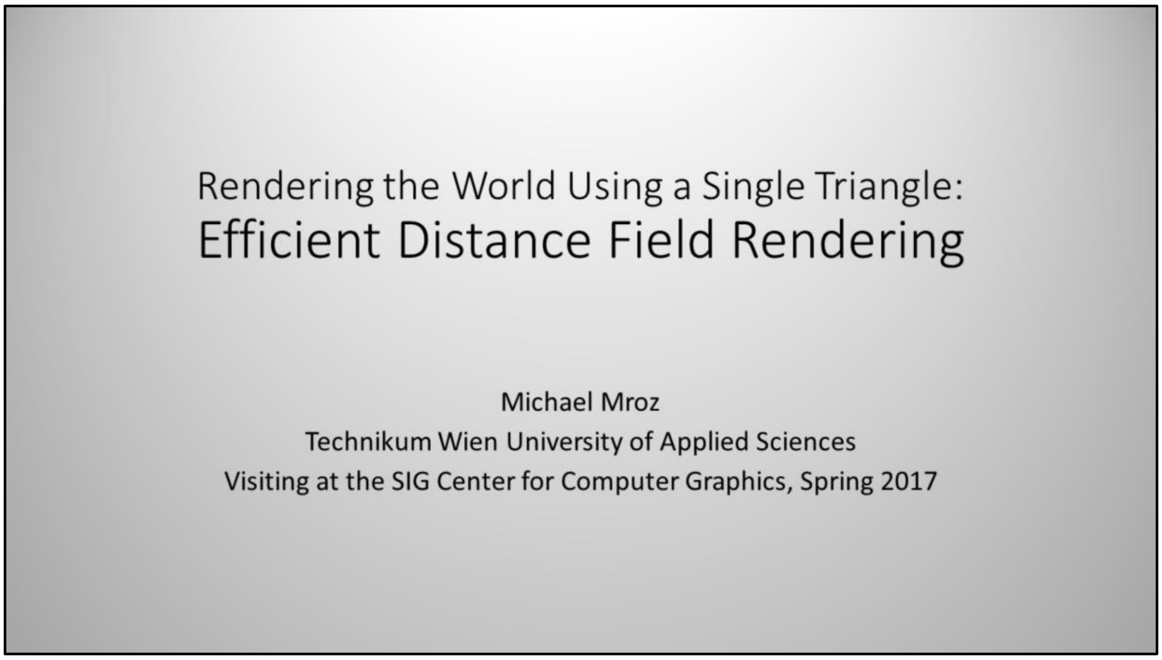
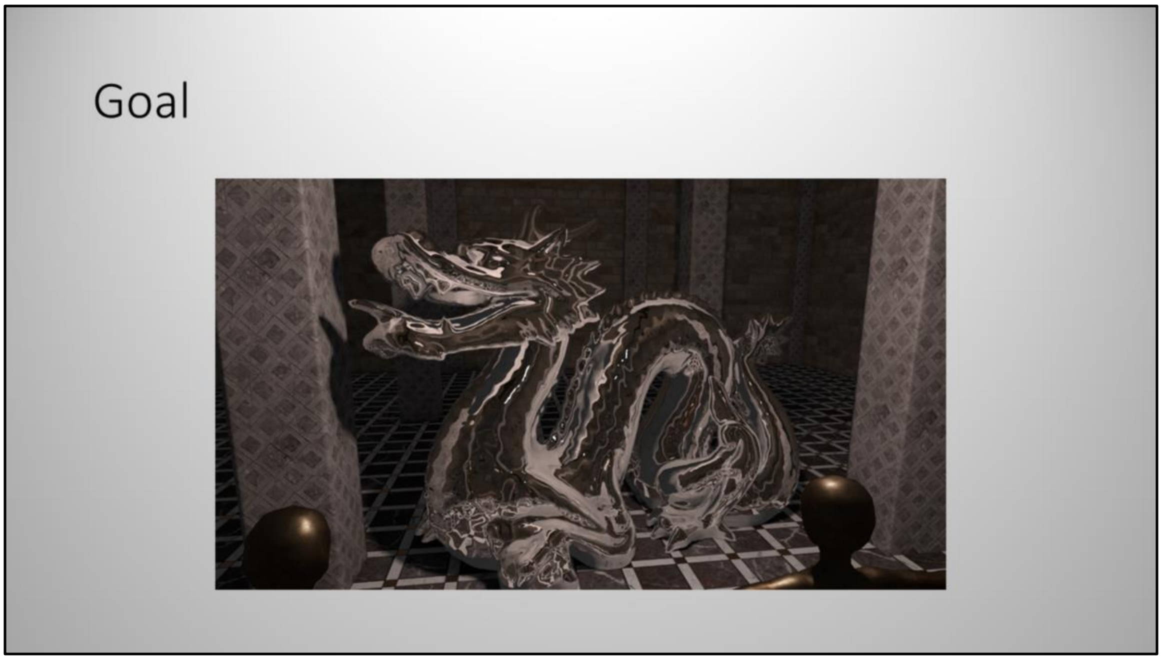


Rendering the World Using a Single Triangle: Efficient Distance Field Rendering

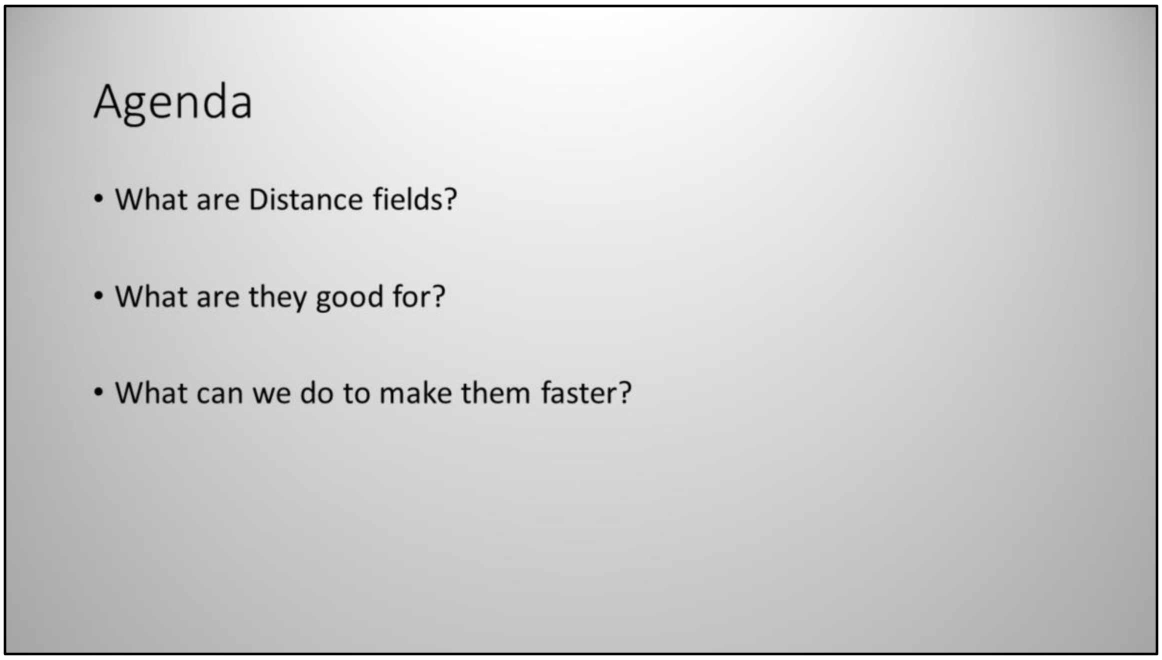
Michael Mroz
Technikum Wien University of Applied Sciences
Visiting at the SIG Center for Computer Graphics, Spring 2017

Hello and welcome to this talk on efficient distance field rendering. Thanks a lot for coming. My goal today is to show you guys how distance fields can enrich your life. So lets get right to it.



Goal

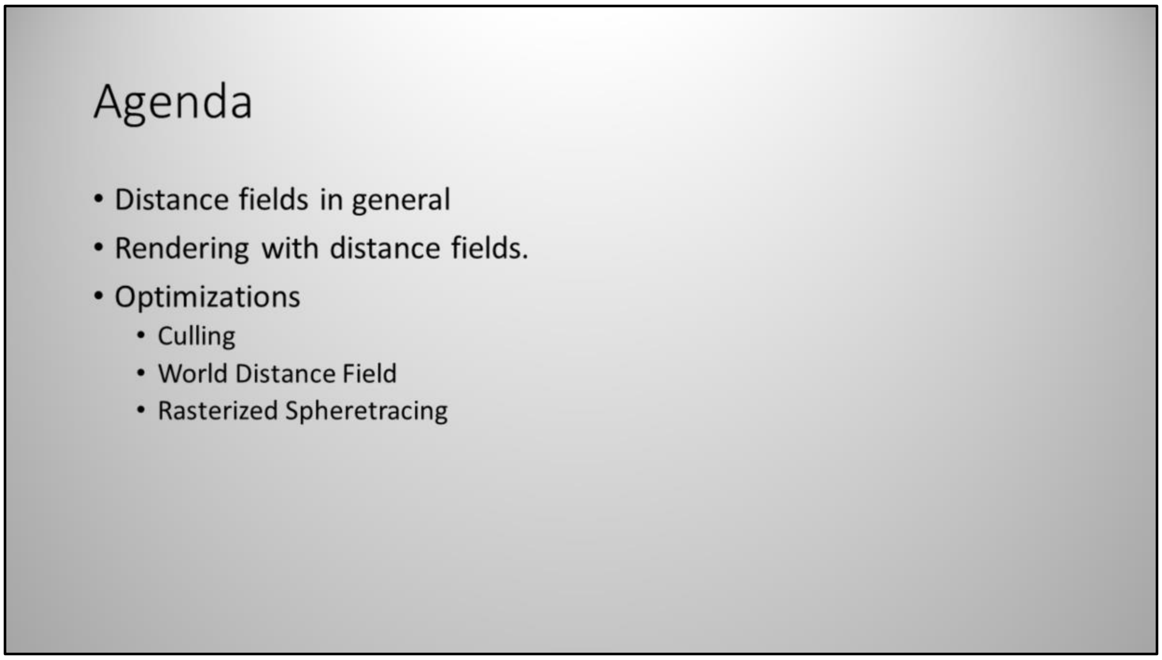
This is our goal. We want to produce physically accurate images in real time through distance fields.



Agenda

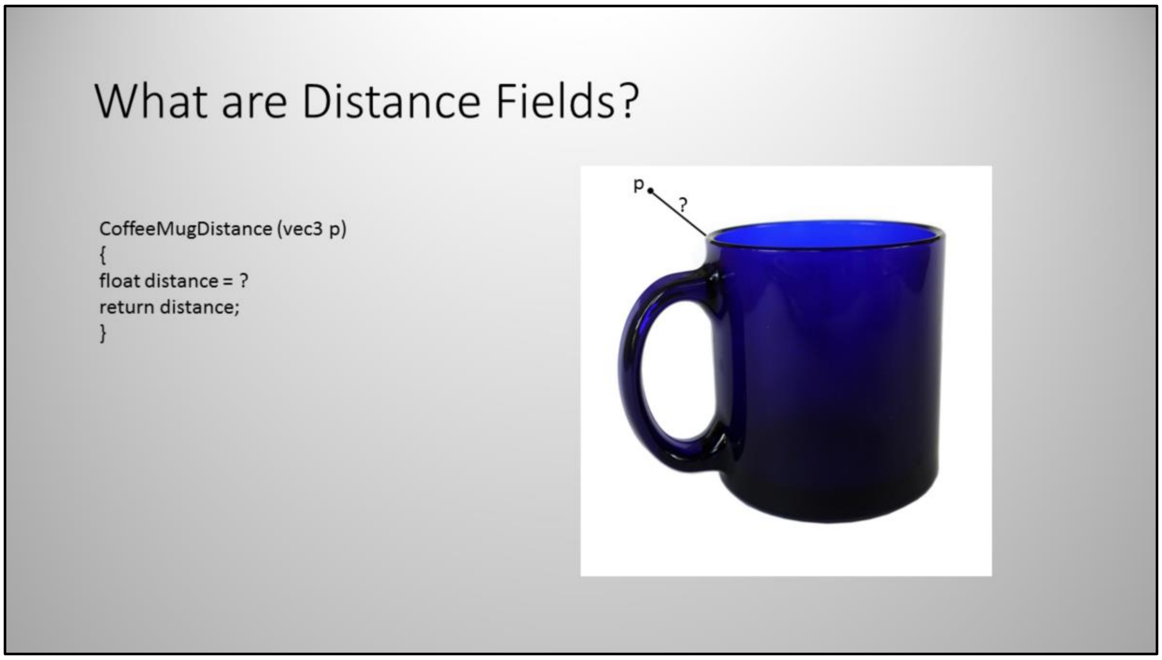
- What are Distance fields?
- What are they good for?
- What can we do to make them faster?

My agenda for today will be to hopefully provide you with the answers to these three questions. What are distance fields, what are they good for and what can we do to make them faster. Those of you who took rachels procedural class may know the answers to the first two questions, but I’m going to go over them real quick in order to get everyone on the same level.



Agenda

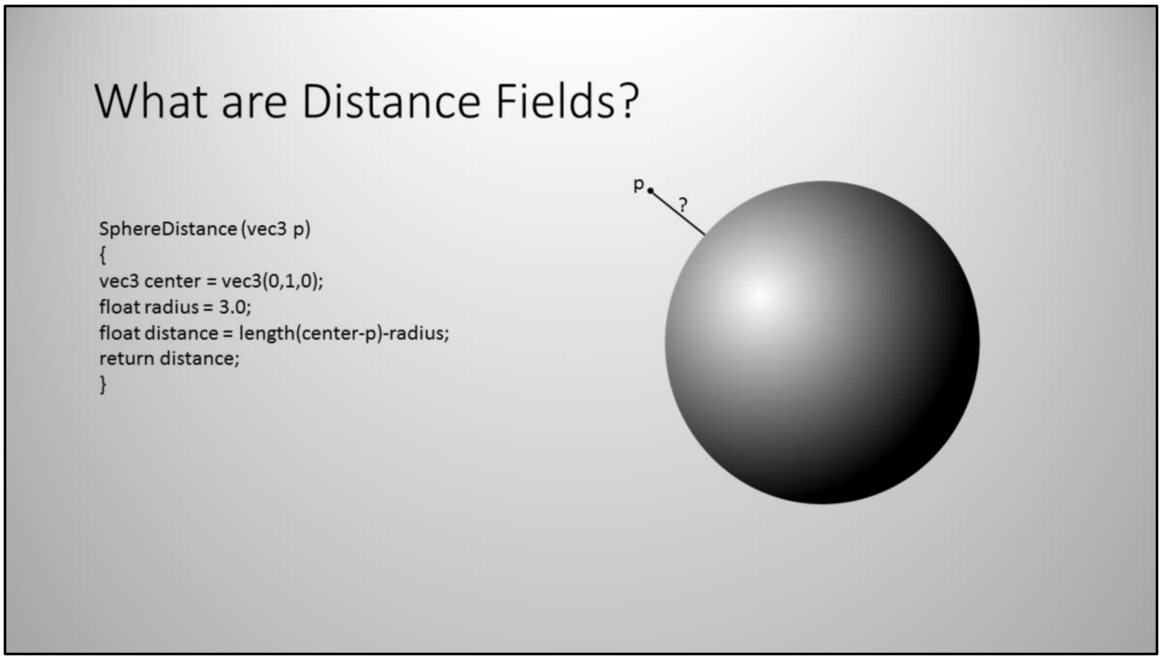
- Distance fields in general
- Rendering with distance fields.
- Optimizations
  - Culling
  - World Distance Field
  - Rasterized Spheretracing



What are Distance Fields?

```
CoffeeMugDistance (vec3 p)
{
float distance = ?
return distance;
}
```

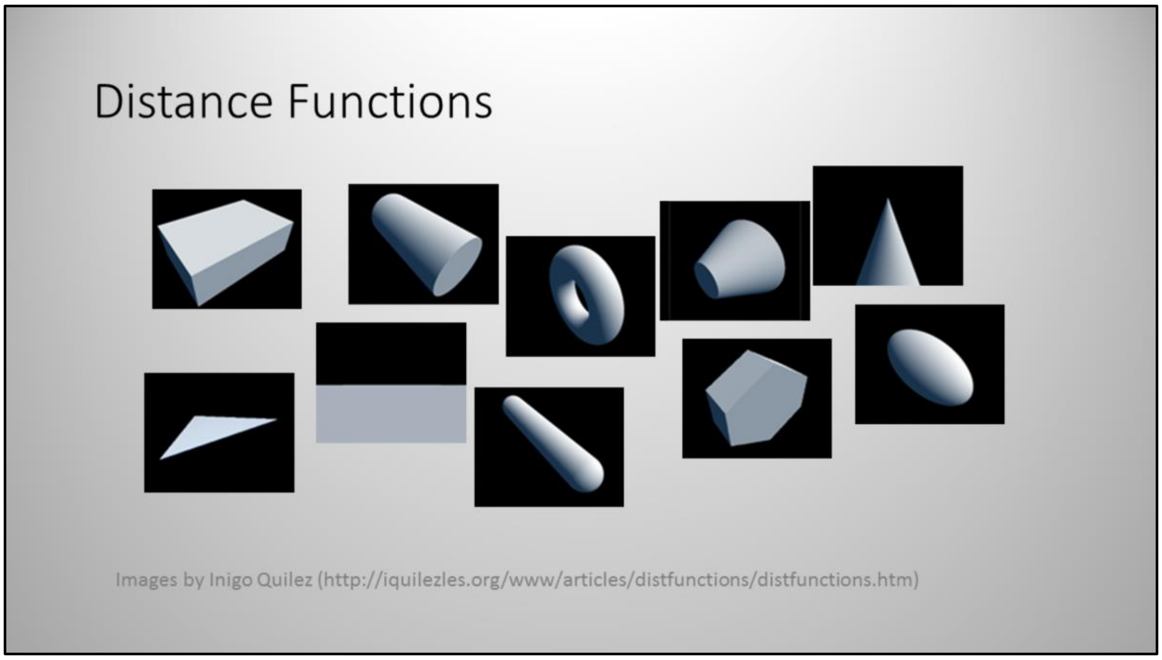
So what IS a distance field? Well distance fields can be seen as all functions that answer a very specific question. This question is: for any arbitrary point in space, how far is this point from the surface of an object. So lets take this coffee mug as an example. In order to have a distance field for this mug, we want a function that we can pass any point to and will return us the distance to the nearest surface of the mug. Well so how could such a function look like for this mug?... That’s not an easy question, is it? This function has to encode the whole surface of this specific object somehow. Lets step back a few steps. Maybe we can break the problem down with simpler terms.



What are Distance Fields?

```
SphereDistance (vec3 p)
{
vec3 center = vec3(0,1,0);
float radius = 3.0;
float distance = length(center-p)-radius;
return distance;
}
```

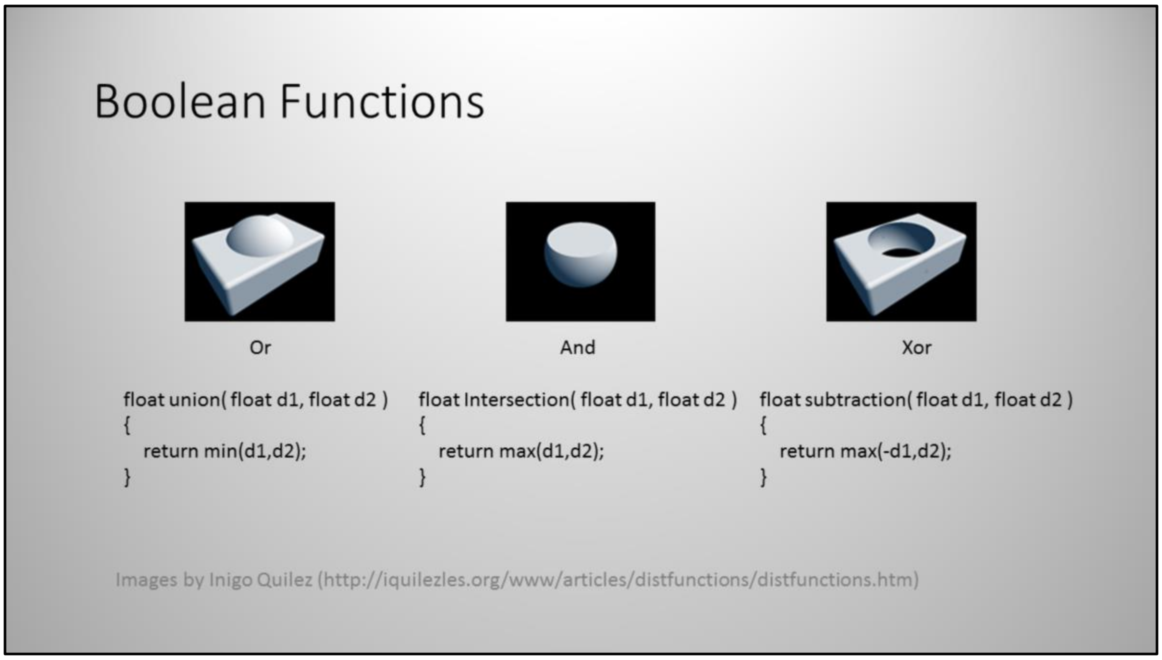
How about we look at a simpler shape first. Lets start with a sphere. Well how would a function look like that tells us the distance from the surface of a sphere for any point in space? That does not seem to hard does it? We can simply take the distance between the point and the center of the sphere, and then subtract the radius from that. This function reveals an interesting property of distance fields. What result do we get if the point happens to be inside the sphere? Our distance will still be the minimal distance to the surface, but it will have a negative sign. This is a very usefull property of distance fields and is the reason why they are called signed distance fields, or SDFs. So this is the simplest known distance function…well are there any other known functions?



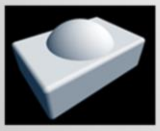
Distance Functions

Images by Inigo Quilez (http://iquilezles.org/www/articles/distfunctions/distfunctions.htm)

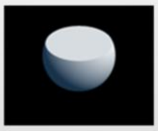
Turns out, yes there are many well defined distance functions for basic platonic solids. These all vary in complexity but are relatively cheap to compute in general. Well…all except the triangle, but more on that later. So now we have these simple implicit surfaces…how can we go more complex?



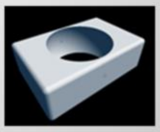
Boolean Functions



Or



And



Xor

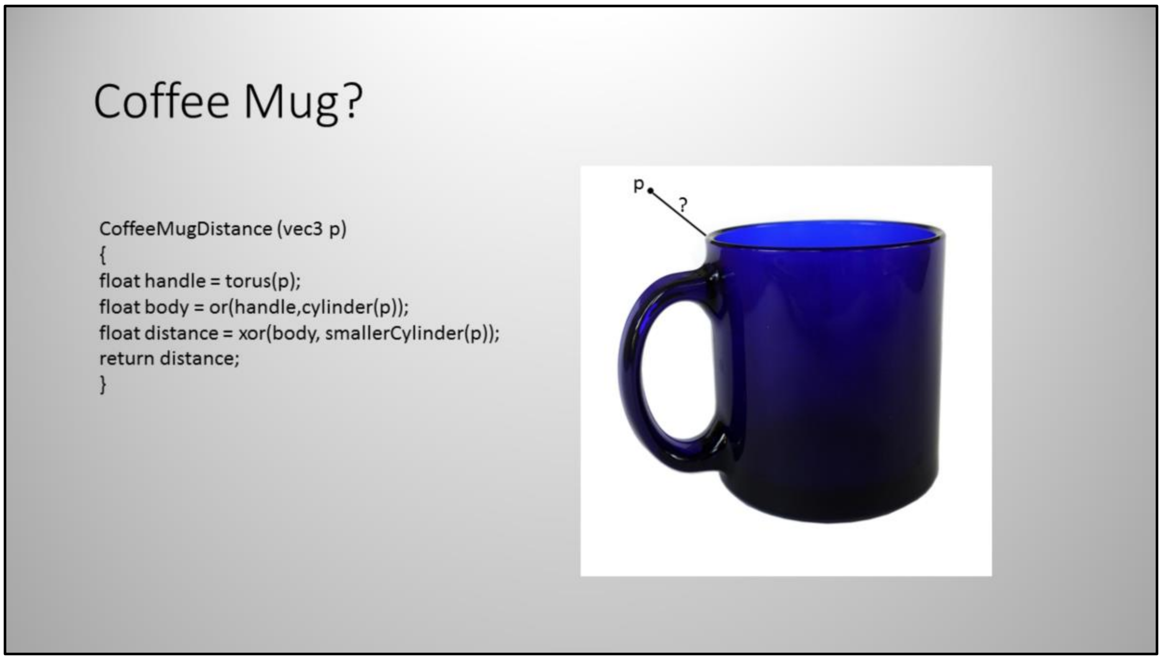
```
float union( float d1, float d2 )
{
    return min(d1,d2);
}
```

```
float Intersection( float d1, float d2 )
{
    return max(d1,d2);
}
```

```
float subtraction( float d1, float d2 )
{
    return max(-d1,d2);
}
```

Images by Inigo Quilez (http://iquilezles.org/www/articles/distfunctions/distfunctions.htm)

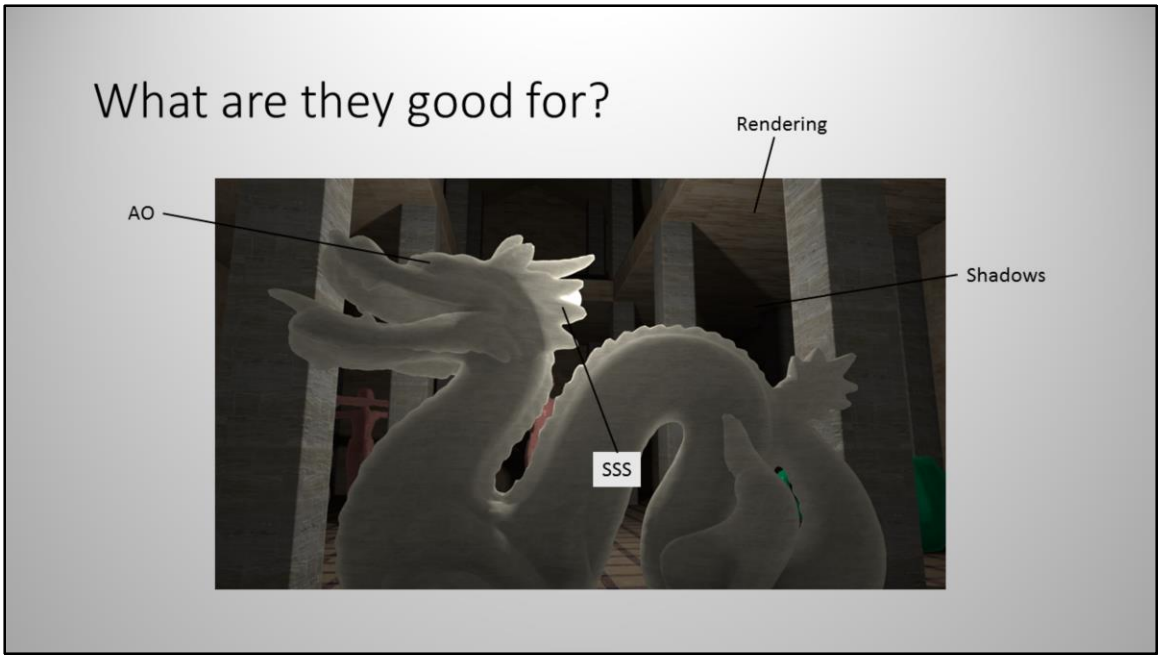
Luckily there is a set of very simple functions that allows us to do Boolean operations on different distance functions. These take the distances returned by two distance functions and return a single distance, thus creating a combined SDF. There are more than these three blending functions, many of which have very interesting visual properties, but lets stay with these for now and see what we can do with them.



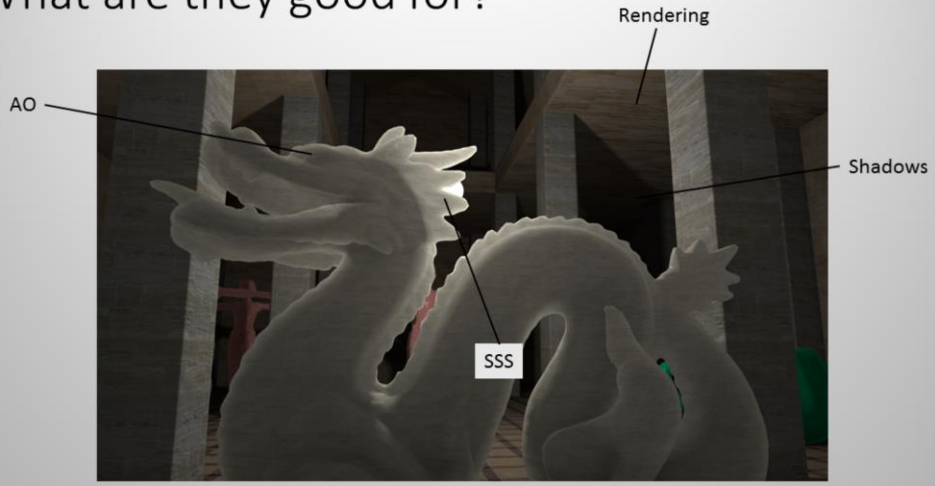
Coffee Mug?

```
CoffeeMugDistance (vec3 p)
{
float handle = torus(p);
float body = or(handle,cylinder(p));
float distance = xor(body, smallerCylinder(p));
return distance;
}
```

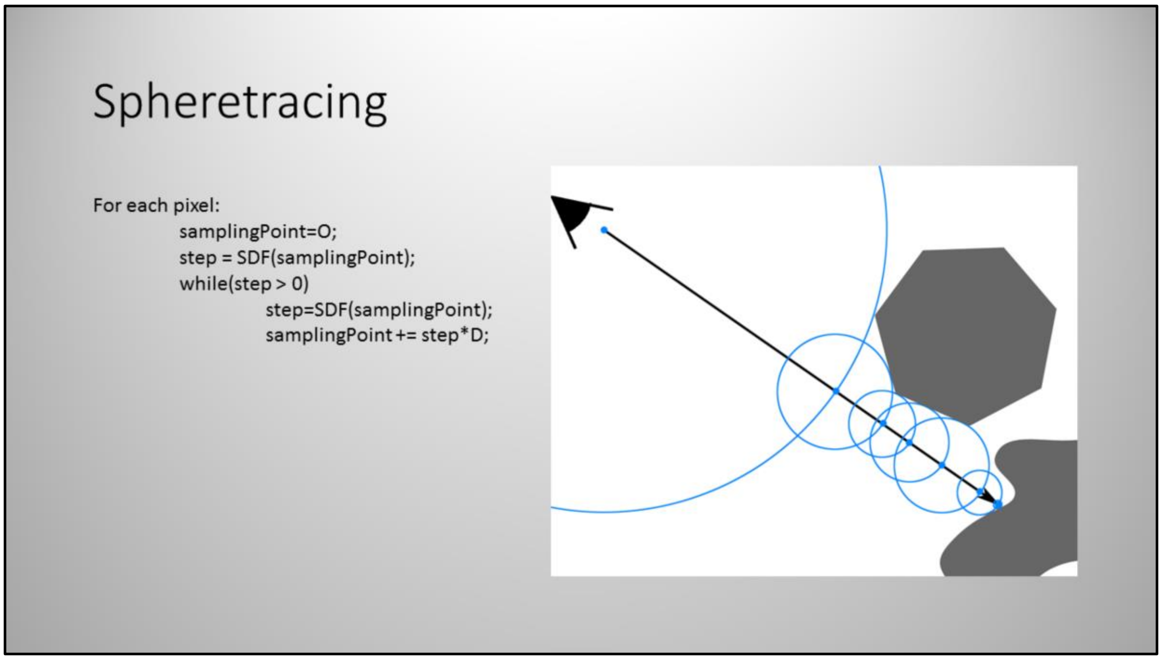
So lets go back to our initial example. With the knowledge that we have now about simpler shapes and blending we could try to model this mug. We could start by taking a torus, merging it with a cylinder, and then hollowing it out by subtracting a smaller cylinder. Of course this would only be a crude approximation of the image that we have here, we would need to use some more distortions and smooth blending to achieve something akin to this. What we have now is something similar to CSG. We could create arbitrary complex shapes out of simple solids.



What are they good for?

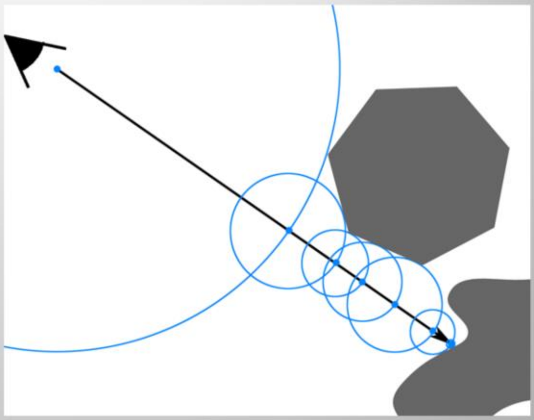


Ok so now we know how to create a distance field….but what are they good for? Distance fields have a very wide field of application in computer graphics. We can render and texture them, calculate soft shadows and geometric AO, as well as estimate SSS, and these are just the most straight forward usecases. Distance field can even help in acceleration of reflections and GI calculations. The main allure of distance fields is the simplicity of the algorithms for these calculations. All of these effects can be created through the same algorithm called spheretracing.

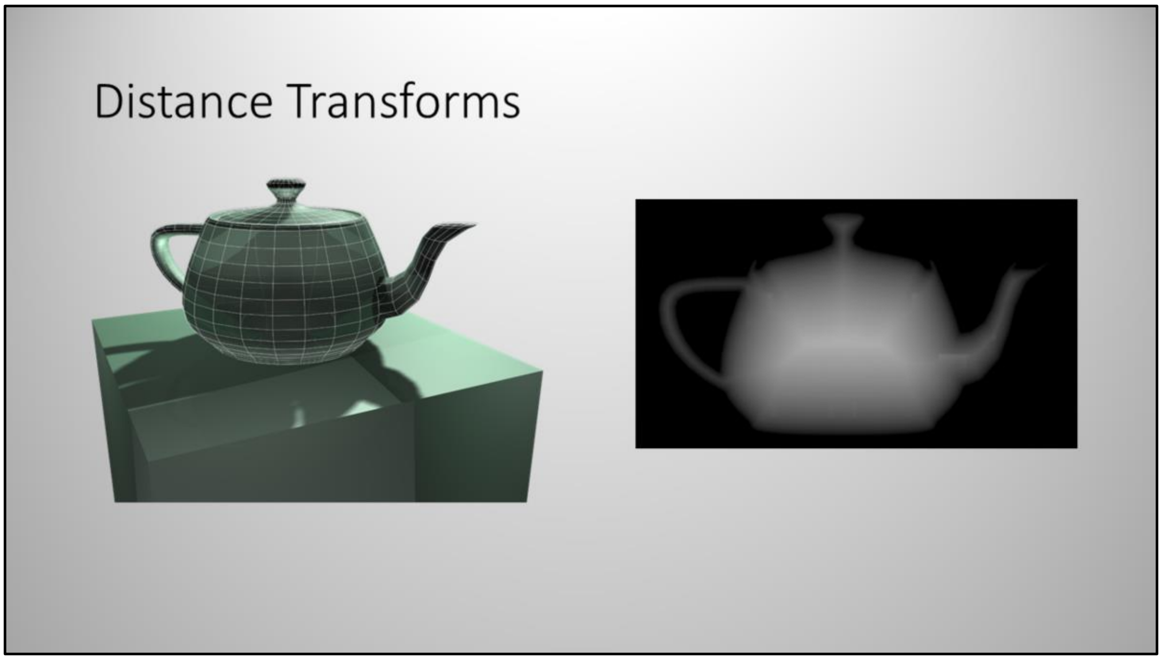


Spheretracing

```
For each pixel:
        samplingPoint=O;
        step = SDF(samplingPoint);
        while(step > 0)
                step=SDF(samplingPoint);
                samplingPoint += step*D;
```

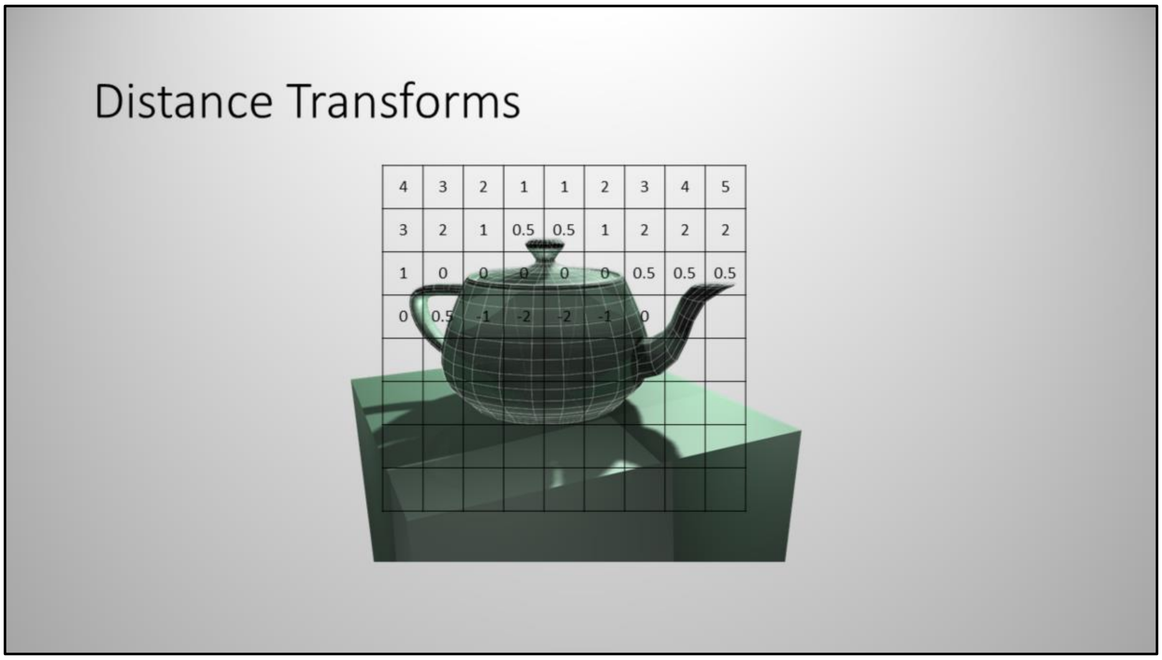


What we essentially want is an algorithm which can tell us the intersection between a ray and the surface of the distance field object. Lets say we have a distance function that describes the implicit shapes seen here. How do we find the intersection between them and a ray? Smart people have come up with this beautifully simple algorithm called spheretracing. We can start at the origin of the ray and poll the distance function at this position. The resulting distance value can be seen as the radius of an unbounding sphere. Lets rephrase that: This value means that there is no object closer than that distance. This is great. It means that we can step this distance along the ray without the risk of hitting something. On this new point we can just check the field again and repeat the entire process. As soon as the polled distance is zero we have found an intersection. So by doing this algorithm for every screen pixel we can render a whole scene in a single fragment shader. And that is really the essence of it all. Want to render a scene: 10 lines of code. Want to have soft shadows? Add two more lines, and just remember how close the ray has passed to occluders, while tracing towards a light source. AO? Just perform 9 traces in a normal oriented hemisphere around a pixel. It IS really that simple.



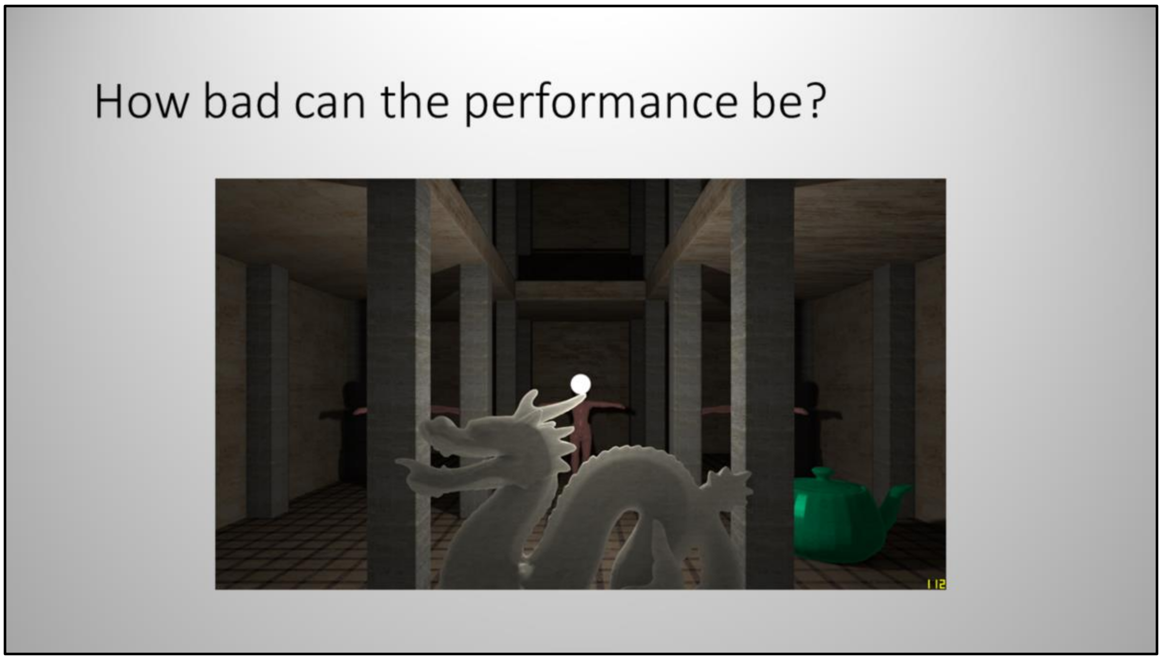
Distance Transforms

Ok so lets say that you work at a game company and for your new project you are looking for a better solution for soft shadows than old-school shadow maps. You remember that that distance fields are great for this and propose the idea to your TD. So he says: great, here are all our mesh assets….What do you do? You cant just remodel all your meshes with implicit distance functions. So the question becomes: how can we take THIS and make it into THIS.

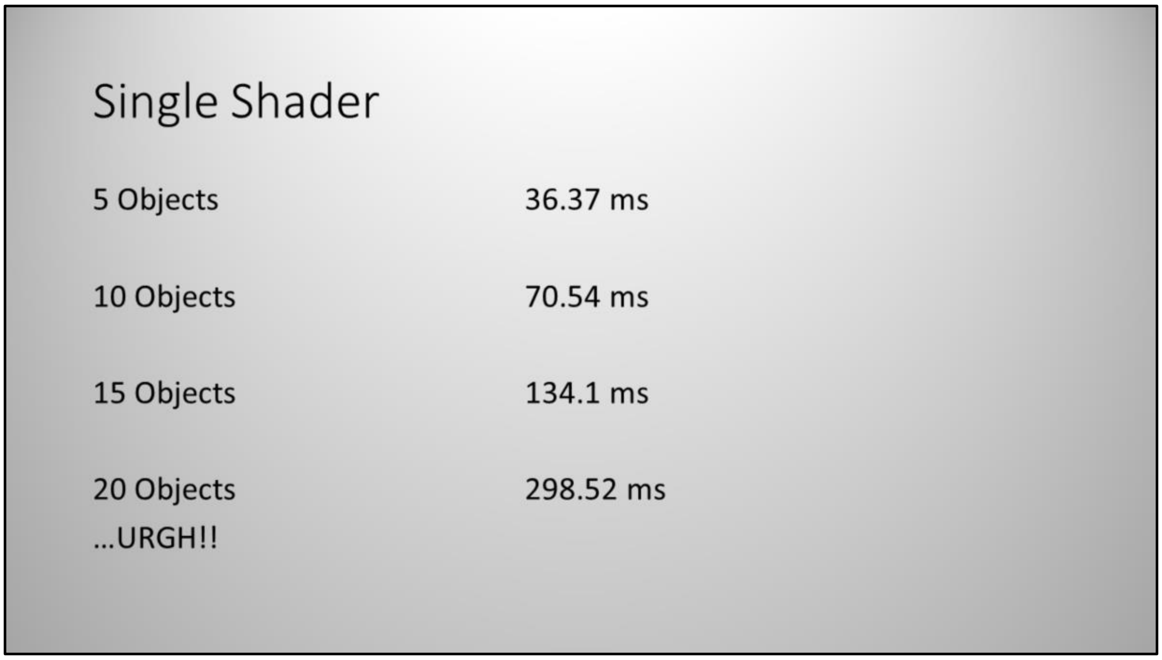


Distance Transforms

Thankfully there has been a lot of research in the past which can help us solve this problem. Lets look at the most straightforward approach. Because an implicit distance function might be very complicated for this mesh, what we can do is to bake a distance field into a 3D texture. The easiest way to do this is to simply overlay the mesh with a regular 3D grid, and perform the costly distance calculation only once for each grid cell. So effectively we have now quantified the distance field of this object. And the really neat part is that even though we only have a discrete amount of sampling points, the GPU lets us sample the texture continuously through bilinear interpolation. This linear interpolation of values is one of the truly important properties of distance fields. It allows us to maintain surface details even with tiny texture sizes. But of course the smaller the distance field texture gets, the less sampling points you have, and the more detail of the object is lost.


How bad can the performance be?

So now you have converted all your mesh assets into distance fields, and write a SDF renderer. You want all the snazzy effects that you have seen on shadertoy, so you implement everything in a single fullscreen fragment shader pass. We start with a very simple scene. There are 5 distance fields represented as 3D textures: the dragon, teapot and 3 humans in the background. The room itself is just implicit geometry. We render the image with simple blinn-phong lighting, soft shadows, AO and SSS. How bad can the performance be, anyways? Right? Unfortunately I have very bad news for you.

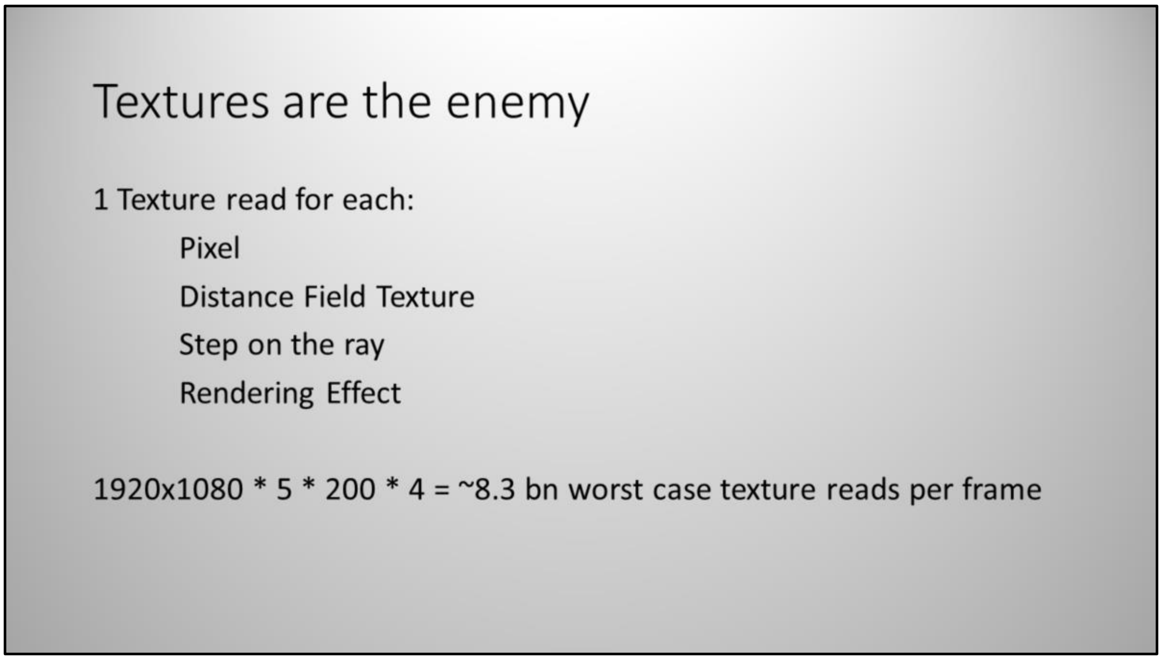


Single Shader

5 Objects	36.37 ms
10 Objects	70.54 ms
15 Objects	134.1 ms
20 Objects	298.52 ms

…URGH!!

With 5 objects we don't even manage to hit 30 fps. And that is on a GTX 980. But which game has only 5 on-screen objects? How does the picture look like for more objects? Let those numbers sink in for a moment. With 20 objects we get 3 FPS…. So there has to be a better way right? Can we get this technique to render 30 fps with reasonable amounts of objects on screen somehow?

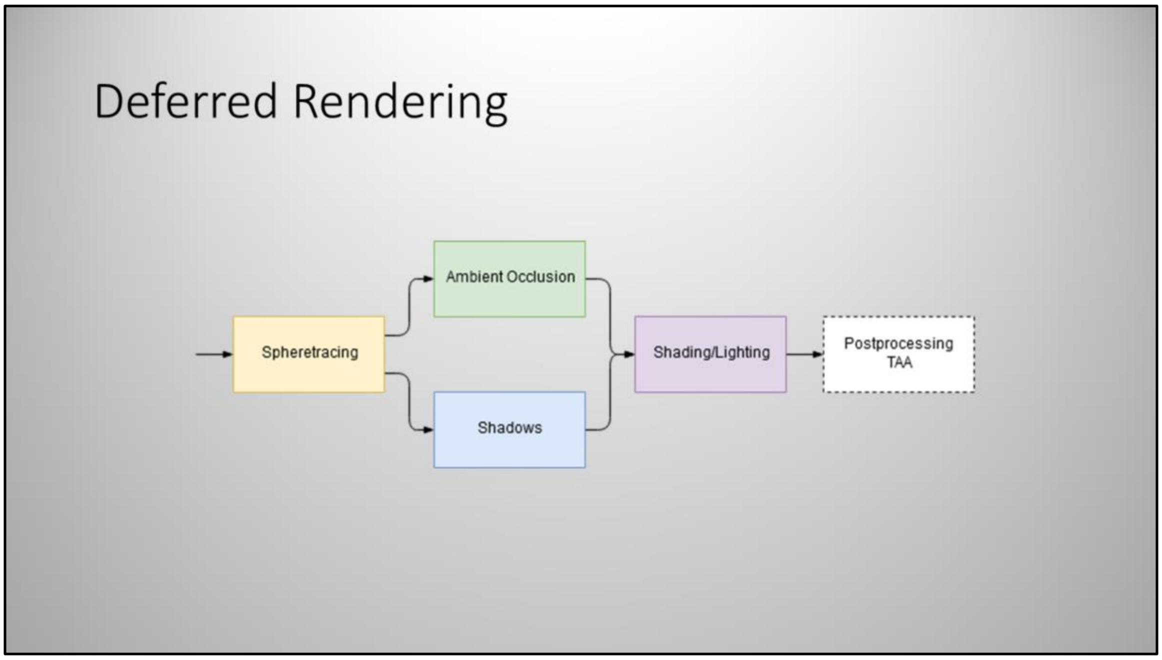

Textures are the enemy

1 Texture read for each:

- Pixel
- Distance Field Texture
- Step on the ray
- Rendering Effect

1920x1080 * 5 * 200 * 4 = ~8.3 bn worst case texture reads per frame

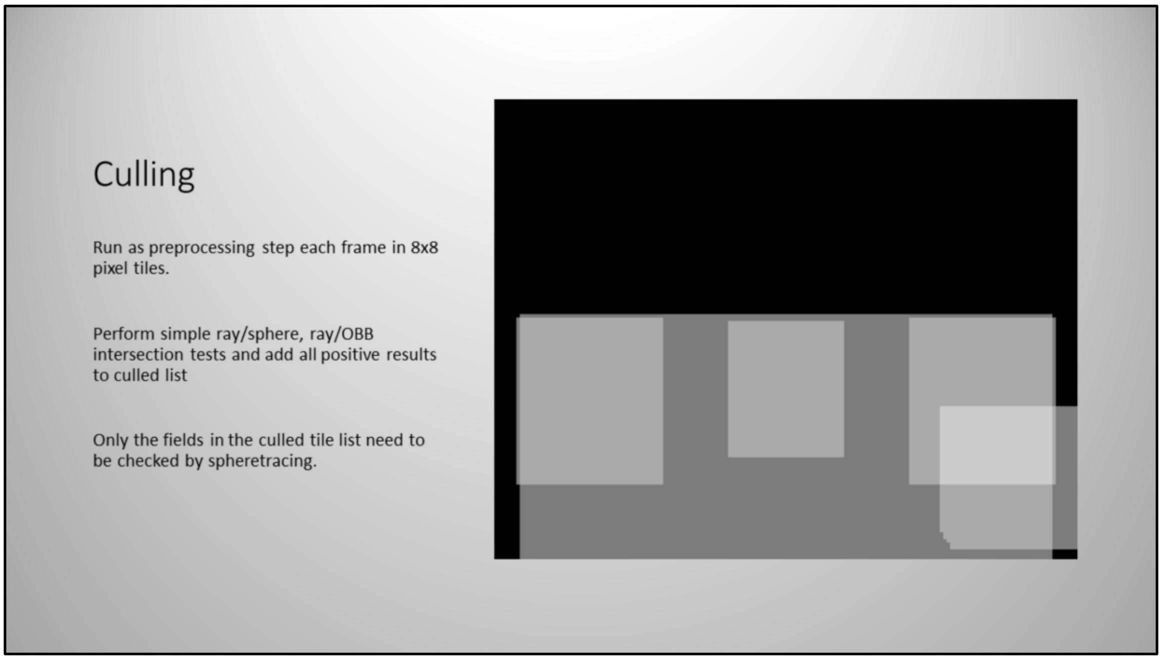
So the best way to optimize any program is to know your enemy. Why is it actually slow? Are there obvious bottlenecks that can be removed? In distance field rendering our enemy is quite easy to identify: texture reads. We need to read a texture for each pixel, each texture, each step on the ray and each spheretrace. Lets do the math… So I hope this makes it apparent why our performance is as bad as it is. The GPU still tries to keep up as much as possible. With only 5 objects, we have enough space in the texture cache to fit small regions of all of them. When we are close to an object, the stepping distances become small and it is likely that the texture accesses happen in similar regions, allowing us to utilize this cache without resorting to the slow global memory. This is a whole nother story with 20 objects. The texture cache has to throw out fetched textures without being able to reuse them, causing the performance of 20 objects to be almost 10x worse than with 5 objects.
So what can we do? Lets start with the basics.



Deferred Rendering

$$\begin{tikzcd} & \text{Ambient Occlusion} \arrow[dr] & & \\ \arrow[r] \text{Spheretracing} \arrow[ur] \arrow[dr] & & \text{Shading/Lighting} \arrow[r] & \text{Postprocessing TAA} \\ & \text{Shadows} \arrow[ur] & & \end{tikzcd}$$

It is actually quite hard to optimize anything if all calculations are still crammed into a single shader. The first step of optimizing distance field rendering is to split all the calculations up into separate steps. This chart illustrates the deferred rendering pipeline that I am currently using. There is no reason why these calculations would need to use the full rendering pipeline, as each one is just a single shader dispatch on a 2D texture. Therefore all these steps can be done in compute shaders. Just by splitting up these calculations into different compute shaders results in gains of about 20%! So that’s a start but can we get better than that? How about tricks that are already heavy in use for regular engines?



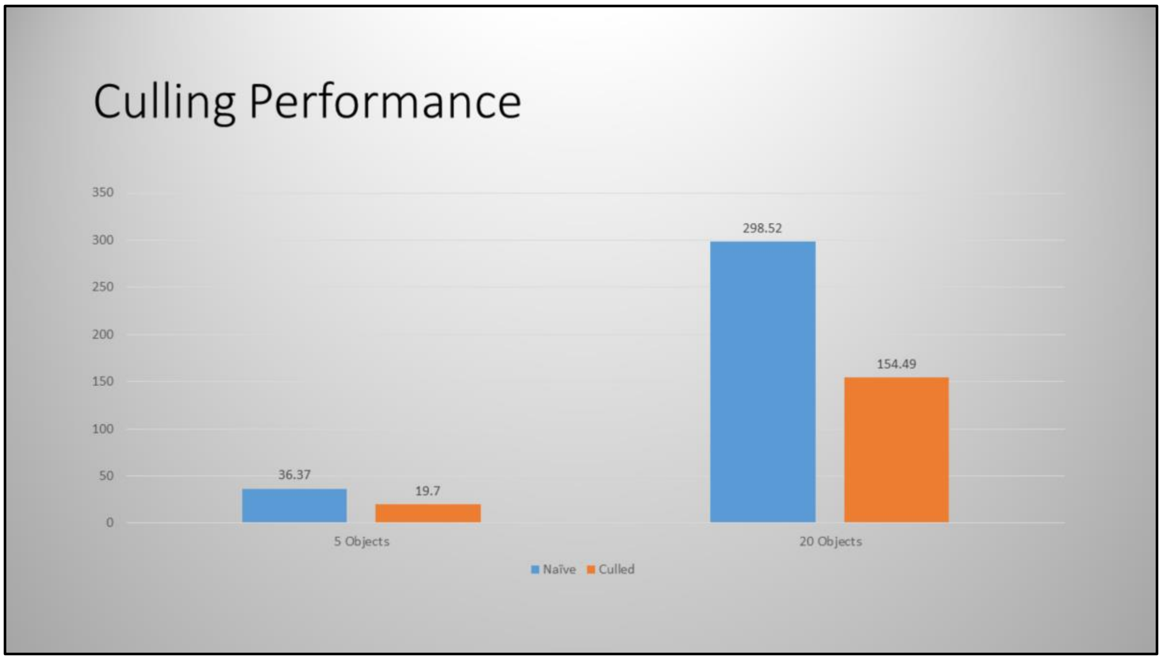
Culling

Run as preprocessing step each frame in 8x8 pixel tiles.

Perform simple ray/sphere, ray/OBB intersection tests and add all positive results to culled list

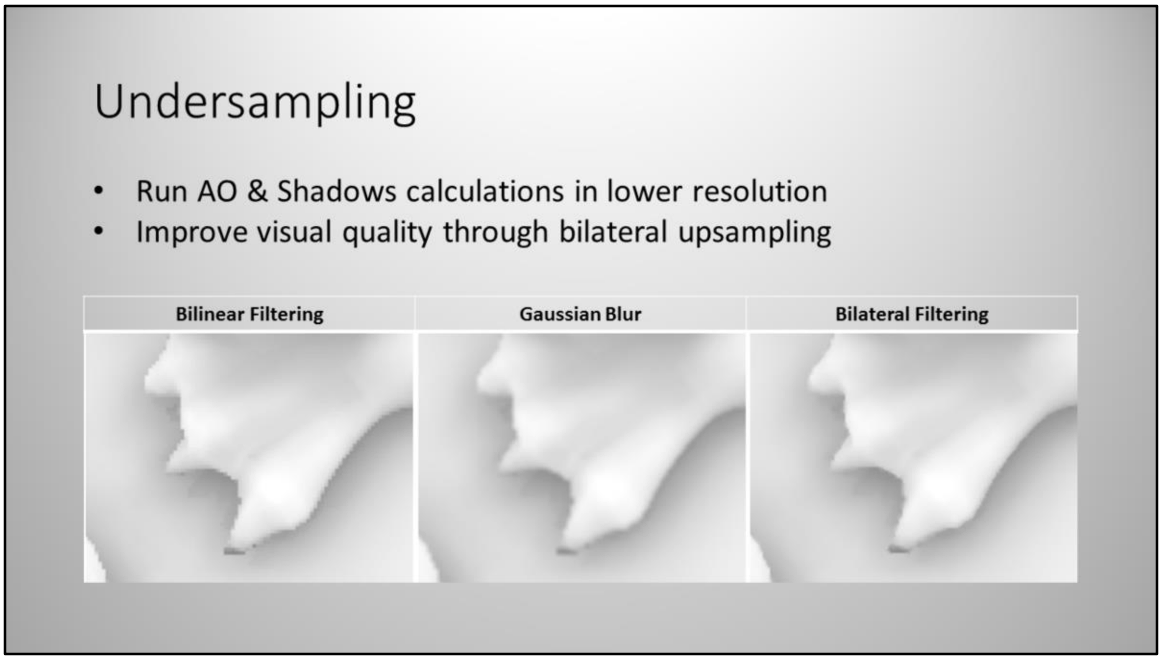
Only the fields in the culled tile list need to be checked by spheretracing.

What we really want is to reduce the number of texture reads. We could start by reducing the number of textures that has to be checked at each step along the ray. This is where our old friend culling comes in. It is pointless to check distance fields of objects that are on the other side of the screen right? So what we can do is wrap all our distance fields into neat little bounding boxes, and run simple intersection tests to find out which fields could possibly intersect our ray. What we do in our renderer is to run the culling step in 1/8 of the screen size in both dimensions, resulting in 8x8 pixel tiles. For each of these tiles a ray is intersected with all the scene objects and a list is created containing intersecting fields. When the spheretracing algorithm checks the distance field, only the actual textures in the culling list for this pixel need to be checked. A BVH or similar structure can be used to speed up these culling calculations.



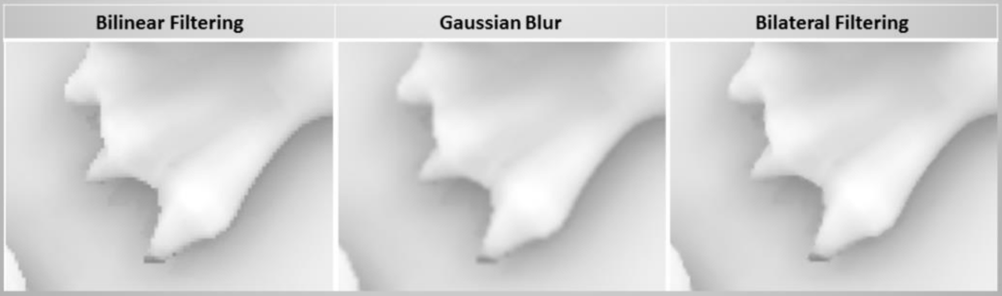
Culling Performance

So this simple culling technique gains us a lot of performance! We are almost twice as fast, but still way off our 60 fps goal.

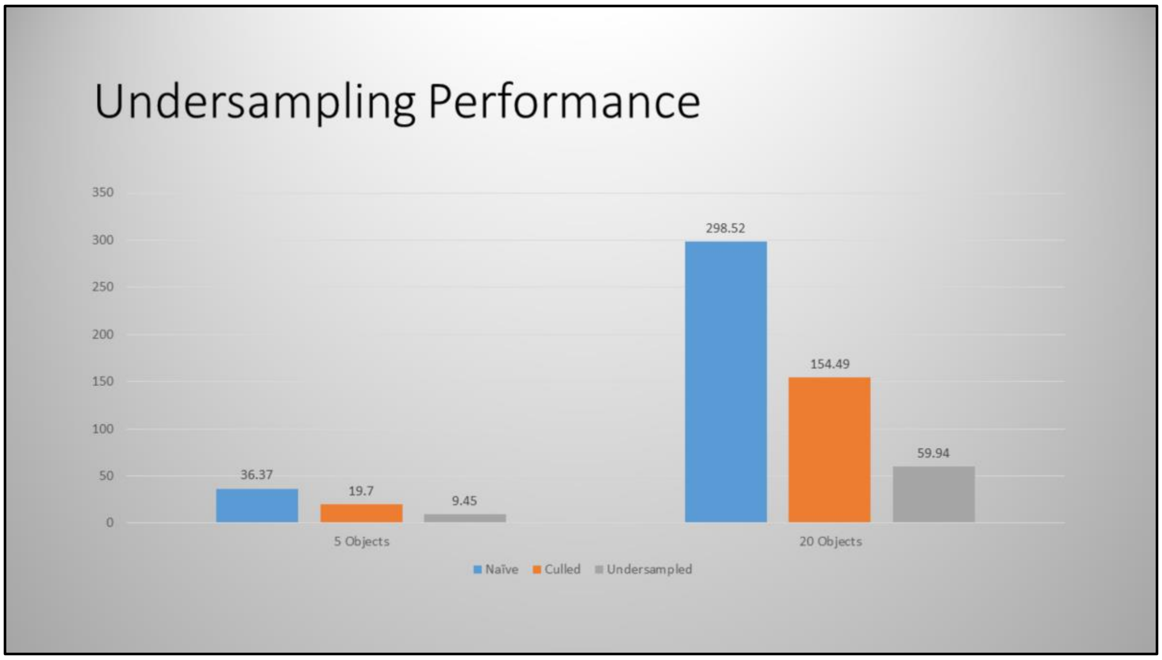


Undersampling

- Run AO & Shadows calculations in lower resolution
- Improve visual quality through bilateral upsampling

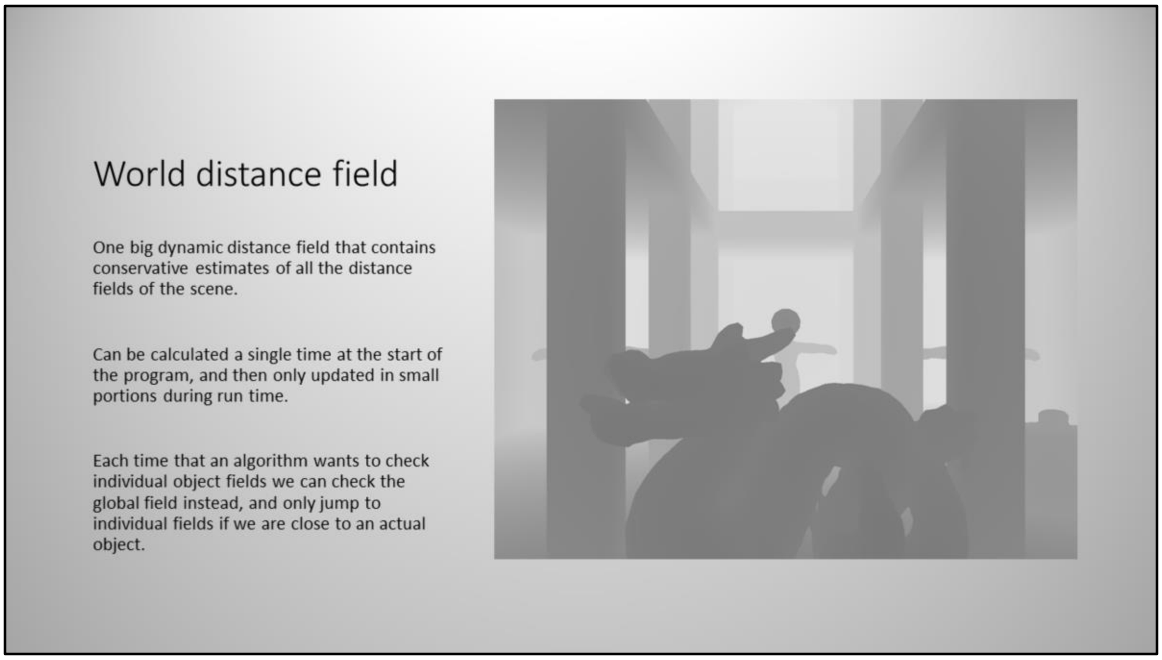


Where else can we reduce texture reads? Well how about the number of screen pixels? The initial spheretracing has to be performed in screen resolution in order to get nice and crisp edges and normals, but how about shadows and AO? The resulting buffers contain mostly low-frequency information anyway, so we could probably get away with rendering them in a lower resolution. Lets reduce the rendering size of shadows to half and the size of AO to a quarter in each dimension. This will obviously lead to vast improvements in rendering times when only a quarter or one sixteenth of all pixels need to be calculated, but will result in visible pixelizations and other undersampling artifacts when applying the data to our lighting calculations. The left image shows such AO buffer that was naively upsampled using bilinear sampling. The ugly pixels can be hidden by blurring the whole buffer with a simple gaussean blur, as seen in the center image. This looks much better, but the gaussean has also smeared our lovely AO information all over the neighboring pixels without honoring object edges. This is where bilateral upsampling comes to the rescue! It is an extension of a gaussean filter, but the neighboring pixels are also weighted by their normals and depth differences. This results in a blur that preserves object edges. So how much do we gain by undersampling these calculations?



Undersampling Performance

Now we are getting somewhere! We are under 10 ms for 5 objects, but 60 ms for 20 objects is still far off our goal. So lets just keep going.



World distance field

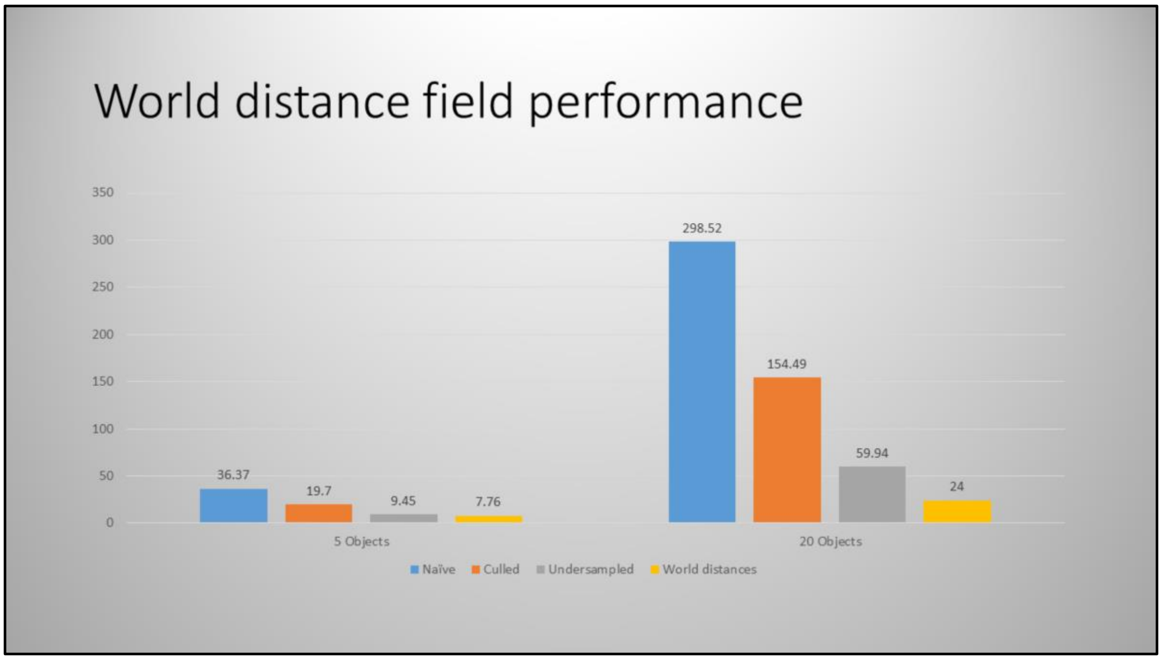
One big dynamic distance field that contains conservative estimates of all the distance fields of the scene.

Can be calculated a single time at the start of the program, and then only updated in small portions during run time.

Each time that an algorithm wants to check individual object fields we can check the global field instead, and only jump to individual fields if we are close to an actual object.

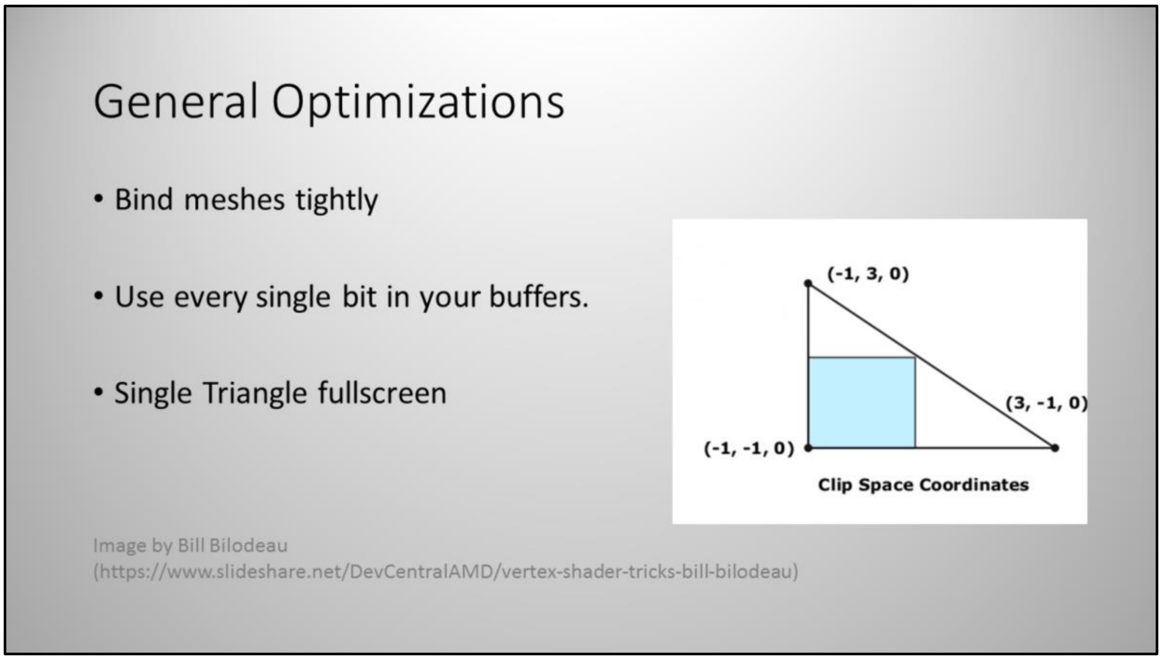
We can reduce the amount of individual object fields hat have to be checked by creating a big distance field spanning the entire scene that contains all the smaller objects. We can run this as a preprocessing step, that calculates this huge field a single time during startup. If we could make a world field that has the same resolution as the objects themselves we would be basically done with optimizing textures. Only this single texture would need to be by the shader as it would contain all the smaller fields. Unfortunately this is not really a possibility as a field of such a resolution would not even fit on our GPU memory for scenes of decent size. What we can do is calculate and save the world field in a low resolution texture, and add a fixed offset to each grid cell. This results in a conservative distance estimate of the actual scene. If we were to render the scene by just using this field, our objects would appear bulky and devoid of details, as seen in this picture.
So why is this important?
Well think back to what our biggest enemy is: texture accesses. We want to reduce the number of textures that need to be checked at every step? What this allows us to do is to just use the world field whenever the ray passes through big empty spaces, where details and field resolutions are irrelevant
and only fall back to the culled object lists if we are close to an object.
If we only need to access a single texture for most of the way, instead of 20, we can

gain a lot of performance.
This approach can even be extended further to contain more than just distances, which results in higher performance gains.
So how dos our world field hold up in our performance chart?



World distance field performance

So where do we stand with these optimizations? Well look at that. We can now even render 20 objects in 24 ms. I guess we can be pretty proud of ourselves and go home now, right? I don’t think so…this is still waay to slow. We can do better.



General Optimizations

- Bind meshes tightly
- Use every single bit in your buffers.
- Single Triangle fullscreen

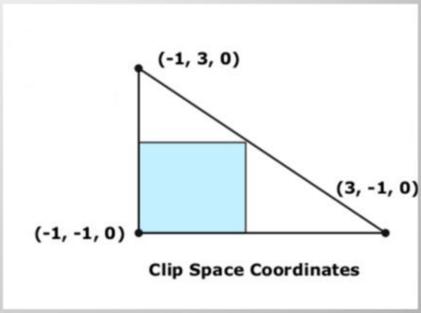
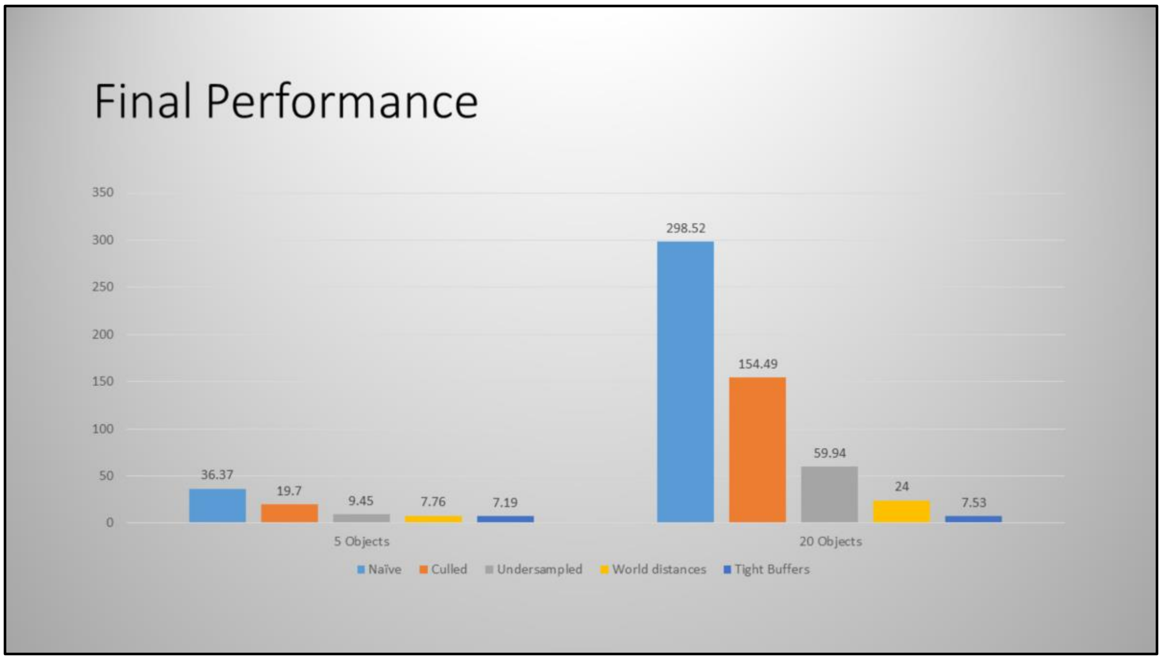


Image by Bill Bilodeau
(https://www.slideshare.net/DevCentralAMD/vertex-shader-tricks-bill-bilodeau)

So what else could we do to gain some valuable ms? We could pack our distance field textures as tightly as possible. The less empty cells we have in a texture the better. Tight distance textures mean that we need less resolution for the same amount of detail. Less resolution means we can fit more of it in our GPU texture cache. Do we really need full 32 bit floats in the textures? We could probably do with half precision for our small objects.
What about our other buffers? Normals can be packed into 2x16 bit, and AO and shadow factors are fine with 8. The object ids in our culling lists don’t need to be 32 bit ints if we never use more than 256 unique objects, right?
The less memory we need the more stuff can we fit into our GPU caches.

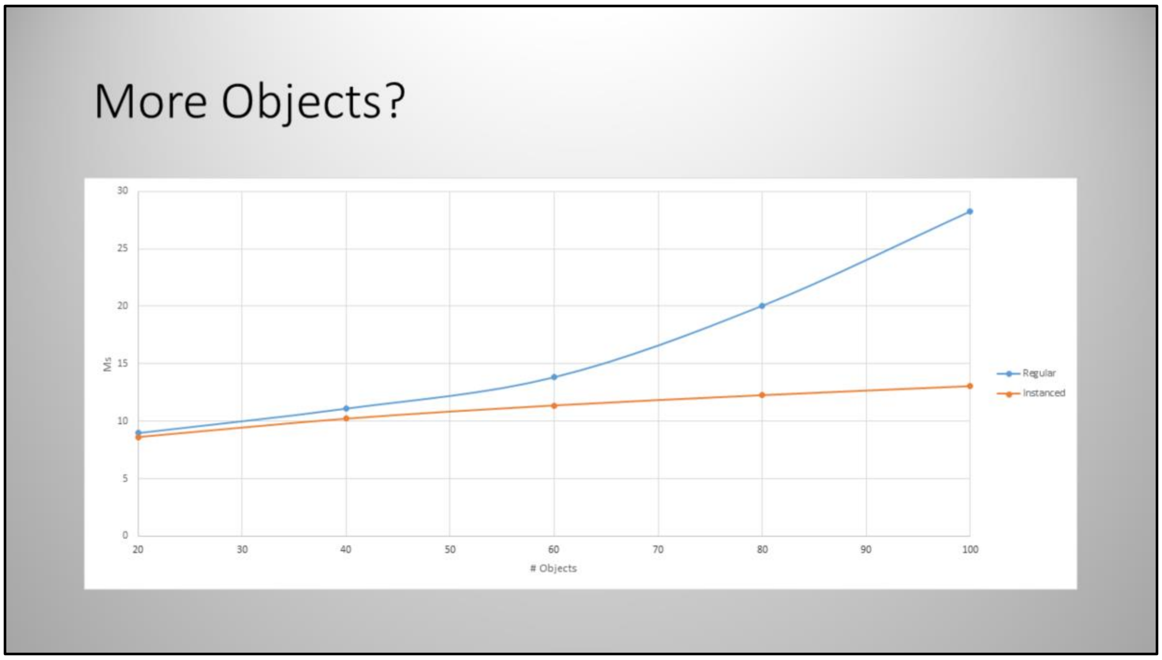
And finally the explanation for the name of this talk: Conventionally, when rendering a post-processing effect two triangles are used that form a fullscreen quad. But what actually happens in the rendering pipeline is that 2x2 pixel quads are rasterized and passed to the fragment shader. This is done so that derivatives for mipmapping can be calculated. So in the end, by using 2 triangles we have slight overdraw at the diagonal. We can get rid of that easily by drawing not 2 trangles but a single one. This triangle has to be big enough to cover the entire screen. The triangle gets clipped to the screen during rendering, resulting in speedups of about 5%.

But these are all pretty small things, right? How much can we realistically gain by this? Well…

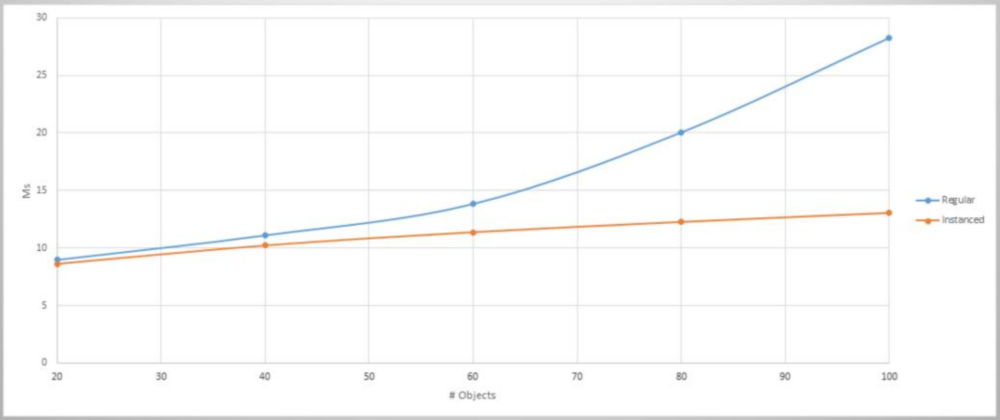


Final Performance

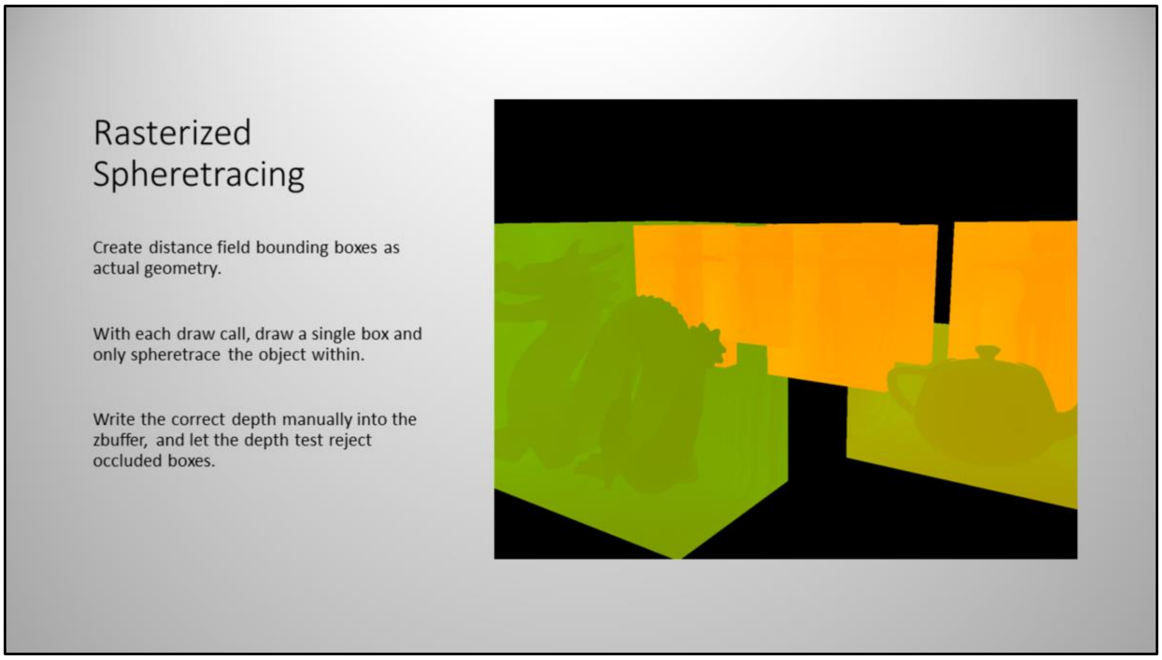
So there we have it. With these optimizations combined we can render 20 objects in under 8 ms. With all the described optimizations we were able to reduce rendering times by about 4000% for 20 objects.



More Objects?



Ok, that is very nice. But which modern game has only 20 on-screen objects, right? So how well does it scale with more objects? The answer is: it depends. As you can see, we are running smoothly until about 60 objects. At this point each object leads to a non-linear increase in rendering time. What we observe here is the exact same issue that we had with our naïve implementation. The only difference is that we can now fit about 60 objects in our GPU texture cache instead of 10. So what can we do about that? One easy solution to this problem is instancing. If we want use the same distance field texture in the scene multiple times we can just re-use its data. As you can see object instancing allows for hundreds of objects while still having reasonable rendering times.
But what if we want reasonable rendering times with a lot of unique objects? What we essentially did until now was to push the problem further and further away from us. What if we could get rid of it altogether?



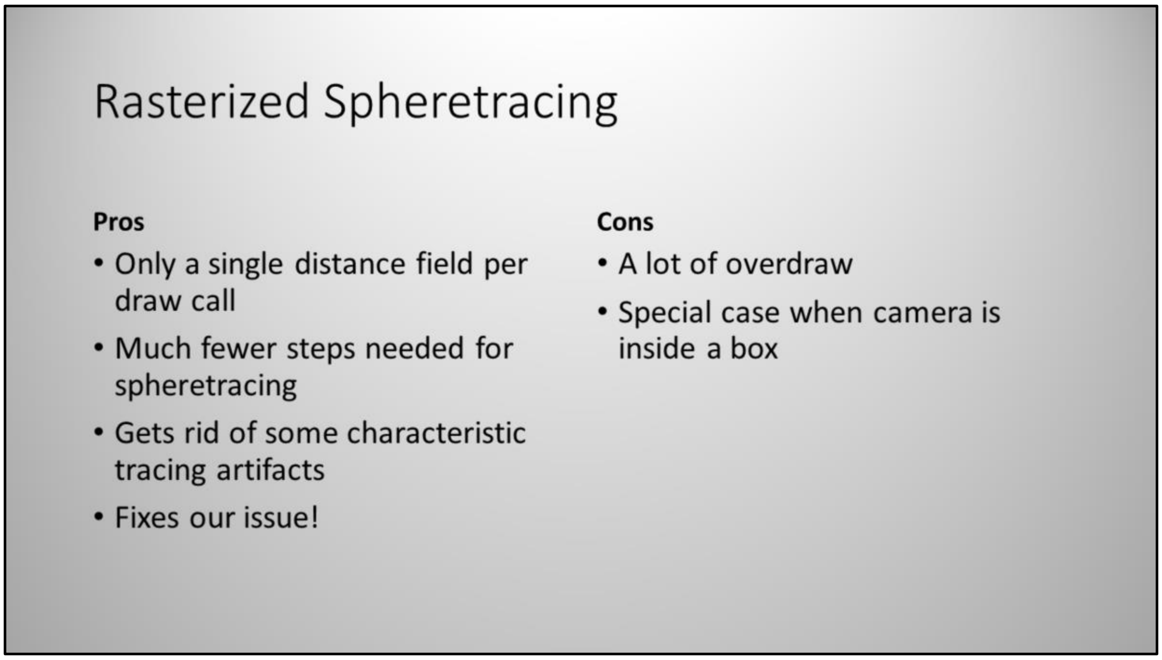
Rasterized Spheretracing

Create distance field bounding boxes as actual geometry.

With each draw call, draw a single box and only spheretrace the object within.

Write the correct depth manually into the zbuffer, and let the depth test reject occluded boxes.

Lets step back a bit. What causes our non-linear performance loss with high object counts? There are simply too many distance field textures that need to be checked each frame. They do not fit in the GPU caches any more and we need to wait for slow fetches from global memory. Can we somehow solve this problem in a straightforward way? We can, by stepping away from a paradigm that has followed us from the very first single shader implementation: That we have to trace all the objects in a single fullscreen pass. Enter rasterized spheretracing. In order to split up the spheretracing process we create actual geometry from the distance field bounding boxes. These cubes are drawn just like normal mesh geometry. The magic happens once we reach the fragment shader. We know that the surface of our object has to be somewhere inside this rasterized box. We can use regular spheretracing to find the correct depths, and reject all rays that exit the box on the back side. By writing those depths to the zbuffer we can perform hardware depth testing to reduce overdraw.



Rasterized Spheretracing

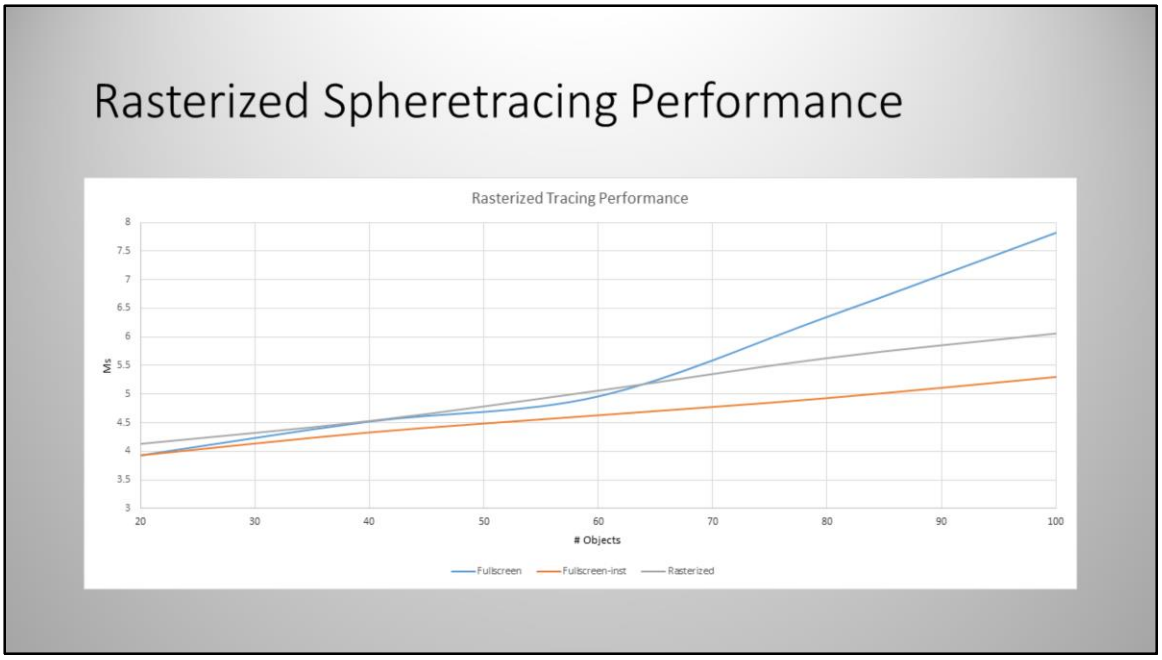
Pros

- Only a single distance field per draw call
- Much fewer steps needed for spheretracing
- Gets rid of some characteristic tracing artifacts
- Fixes our issue!

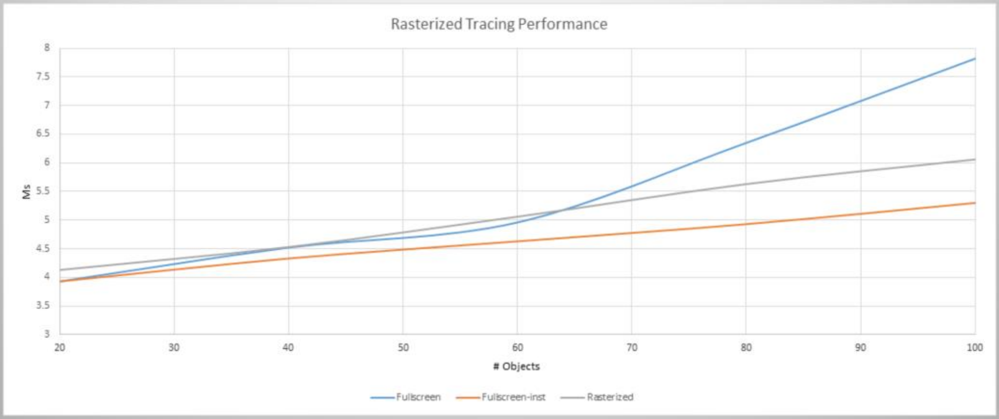
Cons

- A lot of overdraw
- Special case when camera is inside a box

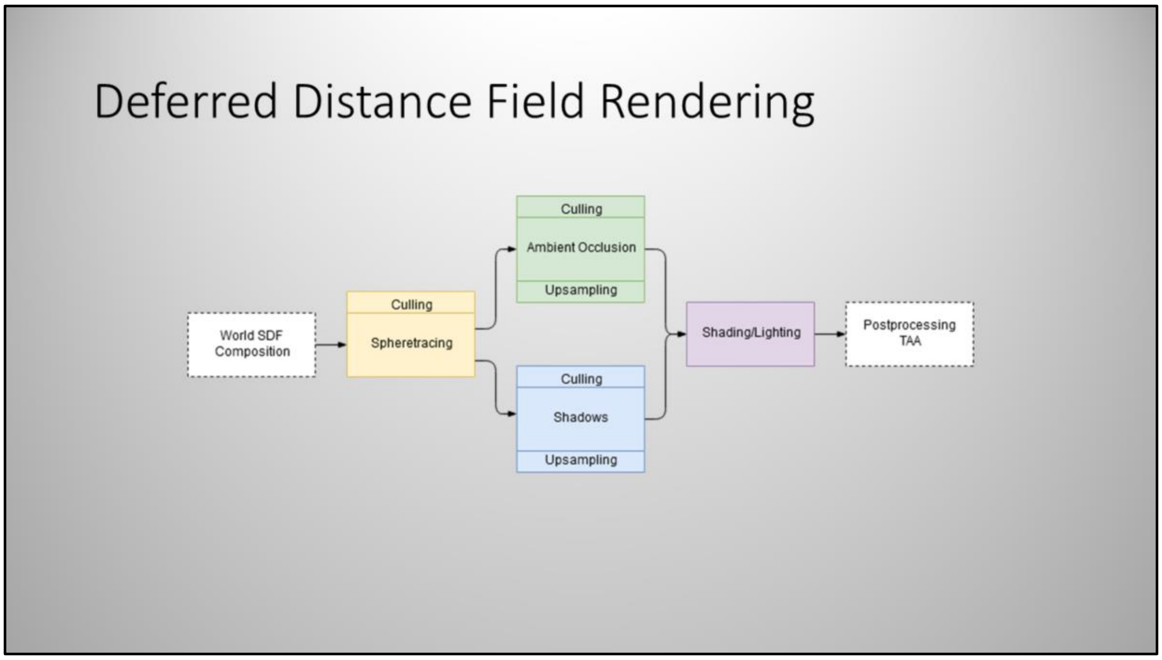
So what do we gain by this change? Well we only need to trace a single object for each draw call. We know exactly when a ray leaves a box so we can abort a trace much earlier than usual. Furthermore it fixes some of the characteristic spheretracing artifacts that result from running out of steps. But most importantly it fixes the GPU cache issue!! If we trace only a single object at a time, we only need to access a single texture! Isn't that great?
So as you may have guessed, the propably wouldn't be a pro section without a con section. The big downside of this algorithm is overdraw. If there are multiple objects right behind each other, then every ray that does not hit the objects itself will be traced multiple times until a surface is found. Overdraw is already bad for mesh renderin, but its even worse in our case as each fragment shader invocation performs spheretracing which is quite costly. In order to keep overdraw costs reasonable it is important to perform front-to-back ordering for the draw calls, and pack objects as tightly as possible into the distance field textures.
Finally we need to take care of the case when the camera itself is inside a distance bounding box. As we only draw the front facing triangles, this would lead to the object simply disappearing. This case can be detected on the CPU side and handled by drawing the backfacing triangles and reconstructing the correct ray positions in the fragment shader.



Rasterized Spheretracing Performance

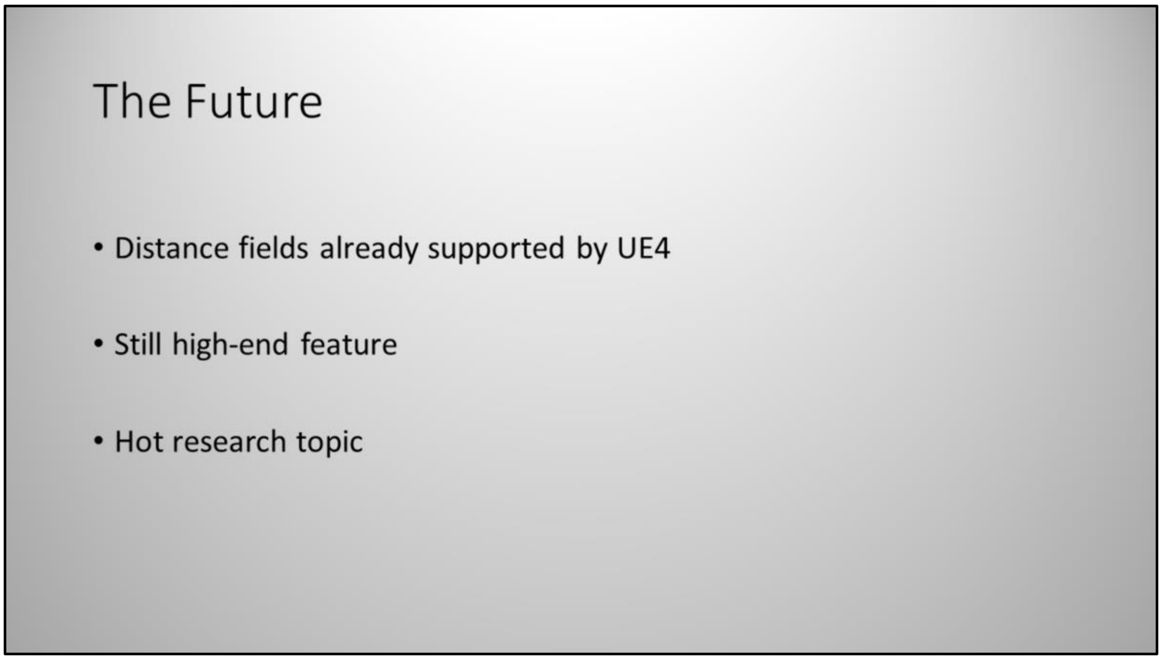


Ok so how does it perform? As you can see, it depends on the number of objects. As long as we are under 60 objects, the conventional spheretracing approach outperforms rasterization, but this changes as soon as we hit out cache issue. Conventional spheretracing times skyrocket while rasterized tracing continues linearly. You should keep in mind that the conventional trace uses all of the previously described optimizations: culling, world distance and id fields while rasterized tracing uses none… If these optimization techniques could be adapted to fit this novel technique it could blow fullscreen spheretracing out of the water.



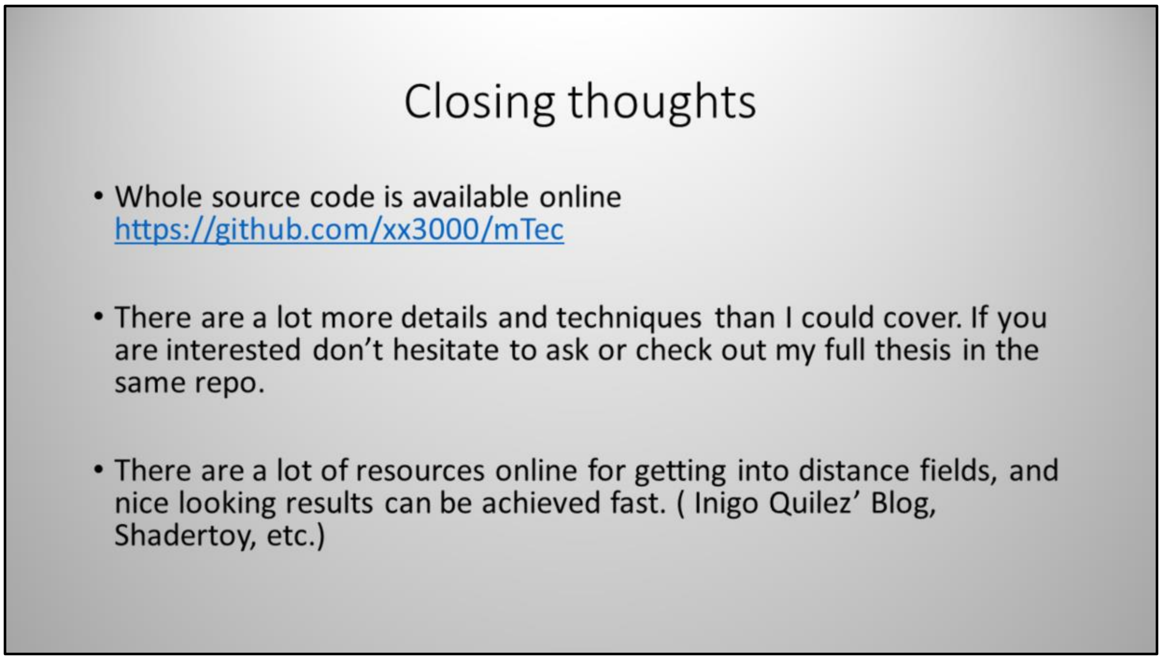
Deferred Distance Field Rendering

Ok so our final rendering pipeline looks somewhat like this. We start with updates to the world distance field, then create the culled tiles and spheretrace. The resulting depth and normal buffers are used by shadows and AO shaders, which also have a culling step beforehand. After these effects are calculated in lower resolution they are upsampled with bilateral filtering. All of these buffers are composited and lighted using basic blinn/phong shading. Finally TAA and motion blur is performed as a postprocess effect in order to improve the overall visual quality.



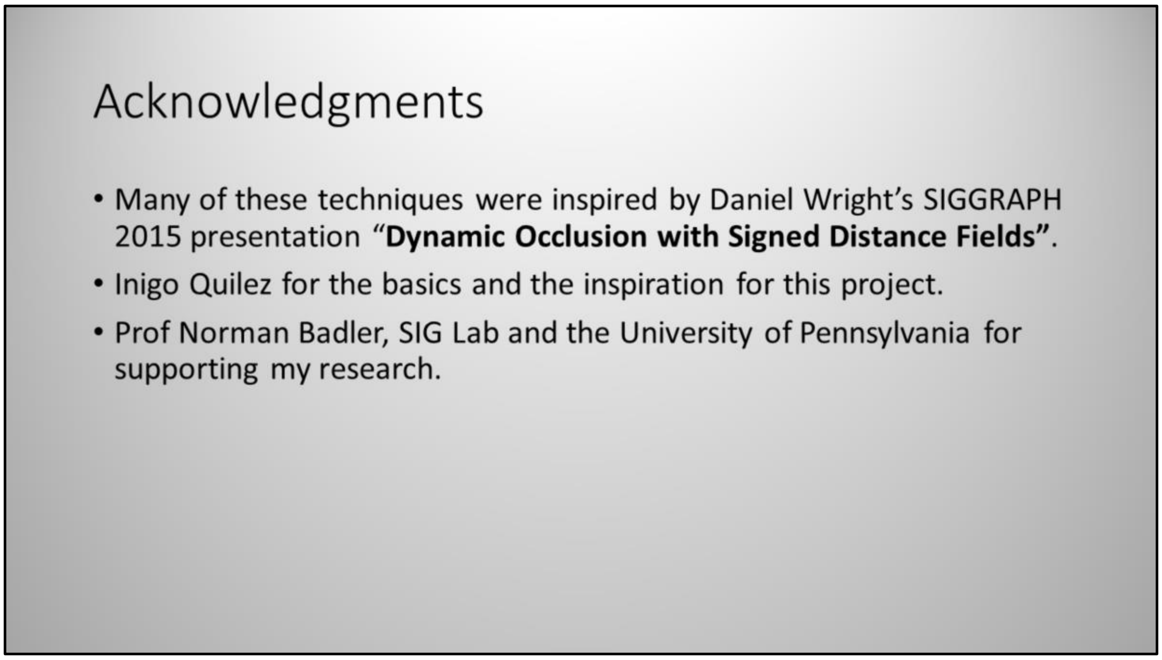
The Future

- Distance fields already supported by UE4
- Still high-end feature
- Hot research topic



Closing thoughts

- Whole source code is available online https://github.com/xx3000/mTec
- There are a lot more details and techniques than I could cover. If you are interested don't hesitate to ask or check out my full thesis in the same repo.
- There are a lot of resources online for getting into distance fields, and nice looking results can be achieved fast. (Inigo Quilez' Blog, Shadertoy, etc.)



Acknowledgments

- Many of these techniques were inspired by Daniel Wright's SIGGRAPH 2015 presentation "**Dynamic Occlusion with Signed Distance Fields**".
- Inigo Quilez for the basics and the inspiration for this project.
- Prof Norman Badler, SIG Lab and the University of Pennsylvania for supporting my research.

Thank You!

And that is how you render the world efficiently with a single triangle.