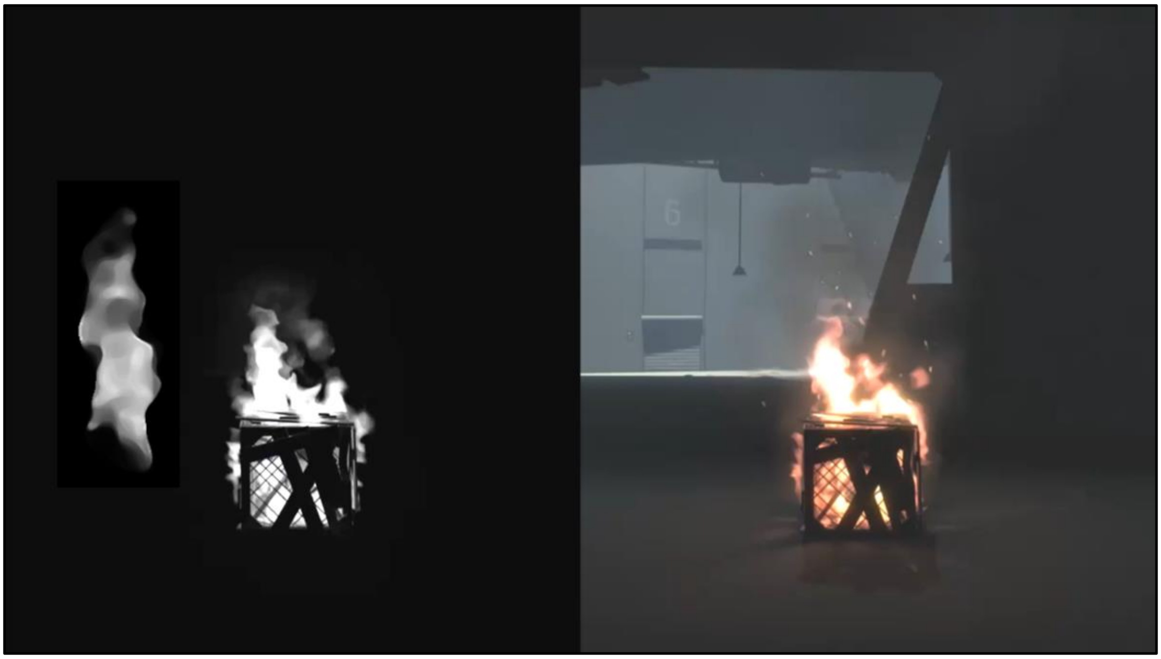


Now, fire.

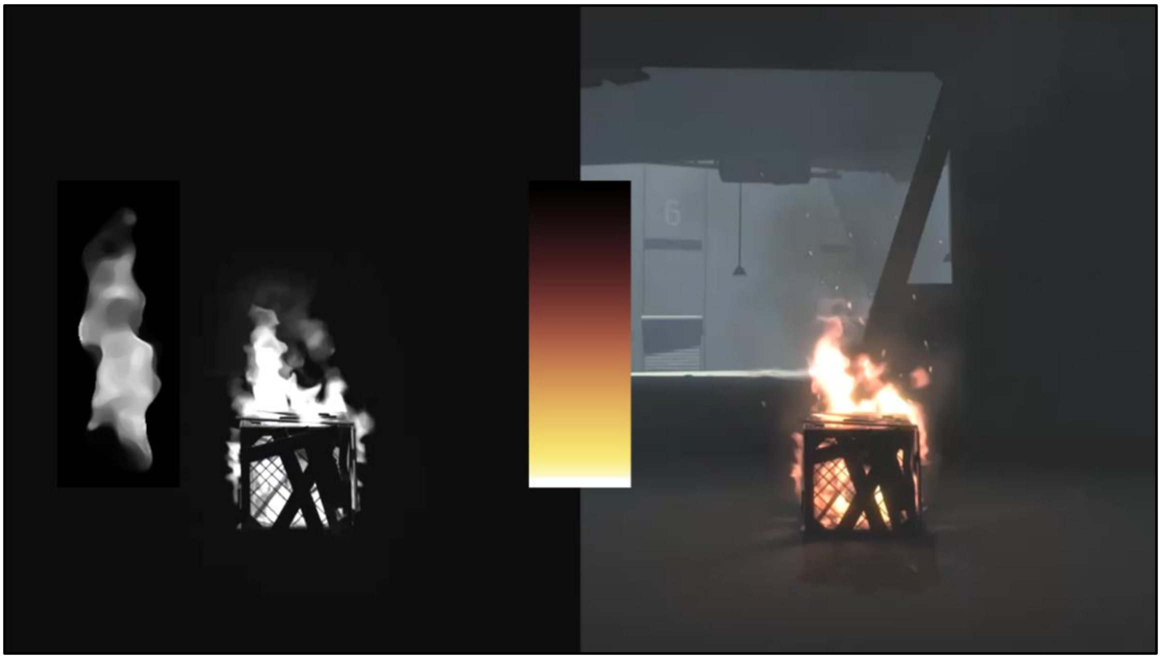
5
6



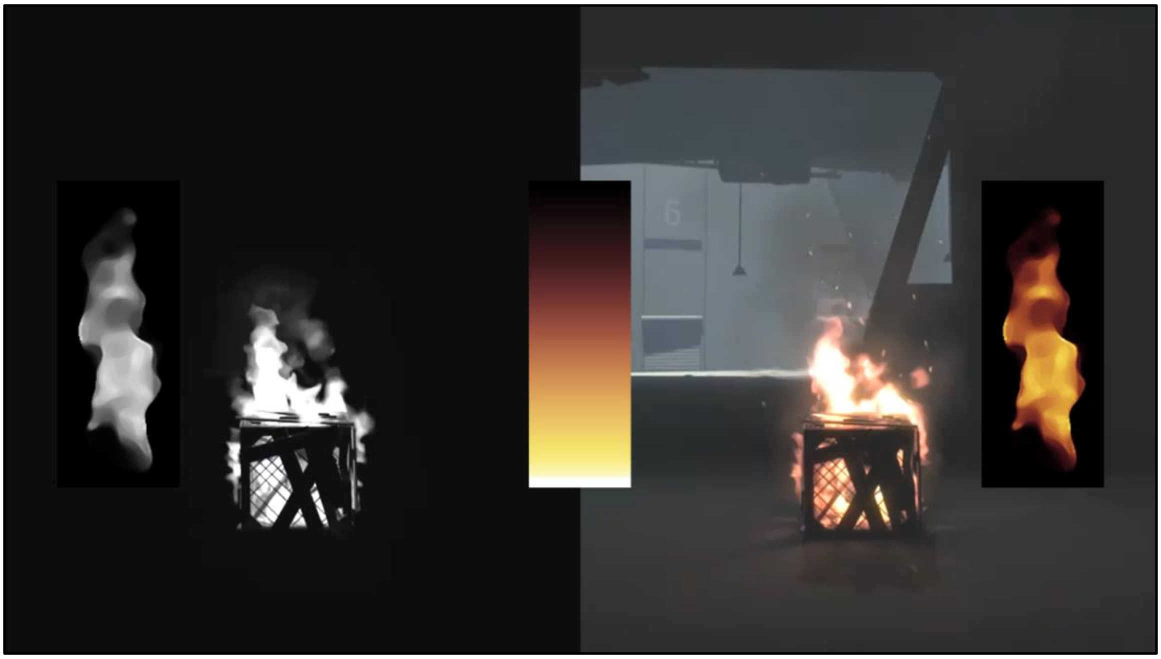
So other than fire we’ve got the same smoke shading from before, plus some sparks and lighting.
We use the same distortion trick from the smoke, plus a constant upwards motion bias since that’s where the fire wants to go.



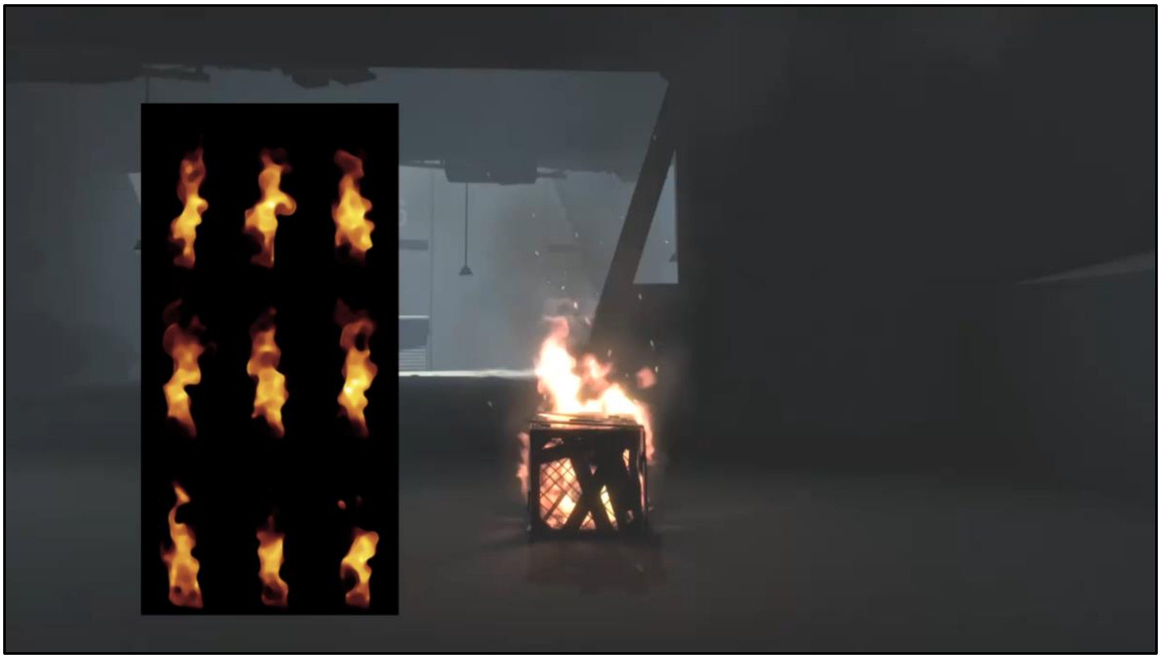
But the most important thing to make the fire look right, when compositing so many layers, was color.
So what we do is we render the fire, at first each sprite only into the alpha aka bloom buffer, since we want it to HDR bloom anyway. We do it in this deferred way, because…



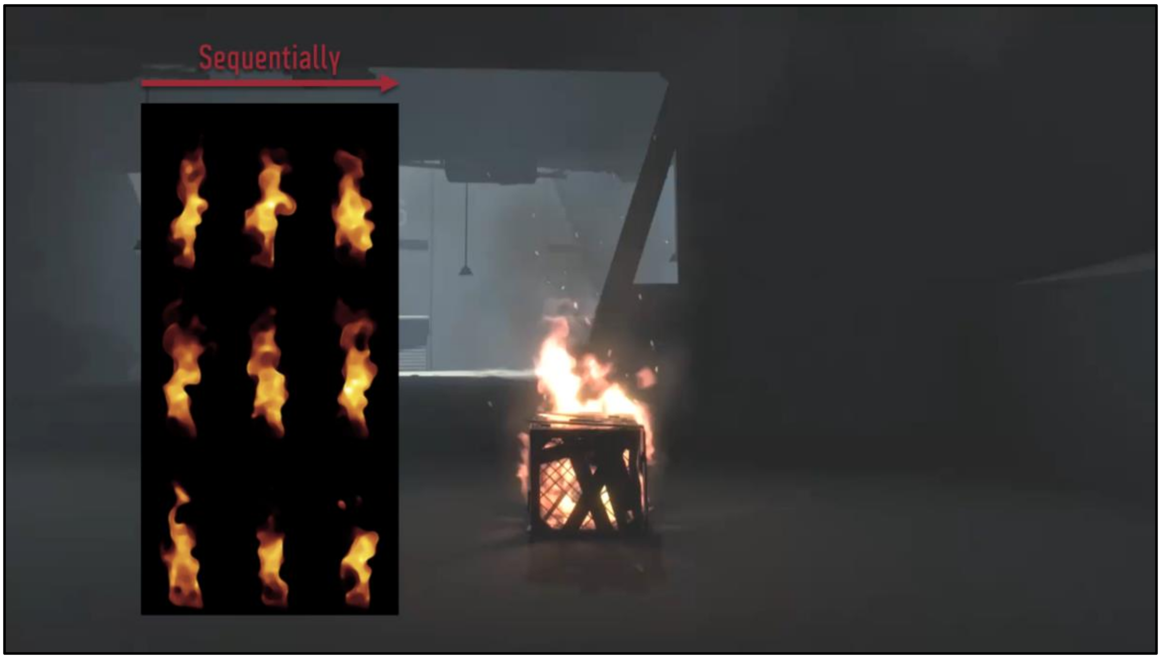
We want a consistent mapping of color to luminance, we don’t want any dark whites or yellows, and we don’t want any bright blues or reds cause by amplification via multiple layers linearly blending.



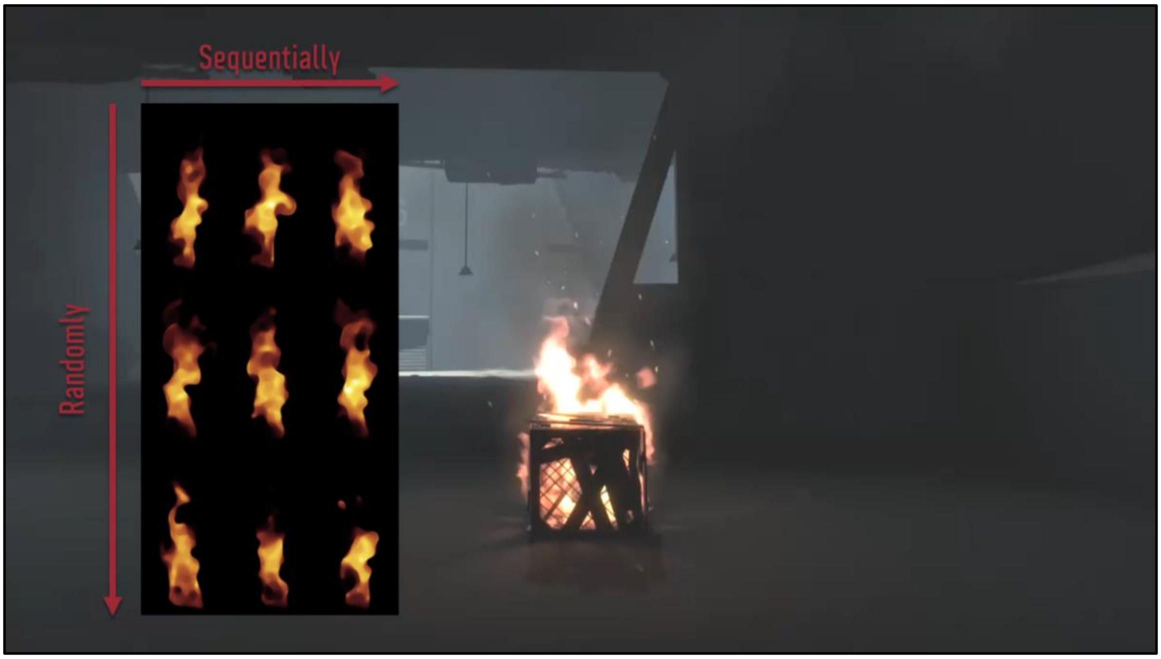
So what we do is write it onto screen by mapping the alpha/fire/bloom channel through a gradient LUT, getting insurance on our color/luma mapping.



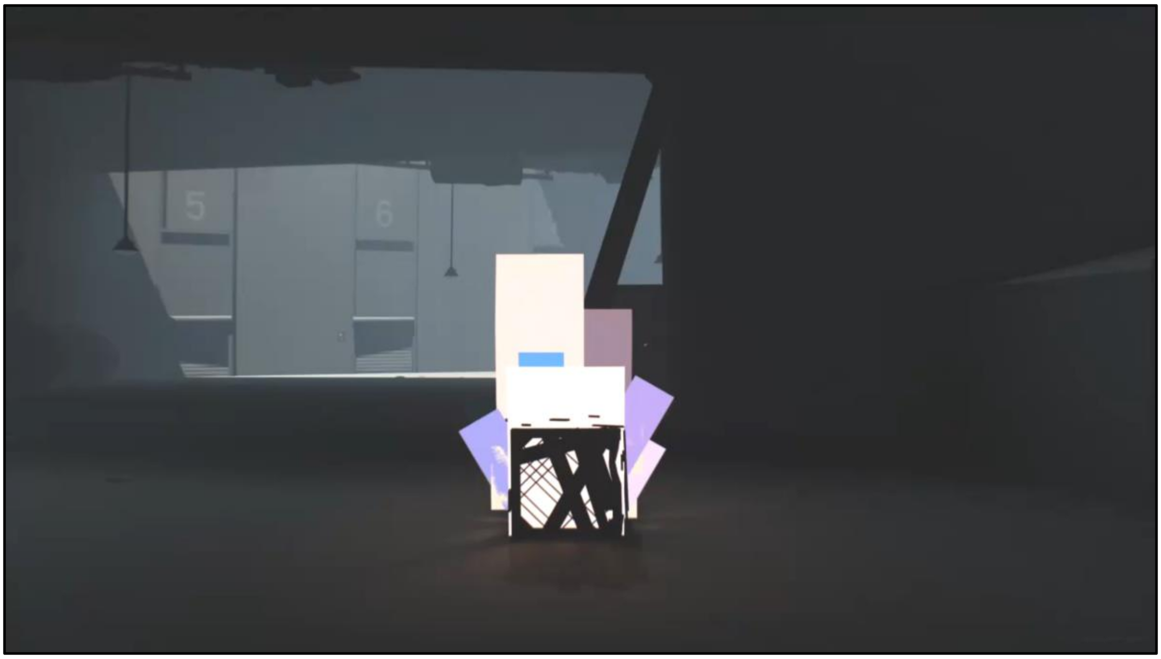
Now, just distortion is not enough motion for us, we need flipbooks! But not a movie that’s too big and too realistic.
So we have this 3x3 of random flame shapes, at fist we just cycled through them, but repetition would happen every second, and it looked bad.
Second attempt we just tried picking random ones, but with only 9 sprites, there’s an 11% chance of hitting the same one twice, which looks like a pop or lag.
Our solution was… Why not both?



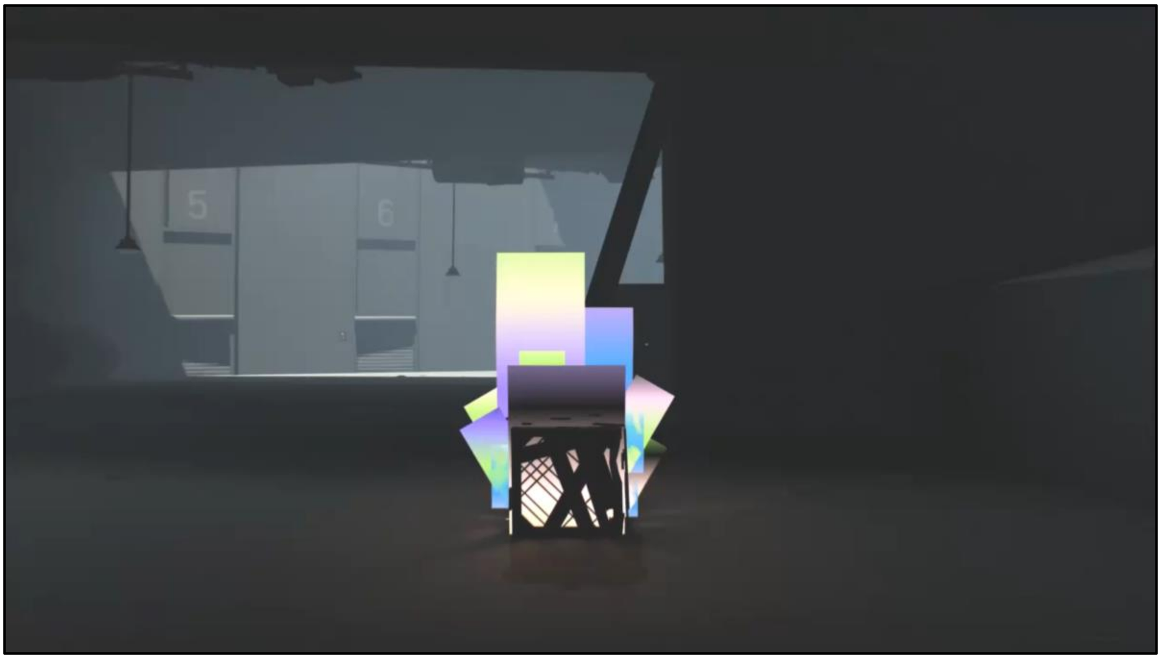
So we choose column sequentially…



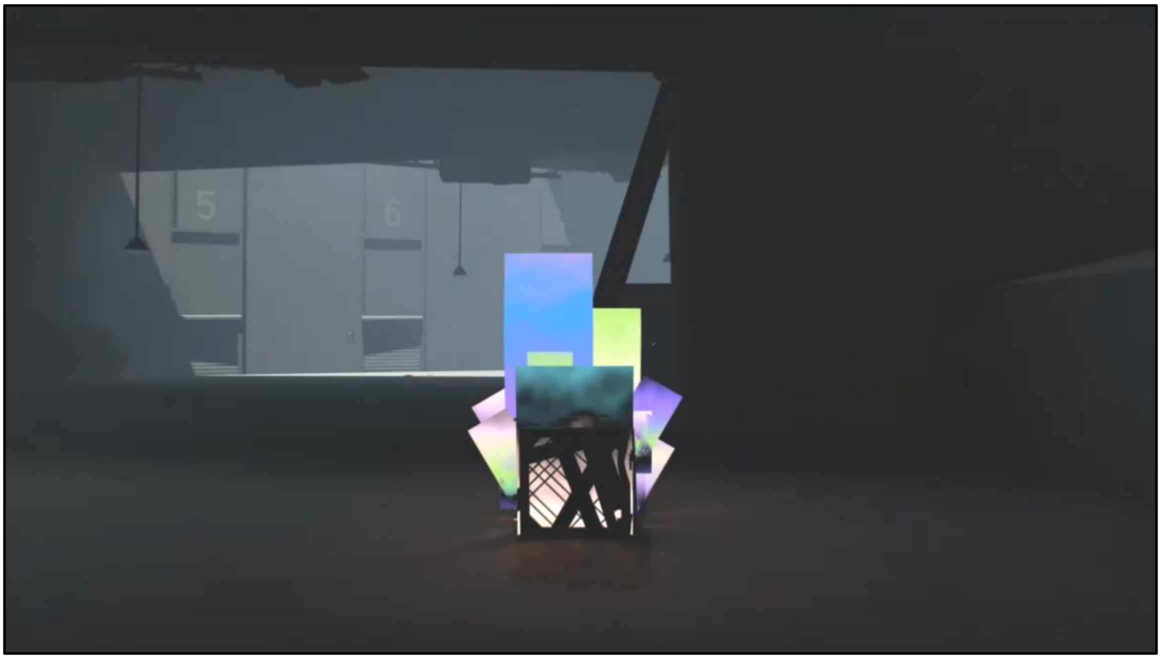
And row randomly, to get the best of both worlds!



Obviously we don’t wanna just cut between them, so we… Fade between them, using time as phase!... No…

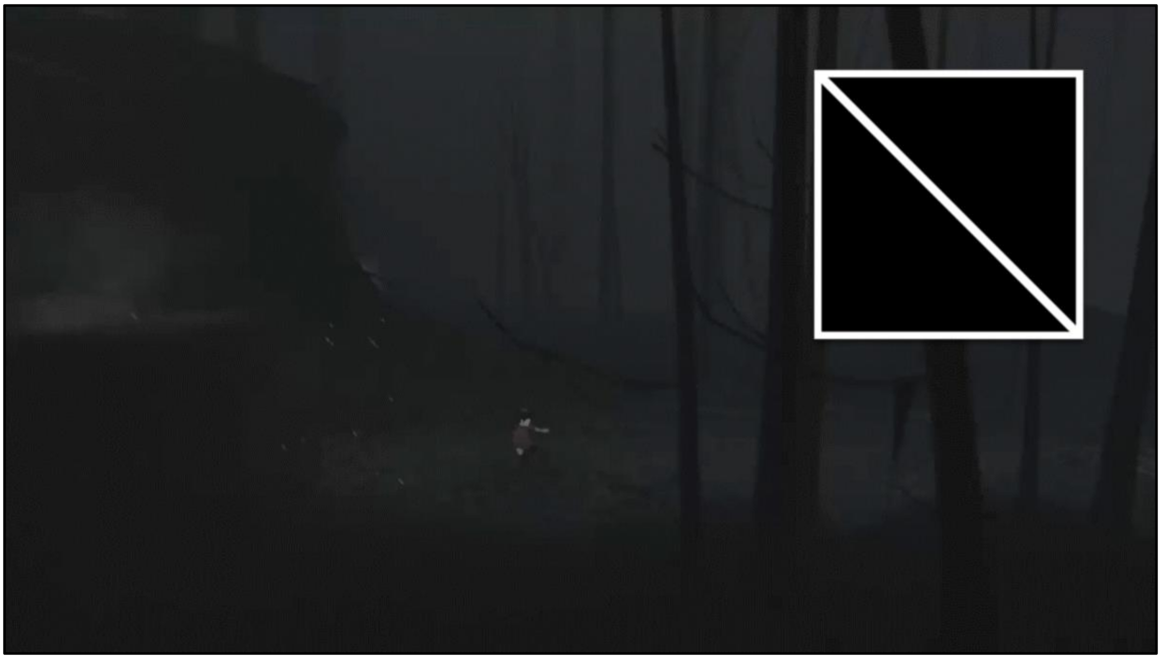


We add a vertical gradient to that phase! Cause that’s the direction of movement!... No…

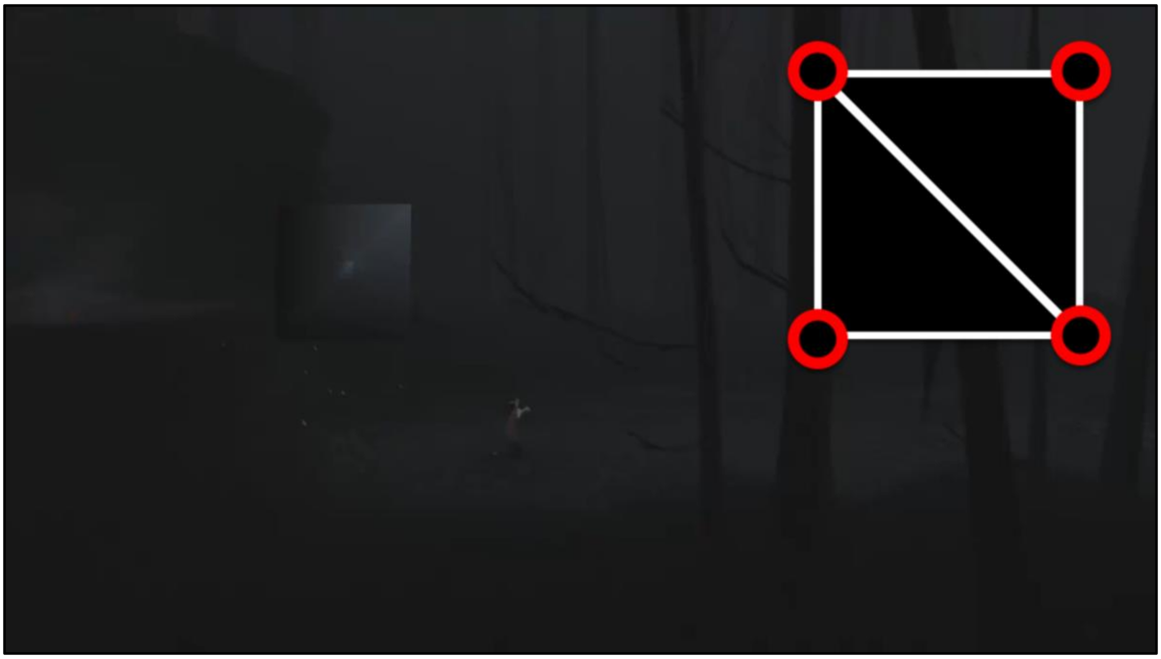


We add a noisy gradient to that phase, cause why not!

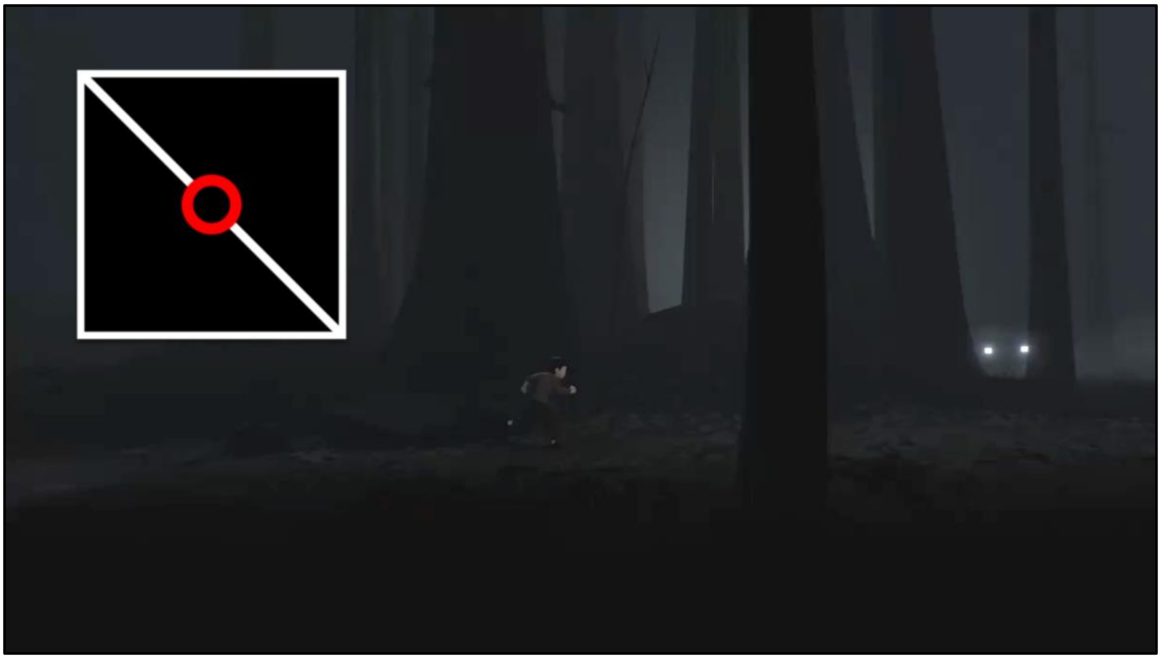
Next up I wanna talk about lens flares. Ones made not by post process but by…



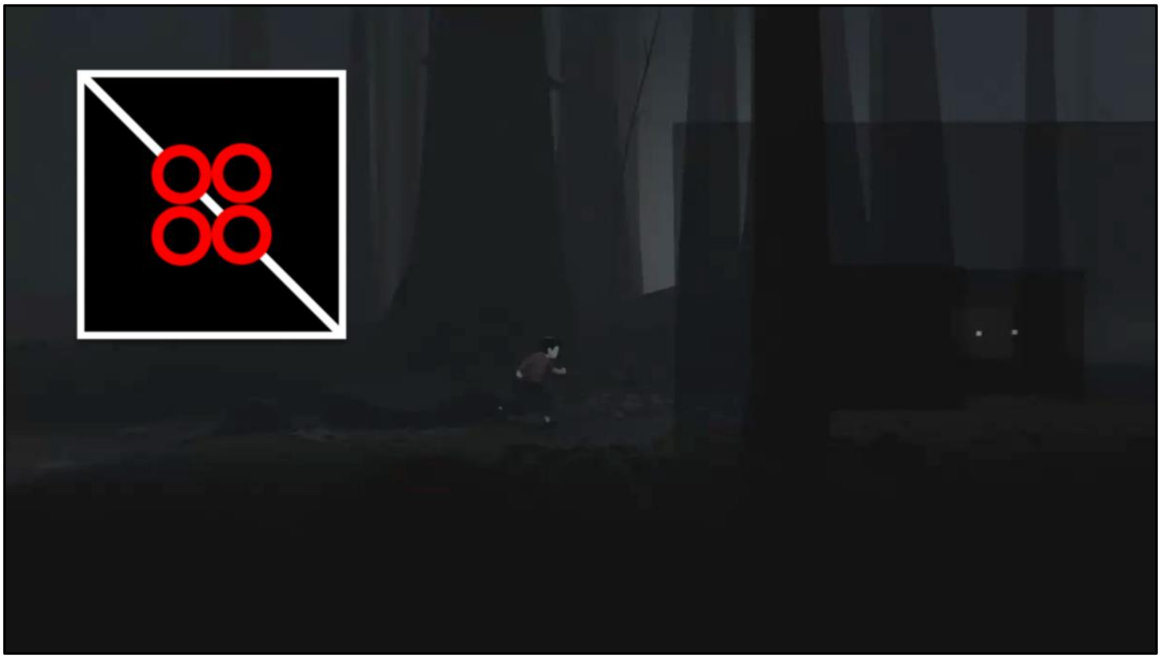
By placing sprites in the world, onto flash lights, like this one.
We didn’t wanna ray trace against collision since that’d be expensive, plus who wants to put collision on these trees that you’ll never even touch anyway?



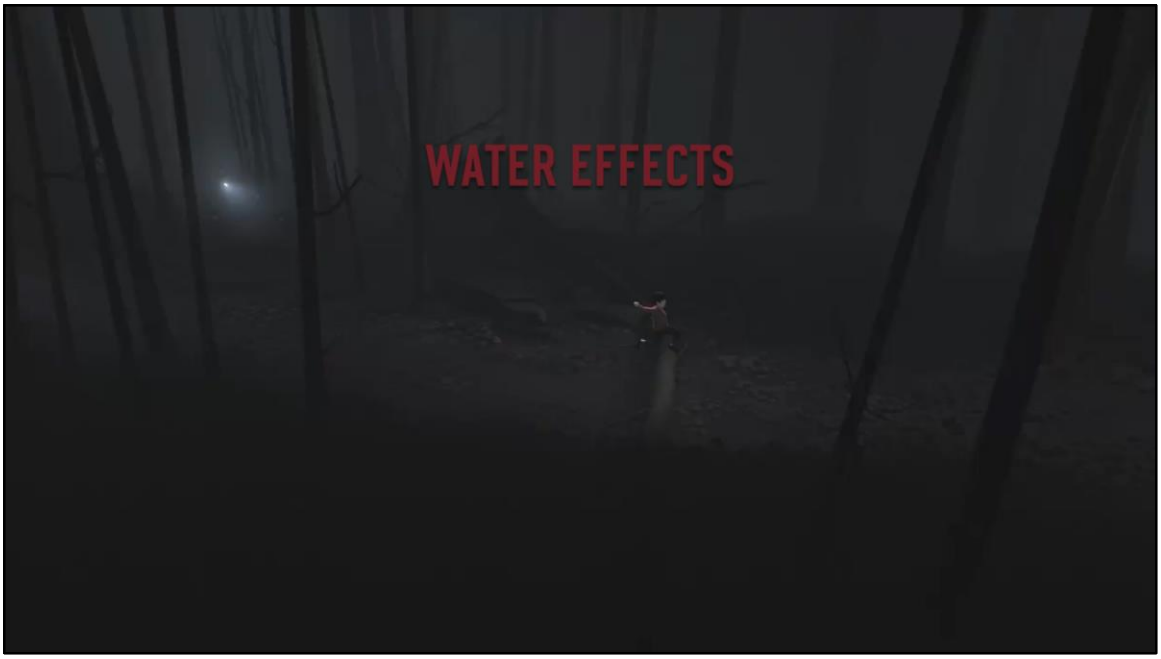
So we sample the depth buffer, in the vertex shader, and multiply our flare texture by the result in fragment.
We sample a bunch of times on a golden ratio spiral, like 32 times, it’s only for 4 verticies anyway.



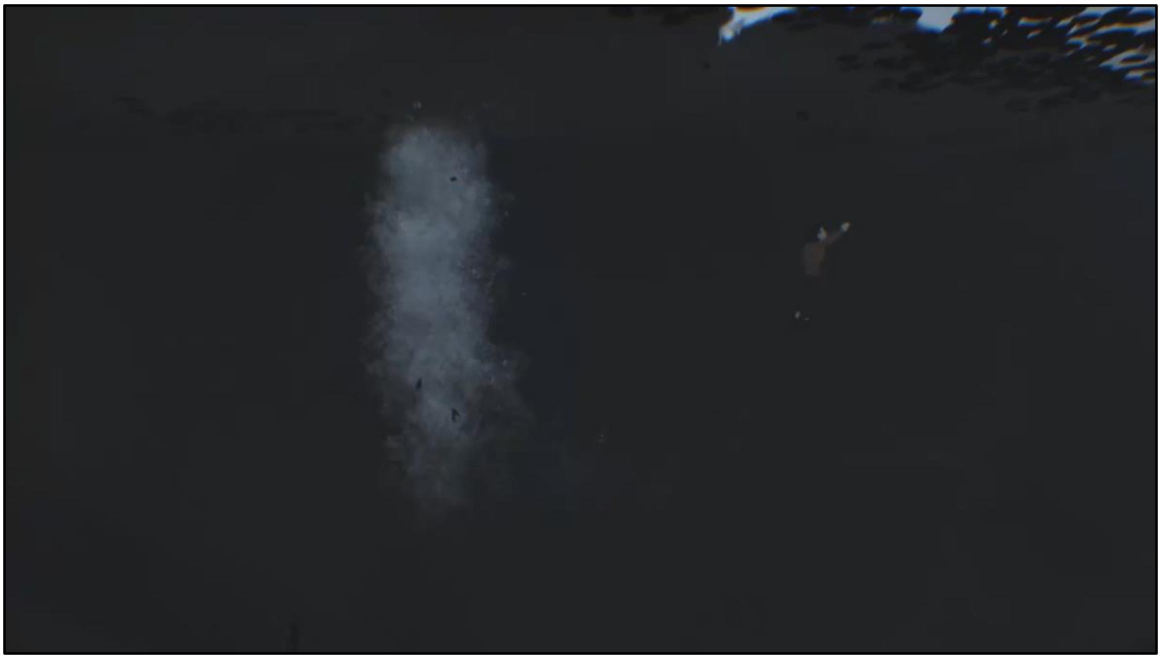
But of course we don’t sample the corners, that’d be silly, so we sample the center, a point we can just get from the modelviewprojection matrix.



But we also don’t sample from *just* there, we sample somewhere in between, use a little offset of like 10% toward the corners so that we get a bit of a gradient across the sprite in the end, for free.

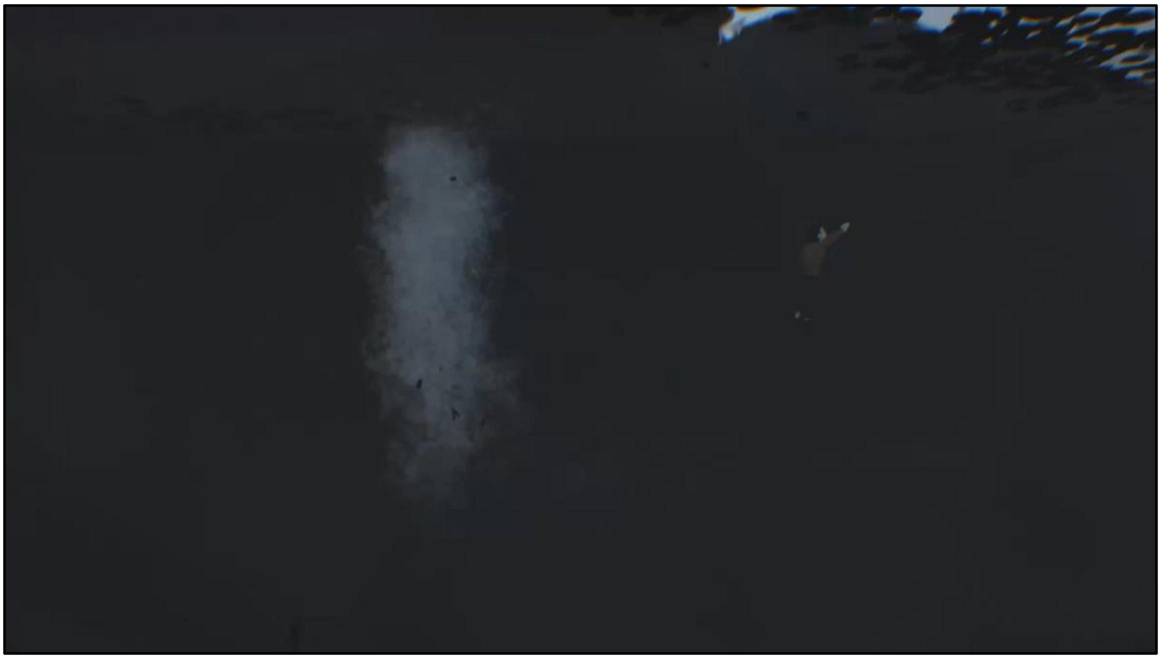


Next up let’s look at some effects for water.

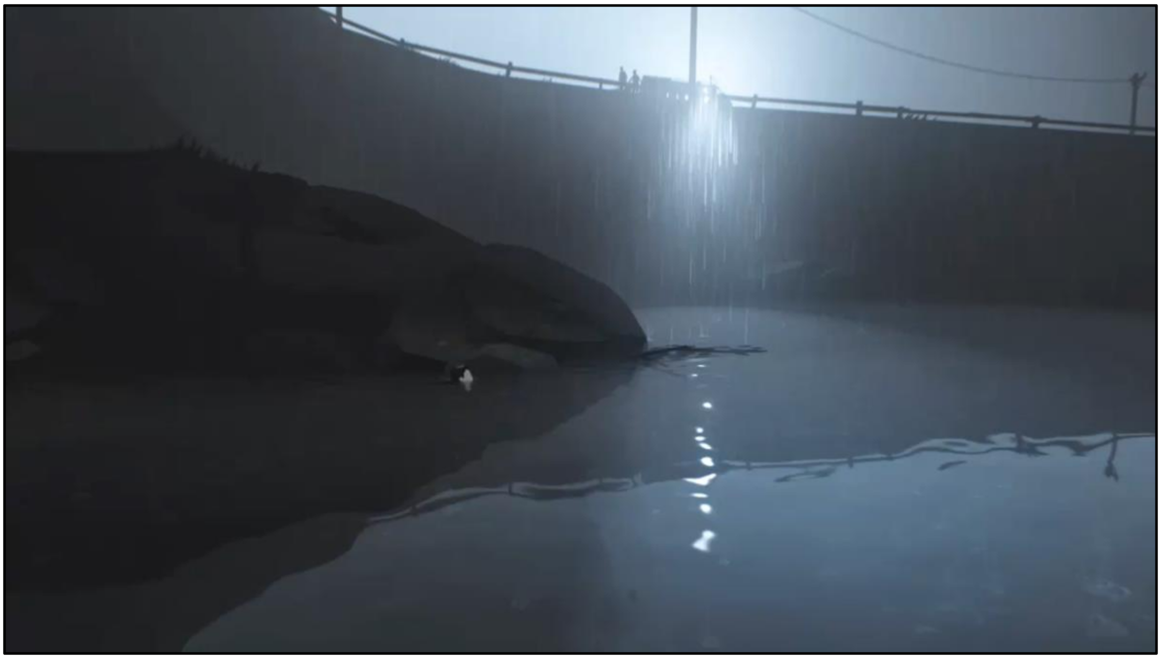


First up is this column of foam, it’s using the same shading as the smoke from before, so not gonna go into details, just gonna show you what it’s like if effects get removed.

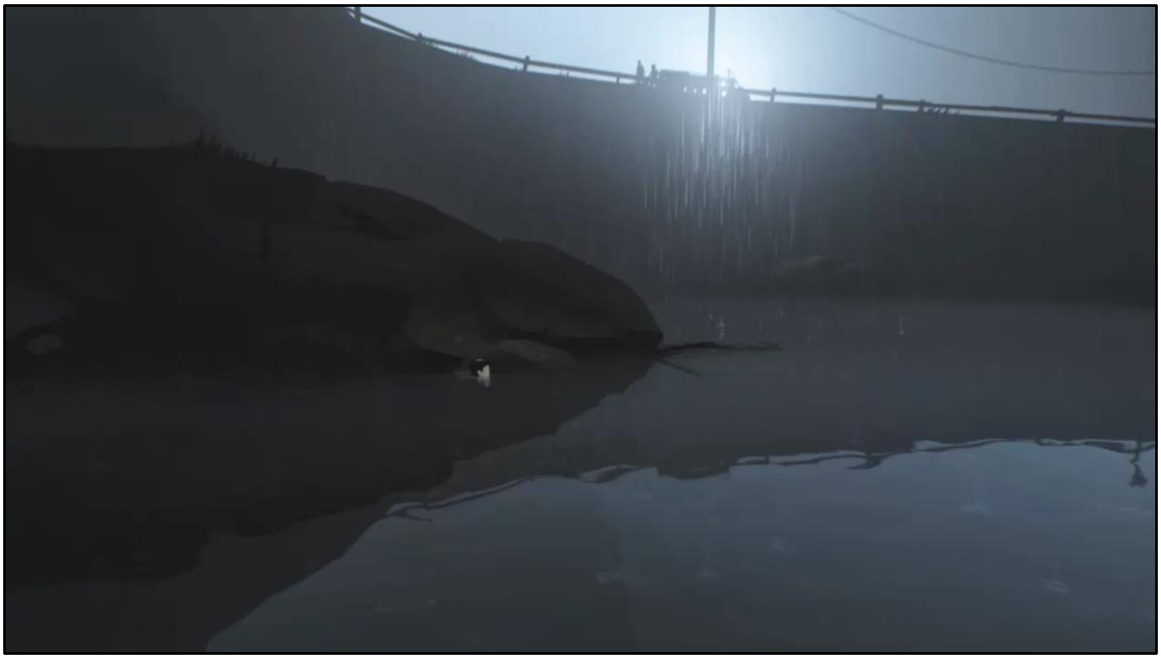
Here’s without motion distortion, the distortion helped sell the bubbling motion.



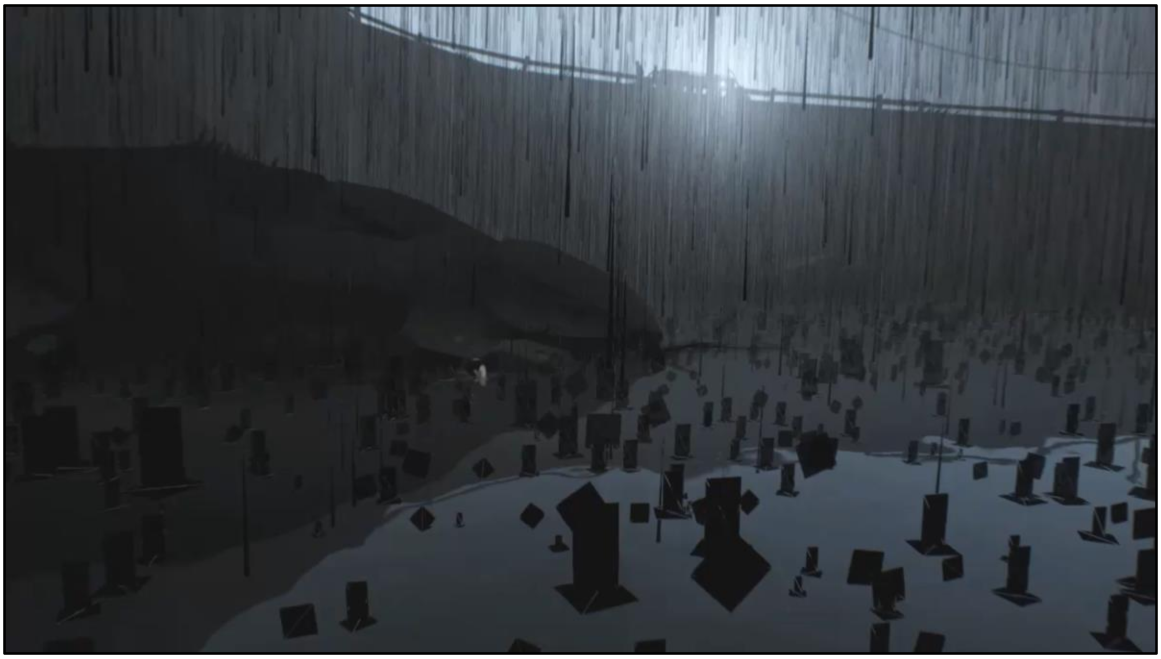
And here’s without the lighting, the lighting gives an otherwise flat foam some depth. It’s just making the top brighter than the bottom per sprite, again.



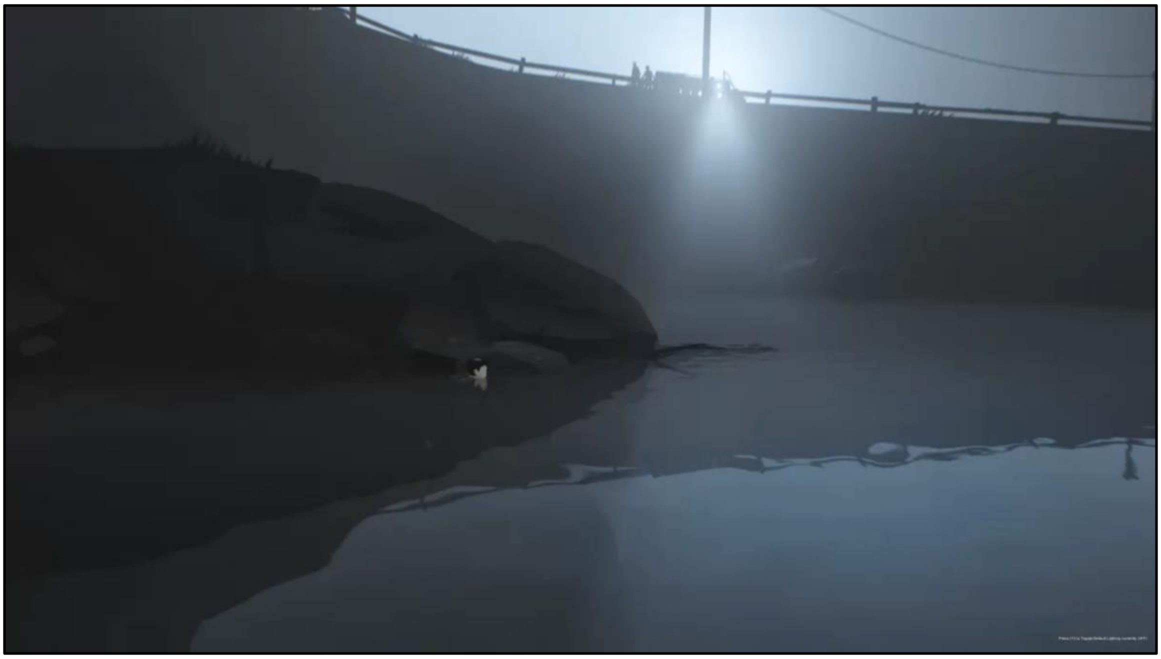
So up here, the importance is with the flash light. If the boy swims into it, he dies. So we emphasise it with 3 effects.



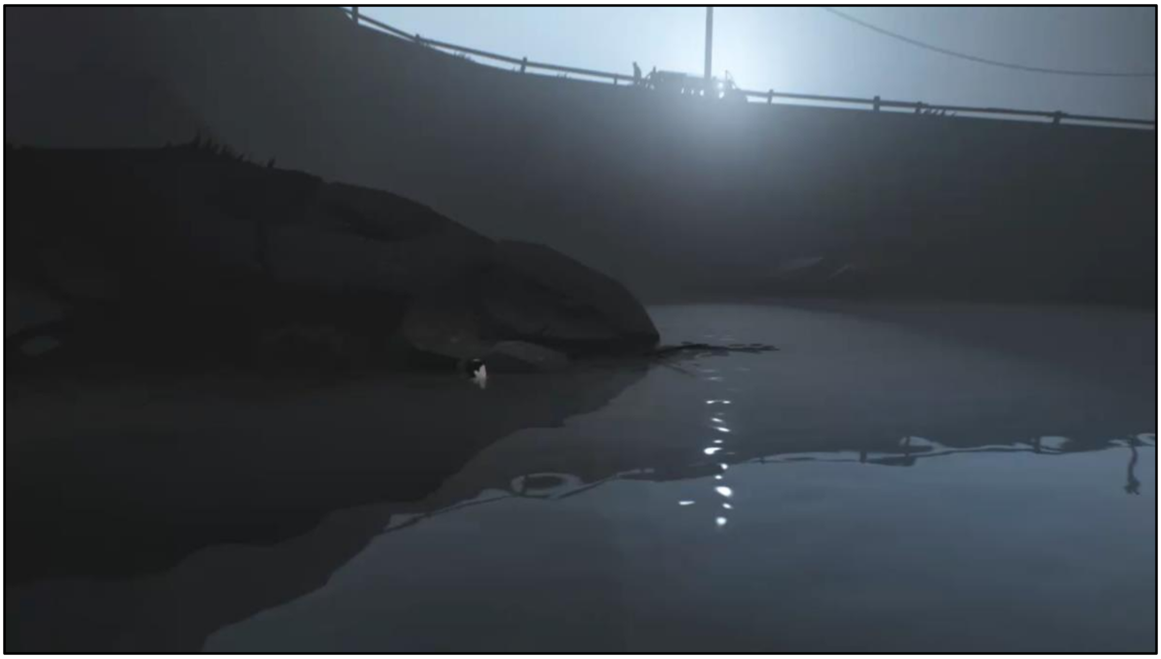
Firstly, the rain.
It can be done in a few ways and we tried 3 of them.
You could use scrolling textures as post, but it'll lack some real parallax.
It could also be individual particles, but the generation cost on the CPU is heavy so also a no-go.
Could also be billboard particles, but the overdraw between the drops is too intensive on the GPU.
So we went with…



Vertex shader animation!
This is the wireframe of the rain, there are a few different effects per drop. They all share the same principle of putting them in a random position seeded with a whole number or integer part of time and an offset, and unfolding their animation over the fractional part of time.
One type is the falling lines, the animation here is just move down from the top to the bottom of the mesh’s bounding box.
Then there are the splashes, they all scale up from 0 to full in their animation and fade out from some intensity to 0.

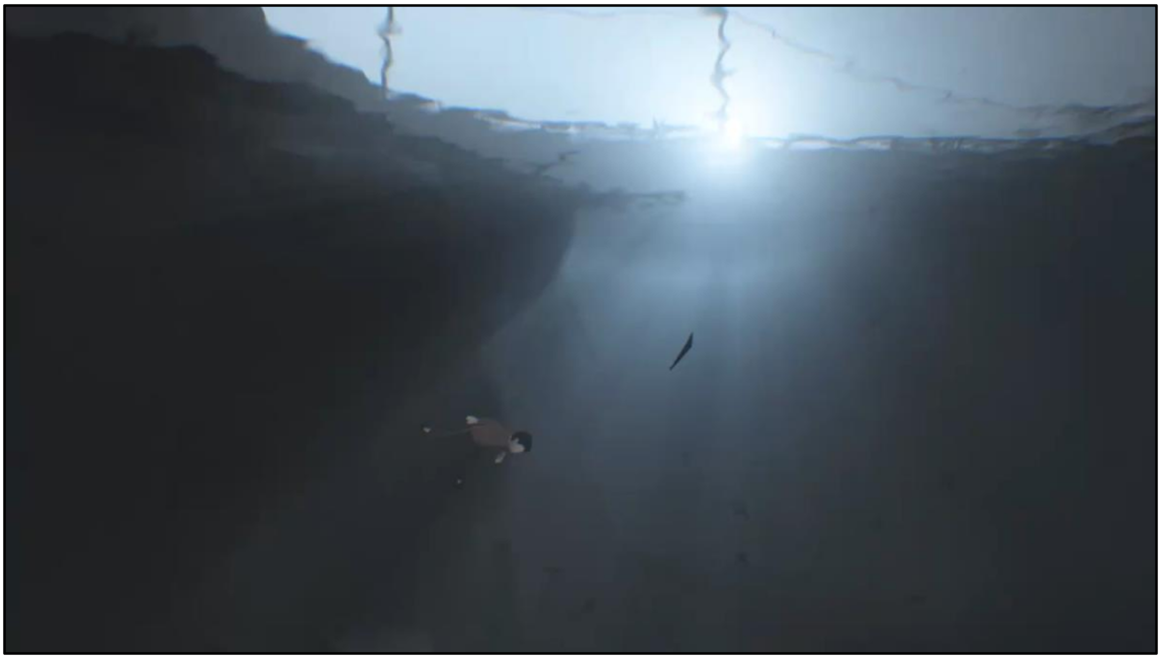


Second effect, already covered is the volume lighting. Both above and below, with different settings and sorting times.

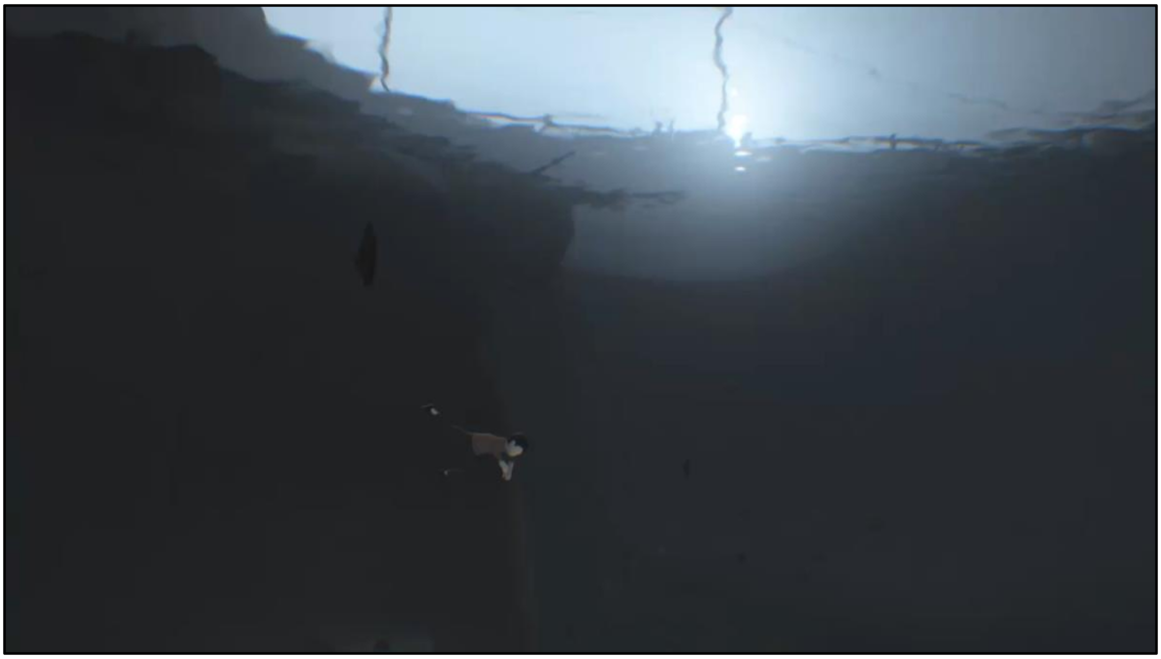


And finally some specular and diffuse lighting on the surface.

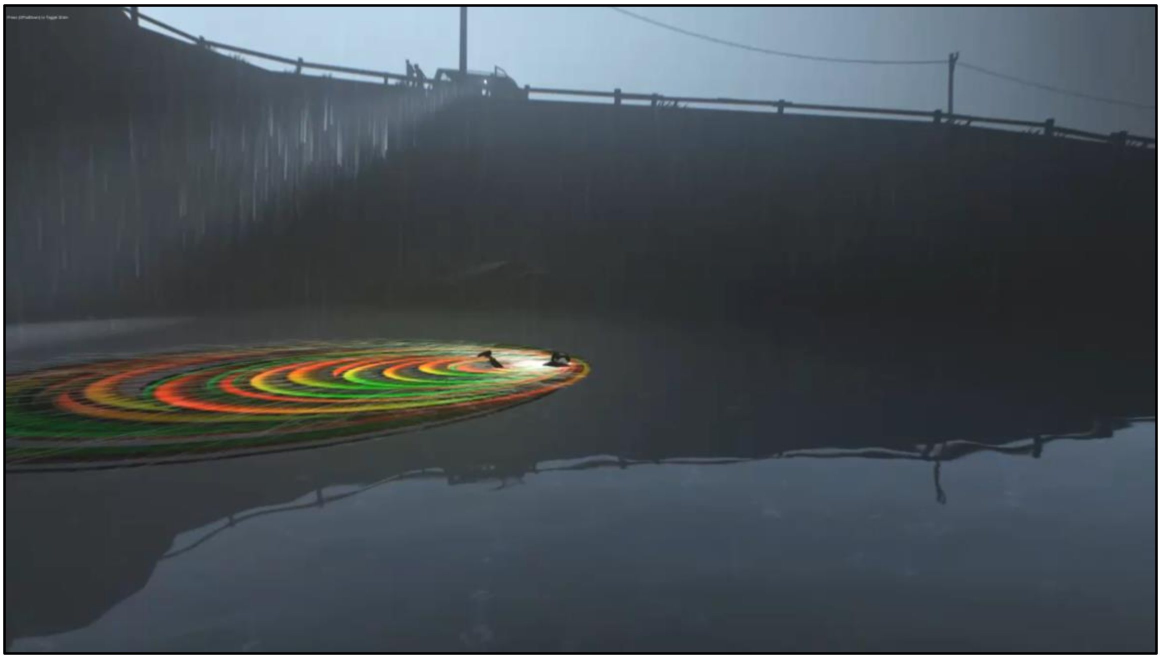
Down below we have something similar to the rain, little dust or dirt particles. They use the same vertex animation technique to make lots of them fast.



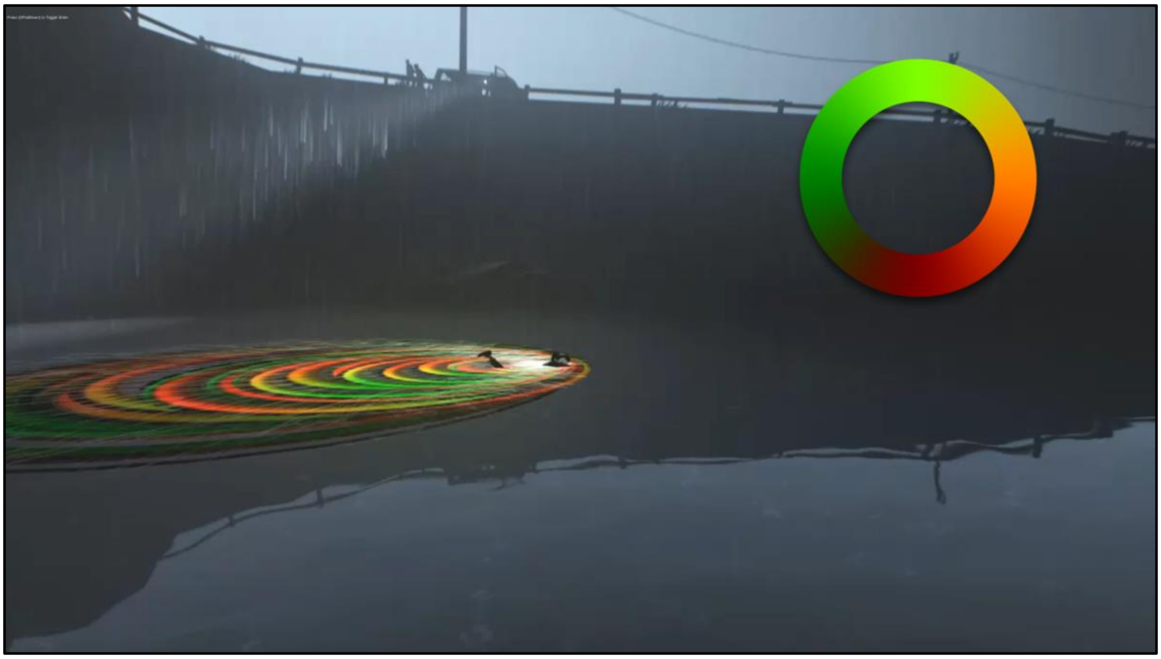
Down here the volume light has an animated texture of caustics to give it some life and underwaterness



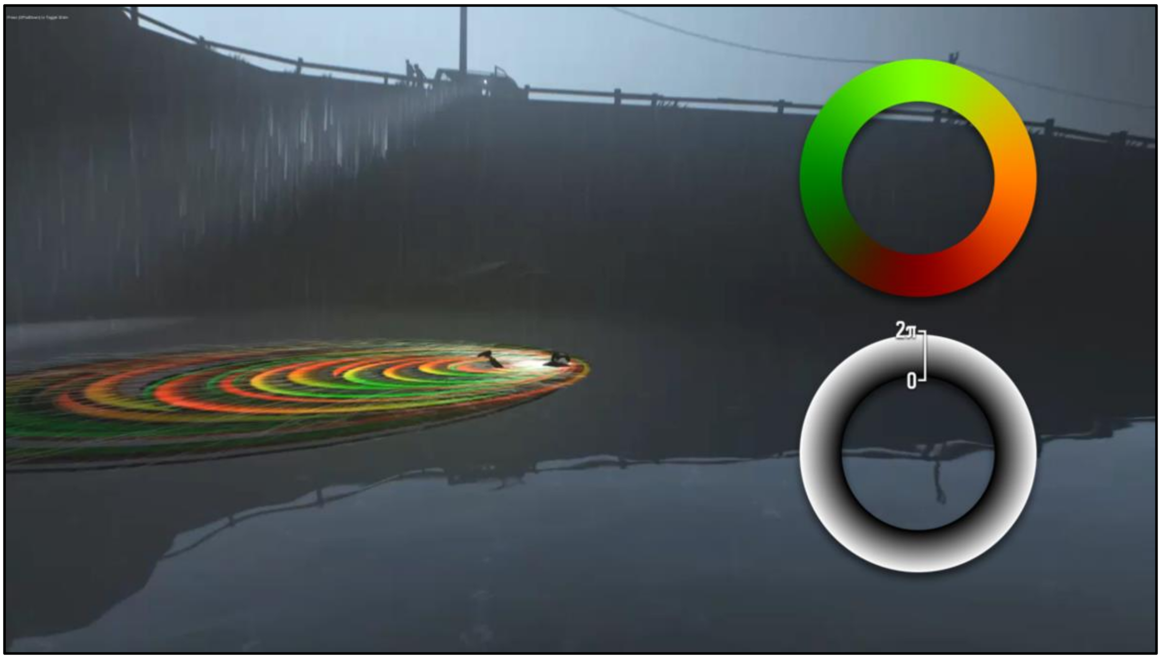
Finally that specular again, except this time it’s not specular reflection, but “specular refraction” pretty simple but fun effect.



Coming up, we see the boy is generating these waves. They’re made from mesh particles, using a ring shaped mesh that expands out.
The shading is simply using some new, analytic normal to resample the reflection and refraction, we generate those normal by…

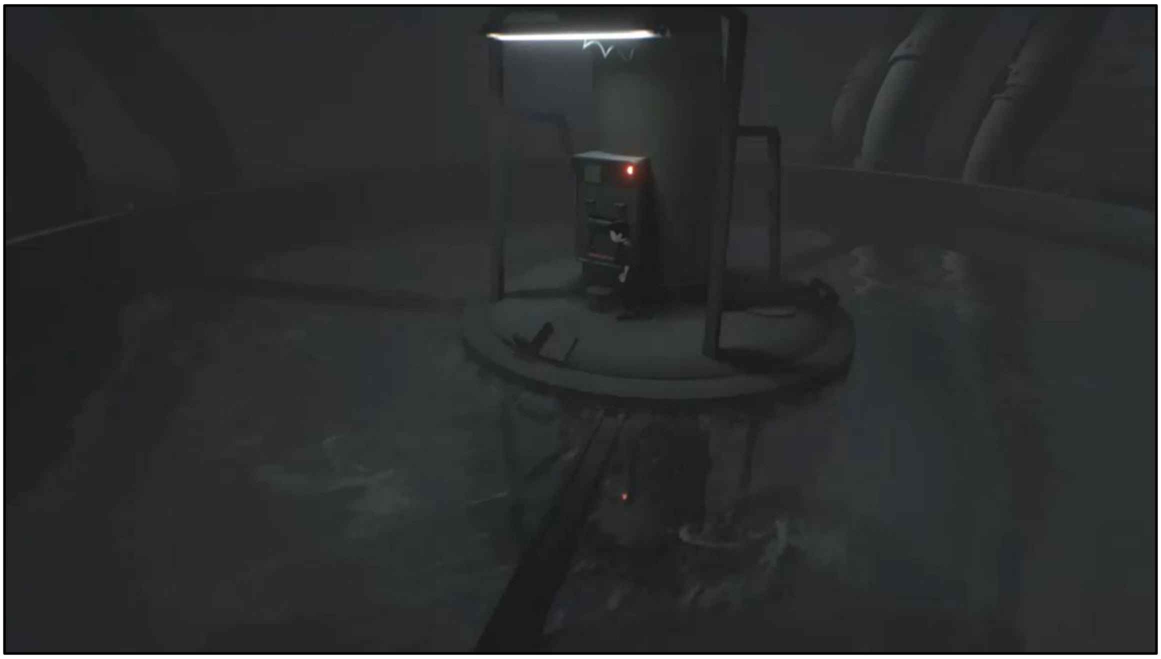


First using the local particle position as a normal, pointing away from the center, then…

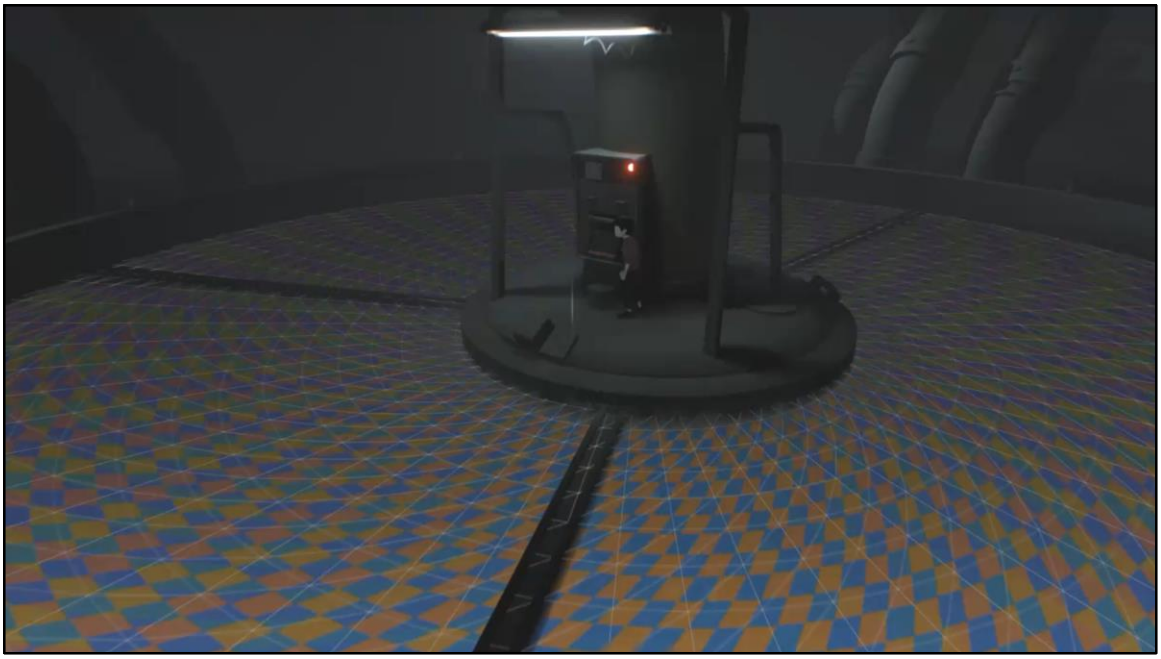


We multiply that by a sin going from 0 to 2pi/tau, to get 0 at the edges and the middle, inward normal at one third and outward normal at two thrids.

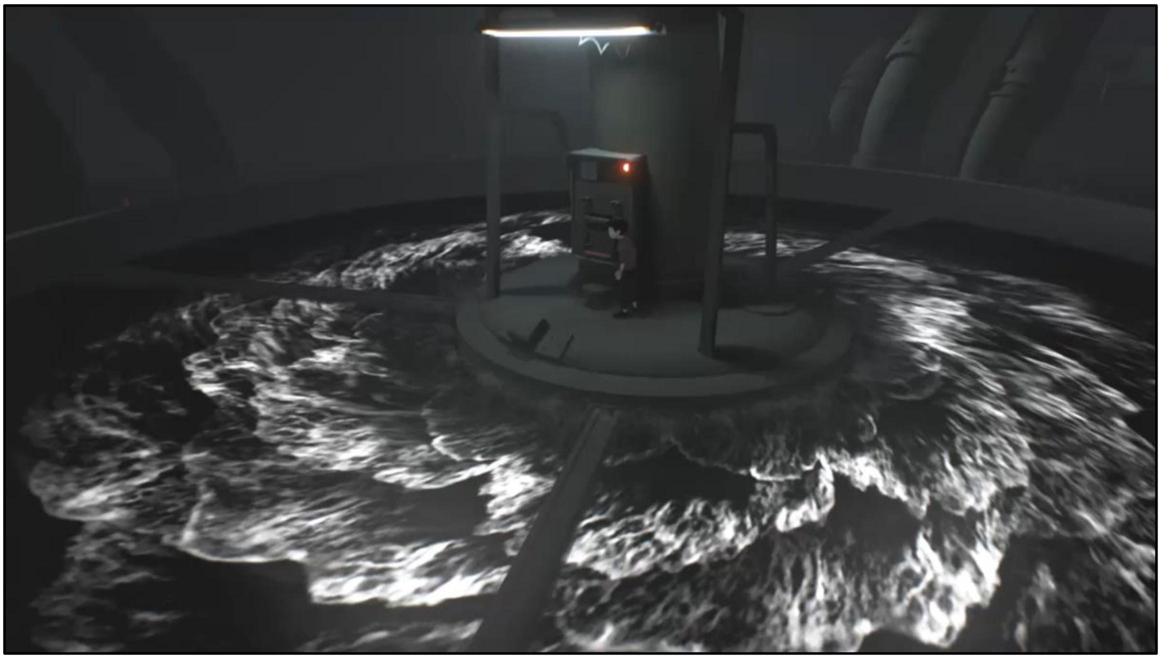
Now, let’s talk about another shape of water.



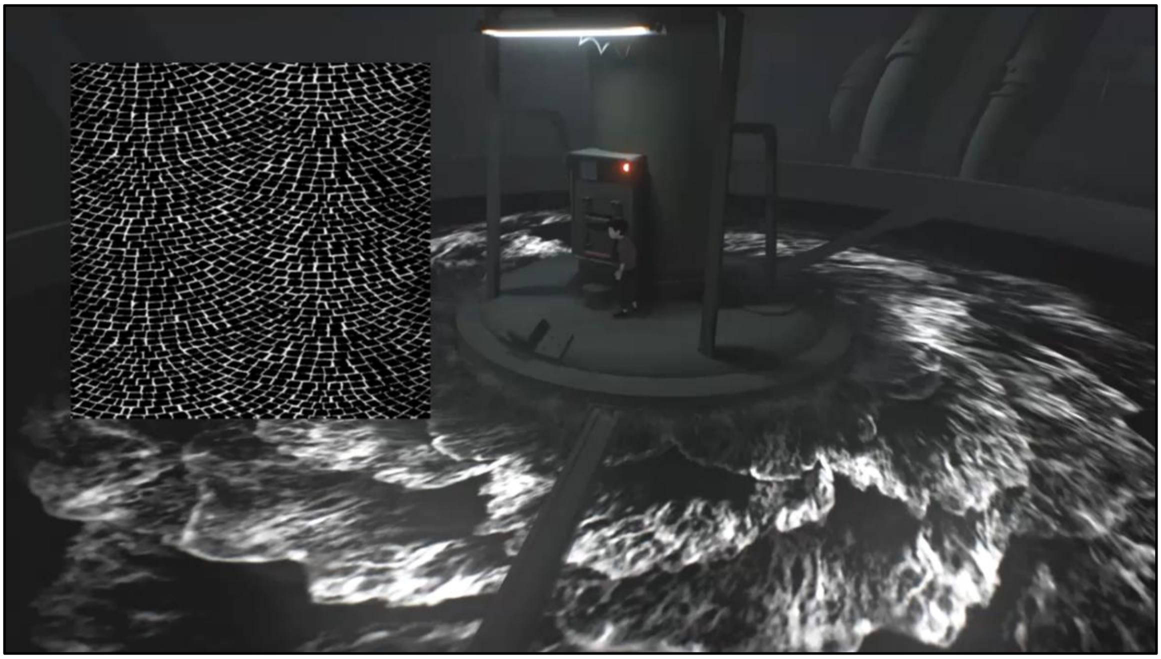
So this water is not a cube, it’s a cylinder, and it’s using outward motion.
We could have done this motion using zoom and fade, but we choose…



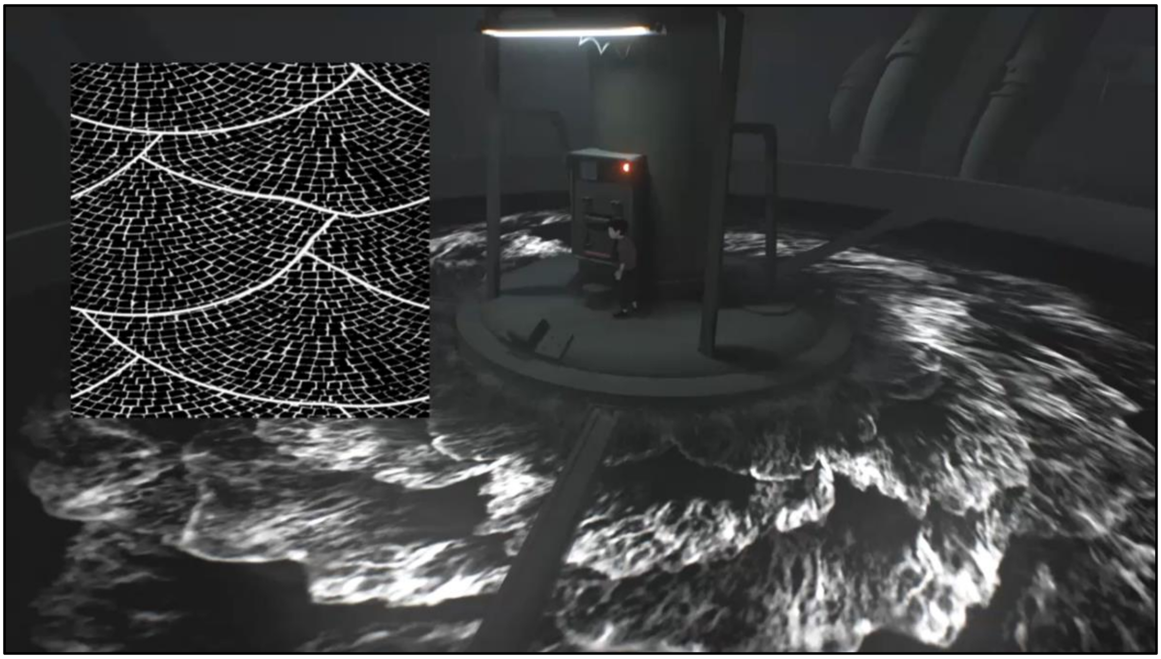
Mapping radially and scrolling outward, the mesh is fairly high poly so we can do all scrolling and fading math in vertex.



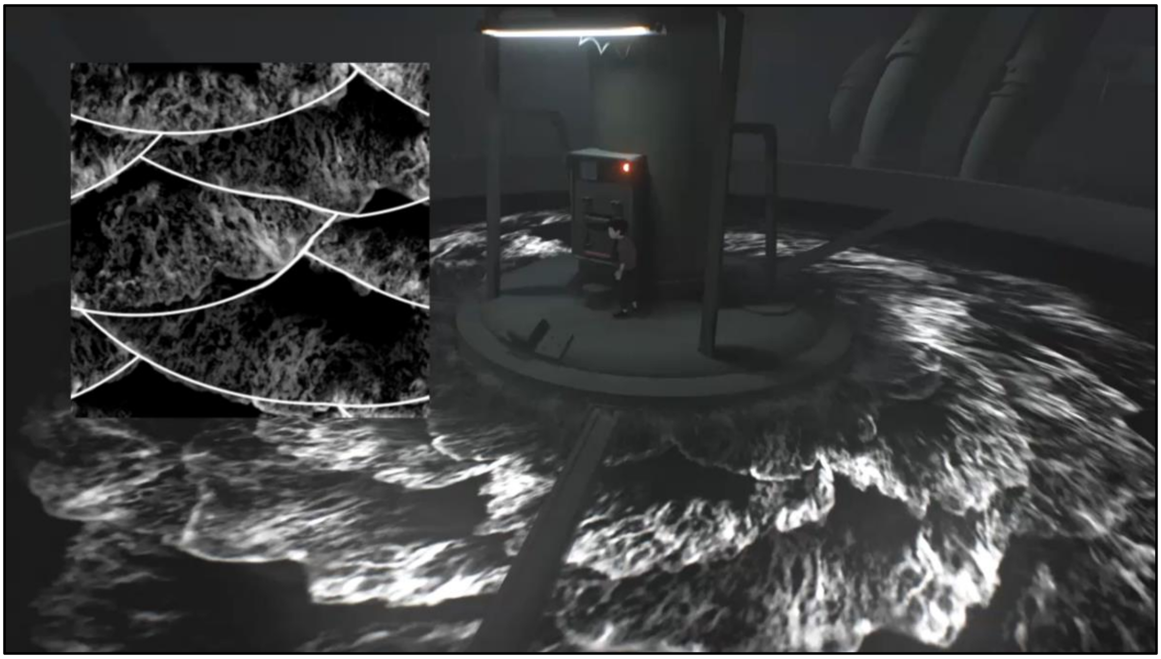
The texture of course has normal, but more importantly the foam. So we needed something that looked like waves of foam, tiled so we could scroll, but in a nonobvious way.



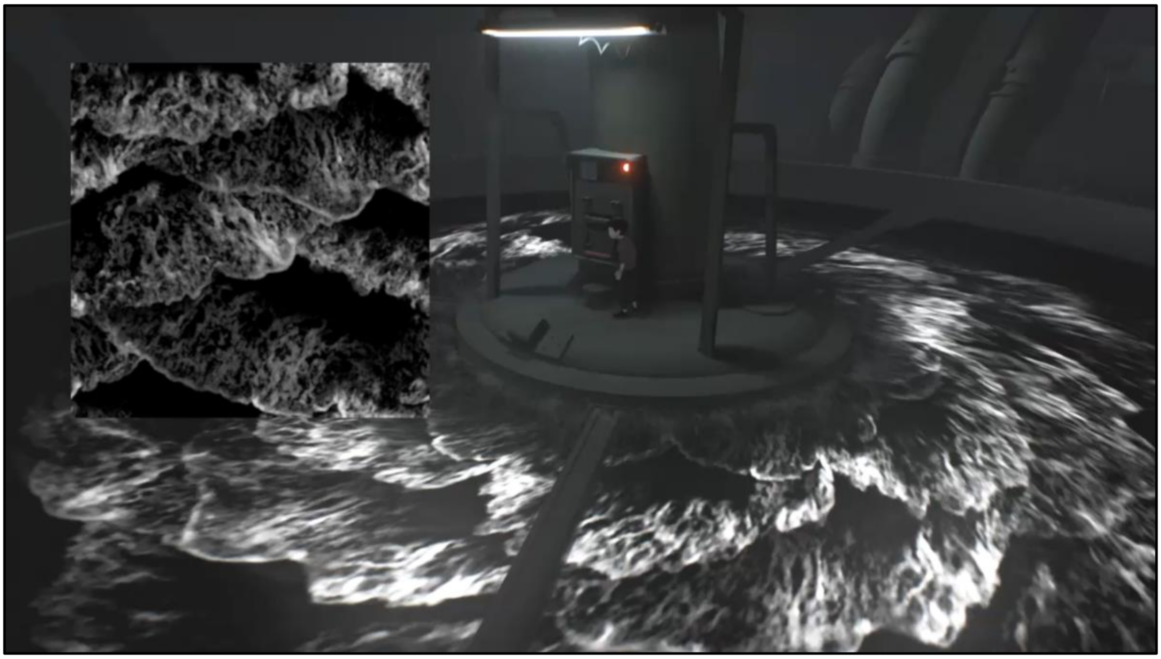
The pattern we went for is called the European fan or wave cobblestone. It does tile, but in a non-tilly way.



Drawing waves on top to get the shape template…

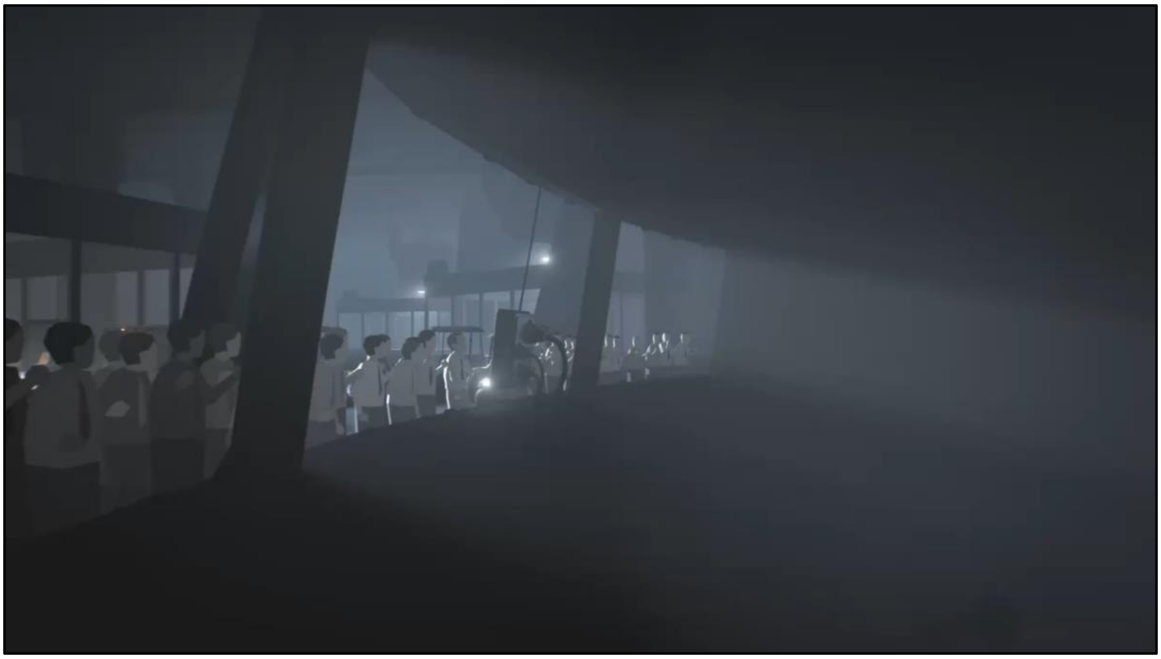


Then using pictures of beach waves that are aligned with said template lines…



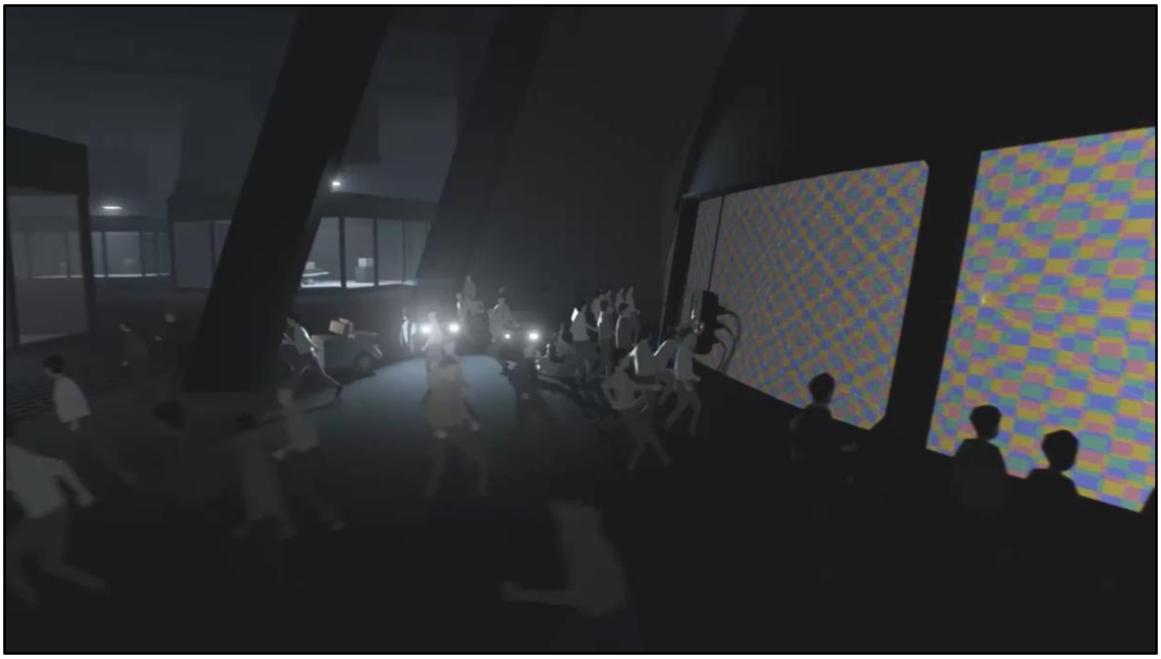
Remove the lines, and we've got a nicely tiling wave map!

Finally…

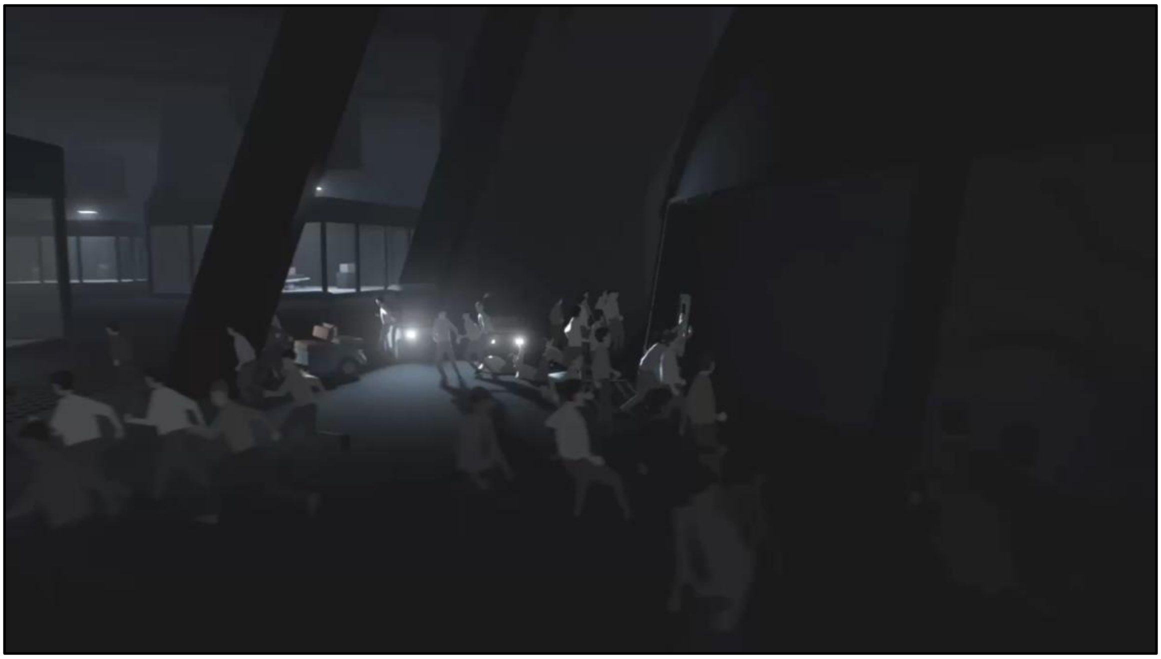


So this effect uses a lot we’ve talk about up until now.

First we got a water volume using the foam texture from before, this one isn’t just a cube though, it morphs using morph targets aka blend shapes. We’ve got 3 targets/shapes, one for before, one for during and one for after



We tessellate the mesh only on the flood and along the windows to show the displacement. We scroll the texture toward the flood and then out along it.



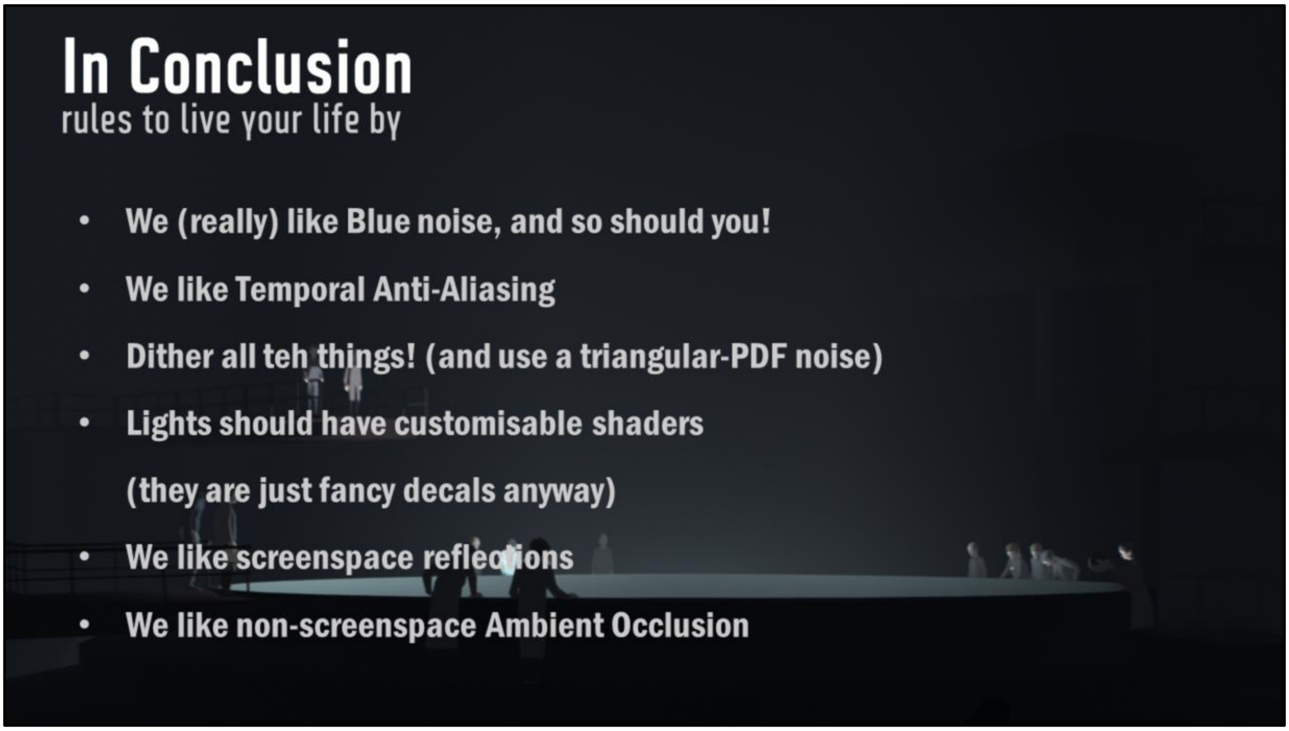
Second we got this decal on the ground using that same foam texture again, as well screen space reflections.

To really sell the motion on this one, we wanted it to look like it was really *pushing* this the foam outward, so going faster and stretcher in the beginning and slowing down at the end. This was achieved by simply adding a power function to the V of the UVs, starting at a power like 4, and gradually going down to simply 1 and linear.

Finally, and most visibly, the foam. First we have a big carpet of foam on top of the flood, with the same movement speed and shape. Lit from behind, using all the same shading as earlier.

Then we’ve got an impact effect to melt together the flood and the decal.

And finally just some sprays to make it more intensive in the beginning.



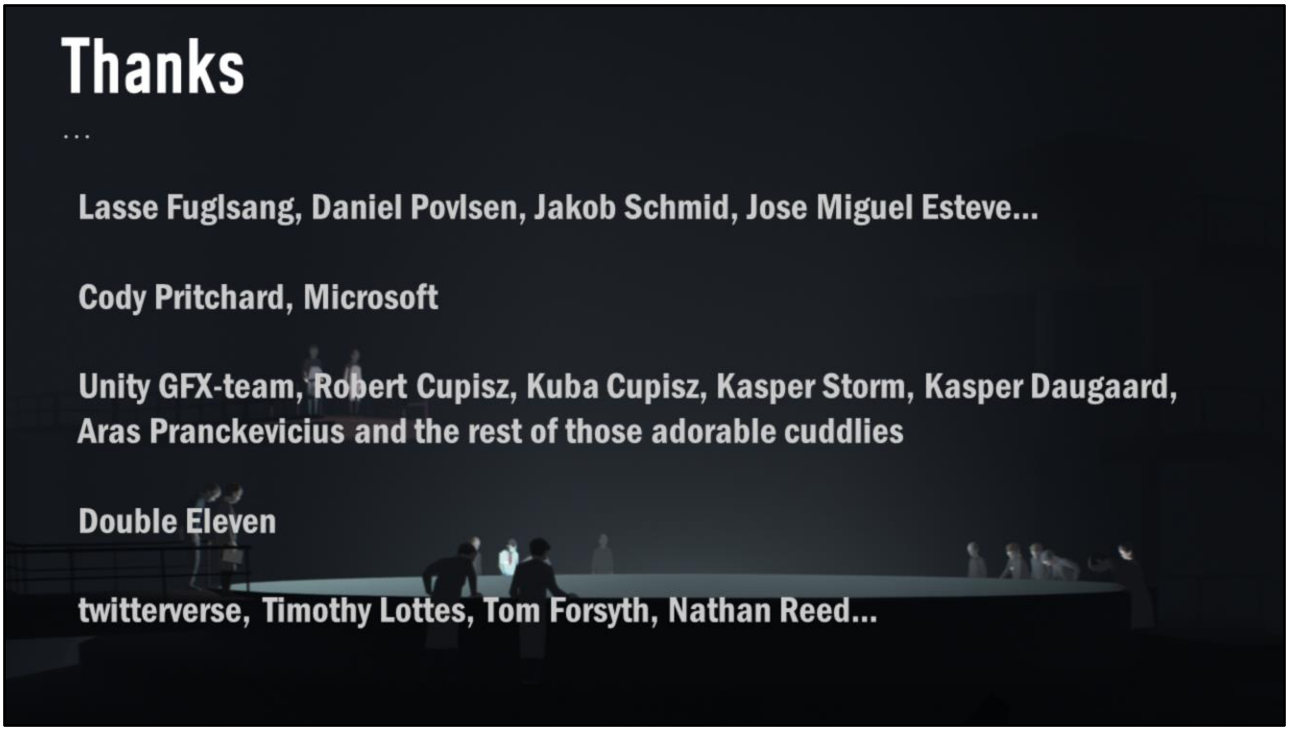
In Conclusion

rules to live your life by

- We (really) like Blue noise, and so should you!
- We like Temporal Anti-Aliasing
- Dither all teh things! (and use a triangular-PDF noise)
- Lights should have customisable shaders

  (they are just fancy decals anyway)
- We like screenspace reflections
- We like non-screenspace Ambient Occlusion

TODO: illustrative images instead of bullethell!



Thanks

…

Lasse Fuglsang, Daniel Povlsen, Jakob Schmid, Jose Miguel Esteve…

Cody Pritchard, Microsoft

Unity GFX-team, Robert Cupisz, Kuba Cupisz, Kasper Storm, Kasper Daugaard, Aras Pranckevicius and the rest of those adorable cuddlies

Double Eleven

twitterverse, Timothy Lottes, Tom Forsyth, Nathan Reed…

TODO: illustrative images instead of bullethell!

PLAYDEAD IS
HIRING

Ba^H^H Bonus Slides!

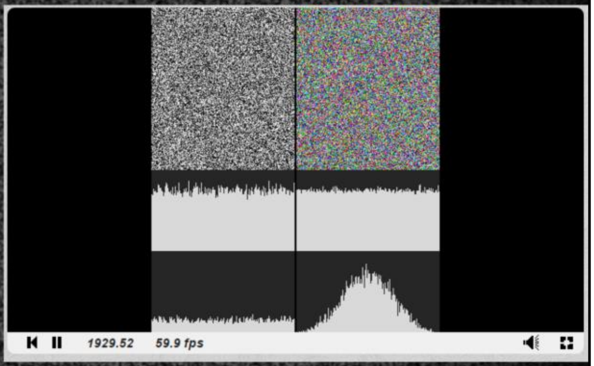
TODO: illustrative images instead of bullethell!

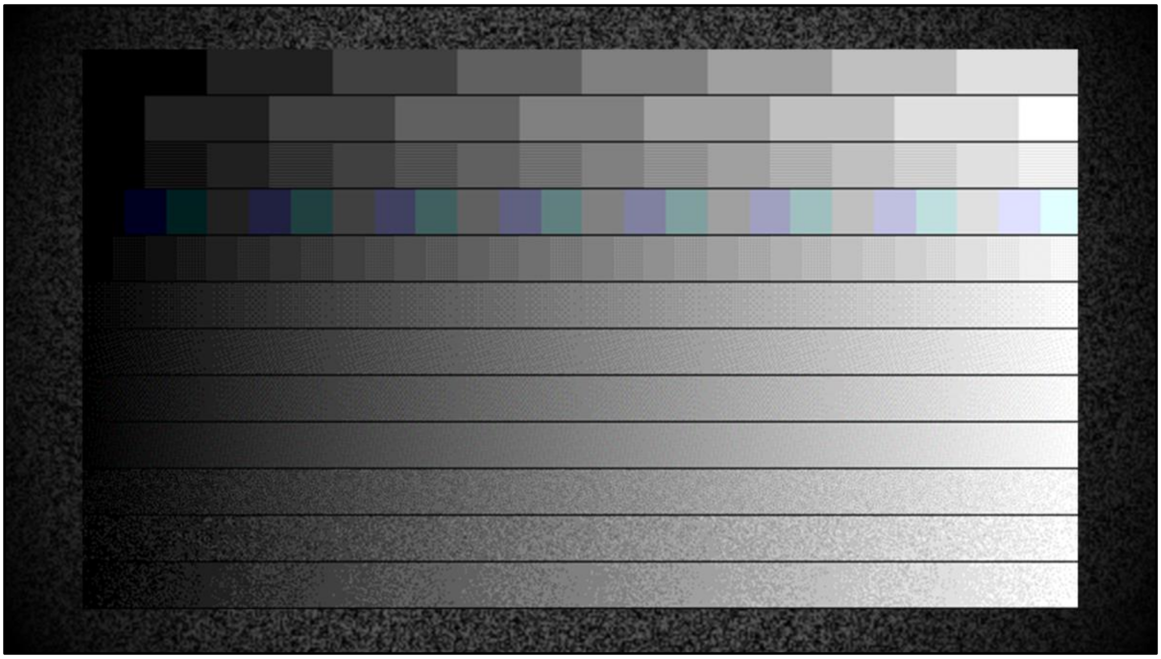
Color Banding - how… err, many noise?

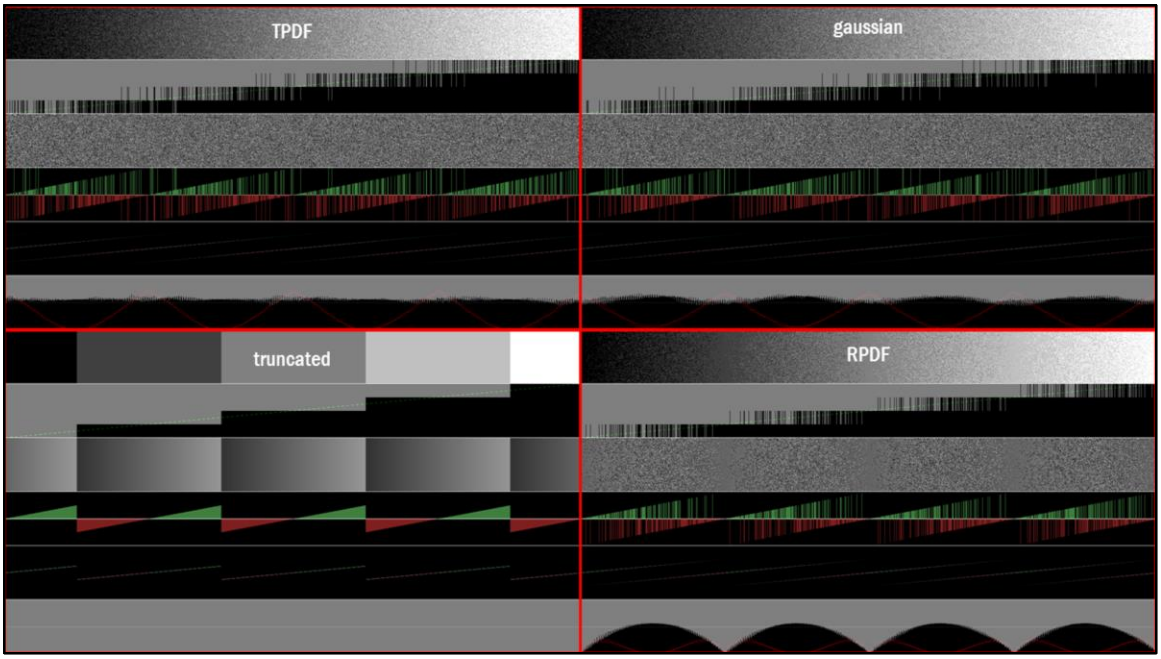
chromatic or monochrome noise?

- Chromatic has smaller variance
- More gaussian distribution (looking at luminance of RGB noise)
- Chromatic likely causes a color-offset due to perceptually different weight of RGB color-channels
- artistic choice? Our game is almost monochrome, we don't notice.

- sidenote: photographers don't like chroma-noise
  - (possibly because their cameras lowpass-filter chroma)






TPDF
gaussian
truncated
RPDF