

“A Big GUI Mess”

or: How to Make Web Pages Respond to Mouse, Keyboard & Browser

Jack Tumblin

Northwestern Univ COMP_SCI 351-1
Fall 2021

How Do we Get CONTROL ?!?!

From Page 1; >4 unfamiliar languages:

HTML-5, and its new ‘Canvas’ object (?)

JavaScript (or ‘ECMAScript’ – WHUT?)

WebGL

(the big mystical 3D drawing API)

GLSL

(the big mystical parallel language WITHIN WebGL ?!??!)

- + NO IDE, no console, no cmd line?
 - + Trivial ‘connect the dots’ ideas,
 - + Awkward on-screen viewing
 - +trivial, fixed, hard-coded images
 - +libraries hide so many details...

FAR more ‘items in use’ than ‘items explained’ !

There's a Method here...

- Early exposure – see where we're going,
THEN learn how to get there;

<https://mapbox.github.io/webgl-wind/demo/> or

<https://webglsamples.org/dynamic-cubemap/dynamic-cubemap.html> or

https://threejs.org/examples/webgl_animation_cloth.html

https://threejs.org/examples/#webgl_geometry_teapot

fired up your curiosity?

- Surprised You A Little?

GOOD! Now let's get a bit more sensible...

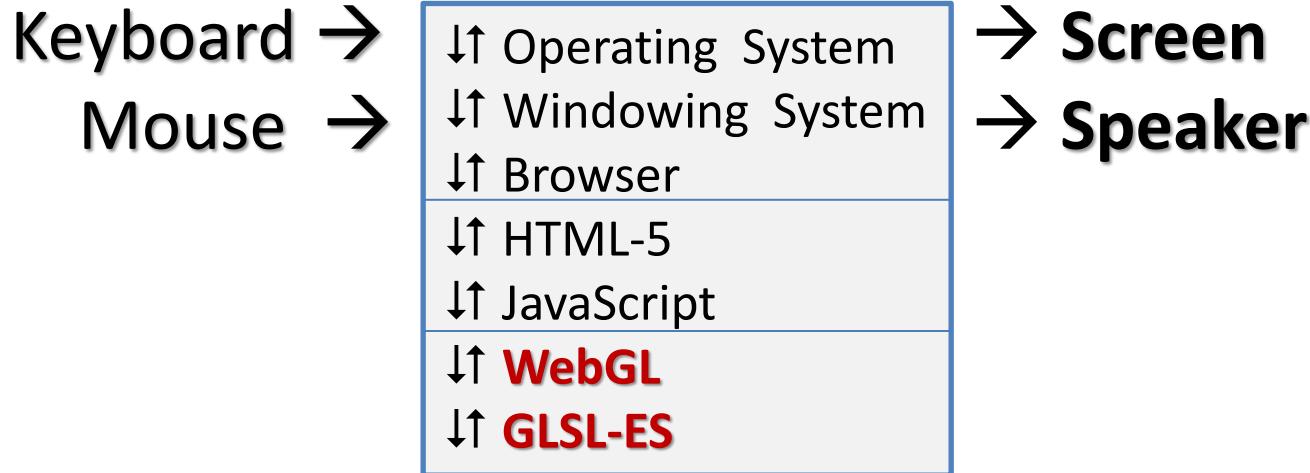
Setup For WebGL Programming

Let's Serialize it a bit;
get some basics that the book left out...

Setup: How to use Textbook's Starter Code

- Open Firefox
- Open Text Editor
- Drag-and-drop HTML file → Firefox
 - Right-click(CMD click) →'inspect' (opens Chrome Dev Tools)
 - Select either 'console' or 'source' tags
- Open .JS (JavaScript) → in Text Editor

Control is the Goal



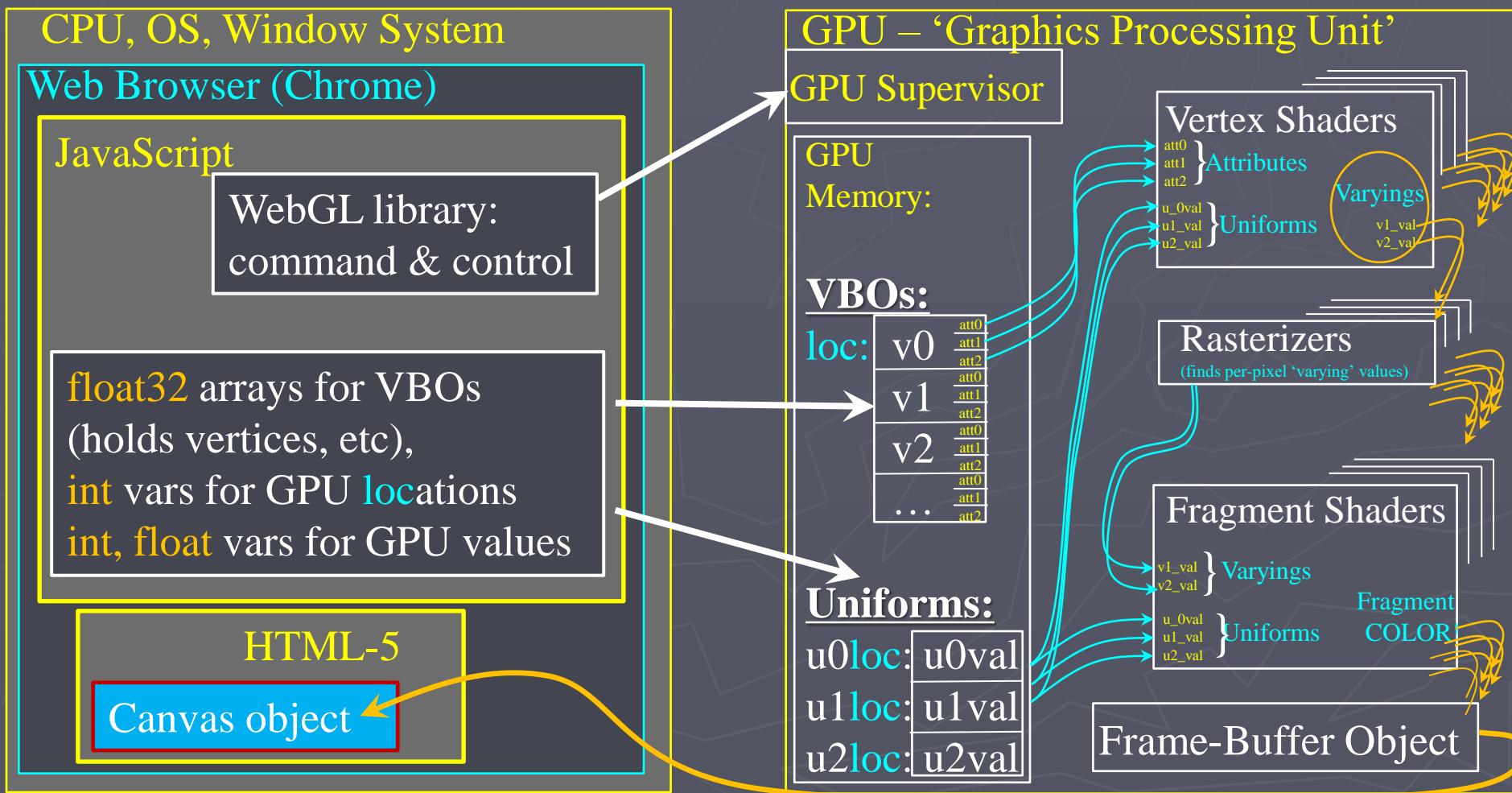
Purpose of WebGL?

Send tasks to your Graphics Programming Unit (GPU)

Purpose of GLSL?

Compute all graphical values (in a Sea of Processors)

- WebGL: sends tasks to GPU
- GPU: executes tasks, flags errors
- WebGL: (optional) read GPU flags



Let's Get Webpage Control first!

Keyboard →

Mouse →

↓↑ Operating System
↓↑ Windowing System
↓↑ Browser
↓↑ HTML-5
↓↑ JavaScript
↓↑ WebGL
↓↑ GLSL-ES

→ Screen

→ Speaker

Part 1: HTML basics:

HTML tags: title,
comments, head, body,
fixed plain text,
headings, paragraphs,
buttons, dynamic text,
& using javascript files.

Part 2: HTML + JavaScript:

printf-like console.log(),
How to capture keystrokes;
How to capture mouse
moves and clicks.

(please take, use this code!)

Try all ‘GUImess’ Starter Code

In HTML file, let's read and explain a bit

- Everything in your browser-based program (Webpage) **starts here**.
- ‘Tags’: keywords inside angle-braces:
 - start: <keyword>
 - end: <\keyword>

MOST-BASIC HTML 5:

From 'GUImess' Starter code:

ch05/5.01a.MultiAttributeSize.html:

```
<!DOCTYPE html>
<html lang="en"> ←
  <head>
    <meta charset="utf-8" />
    <title>Draw points with specified size</title>
  </head>
  Hello. Is this Ickophone erking?

  <body onload="main()">
    <canvas id="webgl" width="400" height="400">
      Please use a browser that supports "canvas"
    </canvas>

    <script src="../lib/webgl-utils.js"></script>
    <script src="../lib/webgl-debug.js"></script>
    <script src="../lib/cuong-utils.js"></script>
    <script src="../lib/cuong-matrix.js"></script>
    <script src="MultiAttributeSize.js"></script>
  </body>
</html> ←
```

XML style:

- Plain-text only
- Blocks of data enclosed within begin/end 'tags'

Begin **End**

e.g. <html> </html>
 <head> </head>
 <title> </title>
 <body> </body>

HTML's internal COMMENTS

From 'GUImess' Starter code:
within **5.04jt.ColoredMultiObject.html**:

```
<body onload="main()">
  <canvas id="webgl" width="500" height="500">
    Please use a browser that supports "canvas"
  </canvas>
  <p>
```

```
    <button type="button" onclick="spinDown()">Spin << </button>
    <button type="button" onclick="runStop()"> Run/Stop</button>
    <button type="button" onclick="spinUp()">Spin >> </button>
```

<! these commands create a push-button with labels ++ CCW and -- CCW.
When users click the button, call the function name held in 'onclick'>

```
</p>
<script src="../lib/webgl-utils.js"></script>
<script src="../lib/webgl-debug.js"></script>
<script src="../lib/cuon-utils.js"></script>
<script src="../lib/cuon-matrix-mod.js"></script>
<script src="ColoredMultiObject.js"></script>
</body>
```

Tag: **<! >**

- IGNORED by the browser
- Lets you *explain* your *intent*

Fixed, Plain TEXT:

From 'GUImess' Starter code:

5.01a.MultiAttributeSize.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Draw points with specified size</title>
  </head>
  Hello. Is this Ickrophone erking?
```

```
<body onload="main()">
  <canvas id="webgl" width="400" height="400">
    Please use a browser that supports "canvas"
  </canvas>

  <script src="../../lib/webgl-utils.js"></script>
  <script src="../../lib/webgl-debug.js"></script>
  <script src="../../lib/cuong-utils.js"></script>
  <script src="../../lib/cuong-matrix.js"></script>
  <script src="MultiAttributeSize.js"></script>
</body>
</html>
```

Just 2 blocks:

<head>

- Sets webpage-title for browser

<body>

- Sets contents of that page

Text Before <body>?
appears on-screen
BEFORE
the 'body' contents

Plain TEXT: Headings, Paragraphs

Very basic (ugly) ‘Heading’ and ‘Paragraph’ tags give on-screen text that wraps and flows to fill browser window width:



```
<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>
</html>
```

Text-Labeled Buttons (1)

From 'GUImess' Starter code:

within **ch05/5.04jt.ColoredMultiObject.html**:

```
<body onload="main()">
  <canvas id="webgl" width="500" height="500">
    Please use a browser that supports "canvas"
  </canvas>
  <p>
    <button type="button" onclick="spinDown()>Spin << </button>
    <button type="button" onclick="runStop()> Run/Stop</button>
    <button type="button" onclick="spinUp()>Spin >> </button>
```



```
  <! these commands create a push-button with labels ++ CCW and -- CCW.
  When users click the button, call the function name held in 'onclick'>
</p>
<script src="../lib/webgl-utils.js"></script>
<script src="../lib/webgl-debug.js"></script>
<script src="../lib/cuon-utils.js"></script>
<script src="../lib/cuon-matrix-mod.js"></script>
<script src="ColoredMultiObject.js"></script>
</body>
```

Text-Labeled Buttons (2)

From 'GUImess' Starter code:

within **ch05/5.04jt.ColoredMultiObject.html**:

```
<body onload="main()">
  <canvas id="webgl" width="500" height="500">
    Please use a browser that supports "canvas"
  </canvas>
  <p>
    <button type="button" onclick="spinDown()>Spin << </button>
    <button type="button" onclick="runStop()> Run/Stop</button>
    <button type="button" onclick="spinUp()>Spin >> </button>

    <! these commands create a push-button with labels ++ CCW and -- CCW.
      When users click the button, call the function name held in 'onclick'>
  </p>
  <script src="../lib/webgl-utils.js"></script>
  <script src="../lib/webgl-debug.js"></script>
  <script src="../lib/cuong-utils.js"></script>
  <script src="../lib/cuong-matrix-mod.js"></script>
  <script src="ColoredMultiObject.js"></script>
</body>
```

Tag: **<button ... >**

- Calls this JavaScript Function when clicked
- Shows this text label

CHANGEABLE Text / HTML (1)

From 'GUImess' Starter code:

within **ch05/5.04jt.ControlMulti.html**

Text you can update? EASY!

- Make a 'div' element
(‘division’ of a page)
- Create its identifier (id= ...)
- Then in your JavaScript code...

CurrentAngle= -124.37999999999751
New Current Angle (-180 < x < +180 deg):

Open the JavaScript Console please.

```
<!-- Make a 'div' element to hold changeable HTML made in our JavaScript file;  
     (where? Look inside main(), inside its 'tick()' function, where we  
      display the 'current angle' value.  
-->  
<div id= CurAngleDisplay></div>
```

CHANGEABLE Text / HTML (2)

From ‘GUImess’ Starter code:

within **ch05/5.04jt.ControlMulti.html**

Text continually updates? EASY!

- Make a ‘div’ element
 (‘division’ of a page)
- Create its identifier (id= ...)
- Then in your JavaScript code use that ID to “inject new HTML code”:
 ‘innerHTML’ object *re-draws* our ‘div’ contents. (displays value)



CurrentAngle= -124.37999999999751
New Current Angle (-180 < x < +180 deg): Submit

Open the JavaScript Console please.

```
138 // Show some always-changing text in the webpage : ↴
139 // --find the HTML element called 'CurAngleDisplay' in our HTML page, ↴
140 // (a <div> element placed just after our WebGL 'canvas' element) ↴
141 // and replace it's internal HTML commands (if any) with some ↴
142 // on-screen text that reports our current angle value: ↴
143 // --HINT: don't confuse 'getElementById()' and 'getElementsByID()' ↴
144 document.getElementById('CurAngleDisplay').innerHTML= ↴
145 'CurrentAngle= '+currentAngle; ↴
```

console.log() -- ‘print-to-console’

- Run **ch05/5.04jt.ColoredMultiObject**
- Open Console. What happens?
- How can you stop that?!?! → it's in the JavaScript!
 - try ‘clear console’ button (upper left )
 - try to find the word ‘console’ in JS code ...
what statement is the cause? How does it work?
comment it?
- Modify it! Learn it! Google it for details!
- Add ‘console’ calls to the ‘First-Day’ starter code...
What happens? Can you make it repeat endlessly?

Keyboard Input: Try it First...

Run **ch05/5.04jt.ControlMulti.html**

- type on keyboard →
- see ‘key-codes’ appear in webpage.

Open Console. ([How? \(right-click or Cmd-click\) → Inspect... → Console](#))

- Is the ‘canvas’ too big? shrink it!
(suggestion: 500x500 → 300x300 in your HTML file).
- Type on Keyboard →
- see much more keyboard data appear in console
- hit ‘clear console’ button (upper left ); type...



How Does it Work? (regions...)

Each HTML ‘element’ claims an on-screen region:

- Firefox Dev Tools: select ‘Elements’ tab
(How? (right-click or Cmd-click) → Inspect... → Console)
 - click some HTML code: see its on-screen region highlighted!
 - try it! find the ‘element’ for the WebGL drawing...)

Any mouse or keyboard action generates an ‘**event**’
(a queued, widely-shared object; an input/output item).

- Many HTML-5 ‘elements’ (including the ‘canvas’) let you write your own ‘**event handler**’, a function called when it receives an ‘event’ in its on-screen region.
- (Look inside ch05/**ControlMulti.js**, in **main()** fcn, line 114 ...)

How Does it Work? (register...)

inside ‘ch05/**ControlMulti.js**’ ,in **main()** fcn, line 114 ...

“Register” your event-handler function ==

specify which function() to call on which event, and
specify its arguments, too.

- Only our ‘canvas’ object
(where we will draw WebGL results)
will respond to mouse events;
thus ‘register’ them with our ‘canvas’ object.

at line 127, 131:

- The ENTIRE HTML-5 webpage or ‘WINDOW’ responds to
keyboard events: thus register them with ‘window’ object.

How Does it Work? (handlers...)

'**ControlMulti.js**' **Mouse** event-handler functions
begin at text line number 466:

- **myMouseDown(), myMouseUp()** finds mouse location within the normalized 'canvas': $-1 \leq x, y \leq +1$, saved in a global variable for everyone (see line 76-83). (note messy pixel-coords to canvas coord. calculations; uncomment `console.log()` call to see what it does)
 - **myMouseMove()** finds distance moved, also normalized. (See line 76-83).
- main(), draw()**, and other fcn use those global variables to make WebGL drawings inside the HTML5 'canvas' element.

How Does it Work? (handlers...)

‘**ControlMulti.js**’ **Keyboard** event-handler funcs at line 591:

- **myKeyDown(), myKeyUp()** called when ANY key pressed or released, even non-ASCII keys (F1...F9,PgUp,Shift,Ctrl,etc). Often filled by a giant ‘switch’ statement, with one ‘case’ for each non-ASCII keyboard response.

HINT: most EECS 351-1 projects need only arrow keys and single-letter controls (e.g. ‘c’ key changes colors...)

Want more Than This?

- **Try it! Apply GUI methods to any Starter Code...**
- Excellent, Authoritative JavaScript Tutorial:
<http://www.w3schools.com/js/>
- ‘21 Javascript Parts I Struggle to Remember’
http://codylindley.com/techpro/2013_11_11_21-javascript-parts-i-struggle/
- Index / Reference of all HTML-5 tags:
<http://www.w3schools.com/tags/default.asp>

Want more Than This?

- Excellent HTML(5) Tutorial Site by W3C
<http://www.w3schools.com/html/default.asp>
- Index / Reference of all HTML-5 tags:
<http://www.w3schools.com/tags/default.asp>

Try Google's `dat.gui` library...

These essential GUI controls
are all very basic, old, & simple.

And ugly.

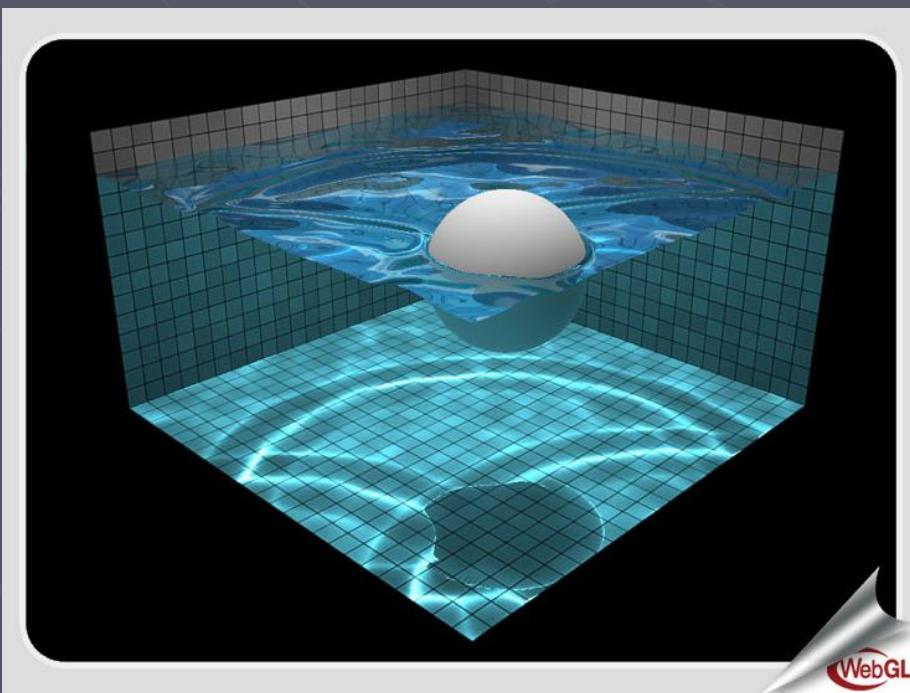
Many other, prettier-but-more-complicated GUIs
are freely available online, such as `dat.gui` :

<https://workshop.chromeexperiments.com/examples/gui/#1--Basic-Usage>

you're welcome to use this in your projects,
and/or any other GUI library you like.

END

WebGL == GPU Commands in JavaScript



<http://madebyevan.com/webgl-water/>

Jack Tumblin, CS 351-1
Northwestern University

WebGL: My First Impression



WebGL: What is it, exactly?

- Uniform set of **JavaScript commands**

formally, an API spec. (for GPU drivers to implement)

`g1.fcn()`

- Organized as one giant **state-machine** to
many sensible default settings you can change + input buffers, handles

- Control a *!!fast!!* '**2D rasterizing engine**'
a sea of FP procs, distrib'd memory + vast fixed-purpose hardware

- To draw colored **dots, lines, or triangles**
the only drawing primitives, but extremely flexible & programmable

- into '**frame buffers**' (2D array of pixels)

'pixel': set of numerical values that describe contents of a picture in a small local neighborhood. Besides 8-bit RGB, webGL 'pixels' can hold any desired fixed or float values: standard (color, stencil masks, z-depth, ID#s, spectrum, velocity, ...) or your own custom values (!)

GPU: MUCH more powerful, but (currently) MUCH less flexible than CPU

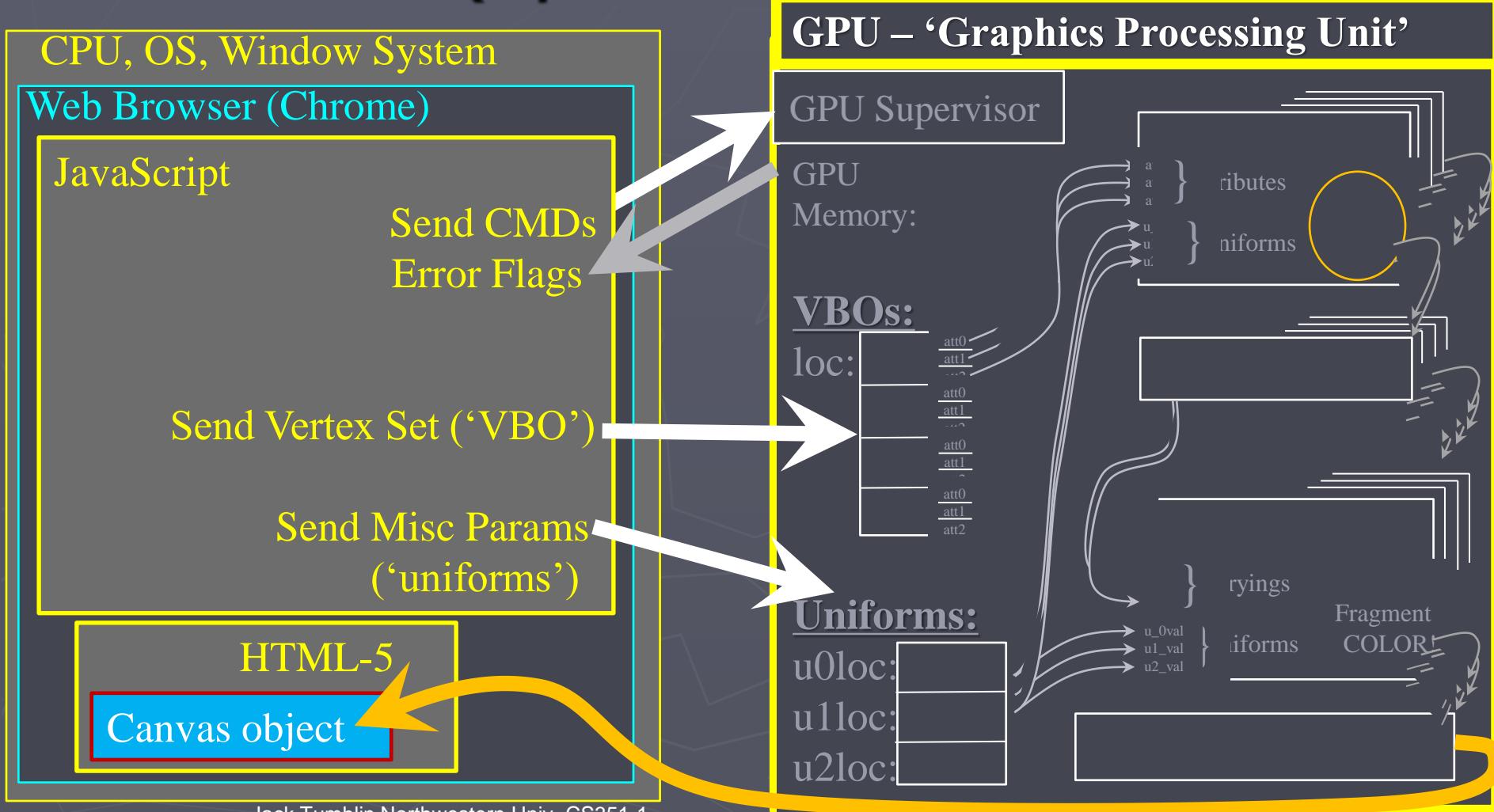
Modern GPU ('Graphics Processing Unit') == (! wow !)

- ▶ **Distributed Memory:** Vast arrays of fast, wide, parallel multi-port, dyn. reconfig., cached memory
- ▶ **Distributed Data Paths:** Vast hyper-grids of wide, fast dynamically-reconfigured bidirectional busses
- ▶ **Distributed ALUs:** Vast arrays of dynamically reconfigured fast floating-point SIMD processors
- ▶ **Distributed 'Rasterizers':** automatically reconfigured arrays of linear interpolators for individual pixels (LERPs for data & address; perspective divide for texture...)
Rasterizers 'connect the dots' across vertex values

--**WebGL**: sends tasks to GPU

--**GPU**: executes tasks, flags errors

--**WebGL**: (optional) **reads** GPU flags



WebGL: Mostly 'API for the GPU'

Make JavaScript calls to WebGL functions to:

- ▶ Load, Compile, Link, Run Shaders on GPU
 - 'Shader' == one SIMD Program, multiple CPUs
 - GLSL == the C-like SIMD 'shading language'
- ▶ Transfers Data Sets into the GPU;
 - 'VBOs' (Vertex Buffer Objects) == dynamically allocated hunks of GPU memory that will hold sets of vertices you transferred from JavaScript
 - 'uniforms' == individual user-defined parameters
- ▶ DRAWS Pictures: Shaders process Data Sets

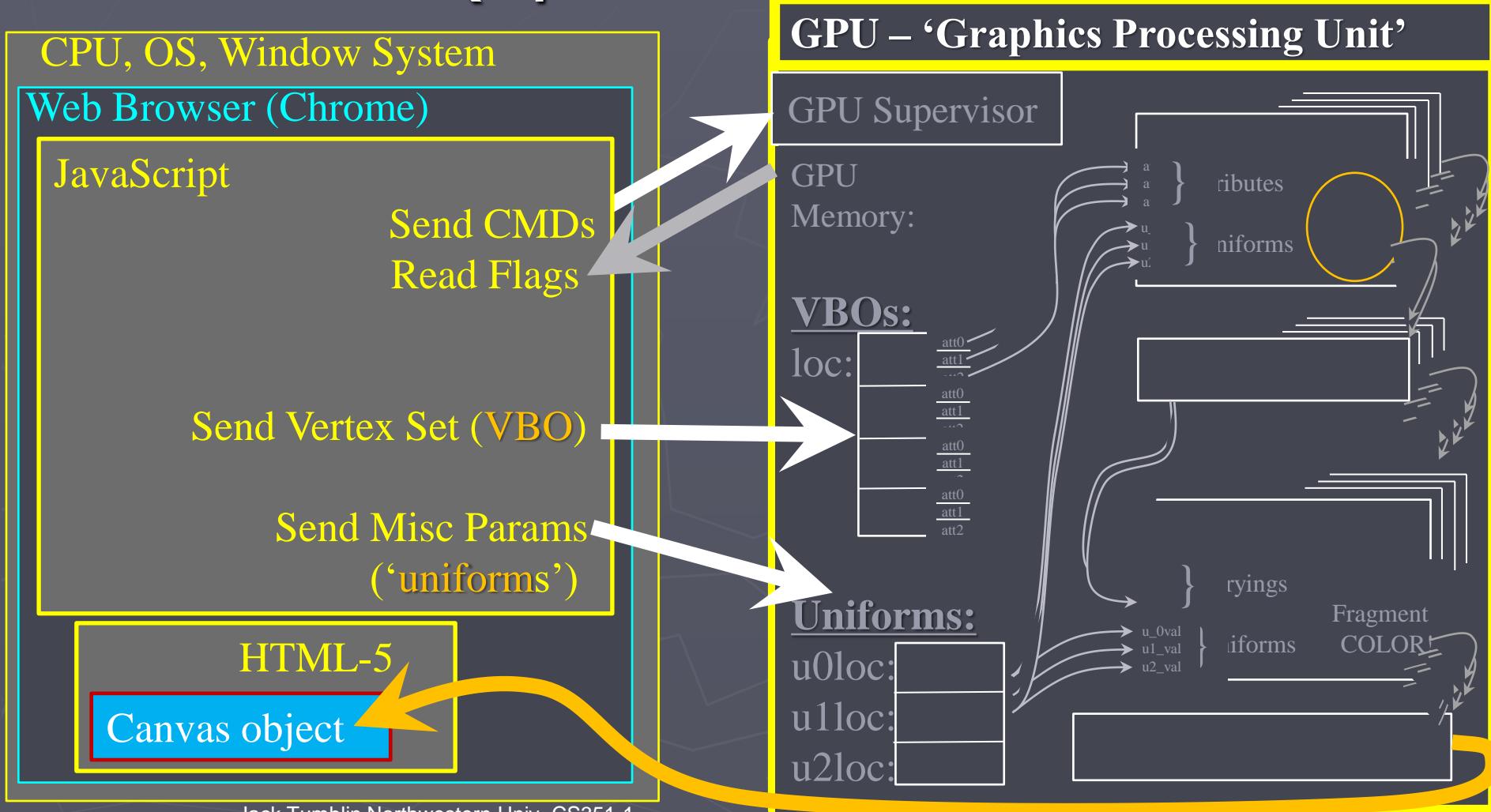
JARGON REVIEW:

- ▶ 'Vertex' == a set of numerical 'attributes' that describe one point on one surface
- ▶ 'Attribute' == one scalar, vector, or matrix that describes one property of a vertex (e.g. position, color, normal vector, material)
- ▶ 'Vertex Buffer Object' (or 'VBO')
A read-only, GPU-resident array of vertices, held in memory suitable for SIMD access.
- ▶ 'Shaders: SIMD Floating-pt Processors that execute compiled GLSL 'Vertex Shader' code and compiled GLSL 'Fragment Shader' code
(Yeah, yeah, yeah – shader = hardware? Shader = code? bad naming... 'varying' is worse!)

--**WebGL**: sends tasks to GPU

--**GPU**: executes tasks, flags errors

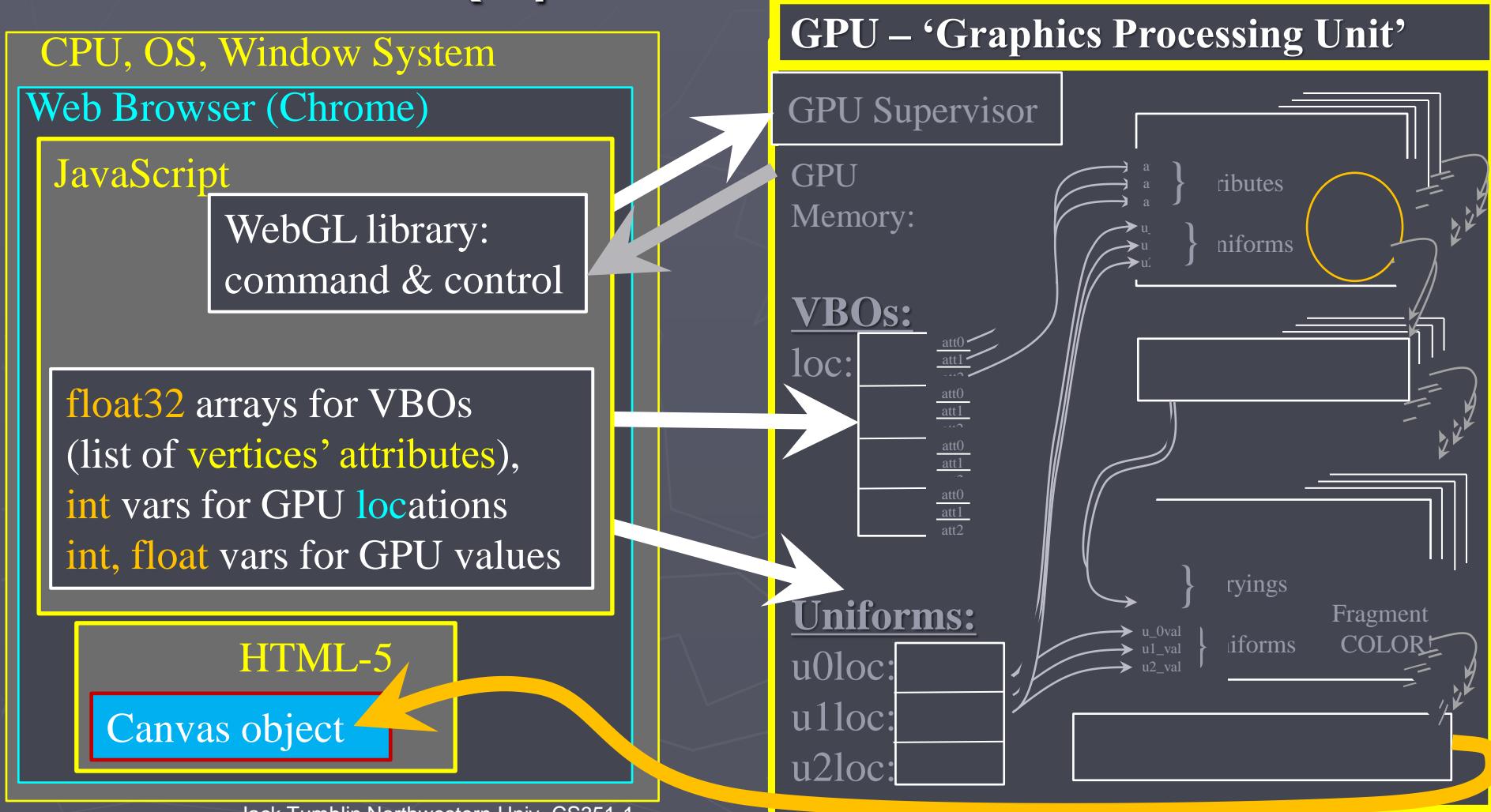
--**WebGL**: (optional) **reads** GPU flags



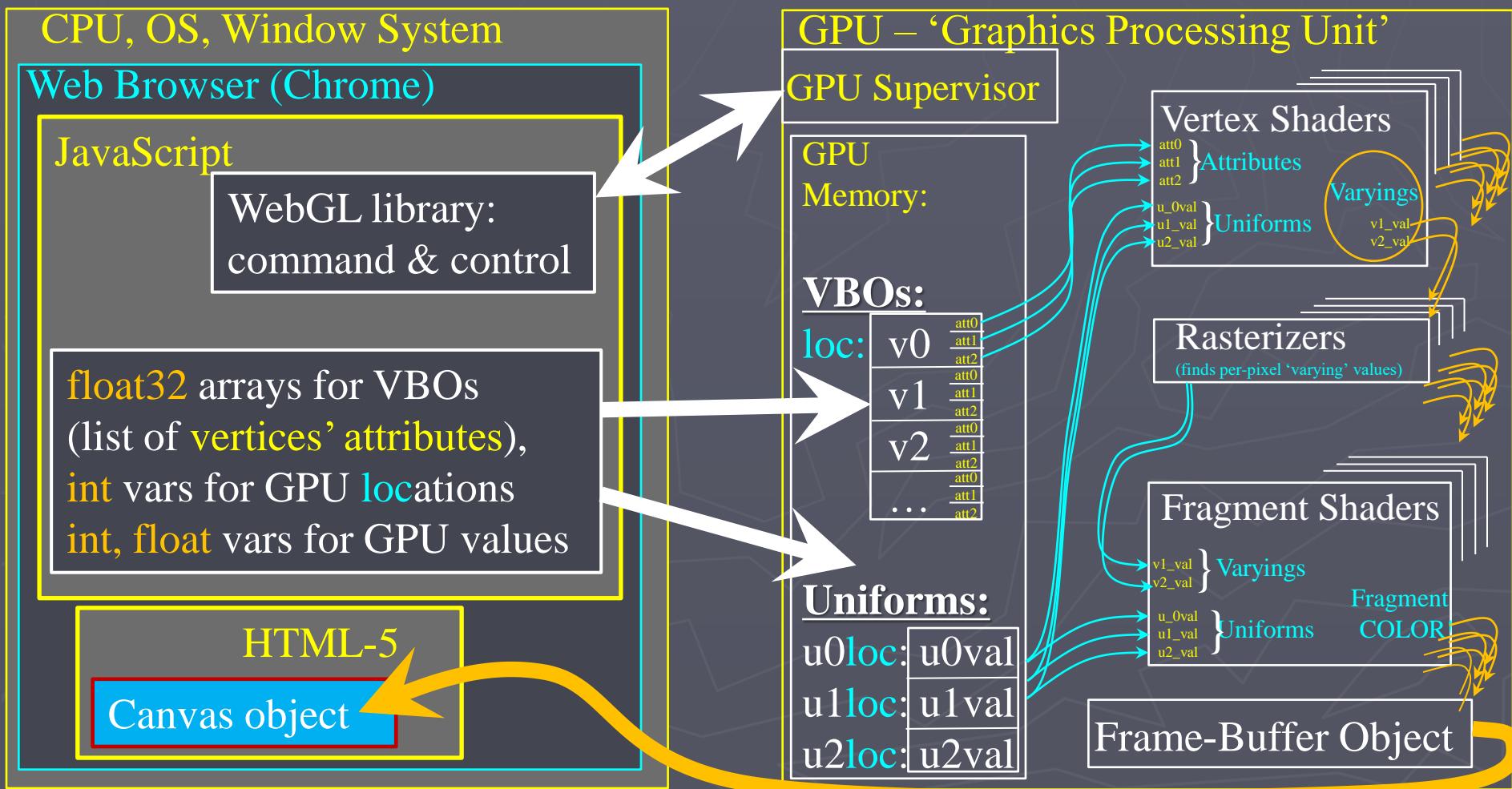
--**WebGL**: sends tasks to GPU

--**GPU**: executes tasks, flags errors

--**WebGL**: (optional) **reads** GPU flags



- WebGL: sends tasks to GPU
- GPU: executes tasks, flags errors
- WebGL: (optional) reads GPU flags



GLSL Programmable Shaders: What?

- **Vertex Shader:**

One program executed for each & every vertex; computes

any & all imaginably useful vertex values

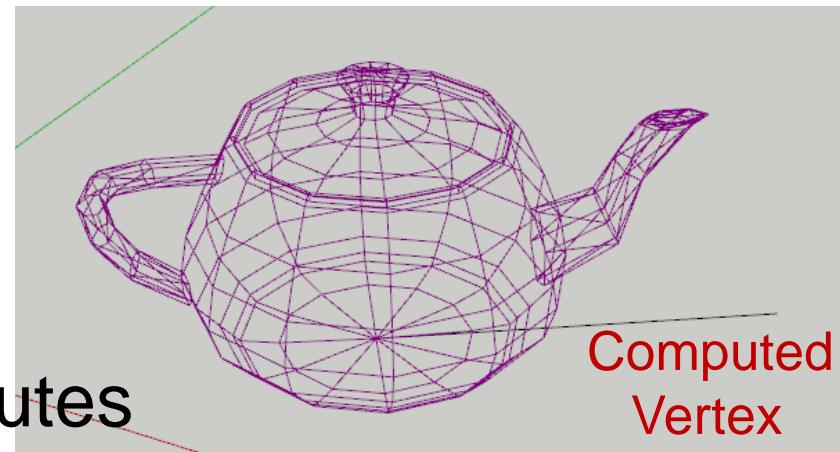
(location, angle, distance, color, etc. for cameras, lights, materials, textures, etc)

- **Fragment Shader:**

One program executed for each & every pixel that we draw between vertices.

Computes pixel's final color, plus any other useful per-pixel values

(depth, texture mix, ID, etc)



Computed
Vertex
Values



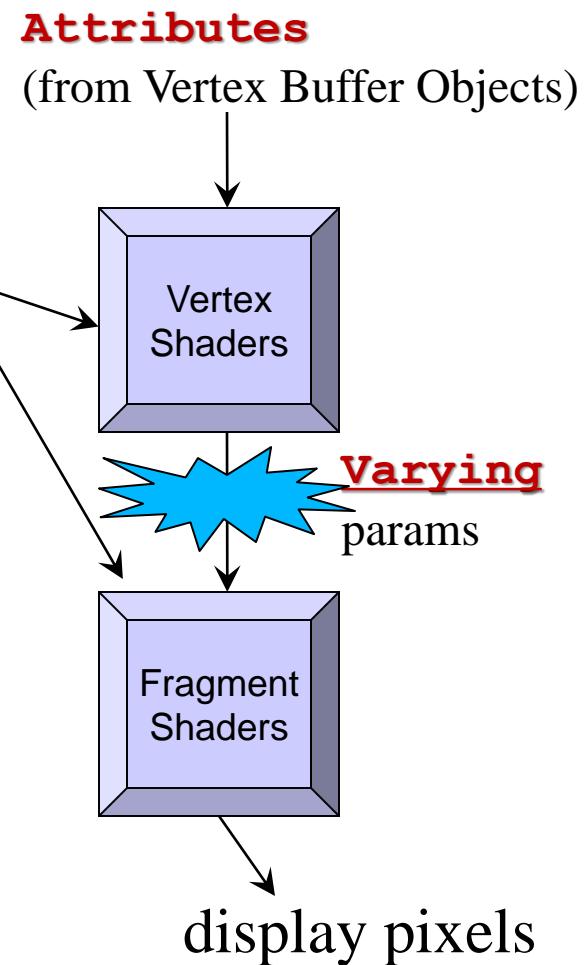
Computed
Fragment
Values



SIMPLIFIED: How Shader Programs Communicate

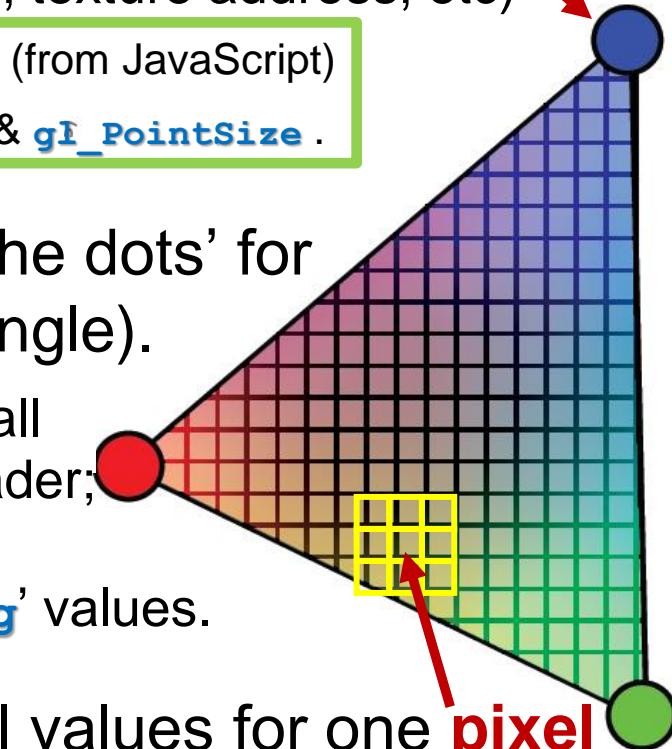
Only 3 ways for your JavaScript code to send data **into** the GPU's shader programs
(and no way to retrieve that data!)

- **Uniform** parameters
 - Set before each drawing command
 - Ex: ModelMatrix
- **Attribute** parameters
 - Set per vertex from VBO contents
 - Ex: position, color, surface normal, ...
- **Varying** parameters
 - Passed from Vertex Shaders to rasterizer, which interpolates per-pixel values for fragment shaders
 - Ex: interpolated colors



GLSL Programmable Shaders: What?

- Each GLSL **Vertex Shader** finds all values for one **vertex**:
(3D transformed position, color, normals, depth, texture address, etc)
 - **Inputs:** ‘`attribute`’ values (from VBO), ‘`uniform`’ vars (from JavaScript)
 - **Outputs:** all ‘`varying`’ vars; built-in vars `gl_Position` & `gl_PointSize`.
- GPU ‘**Rasterizer**’ hardware ‘connects the dots’ for the drawing-primitive (point, line, or triangle).
‘Rasterizers’ interpolates between any / all ‘`varying`’ variables set up by Vertex Shader; they supply each Fragment Shader with separate, smoothly-interpolated ‘`varying`’ values.
- Each GLSL **Fragment Shader** finds all values for one **pixel**
(color, mostly. Also (rarely) depth, textures, lighting, reflection, material...)
 - **Inputs:** ‘`uniform`’ vars (from JS) ‘`varying`’ values (from Rasterizers), and also a few built-in vars: (`gl_PointCoord`, `gl_FragCoord`, `gl_FrontFacing`, etc.)
 - **Outputs:** built-in vars `gl_fragColor` and `gl_fragData`



GLSL Variables Qualifier: **uniform**

- Identifies variables that pass values **from** your JS program **into** your shaders
Both of them! Vertex shader & Fragment shader too!
(e.g. position, orientation, select/unselect...)
- **uniform** variable: values stay **constant** during each WebGL drawing command
- **Can** be changed in JS program between each **drawArray()** or **drawElements()** call
- Shaders **Cannot** change them – input only!

GLSL Variables Qualifier: **attribute**

- **Only** for inputs *from* VBO → Vertex Shader
(inaccessible to Fragment Shader)
- Each V.Shader gets values from **one** vertex supplied from the currently-bound VBO
- **Read-only:** V. Shader can't set VBO values
- NO built-in Vertex-Shader attributes. **None!**
- ALL must connect to VBO &
ALL must affect Shader Result
(or very aggressive GLSL optimizing compiler *ignores* them!)

attribute vec3 a_position; // (always!)

GLSL Variable Qualifier:

varying

- In Vertex Shader:
 - Defines an OUTPUT variable to get ‘rasterized’: INTERPOLATES to all pixels between vertices!
Each fragment shader (& thus each pixel) gets a different, smoothly-varied value!
- Fragment Shader must use them non-trivially as it computes pixel color.
(GLSL compiler **optimizes aggressively**;
unused `varying` vars will vanish!
Causes cryptic ‘undefined variable’ errors, etc)

WebGL Setup(1): HTML file

[STARTER CODE? Ch05/ColoredMultiObject](#)

► Inside HTML <body>:

- Create 'canvas' object:

```
<canvas id="HTML5_canvas" width="500" height="500">  
Please use a browser that supports HTML-5 "canvas" objects  
</canvas>
```

- Create GUI: text, buttons, edit boxes, etc.

```
<button type="button" onclick= "spinDown()"> Spin << </button>  
<button type="button" onclick= "runStop()"> Run/Stop </button>  
<button type="button" onclick= "spinUp()"> Spin >> </button>
```

- Load any WebGL utility libraries we want to use:

```
<script src="../lib/webgl-utils.js" > </script>  
<script src="../lib/webgl-debug.js" > </script>  
<script src="../lib/cuong-utils.js" > </script>  
<script src="../lib/cuong-matrix-mod.js"> </script>
```

- Load JavaScript program file (contains **main()**)

```
<script src="ColoredMultiObject.js" > </script>
```

WebGL Setup(2): in JS file

STARTER CODE? See: Day2→ch05→ColoredMultiObject

- ▶ In our JavaScript file, in **main()** :

- Connect WebGL obj (**gl**) to HTML-5 canvas obj:

```
var myCanvas = document.getElementById('HTML5_canvas');
var gl = getWebGLContext(myCanvas);
// gl object's member functions now include all webGL fcns!
```

- Set WebGL's own screen-clearing color + misc

```
gl.clearColor(0.3, 0.3, 0.3, 1.0); // for muted gray bkgnd
// Color-choosing: 0.0 <= Red, Green, Blue, Alpha <= 1.0
// (?What's 'alpha'? - transparency-opacity. Meaningless here)
gl.enable(gl.DEPTH_TEST); gl.disable(gl.CULL_FACE);
```

(See WebGL standards docs; what do these 2 fcn calls do?
What other useful WebGL settings could you put here?)

- PUZZLED? Try Google/Bing: e.g. 'webgl clearcolor'
--instant WebGL reference manual!

- More →→

WebGL Setup(3): in JS file

STARTER CODE? See: Day2→ch05→ColoredMultiObject

- ▶ In our JavaScript file, in `main()` : (cont'd)
 - Send GLSL shader src code (txt strings) to GPU:
`initShaders()` fcn (in `../lib/cuong-utils.js`) loads, compiles, links...
 - Fill & Set up Vertex Buffer Object (VBO) in GPU:
call JS function `initVertexBuffer()` (explained in later slide)
 - Set up 'uniform' values sent to all shaders in GPU:
 - ▶ Ask GPU for location of each GLSL 'uniform' variable
`var u_ModelLoc = gl.getUniformLocation(gl.program, 'u_Model');`
 - ▶ Create the JS variable that will hold uniform's value:
`var myMatrix = new Matrix4(); // See ../lib/cuong-matrix.js
myMatrix.setIdentity(); // create, initialize`
 - ▶ Transfer JS value to the GPU's 'uniform' memory:
`gl.uniformMatrix4fv(u_ModelLoc, false, myMatrix.elements);`

WebGL Setup(4): in JS file

[STARTER CODE? See: Day2→ch05→ColoredMultiObject](#)

In our JavaScript file, in `main()` : (cont'd)

► **!DRAW!** == Tell GPU to draw VBO contents:

(To run vertex shaders and fragment shaders to draw current VBO contents using GPU's current uniform values)

- You can **DRAW ONCE** and stop:

```
gl.drawArrays(gl.TRIANGLES, // what to draw  
              0,           // start at vertex 0  
              12);         // draw 12 vertices
```

or

- **ANIMATE:** make JS endless loop (the `tick()` fcn) to call a nicely-unified `draw()` function. This fcn updates GPU's **uniforms**, then draws new picture,

!And your WebGL program is finished!

!NO!, wait! Explain these two!

1.

Send GLSL shader src code (txt strings) to GPU:

`initShaders()` fcn (in `../lib/cuong-utils.js`) loads, compiles, links...

Look inside this file to find the relevant WebGL commands:
compile, link, combine 2 shaders into a 'program', as book explains
(more details online, too)

2.

Fill & Set up Vertex Buffer Object (VBO) in GPU:

call JS function `initVertexBuffer()`

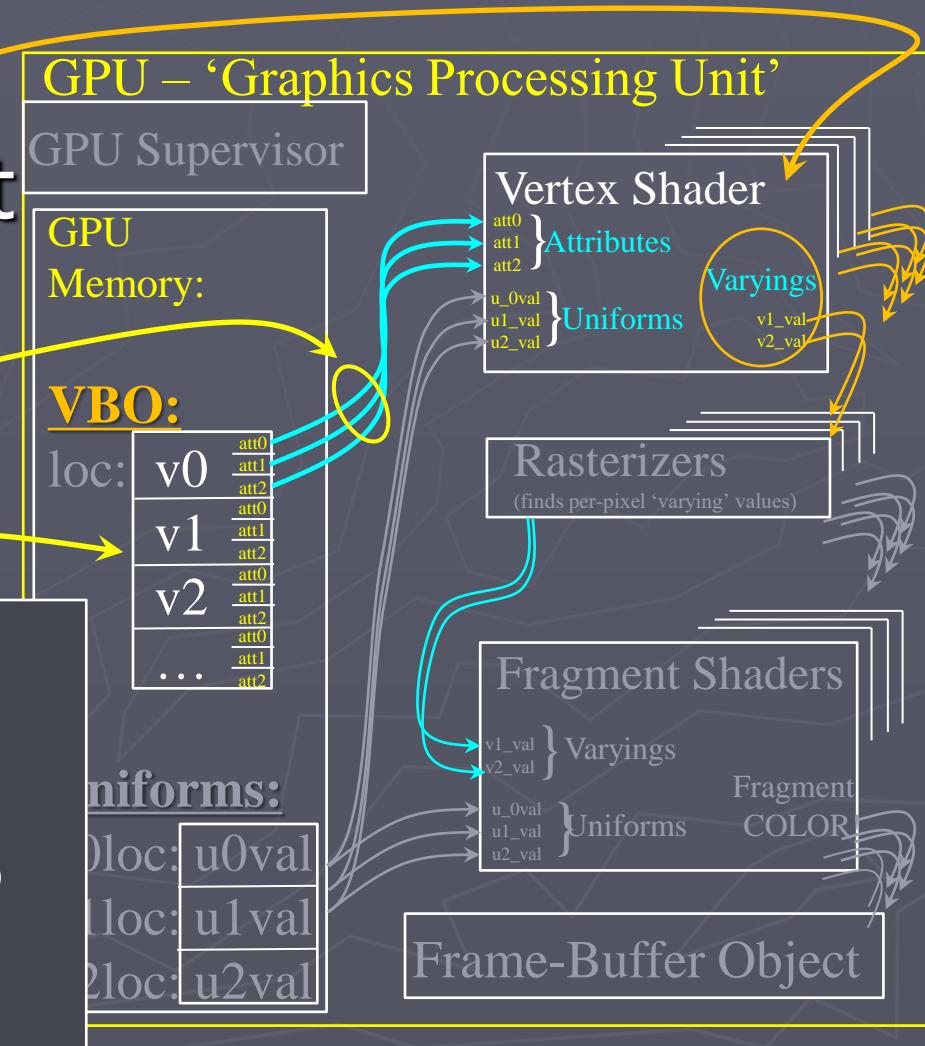
Must CREATE a VBO in the GPU,
then fill it with data contents from JS Float32array,
then 'connect' the VBO's contents to
the 'attribute' vars named in the Vertex shader. Here's how:

vertex Buffer object → 'Shaders'

- ▶ VBO holds our vertex set ($v_0, v_1, v_2 \dots$) in GPU:

▶ Each Vertex shader unit gets its own, separate vertex data set (a set of **attribute** values) from the VBO

?How Does WebGL tie *every attrib* of *every vertex* in the VBO to *every Vertex Shader* unit?



CREATE Vertex Buffer Object

SETUP I: STARTER CODE? See: Week3→ch05→ColoredMultiObject

- ▶ **Create** orderly vertex data in a JS fixed-type **Float32Array**:
initVertexBuffer() fcn: [line 106](#)
 - e.g. `var colorShapes = new Float32Array([0.0, 0.0, sq2, 1.0, ...]);`
 - (set of 'nn' vertices; each vert holds the same attributes)
- ▶ **Create** Vertex Buffer Object (VBO) in the GPU:
initVertexBuffer() fcn: [line 203](#)
 - WebGL command: `var shapeBufferHandle = gl.createBuffer();`
 - Save the GPU-supplied return value: an integer 'handle' that uniquely identifies this VBO for later use (no VBO? returns NULL)
- ▶ **'Bind'** the new VBO (e.g. 'select' it within the GPU)
as the parallel-data source for an array of vertices:
[line 210](#)
 - WebGL cmd: `gl.bindBuffer(gl.ARRAY_BUFFER, shapeBufferHandle);`
- ▶ **Transfer** JavaScript array data to VBO memory in GPU: [line 213](#)
 - `gl.bufferData(gl.ARRAY_BUFFER, colorShapes, gl.STATIC_DRAW);`
 - **Did it work?** You should always catch & report errors!!

CONNECT vertex Buffer object

SETUP II: [STARTER CODE? See: ch05→ColoredMultiObject](#)

VBOs supply vertex 'attribute' values to each vertex shader

► **Declare, Use 'attribute'** vars in GLSL vertex shader code

```
... 'attribute vec4 a_Position;\n' +         VSHADER SRC: line 20, 21  
  'attribute vec4 a_Color;\n' +
```

- (**!CAREFUL!** Requires flawless, non-trivial, ready JS & GLSL code!)

► **Connect** vars in GPU to get the values from VBO contents:

- Get the GPU's storage **location** for the GLSL 'attribute' variable [line 218](#)

```
var a_PositionLoc = gl.getAttribLocation(gl.program, 'a_Position');
```

- Use this **location** 'handle' to specify to find that attribute in the VBO:

```
gl.vertexAttribPointer( a_PositionLoc,                 // which attribute line 224  
                        4,                                 // # of values? 1,2,3 or 4? For us, (x,y,z,w)...  
                        gl.FLOAT,                         // data type of each value we'll xfer to GPU  
                        false,                            // needs normalizing? (ignore: fixed-pt only)  
                        FSIZE * 7,                        // Stride -- # bytes used to store each vertex?  
                        0);                                // Offset -- bytes to reach 1st attribute value  
  
gl.enableVertexAttribArray(a_PositionLoc); // do it!         line 233
```

CREATE a 'uniform' Var. In GPU

CAUTION! START with the JavaScript var !

- Create, init, test in JS before you add any uniforms to your GLSL shader(s)!

CAUTION! GLSL **uniforms** are All-Or-Nothing!

- No partially-complete **uniforms** will work:
Must do all JS setup; all GLSL setup & usage

CAUTION! GLSL compilers optimize! **AGGRESSIVELY!!**

- 'uniform' vars not used in shader program(s)
get **NO** GPU memory: they don't exist! Thus
`gl.getUniformLocation()` returns -1 (error).

(<http://stackoverflow.com/questions/7340150/why-is-glgetuniformlocation-failing-me>)

Important VBO Questions left unanswered by our Textbook:

- ▶ How can I modify the contents of a VBO?
- ▶ How do I properly discard a VBO and free the GPU memory it claimed?

--- A Bigger Question ---

- ▶ How Can I:
 - Make multiple VBOs (e.g. 1 per shape) and then
 - get multiple pairs of shaders (vertex, fragment) to switch between those VBOs for rendering?

END

GLSL Programmable Shaders: Why?

Advanced Programmable Shape, Lighting,
Materials & Texture effects:

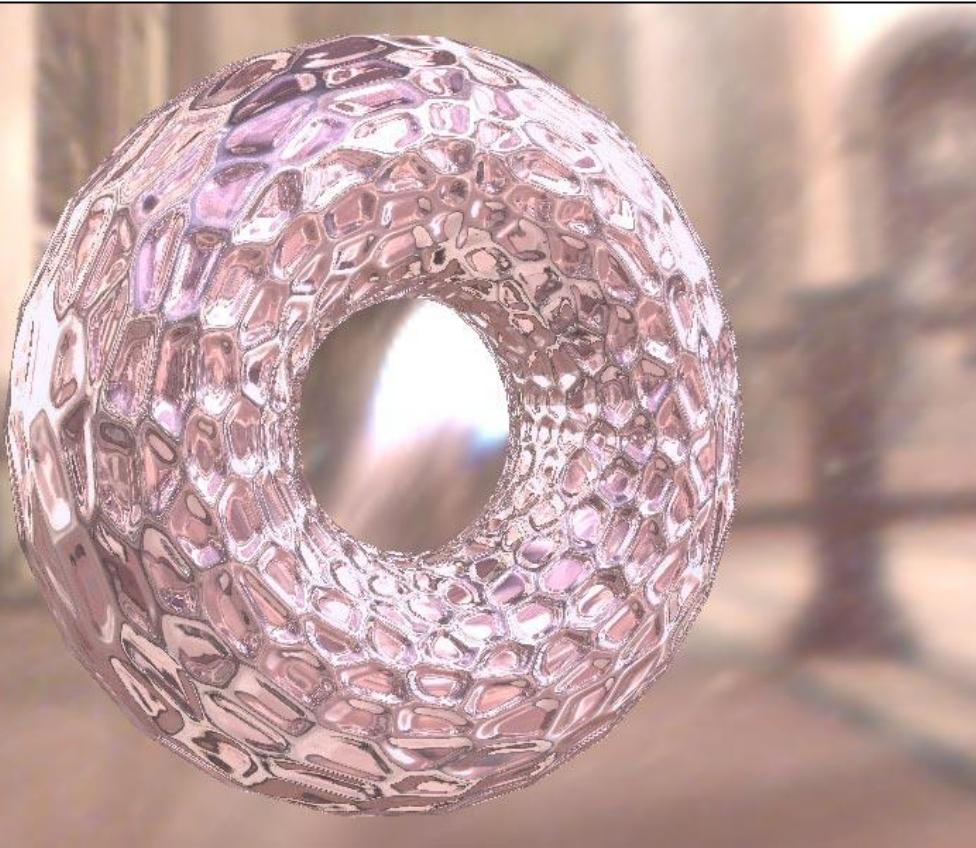


smooth,
per-pixel shading

'Environment
mapping'

'bump' mapping; even
'displacement' mapping

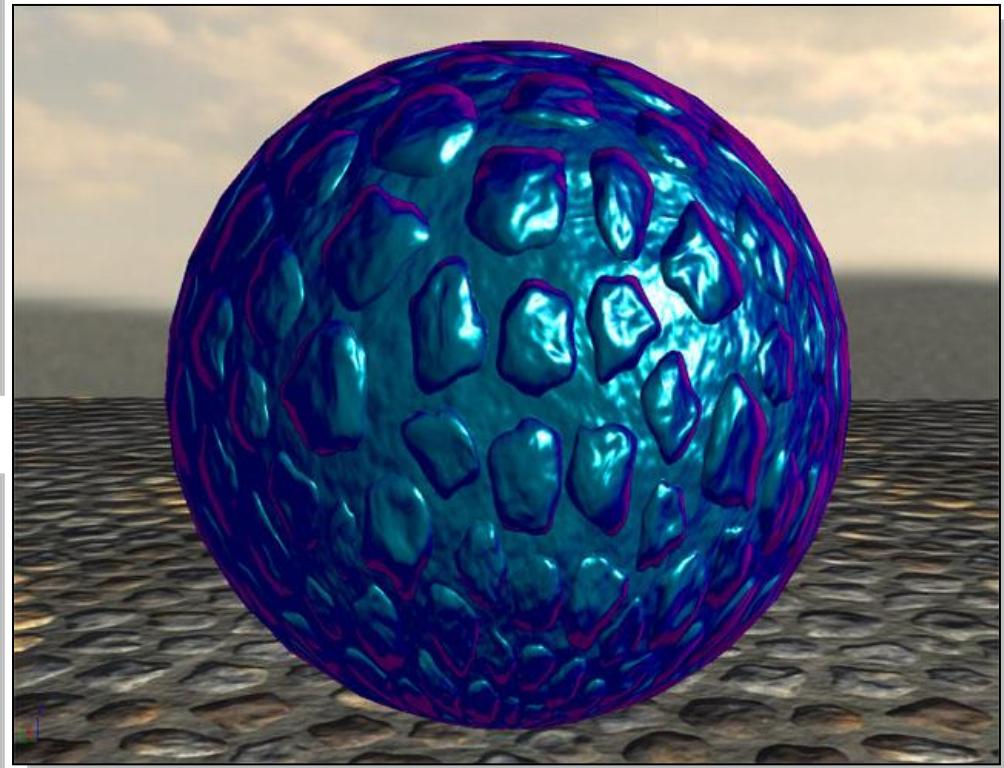
Shader gallery I



Above: Demo of Microsoft's XNA game platform
Right: Product demos by nvidia (top) and Radeon (bottom)



Shader gallery II



Above: Ben Cloward (“Car paint shader”)

Above: Kevin Boulanger (PhD thesis, “*Real-Time Realistic Rendering of Nature Scenes with Dynamic Lighting*”, 2005)

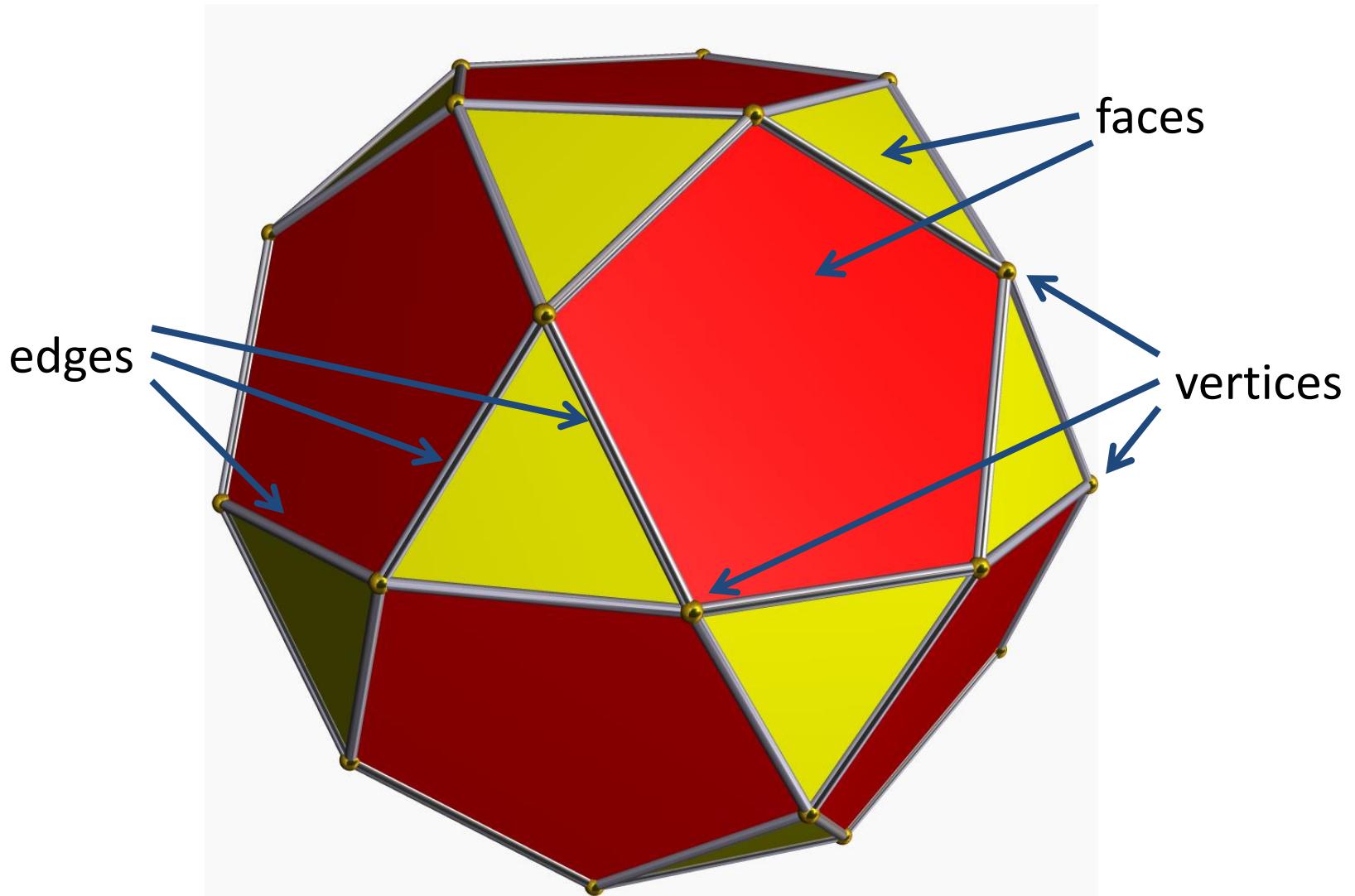
Vertex, Vector, and Matrix Math

(Part 1)

Jack Tumblin

Northwestern Univ COMP_SCI 351-1
Fall 2021

3D objects



Sylvie & Henri Gouraud – Computer Graphics Pioneers

<http://www.historyofcg.com/pages/henri-gouraud/>

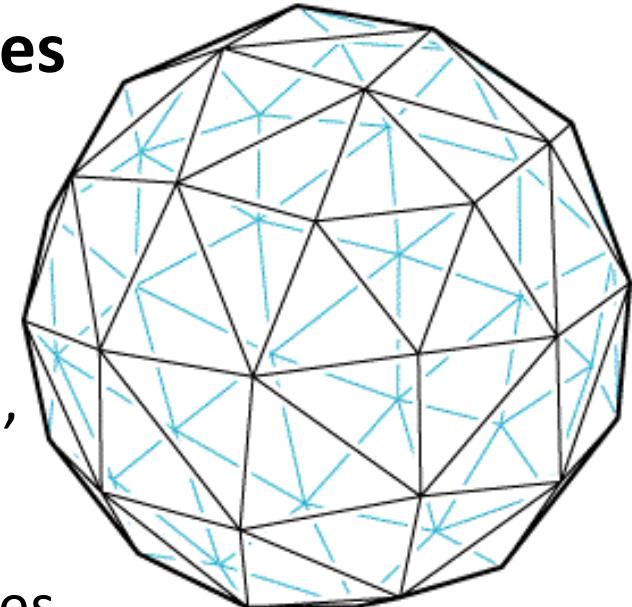


(1972)

How do we measure 3D locations to make interesting shapes?

3D objects as thin hollow shells

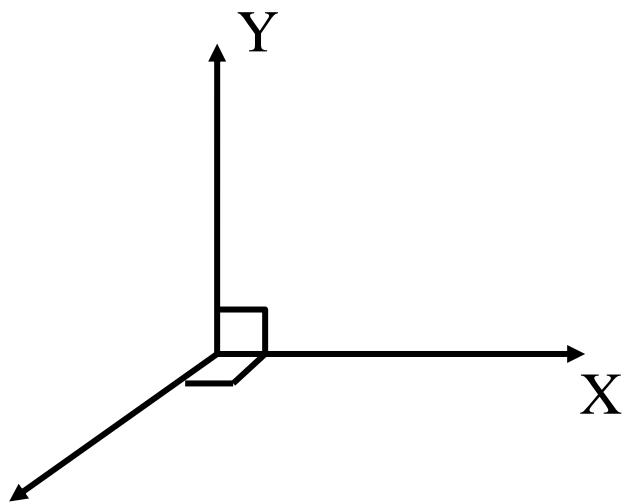
- Triangle ‘shells’ represent the shape **boundary**: the ‘interior’ volume defines a 3D bounded subspace
- Triangle meshes impose only planar/faceted boundaries: they define **faces**, **edges**, and **vertices**
- They don’t explicitly define “interior” vs “exterior”
 - it’s just a shell, a planar boundary
 - **YOU** decide what is ‘inside’ or ‘outside’
 - **YOU** decide what vertices to make, & how to group them into edges & faces.



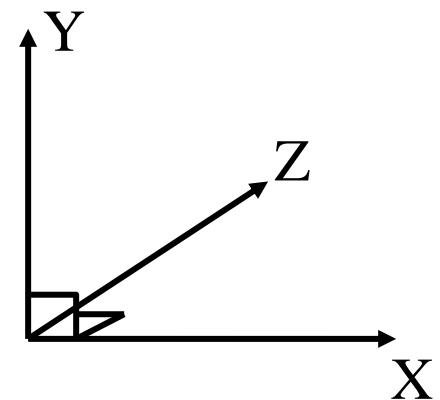
Math for Vertices? Vectors (4x1) and Matrices (4x4)

- Just about ***everything*** we need to compute in computer graphics fits neatly into 4-element vectors:
positions, orientations, normals, color, reflectance, quaternions, and more.
- Just about ***everything*** we need to compute with 4-element vectors fits neatly into a 4x4 matrix:
offset/translation, rotation, scale, skew, camera projections, texture mappers, coordinate axes, camera views, lights, surface normals, reflections, and more
- **BUT** JavaScript lacks 4-tuple vector/matrix library.
StarterCode + Book + my mods: **cuon-matrix-quat03.js**
Better, faster, (but different): Toji's [glmatrix.js library](#)

Coordinate Systems

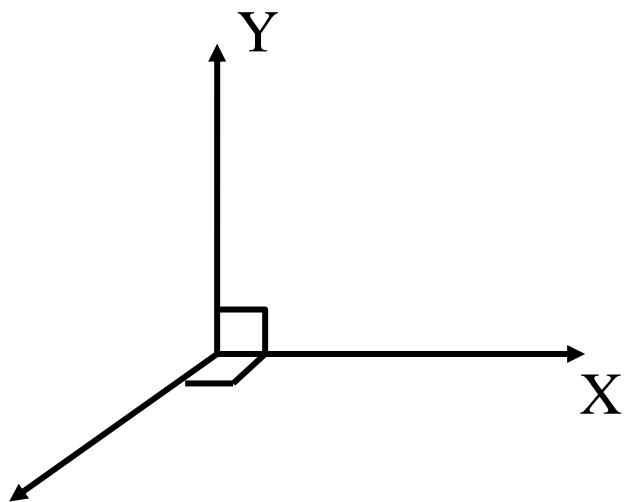


Right-Handed System
(Z comes **out** of the screen)

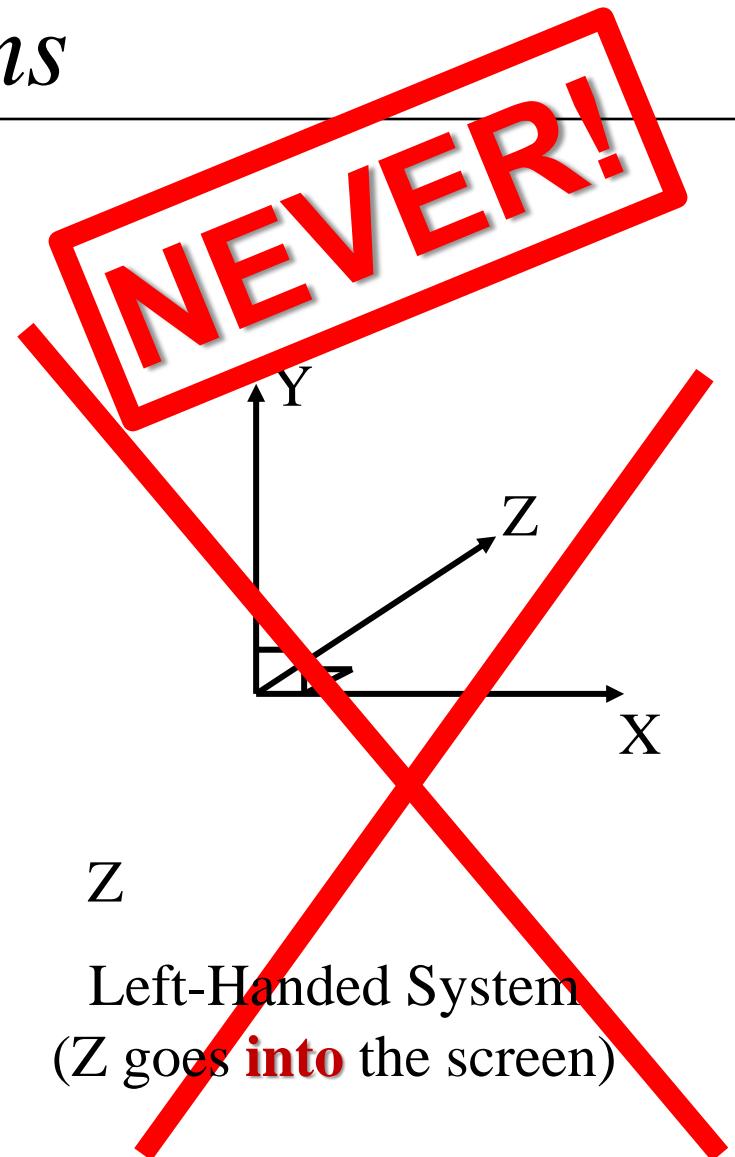


Z
Left-Handed System
(Z goes **into** the screen)

Coordinate Systems

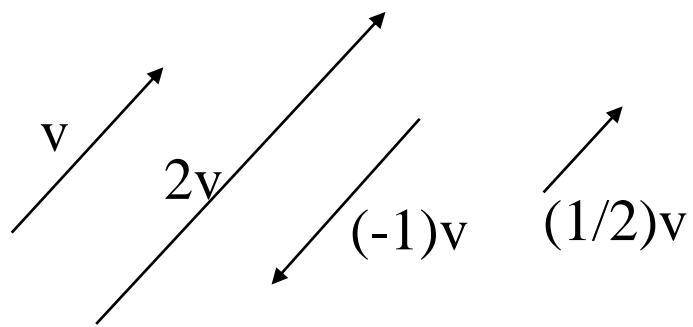


Right-Handed System
(Z comes **out** of the screen)

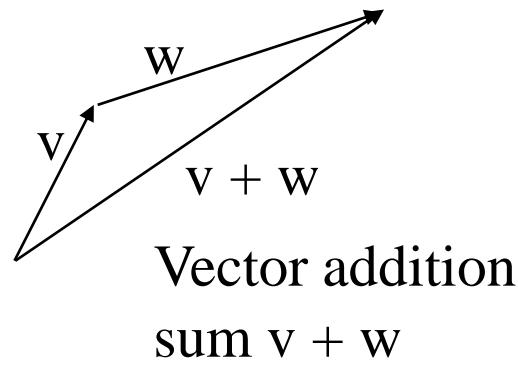


Left-Handed System
(Z goes **into** the screen)

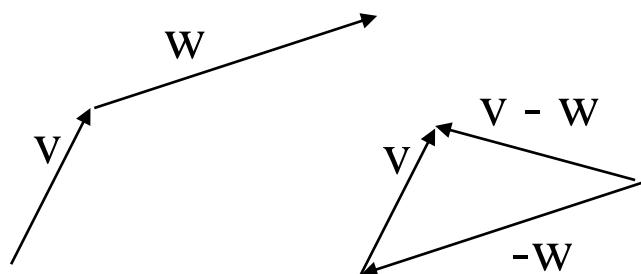
v: vectors in 3D (x, y, z)



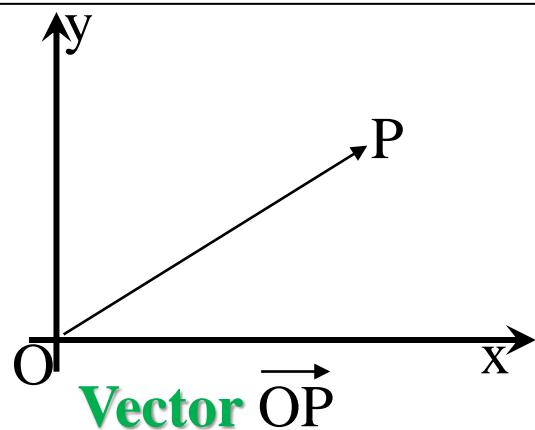
Scalar multiplication of
vectors (they remain parallel)



Vector addition
sum $v + w$



Vector difference
 $v - w = v + (-w)$



(distance and direction only (NOT location!),
from origin **point** O to **point** P)

Vectors v

- Length (magnitude) of vector \mathbf{v} (x, y, z)

$$\|\mathbf{v}\| = \sqrt{x^2 + y^2 + z^2}$$

- A **unit vector**: a vector of length 1
- ‘Normalized’ vector: a vector whose **length was scaled to 1** but retained its original direction:

$$\hat{\mathbf{v}} = \frac{\text{vector } \mathbf{v}}{\text{length}(\mathbf{v})} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$$

Vector Multiply (1): Dot Product

- The ‘dot product’ or ‘inner product’: 
 - Input:** two vectors (**not points!** $w==0$ in homog. coords),
 - Both vectors must be the same N-dimensional size (N=3 for us)
 - Output:** one signed scalar value.

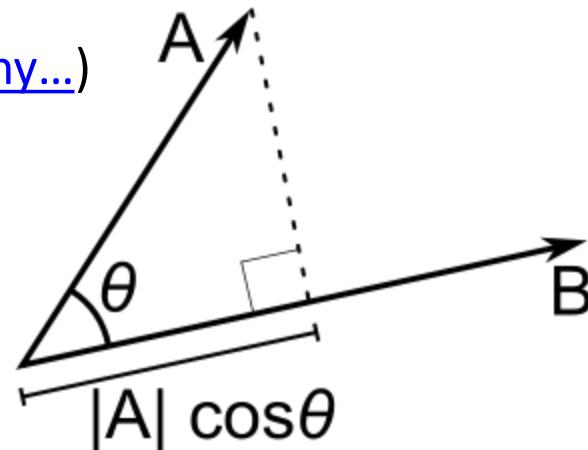
(Symmetric, too! [3Blue1Brown explores why...](#))

$$\text{Vector } A = [ax, ay, az]$$

$$\text{Vector } B = [bx, by, bz]$$

$$A \cdot B \equiv ax * bx + ay * by + az * bz$$

$$A \cdot B \equiv |A| |B| \cos \theta$$

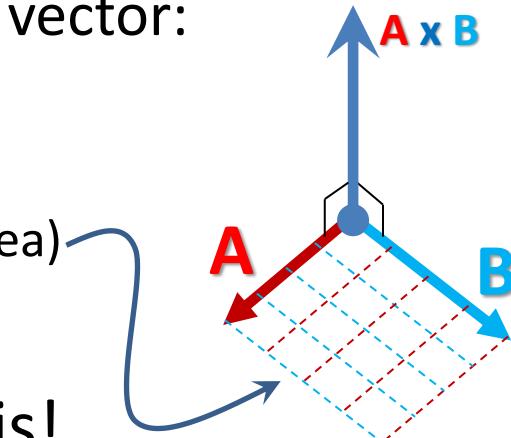


- Any ‘dot product’ function for homogeneous coords:
 - Should ignore the input vectors’ w coords. (because they’re zero)
 - Should **complain** if given nonzero w for either input ‘vector’

Vector Multiply(2): Cross-Product

The ‘cross product’ or ‘3D exterior product’: **A x B**

- Input: two 3-dimensional **vectors**
 - (**not points!** w==0 in homog. coords)
 - **Always 3D**: cross product trivial for 2D (a.k.a. Cramer’s Rule) and ill-defined for higher dimensions (why? Do a web-search on it...)
- Output: one mutually perpendicular 3D vector:
 - **DIRECTION** follows right-hand rule, and
 - **MAGNITUDE** is: $|A \times B| \equiv |A| |B| \sin \theta$
 \equiv (parallelogram area)



COOL WAY TO REMEMBER IT: Watch this!

<http://1ucasvb.tumblr.com/post/76812811092/given-two-vectors-in-three-dimensions-one-can>

Vector Multiply(2): Cross-Product

The ‘cross product’ or ‘3D exterior product’: $\boxed{\mathbf{A} \times \mathbf{B}}$

- How to compute cross product:

$$\mathbf{A} \times \mathbf{B} = [Ay^*Bz - Az^*By, \\ Az^*Bx - Ax^*Bz, \\ Ax^*By - Ay^*Bx]$$

- **ANTI**-COMMUTATIVE! $\mathbf{A} \times \mathbf{B} = -(B \times A)$
- Vectors \mathbf{A}, \mathbf{B} define 2 sides of a parallelogram:
area of parallelogram == length of $(\mathbf{A} \times \mathbf{B})$ vector:
$$|\mathbf{A} \times \mathbf{B}| \equiv |\mathbf{A}| |\mathbf{B}| \sin \theta$$
- If \mathbf{A} and \mathbf{B} are perpendicular, then $|\mathbf{A} \times \mathbf{B}| \equiv |\mathbf{A}| |\mathbf{B}|$
- If \mathbf{A} and \mathbf{B} are parallel, then $|\mathbf{A} \times \mathbf{B}| \equiv 0$

INVESTIGATE:

For any two 3-element vectors \mathbf{A} , \mathbf{B}

1) ? Does $\mathbf{A} \cdot \mathbf{B} \equiv \mathbf{B} \cdot \mathbf{A}$? **Show me why.**

2) ? Does $\mathbf{A} \times \mathbf{B} \equiv \mathbf{B} \times \mathbf{A}$? **Show me why.**

(Review)

Some Important Matrix Types:

□ *Identity Matrices: I*

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

□ *Symmetric Matrices*

$$\begin{pmatrix} a & b & c \\ b & d & e \\ c & e & f \end{pmatrix}$$

□ *Diagonal Matrices*

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -4 \end{pmatrix}$$

- *Diagonal matrices are (of course) symmetric.*
- *Identity matrices are (of course) diagonal*
- *Each diagonal element multiplies a vector element*

(Review) Matrix Multiplication

- Matrix \mathbf{A} : n by k elements
- Matrix \mathbf{B} : k by m elements
- $\mathbf{C} = \mathbf{AB}$ defined by its elements

$$c_{ij} = \sum_{m=1}^k a_{im} b_{mj}$$

$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} \begin{pmatrix} * \\ * \\ * \\ * \\ * \end{pmatrix} = \begin{pmatrix} * \\ * \\ * \\ * \\ * \end{pmatrix}$$

$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} \begin{pmatrix} . & * \\ . & * \\ . & * \\ . & * \\ . & * \end{pmatrix} = \begin{pmatrix} . & * \\ . & * \\ . & * \\ . & * \\ . & * \end{pmatrix}$$

- Careful! Order matters!

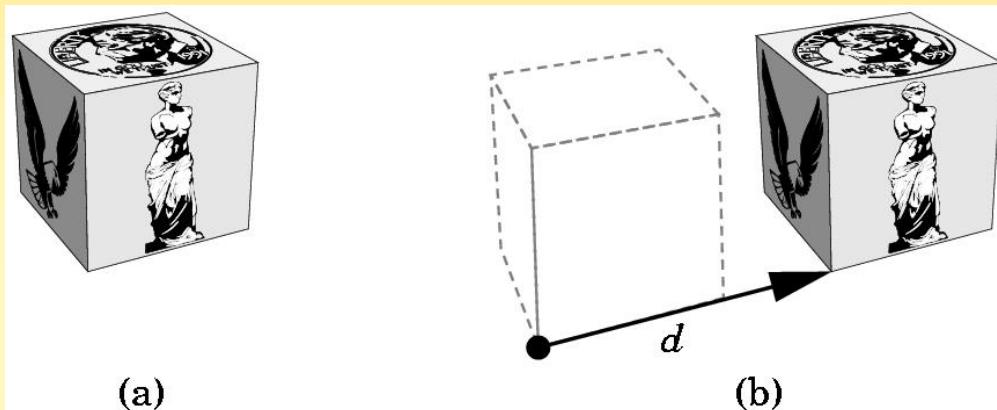
(BA)

!!!-- **NOT EQUIVALENT TO --!!!**

(AB)

$$\begin{pmatrix} . & * \\ . & * \\ . & * \\ . & * \\ . & * \end{pmatrix} \begin{pmatrix} * \\ * \\ * \\ * \\ * \end{pmatrix} = \begin{pmatrix} . & * \\ . & * \\ . & * \\ . & * \\ . & * \end{pmatrix}$$

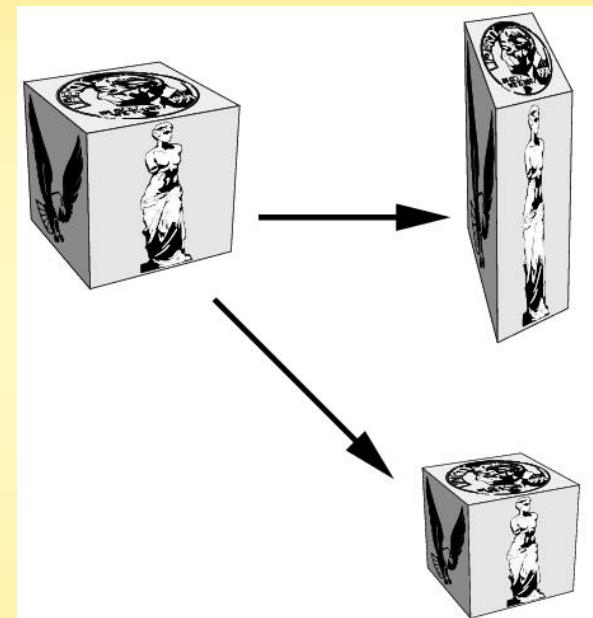
3D Translation Matrix (4x4)



$$\begin{matrix} \hat{x}' \\ \hat{y}' \\ \hat{z}' \\ \hat{1} \end{matrix} = \begin{matrix} \hat{x} & 0 & 0 & d_x \\ \hat{y} & 0 & 1 & d_y \\ \hat{z} & 0 & 1 & d_z \\ \hat{1} & 0 & 0 & 1 \end{matrix} \begin{matrix} \hat{x} \\ \hat{y} \\ \hat{z} \\ \hat{1} \end{matrix}$$

3D Scaling Matrix (4x4)

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



- Origin is the 'fixed point' for scaling. Scaling changes vertex position along a line through the origin, moving closer or further away.

3D Rotation Matrices (4x4)

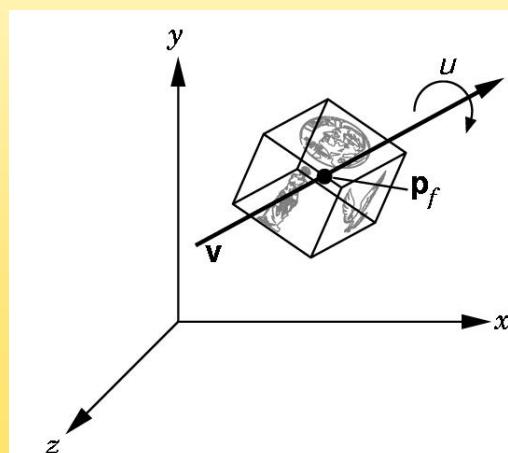
- Origin is the 'fixed point' for rotation matrices.
To rotate around the x, y, or z-axis, use these:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- How could we get a general rotation?
How could we spin around an axis \mathbf{v} by angle of \underline{u} degrees, around a fixed point \mathbf{p} ?



Posing in 3D: Challenges

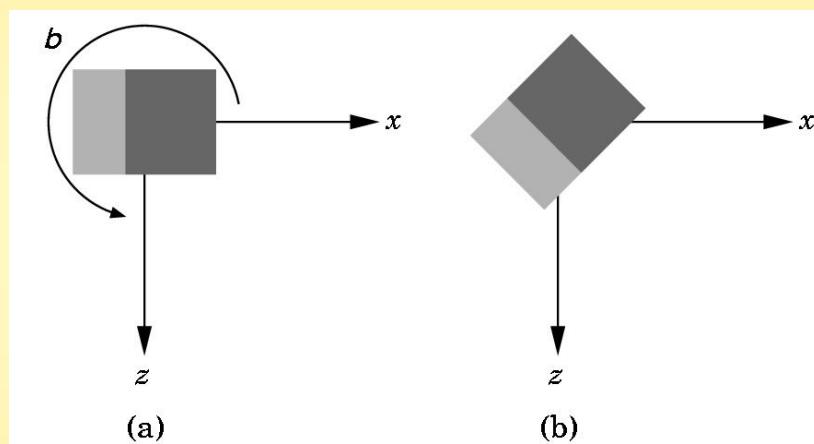
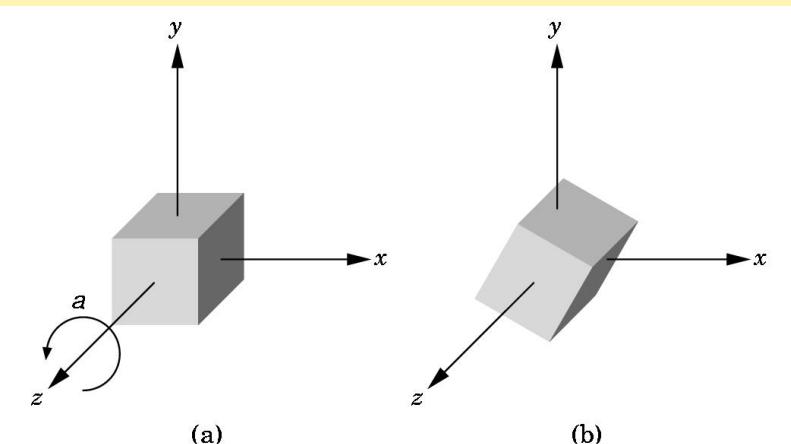
- Given a ‘unit cube’ whose corner points are located at $(x,y,z) == (+/-1, +/-1, +/-1)$, find a short simple sequence of R_x , R_y , or R_z rotations that will:
 - move the $(+1, +1, +1)$ corner onto the $+z$ axis, and
 - move the $(-1, -1, -1)$ corner onto the $-z$ axis, and
 - move the $(-1, +1, +1)$ to the xz plane (where $y==0$)

PUZZLE:

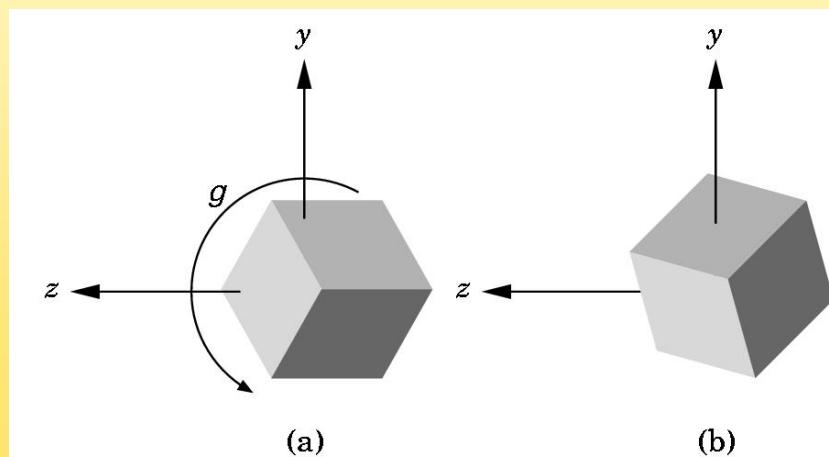
Imagine that the xy plane cuts this rotated cube in half.
What is the shape of the cube/plane intersection?

Rotating about another axis

- How can I rotate about an *arbitrary* axis?



- Can combine rotations about z, y, and x:
 $R_x R_y R_z P = P'$
- Note that order matters and angles can be hard to find...



Good Resources

LucasVB (Lucas V. ?Barbossa? São Paulo, Brazil), ‘Toji’ (Brandon Jones)

- Beautiful visualization of sines and cosines

<http://1ucasvb.tumblr.com/post/79557434791/the-sine-and-cosine-functions-for-the-circle-as>

- Cross Product visualization:

<http://1ucasvb.tumblr.com/post/76812811092/given-two-vectors-in-three-dimensions-one-can>

- Could you tessellate a torus by extrusion or circular sweep of a pair of (discretized) Villarceau circles?

<http://1ucasvb.tumblr.com/post/52482862675/villarceau-circles-on-a-torus-next-time-you-have>

BETTER vector/matrix library than our textbook’s **cuon-matrix.js**

(and my extensions: cuon-matrix-mod.js and cuon-matrix-quat.js)

<http://glmatrix.net/> (by ‘Toji’ – excellent WebGL explainer/author!)

Good Resources

LucasVB (Lucas V. ?Barbossa? São Paulo, Brazil)

BETTER vector/matrix library than our textbook's **cuon-matrix.js**

(and my extensions: cuon-matrix-mod.js and cuon-matrix-quat.js)

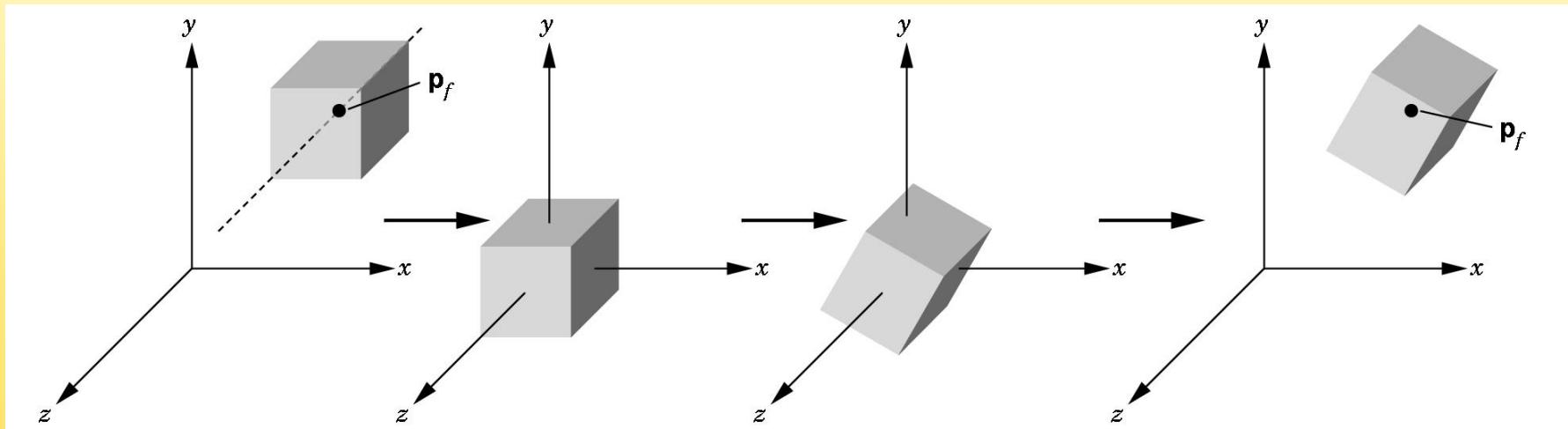
<http://glmatrix.net/> (by 'Toji' – excellent WebGL explainer/author!)

Try it!

(More on this Monday...)

Rotating about another point

- How can I rotate around another fixed point, e.g. [1, 2, 3]?
 - Translate [-1, -2, -3] (\mathbf{T})
 - Rotate (\mathbf{R})
 - Translate back (\mathbf{T}^{-1})
 - $\mathbf{T}^{-1} \mathbf{R} \mathbf{T} \mathbf{P} = \mathbf{P}'$



Rotating about another point

- How can I rotate around a point, e.g. [1, 2, 3]?

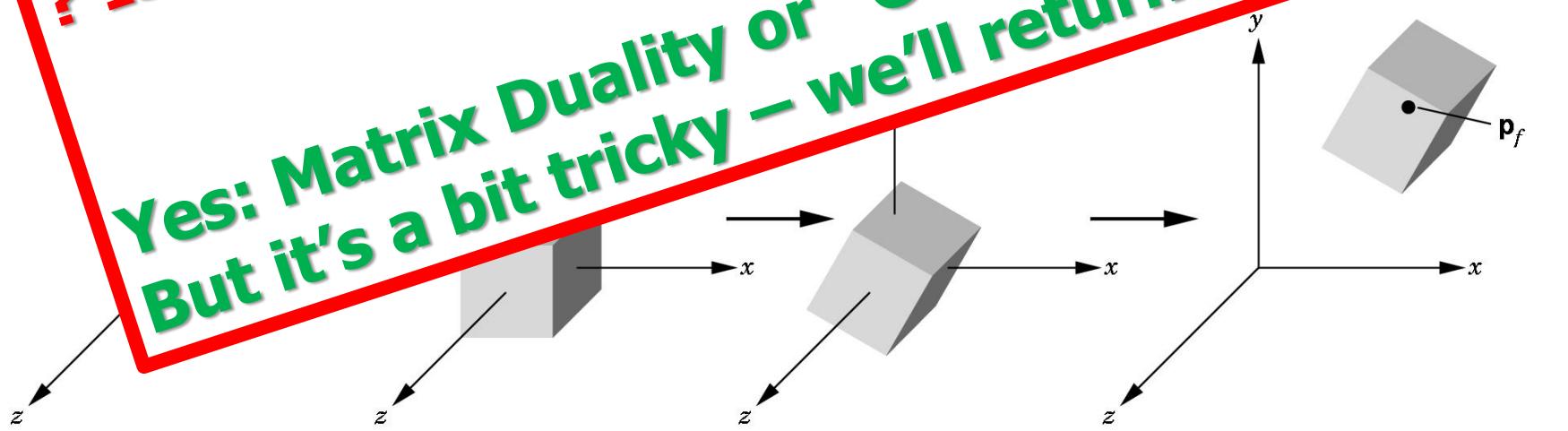
...
...
...
...
...

YUCK!!

WHAT A MESS!

? Isn't there a better way?!?!

**Yes: Matrix Duality or 'Coord. Frames'
But it's a bit tricky – we'll return to this...**



END?

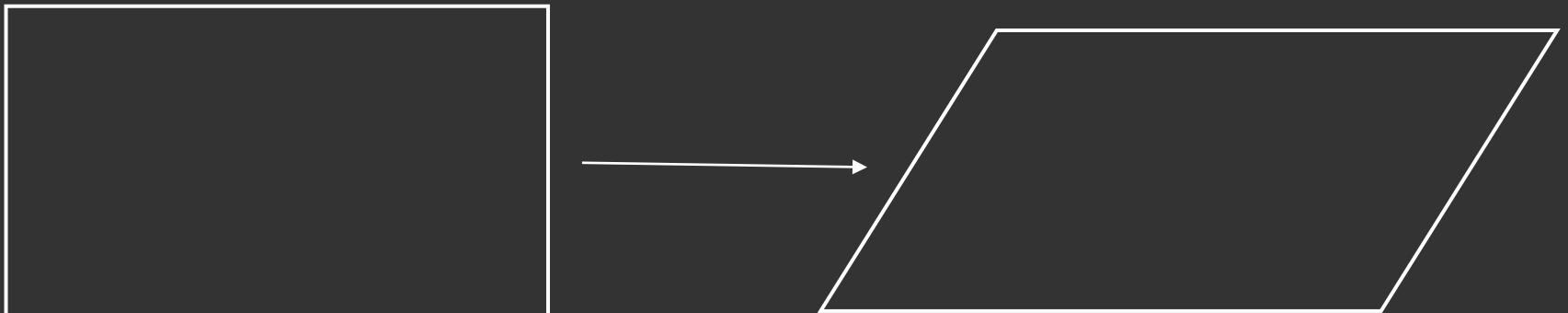
(Nonuniform) Scale

$$Scale(s_x, s_y) = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \quad S^{-1} = \begin{pmatrix} s_x^{-1} & 0 \\ 0 & s_y^{-1} \end{pmatrix}$$

$$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} s_x x \\ s_y y \\ s_z z \end{pmatrix}$$

Shear

$$Shear = \begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix} \quad S^{-1} = \begin{pmatrix} 1 & -a \\ 0 & 1 \end{pmatrix}$$



Rotations

2D simple, 3D complicated. [Derivation? Examples?]

2D?

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- Linear: $R(X+Y)=R(X)+R(Y)$
- Commutative: $R(X) R(Y) = R(Y)R(X)$

BUT these are rotations around Z axis in 3D.
Can you construct rotations around X and Y axes too?

Rotations in 3D

- Rotations about coordinate axes simple

$$R_z = \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{pmatrix}$$

$$R_y = \begin{pmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{pmatrix}$$

- Always linear & orthogonal
 - Rows/cols orthonormal
 - But NOT commutative!

$$R^T R = I$$

$$R(X+Y) = R(X) + R(Y)$$

$$R(X)R(Y) \neq R(Y)R(X)$$

3D Rotation Matrix Contents

- Rows of matrix are 3 unit vectors of new coordinate frame
- Can construct a rotation matrix from 3 orthonormal vectors

$$R_{uvw} = \begin{pmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{pmatrix} \quad u = x_u X + y_u Y + z_u Z$$

$$Rp = \begin{pmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{pmatrix} \begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix} = ? \quad \begin{pmatrix} u \bullet p \\ v \bullet p \\ w \bullet p \end{pmatrix}$$

3D Rotation Matrix Contents

$$Rp = \begin{pmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{pmatrix} \begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix} = \begin{pmatrix} u \bullet p \\ v \bullet p \\ w \bullet p \end{pmatrix}$$

- Rows of matrix are 3 unit vectors of new coord frame
- Can construct rotation matrix from 3 orthonormal vectors
- Effectively, projections of point into new coord frame
- New coord. frame uvw taken to Cartesian components xyz
- Inverse or transpose takes xyz cartesian to uvw

Axis-Angle formula

- Step 1: **a** and **b** have parallel & perpendicular parts:
 - Parallel component unchanged (rotating about an axis leaves that axis unchanged after rotation, e.g. rot. about z axis)
- Step 2: Define **c** orthogonal to both **a** and **b**
 - Analogous to defining Y axis
 - Use cross products and matrix formula for that
- Step 3: With respect to the perpendicular comp of **b**
 - Cos θ of it remains unchanged
 - Sin θ of it projects onto vector **c**
 - Verify this is correct for rotating X about Z
 - Verify this is correct for θ as 0, 90 degrees

Axis-Angle: Putting it together

$$(b \setminus a)_{ROT} = (I_{3 \times 3} \cos \theta - aa^T \cos \theta)b + (A^* \sin \theta)b$$

$$(b \rightarrow a)_{ROT} = (aa^T)b$$

$$R(a, \theta) = I_{3 \times 3} \cos \theta + aa^T (1 - \cos \theta) + A^* \sin \theta$$



 Unchanged Component Perpendicular
 (cosine) along **a** (rotated comp)
 (hence unchanged)

Axis-Angle: Putting it together

$$(b \setminus a)_{ROT} = (I_{3 \times 3} \cos \theta - aa^T \cos \theta)b + (A^* \sin \theta)b$$

$$(b \rightarrow a)_{ROT} = (aa^T)b$$

$$R(a, \theta) = I_{3 \times 3} \cos \theta + aa^T (1 - \cos \theta) + A^* \sin \theta$$

$$R(a, \theta) = \cos \theta \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + (1 - \cos \theta) \begin{pmatrix} x^2 & xy & xz \\ xy & y^2 & yz \\ xz & yz & z^2 \end{pmatrix} + \sin \theta \begin{pmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{pmatrix}$$

(x y z) are cartesian components of **a**

Matrix Math (Part 2)

Duality and Scene Graphs: How to Build Trees of Transformations



Jack Tumblin
COMP_SCI 351-1
Fall 2021

(includes re-worked slides
by Andries van Dam)

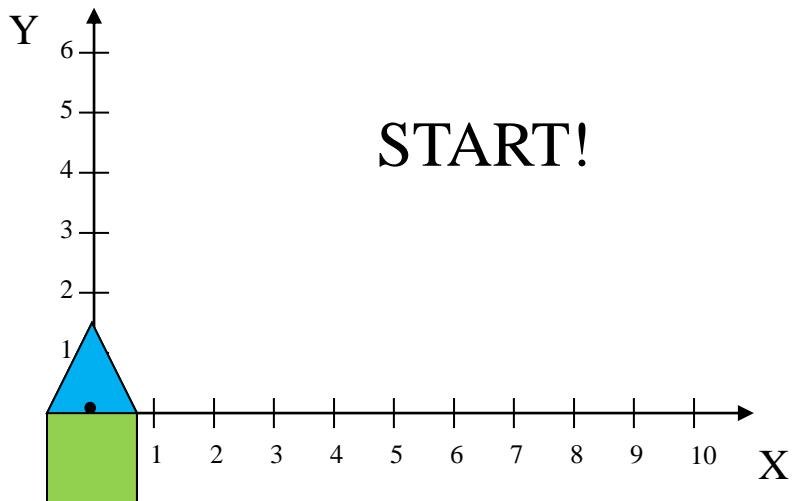
Do It ALL with a 4x4 Matrix...

- Vertex == [Set of Attributes]
- Rigid 3D part==[Set of Vertices]
- How do we control it?
- In Monday class, we learned to multiply points by a 4x4 matrix to:
 - TRANSLATE by (x,y,z)
 - SCALE by (sx,sy,sz)
 - ROTATE by θ on axis (x,y,z)
- TODAY: How to Combine them!
 - How? Matrix multiply.
 - But in what order?
 - And why do transform functions seem “backwards” (initially)?

How do **Sequences** of Matrices Affect How we Draw Vertices?

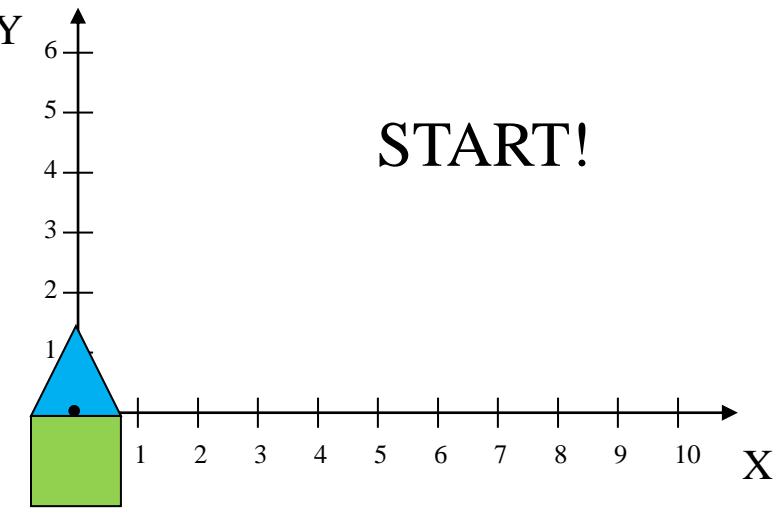
(ANS: Ill-Posed Q: one matrix, TWO Geometric Methods)

Translate()
by $x=6, y=0$
then
Rotate() by 45°



Translation → Rotation

Rotate() by 45°
then
Translate()
by $x=6, y=0$



Rotation → Translation

SURPRISE!

**Every transformation matrix
has not one, but
TWO geometric
interpretations.**

**They're EASY to confuse,
EASY to mix up by accident
(even for textbook authors!)**

**But they yield
OPPOSITE RESULTS!**

**Let's be SURE we
understand this "Matrix Duality"**

And apply it consistently!

How does a Transform Matrix Affect How we Draw Vertices?

METHOD 1: (the obvious one)

--Keep one fixed coordinate system, and
--for each successive transform applied,

Create *new coordinate values*

(the x,y,z,w) for each point or vertex.

--Plot the changed coordinate *numbers* in the original coordinate system.

(Simple, obvious, intuitive, and bad)

**Let's try this 'Method 1'
using pictures**

METHOD 1: (change *numbers*)

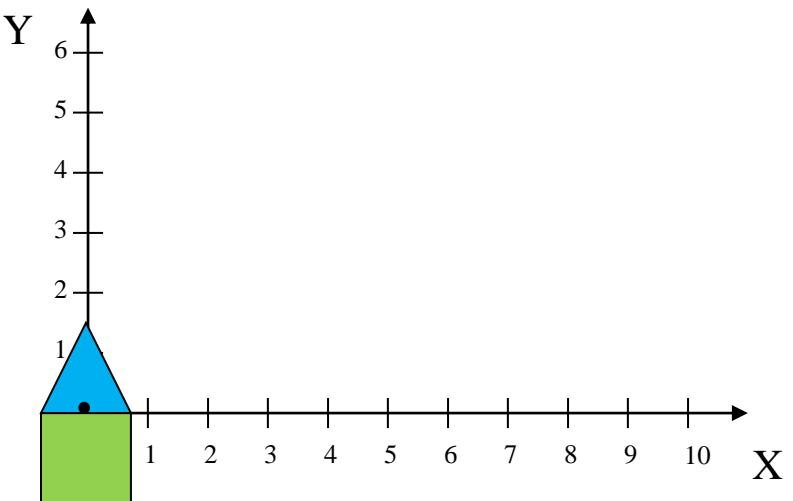
1a) Draw 'house' at origin

Translate()

by $x=6$, $y=0$

then

Rotate() by 45°

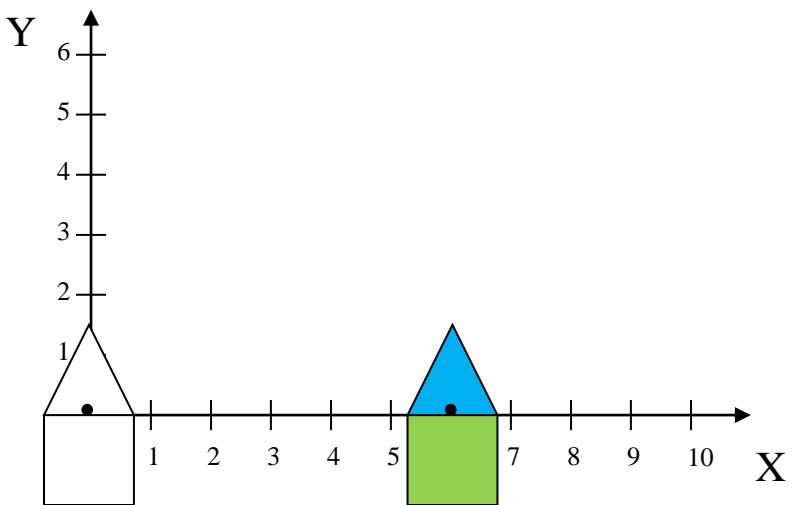


Translation → Rotation

METHOD 1: (change numbers)

1b) Add 6 to all its x coords

Translate()
by $x=6, y=0$
then
Rotate() by 45°

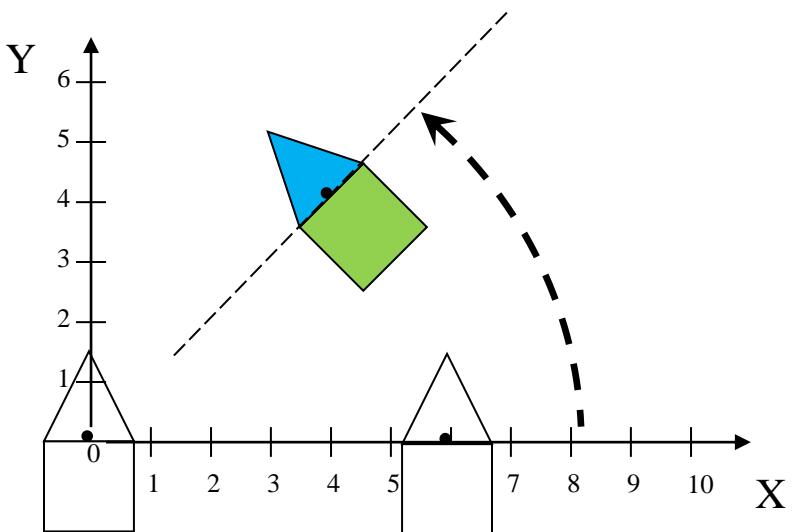


Translation → Rotation

METHOD 1: (change numbers)

1c) Rotate: all-new x,y coords

Translate()
by $x=6$, $y=0$
then
Rotate() by 45°



Translation → Rotation

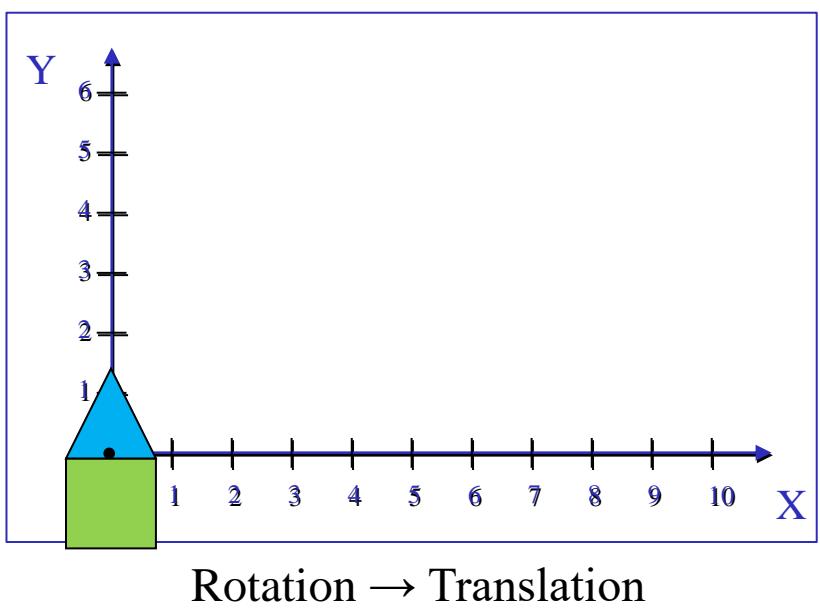
UGH.
Rotates around origin point.
PROBABLY
not what you wanted!

TRY THE REVERSE ORDER:

METHOD 1: (change numbers)

1a) Draw 'house' at origin

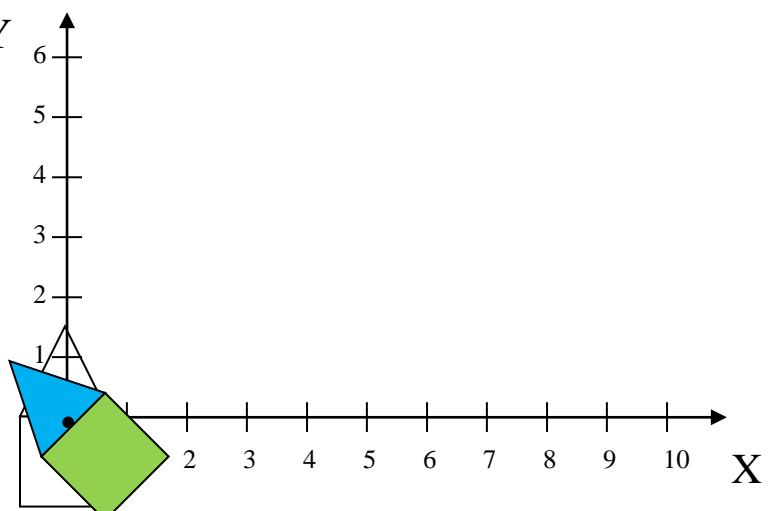
Rotate() by 45°
then
Translate()
by $x=6$, $y=0$



METHOD 1: (change numbers)

1b) Rotate: all new x,y coords:

Rotate() by 45°
then
Translate()
by $x=6$, $y=0$

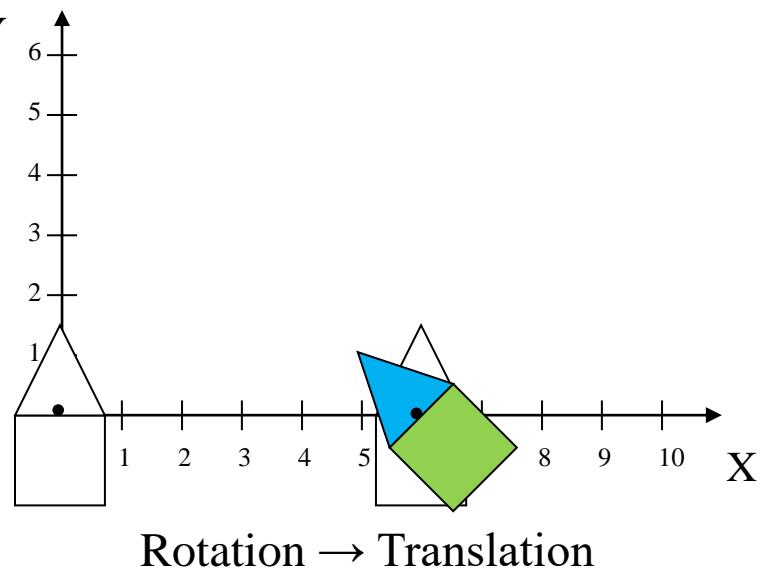


Rotation \rightarrow Translation

METHOD 1: (change numbers)

1c) Add 6 to all its x coords.

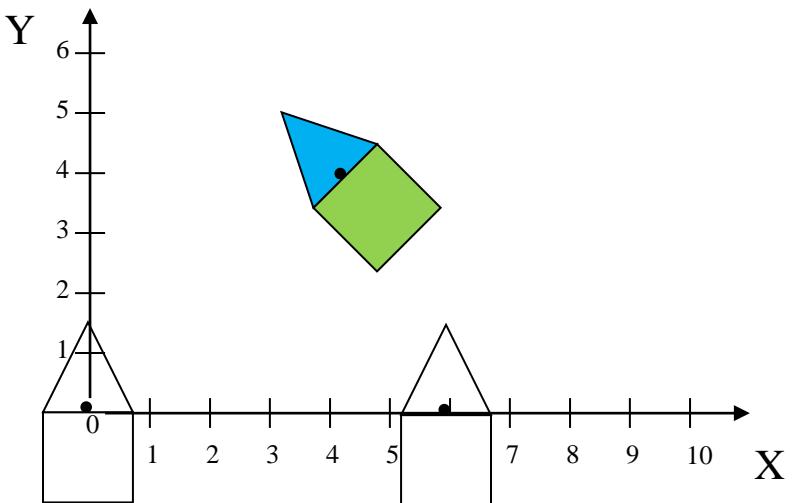
Rotate() by 45°
then
Translate()
by $x=6, y=0$



METHOD 1: (change numbers)

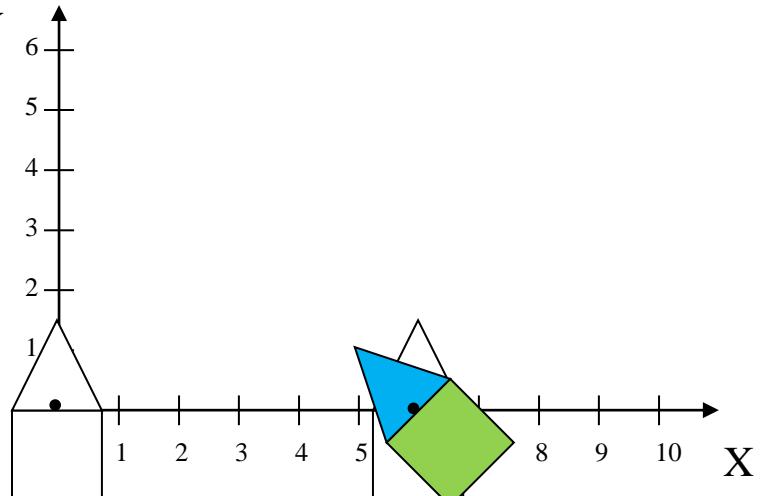
DIFFERENT RESULTS, because
→Matrices are *NOT* Commutative.

Translate()
by $x=6$, $y=0$
then
Rotate() by 45°



Translation → Rotation

Rotate() by 45°
then
Translate()
by $x=6$, $y=0$



Rotation → Translation

METHOD 1:

DIFFERENT RESULTS, because
→ Matrices are *NOT* Commutative.

Translate()
by $x=6$, $y=0$
then
Rotate() by 45°

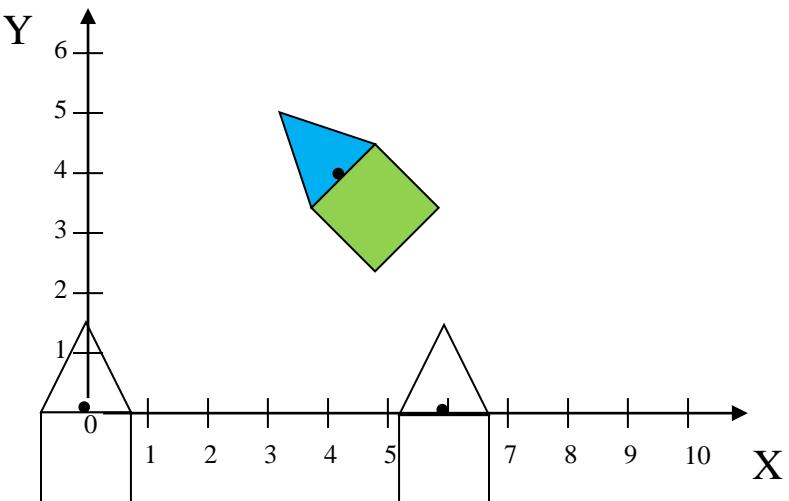


(WHAAA?!?! But it's so very Obvious! How could anyone ever get confused by this?!?!)

A 2D Cartesian coordinate system with X and Y axes ranging from 0 to 10. Two triangles are plotted: one at (1, 1) and another at (5, 5). A blue arrow points from the first triangle to the second, labeled "Rotation → Translation".

METHOD 1: (change numbers)

Translate()
by $x=6$, $y=0$
then
Rotate() by 45°



Translation → Rotation

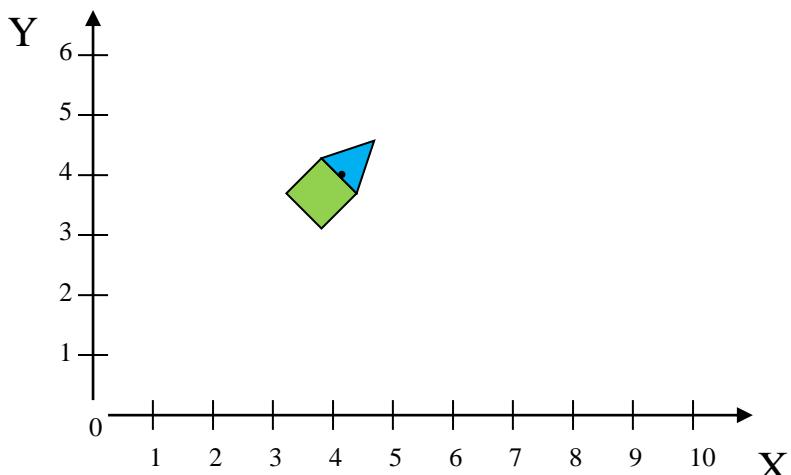
Not really.

Here, try to append these ‘simple’ new tasks:

DON’T move the house center-point, but
--Scale() it to half the original size, and
--Rotate() it clockwise by $+90^\circ$ to get this result:

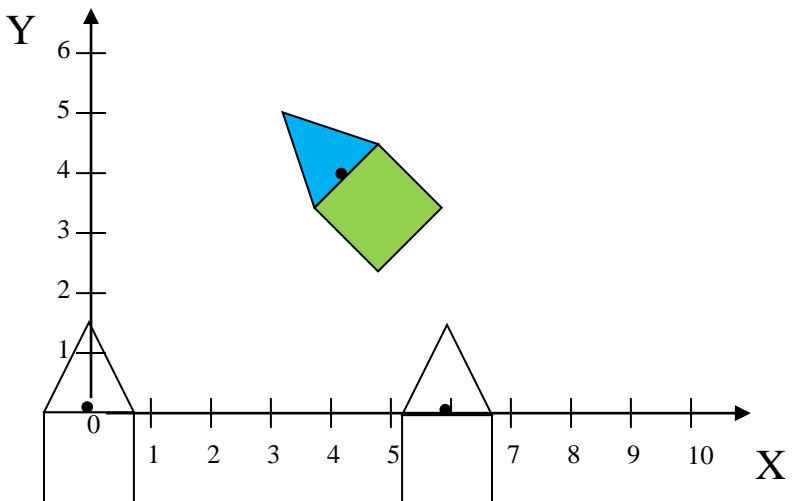
Well?

How can you do it?



METHOD 1: (change numbers)

Translate() by
 $x=6, y=0$
then
Rotate() by 45°



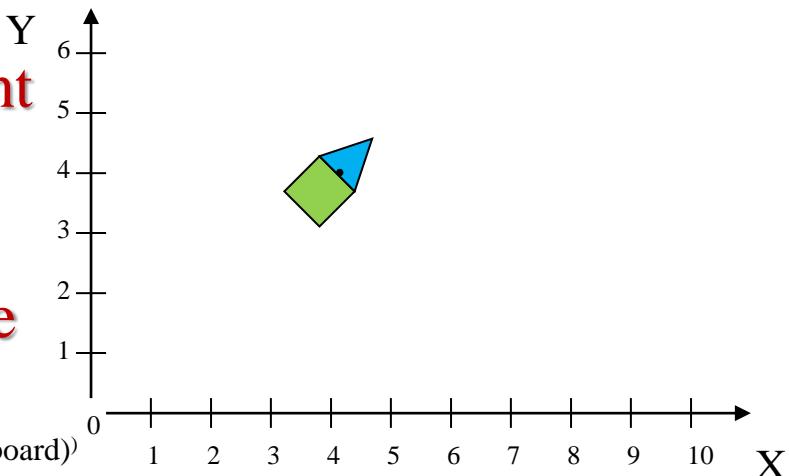
Translation → Rotation

Always Easy? Obvious? Are you sure?
Then how will you append these ‘simple’ new tasks:
DON’T move the house center-point, but
--Scale() it to half the original size, and
--Rotate() it clockwise by $+90^\circ$ to get this result:

Oh Wow! That’s !UGLY!

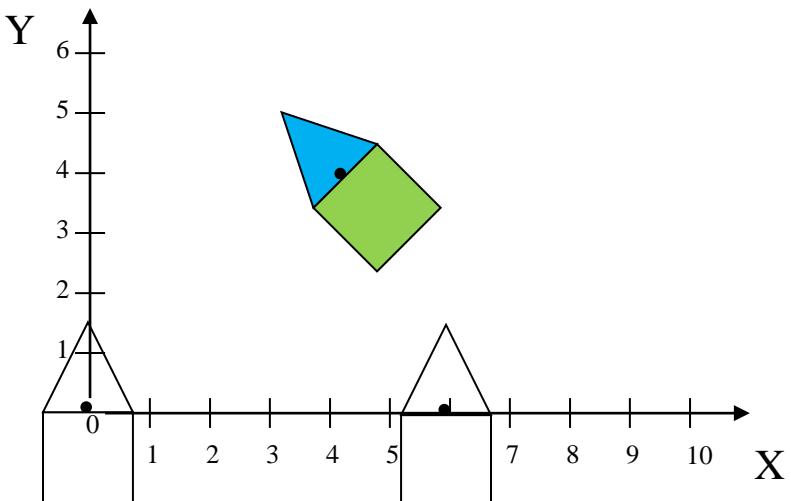
We have to return
the house’s center-point
to the origin before
we can scale() by 0.5,
and return again before
we rotate() by $+90^\circ$!!

(write the sequence of matrices on the blackboard)



METHOD 1: (change numbers)

Translate(6,0);
Rotate(+45);



Translation → Rotation

NOW TRY: Translation → Rotation → Scale → Rotation

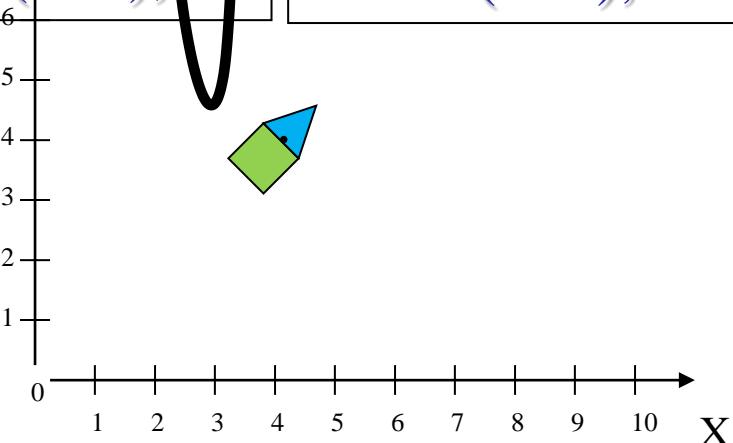
Ooh! That's !UGLY!

Translate(6,0);
Rotate(+45);

Rotate(-45);
Translate(-6,0);
Scale(0.5);
Translate(6,0);
Rotate(+45);

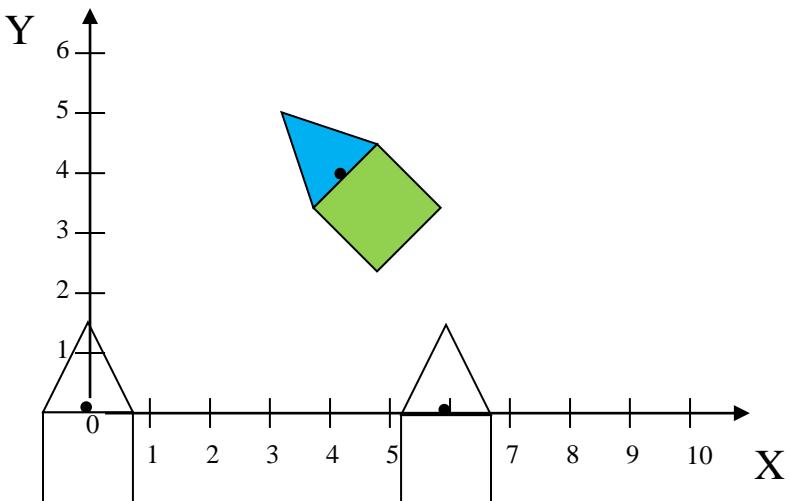
Rotate(-45);
Translate(-6,0);
Rotate(+90);
Translate(6,0);
Rotate(+45);

UGH! Yet using
'Method 1' functions
requires this sequence!



METHOD 1: (change numbers)

Translate(6,0);
Rotate(+45);



Translation → Rotation

NOW TRY: Translation → Rotation → Scale → Rotation

Ooh! That's !UGLY!

Translate(6,0);
Rotate(+45);

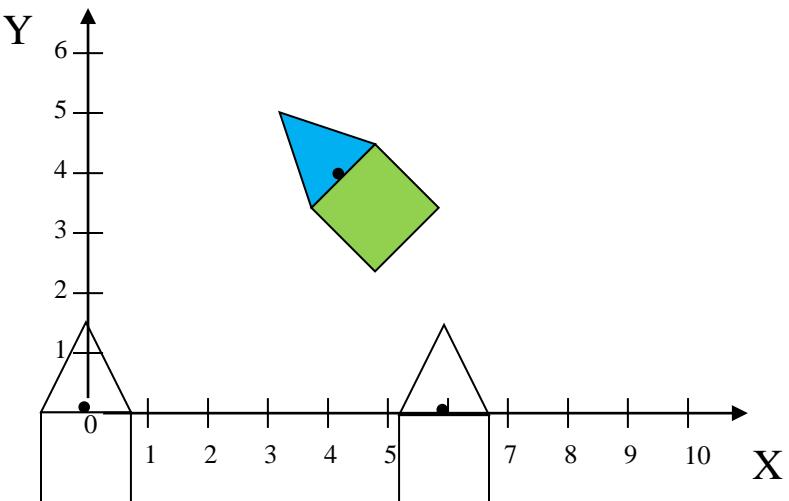
Rotate(-45);
Translate(-6,0);
Scale(0.5);
Translate(6,0);
Rotate(+45);

Rotate(-45);
Translate(-6,0);
Rotate(+90);
Translate(6,0);
Rotate(+45);

We must
--**UNDO** all previous transforms,
--Do the new transform, then
--**REDO** all previous transforms...
(Hmm. Can't we avoid that?)

METHOD 1: (change numbers)

Translate(6,0);
Rotate(+45);



Translation → Rotation

NOW TRY: Translation → Rotation → Scale → Rotation

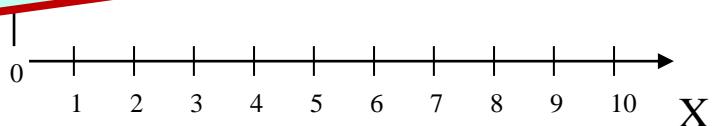
Ooh! That's !UGLY!

Translate(6,0);
Rotate(+45);

Rotate(-45);
Translate(-6,0);
Scale(0.5);
Translate(6,0);
Rotate(+45);

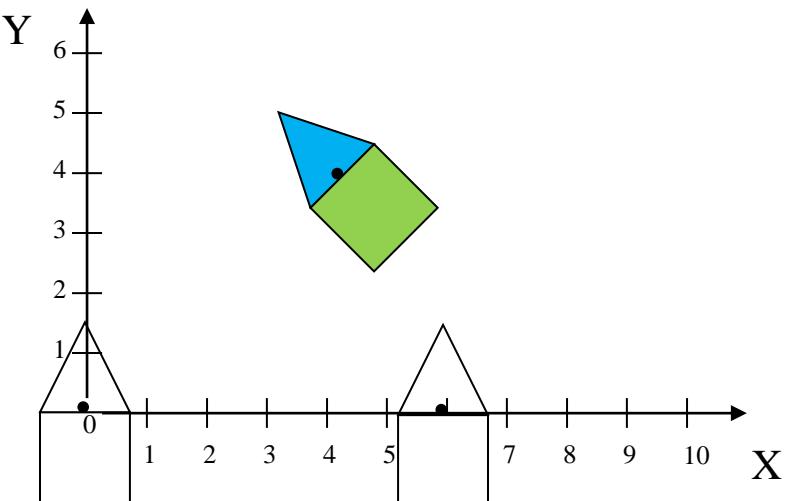
Rotate(-45);
Translate(-6,0);
Rotate(+90);
Translate(6,0);
Rotate(+45);

Hmmm – let's try to write that with math:



METHOD 1: (change numbers)

Translate(6,0);
Rotate(+45);



Translation → Rotation

NOW TRY: Translation → Rotation → Scale → Rotation

Ooh! That's !UGLY!

Translate(6,0);
Rotate(+45);

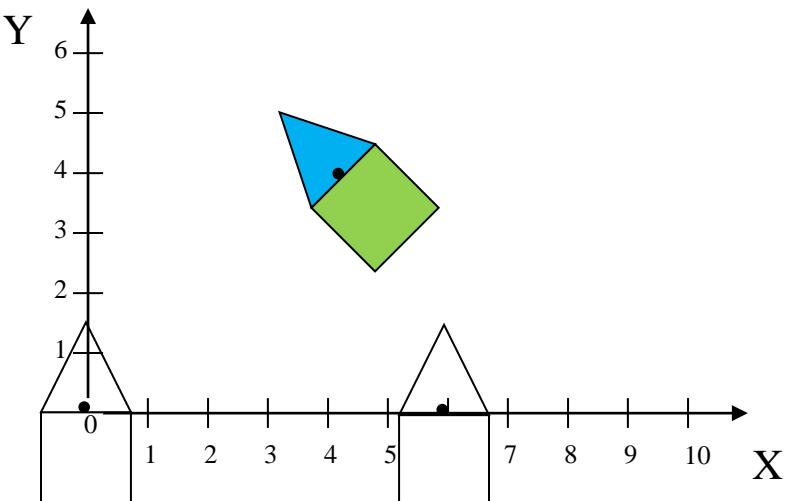
Rotate(-45);
Translate(-6,0);
Scale(0.5);
Translate(6,0);
Rotate(+45);

Rotate(-45);
Translate(-6,0);
Rotate(+90);
Translate(6,0);
Rotate(+45);

4x4 Matrix \mathbf{M} transforms point \mathbf{p} into new point \mathbf{q} :
$$\mathbf{q} = \mathbf{Mp}$$

METHOD 1: (change numbers)

Translate(6,0);
Rotate(+45);



Translation → Rotation

NOW TRY: Translation → Rotation → Scale → Rotation

Ooh! That's !UGLY!

Translate(6,0);
Rotate(+45);

$$q = RTp$$

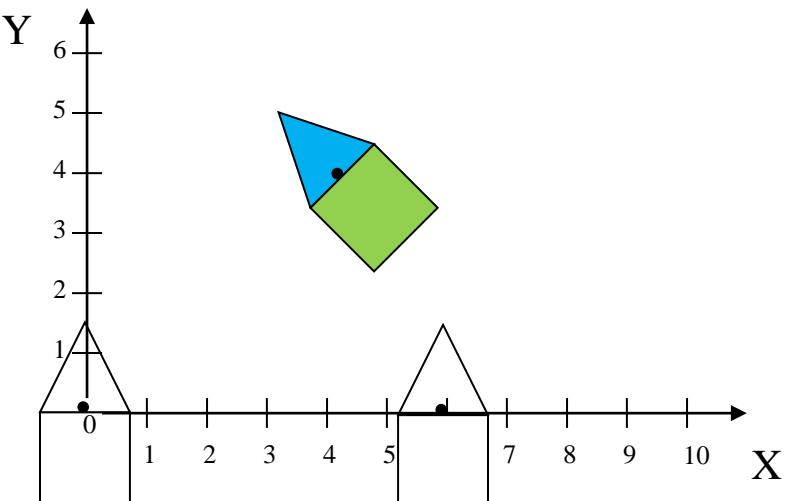
Rotate(-45);
Translate(-6,0);
Scale(0.5);
Translate(6,0);
Rotate(+45);

Rotate(-45);
Translate(-6,0);
Rotate(+90);
Translate(6,0);
Rotate(+45);

Our first two transforms arrange matrices like this:
$$q = RTp$$

METHOD 1: (change numbers)

Translate(6,0);
Rotate(+45);



Translation → Rotation

NOW TRY: Translation → Rotation → Scale → Rotation

Ooh! That's !UGLY!

Translate(6,0);
Rotate(+45);

$$q = RTp$$

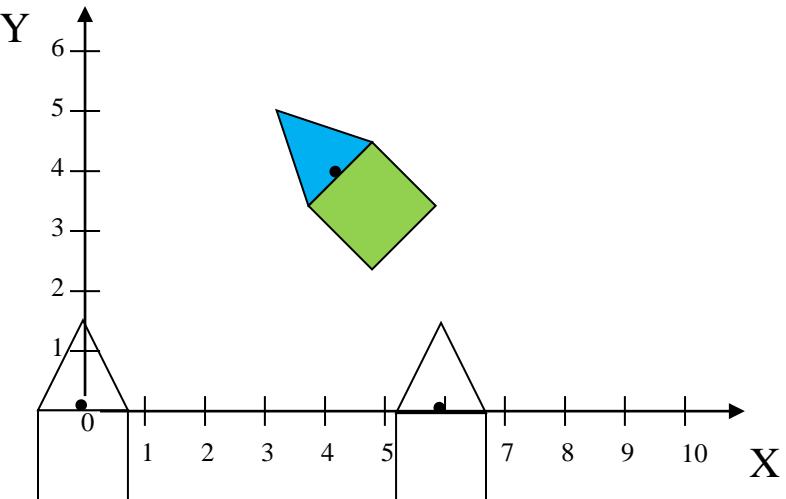
Rotate(-45);
Translate(-6,0);
Scale(0.5);
Translate(6,0);
Rotate(+45);

Rotate(-45);
Translate(-6,0);
Rotate(+90);
Translate(6,0);
Rotate(+45);

The next two transforms are their INVERSES...
(be careful to get correct order of matrices here)
$$q = T^{-1}R^{-1}RTp$$

METHOD 1: (change numbers)

Translate(6,0);
Rotate(+45);



Translation → Rotation

NOW TRY: Translation → Rotation → Scale → Rotation

Ooh! That's !UGLY!

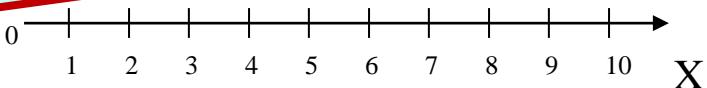
Translate(6,0);
Rotate(+45);

$$\mathbf{q} = \mathbf{R}\mathbf{T}\mathbf{p}$$

Rotate(-45);
Translate(-6,0);
Scale(0.5);
Translate(6,0);
Rotate(+45);

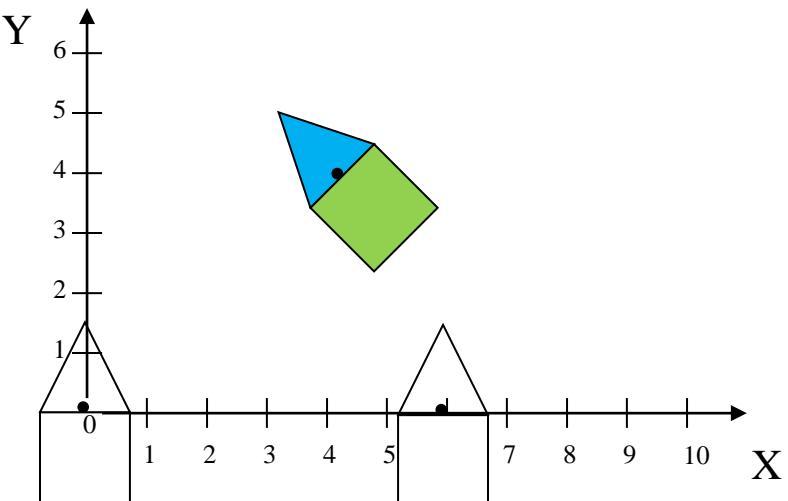
Rotate(-45);
Translate(-6,0);
Rotate(+90);
Translate(6,0);
Rotate(+45);

Then apply the next three transforms;
 $\mathbf{q} = \mathbf{R}\mathbf{T}\mathbf{S}\mathbf{T}^{-1}\mathbf{R}^{-1}\mathbf{R}\mathbf{T}\mathbf{p}$



METHOD 1: (change numbers)

Translate(6,0);
Rotate(+45);



Translation → Rotation

NOW TRY: Translation → Rotation → Scale → Rotation

Ooh! That's !UGLY!

~~Translate(6,0);~~
~~Rotate(+45);~~

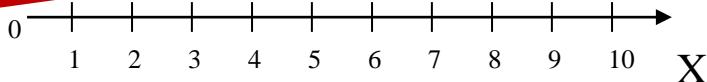
$$\mathbf{q} = \mathbf{R}\mathbf{T}\mathbf{p}$$

~~Rotate(45);~~
~~Translate(-6,0);~~
~~Scale(0.5);~~
~~Translate(6,0);~~
~~Rotate(+45);~~

Rotate(-45);
Translate(-6,0);
Rotate(+90);
Translate(6,0);
Rotate(+45);

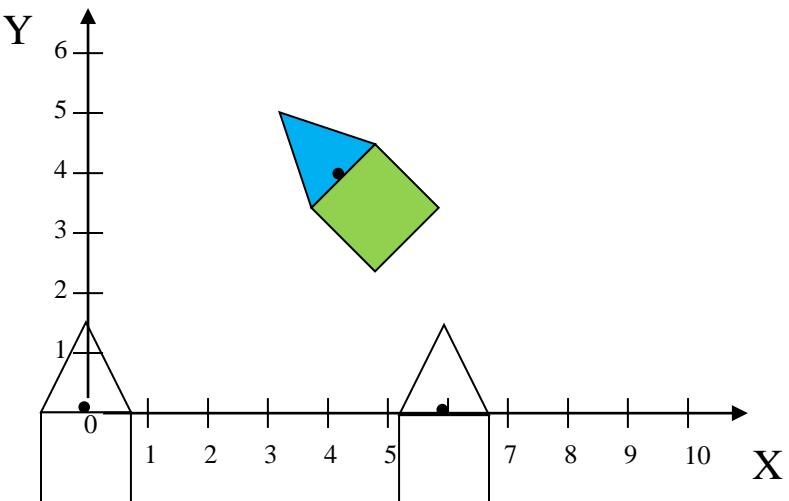
*SIMPLIFY – we're cancelling transforms here ...
Hey, Look! It's just doing the S transform FIRST !!*

$$\mathbf{q} = \mathbf{R}\mathbf{T}\mathbf{S}(\mathbf{T}^{-1}(\mathbf{R}^{-1}\mathbf{R})\mathbf{T})\mathbf{p}$$



METHOD 1: (change numbers)

Translate(6,0);
Rotate(+45);



Translation → Rotation

NOW TRY: Translation → Rotation → Scale → Rotation

Ooh! That's !UGLY!

~~Translate(6,0);~~
~~Rotate(+45);~~

$$q = \mathbf{R}\mathbf{T}p$$

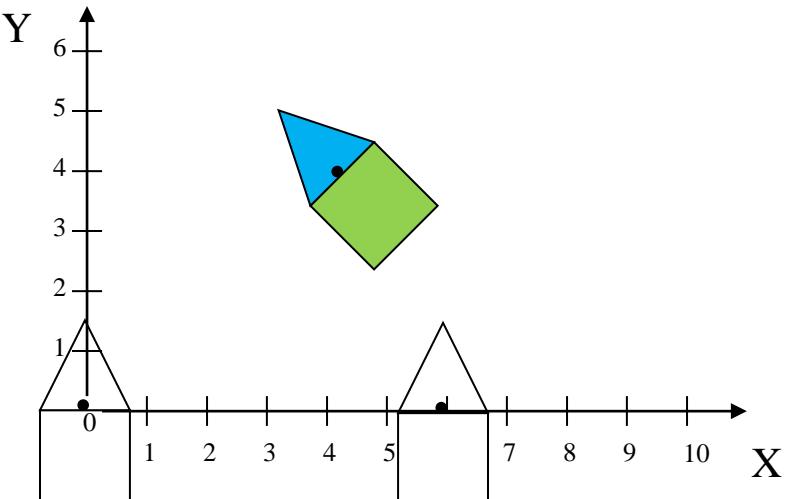
~~Rotate(45);~~
~~Translate(-6,0);~~
~~Scale(0.5);~~
~~Translate(6,0);~~
~~Rotate(+45);~~

Rotate(-45);
Translate(-6,0);
Rotate(+90);
Translate(6,0);
Rotate(+45);

YES; that's right -- we apply scale transform first...
 $q = \mathbf{R}\mathbf{T}\mathbf{S} p$

METHOD 1: (change numbers)

Translate(6,0);
Rotate(+45);



Translation → Rotation

NOW TRY: Translation → Rotation → Scale → Rotation

Ooh! That's !UGLY!

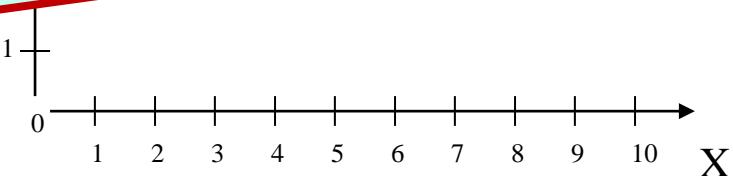
Translate(6,0);
Rotate(+45);

$$\mathbf{q} = \mathbf{R}\mathbf{T}\mathbf{p}$$

Rotate(-45);
Translate(-6,0);
Scale(0.5);
Translate(6,0);
Rotate(+45);

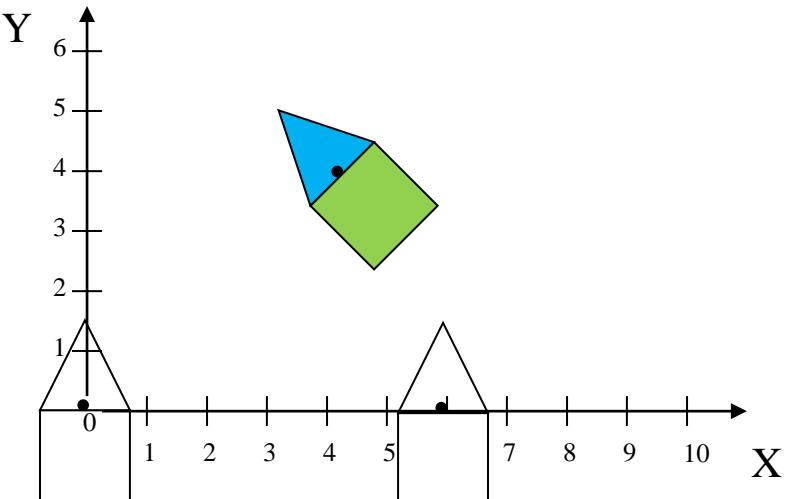
Rotate(-45);
Translate(-6,0);
Rotate(+90);
Translate(6,0);
Rotate(+45);

Now do all the rest:
 $\mathbf{q} = \mathbf{R}\mathbf{T}\mathbf{R}_2\mathbf{T}^1\mathbf{R}^{-1} \mathbf{R}\mathbf{T}\mathbf{S} (\mathbf{T}^{-1}(\mathbf{R}^{-1}\mathbf{R})\mathbf{T})\mathbf{p}$



METHOD 1: (change numbers)

Translate(6,0);
Rotate(+45);



Translation → Rotation

NOW TRY: Translation → Rotation → Scale → Rotation

Ooh! That's !UGLY!

~~Translate(6,0);~~
~~Rotate(+45);~~

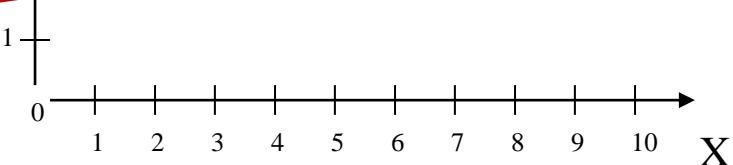
$$q = RTp$$

~~Rotate(45);~~
~~Translate(6,0);~~
~~Scale(0.5);~~
~~Translate(6,0);~~
~~Rotate(+45);~~

~~Rotate(-45);~~
~~Translate(-6,0);~~
~~Rotate(+90);~~
~~Translate(6,0);~~
~~Rotate(+45);~~

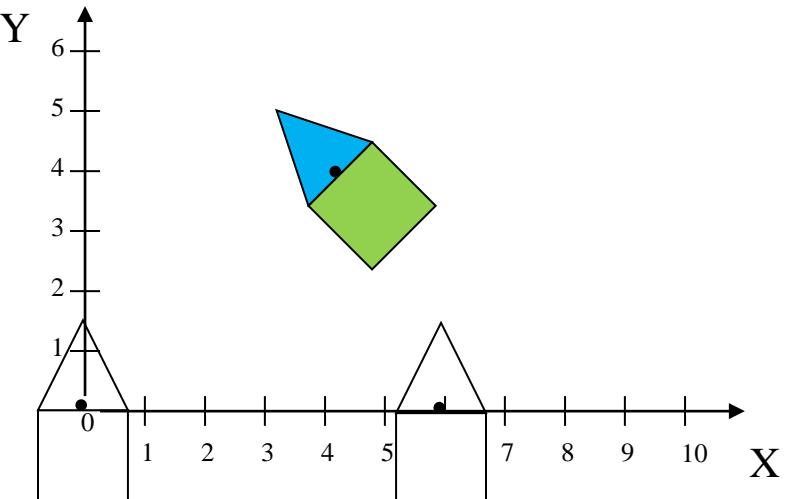
SIMPLIFY: again, cancelling transforms ...

$$q = RTR_2(T^1(R^{-1}R)T)S(T^1(R^{-1}R)T)p$$



METHOD 1: (change numbers)

Translate(6,0);
Rotate(+45);



Translation → Rotation

NOW TRY: Translation → Rotation → Scale → Rotation

Ooh! That's !UGLY!

~~Translate(6,0);~~
~~Rotate(+45);~~

$$q = \mathbf{R}\mathbf{T}p$$

~~Rotate(45);~~
~~Translate(6,0);~~
~~Scale(0.5);~~
~~Translate(6,0);~~
~~Rotate(+45);~~

~~Rotate(-45);~~
~~Translate(-6,0);~~
~~R₂otate(+90);~~
~~Translate(6,0);~~
~~Rotate(+45);~~

$$q = \mathbf{R}\mathbf{T}\mathbf{R}_2\mathbf{S} p$$

Weird! We must add on new 'Body-centered' steps by arranging transforms in 'reverse' order, as if the last transforms were done first (!).

How can we design 3D Transform fcns() to minimize confusions?

METHOD 1: (the obvious one)

--Keep one, fixed coordinate system, and
--at each successive transform applied,

Create *new coordinate values*

(the xyz) for each point

Plot them

$$\mathbf{q} = \mathbf{R}\mathbf{T}\mathbf{R}_2\mathbf{S} \mathbf{p}$$

Weird! **HOW** can we interpret this
geometrically? Without endless confusion?

METHOD 2: (the not-so-obvious)

--Keep **coordinate numbers** of each point **fixed**;
(they never change). Instead,

--For each successive transform applied,

Create a *new coordinate system*:

and transform it from the previous one. (!)

--Plot unchanged, original coordinate **numbers**,
but draw them in the newest coordinate **system**.

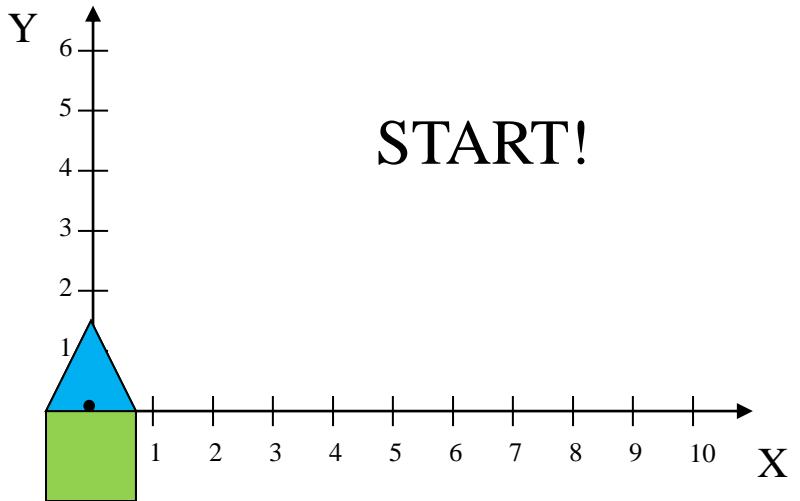
On-screen RESULT:

SAME as Method 1, but **reversed transforms!**
(strange at first, but MUCH EASIER
for assembling complex jointed objects)

METHOD 2: A different view

aka 'Local Frames' or
'transformed drawing axes'

Translate()
by $x=6$, $y=0$
then
Rotate() by 45°



Translation → Rotation

Rotate() by 45°
then
Translate()
by $x=6$, $y=0$



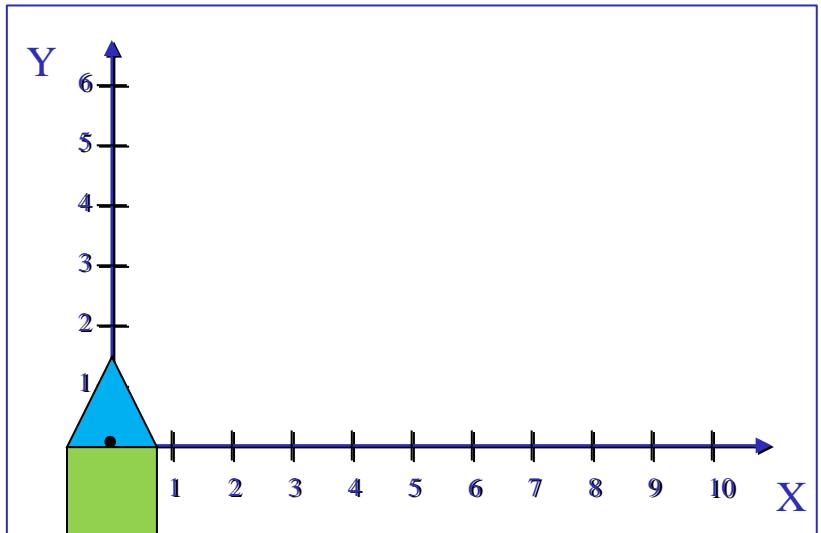
Rotation → Translation

I'll leave this case for YOU to do!

METHOD 2 DEMO:

2a) Copy: new coord system

Translate()
by $x=6$, $y=0$
then
Rotate() by 45°



Translation → Rotation

METHOD 2 DEMO:

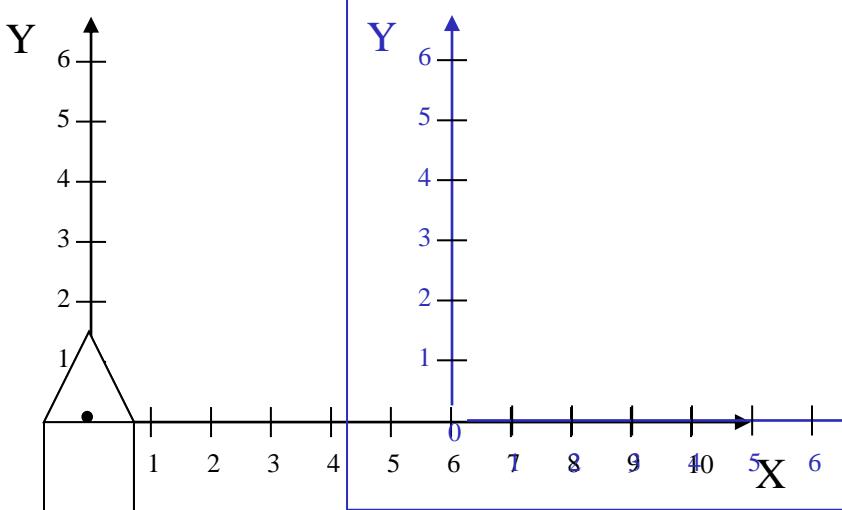
2b) Transform new coord sys as measured from old coord sys

Translate()

by $x=6, y=0$

then

Rotate() by 45°



Translation → Rotation

METHOD 2 DEMO:

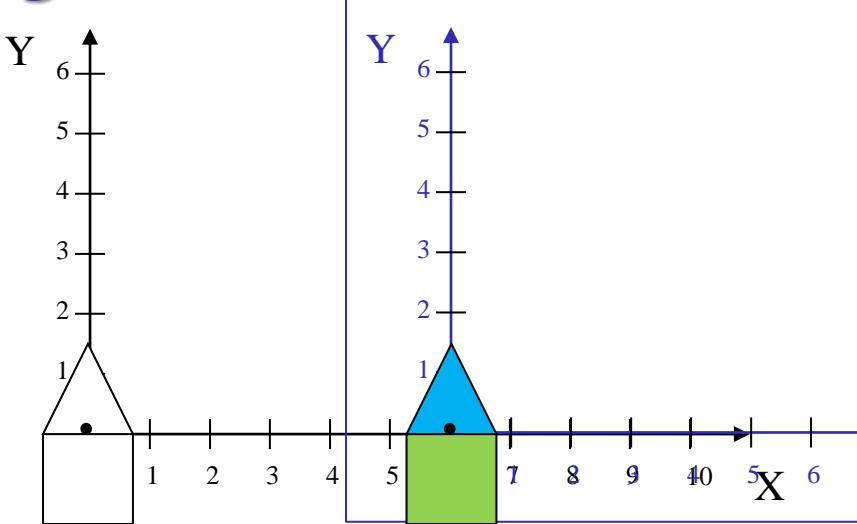
2c) Draw: in new coord. system
with *unchanged* vertex coords

Translate()

by $x=6, y=0$

then

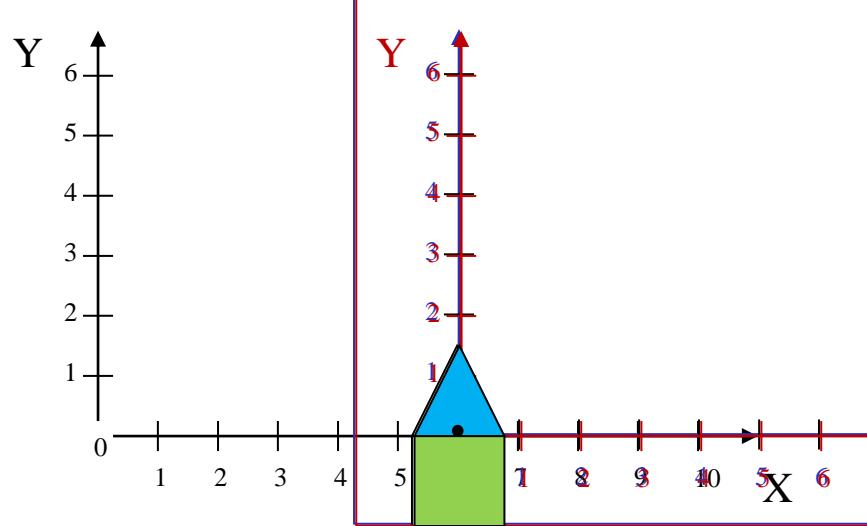
Rotate() by 45°



Translation → Rotation

METHOD 2 DEMO: 2d) Copy: new coord system

Translate()
by $x=6$, $y=0$
then
Rotate() by 45°

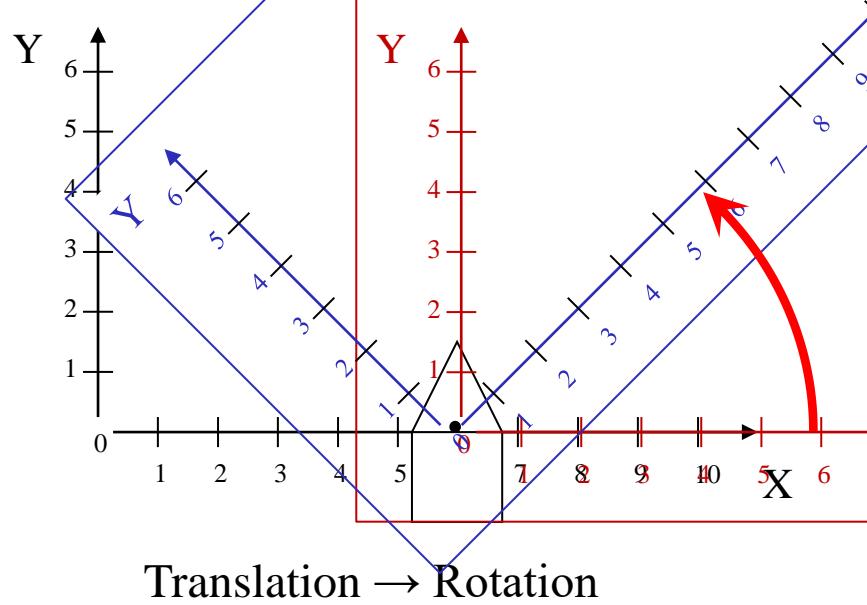


Translation → Rotation

METHOD 2 DEMO:

2b) Transform new coord sys as measured from old coord sys

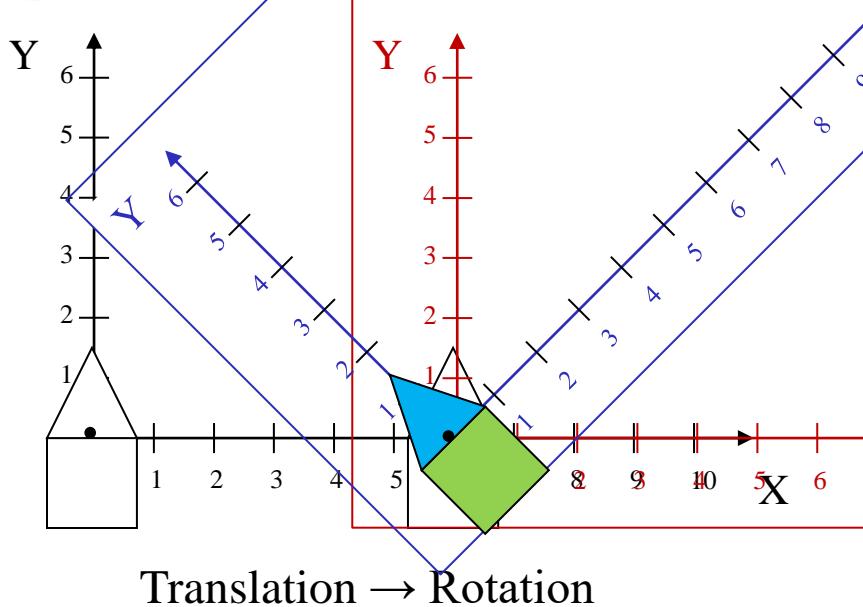
Translate()
by $x=6$, $y=0$
then
Rotate() by 45°



METHOD 2 DEMO:

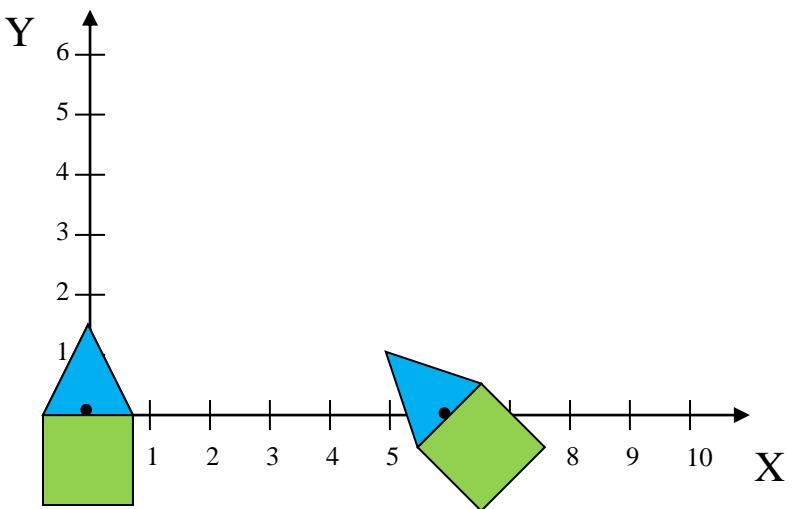
2c) Draw: in new coord system with *unchanged* vertex coords

Translate()
by $x=6$, $y=0$
then
Rotate() by 45°



METHOD 2 DEMO: result: body-centered; 'intuitive'

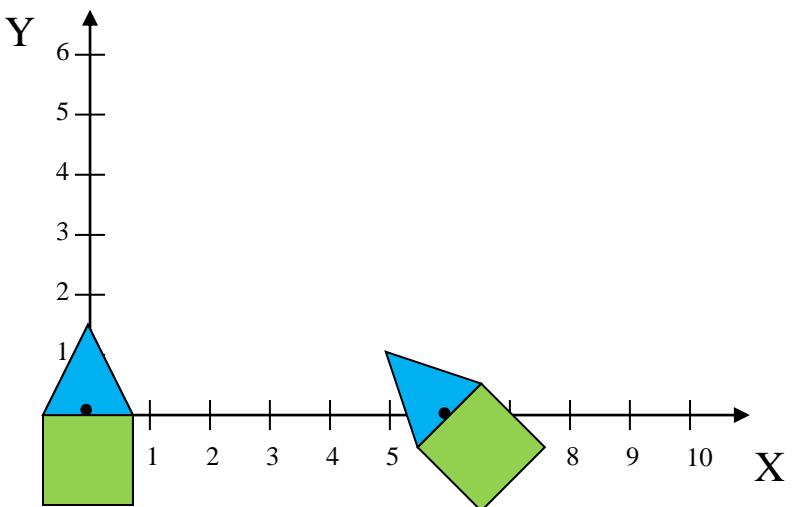
Translate() by
 $x=6, y=0$
then
Rotate() by 45°



Translation → Rotation

METHOD 2 DEMO: result: body-centered; 'intuitive'

Translate() by
 $x=6$, $y=0$
then
Rotate() by 45°



Translation → Rotation

Hmm. Yields the same result as
reversed-order steps in Method 1 ...
(e.g. Method 1 rotate-then-translate)

How can we design 3D Transform fcns() to minimize confusions?

METHOD 1: (the obvious one)

--Keep one, fixed coordinate system, and
--at each successive transform applied,

Create *new coordinate values*

(the xyz) for each point or vertex.

--Plot the changed coordinates
in the original coordinate system.

(Simple, obvious, intuitive, and bad).

$$v_1 = [M_0]v_0;$$

$$v_2 = [M_1]v_1; \rightarrow v_2 = [M_1][M_0]v_0$$



METHOD 2: (the not-so-obvious)

--Keep coordinate numbers of each point fixed;
(they never change). Instead,

--For each successive transform applied,

Creates a *new coordinate system*:

and transform it from the previous one.

--Plot the unchanged original coordinate numbers,
but using the newest coordinate system.

--**?HOW?** Surprisingly simple:

reversed-order matrix multiply:

\rightarrow

$$v_2 = [M_0][M_1]v_0$$



(seems strange at first, but MUCH EASIER
when we assemble complex jointed objects!)

?METHOD 1, 2 are Inverses?!?

Yes! You're right!!

So how, exactly, should our transform-making functions concatenate matrices?

- Method 1:

$$[M] = [M_{\text{new}}][M_{\text{old}}]$$

- Method 2:

$$[M] = [M_{\text{old}}][M_{\text{new}}]$$

Can you see why?

(RECALL: in Method 1, to get Method 2 result you must 'undo' M_{old} , apply M_{new} , and 're-do' M_{old} again...)

METHOD 1, 2 and 3... which is better?

Yes, it's true.

So what's the difference?

translate(), rotate()

and scale() are

convenience methods.

'translate()',
'rotate()' and
'scale()' methods do this in
OpenGL, WebGL &
other Graphics APIs

- Method 1:

$$[M] = [M_{\text{new}}][M_{\text{old}}]$$

- Method 2:

$$[M] = [M_{\text{old}}][M_{\text{new}}]$$

Why use **this** matrix order?

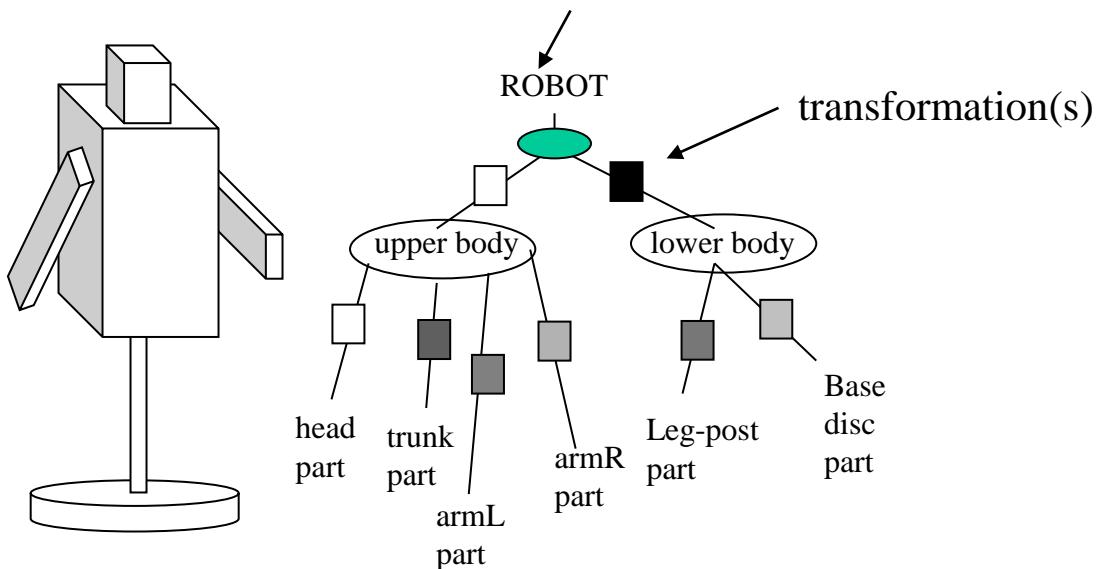
so that users can build complex, jointed objects

by **'body-centered' transformations**.

(See 'robot arm' demo)

How Should We Organize MANY Geometric Transformations (T,R,S) To Build Complex Jointed Objects?

An “is composed of” hierarchy



- **Answer:**
A TREE of TRANSFORMATIONS,
PARTS, & THEIR ATTRIBUTES
- An “is-composed-of” hierarchy,
arranged as a DAG (Directed Acyclic Graph)
- that performs a '**global-to-local**' decomposition of a jointed object made of individual parts, such as this robot...
- The completed tree we will devise is called a

“Scene-Graph”

Scene Graph Assembly (1/4)

- A *scene graph* is a directed acyclic graph (DAG) that describes 3D scene + cameras + lights.
(we'll add cameras & lights later)
- Examples:
 - Open Scene Graph (used in the UIUC Cave)
 - Sun's Java3D™
 - X3D ™ (VRML ™ was a precursor to X3D)

Scene graph edges just connect nodes

Scene graph node types:

- **Object Nodes** (cubes, sphere, cone, triangle etc.) describe re-usable shapes as fixed sets of connected vertices (default: unit size, centered at origin)
- **Transform Nodes** describe parameterized T,R,S matrices used to 'pose' the nodes below it
- **Group Nodes** describe *collections* that share all the attributes & transforms above it in the scene graph
Helps with 'instancing': modified copies of objects
- **Drawing-Attribute Nodes** (Apply a new color, new line width, shading type, texture map, etc.)
Describes how to render the nodes below us.
We won't use these ...

Scene Graph Assembly (2/4)

- A *scene graph* is a directed acyclic graph (DAG) that describes 3D scene + cameras + lights.
(we'll add cameras & lights later)
- Examples:
 - Open Scene Graph (used in the UIUC Cave)
 - Sun's Java3D™
 - X3D ™ (VRML ™ was a precursor to X3D)

Scene graph edges just connect nodes

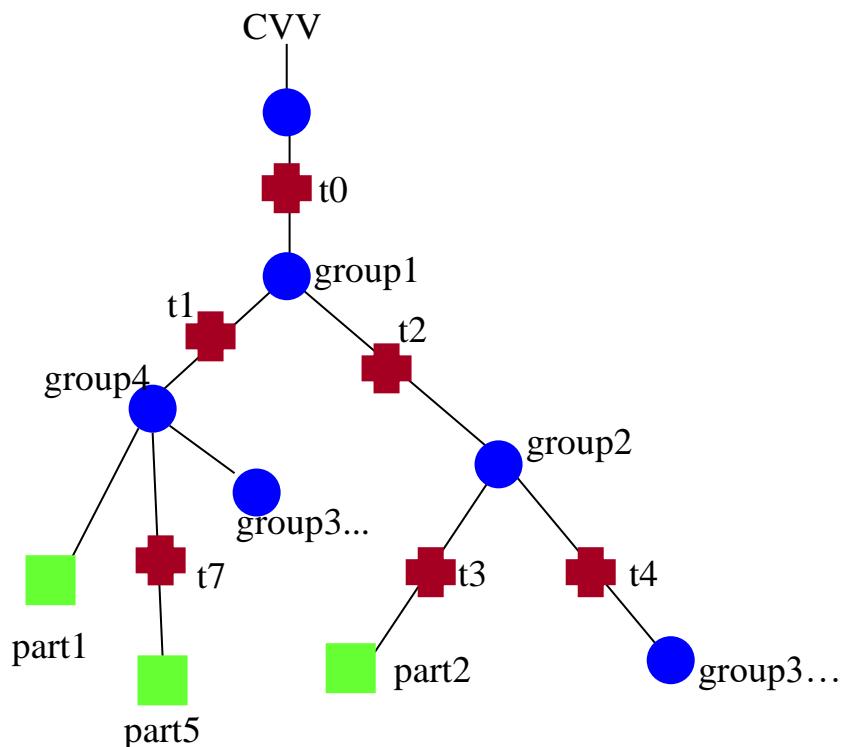
Scene graph node types:

- **Part Nodes** (cubes, sphere, cone, triangle etc.) contain fixed sets of connected vertices (default: unit size, centered at origin).
ONE parent only, NO children!
- **Transform Nodes** contain JUST ONE transform matrix. Usually a T, R, or S matrix used to 'pose' nodes below it. ONE parent only, ONE child only.
- **Group Nodes** create *collections* of transformed parts by grouping together child nodes.
ONE parent only; ONE OR MORE child nodes.

Scene graph ROOT? always 'CVV' group node -- the +/- 1 cube that fills the the HTML-5 Canvas!

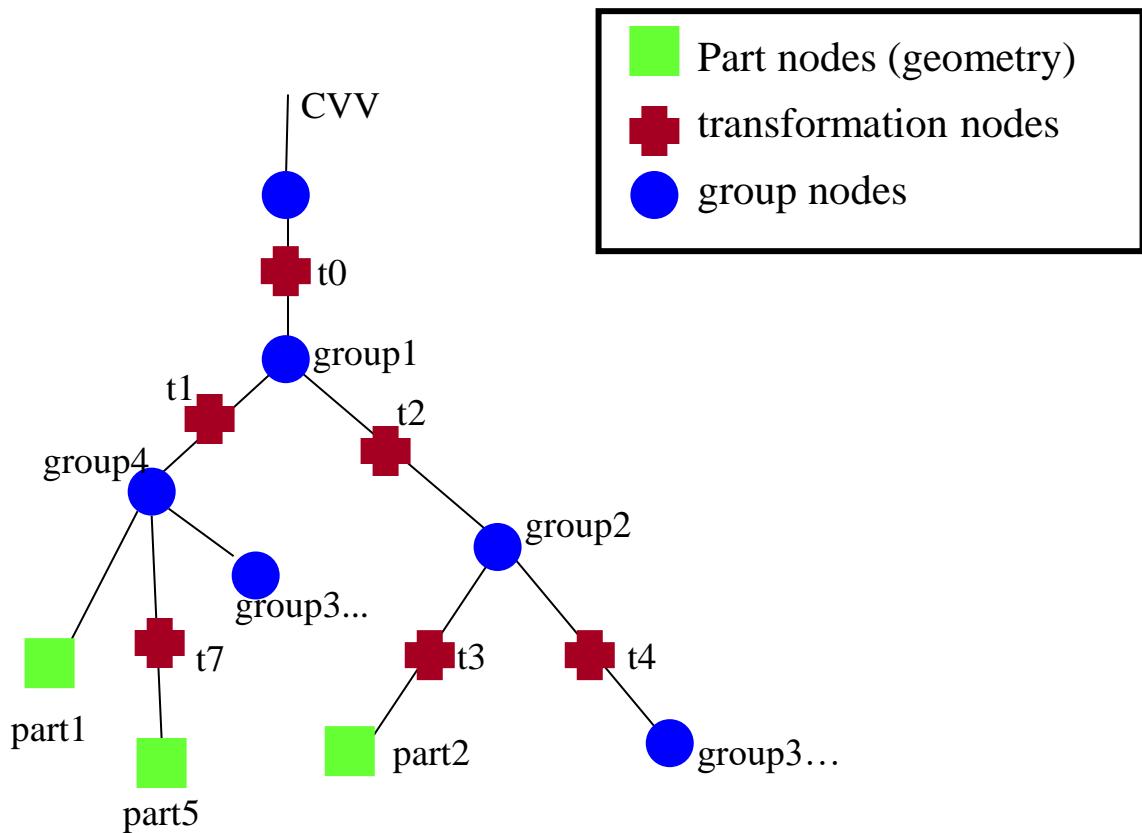
Scene Graph Assembly (3/4)

- **Part Nodes** (cubes, sphere, cone, triangle etc.) contain fixed sets of connected vertices.
ONE parent, NO child
- **Transform Nodes** contain
JUST ONE transform matrix.
ONE parent, ONE child
- **Group Nodes** create *collections* of transformed parts by grouping together child nodes.
ONE parent, ≥ 1 child.



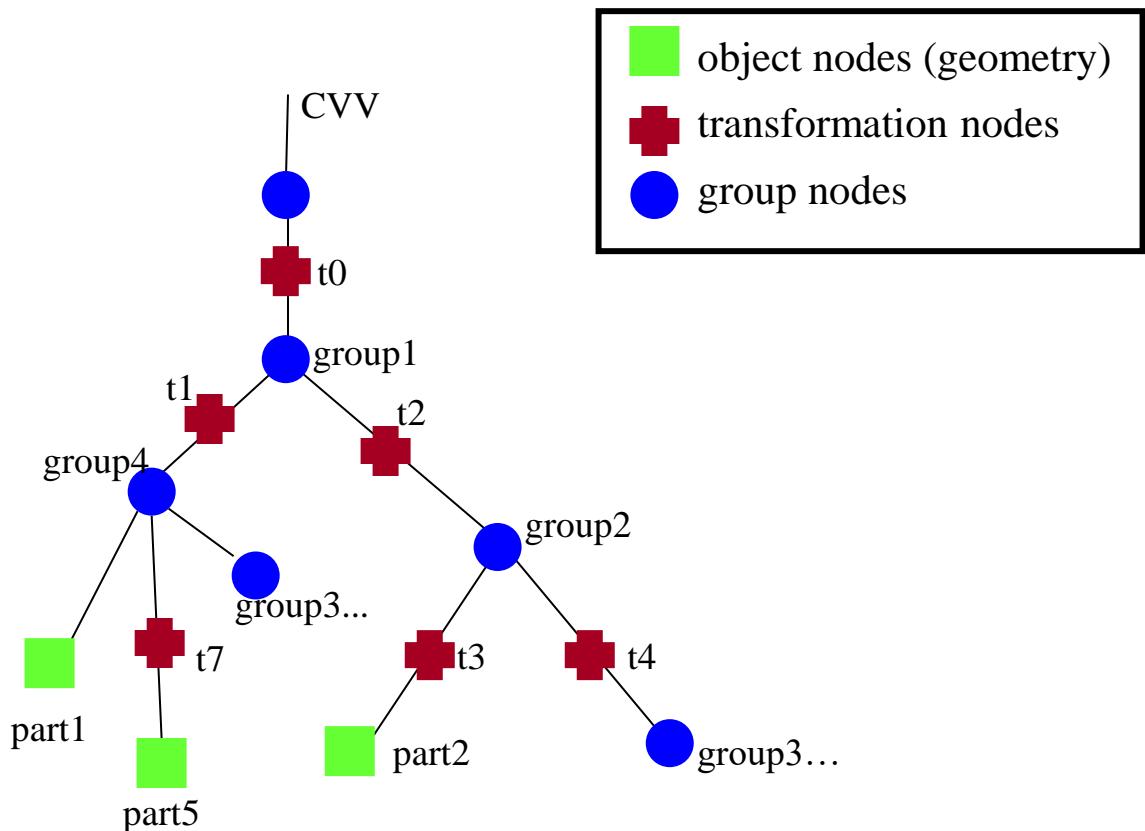
Scene Graph Assembly (4/4)

- **Rendering with WebGL commands:**
 - **Traverse graph** from root (top) to leaf (bottom) in **depth-first order**
 - Drop into Group Node? Call PushMatrix()
 - Rise to prev. Group Node? Call PopMatrix().
 - Arrive at Part Node? Call it's '**drawMe ()** fcn.



Scene Graph Duality:

- **WebGL commands** traverse **top→bottom**
Vertex coordinates traverse **bottom→top**
- from memory to screen; through matrices
- Vertices → modelMatrix → CVV →
HTML-5 canvas → display window pixels.



Scene Graph Duality:

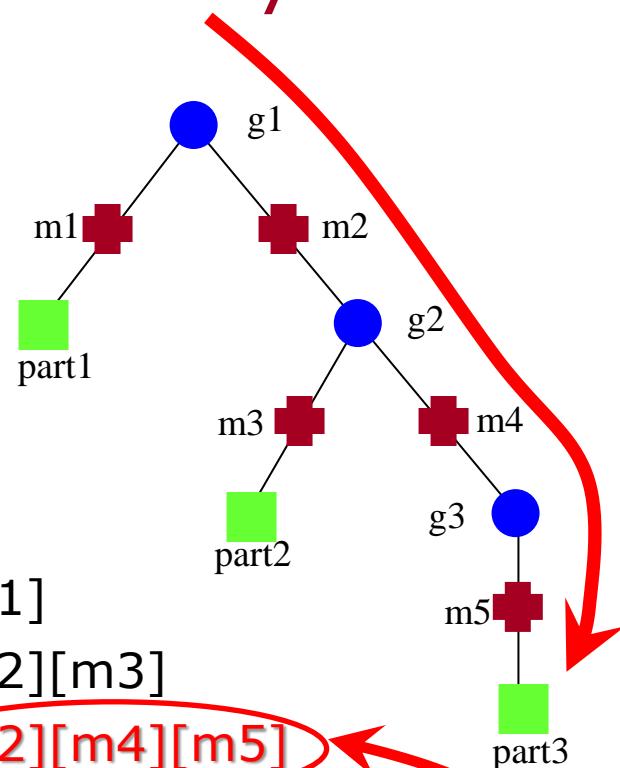
- Example:

g: group nodes

m: matrices of transform nodes

o: object nodes

CTM: composed transform. matrix



- to draw part o_1 , CTM = $[m_1]$
- to draw part o_2 , CTM = $[m_2][m_3]$
- to draw part o_3 , CTM = $\boxed{[m_2][m_4][m_5]}$

To convert vertex coordinate values v in **part3** to its 'world' or 'root' coordinates values r :

$$r = [m_2][m_4][m_5]v$$

To build that matrix in WebGL / OpenGL:

```
modelMatrix.setIdentity();
modelMatrix.apply-the-M2-transform();
modelMatrix.apply-the-M4-transform();
modelMatrix.apply-the-M5-transform();
draw-the-part3-object()
```

“Reversed!”
due to
Matrix Duality

DUALITY: what does WebGL Use?

~~METHOD 1:~~ (the obvious one)

- Keep one, fixed coordinate system, and
- at each successive transform applied,

Create *new coordinate values*

(the xyz) for each point or vertex.

--Plot the changed coordinates
in the original coordinate system.

(Simple, obvious, intuitive, and bad)

METHOD 2: (the not-so-obvious)

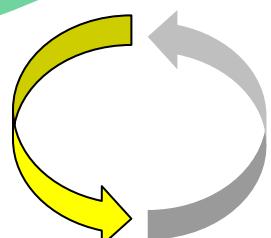
--Keep coordinate numbers of each point fixed;
(they never change). Instead,

--For each successive transform applied,

Create a *new coordinate system*:

and then transform it from the previous one.

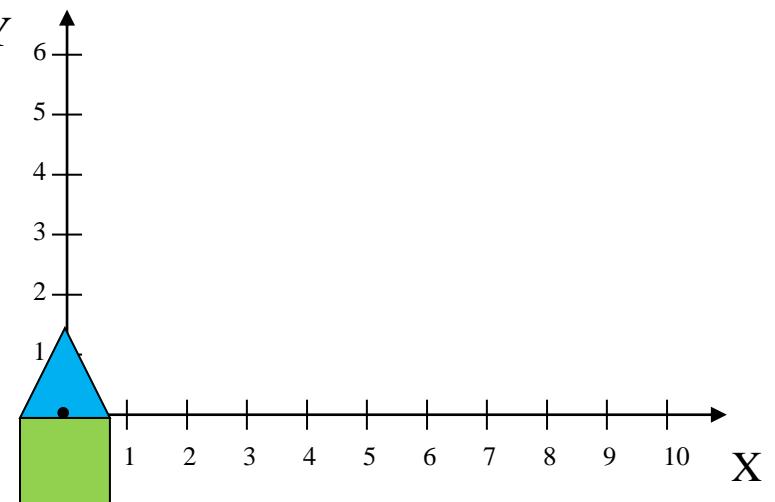
--Plot the unchanged original coordinate numbers,
but draw them in the newest coordinate system.



END

METHOD 2: Try the *other* command order:

Rotate() by 45°
then
Translate() by
 $x=6, y=0$

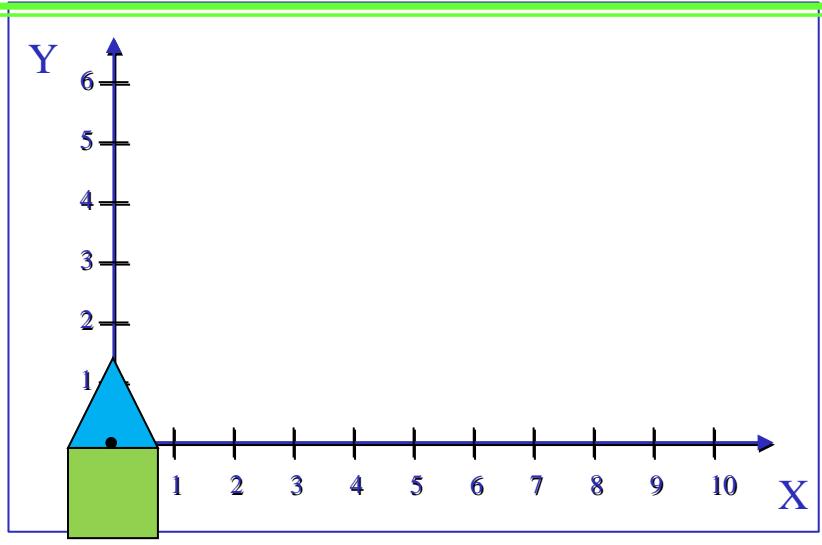


Rotation → Translation

METHOD 2:

2a) Copy: new coord system

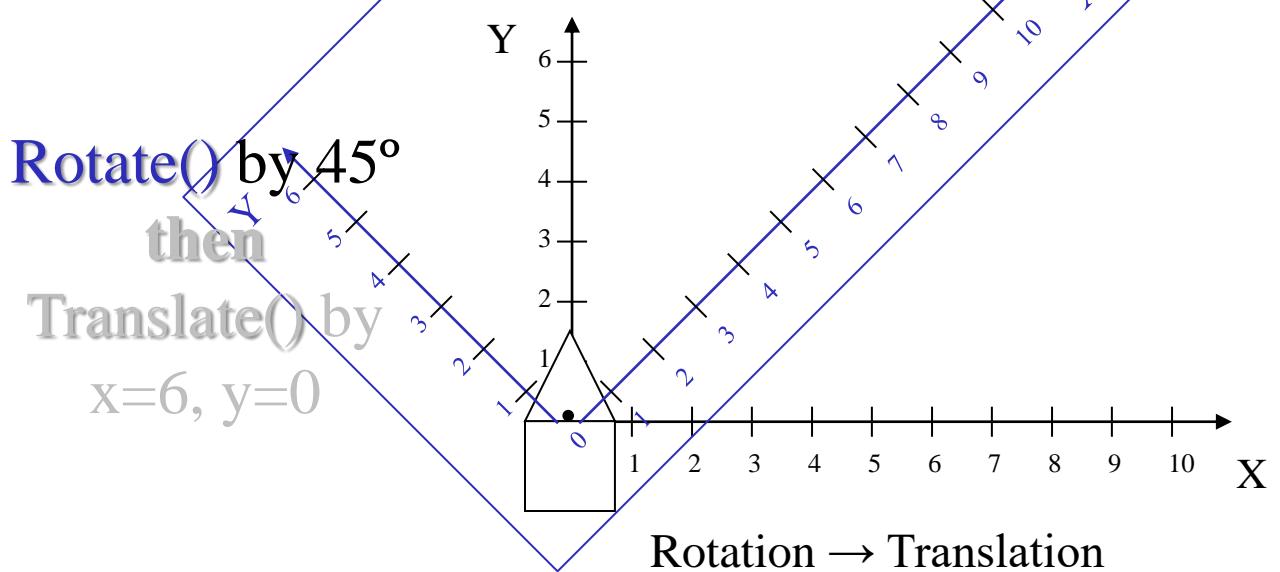
Rotate() by 45°
then
Translate() by
 $x=6, y=0$



Rotation \rightarrow Translation

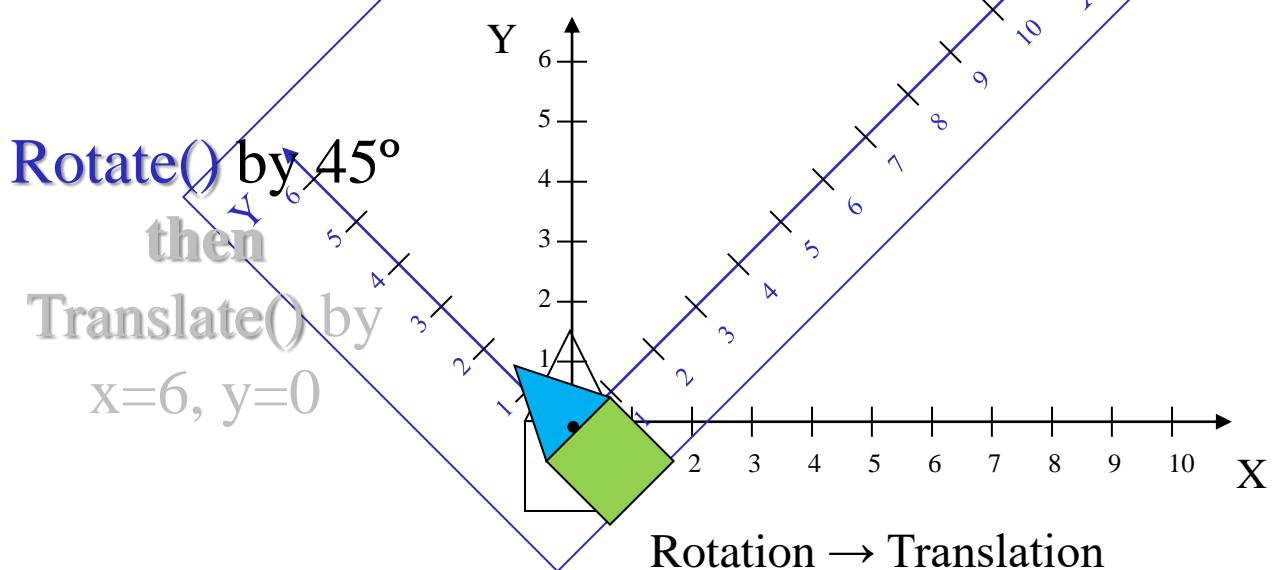
METHOD 2:

2b) Transform new coord sys as measured from old coord sys



METHOD 2:

2c) Draw: in new coord system
with *unchanged* vertex coords



METHOD 2:

2a) Copy: new coord system

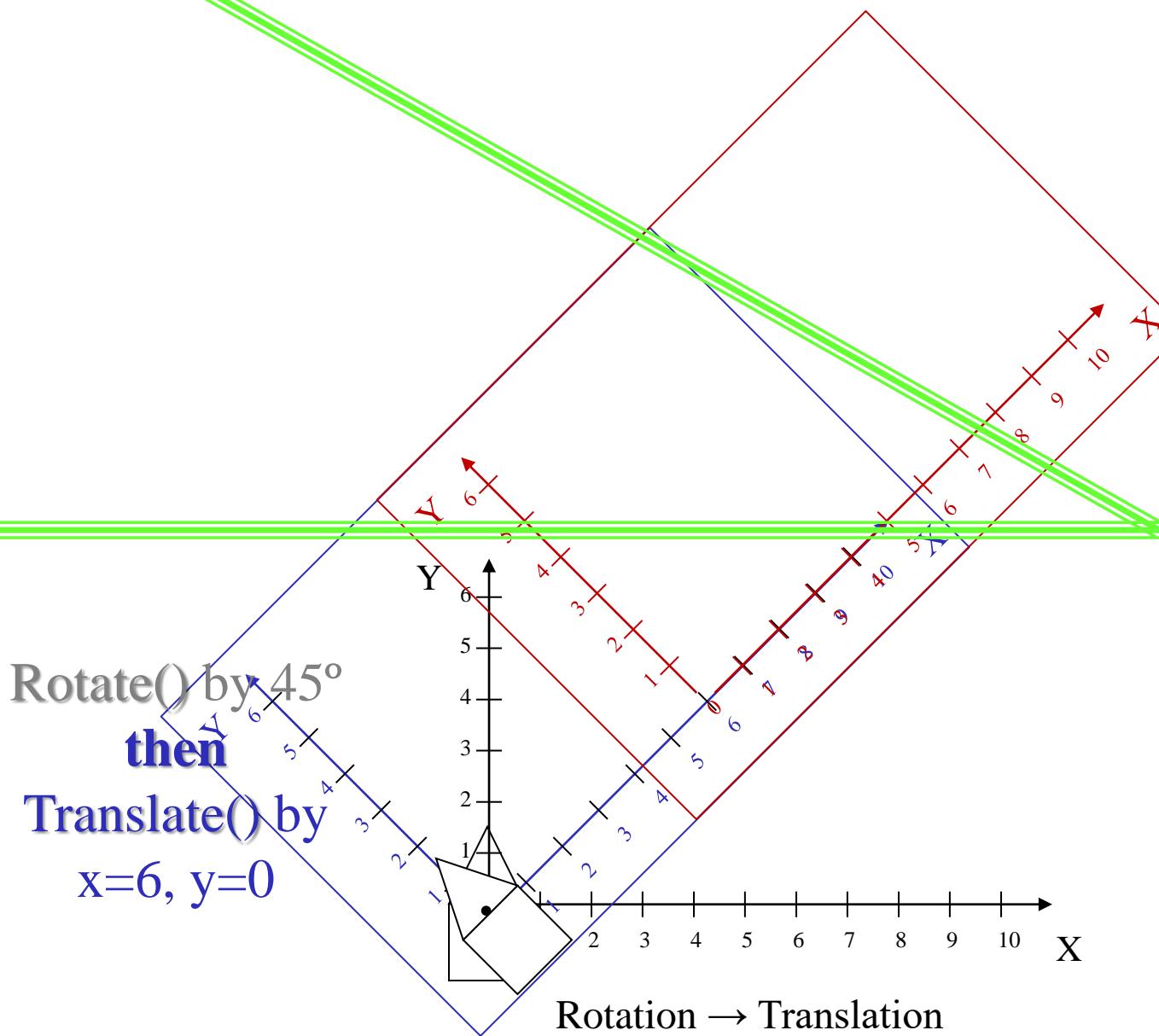
Rotate() by 45°
then

Translate() by
 $x=6, y=0$

Rotation \rightarrow Translation

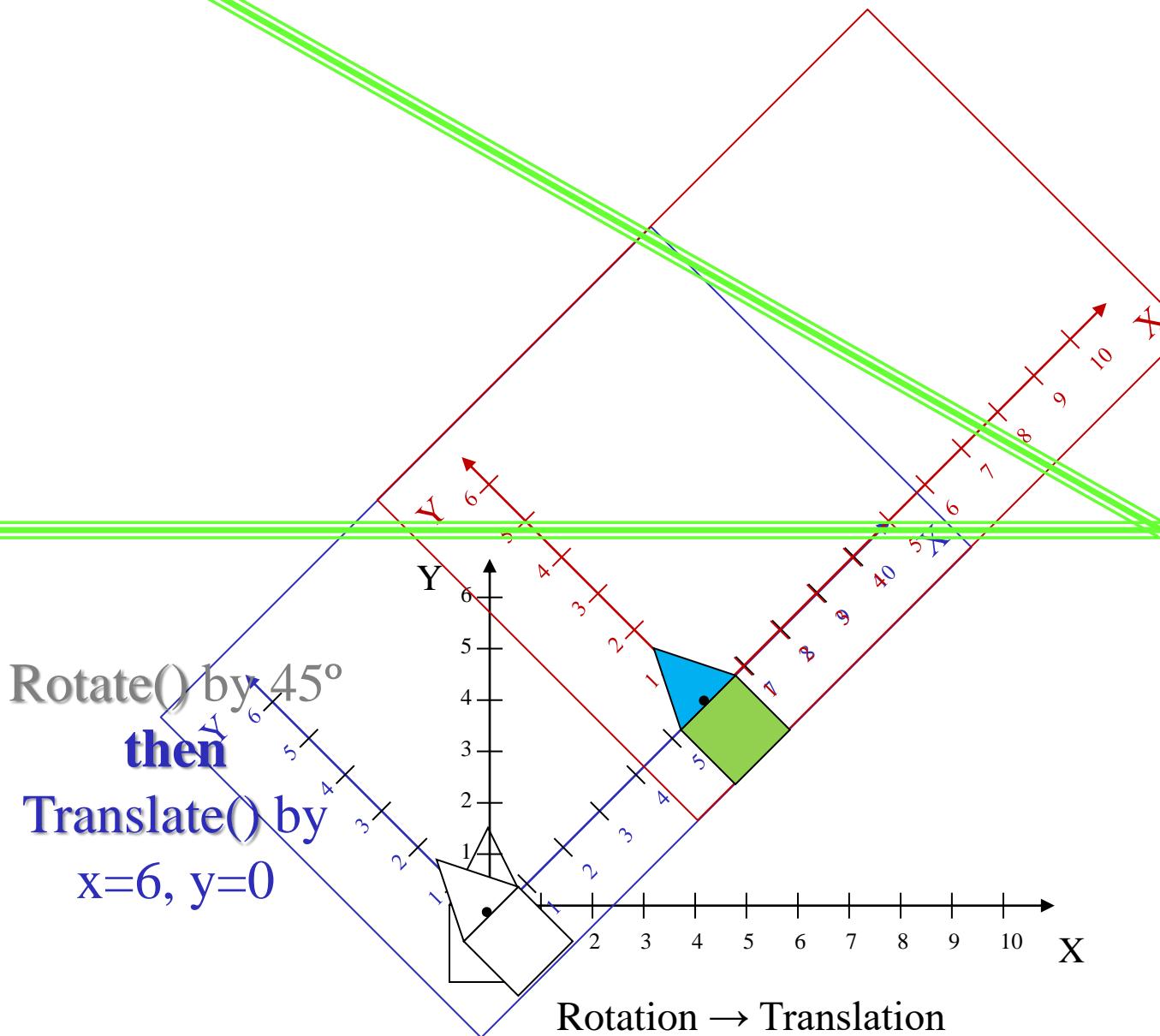
METHOD 2:

2b) Transform new coord sys as measured from old coord sys



METHOD 2:

2c) Draw: in new coord system
with *unchanged* vertex coords



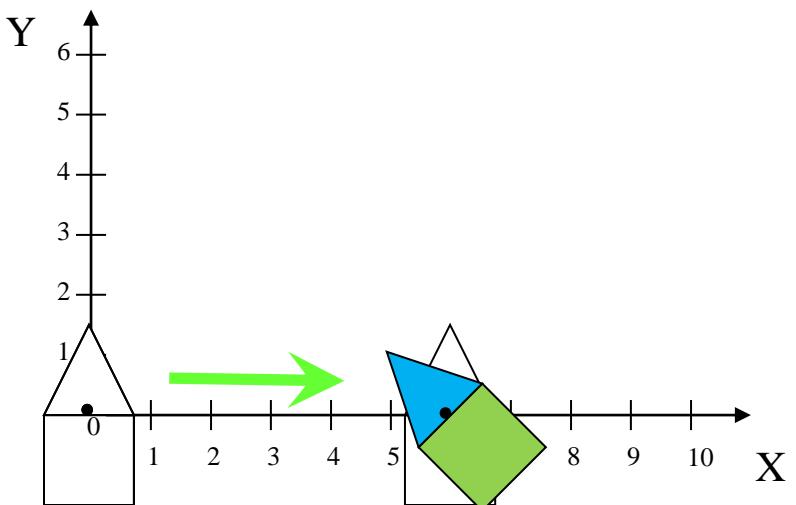
COMPARE 'Method 2' DEMOS: Still NOT Commutative, but actions not tied to fixed origin

Translate by

$$x=6, y=0$$

then

Rotate by 45°



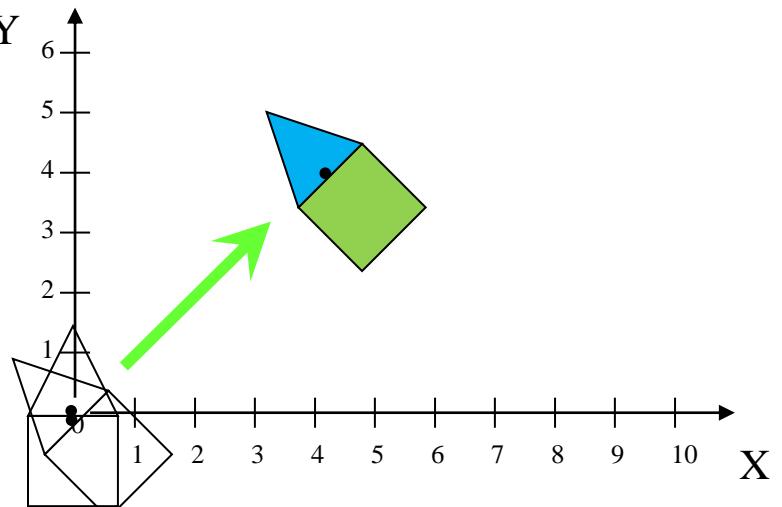
Translation → Rotation

Rotate by 45°

then

Translate by

$$x=6, y=0$$

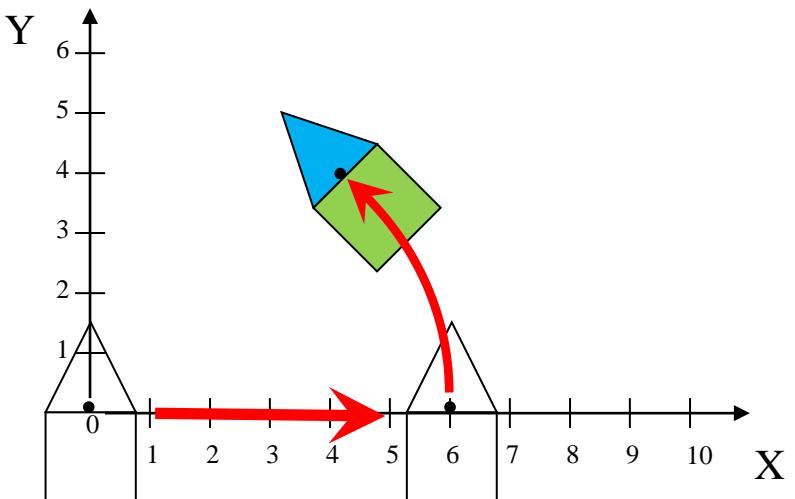


Rotation → Translation

COMPARE Methods 1 & 2

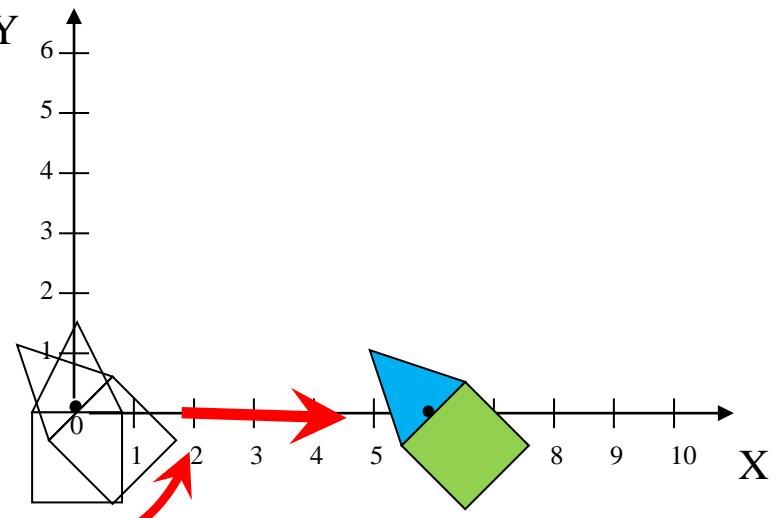
METHOD 1:

Translate() by
 $x=6, y=0$
then
Rotate() by 45°



Translation → Rotation

Rotate() by 45°
then
Translate()
by $x=6, y=0$

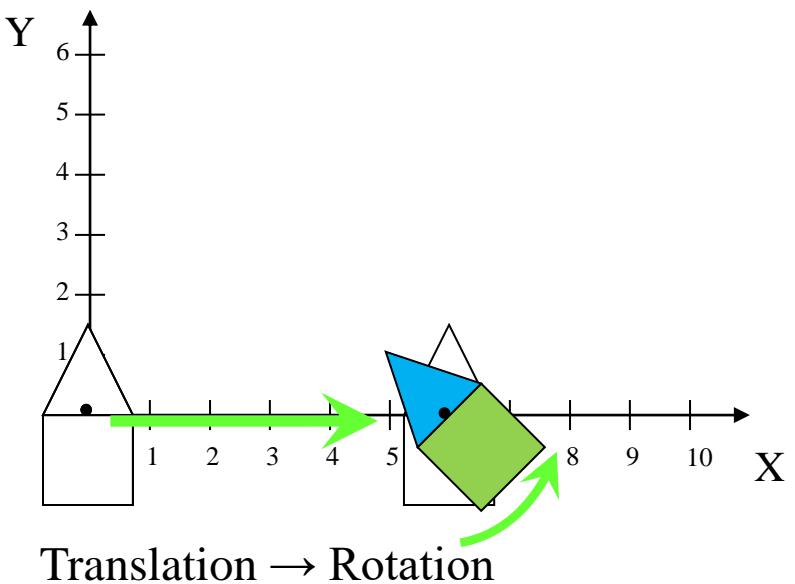


Rotation → Translation

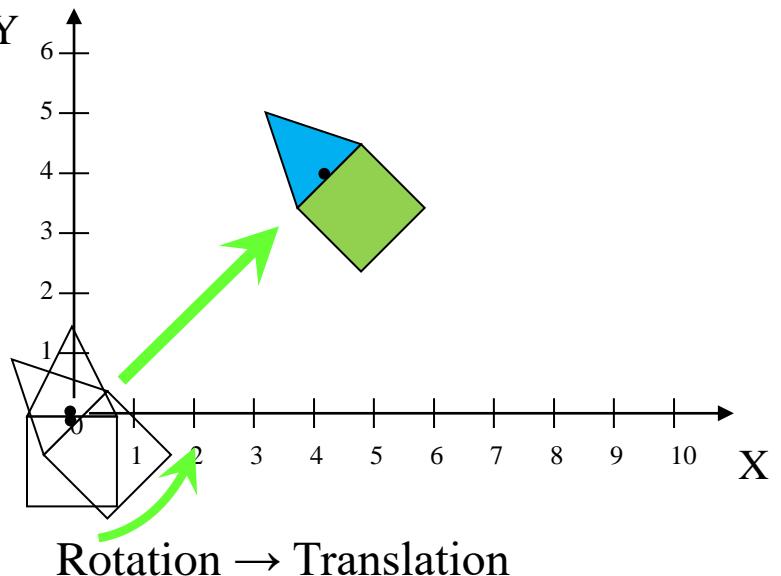
COMPARE Methods 1 & 2

/ METHOD 2:

Translate() by
 $x=6, y=0$
then
Rotate() by 45°



Rotate() by 45°
then
Translate()
by $x=6, y=0$



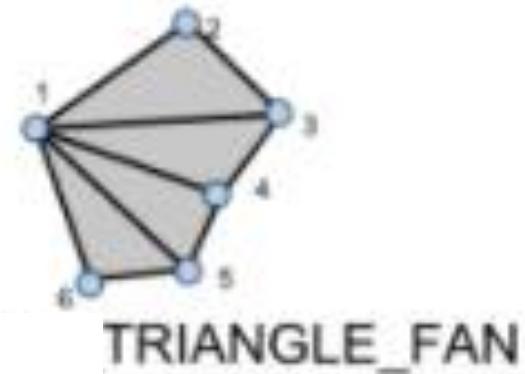
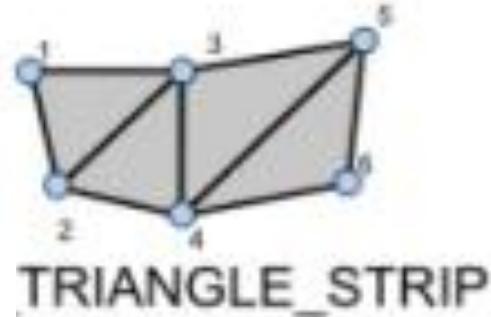
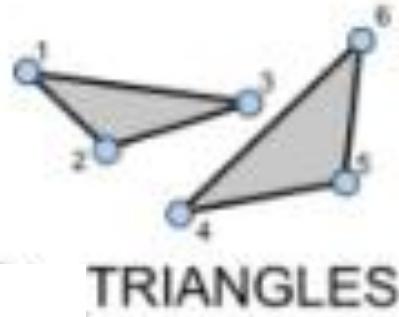
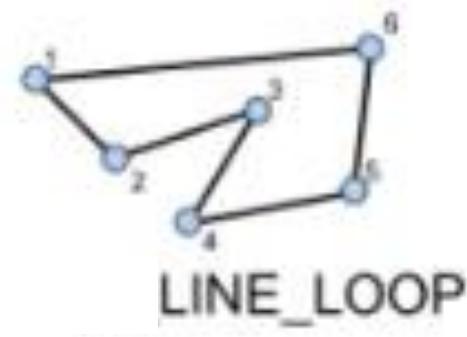
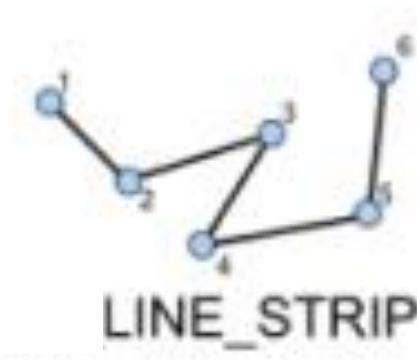
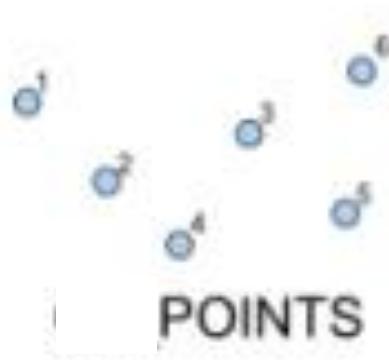
Organizing Your Vertex Arrays to Make Rigid 3D Parts

Jack Tumblin

Northwestern Univ COMP_SCI 351-1

Fall 2021

RECALL: WebGL Drawing Primitives



For `gl.drawArrays()` or `gl.drawElements()` calls:

e.g. `gl.drawArrays(gl.POINTS, vStart, vCount);`

Vertex ordering in the VBO?

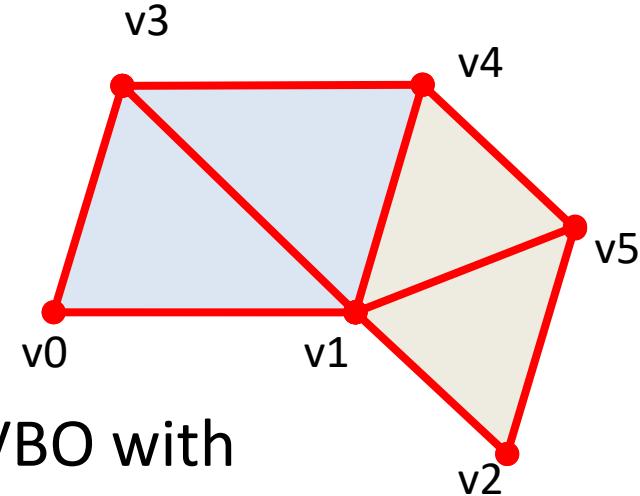
Try to draw *this* set of 4 triangles using *these* 6 vertices (v0...v5):

What does WebGL draw if we filled the VBO with

v0 , v1 , v2 , v3 , v4 , v5

and called:

- `gl.drawArrays(gl.POINTS, 0, 6)` ?
- `gl.drawArrays(gl.TRIANGLES, 0, 6)` ?
- `gl.drawArrays(gl.TRIANGLE_STRIP, 0, 6)` ?
- `gl.drawArrays(gl.TRIANGLE_FAN, 0, 6)` ?



Well? How should we draw the figure above?

And, is there a 'best' vertex ordering for triangle drawing?

Organize Vertices(1): By **Winding Order**

What's the 'correct' order for triangle vertices?

(v1, v2, v3) ? Or (v3, v2, v1)? Or (v1, v3, v2) or... (3 more)...

- **ALWAYS Right-handed Coord. Systems!**

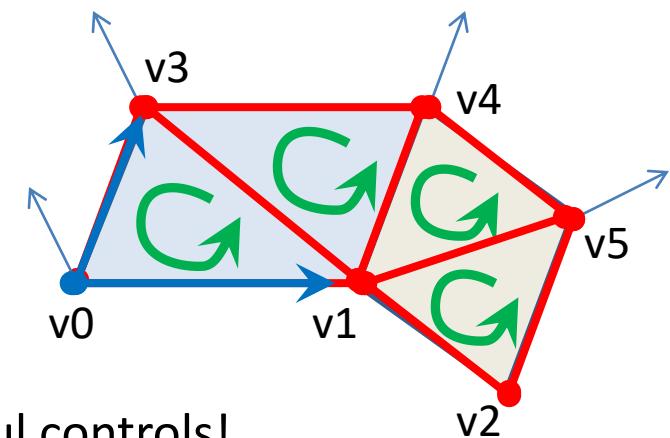
THUS → gl.TRIANGLES expects **CCW-ordered vertices:**

v0, v1, v3; v3, v1, v4; v4, v1, v5; v5, v1, v2; (for example)

- Why? triangle's 'front face' defined by CCW order
 - CCW-order 'curls fingers around' surface normal (right-hand rule)
 - Triangle's surface-normal calcs:
 $(v1-v0) \times (v3-v0)$
 - Enables Fast 'backface removal'
(CCW triangle on-screen? !SKIP IT!)

TRY IT: `gl.enable(gl.CULL_FACE);`

and look into '`gl.enable()`' for more helpful controls!



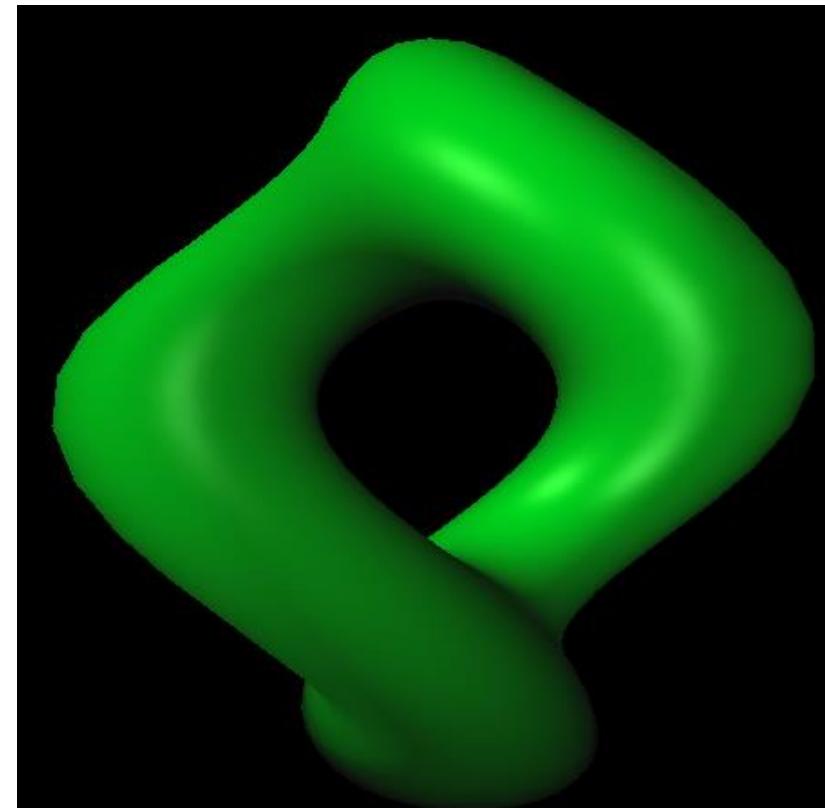
3D Vertex Ambiguity ?

A vertex is **ONE** labeled geometric location
with **ONE** set of attributes

- **ONE** point location,
- **ONE** color,
- **ONE** surface normal direction
- **ONE** texture address (per texture)
- ... etc...

- **GOOD!**

- describes a (smooth) surface at **ONE** point;
- WebGL/OpenGL can linearly interpolate between them. **...BUT...**



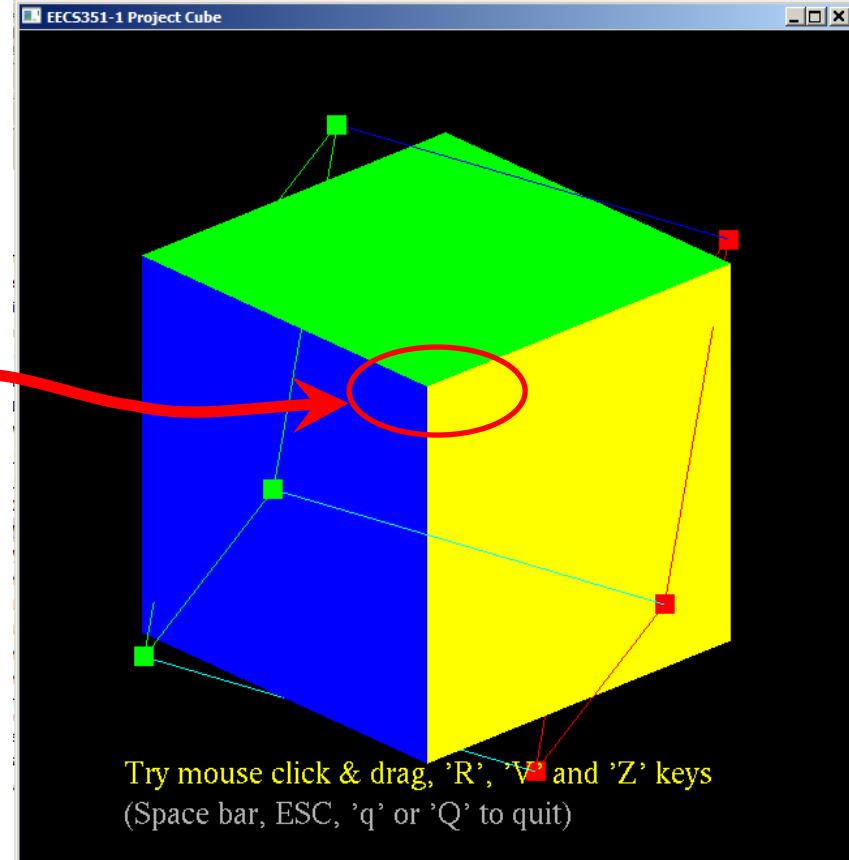
But How can Vertices make 'Corners' ?!?

?Non-Smooth Parts? Sharp Edges? Corners?!?

- Won't a **Corner** require *MULTIPLE* colors? Faces? →*MULTIPLE* attributes?!

- — **NO—**

- **Each face** gets its own *set of vertices*
 - Vertices are infinitesimally tiny.
 - Assembled faces put *MULTIPLE vertices* at each cube corner.
 - Rename any 'corners', that is, any multi-vertex locations,
as 'NODES'



Organize Vertices(2): by **Nodes**

-z face (bottom), CCW ordering:

$-1.0, -1.0, -1.0, \text{ // node 0}$

$-1.0, 1.0, -1.0, \text{ // node 1}$

$1.0, 1.0, -1.0, \text{ // node 2}$

$1.0, -1.0, -1.0, \text{ // node 3}$

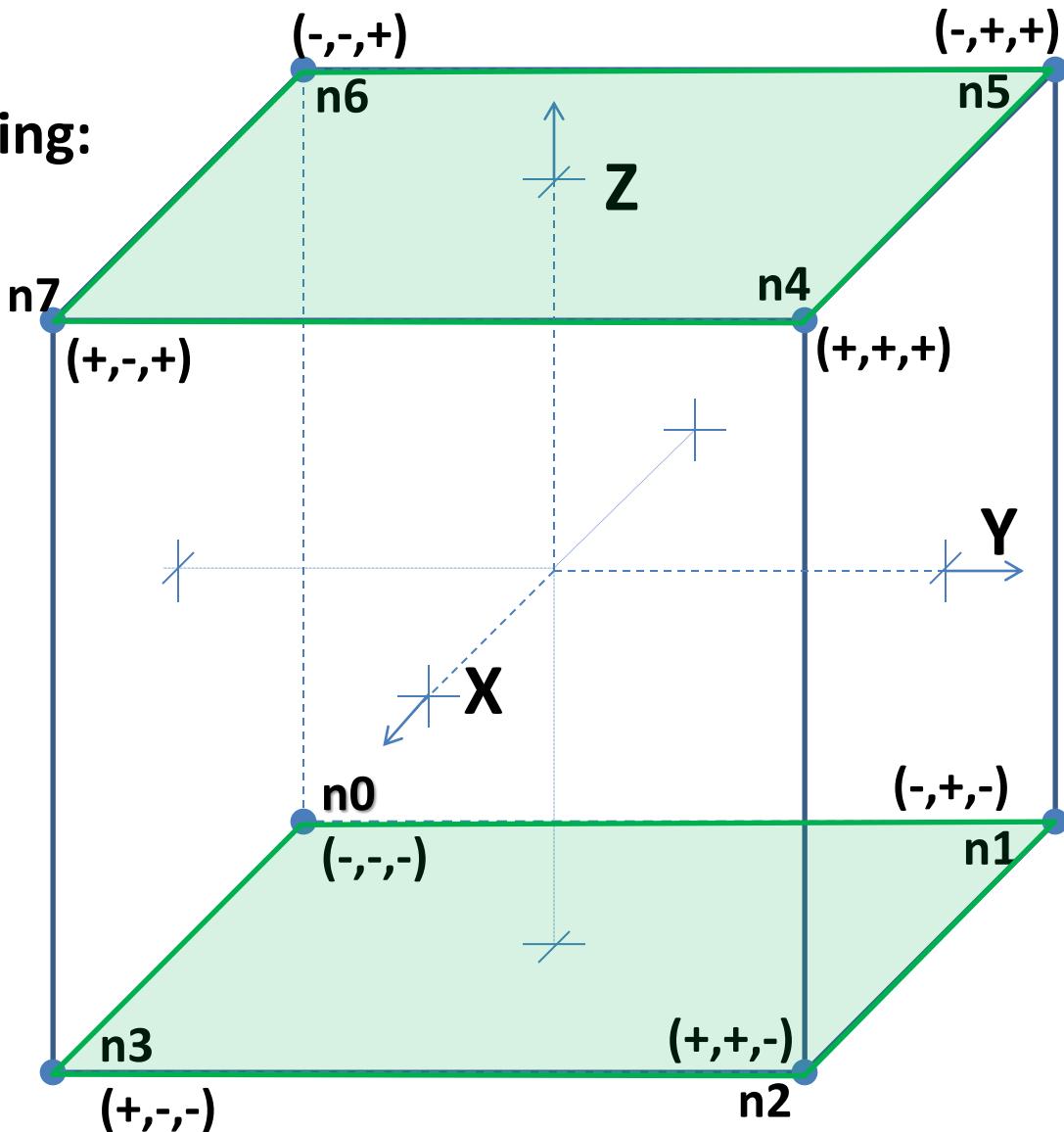
+z face (top), CCW ordering:

$1.0, 1.0, 1.0, \text{ // node 4}$

$-1.0, 1.0, 1.0, \text{ // node 5}$

$-1.0, -1.0, 1.0, \text{ // node 6}$

$1.0, -1.0, 1.0 \text{ // node 7}$



Organize Vertices(3) for `gl.TRIANGLES`

-z face (bottom), CCW ordering:

-1.0, -1.0, -1.0, // node 0

-1.0, 1.0, -1.0, // node 1

1.0, 1.0, -1.0, // node 2

1.0, -1.0, -1.0, // node 3

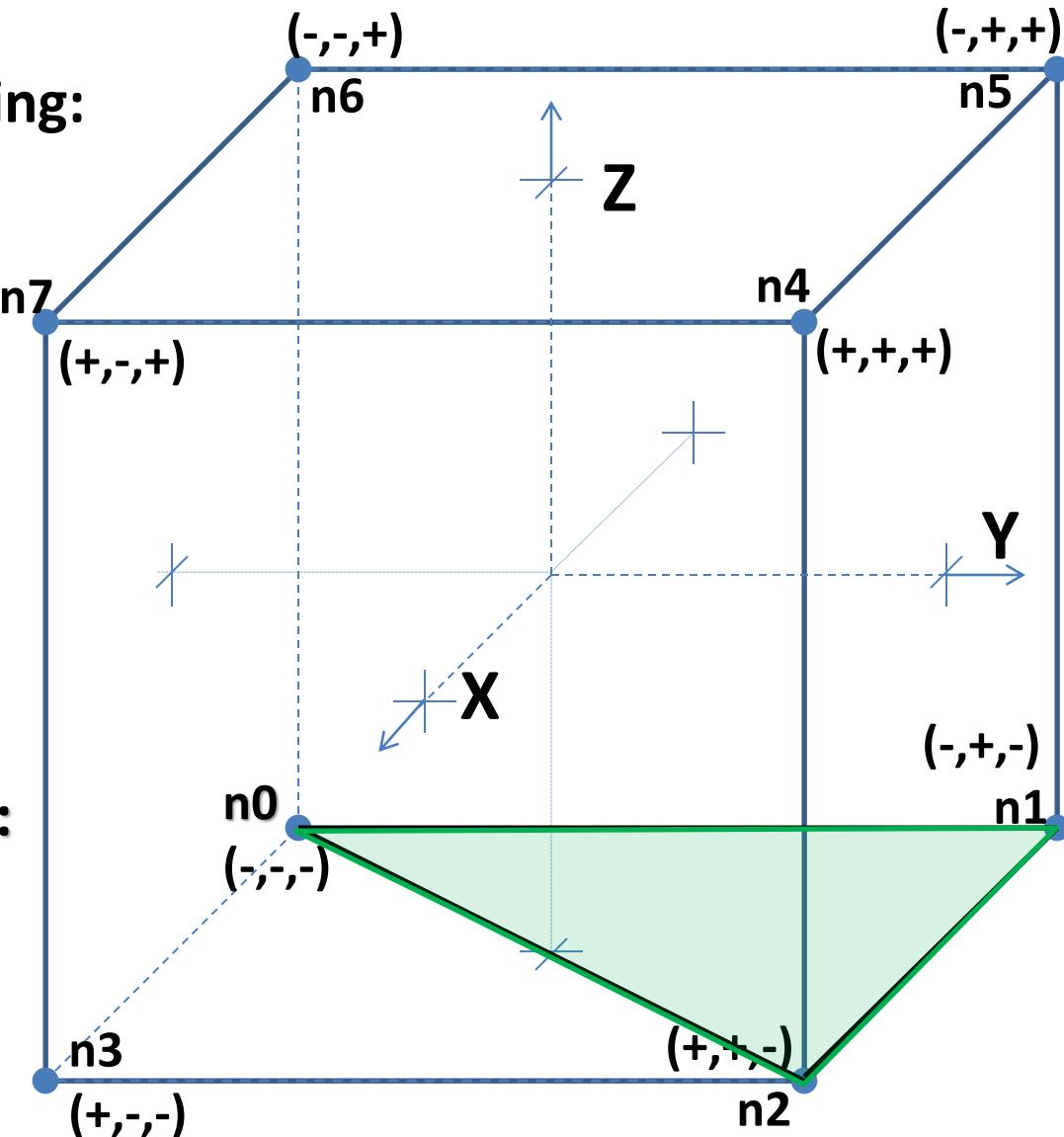
2 Triangles for bottom face?

Two possible diagonals:

this one (**ALL CCW triangles**):

n0,n1,n2; n2,n3,n0

or ? ... (other diagonal?)



Organize Vertices(4) for `gl.TRIANGLES`

-z face (bottom), CCW ordering:

-1.0, -1.0, -1.0, // node 0

-1.0, 1.0, -1.0, // node 1

1.0, 1.0, -1.0, // node 2

1.0, -1.0, -1.0, // node 3

2 Triangles for bottom face?

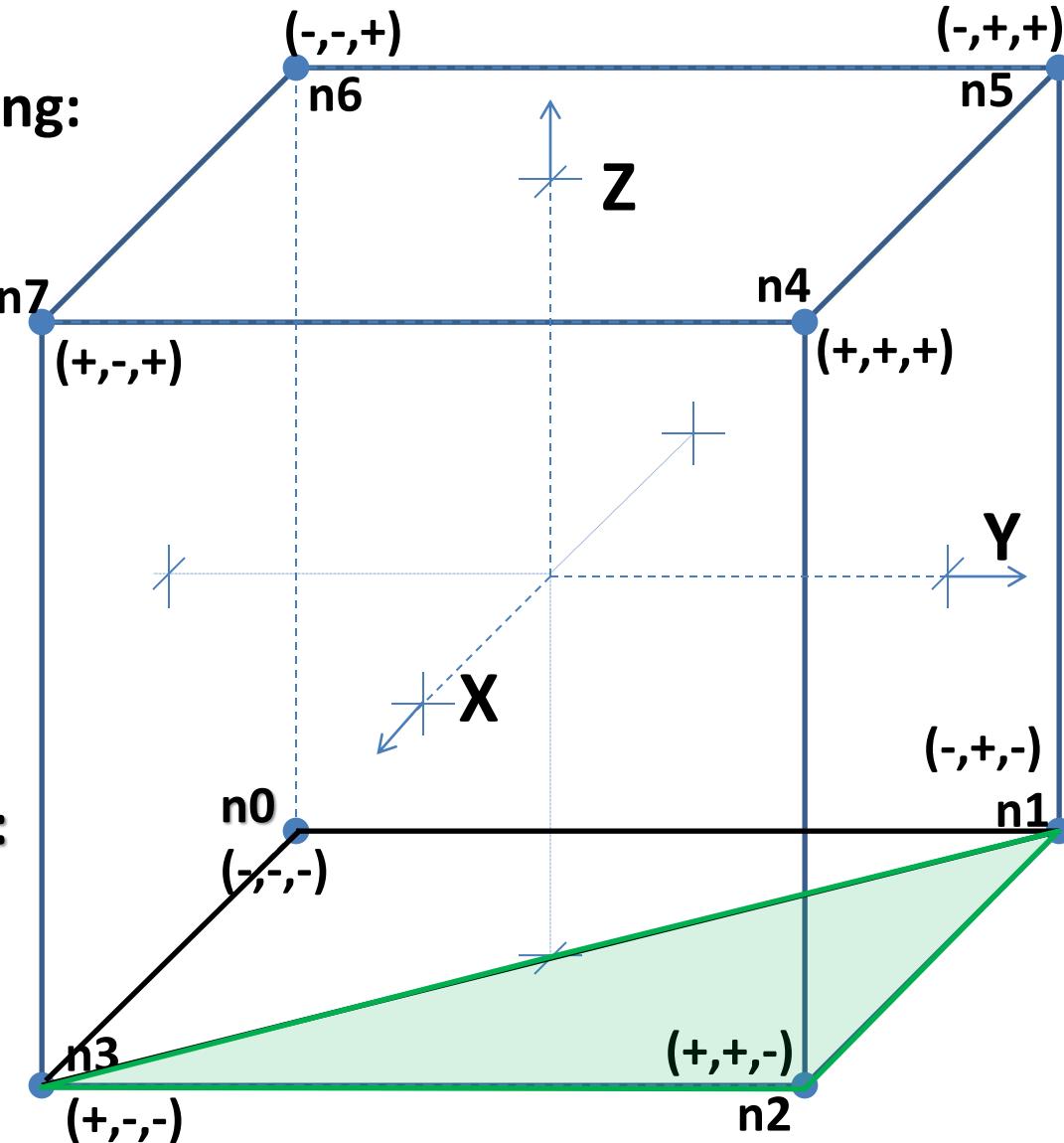
Yes. Two possible diagonals:

this one (**ALL CCW triangles**):

n0,n1,n2, n2,n3,n0

or this one: (other diagonal)

n1,n2,n3, n3,n0,n1.



Organize Vertices(5) for `gl.TRIANGLES`

with 8 SHARED vertices? (node # == vert#)

2,3,0, 0,1,2, // -z face

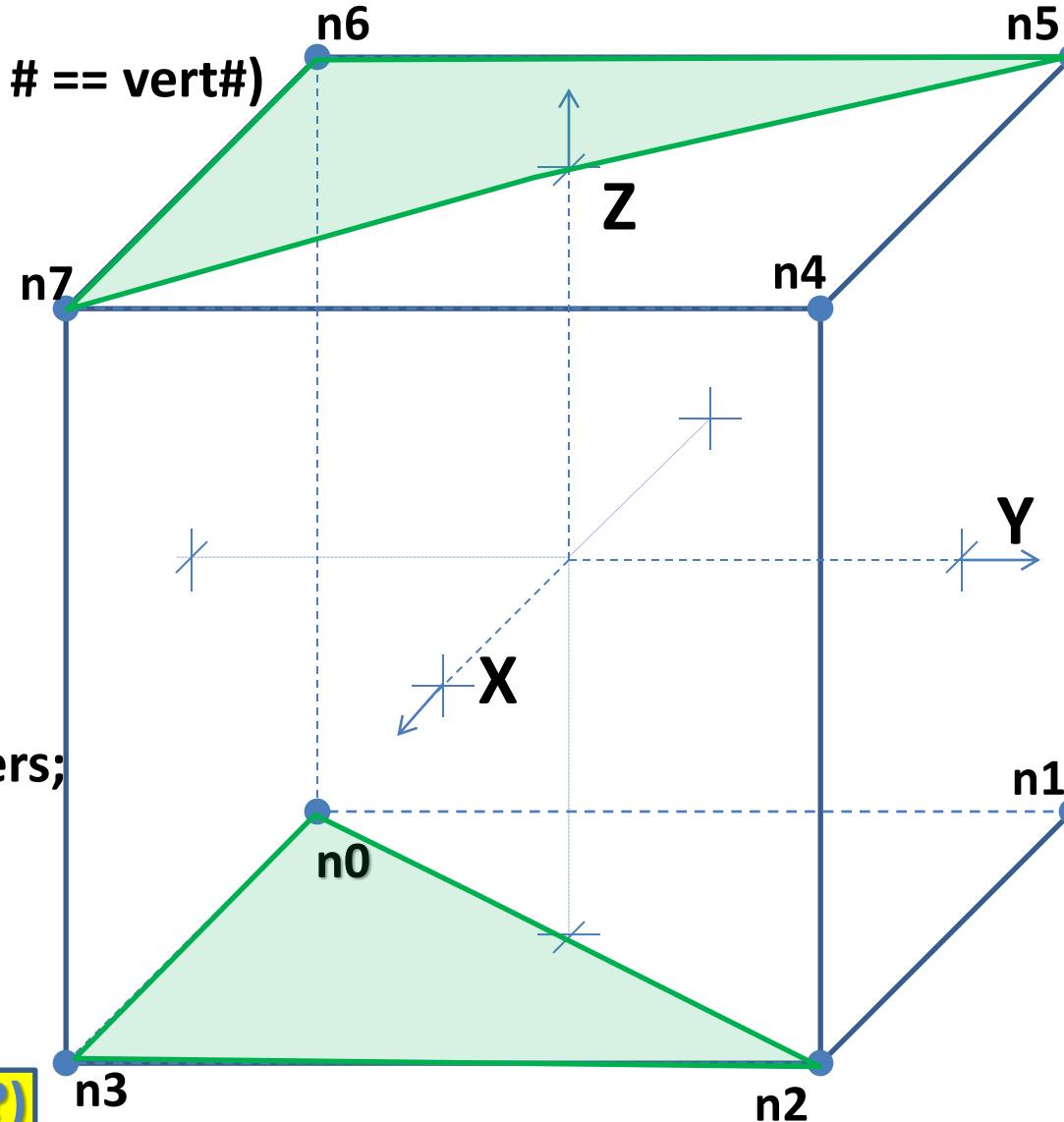
5,6,7, 7,4,5, // +z face

3,2,4, 4,7,3, // +x face

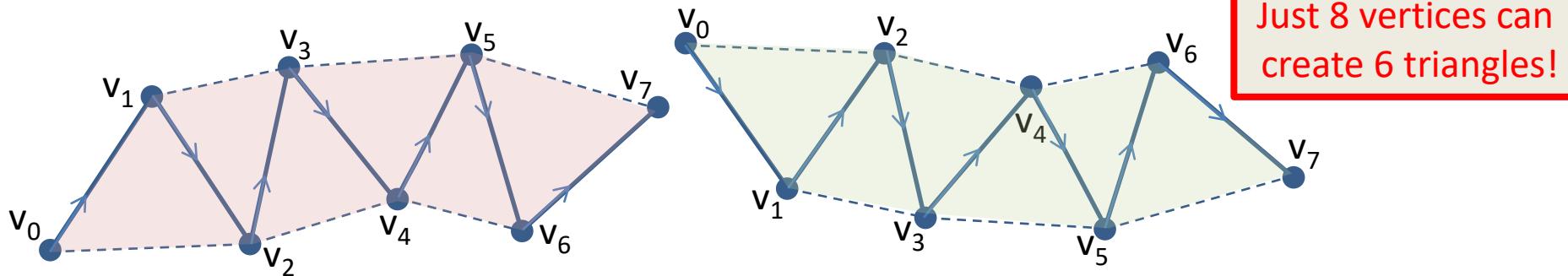
1,5,4, 4,2,1, // +y face

6,5,1, 1,0,6, // -x face

3,7,6, 6,0,3}; // -y face



Organize Vertices (1): `gl.TRIANGLE_STRIP`



IDEA: Build a ‘zig-zag’ vertex sequence
to define a flexible ‘strip’ surface made of triangles

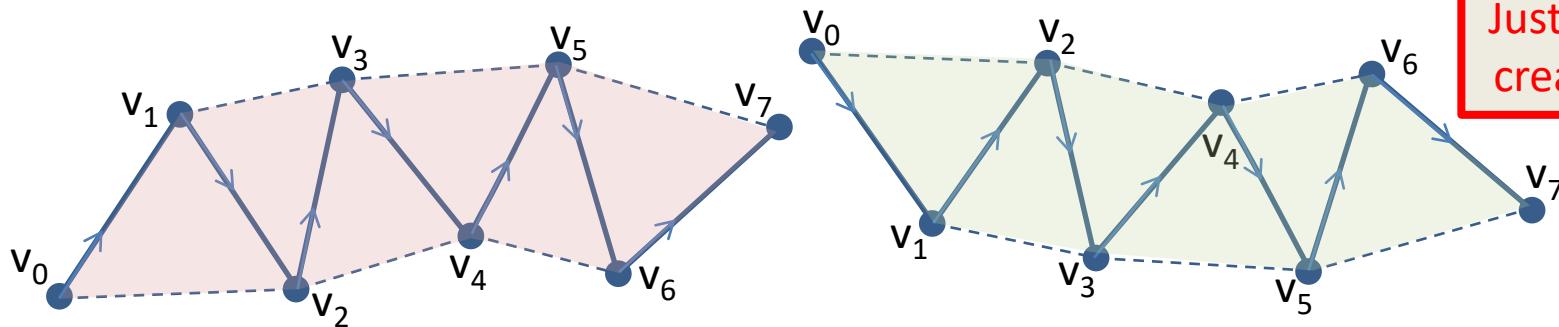
- **Efficient!** Sequence lists each triangle vertex *only once*
- **Efficient!** After 1st triangle: add **N** vertices → add **N** triangles!
- **Powerful!** Put **ANY** vertex **ANYWHERE** (!)

What 3D shapes can you build with JUST ONE triangle strip?

WARNING: most vertices SHARED by 3 adjacent triangles!

HELPFUL: strip can include zero-area ‘degenerate’ triangles...

Organize Vertices (1): `gl.TRIANGLE_STRIP`



Just 8 vertices can
create 6 triangles!

IDEA: Build a 'zig-zag' vertex sequence
to define a flexible 'strip' surface

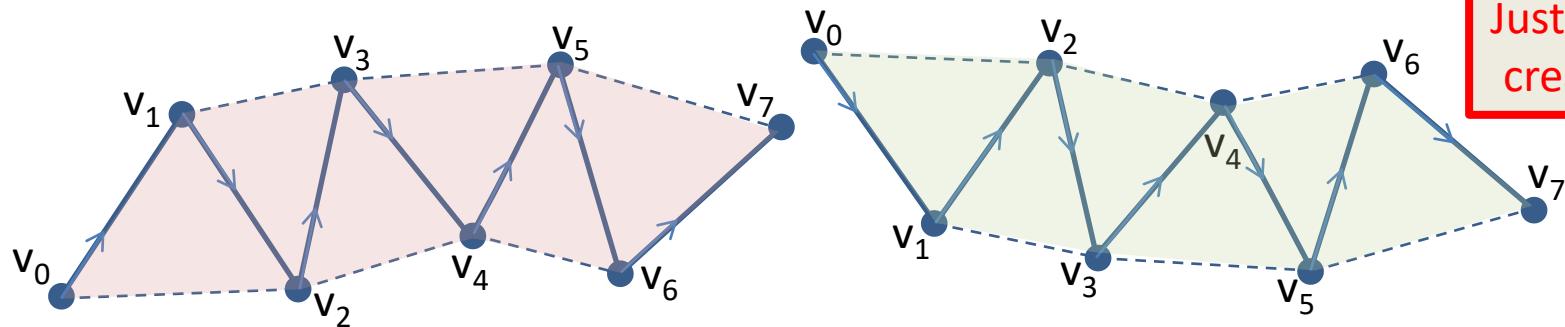
- **Efficient!** Sequence lists each triangle vertex once
- **Efficient!** After 1st triangle: add 1 vertex to start next triangle
- **Powerful!** Put ANY vertex in sequence

What 3D shapes can you make?
WARNING: m vertices → m+1 adjacent triangles!

HELPFUL: strip can have zero-area 'degenerate' triangles...

**SOME (crude) EXAMPLES:
see 'Basic Shapes'
starter code...**

Organize Vertices (2): **gl.TRIANGLE_STRIP**



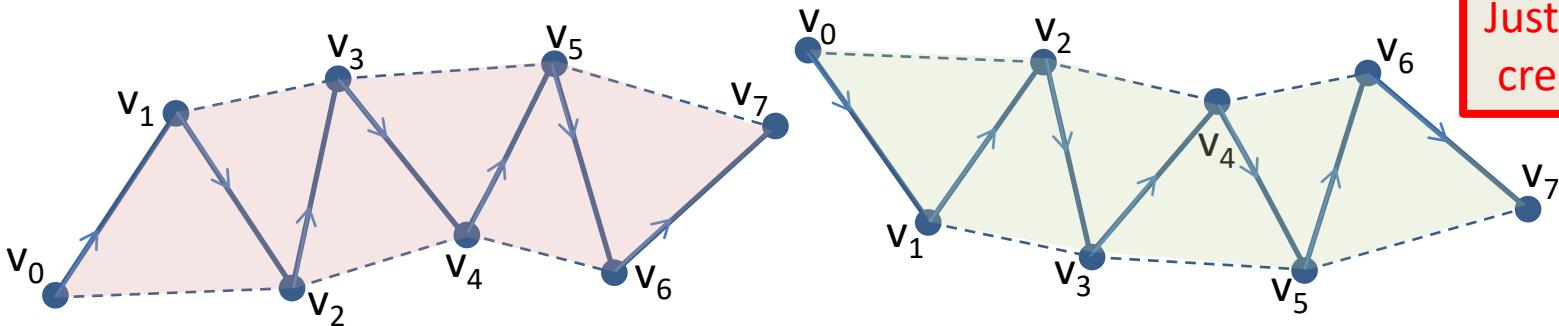
Just 8 vertices can
create 6 triangles

EASY PUZZLE: Create a tetrahedron from a single triangle strip.

HARDER PUZZLE: Create a cube

from a single triangle strip.

Organize Vertices (3): `gl.TRIANGLE_STRIP`

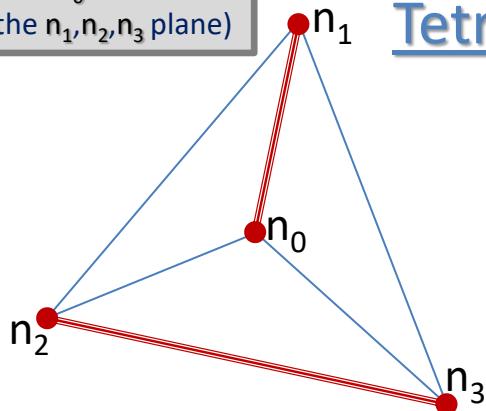


EASY PUZZLE: Create a tetrahedron from a single triangle strip.

HARDER PUZZLE: Create a cube

from a single triangle strip.

(NOTE: n_0 is ABOVE
the n_1, n_2, n_3 plane)



Tetrahedron (in this drawing: largest triangle is *behind* the others)

--Only **4** node locations & only **4** triangles

--How many tri-strip vertices will we need?

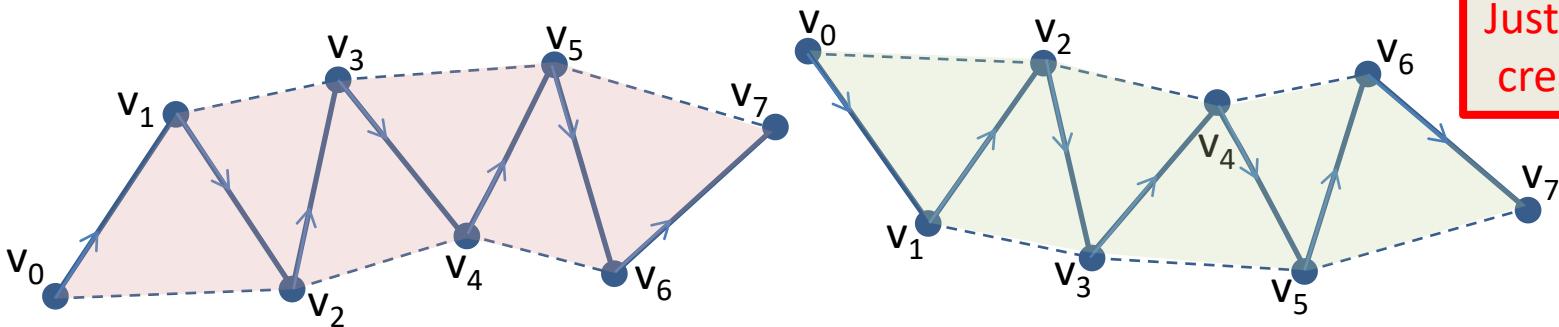
--What triangles will this tri-strip make?

n_0, n_1, n_2, n_3

--What sequence completes the tetrahedron?

Favorite 3D visualization: pair of
separated \perp lines ($n0, n1$) ($n2, n3$)

Organize Vertices (4): `gl.TRIANGLE_STRIP`

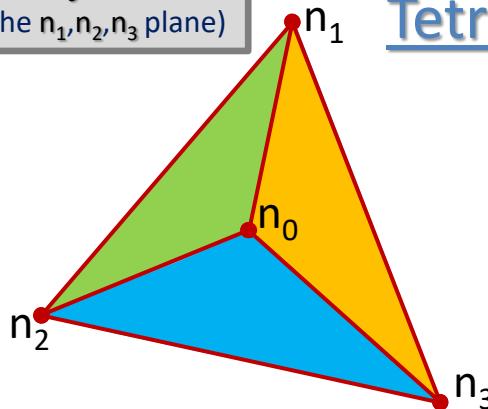


EASY PUZZLE: Create a tetrahedron from a single triangle strip.

HARDER PUZZLE: Create a cube

from a single triangle strip.

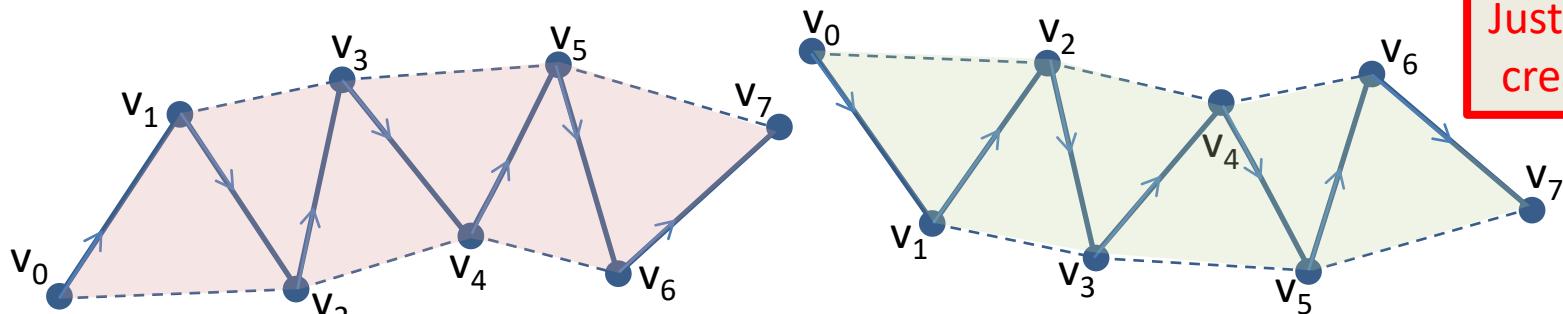
(NOTE: n_0 is ABOVE the n_1, n_2, n_3 plane)



Tetrahedron (in this drawing, largest triangle is *behind* the others)

- Only **4** node locations & only **4** triangles
- How many tri-strip vertices will we need? **6**
- What triangles will **this** tri-strip make?
 n_0, n_1, n_2, n_3 (ANSWER: $(n_0, n_1, n_2), (n_2, n_1, n_3)$)
- What sequence completes the tetrahedron?

Organize Vertices (5): `gl.TRIANGLE_STRIP`

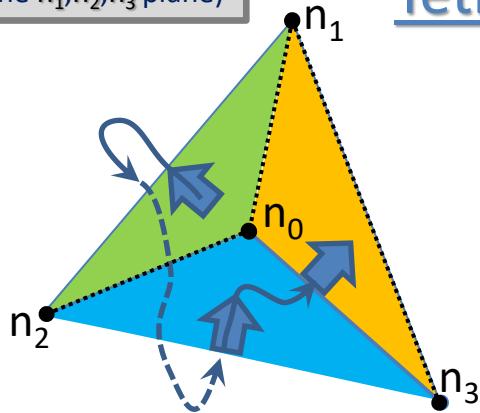


EASY PUZZLE: Create a tetrahedron from a single triangle strip.

HARDER PUZZLE: Create a cube

from a single triangle strip

(NOTE: n_0 is ABOVE
the n_1, n_2, n_3 plane)



Tetrahedron (in drawing, largest triangle is *behind* the others)

--Only 4 node locations & only 4 triangles

--How many tri-strip vertices will we need? 6

--What triangles will this tri-strip make?

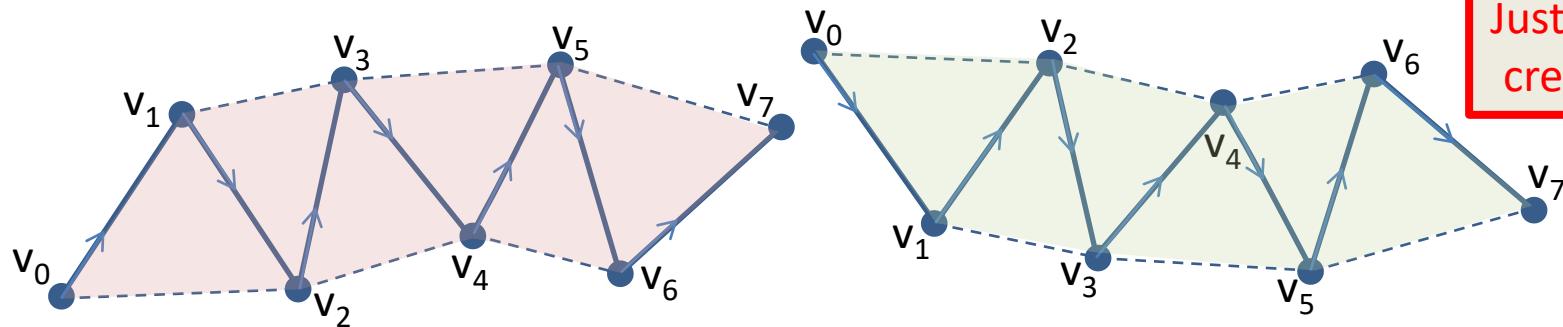
n_0, n_1, n_2, n_3 (ans: $(n_0, n_1, n_2), (n_2, n_1, n_3)$)

--What sequence completes the tetrahedron?

ANS: $(n_0, n_1, n_2, n_3, n_0, n_1)$

creates these triangles $(n_0, n_1, n_2), (n_2, n_1, n_3), (n_2, n_3, n_0), (n_0, n_3, n_1)$)

Organize Vertices (6): `gl.TRIANGLE_STRIP`



Just 8 vertices can
create 6 triangles

EASY PUZZLE: Create a tetrahedron from a single triangle strip.

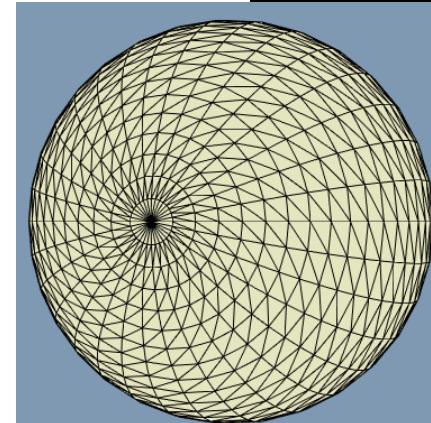
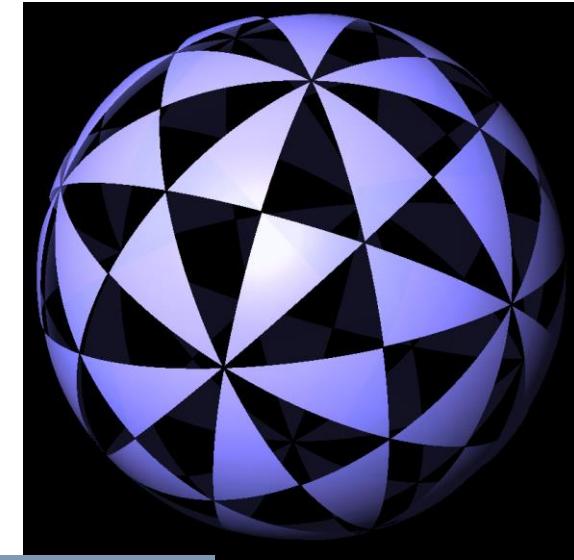
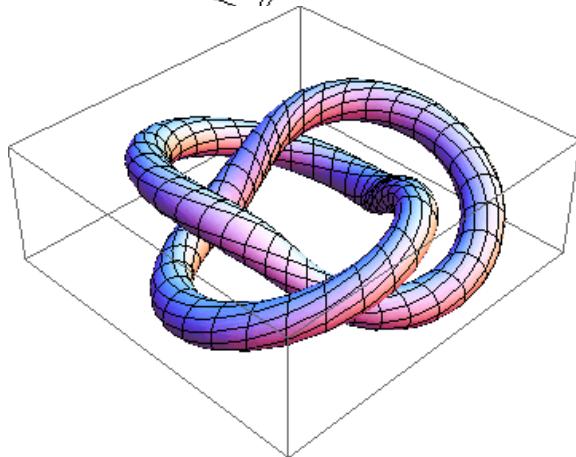
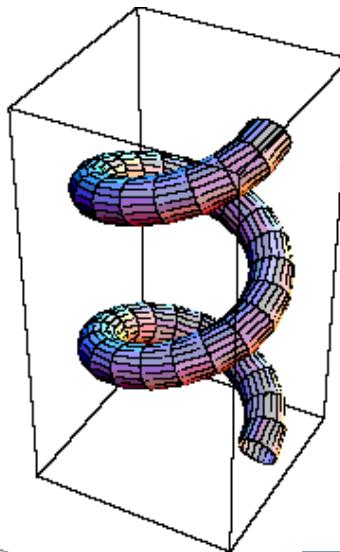
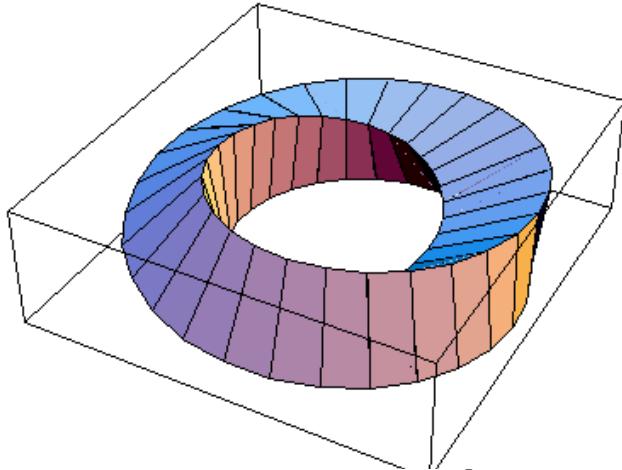
HARDER PUZZLE: Create a cube from a single triangle strip.

YOU TRY IT!

- How many triangles?
- How many vertices in the tri-strip?
- (Multiple solutions exist!)
- POST your solutions on CANVAS
- HINT: first create a 4-sided rectangular ring, then add triangles at both ends of strip to make the cube's top and bottom faces

Shape Tesselation (1): **gl.TRIANGLE_STRIP**

MANY ways to tessellate shapes into triangle strips! (google it)



Could you automate this video's method:

https://youtu.be/ZF4-6_pzJA8 in WebGL? In 3D?

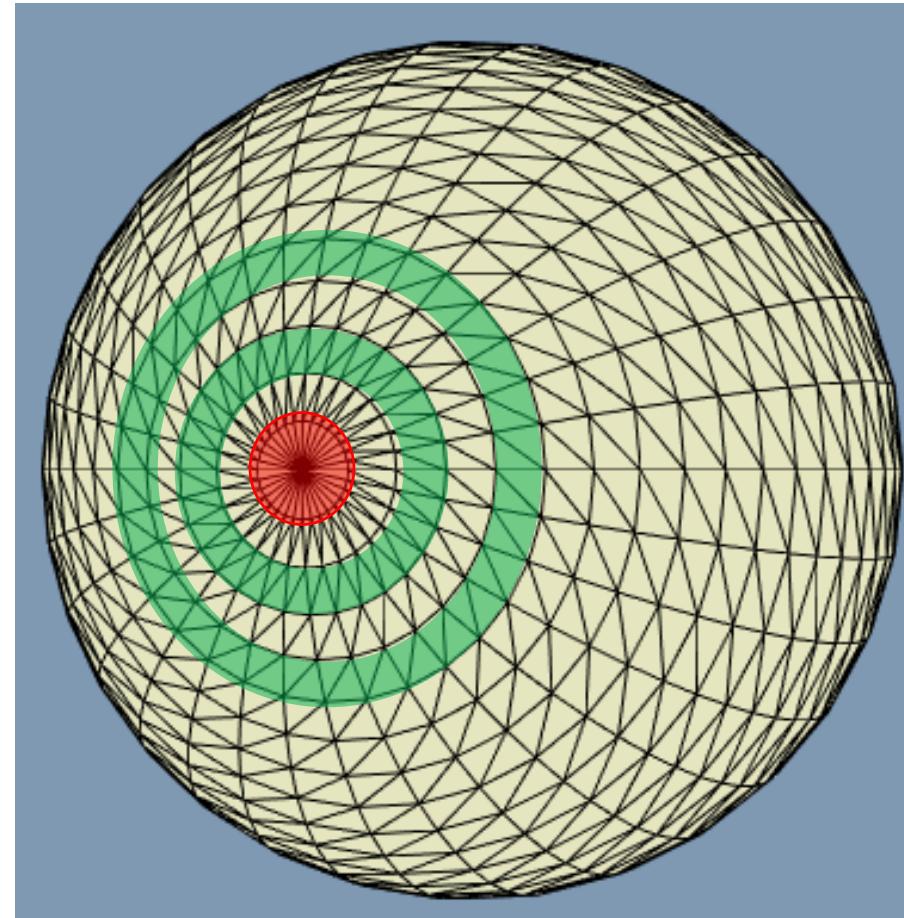
Courtesy Mathematica Journal

Shape Tesselation(2) : **gl.TRIANGLE_STRIP**

MANY ways to tessellate shapes
into triangle strips! (google it)

SIMPLE IDEA:

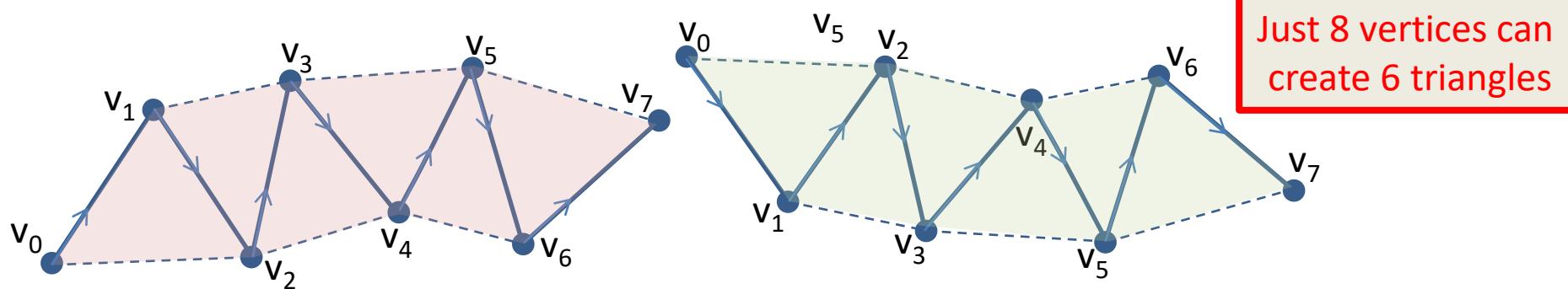
- Decompose 3D surface shape
into **ring-like slices** & **end-caps**
(e.g. latitude on a sphere)
- Connect rings into a
'stepped spiral'
made from just one tri-strip



Could you automate this video's method:
https://youtu.be/ZF4-6_pzJA8 in WebGL? In 3D?

Courtesy Mathematica Journal

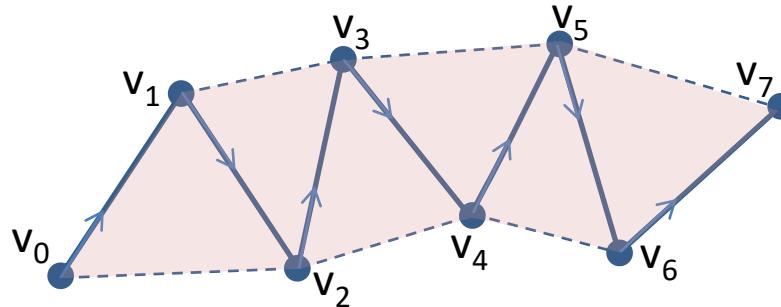
Shape Tesselation(3) : : **gl.TRIANGLE_STRIP**



IDEA: Build a ‘zig-zag’ vertex sequence
to define a flexible ‘strip’ surface made of triangles

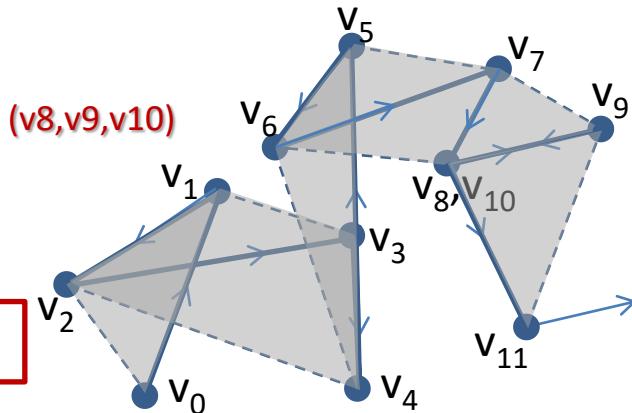
- **Efficient!** Sequence lists each triangle vertex *only once*
- **Efficient!** After 1st triangle: add **N** vertices → add **N** triangles!
- **Powerful!** Put **ANY** vertex **ANYWHERE** (!)
 - But ‘anywhere’ isn’t always good – it can get complicated...

Shape Tesselation(4) : : **gl.TRIANGLE_STRIP**

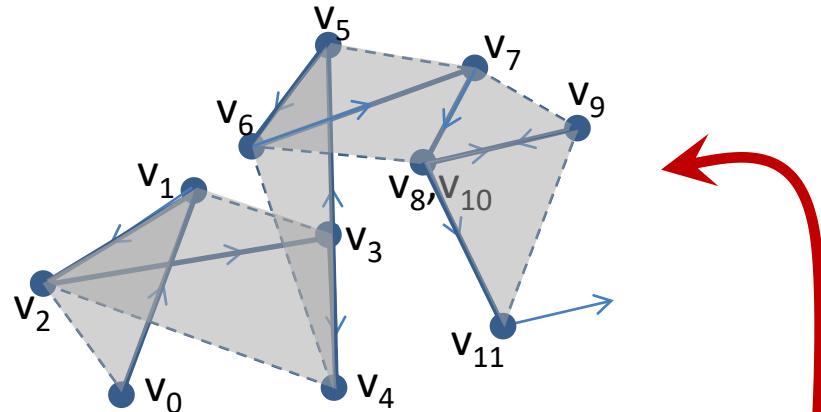
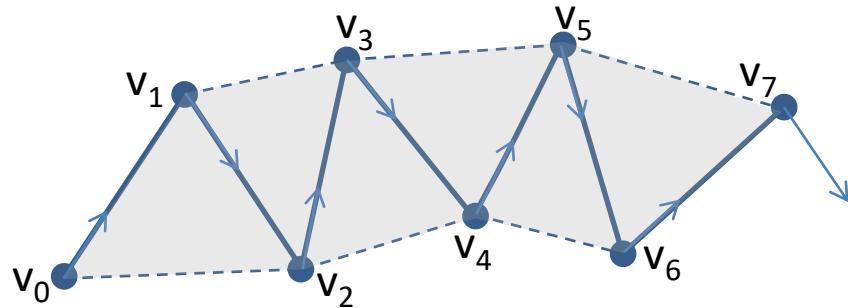


IDEA: Build a ‘zig-zag’ vertex sequence
to define a flexible ‘strip’ surface made of triangles

- **Efficient!** Sequence lists each triangle vertex *only once*
- **Efficient!** After 1st triangle: add **N** vertices → add **N** triangles!
- **Powerful!** Put ANY vertex ANYWHERE (!)
 - Degenerate (zero-area) triangles are OK (v_3, v_4, v_5 ; v_8, v_9, v_{10})
 - ‘fold-over’ triangles are OK ($v_0, v_1, v_2 \rightarrow v_1, v_2, v_3$);
 - redundant vertices & triangles are OK (v_8, v_{10})
 - New vertex v_N makes new triangle (v_N, v_{N-1}, v_{N-2})



Shape Tesselation(5): `gl.TRIANGLE_STRIP`



Wait, Wait, Wait —

?? What defines the 'front' surface of a `TRIANGLE_STRIP`??

ANSWER:

entire strip follows the 'Winding order'

of the strip's first triangle.

But this simple rule

can be tricky for strips that 'fold over' like this:

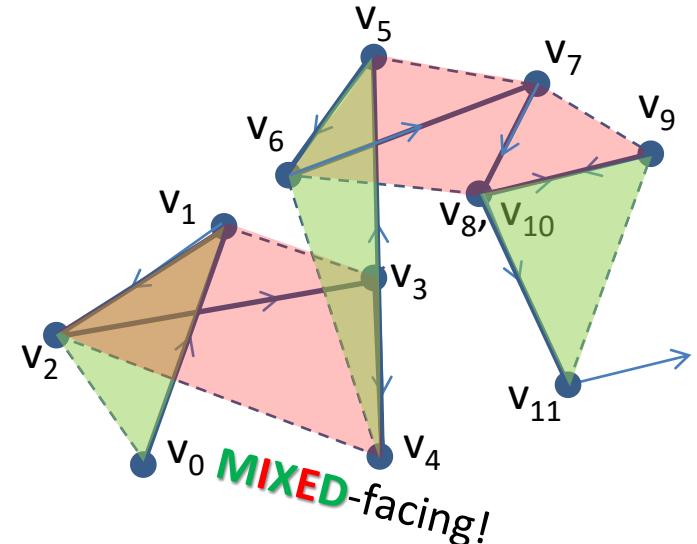
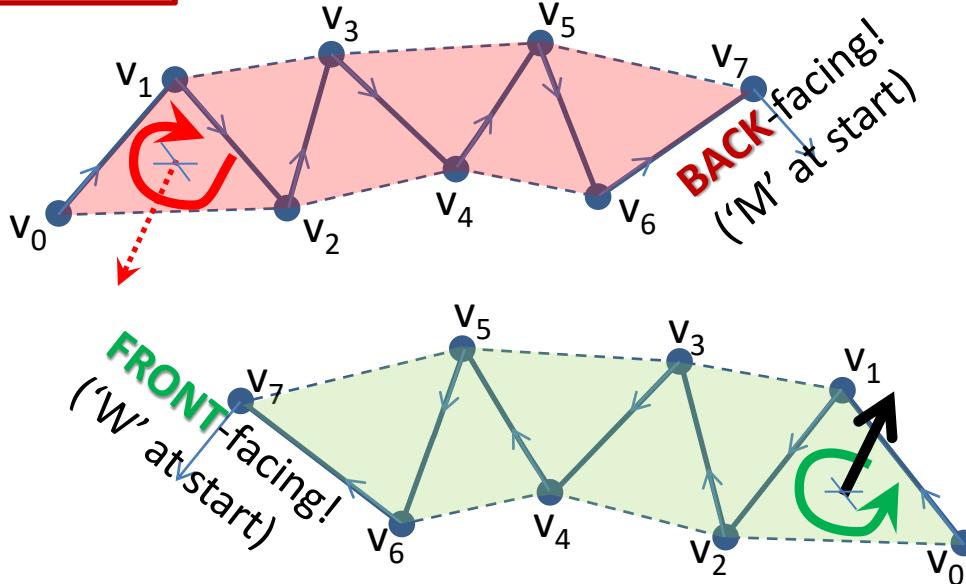
Shape Tesselation(6) : `gl.TRIANGLE_STRIP`

- **Always** count vertices from zero: v_0, v_1, v_2, \dots
- **Always** count triangles from zero: $T_0 = (v_0, v_1, v_2); T_1 = (); T_2 = () \dots$
- From N_{\max} vertices, creates $(N_{\max} - 2)$ **right-handed** triangles
Triangle $T_k == [v_k, v_{(k+1)}, v_{(k+2)}]$ for all even k (when $k \% 2 == 0$)
 $= [v_k, v_{(k+2)}, v_{(k+1)}]$ for all odd k (when $k \% 2 == 1$)



odd-numbered triangles swap their last 2 vertices!

TRY IT: Which triangles are **front-facing**? which are **back-facing**?



Shape Tesselation(7) : **gl.TRIANGLE_STRIP**

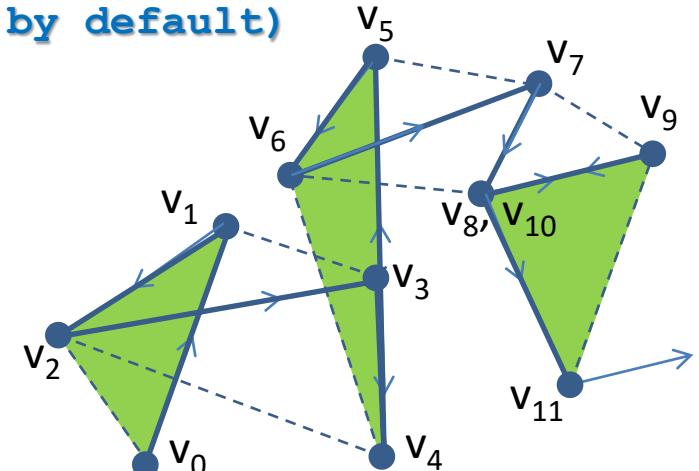
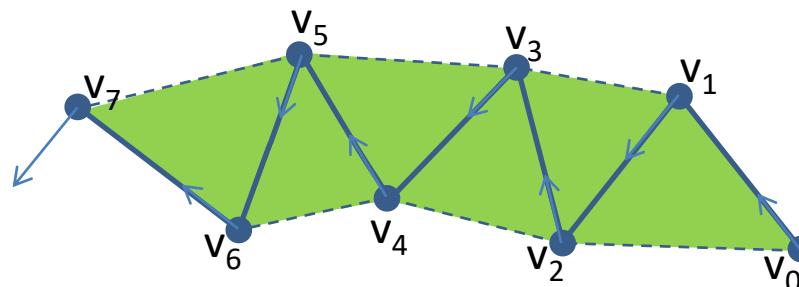
Always count vertices from zero: v_0, v_1, v_2, \dots

- Always count triangles from zero: $T_0 = (v_0, v_1, v_2); T_1 = (); T_2 = (); \dots$
 - From N_{\max} vertices, create $(N_{\max} - 2)$ right-handed triangles
- Triangle $T_k == [v_k, v_{(k+1)}, v_{(k+2)}]$ for all even k (when $k \% 2 == 0$)
 $= [v_k, v_{(k+2)}, v_{(k+1)}]$ for all odd k (when $k \% 2 == 1$)

odd-numbered triangles swap their last 2 vertices

TRY IT: Enable WebGL's Back-face Removal? You'll see only this:

```
gl.enable(gl.CULL_FACE); // (disabled by default)
```

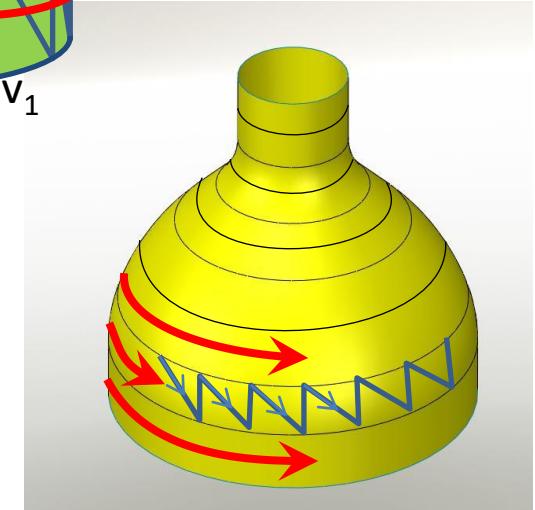
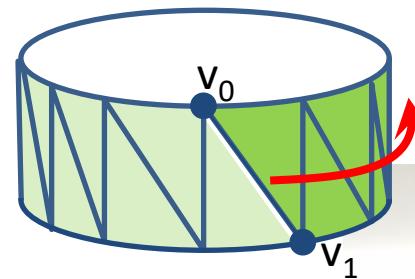
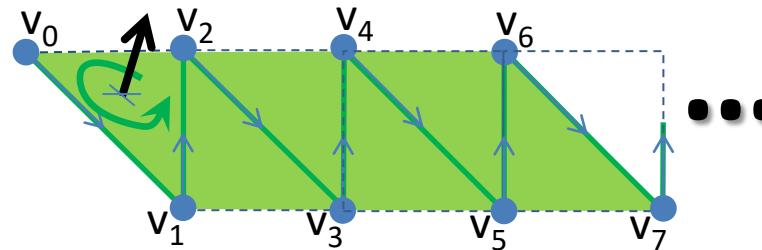


Shape Tesselation(8) : : **gl.TRIANGLE_STRIP**

What's a safe, sensible way to make 3D shapes from tri-strips?

Method A: (Naïve)

- Construct a uniform, right-handed strip ...
make it just as long as needed
- Bend the strip into a closed 'ring':
last two verts == first two verts
- Stack the rings to form 3D shapes :
Ring Top vertices (**even #'d** verts) and
Ring Base vertices (**odd #'d** verts)
(top & bottom can have a different radius...)

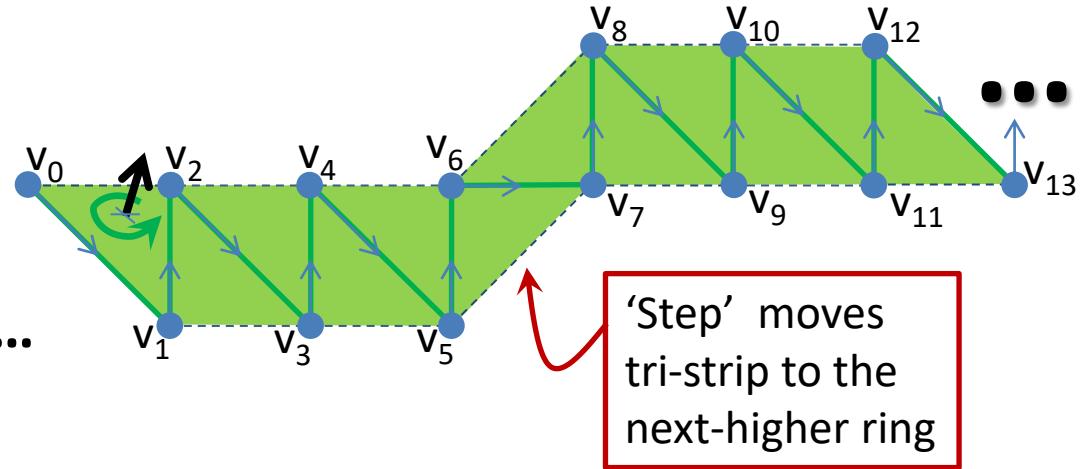


Shape Tesselation(9) : : **gl.TRIANGLE_STRIP**

What's a safe, sensible way to make 3D shapes from tri-strips?

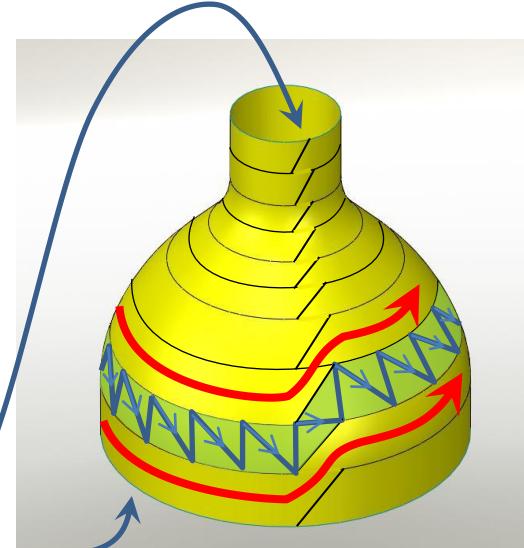
Method B: (Better)

- Design a right-handed 'step-offset' tri-strip made as long as needed...



- Bend suitable-length strips into '**stepped spiral**' to form desired tube-like 3D shapes
FROM JUST ONE TRIANGLE STRIP!

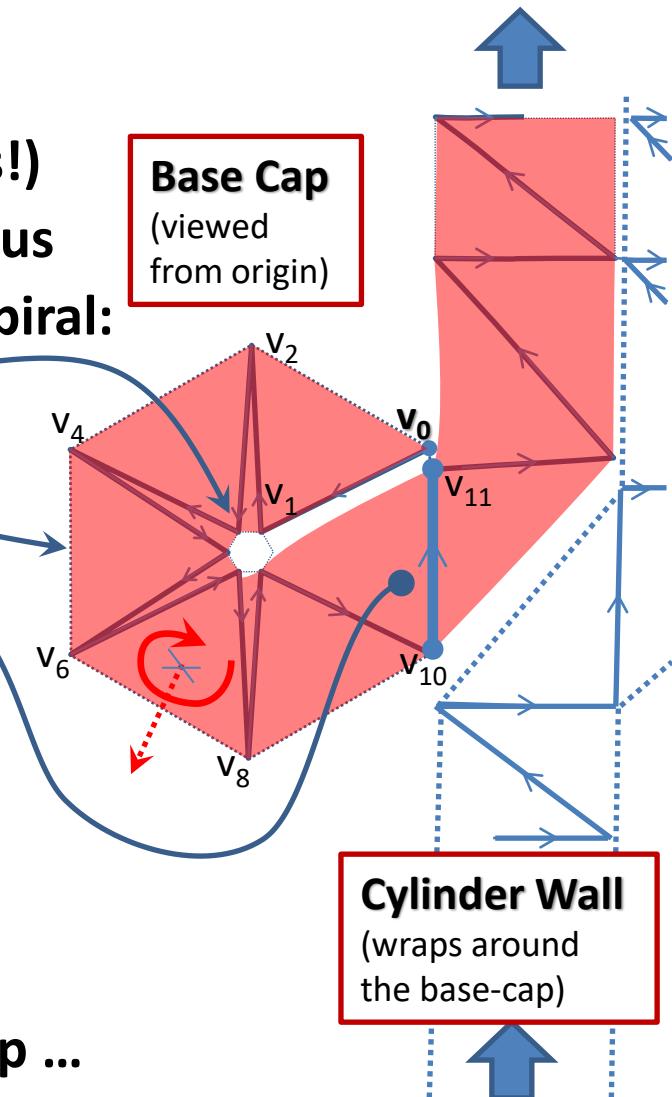
- Cool! But how do we close the ends?
Can we make tri-strip '**end caps**'?
(Another way: remember **TRIANGLE_FAN** drawing primitive?)



Shape Tesselation(10): `gl.TRIANGLE_STRIP`

Method B: 'end caps'? easy!

- Start at shape's base (typically at $z = -1$), and view it from origin (caution! → back-facing triangles!)
- Base-Cap surface normals point *down*: $z < 0$; thus
- Form CCW Base-Cap from 1st ring of stepped spiral:
 - Put all odd-#d verts at Base-Cap center-point (for clarity, shown here with vertices separated)
 - Even #d verts form Base-Cap outer edge
 - Last Base-Cap triangle holds the 'step' up to next ring in the shape (e.g. v10, v11 blue edge)
- Continue CCW stepped-spiral to complete all rings. Spiral upwards from base to top.



EXAMPLE:

Let's build a 6-sided cylinder from just one tri-strip ...

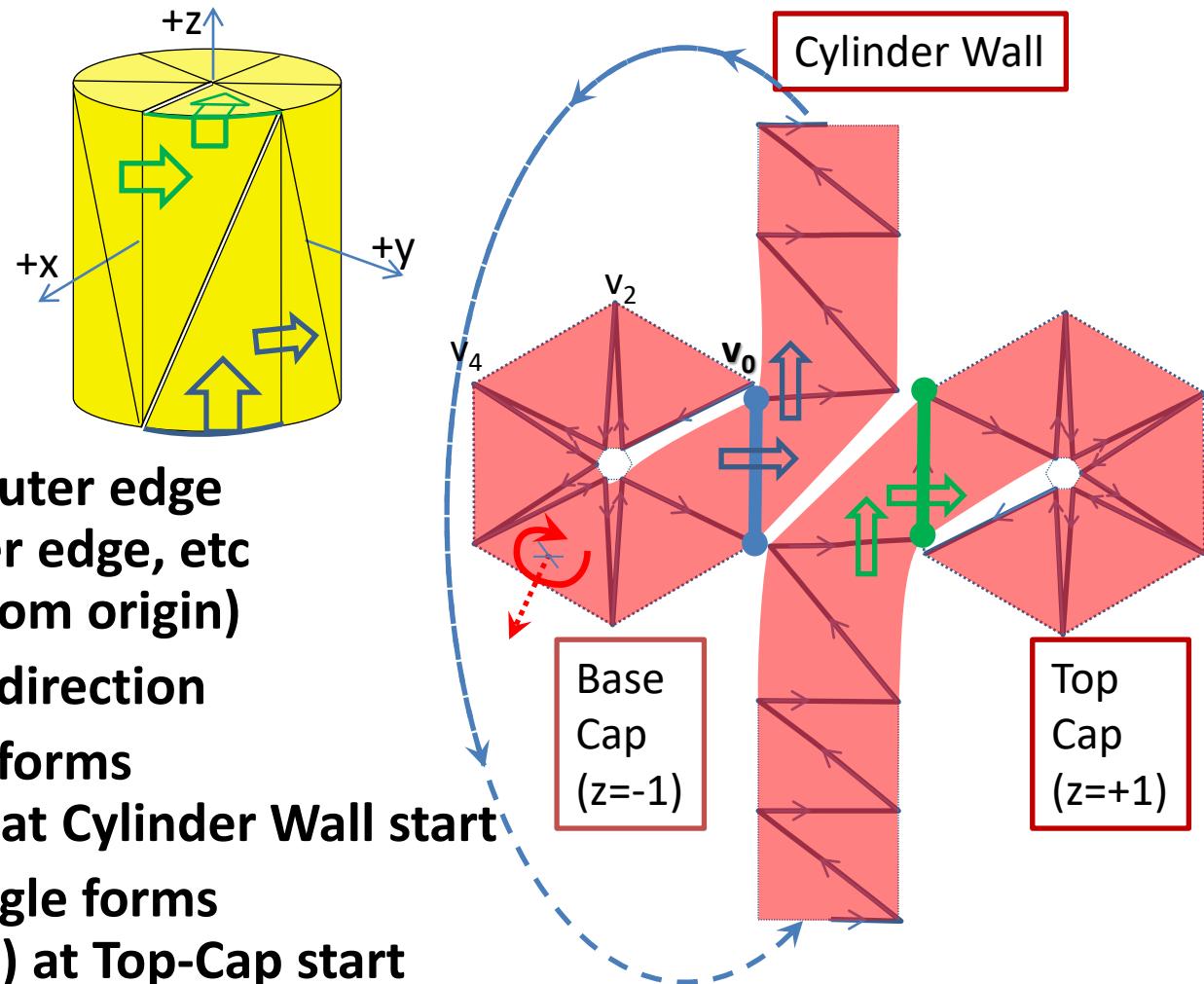
Shape Tesselation(11): `gl.TRIANGLE_STRIP`

Method B:

Example

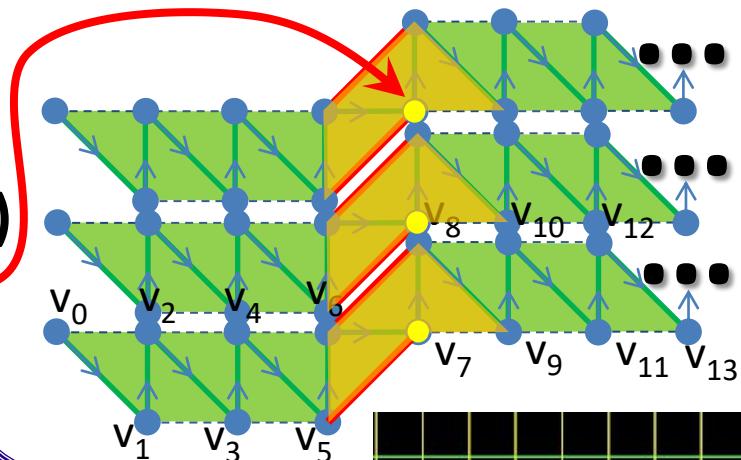
6-sided cylinder:

- Start at base-cap: $z=-1$
- Place v_0 on base-cap outer edge
 v_1 at center, v_2 at outer edge, etc
(tri-strip faces AWAY from origin)
- Make CCW spiral in $+z$ direction
- Last Base-Cap triangle forms
spiral-step edge (blue) at Cylinder Wall start
- Last cylinder-wall triangle forms
spiral-step edge (green) at Top-Cap start
- (sigh) Note Cylinder Wall's dissimilar diagonal
on its 1st and last triangle (with blue, green edges)



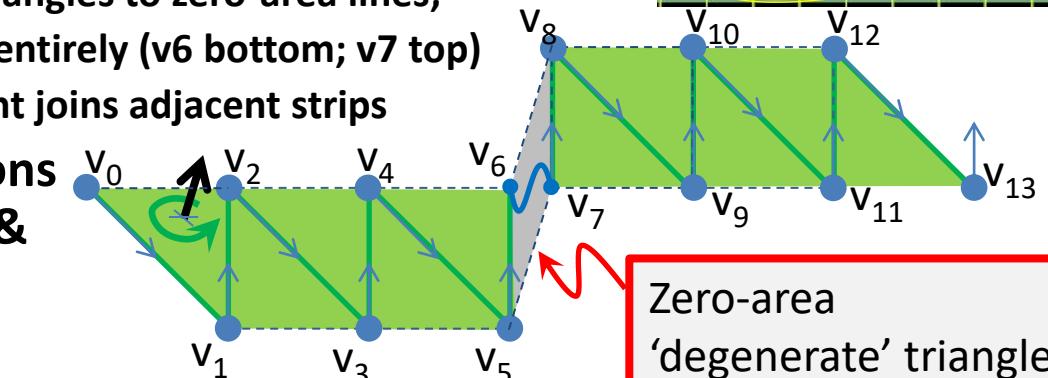
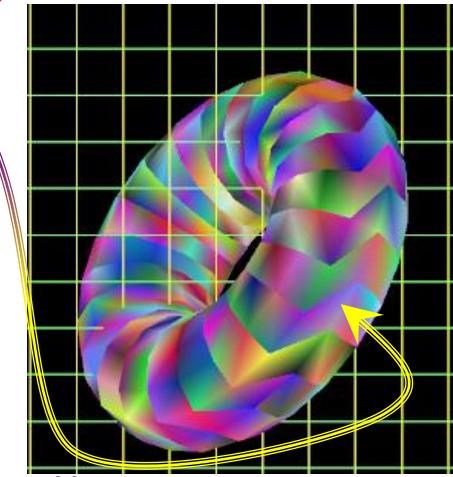
Shape Tesselation(12): `gl.TRIANGLE_STRIP`

- Method B makes an ugly ‘seam’ from **anti-diagonals** on each of two ‘step’ triangles ($v5, v6, v7$) ($v6, v7, v8$)
- Shares one **ugly vertex** with TWO strips & that odd-looking step



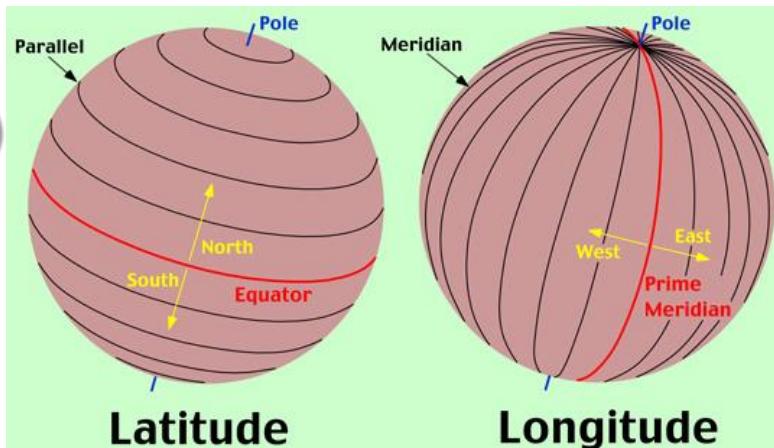
Method C: (Best-Looking) Degenerate Stepped Spiral

- Co-locate the two vertices that start and end the two rings: ($v6 == v7$)
- Why?
 - collapses both ugly ‘step’ triangles to zero-area lines;
 - separates the 2 strip colors entirely ($v6$ bottom; $v7$ top)
 - No weird ‘step’ – single-point joins adjacent strips
- RESULT: orderly tessellations & coloring make smooth & ‘seamless’ shapes



Shape Tessellation(13): `gl.TRIANGLE_STRIP`

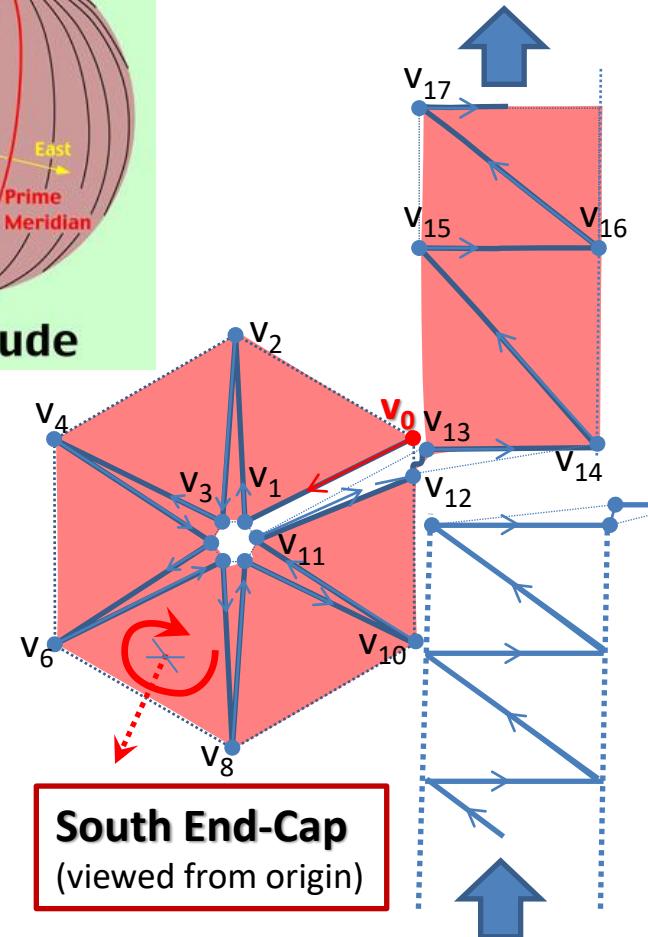
Method C: (cont'd)



Make a Sphere:

- End-caps at poles:
 - ‘South’ pole: $z=-1$, latitude -90°
 - ‘North’ pole: $z=+1$, latitude $+90^\circ$
- And slices of constant latitude:
 - $\cos(\text{lat})$ sets slice radius,
 - $\sin(\text{lat})$ sets slice z coordinate
- other ways:

https://en.wikipedia.org/wiki/Spherical_polyhedron

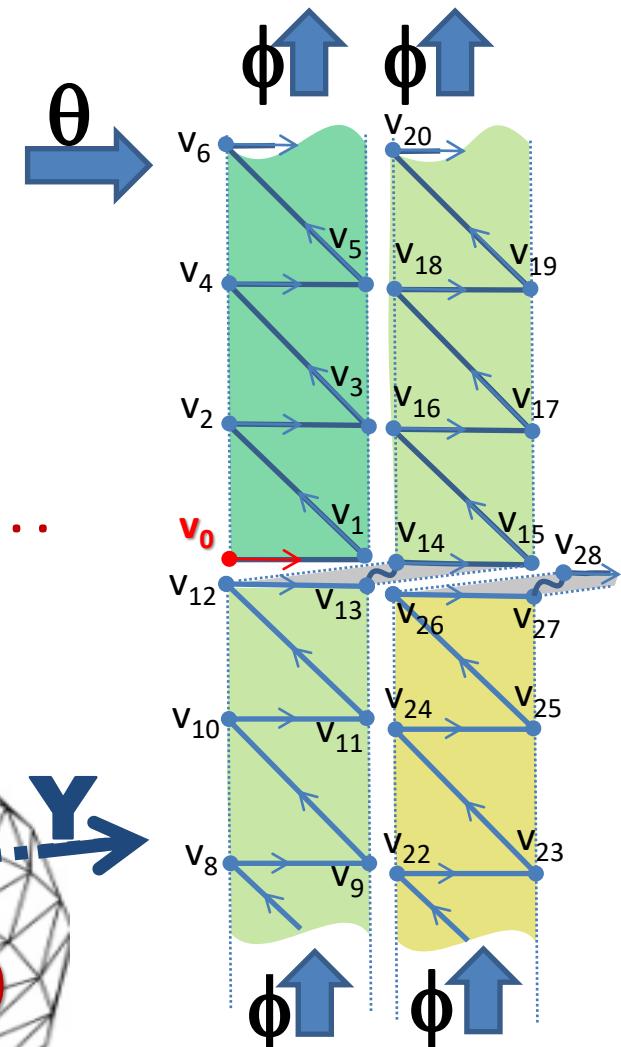
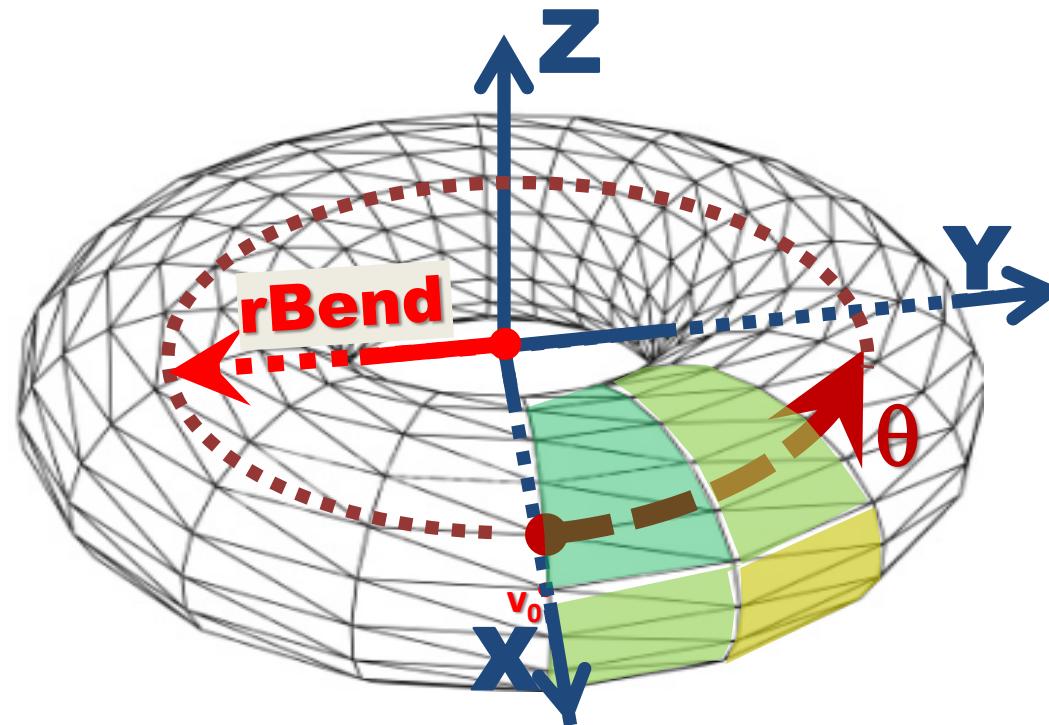


Longitude Rings
(wraps around
the base-cap)

Shape Tesselation(14): `gl.TRIANGLE_STRIP`

Method C: (cont'd) Make a Torus:

- Bend 'tube' CCW around $+z$ axis by angle θ
- Tube centerline begins at $+x$ axis at $(rBend, 0, 0)$, and forms a CCW circle in $z=0$ plane, with radius $rBend$
- To make the tube, 'wrap' the centerline in a Degenerate Stepped Spiral, circling tube by angle $\theta \dots$

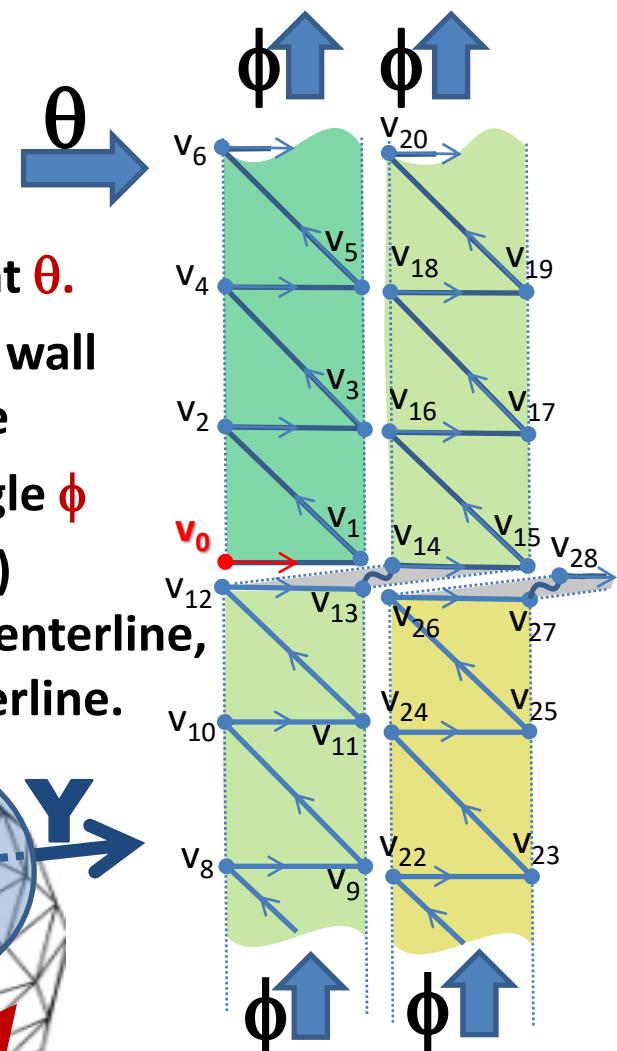
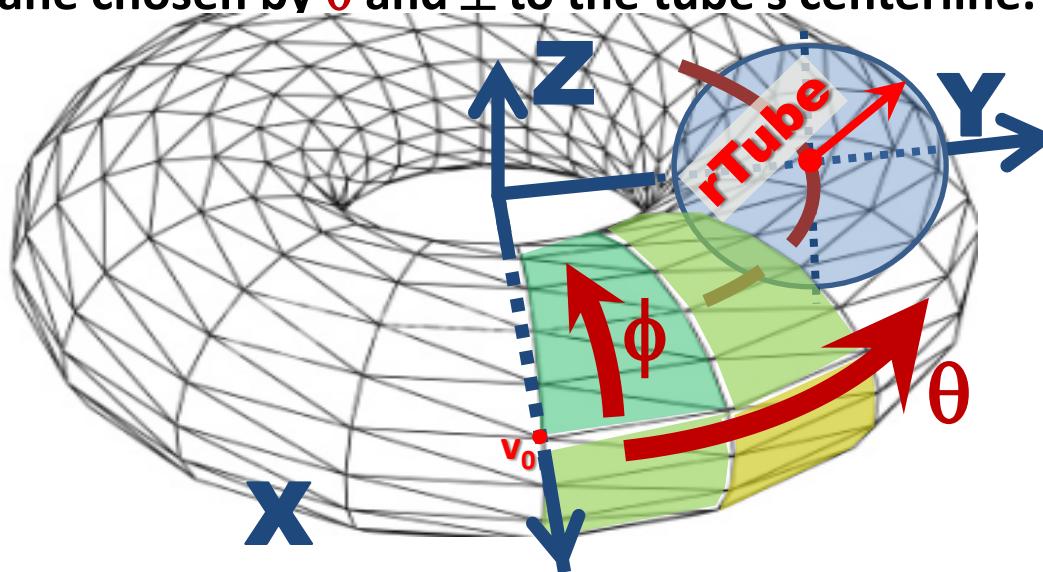


Ring Walls
(varied diameters)

Shape Tesselation(15): `gl.TRIANGLE_STRIP`

Method C: (cont'd) Make a Torus:

- Each step of the Degenerate Stepped Spiral circles the bent tube's centerline in a ring of constant θ .
- Angle ϕ increases for every new vertex pair in a ring wall
Angle θ increases for every new ring wall in the tube
- Each ring has radius **rTube** and and ring-defining angle ϕ
- The first ring begins +x axis at $v_0 = (rBend+rTube, 0, 0)$
made from a strip that circles **CW**around the tube centerline,
within a plane chosen by θ and \perp to the tube's centerline.

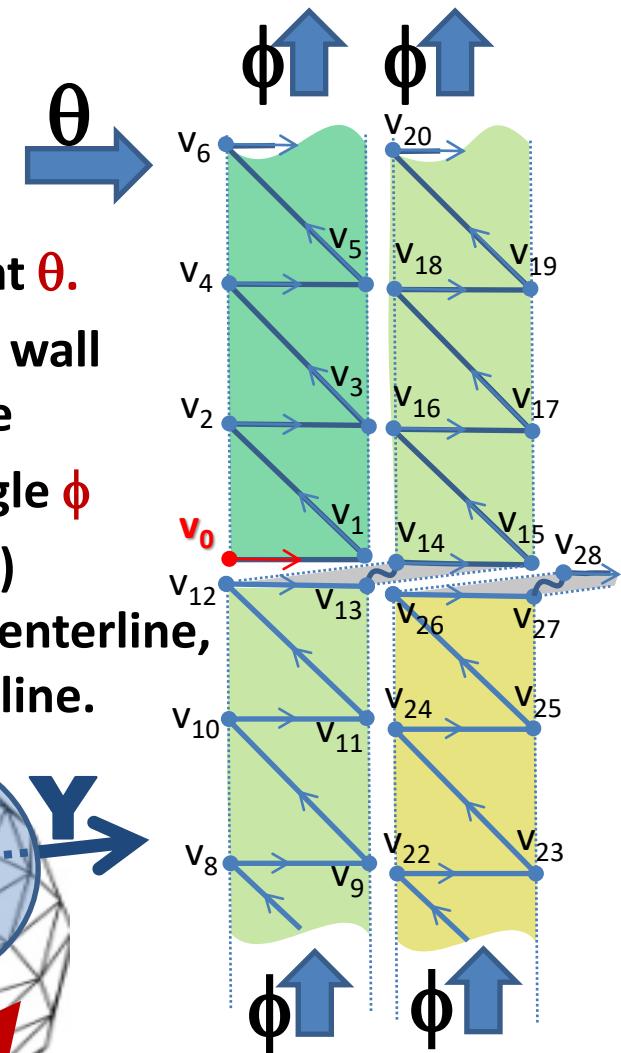
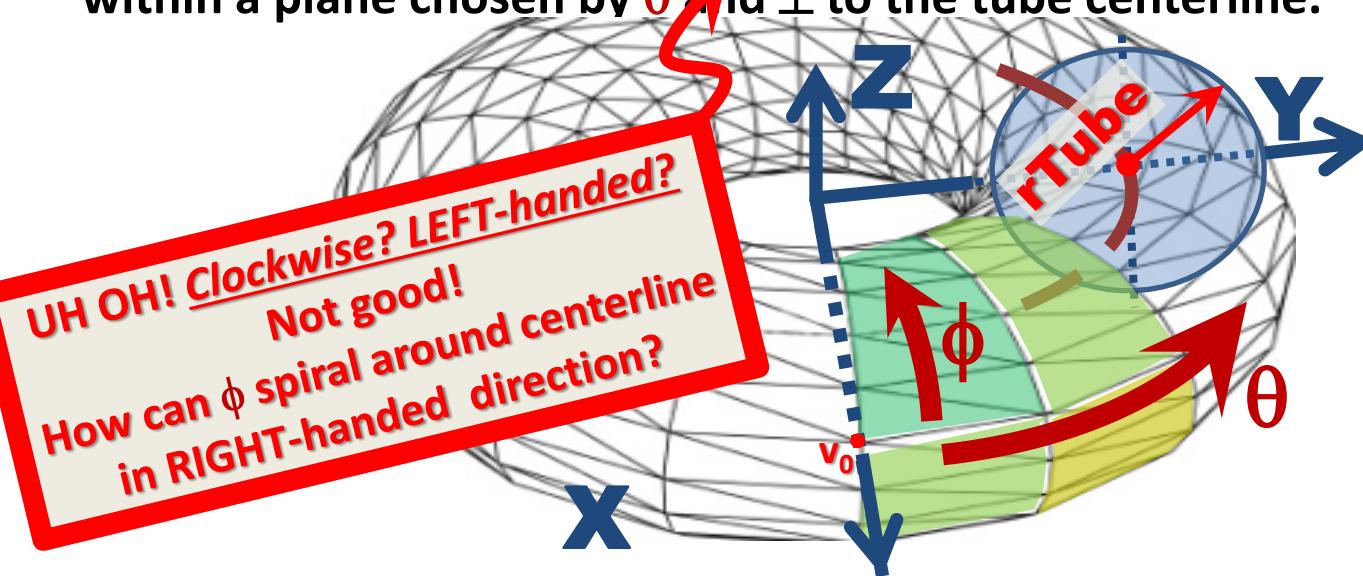


Ring Walls
(varied diameters)

Shape Tesselation(16): `gl.TRIANGLE_STRIP`

Method C: (cont'd) Make a Torus:

- Each step of the Degenerate Stepped Spiral circles the bent tube's centerline in a ring of constant θ .
- Angle ϕ increases for every new vertex pair in a ring wall
Angle θ increases for every new ring wall in the tube
- Each ring has radius **rTube** and and ring-defining angle ϕ
- The first ring begins $+x$ axis at $v_0 = (rBend+rTube, 0, 0)$
made from a strip that circles **CW**around the tube centerline,
within a plane chosen by θ and \perp to the tube centerline.



Ring Walls
(varied diameters)

Shape Tesselation(17): **gl.TRIANGLE_STRIP**

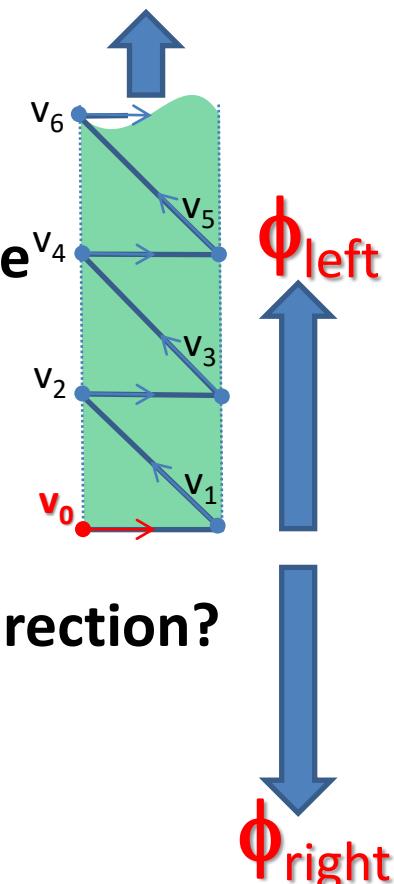
Method C: (cont'd) Make a **Torus**:

UH-OH #1:

Stepped spiral wraps around the tube centerline
in the clockwise (CW, left-handed) direction.

Could we wrap around the tube

in the counter-clockwise (**CCW** right-handed) direction?



Shape Tesselation(18): `gl.TRIANGLE_STRIP`

Method C: (cont'd) Make a **Torus**:

UH-OH #1:

Stepped spiral wraps torus centerline
in the clockwise (CW, left-handed) direction.

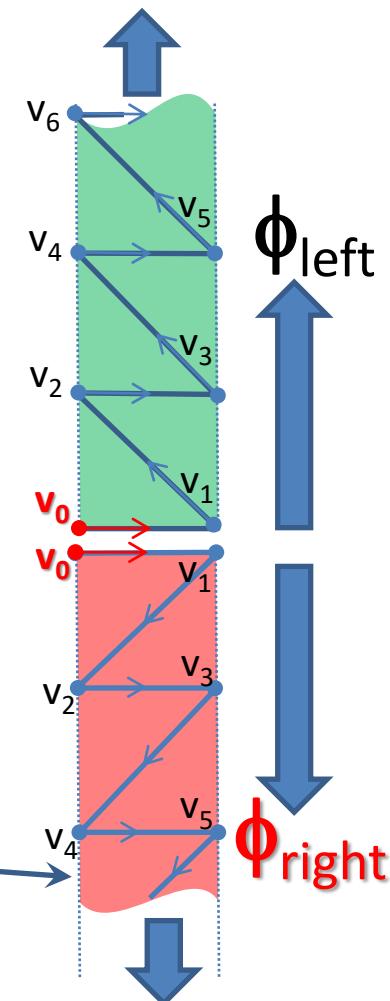
Hmm.

Let's make rings in the opposite ϕ direction:

UH-OH #2:

right-handed spiral gives wrong winding order;
we get **back-facing** triangles

How could we solve this (very common) problem?



Shape Tesselation(19): **gl.TRIANGLE_STRIP**

Method C: (cont'd) Make a Torus:

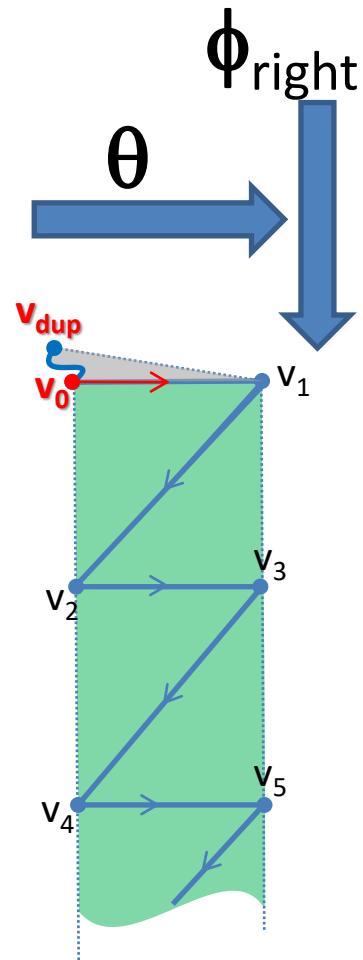
UH-OH #2:

right-handed spiral gives wrong winding order;
we get **back-facing triangles**

How could we solve this (very common) problem?

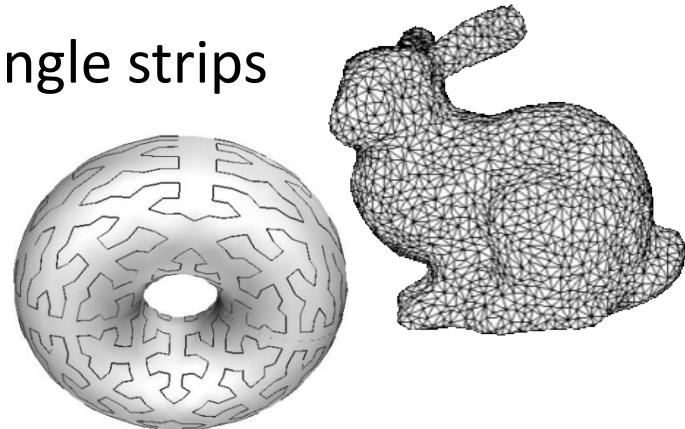
ANSWER: *DUPLICATE* the starting vertex

- creates a degenerate (zero-area) first triangle
- reverses the winding order of all subsequent triangles!



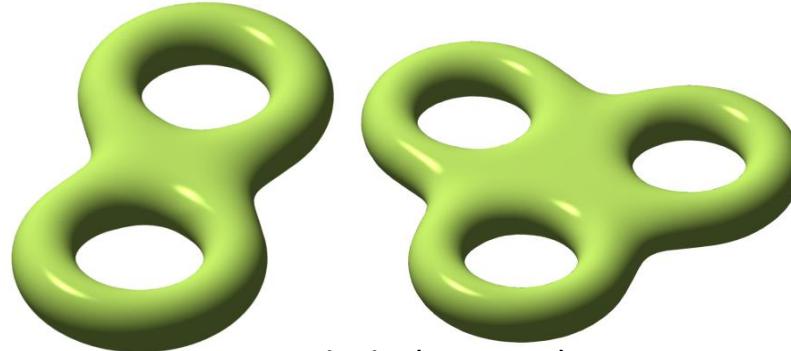
Shape Tesselation: Other Ways

Many more ways to **tessellate** shapes into triangle strips
(web-search: ‘tesselation’ ‘triangulation’ etc
—you’ll find research papers)

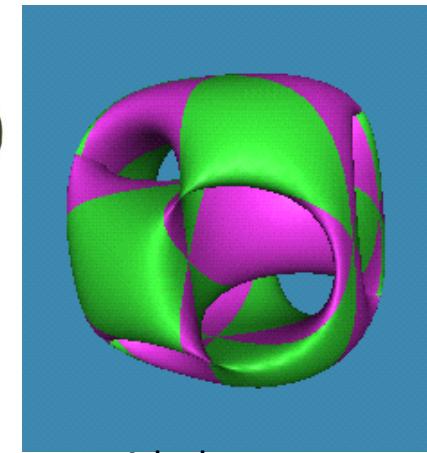


Challenges:

- Can you tessellate a sphere with two interleaved tri- strips?
- a tetrahedron?
- a cube?
- a sphere?
- a torus?
- a 2,3 or 4-hole torus?



2-hole (genus 2) torus,
3-hole “Pretzel” torus
by Oleg Alexandrov



4-hole torus
Andrew J Hanson

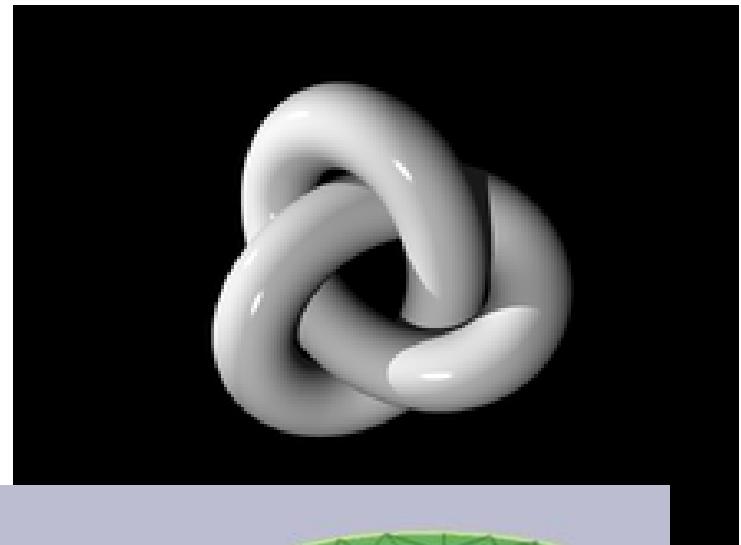
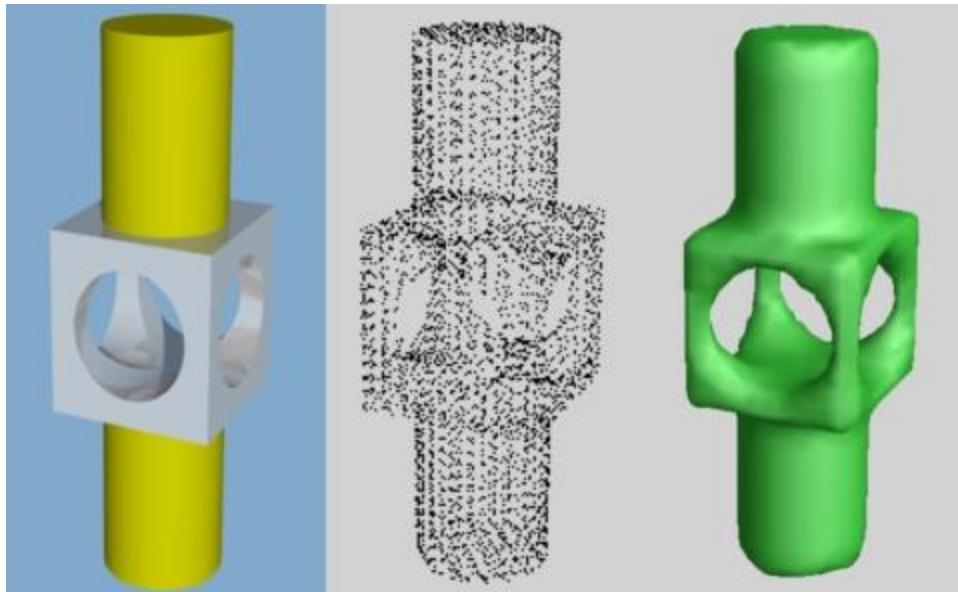
or, can you make these real illusion-causing shapes?:

<https://youtu.be/vCt508hknuU?t=1m1s> ? <https://youtu.be/S5fPwE7GQOA>?

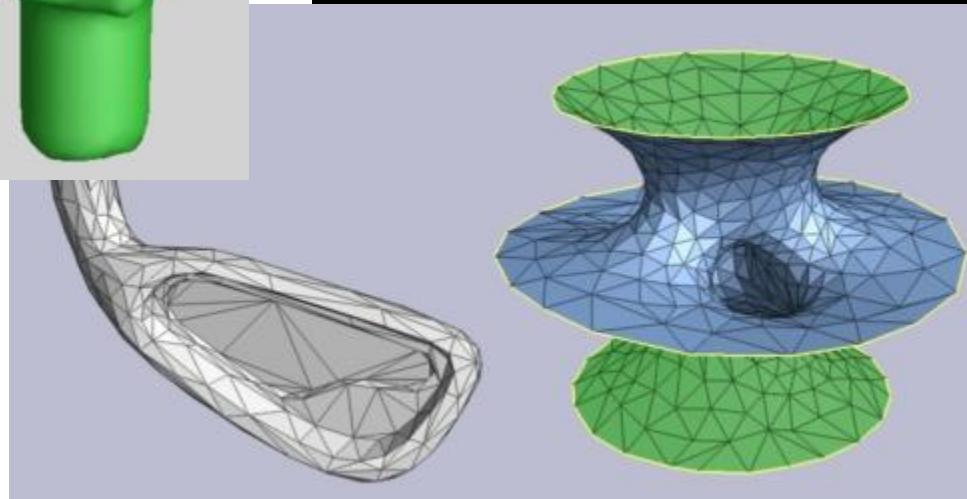
Will they look like the video if you draw them in the WebGL CVV? why/why not?

Topology is hard / Grids are easy

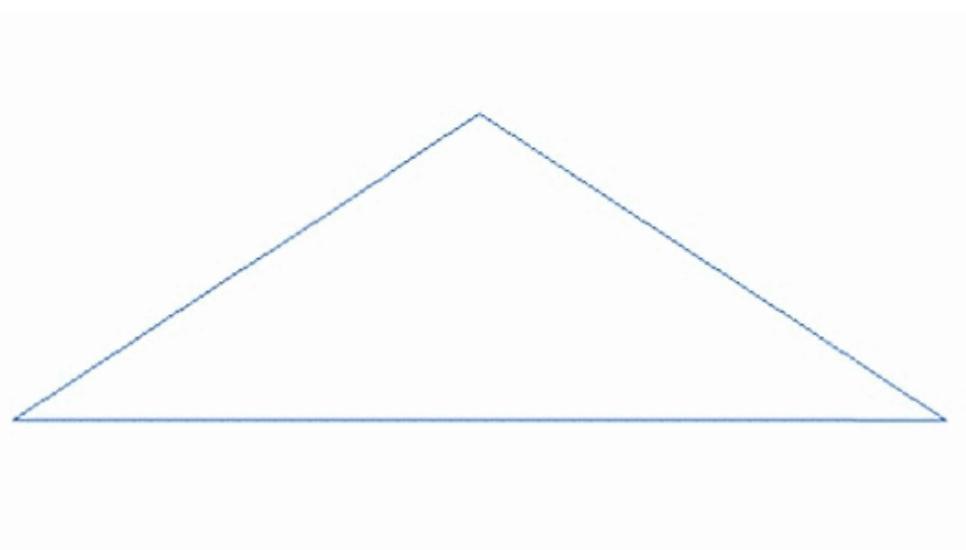
- Closed surfaces get messy, especially with topological ‘holes’:



Source: Hughes Hoppe,
Microsoft Research.
(look him up on Google!)



Topology is hard / Grids are easy

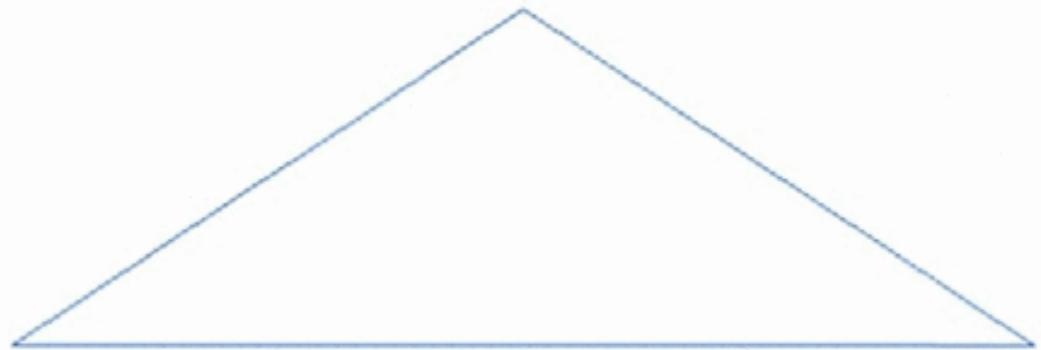


- Height-fields and/or subdivision for terrain are easy!
 - vertices uniformly spaced in x,y
 - vertex z value set by sum of a few randomly-chosen sinusoids (Google ‘terrain height-field’)
 - vertex colors set by sum of a few randomly-chosen RGB sinusoids

HINT: Topology is hard/Grids are easy

- Height-fields for terrain/water/waves ? easy!

http://en.wikipedia.org/wiki/Fractal_landscape



- Like 3D geometry?

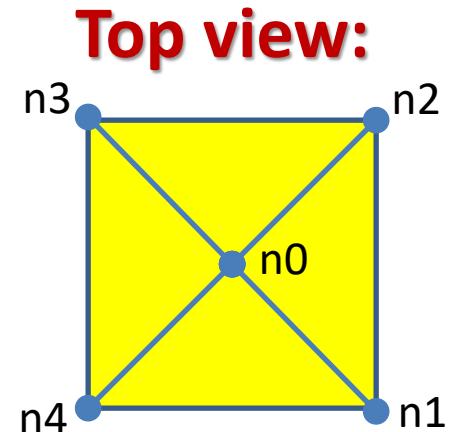
Meet a modest master of it: **Hughes Hoppe**

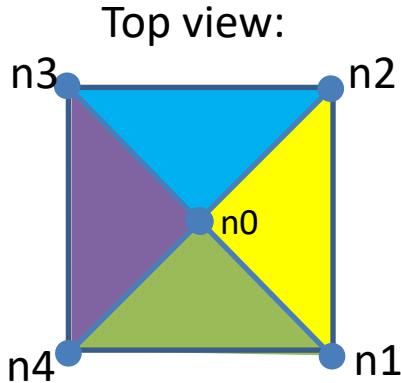
<http://research.microsoft.com/en-us/um/people/hoppe/>

Exercise:

Without writing any code,
tesselate
a 4-sided pyramid with a square base
into **one** triangle strip:

- Use this set of 5 node locations
- Create a triangle-strip vertex/node sequence
that creates all faces of the pyramid,
with correct winding order to make
all triangle front faces point outwards.

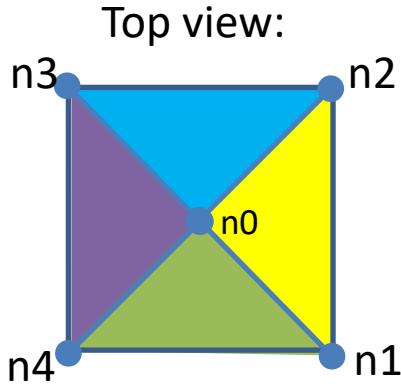




SOLUTION(S):

?Every way to ‘cut’ the pyramid into CCW non-degenerate tri-strip(s)?

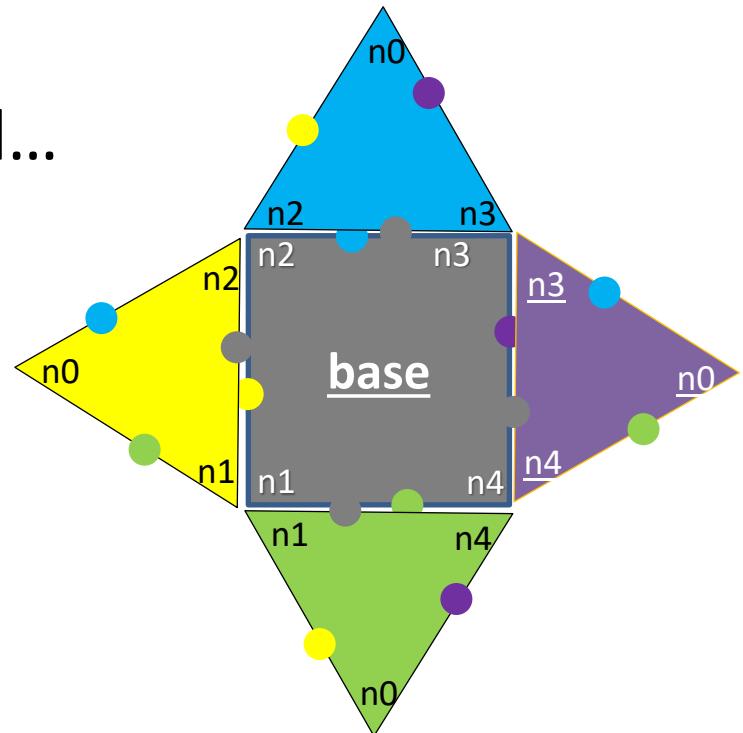
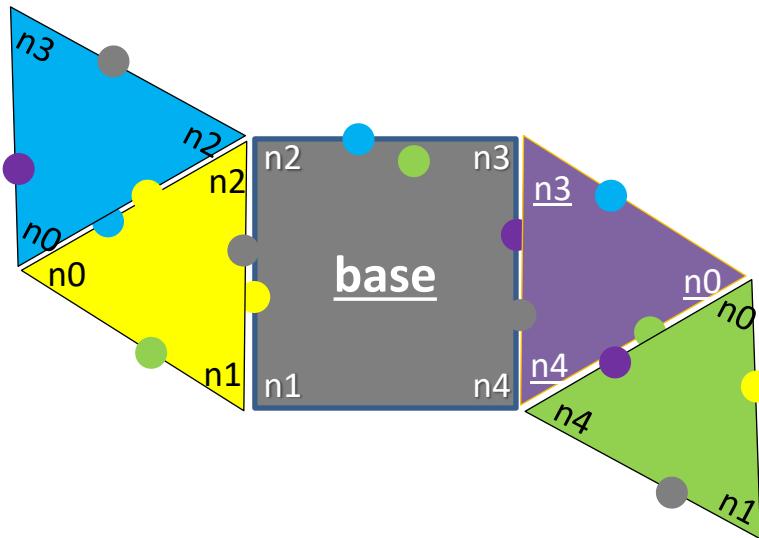
- 1st node: n0, or any of 4 symmetric others (n1,n2,n3,n4). Let’s try just n0 or n1.
- 2nd node: $n0 \rightarrow n1$; or $n1 \rightarrow n0$, $n1 \rightarrow n2$, $n1 \rightarrow n4$
- 3rd node: MUST have CCW winding order:
 $n0 \rightarrow n1, n2$; or $n1 \rightarrow n0, n4$; or $n1 \rightarrow n2, n0$; or
 $n1 \rightarrow n4, n2$;



Some SOLUTION(S):

?Every way to 'cut' the pyramid into CCW non-degenerate tri-strip(s)?

- 1st node: n0, or any of 4 symmetric others (n1,n2,n3,n4).
2 unique choices: n0 or n1.
- 2nd node: n0 → n1 or symmetrical...



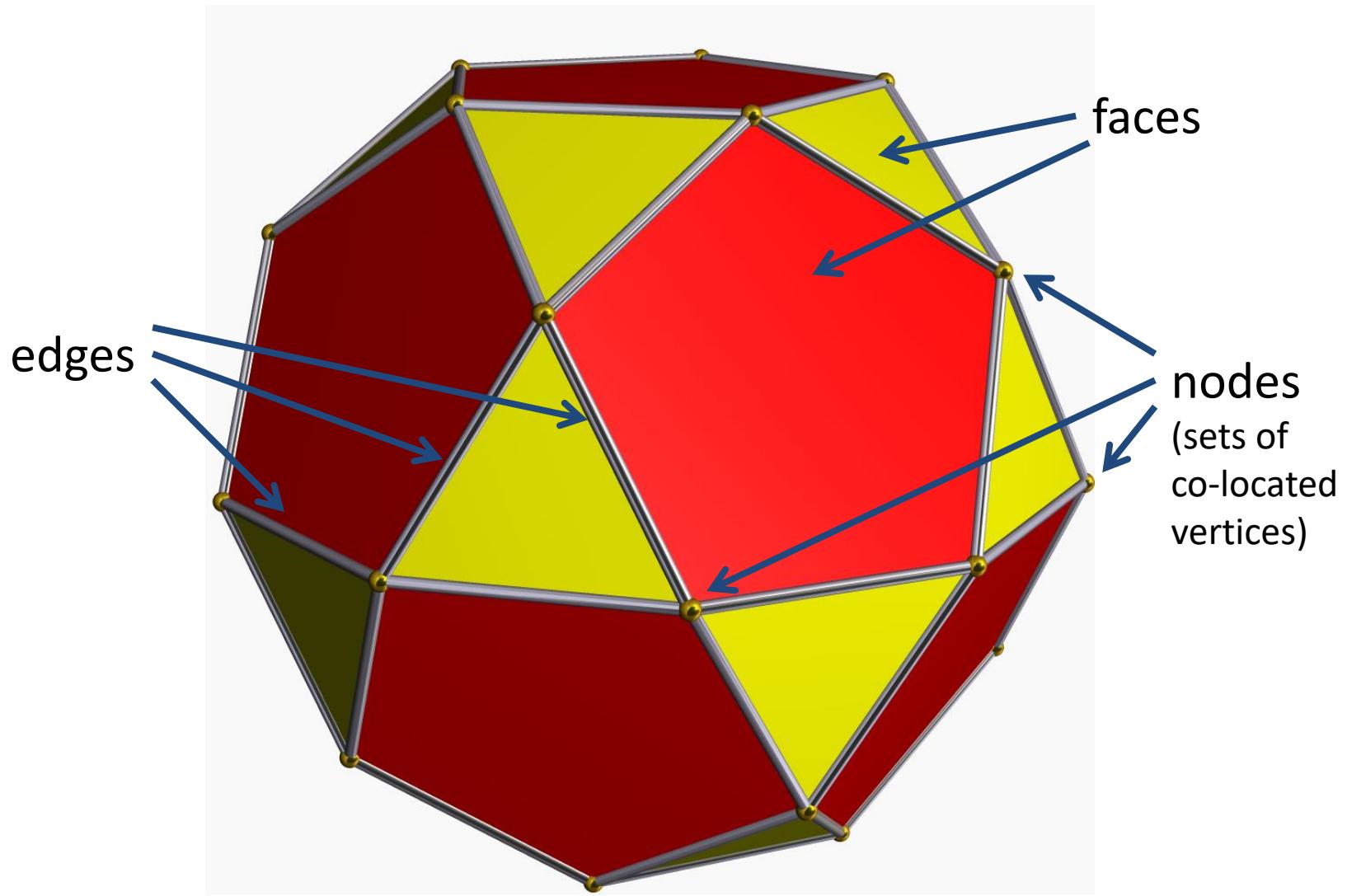
B-REP Mesh Geometry: Winged Edge & Half-Edge

Jack Tumblin

Northwestern Univ COMP_SCI 351-1

Fall 2021

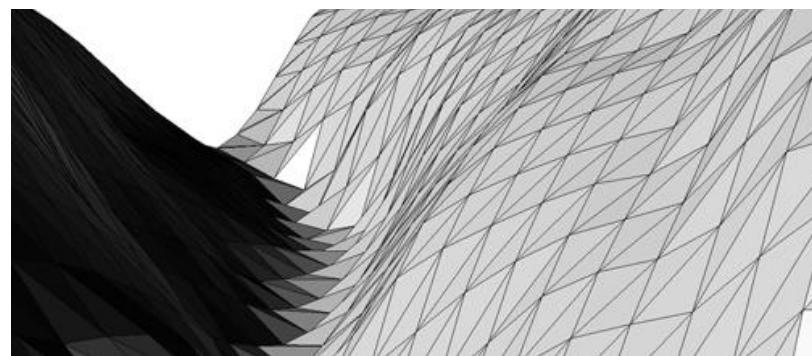
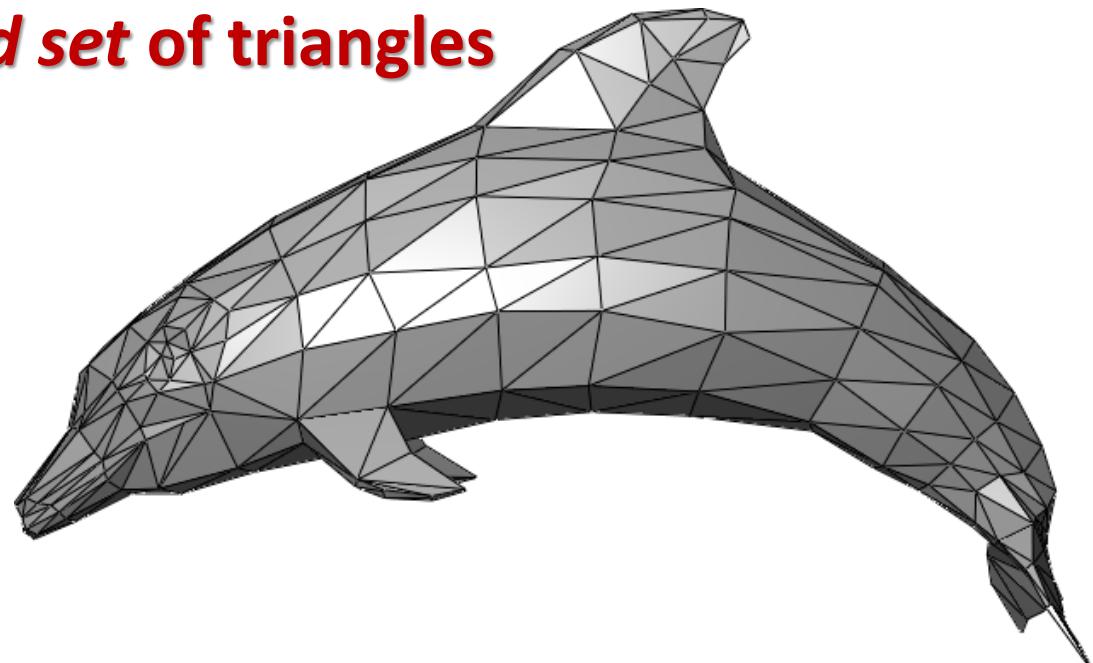
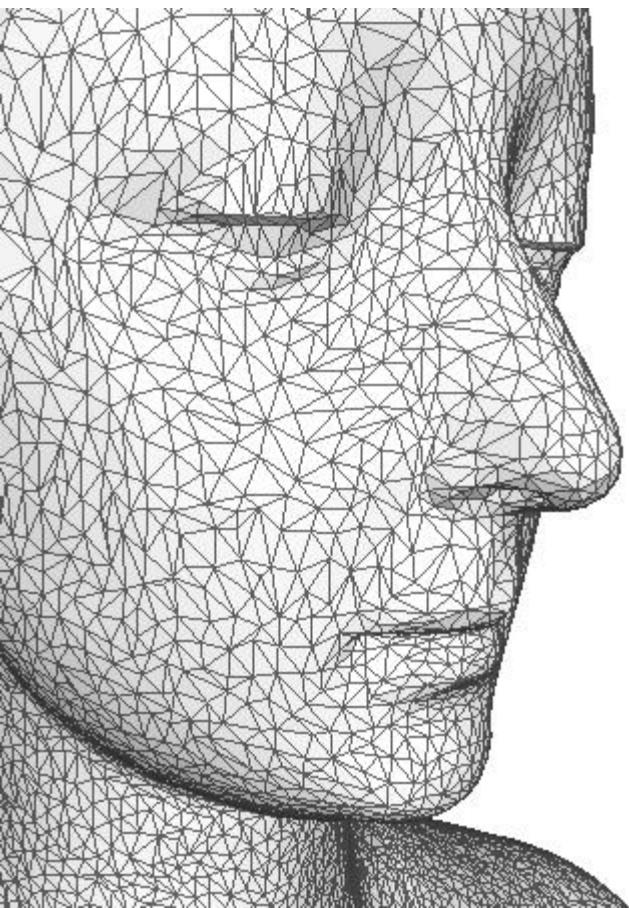
3D objects: Consider this Sphere...



Q: At any given vertex, how should we compute the 'surface normal' vector?
(\perp to just one face? Which one? \perp to the ideal sphere? How do we find it?) ²

Triangle meshes

- Many freeform shapes consist of ***connected*** triangles:
a ‘mesh’ == *connected set of triangles*



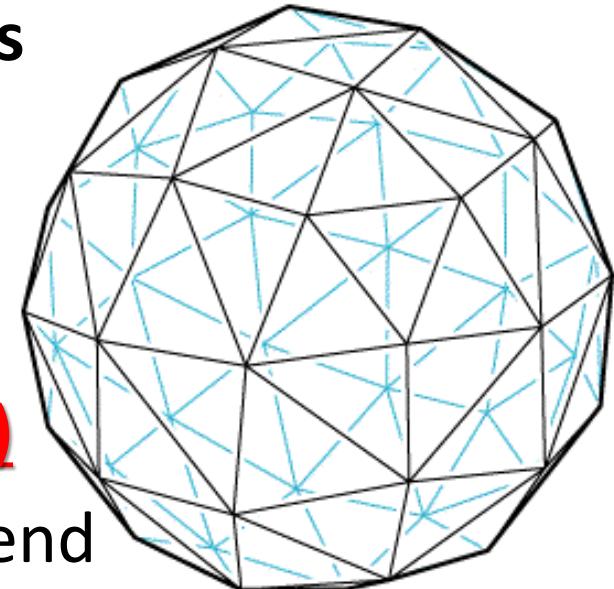
Is a ‘Triangle List’ Enough? no.

- Triangle Lists can render in parallel, independently:
- Good for fast beautiful rendering in WebGL/OpenGL
 - **GOOD**: quick to store, to draw, & to extract from shapes
 - **BAD** for shape creation, editing, geometric descriptions
 - **BAD** for shape queries: geodesics, tri-strip creation, etc.
- Discards all “connectivity” of vertices, edges, & faces (!Adjacency-finding may force whole-model search!!)

Basic Terminology: how do we best describe
3D Geometry assembled from Triangles?

3D objects as thin hollow shells

- Triangle ‘shells’ describe only a shape **boundary**
- The ‘interior’ solid defines the bounded subspace
- Triangle meshes set only *linear or planar* boundaries:
defined by **faces**, **edges**, and **vertices**
(and sometimes **nodes**:
sets of co-located vertices)



- **‘Boundary Representations’ (B-REP)**

from faces, edges and vertices can extend

‘free-form’ curved edges & surfaces too!

(https://en.wikipedia.org/wiki/Freeform_surface_modelling)

Geometric Computing?

Triangle Lists woefully inadequate...

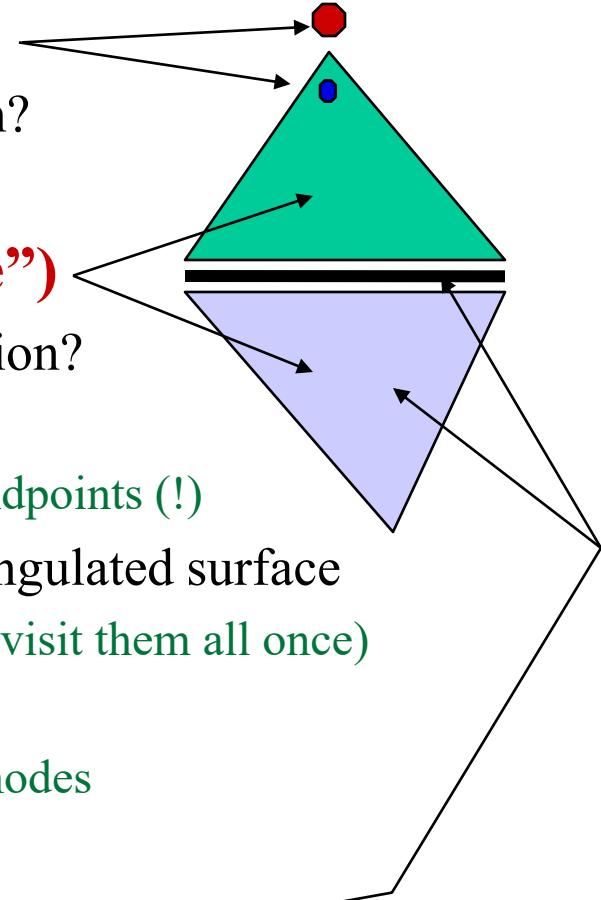
How can we organize vertices, edges, triangles, faces to more easily answer deeper questions, such as:

- what triangles share a given edge?
 - What triangles are separated by only 2 triangles?
 - What ‘faces’ have exactly 1 triangle? two? three? four?
 - What set of triangles meets at one given node (3D location)?
 - What triangles outside a face share an edge with that face?
 - What’s the ‘geodetic’ (shortest-possible curve on the surface) from a point inside one face to a point inside another face?
 - And many more!
- See: <https://www.cgal.org/> , <https://www.meshlab.net/>

Connectivity == Incidence + Adjacency

- **Triangle/vertex incidence (at a “node”)**

- What other vertices share this same 3D location?
 - Defines Corners



- **Triangle/triangle adjacency (at an “edge”)**

- What other triangles share this same edge location?
 - Neighboring triangles share a common edge
 - But NOT strictly required to share same edge endpoints (!)
 - Adjacency data can accelerate **traversal** of triangulated surface
 - “Walk” from one triangle to an adjacent one (& visit them all once)
 - **Used to build triangle strips**
 - Used to estimate surface normals at vertices or nodes
 - Used to compress triangulated surfaces

- **Triangle/edge incidence (border)**

- Associates triangles with their bounding edges. Useful for ‘wireframes’

Triangle meshes

- How can we best represent/store triangle *meshes*?
 - To tell ***ALL*** ways faces, edges, and vertices connect?
 - Can it be efficient, complete, and non-redundant?
-

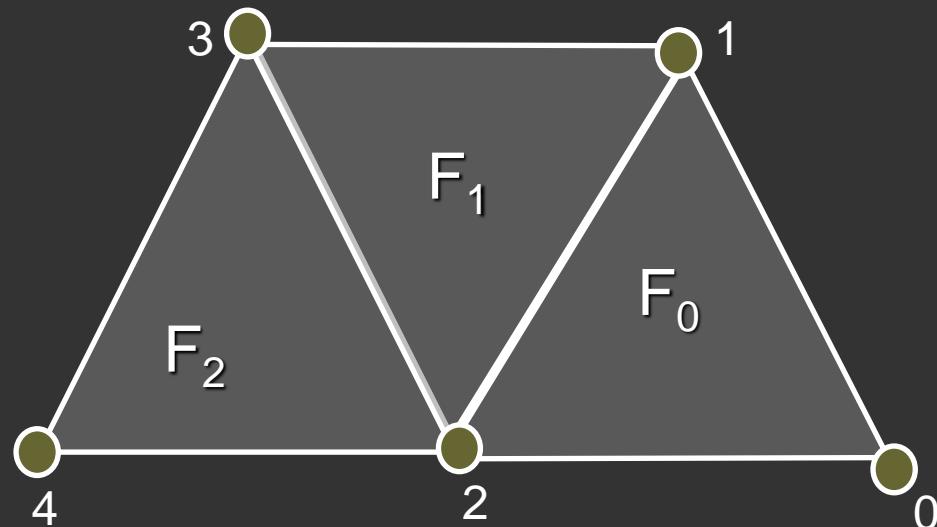
MANY possibilities; you can even invent your own...

- List of independent faces (e.g. triangles), or...
- Vertices indexed for re-use; index list for each face...
- Face- adjacency, edge-adjacency, or its variants...
- ***Winged-Edge structure*** (old, reliable, widely used)
- ***Half-Edge structure*** (more recent, simpler, deft)

Independent Faces ('Vertex Lists')

Each face is a simple list of vertex coordinates;

- ++Trivial to render in WebGL/OpenGL
- -- Redundant vertices
- -- No connectivity information; slow/messy to derive it



Face Table		(==a vertex list)
F_0 :	$(x_0, y_0, z_0), (x_1, y_1, z_1), (x_2, y_2, z_2)$	
F_1 :	$(x_2, y_2, z_2), (x_1, y_1, z_1), (x_3, y_3, z_3)$	
F_2 :	$(x_2, y_2, z_2), (x_3, y_3, z_3), (x_4, y_4, z_4)$	

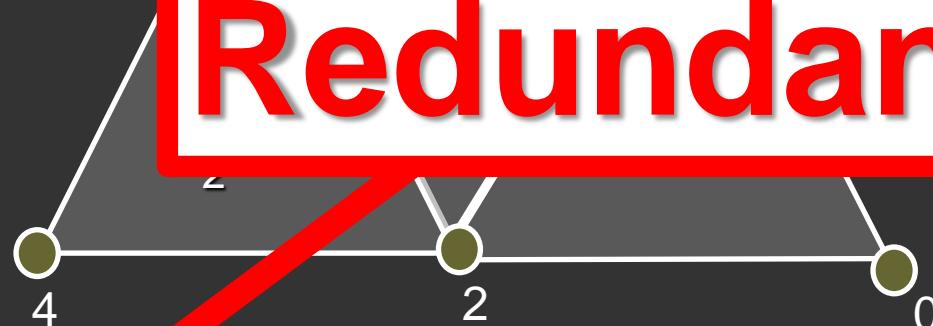
Independent Faces ('Vertex Lists')

Faces list vertex coordinates only

- Trivial to render in WebGL/OpenGL
- Trivial to convert to other formats
- Trivial to edit

VERBOSE!

**~6X vertex count,
Redundant Edges**

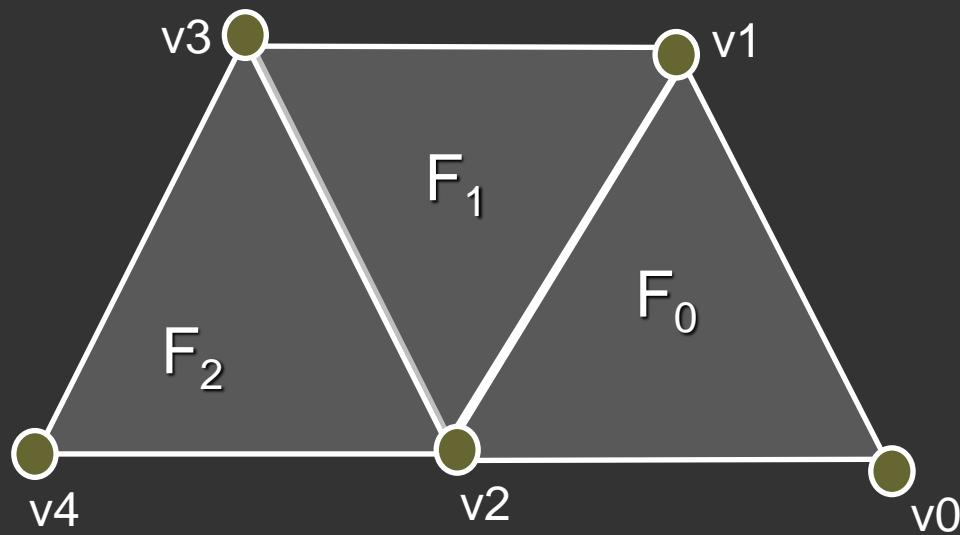


$\Gamma_2: (x_2, y_2, z_2), (x_3, y_3, z_3), (x_4, y_4, z_4)$

vertex list)
(x_2, y_2, z_2)
(x_3, y_3, z_3)
(x_4, y_4, z_4)

Indexed Face Sets ('incidence table')

- Faces as vertex# lists. “shared vertices” (nodes)
- Commonly used (e.g. ‘OBJ’ & ‘OFF’ file formats)
- easy-to-use for mesh processing (esp. if augmented)



Vertex Table		Face Table	
v_0 :	(x_0, y_0, z_0)	F_0 :	0, 1, 2
v_1 :	(x_1, y_1, z_1)	F_1 :	2, 1, 3
v_2 :	(x_2, y_2, z_2)	F_2 :	2, 3, 4
v_3 :	(x_3, y_3, z_3)		
v_4 :	(x_4, y_4, z_4)		

Note CCW ordering

A white arrow points from the text "Note CCW ordering" to the vertex index list for face F_2 in the Face Table, which lists indices 2, 3, and 4. This indicates that the vertices are listed counter-clockwise.

~~Indexed Face Sets ('incidence table')~~

- Faces listed by vertex # – “shared vertices”

“mmm...MMEH!”

**Redundant edges,
Awkward, slow
mesh traversal**



Note CCW ordering

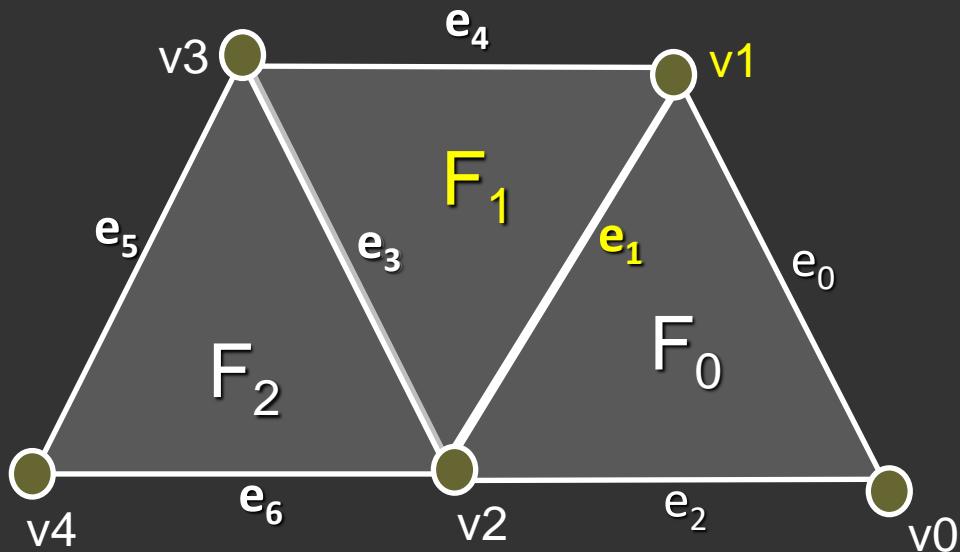
Face Table

0, 1, 2
2, 1, 3
2, 3, 4
4, 3, 0

(rotated)

Full Adjacency Lists

- Store **all** vertex, face, and edge adjacencies



Edge Adjacency Table

e ₀ :	v ₀ , v ₁	∅, F ₀	∅, e ₂ ; e ₁ , ∅;
e ₁ :	v ₁ , v ₂	F ₁ , F ₀	e ₅ , e ₀ ; e ₂ , e ₆
⋮	⋮	⋮	⋮

Face Adjacency Table

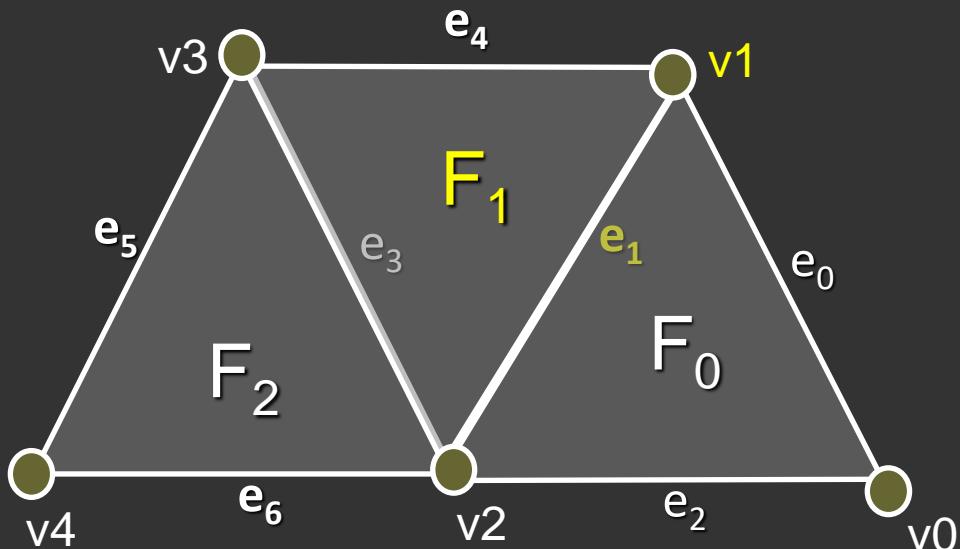
F ₀ :	v ₀ , v ₁ , v ₂	∅, F ₁ , ∅	e ₀ , e ₁ , e ₂
F ₁ :	v ₂ , v ₁ , v ₃	F ₀ , ∅, F ₂	e ₁ , e ₄ , e ₃
F ₂ :	v ₂ , v ₃ , v ₄	F ₁ , ∅, ∅	e ₃ , e ₅ , e ₆

Vertex Adjacency Table

v ₀ :	v ₁ , v ₂	F ₀ , ∅	e ₀ , e ₂
v ₁ :	v ₃ , v ₂ , v ₀ , ∅	F ₁ , F ₀ , ∅	e ₄ , e ₁ , e ₀
⋮	⋮	⋮	⋮

Full Adjacency Lists? NO....

- INSTEAD OF
all vertex, face,
and edge adjacencies



Edge Adjacency Table

e ₀ :	v ₀ , v ₁	∅, F ₀	∅, e ₂ ; e ₁ , ∅;
e ₁ :	v ₁ , v ₂	F ₁ , F ₀	e ₅ , e ₀ ; e ₂ , e ₆
⋮	⋮	⋮	⋮

Face Adjacency Table

F ₀ :	v ₀ , v ₁ , v ₂	∅, F ₁ , ∅	e ₀ , e ₁ , e ₂
F ₁ :	v ₂ , v ₁ , v ₃	F ₀ , ∅, F ₂	e ₁ , e ₄ , e ₃
F ₂ :	v ₂ , v ₃ , v ₄	F ₁ , ∅, ∅	e ₃ , e ₅ , e ₆

Vertex Adjacency Table

v ₀ :	v ₁ , v ₂	F ₀ , ∅	e ₀ , e ₂
v ₁ :	v ₃ , v ₂ , v ₀ , ∅	F ₁ , F ₀ , ∅	e ₄ , e ₁ , e ₀
⋮	⋮	⋮	⋮

Full Adjacency Lists? NO....

- Store **all** vertex, face, and

Edge Adjacency Table

**DANGEROUS
REDUNDANCY!**

Inconsistent?

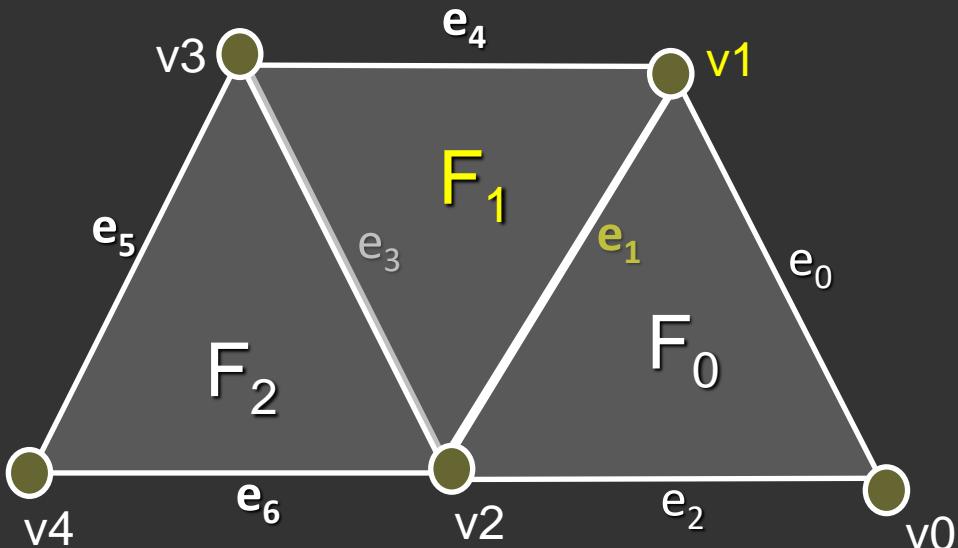
→ *undefined shape!*



$\begin{bmatrix} V_1: & [v_3, v_2, v_0, \emptyset] \\ \vdots & \vdots \end{bmatrix} \quad \begin{bmatrix} F_1: & [f_1, f_0, \emptyset] \\ \vdots & \vdots \end{bmatrix} \quad \begin{bmatrix} E_1: & [e_1, e_0] \\ \vdots & \vdots \end{bmatrix}$

Partial Adjacency Lists?

- Store *selected* adjacencies, use them to derive others
- Many possibilities, but...



Edge Adjacency Table

e ₀ :	v ₀ , v ₁	∅, F ₀	∅, e ₂ ; e ₁ , ∅;
e ₁ :	v ₁ , v ₂	F ₁ , F ₀	e ₅ , e ₀ ; e ₂ , e ₆
⋮	⋮	⋮	⋮

Face Adjacency Table

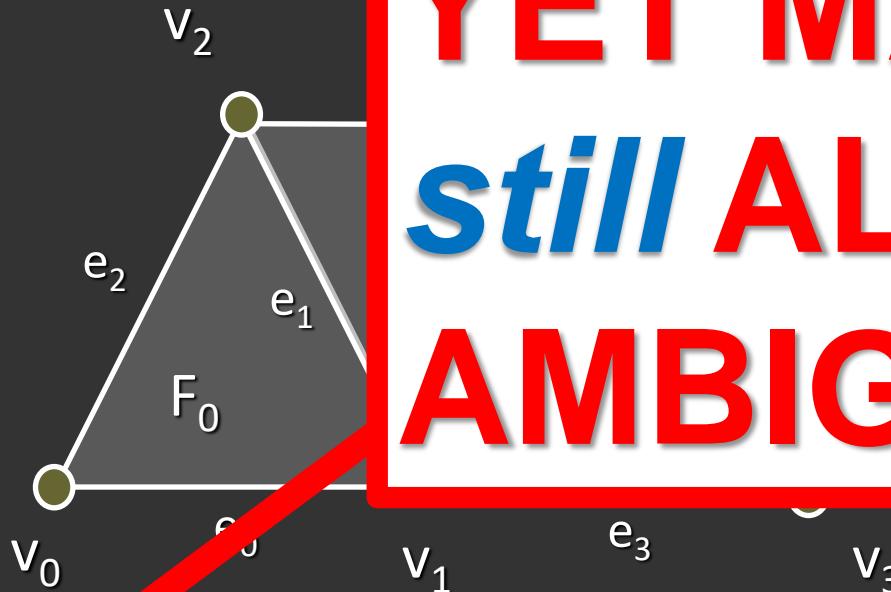
F ₀ :	v ₀ , v ₁ , v ₂	∅, F ₁ , ∅	e ₀ , e ₁ , e ₂
F ₁ :	v ₂ , v ₁ , v ₃	F ₀ , ∅, F ₂	e ₁ , e ₄ , e ₃
F ₂ :	v ₂ , v ₃ , v ₄	F ₁ , ∅, ∅	e ₃ , e ₅ , e ₆

Vertex Adjacency Table

v ₀ :	v ₁ , v ₂	F ₀ , ∅	e ₀ , e ₂
v ₁ :	v ₃ , v ₂ , v ₀ , ∅	F ₁ , F ₀ , ∅	e ₄ , e ₁ , e ₀
⋮	⋮	⋮	⋮

Partial Adjacency Lists? NO

- Store selected edges, use them
- Many possibilities



**COMPLEX,
YET MAY
*still ALLOW
AMBIGUITY!***

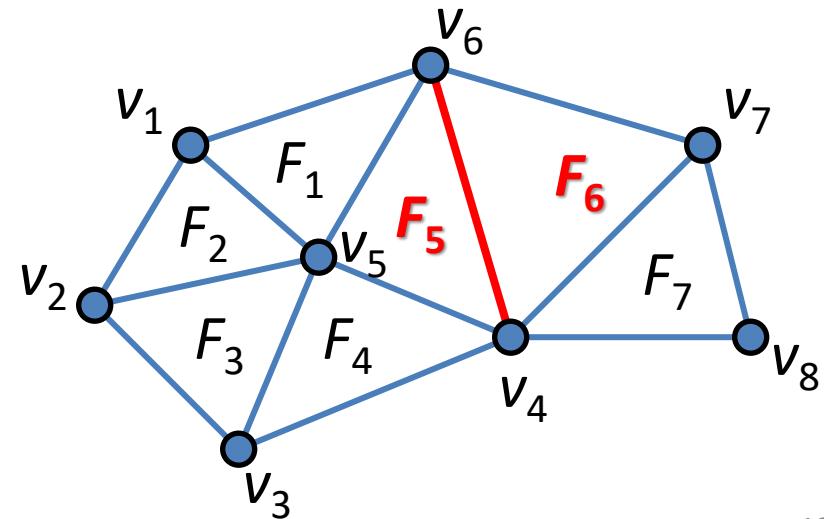
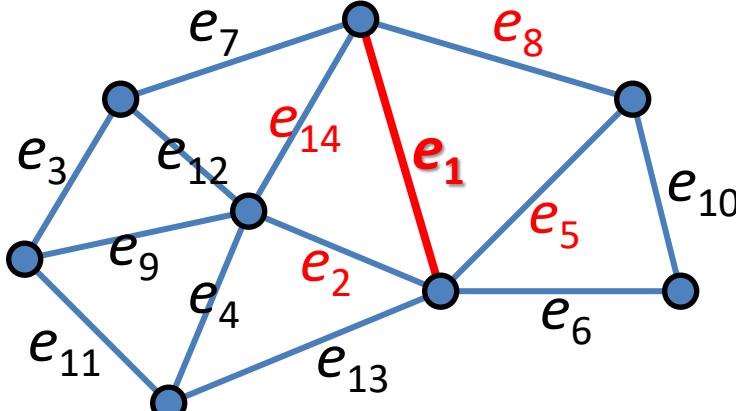
Adjacency Table
 $v_0: e_0, e_2, e_1, \emptyset$
 $v_1: e_5, e_0, e_2, e_6$
 \vdots

Adjacency Table
 $\emptyset; e_0, e_2, e_0$
 $F_2; e_6, e_1, e_5$
 $\emptyset; e_4, e_5, e_3$
 \vdots

Adjacency Table
 $v_0: v_1, v_2; F_0; e_0, e_2$
 $v_1: v_3, v_4, v_2, v_0; F_2, F_1, F_0; e_5, e_1, e_0$
 \vdots

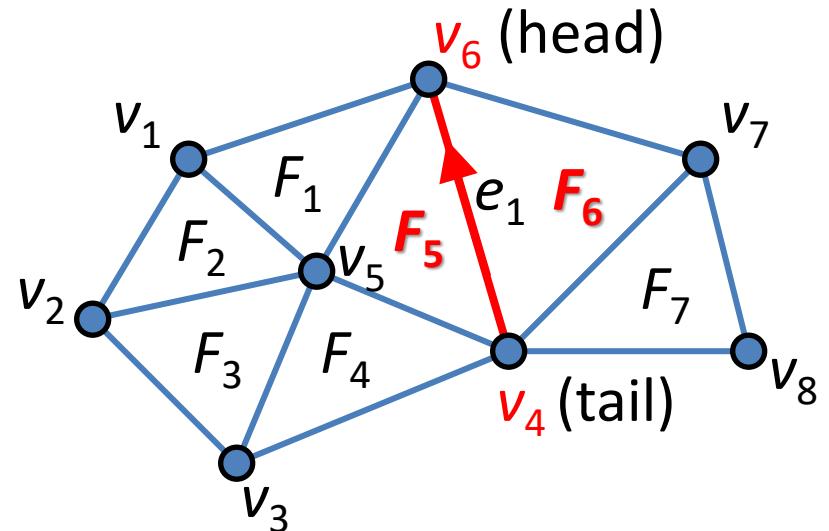
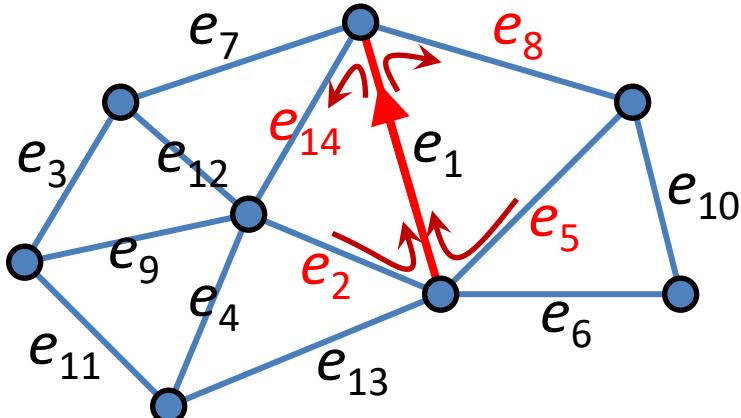
Solution 1: Winged-edge structure

- Stores ***connectivity at edges*** instead of at vertices
- For one edge, such as e_1 :
 - Two vertices define the edge: v_4 and v_6
 - Two faces (triangles) adjacent to edge: F_5 and F_6
 - Four edges identify those two faces: e_2, e_{14}, e_5 , and e_8



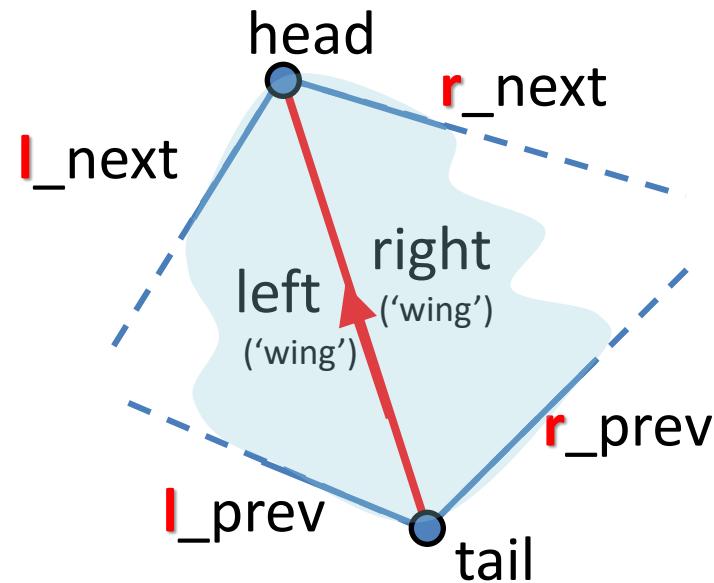
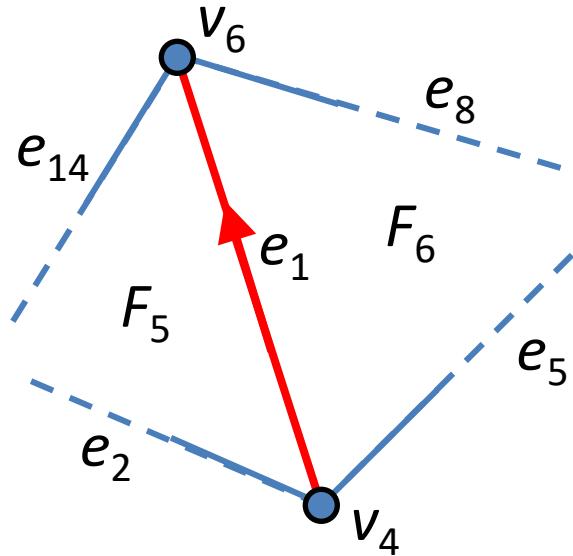
Solution 1: Winged-edge structure

- Give e_1 a (frequently arbitrary) direction, then
 - v_4 is the tail and v_6 is the head
 - F_5 is to the left and F_6 is to the right
 - e_2 is previous on the left side, e_{14} is next on the left side, e_5 is previous on the right side, and e_8 is next on the right side



Solution 1: Winged-edge structure

- Give e_1 a direction, then view it from above the surface:
 - v_4 is the **tail** and v_6 is the **head**.
 - Face F_5 is the **left** ‘wing’, and face F_6 is the **right** ‘wing’
 - e_2 is **previous on the left** side, e_{14} is **next on the left** side, e_5 is **previous on the right** side, and e_8 is **next on the right** side



Solution 1: Winged-edge scheme

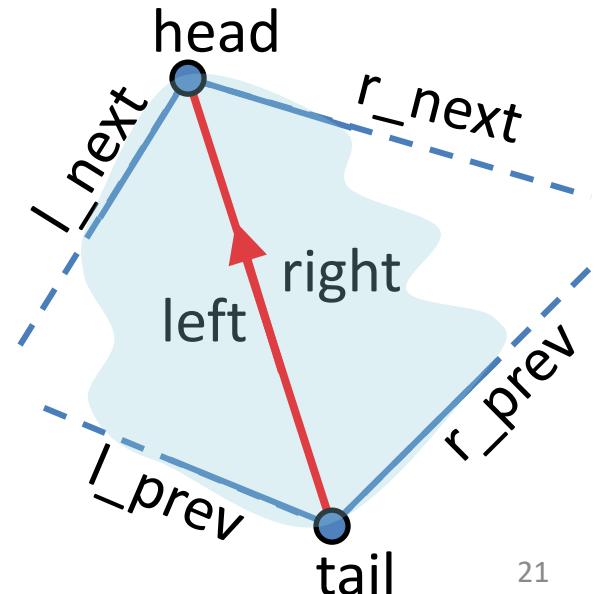
```
Edge { // Each edge holds “wings” indices:  
    Edge#     l_prev, l_next, r_prev, r_next;  
    Vertex#   head, tail;  
    Triangle# left, right;  
}
```

```
Vertex {  
    double x, y, z;  
    Edge# e; // any incident edge(just 1)  
} // (easy to trace out all the rest of them)
```

```
Triangle {  
    Edge# e; // any incident edge(just 1!)  
} // (easy to trace out all the rest of them)
```

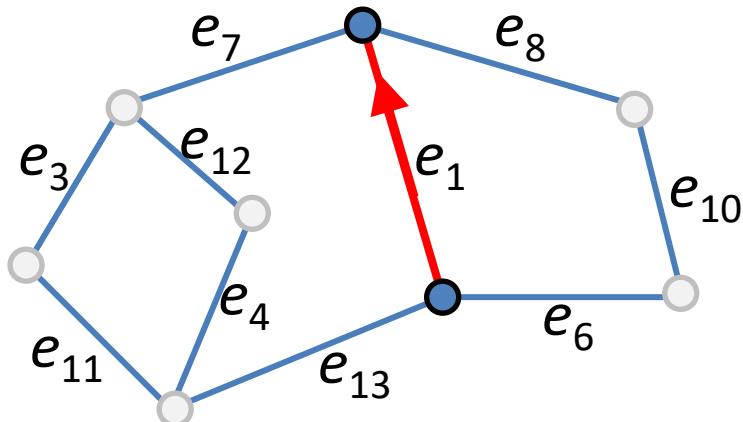
Winged-Edge B-REP:
--Array of Edge objects
--Array of Vertex objects
--Array of Triangle objects

Fixed-size data structs!
easy to store,
index and search.--



Solution 1: Winged-edge structure

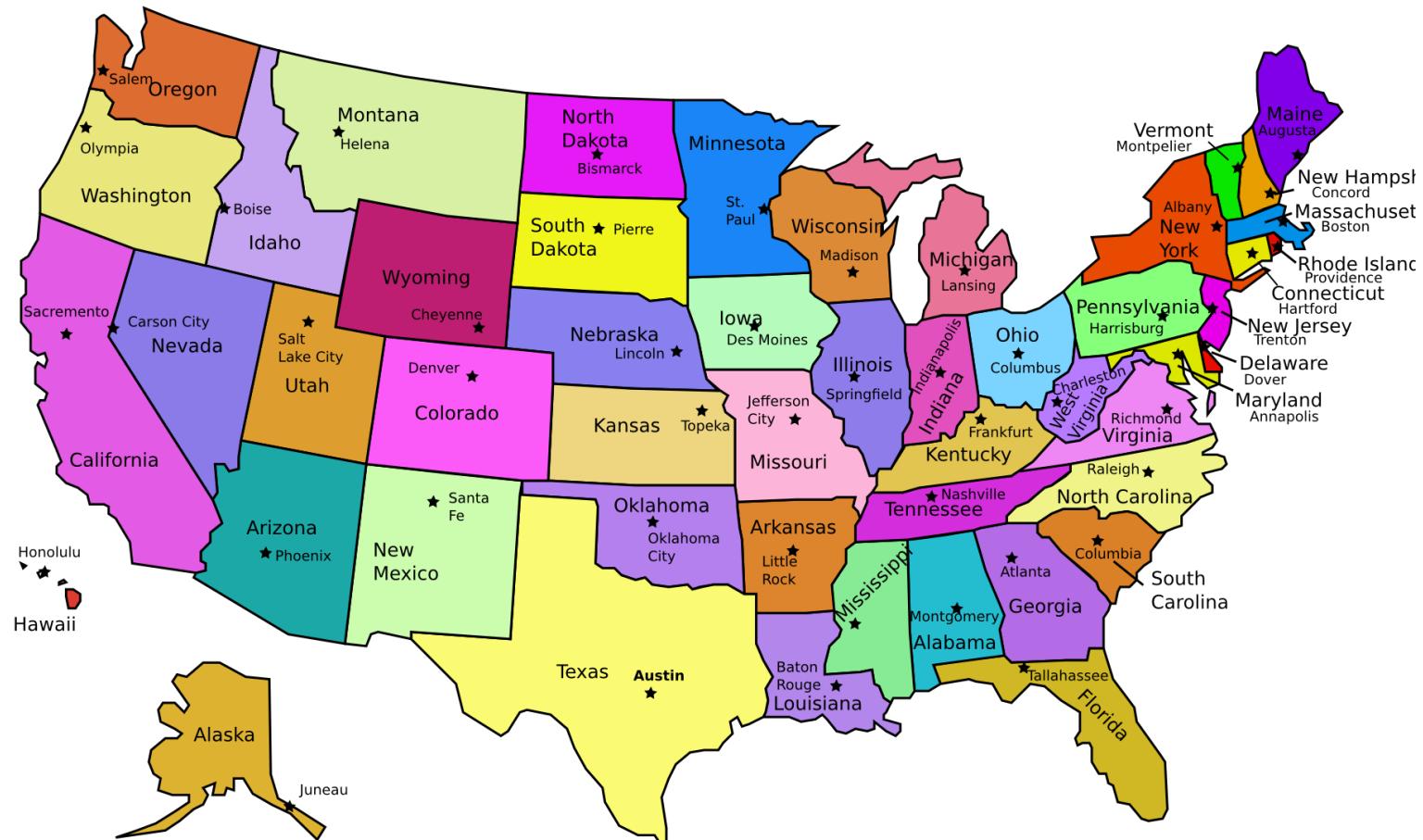
- **Also** good for meshes with arbitrary polygonal faces:
 - Still just one head and one tail per edge;
 - Still just one left face and one right face per edge;
 - Still just one previous edge and just one next edge for each of the two faces (left, right):
EXAMPLE: e_{13}, e_7 for left face, and e_6, e_8 for right face.



Note: The naïve triangle-list structure of VBOs alone lacks connectivity, & is tough to edit interactively.
Winged Edge is better!

Solution 1: Winged-edge structure

- Also good for planar partitions such as land maps
(widely used in digital cartography)



Winged-edge structure: storage

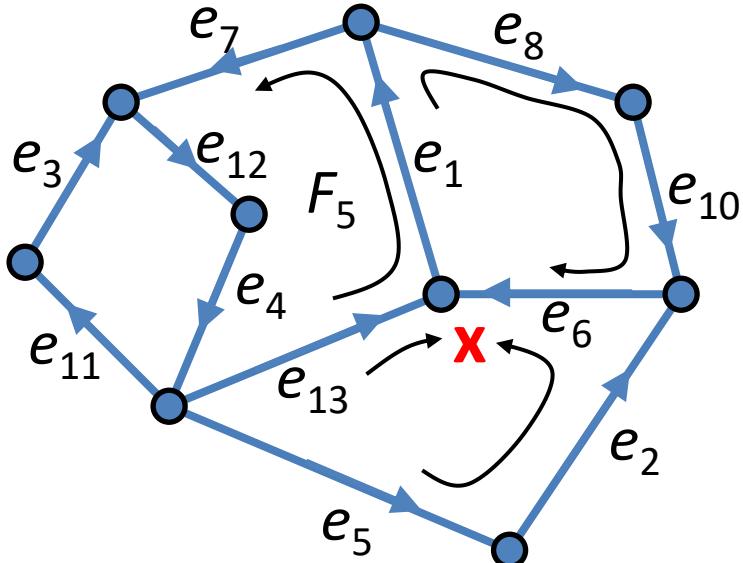
- A mesh of triangles with nv vertices has $\approx 3nv$ edges and $\approx 2nv$ triangles
- A vertex needs 3 (or 4) units of storage (with w coord)
- An edge needs 6 (or 8) units of storage: (2 vertices)
- A face needs 1 unit of storage (no matter how large!)

$$\rightarrow 3(4) + 3 \cdot 8 + 2 \cdot 1 = \mathbf{29(\text{or } 30) nv \text{ units}}$$

for each winged-edge

Winged-edge Method: Weakness

- **But** the meaningless, arbitrary edge orientation makes face boundary traversal a bit awkward:



With **consistent** orientations, we could report the vertices of a face sequentially (CCW):

```
while ( e != e_start ) {  
    e = e.lnext;  
    report e.tail.coordinates;  
}
```

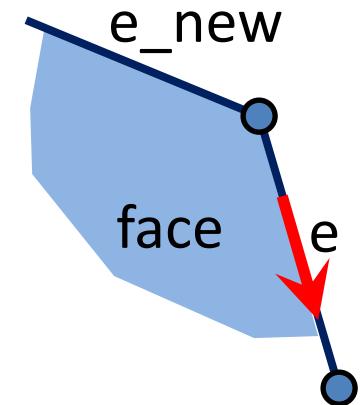
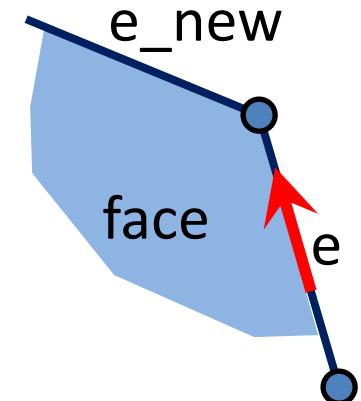
As soon as a vertex has three (or any odd number) of incident edges, no consistent orientation can exist!

Winged Edge: CCW Face Traversal

UGLY! Must traverse edges in EITHER direction...

(Forward == Boolean flag for CCW traversal of the face)

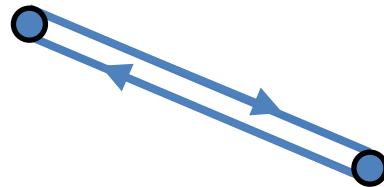
```
while ( e != e_start ) { // FIND NEXT VERTEX:  
    if (forward) {  
        report e.tail.coordinates;  
        enew = e.l_next;  
        if (enew.head == e.head) forward = false;  
    }  
    else { // backward!  
        report e.head.coordinates;  
        enew = e.r_prev;  
        if (enew.tail == e.tail) forward = true;  
    }  
    e = e_new;  
}
```



Hey! A Better Idea?

The ‘Half-Edge’ structure or ‘DCEL’

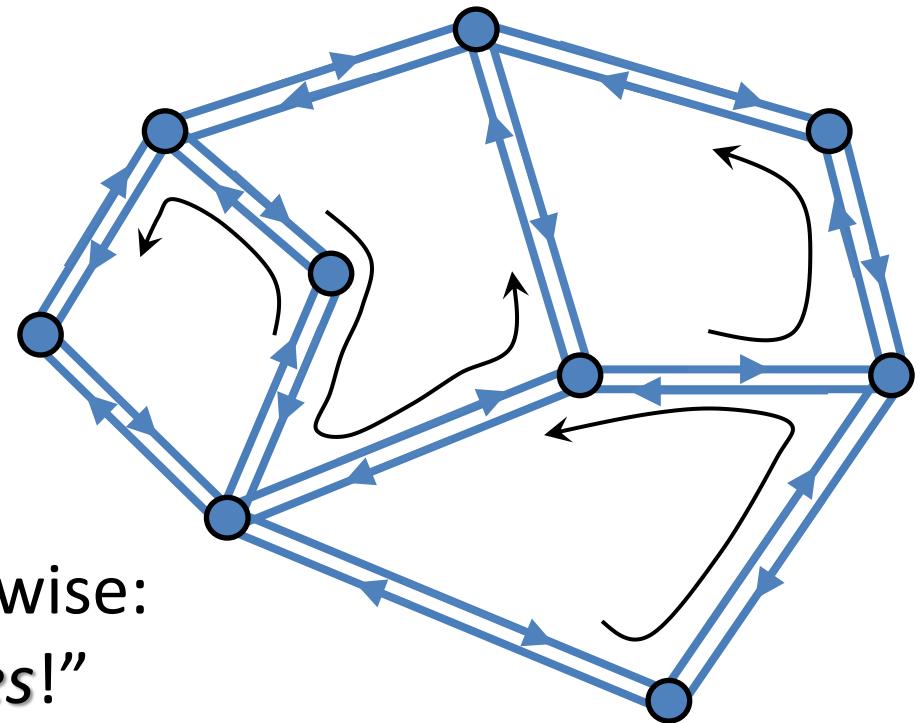
- Every edge consists of **two** half-edges (split lengthwise!) with opposite directions



- a.k.a. **Doubly-Connected Edge List**, a.k.a. “DCEL”
 - Enables purely forward traversal for face boundaries
- (see: <https://www.openmesh.org/Daily-Builds/Doc/a04080.html>
and: <https://doc.cgal.org/latest/HalfedgeDS/index.html>)

Half-edge structure

- AHA! Gives consistent orientation around every face!
- Every half-edge is incident **only** to the face at its left (by convention)
→ then every face has its half-edges oriented counterclockwise:
“the right hand; it rules!”

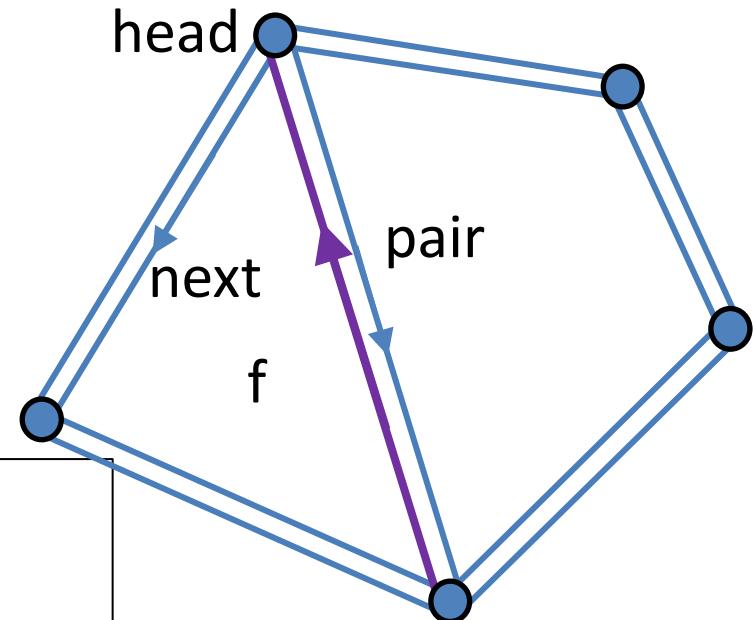


Half-edge structure

```
HEdge {  
    Hedge# next, pair;  
    Vertex# head;  
    Face# f;  
}
```

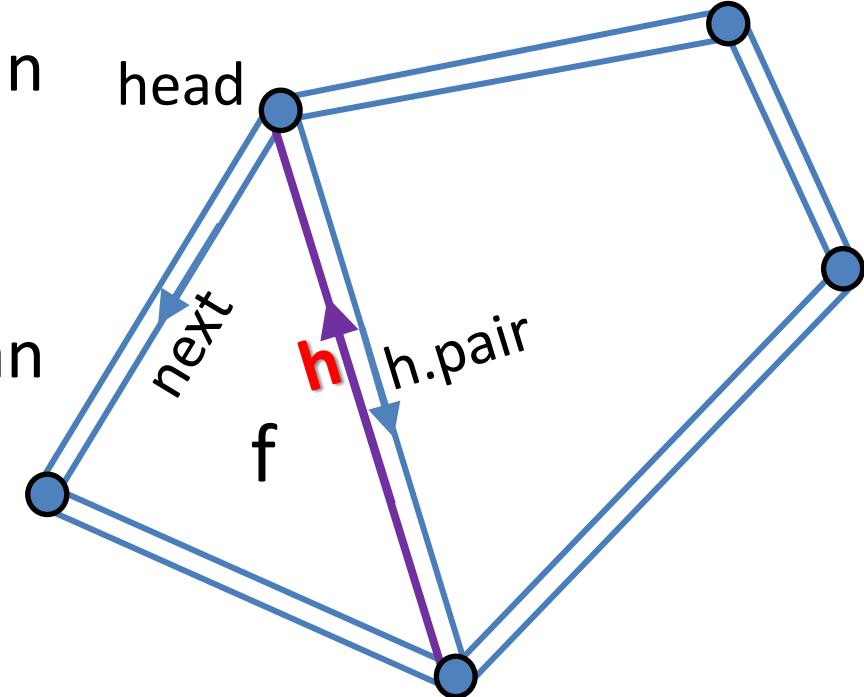
```
Vertex {  
    double x, y, z;  
    Hedge# h; // any incident half-edge  
                // pointing to this vertex  
}
```

```
Face {  
    Hedge# h; // any incident half-edge in face's boundary  
}
```



Half-edge structure

- Given a half-edge **h**, you can find its tail vertex easily:
h.pair.head
- Given a half-edge **h**, you can retrieve face **f** (as **h.f**) but also find its adjacent face easily: **h.pair.f**
- You can't find '**prev**' easily for a given half-edge **h**. (**prev** is the opposite of **next**). Some half-edge structures include a '**prev**' member in **HEdge** objects (e.g. face edges form a doubly-linked list)



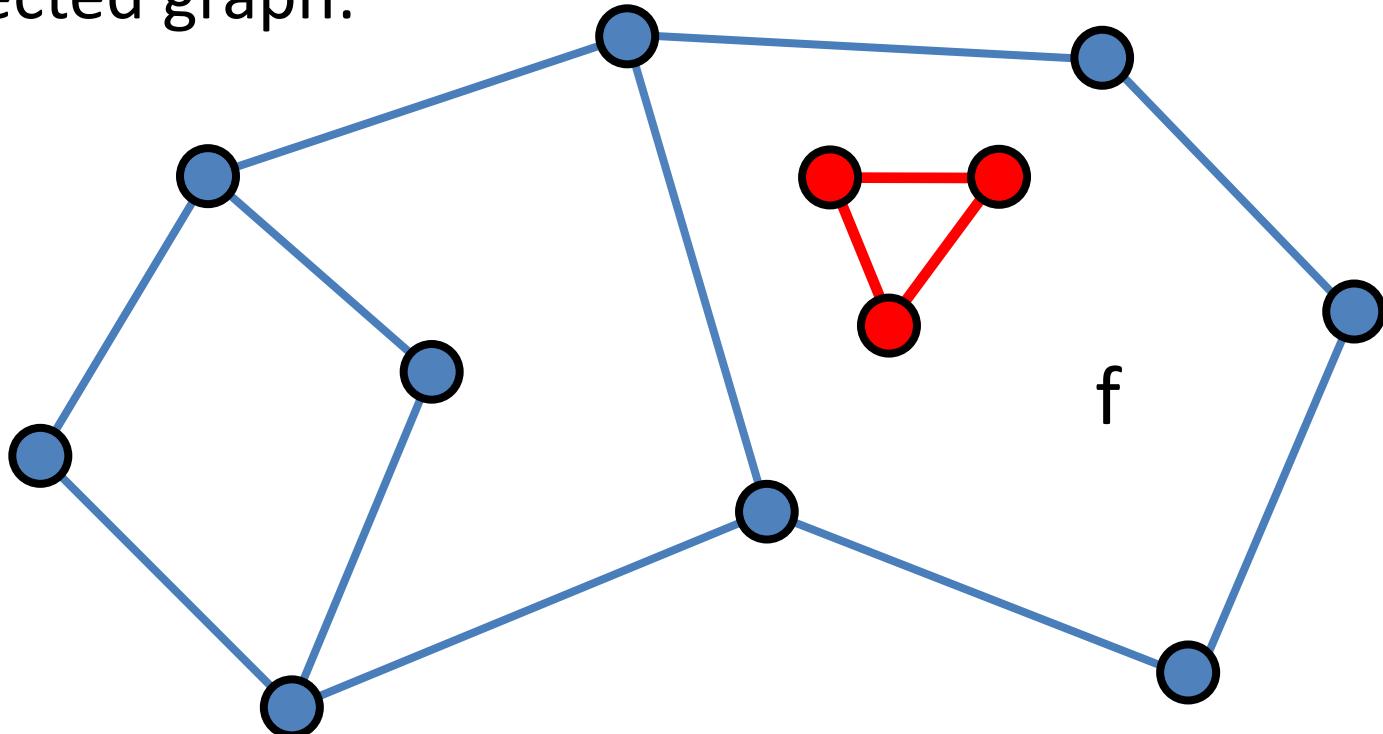
Half-edge structure: storage

- A triangular mesh with nv vertices has $\approx 3nv$ edges and $\approx 2nv$ triangles
- A vertex needs 3(or 4) units of storage (with w coord.)
- A half-edge needs 4 units of storage
- A triangle needs 1 unit of storage

$$\rightarrow 3(4) + 3 \cdot 2 \cdot 4 + 2 \cdot 1 = \mathbf{29(\text{or } 30) nv \text{ units of storage}}$$

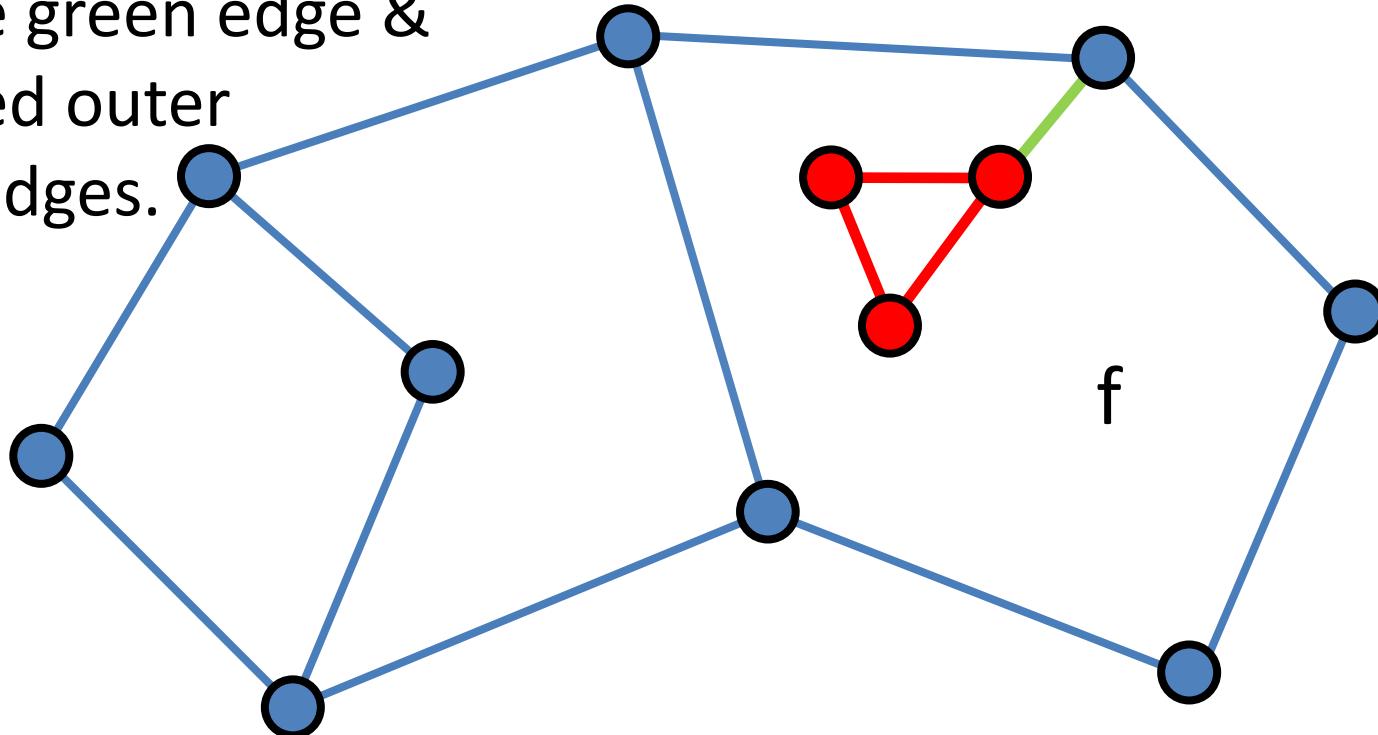
Half-edge structure: Holes?

- Planar meshes might need “islands/holes” in faces: faces from which pieces are excluded
- In this case, vertices and edges ***do not*** form a connected graph:



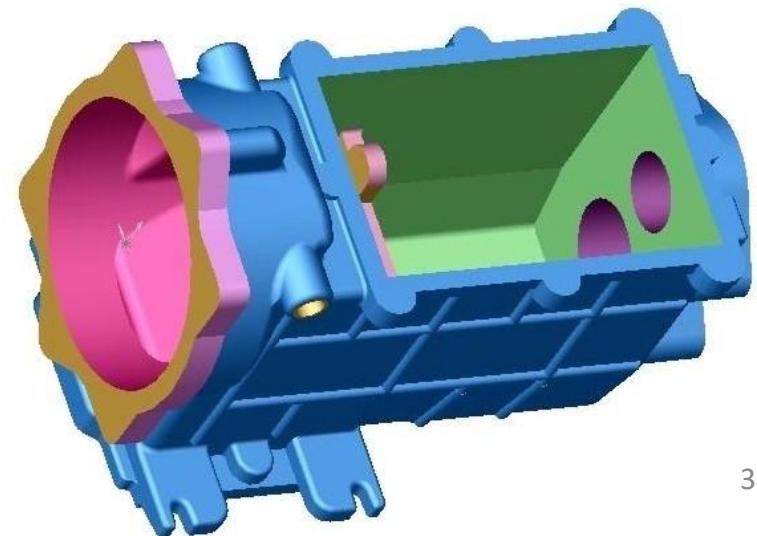
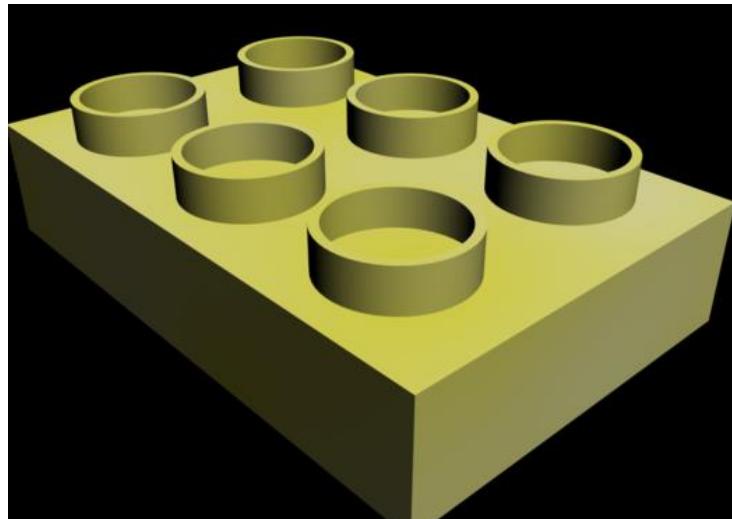
Half-edge structure: Yes -- Holes!

- SOLUTION: add an edge (a ‘hair’ or a ‘cut’ edge) to connect the ‘hole’ to the containing face.
- Now face ‘f’ (a loop of half-edges) includes both sides of the green edge & the red outer half-edges.



Half-edge structure

- Specifying faces with holes can be quite useful in 3D CAD/CAM models too...



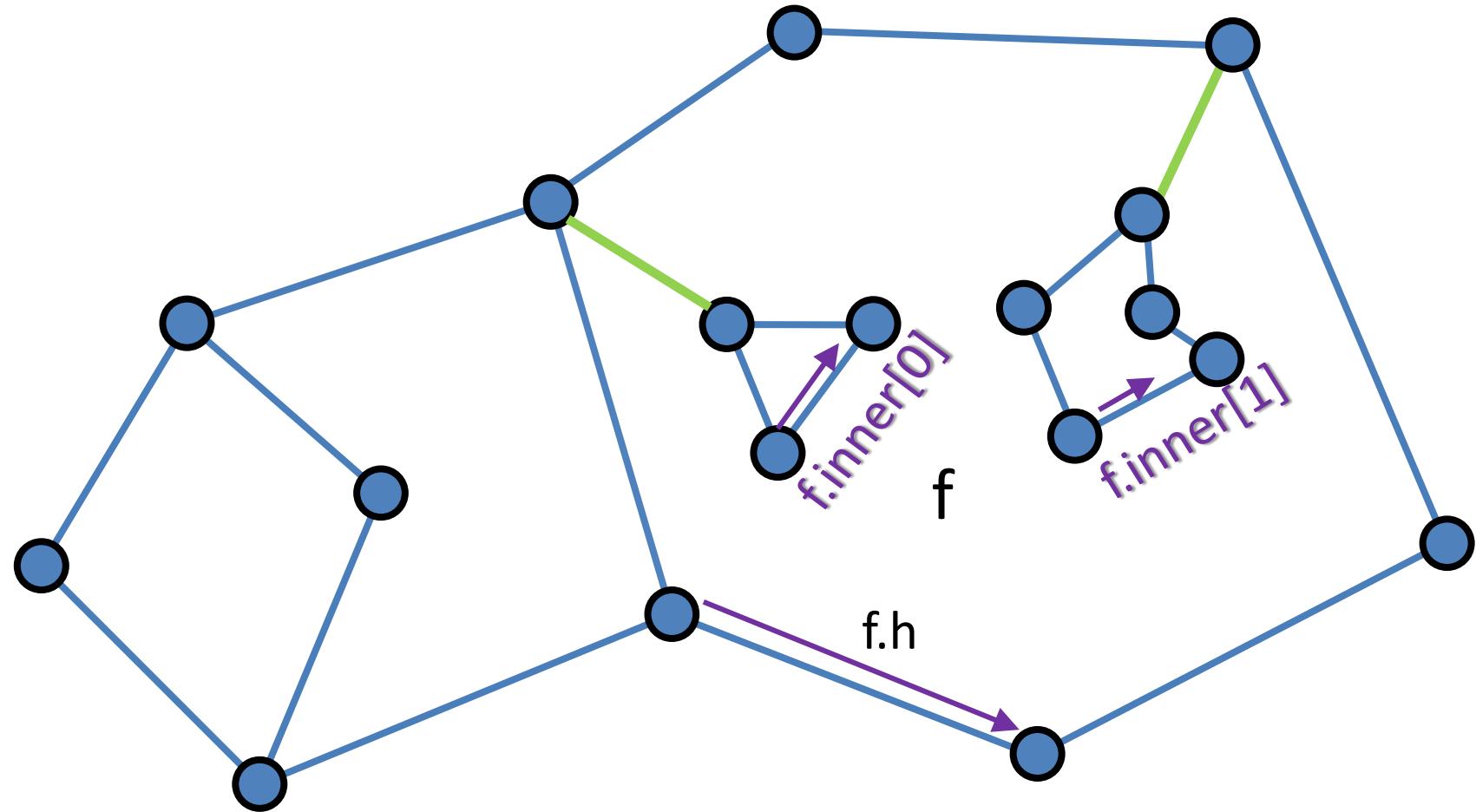
Half-edge structure that includes holes? **Easy!**

```
HEdge {  
    Hedge# next, pair;  
    Vertex# head;  
    Face# f;  
}
```

```
Vertex {  
    double x, y, z;  
    Hedge# h; // any incident half-edge  
              // pointing to this vertex  
}
```

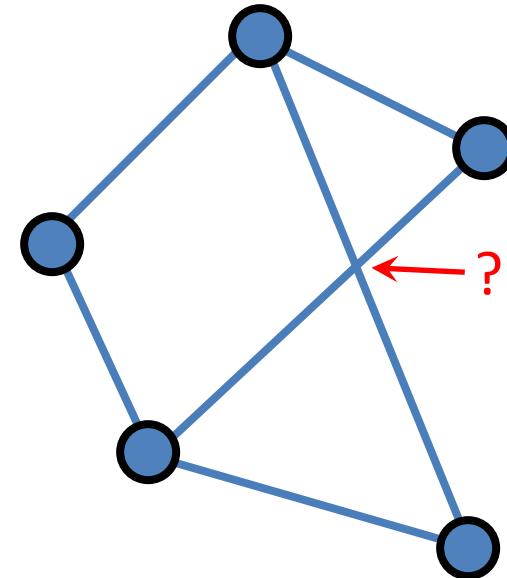
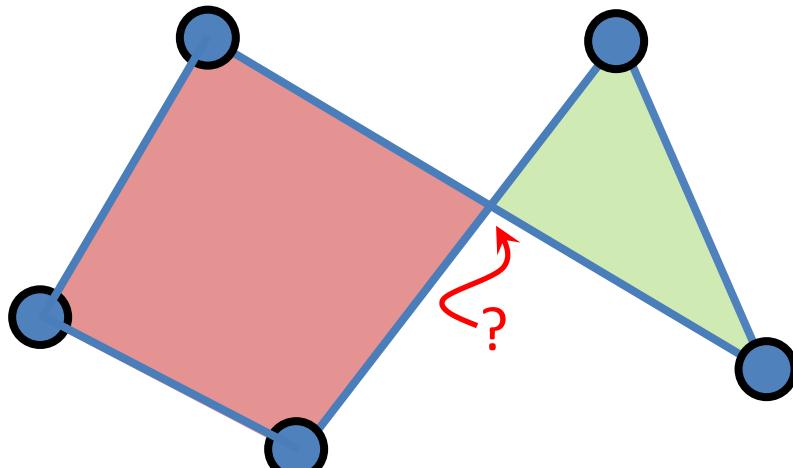
```
Face {  
    Hedge# h;          // any incident half-edge in its boundary  
    Hedge# inner[k];  // for each hole, any incident half-edge;  
                      // allows up to k holes in the face  
}
```

Half-edge structures that include holes



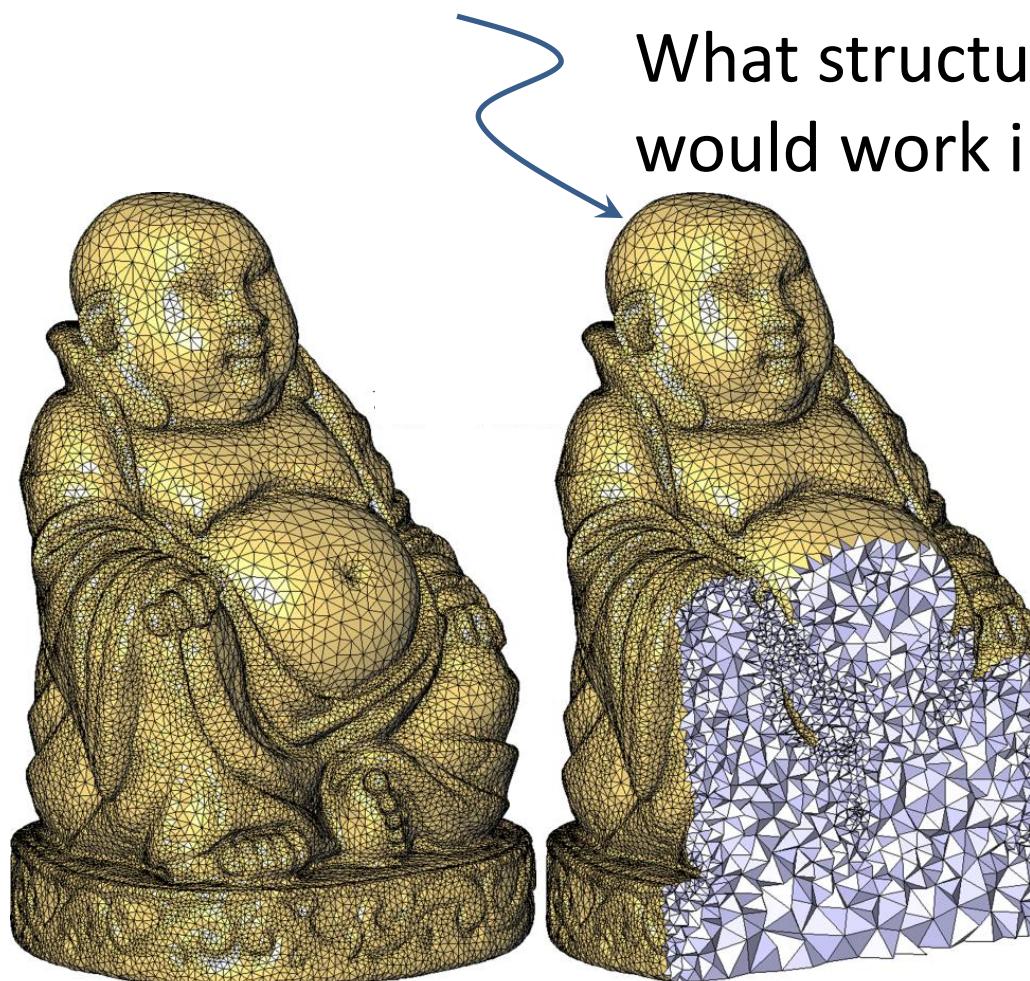
Planar versus non-planar

- **None** of the structures permit edge-crossings (or ‘edge intersections’), because they would create ill-defined faces; some non-edge 3D points could be members of multiple faces!



How would you organize 3D ‘volume’ meshes?

Our meshes define boundaries of 3D solids,
but not their 3D interiors.



END

***SOME of these SLIDES
WERE ADAPTED FROM:***

Mesh Representation, part II

based on:

Data Structures for Graphics, chapter 12 in
Fundamentals of Computer Graphics, 3rd ed.

(Shirley & Marschner)

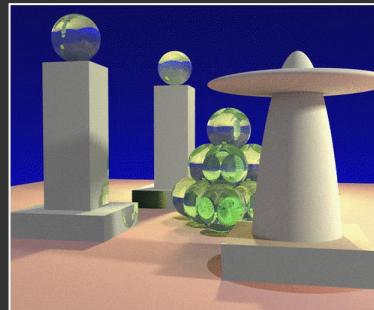
Source slides by Marc van Kreveld

**AND SOME OTHER SLIDES ADAPTED FROM:
Advanced Computer Graphics
(Spring 2013)**

CS 283, Lecture 4: Mesh Data Structures

Ravi Ramamoorthi

<http://inst.eecs.berkeley.edu/~cs283/sp13>



B-REP Mesh Geometry: Winged Edge & Half-Edge

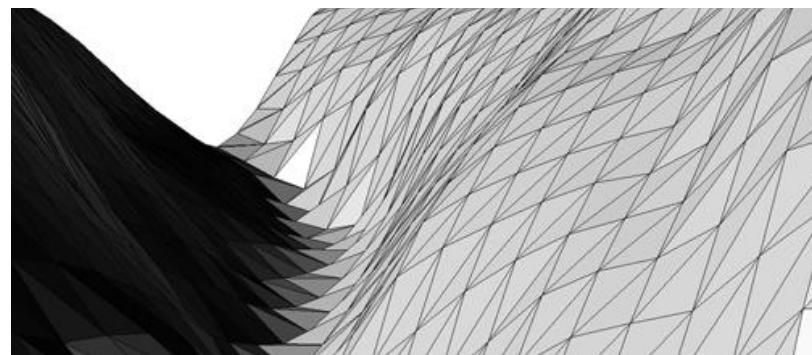
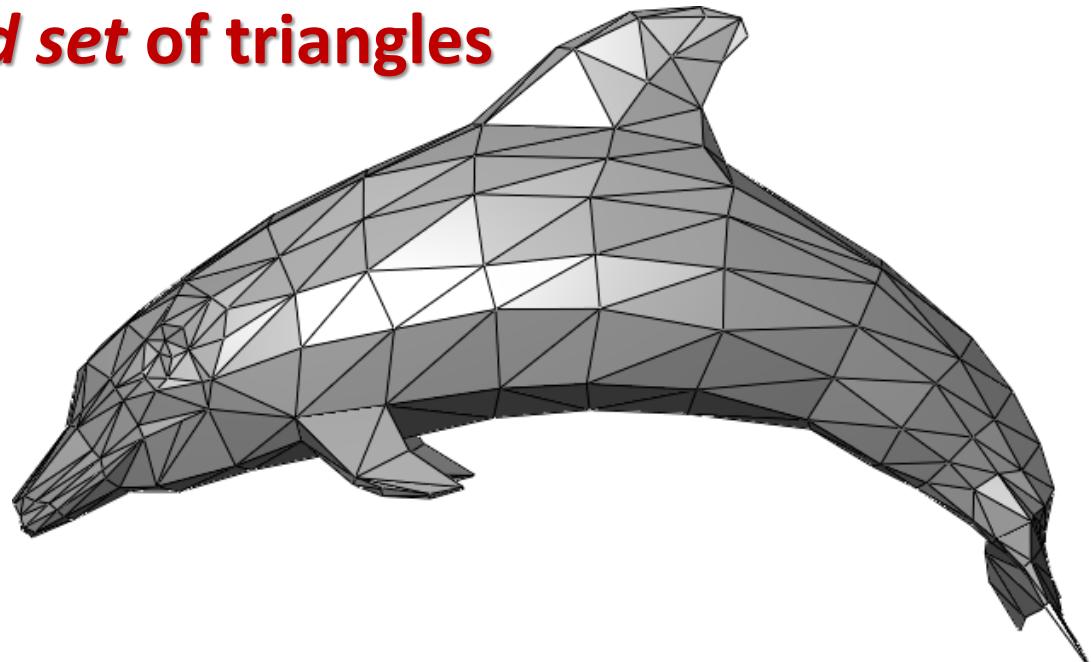
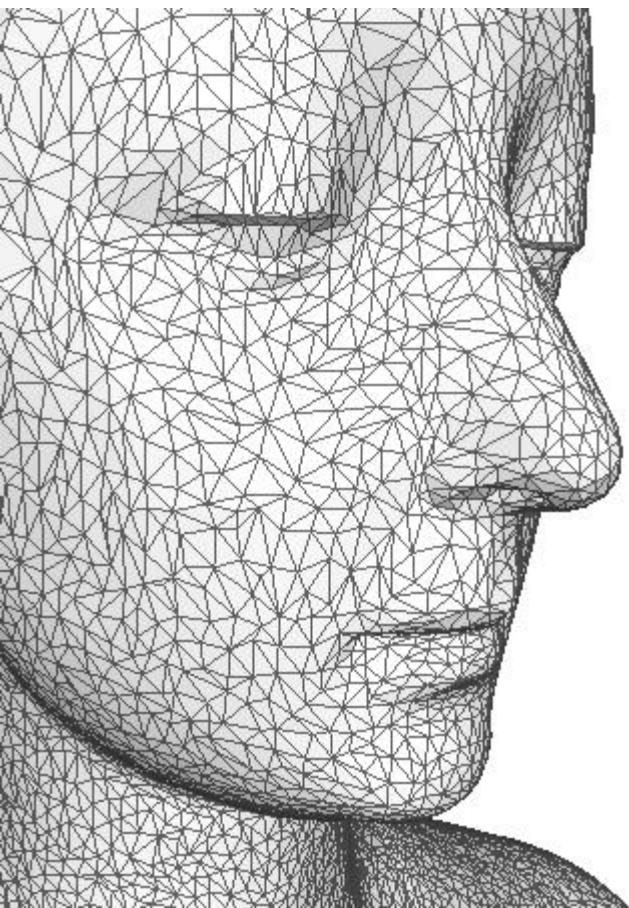
Jack Tumblin

Northwestern Univ COMP_SCI 351-1

Fall 2021

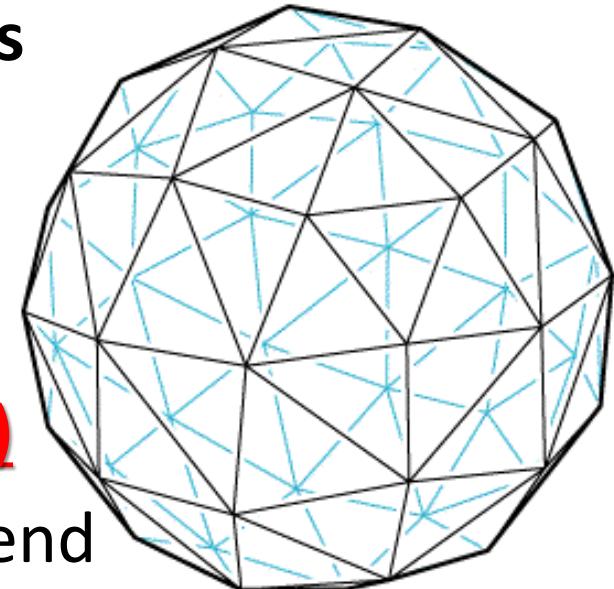
Triangle meshes

- Many freeform shapes consist of ***connected*** triangles:
a ‘mesh’ == *connected set of triangles*



3D objects as thin hollow shells

- Triangle ‘shells’ describe only a shape **boundary**
- The ‘interior’ solid defines the bounded subspace
- Triangle meshes set only *linear or planar* boundaries:
defined by **faces**, **edges**, and **vertices**
(and sometimes **nodes**:
sets of co-located vertices)



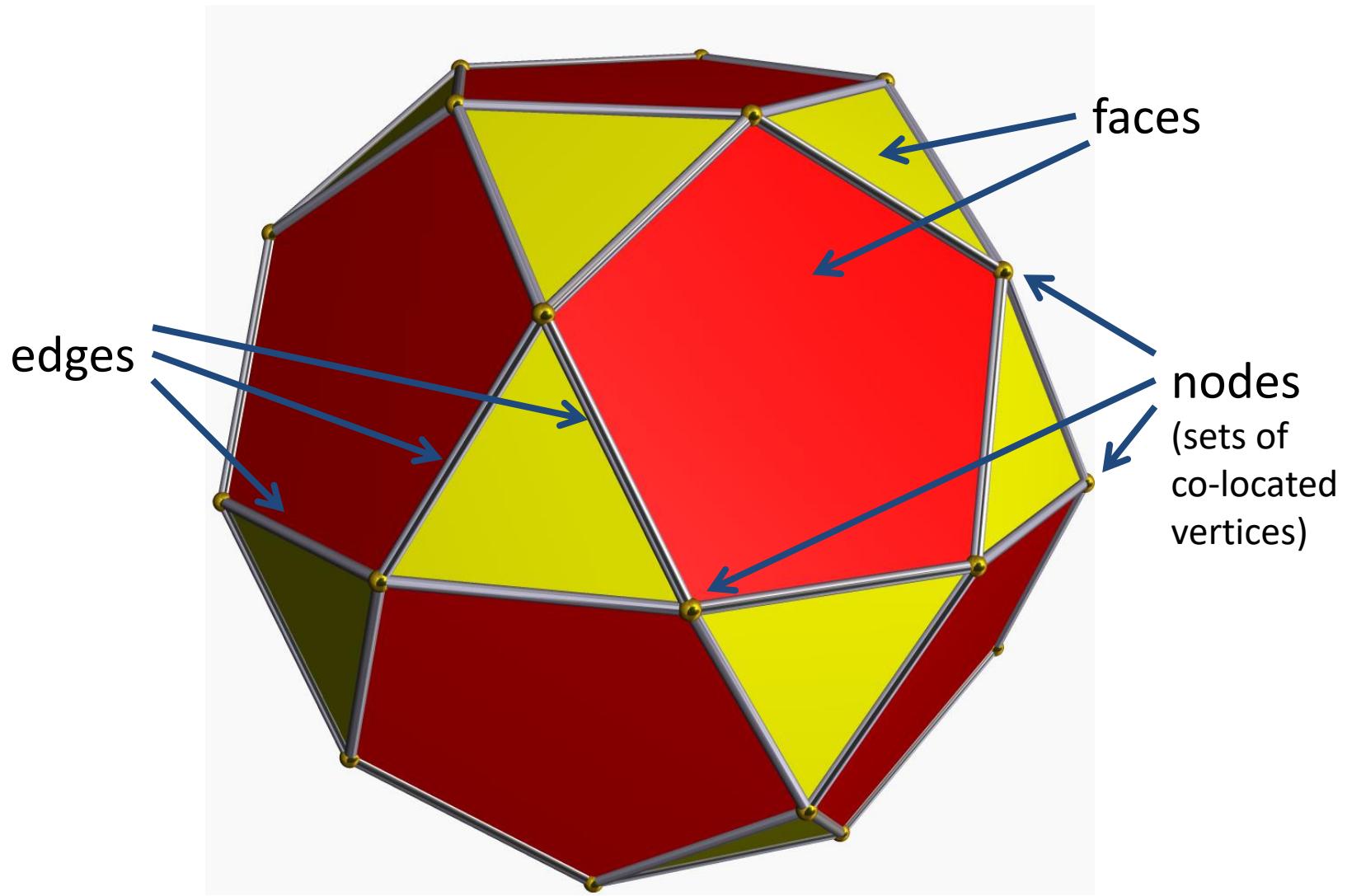
- **‘Boundary Representations’ (B-REP)**

from faces, edges and vertices can extend

‘free-form’ curved edges & surfaces too!

(https://en.wikipedia.org/wiki/Freeform_surface_modelling)

3D objects: Consider this Sphere...



Q: At any given vertex, how should we compute the 'surface normal' vector?
(\perp to just one face? Which one? \perp to the ideal sphere? How do we find it?) ⁴

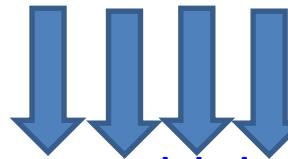
Is a ‘Triangle List’ Enough? no.

- Triangle Lists can render in parallel, independently:
- Good for fast beautiful rendering in WebGL/OpenGL
 - **GOOD**: quick to store, to draw, & to extract from shapes
 - **BAD** for shape creation, editing, geometric descriptions
 - **BAD** for shape queries: geodesics, tri-strip creation, etc.
- Lacks any “connectivity” of vertices, edges, & faces
(!Adjacency-finding may force whole-model search!!)

What ‘geometric computing’ tasks do we want to do?

Geometric Computing? What would you like to do?

Can we re-organize our vertices, nodes, edges, triangles, faces to more easily answer non-trivial questions, such as:

- what triangles share a given edge?
 - What triangles are separated by only 2 triangles?
 - What ‘faces’ have exactly 1 triangle? two? three? four?
 - What set of triangles meets at one given node (3D location)?
 - What triangles outside a face share an edge with that face?
 - What’s the ‘geodetic’ (shortest-possible curve on the surface) from a point inside one face to a point inside another face?
 - And many more!
- See: <https://www.cgal.org/> , <https://www.meshlab.net/>
- 

‘Triangle Meshes’? TL:DR;

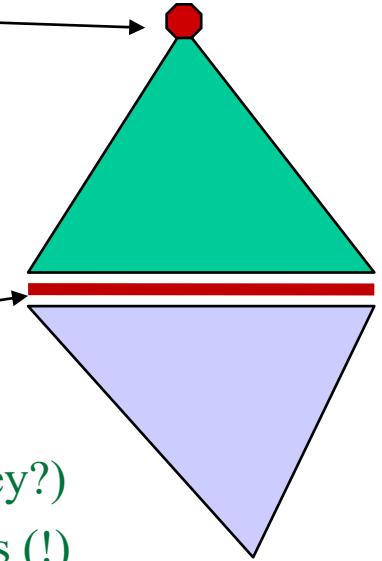
- How can we best represent/store triangle *meshes*?
- To tell **ALL** ways faces, edges, and vertices connect?
- Can it be efficient, complete, and non-redundant?

MANY possibilities; you can even invent your own...

- List of independent faces (e.g. triangles), or...
- Vertices indexed for re-use; index list for each face...
- Face- adjacency, edge-adjacency, or its variants...
- **Winged-Edge structure** (old, reliable, widely used)
- **Half-Edge structure** (more recent, simpler, deft)

Connectivity == Incidence + Adjacency

- **Incidence: (two edges tied to one vertex)**
 - What *other* vertices share this ~same 3D location?
 - How should we include their edges as ‘incident’?
 - Can we formalize the idea of a ‘node’? Of ‘nearby’ verts?
- **Adjacency: (2 vertices tied to one edge)**
 - What *other* edges share this ~same 3D location?
 - Neighboring triangles should share a common edge (do they?)
 - But are NOT strictly required to share same edge endpoints (!)



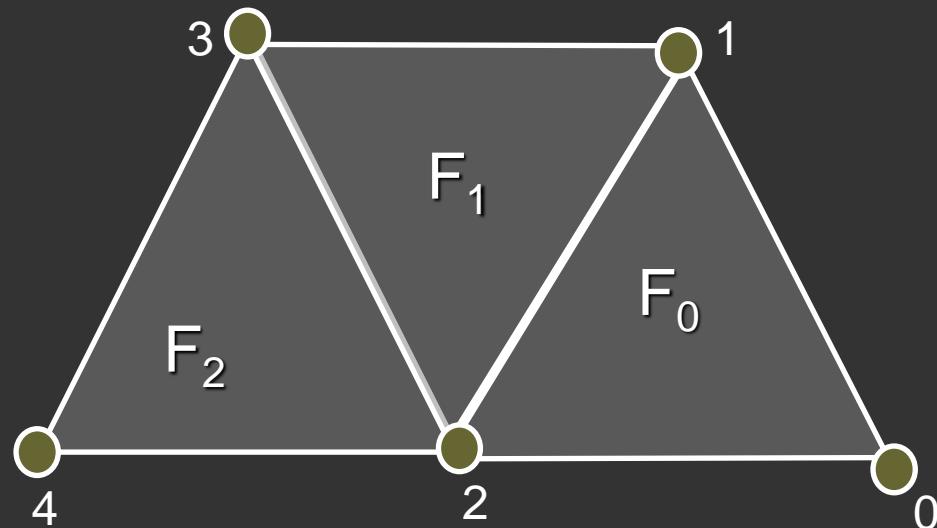
Connectivity data simplifies traversal of triangulated surfaces

- “Walk” from one triangle to an adjacent one (& visit them all just once)
- **Used to build triangle strips**
- Used to estimate surface normals at vertices or nodes
- Used to compress triangulated surfaces (helps find predictable patterns)

Independent Faces ('Vertex Lists')

Each face is a simple list of vertex coordinates;

- ++Trivial to render in WebGL/OpenGL
- -- Redundant vertices
- -- No connectivity information; slow/messy to derive it



Face Table		(==a vertex list)
F_0 :	$(x_0, y_0, z_0), (x_1, y_1, z_1), (x_2, y_2, z_2)$	
F_1 :	$(x_2, y_2, z_2), (x_1, y_1, z_1), (x_3, y_3, z_3)$	
F_2 :	$(x_2, y_2, z_2), (x_3, y_3, z_3), (x_4, y_4, z_4)$	

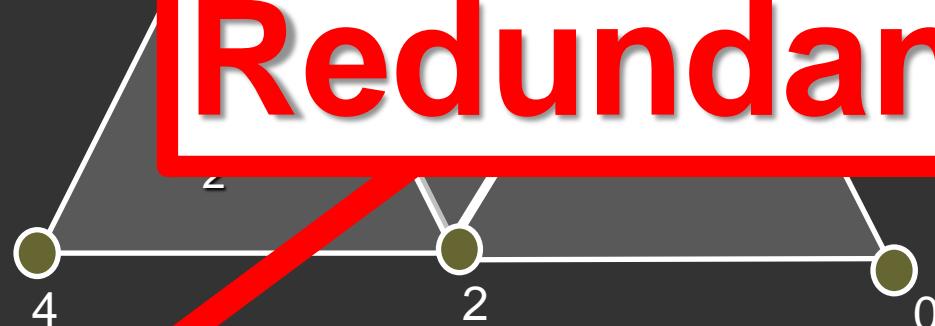
Independent Faces ('Vertex Lists')

Faces list vertex coordinates only

- Trivial to render in WebGL/OpenGL
- Trivial to convert to other formats
- Trivial to edit

VERBOSE!

**~6X vertex count,
Redundant Edges**

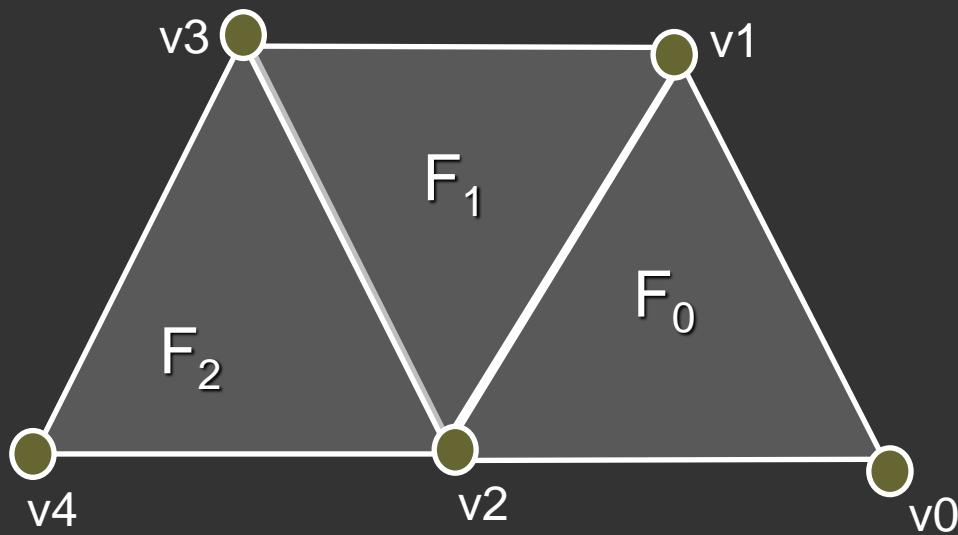


$\Gamma_2: (x_2, y_2, z_2), (x_3, y_3, z_3), (x_4, y_4, z_4)$

vertex list)
(x_2, y_2, z_2)
(x_3, y_3, z_3)
(x_4, y_4, z_4)

Indexed Face Sets ('incidence table')

- Faces as vertex# lists. “shared vertices” (nodes)
- Commonly used (e.g. ‘OBJ’ & ‘OFF’ file formats)
- easy-to-use for mesh processing (esp. if augmented)



Vertex Table		Face Table	
v_0 :	(x_0, y_0, z_0)	F_0 :	0, 1, 2
v_1 :	(x_1, y_1, z_1)	F_1 :	2, 1, 3
v_2 :	(x_2, y_2, z_2)	F_2 :	2, 3, 4
v_3 :	(x_3, y_3, z_3)		
v_4 :	(x_4, y_4, z_4)		

Note CCW ordering

A white arrow points from the text "Note CCW ordering" to the vertex index list for face F_2 in the Face Table, indicating that the vertices are listed in counter-clockwise order.

~~Indexed Face Sets ('incidence table')~~

- Faces listed by vertex # – “shared vertices”

“mmm...MMEH!”

**Redundant edges,
Awkward, slow
mesh traversal**



Note CCW ordering

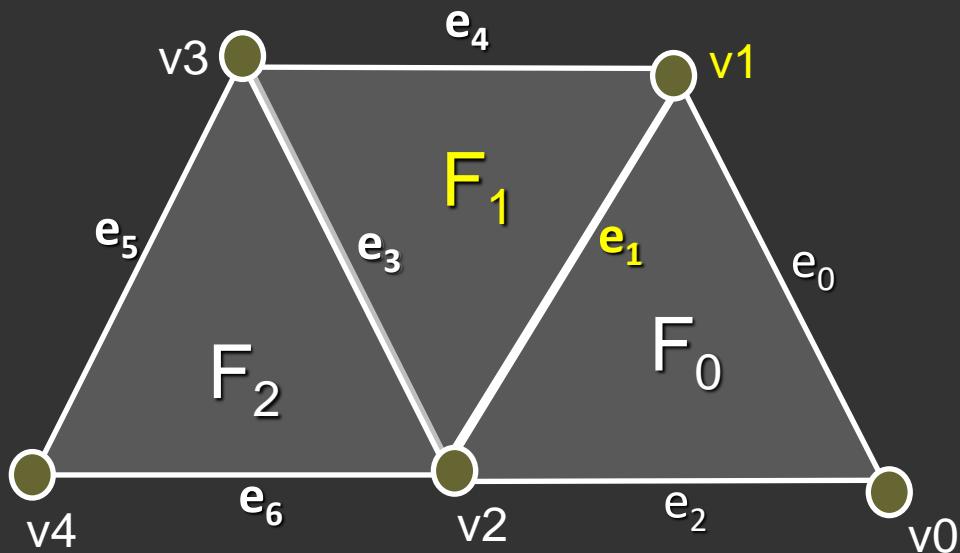
Face Table

0, 1, 2
2, 1, 3
2, 3, 4
4, 3, 0

(rotated)

Full Adjacency Lists

- Store **all** vertex, face, and edge adjacencies



Edge Adjacency Table

e ₀ :	v ₀ , v ₁	∅, F ₀	∅, e ₂ ; e ₁ , ∅;
e ₁ :	v ₁ , v ₂	F ₁ , F ₀	e ₅ , e ₀ ; e ₂ , e ₆
⋮	⋮	⋮	⋮

Face Adjacency Table

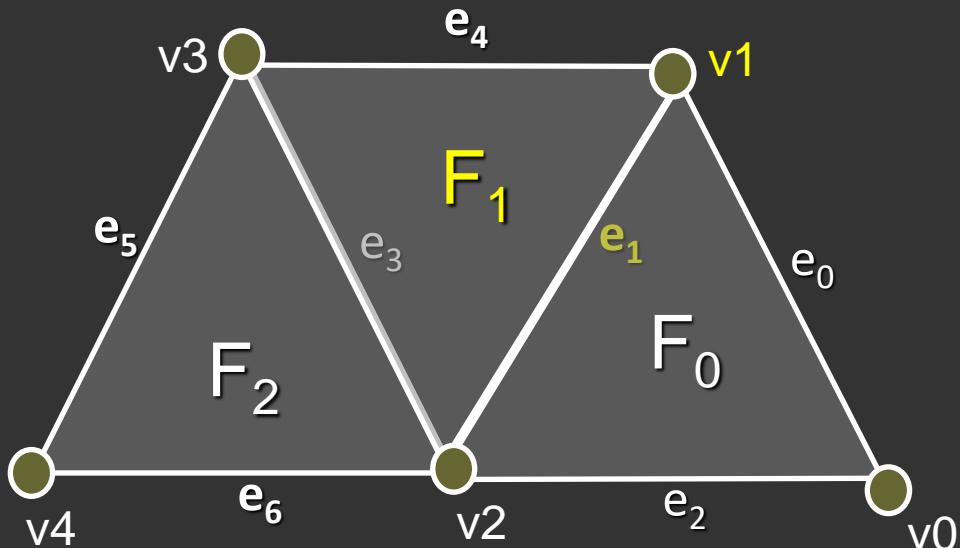
F ₀ :	v ₀ , v ₁ , v ₂	∅, F ₁ , ∅	e ₀ , e ₁ , e ₂
F ₁ :	v ₂ , v ₁ , v ₃	F ₀ , ∅, F ₂	e ₁ , e ₄ , e ₃
F ₂ :	v ₂ , v ₃ , v ₄	F ₁ , ∅, ∅	e ₃ , e ₅ , e ₆

Vertex Adjacency Table

v ₀ :	v ₁ , v ₂	F ₀ , ∅	e ₀ , e ₂
v ₁ :	v ₃ , v ₂ , v ₀ , ∅	F ₁ , F ₀ , ∅	e ₄ , e ₁ , e ₀
⋮	⋮	⋮	⋮

Full Adjacency Lists? Really?....

- **Tedious! !All** vertex, face, and edge adjacencies?!?!



Edge Adjacency Table

e_0 :	v_0, v_1	$ \emptyset, F_0$	$ \emptyset, e_2; e_1, \emptyset;$
e_1 :	v_1, v_2	$ F_1, F_0$	$ e_5, e_0; e_2, e_6$
\vdots	\vdots	\vdots	\vdots

Face Adjacency Table

F_0 :	v_0, v_1, v_2	$ \emptyset, F_1, \emptyset$	$ e_0, e_1, e_2$
F_1 :	v_2, v_1, v_3	$ F_0, \emptyset, F_2$	$ e_1, e_4, e_3$
F_2 :	v_2, v_3, v_4	$ F_1, \emptyset, \emptyset$	$ e_3, e_5, e_6$

Vertex Adjacency Table

v_0 :	v_1, v_2	$ F_0, \emptyset$	$ e_0, e_2$
v_1 :	v_3, v_2, v_0, \emptyset	$ F_1, F_0, \emptyset$	$ e_4, e_1, e_0$
\vdots	\vdots	\vdots	\vdots

Full Adjacency Lists? NO....

- Store **all** vertex, face, and

Edge Adjacency Table

**DANGEROUS
REDUNDANCY!**

Inconsistent?

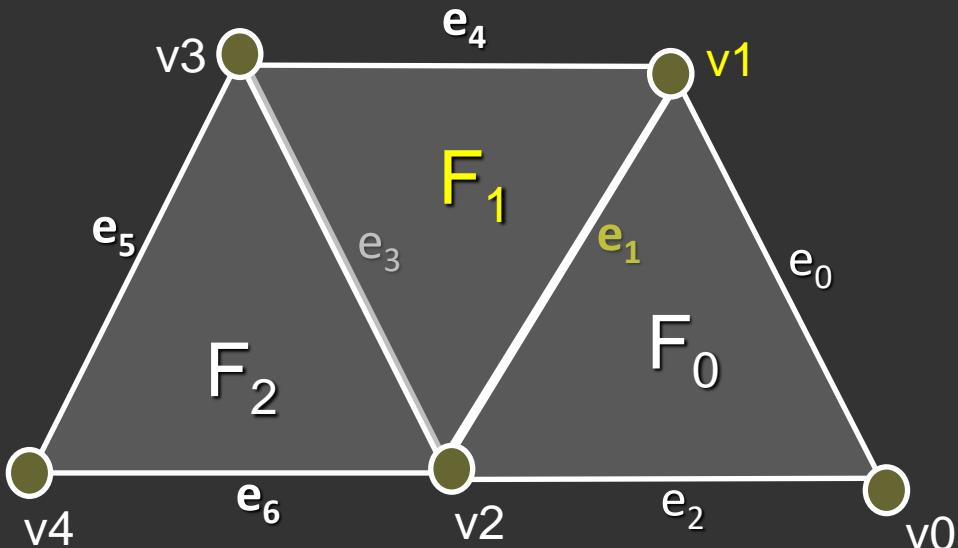
→ *undefined shape!*



$\begin{bmatrix} V_1: & [v_3, v_2, v_0, \emptyset] \\ \vdots & \vdots \end{bmatrix} \quad \begin{bmatrix} F_1: & [f_1, f_0, \emptyset] \\ \vdots & \vdots \end{bmatrix} \quad \begin{bmatrix} E_1: & [e_1, e_0] \\ \vdots & \vdots \end{bmatrix}$

Partial Adjacency Lists?

- Store *selected* adjacencies, use them to derive others
- Many possibilities, but...



Edge Adjacency Table

e ₀ :	v ₀ , v ₁	∅, F ₀	∅, e ₂ ; e ₁ , ∅;
e ₁ :	v ₁ , v ₂	F ₁ , F ₀	e ₅ , e ₀ ; e ₂ , e ₆
⋮	⋮	⋮	⋮

Face Adjacency Table

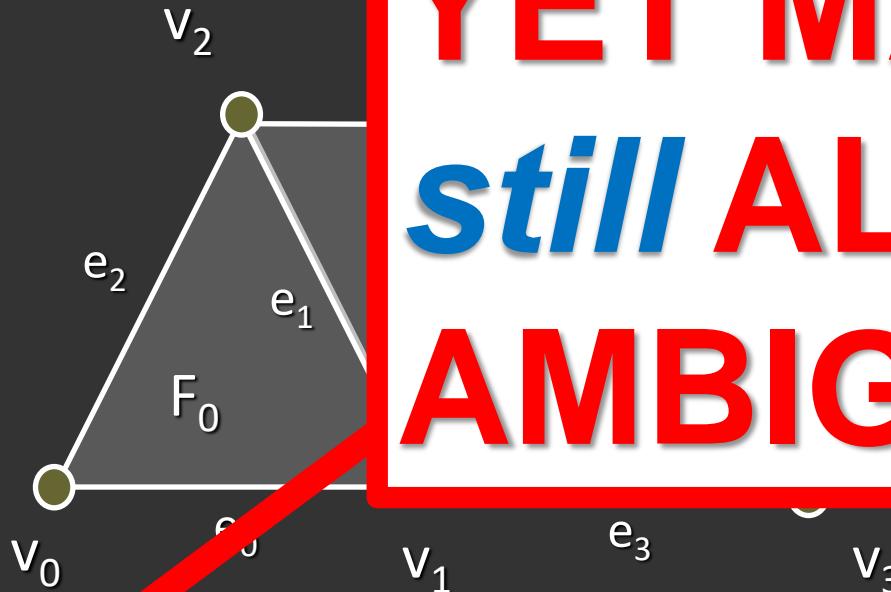
F ₀ :	v ₀ , v ₁ , v ₂	∅, F ₁ , ∅	e ₀ , e ₁ , e ₂
F ₁ :	v ₂ , v ₁ , v ₃	F ₀ , ∅, F ₂	e ₁ , e ₄ , e ₃
F ₂ :	v ₂ , v ₃ , v ₄	F ₁ , ∅, ∅	e ₃ , e ₅ , e ₆

Vertex Adjacency Table

v ₀ :	v ₁ , v ₂	F ₀ , ∅	e ₀ , e ₂
v ₁ :	v ₃ , v ₂ , v ₀ , ∅	F ₁ , F ₀ , ∅	e ₄ , e ₁ , e ₀
⋮	⋮	⋮	⋮

Partial Adjacency Lists? NO

- Store selected edges, use them
- Many possibilities



**COMPLEX,
YET MAY
*still ALLOW
AMBIGUITY!***

Adjacency Table

$\{v_0, v_2, v_1, \emptyset\}$
 $\{v_5, v_0, v_2, v_6\}$

Adjacency Table

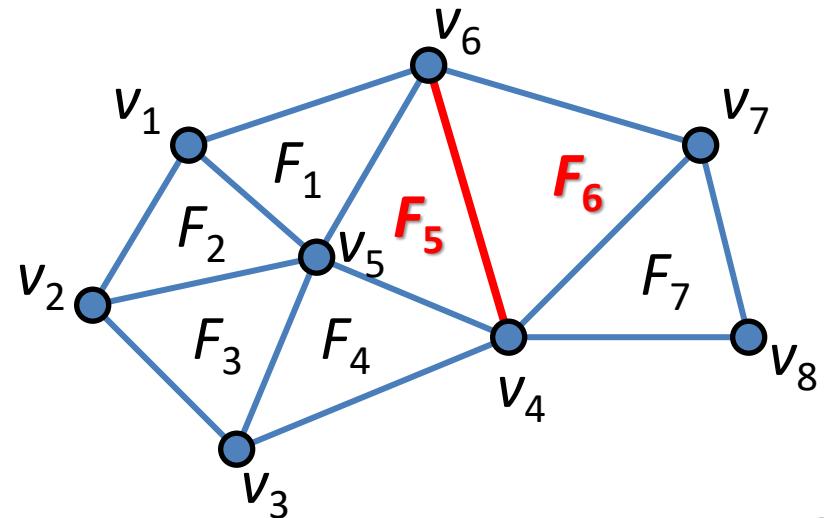
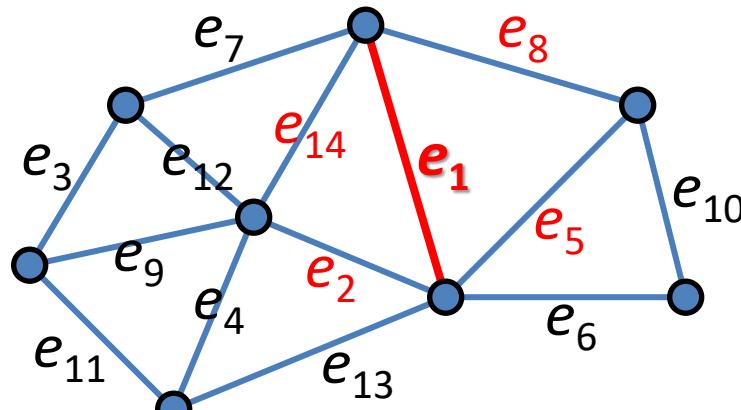
$\emptyset; e_0, e_2, e_0$
 $F_2; e_6, e_1, e_5$
 $\emptyset; e_4, e_5, e_3$

Adjacency Table

$v_0: v_1, v_2; F_0; e_0, e_2$
 $v_1: v_3, v_4, v_2, v_0; F_2, F_1, F_0, \dots, e_5, e_1, e_0$
 \vdots

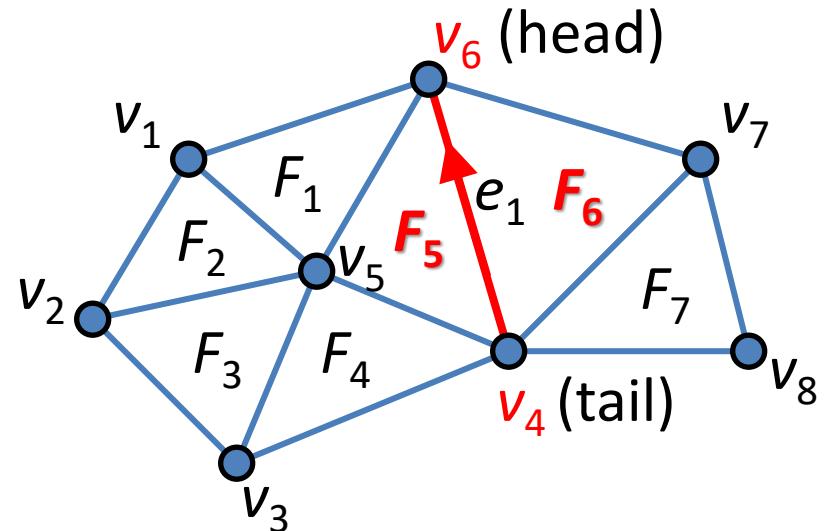
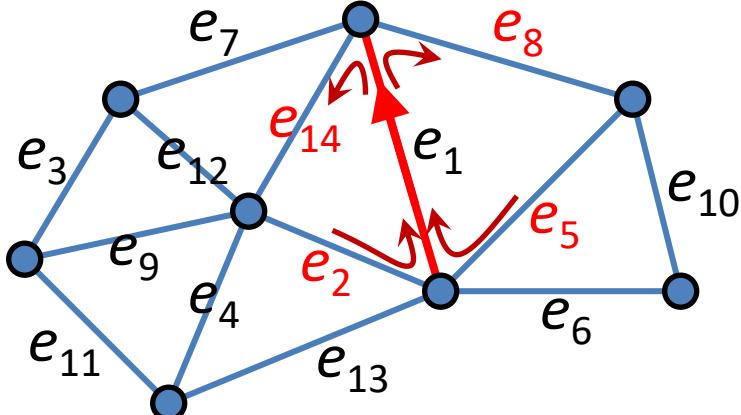
Solution 1: Winged-edge structure

- Stores **connectivity at edges** instead of at vertices
- **For each edge**, such as e_1 , store:
 - Two vertices define the edge: v_4 and v_6
 - Two faces (triangles) adjacent to the edge: F_5 and F_6
 - Four edges to identify those two faces: e_2, e_{14}, e_5 , and e_8



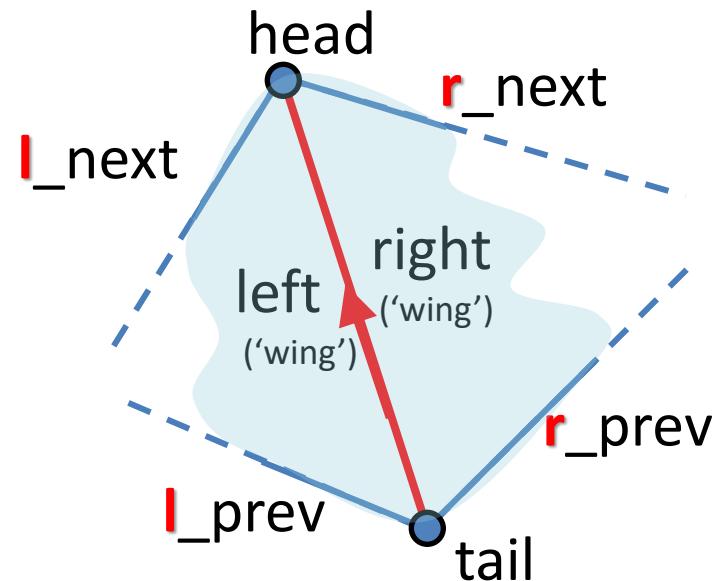
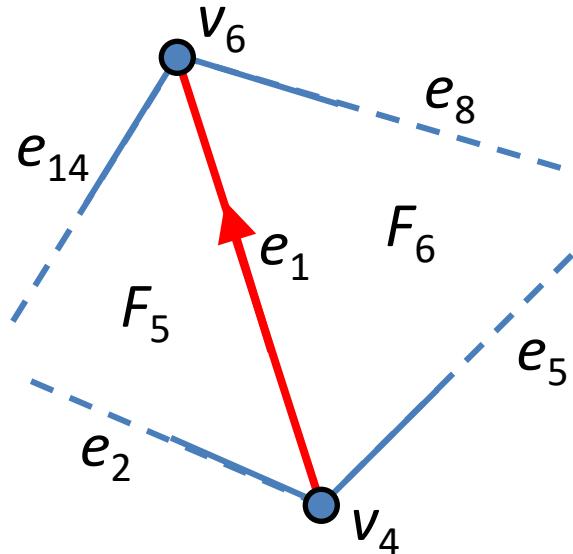
Solution 1: Winged-edge structure

- Give e_1 a (frequently arbitrary(!)) direction, then
 - v_4 is the tail and v_6 is the head
 - F_5 is to the left and F_6 is to the right
 - e_2 is previous on the left side, e_{14} is next on the left side, e_5 is previous on the right side, and e_8 is next on the right side



Solution 1: Winged-edge structure

- Give e_1 a direction, then view it from above the surface:
 - v_4 is the **tail** and v_6 is the **head**.
 - Face F_5 is the **left** ‘wing’, and face F_6 is the **right** ‘wing’
 - e_2 is **previous on the left** side, e_{14} is **next on the left** side, e_5 is **previous on the right** side, and e_8 is **next on the right** side



Solution 1: Winged-edge scheme

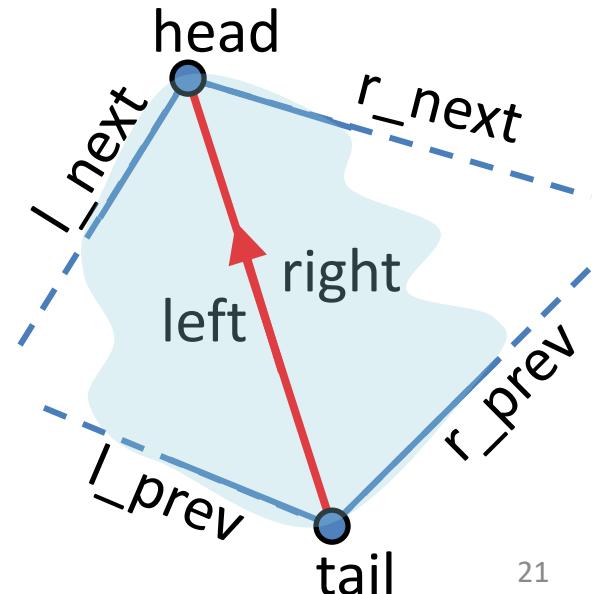
```
Edge { // Each edge holds “wings” indices:  
    Edge#     l_prev, l_next, r_prev, r_next;  
    Vertex#   head, tail;  
    Triangle# left, right;  
}
```

```
Vertex {  
    double x, y, z;  
    Edge# e; // any incident edge(just 1)  
} // (easy to trace out all the rest of them)
```

```
Triangle {  
    Edge# e; // any incident edge(just 1!)  
} // (easy to trace out all the rest of them)
```

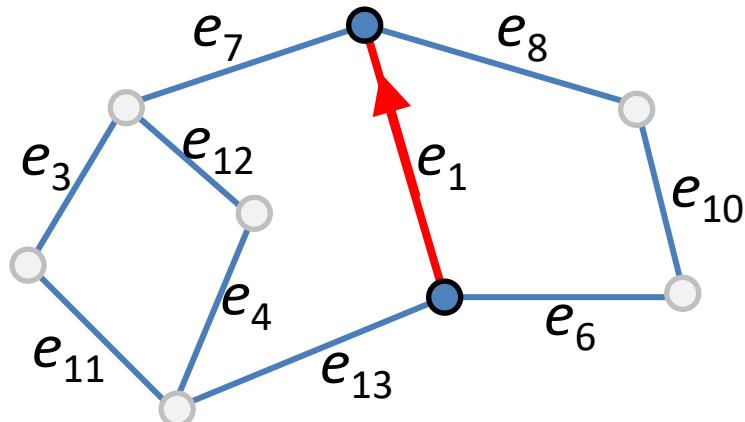
Winged-Edge B-REP:
--Array of Edge objects
--Array of Vertex objects
--Array of Triangle objects

Fixed-size data structs!
easy to store,
index and search.--



Solution 1: Winged-edge structure

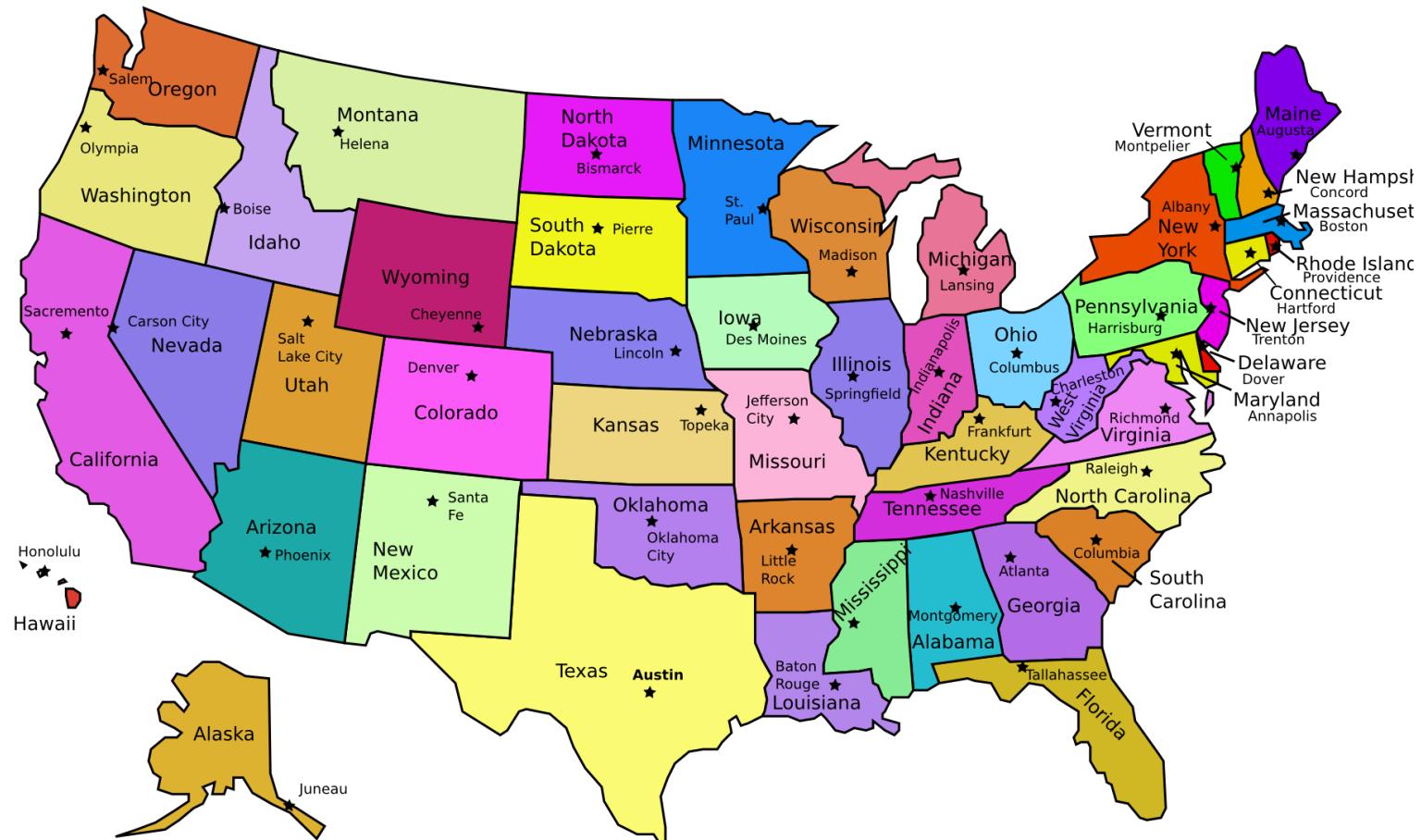
- Also good for meshes with **arbitrary polygonal faces**:
 - Still just one head and one tail per edge;
 - Still just one left face and one right face per edge;
 - Still just one previous edge and just one next edge for each of the two faces (left, right):
EXAMPLE: e_{13}, e_7 for left face, and e_6, e_8 for right face.



Note: The naïve triangle-list structure of VBOs alone lacks connectivity, & is tough to edit interactively.
Winged Edge is better!

Solution 1: Winged-edge structure

- Also good for planar partitions such as land maps
(widely used in digital cartography)



Winged-edge structure: storage

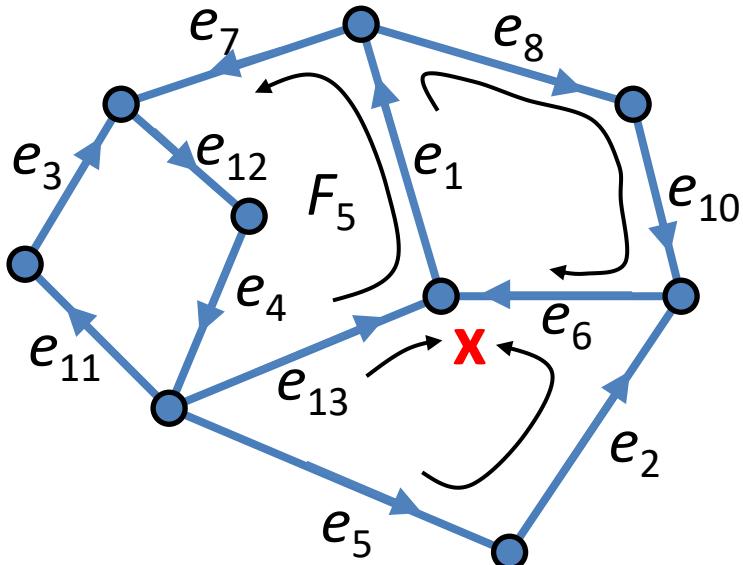
- A mesh of triangles with nv vertices has $\approx 3nv$ edges and $\approx 2nv$ triangles
- A vertex needs 3 (or 4) units of storage (with w coord)
- An edge needs 6 (or 8) units of storage: (2 vertices)
- A face needs 1 unit of storage (no matter how large!)

$$\rightarrow 3(4) + 3 \cdot 8 + 2 \cdot 1 = \mathbf{29(\text{or } 30) nv \text{ units}}$$

for each winged-edge

Winged-edge Method: Weakness

- **But** the meaningless, arbitrary edge orientation makes face boundary traversal a bit awkward:



With **consistent** orientations, we could report the vertices of a face sequentially (CCW):

```
while ( e != e_start ) {  
    e = e.lnext;  
    report e.tail.coordinates;  
}
```

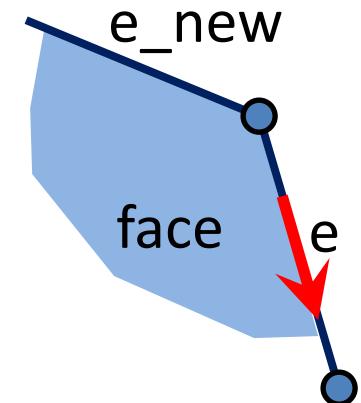
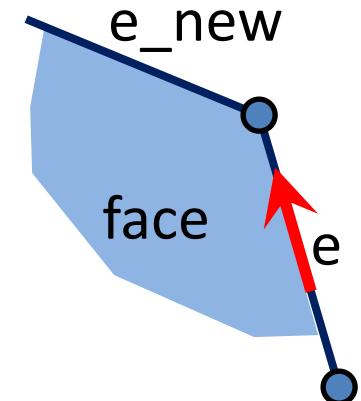
As soon as a vertex has three (or any odd number) of incident edges, no consistent orientation can exist!

Winged Edge: CCW Face Traversal

UGLY! Must traverse edges in EITHER direction...

(Forward == Boolean flag for CCW traversal of the face)

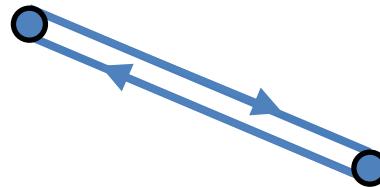
```
while ( e != e_start ) { // FIND NEXT VERTEX:  
    if (forward) {  
        report e.tail.coordinates;  
        enew = e.l_next;  
        if (enew.head == e.head) forward = false;  
    }  
    else { // backward!  
        report e.head.coordinates;  
        enew = e.r_prev;  
        if (enew.tail == e.tail) forward = true;  
    }  
    e = e_new;  
}
```



Hey! A Better Idea?

The ‘Half-Edge’ structure or ‘DCEL’

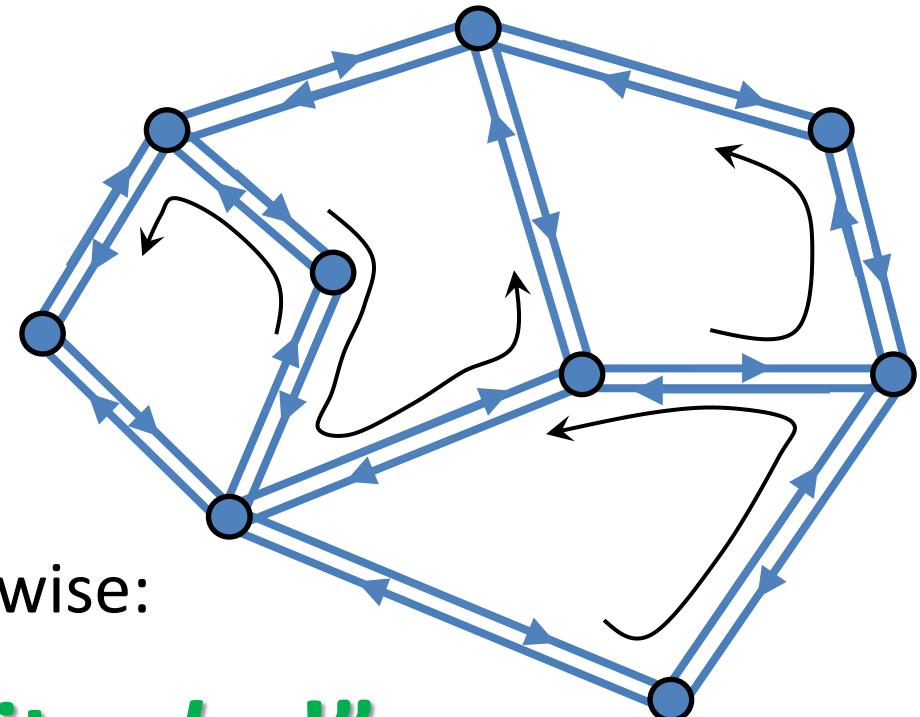
- Every edge consists of **two** half-edges
(split lengthwise!) with opposite directions



- a.k.a. **Doubly-Connected Edge List**, a.k.a. “**DCEL**”
 - Enables purely forward traversal for face boundaries
- (see: <https://www.openmesh.org/Daily-Builds/Doc/a04080.html>
and: <https://doc.cgal.org/latest/HalfedgeDS/index.html>)

Half-edge structure

- AHA! Gives consistent orientation around every face!
- Every half-edge is incident **only** to the face at its left (by convention)
→ then every face has its half-edges oriented counterclockwise:



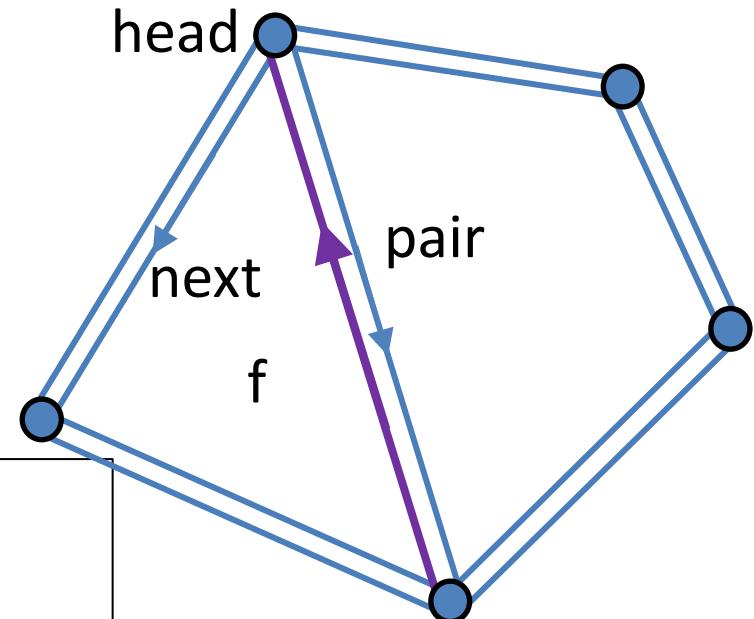
“the right hand; it rules!”

Half-edge structure

```
HEdge {  
    Hedge# next, pair;  
    Vertex# head;  
    Face# f;  
}
```

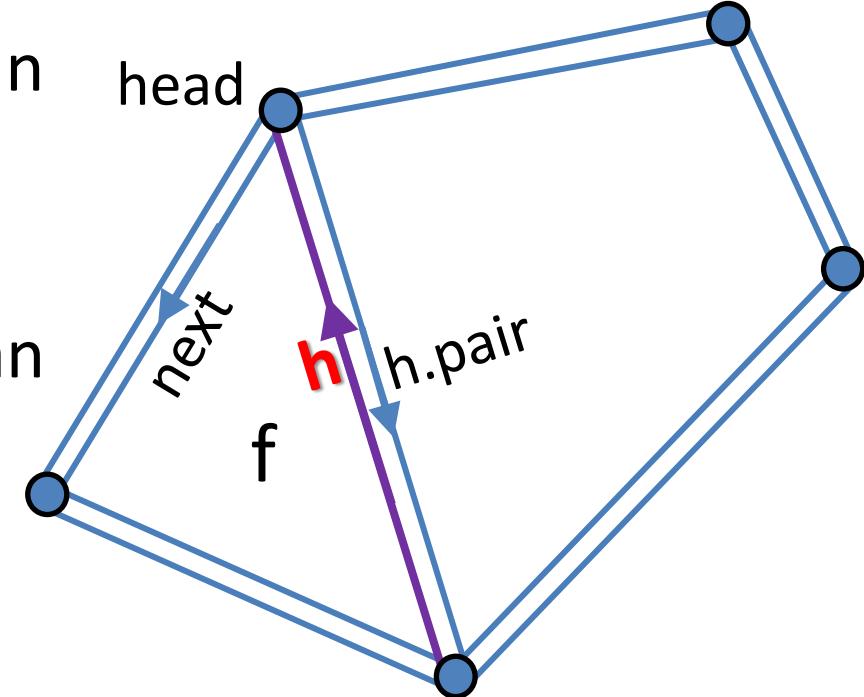
```
Vertex {  
    double x, y, z;  
    Hedge# h; // any incident half-edge  
                // pointing to this vertex  
}
```

```
Face {  
    Hedge# h; // any incident half-edge in face's boundary  
}
```



Half-edge structure

- Given a half-edge **h**, you can find its tail vertex easily:
h.pair.head
- Given a half-edge **h**, you can retrieve face **f** (as **h.f**) but also find its adjacent face easily: **h.pair.f**
- You can't find '**prev**' easily for a given half-edge **h**. (**prev** is the opposite of **next**). Some half-edge structures include a '**prev**' member in **HEdge** objects (e.g. face edges form a doubly-linked list)



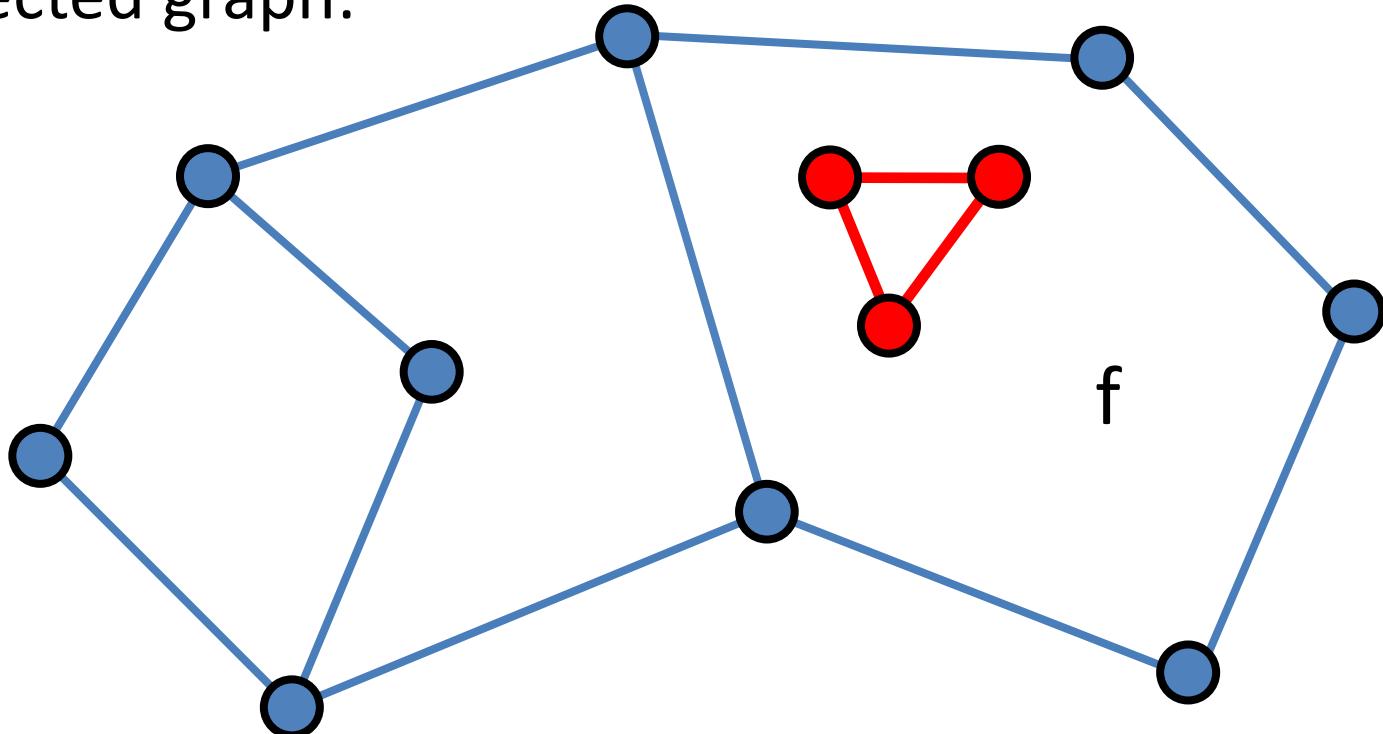
Half-edge structure: storage

- A triangular mesh with nv vertices has $\approx 3nv$ edges and $\approx 2nv$ triangles
- A vertex needs 3(or 4) units of storage (with w coord.)
- A half-edge needs 4 units of storage
- A triangle needs 1 unit of storage

$$\rightarrow 3(4) + 3 \cdot 2 \cdot 4 + 2 \cdot 1 = \mathbf{29(\text{or } 30) nv \text{ units of storage}}$$

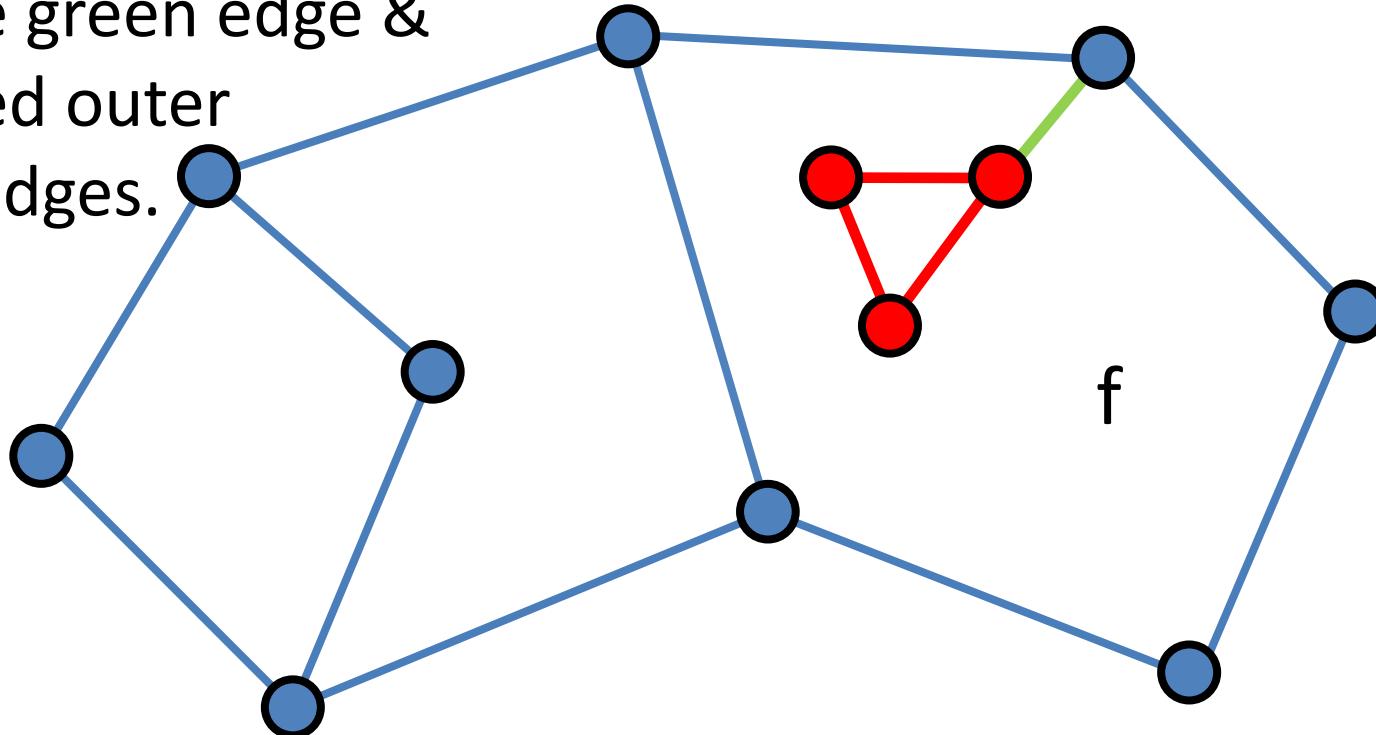
Half-edge structure: Holes?

- Planar meshes might need “islands/holes” in faces: faces from which pieces are excluded
- In this case, vertices and edges ***do not*** form a connected graph:



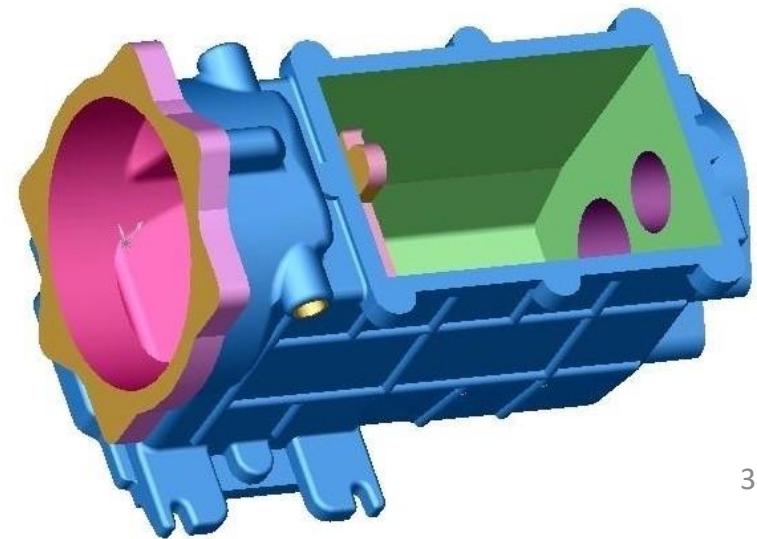
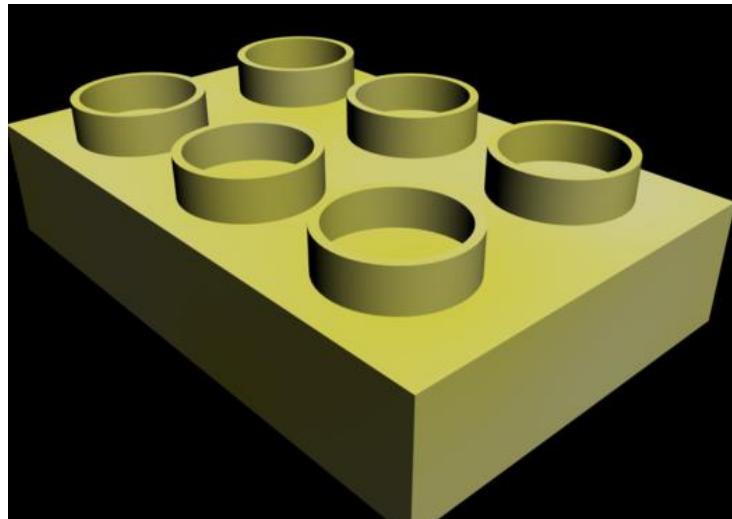
Half-edge structure: Yes -- Holes!

- SOLUTION: add an edge (a ‘hair’ or a ‘cut’ edge) to connect the ‘hole’ to the containing face.
- Now face ‘f’ (a loop of half-edges) includes both sides of the green edge & the red outer half-edges.



Half-edge structure

- Specifying faces with holes can be quite useful in 3D CAD/CAM models too...



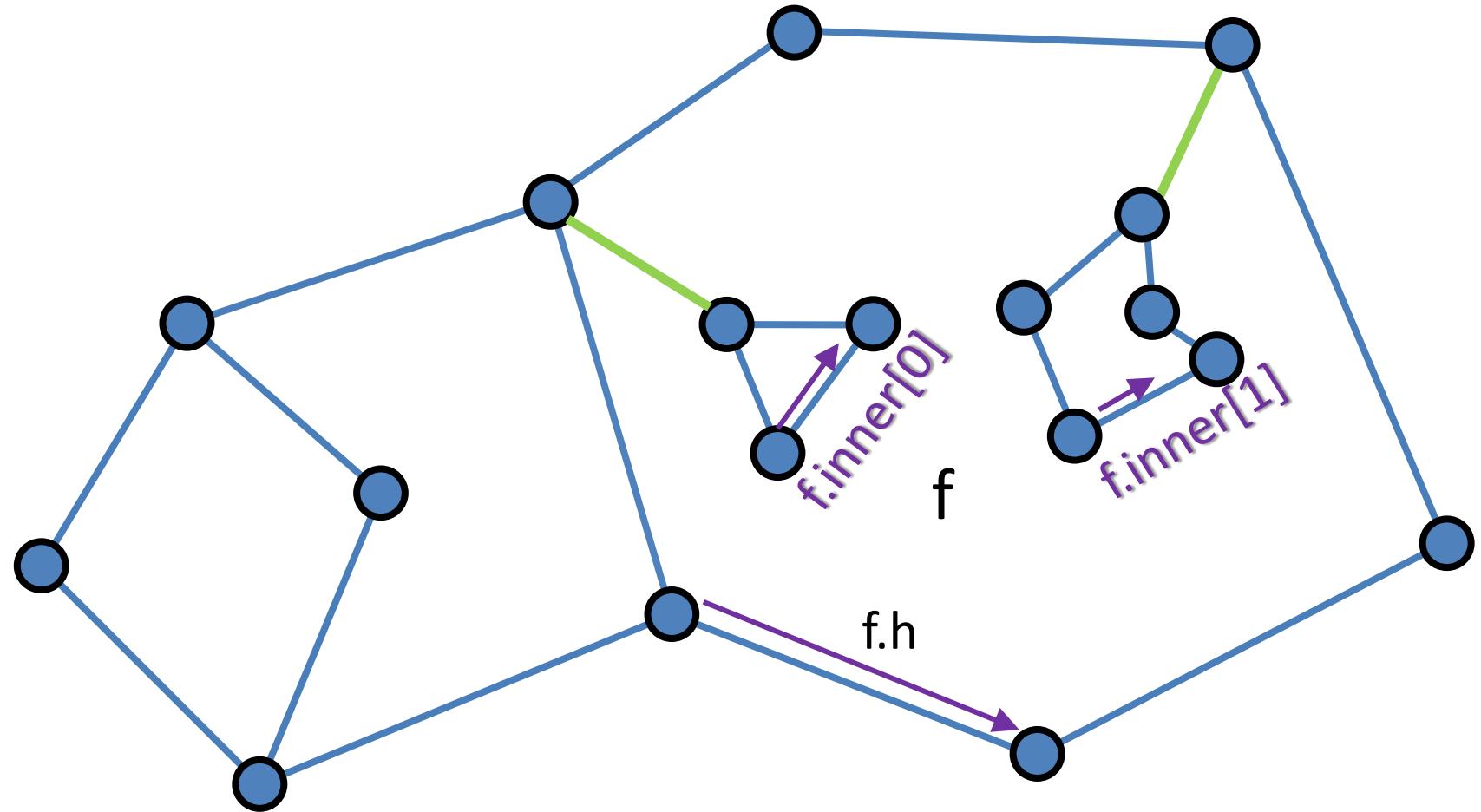
Half-edge structure that includes holes? **Easy!**

```
HEdge {  
    Hedge# next, pair;  
    Vertex# head;  
    Face# f;  
}
```

```
Vertex {  
    double x, y, z;  
    Hedge# h; // any incident half-edge  
              // pointing to this vertex  
}
```

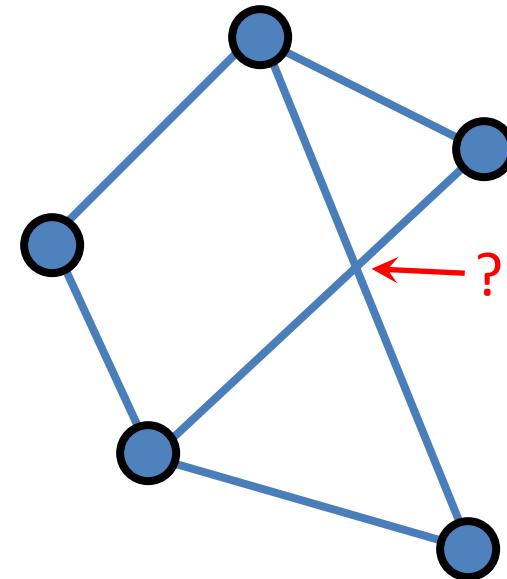
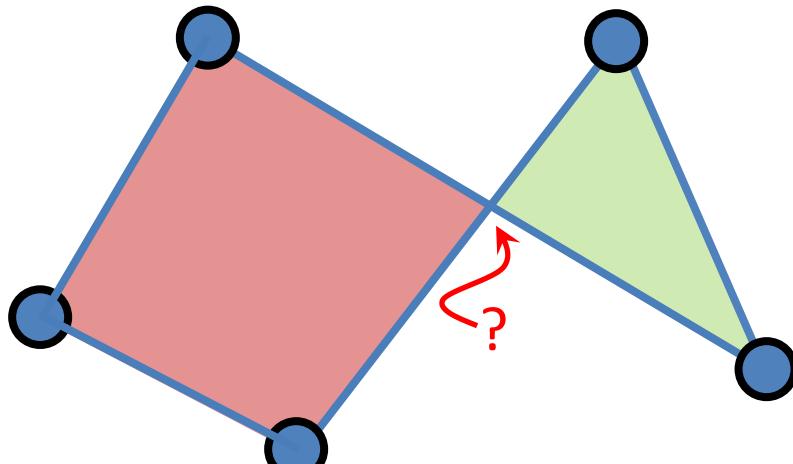
```
Face {  
    Hedge# h;          // any incident half-edge in its boundary  
    Hedge# inner[k];  // for each hole, any incident half-edge;  
                      // allows up to k holes in the face  
}
```

Half-edge structures that include holes



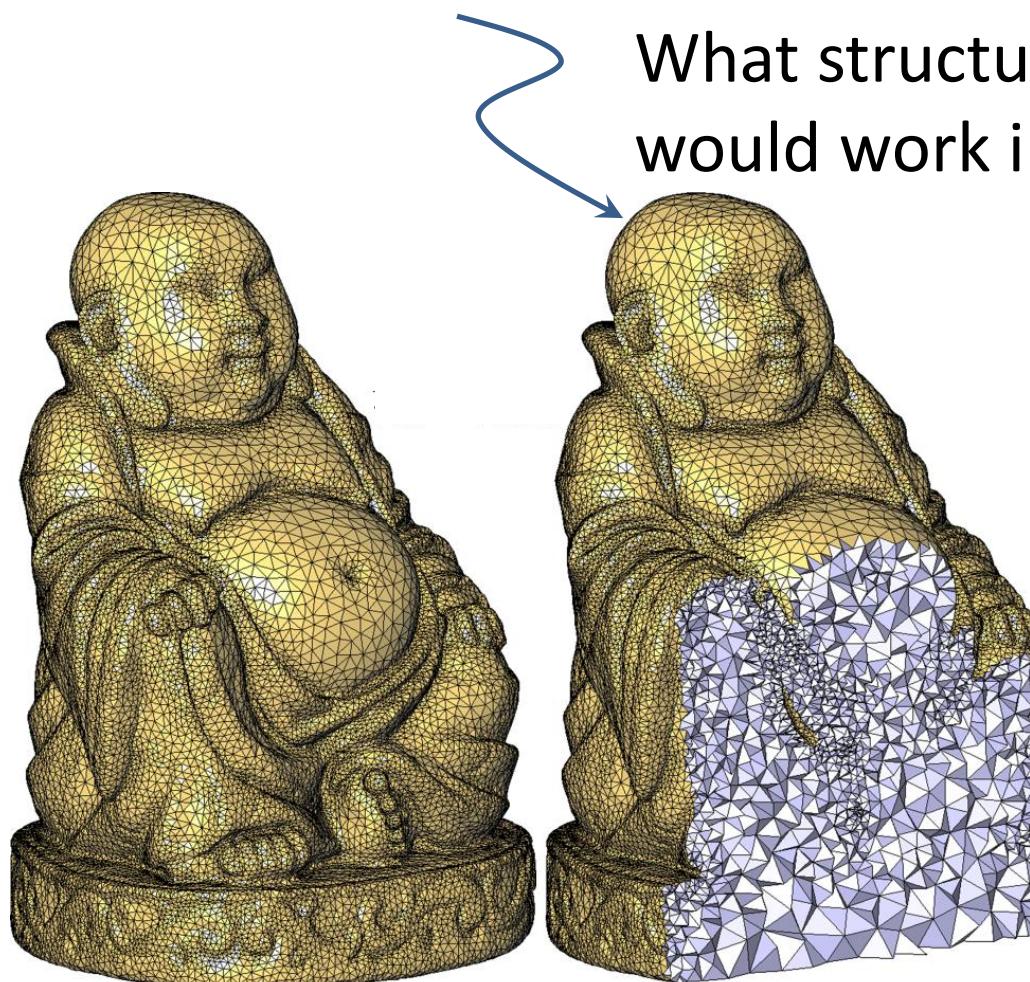
Planar versus non-planar

- **None** of the structures permit edge-crossings (or ‘edge intersections’), because they would create ill-defined faces; some non-edge 3D points could be members of multiple faces!



How would you organize 3D ‘volume’ meshes?

Our meshes define boundaries of 3D solids,
but not their 3D interiors.



END

***SOME of these SLIDES
WERE ADAPTED FROM:***

Mesh Representation, part II

based on:

Data Structures for Graphics, chapter 12 in
Fundamentals of Computer Graphics, 3rd ed.

(Shirley & Marschner)

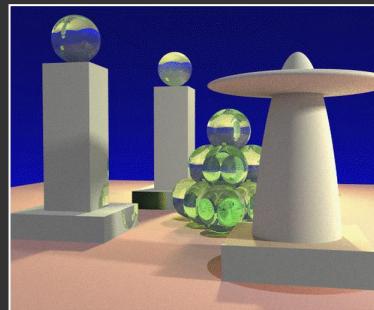
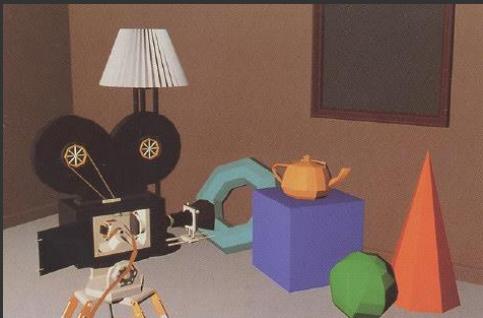
Source slides by Marc van Kreveld

**AND SOME OTHER SLIDES ADAPTED FROM:
Advanced Computer Graphics
(Spring 2013)**

CS 283, Lecture 4: Mesh Data Structures

Ravi Ramamoorthi

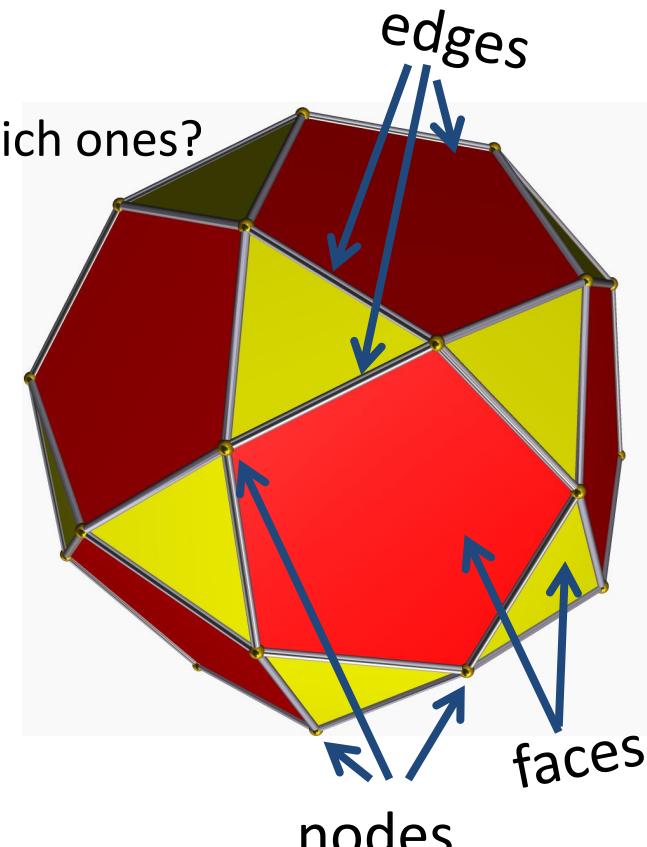
<http://inst.eecs.berkeley.edu/~cs283/sp13>



3D objects: Consider this Sphere...

MORE AND MORE QUESTIONS:

- Given a face F , what is the set of triangles that make up that face?
 - What is the set of all nodes for that face?
 - What is the set of all adjacent edges?
 - What is the set of all adjacent faces?
- Given a triangle, what is the set of adjacent triangles?
 - Are any adjacent triangles in a different face? Which ones?
 - Which face contains our triangle?
 - Does our face contain any other triangles?
What are they?
- Given an edge, what triangles include it?
 - Which faces meet at our edge?
 - Which vertices will our edge connect?
 - What other vertices share the same nodes?
 - ... (and many more!)...



... ENDLESS GEOMETRIC QUESTIONS!

WHAT GEOMETRIC DATA STRUCTURES

CAN ANSWER THEM EASILY/QUICKLY/RELIABLY?

Quaternions

For Blended 3D Rotations

(Part 1:)

Jack Tumblin

Northwestern Univ COMP_SCI 351-1

Fall 2021

from 2015 Ian Horswill slides, Northwestern Univ.
Heavily revised by Jack Tumblin

Quaternions for Blended 3D Rotations

wait: **WHUT?? WHY do we need them?**

- We can **already** blend together rotations by using Scene Graphs.
- We can **already** create and apply ANY 3D rotation as a **4x4 matrix**.
- We can **already** make that matrix from ANY **3D axis + angle**.
- We can **already** use handy 3-tuples (Yaw, Pitch, Roll; Euler Angles...)

BECAUSE: Rotation Matrices have THREE Ugly Failures:

- matrix degeneracies accumulate from finite precision, and
- no good, simple interpolation: weird, twisted, kinked results
- 3-tuple rotation models don't accumulate well: 'gimbal lock'

Axis/angle matrices fail in 2 BIG WAYS:

(1) When you try to **COMBINE** an endless stream of rotations...

- Endless **4x4 matrix** Multiplies? The matrix gradually degenerates.
Tiny numerical errors add in non-rotation transforms
(unwanted translation, scale, skew, or worse)

Project B → Starter Code → Quaternions →

[2.02.HelloMatrix_Degen.html](#)

- **3D axis + angle**: How would we combine just 2 of them?
it's possible, but ugly, messy & elaborate.

(2) When you try to **INTERPOLATE** between 2 very different rotations

- **4x4 matrix**: Hmm. How would you do it?
No well-known, practical, plausible geometrically-valid interpolation method.
3-angle sets: Euler angles,
- **3D axis + angle**: UGLY! Nothing good! Full of treacherous discontinuities,
kinks, and quirks! Gimbal Lock! Euler angle interpolations near 'north-pole'?

Quaternions: The Exotic Answer

Quaternions: EASY TO USE!

- we can **COMPUTE** them from **3D axis + angle**
- we can **INTERPOLATE** them quickly, easily, accurately
(quaternion ‘SLERP’, or by a fast approx.)
- we can **CONVERT** them back to **3D axis + angle** or **4x4 matrix**
- we can **APPLY** all with [cuon-matrix-quat03.js](#) library

Quaternions: WEIRDNESS-FREE 3D ROTATIONS!

- they **never ‘degenerate’** : numerical error == rotation error *only*
- their interpolated rotations **always look good**, because:
 - **minimum-energy paths** → maximum smoothness guaranteed
 - suitable for computing **3D rotational inertia**
(BTW, spacecraft attitude-control tasks revived quaternion use)

What Math Best Describes
3D Orientation
& All Applied **Rotations?**

part 1: angles, axes, and matrices

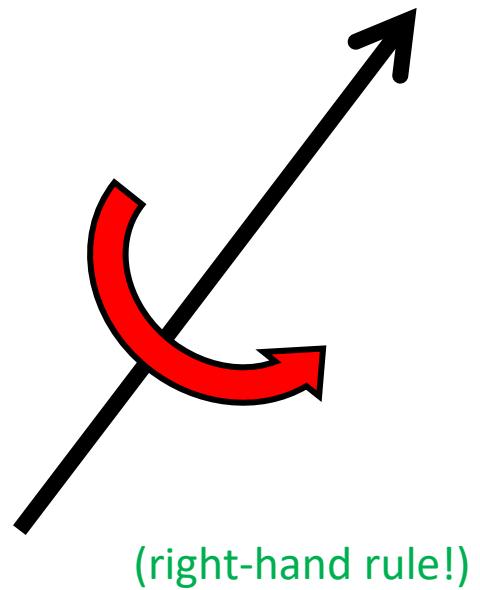
A Good & Sensible Answer: a 4-tuple -- Axis direction + Angle

- **Euler's rotation theorem** (Roughly):

In 3D-space, the result of
any series of rotations is same as
one single rotation about some axis

- The best rotation data structure
then *seems* obvious:

- Figure out the equivalent axis/angle
- Represent the axis direction (somehow)
- Represent the angle (e.g. in degrees)
to rotate about the axis



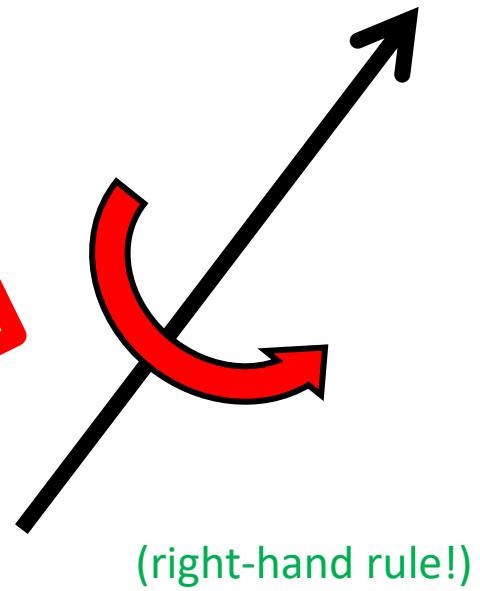
A Good & Sensible Answer:

Axis direction + Angle

- **Euler's rotation theorem** (Roughly):

In 3D-space, the result of
any series of rotations is same as
one single rotation about some axis

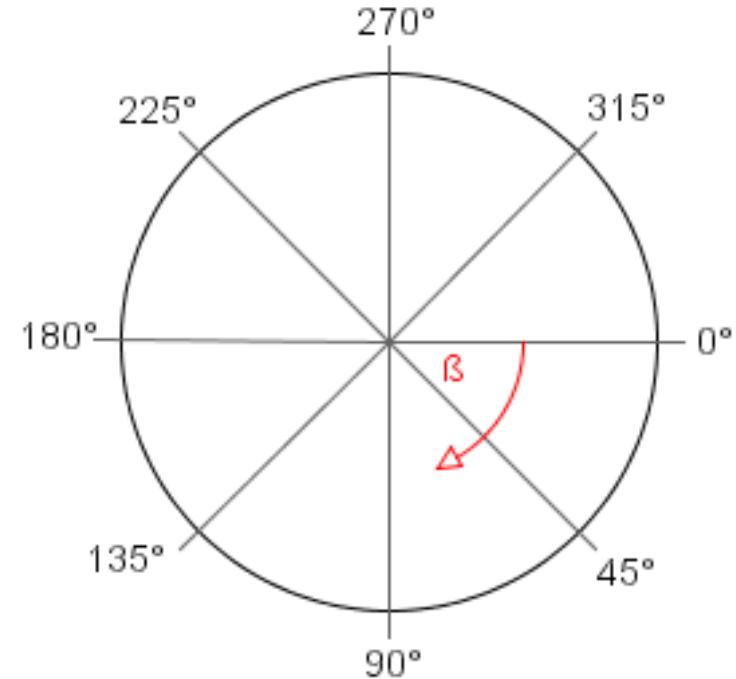
- The best rotation data
then seems to be:
 - Figure out the rotation axis/angle
 - But we still need the axis direction (somehow)
 - And the angle (e.g. in degrees)
 - So we have to state something about the axis



(right-hand rule!)

Smaller Troubles: Numbers for Angles (even in 2D!)

- Numbers and angles are not topologically equivalent
 - Angles are positions on a **circle**
 - Real numbers are positions on a **line**
- Ugh! Requires either:
 - **Discontinuities**, or
(wrap-around from 360 to 0) or
 - **Duplications**
(0, 360, 720, ... all represent the same angle)
- (*Either way: more messy & more difficult*)
How would you smoothly interpolate from 340° to 17° ?

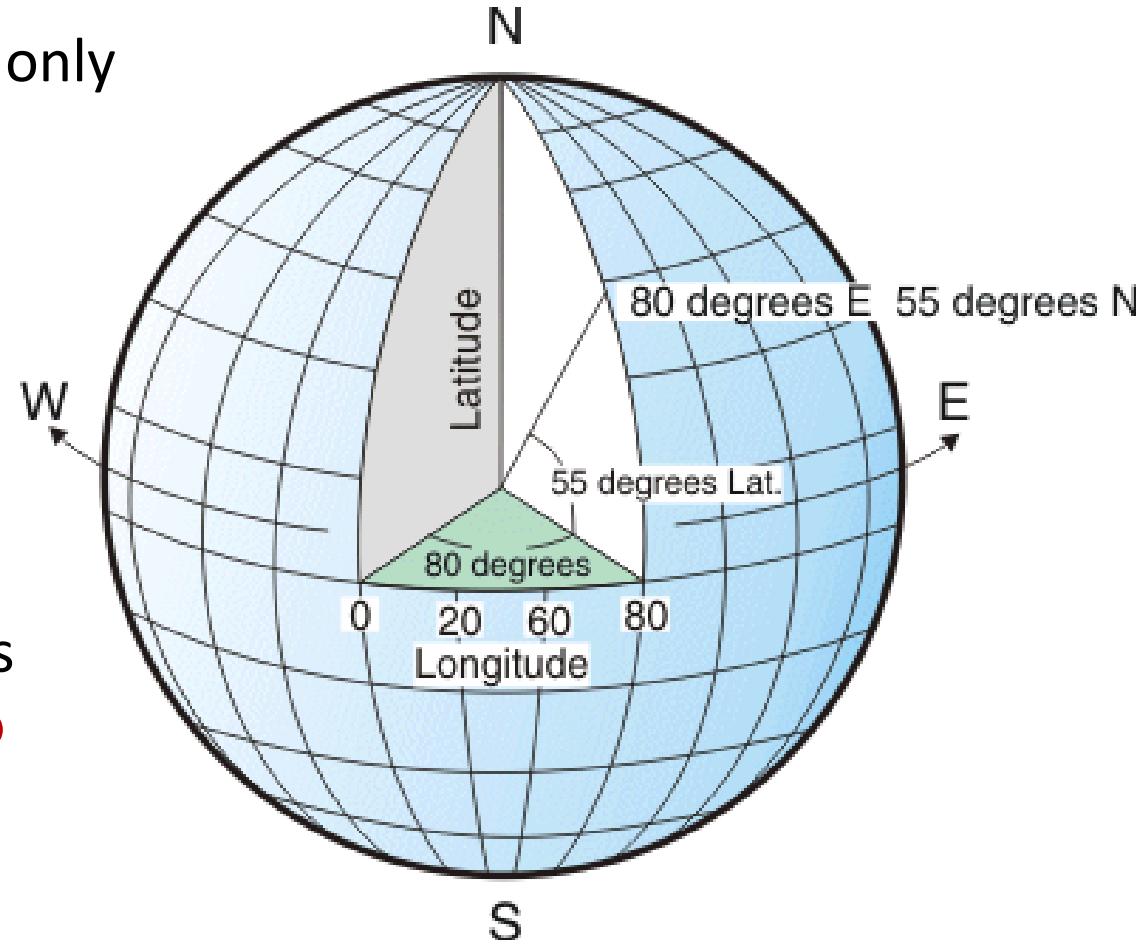


Should angle numbers increase
'Clockwise' or 'Counter-Clockwise'?
Which is 'right-handed',
Which is 'left-handed'?

Larger Troubles: How should we specify axis ‘direction’?

The ‘numbers-for-angles’ problem gets worse:

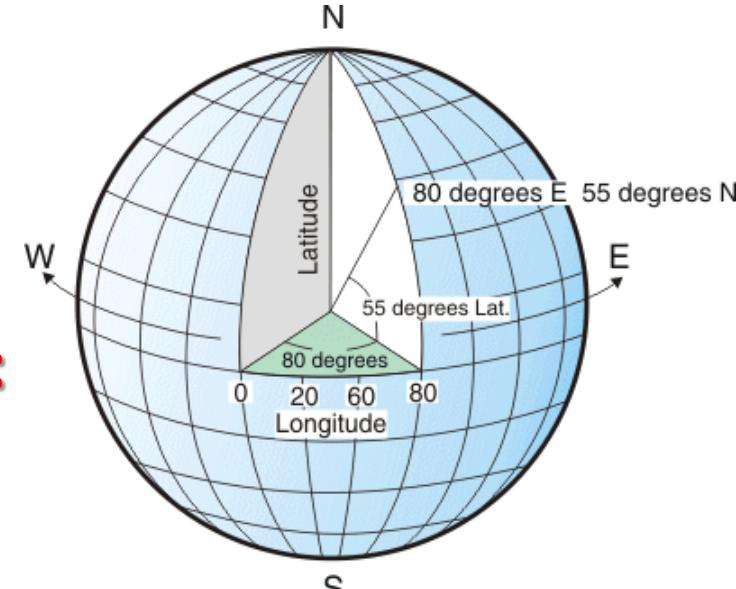
- In **3**-space, a direction has only **2** degrees of freedom:
- So you would **expect** to use just **2** numbers,
- For **2** separate, wholly independent angles
 - E.g. **longitude** and **latitude**
 - (*But these two numbers not really separate, and not really independent!*)



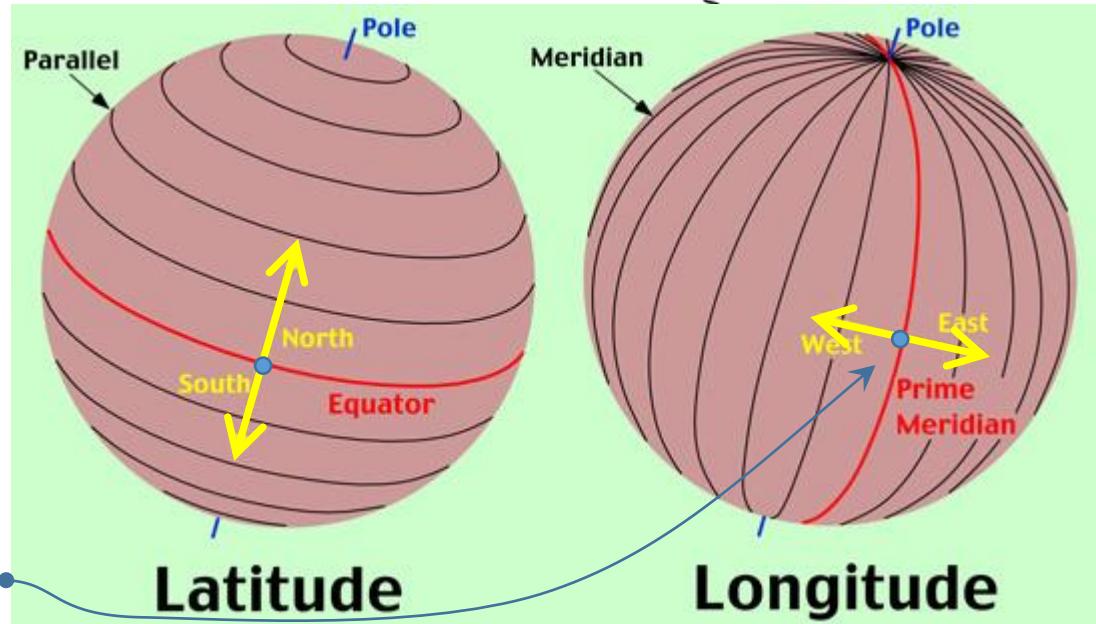
Larger Troubles: Coupled angles for 3-D Axis direction

Mismatched Angle Topology:

- We can choose to use **two angles**
- But they **interact in strange ways:**



- What's the longitude of the north pole? south pole?
 - No longitude?
 - Every longitude?
- What's the best 'origin' for longitude? Why Greenwich?

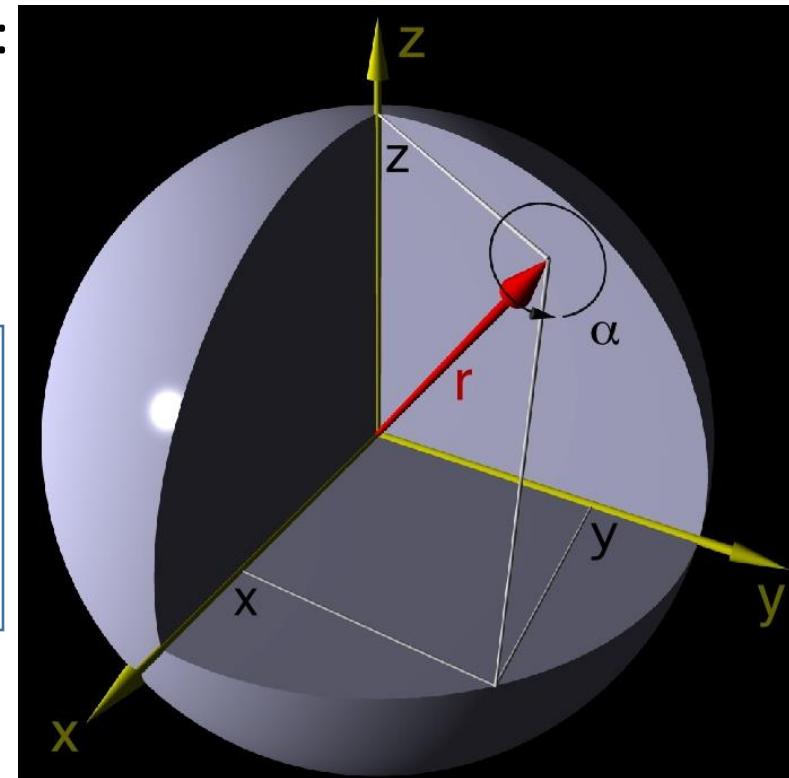


Cartesian Coords can help a bit:

Don't use angles for axis – Set 3D positions instead!

- A ***direction*** in ***n - space*** has only ***n – 1*** degrees of freedom (DOF), but we can
- Describe the axis by ***n*** numbers anyway:
make a unit vector in *n - space*
by normalizing the vector from origin.

Result: **all *n*-space points**
except the origin point (trouble? No...)
describe one axis
with ***n – 1*** DOF.



Again:

How can we represent 3D Axis + Angle?

Separately: as explicit **axis and angle**

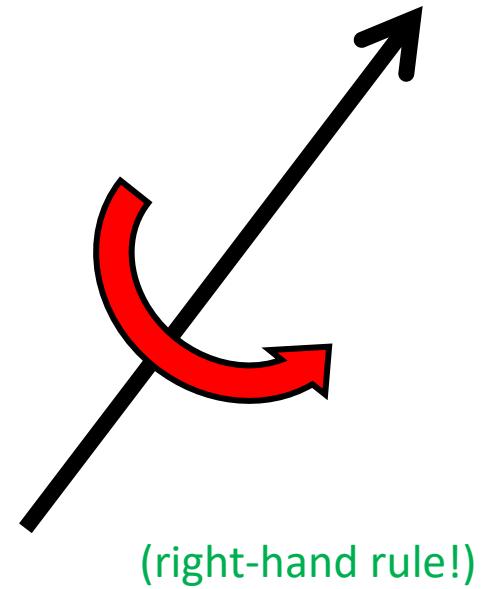
- Axis = unit-length vector in 3D space
- Angle = separate scalar value

OR

Combined: as one scaled **Rotation vector**

Make a unit vector axis, scaled by angle amount

- Vector Direction == axis of rotation
- Vector Magnitude = angle of rotation



Apply it: Rotate one point \mathbf{v}

Rodrigues' rotation formula (google it!) lets us:

- Rotate a point \mathbf{v} about the origin
- By a specified angle θ
- Around the axis ω specified by unit vector

$$\mathbf{v}_{new} = \mathbf{v} \cos \theta + (\boldsymbol{\omega} \times \mathbf{v}) \sin \theta + \boldsymbol{\omega}(\boldsymbol{\omega} \cdot \mathbf{v})(1 - \cos \theta)$$

Cool! -- You can also write this as a **4x4 matrix**,
as was already done for you in the
textbook's **cuon-matrix-quat03.js** library
(use **Matrix4.rotate()** or **Matrix4.setRotate()** fcns)

Angle & Axis: Pros and cons

Pros

- Relatively **simple**
- **Compact**
(uses very little storage)
- **~Efficient** to compute
- Good for Scene Graphs:
a 4x4 matrix,
just like all other transforms.

Cons

- Messy to **combine** rotations:
 - If we first apply one rotation
 - And then apply another,
 - What's their combined axis?
what's their combined angle?
- Messy to **interpolate** rotations:
 - What's the axis-angle pair
exactly halfway
between two axis-angle pairs?
 - What if those axes point in
exactly opposite directions?
 - Or *nearly*-opposite directions?

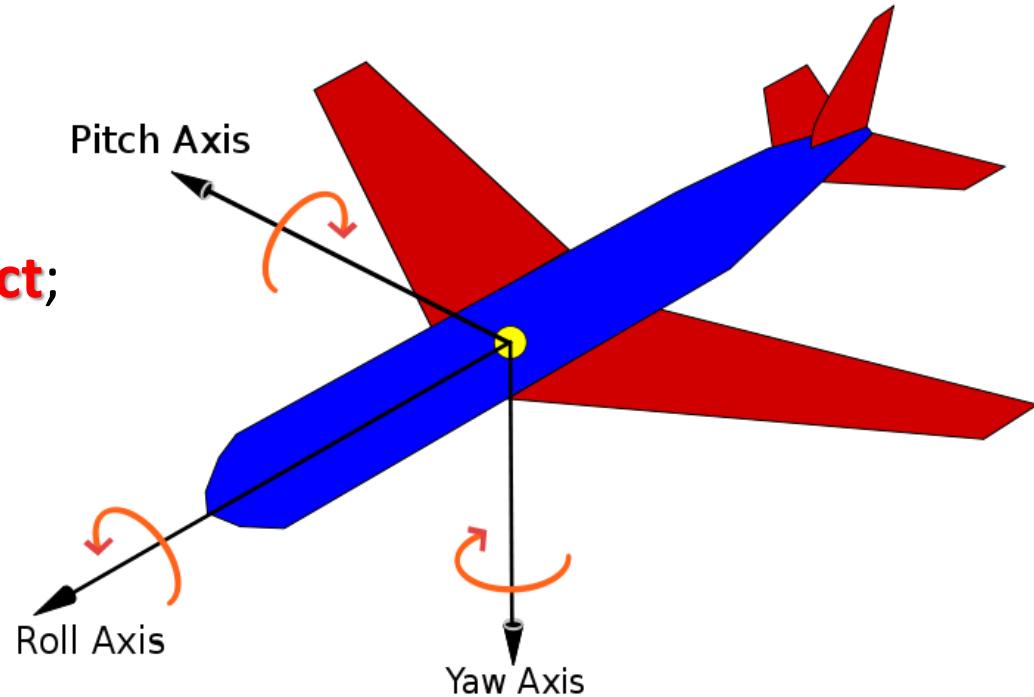
Isn't 4 too much? Why not a 3-tuple?

Yaw, pitch, and roll angles (“ypr”)

- Popular rotation method (aviation, etc) that uses standard set of **body-centered axes**:

- Up/down axis (yaw)
- Left/right axis (pitch)
- Front/back axis (roll)

- The axes are **fixed to the object**; they move and rotate with it.



- Great for small maneuvers: airplane steering, etc.
- Any rotation can be described as a **sequence** of these rotations

Yaw, Pitch, Roll: Pros and cons

Pros

- Easy to understand
- Compact
- Fairly easy to compute
- **Sensible for airplanes**

(for navigating a mostly-2D world with gravity that limits pitch, roll & z-axis travel)

Cons

- **Not** simple to convert between Angle-Axis & Yaw-Pitch-Roll angles
- **Not** commutative; sequence matters!
 - Roll ϕ , then yaw θ
 - Is different from yaw θ , then roll ϕ

- **Still** quite hard to compose
- **Still** can't interpolate well

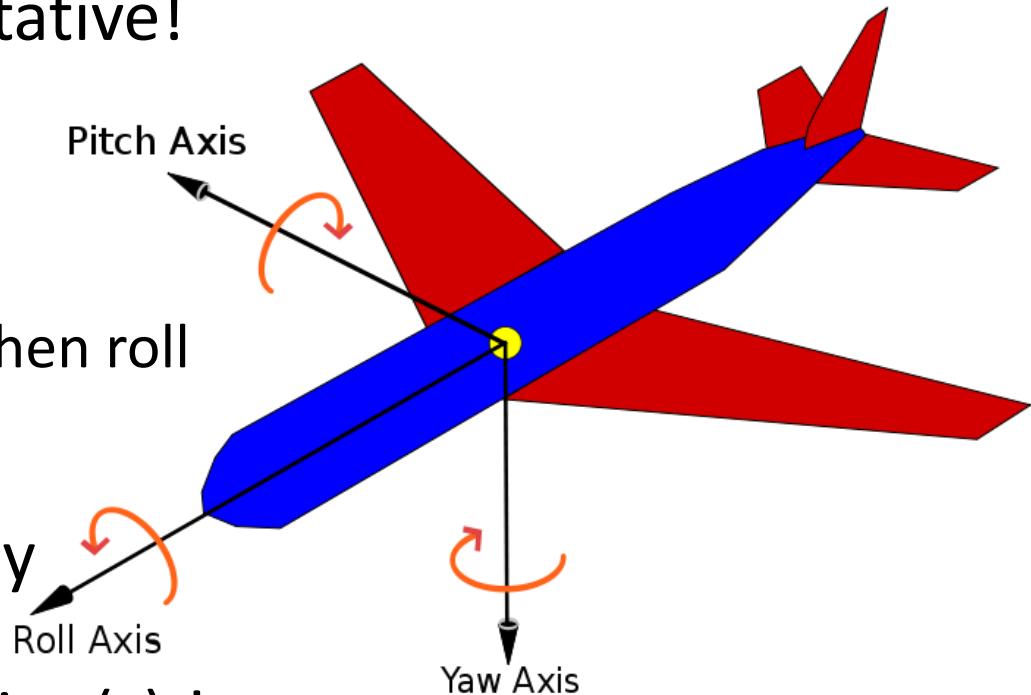
Yaw, Pitch, Roll? In which order?

- Rotations are not commutative!
Must choose & use
a consistent **order**
for rotations:

- E.g. yaw first, then pitch, then roll

- But each rotation we apply
rotates the body axes
used for all the later rotation(s) !

- TRY IT: What happens if the
second angle (pitch) is 90 degrees?



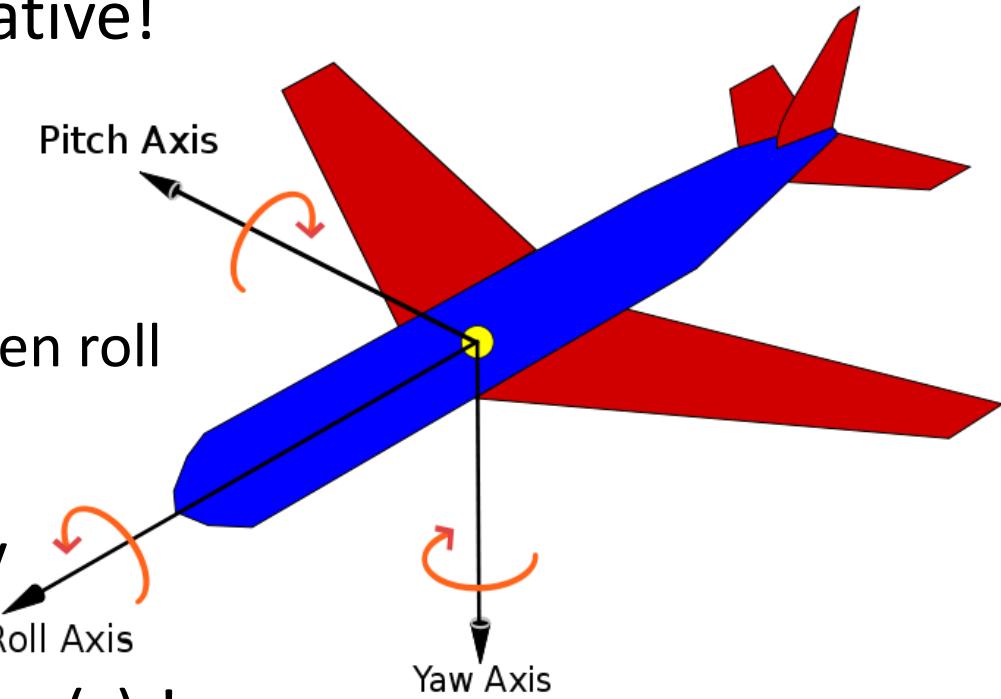
Yaw, Pitch, Roll? In which order?

- Rotations are not commutative!
Must choose & use
a consistent **order**
for rotations:

- E.g. yaw first, then pitch, then roll

- But each rotation we apply
rotates the body axes
used for all the later rotation(s) !

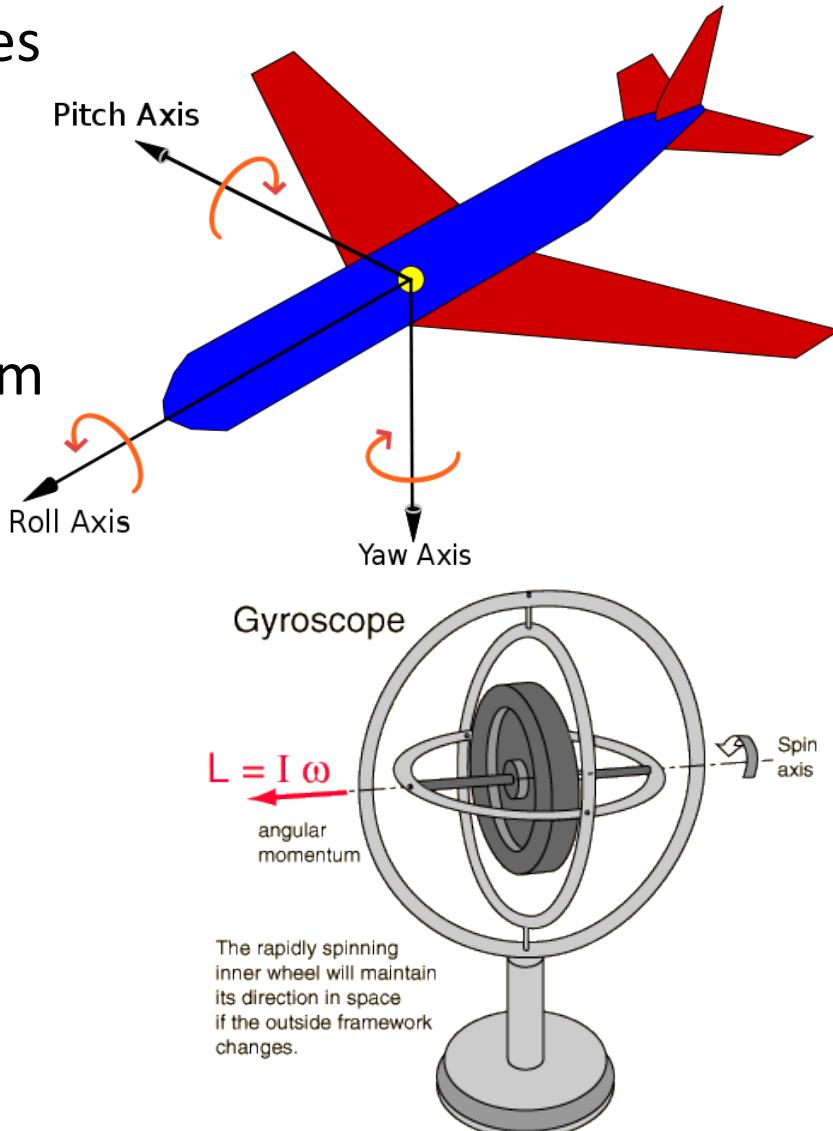
- TRY IT: What happens if the
second (pitch) angle is 90 degrees?



UH-OH.
BIG TROUBLE!

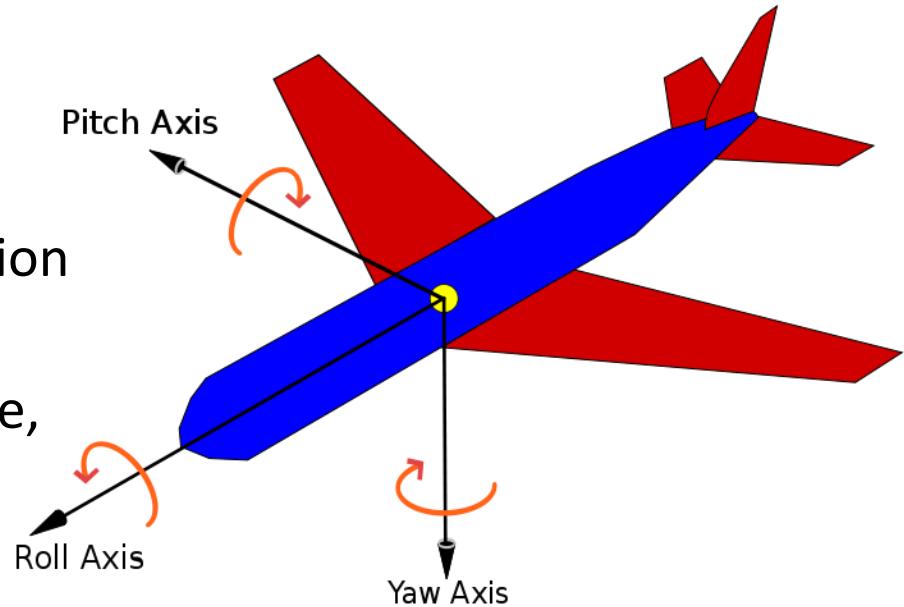
YPR: “Gimbal lock” (a toy-gyroscope problem)

- If the second angle (pitch) is 90 degrees
 - Then it rotates the **last axis** (roll) to
 - **The same as the first axis!** (-yaw)
- The **three axes** of Yaw-Pitch-Roll system can collapse to just **two axes** for any multiples of 90 degrees!
- Called “**gimbal lock**”, this problem bedevils mechanical gyroscopes used in inertial guidance systems



Gimbal lock

- ? Is gimbal lock a problem for *your* graphics programs ?
- ANS: It depends on your application
- For an ‘First-Person Shooter’ game, representing the character’s gun by its yaw and pitch (*aka* longitude and latitude) makes perfect sense (?do you need roll for bullets?)
- But it’s a terrible complication for:
tumbling spacecraft, steering by rocket-power, or spinning objects tossed into the air (interpolation)



How are “Euler angles” different?

- Don't confuse them with yaw/pitch/roll:

- YPR angles apply to axes that stay **fixed to the rotated object**
- Euler angles apply to axes that stay **fixed to the Earth.**

- They rotate **axes**, not objects:
THUS no gimbal lock!

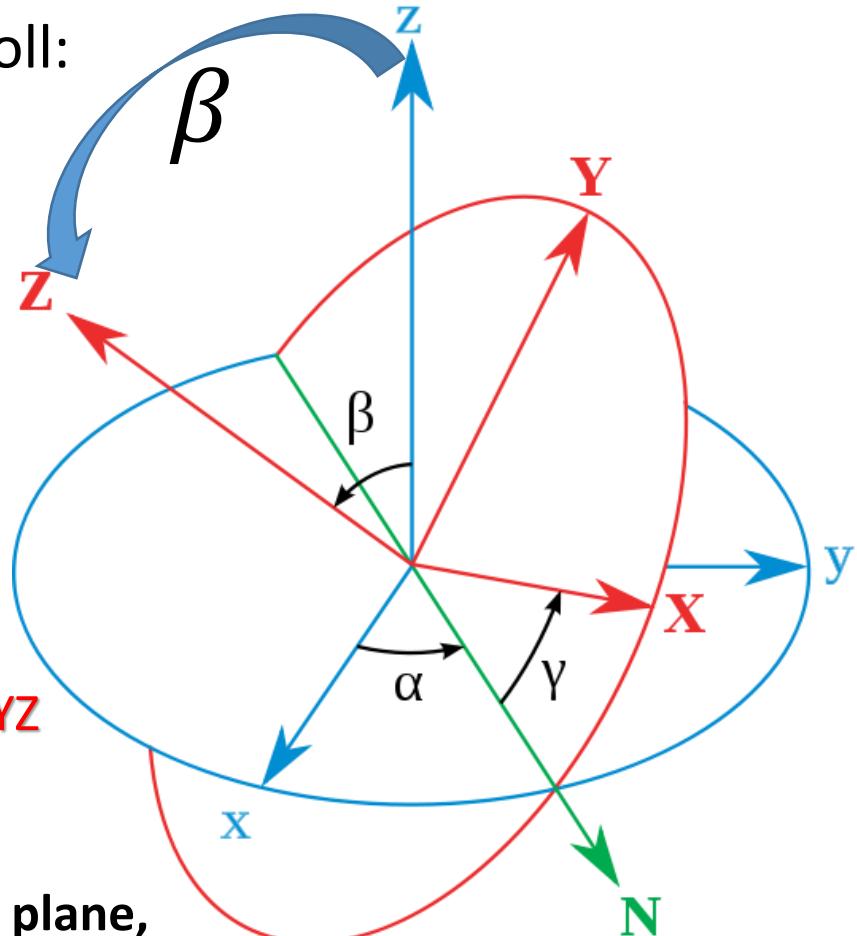
- Define:

- Fixed ‘3D World’ axes labeled x, y, z
- Our object’s new, rotated axes labeled XYZ

What happens if we move the **Z** axis
away from **z** axis by some angle β ?

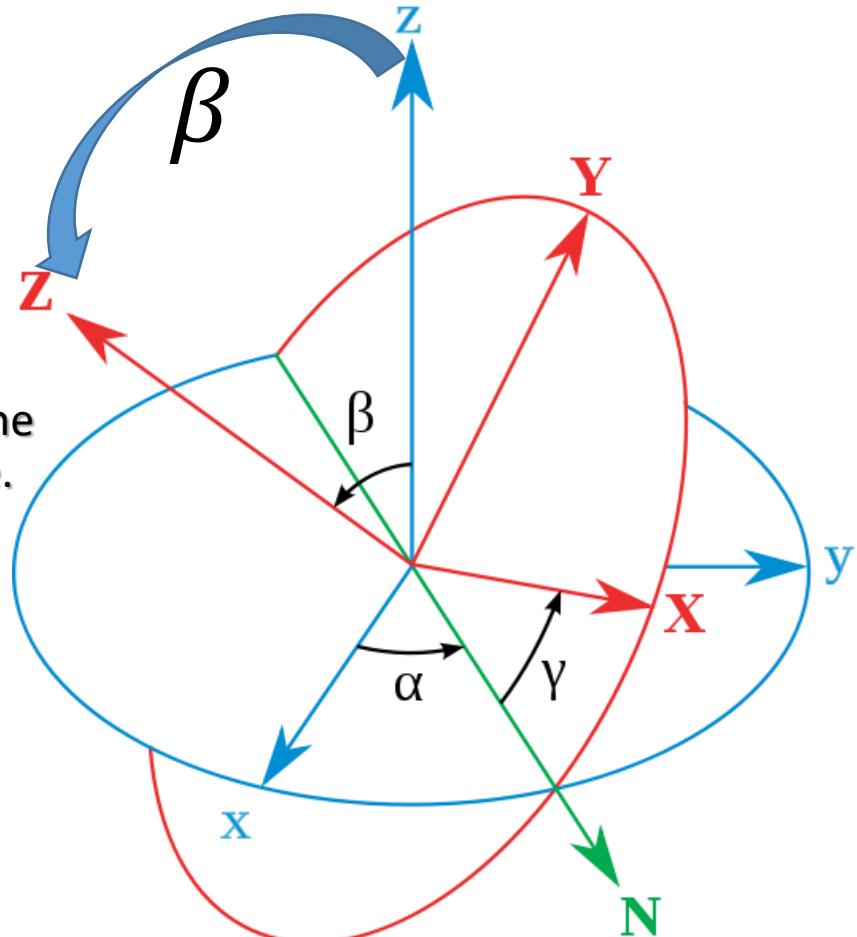
→ We Tilted the old x, y plane to form new XY plane,

→ Old/new planes intersect to form a line ('**N**' vector)



Euler angles (α, β, γ) set the axis displacements

- Define:
 - Fixed '3D World' axes labeled x,y,z
 - Our object's new, rotated axes labeled XYZ
 - The temporary vector N , *always* in the x,y plane where it intersects with the (rotated) XY plane.
- Then we can define Euler angles formally as:
 - α , the angle between the old x axis and N
 - β , the angle between the old z and new Z axis
 - γ , the angle between N and the new X axis



(Why bother with N ?

Why not measure angles directly between old & new x,y,z axes?

Try it!

You will find that using N greatly simplifies axis-finding...)

Rotation Matrix from Euler Angles?

SCARY-LOOKING FORMALISMS:

- Rodriguez's formula is **linear** in \mathbf{v} , thus
- For any $\boldsymbol{\omega}, \theta$ we can find a **matrix** that rotates vectors around $\boldsymbol{\omega}$ by θ :

$$\mathbf{v}_{rot} = \mathbf{v} \cos \theta + (\boldsymbol{\omega} \times \mathbf{v}) \sin \theta + \boldsymbol{\omega}(\boldsymbol{\omega} \cdot \mathbf{v})(1 - \cos \theta)$$
$$\mathbf{M}_{\boldsymbol{\omega}, \theta} = \mathbf{I} + [\boldsymbol{\omega}]_{\times} \sin \theta + (1 - \cos \theta) \boldsymbol{\omega} \boldsymbol{\omega}^T$$

- Ugh.

--- A Simple (but inefficient) Solution for WebGL---

Use your angle/axis functions step-by-step to construct the matrix:

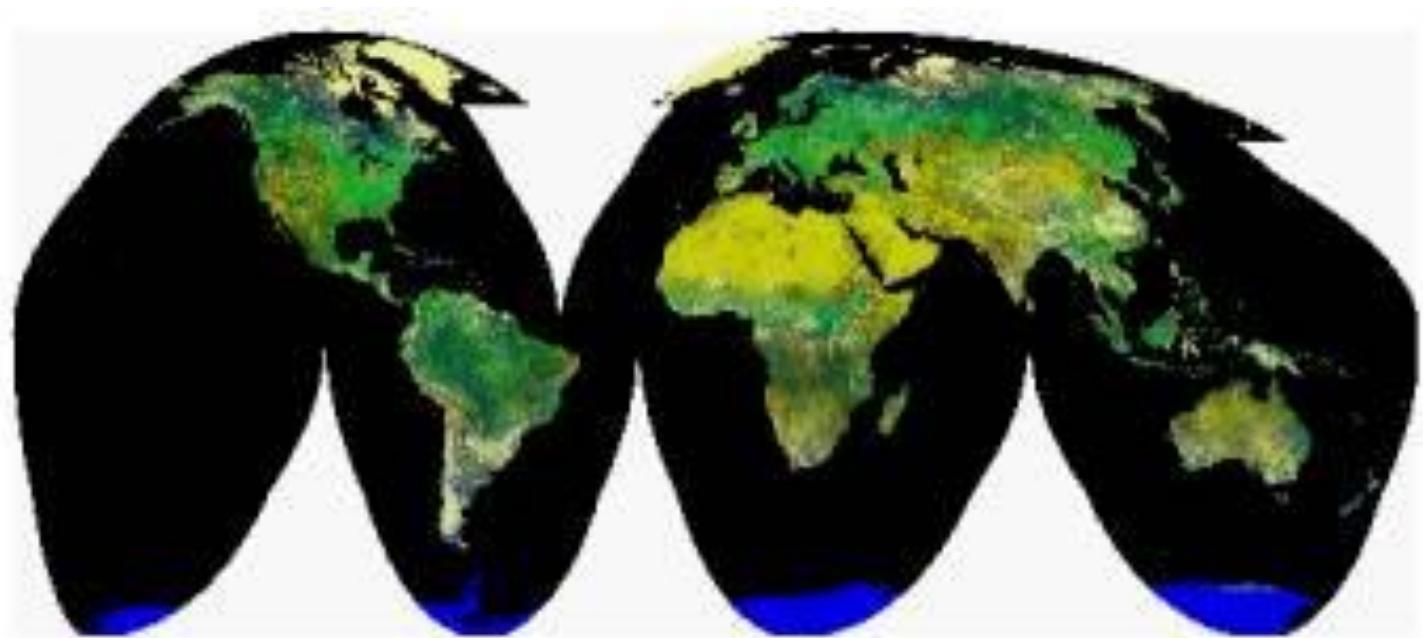
```
rotate( $\alpha$ , 0,0,1);    // rotate on z axis to put x axis onto N vector,  
rotate( $\beta$ , 1,0,0);    // then rotate on x axis to put z axis onto Z axis,  
rotate( $\gamma$ , 0,0,1);    // then rotate on Z axis again to put y on Y axis.
```

Recall:

Rodrigues' rotation formula:

- Rotate **a point** \mathbf{v} to find \mathbf{v}_{rot}
- About the **origin**
- By a specified **angle** θ
- On axis of the **unit vector** $\boldsymbol{\omega}$

Why is Combining Hard? Discontinuities!



Ultimately, the problem is that

- There can be no mapping
from any set of 3 **real** numbers **to** rotations
that ***will not*** have discontinuities.

Analogous problem: mapping globe's surface on a plane

“imaginary number”
representations

Complex numbers: Review

Can be represented as
 $a + bi$

We must assume

- $i = \sqrt{-1}$ exists, and follows rules of ordinary arithmetic:
 - Associativity
 - Commutativity
 - Distributivity
- Works with all the usual algebraic operators: $+$, $-$, \div , \times , (\cdot) , ...
- ‘complex’ numbers: $(a + bi)$ mix real: a and imaginary: bi
- Addition & Multiplication holds no surprises
(just remember $i^2 = -1$)

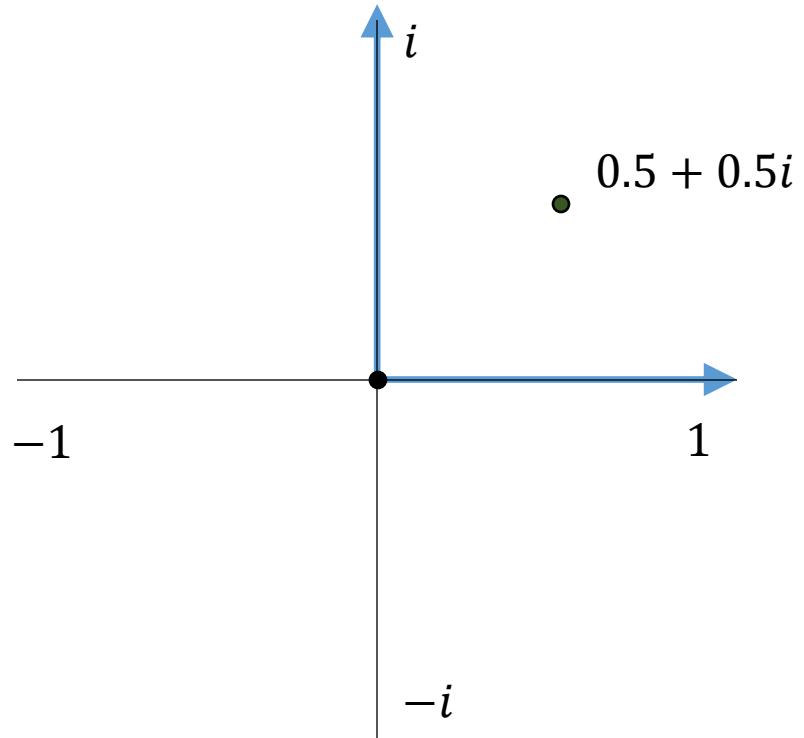
$$\begin{aligned}(a + bi) + (c + di) &= (a + c) + (b + d)i \\(a + bi)(c + di) &= (ac) + bci + adi + bdi^2 \\&= (ac - bd) + (bc + ad)i\end{aligned}$$

The complex plane

- We can visualize complex numbers as a 2D space

- With the axes

- Real (horizontal): ± 1
- Imaginary (vertical): $\pm i$



Geometry of complex numbers

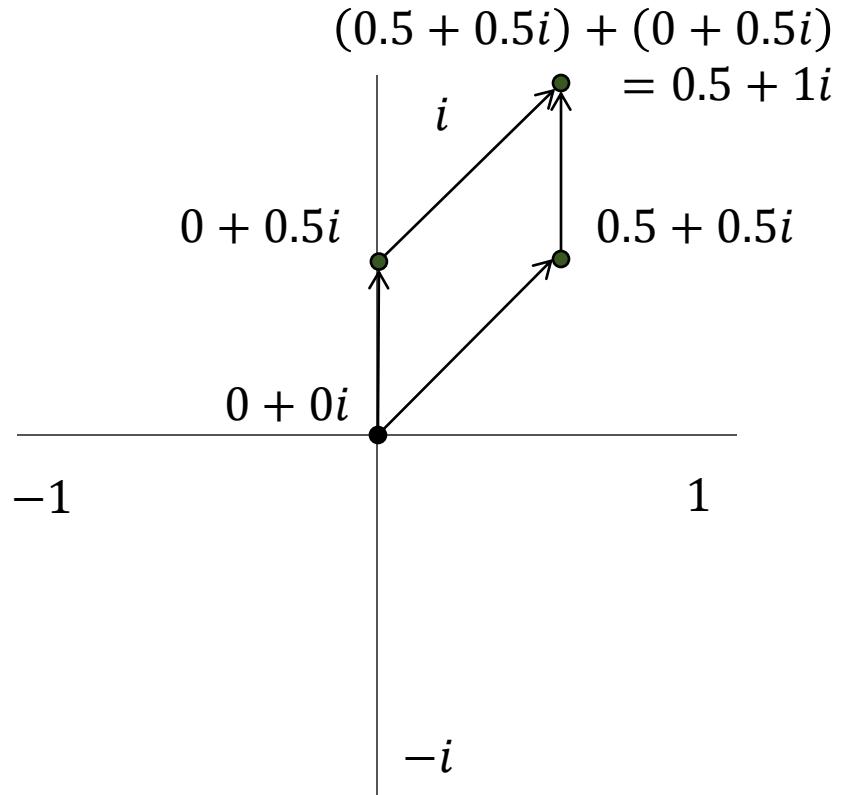
- ON this ‘complex’ 2D plane,
complex number addition
is same as
vector addition

Complex numbers:

$$\begin{aligned}(a + bi) + (c + di) \\= (a + c) + (b + d)i\end{aligned}$$

Vectors:

$$\begin{aligned}(a, b) + (c, d) \\= (a + c, b + d)\end{aligned}$$



Complex Multiplication

“rotates” points around the origin

- People quickly realized that multiplying by **powers of i** rotates a number by 90 degrees

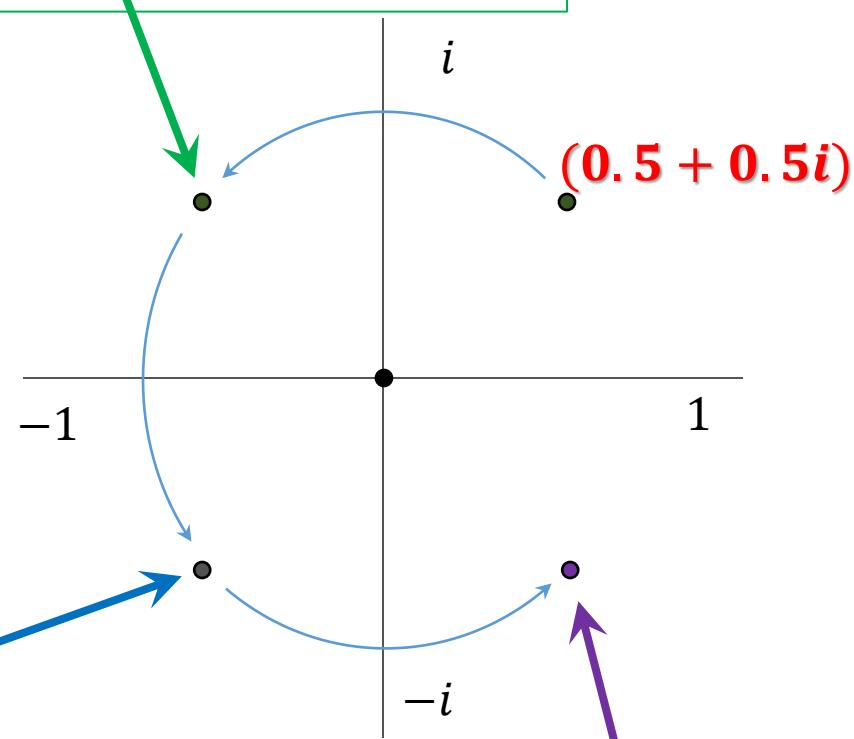
$$\begin{aligned} i(0.5 + 0.5i) &= 0.5i^2 + 0.5i \\ &= -0.5 + 0.5i \end{aligned}$$

- And, in fact:

- Multiplying by $\sqrt{i} = \frac{1+i}{\sqrt{2}}$ rotates by 45 degrees

- Multiplying by $\sqrt[3]{i}$ rotates by 30 degrees

$$\begin{aligned} i^2(0.5 + 0.5i) &= -1(0.5 + 0.5i) \\ &= -0.5 - 0.5i \end{aligned}$$



$$\begin{aligned} i^3(0.5 + 0.5i) &= -i(0.5 + 0.5i) \\ &= 0.5 - 0.5i \end{aligned}$$

Complex Numbers in Polar Coordinates

- We can rewrite complex numbers as

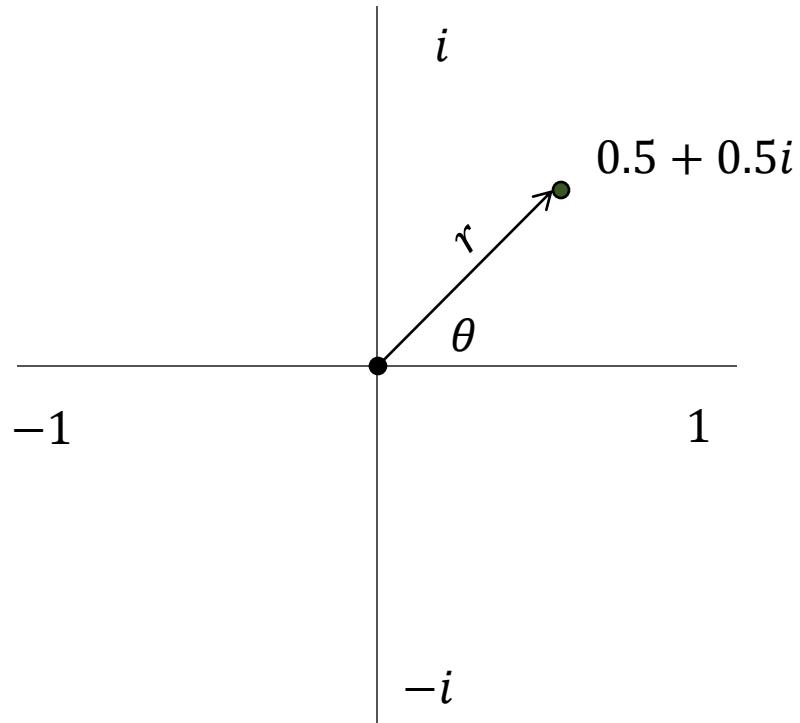
- A **magnitude** and
(distance from the origin)

$$|a + bi| = \sqrt{a^2 + b^2}$$

- An **angle** from the real axis:

$$\theta(a + bi) = \text{atan2}(a, b)$$

- Multiplication will then
 - Add angles, and
 - Multiply magnitudes



Rotations

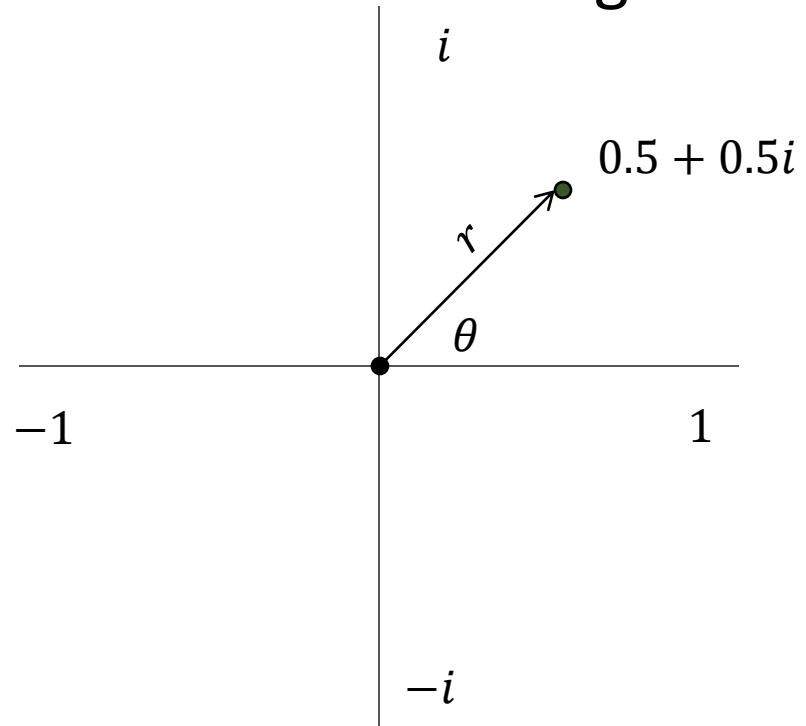
as unit-magnitude numbers

We can then **represent any rotation about the origin as**

- A **multiplication** by a complex number
- with a **magnitude of 1**

Awesome!

Uh, **where** do we get these magical “unit magnitude” numbers?

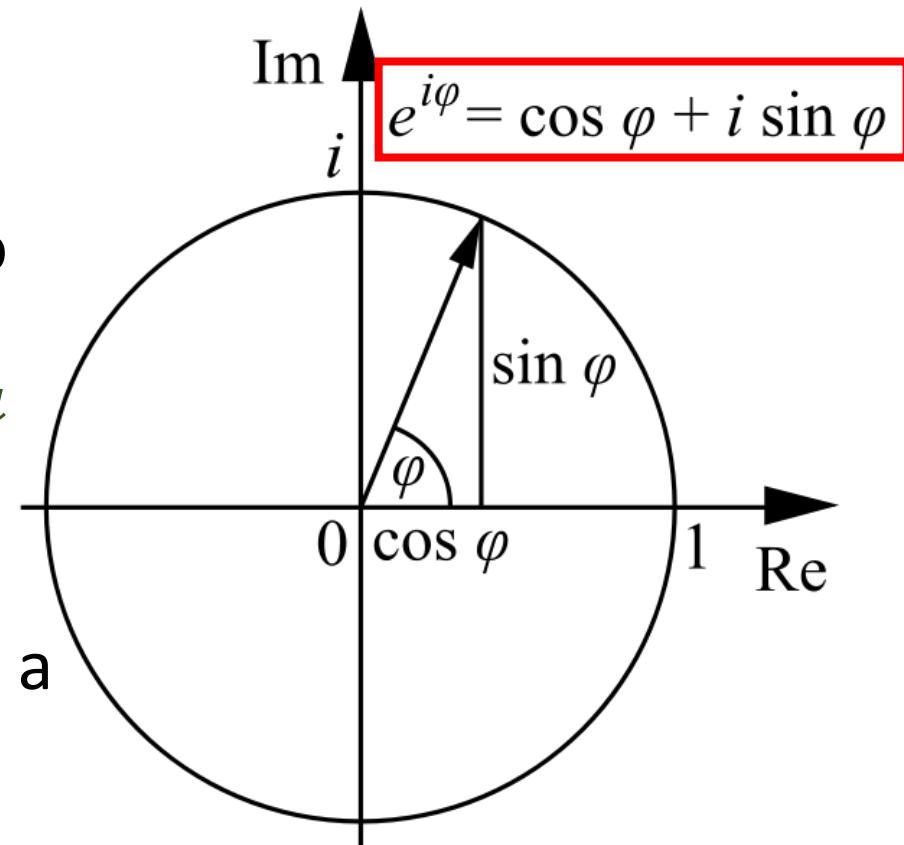


Exponent for Complex Numbers?

Euler's formula

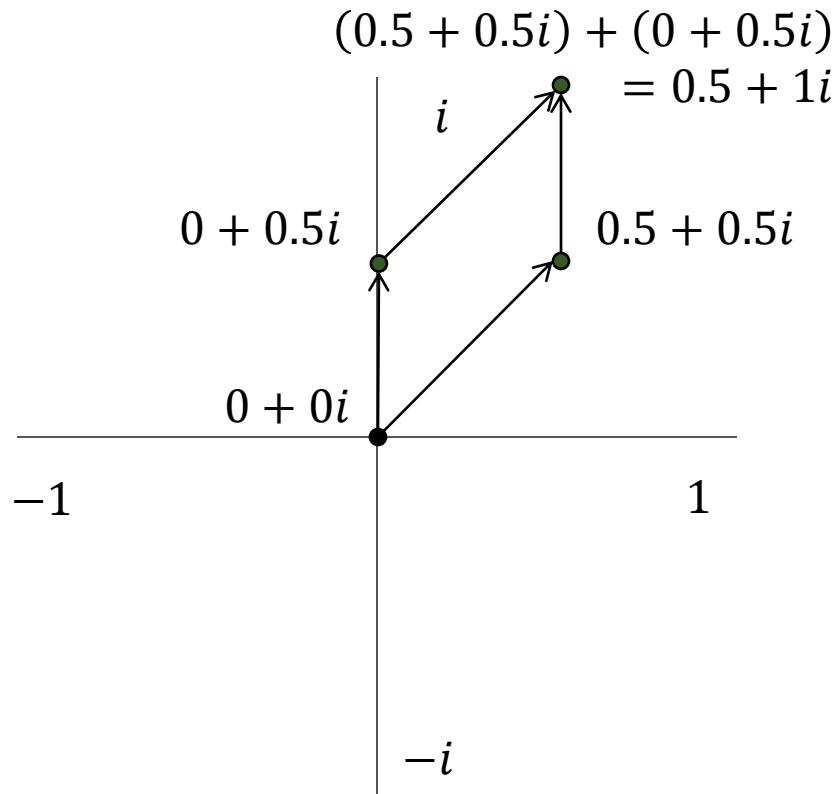
Euler showed that:

- When raising a real number to a complex power: $a + bi$
 - The **magnitude** was scaled by a
 - The **angle** was rotated by b
- We can get unit numbers with a given angle by raising e to a complex power
 - $e^{i\phi}$ is a ‘magical’ unit-magnitude number that rotates by ϕ radians



SUMMARY: Geometry from Complex Numbers

- Complex numbers let us perform
 - Translation using addition
 - Scaling by multiplication with a real-only number
 - **Rotation** by multiplication with a unit-length complex number



... **but only in 2D...**

What about 3D?

- early 1800s: a race to represent 3-space using complex numbers
- One obvious approach was to assume -1 had ***two*** square roots
- But this introduced several nasty new problems, and stumped some great thinkers for good while ...

Quaternions Invented

1843: W. R. Hamilton's great 'AHA!' on the Broome Bridge:
epically ingenious! http://en.wikipedia.org/wiki/Broom_Bridge

- Start with the *assumption* that -1 has **3** linearly independent square roots, not 2. i, j , and k
- Quaternions are linear combinations of one real scalar a and the imaginary basis elements (i, j, k) with coordinates (or 'weights') of b, c, d :

$$q = a + bi + cj + dk$$

- Quaternion addition ?
Easy, obvious, and what you'd expect

$$(a_1 + b_1 \mathbf{i} + c_1 \mathbf{j} + d_1 \mathbf{k}) + (a_2 + b_2 \mathbf{i} + c_2 \mathbf{j} + d_2 \mathbf{k}) = \\ (a_1 + a_2) + (b_1 + b_2)\mathbf{i} + (c_1 + c_2)\mathbf{j} + (d_1 + d_2)\mathbf{k}$$

Quaternion multiplication

(Why? 3D rotation!)

Quaternion \mathbf{q} defined as:

$$\mathbf{q} = a + bi + cj + dk.$$

How can we find $\mathbf{q}_1\mathbf{q}_2$?

Quaternion multiplication
gets complicated.

- Multiplication of the real components behaves normally. No surprises.
- But multiplication of the quaternion's **imaginary components** is quite strange:

Why, it's **anti-commutative!**

$$AB = -BA$$

Quaternion axiom:

$$i^2 = j^2 = k^2 = ijk = -1$$

From this you can prove that:

- $ij = k$
- $jk = i$
- $ki = j$

And by anti-commutativity:

- $ji = -(ij) = -k$
- $kj = -(jk) = -i$
- $ik = -(ki) = -j$

Quaternion multiplication

(Why? 3D rotation!)

The Hamilton product of two quaternions:

$$(\mathbf{a}_1 + \mathbf{b}_1\mathbf{i} + \mathbf{c}_1\mathbf{j} + \mathbf{d}_1\mathbf{k})(\mathbf{a}_2 + \mathbf{b}_2\mathbf{i} + \mathbf{c}_2\mathbf{j} + \mathbf{d}_2\mathbf{k})$$

$$\begin{aligned} &= a_1a_2 + a_1b_2i + a_1c_2j + a_1d_2k \\ &+ b_1a_2i + b_1b_2i^2 + b_1c_2ij + b_1d_2ik \\ &+ c_1a_2j + c_1b_2ji + c_1c_2j^2 + c_1d_2jk \\ &+ d_1a_2k + d_1b_2ki + d_1c_2kj + d_1d_2k^2 \end{aligned}$$

$$\begin{aligned} &= a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2 \\ &+ (a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)i \\ &+ (a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2)j \\ &+ (a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)k \end{aligned}$$

(yuk)

(Let's get a computer to do it for us!)

A Geometric Interpretation

**Consider two purely imaginary quaternions
(real part is zero):**

- $p = b_1i + c_1j + d_1k$,
- $q = b_2i + c_2j + d_2k$

Multiply them:

$$\begin{aligned} pq &= (b_1i + c_1j + d_1k)(b_2i + c_2j + d_2k) \\ &= \boxed{-b_1b_2 - c_1c_2 - d_1d_2} \\ &\quad \boxed{+ (c_1d_2 - d_1c_2)i + (d_1b_2 - b_1d_2)j + (b_1c_2 - c_1b_2)k} \end{aligned}$$

Do these two expressions look familiar?

A Geometric Interpretation

**Consider two purely imaginary quaternions
(real part is zero):**

- $p = b_1i + c_1j + d_1k$,
- $q = b_2i + c_2j + d_2k$

Multiply them:

$$\begin{aligned} pq &= (b_1i + c_1j + d_1k)(b_2i + c_2j + d_2k) \\ &= \boxed{-b_1b_2 - c_1c_2 - d_1d_2} \\ &\quad \boxed{+ (c_1d_2 - d_1c_2)i + (d_1b_2 - b_1d_2)j + (b_1c_2 - c_1b_2)k} \end{aligned}$$

Do these two expressions look familiar?

- the **real parts** look like a **dot-product** (negated),
- the **imaginary parts** look like a **cross-product...**

A Geometric Interpretation

You're right! If we write the imaginary parts of quaternions p and q as 3D vectors:

- $\vec{p}_v = (b_1, c_1, d_1)$
- $\vec{q}_v = (b_2, c_2, d_2)$

And Multiply those imaginary parts,
then result is **dot-product + cross-product:**

$$\begin{aligned} pq &= (b_1i + c_1j + d_1k)(b_2i + c_2j + d_2k) \\ &= -b_1b_2 - c_1c_2 - d_1d_2 \\ &\quad + (c_1d_2 - d_1c_2)i + (d_1b_2 - b_1d_2)j + (b_1c_2 - c_1b_2)k \\ &= -(\vec{p}_v \cdot \vec{q}_v) + \vec{p}_v \times \vec{q}_v \end{aligned}$$

Quaternions and 3-space

We can think a full quaternion:

$$q = a + bi + cj + dk$$

as a weird hybrid number-vector

- A scalar (real) part \mathbf{q}_s
(in this case, a)
- Plus an imaginary vector $\vec{\mathbf{q}}_v$
(in this case, (b, c, d))

**A full quaternion
is *already* an odd sort
of axis-angle
representation**

Given two full quaternions:

- $p = p_s + \vec{p}_v$
- $q = q_s + \vec{q}_v$

Write their product in vector form,
in parts:

$$\begin{aligned} pq = & p_s q_s \\ & -\vec{p}_v \cdot \vec{q}_v \\ & + p_s \vec{q}_v \\ & + q_s \vec{p}_v \\ & + \vec{p}_v \times \vec{q}_v \end{aligned}$$

(This will eventually turn out to be useful...)

Four Important Definitions

Define the quaternion q as:

$$\begin{aligned} q &= a + bi + cj + dk \\ &= q_s + \vec{q}_v \end{aligned}$$

Define the conjugate of q as:

$$\begin{aligned} q^* &= a - bi - cj - dk \\ &= -\frac{1}{2}(q + iqi + jqj + kqk) \end{aligned}$$

Define the magnitude of q as:

$$\begin{aligned} \|q\| &= \sqrt{qq^*} = \sqrt{a^2 + b^2 + c^2 + d^2} \\ &= \sqrt{q_s^2 + \vec{q}_v^2} \end{aligned}$$

Define the inverse of q as:

(inverse or reciprocal)

$$q^{-1} = \frac{q^*}{\|q\|}$$

So that:

$$qq^{-1} = q^{-1}q = 1$$

Polar decomposition

- A **unit quaternion**, like a unit vector, is a quaternion with a magnitude of 1
- Any quaternion can be represented in **polar form** as the product of its magnitude and its unit quaternion
- Any quaternion can be made into a unit quaternion by **dividing by its magnitude**

$$U_q = \frac{q}{\|q\|}$$

- (**In computer graphics, we use unit quaternions and almost no other kind**)

How to Use Unit Quaternions for 3D Rotations

- Unit quaternions describe all 3D rotations,
- Just as unit complex numbers describe all rotations in 2-space
- The rotation of a vector by a unit quaternion is given by:

$$\text{rotate}(q, \vec{v}) = q\vec{v}q^{-1}$$

- This unit quaternion:
$$q_{\vec{\omega}, \theta} = \cos \frac{\theta}{2} + \sin \frac{\theta}{2} \vec{\omega}$$
- Performs a rotation of:
- θ degrees
 - About axis vector $\vec{\omega}$ with coordinates (ax, ay, az)
- Multiplying unit quaternions combines their rotations
 - pq does the equivalent of
 - First rotating by q ,
 - Then rotating by p .

Quaternions: Why is this good?

- **No gimbal lock!**
- The extra, 4th degree of freedom in quaternions
 - Lets us represent 3D rotations **without discontinuities (!)**
 - Just as the extra degree within ordinary complex numbers ($a+bi$) lets us represent 2D rotations
- **Composition: easy**
 - Multiplication = composition
 - pq does q then p (**'backwards'**)
 - Exponentiation = iteration
 - q^n does q , n times, and thus
 - $\sqrt{q} = q^{\frac{1}{2}}$ does q halfway
- **Interpolation: ~easy**
 - q^t , where $0 \leq t \leq 1$, exact t fraction of the q rotation

How to Use **Quaternions** – in WebGL, please!

This unit quaternion:

$$q_{\vec{\omega}, \theta} = \cos \frac{\theta}{2} + \sin \frac{\theta}{2} \vec{\omega}$$

Describes a 3D rotation of:

- θ degrees
- about the axis vector $\vec{\omega}$
that I will write as (x_a, y_a, z_a)

And is written as the 4-tuple

$< q_w, q_x, q_y, q_z >$

where: $q_w = \cos(\theta/2)$,
 $q_x = x_a * \sin(\theta/2)$,
 $q_y = y_a * \sin(\theta/2)$,
 $q_z = z_a * \sin(\theta/2)$.

Multiplication of unit quaternions
performs **composition** of their rotations
 $p q$ does the equivalent of
First rotating by q ,
Then rotating by p ('it seems backwards')

- **Store** your rotations as quaternions (4-tuples)
- **Compose** your rotations as quaternions
- **Interpolate** rotations as quaternions (next class)
- **To apply** your rotations in a scene graph,
convert quaternions to 4x4 rotation matrix:

Rot(q) =

$$\begin{bmatrix} 1 - 2q_y^2 - 2q_z^2 & 2q_x q_y - 2q_w q_z & 2q_x q_z + 2q_w q_y & 0 \\ 2q_x q_y + 2q_w q_z & 1 - 2q_x^2 - 2q_z^2 & 2q_y q_z - 2q_w q_x & 0 \\ 2q_x q_z - 2q_w q_y & 2q_y q_z + 2q_w q_x & 1 - 2q_x^2 - 2q_y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

See Reading: Summary on pg 91

Lengyel: "Math for 3D Game Programming"

How to Use Quaternions – in WebGL, Please!

This unit quaternion:

$$q_{\vec{\omega}, \theta} = \cos \frac{\theta}{2} + \sin \frac{\theta}{2} \vec{\omega}$$

Describes a 3D rotation of:

- θ degrees
- about the axis vector $\vec{\omega}$ that I will write as (x_a, y_a, z_a)

And is written as the 4-tuple

$< q_w, q_x, q_y, q_z >$

where: $q_w = \cos(\theta/2)$

$q_x = x_a * \sin(\theta/2)$

$q_y = y_a * \sin(\theta/2)$

$q_z = z_a * \sin(\theta/2)$

Multiplication of unit quaternions performs **composition** of rotations

pq does the

First rotation

Then

rotation (it seems backwards!)

- Standard operations on quaternions (4-tuples)

– Addition, subtraction, multiplications as quaternions

– Conjugation, inverse, rotations as quaternions (next class)

– Apply your rotations in a scene graph,

– Convert quaternions to **4x4 rotation matrix**:

$\text{rot}(q) =$

$$\begin{bmatrix} 1 - 2q_y^2 - 2q_z^2 & 2q_x q_y - 2q_w q_z & 2q_x q_z + 2q_w q_y & 0 \\ 2q_x q_y + 2q_w q_z & 1 - 2q_x^2 - 2q_z^2 & 2q_y q_z - 2q_w q_x & 0 \\ 2q_x q_z - 2q_w q_y & 2q_y q_z + 2q_w q_x & 1 - 2q_x^2 - 2q_y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

See Reading: Summary on pg 91

Lengyel: “Math for 3D Game Programming”

See: cuon-matrix-quat03.js

How can you TEST Quaternion code?

- A) Do simple tasks where you already know the answer
(e.g. sequences of 90° rotations)

- B) Compare to this online interactive Quaternion calculator:
<http://www.energid.com/resources/orientation-calculator/>

The Need for SLERPs

(and why we don't use them)

For smooth animation,
we want our object to:

- start at rotation q_0 at time $t = 0$
- End at rotation q_1 at time $t = 1$

- What's an expression for
the correct rotation
at time $0 \leq t \leq 1$?

"Slerp" is widely-used jargon for
"Spherical Linear intERPolation"

The Need for SLERPs

(and why we don't use them)

For smooth animation,
we want our object to:

- start at rotation q_0 at time $t = 0$
- End at rotation q_1 at time $t = 1$

- What's an expression for
the correct rotation
at time $0 \leq t \leq 1$?

• **Yikes! Let's look for
something easier...
(next class)**

"Slerp" is widely-used jargon for
"Spherical Linear intERPolation"

- Between orientations q_0 and q_1 the
object performs a smooth rotation:
- But *what* rotation? recall: ('backwards')
 - The rotation that,
when multiplied by q_0
will give us q_1
 - Thus it must be: $q_1 q_0^{-1}$
- So at time t , we want to
 - Start with q_0
 - Then apply a fraction t of $q_1 q_0^{-1}$
 - Which would be $(q_1 q_0^{-1})^t$
- So the correctly animated rotation is
 $\text{Slerp}(q_0, q_1, t) = (q_1 q_0^{-1})^t q_0$



Further reading

- Eric Lengyel, “Math for 3D Game Pgms”
Chap 4.6 (on Canvas: Project B Reading)
- Jonathan Blow, [Hacking Quaternions](#)
- Jonathan Blow, [Understanding Slerp, Then Not Using It](#)
- Ken Shoemake, “Animating Rotation with Quaternion Curves”, Computer Graphics, Volume 19, Number 3, 1985
 - Original SIGGRAPH paper on quaternions for animation
 - Appendices include formulae for converting between quaternions, rotation matrices, and Euler angles

End.

END

Quaternion multiplication

(Why? 3D rotation!)

- This table, plus the normal rules of associative and distributive properties
- is enough to let you compute the product of any two quaternions

Quaternion multiplication				
x	1	i	j	k
1	1	<i>i</i>	<i>j</i>	<i>k</i>
<i>i</i>	<i>i</i>	-1	<i>k</i>	- <i>j</i>
<i>j</i>	<i>j</i>	- <i>k</i>	-1	<i>i</i>
<i>k</i>	<i>k</i>	<i>j</i>	- <i>i</i>	-1

ASIDE: Why is Euler's formula true?

Taylor series for e^x :

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Taylor series for sin and cos:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

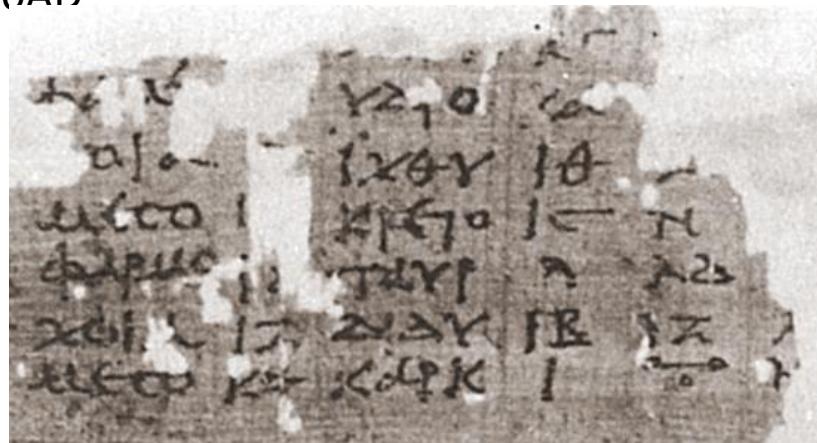
Just plug in the values:

$$\begin{aligned} e^{i\phi} &= 1 + i\phi + \frac{(i\phi)^2}{2!} + \frac{(i\phi)^3}{3!} + \dots \\ &= 1 + i\phi + \frac{i^2\phi^2}{2!} + \frac{i^3\phi^3}{3!} + \dots \\ &= 1 + i\phi - \frac{\phi^2}{2!} - i\frac{\phi^3}{3!} + \dots \\ &= \left(1 - \frac{\phi^2}{2!} + \frac{\phi^4}{4!} + \dots \right) \\ &\quad + i \left(\phi - \frac{\phi^3}{3!} + \frac{\phi^5}{5!} + \dots \right) \\ &= \cos \phi + i \sin \phi \end{aligned}$$

Development of number systems

BRIEF HISTORICAL SUMMARY:

- Imaginary numbers, complex numbers:
See Gerolamo Cardano, circa 1545
- The Renaissance shock;
earlier math was stunted in Europe;
stalled for approx. 1,000 years
- Took an extra 1500+ years
to absorb negative numbers and
other middle-eastern imports
 - Indians and Chinese developed negative numbers independently somewhere in the 200BC - 400AD range
 - Spread from India to the Islamic world in the 8th century
 - Strengthened, spread from there to Europe
 - Largely accepted worldwide by 17th century
 - Although still resisted by some European mathematicians as late as the 18th century



Solution to quadratic equations

- The quadratic formula
 - Equation of a parabola: plot (x,y) pairs for
 $ax^2 + bx + c = y$
 - Thus quadratic ‘roots’ at $y==0$ defines the x values where the plotted parabola intersects the $y=0$ line.
 - But what do we do if the plotted parabola **doesn’t intersect $y=0$?**
- Two possible interpretations
 - a) Those quadratic special cases **don’t have any roots** and therefore negative numbers simply **don’t have any square roots**, or
 - b) Negative numbers **DO** have square roots, but they’re a weird & different kind of number.
- At first, (a) won...

$$ax^2 + bx + c = 0$$
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

What about cubics?

Cubics are more complicated

- Also have **closed-form solutions** for their roots, but they're messy
- Cardano published solution formulas for the different cases in 1545
- His solution formula for

$$x^3 = px + q$$

forced CUBE roots

of negative numbers!

(He used the archaic form shown above, rather than

$$x^3 + px + q = 0$$

because people still weren't sure they believed negative numbers were valid!

Old way of writing the equation kept p and q values greater than zero.

Cardano's Solution for
 $x^3 = px + q$

was:

$$x = \sqrt[3]{\frac{1}{2}q + w} + \sqrt[3]{\frac{1}{2}q - w}$$

where:

$$w = \sqrt{\left(\frac{1}{2}q\right)^2 - \left(\frac{1}{3}p\right)^2}$$

(Trust your ancestors' proof of it)

What about cubics?

Here's the problem.
The polynomial:

$$x^3 = 15x + 4$$

has the solution:

$$x = 4$$

But Cardano's formula gives:

$$x = \sqrt[3]{2 + \sqrt{-121}} + \sqrt[3]{2 - \sqrt{-121}}$$

ouch.
Is that actually “4”?

What about cubics?

Here's the problem:
The polynomial

THUS

we must have imaginary numbers
to solve cubics,
even for some non-imaginary roots!

formula gives:

$$\sqrt{2 + \sqrt{-121}} + \sqrt[3]{2 - \sqrt{-121}}$$

has the solution:

ouch.

Is that actually “4”?

$$x = 4$$

OPTIONAL : Try it Yourself!

Bombelli showed (1572) that if you assume

$$\sqrt[3]{2 + \sqrt{-121}} = a + bi$$

For some a, b , then

$$\sqrt[3]{2 - \sqrt{-121}} = a - bi$$

So:

$$x = 4 = a + bi + a - bi = 2a$$
$$a = 2$$

- So we NEED **imaginary numbers** to solve for cubics, even for some non-imaginary roots!
-

- And indeed, if we plug in $a = 2, b = 1$, we get
$$(a + bi)^3 = (2 + i)^3$$
$$= 2 + 11i$$
$$= 2 + 11\sqrt{-1}$$
$$= 2 + \sqrt{-121}$$
$$(a - bi)^3 = 2 - \sqrt{-121}$$

Quaternions Part 2: Smooth Interpolation

Jack Tumblin

Northwestern Univ COMP_SCI 351-1

Fall 2021

From Ian Horswill 2015, Northwestern Univ.
Heavily Modified by Jack Tumblin 2016-2020

RECALL: How to Use Quaternions

- A quaternion is a complex number with a **real** part & **3** imaginary parts:

$$q = \textcolor{teal}{a} + (b\mathbf{i} + c\mathbf{j} + d\mathbf{k})$$

- A unit quaternion:

$$U_q = \frac{q}{\|q\|}$$

describes a 3D rotation

(Just as
a unit-length complex number
($a + bi$)
describes a 2D rotation)

- This unit quaternion:

$$q_{\vec{\omega}, \theta} = \cos \frac{\theta}{2} + \sin \frac{\theta}{2} \vec{\omega}$$

Performs a rotation of:

- θ degrees around the
- 3D axis vector $\vec{\omega}$
(see next slide for details)

- **Multiply** unit quaternions to **combine** their rotations:
 - (pq) is equivalent to
 - **First** rotate coord. axes by q ,
 - **Then** rotate coord. axes by p .

How to Use **Quaternions** – in WebGL, please!

This unit quaternion:

$$q_{\vec{\omega}, \theta} = \cos \frac{\theta}{2} + \sin \frac{\theta}{2} \vec{\omega}$$

Describes a 3D rotation of:

- θ degrees
- about the axis vector $\vec{\omega}$
that I will write as (xa,ya,za)

And is written as the 4-tuple

< qw, qx, qy, qz >

where: $qw = \cos(\theta/2)$,
 $qx = xa * \sin(\theta/2)$,
 $qy = ya * \sin(\theta/2)$,
 $qz = za * \sin(\theta/2)$

Multiplication of unit quaternions
performs **composition** of their rotations
 pq does the equivalent of
First rotating by q ,
Then rotating by p ('it seems backwards')

- **Store** your rotations as quaternions (4-tuples)
- **Compose** your rotations as quaternions
- **Interpolate** rotations as quaternions
- **To apply** your rotations in a scene graph,
convert quaternions to 4x4 rotation matrix:

Rot(q) =

$$\begin{bmatrix} 1 - 2qy^2 - 2qz^2 & 2qx qy - 2qw qz & 2qx qz + 2qw qy & 0 \\ 2qx qy + 2qw qz & 1 - 2qx^2 - 2qz^2 & 2qy qz - 2qw qx & 0 \\ 2qx qz - 2qw qy & 2qy qz + 2qw qx & 1 - 2qx^2 - 2qy^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

See Reading: Summary on pg 91

Lengyel: "Math for 3D Game Programming"

How to Use Quaternions – in WebGL, Please!

This unit quaternion:

$$q_{\vec{\omega}, \theta} = \cos \frac{\theta}{2} + \sin \frac{\theta}{2} \vec{\omega}$$

Describes a 3D rotation of:

- θ degrees
- about the axis vector $\vec{\omega}$ that I will write as (x_a, y_a, z_a)

And is written as the 4-tuple

$< q_w, q_x, q_y, q_z >$

where: $q_w = \cos(\theta/2)$

$q_x = x_a * \sin(\theta/2)$

$q_y = y_a * \sin(\theta/2)$

$q_z = z_a * \sin(\theta/2)$

Multiplication of unit quaternions performs **composition** of rotations

pq does the

First rotation

Then

rotation (it seems backwards!)

- Standard API uses quaternions (4-tuples)

• Convert 3D rotations as quaternions

• Convert quaternions to 3D rotations as quaternions

• Apply your rotations in a scene graph,

Convert **quaternions to 4x4 rotation matrix**:

$\text{Rot}(q) =$

$$\begin{bmatrix} 1 - 2q_y^2 - 2q_z^2 & 2q_x q_y - 2q_w q_z & 2q_x q_z + 2q_w q_y & 0 \\ 2q_x q_y + 2q_w q_z & 1 - 2q_x^2 - 2q_z^2 & 2q_y q_z - 2q_w q_x & 0 \\ 2q_x q_z - 2q_w q_y & 2q_y q_z + 2q_w q_x & 1 - 2q_x^2 - 2q_y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

See: cuon-matrix-quat03.js

See Reading: Summary on pg 91

Lengyel: "Math for 3D Game Programming"

RECALL: Why is this good?

- No gimbal lock
- That 4th degree of freedom and normalizing in quaternions:
$$q = \textcolor{teal}{a} + (\textcolor{red}{bi} + \textcolor{red}{cj} + \textcolor{red}{dk})$$
Lets it represent 3D rotations without discontinuities (!)
- (Similarly, the 2nd degree of freedom and normalizing in ordinary complex numbers ($\textcolor{teal}{a}+\textcolor{red}{bi}$) Lets it represent 2D rotations without discontinuities.)

- Composition: easy
 - Multiplication = combine
 - pq does q then p ('backwards')
 - Exponentiation = repeat
 - q^n applies q exactly n times, thus
 - $\sqrt{q} = q^{\frac{1}{2}}$ does q halfway...
- Interpolation: ~easy
 - q^{frac} for $0 \leq frac \leq 1$, for a fraction of q rotation

The Need for SLERPs

(and why we don't use them)

For smooth animation,
we want our object to:

- start at rotation q_0 at time $t = 0$
- End at rotation q_1 at time $t = 1$

- What's an expression for
the correct rotation
at time $0 \leq t \leq 1$?

"Slerp" is widely-used jargon for
"Spherical Linear intERPolation"

The Need for SLERPs

(and why we don't use them)

For smooth animation,
we want our object to:

- start at rotation q_0 at time $t = 0$
- End at rotation q_1 at time $t = 1$

- What's an expression for
the correct rotation
at time $0 \leq t \leq 1$?
We can figure it out ... →→→

• **Yikes! Let's look for
something easier
than this!**



"Slerp" is widely-used jargon for
"Spherical Linear intERPolation"

- Between orientations q_0 and q_1 the
object performs a smooth rotation:
- But *what* rotation? recall: ('backwards')
 - The rotation that,
when multiplied by q_0
will give us q_1
 - Thus it must be: $q_1 q_0^{-1}$
- So at time t , we want to
 - Start with q_0
 - Then apply a fraction t of $q_1 q_0^{-1}$
 - Which would be $(q_1 q_0^{-1})^t$
- So the correctly animated rotation is

$$\text{Slerp}(q_0, q_1, t) = (q_1 q_0^{-1})^t q_0$$

But computing

$$(q_1 q_0^{-1})^t$$

is **SUCH** a pain!

In practice, almost nobody actually implements SLERPs – computing fractional exponentials eats up too much computing time. Let's see why...

(1) ?What's a quaternion's exponent (\mathbf{q}^t)?

Step 1: find e^q (the natural exponent of quaternion q)

- As always, define $q = q_s + \vec{q}_v$
 - q_s - the scalar, real part <qw>
 - \vec{q}_v - the vector, imag part <qa,qb,qc>
- Put it in exponential, simplify:
$$e^q = e^{q_s + \vec{q}_v} = (e^{q_s})(e^{\vec{q}_v})$$
- Great! A scalar. Wait--what's $(e^{\vec{q}_v})$?
- Solution: Apply the Taylor series:

$$e^{\vec{q}_v} = \sum_{n=0}^{\infty} \frac{(\vec{q}_v)^n}{n!}$$

- The algebra is uglier than Euler's formula, but end result is similar:

$$e^{\vec{q}_v} = \cos(\|\vec{q}_v\|) + \frac{\vec{q}_v}{\|\vec{q}_v\|} \sin(\|\vec{q}_v\|)$$

Once again, we find:
-- a cosine sets the real part, and
-- a sine sets the imaginary part.

(2) ?What's a quaternion's exponent (\mathbf{q}^t)?

Step 2: And find $\ln(\mathbf{q})$ (the natural logarithm of quaternion \mathbf{q})

$$\begin{aligned}\ln \mathbf{q} &= \ln(q_s + \vec{q}_v) \\ &= \ln\|\mathbf{q}\| + \frac{\vec{q}_v}{\|\vec{q}_v\|} \cos^{-1} \frac{q_s}{\|\vec{q}_v\|}\end{aligned}$$

**Step 3: Apply both to \mathbf{q}^t to make a computable expression:
thus we could compute SLERPs using:**

$$\mathbf{q}^t = \exp(\ln(\mathbf{q}^t)) = \exp(t \cdot \ln(\mathbf{q})) = e^{t \ln \mathbf{q}} = \mathbf{q}^t$$

(yuk) **Wildly impractical:**

far too many expensive transcendentals!

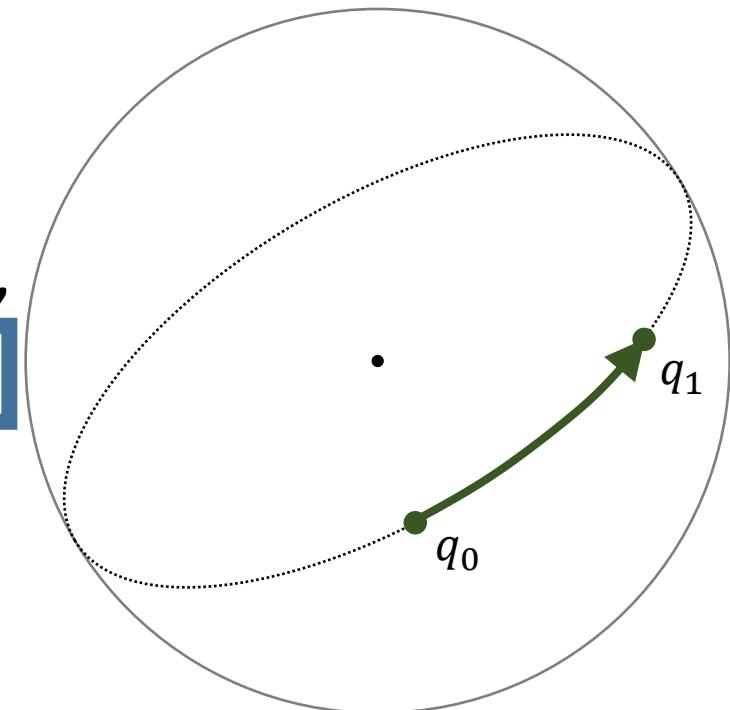
lets find a simpler approximation...

What are we trying to Approximate?

- Slerp's primary features are that, as t changes for $0 \leq t \leq 1$
 - Starts at q_0 , moves smoothly to q_1 along
 - a **minimum torque path:**
 - The path that requires the least possible energy
 - (Trust the physicists: they've proven this is true.
GREAT for animation! "***It Just Looks Right!***")
 - It changes the 3D orientation at a **constant angular speed**
 - → Rotation won't 'speed up' or 'slow down' during the interpolation
 - Let's plot **what that path looks like...**

What are we trying to Approximate?

- In 3D, suppose we rotate a **single point on the unit sphere** from orientation q_0 to q_1
- If it's on the minimum-torque path, then it moves along a **great circle:**
 - AKA a “geodesic” on the sphere
 - AKA the shortest-length path
- And every ‘great circle’ lies in a **plane thru the sphere’s center...**



Unit Quaternions: ALSO on a Sphere...

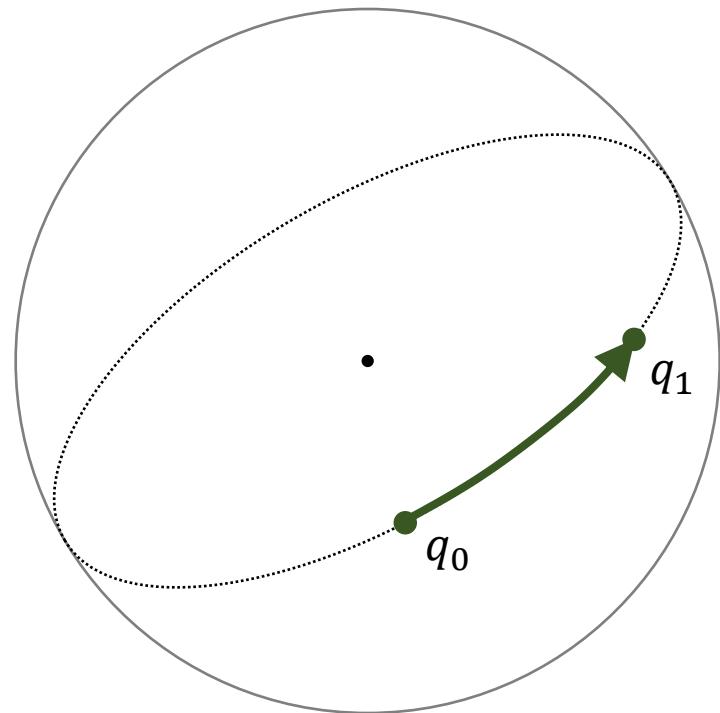
- As we're representing rotations using **unit** quaternions:

$$a^2 + b^2 + c^2 + d^2 = 1$$

- We can think of the unit quaternion as a point in a **4-dimensional space** (a, b, c, d)

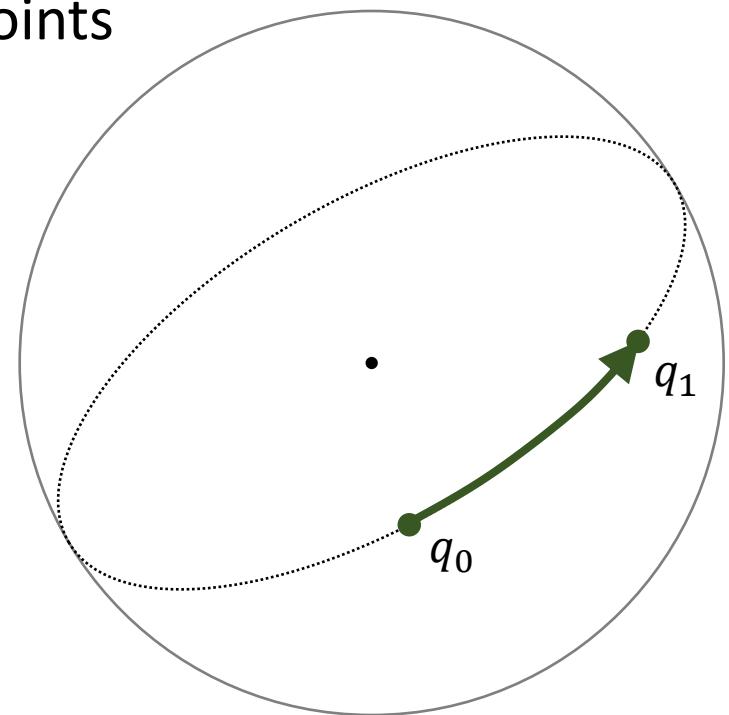
- Then all the unit quaternions will lie on a **hypersphere** in that 4-dimensional space.

- Does SLERP also follow a 'Great Circle' in 4D? YES!



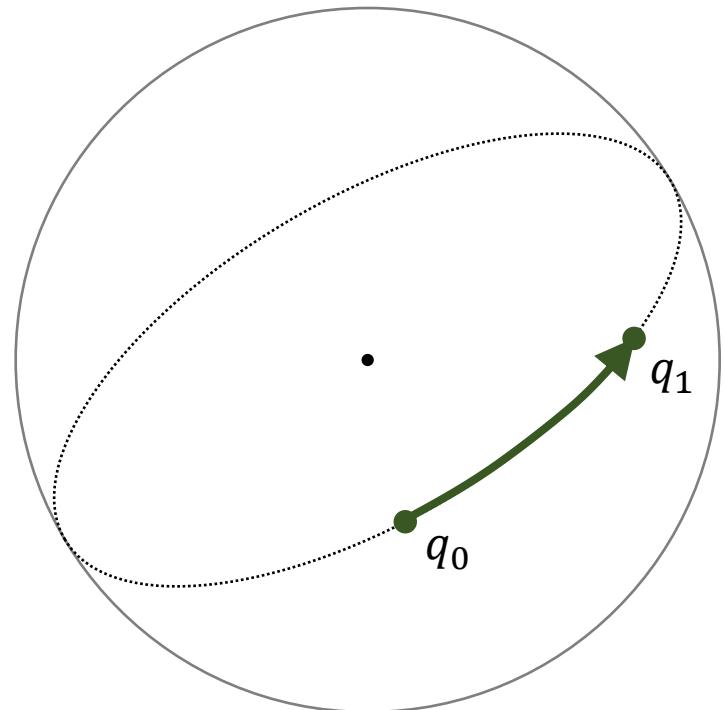
SLERP: a Great Circle on 4D Sphere!

- It can be proven that
 - The path SLERP takes between two points also lies on a **great circle** of the 4D hypersphere
 - It also traces through that path at **constant angular velocity**
- That means the 4D circular path that we want to follow is on a **plane!**



Why is that plane important?

- Any point on a plane can be represented as a **weighted sum** of two vectors in the plane, plus the origin point.
- We *already have* two vectors in the plane: q_0 and q_1
- So we know Slerp(q_0, q_1, t) is always the origin point PLUS a **weighted sum of q_0 and q_1**



SLERP: Now look at that plane...

- Since the Slerp value is co-planar with q_0 and q_1 , we can show that

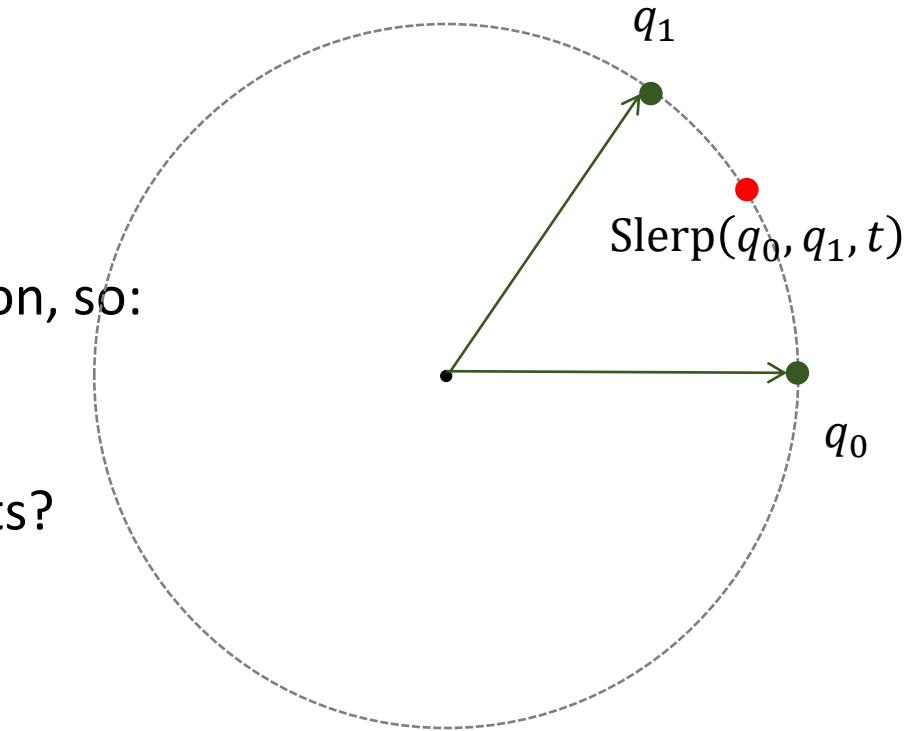
$$\text{Slerp}(q_0, q_1, t) = \textcolor{teal}{a}q_0 + \textcolor{red}{b}q_1 + (0,0,0)$$

For some $\textcolor{teal}{a}$ and $\textcolor{red}{b}$
that we don't yet know.

- We also know that it's a unit quaternion, so:

$$\|\textcolor{red}{a}q_0 + \textcolor{teal}{b}q_1\| = 1$$

- How shall we compute the a, b , weights?



SLERP: Blow's construction

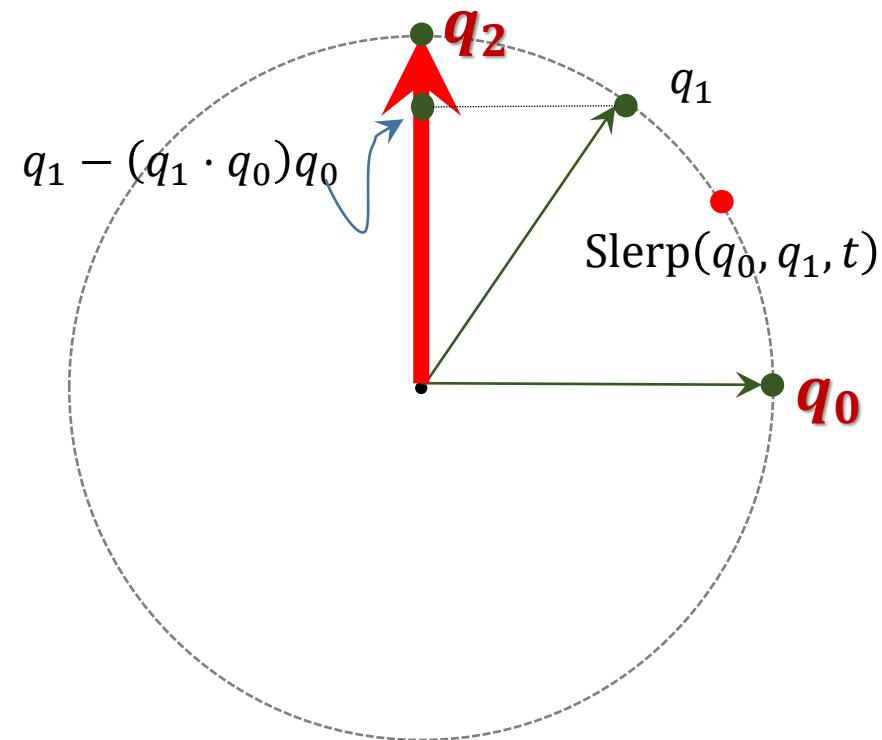
- Jonathan Blow (author of *Braid*, among other great games) has an easy to understand construction for this:

$$q_2 = \frac{q_1 - (q_1 \cdot q_0)q_0}{\|q_1 - (q_1 \cdot q_0)q_0\|}$$

- Start out by using Gram-Schmidt Orthogonalization to construct a new quaternion: (puzzled?
don't worry - just do this)

$$q_2 = \frac{q_1 - (q_1 \cdot q_0)q_0}{\|q_1 - (q_1 \cdot q_0)q_0\|}$$

- This new unit quaternion q_2 is perpendicular to q_0 and
- Think of q_0 and q_2 as the x,y axes of the plane that contains the SLERP we want...



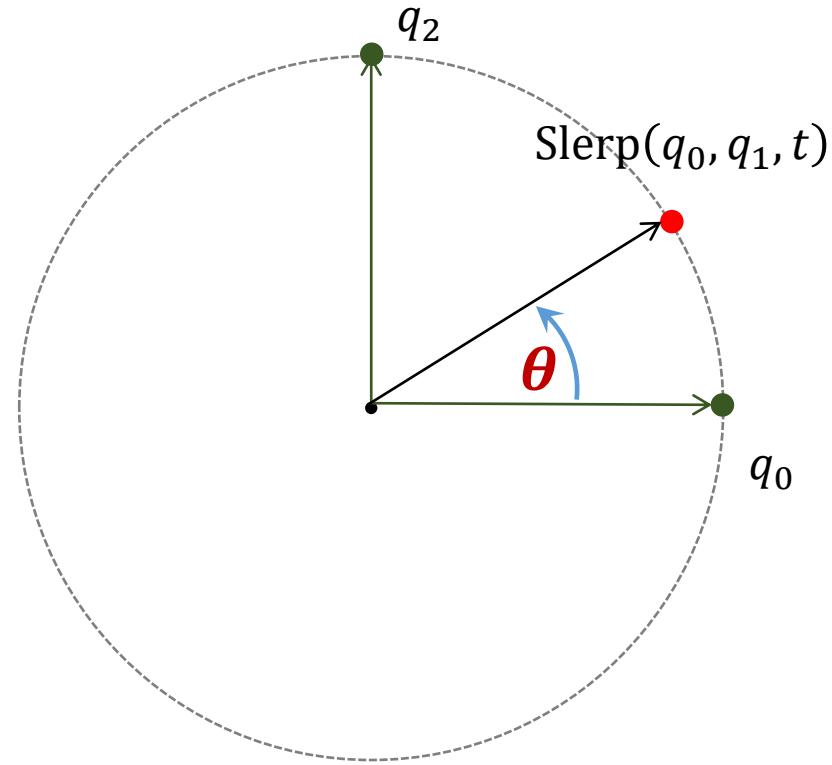
SLERP: Blow's construction

- Now we can construct any point on the SLERP's circular path with sines and cosines ---

Slerp(q_0, q_1, t)

$$= q_0 \cos \theta + q_2 \sin \theta$$

All we have to do is find θ ...

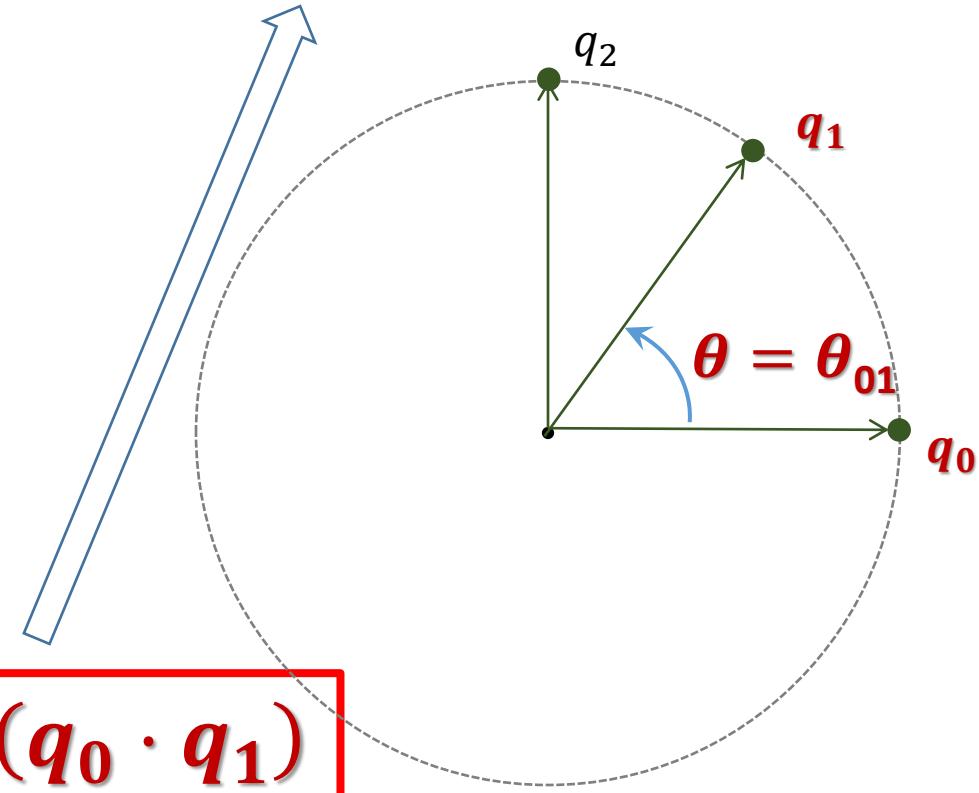


SLERP: Blow's construction

- How do we find θ ?
 - Find the angle θ_{01} from the vector q_0 to q_1 from their dot product
$$q_0 \cdot q_1 = \cos \theta_{01}$$
 - Thus
$$\theta_{01} = \cos^{-1}(q_0 \cdot q_1)$$
 - Use this to find θ :
 - When $t=0$ we want $\theta = 0$;
 - When $t=1$ we want $\theta = \theta_{01}$;
so...

And then

$$\begin{aligned}\text{Slerp}(q_0, q_1, t) \\ = q_0 \cos \theta + q_1 \sin \theta\end{aligned}$$



$$\theta = t \cos^{-1}(q_0 \cdot q_1)$$

Exact SLERP: Blow's SUMMARY

- Make unit quaternion \mathbf{q}_2 :

- co-planar with \mathbf{q}_1 and \mathbf{q}_2
- perpendicular to \mathbf{q}_0

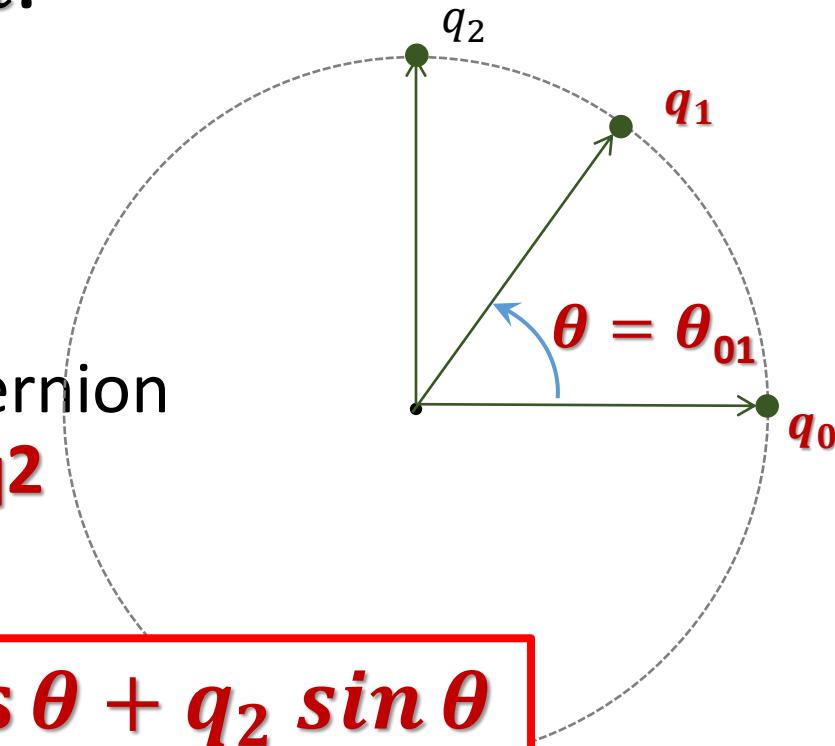
$$\mathbf{q}_2 = \frac{\mathbf{q}_1 - (\mathbf{q}_1 \cdot \mathbf{q}_0)\mathbf{q}_0}{\|\mathbf{q}_1 - (\mathbf{q}_1 \cdot \mathbf{q}_0)\mathbf{q}_0\|}$$

- Find great-circle angle θ from t :

$$\theta = t \cos^{-1}(\mathbf{q}_0 \cdot \mathbf{q}_1)$$

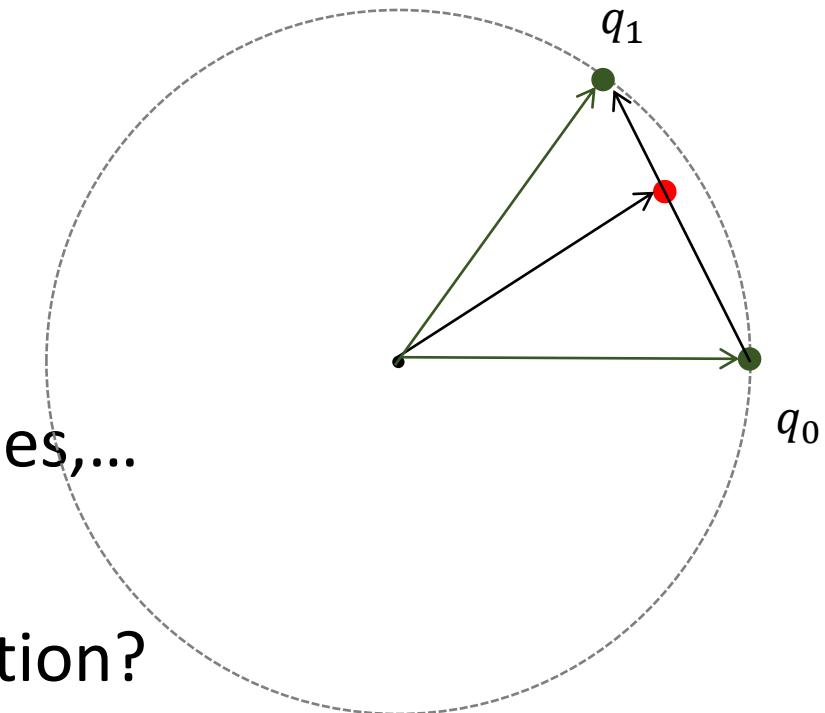
- Create your interpolated quaternion by a weighted sum of \mathbf{q}_0 and \mathbf{q}_2

$$\text{Slerp}(\mathbf{q}_0, \mathbf{q}_1, t) = \mathbf{q}_0 \cos \theta + \mathbf{q}_2 \sin \theta$$



Linear interpolation

- Well, OK.
But that's ***still*** rather expensive:
finding \mathbf{q}_2 and using sines, cosines,...



Why not a simple ***linear*** interpolation?

$$t\mathbf{q}_1 + (1 - t)\mathbf{q}_0$$

- The result point isn't quite on the circle, &
- the result is no longer a unit quaternion,
- And thus the result isn't yet ready
to convert to a rotation matrix, but

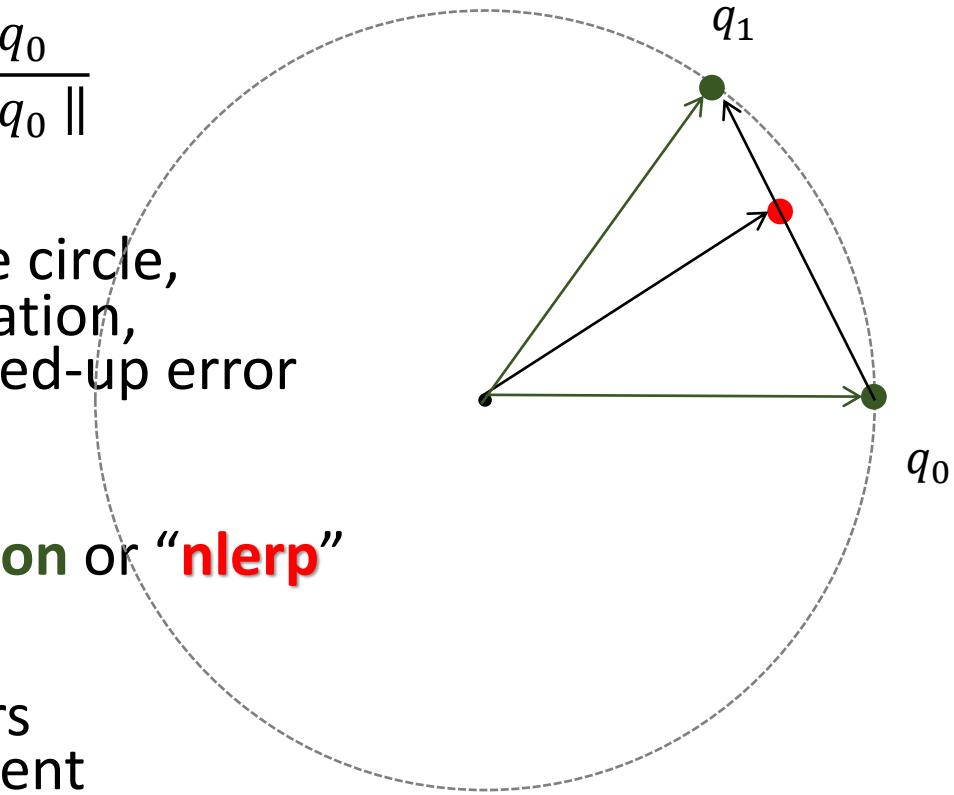
Normalized linear interpolation:

- But if we just normalize its magnitude?

$$\frac{tq_1 + (1 - t)q_0}{\|tq_1 + (1 - t)q_0\|}$$

- **YES!** Then we're back on the circle, and back to being a valid rotation, but with small, mid-path speed-up error

- Widely used! Known as **normalized linear interpolation** or “**nlerp**”
- MUCH faster to compute
- You'll never see the tiny errors if q_1 and q_0 aren't very different
- **For animation, J. Blow recommends nlerp in small steps instead of full SLERP!**



ASIDE: Shoemake/Davis Method for Exact SLERP

Jonathan Blow's formula for the exact (but too-expensive)

$$\text{Slerp}(q_0, q_1, t) = q_0 \cos \theta + q_1 \sin \theta$$

where:

- $q_2 = \frac{q_1 - (q_0 \cdot q_1)q_0}{\|q_1 - (q_0 \cdot q_1)q_0\|}$ and
- $\theta = \cos^{-1}(q_0 \cdot q_1)$

isn't the ONLY formulation for exact SLERP.

From Ken Shoemake's pioneering paper on quaternions for animation:

$$\text{Slerp}(q_0, q_1, t) = \frac{\sin((1-t)\theta)}{\sin \theta} q_0 + \frac{\sin t\theta}{\sin \theta} q_1$$

This 'Shoemake/Davis' exact SLERP is less obvious, but faster (to compute) than Blow's exact SLERP. (See Shoemake's paper for the derivation).

Further reading

- Eric Lengyel, “Math for 3D Game Programs”
Chap 4.6 (on Canvas: Project B Reading)
- Jonathan Blow, [Hacking Quaternions](#)
- Jonathan Blow, [Understanding Slerp, Then Not Using It](#)
- Ken Shoemake, “Animating Rotation with Quaternion Curves”, Computer Graphics, Volume 19, Number 3, 1985
 - The pioneering SIGGRAPH paper that introduced quaternions for animation to the computer-graphics community.
 - Appendices include formulae for converting between quaternions, rotation matrices, and Euler angles

END

Essential GLSL-ES: A Quick Tour

VITAL READING:

**Textbook, Chapter 6,
and**

“The GLSL-ES Shading Language”

(Version 1.0.17 standards document)

J.Tumblin-Modified SLIDES:

and a few from

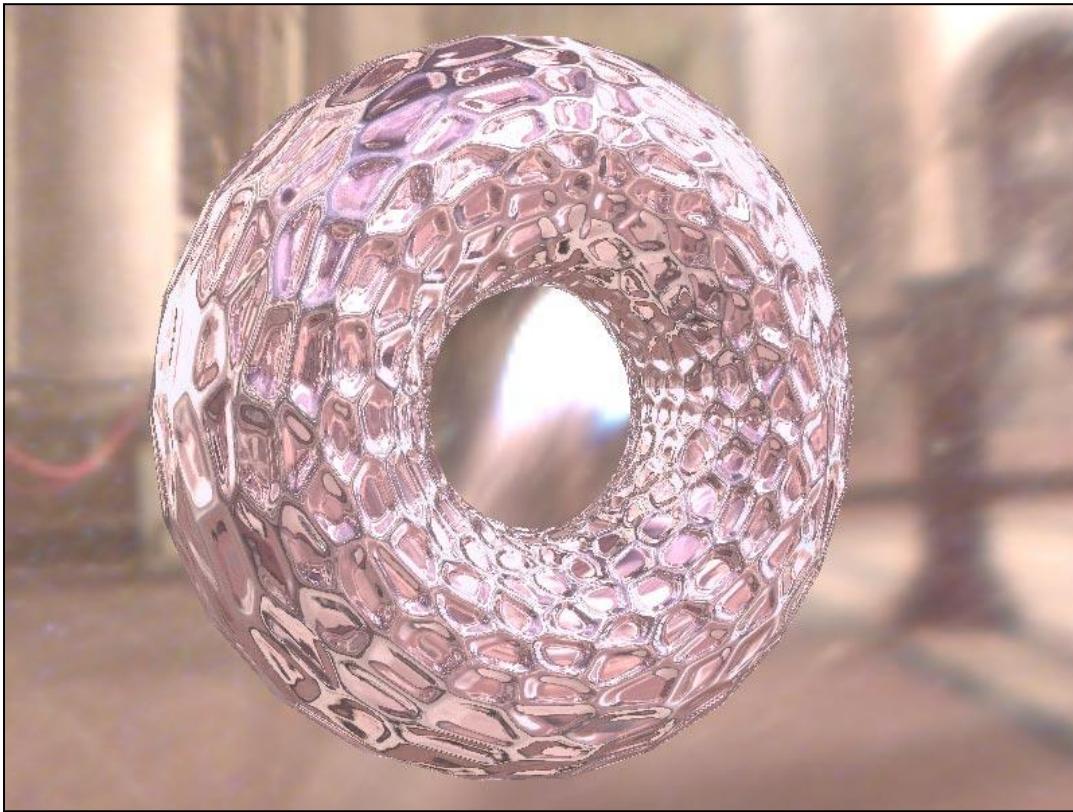
UCSC CMPS 160 Introduction to Computer Graphics
Lab 4: Shaders and RTI Rendering

<http://classes.soe.ucsc.edu/cmpps160/Fall10/labs/lab4.html>

Why Write Shaders?

Why Write Shaders?

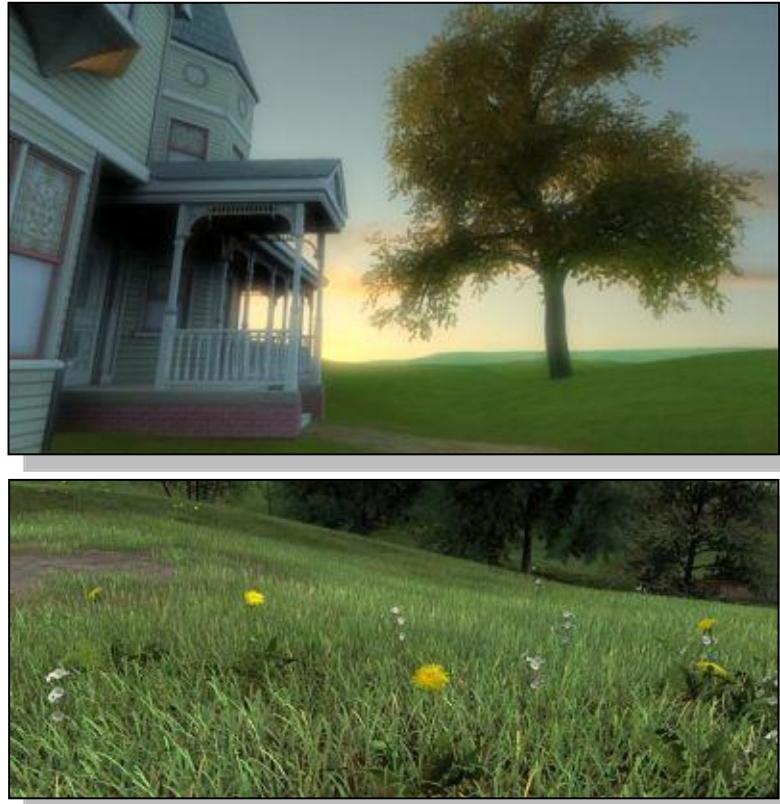
So we can do this...



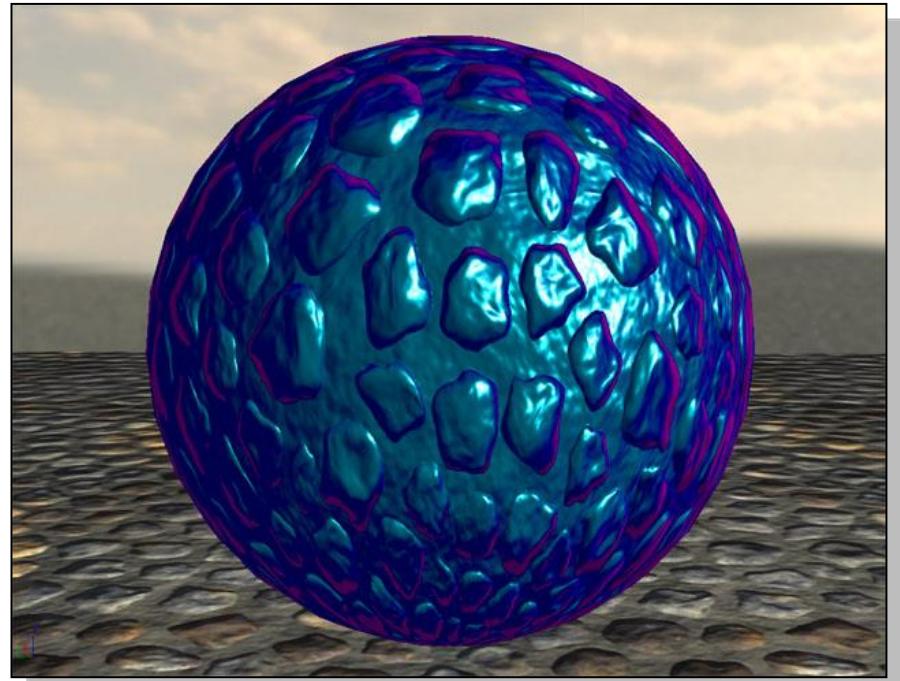
Above: Demo of Microsoft's XNA game platform
Right: Product demos by nvidia (top) and Radeon (bottom)



Why Write Shaders? and this ... and SO MUCH MORE!!



Above: Kevin Boulanger (PhD thesis, “*Real-Time Realistic Rendering of Nature Scenes with Dynamic Lighting*”, 2005)



Above: Ben Cloward (“Car paint shader”)

GLSL Pipeline ESSENTIALS: (Review)

How to Communicate with Shader Programs

Only 3 ways: (MEMORIZE THEM!) for your

JavaScript code to send data *into* the
GPU's shader programs
(and no way to retrieve that data!)

- **Uniform** parameters

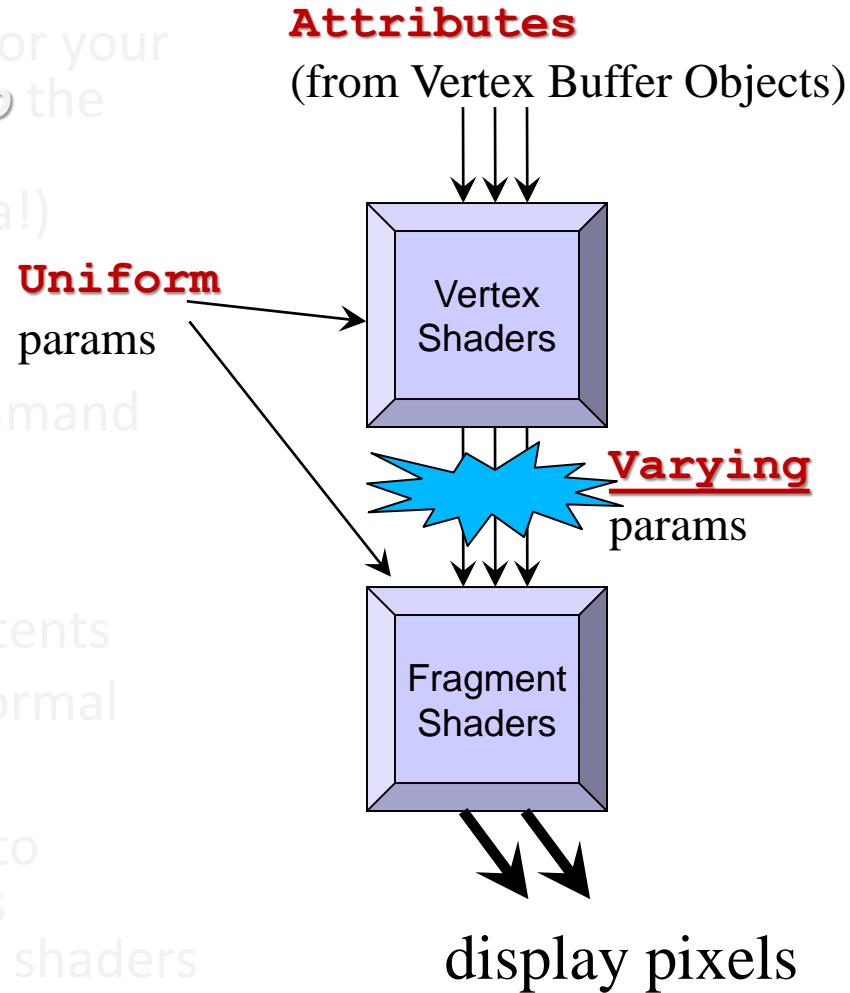
- Set before each drawing command
- Ex: ModelMatrix

- **Attribute** parameters

- Set per vertex from VBO contents
- Ex: position, color, surface normal

- **Varying** parameters

- Passed from Vertex Shaders to
~~rasterizer~~ which interpolates
per-pixel values for fragment shaders
- Ex: interpolated colors

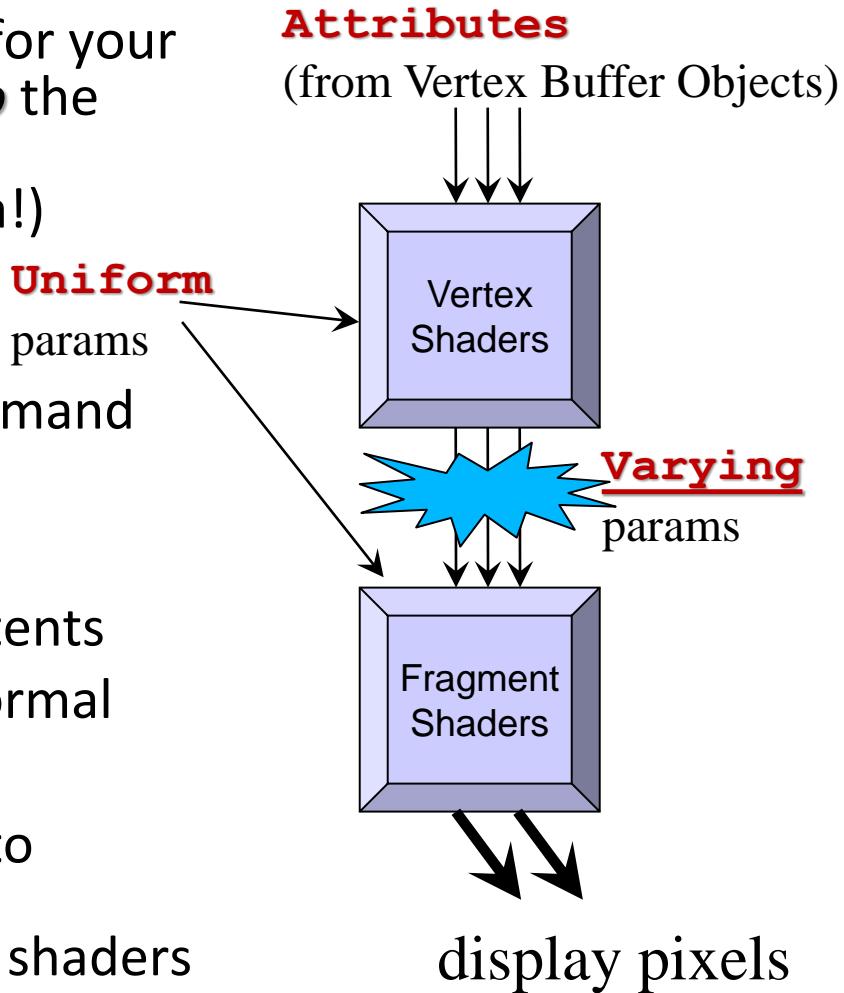


GLSL Pipeline ESSENTIALS: (Review)

How to Communicate with Shader Programs

Only 3 ways: (MEMORIZE THEM!) for your JavaScript code to send data **into** the GPU's shader programs (and no way to retrieve that data!)

- **Uniform** parameters
 - Set before each drawing command
 - Ex: ModelMatrix
- **Attribute** parameters
 - Set per vertex from VBO contents
 - Ex: position, color, surface normal
- **Varying** parameters
 - Passed from Vertex Shaders to rasterizer, which interpolates per-pixel values for fragment shaders
 - Ex: interpolated colors



GLSL Pipeline ESSENTIALS

[https://www.khronos.org/opengl/wiki/Built-in_Variable_\(GLSL\)](https://www.khronos.org/opengl/wiki/Built-in_Variable_(GLSL))

- Each GLSL **Vertex Shader** finds all values for one **vertex**:
(3D transformed position, color, normals, depth, texture address, etc)

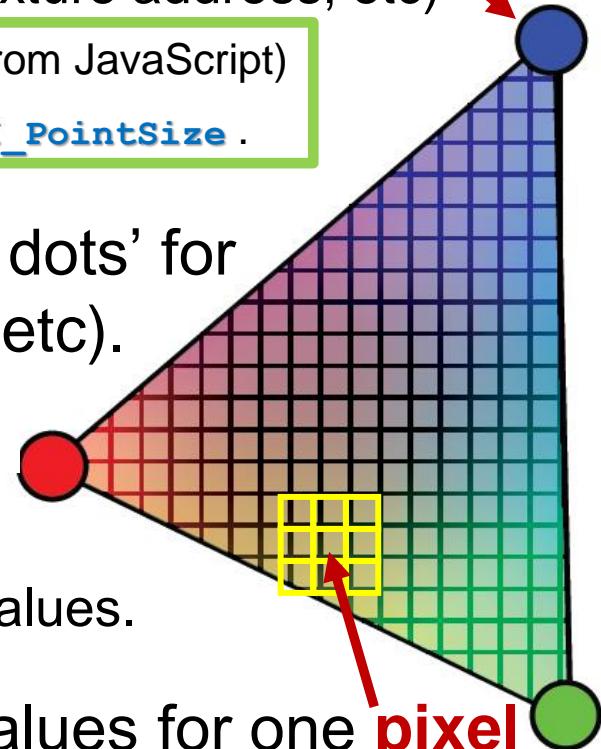
- **Inputs:** '`attribute`' values (from VBO) & '`uniform`' vars (from JavaScript)
 - **Outputs:** all '`varying`' vars; built-in vars `gl_Position` & `gl_PointSize`.

- GPU '**Rasterizer**' hardware 'connects the dots' for the drawing-primitive (point, line, triangle, etc).

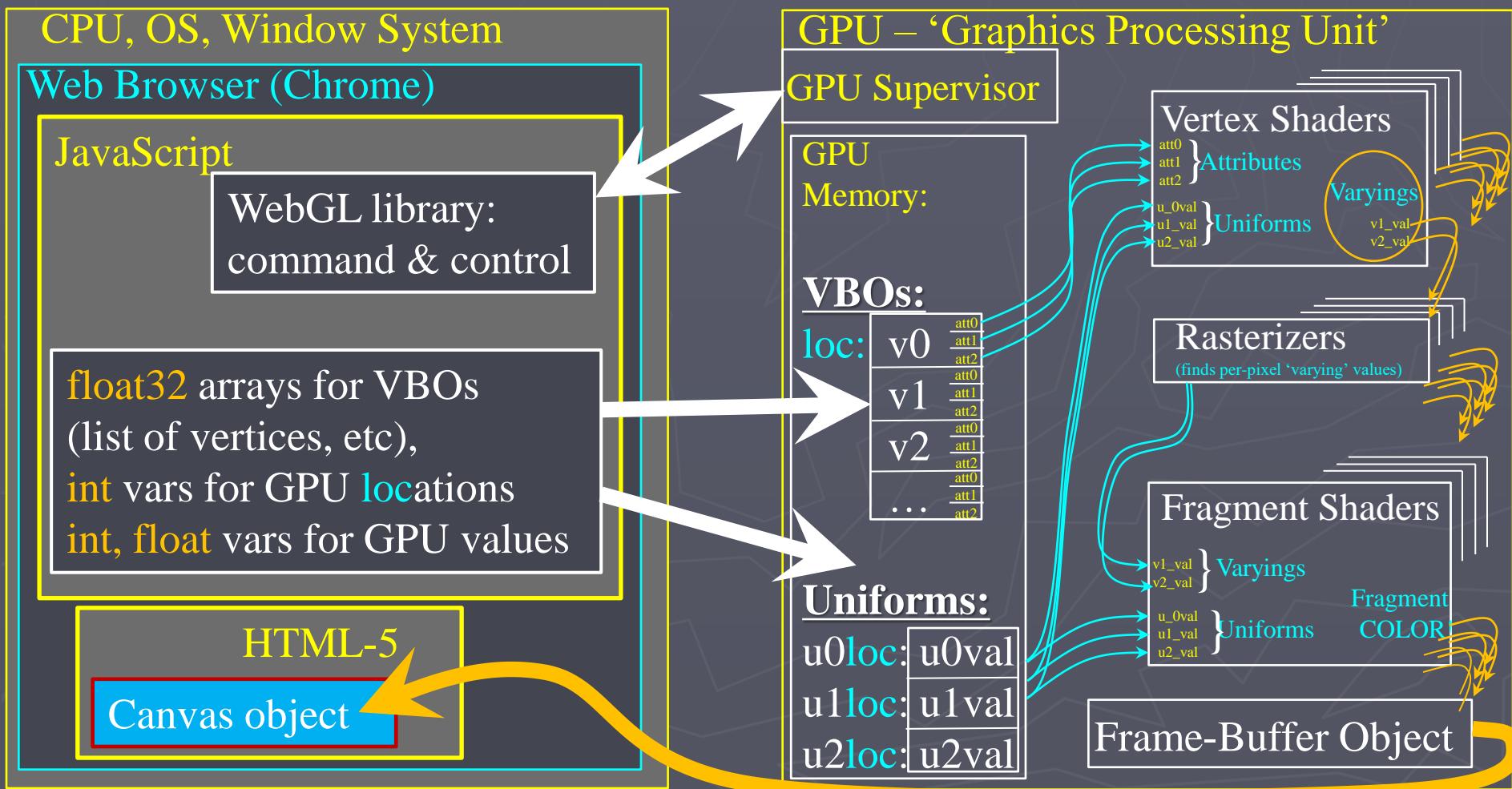
'Rasterizers' interpolate any and all '`varying`' variables set by Vertex Shader; they supply each Fragment Shader with separate, smoothly-interpolated '`varying`' values.

- Each GLSL **Fragment Shader** finds all values for one **pixel**
(color, mostly. Also (rarely) depth, textures, lighting, reflection, material...)

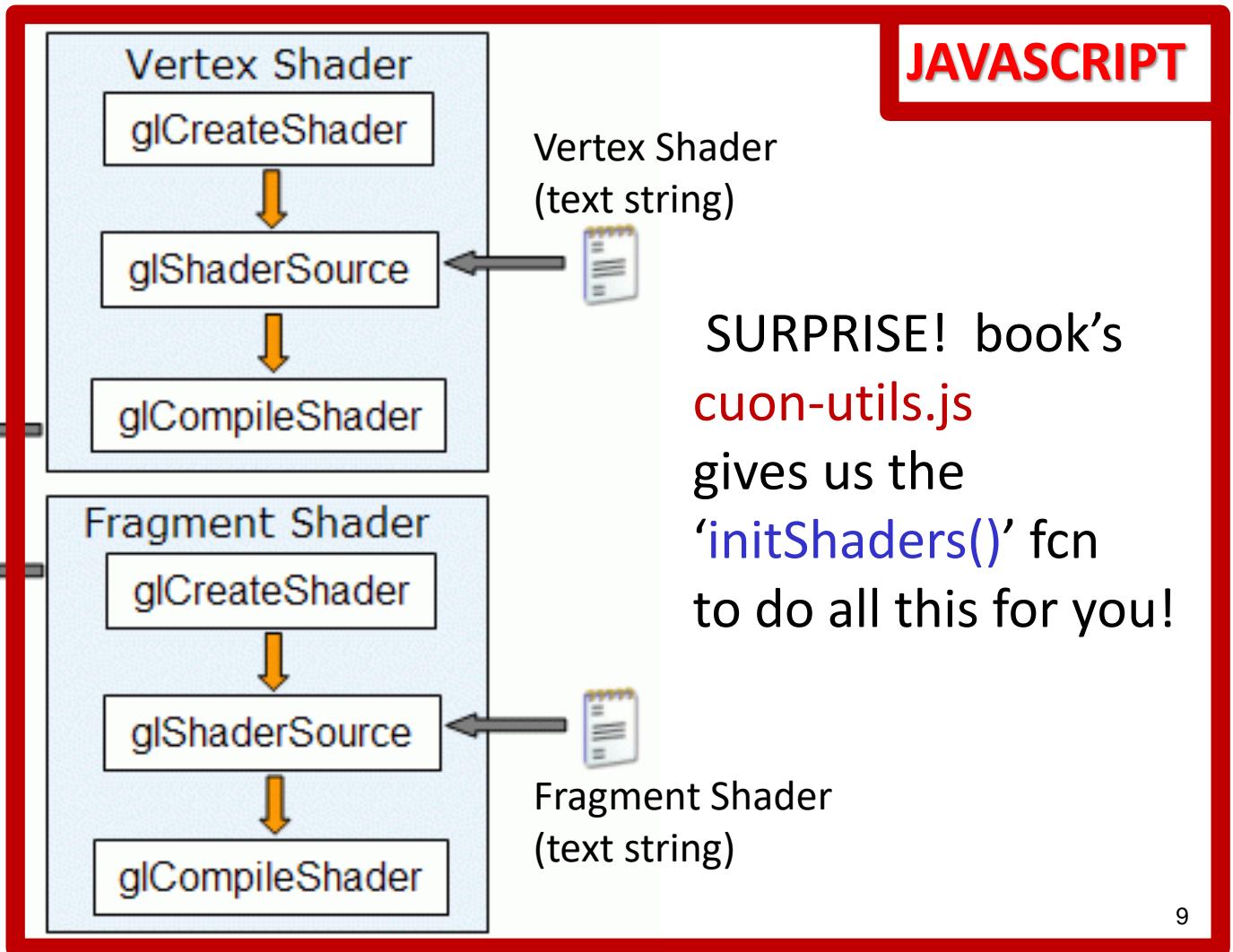
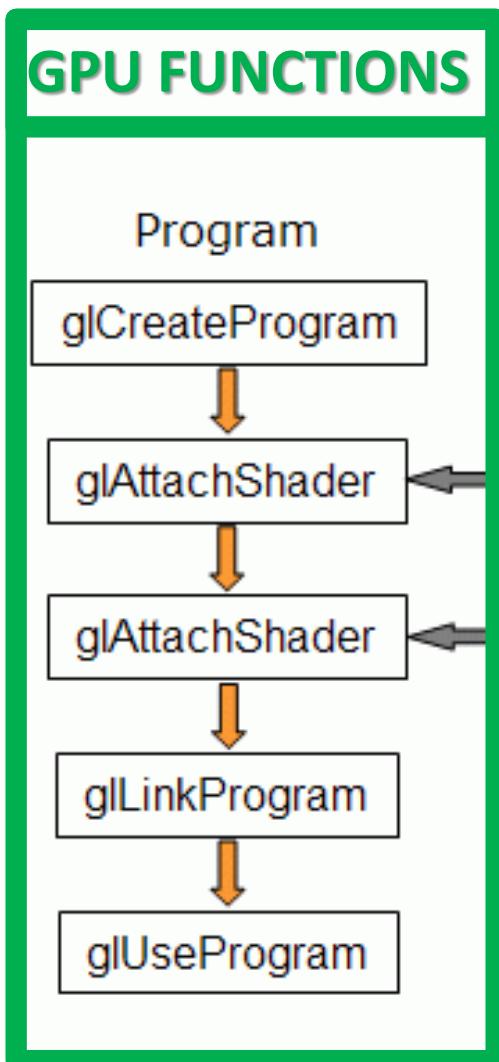
- **Inputs:** '`uniform`' vars (from JS) '`varying`' values (from Rasterizers), and also a few built-in vars: ('`gl_PointCoord`', '`gl_FragCoord`', '`gl_FrontFacing`', etc.)
 - **Outputs:** built-in vars `gl_fragColor` and `gl_fragData`



- WebGL: sends tasks to GPU
- GPU: executes tasks, flags errors
- WebGL: (optional) reads GPU flags



WebGL Setup for GLSL Shader Programs



GLSL-ES: What is it?

- “Small-device” WebGL Shading Language
- All essentials of full GLSL 2.0 (OpenGL)
- High level C-like language
- New native data types (not in C)
 - Matrices
 - Vectors
 - “Rasterizers” for GLSL **varying** vars
& “Samplers” for texture coords
(BOTH interpolate values between vertices)
- *Nearly all* WebGL internal state available through “built-in” variables (see specs)

Other Shading Languages available?

- several popular languages can describe shaders, such as:
 - **HLSL**, the *High Level Shading Language*
 - Author: Microsoft
 - DirectX 8+
 - **Cg**
 - Author: nVIDIA
 - **GLSL**, the *OpenGL/WebGL Shading Language*
 - Author: the Khronos Group, a self-sponsored group of industry affiliates (ATI, 3DLabs, etc)
 - Which Version? Isn't '1.0' out-of-date? (?2009?!)
GLSL 1.0 universally supported, polished – learn this.
GLSL 2.0 a mild super-set; 3.0 is coming. Learn 1.0 first.

GLSL-ES HIGHLIGHTS

GLSL language design strongly based on ANSI C,
simplified, with modest bits of C++ added

- There is a preprocessor -- **#include**, **#define**, etc!
- Basic types: **int**, **float**, **bool**
- Vectors and matrices are built-in and easy to use:
vec2, **mat2** = 2x2;
vec3, **mat3** = 3x3;
vec4, **mat4** = 4x4
- Supports fcn. prototypes (**declare** fcns. before definition)
- Allows (limited forms of) operator overloading.
- Texture samplers are built-in: **sampler1D**, **sampler2D**,
can sample 1D, 2D and higher-dimensional textures..

GLSL-ES: Outline

1. Variables, Data Types, & Constructors

2. Structs – bundled sets of variables

3. Operators – a rich, complete set

4. Functions

- Built-in Functions
- User-Defined Functions (strange params!)

5. Pre-Processor Directives

GLSL-ES: Outline

1. Variables, Data Types, & Constructors

2. Structs – bundled sets of variables

3. Operators –

4. Functions

READ:
“**The GLSL-ES Shading Language**” Version 1.0.17
(on CANVAS → WebGL: The Full Specifications’)

Defined Functions (strange params!)

5. Pre-Processor Directives

1. Data Types & their Constructors

GLSL Built-in Types (**struct** uses them...)

- void** -- fcns with no return value; empty fcn param list.
- bool** – 2 values: ‘true’ or ‘false’; for conditionals, etc.
- int** -- a signed integer (32 bit)
- float** -- a single floating-point scalar
- vec2**, **vec3**, **vec4** -- floating-point vectors
- mat2**, **mat3**, **mat4** -- 2×2 , 3×3 4×4 floating-point matrix
- bvec2**, **bvec3**, **bvec4** -- vectors w/ Boolean elements
- ivec2**, **ivec3**, **ivec4** -- integer vector
- sampler2D**; 2D texture handles
- samplerCube** cube-mapped texture handles.

1. Data Types & their Constructors

GLSL Built-in Types

In WebGL, vertex '**attribute**' variables in all shaders will get fed data from Vertex Buffer Objects (VBOs),
Limited to these types **only**:

- float** -- a single floating-point scalar
- vec2**, **vec3**, **vec4** -- floating-point vectors
- mat2**, **mat3**, **mat4** -- 2x2, 3x3 4x4 floating-point matrix

HINT: want **reliable** GLSL integers from float attributes?
(assume 1.0 → 1? 2.0→2? 3.0 → 3? ... !BE CAREFUL!
May need 'rounding': **myInt = int(floor(a_float + 0.5));**
SEE: http://learnwebgl.brown37.net/12_shader_language/glsl_data_types.html

1. Data Types & their Constructors

the **OTHER** GLSL Built-in Types:

- void** -- fcns with no return value; empty fcn param list.
- bool** – 2 values: ‘true’ or ‘false’; for conditionals, etc.
- int** -- a signed integer (32 bit) **(internal use only)**
- float** -- a single floating-point scalar
- vec2**, **vec3**, **vec4** -- floating-point vectors
- mat2**, **mat3**, **mat4** -- 2x2, 3x3 4x4 floating-point matrix
- bvec2**, **bvec3**, **bvec4** -- vectors w/ Boolean elements **(internal use only)**
- ivec2**, **ivec3**, **ivec4** -- integer vector
- sampler2D**; 2D texture handle
CAUTION! May need Web Server (to read files)
- samplerCube** cube-mapped texture handle.

1. Data Types & Precision: Qualifiers

GLSL Precision Qualifiers (none? → **highp**)
allow you to ‘downgrade’ (& speed up) shaders

- **highp float**: $\geq \pm/2^{62}$ Range; 16-bit precision
- **highp int**: $\geq \pm/2^{14}$ ($\pm/16,384$)
- **mediump float**: $\geq \pm/2^{14}$ Range; 10-bit precision
- **mediump int**: $\geq \pm/2^{10}$ ($\pm/1,024$)
- **lowp float**: $\geq \pm/2$ Range; 8-bit precision
- **lowp int**: $\geq \pm/2^8$ ($\pm/256$)

Set one var like this: `lowp vec3 shadowColr;`
And / or change default – put this at top of shader:
`precision mediump float; // not highp!`

1. Data Types & Precision: Qualifiers

GLSL Precision Qualifiers (none? → **highp**)

allow you to ‘downgrade’ (& speed up) shaders

- **highp float**: $\geq \pm 2^{62}$ Range; 16-bit precision
- **highp int**: $\geq \pm 2^{14}$ (**$\pm 16,384$**)
- **mediump float**: $\geq \pm 2^{14}$ Range; 10-bit precision
- **mediump int**: $\geq \pm 2^{10}$ (**$\pm 1,024$**)
- **lowp float**: $\geq \pm 2$ Range; 8-bit precision
- **lowp int**: $\geq \pm 2^8$ (**± 256**)

Set one var like this: **lowp vec3 shadowColr;**

And / or change default – put this at top of shader:

precision mediump float; // not highp!

1. Data Types & Precision: Qualifiers

GLSL Precision Qualifiers (none? → **highp**)
allow you to ‘downgrade’ (& speed up) shaders

- **highp float**: $\geq +/-2^{62}$ Range; 16-bit precision
- **highp int**: $\geq +/-2^{14}$ (**+/-16,384**)
- **mediump float**: $\geq +/-2^{14}$ Range; 10-bit precision
- **mediump int**: $\geq +/-2^{10}$ (**+/-1,024**)
- **lowp float**: $\geq +/-2$ Range; 8-bit precision
- **lowp int**: $\geq +/-2^8$ (**+/-256**)

Set one var like this: **lowp vec3 shadowColr;**
And / or change default – put this at top of shader:

precision mediump float; // not highp!

Precision Qualifiers – Ugh. Really?

- Needed only for the most-demanding shaders;
- (May vanish eventually, as GPUs improve ...)
- **?WHY?** do some shaders have this mess?:

```
// '#ifdef GL_ES \n' +
'precision mediump float;\n' +
// '#endif\n' +
```

and **?WHY?** Is it *only* in the Fragment Shader?

It's an Historical Quirk:

- Some early GPUs had no **highp** for Frag Shaders!
- (Poor design) for these, ‘default’ precision undefined
- Problem will vanish entirely in GLSL 2.0 ...

Precision Qualifiers – Ugh. Really?

SOLUTION:

-- JUST USE THIS --

```
// '#ifdef GL_ES\n' +
'precision mediump float;\n' +
// '#endif\n' +
```

and ~~?WHY? Is it only in the Fragment Shader?~~

It's an Historical Quirk:

- On some early GPUs, no **highp** for Frag Shaders!
- Poor design: for these, ‘default’ precision undefined
- Problem will vanish entirely in GLSL 2.0 ...

1. Vector Types: Swizzling & Selection

- **!Cool!** Access array elements by name, or by #, or by using [], or by selection (.) operator with

- **x, y, z, w** (3D homogeneous coord names)

- **r, g, b, a** (pixel color-channel names)

- **s, t, p, q** (texture-map coord names)

- **a[2], a.z, a.b, a.p** are all the same element!

- **Swizzling** operator

lets us select, or set, or swap any of our four vector components using their 1-letter names:

```
vec4 a; // set a==(1.0, 2.0, 2.0, 1.0):  
a.xz = vec2(1.0, 2.0); a.wy = a.xz; //Cool!
```

See: [https://www.khronos.org/opengl/wiki/Data_Type_\(GLSL\)#Swizzling](https://www.khronos.org/opengl/wiki/Data_Type_(GLSL)#Swizzling)

1. GLSL Qualifiers for variables

- Many are the same as C/C++ such as **const**, and
- GLSL defines several more to match its execution model;
- Specify a GLSL Variable that can change ...
 - Once per drawing primitive: (**uniform**)
 - Once per vertex: (**attribute**)
 - Once per fragment: (**varying**)
 - Within the shader's runtime: <no qualifier!>
(unrestricted read/write, as needed)
- Rasterizer linearly-interpolates **varying** values.
Input **varying** values from **Vertex Shaders** get smoothly interpolated between vertices to form output **varying** values for each **Fragment Shader**.²⁴

2. GLSL Arrays: sets of same-type vars

Follows limited C syntax, but **1-D** arrays ONLY.

DECLARE array – (size fixed at **compile** time!)

```
float freq[3];  int iD[2];
uniform vec4 lightPos[4];
```

INITIALIZE each array element (manually):

```
int iD[0]=5; iD[1]=2;
```

Sorry; no way to init during declaration:

```
int iD[2] = {5,2};
```

ACCESS array with the **[]** operator;

```
freq[2] = 3.5 * freq[2];
```

(checks for index errors! `index <0` or `>= array_size` is ‘illegal’)

2. GLSL Structs: Bundled, mixed types

Follows simplified C syntax:

Must first **DEFINE** the new data type:

```
struct light {  
    float intensity;  
    vec3 position;  
} lightVar;
```

Then **DECLARE** variables of that new type:

```
light lamp0; // var. of type 'light'
```

Then **ACCESS** members using the **dot** operator:

```
lamp0.intensity *= 1.2; //vec3 mpy  
lamp0.position.z = 12.3; // swizzle!
```

2. Pointers? Sadly, no...

- There are **no pointers in GLSL**;
no call-by-reference; **only** call-by-value,
BUT really, it's not so bad....

1. Matrices and Vectors are built-in, basic types:
can use them as arguments **and** return values (!):
mat3 func(mat3 a) { ...
2. C-style **struct** bundles together many vars
 - For function that returns one multi-valued var:
 - To reduce & simplify fcn arg. lists
3. PLUS a GLSL-exclusive! **in, out, inout** qualifiers
for fcn arguments can **mimic pass-by-reference...**

3. GLSL Operators: Very Few Surprises

- Operators – as ordinary as possible
(SEE **PAGE 40** in GLSL 1.0 standard for full table)
 - (), array index [], struct member and swizzler ‘.’
 - pre-fix, post-fix ++, --
 - Arithmetic: +,-,* ,/,%
 - Arithmetic assign: +=, -=, *=, /=
 - Bit-wise: >>,<<, &, |, ^, !
 - Relational/logical: >, >=, ==, !=, <=, <, &&, ||, ^^
 - Selection ? : (ugh--please use if/else instead!)
 - NOTE! No address-of, no de-reference (no pointers!)

No typecast: use constructors instead

3. GLSL Vector-Matrix Operators

- Standard C functions
 - Trigonometric
 - Arithmetic
 - Normalize, reflect, length
- Operator overloading neatly combines our vectors and matrix types. For example:

```
mat4 a; // 4x4 matrix.
```

```
vec4 b, c, d;
```

```
c = a*b; // mpy by column vector kept as a 1-D array
```

```
d = b*a; // mpy by row vector kept as a 1-D array
```

3. GLSL Vector-Matrix Operators

- Standard C functions

- Trigonometric
- Arithmetic
- Normalize, reflect

- Operator overloading for vectors and matrices

```
mat4 a; // 4x4 matrix.
```

```
vec4 b, c, d;
```

```
c = a * b; // mpy by column vector kept as a 1-D array
```

```
d = b * a; // mpy by row vector kept as a 1-D array
```

Please use **COLUMN vectors**,
not **row vectors!**

--Long-standing OpenGL convention
--Consistent == easier debug
--What I want & expect in this class

4. GLSL Functions:

wow!...WOW! So MANY Built-in Riches!

- Trigonometry
- Exponential
- Common
- Geometric
- Matrix
- Vector Relational (Comparisons)
- Texture Mapping ...

4. GLSL Functions: Built-in Riches

- **Trigonometry:** for `float, vec2, vec3, vec4...`

- `Rad = radians (deg);` // Converts degrees to radians
- `Deg = degrees (rad);` // Converts radians to degrees
- `Val = sin (rad);` // trigonometric sine function.
- `Val = cos (rad);` // trigonometric cosine function.
- `Val = tan (rad);` // trigonometric tangent function.
- `Rad = asin (x);` // Arc sine; undefined for $|x| > 1$
- `Rad = acos(x);` //Arc cosine; undefined for $|x| > 1$
- `Rad = atan (y,x);` // Arc tangent; $\tan(\text{Rad}) == y/x$

4. GLSL Functions: Built-in Riches

4. GLSL Functions: Built-in Riches

- **Common:** **for float, vec2, vec3, vec4...**

- Val = abs(x); // Returns x if $x \geq 0$, otherwise $-x$.
- Val = sign(x); // 1.0 if $x > 0$; 0.0 if $x = 0$; -1.0 if $x < 0$
- Val = floor(x); // nearest integer $\leq x$
- Val = ceil (x); // nearest integer $\geq x$
- Val = fract (x); // Returns $x - \text{floor}(x)$
- Val = mod (x, float y); // Returns $x - y * \text{floor}(x/y)$
- Val = min (x,y); // Return y if $y < x$, else return x.
- Val = max(x,y); // Return y if $x < y$, else return x.
- Val = clamp (x, minVal, maxVal);
 // Returns min (max (x, minVal), maxVal)
- Val = mix (x,y,a); // Returns $x*(1-a) + y*a$;

4. GLSL Functions: Built-in Riches

- **Geometric:** `for float, vec2, vec3, vec4...`

- float length(x); magnitude of vector x
- float distance(p0, p1); distance between p0 and p1,
- float dot(x,y); dot product of x and y
- vec3 cross(vec3 x, vec3 y); cross-product of x,y
- vec normalize(x); returns vector x with length of 1
- vec faceforward(N,I,Nref); if dot(Nref, I) < 0 return N, else -N.
- vec reflect(I, N); returns I ‘reflected’ around N
- Vec refract(I, N, float eta); Snell’s law refraction

4. GLSL Functions: Built-in Riches

- **Matrix Fcns?** **for float, vec2, vec3, vec4...**
- Just one!

```
mat matrixCompMult (mat x, mat y);  
// multiply matrix x times matrix y; find [x][y]
```

- Why so few?

Because built-in math operators (*, +, -, etc)
work for all built-in vector & matrix types...

4. GLSL Functions: Built-in Riches

- **Vector Relational (1)**: vec element pairs
for `vec2`, `vec3`, `vec4`, and also for
`bvec2`, `bvec3`, `bvec4`, `ivec2`, `ivec3`, `ivec4`.
 - `bvec lessThan(vec x, vec y);` // is each (x element)
 - `bvec lessThan(ivec x, ivec y);` // < each (y element)?
 - `bvec lessThanEqual(vec x, vec y);` // ... <= ...?
 - `bvec lessThanEqual(ivec x, ivec y);`
 - `bvec greaterThan(vec x, vec y);` // ... > ...?
 - `bvec greaterThan(ivec x, ivec y);`
 - `bvec greaterThanEqual(vec x, vec y);` // ... >= ...?
 - `bvec greaterThanEqual(ivec x, ivec y);`

4. GLSL Functions: Built-in Riches

- **Vector Relational (2)**: for vector Booleans:
bvec2, bvec3, or bvec4:
- `bool any(bvec x);` // vector ‘OR’
// – true if ANY element true
- `bool all(bvec x);` // vector ‘AND’
//–true if ALL elements true
- `bvec not(bvec x);` // vector ‘NOT’
//–each true element →false,
//–each false element →true.

4. GLSL Functions: Built-in Riches

- **Texture Map Lookup:**

For flat, screen-space textures – images, etc; (**no perspective**)

- `vec4 texture2D (sampler2D sampler, vec2 coord)`
- `vec4 texture2D (sampler2D sampler, vec2 coord, float bias)`

For textures ‘painted’ onto 3D objects w/ **3D camera with perspective**

- `vec4 texture2DProj (sampler2D sampler,vec3 coord)`
- `vec4 texture2DProj (sampler2D sampler,vec3 coord, float bias)`
- `vec4 texture2DProj (sampler2D sampler,vec4 coord)`
- `vec4 texture2DProj (sampler2D sampler,vec4 coord, float bias)`

For **anti-aliased, high-quality** textures ‘painted’ on **3D obj + 3D camera**

- `vec4 texture2DLod (sampler2D sampler,vec2 coord, float lod)`
- `vec4 texture2DProjLod (sampler2D sampler,vec3 coord, float lod)`
- `vec4 texture2DProjLod (sampler2D sampler,vec4 coord, float lod)`

4. GLSL Functions: Make Your Own

Let's try this in class! (Page 55 of GLSL spec)

- How do you declare your own function? function prototypes? Scope?
-
- How, where do you write function bodies?
- Function arguments can have qualifiers:
 - What do the '**in**', '**out**' and '**inout**' qualifiers mean?
 - What is default?(no **in**, **out**, or **inout** qualifier given)?

5. GLSL Preprocessor Directives

- Strong subset of C/C++ directives:

- `#define`, `#undef`,
- `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`
- `#error`, `#pragma`,
- `#extension`, `#version`, `#line`

- Full set of arithmetic operators `(,) , + , - , ~ , * , / , %`
- Full set of relational operators: `>`, `>=`, `==`, `!=`, `<`, `<=`
- Full set of logical/bitwise oprs: `&`, `|`, `^`, `&&`, `||`, `<<`, `>>`
- Limited set of pre-defined macros:
`__LINE__`, `__FILE__`, `__VERSION__`, `GL_ES`

GLSL-ES for WebGL

- You can **always** rely on this GLSL Reference
“The GLSL-ES Shading Language” Version 1.0.17
(on CANVAS→‘WebGL: The Full Specifications’)
- There is no substitute for experience;
you can't master GLSL by just reading about it
- **Very few GLSL debugging tools** – you must
write incrementally, test often, test meticulously.
- Project C demands GLSL mastery for mildly
lengthy vertex shaders and fragment shaders.
- Write & test plenty progressively-better
GLSL code; build up; Don't wait to get started!

GLSL Surprises:

- Some differences and gotchas from C/C++:
 - **No pointers**, strings, chars; no unions, enums; no bytes, shorts, longs; no unsigned. No switch() statements.
 - **No implicit casting** (type promotion):

```
float foo = 1;      // ERROR!      (will 1.0 work?)
```

fails because **you can't implicitly cast int to float.**
 - Instead, **use constructors** for explicit type casts:

```
vec3 foo = vec3(1.0, 2.0, 3.0);
```

```
vec2 bar = vec2(foo); // Drops foo.z component
```
- Uses three **weird** new function parameter qualifiers:
in (default), **out**, or **inout** **to replace pass-by-reference:**
- Functions called by **value-return**: argument values **copied into & out of** function's local parameters
at the **start** of the function call, AND
at the **end** of the function call (!) as needed by **in/out** qualifiers

EX: **Vertex** Shader with **varying** variable, and...

```
const vec4 blu = vec4(0.0, 0.0, 1.0, 1.0);
varying vec3 v_colorOut; // frag shader req'd
void main(void)
{
    gl_Position =
        gl_ModelViewProjectionMatrix * gl_Vertex;
    v_colorOut = blu; // set varying...
    (waitwaitwait—you set value to constant? The var 'blu' ?  
for every vertex? What will be the on-screen result?)
}
```

EX: **Fragment** Shader (REQ'D) with **varying** variable in...

```
varying vec3 v_colorOut;  
// matches varying var name in vert shader  
void main(void)  
{  
    gl_FragColor = v_colorOut;  
    // Set fragment color to v_colorOut value  
    // that was automatically rasterized;  
    // interp'd between each vertex for each pixel...  
}
```

EX: **Fragment** Shader (REQ'D) with **varying** variable ...

```
varying vec3 v_color
```

```
// mat4
```

```
v {
```

```
gl_FragColor = v_color;
```

```
// So if varying is constant, then rasterizer would waste time
```

```
// that's interpolating a constant between vertices!
```

```
// for compiler may silently DISCARD v_colorOut,
```

!SURPRISE! !ERROR!

If Vertex Shader sets varying to constant 'blu', then rasterizer would waste time
and cause surprise errors everywhere (e.g. 'fragment shader: v_colorOut undefined')
seen each vertex

END

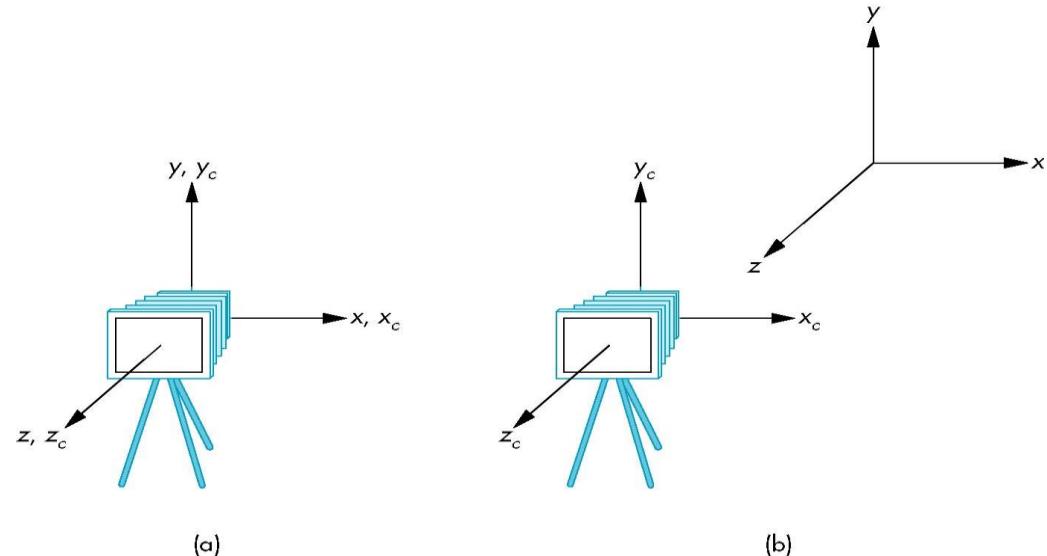
1. Data Types & their Constructors

GLSL Constructors – very flexible!

- Declare & init variables this way:
- **vec3 a = vec3(1.0, 2.0, 3.0);**
- **vec2 b = vec2(a.x, a.y);**

```
vec3 normal = normalize(v_Normal);
```

- Or other ways –
 - web-search ‘WebGL GLSL constructors’ for novel uses, and
 - Search GLSL standards doc to find exact specifications...



Virtual Cameras & Their Matrices

COMP SCI 351-1 Northwestern Univ. Fall 2021

Jack Tumblin Modified, highly edited SLIDES from:

Ed Angel, Professor Emeritus of Computer Science
University of New Mexico

Two Vital Questions:

Where is the Camera?

How Wide is the Lens?

(e.g. the ‘zoom setting’, the image width in degrees)



“Dolly Zoom”

←Unmodified photo sequence from an ordinary camera.

Where is the camera?

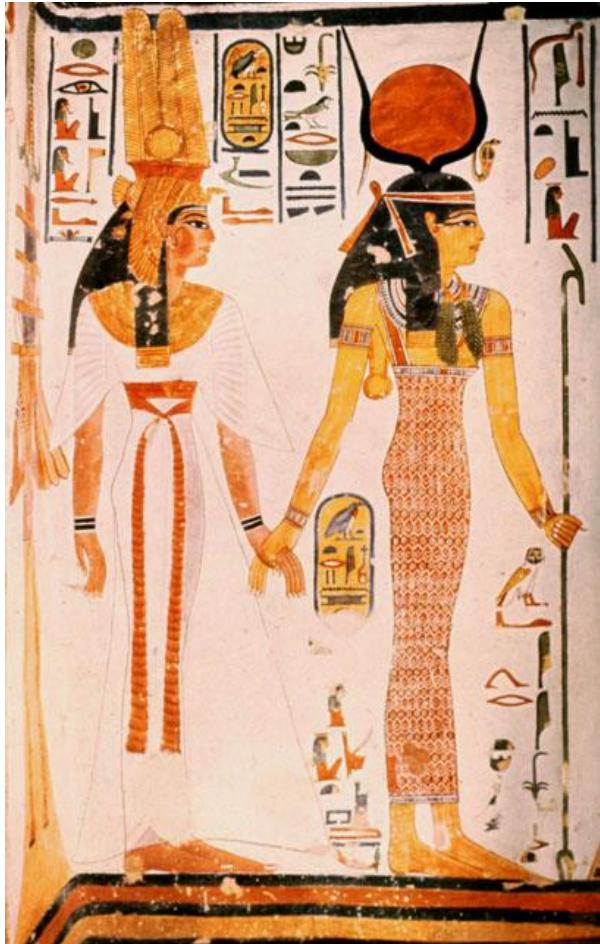
How wide is the lens?

(HINT: slow down!

View it frame-by-frame)

Perspective **SEEMS** hard...

**Because an artwork must ‘undo’ what we see;
then make a new source for visual perception**



History of Perspective?

Try this BEAUTIFUL webpage:

<http://www.essentialvermeer.com/technique/perspective/history.html>



Classical Projections

Painter's GOAL:

Describe all the useful ways
we can “**map**”

‘what we see’ →
of a **3D** ‘scene’ →
onto a **2D** ‘plane’:

History of Perspective?

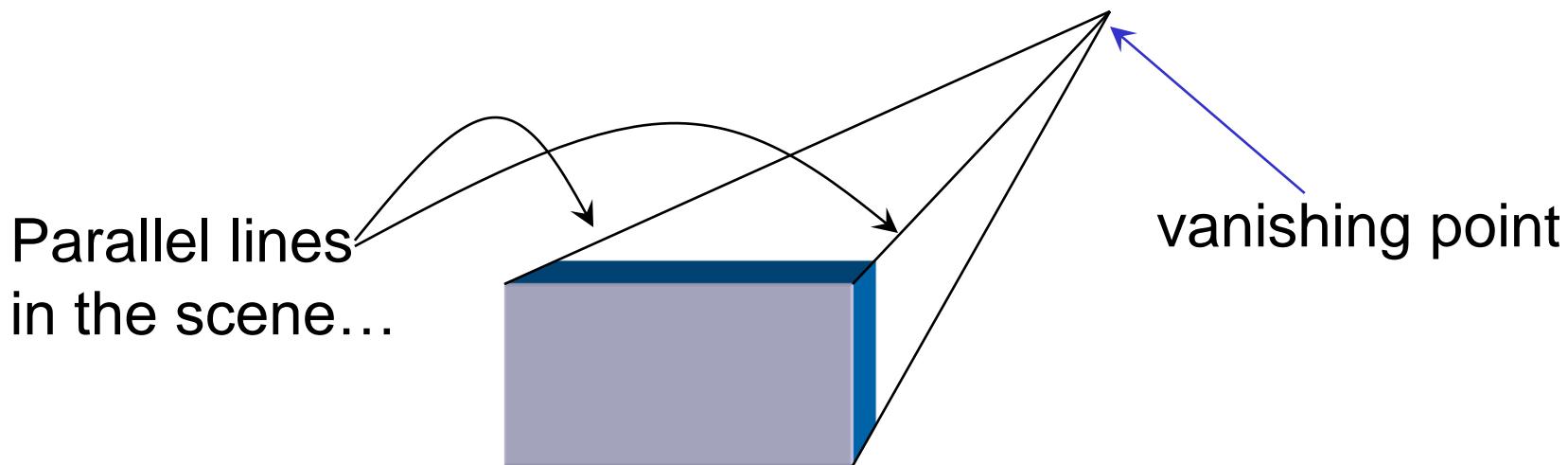
Try this BEAUTIFUL webpage:

<http://www.essentialvermeer.com/technique/perspective/history.html>



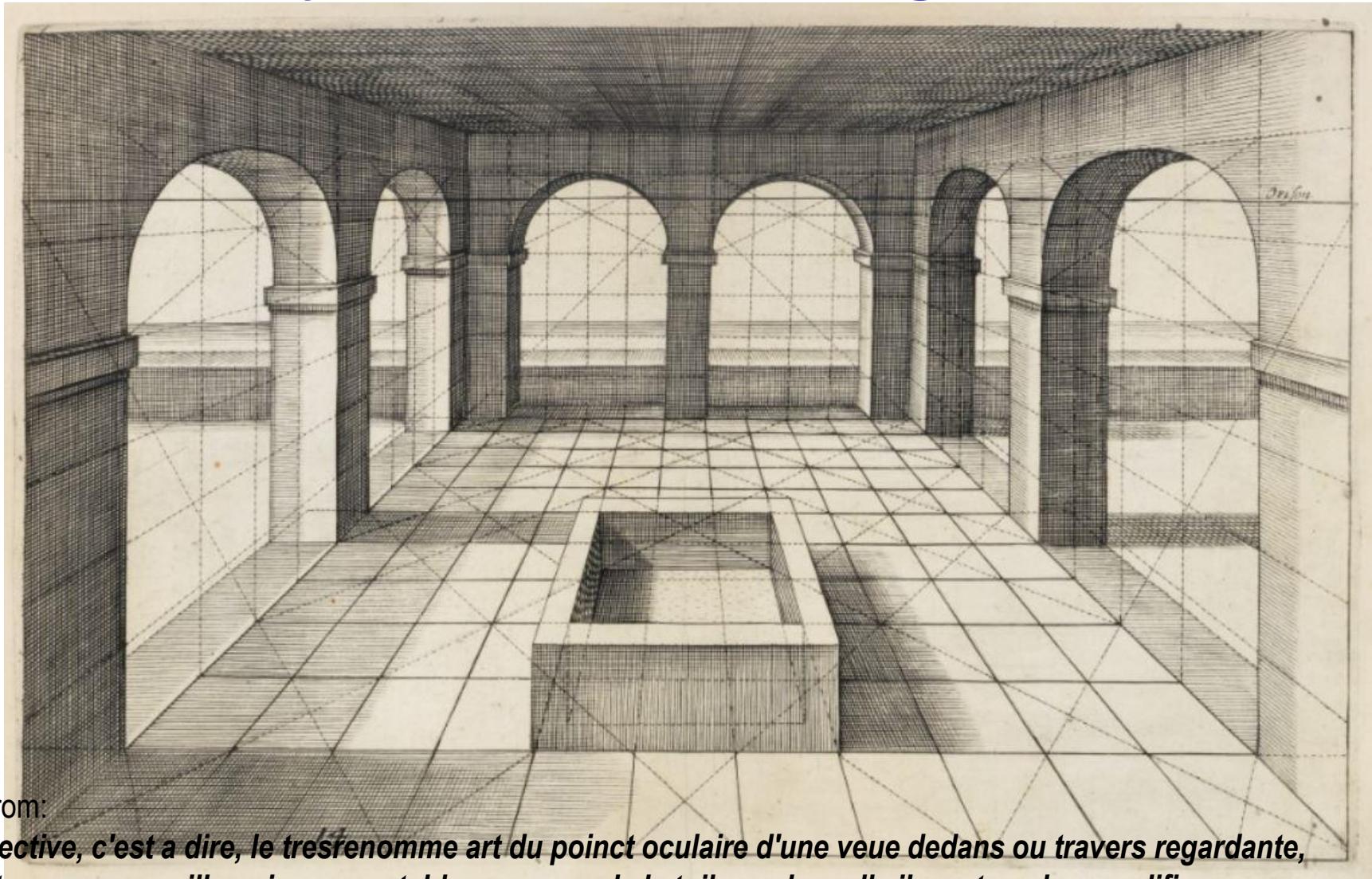
Painter's Tool: Vanishing Points

- All parallel lines *in the viewed scene* converge at one **vanishing point** *in the image display,*
- Idea: choose vanishing points FIRST: draw radiating lines to guide depth depiction



EXAMPLE:

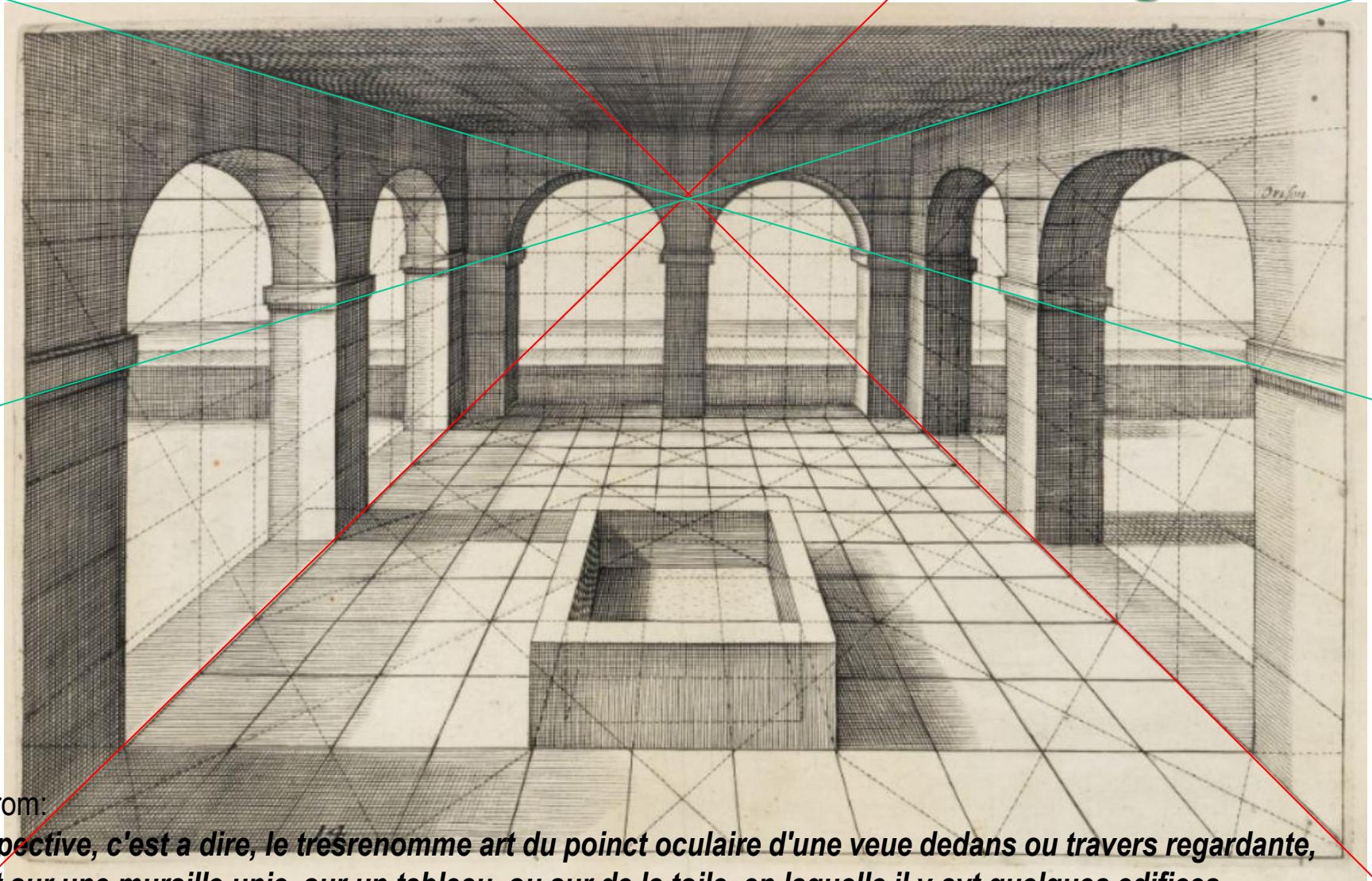
Can you find the Vanishing Point?



Print from:

Perspective, c'est à dire, le très renommé art du point oculaire d'une veue dedans ou travers regardante, estant sur une muraille unie, sur un tableau, ou sur de la toile, en laquelle il y ayt quelques edifices, soyt d'églises, temples, palais, sales, chambres, galeries, places, allees, jardins, marches & rües
by Vredeman de Vriesm, published 1604–1605: The Hague

Draw Lines on wall at floor and ceiling...



Print from:

"Perspective, c'est a dire, le tresrenomme art du poinct oculaire d'une veue dedans ou travers regardante, estant sur une muraille unie, sur un tableau, ou sur de la toile, en laquelle il y ayt quelques edifices, soyt d'eglises, temples, palais, sales, chambres, galeries, places, allees, jardins, marches & rües"
by Vredeman de Vriesm, published 1604–1605: The Hague

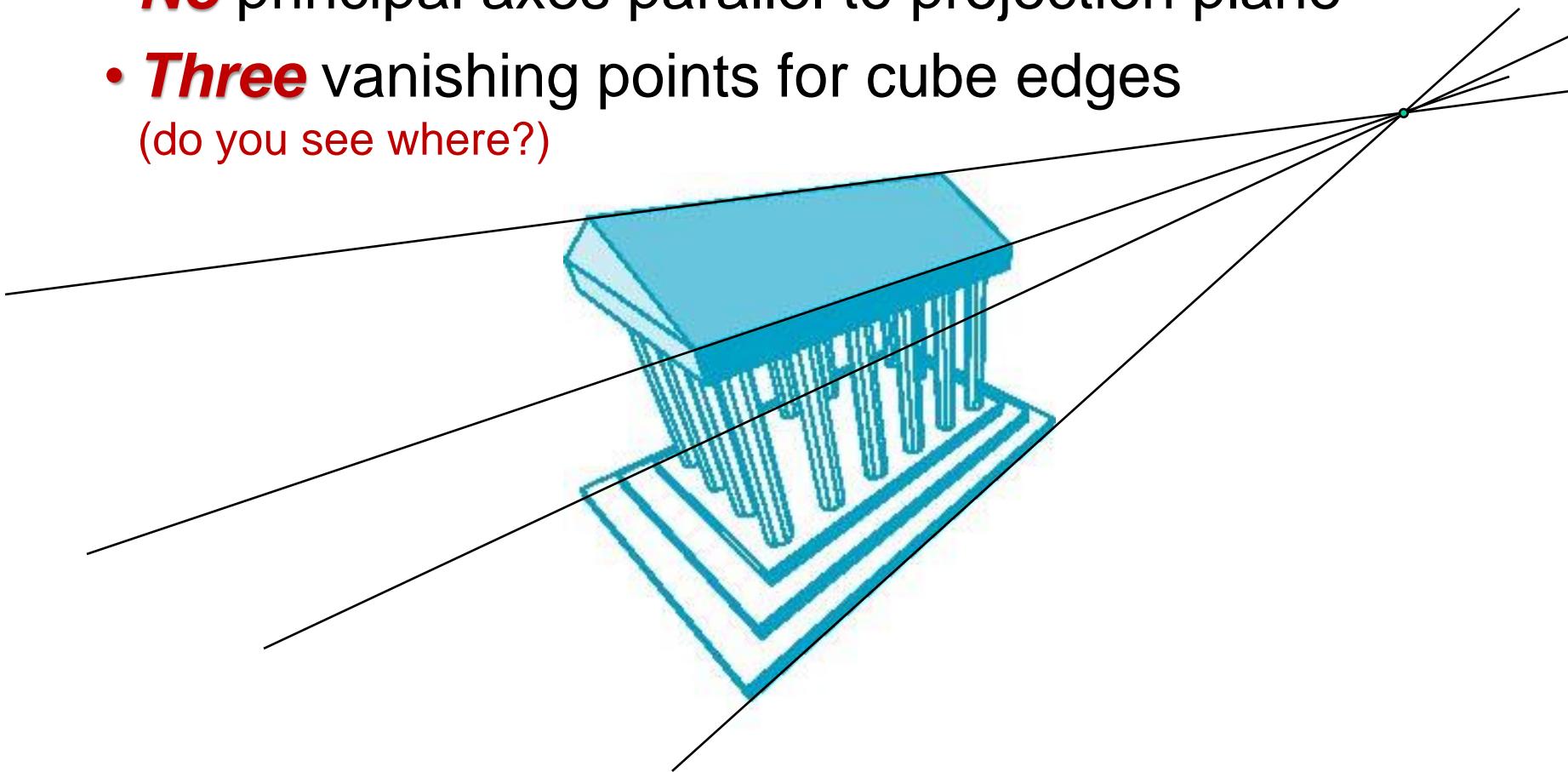
Three-Point Perspective

- **No** principal axes parallel to projection plane
- **Three** vanishing points for cube edges
(do you see where?)



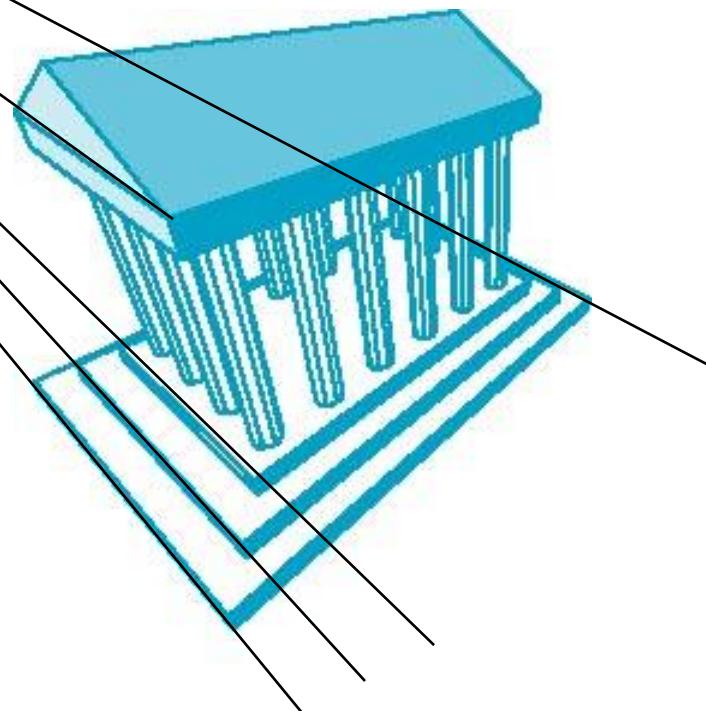
Three-Point Perspective

- **No** principal axes parallel to projection plane
- **Three** vanishing points for cube edges
(do you see where?)



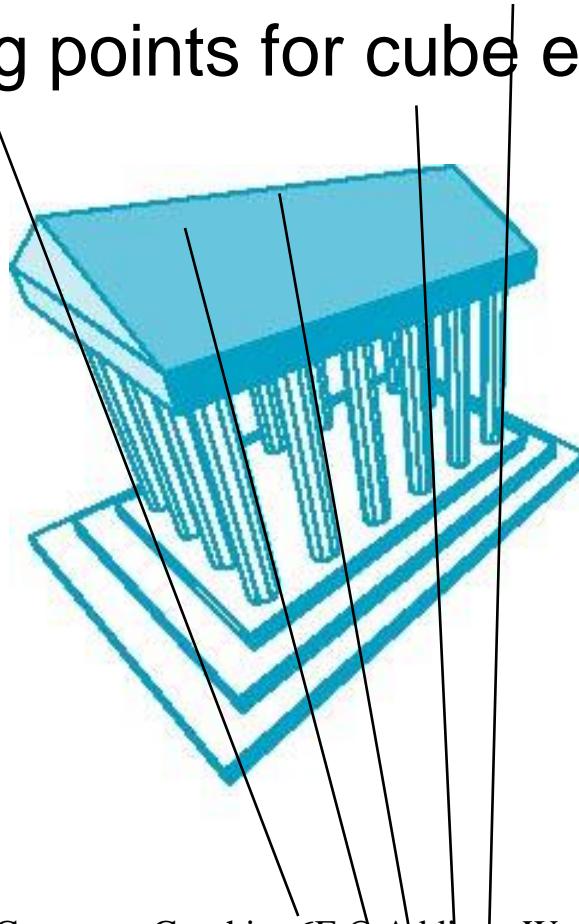
Three-Point Perspective

- **No** principal axes parallel to projection plane
- **Three** vanishing points for cube edges
(do you see where?)



Three-Point Perspective

- **No** principal axes parallel to projection plane
- **Three** vanishing points for cube edges
(do you see where?)

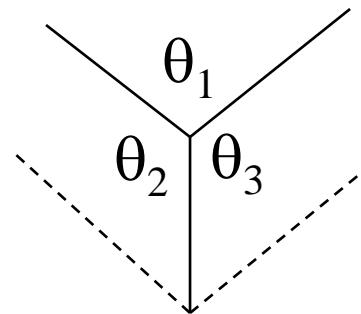


Axonometric Projections

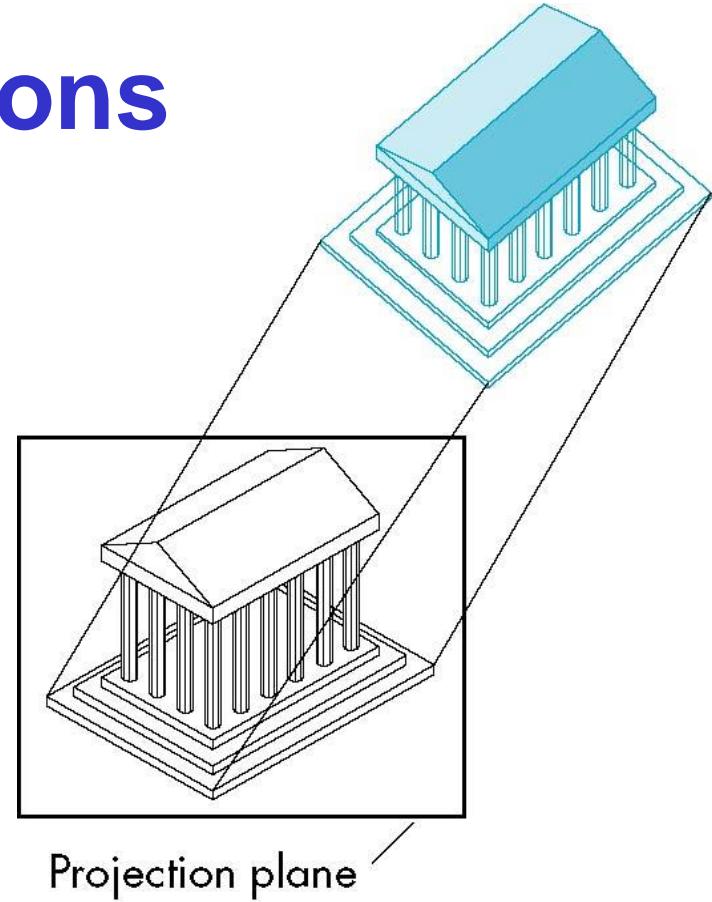
- Vertical **Scene** lines become Vertical **Display** lines;
- Parallel **Scene** lines become Parallel **Display** lines;

- Angles at a cube's corner defines the projections:

- **Isometric**: $\theta_1 = \theta_2 = \theta_3$
- **Dimetric**: $\theta_1 \neq \theta_2 = \theta_3$
- **Trimetric**: $\theta_1 \neq \theta_2 \neq \theta_3$



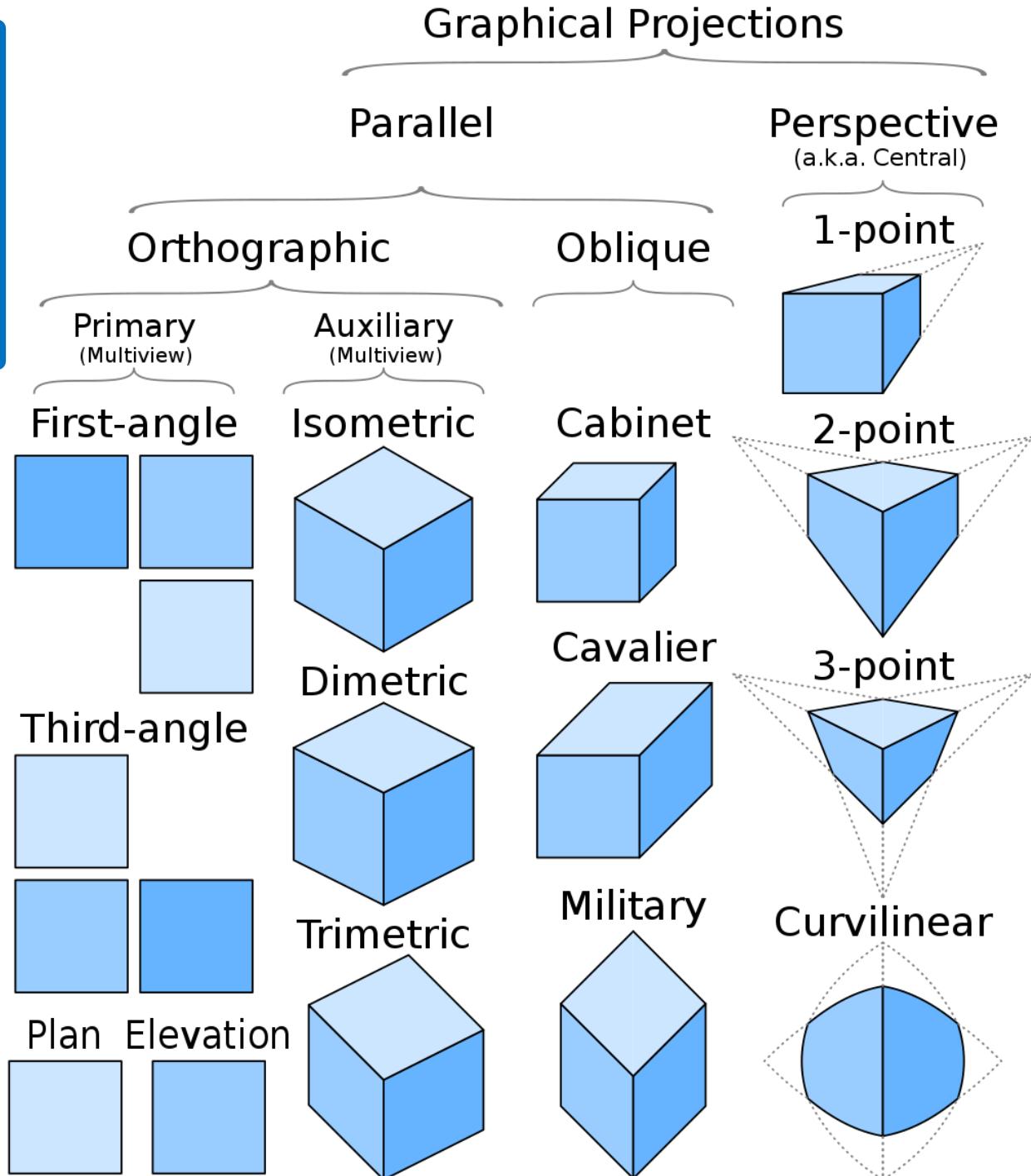
- And that's not all of them! ...



A Taxonomy of Projections

for Traditional
Artists & Illustrators

Wikipedia, “[3D Projection](#)”,
figure by:
Cmglee - Own work,
CC BY-SA 4.0,
[https://commons.wikimedia.org/
w/index.php?curid=83384053](https://commons.wikimedia.org/w/index.php?curid=83384053)



A Taxonomy of Projection

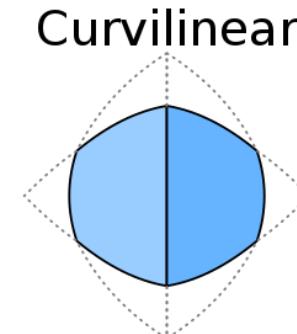
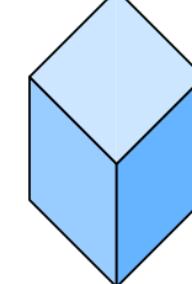
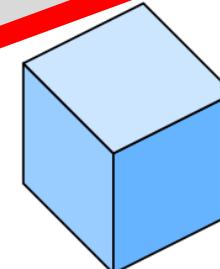
!OH NO! What a mess. Really?

Why all these choices?
Choices missing from our cameras?

How do we Implement it all?
Select? Debug? Support?
Why is this such a mess? ...

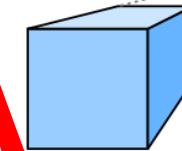
Arti

Wikipedia,
figure by:
Cmglee - Own
CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=10000000>

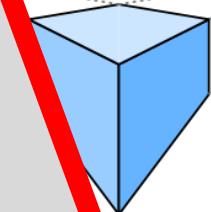


Perspective
(a.k.a. Central)

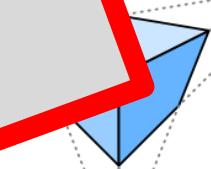
1-point



2-point



point



Curvilinear

Classical VS Computer Graphics

- **Classical Methods:**

for Viewing / Drawing / Painting: highly developed.
GOOD, CLEVER, ad-hoc ways to draw each one!

BUT

mixes scene contents/axes & **camera** pose with
camera optics. ***Heavily inter-dependent.*** Ugh. **UGH!**

- **Computer Graphics Methods:** **.AS SIMPLE AS POSSIBLE.**

Untangle it! Keep scene & camera separate!

ALL WE NEED are 4x4 Matrices:

- **Model·View:** (scene contents) · (camera pose):

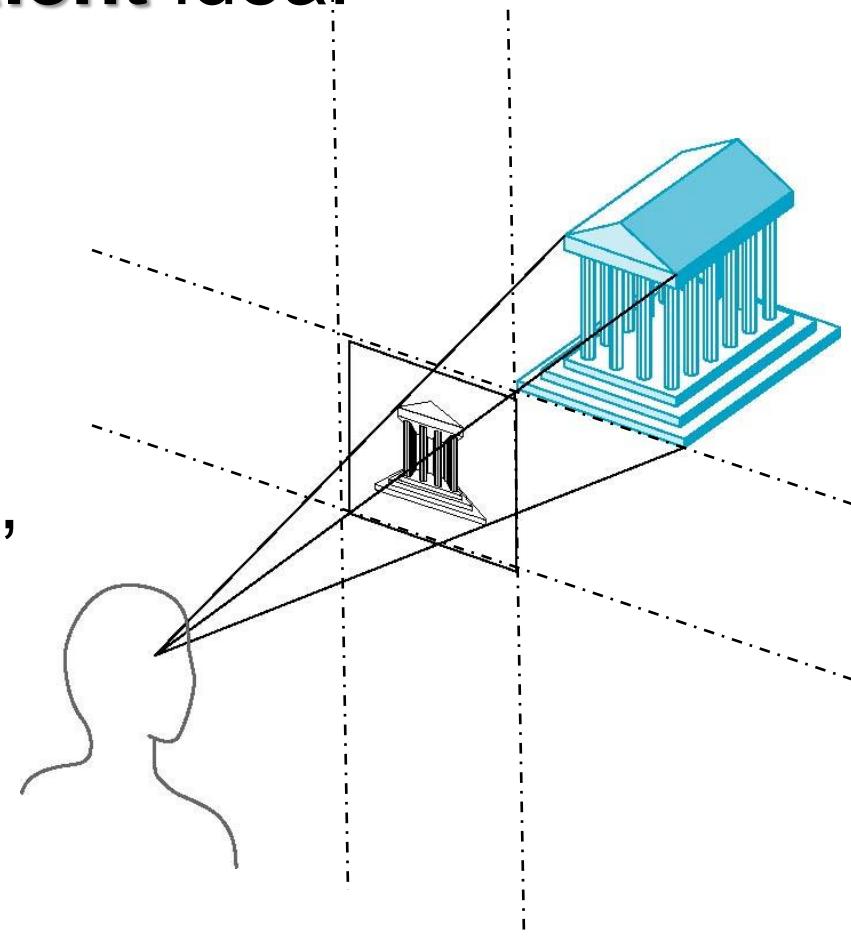
- **Projection:** camera optics only. **Start simply:**

KEY IDEA: Point-Planar Projection

An alternative, **fully-equivalent** idea:

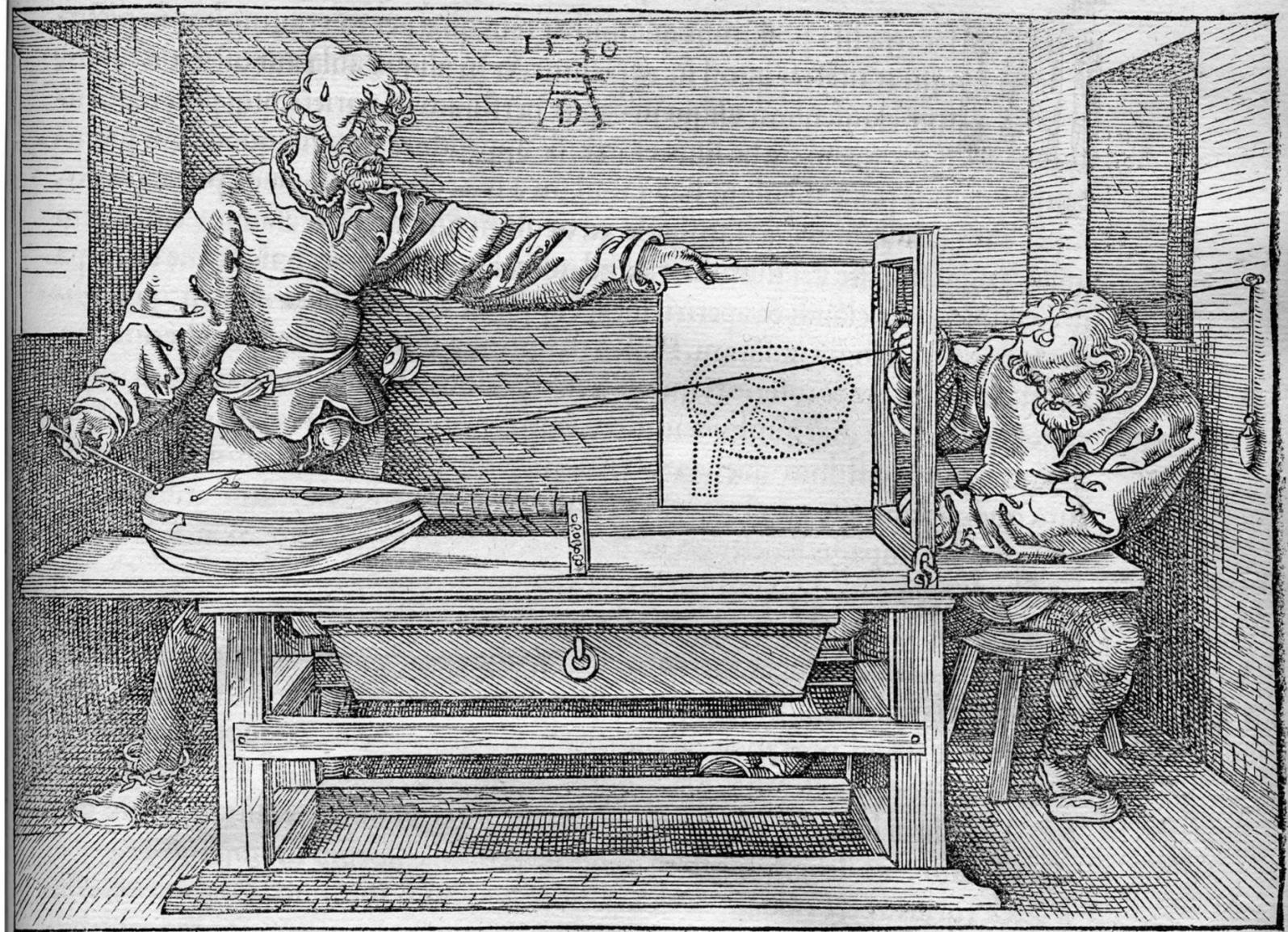
Point-plane perspective:

- Choose a viewpoint,
- Choose a viewing plane,
- Choose a rectangle on that plane as your ‘picture’



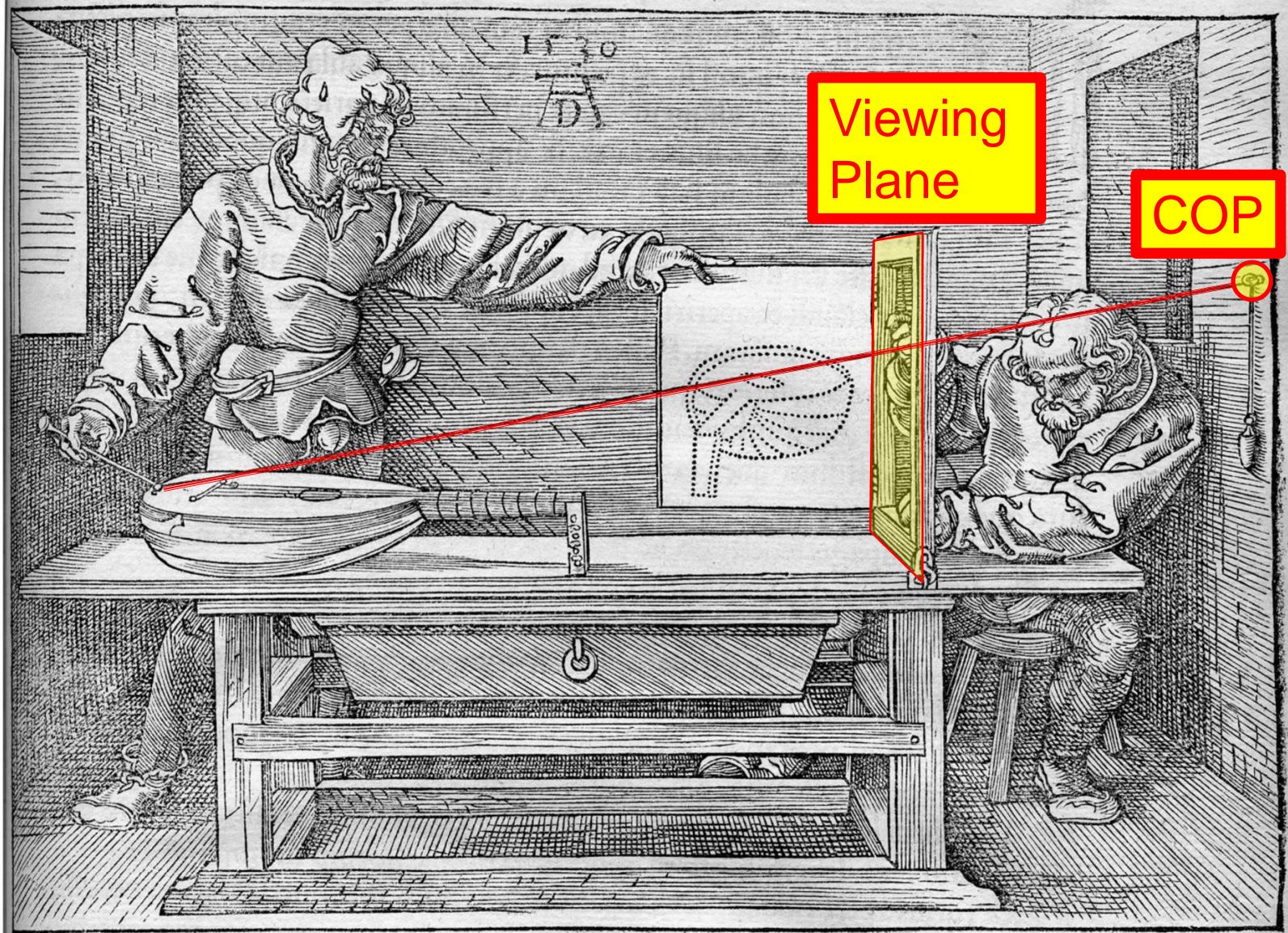
THEN: Viewing rays copy
3D scene colors to plane

Albrecht Durer understood (1530 woodcut)

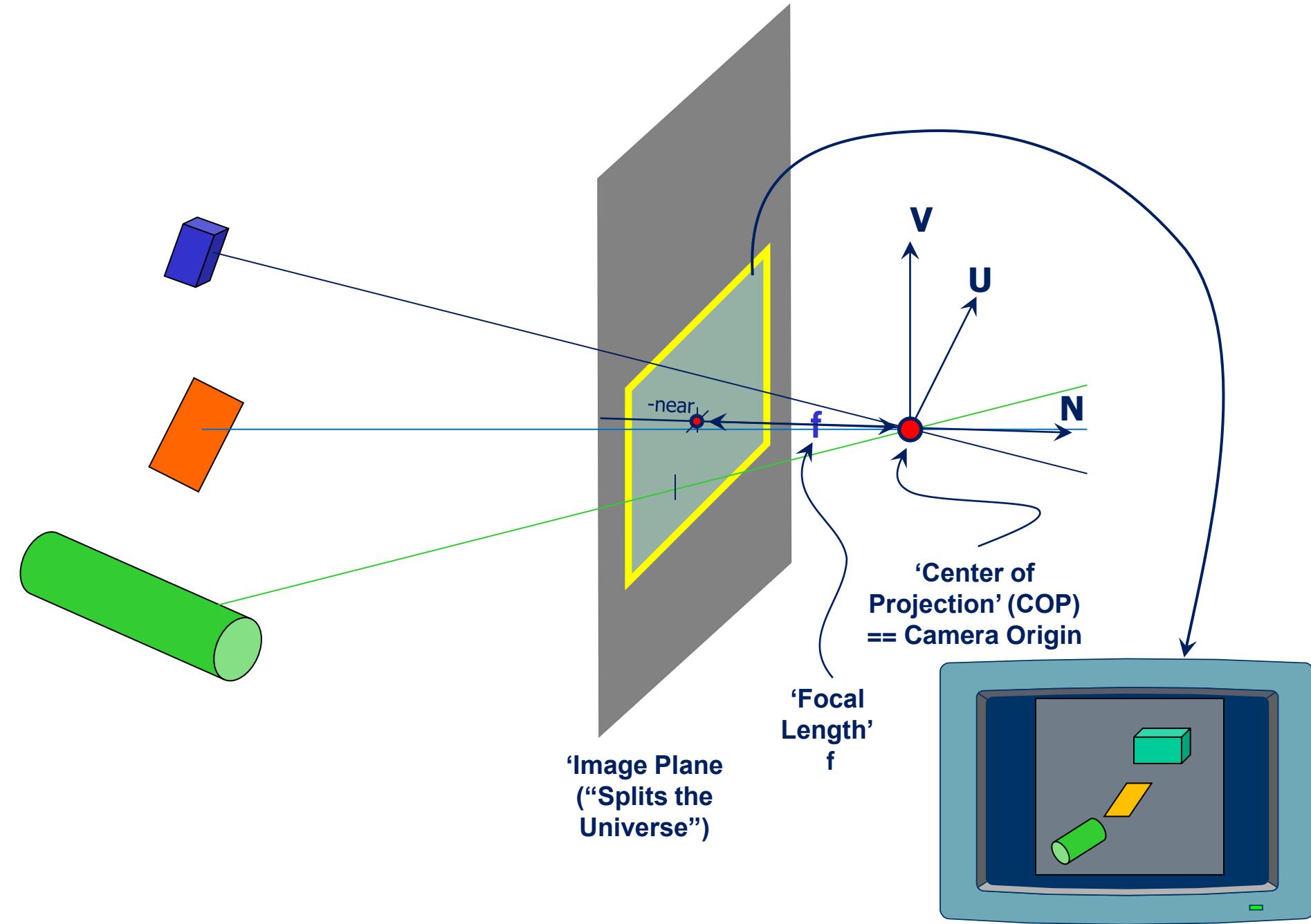


Collection of novel machines for drawing & drafting: <https://drawingmachines.org/index.php> (I built a harmonograph --- can you?)
<https://www.gettyimages.com/detail/illustration/albrecht-d%BCrer-perspective-machine-for-royalty-free-illustration/504822919>

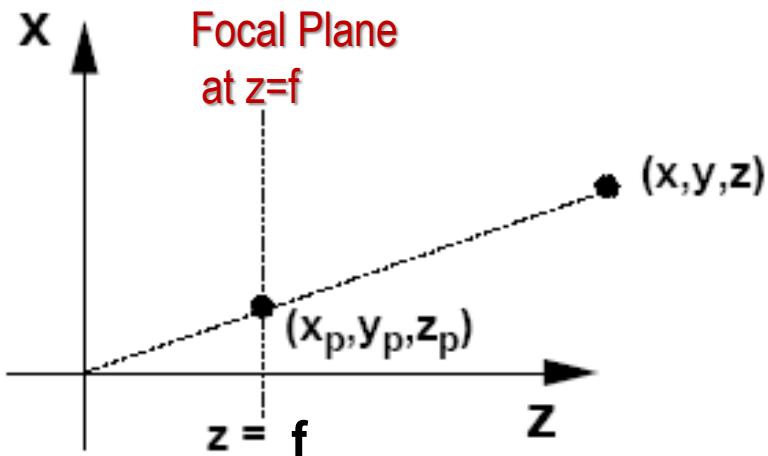
Viewing Plane + Point + Rays



Collection of novel machines for drawing & drafting: <https://drawingmachines.org/index.php> (I built a harmonograph --- can you?)
<https://www.gettyimages.com/detail/illustration/albrecht-d%BCrer-perspective-machine-for-royalty-free-illustration/504822919>



Perspective Projection: Just Divide by z!



What are coordinates of projected point x_p, y_p, z_p ?

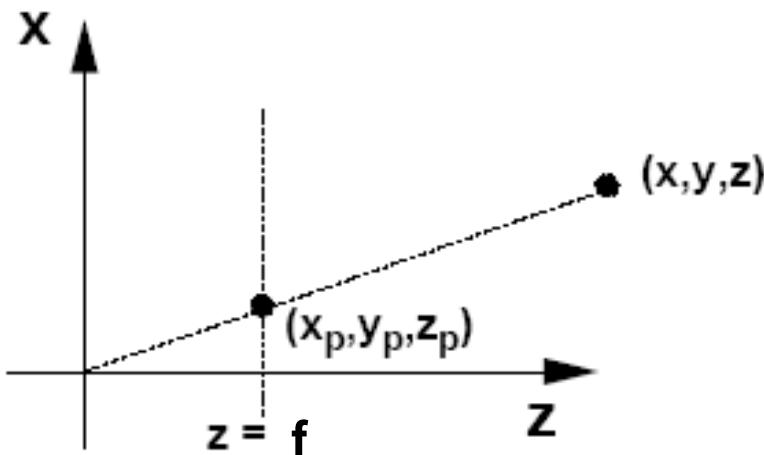
By similar triangles,

$$\frac{x_p}{\mathbf{f}} = \frac{x}{z} \quad \frac{y_p}{\mathbf{f}} = \frac{y}{z}$$

Multiplying through by \mathbf{f} yields

$$x_p = \frac{\mathbf{f} \cdot x}{z} = \frac{x}{z/\mathbf{f}} \quad y_p = \frac{\mathbf{f} \cdot y}{z} = \frac{y}{z/\mathbf{f}} \quad z_p = \mathbf{f}$$

Perspective Projection: Just Divide by z!



What are coordinates of projected point x_p, y_p, z_p ?

By similar triangles,

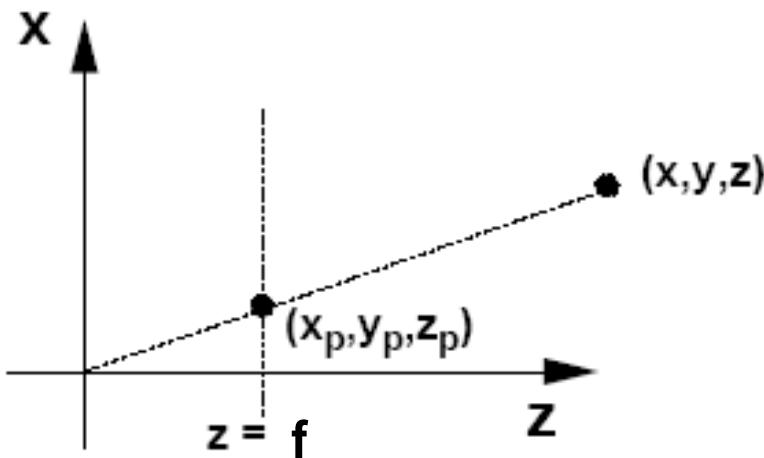
$$\frac{x_p}{f} = \frac{x}{z} \quad \frac{y_p}{f} = \frac{y}{z}$$

Multiplying through by f yields

$$x_p = \frac{f \cdot x}{z} = \frac{x}{z/f} \quad y_p = \frac{f \cdot y}{z} = \frac{y}{z/f} \quad z_p = f$$

Puzzle: What's (x_p, y_p, z_p) when $f = 0$?

Perspective Projection: Just Divide by z!



What are coordinates of projected point x_p, y_p, z_p ?

By similar triangles,

$$\frac{x_p}{f} = \frac{x}{z} \quad \frac{y_p}{f} = \frac{y}{z}$$

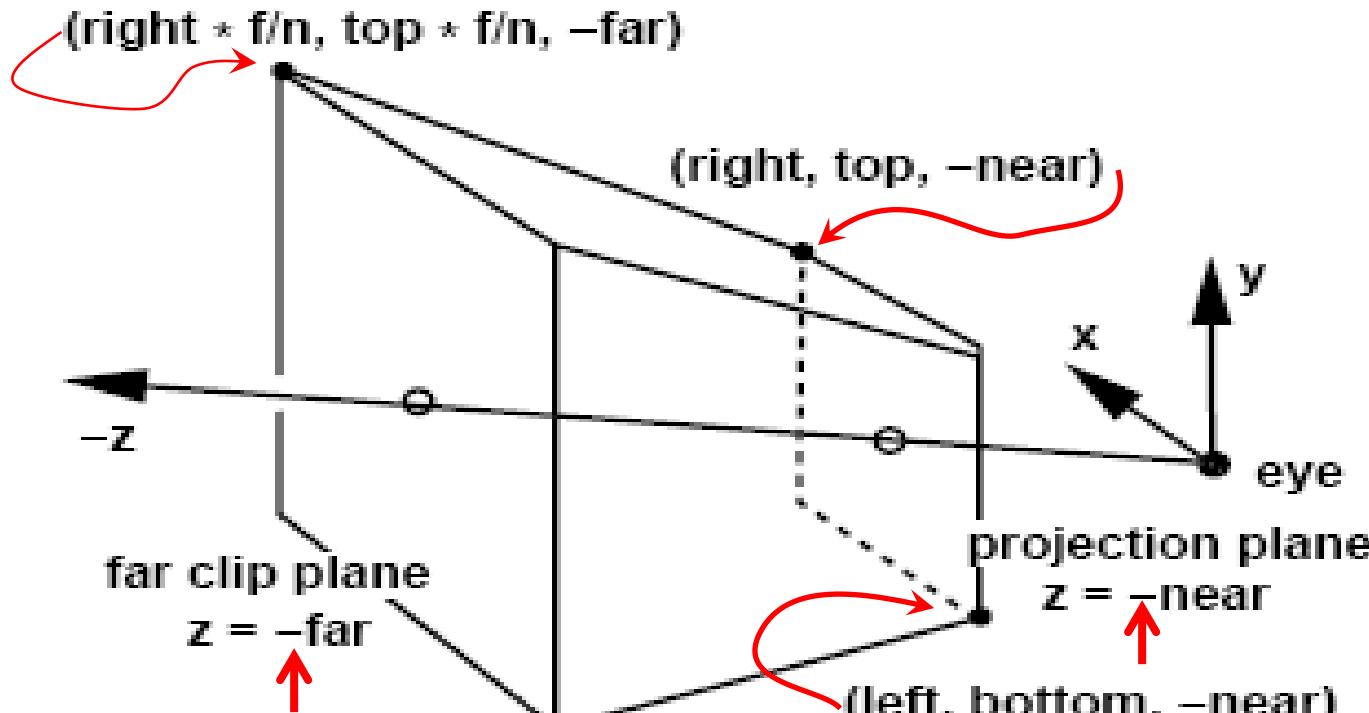
Mul **ANSWER:**
Camera with $f = 0$ undefined;
similarly, perspective drawing of point $(x, y, z) = (0,0,0)$ undefined.

- what happens to the image in the limit as $f \rightarrow 0$?
- ‘Perspective’ is well-defined for all other points & f values.
- Divide by zero? !HARDWARE TROUBLE! NaN. Math stops.

What WebGL camera will draw: Contents of ‘Viewing Frustum’

WebGL 3D perspective camera capture a VOLUME (a frustum)

defined by 6 parameters: left, right, bottom, top, near, far



right-handed; view is along $-z$ axis

How does Projection matrix create this ‘frustum’?

Homogeneous → Cartesian Coords

WHY use homogeneous coords?

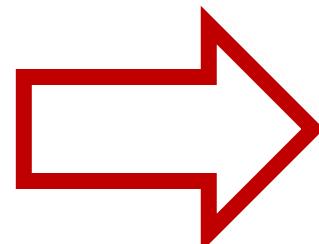
→ For robust 3D perspective cameras

(e.g. the best reason WHY to set $w=1$ for points!)

Homogeneous

Coordinates:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$



Cartesian

Coordinates:

$$\begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}$$

Perspective in Homogeneous Coordinates: (Naïve Form – not used!)

consider $\mathbf{q} = \mathbf{Mp}$ where $\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$

$$\mathbf{q} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \Rightarrow \mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

Perspective Division: Naïve Form

- Note that $w \neq 1$, so we **must divide by w** for Homogeneous \rightarrow Cartesian coordinates
- This ***perspective division*** yields

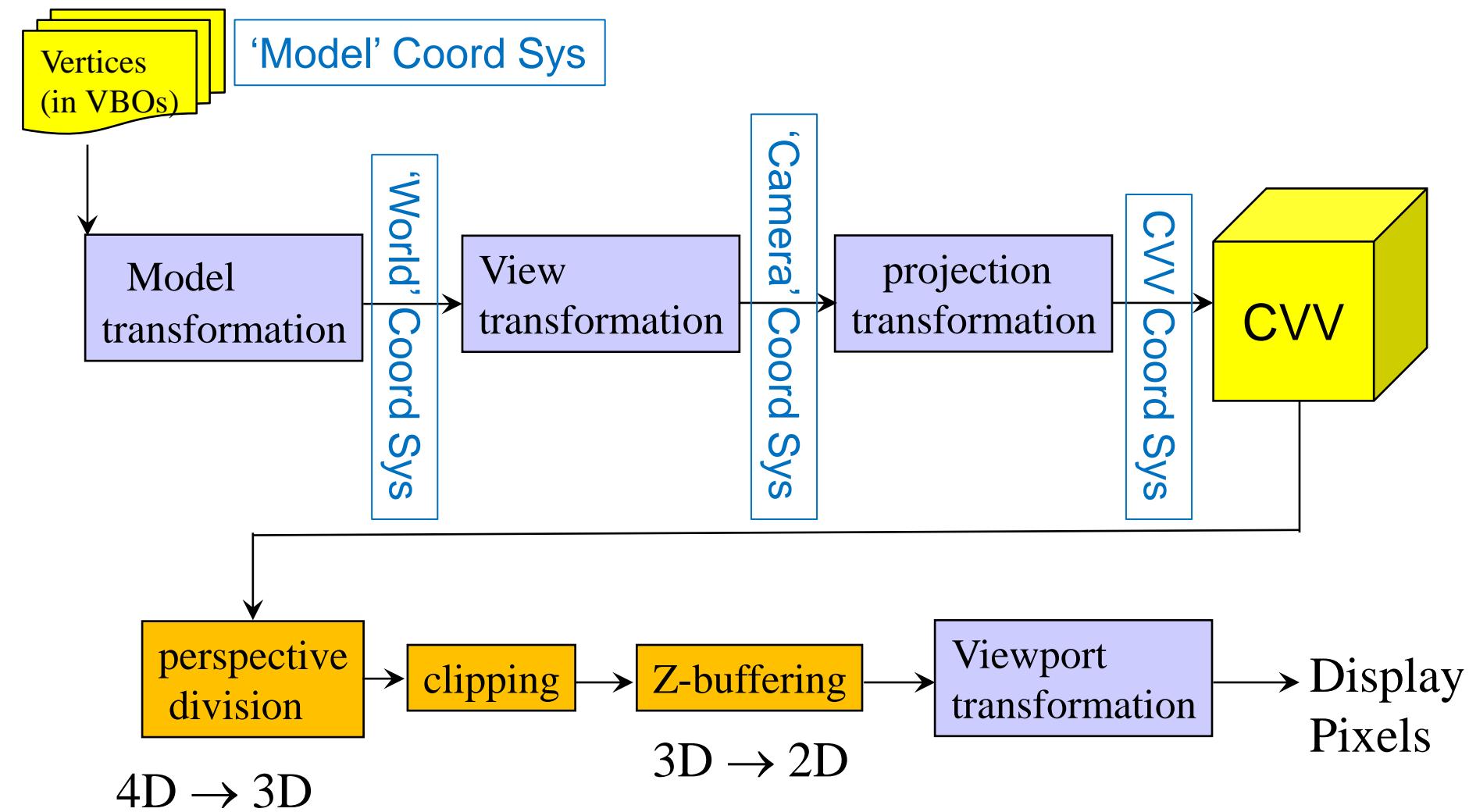
$$x_p = \frac{x}{z/d} \quad y_p = \frac{y}{z/d} \quad z_p = d$$

(if $d==f$, exactly the perspective equations we wanted!)

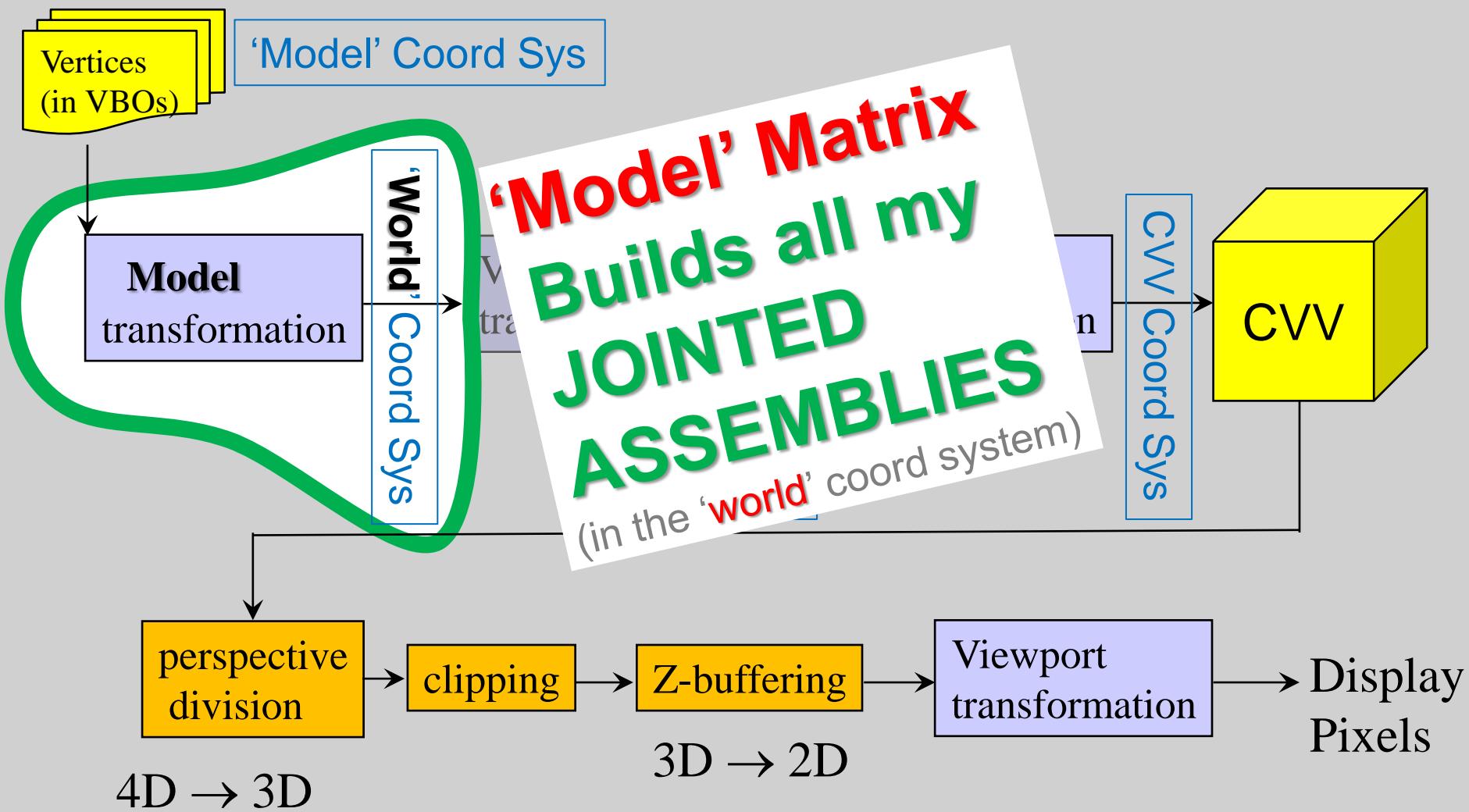
Good Start! How can we make it better?

- What form of ' $1/d$ ' in matrix **M** can include the lens focal length ' f '?
- How can we preserve 'depth' for "depth-buffered" drawing?

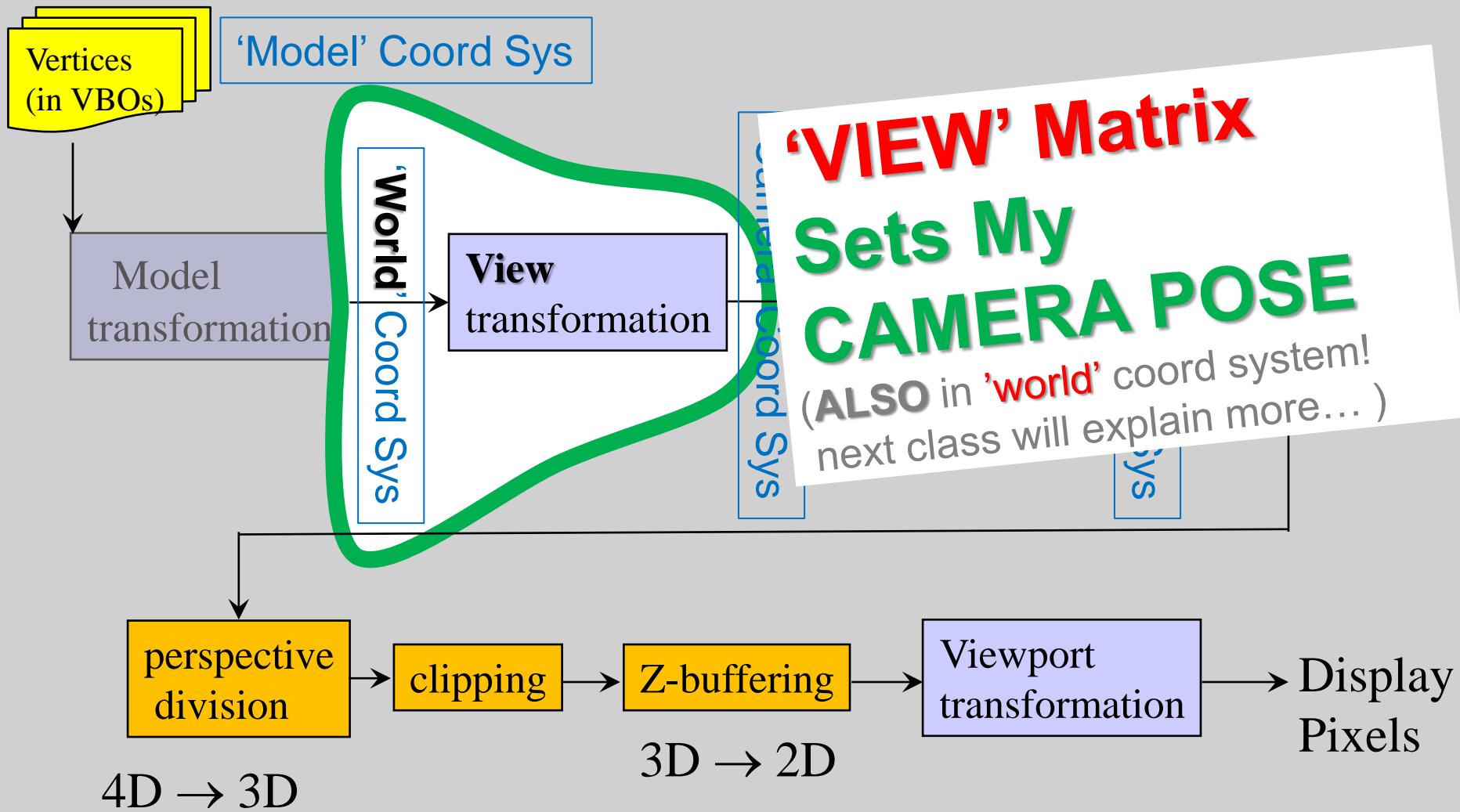
Traditional Vertex Position Pipeline



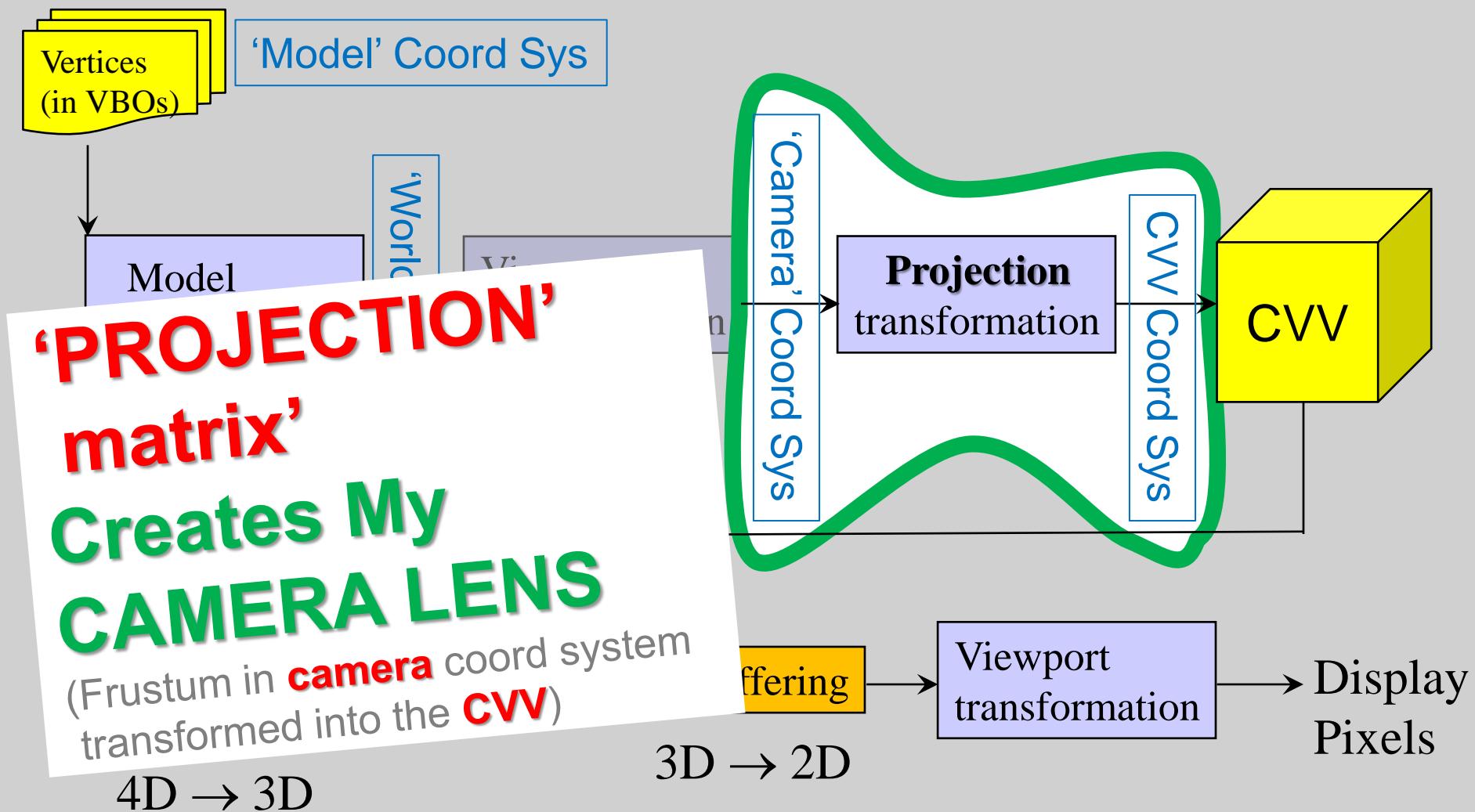
Traditional Vertex Position Pipeline



Traditional Vertex Position Pipeline



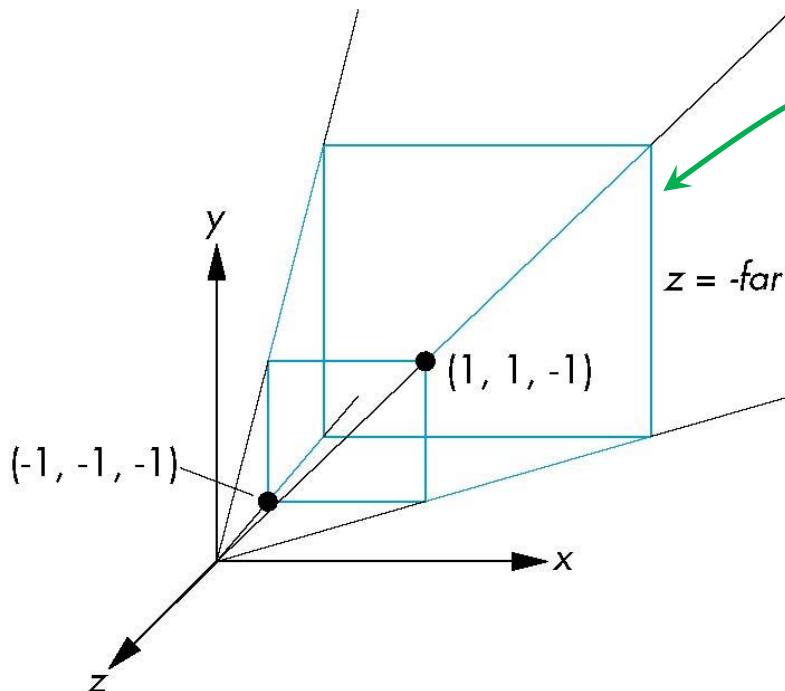
Traditional Vertex Position Pipeline



Point-Perspective Camera

Consider a simple point-perspective image with the COP at the origin, the near clipping plane at $z = -1$, and a 90 degree field of view determined by the planes

$$x = \pm z, y = \pm z$$



Camera's
'Viewing
Frustum'
(3D volume
captured by
Our camera)

Point-Perspective Camera

Consider a simple point perspective camera.
the COP is at $(0, 0, 0)$.

$z =$

$b \cdot v$

$x =$

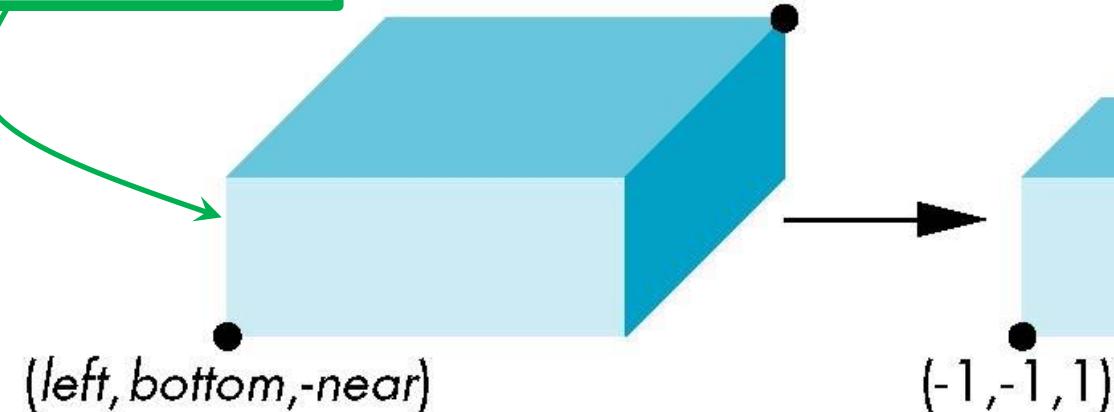
No: Not Yet!
**Try a Simpler
Camera!**



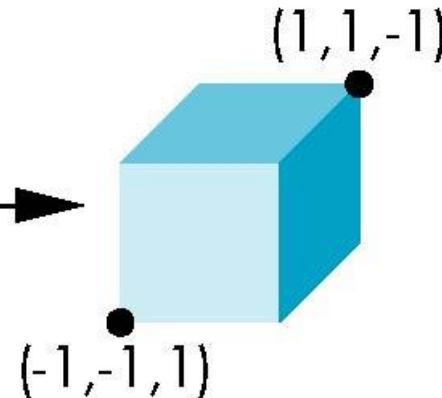
Orthographic Camera

Camera's
'Viewing
Frustum'
(3D volume
captured by
Our camera)

View Volume in
CAMERA coords:
(right,top,-far)



View Volume
CVV coords:
(1,1,-1)



Orthographic Camera

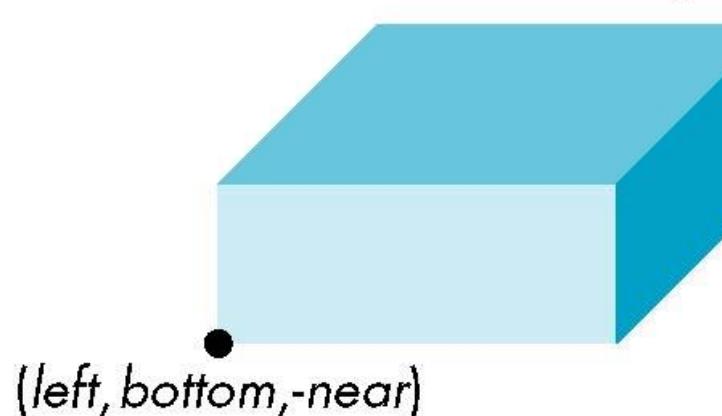
Simple Orthographic **Projection** Matrix:

re-scale a rectangular volume in CAM coords
to fit within the CVV

Matrix4.ortho(left, right, bottom, top, near, far)

View Volume

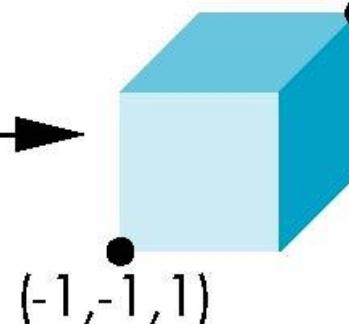
CAMERA coords: (right,top,-far)



View Volume

CVV coords:

$(1, 1, -1)$



How to Make ‘Ortho’ Matrix:

- Two steps:
 - Translate camera’s **frustum center to the origin:**
 $T(-(left+right)/2, -(bottom+top)/2, (near+far)/2))$
 - **Scale to have sides of length 2** (to match the CVV)
 $S(2/(left-right), 2/(top-bottom), 2/(near-far))$

RESULT:

$$\mathbf{P} = \mathbf{ST} = \begin{bmatrix} \frac{2}{right - left} & 0 & 0 & -\frac{right - left}{right - left} \\ 0 & \frac{2}{top - bottom} & 0 & -\frac{top + bottom}{top - bottom} \\ 0 & 0 & \frac{2}{near - far} & \frac{far + near}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

WebGL Key Idea: Display ***only*** the CVV contents!

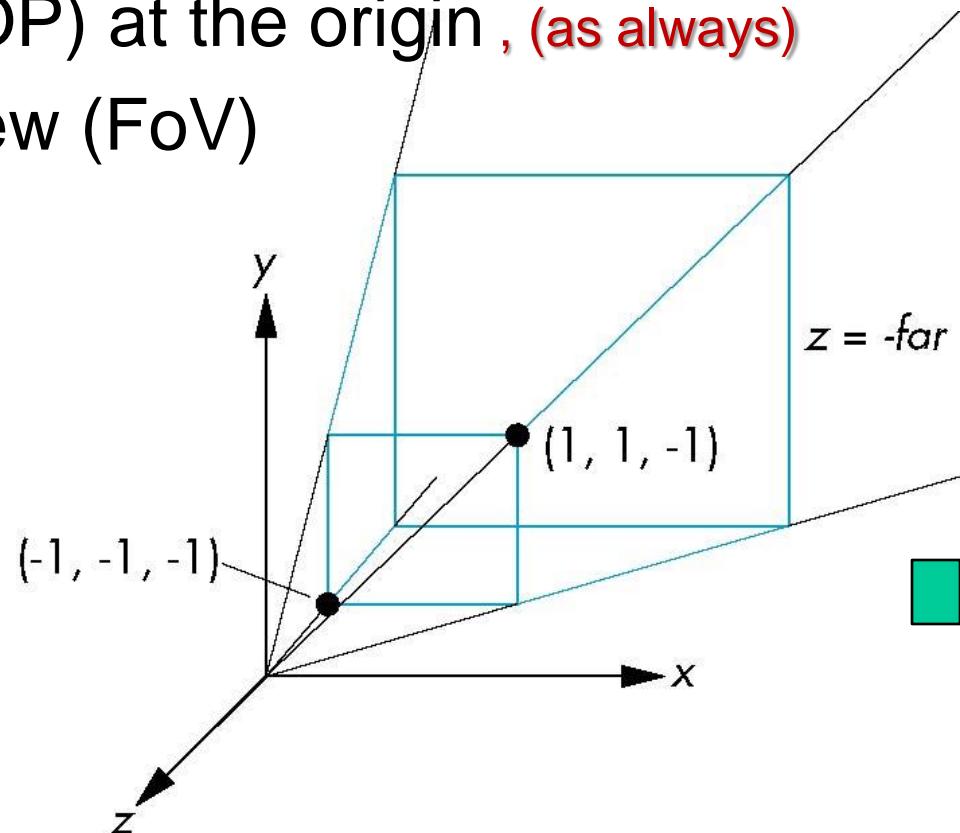
- Keeps 3D clipping simple, fixed and fast
 - *Won't* need different clipping volumes for each kind of frustum, lens or camera:
 - Instead, **map their limits into the CVV** for orthogonal projections onto the display
- RESULT:

- One camera → One matrix;
- One camera type →
One matrix-making function.

Return: Point-Plane Perspective

Try this – a point-perspective image, ****BUT****
aimed at scene in the **MINUS Z AXIS (-Z)**:

- Center of Projection (COP) at the origin , **(as always)**
- $+/-45^\circ$ degree field of view (FoV)
- Image forms on
near clip plane at $z = -1$
- Scene ends at a chosen
far clip plane: $z = -\text{far}$



**Can we make a
matrix for this camera?**

(Naïve) Perspective Matrix

! YES ! Simple '-Z' projection matrix
in homogeneous coordinates:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Homogeneous result:

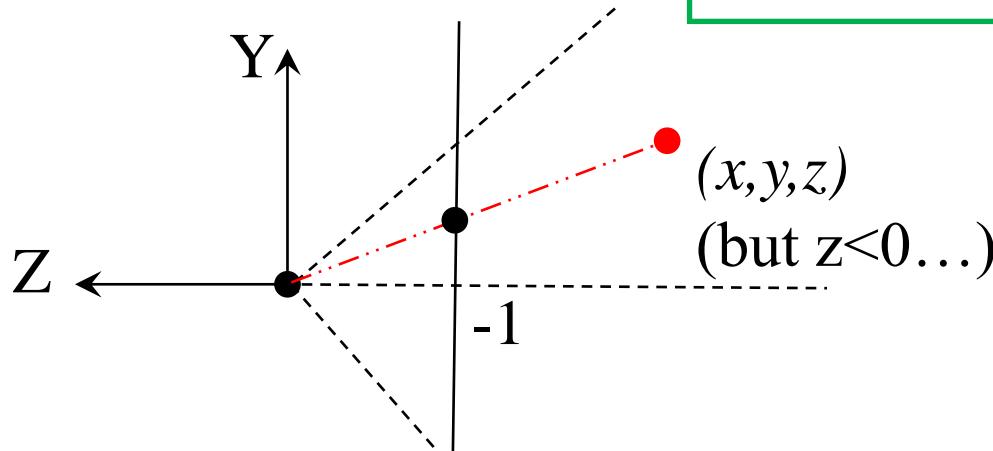
$$x' = x$$

$$y' = y$$

$$z' = z$$

$$w' = -z$$

*So, what's the
Cartesian result?*



(Naïve) Perspective Matrix

! YES ! Simple '-Z' proj:

in horiz.

$$M = \begin{bmatrix} 1 & & & \\ 0 & & & \\ 0 & & & \\ 0 & 0 & & \end{bmatrix}$$

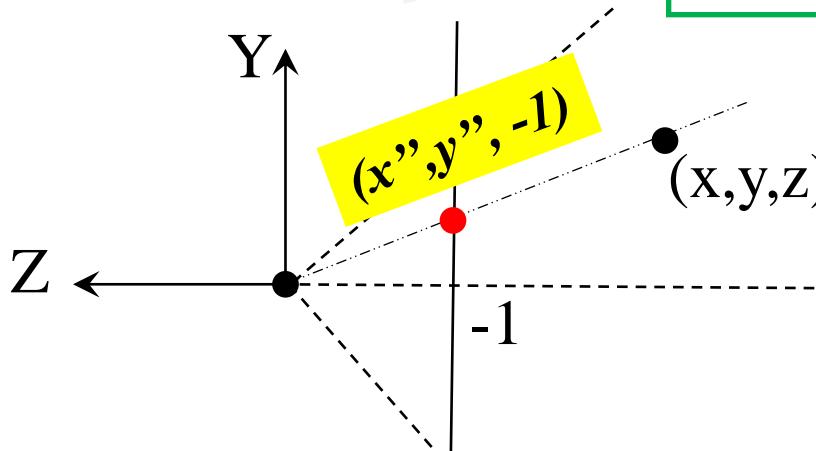
Trouble!

Image forms at $z = -1$;

- ignores 'near' clipping plane, and
- CVV limits us to $+1 \leq z \leq -1$ so
- 'far' plane is ALSO $-1 \leq z \leq +1$ Ugh!

, what's the

Cartesian result?



BUT $z < 0$ for all scene points, so **no sign change for** $(x, y) \rightarrow (x'', y'')$

One Weird Trick to fix it All

(perfectionists hate it 😒)

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

after perspective division, the point at $(x, y, -z, 1)$ becomes

$$x'' = x/z$$

$$y'' = y/z$$

$$z'' = -(\alpha z + \beta)/z$$

Choose α and β carefully...

Generalize to fix it:

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

after perspective division, the point at $(x, y, z, 1)$ becomes

$$x'' = x/z$$

$$y'' = y/z$$

$$z'' = -(\alpha z + \beta)/z$$

What Weirdness
is THIS?!??!

Could it span -1 to +1 (as x'' and y'' might)?

Could we fit it into the CVV?

‘Pseudo-Depth’: How to Pick α and β

If we pick

$$\alpha = \frac{\text{near} + \text{far}}{\text{far} - \text{near}}$$

$$\beta = \frac{2\text{near} * \text{far}}{\text{near} - \text{far}}$$

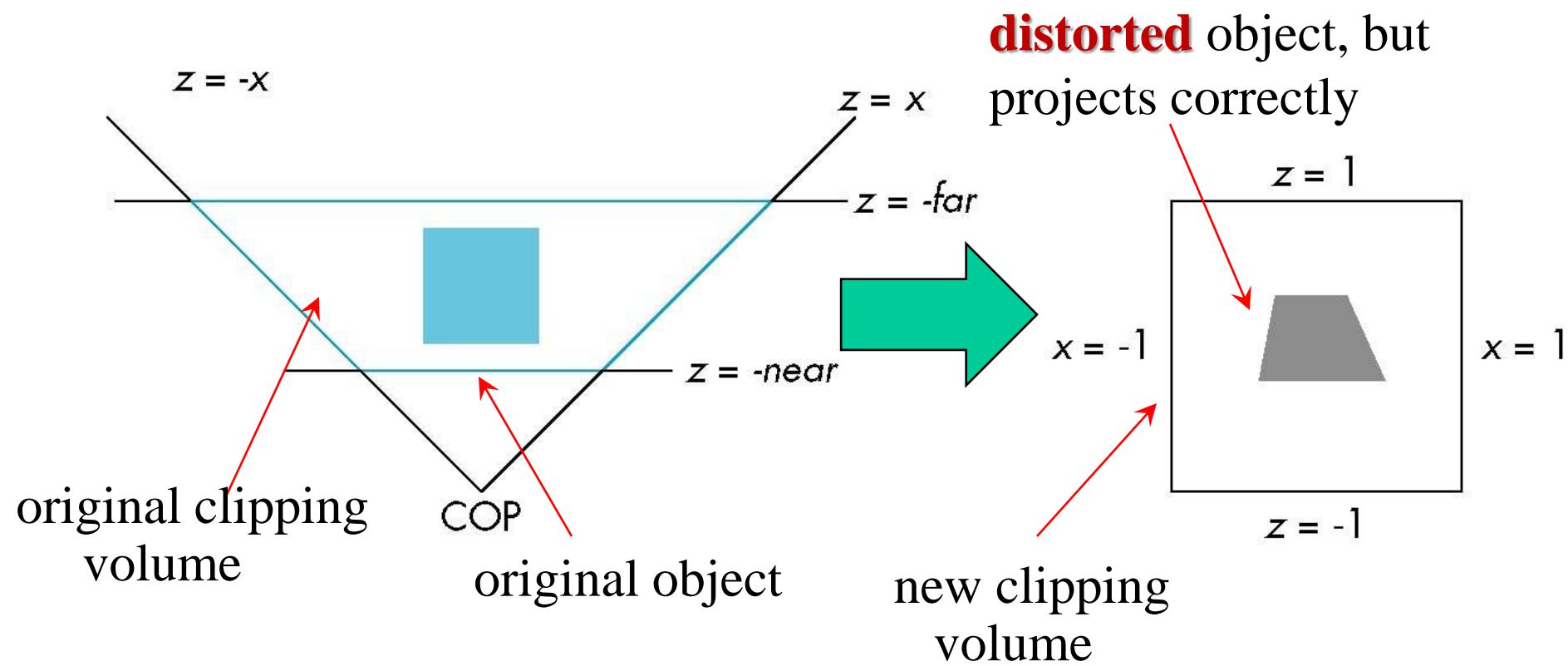
maps the **near plane** to $z = 1$ ('front' of CVV)

maps the **far plane** to $z = -1$ ('back' of CVV)

maps the frustum sides to $x = \pm 1, y = \pm 1$ ('sides' of CVV)

THUS *the viewing frustum fills the CVV exactly!*

Why use this Weird Formulation?

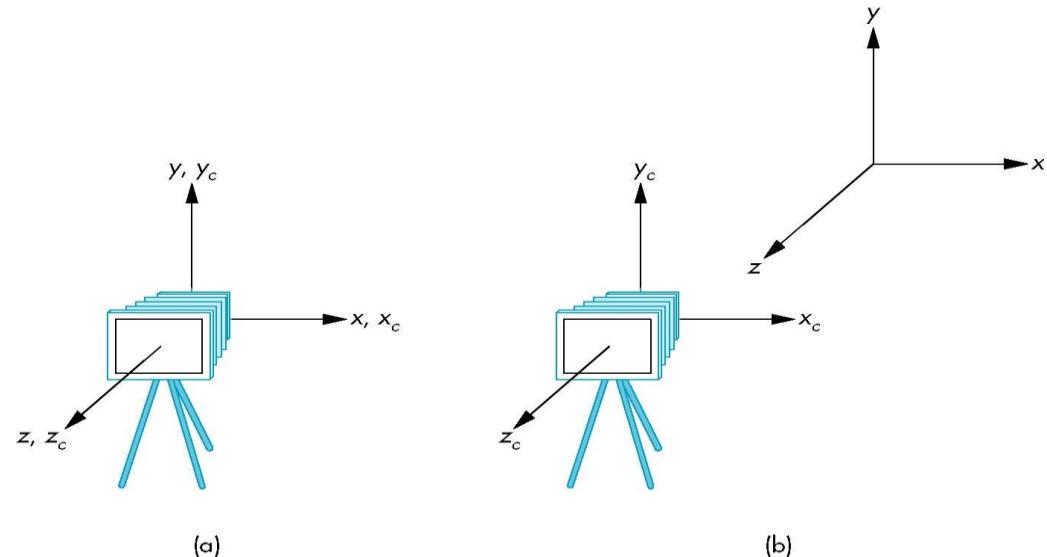


ANS: Normalization, and Hidden-Surface Removal

- Although our '**pseudo-depth**' may SEEM weirdly arbitrary, it **ensures monotonic depth**: if $z_1 > z_2$ in the original clipping volume, then inside the CVV we always have $z'_1 > z'_2$
- Thus hidden surface removal *always* looks right & works perfectly using the 'z' values in the CVV
- *BUT* this '**pseudo-depth**' **DISTORTS** distances. In the CVV, it compresses far-away depths and expands nearby depths: $z'' = -(\alpha z + \beta) / z$
(and it leaves x,y perspective unchanged: $x'' = x/z$; $y'' = y/z$)
- Tiny **z_{near}**? Giant **z_{far}**?
Expect **coarse** depth quantization !

END

END



Virtual Cameras & Their Matrices

COMP SCI 351-1 Northwestern Univ. Fall 2021

Jack Tumblin Modified, highly edited SLIDES from:

Ed Angel, Professor Emeritus of Computer Science
University of New Mexico

Two Vital Questions:

Where is the Camera?

How Wide is the Lens?

(e.g. the ‘zoom setting’, the image width in degrees)



“Dolly Zoom”

←Unmodified photo sequence from an ordinary camera.

Where is the camera?

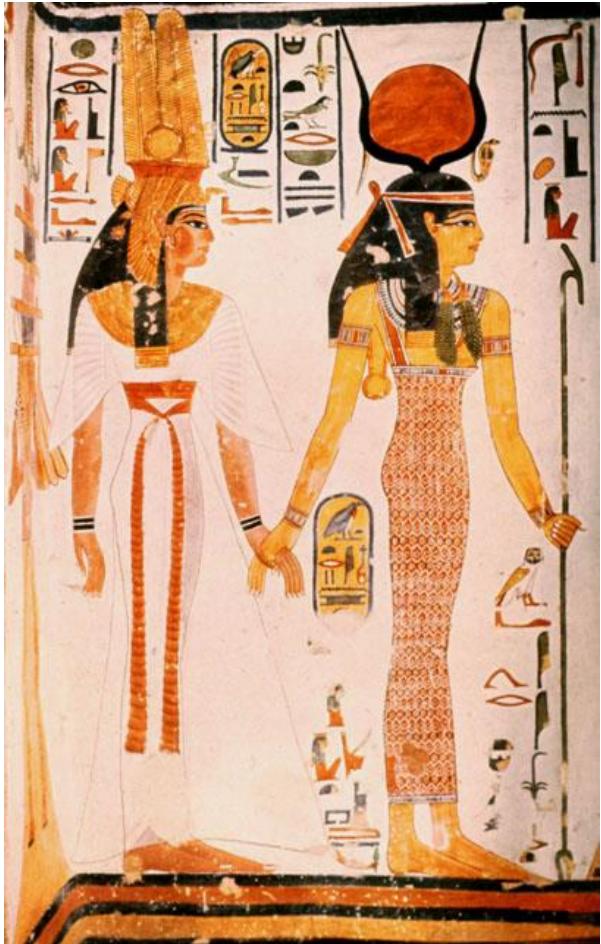
How wide is the lens?

(HINT: slow down!

View it frame-by-frame)

Perspective *SEEMS* hard...

*Because an artwork must ‘undo’ what we see;
then make a new source for visual perception*



History of Perspective?

Try this BEAUTIFUL webpage:

<http://www.essentialvermeer.com/technique/perspective/history.html>



Classical Projections

Painter's GOAL:

Describe all the useful ways
we can “**map**”

‘what we see’ →
of a **3D** ‘scene’ →
onto a **2D** ‘plane’:

History of Perspective?

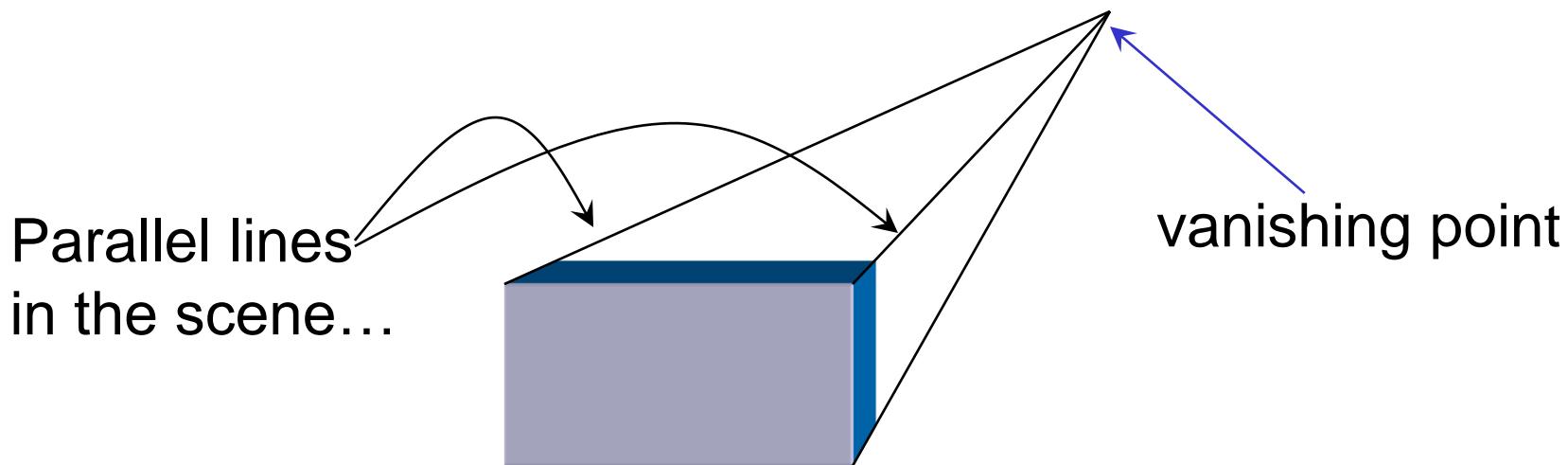
Try this BEAUTIFUL webpage:

<http://www.essentialvermeer.com/technique/perspective/history.html>



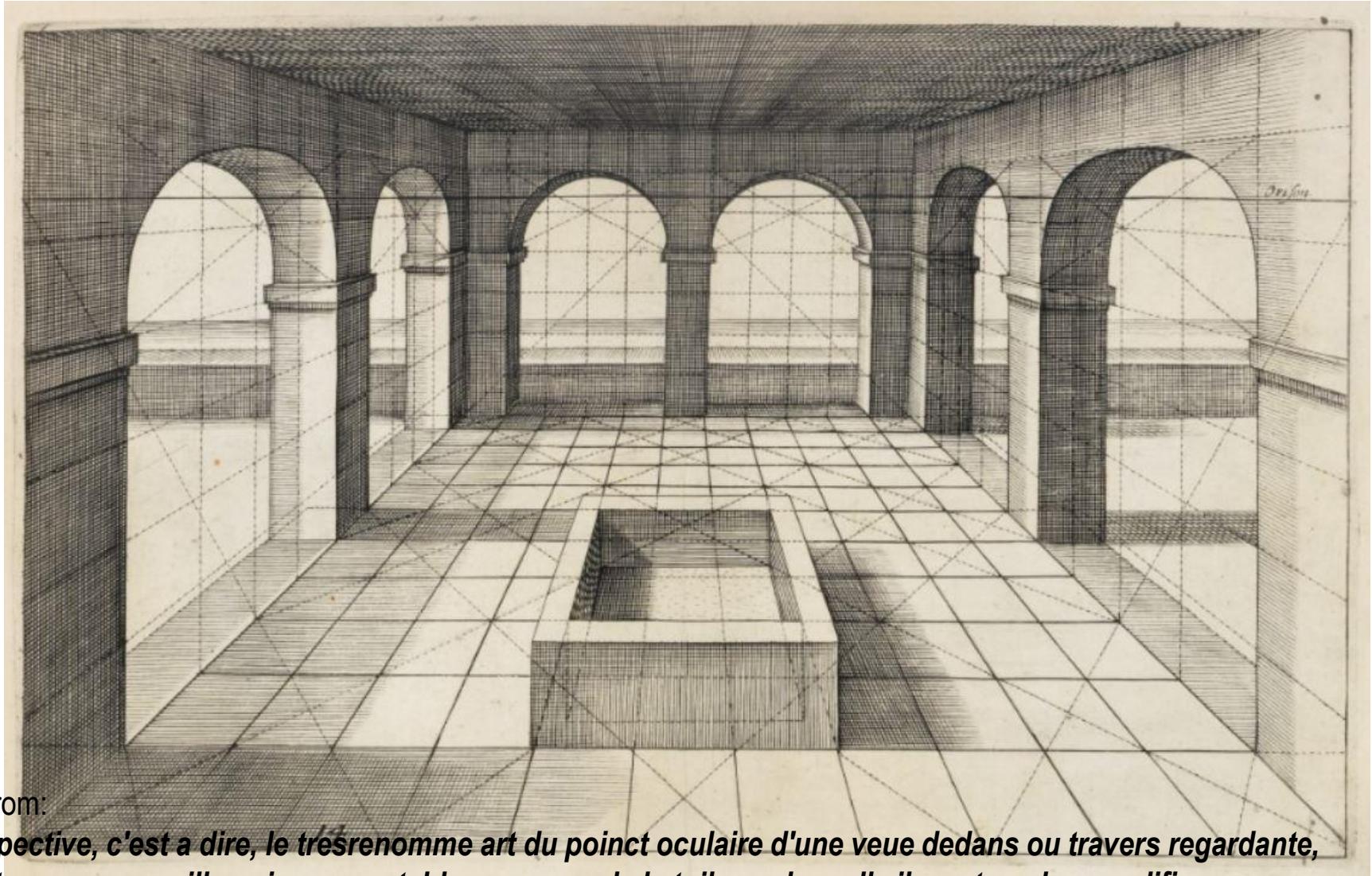
Painter's Tool: Vanishing Points

- All parallel lines *in the viewed scene* converge at one **vanishing point** *in the image display,*
- Idea: choose vanishing points FIRST: draw radiating lines to guide depth depiction



EXAMPLE:

Can you find the Vanishing Point?

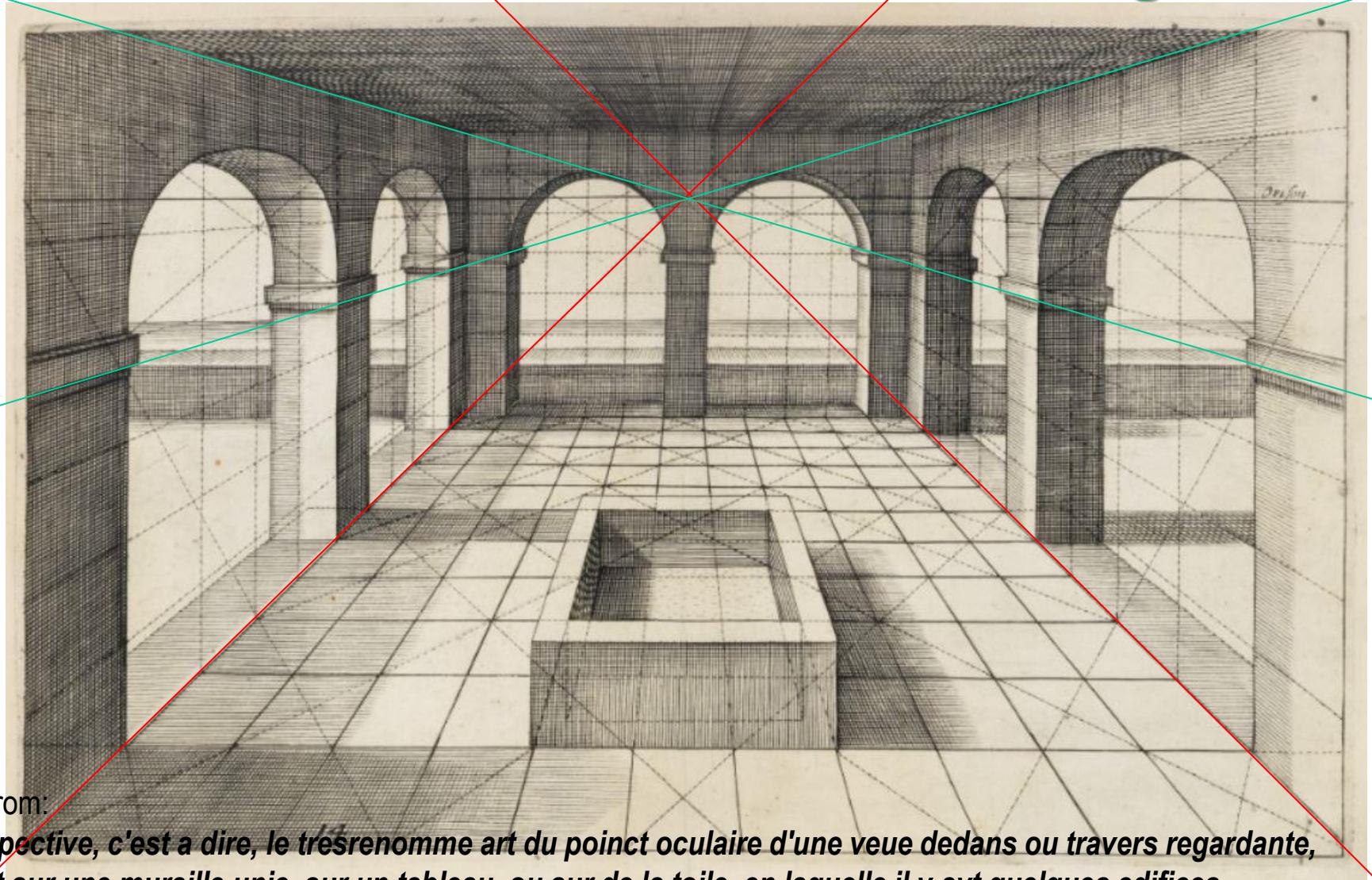


Print from:

"Perspective, c'est à dire, le très renommé art du point oculaire d'une veue dedans ou travers regardante, estant sur une muraille unie, sur un tableau, ou sur de la toile, en laquelle il y ayt quelques edifices, soyt d'églises, temples, palais, sales, chambres, galeries, places, allees, jardins, marches & rües"

by Vredeman de Vriesm, published 1604–1605: The Hague

Draw Lines on wall at floor and ceiling...



Print from:

"Perspective, c'est a dire, le tresrenomme art du poinct oculaire d'une veue dedans ou travers regardante, estant sur une muraille unie, sur un tableau, ou sur de la toile, en laquelle il y ayt quelques edifices, soyt d'eglises, temples, palais, sales, chambres, galeries, places, allees, jardins, marches & rües"
by Vredeman de Vriesm, published 1604–1605: The Hague

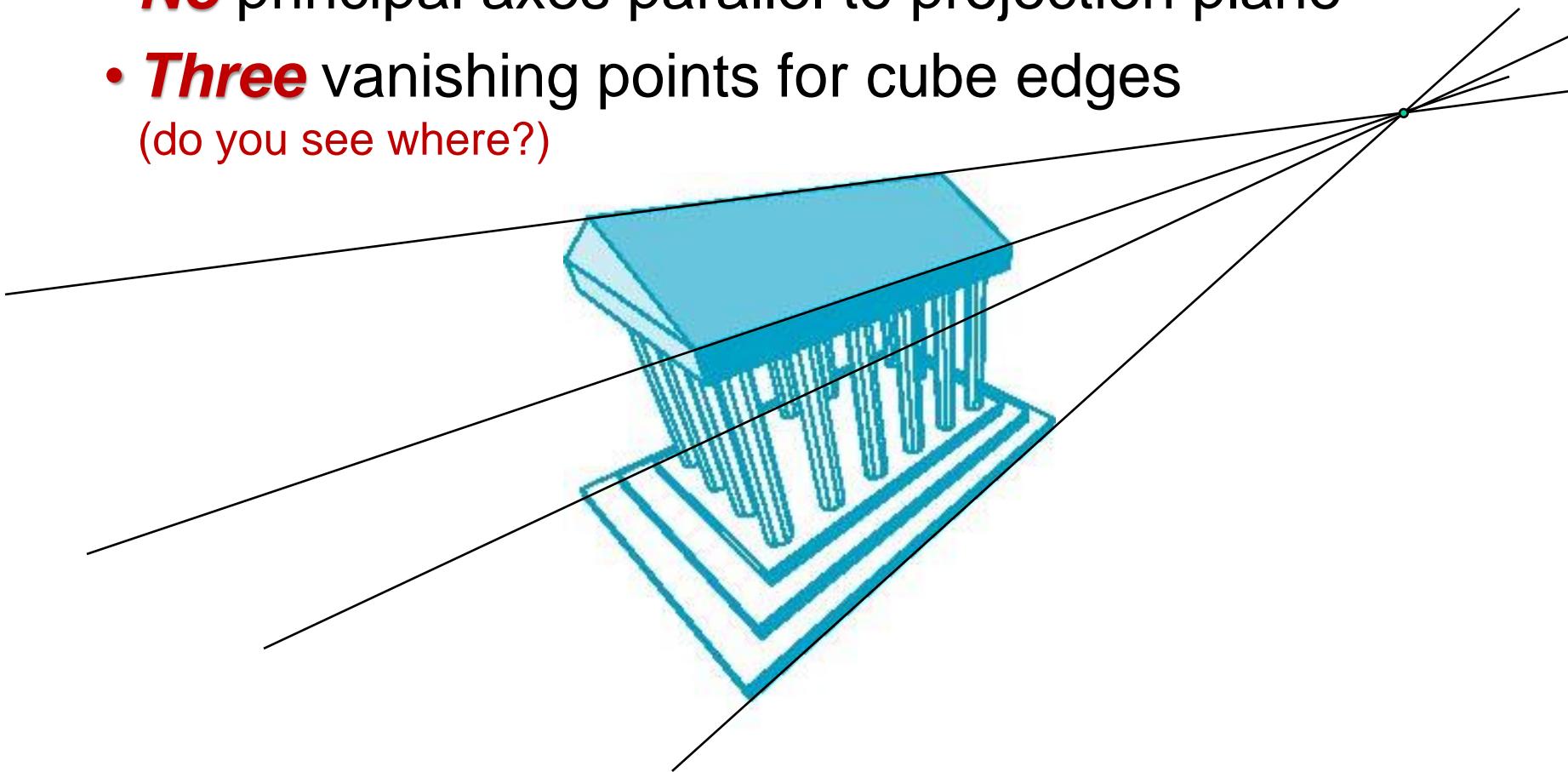
Three-Point Perspective

- **No** principal axes parallel to projection plane
- **Three** vanishing points for cube edges
(do you see where?)



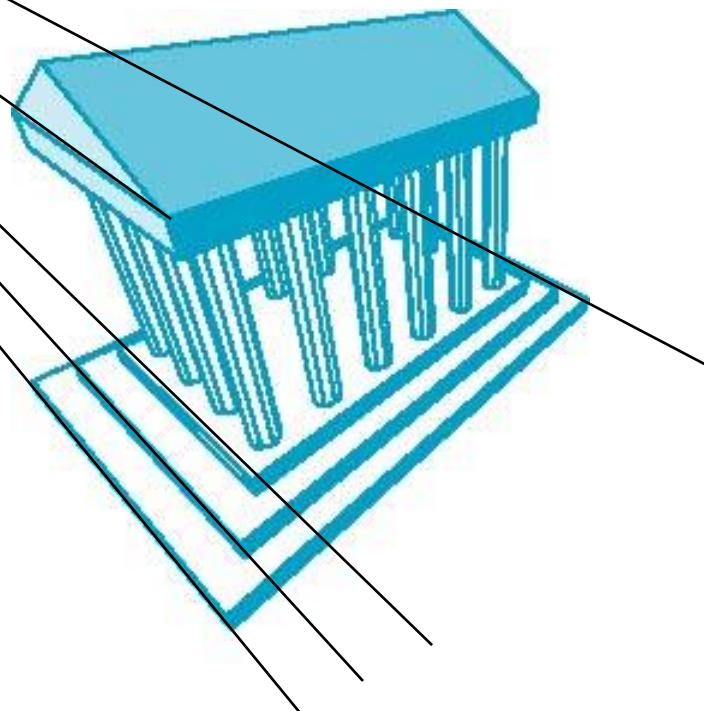
Three-Point Perspective

- **No** principal axes parallel to projection plane
- **Three** vanishing points for cube edges
(do you see where?)



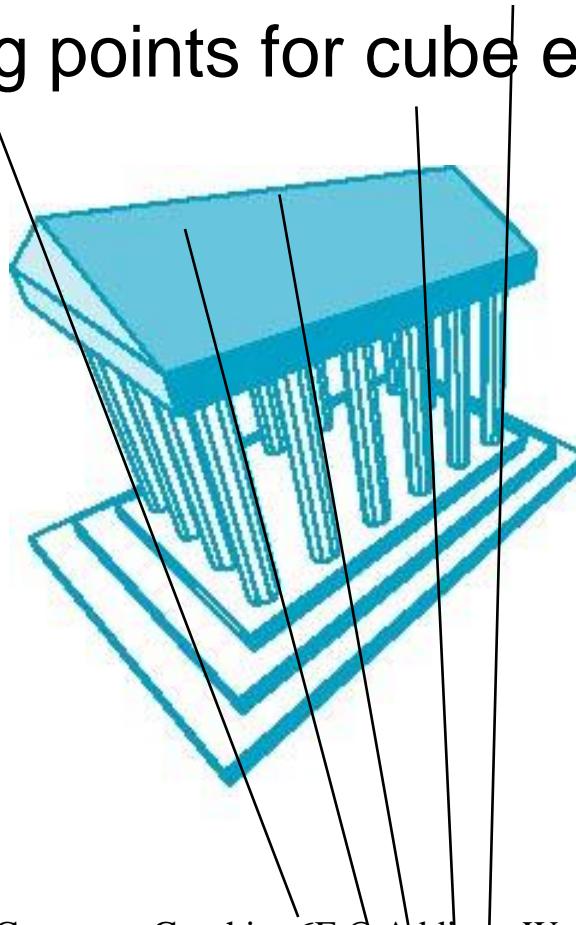
Three-Point Perspective

- **No** principal axes parallel to projection plane
- **Three** vanishing points for cube edges
(do you see where?)



Three-Point Perspective

- **No** principal axes parallel to projection plane
- **Three** vanishing points for cube edges
(do you see where?)

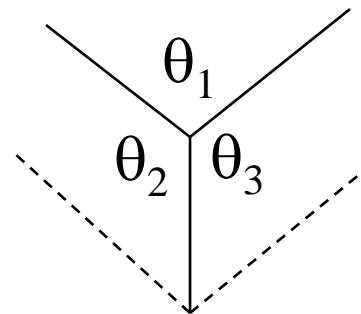


Axonometric Projections

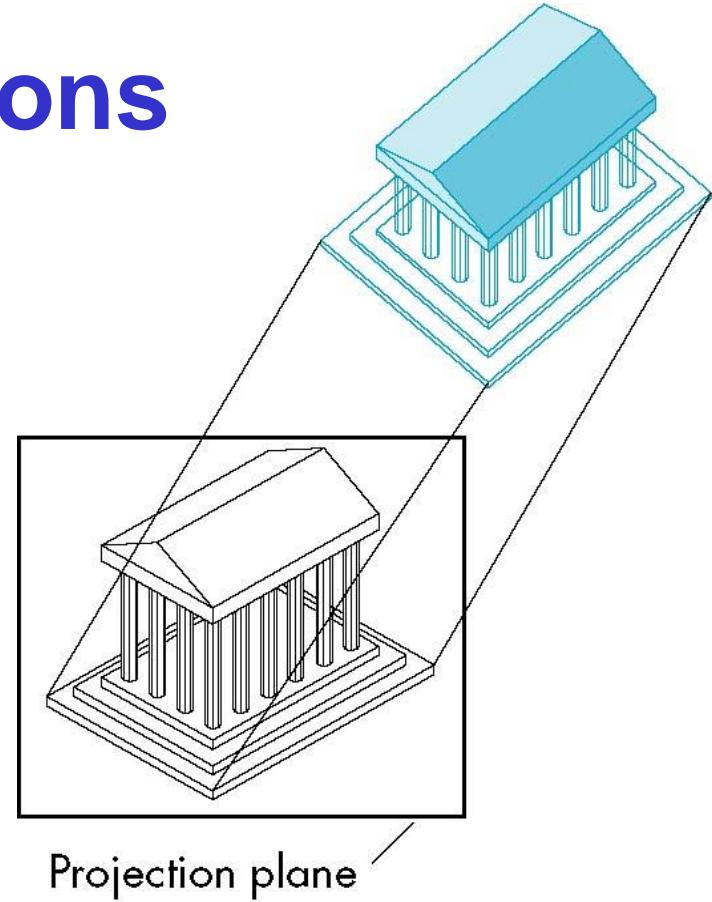
- Vertical **Scene** lines become Vertical **Display** lines;
- Parallel **Scene** lines become Parallel **Display** lines;

- Angles at a cube's corner defines the projections:

- **Isometric**: $\theta_1 = \theta_2 = \theta_3$
- **Dimetric**: $\theta_1 \neq \theta_2 = \theta_3$
- **Trimetric**: $\theta_1 \neq \theta_2 \neq \theta_3$



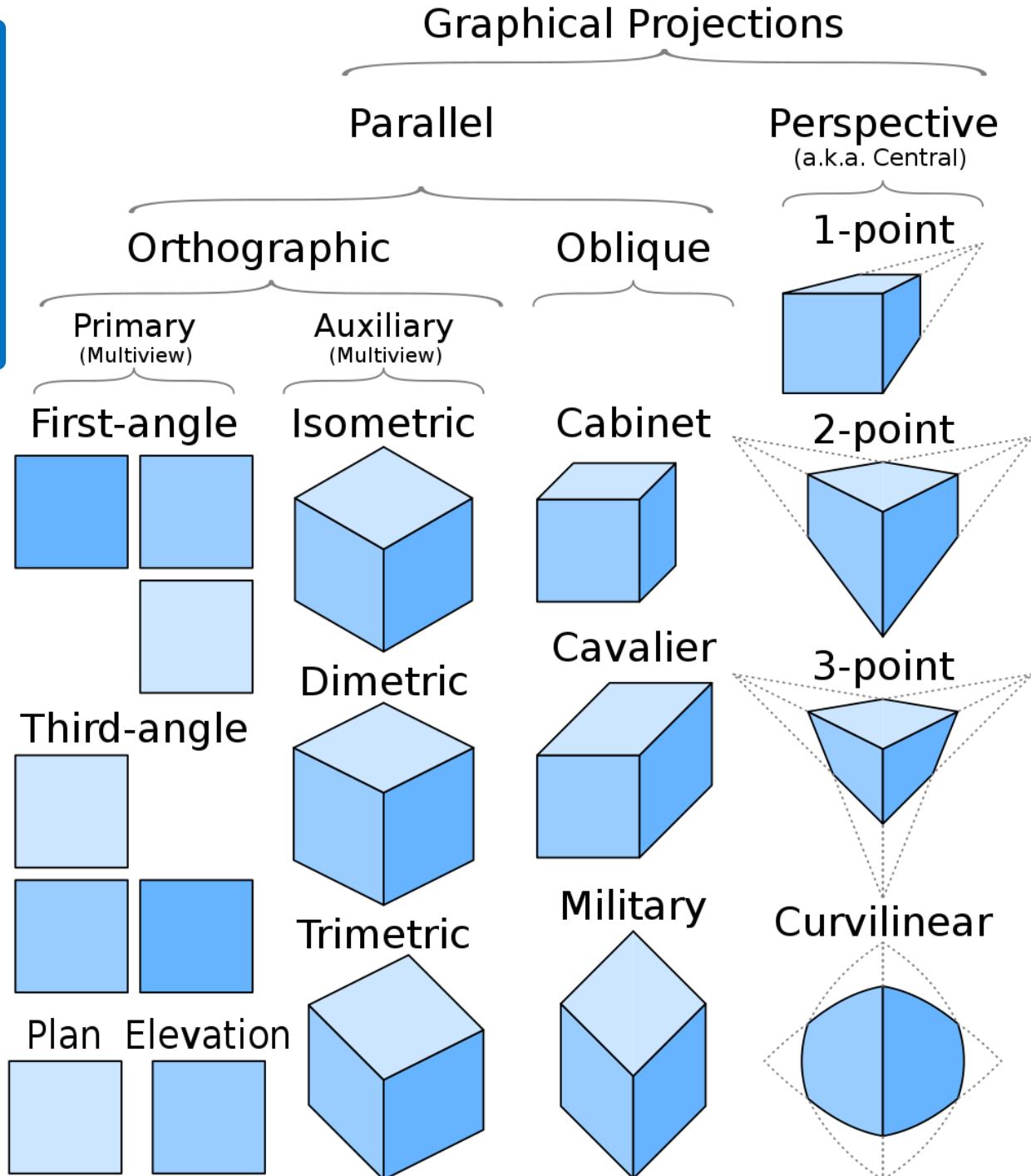
- And that's not all of them! ...



A Taxonomy of Projections

for Traditional
Artists & Illustrators

Wikipedia, “[3D Projection](#)”,
figure by:
Cmglee - Own work,
CC BY-SA 4.0,
[https://commons.wikimedia.org/
w/index.php?curid=83384053](https://commons.wikimedia.org/w/index.php?curid=83384053)



A Taxonomy of Projection

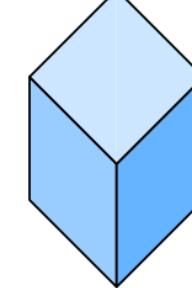
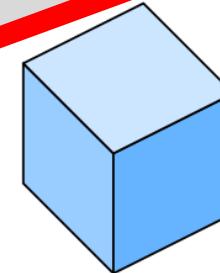
!OH NO! What a mess. Really?

Why all these choices?
Choices missing from our cameras?

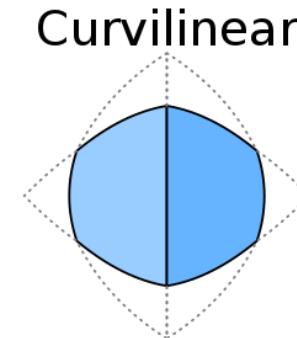
How do we Implement it all?
Select? Debug? Support?
Why is this so complicated? ...

Arti

Wikipedia,
figure by:
Cmglee - Own
CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=10000000>

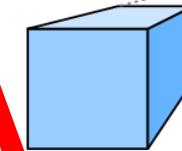


Military

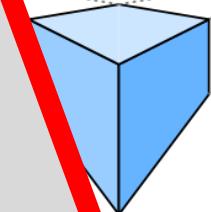


Perspective
(a.k.a. Central)

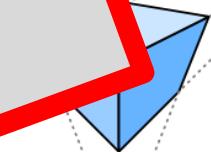
1-point



2-point



point



Classical .vs. Computer Graphics

- **Classical Methods:**

for Viewing / Drawing / Painting: highly developed.
GOOD, CLEVER, ad-hoc ways to draw each one!

BUT

mixes scene contents/axes & **camera** pose with
camera optics. ***Heavily inter-dependent.*** Ugh. **UGH!**

- **Computer Graphics Methods:** **.AS SIMPLE AS POSSIBLE.**

Untangle it! Keep scene & camera separate!
then ALL WE NEED are 4x4 Matrices:

- **Model·View:** (scene contents) · (camera pose):

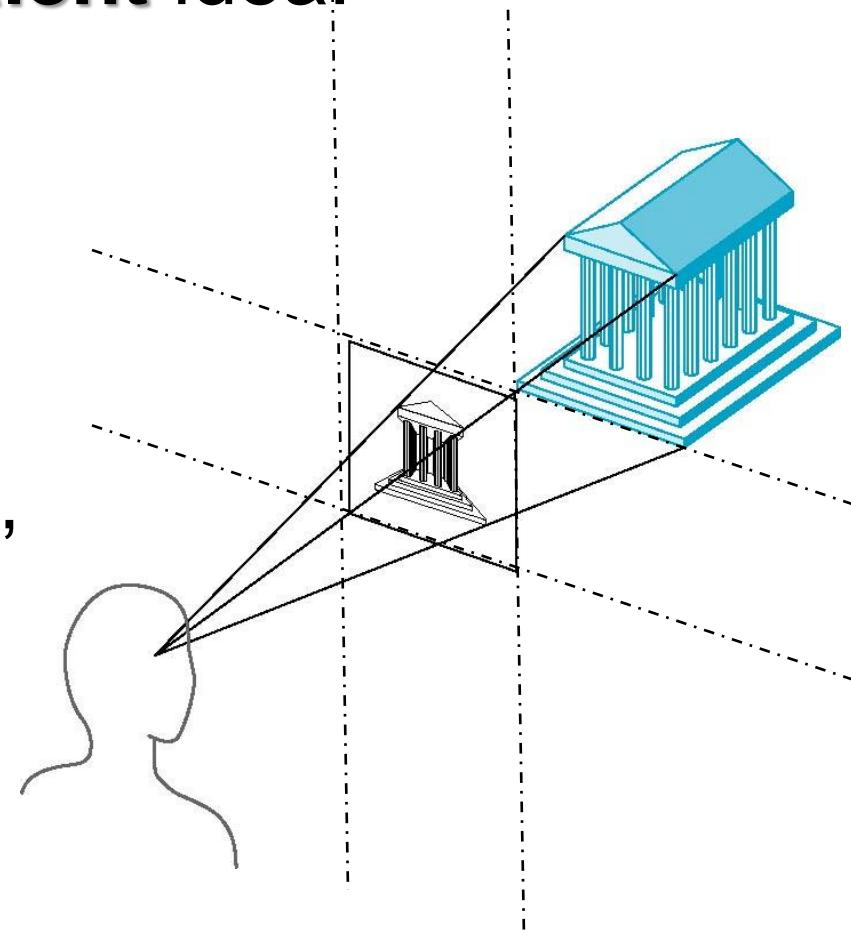
- **Projection:** camera optics only. **Start simply:**

KEY IDEA: Point-Planar Projection

An alternative, **fully-equivalent** idea:

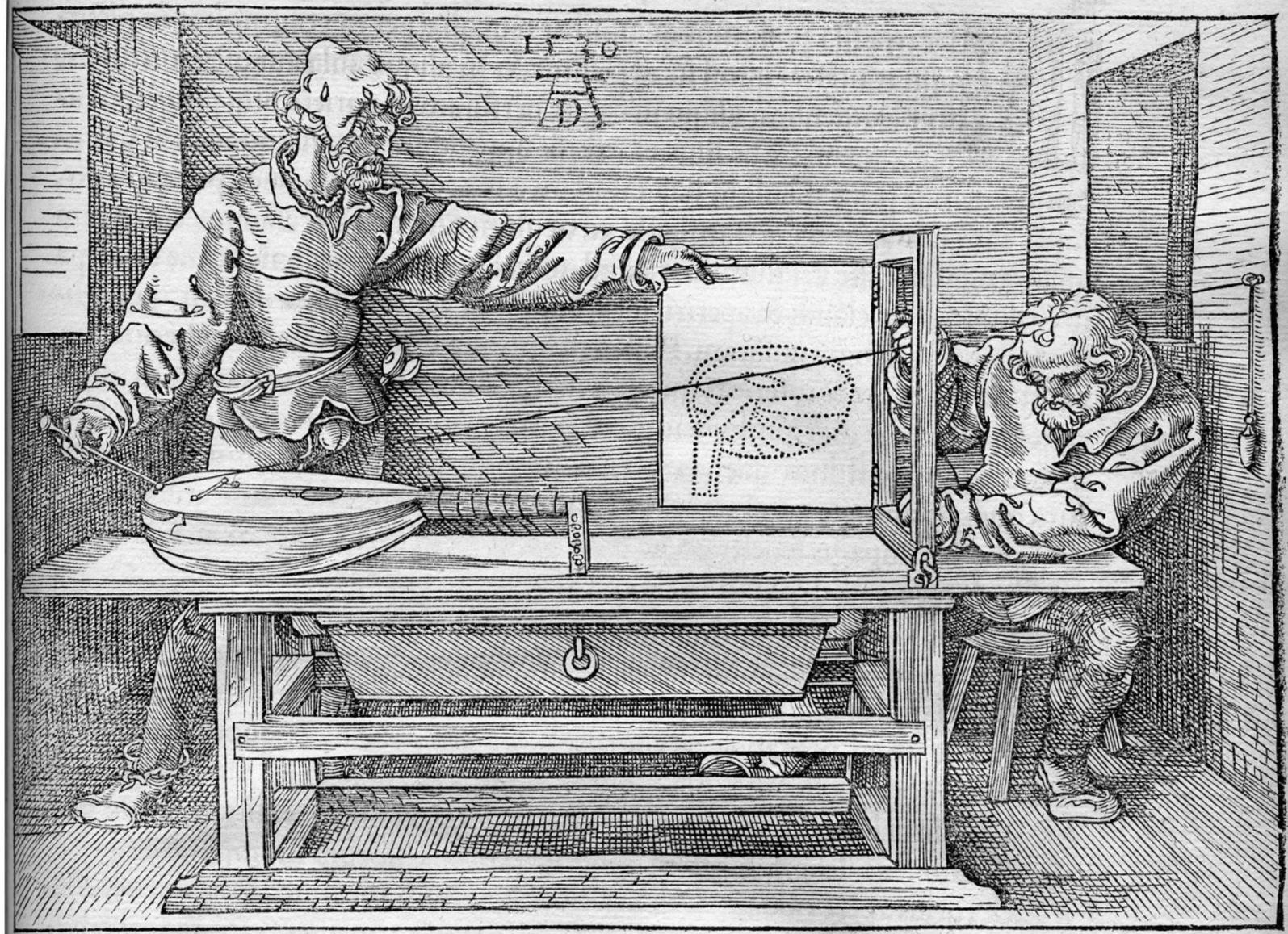
Point-plane perspective:

- Choose a viewpoint,
- Choose a viewing plane,
- Choose a rectangle on that plane as your ‘picture’



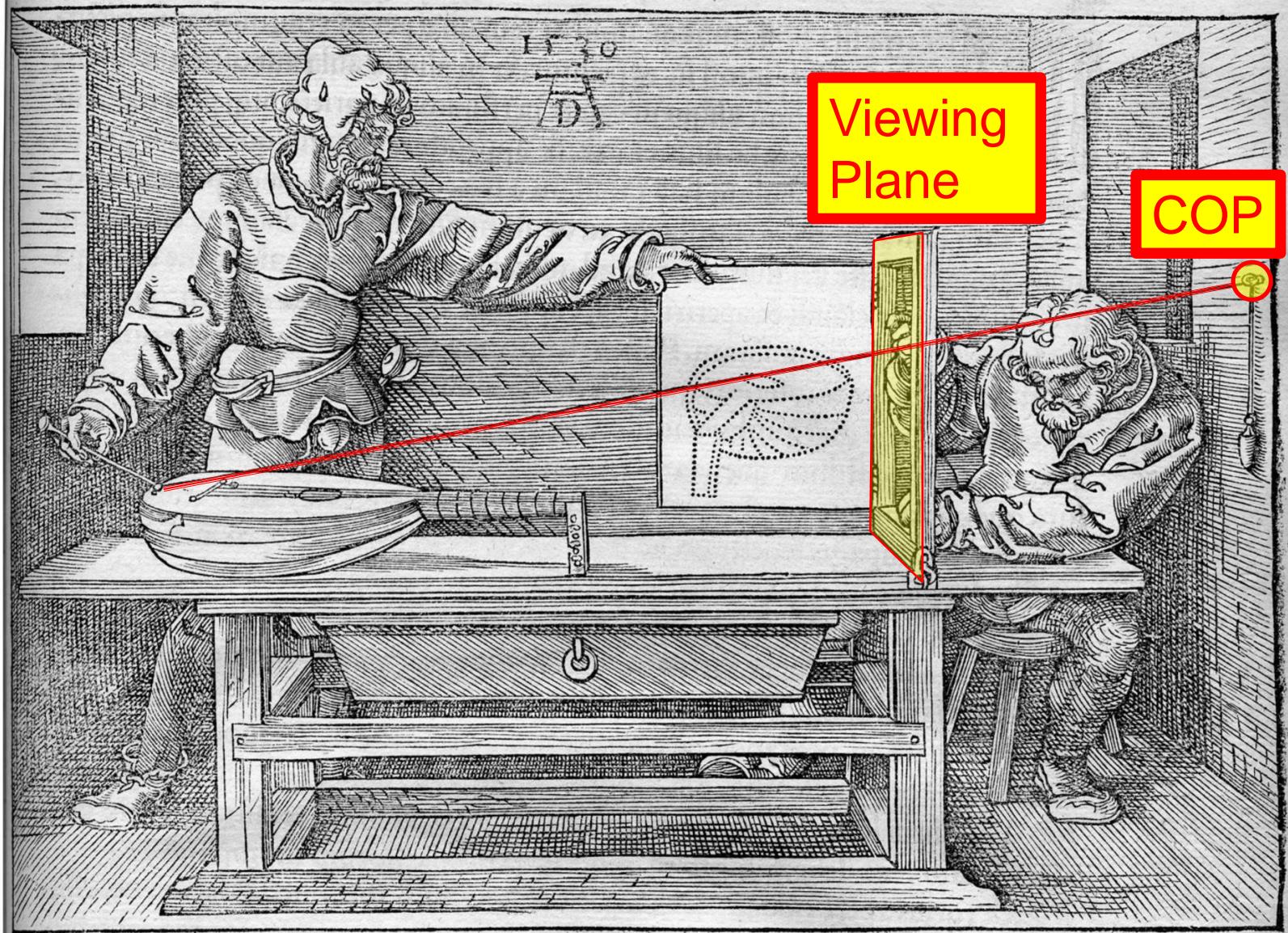
THEN: Viewing rays copy
3D scene colors to plane

Albrecht Durer understood (1530 woodcut)

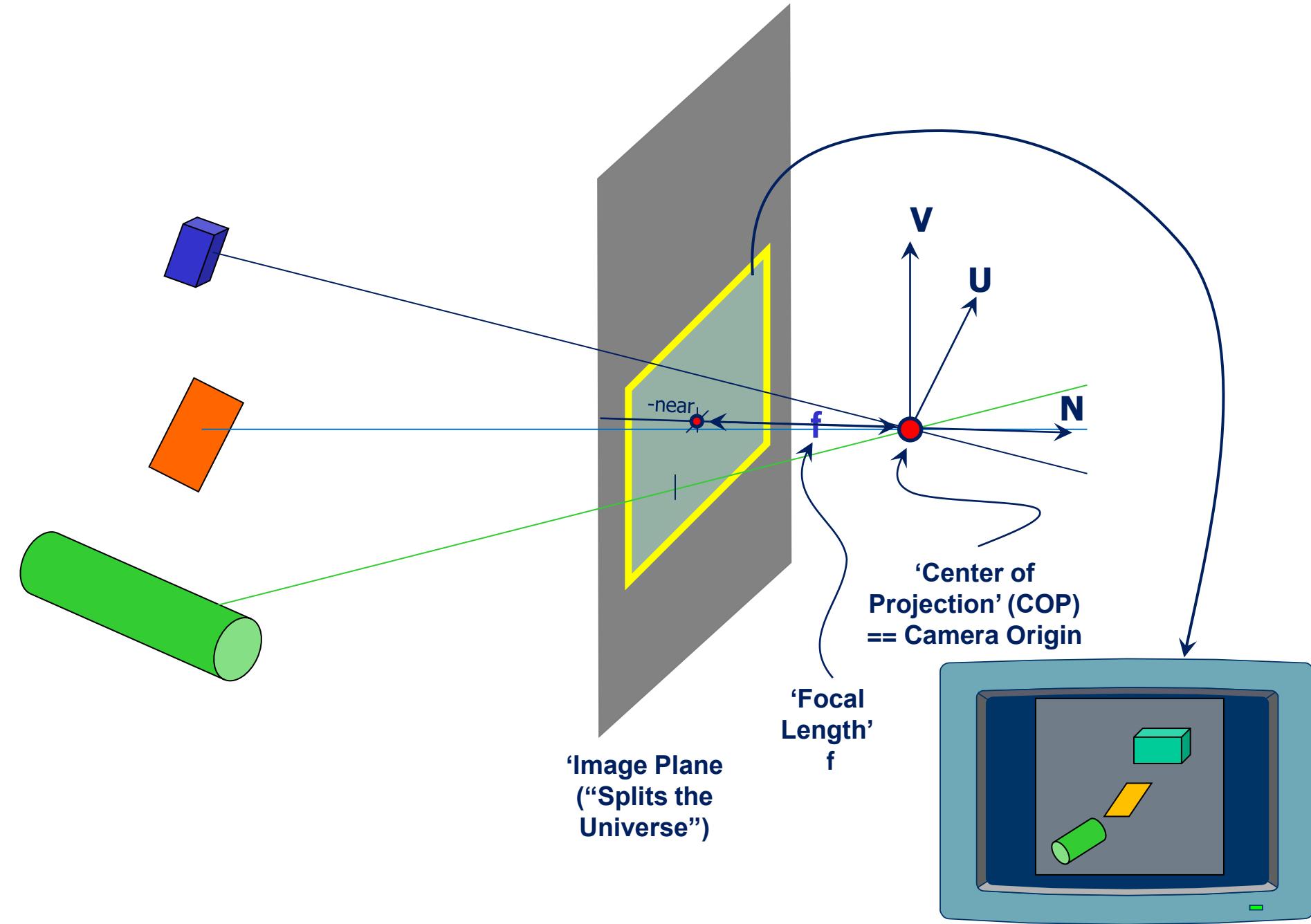


Collection of novel machines for drawing & drafting: <https://drawingmachines.org/index.php> (I built a harmonograph --- can you?)
<https://www.gettyimages.com/detail/illustration/albrecht-d%BCrer-perspective-machine-for-royalty-free-illustration/504822919>

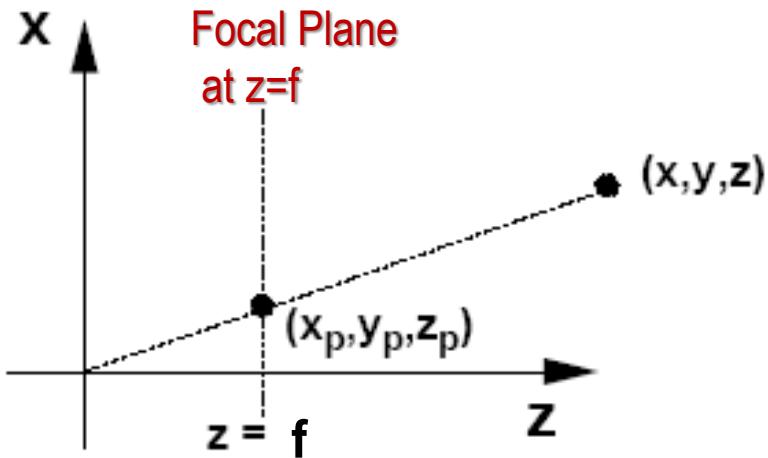
Viewing Plane + Point + Rays



Collection of novel machines for drawing & drafting: <https://drawingmachines.org/index.php> (I built a harmonograph --- can you?)
<https://www.gettyimages.com/detail/illustration/albrecht-d%BCrer-perspective-machine-for-royalty-free-illustration/504822919>



Perspective Projection: Just Divide by z!



What are coordinates of projected point x_p, y_p, z_p ?

(Steve Seitz, probably)

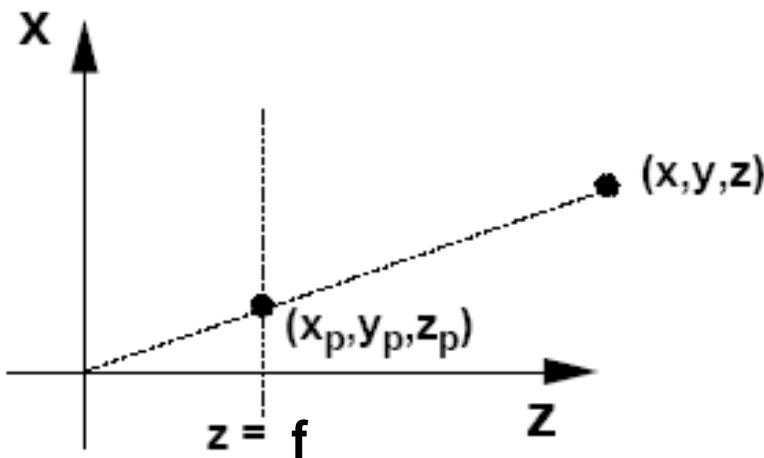
By similar triangles,

$$\frac{x_p}{f} = \frac{x}{z} \quad \frac{y_p}{f} = \frac{y}{z}$$

Multiplying through by f yields

$$x_p = \frac{f \cdot x}{z} = \frac{x}{z/f} \quad y_p = \frac{f \cdot y}{z} = \frac{y}{z/f} \quad z_p = f$$

Perspective Projection: Just Divide by z!



What are coordinates of projected point x_p, y_p, z_p ?

(Steve Seitz, probably)

By similar triangles,

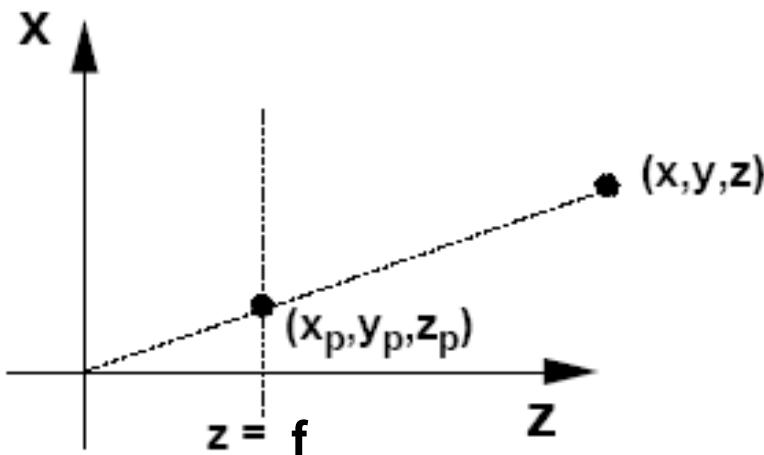
$$\frac{x_p}{f} = \frac{x}{z} \quad \frac{y_p}{f} = \frac{y}{z}$$

Multiplying through by f yields

$$x_p = \frac{f \cdot x}{z} = \frac{x}{z/f} \quad y_p = \frac{f \cdot y}{z} = \frac{y}{z/f} \quad z_p = f$$

Puzzle: What's (x_p, y_p, z_p) when $f = 0$?

Perspective Projection: Just Divide by z!



What are coordinates of projected point x_p, y_p, z_p ?

(Steve Seitz, probably)

By similar triangles,

$$\frac{x_p}{f} = \frac{x}{z} \quad \frac{y_p}{f} = \frac{y}{z}$$

Mul

ANSWER:

Camera with $f = 0$ undefined;

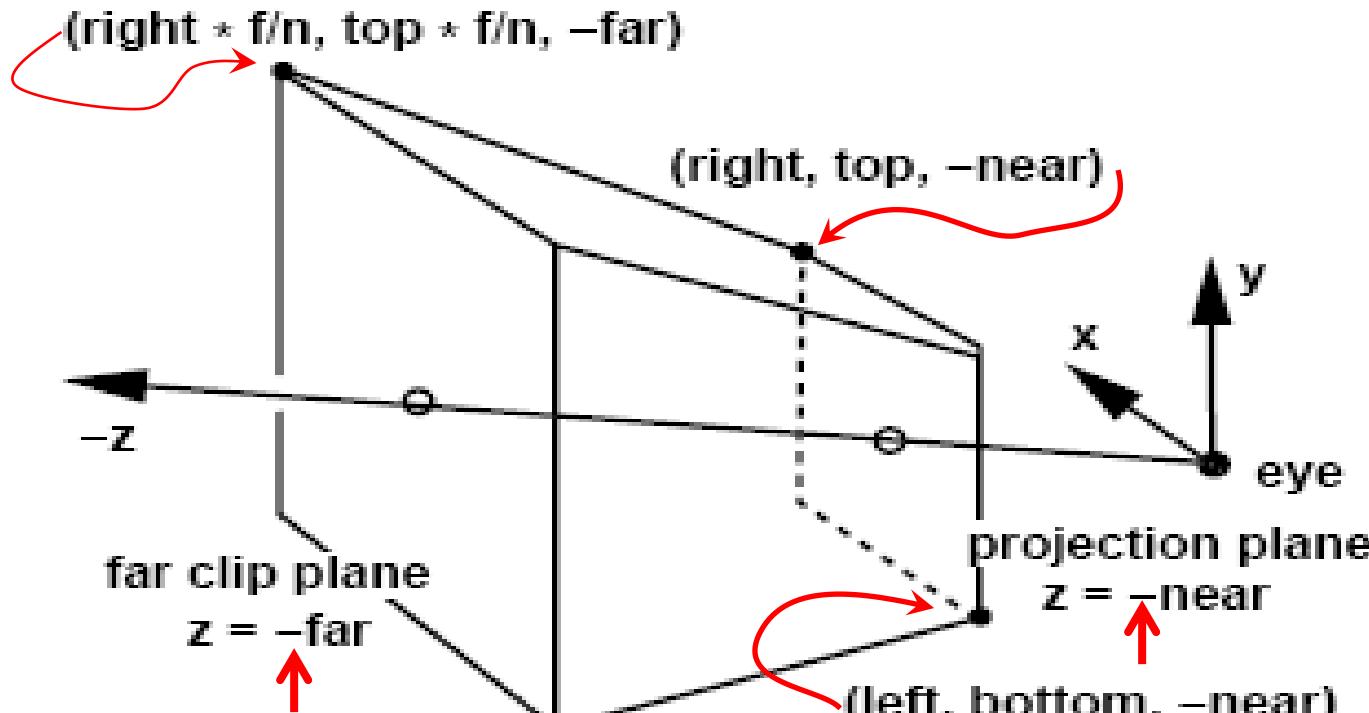
similarly, perspective drawing of point $(x, y, z) = (0,0,0)$ undefined.

- what happens to the image in the limit as $f \rightarrow 0$?
- ‘Perspective’ is well-defined for all other points & f values.
- Divide by zero? !HARDWARE TROUBLE! NaN. Math stops.

What WebGL camera will draw: Contents of ‘Viewing Frustum’

WebGL 3D perspective camera capture a VOLUME (a frustum)

defined by 6 parameters: left, right, bottom, top, near, far



right-handed; view is along $-z$ axis

How does Projection matrix create this ‘frustum’?

Homogeneous → Cartesian Coords

WHY use homogeneous coords?

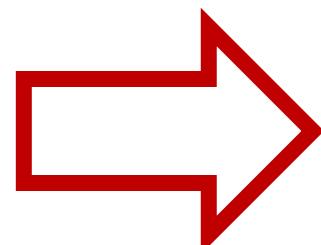
→ For robust 3D perspective cameras

(e.g. the best reason WHY to set $w=1$ for points!)

Homogeneous

Coordinates:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$



Cartesian

Coordinates:

$$\begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}$$

Perspective in Homogeneous Coordinates: (Naïve Form – not used!)

consider $\mathbf{q} = \mathbf{Mp}$ where $\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$

$$\mathbf{q} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \Rightarrow \mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

Perspective Division: Naïve Form

- Note that $w \neq 1$, so we **must divide by w** for Homogeneous \rightarrow Cartesian coordinates
- This ***perspective division*** yields

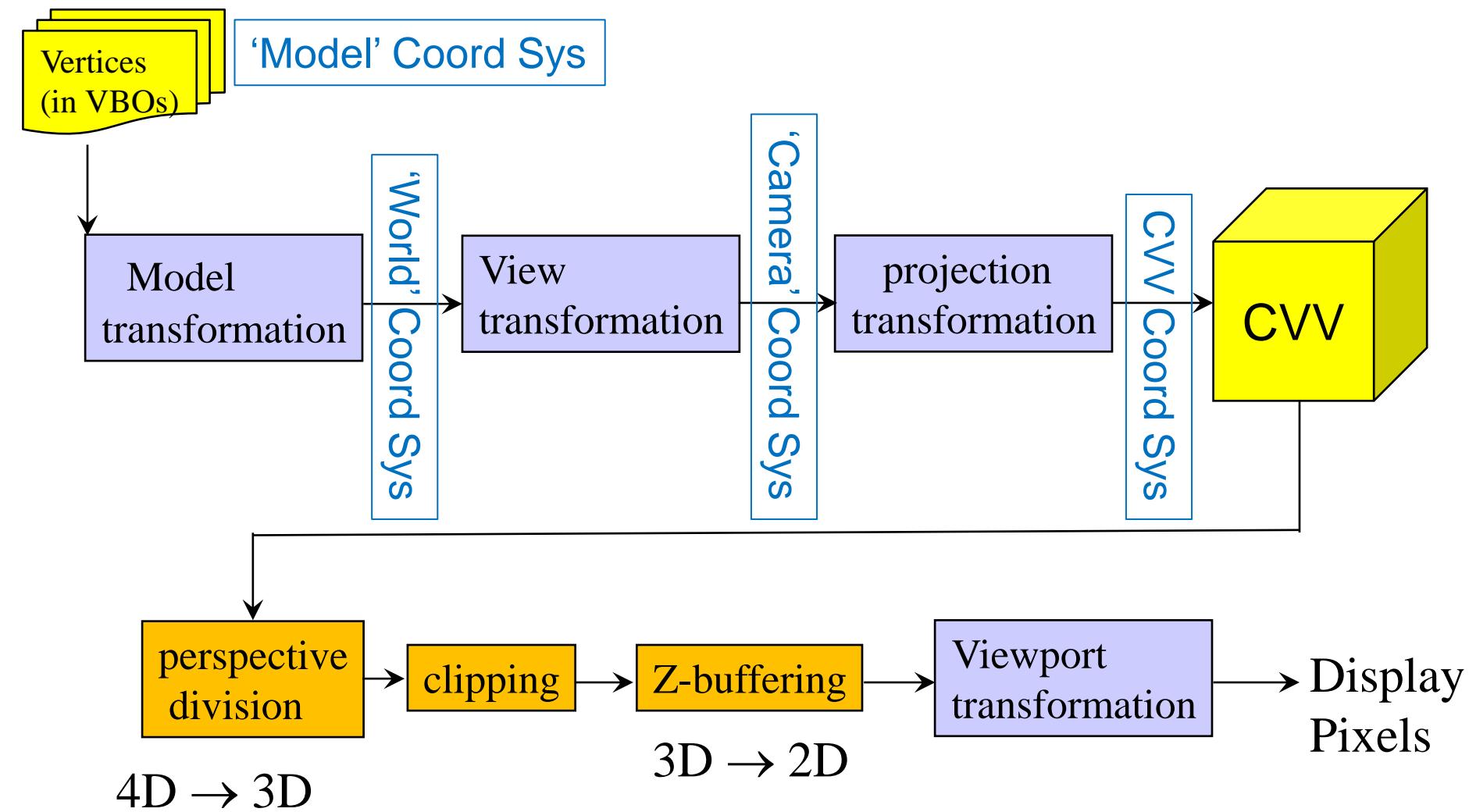
$$x_p = \frac{x}{z/d} \quad y_p = \frac{y}{z/d} \quad z_p = d$$

(if $d==f$, exactly the perspective equations we wanted!)

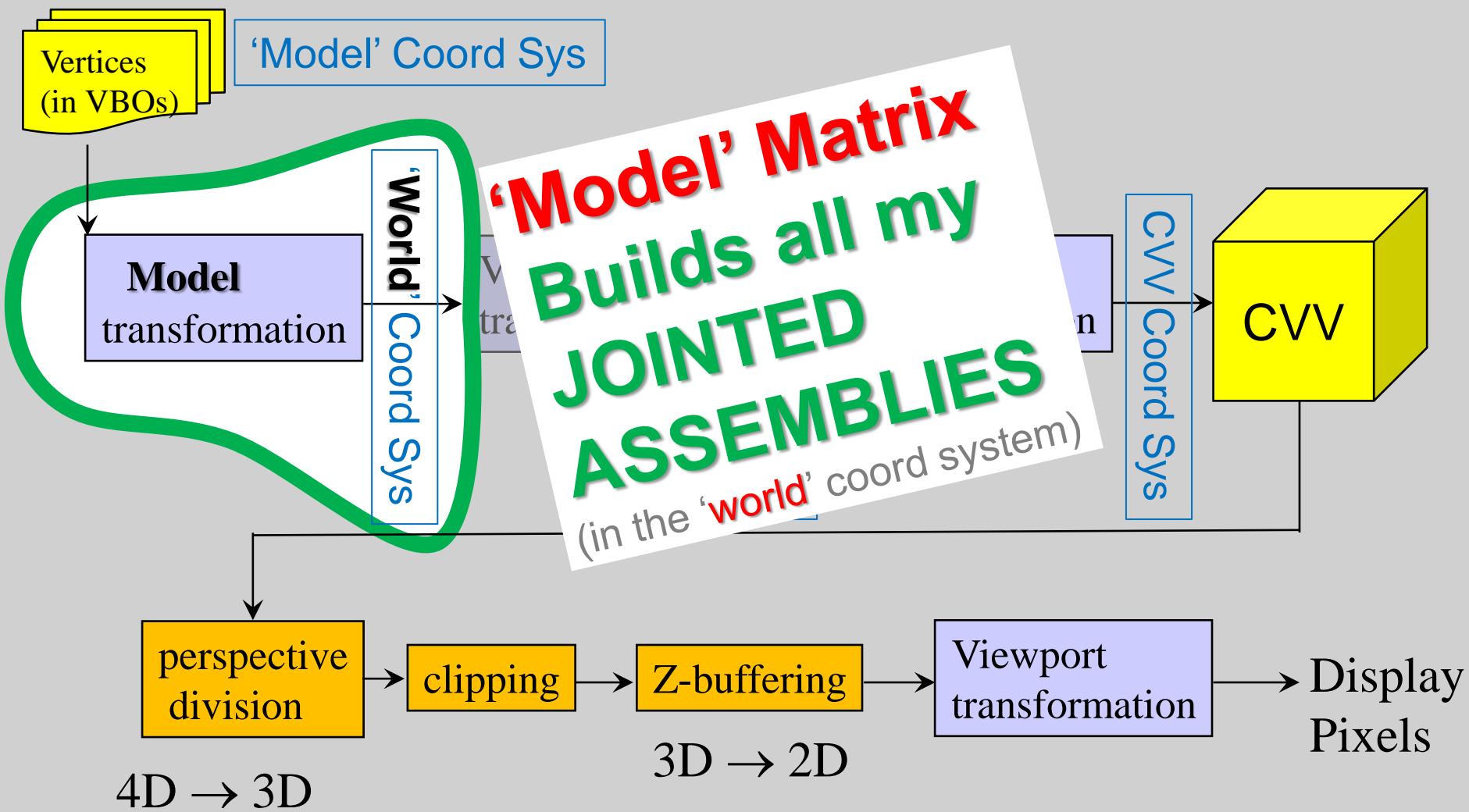
Good Start! How can we make it better?

- What form of ' $1/d$ ' in matrix **M** can include the lens focal length ' f '?
- How can we preserve 'depth' for "depth-buffered" drawing?

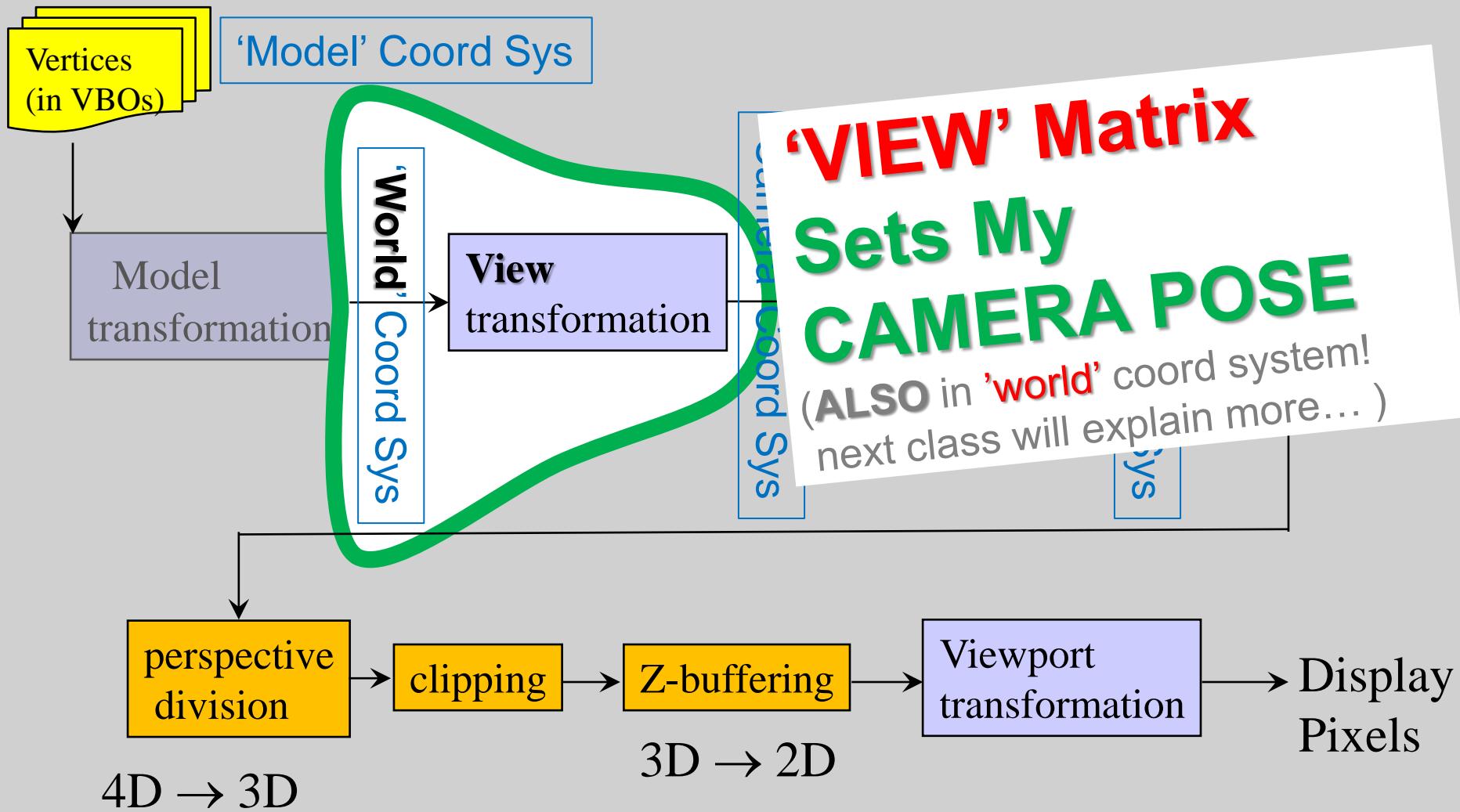
Traditional Vertex Position Pipeline



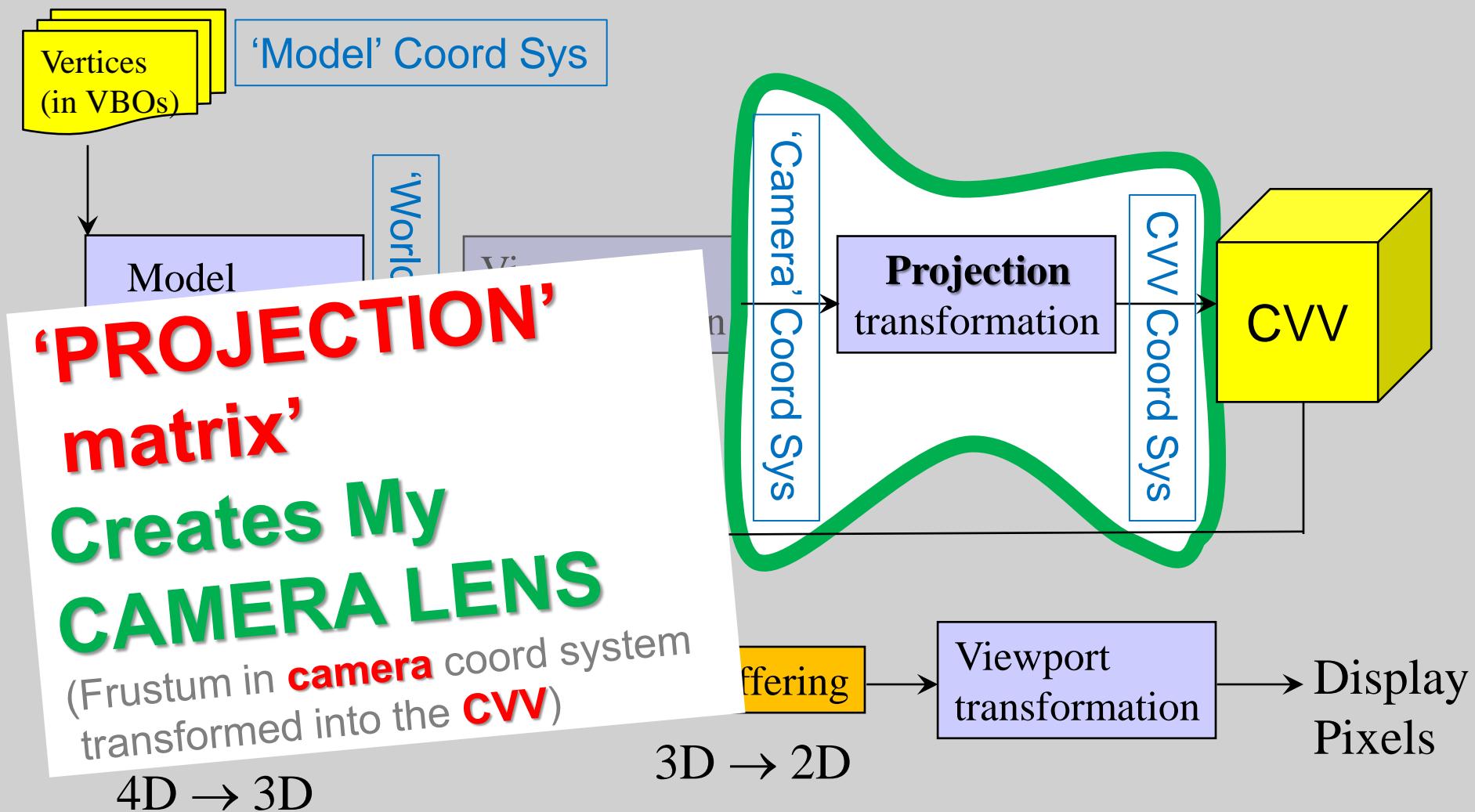
Traditional Vertex Position Pipeline



Traditional Vertex Position Pipeline



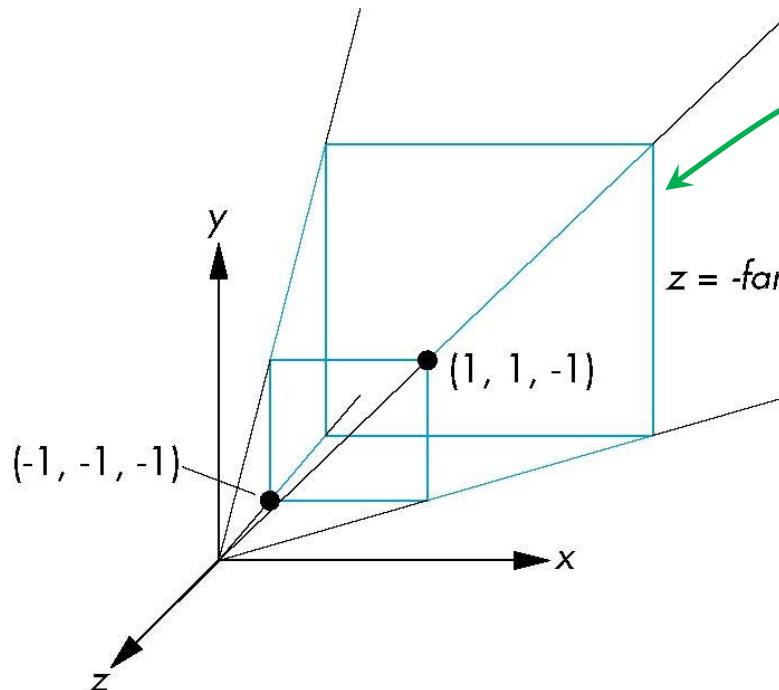
Traditional Vertex Position Pipeline



Point-Perspective Camera

Consider a simple point-perspective image with the COP at the origin, the near clipping plane at $z = -1$, and a 90 degree field of view determined by the planes

$$x = \pm z, y = \pm z$$



Camera's
'Frustum' or
'View Volume',
(3D volume
captured by
Our camera)

Point-Perspective Camera

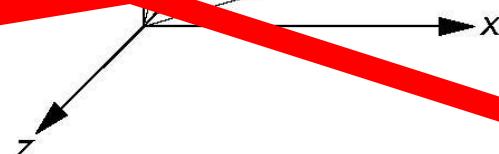
Consider a simple point perspective camera.
the COP is at $(0, 0, 0)$.

$z =$

$b v$

$x =$

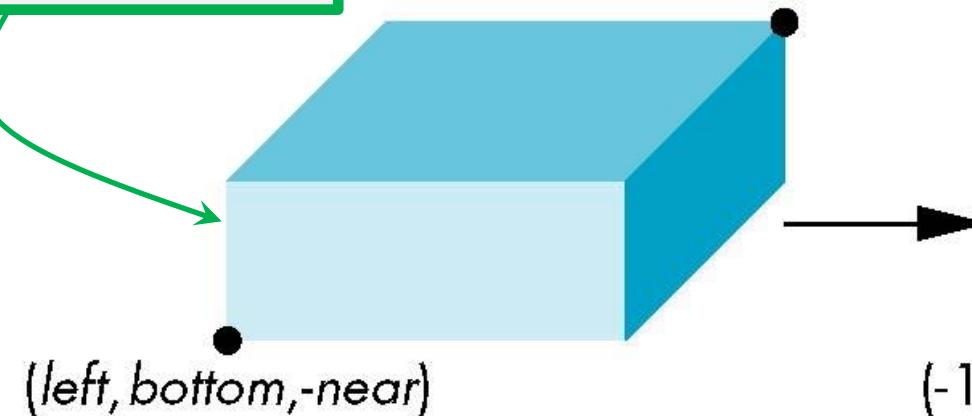
No: Not Yet!
**Try a Simpler
Camera!**



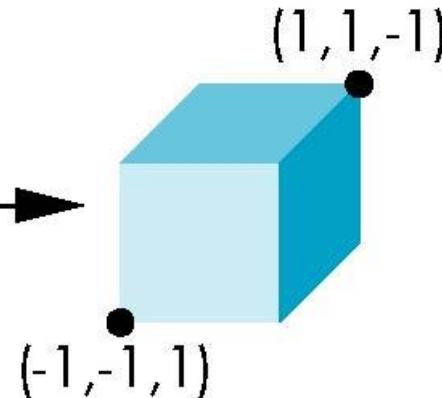
Orthographic Camera

Camera's
**'View
Volume'**
(3D volume
captured by
Our camera)

View Volume in
CAMERA coords:
(right,top,-far)



View Volume
CVV coords:
 $(1, 1, -1)$



Orthographic Camera

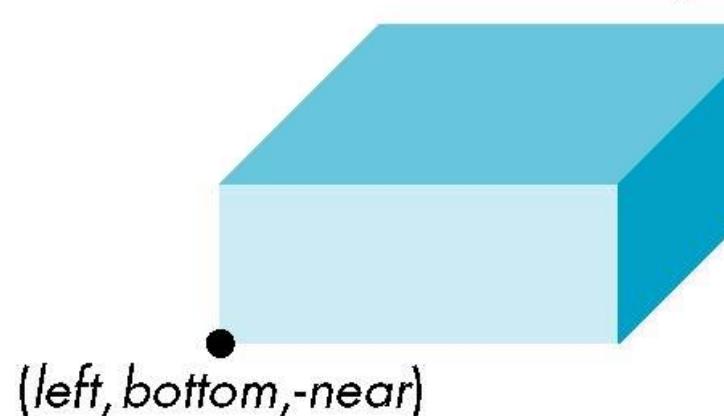
Simple Orthographic **Projection** Matrix:

re-scale a rectangular volume in CAM coords
to fit within the CVV

Matrix4.ortho(left, right, bottom, top, near, far)

View Volume

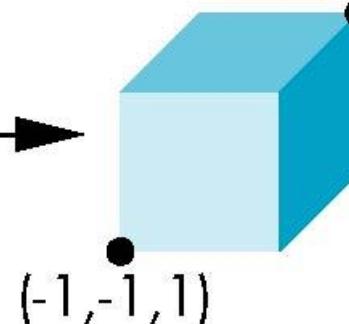
CAMERA coords: (right,top,-far)



View Volume

CVV coords:

$(1, 1, -1)$



How to Make ‘Ortho’ Matrix:

- Two steps:
 - Translate camera’s **frustum center to the origin:**
 $T(-(left+right)/2, -(bottom+top)/2, (near+far)/2))$
 - **Scale to have sides of length 2** (to match the CVV)
 $S(2/(left-right), 2/(top-bottom), 2/(near-far))$

RESULT:

$$\mathbf{P} = \mathbf{ST} = \begin{bmatrix} \frac{2}{right - left} & 0 & 0 & -\frac{right + left}{right - left} \\ 0 & \frac{2}{top - bottom} & 0 & -\frac{top + bottom}{top - bottom} \\ 0 & 0 & \frac{2}{near - far} & \frac{far + near}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

WebGL Key Idea: Display ***only*** the CVV contents!

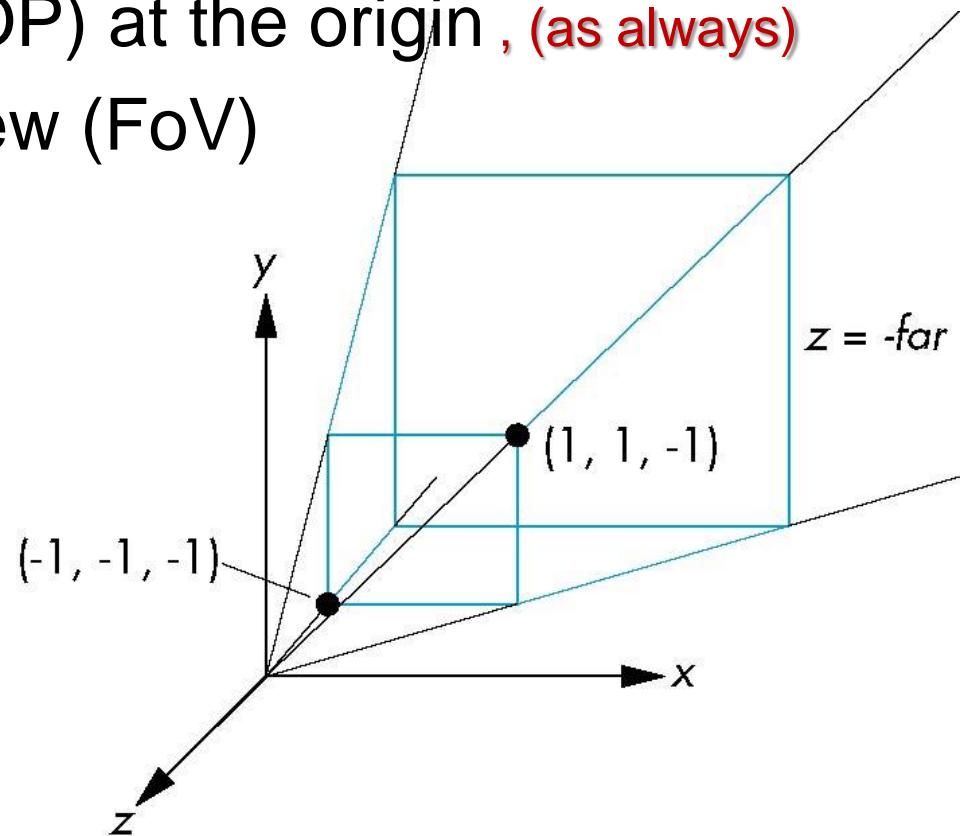
- Keeps 3D clipping simple, fixed and fast
 - *Won't* need different clipping volumes for each kind of frustum, lens or camera:
 - Instead, **map their limits into the CVV** for orthogonal projections onto the display
- RESULT:

- One camera → One matrix;
- One camera type →
One matrix-making function.

Return: Point-Plane Perspective

Try this – a point-perspective image, ****BUT****
aimed at scene in the **MINUS Z AXIS (-Z)**:

- Center of Projection (COP) at the origin , **(as always)**
- $+/-45^\circ$ degree field of view (FoV)
- Image forms on
near clip plane at $z = -1$
- Scene ends at a chosen
far clip plane: $z = -\text{far}$



**Can we make a
matrix for this camera?**

(Naïve) Perspective Matrix

! YES ! Simple ‘-Z’ projection matrix
in homogeneous coordinates:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Homogeneous result:

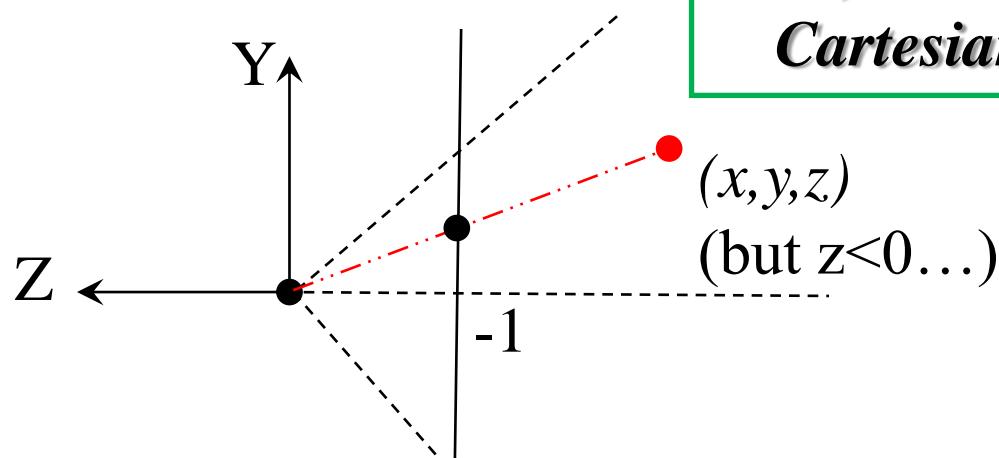
$$x' = x$$

$$y' = y$$

$$z' = z$$

$$w' = -z$$

*So, what's the
Cartesian result?*



(x, y, z)
(but $z < 0 \dots$)

(Naïve) Perspective Matrix

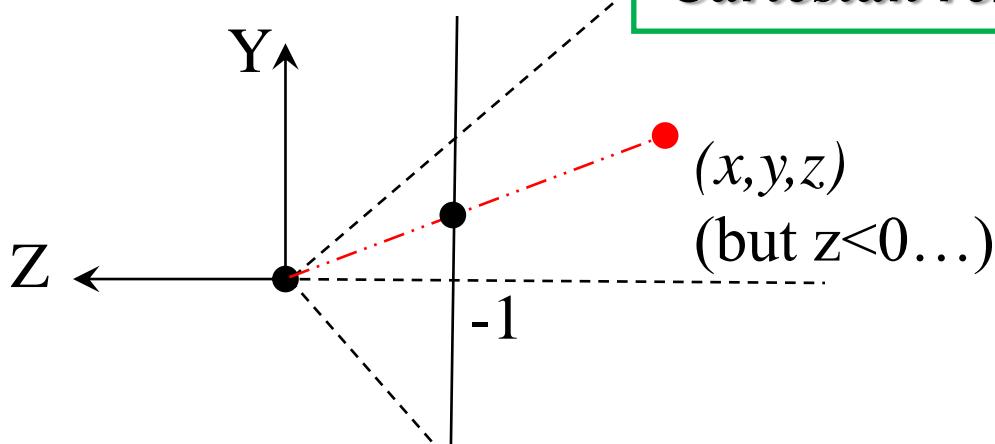
! YES ! Simple ‘-Z’ projection matrix
in homogeneous coordinates:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Homogeneous result:

$$\begin{aligned} x' &= x \\ y' &= y \\ z' &= z \\ w' &= -z \end{aligned}$$

*So, what's the
Cartesian result?*



Cartesian:

$$\begin{aligned} x'' &= x / -z \\ y'' &= y / -z \\ z'' &= -1 \end{aligned}$$

BUT $z < 0$ for
all scene points, so
no sign change for
 $(x, y) \rightarrow (x'', y'')$

(Naïve) Perspective Matrix

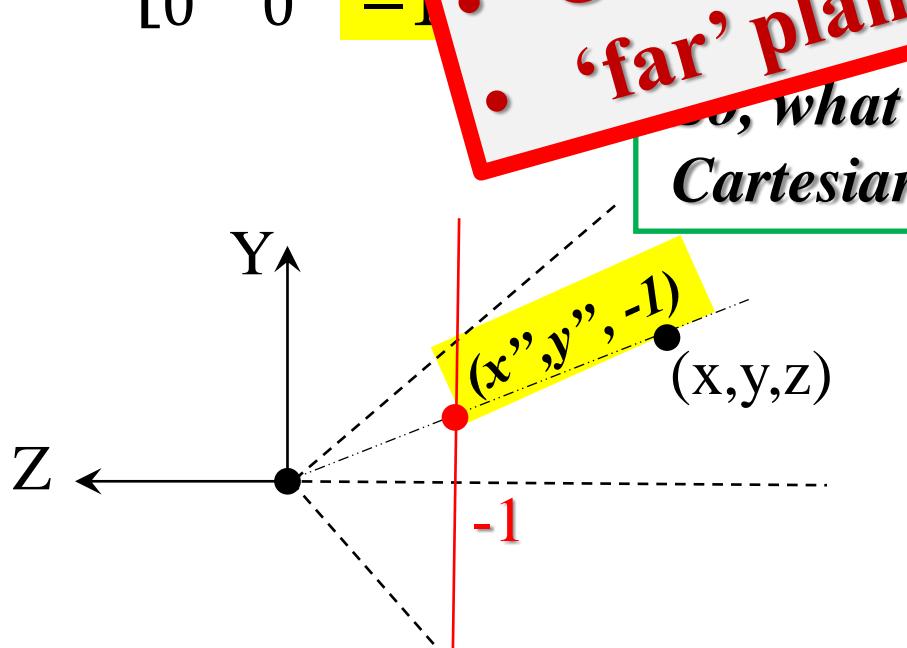
! YES ! Simple ' $-z'$ '

Trouble!

Image forms at $z = -1$;

- ignores 'near' clipping plane, and
- CVV limits us to $+1 \leq z \leq -1$ so
- 'far' plane is ALSO -1 !?!! Ugh!

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}$$



So, what's the
Cartesian result?

$$z'' = -1$$

BUT $z < 0$ for
all scene points, so
no sign change for
 $(x, y) \rightarrow (x'', y'')$

One Weird Trick to fix it All

(perfectionists hate it 😒)

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

after perspective division, the point at $(x, y, -z, 1)$ becomes

$$x'' = x/z$$

$$y'' = y/z$$

$$z'' = -(\alpha z + \beta)/z$$

Choose α and β carefully...

Generalize to fix it:

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

after perspective division, the point at $(x, y, z, 1)$ becomes

$$x'' = x/z$$

$$y'' = y/z$$

$$z'' = -(\alpha z + \beta)/z$$

What Weirdness
is THIS?!??!

Could it span -1 to +1 (as x'' and y'' might)?

Could we fit it into the CVV?

‘Pseudo-Depth’: How to Pick α and β

If we pick

$$\alpha = \frac{\text{near} + \text{far}}{\text{far} - \text{near}}$$

$$\beta = \frac{2\text{near} * \text{far}}{\text{near} - \text{far}}$$

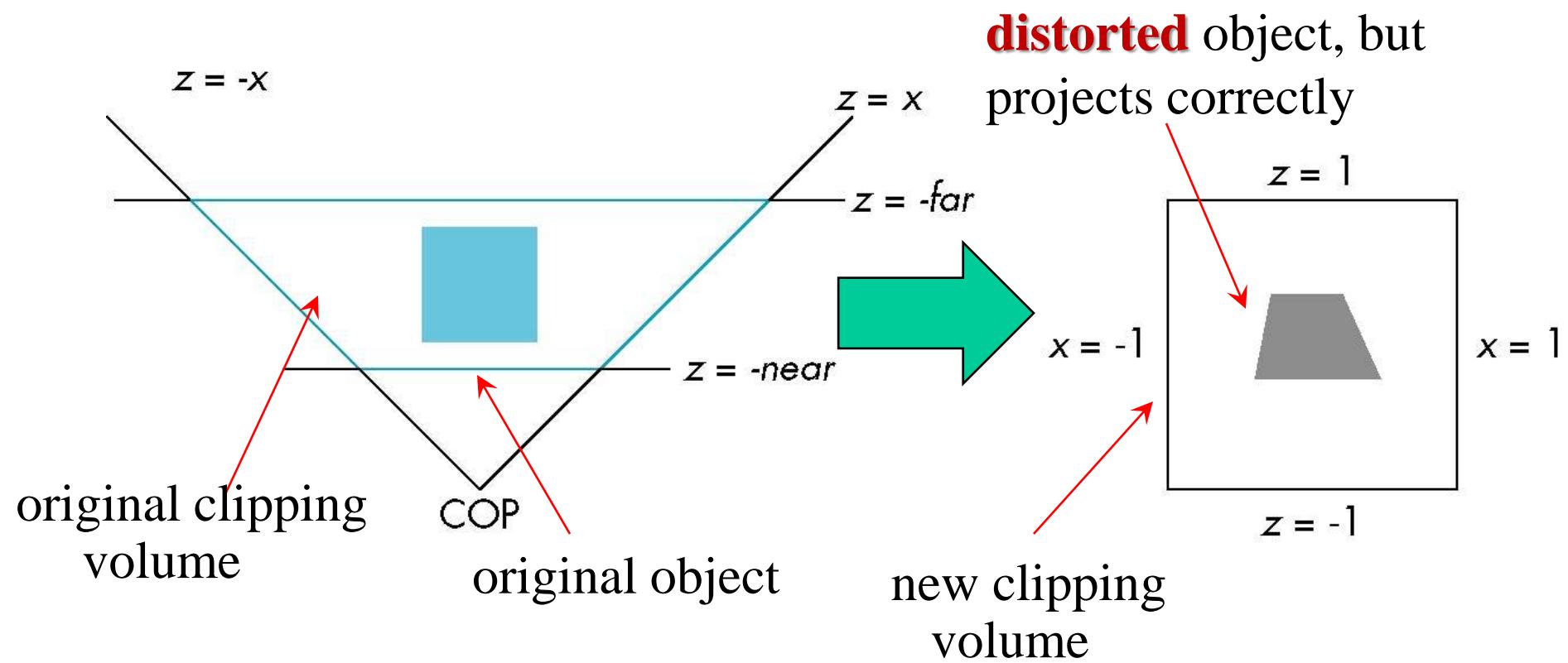
maps the **near plane** to $z = 1$ ('front' of CVV)

maps the **far plane** to $z = -1$ ('back' of CVV)

maps the frustum sides to $x = \pm 1, y = \pm 1$ ('sides' of CVV)

THUS *the viewing frustum fills the CVV exactly!*

Why use this Weird Formulation?



ANS: Normalization, and Hidden-Surface Removal

- Although our '**pseudo-depth**' may SEEM weirdly arbitrary, it **ensures monotonic depth**: if $z_1 > z_2$ in the original clipping volume, then inside the CVV we always have $z'_1 > z'_2$
- Thus hidden surface removal *always* looks right & works perfectly using the 'z' values in the CVV
- *BUT* this '**pseudo-depth**' **DISTORTS** distances. In the CVV, it compresses far-away depths and expands nearby depths: $z'' = -(\alpha z + \beta) / z$
(and it leaves x,y perspective unchanged: $x'' = x/z$; $y'' = y/z$)
- Tiny **z_{near}**? Giant **z_{far}**?
Expect **coarse** depth quantization !

END

END



Virtual Cameras: An Overview

Some of these are

J.Tumblin-Modified SLIDES from:

- Ed Angel, Professor Emeritus
of Computer Science, University of New Mexico
- Peter Shirley (formerly) Professor, School of
Computing, University of Utah School
(now Principal Research Scientist, nVidia)
- Both published good books on graphics...



The University of New Mexico

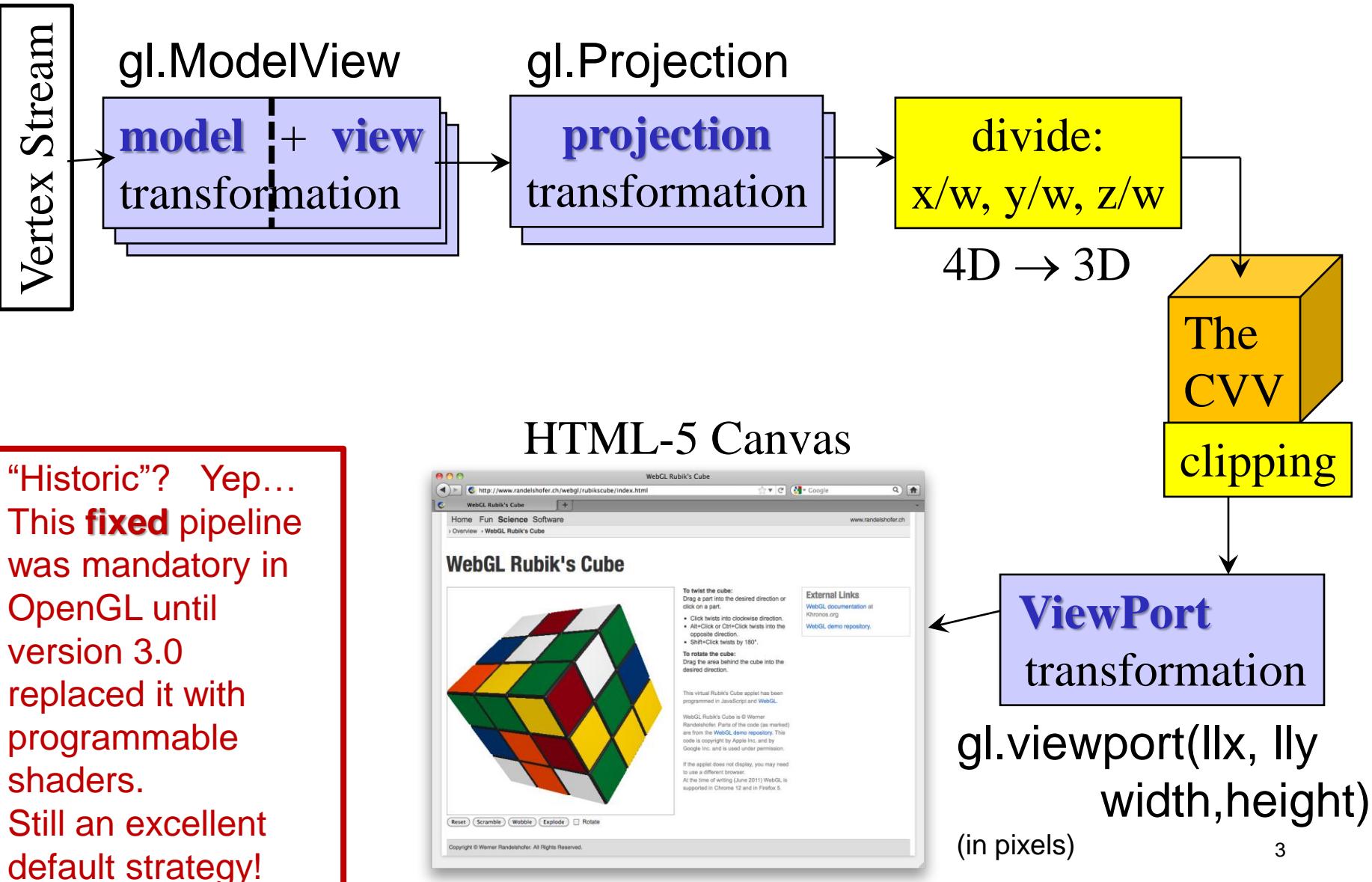
3D Graphics System Transforms: Overview

- The default camera == none == CVV is our 3D ‘world’.
- Most 3D graphics systems now share a common design:
- Same *matrix set* for all objects, views, cameras, displays:

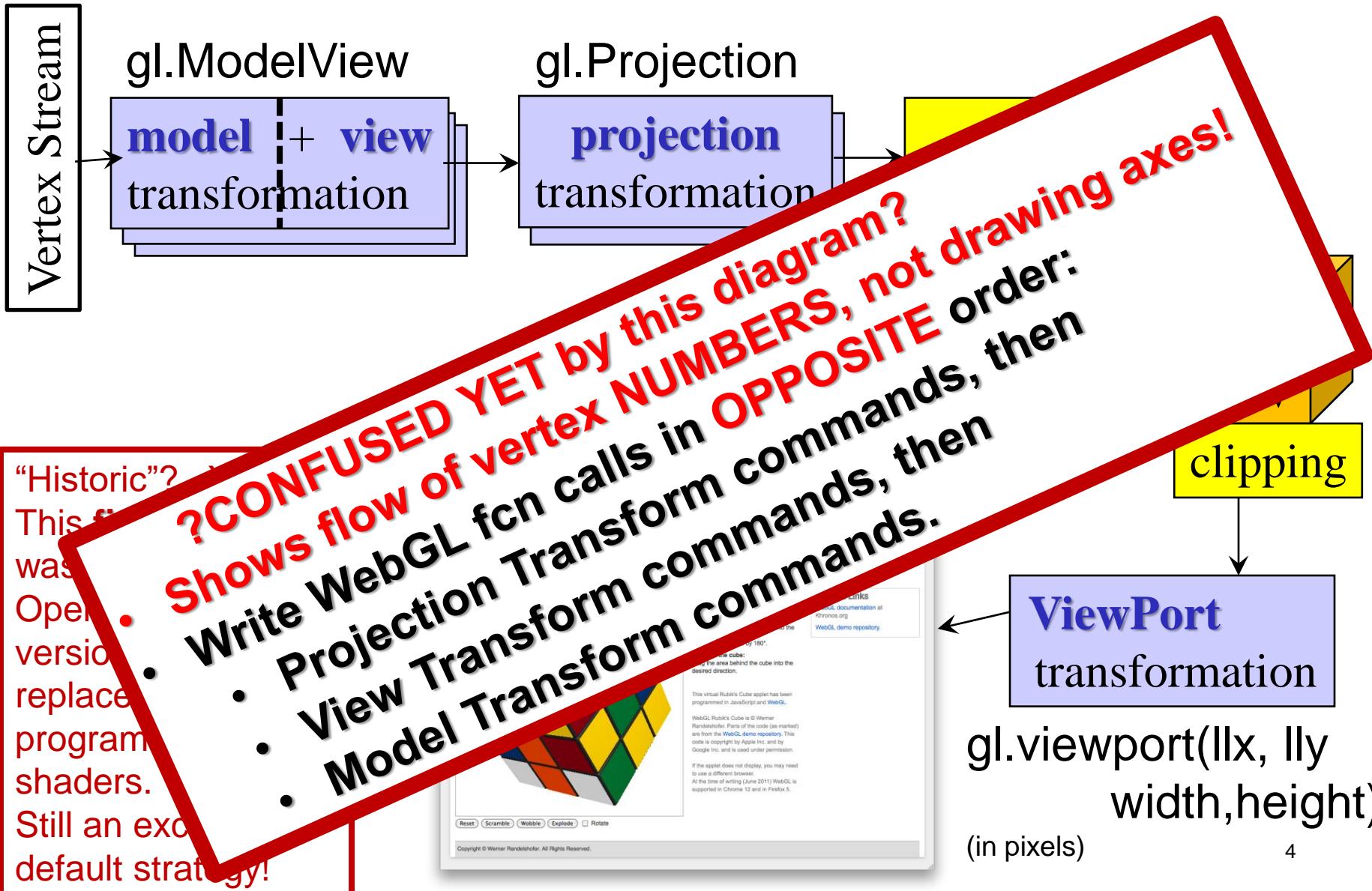
Model → *View* → *Projection* → **CVV** → *Viewport* → *Display*

- Just 4 pipelined matrices:
 - ‘**Model**’ matrix assembles & positions parts in 3D world
 - ‘**View**’ matrix positions camera in the 3D world
 - ‘**Projection**’ matrix fits camera’s 3D frustum into the CVV
 - ‘**Viewport**’ matrix maps CVV contents to display screen

Historic Vertex Position Pipeline



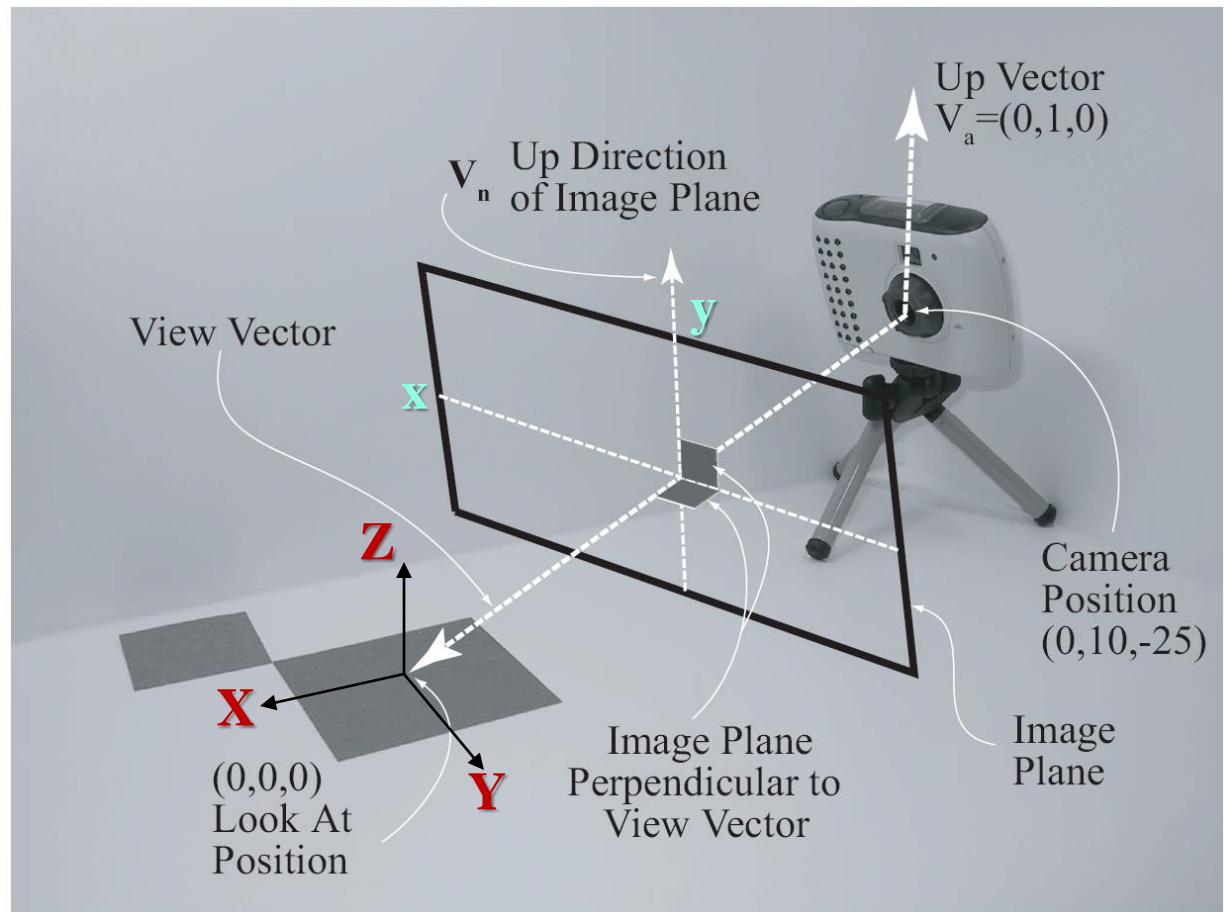
Historic Vertex Position Pipeline



View Matrix: Sets the Camera Pose

TERMINOLOGY:

- Camera position
- 'Aim-point' or 'Look-At' position
- 'Up' direction
- Related terms:
 - Image Plane
 - Viewing Direction
 - View Vector



Basic Shapes: Can we ‘Spin the World’?

- Screen coords: X,Y define 2D window into a virtual 3D world.
- World coordinates (typical): +Z==UP, X,Y==horizontal

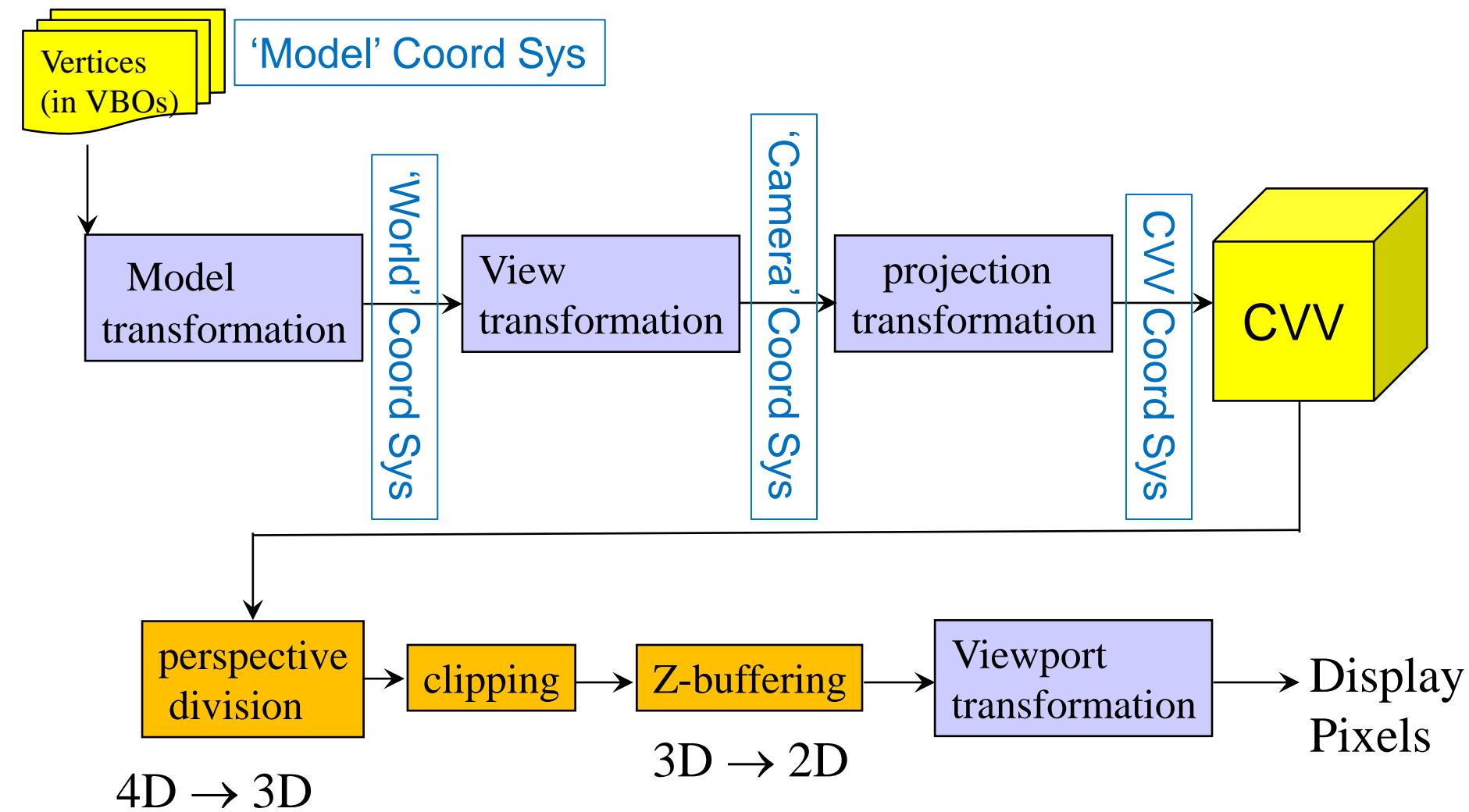
View matrix ‘positions the camera’. What’s the **simplest** way?

- Open starter code:

2021.11.01.Cameras3.1 →
Ch07_class → **‘BasicShapesCam’**

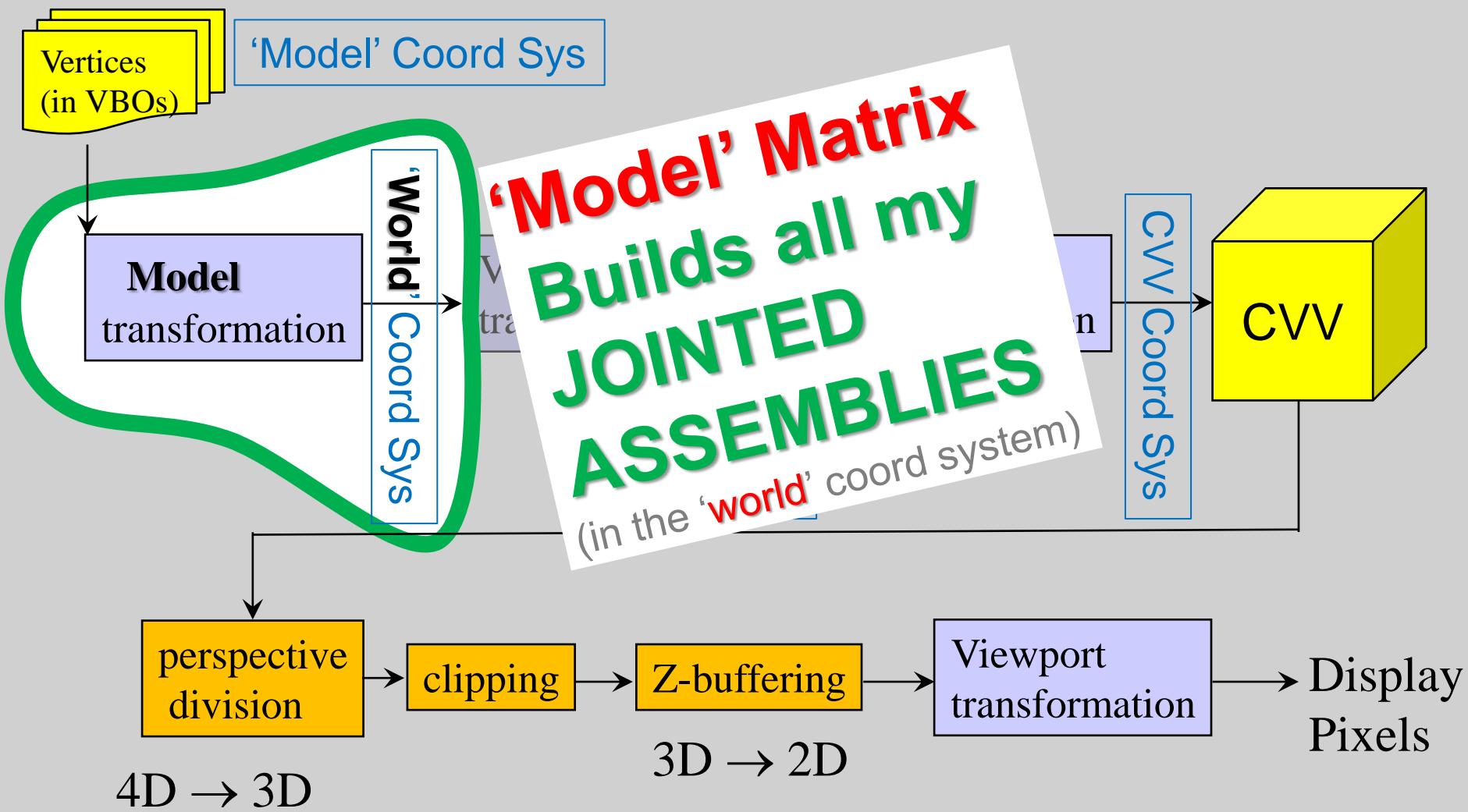
- In draw() function, modify transformations (~line 627...)
? How can make the grid into a ‘ground plane’?
- Typical ‘world’ coordinates: X,Y ground plane, ‘Up’ is +Z
- How can we transform these ‘world’ coordinates to CVV?
- Why does this all seem ‘backwards’ somehow?

Traditional Vertex Position Pipeline

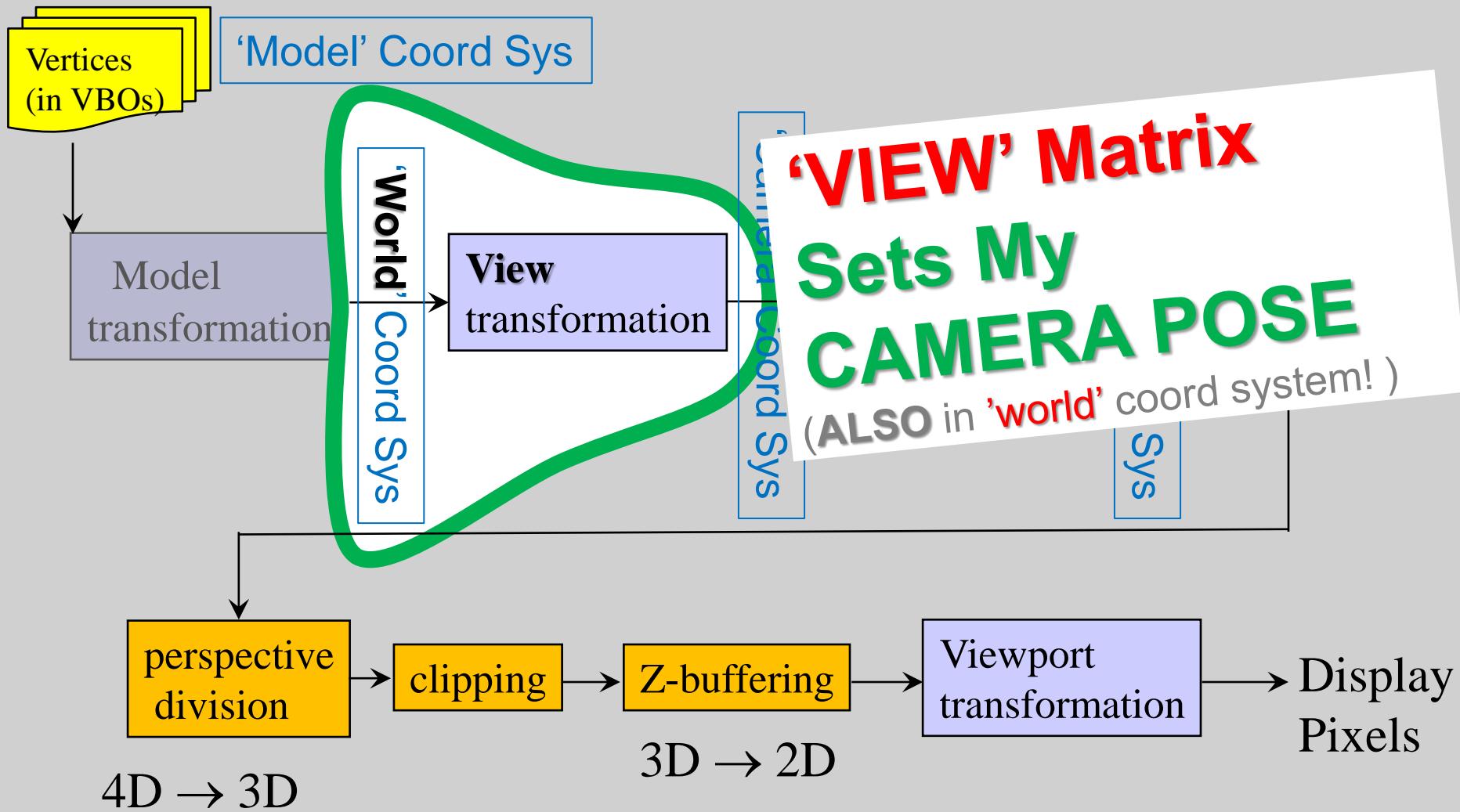


4D → 3D

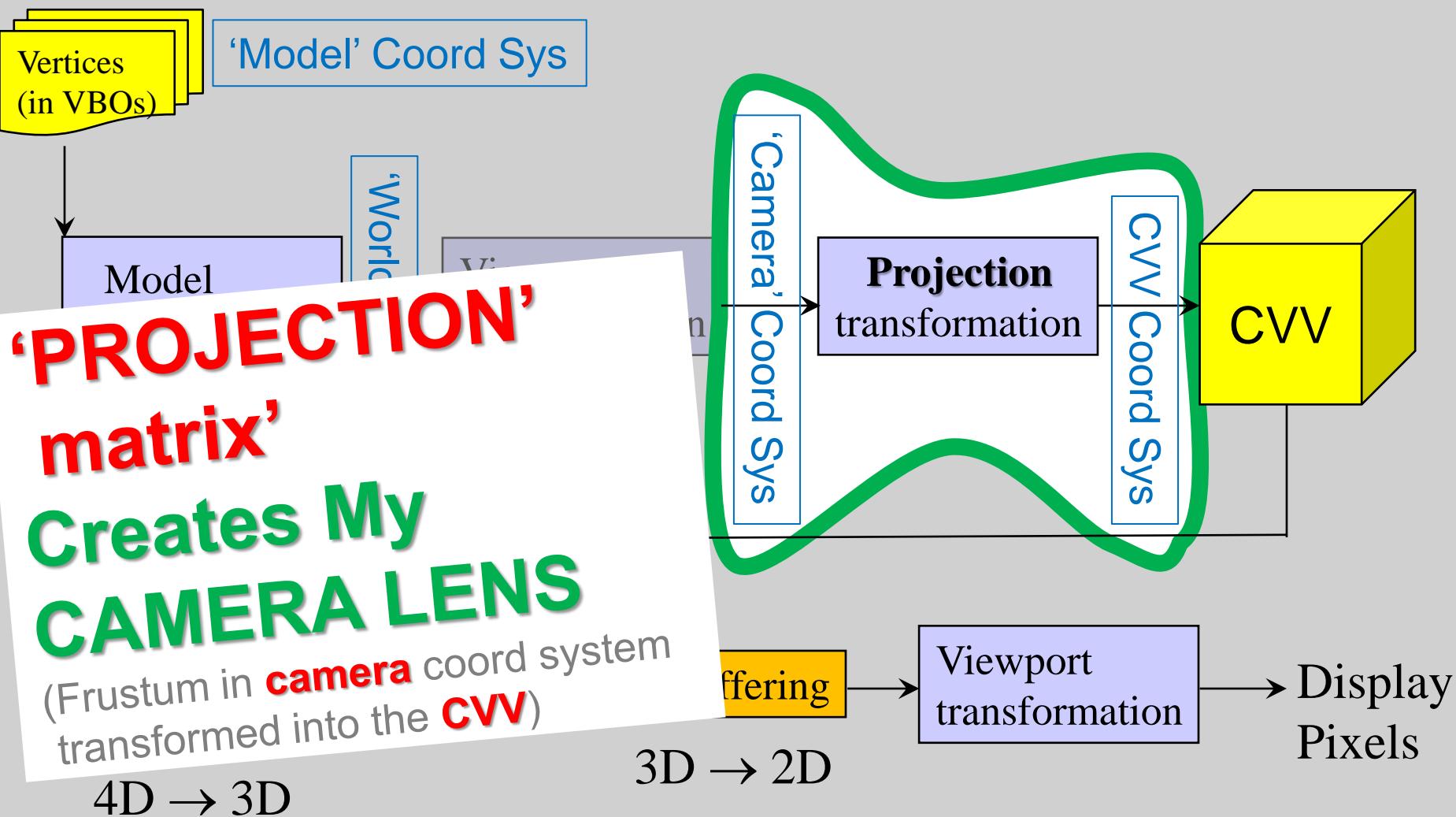
Traditional Vertex Position Pipeline



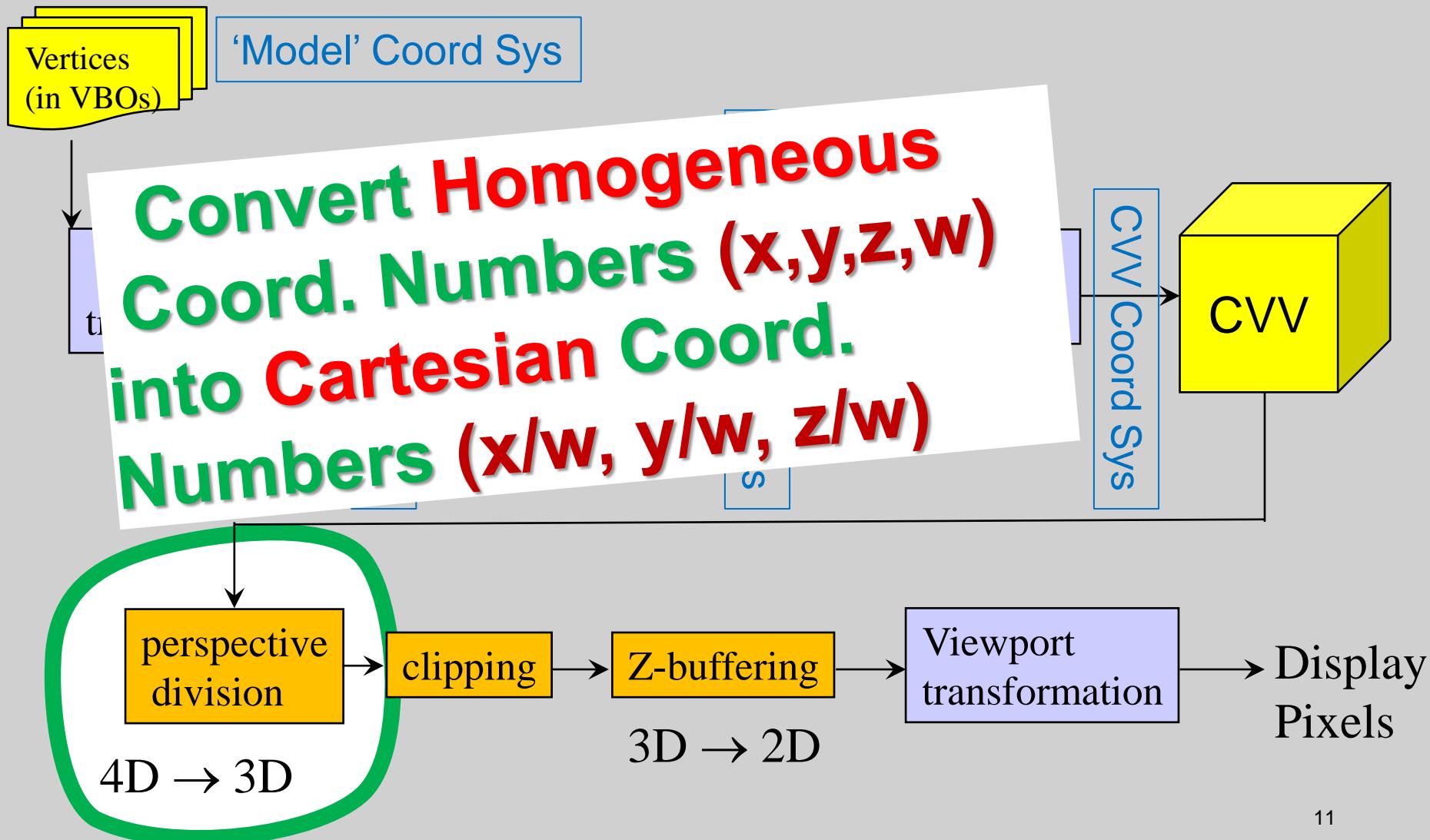
Traditional Vertex Position Pipeline



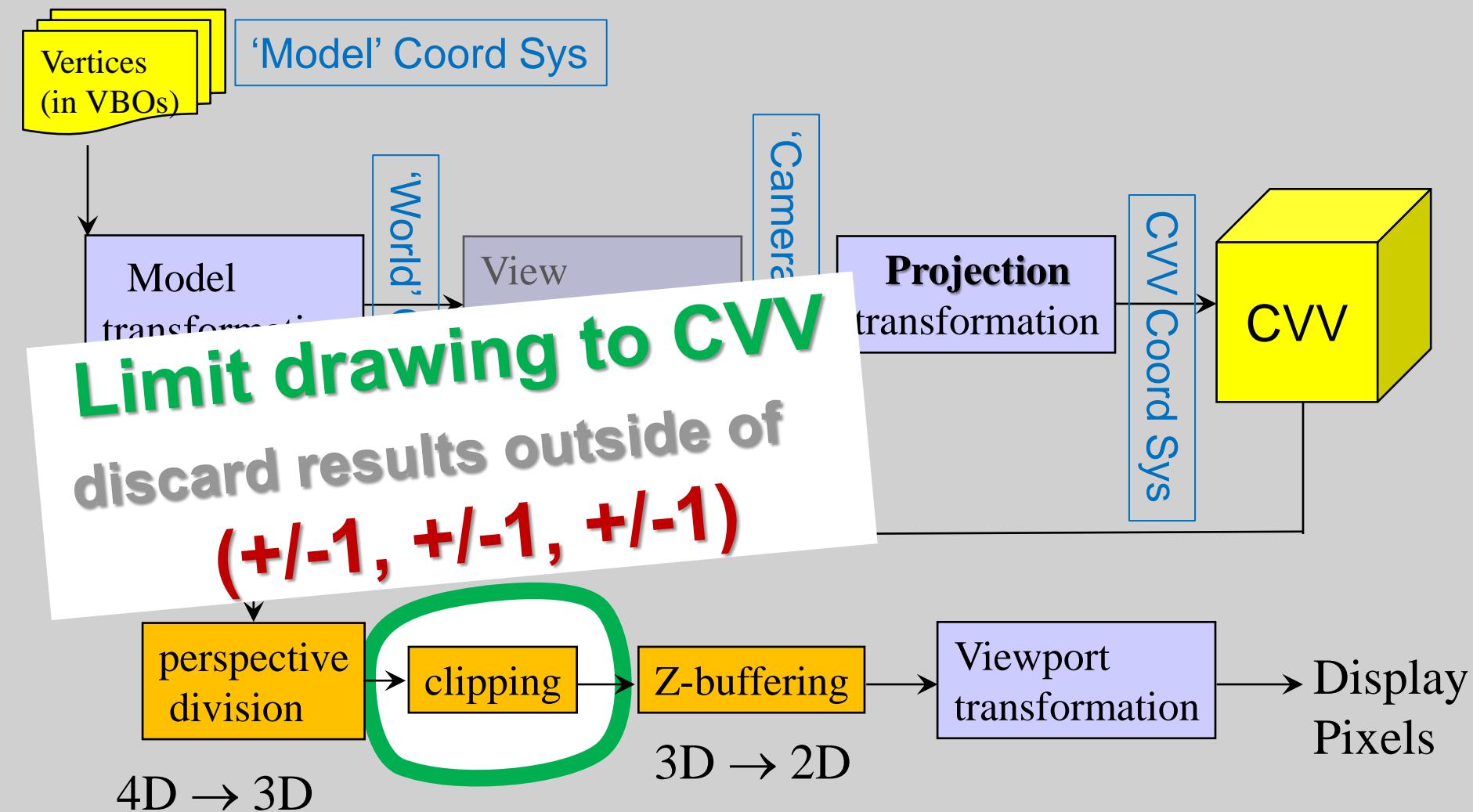
Traditional Vertex Position Pipeline



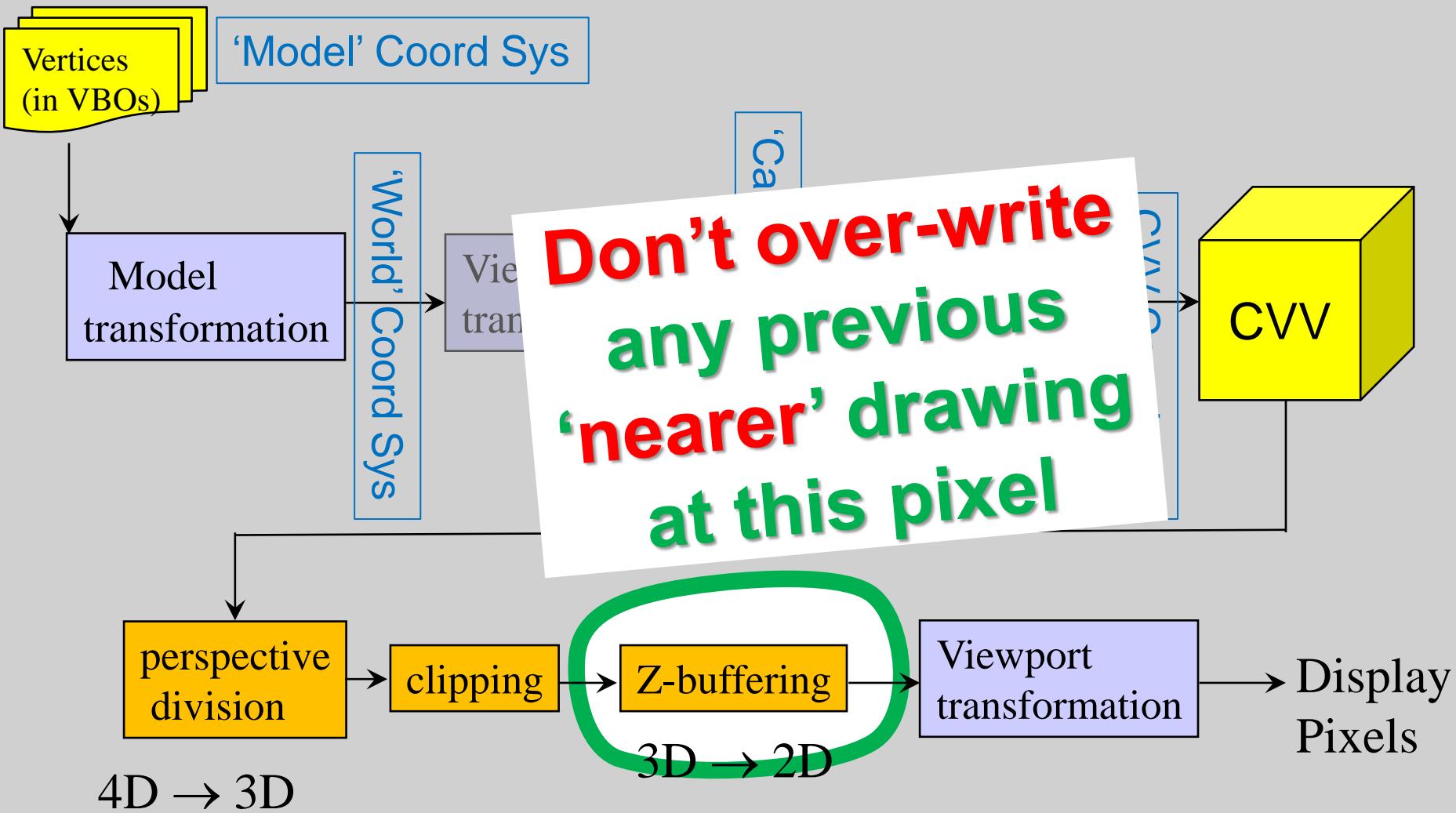
Traditional Vertex Position Pipeline



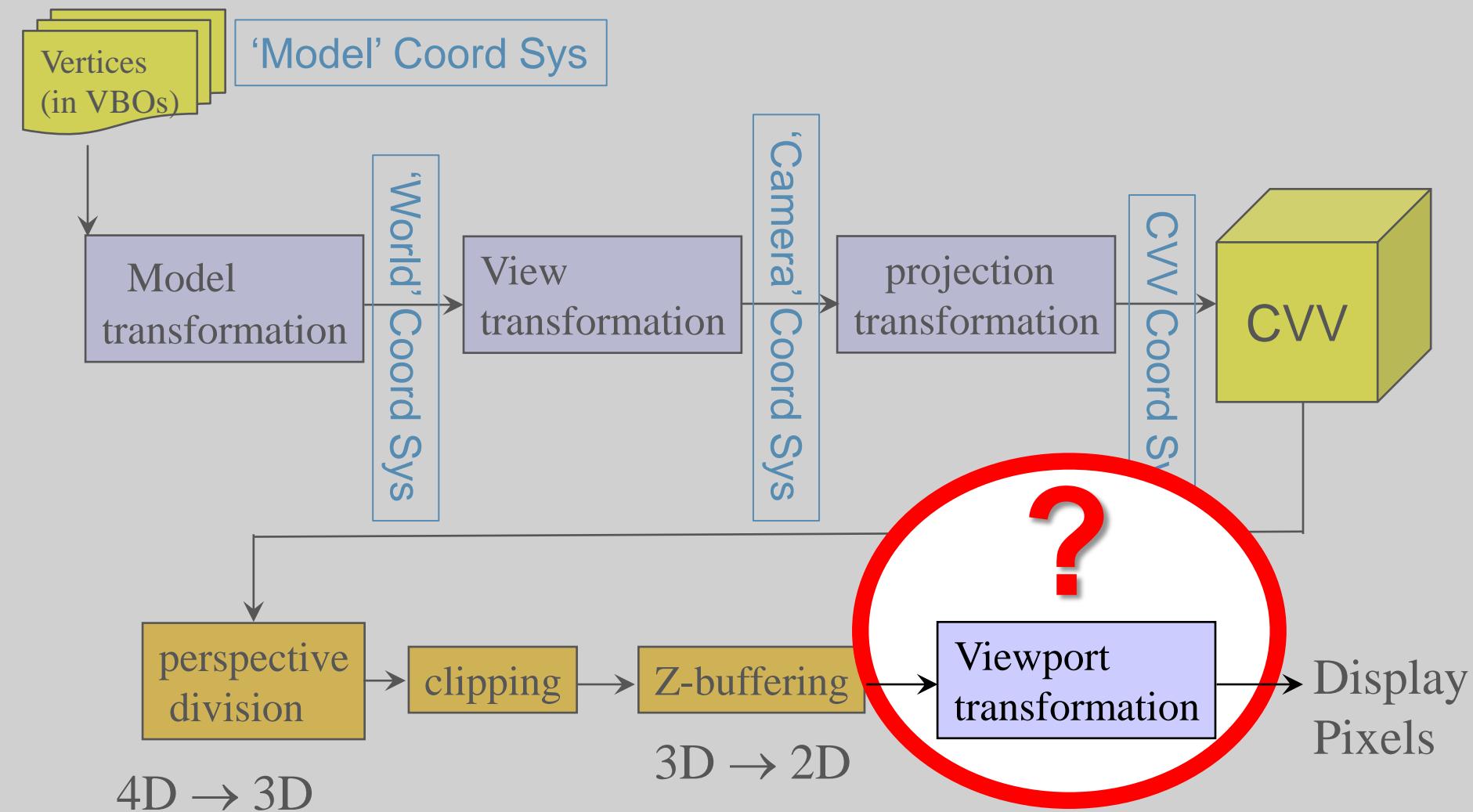
Traditional Vertex Position Pipeline



Traditional Vertex Position Pipeline

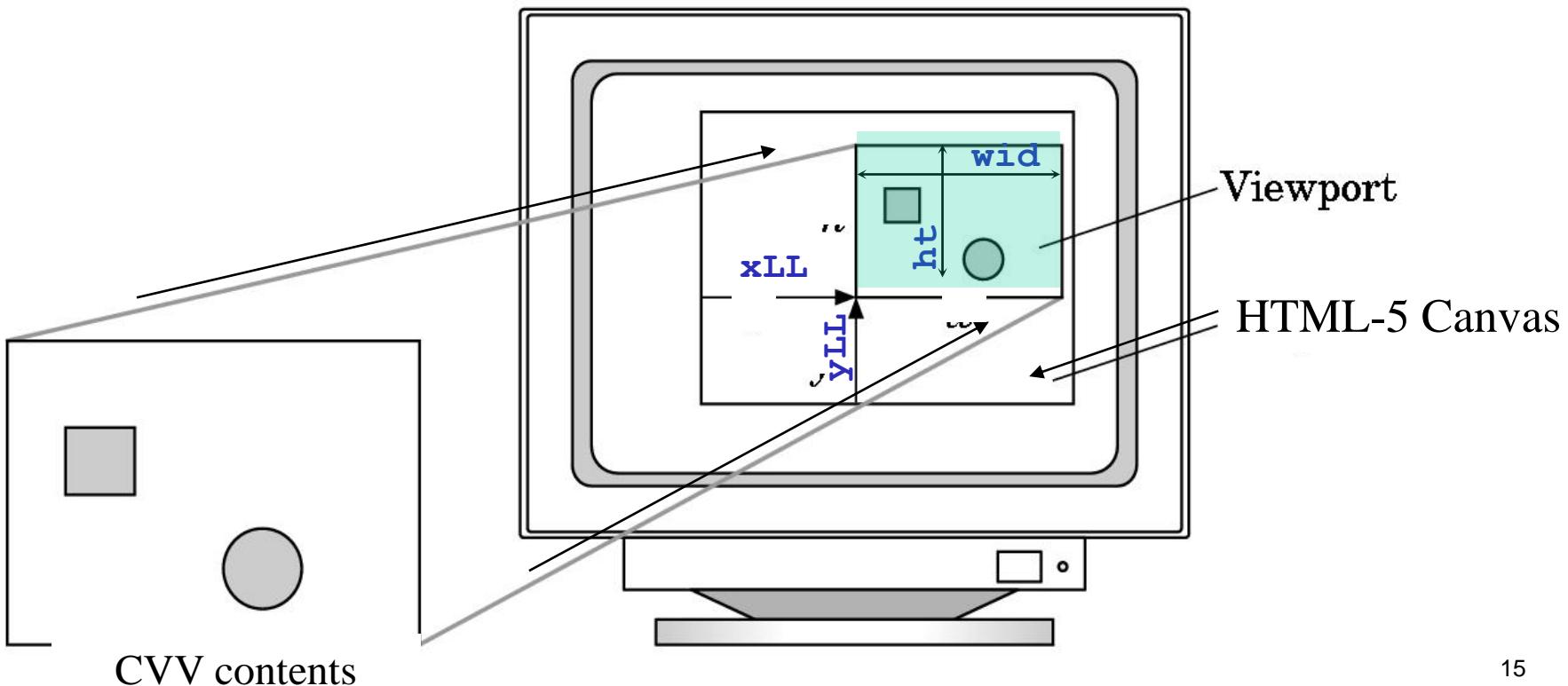


Traditional Vertex Position Pipeline



Viewport: How CVV maps to Display

- Default: fill entire HTML-5 Canvas with WebGL image:
- Gain control using this WebGL command:
gl.viewport(xLL, yLL, wid, ht)
- Values in pixels, origin at lower left





Notes

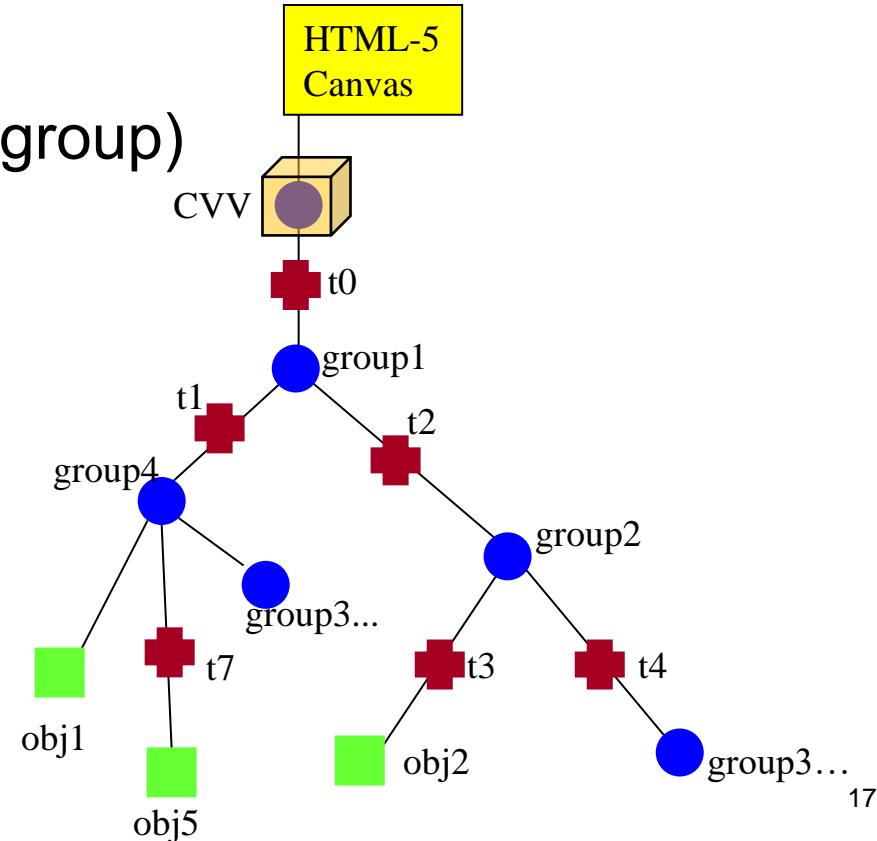
- **KEEP** our four-dimensional homogeneous coordinates throughout all transforms by **MODEL**, **VIEW**, and **PROJECTION** matrix:
 - All are nonsingular
 - All default to identity matrices (CVV sets limits)
- **Delay final perspective-divides until the end**
 - Important for hidden-surface removal:
we NEED depth information to draw pixels!
 - Efficient : FP division == expensive to compute

SceneGraphs in Project A

RECALL: in scene graphs,
Vertices & values move upwards,
transform calls move downwards

Project A:

- CVV == “World” drawing axes (group)



SceneGraph with the Historic Vertex Position Pipeline

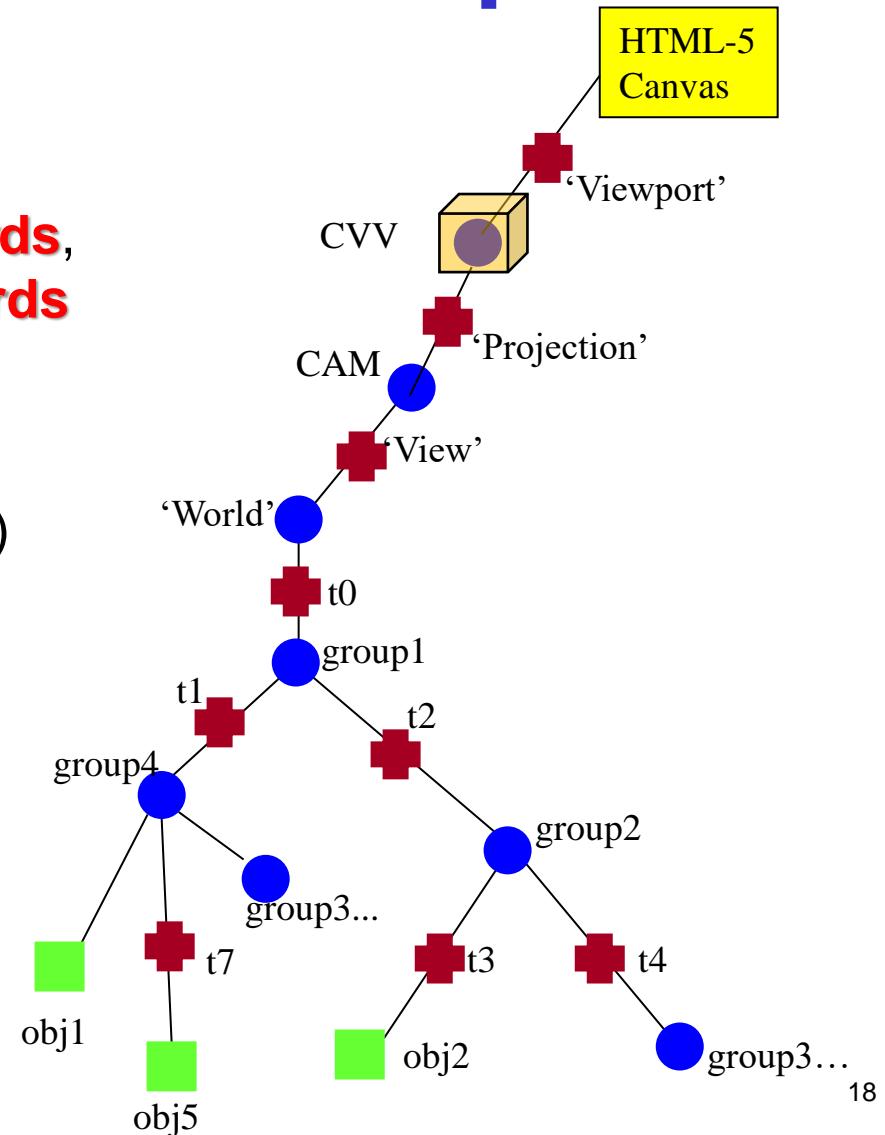
RECALL: in scene graphs,
Vertices & values move **upwards**,
transform calls move **downwards**

Project A:

- CVV == “World” drawing axes (group)

Project B:

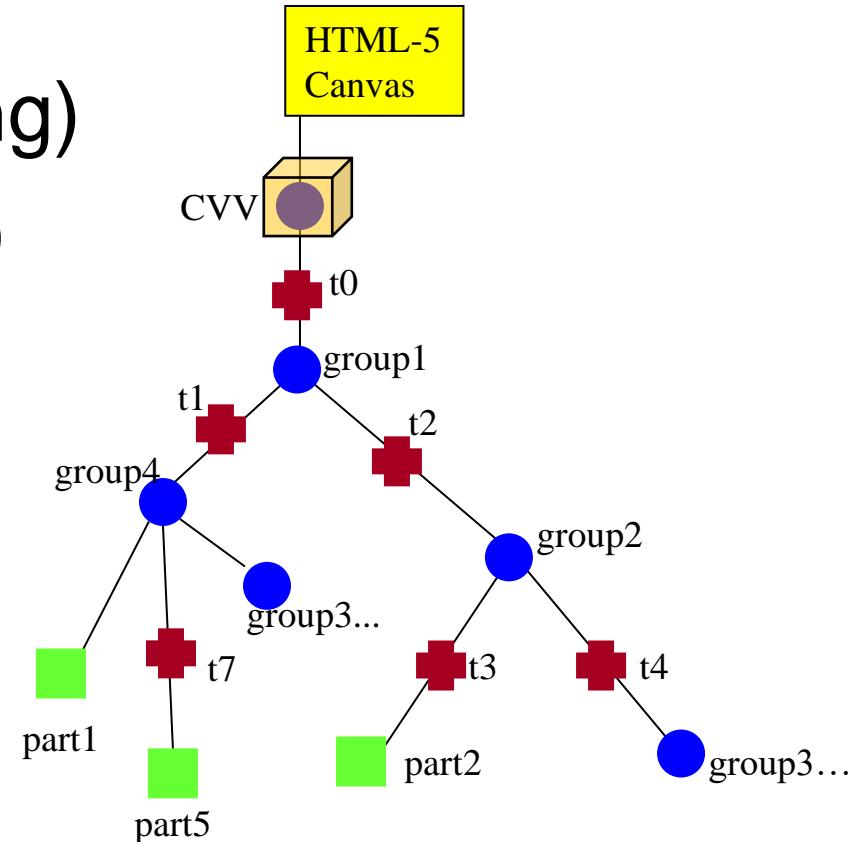
- Viewport output → HTML Canvas
- CVV output → Viewport input
- Lens Axes → CVV input
- Camera Axes → Lens Input
- ‘World’ Coords → Camera Axes



WebGL: Vertex Position Pipeline

Where do these matrices fit into the SceneGraph?

- MODEL (jointed objects)
- VIEW (camera-positioning)
- PROJECTION (lens-like)
- VIEWPORT (screen)



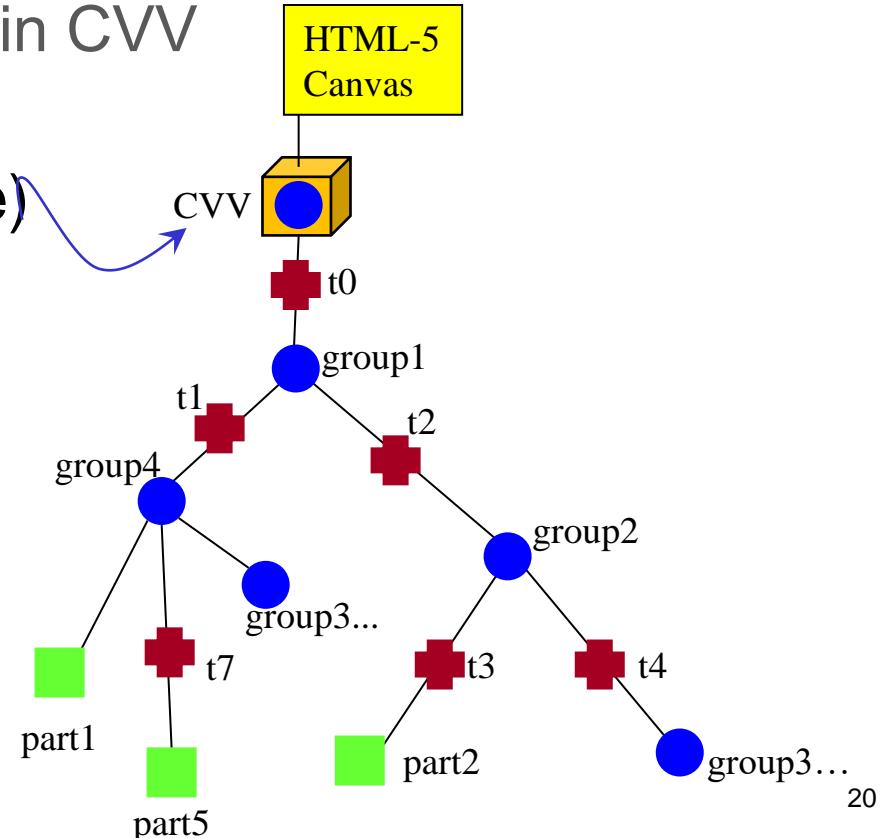
WebGL: Vertex Position Pipeline

RECALL: in scene graphs,
Vertices & their position coordinate number values
move upwards,
transform calls move downwards.

Project A: Draw ‘world’ directly in CVV

- CVV == “World” drawing axes
(at this **group** node)

```
Model.setTransform-t0(); // from CVV;  
pushMatrix(Model); // enter group 1  
  
Model.doTransform-t1()  
pushMatrix(Model); // enter group 4  
    drawPart1();  
    Model.doTransform-t7();  
    drawPart5();  
  
Model = popMatrix(); // return to group 4  
pushMatrix(Model); // enter group 3 ...
```



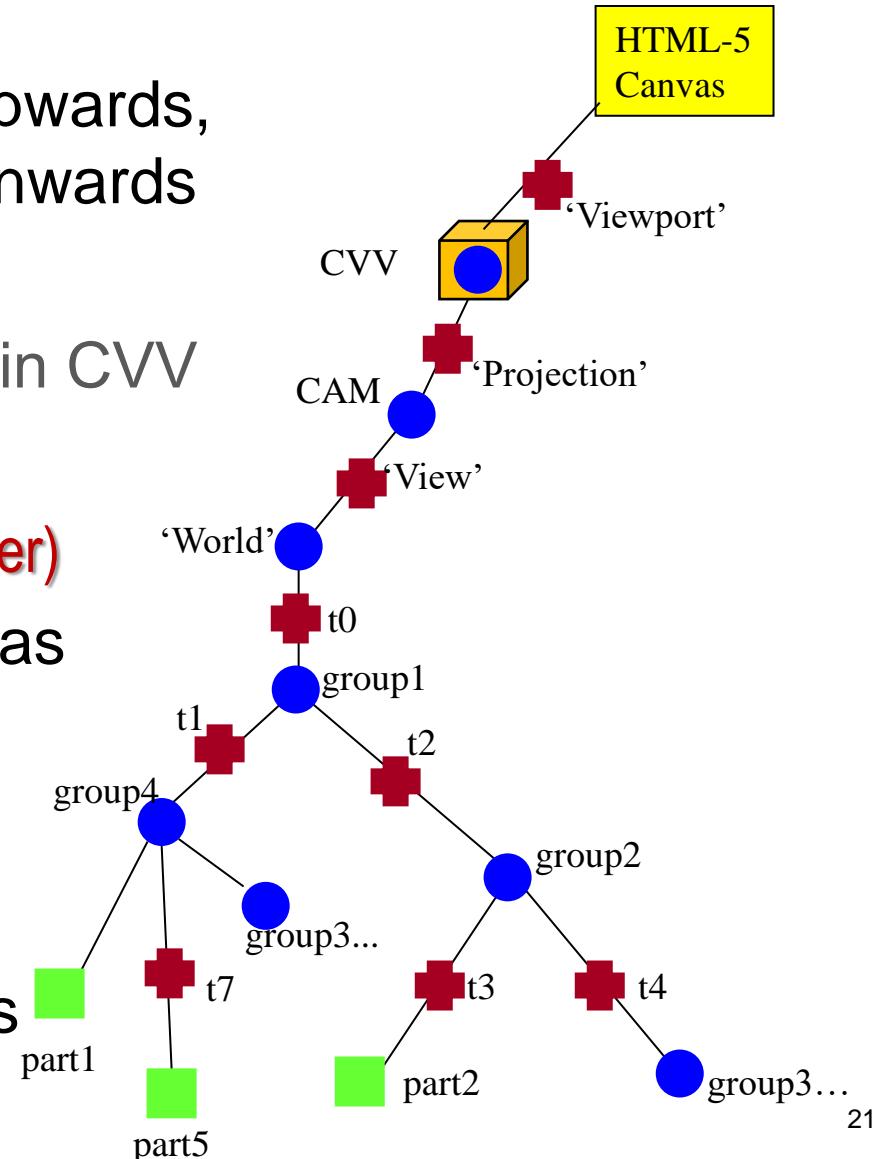
WebGL: Vertex Position Pipeline

RECALL: in scene graphs,
Vertices & values move upwards,
transform calls move downwards

Project A: Draw ‘world’ directly in CVV

Project B: (read these in reverse order)

- Viewport output → HTML Canvas
- CVV output → Viewport input
- Lens Axes → CVV input
- Camera Axes → Lens Input
- ‘World’ Coords → Camera Axes



Historic Vertex Position Pipeline

Vertex Stream

gl.ModelView

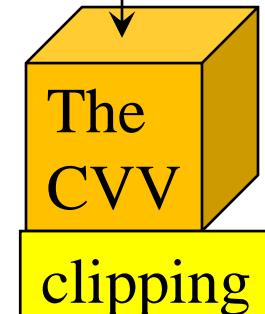
model + **view**
transformation

gl.Projection

projection
transformation

divide:
 $x/w, y/w, z/w$

4D → 3D

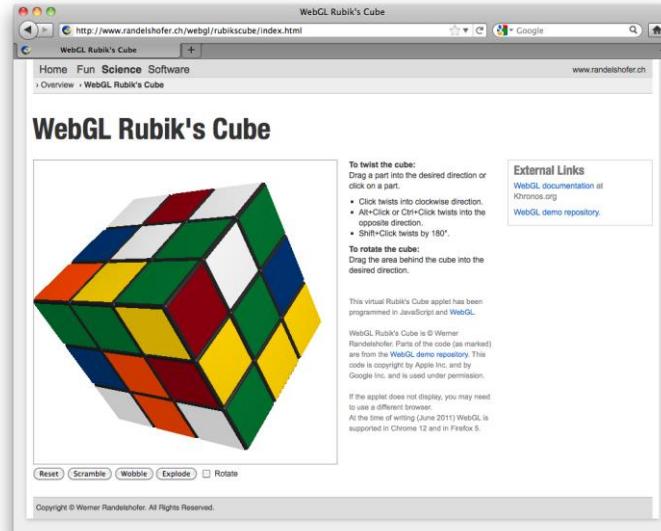


ViewPort
transformation

gl.viewport(lx, ly
width, height)
(in pixels)

“Historic”? Yep...
This **fixed** pipeline
was mandatory in
OpenGL until
version 3.0
replaced it with
programmable
shaders.
Still an excellent
default strategy!

HTML-5 Canvas



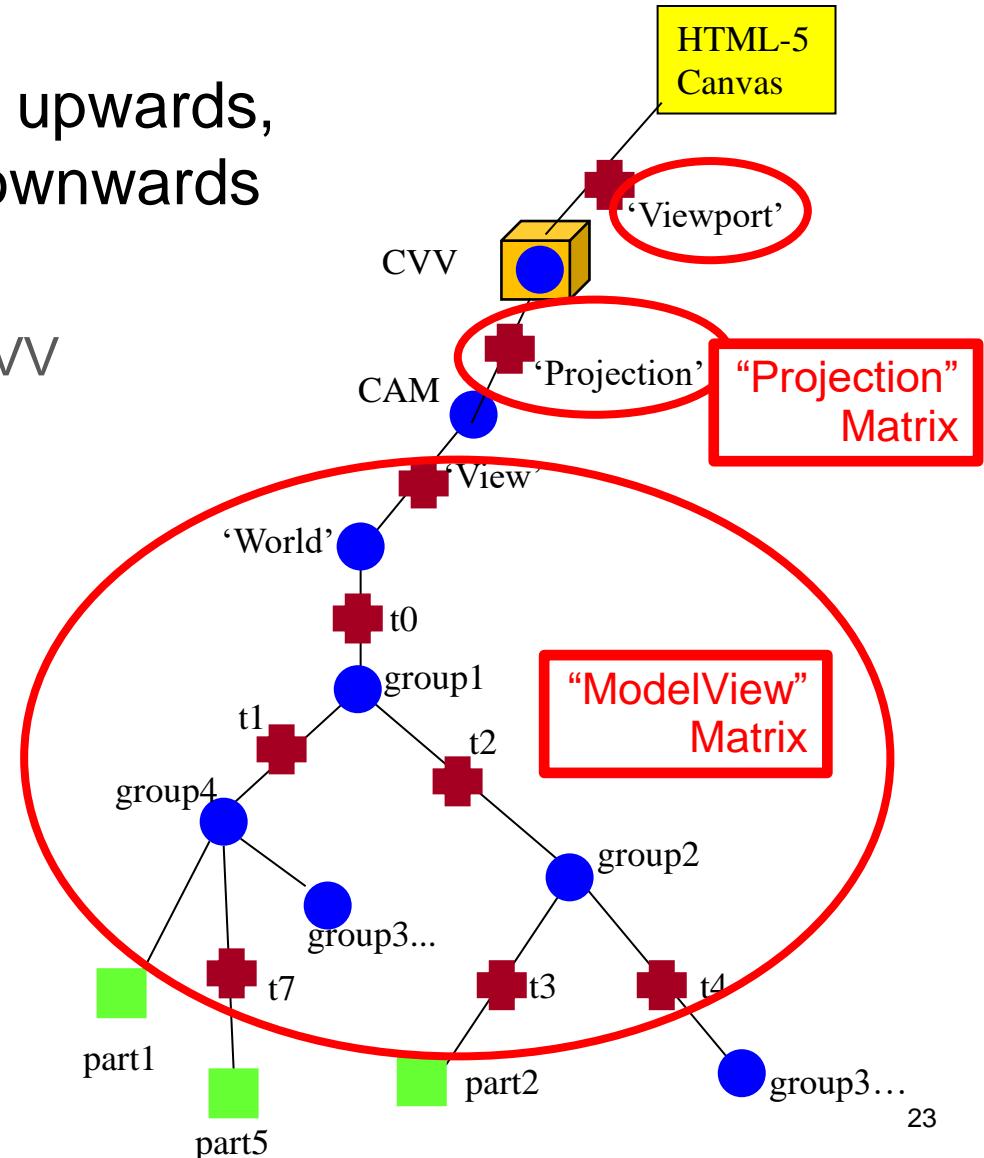
WebGL: Vertex Position Pipeline

RECALL: in scene graphs,
Vertices & values move upwards,
transform calls move downwards

Project A: Draw ‘world’ directly in CVV

Project B:

- Viewport output → HTML Canvas
- CVV output → Viewport input
- Lens Axes → CVV input
- Camera Axes → Lens Input
- ‘World’ Coords → Camera Axes



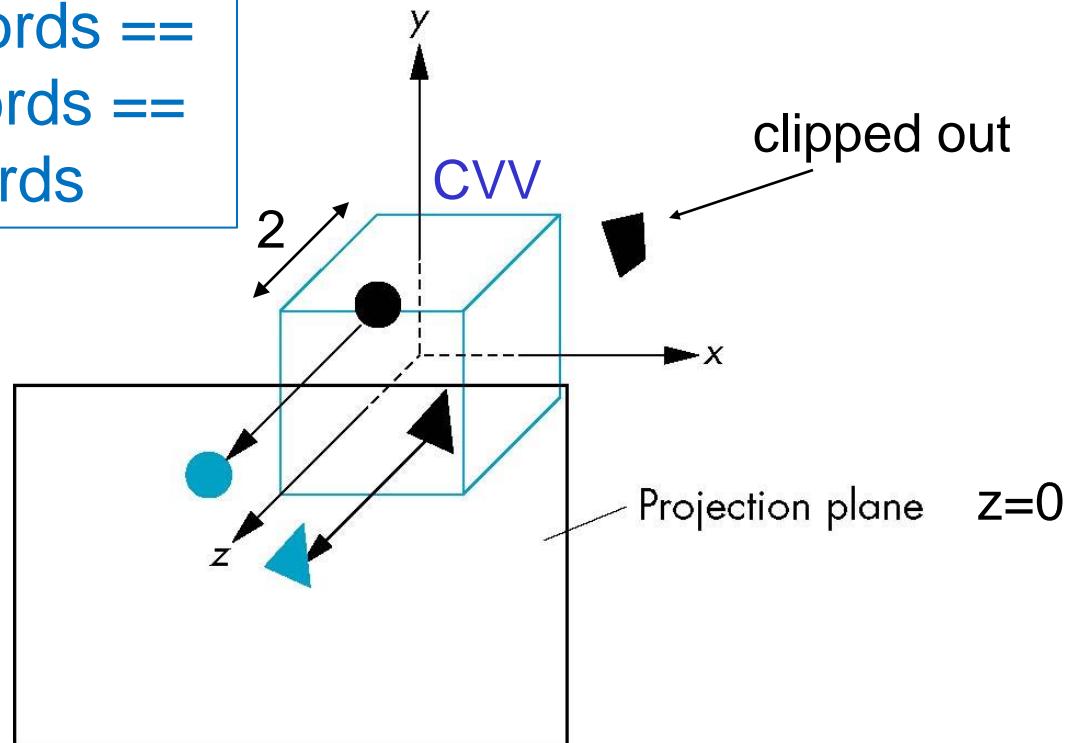


Default: no View, no Projection

The University of New Mexico

Default (nonexistent) ‘camera’

‘Camera’ x,y Coords ==
‘World’ x,y Coords ==
CVV x,y Coords





Default: no View, no Projection

Default (nonexistent) ‘camera’

‘Camera’ x,y Coords ==

‘World’ x,y Coords

CVV x,y Coords

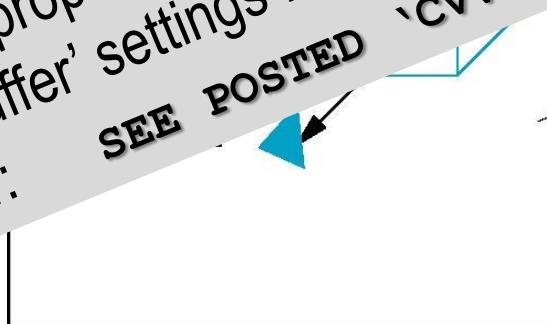
!!WARNING!!

‘Default Depth Reversal’ (see Proj A. starter code)

Historic quirk: OpenGL/WebGL relies on Projection matrix
for proper depth-buffer operation.

Default ‘depth-buffer’ settings render $-z$ as depth 0 ('behind' $+z$ at depth 1)
HOW TO FIX IT:

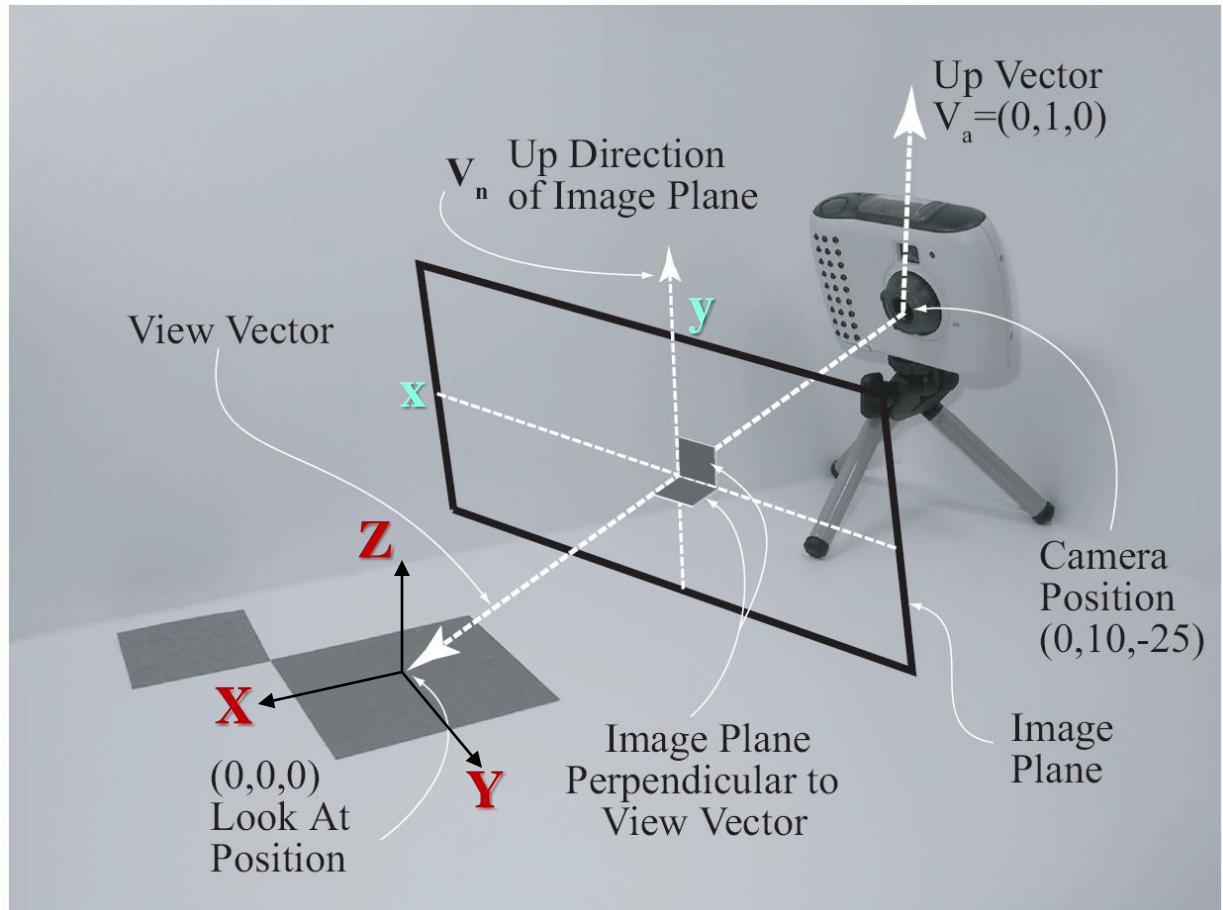
SEE POSTED ‘CVV Depth Reversal’ w/ CODE TO USE



View Matrix: Sets the Camera Pose

TERMINOLOGY:

- Camera position
- ‘Aim-point’ or ‘Look-At’ position
- ‘Up’ direction
- Related terms:
 - Image Plane
 - Viewing Direction
 - View Vector



Basic Shapes: Can we ‘Spin the World’?

- Screen coords: X,Y define 2D window into a virtual 3D world.
- World coordinates (typical): +Z==UP, X,Y==horizontal

View matrix ‘positions the camera’. What’s the **simplest** way?

- Open starter code:

2021.11.01.Cameras3.1 →
Ch07_class → **‘BasicShapesCam’**

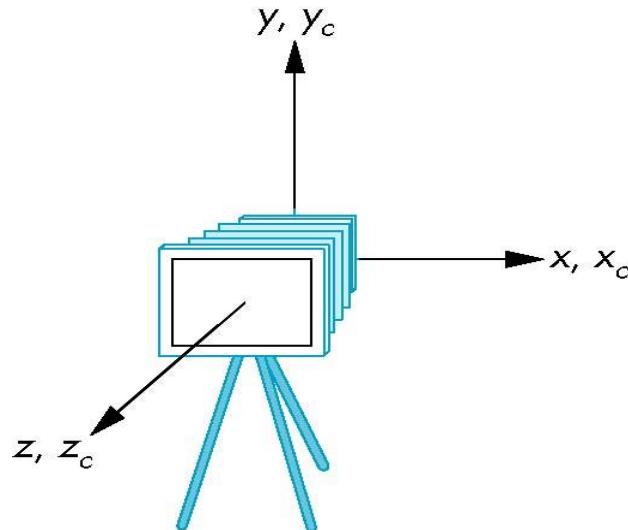
- In draw() function, modify transformations (~line 627...)
? How can make the grid into a ‘ground plane’?
- Typical ‘world’ coordinates: X,Y ground plane, ‘Up’ is +Z
- How can we transform these ‘world’ coordinates to CVV?
- Why does this all seem ‘backwards’ somehow?



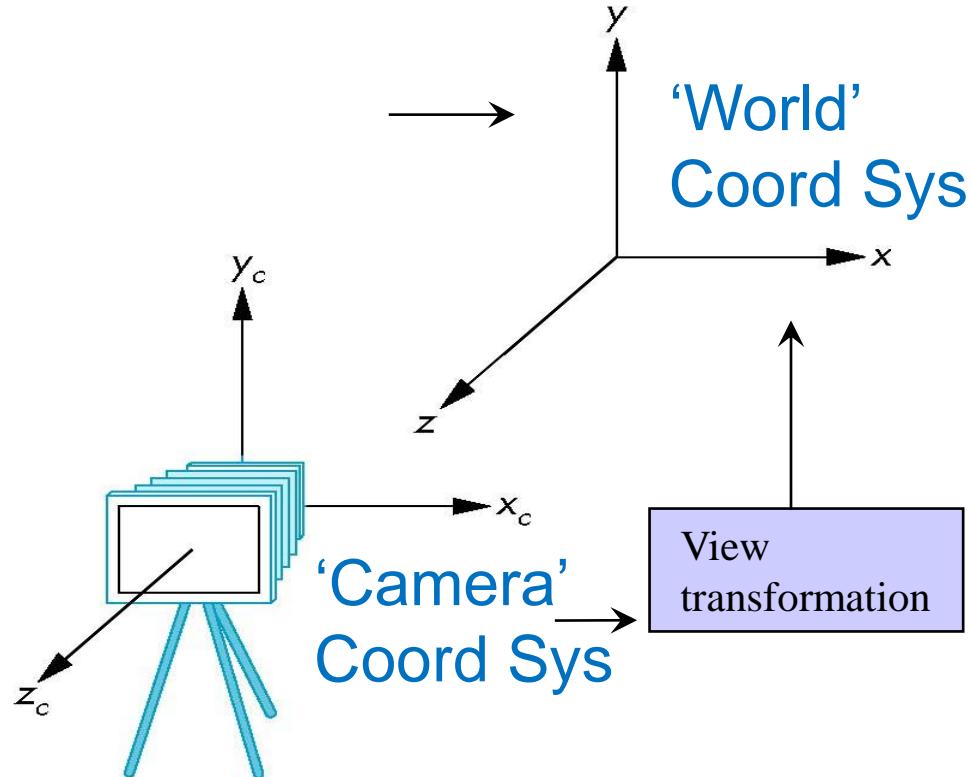
The University of New Mexico

Move the Camera? No; “Push the world away”

‘Camera’ x,y Coords ==
‘World’ x,y Coords ==
CVV x,y Coords



Default drawing axes →



after Translation by $(-d)$
(where $d > 0$)

(Caution! CVV clipping! We only ‘see’
the world within $(+/-1, +/-1, d+/-1)$ cube)



Move the Camera? No; “Push the world away”

- What happens next if I ‘rotate’?

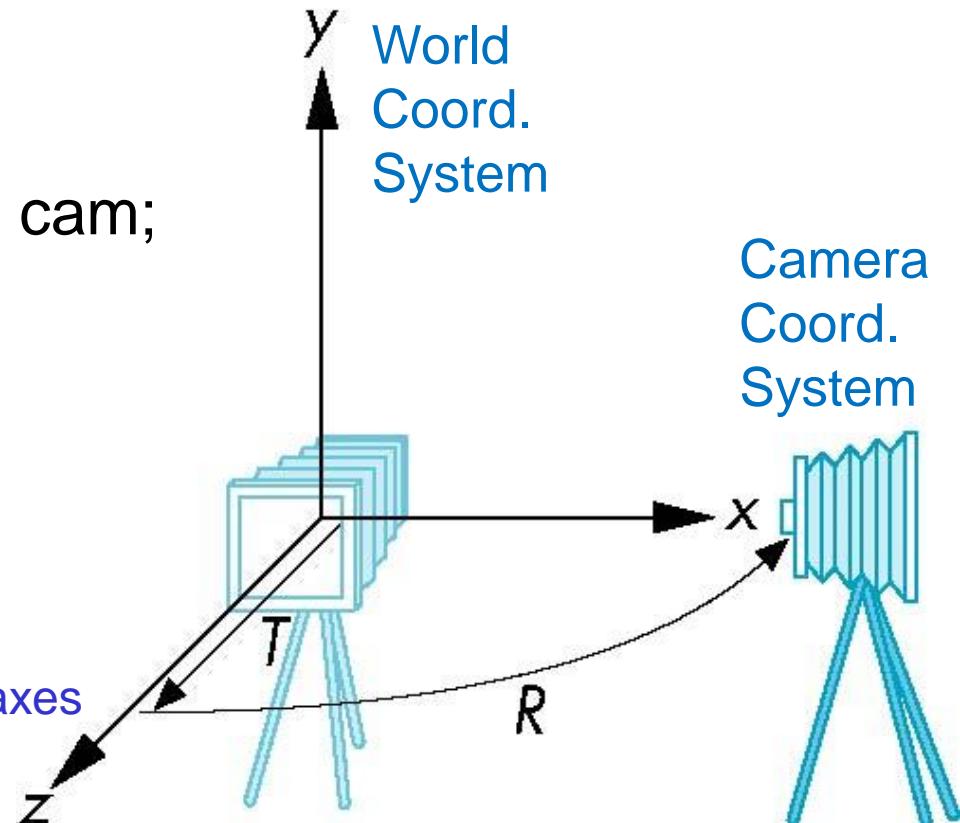
- Example: side view

- Push world away from cam;
 - Rotate (the world).
 - Model-view matrix:

$$C = TR$$

- AWKWARD!

- Always looks at world origin
 - **Positions World** coord sys. axes **from the Camera** coord sys.
 - **How can I position the Camera coord sys. axes from the World coord sys.?**



Posing the Camera

- !!AWKWARD!! Not intuitive! Instead,
We *want* to specify camera positions
measured in WORLD coords...

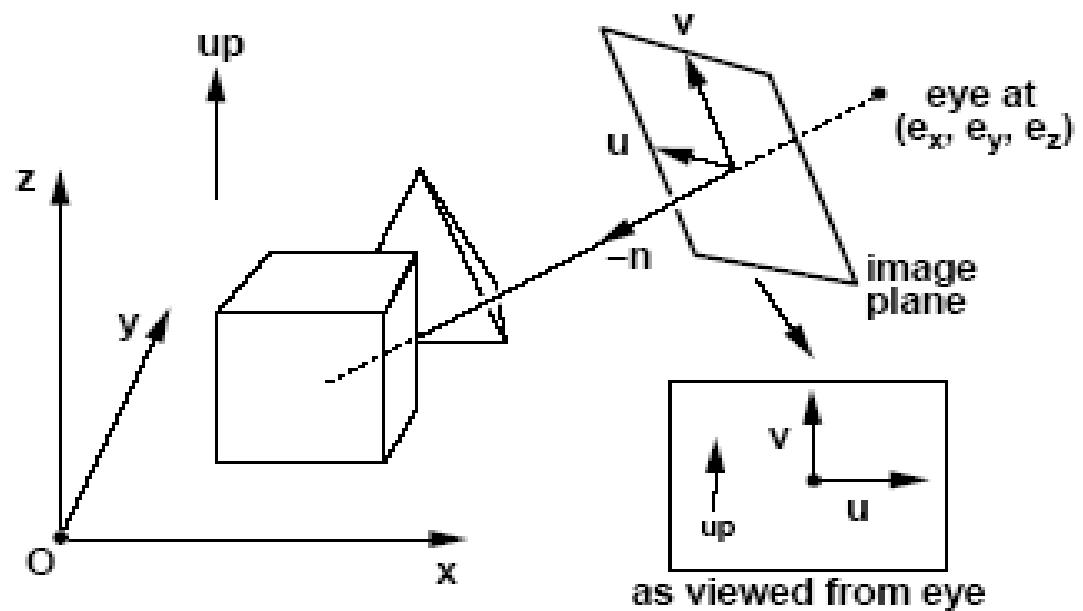
WANTED:

one 'view' matrix that

- changes vertex coords:
World \rightarrow Cam (e.g. Method1)

--- AND (EQUIVALENTLY) ---

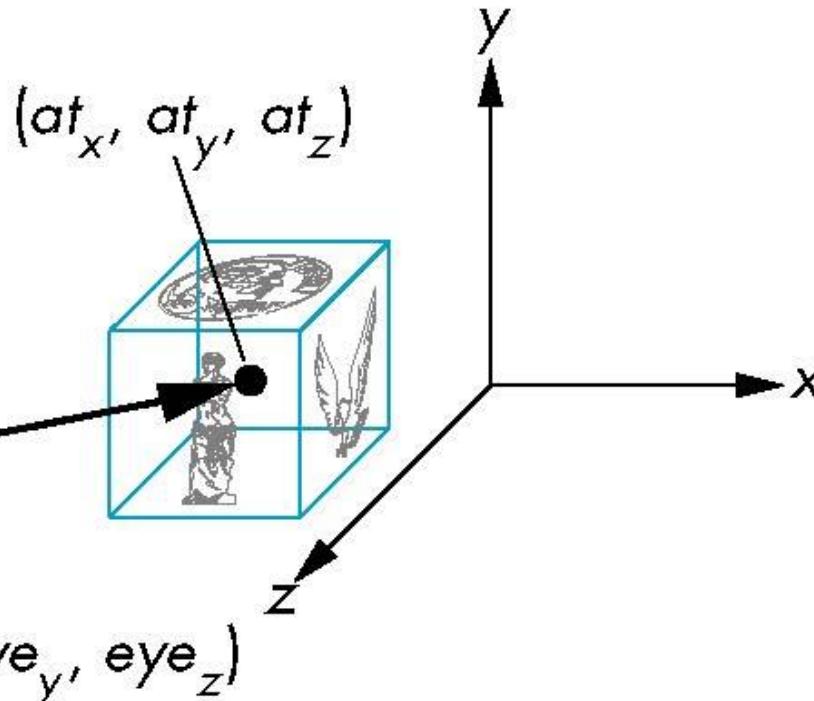
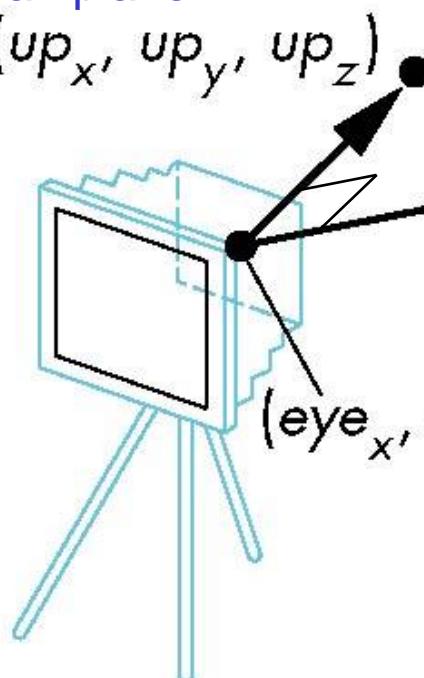
- changes drawing axes:
Cam \rightarrow World (e.g. Method 2)



LookAt() makes ‘view’ matrix

Up point: put anywhere in camera's +z-axis half-plane

(up_x, up_y, up_z)



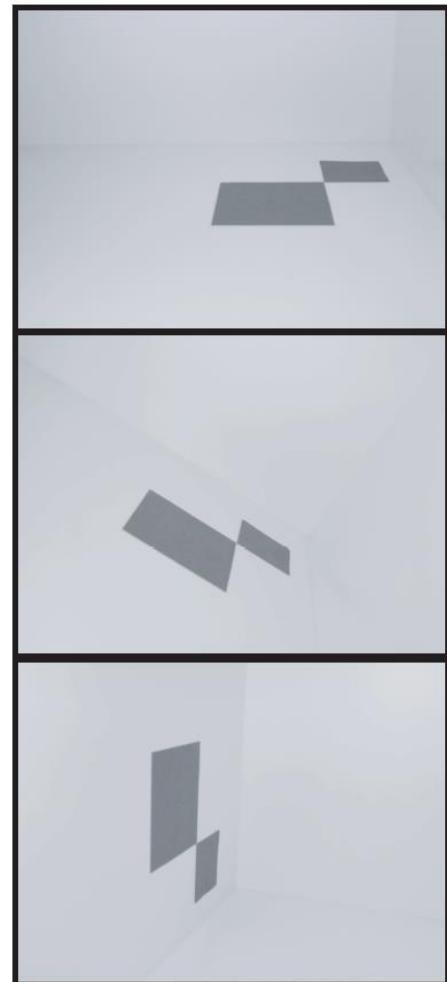
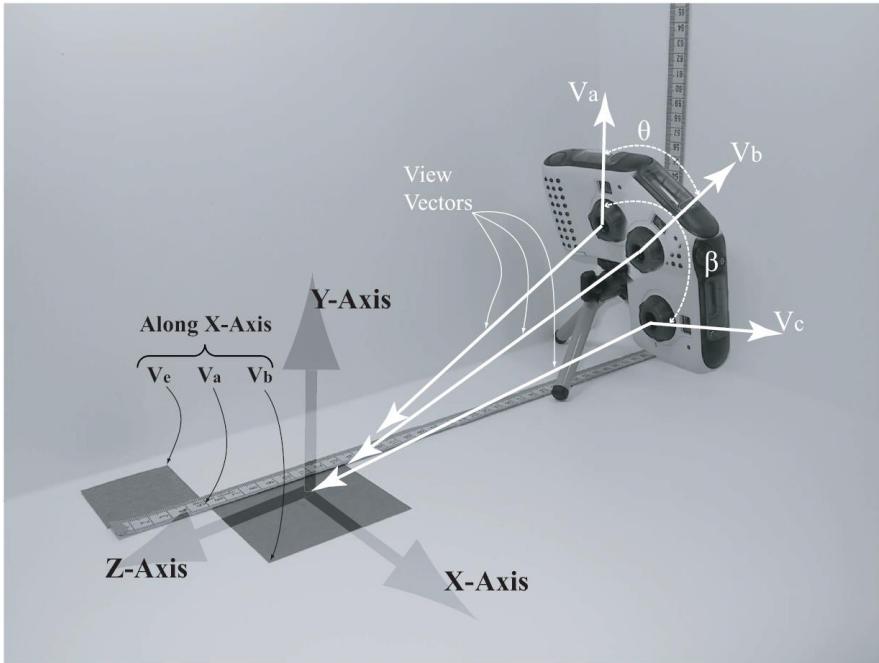
Matrix4 members :

lookAt(eye, at, up);

setLookAt(eye, at, up);

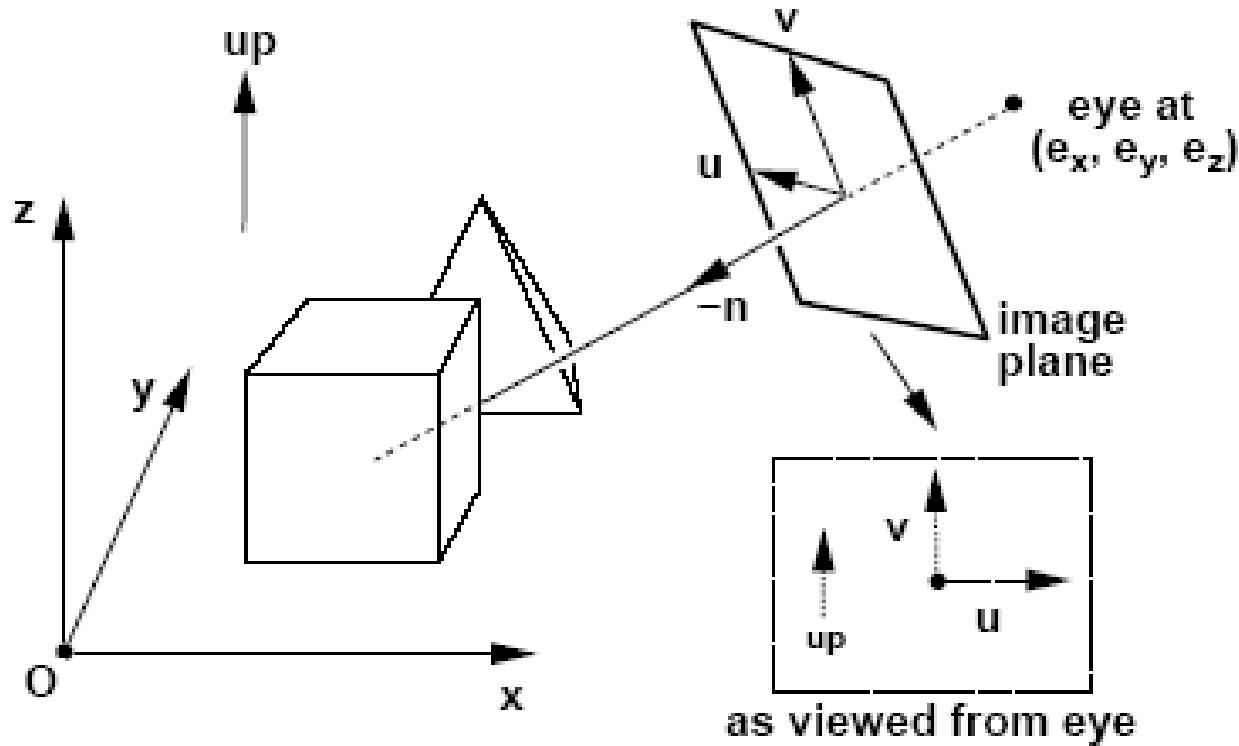
(see cuon-matrix-quat03.js)

The Up Direction (**Up** Vector): Chooses the 'camera twist'



- Also referred to as: **Twist Angle**
 - Cannot be parallel to viewing direction
 - Does not need to be normalized
 - Does not need to be perpendicular to viewing direction

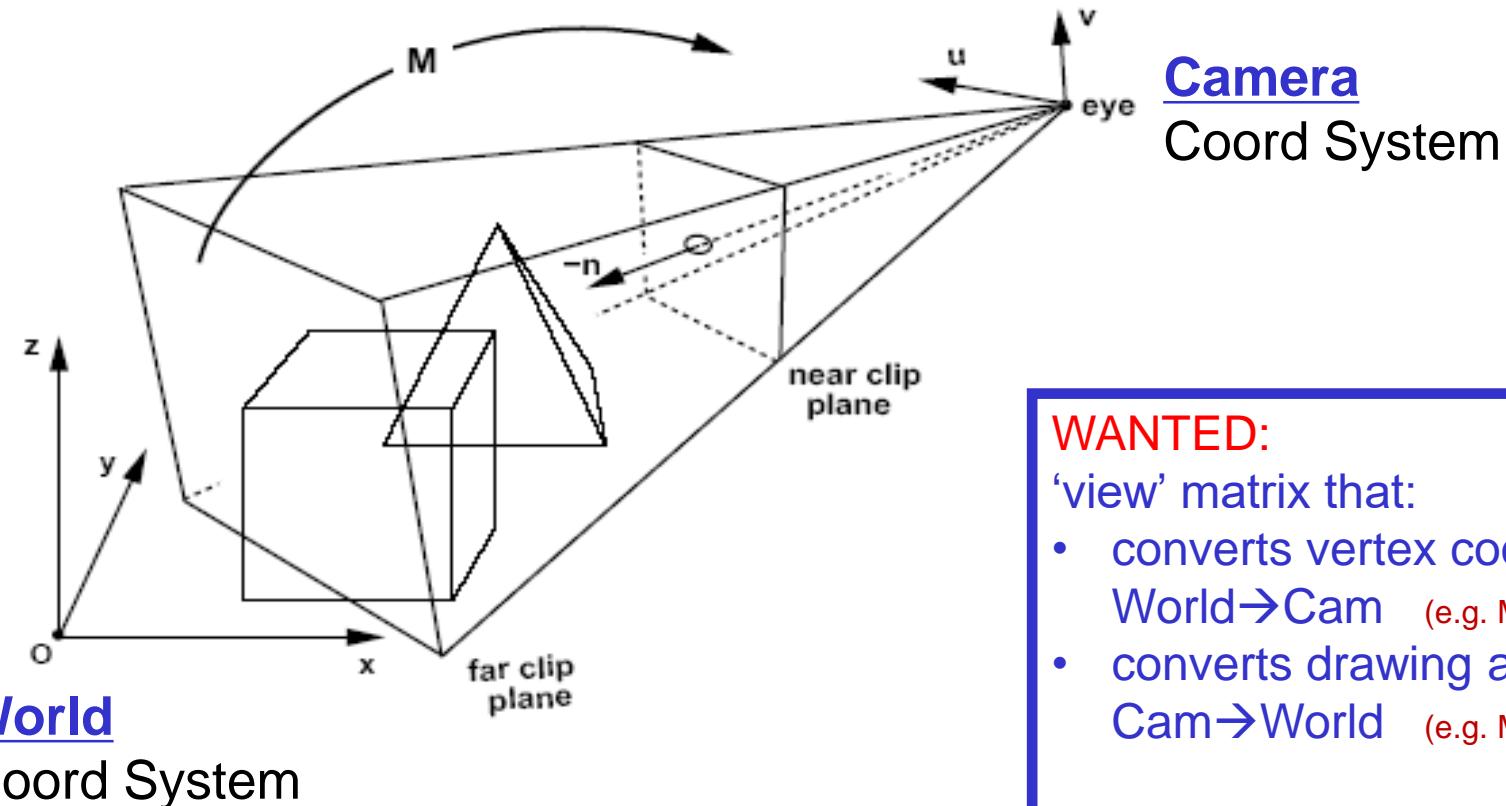
HOW? Define Eye Coords (u, v, n)



Eye Coordinate System: (a Camera Coord. System re-named)

- Eyepoint defines origin;
- u axis toward “right” of image plane
- v axis toward “top” of image plane
- n axis “pokes you in the eye”; Camera “Looks down the $-n$ axis” (!)

Find View Matrix?



WANTED:

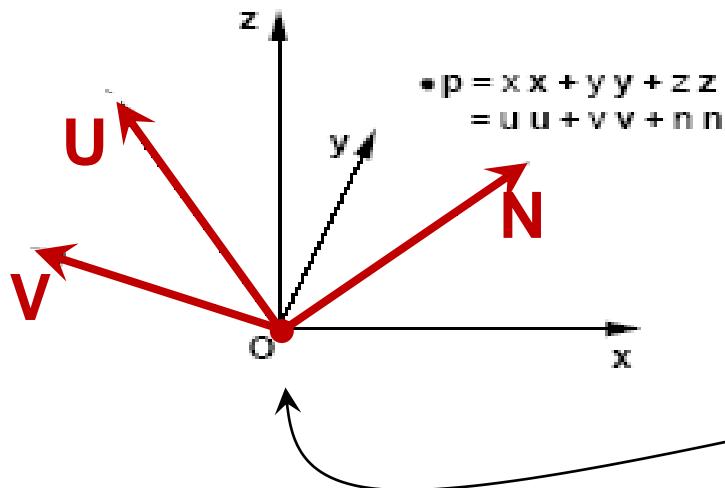
'view' matrix that:

- converts vertex coords:
World \rightarrow Cam (e.g. Method 1)
- converts drawing axes:
Cam \rightarrow World (e.g. Method 2)

Construct the transformation matrix M that:

- Creates drawing axes backwards: 'World' \rightarrow 'Camera', or equivalently,
- Converts vertex coordinate numbers: 'Camera' \rightarrow 'World' values.

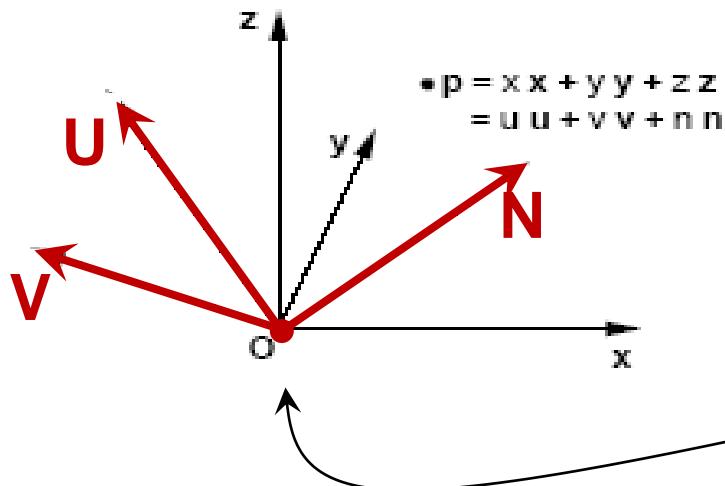
Matrix Machinery: Changing Orthogonal Bases



$$M = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Suppose you are **given** unit-length, orthogonal basis vectors **U**, **V**, **N** that describe camera axes defined in world-space.
- Imagine these **U**, **V**, **N** vectors shown at the ‘world’ space origin, like this.
- If we create matrix **M** with **rows** made of the **U**, **V**, **N** vectors as shown, what does M do to point **(P0x, P0y, P0z, 1)**?

Matrix Machinery: Changing Orthogonal Bases



$$M = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Suppose you are **given** unit-length, orthogonal basis vectors **U, V, N** that describe camera axes defined in world-space.
- Imagine these **U, V, N** vectors shown at the ‘world’ space origin, like this.
- If we create matrix **M** with **rows** made of the **U, V, N** vectors as shown, what does M do to point **(P0x, P0y, P0z, 1)**?

ANSWER: finds dot-product of vector **(P0 – origin)** with each of the **U,V,N** basis vectors

What happens when you apply M to those same u, v, n unit points?

'World' 'CAM'
point → point

$$M \begin{pmatrix} u_x \\ u_y \\ u_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

'World' 'CAM'
point → point

$$M \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

'World' 'CAM'
point → point

$$M \begin{pmatrix} n_x \\ n_y \\ n_z \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

WANTED:

'view' matrix that:

- converts vertex coords:
World→Cam (e.g. Method 1)
- converts drawing axes:
Cam→World (e.g. Method 2)

What happens when you apply **M** to those same **u, v, n** unit points?

'World' 'CAM'
point → point

$$M \begin{pmatrix} u_x \\ u_y \\ u_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

'World' 'CAM'
point → point

$$M \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

'World' 'CAM'
point → point

$$M \begin{pmatrix} n_x \\ n_y \\ n_z \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

WANTED:

'view' matrix that:

- converts vertex coords:
World→Cam (e.g. Method1)
- converts drawing axes:
Cam→World (e.g. Method 2)

ANSWER: "Matrix Duality" again.

two equally valid interpretations:

Method 1:

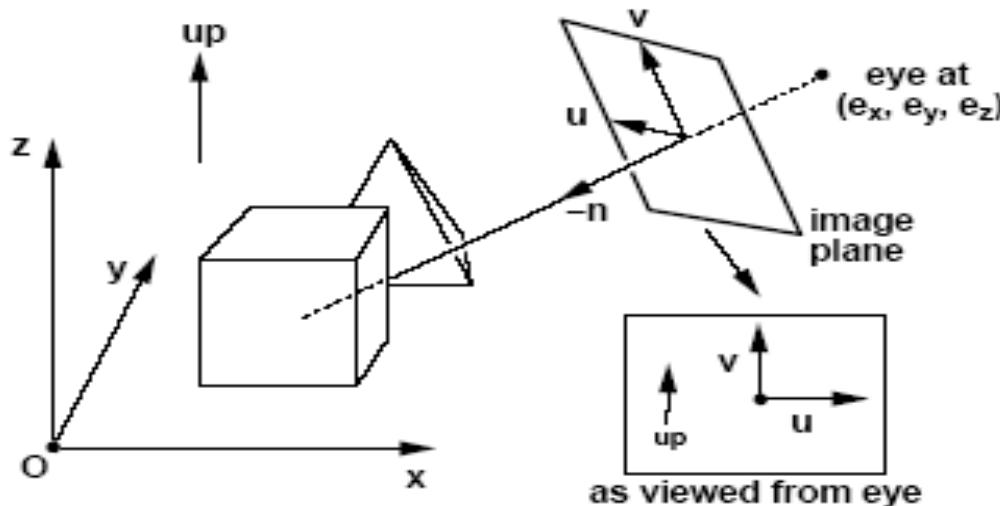
Matrix **M** converts old 'World' point-coordinate values into new 'CAM' point-coordinate values. You can draw both using the same on-screen drawing axes.

Method 2:

Matrix **M** leaves all coordinate values unchanged,
BUT converts CAM **u,v,n drawing axes** into
World x,y,z drawing axes.

Draws the unchanged 'world' coord. values onscreen,
but does it using the new 'world' xyz drawing axes.

Simple ‘View’ Matrix Construction



Given eyepoint \mathbf{e} and CAM coordinate axes vectors $\hat{\mathbf{u}}, \hat{\mathbf{v}}, \hat{\mathbf{n}}$

- 1) Create rotation matrix **M**: CAM drawing axes \rightarrow WORLD drawing axes.
- 2) Then translate them: to move ‘push the world-drawing axes out of your eye’, away from the camera origin: translate the world by vector $(-\mathbf{EYE})$.

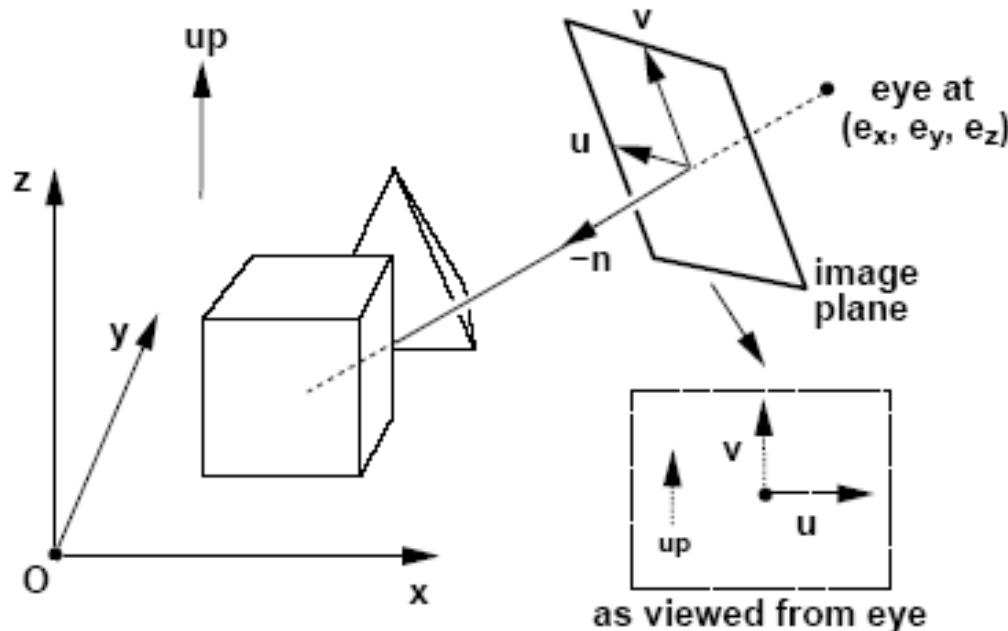
$$\mathbf{M} = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{T}(-\mathbf{EYE}) = \begin{bmatrix} 1 & 0 & 0 & -EYE_x \\ 0 & 1 & 0 & -EYE_y \\ 0 & 0 & 1 & -EYE_z \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

VIEW = MT

Camera Pose: “`lookAt()`” function

Trick: *construct \mathbf{u} and \mathbf{v} from available information!*

“Twist” constraint: Align \mathbf{v} with world **up** vector



Given: eyepoint e , view direction \mathbf{n} , and world **up** vector:

1. Compute $\mathbf{u} = -\mathbf{n} \times \mathbf{up}$
2. Compute $\mathbf{v} = \mathbf{u} \times -\mathbf{n}$
3. Construct M as above from \mathbf{u} , \mathbf{v} , \mathbf{n} , and e



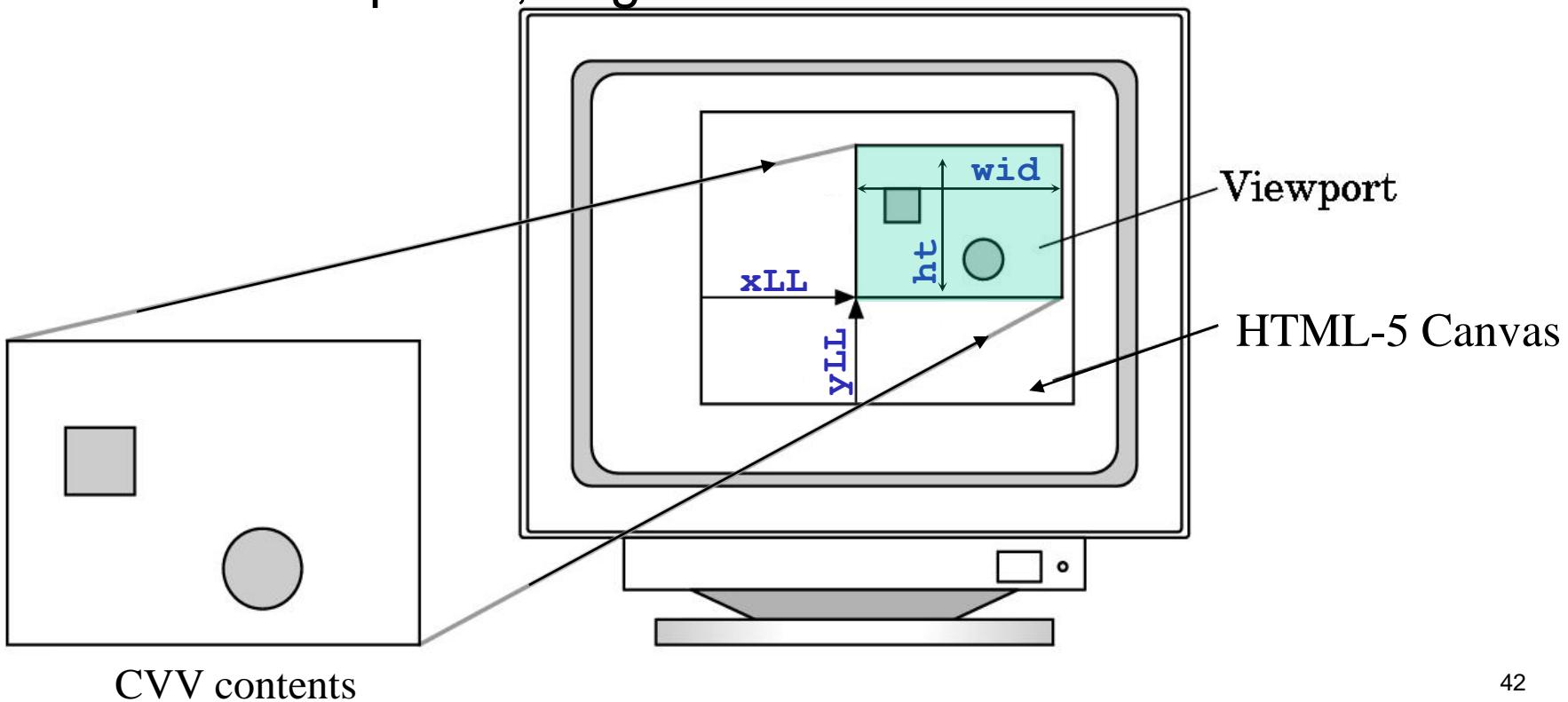
The University of New Mexico

END

END

Viewport: How CVV maps to Display

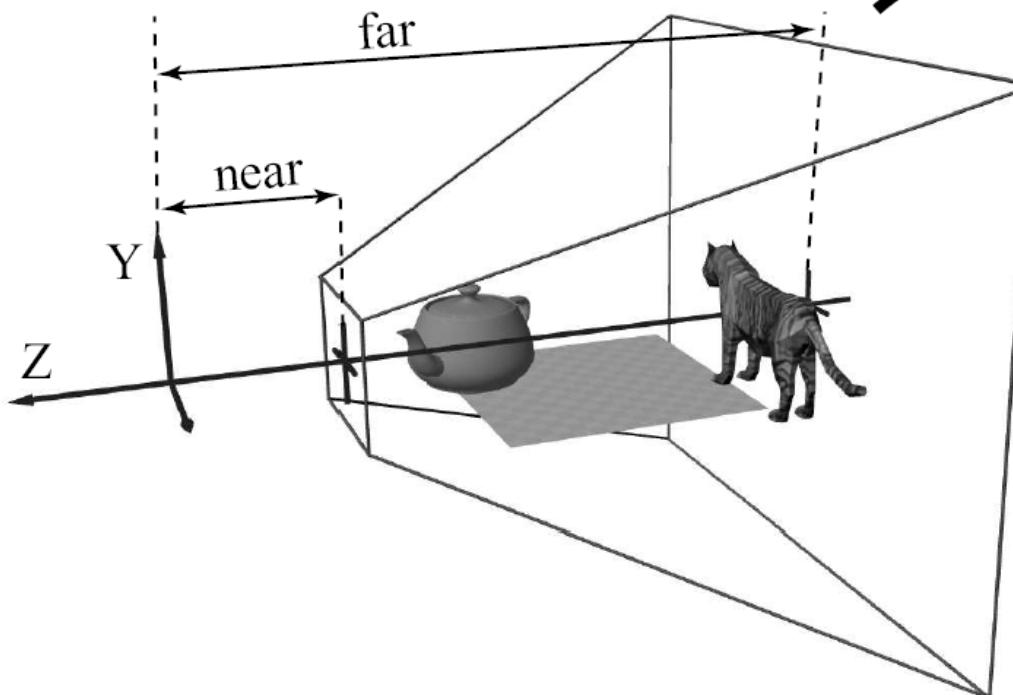
- Default: fill entire HTML-5 Canvas with WebGL image:
- Gain control using this WebGL command:
gl.viewport(xLL, yLL, wid, ht)
- Values in pixels, origin at lower left



View Frustum to CVV Cube

EYE coord system

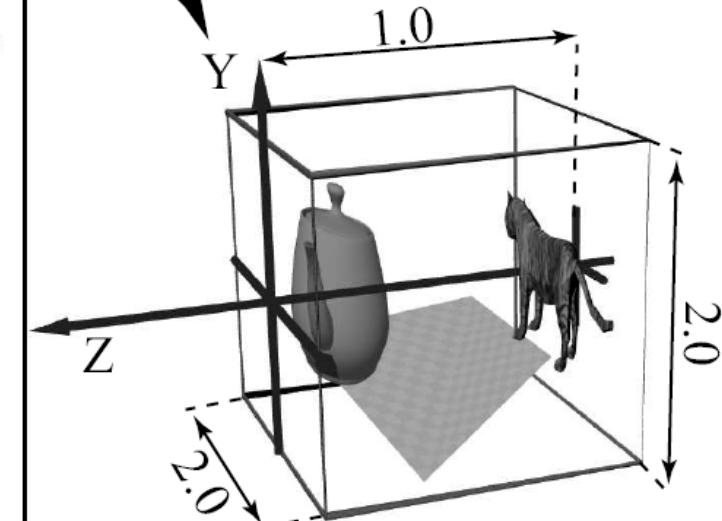
EC Visible Volume (Frustum)



CVV coord system

NDC Rectangular
Visible Volume

Projection Transform



All Together: 3D Camera Summary + Navigation by GUI Functions or

“How to Steer A Glass Cylinder”

Some of these are
J.Tumblin-Modified SLIDES from:
Ed Angel, Professor Emeritus of Computer
Science, University of New Mexico, and
Peter Shirley, Professor, Univ. of Utah

Historic Vertex Position Pipeline

Vertex Stream

gl.ModelView

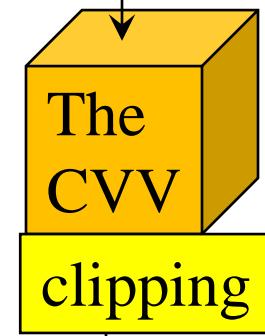
model + **view**
transformation

gl.Projection

projection
transformation

divide:
 $x/w, y/w, z/w$

$4D \rightarrow 3D$

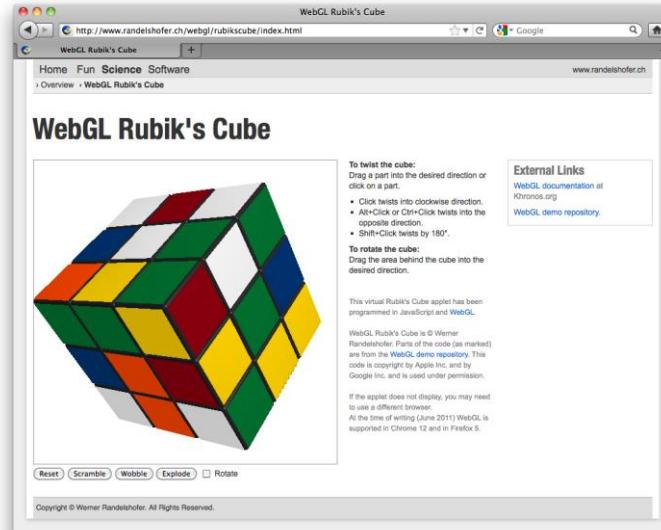


ViewPort
transformation

gl.viewport(lx, ly
width, height)
(in pixels)

“Historic”? Yep...
This **fixed** pipeline
was mandatory in
OpenGL until
version 3.0
replaced it with
programmable
shaders.
Still an excellent
default strategy!

HTML-5 Canvas



WebGL: Unified Scene Graph

RECALL: in scene graphs,

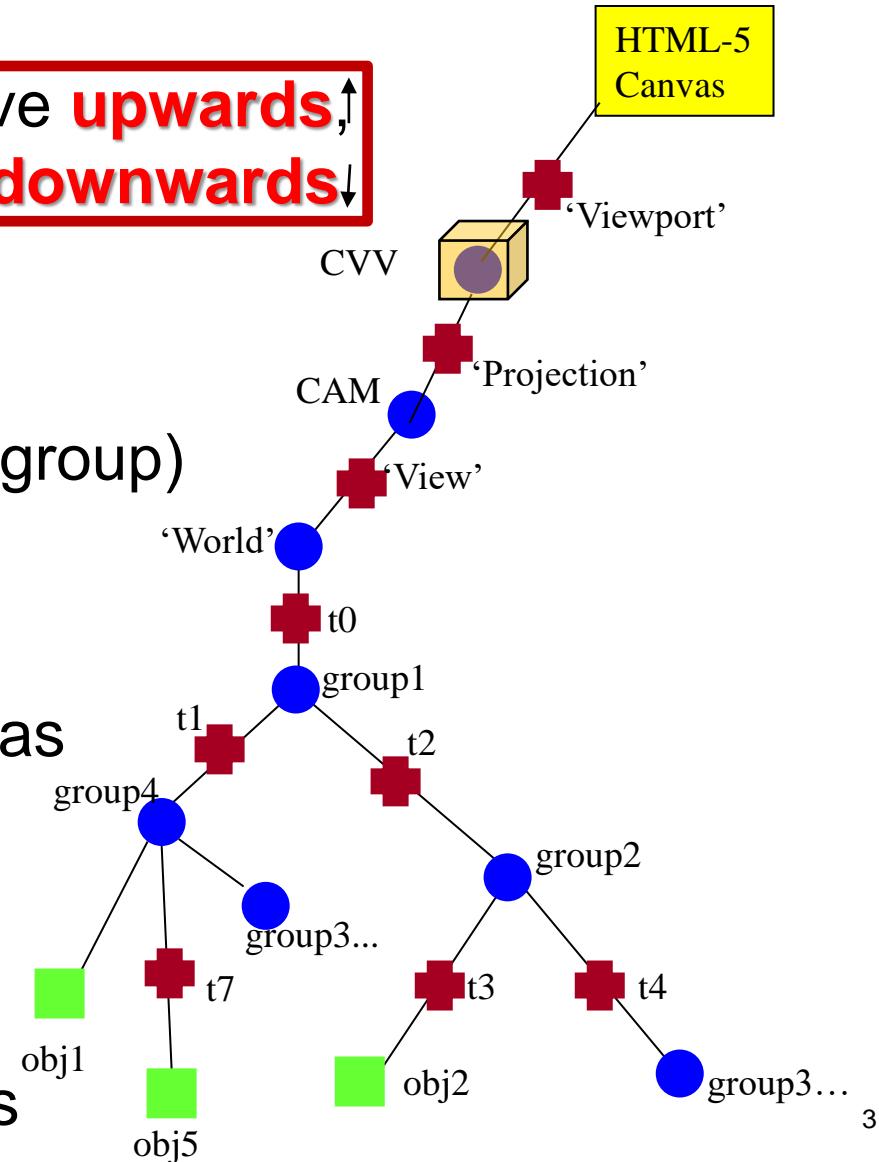
Vertices & position values move **upwards**,
WebGL transform calls move **downwards**

Project A:

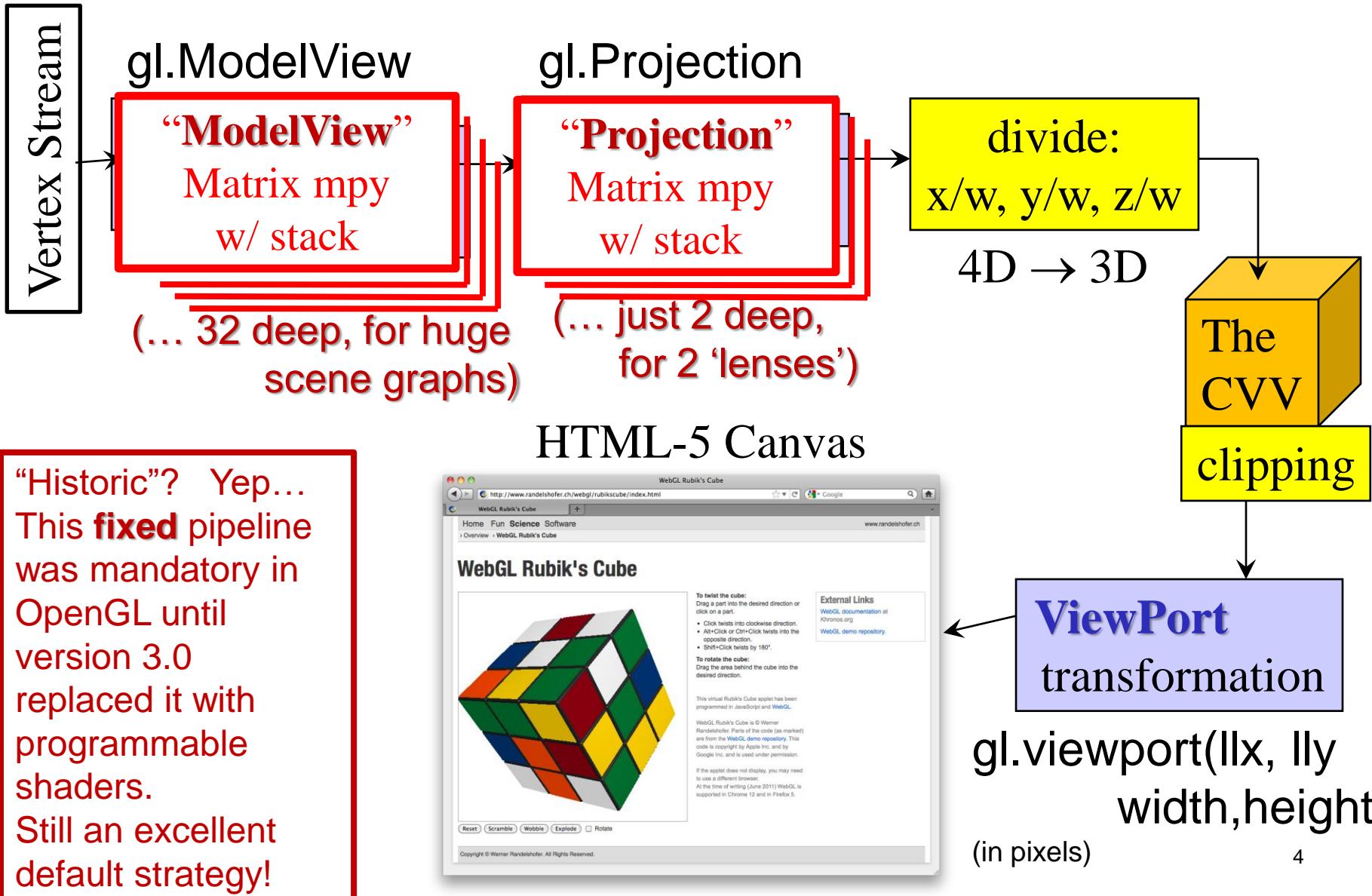
- CVV == “World” drawing axes (group)

Project B:

- Viewport output → HTML Canvas
- CVV output → Viewport input
- Lens Axes → CVV input
- Camera Axes → Lens Input
- ‘World’ Coords → Camera Axes



Historic Vertex Position Pipeline

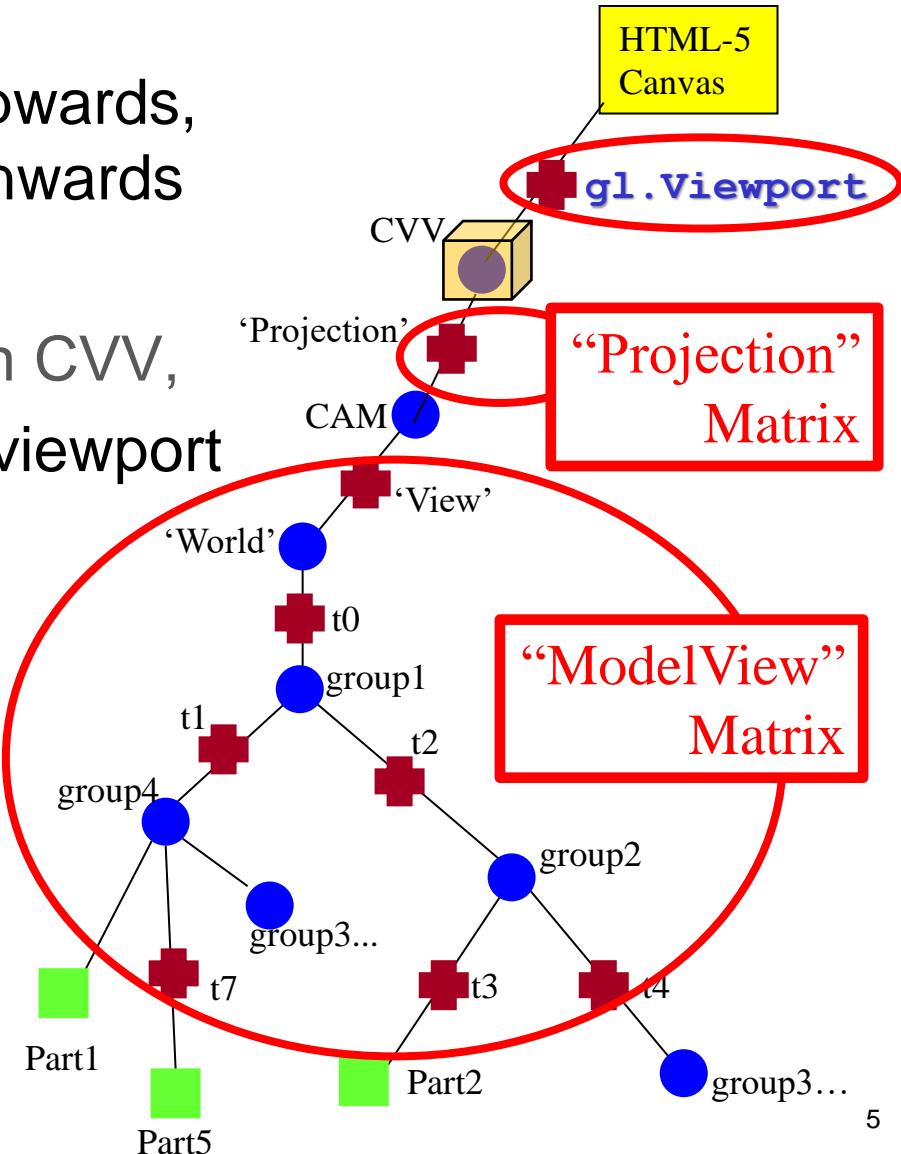


Traditional Implementation (OpenGL)

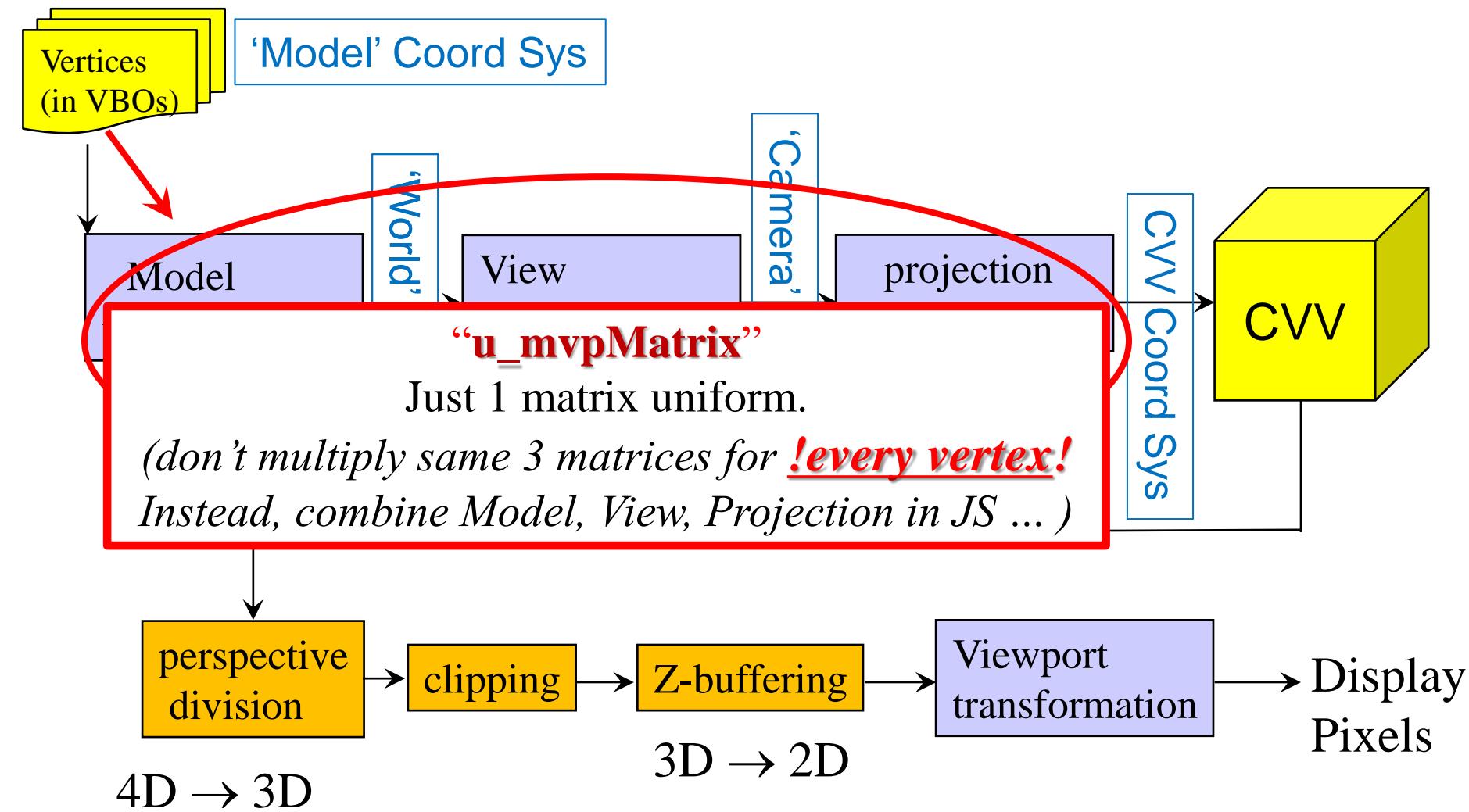
RECALL: in scene graphs,
Vertices & values move upwards,
transform calls move downwards

Project A: Draw ‘world’ directly in CVV,

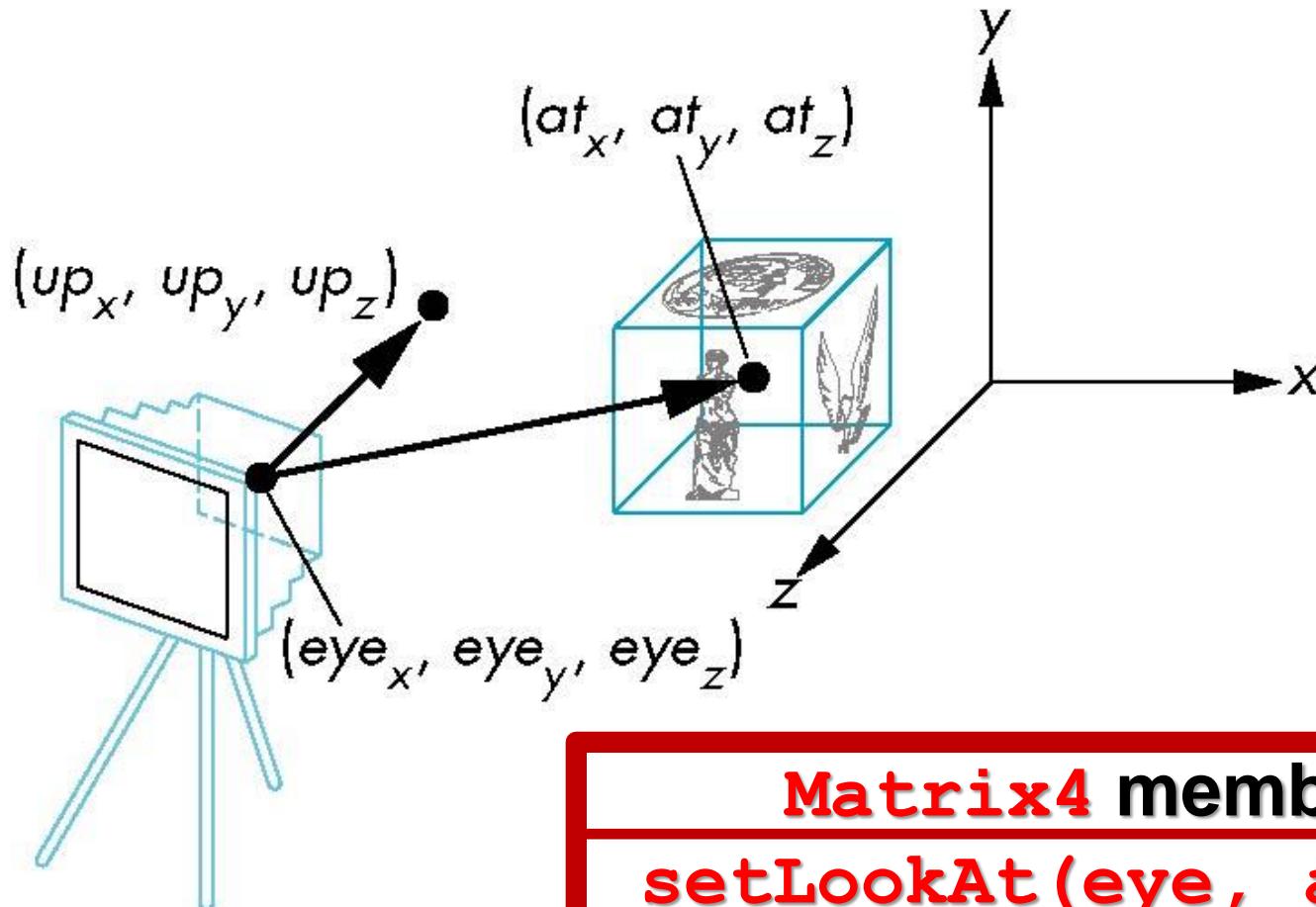
Project B: Add view, projection, viewport



Efficient Vertex Shader: mvpMatrix



LookAt() makes ‘View’ matrix



Matrix4 members :

`setLookAt(eye, at, up);`

OR: `lookAt(eye, at, up);`

(see cuon-matrix-quat.js)

Projection Matrix:

Orthographic

Projection

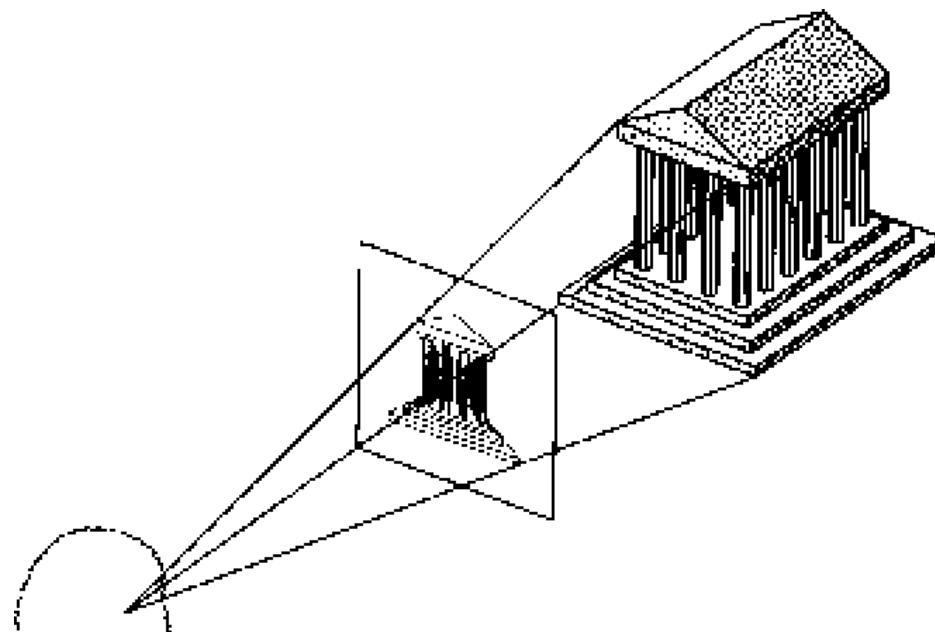
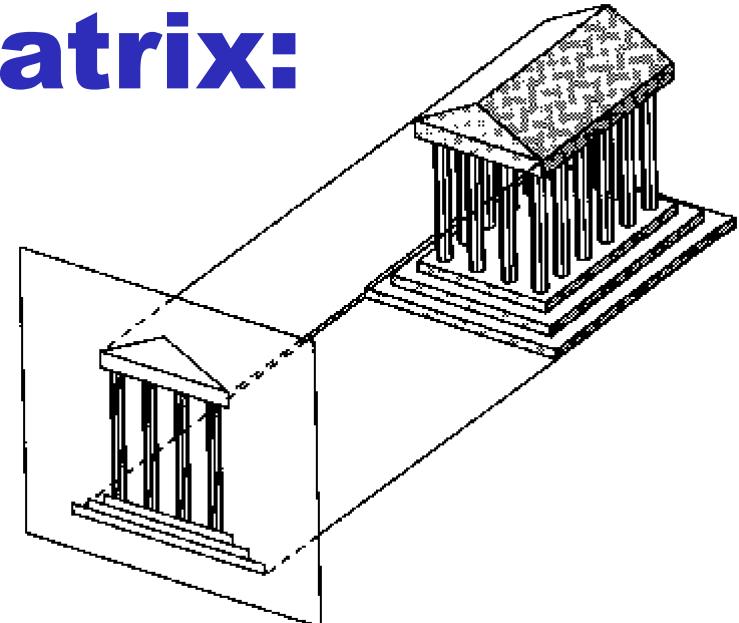
(rare)

vs.

(common)

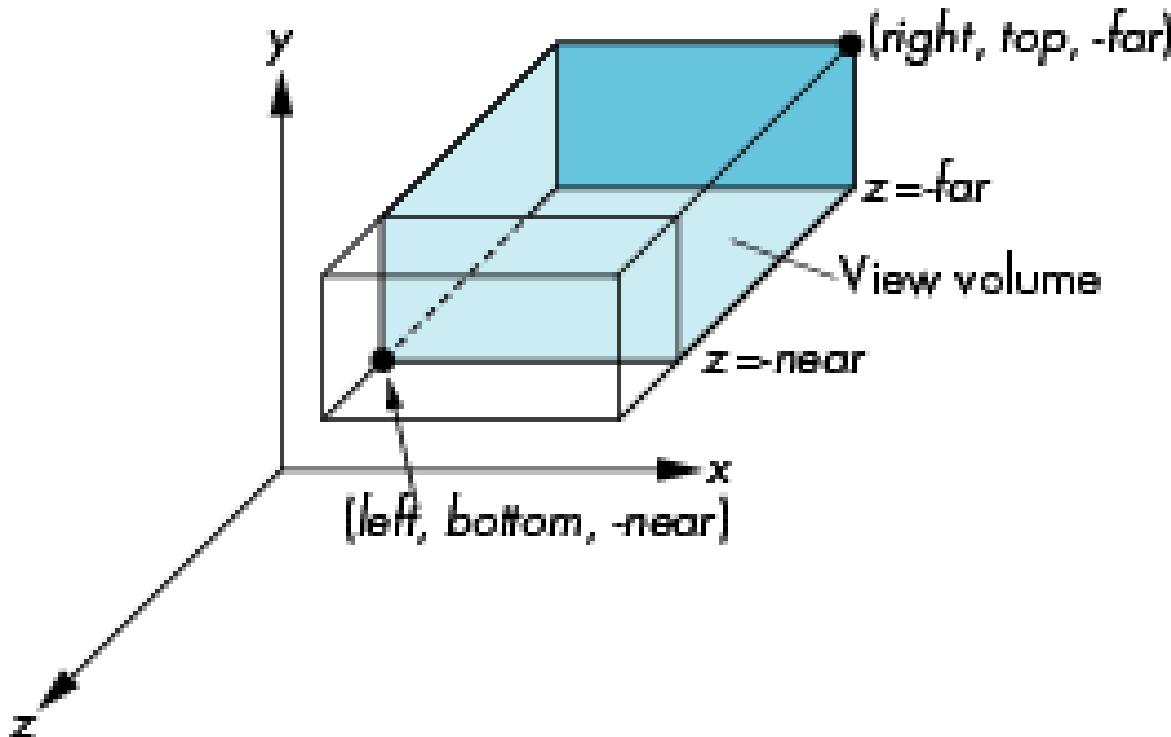
Perspective

Projection



Orthographic Camera

`ortho(left, right, bottom, top, near, far)`
(`Matrix4` member, `cuon-matrix-quat03.js`)



CAREFUL! `near` and `far` are *always positive* numbers!

Orthographic Camera's Visible Volume:

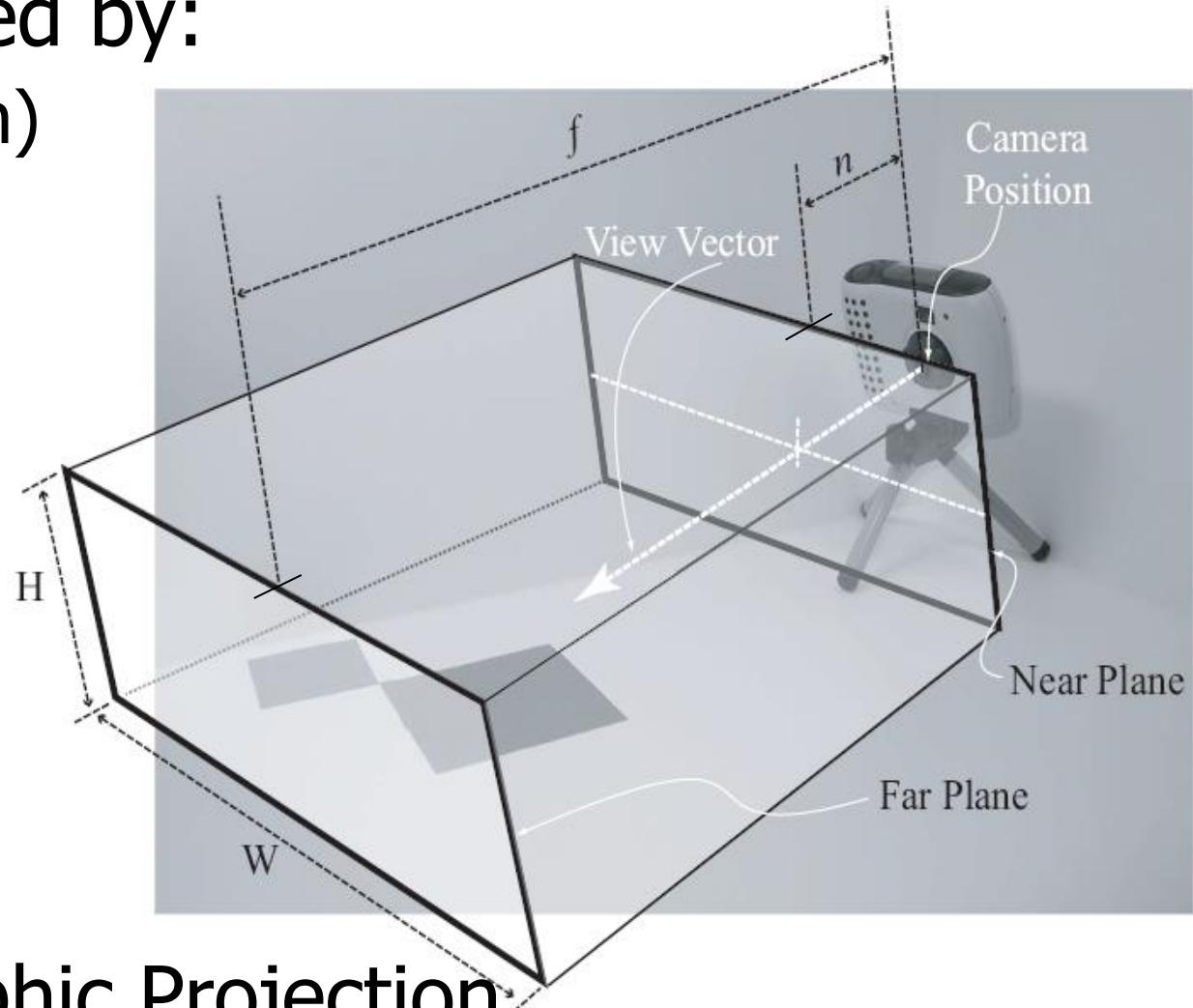
- Volume defined by:

- Near Plane (n)
- Far Plane (f)
- Width (W)
- Height (H)

- (All args positive, >0)

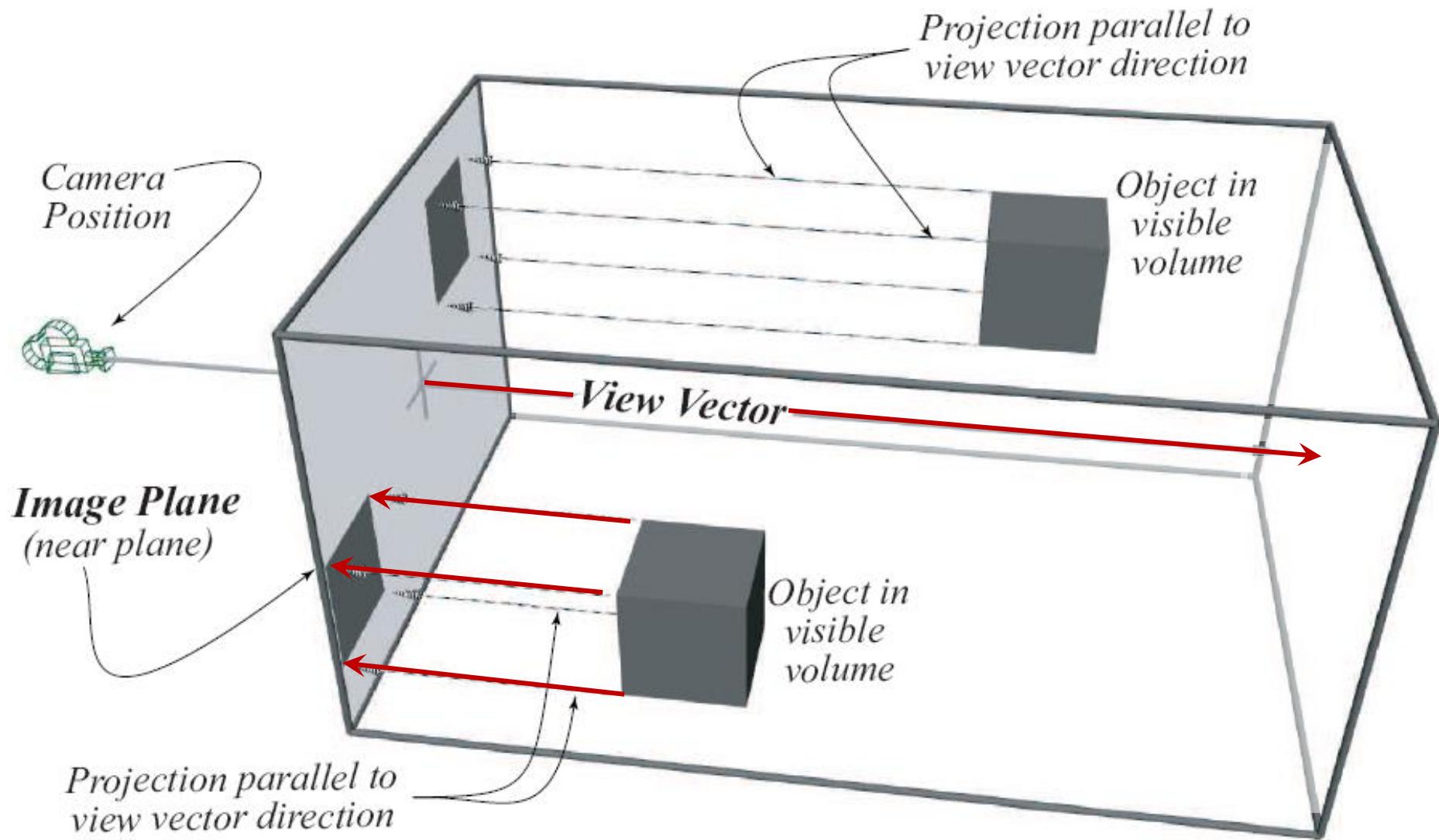
$$\text{width} \times \text{height} \times \text{depth} =$$

$$W \times H \times (f - n)$$



- For Orthographic Projection

Orthographic Projection: Parallel only



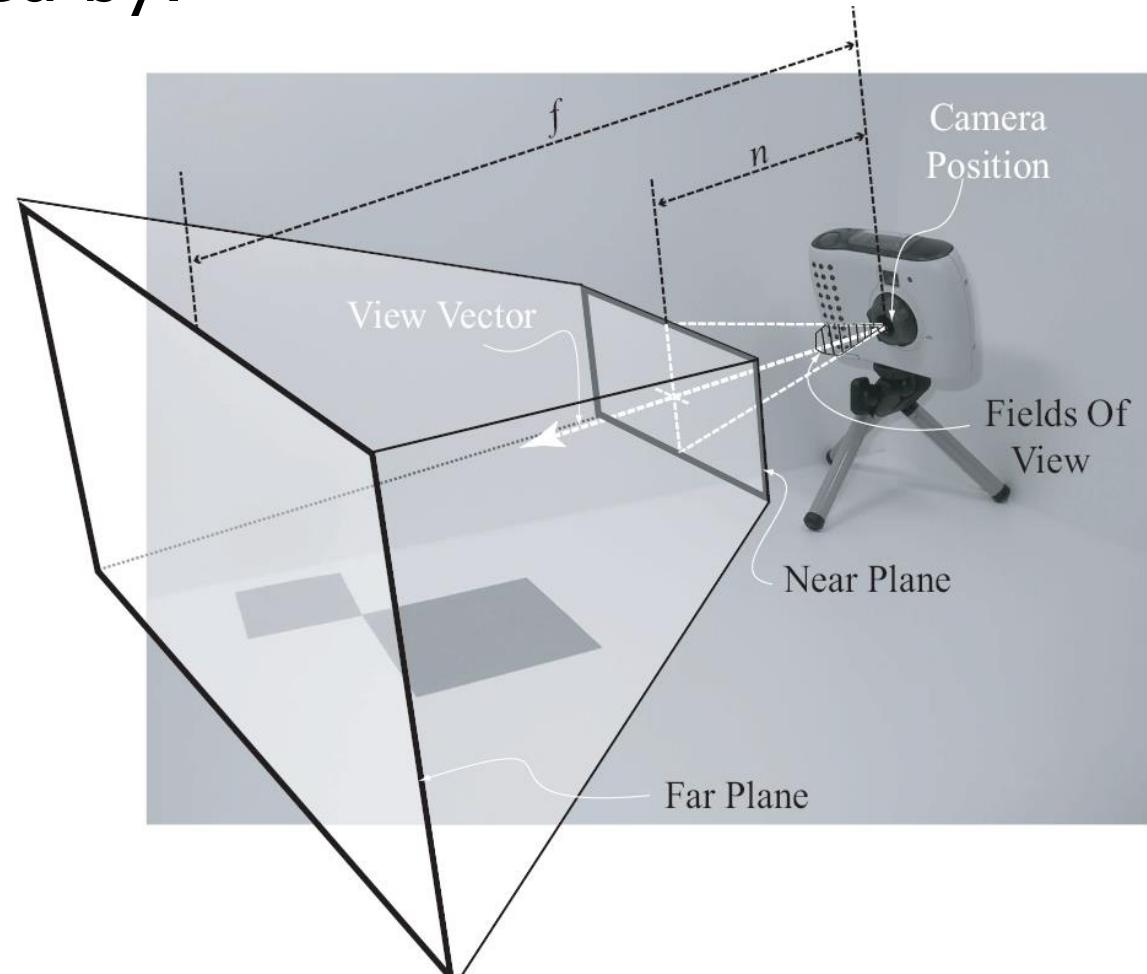
Perspective Camera I:

- View Volume defined by:

- Near Plane (n)
- Far Plane (f)
- Fields of view (fov)

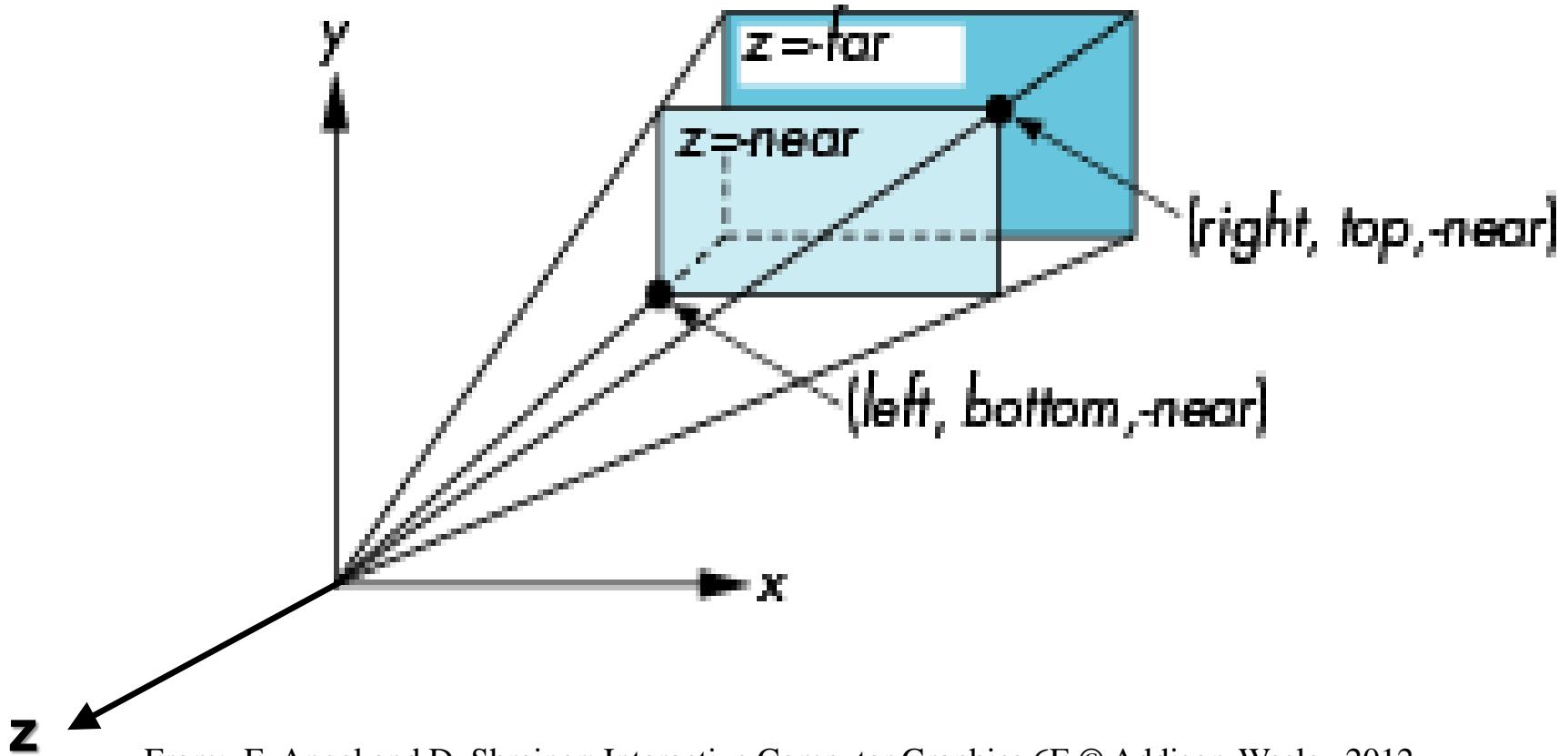
or

sides of the 'frustum'



Perspective Camera I:

frustum(left, right, bottom, top, near, far)



How should I pick **znear** & **zfar**?

Be sensible. Adapt to task & Hdwe. resolution:

- Early/Small hardware depth buffers: 16 bits or just 65,536 unique depth values:
 - If $(\text{zfar} - \text{znear}) = \mathbf{12}$,
smallest resolvable distance = $12/65,536 = \mathbf{0.000183}$
Microscopic! depth errors, z-fighting extremely unlikely
 - If $(\text{zfar} - \text{znear}) = \mathbf{10^6}$,
smallest resolvable distance = $10^6/65,536 = \mathbf{15.259}$
Huge! Only the largest 3D parts avoid depth errors & z-fighting!
- General Guide:
 - *Small-ish* **zfar** and *Large-ish* **znear**
 - How? → Be **sensible**; not needlessly fanatical

Modern GPUs offer
32-bit floating-point
for depth & geometry



Field-of-View (FOV) Angles & Aspect Ratio

Common Lens Specifications:

- **Aspect Ratio (W/H)? Easy:**

$$\text{Aspect} = (\text{right} - \text{left}) / (\text{top} - \text{bottom})$$

- **Field-of-View (FOV) Angles?**

Measure from $-z$ axis in 2 parts:

- Vert. FOV = $\theta_{\text{top}} + \theta_{\text{bot}}$
- Horiz. FOV = $\theta_{\text{right}} + \theta_{\text{left}}$

- To **specify** frustum limits **from FOV angles**:

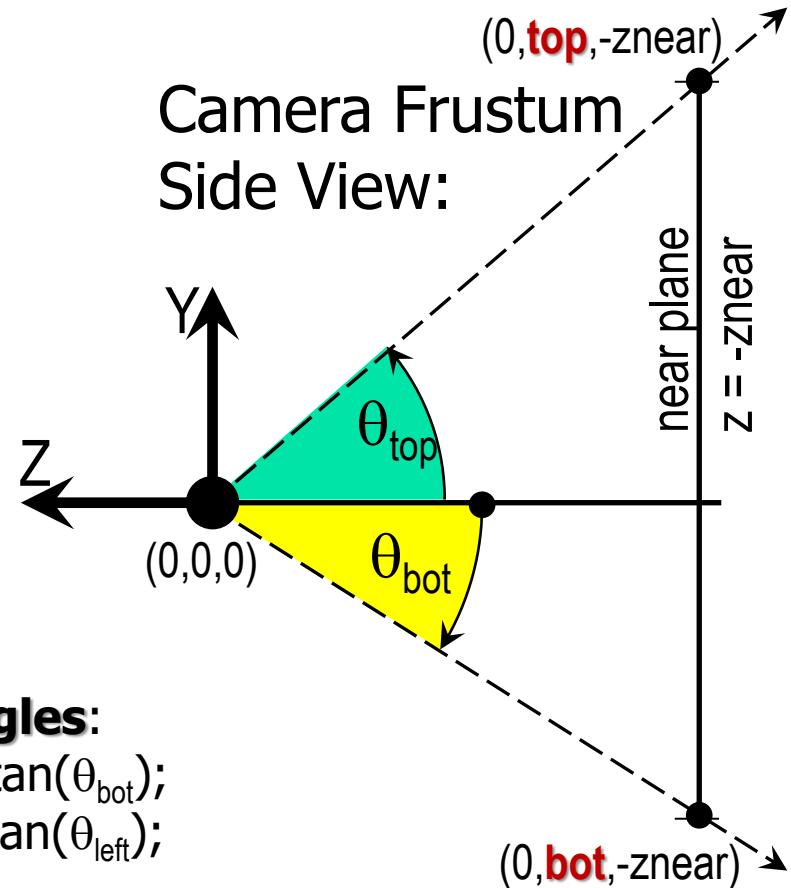
$$\begin{aligned}\text{top} &= \text{znear} \cdot \tan(\theta_{\text{top}}); \quad \text{bot} = -\text{znear} \cdot \tan(\theta_{\text{bot}}); \\ \text{right} &= \text{znear} \cdot \tan(\theta_{\text{right}}); \quad \text{left} = -\text{znear} \cdot \tan(\theta_{\text{left}});\end{aligned}$$

- To **find FOV angles** from frustum limits:

$$\text{Vert. FOV} = \arctan(\text{top} / \text{znear}) + \arctan(\text{bot} / -\text{znear});$$

$$\text{Horiz. FOV} = \arctan(\text{right} / \text{znear}) + \arctan(\text{left} / -\text{znear});$$

- **Tangents? Arctangents? UGLY; not intuitive!**

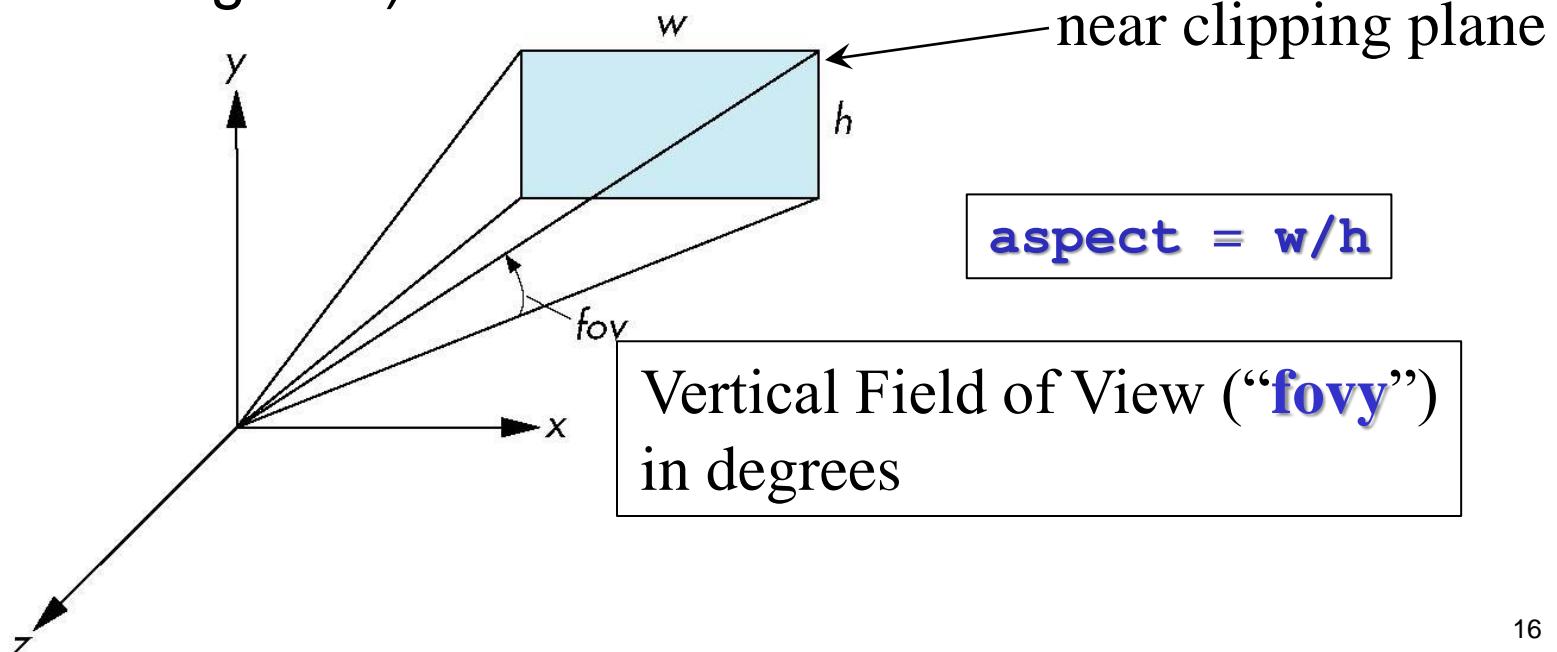


Perspective Camera II

Using **frustum()** is often frustrating, complex:
? isn't there an easier way ? YES--

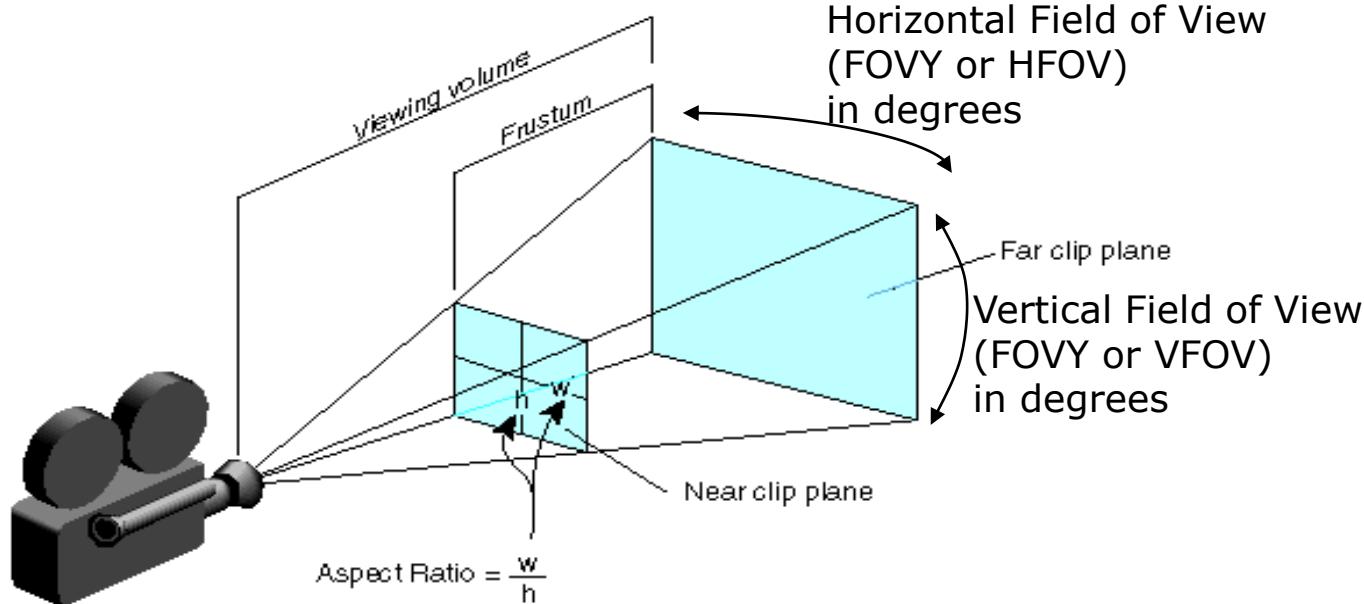
perspective(fovy, aspect, near, far);

symmetric, & uses commonplace camera values:
(no arc-tangents!)



Perspective() function

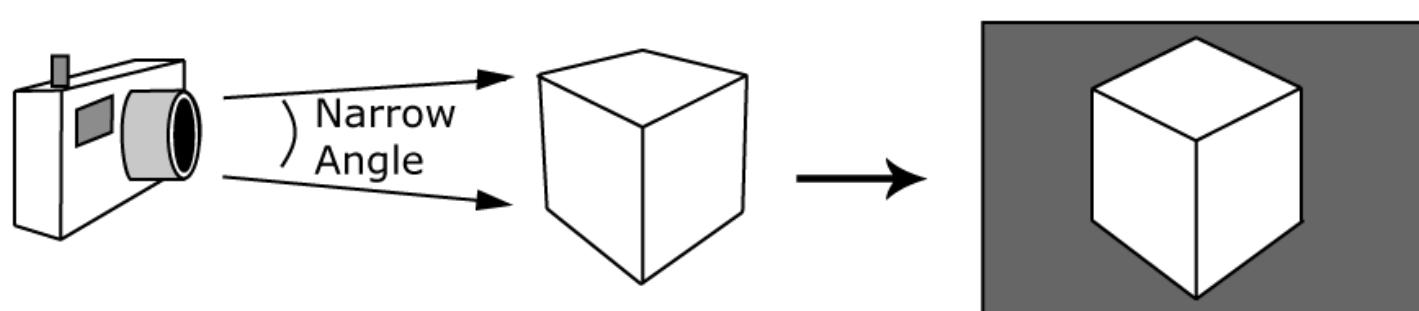
- Creates projection matrix for *symmetric* camera
 - (Method 1: CAM coordinate numbers → CVV coordinate numbers)
 - (Method 2: CVV drawing axes → CAM drawing axes)
- Specify horiz. field of view, aspect ratio, near, far.
- **WARNING:** near >0, far >0!
despite right-handed camera that 'looks down the CAM -z axis!')



DEMOS! http://learnwebgl.brown37.net/08_projections/projections_perspective.html

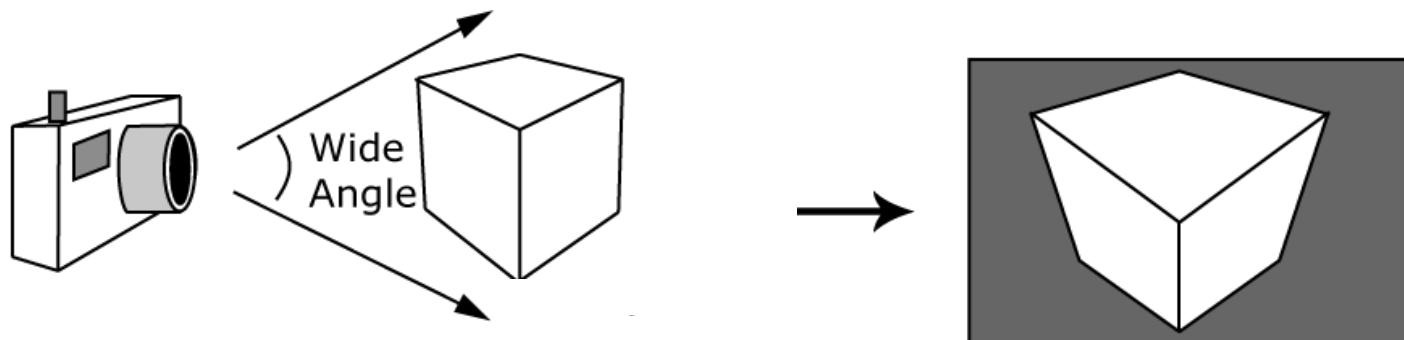
Perspective() function

- 'Long', 'Narrow', 'Telephoto' or 'Telescopic' Lens?
 - Narrow field-of-view: small FOVY (e.g. $< 20^\circ$)
 - High magnification: tiny angle fills the screen
 - Nearly parallel rays from COP to scene:
 - very little foreshortening (looks almost orthographic)
 - horizon forms very, very far away: needs large zfar to see it!

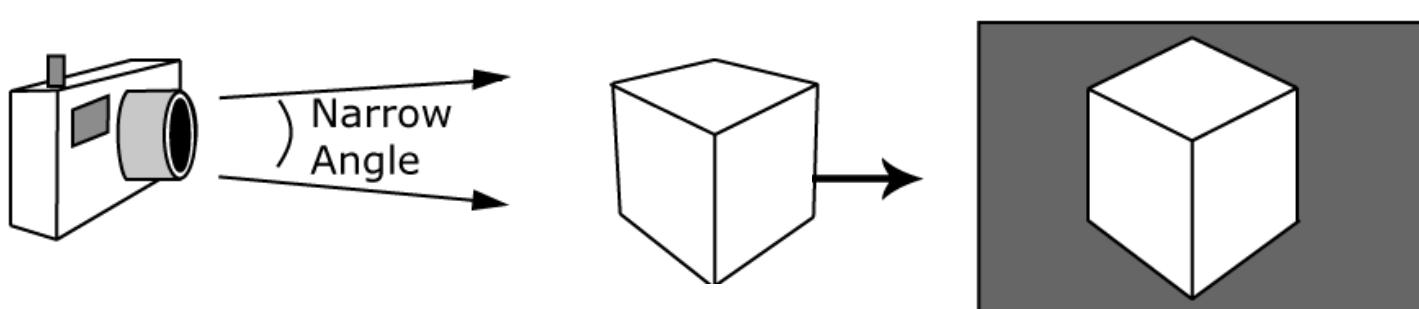


Perspective() function

- 'Short' or 'Wide' or 'Fish-Eye' or 'Panoramic' Lens?
 - wide field-of-view: large FOVY (e.g. $> 45^\circ$)
 - STRONG foreshortening: nearby objects look **HUGE**



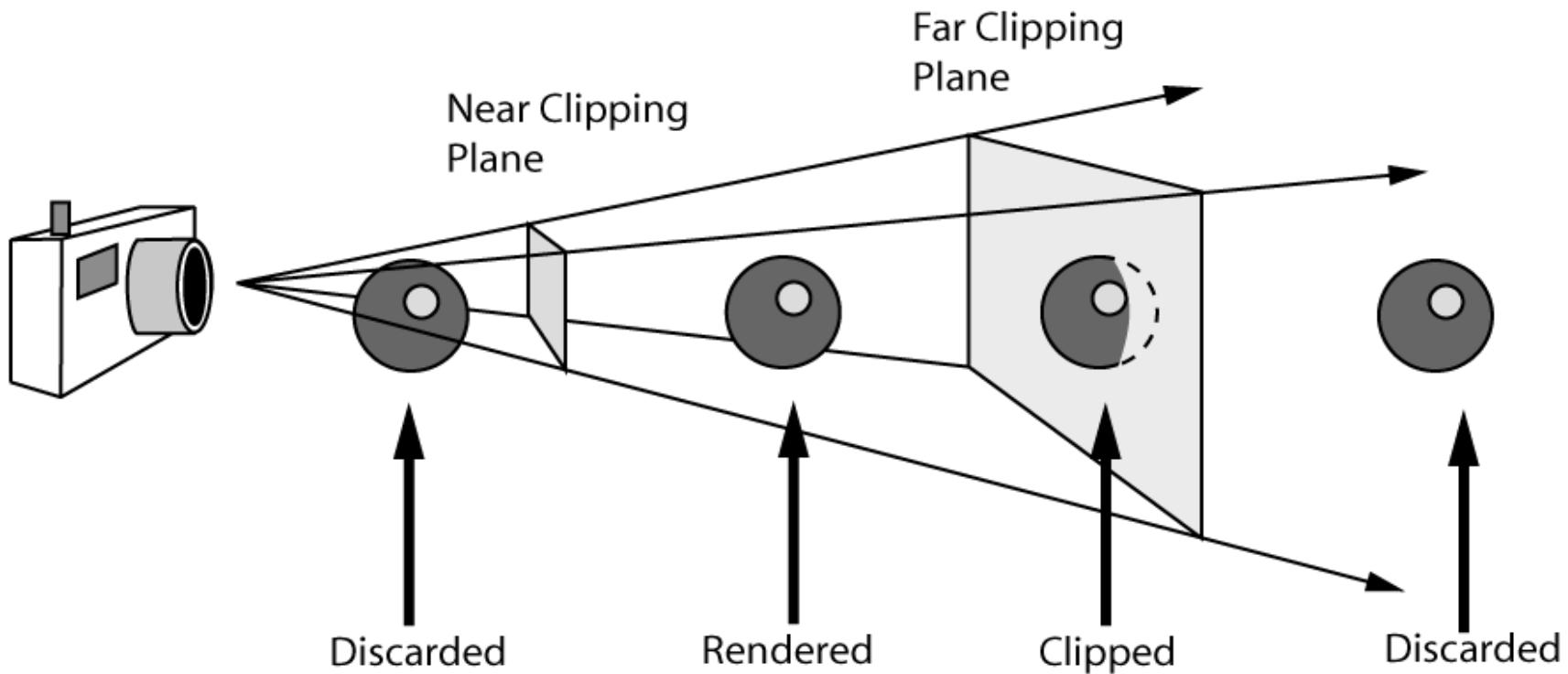
- "**Normal**" lens? about 35° - 40° FOVY, Aspect: 4/3



Unforgiving 'Clipping Plane' Effects

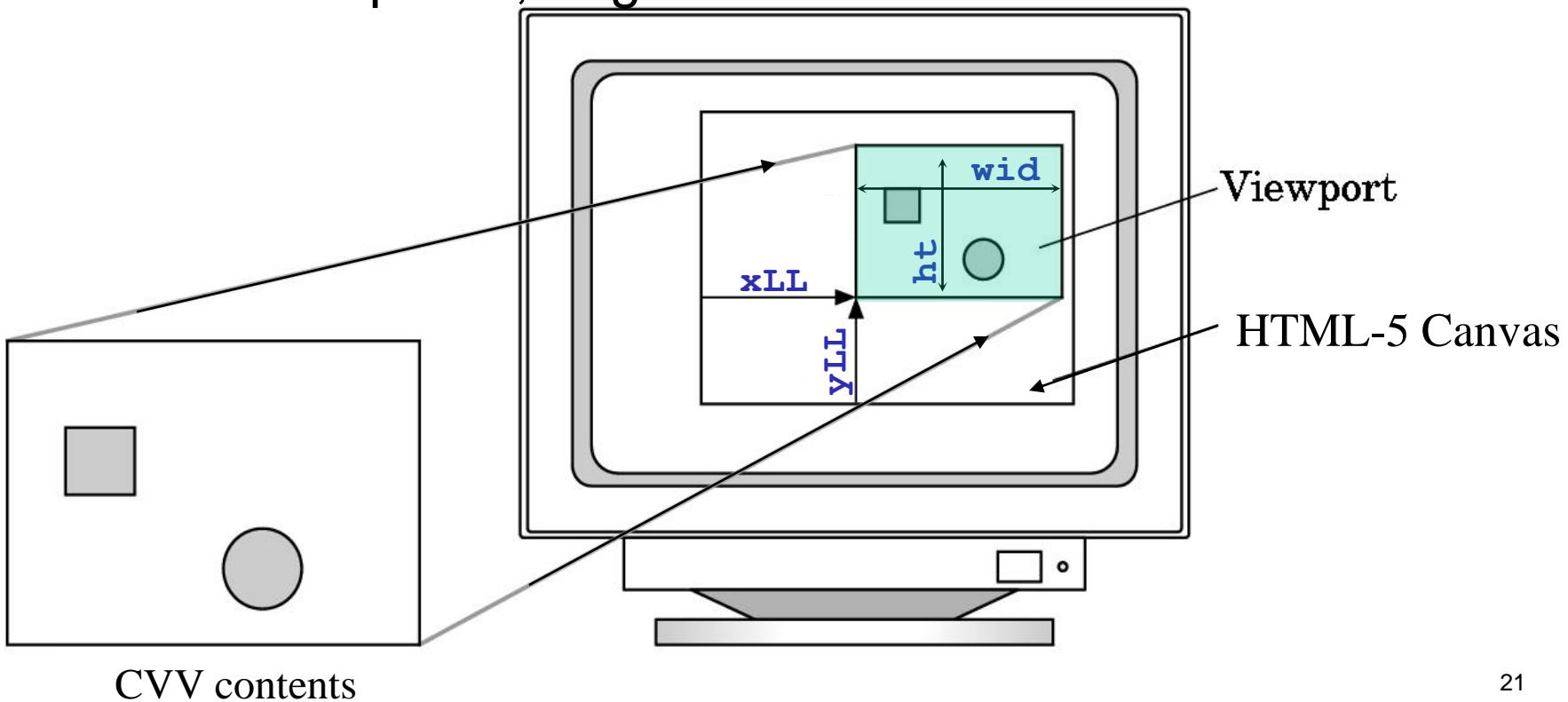
Careful! CVV clipping 'cuts' any and all WebGL drawing primitives beyond its limits.

----(Nate Robins Tutor Demo)----



Viewport: Maps CVV to the Canvas

- Default: fill entire HTML-5 Canvas with WebGL image:
- Gain control using this WebGL command:
gl.viewport(xLL, yLL, wid, ht)
- Values in pixels, origin at lower left



MULTIPLE Viewports? HOW?

Make multiple calls to

gl.viewport(xLL, yLL, width, height)

inside your **drawAll()** function:

- Left side: **glViewport(0, 0, w/2, h/2);**
< draw stuff >
- Right side: **glViewport(w/2, 0, w/2, h/2);**
< draw stuff >
- And so forth – make your own custom layouts!

Multiple Viewports!

-----DEMO-----

See 2021.11.01Cameras3.1 Starter code:

→Ch07_Book →

7.07b.JT.LookAtSceneWithKeys_4ViewVolume.html

See line 96:

```
projMatrix.setPerspective(30, canvas.width/canvas.height, 1, 100);
```

Important! Not
'1.0' anymore!

**?What if we RESIZE the canvas?
non-square? 600w x600h → 600w x 300h?**

RE-SIZEABLE Canvas? YES!

1) In your HTML file, create canvas with sensible **initial** size:

```
<canvas id="webgl" width="600" height="300">  
    Please use browser that supports HTML-5 "canvas"  
</canvas>
```

2) In HTML file, set browser-window resizing to call your JS fcn:

```
<body onload = "main()" onresize="drawResize()">
```

3) In your JS file, in your resizing function:

- current browser window-size (in pixels; read-only) given by:

window.innerWidth, **window.innerHeight**

(or you can use just **innerWidth**, **innerHeight**)

- current HTML-5 canvas size (in pixels; read/write) is set by:

canvas.width, **canvas.height**

(RECALL: **var canvas = document.getElementById('webgl');**)

RE-SIZEABLE Canvas display? YES!

CAREFUL!

Initializing re-sizeable canvases can be tricky

HINTS:

- Unlike book's starter code, **use global variables** (`gl`, `g_canvas`, `g_innerWidth`, `g_innerHeight`, `mvpMatrix`, etc) to avoid unnecessary function arguments.
- Make separate '`drawAll()`' and '`drawResize()`' functions.
- Use no arguments for `drawAll()` and `drawResize()` functions.
- Be sure to call your `drawAll()` function at the end of `drawResize()`
- Before you begin animations in `main()`,
call `drawResize()` function to correctly initialize for the HTML-5 canvas size.

Adjustable Camera Aspect Ratio Enables Distortion-Free Window Re-sizing!

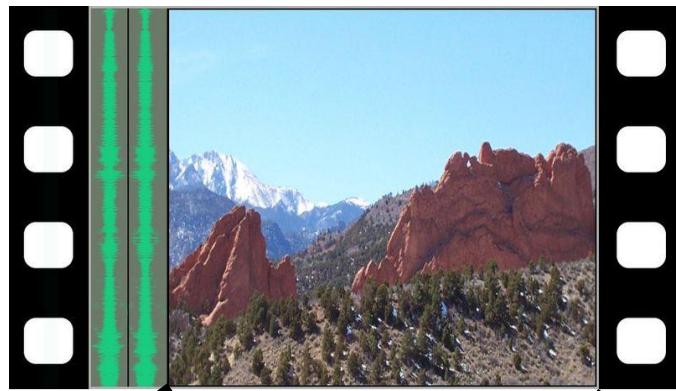
-----DEMO-----

See [2021.11.01.Cameras3.1](#) starter code:

[**7.11.JTHelloCube_Resize.html**](#)

Non-Square HTML-5 Canvas Display : 1

35mm movie film frame



Hollywood ‘**stretches**’ fixed-size
35mm film for ‘wide-screen’ displays:
1.33:1 aspect ratio film image
stretched to fill
2.35:1 aspect ratio screen.



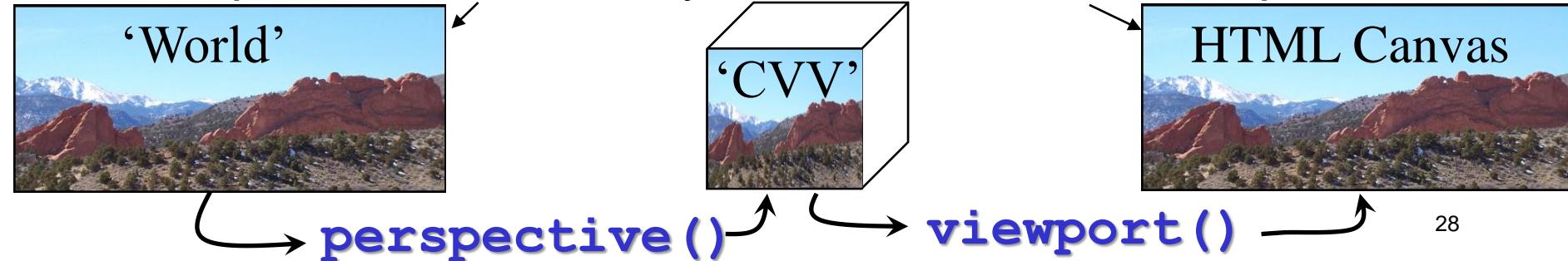
Camera Lens ‘**squeezed**’
wide-screen image onto film
using non-square ‘anamorphic’
lens:



Non-Square HTML-5 Canvas Display: 2

SAME IDEA:

- By default, WebGL ‘**stretches**’ fixed-size CVV contents to fill the entire HTML-5 canvas display:
1:1 aspect ratio of Canonical View Volume
stretched to fill
any aspect ratio HTML-5 Canvas
(CAREFUL! `gl.viewport()` may change default CVV→Canvas mapping)
- Camera Lens-like `perspective()` ‘**squeezes**’ its image aspect-ratio to fit into the CVV. Be sure your ‘camera lens’ aspect ratio matches your HTML-5 Canvas aspect ratio!



Non-Square HTML-5 Canvas Display: 3

Example:

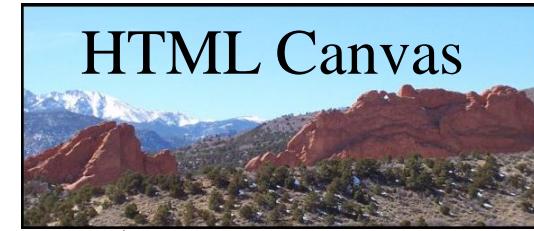
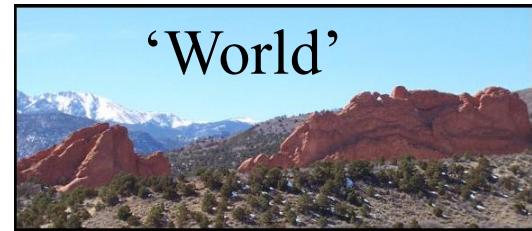
Wide HTML-5 canvas element: aspect ratio **2** (==width/height):

```
<canvas id="webgl" width="600" height="300">  
    Please use browser that supports HTML-5 "canvas"  
</canvas>
```

By default, WebGL ‘**stretches**’ CVV contents to fill canvas

- In WebGL, use wide camera-lens-like ‘projection’ matrix:

```
g_myMatrix.setPerspective(42.0,          // VFOV deg,  
    canvas.width / canvas.height,        // aspect  
    1.0, 100.0);                      // near,far
```



perspective() → **(default)** →

‘Flying’ the 3D Camera

- An interesting 3D scene requires a great deal of 3D camera control:

<http://youtu.be/IJhID6q71YA?t=29s>

How can we **move** the camera
and **aim** the camera easily & intuitively,
and do it smoothly, fluidly?

Interactive “3D Flying” Controls

`LookAt()` puts the camera anywhere, but
?!? **NINE** numbers?! (`eyex`, `eyey`, `eyez`,
`aimx`, `aimy`, `aimz`,
`upx`, `upy`, `upz`)

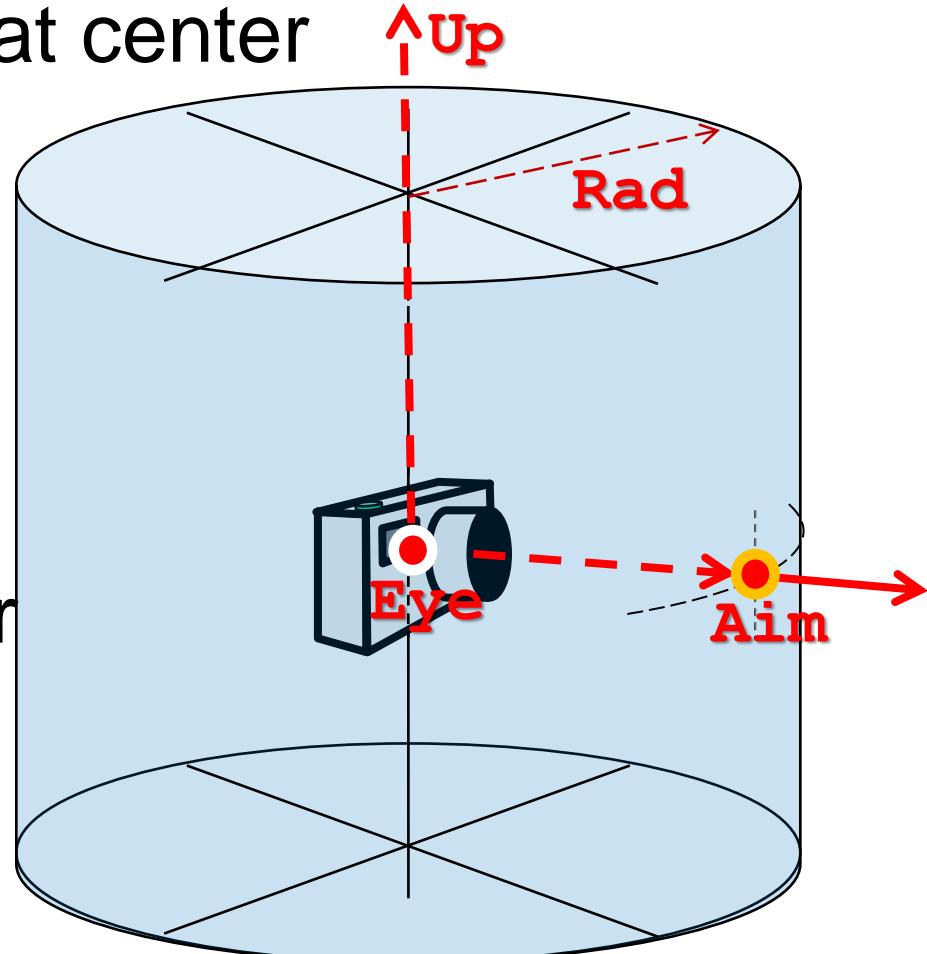
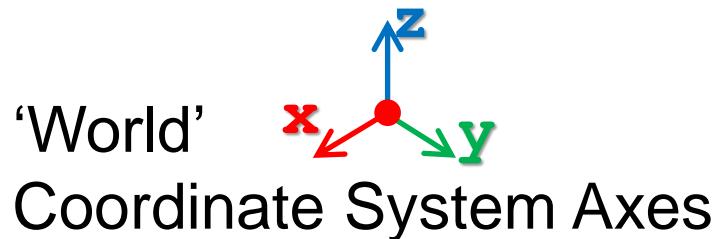
How can keyboard/mouse interactions
set all 9 of them sensibly, easily, simply,
using only the mouse & the keyboard?

How can we
‘steer’ in 3D to fly anywhere, look anywhere?

Interactive “3D Flying” Controls

One solution: The invisible ‘Glass Cylinder’

- Camera eyepoint **Eye** at center
- Cam Aim point **Aim** on the cylinder
- Cyl. Radius **Rad** = 1.0,
 $\| \text{Aim}-\text{Eye} \| = 1.0$
- Cyl. Axis == ‘Up’ vector



Interactive “3D Flying” Controls

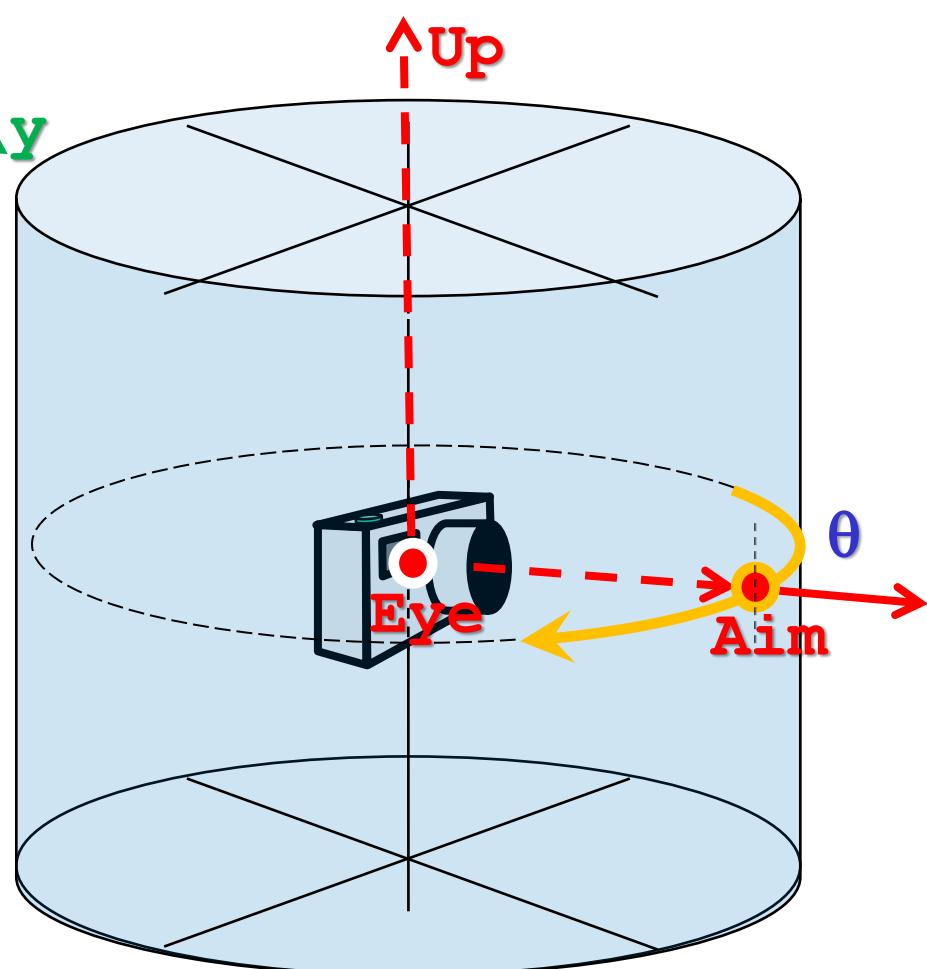
One solution: The invisible ‘Glass Cylinder’

To steer the Camera horizontally to compass angle θ , use this Aim point:

Aim: $(\text{aim}_x, \text{aim}_y, \text{aim}_z, 1)$

$$\text{aim}_x = \text{eye}_x + \cos(\theta)$$

$$\text{aim}_y = \text{eye}_y + \sin(\theta)$$



Interactive “3D Flying” Controls

One solution: The invisible ‘Glass Cylinder’

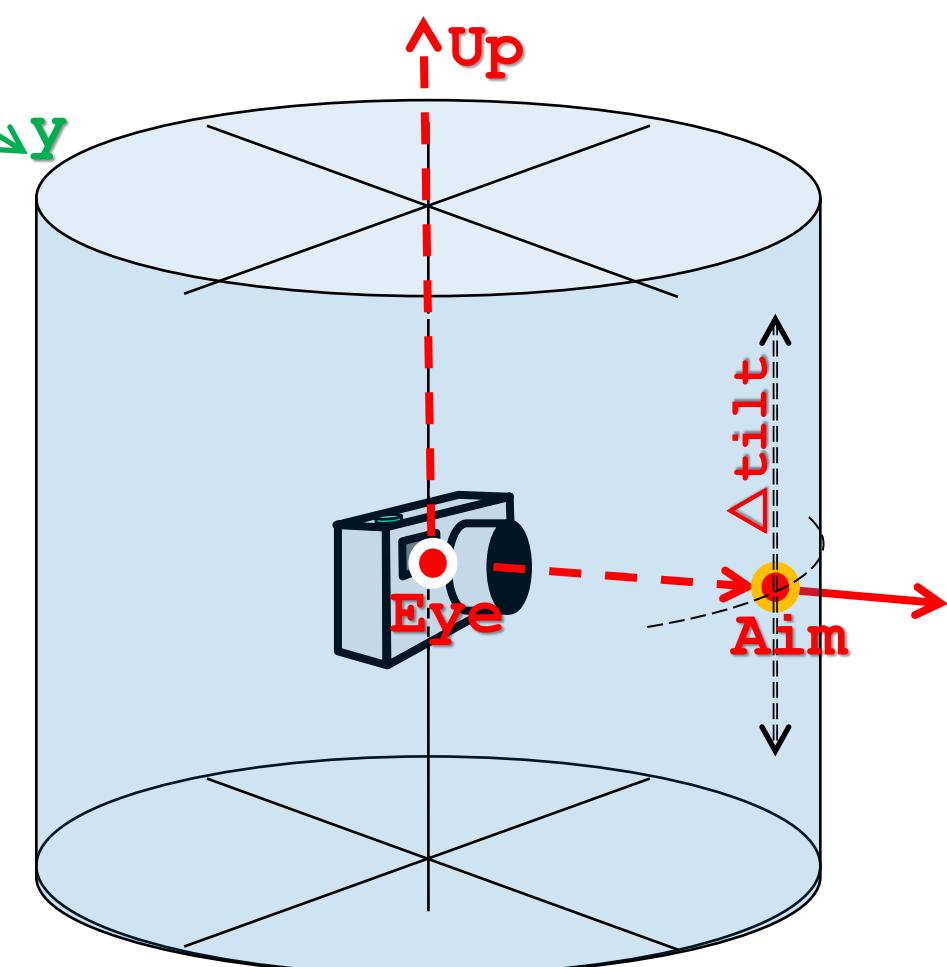
To **tilt** the Camera vertically up or down, change only the Aim point:

Aim:

$$\text{aim}_x = \text{eye}_x + \cos(\theta)$$

$$\text{aim}_y = \text{eye}_y + \sin(\theta)$$

$$\text{aim}_z += \Delta\text{tilt}$$



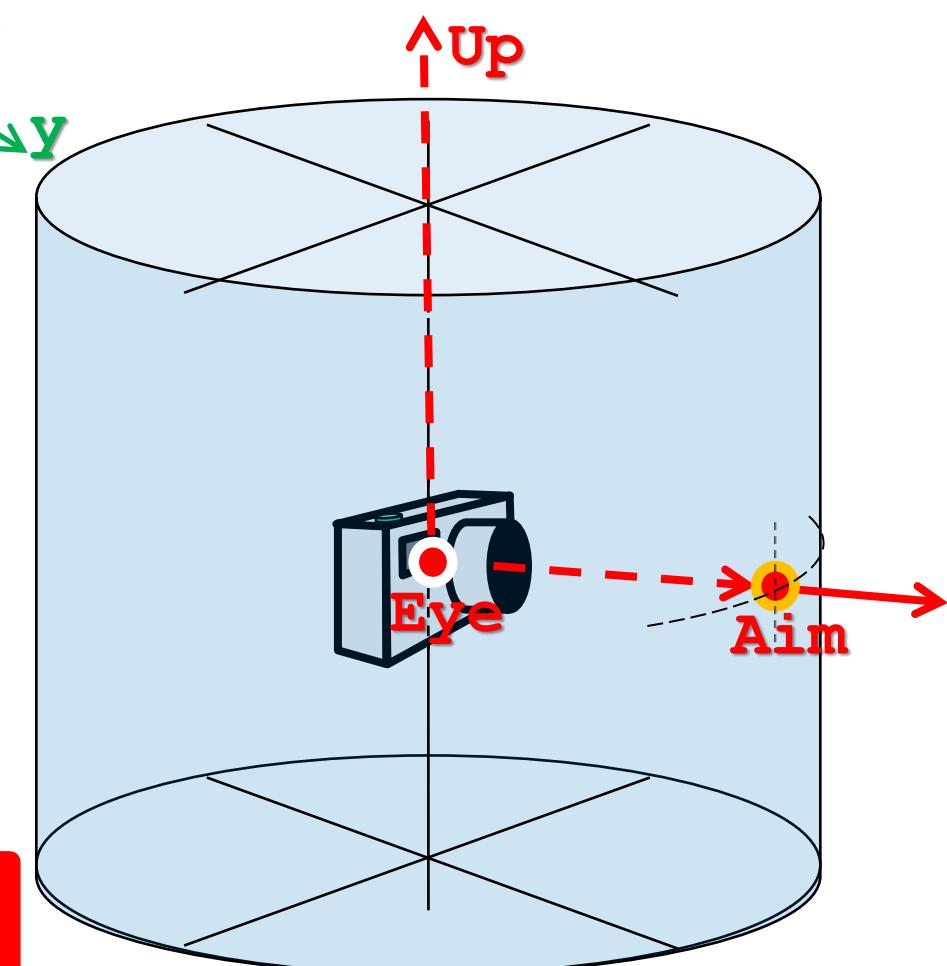
Interactive “3D Flying” Controls

One solution: The invisible ‘Glass Cylinder’

- To **Move/Displace** The Camera without ‘steering’, you must

add displacement to **both**
--the eyepoint **and**
--the look-at point

BUT Store only
Eye, theta, and Δ tilt ...



Interactive “3D Flying” Controls

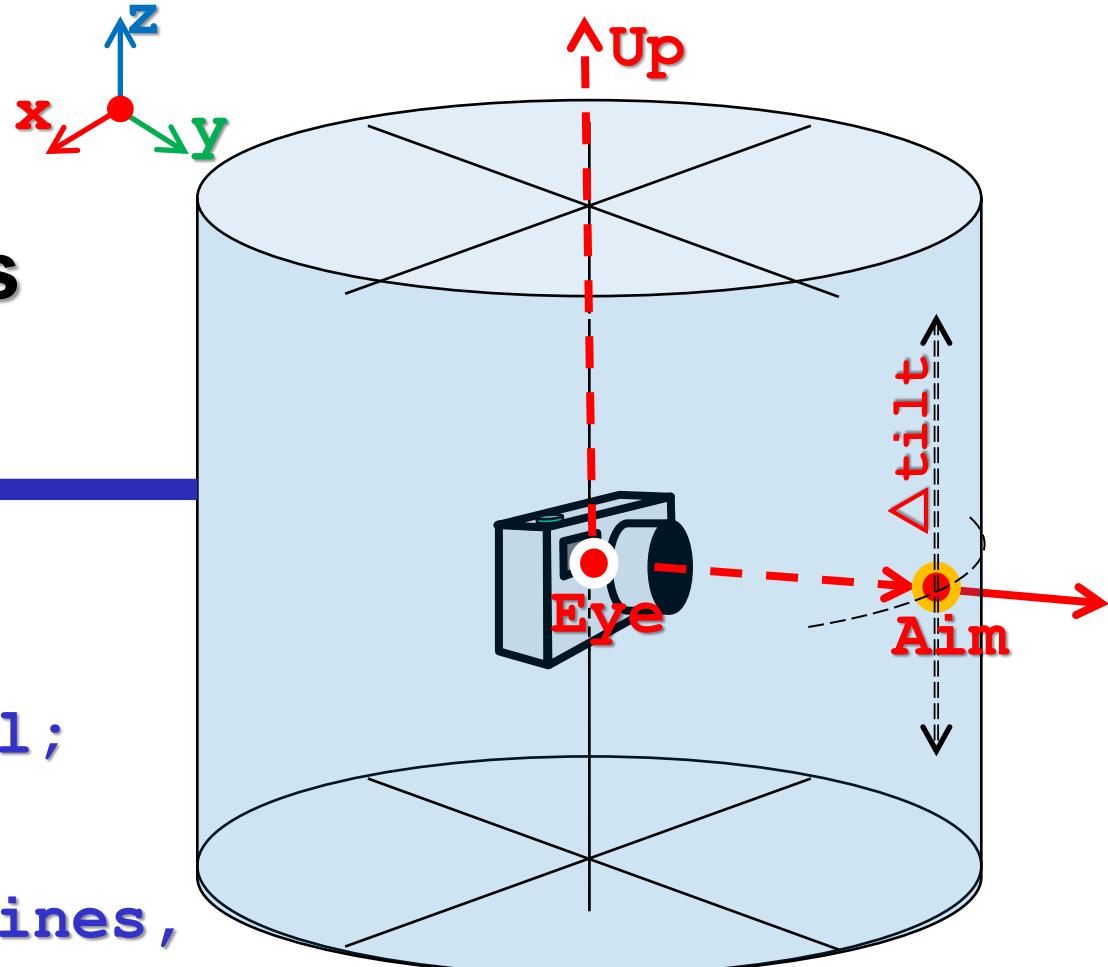
One solution: The invisible ‘Glass Cylinder’

To fly the Camera
‘forward’ (towards
Aim point **Aim**)?

Displace by the
viewing direction:

$$D = (\text{Aim} - \text{Eye}) * \text{vel};$$

(but that's just sines,
cosines, and Δtilt !)



Interactive “3D Flying” Controls

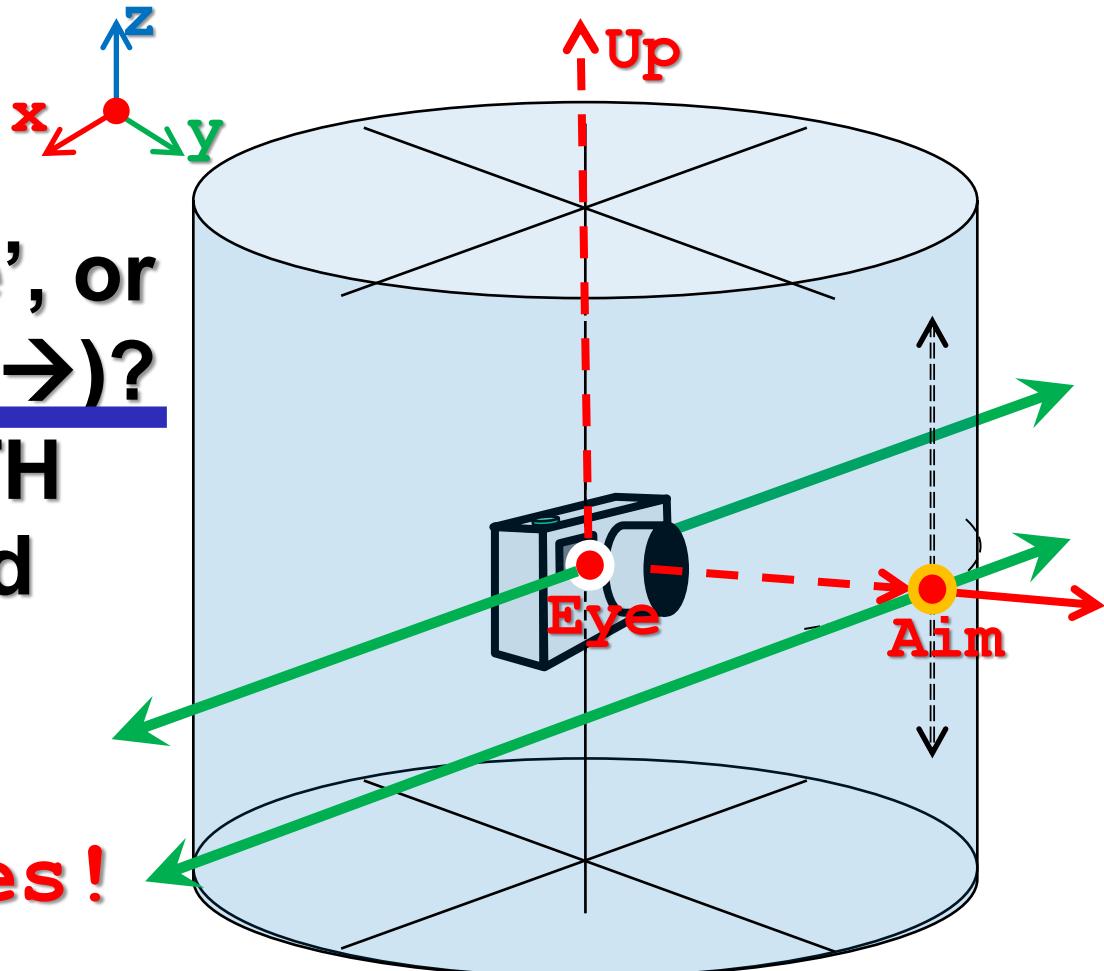
One solution: The invisible ‘Glass Cylinder’

QUESTION:

How do we ‘strafe’, or
move sideways ($\leftarrow\rightarrow$)?

Must change BOTH
Eye point **Eye** and
Aim point **Aim**:

HINT: use
sines & cosines!



ASIDE: Recall MATRIX DUALITY?

TWO ways to describe geometric effects of a matrix:

- **Method 1:** (for making hardware)
 - One fixed coordinate system (CVV); then
 - Matrix changes the vertex coord. **numbers**.
- **Method 2:** (for making **WebGL** calls)
 - One fixed set of coord. numbers (in VBO);
 - Matrix changes the drawing **axes** used to draw those fixed numbers.

What is it for **View** and **Projection** matrices?
(can be confusing – volumes, not vertices!)

ASIDE: Duality for View Matrix:

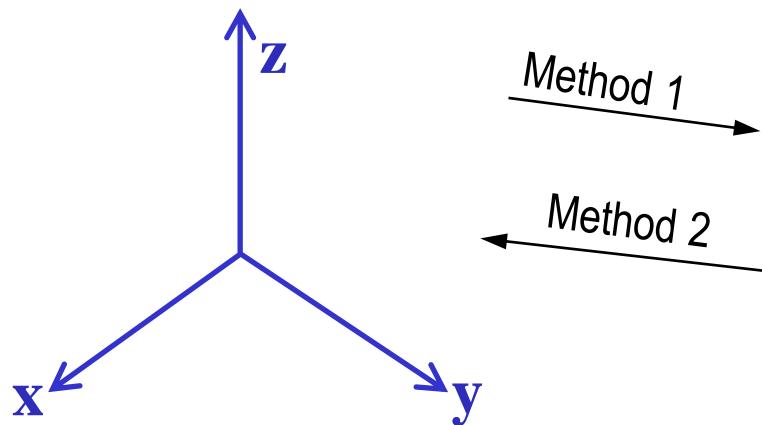
Method 1 Description:

This 4×4 matrix converts *vertex coord numbers from the ‘world’* coord. system **to** vertex numbers in *the ‘camera’* coord-system.

Method 2 Description:

4×4 matrix *moves* a copy of the *‘camera’ drawing axes* **to the** position & orientation where the camera finds the *‘world’ axes*

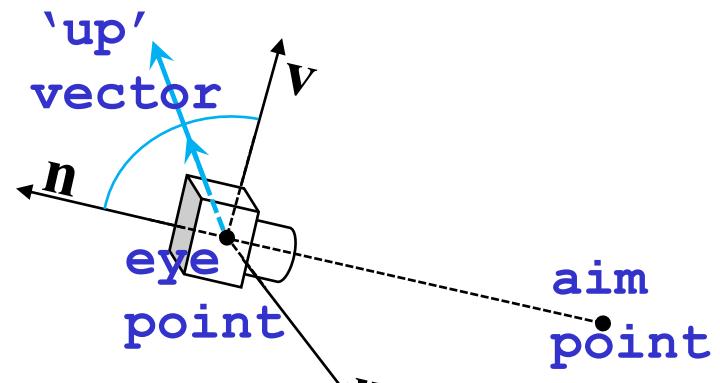
‘World’ Coord Axes



Method 1

Method 2

‘Camera’ Coord Axes



ASIDE: Duality for the Orthographic Projection Matrix

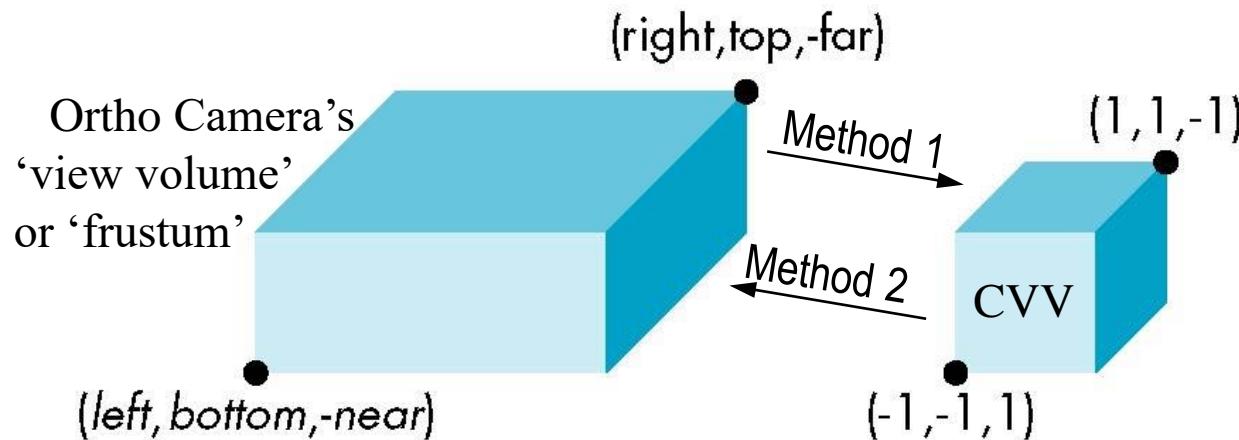
Method 1 Description:

This 4×4 matrix shifts & scales *vertex coord. numbers* in the rectangular view volume to squeeze them *into the CVV*

Method 2 Description:

4×4 matrix shifts & scales the *CVV volume & drawing axes* to new volume and axes that define *camera's 'view volume'*

ortho(left, right, bottom, top, near, far)



ASIDE: Duality for the Perspective Projection Matrix

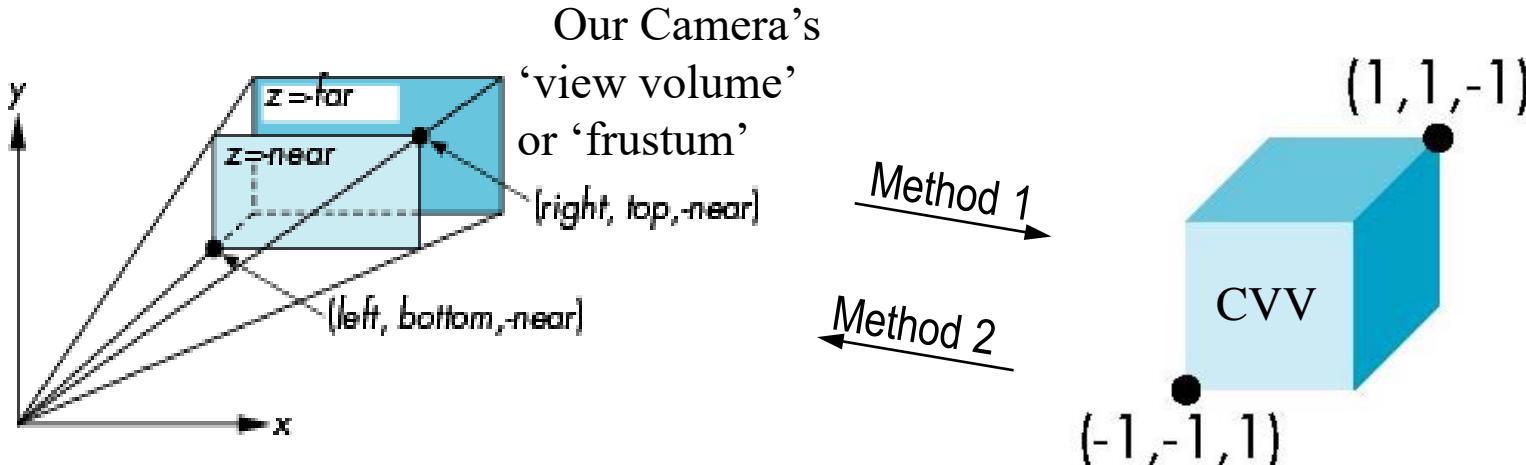
Method 1 Description:

This 4×4 matrix moves & warps the *vertex coord. numbers* in the *view frustum* (truncated 4-pyramid) *to fit them within the CVV*

Method 2 Description:

4×4 matrix moves & warps a copy of the *CVV volume & axes* *to new volume and axes that define the viewing frustum*

- **frustum(left, right, bottom, top, near, far)**
- **perspective(fovy, aspect, near, far);**



Challenge: Camera On Robot Arm

Many games use ‘3rd Person’ camera:

-----DEMO----- (unity game engine)

<https://youtu.be/iEm88-SkyUM?t=15s>

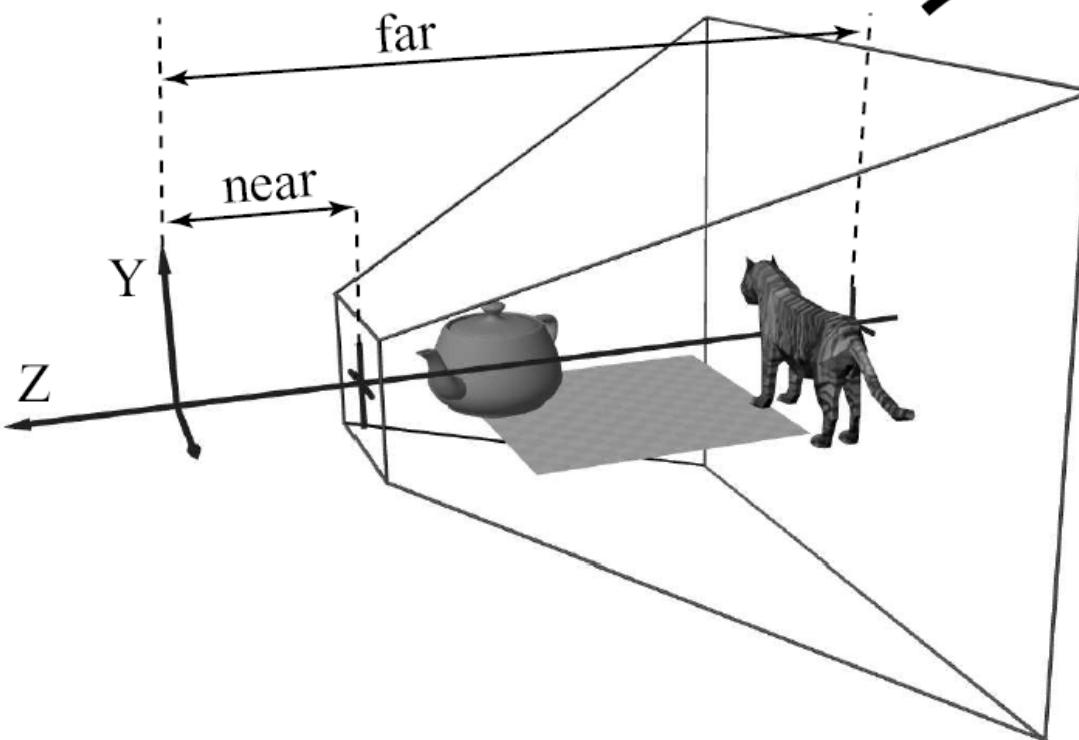
What ‘view’ matrix would attach a camera to the end of a moving robot arm?

END

END

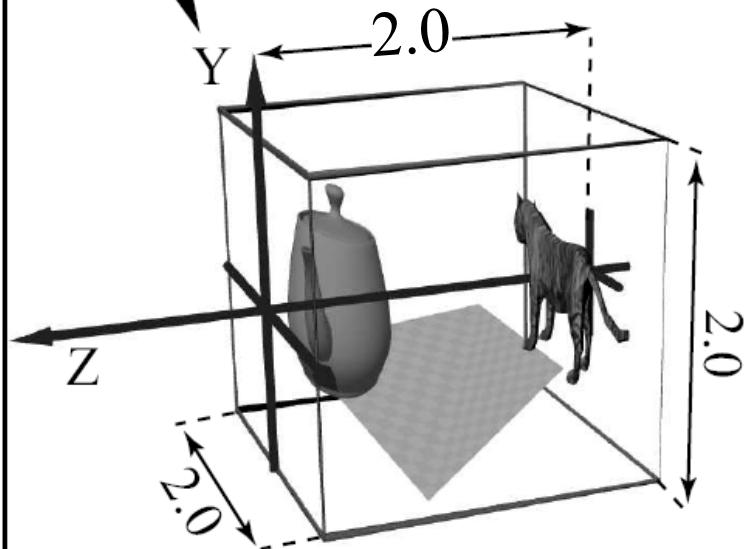
View Frustum to CVV Cube

EYE Volume (Frustum)



Projection Transform

CVV
Visible Volume



Peter Shirley, Chapter 14

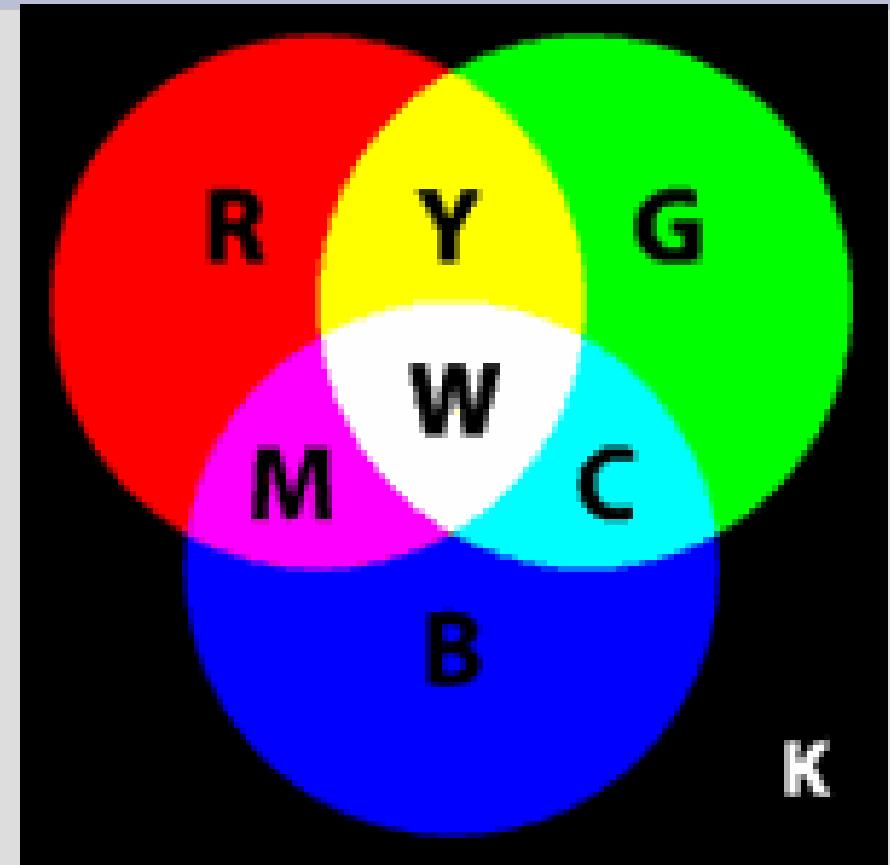
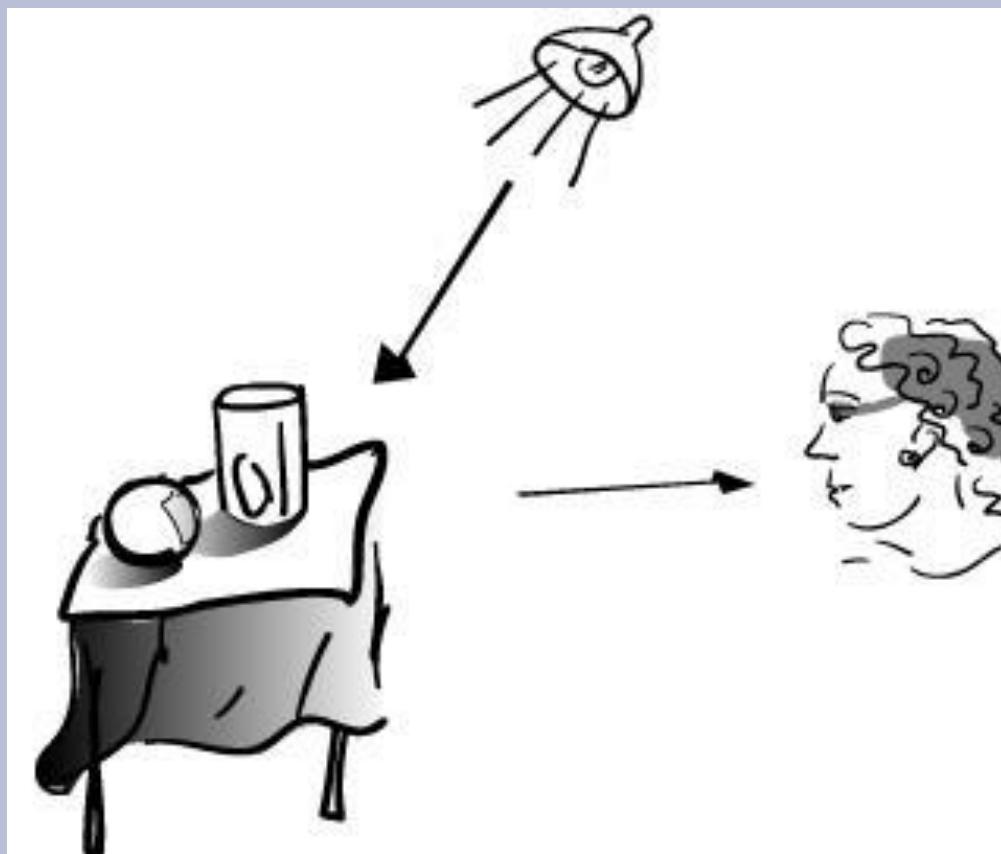
CAREFUL! SHOWS INCORRECT CVV AXES ON RIGHT! (SHOULD BE CENTERED IN CUBE)

INTRO to Lighting & Materials



Some slides modified from: David Kabala
Others from: Andries Van Damm, Brown Univ.

What is ‘Light’? What is ‘Color’?



READ ALL ABOUT IT HERE:

CANVAS → Project C: Lighting & Shading →

Project C Reading

→ [2021.11.illumination.Shading.pdf](#)

What is Light? Reflectance? ‘Color’?

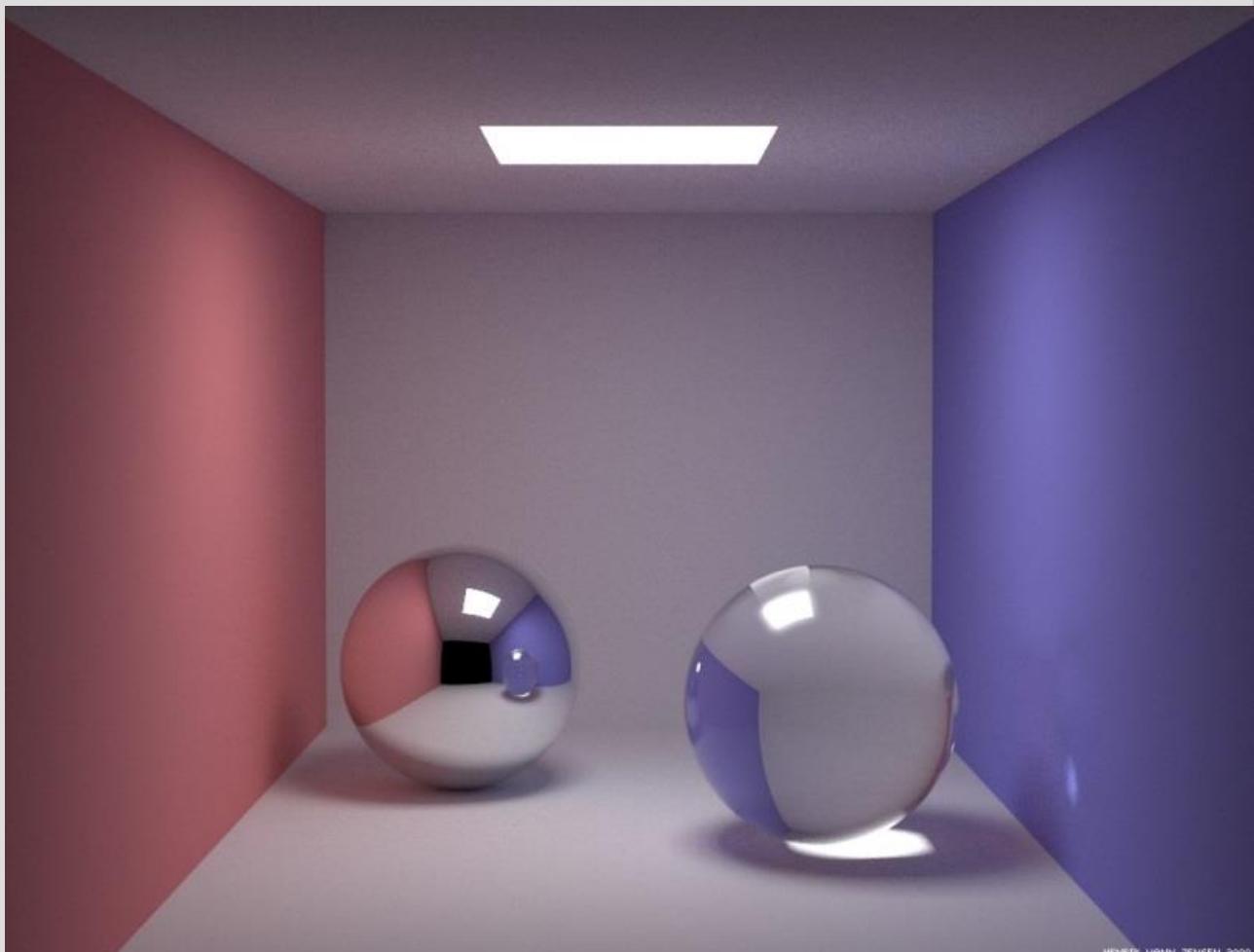
Simple Answers:

‘Visible Light’ ==
Radiative transfer of
electromagnetic energy
(at visible wavelengths)

‘Reflectance’ ==
Fraction of incident light
re-emitted from a surface.

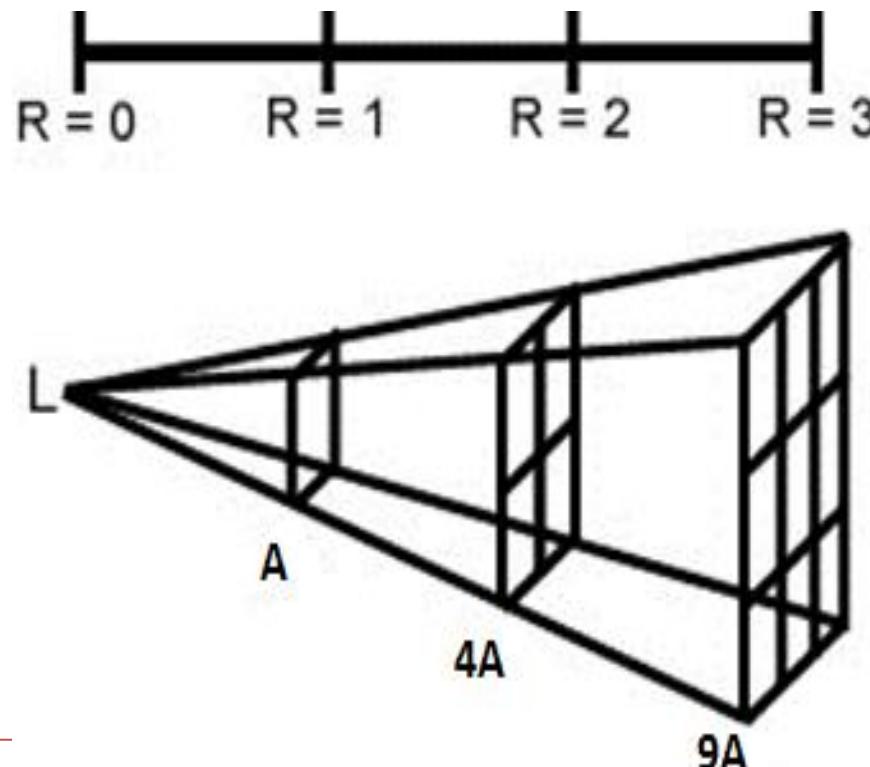
Often written as a
unit-less weighting
between 0 & 1.

‘Color’ == Subjective
HUMAN PERCEPTION
of light power and (https://en.wikipedia.org/wiki/Visible_spectrum)
its spectrum (between ~380nM and ~740nM)



Light Transport: Inverse Square Law

- ▶ Amount of light that illuminates a surface decreases with square of distance from a point-like light source. (a large lig source is a collection of points) Why? because:
- ▶ For a given 3D angle (solid angle), the covered **area grows by the square** of the distance.



Effect of Light Direction?

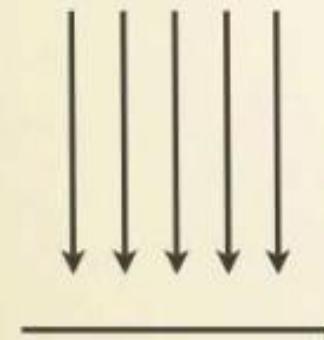
Simple Answers:

'Reflectance' == Fraction of visible light returned from a 3D scene

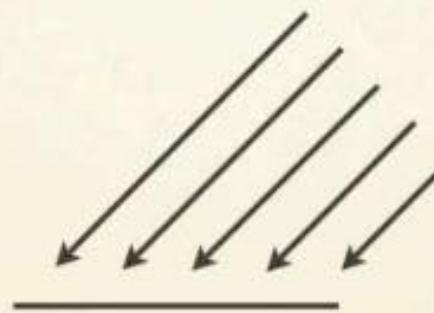
"Incident Light" (Irradiance)

has Cosine Weighting:

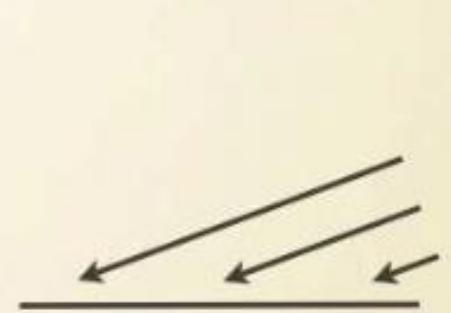
At shallow angles,
parallel rays of light spread over a wider surface area:



$$\cos(0^\circ) = 1$$



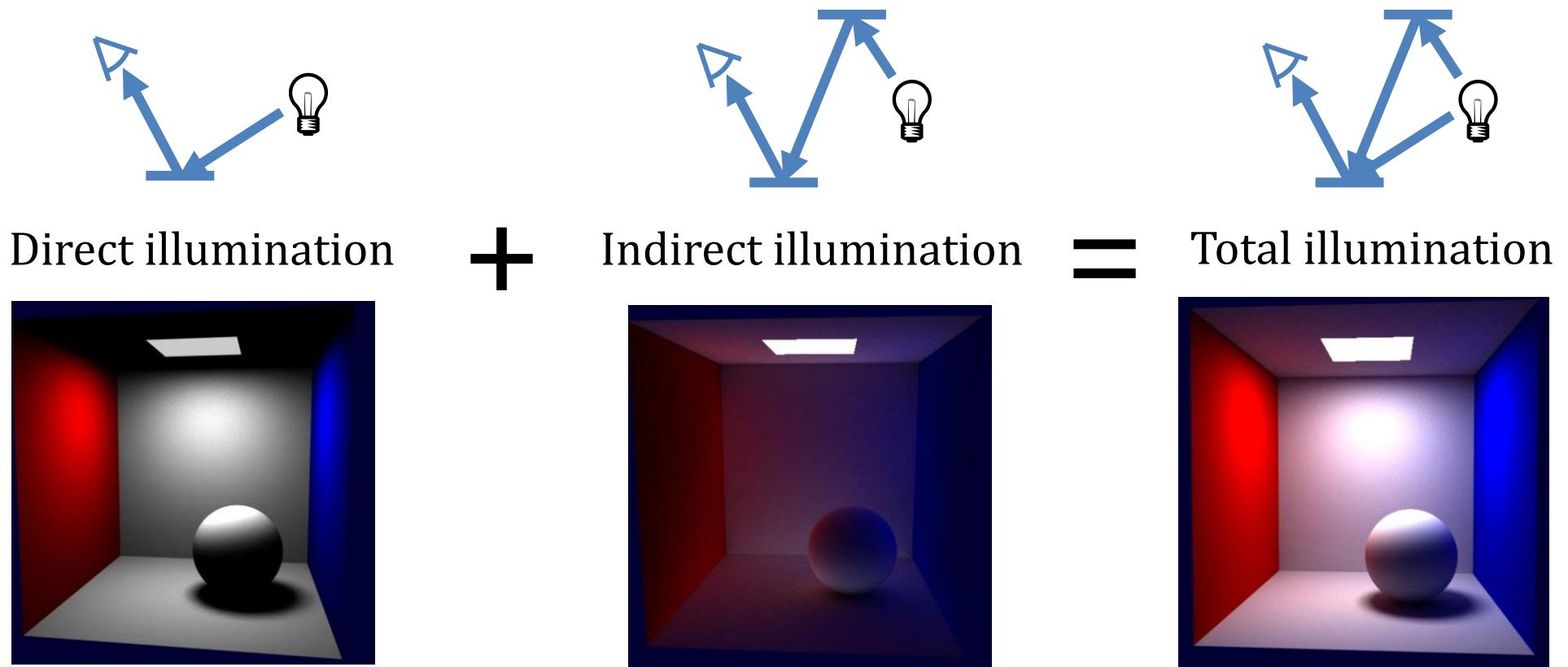
$$\cos(45^\circ) \approx .71$$



$$\cos(70^\circ) \approx .34$$

Modified slide from Mark Kilgard, nVidia

Global Illumination: (How light *really* works...)



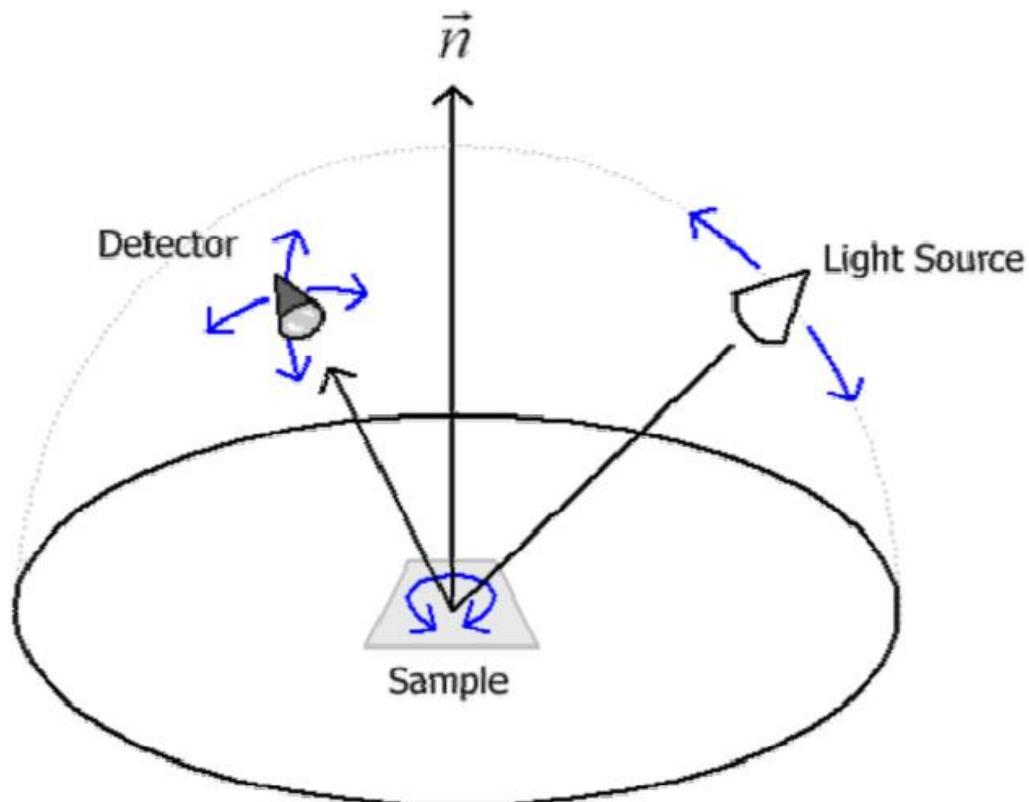
Light (in cd/m^2 , the visible part of ‘radiance’) is Linear! Additive! Proportional!

COOL TRICKS: add, multiply, subtract light efx on a 3D scene...

SEE: Graphics pioneer Paul Haeberli: <http://www.graficaobscura.com/synth/index.html>

Reflectance (scalar) .vs. ‘BRDF (4D) ?Is this a Complete Measurement?

- Gonioreflectometer is device to measure BRDFs



Diffuse Interreflection: **TOTAL** = DIRECT + INDIRECT
(normal image)
from point-like source



Diffuse Interreflection: TOTAL = DIRECT + INDIRECT
No ‘bounces’!



Diffuse Interreflection: TOTAL = DIRECT + INDIRECT

ONLY bounced light!

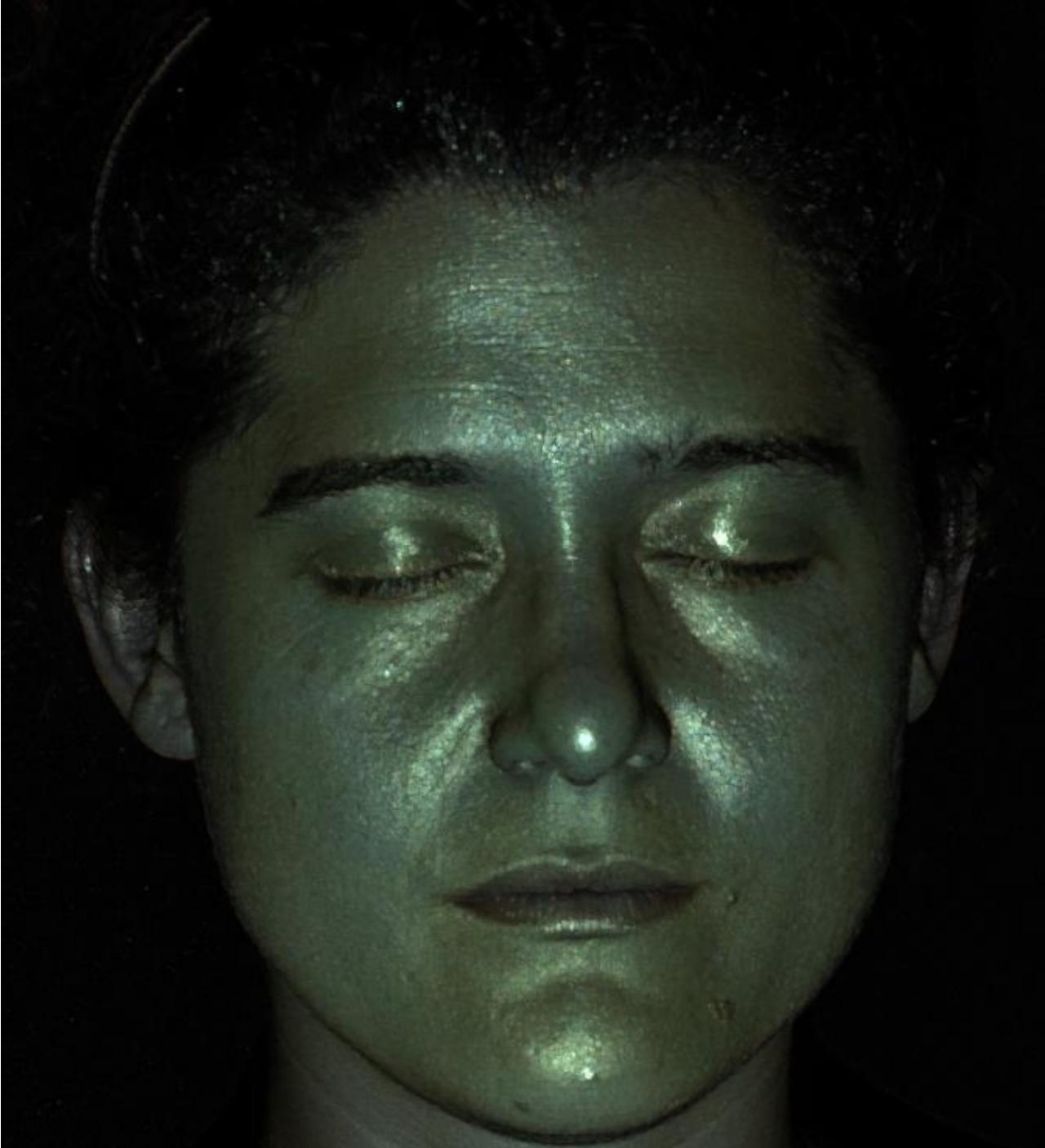


Human face



Total illumination
(normal image)

Human face



DIRECT illumination
no 'bounced', scattered light →
oily, almost-colorless surface

Human face



INDIRECT illumination
ONLY bounced/scattered light:
→
Translucency & scattering in
skin layers (cuticle), nerves,
blood-filled capillaries,
subcutaneous muscles & fat

COOL!

How did they separate direct and indirect light amounts?

Video projector illuminates scene with a few sharp-edged black/white patterns:

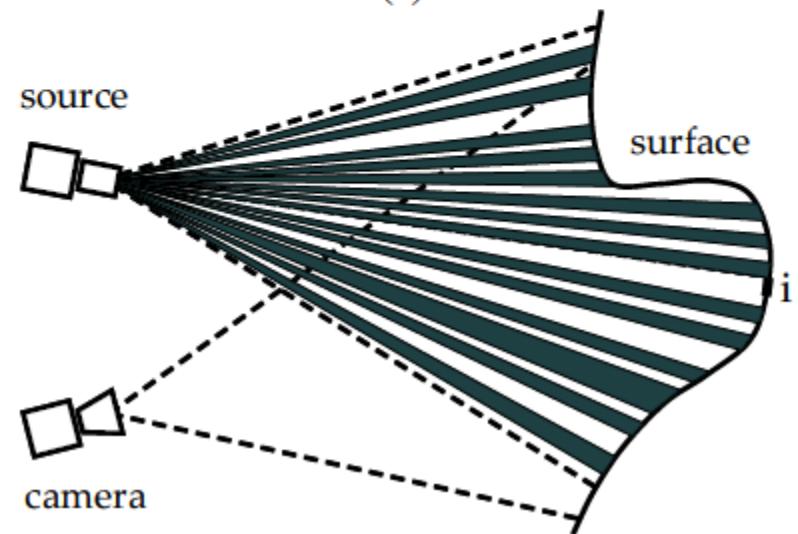
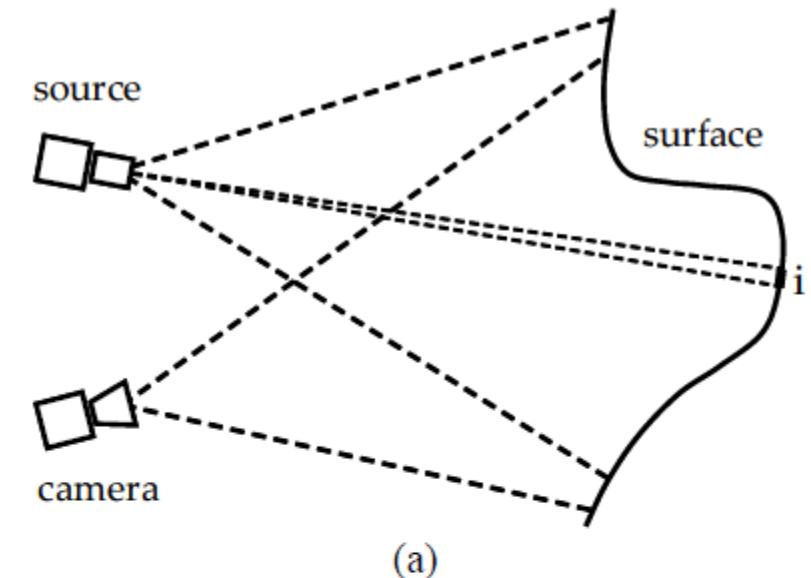
For example,

- ▶ a black/white checkerboard pattern (interleaved grid of lit and unlit patches)
- ▶ the inverse checkerboard (white/black)

Take photos with each illumination pattern.

- ▶ Each 'unlit' patch returns only the indirect illumination result.
- ▶ Each 'lit' patch returns (direct + indirect)

THEN for each pixel: total == lit patch value;
indirect== unlit patch value;
direct == (lit patch) - (unlit patch)



Vital Definitions (1): Lighting and Shading

- ▶ **Lighting**, (or *illumination*) === process to find light color and intensity.
- ▶ We compute light that **arrives** at and **leaves** from **a point** on a surface in a scene.
Some of that arriving light gets **reflected** towards your eye...
- ▶ ***lighting results depend on:***
 - ▶ The **light source(s)** strengths and positions,
 - ▶ The **camera** position, (surprise! Camera view direction doesn't matter!)
 - ▶ the **viewed surface** orientation (or '**normal' vector**)
 - ▶ the **viewed surface** material properties,

Vital Definitions(2): Lighting and Shading

- ▶ A **vertex** describes just one point on a surface, (position, surface normal, material, etc.) but GPU draws triangles!
- ▶ **Shading** == *find the color values for all pixels between vertices.*
== old graphics hardware interpolated colors only, but
== GLSL now lets us do *lighting* calculations at each pixel!
- ▶ On the GPU systems we use with WebGL,
the **vertex shader** usually computes **lighting** for each vertex,
the **fragment shader** usually computes **shading** for each pixel
- ▶ **Fast Shading hardware is crucial** to real time graphics(e.g., games) because lighting calculations are often complex and expensive.
- ▶ **An old, slow but great alternative:** ray-tracing computes lighting from the paths of photons through the 3D scene(!). Ray-traced images can include recursive inter-reflected light, and fine 'sub-pixel' sampling for antialiasing, THUS in Ray-tracing 'shading' is much more accurate, but much slower.

Phong Lighting:

Step 1: Find the Scene Vectors

To find ***On-Screen RGB Color*** at point **P (start)**:



1) Find all 3D scene vectors first:

a) Light Vector **L**:

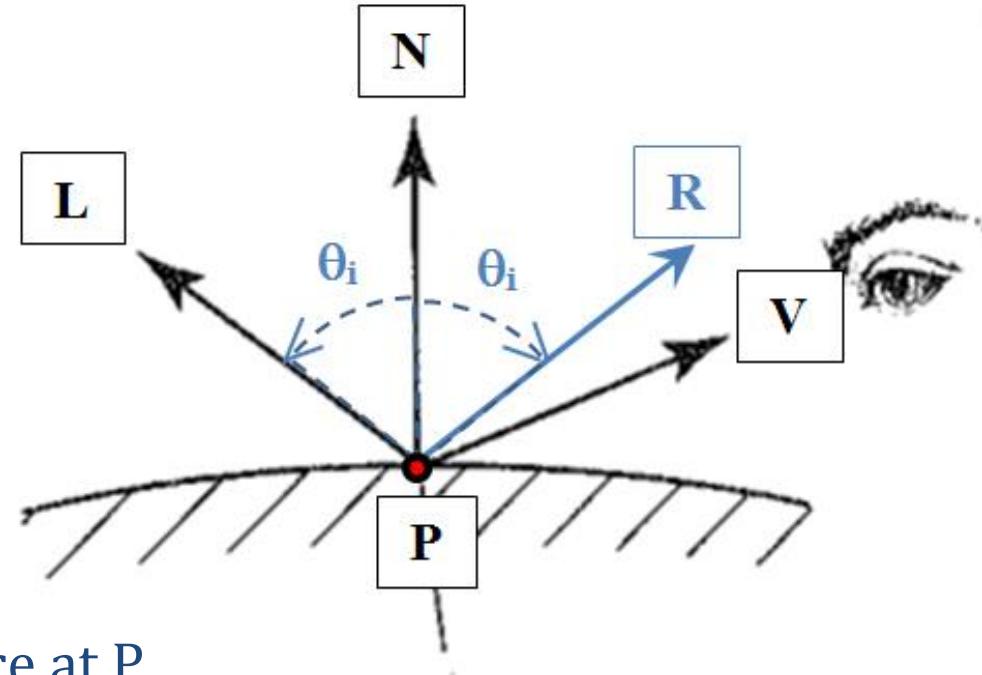
unit vector towards light source

b) Normal Vector **N**:

unit vector perpendicular to surface at P

c) View Vector **V**:

unit vector towards camera eye-point



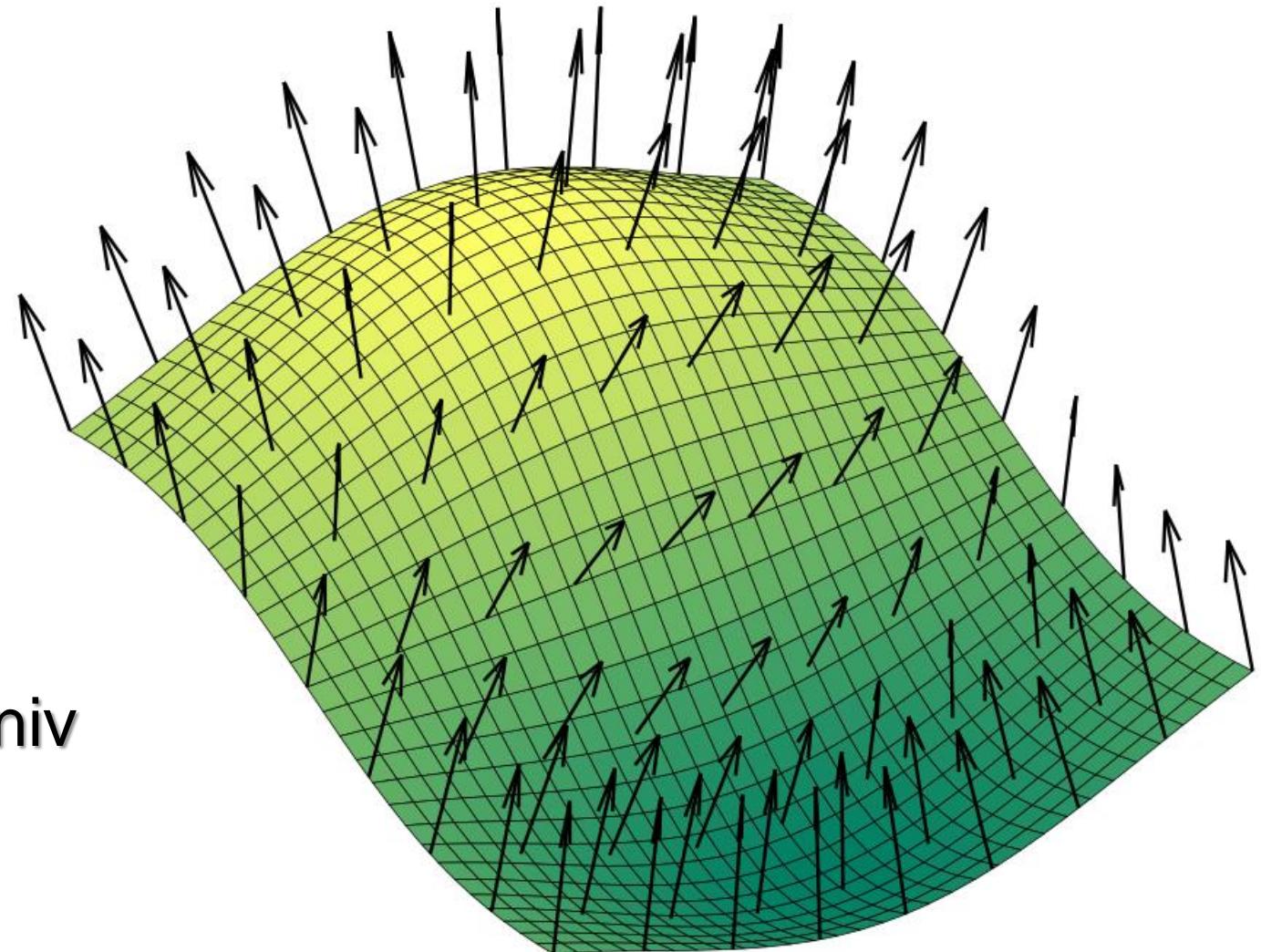
Can you do step 2? how do we find the Reflected-light Vector **R**?

More about 'normal vector' next class...

!END!

How To Build and Use Surface Normals in WebGL

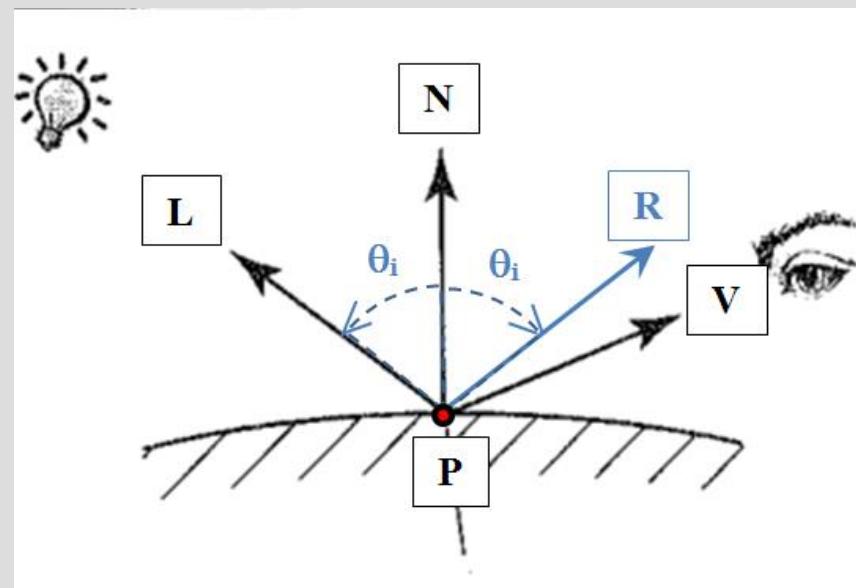
Jack Tumblin
Northwestern Univ
Comp Sci 351-1
Fall 2021



Why We Need Normal **N** At Every Vertex:

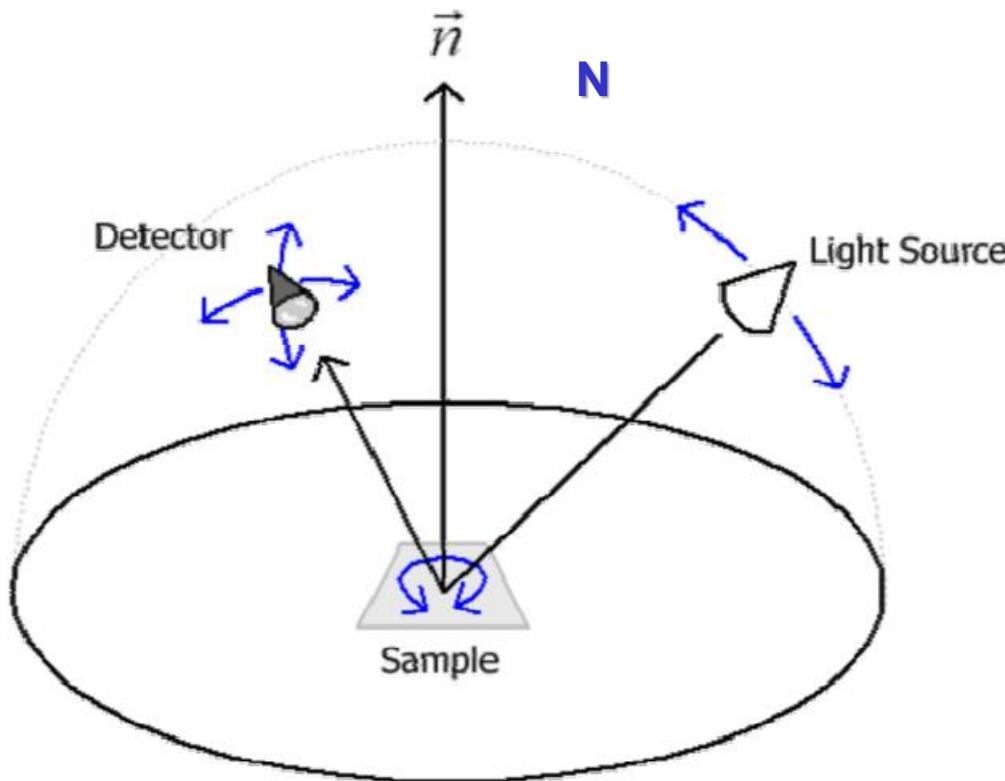
<http://math.hws.edu/graphicsbook/source/webgl/cube-camera.html>

How else could we know the ‘reflected’ direction for each vertex of a mirrored surface?



WHY do we need **N**? Because **Reflectance** set by **ANGLES** from **N**:

- Gonioreflectometer is device to measure BRDFs Bi-directional Reflectance Distribution Function (4D!)

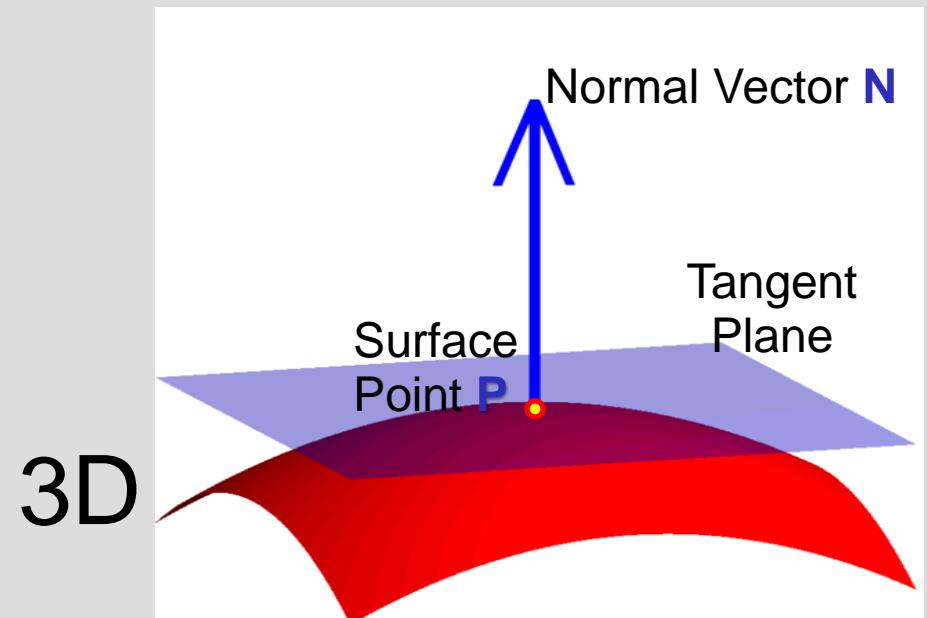
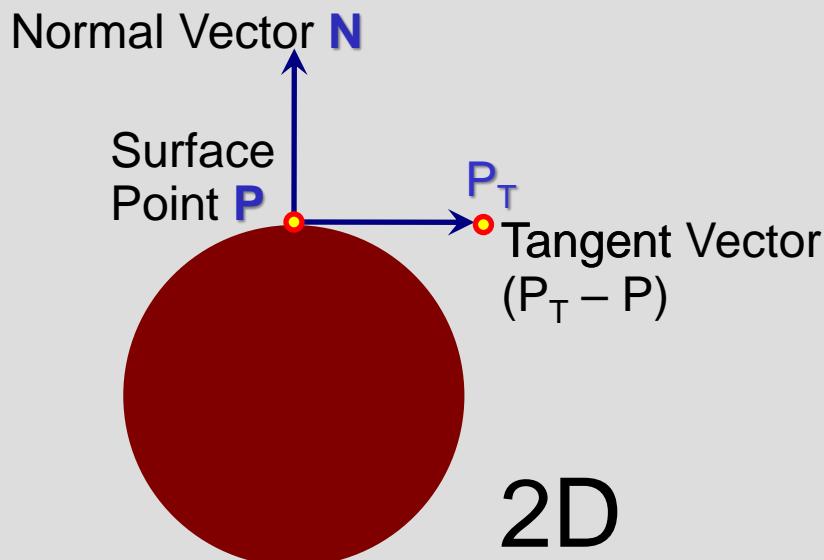


Define: “Normal” Vector $\mathbf{N} ==$ Local Surface Orientation

\mathbf{N} = Perpendicular Vector at any Surface Point

More formally:

- Unit-length vector \mathbf{N} at surface point \mathbf{P}
- Vector \mathbf{N} also defines tangent plane at \mathbf{P} :
(For all points \mathbf{P}_T in tangent plane, $(\mathbf{P}_T - \mathbf{P}) \cdot \mathbf{N} = 0$)



OK; so we need ‘normal’ attribute..

‘Unit-Length Surface Normal Vector **Attribute**’

Usually just called ‘**the normal**’ (vector, direction only)

How do I compute it?

- For a triangle:
cross-product of edge vectors
- For a mesh node:
good: avg. of adjacent triangle normals
better: area-weighted avg. of normals
- For ideal shape: (e.g. sphere, cylinder, torus...)
best: find **⊥** direction by trigonometry

2D Lines: A New, Homogeneous Form for you...

We can write any **2D Line** as the set of points (x,y,w) that satisfy this homogeneous equation:

$$Ax + By + Cw = 0 \quad (\text{not a function } f(x) = y; \text{ an equation!})$$

Where:

- each parameter A, B and C is real & scalar, AND
- Either A or B is nonzero, $(A=B=0; C\neq 0? \rightarrow \text{no solutions possible for } (x,y)!$
 $(A=B=0; C=0? \rightarrow \text{all } (x,y) \text{ are solutions.})$
- **2D Line** params (A,B,C) are ‘homogeneous’;
scaling has no effect: (A,B,C) & $(7A,7B,7C)$
describe the same line \rightarrow THUS
- Line crosses X axis at $x,y = (-C/A, 0)$
- Line crosses Y axis at $x,y = (0, -C/B)$
- Perpendicular 2D line? (I want YOU to figure it out...)

3D Planes? Similarly, here's a *Homogeneous Form*

We can write any **3D Plane** as the set of points (x,y,z,w) that satisfy this homogeneous equation:

$$Ax + By + Cz + Dw = 0$$

(*not* a function $f(x) = y$; this is an **equation!**)

Where:

- each parameter **A**, **B**, **C** and **D** is real & scalar, AND
- Either **A**, **B** or **C** is nonzero, ($A=B=C=0; D \neq 0 \rightarrow$ *no* (x,y) are solutions;
 $A=B=C=0; D=0 \rightarrow$ *all* (x,y) are solutions.)
- **3D Plane** params (A,B,C,D) are ‘**homogeneous**’;
scaling has no effect: (A,B,C,D) & $(7A,7B,7C,7D)$
- Plane’s surface normal vec: $\mathbf{n} = (A,B,C) / \|(A,B,C)\|$
- Unit surface-normal vector **n** is \perp to plane, and
- For any point **P** = (px,py,pz) on plane, **P · n = -D**

? Surface Normals for TRIANGLES ?

- For one flat isolated triangle:
find normal vector perpendicular to
the triangle's 3D plane

And in WebGL,

‘**the Unit-Length Surface Normal Vector Attribute**’
is usually just called ‘**the normal**’ (vector direction)

A single isolated flat triangle? Easy!
same **normal** for all 3 vertices.

? Surface Normals for TRIANGLES ?

‘the Unit-Length Surface Normal Vector Attribute’
is usually just called ‘the normal’ (vector direction)

How do I compute it?

- For a triangle:
cross-product of edge vectors
- For a mesh node:
good: *avg. of adjacent triangle normals*
better: *area-weighted avg. of normals*
- For ideal shape: (e.g. sphere, cylinder, torus...)
best: find \perp direction by **trigonometry**

REVIEW:

*Vector **Dot** Product: ($\mathbf{u} \cdot \mathbf{v}$)*

$$\mathbf{u} \cdot \mathbf{v} = x_u x_v + y_u y_v + z_u z_v$$

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}| |\mathbf{v}| \cos(\theta)$$

THUS: $\cos\theta = (\mathbf{u} \cdot \mathbf{v}) / (|\mathbf{u}| |\mathbf{v}|)$

- *Purely a **scalar** result -- not a vector!*
- *What happens when the vectors are unit-length?*
- *What does it mean if dot product == 0 or == 1?*

REVIEW:

*Vector **Cross** Product: $(\mathbf{u} \times \mathbf{v})$*

- *The result is a vector, not a scalar, **perpendicular** to the plane formed by the two crossed vectors*
- *Can be computed using the determinant of:*

$$\begin{vmatrix} i & j & k \\ x_v & y_v & z_v \\ x_u & y_u & z_u \end{vmatrix}$$

$$\mathbf{u} \times \mathbf{v} = (y_u z_v - z_u y_v) \mathbf{i} + (z_u x_v - x_u z_v) \mathbf{j} + (x_u y_v - y_u x_v) \mathbf{k}$$

- *Vector magnitude: $\|\mathbf{u} \times \mathbf{v}\| = \|\mathbf{u}\| \cdot \|\mathbf{v}\| \cdot \sin(\theta)$*
- *Self-Cross? Always zero! $\mathbf{u} \times \mathbf{u} = 0$*

Effect of Light Direction?

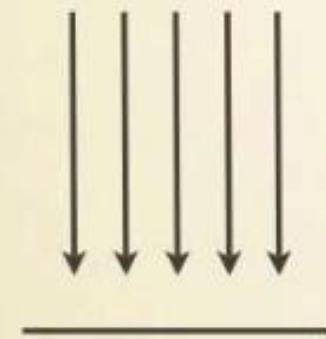
Simple Answers:

'Reflectance' == Fraction of visible light returned from a 3D scene

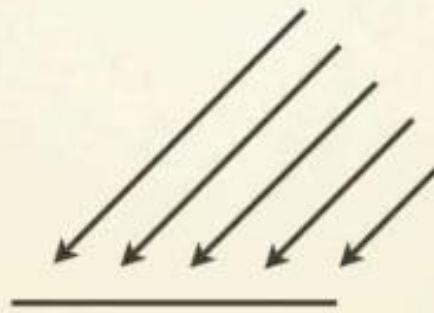
"Incident Light" (Irradiance)

has Cosine Weighting:

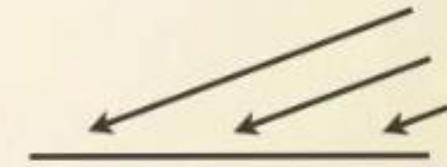
At shallower angles,
parallel rays of light spread over a wider surface area:



$$\cos(0^\circ) = 1$$



$$\cos(45^\circ) \approx .71$$



$$\cos(70^\circ) \approx .34$$

Lambertian Material: color varies by $\cos(\theta)$ falloff only

- Just one ‘Material’ parameter (with several names):
 K_{diffuse} or ‘albedo’ or ‘diffuse reflectance’ or ...
== returned fraction of incident light intensity I_{diffuse}
== $0 \leq R, G, B \leq 1$.
- Viewed color depends ONLY on illumination direction:
find $(\mathbf{N} \cdot \mathbf{L})$ to impose $\cos(\theta)$ falloff
- Color is same for all viewing directions \forall
- ‘Diffuse Lighting’ defined by:

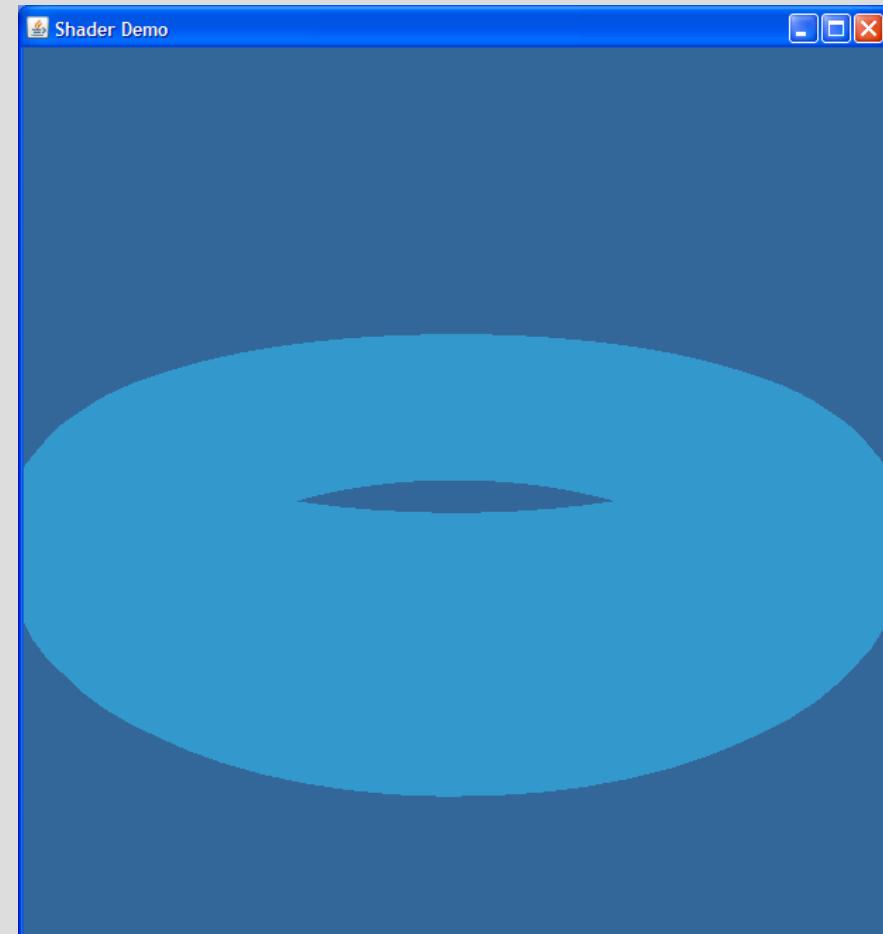
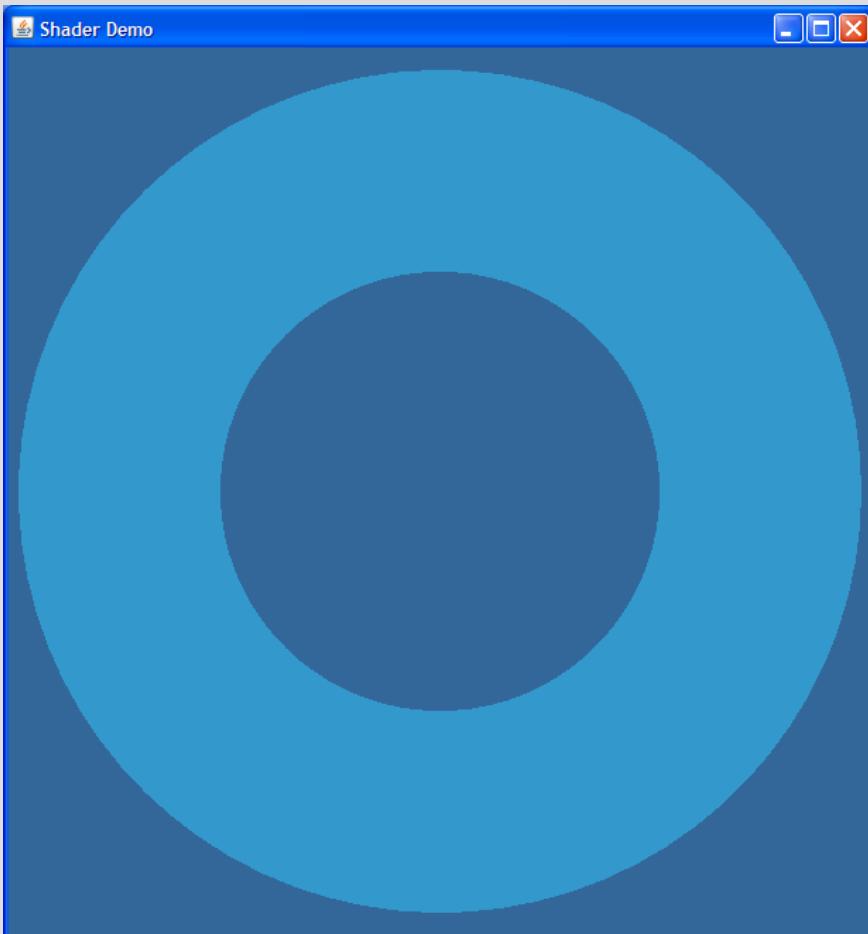
$$\text{On-Screen}[r,g,b] = K_{\text{diffuse}}[r,g,b] * I_{\text{diffuse}}[r,g,b] * \text{dotProduct}(\mathbf{N}, \mathbf{L});$$

\mathbf{N} == surface normal vector

\mathbf{L} == light vector (from surface point to light source)

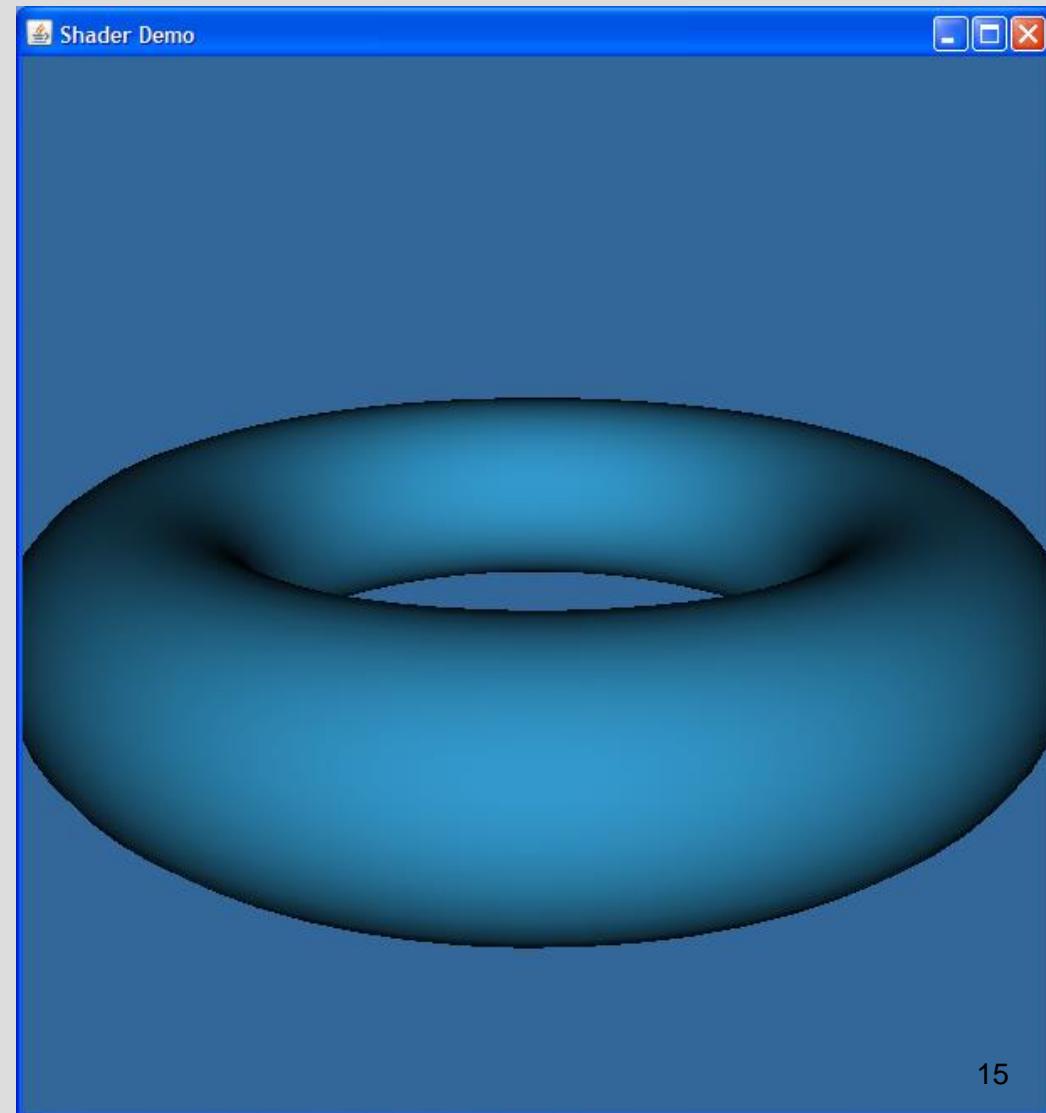
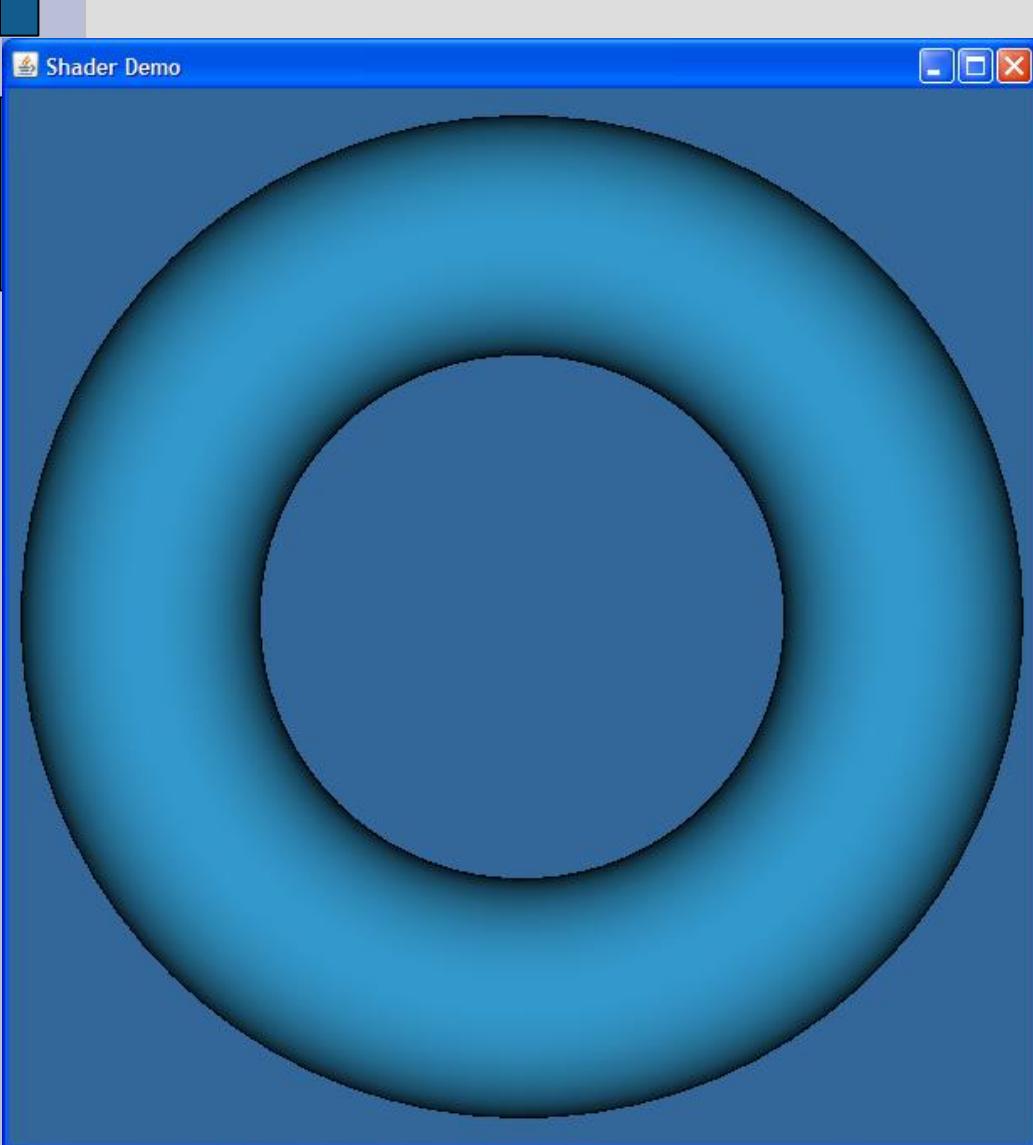
‘Ambient’ Lighting: (mimics Proj A, B)

- equal light **from** all directions (ambient illum),
- and equal light reflected **to** all viewing directions



“diffuse lighting”: $\mathbf{N} \cdot \mathbf{L}$

Shown: Point-Light Source (\mathbf{L}) at Eye-point.



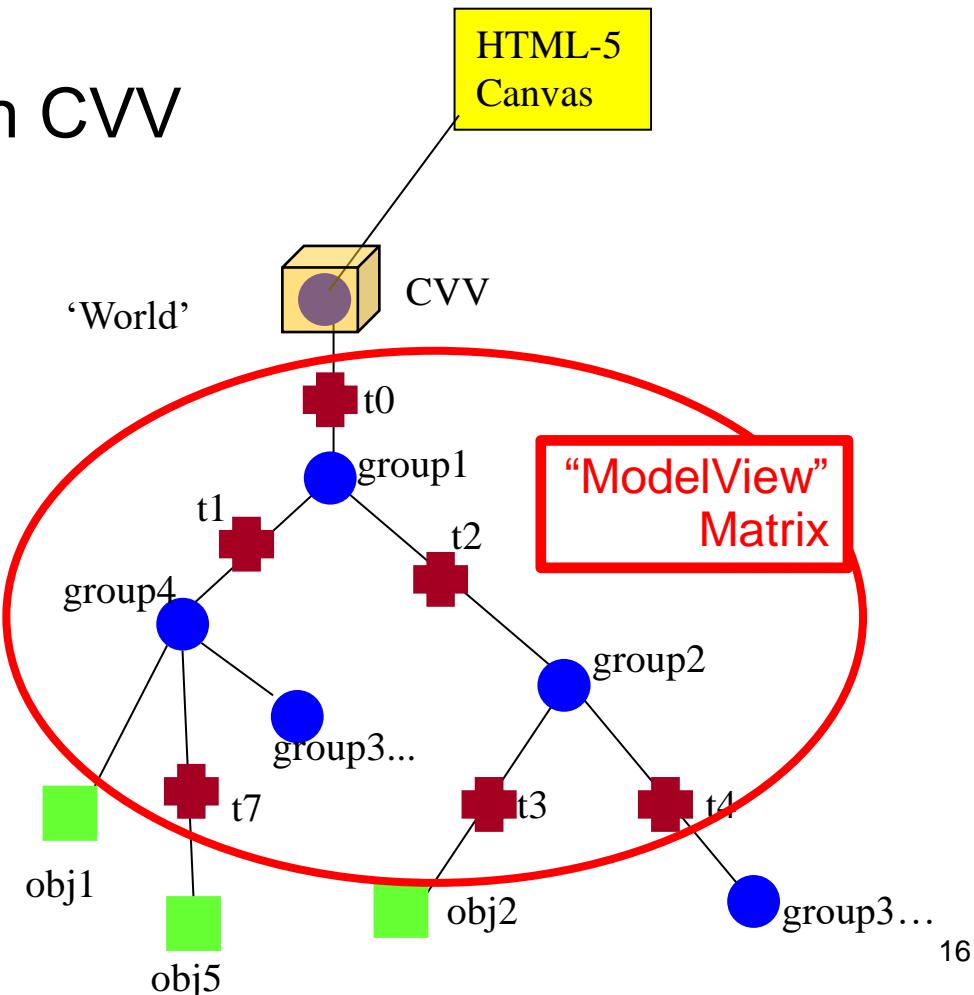
WebGL: Vertex Position Pipeline

RECALL: in scene graphs,
Vertices & values move upwards,
transform calls move downwards

Project A: Draw ‘world’ directly in CVV

? Where do ‘normal’ vectors N
Fit into the Scene Graph?

? HOW do we transform
The ‘normal’ vectors N?

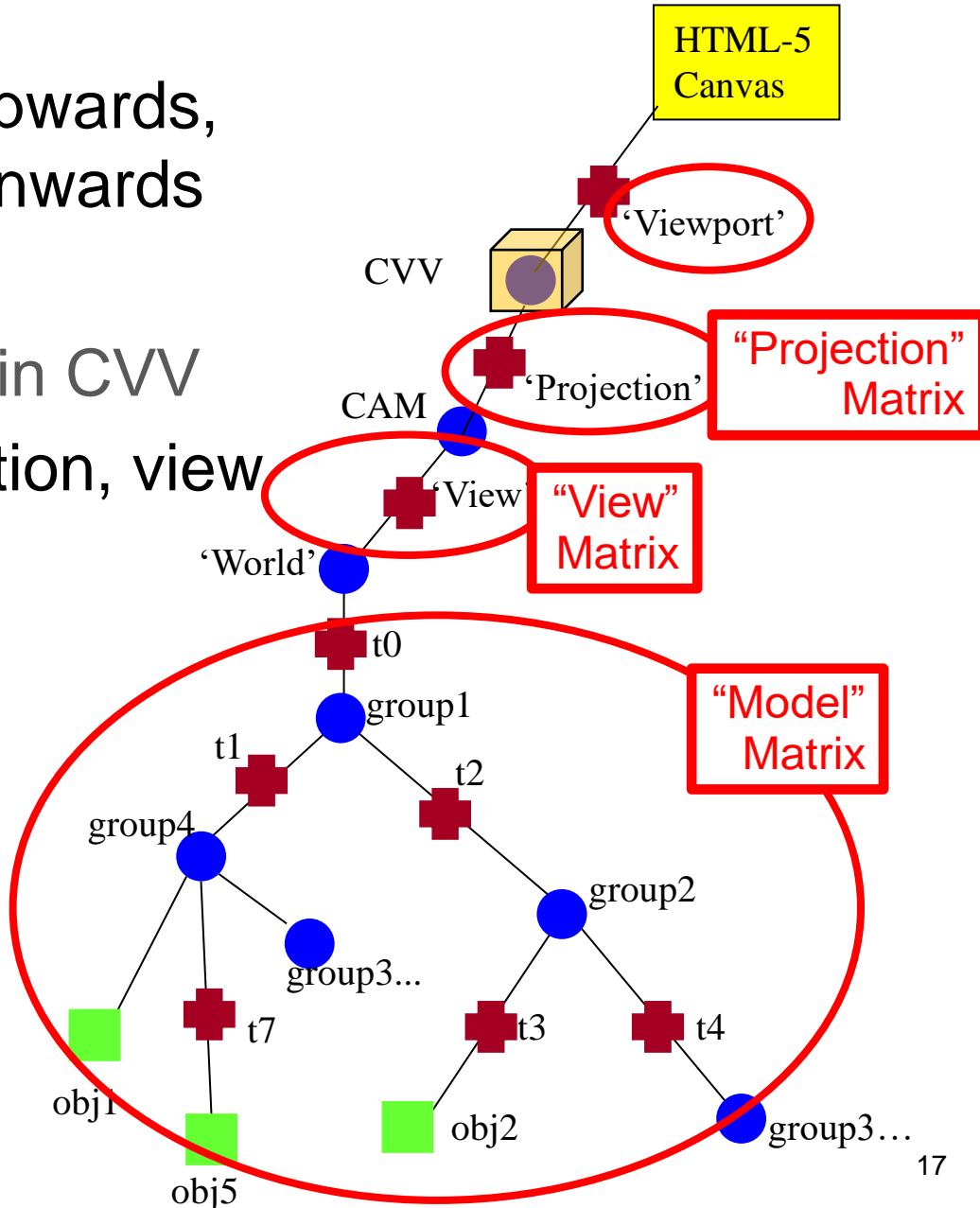


WebGL: Vertex Position Pipeline

RECALL: in scene graphs,
Vertices & values move upwards,
transform calls move downwards

Project A: Draw ‘world’ directly in CVV

Project B: Add viewport, projection, view



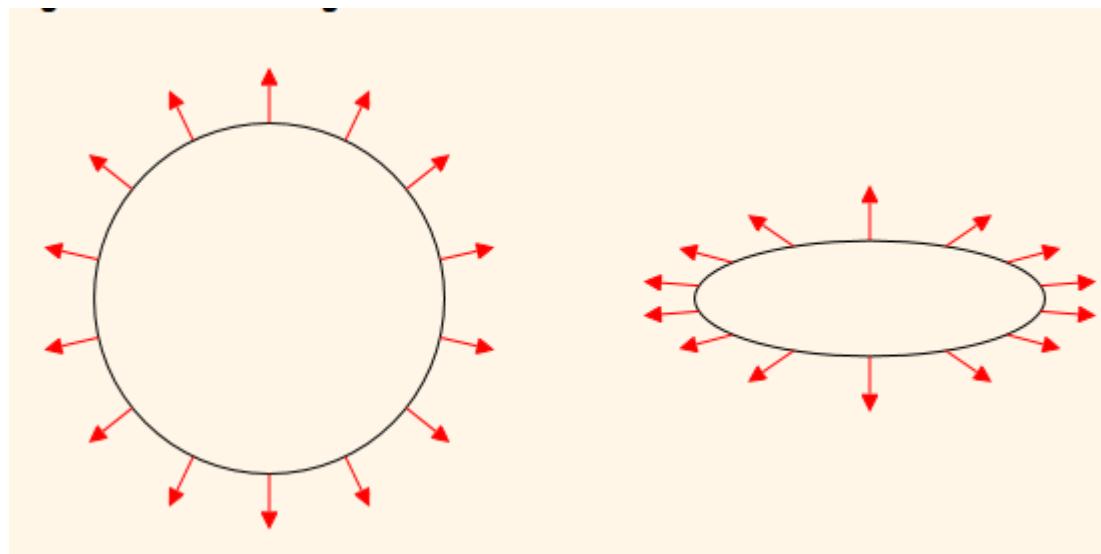
Transforming Normals

We KNOW how to transform vertex positions.

*Can we transform **Normal Vectors** with the same matrix?*

ALMOST
but not always!

non-uniform scaling? (stretched robot arm, etc)



Transforming Normals

We *KNOW* how to transform vertex positions.

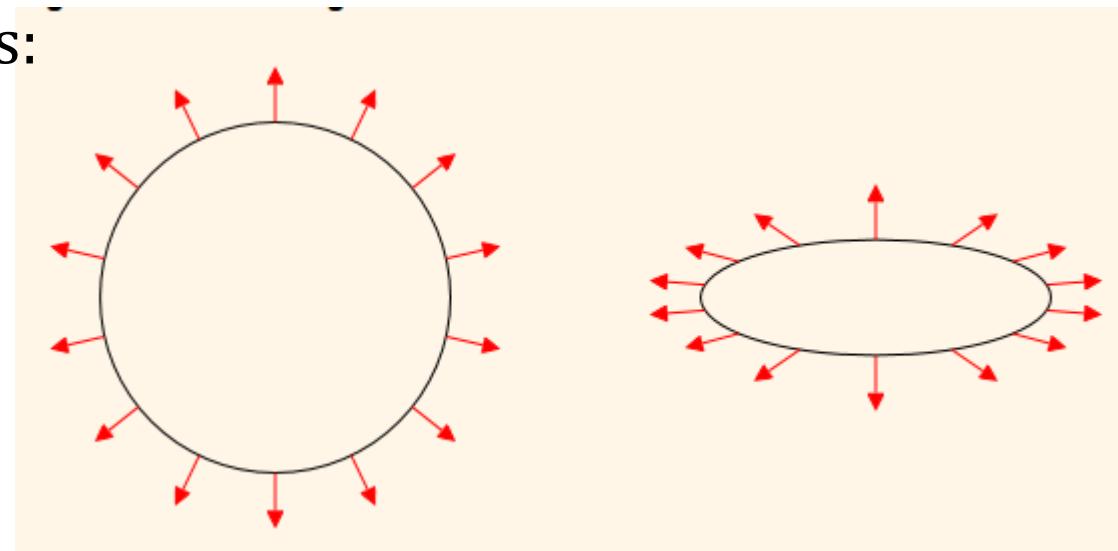
Can we transform Normal Vectors with the same matrix?

ALMOST always yes,

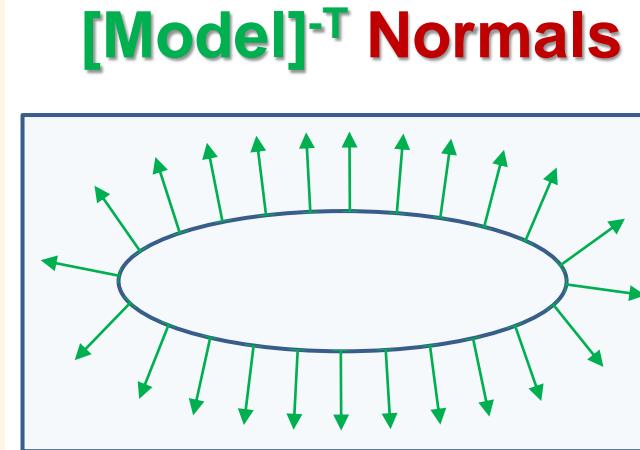
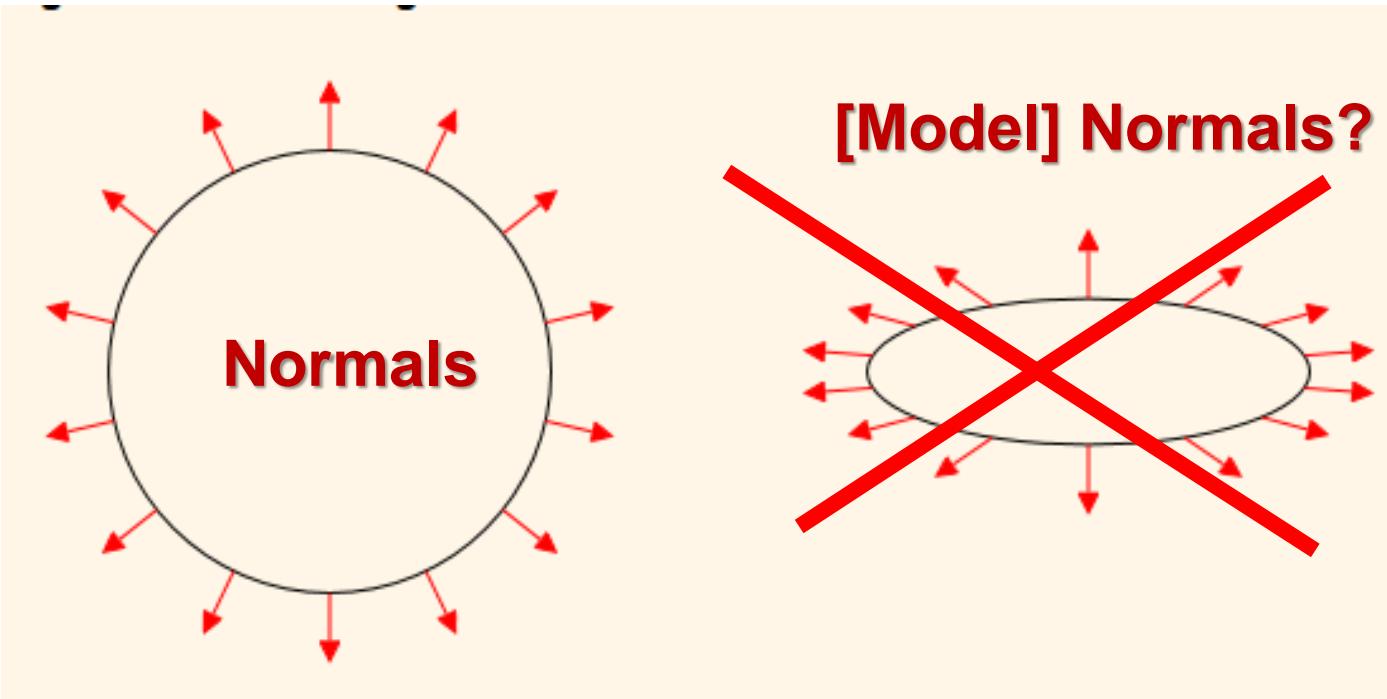
*but not always: → thus the answer is **NO.***

we need a special ‘**normal** transform **matrix**’

because non-uniform scaling of shapes (stretched robot arm, etc) distorts these normals:



Transforming Normals



SOLUTION: use *inverse-transpose*:

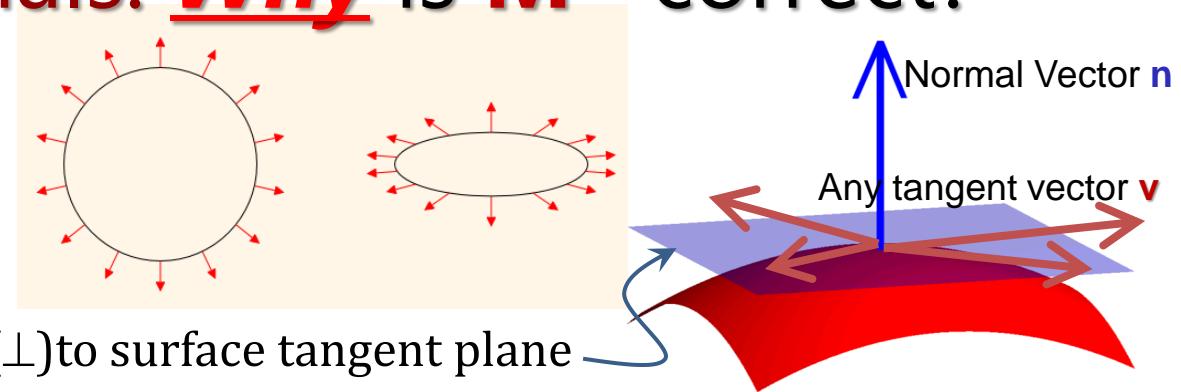
Normal Matrix == (Model Matrix)^{-T}

Why? see: <http://www.lighthouse3d.com/tutorials/glsl-12-tutorial/the-normal-matrix/>

or Canvas: Project C Reading: "How To Transform Normals"

How? [cuon-matrix-quat03.js](#) functions ...

Transforming Normals: Why is \mathbf{M}^{-T} correct?



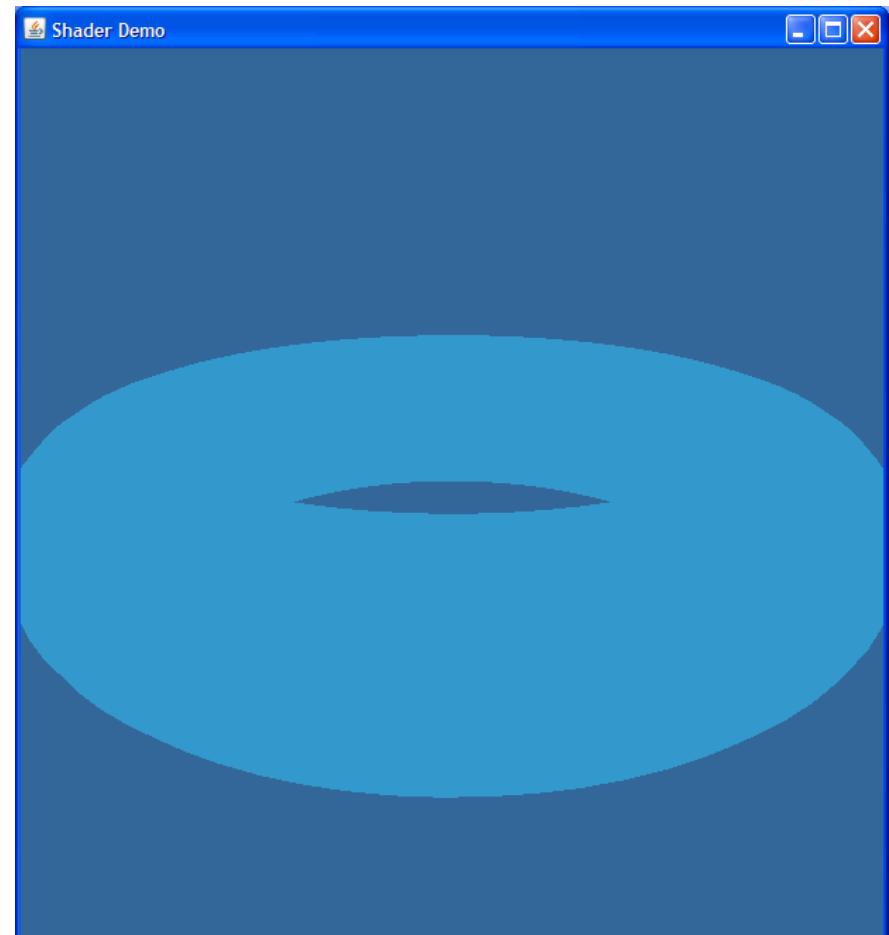
- ▶ normal vector == perpendicular (\perp) to surface tangent plane
- ▶ Any transform matrix \mathbf{M} applied to the surface applies to the tangent plane too.
- ▶ Any vector \mathbf{v} in the tangent plane is \perp to \mathbf{n} , thus $\mathbf{n} \cdot \mathbf{v} = \mathbf{0}$, or equivalently: $\mathbf{n}^T \mathbf{v} = \mathbf{0}$
- ▶ REVIEW:
 - ▶ For any non-singular matrix \mathbf{M} we can find an inverse \mathbf{M}^{-1} that cancels it: $\mathbf{M}^{-1} \mathbf{M} = \mathbf{I}$
 - ▶ Transpose lets us multiply column vector \mathbf{v} and matrix \mathbf{A} in either order: $\mathbf{A}\mathbf{v} = \mathbf{v}^T \mathbf{A}^T$
- ▶ Expand $\mathbf{n}^T \mathbf{v} = \mathbf{0}$ with the 'do-nothing' identity matrix: $\mathbf{n}^T (\mathbf{M}^{-1} \mathbf{M}) \mathbf{v} = \mathbf{0}$
- ▶ Associate each matrix with its neighbor: $(\mathbf{n}^T \mathbf{M}^{-1})(\mathbf{M}\mathbf{v}) = \mathbf{0}$ and then look closely:
 - ▶ $(\mathbf{M}\mathbf{v})$ == Any and all transformed tangent-plane vectors
 - ▶ $(\mathbf{n}^T \mathbf{M}^{-1})$ == The transformed normal vector *guaranteed* \perp to all the tangent-plane vectors
- ▶ Rearrange **transformed normal vector** using transpose: $(\mathbf{n}^T \mathbf{M}^{-1}) = (\mathbf{M}^{-1})^T \mathbf{n} = \boxed{\mathbf{M}^{-T} \mathbf{n}}$

!END!



The University of New Mexico

EX 1 – ambient lighting





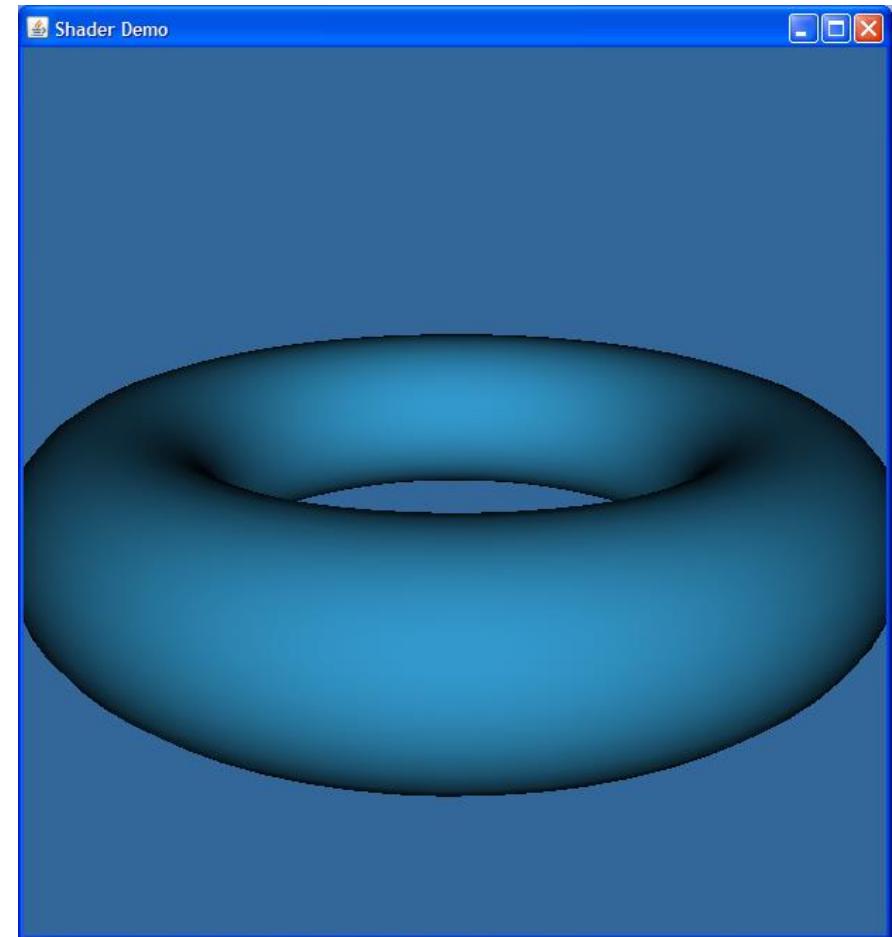
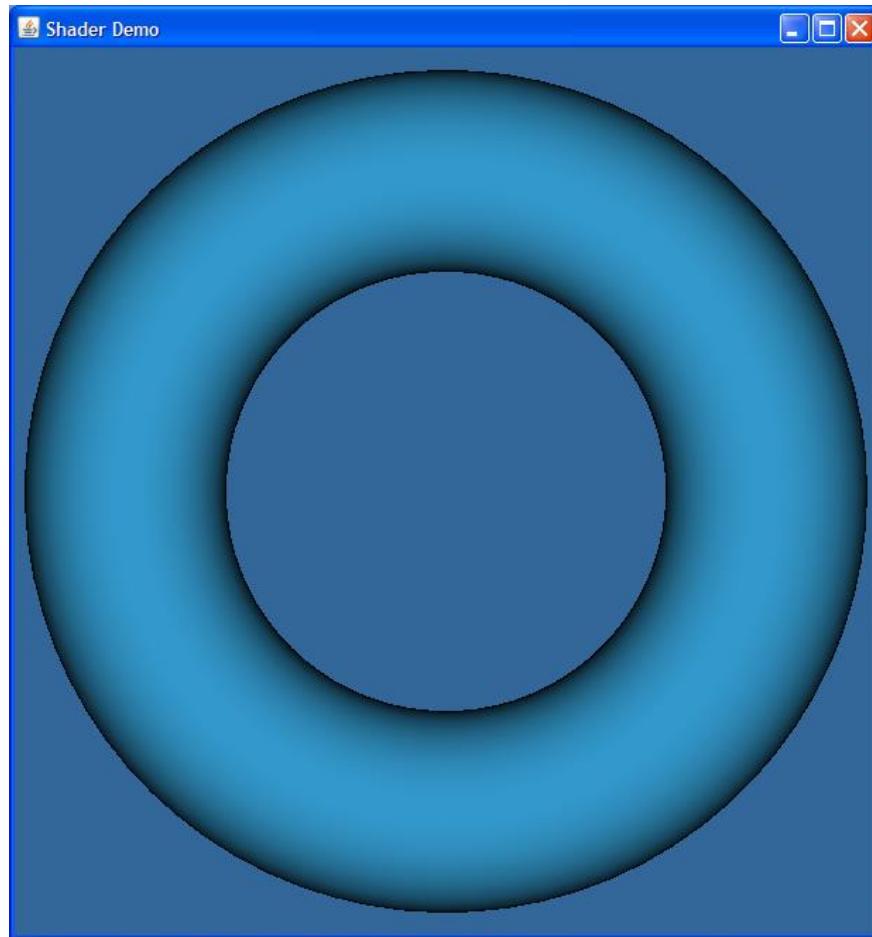
The University of New Mexico

EX 1– ambient lighting

```
// Vertex Shader
attribute highp vec4 a_vertPos;
uniform mediump mat4 modelviewmatrix;
void main() {
    gl_Position = modelviewmatrix * vertex;
}

// Fragment Shader
void main() {
    gl_FragColor = vec4(0.2, 0.6, 0.8, 1);
}
```

Shader EX-2: diffuse lighting



EXAMPLE: Phong-Shaded Diffuse-only lighting...

```
// Vertex Shader

attribute vec4 a_vertPos;
attribute vec4 a_normalVec;
uniform mat4 u_modelviewMat;
uniform mat4 u_normalMat;
uniform vec3 u_lightPos;

varying vec3 v_norm;
varying vec3 v_toLight;

void main()
{
    gl_Position = u_modelviewMat * a_vertPos;
    v_norm = u_normalMat * a_normalVec;
    v_toLight =
        vec3(a_lightPos - a_vertPos);
}
```

```
// Fragment Shader

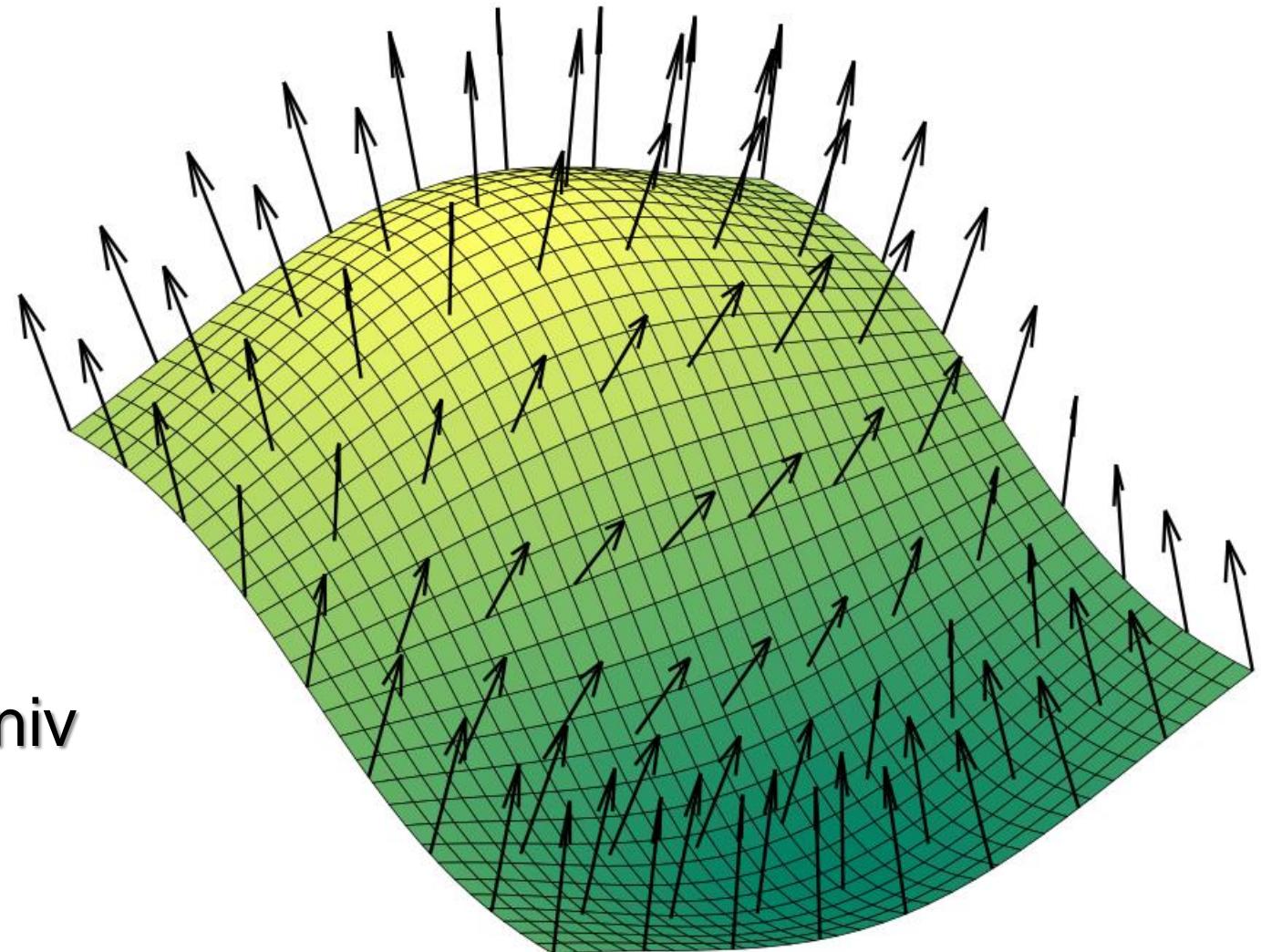
varying vec3 v_norm;
varying vec3 v_toLight;

void main()
{
    const vec3 DiffColr =
        vec3(0.2, 0.6, 0.8);
    float diff = clamp(
        dot(normalize(v_norm),
            normalize(v_toLight))
        ,
        0.0, 1.0); // stay in [0,1]

    gl_FragColor =
        vec4(DiffColr * diff, 1.0);
}
```

How To Build and Use Surface Normals in WebGL

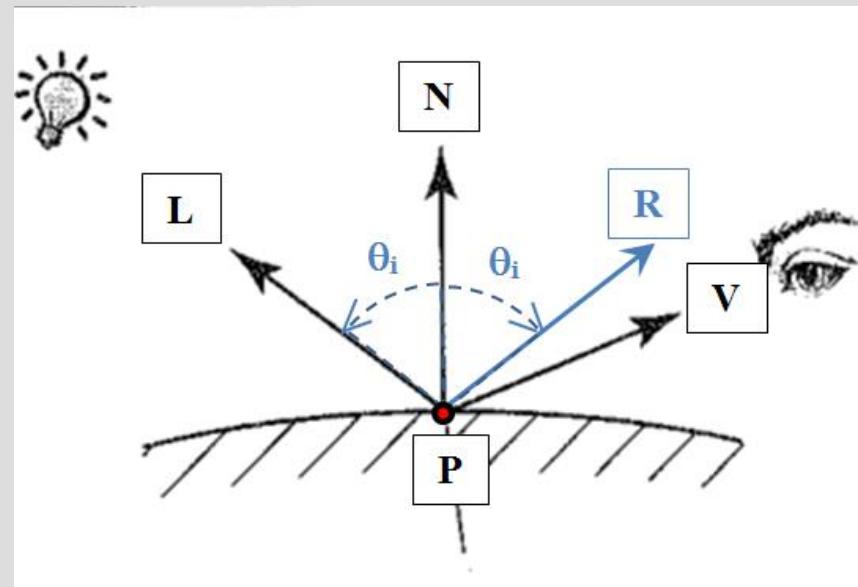
Jack Tumblin
Northwestern Univ
Comp Sci 351-1
Fall 2021



Why We Need Normal **N** At Every Vertex:

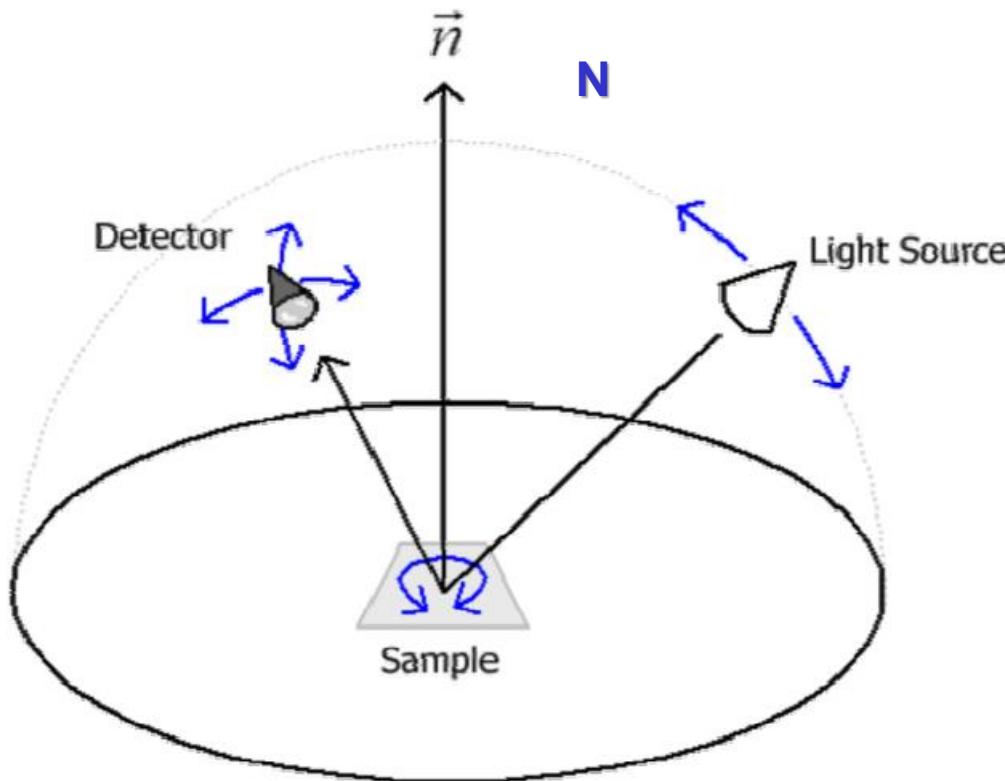
<http://math.hws.edu/graphicsbook/source/webgl/cube-camera.html>

How else could we know the ‘reflected’ direction for each vertex of a mirrored surface?



WHY do we need **N**? Because **Reflectance** set by **ANGLES** from **N**:

- Gonioreflectometer is device to measure BRDFs Bi-directional Reflectance Distribution Function (4D!)

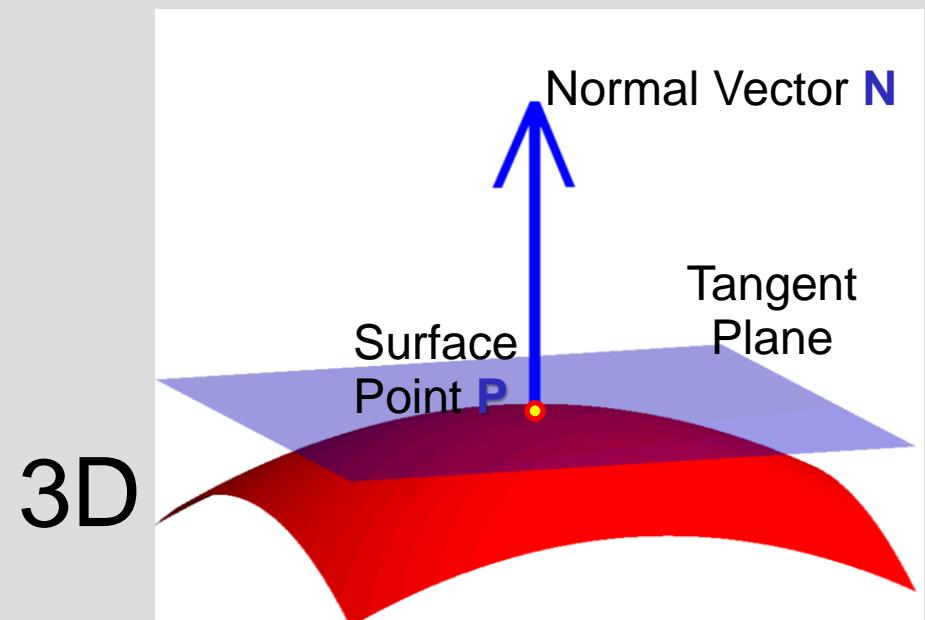
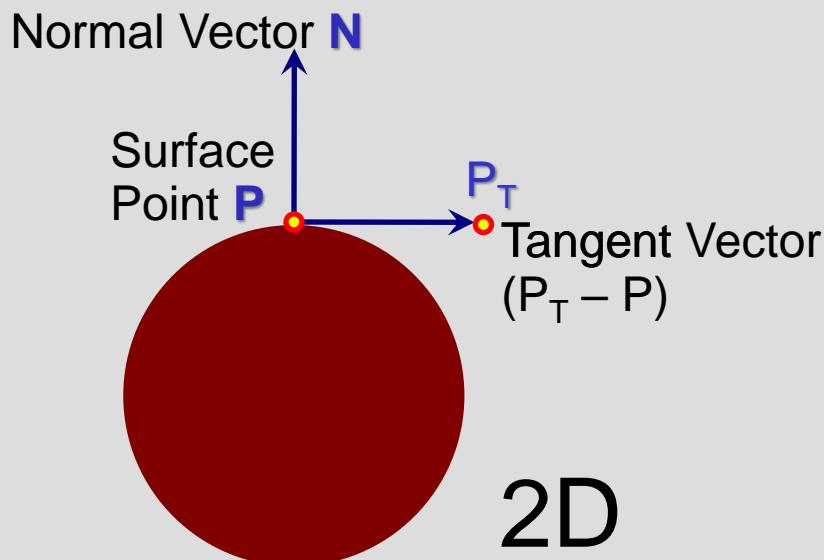


Define: “Normal” Vector $\mathbf{N} ==$ Local Surface Orientation

\mathbf{N} = Perpendicular Vector at any Surface Point

More formally:

- Unit-length vector \mathbf{N} at surface point \mathbf{P}
- Vector \mathbf{N} also defines tangent plane at \mathbf{P} :
(For all points \mathbf{P}_T in tangent plane, $(\mathbf{P}_T - \mathbf{P}) \cdot \mathbf{N} = 0$)



OK; so we need ‘normal’ attribute..

‘Unit-Length Surface Normal Vector **Attribute**’

Usually just called ‘**the normal**’ (vector, direction only)

How do I compute it?

- For a triangle:
cross-product of edge vectors
- For a mesh node:
good: avg. of adjacent triangle normals
better: area-weighted avg. of normals
- For ideal shape: (e.g. sphere, cylinder, torus...)
best: find **⊥** direction by trigonometry

2D Lines: A New, Homogeneous Form for you...

We can write any **2D Line** as the set of points (x,y,w) that satisfy this homogeneous equation:

$$Ax + By + Cw = 0 \quad (\text{not a function } f(x) = y; \text{ an equation!})$$

Where:

- each parameter A, B and C is real & scalar, AND
- Either A or B is nonzero, $(A=B=0; C\neq 0? \rightarrow \text{no solutions possible for } (x,y)!$
 $(A=B=0; C=0? \rightarrow \text{all } (x,y) \text{ are solutions.})$
- **2D Line** params (A,B,C) are ‘homogeneous’;
scaling has no effect: (A,B,C) & $(7A,7B,7C)$
describe the same line \rightarrow THUS
- Line crosses X axis at $x,y = (-C/A, 0)$
- Line crosses Y axis at $x,y = (0, -C/B)$
- Perpendicular 2D line? (I want YOU to figure it out...)

3D Planes? Similarly, here's a *Homogeneous Form*

We can write any **3D Plane** as the set of points (x,y,z,w) that satisfy this homogeneous equation:

$$Ax + By + Cz + Dw = 0$$

(*not* a function $f(x) = y$; this is an **equation!**)

Where:

- each parameter **A**, **B**, **C** and **D** is real & scalar, AND
- Either **A**, **B** or **C** is nonzero, ($A=B=C=0; D \neq 0 \rightarrow$ *no* (x,y) are solutions;
 $A=B=C=0; D=0 \rightarrow$ *all* (x,y) are solutions.)
- **3D Plane** params **(A,B,C,D)** are ‘**homogeneous**’;
scaling has no effect: **(A,B,C,D)** & **(7A,7B,7C,7D)**
- Plane’s surface normal vec: **n** = $(A,B,C) / \|(A,B,C)\|$
- Unit surface-normal vector **n** is \perp to plane, and
- For any point **P** = (px,py,pz) on plane, **P · n = -D**

? Surface Normals for TRIANGLES ?

- For one flat isolated triangle:
find normal vector perpendicular to
the triangle's 3D plane

And in WebGL,

‘**the Unit-Length Surface Normal Vector Attribute**’
is usually just called ‘**the normal**’ (vector direction)

A single isolated flat triangle? Easy!
same **normal** for all 3 vertices.

? Surface Normals for TRIANGLES ?

‘the Unit-Length Surface Normal Vector Attribute’
is usually just called ‘the normal’ (vector direction)

How do I compute it?

- For a triangle:
cross-product of edge vectors
- For a mesh node:
good: *avg. of adjacent triangle normals*
better: *area-weighted avg. of normals*
- For ideal shape: (e.g. sphere, cylinder, torus...)
best: find \perp direction by **trigonometry**

REVIEW:

*Vector **Dot** Product: ($\mathbf{u} \cdot \mathbf{v}$)*

$$\mathbf{u} \cdot \mathbf{v} = x_u x_v + y_u y_v + z_u z_v$$

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}| |\mathbf{v}| \cos(\theta)$$

THUS: $\cos\theta = (\mathbf{u} \cdot \mathbf{v}) / (|\mathbf{u}| |\mathbf{v}|)$

- *Purely a **scalar** result -- not a vector!*
- *What happens when the vectors are unit-length?*
- *What does it mean if dot product == 0 or == 1?*

REVIEW:

*Vector **Cross** Product: $(\mathbf{u} \times \mathbf{v})$*

- *The result is a vector, not a scalar, **perpendicular** to the plane formed by the two crossed vectors*
- *Can be computed using the determinant of:*

$$\begin{vmatrix} i & j & k \\ x_v & y_v & z_v \\ x_u & y_u & z_u \end{vmatrix}$$

$$\mathbf{u} \times \mathbf{v} = (y_u z_v - z_u y_v) \mathbf{i} + (z_u x_v - x_u z_v) \mathbf{j} + (x_u y_v - y_u x_v) \mathbf{k}$$

- *Vector magnitude: $\|\mathbf{u} \times \mathbf{v}\| = \|\mathbf{u}\| \cdot \|\mathbf{v}\| \cdot \sin(\theta)$*
- *Self-Cross? Always zero! $\mathbf{u} \times \mathbf{u} = 0$*

Effect of Light Direction?

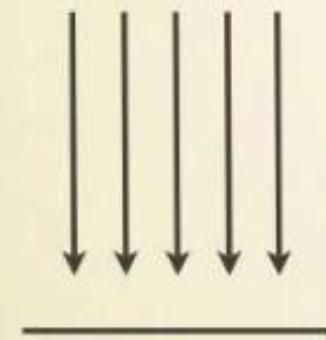
Simple Answers:

'Reflectance' == Fraction of visible light returned from a 3D scene

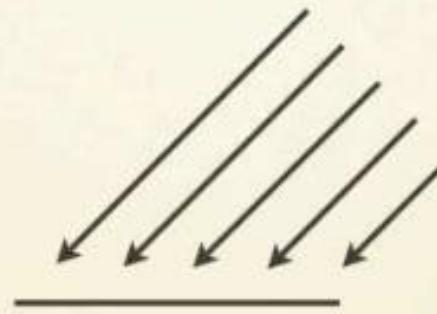
"Incident Light" (Irradiance)

has Cosine Weighting:

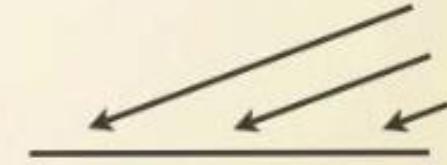
At shallower angles,
parallel rays of light spread over a wider surface area:



$$\cos(0^\circ) = 1$$



$$\cos(45^\circ) \approx .71$$



$$\cos(70^\circ) \approx .34$$

Lambertian Material: color varies by $\cos(\theta)$ falloff only

- Just one ‘Material’ parameter (with several names):
 K_{diffuse} or ‘albedo’ or ‘diffuse reflectance’ or ...
== returned fraction of incident light intensity I_{diffuse}
== $0 \leq R, G, B \leq 1$.
- Viewed color depends ONLY on illumination direction:
find $(N \cdot L)$ to impose $\cos(\theta)$ falloff
- Color is same for all viewing directions \forall
- ‘Diffuse Lighting’ defined by:

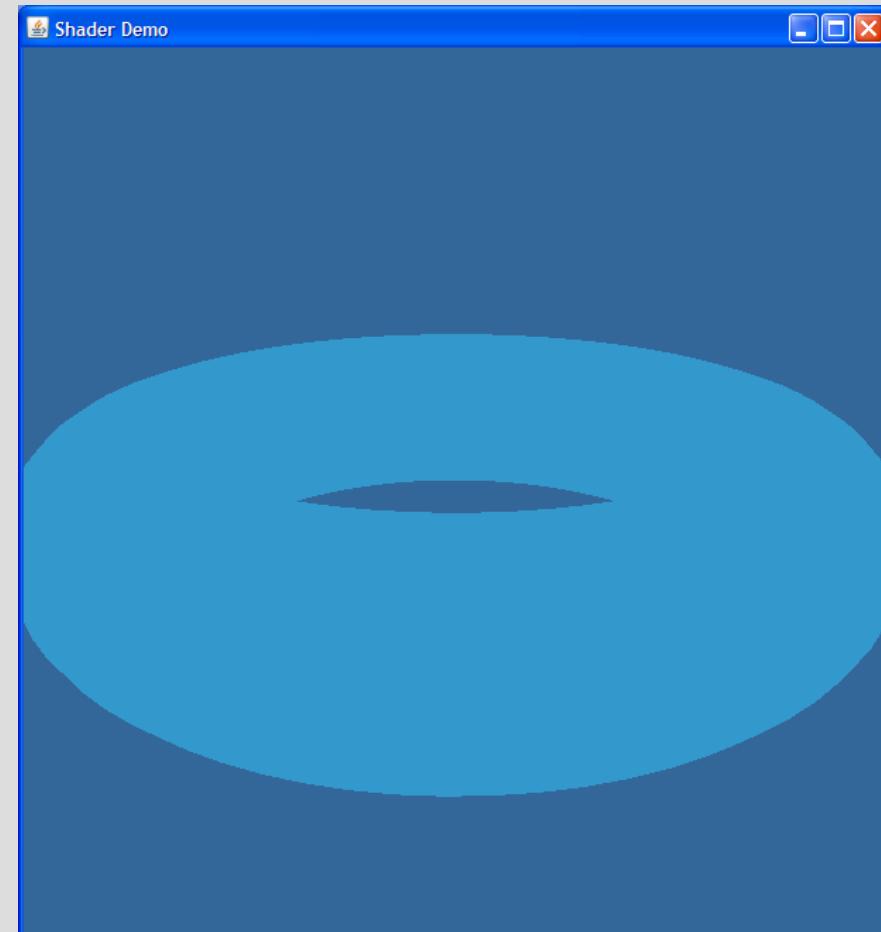
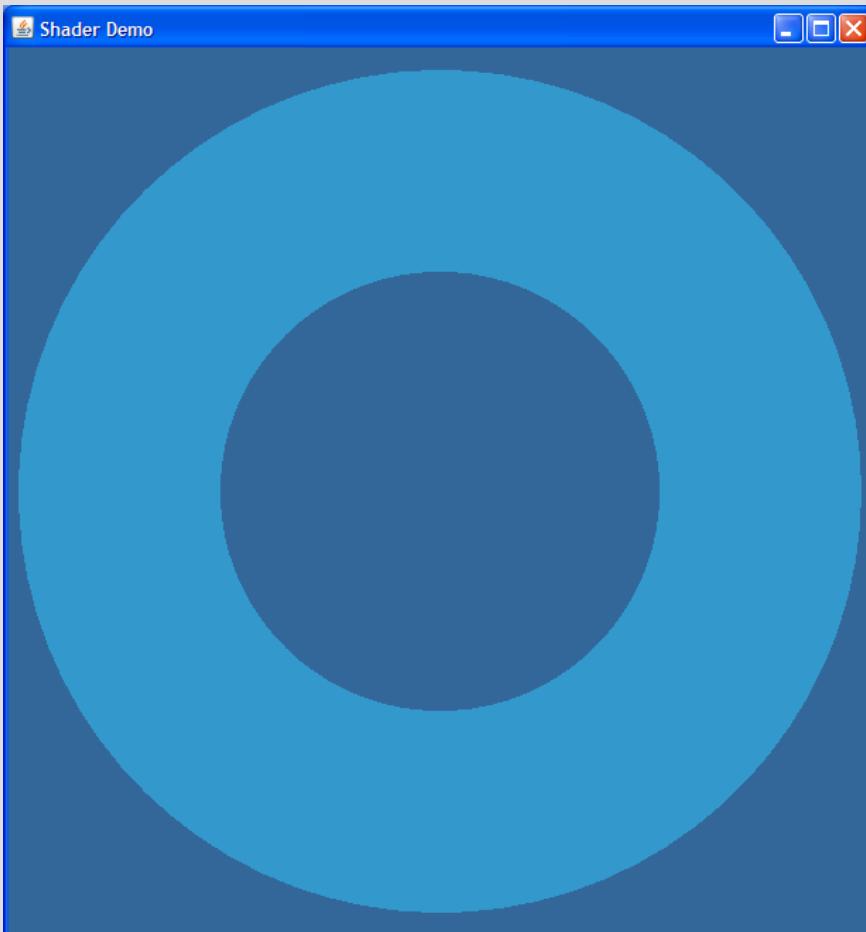
$$\text{On-Screen}[r,g,b] = K_{\text{diffuse}}[r,g,b] * I_{\text{diffuse}}[r,g,b] * \text{dotProduct}(N, L);$$

N == surface normal vector

L == light vector (from surface point to light source)

‘Ambient’ Lighting: (mimics Proj A, B)

- equal light **from** all directions (ambient illum),
- and equal light reflected **to** all viewing directions

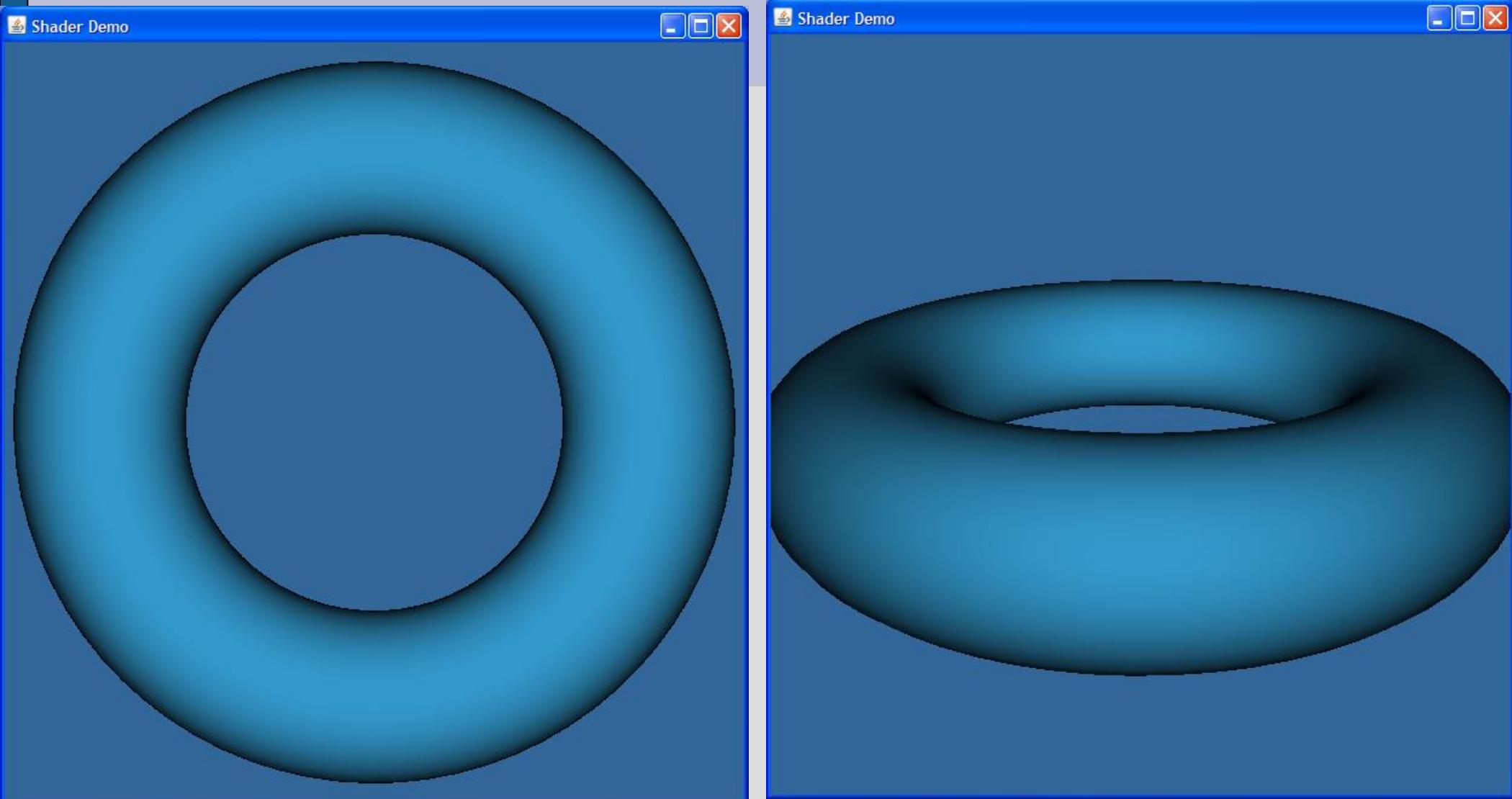


SEE STARTER CODE:

2021.11.12.ControlDiffuseShading

“diffuse lighting”: $\mathbf{N} \cdot \mathbf{L}$

Shown: Point-Light Source (\mathbf{L}) at Eye-point.



SEE STARTER CODE:

2021.11.12.ControlDiffuseShading

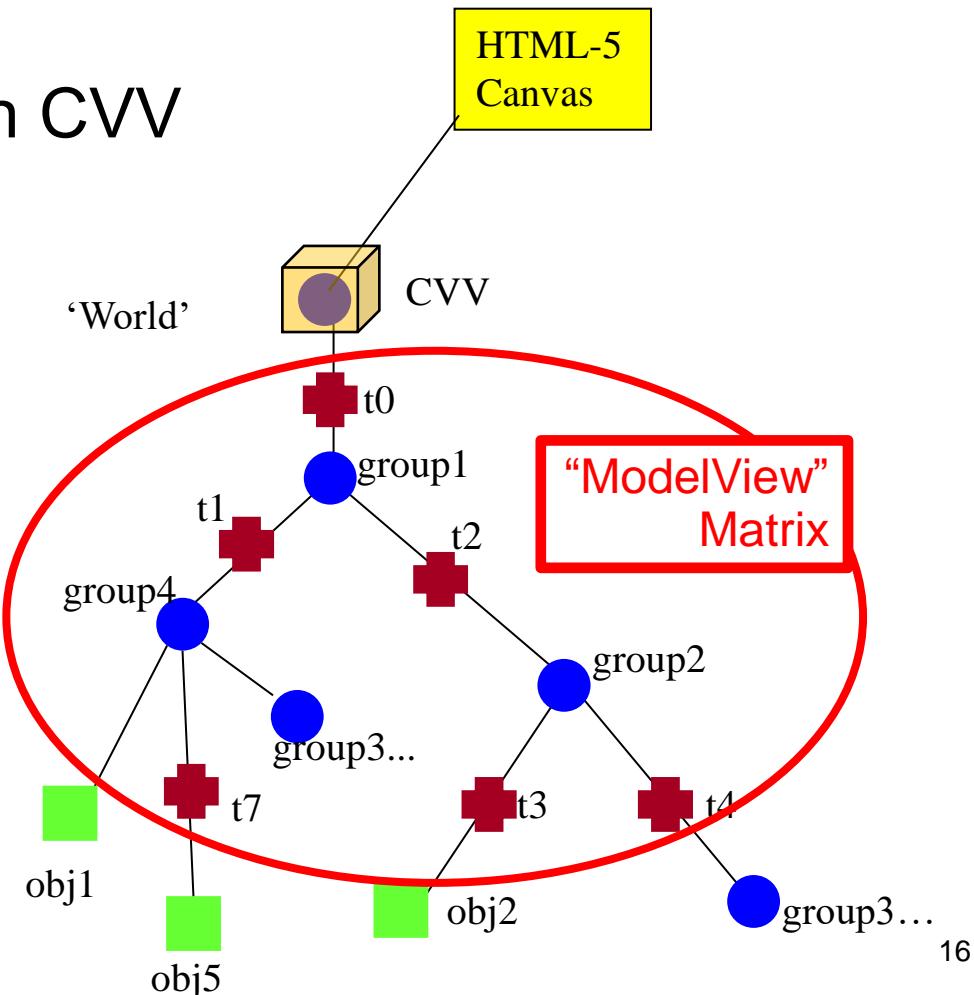
WebGL: Vertex Position Pipeline

RECALL: in scene graphs,
Vertices & values move upwards,
transform calls move downwards

Project A: Draw ‘world’ directly in CVV

? Where do ‘normal’ vectors N
Fit into the Scene Graph?

? HOW do we transform
The ‘normal’ vectors N?

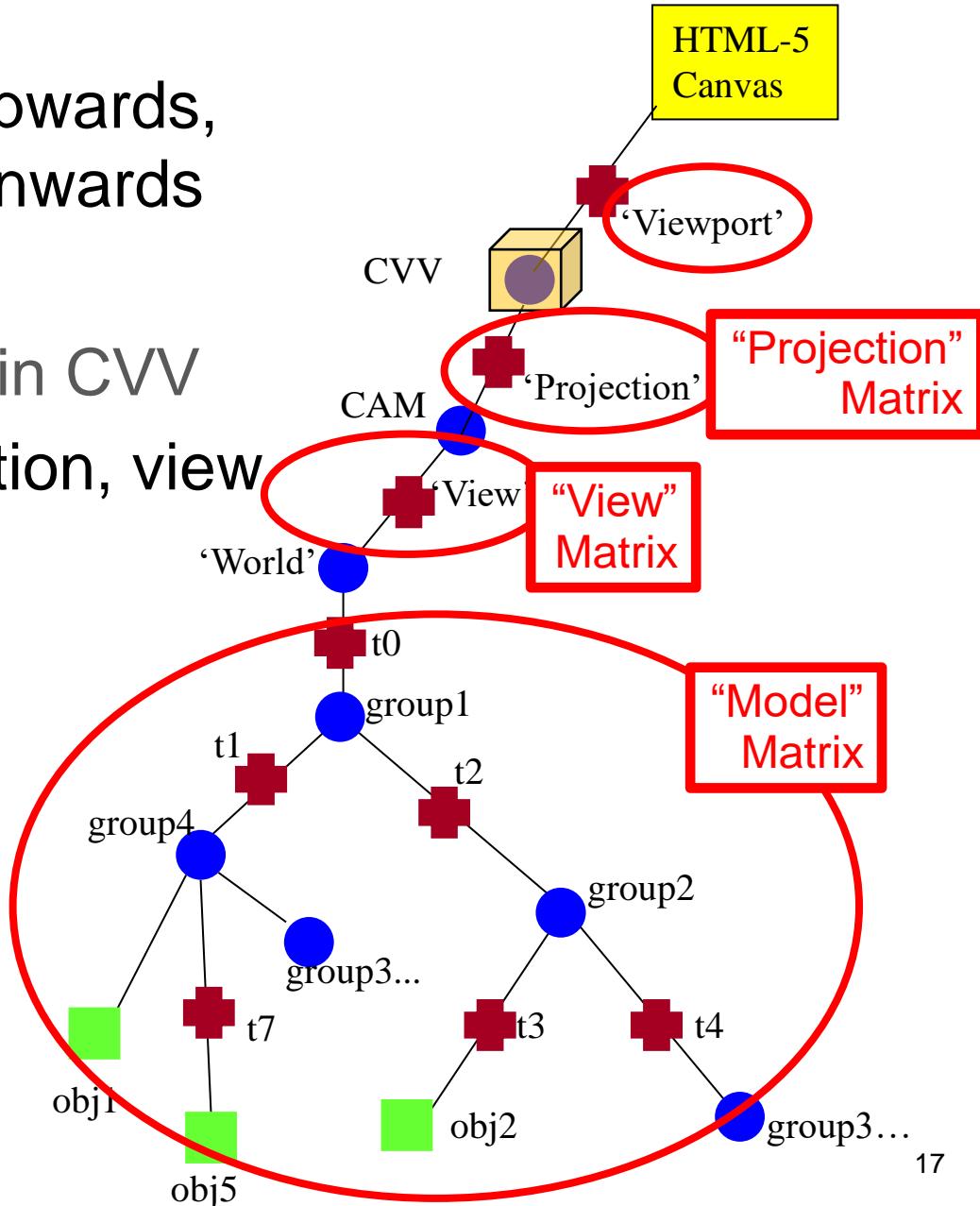


WebGL: Vertex Position Pipeline

RECALL: in scene graphs,
Vertices & values move upwards,
transform calls move downwards

Project A: Draw ‘world’ directly in CVV

Project B: Add viewport, projection, view



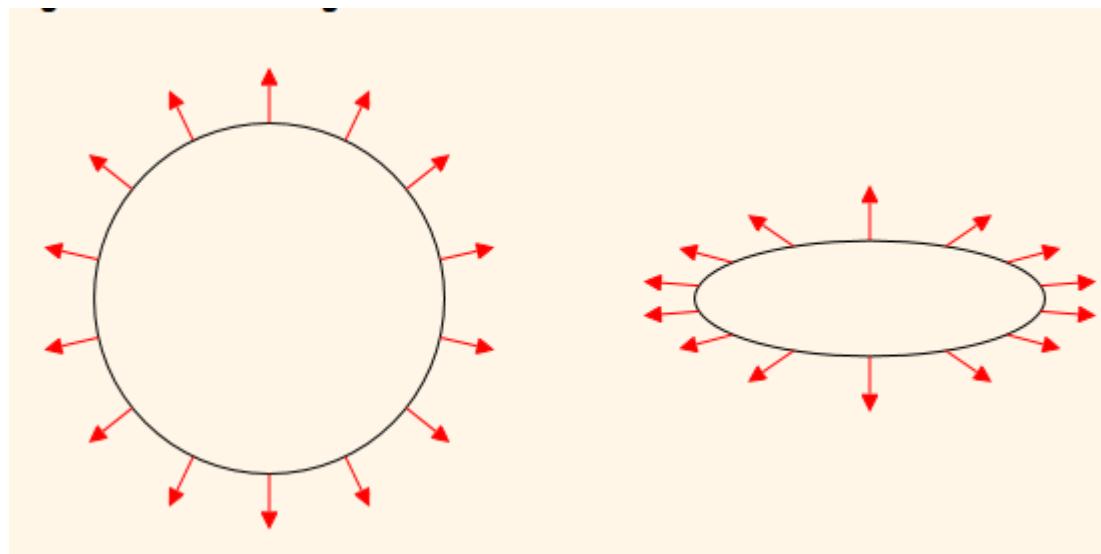
Transforming Normals

We KNOW how to transform vertex positions.

*Can we transform **Normal Vectors** with the same matrix?*

ALMOST
but not always!

non-uniform scaling? (stretched robot arm, etc)



Transforming Normals

We *KNOW* how to transform vertex positions.

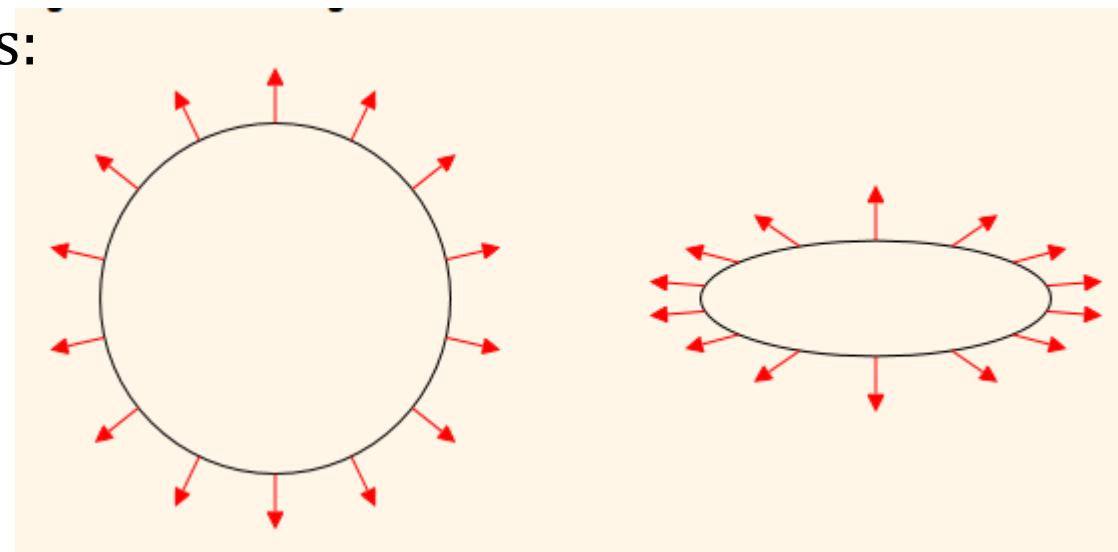
Can we transform Normal Vectors with the same matrix?

ALMOST always yes,

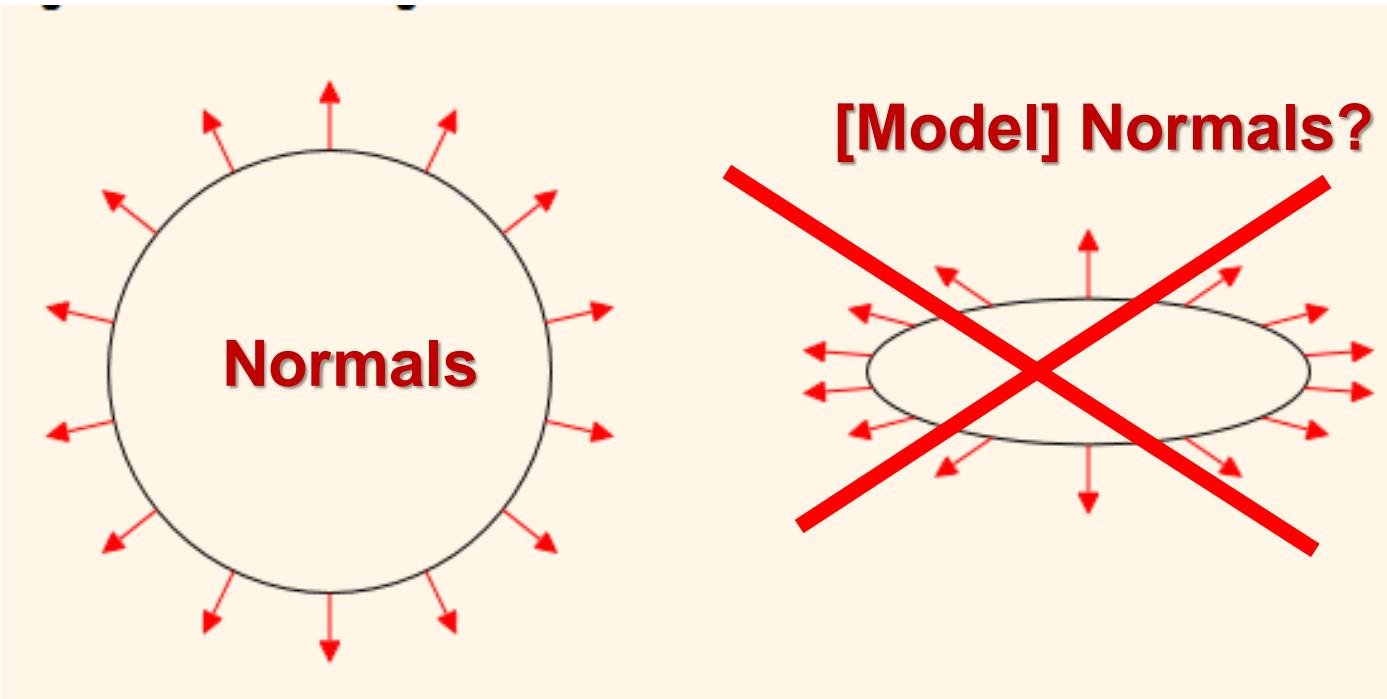
*but not always: → thus the answer is **NO.***

we need a special ‘**normal** transform **matrix**’

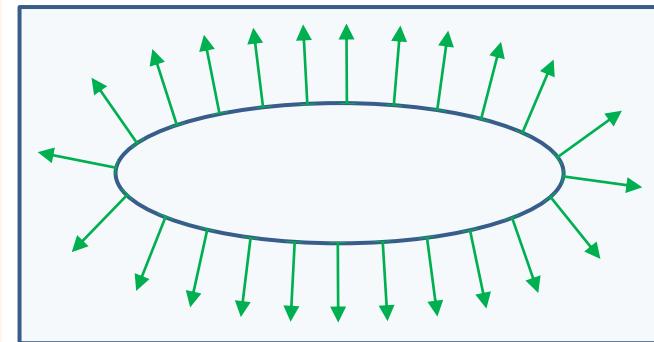
because non-uniform scaling of shapes (stretched robot arm, etc) distorts these normals:



Transforming Normals



[Model]^{-T} Normals



SOLUTION: use *inverse-transpose*:

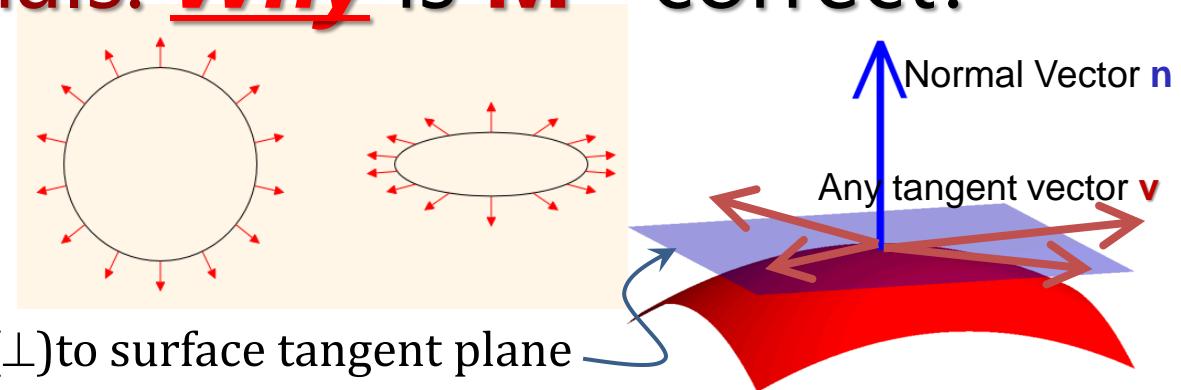
Normal Matrix == (Model Matrix)^{-T}

Why? see: <http://www.lighthouse3d.com/tutorials/glsl-12-tutorial/the-normal-matrix/>

or Canvas: Project C Reading: "How To Transform Normals"

How? [cuon-matrix-quat03.js](#) functions ...

Transforming Normals: Why is \mathbf{M}^{-T} correct?



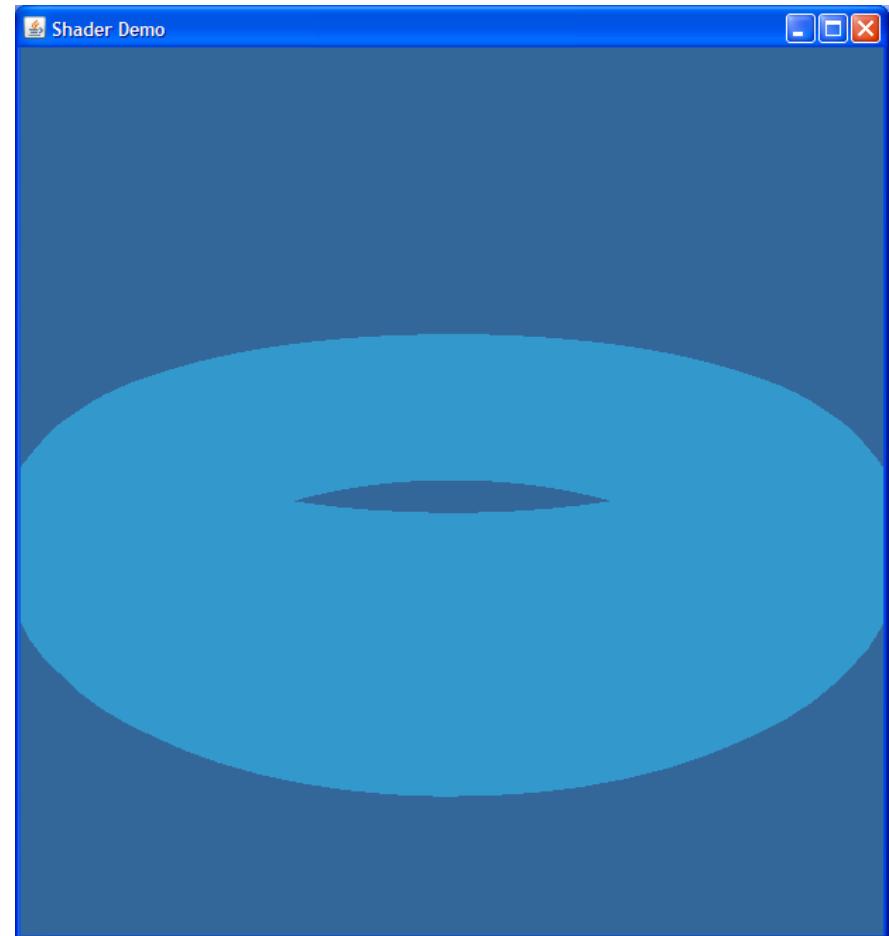
- ▶ normal vector == perpendicular (\perp) to surface tangent plane
- ▶ Any transform matrix \mathbf{M} applied to the surface applies to the tangent plane too.
- ▶ Any vector \mathbf{v} in the tangent plane is \perp to \mathbf{n} , thus $\mathbf{n} \cdot \mathbf{v} = \mathbf{0}$, or equivalently: $\mathbf{n}^T \mathbf{v} = \mathbf{0}$
- ▶ REVIEW:
 - ▶ For any non-singular matrix \mathbf{M} we can find an inverse \mathbf{M}^{-1} that cancels it: $\mathbf{M}^{-1} \mathbf{M} = \mathbf{I}$
 - ▶ Transpose lets us multiply column vector \mathbf{v} and matrix \mathbf{A} in either order: $\mathbf{A}\mathbf{v} = \mathbf{v}^T \mathbf{A}^T$
- ▶ Expand $\mathbf{n}^T \mathbf{v} = \mathbf{0}$ with the 'do-nothing' identity matrix: $\mathbf{n}^T (\mathbf{M}^{-1} \mathbf{M}) \mathbf{v} = \mathbf{0}$
- ▶ Associate each matrix with its neighbor: $(\mathbf{n}^T \mathbf{M}^{-1})(\mathbf{M}\mathbf{v}) = \mathbf{0}$ and then look closely:
 - ▶ $(\mathbf{M}\mathbf{v})$ == Any and all transformed tangent-plane vectors
 - ▶ $(\mathbf{n}^T \mathbf{M}^{-1})$ == The transformed normal vector *guaranteed* \perp to all the tangent-plane vectors
- ▶ Rearrange **transformed normal vector** using transpose: $(\mathbf{n}^T \mathbf{M}^{-1}) = (\mathbf{M}^{-1})^T \mathbf{n} = \boxed{\mathbf{M}^{-T} \mathbf{n}}$

!END!



The University of New Mexico

EX 1 – ambient lighting





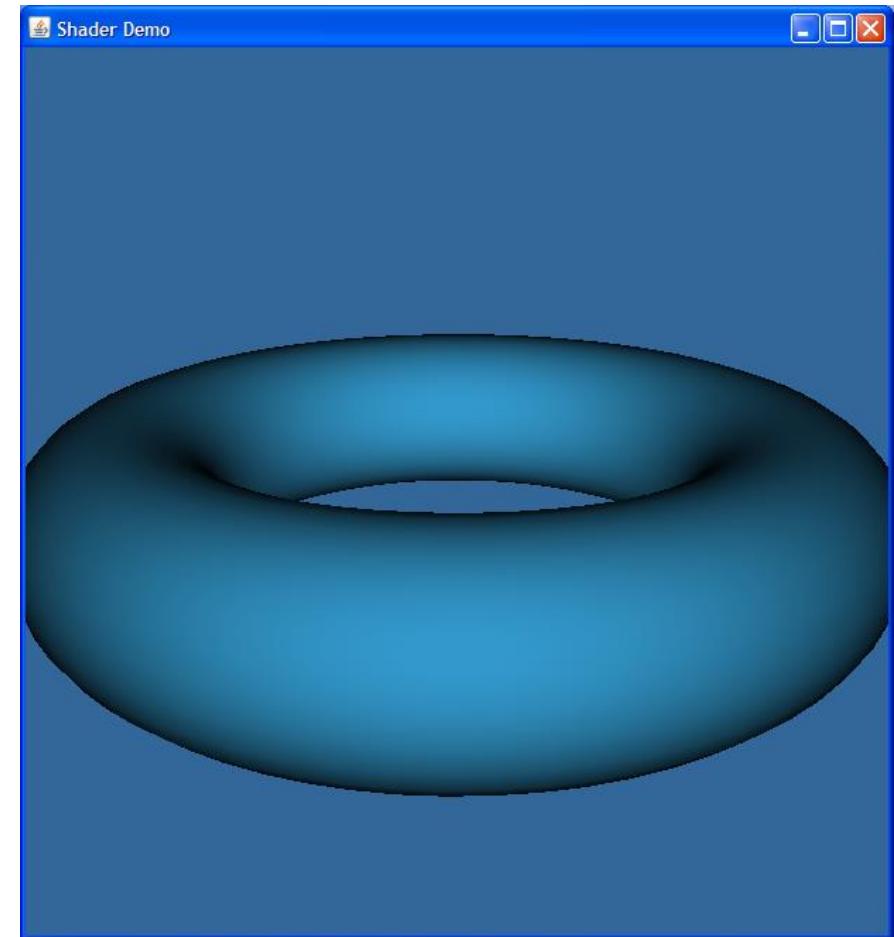
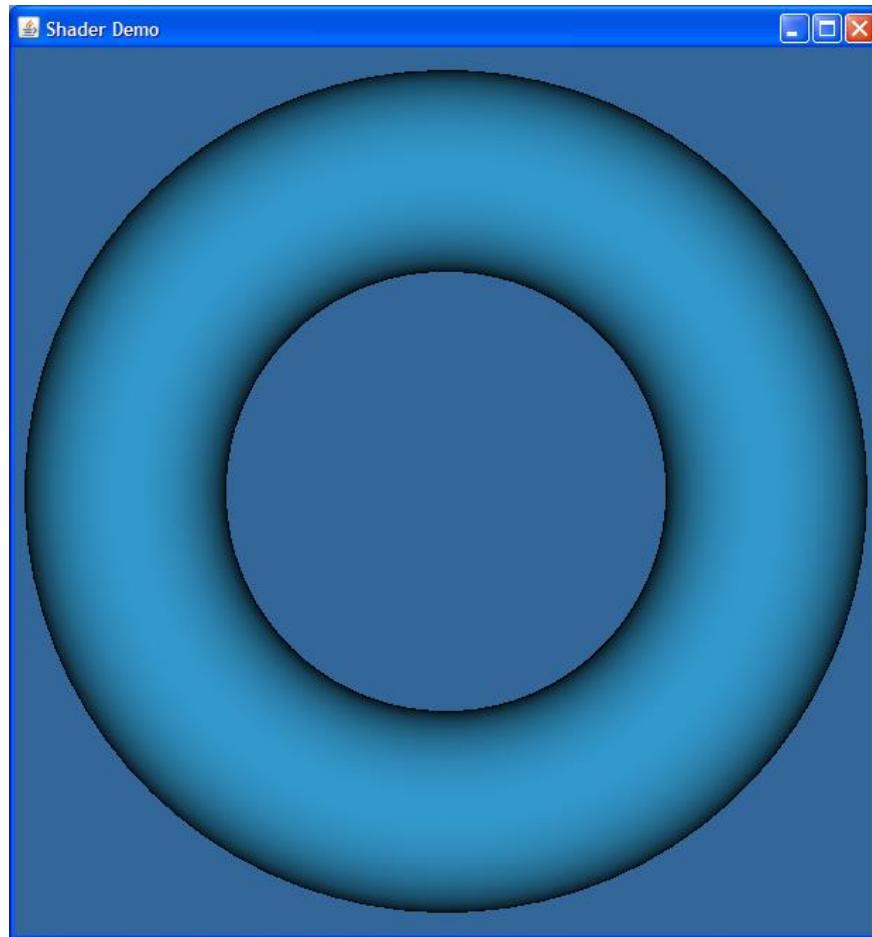
The University of New Mexico

EX 1– ambient lighting

```
// Vertex Shader
attribute highp vec4 a_vertPos;
uniform mediump mat4 modelviewmatrix;
void main() {
    gl_Position = modelviewmatrix * vertex;
}

// Fragment Shader
void main() {
    gl_FragColor = vec4(0.2, 0.6, 0.8, 1);
}
```

Shader EX-2: diffuse lighting



EXAMPLE: Phong-Shaded Diffuse-only lighting...

```
// Vertex Shader

attribute vec4 a_vertPos;
attribute vec4 a_normalVec;
uniform mat4 u_modelviewMat;
uniform mat4 u_normalMat;
uniform vec3 u_lightPos;

varying vec3 v_norm;
varying vec3 v_toLight;

void main()
{
    gl_Position= u_modelviewMat*a_vertPos;
    v_norm = u_normalMat * a_normalVec;
    v_toLight =
        vec3(a_lightPos - a_vertPos);
}
```

```
// Fragment Shader

varying vec3 v_norm;
varying vec3 v_toLight;

void main()
{
    const vec3 DiffColr =
        vec3(0.2, 0.6, 0.8);
    float diff = clamp(
        dot( normalize(v_norm),
            normalize(v_toLight)
        ),
        0.0, 1.0); // stay in [0,1]

    gl_FragColor =
        vec4(DiffColr * diff, 1.0);
}
```

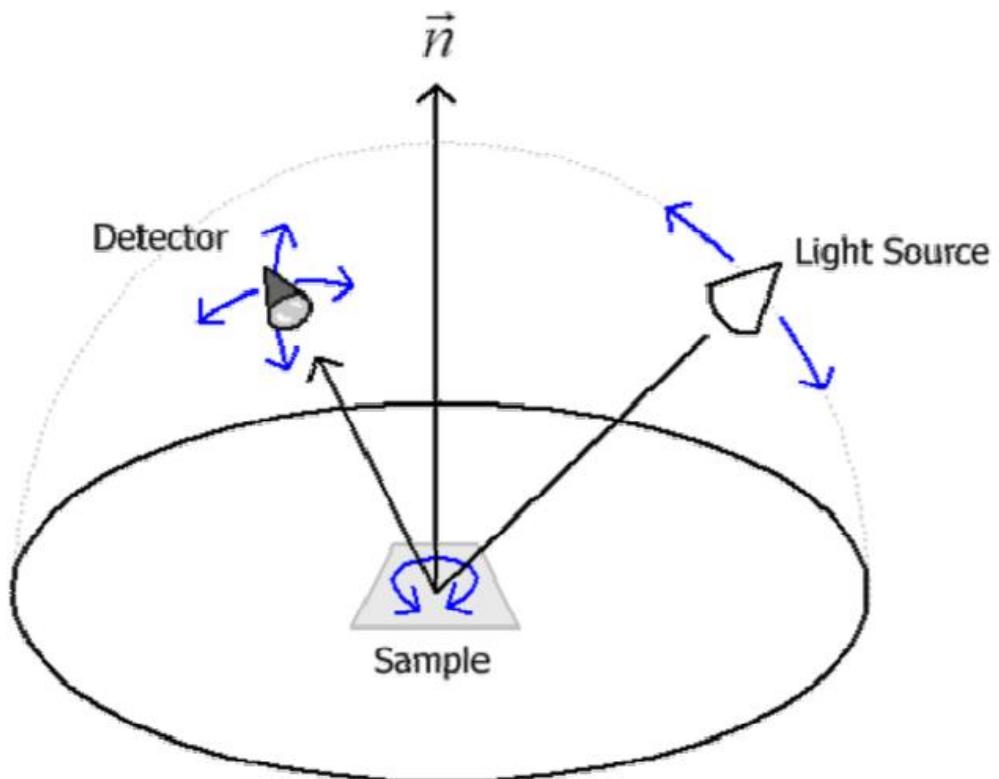
Lighting, Materials & Shading



Some slides modified from: David Kabala
Others from: Andries Van Damm, Brown Univ.

Reflectance (scalar) .vs. ‘BRDF (4D) ?Is this a Complete Measurement?

- Gonioreflectometer is device to measure BRDFs



Effect of Light Direction?

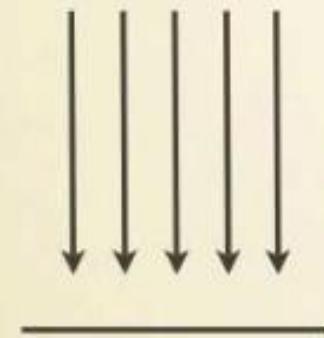
Simple Answers:

'Reflectance' == Fraction of visible light returned from a 3D scene

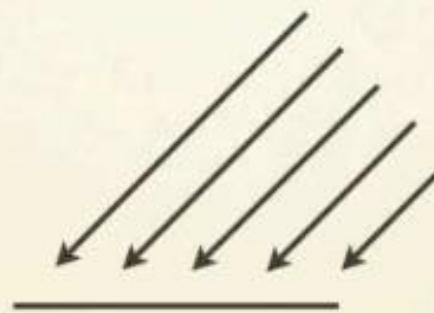
"Incident Light" (Irradiance)

has Cosine Weighting:

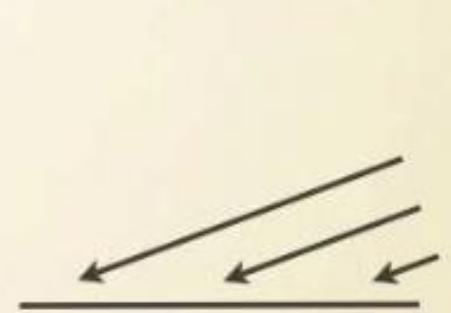
At shallow angles,
parallel rays of light spread over a wider surface area:



$$\cos(0^\circ) = 1$$

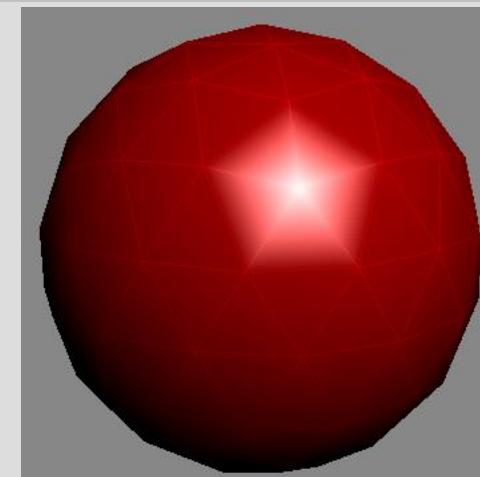
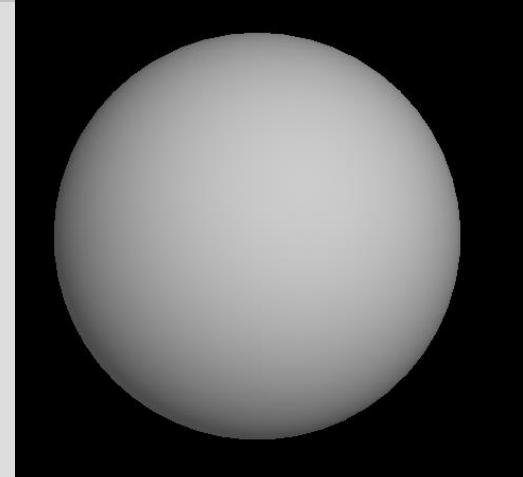
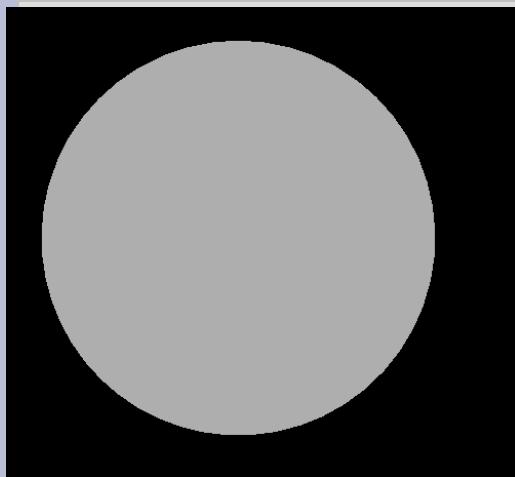


$$\cos(45^\circ) \approx .71$$

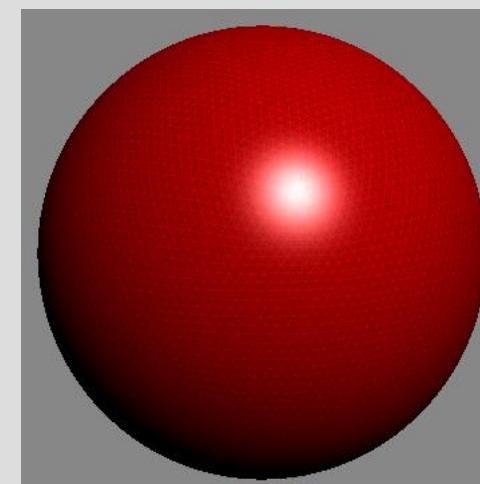


$$\cos(70^\circ) \approx .34$$

Cartoon → Diffuse → Specular



Gouraud
Shading



Phong
Shading

Chalky ‘Lambertian’ Materials: color set by $\cos(\theta)$ falloff *only*

- Just one ‘material’ parameter, but several names:
‘ K_{diffuse} ’ or ‘albedo’ or ‘diffuse reflectance’ or ...
== returned fraction of incident light intensity I_{diffuse}
== $0 \leq R, G, B \leq 1$.
- ‘Diffuse Lighting’ defined by:

$$\text{On-Screen}[r,g,b] = K_{\text{diffuse}}[r,g,b] * I_{\text{diffuse}}[r,g,b] * \text{dotProduct}(\mathbf{N}, \mathbf{L});$$

\mathbf{N} == surface normal vector

\mathbf{L} == light vector (from surface point to light source)

- Color varies **ONLY** with **illumination** direction:
find $(\mathbf{N} \cdot \mathbf{L})$ to impose $\cos(\theta)$ falloff
- Color is **same** for all **viewing** directions \forall

Phong Lighting, Phong Shading Demo:

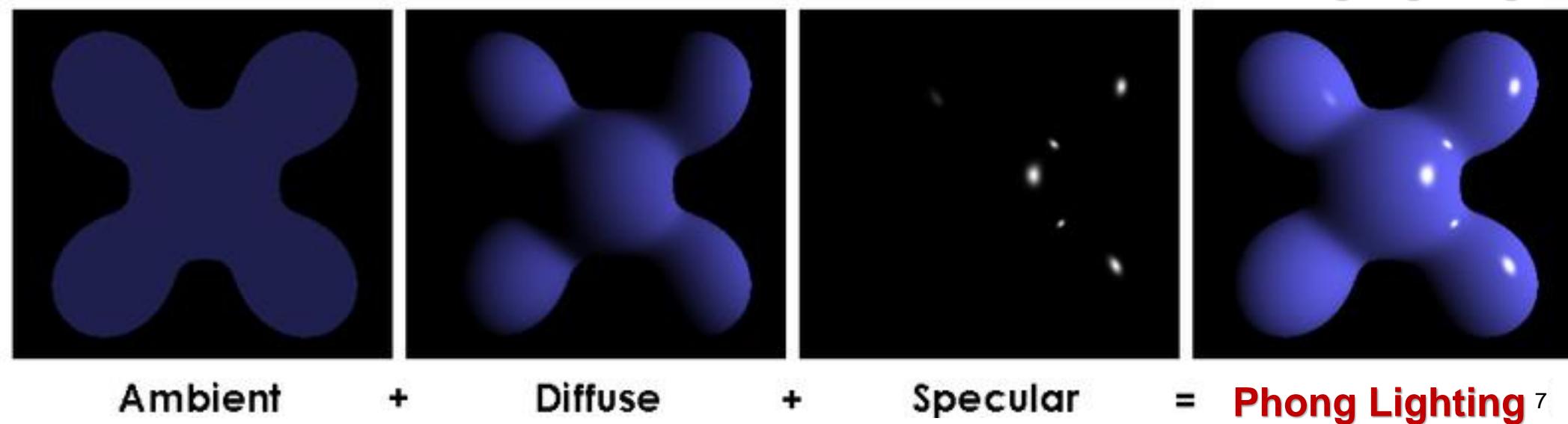
<http://www.cs.toronto.edu/~jacobson/phong-demo/>

- ▶ Demo 1: **Light Direction** vector $\mathbf{L} = (L_x, L_y, L_z)$
- ▶ Demo 2: **Phong Lighting Parameters**:
7 RGB values [0,1] + shininess
 - ▶ 3 Light-Source Strengths (RGB): I_a, I_d, I_s
 - ▶ 4 Material Reflectances: K_e, K_a, K_d, K_s
 - ▶ 1 ‘shiny-ness’ scalar S_e
aka the specular exponent: [0,~200]
- ▶ Demo 3: **Gouraud Shading / Phong Shading**
(per-vertex color) / (per-fragment color)

Phong Lighting: ~1st, ~Fast ~Good:

- ▶ Baseline Surface-Point Lighting Model: early (1975 UofU), popular, vectorized, looks good, maybe too many params...
- ▶ Sum of RGB components:
 - ▶ =Ambient – $I_a * K_a$ – 'room light' to fill shadows; approx. global illum.
 - ▶ +Diffuse – $I_d * K_d * (N \cdot L)$ – Light-direction term reveals shapes
 - ▶ +Specular – $I_s * K_s * (R \cdot V)^s$ -- Shiny Highlights show glossiness
 - ▶ +Emissive – K_e – for glow-in-the-dark objects. Usually zero.

Phong Lighting



Phong Lighting

Step 1: Find Scene Vectors L, N, V

To find *On-Screen RGB Color* at point P (*start*):



1) Find all 3D scene vectors first:

a) Light Vector L :

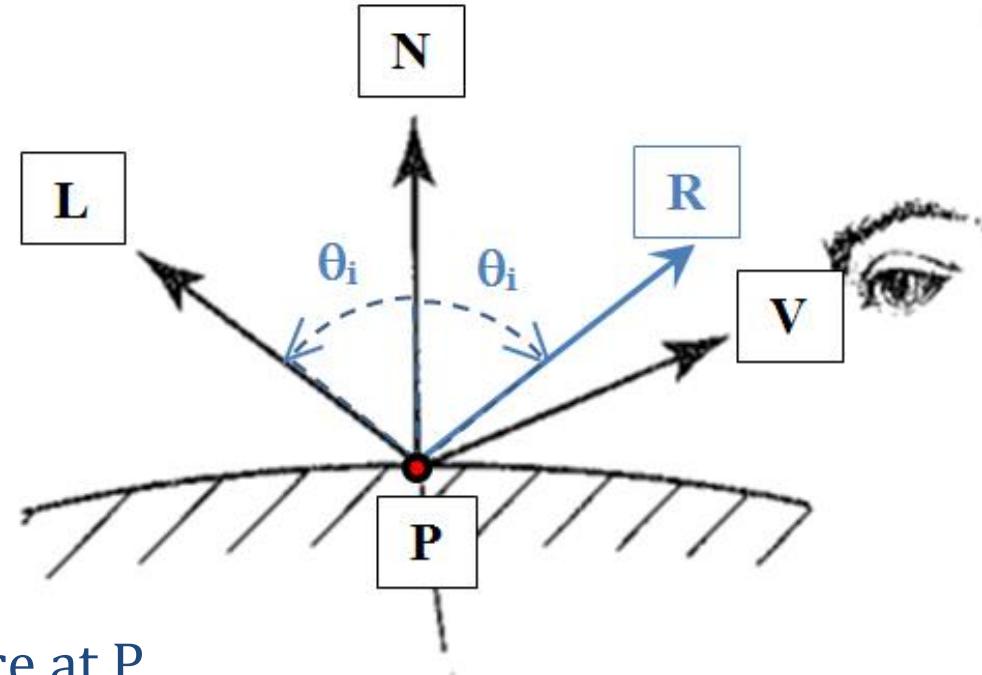
unit vector towards light source

b) Normal Vector N :

unit vector perpendicular to surface at P

c) View Vector V :

unit vector towards camera eye-point



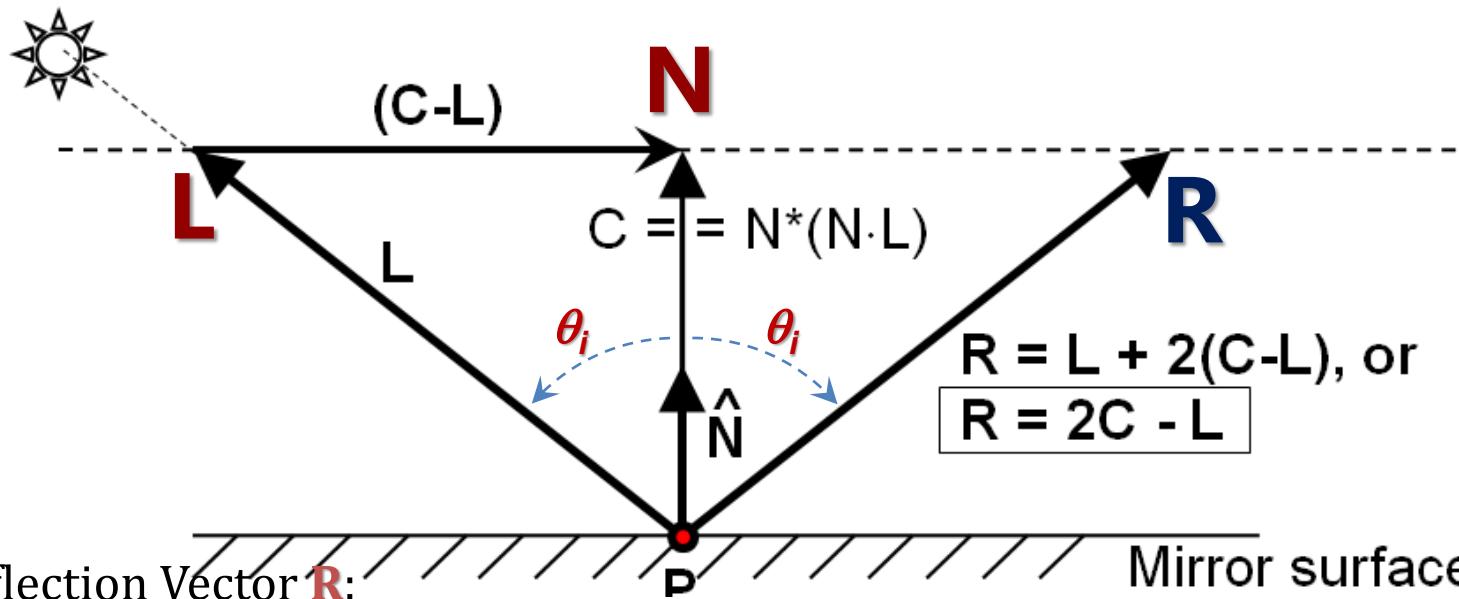
On to step 2: how do we find the Reflected-light Vector R ?

Phong Lighting

Step 2: Find reflection Vector R

To find *On-Screen RGB Color*

at point P (*cont'd*):



2) COMPUTE the Light Reflection Vector R :

- Given unit light vector L , find lengthened normal C
 $C = N(L \cdot N)$
- In diagram, if we add vector $2^*(C-L)$ to L vector we get R vector. Simplify:
 $R = 2C - L$
- Result:** unit-length R vector GLSL-ES → See built-in '`reflect()`' function
(If N is a unit-length vector, then R vector length matches L vector length)

Phong Lighting

Step 3: Gather Light & Material Data

To find *On-Screen RGB Color*
at point **P** (*cont'd*):

3) For each light source, gather:

- ▶ RGB triplet for **Ambient** Illumination **I_a** $0 \leq I_{ar}, I_{ag}, I_{ab} \leq 1$
- ▶ RGB triplet for **Diffuse** Illumination **I_d** $0 \leq I_{dr}, I_{dg}, I_{db} \leq 1$
- ▶ RGB triplet for **Specular** Illumination **I_s** $0 \leq I_{sr}, I_{sg}, I_{sb} \leq 1$

For each surface material, gather:

- ▶ RGB triplet for **Ambient** Reflectance **K_a** $0 \leq K_{ar}, K_{ag}, K_{ab} \leq 1$
- ▶ RGB triplet for **Diffuse** Reflectance **K_d** $0 \leq K_{dr}, K_{dg}, K_{db} \leq 1$
- ▶ RGB triplet for **Specular** Reflectance **K_s** $0 \leq K_{sr}, K_{sg}, K_{sb} \leq 1$
- ▶ RGB triplet for **Emissive** term(often zero) **K_e** $0 \leq K_{er}, K_{eg}, K_{eb} \leq 1$
- ▶ Scalar 'shinyness' or 'specular exponent' term **S_e** $1 \leq S_e \leq \sim 100$



Phong Lighting

Step 4: Sum of Light Amounts

To find **On-Screen RGB Color**
at point **P (cont'd)**:

sum of each kind of light at **P**:

$$\text{Phong Lighting} = \text{Ambient} + \text{Diffuse} + \text{Specular} + \text{Emissive}$$

SUMMED for all light sources

4) For the i-th light source, find:

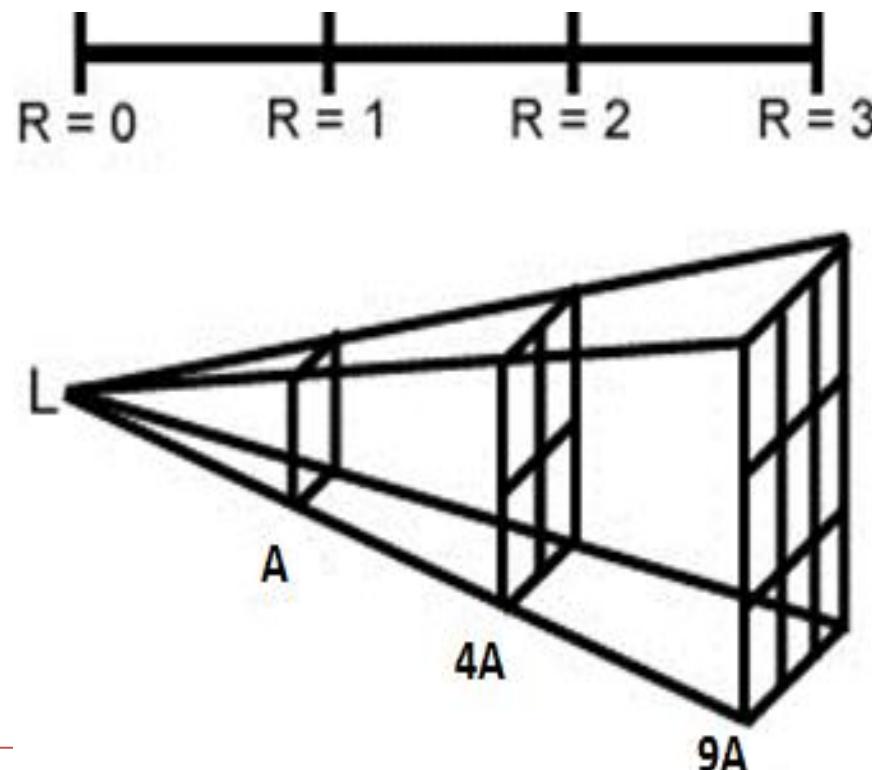
$$\begin{aligned}
 \text{RGB} &= \mathbf{K}_e + && // \text{'emissive' material; it glows!} \\
 \mathbf{I}_a^* \mathbf{K}_a + && // \text{ambient light} * \text{ambient reflectance} \\
 \mathbf{I}_d^* \mathbf{K}_d^* \mathbf{A}_{\text{Att}}^* \max(0, (\mathbf{N} \cdot \mathbf{L})) && // \text{diffuse light} * \text{diffuse reflectance} \\
 \mathbf{I}_s^* \mathbf{K}_s^* \mathbf{A}_{\text{Att}}^* (\max(0, \mathbf{R} \cdot \mathbf{V}))^{\mathbf{s}_e} && // \text{specular light} * \text{specular reflectance}
 \end{aligned}$$

- ▶ Distance Attenuation scalar: $0 \leq \mathbf{A}_{\text{Att}} \leq 1$
 - ▶ Fast, OK-looking default value: $\mathbf{A}_{\text{Att}} = 1.0$
 - ▶ Physically correct value: $\mathbf{A}_{\text{Att}}(\mathbf{d}) = 1 / (\mathbf{d} \cdot \text{distance to light})^2$ (too dark too fast!)
 - ▶ Faster, Nice-looking 'Hack': $\mathbf{A}_{\text{Att}}(\mathbf{d}) = 1 / (\mathbf{d} \cdot \text{distance to light})$
 - ▶ OpenGL compromise: $\mathbf{A}_{\text{Att}}(\mathbf{d}) = \min(1, 1 / (c_1 + c_2 * d + c_3 * d^2))$
- ▶ 'Shininess' or 'specular exponent' $1 \leq \mathbf{s}_e \leq \sim 100$ (large for sharp, small highlights)



Light Transport: Inverse Square Law

- ▶ Amount of light that illuminates an object decreases with square of distance between them. Why? because:
- ▶ For a given 3D angle (solid angle),
the covered **area grows by the square** of the distance.



2 Definitions: Lighting and Shading

- ▶ **Lighting**, (or *illumination*) === process to find light color and intensity.
- ▶ ‘Point Lighting’: Compute light amounts that *arrive* at and *leave* from **a point** on a surface in a scene, as some of that arriving light gets *reflected* towards your eye...
- ▶ ***lighting results depend on:***
 - ▶ The ***light source(s) strengths*** and ***positions***,
 - ▶ The ***camera position***,
 - ▶ the ***viewed surface orientation*** (or ‘***normal***’ ***vector***)
 - ▶ the ***viewed surface material properties***,

2 Definitions: Lighting and Shading

- ▶ A vertex describes just one point on a surface, (position, surface normal, material, etc.) but GPU draws triangles!
- ▶ ***Shading == lighting estimates for all pixels between vertices.***
== lighting interpolation!
- ▶ ***Fast Shading is crucial*** to real time graphics(e.g., games) because lighting calculations are often expensive.
- ▶ ***Slow but good alternative:***
Ray-tracing computes all lighting for ***all*** pixels, can include recursive inter-reflected light, and fine ‘sub-pixel’ sampling for antialiasing, THUS more accurate & complete, but will be slow.

“Flat” Shading: → ‘per-triangle’ color

one lighting calc per **Triangle**

One computed RGB color
for all triangle vertices & all triangle pixels

Vertex Shader:

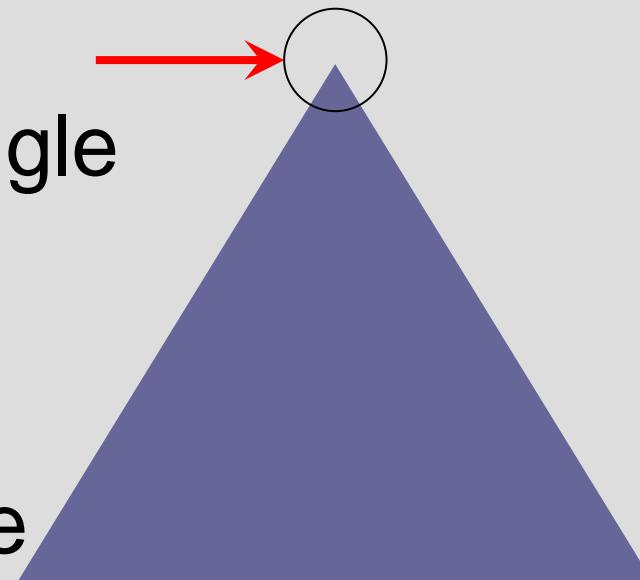
compute lighting per triangle

Fragment Shader

sets **`gl_fragColor`**

from a constant,

or from a **`uniform`** variable



Gouraud Shading: ("Goorr-Rowe"): → per-**vertex** color

one lighting calc per **Vertex**

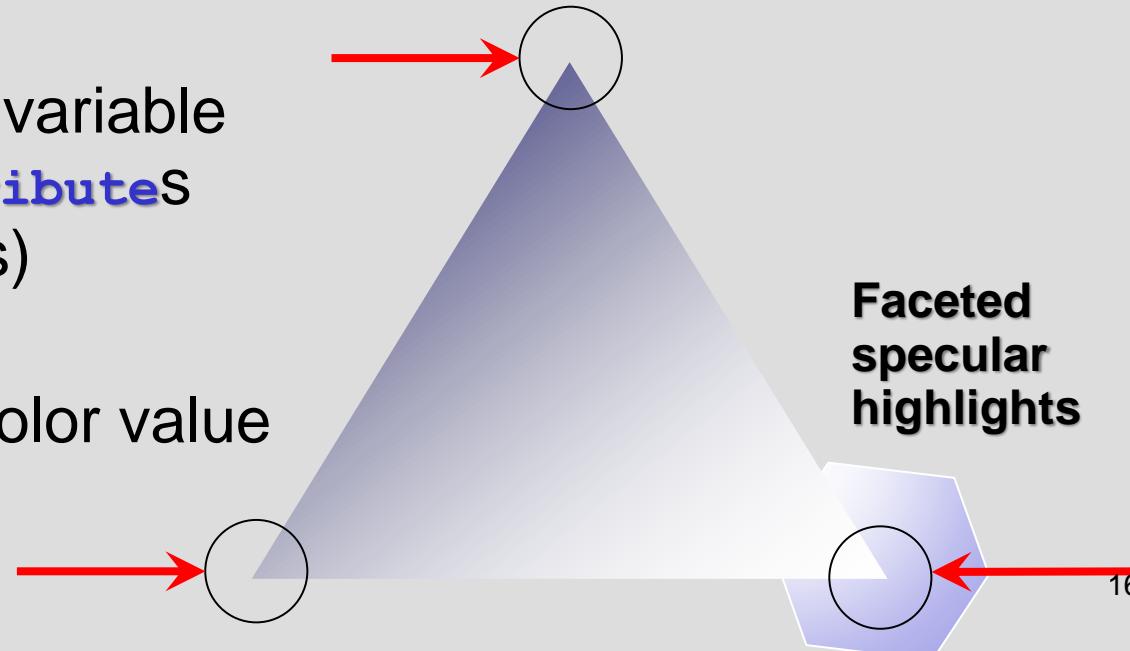
- For each **vertex**,
compute on-screen RGB color
- For each **pixel**,
bilinearly **interpolate** on-screen RGB color:

Vertex Shader

computes a '**varying**' color variable
(from vertex **attribute**s
&/or **uniform** vars)

Fragment Shader

gets a rasterized **varying** color value
uses it to set **gl_fragColor**



Why? Flat vs Gouraud shading



Flat

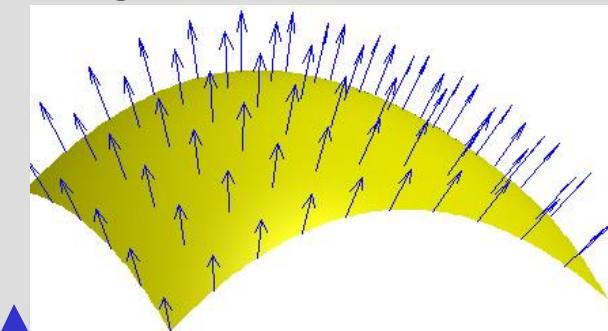


Gouraud

Phong Shading → per-pixel light efx

One lighting calc per **Pixel** (or per **Fragment**)

- For each **vertex**,
compute lighting vectors (**Norm**, **Light**, **View**, etc)
- For each **pixel**, compute or
bilinearly **interpolate** vectors,
compute lighting to set color

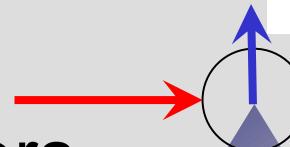


Vertex Shader

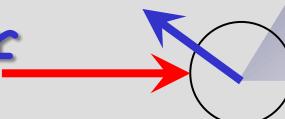
computes '**varying**' lighting vectors
(from vertex **attributes**
&/or **uniform** vars)

Fragment Shader

gets rasterized **varying** vectors
computes **gl_fragColor**



Smooth
specular
highlights!



Gouraud Shading vs Phong Shading

Tricky Terminology!

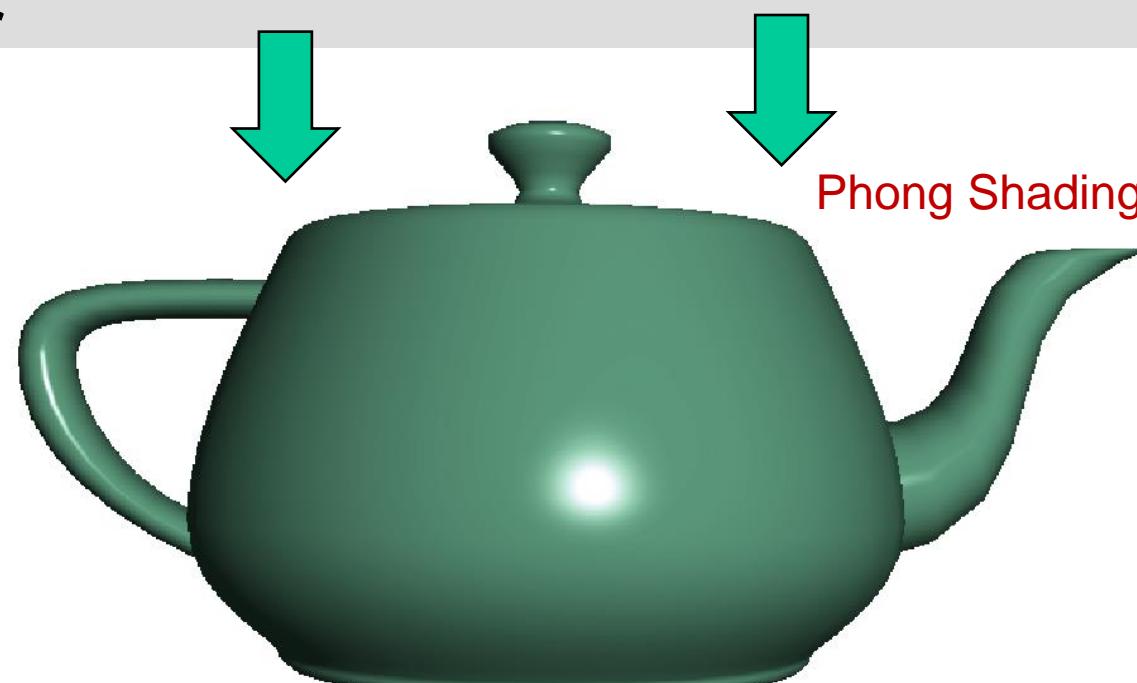
Phong Shading:
→ per-fragment color

Phong Lighting

Emissive +
Ambient +
Diffuse +
Specular = →→



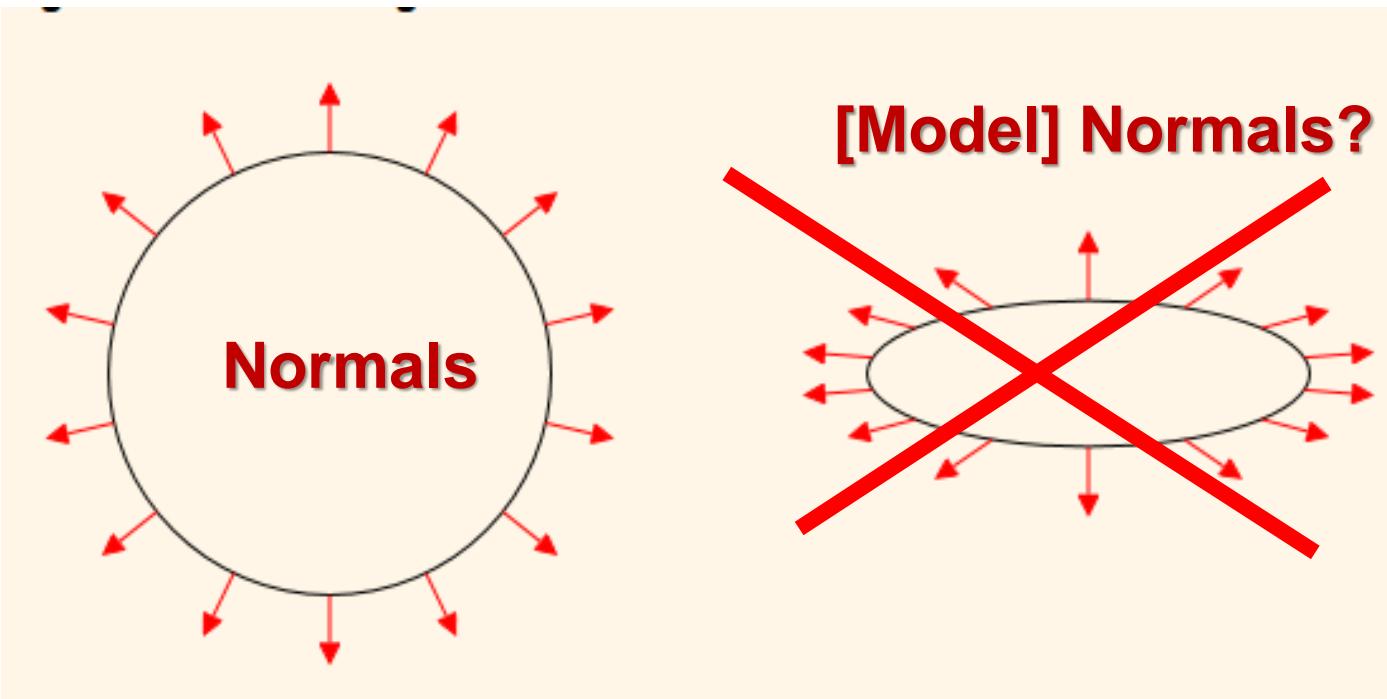
Computed per VERTEX



Phong Shading

Computed Per PIXEL

Transforming Normals



SOLUTION: use *inverse-transpose*:

Normal Matrix == (Model Matrix)^{-T}

Why? see: <http://www.lighthouse3d.com/tutorials/glsl-12-tutorial/the-normal-matrix/>

or Canvas: Project B: Reading: "How To Transform Normals"

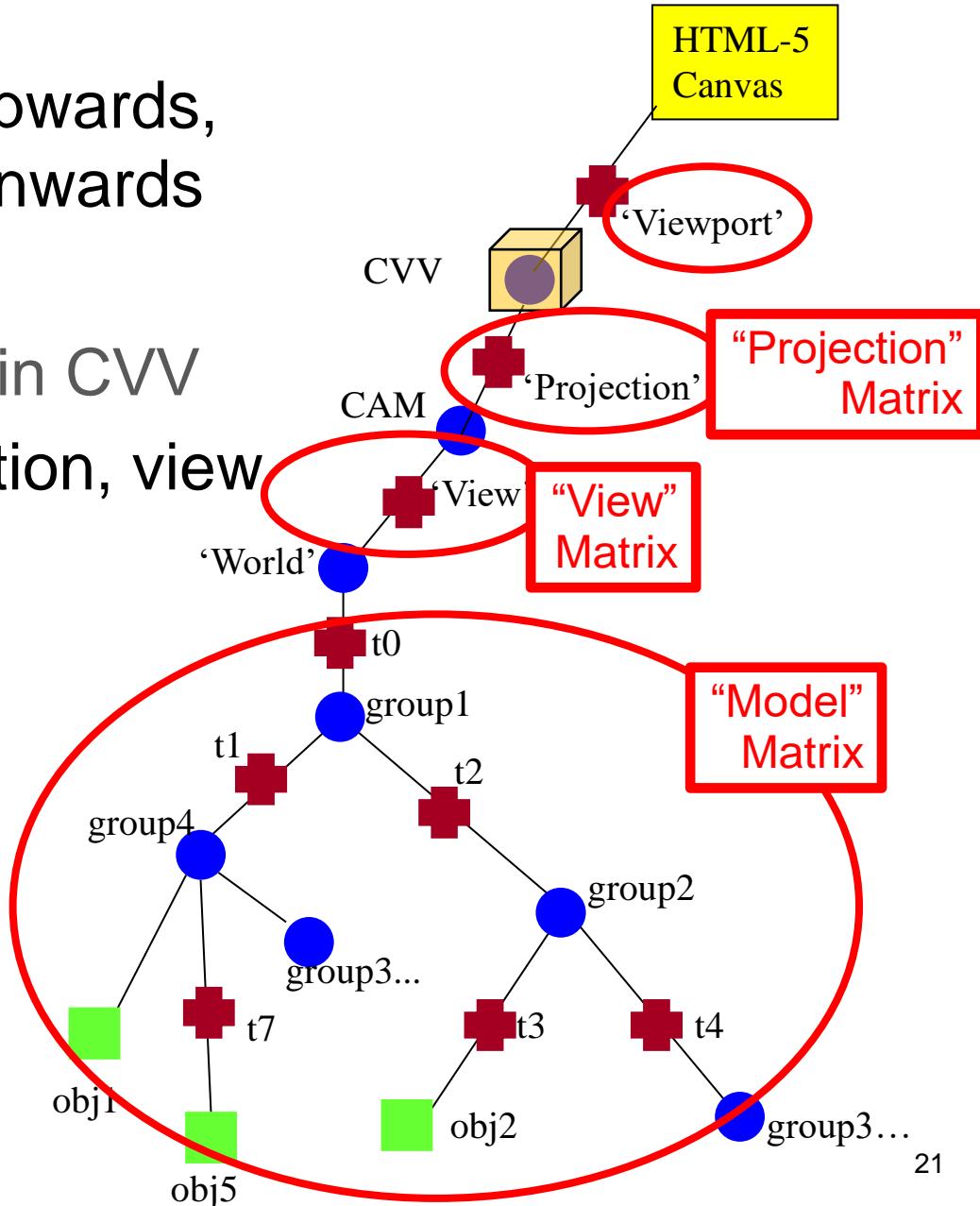
How? cuon-matrix-quat.js functions ...

WebGL: Vertex Position Pipeline

RECALL: in scene graphs,
Vertices & values move upwards,
transform calls move downwards

Project A: Draw ‘world’ directly in CVV

Project B: Add viewport, projection, view



WebGL: Vertex Position Pipeline

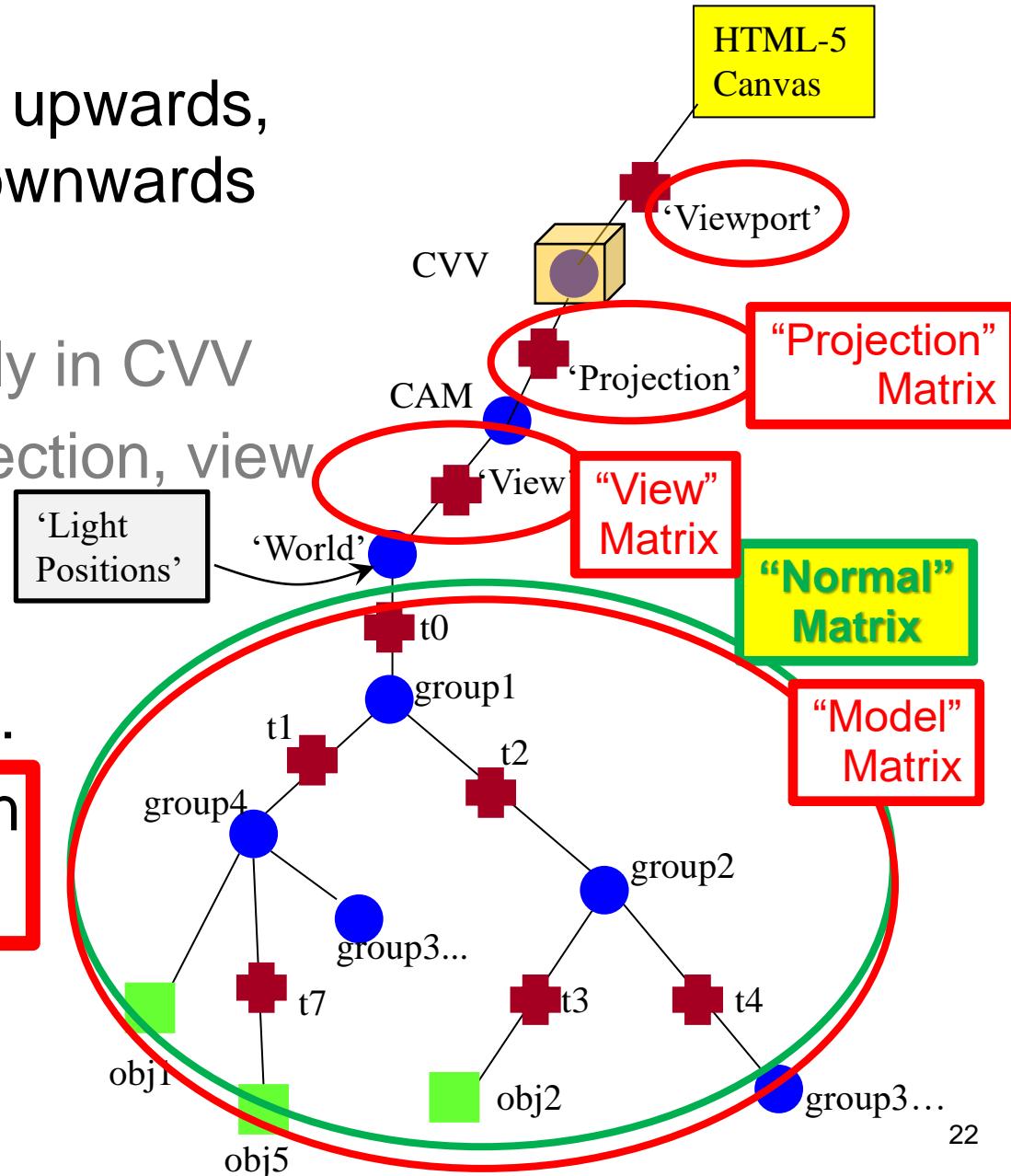
RECALL: in scene graphs,
Vertices & values move upwards,
transform calls move downwards

Project A: Draw ‘world’ directly in CVV

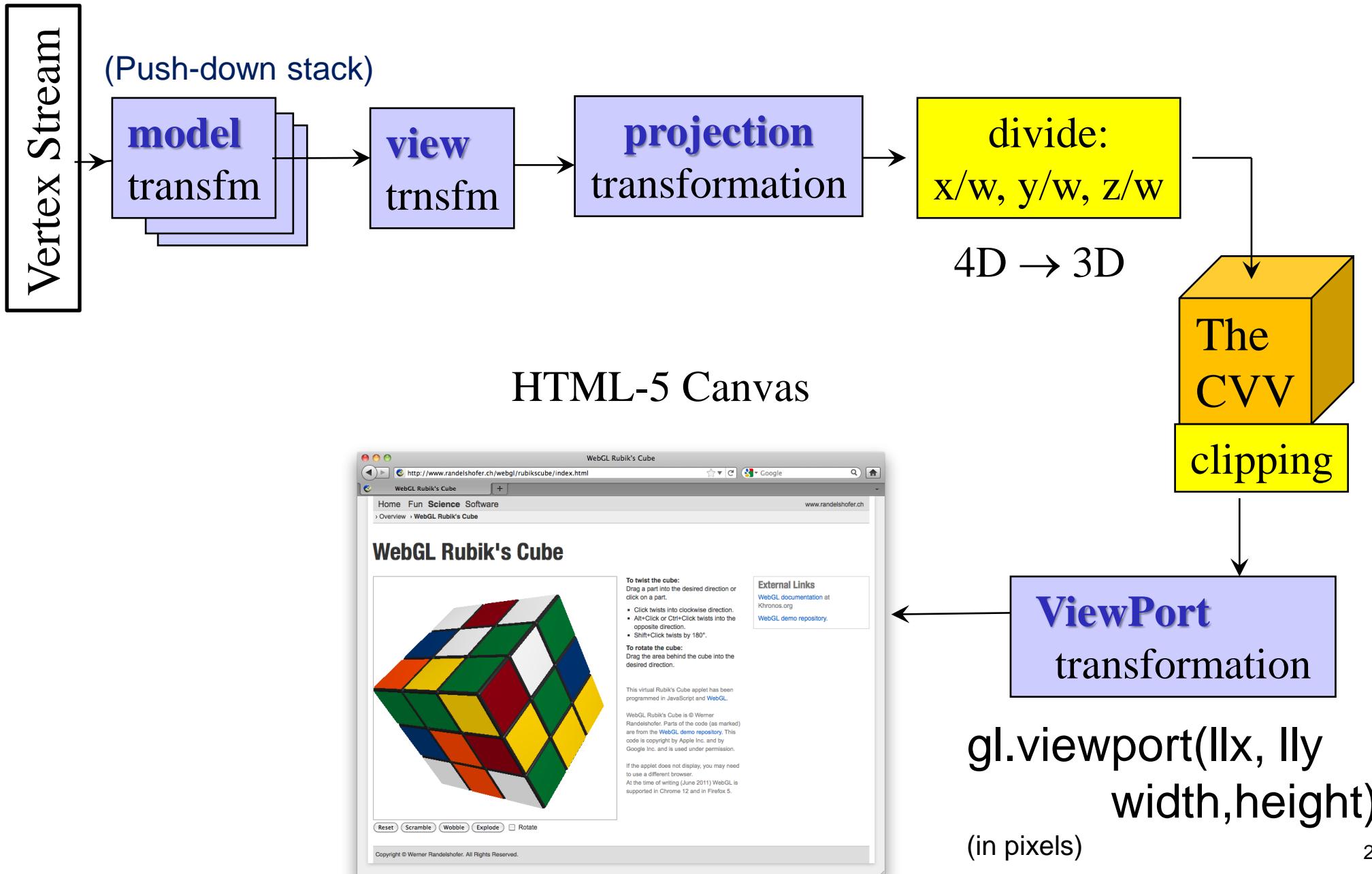
Project B: Add viewport, projection, view

Project C:

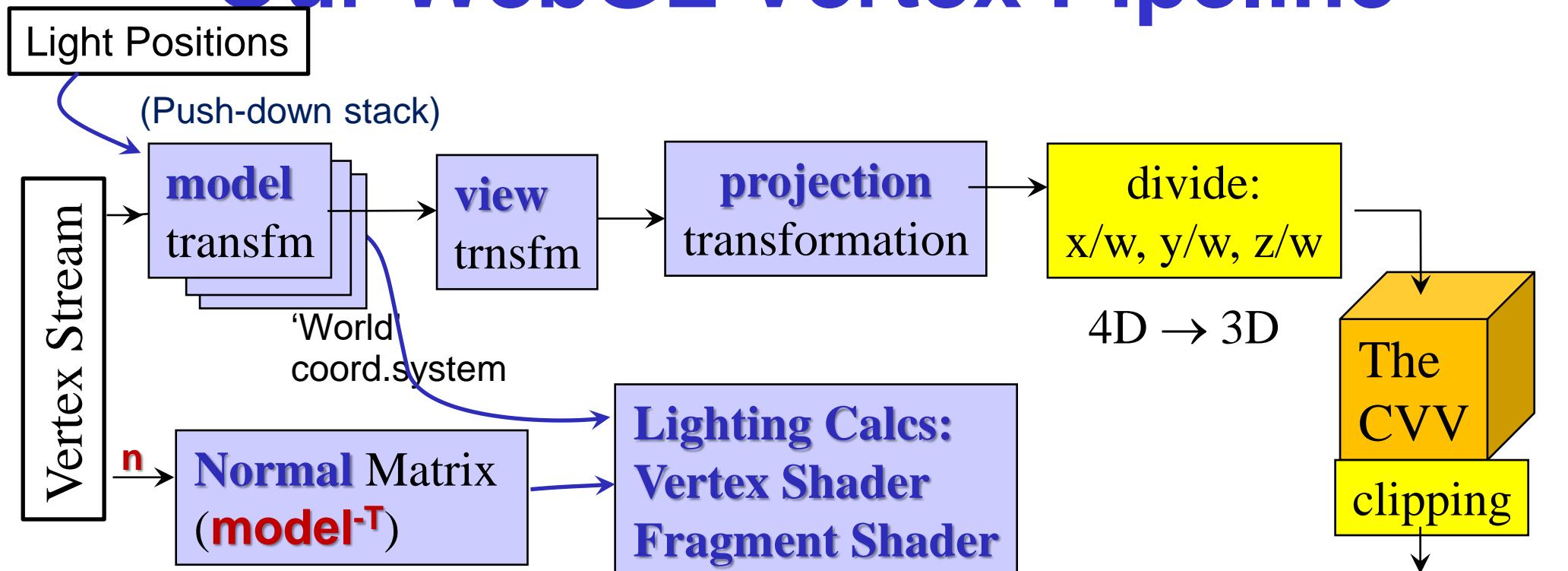
- Normals: in ‘World’ coords
normal matrix == model^{-T} ...
- Compute all lighting values in
‘World’ coordinate system



Our WebGL Vertex Pipeline

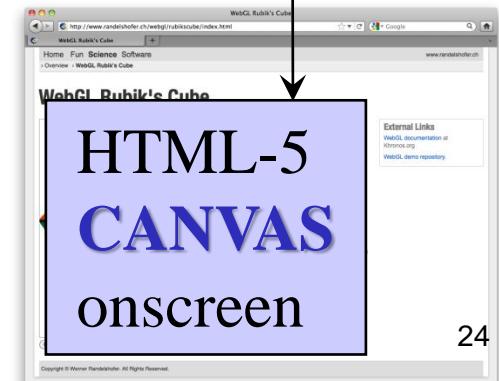


Our WebGL Vertex Pipeline



Project C:

- Compute lighting in **World** coordinate system
(e.g. find $\mathbf{N} \cdot \mathbf{L}$, find \mathbf{R} vector, etc).
- View: transforms from World to **CAM**
(How? applies '**View**' matrix in modelView)
- Normals: from Object to **World**
 $\text{normal matrix} == \text{model}^{-T} \dots$

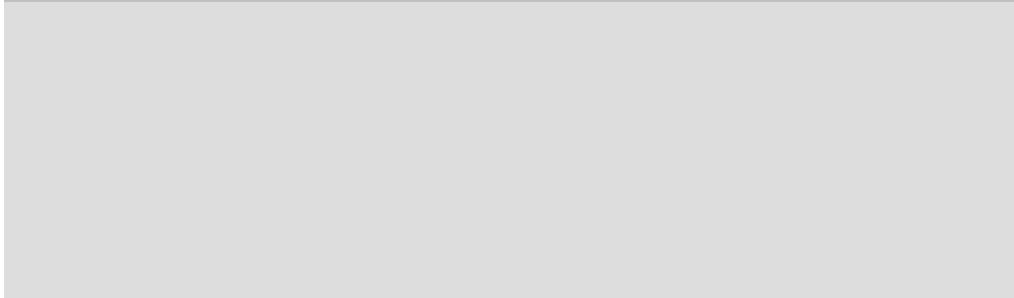


!END!

Coding for Phong Lighting & Shading



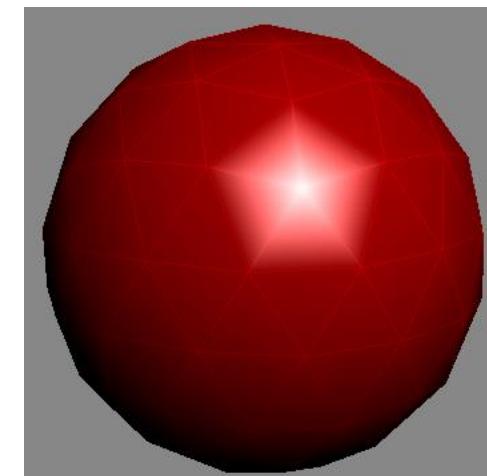
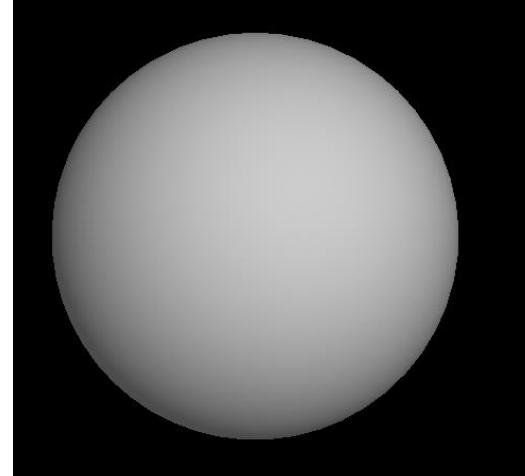
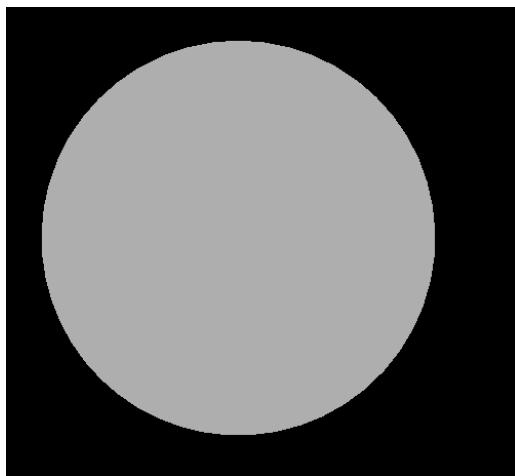
Gouraud Shading



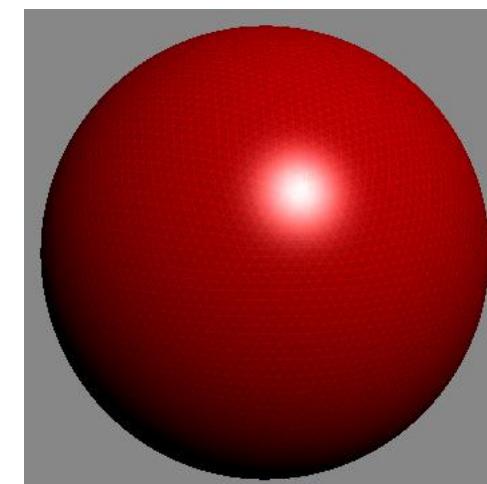
Phong Shading

Some slides modified from: David Kabala
Others from: Andries Van Damm, Brown Univ.

Cartoon → Diffuse → Specular



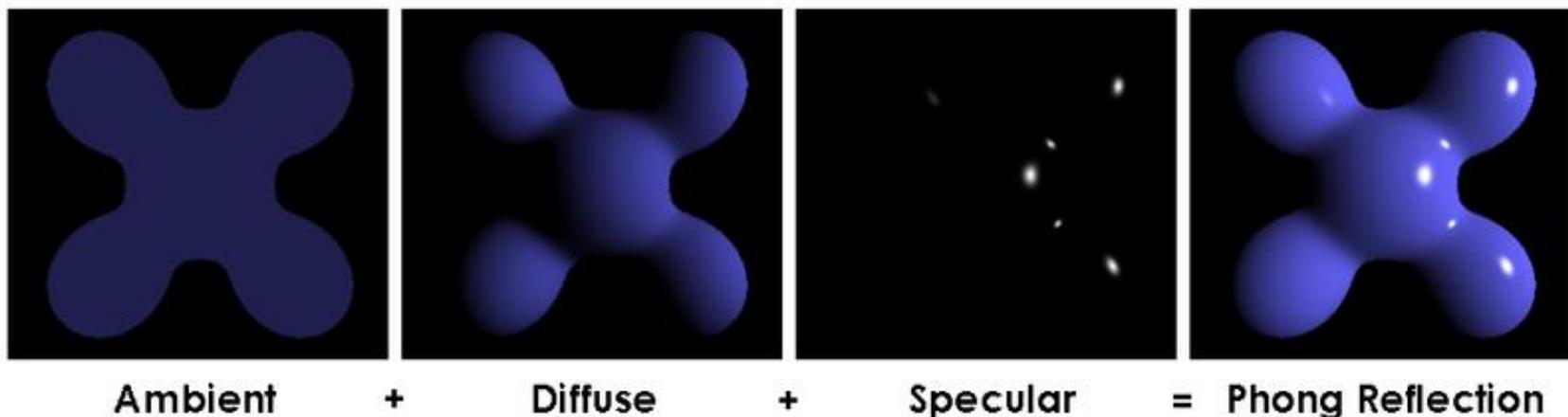
Gouraud
Shading



Phong
Shading

FAST Illumination Models: we use Phong:

- Simple model (*not physically based*)
- Splits illumination at a surface into three components
 - Ambient – Non-specific constant global lighting (hack)
 - Diffuse – Color of object under normal conditions using Lambert's model
 - Specular – Highlights on shiny objects (hack)
 - Proportional to $(R \cdot V)^\alpha$
 - larger α results in a more concentrated highlight and glossier-looking object



Lambertian Material: color set by $\cos(\theta)$ falloff only

- Just one ‘Material’ parameter (with several names):
 K_{diffuse} or ‘albedo’ or ‘diffuse reflectance’ or ...
== returned fraction of incident light intensity I_{diffuse}
== $0 \leq R, G, B \leq 1$.
- Color depends ONLY on illumination direction:
find $(N \cdot L)$ to impose $\cos(\theta)$ falloff
- Color is same for all viewing directions \forall
- ‘Diffuse Lighting’ defined by:

$$\text{On-Screen}[r,g,b] = K_{\text{diffuse}}[r,g,b] * I_{\text{diffuse}}[r,g,b] * \text{dotProduct}(N, L);$$

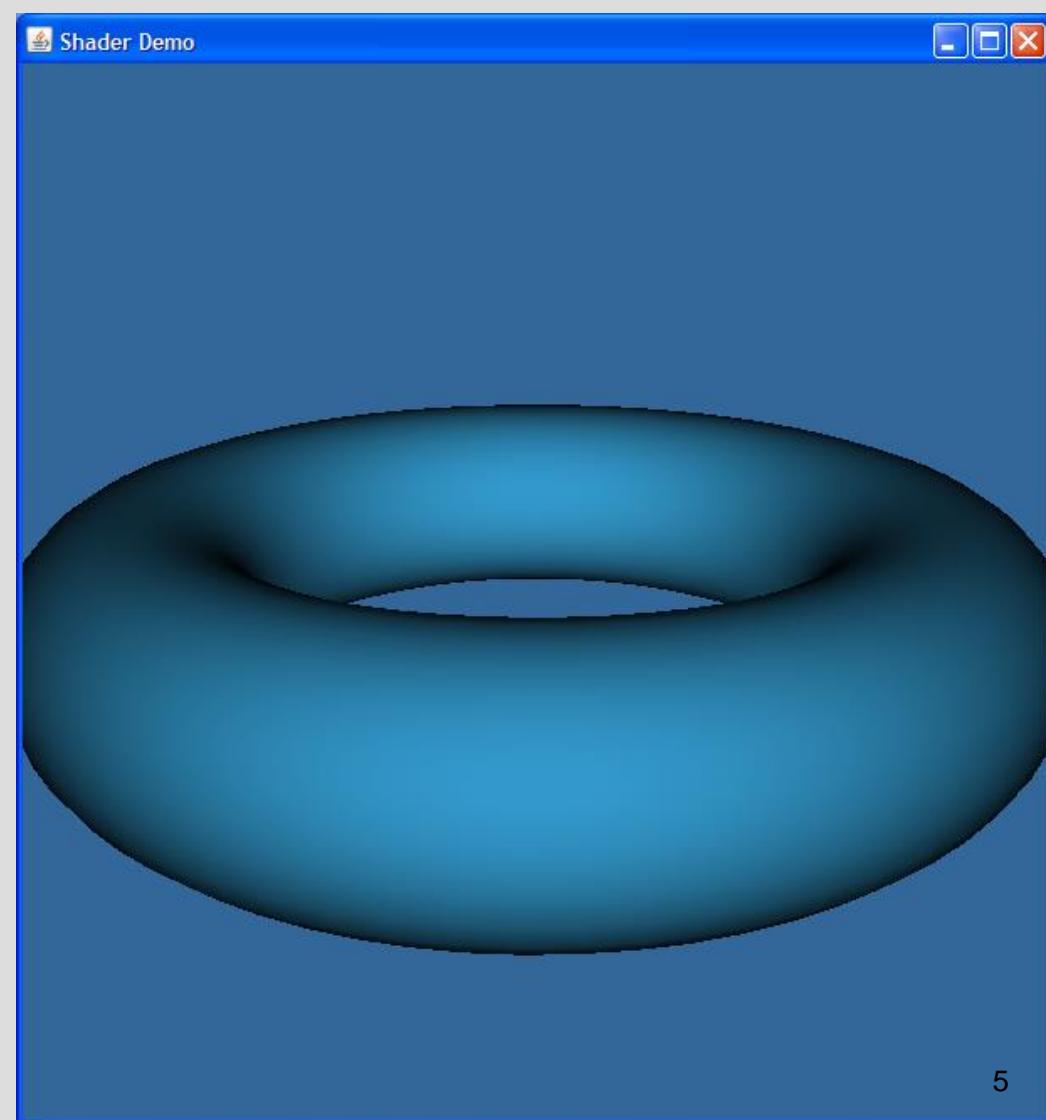
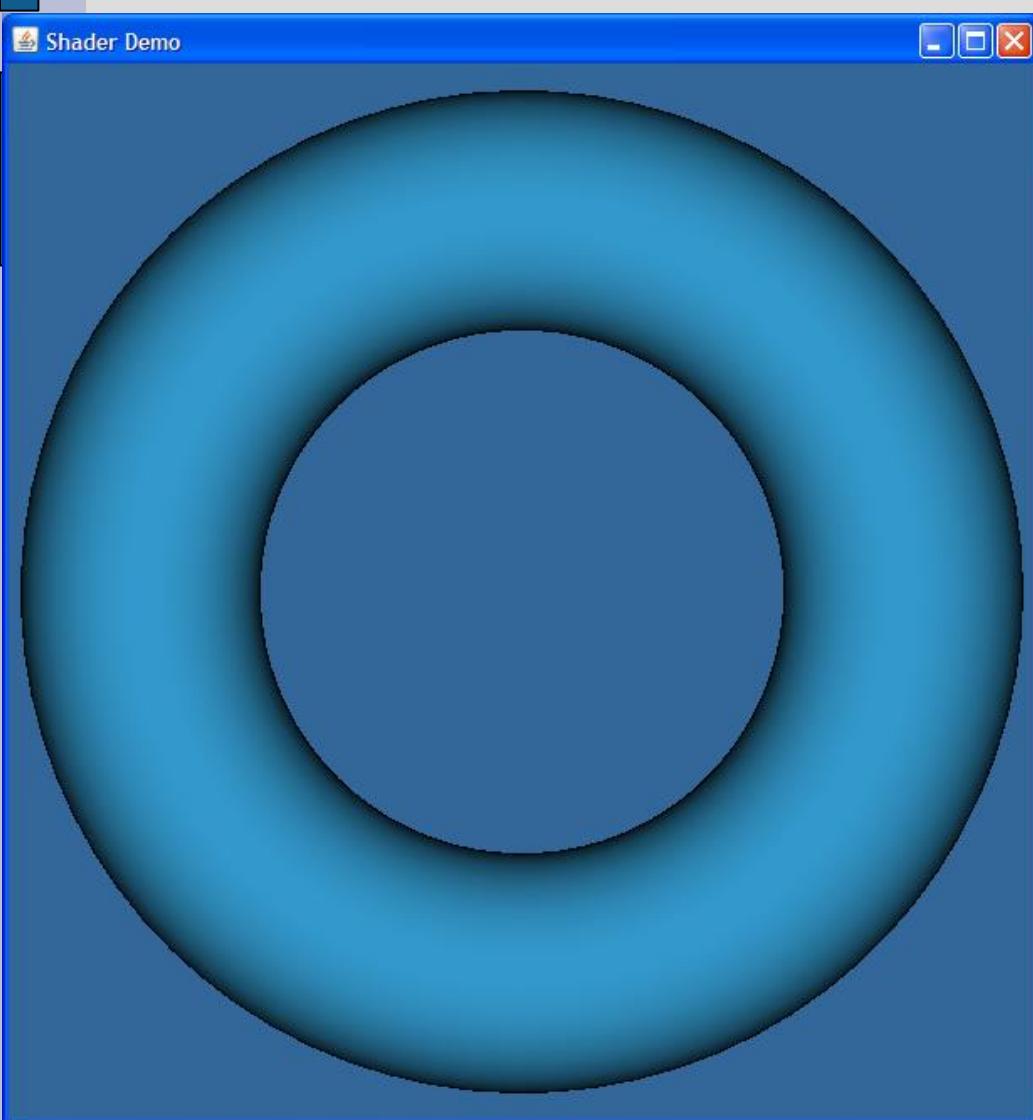
N == surface normal vector

L == light vector (from surface point to light source)

“diffuse lighting”: N·L

‘DEMO: ‘ControlDiffuseShading’ starter code

Shown: Light Source at Eye-point)



“diffuse lighting”: N.I.

‘DEMO: ‘ControlDiffuseShading’’

Shown: Light Source

For Project C:
Which coordinate system for surface normal N ?

Which coordinate system for Light direction L ?

Which coordinate system for Light Position P_L ?

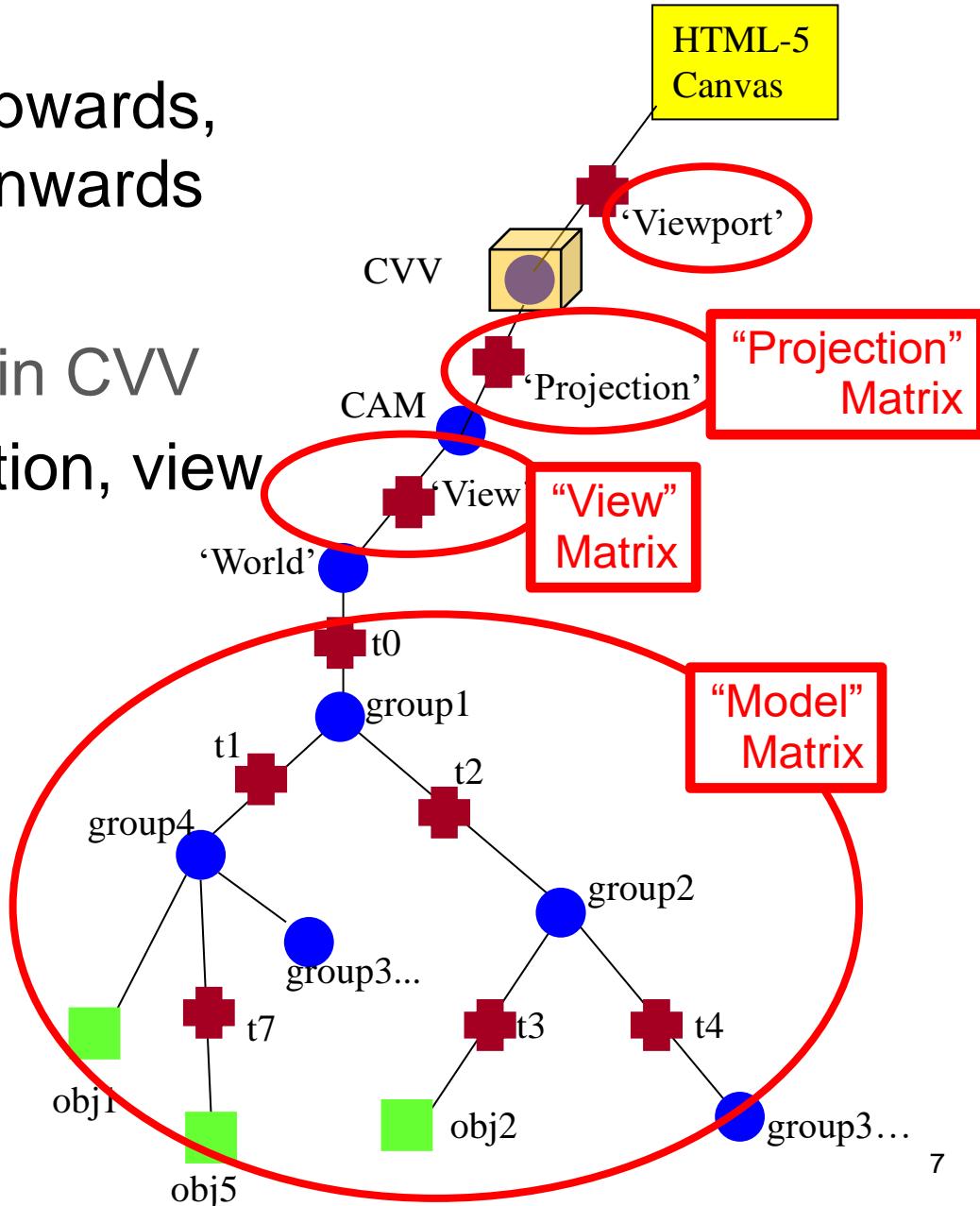
A popular answer: ‘CAM’ coords,
(after modelView transform) but others OK...

WebGL: Vertex Position Pipeline

RECALL: in scene graphs,
Vertices & values move upwards,
transform calls move downwards

Project A: Draw ‘world’ directly in CVV

Project B: Add viewport, projection, view



WebGL: Vertex Position Pipeline

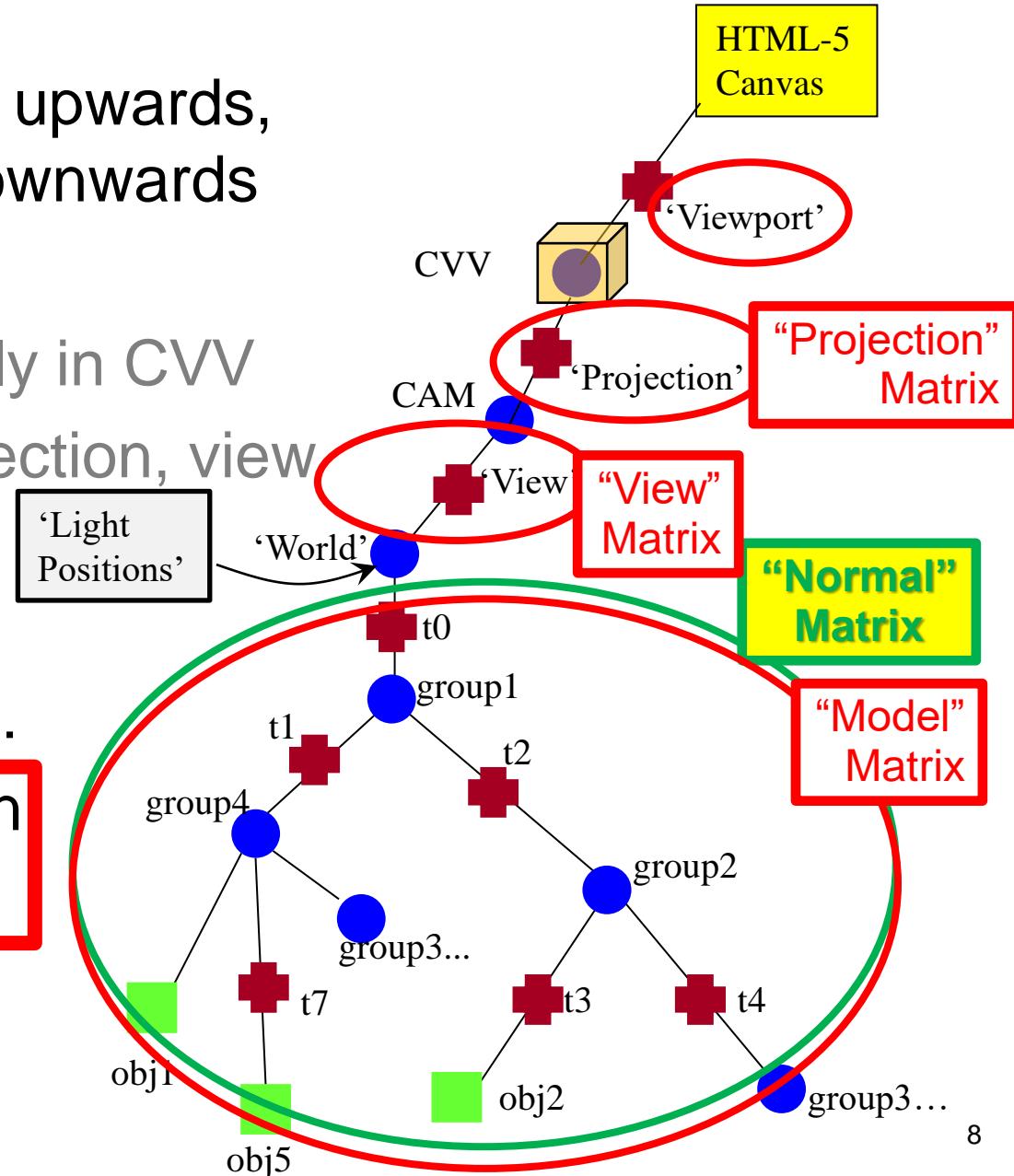
RECALL: in scene graphs,
Vertices & values move upwards,
transform calls move downwards

Project A: Draw ‘world’ directly in CVV

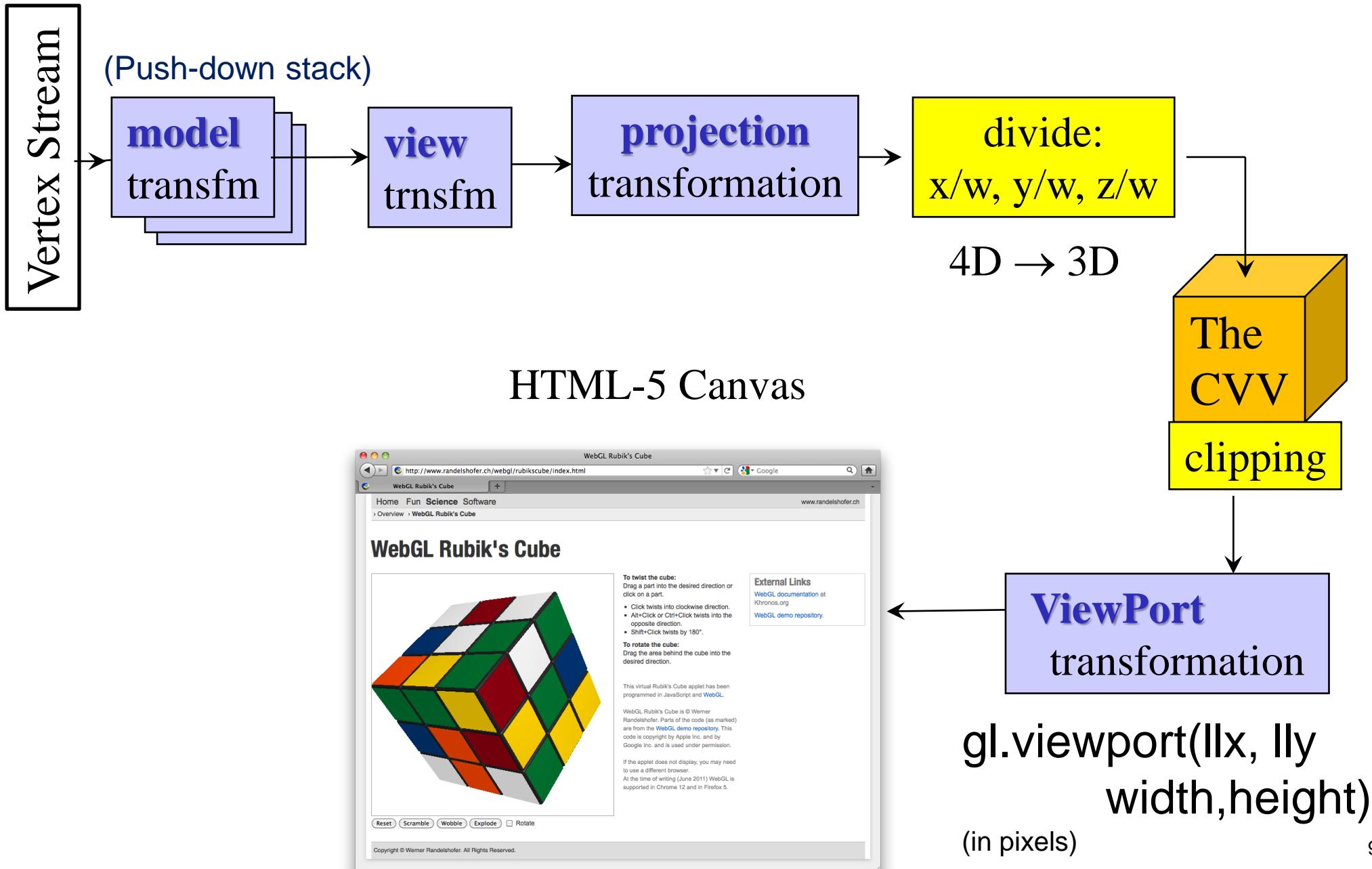
Project B: Add viewport, projection, view

Project C:

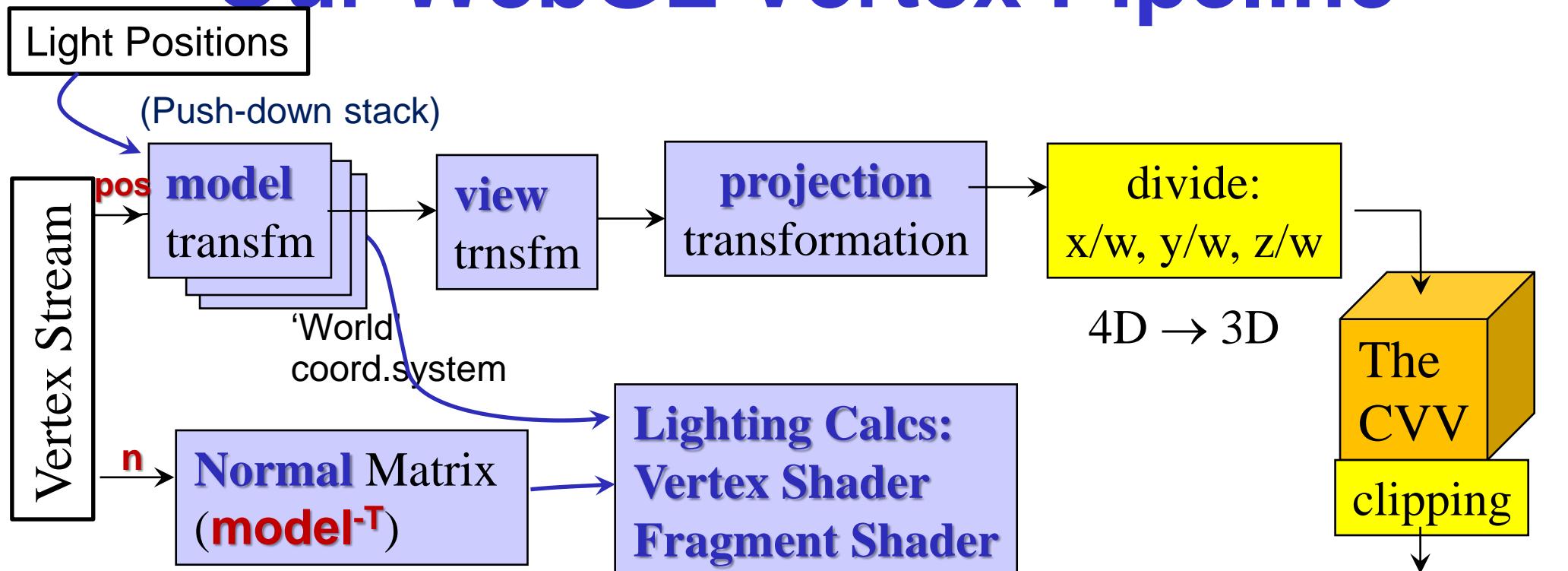
- Normals: in ‘World’ coords
normal matrix == model^{-T} ...
 - Compute all lighting values in
‘World’ coordinate system



Our WebGL Vertex Pipeline

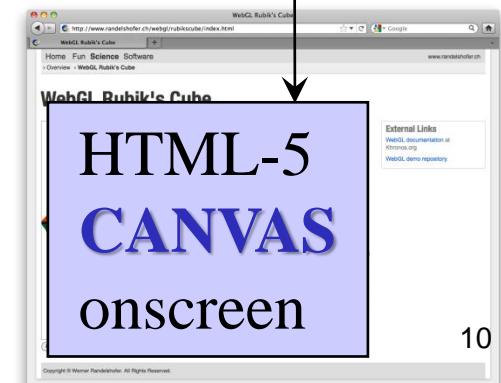


Our WebGL Vertex Pipeline

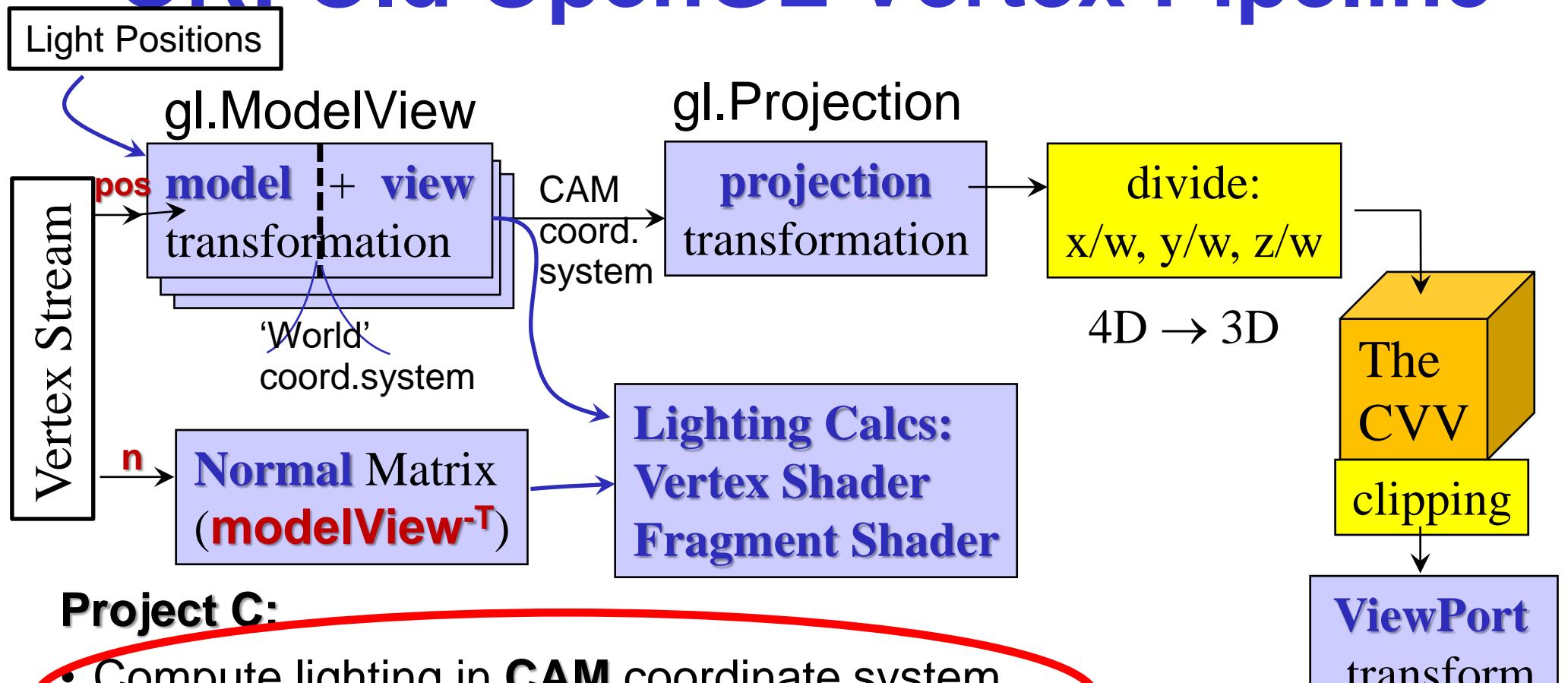


Project C:

- Compute lighting in **World** coordinate system
(e.g. find **N·L**, find **R** vector, etc).
- View: transforms from World to **CAM**
(How? applies '**View**' matrix in **modelView**)
- Normals: from Object to **World**
normal matrix == model^{-T} ...

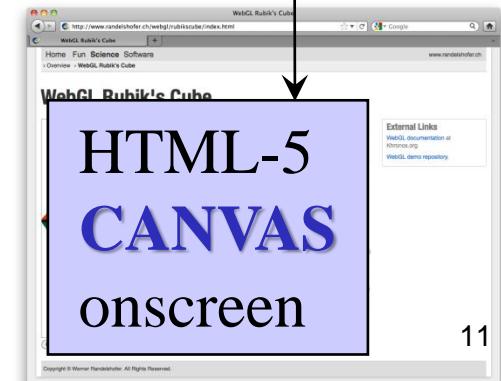


OR: Old OpenGL Vertex Pipeline



Project C:

- Compute lighting in **CAM** coordinate system
(e.g. find **N·L**, find **R** vector, etc).
- Lights: transform from World to CAM
(How? apply '**View**' matrix in **modelView**)
- Normals: from Object to **CAM**
 $\text{normal matrix} == \text{modelView}^{-T} \dots$



RECALL: Vertex Position Pipeline

RECALL: in scene graphs,
Vertices & values move upwards,
WebGL transform calls move downwards

Project A: Draw ‘world’ directly in CVV

Project B: Add viewport, projection, view

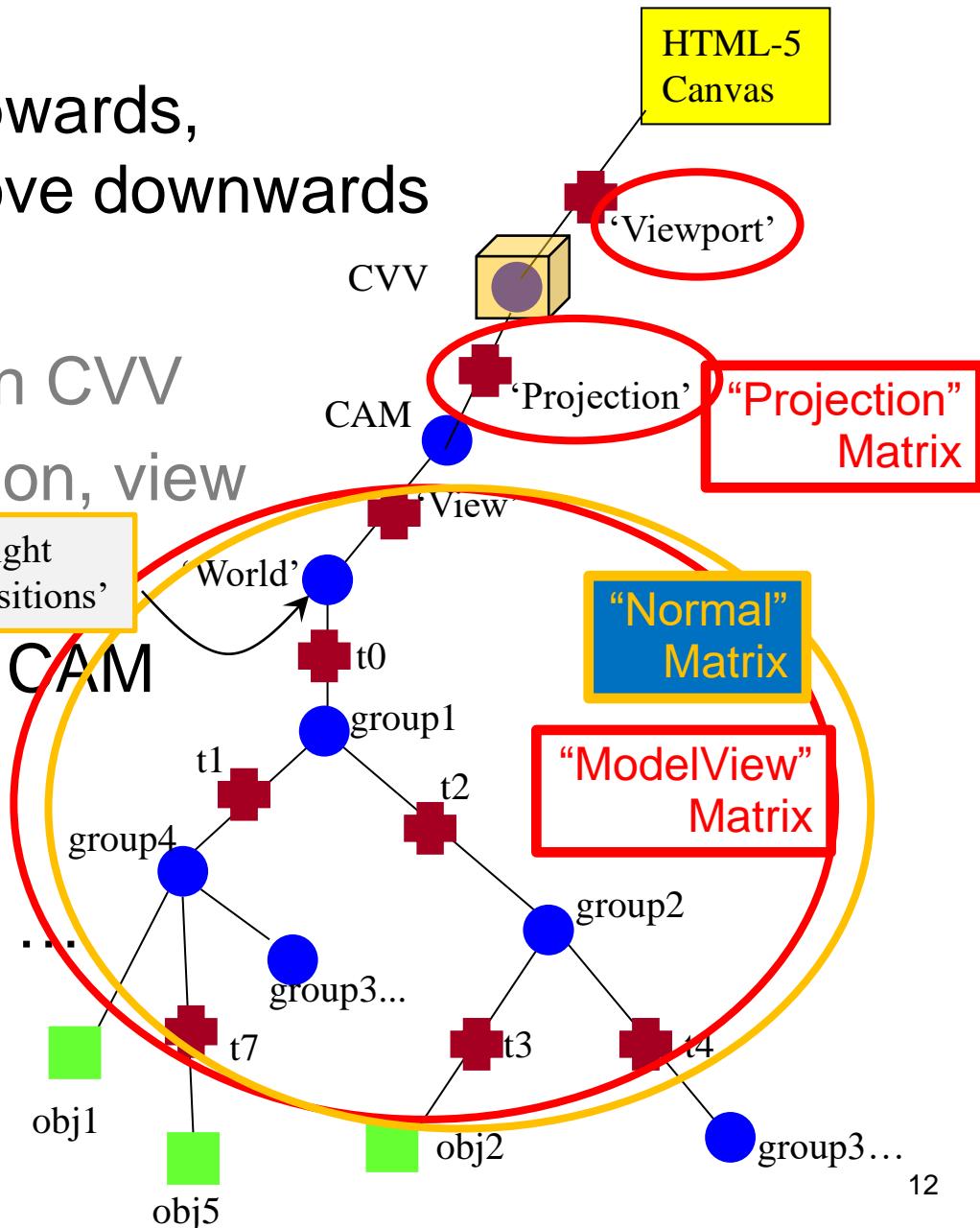
Project C:

- Lights: transform from World to CAM
(How? apply ‘view’ matrix)

- Normals: from Object to CAM

$$\text{normal matrix} == \text{modelView}^{-T}$$

- Compute all lighting values in
CAM coordinate system



CAUTION! **WAY** too easy to get lost!

Shader Writing in GLSL:

- Non-trivial vertex shader + fragment shader in GLSL
== most of the work in Project C.
- **No GLSL** debugger! No ‘printf()’ equivalent!
- **GLSL is VERY unforgiving:** mistake?
→ blank screen, + maybe a brief console err. msg.
(Don’t miss it!--keep browser’s ‘Console’ open!)
- Prime strategy:
incremental development & version control:
 - Always start with a simple program that works.
 - Improve it very slightly: test each & every tiny new step.
 - Save higher-numbered versions OFTEN: every step or so...

GLSL? Multiple shaders? Multiple VBOs?

Build your Programs Incrementally!

-
- Spend all your time making and testing tiny, quick steps.
Never leave any mistakes for later – Stay clean, clear & correct
 - Save higher-numbered copies **often**; after each new step works.
 - Puzzled for >20-30 minutes? **STOP.**
You're not going to find the bug this way; **STOP.** go back!
If last good version was 'ver027', save current as 'ver028BAD'.
Copy 'ver027' to make 'ver029'. Make smaller, simpler,
better-tested improvements. One will reveal the hidden bug!
-
- **Never** assemble a giant untested program,
and then try to debug it. Hopeless: you may never fix it all!
 - ***Big broken programs can hide big mistakes well.
(and the search to find them creates NEW problems!)***
 - ***Far faster to make a long series of tiny, tested improvements to
a really dumb simple program that does work than to struggle
fixing an 'almost finished' complicated program that doesn't.***

GLSL? Multiple shaders? Multiple VBOs?

Version Control is Crucial.

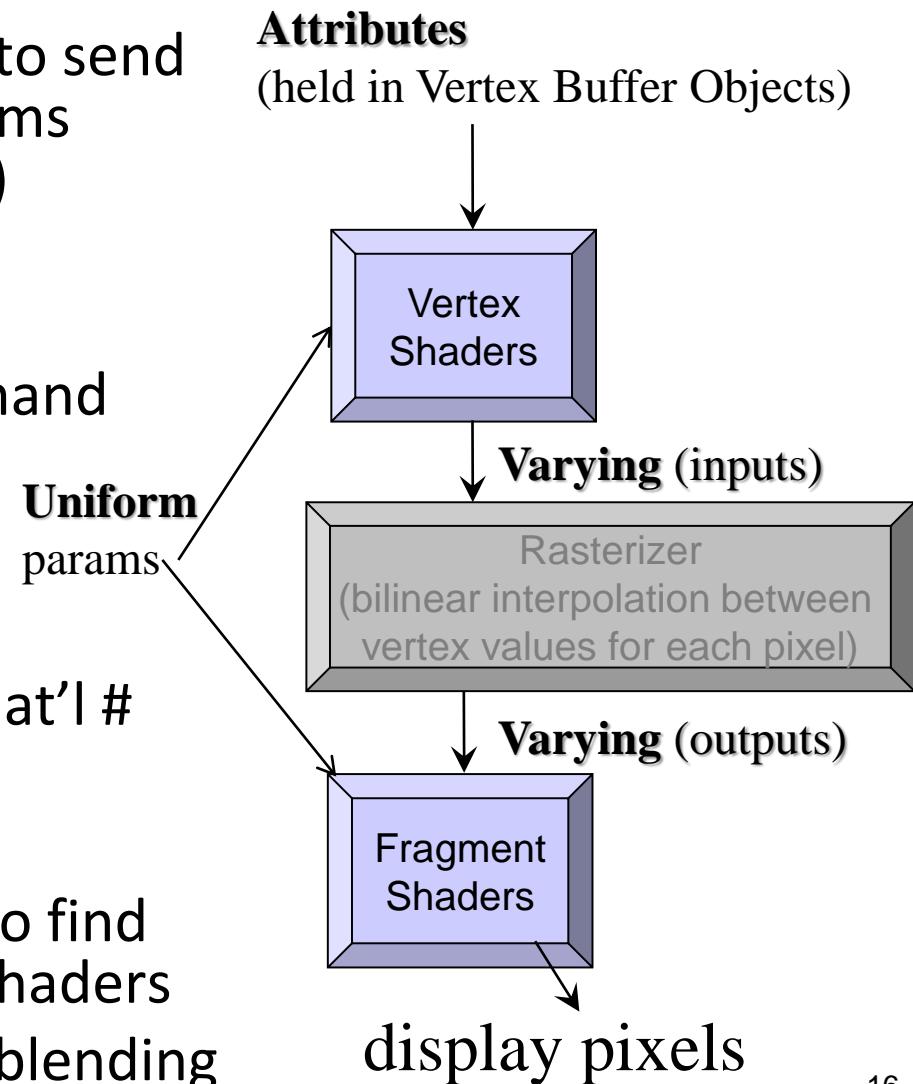
- ***Comments first –***
*Find the big problems before you write code.
Think through the entire problem. Write your INTENT,
your current plans. Start to finish -- everything.*
- ***Start with some simple code that you **KNOW** is correct.***
*correctly. Doesn't matter how simple – even ‘hello world’ is
good. Save that as your first version. Make sure it still works.
Make a higher-numbered copy.*
- ***Never modify your earlier versions*** –
*they're your record of
current thinking, including mistakes, so you can find out
'what was I thinking?!?!' later.*
- ***Save copious versions;*** *memory is trivially cheap, but your
time is not. I routinely write 10-12 progressively better
versions for Project- C-sized programs*

RECALL:

How Shader Programs Communicate

Only 3 ways for your JavaScript code to send data **into** the GPU's shader programs
(and no way to retrieve that data!)

- **Uniform** parameters
 - Set before each drawing command
 - Ex: modelView Matrix
- **Attribute** parameters
 - Set per vertex
 - Ex: position, surface normal, mat'l #
- **Varying** parameters
 - Passed from Vertex Shaders to rasterizer, which interpolates to find per-pixel values for fragment shaders
 - Ex: triangle with smooth color blending



Part 1: Gouraud Shading (“Goorr-Rowe”)



one lighting calc per **Vertex**

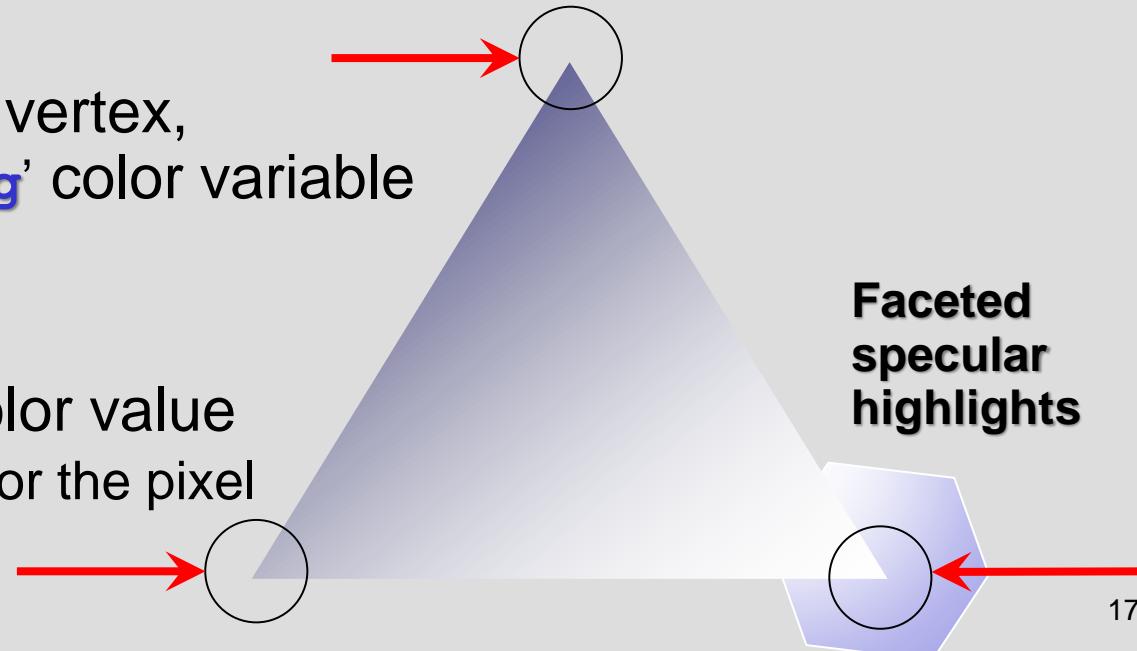
- For each **vertex**, (in GLSL *vertex* shader)
compute on-screen RGB color
- For each **pixel**, (in GLSL *fragment* shader)
bilinearly interpolate on-screen RGB color:

Vertex Shader

computes Phong lighting for vertex,
export result as one ‘**varying**’ color variable

Fragment Shader

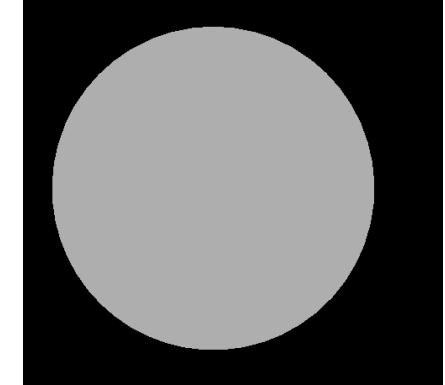
gets a rasterized **varying** color value
which sets **gl_fragColor** for the pixel



Gouraud Step-by-Step Goals:

1) Surface Normal Attributes

- Set dark-color, **non-black background** (e.g. (0.0,0.2,0.1) why? surfaces won't vanish)
- Just one object: unit sphere at world origin
- Sphere fills screen: put camera on world +z axis
- Add surface normal **attribute** to sphere vertices
- **TEST:** do your surface-normal attribute values actually arrive at your Vertex Shader?
 - How? In Vertex Shader, compute vertex color as $(\text{normal vector} + 1,1,1)/2$
 - send that color to Fragment Shaders as a 'varying' variable to interpolate colors between vertices (Gouraud shading!)
 - In Fragment Shader, use that 'varying' variable to set the final color of the pixel in `gl_FragColor`

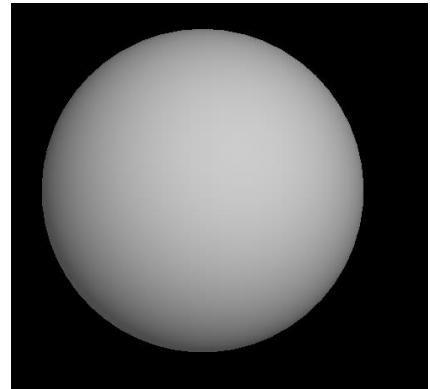


Gouraud Step-by-Step Goals:

2) ADD Light Source Uniforms

Add ‘**uniform**’ vars for one light source:

- At first, just the light-source position
(convert to world coords in Javascript:
don’t repeat the same transform for each vertex!)
- Vertex Shader: transform normals to World coords,
compute $(N \cdot L)$, use result as color.
- Then add other light-source uniforms: I_a , I_d , I_s , etc.
(RGB ambient, diffuse, specular illum; all [0-1])



TEST: do your light-source uniform values actually arrive at your Vertex Shader?

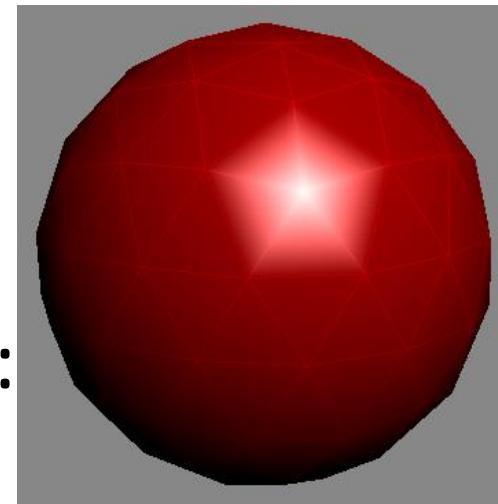
- Create **one** uniform, test it, then the next uniform...
- **How?** Let the newest uniform set a vertex color

Gouraud Step-by-Step Goals:

3) ADD Materials Uniforms

Add ‘uniform’ vars for just one material:

- Start with diffuse reflectance K_d ;
- Vertex Shader: compute diffuse color:
 $K_d * I_d * Att * (N \cdot L)$ (note entirely black shadows)
- Then add weak ambient lighting & reflectance
(lightens the shadows: no longer entirely black)
- Then add specular term:
 - initially, try $S_e = 20, K_s = I_s = (0.9, 0.9, 0.9)$
 - GLSL-ES: use the ‘`pow()`’ and ‘`reflect()`’ functions
 - Note the faceted, hexagonal specular highlights!



TEST: do your Materials uniform values
actually arrive at your Vertex Shader?

- Create **one** uniform, test it, then the next uniform...
- How? Let the newest uniform set the vertex color

Part 2: Phong Shading

One lighting calc per **Pixel**

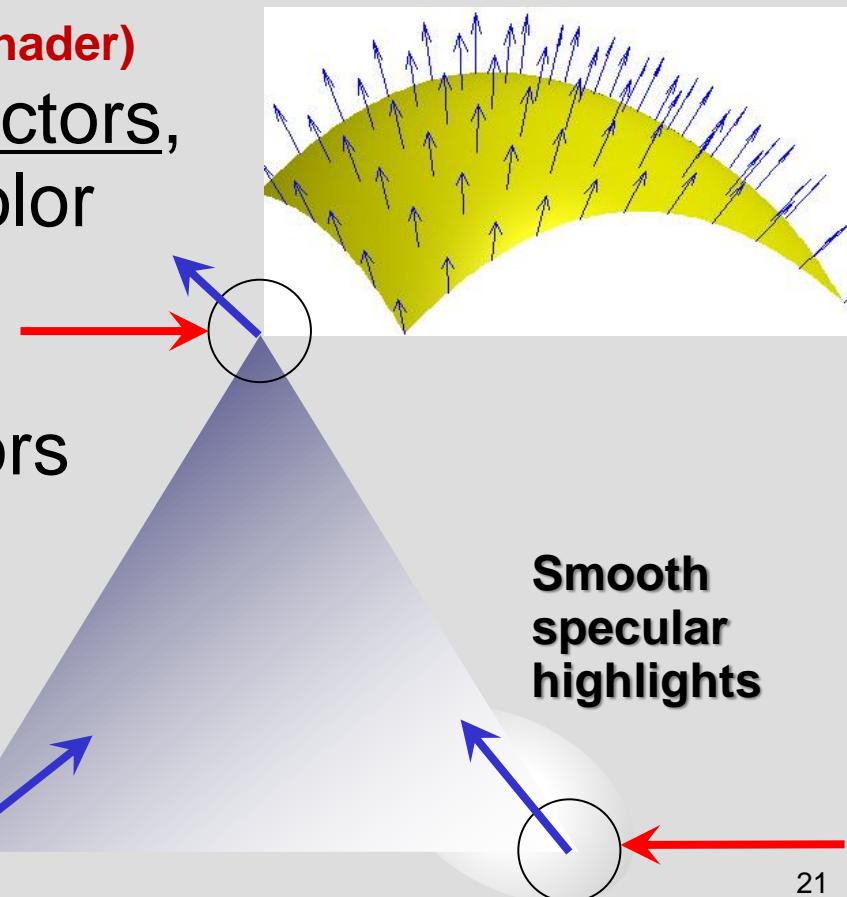
- For each **vertex**, (in GLSL *vertex* shader)
compute on-screen RGB color
- For each **pixel**, (in GLSL *fragment* shader)
bilinearly **interpolate vectors**,
compute lighting, set color

Vertex Shader

computes '**varying**' lighting vectors
(from vertex **attribute**s
&/or **uniform** vars)

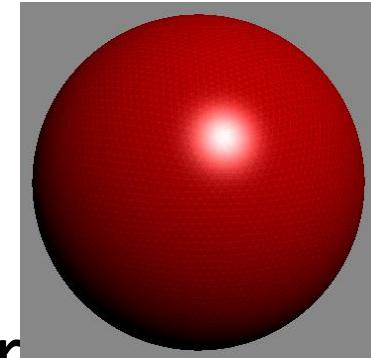
Fragment Shader

gets rasterized **varying** vectors
computes **gl_fragColor**



Phong *Shading* Step-by-Step Goals: Per-pixel Vectors

- Add new ‘**varying**’ vars to Vertex Shader to interpolate all vectors for per-pixel lighting
- **Move** lighting calcs to Frag Shader. Diffuse 1st:
 - Start by adding a ‘varying’ var to interpolate the surface normal N, and the vertex position P.
 - In Fragment Shader, find unit light direction vec L (light-position uniform - vertex position varying)
 - In Fragment Shader, compute (N·L) to set color.
- Then specular:
 - Compute reflected direction R or half-vector H (test it – use R or H to set sphere color, move light)
 - Complete the specular term: compare results to earlier Gouraud-shaded version – they should look similar, but Phong Shading highlights are round with no faceting: looks perfectly smooth, flawless



Phong or Blinn-Phong Lighting

Step 1: Find Scene Vectors

To find *On-Screen RGB Color* at point **P (start)**:



1) Find all 3D scene vectors first:

a) Light Vector **L**:

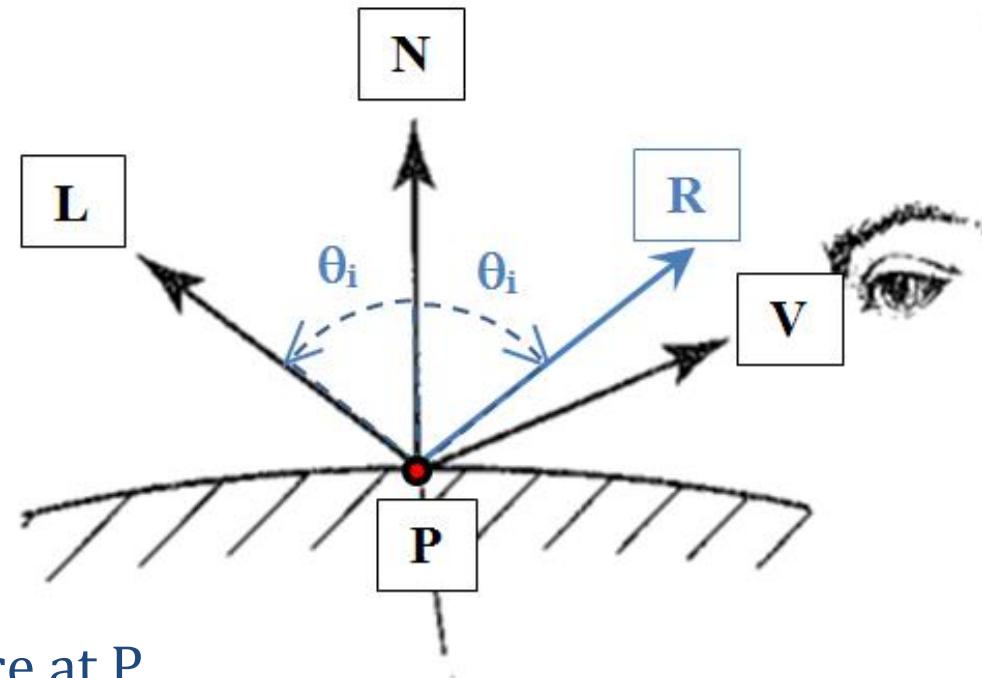
unit vector towards light source

b) Normal Vector **N**:

unit vector perpendicular to surface at P

c) View Vector **V**:

unit vector towards camera eye-point

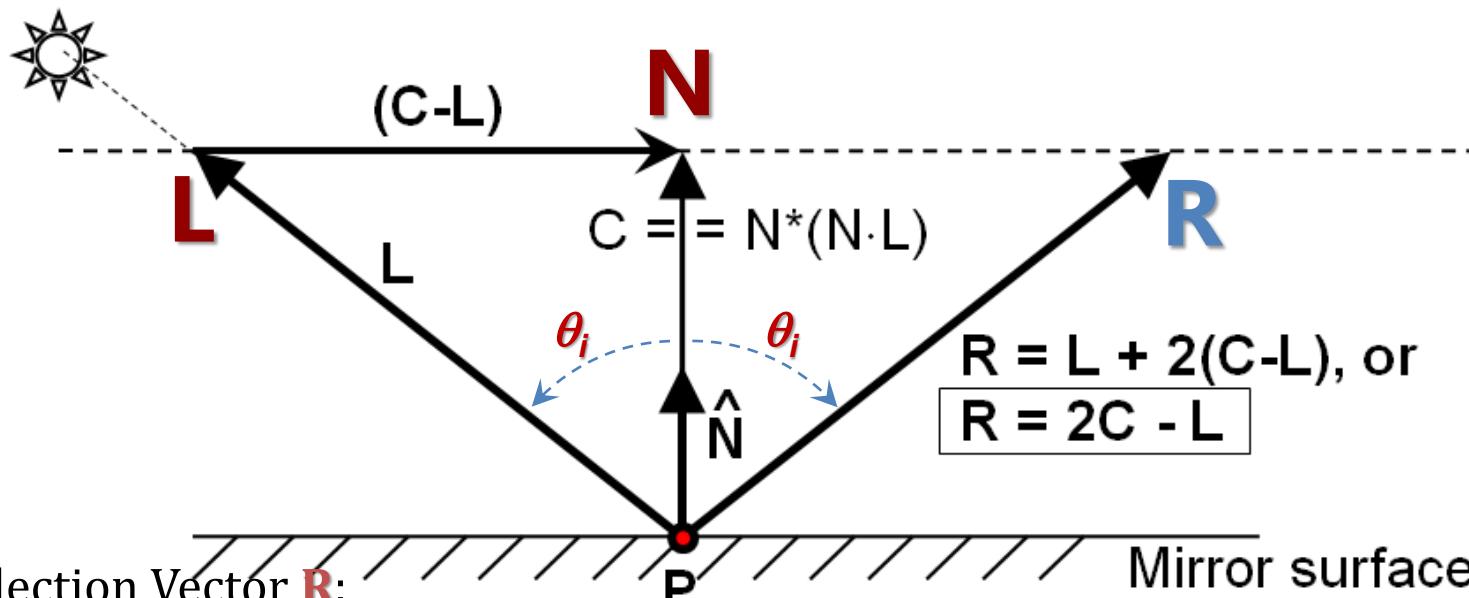


On to step 2: how do we find the Reflected-light Vector **R**?

Phong Lighting ONLY:

Step 2: Find reflection Vector R

To find *On-Screen RGB Color*
at point **P** (*cont'd*):



2) COMPUTE the Light Reflection Vector **R**:

- Given unit light vector \mathbf{L} , find lengthened normal \mathbf{C}
$$\mathbf{C} = \mathbf{N} (\mathbf{L} \cdot \mathbf{N})$$
- In diagram, if we add vector $2^*(\mathbf{C}-\mathbf{L})$ to \mathbf{L} vector we get \mathbf{R} vector. Simplify:
$$\mathbf{R} = 2\mathbf{C} - \mathbf{L}$$
- If \mathbf{N} is a unit-length vector, then \mathbf{R} vector length matches \mathbf{L} vector length.
Result: unit-length \mathbf{R} vector

GLSL-ES → See built-in '`reflect()`' function

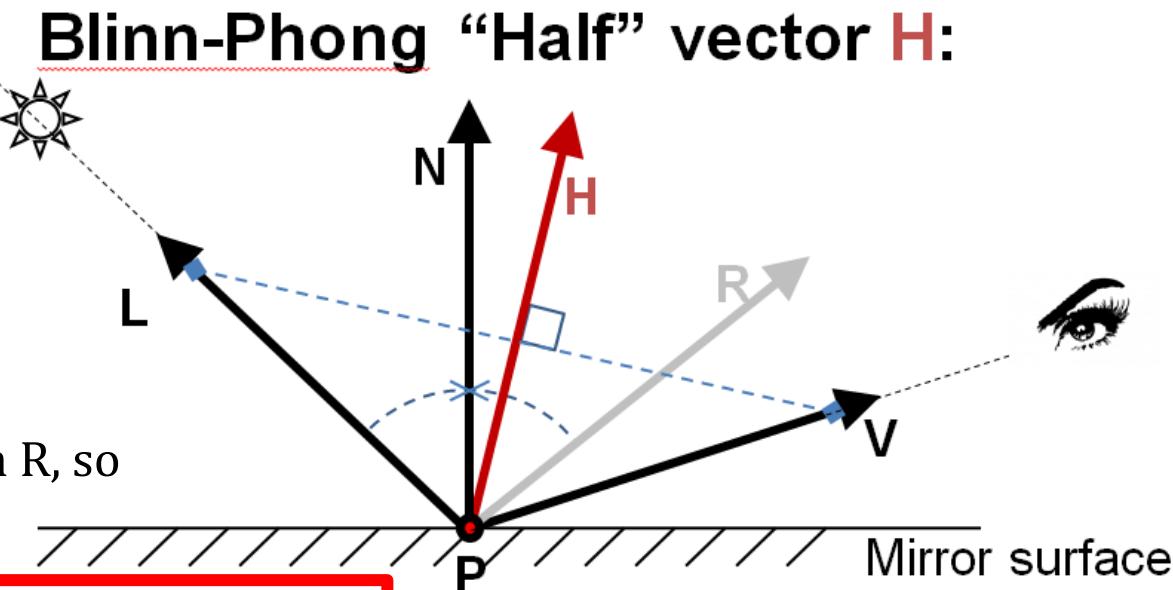
Blinn-Phong Lighting ONLY: Fast (but Approximate) Specular Reflection

- ▶ Skip reflection-vector R calculation!
- ▶ Instead, define the 'half angle' H :

$$H = \frac{L+V}{|L+V|} \quad (\text{careful! unit-length } L, V)$$

'halfway' between light and eye.

- ▶ $H=N$ when eye aligns with reflection R , so



- ▶ Replace Phong specular term $(R \cdot V)^{Se}$ with Phong-Blinn specular term $(N \cdot H)^{Se}$
- ▶ CAREFUL! must have equal-length L,V (e.g. normalize both L and V first)
- ▶ Should we use Phong or Blinn-Phong?
 - ▶ Blinn-Phong slightly simpler, faster to compute
 - ▶ slight difference on-screen, but hard to see
 - ▶ implemented in original OpenGL for simplicity C.

BOTH REQUIRED for PROJECT C!

Phong or Blinn-Phong Lighting

Step 3: Gather Light & Material Data

To find *On-Screen RGB Color*
at point **P** (*cont'd*):

3) For each light source, gather:

- ▶ RGB triplet for **Ambient** Illumination **I_a** $0 \leq I_{ar}, I_{ag}, I_{ab} \leq 1$
- ▶ RGB triplet for **Diffuse** Illumination **I_d** $0 \leq I_{dr}, I_{dg}, I_{db} \leq 1$
- ▶ RGB triplet for **Specular** Illumination **I_s** $0 \leq I_{sr}, I_{sg}, I_{sb} \leq 1$

For each surface material, gather:

- ▶ RGB triplet for **Ambient** Reflectance **K_a** $0 \leq K_{ar}, K_{ag}, K_{ab} \leq 1$
- ▶ RGB triplet for **Diffuse** Reflectance **K_d** $0 \leq K_{dr}, K_{dg}, K_{db} \leq 1$
- ▶ RGB triplet for **Specular** Reflectance **K_s** $0 \leq K_{sr}, K_{sg}, K_{sb} \leq 1$
- ▶ RGB triplet for **Emissive** term(often zero) **K_e** $0 \leq K_{er}, K_{eg}, K_{eb} \leq 1$
- ▶ Scalar 'shinyness' or 'specular exponent' term **S_e** $1 \leq S_e \leq \sim 100$



Phong Lighting

Step 4: Sum of Light Amounts

To find *On-Screen RGB Color*
at point **P** (*cont'd*):

sum of each kind of light at **P**:

$$\text{Phong Lighting} = \text{Ambient} + \text{Diffuse} + \text{Specular} + \text{Emissive}$$

SUMMED for all light sources

4) For the i-th light source, find:

$$\begin{aligned} \text{RGB} &= \mathbf{K}_e + \\ &\mathbf{I}_a * \mathbf{K}_a + \\ &\mathbf{I}_d * \mathbf{K}_d * \mathbf{A}\mathbf{t}\mathbf{t} * \max(0, (\mathbf{N} \cdot \mathbf{L})) \\ &\mathbf{I}_s * \mathbf{K}_s * \mathbf{A}\mathbf{t}\mathbf{t} * (\max(0, \mathbf{R} \cdot \mathbf{V}))^{\mathbf{s}\mathbf{e}}, \end{aligned} \quad \begin{aligned} &\text{// 'emissive' material; it glows!} \\ &\text{// ambient light * ambient reflectance} \\ &\text{// diffuse light * diffuse reflectance} \\ &\text{// specular light * specular reflectance} \\ &\text{// (or for Blinn-Phong: } \mathbf{I}_s * \mathbf{K}_s * \mathbf{A}\mathbf{t}\mathbf{t} * (\max(0, \mathbf{N} \cdot \mathbf{H}))^{\mathbf{s}\mathbf{e}} \end{aligned}$$

- ▶ Distance Attenuation scalar: $0 \leq \mathbf{A}\mathbf{t}\mathbf{t} \leq 1$
 - ▶ Fast, OK-looking default value: $\mathbf{A}\mathbf{t}\mathbf{t} = 1.0$
 - ▶ Physically correct value: $\mathbf{A}\mathbf{t}\mathbf{t}(\mathbf{d}) = 1 / (\text{distance to light})^2$ (too dark too fast!)
 - ▶ Faster, Nice-looking 'Hack': $\mathbf{A}\mathbf{t}\mathbf{t}(\mathbf{d}) = 1 / (\text{distance to light})$
 - ▶ OpenGL compromise: $\mathbf{A}\mathbf{t}\mathbf{t}(\mathbf{d}) = \min(1, 1 / (c_1 + c_2 * d + c_3 * d^2))$
- ▶ 'Shinyness' or 'specular exponent' $1 \leq \mathbf{s}\mathbf{e} \leq \sim 200$ (large for sharp, small highlights)



EXAMPLE: Phong-Shaded Diffuse-only lighting...

```
// Vertex Shader

attribute vec4 a_vertPos;
attribute vec4 a_normalVec;
uniform mat4 u_mvpMat;
uniform mat4 u_normalMat;
uniform vec3 u_lightPos;

varying vec3 v_norm;
varying vec3 v_toLight;

void main()
{
    gl_Position = u_modelviewMat * a_vertPos;
    v_norm = u_normalMat * a_normalVec;
    v_toLight =
        vec3(a_lightPos - a_vertPos);
}
```

```
// Fragment Shader

varying vec3 v_norm;
varying vec3 v_toLight;

void main()
{
    const vec3 DiffColr =
        vec3(0.2, 0.6, 0.8);
    float diff = clamp(
        dot(normalize(v_norm),
            normalize(v_toLight))
        ,
        0.0, 1.0); // stay in [0,1]

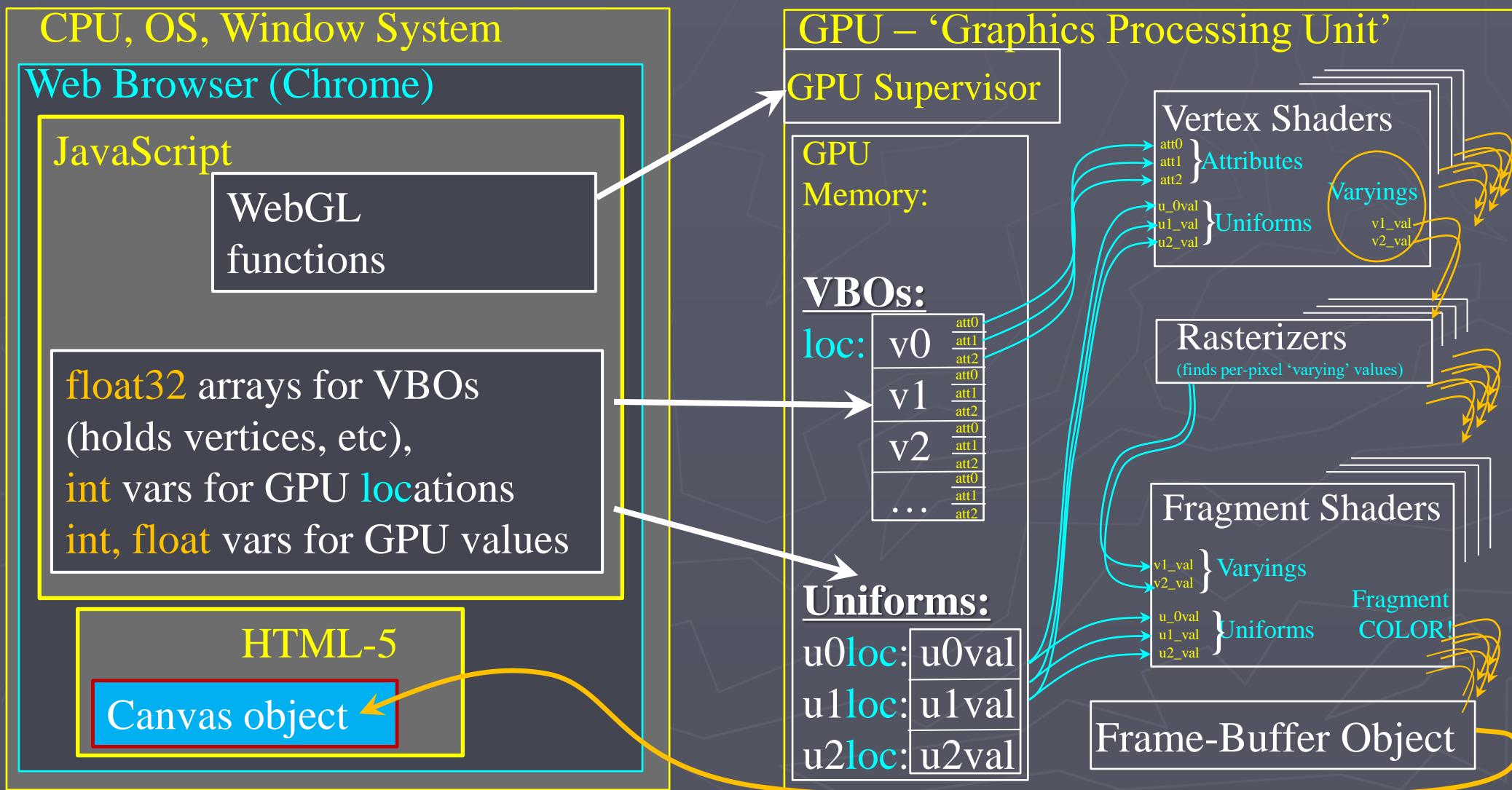
    gl_FragColor =
        vec4(DiffColr * diff, 1.0);
}
```

!END!

--**WebGL**: sends tasks to GPU

--**GPU**: executes tasks, flags errors

--**WebGL**: (optional) read GPU flags

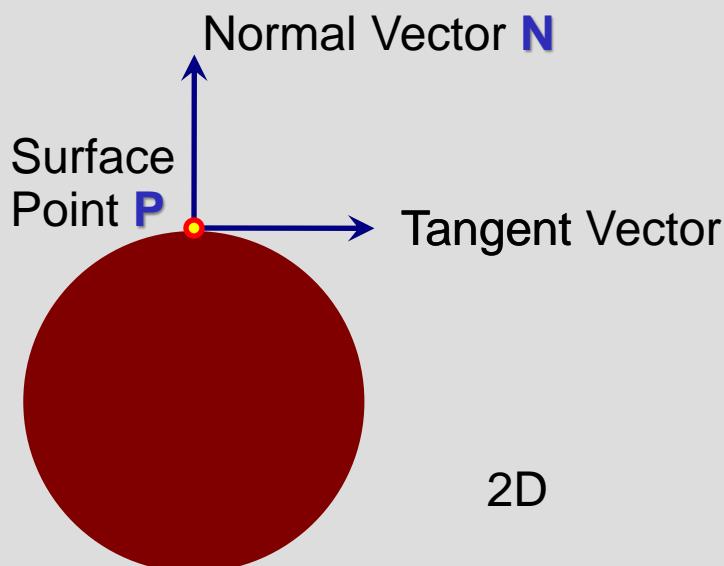


“Normal” == Surface Orientation

Perpendicular Vector at any Surface Point

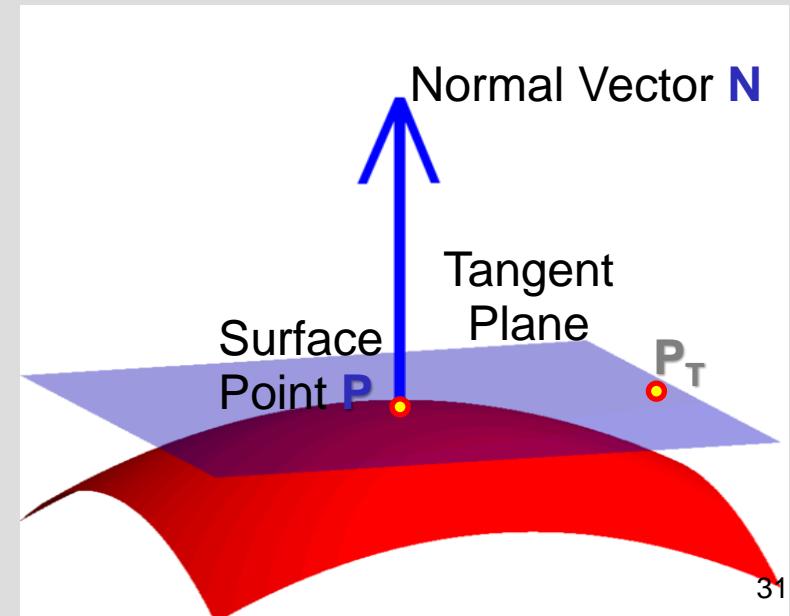
More formally:

- Unit-length vector **N** at surface point P
- Vector **N** defines surface tangent plane at P:
(For all points P_T in tangent plane, $(P_T - P) \cdot N = 0$)



2D

3D



31

RECALL: Transforming Normals

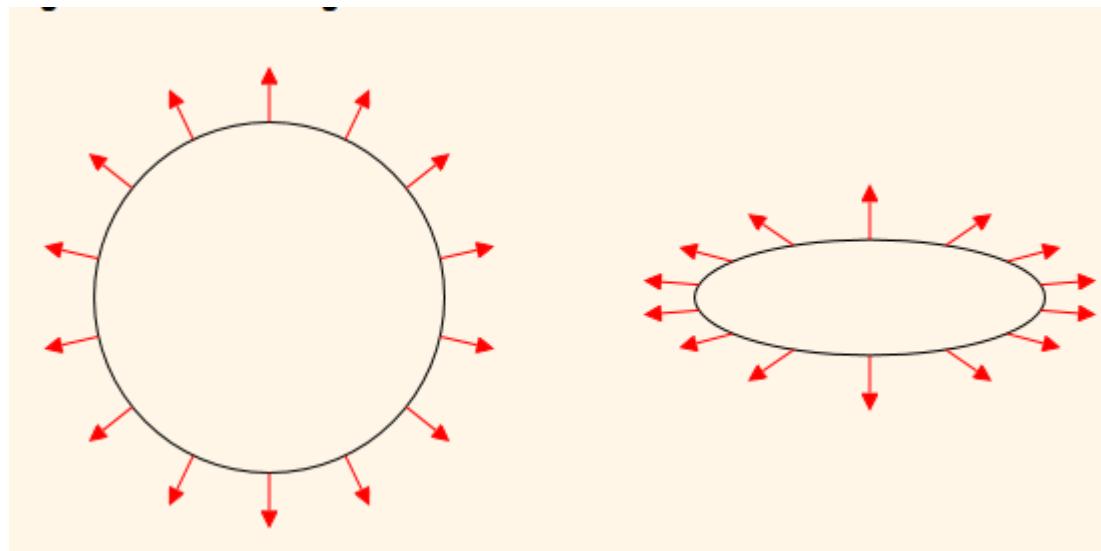
SOLUTION: use inverse-transpose:

$$\text{Normal Matrix} == (\text{Model Matrix})^{-T}$$

Why? see:

<http://www.arc synthesis.org/gltut/Illumination/Tut09%20Normal%20Transformation.html>

How? cuon-matrix-quat.js functions ...



ADVANCED: Normal Mapping Method

DEMO VIDEO:

<http://www.youtube.com/watch?v=RSmjxcAhkfE>



The University of New Mexico

(OPTIONAL) WebGL Texture Mapping

Many Slides from:
Ed Angel

Professor Emeritus of Computer Science

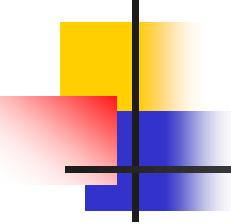
Founding Director, Arts, Research, Technology and Science Laboratory
University of New Mexico

MODIFIED by Jack Tumblin

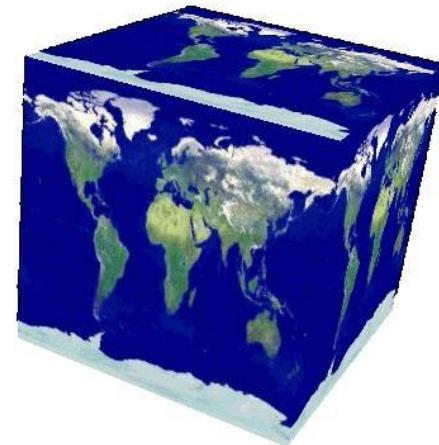
Objectives

- To **Introduce** WebGL texture mapping:
 - two-dimensional texture maps
 - assigning texture coordinates
 - forming texture images
- **Read** it: Textbook (Matsuda) Chapter 5
Optional Text (Lengyel)
- **Try** it: 2019.11.25.Texture04 (Chap 5 examples)
- **Search** it: plenty of help & tutorials online...

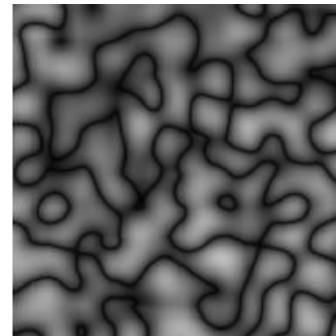
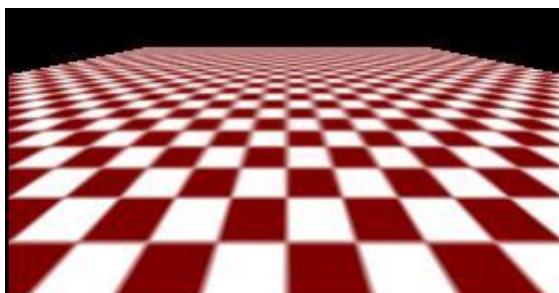
Nate Robins Tutors
DEMO



Texture Mapping



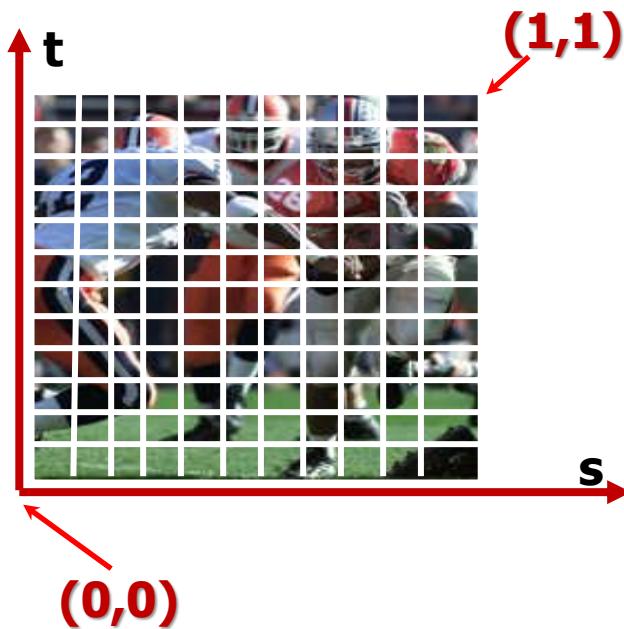
- A way of adding surface details
- Two ways can achieve the goal:
 - ❖ Surface detail polygons: create extra polygons to model object details
 - ❖ Add scene complexity and thus slow down the graphics rendering speed
 - ❖ Some fine features are hard to model!
 - ✓ Map a texture to the surface (a more popular approach)



Complexity of images does
Not affect the complexity
Of geometry processing
(transformation, clipping...)

Texture Representation

- ✓ Bitmap (pixel map) textures (OpenGL/WebGL - native)
- ✗ Procedural textures (used in advanced rendering programs. *Not implemented* within OpenGL/WebGL)



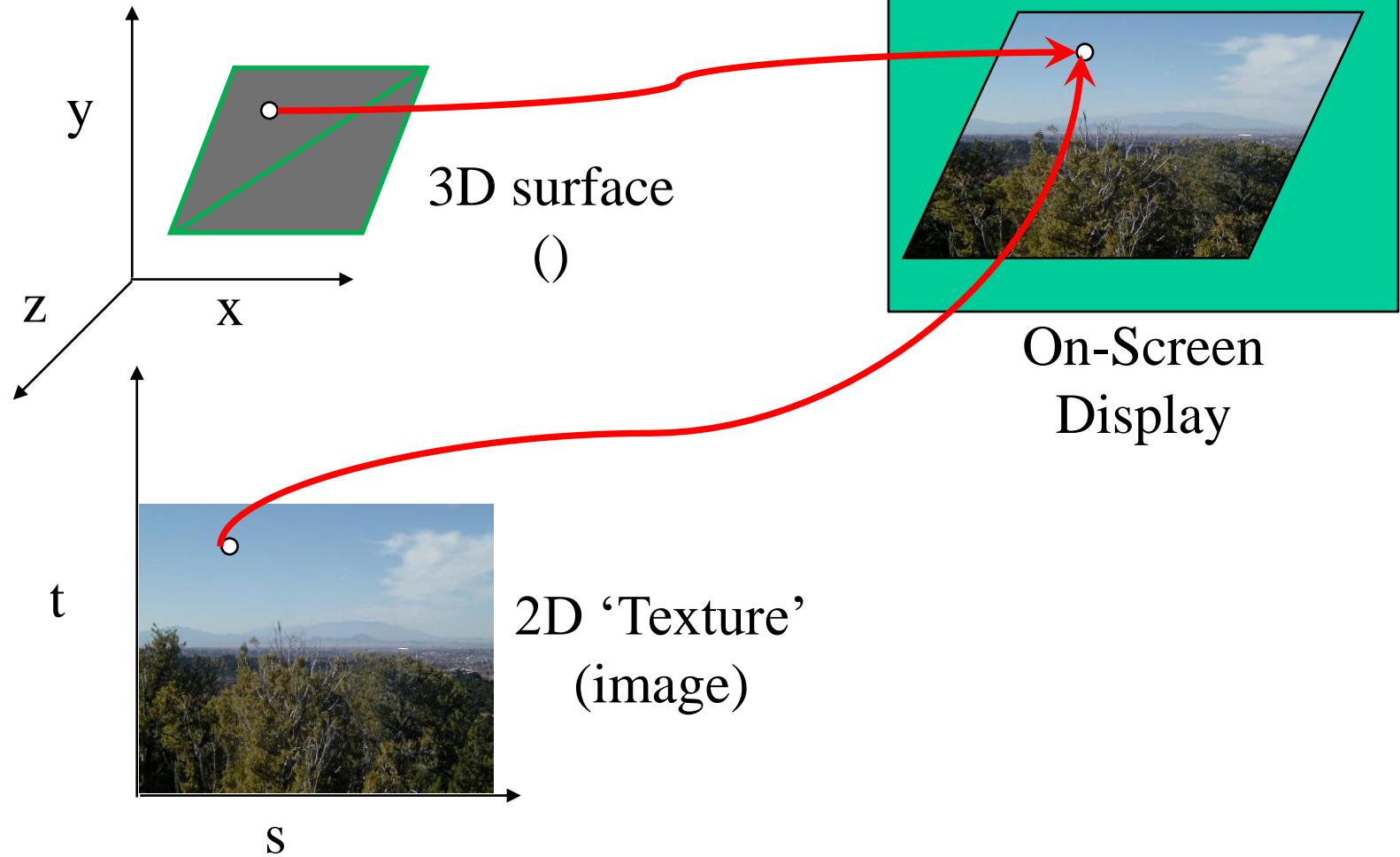
Bitmap texture:

- A 2D image - represented by 2D array `texture[height][width]`
- Each pixel (or called **texel**) located by a unique pair texture coordinates (**s**, **t**)
- OpenGL and WebGL normalize **s** and **t** to [0,1] range to span the image
- For any given (s,t) in the normalized range, there is a unique image value
(i.e. a unique [red, green, blue] set)



The University of New Mexico

Texture Mapping



Texture Example

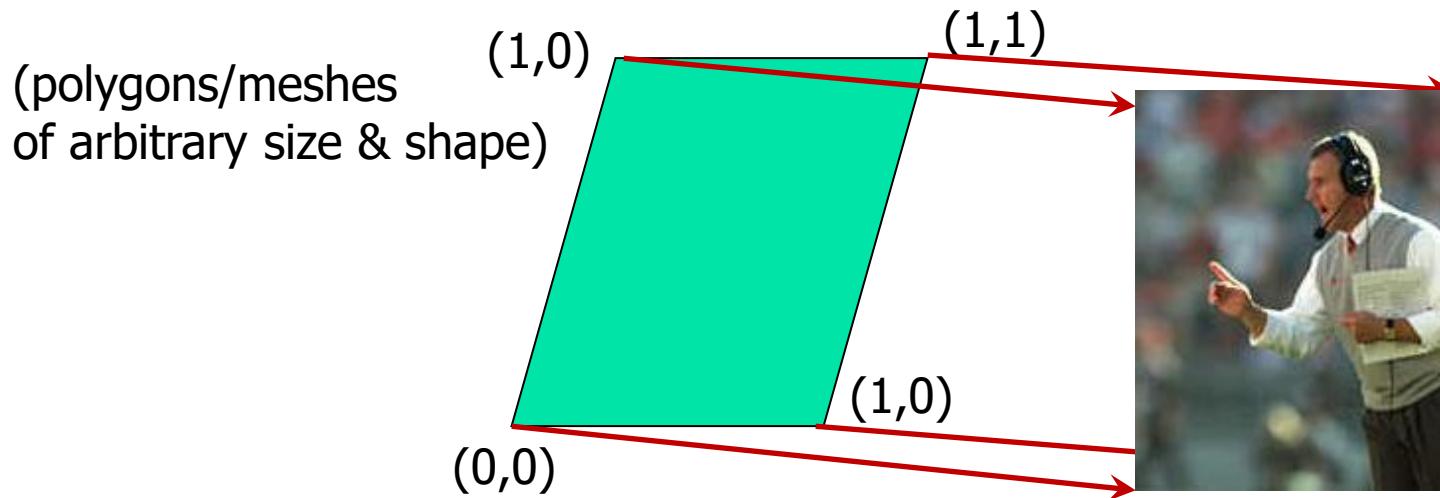
- The texture (below) is a 256×256 image that has been mapped to a rectangular polygon which is viewed in perspective
- Let's look at some Starter Code:

2021.11.24.texture04 ...



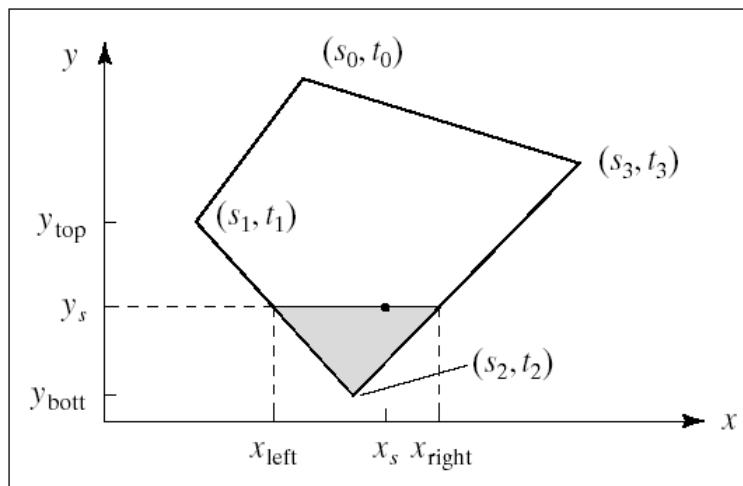
Map textures to surfaces

- Establish mapping from texture to surfaces (polygons):
 - Application specifies **texture coordinates** for each vertex in the surface; WebGL interpolates to apply texture to all on-screen pixels/fragments



Texture Rasterization

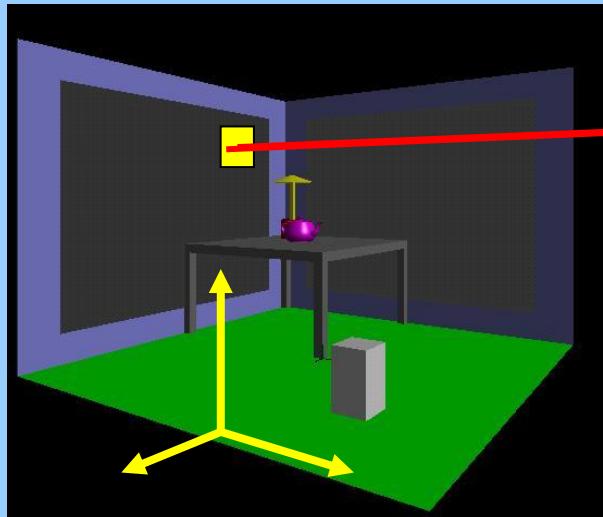
- Texture coordinates are interpolated from polygon vertices just like ... remember ...
 - Color : Gouraud shading
 - Depth: Z-buffer
 - First along polygon edges between vertices
 - Then along scanlines between left and right sides



from F. S. Hill "Interactive Computer Graphics with OpenGL 2001"

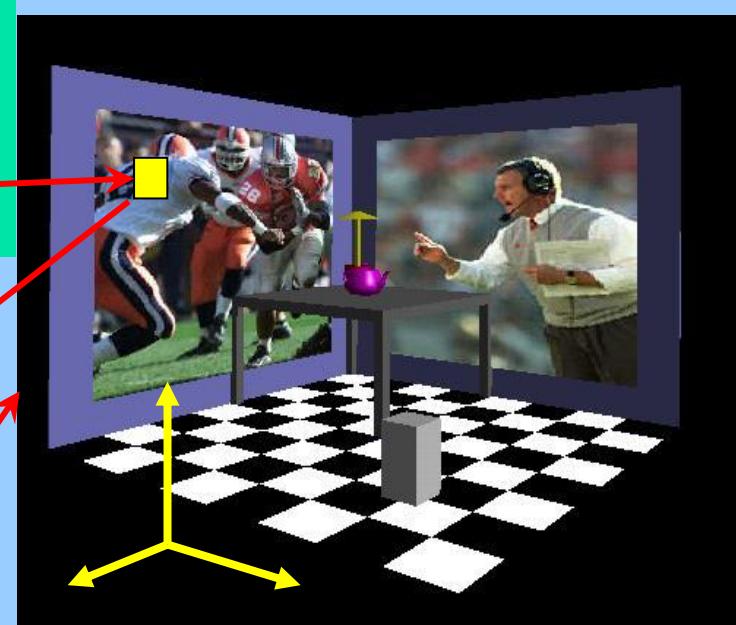
Texture Mapping Steps

3D geometry



1. projection:

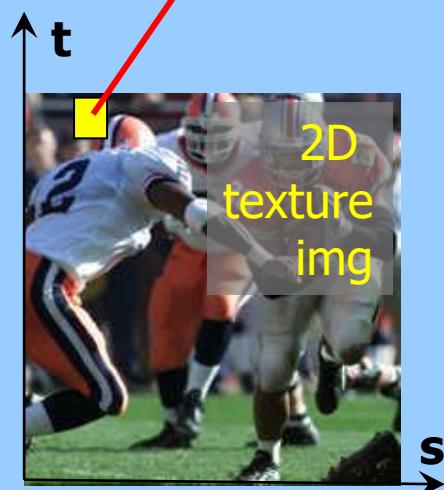
find each fragment's
on-screen coords
(interpolate screen pixels)



2. texture lookup:

Texture Sampler Unit solves an Inverse problem: find s,t for a given display pixel/fragment

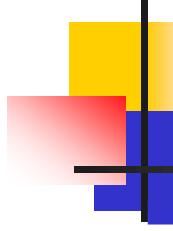
- (s,t) params linearly interpolate between vertex x,y,z,w values.
Solve for s,t : $x_{\text{disp}} = x(s,t)/w(s,t);$
 $y_{\text{disp}} = y(s,t)/w(s,t);$
- retrieve texel color at (s,t)



2D projection of 3D geometry

3. Set Fragment Color

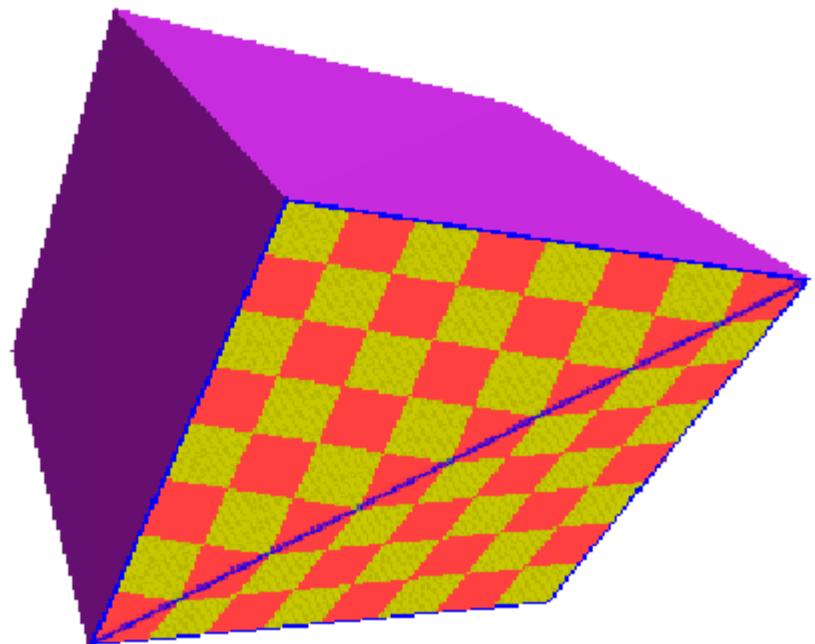
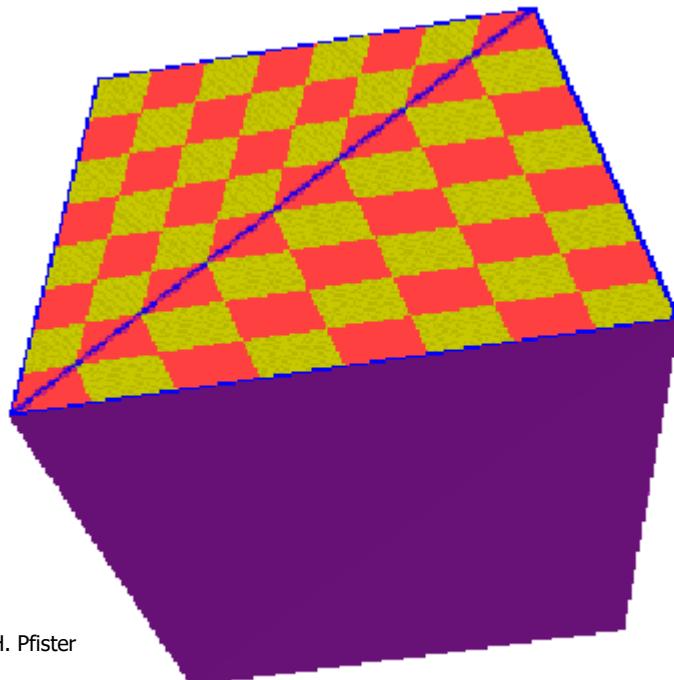
GLSL fragment shader computes final pixel color from texel color + more



Linear Texture Coordinate Interpolation

This doesn't work in perspective projection!

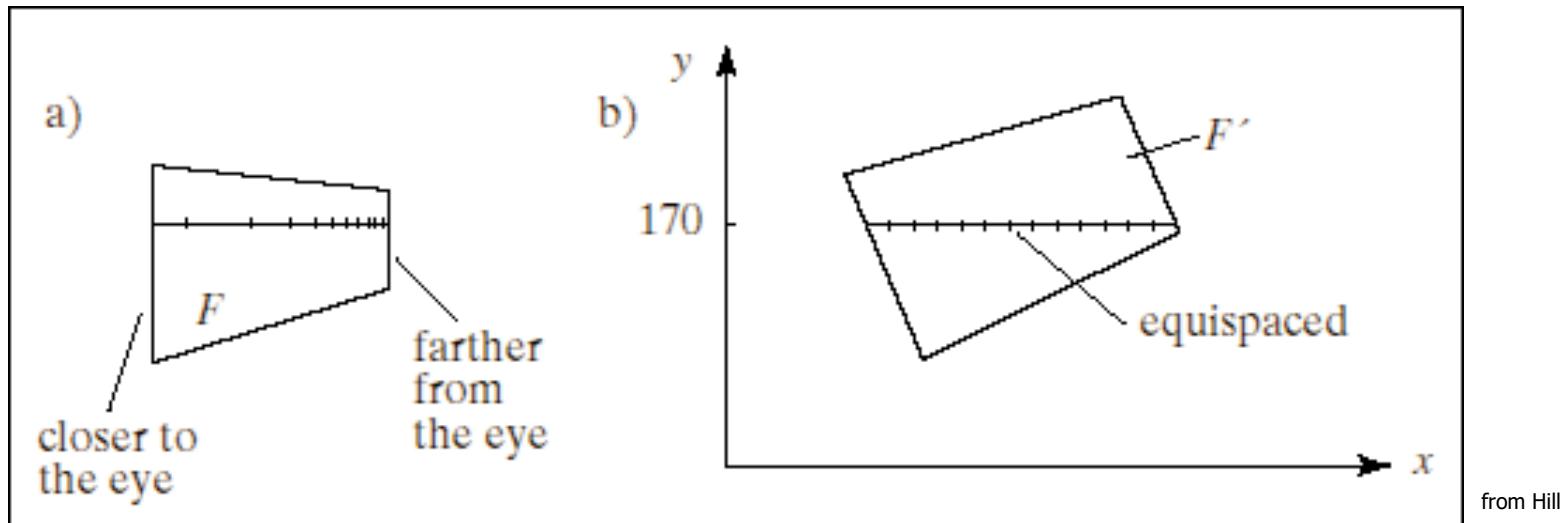
- The textures look warped along the diagonal
- Noticeable during an animation



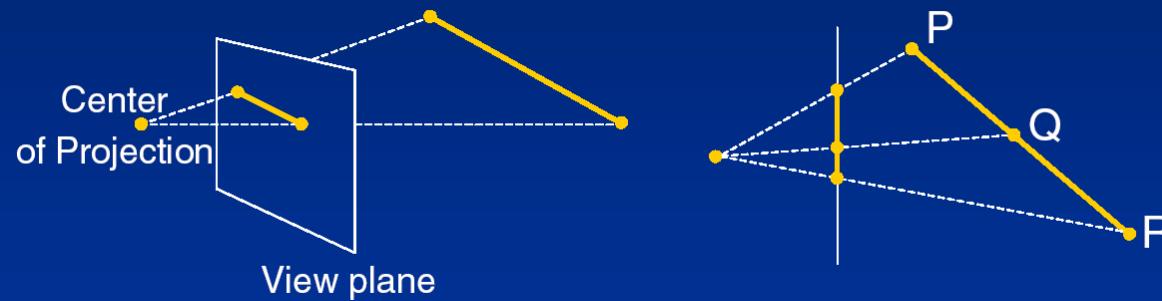
courtesy of H. Pfister

Why?

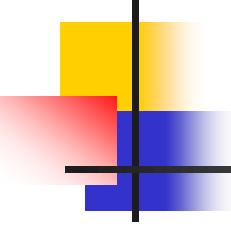
- Equal spacing in screen (pixel) space is **not** the same as in homogeneous coordinates used in perspective projection
 - **Perspective foreshortening**



from Hill

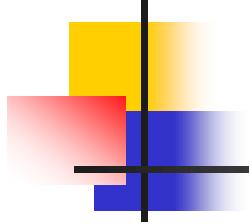


courtesy of
H. Pfister



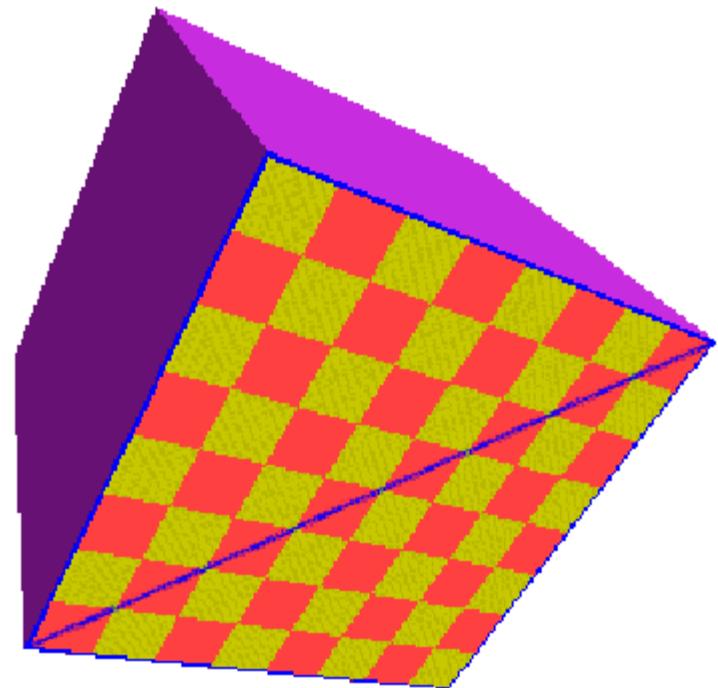
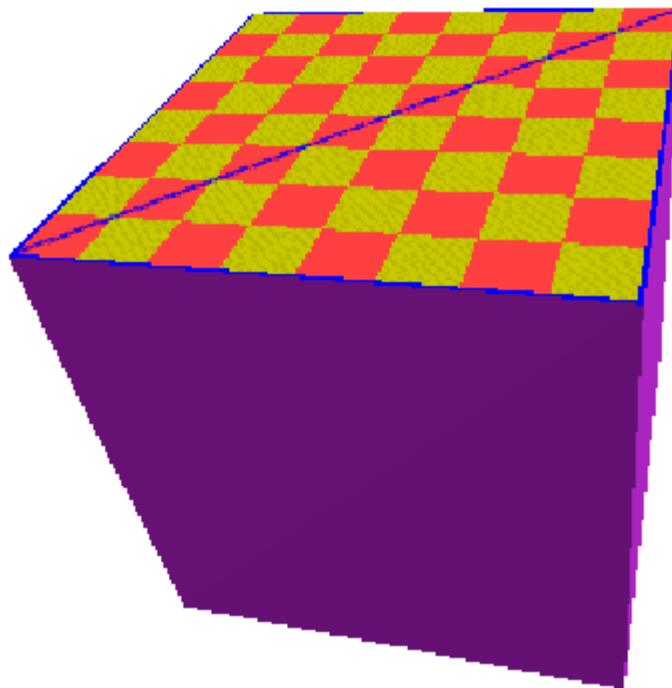
Perspective-Correct Texture Coordinate Interpolation

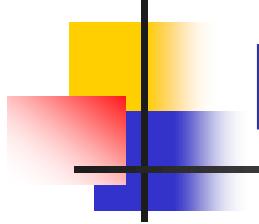
- Interpolate (`tex_coord AND w`) over the polygon, then do perspective divide after interpolation
- Compute at each vertex after perspective transformation
 - “Numerators” $s/w, t/w$
 - “Denominator” $1/w$
- Linearly interpolate $1/w, s/w$, and t/w across the polygon
- At each pixel
 - Perform perspective division of interpolated texture coordinates $(s/w, t/w)$ by interpolated $1/w$ (i.e., numerator over denominator) to get (s, t)



Perspective-Correct Interpolation

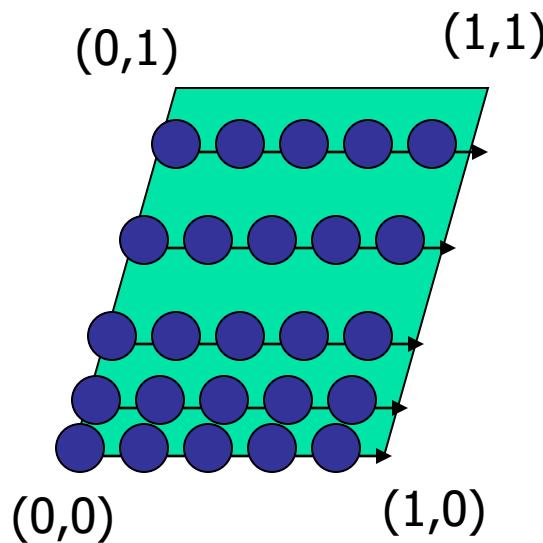
- That fixed it!



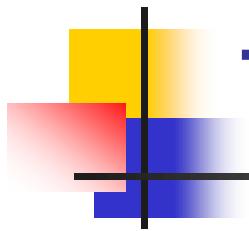


Map textures to surfaces

- WebGL texture mapping method:
rasterization (or 'backward' mapping)

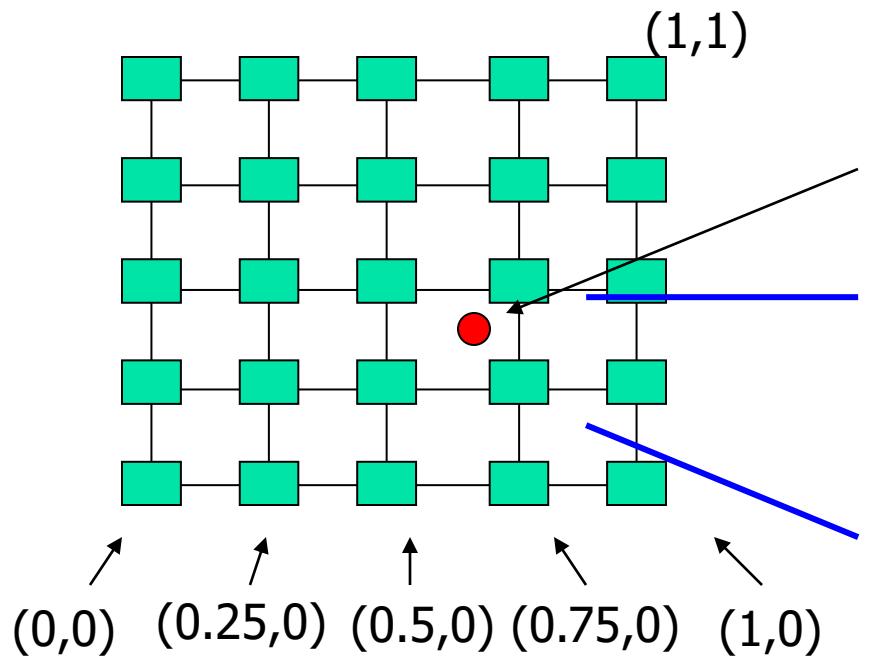


- ❑ For each on-screen pixel,
find the texture coordinates (s, t).
How? Interpolate between texture coords
stretched between x,y,z,w vertex positions.
(Not just fixed attributes! GLSL Vertex Shaders can
compute more complex texture coords...)
- ❑ THEN: Use texture coordinates result
to 'look up' a color value within the image
How? GPU 'texture units' do a specialized form
of rasterizing...



Texture @ Fractional Addresses

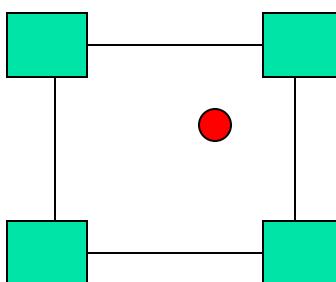
- Given a texture coordinate (s,t) , estimate the unique color value at that image location



How about coordinates that are not exactly at the intersection (pixel) positions?

ANS: use a selected Texture 'Filter' methods:

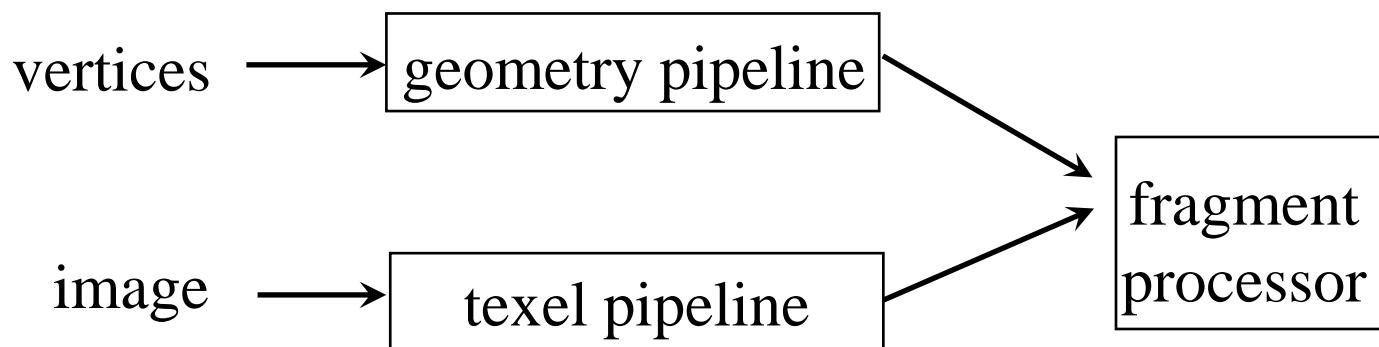
- A) Nearest neighbor (default)
- B) Linear Interpolation
- C) Other filters...





Texture Mapping and the WebGL Geometry Pipeline

- Texture images and vertex geometry flow through separate pipelines; they merge during fragment processing
→ “complex” image textures, multiple images etc. do not affect geometric complexity



How Can I do all this in WebGL?

Several steps in texture-map code:

1. Set up WebGL to apply texture-image to surfaces:
 - read or generate texture image; xfer into the GPU
 - assign WebGL samplers (hdwe) to stored texture
 - enable texturing in WebGL (see textbook, Chapter 5)
2. assign texture coords to through vertex attributes
 - YOU must devise a proper mapping function in your JS code... (it's an interesting problem)
3. Specify boundary conditions & misc for samplers
 - e.g. wrap, clamp, mirror; texture-sample filtering
4. Draw the textured shapes

Texture Images in WebGL

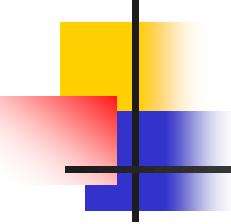
1) Within the GPU:

- a 2D array of *texels* (texture elements) stored in GPU
- Size? ALWAYS square, always a power-of-2(POT);
32x32, 64x64, 128x128; any POT size up to about 4096x4096
(larger, non-POT textures now available in webGL2.0 ...)

2) Load Image: (use a standard format: PNG,JPG, etc)

- Load from a separate a file* (tricky!), or
- encoded,stored within your .js code (messy!), or even
- generated by your JavaScript code (synthetic!)

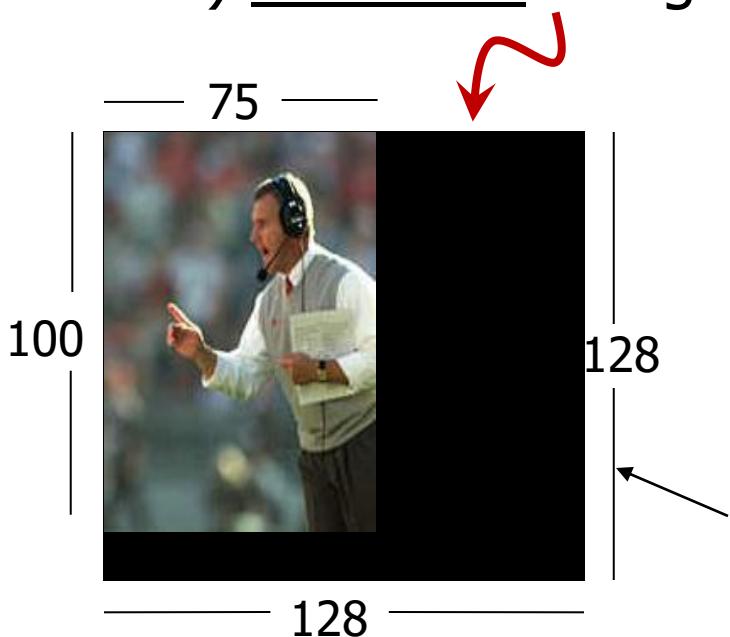
- WebGL supports only **2 dimensional** texture maps
 - no need to enable 2D textures (as in desktop OpenGL):
 - why? desktop OpenGL supports 1D,2D,3D, & 4D textures



Fixed texture sizes ?!?!?



- If the dimensions of the texture map are *not* power of 2, you can
 - 1) Zero Pad filling!
 - 2) **WAIT** for a later version of WebGL, or give up and write program in OpenGL



Remember to adjust the texture coordinates to avoid the 'zero pad' areas – to avoid black texels on texture-mapped surfaces

How to Make Checkerboard Image (.js)

```
// tl;dr: Exclusive-OR on modulo 2: XOR(x%2, y%2)
-----
var image1 = new Uint8Array(4*texSize*texSize);
for ( var i = 0; i < texSize; i++ ) {
    for ( var j = 0; j < texSize; j++ ) {
        var patchx =Math.floor(i/(texSize/numChecks));
        var patchy =Math.floor(j/(texSize/numChecks));
        if(patchx%2 ^ patchy%2) c = 255; // ^ ==EX-OR
        else c = 0;
        //c = 255*(((i & 0x8) == 0) ^ ((j & 0x8) == 0))
        image1[4*i*texSize+4*j] = c;
        image1[4*i*texSize+4*j+1] = c;
        image1[4*i*texSize+4*j+2] = c;
        image1[4*i*texSize+4*j+3] = 255;
    }
}
```

Using a GIF image

// How to specify an image in JS file:

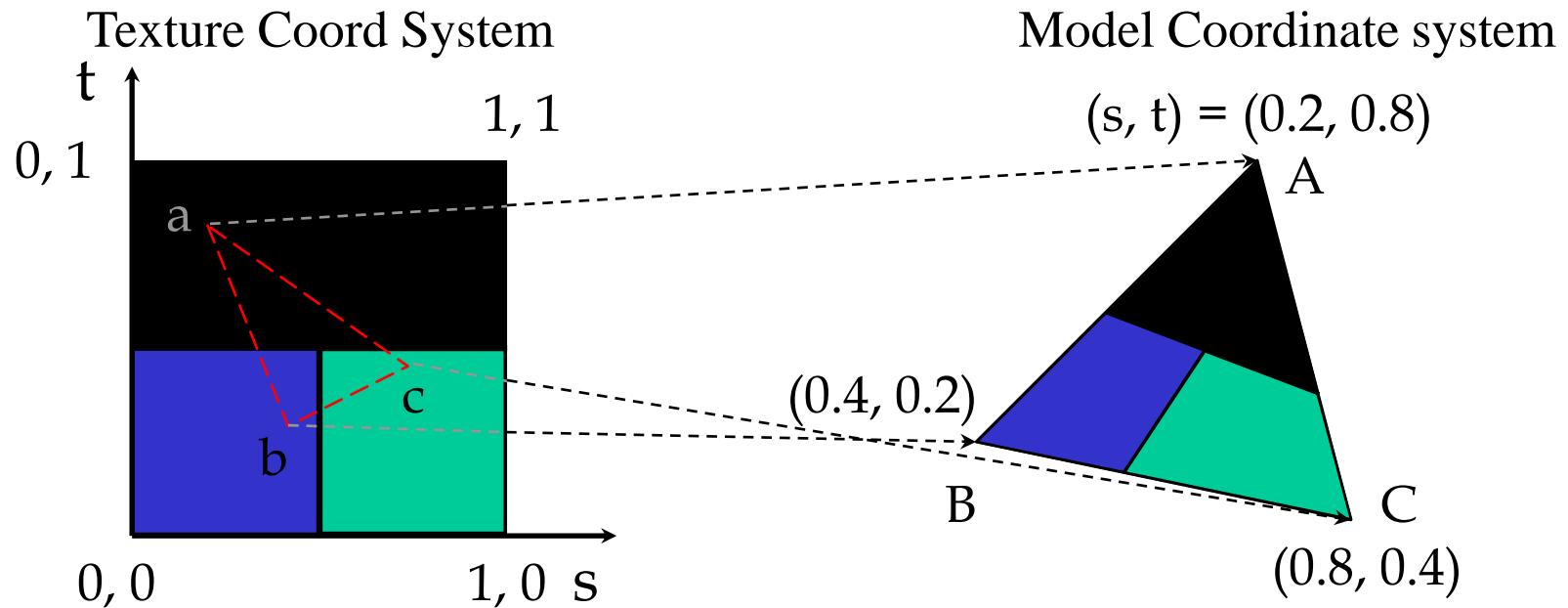
```
var image = new Image();
    image.onload = function() {
        configureTexture( image );
    }
image.src = "SA2011_black.gif" // filename
```

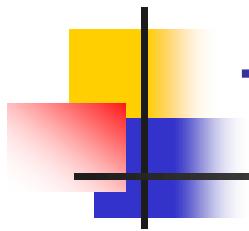
// or specify image in HTML file with tag

```
// <img id = "texImage" src = "SA2011_black.gif"></img>
var image = document.getElementById("texImage")
window.onload = configureTexture( image );
```

Mapping a Texture

- Parametric texture coordinates $0.0 \leftarrow (s,t) \leftarrow 1.0$
- Specify as a new kind of 2D vertex attribute
- *Later:* may have *multiple textures* per vertex, thus multiple texture coordinates...

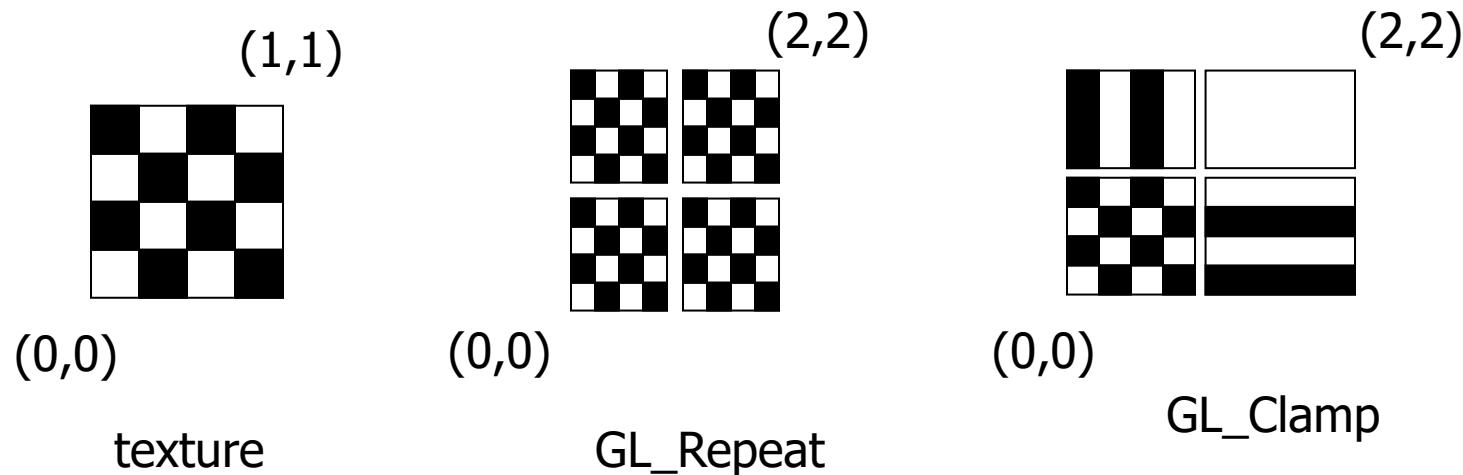




Texture mapping parameters

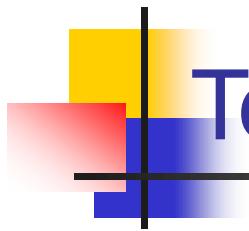


- What will the GPU do when given texture coordinates (s,t) beyond/outside the $[0,1]$ range?



- Example: `gl.TexParameteri(GL_TEXTURE_2D, gl.TEXTURE_WRAP_S, GL_CLAMP)`

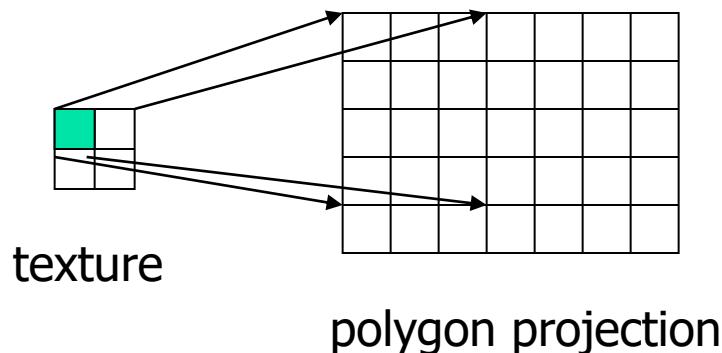
If $(s > 1)$ $s = 1$
If $(t > 1)$ $t = 1$



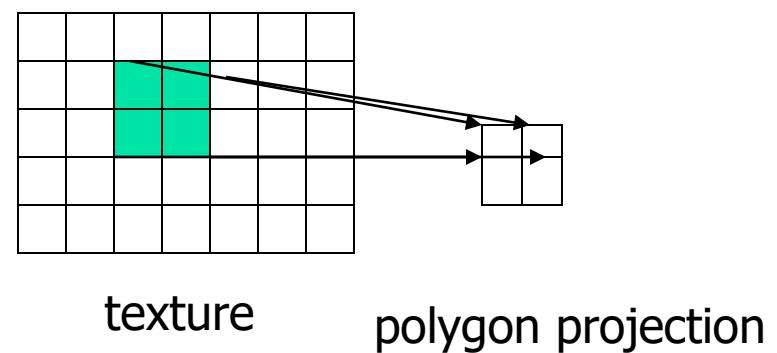
Texture mapping parameters(2)



- Since a polygon can get transformed to arbitrary screen size, texels in the texture map can get magnified or minified.



Magnification



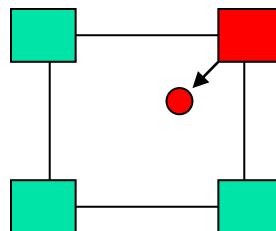
Minification

- Filtering: interpolate a texel value from its neighbors or combine multiple texel values into a single one

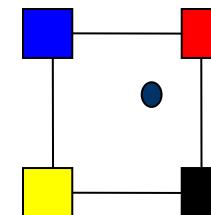
Texture mapping parameters(3)

■ OpenGL texture filtering:

- 1) Nearest Neighbor
(lower image quality)



- 2) Linear interpolate the neighbors
(better quality, slower)



```
glTexParameteri(GL_TEXTURE_2D,  
GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

```
glTexParameteri(GL_TEXTURE_2D,  
GL_TEXTURE_MIN_FILTER,  
GL_LINEAR)
```

Or GL_TEXTURE_MAX_FILTER

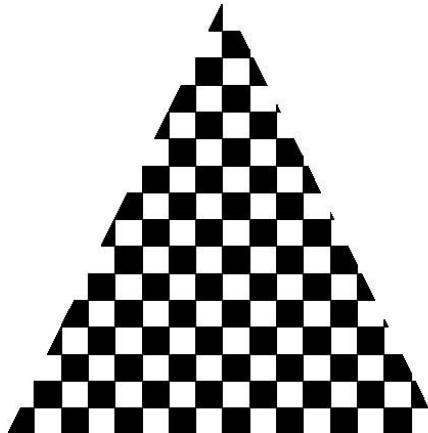
Interpolation

WebGL interpolates between texture coords to fill entire triangle surface; each pixel(fragment) gets a texel value:

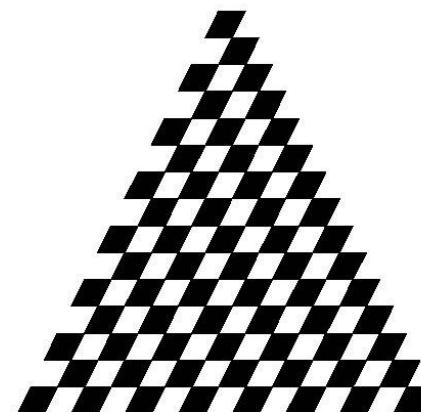
Poor Texture Coord choices cause ugly distortions

(DEMO: Nate Robins Tutor)

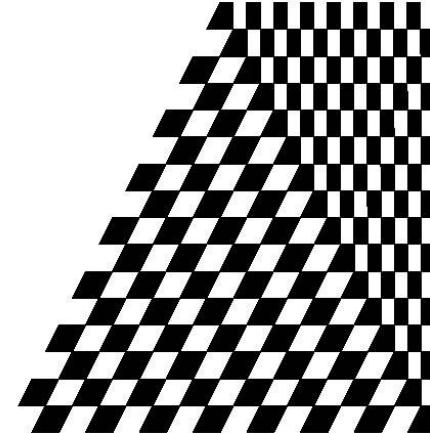
good selection
of tex coordinates



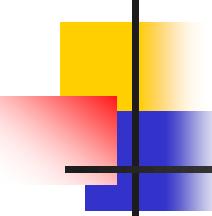
poor selection
of tex coordinates



Poor choices for two adjacent triangles
(bilinear interpolation)



END

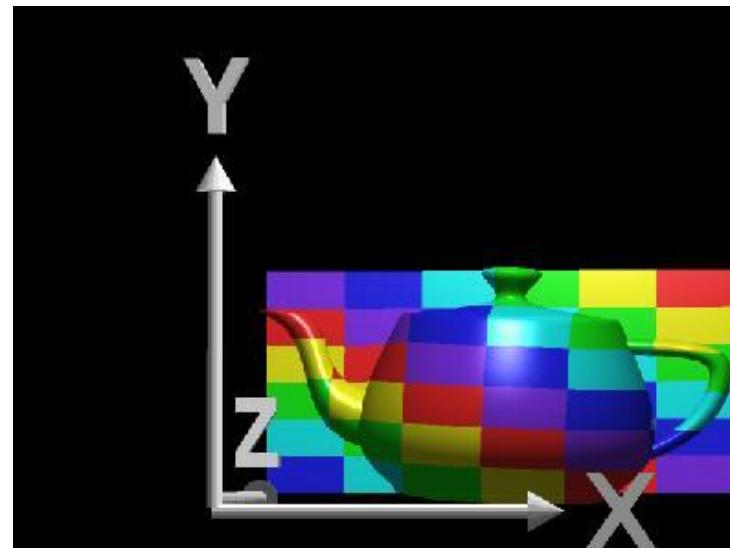


Projector Functions

- How do we map the texture onto an arbitrary (complex) object?
 - Construct a simple mapping from each 3-D surface point to an easy-to-texture intermediate surface, such as a plane
- Mapping Idea: Project object points to the intermediate surface with a parallel or perspective projection
 - HINT: Perspective? try putting the focal point *inside* the object!
- Plane
- Cylinder
- Sphere
- Cube

courtesy of R. Wolfe

Planar projector:

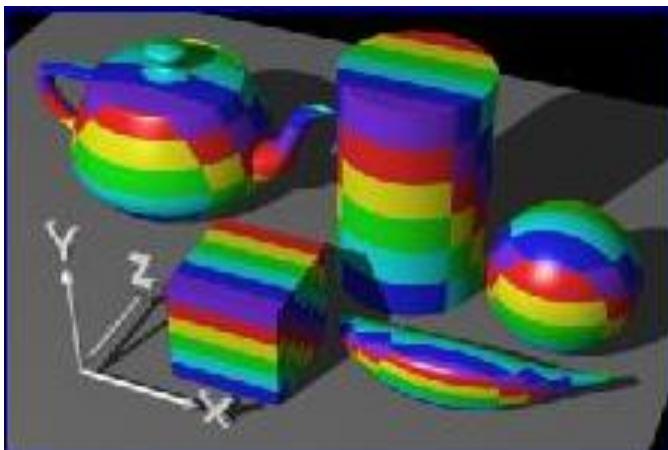
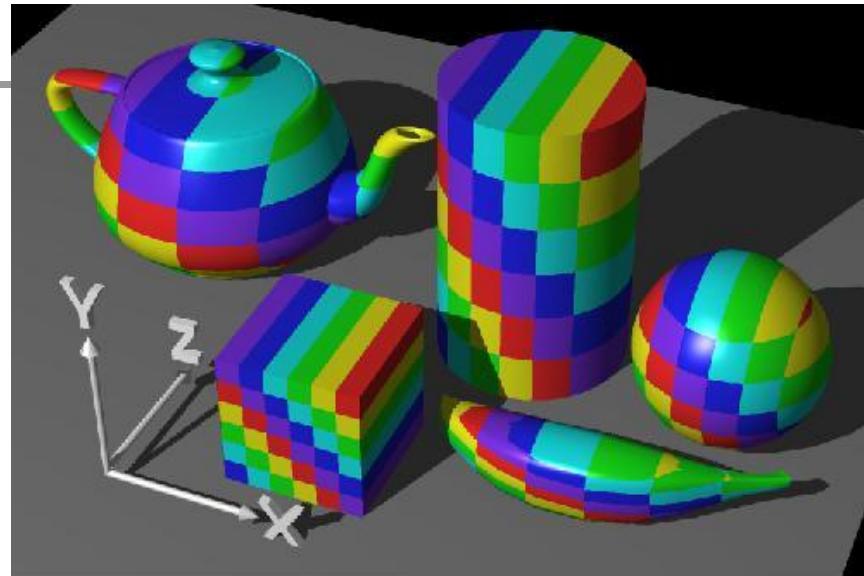


Planar Projector

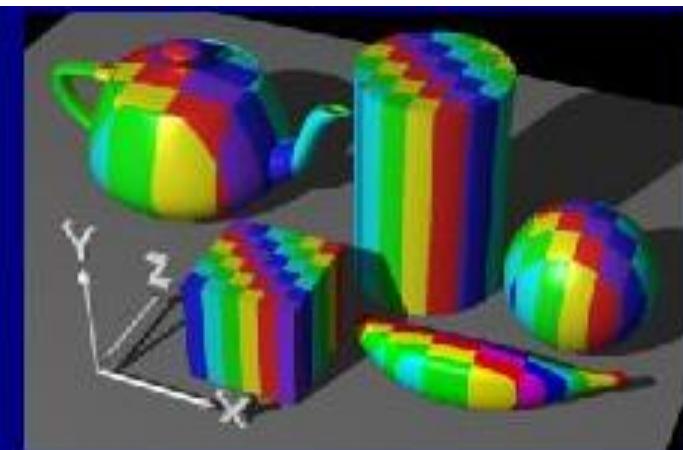


Orthographic projection
onto XY plane:

$$s = x, \quad t = y$$



...onto YZ plane



...onto XZ plane

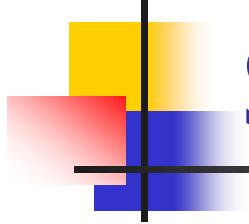
courtesy of
R. Wolfe

Cylindrical Projector

- Convert rectangular coordinates (x, y, z) to cylindrical (r, θ, h) :
- use only (h, θ) to index texture image

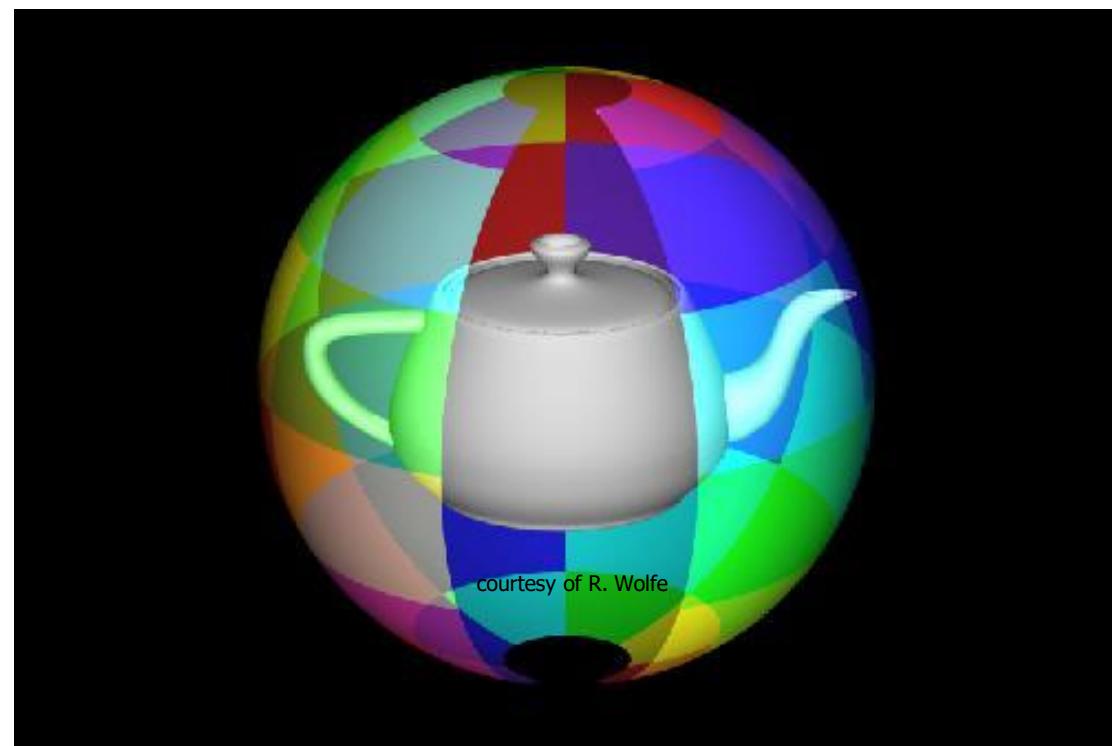
courtesy of R. Wolfe





Spherical Projector

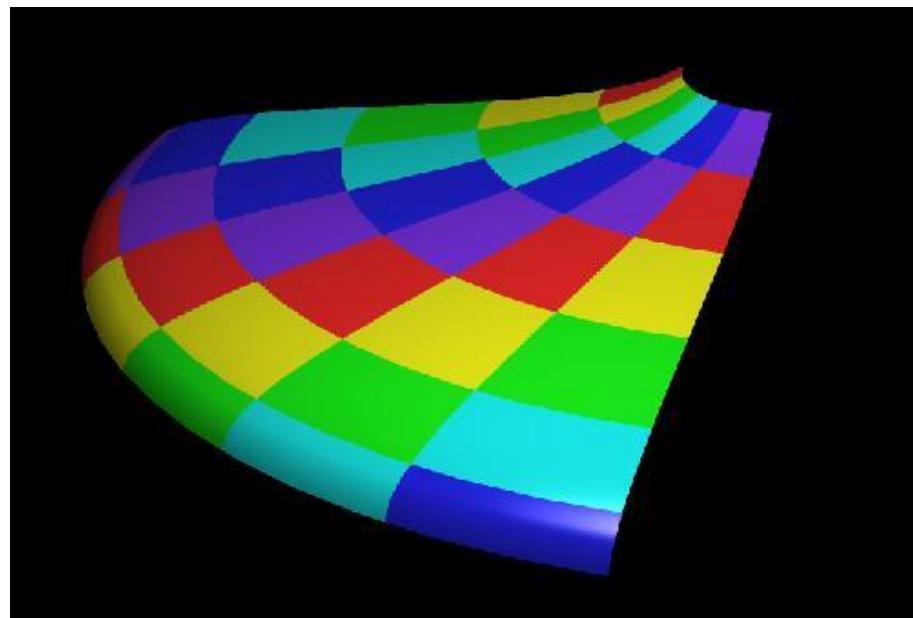
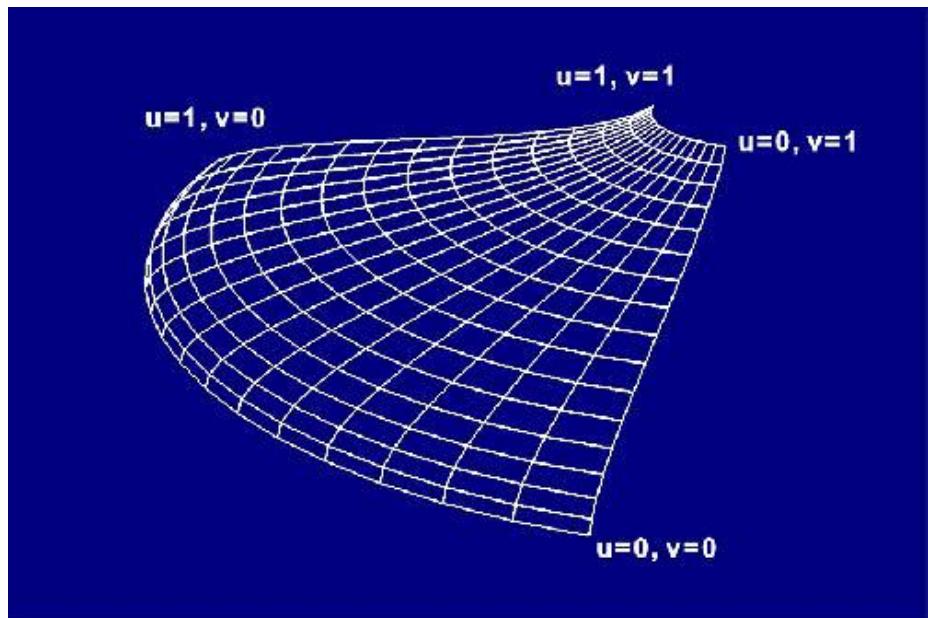
- Convert rectangular coordinates (x, y, z) to spherical (θ, ϕ)



Parametric Surfaces

A parameterized surface patch

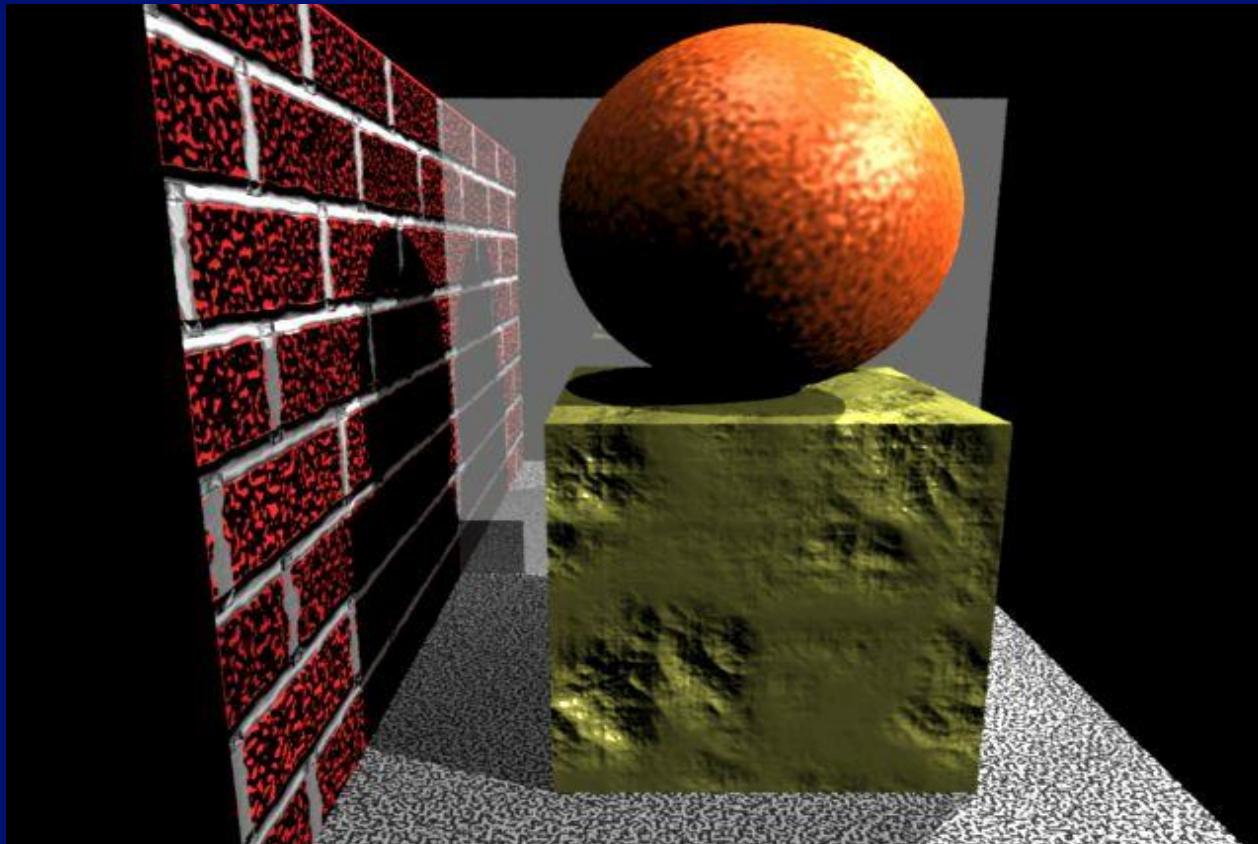
- $x = f(u, v), y = g(u, v), z = h(u, v).$



courtesy of R. Wolfe



Bump Mapping



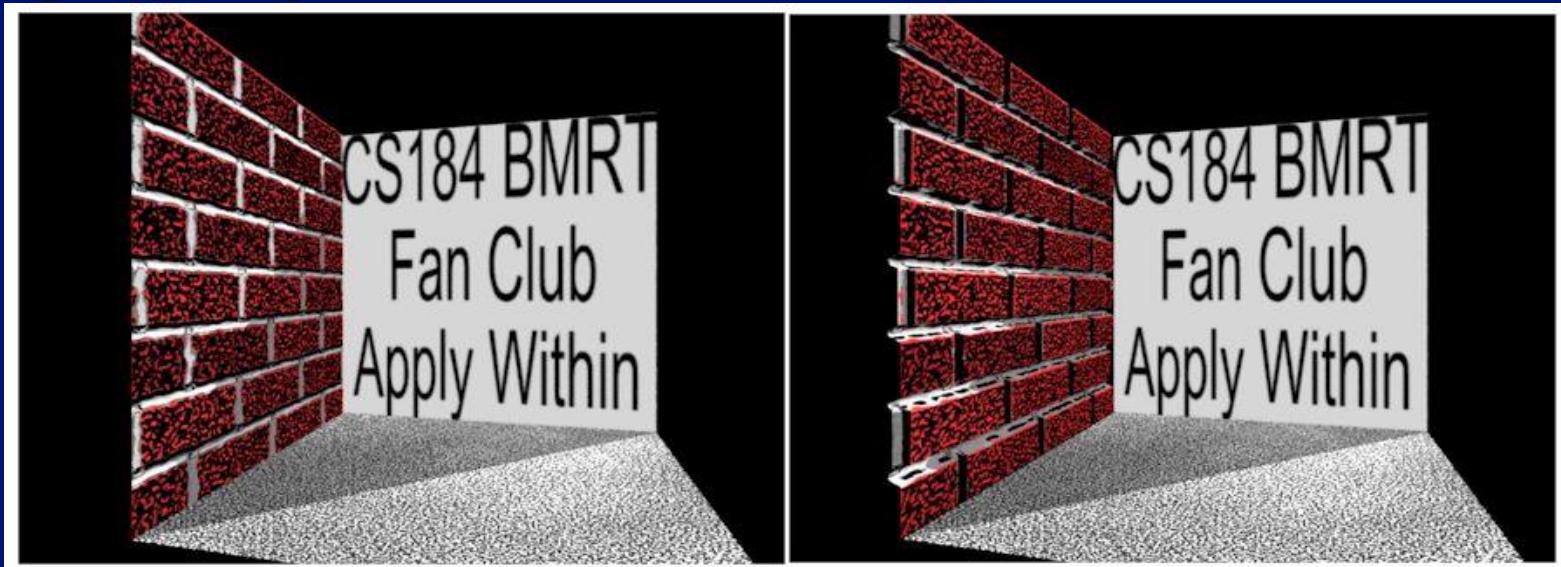
UNIVERSITY
OF
VIRGINIA



Displacement Mapping

Bump mapped normals are inconsistent with actual geometry. Problems arise (shadows).

Displacement mapping actually affects the surface geometry





Mipmaps

multum in parvo -- many things in a small place

A texture Level-Of-Detail technique

*Pre-specify a series of prefiltered texture maps
of decreasing resolutions*

Requires 4/3^{rds} more texture storage

*Eliminates most or all shimmering and flashing
(aliasing) as objects move*

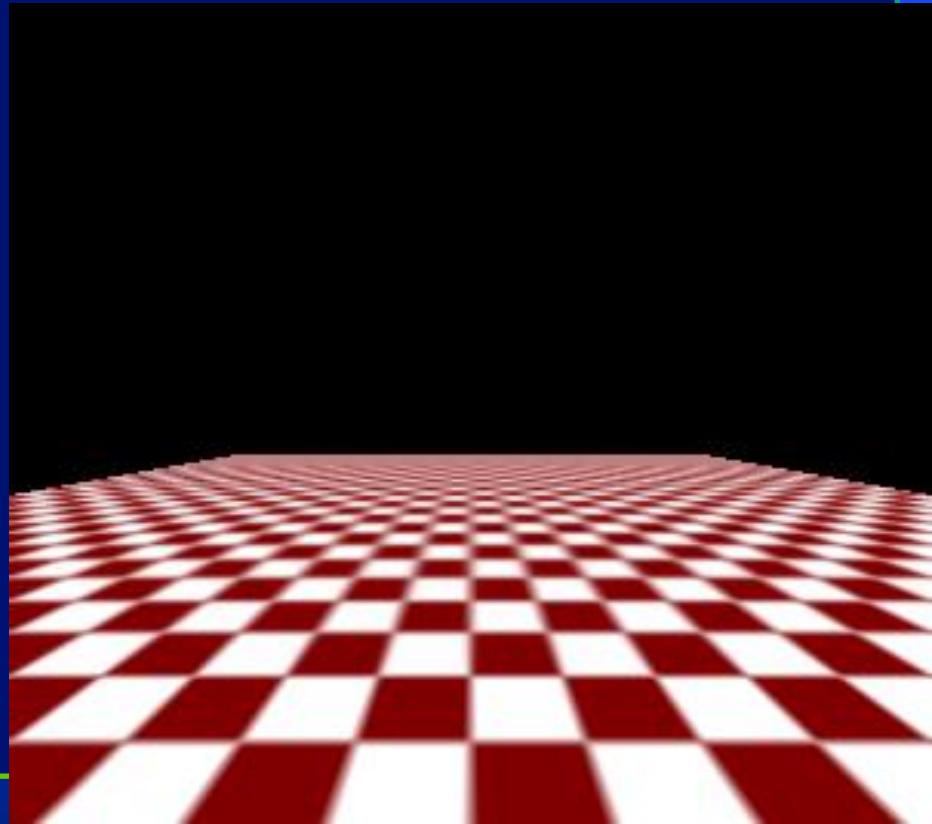
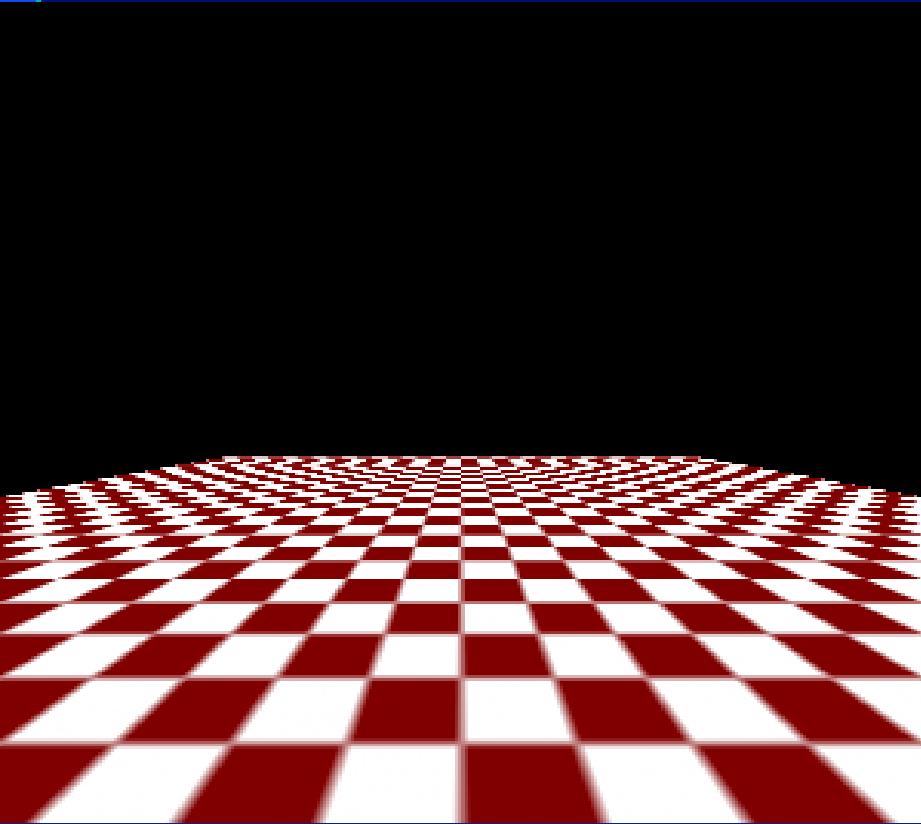


MIPMAPS

Without MIPMAP



With MIPMAP

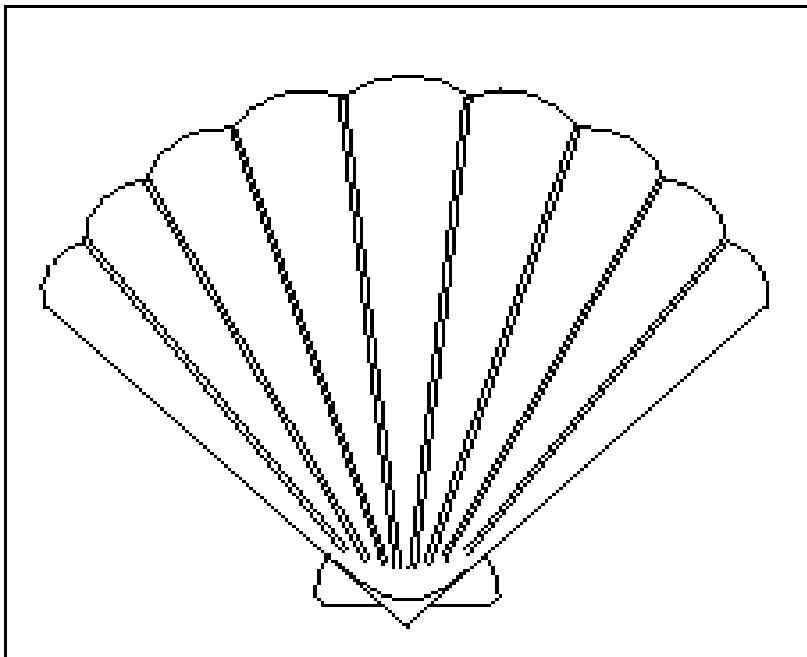




MIPMAPS

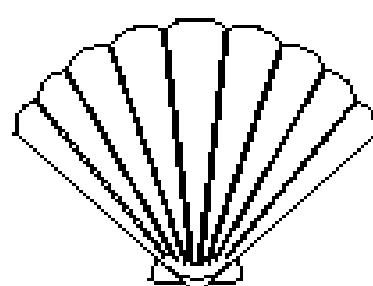
*Arrange progressively-double-sized images
into one block of memory*

Original Texture

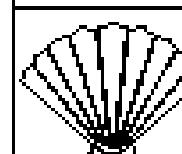


Pre-Filtered Images

1/4



1/16



1/64
etc.

1 pixel Y



Filtering

OpenGL tries to pick best mipmap level

Question: Which texel corresponds to a particular pixel?

GL_NEAREST (Point Sampling)

- Pick the texel with center nearest pixel

GL_LINEAR (Bilinear Sampling)

- Weighted average of 2x2 closest texels

GL_NEAREST_MIPMAP_LINEAR

- Average nearest texels from two mipmap levels

GL_LINEAR_MIPMAP_LINEAR (Trilinear)

- Average two averaged texels from two mipmaps