DirectX 11
First 3D objects

Cartesian coordinate system

In the real world, objects exist in 3D space.

This means that to place an object in a particular position in the world, we would need to use a **coordinate system** and define three coordinates that correspond to the position.

In computer graphics, 3D spaces are most commonly in **Cartesian** coordinate system.

In this coordinate system, **three axes**, X, Y, and Z, **perpendicular to each other**, dictate the coordinate that each point in the space has.

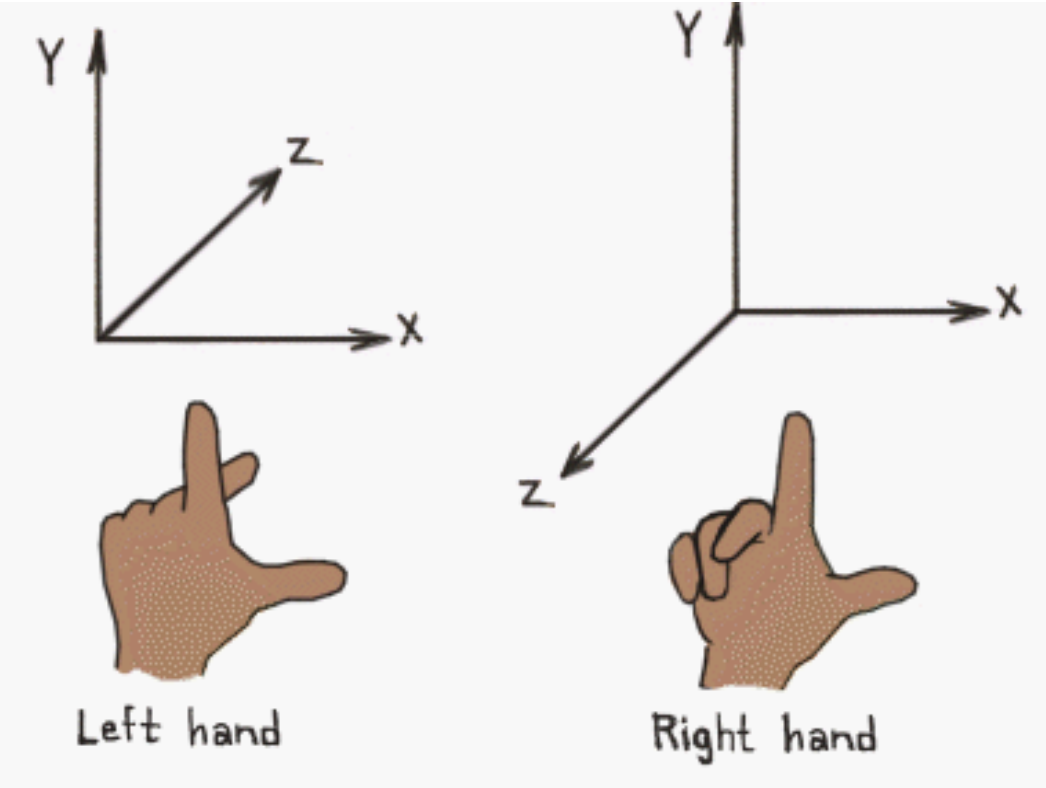
Left & Right handed systems

This coordinate system is further divided into left-handed and right-handed systems.

In a **left-handed** system, when X axis points to the right and Y axis points to up, **Z axis points forward**.

In a **right-handed** system, with the same X and Y axes, **Z axis points backward**.

Left & Right handed systems



3D spaces

In 3D, a space is typically defined by an origin and three unique axes from the origin: X, Y and Z.

There are several spaces commonly used in computer graphics:

- Object space,
- World space,
- View space,
- Projection space, and
- Screen space.

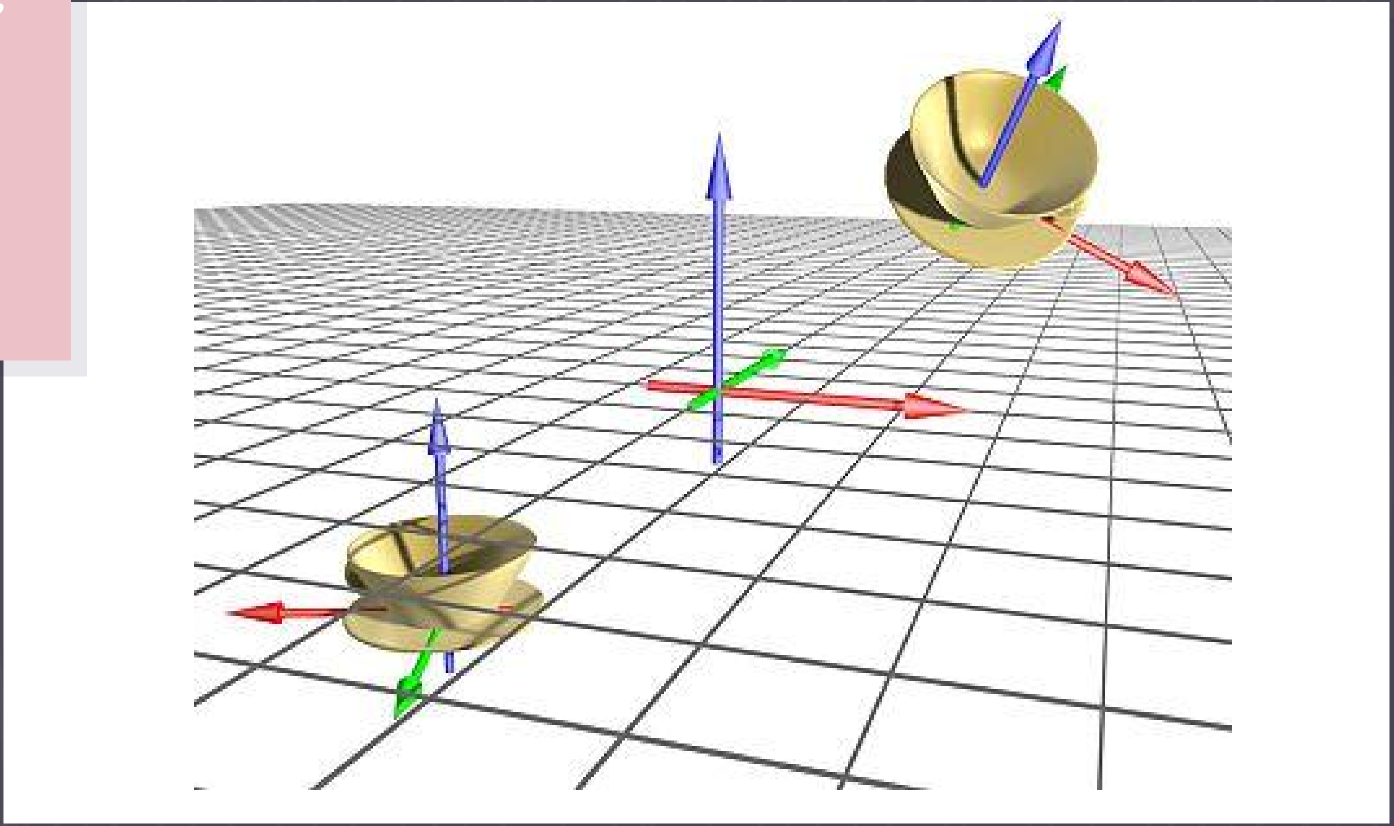
Object space

Object space, also called model space, refers to the space used by artists when they create the 3D models.

Usually, artists create models that are **centered around the origin** so that it is easier to perform transformations such as rotations to the models. That models are stored on disk are also in object space.

Vertices in the **vertex buffer** will usually be in object space. This also means that the **vertex shader** receives input vertex data in **object space**.

Object space

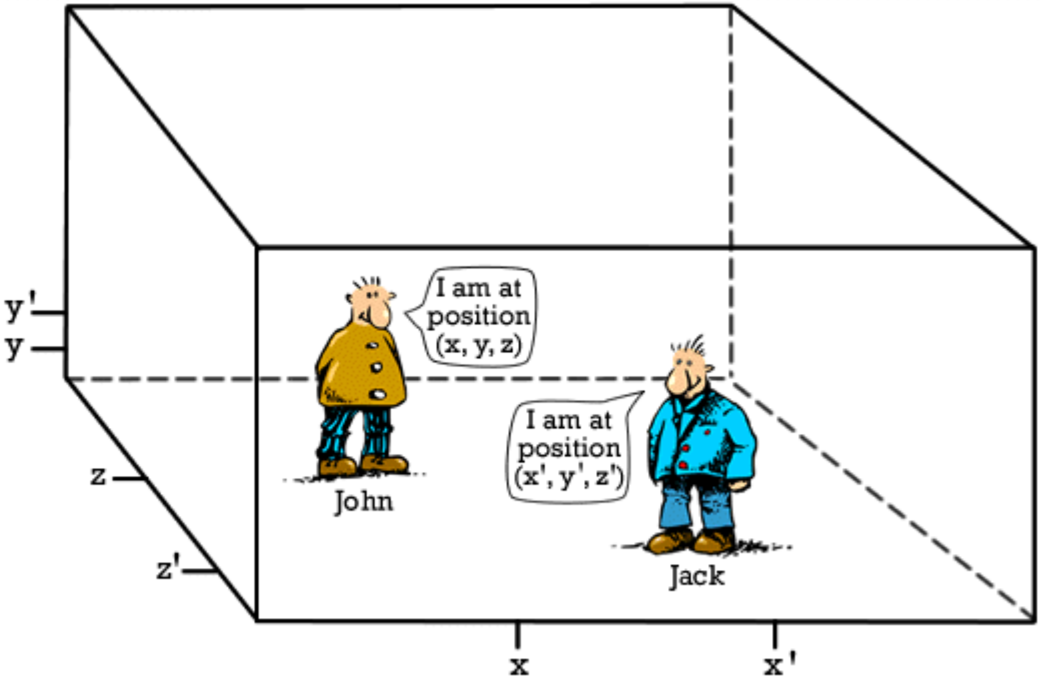


World space

World space is a space **shared by every object** in the scene.

It is used to define **spatial relationship** between objects that we wish to render.

World space



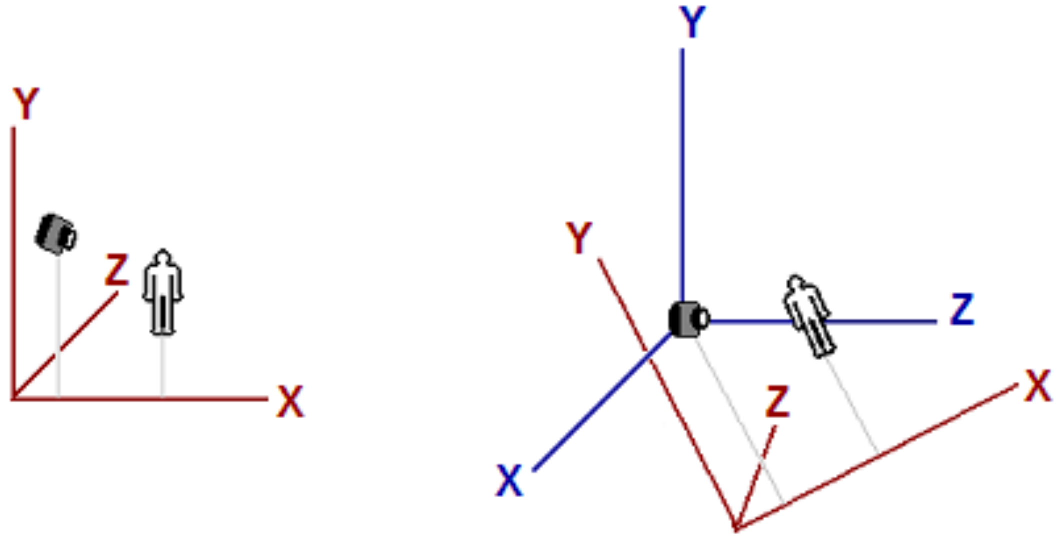
Space is not a box

View space

View space, sometimes called camera space, is similar to world space in that it is typically used for the entire scene.

However, in view space, the **origin is at the viewer** or camera. The **view direction** (where the viewer is looking) defines the **positive Z axis**. An "**up**" direction defined by the application becomes the **positive Y axis**.

View space

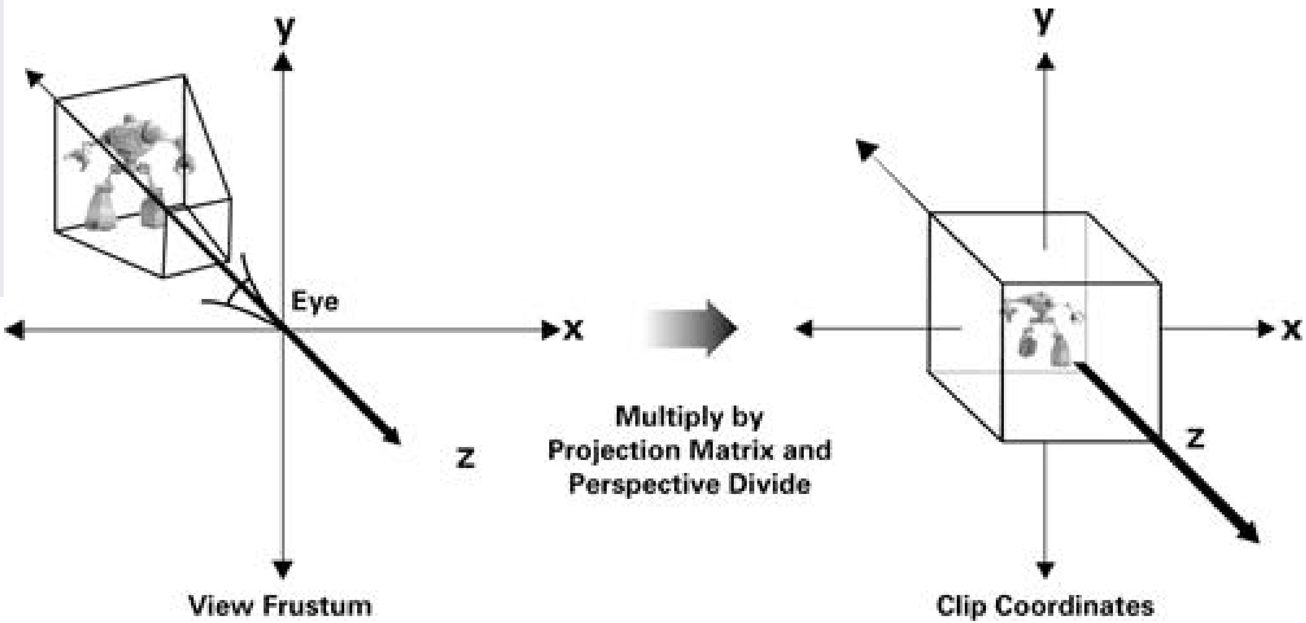


Projection space

Projection space refers to the space after applying projection transformation from view space.

In this space, visible content has X and Y coordinates ranging from -1 to 1, and Z coordinate ranging from 0 to 1.

Projection space



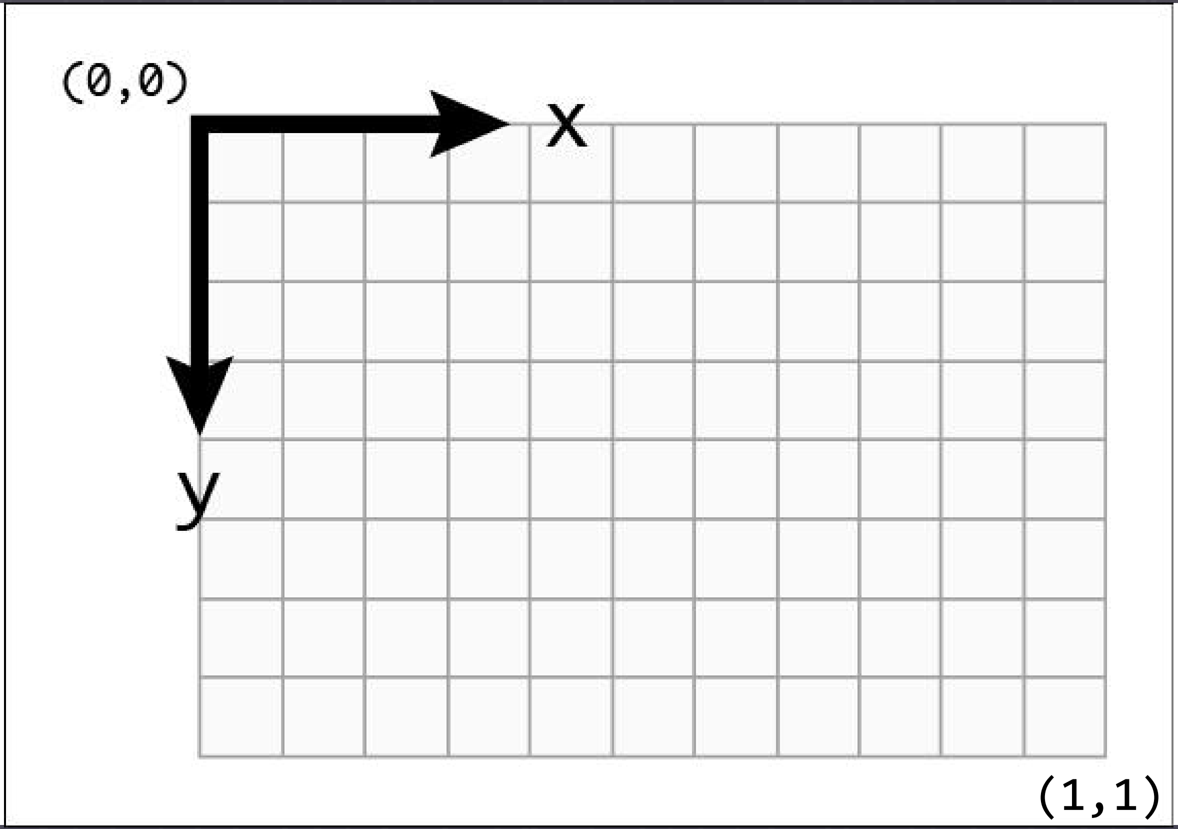
Screen space

Screen space is often used to refer to locations in the frame buffer. Because frame buffer is usually a 2D texture, screen space is a 2D space.

The **top-left corner is the origin** with coordinates (0, 0). The positive X goes to right and positive Y goes down.

For a buffer that is w pixels wide and h pixels high, the most lower-right pixel has the coordinates (w - 1, h - 1).

Screen space



Space transformations

Transformation is most commonly used to convert vertices from one space to another.

In 3D computer graphics, there are logically three such transformations in the pipeline:

- World,
- View, and
- Projection transformation.

World transformation

World transformation, as the name suggests, converts vertices from object space to world space.

It usually consists of one or more **scaling, rotation, and translation**, based on the size, orientation, and position we would like to give to the object.

Every **object** in the scene has its **own** world transformation matrix. This is because each object has its own size, orientation, and position.

View transformation

After vertices are converted to world space, **view transformation** converts those vertices from world space to view space.

View space is what the world appears from the viewer's (or camera's) perspective.

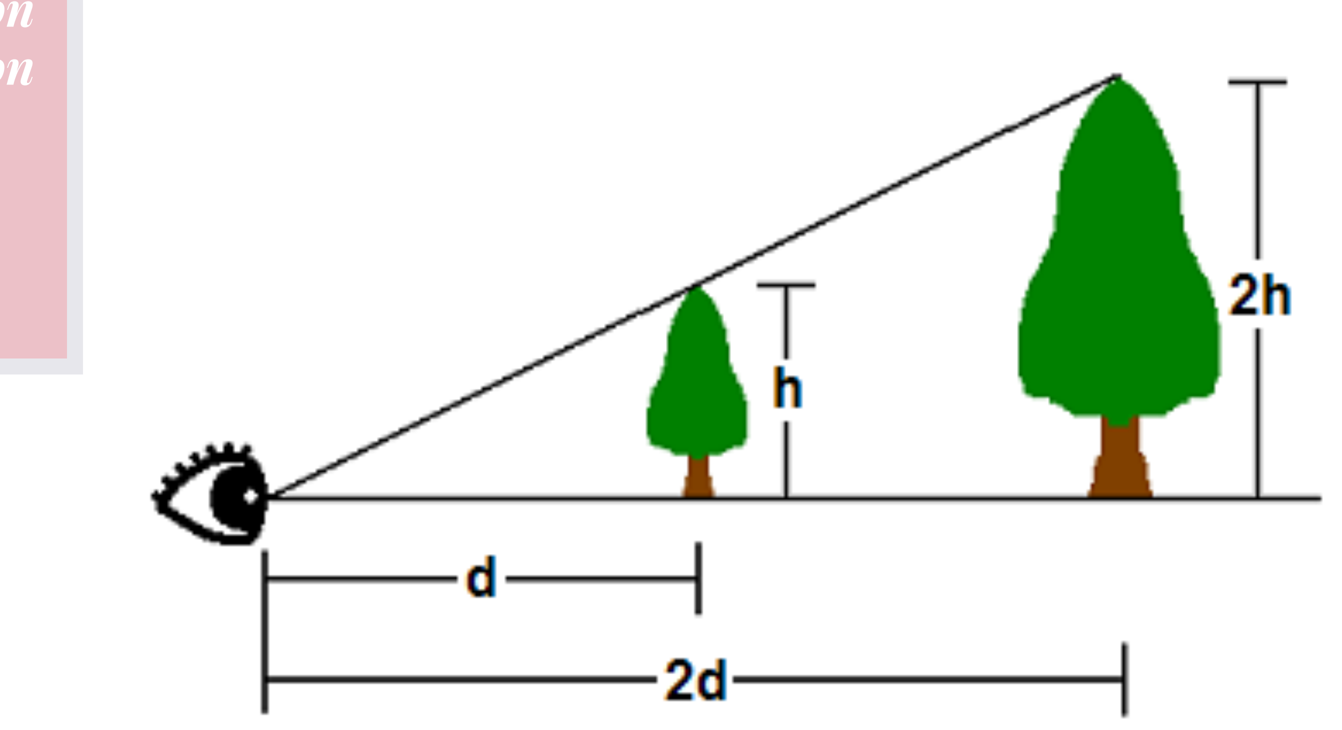
In view space, the viewer is located at origin looking out along the positive Z axis.

Projection transformation

Projection transformation converts vertices from 3D spaces such as world and view spaces to projection space.

In projection space, X and Y coordinates of a vertex are obtained from the X/Z and Y/Z ratios of this vertex in 3D space.

Projection transformation



Field of view

One of the parameters that defines a 3D space is called the **field-of-view** (FOV).

FOV denotes **which objects are visible** from a particular position, while looking in a particular direction.

Humans have a FOV that is **forward-looking** (we can't see what is behind us), and we can't see objects that are too close or too far away.

In computer graphics, the FOV is contained in a **view frustum**.

View frustum & perspective projection

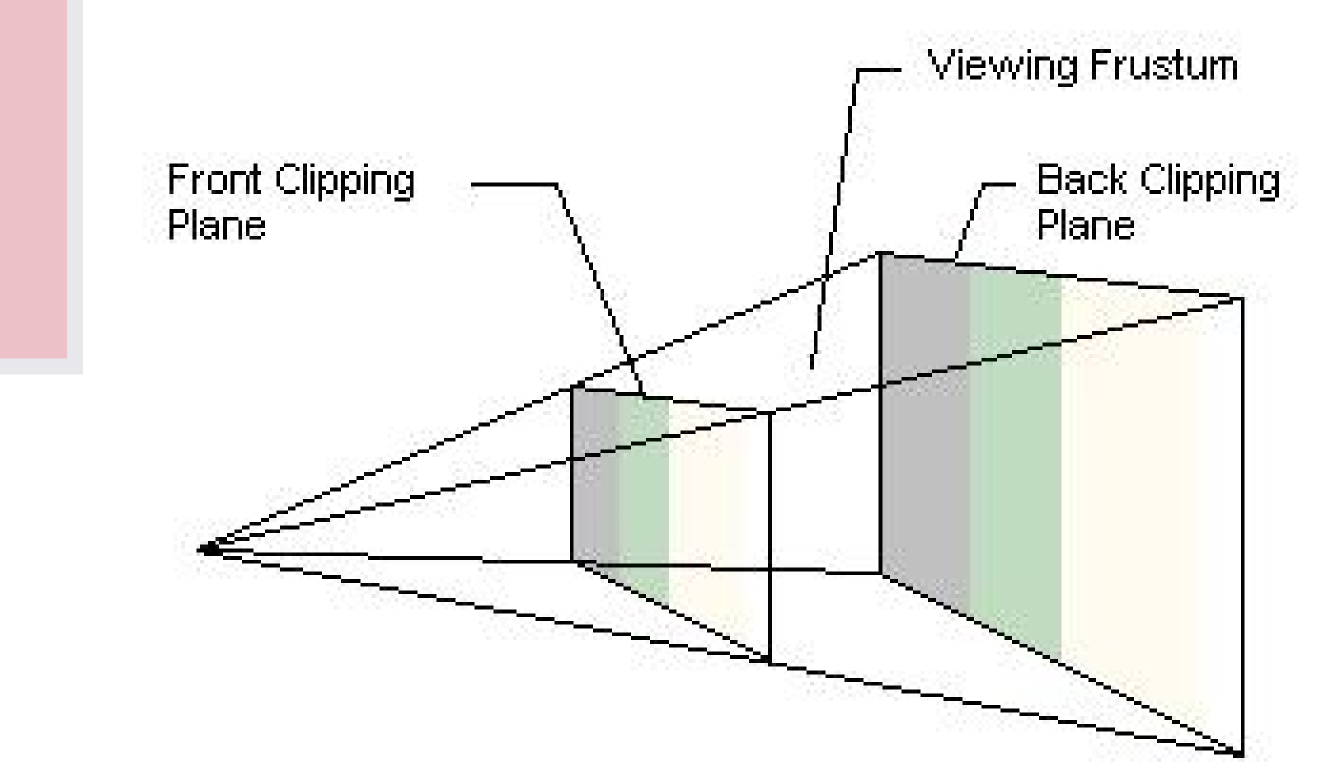
A **view frustum** is a 3D volume that defines how models are projected from camera space to projection space.

Objects must be positioned **within** the 3D volume to be **visible**.

The most common type of projection is called a **perspective projection**, which makes objects **near** the camera appear **bigger** than objects in the distance.

For a perspective projection, the view frustum can be visualized as a **clipped pyramid** whose top and bottom are defined by the near and far clipping planes

View frustum & perspective projection



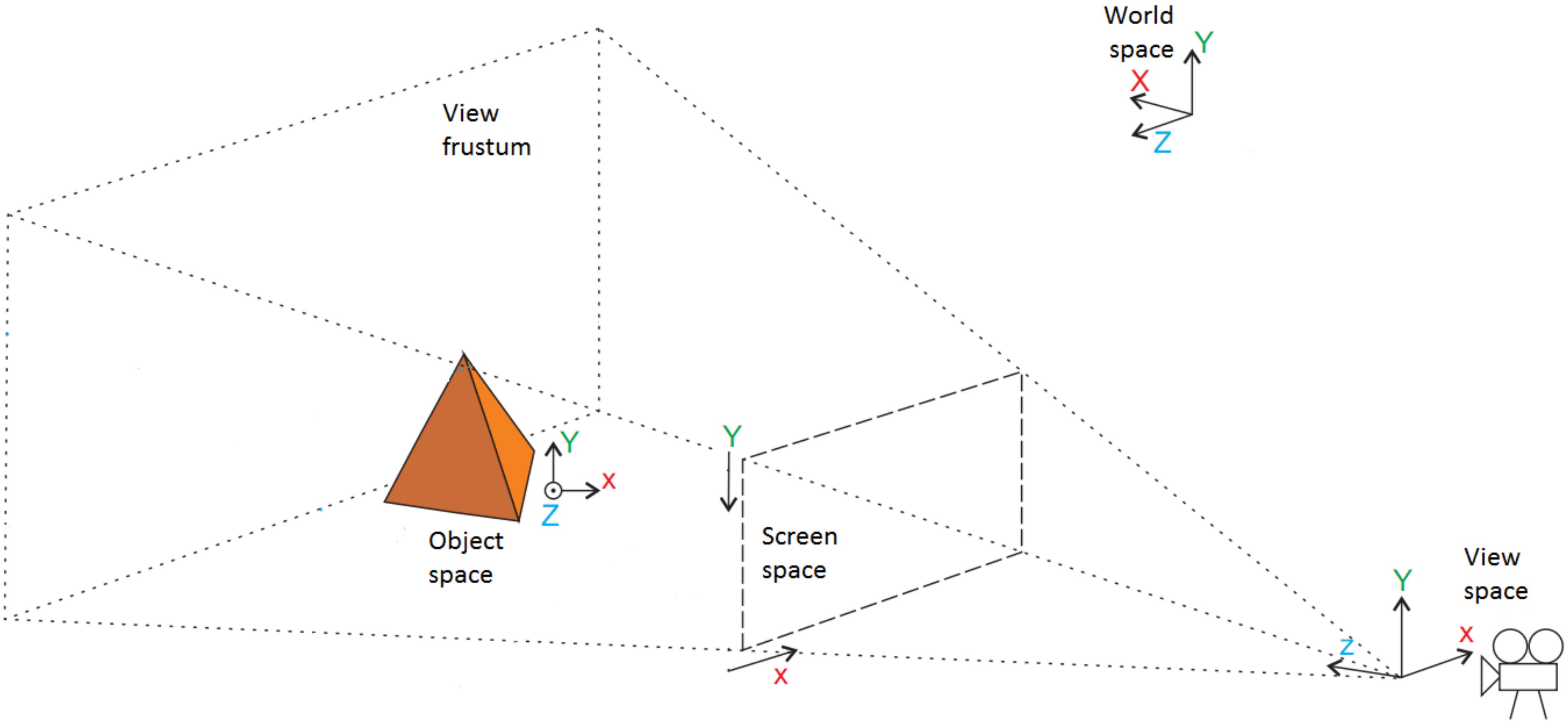
Clipping

Clipping is the GPU process **filters** out objects that are **outside** the **view frustum** so that it does not have to spend time rendering something that will not be displayed.

For avoid complex comparisons, GPU generally performs **projection transformation first**, and then **clips** against the view **frustum** volume.

The effect of projection transformation on the view frustum is that the pyramid shaped view frustum becomes a **box** in projection space.

3D spaces


World
space
Y
X
Z
View
frustum
Y
X
Z
Object
space
Y
Screen
space
X
View
space
Y
X
Z

Triangle faces & Vertex normals

Each **face** in a mesh has a perpendicular unit **normal** vector.

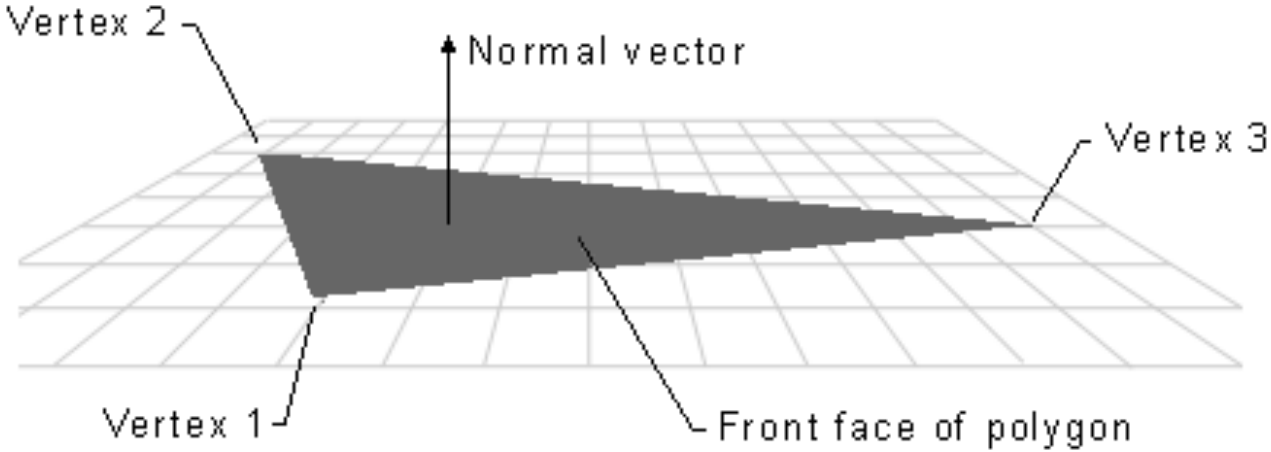
The vector's **direction** is determined by the order in which the vertices are defined and by whether the **coordinate system** is right- or left-handed.

The face normal **points away** from the front side of the face.

By default in Direct3D, **only** the **front** of a face is **visible**.

A front face is one in which vertices are defined in **clockwise** order on a **left-handed** system and **counter-clockwise** on a **right-handed** system.

Triangle faces & Vertex normals



DirectX 11
Basic lighting

Lambertian lighting

Lambertian lighting has **uniform intensity** irrespective of the distance away from the light.

When the light hits the surface, the amount of light reflected is calculated by the **angle of incidence** the light has on the surface.

When a light is shined **directly** on a surface, it is shown to reflect **all the light back**, with maximum intensity.

However, as the **angle** of the light is **increased**, the intensity of the **light will fade away**.

Lambertian lighting

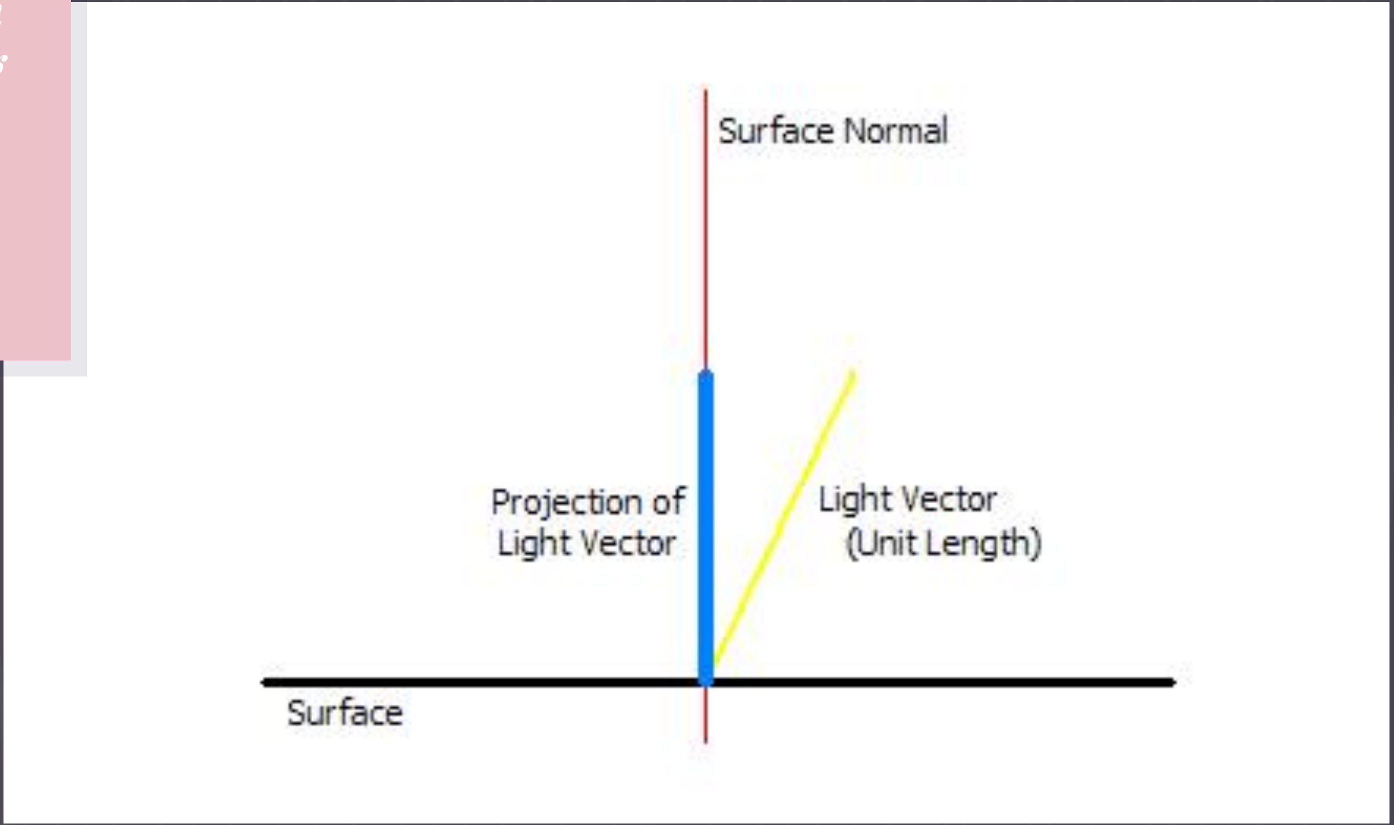
To calculate the **intensity** that a light has on a surface, the **angle** between the light **direction** and the **normal** of the surface has to be calculated.

The **normal** for a surface is defined as a vector that is **perpendicular** to the surface.

The calculation of the angle can be done with a simple **dot product**, which will return the projection of the light direction vector onto the normal.

The **wider** the angle, the **smaller** the projection will be. Thus, this gives us the correct function to modulate the diffused light with.

Lambertian lighting



Directional lighting

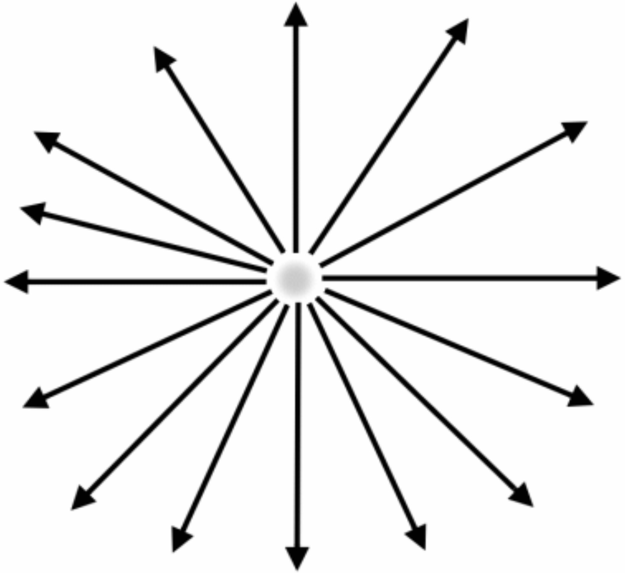
The **vector** which describes the **light** source determines the **direction** of the light.

Since it's an approximation, no matter where an object is, the direction in which the **light shines** towards it is the **same**.

An example of this light source is the **sun**. The sun is always seen to be shining in the same direction for all objects in a scene. In addition, the intensity of the light on individual objects is not taken into consideration.

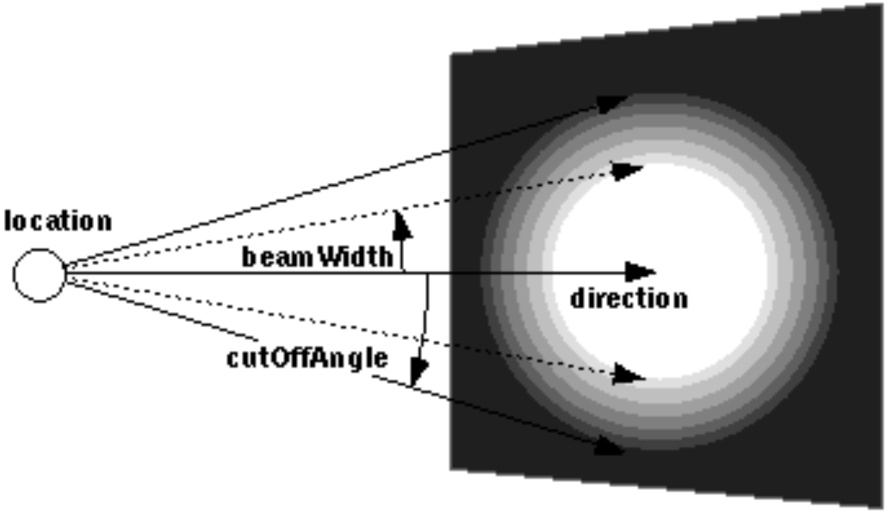
Point light

Other types of light include **point lights**, which radiate uniform light from their centers.



Spot light

...and **spot lights**, which are directional but not uniform across all objects.



DirectX 11 Texturing

Texture mapping

Texture mapping refers to the projection of a 2D image into 3D geometry.

We can think of it as **wrapping** a present, by placing decorative paper over an otherwise bland box.

To do this, we have to specify how the **points** on the **surface** of the geometry **correspond** with the 2D image.

Texture mapping

We must **align** the coordinates of the model with the texture.

For complex models, it is difficult to determine the coordinates for the textures by hand.

3D modeling packages generally will export models with corresponding texture coordinates.

Texture coordinates are **defined at the vertices**, and are then **interpolated** for individual pixels on a surface.

Texturing

We can use the Direct3D functions to load many different image file formats:

- ▷ Windows Bitmap (BMP)
- ▷ Joint Photographic Expert Group (JPG)
- ▷ Portable Network Graphics (PNG)
- ▷ Tagged Image Format (TIFF)
- ▷ Graphics Interchange Format (GIF)
- ▷ DirectDraw Surface (DDS)
- ▷ Windows Media Player (WMP)

Texture Interfaces

Texture interfaces are used to manage image data of a certain type. Within Direct3D there are three main types of texture interfaces:

ID3D11Texture1D — Handles a 1D or image strip type of texture.

ID3D11Texture2D — 2D image data. This is the most common type of texture resource.

ID3D11Texture3D — Image data used to represent volume textures (3D textures)

Texels

Each **pixel in a texture** is known as a texel.

A texel in a **color map** is a color value, usually between the values of 0 and 255 in common image formats.

A **32-bit image** is made up of four 8-bit values, with one for the red, one for the green, one for the blue, and one for the alpha.

RGB image, where each component is stored in a single byte, is 24 bits in size.

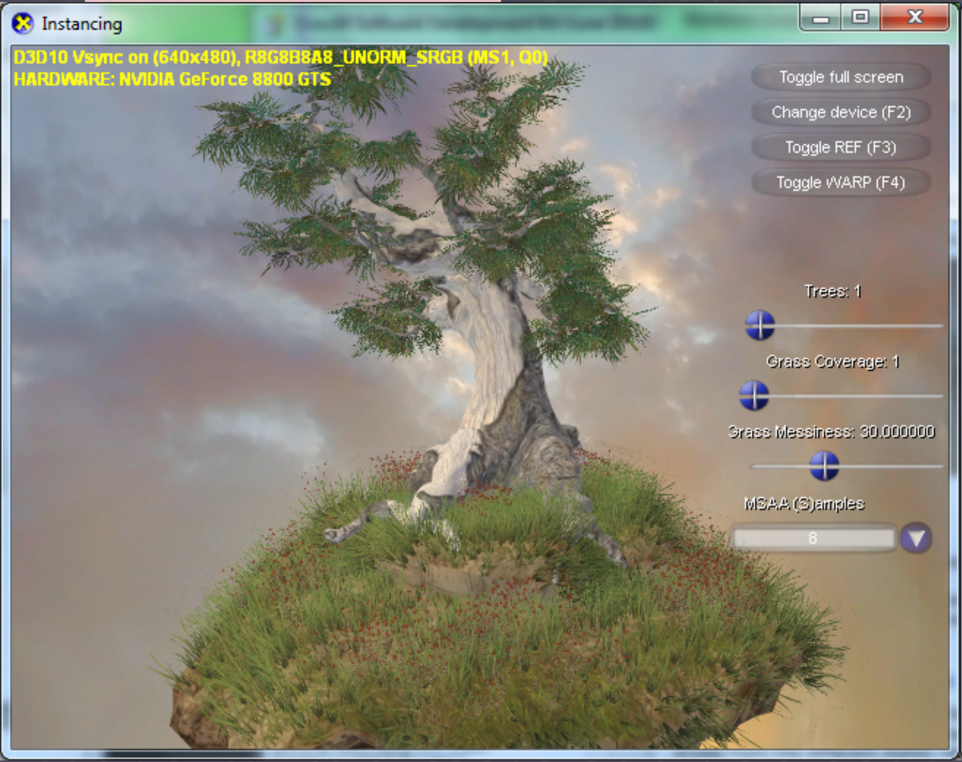
MIP maps

MIP levels are **decreasingly lower** resolution versions of the same texture.

MIP levels allow the system to **swap** in the proper texture resolution based on a surface’s **distance**.

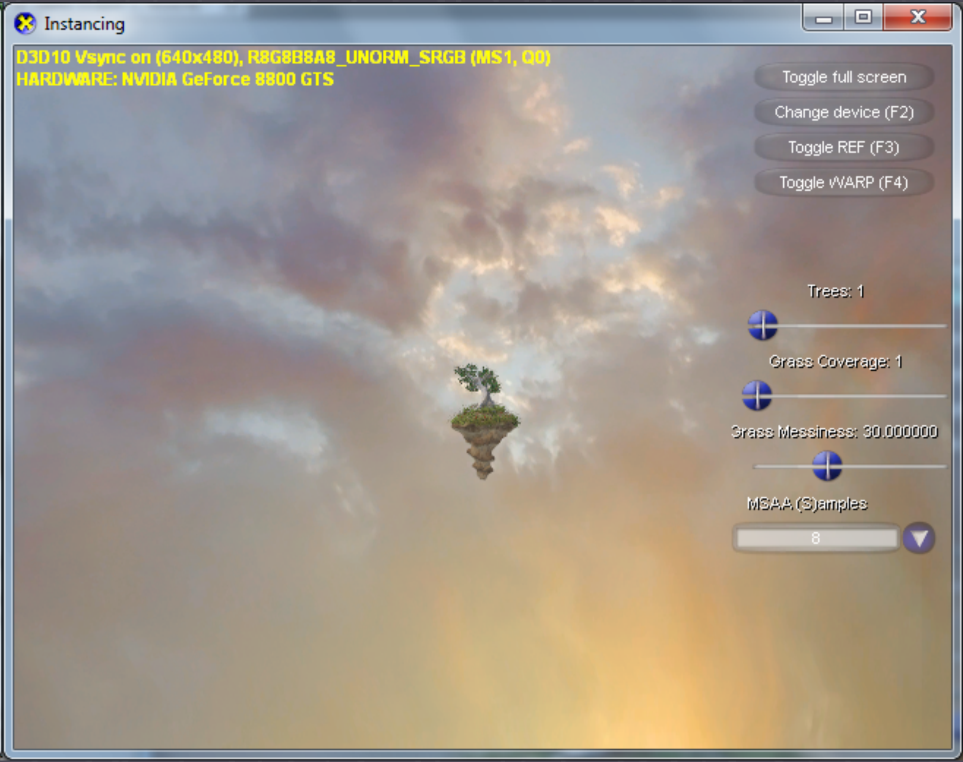
Objects further **away** need a **lower** texture applied to them because they are not close enough to see all the detail anyway.

Mipmapping



D3D10 Vsync on (640x480), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA GeForce 8800 GTS

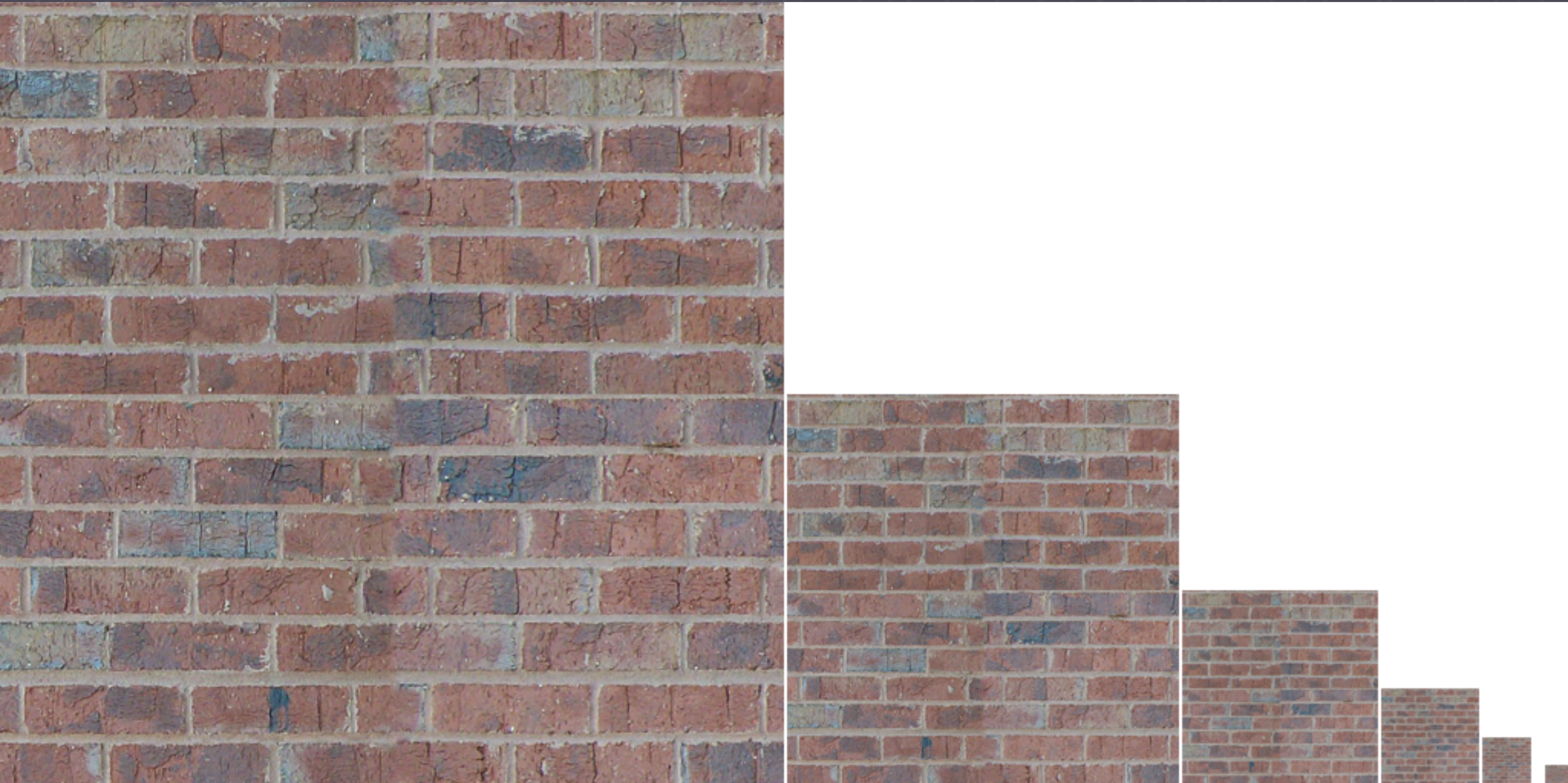
Trees: 1
Grass Coverage: 1
Grass Messiness: 30.000000
MSAA (S)amples: 8



D3D10 Vsync on (640x480), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA GeForce 8800 GTS

Trees: 1
Grass Coverage: 1
Grass Messiness: 30.000000
MSAA (S)amples: 8

Mipmap levels



MIN and MAG

MIN. Minification occurs as the textured surface moves away from the viewer.

As the **surface moves away**, more **texels** from the texture are combined to color a **single** screen **pixel**.

This is because multiple textures occupy the same pixel for surfaces as they move away from the view.

MIN and MAG

MAG. Magnification of the texture occurs when the textured **surface gets closer** to the camera, causing more **pixels** to render the **same texels**.

If each texel is being applied to a single pixel, then MIN and MAG do not occur, but in 3D games the reality is that these are pretty much always present for all textured surfaces.

Texture details

Texture details is sometimes needed and it’s available using the ID3D11Texture2D::**GetDesc** function. This function fills in D3D11_TEXTURE2D_DESC structure with all the details.

```
typedef struct D3D11_TEXTURE2D_DESC {
      UINT Width;
      UINT Height;
      UINT MipLevels;
      UINT ArraySize;
      DXGI_FORMAT Format;
      DXGI_SAMPLE_DESC SampleDesc;
      D3D11_USAGE Usage;
      UINT BindFlags;
      UINT CPUAccessFlags;
      UINT MiscFlags;
} D3D11_TEXTURE2D_DESC;
```

Texturing

Textures, like other data, will usually be loaded at runtime.

```
HRESULT D3DX11CreateTextureFromFile(
      ID3D11Device* pDevice,
      LPCTSTR pSrcFile,
      D3DX11_IMAGE_LOAD_INFO* pLoadInfo,
      ID3DX11ThreadPump* pPump,
      ID3D11Resource** ppTexture,
      HRESULT* pHResult
);
```

Shader resource

When we load the texture into memory we must create a shader resource view in order to **access that data via a shader**, and that is what we will be binding to the input assembler.

Shader resource views have **other uses**, such as providing general purpose data to DirectCompute for parallel computing.

`D3DX11CreateShaderResourceViewFromFile`

Shader resource

This function is useful when we want to both load a texture and create a new shader resource view conveniently at once.

```
HRESULT D3DX11CreateShaderResourceViewFromFile(
    ID3D11Device* pDevice,
    LPCTSTR pSrcFile,
    D3DX11_IMAGE_LOAD_INFO* pLoadInfo,
    ID3DX11ThreadPump* pPump,
    ID3D11ShaderResourceView** ppShaderResourceView,
    HRESULT* pHResult
);
```

Sampler state

A sampler state allows us to access sampling state information of a texture.

The sampler state that controls how the shader handles filtering, MIPs, and addressing.

To create the sampler state, we will use

```
ID3D11Device::CreateSamplerState()
```

Sampler state

```
typedef struct D3D11_SAMPLER_DESC {
    D3D11_FILTER Filter;
    D3D11_TEXTURE_ADDRESS_MODE AddressU;
    D3D11_TEXTURE_ADDRESS_MODE AddressV;
    D3D11_TEXTURE_ADDRESS_MODE AddressW;
    FLOAT MipLODBias;
    UINT MaxAnisotropy;
    D3D11_COMPARISON_FUNC ComparisonFunc;
    FLOAT BorderColor[4];
    FLOAT MinLOD;
    FLOAT MaxLOD;
} D3D11_SAMPLER_DESC;
```

Texture filtering

Texture filtering refers to the way values are read and combined from the source and made available to the shader.

Filtering can be used to **improve quality** but at the cost of more expensive texture sampling, since some filtering types can cause more than one value to be read and combined to produce a single color value that the shader sees.

Different combinations of the texture’s MIN, MAG, and MIP levels can be choosen besides the sampling type.

Filtering methods

The most common texture filtering methods, in increasing order of computational cost and image quality are:

- ▷ Point sampling
- ▷ Bilinear sampling
- ▷ Trilinear sampling
- ▷ Anisotropic filtering

Point sampling

Point sampling, also known as nearest-neighbor sampling, for the filter is the **fastest** type of sampling.

It works by fetching a **single value** from the texture with no further modifications.

The value chosen is the **texel closest** to the pixel's center.

Bilinear sampling

Bilinear sampling, will perform bilinear **interpolation** on the value sampled at that texture coordinate as well as **several samples** surrounding it.

The interpolated value (combined results) is what the shaders will see.

The samples selected by this filtering are the **four texels closest** to the pixel’s center.

This combination of multiple nearby values can **smooth** the results a bit, which can cause a **reduction** in rendering **artifacts**.

Trilinear sampling

Trilinear filtering works by performing bilinear sampling around the texel for the **two closest MIP levels** and interpolating the results.

When a surface goes from using one MIP level to another, there can be a **noticeable change** in the appearance of that surface at that moment of change.

Interpolating between the closest MIP levels, after bilinearly filtering both levels, can greatly help to **reduce these** rendering **artifacts**.

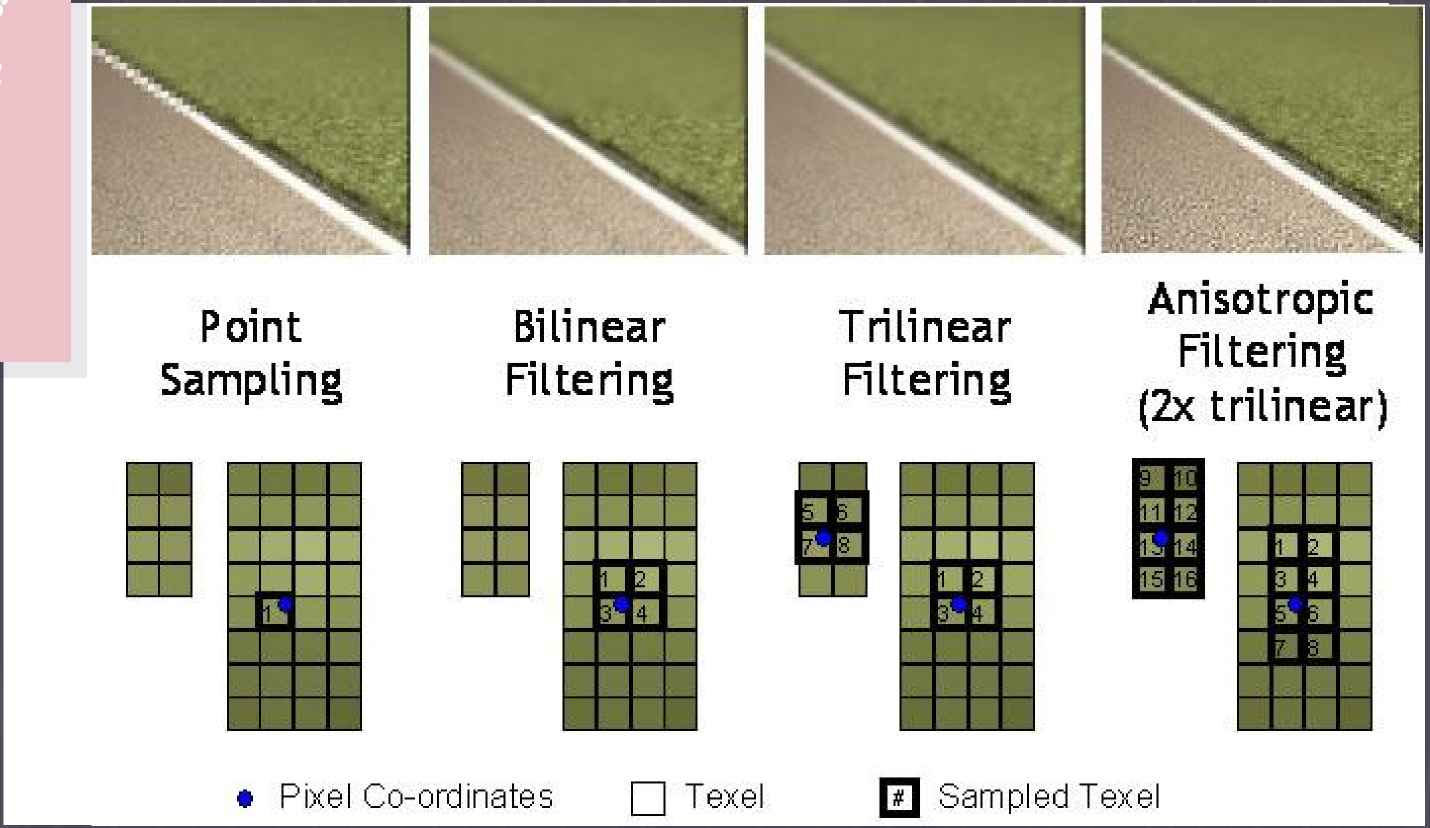
Anisotropic filtering

Anisotropic filtering works by trilinearly sampling in a **trapezoid area** instead of a square area.

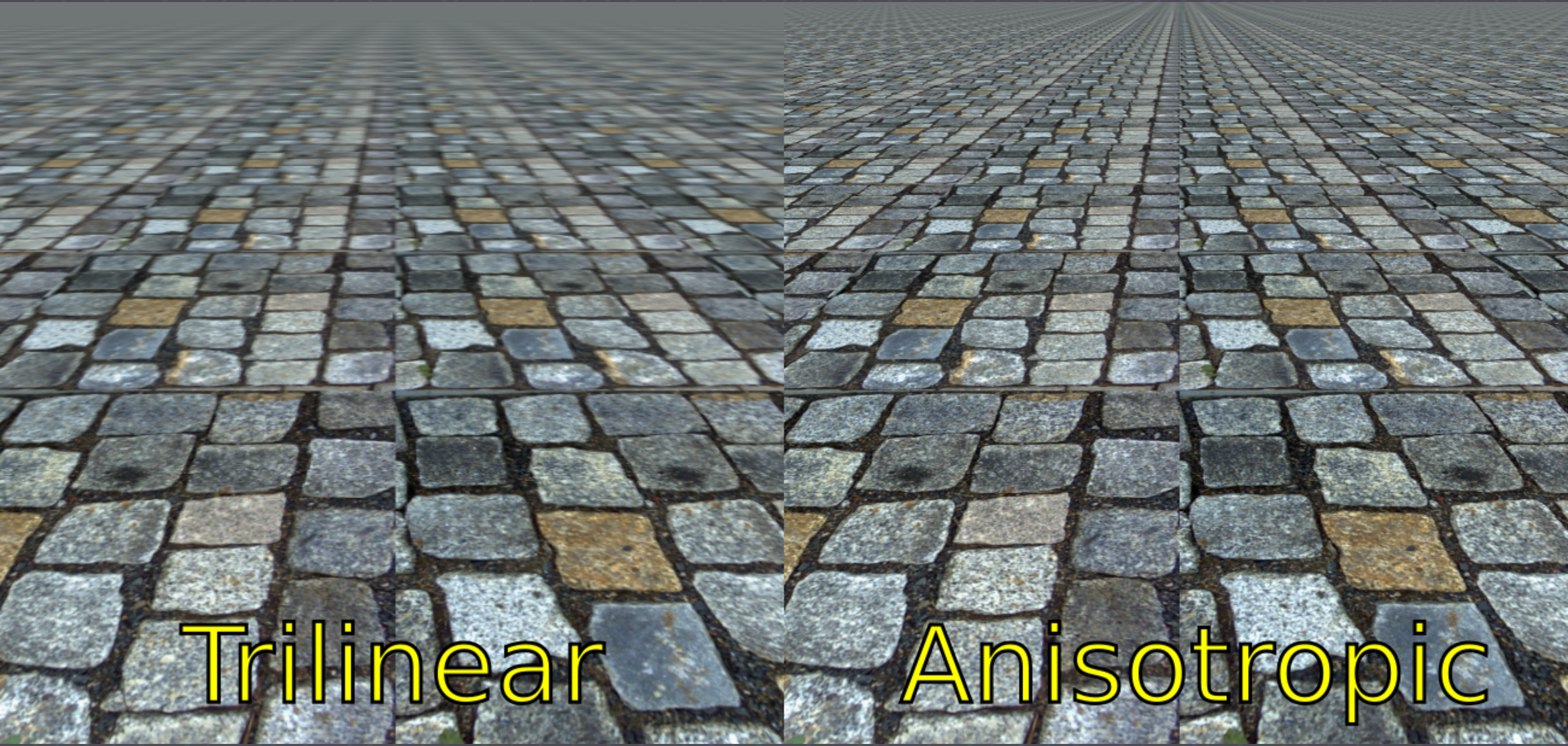
Bilinear and trilinear filtering work best when looking directly at the surface because of the square area of sampling, but when viewing a **surface at an angle**, such as a **floor or terrain** in a 3D game, a noticeable amount of **blurriness** and rendering **artifacts** can appear.

Anisotropic filtering takes angles into consideration and samples using a different shaped area.

Filtering comparison



Anisotropic filtering



Texture address modes

The texture address mode tells Direct3D how to handle values outside of this range. The texture address mode for the U, V, and R can be one of the following values:

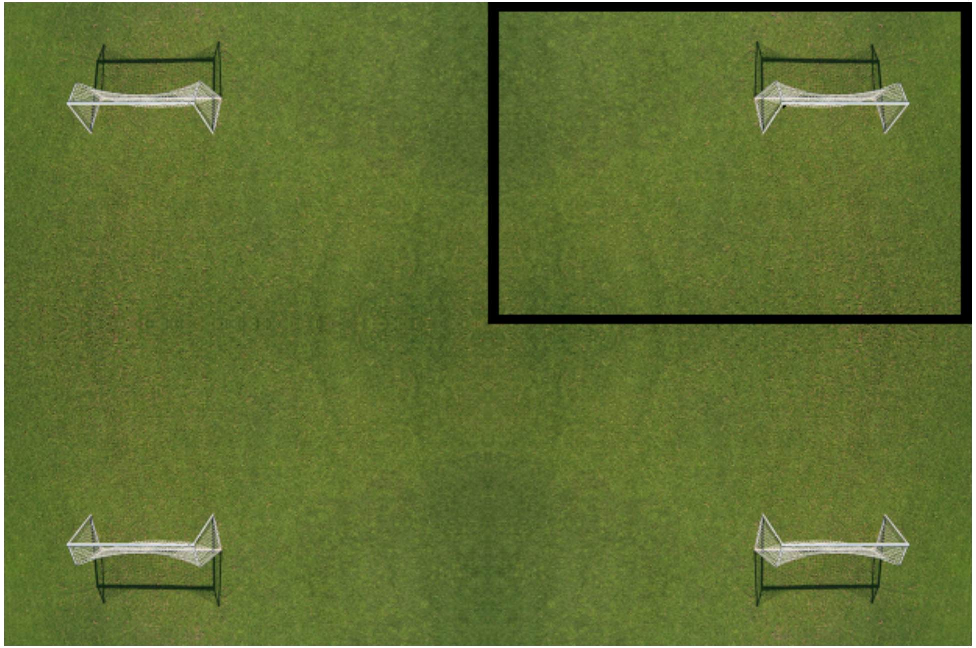
```
typedef enum D3D11_TEXTURE_ADDRESS_MODE {
    D3D11_TEXTURE_ADDRESS_WRAP,
    D3D11_TEXTURE_ADDRESS_MIRROR,
    D3D11_TEXTURE_ADDRESS_CLAMP,
    D3D11_TEXTURE_ADDRESS_BORDER,
    D3D11_TEXTURE_ADDRESS_MIRROR_ONCE,
} D3D11_TEXTURE_ADDRESS_MODE;
```

Texture address modes: WRAP

WRAP for the texture address will cause the texture to wrap around and repeat.

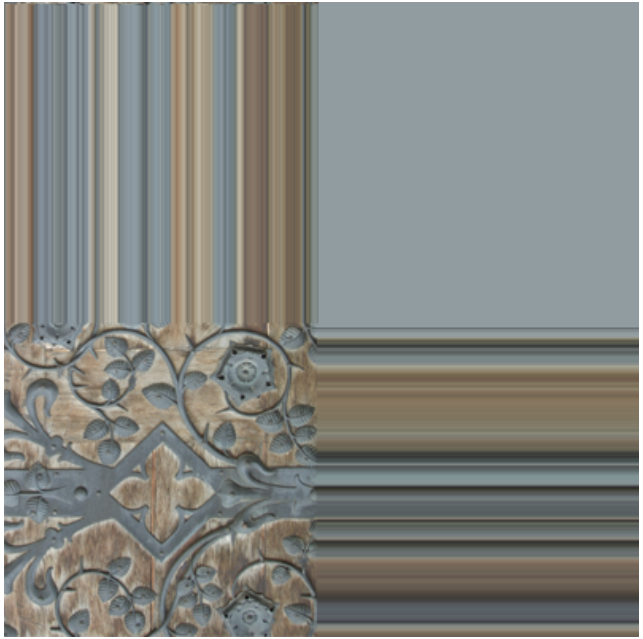
Texture address modes: MIRROR

The MIRROR texture address mode will cause the texture to repeat but in a mirror direction.



Texture address modes: CLAMP

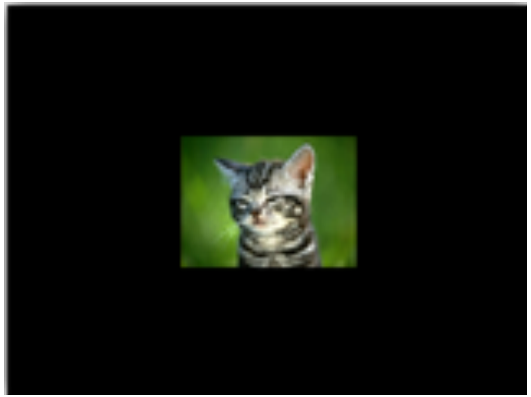
CLAMP will simply clamp the values in the 0.0 to 1.0 range.



Texture address modes: BORDER

BORDER address mode will set any pixels outside of the 0.0 to 1.0 range to a specified border color.

The border color is specified by another member of the D3D11_SAMPLER_DESC called BorderColor, which is an array of four floating-point values.



MIP maps configuration

Next member in the D3D11_SAMPLER_DESC structure is the **level-of-detail** (LOD) bias for the MIP.

This value is an **offset** of the MIP level to use by Direct3D. For example, if Direct3D specifies that the MIP level to use is 2 and the offset is set to 3, then the MIP ultimately used will be level 5.

The last two members of the D3D11_SAMPLER_DESC structure are used to **clamp the min and max** MIP levels that can be used. For example, if the max is set to 1, then level 0 will not be accessed (note that level 0 is the highest resolution).

Anisotropic quality

Following the LOD bias is the **max anisotropic** value to use during anisotropic filtering and the comparison function.

The max anisotropic value can be **between 1 and 16** and is not used for point or bilinear filtering.

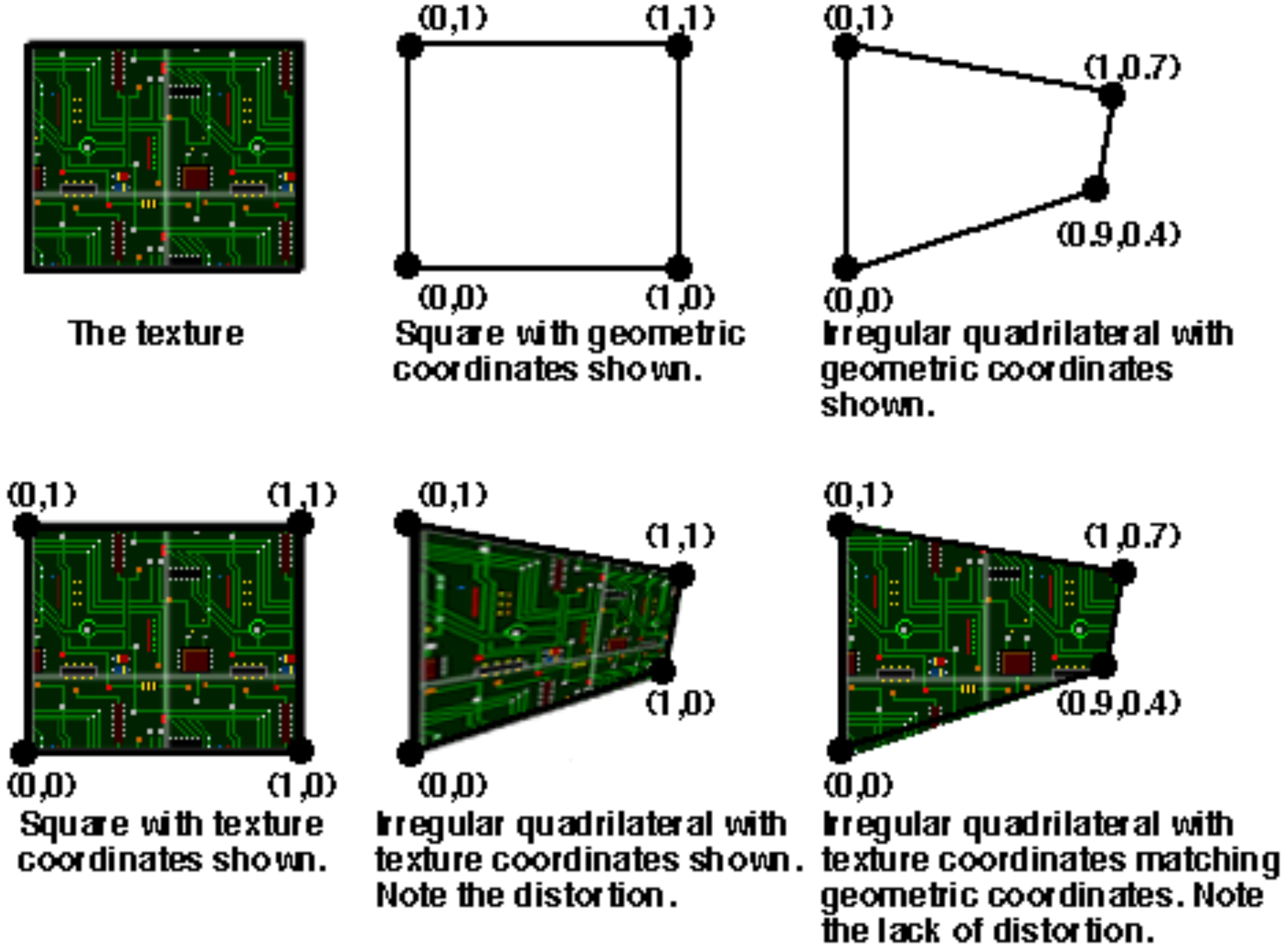
Texture coordinates

Before we can map the image onto our model, we must first define the texture coordinates on each of the vertices.

Since images can be of any size, the coordinate system used has been normalized to [0, 1].

The top left corner of the texture corresponds to (0,0) and the bottom right corner maps to (1,1).

Texture coordinates



The texture

Square with geometric coordinates shown.

Irregular quadrilateral with geometric coordinates shown.

Square with texture coordinates shown.

Irregular quadrilateral with texture coordinates shown. Note the distortion.

Irregular quadrilateral with texture coordinates matching geometric coordinates. Note the lack of distortion.