Coordinate Systems, Animation, and Depth Buffering

CS 385 - Class 7
15 February 2022

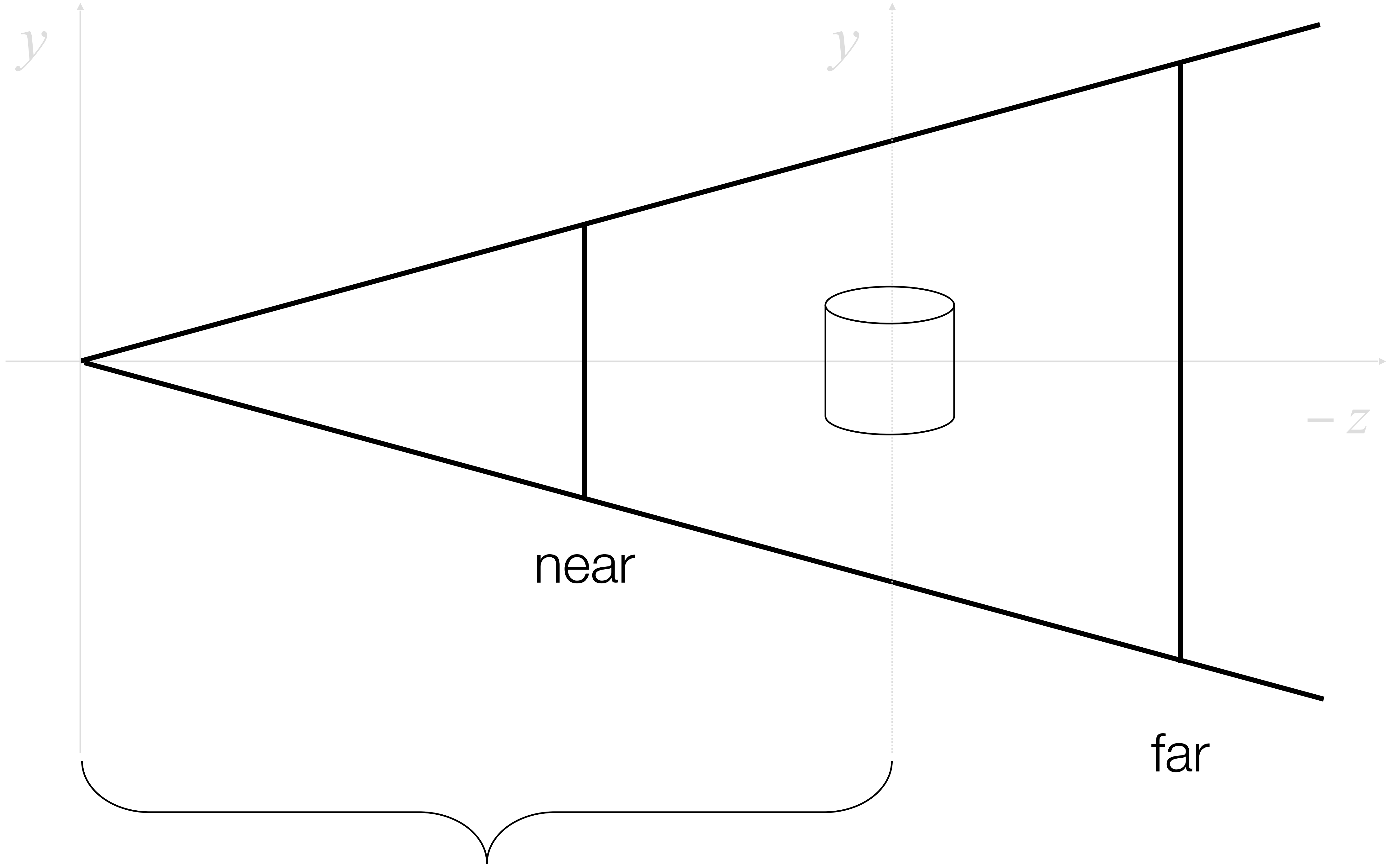
Viewing Transformations

Viewing Transformations

- Reorient world coordinates to match eye coordinates
- Basically just a modeling transform
  - affects the entire scene
  - usually a translation and a rotation
- Usually set up after the projection transform, but before any modeling transforms

The Simplest Viewing Transform

- “Push” the origin into the viewing frustum



$$z = -\frac{1}{2}(near + far)$$

Transforming World to Eye Coordinates

- Viewing transform

```
vec3 eye, look, up;
// assign values for eye, look, ...
var m = lookAt(eye, look, up);
```

- Creates an *orthonormal basis*
  - a set of linearly independent vectors of unit length

Creating an Orthonormal Basis

$$\hat{n} = \frac{\overrightarrow{look} - \overrightarrow{eye}}{\left\| \overrightarrow{look} - \overrightarrow{eye} \right\|}$$

$$\hat{u} = \frac{\hat{n} \times \overrightarrow{up}}{\left\| \hat{n} \times \overrightarrow{up} \right\|} \quad \Rightarrow \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ -n_x & -n_y & -n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\hat{v} = \hat{u} \times \hat{n}$$

- To complete `lookAt()`, we need a final translation to the eye position

Multiplying Matrices in JavaScript (using MV.js)

- This process creates a lot of matrices
  - you’ll use them in your shader (or perhaps even your application)
- Recall, order of matrices for multiplication is important
- Always **multiply on the right**

```
v = vec4(...);
S = scale(...);
T = translate(...);
V = lookAt(...);
MV = mul(mul(V, T), S);
P = perspective(...);

pos = mul(mul(P, MV), v);
```

Multiplying Matrices in a Shader

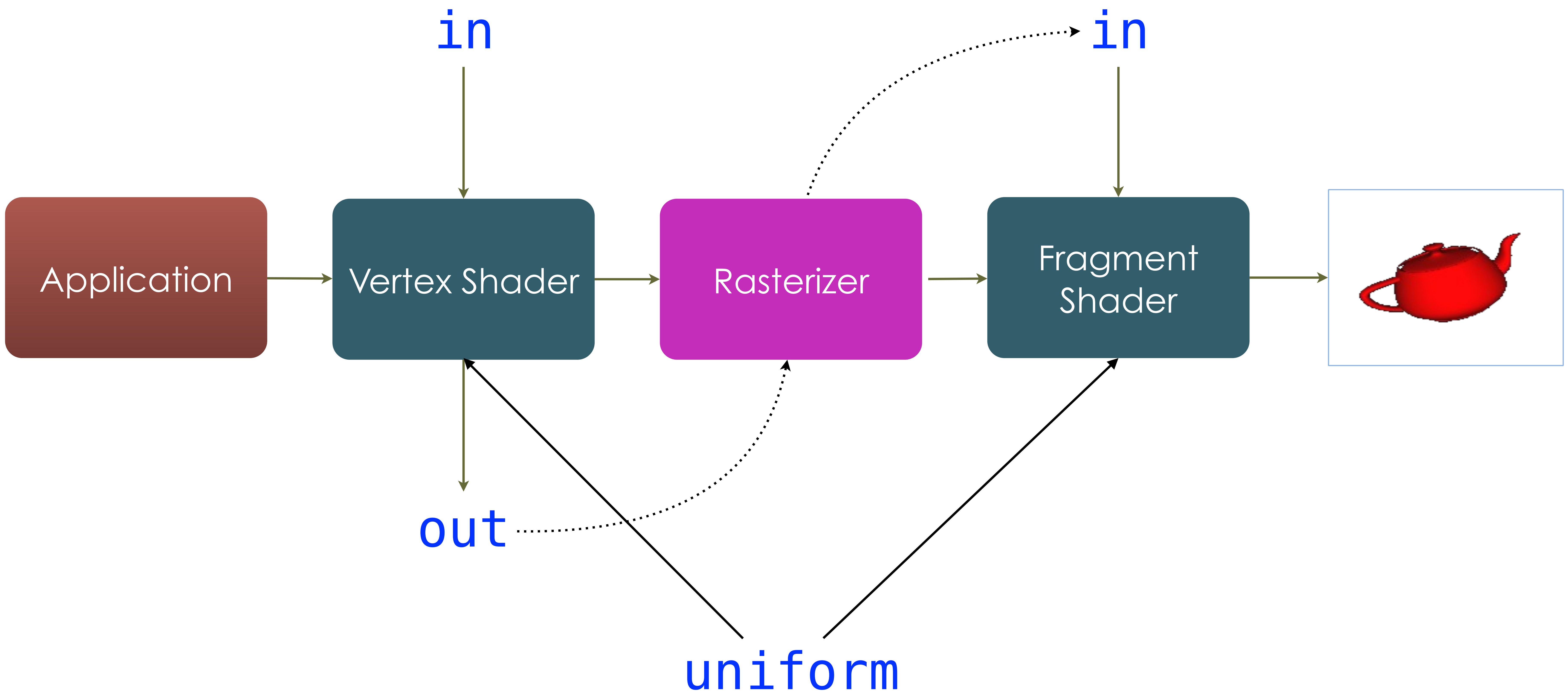
- GLSL makes that much cleaner
- Again, build up transforming the vertex from left-to-right
  1. projection transform
  2. viewing transform
  3. modeling transforms
  4. vertex position

```
in  vec4 aPosition;
in  vec4 aColor;
out vec4 vColor;

// Magic we'll discuss momentarily

void main()
{
  vColor = aColor;
  gl_Position = P * MV * aPosition;
}
```

Always **multiply on the right**

Uniform Variables

Graphics Pipeline



Uniform Variables

- Shared between vertex and fragment shaders in the same shader program
- Uniforms are like “constants” inside of a shader
  - their value doesn’t change until the application updates it
- declared as a `uniform`

```
in  vec4 aPosition;
in  vec4 aColor;
out vec4 vColor;

uniform float t;

void main()
{
  vColor = aColor;
  gl_Position = t * aPosition;
}
```

Transformations are (usually) Uniforms

- Use `uniforms` for sending transformation matrices into shaders

```
in  vec4 aPosition;
in  vec4 aColor;
out vec4 vColor;

uniform mat4 MV;
uniform mat4 P;

void main()
{
  vColor = aColor;
  gl_Position = P * MV * aPosition;
}
```

Managing Uniforms

- Since we're encapsulating our models as JavaScript objects, we can create a useful interface for managing our uniforms
- Create a uniforms property to hold all the uniform locations used in a shader
  - those values you get back from `gl.getUniformLocation()`

```
function Cylinder( gl, ... ) {

  this.positions = { ... };
  this.colors = { ... };

  this.program = initShaders( ... );

  this.uniforms = {
    MV : gl.getUniformLocation(this.program, "MV"),
    P : gl.getUniformLocation(this.program, "P")
  };

  this.render = function () { ... };
}
```

Managing Uniforms

- It can be helpful to have an interface for the application to set the uniform's values
- Create some top-level properties to hold the uniforms values
- In your application, you can set their values:

```
Cylinder.P = perspective( ... );
Cylinder.MV = mult( ... );
```

```
function Cylinder( gl, ... ) {

  this.positions = { ... };
  this.colors = { ... };

  this.program = initShaders( ... );

  this.uniforms = {
    MV : gl.getUniformLocation(this.program, "MV"),
    P : gl.getUniformLocation(this.program, "P")
  };

  this.P = mat4();
  this.MV = mat4();

  this.render = function () { ... };
}
```

Drawing with Uniforms

- In the object's `render()` function
  - set the matrix uniform's values using `gl.uniformMatrix4fv()`
  - use `gl.uniform1f()` for things like time
  - Be care with which variable is which
    - `this.uniforms.MV` - uniform *location* from the shader program
    - `this.MV` - matrix's *value* set by your JavaScript application
- the `false` parameter indicates if the matrix should be transposed
  - it will always be false

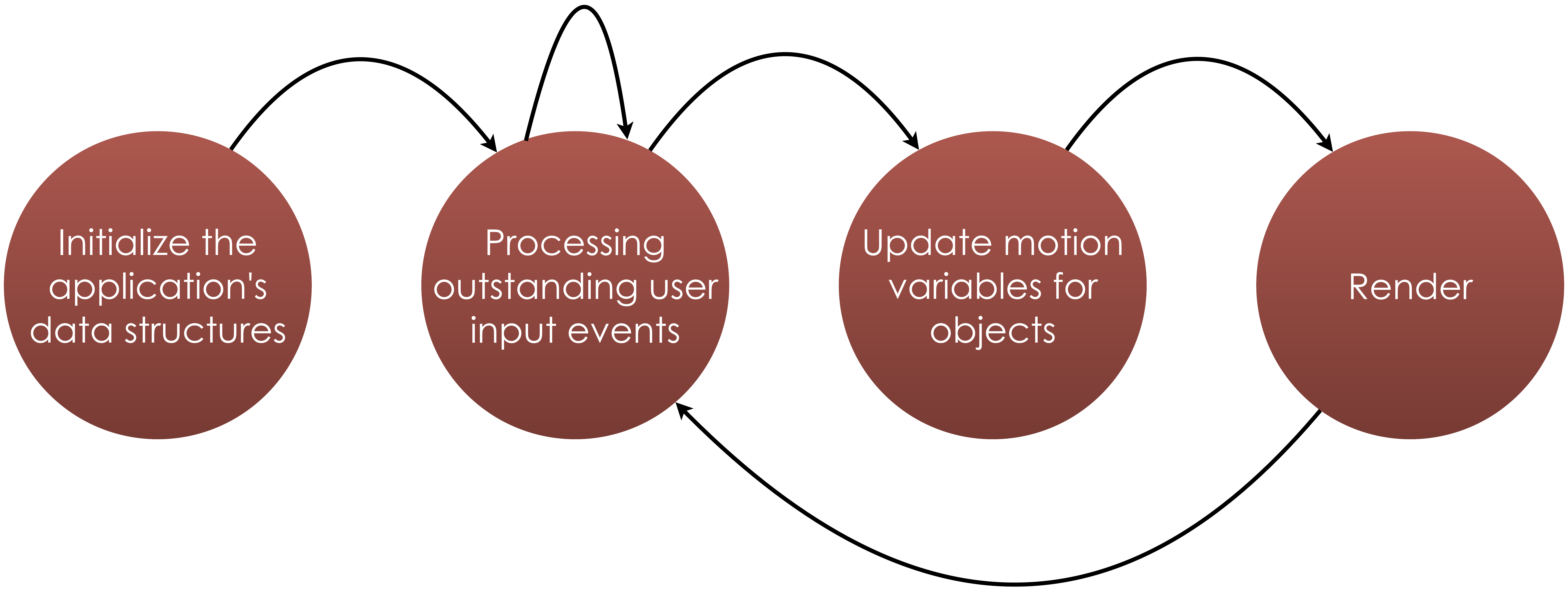
```
function Cylinder( gl, ... ) {

  this.positions = { ... };
  this.colors = { ... };

  this.program = initShaders( ... );

  this.uniforms = {
    t : gl.getUniformLocation(this.program, "t"),
    MV : gl.getUniformLocation(this.program, "MV"),
    P : gl.getUniformLocation(this.program, "P")
  };

  this.P = mat4();
  this.MV = mat4();

  this.render = function () {
    gl.useProgram(this.program);
    gl.uniform1(this.uniforms.t, t);
    gl.uniformMatrix4fv(this.uniforms.MV, false,
      flatten(this.MV));
    gl.uniformMatrix4fv(this.uniforms.P, false,
      flatten(this.P);
  };
}
```

`flatten()`

- Helper function in **MV.js** that converts JavaScript arrays into `Float32Arrays`
- All of the **MV.js** types are JavaScript arrays or objects
- They all need to be `flatten()`ed before being passed into a WebGL function

Animation

The Tao of Interactive Applications



Animation

- Display one frame while rendering the next frame
- Often referred to as *double buffering*
  - front buffer is displayed
  - back buffer is being drawn into
- *Swap buffers* to present the newly rendered frame
  - The Web browser implicitly does the swap
  - other APIs require an explicit call for the swap
- See discussion of *frame rate* in Canvas.

Animation

- Specify a function to be executed when it's time to make a new frame
- Call `requestAnimationFrame()`
- This allows input processing and other browser activities to complete

```
function init() {
  ...
  requestAnimationFrame(render);
}

function render() {
  gl.clear(gl.COLOR_BUFFER_BIT);

  // Update transforms and objects
  //   with updated motion variables

  // Set transforms and render

  requestAnimationFrame(render);
}
```

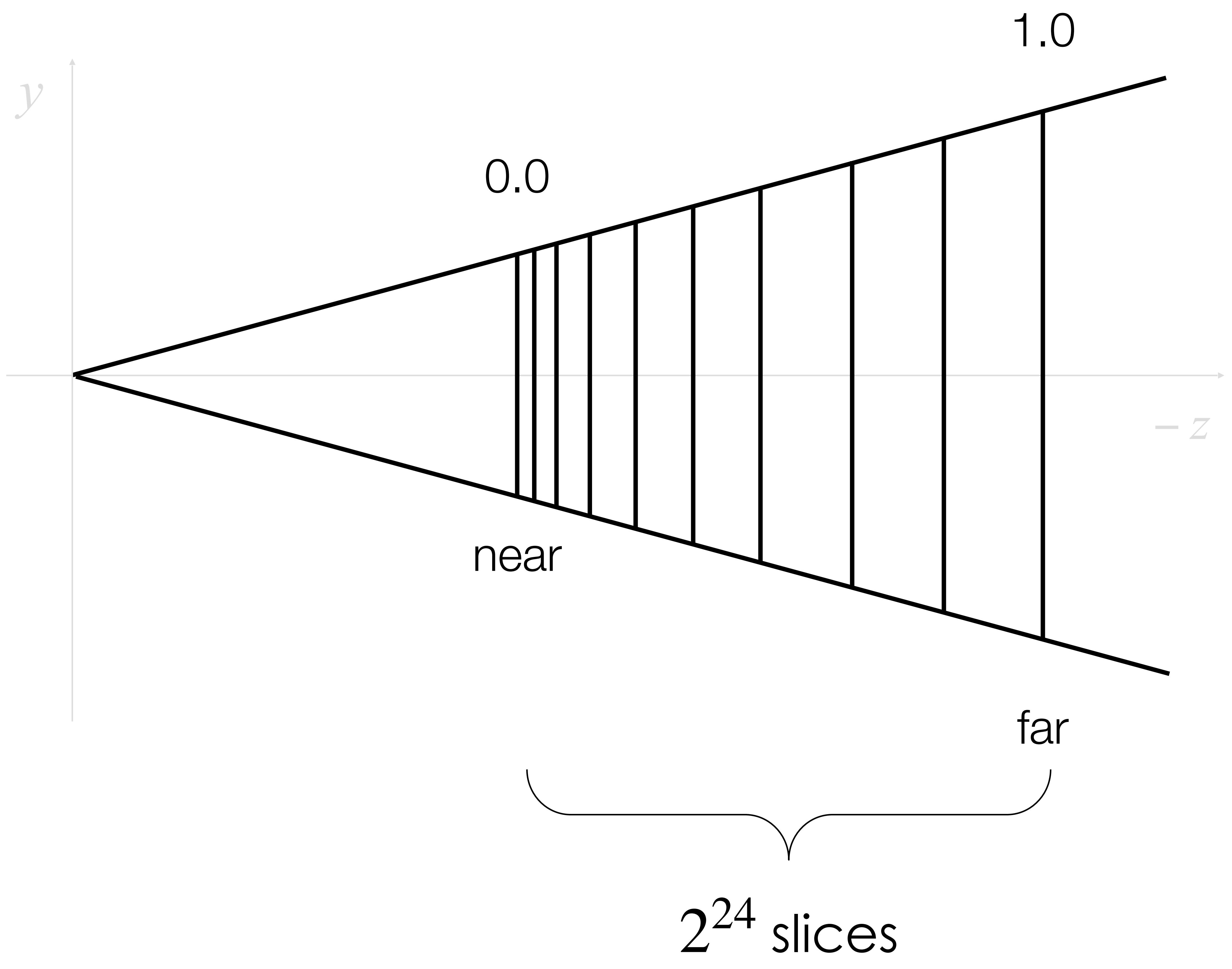
Depth Buffering and Hidden Surface Removal

Occlusion and the Determination of What's Visible

- Drawing points is fun, but it gets a lot more interesting with filled primitives
- Currently, the last primitive drawn to a pixel is what's shown
- To correctly show a scene, an application would need to draw all the primitives in order, farthest to nearest
  - this is called the *Painter's Algorithm*
- Works for simple scenarios, but *interpenetrating geometry* causes it to fail
- Also need to sort primitives each frame if objects have moved relative to each other

Depth Buffering

- Use an additional buffer that stores *depth*, the distance from the eye
- Buffers are often specified by number of bits
  - 24-bit depth $\implies 2^{24}$ depth quantizations
- Depth is mapped to the *depth range*
  - normally the range $[0,\ 1]$, but it's configurable
- Near plane is mapped to zero; far plane to one



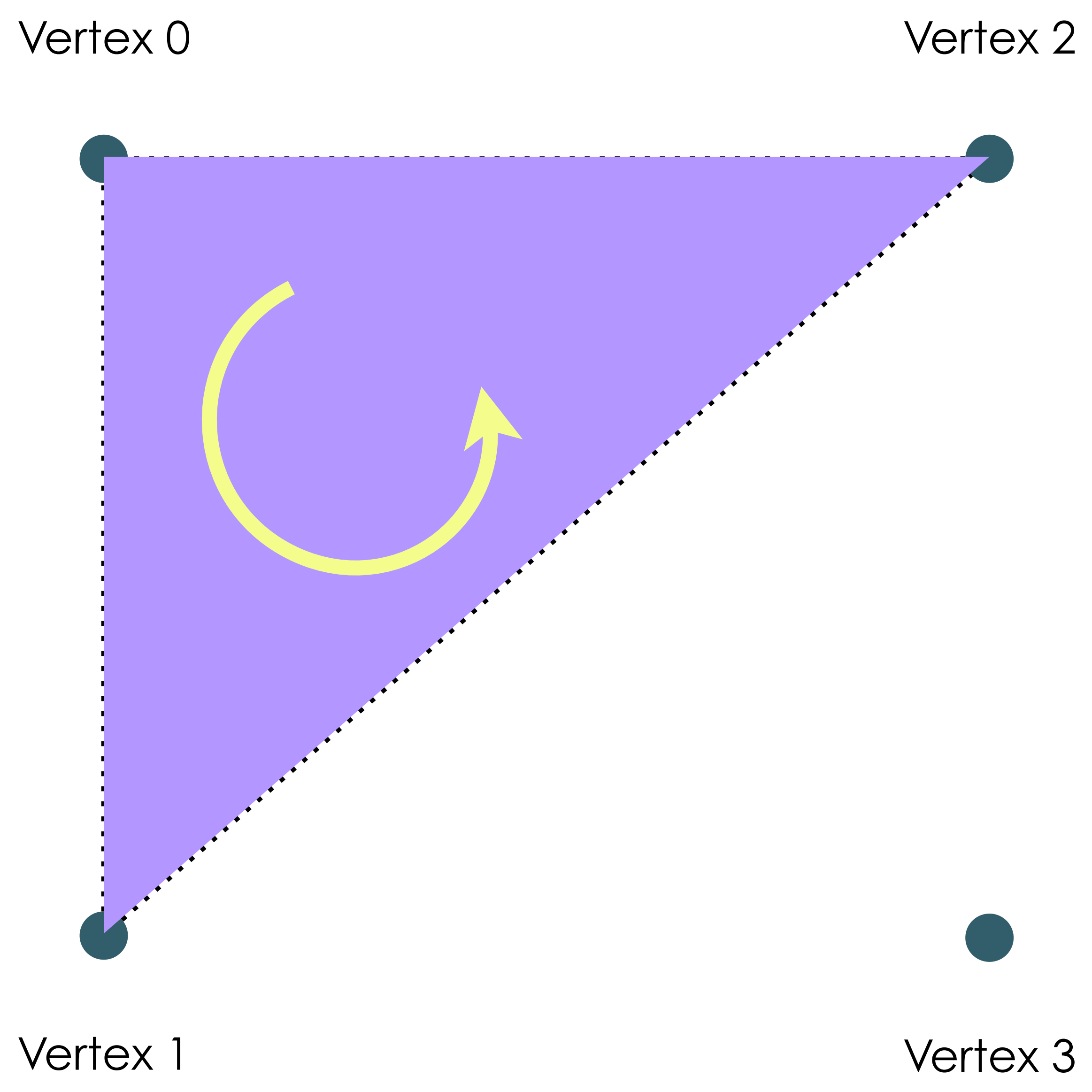
Enabling the Depth Buffer

- Specify a clear value
  - just like our clear color
- Need to enable depth testing
- Clear the depth buffer each frame
  - again, just like color

```
function init() {
  gl.clearColor(r, g, b, a);
  gl.clearDepth(1.0); // default
  gl.enable(gl.DEPTH_TEST);
}

function render() {
  gl.clear(gl.COLOR_BUFFER_BIT|
    gl.DEPTH_BUFFER_BIT);

  // Update transforms and objects
  //   with updated motion variables

  // Set transforms and render

  requestAnimationFrame(render);
```

Hidden Surface Removal

- Recall how vertex ordering specifies the *facedness* of a triangle
- We can have WebGL remove faces based on their winding order



Face Culling

- The rasterizer looks at the facedness of the triangle, and decides if it should be rasterized

$$area = \frac{1}{2}\sum_{i=0}^{n-1} x_i y_{i\oplus 1} - x_{i\oplus 1} y_i$$

where $i \oplus 1 = (i+1) \mod n$

```
function init() {
  gl.clearColor(r, g, b, a);
  gl.enable(gl.CULL_FACE);
  gl.cullFace(gl.BACK_FACE);
}

function render() {
  gl.clear(gl.COLOR_BUFFER_BIT|
    gl.DEPTH_BUFFER_BIT);

  // Update transforms and objects
  //   with updated motion variables

  // Set transforms and render

  requestAnimationFrame(render);
}
```

Visualizing Facedness

- The rasterizer can let a shader know whether a primitive's fragment is front- or back-facing

```
out vec4 fColor;

void main()
{
  vec4 frontColor = vec4(...);
  vec4 backColor = vec4(...);

  fColor = gl_FrontFacing ?
    frontColor : backColor;
}
```

Assignment

Lab Objectives

- Create a Cube object that will render a unit cube centered at the origin
    - determine both the positions and indices for the cube
- Set up projection, viewing, and modeling matrices to have your Cube rotate in 3D
- Helpful steps:
    - create matrices in the application for storing you transformations
    - add uniform matrix variables into your vertex shader
    - add the plumbing to connect your application matrix to the shader matrix
        - you'll need to use calls like `gl.getUniformLocation()`
    - verify you have the cube's faces winding correct using `gl_FrontFacing` in your fragment shader