Textures

Digital images

- Images (pictures) are visual representations of things
- Digital images are grids of colors
- Essentially 2D arrays of color values
- Pixel = Picture Element

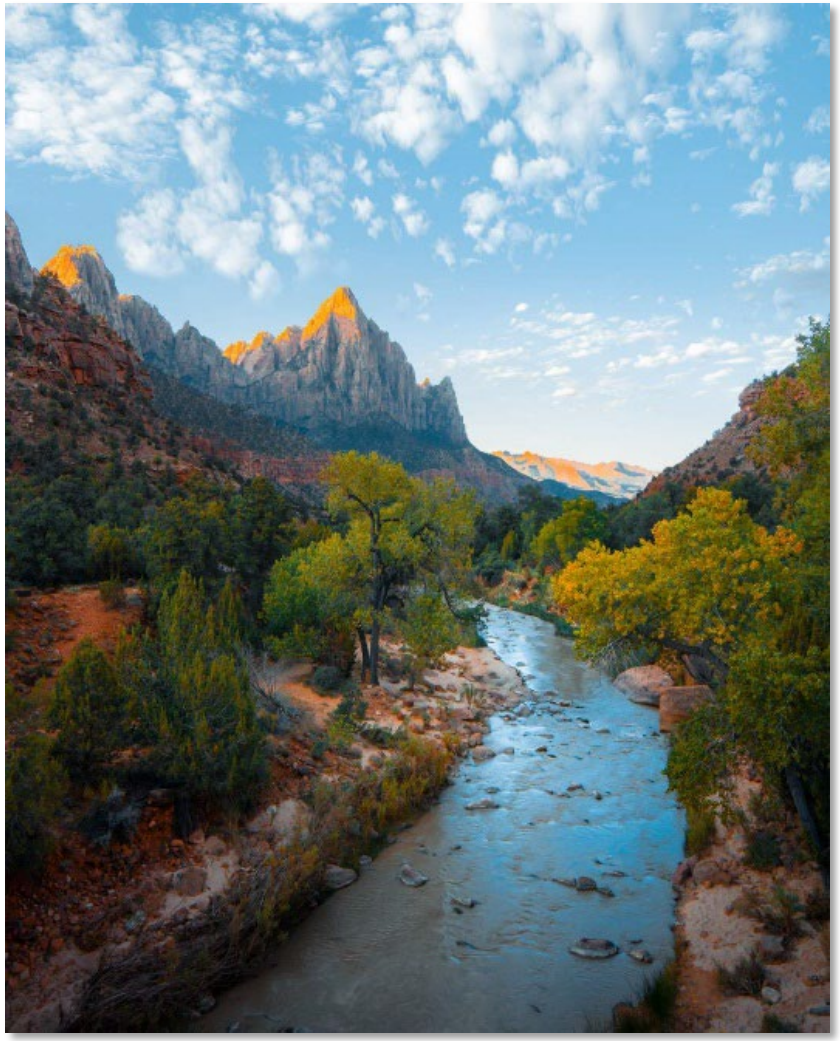
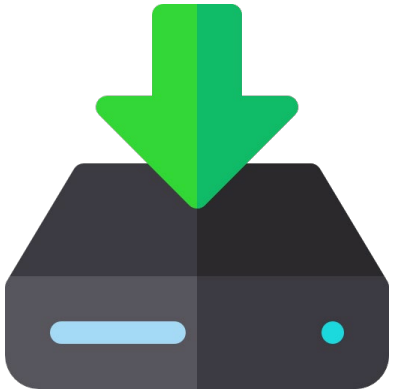
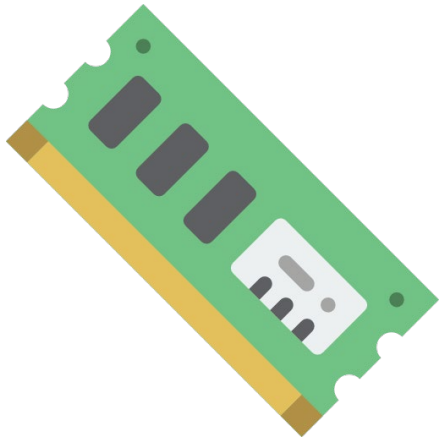


Image storage & file formats



- Many file formats exist for image storage
- **Uncompressed** formats store every single pixel color (BMP)
- **Lossless compression** formats store less data (PNG, GIF)
  - Original image can be reconstructed flawlessly
- **Lossy compression** formats trade accuracy for size (JPG)
  - Some data is permanently lost
  - But overall file size is much smaller

Digital images in RAM



- Images are uncompressed in RAM
- Regardless of file format or compression
- Once loaded, all 512x512 images take up same space
  - JPG, BMP, PNG, etc.
  - Regardless of compression!
  - We're using final pixel colors
- Assets for games should be lossless to maintain quality

Textures

Some vocabulary

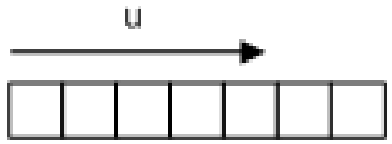
- **Texture** – a GPU resource holding a grid of pixel colors
- **Texel** – A texture element: one “pixel” of a texture
- **UV Coordinate** – an “address” in a texture
  - Where in the texture we’re getting a data from
- **Texture Mapping** – Mapping a texture to a surface
  - Most often a 2D image “wrapping around” a 3D mesh

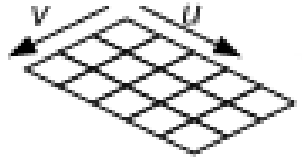
Textures

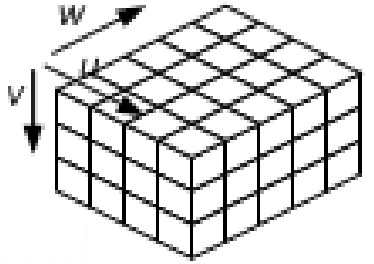
- GPU resources holding pixel data
- Essentially buffers of pixels
- In addition to basic buffer definition, also defines:
  - Color format
  - Width
  - Height
  - Array size
  - Mip levels

Texture dimensions

- Textures can be 1D, 2D or 3D
  - 1D textures are often used as gradients
  - 3D textures often hold volumetric data







- We'll be sticking with 2D textures for a while

Subresources – Subsets of a resource

- Two types of subresources for textures in particular:

- **1. Arrays of textures**
  - Each subresource is treated as its own texture
  - All must be the same size & color format

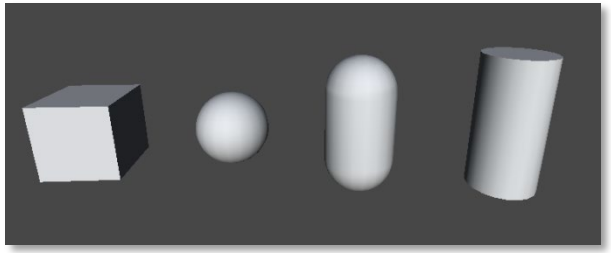
- **2. Mip levels**
  - Multiple, smaller versions of the same texture
  - Used for smooth texture sampling

- More on these soon!

Texture Mapping

Texture mapping

- Applying an images to our rasterized geometry

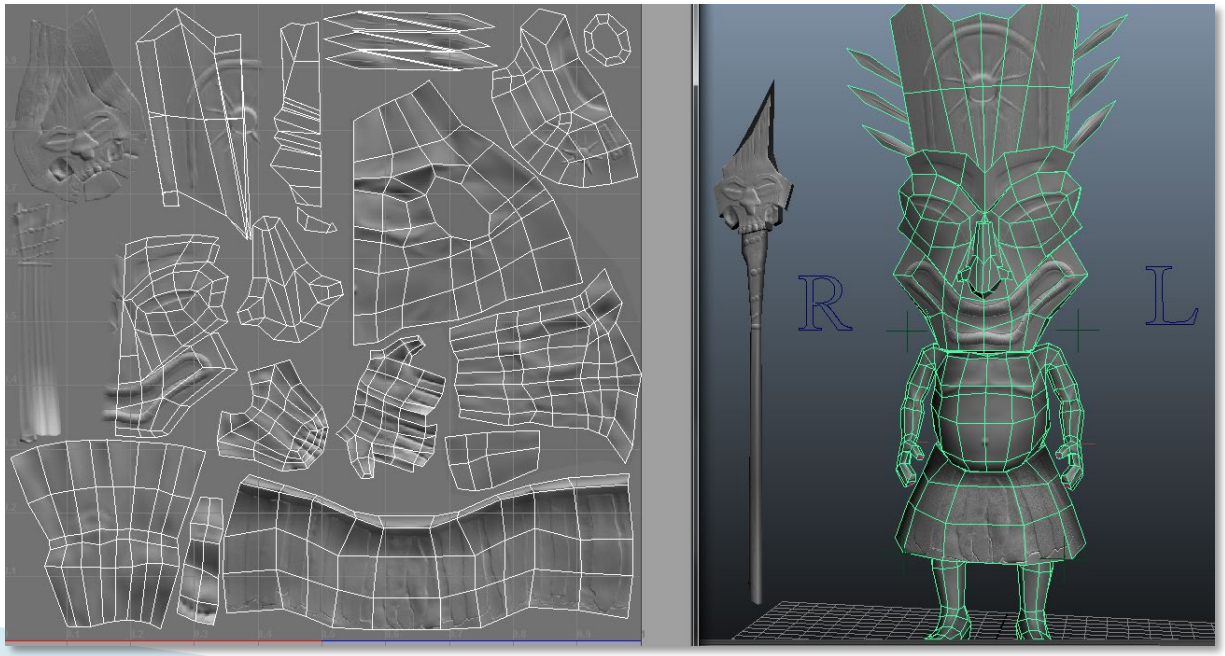


- Exact mapping is dictated by UV coordinates

UV coordinates

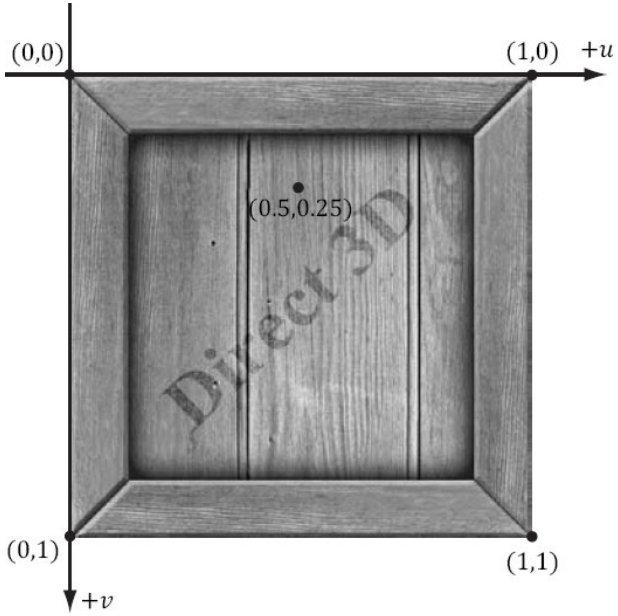
- Float2 stored at each vertex
  - Loaded from 3D model (or procedurally generated)

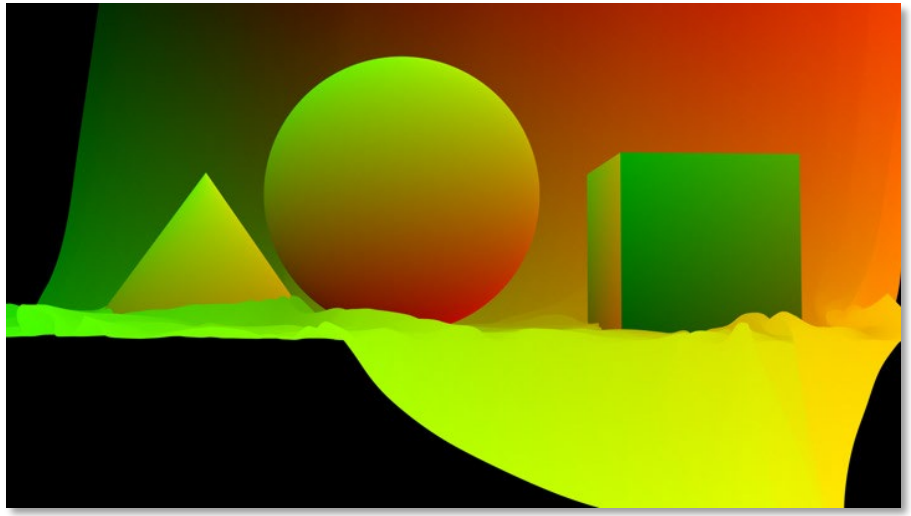
- Maps a texture region to individual triangles



UV coordinates

- Images have inherent UV coordinates
  - From (0,0) to (1,1)
  - Percent across the image
  - Resolution-agnostic addressing
- Usually range from 0 – 1
  - Could go outside that range
  - For certain effects (tiling)
- UV coords returned as colors →
  - Interpolated by rasterizer





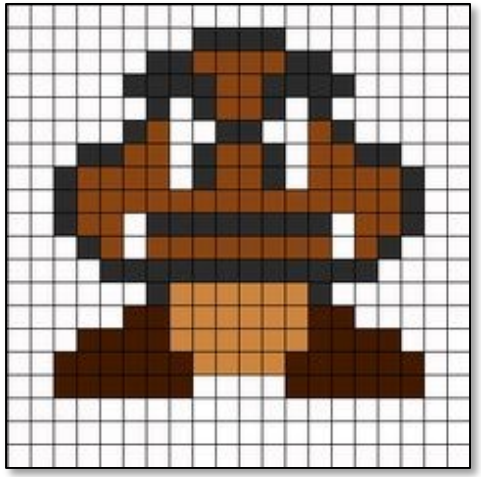
Working with Textures

In Direct3D 11

Basic D3D texture requirements for shaders

- **`ID3D11Texture2D*`**
  - Raw texture data
  - Not directly bound to the pipeline
- **`ID3D11ShaderResourceView*`**
  - Describes which texture subresources the pipeline can access
  - Used to bind texture resources to the pipeline
  - Need one of these *per texture*
- **`ID3D11SamplerState*`**
  - Options for sampling (choosing) pixels from texture
  - Need at least one of these *total*

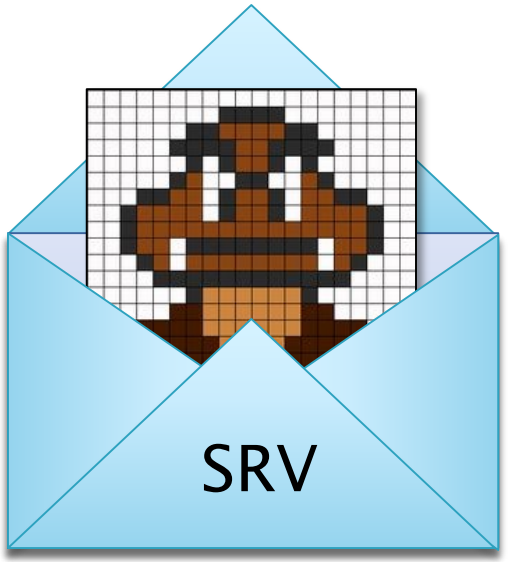
ID3D11Texture2D



- A Direct3D resource
  - Holds pixel data
  - Essentially an array of pixels
- Can contain subresources
  - Multiple groups of similar data
- Examples of subresources
  - Arrays of textures – multiple images in one GPU resource
  - Mipmaps – smaller versions of same image for interpolation

ID3D11ShaderResourceView

- Can't bind resources directly to pipeline
- Must create an SRV to describe the resource
  - Internally points to the resource
  - Describes which subresources can be accessed
- SRVs are then bound to the pipeline
- Note: SRVs do *not* dictate pixel regions!
  - Not all shader resources are textures



ID3D11SamplerState

- Defines sampling options for shader resources
  - Filter type
  - Addressing mode
  - Level of detail options
  - Maximum anisotropy
  - Etc.

- `ID3D11SamplerState*`
  - `device->CreateSamplerState( )`
  - Needs a `D3D11_SAMPLER_DESC`

DirectX Tool Kit

Our helper for loading image files

Creating texture resources

- Can be created programmatically
  - Similar to making vertex or index buffers
  - Fill out a description and ask DirectX to create
  - Pass in pixel data as an array of colors

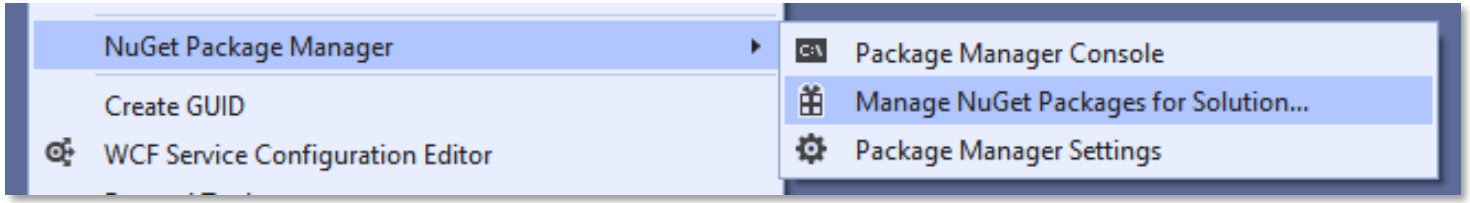
- Often we want to load pixel data from files
  - DirectX doesn't come with a texture loader
  - **DirectX Tool Kit** to the rescue

DirectX Tool Kit

- A collection of official DX helper classes
  - Texture loading
  - 2D sprite drawing
  - Basic gamepad input
  - And more
- We'll be using it for loading images (textures)
- Source can be found on github
  - https://github.com/Microsoft/DirectXTK

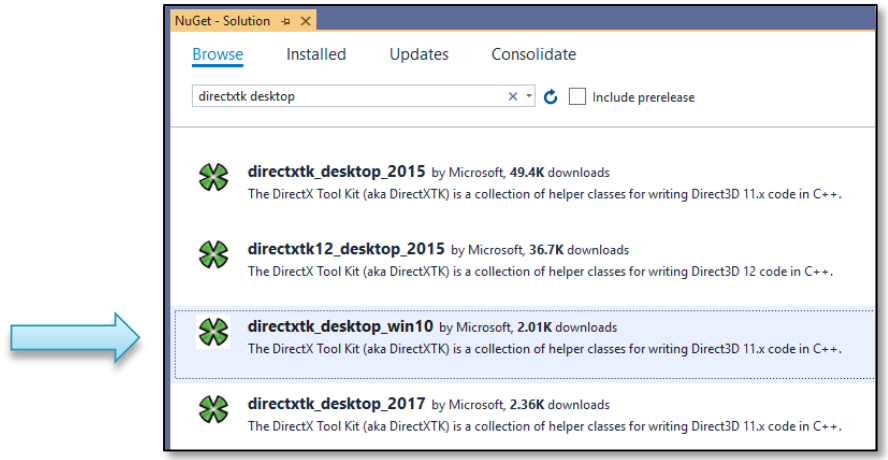
Acquiring the DirectX Tool Kit

- Can be added directly inside Visual Studio
- Available as a *NuGet package*
  - Tools > NuGet Package Manager > Manage…



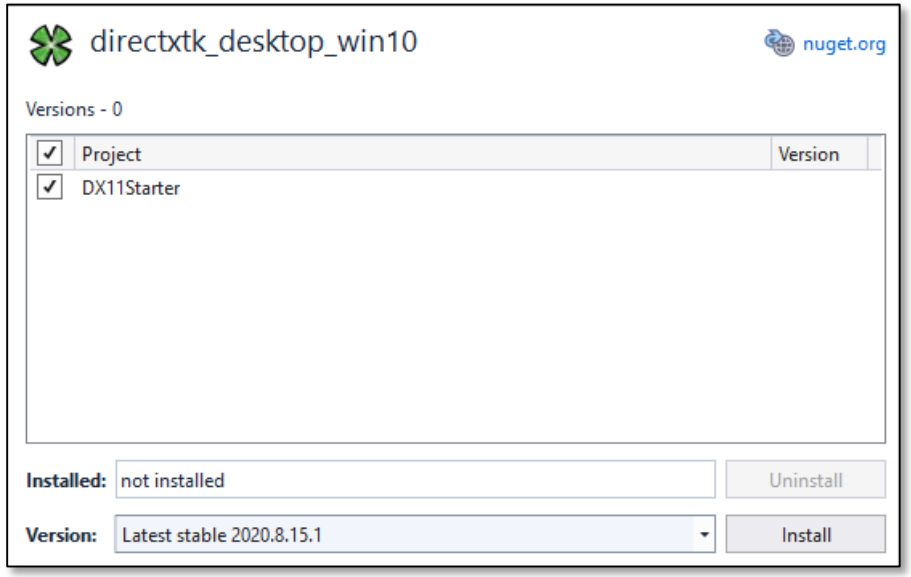
NuGet Package Manager

- Browse > Search “directxtk”
- Choose one of the following:
  - “directxtk_desktop_**win10**”
  - “directxtk_desktop_**2017**”
  - Both work w/ VS 2022



- Note: Do NOT choose directxtk12!!!

- On the right side of the window
  - Select “DX11Starter”
  - Click Install



Using DXTK – Texture loaders

- **WICTextureLoader** – Loads jpgs, pngs, etc.
  - "Windows Imaging Component"
  - Automatically loads textures *and creates SRVs!*

- Header: `#include "WICTextureLoader.h"`
- Usage: `DirectX::CreateWICTextureFromFile(…);`

- Want mip-maps? (Yes, you do)
  - Use overload that requires both *device* & *context* parameters
  - This will auto-generate mip-maps!

Textures & Shaders

Shader Requirements

- Add some new types of variables to your pixel shader
- SamplerState
- Texture2D

HLSL – Defining a SamplerState

- Create sampler state in C++
  - Use SimpleShader to send to GPU
- Define the sampler in your shader
  - Not part of a constant buffer!

```
SamplerState BasicSampler : register(s0);
```

HLSL – Defining a Texture

- Shader Resource View in C++
  - Use SimpleShader to send to DirectX

- Define a Texture variable directly in shader
  - Not part of a constant buffer!

```
// A 2D texture object bound to the
// first texture register on the GPU
Texture2D DiffuseTexture : register(t0);
```

HLSL – Sampling a Texture

- Textures have a “Sample” function
  - Two parameters: SamplerState & UV

```
float4 color = DiffuseTexture.Sample(
                        BasicSampler,
                        uvCoords);
```

- **DiffuseTexture** is a Texture2D shader variable
- **BasicSampler** is a SamplerState shader variable
- **uvCoords** is a float2

Texture Sampling

More vocabulary

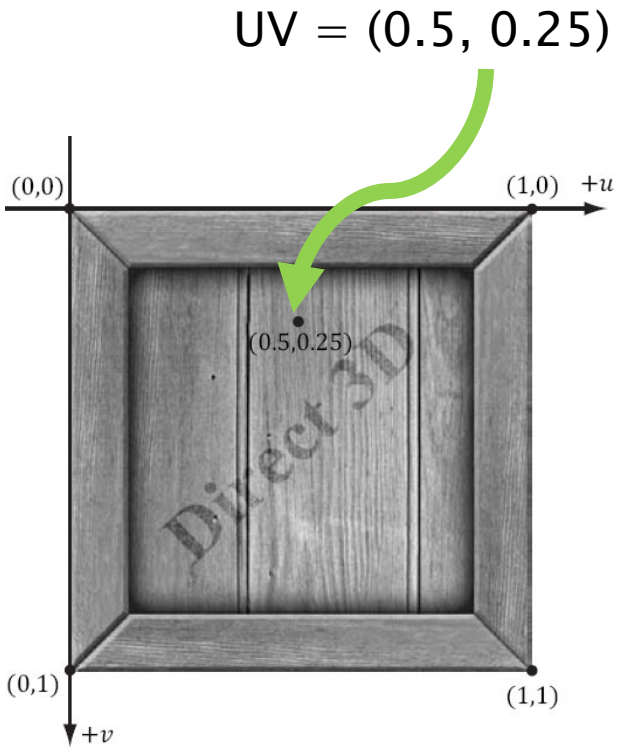
- **Texture Addressing**
  - Determining a final pixel position (based on UVs)
  - Sometimes coordinates go outside the image
- **Filtering**
  - Algorithm used to determine a color
  - When texture pixels and screen pixels aren't 1:1
- **Sampling**
  - Overall act of retrieving data from a texture
  - Based on UV coordinates, filtering options, etc.

Texture addressing

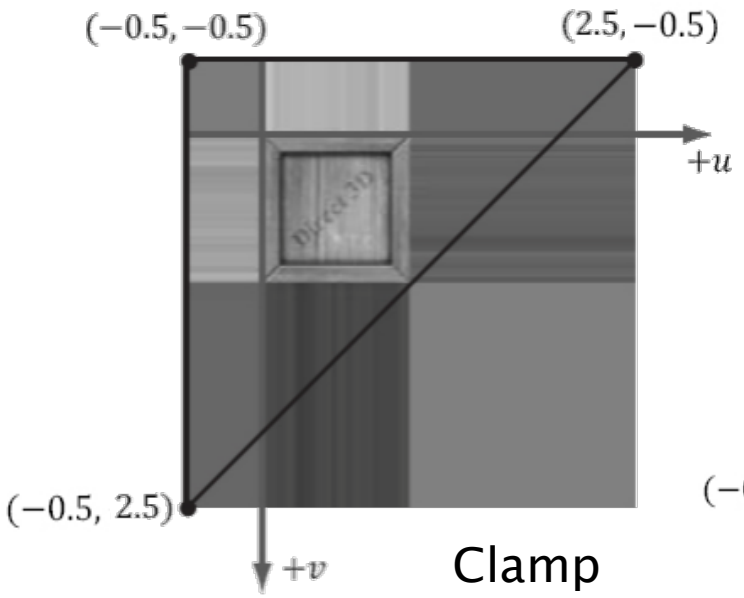
- Answers the question: “What if address is outside 0-1 range?”

- UV coordinates define where to sample
  - Basically an “address” inside a texture
  - Images have an inherent (0,1) UV range
  - UV coordinates could go outside that range

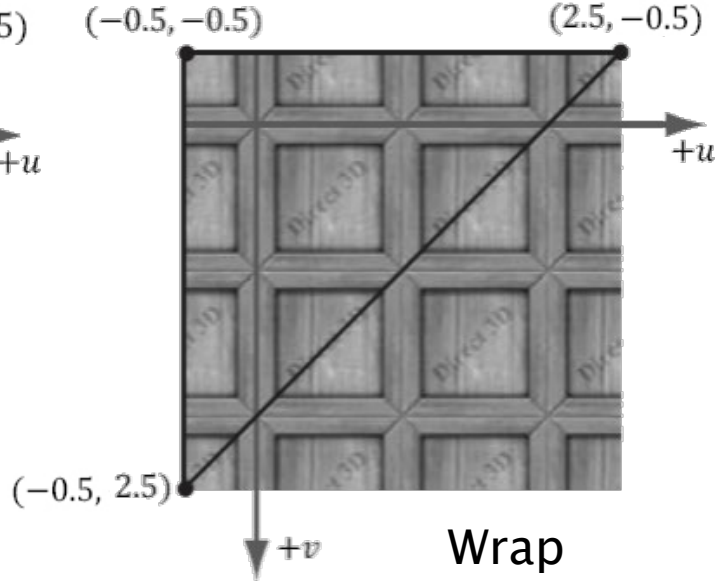
- GPUs support several options
  - Set via ID3D11SamplerState



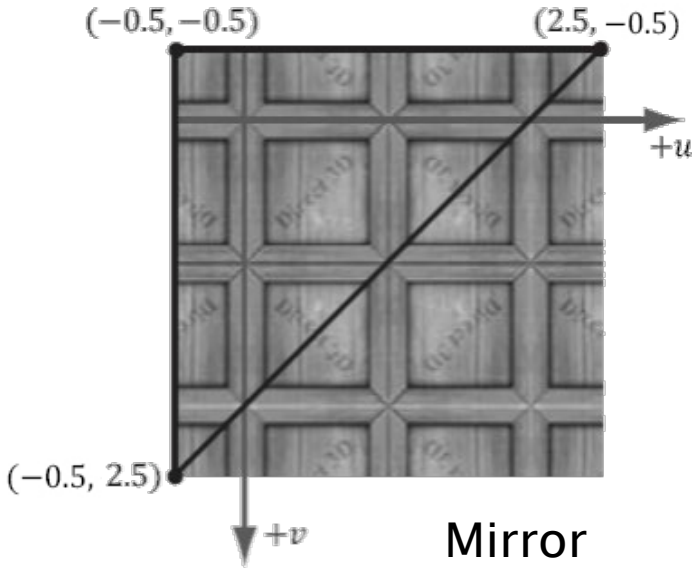
Texture address modes



Clamp

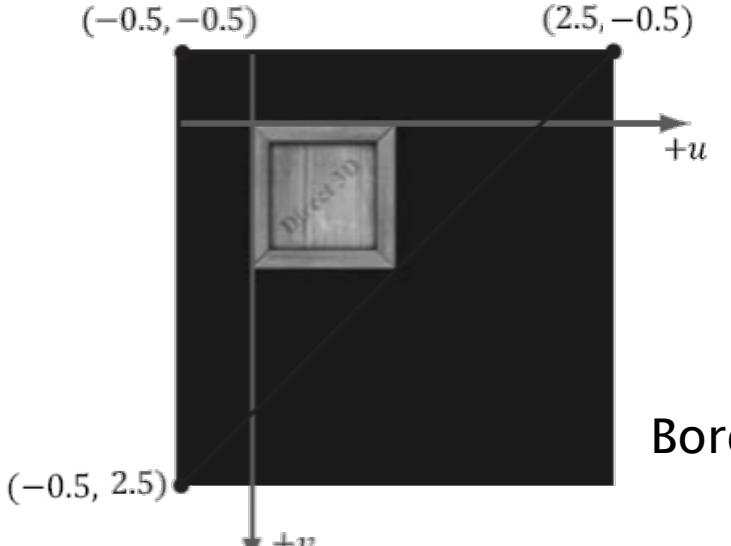


Wrap



Mirror

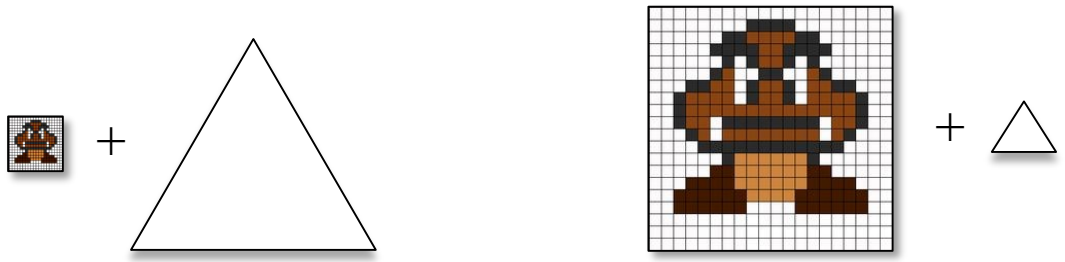
- Why go outside the [0 – 1] range?
  - Tiling
  - Scrolling textures
  - Other special texture effects



Border

Filtering

- Determines color to retrieve from a texture
- Triangles and textures rarely the same size
  - Pixels on our screen != pixels in a texture
- Use near-by texels to determine final color



Linear filtering

- Linearly interpolate the texels
  - Smooth change between colors
  - Can look blurry

Point filtering

- Use the nearest texel's color
  - No interpolation
  - Always looks "blocky"

What causes filtering?

- Triangle pixel area != texture pixel area
  - Can change as triangle moves
  - Can vary across a single triangle

- **Minification:**
  - More than one *texel* covers a *pixel*
  - Multiple texture colors must be turned into one

- **Magnification**
  - More than one *pixel* covers a *texel*
  - A single texture color is applied to multiple pixels

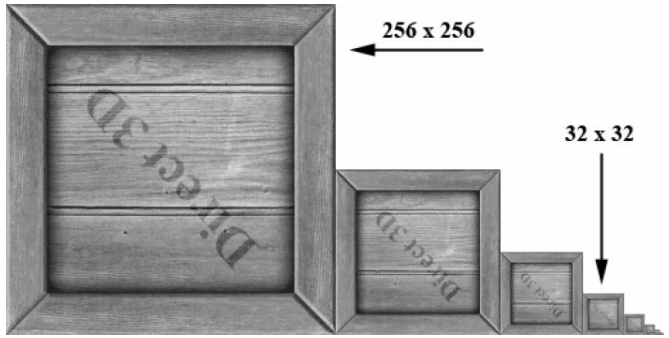
Problem: Extreme minification

- Minification can get extreme
  - 1024x1024 texture -> 16x16 screen area
  - Which 16 pixel colors get used?

- Results in texture aliasing and “shimmer”

Solution: Mipmapping

- **Mipmapping** utilizes premade smaller textures

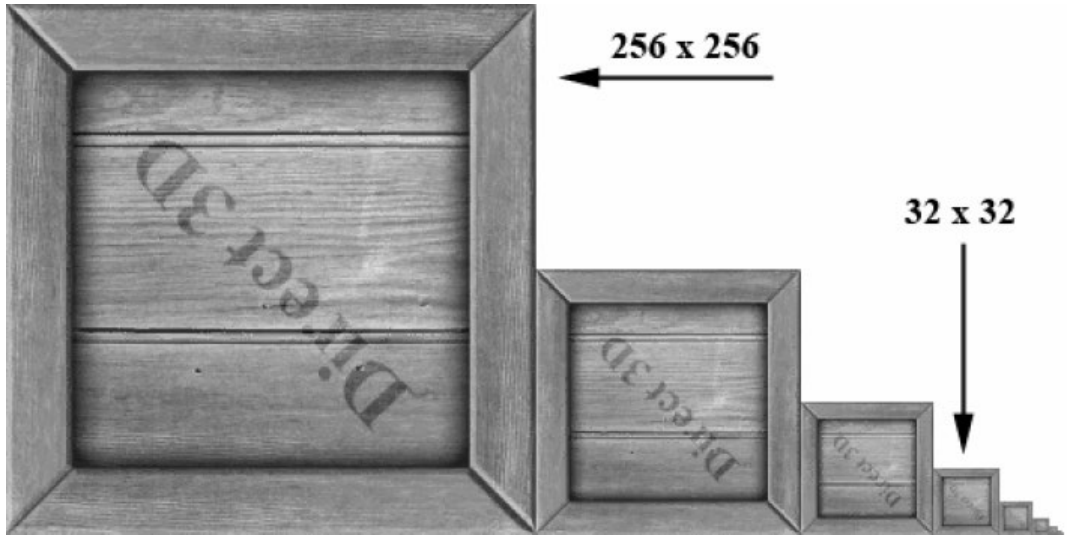


- GPU chooses most similarly sized texture
  - Automatically!
  - When minification becomes problematic
  - As long as a Mipmap Chain exists in your texture

Mipmap chain

- Series of pre-scaled & filtered versions of a texture
  - Can be generated at load-time (DirectXTK)
  - Or created by hand & stored in an image (.DDS)

- Successive images are ½ size in each dimension
  - Down to 1x1



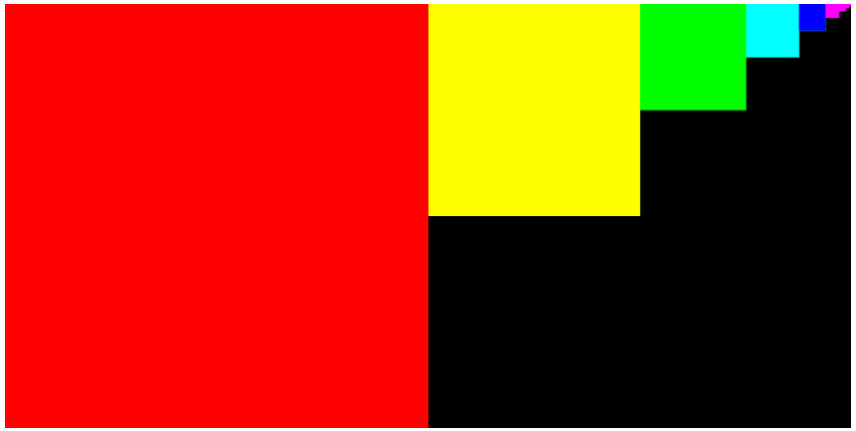
“Mipmap”?

- Where does this term come from?
- **MIP**: From the Latin “**Multum In Parvo**”
  - Meaning “Much in a small space”
- **Map**: As in bitmap
  - Or “texture map”
  - Basically: images
- Hence, Mipmap

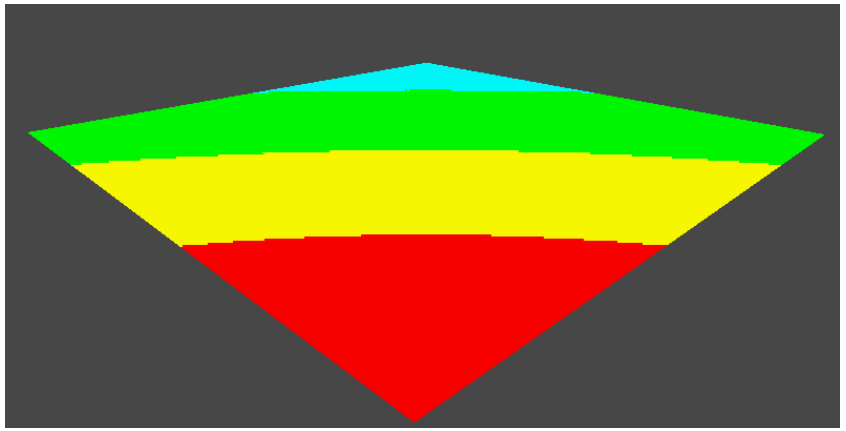
Mipmap Example

- Usually smaller versions of the same texture
- But to illustrate the point:

Mipmap with obvious levels

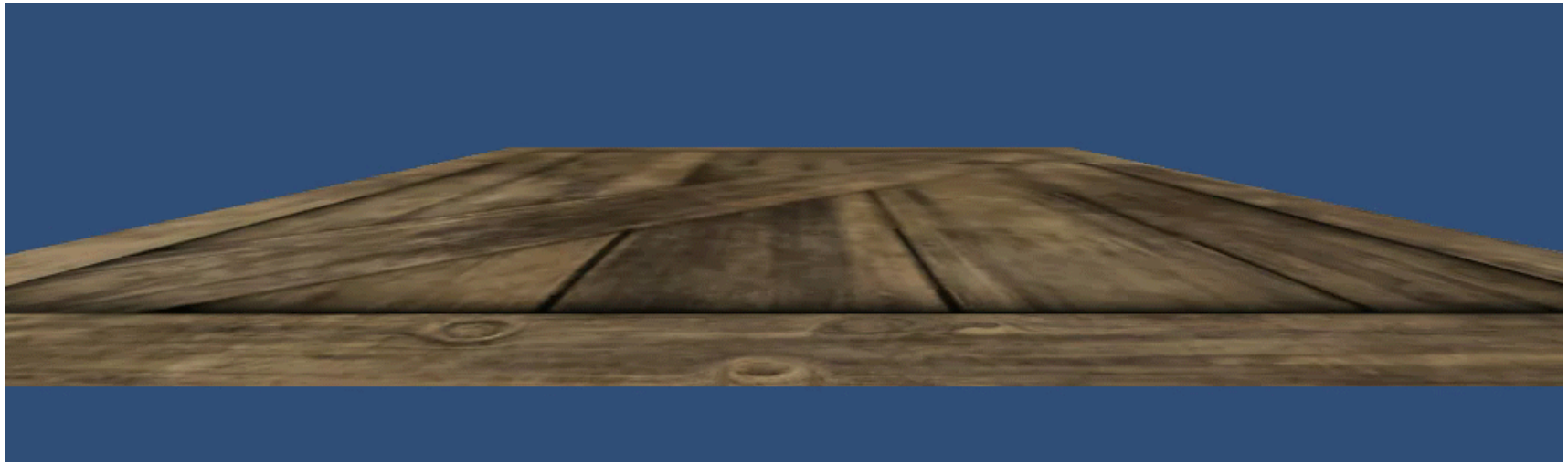


Automatically applied to a plane

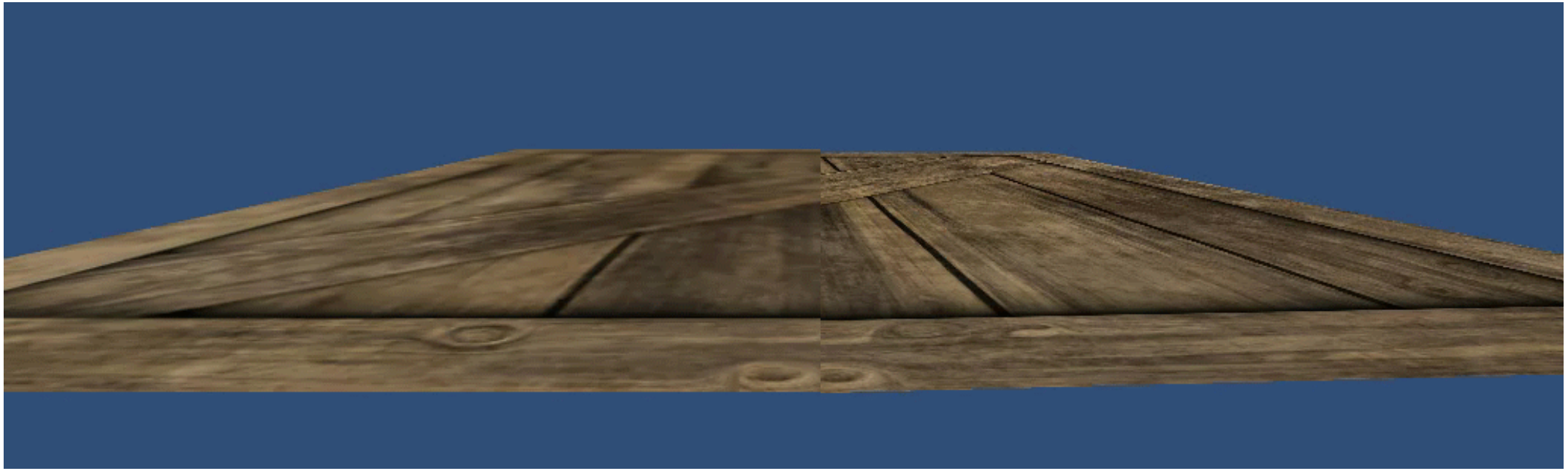


Mipmap Example

- Same plane & texture – With mipmaps
  - Removes texture aliasing and “shimmer”
  - Unfortunately results in “blurriness”



Mipmap vs. No Mipmap



Texture with
Mipmap Chain

Texture without
Mipmap Chain

Filtering Between Mipmap Levels

- Can be controlled with sampler options
- **Point** mipmap filtering
  - Selectes “nearest” mipmap level
  - Hard change between levels
- **Linear** mipmap filtering
  - Interpolates between mipmap levels
  - Smooth change

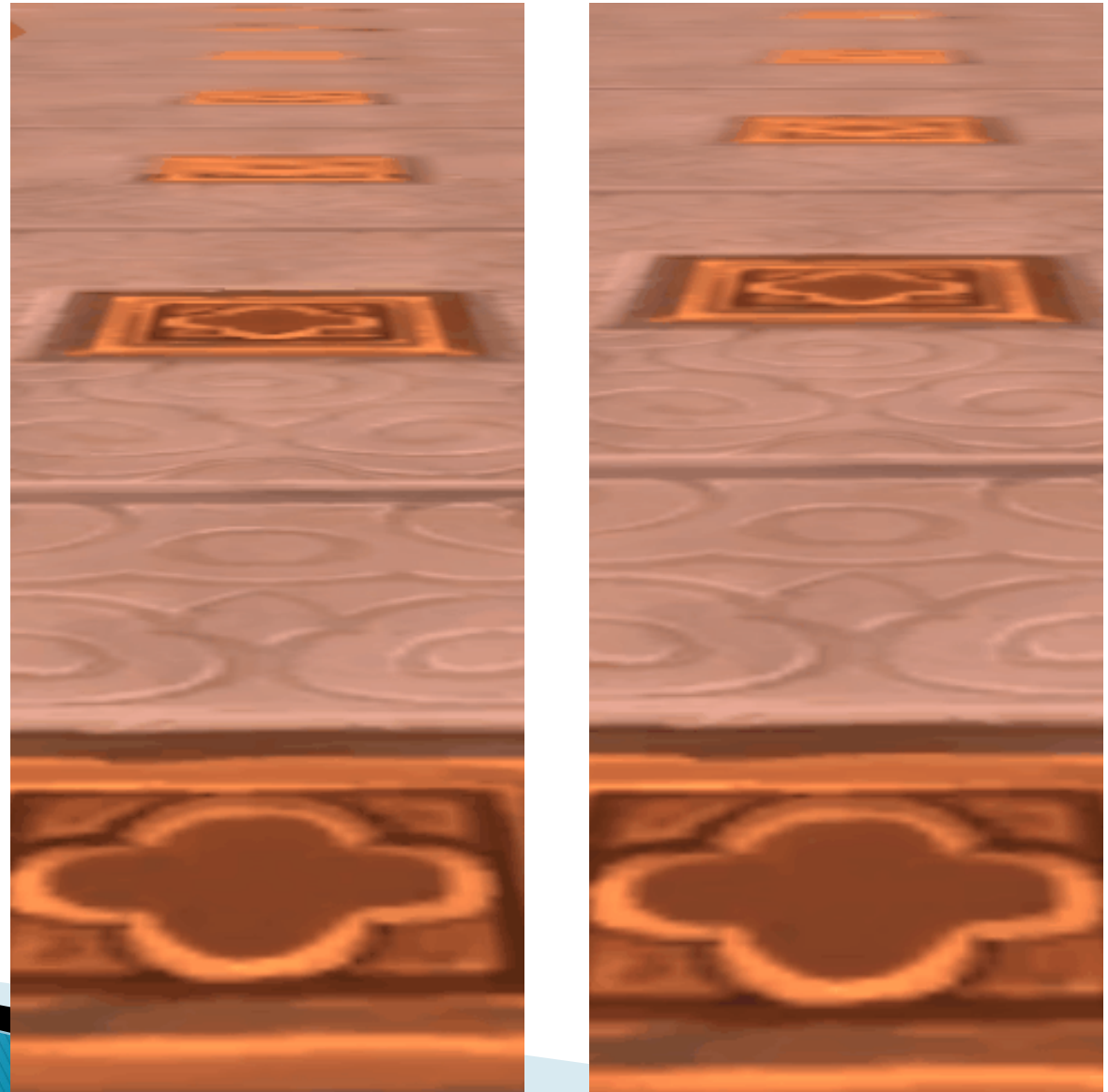
Point mipmap filtering

- Selects nearest mip level
- Results appear banded
  - Hard line when changing mips
  - Most obvious on ground
- **Bilinear** filtering
  - Linear for minification
  - Linear for magnification
  - Point for mip levels

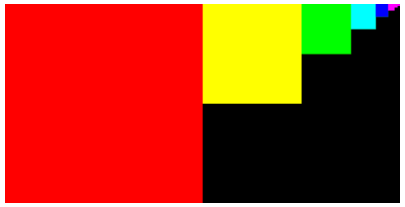
Linear mipmap filtering

- Blends between mip levels
- Results appear smooth
  - However, still blurry
- **Trilinear** filtering
  - Linear for minification
  - Linear for magnification
  - Linear for mip levels

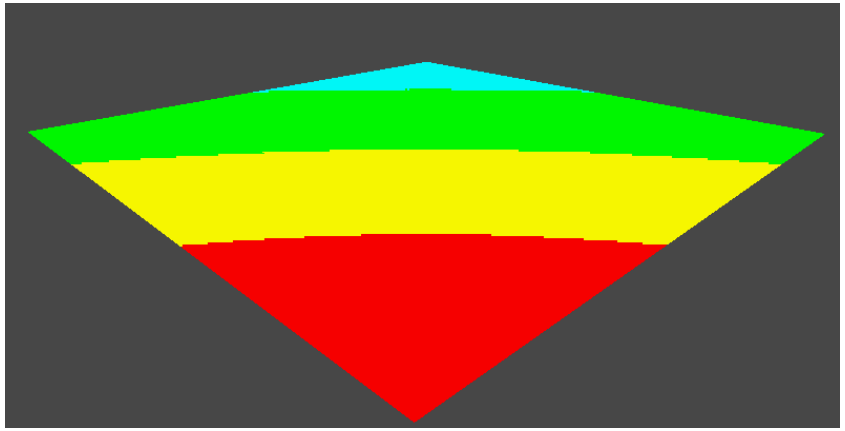
Point vs Linear mipmap filtering



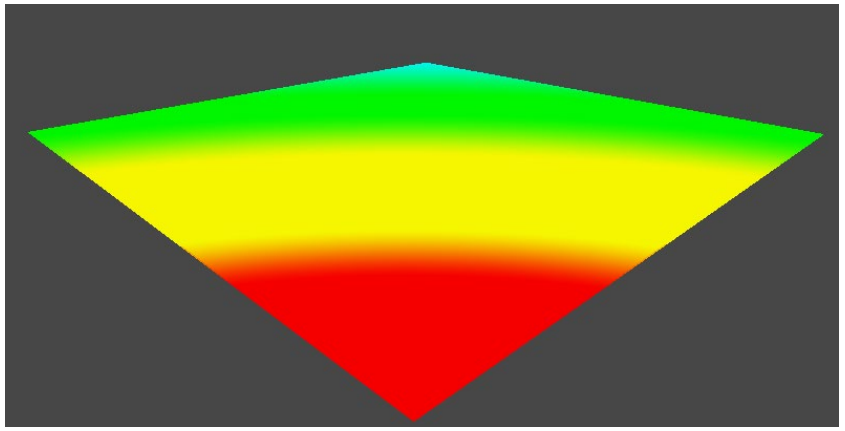
Point vs Linear mipmap filtering



Mipmap chain



Point



Linear

Filtering on orthogonal surfaces

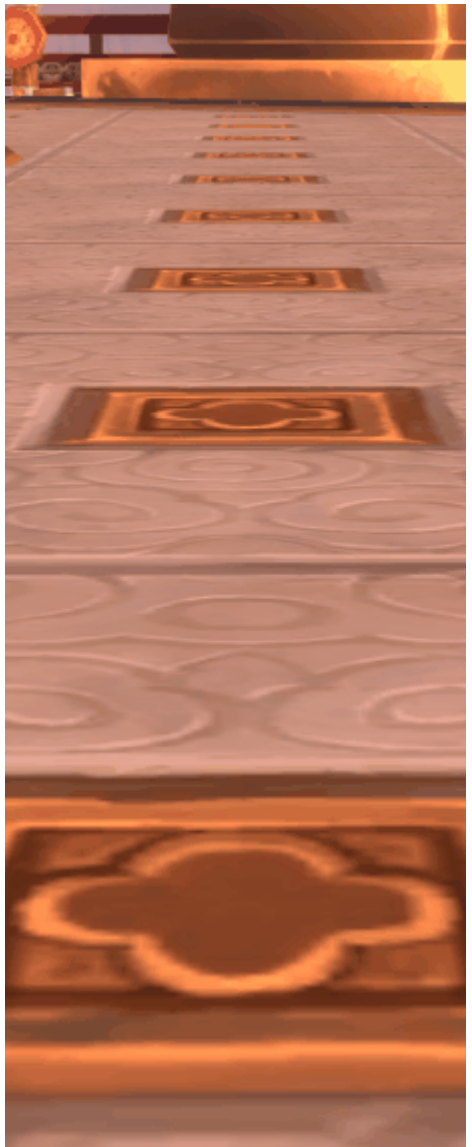
- Orthogonal
  - Perpendicular to camera
  - Close to right angles
- Problem:
  - Texture is square
  - Area to texture is not
- Huge loss of detail
  - With *linear* filtering

Solution: Anisotropic filtering

- Takes extra samples
  - Matching projected shape
  - Trapezoidal pattern
- Performance intensive
  - Several levels available
  - 1x, 2x, 4x, 8x, 16x
- Benefits are obvious

Anisotropic side-by-side

Bilinear, Trilinear, Anisotropic



Texture Best Practices

Texture size – Best practices

- Texture sizes should be powers of 2
  - 64x64, 1024x1024, 256x32, etc.
- Non-power-of-2 textures are slower to use
  - “Power-of-2” sizes allow for optimizations
  - Older hardware only supported power-of-2
- Some older hardware would pad textures
  - Up to next power of two
  - 513x513 pixels would take up 1024x1024 space

Compression – Best practices

- Textures are **uncompressed** in memory
  - 512x512 JPG
    vs.
  - 512x512 PNG
  - They take up the same amount of GPU memory!

- Create your textures in a loss-less format
  - Your files will be bigger
  - But your game will look way better

Advanced Texturing Techniques

“Multi-texturing”

- Applying more than one texture at a time

- Combine the texture colors in unique ways
  - Create different effects by adding, multiplying, etc.
  - Sample two textures in the same shader

Specular maps

- Specular lighting makes objects look shiny
  - But objects aren't always uniformly shiny
- You can control the shininess (specularity) of each pixel
  - A specular map
  - Sometimes called a Gloss Map
- A single value from 0 – 1
  - Determines how "shiny" this pixel can be
  - Apply after the specular calculation: specResult * specMapValue

Specular map example

Color Map

Notice there's no alpha
(only RGB channels used)

Specular Map

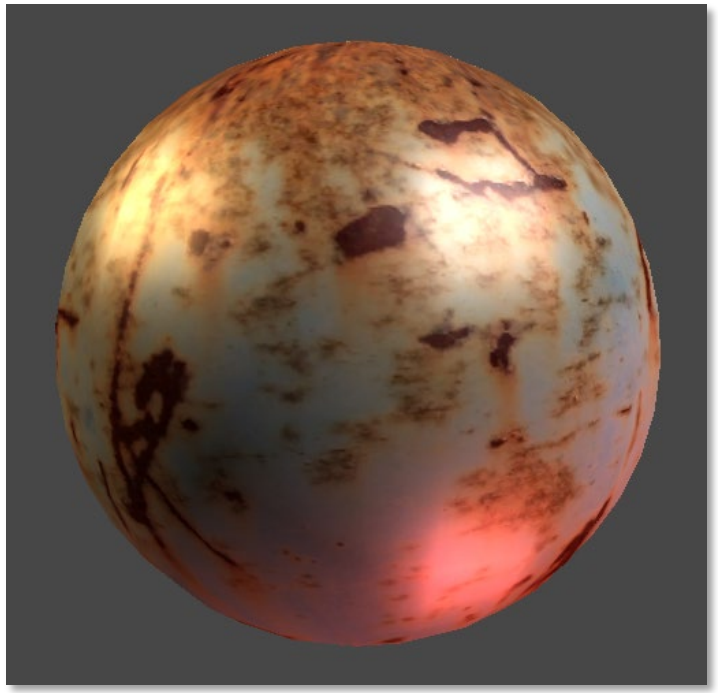
Basically a "percent" (one number)
Why not store in the alpha channel?

Specular map data

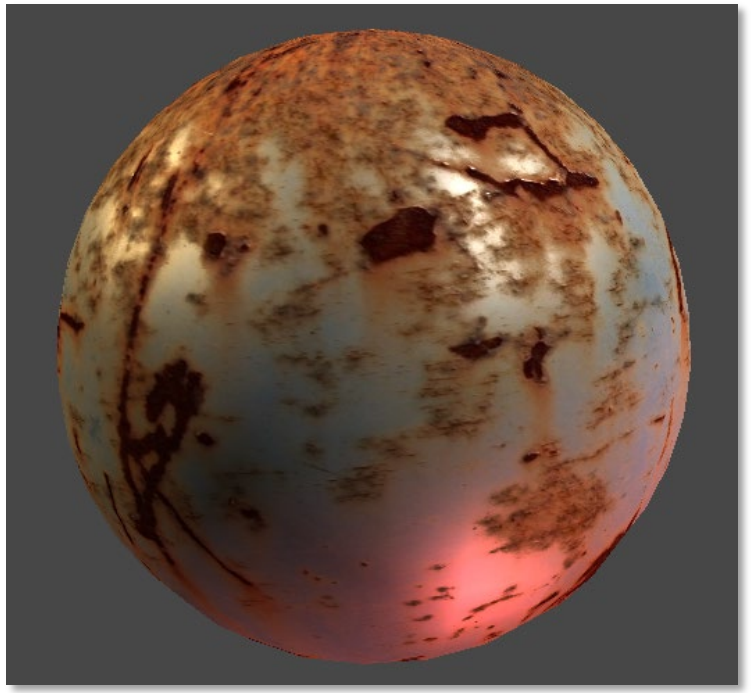
- Only need 1 channel for the spec map data
- Store in an unused channel of another texture
  - Alpha channel of base texture
  - Alpha channel of “normal map” (more later)
- Since you’re sampling that texture anyway, you get this data for “free”!

Specular map example

Lighting

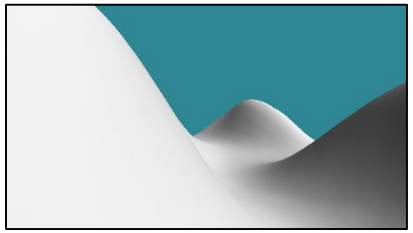


Lighting with Spec Map

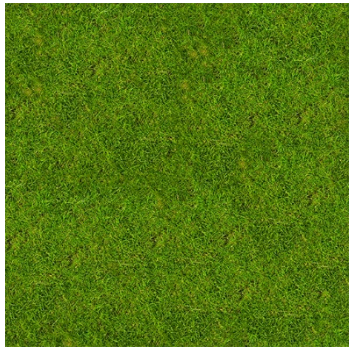


Problem – One large texture

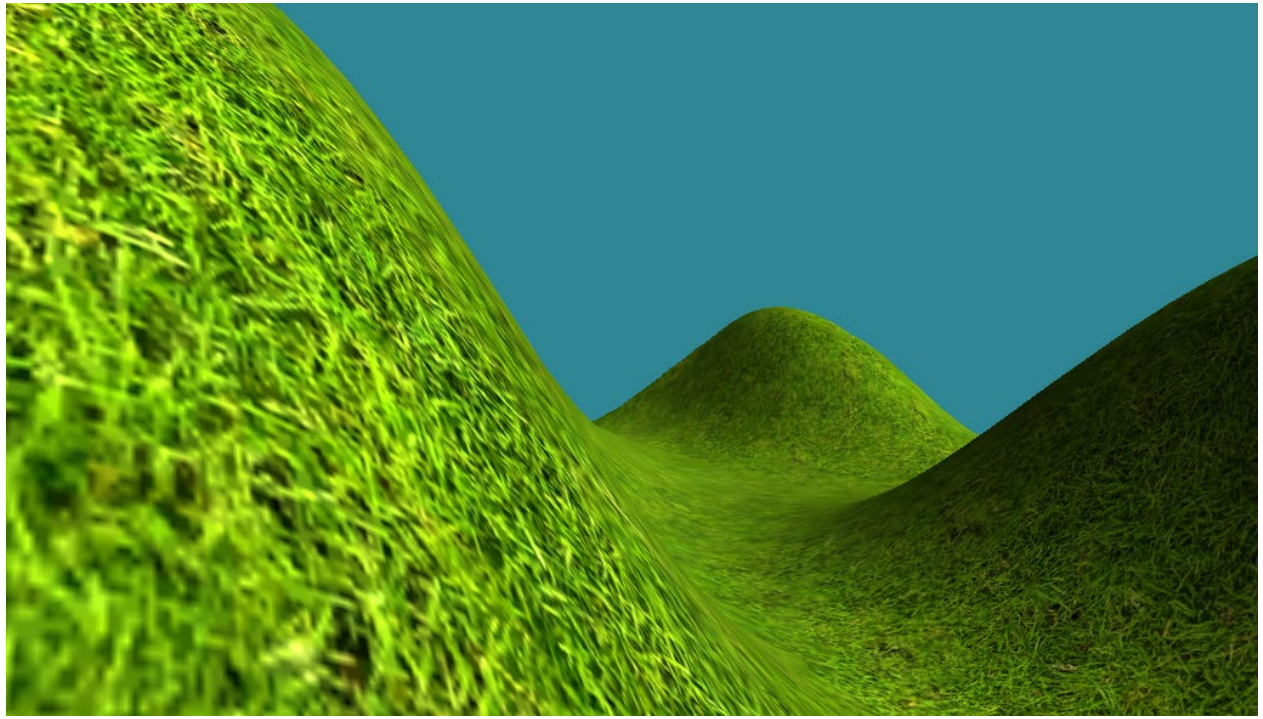
- Applying a single texture to a huge mesh
  - Looks good far away
  - But blurry/pixelated up close



+

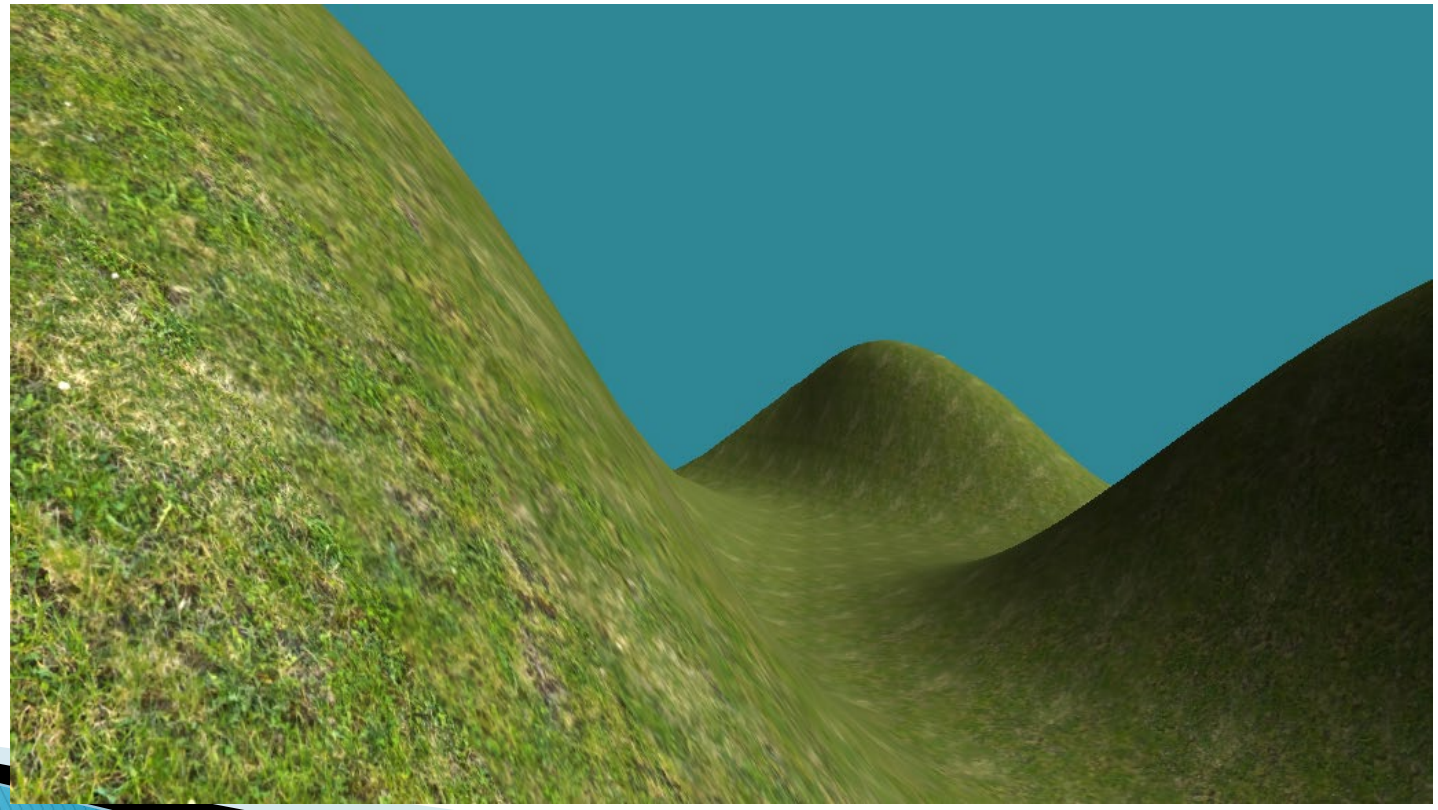


=



Problem – Tiled texture

- Up close it looks great
- Texture tiling is obvious at a distance



Solution: Detail textures

- Applying multiple textures *at different scales*
  - Each texture uses a different uv scale in the shader
- Merges:
  - High frequency detail
  - Low frequency base texture
- Sample both textures and combine
  - Simply average the colors
  - Or have one tint the other

Detail textures

- Detail color is blended with the base texture
- Looks better close and far away