GRAPHICS PROGRAMMING I

HARDWARE RASTERIZATION PART II

DirectX: Camera

- Last week we rendered a triangle that was defined in homogenous coordinates (NDC). Let's adjust our code and render a triangle that is defined in world space.
  - Vector3(0, 3, 2), Vector3(3, -3, 2) & Vector3(-3, -3, 2), using index buffer { 0, 1, 2 }
- Just as with the software rasterizer, we need to transform our vertices into projection space (before the perspective divide) before we pass it into the rasterization stage. Luckily for us, the rasterizer stage is done by DirectX, this includes the perspective divide.
- We just need to define the correct matrix. You've already done this! We must make a WorldViewProjection matrix, pass it to our shader and transform every vertex that passes the vertex shader stage. As with the software rasterizer, we create the View matrix based on our camera's ONB and our Projection matrix based on the camera settings.
- Add a camera class to your project and make sure you have the following functions:
  - Matrix GetViewMatrix()
  - Matrix GetProjectionMatrix()
- You can also implement/use the Matrix::CreateLookAtLH & Matrix::CreatePerspectiveFovLH functions

DirectX: Camera

- Next, we need to “transfer” the matrix from CPU to GPU so we can use it in our shader.
- As mentioned last week, setting up the pipeline for a **draw call** is done through the **device context** using **resource views**. When using shaders, there are several options, but these are the common ones:
  - Constant Buffer View : ID3D11Buffer → CBuffer and TBuffer
  - Unordered Access View (UAV) : ID3D11UnorderedAccessView
  - Shader Resource View (SRV) : ID3D11ShaderResourceView
- We are going to use the **DX11Effects framework** which has some predefined types that will make our lives a bit easier.
- Update your shader by adding a **global** variable that represents your WorldViewProjection matrix. In your effect class get a ID3DX11EffectMatrixVariable that links to this variable. You can do this using the following code:

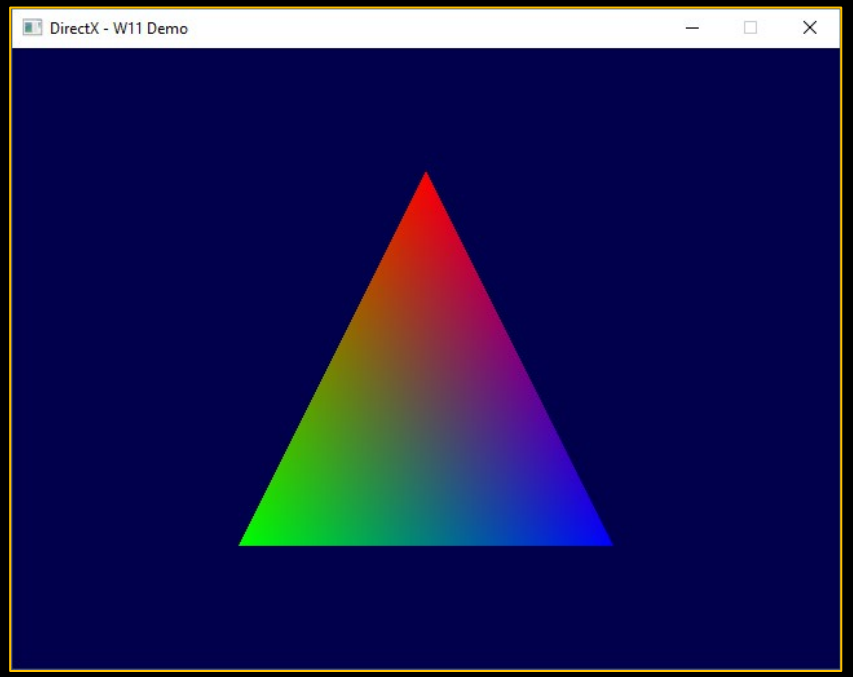
```
float4x4 gWorldViewProj : WorldViewProjection;
```

```
m_pMatWorldViewProjVariable = m_pEffect->GetVariableByName("gWorldViewProj")->AsMatrix();
if (!m_pMatWorldViewProjVariable->IsValid())
{
    std::wcout << L"m_pMatWorldViewProjVariable not valid!\n";
}
```

DirectX: Camera

- Inside your Vertex Shader Function, you now have to transform the input Position with the WorldViewProjection matrix (https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-mul)
- The last thing you need to do is update the data every frame using the SetMatrix(...) function of the (c++ side) matrix effect variable, same as you update the vertex buffer etc. Using your camera, you can build the WorldViewProjection matrix that you then pass to that function.
  Hint, you'll have to reinterpret the Matrix data…

Camera
FovAngle: 45.f
Origin: {0.f, 0.f, -10.f}



DirectX: Textures

- Let's upgrade our triangle into a quad, using texture (UV) coordinates, to be able to preview a texture. Just as with the software rasterizer, update your vertex structure accordingly
  (in both C++ and HLSL). For the UV coordinates you can use the semantic TEXCOORD.
- This also affects your vertex input layout ! Fix this by adding another element! ☺
- As with the software rasterizer, create a texture class that loads a texture from memory using IMG_Load. The only difference is that we now need to create a DirectX resource and a resource view. Do this resource/resource view creation when you load the texture, so only once!
- After pushing the data from the SDL_Surface to the resource, the SDL_Surface is no longer needed in memory, thus free it using SDL_FreeSurface.
- Make sure your texture class has a getter function to retrieve the ID3D11ShaderResourceView you've just loaded!

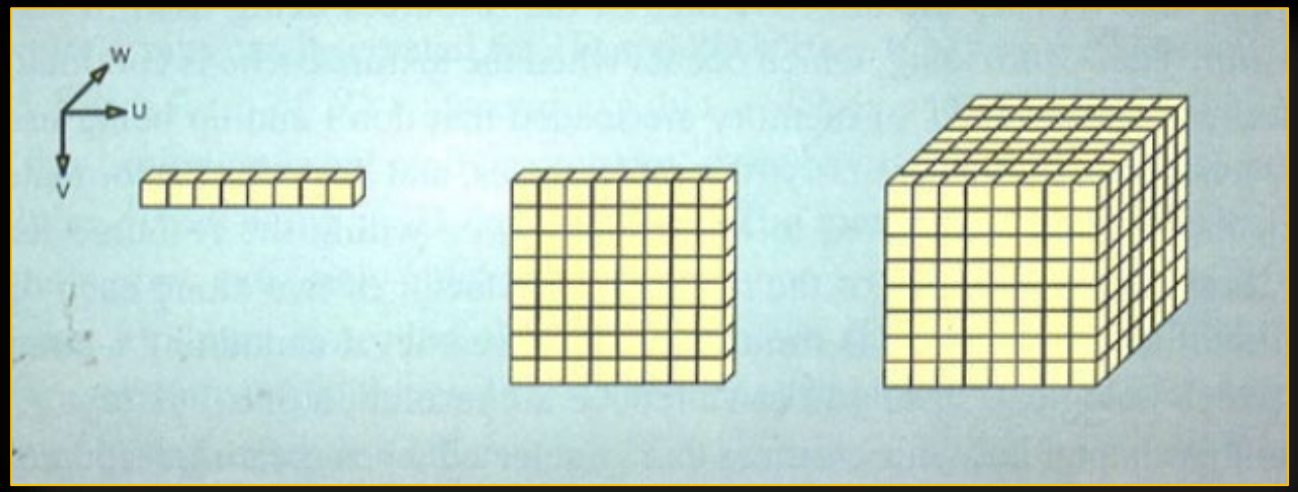
DirectX: Textures

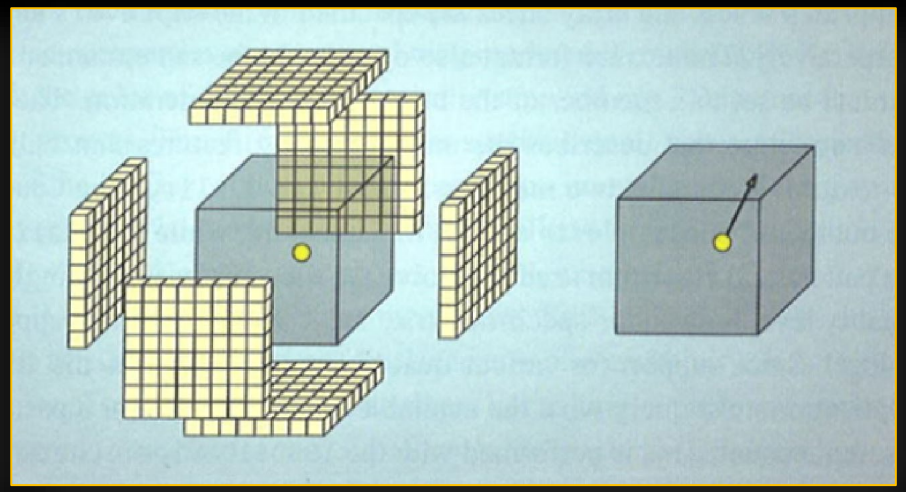
```
DXGI_FORMAT format = DXGI_FORMAT_R8G8B8A8_UNORM;
D3D11_TEXTURE2D_DESC desc{};
desc.Width = pSurface->w;
desc.Height = pSurface->h;
desc.MipLevels = 1;
desc.ArraySize = 1;
desc.Format = format;
desc.SampleDesc.Count = 1;
desc.SampleDesc.Quality = 0;
desc.Usage = D3D11_USAGE_DEFAULT;
desc.BindFlags = D3D11_BIND_SHADER_RESOURCE;
desc.CPUAccessFlags = 0;
desc.MiscFlags = 0;

D3D11_SUBRESOURCE_DATA initData;
initData.pSysMem = pSurface->pixels;
initData.SysMemPitch = static_cast<UINT>(pSurface->pitch);
initData.SysMemSlicePitch = static_cast<UINT>(pSurface->h * pSurface->pitch);

HRESULT hr = pDevice->CreateTexture2D(&desc, &initData, &m_pResource);
```

```
D3D11_SHADER_RESOURCE_VIEW_DESC SRVDesc{};
SRVDesc.Format = format;
SRVDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
SRVDesc.Texture2D.MipLevels = 1;

hr = pDevice->CreateShaderResourceView(m_pResource, &SRVDesc, &m_pSRV);
```

DirectX: Textures

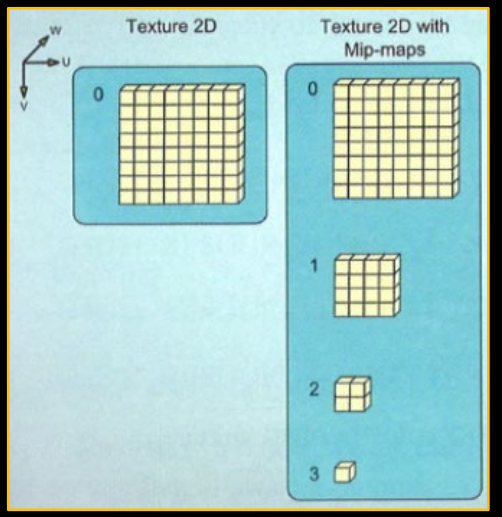
- We create a **Texture2D**, but there are different types of textures in DirectX:
  - Texture1D
  - Texture2D
  - TextureCube
  - Texture3D
  - … → https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-to-type
- It's important to realize they are just like **arrays**!

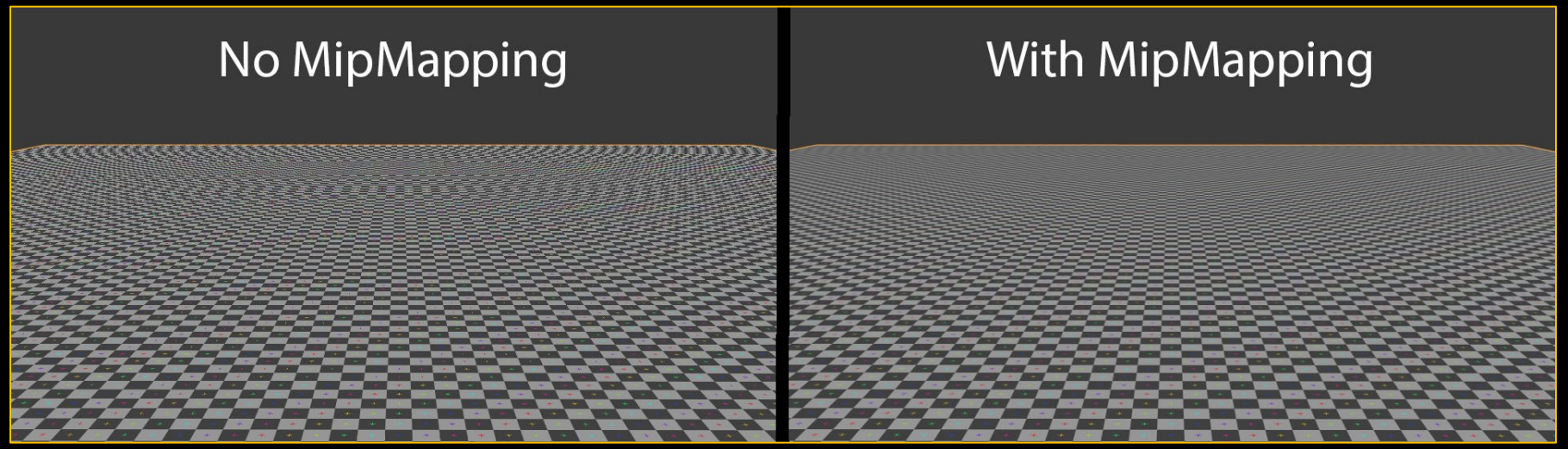




DirectX: Textures

- Texture can also have mipmaps. A **mipmap** is a sequence of textures, each of which is a progressively lower resolution representation of the same image. A high-resolution mipmap is used for objects that are close to the viewer. You can create them through the **Device Context** using **GenerateMips**.
- Mipmapping improves the quality of rendered textures at the **expense of using more memory**!





DirectX: Textures

- Finally, update your shader to accept a Texture2D shader resource view and update your Effect class to create a shader resource variable (ID3DX11EffectShaderResourceVariable) so we can push the data to the GPU. This type is from the external Effect Framework, just as the matrix!

```
Texture2D gDiffuseMap : DiffuseMap;
```

```
m_pDiffuseMapVariable = m_pEffect->GetVariableByName("gDiffuseMap")->AsShaderResource();
if (!m_pDiffuseMapVariable->IsValid())
{
    std::wcout << L"m_pDiffuseMapVariable not valid!\n";
}
```

- Make sure to bind the ShaderResourceView to the corresponding ID3D11ShaderResourceView effect variable, this can be achieved by using the SetResource(...) function.

```
void Effect_PosTex::SetDiffuseMap(Texture* pDiffuseTexture)
{
    if (m_pDiffuseMapVariable)
        m_pDiffuseMapVariable->SetResource(pDiffuseTexture->GetSRV());
}
```

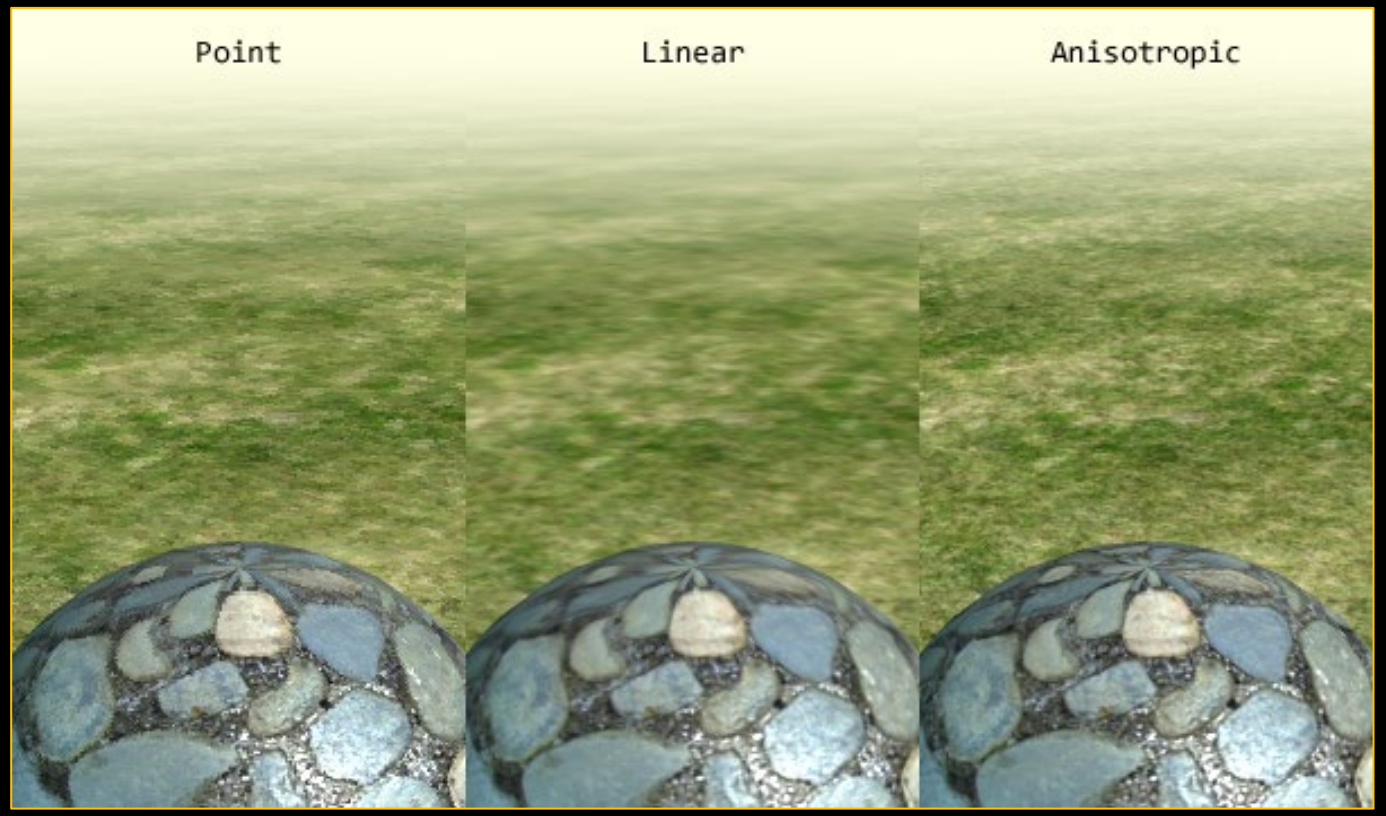
DirectX: Textures

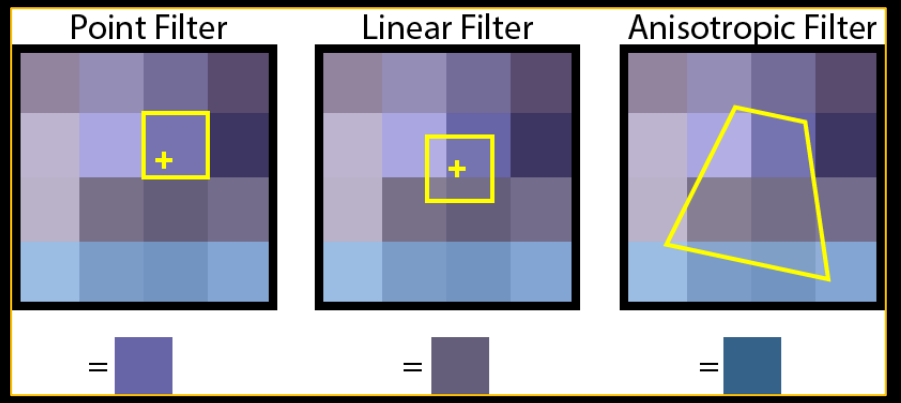
- Don't forget to release all the resources and resource views in the destructor! ☺
- As you can see, DirectX is all about resource management and setting up the state of the pipeline before a draw call. It's not that different from our software rasterizer.
- So how do we render now? How do we sample in our shader?
- In HLSL you must define how you want to sample using a SamplerState. This state is just a struct that keeps track of all the settings. We will again use a global variable.
  - https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-to-sample

```
SamplerState samPoint
{
    Filter = MIN_MAG_MIP_POINT;
    AddressU = Wrap; //or Mirror, Clamp, Border
    AddressV = Wrap; //or Mirror, Clamp, Border
};
```

- Let's go over some of these settings.

DirectX: Textures





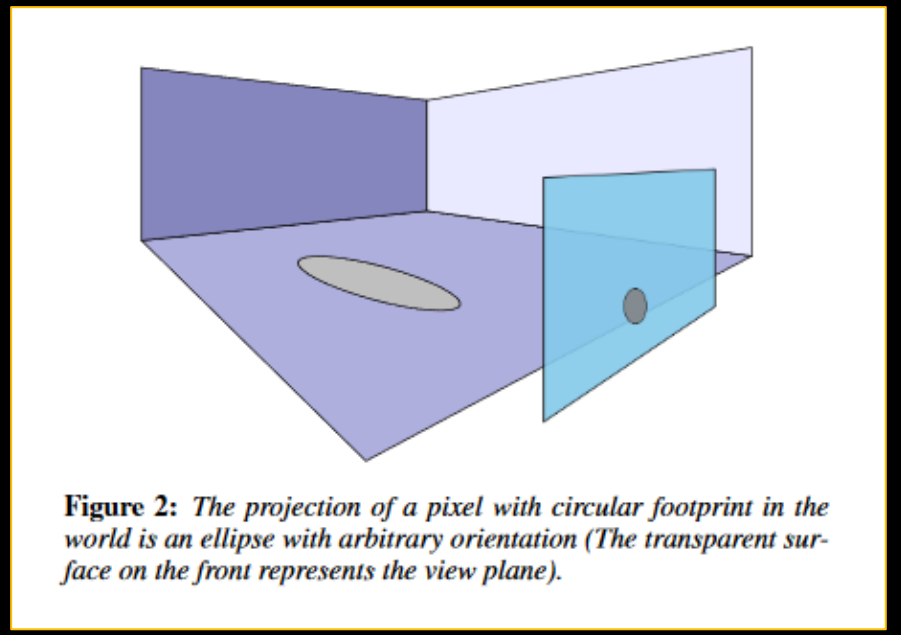
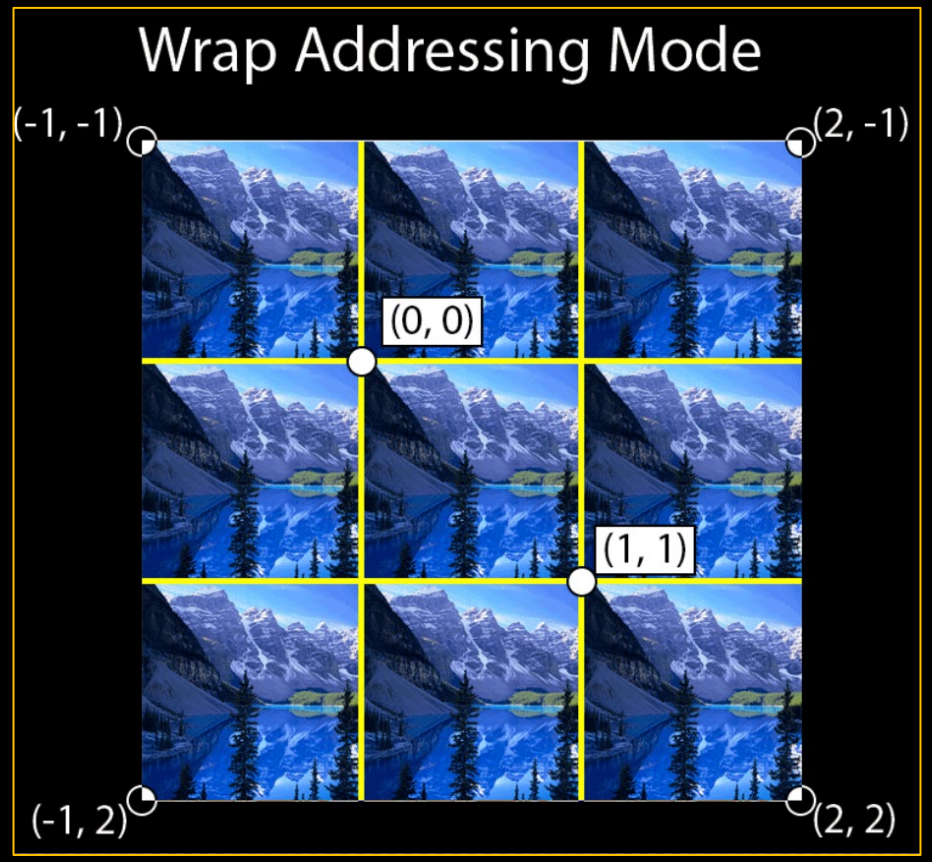
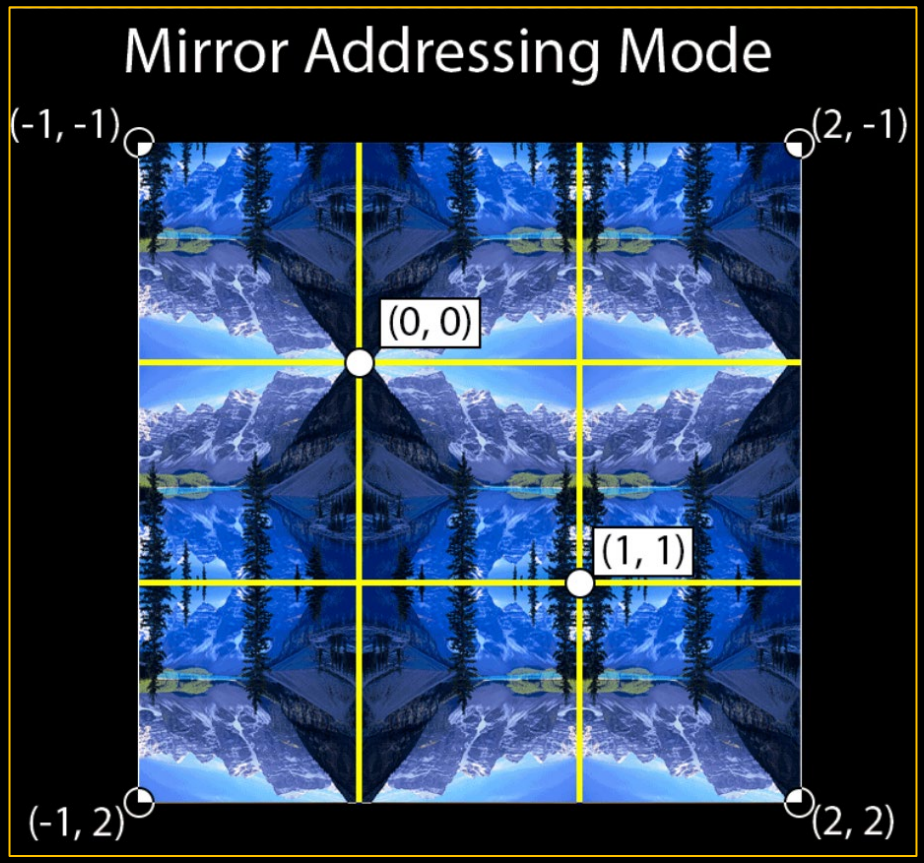


Figure 2: *The projection of a pixel with circular footprint in the world is an ellipse with arbitrary orientation (The transparent surface on the front represents the view plane).*

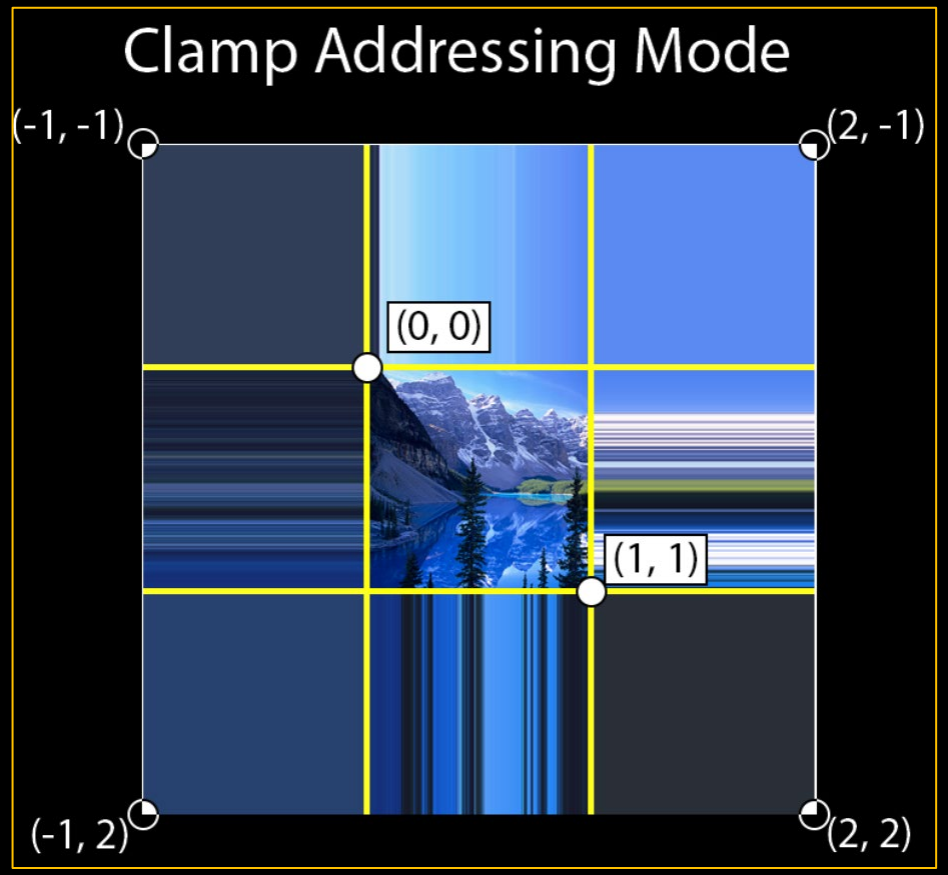
1. High Quality Elliptical Texture Filtering on GPU

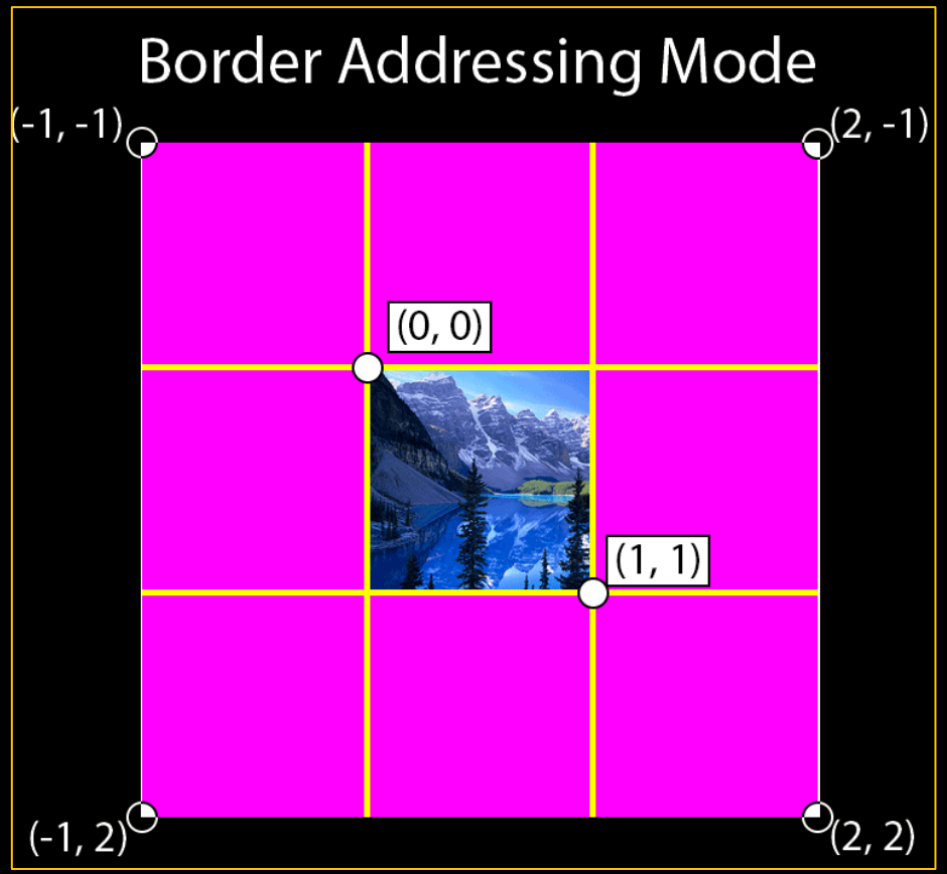
DirectX: Textures





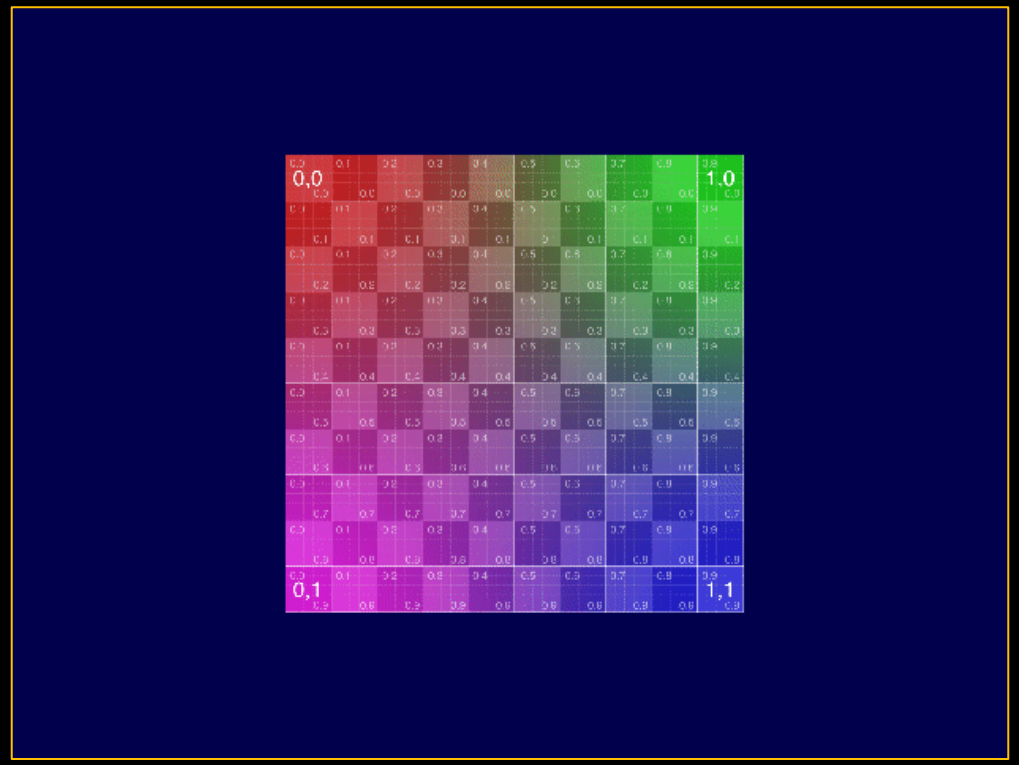
DirectX: Textures





DirectX: Textures

- With the SamplerState defined in HLSL you can now sample the Texture2D using the **Sample(...)** function. Do this in the **PixelShader** and return the float4 value.
  - https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-to-sample
- Load the texture from a few weeks ago, update the shader variables and draw the quad.



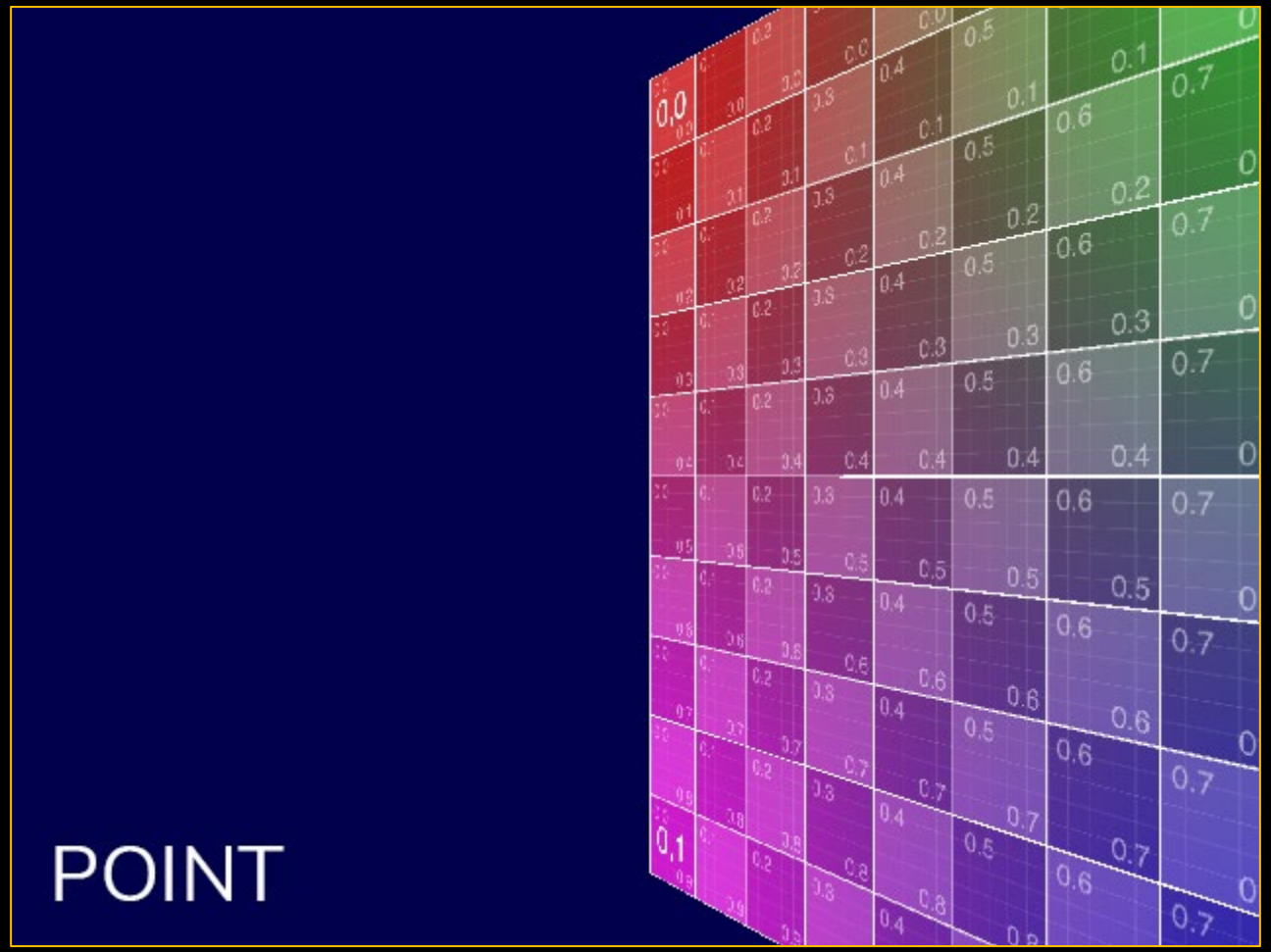
DirectX: Textures

- Implement three Sampler States with different Filtering Methods: Point, Linear and Anisotropic. Make sure you can toggle between them using the 'F2' key.
- Hint: define three different techniques with three different pixel shaders that use a different sample method. Do reuse calculations in the pixel shader by writing separate functions. ☺
  Capture all techniques in the effect class and when rendering, render using the correct technique! Basically, change the code below allowing you to toggle between techniques.

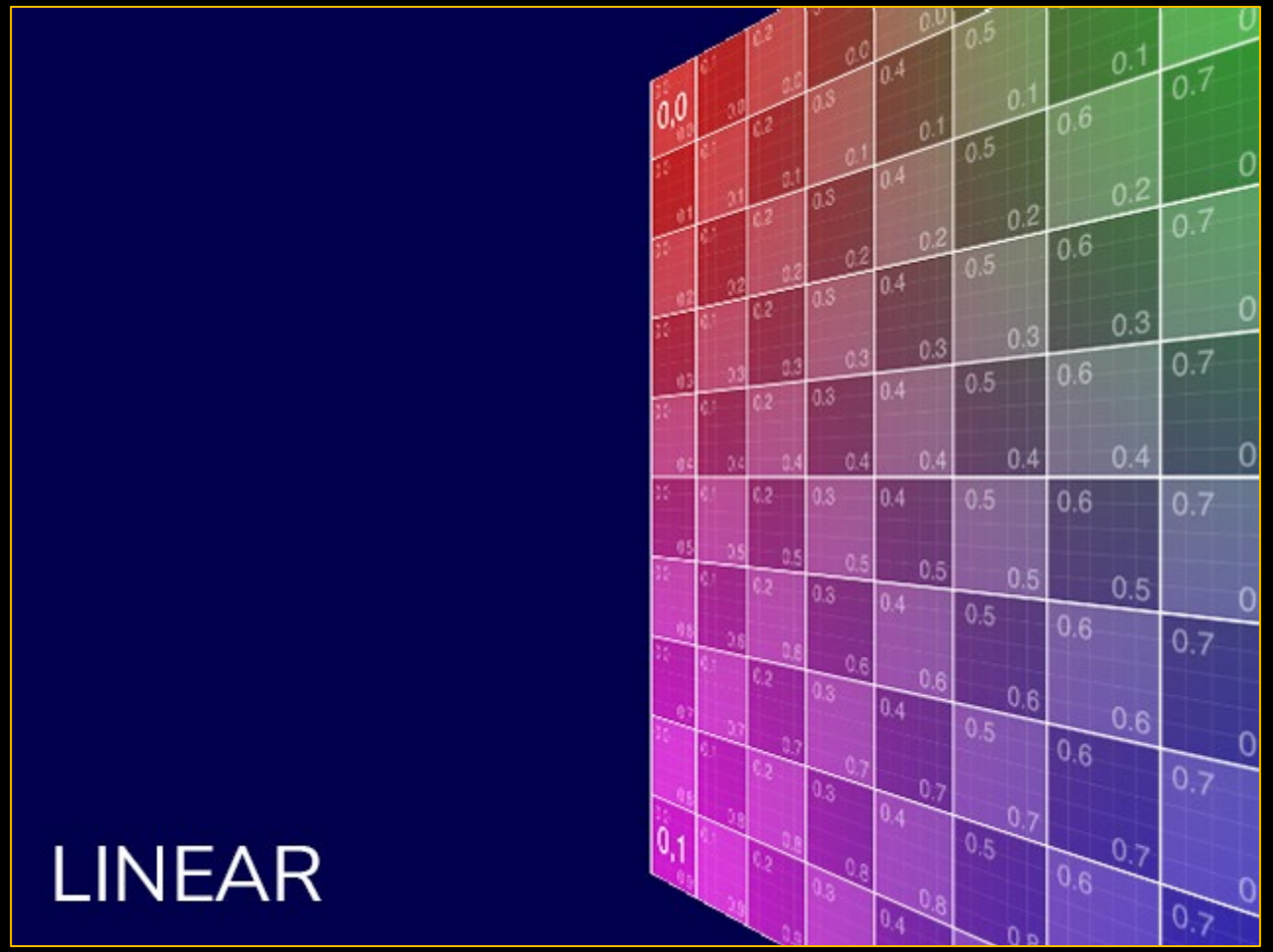
```
// Render a triangle
D3DX11_TECHNIQUE_DESC techDesc;
m_pEffect->GetTechnique()->GetDesc(&techDesc);
for (UINT p = 0; p < techDesc.Passes; ++p)
{
    m_pEffect->GetTechnique()->GetPassByIndex(p)->Apply(0, pDeviceContext);
    pDeviceContext->DrawIndexed(m_AmountIndices, 0, 0);
}
```

- Alternative : SamplerStates can also be changed/updated from c++ (this way you only need a single Technique of course) [ID3D11SamplerState]

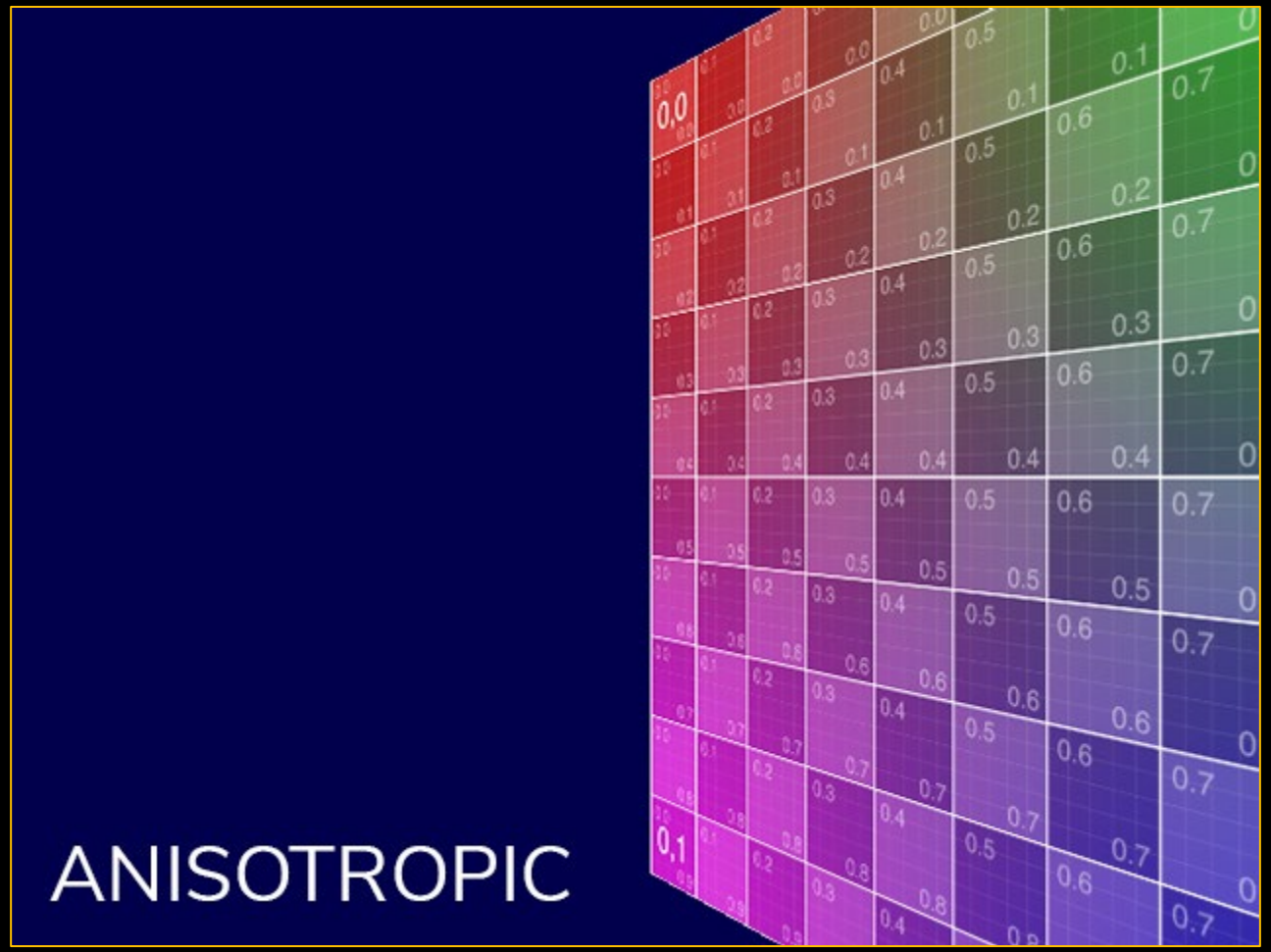
DirectX: Textures



DirectX: Textures



DirectX: Textures



DirectX: Mesh

- Use the vehicle mesh and diffuse texture from the previous weeks. Load the model with your parser and use those vertex and index buffers.
  Hint: disable the normal/tangent parsing for now. (Pos/uv & DiffuseMap only)

Camera
FovAngle: 45.f
Origin: {0.f, 0.f, -50.f}

Rotation: 90DEG/S

GOOD LUCK!