Real Time VFX Mike

This is just a repository for random things I work on in my spare time including some UE4 and Unity 3D projects.

Monday, August 6, 2018

Lucky Bioms: Clouds

To try and make things easy for the design team, the backgrounds in Super Lucky's Tale are template scenes that go on infinitely. The designers can then place down platforms and art without worrying too much about what is going on in the background. These template scenes are usually a collection of assets that follow the camera around and use world space shaders and whatnot to make a procedural-ish, never ending background. In the Lucky Bioms series I'll go over a few of these background templates and explain an bit about how they were made.

The first biom I made was the clouds. This biom is in one of the early public demos for Super Lucky's Tale. The main reference was the Mario Kart 8 Cloudtop Cruise race course, which looks really cool but is static. Static in Mario Kart is not so bad since you are racing around the course and stuff is usually flying past you pretty quickly. But for the slower pace of Lucky's Tale Things need to have a little motion all the time so a tessellated displacement shader was the way to go.

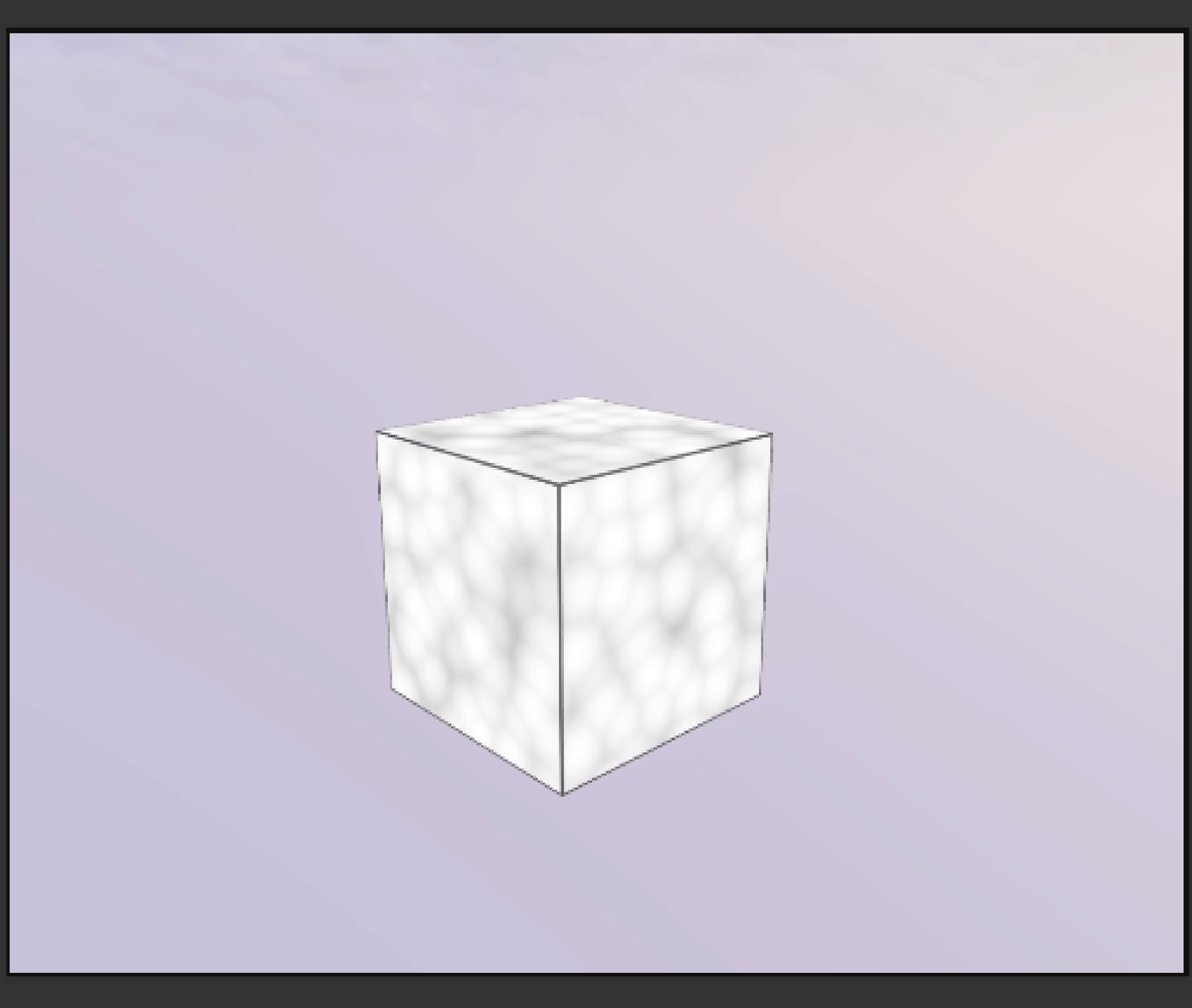
3D Tiling Worley Noise

To make puffy clouds you could use a couple of displacement maps and scroll them on top of each other but you would end up seeing the pattern and may end up looking a bit meh. So I decided to try out 3D Worley noise. It's expensive to calculate Worley noise in 2D in a shader so 3D would have been even worse, and with the performance budget I thought it best to store the noise in a 3D texture. Using textures often produces softer results as opposed to calculating each pixel and since Lucky's Tale has a soft look it was even more reason to store the displacement in a texture.

Now I don't know much about Worley noise or making it tile but luckily I found this shader toy by David Hoskins which covered most of the work. From there it was pretty simple to extend it to 3D.

```
float worley3d(float3 p) {
 float d = 100.0;
 for (int xo = -1; xo <= 1; ++xo){
  for (int yo = -1; yo <= 1; ++yo){
   for (int zo = -1; zo <= 1; ++zo){
    float3 tp = floor(p) + float3(xo, yo, zo);
    d = min( d, length( p - tp - noise3d(tp) ) );
   }
  }
 }
 return cos(d); // use cosine to get round gradient
 //return 1.0 - d;
}
```

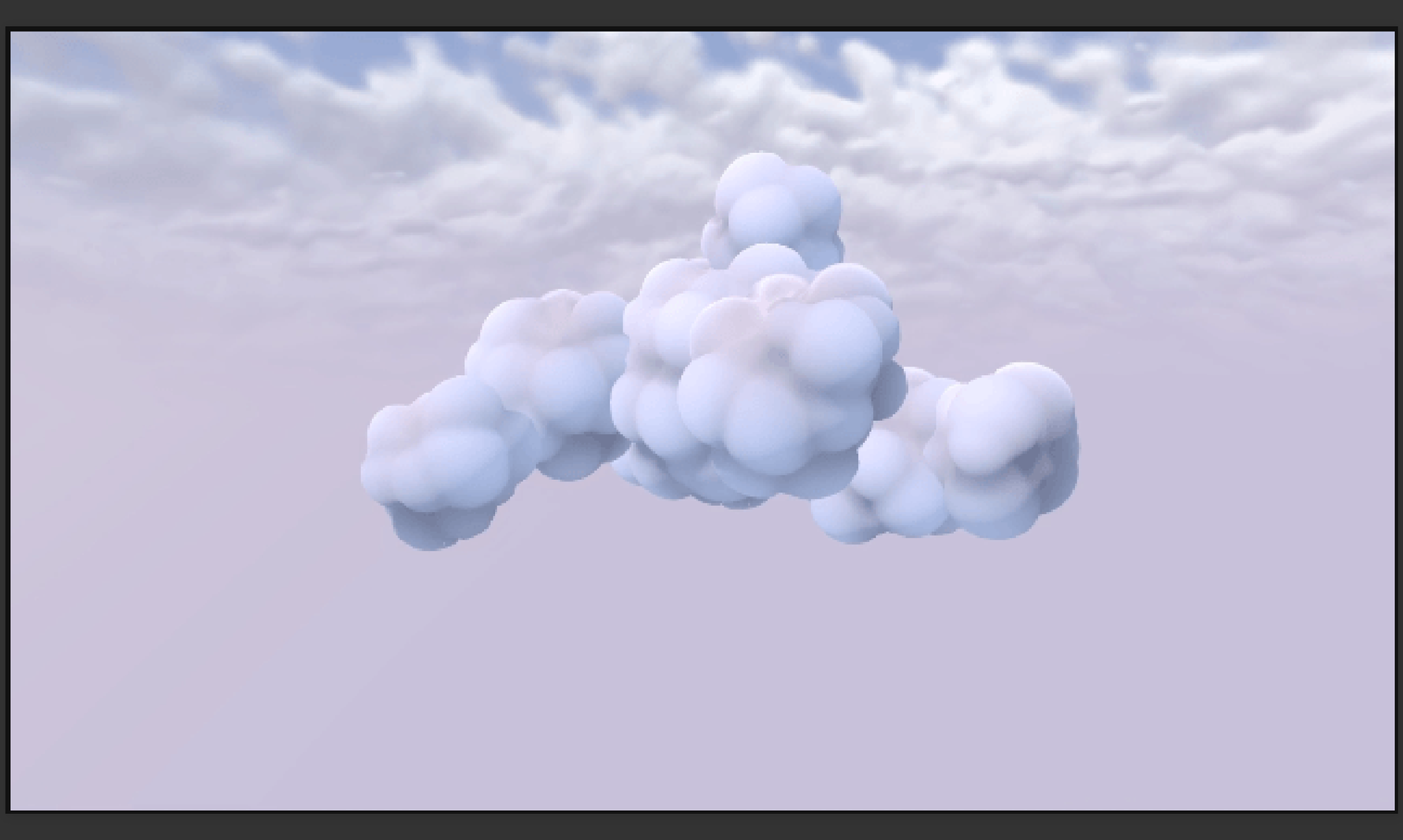
The first implementation was in C# script and took about 15 seconds to generate a 32x32x32 noise texture, it was also a bit harsh on the edges where cells met. I moved the noise generation to a compute shader and could generate a 128x128x128 noise texture with 5x5x5 super sampling instantly! Below is a shader visualizing the noise texture in world space. As the cube is dragged around you can see how cells appear and disappear.



Cloud Shader

The noise texture is mapped to surfaces using world space coordinates and is scrolled "through" a surface rather than across it. As the surface samples different layers of the noise, the cloud cells expand and shrink in a way that scrolling 2D textures just can't replicate.

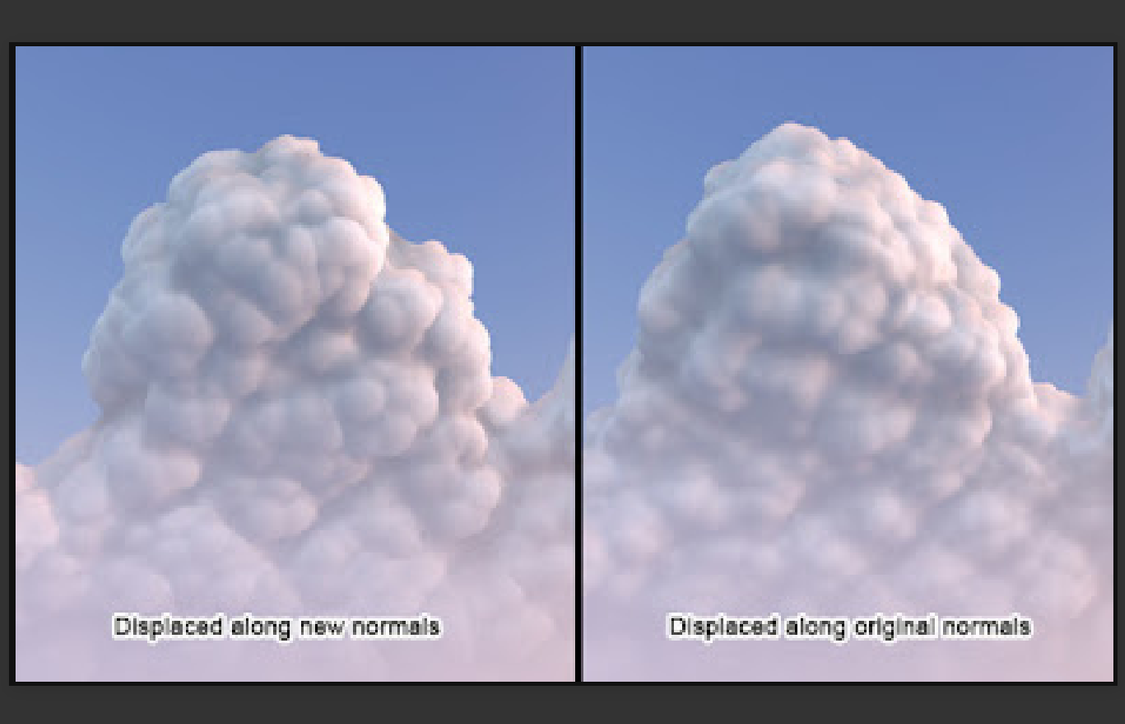
One thing about displacing geo with a shader is that the normals aren't changed to reflect the displacement. So to fix this, tangent normals are generated in the vertex shader by sampling the noise map a little bit in the world tangent and world bi-normal direction and using the difference to make tangent normals. These new tangent normals are then passed through to the pixel shader where they are treated like a tangent space normal map. The normals are not super accurate but they were nice and soft and fit well with the soft cartoony look of the game.



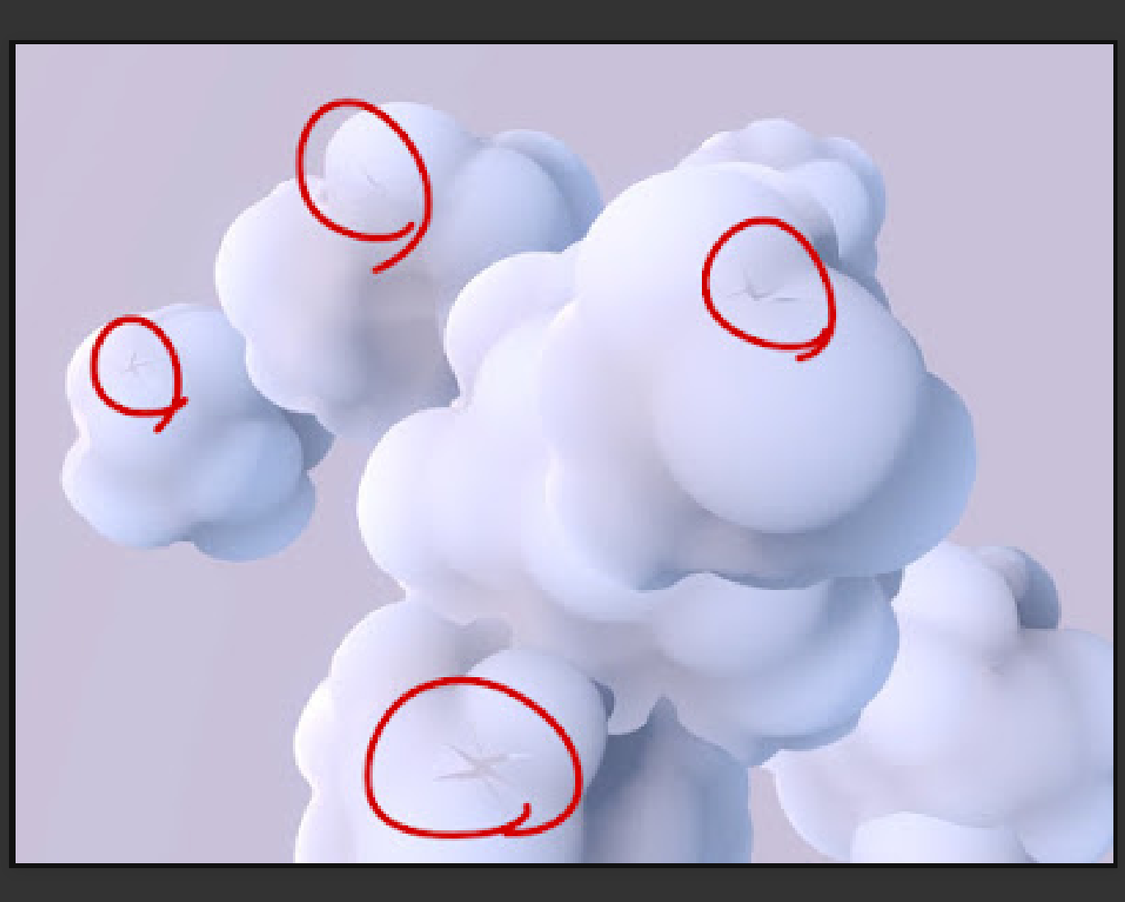
This is the displacement function from the cloud vertex shader. This builds the tangent normals from the displacement map, adds the displacement values together and updates the world normals and occlusion. The tangent normals and occlusion gets passed to the pixel shader in the vertex color.

```
void displaceStuff ( inout float3 worldPos, inout float3 worldNormal, float3 worldTangent, float3 worldBinormal, inout float3 localNormal, inout float occlusion, float tile ){

 // Main tex coords for cloud displacement
 float3 texCoords = worldPos.xyz * 0.01 * _Tiling1.xyz * tile + _Time.y * _Tiling2.xyz;
 float4 disp = tex3Dlod (_WorleyNoiseTex, float4( texCoords, 0) );

 // get the coords for the tangent and binormal displacement
 float3 texCoordsTan = texCoords + normalize( worldTangent.xyz ) * 0.05;
 float3 texCoordsBiNorm = texCoords + normalize( worldBinormal.xyz ) * 0.05;

 // sample the displacement for the tangent normal
 float4 dispTan = tex3Dlod (_WorleyNoiseTex, float4( texCoordsTan, 0) );
 float4 dispBiNorm = tex3Dlod (_WorleyNoiseTex, float4( texCoordsBiNorm, 0) );

 // get tangent normal offset from displacements
 float2 localNormOffset = float2( disp.x - dispTan.x, disp.x - dispBiNorm.x );

 // scale the normal by the one over the tiling value
 float oneOverTile = ( 1.0 / tile );
 localNormal.xy += localNormOffset * _BumpIntensity * oneOverTile;
 occlusion *= pow( disp.x, oneOverTile );

 // set up the tSpace thingy for converting tangent directions to world directions
 float3 tSpace0 = float3( worldTangent.x, worldBinormal.x, worldNormal.x );
 float3 tSpace1 = float3( worldTangent.y, worldBinormal.y, worldNormal.y );
 float3 tSpace2 = float3( worldTangent.z, worldBinormal.z, worldNormal.z );

 // update the world normal
 worldNormal.x = dot(tSpace0, localNormal);
 worldNormal.y = dot(tSpace1, localNormal);
 worldNormal.z = dot(tSpace2, localNormal);
 worldNormal = normalize( worldNormal );

 // push the world position in by the average value of the displacement map
 worldPos -= worldNormal * 0.7937 * _Displacement * oneOverTile;
 // push the world position out based on the new world normal
 worldPos += worldNormal * disp.x * _Displacement * oneOverTile;

}
```

The displacement happens along those new normals to make the clouds puff out more instead of displace straight up.

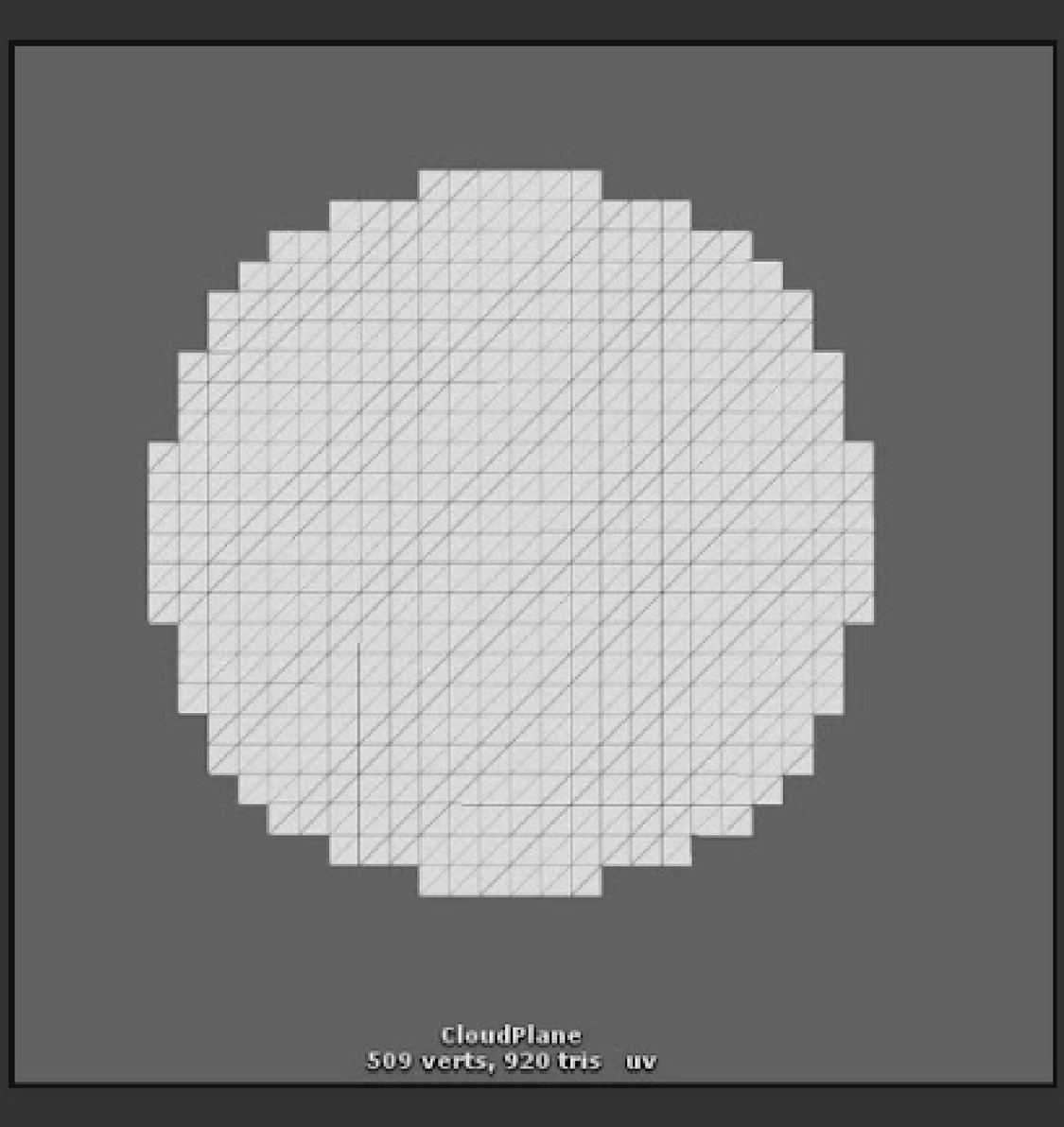


This unfortunately splits the mesh on texture seams so you have to hide the seems of anything in the world you want to put the shader on.

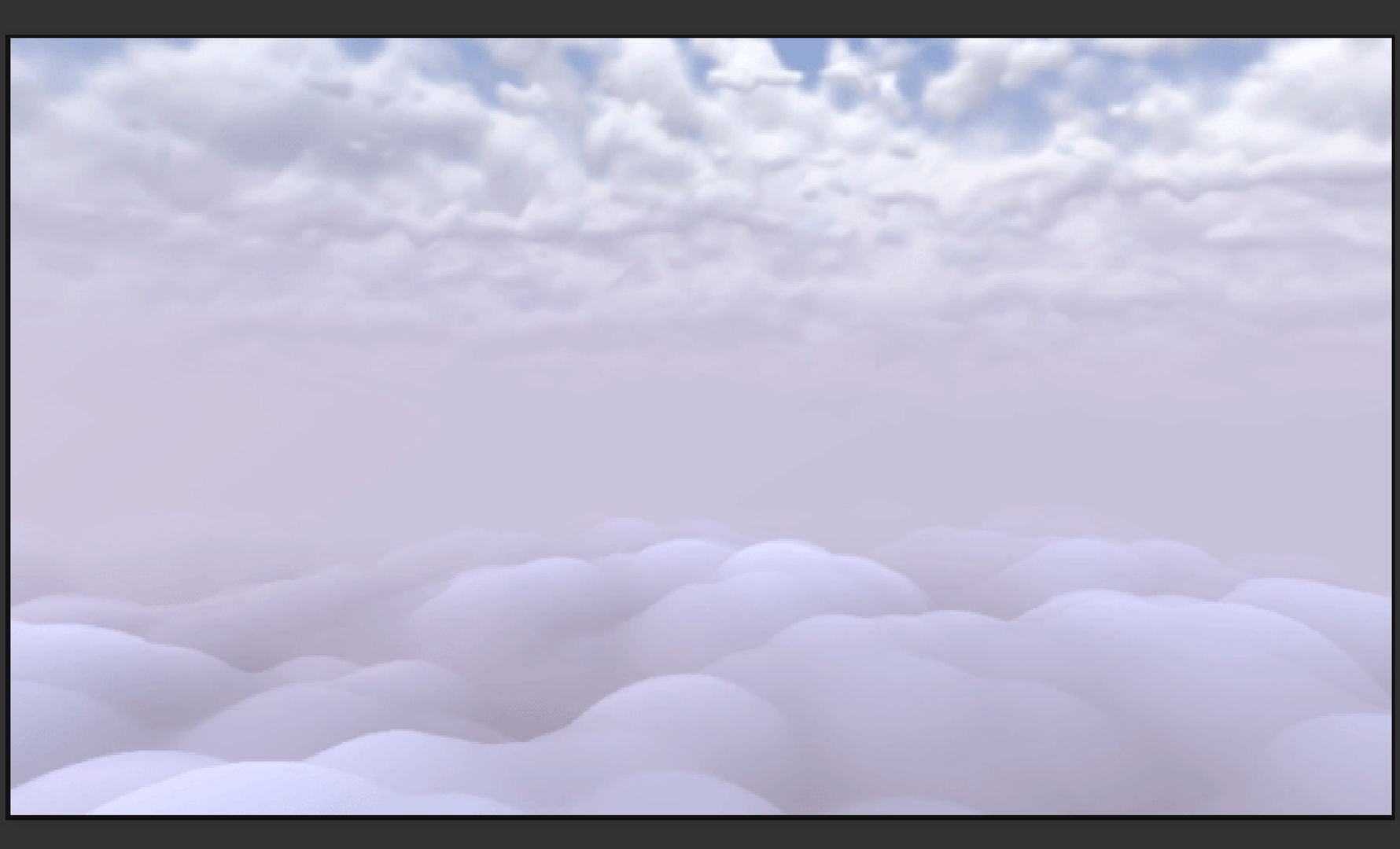


The Cloud Plane

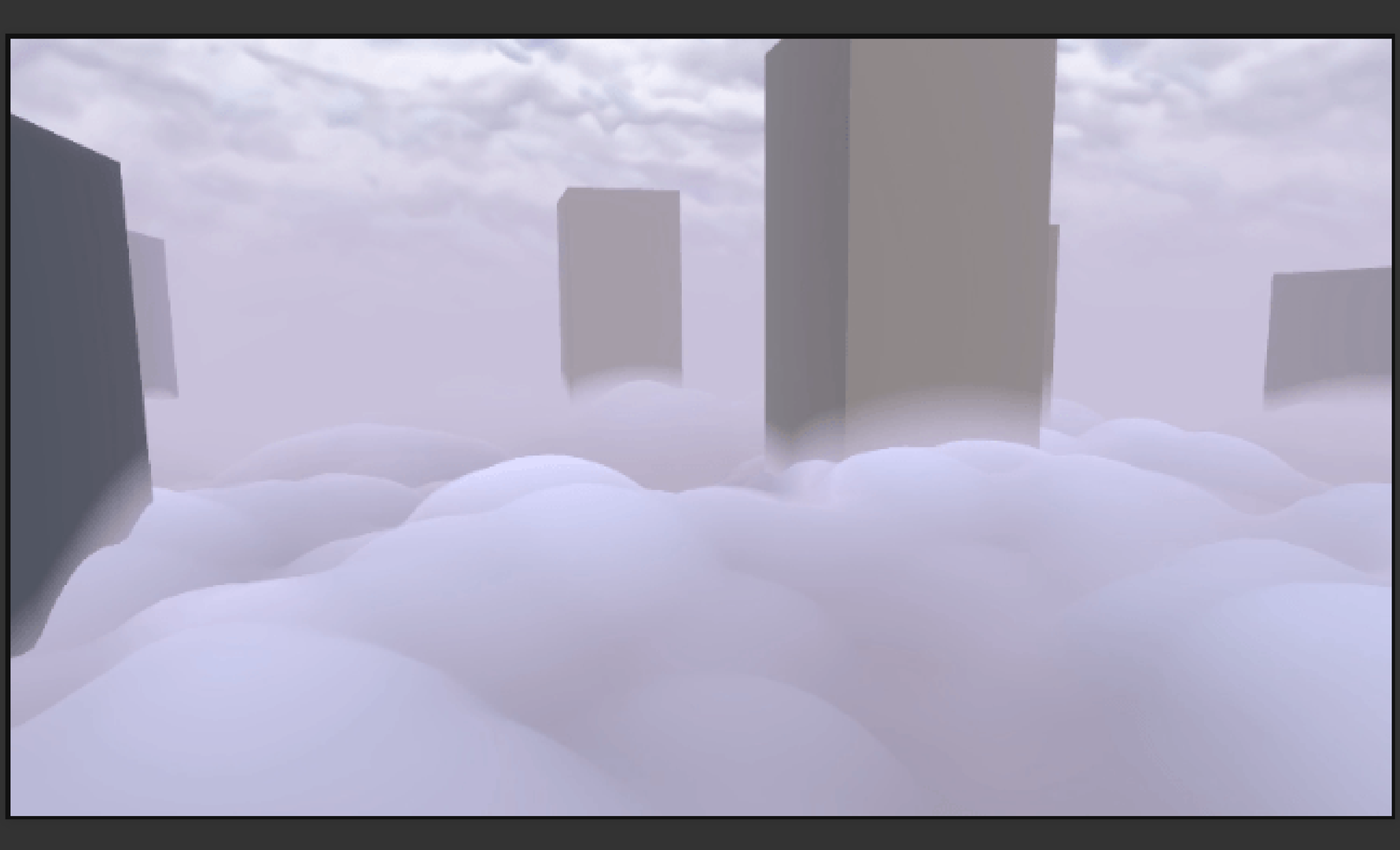
The cloud ground plane is a round mesh made up of 1x1 meter squares.



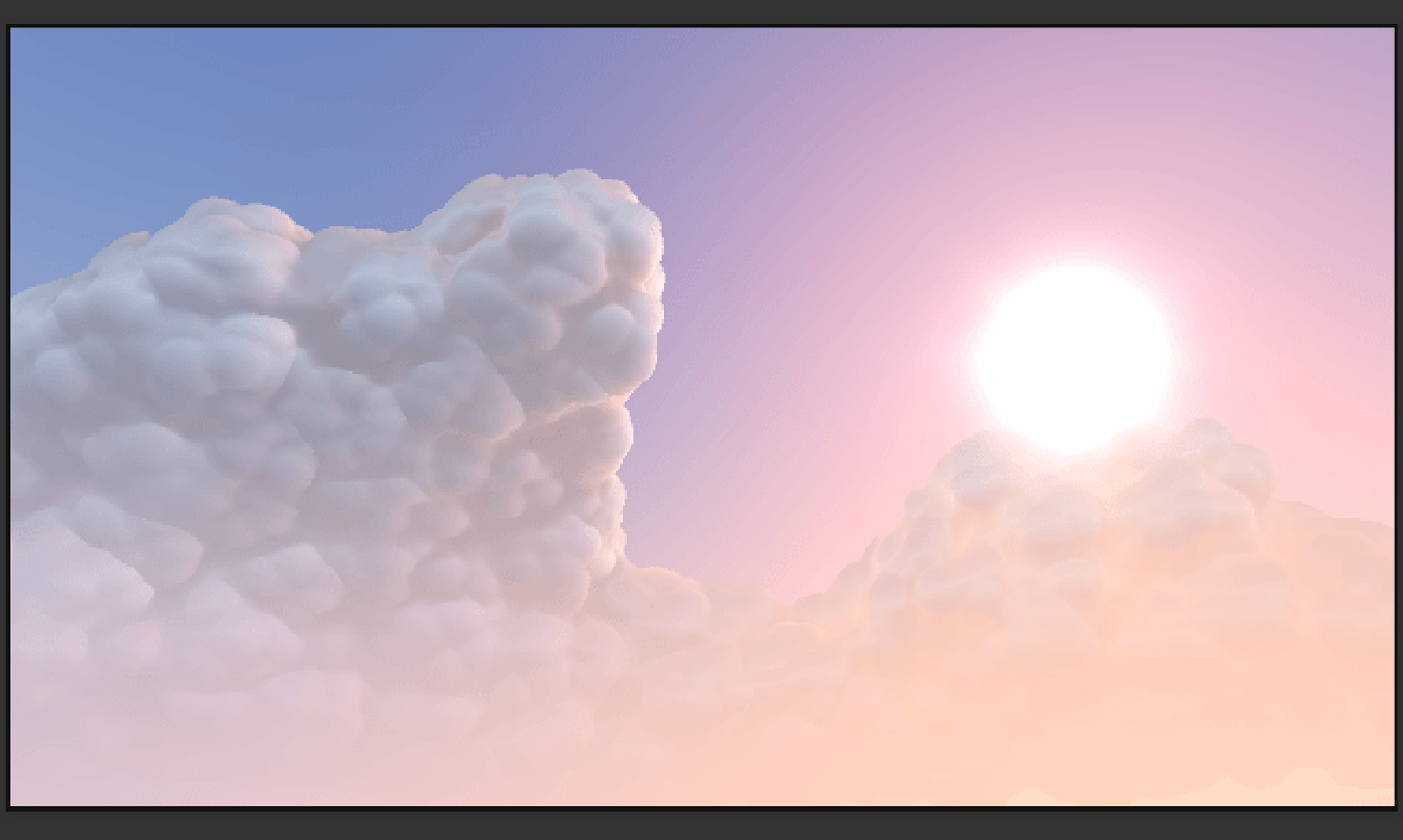
This mesh follows the camera on the x and z with its position being snapped to the closest meter. This ensures that while the mesh is following you you never see it pop abruptly into place because the topology doesn't change as it moves.



So obviously this mesh can't be tessellated and stretch off into the distance as that would be too expensive. So instead the 3D Worley noise map is used in the global fog function and is sampled at the level that the ground plane is at. Then the approximate height of the displaced cloud plane can be figured out and objects beyond the cloud plane mesh can be faded to the solid fog color where the cloud plane would have intersected them. The Cloud plane itself fades to the solid fog color at a distance so you rarely see the edge of it.



There is another shader for background clouds that supports transmission from the sun light and has a simpler pixel shader. This shader also has a bit of code that pushes it back towards the camera far plane to keep it from drawing in front of the world.



Posted by Unknown at 2:34 PM

2 comments:

iLeoPod April 6, 2019 at 1:50 PM

Awesome!, Is there a way i can get a copy if this shader, or a simpler version of it if is possible?? Thank You!

Reply