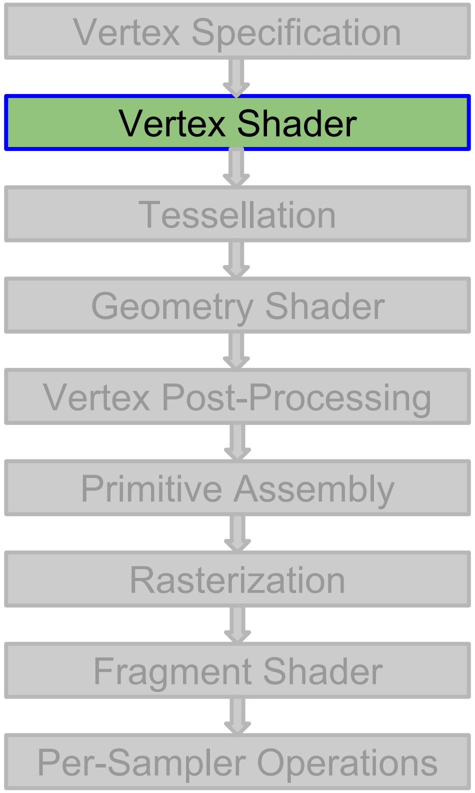
Game Architecture
Graphics II

Attack of the Matrices

Recap

- We’ve described our world using **triangles**.
- Triangles are made up of **vertices**.
- We’ve specified vertex data in **buffers** of a particular **vertex format**.
- We’ve issued a **draw call** to draw the geometry.
- Now what?

Programmable Pipeline



Scene Representation

- How do we describe objects in the scene?
- How do we describe scene topology?
- How do we describe where things are?
- How do we describe how we see things?

Scene Representation

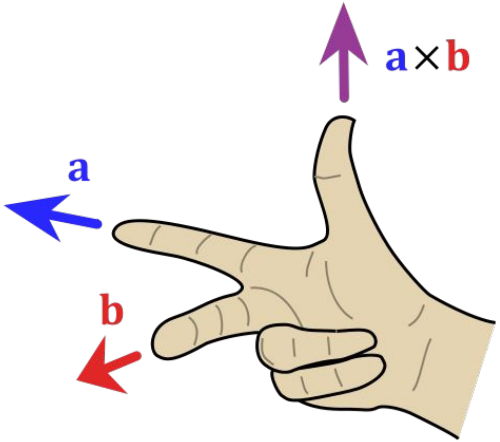
- How do we describe objects in the scene?
- How do we describe scene topology?
- **How do we describe where things are?**
- How do we describe how we see things?

Coordinate Systems

- Handedness
- World space
- Model space
- View space

Coordinate Systems

- **Handedness**
- World space
- Model space
- View space



Handedness determines the direction of the cross product result.

Assuming positive x to the right, and positive y up…

- In a right-handed system, the z-axis comes out of the screen.
- In a left-handed system, the z-axis goes into the screen.

Image courtesy of Wikipedia

Coordinate Systems

- Handedness
- **World space**
- Model space
- View space

World space is the highest order coordinate system which describes absolute positions for all objects in the game.

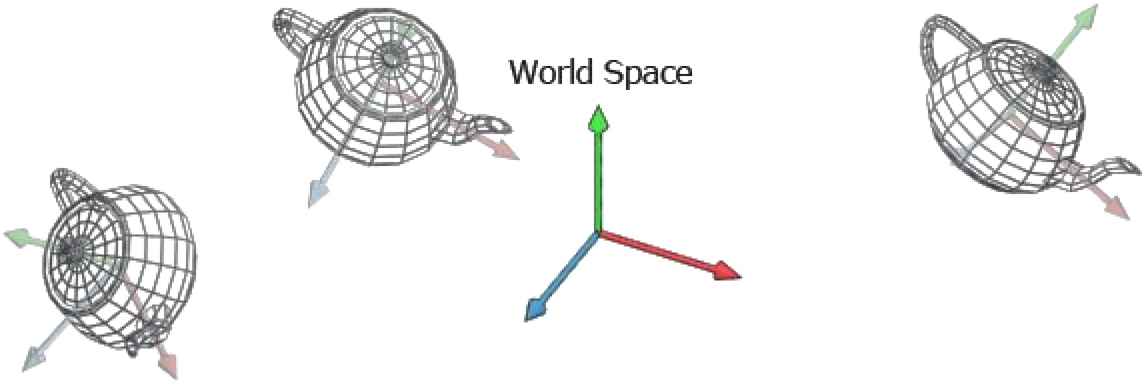


Image courtesy of Marco Alamia via Coding Labs

Coordinate Systems

- Handedness
- World space
- **Model space**
- View space

Model space is the coordinate system local to a single object in the game world.

The origin is the origin point of the model, and is typically user defined. It could be the exact center, a corner, the bottom center; anywhere.

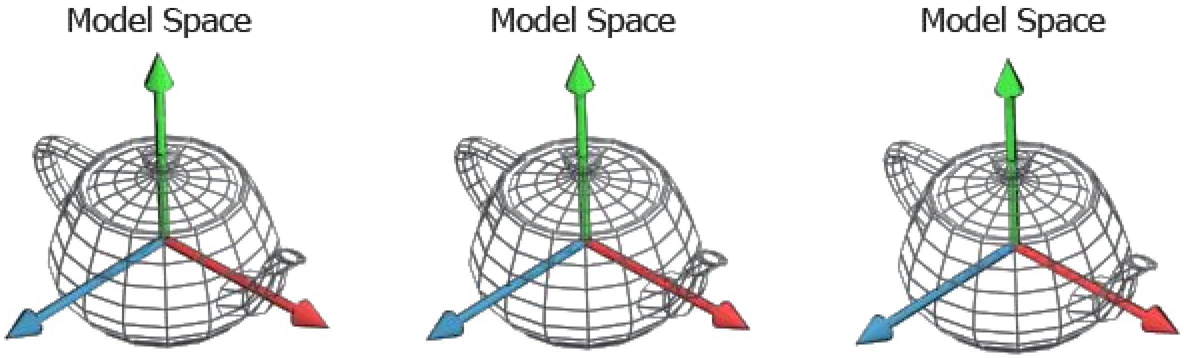


Image courtesy of Marco Alamia via Coding Labs

Coordinate Systems

- Handedness
- World space
- Model space
- **View space**

View space describes the coordinate system of the camera, with the origin being the position of the camera.

Transforming into view space is an essential part of the graphics pipeline.

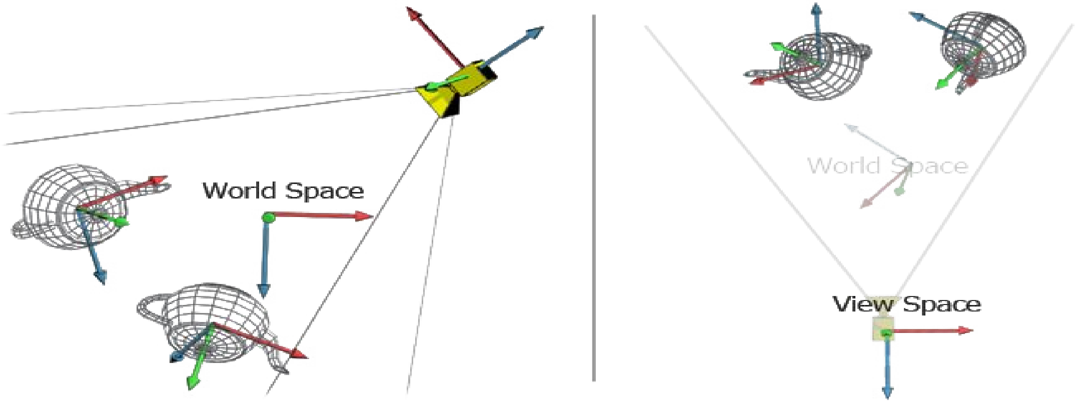


Image courtesy of Marco Alamia via Coding Labs

Scene Representation

- How do we describe objects in the scene?
- How do we describe scene topology?
- How do we describe where things are?
- **How do we describe how we see things?**

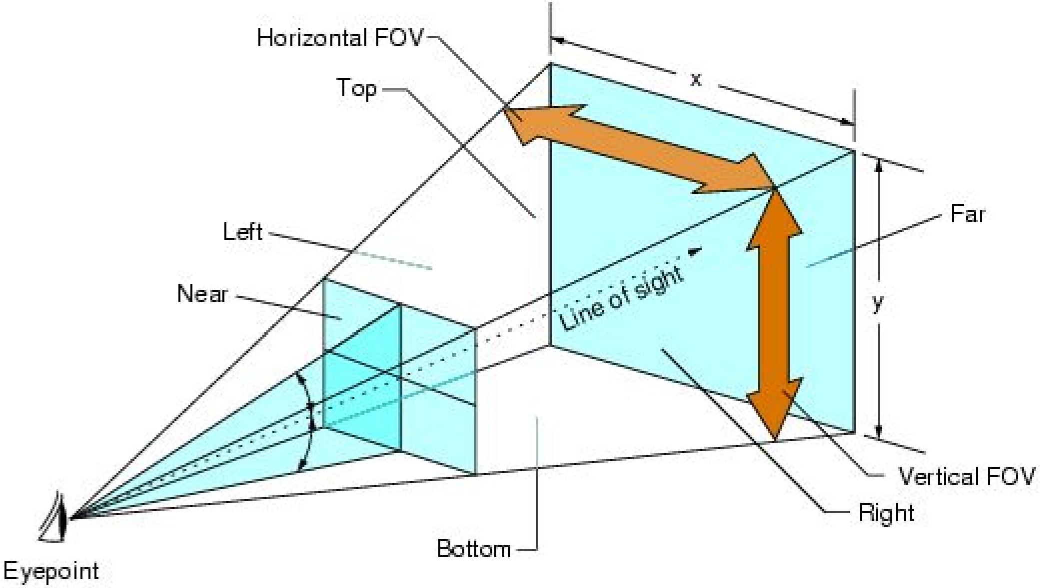
Virtual Camera

- A virtual camera can be created with three components:
    - A position.
    - A forward, or ‘look-at’ vector.
    - An up vector.
- It also needs a frustum.
    - A frustum defines the bounds that the camera can see.
    - To define the frustum, we also need...

Virtual Screen

- The screen is defined by dimensions, and therefore aspect ratio.
  - 1280x720 - 16:9
  - 1920x1080 - 16:9
  - 1920x1200 - 16:10
  - 1024x768 - 4:3
- The screen dimensions define the near plane of the frustum.
  - The near plane is defined to be a certain distance from the camera position.
  - The far plane as well.
  - The field-of-view angles defines the rest of the frustum planes.
  - Note that vertical or horizontal field-of-view can be calculated using one another and the aspect ratio.

Example Frustum



$$\text{Aspect Ratio} = \frac{y}{x} = \frac{\tan(\text{vertical FOV}/2)}{\tan(\text{horizontal FOV}/2)}$$

Image courtesy of Northeastern University

Transformations

- Our goal is to get something from the world to the screen. How?
- We need a series of transformations (matrices)
  - Model - Transform from model into world space.
  - View - Transform from world space into view space.
  - Projection - Transform from view space into clip space.
  - *Viewport - Transform from NDC to screen space.*
    - This is done in the hardware.

Calculating the View Matrix

- The view matrix transforms the world into camera space.
- This essentially means applying the inverse of the camera's transform.
  - Translation
  - Orientation
- Translation is easy:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x & -y & -z & 1 \end{bmatrix}$$

Calculating the View Matrix

- Orientation is a little trickier.
- Recall that rows (or columns) of a transformation matrix are orthogonal vectors representing an orientation.
- Find them for the camera!
  - Given **direction (z)** (eye - at) and **up** vector…
  - Cross **direction** and **up** to get **right (x)**.
  - Cross **right (x)** and **direction (z)** to get an accurate **up (y)**.
  - Then stick ‘em in a matrix:

$$\begin{bmatrix} x_x & x_y & x_z & 0 \\ y_x & y_y & y_z & 0 \\ z_x & z_y & z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Calculating the View Matrix

- But wait!
  - That describes a transformation from the camera space to world space.
  - We want world space to camera space.
- Solution: invert the matrix.
  - It’s orthogonal, so the inverse is just the transpose.
- Finally, combine with translation.
  - The translation is pre-multiplied with the orientation.

Calculating the View Matrix

- So, given:
  - Camera position (eye)
  - Right vector (x)
  - Up vector (y)
  - Forward vector (z)

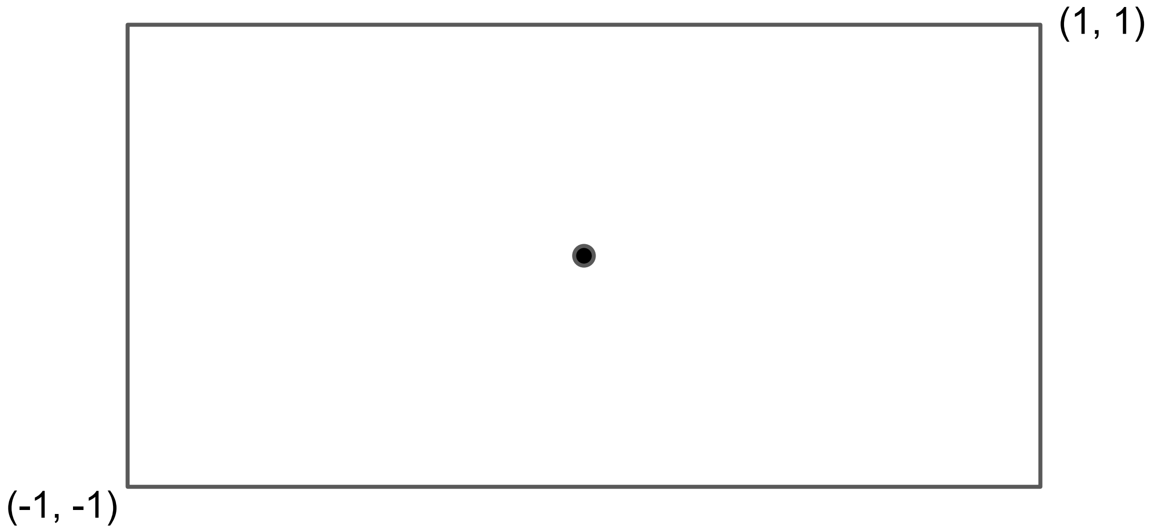
$$\begin{bmatrix} x_x & y_x & z_x & 0 \\ x_y & y_y & z_y & 0 \\ x_z & y_z & z_z & 0 \\ -dot(eye, x) & -dot(eye, y) & -dot(eye, z) & 1 \end{bmatrix}$$

Calculating the Projection Matrix

- The projection matrix takes a point in the frustum and prepares it for a transformation to a point on a plane.
  - Basically flattening the 3D scene onto a 2D plane.
- The destination is typically a rectangle.
  - Output screen
  - Texture
  - GUI minimap

Normalized Device Coordinates

- Abbreviated NDC
- Describe the screen as a square from (-1, -1) to (1, 1)
  - Origin at (0, 0)



Orthographic vs. Perspective

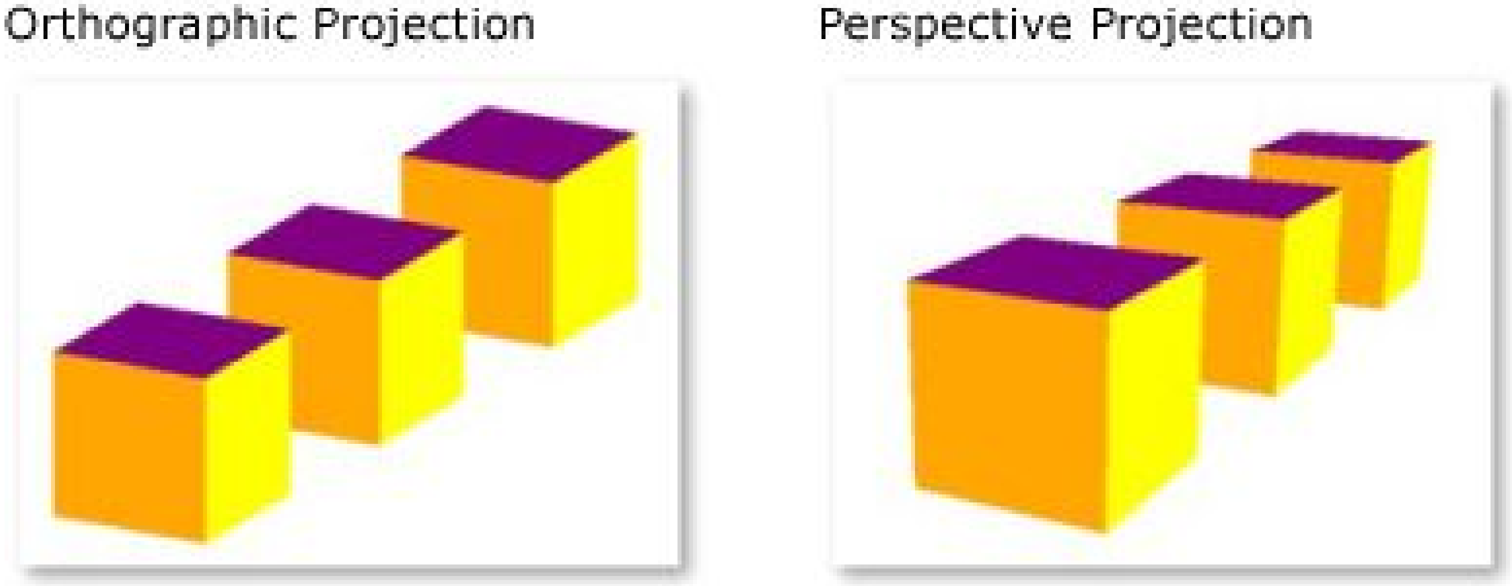
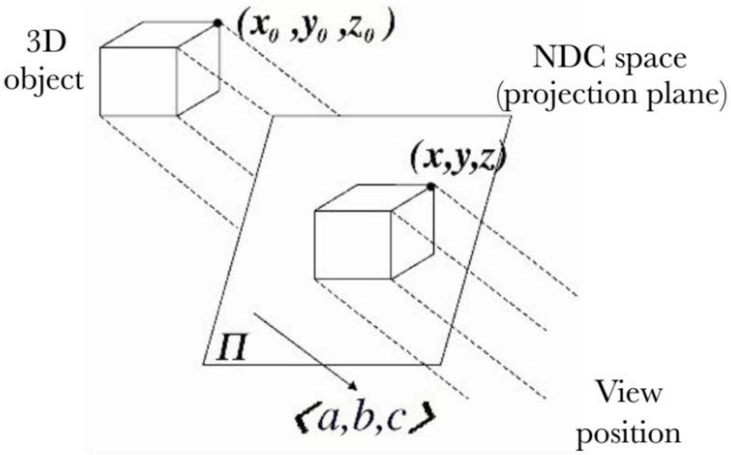
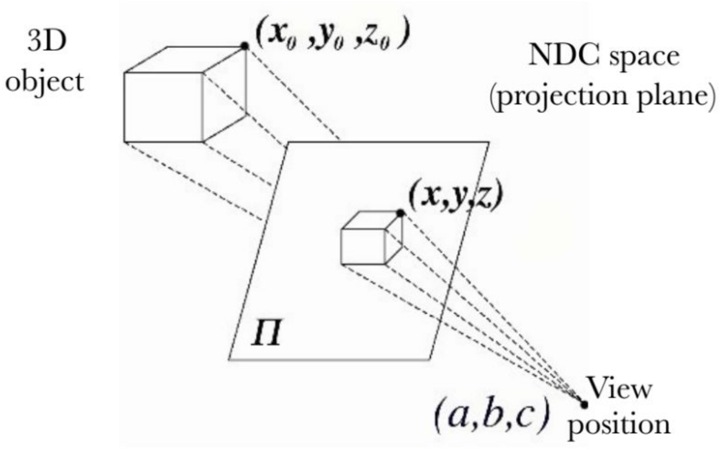


Image courtesy of MSDN

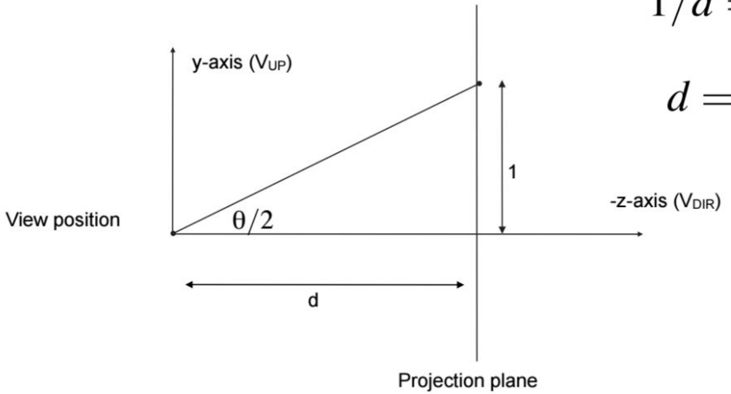
Orthographic vs. Perspective





Field of View

- Related to the location of the projection plane

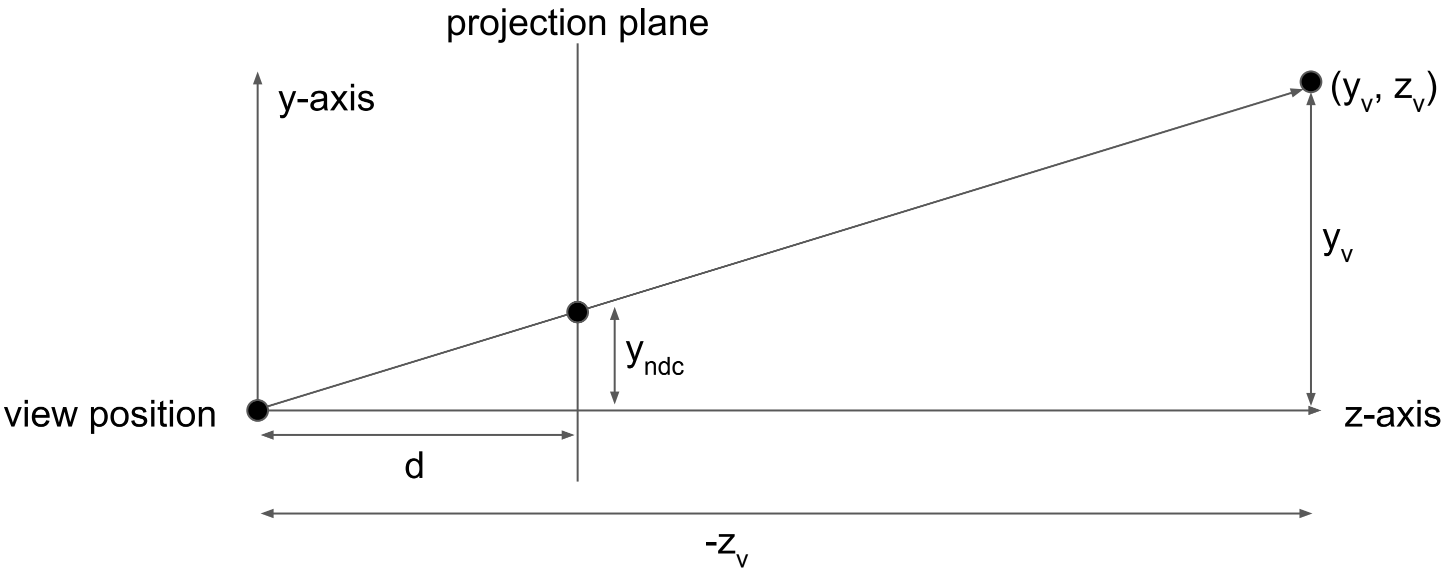


$$1/d = \tan(\theta_{fov}/2)$$

$$d = \frac{1}{\tan(\theta_{fov}/2)}$$

- We'll need this distance next.

Perspective Projection



$$\frac{y_{ndc}}{d} = \frac{y_v}{-z_v} \quad \text{or} \quad y_{ndc} = \frac{dy_v}{-z_v}$$

Perspective Projection

- The target probably isn't a square.
- We need to adjust by the aspect ratio:

$$a = \frac{w_s}{h_s}$$

- If we assume that *y* height is 1, we only need to adjust *x*:

$$x_{ndc} = \frac{dx_v}{-az_v}$$

Perspective Projection

- The further away an object is, the smaller it should appear.
  - Solution is to divide *x* and *y* by the *z* coordinate.
  - Hardware is going to perform division by the *w* component.
  - So we need to move the *z* coordinate into the *w* position.
- Which gives us the homogenous perspective matrix:

$$\begin{bmatrix} \frac{d}{a} & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & -d & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Perspective Projection in Action

$$\begin{bmatrix} \frac{d}{a} & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & -d & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} x_v & y_v & z_v & w_v \end{bmatrix} \quad \begin{bmatrix} \frac{dx_v}{a} & dy_v & -dz_v & -z_v \end{bmatrix}$$

Projecting the Z Coordinate

- We want to use the z coordinate for depth and sorting.
- It needs to be mapped onto a (-1, 1) range depending on the near and far planes.

$$z_c = \frac{z_v(n+f) + 2nf}{n-f}\left(\frac{1}{-z_v}\right)$$

- For DirectX, the range is (0, 1). Ugh.

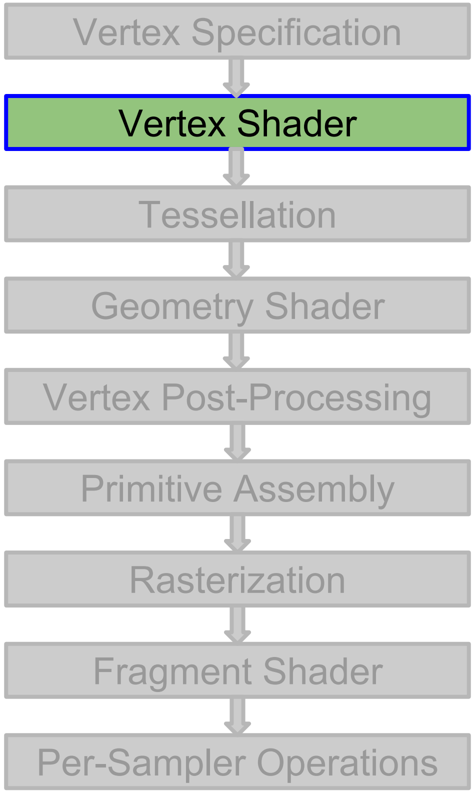
Calculating the Projection Matrix

$$\begin{bmatrix} \frac{d}{a} & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & -1 \\ 0 & 0 & \frac{2nf}{n-f} & 0 \end{bmatrix}$$

Order of Multiplication?

- This depends on your matrix layout and multiplication code.
- For A * B:
    - Row major layout and A rows * B columns **or**
    - Column major layout and A columns * B rows
        - Pre-multiplication = local
        - Post-multiplication = world
    - Row major layout and A columns * B rows **or**
    - Column major layout and A rows * B columns
        - Pre-multiplication = world
        - Post-multiplication = local

Programmable Pipeline



Programmable Shaders

- Miniature programs which operate on vertex and fragment data.
  - Vertex shader outputs vertex position in homogenous clip space, and per-vertex values.
    - gl_Position
  - Fragment shader outputs fragment color(s).
    - gl_FragColor
    - And possibly depth information.
- There are many built-in functions for common operations.
  - See the OpenGL or HLSL reference pages for a list.

Shading Languages

- Most shading languages are C-like.
  - GLSL - OpenGL Shading Language
  - HLSL - High Level Shading Language (DirectX)
  - Cg
- They are compiled to bytecode similar to assembly.
- GPU execution units execute the bytecode.

GLSL

- Program variable qualifiers.
  - **in** - A per-element input value, like position.
  - **out** - A per-element output, like fragment color.
  - **uniform** - A per-object constant, like model-view-projection matrix or texture.
- Basic types.
  - C++ like - void, float, int, bool
  - Vectors - vec2, vec3, vec4, ivec2, etc.
  - Matrices - mat2, mat3, mat4
  - Textures - sampler2D
  - Arrays - e.g. float[5]

Vector Types

- Shading language vector types offer different access patterns.
  - { x, y, z, w }, typically used for positions or normals.
  - { r, g, b, a }, typically used for colors.
  - { s, t, p, q }, typically used for texture coordinates.
- Swizzling
  - By definition, rearranging components of a vector.
  - Given vector = vec4(1.0, 2.0, 3.0, 4.0)
    - vector.xxxx = { 1.0, 1.0, 1.0, 1.0 }
    - vector.wzyx = { 4.0, 3.0, 2.0, 1.0 }
    - vector.yyzz = { 3.0, 3.0, 4.0, 4.0 }
    - And so on...

GLSL

- Control flow
  - Mostly the same as C/C++.
  - Notable addition is **discard**, which rejects a current fragment from drawing.
- Built-in functions
  - Trig - sin, cos, tan, asin, acos, atan
  - Math - pow, exp, log, exp2, log2, sqrt, inversesqrt
  - And many more - dot, cross, normalize, clamp, mix, step, reflect, etc.
- Texture lookup
  - texture2D(sampler2D, vec2)
  - There are others, but this is most often used.

Example

```
#version 400

layout (location = 0) in vec3 in_position;

uniform vec3 u_color;

out vec3 out_color;

void main(void)
{
    // Do something with color and position.
    out_color = vec4(1.0, 0.0, 0.0, 1.0);
    // Assign gl_Position as position in homogenous clip space.
}
```

Example: Linking Shaders

Vertex shader declarations:

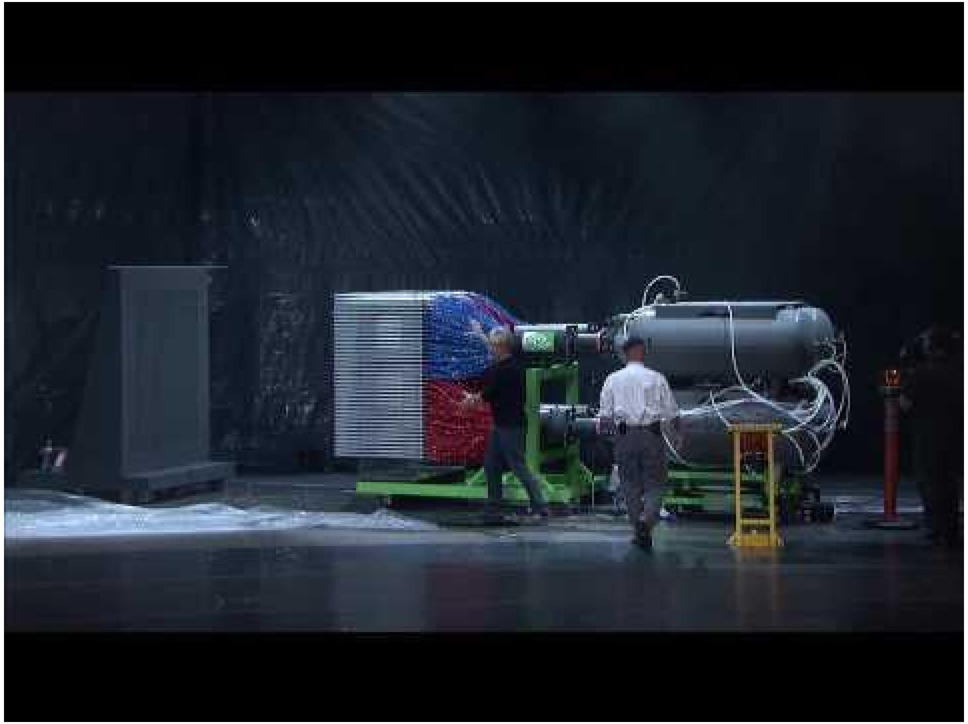
```
layout(location = 0) in vec3 in_position;
layout(location = 1) in vec3 in_color;
layout(location = 2) in vec2 in_texcoord;

out vec3 color;
out vec2 texcoord0;
```

Fragment shader declarations:

```
in vec3 color;
in vec2 texcoord0;
```

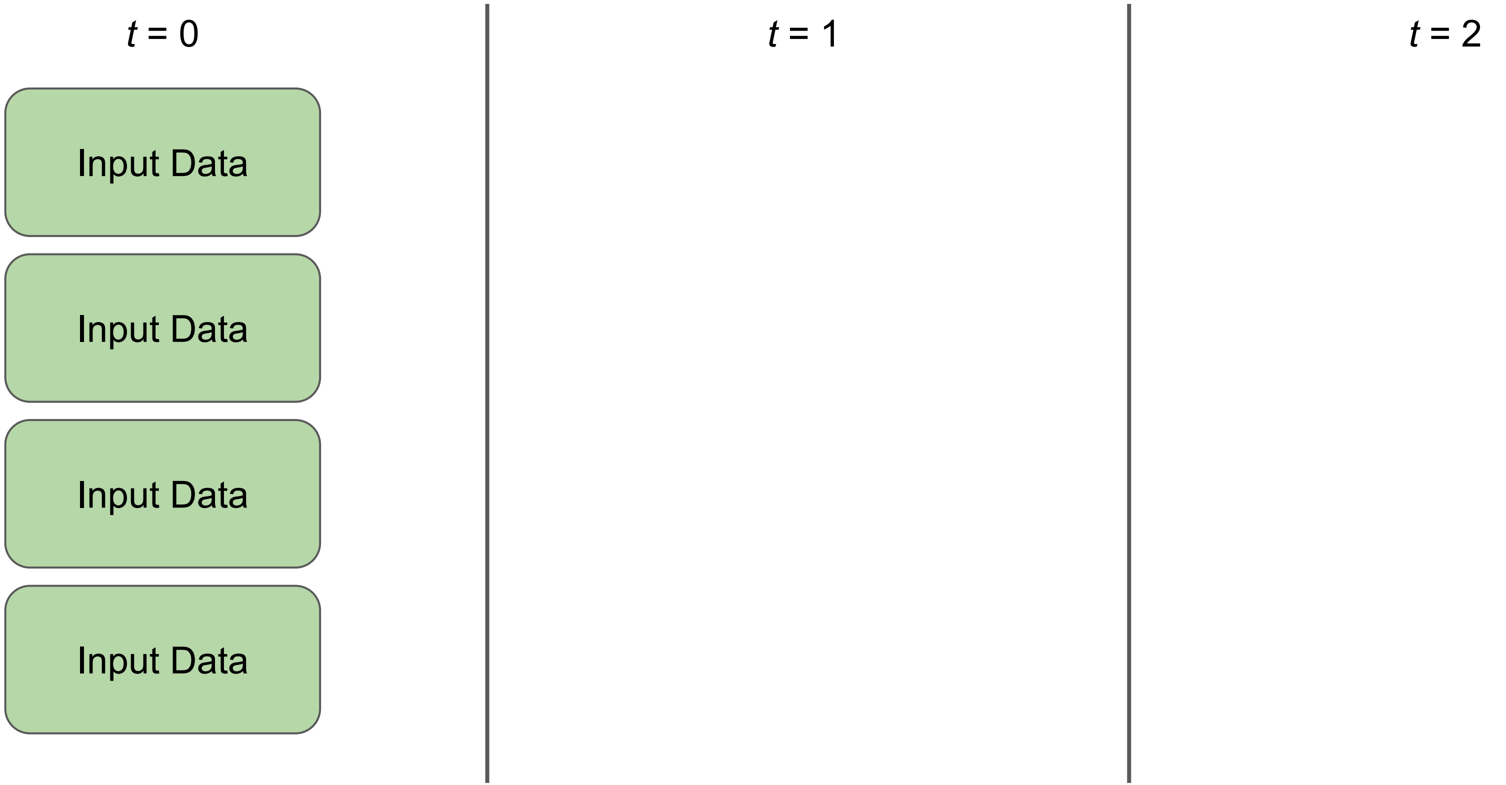
Mythbusters!



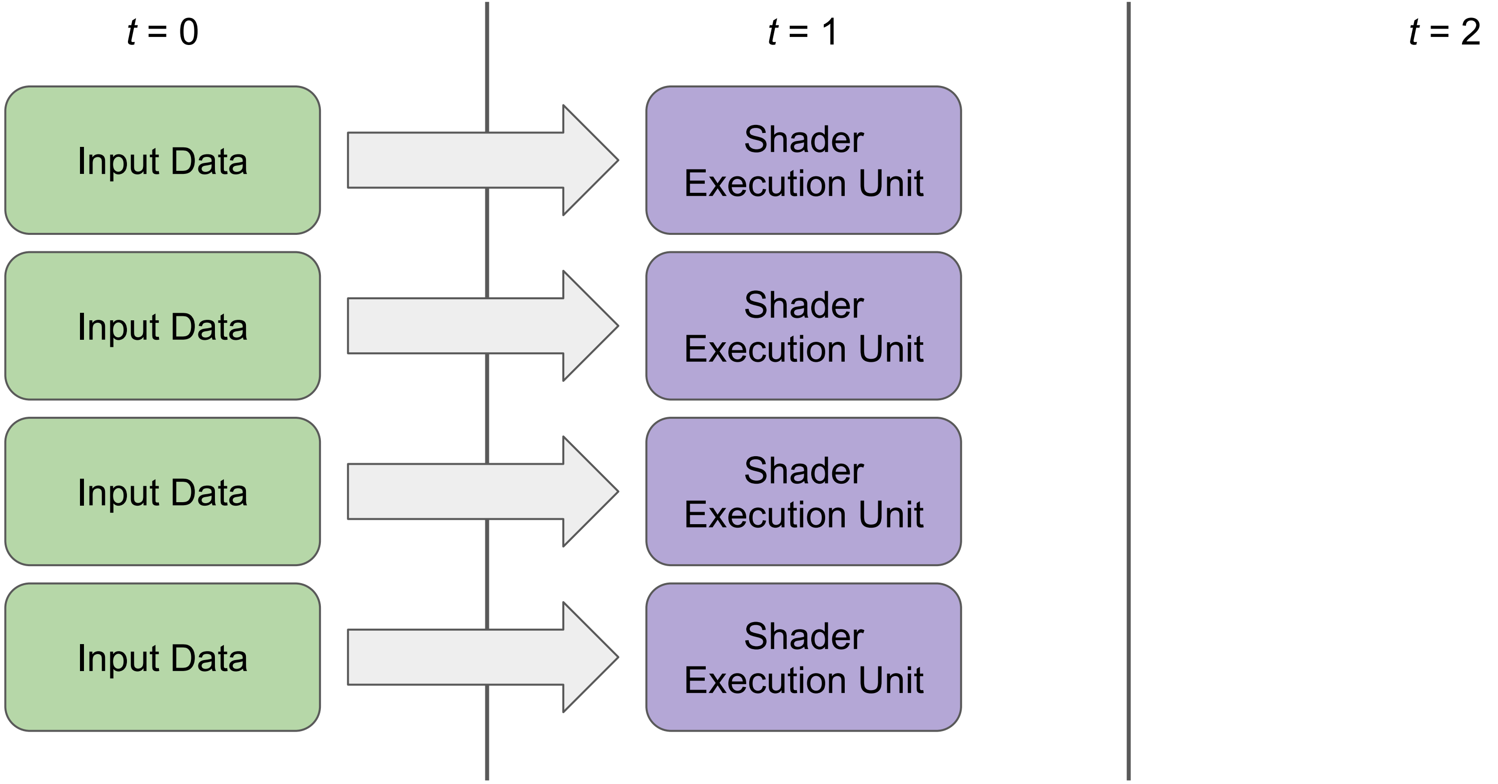
Shader Units

$t = 0$	$t = 1$	$t = 2$

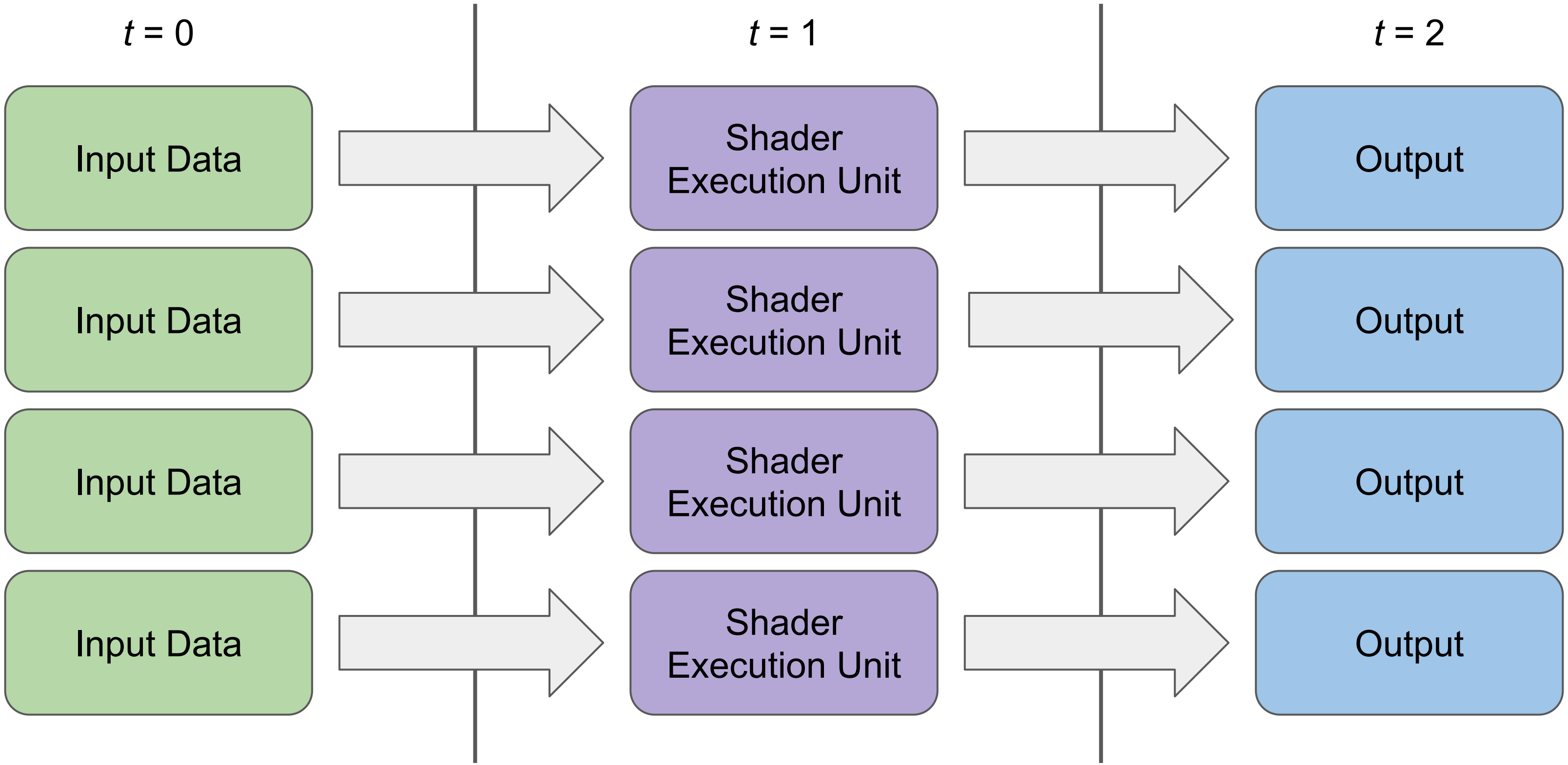
Shader Units



Shader Units



Shader Units



Vertex Shaders

- Modern vertex shaders:
  - Animate vertex positions.
  - Transform vertex positions into homogenous clip space.
  - Pass through (and possibly manipulate) per-vertex attributes.
- And that's really about it…
  - Fragment shaders get more interesting.
- They can be leveraged for optimization.
  - It's possible to use built-in hardware interpolation to your advantage.
    - Perform heavy calculations per vertex, rather than per-fragment.
  - This is really a last-resort optimization.

OpenGL Shaders

- OpenGL manages shaders with **shader objects**.
  - glCreateShader
  - glShaderSource
- Shader **programs** are pairings of vertex and fragment shaders.
  - glCreateProgram
  - glAttachShader
  - glLinkProgram
  - glUseProgram
- Fun fact: shaders are usually not compiled until you actually use them.

End of Lecture

- Homework 3 is now actually assigned.
- It’s due next Thursday, 2/16.
- On Monday, we’ll finish up our trip through the graphics pipeline.