Chapter 3

GLSL / Shaders

WebGL Basic Outline

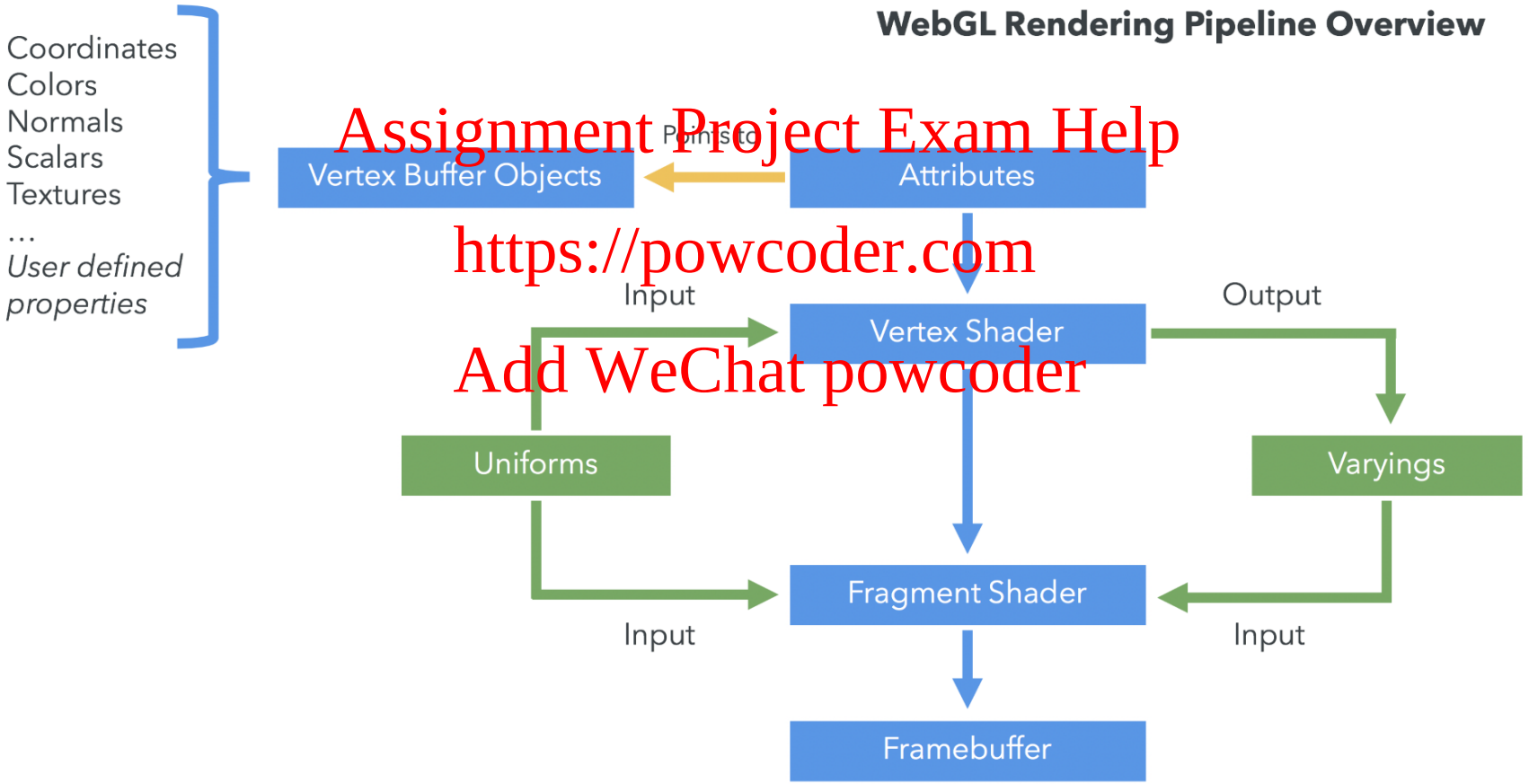
- HTML describes the overall page
  - Specifies region for WebGL canvas
  - Specifies external files to load (like the JS file)
- JS File – `load` event
  - Setup WebGL context
  - Define, compile, and link shaders
  - Compute/load/specify data in VBOs
  - Setup additional events
  - Call `render`
- JS File – other events
  - Possibly update data and call `render`

Shaders

- There are two main shaders:
  - Vertex shader
    - Perform transformations for each vertex and any per-vertex lighting varyings
  - Fragment shader
    - Assigns the color and depth values of each fragment
- Each shader is a complete program
  - Massively parallel execution on GPU
- The first two lines in both shaders is:

```
#version 300 es                  says we use modern version
precision mediump float;         says we use medium precision
```

Rendering Pipeline Summary



Vertex Shader

- **Inputs:** attributes (pointing to VBOs)
- **Outputs:**
  - `gl_Position` (must be in camera/eye position)
  - Optional: varyings (interpolated & sent to fragment shader)
- **Simplest Example:**
  - Copy `aPosition` attribute to `gl_Position`


```
in vec4 aPosition;
void main() {
    gl_Position = aPosition;
}
```

Fragment Shader

- **Inputs:**
  - Optional: varyings (interpolated outputs of vertex shader)
- **Output:**
  - `fragColor` (color of the fragment)
    - Technically can be any name
- **Simplest Example:**
  - Set `fragColor` to fixed color


```
out vec4 fragColor;
void main() {
   fragColor = vec4(1,0,0,1);
}
```

GLSL: GL Shading Language

- "High Level" language for writing the shaders
- C-like with quirks/additions:
  - Matrix/vector types for 2-4 dimensions
  - Overloaded operators
  - C++ like constructors
- Similar to NVIDIA Cg and Microsoft HLSL
- Use WebGL functions to compile, link, and check the shaders

GLSL: Data Types

- C Types: `int`, `float`, `bool`
  - Very strict about mixing `int`s and `float`s
    - `1.0 + 2` not allowed
- Vector types: `vec2`, `vec3`, `vec4`
  - Also have `int`/`boolean` versions (`ivec2`/`bvec4`)
- Matrix types: `mat2`, `mat3`, `mat4`
  - Stored by columns
  - Lookup like array-of-arrays: `m[row][col]`
- Constructors for each type:

```
vec3 a = vec3(1.0, 2.0, 3.0);
vec2 b = vec2(a);
```

Vectors and Matrices

- Vertices/points can be thought of as a mathematical *vector*
  - I will use the terms interchangeably
- The space can be transformed by multiplying the *vector* with a *transformation matrix*
  - The transformation matrix contains information about translation, rotation, scaling, and skew
  - It is just a matrix that has values in special spots
- Lots of graphics is linear algebra…

GLSL: Data Types

- Supports C-like `structs`
  - Think of a class without any methods
- No pointers `*` or reference `&` arguments
  - All variables passed by value (i.e. copied)
- If more than one value needs to be returned from a function than we must use a vector, matrix, or `struct`

GLSL: Operators and Functions

- Several standard math and matrix/vector functions
  - Trigonometric: `sin`, `cos`, `radians`, `degrees`, …
  - Arithmetic: `sqrt`, `distance`, `abs`, `min`, `max`, `floor`, …
  - Linear algebra: `length`, `dot`, `normalize`, `reflect`, …
- Operators work as expected with vectors and matrices; `*` is matrix multiplication:

```
mat4 a;
vec4 b, c, d;
c = b*a; // column vec as a 1D array
d = a*b; // row vector as a 1D array
```

GLSL: Vectors and Swizzling

- Can look up values in a vector like an array or use the attributes as follows:
  - `x, y, z, w` (position)
  - `r, g, b, a` (color)
  - `s, t, p, q` (texture)
- The following are all identical:

```
v[2] == v.z == v.b == v.p
```

- *Swizzling* lets you modify or retrieve multiple attributes at once

```
vec4 a, b;
a.yz = vec2(1.0, 2.0); // assign 2 values
b = a.yxzw; // reorder values
```

GLSL: Data Type Qualifiers

- Supports many C/C++ qualifiers like `const`
- Includes several others due to the nature of the execution model, for example:
  - `in`
  - `out`
  - `uniform`

WebGL1 had no `in` or `out`, instead `attribute` and `varying` were used as appropriate

GLSL: `in` Qualifier

- When used in vertex shader:
  - Variable is an ***attribute***, gets data from VBO
- When in fragment shader:
  - Variable is a ***varying***, get data from interpolation of vertex shader outputs
- Can change at most once per vertex/fragment

GLSL: out Qualifier

- When in vertex shader:
  - Variable is a *varying* outputting data to fragment shader (after interpolation)
- When in fragment shader:
  - Only 1 allowed, variable is the output color of the fragment

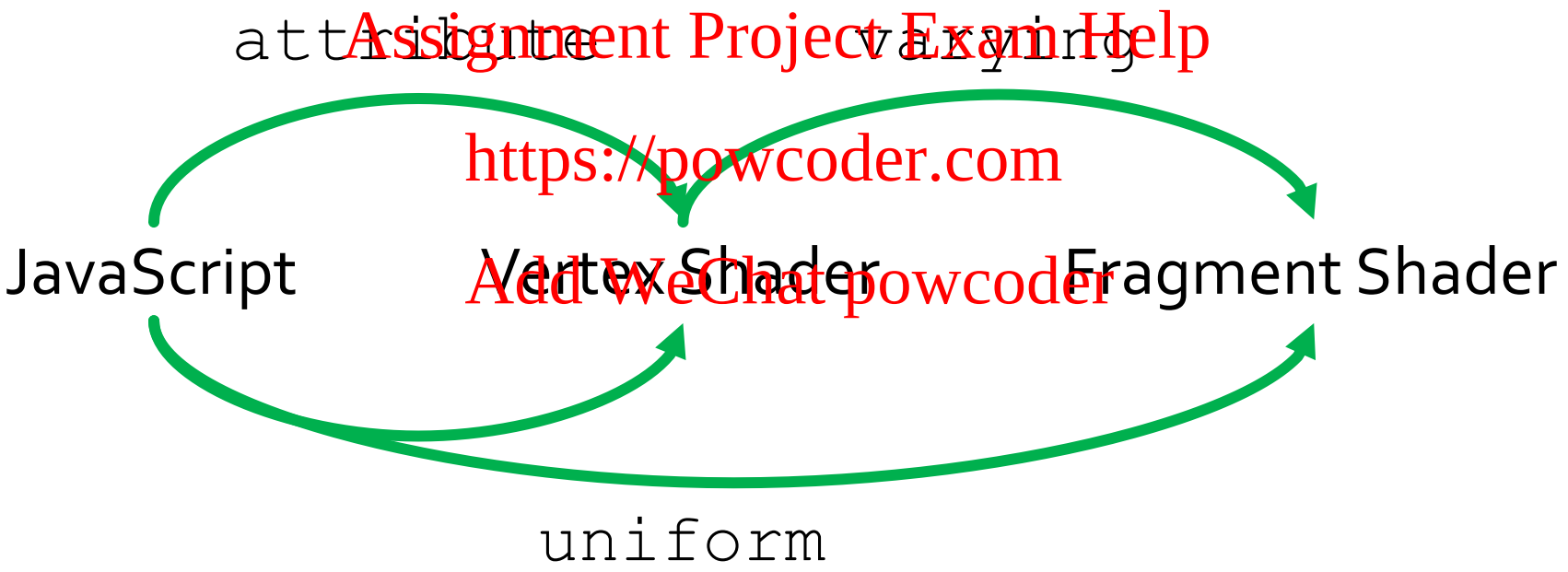
GLSL: uniform Qualifier

- Variable that is constant for an entire primitive (e.g. triangle or line)
- Can easily be changed in JS
- Can be used in either or both shaders
- Cannot be modified in shaders
- Used to pass a piece of information to a shader such as the timestep of an animation, transformation matrix to be applied, or locations of lights

GLSL: Our Naming Convention

- Used in the book and in this class
- `a`: prefix for a vertex `attribute`
  - Vertex shader uses `in`
  - Variable name in JS does not need to be the same but keep them the same for sanity
- `v`: prefix for a `varying` being sent from the vertex to fragment shader
  - Must have the same name in both shaders; vertex shader uses `out` and fragment shader uses `in`
  - Not accessible in JS at all
- `u`: prefix for a `uniform`
  - Variable name in JS does not need to be the same but keep them the same for sanity

GLSL: Passing Values



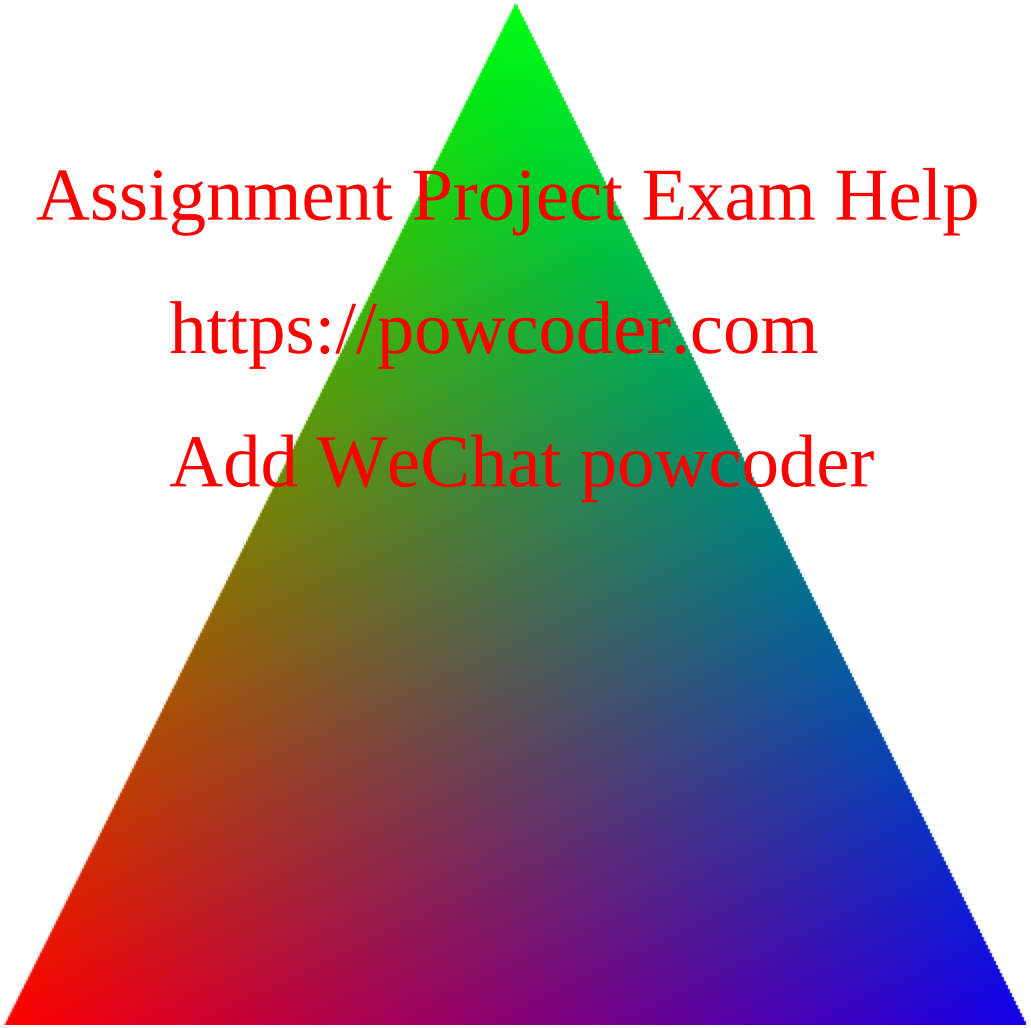
Compiling and Linking Shaders

- Get the shader code into a string
  - Multiple JS string using backticks `
  - A hidden element in the HTML file
  - A separate file loaded dynamically – see `loadFile` function in `common/init-shader.js`
- Compile shaders individually
- Link the shaders together into a single program
- Associate variables in shader program with data from JS application
  - Vertex attributes loaded into GPU buffers
  - Uniform variables sent directly

JS: Vectors and Matrices

- gl-matrix.js library emulates GLSL features
  - `glMatrix.vec3.create()` creates a length-3 vector of 0s
  - `glMatrix.vec2.fromValues(2, 3)` creates vector [2, 3]
  - `glMatrix.mat4.create()` creates a 4x4 matrix of 0s
  - Linear algebra functions: `length`, `dot`, `normalize`, …
  - See glmatrix.net/docs/ for documentation
- Features it cannot emulate, like operator overloading, are provided as functions:
  - `glMatrix.vec2.mult(u, v)`, `glMatrix.vec2.add(u, v)`,…
- Other functions are in built-in `Math` library:
  - `Math.sin(t)`,`Math.abs(t)`,…

Example: Maxwell's Triangle



Maxwell's Triangle with Attributes

- To make Maxwell's Triangle each vertex needs to have a color attribute
  - That attribute must be passed to a varying which will be interpolated across the fragments
- What changes do we need to make to accomplish this?

Maxwell's Triangle with Attributes

- To make Maxwell's Triangle each vertex needs to have a color attribute
  - That attribute must be passed to a varying which will be interpolated across the fragments
- What changes do we need to make to accomplish this?
  - *Create a color VBO*
  - *Associate color VBO with color attribute*
    - Must also get the color attribute location at some point
  - Add an in color variable to vertex shader (attribute)
  - Add an out color variable to vertex shader (varying)
  - Add an in color variable to fragment shader (varying)
  - Use the color variable in the fragment shader

Vertex and Fragment Shaders

- In the JavaScript code we:
  - Compute/load raw data
  - Load data into GPU buffer
  - Associate buffer with a vertex shader attribute
- In vertex shader we are working with a single vertex
  - Have access to the all of attributes for that vertex
  - Must set the magic variable `gl_Position`
  - Can also set any varying variables
- In fragment shader we are working with a single fragment
  - Have access to all of the varyings from the vertex shader and they have been interpolated
  - Must set the magic output variable `fragColor`

Setting Colors

- Colors are ultimately set in the fragment shader but can be determined in either shader or in the JS application

  - Application: pass to vertex shader as a vertex attribute

    - Or as a uniform variable (next example)
  - Vertex shader: pass to fragment shader as a varying variable
  - Fragment: can alter via shader code

Custom Solid-Colored Triangle

- Let's make a custom solid-colored triangle
- Using what we have already seen how would we do this?

Custom Solid-Colored Triangle

- Let's make a custom solid-colored triangle
- Using what we have already seen how would we do this?

  - Using `gl.bufferSubData()` is wasteful as the model gets larger

  - Instead we can use a ***`uniform`*** variable
    - Send just one piece of data to all vertex and fragment shaders at once
    - It is *uniform* across all shaders

Using Uniform Variables

- Shader: Add `uniform` variable to one (or both) shaders as necessary
  - Remember: our convention will be to prefix with `u`
- JavaScript:
  - Get uniform variable location (basically like getting attribute location except it's a uniform)
  - Set the uniform variable:

```
gl.uniform4fv(uni_loc, x)
```

    - Where `uni_loc` is the location previously looked up and `x` is a `Float32Array` (in this case: [red, green, blue, alpha])

OpenGL is a C library

- C does not allow two functions with the same name even if they take different types or numbers of arguments
  - WebGL follows the same names from OpenGL
- Example: send a value to a shader:
  - `gl.uniform3f`
  - `gl.uniform2f`
  - `gl.uniform3dv`

WebGL Function Naming

- For functions, the name is the same as the OpenGL name except the `gl` prefix is dropped, it is a method of the WebGL context, and the first uppercase letter is made lowercase
- For constants the `GL_` prefix is dropped and it is a property of the WebGL context
- Example:
  - OpenGL: `glEnable(GL_DEPTH_TEST)`
    WebGL: `gl.enable(gl.DEPTH_TEST)`

WebGL Function Naming

WebGL Context Variable

Function Name

Dimension (1-4)

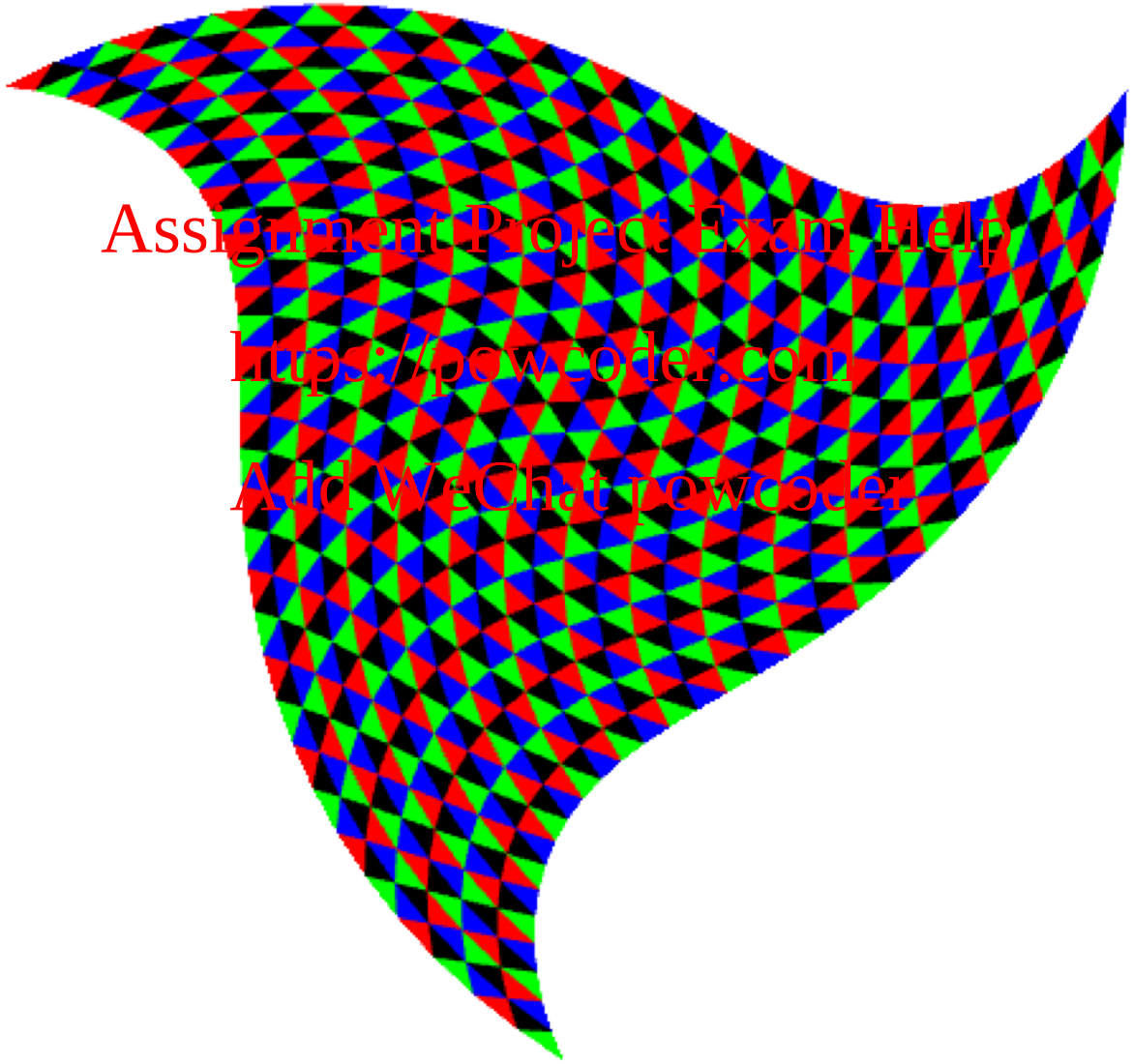

```
gl.uniform3f(loc, x, y, z)
```

Variable Type

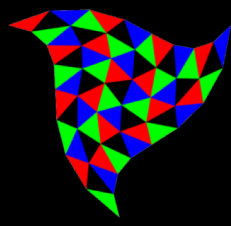
(f = float, i = int, ui = unsigned int)

If this ends with a **v** then the arguments are given as a typed array

Exercise: Twist Transformation



Exercise: Twist Transformation

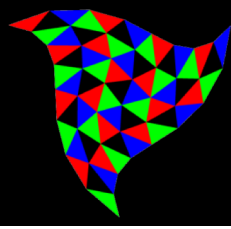


- Tessellated triangle (based on Sierpinski's triangle) except main triangle is an equilateral triangle with the center at the origin:

$$(0,1), \left(-\sqrt{3}/2\,, -1/2\right), \left(-\sqrt{3}/2\,, -1/2\right)$$

- Now we want to apply a transform in the vertex shader to "twist" the triangle by rotating all of the vertices around the origin based on their distance from the origin

Exercise: Twist Transformation



- How do you rotate a 2D point around the origin?

$$x' = x\cos(\theta) - y\sin(\theta)$$

$$y' = x\sin(\theta) + y\cos(\theta)$$

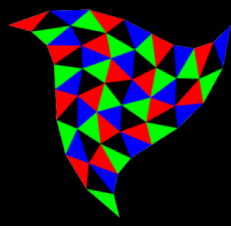
- How do you make the amount rotated based on the distance from the origin?

$$x' = x\cos(d\theta) - y\sin(d\theta)$$

$$y' = x\sin(d\theta) + y\cos(d\theta)$$

$$d = \sqrt{x^2 + y^2}$$

Exercise: Twist Transformation



- Modify the vertex shader to perform the following transformation:

$$x' = x\cos(d\theta) - y\sin(d\theta)$$

$$y' = x\sin(d\theta) + y\cos(d\theta)$$

$$d = \sqrt{x^2 + y^2}$$

- Recommendation: try setting θ variable to different values (is it $\pi/3$ initially)

Exercise: Twist Click

- Start the twisting example with `theta = 0`
- Every time the user clicks the `canvas` the `theta` will be set to $d\pi$ where d is the distance of the click from the $(0,0)$ within the `canvas`
- To do this we need to make the `theta` in the vertex shader updatable from JS

What type of variable should it be?

Instancing

Instancing

- What if you had to draw several of the same exact objects such as:
  - Balls, falling squares, walls of mazes, checkers, …
- Should you load all of them onto the GPU separately? Or should you just load one and tell it to draw multiple times?
  - Depends on how complex the object is, what needs to be changed between each instance of the object, etc

Instancing

- In general complex 3D models will be centered at the origin with a pre-determined size and orientation
  - Easy to do with simple models as well – circle formula is a piece of cake at origin
- We then apply an instance transform to the vertices to get it where we want it, the size we want it, and the orientation we want it
  - We draw multiple copies of the same instance
    - Vertex data on GPU only once
    - Each uses a different transformation uniform(s)

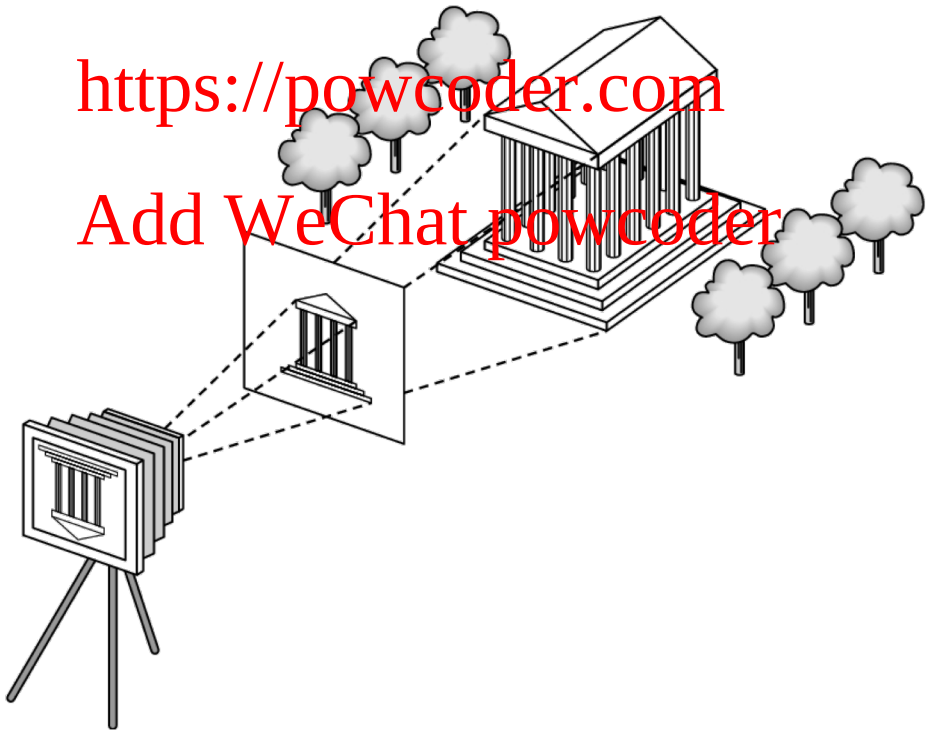
Instancing

- Uniforms are perfect for instancing!
- Can adjust position, color, etc with a uniform then draw it
  - Then adjust position, color, etc again and call draw again (without clearing) to get another instance

World Setup

Synthetic Camera Model

- Need to setup a camera, lights, objects and their materials

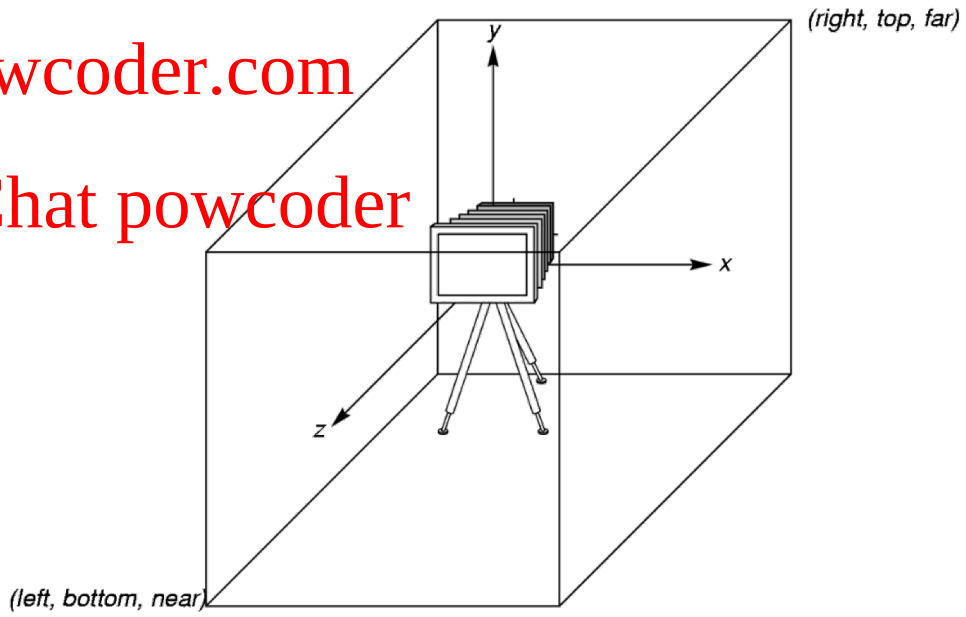


Camera and Light Properties

- Camera:
  - **Six** degrees of freedom
    - (x,y,z) position of center of lens
    - (pitch,roll,yaw) orientation of camera
  - Lens, size of film, orientation of film plane
    - Perspective projection (real world)
    - Parallel projection (objects do not shrink with distance)
- Lights:
  - Location and possibly orientation
  - Type: Point, distributed (e.g. ambient), spot
  - Color

Default Camera and View

- Default camera is placed at the origin (0,0,0) in object coordinates and points in the negative z direction
- The default viewing volume is a box centered at the origin with sides of length 2

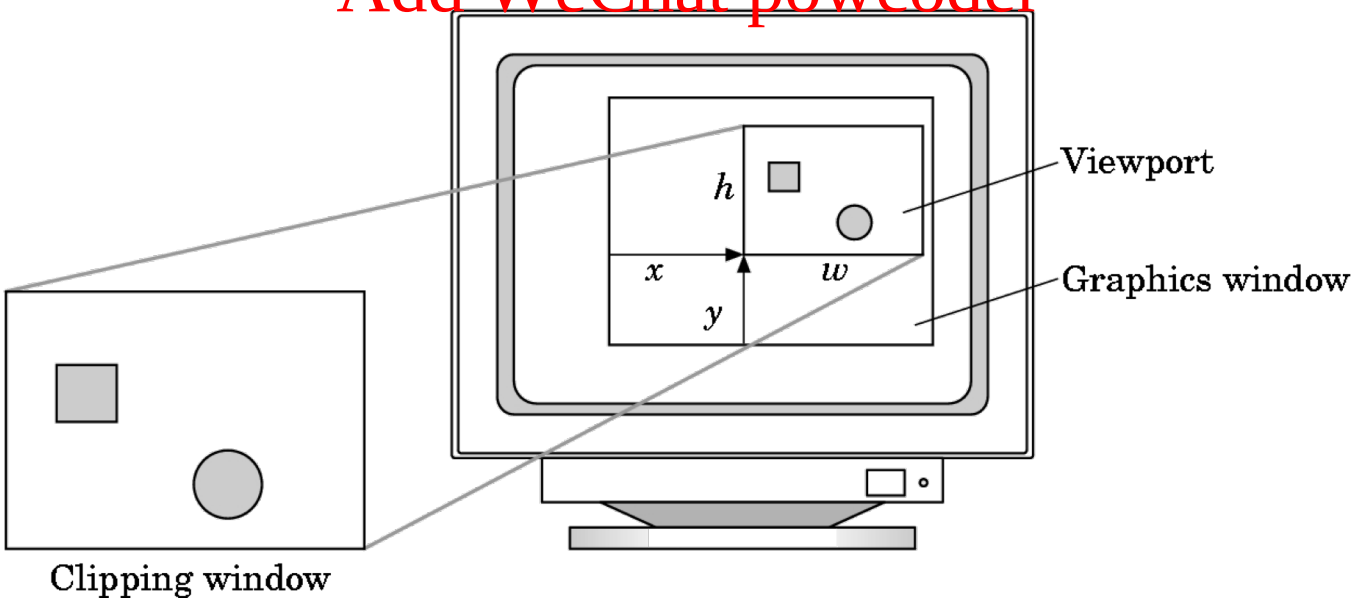


Viewport

- Do not have to output to the entire canvas
- Output values given with


```
gl.viewport(x,y,w,h)
```


  - Values are in pixels / window coordinates



Object and Material Properties

- Objects are built from primitives including:
  - Points (0D)
  - Line segments (1D)
  - **Triangles (2D)**
  - Surfaces and curves
    - Quadrics and parametric polynomials
  - All primitives are defined by *vertices* (locations)
- Materials:
  - Absorption: what colors are absorbed/reflected?
  - Scattering: when light reflects off the surface does it scatter (diffuse) or not (specular/mirror)?