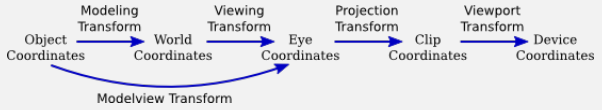
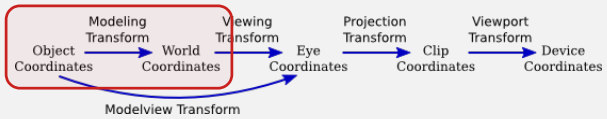
The Graphics Pipeline
WebGL

Coordinate Systems for 3D Graphics



Coordinate Systems for 3D Graphics



single objects are defined in *object coordinates* (or *model coordinates*)

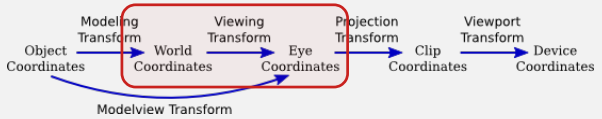
what is used when defining geometry for primitives

the scene as a whole is defined in *world coordinates*

define *modeling transformations* to set the size, orientation, position of individual objects in the overall scene

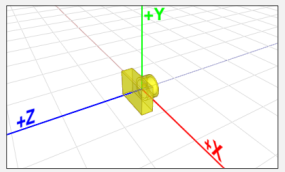
in hierarchical modeling, this is repeated on many levels
- define a modeling transform to size/orient/place a primitive into the OC of a larger piece
- define a modeling transform to size/orient/place the larger piece into the OC of a still larger piece
- ...

Coordinate Systems for 3D Graphics

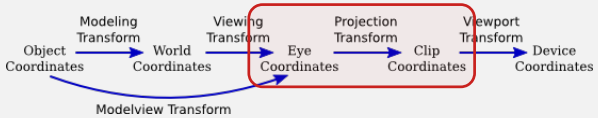


eye coordinates (or *view coordinates*) are from the perspective of the viewer
- (0,0,0) at the viewer (viewer's eye)
- viewer looks down negative z
- positive y points up
- positive x points right

viewing transform transforms WC→EC

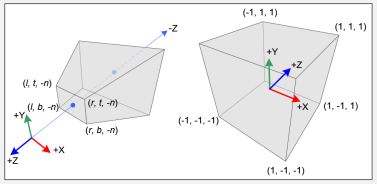


Coordinate Systems for 3D Graphics



the world is an infinite space, but only a finite region of it can be displayed in a window on the screen

- window size limits the xy extent
- near/far extent imposed by the depth buffer implementation



anything outside this *view volume* is *clipped* and not drawn

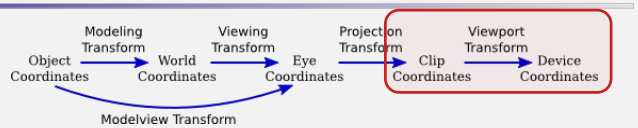
OpenGL defines the view volume as a cube centered at the origin with extents -1 to 1 in all directions

- coordinates within this cube are *clip coordinates* (*) or *normalized device coordinates*

(*) clip coordinates may also refer to an intermediate step

the *projection transformation* transforms EC→CC

Coordinate Systems for 3D Graphics



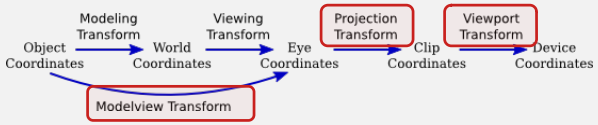
the *viewport* defines where the drawing appears on the screen

- in *device coordinates* (or *screen coordinates*)

the *viewport transformation* maps CC→DC

- corresponds to the 2D window-to-viewport transformation
- view window is the front of the clipping cube
- 3D→2D amounts to dropping z coordinate (though kept for depth buffer purposes)

OpenGL Viewing Pipeline



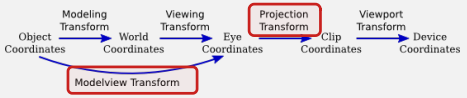
- three transforms to specify
  - modelview transform
    - OpenGL does not have an internal notion of WC
  - projection transform
  - viewport transform

OpenGL 1.1

- OpenGL 1.1 uses a *fixed pipeline*

programmer –
– specifies geometry, modelview transform, projection transform

system –
– tracks modelview, projection matrices
– transforms geometry OC → DC
– clips against view volume (in CC)
– uses lighting equation to determine vertex colors
– rasterizes primitives, interpolating vertex colors for each pixel



OpenGL 1.1 vs OpenGL 2.0

- OpenGL 2.0 added a *programmable pipeline*

fixed pipeline	**programmable pipeline**
programmer specifies geometry, modelview transform, projection transform	programmer specifies geometry, vertex shader, fragment shader – vertex shader does OC → CC, computes colors for vertices – fragment shader determines final pixel color
system – – tracks modelview, projection matrices – transforms geometry OC → DC – clips against view volume (in CC) – uses lighting equation to determine vertex colors – rasterizes primitives, interpolating vertex colors for each pixel	system – – calls vertex shader for each vertex – clips against view volume (in CC) – rasterizes primitives, calling fragment shader for each pixel with interpolated values (not limited to color)

WebGL

- based on a version of OpenGL for systems like smartphones and tablets
- programmable pipeline is required

- gone
  - functions for working with transformations (`glRotate`, `glTranslate`, `glScale`, `glPushMatrix`/`glPopMatrix`, etc)
  - `glBegin`/`glEnd`
  - `glColor`, `glNormal`

- added
  - vertex, fragment shaders where programmer implements transformations, lighting equation

WebGL

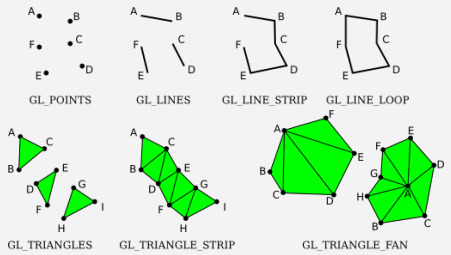
- WebGL runs in a web browser

- new environments and languages
  - HTML for the web page containing the WebGL canvas
  - JavaScript for specifying the geometry
  - GLSL for shaders

Programmable Pipeline Concepts

`gl` refers to the WebGL graphics context
the variable name is up to the programmer; the name `gl` is a convention, not a requirement

- a primitive is defined by its type, a list of vertices, and properties associated with each vertex
  - type can be `gl.POINTS`, `gl.LINES`, `gl.LINE_STRIP`, `gl.LINE_LOOP`, `gl.TRIANGLES`, `gl.TRIANGLE_STRIP`, `gl.TRIANGLE_FAN`



Programmable Pipeline Concepts

- a primitive is defined by its type, a list of vertices, and properties associated with each vertex
  - *uniform variables* (*uniforms*) have a single value for the whole primitive
    - e.g. geometric transforms
  - *attribute variables* (*attributes*) can have different values for each vertex
    - e.g. vertex coordinates, texture coordinates
  - many things can be uniforms or attributes
    - e.g. color / material properties, normal vectors

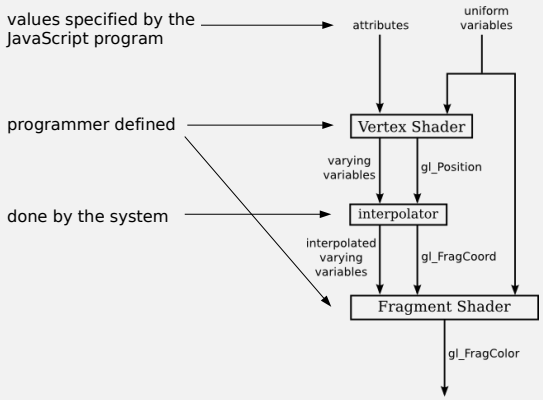
there are no predefined or required uniforms/attributes – WebGL just passes the values to the shaders to use as they wish

Pipeline Sequencing

bold indicates the programmer's responsibility

- **JavaScript program specifies the geometry and draws the primitives**
  - **sets the values for attributes and uniforms used by the shaders**
    - **for attributes, an array of values (one for each vertex)**
    - **for uniforms, a single value**
  - **draws the primitives**
- system calls vertex shader once for each vertex
  - uniforms and attributes for that vertex are passed as input to the shader
- **vertex shader (GLSL)**
  - **transforms vertex coordinates from OC → CC**
  - **computes values for properties used by fragment shader (e.g. color)**
  - **sets `gl_Position` to the coordinates of the vertex in CC**
- system clips everything outside the view volume
- system rasterizes the primitive, calling the fragment shader for each pixel with interpolated values for vertex properties
- **fragment shader (GLSL)**
  - **sets `gl_FragColor` to the color of the pixel**

Pipeline Data Flow



Web Page Structure

```
<html>

<head>
  <title>Hello WebGL</title>

<body>
  <h2>Hello WebGL/h2>

    <div id="canvas-holder">
      <canvas width="500" height="500" id="webglcanvas"></canvas>
    </div>
</body>

</html>
```

Getting a WebGL Graphics Context

```
<head>
  <title>Hello WebGL</title>

  <script>
   "use strict";

   let canvas;   // DOM object for the canvas
   let gl;       // WebGL graphics context

   // initialize and draw when page is loaded
   window.onload = function () { init(); draw(); };

   // basic application initialization
   function init() {
     try {
       canvas = document.getElementById("webglcanvas");

       let options = {
         alpha: false,      // canvas is opaque - page background won't show through
         depth: true,       // allocate a depth buffer - necessary for 3D graphics
         antialias: true    // can improve image quality but also increases computatio
       };
       gl = canvas.getContext("webgl");
       if (!gl) {
         throw new Error("WebGL not supported; can't create graphics context.");
       }

       initGL();

     } catch (e) {
       document.getElementById("canvas-holder").innerHTML =
         "<p>" + e.message + "</p>";
       return;
     }
   }

   // initialize the webgl graphics context
   function initGL() {
   }

   // draw the canvas
   function draw() {
   }
  </script>
</head>
```

`use strict` disallows certain "sloppy syntax", helping to prevent errors – use it!

`webglcanvas` and `canvas-holder` are names of elements declared in the <body> section

options –
– `alpha` true (default) allows transparent canvas pixels; safe to make false (does not block alpha blending of drawing color with image color)
– `depth` true (default) enables depth buffer
– `antialias` true (default) requests antialiasing; nicer image but increased computation time
– `preserveDrawingBuffer` false (default) discards contents of drawing buffer after display; only need true if drawing is constructed incrementally