Shadow Mapping

A texture-based solution for real-time shadows

Shadows

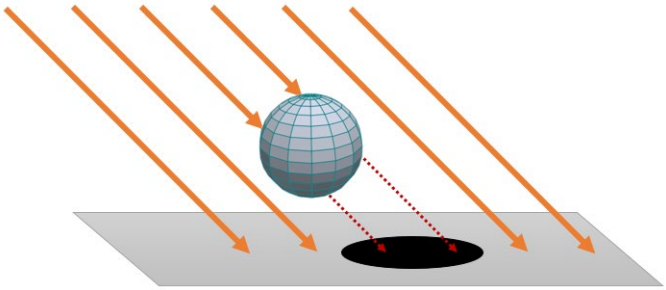
Shadows are caused by light being blocked before it reaches a particular surface. Our lighting equations, however, only compare the light's *direction* and the current surface's *normal*; they do not consider any objects *between* the surface being rendered and the light.

If we want shadows, we'll need a way to determine if an object sits between the current pixel and a particular light source. If this light is in fact being blocked, it should not contribute to the total light of the current pixel. However, light from other light sources might.

> *This means that this "blocking information" needs to be gathered separately for <u>each light</u> that should cast shadows! In many games, the primary light source (often the one that represents the sun) is the only one that casts shadows, even if there are other lights in the scene.*

Visibility: A Brute-Force Approach

To implement shadows, we need to know if light from a particular light source will strike the current pixel we're rendering. In other words, we need to know the *visibility* of the surface from the light source. How could we make this determination?



The most straightforward (and least efficient) way would be to *check everything*. If, in a pixel shader, we had access to every single triangle of every single object in the scene, we could theoretically check to see if any of those triangles block the light to this pixel.

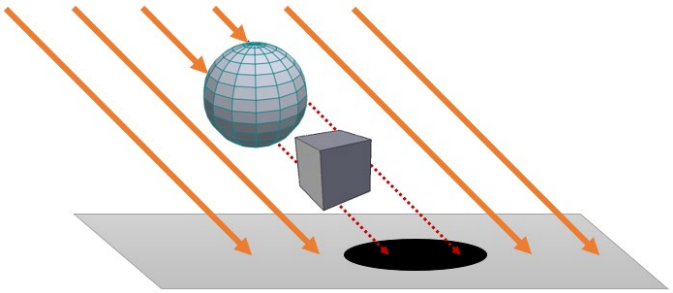
Unfortunately, in a complex scene, this would require upwards of hundreds of thousands or even millions of ray-triangle checks *per pixel*. *For each object*. *Every frame*. Additionally, we simply can't access "every single triangle of every single object in the scene" in a pixel shader. That is potentially a massive amount of data, and constant buffers – our main mechanism for passing arbitrary data to a shader – have size limitations.

As you can imagine, this brute force approach simply isn't feasible. We need a different approach to establish the visibility of a surface from a particular light source. Ideally, our approach will store visibility information in a way that is easily and efficiently accessible in a pixel shader.

> *Generally speaking, any idea that starts with "I'll just loop thousands of times in my pixel shader" probably isn't viable for real-time rendering.*

Simplifying Visibility

In the brute force approach described above, we'd have to check every single triangle in the scene because we have no way of knowing which ones are important. To simplify this a bit, consider that we don't actually need to find *all* of the triangles between the light and the surface; we just need the ones that are *closest to the light* along each "light ray". Furthermore, we don't need to record which *triangle* actually blocked the light, just its *distance from the light*.

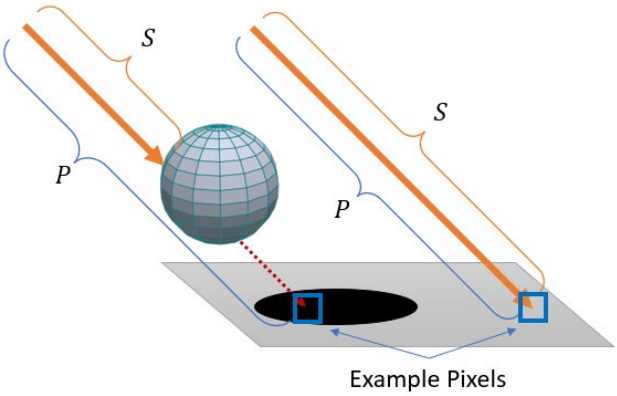


In the diagram to the right, the cube has no effect on the shadow being cast on the ground since the cube is entirely "behind" the sphere. The closest surface to the light – the sphere's triangles – are all we need. Another way to describe this is that the light "can't see" the cube.

Another potential improvement: we can reuse this visibility information for all objects we intend to draw (and, therefore, all pixels we need to shade) this frame. For example, in the diagram above, the distances of the sphere's triangles are useful for shadows on both the cube and the ground.

Using Visibility Information

If we capture this visibility data – the distance to the closest surface along each "light ray" – at the beginning of each frame, we can use it when rendering all of our entities. Since each pixel we're shading may or may not be in shadow, we'll be doing our final shadow determination a pixel shader. But how does the "distance from the light" help us? What do we actually do with this data?



Consider the two example pixels at the bottom of the diagram to the right. P is the distance from the light source to that pixel. S is the distance to the closest surface along the same ray.

When $S = P$, the current pixel *is* the closest surface to this light. Therefore, this light should be included in this pixel's shading.

When $S < P$, the current pixel is *not* the closest surface to this light, so this light should not contribute to the current pixel's shading.

That's the basic premise of our shadow determination: is the current pixel the closest surface to the light along this particular light ray? This still leaves some lingering issues: how do we "gather" visibility data and where do we store it? Let's look at our overall requirements for our potential shadow implementation.

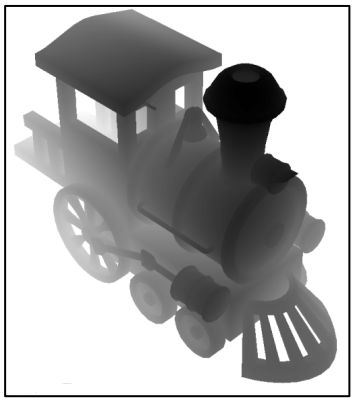
Requirements

Based on the details above, here are our current shadow requirements:

- For a given light, we ultimately just need the distance to the closest surface along each "light ray".
- As this will be lot of data, gathering and storing it needs to be efficient.
- This data should be stored in a structure that a pixel shader can easily read.
- This data is useful for *all* objects rendered this frame, so we need it *before* rendering anything.

Efficiently processing lots of geometry and storing the results in a GPU-friendly way? That sounds a whole lot like what's happening every time we render. Can we somehow use the GPU for this work? As you may recall, there's already a mechanism in the rendering pipeline that tracks "closest surfaces".

Hey, That's a Depth Buffer!

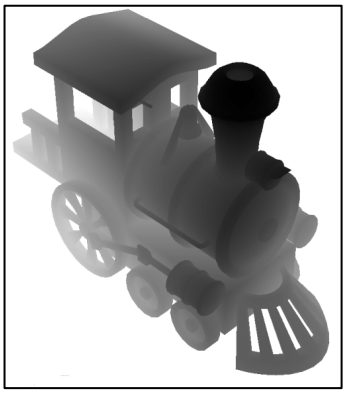


A depth buffer is a texture that stores scalar "depth information": the distance of the closest surface along each "ray" from the camera, in the range [0,1]. (Technically, the distance between the camera's near and far planes.) To the right is an example depth buffer: lower values (black) are closer to the camera, while higher values (white) are farther from the camera.

Each time we render an object, each rasterized pixel gets checked against the corresponding pixel's depth in the depth buffer. If the new pixel is closer than the existing depth buffer data, the pixel is shaded and the depth buffer is updated accordingly. If the new pixel is farther, it gets culled (no shading occurs).

This is the exact information we need for shadows! It's also quite efficient to create, since the GPU does most of the work. And the results are stored in a texture, which is easily accessible in another shader.

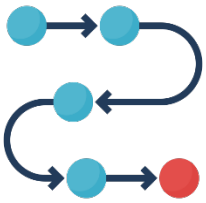
Enter the Shadow Map



A texture that stores "closest distance" information for shadows is called a *shadow map*. Each of its pixels holds a single float value, representing the closest surface along a vector (or "ray") from the associated light source into the scene. To the right is an example of a shadow map. Look familiar?

Since a shadow map really just needs to hold a single float value, it can double as a depth buffer. In other words, our shadow map texture can *sometimes* be a depth buffer and can *sometimes* be a texture resource read by a pixel shader. The only caveat is that it cannot be both at the same time.

Shadow Map Workflow



As part of our overall rendering work each frame, we'll first need a phase that *writes* to the shadow map, populating it with up-to-date shadow information based on the transformations of a shadow-casting light and all scene objects. This must occur before rendering to the *screen*, as we need the shadow map when shading entities.

> *If you have multiple shadow-casting lights, each must have its own shadow map! This means each must be "rendered into" independently. As you might imagine, this isn't exactly cheap.*

Given this, our overall rendering work for a single frame will now be:

- Clear render target, depth buffer and shadow map(s)
- For each shadow-casting light:
  - Render the *entire scene* (all objects), using this light's shadow map as a depth buffer
- Activate our "standard" depth buffer
- Render all opaque objects to the screen, using shadow map(s)
- Render the sky
- Render all transparent objects, if any, to the screen (sorted properly)
- Present the results

Creating a Shadow Map

A shadow map is a texture, which we can create manually through Direct3D. However, to bind a texture to the pipeline, as either input (a shader resource) or output (a depth buffer), we need associated *views*: a *shader resource view* and a *depth/stencil view*. With these, we can use the shadow map in either situation while rendering. Implementation details for all of these can be found later in this document.

One of the first things we need to decide when creating a shadow map are the dimensions of the actual texture. Shadow maps are most often square and each dimension should be a power of two for GPU efficiency, but their exact size depends on several factors. The larger the shadow map, the more detail it holds, so the more accurate our shadows will be. The smaller the shadow map, however, the faster it is to render into (as there are fewer pixels to rasterize). A decent size to start with is 1024x1024, adjusting based on performance and quality.

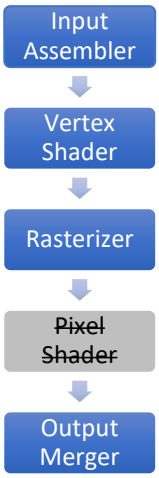
Rendering to the Shadow Map

How do we get data *into* the shadow map? Recall that we need the *depth* of each pixel. We're already calculating this in our vertex shader: after applying the projection matrix, the resulting vector's Z value is "depth". The GPU automatically populates the active depth buffer with this value as we render.

Therefore, we need to render each object in the scene using the shadow map as our depth buffer. Implementation-wise, this is another loop in which we draw all of our entities. Before this loop, we change the active depth buffer to be our shadow map so the visibility information is recorded there. Note that the visibility we need is from the *light's perspective*, not the player's perspective. As such, we'll need to render from the *light's point of view* (more on this below) instead of our standard camera.

Shadow Shaders & The Pipeline

We're invoking the rendering pipeline to get the GPU to record the visibility data into our fancy new shadow map. This means we'll need shaders. However, our existing shaders are mostly concerned with calculating a color, which is irrelevant for visibility; we just want *depth*.

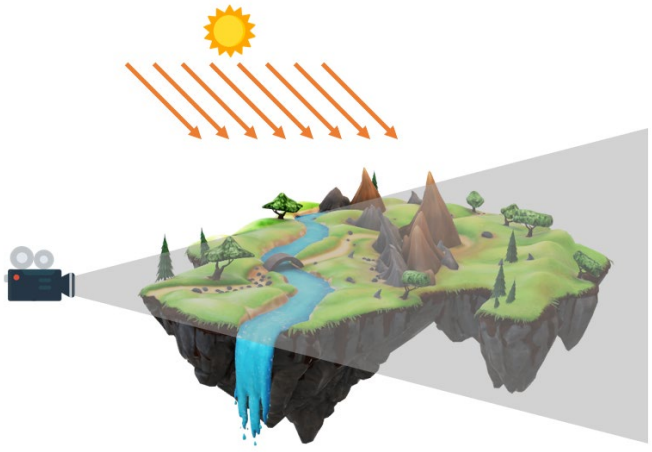


Instead of just reusing our rather expensive shaders for this, we can create new ones that perform the minimum amount of work necessary. We're only concerned with *where the surface is* at this point, not its color, so a new "shadow vertex shader" only needs to handle world/view/projection matrices and nothing else. No surface information – like normals, tangents, or UV coordinates – required.

Additionally, since we don't need color information at all for the shadow map, we can *turn off the pixel shader* and color render target (the back buffer) entirely! Even with no pixel shader or render target, the pipeline will still process depth information and write to the depth buffer. This makes our "shadow map render" phase much more efficient.

Points of View: Camera vs. Light

The data we need is the depth of each object from the *point of view of the light*. But when we render, we're doing so from the point of view of our 3D *camera*.



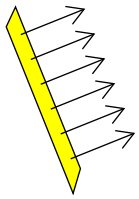
How do we reconcile this? We simply don't involve our regular camera at all. Instead, we need a new set of camera matrices – view and projection – that match the *light*.

This "light view matrix" will use the light's position and orientation, while this "light projection matrix" must correspond to the light's bounds. If we use these new view & projection matrices instead of our standard camera's matrices when rendering the scene, it'll be as if we're looking at the scene through the light itself. Thus, our shadow map will contain the depth of each pixel from the light's point of view.

Shadow Maps & Light Types

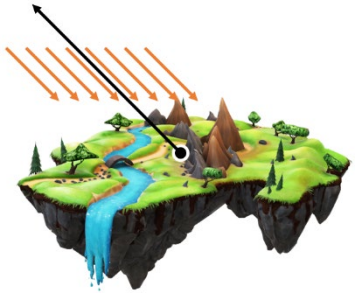
Each type of light – directional, point and spot – requires a different setup for its view and projection matrices. Directional lights are most often used as the main light source in a scene, so that's the one we'll focus on, though we'll briefly look at point and spot light matrix details.

Directional Lights



Directional lights have a simple definition: just a direction in space, no position necessary. This complicates our light view matrix setup slightly, as we must have a position when creating our light's view matrix. Since there is no inherent position, we'll have to come up with one.

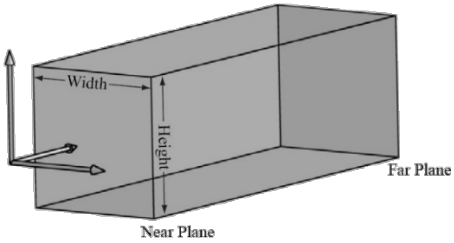
View Matrix



If our shadow map needs to see the whole scene, then it should probably be focused on the center of that scene. The position of our directional light's view matrix, then, can be determined by starting at the center of the world and backing up in the negative direction of the light. This idea can be seen in the diagram to the right. Now we have a suitable position and direction to create the view.

Projection Matrix

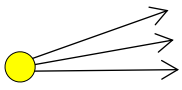
The projection should match the "shape" of the light. All light from a directional light is parallel, so a perspective projection isn't ideal here.



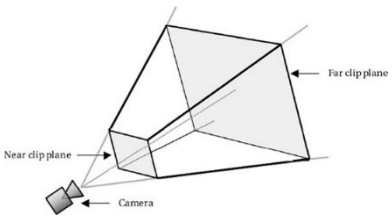
Instead, we should use an orthographic projection, as seen to the left. In an orthographic projection, all rays are parallel rather than spreading out as they travel. The width and height of the projection determine the size of the projection (how much of the world can be "seen"). The larger the projection, the more of the world is covered by the shadow map. As the shadow map is square, the width should be equal to the height here.

Increasing the projection's size does not inherently increase the *resolution* of the shadow map, however, so shadows will appear coarser as the projection size increases. More on this later.

Spot Lights

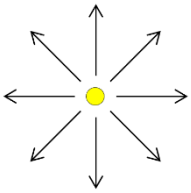


Creating a view matrix that matches a spot light is easy: spot lights are defined with both a position and a direction, so just plug those in when creating the view matrix.



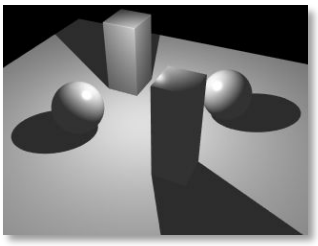
Since light rays spread out as they travel from a spot light, the projection's shape will also need to represent rays spreading out as they travel. This is accomplished with a perspective projection. Its field of view must match the spot light's cone angle. As shadow maps are generally square, the projection's aspect ratio should be 1.0.

Point Lights



Point light shadows are entirely possible, but are much more complex. As such, their full implementation is outside the scope of this document.

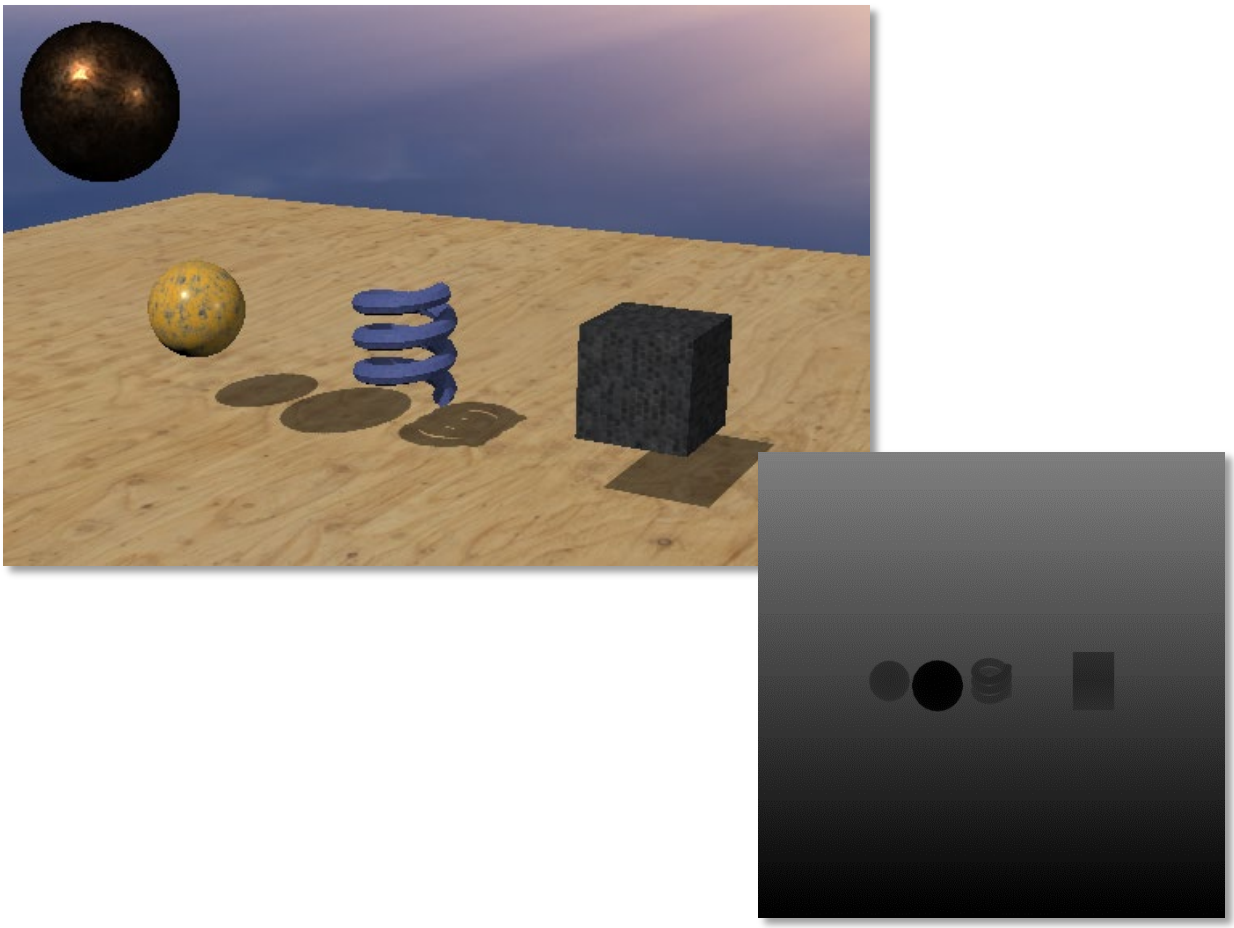
Recall that point lights emanate light in all directions. This means we need a shadow map that has shadow information in "all directions". What kind of texture can represent information in "all directions"? A cube map.



Thus, the shadow map for a point light is actually an entire cube map. Rendering into a cube map generally entails rendering into each face successively (one at a time), which means *each shadow-casting point light* requires rendering the scene *six times*, once for each face of the cube map. While object culling can help speed this up, this is still a significant amount of work per light.

Shadow Map Example

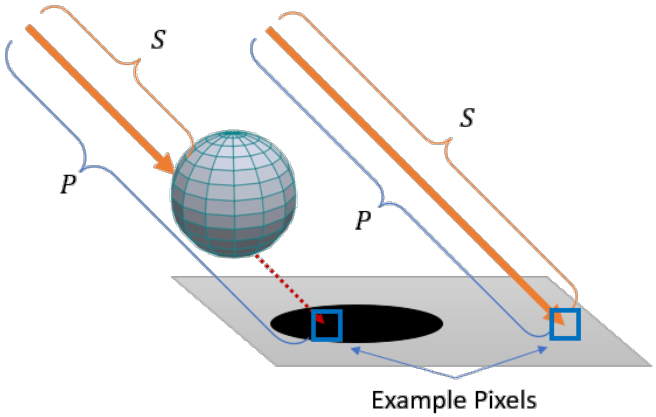
Below is an example of a scene and the corresponding shadow map for the same frame. The shadow-casting directional light is angled downward at 45°. Darker colors in the shadow map represent surfaces that are closer to the light, such as the bottom of the texture (the floor) and the floating bronze sphere.



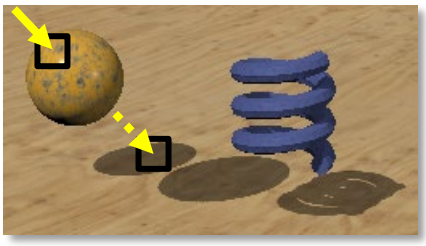
Using a Shadow Map

The actual determination of whether or not a surface is in shadow is handled in the pixel shader we use when rendering that surface. For us, this means we'll need to add the shadow map as another texture in our standard pixel shader. But then what? How do we know where to sample it?

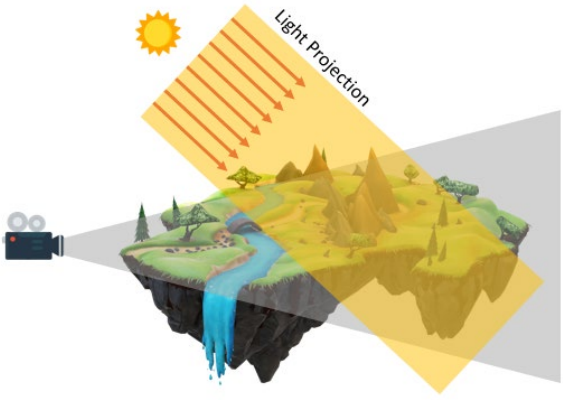
Recall the diagram to the right. We'll need both the distance from the light to this pixel, P, and the distance from the light to the closest surface, S, along the same ray. That "ray" corresponds to a single pixel in the shadow map. That's what the pixels in the shadow map are: they're the "rays" coming out of our light when we rendered from its point of view.



To properly sample the shadow map, we need to know where our current pixel was *from the point of view of the light.* When we render the surface to the screen, however, we only know where the pixel is from the point of view of our camera. Since we had to use the light's view and projection matrices when *rendering to* the shadow map, we'll also need them when *sampling from* the shadow map. Below, the same location is highlighted on both the shadow map (left) and in the world (right).

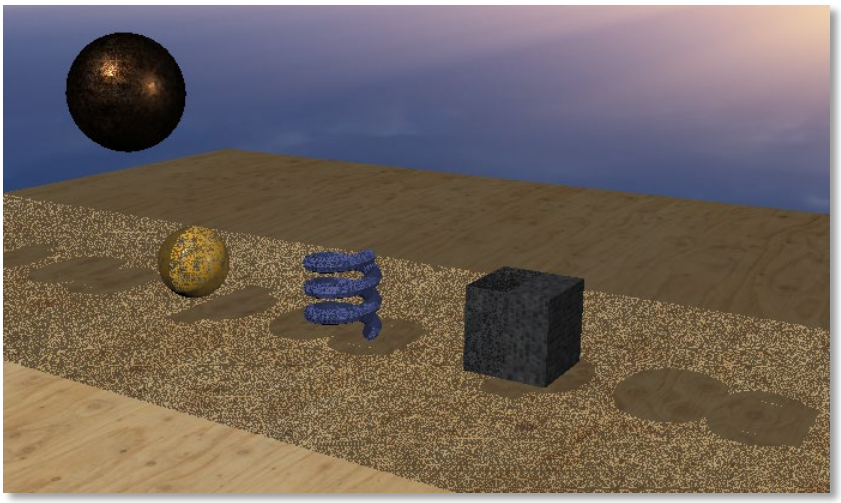


We'll need to pass the light's view and light's projection matrices in to our standard vertex shader when rendering our entities to the screen. Normally, we perform $world * view * projection$ to determine where a vertex is on the screen. We now also need to determine where it was in the shadow map, using $world * lightView * lightProjection$. This second result will then be passed down to the pixel shader. There, we'll convert it to appropriate UV coordinates and sample the shadow map to get S, the distance to the closest surface from the light. Since the result of applying the projection matrix includes a depth, too, we now have the distance of the current pixel from the light, P, as well. We then compare P and S to determine if our screen pixel is in shadow. Details of all of these steps can be found later in the next section.



Shadow Mapping Implementation

Below are implementation details for what we've discussed so far. Note that, while this will get the basics up and running, ***there will be problems!*** As seen below, the shadows aren't quite right, but we can see hints of the correct shadows, along with some unwanted repetition and artifacts.



After covering the basic implementation details, we'll solve the problems one by one.

Required Data

The following variables will track the data required for our basic shadow map implementation. We'll need to add a few more later to fix some of the aforementioned issues.

```
Microsoft::WRL::ComPtr<ID3D11DepthStencilView> shadowDSV;
Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> shadowSRV;
DirectX::XMFLOAT4X4 shadowViewMatrix;
DirectX::XMFLOAT4X4 shadowProjectionMatrix;
```

Creating the Shadow Map Texture

Creating a shadow map is like any other GPU resource: Fill out a description and have D3D create it.

```
// Create the actual texture that will be the shadow map
D3D11_TEXTURE2D_DESC shadowDesc = {};
shadowDesc.Width  = shadowMapResolution; // Ideally a power of 2 (like 1024)
shadowDesc.Height = shadowMapResolution; // Ideally a power of 2 (like 1024)
shadowDesc.ArraySize = 1;
shadowDesc.BindFlags = D3D11_BIND_DEPTH_STENCIL | D3D11_BIND_SHADER_RESOURCE;
shadowDesc.CPUAccessFlags = 0;
shadowDesc.Format = DXGI_FORMAT_R32_TYPELESS;
shadowDesc.MipLevels = 1;
shadowDesc.MiscFlags = 0;
shadowDesc.SampleDesc.Count = 1;
shadowDesc.SampleDesc.Quality = 0;
shadowDesc.Usage = D3D11_USAGE_DEFAULT;
Microsoft::WRL::ComPtr<ID3D11Texture2D> shadowTexture;
device->CreateTexture2D(&shadowDesc, 0, shadowTexture.GetAddressOf());
```

A few notes about the code above. First, the local "shadowTexture" variable: we'll ultimately just need a Shader Resource View (to sample from the texture) and a Depth Stencil View (to use it as a depth buffer), so we won't need to keep the texture pointer itself after creating those views. Hence why it's just a local variable.

Second, there are some non-obvious values we use for the description members above:

- **Width** and **Height:** Shadow maps are generally square, and powers of two are efficient for the GPU. The exact size is up to you, though having it parameterized means easily testing variations.
- **BindFlags:** A logical OR to denote we'll use it as both a shader resource and a depth buffer.
- **Format:** We haven't seen R32_TYPELESS as a format before. The R32 means "reserve all 32 bits for a single value" and TYPELESS means "exact numerical details are left up to the views".

Creating the Associated Views

In addition to the actual texture resource, we need *views* to actually bind it to the pipeline. As these are necessary for drawing, they'll need to be stored for later:

```
Microsoft::WRL::ComPtr<ID3D11DepthStencilView> shadowDSV;
Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> shadowSRV;
```

The creation of these views follows. It is assumed these occur directly after texture creation.

```
// Create the depth/stencil view
D3D11_DEPTH_STENCIL_VIEW_DESC shadowDSDesc = {};
shadowDSDesc.Format = DXGI_FORMAT_D32_FLOAT;
shadowDSDesc.ViewDimension = D3D11_DSV_DIMENSION_TEXTURE2D;
shadowDSDesc.Texture2D.MipSlice = 0;
device->CreateDepthStencilView(
	shadowTexture.Get(),
	&shadowDSDesc,
	shadowDSV.GetAddressOf());

// Create the SRV for the shadow map
D3D11_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Format = DXGI_FORMAT_R32_FLOAT;
srvDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
srvDesc.Texture2D.MipLevels = 1;
srvDesc.Texture2D.MostDetailedMip = 0;
device->CreateShaderResourceView(
	shadowTexture.Get(),
	&srvDesc,
	shadowSRV.GetAddressOf());
```

Some details on the specifics of these descriptions:

- **ViewDimension** describes the dimensionality of the resource. In this case it's just a 2D texture.
- **MipSlice** tells the depth view which mip map to render into. We only have 1, so it's index 0.
- **MipLevels** and **MostDetailedMip** tell the SRV which mips it can read. Again, we only have 1.
- **Format** is slightly different for each view (D32_FLOAT vs. R32_FLOAT). These both mean "treat all 32 bits as a single value", but D32_FLOAT is specific to depth views.

Light Matrices

To render from the light's point of view, we'll need view and projection matrices that match the light. The details below assume you're creating a shadow map for a *directional light*. Adjust as necessary for other light types.

Light View Matrix

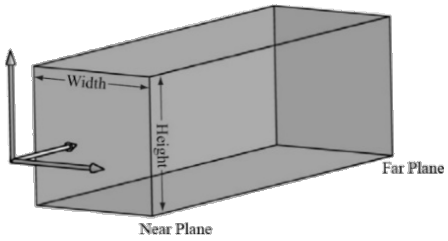
View matrix creation requires three things: a position, a direction and an up vector. The *up vector* is just the world's up vector: (0, 1, 0). The *direction* should match whichever directional light is casting shadows in your scene. *Position* is the trickiest as directional lights don't inherently have a position. As discussed above, picking a point close to the center of your game world (perhaps the origin) and backing up along the negative light direction works well for a simple scene. The exact distance depends on how large your scene is.

```
XMMATRIX lightView = XMMatrixLookToLH(
       -lightDirection * 20,      // Position: "Backing up" 20 units from origin
       lightDirection,            // Direction: light's direction
       XMVectorSet(0, 1, 0, 0));  // Up: World up vector (Y axis)
```

You'll need to recreate this matrix whenever your light's direction changes, including coming up with a new, suitable position.

Light Projection Matrix

To match the parallel nature of light rays from a directional light, the projection matrix should be orthographic. Defining an orthographic projection requires four things: projection width, projection height, near plane distance and far plane distance.



The *near distance* should be a small number. The perfect *far distance* depends on your scene, but like any projection, should be as small as possible. If your shadows seem to end abruptly at a distance, this may need to be increased slightly.

As shadow maps are generally square, the width and height of this projection should be equal. However, the exact values again depend on your scene. The size of this "box" determines how much of the world is included within the shadow map (its *coverage*). For a small, relatively static scene, tweak these numbers until you find the smallest suitable value. Increasing projection size increases coverage, but since the resolution of the shadow map is constant, each pixel is representing a larger and larger area. This has the effect of making shadows appear more blocky (or "pixelated").

```
lightProjectionSize = 15.0f; // Tweak for your scene!
XMMATRIX lightProjection = XMMatrixOrthographicLH(
       lightProjectionSize,
       lightProjectionSize,
       1.0f,
       100.0f);
```

Shadow Map Vertex Shader

This is the vertex shader to use when *rendering to* the shadow map. As noted previously, this only needs to move vertices to the correct place; no shading information is necessary. Since it only returns a single vector (the screen position of the vertex), it doesn't even need an output struct anymore.

```
// Constant Buffer for external (C++) data
cbuffer externalData : register(b0)
{
	matrix world;
	matrix view;
	matrix projection;
};

// --------------------------------------------------------
// A simplified vertex shader for rendering to a shadow map
// --------------------------------------------------------
float4 main(VertexShaderInput input) : SV_POSITION
{
	matrix wvp = mul(projection, mul(view, world));
	return mul(wvp, float4(input.localPosition, 1.0f));
}
```

Shadow Map Render

Before drawing anything to the screen each frame, we'll need to render fresh information to the shadow map. This should occur at the beginning of `Game::Draw()`. The general steps are as follows:

1. Clear the shadow map

This will reset all depth values to 1.0.

```
context->ClearDepthStencilView(shadowDSV.Get(), D3D11_CLEAR_DEPTH, 1.0f, 0);
```

2. Set up the output merger stage

Here we set the shadow map as the current depth buffer and unbind the back buffer, as we don't need any color output.

```
ID3D11RenderTargetView* nullRTV{};
context->OMSetRenderTargets(1, &nullRTV, shadowDSV.Get());
```

3. Deactivate pixel shader

Unbind the pixel shader to prevent pixel processing entirely.

```
context->PSSetShader(0, 0, 0);
```

4. Change viewport

A viewport describes the exact pixel dimensions that the rasterizer uses. Most of the time, this simply matches the screen size, since we want to render to the entire window. However, we're about to render into the shadow map, so the viewport needs to perfectly match the shadow map's resolution.

```
D3D11_VIEWPORT viewport = {};
viewport.Width    = (float)shadowMapResolution;
viewport.Height   = (float)shadowMapResolution;
viewport.MaxDepth = 1.0f;
context->RSSetViewports(1, &viewport);
```

5. Entity render loop

Loop through all scene entities and draw them using the specialized shadow map vertex shader described above. Since we need no material data at all, be sure to **avoid** using the entity's material entirely (as that might activate a different set of shaders).

```
shadowVS->SetShader();
shadowVS->SetMatrix4x4("view", shadowViewMatrix);
shadowVS->SetMatrix4x4("projection", shadowProjectionMatrix);

// Loop and draw all entities
for (auto& e : entities)
{
	shadowVS->SetMatrix4x4("world", e->GetTransform()->GetWorldMatrix());
	shadowVS->CopyAllBufferData();

	// Draw the mesh directly to avoid the entity's material
	// Note: Your code may differ significantly here!
	e->GetMesh()->SetBuffersAndDraw(context);
}
```

6. Reset the pipeline

Change pipeline settings back to prepare to render to the screen once again. This includes resetting the viewport to match the screen and replacing the original render target and depth buffer.

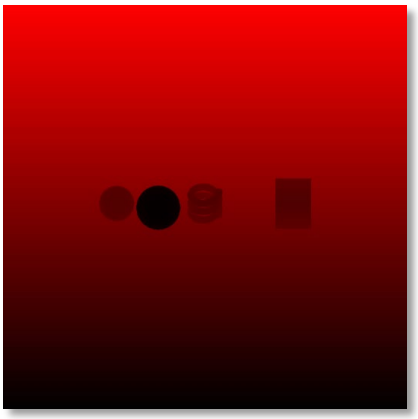
```
viewport.Width  = (float)this->windowWidth;
viewport.Height = (float)this->windowHeight;
context->RSSetViewports(1, &viewport);
context->OMSetRenderTargets(
	1,
	backBufferRTV.GetAddressOf(),
	depthBufferDSV.Get());
```

Checking the Shadow Map

At this point, there should be data in the shadow map. The easiest way to verify is to draw the shadow map to the screen using ImGui:

```
ImGui::Image(shadowSRV.Get(), ImVec2(512, 512));
```

As it only has a single color channel, it will appear red when drawn, as shown below.



Using the Shadow Map

Now that the shadow map has data, it can be used when rendering. This requires passing some extra data to both our standard vertex and pixel shaders, as well as passing some extra data between the two (the VertexToPixel struct). Remember that this will produce shadows from *one single directional light source*, not from all the lights in the scene! The code presented below will assume it's the first directional light, but this approach could be extended to tweak that or support multiple shadow maps.

> *While not explicitly shown below,* ***ensure the shadow map SRV is set*** *when drawing your entities to the screen. This can be done directly inside your standard render loop.*

Updates to Standard Vertex Shader

In addition to calculating the current vertex's screen position, the vertex shader now also needs to calculate the vertex's *shadow map position*. This is done by re-running the same calculation that was used when this object was initially rendered to the shadow map: $world * lightView * lightProjection$. Like screen position, this will need to be passed through the pipeline in the VertexToPixel struct. Details follow:

- Add two matrices – *lightView* and *lightProjection* – to the cbuffer in your vertex shader
- Update your VertexToPixel struct to include the shadow map position of this vertex:

```
float4 shadowMapPos : SHADOW_POSITION;
```

- Perform the WVP calculation for the shadow map:

```
matrix shadowWVP = mul(lightProjection, mul(lightView, world));
output.shadowMapPos = mul(shadowWVP, float4(input.localPosition, 1.0f));
```

Updates to Standard Pixel Shader

The pixel shader's VertexToPixel struct needs the same update as the vertex shader so we can grab the interpolated *shadow map position* data. It will also need an extra texture: the shadow map itself. Lastly, it must compare the pixel's actual distance from the light to the corresponding closest surface data from the shadow map.

Since the shadow map position is the result of a projection, we must the XYZ components by W. The rasterizer does this for *screen position* automatically, but we have to do it ourselves here. Note that this is only necessary for perspective projections, as W should be 1.0 for orthographic projections, but it's included here for completeness.

The XY components are a position within the shadow map, but they're in [-1, 1] normalized device coordinates instead of [0, 1] UV coordinates. This is easy enough to fix with a scale and offset, though we'll need to flip the Y component as the two coordinate systems are upside down from each other. How do we know the pixel's distance from the light? That's the Z component of the shadow map position vector!

Details follow:

- Update the VertexToPixel struct to include the same data as above:

```
float4 shadowMapPos : SHADOW_POSITION;
```

- Add the shadow map as an additional texture:

```
Texture2D ShadowMap : register(t4); // Adjust index as necessary
```

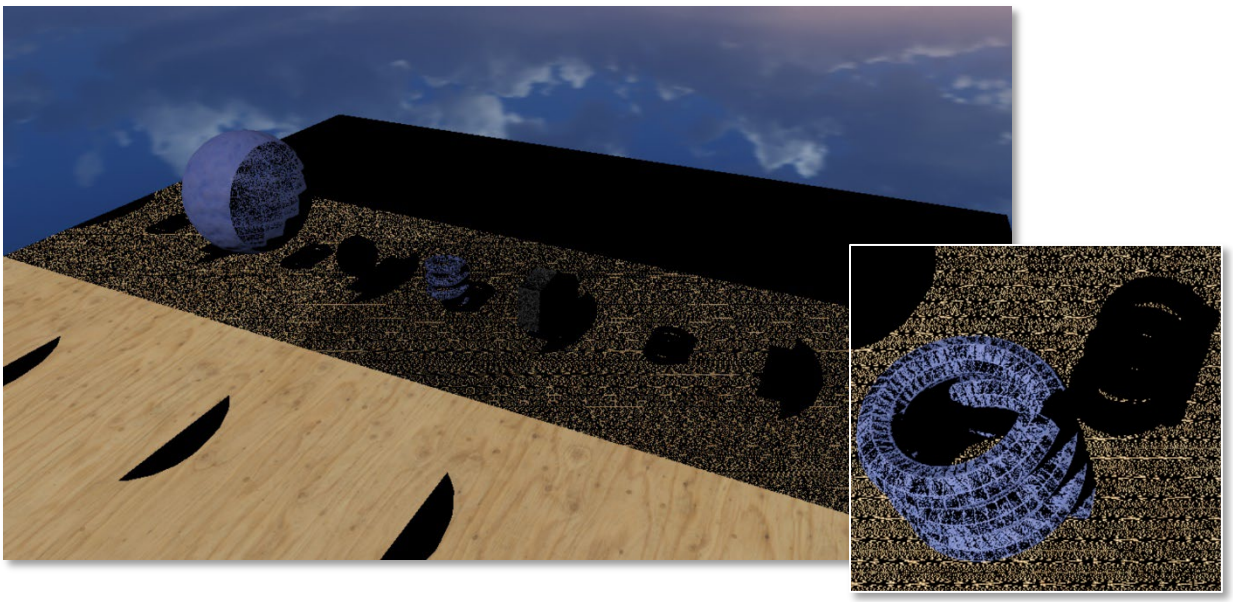
- At the top of your pixel shader, before any lighting, check the shadow map.

```
// Perform the perspective divide (divide by W) ourselves
input.shadowMapPos /= input.shadowMapPos.w;

// Convert the normalized device coordinates to UVs for sampling
float2 shadowUV = input.shadowMapPos.xy * 0.5f + 0.5f;
shadowUV.y = 1 - shadowUV.y; // Flip the Y

// Grab the distances we need: light-to-pixel and closest-surface
float distToLight   = input.shadowMapPos.z;
float distShadowMap = ShadowMap.Sample(BasicSampler, shadowUV).r;

// For testing, just return black where there are shadows.
if (distShadowMap < distToLight)
	return float4(0, 0, 0, 1);
```

Note that this code takes a shortcut just for testing: it returns pure black where there are shadows. This isn't what we ultimately want, but it will make shadows very obvious. We'll update this in the "Fixing Problems" section below.

Results

We have shadows! Sort of! And a few problems. There are hints of the correct shadows, though, so we're certainly on the right track. We'll discuss fixes for these issues next.

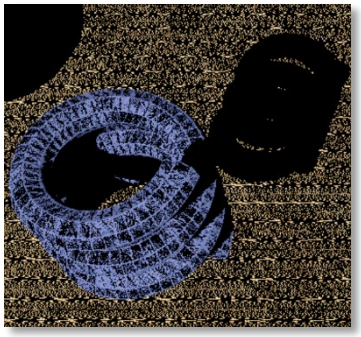


Fixing Problems

There are two major issues we'll need to fix: the "speckled" look, and the fact that the shadows repeat. Additionally, our fix for the repeating shadows will afford us the chance to soften the edges of the shadows slightly, too.

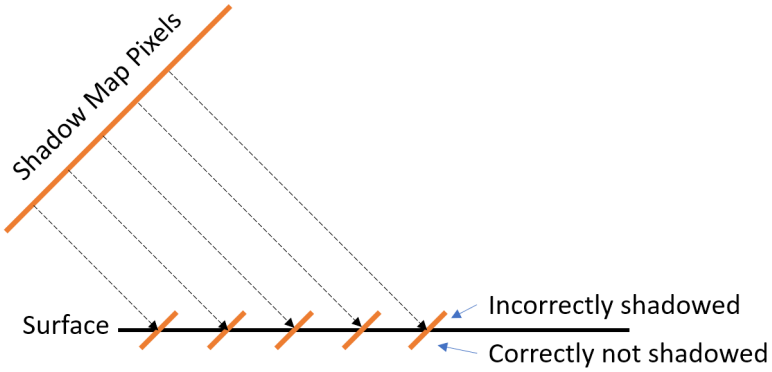
Shadow Acne

The speckling we see on all of the surfaces is known as *shadow acne*. It is due to inconsistencies in our distance comparisons (and probably some floating-point rounding errors, too).



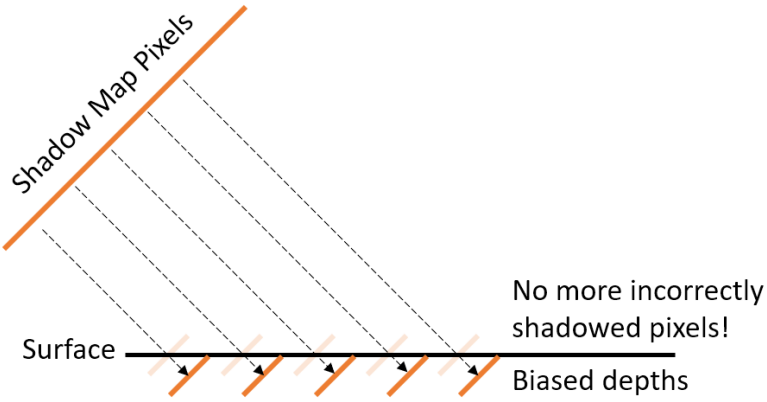
Since our shadow map has a finite resolution, each pixel in the shadow map represents some area or "chunk" of space in the world. However, we can only store a single depth value for that whole area. When we apply shadows, several screen pixels might be sampling the same shadow map pixel, especially as we get closer to a surface.

About half of the screen pixels will get a shadow map value that is closer to the light than the actual surface, resulting in incorrect shadows. The other half will get values that are farther away from the surface, which will be lit correctly. See the diagram to the right for a visualization of this issue.



Depth Biasing

The fix for this issue is to *bias* the depths in the depth buffer. This means adding a small amount to the values stored in the shadow map to avoid incorrect shadowing. A visualization of slight depth biasing is shown to the right. The value used for this should be quite small; just enough to make up for incorrectly shadowed pixels.



This is such a common issue that the rasterizer in most Graphics APIs can bias depth values for us. In Direct3D, this is handled through a `ID3D11RasterizerState` object. We've previously used these to change the cull mode. Here, we'll create a rasterizer state with depth biasing that we'll enable only while rendering to the shadow map.

Here are the steps for fixing shadow acne with depth biasing:

- Declare a rasterizer state object:

```
Microsoft::WRL::ComPtr<ID3D11RasterizerState> shadowRasterizer;
```

- During initialization, create a rasterizer state for depth biasing:

```
D3D11_RASTERIZER_DESC shadowRastDesc = {};
shadowRastDesc.FillMode = D3D11_FILL_SOLID;
shadowRastDesc.CullMode = D3D11_CULL_BACK;
shadowRastDesc.DepthClipEnable = true;
shadowRastDesc.DepthBias = 1000; // Min. precision units, not world units!
shadowRastDesc.SlopeScaledDepthBias = 1.0f; // Bias more based on slope
device->CreateRasterizerState(&shadowRastDesc, &shadowRasterizer);
```

- Before rendering to the shadow map, enable this rasterizer state:

```
context->RSSetState(shadowRasterizer.Get());
```

- After rendering to the shadow map, disable it:

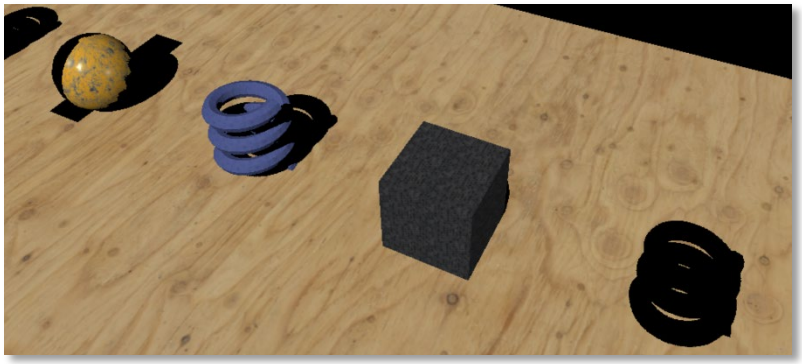
```
context->RSSetState(0);
```

Excessive Depth Bias: "Peter Panning"

Be careful when dialing in a depth bias. Biasing is really just pretending that the surface is farther away than it really is. Too much bias will cause your shadows to "disconnect" from the geometry, a visual error commonly known as "Peter Panning" (named after Peter Pan, whose shadow runs away).

Shadow Acne Be Gone

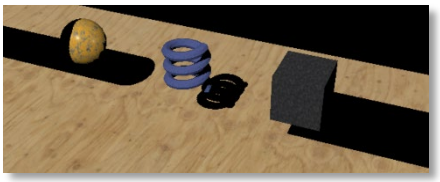
With proper depth biasing, the shadow acne disappears! One problem down, one to go.



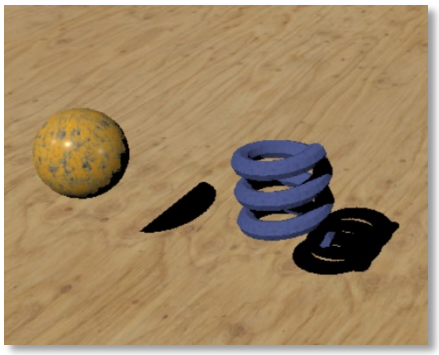
Shadow Repetition & Coverage

As seen in several previous screenshots, the shadows appear to repeat across the scene. This is due, in part, to the fact that our shadow map only encompasses a portion of the world. This means there are surfaces that are outside the bounds of the shadow map itself. When we calculate the shadow UV coordinate for these surfaces, we get values outside the [0, 1] UV range. Our default sampler, which we're currently using when sampling the shadow map, is presumably set to wrap UV coordinates outside this range. This is the primary cause of the shadow repetition.

Increasing the *coverage* of the shadow map will help, but we still need a better result when rendering surfaces that fall outside the bounds of the shadow map itself. Clamping UVs instead of wrapping will clamp the shadows on the edges of the shadow map, as seen to the right. Still not ideal.



There is a third address mode for samplers called "border". A border address mode will return a singular solid color for any sampling outside the [0, 1] UV range. As part of the sampler setup, we can define the "border color" that is returned. If we use a color with 1.0 in the red channel, which is the maximum possible depth value, then sampling outside the shadow map will always return max depth. This results in no shadows at all outside of the shadow map's coverage area, as seen to the right. While not physically *correct*, it's the least offensive option.



This requires us to make another sampler specifically for sampling the shadow map. The exact details of this can be found in a following section. Since we'll be creating another sampler for this anyway, below we'll look at another interesting facet of samplers that will result is slightly softer shadow edges.

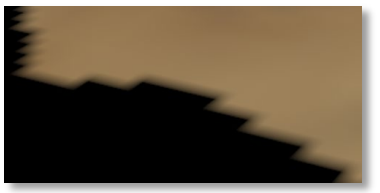
Hard Shadows

Our shadows currently have very hard edges: a pixel is either in shadow or it isn't. Increasing the resolution of the shadow map can help refine the edge, but it's still going to be either on or off.

Below are example shadow maps at 128x128, 1024x1024 and 4096x4096. Larger shadow maps take longer to render to, so just increasing the size isn't always the best option.

(Slightly) Softer Shadows

The opposite of a hard shadow is a *soft shadow*, where pixels can be partially shadowed. While fully soft shadowing is an advanced topic, we can have the GPU apply a slight gradient to the edges of shadows for us. This requires a slightly different sampler setup and slightly different syntax when sampling the shadow map.



Currently, we're grabbing a single depth value from the shadow map and doing a comparison ourselves which leads to a binary result. Technically, the depth value from the shadow map was an interpolation of 4 nearby texel values (see the blue square in the image to the right). If we could instead compare each of those 4 texels to our "distance from the light" independently, we could calculate a ratio of how many passed vs. how many failed the comparison. Thus, a point on a surface that's very close to the edge of a shadow could be partially shadowed. This technique is called *percentage-closer filtering* (PCF).

0	0	0	1
0	0	1	1
1	1	1	1
1	1	1	1

As an advanced technique, [PCF can be expanded further] to create even softer shadows.

Comparison Sampling for PCF

Rather than sampling the shadow map and doing a depth comparison ourselves, we can pass a value to the GPU and have it perform both the sampling and comparison for us. In doing so, it will perform the comparison on all 4 texels that are normally used for interpolation, returning a ratio of the comparison results. This is called *comparison sampling*, and can be enabled with a specialized sampler object. Details on implementing comparison sampling follow:

- Declare a sampler state object:

```
Microsoft::WRL::ComPtr<ID3D11SamplerState> shadowSampler;
```

- Set up the sampler for comparison during initialization:

```
D3D11_SAMPLER_DESC shadowSampDesc = {};
shadowSampDesc.Filter = D3D11_FILTER_COMPARISON_MIN_MAG_MIP_LINEAR;
shadowSampDesc.ComparisonFunc = D3D11_COMPARISON_LESS;
shadowSampDesc.AddressU = D3D11_TEXTURE_ADDRESS_BORDER;
shadowSampDesc.AddressV = D3D11_TEXTURE_ADDRESS_BORDER;
shadowSampDesc.AddressW = D3D11_TEXTURE_ADDRESS_BORDER;
shadowSampDesc.BorderColor[0] = 1.0f; // Only need the first component
device->CreateSamplerState(&shadowSampDesc, &shadowSampler);
```

- Add a comparison sampler to your pixel shader:

```
SamplerComparisonState ShadowSampler : register(s1);
```

- Replace the previous shadow map sample, comparison and return statements:

```
// Get a ratio of comparison results using SampleCmpLevelZero()
float shadowAmount = ShadowMap.SampleCmpLevelZero(
	ShadowSampler,
	shadowUV,
	distToLight).r;
```

- Use this value during lighting (on the first directional light):

```
float3 lightResult = DirLightPBR(...); // Calculate directional light

// If this is the first light, apply the shadowing result
if (lightIndex == 0)
{
	lightResult *= shadowAmount;
}

// Add this light's result to the total light for this pixel
totalLight += lightResult;
```

*Don't forget to **set the new comparison sampler** during your rendering loop, similar to setting the shadow map! Otherwise the sampling will use default options and won't work correctly.*

One Last Problem

You may notice the following D3D warnings in Visual Studio's output window.

```
D3D11 WARNING: ID3D11DeviceContext::OMSetRenderTargets: Resource being set to OM DepthStencil is
  still bound on input! [ STATE_SETTING WARNING #9: DEVICE_OMSETRENDERTARGETS_HAZARD]
D3D11 WARNING: ID3D11DeviceContext::OMSetRenderTargets[AndUnorderedAccessViews]: Forcing PS shader
  resource slot 4 to NULL. [ STATE_SETTING WARNING #7: DEVICE_PSSETSHADERRESOURCES_HAZARD]
```

These are related to the shadow map itself. Recall that it cannot be both a depth buffer and a shader resource at the same time. At the very end of a frame, the last thing we did with the shadow map was use it as a shader resource. At the beginning of the next frame, we use it as a depth buffer. However, at that point it's still bound as an SRV, too, causing these warnings.

The fix is to unbind the shadow map (or simply all of the SRVs) at the very end of the frame:

```
ID3D11ShaderResourceView* nullSRVs[128] = {};
context->PSSetShaderResources(0, 128, nullSRVs);
```

Results

We've got real-time shadows for one directional light! Want more? Do it again (and again, and again…)