Skyboxes & Cube Maps

The Sky

What's a skybox?

- The infinitely-far-away backdrop to a 3D scene
- Always "behind" the game world

Mixing with geometry

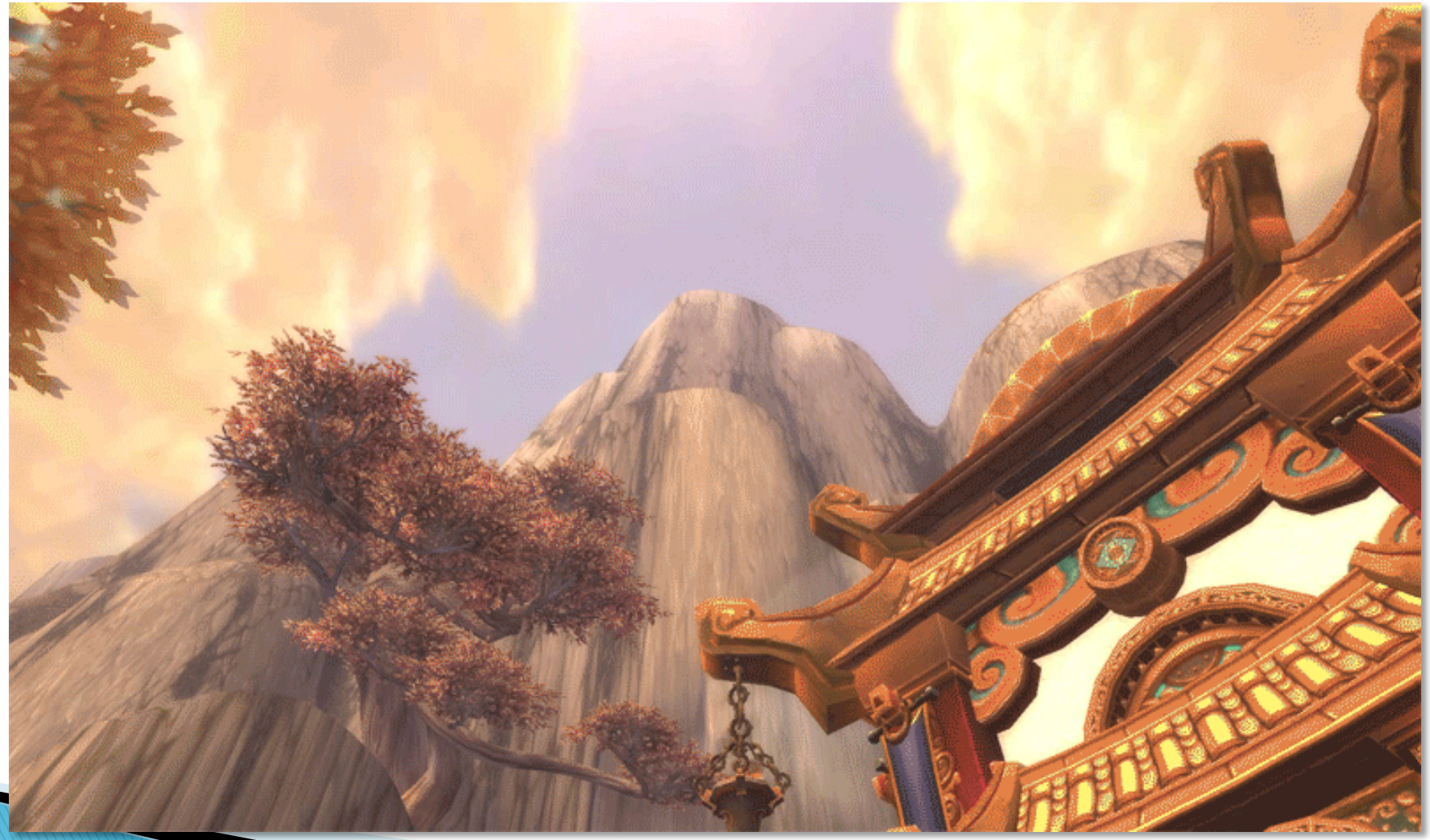
- Skybox is all the way in the back

- Not…
  - The dragon
  - Northern Lights

We never get closer to the sky

- Otherwise it would look like we're approaching a wall

Moving up doesn't make clouds larger



Sky appears to “move” with you

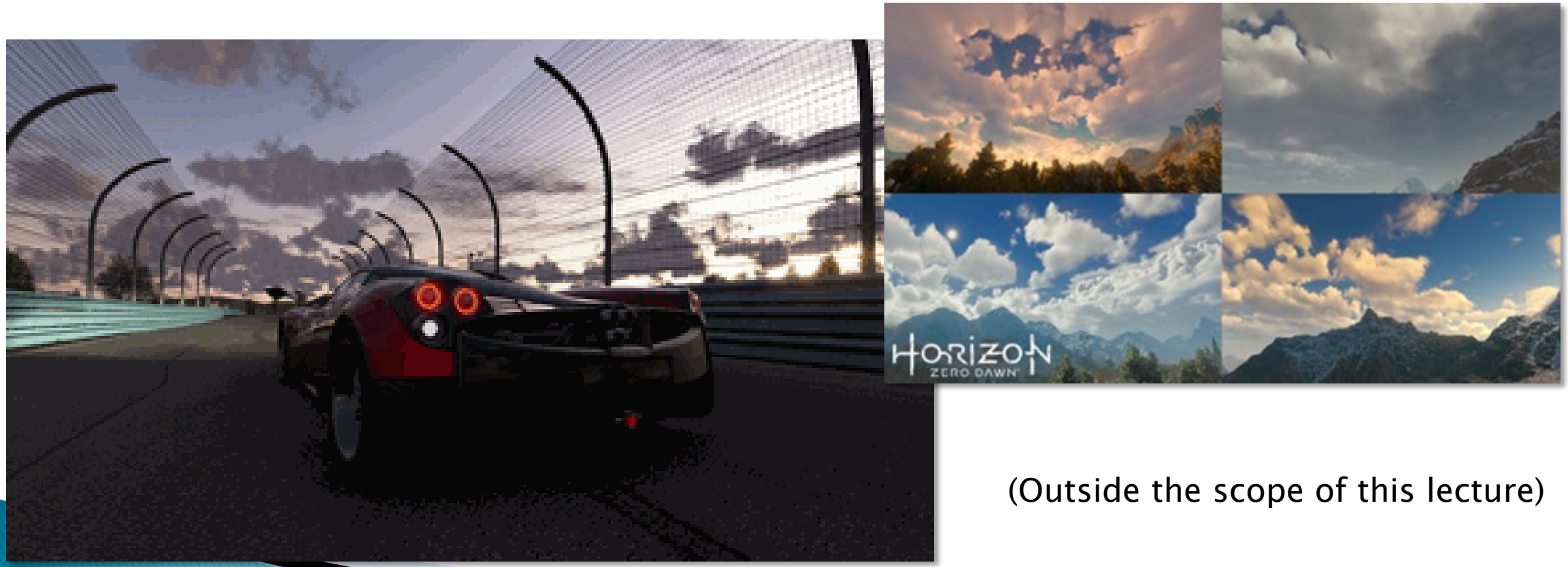
Sky can be animated and/or layered

Should lights affect the sky?

- Nope!
- Just use raw texture sample
- Sky should appear to *provide* light to a scene

What about clouds?

- Some modern games do render volumetric clouds
  - In front of the skybox



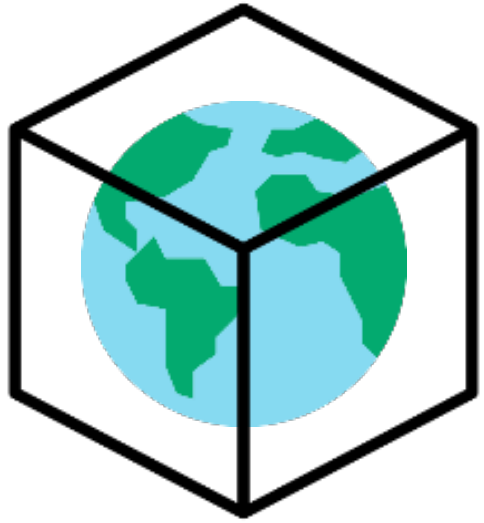
(Outside the scope of this lecture)

Skybox Requirements

Sky rendering overview

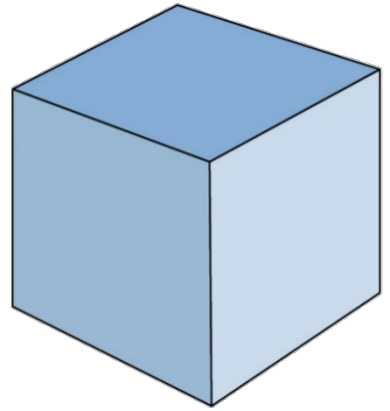
- To render the sky, we'll need:
  - A mesh
  - A texture
  - Specialized "sky" shaders
- The sky isn't a regular "game entity"
  - Lights don't affect it
  - Doesn't need a transform
  - There is an optimal time to render the sky

Sky mesh: An actual box!



- The sky should fill the screen
  - Need to rasterize geometry “all around us”
  - Need a shape that fully surrounds us
- A literal box!
  - Cubes are nice and simple (12 triangles)
  - Not actually using its UVs
  - Just need a direction to each pixel in 3D space

Sky texture?



- Need to texture our sky mesh
  - Seamlessly
  - Should look like a continuous image

- How do we do this with a single 2D texture?

- Spoilers
  - We generally *don't*
  - We need another type of GPU texture resource: ***a cube map***

Cube Maps

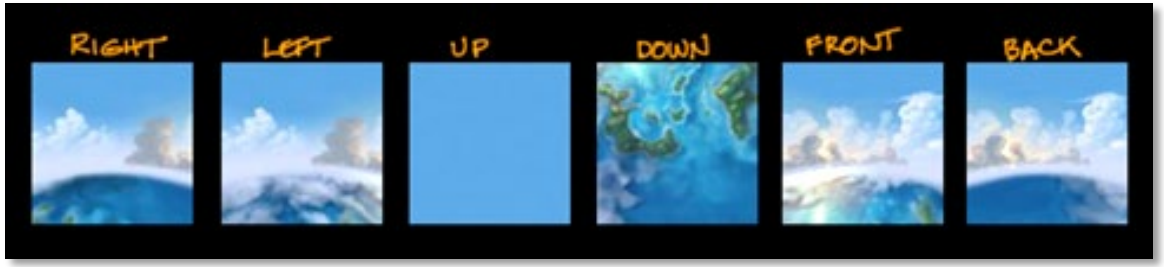
Six textures treated like they’re a cube

Cube maps

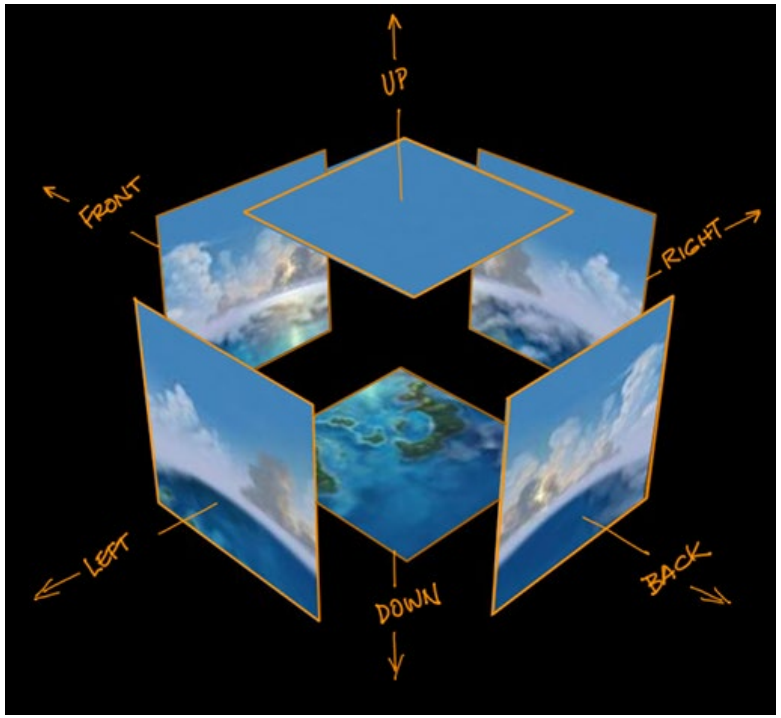
- A special type of GPU texture resource
- Internally, contains six Texture2Ds
  - Each of which represents one “face” of a cube
- Sometimes referred to as “Skyboxes”
  - This is not entirely accurate
  - Though it is a common colloquialism
  - Cube maps can be used for more than just the sky

Cube map example

- 6 textures



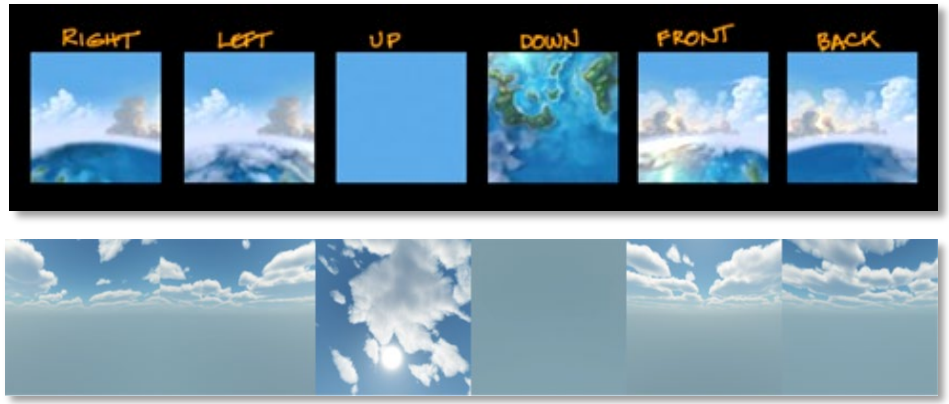
- Cube Map containing those textures



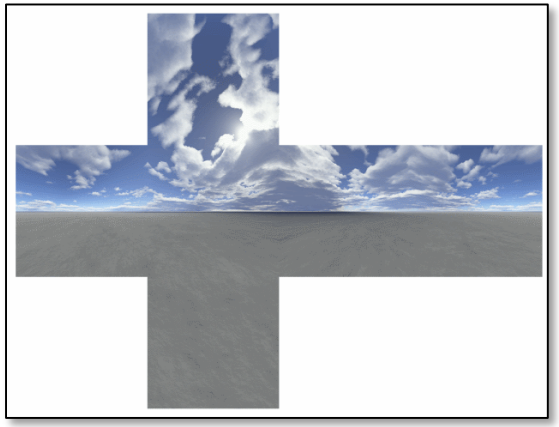
- Specific face requirements
  - Each texture edge must match neighbors
  - Each texture must represent a 90º fov
  - Otherwise it won't look seamless

Cube map storage (file) options

- Many ways to organize these textures
  - 6 separate image files
  - 1 file with a horizontal layout
  - 1 file with a “cross” layout



- All of these are feasible
  - With the right code
  - To “chop” them up



- We’ll use the “6 image” approach

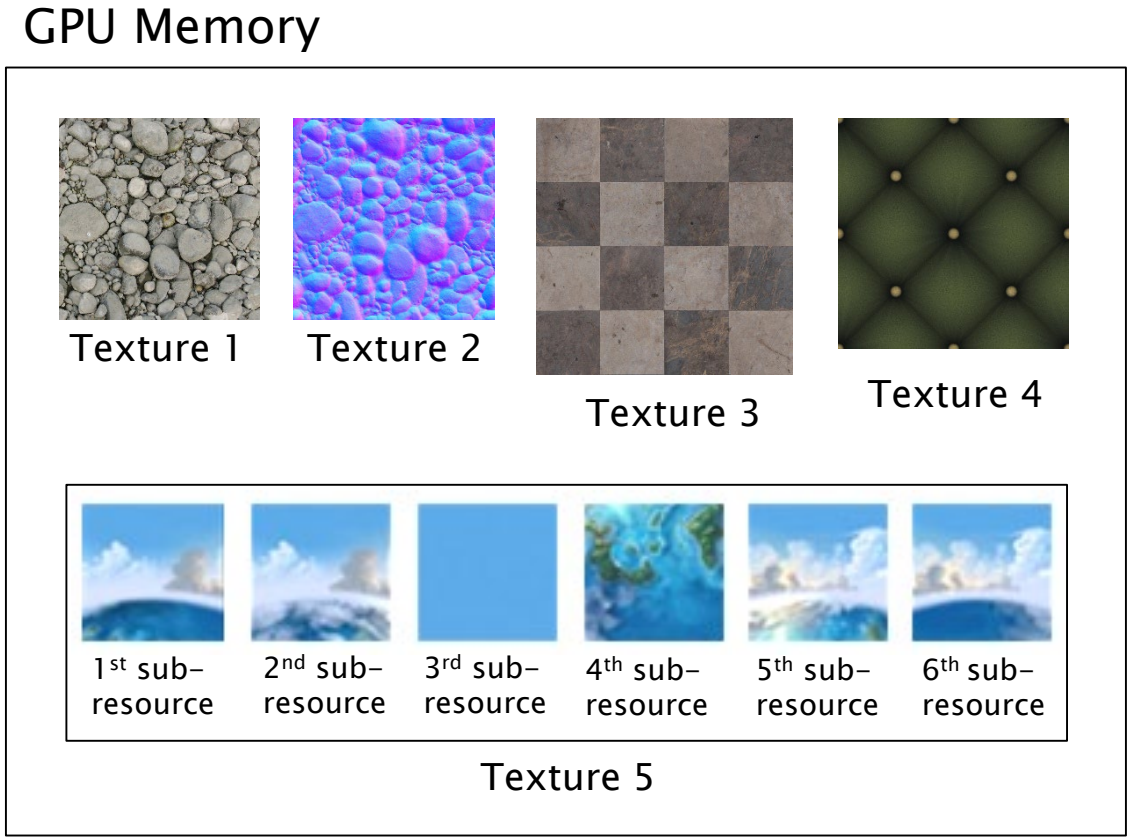
Cube maps on the GPU

- Reminder: Textures can have 1 or more sub-resources
  - Can treat sub-resources as “array elements”
  - And flag the resource as a cube map instead of an array
- Note: This is an explicit GPU resource type!
  - Texture2DArray
  - NOT a C++ array of Texture2Ds
- We’ll look at how to create this through code soon

GPU texture example

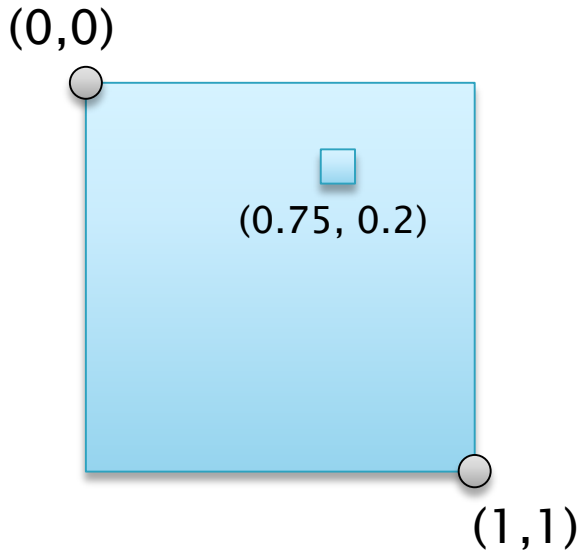
Several individual 2D textures →

Single cube map texture
with 6 sub-resources →

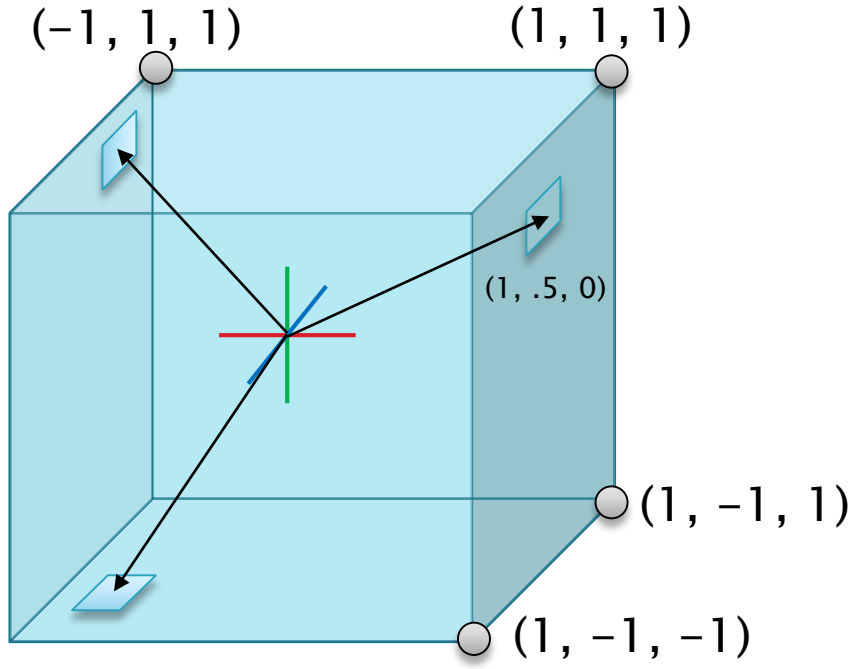


Sampling: Texture2Ds vs. Cube Maps

- CubeMaps are sampled using a ***direction*** in space



Texture 2D
(Uses 2D coordinates)



Texture Cube
(Uses a direction)

Cube maps in HLSL

- Specify **TextureCube** in shader

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

- Sampling a TextureCube requires a 3D *direction*

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

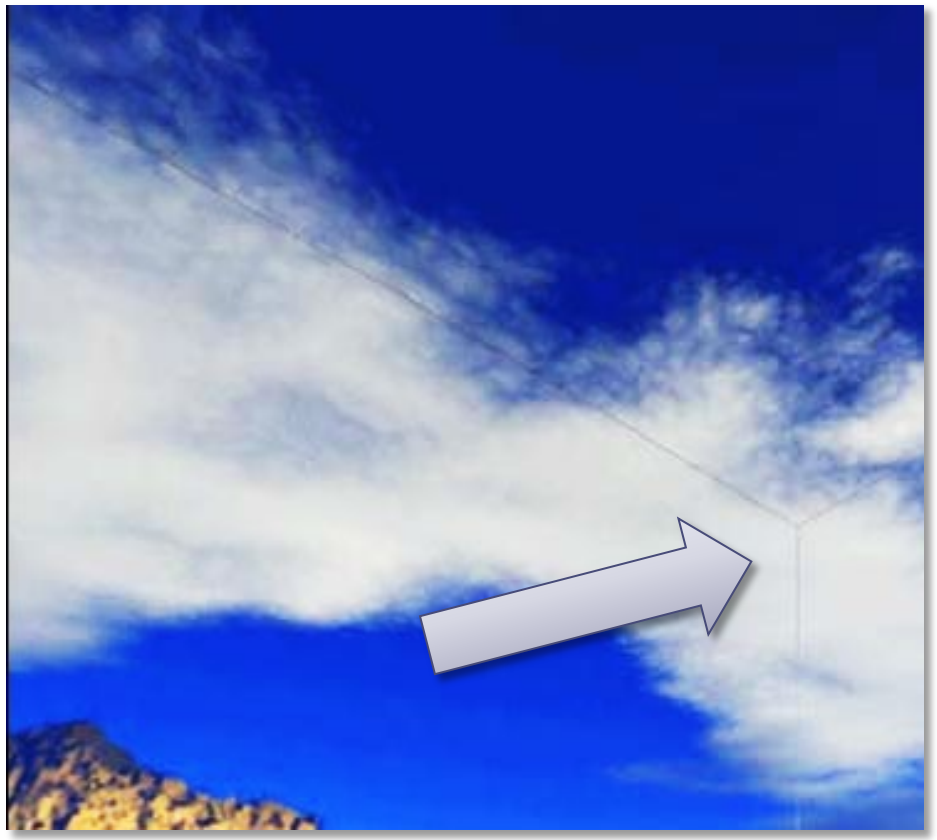
- Imagine being in the center of the cube
  - Given a direction in 3D space…
  - Sample the cube map's color in that direction

Uses for cube maps

- Skies
- Environment mapping
  - “Pre-baked” reflections
- Omnidirectional shadows
  - More on this when we get to shadow mapping
- And more!

Quick note: Why not just UV map a cube?

- Texture filtering may result in obvious lines at geometry borders!
- Cube maps don't have this issue
- GPUs properly filter cube map "neighbors"



Actually Rendering the Sky

A few issues to tackle

- Need to render the inside of the cube
- Where should the cube be in our 3D scene?
- How do we make the cube appear “behind” everything?
- When should we render the sky?
- How do we get the direction to each pixel?

Rendering the inside

- By default, the pipeline only rasterizes “front faces”
  - We see the outside of objects
  - But not the insides
  - Useful optimization
- We can change this behavior
- Need a D3D API object that contains rasterizer options

ID3D11RasterizerState

- Define the object

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

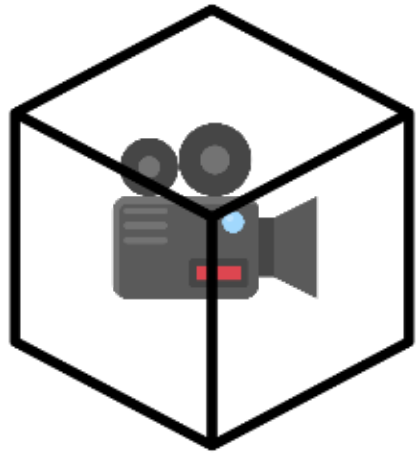
- Create during initialization

```
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());
```

- Set/unset during Draw()

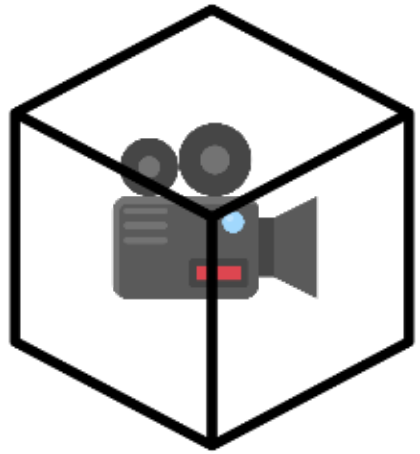
```
context->RSSetState(skyRasterState.Get());
// Draw the skybox here!
context->RSSetState(0); // Null (or 0) restores the default state
```

Where is the cube?



- Skybox should be centered on camera
  - Always the same distance
  - We never get closer to the skybox
- How do we ensure two objects are at the same position?
  - Move *both*?
- Even better: move *neither!*
  - Keep them both at the origin
  - Results will be the same

Centering the cube



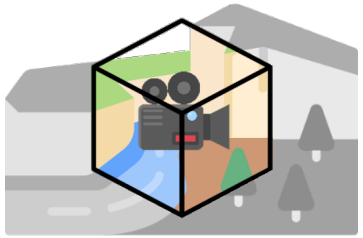
- View matrix reminder
  - Makes everything *relative to the camera*
  - Essentially a “reverse transformation”
- If we move an object to the camera, view matrix *undoes that*
- If you have a box on your head…
  - It looks the same no matter where you are
  - Same idea here

Vert shader: Center skybox on camera

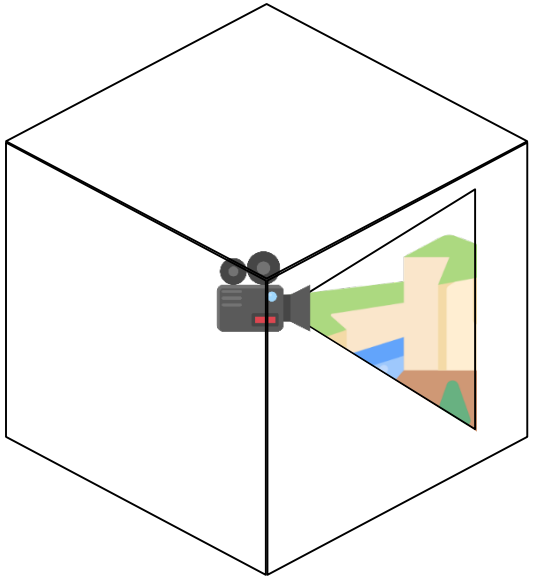
```
// Use this when rendering skybox geometry
// - Removes translation portion of camera's view matrix
// - And does NOT use a world matrix for skybox geometry
// - End result: Sky always appears centered on camera

// Create a version of the view matrix without translation
matrix viewNoTranslation = view;
viewNoTranslation._14 = 0;
viewNoTranslation._24 = 0;
viewNoTranslation._34 = 0;

// Calculate output position w/ just view & projection
matrix viewProj = mul(projection, viewNoTranslation);
output.position = mul(viewProj, float4(input.position, 1.0f));
```

Ensuring the cube is behind everything

- If the box is too small
  - It blocks the world
  - Like a wall around us

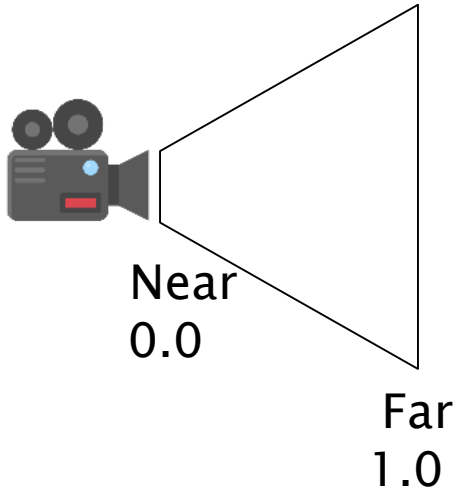


- If the box is too big
  - It's beyond our camera's far plane
  - Can't actually see it



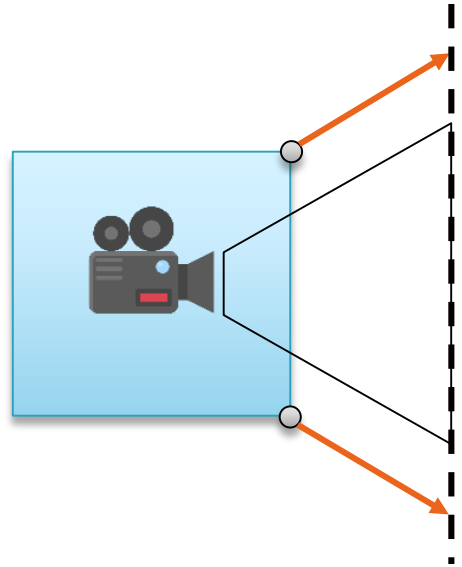
Reminder: Depth & clip planes

- Post-projection Z value is *depth*
  - Distance between near and far clip planes
  - Depth values between 0 and 1 are “visible”

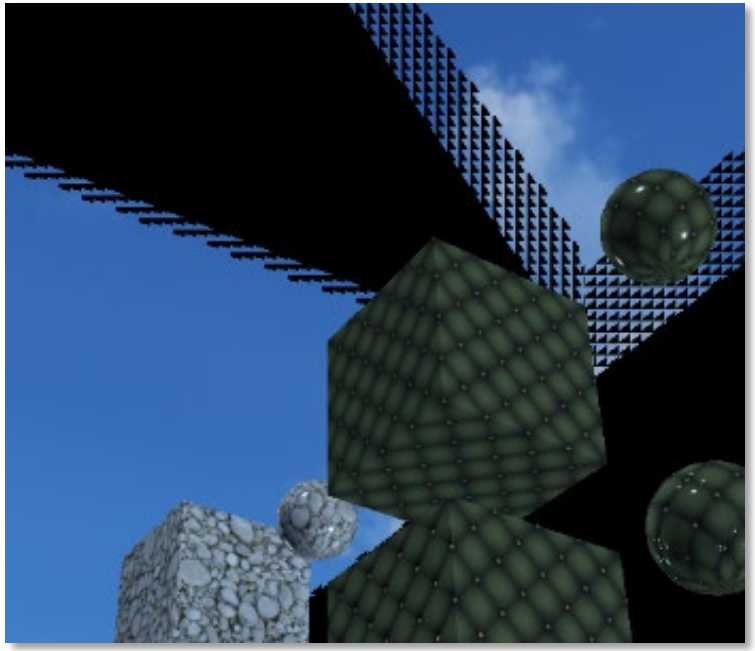


- Largest visible depth value? 1.0

- How do we ensure that?
  - We overwrite depth in the vertex shader
  - Essentially “pushing” the vertices onto far plane
  - Set Z = W, so Z/W division yields a depth of 1.0



A new depth issue



- We might run into the following issue
  - Parts of the sky are missing!
  - And they flicker wildly

- What's going on?

- When we clear the depth buffer, we set all values to 1.0
- When we draw the skybox, we set depths to 1.0
- By default, pipeline *ignores* pixels w/ depth == depth buffer
  - We still see a few due to floating point rounding errors

Change depth state to accept more pixels

- Define the object

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

- Create 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());
```

- Set/unset during Draw()

```
context->OMSetDepthStencilState(skyDepthState.Get(), 0);
// Draw skybox here!  (And also set/unset rasterizer state)
context->OMSetDepthStencilState(0, 0);
```

When do we render the sky?

- Before other objects?
  - Pro: Easy to implement & probably fast enough
  - Con: Could result in lots of *overdraw*
- Even better: After all other (opaque) objects!
  - Rasterizer will skip pixels based on depth buffer
  - All scene objects will have depth < 1
  - Only shade pixels we can see!
- Note: Transparent objects need to come after skybox

Direction to each pixel for cubemap sampling

- Sky mesh's verts ideally centered around origin
  - They're basically directions from the origin
  - Vertex position == sampling direction
- Pass that vector from vertex shader to pixel shader
  - `vsOutput.sampleDirection = input.localPosition;`
- Pixel shader is extremely simple
  - Sample cube map in given direction
  - Return that color

Environment Mapping

Using a cube map for reflections

Environment mapping

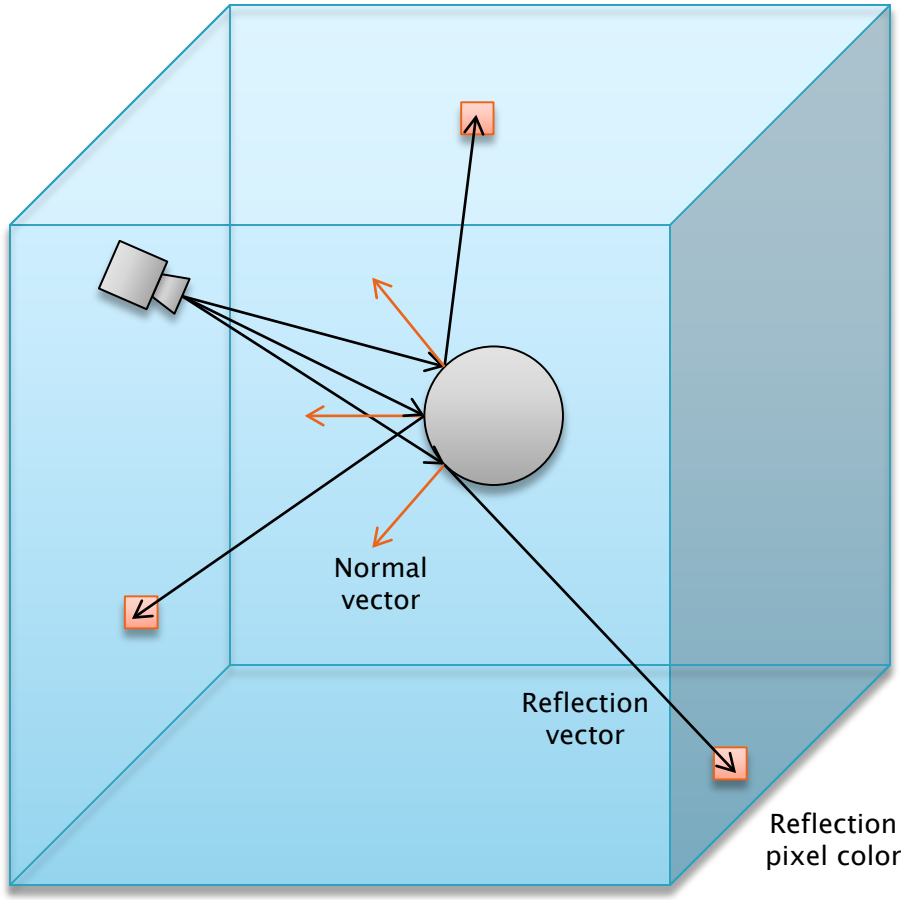
- If we have a cube map that represents “the world around us”
- We can use it on reflective surfaces

Reflections

- Use the same map for the sky and reflections
- Looks like objects reflect the environment!

Calculating a reflection

- Figure out the direction from the camera to the current pixel
- Calculate the reflection vector
- Sample cube map in that direction



Reflection sampling

- Remember: Cube Map sampling uses a direction

- Calculate direction from camera to vertex
  - Do this all in world-space!
  - Camera position in world space
  - Vert position in world space: vertPos * worldMatrix

- Use HLSL “reflect” function
  - Reflects a vector based on a normal
  - Use vertex’s normal
  - Use output of function as cube map sample vector

Environment mapping problems

- Environment maps are usually static
  - Doesn't always match up with real environment

Environment mapping problems

- Static cube maps aren't always perfect

Real-time cube maps

- You can render to a cube map!
  - Basically rendering 6 times
  - Each time to a different face
- Use resulting cube map as the reflection map
  - Too expensive to render for EACH object
  - Usually done on “main character”
- Can also render a cube shadow map
  - Used for point light shadow maps

Real-time reflection example

- Watch the very top of the car

Creating a Cubemap From Six Textures

Sample Code

Signature

```
// ------------------------------------------------------
// Loads six individual textures (the six faces of a cube map), then
// creates a blank cube map and copies each of the six textures to
// another face.  Afterwards, creates a shader resource view for
// the cube map and cleans up all of the temporary resources.
// ------------------------------------------------------
Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> Game::CreateCubemap(
        const wchar_t* right,
        const wchar_t* left,
        const wchar_t* up,
        const wchar_t* down,
        const wchar_t* front,
        const wchar_t* back)
{
```

Load the textures & get info

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

// We'll assume all of the textures are the same color format and resolution,
// so get the description of the first shader resource view
D3D11_TEXTURE2D_DESC faceDesc = {};
textures[0]->GetDesc(&faceDesc);
```

Create the cube texture

```
// Describe the resource for the cube map, which is simply
// a "texture 2d array".  This is a special GPU resource format,
// NOT just a C++ array of textures!!!
D3D11_TEXTURE2D_DESC cubeDesc = {};
cubeDesc.ArraySize      = 6; // Cube map!
cubeDesc.BindFlags      = D3D11_BIND_SHADER_RESOURCE; // We'll be using as a texture in a shader
cubeDesc.CPUAccessFlags = 0; // No read back
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; // Only need 1
cubeDesc.MiscFlags      = D3D11_RESOURCE_MISC_TEXTURECUBE; // This is a CUBE, not 6 separate textures
cubeDesc.Usage          = D3D11_USAGE_DEFAULT; // Standard usage
cubeDesc.SampleDesc.Count   = 1;
cubeDesc.SampleDesc.Quality = 0;

// Create the actual texture resource
ID3D11Texture2D* cubeMapTexture = 0;
device->CreateTexture2D(&cubeDesc, 0, &cubeMapTexture);
```

Loop and copy textures to cube

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

	// Copy from one resource (texture) to another
	context->CopySubresourceRegion(
		cubeMapTexture, // Destination resource
		subresource,    // Dest subresource index (one of the array elements)
		0, 0, 0,        // XYZ location of copy
		textures[i],    // Source resource
		0,  // Source subresource index (we're assuming there's only one)
		0); // Source subresource "box" of data to copy (zero = whole thing)
}
```

Create final SRV, clean up, return

```
// 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 texture
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, &srvDesc, cubeSRV.GetAddressOf());

// Now that we're done, clean up the stuff we don't need anymore
cubeMapTexture->Release();  // Done with this particular reference (the SRV has another)
for (int i = 0; i < 6; i++)
          textures[i]->Release();

// Send back the SRV, which is what we need for our shaders
return cubeSRV;
}
```