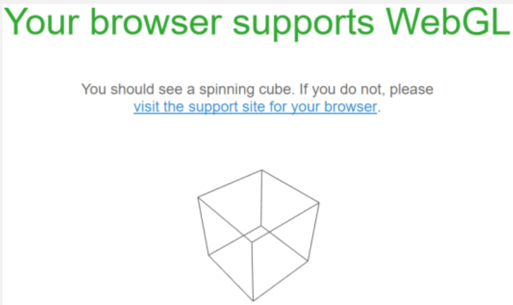
P.PORTO

POLITÉCNICO
DO PORTO
ESMAD

COMPUTAÇÃO GRÁFICA
TSIW

WebGL

- New **standard** for 3D graphics on the web
- It appeared in 2006 as *Mozilla Canvas3D*
- Joined Opera, Apple and Google: renamed to ***WebGL***
- Developed and maintained by the Khronos group (khronos.org/webgl)
- Part of the **HTML5** family of technologies
  - Not an official W3C specification
  - Supported by most browsers with HTML5 support (get.webgl.org / webglreport.com)
- Graphics are rendered using **JavaScript**
- Features are downloaded by the browser, but the **code runs directly on the GPU**



WebGL

JavaScript API for rendering interactive 2D and 3D graphics inside an HTML <canvas> element

WebGL

WebGL - 3D Canvas graphics - OTHER

Usage % of all users ?

Global 97.97%

Method of generating dynamic 3D graphics using JavaScript, accelerated through hardware

Current aligned | Usage relative | Date relative | Filtered | All

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on iOS *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
		2-3.6	4-7	3.1-5	10-11.5											
	[1] 12-18	[1] 4-23	[1] 8-32	[1] 5.1-7.1	[1] 12.1-18	3.2-7.1										
6-10	79-95	24-94	33-95	8-15	19-81	8-14.8		2.1-4.4.4	12-12.1				4-14.0			
[1] 11	96	95	96	15.1	82	15.1	all	96	64	96	94	[1] 12.12	15.0	10.4	7.12	[1] 2.5
		96-97	97-99	TP												

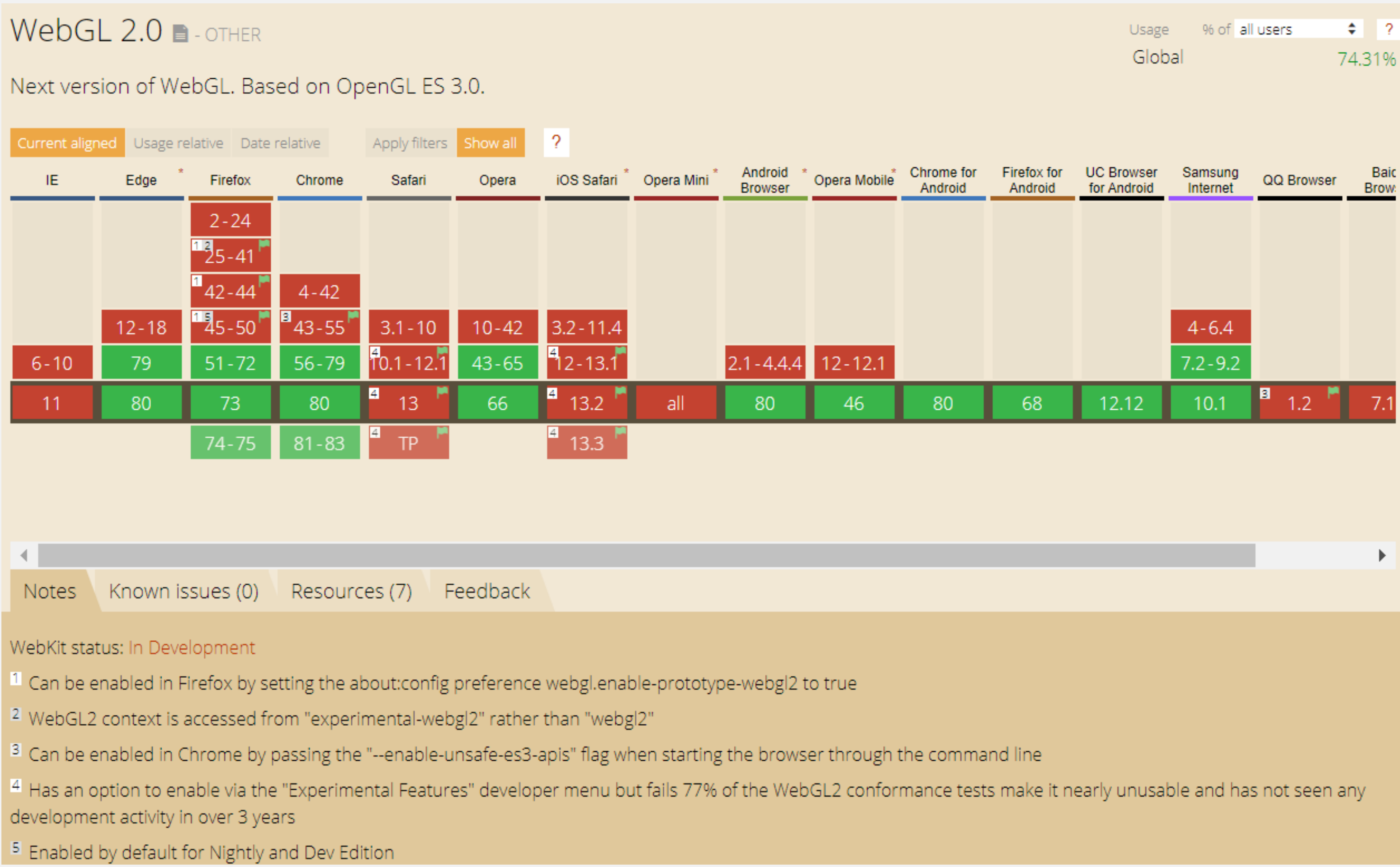
Notes | Test on a real browser | Known issues (1) | Resources (9) | Feedback

WebGL support is dependent on GPU support and may not be available on older devices. This is due to the additional requirement for users to have [up to date video drivers](). Note that WebGL is part of the [Khronos Group](), not the W3C.

[1] WebGL context is accessed from "experimental-webgl" rather than "webgl"

https://caniuse.com/#feat=webgl

WebGL 2.0



https://caniuse.com/#feat=webgl2

https://registry.khronos.org/webgl/specs/latest/2.0/#webgl_gl_differences

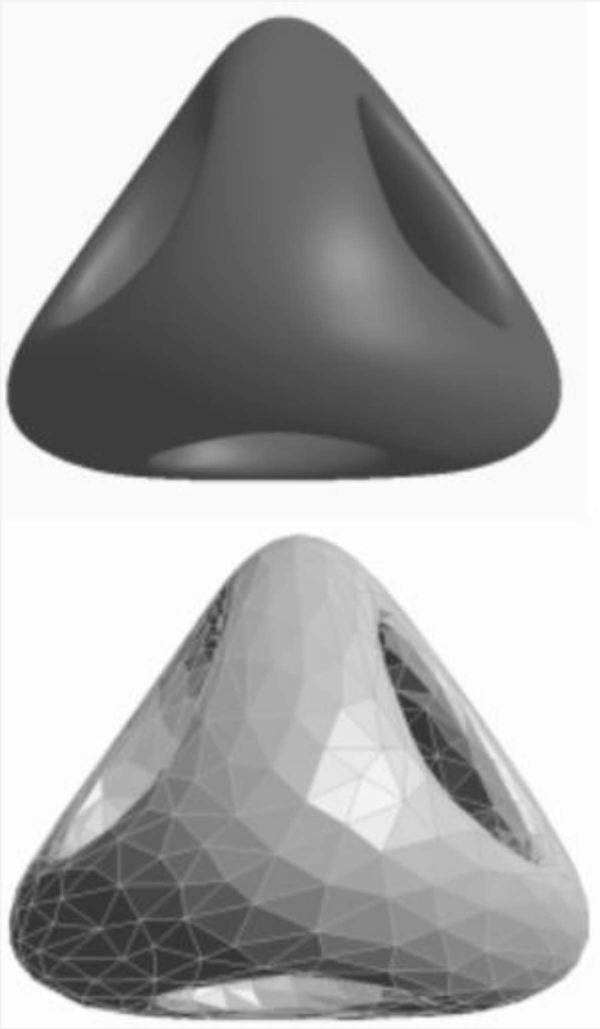
3D Graphics

- **Modelling**
  - How to represent objects
  - How to build those representations
- **Animation**
  - Controlling the way things move
- **Rendering**
  - How to display a scene

3D Modelling

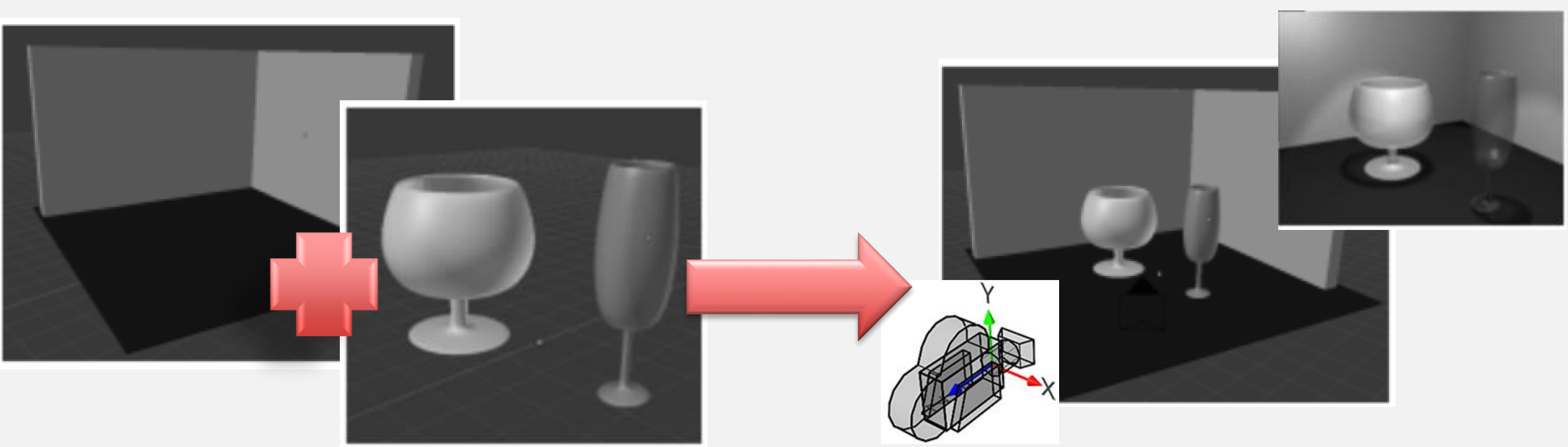
- **3D models**
  - Point meshes
  - Curves and surfaces

- But when data is sent to the graphics card
  - everything is transformed into **triangles**



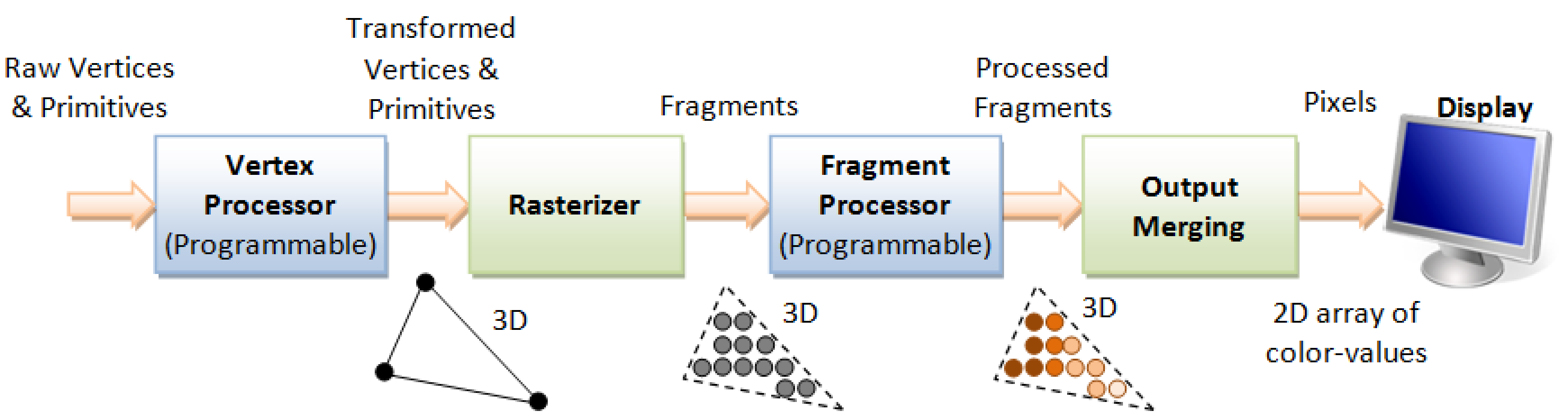
3D Rendering

- Geometric transformations to the vertices
  - To **position the models** and **create a 3D scene**
  - The pretended **view of the scene** is chosen by **camera** positioning
  - **Projection** from 3D to 2D
- Triangle painting
- Lighting and materials



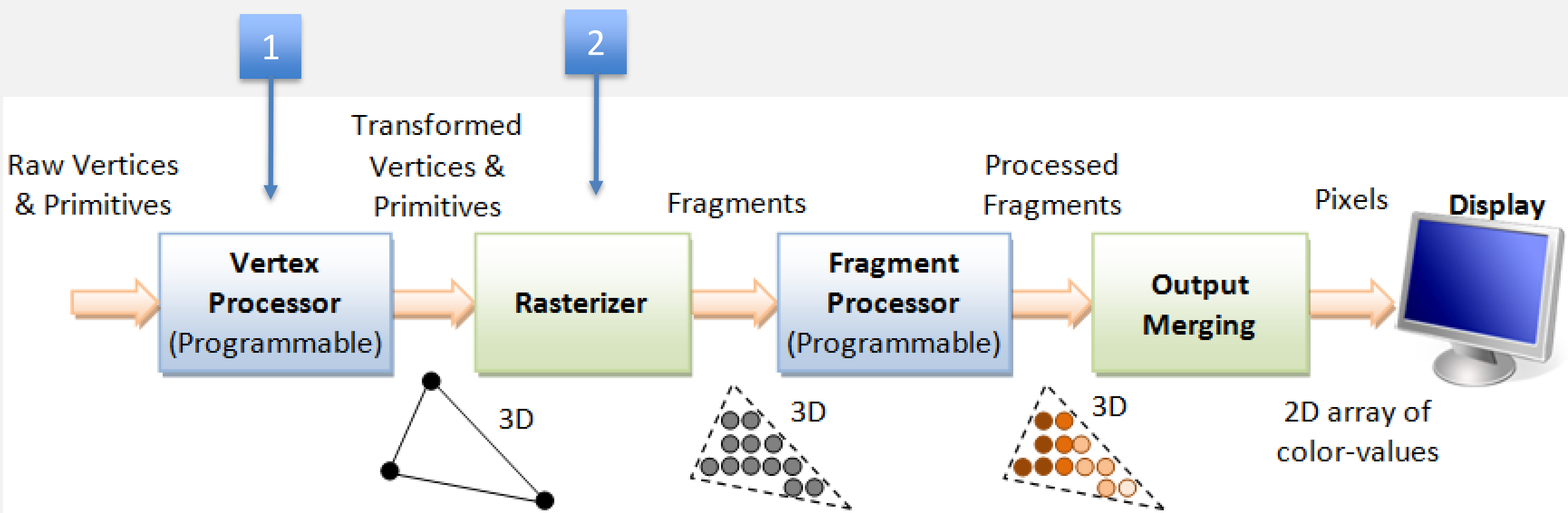
3D rendering pipeline

- **Rendering Pipeline**: series of processing stages in order to produce an image on the display from a 3D model description
  - accepts description of 3D objects in terms of vertices of primitives (such as triangles, points, lines, …)
  - produces the color-value for the pixels on the display



3D rendering pipeline

- Main stages
  1. **Vertex Processing**: process and transform individual vertices
  2. **Rasterization**: convert each primitive (**connected vertices**) into a set of fragments; a fragment can be treated as a pixel in 3D space, which is aligned with the pixel grid, with attributes such as position, color, normal and texture

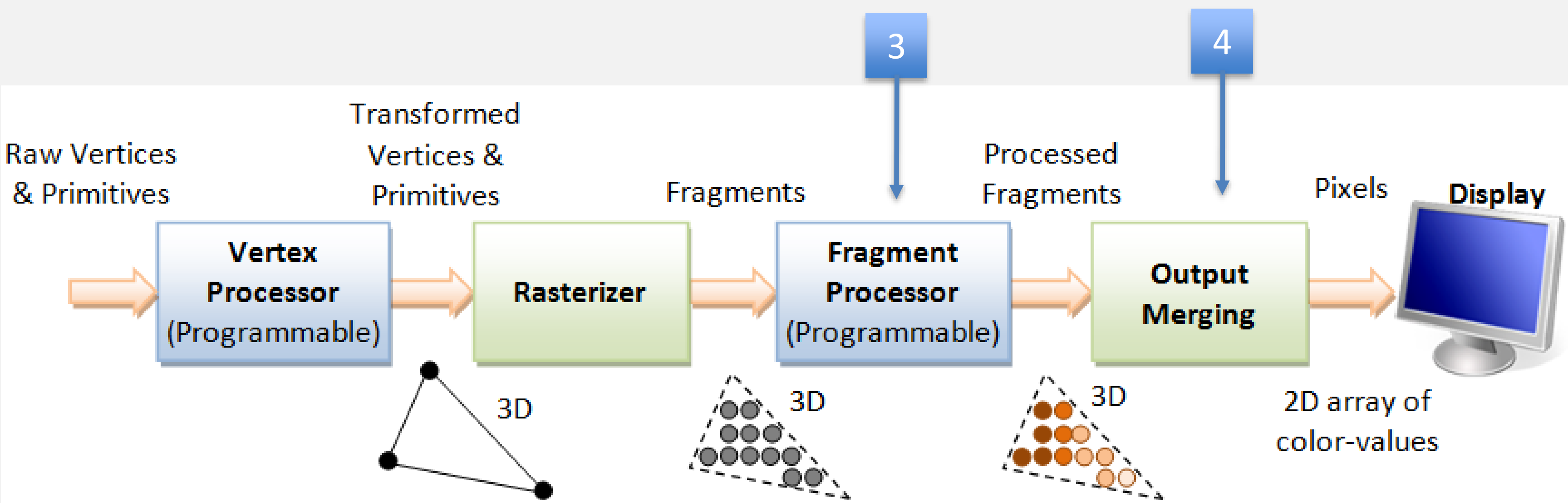


3D rendering pipeline

- Main stages

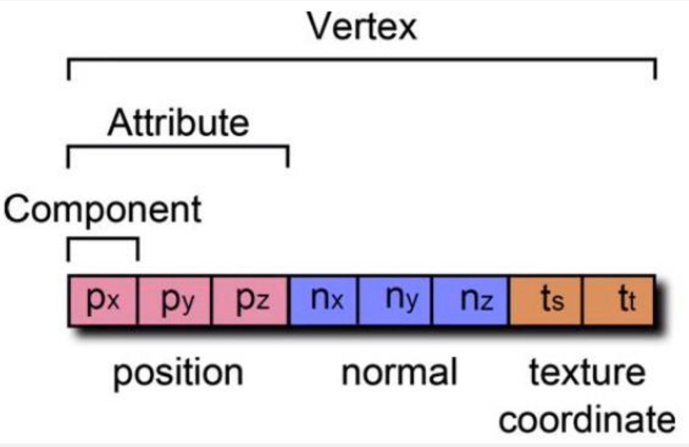
  (...)

  3. **Fragment Processing**: process (color) individual fragments
  4. **Output Merging**: combine the fragments of all primitives (in 3D space) into 2D color-pixel for the display



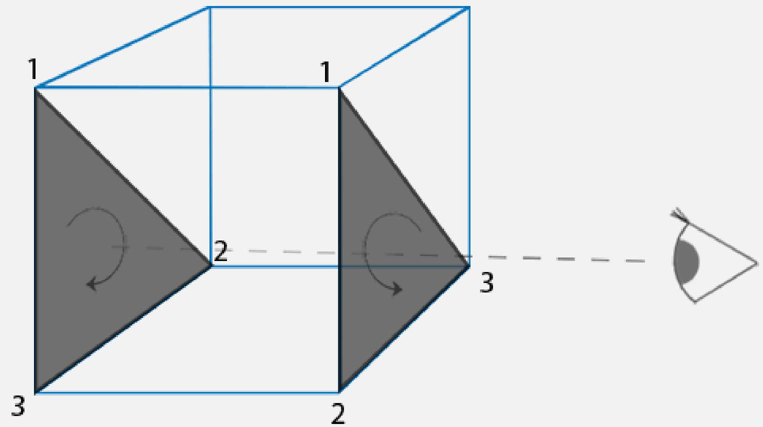
Concepts

- **Vertex**:
  - **Position** in 3D space, $P=(p_x, p_y, p_z)$: typically expressed in floating point numbers
  - **Color**: expressed in RGB or RGBA components; values are typically normalized [0.0 ; 1.0] (or 8-bit unsigned integer [0; 255]); alpha (A) is used to specify the transparency, with alpha of 0 for totally transparent and alpha of 1 for opaque
  - **Vertex-Normal** $N=(n_x, n_y, n_z)$: used to differentiate the front- and back-face, and for other processing steps such as lighting
  - **Texture** $T=(t_s, t_t)$: in computer graphics, often a 2D image is wrapped to an object to make it seen realistic; therefore, the texture coordinates provides a reference point to a 2D texture image



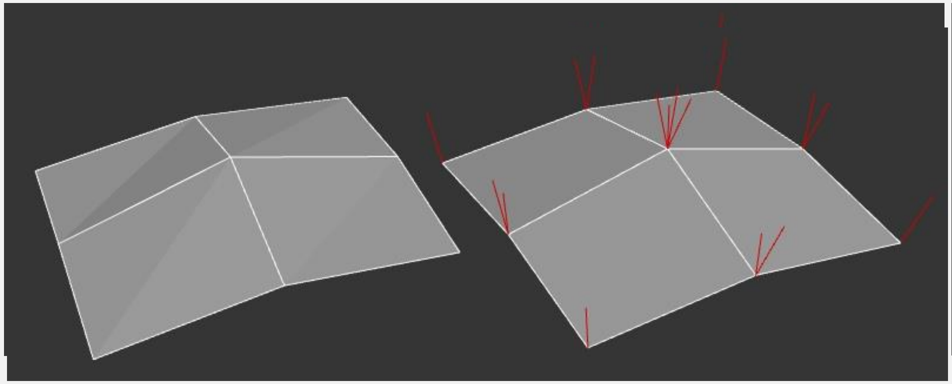
Concepts

- **Normals**:
  - **Face (polygon or triangle) normal**:
    A triangle is defined by its vertices - the corner points. Each of those points is referenced by a number, and the order of those points defines the direction in which the face normal is orientated.
    WebGL uses the face normal, to determine what is the inside and what is the outside of an object. By default, triangles defined with counter-clockwise vertices are processed as front-facing triangles. It does not render the backfaces unless explicitly told to do so.

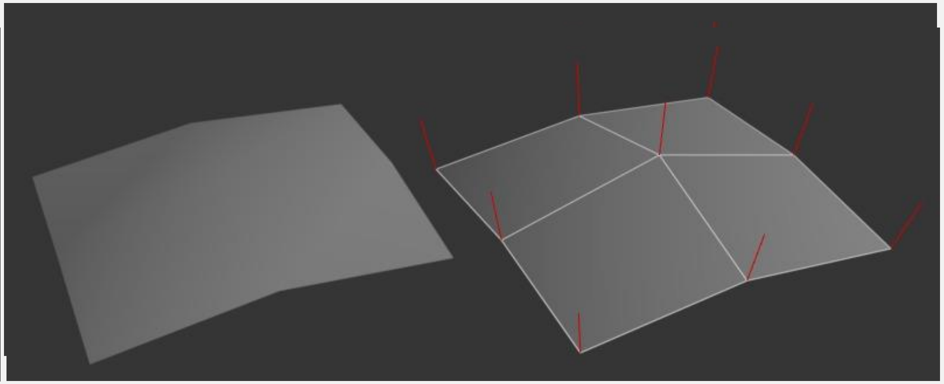


Concepts

- **Normals**:
  - **Vertex normals**:
    Are a continuation of the basic principle of face normals. Each vertex normal is a vector pointing in some direction. In the case of a single triangle, all the vertex normals point in the same direction as the face normal - unless explicitly changed. This is not particularly useful for a single triangle, but it gets interesting if we have neighboring polygons.



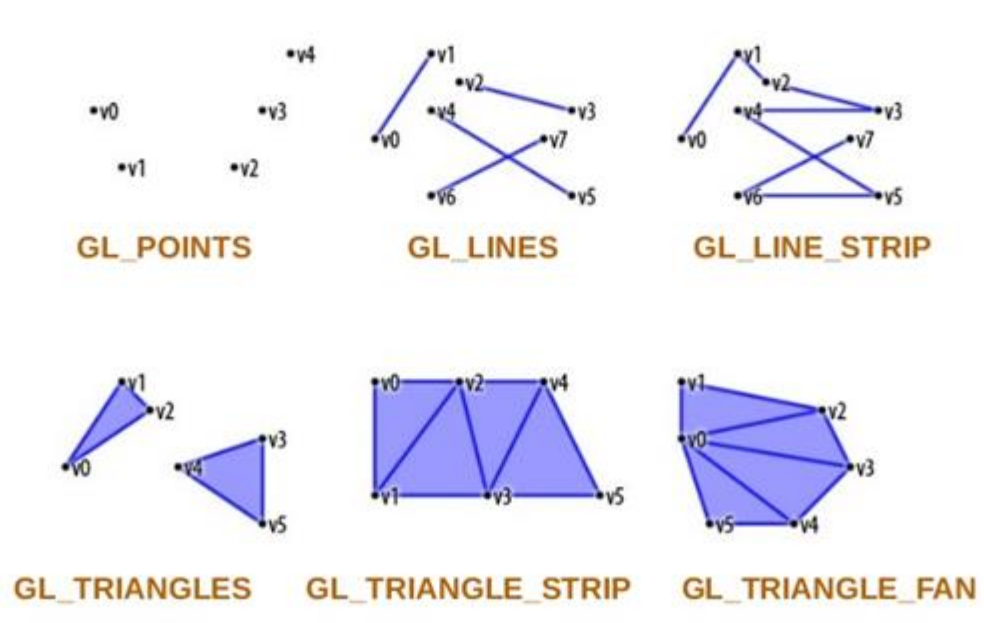
vertex normals in a multi-polygon mesh



vertex normals in a smoothed mesh

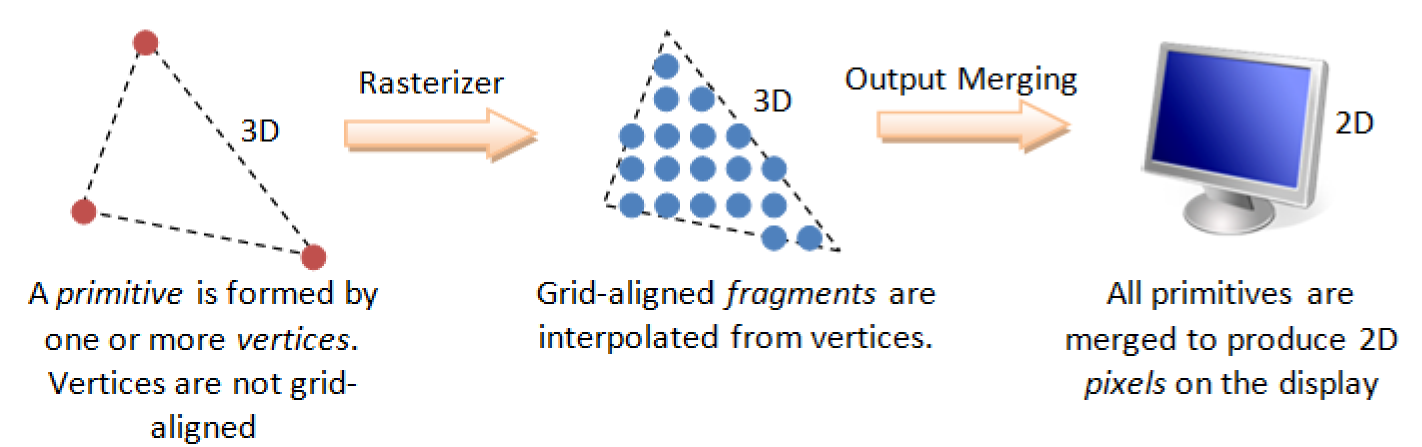
Concepts

- **Primitive**:
  - Formed by of **one or more vertices**
  - OpenGL supports three **classes** of geometric primitives: points, line segments, and closed polygons
  - **Primitive assembly** groups vertices forming one primitive
  - Primitives often **share** vertices
  - Instead of repeatedly specifying the vertices, it is more efficient to create an index list of vertices, and use the indexes in specifying the primitives



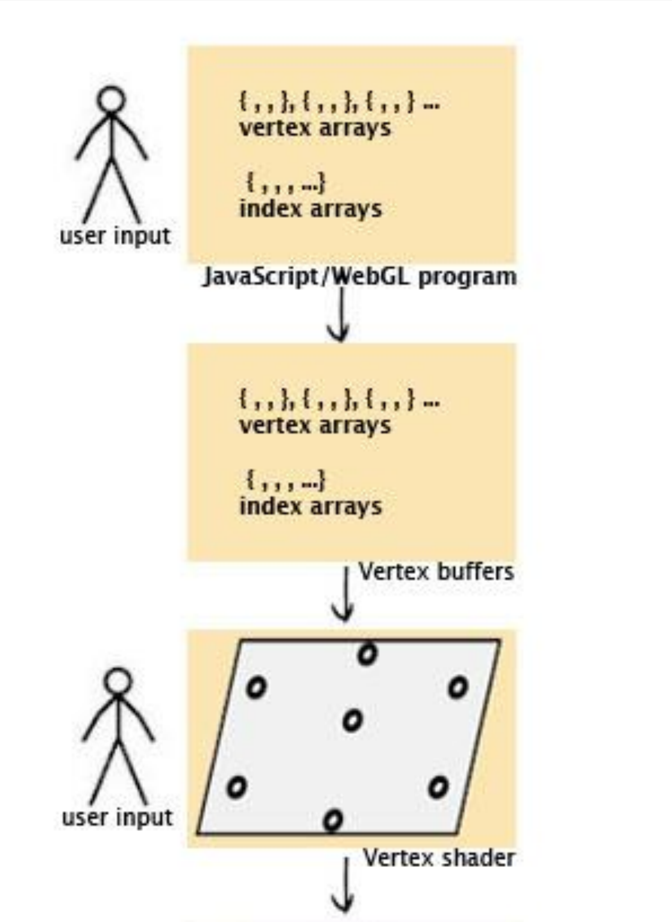
3D rendering pipeline

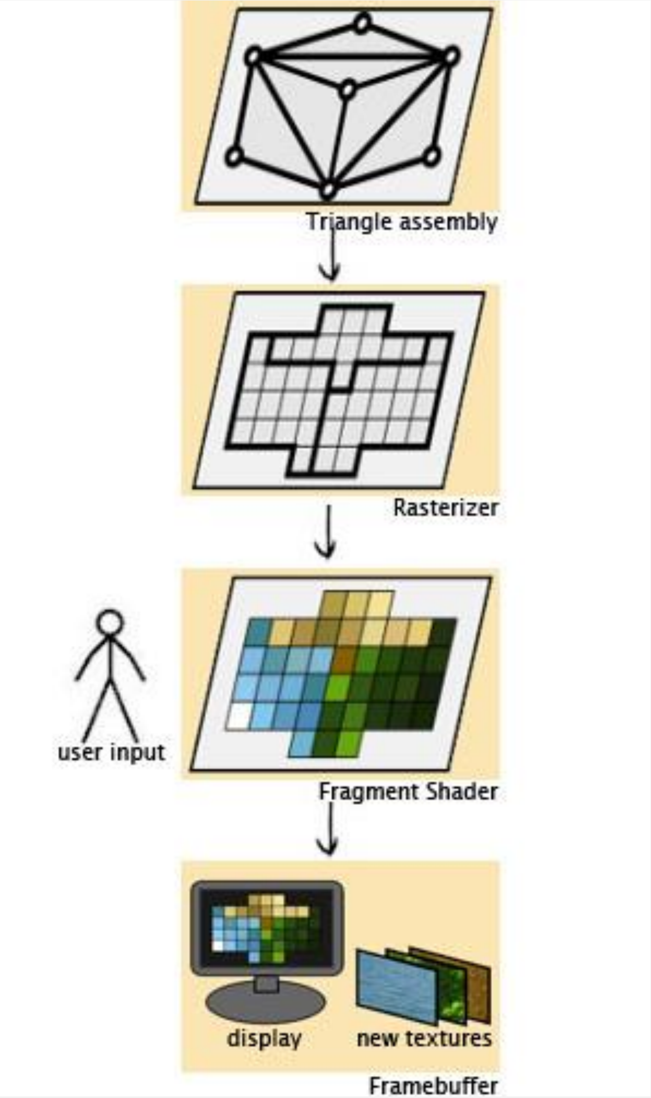
- **Fragment**
  - **Pixels** refers to the dots on the display, which are aligned in a 2D grid of a certain rows and columns corresponding to the display's resolution
  - In order to produce the **grid-aligned pixels** for the display, the rasterizer takes each input primitive and perform raster-scan to produce a set of grid-aligned fragments enclosed within the primitive
  - A **fragment** is 3D, with a `(x,y,z)` position: the `(x,y)` are **aligned** with the 2D pixel-grid and the `z`-value (not grid-aligned) denotes its depth



Vertex, Primitives, Fragment and Pixel

WebGL pipeline





WebGL pipeline

- All WebGL programs must:
  - Configure the Canvas element to run WebGL
  - Create shader programs
  - Generate geometric data in the application
  - Create buffer objects where to place the geometric data
  - Interconnect the data with the variables of the shaders (*shader plumbing*)
  - Perform object rendering
- Open sample program available at Moodle (WebGL - 2D triangle)

WebGL example - triangle

- Configure the Canvas element to run WebGL

```
// Gets 3D Canvas context
var canvas = document.getElementById('gl-canvas');

// JavaScript utilities for common WebGL tasks (checks for success or failure)
gl = WebGLUtils.setupWebGL(canvas);
if (!gl) { alert("WebGL not available"); }

// Sets WebGL viewport (same size as Canvas element)
gl.viewport(0, 0, canvas.width, canvas.height);
// Sets background color
gl.clearColor(0.9, 0.9, 0.8, 1.0);
```

WebGL example - triangle

- Create **shader** programs - vertex and fragment shaders – in GLSL language (*OpenGL Shading Language*)
- **Vertex shader**:
  - Executed in parallel for each vertex sent to the GPU
  - Used to transform geometry
  - Uses **attributes** (per-vertex input variables) to access information contained in **Vertex Buffer Objects**

```
<script id="vertex-shader" type="x-shader/x-vertex">
    attribute   vec4    vPosition;

    void main()
    {
        gl_Position=vPosition;
    }
</script>
```

WebGL example - triangle

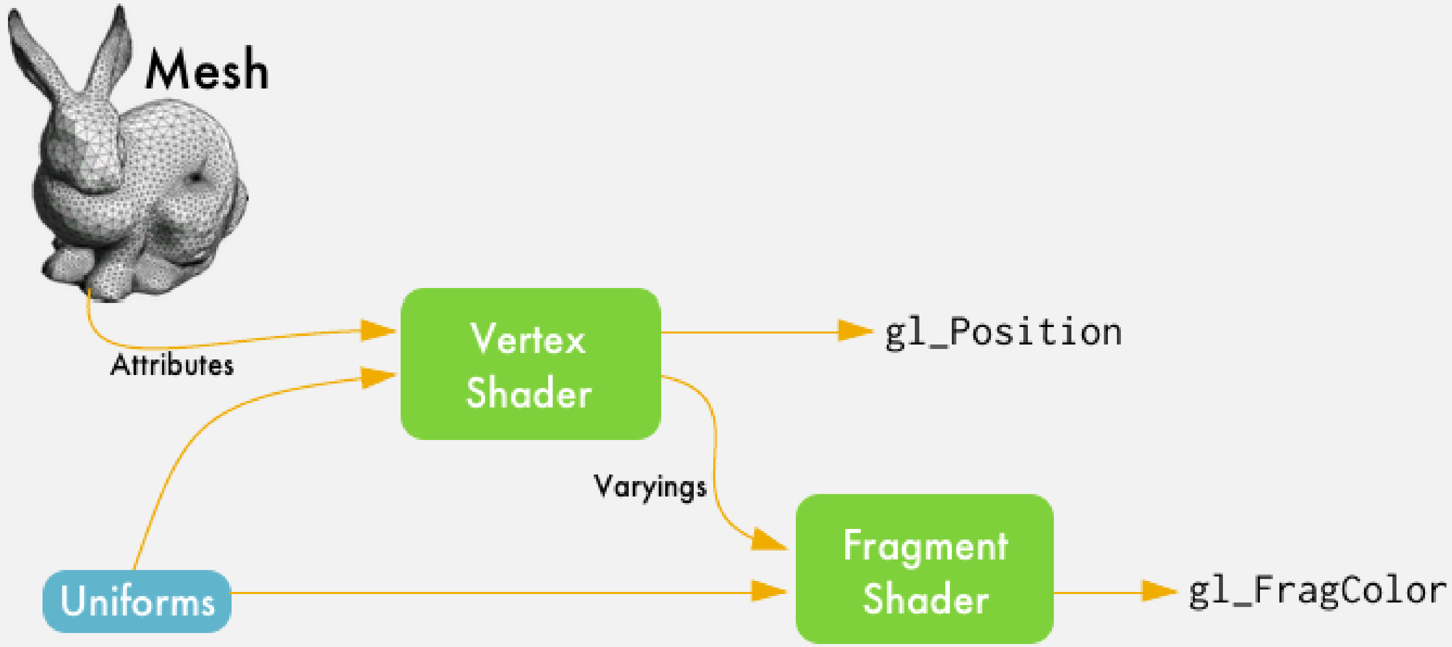
- **Vertex shader**:
  - Once the vertex coordinates have been processed in the vertex shader, they should be in NDC: *Normalized Device Coordinates*, meaning all x, y and z values vary from -1.0 to 1.0 (any coordinates that fall outside this range will be discarded/clipped and won't be visible on your screen)
  - The NDC coordinates will then be transformed to screen-space coordinates via the viewport transform using the data provided in `glViewport`
  - In the provided example, all vertices are already provided in NDC, therefore no transformations need to be performed

```
<script id="vertex-shader" type="x-shader/x-vertex">
    attribute   vec4    vPosition;

    void main()
    {
        gl_Position=vPosition;
    }
</script>
```

WebGL shaders

- **Shader variables**
  - **Attributes**: exclusive data from each vertex
  - **Uniforms**: common data for all vertices (such as transformations, texture coordinates,…)
  - **Varyings**: used to pass data from the vertex shader to the fragment shader



WebGL example - triangle

- **Fragment (pixel) shader**:
  - Written to calculate the color for each of one of the pixels of each fragment
  - Used for operations with interpolated values, access to textures, color composition, ...
  - In the provided example, all pixels are colored with a fixed opaque color (blue – RGB(0,0,1))

```
<script id="fragment-shader" type="x-shader/x-fragment">
    precision mediump float;

    void main()
    {
        gl_FragColor=vec4(0.0, 0.0, 1.0, 1.0);
    }
</script>
```

WebGL example - triangle

- Compile **shader** programs

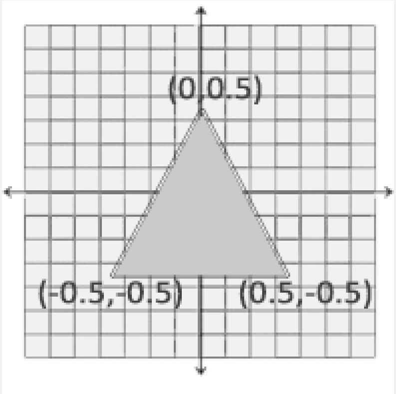
```
// Compiles both vertex and fragment shaders in GPU
var program = initShaders(gl, "vertex-shader", "fragment-shader");
gl.useProgram(program);
```

WebGL example - triangle

- Generate geometric data in the application

```
var vertices = new Float32Array([
    -0.5, -0.5, // Triangle vertex 0
    0, 0.5,     // Triangle vertex 1
    0.5, -0.5   // Triangle vertex 2
]);
```

```
  Triangle in plane Z=0:
vertex 0 (-0.5, -0.5, 0)
vertex 1 ( 0.0,  0.5, 0)
vertex 2 ( 0.5, -0.5, 0)
```



WebGL example - triangle

- Create buffer objects where to place the geometric data
  - In the example, *vertices* array data is binded to the buffer array

```
// Uploads data into GPU
var bufferId = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, bufferId);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
```

- Interconnect the data with the shaders variables (*shader plumbing*)
  - In the example, shader attribute variable *vPosition* is associated with data

```
// Links shader variables to data buffers
var vPosition = gl.getAttribLocation(program, "vPosition");
gl.vertexAttribPointer(vPosition, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(vPosition);
```

Vertex array size (2D)

WebGL example - triangle

- Perform object rendering

```
    // Object rendering
    render();
};

function render() {
    gl.clear(gl.COLOR_BUFFER_BIT);
    // Draw data as triangle primitives
    gl.drawArrays(gl.TRIANGLES, 0, 3);
}
```

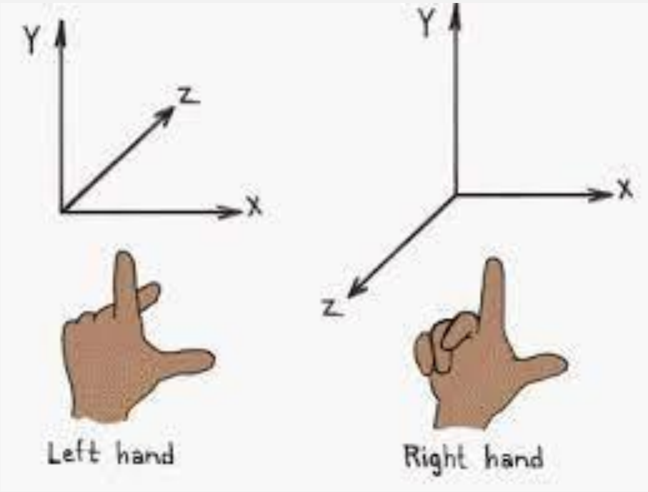
Primitive

Index of the first vertex

Number of vertices (3)

WebGL – Coordinate Systems

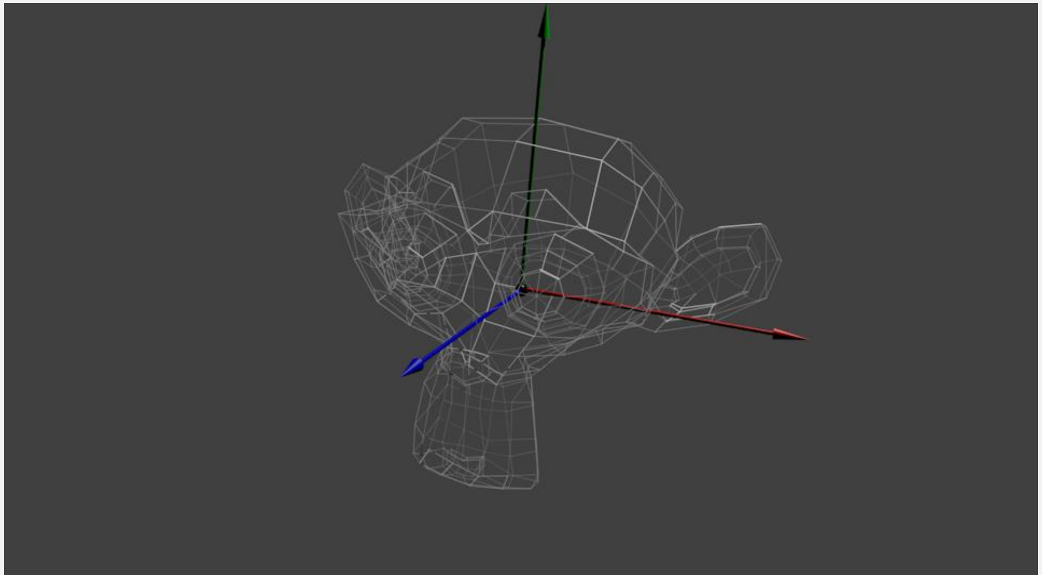
- A three-dimensional coordinate system (CS) can be right-handed or left-handed



- WebGL / OpenGL convention is to assume **right-handed CS**
  - the Z axis points out, toward the viewer

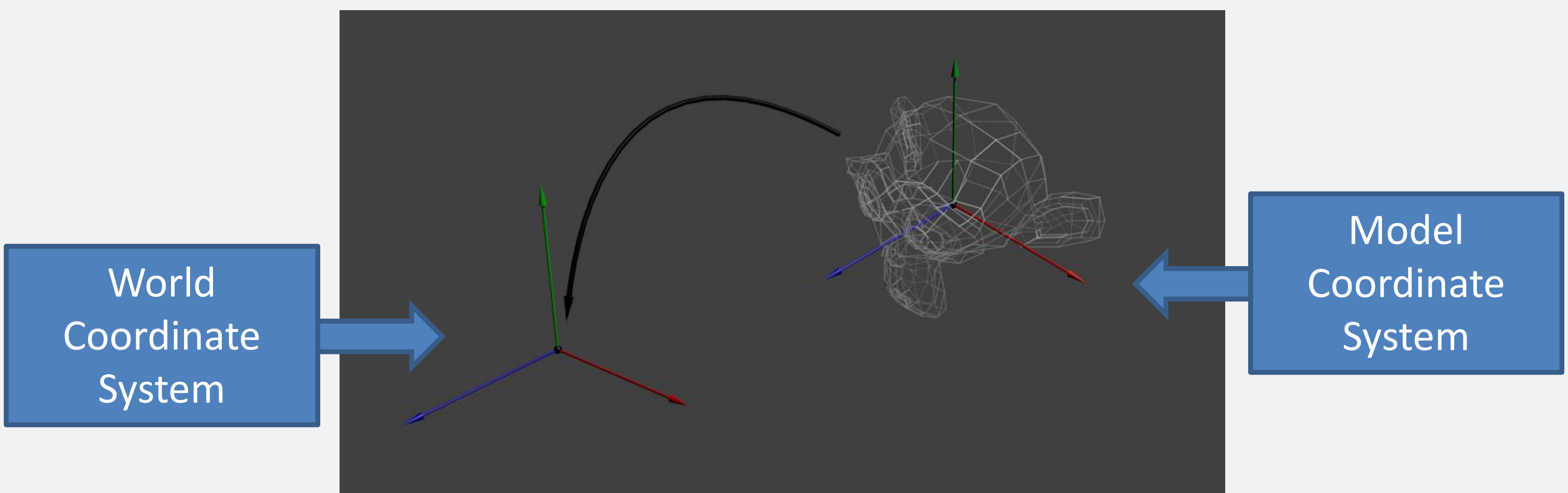
WebGL – Coordinate Systems

- **Model CS**: used to draw an object (e.g. in a 3D modelling tool)
  - Axis are centred in the model point of rotation, usually its geometric center
  - This system is aligned with the model, does not matter its position in the world
  - You can use the model coordinate system to **define each modeling object** independently



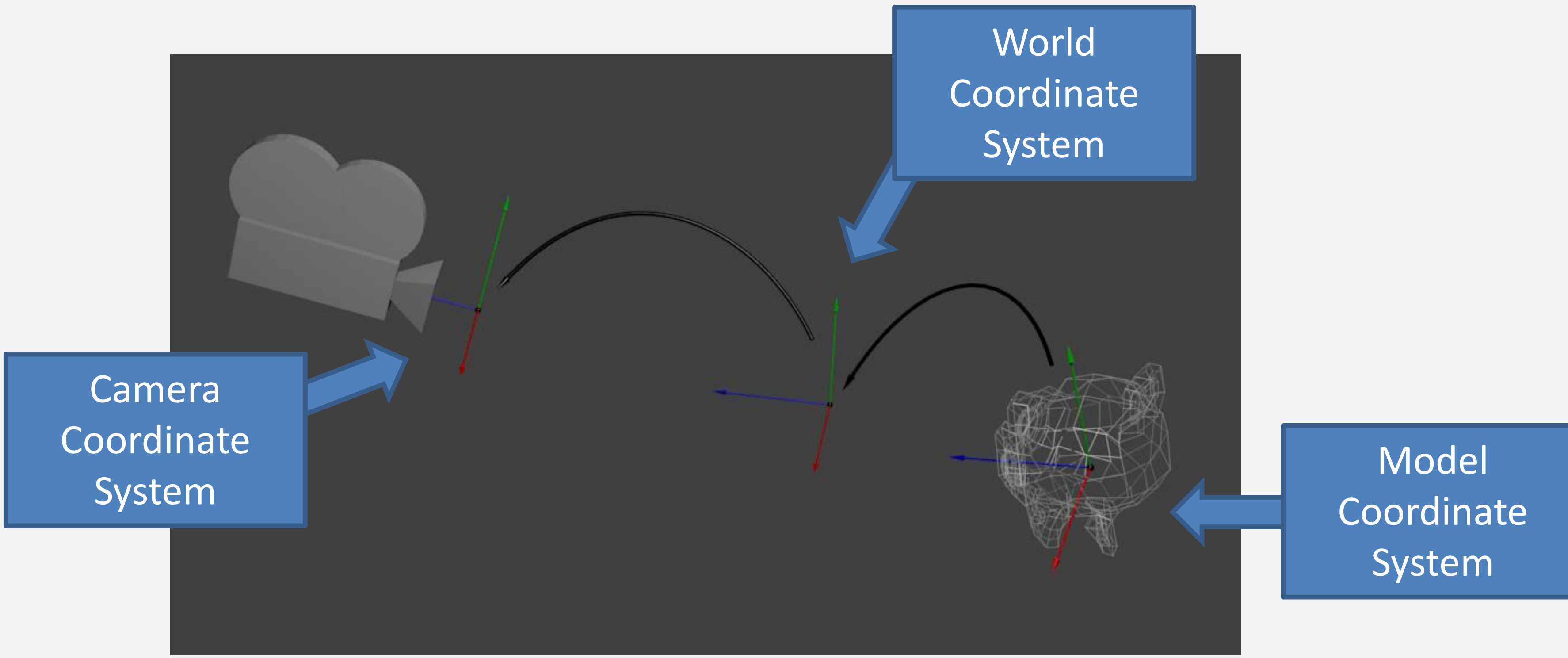
WebGL – Coordinate Systems

- **World CS**: where all objects are located (all objects will have a geometric relation between them)
  - This is the system we usually think **when objects are placed in a scene** (by object we mean 3D models, cameras or light sources)
  - In this system, we usually care about details like object's scale, their relative position, etc…

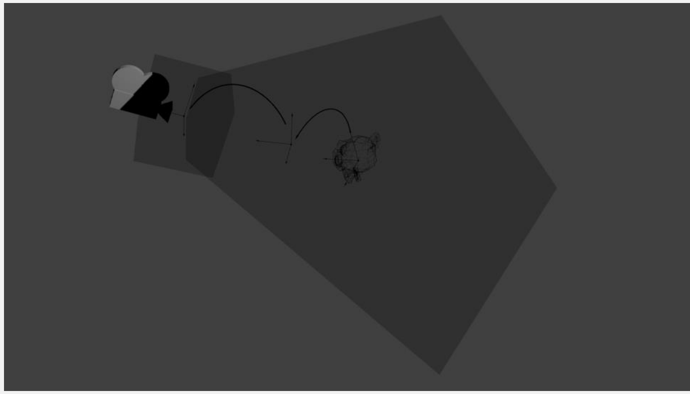


WebGL – Coordinate Systems

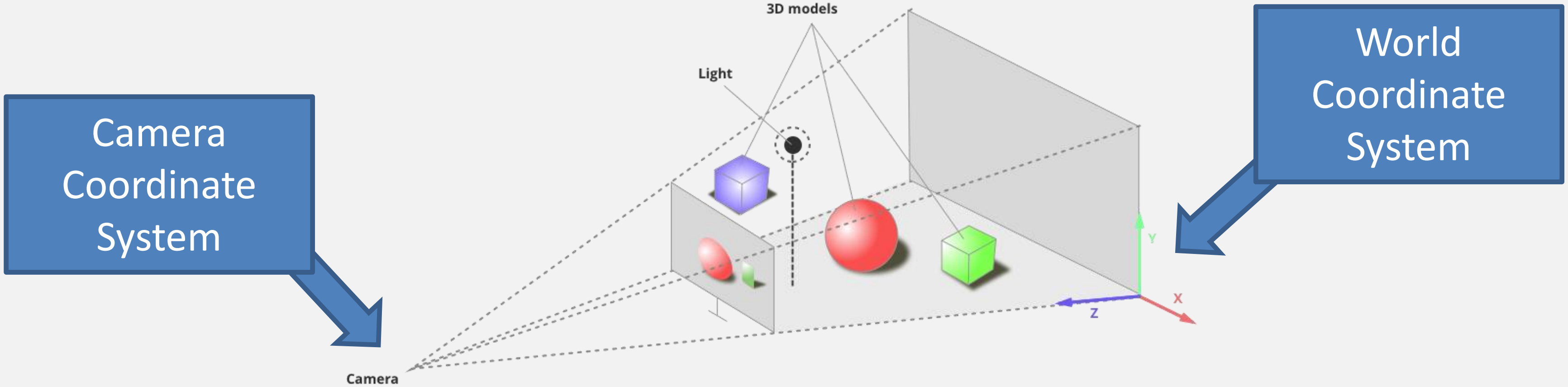
- **Camera CS**: the observer of the scene



WebGL – Coordinate Systems

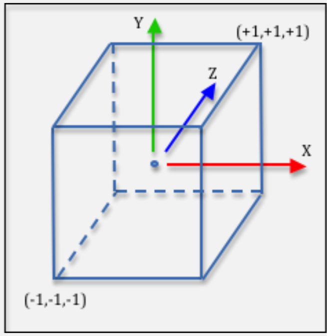


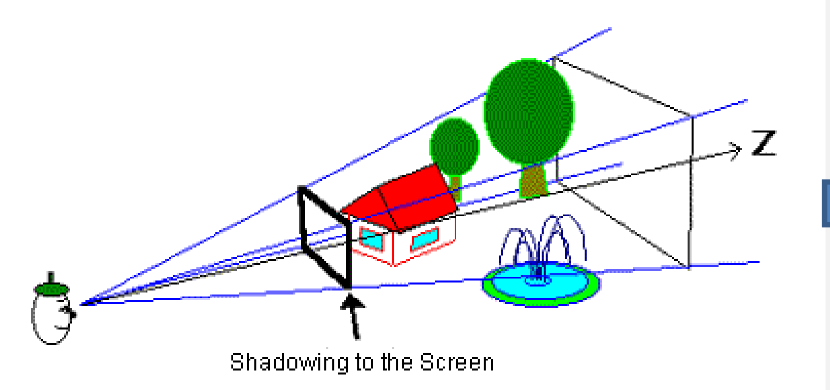
- **Camera CS**: the observer of the scene
  - System centered in a **virtual camera**: the **position** where the camera placed is perceived as the world origin (0,0,0); its viewing **direction** is perceived as the Z axis (depth)
  - This CS is what the computer graphics hardware uses to **view the virtual 3D** world set up in the computer; the range actually seen is the part that falls inside a volume called the "**view volume**"

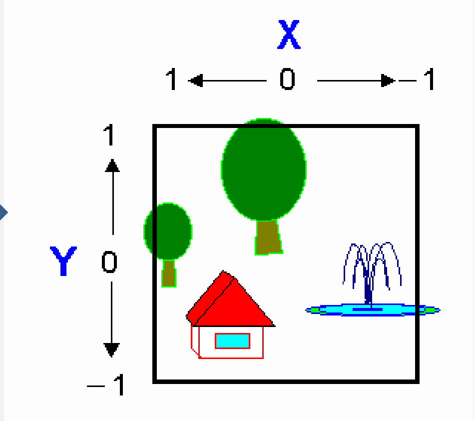


WebGL – Coordinate Systems

- **Clip CS**: defines the clipping volume
  - It marks the centre of the canvas: axis X, Y and Z vary from **–1 to 1**: coordinates in this space are also called **Normalized Clip Coordinates** (NDC)
  - Coordinates X and Y already consider the impact caused by the Z-axis transformation, due to the **projection**
  - Everything outside the **viewing volume** is not draw



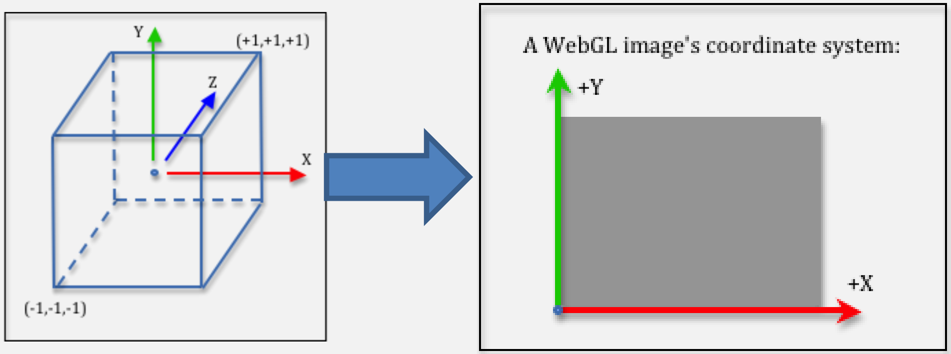


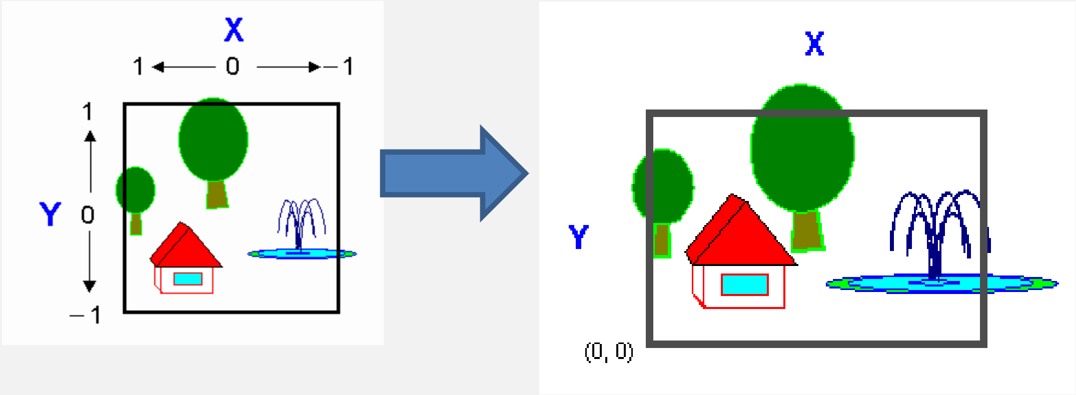


The Clip CS is the display equipment for the virtual 3D space inside your computer.
Like the camera CS, the viewing direction is the Z-axis and the center of the screen is the origin

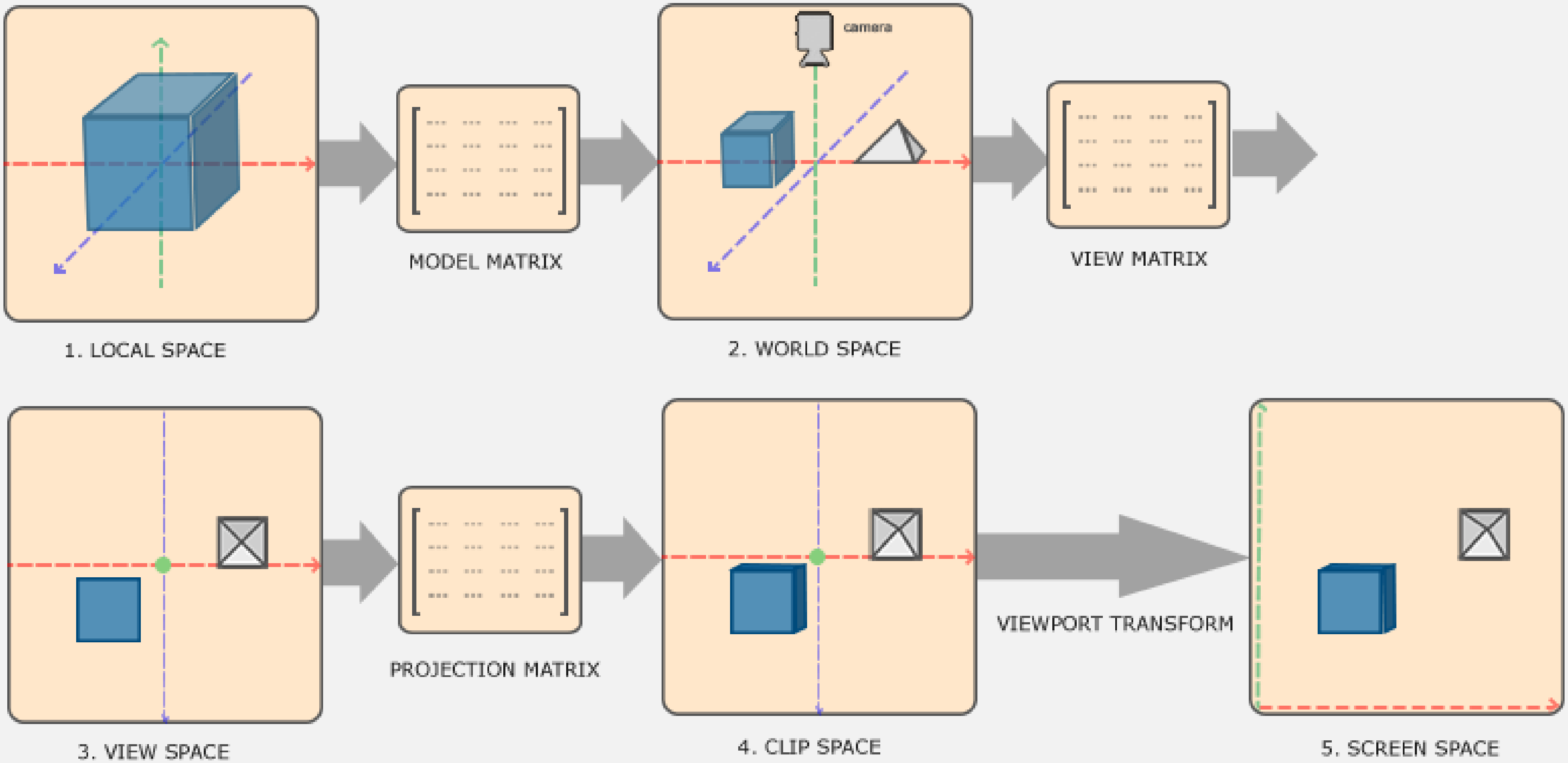
WebGL – Coordinate Systems

- **Viewport CS**: defines the 2D screen space
  - CS for the rendered 2D: the 3D normalized vertex data is mapped into 2D pixels that will compose an image that will be rendered on the HTML canvas element – called the **viewport**





WebGL - Transformations

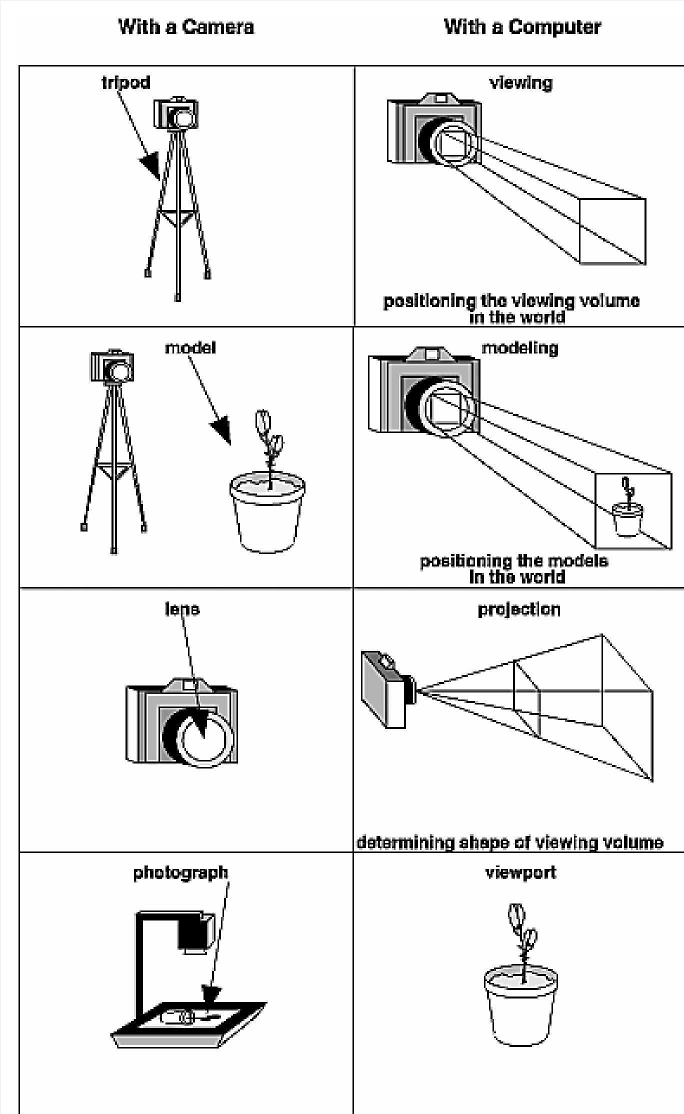


FROM: https://learnopengl.com/Getting-started/Coordinate-Systems

WebGL - Transformations

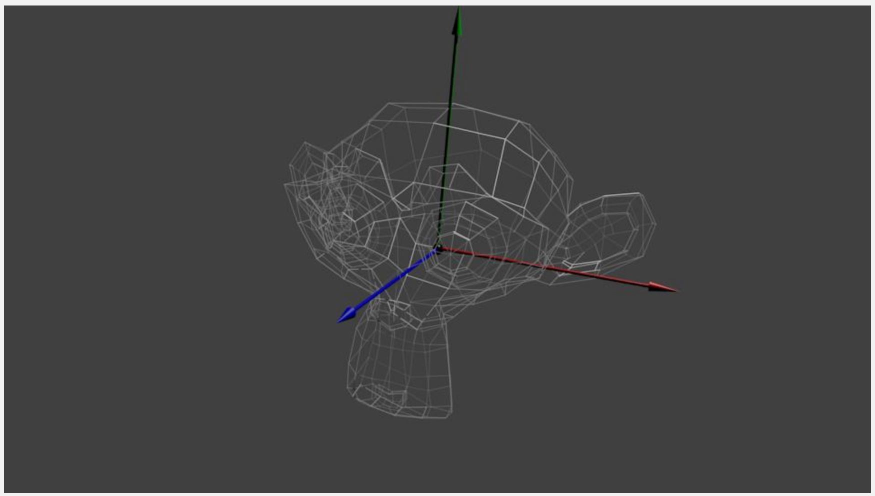
Analogy with a photographic camera

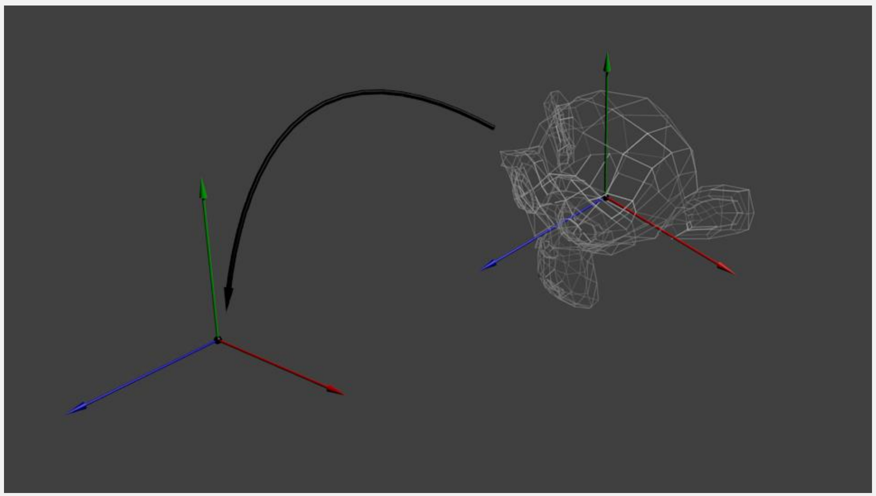
1. *Viewing transformation*: aim the camera to the scene (defines the camera CS relatively to the world CS)
2. *Modeling transformation*: compose the scene (defines the model CS relatively to the world CS)
3. *Projection transformation*: chooses the lens and sets the zoom (defines how to transform from the camera CS to the projection CS)
4. *Viewport transformation*: determines the scene size



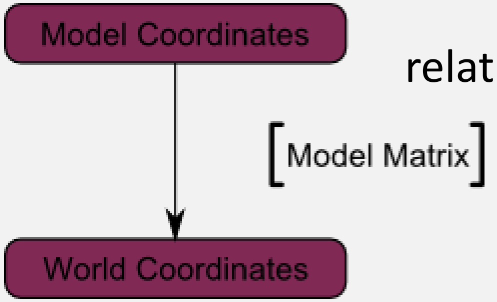
WebGL - Transformations

Model matrix: transform from Model to World coordinates





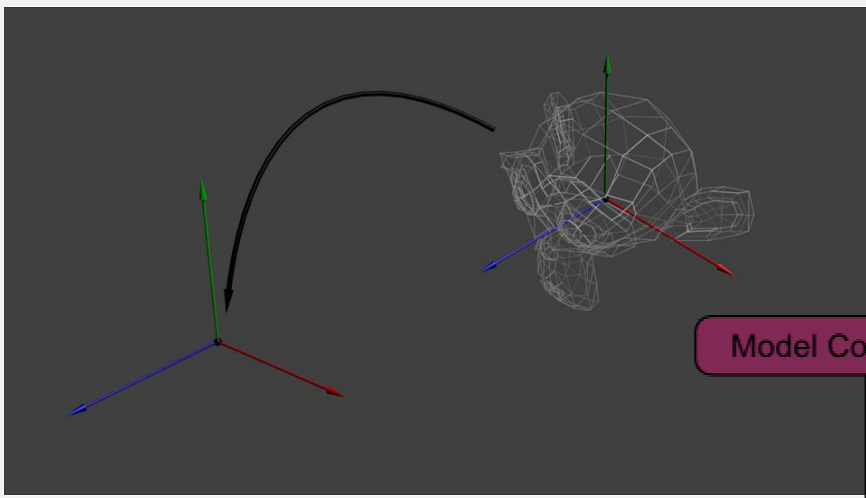
all vertices are defined relatively to the center of the **model**



all vertices are defined relatively to the center of the **world**

WebGL - Transformations

View matrix: transform from World to Camera (or Eye) coordinates



all vertices are defined relatively to the center of the **world**

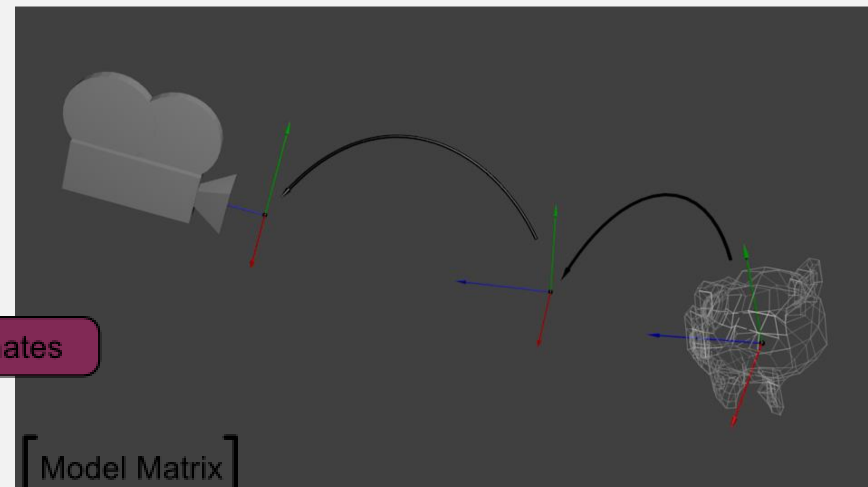
Model Coordinates

↓ [Model Matrix]

World Coordinates

↓ [View Matrix]

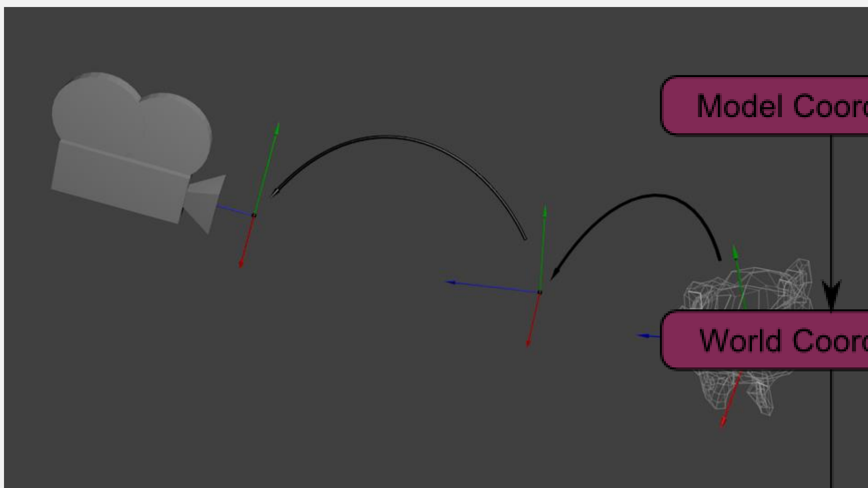
Camera Coordinates

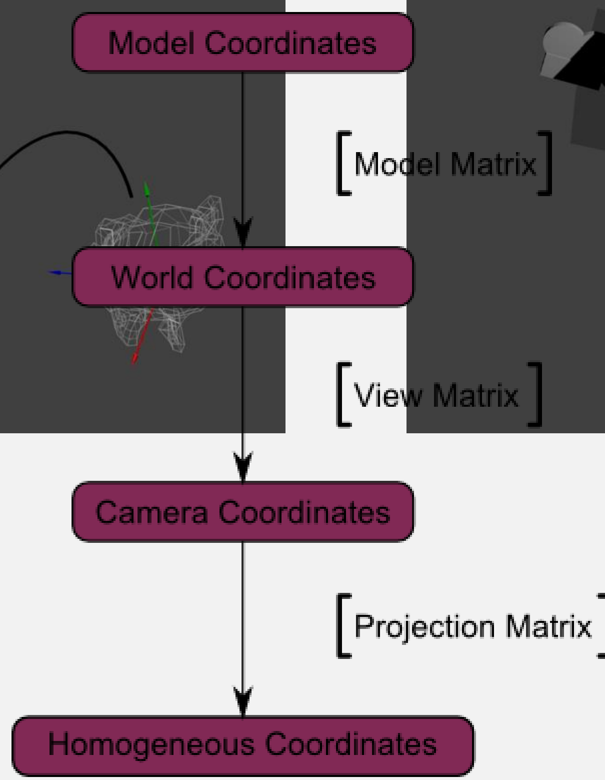


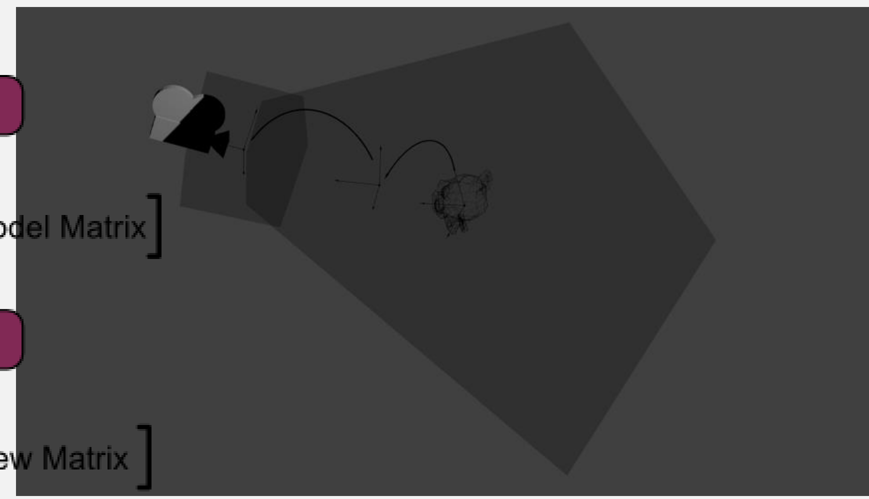
all vertices are defined relatively to the **camera**

WebGL - Transformations

Projection matrix: transform from Camera to Projection (or Clip) coordinates



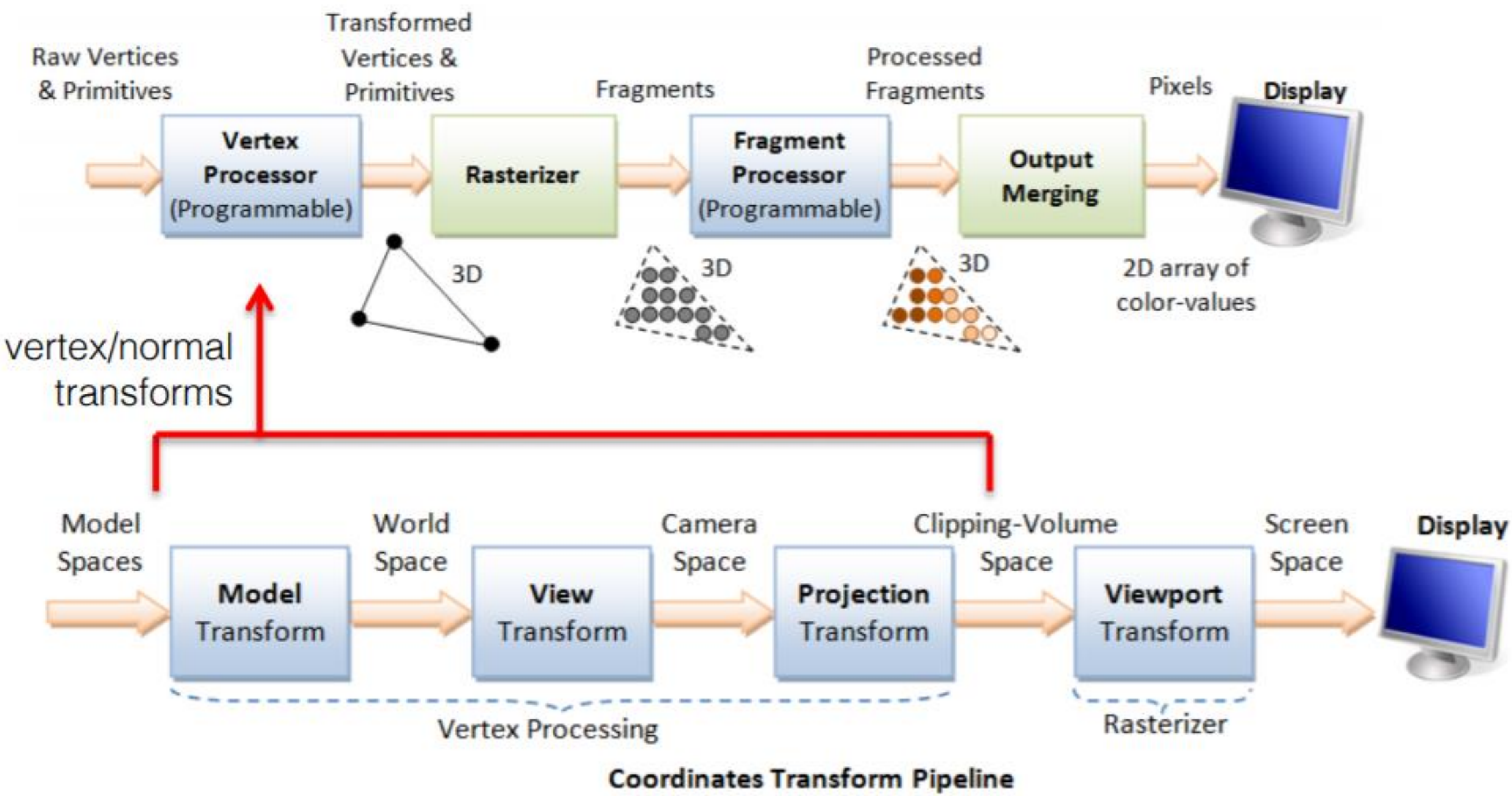




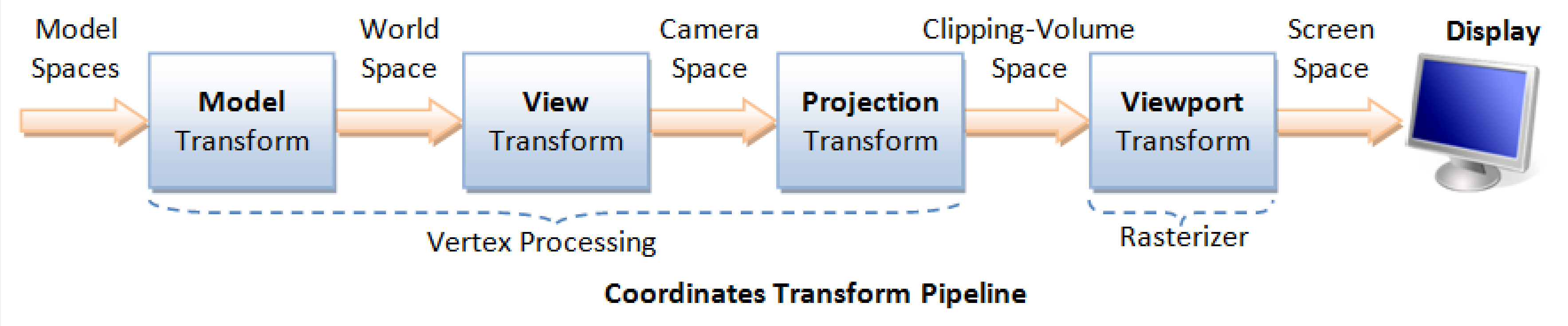
all vertices are defined relatively to the **camera**

- all vertices are defined in a small volume (**view frustum**)
- everything inside that volume is **visible**

Coordinates Transformation



Coordinates Transformation



As we saw, WebGL is a rasterization tool: it starts from coordinates defined in a 3D space to a result on pixels a 2D screen. To do so, vertices are multiplied by several matrices:

1. Arrange the objects (or models) in the world (**Model matrix**)
2. Position and orientation the camera (**View matrix**)
3. Select a camera type (**Projection matrix**)
4. Display the image on a selected area of the screen (**Viewport transformation**) - in rasterization stage
   NOTE: it is done internally by the graphics pipeline; programmer only can specify which part of a canvas is the viewport

WebGL - Transformations

Vertex Shader: where Model, View and Projection matrices are applied to the vertex data

- In JS, it is necessary to create and pass these matrices to the *vertex shader*

```
<!-- The vertex shader operates on individual vertices in our model data by
setting gl_Position -->
<script id="vertex-shader" type="x-shader/x-vertex">
  //Each point has a position
  attribute vec3 position;

  // The transformation matrices
  uniform mat4 model;
  uniform mat4 view;
  uniform mat4 projection;

  void main() {
    gl_Position = projection * view * model * vec4( position, 1.0 );
  }
</script>
```

WebGL - Transformations

View + Model

- OpenGL: matrix **glMatrixMode(GL_MODELVIEW)**
- Camera positioning (***View***)
- Rotations, translations, scales (***Model***)
- In WebGL, it is necessary to create and pass the matrices to the *vertex shader*

Projection

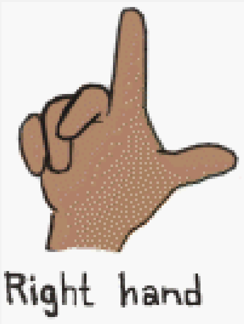
- OpenGL: matrix **glMatrixMode(GL_PROJECTION)**
- Defines the projection volume
- In WebGL, it is necessary to create and pass the matrix to the *vertex shader*

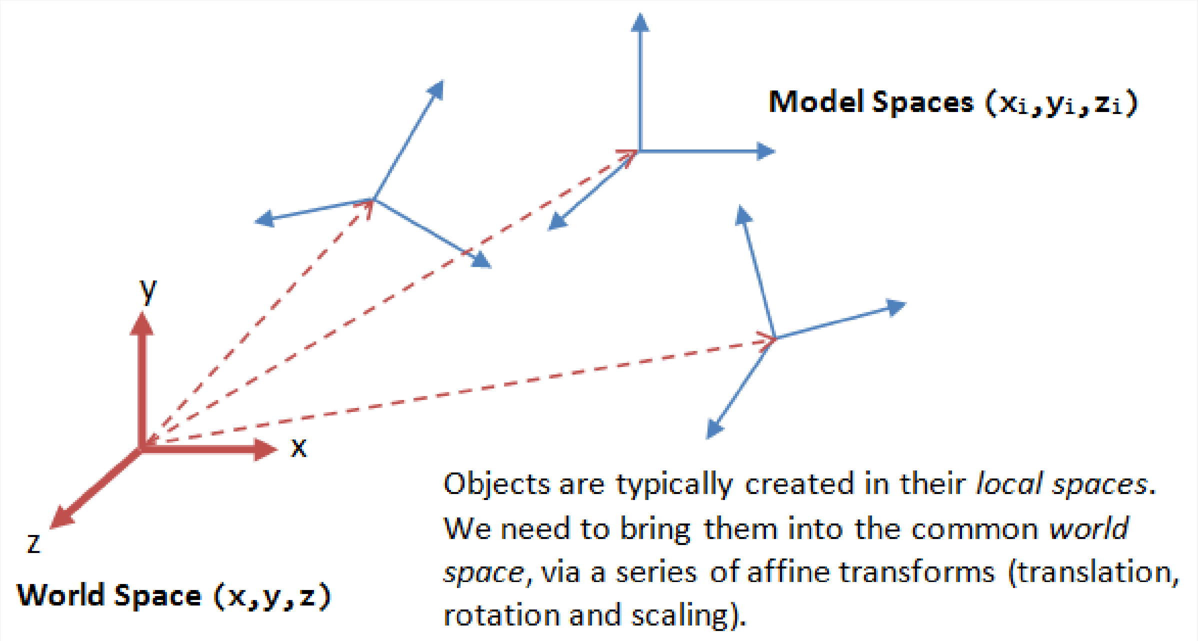
Viewport

- OpenGL: **glViewport (x, y, vpWidth, vpHeight)**
- WebGL (it's identical): **gl.viewport(x, y, vpWidth, vpHeight)**

Model Transformation

- Converts a vertex $v = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$ from one coordinate system to another $\acute{v}$
- Arrange the objects in the world In computer graphics, transform is carried by multiplying the vector with a transformation matrix M:





Model Transformation

- Transforms each vertex v from object to world coordinates $\acute{v} = Mv$

1. Scaling: $S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$

$$\acute{v} = Sv = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} s_x * x \\ s_y * y \\ s_z * z \end{bmatrix}$$

Model Transformation

- Transforms each vertex v from object to world coordinates $\acute{v} = Mv$

2. Rotation: $R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$

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

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

$$\text{e.g. } \acute{v} = R_z v = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \\ z \end{bmatrix}$$

Model Transformation

- Transforms each vertex v from object to world coordinates $\acute{v} = Mv$

3. Translation: $\acute{v} = v + T = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$

cannot be represented as a 3x3 matrix

Solution: use ~~homogeneous coordinates~~, where a vertex $v = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$

$$\acute{v} = Tv = \begin{bmatrix} 0 & 0 & 0 & t_x \\ 0 & 0 & 0 & t_y \\ 0 & 0 & 0 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

WebGL – Homogeneous coordinates

- A 3D point is defined in a typical Cartesian coordinate system: `(x,y,z)`
- The added 4th dimension changes this point into a **homogeneous coordinate**: `(x,y,z,1)`
- It still represents a point in 3D space and allows for lots of nice techniques for manipulating 3D data

```
function cartesianToHomogeneous(point)
  let x = point[0];
  let y = point[1];
  let z = point[2];

  return [x, y, z, 1];
}

function homogeneousToCartesian(point) {
  let x = point[0];
  let y = point[1];
  let z = point[2];
  let w = point[3];

  return [x/w, y/w, z/w];
}
```

Functions to convert coordinates between the cartesian and homogeneous systems

Model Transformation

Summary of Homogeneous Matrix Transforms

Scale: $S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Rotation: $R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ $R_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

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

Translation:$T = \begin{bmatrix} 0 & 0 & 0 & t_x \\ 0 & 0 & 0 & t_y \\ 0 & 0 & 0 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Model Transformation

Summary of Homogeneous Matrix Transforms

EXAMPLE:

Successive transforms: $\acute{v} = (T.S.R_z.R_x).v$

Inverse successive transforms:

$$v = (T.S.R_z.R_x)^{-1}.\acute{v} = R_x{}^{-1}.R_z{}^{-1}.S^{-1}.T^{-1}.\acute{v}$$

Model Transformation

Initialization or reset (4x4 identity matrix)

- OpenGL: function glLoadIdentity()
- **WebGL**: you need to know matrix math!

```
let identityMatrix = [
  1, 0, 0, 0,
  0, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 0, 1
];
```

NOTE: WebGL is just a rasterization engine, similar to Canvas 2D; it is not a 3D library (like Three.js) where you supply 3D data and the libraries take care of calculating clip space points from 3D!
Fortunatelly there are plenty of matrix libraries available for WebGL, like glMatrix.js

	glMatrix.js
M = I	let M = mat4.create();

Read more:
https://webgl2fundamentals.org/webgl/lessons/webgl-2d-vs-3d-library.html
https://math.hws.edu/graphicsbook/c7/s1.html#webgl3d.1.2

Model Transformation

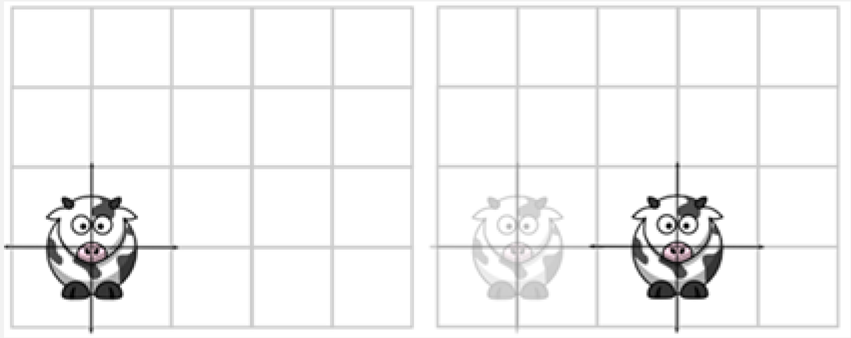
Translation

- OpenGL: function glTranslate{f|d}(dx, dy, dz)
- **WebGL** (using library **glMatrix.js**):

M = T	mat4.fromTranslation(M, [tx, ty, tz]) *
M2 = M1*T	mat4.translate(M2, M1, [tx, ty, tz])

* This is equivalent to (but much faster than):

```
mat4.identity(dest);
mat4.translate(dest, dest, vec);
```



Model Transformation

Rotation

- OpenGL: function glRotate{f|d}(angle, ex, ey, ez)
- **WebGL** (using libray **glMatrix.js**):

M = R	mat4.fromRotation(M, angle*, axis**)
M2 = M1*R	mat4.rotate(M2, M1, angle, axis*)

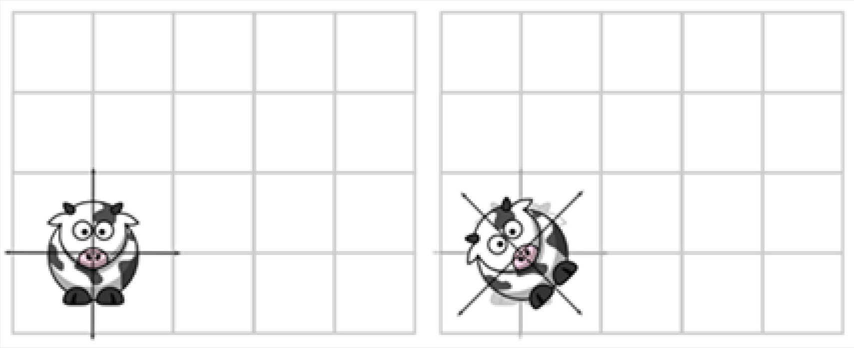
* angle in radians

** axis is 3x1 vector indicating the rotation axis

X-rotation: [1,0,0]

Y-rotation: [0,1,0]

Z-rotation: [0,0,1]



Model Transformation

Scaling

- OpenGL: function glScale{f|d}(sx, sy, sz)
- **WebGL** (using libray **glMatrix.js**):

M = S	mat4.fromScaling(M, [sx, sy, sz])
M2 = M1*S	mat4.scale(M2, M1, [sx, sy, sz])

Model Transformation

Remember: it is on the ***Vertex Shader*** where **Model** (and **View** and **Projection**) matrix is applied to the vertex data

```
<!-- The vertex shader operates on individual vertices in our model data by
setting gl_Position -->
<script id="vertex-shader" type="x-shader/x-vertex">
  //Each point has a position and color
  attribute vec3 position;

  // The transformation matrices
  uniform mat4 model;
  uniform mat4 view;
  uniform mat4 projection;

  void main() {
    gl_Position = projection * view * model * vec4( position, 1.0 );
  }
</script>
```

In JavaScript:

- Get address for the vertex shader *uniform* variable

```
let modelMatrix = gl.getUniformLocation(program, "model")
```

- In render routine, after applying transformations to a 4x4 `matrix` variable and before drawing a new object

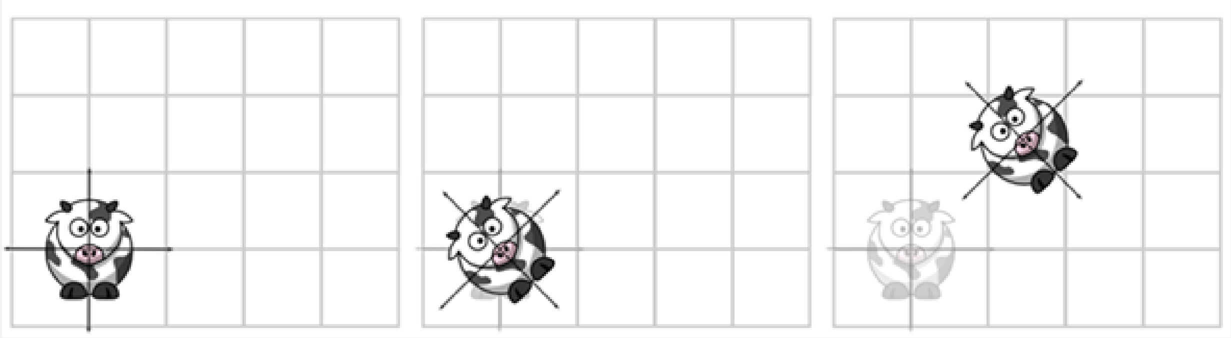
```
gl.uniformMatrix4fv(modelMatrix, gl.FALSE, matrix)
```

WebGL - Transformations

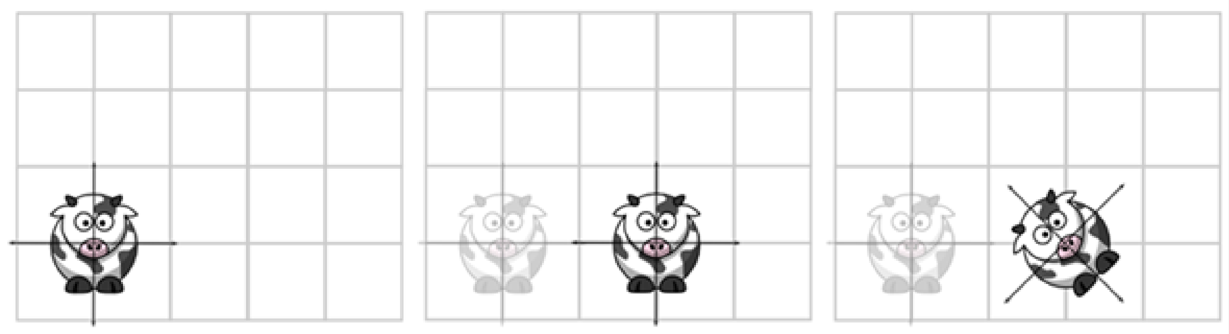
In WebGL, transformations are accumulated using matrix **multiplications**

The order is important!

Rotation followed by translation

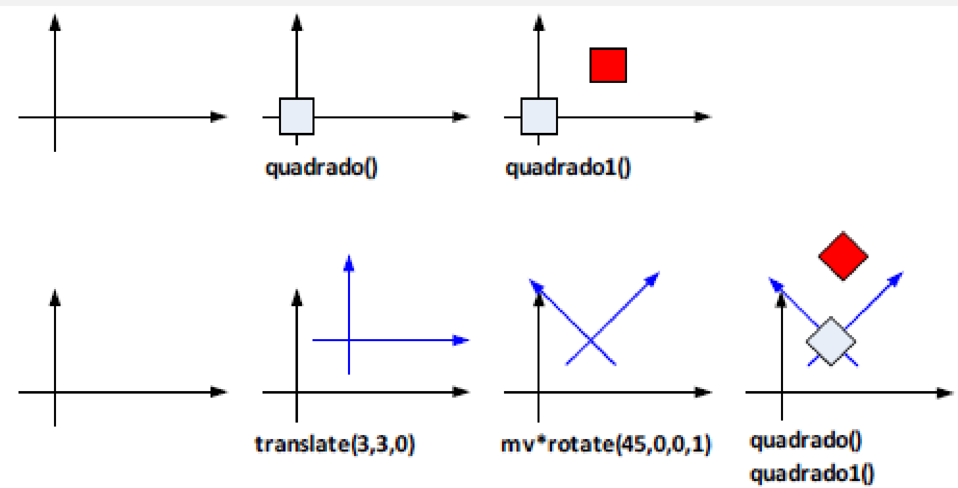


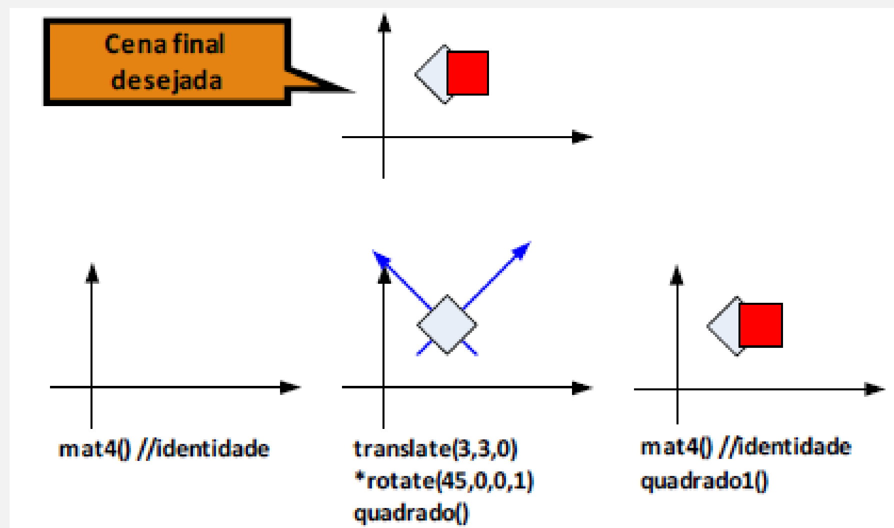
Translation followed by rotation



WebGL - Transformations

Local (model) versus **global (world)** coordinate system transformations





WebGL - Transformations

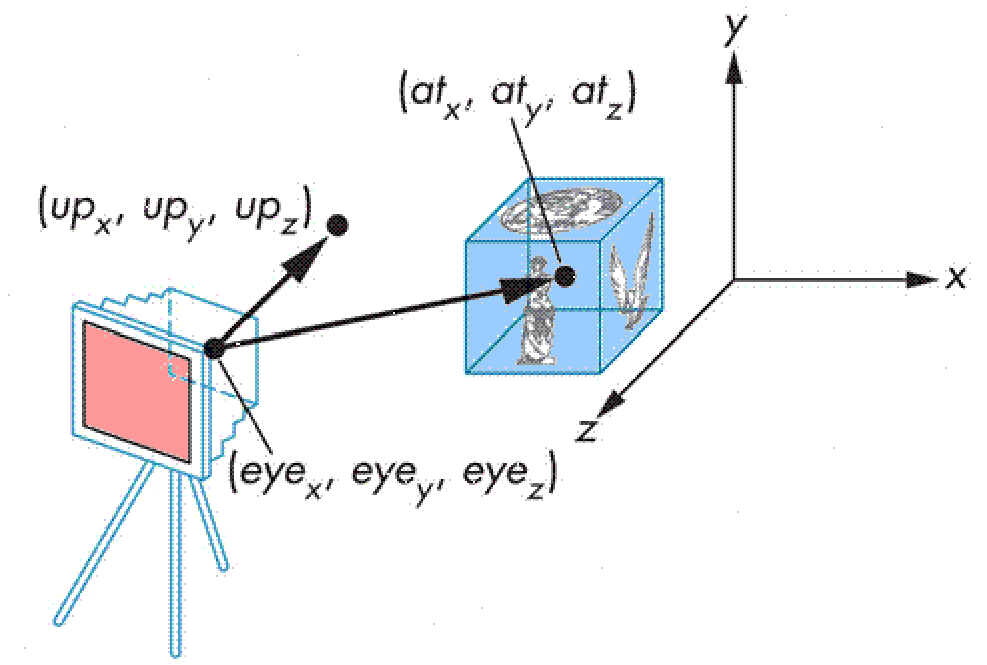
Local versus **global** coordinate system

- Whichever method you use, you will almost always need to either reset the matrix to the identity matrix, or save and restore a previous matrix state
- In OpenGL a transformation matrix can be **stored** using function glPushMatrix() and it can be recovered using glPopMatrix()
- In WebGL, this can be simulated by creating a heap system:
  - Create an array to store your transformation matrices and use pop() and push() array methods
  - Be carefull in pairing the *pushes* and *pops*
  - Make sure that pushing

View Transformation

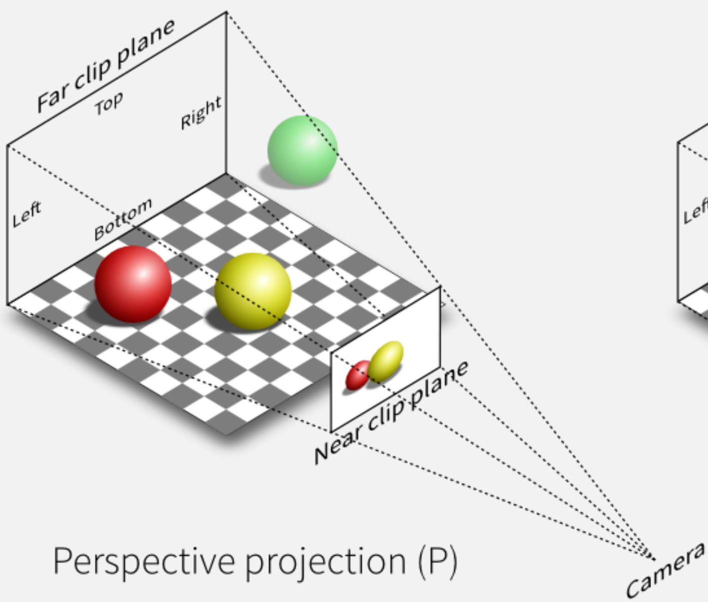
- OpenGL: function gluLookAt(eye, at, up)
- **WebGL** (using libray **glMatrix.js**):

```
mat4.lookAt(M,  [eyeX, eyeY, eyeZ], [atX, atY, atZ], [upX, upY, upZ])
```

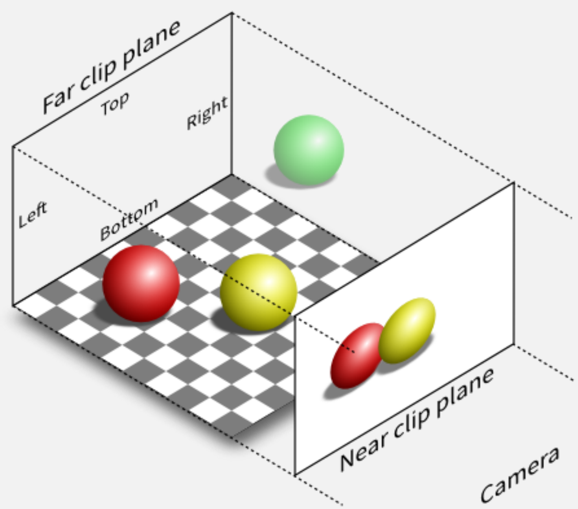


Projection transformation

- Projection transformation defines the ***visualization volume***, which is used in two ways:
  - Determines how an object is projected into the screen (using **perspective** or **orthographic** projection)
  - Defines which objects (or parts of it) are eliminated from the final image



Perspective projection (P)



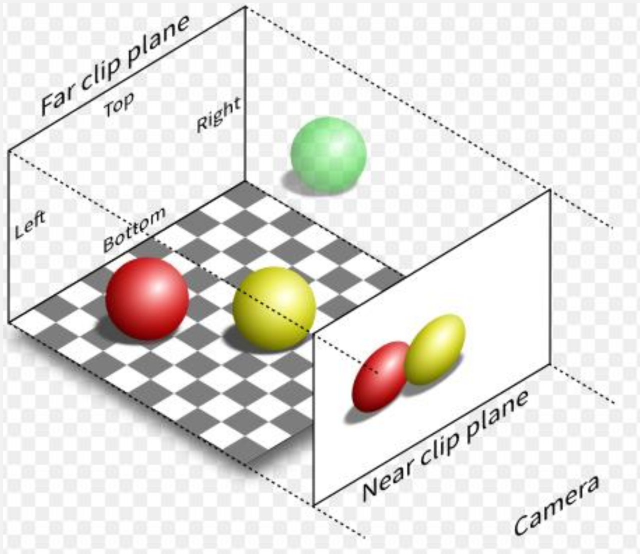
Orthographic projection (O)

Projection transformation

Orthographic Projection

- OpenGL: function glOrtho(left, right, bottom, top, near, far)
- **WebGL** (using libray **glMatrix.js**):

```
mat4.ortho(M,  left, right, bottom, top, near, far)
```



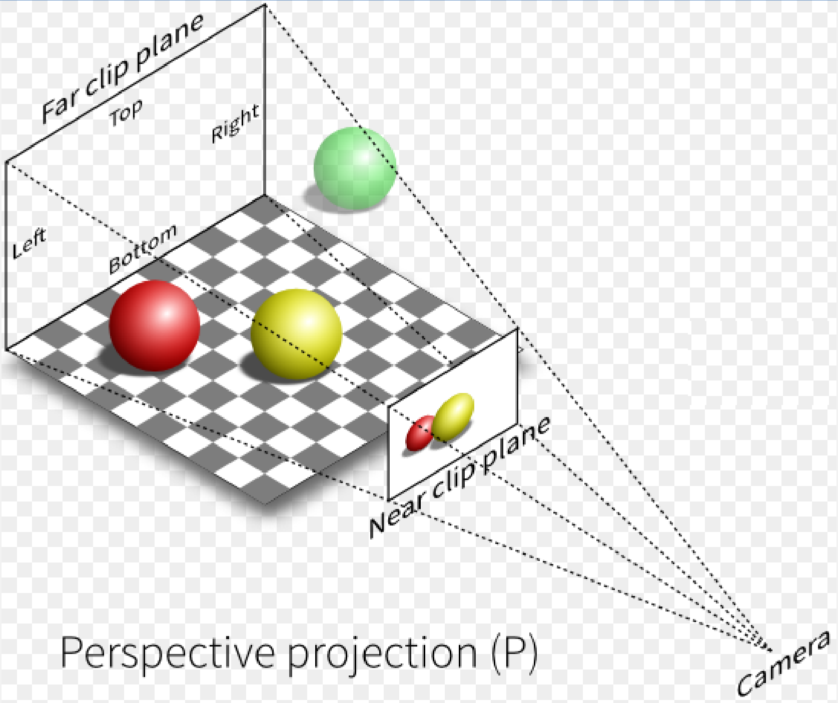
Orthographic projection (O)

Projection transformation

Perspective Projection

- OpenGL: function glPerspective(fovy, aspect, zNear, zFar)
- **WebGL** (using libray **glMatrix.js**):

```
mat4.perspective(M,  fovy, aspect, zNear, zFar)
```



WebGL example – rotating triangle

- **Vertex shader**:
  - In the provided example, all vertices are already provided in NDC, therefore no projection transformation needs to be performed
  - The rotation will be passed onto the **model/view matrix**:

```
<script id="vertex-shader" type="x-shader/x-vertex">
    attribute vec3 vPosition;
    uniform mat4 projection, modelview;

    void main()
    {
        // gl_Position = projection * modelview * vec4(vPosition, 1);
        gl_Position = modelview * vec4(vPosition, 1);
    }
</script>
```

WebGL example – rotating triangle

- **JS**:
  - The main program should pass the rotation matrix onto the vertex shader uniform variable **modelview**
  - First, get a pointer to the shader variable:

```
//Get address for the vertex shader uniform modelview variable
mvLoc = gl.getUniformLocation(program, "modelview");
```

  - Use a rendering loop to update the rotation matrix and pass it to the vertex shader uniform variable **modelview**:

```
mv = new mat4();
mv = mult(mv, rotate(rotation++, [0, 1, 1])); // Y & Z axis rotation
// update shader variable with the new rotation value
gl.uniformMatrix4fv(mvLoc, gl.FALSE, flatten(mv));
```