Skyboxes & Cube Maps

What's a Skybox?

A skybox is the infinitely-far-away backdrop to a 3D scene. Skyboxes represent the atmosphere, clouds and other distant elements that exist beyond the player's reach, "behind" the actual game world.

In practice, a traditional skybox is simply a set of images used to fill in the area of the screen not covered by the geometry of the game world itself. These images can contain more than just *the sky*; for instance, distant mountain ranges or urban sprawl outside the playable area. However, since the bottom half of the screen is generally filled with level geometry, the skybox fills in the remainder (upper half). Hence, *sky*-box.

The screenshot above, from *Rage*, shows a very detailed, but static sky; the clouds never actually move. The screenshot to the right, from *Skyrim*, shows a static nighttime skybox – the stars and other celestial phenomena – mixed with moving geometric elements such as the aurora borealis. Mixing geometry and a skybox texture, and/or layering several slowly-rotating skybox textures at once, can further enhance the effect.

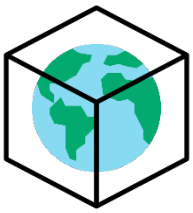
What About Clouds?

More modern games, such as Horizon: Zero Dawn, go as far as rendering *volumetric clouds*, which move and change over time, in front of the skybox. This gives actual depth to the environment above the player.

We won't be covering cloud rendering here, but feel free to check out the presentation slides or PDF from "The Real-time Volumetric Cloudscapes of Horizon: Zero Dawn" at SIGGRAPH 2015. Note that the slide deck contains videos and is well over 900MB!

Why a "Box"?

Isn't the atmo*sphere*, you know, spherical? That's true on Earth, but in a game engine we just need a simple enclosed shape on which to place our sky texture(s).



> *Remember that, in order for a pixel shader to run, some sort of geometry has to be rasterized. This means we must have a mesh to render when we "draw" our skybox to the screen, and the simpler the better!*

As we'll see, most skybox textures are really comprised of 6 equally-sized, square textures. Those six square textures are then arranged in the shape of a cube. What's the most cube-like mesh? A cube! Also known as a *box*. A few common questions and comments at this point:

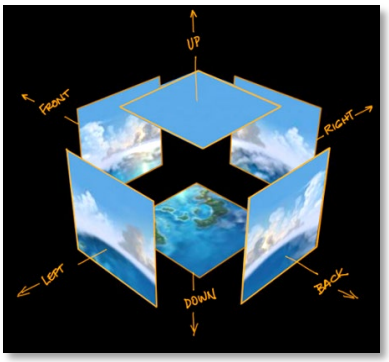
But it doesn't look like a box!
That's true (and kind of the point)! The textures we'll be using are designed to fit together seamlessly, hiding the true shape of the geometry we use to render the sky. As you may as guessed, we won't be applying any shading to the box – we're simply plastering it with the sky texture(s).

Shouldn't we use a sphere or dome? That feels more correct.
As it turns out, it won't matter! We won't be using the texture coordinates of the mesh for this process. We simply need geometry that fully encases us so we know which pixels to color. We'll be using the *direction* to those pixels *from the camera*, which is the same regardless of the geometry we choose.

But wait, *Game X* or *Engine Y* does it differently!
There are multiple ways of achieving this effect, and various ways of handling the necessary assets. The implementation we're using is quite standard and easy to incorporate into our engines, but countless tweaks and additional features are possible once you have the basics up and running.

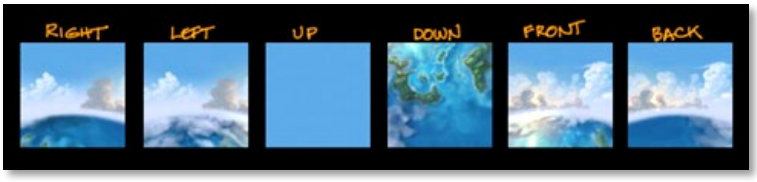
Skybox Assets

What kind of image(s) do we need for a skybox? Since the sky needs to fully surround our 3D environment, a single, standard 2D texture probably won't cut it. There are a few options for handling skybox textures, but one of the most common (and easy to find) setups uses 6 equally-sized, square textures. Each of these textures then corresponds to one of the major axes: +X, -X, +Y, -Y, +Z, -Z



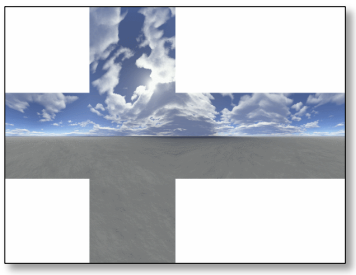
As seen to the right, arranging these textures into a cube shape would result in a completely seamless image. From the inside of the cube, it would appear as if this was one all-encompassing image.

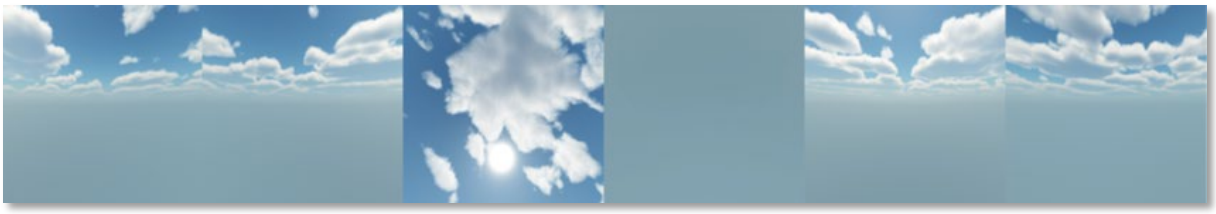
Note that, for this effect to work properly, we cannot just use any six pictures; each image has to seamlessly match its neighbors *and* represent a 90° field of view, whether it is captured from a camera, hand-painted or procedurally generated.



Skybox Storage Options

While we can store all six faces of our skybox as separate image files, some engines support storing all six in a single image and dynamically "chopping them up" at load time. For instance, you may have seen skybox images that look like one of the following examples. In both cases, the overall image contains the 6 equal-sized, square textures necessary, just arranged differently.





We'll be using the 6-separate-image approach, but any of these are feasible with the proper code.

We don't, however, just send all 6 individual textures to the pixel shader, as we wouldn't know which pixels on the screen correspond to which textures. The cube's UV coordinates won't really help either, as they're generally all between 0-1 and are meant to address into a single texture.

Instead, we'll need a new type of GPU texture resource called a *cube map*.

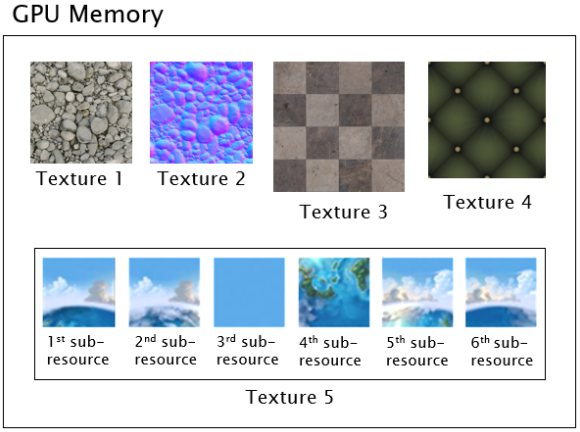
Cube Maps

Representing the world around us with a set of images is a common task in graphics programming. So common, in fact, that modern GPUs have a special resource type for this: a *cube map*.

A *cube map* is a single texture resource on the GPU containing 6 equally-sized, square *subresources*. Recall that a *subresource* of a GPU resource is simply a portion of its overall data. Textures can have two types of subresources: array elements and/or mipmaps. In the case of a cube map, these subresources are array elements. Thus, a cube map is technically just 6-element texture array, but with an internal flag specifying that the GPU treat it as a cube when we sample it.

The image to the right is a simple visualization of textures within GPU memory. The first four are standard 2D textures.

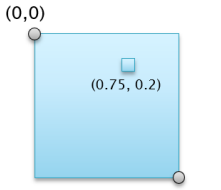
The fifth texture is a cube map with six subresources. In C++, we'd only need a single shader resource view (SRV) for this texture, as it really is a single resource as far as the GPU is concerned. The initial creation of this texture, however, is a bit more complicated. Sample code for loading six distinct images and building a cube map out of them will be provided.



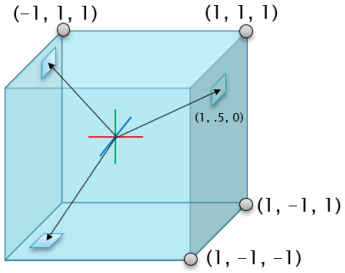
Sampling Cube Maps

How do we use this new texture type? And how is it different from a 2D texture?

Recall that when sampling a standard texture, we use two-dimensional UV coordinates to denote a point within the 2D grid of pixels (seen to the right). Images have inherent coordinate values [0, 1], so we're really just saying "go this far over from the left and this far down from the top".



Texture 2D
(Uses 2D coordinates)



Texture Cube
(Uses a direction)

In contrast, when sampling a cube map, we instead use a *direction in 3D space* to describe where to sample. This direction is relative to the center of the box, as if we are in the very center looking around.

For example, if we want the color of the pixel *directly ahead on the Z axis*, we'd use (0,0,1) for the sampling direction. If we want the color of the pixel to the *right and up slightly*, we might use (1, 0.5, 0) for the direction. *Left and back slightly* would be (-1, 0, -0.5). However, (0,0,0) is not a valid sampling direction, as it has no magnitude (it doesn't point anywhere).

Cube Maps in HLSL

When defining a cube map texture in HLSL, we use the TextureCube data type instead of Texture2D, as seen below. Binding the texture in C++ remains exactly the same, as long as the texture you're binding to this slot truly is a cube map.

```
Texture2D   SurfaceTexture  : register(t0);
TextureCube SkyboxTexture   : register(t1);
```

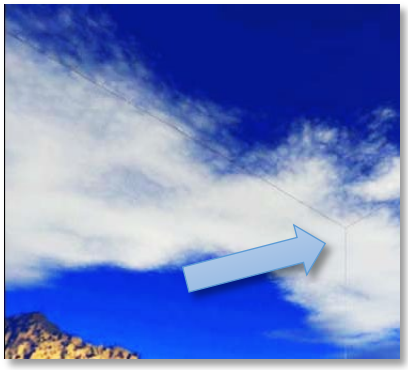
When sampling the cube map, we use a 3D direction vector (a float3, normalization not required). The GPU handles the math of translating the given direction to one of the six internal textures. The end result is very simple HLSL code for cube map sampling:

```
float3 direction = // Some useful direction for this pixel
float4 skyColor = SkyboxTexture.Sample(BasicSampler, direction);
```

Creating a Cube Map

While we can use the DirectX Toolkit to easily load standard textures, creating a cube map requires a little bit of manual work on our part. The details of this process are outlined at the end of this reading. It will require the six individual textures that make up the cube, all of which must be the same size and exactly square.

Quick Aside: Can't We Just Texture a Cube?



At this point, you may be thinking "this is great, but couldn't we just slap a texture on a cube and call it a day?" We could, but one common issue with attempting to seamlessly texture the inside of a cube is that, due to *texture filtering*, you often *see the seams!*

In the image to the right, you can clearly see where three sides of the cube meet. One solution to this issue is to use a proper cube map, as the GPU will correctly filter edge pixels with their neighboring faces.

Implementing a SkyBox

Okay, we have a cube map! We just render a cube and…that's it? Well, no. While we will be "rendering a cube" as part of the process, there are a few small issues we'll need to solve:

- How do we render the inside of the cube?
- Where is the cube?
- How do we ensure the cube is always behind everything?
- Which shaders do we use?
- When should we render it?

Rendering the Inside

We already have a cube mesh, but if we fly inside of it, we won't see anything. This is because, by default, the rasterizer is *culling back faces*: it ignores triangles that aren't facing the camera. When you're inside a mesh, that's all of them! The fix is relatively straightforward: tell the rasterizer to cull *front faces* instead, but *only for this particular object*.

The rasterizer's cull mode is part of the overall rasterizer *render state*. If we want to sometimes cull front faces and sometimes cull back faces, we'll need to change that render state at some point, and then change it back when we're done. This is extremely simple once we have a *rasterizer state object* that represents our desired options, which is stored as an ID3D11RasterizerState object:

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

We'll create this state once, during the initialization step of our program, and turn it on and off as necessary. Creating a rasterizer state to cull front faces (meaning it draws the inside instead of the outside) is only a few lines of code:

```
D3D11_RASTERIZER_DESC rastDesc = {};
rastDesc.CullMode = D3D11_CULL_FRONT; // Draw the inside instead of the outside!
rastDesc.FillMode = D3D11_FILL_SOLID;
rastDesc.DepthClipEnable = true;
device->CreateRasterizerState(&rastDesc, skyRasterState.GetAddressOf());
```

Just before rendering the sky's mesh itself (a cube), we'll change the rasterizer state like so:

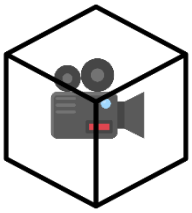
```
context->RSSetState(skyRasterState.Get());
```

And then change it back to the default settings right after rendering:

```
context->RSSetState(0); // Null (or 0) restores the default state
```

Where is the Cube?

While it may not be immediately obvious, one of the defining characteristics of a skybox is that you can never get closer to it. It is meant to represent the infinitely-far-away backdrop to a 3D environment, which means no matter how far you move, it's always infinitely far away.



So, where do we put the cube when we render it so it's always the same distance away from the camera? We *center it on the camera*. Thus, the camera can never get closer or further away from the cube because they're always at the same position.

However, rather than physically moving the vertices of the cube to the camera, we could be even more efficient and do the opposite: put both the cube and the camera at the origin. This may seem counterintuitive, but think about it like this: if you have a box on your head, it looks the same to you whether you're in your kitchen or your living room. The same idea applies here. Recall that a view matrix's entire goal is to make all of vertices relative to the camera, as if the camera were the origin. We're just going to skip that step and actually place it at the origin.

The implementation is quite simple. We can assume the cube is, by default, already at the origin and skip applying a world matrix. We can also zero-out the translation portion of our view matrix, thereby making the camera's position the origin, and *voilà*, the sky is centered on the camera! We'll look at the shader code for this soon.

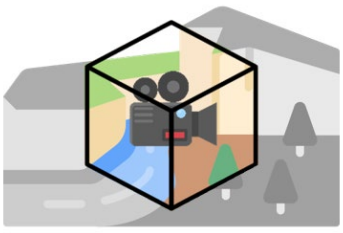
Note that the *scale* of the box is still an issue, as a box on our head blocks everything else!

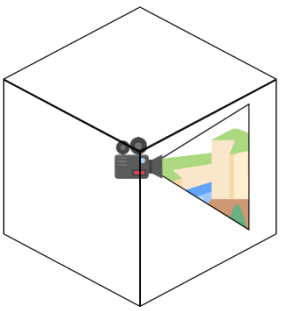
Ensuring the Cube is Behind Everything

This part is a little tricker, but easily solved once we understand the problem. We need the box to be centered on the camera, but it needs to be scaled up so that we can still see the rest of the game world. However, exactly how big should it be?

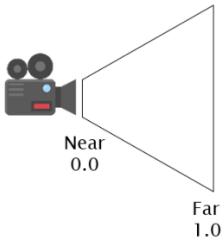
If it's too small, it obscures the scene beyond it, as shown in the left diagram.

If it's too big, it may extend past the camera's far clip plane and we can't see it at all, as shown in the right diagram.





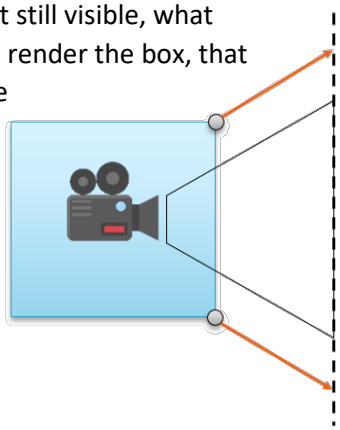
So, what's the ideal size? As it turns out, that's not the right question.



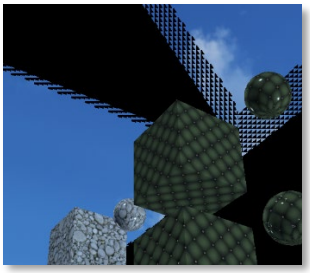
A quick review of depth values: For an object to be visible, it needs to be between our camera's near and far clip planes. This is determined by its post-projection depth value. In other words, the vertex's Z value *after* we apply view & projection matrices (and divide by the homogenous W coordinate).

If that depth is between 0.0 and 1.0, we can see the vertex. If the depth is less than 0.0, it's behind us. If the depth is greater than 1.0, it's too far away.

How does this help? If we want something to be as far away as possible, but still visible, what does its depth value need to be? *Exactly 1.0*. Can we ensure that, when we render the box, that the depth of every vertex is exactly 1.0? Sure! We'll literally just change the depth value before sending it out of the vertex shader. Remember that the XYZ portion will get divided by W by the rasterizer. To ensure the Z ends up as 1.0, we'll need to overwrite it with the W value. After the Z/W division, the depth will be exactly 1.0. This will effectively push the entire box up against the far clip plane, ensuring it is as far away as possible while still being visible.



Skyboxes and Depth



There's one more issue related to depth that we'll need to solve, otherwise the skybox will flicker wildly as we look around. See the image to the left for an example. What's going on here?

Each frame, we reset the depth buffer. This is done by setting every depth to the largest possible value. What's the value we use? 1.0. And what is the depth we're using for the sky? 1.0.

So, what's the problem? By default, the GPU only keeps a pixel if its depth is *less than* the current value in the depth buffer. If they're equal, the new pixel is *discarded*. The only reason we see any sky pixels in the image above is due to the imprecision of floating-point numbers!

This will also be an easy fix: we need to change another render state to tell the GPU to keep pixels if their depth values are *less than or equal to* existing depth values. This will look a lot like our rasterizer fix above. First define a depth/stencil state variable.

```
Microsoft::WRL::ComPtr<ID3D11DepthStencilState> skyDepthState;
```

Now create the state itself during initialization.

```
// Depth state so that we accept pixels with a depth <= 1
D3D11_DEPTH_STENCIL_DESC depthDesc = {};
depthDesc.DepthEnable    = true;
depthDesc.DepthFunc      = D3D11_COMPARISON_LESS_EQUAL;
depthDesc.DepthWriteMask = D3D11_DEPTH_WRITE_MASK_ALL;
device->CreateDepthStencilState(&depthDesc, skyDepthState.GetAddressOf());
```

Turn it on *before* rendering the sky: `context->OMSetDepthStencilState(skyDepthState.Get(), 0);`

Turn it off *afterwards*: `context->OMSetDepthStencilState(0, 0);`

Skybox Shaders

As you can probably tell, we'll need custom shaders for the skybox. We've got some skybox-specific tasks to take care of, and there's no lighting or other effects to apply. Let's dig in to the specifics.

Vertex Shader

The vertex shader will be in charge of centering the camera (removing its translation), adjusting the final Z value (replace it with W) and determining a sampling direction for this vertex (to be used when sampling the cube map in the pixel shader).

Since this is a very simple vertex shader, we'll be using a simplified VertexToPixel struct for its output:

```
struct VertexToPixel
{
	float4 screenPosition : SV_POSITION;
	float3 sampleDir      : DIRECTION;
};
```

The vertex shader itself will first center the camera by removing the translation from the view matrix and then multiplying just view and projection, skipping a world matrix entirely. We cannot edit variables from a constant buffer directly, so a copy needs to be made first.

```
// Modify the view matrix and remove the translation portion
matrix viewNoTranslation = view;
viewNoTranslation._14 = 0;
viewNoTranslation._24 = 0;
viewNoTranslation._34 = 0;

// Multiply the view (without translation) and the projection
matrix vp = mul(projection, viewNoTranslation);
output.screenPosition = mul(vp, float4(input.localPosition, 1.0f));
```

Next up, replace the output position's Z by its own W to ensure a depth of 1.0 after division:

```
output.screenPosition.z = output.screenPosition.w;
```

Lastly, we need to know the direction to this vertex from the camera so we can sample the cube map in the same direction. How do we calculate a direction from the camera to a point in space? Vector subtraction: vertexPosition - cameraPosition.

However, what is the camera's effective position? (0,0,0). Subtracting that would be useless! We can simply use the vertex's original position as its sampling direction, since position ultimately is just a direction from the origin.

```
output.sampleDir = input.localPosition;
```

Pixel Shader

The pixel shader is about as simple as we can make it. Ensure you have the updated VertexToPixel struct and the proper resources bound:

```
TextureCube SkyTexture    : register(t0);
SamplerState BasicSampler : register(s0);
```

The entire pixel shader is then just one line:

```
return SkyTexture.Sample(BasicSampler, input.sampleDir);
```

Render Order & Overdraw

When is the best time to render the sky? You might think we should render it first, before any other objects. It needs to be "behind" everything, right?

This has a distinct issue: we'd be spending the time processing and shading every single pixel of the screen, even though we'll end up covering half or more of them up with the geometry of the world! This is known as *overdraw*, and it should ideally be avoided whenever possible. Why waste time rendering something we'll never see?

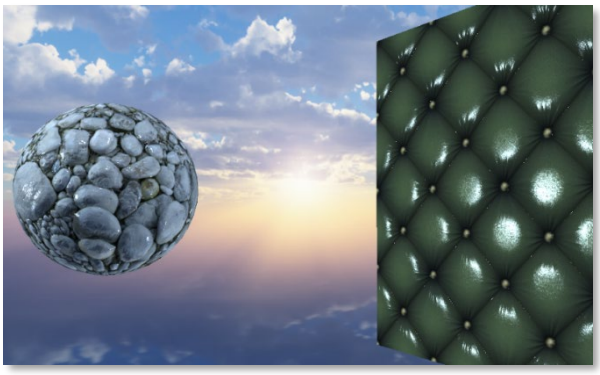
How do we ensure we only draw the portion of the sky that the user will see? We draw it *last*. After all other opaque (non-transparent) objects. Because of our depth-tweaking vertex shader, we're guaranteed that anything already drawn will be "in front" of the skybox.

That's it! Optimization complete.

> *Notice I mentioned opaque objects specifically? Transparent objects must be drawn <u>after</u> the skybox, but we don't have to worry about them just yet.*

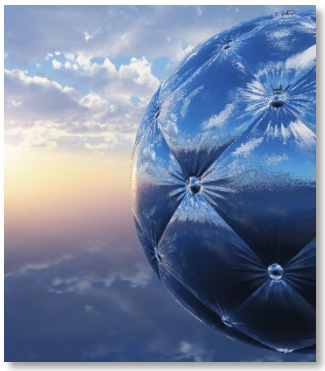
The Sky!

If all went well, you've got a sky fully surrounding your 3D scene. If you use an ambient color in your lighting calculations, make sure it matches your new skybox, since the sky should appear to provide light to the scene. If your skybox texture has a vert bright area meant to be the sun, you could even have one of your directional lights match that orientation to further sell the effect.



One interesting behavior to note: since the sky is centered on the camera, moving (without rotating) will not change what the sky looks like. If there are no other objects on the screen, you won't be able to tell you're actually moving! Camera rotation should work as expected, though, since we left that portion of the view matrix alone.

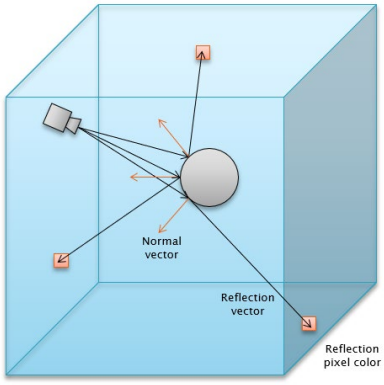
Environment Mapping: Cube Maps for Reflections



Not only can we use cube maps for the sky itself, but we can also use them to make objects in our scene look like they're reflecting the environment. This technique is often called *environment mapping*.

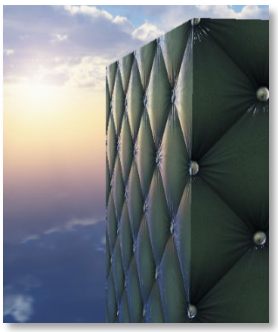
A cube map is like any other texture resource in that we can bind it and sample from it in any shader. This includes the shaders we use to render objects in our scene (in other words, our "standard" pixel shaders). We just need a direction that represents a reflection off the surface.

As it turns out, we already have everything we need to implement environment mapped reflections! From our implementation of the Phong specular equation, we know that HLSL has a reflect() function which requires an *incident* (incoming) vector and a *normal*.



We have the *normal*: that's from our geometry and/or normal mapping. The incident vector is really the vector from our camera to the pixel, which we've probably already calculated (in reverse) for the Phong equation. Putting it all together:

```
// Sample the skybox using the reflected view vector
float3 viewVector = normalize(cameraPos - input.worldPos);
float3 reflVector = reflect(-viewVector, input.normal); // Need camera to pixel vector, so negate
float3 reflectionColor = SkyboxTexture.Sample(BasicSampler, reflVector).rgb;
```



If we just return that reflection color from the pixel shader, our objects will look like perfect mirrors, as in the screenshot above. This might not be what we want, however, so we need a way of mixing the surface's final color (including lighting/shading) and the reflection color. One property of real-world objects is that they become more reflective as we see them at glancing angles. Head on, we get the object's color, but at an angle, we should see the reflection. This phenomenon is described by a Fresnel equation, which can be coded succinctly using Schlick's approximation:

```
// Schlick's approx. of Fresnel term: F(n,v,f0) = f0 + (1-f0)(1 - (n dot v))^5
//   n - Normal vector
//   v - View vector
//   f0 - Specular value (usually 0.04 for non-metal objects)
float SimpleFresnel(float3 n, float3 v, float f0)
{
	float NdotV = saturate(dot(n, v));
	return f0 + (1 - f0) * pow(1 - NdotV, 5);
}
```

Putting this into practice, we interpolate between the final shaded color and the reflection based on the Fresnel result, often using a specular value of 0.04. This matches most real-world non-metallic objects.

```
float3 result =
   lerp(totalLight, reflectionColor, SimpleFresnel(input.normal, viewVector, 0.04f));
```

Alternative to Cube Maps: Latitude-Longitude Maps

While we won't be making use of them, I'd be remiss if I didn't at least mention *latitude-longitude (equirectangular) maps.*

These are standard 2D textures that essentially contain a warped version of an entire environment map, as seen to the left, similar to a map of Earth. Notice the extreme distortion along the top and bottom edges.

You could replace a cube map with this type of texture by converting from Cartesian coordinates to spherical coordinates before sampling.

Code Breakdown: Creating a Cube Map from Six Distinct Textures

While we can use the DirectX Toolkit to easily load standard textures, creating a cube map requires a little bit of manual work on our part. The details of this process are outlined below, and can be easily placed inside a helper function that accepts 6 file paths and returns a shader resource view for the new cube map (as a ComPtr, of course).

We can still load the 6 images representing the faces of the cube map using the DirectX Toolkit, but we'll need to create the final cube map resource (and associated SRV) by ourselves. Then we can use a built-in Direct3D command to copy the pixel data from one resource (the first image we loaded) to another resource (the first face of the cube map). We'll perform that step 6 times, copying into a different subresource of the cube map each time.

Step 1: Load the 6 textures and grab the description of one

Assuming we have the file paths to the 6 textures of our eventual cube map in variables named *right*, *left*, *up*, etc., we load them into an array of textures (*not* SRVs, since we need the raw texture data). Then we grab the description of the first to get its pixel dimensions and color format.

```
// Load the 6 textures into an array.
// - We need references to the TEXTURES, not SHADER RESOURCE VIEWS!
// - Explicitly NOT generating mipmaps, as we don't need them for the sky!
// - Order matters here!  +X, -X, +Y, -Y, +Z, -Z
Microsoft::WRL::ComPtr<ID3D11Texture2D> textures[6] = {};
CreateWICTextureFromFile(device.Get(), right, (ID3D11Resource**)textures[0].GetAddressOf(), 0);
CreateWICTextureFromFile(device.Get(), left,  (ID3D11Resource**)textures[1].GetAddressOf(), 0);
CreateWICTextureFromFile(device.Get(), up,    (ID3D11Resource**)textures[2].GetAddressOf(), 0);
CreateWICTextureFromFile(device.Get(), down,  (ID3D11Resource**)textures[3].GetAddressOf(), 0);
CreateWICTextureFromFile(device.Get(), front, (ID3D11Resource**)textures[4].GetAddressOf(), 0);
CreateWICTextureFromFile(device.Get(), back,  (ID3D11Resource**)textures[5].GetAddressOf(), 0);

// Get the description of the first texture
D3D11_TEXTURE2D_DESC faceDesc = {};
textures[0]->GetDesc(&faceDesc);
```

Step 2: Create the blank cube map

Next up, we need to create a cube map resource in GPU memory, which is a texture 2D with an array size of 6 and a flag informing the GPU that this should be treated as a cube.

```
D3D11_TEXTURE2D_DESC cubeDesc = {};
cubeDesc.ArraySize = 6; // Cube maps must have exactly 6 sides
cubeDesc.BindFlags = D3D11_BIND_SHADER_RESOURCE; // We'll be using in a shader
cubeDesc.CPUAccessFlags = 0;
cubeDesc.Format    = faceDesc.Format; // Match the loaded texture's color format
cubeDesc.Width     = faceDesc.Width;  // Match the size
cubeDesc.Height    = faceDesc.Height; // Match the size
cubeDesc.MipLevels = 1; // No mipmap chain
cubeDesc.MiscFlags = D3D11_RESOURCE_MISC_TEXTURECUBE; // It's a CUBE
cubeDesc.Usage     = D3D11_USAGE_DEFAULT;
cubeDesc.SampleDesc.Count   = 1;
cubeDesc.SampleDesc.Quality = 0;

// Create the final texture resource to hold the cube map
Microsoft::WRL::ComPtr<ID3D11Texture2D> cubeMapTexture;
device->CreateTexture2D(&cubeDesc, 0, cubeMapTexture.GetAddressOf());
```

Step 3: Copy each texture into the cube map

We have exactly 6 textures, so we'll need to loop exactly 6 times. Each time, we calculate the index of the current subresource in the cube map. Then we use CopySubresourceRegion() to copy the pixel data from one texture into the corresponding subresource in the cube map.

```
// Loop through the individual face textures and copy them,
// one at a time, to the cube map texture
for (int i = 0; i < 6; i++)
{
	// Calculate the subresource position to copy into
	unsigned int subresource = D3D11CalcSubresource(
		0,  // Which mip level (zero, since there's only one)
		i,  // Which array element?
		2); // How many mip levels are in the texture?

	// Copy from one resource (texture) to another
	context->CopySubresourceRegion(
		cubeMapTexture.Get(), // Destination resource
		subresource,          // Destination subresource index (one array element)
		0, 0, 0,              // XYZ location of copy
		textures[i].Get(),    // Source resource
		0,                    // Source subresource index (there's only one)
		1);                   // Source subresource region (zero = the whole thing)
}
```

Step 4: Create the SRV for the cube map

Now that we've copied all of the data into the cube map, the last step is to create a shader resource view for it so we can bind it to the pipeline and use it in a shader. Note that after we do this, we no longer need the reference to the texture itself, as the SRV internally tracks it.

```
// At this point, all of the faces have been copied into the
// cube map texture, so we can describe a shader resource view for it
D3D11_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Format = cubeDesc.Format;  // Same format as the texture above
srvDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURECUBE; // Treat this as a cube!
srvDesc.TextureCube.MipLevels = 1;  // Only need access to 1 mip
srvDesc.TextureCube.MostDetailedMip = 0; // Index of the first mip we want to see

// Make the SRV
Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> cubeSRV;
device->CreateShaderResourceView(cubeMapTexture.Get(), &srvDesc, cubeSRV.GetAddressOf());
```

At this point, the variable *cubeSRV* is all we need! If this is in a helper function, that's what you'd return.