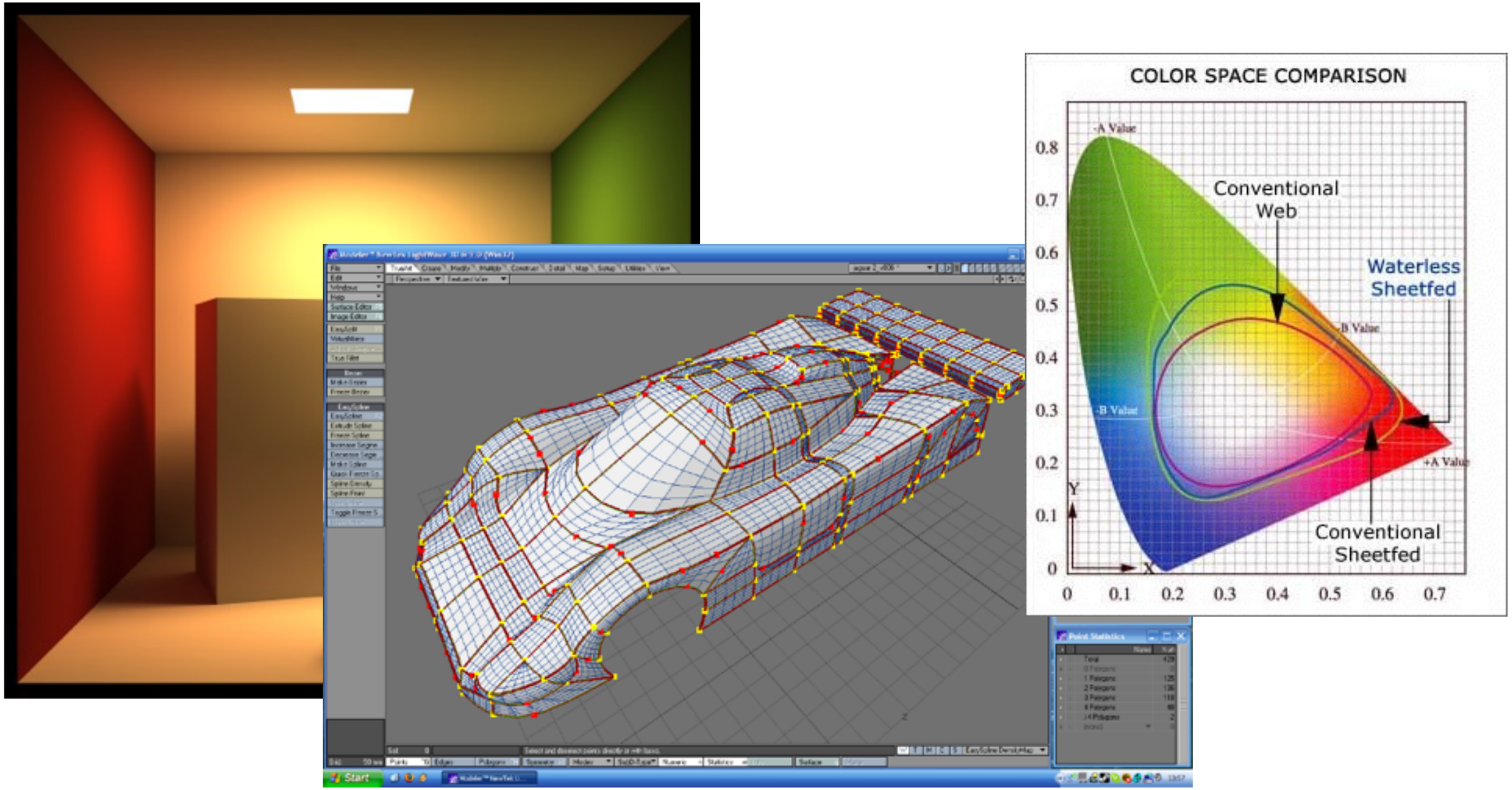


Lecture 13 : More details on Viewing and Drawing in 3D: Coordinate systems, transforms, representations

Thursday October 21st 2021

Logistics (reminder)

- Midterm will take place as scheduled, Friday Oct 29th, at 7:15pm. This will be an online exam (Canvas quiz)
- The entirety (or majority) of next week will be exam review. Practice exam(s) will be provided/reviewed.
- Material up to (and including) this week's lectures will be included in your midterm.
- Programming Assignment #3 due Friday.
- Assignment #4 won't be due until after the midterm.

Today's lecture

- More details about drawing/viewing in 3D
  - Key concept: What are the various coordinate systems (world/camera/NDC/viewport)
  - What are the semantics of transforms (e.g. lookAt and projection transforms) and how are they implemented in glMatrix
- Some new details about homogeneous coordinates
- There will be good reference materials to review offline (from your textbook FCG Chapter 7, and Big Fun Graphics Book Chapter 8 and Chapter 9)

Demo quick reference

2D version of the examples …

jsbin.com/bodorun

Week6/Demo1

Just moving points and transforms to 3D

jsbin.com/dovehav

Week7/Demo0

Rotating camera and lookAt transform

jsbin.com/xequtul

Week7/Demo1

Separating out viewport, camera, and model transform

jsbin.com/xudufet

Week7/Demo2

Explicit projection transform (orthographic) … and a teaser for perspective projection

jsbin.com/peredov

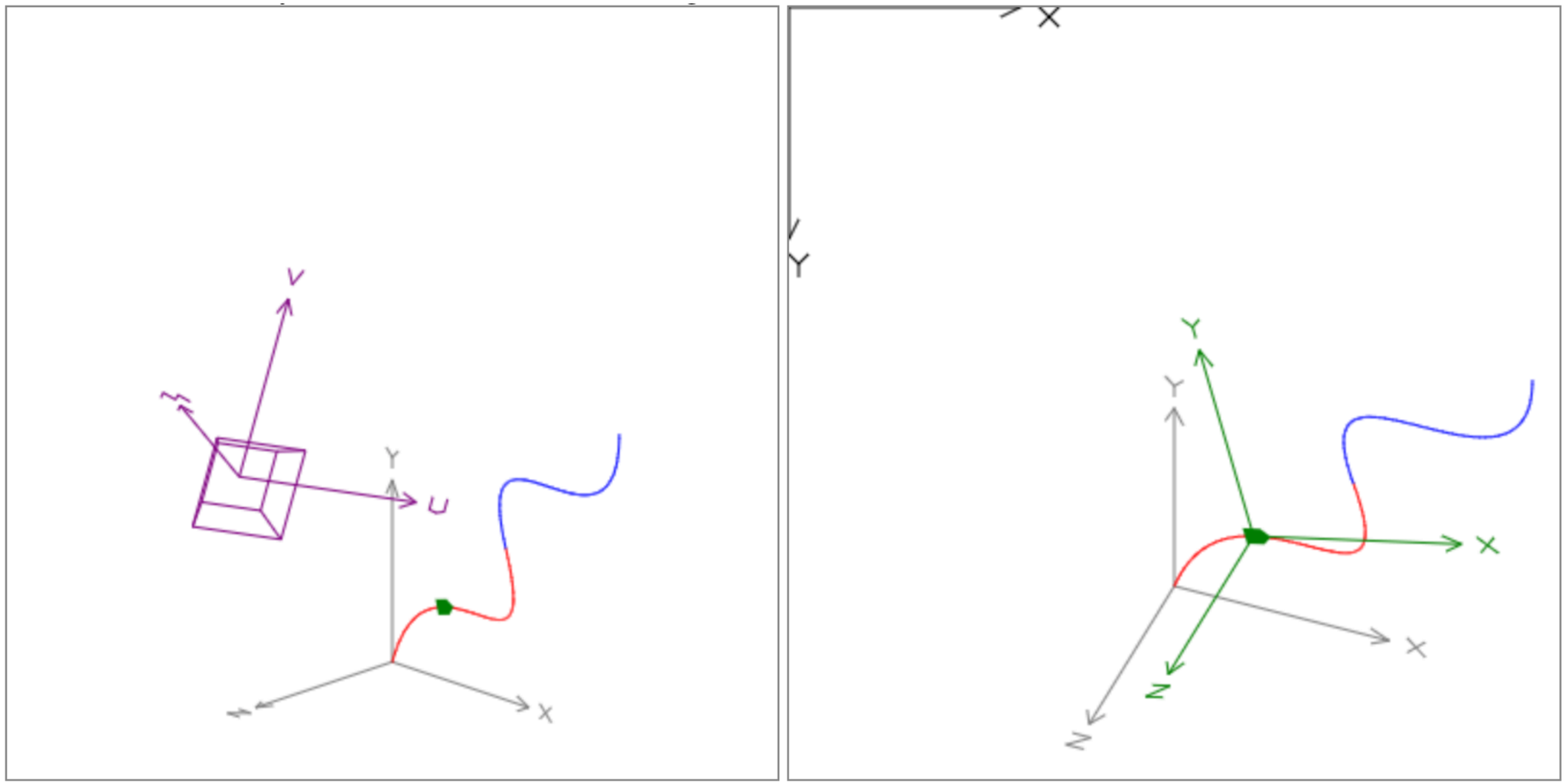
Week7/Demo3

Demo walkthrough

RECAP

jsbin.com/ficoxeh

Week7/Demo4



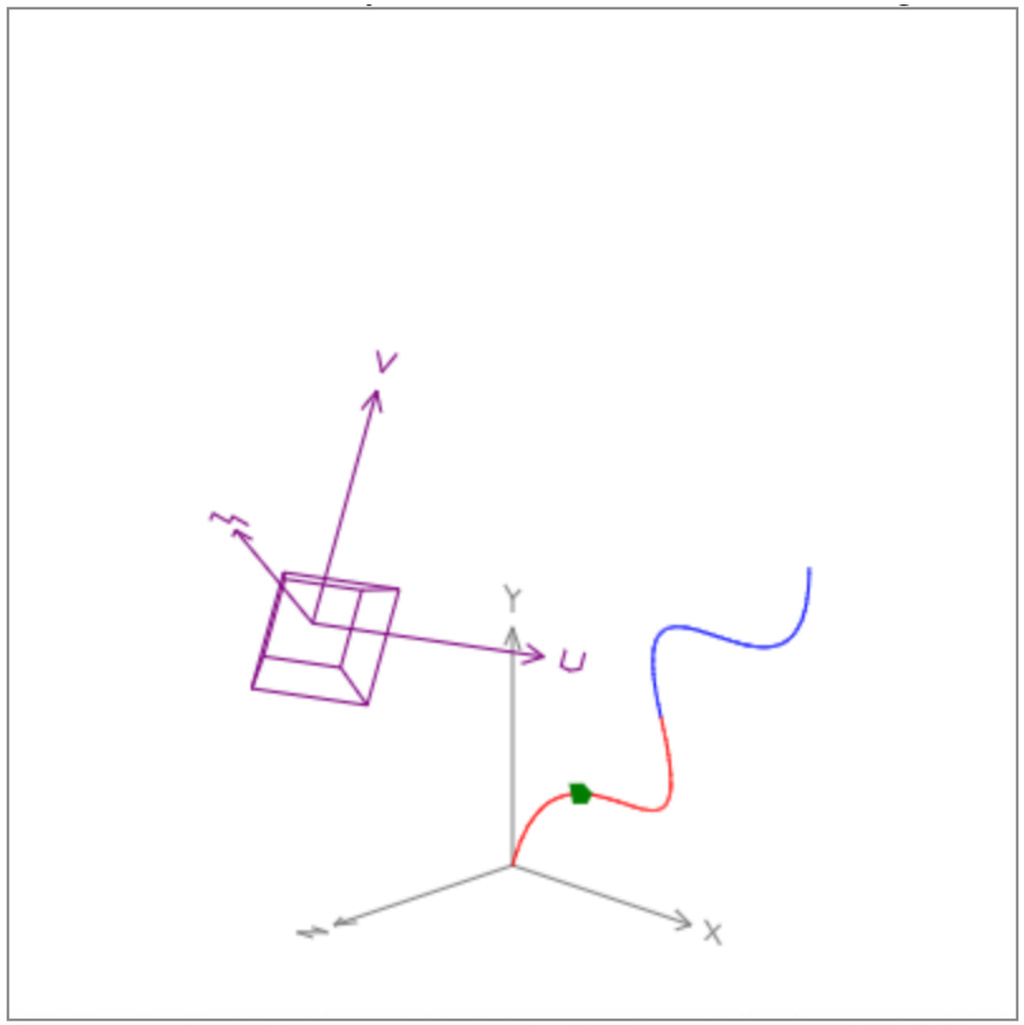
A conceptual 3D world, with some familiar content
(a piecewise-cubic curve on the XY-plane, and an object moving along it)

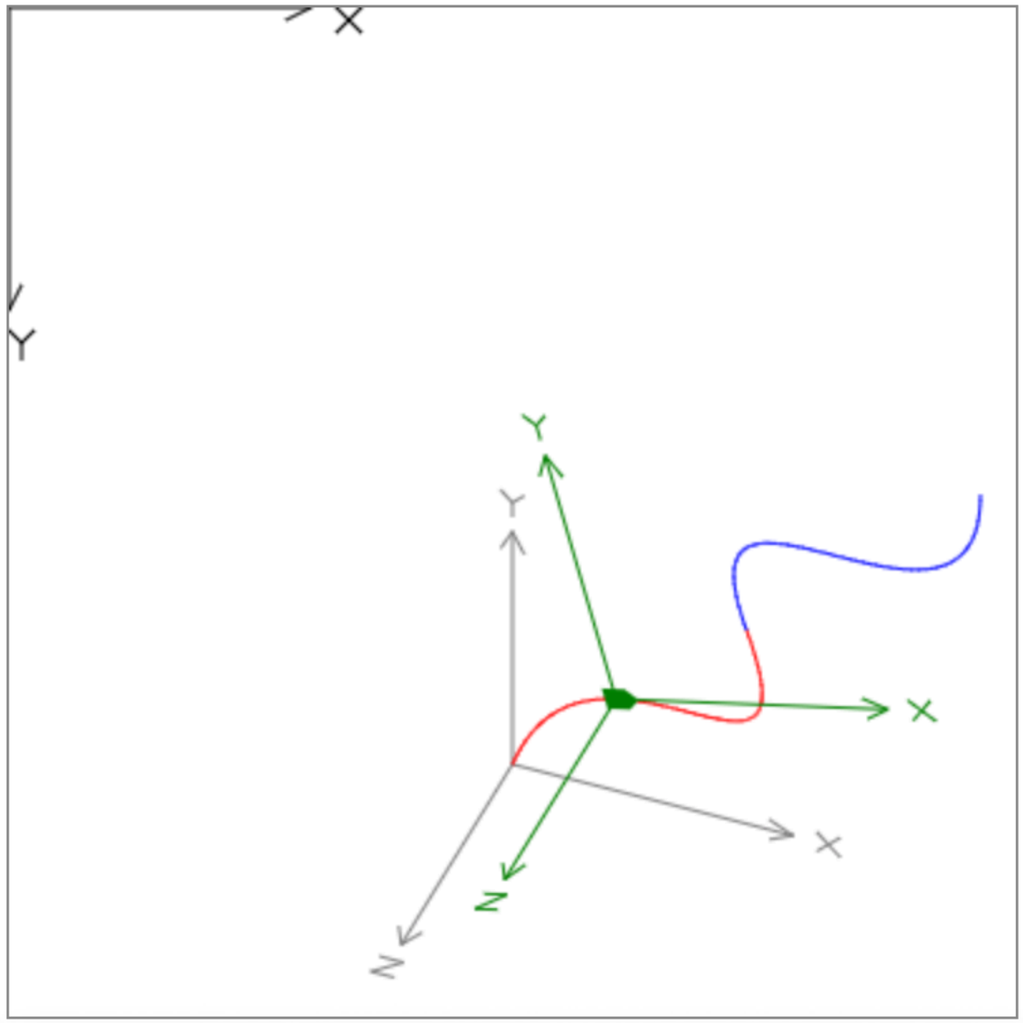
Demo walkthrough

RECAP

jsbin.com/ficoxeh

Week7/Demo4





Left window:
A view of the world from a faraway vantage point, that not only shows the 3D content we *intend* to draw (i.e. the curve and moving object) but also the *camera* observing it.

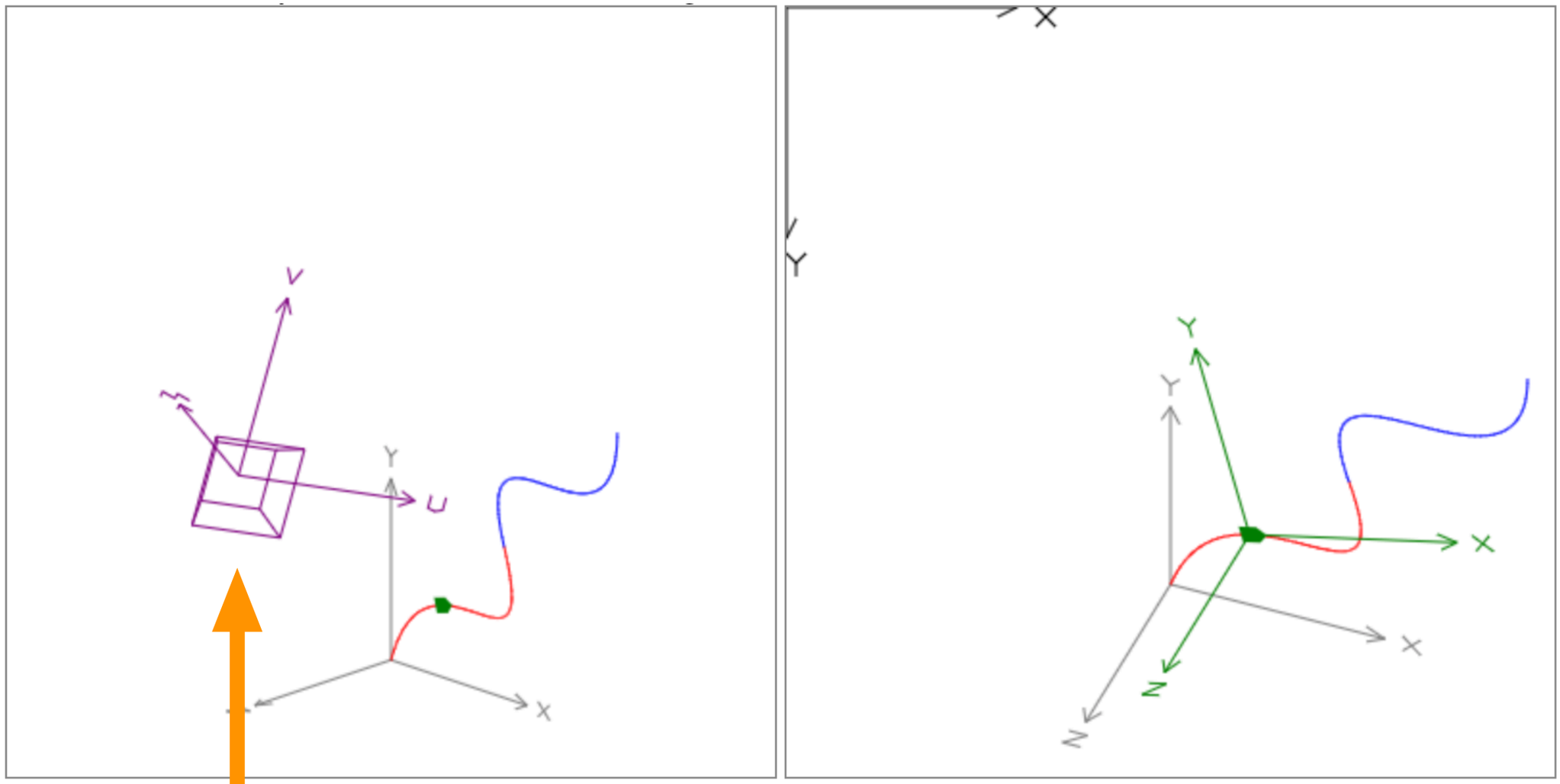
Right window:
A view of the world from the point of view of the *camera* shown on the left. As the camera position moves, the world as-seen from this point of view also gets updated.

The camera

RECAP

jsbin.com/ficoxeh

Week7/Demo4



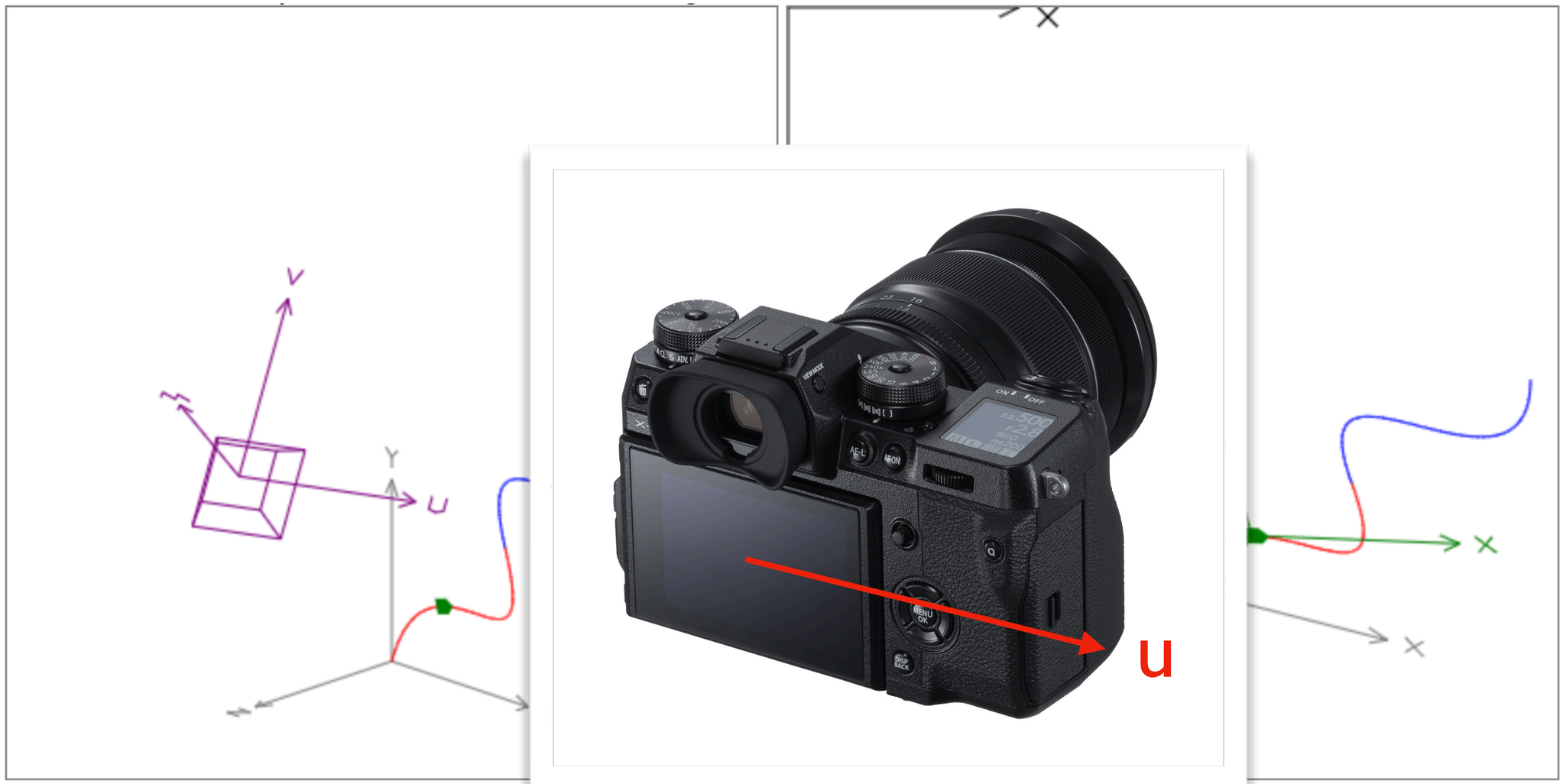
Shown in purple in this illustration (movable along a circle in the demo) The *camera coordinate system* is permanently affixed to it, and its axes are labeled “u”, “v”, and “w”.

The camera

RECAP

jsbin.com/ficoxeh

Week7/Demo4



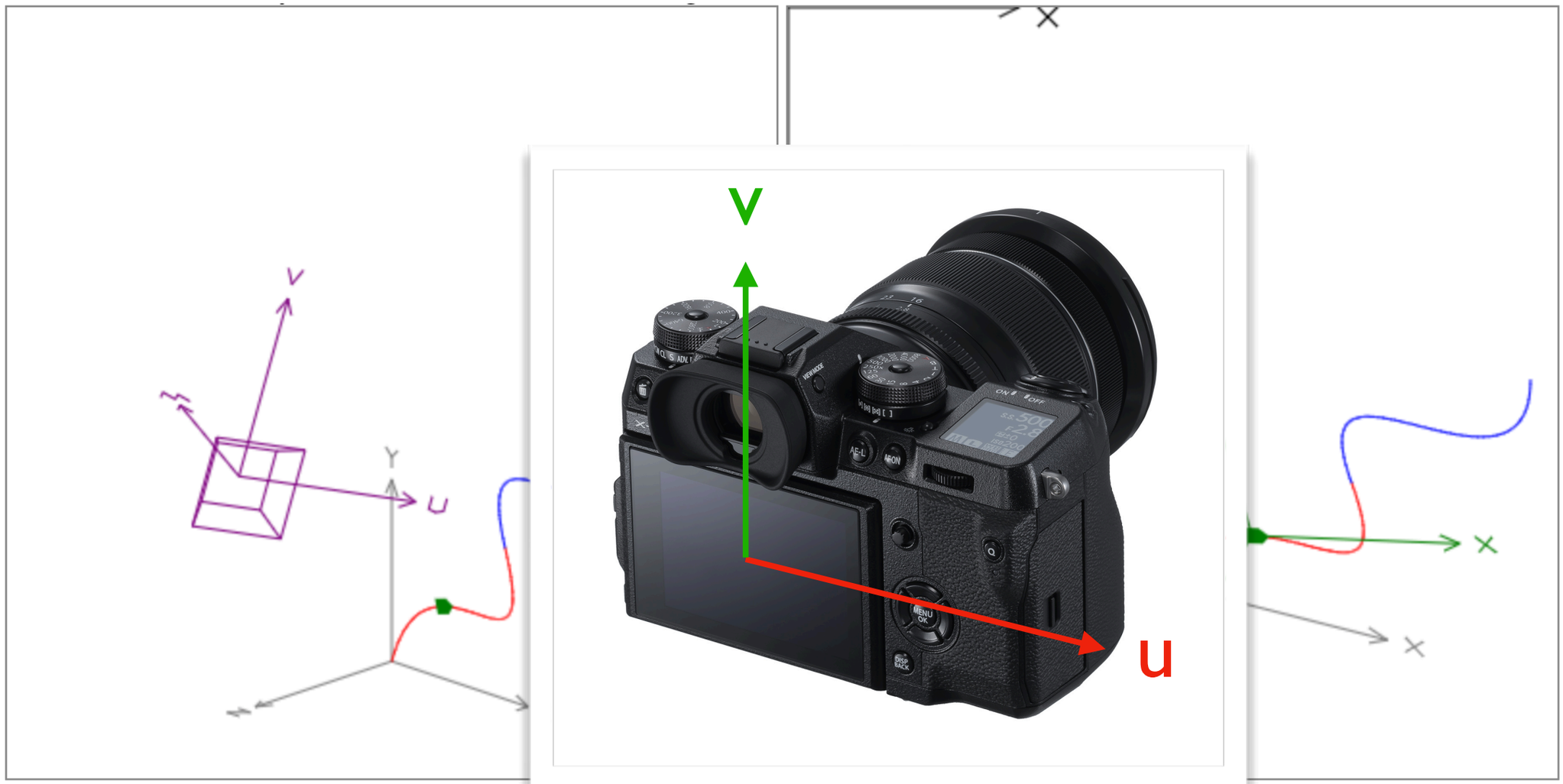
The origin of the coordinate system is at the "center" of the image (say, the center of the sensor where the image is being captured). The "u" axis is on the plane of the sensor, and oriented along the horizontal direction in the camera

The camera

RECAP

jsbin.com/ficoxeh

Week7/Demo4



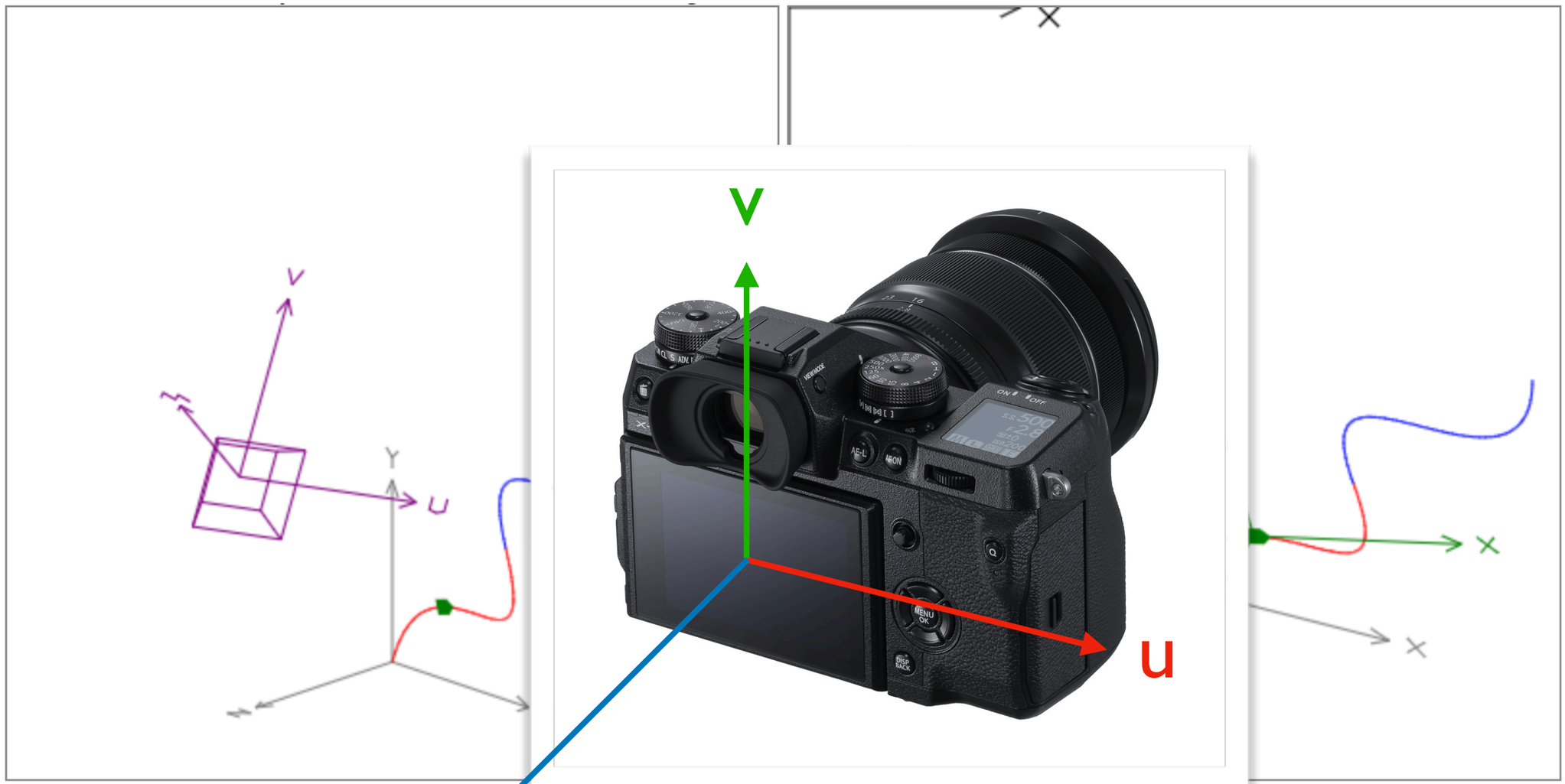
The “v” axis is also on the plane of the sensor, and oriented along the vertical direction in the camera

The camera

RECAP

jsbin.com/ficoxeh

Week7/Demo4



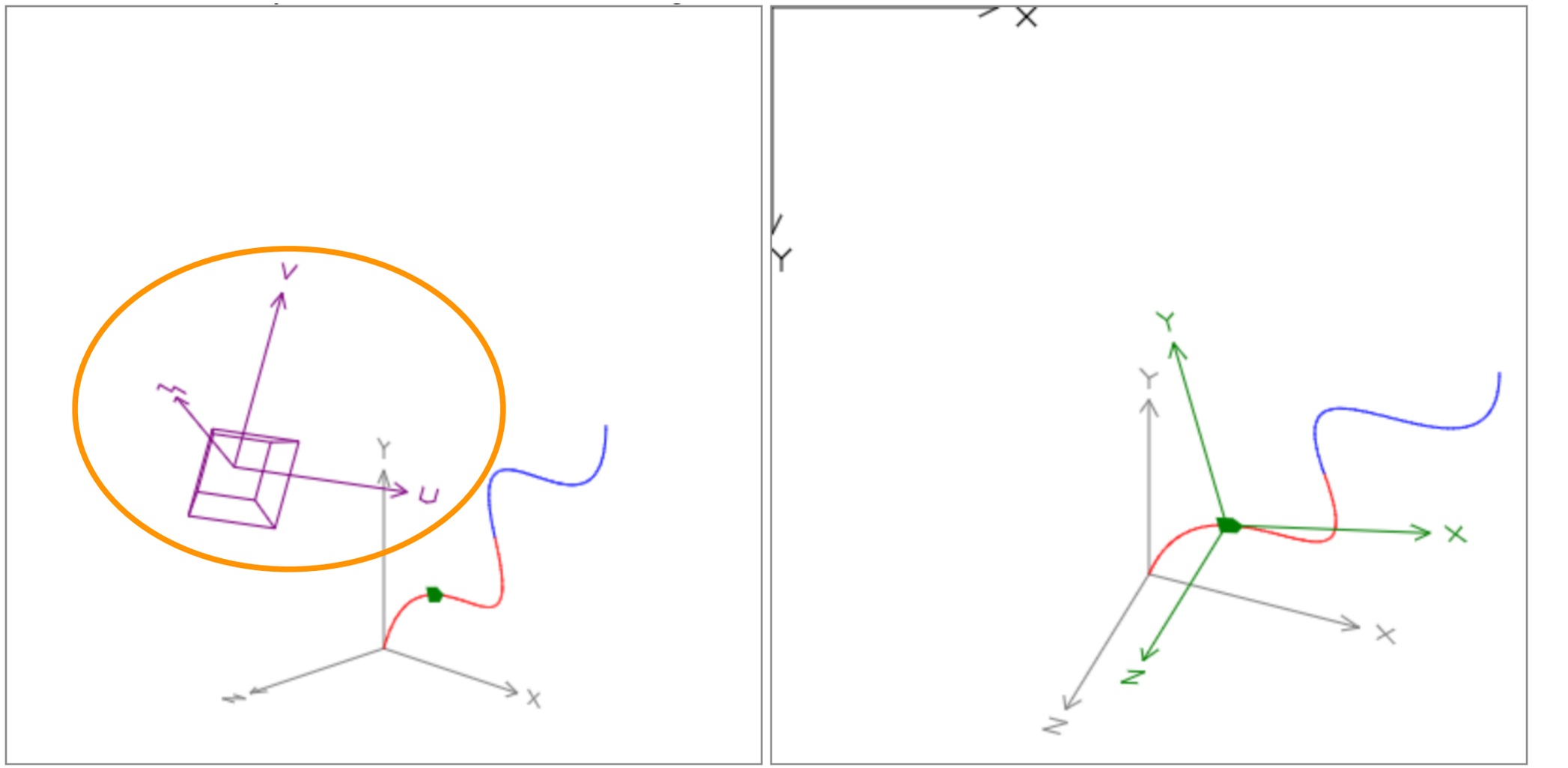
The “w” axis is perpendicular to the camera sensor, and points *away* from the scene being observed (i.e. towards the person holding the camera)

The camera

RECAP

jsbin.com/ficoxeh

Week7/Demo4

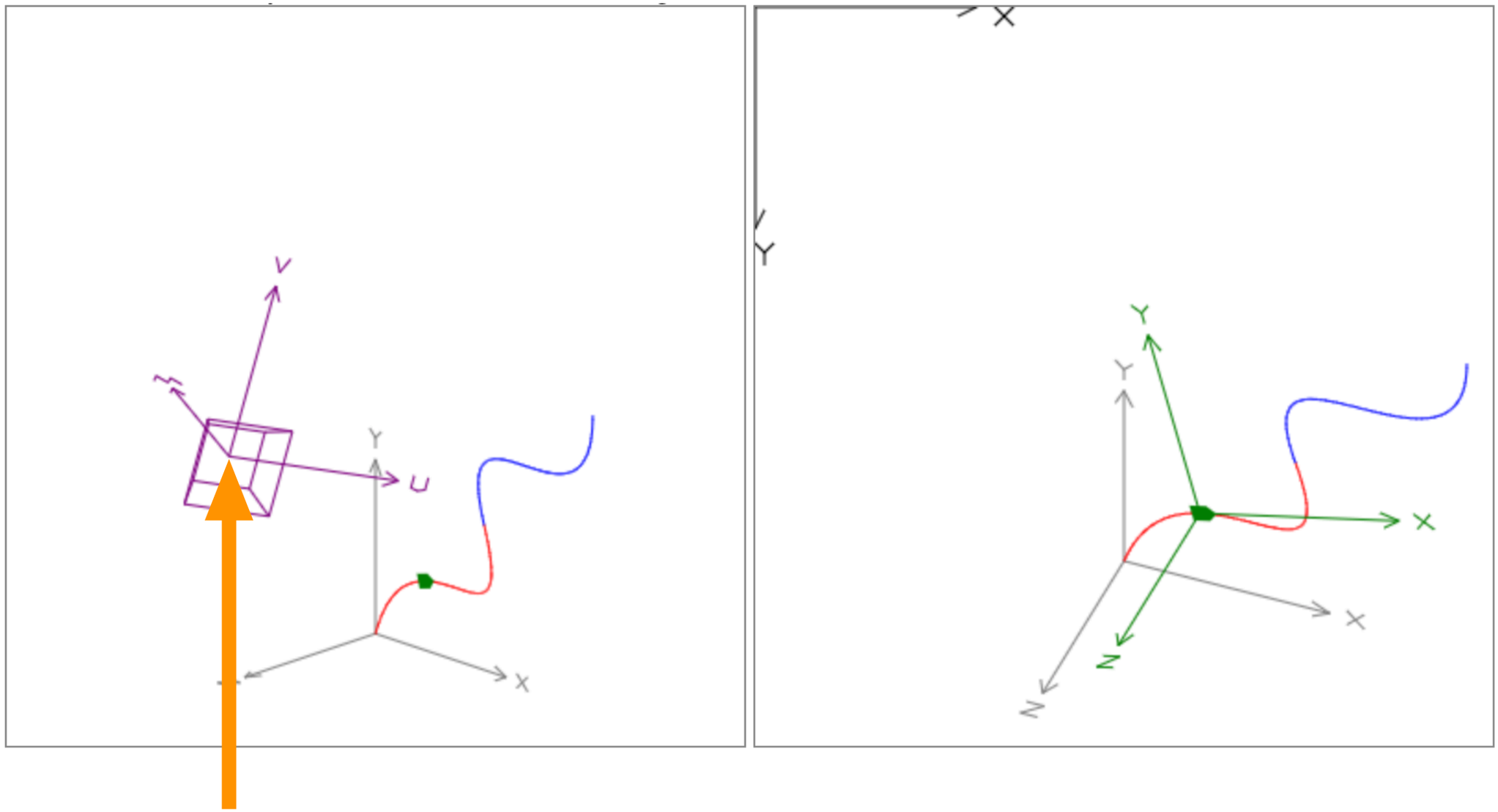


In our code demo, the camera is depicted as a cropped pyramid (with the short face being the "sensor/back side" of the camera, and the large face being the "lens/front side")

The camera

jsbin.com/ficoxeh

Week7/Demo4



The coordinate system (“uvw”) affixed to the camera as shown, is the *camera coordinate system*.

The camera

jsbin.com/ficoxeh

Week7/Demo4

- Why do we care about the *camera coordinate system?*
  - If we have a way of figuring out what are the coordinates of every drawing primitive (control points of curves, vertices of polygons, etc) in the camera coordinate system, we can easily get *some* version of a 2D visualization of this scene, by simply "dropping" the w-coordinate (and using u- and v- coordinates as pixel locations).
- Examples :

jsbin.com/xequtul

Week7/Demo1

jsbin.com/xudufet

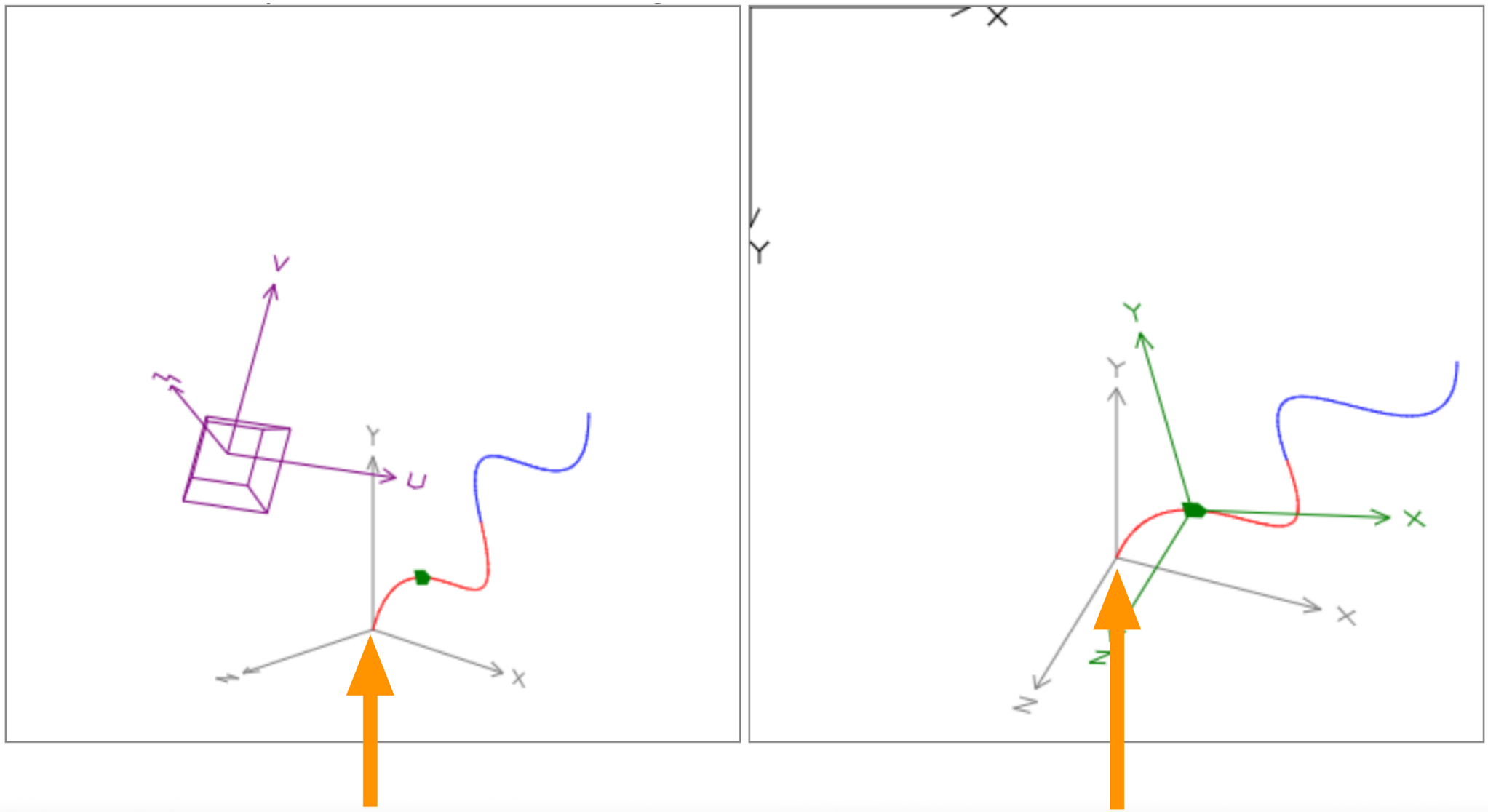
Week7/Demo2

- (We'll get much more sophisticated about this process; "droping the w-coordinate" is just an oversimplified example)

“World” coordinates

jsbin.com/ficoxeh

Week7/Demo4

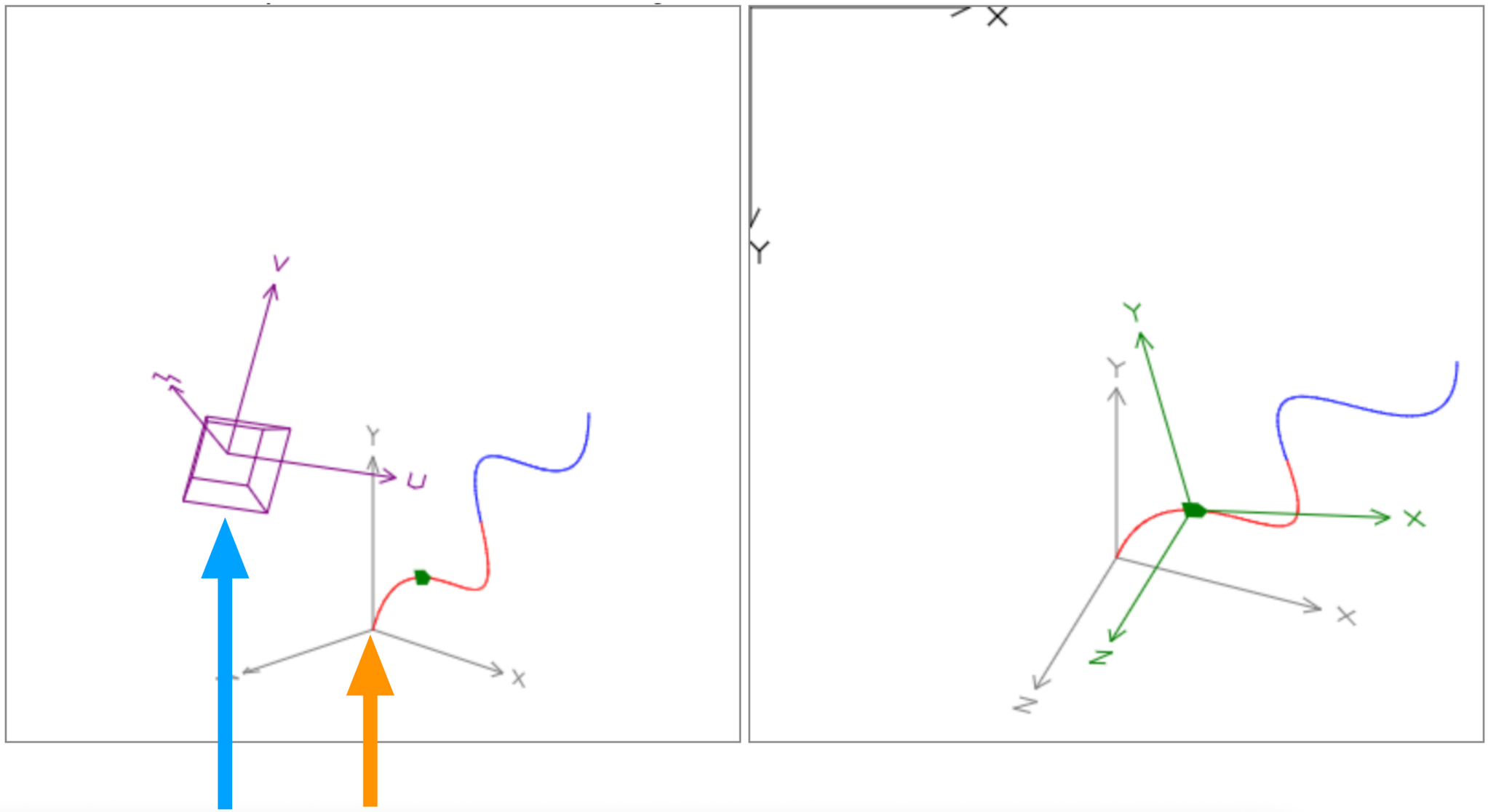


Illustrated in this demo in grey color, the *world coordinate system* is a system that we arbitrarily choose, in such a way that describing locations/vectors/coordinates of things we want to draw becomes convenient.
(For example, the locations of control points for the 2 piecewise-cubics are given in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

“World” coordinates

jsbin.com/ficoxeh

Week7/Demo4

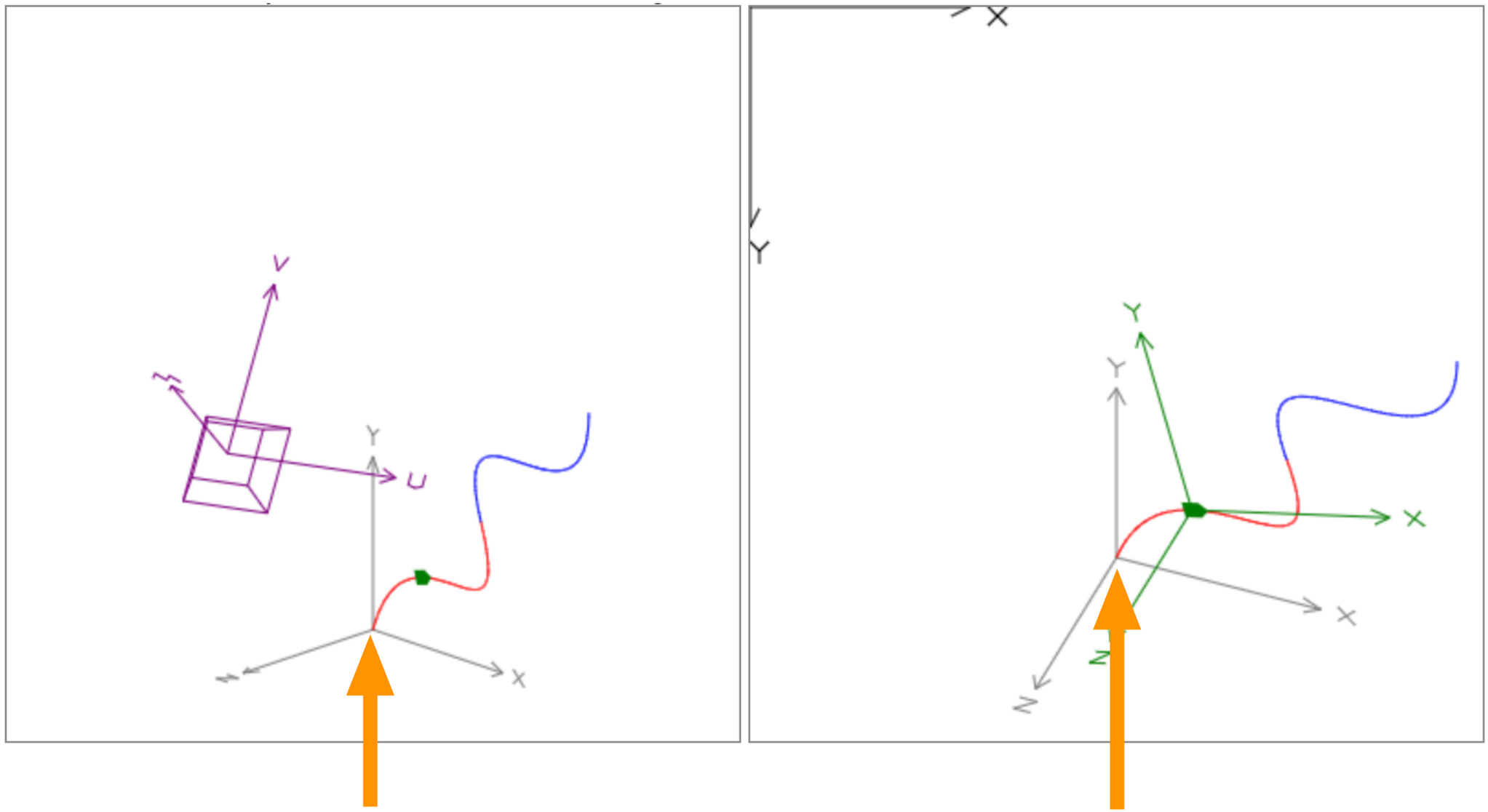


For purposes of “what do we see on the screen”, it doesn’t quite matter where the coordinate system is placed (it’s an arbitrary choice). What matters is “how is the camera positioned” relative to that world coordinate system …

“World” coordinates

jsbin.com/ficoxeh

Week7/Demo4

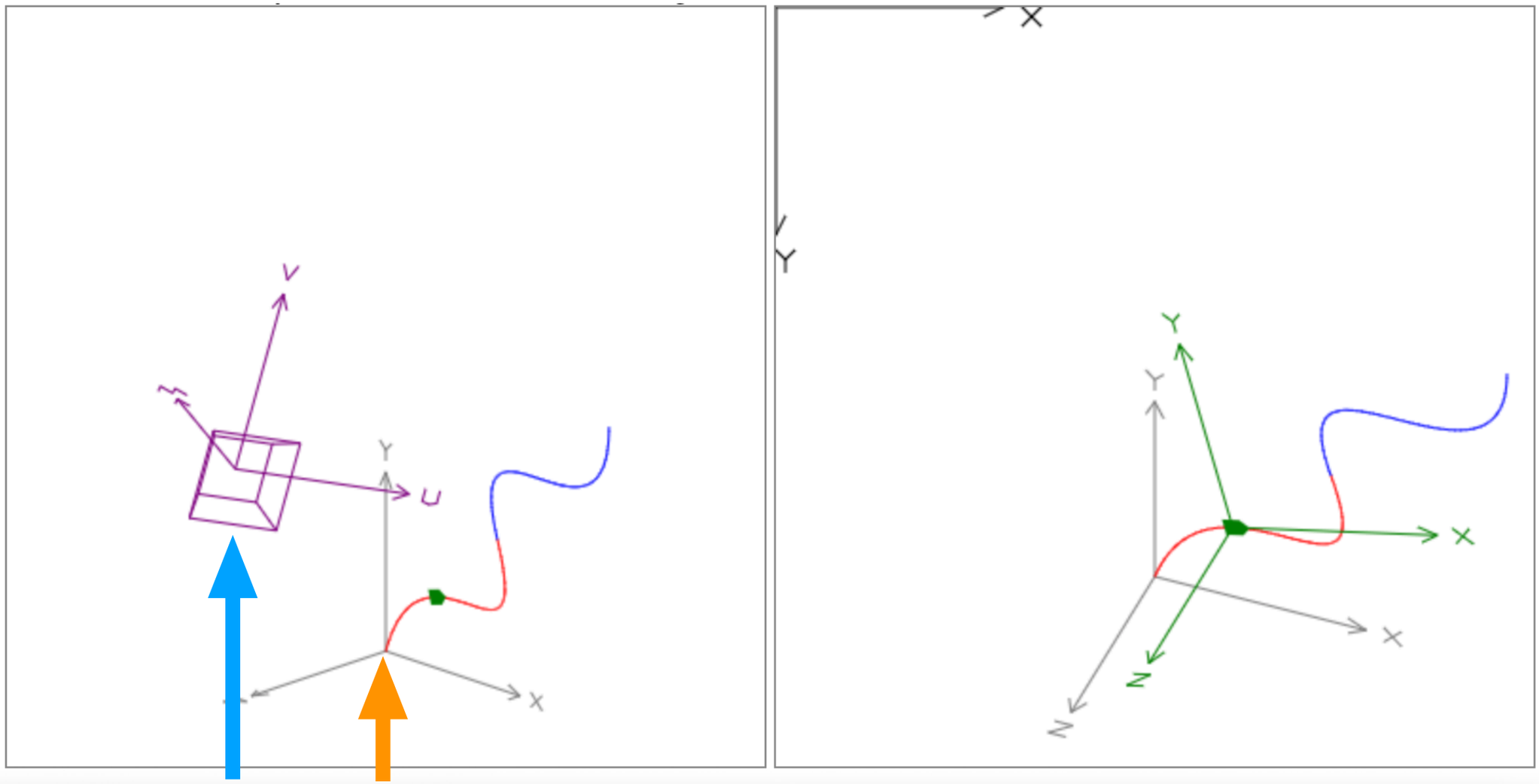


Note that this same coordinate system (grey axes) shows up differently in our two windows … this is because the relative placement of the point of observation (on the right, being the “purple” camera; on the left a “faraway” observer) is different in the 2 cases.

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*.

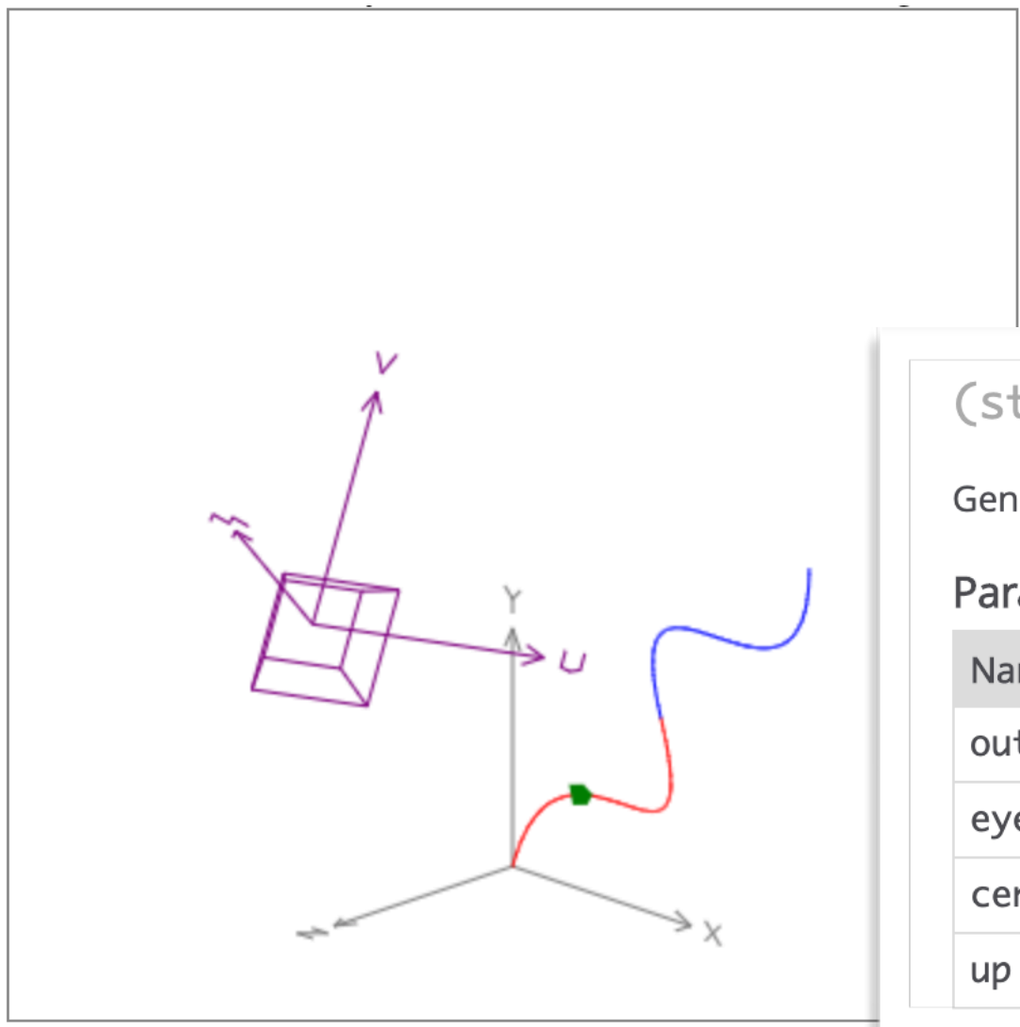
This is called the ***lookAt transform***.

(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:

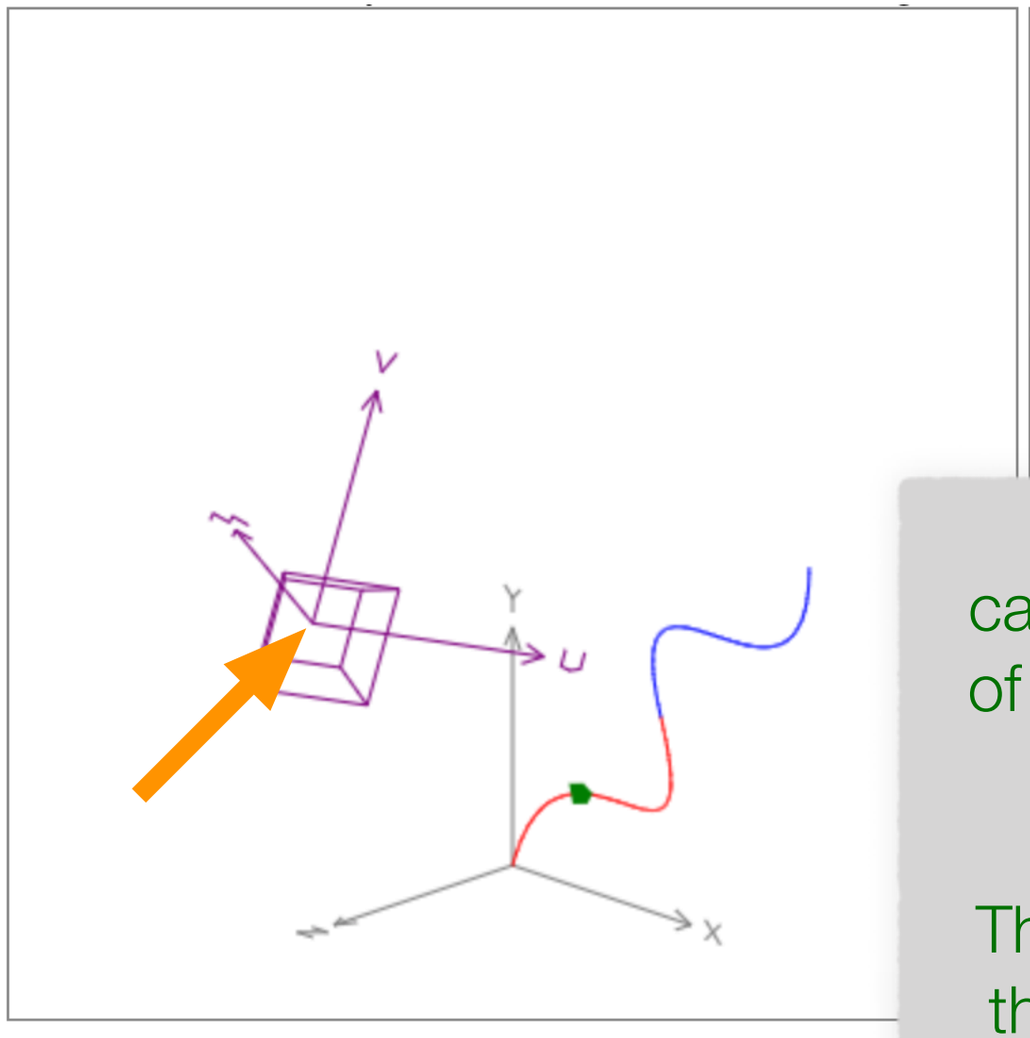
Name	Type	Description
out	mat4	mat4 frustum matrix will be written into
eye	ReadonlyVec3	Position of the viewer
center	ReadonlyVec3	Point the viewer is looking at
up	ReadonlyVec3	vec3 pointing up

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

The lookAt transform

jsbin.com/ficoxeh

Week7/Demo4



The *eye* location is the origin of the camera coordinate system (i.e. the center of the camera sensor), described in world coordinates.

This means, for example, that if we apply the lookAt transform to the *eye* location, then the result will be (0,0,0)!!

In the demo, the way we control the camera position is by modifying the *eye* location!

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

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

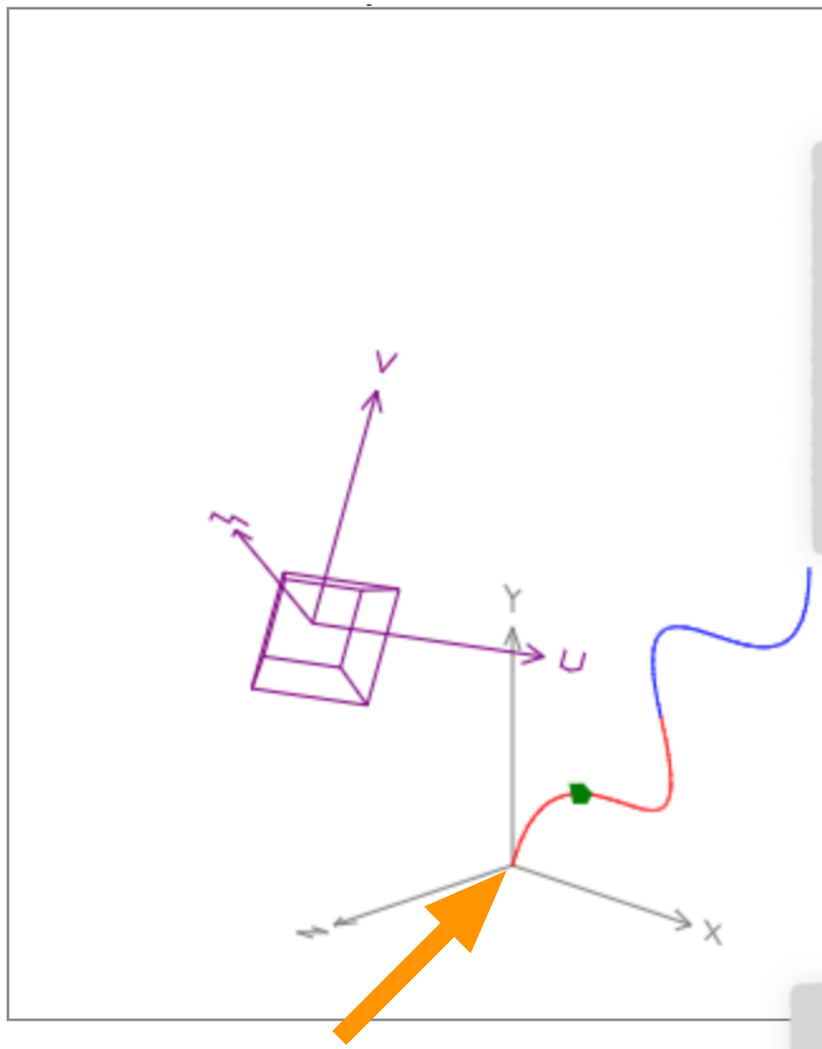
Parameters:

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

The lookAt transform

jsbin.com/ficoxeh

Week7/Demo4



The *target* location ("center" in the glMatrix documentation) is a point in world space that the camera is pointed towards. *This point, when converted to camera coordinates will lie along the (negative part) of the w-axis.*

In the demo, we are pointing the camera to the origin of the world coordinate system

By adjusting the *target*, we effectively tilt the camera towards a specific location in space

(Note that the "faraway observer" actually targets a point just above the WCS origin, along the y-axis!)

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

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

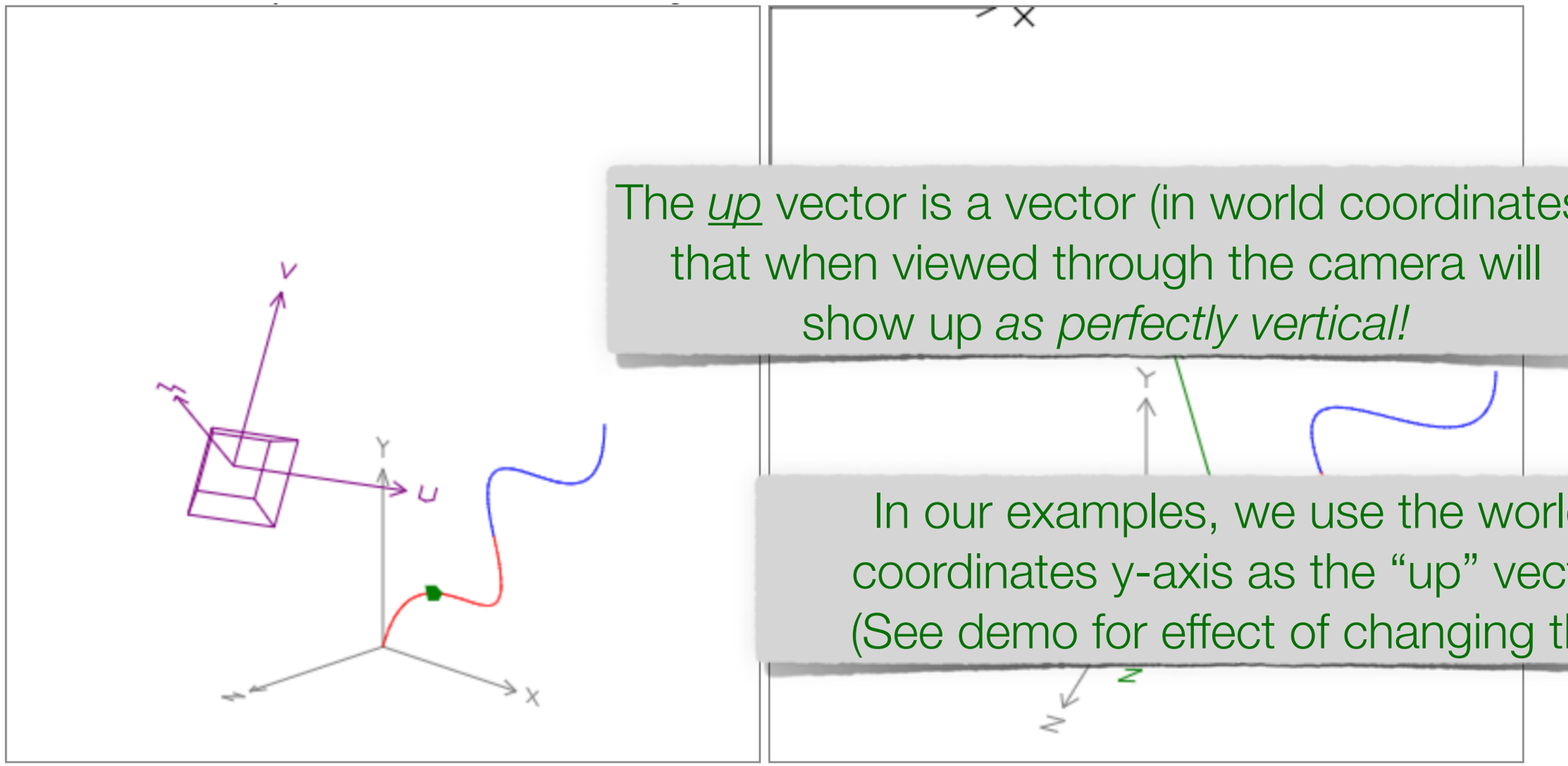
Parameters:

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

The lookAt transform

jsbin.com/ficoxeh

Week7/Demo4



The *up* vector is a vector (in world coordinates) that when viewed through the camera will show up *as perfectly vertical!*

In our examples, we use the world-coordinates y-axis as the “up” vector. (See demo for effect of changing this)

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

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

Parameters:

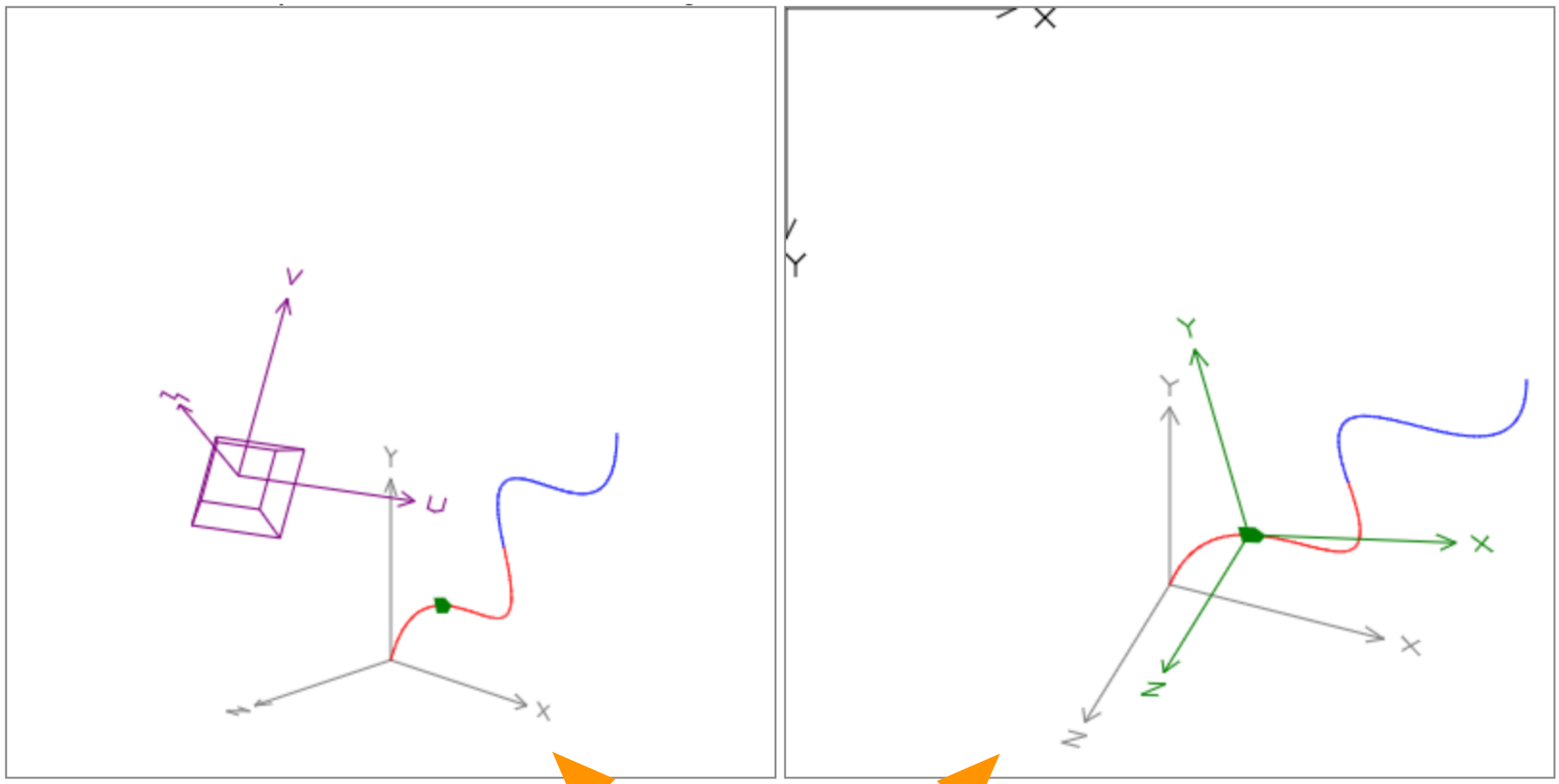
Name	Type	Description
out	mat4	mat4 frustum matrix will be written into
eye	ReadonlyVec3	Position of the viewer
center	ReadonlyVec3	Point the viewer is looking at
up	ReadonlyVec3	vec3 pointing up

Common misconception: The up vector is not the same as the v-axis of the camera system! (they reside on the same plane, but they don’t have to be identical!)

The lookAt transform

jsbin.com/ficoxeh

Week7/Demo4



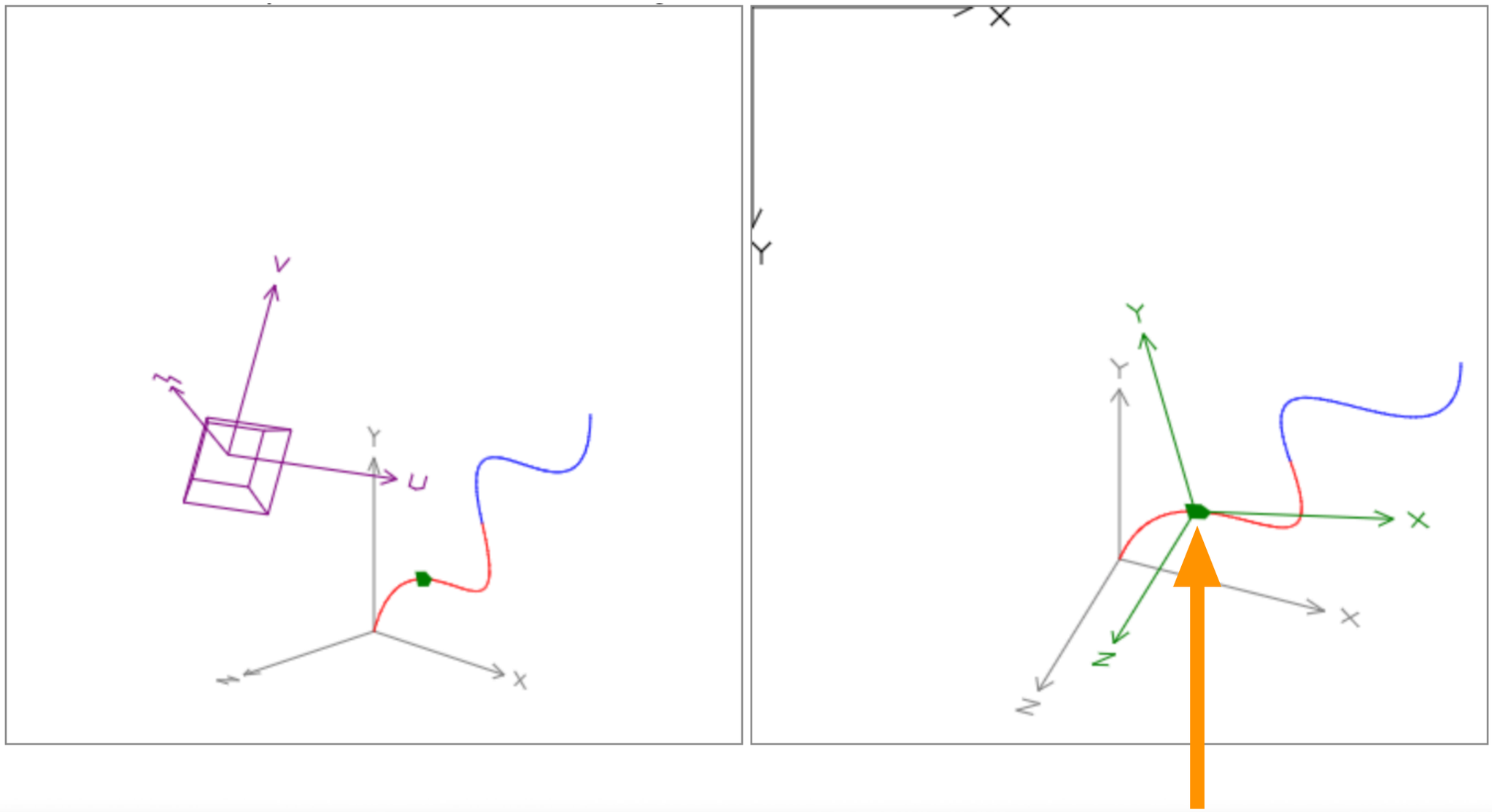
(A side observation …) these two visualizations have been created with a *different lookAt* transform! (the one of the camera used on the right; on the left, the location of the external observer was used to place a fictitious camera there!)

(Question: is the lookAt of the “purple” camera used on the left, and how?)

Model(ing) transform(s)

jsbin.com/ficoxeh

Week7/Demo4



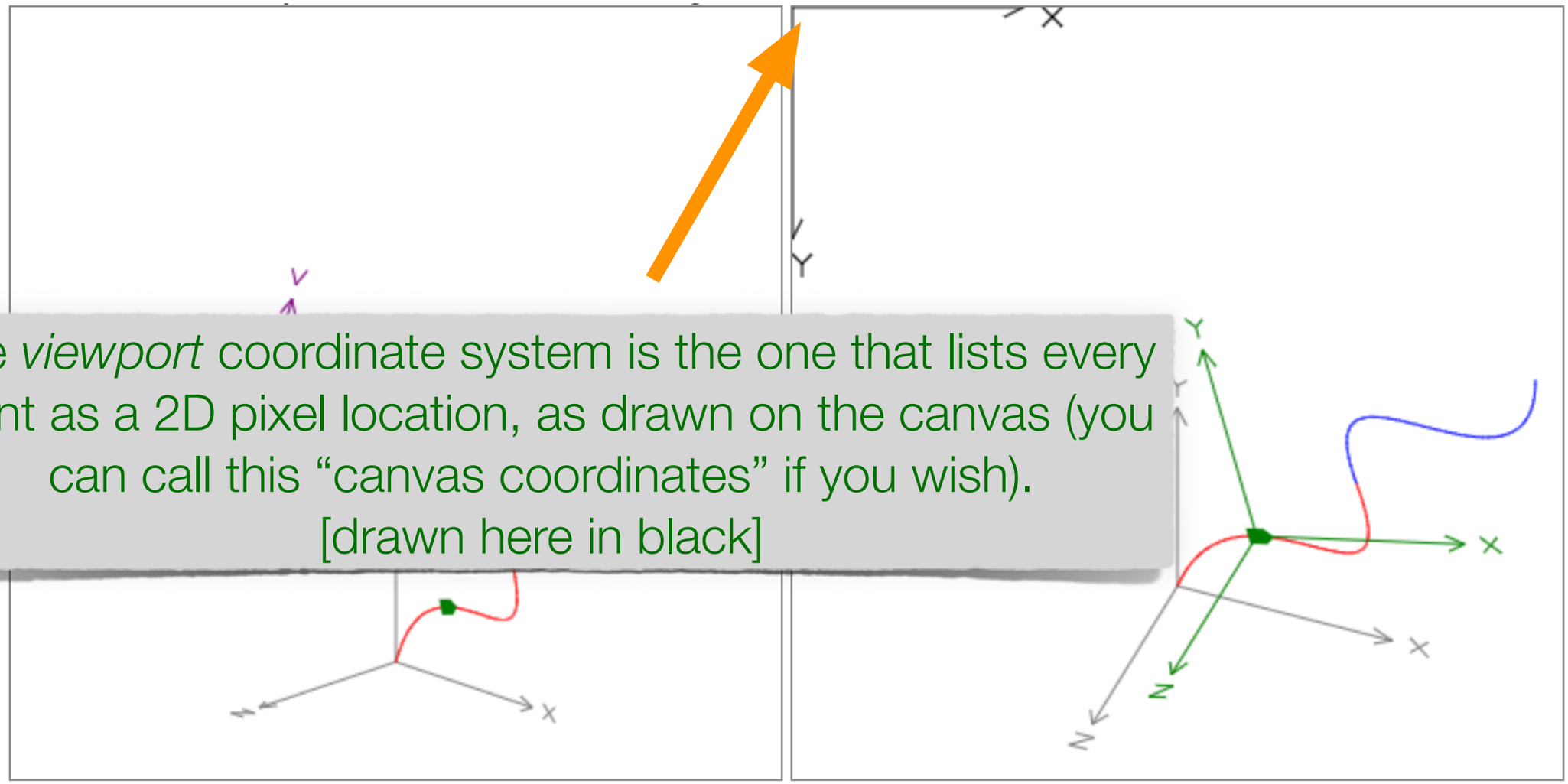
We can also have additional transforms that define coordinate systems (typically for moving objects, or displaced object instances) relative to the world coordinates. (Shown here in green …)

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

Canvas/Viewport coordinates

jsbin.com/ficoxeh

Week7/Demo4



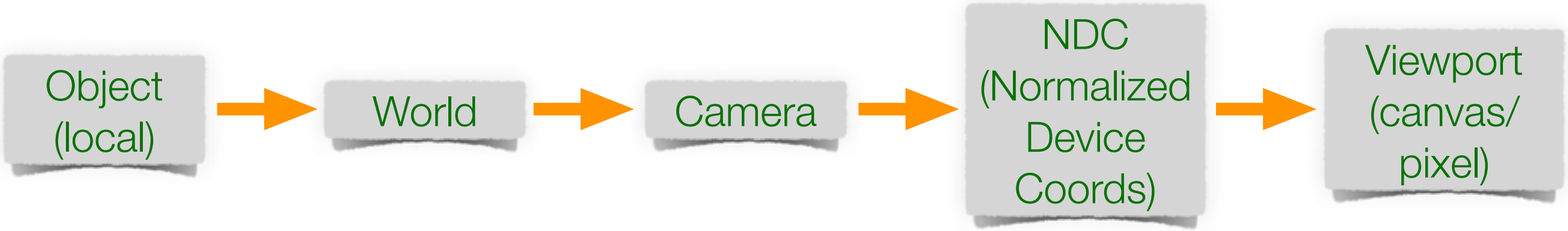
The *viewport* coordinate system is the one that lists every point as a 2D pixel location, as drawn on the canvas (you can call this "canvas coordinates" if you wish).
[drawn here in black]

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

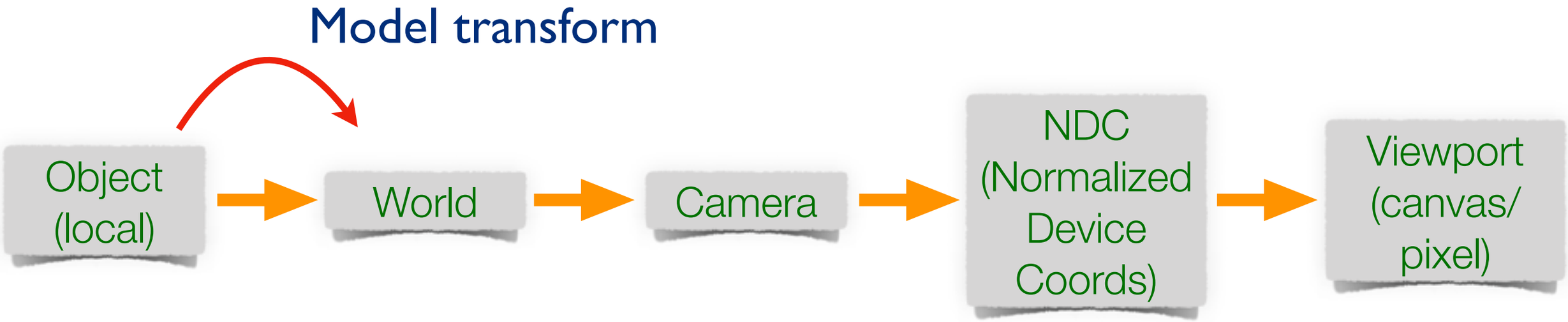
Week7/Demo4



A (viewing) transform pipeline

jsbin.com/ficoxeh

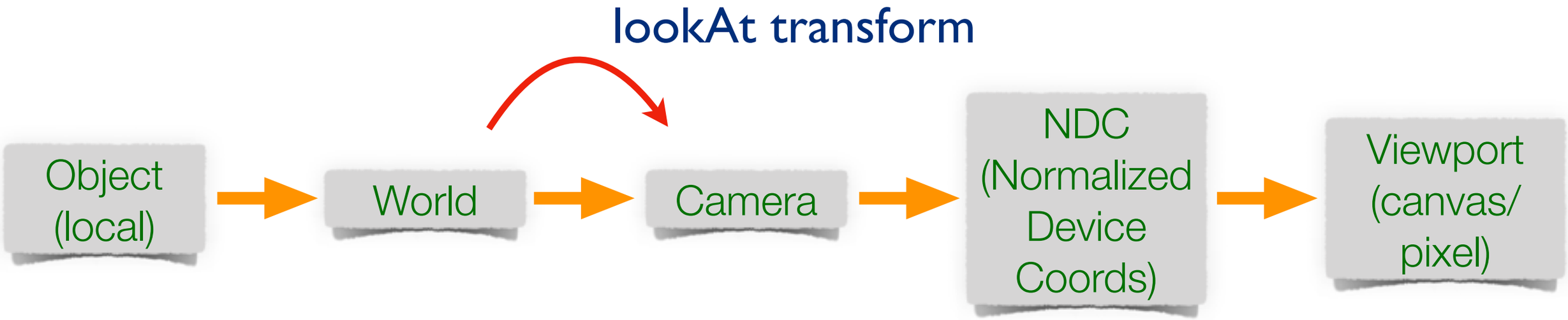
Week7/Demo4



A (viewing) transform pipeline

jsbin.com/ficoxeh

Week7/Demo4

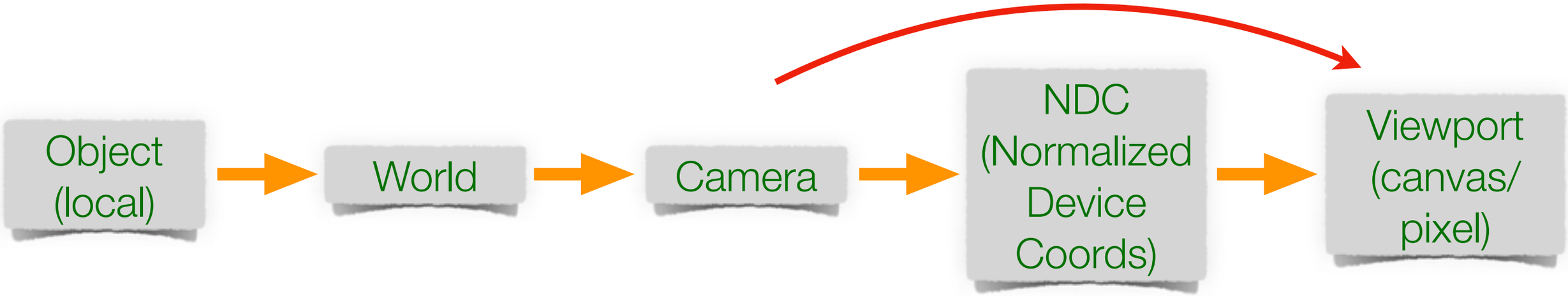


A (viewing) transform pipeline

jsbin.com/ficoxeh

Week7/Demo4

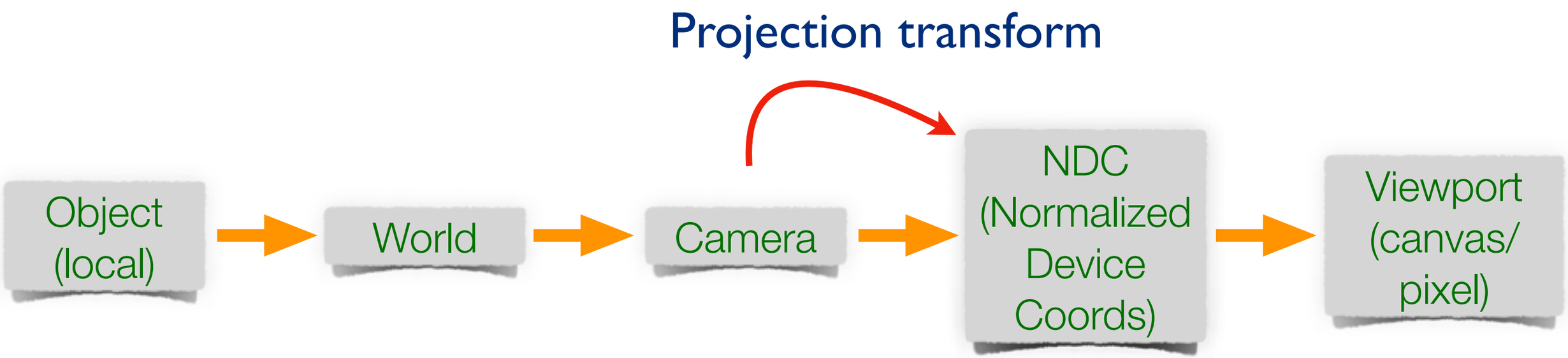
Just "throw away the z-value" ???



A (viewing) transform pipeline

jsbin.com/ficoxeh

Week7/Demo4

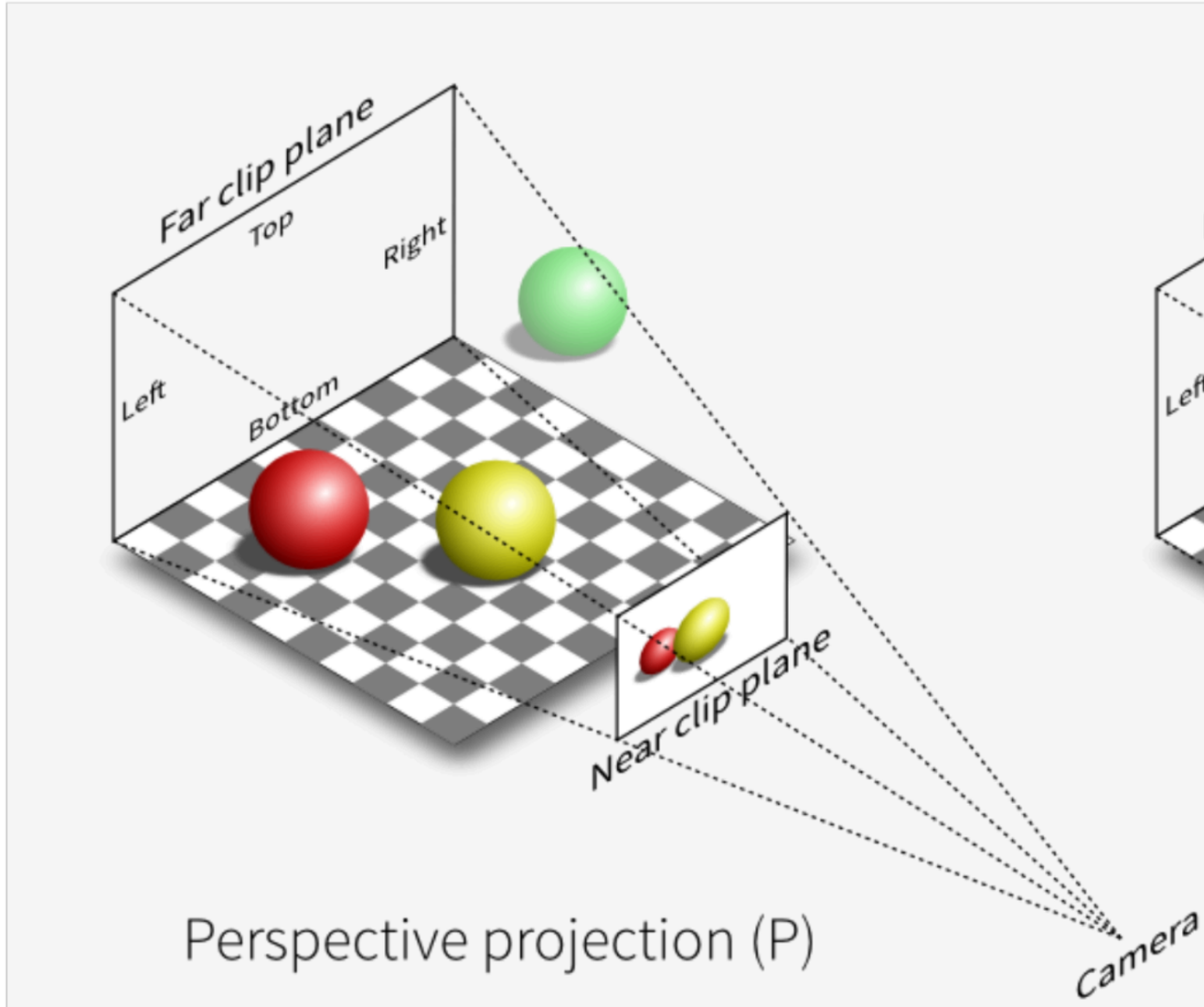


Semantics:
Project/transform the viewing frustum into the cube $[-1,1]^3$

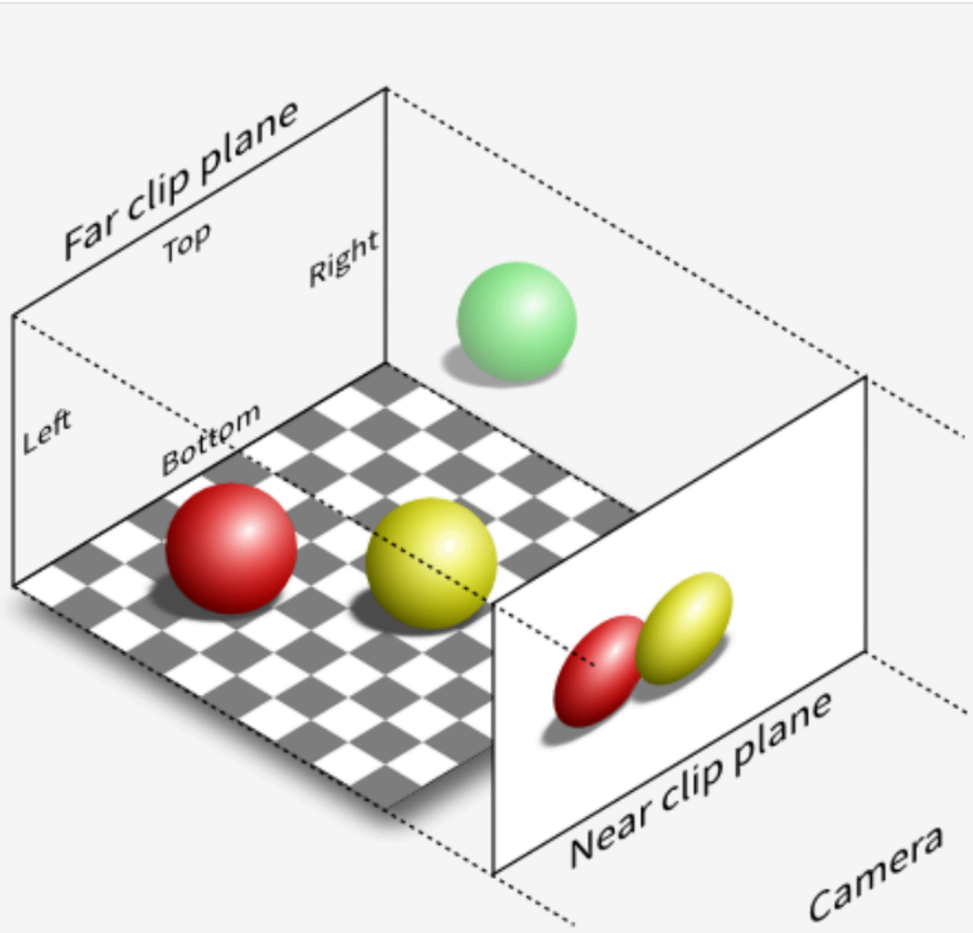
Orthographic/Perspective Projections

jsbin.com/ficoxeh

Week7/Demo4



Perspective projection (P)

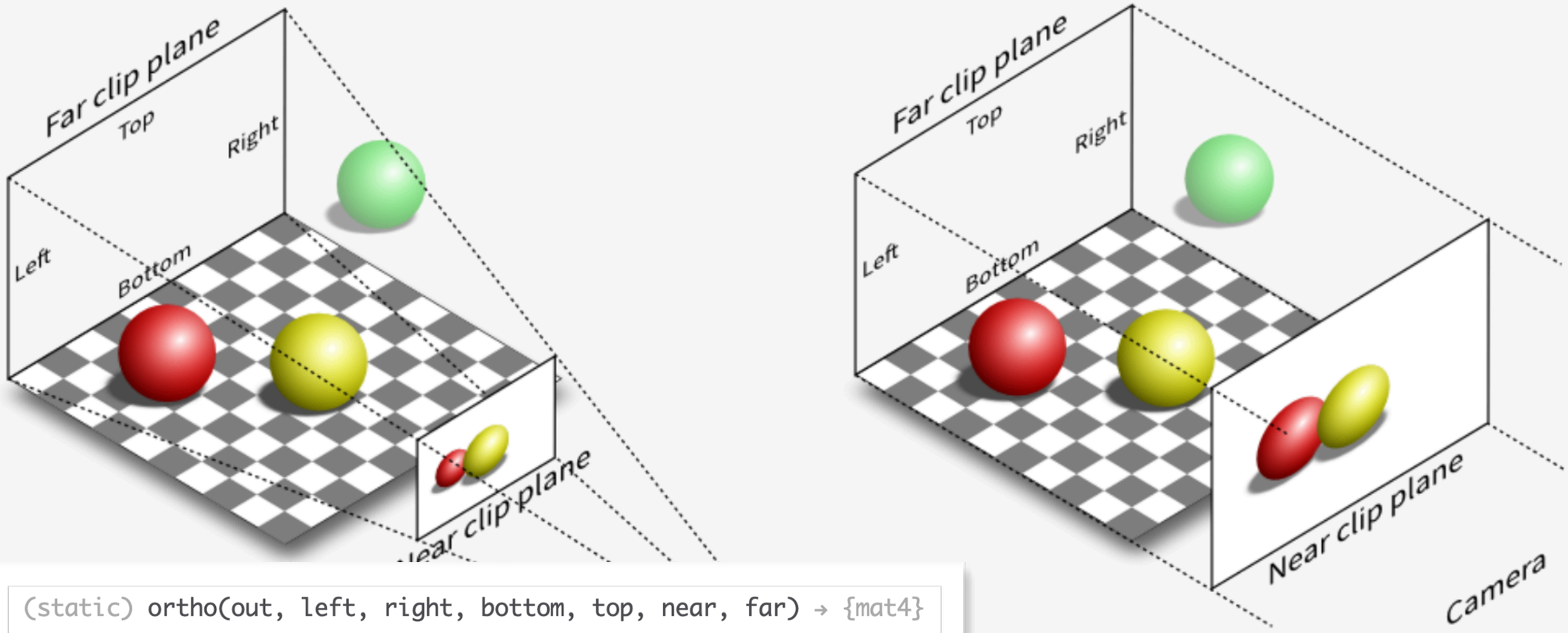


Orthographic projection (O)

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

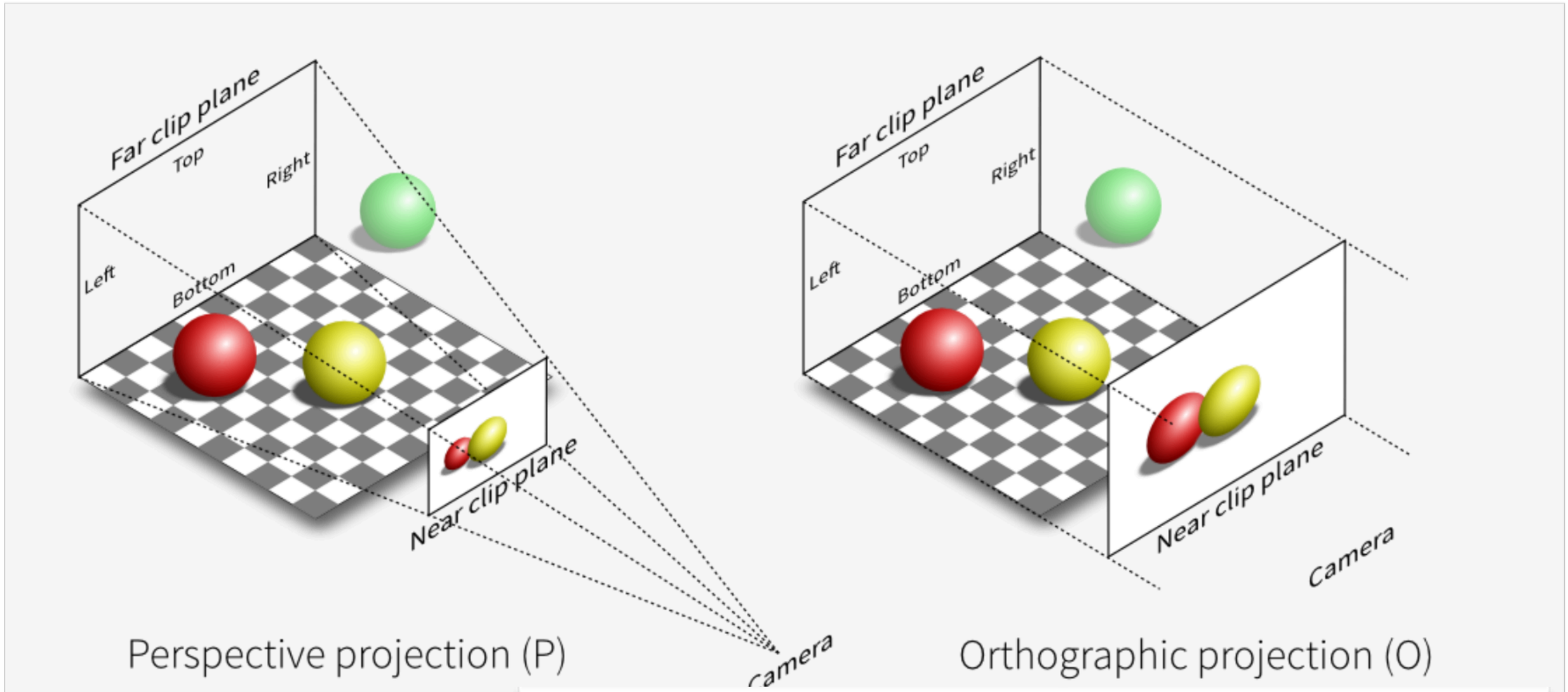
Parameters:

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.

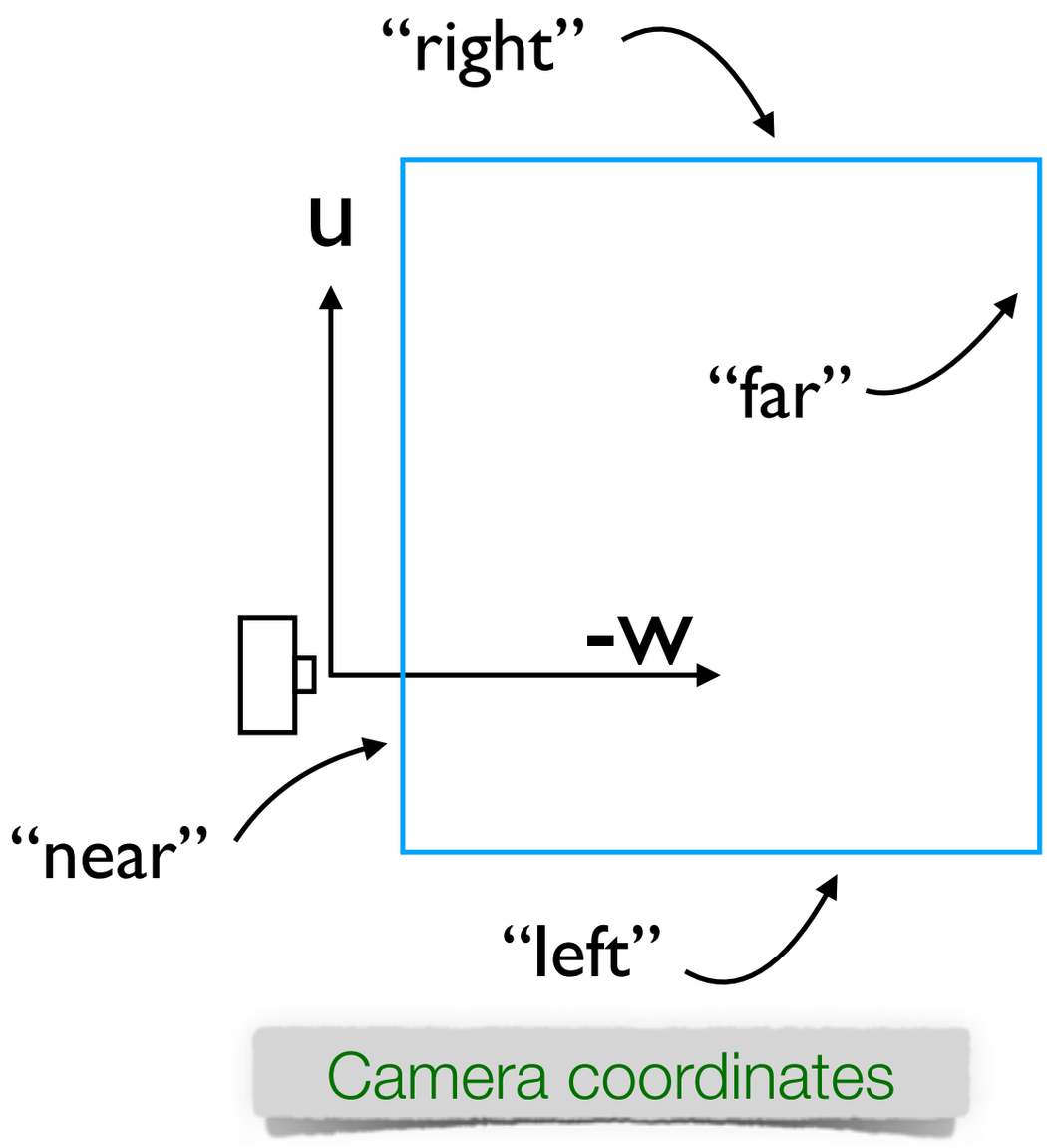
Parameters:

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

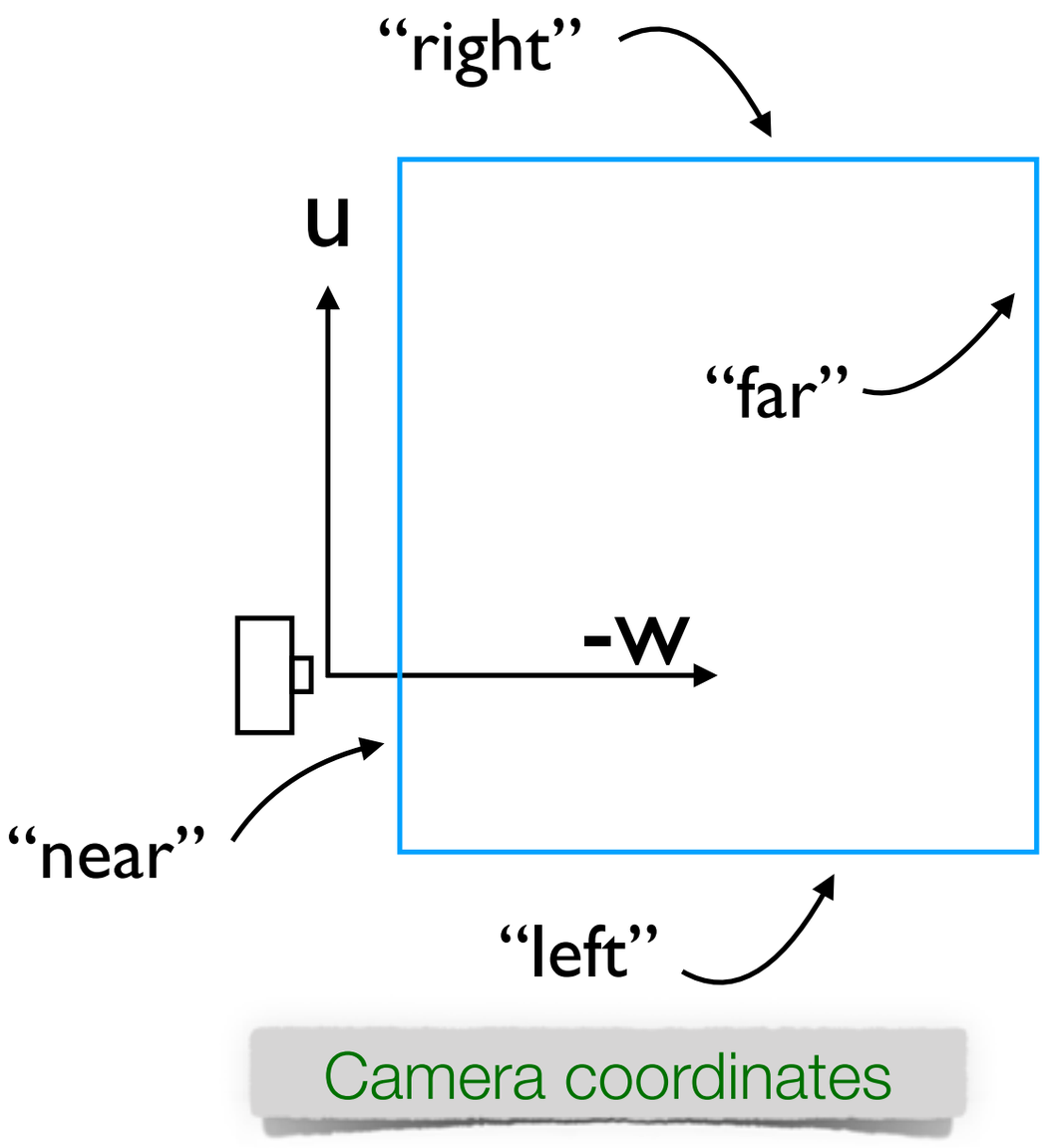


Camera coordinates

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

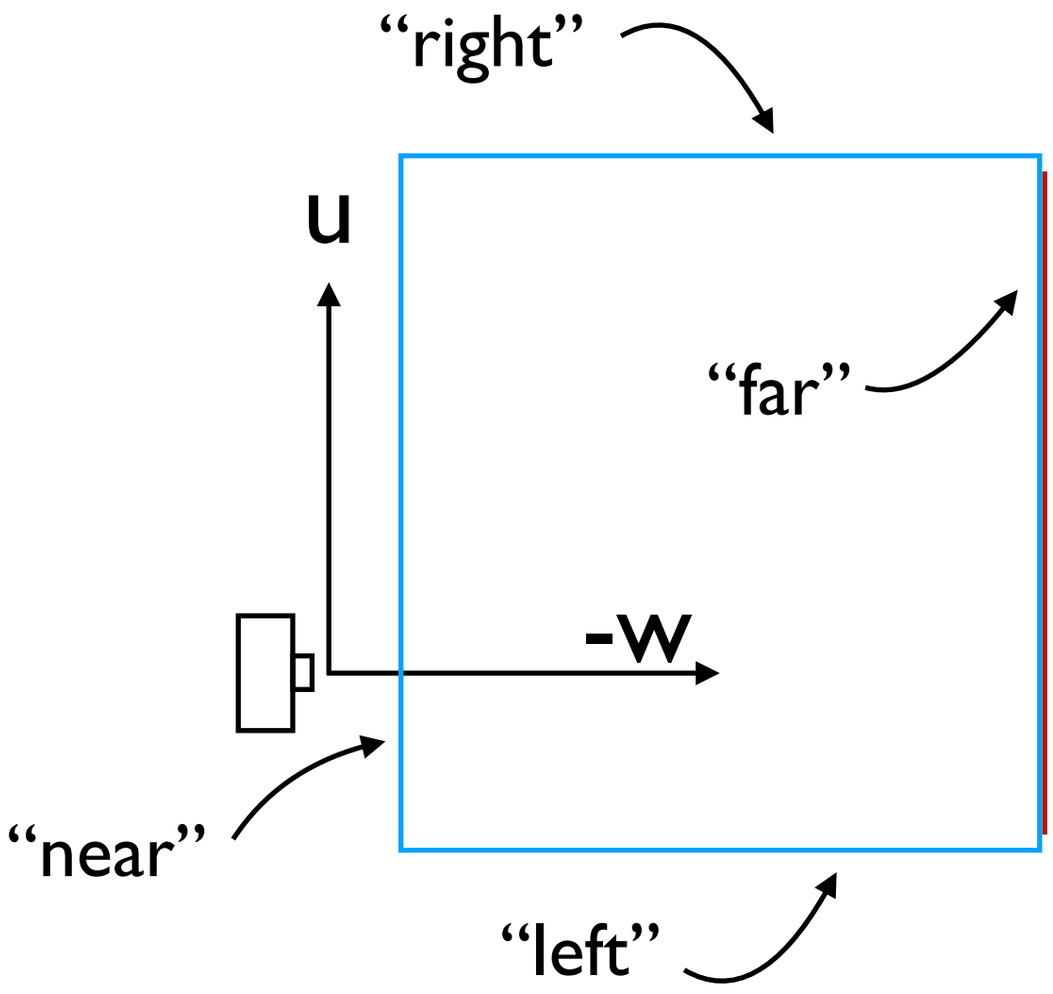
Parameters:

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

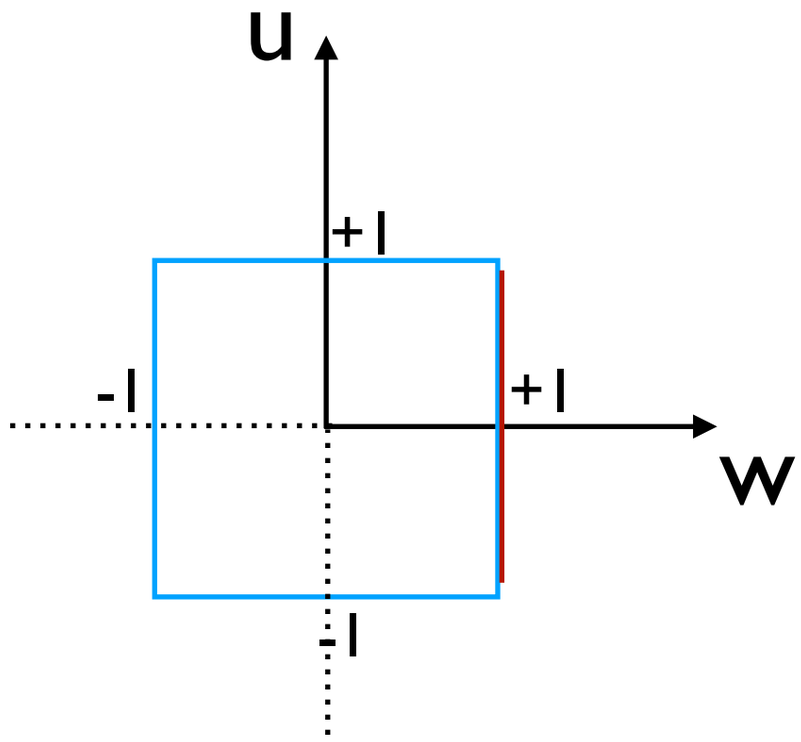
Orthographic Projection

jsbin.com/ficoxeh

Week7/Demo4



Camera coordinates

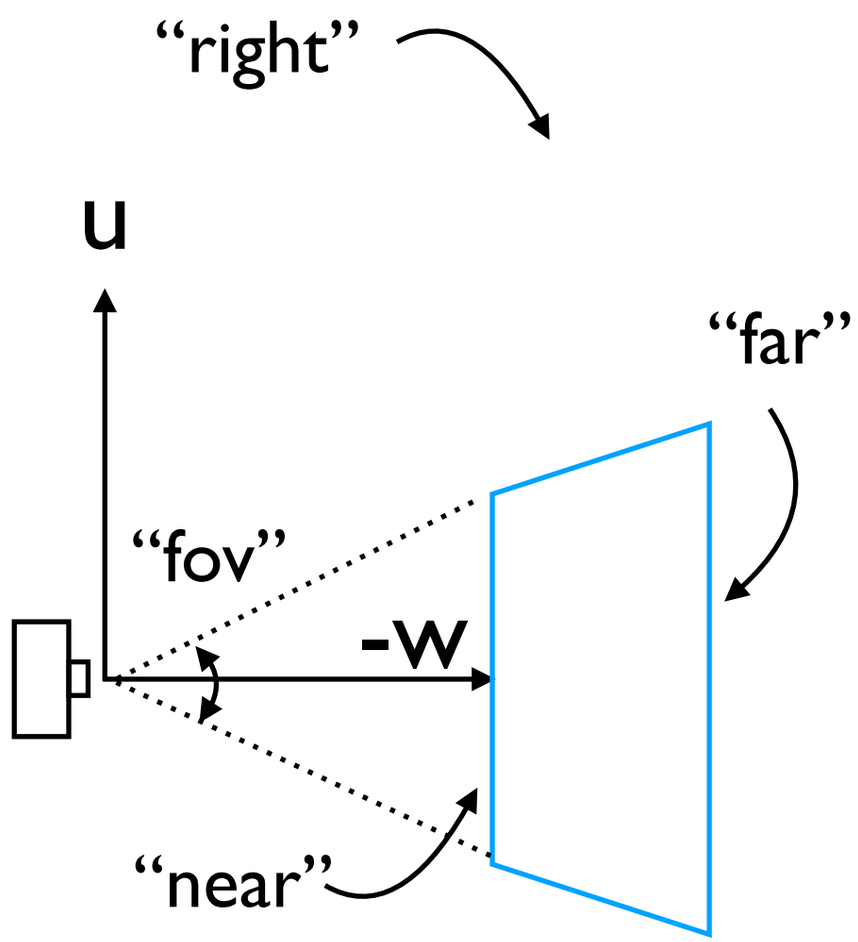


Normalized Device coordinates

Perspective Projection

jsbin.com/ficoxeh

Week7/Demo4

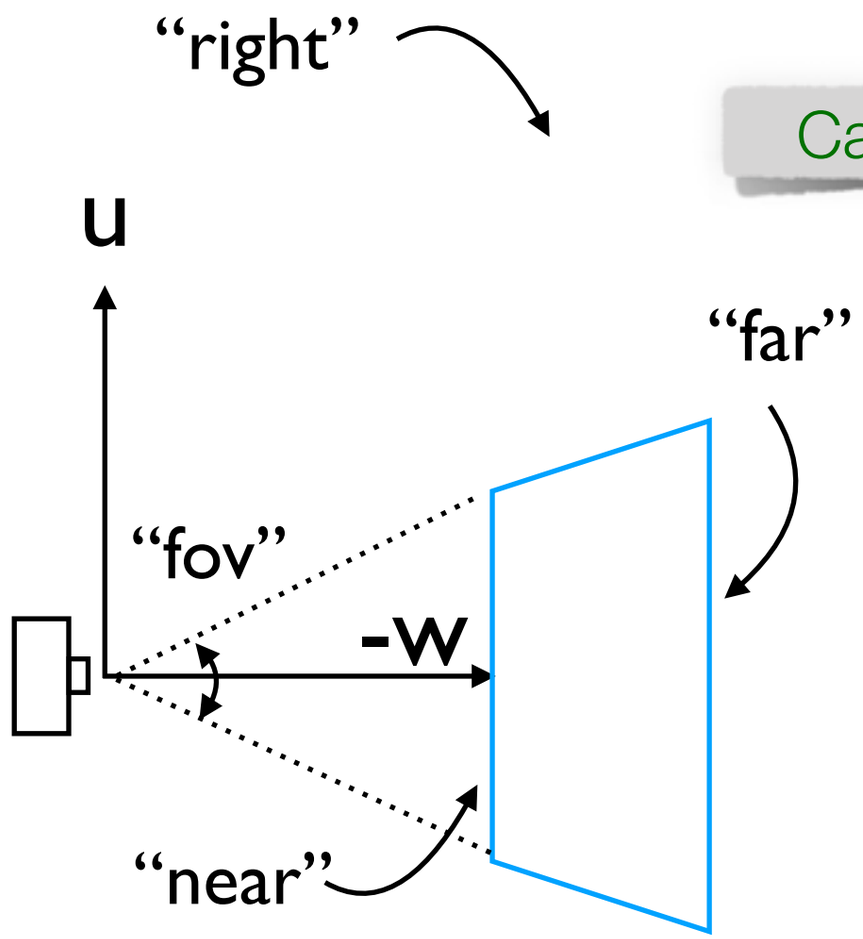


Camera coordinates

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.

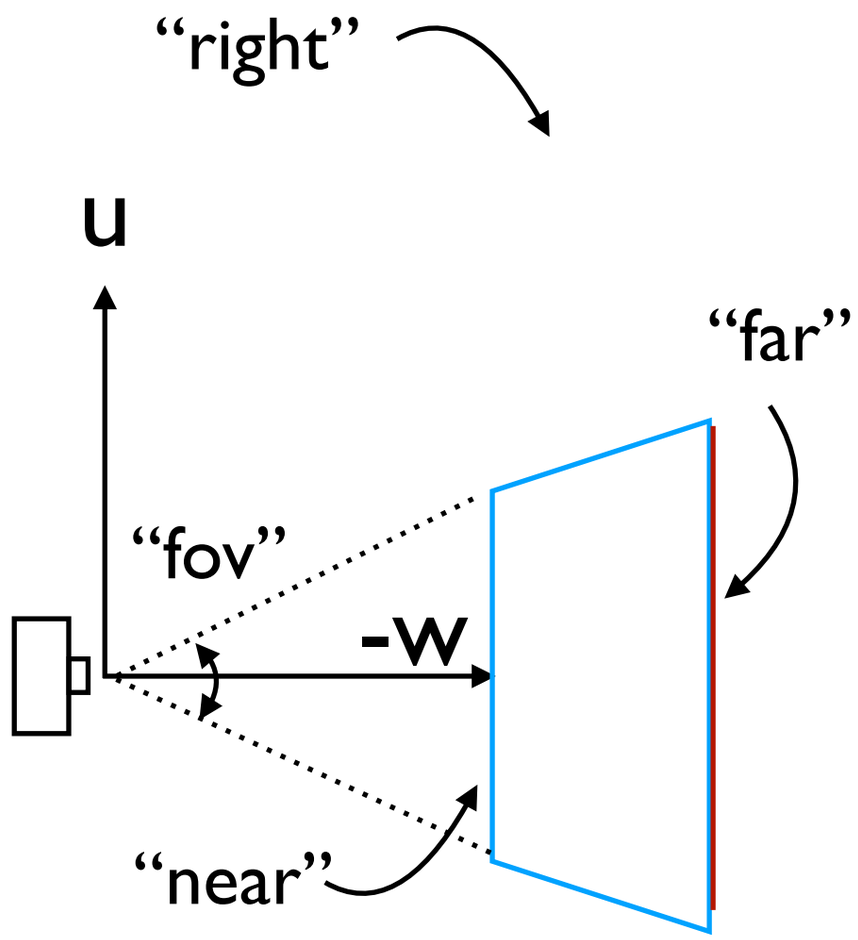
Parameters:

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

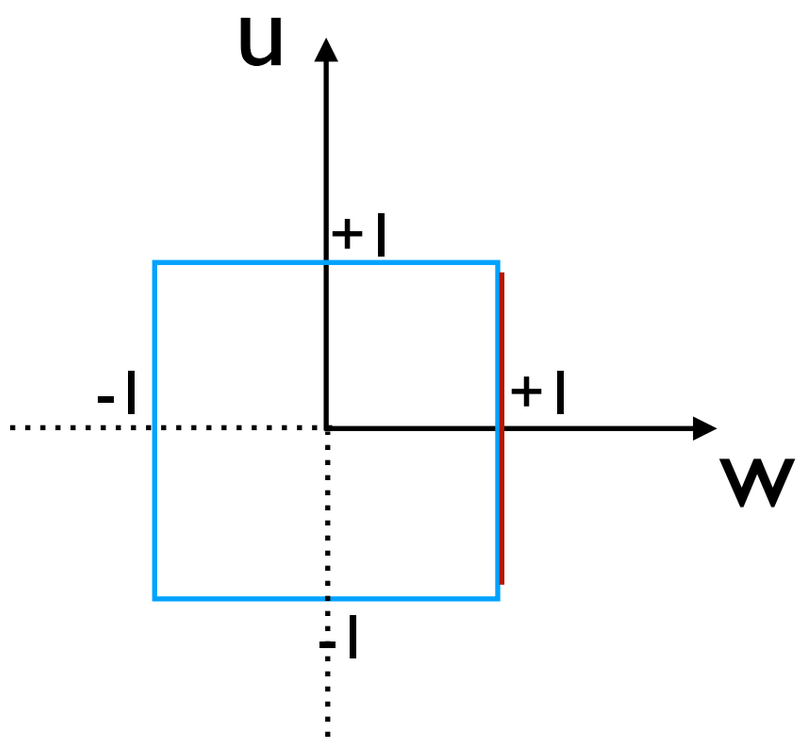
Perspective Projection

jsbin.com/ficoxeh

Week7/Demo4



Camera coordinates



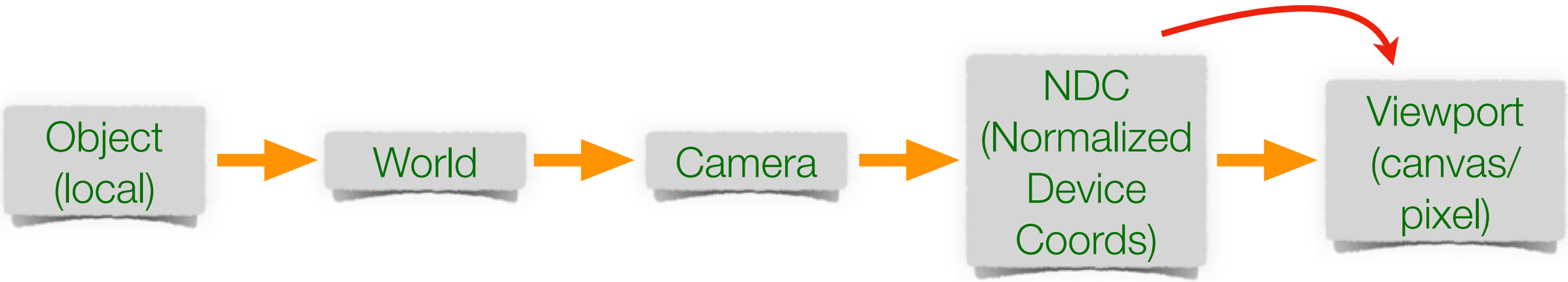
Normalized Device coordinates

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

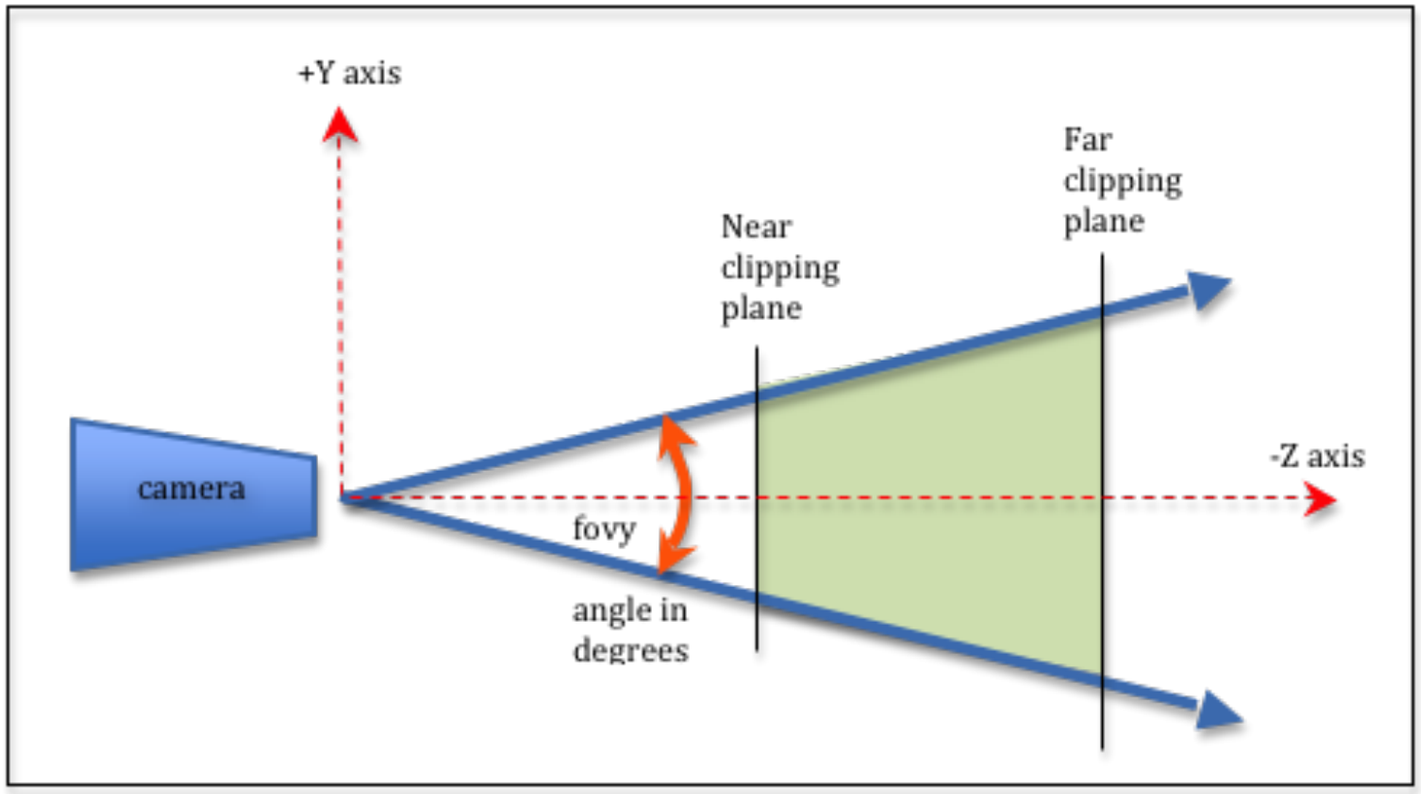
Week7/Demo4

Viewport transform

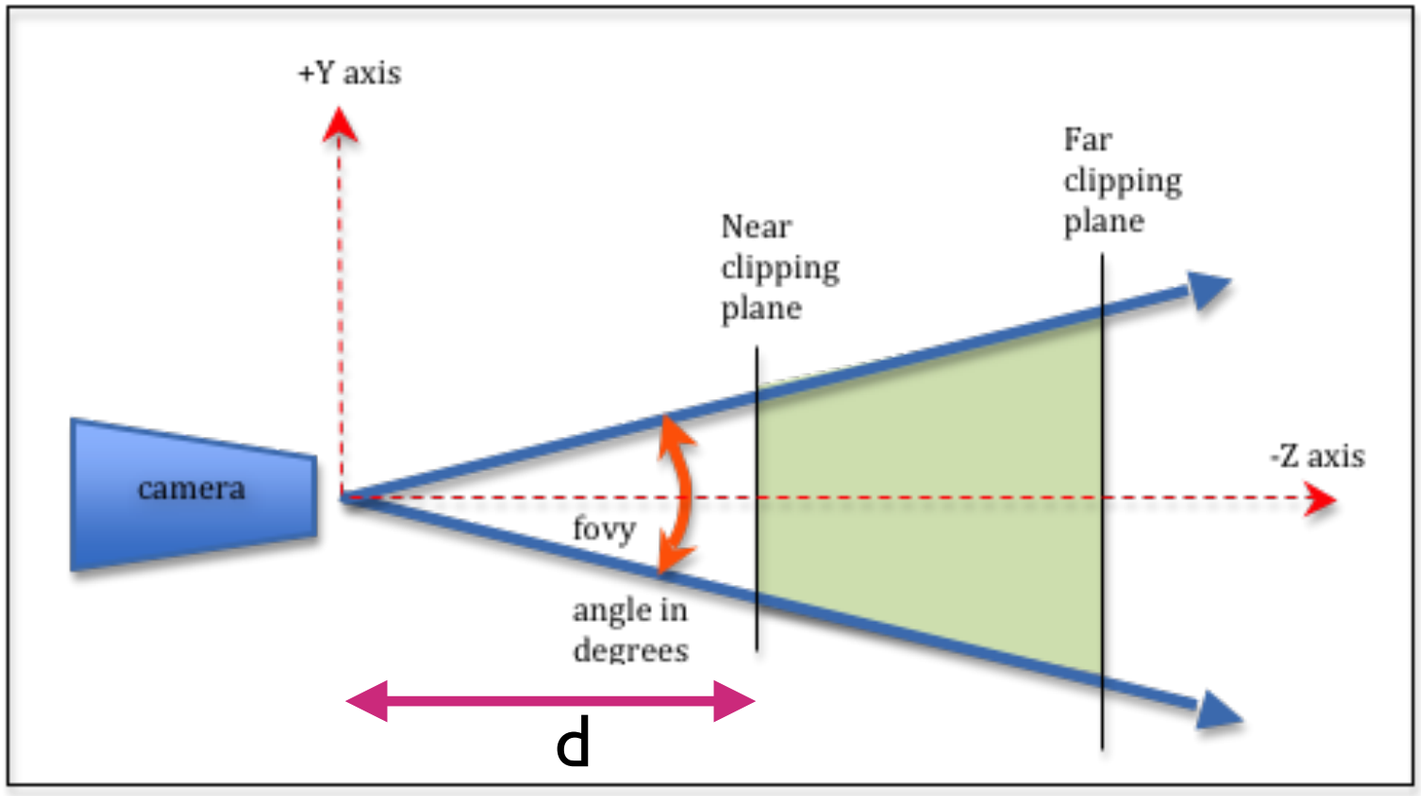


Semantics:
Scale/translate the view frustum (assuming cube $[-1,1]^3$) into the viewport

Algebra of perspective projection

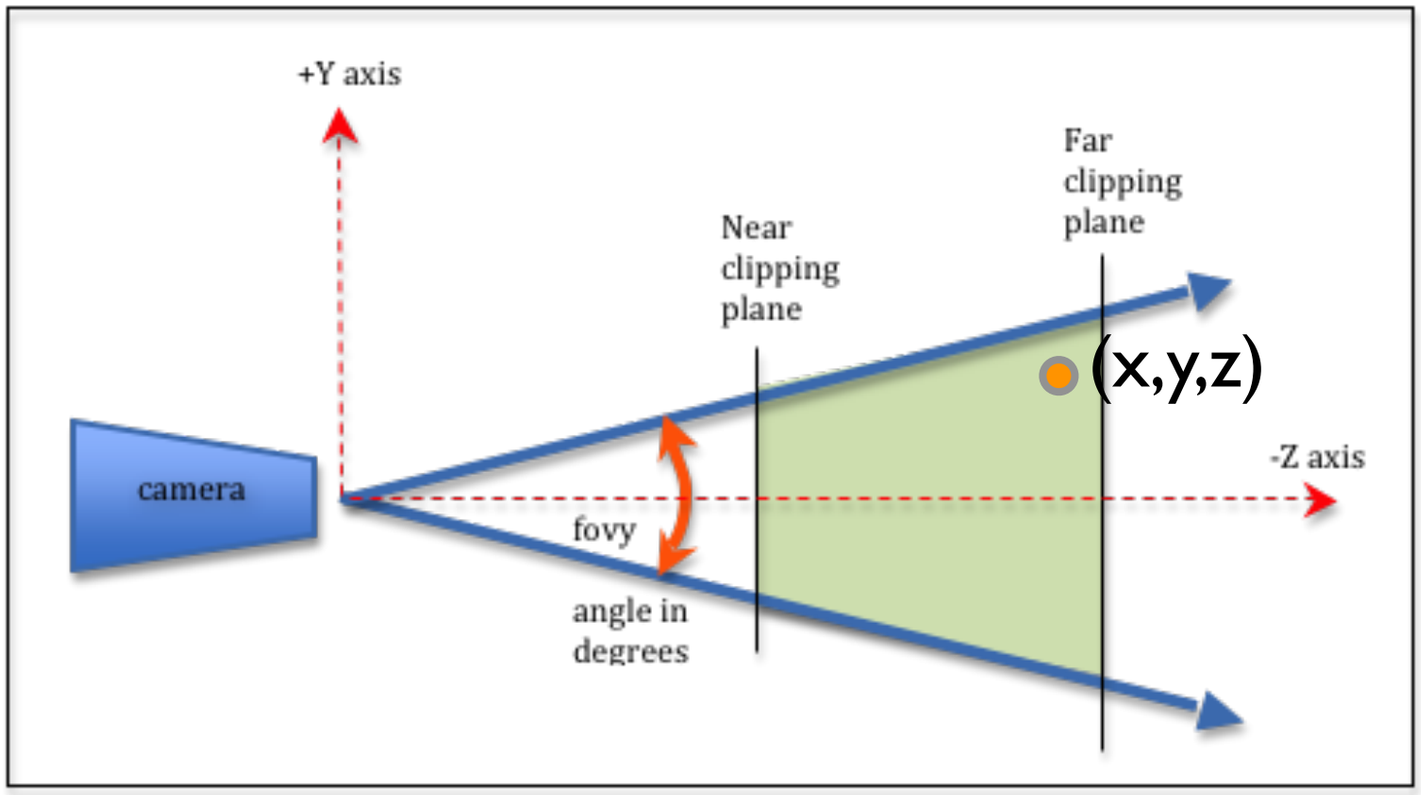


Algebra of perspective projection



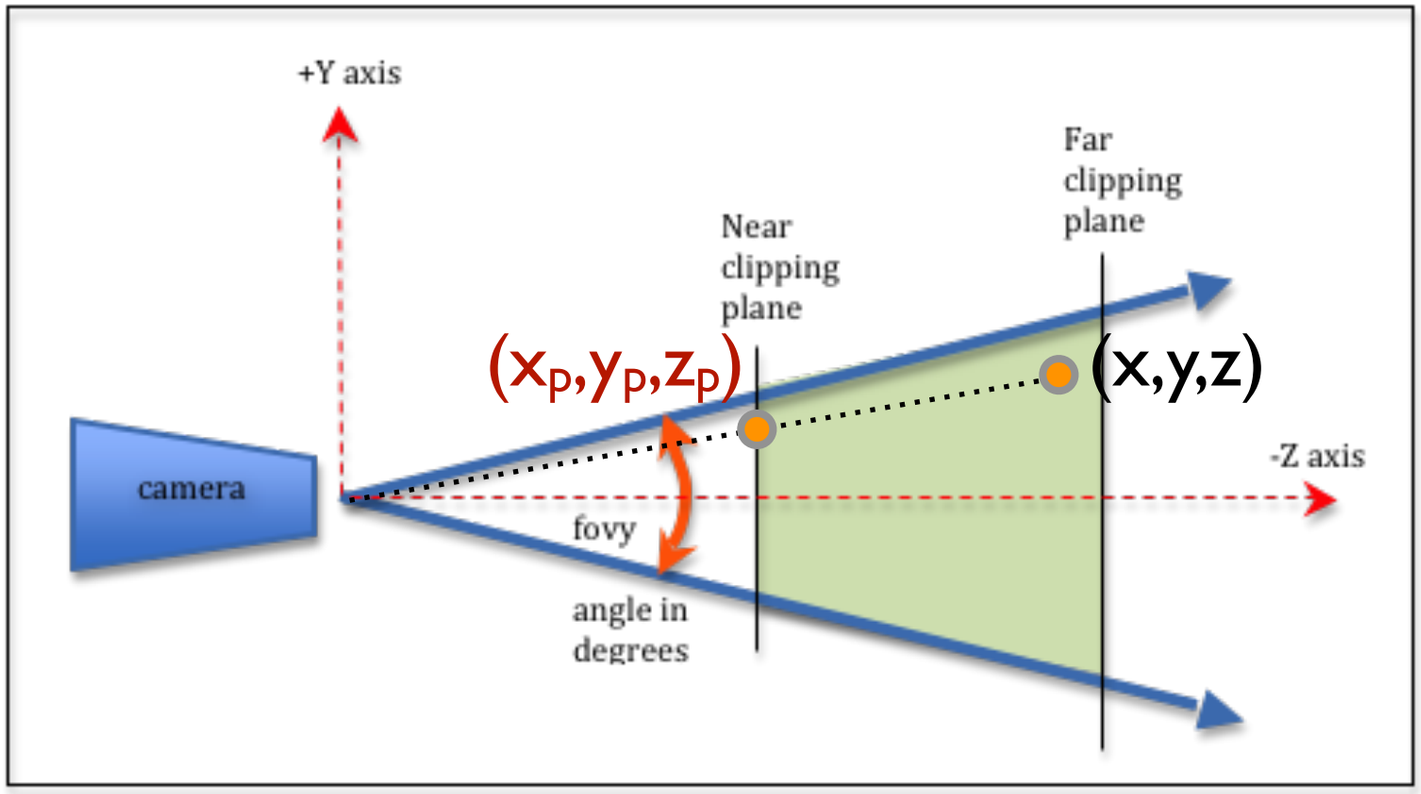
The *focal length* **d** is the distance of the "center" or projection - the "eye" in camera terms - to the projection plane (here we identified the projection plane with the near clipping plane)

Algebra of perspective projection



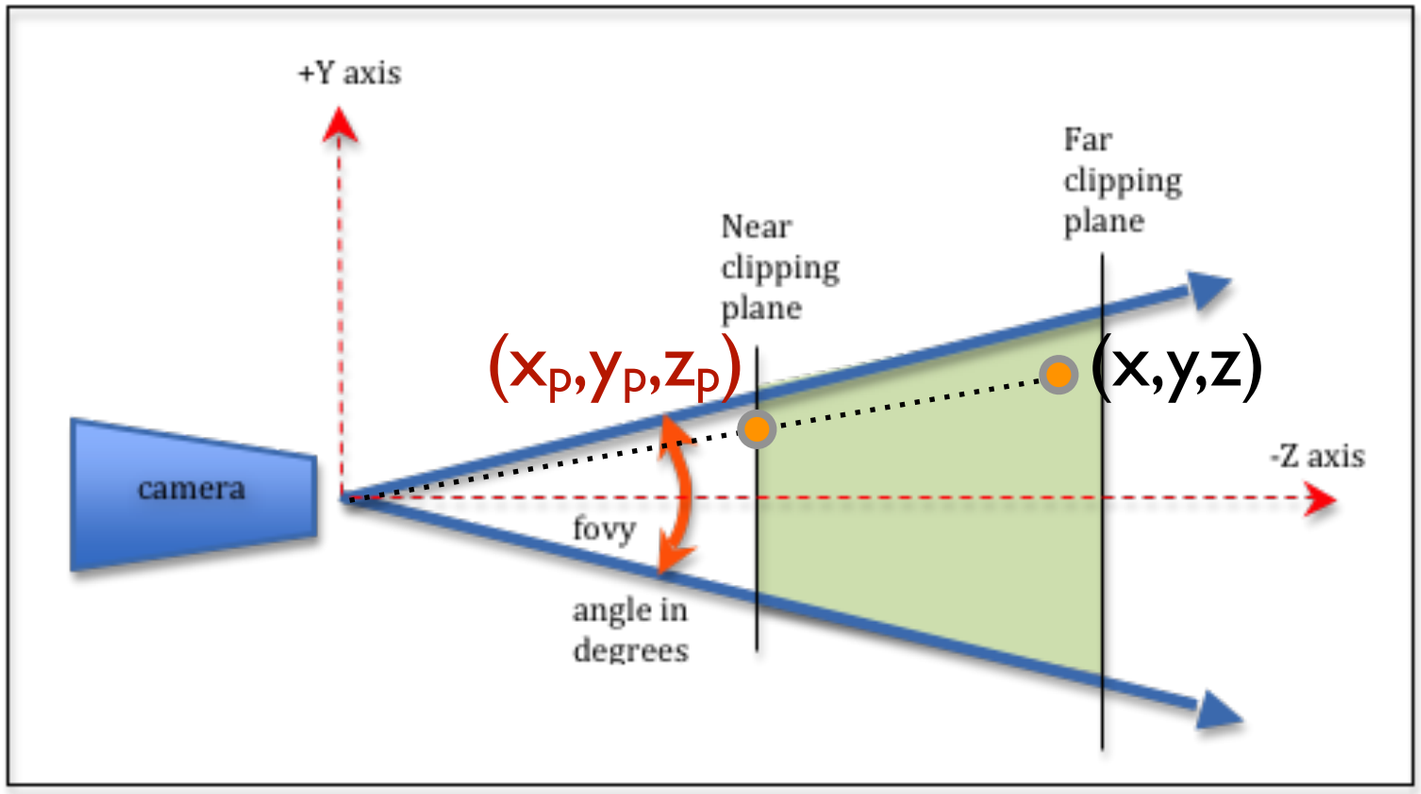
A point (x,y,z) in the view frustum …

Algebra of perspective projection



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

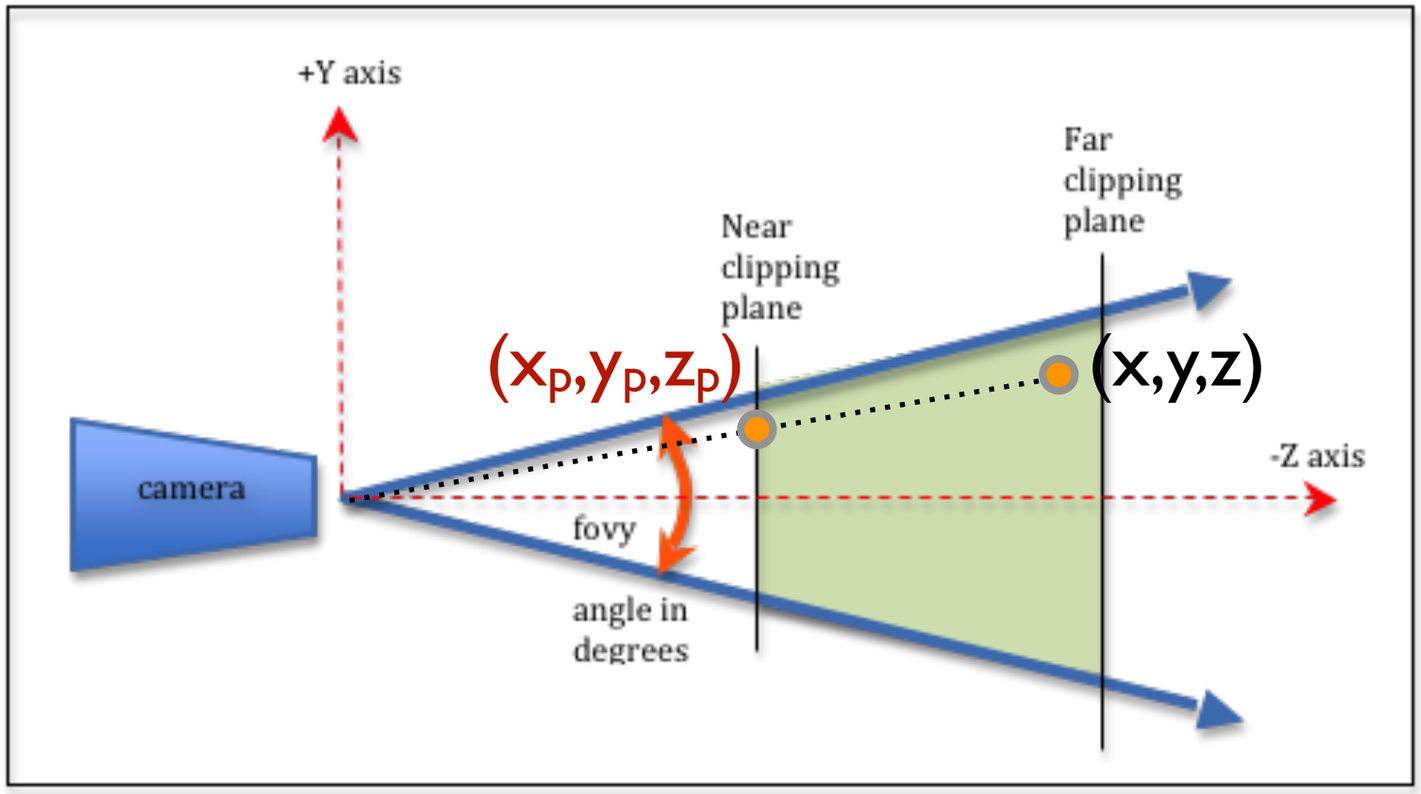
Algebra of perspective projection



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!)