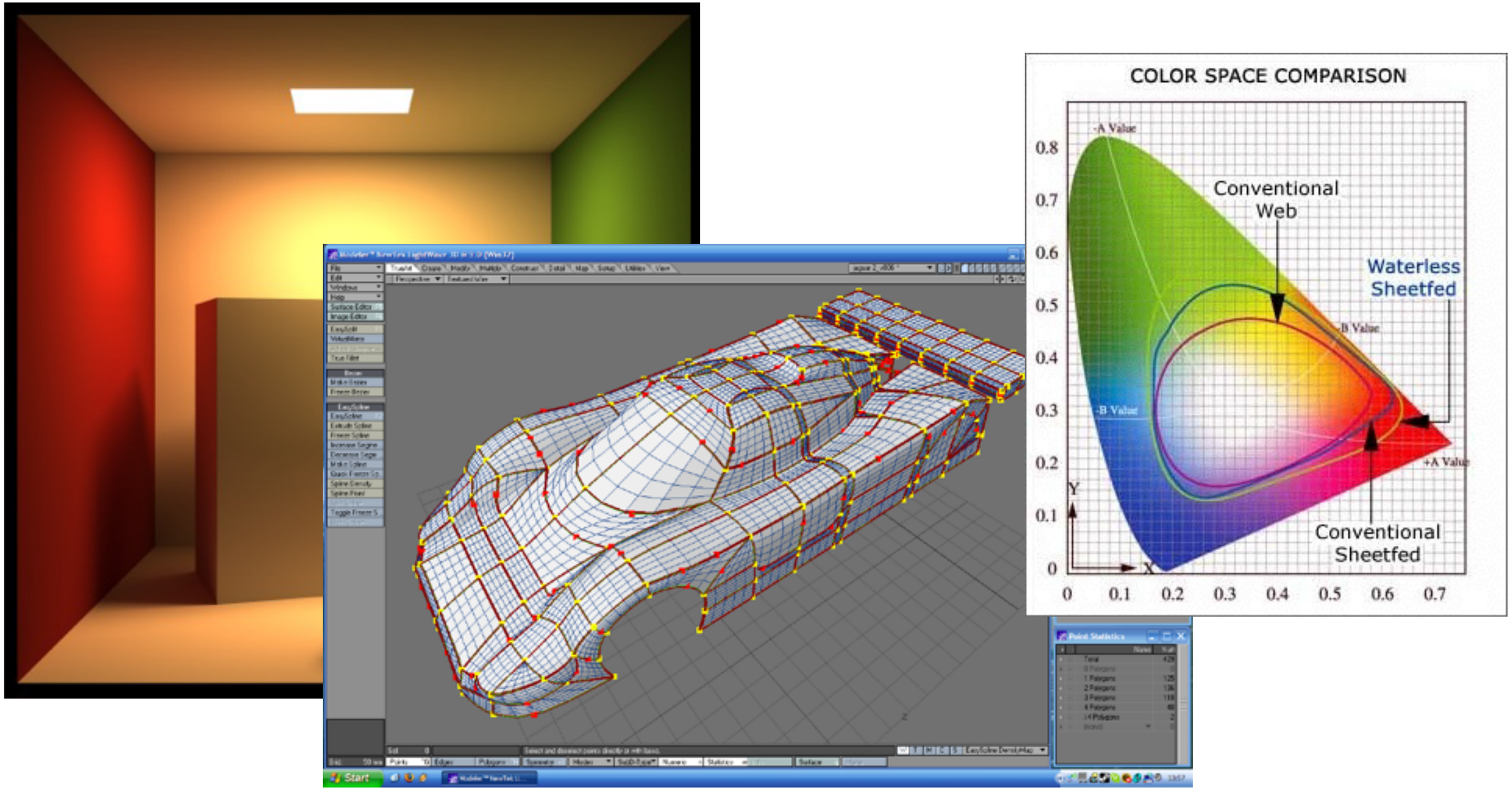


Lecture 16 : Algebra of viewing transformations, projection, homogeneous representations. Viewing pipeline

Tuesday November 2nd, 2021

Logistics

- We will have your midterm graded by sometime next week (at the latest; we aim for Nov 10th or earlier)
- Programming assignment #4 has been released
  - Due next Wednesday, Nov 10th
  - Includes some readings on curves, transforms.
  - (more readings forthcoming on viewing, with h/w #5)
- Be mindful of the requirements for Assignment #4

Logistics

- **Requirement #1**. Your drawing has to include, at minimum, one of the following:
  (a) A piecewise-defined curve, that shifts from one formula to another (see the example in this [JSBin demo]), that is also closed in the sense that the curve returns to the place where it started, forming a closed "loop"; note that this is different from the example we saw in class, where all curves, including those that had piecewise-definitions, were "open".
  or (you can do both if you wish!)
  (b) Multiple separate curves, out of which one has to have a piecewise-defined formula (i.e. at least two components with different formulas, joined together), but in this case the curves don't have to be closed (unless you want them to be!).
- **Requirement #2.** Your drawing needs to have at least one object that is animated using one of the curves defined. It is sufficient to have the parametric curve control only the *position* of the moving object (as in this [JSBin demo]), but if you want to be extra fancy you can also control the *orientation* of the moving object (see this [JSBin demo]).
- **Requirement #3.** You must "draw" the path of at least one of the curves involved in your scene (consider the way we discussed in our demos, by splitting the curve up into small line segments). It's ok to have a "switch" of sorts (e.g. the value of a slider) that "turns off" drawing the curve line, if you feel your scene looks better without it! As long as there is the option to display that path, you are good!
- **Requirement #4.** At the point where the different formulas of a piecewise-defined curve in your scene, you must enforce at least C0-continuity (G1-continuity or higher would be nice; C2 might be overkill, but you are welcome to do it). This is just for *one of the junctions* in one of your piecewise-defined curves; you are free to use any degree of continuity for all other cases.
- **Requirement #5.** At least one of the curves you use in your implementation has to be a parametric cubic (Hermite, Bezier, B-spline, etc). It doesn't have to be the case that all components of a piecewise curve will have to be a cubic ... just that *somewhere* in your scene there must be a curve that has *at least one component* (if it's a piecewise/component curve) that's a cubic. Frankly, you might find that one of the most natural ways to enforce C1 continuity (if you choose to have at least that degree of smoothness) is to resort to cubics anyhow ...

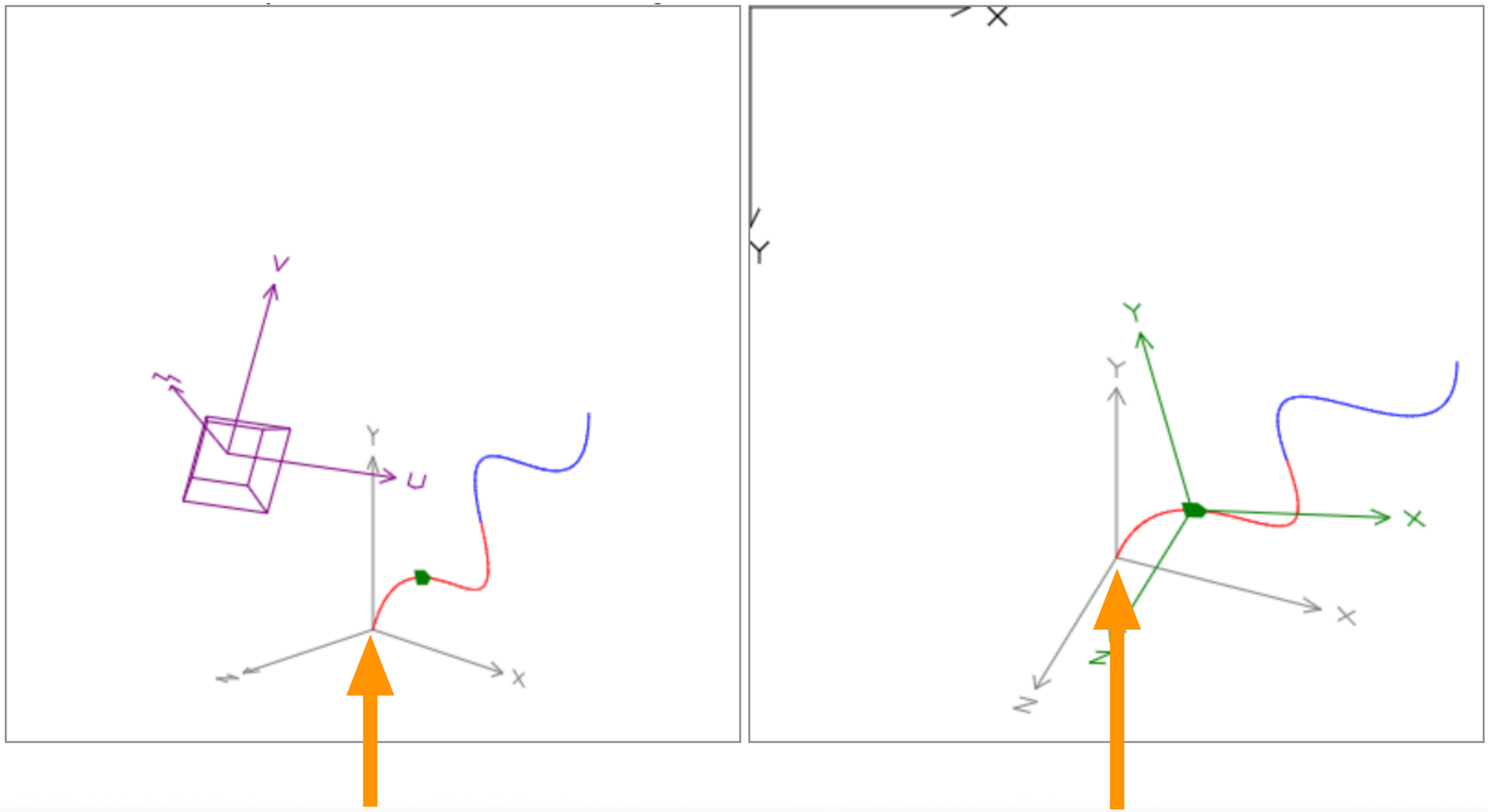
Today's lecture

- More details about drawing/viewing in 3D
  - Recap model/camera transforms, focus on projection transforms, NDC coordinates, and viewport
  - Orthographic and perspective projection in more details
  - Some new details about homogeneous coordinates, including crucial details on projective math
  - Textbook references:
    (Fundamentals of Computer Graphics FCG Chapter 7, and Big Fun Graphics Book Chapter 8 and Chapter 9)

“World” coordinates

jsbin.com/ficoxeh

Week7/Demo4

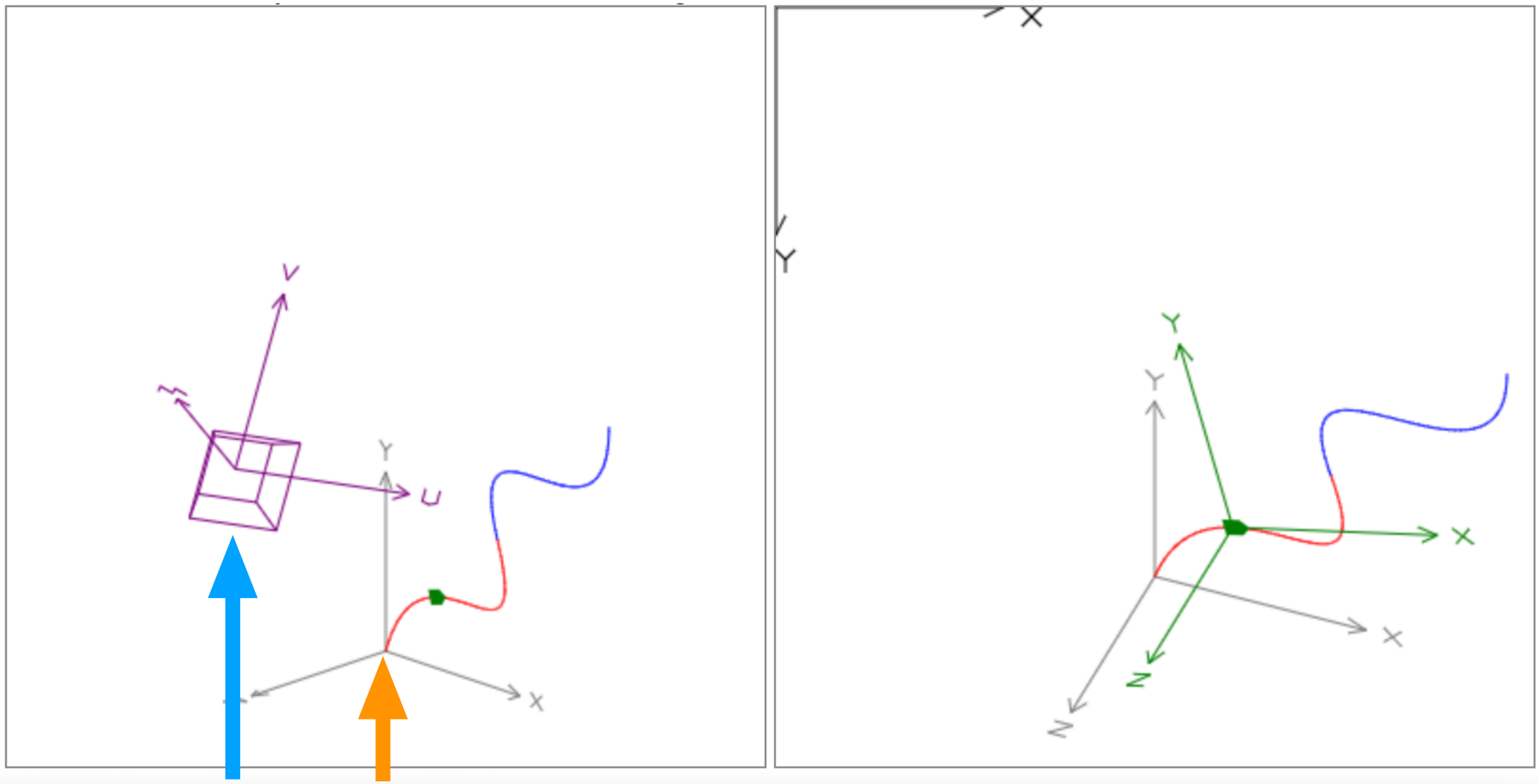


Illustrated in this demo in grey color, the *world coordinate system* is a system that we arbitrarily choose, in such a way that describing locations/vectors/coordinates of things we want to draw becomes convenient.
(For example, the locations of control points for the 2 piecewise-cubics are given in “world coordinates”).

The lookAt transform

jsbin.com/ficoxeh

Week7/Demo4



A linear transform exists (it just takes some mixture of rotation and translation …) that converts *world coordinates* into *camera coordinates*.

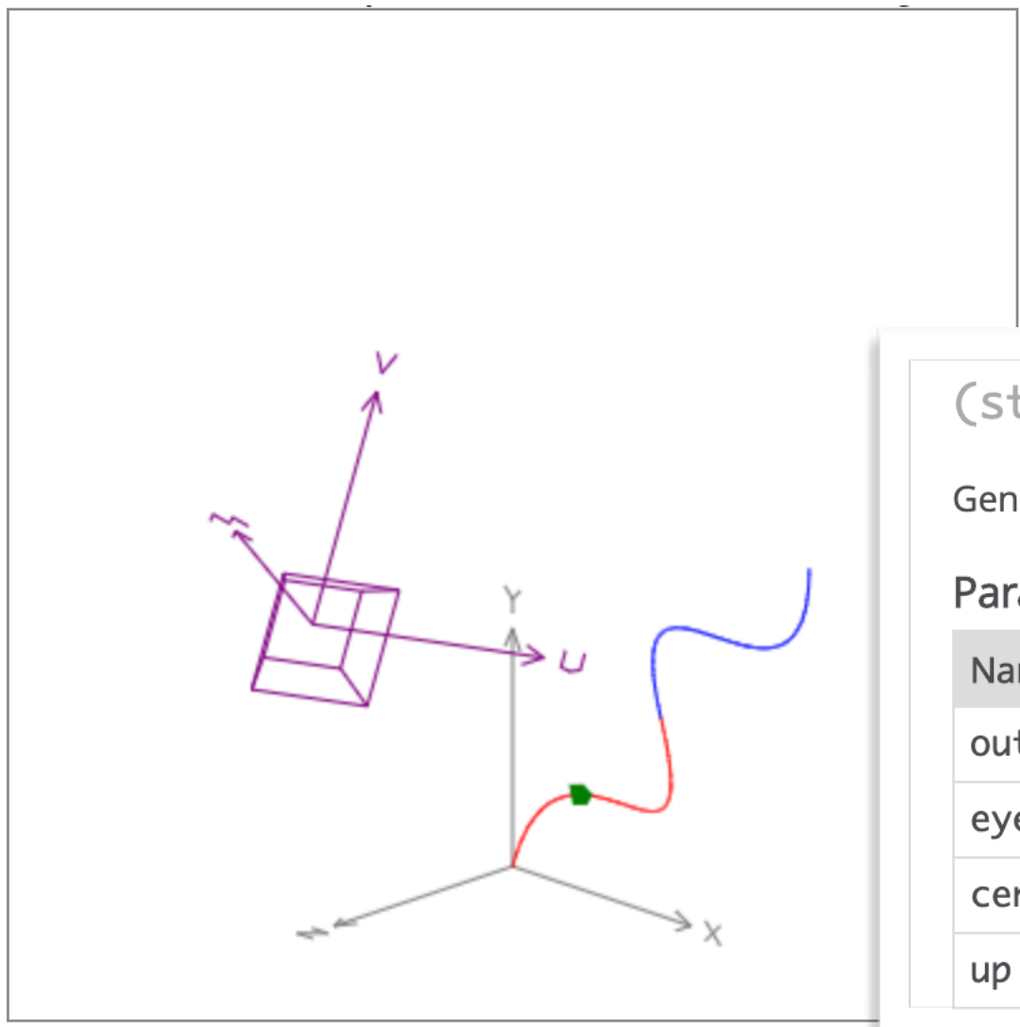
This is called the ***lookAt transform***.

(Your textbook details how exactly we might compute the numerical values that go into this transform matrix … here, we focus on what are the necessary ingredients we need to give a library, e.g. glMatrix, to compute it for us!)

The lookAt transform

jsbin.com/ficoxeh

Week7/Demo4



(static) lookAt(out, eye, center, up) → {mat4}

Generates a look-at matrix with the given eye position, focal point, and up axis.

Parameters:

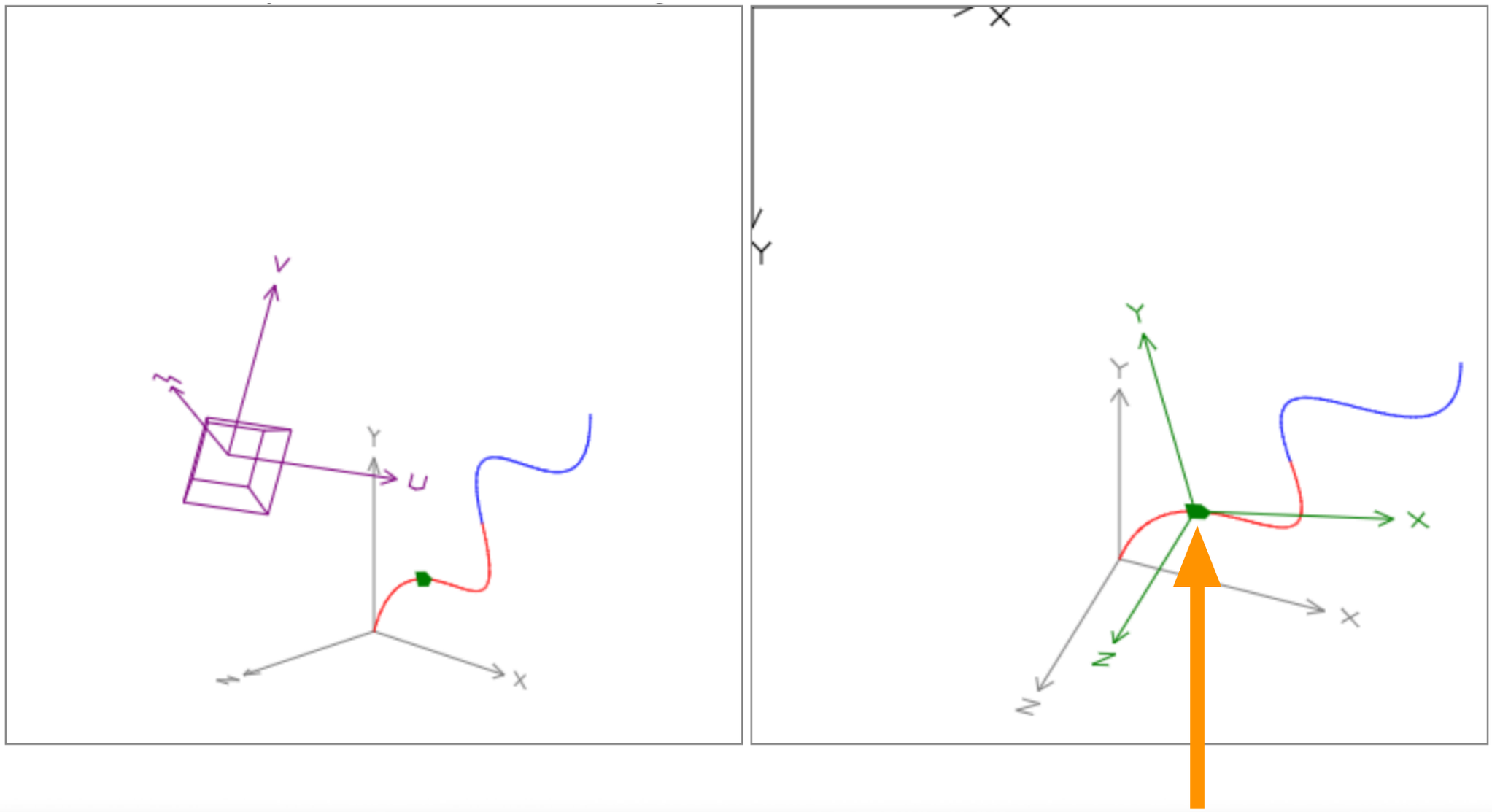
Name	Type	Description
out	mat4	mat4 frustum matrix will be written into
eye	ReadonlyVec3	Position of the viewer
center	ReadonlyVec3	Point the viewer is looking at
up	ReadonlyVec3	vec3 pointing up

Three ingredients for defining/computing the lookAt transform:
(a) the *eye* location, (b) the *target* location, (c) the *up* vector

Model(ing) transform(s)

jsbin.com/ficoxeh

Week7/Demo4



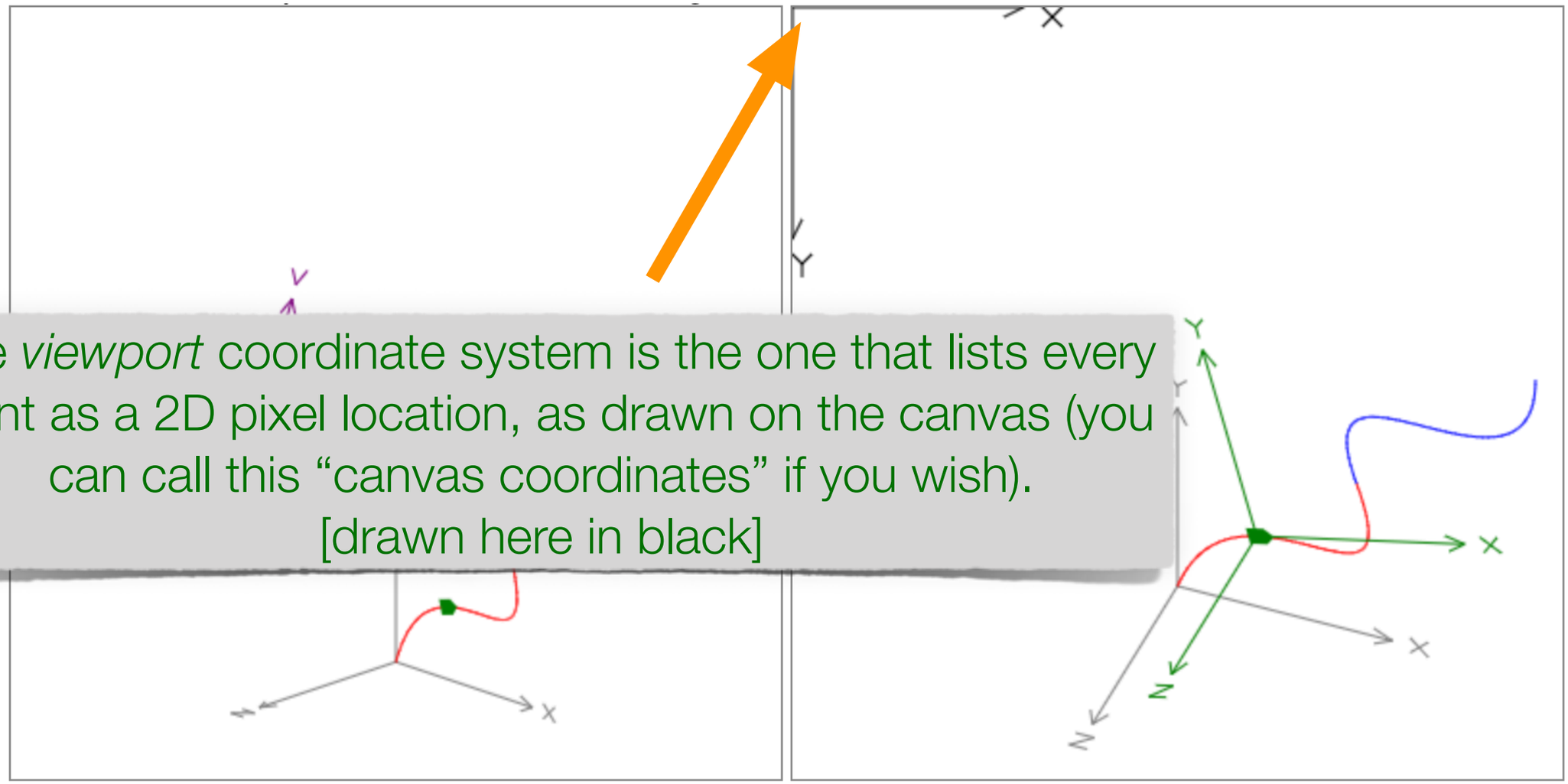
We can also have additional transforms that define coordinate systems (typically for moving objects, or displaced object instances) relative to the world coordinates. (Shown here in green …)

For hierarchically modeled objects, you can have several nested modeling transforms

Canvas/Viewport coordinates

jsbin.com/ficoxeh

Week7/Demo4



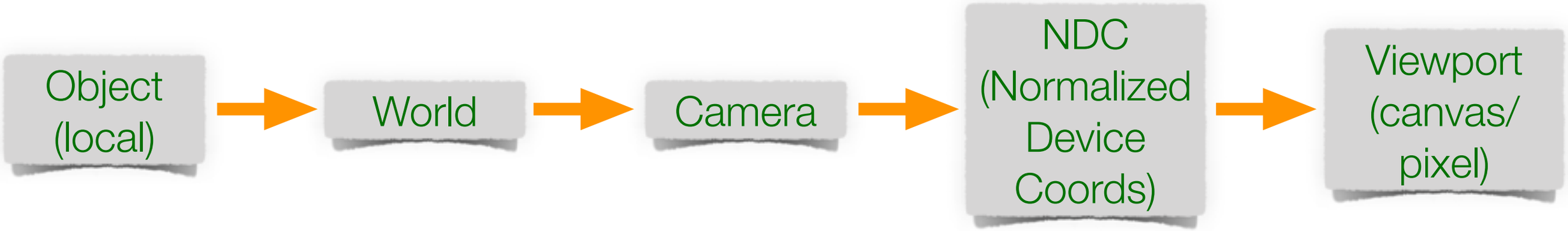
The *viewport* coordinate system is the one that lists every point as a 2D pixel location, as drawn on the canvas (you can call this "canvas coordinates" if you wish).
[drawn here in black]

You can consider this as a 3D coordinate system, from which we ignore the z-coordinate component (exception: we could still use this information to "hide" objects drawn behind others)

A (viewing) transform pipeline

jsbin.com/ficoxeh

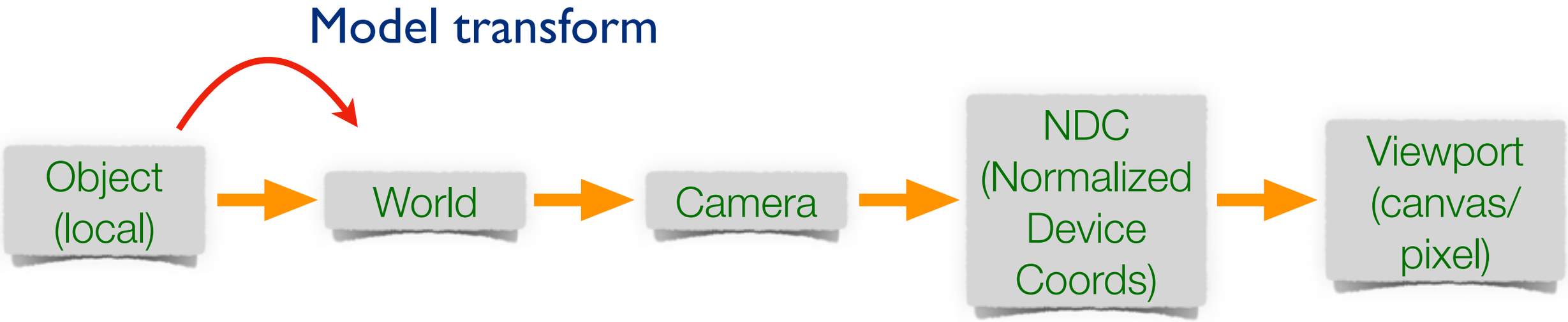
Week7/Demo4



A (viewing) transform pipeline

jsbin.com/ficoxeh

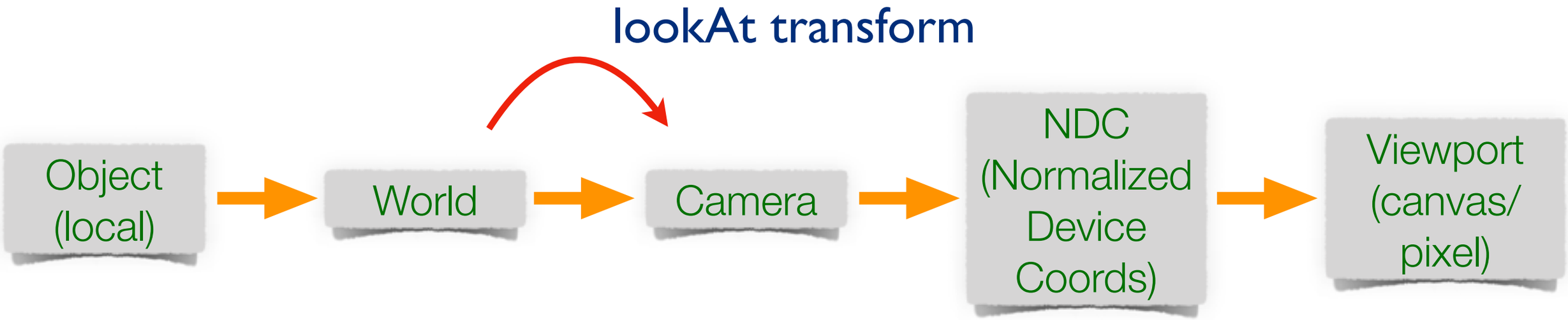
Week7/Demo4



A (viewing) transform pipeline

jsbin.com/ficoxeh

Week7/Demo4

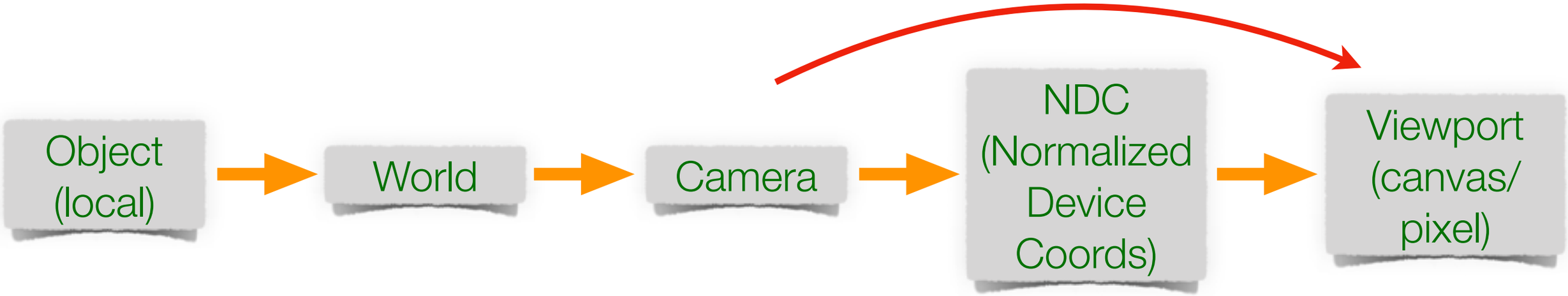


A (viewing) transform pipeline

jsbin.com/ficoxeh

Week7/Demo4

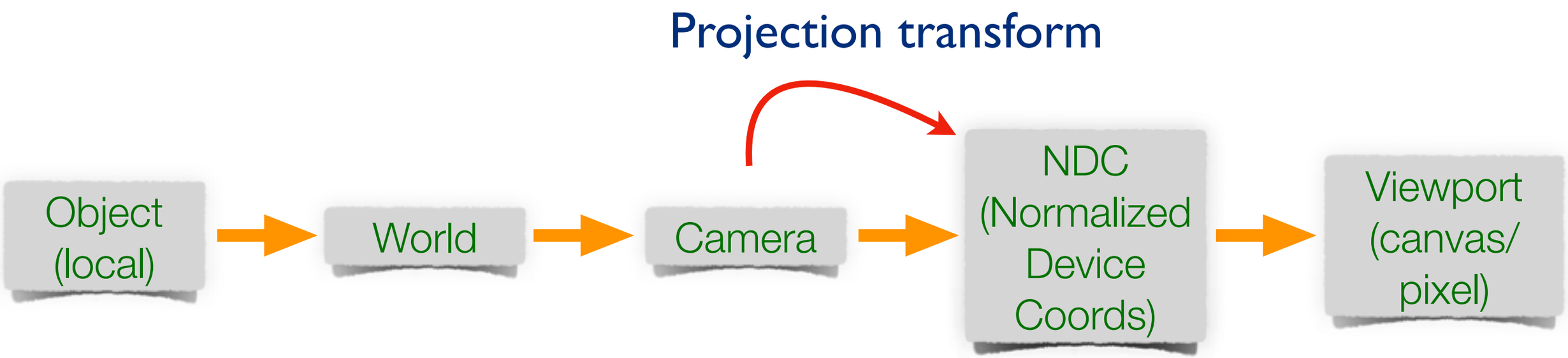
Just "throw away the z-value" ???



A (viewing) transform pipeline

jsbin.com/ficoxeh

Week7/Demo4

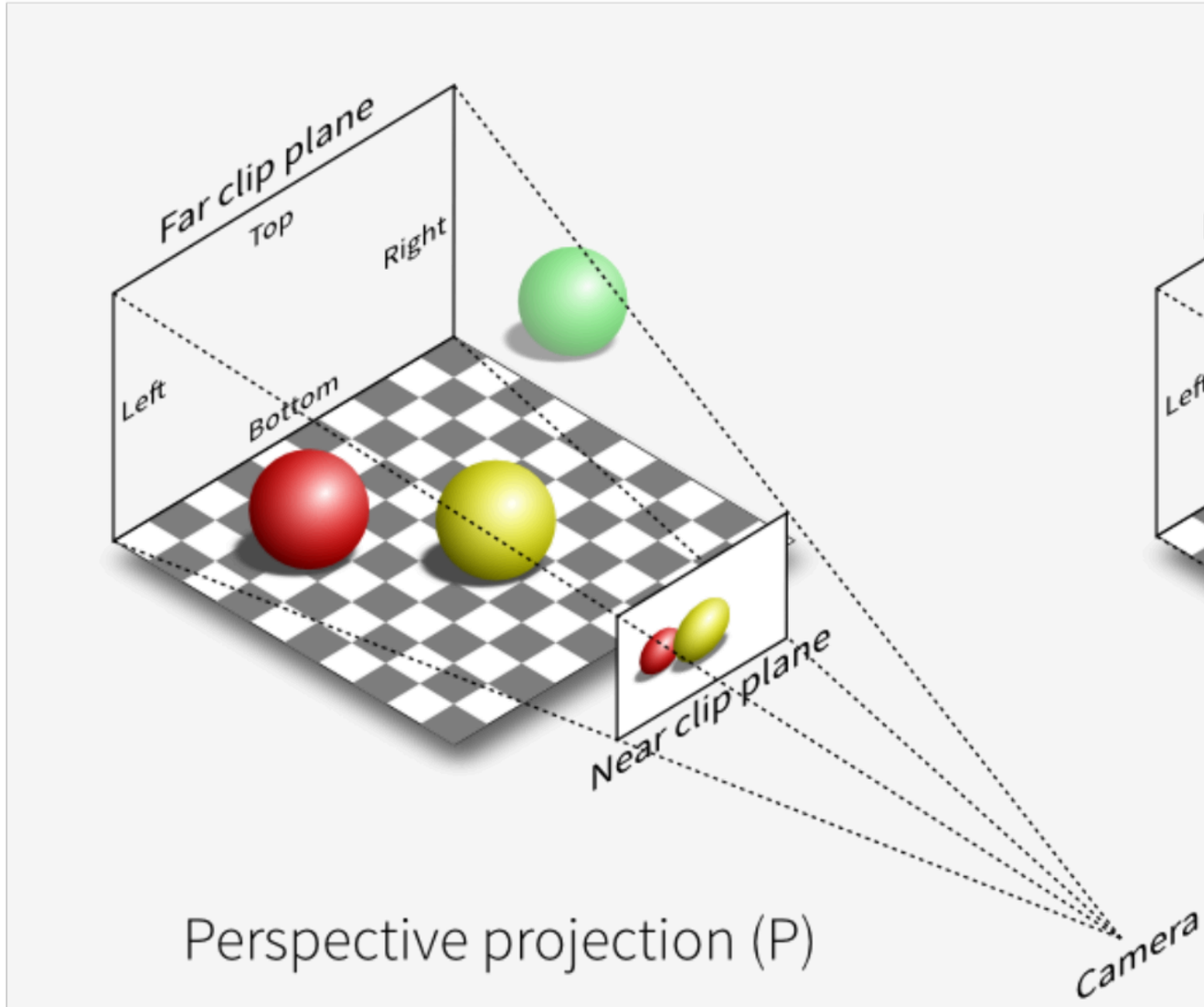


Semantics:
Project/transform the viewing frustum into the cube $[-1,1]^3$

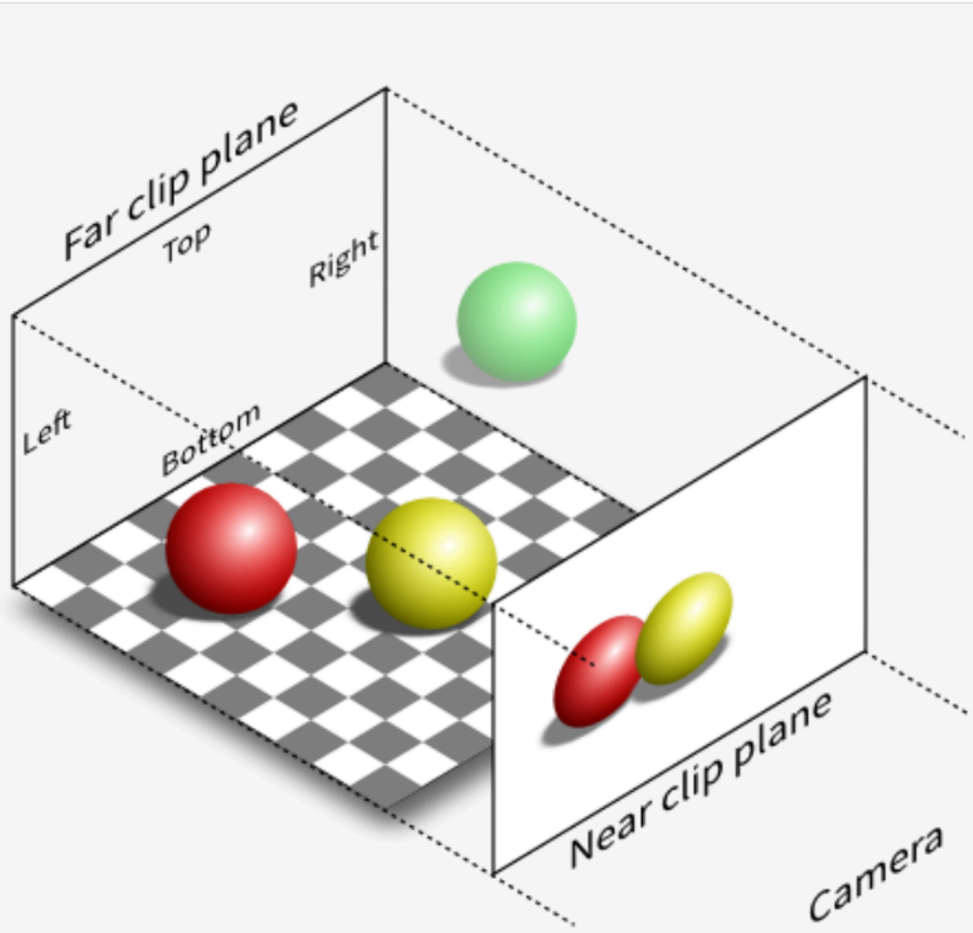
Orthographic/Perspective Projections

jsbin.com/ficoxeh

Week7/Demo4



Perspective projection (P)

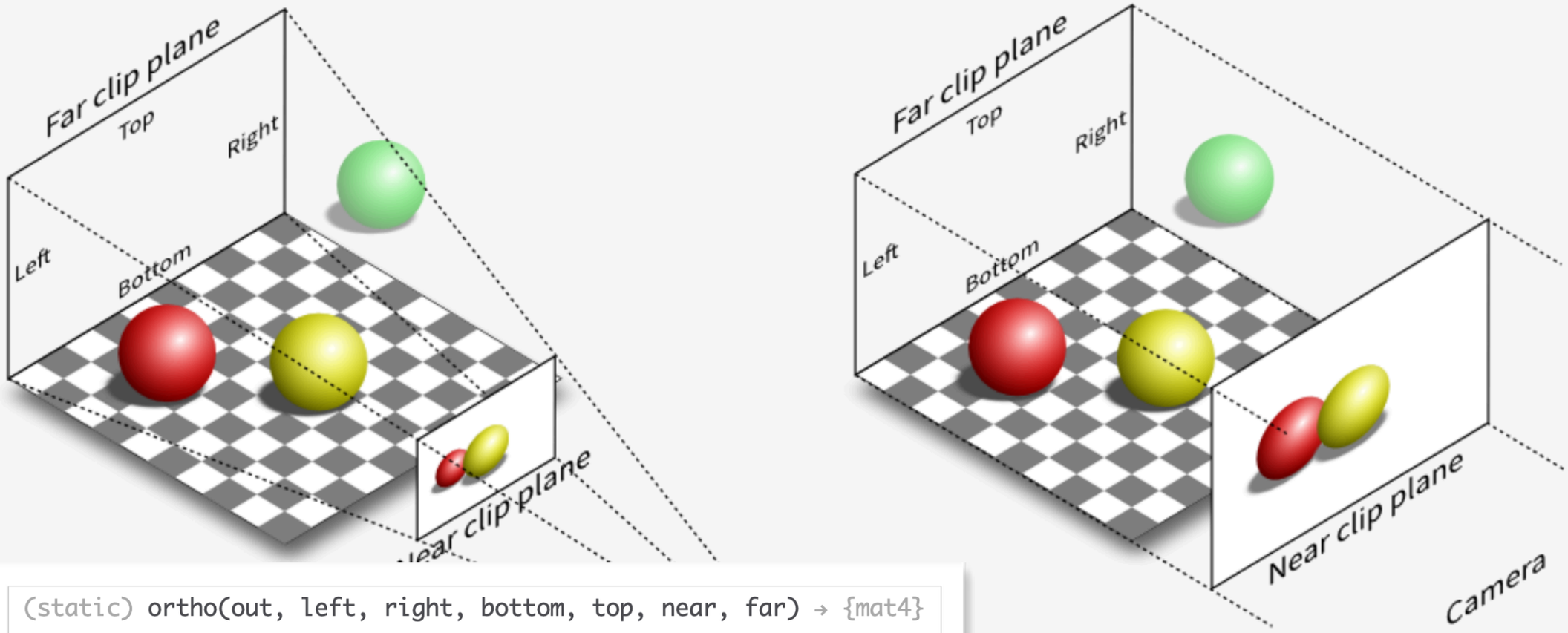


Orthographic projection (O)

Orthographic/Perspective Projections

jsbin.com/ficoxeh

Week7/Demo4



Orthographic projection (O)

`(static) ortho(out, left, right, bottom, top, near, far) → {mat4}`

Generates a orthogonal projection matrix with the given bounds

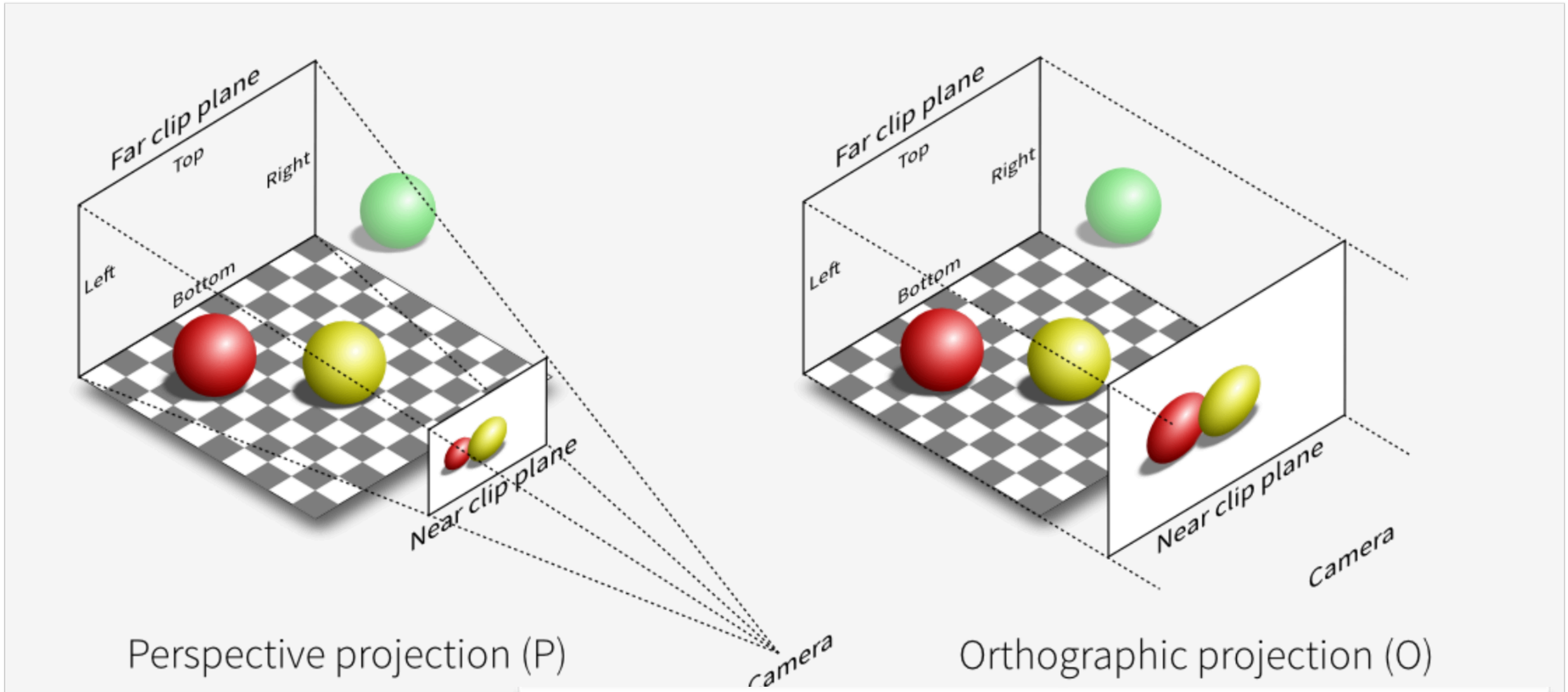
Parameters:

Name	Type	Description
`out`	mat4	mat4 frustum matrix will be written into
`left`	number	Left bound of the frustum
`right`	number	Right bound of the frustum
`bottom`	number	Bottom bound of the frustum
`top`	number	Top bound of the frustum
`near`	number	Near bound of the frustum
`far`	number	Far bound of the frustum

Orthographic/Perspective Projections

jsbin.com/ficoxeh

Week7/Demo4



(static) perspective(out, fovy, aspect, near, far) → {mat4}

Generates a perspective projection matrix with the given bounds. Passing null/undefined/no value for far will generate infinite projection matrix.

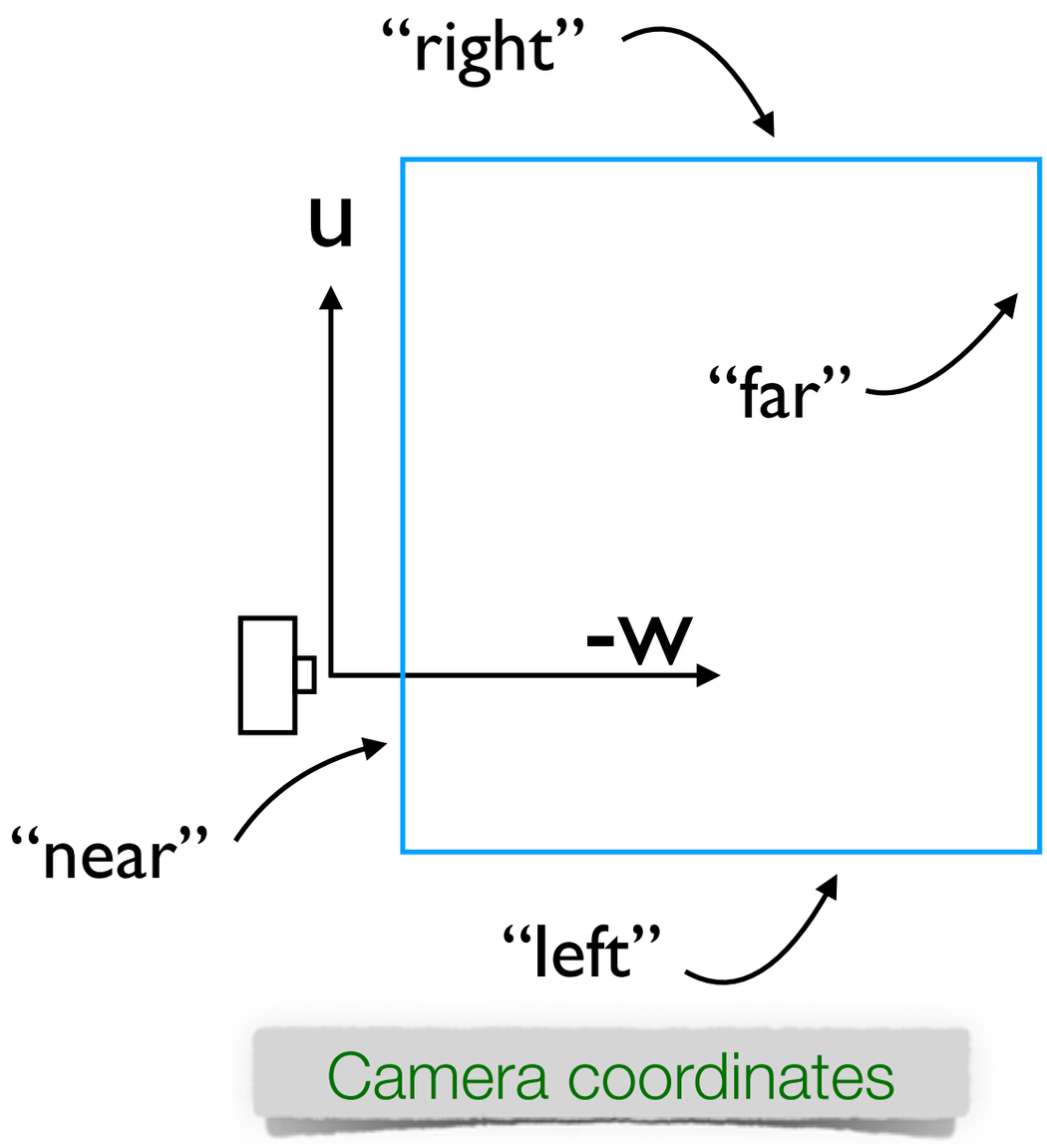
Parameters:

Name	Type	Description
out	mat4	mat4 frustum matrix will be written into
fovy	number	Vertical field of view in radians
aspect	number	Aspect ratio. typically viewport width/height
near	number	Near bound of the frustum
far	number	Far bound of the frustum, can be null or Infinity

Orthographic Projection

jsbin.com/ficoxeh

Week7/Demo4

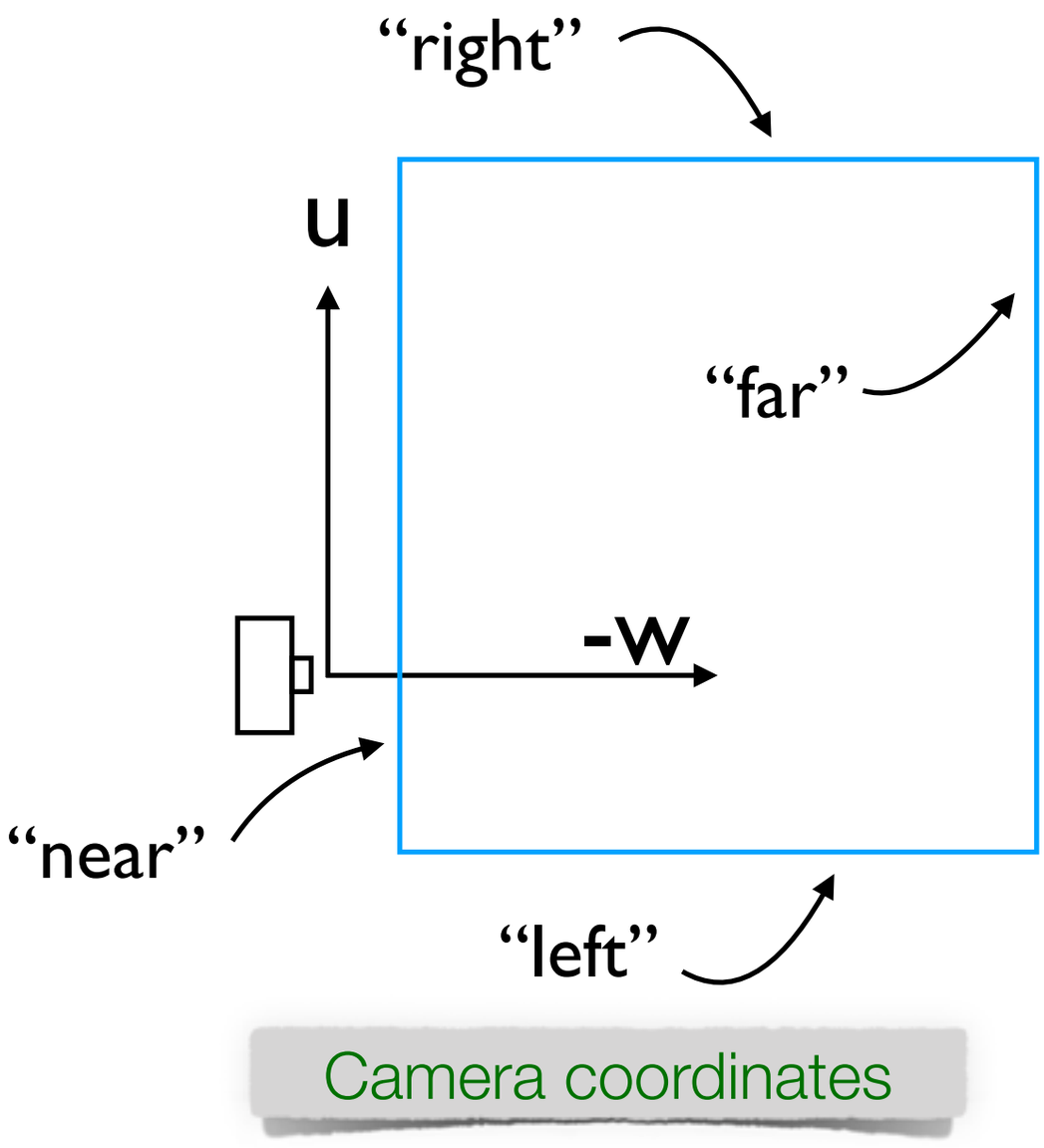


Camera coordinates

Orthographic Projection

jsbin.com/ficoxeh

Week7/Demo4



Camera coordinates

`(static) ortho(out, left, right, bottom, top, near, far) → {mat4}`

Generates a orthogonal projection matrix with the given bounds

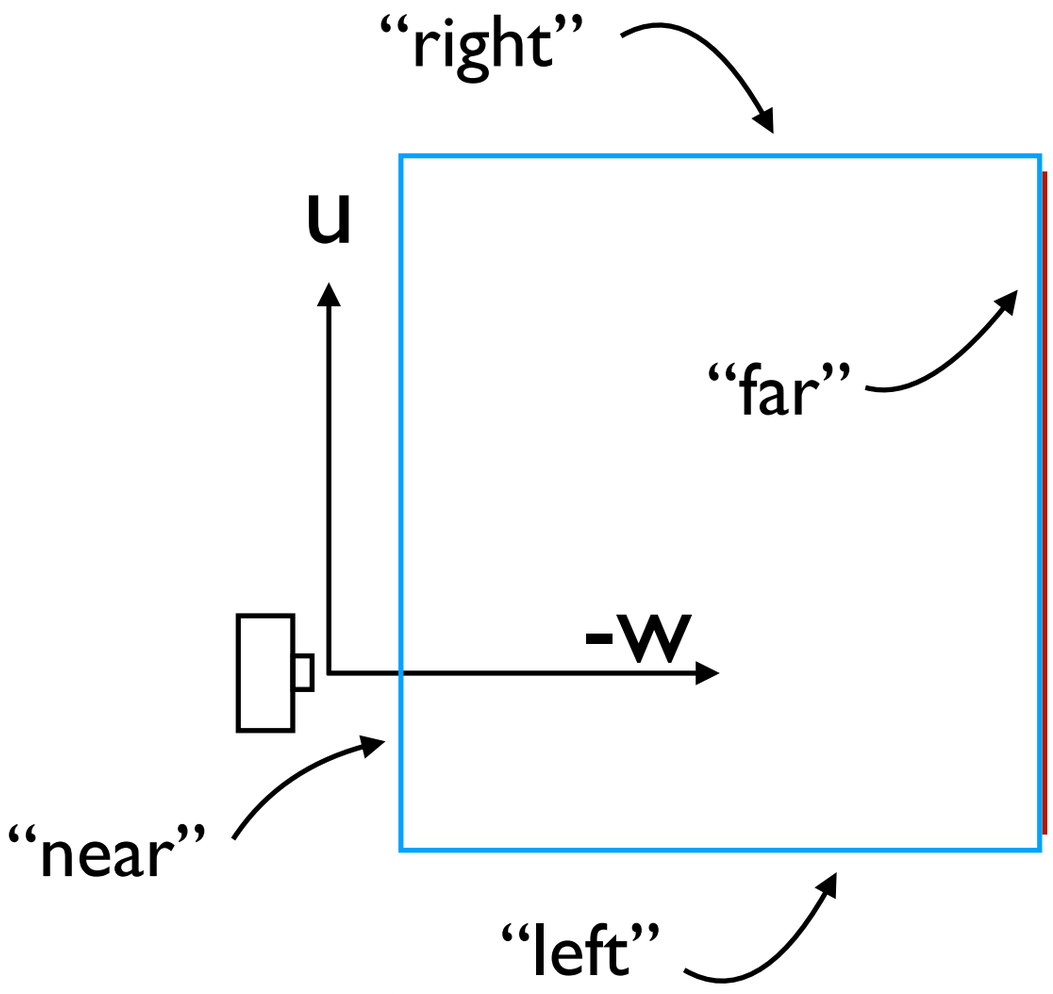
Parameters:

Name	Type	Description
`out`	mat4	mat4 frustum matrix will be written into
`left`	number	Left bound of the frustum
`right`	number	Right bound of the frustum
`bottom`	number	Bottom bound of the frustum
`top`	number	Top bound of the frustum
`near`	number	Near bound of the frustum
`far`	number	Far bound of the frustum

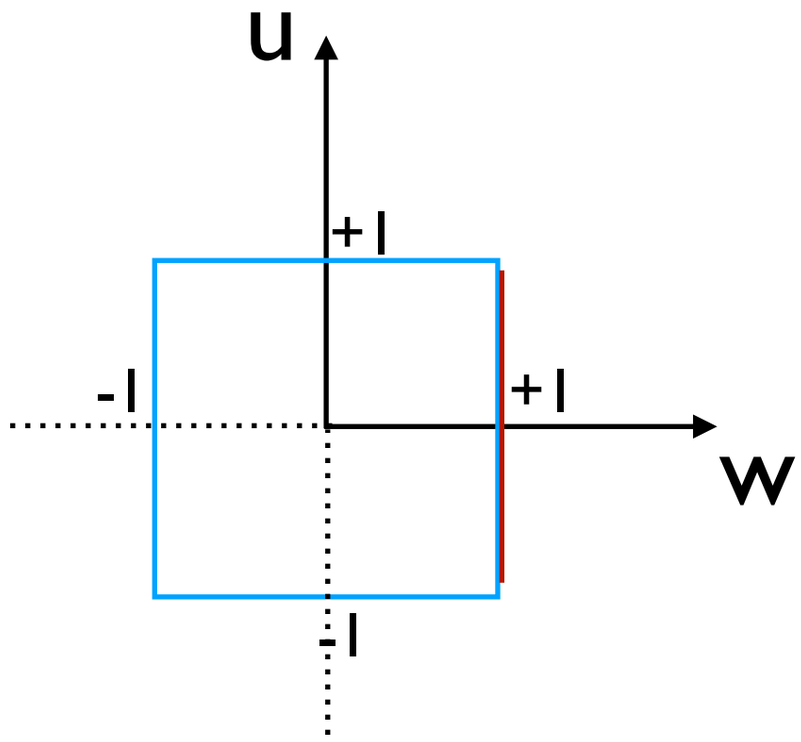
Orthographic Projection

jsbin.com/ficoxeh

Week7/Demo4



Camera coordinates

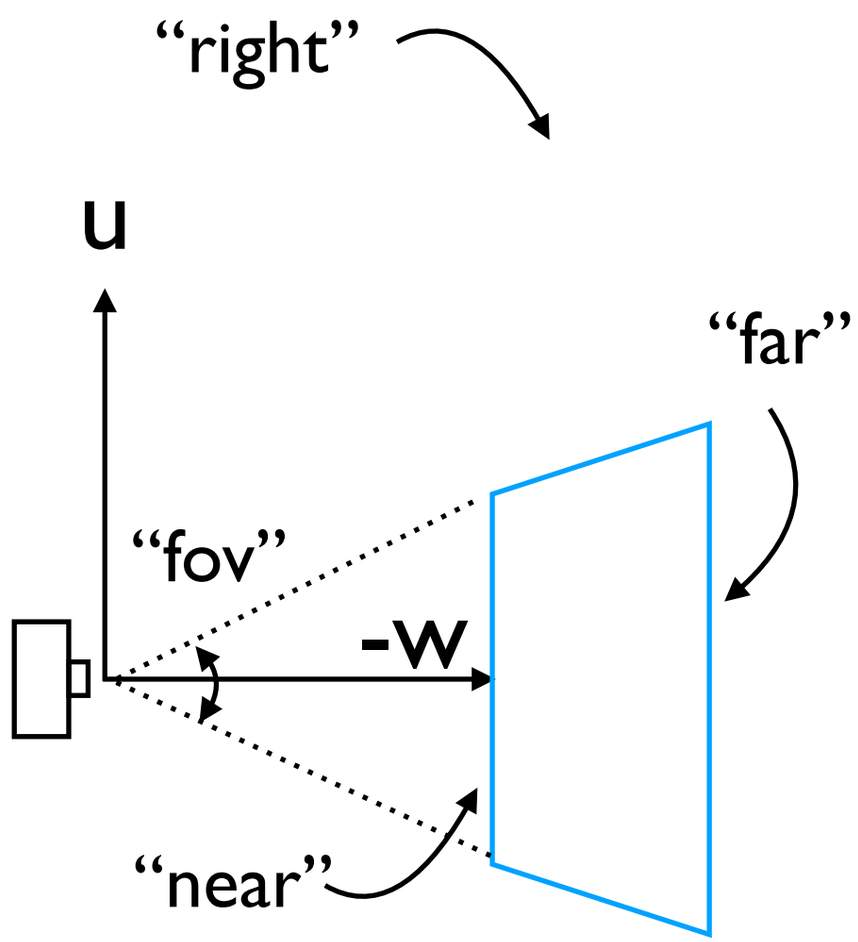


Normalized Device coordinates

Perspective Projection

jsbin.com/ficoxeh

Week7/Demo4

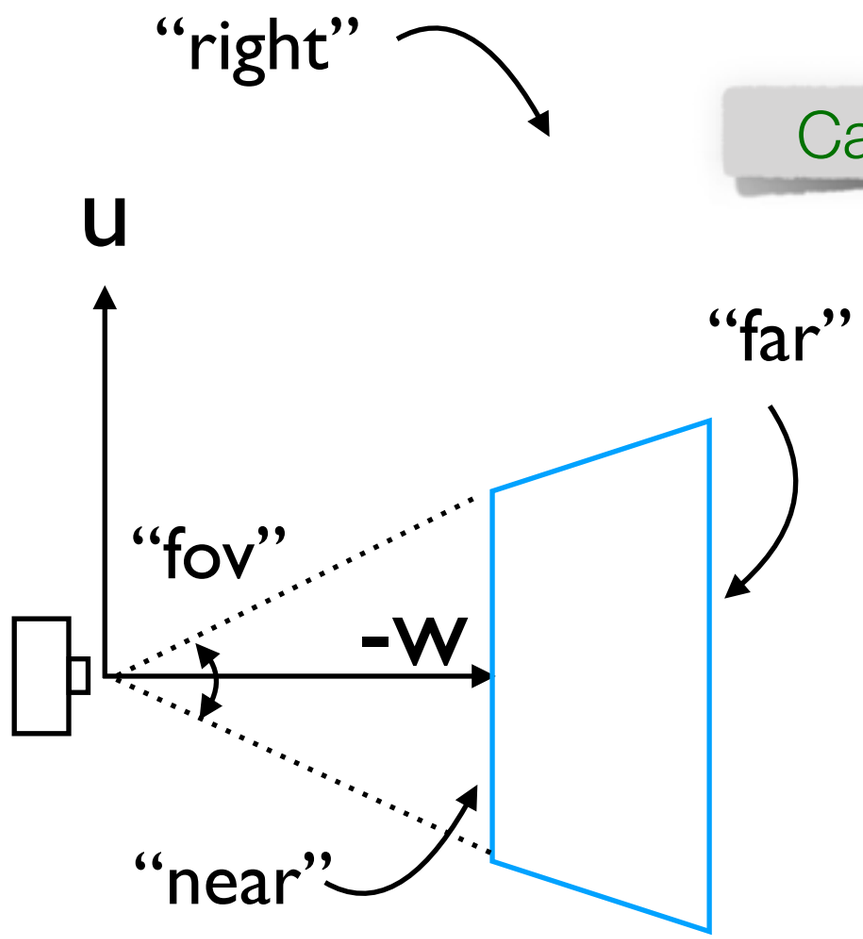


Camera coordinates

Perspective Projection

jsbin.com/ficoxeh

Week7/Demo4



Camera coordinates

`(static) perspective(out, fovy, aspect, near, far) → {mat4}`

Generates a perspective projection matrix with the given bounds. Passing null/undefined/no value for far will generate infinite projection matrix.

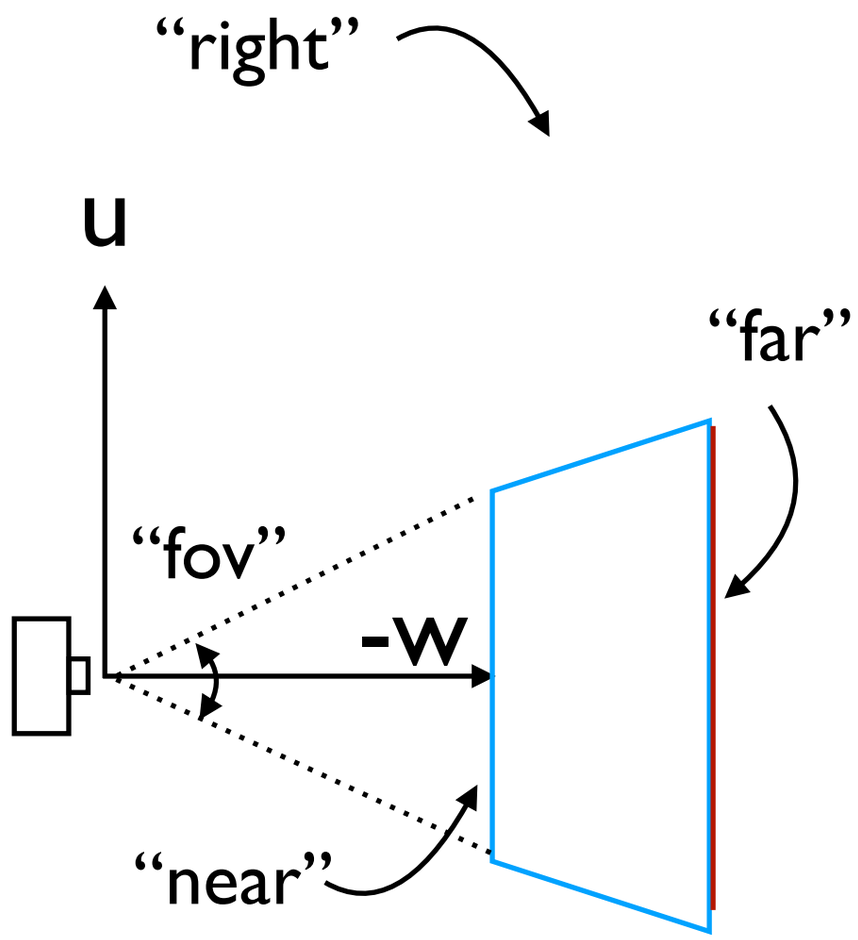
Parameters:

Name	Type	Description
`out`	mat4	mat4 frustum matrix will be written into
`fovy`	number	Vertical field of view in radians
`aspect`	number	Aspect ratio. typically viewport width/height
`near`	number	Near bound of the frustum
`far`	number	Far bound of the frustum, can be null or Infinity

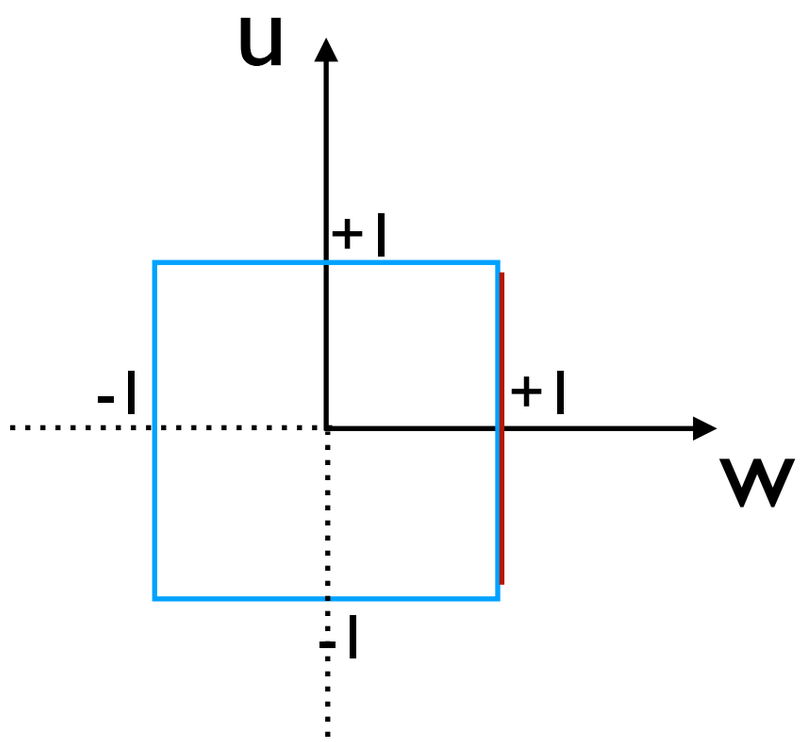
Perspective Projection

jsbin.com/ficoxeh

Week7/Demo4



Camera coordinates



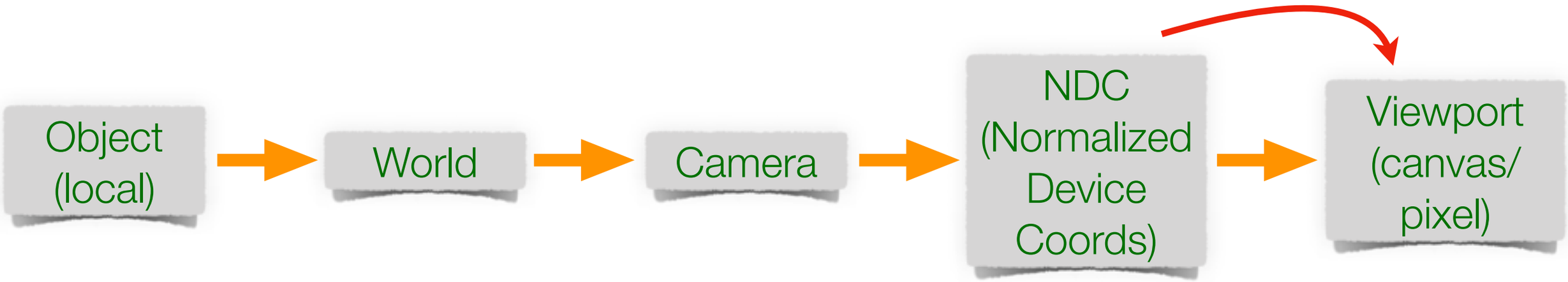
Normalized Device coordinates

Note: glMatrix (and some other libraries) adopt the convention that the near/far clipping parameters are measured along the negative w axis (i.e. given as positive values, with near<far)

jsbin.com/ficoxeh

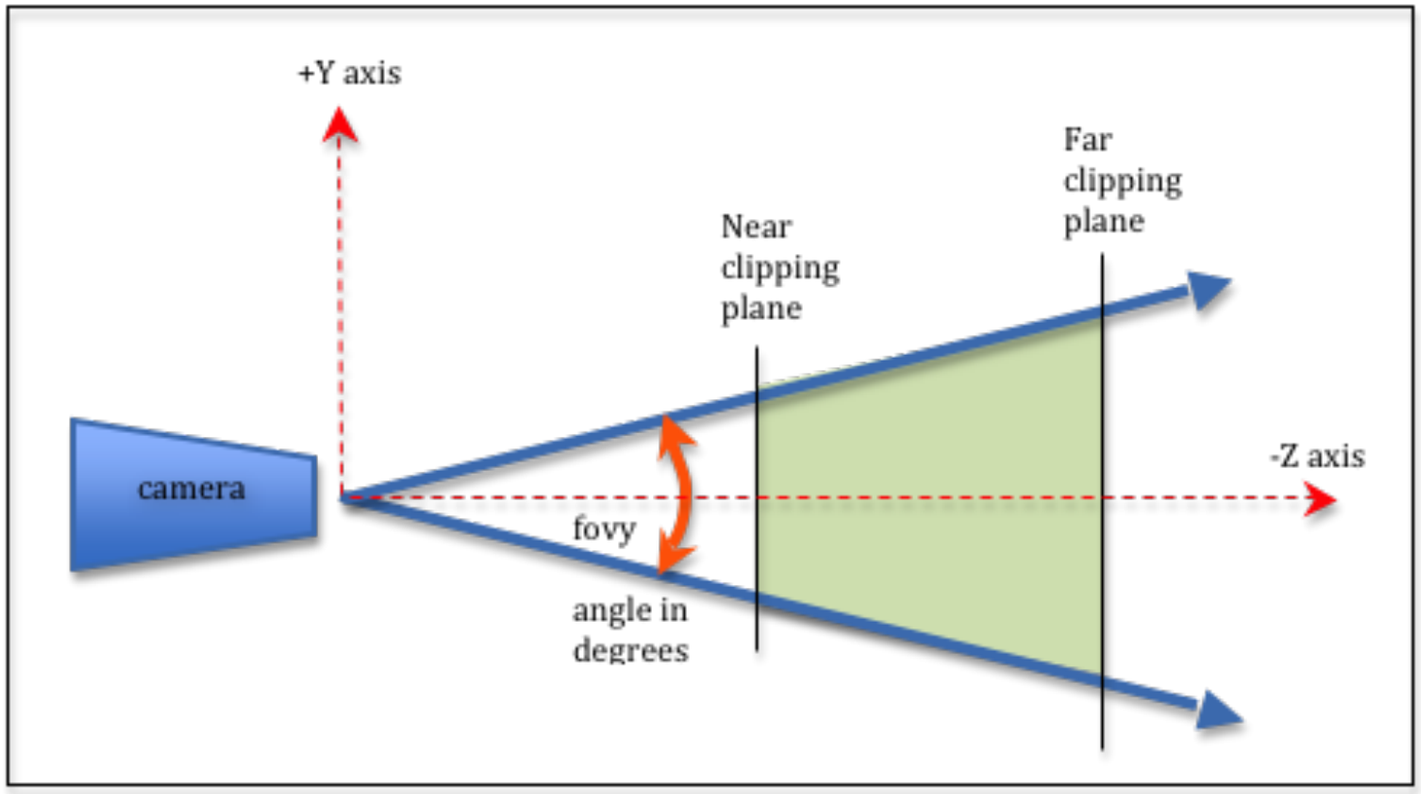
Week7/Demo4

Viewport transform

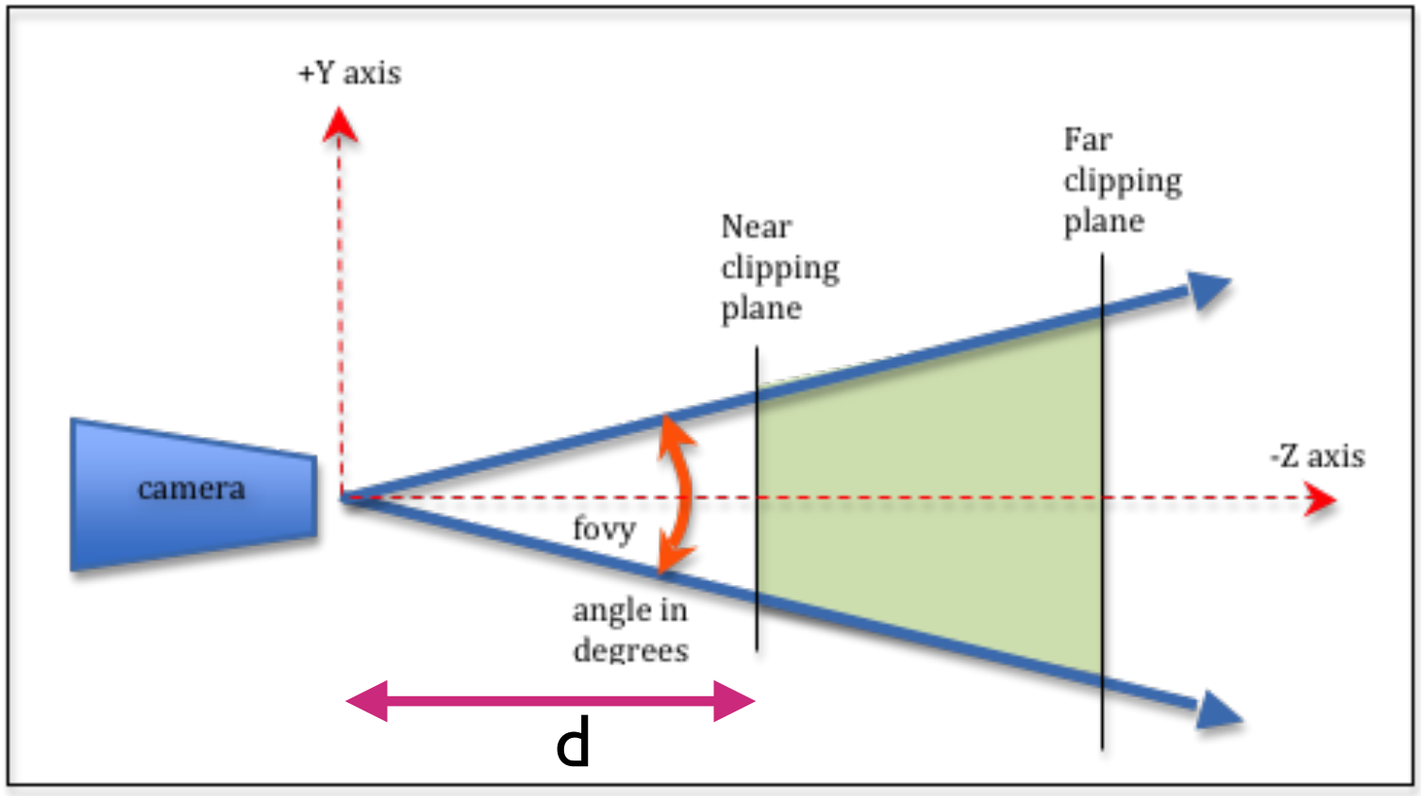


Semantics:
Scale/translate the view frustum (assuming cube $[-1,1]^3$) into the viewport

Algebra of perspective projection

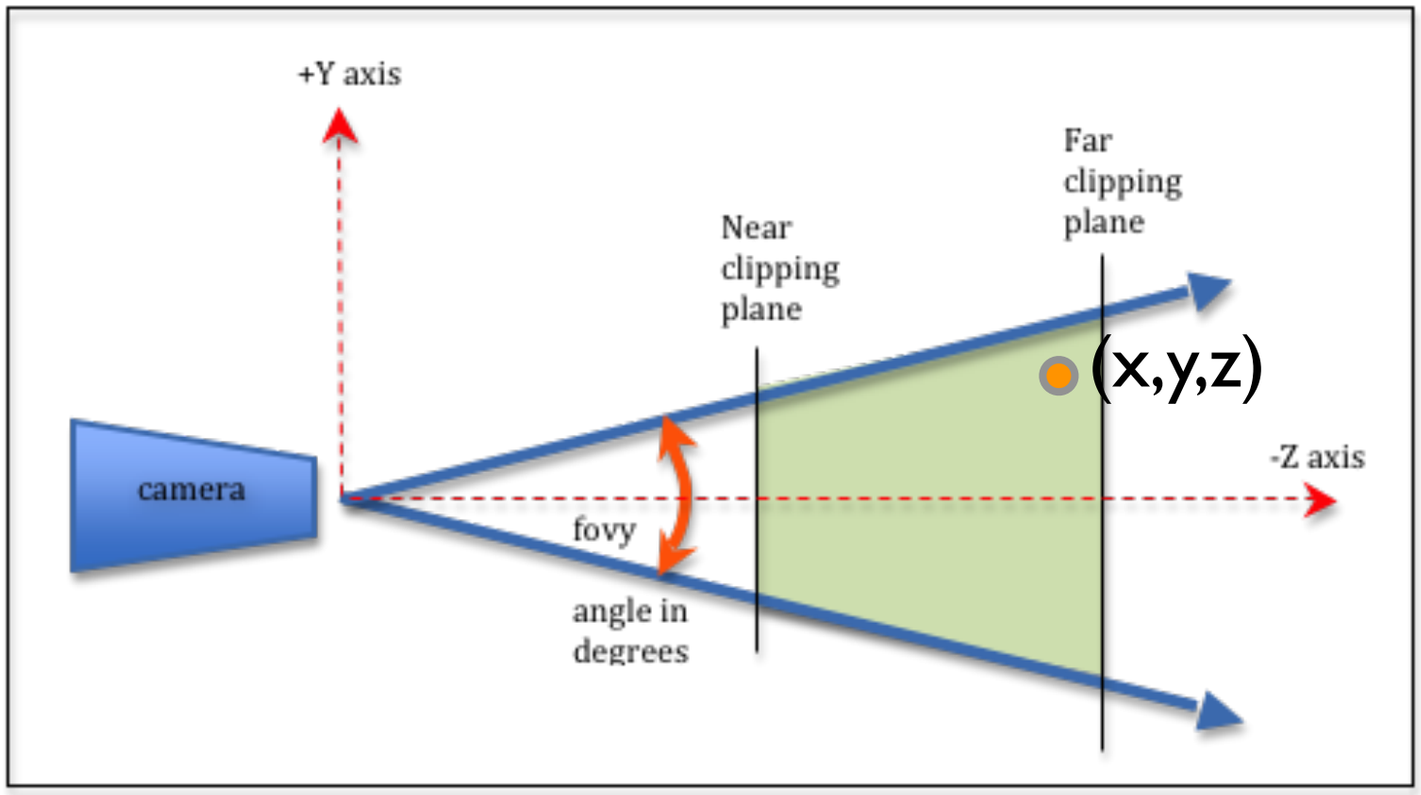


Algebra of perspective projection



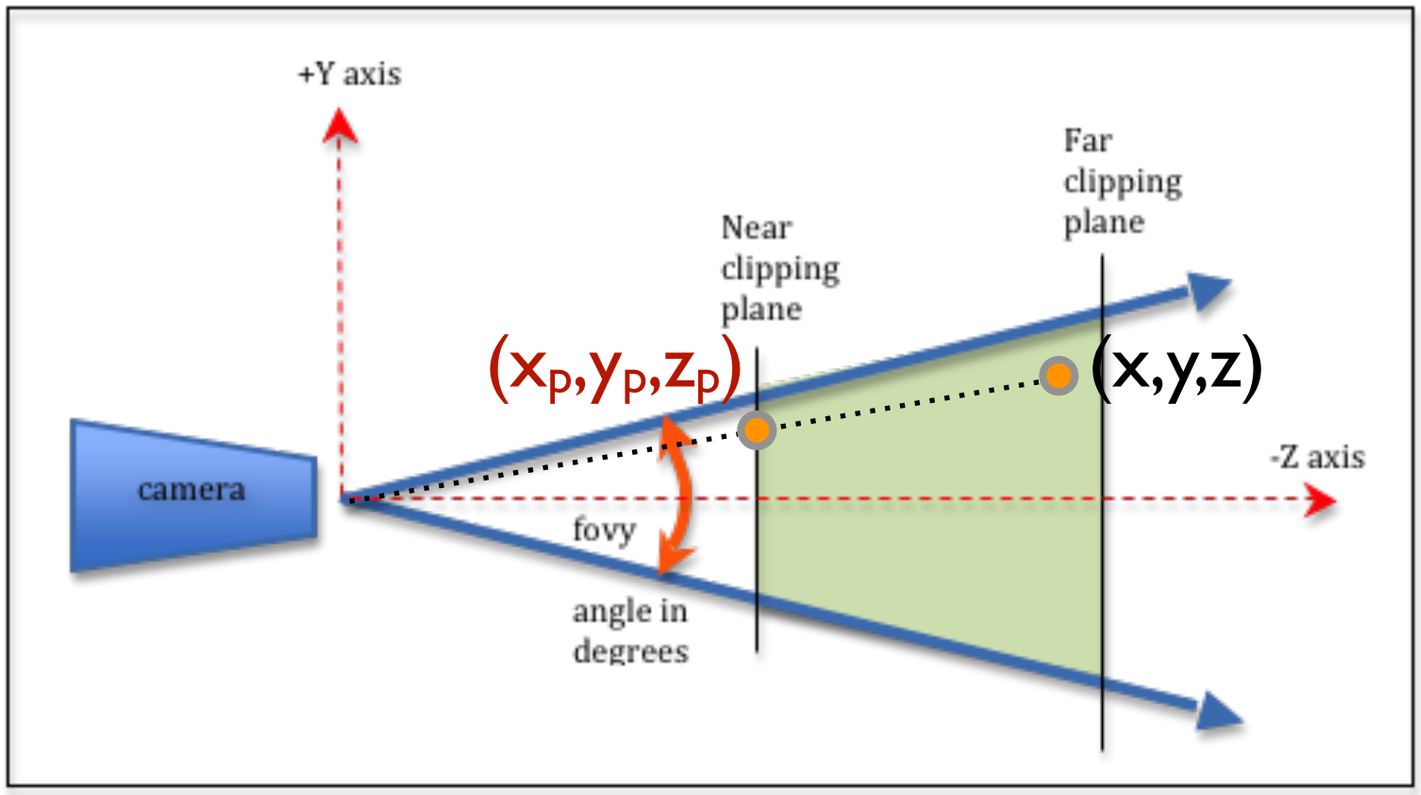
The *focal length* **d** is the distance of the "center" or projection - the "eye" in camera terms - to the projection plane (here we identified the projection plane with the near clipping plane)

Algebra of perspective projection



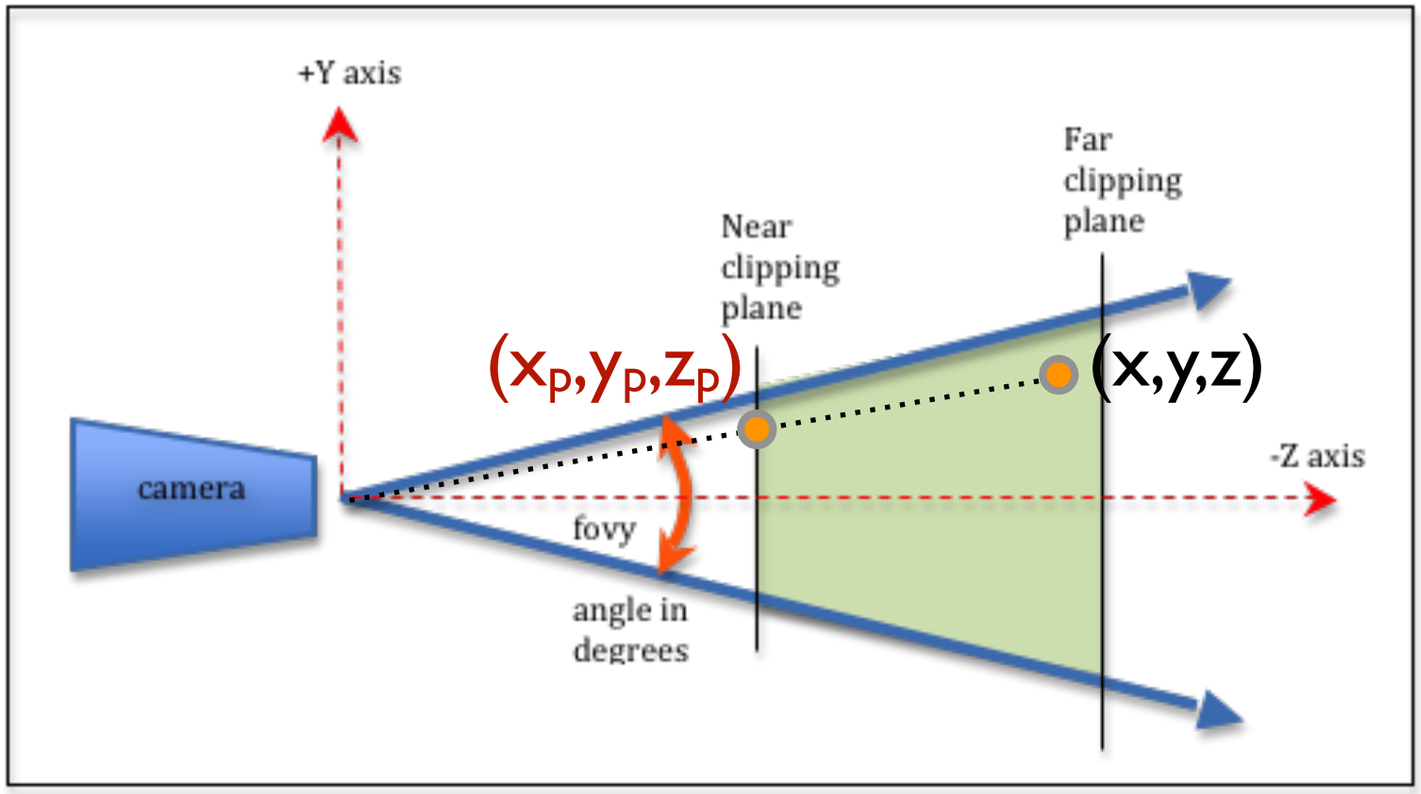
A point (x,y,z) in the view frustum …

Algebra of perspective projection



… projects to a point (x_p,y_p,z_p) in the "image plane"

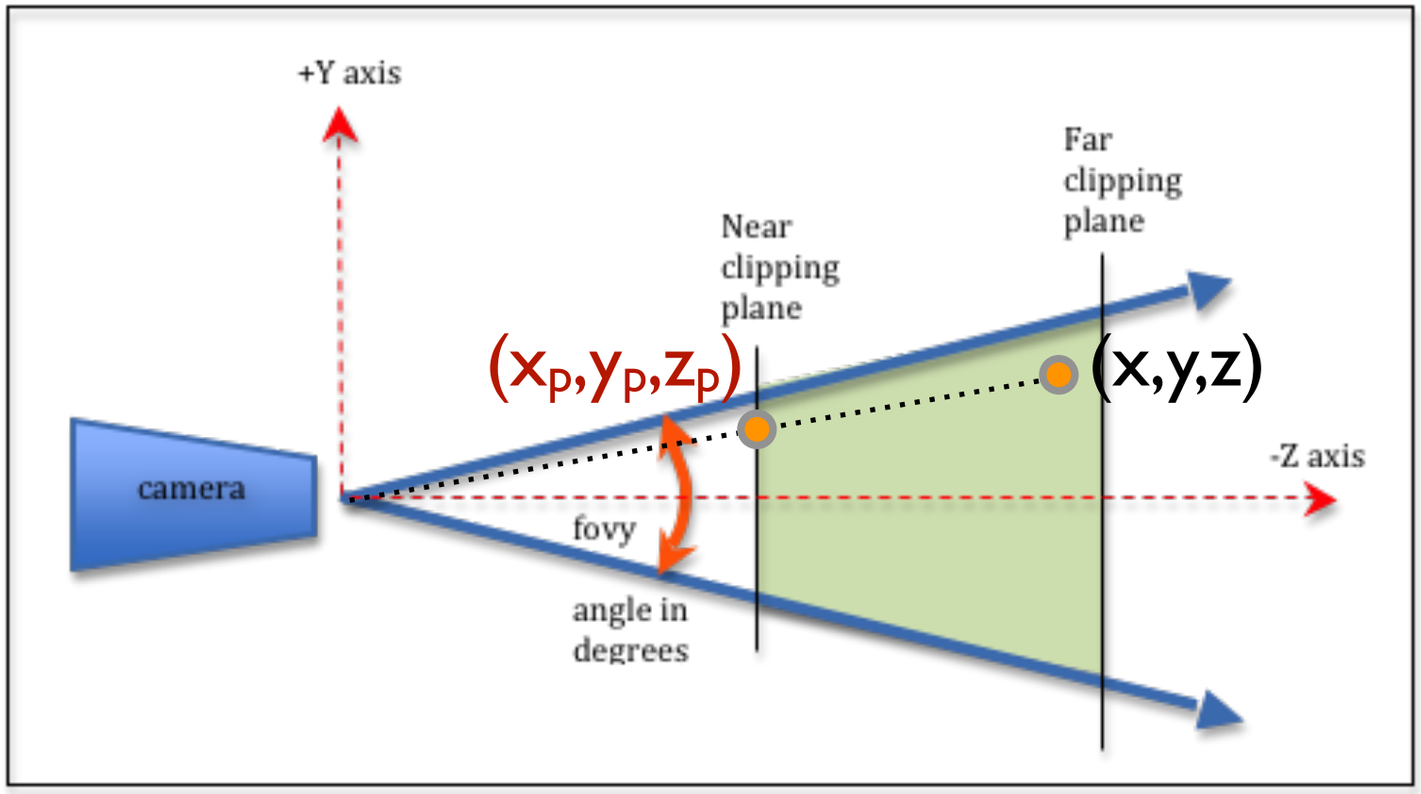
Algebra of perspective projection



Based on the geometry of the projection operation, the coordinates of the projection are as given by:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} d\frac{x}{z} \\ d\frac{y}{z} \\ -d \end{pmatrix}$$

Algebra of perspective projection



Problem : Division by z makes this transform not be linear anymore!

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} d\frac{x}{z} \\ d\frac{y}{z} \\ -d \end{pmatrix}$$

Homogeneous representations (revisited)

We previously envisioned homogeneous vector representations (in 3D) as 4-vectors with an extra “1” appended (but semantically identical)

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \Leftrightarrow \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Homogeneous representations (revisited)

We previously envisioned homogeneous vector representations (in 3D) as 4-vectors with an extra “1” appended (but semantically identical)

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \Leftrightarrow \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

… but this is just a special case of the full-fledged potential of this representation!
Let’s see a different definition!

Homogeneous representations (revisited)

A homogeneous representation of a 3D location (x,y,z) is a quadruple of numbers with the following properties:

Homogeneous representations (revisited)

A homogeneous representation of a 3D location (x,y,z) is a quadruple of numbers with the following properties:

Property #1:
The 4-vector (x,y,z,1) ***is*** *an equivalent homogeneous representation of the 3D vector (x,y,z)*
(but not necessarily the only one!)

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \Leftrightarrow \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Homogeneous representations (revisited)

A homogeneous representation of a 3D location (x,y,z) is a quadruple of numbers with the following properties:

Property #1:
The 4-vector (x,y,z,1) ***is*** *an equivalent homogeneous representation of the 3D vector (x,y,z)*
(but not necessarily the only one!)

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \Leftrightarrow \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Property #2:
Two homogeneous 4-vectors are treated as *equal (i.e. representing the same geometric entity)* if their respective elements are scaled copies of one another

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \equiv \begin{pmatrix} \alpha x \\ \alpha y \\ \alpha z \\ \alpha w \end{pmatrix}$$

Homogeneous representations (revisited)

So, what is the geometric 3D vector that corresponds to a homogeneous quadruple that does *not* have a “1” at the end?

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Homogeneous representations (revisited)

So, what is the geometric 3D vector that corresponds to a homogeneous quadruple that does *not* have a "1" at the end?

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \equiv \begin{pmatrix} x/w \\ y/w \\ z/w \\ w/w = 1 \end{pmatrix}$$

(divide by the last entry, to "create" an ace at the end!)

Homogeneous representations (revisited)

So, what is the geometric 3D vector that corresponds to a homogeneous quadruple that does *not* have a "1" at the end?

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \equiv \begin{pmatrix} x/w \\ y/w \\ z/w \\ w/w = 1 \end{pmatrix} \Leftrightarrow \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

(divide by the last entry, to "create" an ace at the end!)

Perspective as a homogeneous transform?

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Consider the point (x,y,z), in homogeneous representation, transformed by a “special matrix” (note the pattern …)

Perspective as a homogeneous transform?

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ -z \\ z/d \end{pmatrix}$$

The transformation (still a matrix-vector multiplication) produces a different homogeneous vector

Perspective as a homogeneous transform?

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ -z \\ z/d \end{pmatrix} \equiv \begin{pmatrix} dx/z \\ dy/z \\ -d \\ 1 \end{pmatrix}$$

We divide everything by (z/d) to create an ace at the last place …

Perspective as a homogeneous transform?

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ -z \\ z/d \end{pmatrix} \equiv \begin{pmatrix} dx/z \\ dy/z \\ -d \\ 1 \end{pmatrix} \Leftrightarrow \begin{pmatrix} x_p = dx/z \\ y_p = dy/z \\ z_p = -d \end{pmatrix}$$

Which produces exactly the homogeneous representation of the perspective projection!

Perspective as a homogeneous transform?

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ -z \\ z/d \end{pmatrix} \equiv \begin{pmatrix} dx/z \\ dy/z \\ -d \\ 1 \end{pmatrix} \Leftrightarrow \begin{pmatrix} x_p = dx/z \\ y_p = dy/z \\ z_p = -d \end{pmatrix}$$

Which produces exactly the homogeneous representation of the perspective projection!

(In practice the perspective matrix created by glMatrix is a bit more complex, but still a 4x4 matrix, and we need to do the division in the end to convert to a "geometric" location!)

Other considerations about drawing in 3D

$$\mathbf{f}(u) = \mathbf{uBP}$$

Same representation applies for polynomial curves in 3D - but f(u) here returns a 3-vector

$$\mathbf{P} = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

Correspondingly, the **P** matrix for cubics would have dimension 4x3 (each point is a 3-vector)

Other considerations about drawing in 3D

jsbin.com/sumoxexiku

Rotations are more complex in 3D
All rotations *can* be described as rotations around an axis, but might not be as convenient to create them in this fashion (look in textbook about “Euler Angles”)

(static) fromRotation(out, rad, axis) → {mat4}

Creates a matrix from a given angle around a given axis This is equivalent to (but much faster than): mat4.identity(dest); mat4.rotate(dest, dest, rad, axis);

Parameters:

Name	Type	Description
out	mat4	mat4 receiving operation result
rad	Number	the angle to rotate the matrix by
axis	ReadonlyVec3	the axis to rotate around

Source: mat4.js, line 937

But for those cases that this definition can work, glMatrix provides a functionality that generates such rotation matrix/transform