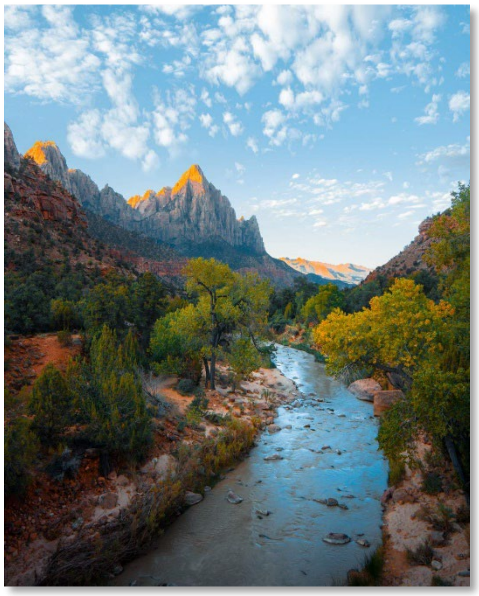
Textures

Applying images to our 3D models



Digital Images

Outside of the digital space, a picture or image is a visual representation of something. In the digital realm, images serve the same purpose: they allow us to present data visually on a display device like a monitor. Really, a digital image boils down to a two-dimensional grid of pixel colors.

> *In fact, the word pixel comes from the term "picture element", and represents the smallest single element of a digital image/picture.*

Image Storage & File Formats

While the end goal of an image is to provide a 2D grid of pixel colors, that is not necessarily how they're stored in files on a hard drive. Some image file formats, like .BMP (bitmap), are always uncompressed, meaning each and every pixel's color is fully represented in the file itself. An image that is twice as wide will result in a file that is twice as large. Reading the pixel data from an uncompressed image file is relatively straightforward: loop & read each color value, storing them in a 2D array in memory.

Some file formats, like .GIF and .PNG, support a lossless compression called Run-Length Encoding (or RLE). In RLE, if multiple contiguous pixels within a single row have the exact same color, the file simply stores the color and the count of how many pixels should receive that color, rather than repeating the color value multiple times. In images with large areas of flat color, like flat-shaded cartoons, this can greatly reduce the file size. Images representing photographs rarely have the same color more than once in a row so RLE doesn't help there.

Some other file formats, like .JPG, use a lossy compression that can greatly reduce the file size at the cost of losing some information within the file. Due to this more complex compression scheme, loading pixel data from a .JPG is much more complex than a simple loop. Ideally, we'd use some pre-existing code or library to open and read common image file formats.

> *JPG, or the full version, JPEG, is an acronym for Joint Photographic Experts Group, the group that created this type of compression for digital images in the early 90's.*

Images in RAM (CPU and GPU)

Regardless of the type of compression and file format, if you want to display an image on a digital display, that image needs to be uncompressed: you need the final value of every single pixel to actually use it for displaying or rendering. This means if you have a 512x512 pixel image, it will take up 512x512 pixels worth of memory once the image is used by an application. It doesn't matter if the image was originally stored as a .BMP, .GIF, .PNG or .JPG, once an application opens it, 512x512 pixels worth of RAM is used!

As you might infer from this, the same is true for image data we'll eventually store on the GPU: an image's file format won't impact the amount of memory it takes up while rendering. At the end of the day, images you use for your application should probably be stored in a lossless image format – .PNG is a great choice – so they look as good as possible in-game. Using a lossy format like .JPG won't save any GPU memory but *will* make your images look worse during gameplay!

Note that there are some compressed-in-memory formats available on modern GPUs, though their usage is more complex. This is outside of the scope of this reading and our course, but Aras Pranckevičius, one of Unity 3D's first graphics programmers, has a great blog post about texture compression if you want to read more about it!

Textures

Textures are GPU resources, similar to vertex and index buffers, most often used to hold pixel data representing images. They're simply *buffers of pixels*, instead of vertices or indices, in GPU memory. Textures are how we provide image data to shaders during rendering. Whereas vertex and index buffers act as the pipeline's input, texture data can be directly read from any shader stage, so long as the texture is properly bound to a corresponding register prior to rendering.

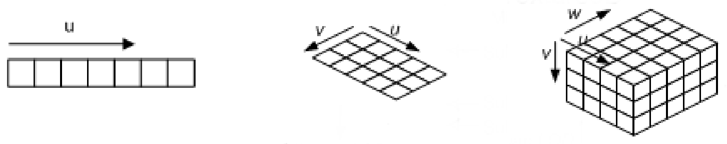
Just as a pixel is a picture element, a texel is a texture element (a pixel of a texture).

While it is quite common for the pixel data within a texture to match our standard color representation, (four 8-bit color channels for 32-bit color), there are many different channel arrangements and bit depths we can choose from. Unlike our custom vertex definition, the color formats supported by a Graphics API are not usually customizable; we choose from a predefined list of possible formats.

We'll be sticking with standard 32-bit color for now, but do know that some advanced techniques make use of formats with different color representations.

Texture Dimensions

Most textures are two-dimensional, as most images are. However, GPUs and Graphics APIs support 1D, 2D and even 3D textures. 1D Textures are effectively single rows of pixels, often used as simple gradients. 3D textures are like multiple 2D textures stacked on top of each other, often used to represent volumetric data. That being said, we'll be using 2D textures exclusively for now.

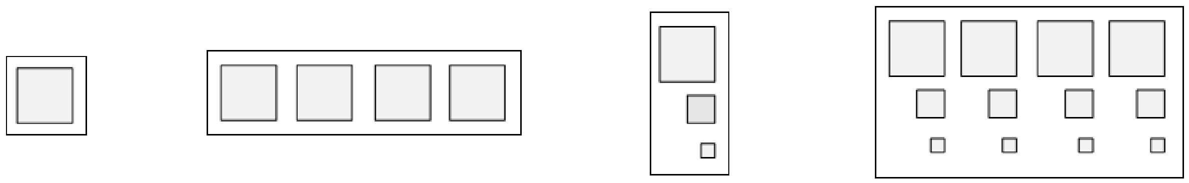


1D, 2D and 3D texture layouts.

Subresources

For any GPU resource, it is possible for the rendering pipeline to access the entire thing or just a subset of it. A subset of a resource is known as a *subresource*. For textures in particular, subresources can represent one or both of the following:

1. *Arrays of textures*, where each element is the same size but contains unique data
2. *Mipmaps,* which are smaller versions of the same texture used for smooth texture sampling



A simple texture, a texture array, a texture with mipmaps and a texture array with mipmaps

This feature isn't often used with vertex, index or constant buffers, but is quite useful for several advanced texture techniques. Texture arrays can be dynamically indexed in a shader, though that ends up being less flexible than you might imagine, as the creation of subresource texture arrays is cumbersome and requires each texture in the array to have the same dimensions. Mipmaps, however, will be discussed later in this document, as they are critically important for quality texture sampling.

One very useful feature of texture arrays is that an array of exactly six textures can be flagged as a texture cube. In a texture cube, the six subresources are regarded as the six faces of a cube. Rather than using UV coordinates to sample these textures, a direction in 3D space originating from the center of the cube is used instead. This allows us to treat a set of textures as our environment backdrop, which is very useful for rendering skyboxes or baked reflections.

Creating Textures

Creating a texture through code is fairly straightforward; we just need the color values of each pixel first. Once we have that initial data, either read from a file or procedurally created through code, texture creation in Direct3D is similar to creating any other buffer: fill out a struct describing the resource you want to make (texture dimensions, color format, etc.), provide the initial data (the pixel colors) and then ask the Graphics API to create it.

```
// Describe a 2D texture of the given width & height
D3D11_TEXTURE2D_DESC td = {};
td.Width              = width;
td.Height             = height;
td.ArraySize          = 1;                          // Not an array
td.BindFlags          = D3D11_BIND_SHADER_RESOURCE; // Can be read by shaders
td.CPUAccessFlags     = 0;                          // Not accessible by C++
td.Format             = DXGI_FORMAT_R8G8B8A8_UNORM; // 32-bit unsigned format
td.MipLevels          = 1;                          // No extra mipmaps
td.MiscFlags          = 0;
td.SampleDesc.Count   = 1;
td.SampleDesc.Quality = 0;
td.Usage              = D3D11_USAGE_DEFAULT;
```

One tricky part of procedural texture creation is ensuring your color data perfectly matches the color format of the texture. For instance, our standard 32-bit color format uses 8-bit unsigned integer values for each color channel. This means a single pixel is exactly four bytes, where each byte is an integer between 0-255. For a 100x100 image, you need an array of 40,000 unsigned bytes (100x100x4).

> *The Direct3D constant for this format is DXGI_FORMAT_R8G8B8A8_UNORM, or unsigned normalized values. "Normalized" in this case means individual channel values don't go above 1.0 when interpreted as floats, though as bytes the upper limit ends up being 255 before conversion.*

Since our shaders work with colors as vectors of 0.0 - 1.0 float values, your instinct might be to just use arrays of float values or arrays of XMFLOAT4 vectors for procedurally created texture data in C++. However, floats and integers have different binary representations; binary for the integer 500 and the float 500.0 are *not* equivalent. As such, we can't just use arrays of float values, or even the XMFLOAT4 data type, for colors when creating texture resources. Since the data is just copied bit-by-bit into the GPU resource, the resulting color values would be malformed! It needs to be converted to actual bytes first, as shown here:

```
// Each color channel is a single unsigned byte, so we can use
// "unsigned char" in C++.  Each pixel needs 4 color channels (as
// denoted by our choice of color formats), so we multiply the total
// number of array elements by 4.  Different color formats might have
// different numbers of channels, so adjust accordingly!
unsigned char* finalPixelData = new unsigned char[width * height * 4];

// The following loop assumes you already have all of your pixel colors
// for this texture in an array of XMFLOAT4's called "colors", where
// each component is in the range 0.0 - 1.0.  It then converts from that
// range to the 0-255 range and casts to a byte (unsigned char).
for (int i = 0; i < width * height * 4;)
{
	int pixelIndex = i / 4;
	finalPixelData[i++] = (unsigned char)(colors[pixelIndex].x * 255);
	finalPixelData[i++] = (unsigned char)(colors[pixelIndex].y * 255);
	finalPixelData[i++] = (unsigned char)(colors[pixelIndex].z * 255);
	finalPixelData[i++] = (unsigned char)(colors[pixelIndex].w * 255);
}
```

The last step is to ask the API to create the resource, which will look very much like your vertex and index buffer creation code. However, since this pixel data is meant to be interpreted as a two-dimensional array, we need to provide the width (in bytes) of a single row of pixels. The API calls this the "pitch" of the data, and is a required piece of the initial subresource data definition:

```
// Initial data for the texture
D3D11_SUBRESOURCE_DATA data = {};
data.pSysMem		= finalPixelData;
data.SysMemPitch	= sizeof(unsigned char) * 4 * width;

// Actually create the resource on the GPU
Microsoft::WRL::ComPtr<ID3D11Texture2D> texture;
device->CreateTexture2D(&td, &data, texture.GetAddressOf());
```

Shader Resource Views

While we obviously need a texture resource to hold the raw pixel data of a texture, Direct3D doesn't bind textures to the pipeline in the same way you might bind your vertex or index buffers. Instead, we need to create an API object in C++ called a Shader Resource View (or SRV) which describes how much of a given resource, like a texture, a shader is allowed to access. Note that this does *not* mean "how many pixels" the can shader access; it means how many subresources (array elements and/or mip levels) a shader is allowed to use. Once we have an SRV, we bind *that* to the pipeline so the shader can access the underlying texture data.

> *You can think of a Shader Resource View as a set of rules for viewing a texture. "The shader can only see the first three array elements" or "The shader can only access mip levels 2 through 6."*

Why not just allow the shader to see the whole resource? Most of the time you absolutely will! But in the rare circumstance that these sorts of limits are useful, we can do that too. However, since creating a Shader Resource View that allows full resource access is so common, there's an easy way to do so: skip the SRV description entirely and just pass in a null pointer to the CreateShaderResourceView() function.

```
// Create a shader resource view for our texture so we can bind it
Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> srv;
device->CreateShaderResourceView(
  texture.Get(),       // The texture resource itself
  0,                   // Skip the description! Pass in 0 for a default SRV
  srv.GetAddressOf()); // The resulting SRV
```

Since we bind using the SRV and not the texture directly, we don't technically need to keep track of the texture resource pointer anymore. Most Direct3D engines will simply store the SRV, as that internally has a reference to the corresponding texture.

Binding Textures to the Pipeline

Once you have a Shader Resource View, you'll need to bind it to the pipeline if you want to access the corresponding texture in a shader. The pixel shader is where you'll do almost all of your texture work, but technically each shader stage can access textures. As such, each shader stage has its own function for binding textures, such as VSSetShaderResources() and PSSetShaderResources(). This is similar to binding constant buffers.

When using these functions, provide the starting register, the number of resources to bind and an array of them (or the address to a single SRV pointer). Each resource in the array will be bound to contiguous registers, starting at the index you've specified.

```
context->PSSetShaderResources(0, 1, srv.GetAddressOf());
```

To ease texture binding, SimpleShader has simplified functions for setting SRVs. These functions allow you to reference texture variable names, as defined in your shaders, directly:

```
ps->SetShaderResourceView("surfaceColorTexture", srv);
```

Loading Images

More often than not, however, we'll be getting our texture data from image files. While Graphics APIs can store data on the GPU, they generally do *not* contain functionality for loading and uncompressing image data from external files. We'll either need to do that ourselves or use an existing library to handle loading images. Luckily, Microsoft makes just such a library, called the **DirectX Tool Kit**.

DirectX Tool Kit

The DirectX Tool Kit, DXTK for short, is a collection of common helper classes to aid in creating Direct3D applications. This includes classes for creating geometric primitives, handling gamepad input and loading texture data from image files. The library is open source and available on GitHub, though we won't need to copy the code into our projects manually.

Get DXTK with the NuGet Package Manager

The DirectX Tool Kit is available through NuGet, an online code sharing platform that we can access directly from Visual Studio. Code libraries like DXTK can be added to a project with just a few clicks.

To add the DirectX Tool Kit to your project, follow the "NuGet Package Manager" steps outlined on the DirectX Tool Kit wiki. You will not need to perform any of the other steps on that page. However, ensure you *do not* add the directxtk12 version by accident, as that is not compatible with Direct3D 11!

Loading Images with DXTK

Once you've added the tool kit to your project, you can begin using it in your own code. Note that all functions within DXTK are in the DirectX namespace (which you might already be using due to DirectX Math also being in that namespace).

To load images, you'll include the WIC Texture Loader header. WIC stands for Windows Imaging Component, which is a built-in Windows library for opening and decompressing common image formats.

```
#include "WICTextureLoader.h"
```

Once you have that header included, loading an image is a single function: CreateWICTextureFromFile(). This function has two overloads, one that takes just the Direct3D Device object, and one that also takes the Direct3D Context object. If you use the overload that accepts the Context object, mipmaps for the texture will automatically be created (more on these soon), which is almost always what you want.

```
// Load the specified texture, generate mipmaps and get references
// to both the texture itself and an SRV for that texture
CreateWICTextureFromFile(
	device.Get(),
	context.Get(),
	L"path/to/image.png",
	textureToCreate.GetAddressOf(),
	srvToCreate.GetAddressOf());
```

While the function can both create the texture resource on the GPU and create an SRV for you, it's rare to actually need the reference to the texture itself. Since the SRV is what we ultimately need to bind the texture, it's quite common to pass in a null pointer for the texture parameter.

```
// Load the specified texture, generate mipmaps but only return the SRV
// Note: The texture resource is still created!  We just don't need a
// reference directly to it since our materials will just use SRVs directly
CreateWICTextureFromFile(
	device.Get(),
	context.Get(),
	L"path/to/image.png",
	0,
	srvToCreate.GetAddressOf());
```

Textures & Shaders

Now that we can create textures on the GPU, either manually or by loading them from a file, and bind them to the pipeline, we need to actually read from them in our shaders. In HLSL, there are predefined data types for each type of texture – Texture1D, Texture2D, Texture3D and TextureCube – as well as types for arrays of 1D and 2D textures: Texture1DArray & Texture2DArray.

Whether or not a texture is considered an array depends on how many array subresources it was created with. A texture with an array size greater than 1 is considered a texture array.

Since textures are resources accessible by a shader, each texture is bound to a register, just like constant buffers. Recall that constant buffers used the identifier "b" followed by an index for their registers. Texture use "t" for their registers. Texture variable definitions look like this in HLSL:

```
Texture2D SurfaceTexture : register(t0); // Textures use "t" registers
Texture2D SpecularMap    : register(t1);
```

These are the registers to which you bind your SRVs if directly using the D3D bind functions. If using SimpleShader, you'd use these variable names instead of the register indices.

Direct3D 11 Resource Limits

The following table describes various limits for resources in Direct3D 11.

Resource	Limit
Max Texture1D & Texture2D dimensions	16,384 pixels per dimension
Max Texture3D dimensions	2048 pixels per dimension
Max elements for 1D & 2D texture arrays	2048 elements
Total textures per shader	128
Total sampler states per shader	16
Total constant buffers per shader	14

Texture Sampling

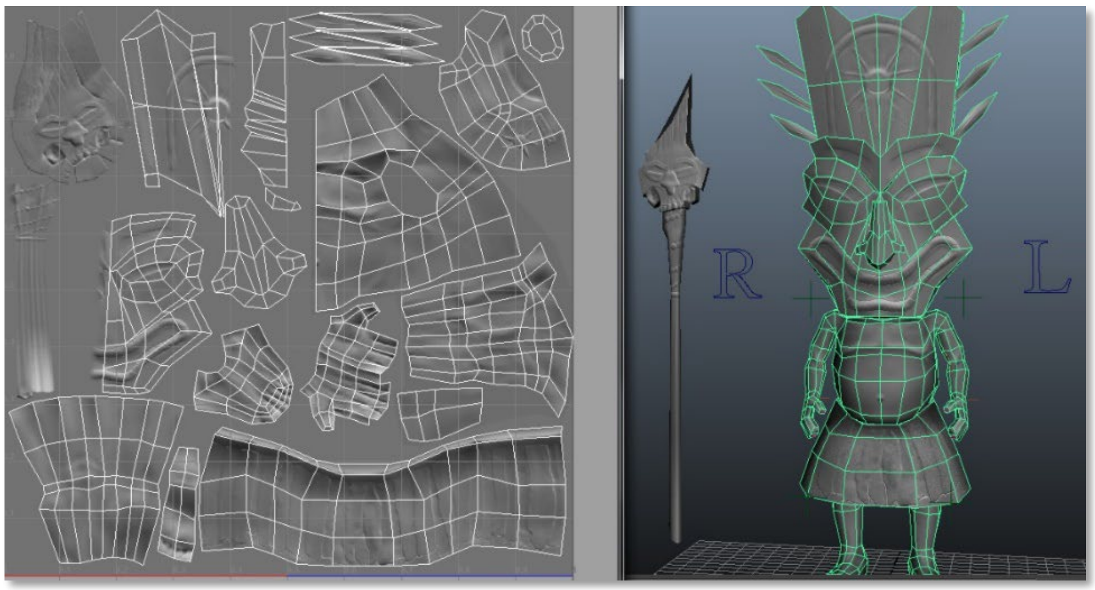
The act of retrieving a color from a texture is known as **Texture Sampling**, and requires a few other pieces of information, namely, the location to sample from (the UV coordinates) and any sampling options. These options include *address mode* – what happens when our UV coordinates go outside the 0 to 1 range? – and *filter* – what color do we return when a UV coordinate is between two pixels?

UV Coordinates

UV coordinates, also known as texture coordinates, describe resolution-agnostic coordinates within a texture. In other words, they describe a location as a percentage across the image (a 0.0 – 1.0 value) rather than as specific pixel coordinates. These coordinates are either generated programmatically or created by the artist of a particular mesh.

UV coordinates are part of our vertex data. Since we'll primarily be doing our texture sampling in pixel shaders, UV coordinates of each vertex are passed through the vertex shader so they can be interpolated by the rasterizer. Each rasterized pixel will then have unique coordinates to use for texture sampling.

See Reading 7 – 3D Models & Materials for more information on UV coordinates.

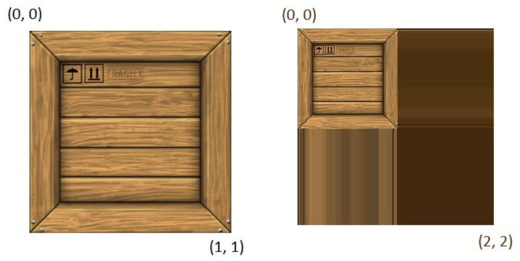


Left: A texture with distinct UV-mapped areas.
Right: 3D mesh with the texture applied.

Sampling Option – Address Mode

UV coordinates represent an address within a texture as, usually, values between 0.0 and 1.0 on the U (horizontal) and V (vertical) axes. Technically, our UV values are floats, so they could go above 1 or below 0. What happens when they do? That's the first of our texture sampling options.

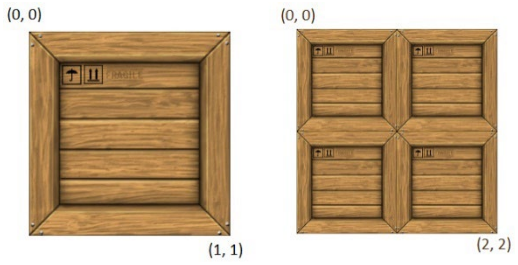
Clamp



The default address mode when sampling textures is called **Clamp**. As the name suggests, our UV values are clamped to the 0 – 1 range. Anything below 0 becomes 0; anything above 1 becomes 1. This causes any coordinates outside 0 – 1 to use the colors on the very edge of the texture.

This is probably not the mode you'd want for basic mesh rendering, as the streaks of colors shown on the right don't look great. However, this address mode is useful for certain advanced techniques.

Wrap

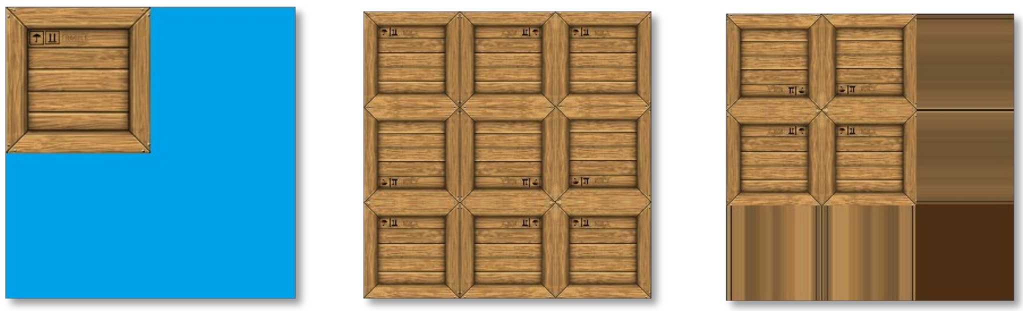


Much more commonly used for standard mesh rendering, the **Wrap** address mode wraps any values outside the 0 – 1 range back into that range. Thus: 1.3 becomes 0.3, -0.2 becomes 0.8, etc. This effectively repeats a texture.

Other Address Modes

While Wrap and Clamp are the most common, they aren't the only address modes:

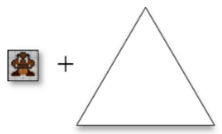
> **Border** address mode always returns the same user-defined color outside the 0-1 range.
>
> **Mirror** flips UV coordinates back and forth, so 0-1 is usual, 1-2 is mirrored, 2-3 is usual, etc.
>
> **Mirror Once**, as the name suggests, mirrors the UVs once across 0 and then clamps after that. This is the same as taking the absolute value of the UV coordinates and then clamping 0-1.



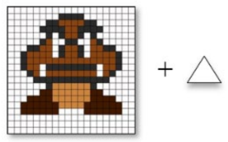
Address mode examples: Border (UVs 0 – 2), Mirror (UVs 0 – 3) and Mirror Once (UVs -1 – 2)

Sampling Option – Filter

When sampling a texture, it is rare for the pixels of our rasterized triangles to perfectly match the pixel grid of our texture. The triangle may have many more pixels than the texture, far fewer pixels than the texture or the triangle may be seen at a harsh angle causing both of these to occur at different points.



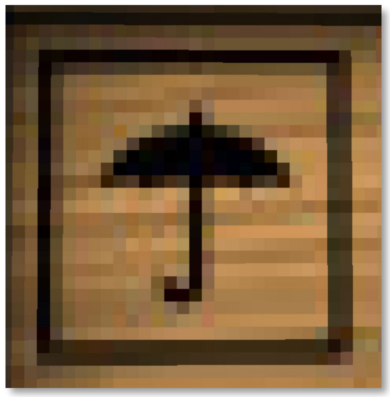
The situation where the triangle has been rasterized to more pixels than the texture results in *texture magnification*. In this case, multiple pixels (on the screen) map to the same texel (from the texture). In other words, the texture needs to be "blown up" to cover all the pixels.



Conversely, a triangle could be rasterized to very few pixels while the texture contains many more pixels, resulting in *texture minification*. In this case, a single pixel (on the screen) would map to multiple texels (from the texture). In other words, the texture needs to be "shrunk down" to fit the triangle.

The process of mapping the texture to a set of pixels of a different size is called **texture filtering**. As you might expect, there are several different approaches to filtering, each with their own pros and cons. Different types of filtering could be used for minification and magnification, though it is common to use just one.

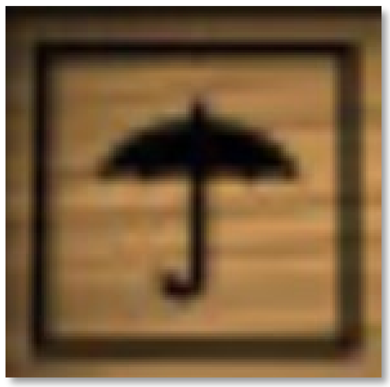
Point Filtering



Point filtering, also known as nearest-point or nearest-neighbor filtering, is the most basic and least-costly type of filtering. With point filtering, the nearest texel to the UV coordinate is used for the color of the pixel. During texture minification, this means some texels will be skipped. During magnification, on the other hand, some texels will be repeated. As there is no interpolation of colors, this results in large blocks of the same color.

The game Minecraft uses point filtering for its textures to give a retro, 8-bit look to its 3D models.

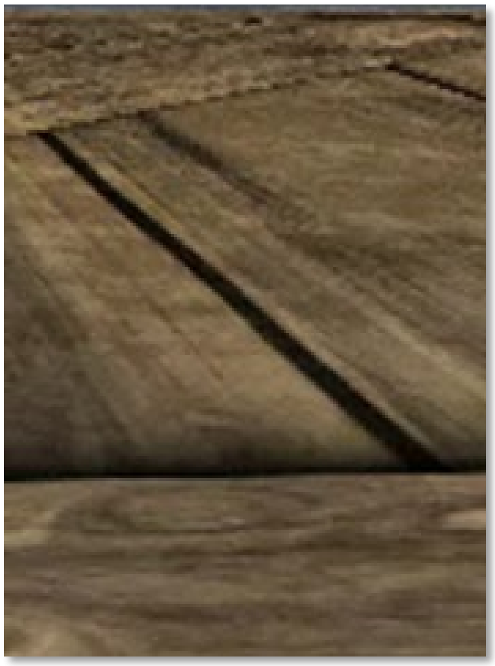
Linear Filtering



With *linear filtering*, instead of just choosing an existing texel, a linear interpolation of the four closest texels is used. During magnification, this results in a smooth (and potentially blurry) result. During minification, some image details might still be skipped, as only 4 texels are interpolated rather than *all* of the texels that the pixel might cover.

When linear filtering is used for both minification and magnification, it is referred to as *bilinear filtering*.

Extreme Minification

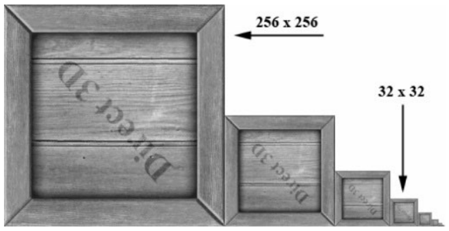


One potential issue with these filtering modes is extreme minification. During these instances, a large number of texels are within the area of a single screen pixel. Since only 4 of those texels are used for linear filtering (or a single texel for point filtering), most of the texels will be skipped. If those skipped texels contain some very small but obvious detail, like a set of very dark pixels amongst otherwise bright pixels, that detail may completely disappear.

As the camera and/or object moves or rotates, those dark pixels may sometimes be visible. Over the course of several frames, this produces a very noticeable (and visually distracting) noisy effect called *texture aliasing* or *shimmer*. This effect is far more noticeable in motion, but as a quick example, see the image to the right. Texture detail near the top, where there is more minification, appears noisier and more aliased (the stair-step pattern) than the bottom.

Mipmapping

How can we prevent shimmer during extreme minification? The issue stems from texture details being skipped when greatly "shrinking" the texture. We could either take far more samples to accurately approximate that detail (very costly at run time) or we could store pre-shrunk versions of the texture. The "pre-shrunk" method, known as mipmapping, is a common way of handling this issue.



Mipmaps are smaller, pre-filtered versions of a texture; they themselves are just *more textures*. A series of mipmaps, where each is half the dimensions of the previous one (all the way down to 1x1), is called a *mipmap chain*. Mipmap chains are stored as subresources of the original texture, allowing the GPU to easily access them during sampling.

Each pixel color of one mipmap level is the average of the corresponding 2x2 block of pixels from the previous (larger) level. In this way, the colors in each mipmap are influenced by all of the pixels of the previous level. Thus, the mipmaps are pre-filtered, and every level has influence in some way from all of the levels above it.

The term "mip" comes from the Latin "Multum In Parvo", meaning "much in a small space".

Creating Mipmaps

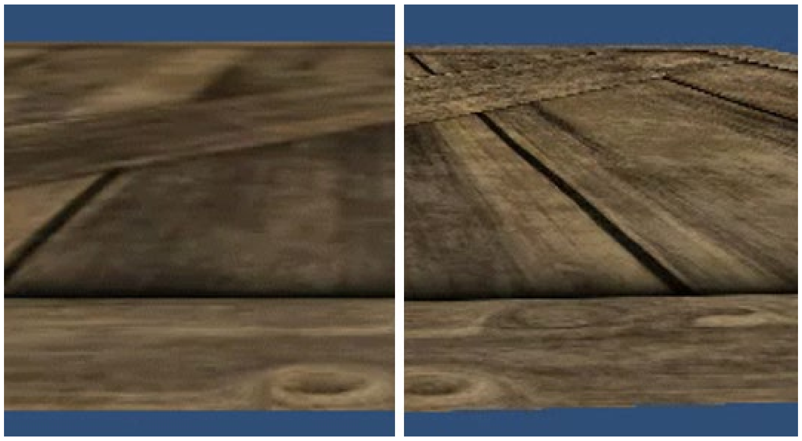
Mipmaps can be created by hand by an artist and loaded as subresources of a texture. However, it is usually good enough to allow the GPU to generate them automatically from the base texture. When using the DirectX Tool Kit to load an image, ensure you call the overload of CreateWICTextureFromFile() that includes the Direct3D Context object; this version automatically generates mipmaps for you!

Using Mipmaps

Mipmaps are such a common solution to the problem of extreme minification that all modern GPUs support them. If mipmaps are present in a texture (and we've enabled their use), we can manually choose which one we want to sample from as part of the sampling process in our shaders. Even better, we can let the GPU automatically choose the best mip level when sampling. Instead of always using our very large texture, it will choose one of the smaller versions that are part of the same resource as necessary. This makes mipmaps incredibly easy to use as we don't really need to do anything special in our shaders; the GPU does all the heavy lifting.

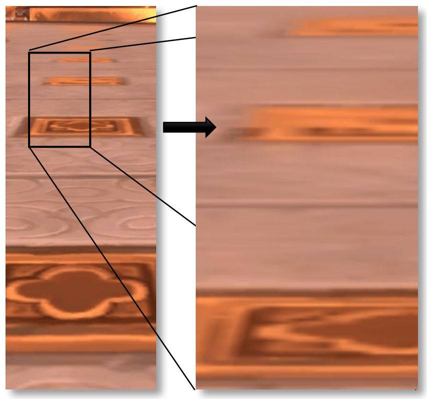
> *Note that automatic mipmap selection only works within pixel shaders. Sampling from other shaders requires us to specify the exact mip level to use (even if there is only one). What's special about the pixel shader? Behind the scenes, all pixel shaders are run in small groups of 2x2 screen pixels, allowing the GPU to calculate the rate of change (or derivative) of data across this small area of the screen. This rate of change is then used by texture sampling functions within a pixel shader to automatically choose the correct mipmap. When the rate of change is low, there is less minification, so a larger mip level is fine. As the rate of change increases, minification does as well and smaller mip levels are more appropriate.*

For an example of results using mipmaps, see the images below. The left image is using mipmaps; the right image is not. The right image is much noisier, especially at a distance. In motion, this becomes incredibly distracting as various texture details shimmer in and out. In the left image, the noise is smoothed out (though it is blurry; more on that soon).



Filtering Between Mipmaps

Various pixels of a single triangle will sample from different mip levels, resulting in obvious changes to the quality of the texture across the triangle. Again, this is more obvious in motion, but see the image to the far right for a quick example. The detail on the top half of the golden symbol is noticeably blurrier than the bottom half.

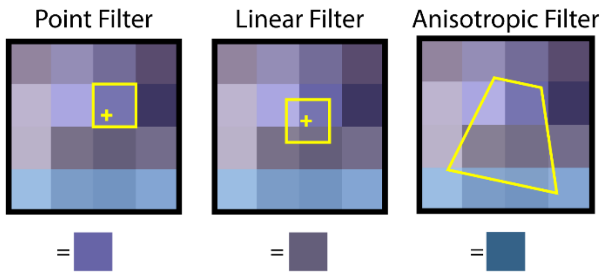


This is due to *point filtering* between the mip levels; the nearest mip level is chosen for any given texture sample. While this is the least costly use of mipmaps, the results aren't ideal. Alternatively, mip levels can be *linearly filtered* (interpolated), resulting in a smooth blend between each mipmap. This is slightly slower than point mip filtering, as two mip levels need to be fully sampled and then combined, but the results are almost always worth the extra cost. When linear filtering is used for minification, magnification and between mip levels, it is known as *trilinear filtering*.

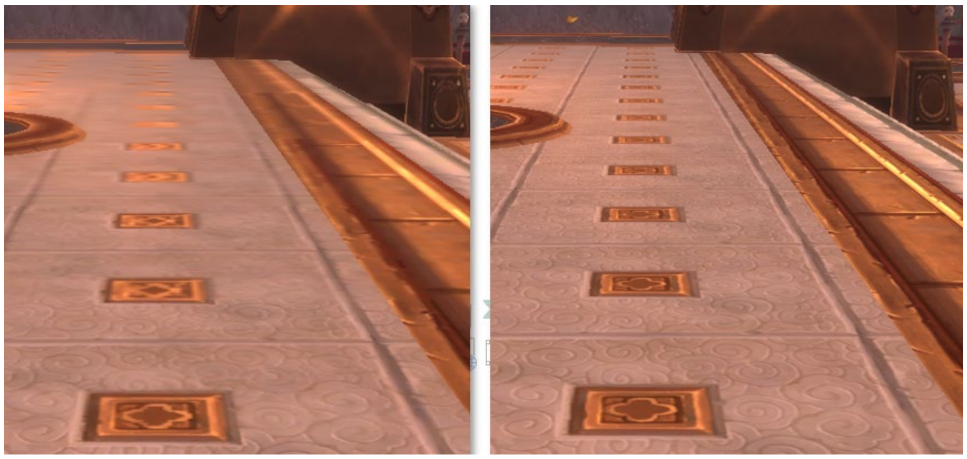
However, even with trilinear filtering, one major problem remains: surfaces orthgonal to the camera, like the floor in a virtual environment, end up looking blurry in the distance. We've traded one problem for another: mipmaps reduced visual noise but ended up blurring things!

Anisotropic Filtering

Mipmaps are great, but if we want orthogonal surfaces to look crisp in the distance, we need a different type of filtering altogether. **Anisotropic filtering** to the rescue!



Linear filtering works best for surfaces that we're looking at head-on. Surfaces viewed at a sharp angle, like a floor, are thinner at the top than the bottom once projected onto the screen (effectively a trapezoid). Anisotropic filtering takes extra samples in a trapezoidal pattern matching the projected shape of the surface, increasing detail. The number of extra samples depends on the level of anisotropy we set, and can be 1x, 2x, 4x, 8x or 16x. The higher the multiplier, the crisper the final image and the longer the process takes. This is why games often expose the level of anisotropy as an option.



Left: trilinear filtering. Right: anisotropic filtering. Quite a difference!

Sampler States

Address mode? Filter? Level of anisotropy? How do we denote all of this for the GPU? In Direct3D, we create objects called **sampler states** that contain sampling options. If our entire rendering process only ever uses one set of options, we'll only need to create a single sampler state for our entire application. If we'll need multiple sets of options, we'd create multiple sampler states.

We can then bind one or more of these objects to the pipeline to provide these settings to our shader at runtime. Most, if not all, of our shaders will probably just use a single sampler state at a time, though it is possible to provide multiple to a single shader if we intend to sample various textures differently.

Creating Sampler States

As with any Direct3D object, we need to fill out a description and have the API create the sampler state object. There are several advanced options that can have values of 0 in the description, so the following code only focuses on the required address mode, filter and mipmap-related settings.

```
// Create a sampler state representing our texture sampling options
Microsoft::WRL::ComPtr<ID3D11SamplerState> sampler;

D3D11_SAMPLER_DESC sampDesc = {};
sampDesc.AddressU      = D3D11_TEXTURE_ADDRESS_WRAP; // Each dimension can
sampDesc.AddressV      = D3D11_TEXTURE_ADDRESS_WRAP; // have a different mode
sampDesc.AddressW      = D3D11_TEXTURE_ADDRESS_WRAP; // but that is uncommon
sampDesc.Filter        = D3D11_FILTER_ANISOTROPIC;
sampDesc.MaxAnisotropy = 16;
sampDesc.MaxLOD        = D3D11_FLOAT32_MAX;          // Maximum mip level

device->CreateSamplerState(&sampDesc, sampler.GetAddressOf());
```

The above sampler is set up for anisotropic filtering. The constants for other common filters are:

- Bilinear: D3D11_FILTER_MIN_MAG_LINEAR_MIP_POINT
- Trilinear: D3D11_FILTER_MIN_MAG_MIP_LINEAR

Sampler States in HLSL

Similar to textures and constant buffers, sampler states in HLSL are bound to registers using the identifier "s" followed by an index. Sampler state variable definitions look like this in HLSL:

```
SamplerState BasicSampler : register(s0); // Samplers use "s" registers
```

A sampler state is necessary every time we sample from a texture in HLSL, as the intrinsic Sample() function takes a sampler state as a parameter. However, if we never bind a sampler state to this register, Direct3D will fallback to a default sampler state that uses bilinear filtering and a clamp address mode. In this case you'll also get warnings about a null sampler in Visual Studio's output window as your program runs, though the default sampler will be used.

Binding Sampler States

Just as each shader stage has its own functions for binding textures, each also has a function for binding sampler states: VSSetSamplers(), PSSetSamplers(), etc. When using these functions, provide the starting register, the number of samplers to bind and an array of them (or the address to a single sampler state). Each sampler in the array will be bound to contiguous registers, starting at the index you've specified.

```
context->PSSetSamplers(0, 1, sampler.GetAddressOf());
```

As with SRVs, SimpleShader has simplified functions you can use to bind samplers. These functions allow you to reference sampler state variable names, as defined in your shaders, directly:

```
ps->SetSamplerState("BasicSampler", sampler);
```

Sampling in HLSL

We have a texture. We have a sampler. We have UV coordinates. Now what? Sampling a texture in HLSL, once we have these three requirements, is quite simple. Every Texture2D variable has a Sample() function that takes two parameters: a sampler and a float2 for UV coordinates.

```
float4 color = Texture.Sample(Sampler, UV);
```

This function always returns a float4, which is the sampled color from the texture. Each component of that float4 has been converted to the 0.0 – 1.0 float range to make it easy to use as a color. If you only need the RGB components, you can simply swizzle them out of the float4 using .rgb:

```
// Sample the texture at the specified UV using the given options
float3 surfaceColor = SurfaceTexture.Sample(BasicSampler, input.uv).rgb;
```

That's it! While this may seem like a simple line of code, think about all of the *stuff* that needs to happen to retrieve this color value: UV adjustments, filtering, mipmap choosing, etc. This means that, even though it seems like a simple function from the outside, it's actually one of the most expensive things we can do in a shader.

As such, you'll want to ensure you don't perform this exact same texture sample more than once in any given shader; if you need the result in multiple places, call Sample() once and store the result in a variable. If you need to sample two or more different textures, that's fine, but again save those results in variables rather than duplicating the sampling more than necessary!