

SPECIAL EFFECTS WITH DEPTH

Kuba Cupisz
Ole Ciliox



OVERVIEW

- Special Effects
- About Buffers
- Examples

The talk is about simple and effective techniques, that can be quickly implemented based on the data (easily) available to Unity developers.

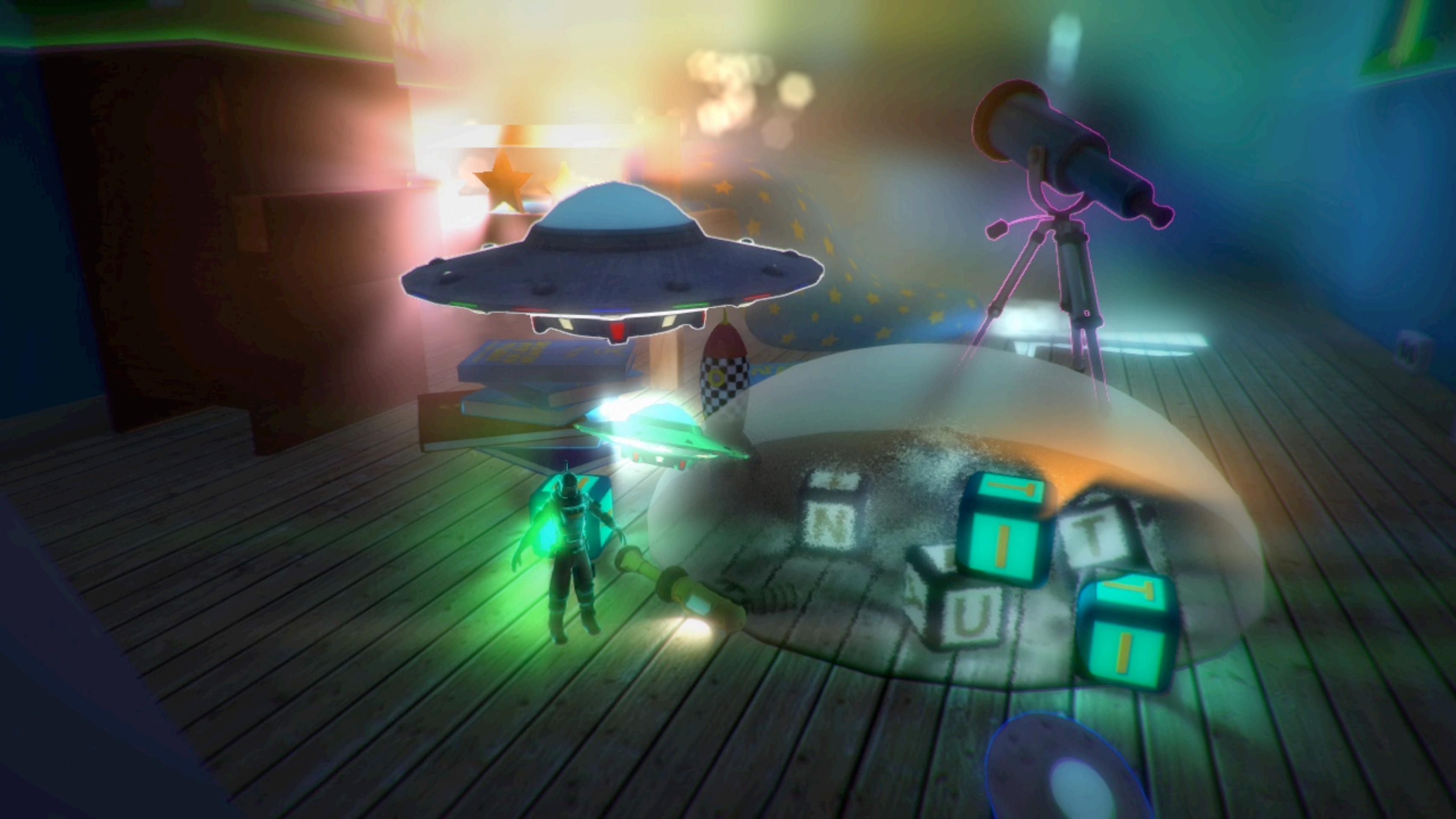
SPECIAL EFFECTS



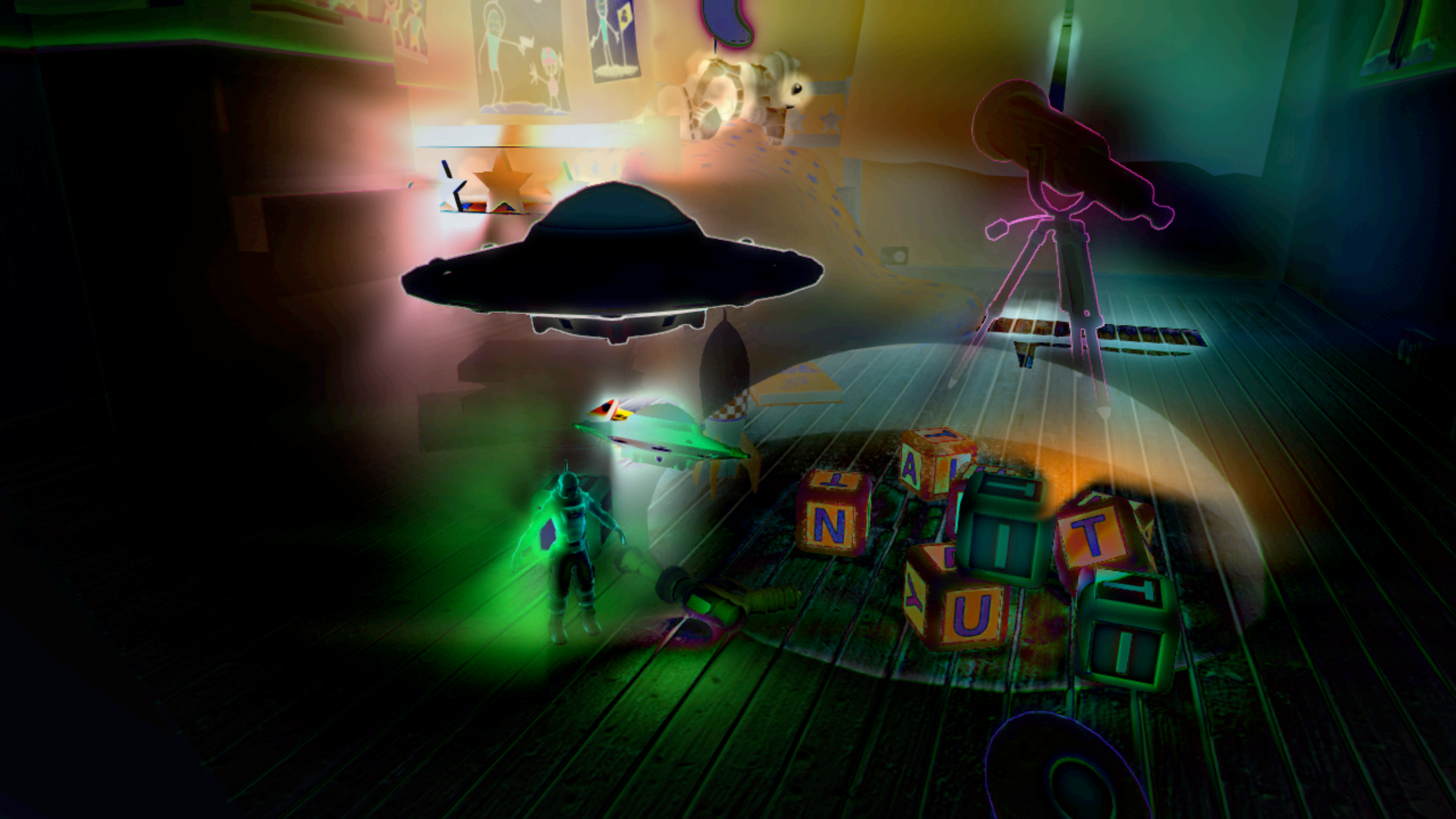
What are special effects?
Why are some effects they "special"?
How to "enable" them in Unity?



Ok, so here we see a lightmapped scene (running deferred), so let's just define "special effects", for the purpose of this talk, as the result of this image...



... subtracted from this image. This results in...



... this. As you can see, we're mostly adding things to the color buffer, so let's just say for now that doing special effects is "adding stuff to the color buffer at some later point in rendering".

TOOLS OF TRADE

- Depending on what you need ...
- ... several options in Unity to *add stuff to the screen*
 - (Multi-pass) shaders
 - Surface shaders or oldschool CG vertex/pixel
 - Particle systems
 - Camera.OnRenderImage()
 - Camera.OnPostRender()
 - Graphics.DrawMesh(), DrawMeshNow()
 - also, MonoBehaviour.OnWillRenderObject() for visibility determination

There are several ways to create effects in unity, all depending on the kind of effect you want to get.

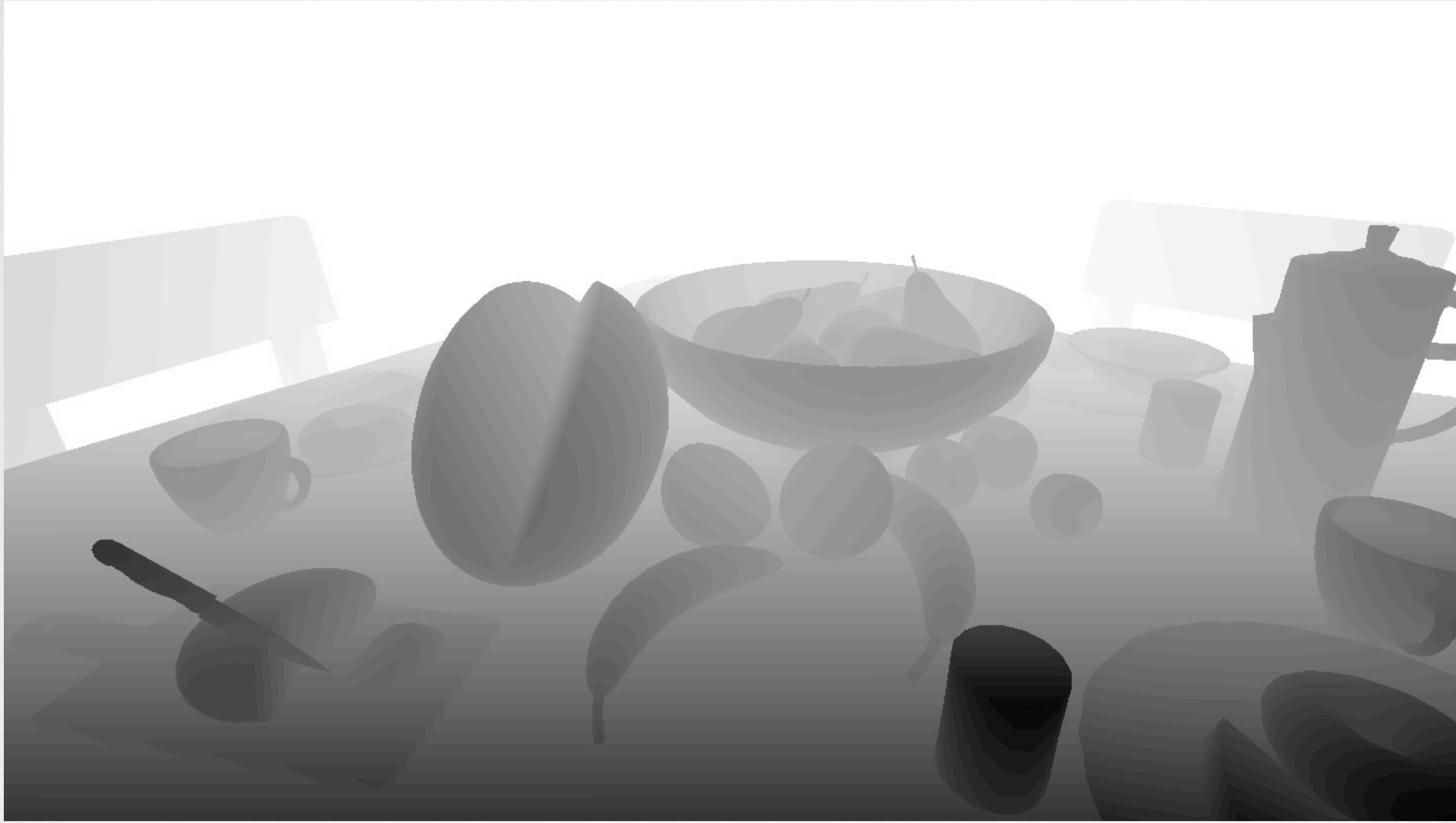
Your tools of trade in Unity for making effects are the following: the multi-pass shader system, along with the OnRenderImage and OnPostRender functions to add stuff to the screen. Furthermore, DrawMeshNow() and DrawMesh() are very useful for drawing selected objects a second time (when shader passes won't be enough), also OnWillRenderObject is good to find out if an object is seen by a camera or not.

ABOUT BUFFERS



Let's talk about various buffers that are available in Unity, which are either part of the rendering or can be easily created.

DEPTH



DEPTH

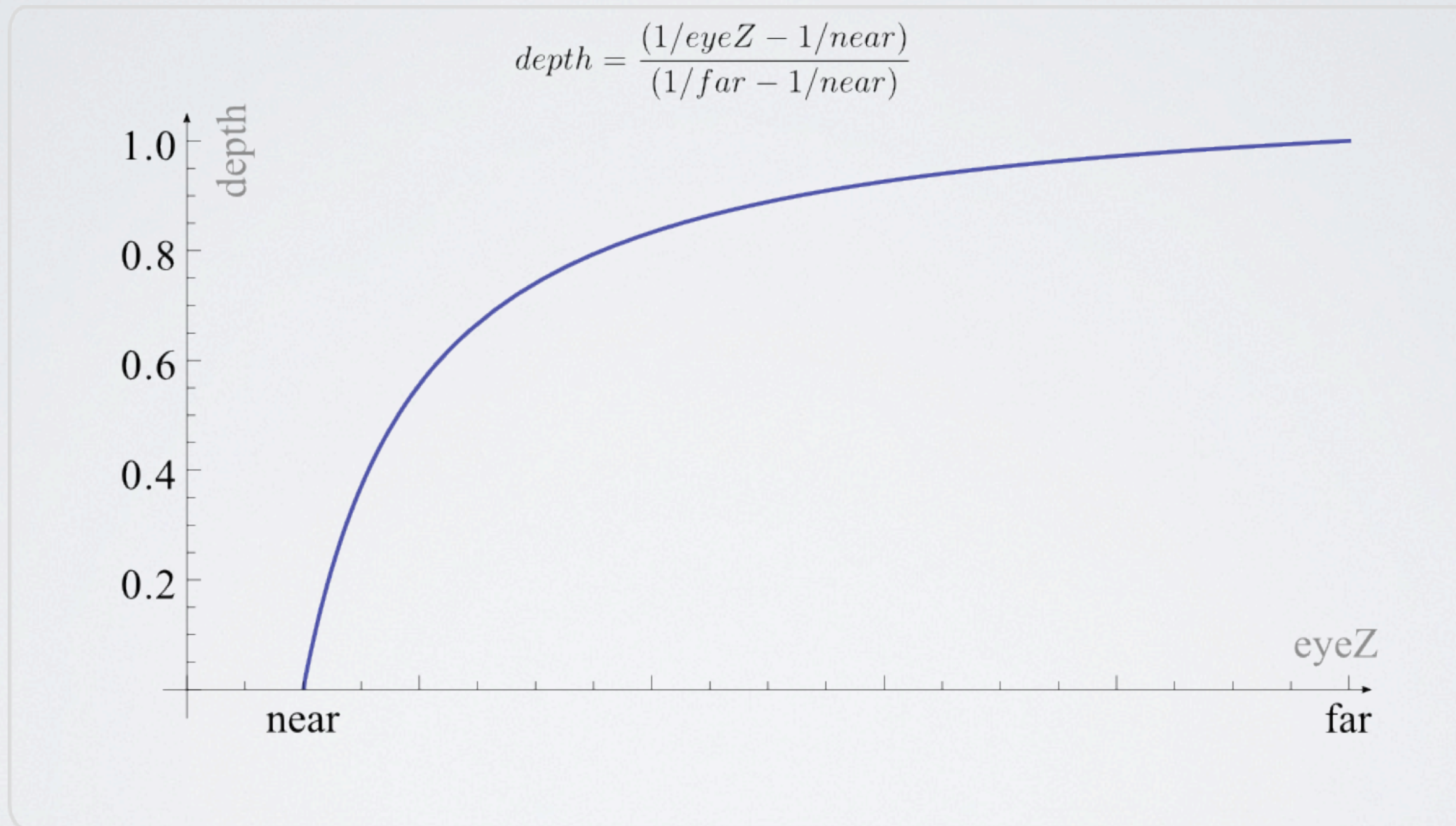
- Distance from the camera
 - In $[0, 1]$ range
 - Nonlinear distribution
 - Precision: 24 or 16 bits
-
- How to get it?
 - Just ask for it:
 - `camera.depthTextureMode = DepthTextureMode.Depth;`
 - it's just there when running deferred

The depth buffer is storing the distance from camera's near clip plane. The values are in the $[0;1]$ range with nonlinear distribution. The precision is usually 24 or 16 bits.

How do you get it? Just ask for it by setting `camera.depthTextureMode` to `Depth`. Then the depth buffer will be available as `_CameraDepthTexture`.

Unity is hiding the burden of getting the depth buffer from you. We are binding the buffer when possible or rendering the scene with a replacement shader when it's not.

NONLINEAR DISTRIBUTION



Where does the nonlinearity come from?

The input to the vertex shader is the object space position of the vertex (`v.vertex`, POSITION semantic). The output of the vertex shader should be the position of the vertex in clip space. This is the space where things outside of the viewing frustum get clipped.

Typically you multiply `v.vertex` by the model-view-projection matrix in one go.

Multiplying by the model matrix places the object in the world. Multiplying by the view matrix makes the position relative to the camera. Multiplying by the projection matrix projects the vertex on the near clip plane, resulting in a value in clip space.

SAMPLING

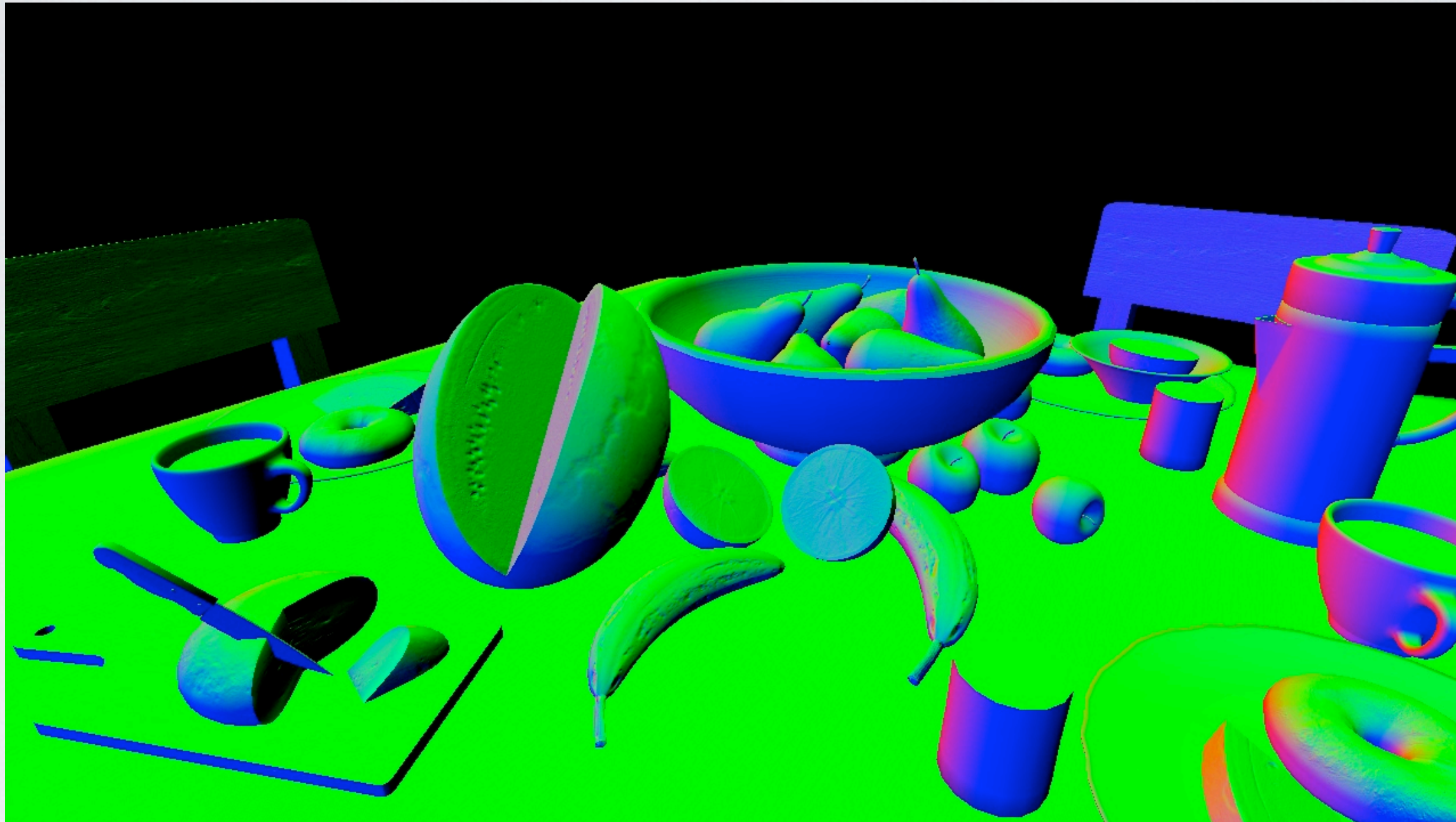
- `float depth = UNITY_SAMPLE_DEPTH(tex2D(_CameraDepthTexture, uv));`
- Linearize when needed
 - distance from the eye in world units
 - `float linearEyeDepth = LinearEyeDepth(depth);`
 - distance from the eye in $[0;1]$
 - `float linear01Depth = Linear01Depth(depth);`

Just use the `UNITY_SAMPLE_DEPTH` macro on the value coming from `_CameraDepthTexture`.

If you need a value in linear space, use the built-in functions.

`LinearEyeDepth` will get you the distance from the eye in world units. `Linear01Depth` will give the distance from the eye in $[0;1]$ range.

NORMALS



NORMALS

- Only accessible in deferred lighting
- .xyz, world space normals
- .w, shininess (0.03, 1.0)
- Sampling
 - `float3 worldNormal = tex2D(_CameraNormalsTexture, uv) * 2.0 - 1.0;`

The normals buffer is a natural product of the deferred lighting pipeline. It stores world space normals in .xyz components. In the .w component it stores shininess.

Just sample it and move back into the [-1;1] range.

DEPTH AND NORMALS

- Almost free in deferred
- In forward the scene is rendered a second time
- Sampling
 - Depth in .zw via `DecodeFloatRG()`
 - Normal in .xy via `DecodeViewNormalStereo()`
 - Both via `DecodeDepthNormals(sample, linear0 | Depth, viewNormal)`
- How to get it?
 - `camera.depthTextureMode = DepthTextureMode.DepthNormals;`

There are cases where you would like to access both the depth and the normals buffer. That's where `DepthTextureMode.DepthNormals` comes into play.

It gives you a buffer that has view space normals encoded into two channels (.xy) and the depth encoded in .zw.

It comes quite cheaply in deferred, the depth and the normals buffer just need to be combined.

In forward the normals buffer is not created by default, so the scene has to be rendered the second time there.

COLOR



COLOR

- How to get it?
 - request it via GrabPass and access as `_GrabTexture`
 - `OnRenderImage`

And of course you have access to the color buffer as well.

You can request it via `GrabPass` and then access the texture as `_GrabTexture` in the shader.

Or you can use `camera.OnRenderImage` callback. The incoming image is the source render texture. The result should end up in the destination render texture. When there are multiple image filters attached to the camera, they process the image sequentially, by passing first filter's destination as the source to the next filter.

You write to the destination render texture by using `Graphics.Blit` with a specific material.

The source image is then accessible in the shader as `_MainTex`.

FURTHERMORE ...

- You can build any (helper) buffer you want, either:
- Render scene a second time via replacement shaders
 - `camera.RenderWithShader(shader, replacementTag);`
- Create it out of the existing ones
 - `_CameraDepthTexture + _CameraNormalsTexture + ALU => _CameraDepthNormalsTexture`

You are of course not limited to the buffers above. You can render the scene a second time with a replacement shader. You can also combine the existing buffers into new ones. That's how the depth/normals buffer is created.

EXAMPLES



All of the examples we are showing run in the vanilla Unity 3.4 (probably 3.3 etc as well, no voodoo required)

WHAT WE WON'T TALK ABOUT ...

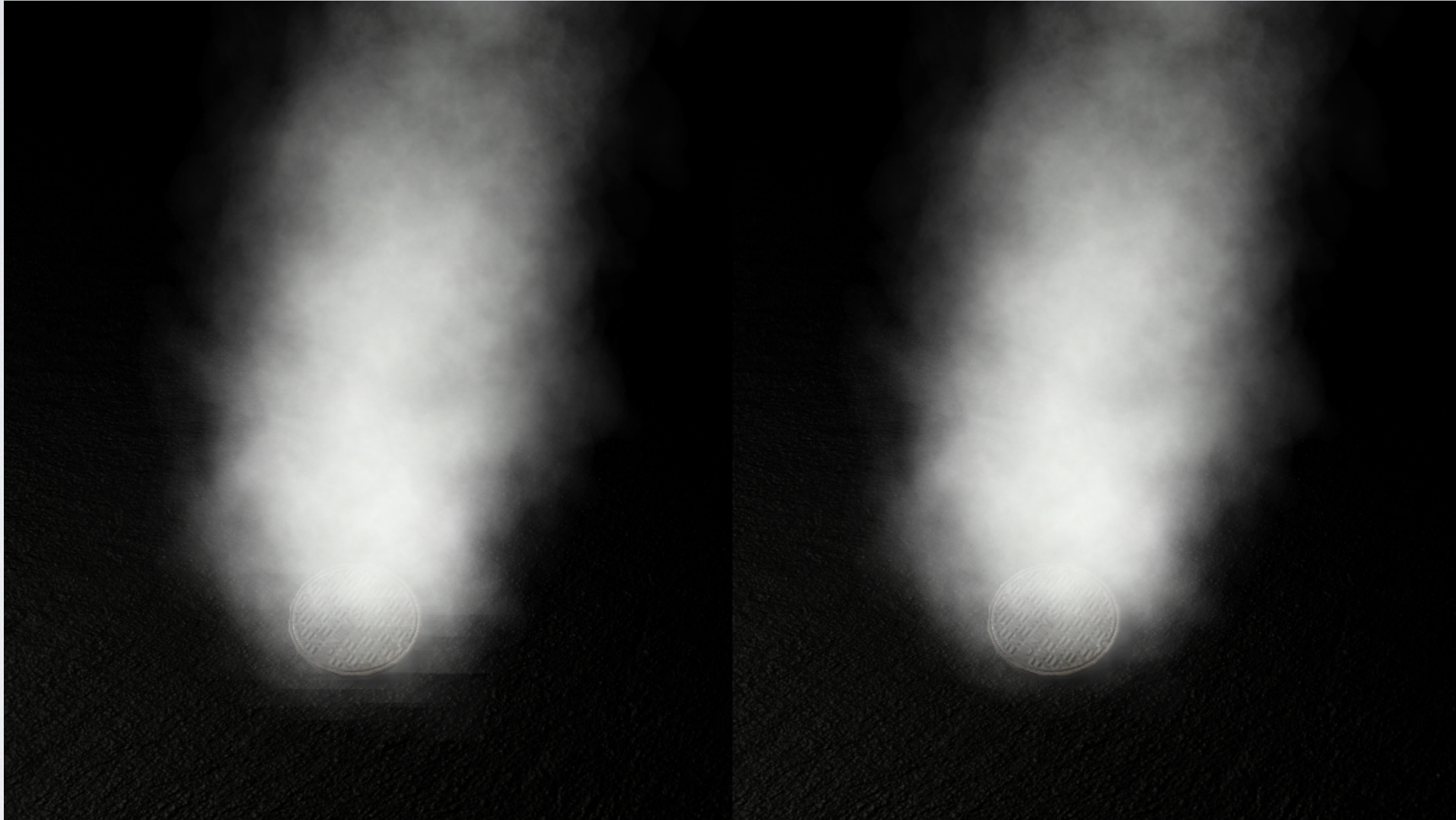


We are not going to show the traditional depth/normals based effects such as SSAO, depth of field, etc. but talk about more unusual scenarios...

WHAT WE WILL TALK ABOUT ...

- Soft particles
- Animated shields
- Intersection highlights
- GPU simulated particles
- Refractive force field effect
- Object outlines
- Vertical, fluffy fog
- Point light shafts

SOFT PARTICLES



Probably a very common technique that is almost impossible without being able to sample camera's depth buffer is **soft particles**,

SOFT PARTICLES

- More of a fix instead of a special effect
- Built-in into Unity's default particle shaders
 - Project Settings > Quality > Soft Particles
- Based on comparing z values in view space
 - $\text{float softFactor} = \text{saturate}((\text{depthEye} - \text{zEye}) * \text{fade})$
- Alpha blend using the soft factor

In Unity, this is built in and all built-in particle shaders support it.

It works by simply comparing depth values of the particle with depth values of world geometry (in view space).

This results in just an additional **blend factor** to smoothen particle edges where they intersect with the world.

ANIMATED SHIELDS

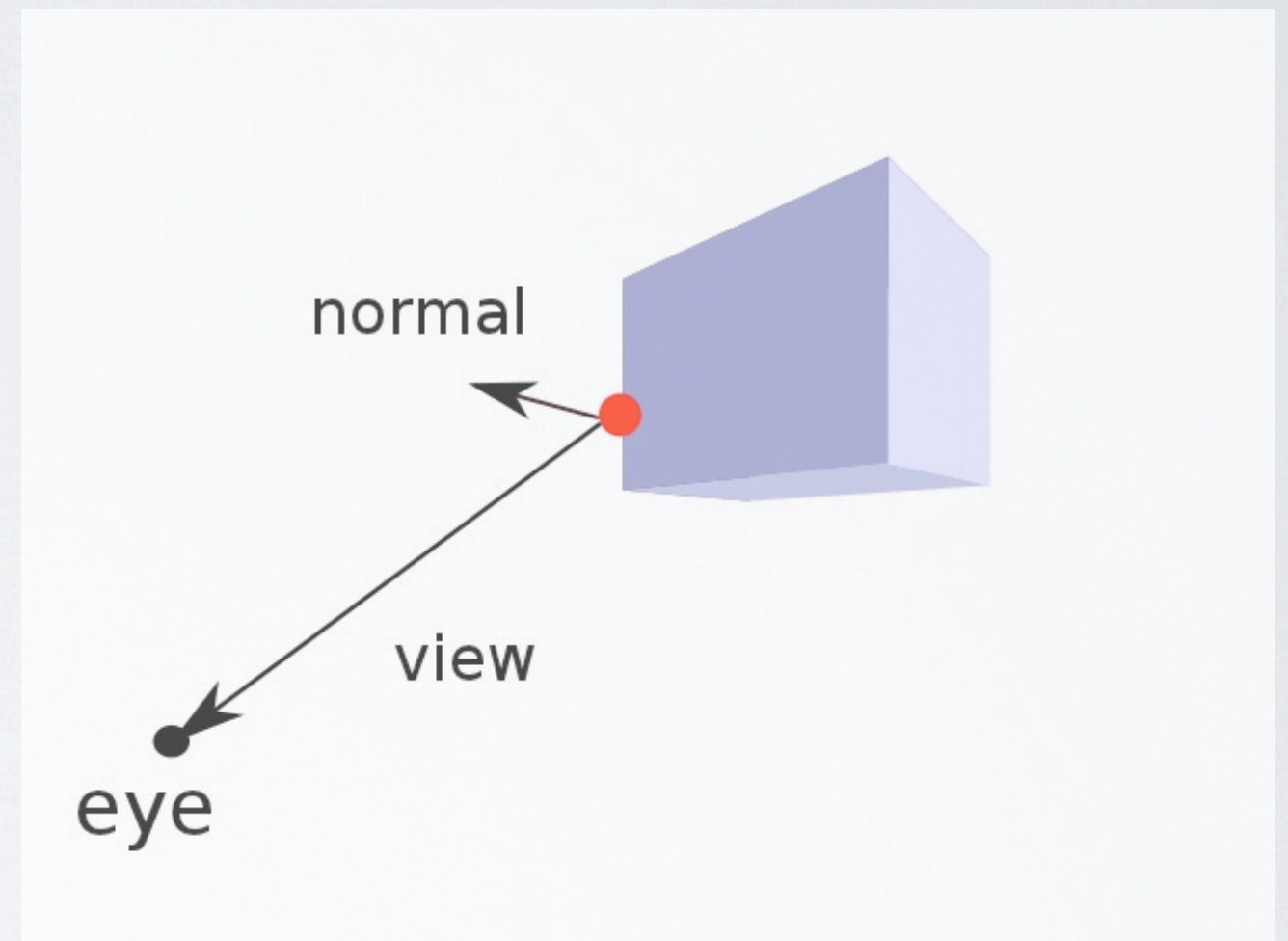


A more effect oriented extension of soft particles is a simple animated shield effect (used e.g. in Halo Reach).

It doesn't look like particles at the moment, but here is what really happens...

ANIMATED SHIELDS

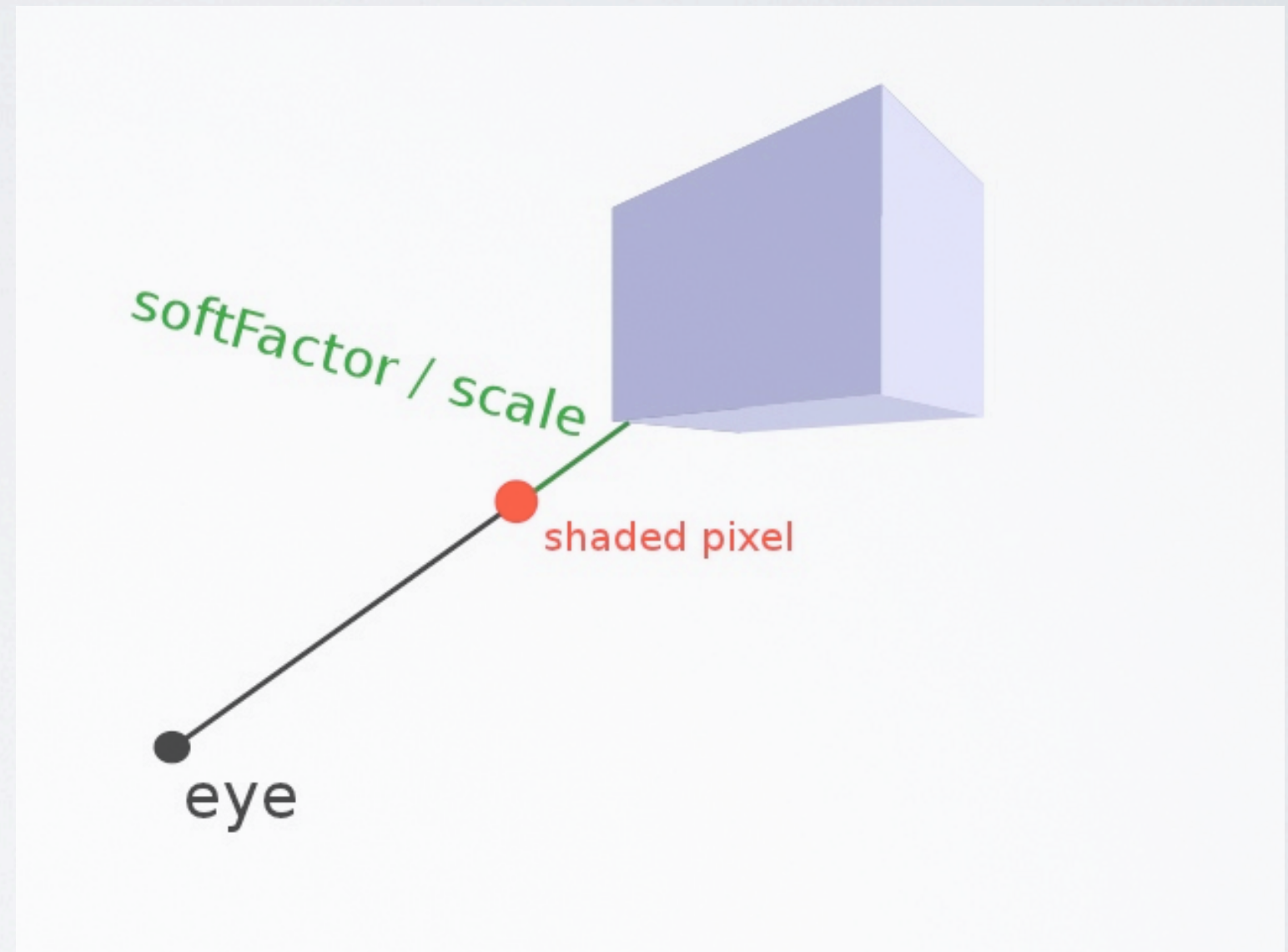
- Extrude vertices along normals
- 2 fadeout factors
 - Soft factor
 - Glancing factor
 - $\text{glancing}(\text{dot}(\text{view}, \text{normal}))$
- Animate between 2 parameter sets



... The object is rendered a 2nd time and all vertices are extruded along their normals. This new geometry is faded out based on 2 factors: the soft particle factor (or just soft factor) and a glancing factor which is a function of the view vector and the normal. Having 2 parameter sets and interpolating between them creates nice shield animations that don't hide too much of the character itself (read: prevent artist rage).

INTERSECTION HIGHLIGHTS

- Interpretation of the softFactor
- So what if ... $1.0 - \text{softFactor}$?
 - Intersection highlights!



So what we just called a “softness factor” in the soft particles fix offers a nice piece of spatial information:

The distance along a view ray between the currently shaded pixel and the intersection with world geometry.

So, what if we use ONE MINUS that factor? This results in intersection highlights with world geometry.

INTERSECTION HIGHLIGHTS



One use case might be some weirdo "scanning effect" that seems like projecting lines on all intersecting objects (could be extended to power or force fields).

CHEAP COLLIDING PARTICLES

- Simulate particles on the GPU
- Base collisions on the information you have available: normals and depth buffers
- Store world space position and velocity in floating point render targets
- Ping-pong between two textures

For some effects simple particle simulation is not enough. You might want the particles to interact with the scene. Evaluating collisions for each particle by casting rays on the CPU might be slow and is dependent on the complexity of scene geometry.

Simulating particles on the GPU comes to the rescue. On the GPU you already have the information about the scene and that is available to you in the depth and normals buffers.

Simulating particles means changing their velocity by a number of factors (acceleration, collision) and then updating the positions based on the position from the last frame and the velocity.

We store the world space position, since it doesn't need to be updated when the camera moves.

Velocity vectors are also in world space, since it's easy to compare them with world space normals.

Main simulation happens on a pair of velocity textures, between which we are ping-ponging. We are reading from the first texture and writing to the second and in the next frame do the opposite - reading from the second and writing to the first one.

REQUIREMENTS

- Vertex textures
- Floating point render targets
- Depth and normals buffers

The current implementation of the technique requires the hardware to support vertex textures and floating point render targets.

Unfortunately Vertex Texture Fetch is not supported across the board in SM 3.0 GPUs, e.g. ATI cards support it only since the HD series.

As mentioned before, just say "gimmeh depth-normals" to get the buffer with depth and normals.

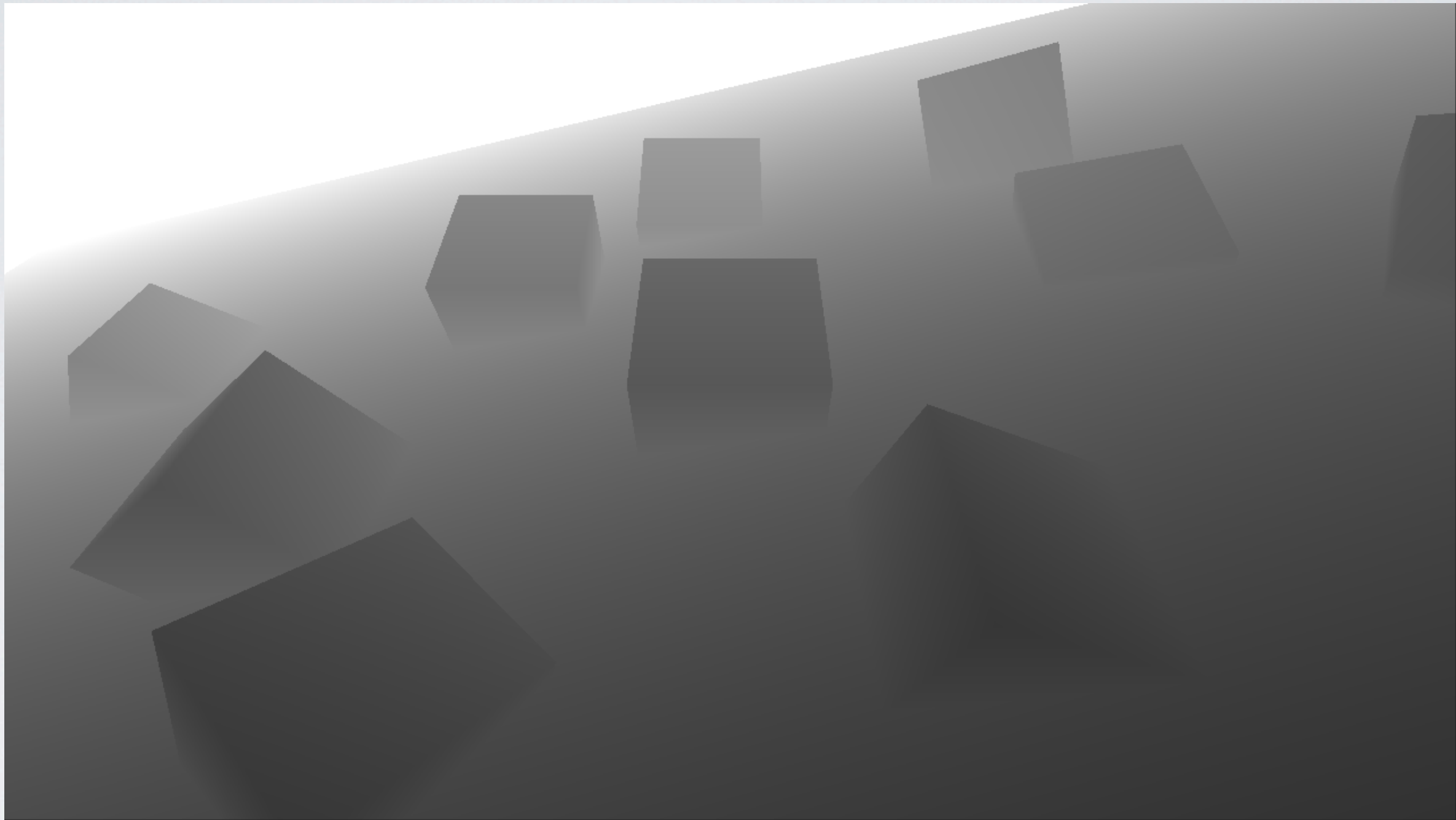
UPDATE VELOCITIES

- read from positions and velocities textures
- change of velocity:
 - increase: $\text{worldVel.xyz} = \text{worldVel.xyz} + _Acceleration * _DeltaTime;$
 - reflect around normal and decrease on collision
 - zero it out when the square magnitude is smaller than settle epsilon

So the bulk of the simulation happens in one pixel shader that reads from a velocity texture and writes to a second one.

The velocity is changed in 3 ways. It is increased by the acceleration, e.g. caused by gravity. On collision it is reflected around normal and decreased. When it's square magnitude is small, we just zero the velocity out to make the particle settle.

DETERMINING COLLISIONS

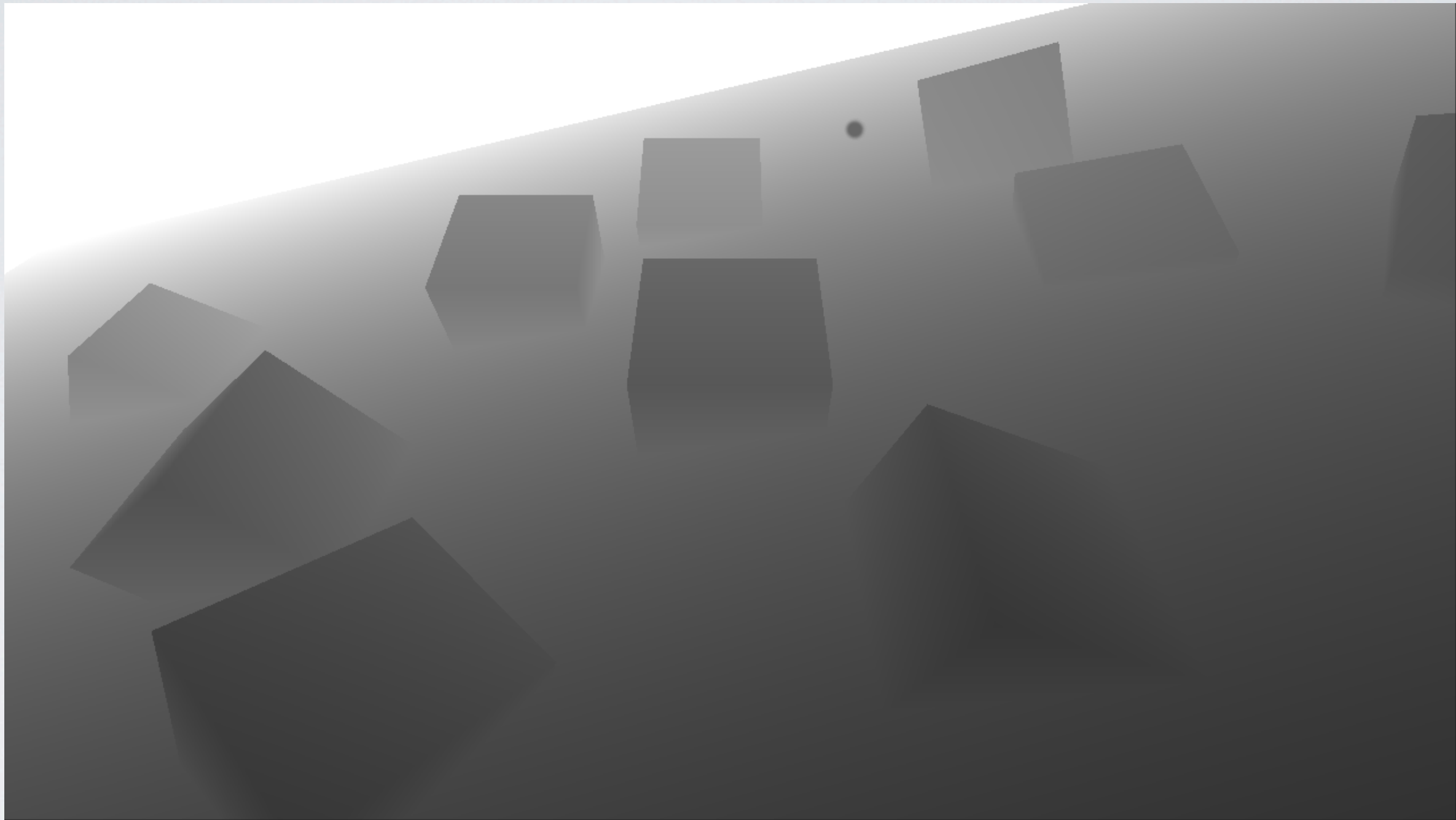


Use linearEyeDepth

So how do we determine collisions?

First we take the values from the depth buffer and transform with LinearEyeDepth.

DETERMINING COLLISIONS

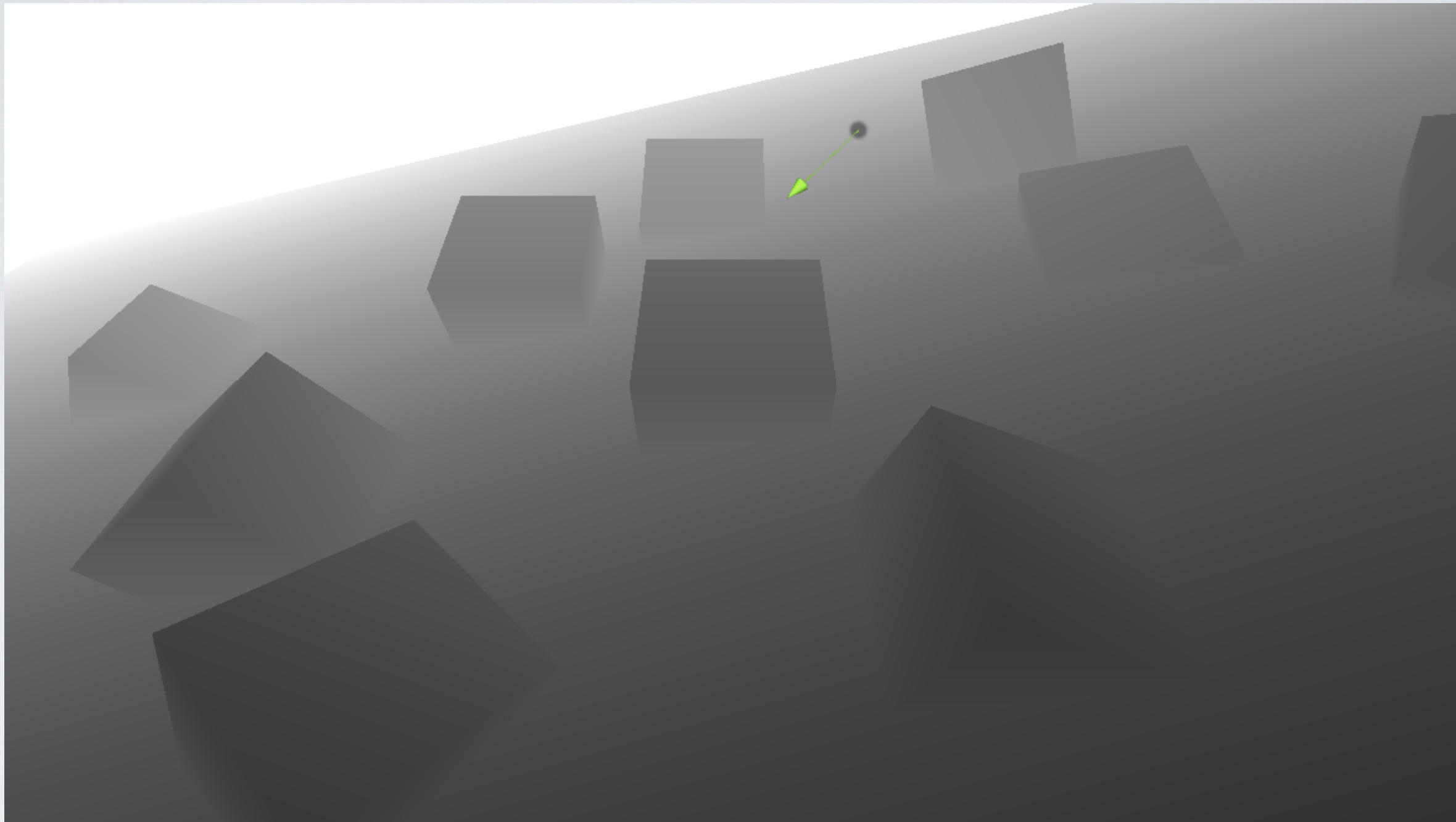


```
float4 clipPos = mul(_ViewProjectionMatrix, worldPos);  
float eyeZ = clipPos.w;
```

We read the world space position from the positions texture and multiply it by the view-projection matrix to get the clip space position.

Then we use clipPos.w as eyeZ which we can directly compare with linearEyeDepth from the depth buffer.

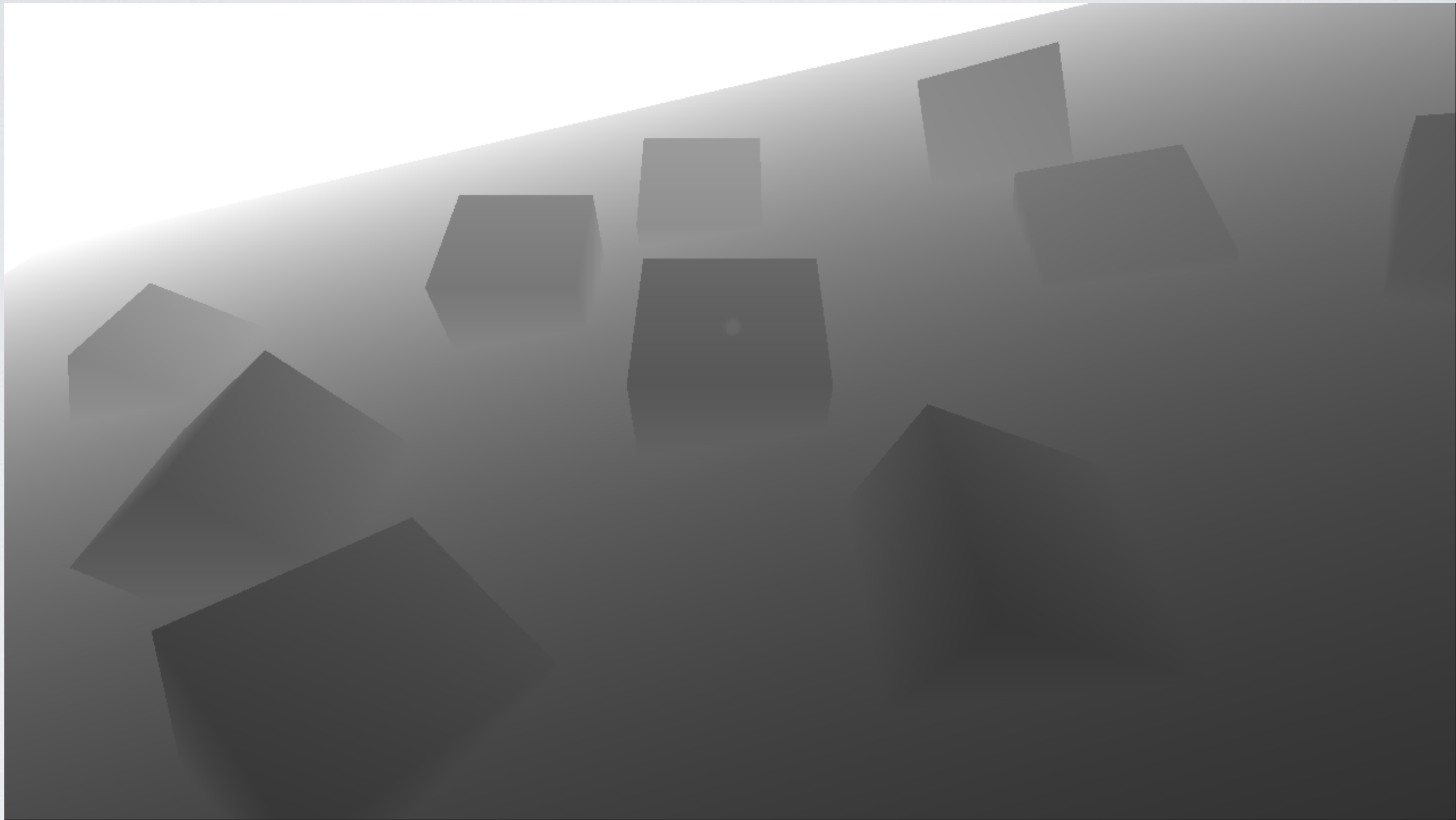
DETERMINING COLLISIONS



Big depth difference - no collision

There is a big difference in depth values, so no collision. The velocity vector does not matter here.

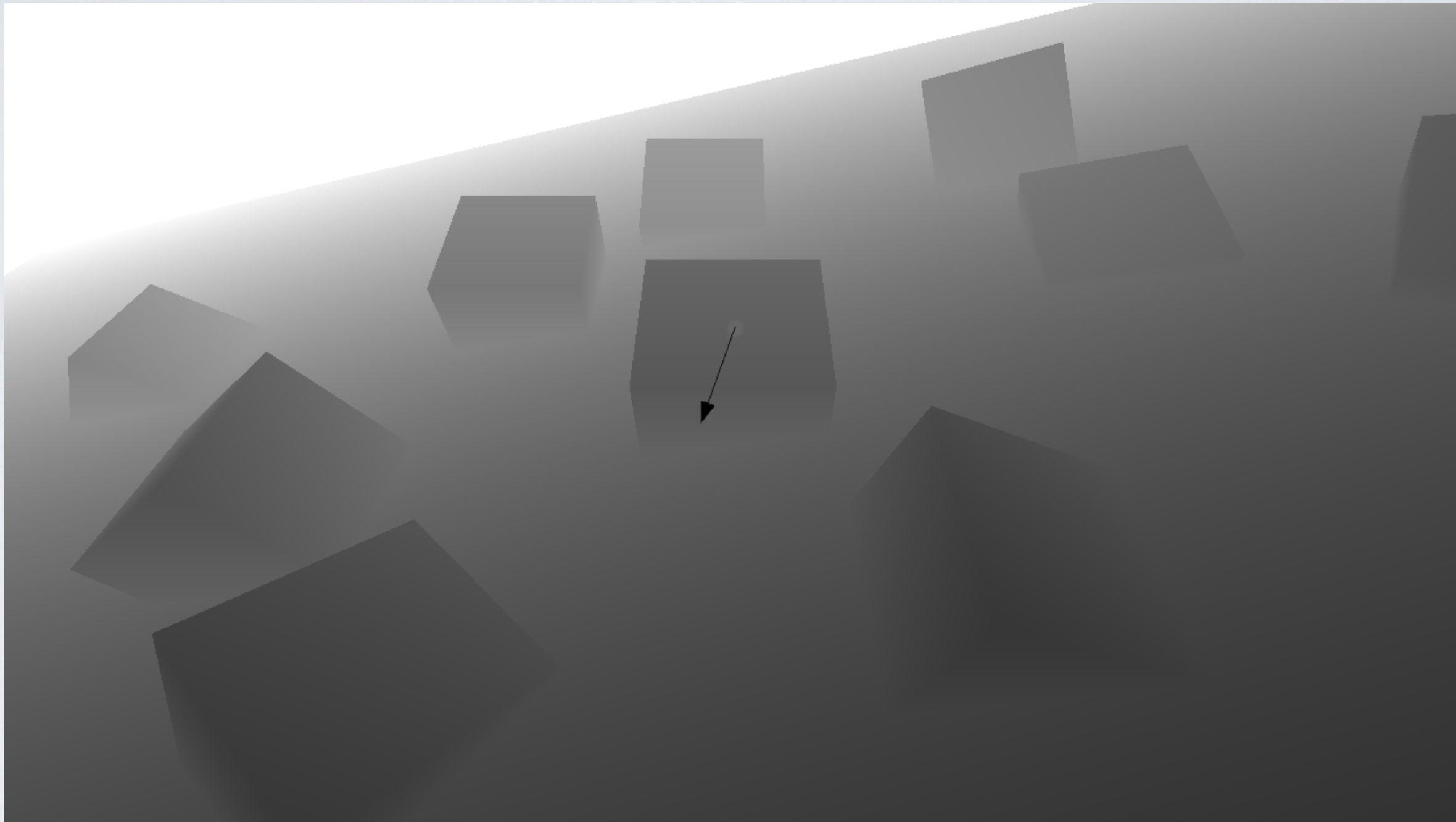
DETERMINING COLLISIONS



Small difference in depth - possible collision

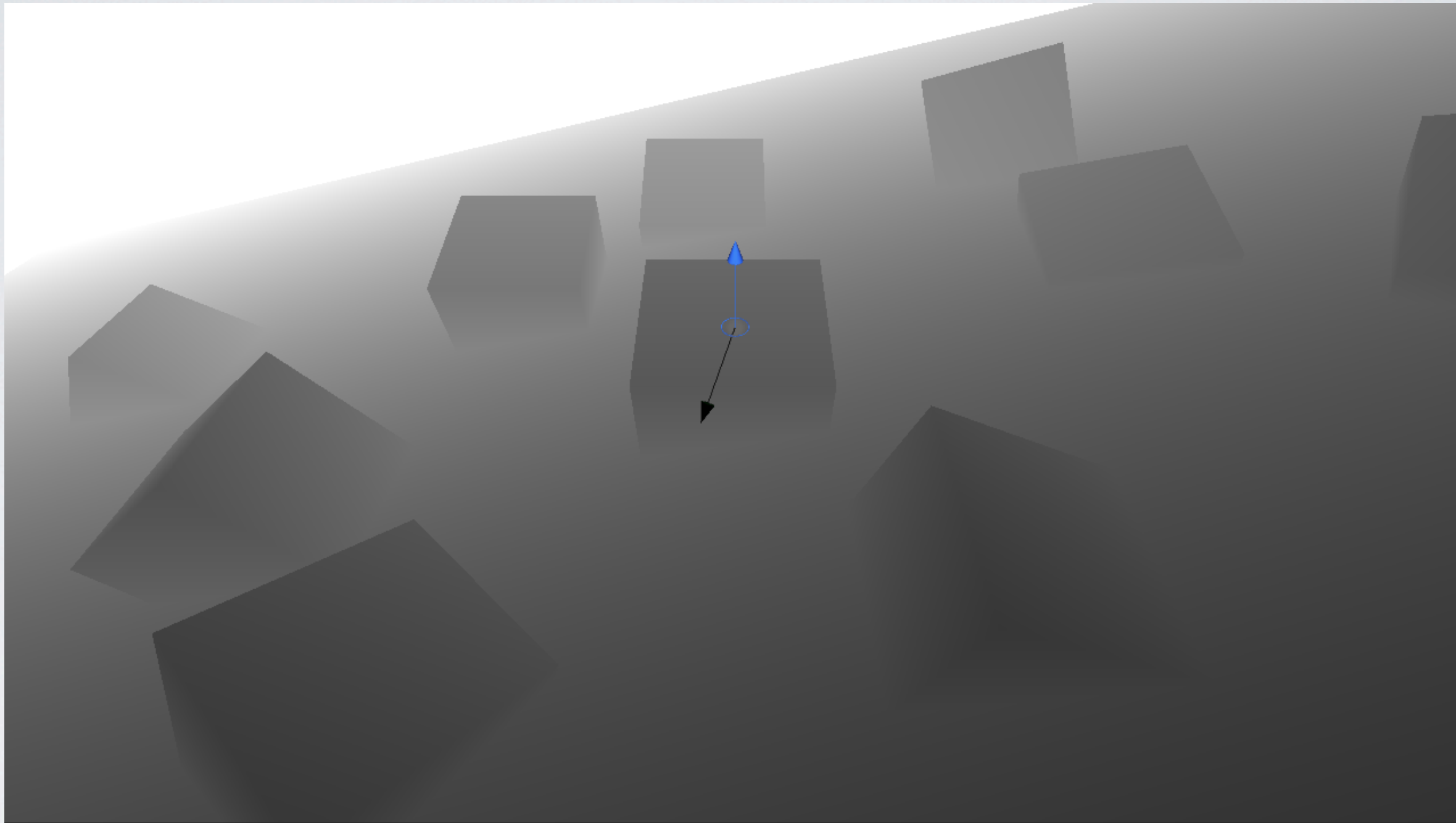
For this particle the difference in depth is small, so we possibly have a collision.

DETERMINING COLLISIONS



Let's add the velocity vector to the mix...

DETERMINING COLLISIONS



Collision!

... and the normal.

So, again, the difference in depth is small. The velocity vector points into the surface, so we have a collision.

Since the original velocity vector points into the surface, the reflected velocity will point away from it. So the next frame we will not have a collision and the particle will be able to bounce off.

UPDATE VELOCITIES

- Compute the distance to the surface behind the particle
 - `float distance = abs(eyeZ - depth);`
- Check whether the particle is moving away from the surface
 - `float movesAway = dot(worldVel.xyz, worldNormal);`
- Determine the collision:
 - `distance < _CollisionEps && movesAway < 0`

So to sum up, the collision happens when the distance to the surface is small and when the particle is moving into the surface (so the dot product of the velocity vector and the normal is smaller than 0).

UPDATE POSITIONS

- As simple as:
 - $\text{worldPos.xyz} = \text{worldPos.xyz} + \text{worldVel.xyz} * _DeltaTime$

Updating positions is super simple, just add velocity multiplied by delta time to the previous position.

RENDER

- Sample the positions texture in the vertex shader and do:
 - `mul (_ViewProjectionMatrix, v.vertex + worldPos)`
- Use the alpha channel of simulation textures to store more info
 - set to **1** on collision and decrease over time
 - then use when rendering to lerp to a certain color, change size, etc.
- Rendering the particles is usually an order of magnitude slower than the simulation

Now it's high time to use the simulation data. Just sample the world space position of the particle in the vertex shader add it to `v.vertex` and multiply by the view-projection matrix.

The simulation textures have 4 channels, but so far we have only used 3. Use the alpha channel to store more info. For example, you can set it to 1 on collision and then decrease over time. Then use that when rendering to lerp to some color, change the size, etc.

It's worth noting that rendering the particles is still usually an order of magnitude slower than the simulation step. So even though you can simulate a shit-ton of particles, it's still the rendering that will hold you back. Oh well...

VERTEX TEXTURE

- Can be cheap or expensive
- Depends on the scene and the hardware
- Might be faster to render the particles with VTF than to render plain particles

So how does the vertex texture fetch fit into the performance picture? It depends. VTF can be cheap or expensive. It depends on your scene. It depends on the specifics of the graphics hardware.

in our test scene it was **faster** to render the particles that were sampling the positions texture than to render plain particles.

SCALABILITY OF THE SIMULATION

- nonlinear with the number of particles:
- 24 μs^* for 16,000 particles (127x127 pixels simulation texture)
- 35 μs^* for 100,000 particles (316x316 pixels simulation texture)

* measured on ATI Radeon HD 5800

The simulation scales nonlinearly with the number of particles. This is very much dependent on the hardware.

On an ATI Radeon HD 5800 card it took 24 microseconds for 16,000 particles, whereas simulating 6 times the amount of particles took just 1.5 times more.

PRECISION

- Using `_CameraDepthTexture` gives inconsistent results across machines
- Use `_CameraDepthNormalsTexture` instead
 - stores values quantized even further by `EncodeFloatRG()`

Simulations can be heavily influenced by the precision of the input data. Sometimes less is more. ;> It might be better to have data that is stored with less precision, but consistently between the platforms.

Using `_CameraDepthTexture` gives inconsistent results in the simulation across different machines. We use `_CameraDepthNormalsTexture` instead, since it stores values that are quantized even further and those are more hardware agnostic.

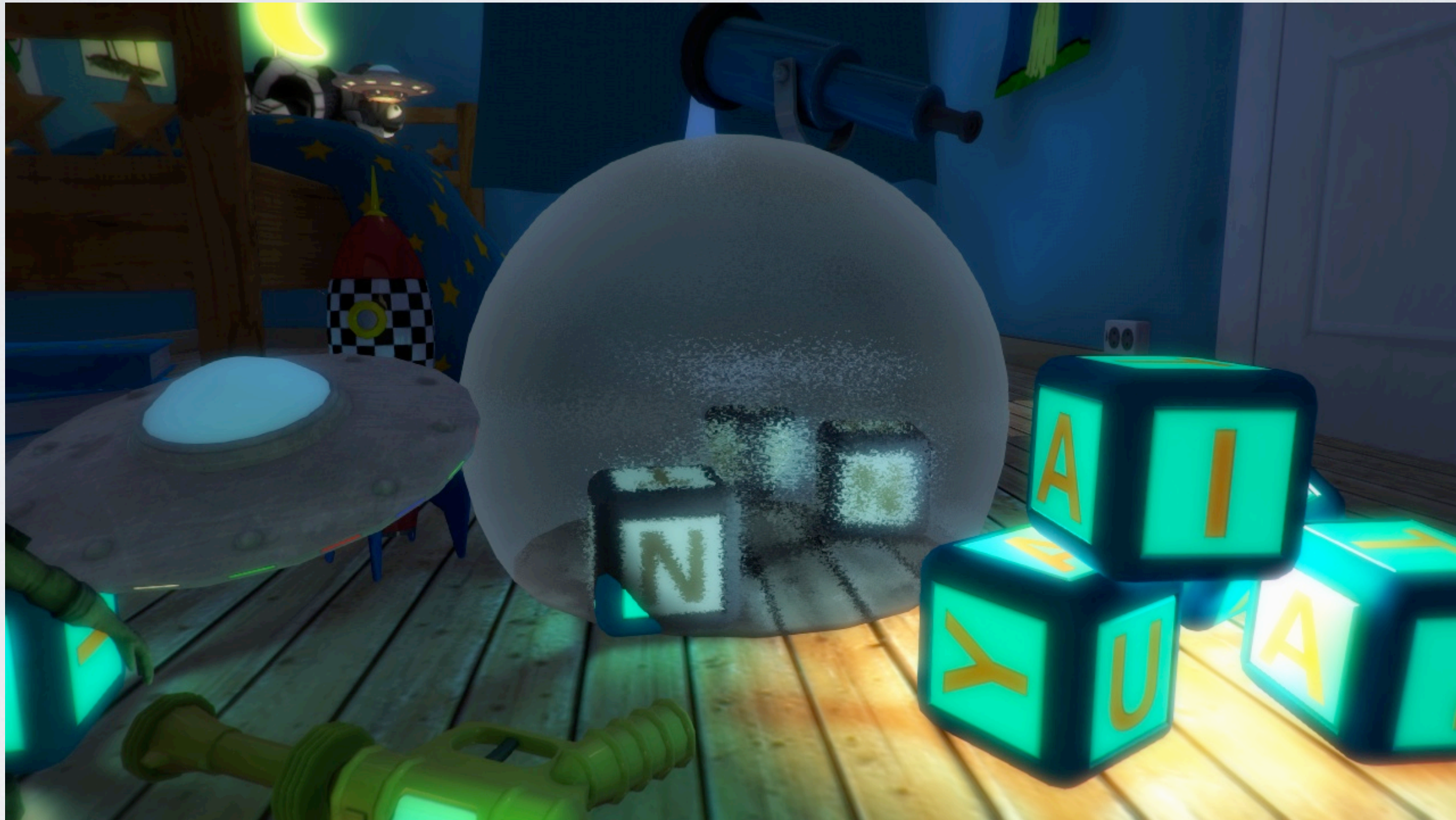
TEXTURE SAMPLING VS ALU

- It might be faster to sample `_CameraDepthNormalsTexture` only and calculate the world space normal from that ...
- ... than to get depth from that texture and the world space normal from `_CameraNormalsTexture`
- 35 vs 39 μ s for 16,000 particles
- When in doubt - profile it!

It might be faster to sample `_CameraDepthNormalsTexture` only and calculate the world space normal from that, than to get depth from that texture and the world space normal from `_CameraNormalsTexture`. For us it was 35 vs 39 μ s for 16,000 particles.

Profile on the target hardware!

FORCE FIELD EFFECT



Talking about science fiction (wait, what?):

Force field effects!

(or, if you remove the animation: frozen glass!)

FORCE FIELD EFFECT

- Want: depth based refraction!
- Access color: Unity's named GrabPass functionality
 - GrabPass { "_Grab" }
 - new in 3.4: only copies the color buffer once per frame (*_Grab* is shared)
- tex2Dproj(_Grab, screenSpaceUVs + offset)
 - use built-in *_Time* shader parameter for simple animations
- Depth based
 - Consider the Soft factor and add blur and refraction
- Bonus
 - Rim shading, darken the back facing tris (alpha blending)

This effect is using both the color and the depth buffer.

In Unity you can sample the color buffer at any point using the GrabPass functionality, which copies a part of the current color buffer into a render texture. Then just offset image space UV's by a noise texture based on the oh-so-well-known soft factor to refract objects that are far away more than the ones close by.

OBJECT OUTLINES



An effect that doesn't use the soft factor, but still relies on depth samples (and maybe normals if you want more outlines) is this simple per-object outlines effect. It can use any feature detection filter of your choice ...

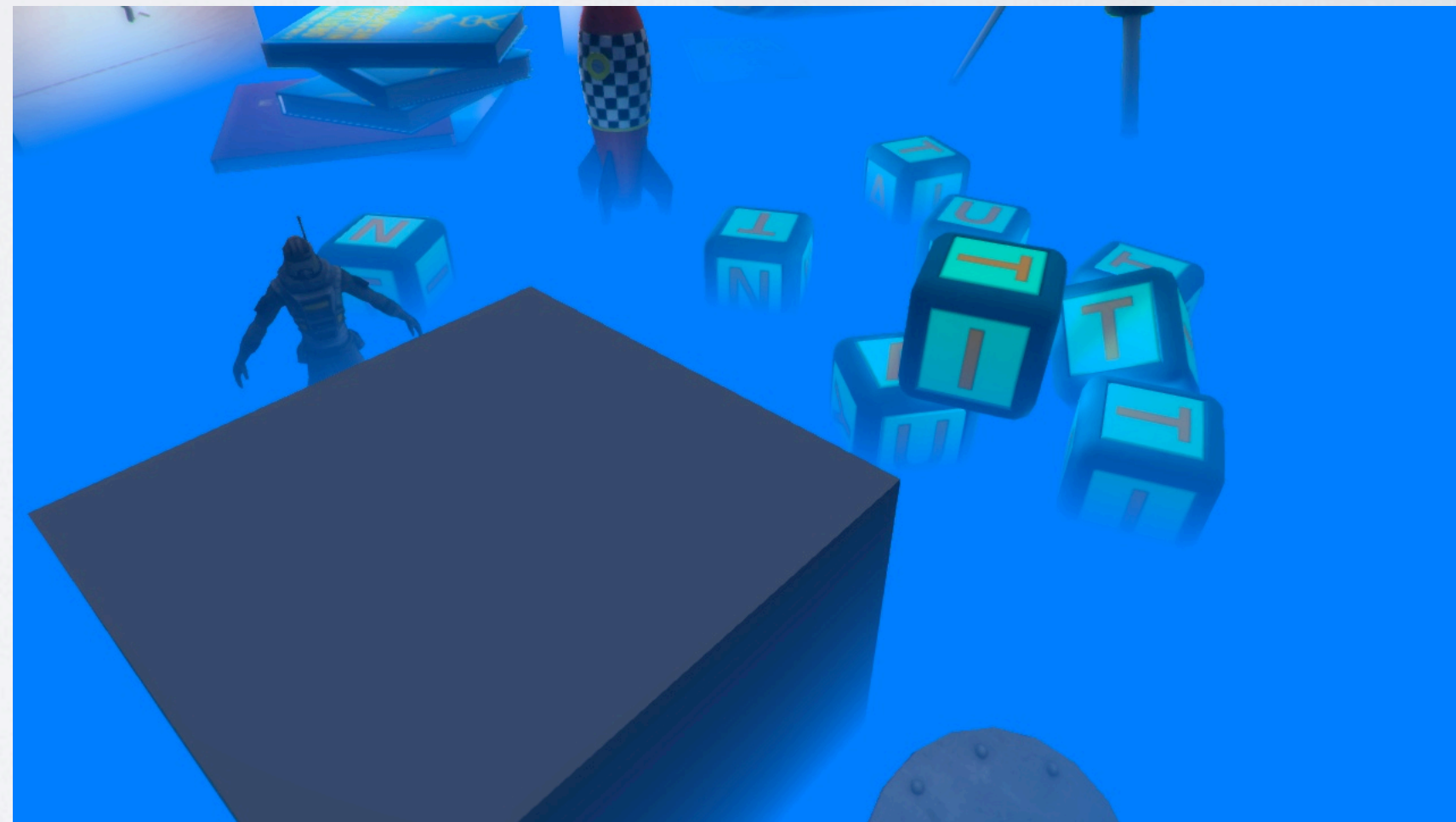
OBJECT OUTLINES

- Edge detection is usually a global image effect which operates on the entire screen
- For per object outlines
 - Render chosen ones a 2nd time after geometry pass
 - `Graphics.DrawMesh()` or `DrawMeshNow()`
 - Sobel filter in pixel shader
 - sample neighbouring depth values in screen space
 - Useful: `_CameraDepthTexture_TexelSize` parameter
 - `clip()` gradient magnitude against a threshold

Edge detection usually comes as a global post effect, so in case of only outlining selected objects, we just render them again after the opaque queue and use for instance a sobel filter with depth and perhaps normals buffers to find the edges we want. One just samples neighbouring pixels (Unity offers help here via the automatically set `_TextureSize` shader parameter to get the right pixel offsets) and calculates the gradient magnitude (which is basically just the change in depth values) to clip() against a threshold.

VERTICAL FOG

- World space reconstruction in shader from depth buffer
 - `float4 wsPos = (_CameraWorld + linearDepth * fromVtx.interpolatedRay);`
 - Much faster than per pixel matrix multiplication
- We get `wsPos.y` for our fog density
 - e.g. `saturate(exp(-wsPos.y - start) * scale)`



And now for something completely different ...

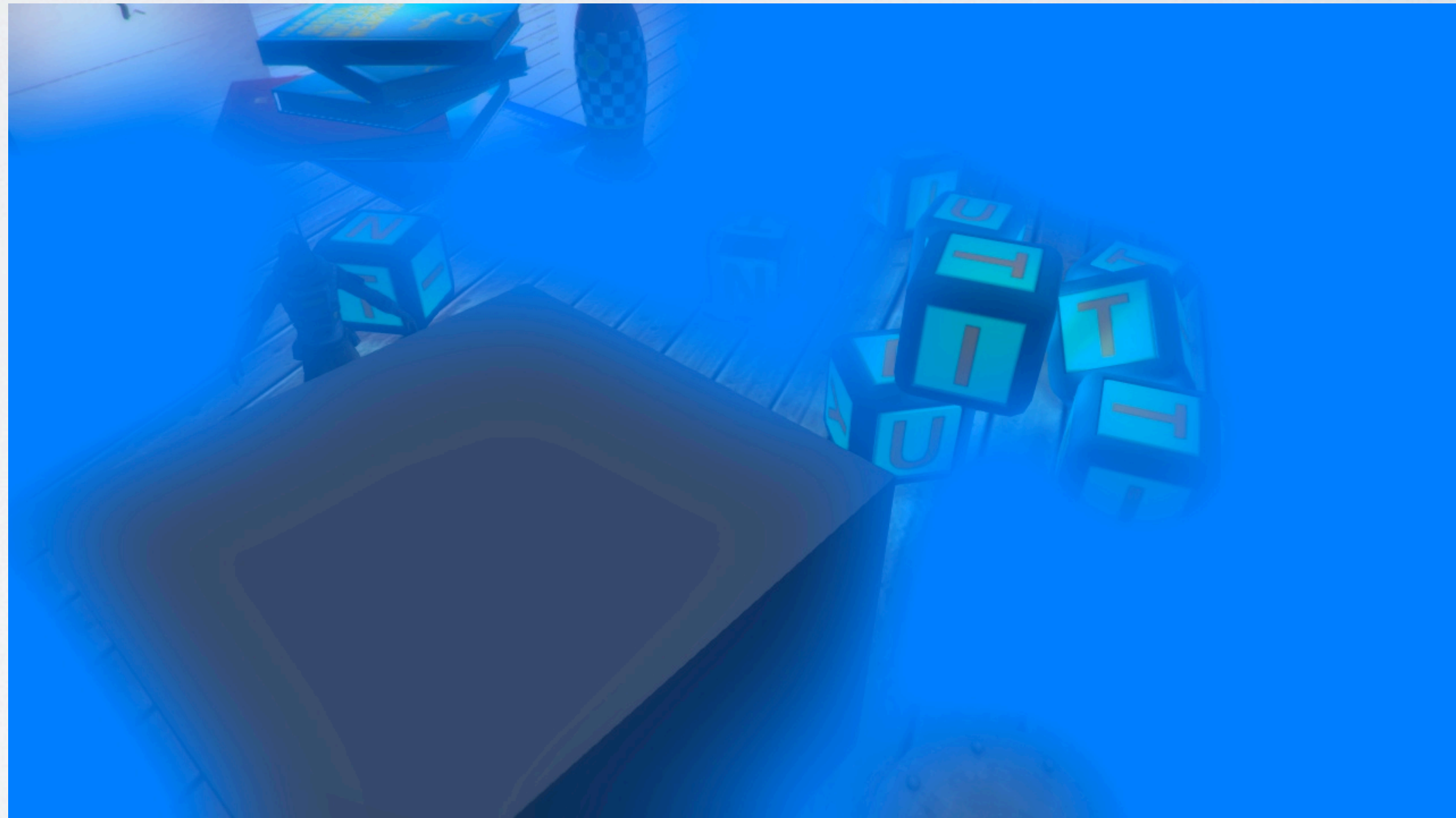
Some effects require you to calculate the world space position of a pixel in the shader, however doing this via per pixel matrix multiplication is very costly, hence the need for a fast world space reconstruction:

one simply interpolates the frustum rays in image space, multiplies that with the linear eye depth value from the depth buffer lookup and adds world space camera position.

Then we have e.g. the `world.y` value available to tint everything based on some simple fog density function.

FLUFFY FOG

- Blur?
 - Put fog factor in $[0,1]$ into RT
 - Separable blur!

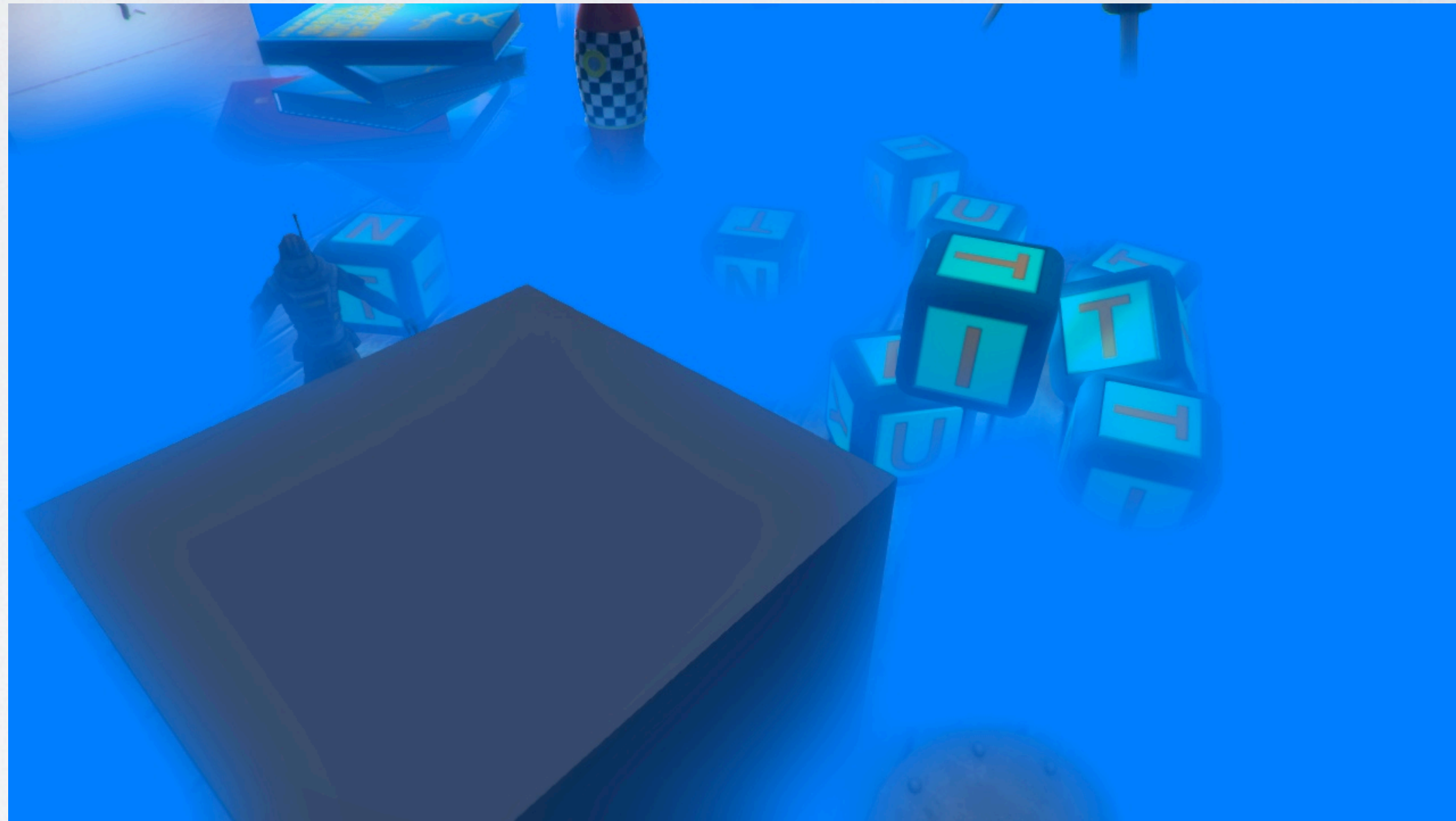


Let's stay with our fog for a little while. Only using one height value per pixel can look a little too harsh to call it a fluffy fog. so why not store fog values in a render texture and blur the beast?

Although it does make it look more fluffy we get bad artifacts with the fog bleeding into areas it shouldn't affect.

FLUFFY FOG

- Bilateral blur to prevent most annoying artifacts
 - Weight options
 - Depth distance
 - Fog factor
 - World space distance



But selectively blurring only certain regions is a common problem. We can easily just weigh the samples during the blur with some cleverly picked value. People are usually doing similar things for calculating defocused depth of field buffers.

We tried several weight factors and found that the distance in world space (which we can recreate quickly, see previous slides) works best.

RESULT



World space distance

Another example with a top-down view.

Notice how the fog gets thinner in "crowded" areas and at the same time grounds small objects.

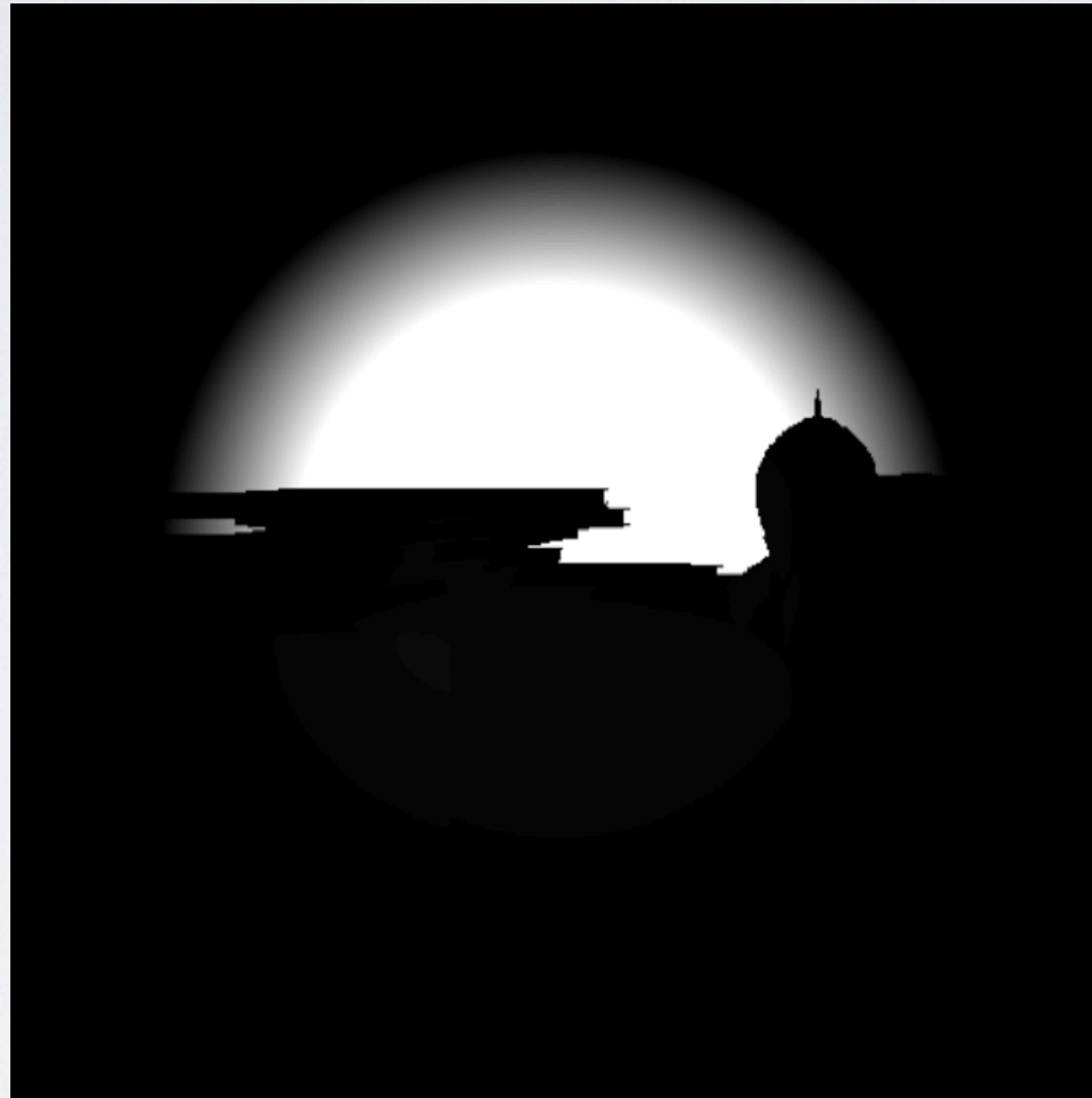
POINT LIGHT SHAFTS



Next up: point light shafts ... or just a little nicer light halos. A simple, cheap and effective fake volumetric technique based on a simple depth-based radial blur.

One example flow of operations could be to ...

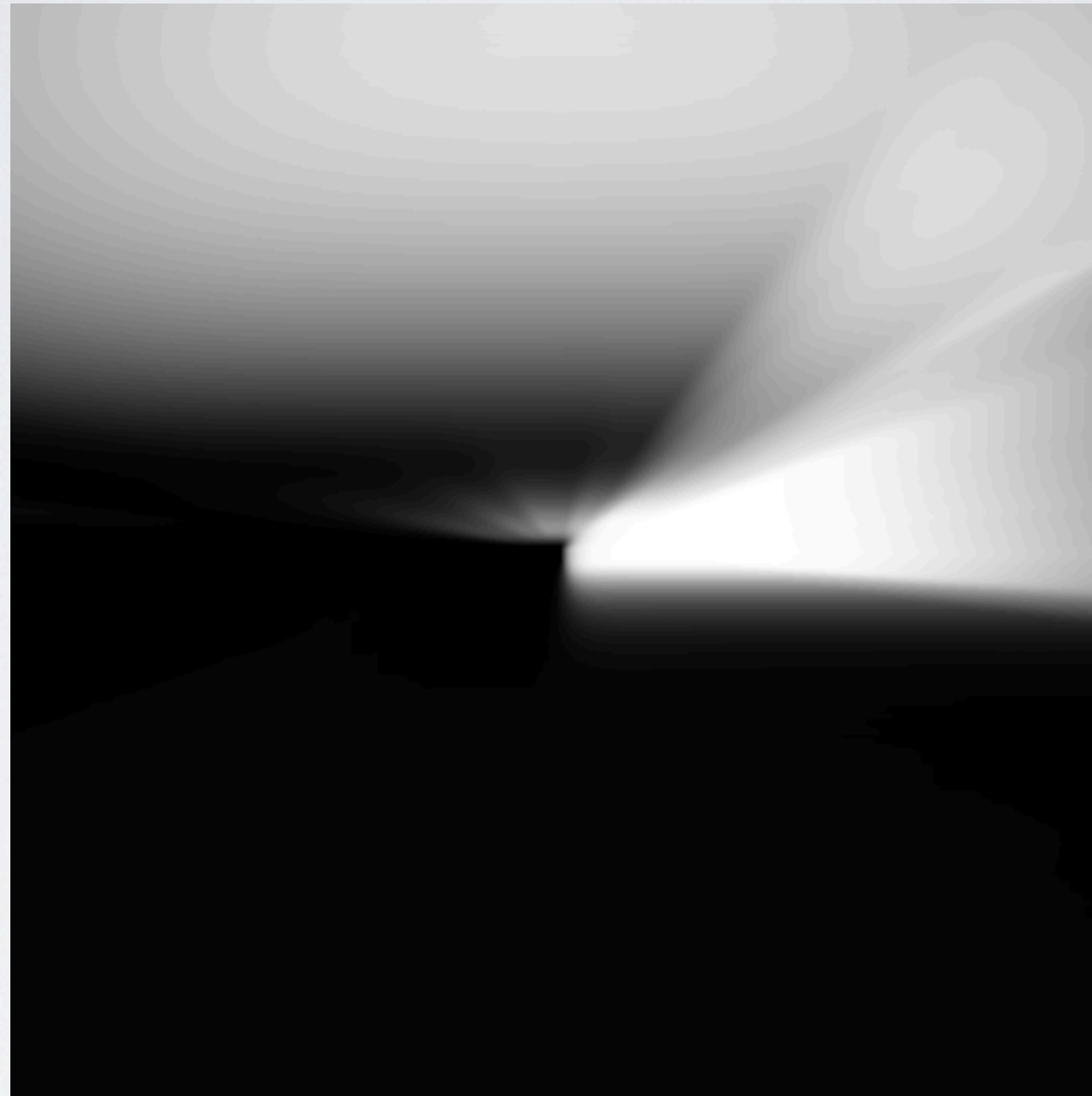
POINT LIGHT SHAFTS



"Stamp" local quad of the depth buffer into a RT

... first stamp the part of the depth buffer you want to blur into a render texture ...

POINT LIGHT SHAFTS



Radial blur

$$\text{blurVector2} = (\text{uv} * 2 - 1) * \text{radius}$$

... then radially blur this texture from the center point n times, doubling the radius with each iteration and then just ...

POINT LIGHT SHAFTS

- Apply (blend masked & blurred "stamps" into the color buffer)
- Possibly sort
- Consult depth buffer to consider max depth radius
 - `color * saturate(linearDepthSample + radius - _Position.z);`



... mask it with a texture, apply a color, and blend it onto the screen.

Two things of interest here:

all point light shafts can be sorted by camera distance (1) and additional depth bounds checking can be done so that objects in front of the shafts source can actually act as occluders (2)

All that can be performed in `OnRenderImage()`, optimally combined with other effects (otherwise, you'll pay for a fullscreen blit).

FIN



Join us on 28th Sept in San Francisco
<http://unity3d.com/unite/>

