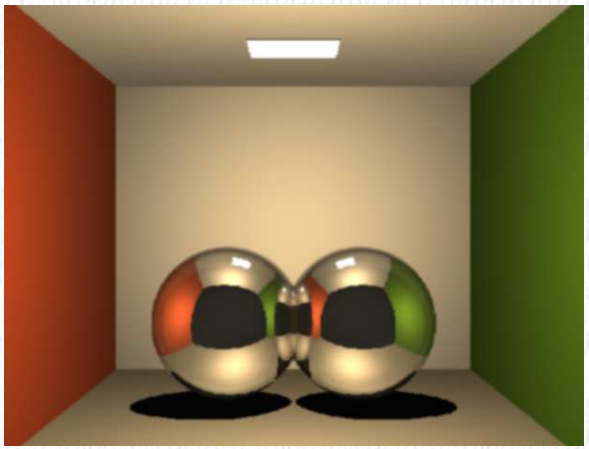
Lab 2

Kostas Vardis

http://graphics.cs.aueb.gr/graphics/people.html

Athens University of Economics and Business

Computer Graphics BSc



Introduction

- Transformations
- GLSL
- GLM

GLM

- OpenGL Mathematics Library
- Same syntax with GLSL
- Strict type checking
- Use glm namespace
- glm::translate, glm::rotate, glm::ortho, etc.

GLSL

- OpenGL Shading Language
- C like syntax
- Write vertex, geometry, fragment shaders
- Strict input-output
- Pass variables from CPU side using uniforms
- Swizzling
- Allow for very cool tricks ☺

GLSL Basic Types

Scalar values

- bool, int, uint, float, double

Vector values

- vec2, vec3, vec4, ivec2, ivec3, ivec4, uivec2, etc.

Matrices

- mat2, mat3, mat4

Create Uniforms

```
// Get an index to the variable in the shader
GLint uniform_mvp =
glGetUniformLocation(m_program,
"uniform_mvp");
```

Render with shaders

```
// Set it active
glUseProgram(m_program);
// pass a matrix uniform
glUniformMatrix4fv(uniform_mvp, 1, false,
&mvp_matrix[0][0]);
// draw!!
e.g. glDrawArrays…..
```

GLSL Vertex Shader

```
#version 330
layout(location = 0) in vec4 position;
uniform mat4 uniform_mvp;
void main() {
   gl_Position = uniform_mvp * position;
// or gl_Position = uniform_mvp * position.xyzw;
}
```

GLSL Fragment Shader

```
#version 330
layout(location = 0) out vec4 out_color;
void main() {
// RGBA color (0-1 values)
 out_color = vec4(1.0f, 1.0f, 1.0f, 1.0f);
}
```

Transformations

- OpenGL uses a right-handed coordinate system

(Right is +X, Up is +Y, Back is +Z)

- Matrix multiplication depends on the vector type

(i.e. for column vectors multiplication is right to left,, for rows is left to right)

Example: for column vectors

$$A = P * V * M$$

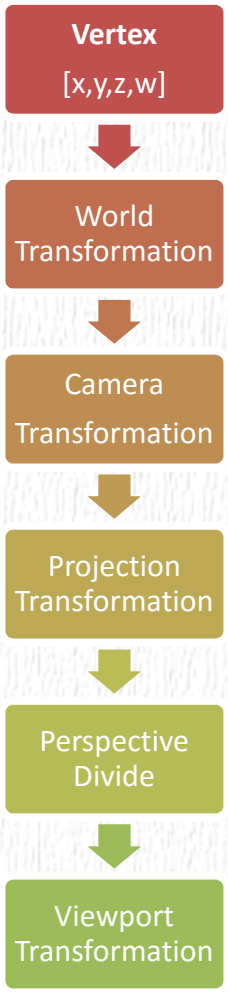
$$v' = A * v$$

Use the inverse to go back

$$A^{-1} = M^{-1} * V^{-1} * P^{-1}$$

$$v = A^{-1} * v'$$

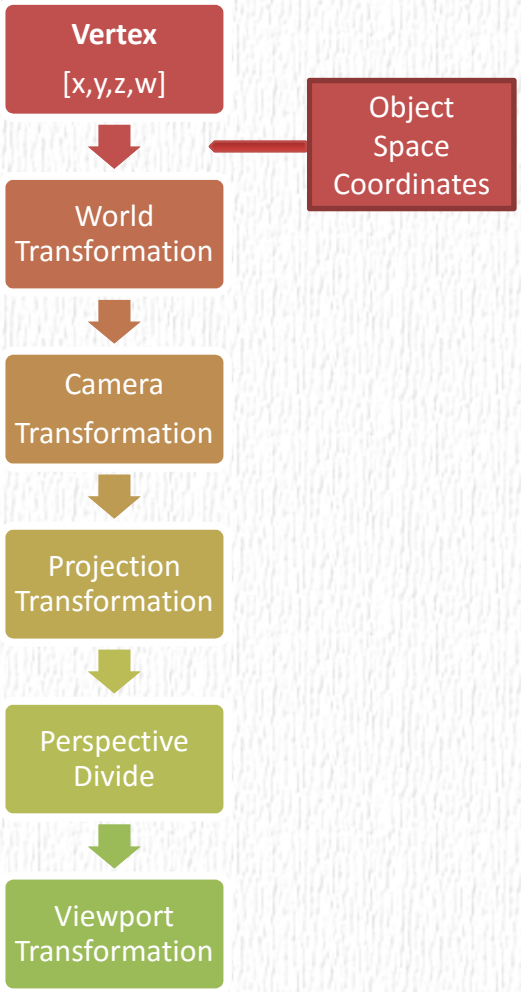
Transformations



General idea

- Set the dimensions of your window (viewport)
- Define your world
- Place objects in the world
- Create a “camera” and transform the world so that the camera is at the origin
- Project the world on the screen

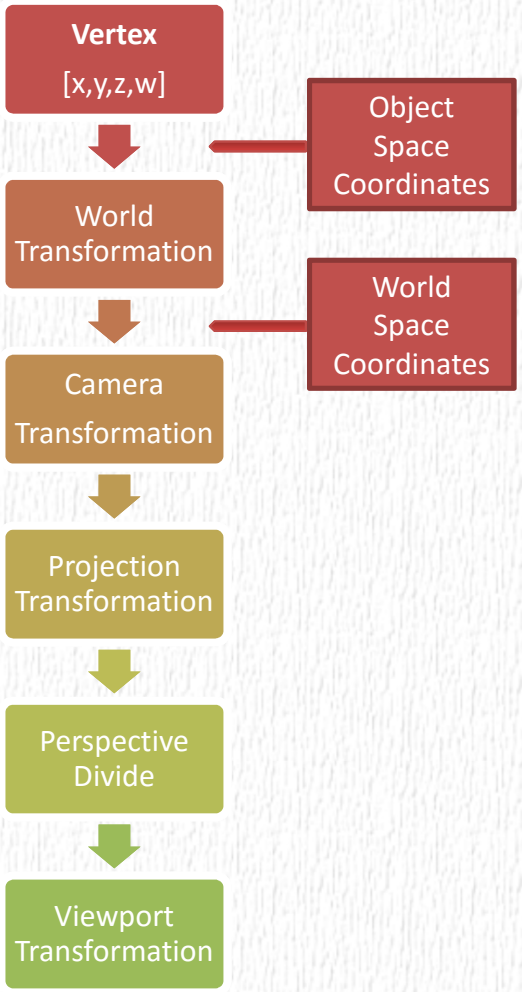
Transformations



Object Space Coordinates (OSC)

- What we get from a 3D modelling software (3DS Max, Maya, etc.)
- Usually object is centered at its own origin in OSC
- The coordinates we pass to vertex buffers

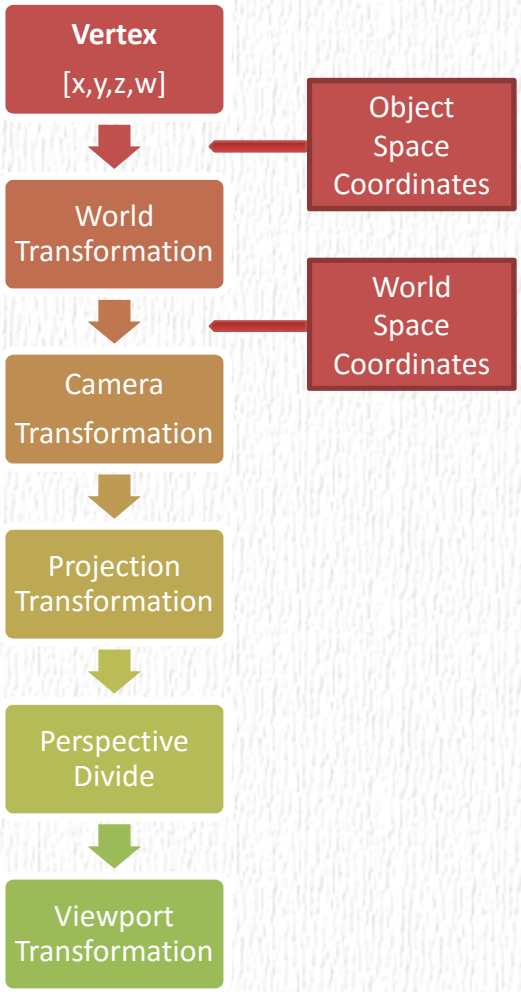
Transformations



World Space Coordinates (WSC)

- Place our object in the scene
- Use matrix transformations to move from OSC->WSC
- Translate
- Rotate
- Scale

Transformations



World Coordinate System

- glm::translate(x,y,z)

```
// Move an object in the x-axis by 8 units
glm::mat4 translation_mat = glm::translate(8.0f, 0.0f, 0.0f);
```

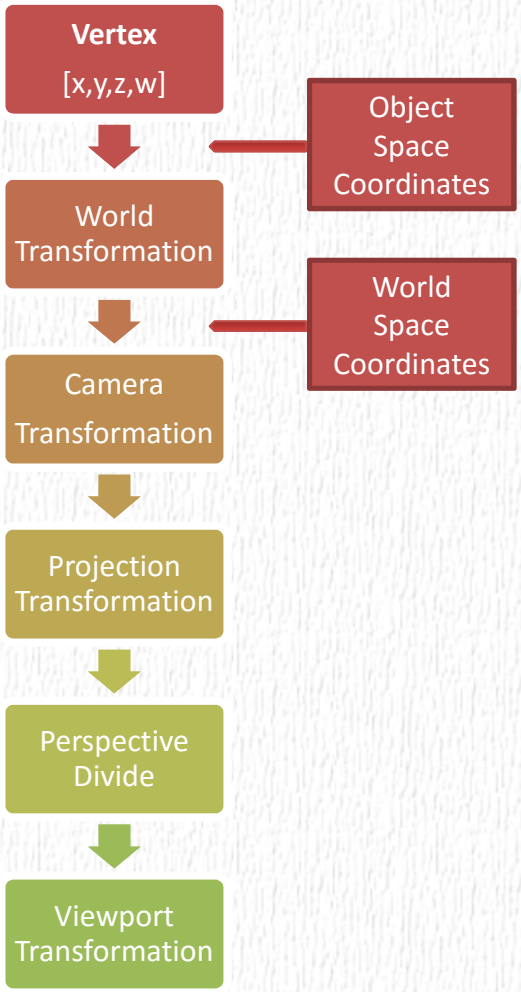
- glm::rotate (angle,x,y,z)

```
// Rotate object around the y-axis by 45 degrees counter-clockwise
glm::mat4 rotate_y_axis_mat = glm::rotate(45, 0.0f, 1.0f, 0.0f);
```

- glm::scale(x,y,z)

```
// Scale object uniformly by a factor of 2
glm::mat4 scale_mat = glm::scale(2.0f, 2.0f, 2.0f);
```

Transformations



Order Matters!

- Matrix multiplication is not commutative (AB ≠ BA)
- Common usage is first scale, then rotate and finally translate (T*R*S)

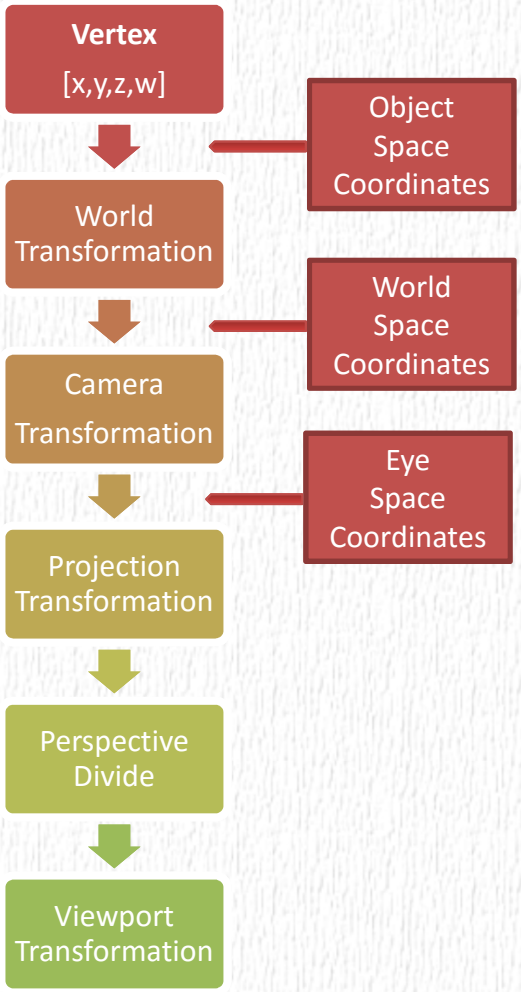
Example 1:

```
// Rotate the object by 45 degrees and THEN translate it
glm::ocs_to_wcs_mat =
translation_mat * rotate_y_axis_mat;
```

Example 2:

```
// Translate the object by 8 units and then rotate the translated object by 45 degrees
glm::ocs_to_wcs_mat =
rotate_y_axis_mat* translation_mat;
```

Transformations



Eye Coordinate System (ECS)

- Need to view the world from our point of view
- Need a camera center, a direction and a new coordinate system
- Use glm::lookat

Transformations

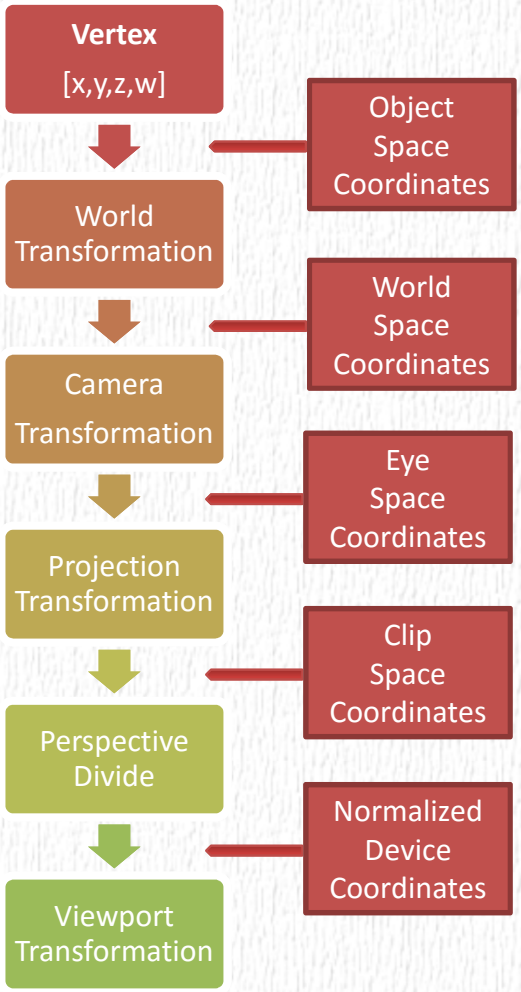
glm::lookAt(vec3 *eye, vec3 center, vec3 up)*

- Need only to specify a direction and an up vector (for now 0,1,0)
- dir = center – eye
- Coordinate System uvn (similar to xyz)
- n = dir
- v = n x up
- u = v x n

Example:

```
// Create a camera at the origin (0,0,0), looking at –Z
glm::wcs_to_ecs_mat =
glm::lookat(0.0f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f, 0.0f);
```

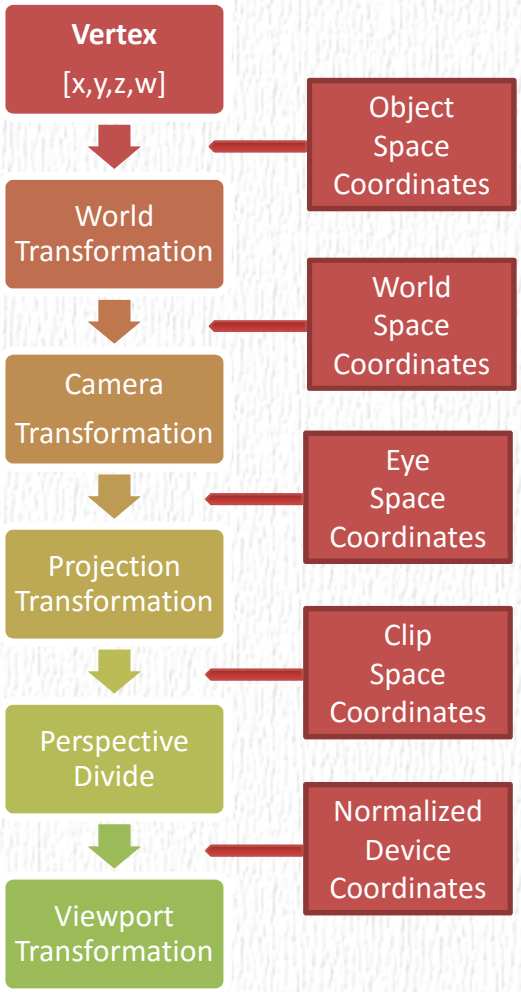
Transformations



Projective Space (CSS)

- Orthographic, Perspective
- Transform scene from ECS extents to a cuboid (CSS)
- NDC after perspective divide

Transformations



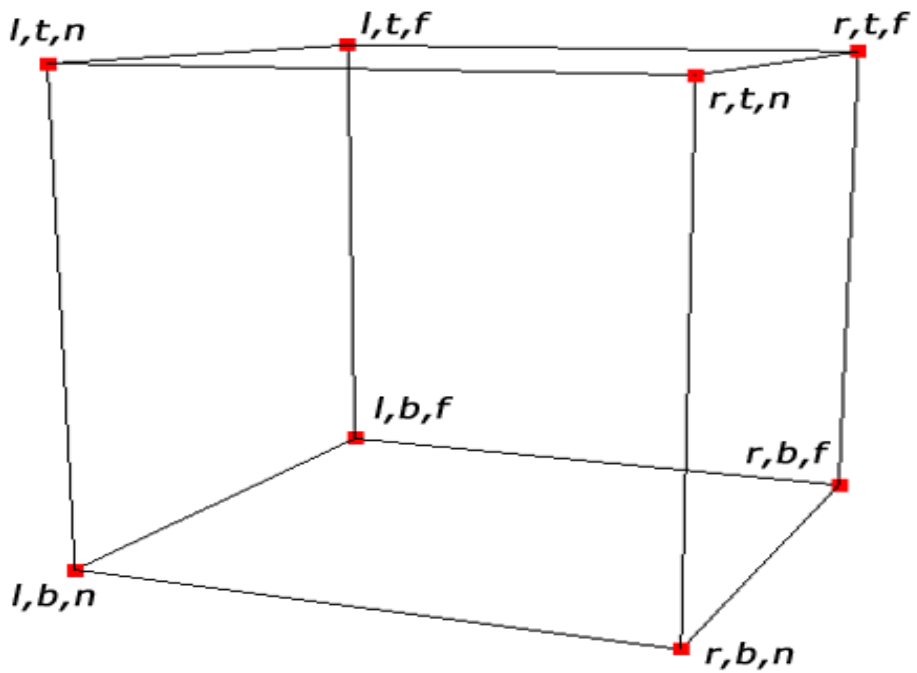
Orthographic Projection

- Simplest form of projection
- Keeps line parallellism
- Linear relationship between coordinates in eye space and NDC

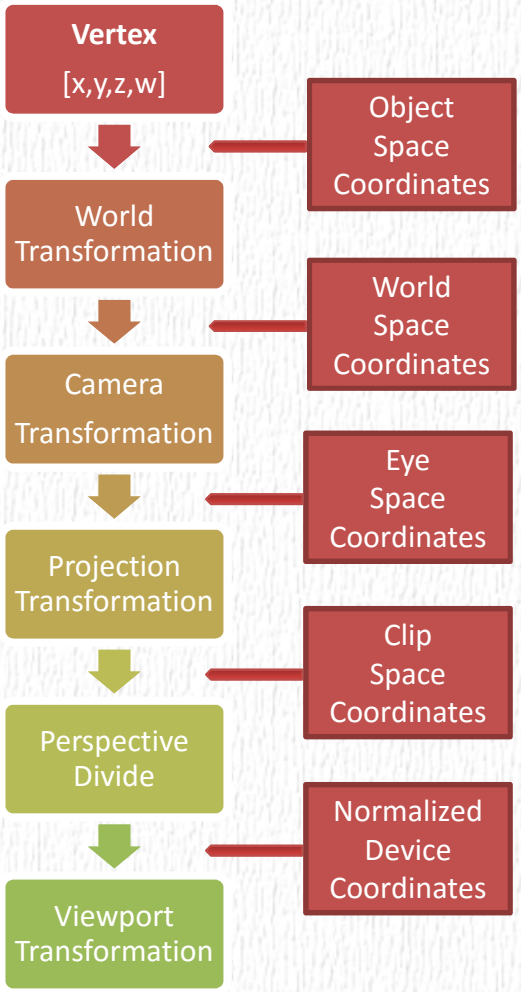
Transformations

glm::ortho(*left*, *right*, b*ottom*, *top*, *near*, *far*)

- Parameters are coordinates to clipping planes
- Useful for 2D rendering (text, etc)



Transformations



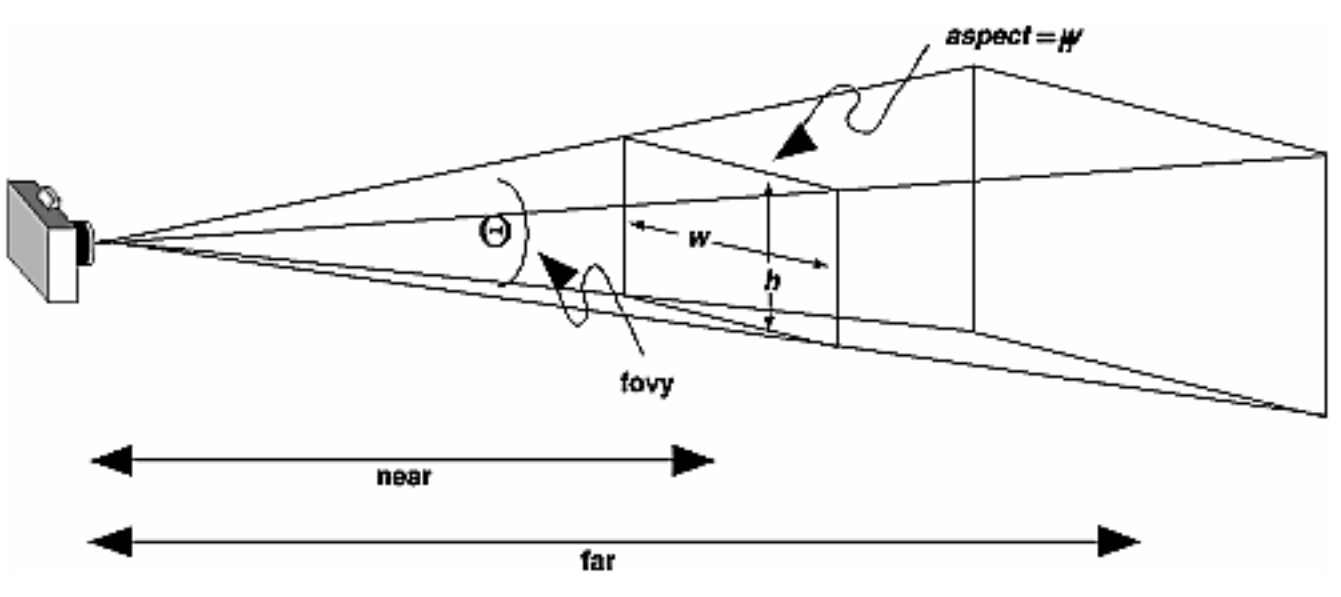
Perspective Projection

- Mimics way we perceive objects
- Parallel lines may converge at infinity
- Objects appear smaller as they move further away
- Non-linear z relationship between coordinates in eye space and NDC

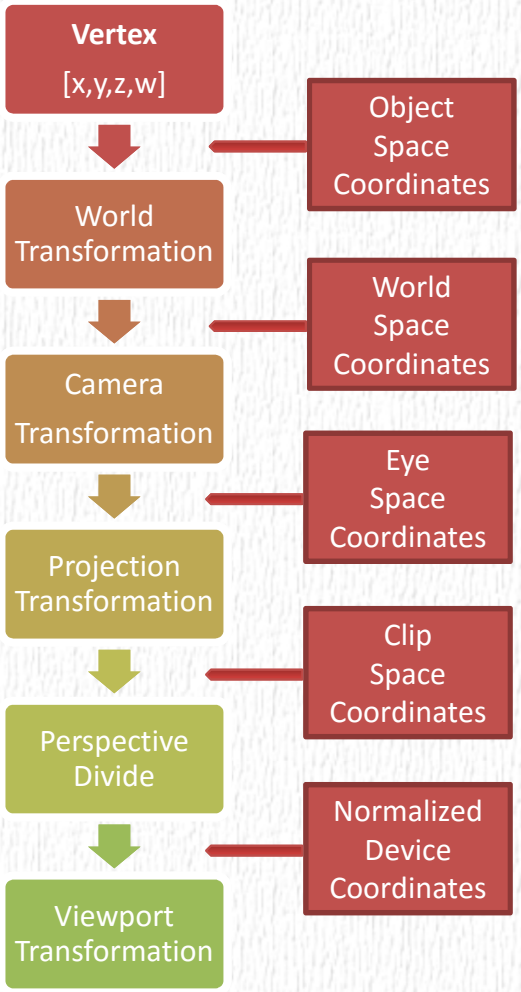
Transformations

glm:perspective (*fov, aspect_ratio, near, far*)

glm::mat4 persp_proj_mat = glm::perspective(30.0f, (float)w/(float)h, near_field_value, far_field_value);



Transformations



Window Space

- Transforms NDC to Window cords
- Happens internally

glViewport(*x,y,width,height*)

Recap

- Transform object space to world space
- Transform world space to eye space
- Transform eye space to clip space
- Combine them

```
glm::mat4 mvp_matrix = persp_proj_mat *
wcs_to_ecs_mat * ocs_to_wcs_mat;
```

- Pass final matrix to shader

Done!

- Check lab2 project

- Check pdf for more information and online content