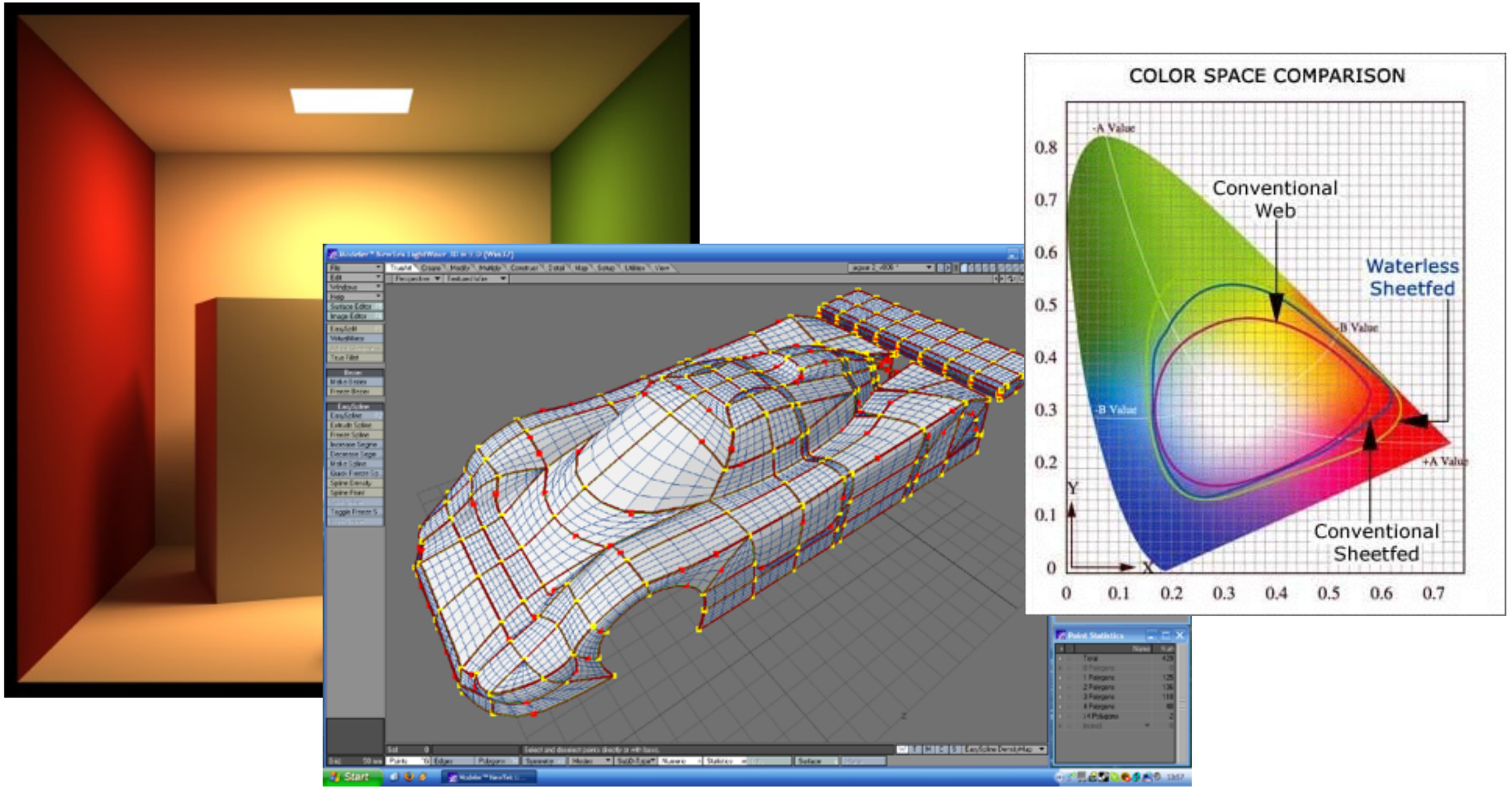


Lecture 12 : Cubics wrap-up (Bezier, B-Spline,subdivision) Intro to Viewing/Drawing in 3D: Coordinates/Transforms

Tuesday October 19th 2021

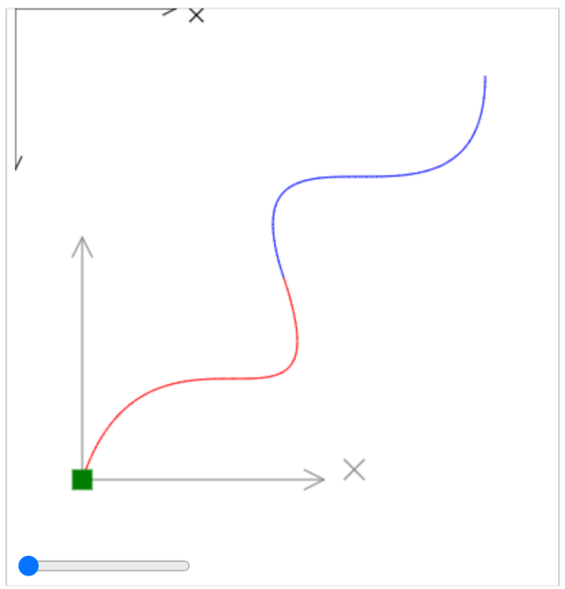
Logistics

- Midterm <u>will</u> 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.

Implementation example : 2-piece Hermite

jsbin.com/bodorun

Week6/Demo1



Brief wrap-up

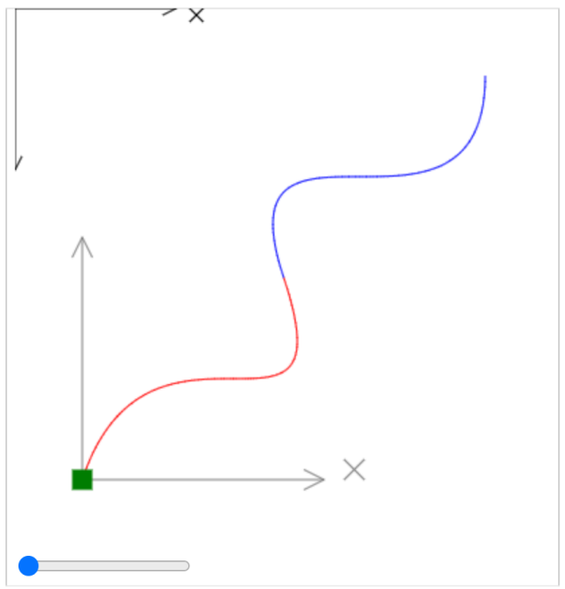
JavaScript

```
[...]
    var Hermite = function(t) {
        return [
        2*t*t*t-3*t*t+1,
        t*t*t-2*t*t+t,
        -2*t*t*t+3*t*t,
        t*t*t-t*t
        ];
    }

    var HermiteDerivative = function(t) {
        return [
        6*t*t-6*t,
        3*t*t-4*t+1,
        -6*t*t+6*t,
        3*t*t-2*t
        ];
    }

    function Cubic(basis,P,t){
        var b = basis(t);
        var result=vec2.create();
        vec2.scale(result,P[0],b[0]);
        vec2.scaleAndAdd(result,result,P[1],b[1]);
        vec2.scaleAndAdd(result,result,P[2],b[2]);
        vec2.scaleAndAdd(result,result,P[3],b[3]);
        return result;
    }

    var p0=[0,0];
    var d0=[1,3];
    var p1=[1,1];
    var d1=[-1,3];
    var p2=[2,2];
    var d2=[0,3];

    var P0 = [p0,d0,p1,d1]; // First two points and tangents
    var P1 = [p1,d1,p2,d2]; // Last two points and tangents

    var C0 = function(t_) {return Cubic(Hermite,P0,t_);};
    var C1 = function(t_) {return Cubic(Hermite,P1,t_);};

    var C0prime = function(t_) {return Cubic(HermiteDerivative,P0,t_);};
    var C1prime = function(t_) {return Cubic(HermiteDerivative,P1,t_);};
[...]
```

Implementation example : 2-piece Hermite

jsbin.com/bodorun

Week6/Demo1



JavaScript

```
[...]
    var p0=[0,0];
    var d0=[1,3];
    var p1=[1,1];
    var d1=[-1,3];
    var p2=[2,2];
    var d2=[0,3];

    var P0 = [p0,d0,p1,d1]; // First two points and tangents
    var P1 = [p1,d1,p2,d2]; // Last two points and tangents

    var C0 = function(t_) {return Cubic(Hermite,P0,t_);};
    var C1 = function(t_) {return Cubic(Hermite,P1,t_);};

    var C0prime = function(t_) {return Cubic(HermiteDerivative,P0,t_);};
    var C1prime = function(t_) {return Cubic(HermiteDerivative,P1,t_);};

    var Ccomp = function(t) {
        if (t<1){
            var u = t;
            return C0(u);
        } else {
            var u = t-1.0;
            return C1(u);
        }
    }

    var Ccomp_tangent = function(t) {
        if (t<1){
            var u = t;
            return C0prime(u);
        } else {
            var u = t-1.0;
            return C1prime(u);
        }
    }
[...]
```

B-splines

jsbin.com/viwinoj

Week6/Demo2

- Using 4 control points (p_0,p_1,p_2,p_3) it creates a curve that *approximates the arc from* p_1->p_2 (but doesn't necessarily go through either point)
- A sequence of curves that respectively use points (p_0,p_1,p_2,p_3), (p_1,p_2,p_3,p_4), (p_2,p_3,p_4,p_5) etc will join with C2-continuity! (but will only approximate the points)
- If passing through a point is needed, duplicate it in the sequence of control pts! (but C2-continuity is lost)
- Correction from notes …

$$\mathbf{B} = \frac{1}{6}\begin{bmatrix} 1 & 4 & 1 & 0 \\ -3 & 0 & 3 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}$$

B-splines

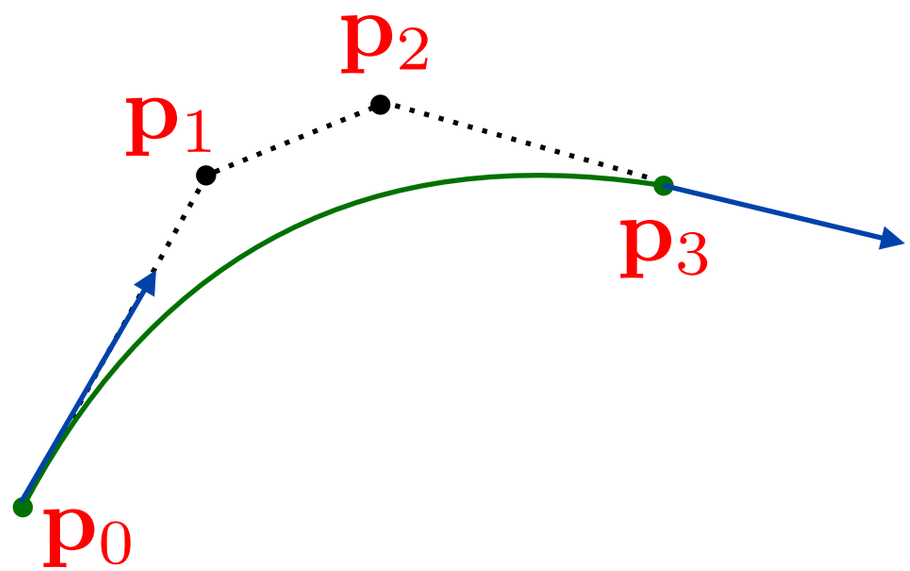
jsbin.com/viwinoj

Week6/Demo2

- Using 4 control points (p_0,p_1,p_2,p_3) it creates a curve that *approximates the arc from* p_1->p_2 (but doesn't necessarily go through either point)
- A sequence of curves that respectively use points (p_0,p_1,p_2,p_3), (p_1,p_2,p_3,p_4), (p_2,p_3,p_4,p_5) etc will join with C2-continuity! (but will only approximate the points)
- If passing through a point is needed, duplicate it in the sequence of control pts! (but C2-continuity is lost)
- Correction from notes …

$$\mathbf{B} = \frac{1}{6}\begin{bmatrix} 1 & 4 & 1 & 0 \\ -3 & 0 & 3 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}$$

Bezier cubics (and subdivision)

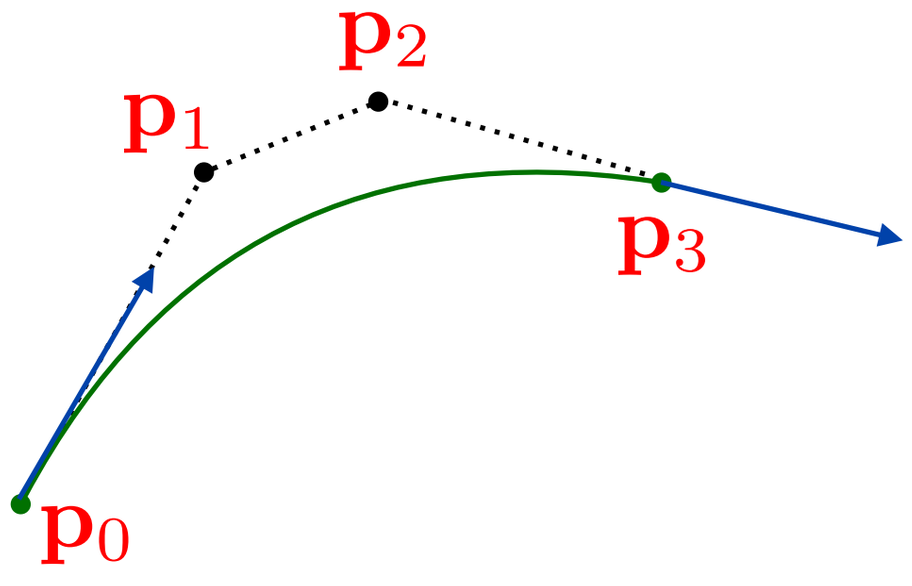


- Conceptually, a "variant" of Hermite!
- Initial gradient set to $\mathbf{f}'(0) = \gamma(\mathbf{p}_1 - \mathbf{p}_0)$
- Final gradient set to $\mathbf{f}'(1) = \gamma(\mathbf{p}_3 - \mathbf{p}_2)$

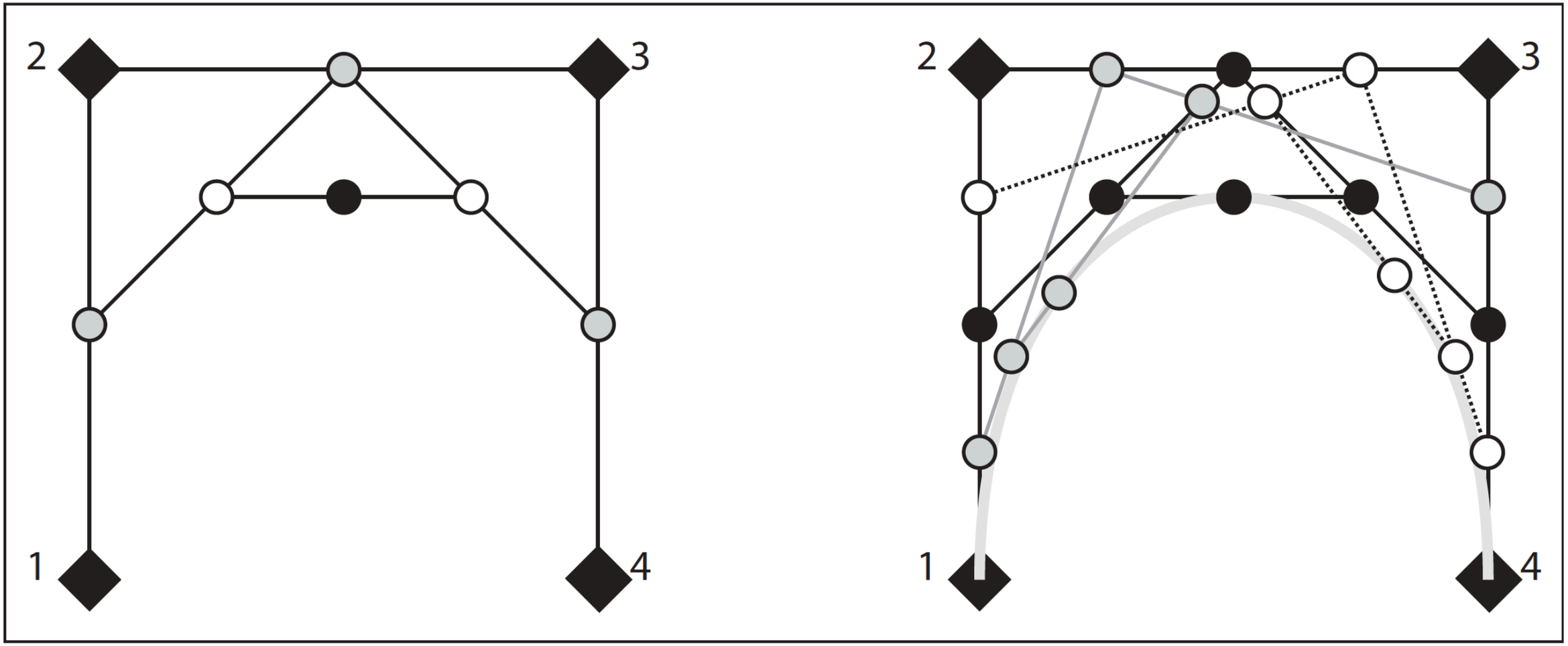
Basis matrix in FCG Ch.15

- Special value $\boldsymbol{\gamma}$=3 guarantees the *interpolation property*(this is when we speak of Bezier curves)

Bezier cubics (and subdivision)



- Can be constructed via *De Casteljau Subdivision*



Natural cubics

- Specify up to 2nd derivative at origin *C(0),C'(0),C''(0)* and just position *C(1)* at end
- (Look in FGC Chapter 15 for basis matrix)
- Methodology:
  - Define the k-th curve component (using value at start/ end, 1st/2nd derivative at start).
  - Evaluate what 1st/2nd derivatives will be at end
  - Define the (k+1)-st component using the previously computed derivatives as conditions for the start
- Yields C2-continuity; loses local control

Today's lecture

- (starting) Intro to drawing and viewing in 3D
  - Starting with some software examples
  - The concepts can appear tricky and difficult to build intuition on … that's why we will go slow and revisit concepts both today and on Thursday
  - Objective for today: Start building intuition via visuals, continue with theoretical concepts
  - 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)

Today's lecture (build-up to final 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

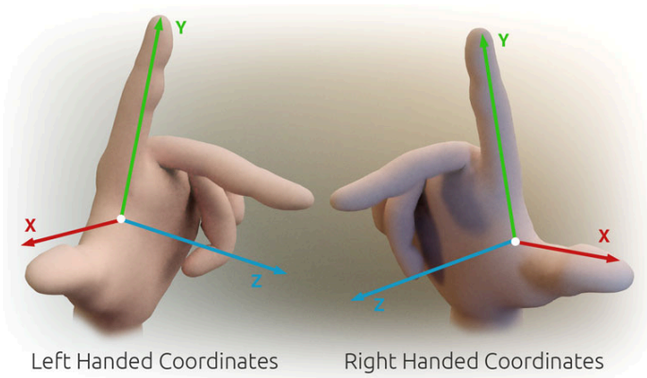
Explicit projection transform (orthographic) … and a teaser for perspective projection

jsbin.com/peredov

Week7/Demo3

3D theory primer

- Coordinate systems have a 3rd (z-axis) component
- Points/vectors depicted as triplets (x,y,z)
- We use *right-handed* coordinate systems



Left Handed Coordinates Right Handed Coordinates

- Homogeneous representations of vectors/transforms
- What do rotations look like?
  - Rotation around one of the 3 axes
  - Rotation around an arbitrary vector (more soon)
- Look at glMatrix vec3/mat4 for relevant tools!

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad \begin{pmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Today's lecture

jsbin.com/ficoxeh

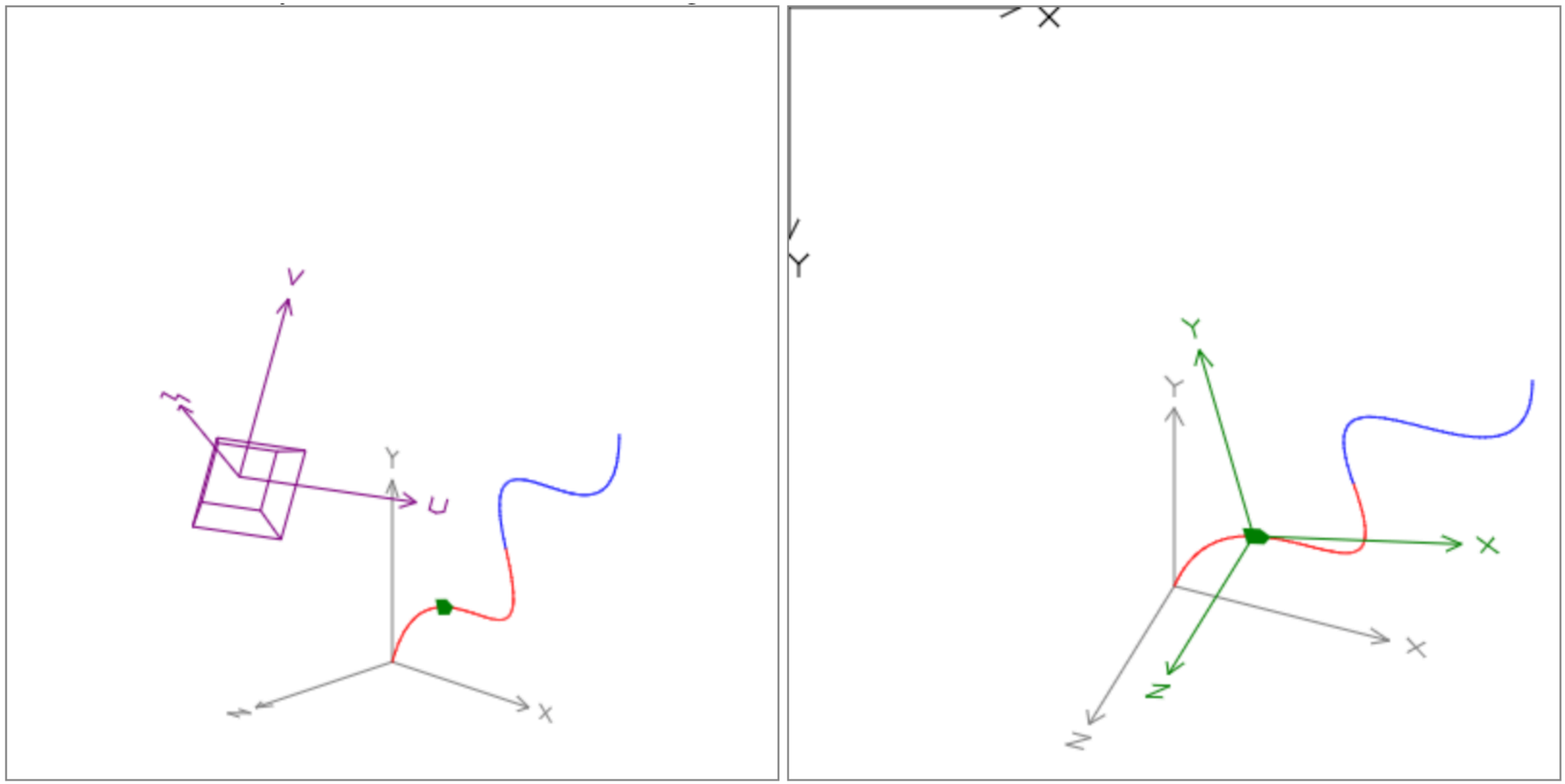
Week7/Demo4

- (bit later) A deep dive into viewing and drawing in 3D
  - Emphasis on the insights between the coordinate systems involved (there are several of them …) and the transformations between them
  - We will continue to build exposure to 3D points and transforms, and the homogeneous representations of the two (i.e. transforms in 3D as 4x4 matrices)
- Key terms and concepts (some for Thursday)
  - The various coordinate systems (*object, world, camera, Normalized-Device-Coordinates, canvas/screen*)
  - Projection and relation to homogeneous representations

Demo walkthrough

jsbin.com/ficoxeh

Week7/Demo4

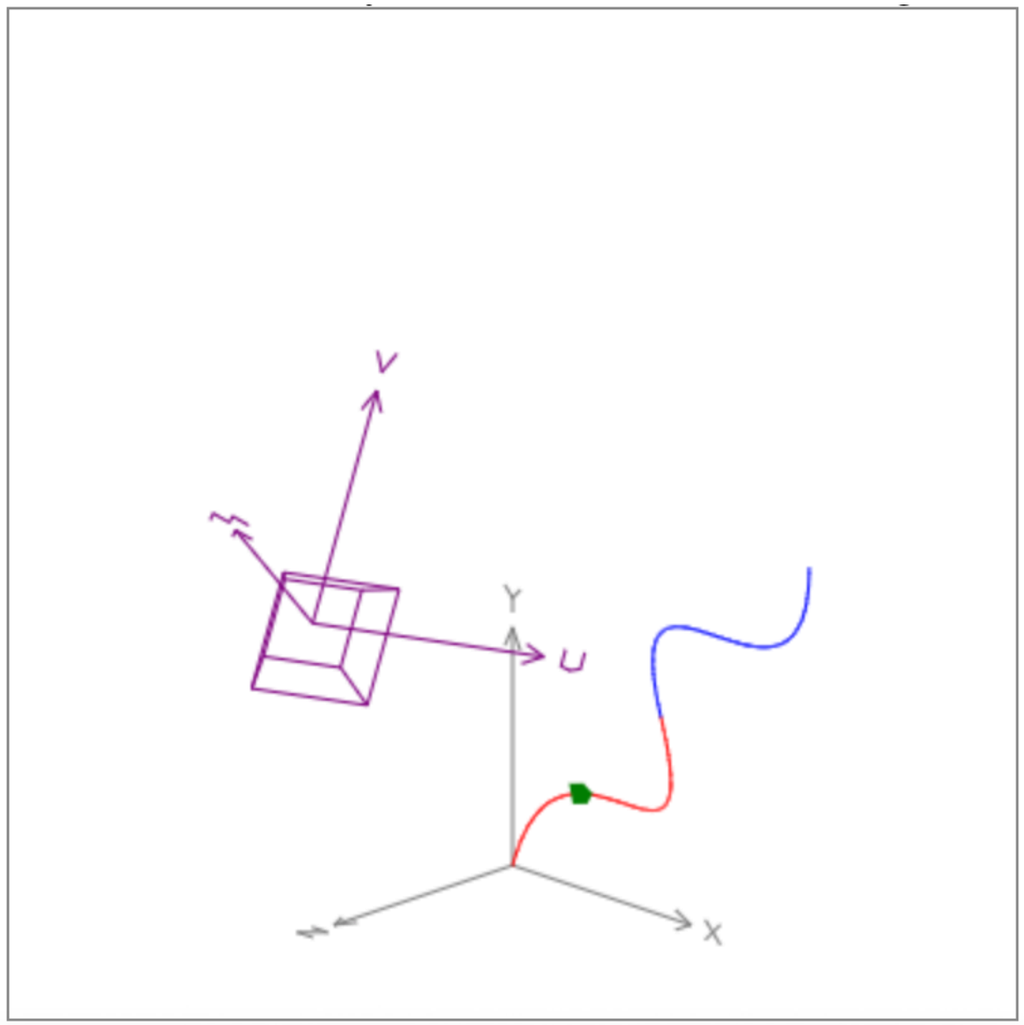


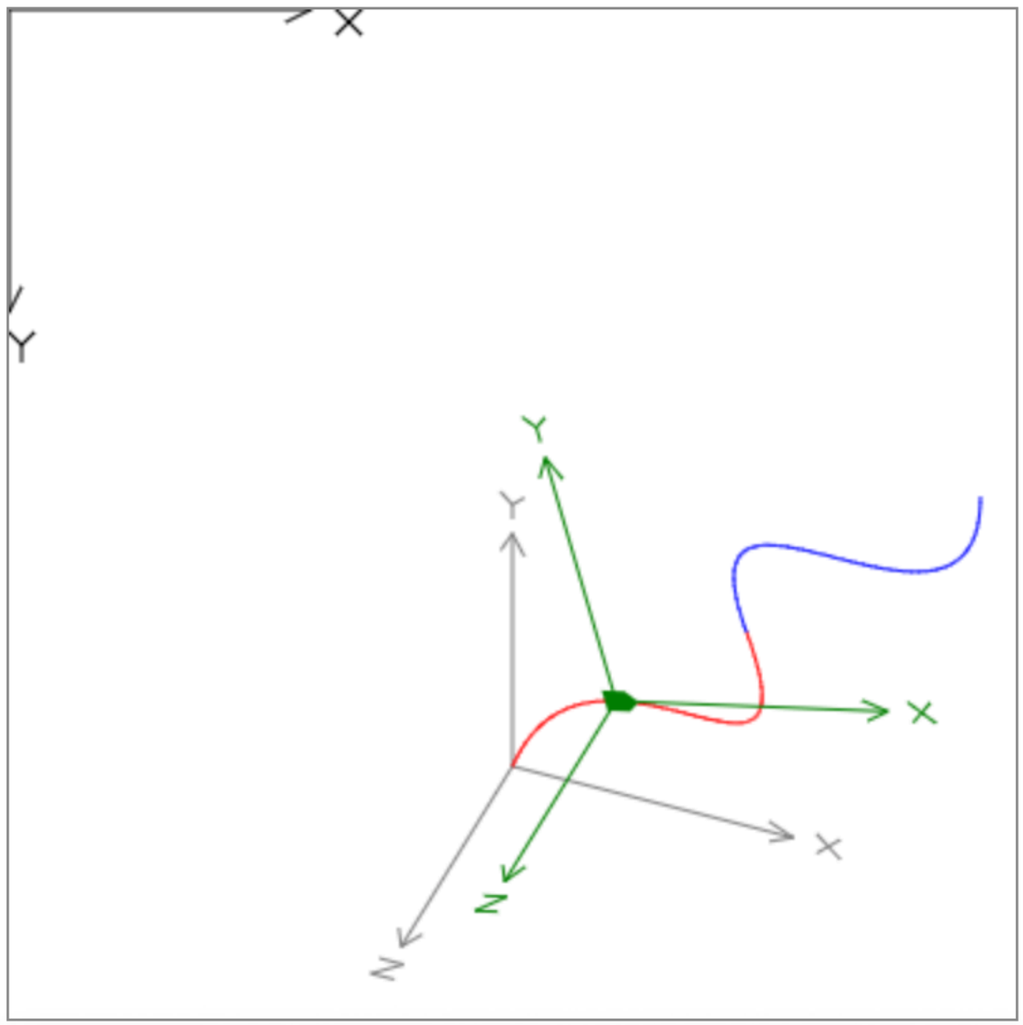
A conceptual 3D world, with some familiar content
(a piecewise-cubic curve on the XY-plane, and an object moving along it)

Demo walkthrough

jsbin.com/ficoxeh

Week7/Demo4





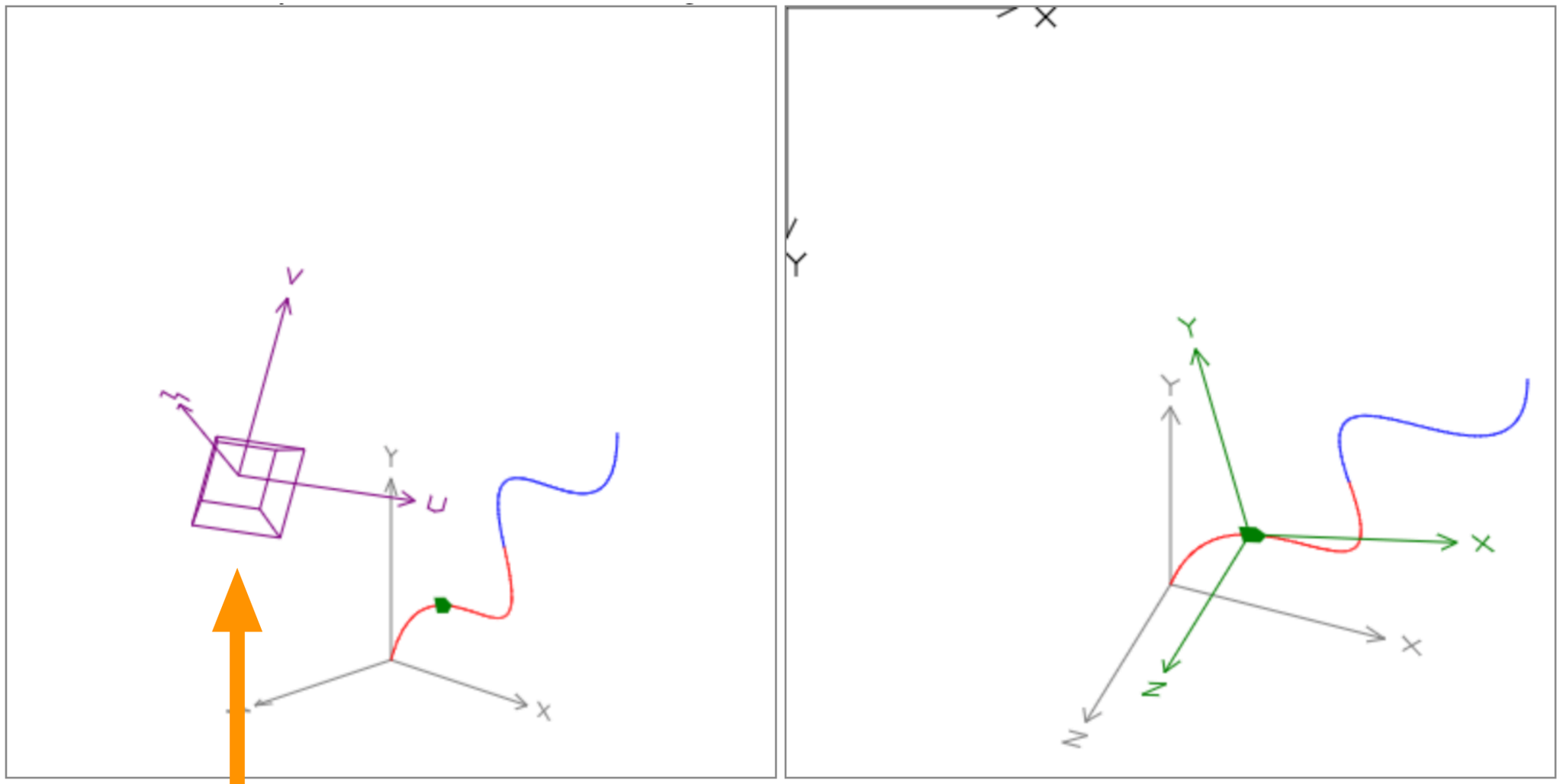
Left window:
A view of the world from a faraway vantage point, that not only shows the 3D content we *intend* to draw (i.e. the curve and moving object) but also the *camera* observing it.

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

jsbin.com/ficoxeh

Week7/Demo4

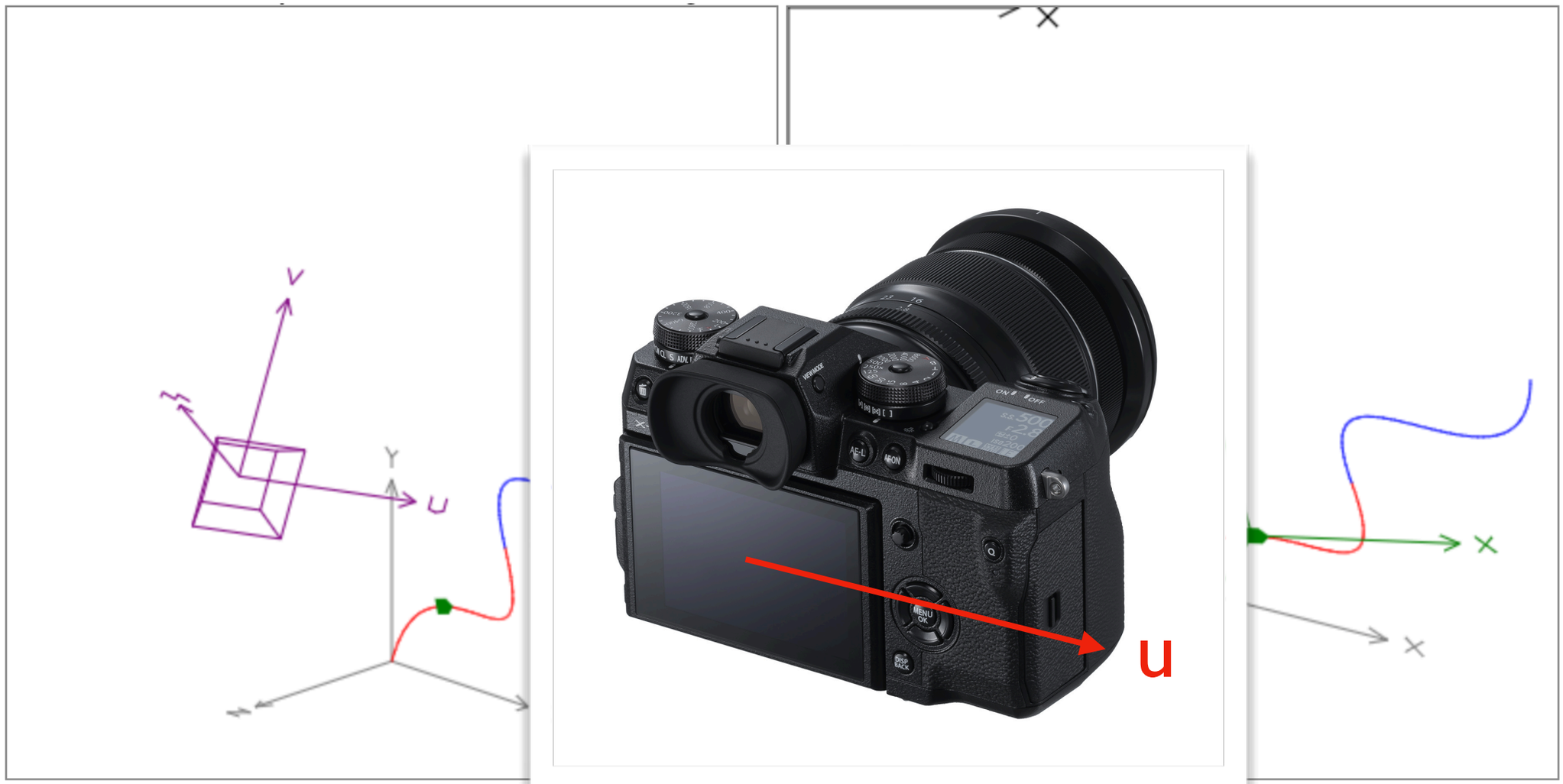


Shown in purple in this illustration (movable along a circle in the demo) The *camera coordinate system* is permanently affixed to it, and its axes are labeled “u”, “v”, and “w”.

The camera

jsbin.com/ficoxeh

Week7/Demo4

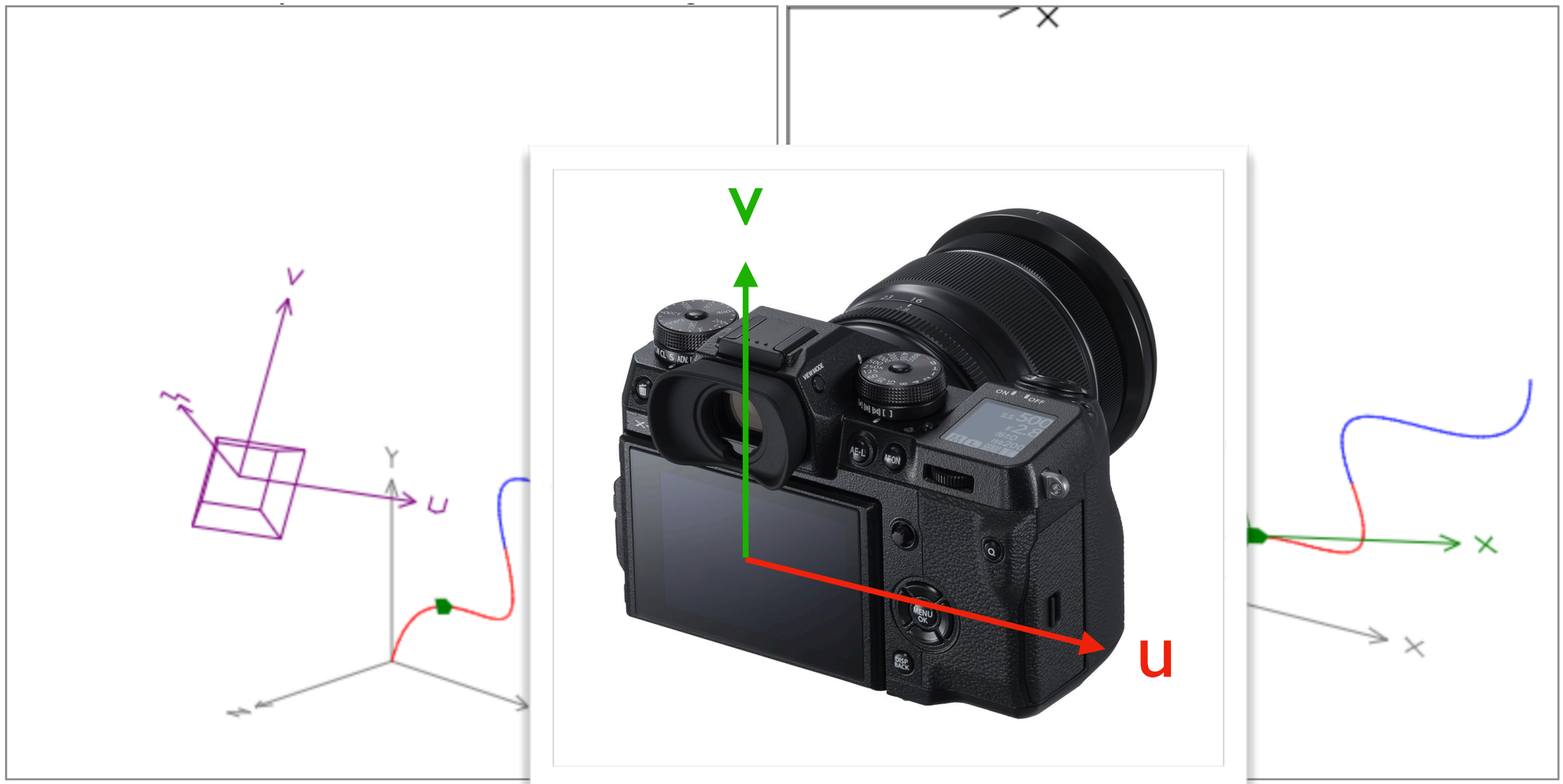


The origin of the coordinate system is at the “center” of the image (say, the center of the sensor where the image is being captured). The “u” axis is on the plane of the sensor, and oriented along the horizontal direction in the camera

The camera

jsbin.com/ficoxeh

Week7/Demo4

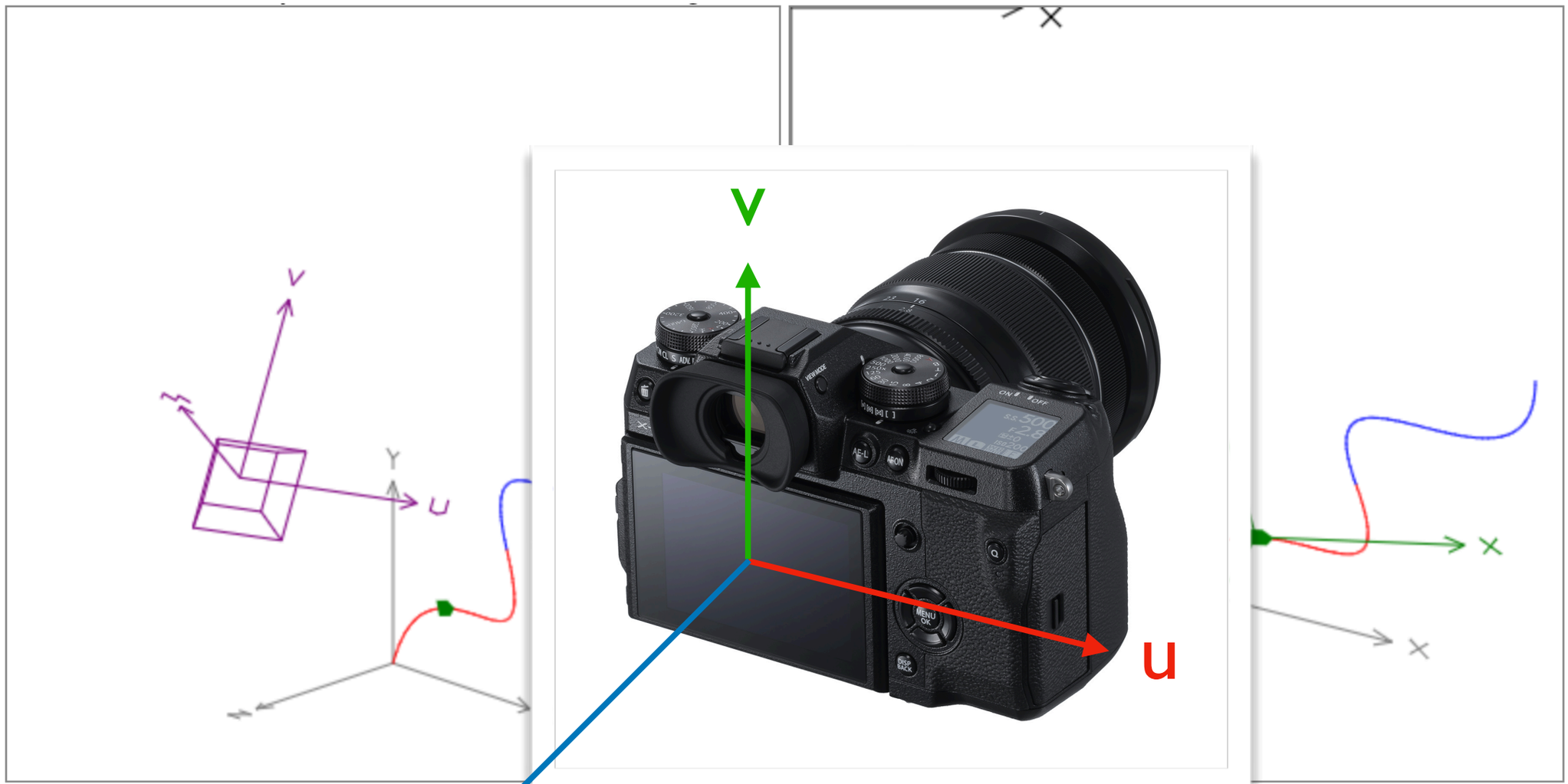


The “v” axis is also on the plane of the sensor, and oriented along the vertical direction in the camera

The camera

jsbin.com/ficoxeh

Week7/Demo4

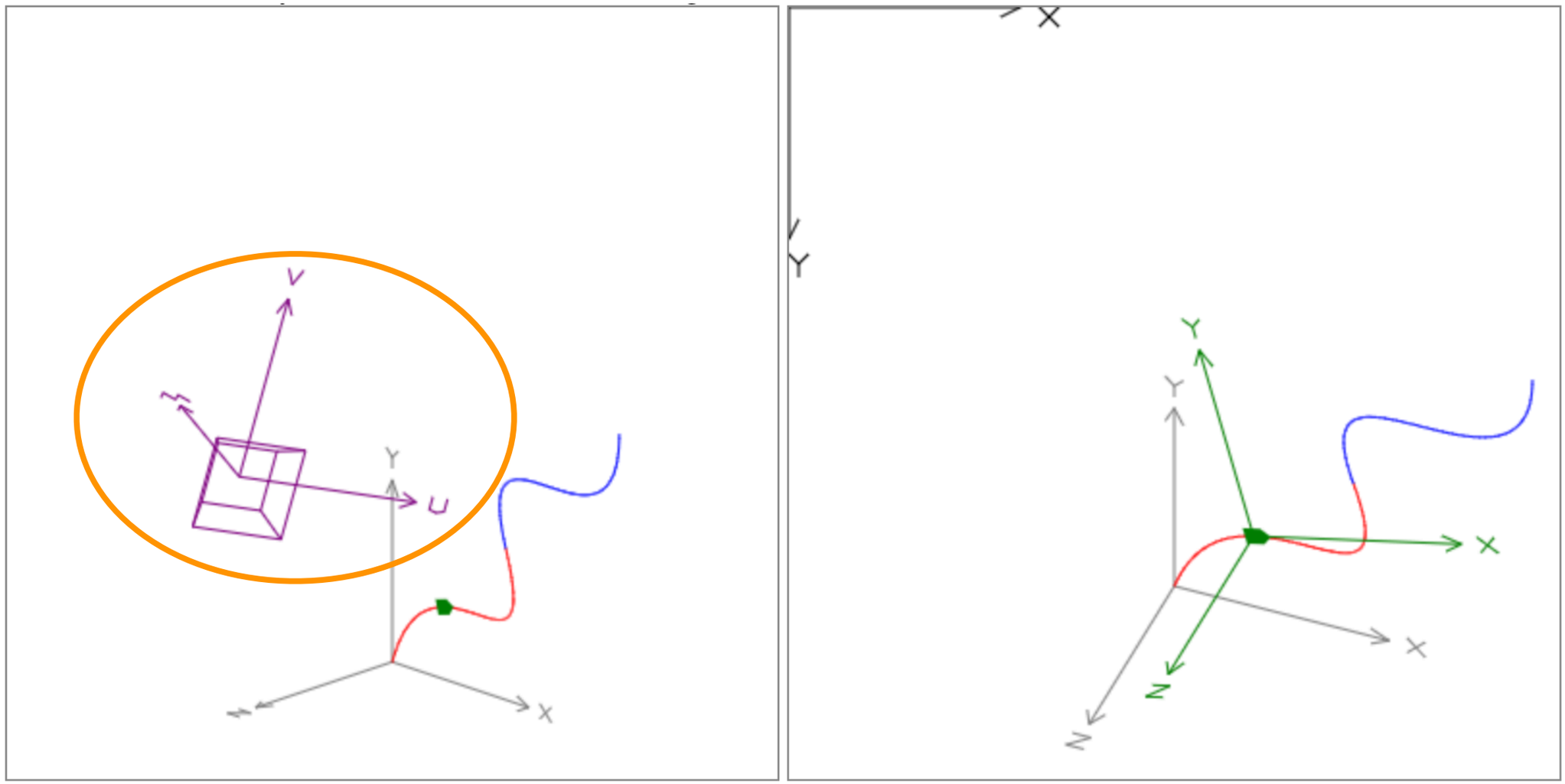


The “w” axis is perpendicular to the camera sensor, and points *away* from the scene being observed (i.e. towards the person holding the camera)

The camera

jsbin.com/ficoxeh

Week7/Demo4

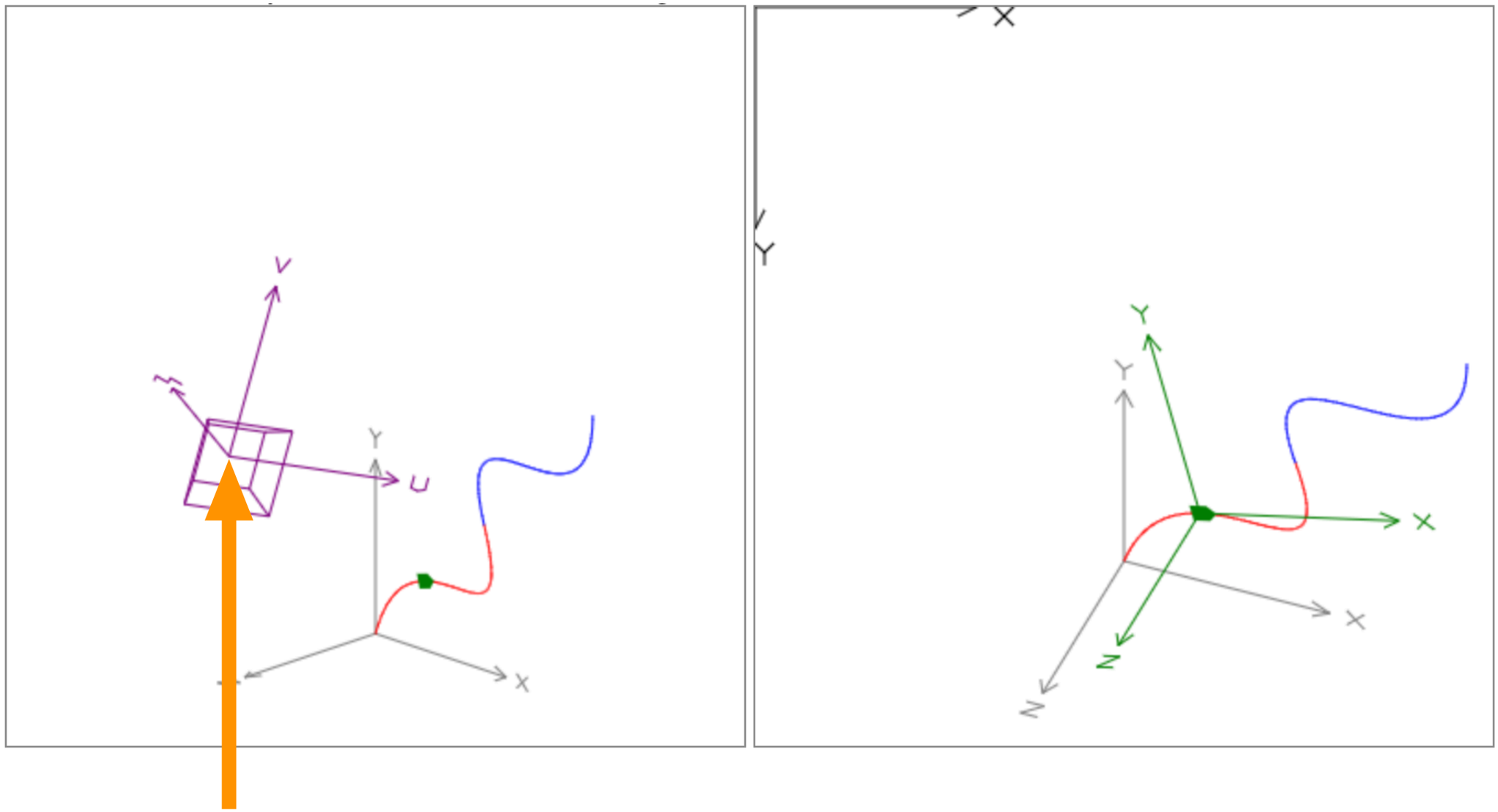


In our code demo, the camera is depicted as a cropped pyramid (with the short face being the "sensor/back side" of the camera, and the large face being the "lens/front side")

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

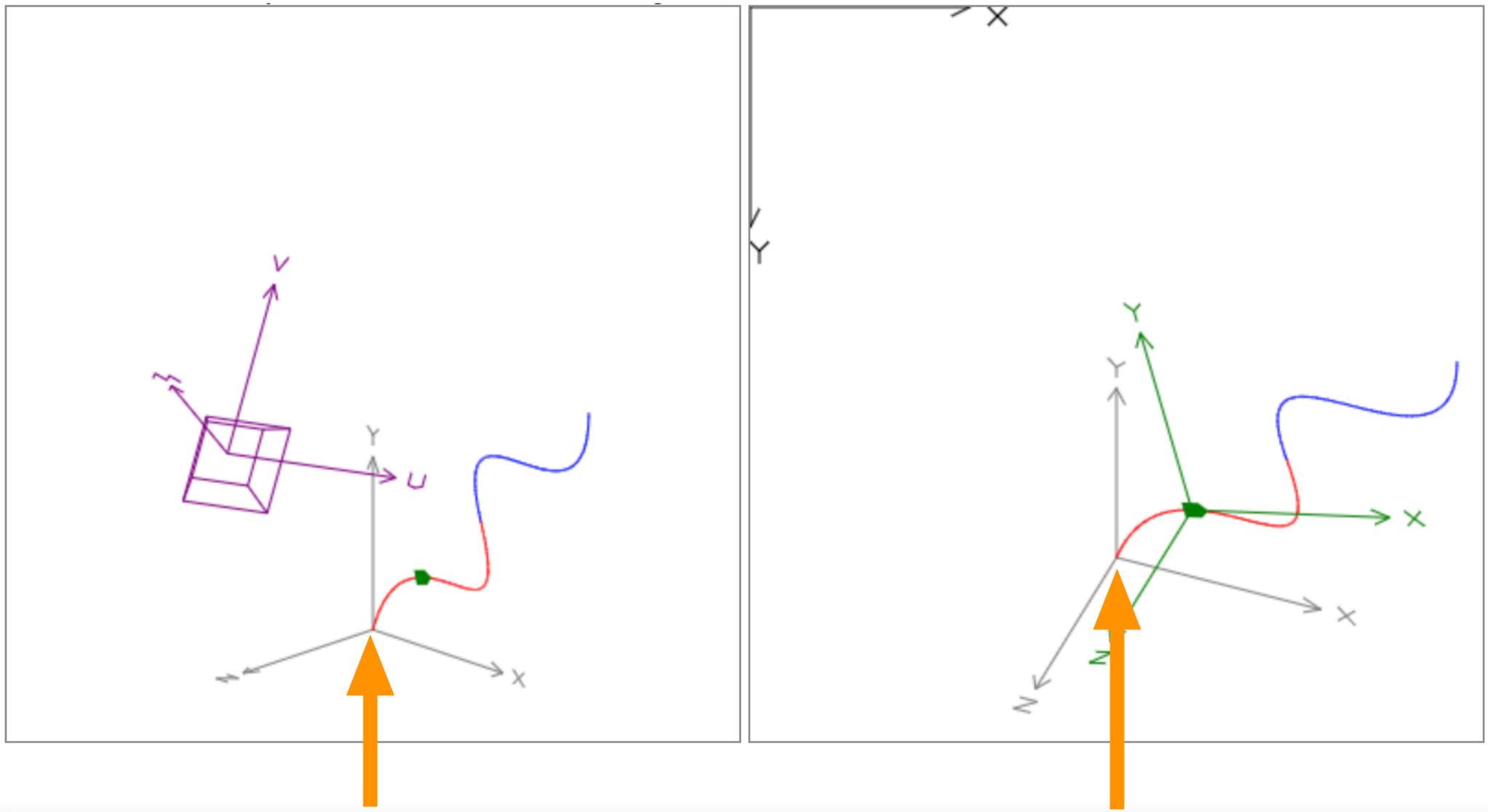
Week7/Demo2

- (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

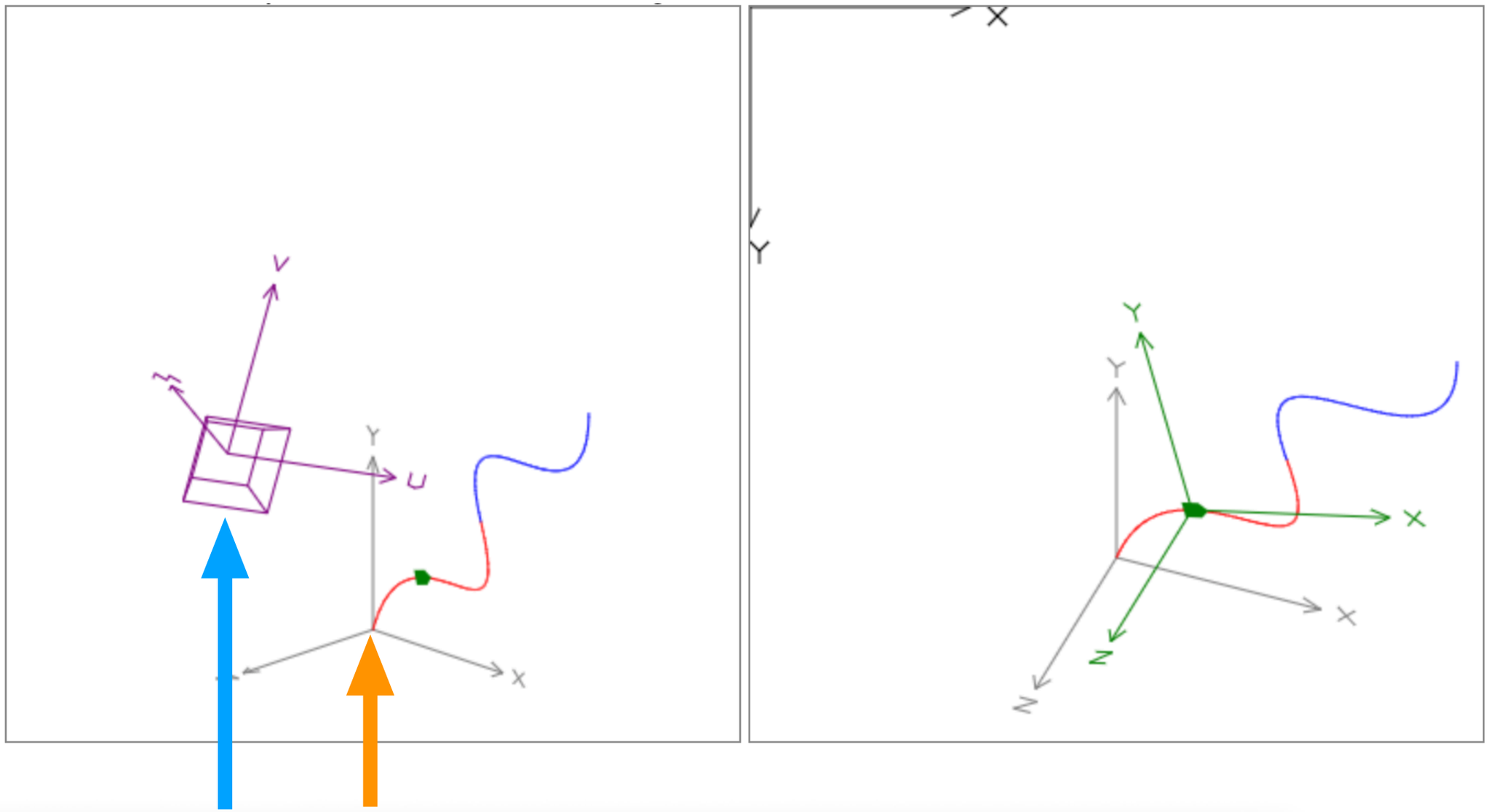


Illustrated in this demo in grey color, the *world coordinate system* is a system that we arbitrarily choose, in such a way that describing locations/vectors/coordinates of things we want to draw becomes convenient.
(For example, the locations of control points for the 2 piecewise-cubics are given in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

“World” coordinates

jsbin.com/ficoxeh

Week7/Demo4

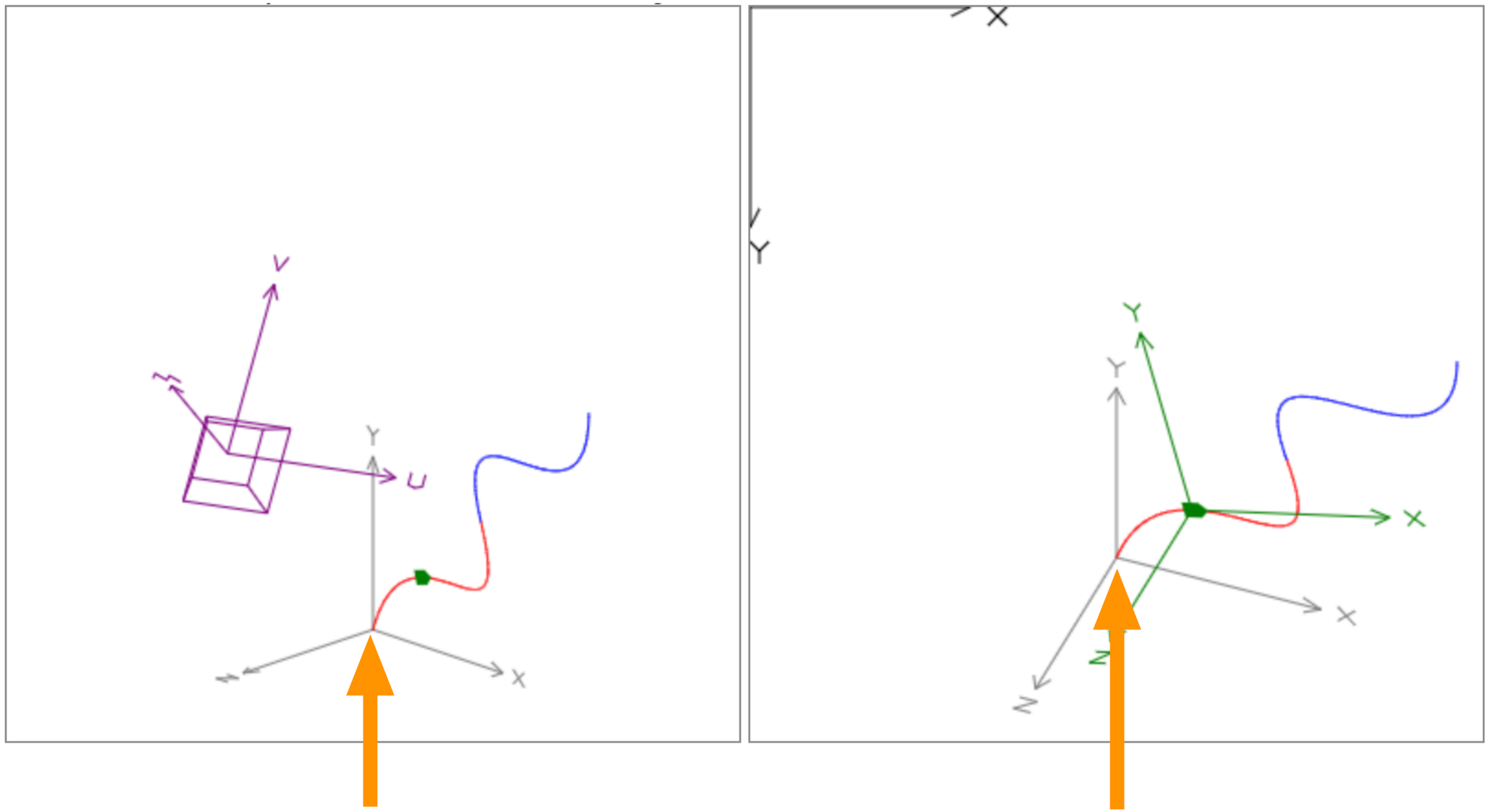


For purposes of “what do we see on the screen”, it doesn’t quite matter where the coordinate system is placed (it’s an arbitrary choice). What matters is “how is the camera positioned” relative to that world coordinate system …

“World” coordinates

jsbin.com/ficoxeh

Week7/Demo4

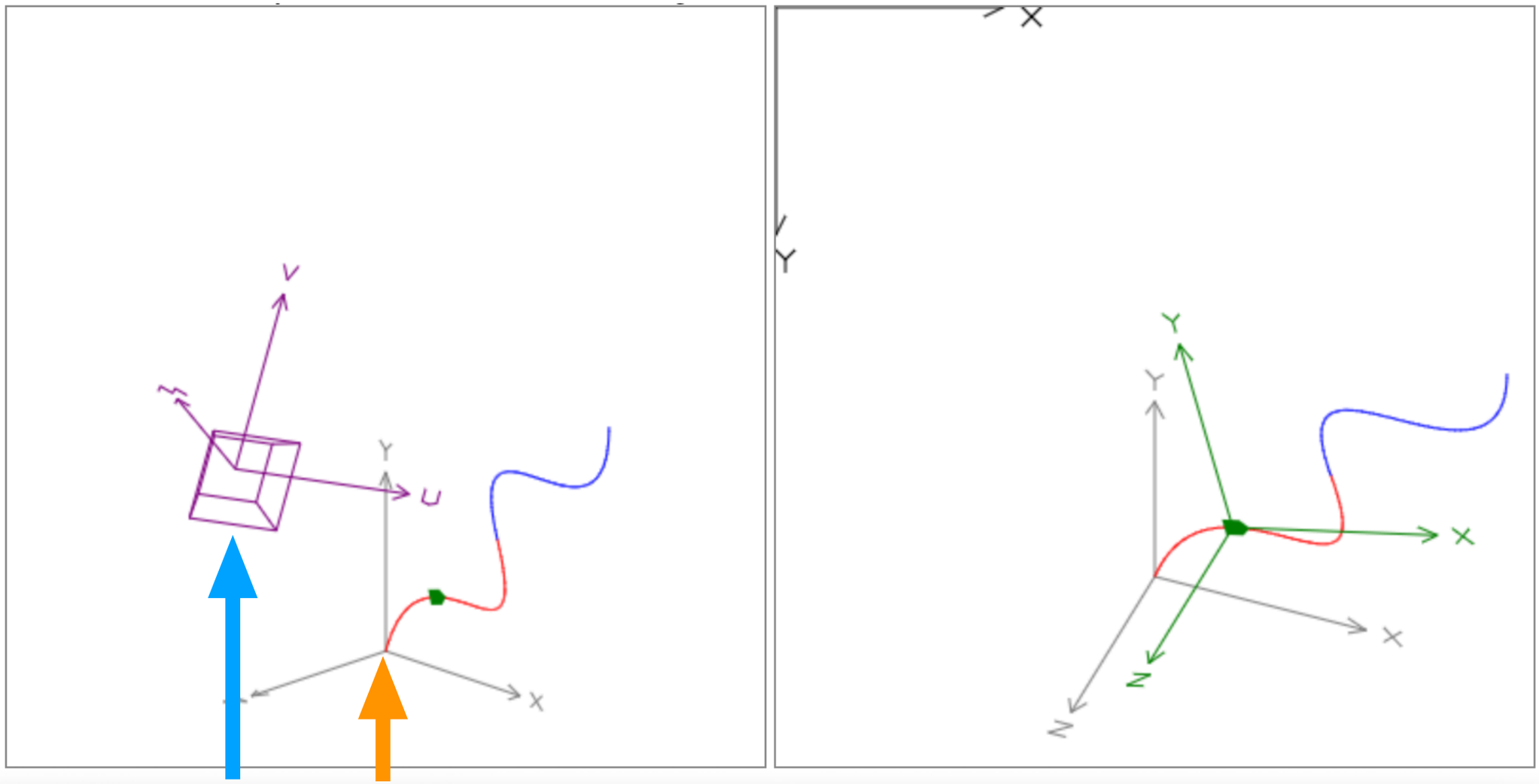


Note that this same coordinate system (grey axes) shows up differently in our two windows … this is because the relative placement of the point of observation (on the right, being the “purple” camera; on the left a “faraway” observer) is different in the 2 cases.

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*.

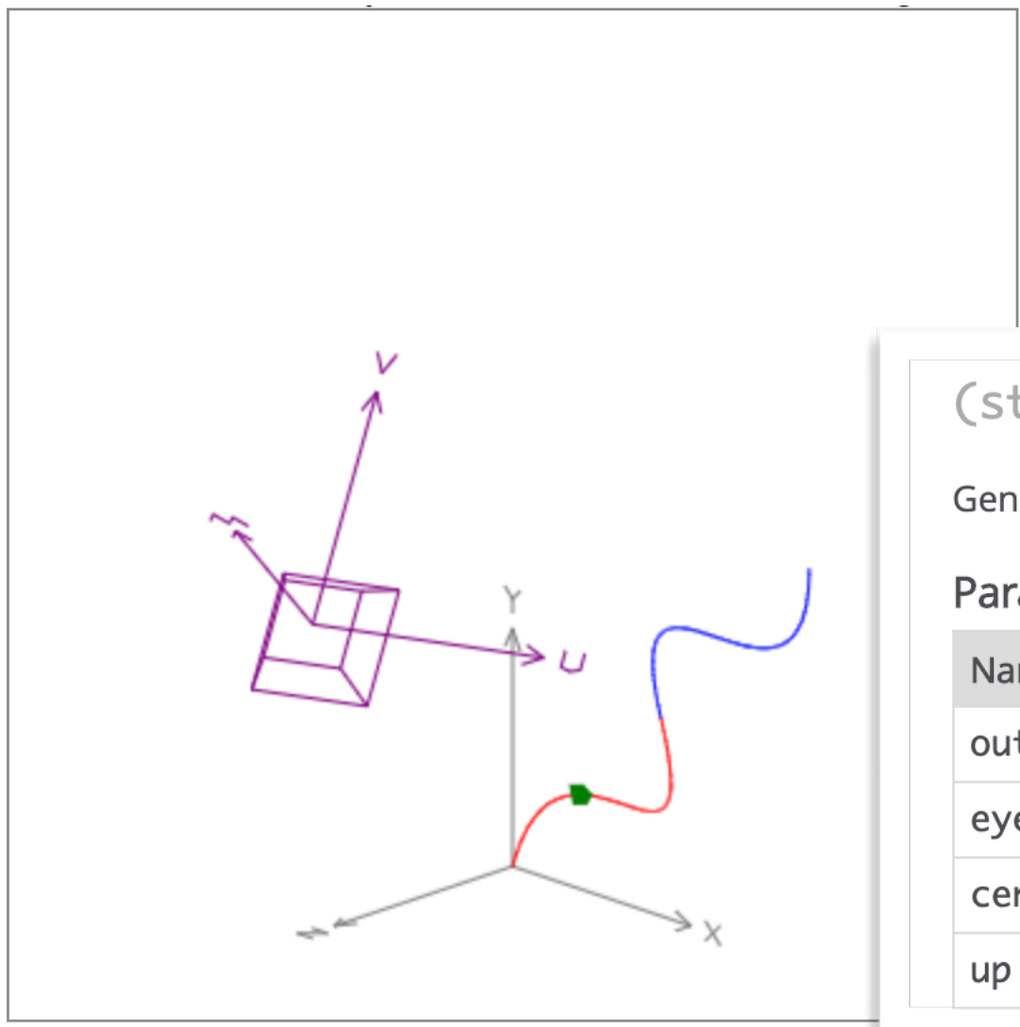
This is called the ***lookAt transform***.

(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:

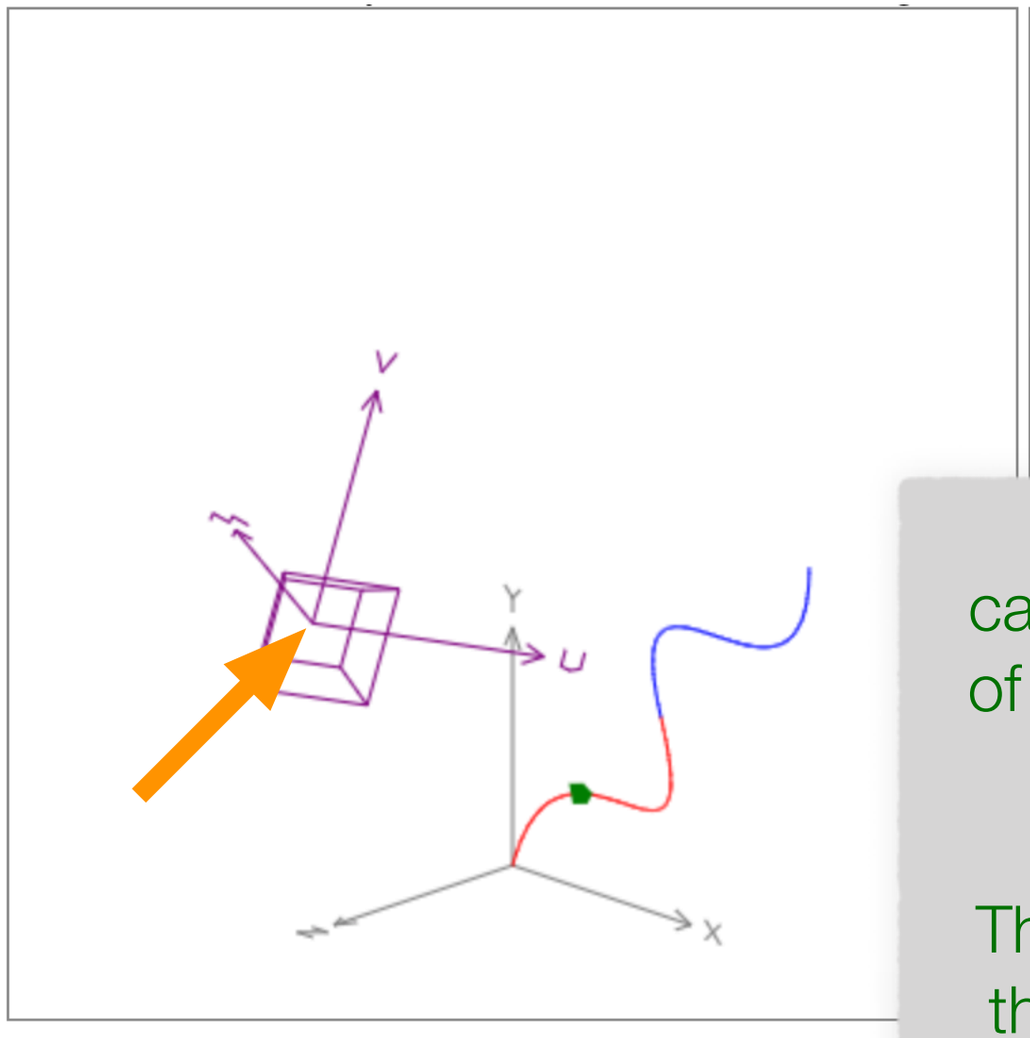
Name	Type	Description
out	mat4	mat4 frustum matrix will be written into
eye	ReadonlyVec3	Position of the viewer
center	ReadonlyVec3	Point the viewer is looking at
up	ReadonlyVec3	vec3 pointing up

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.

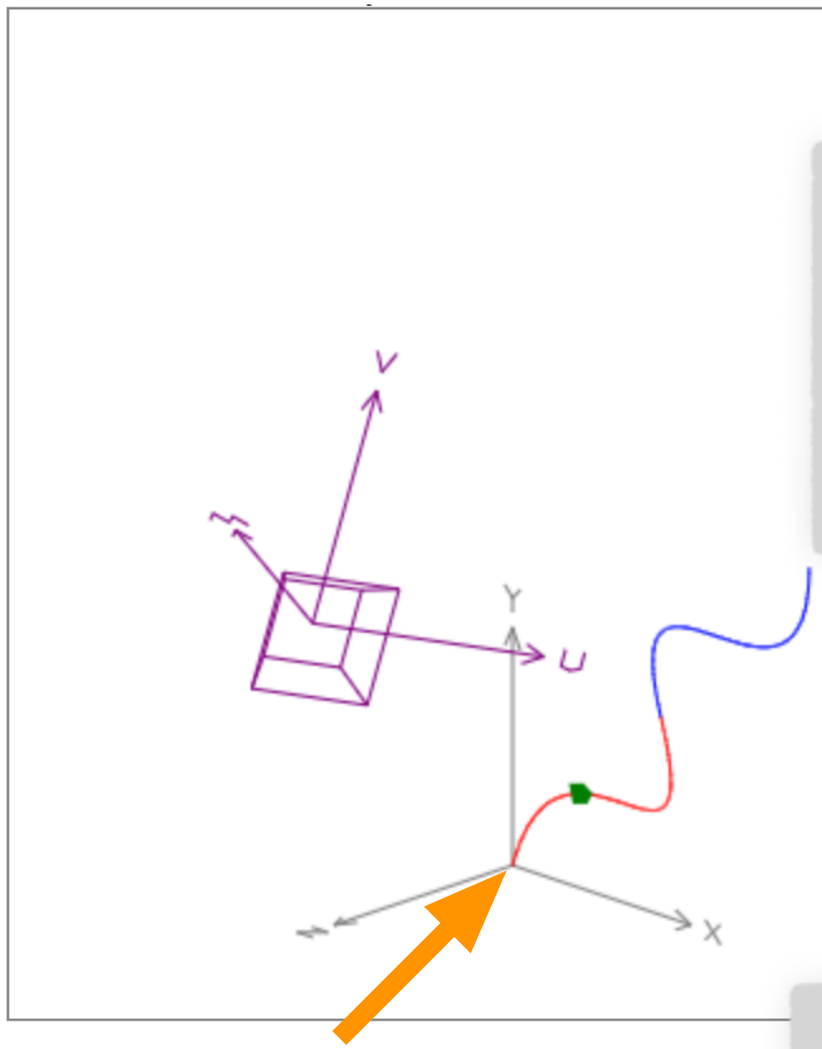
Parameters:

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.

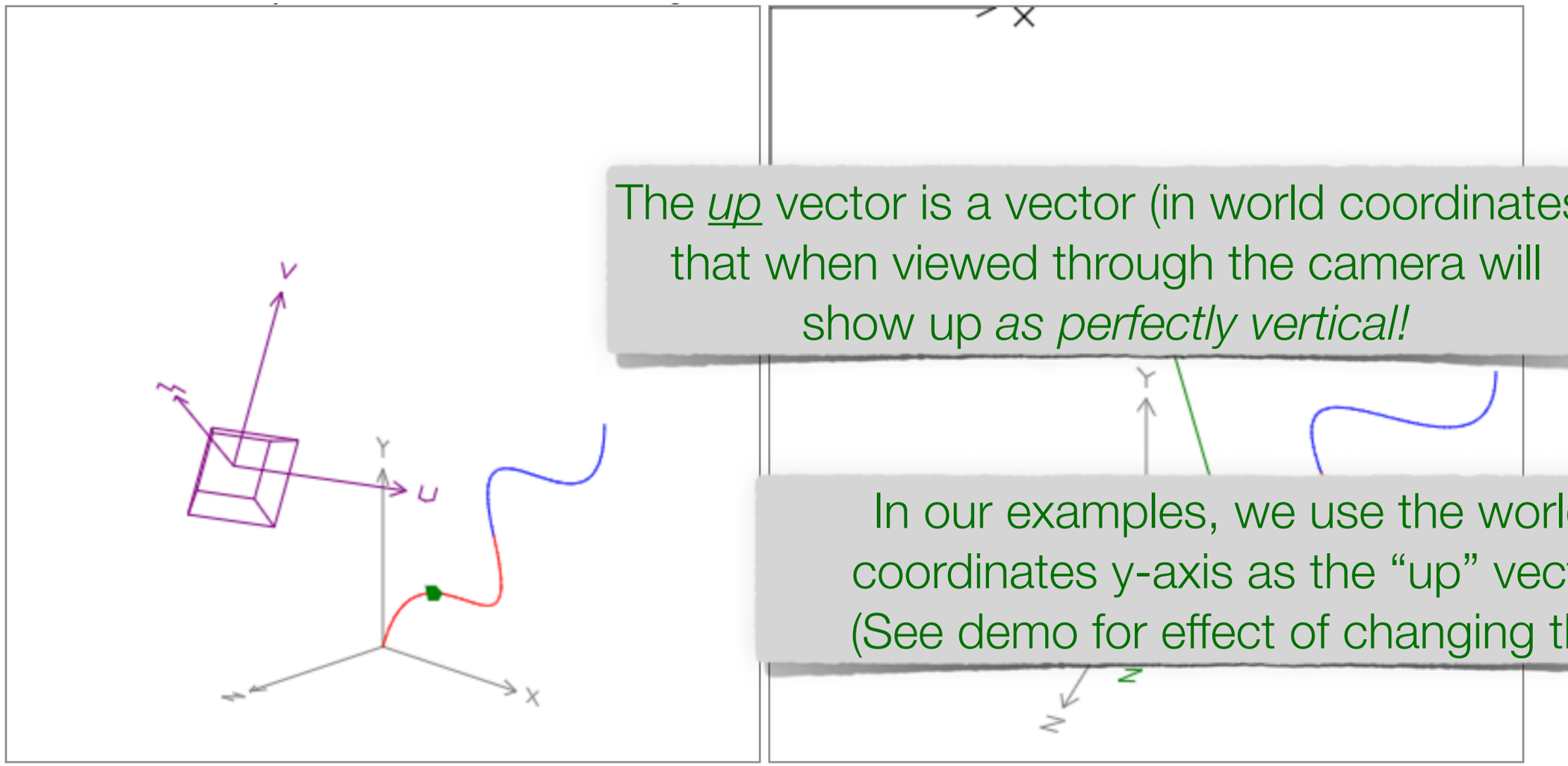
Parameters:

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:

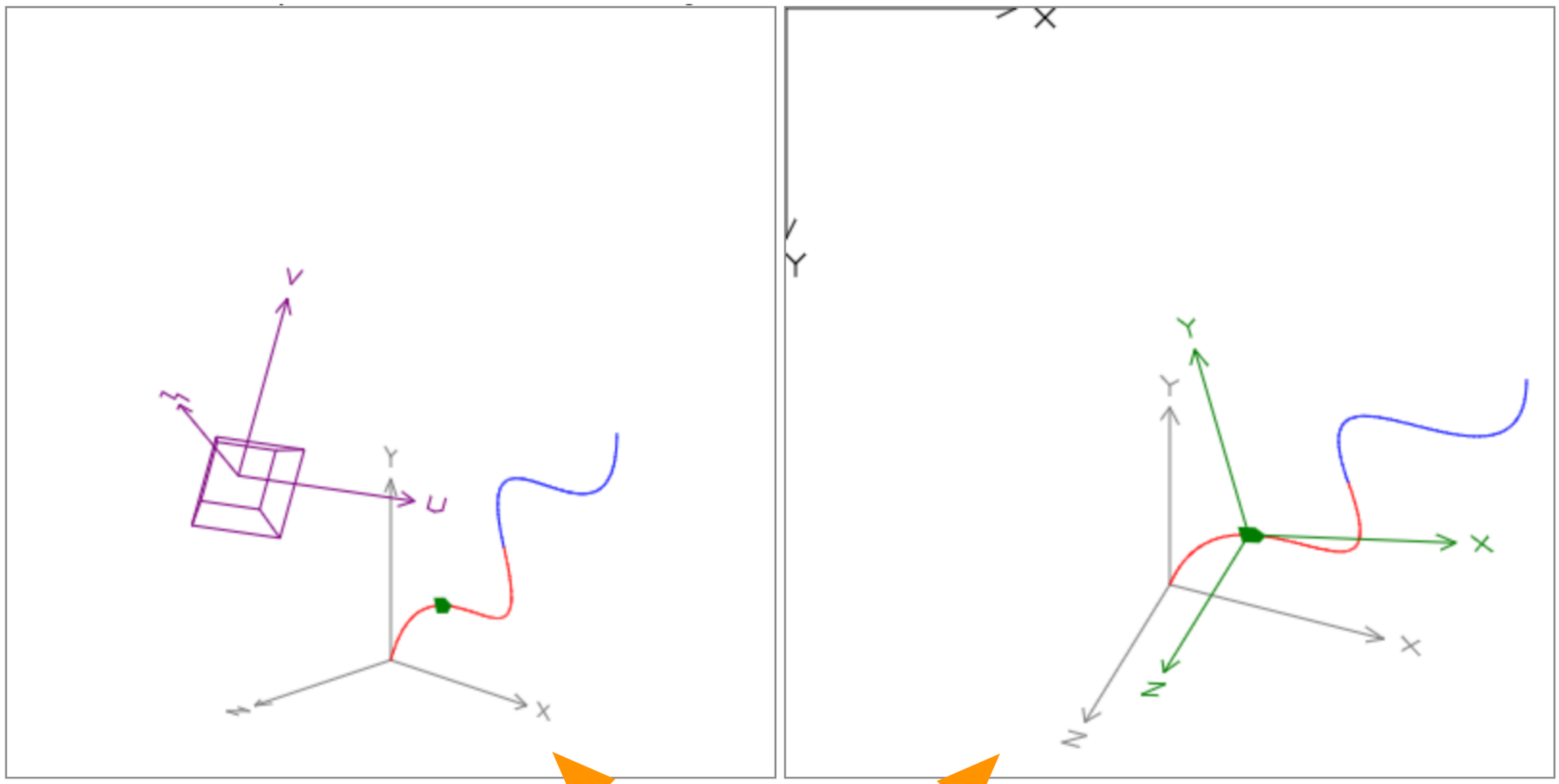
Name	Type	Description
out	mat4	mat4 frustum matrix will be written into
eye	ReadonlyVec3	Position of the viewer
center	ReadonlyVec3	Point the viewer is looking at
up	ReadonlyVec3	vec3 pointing up

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



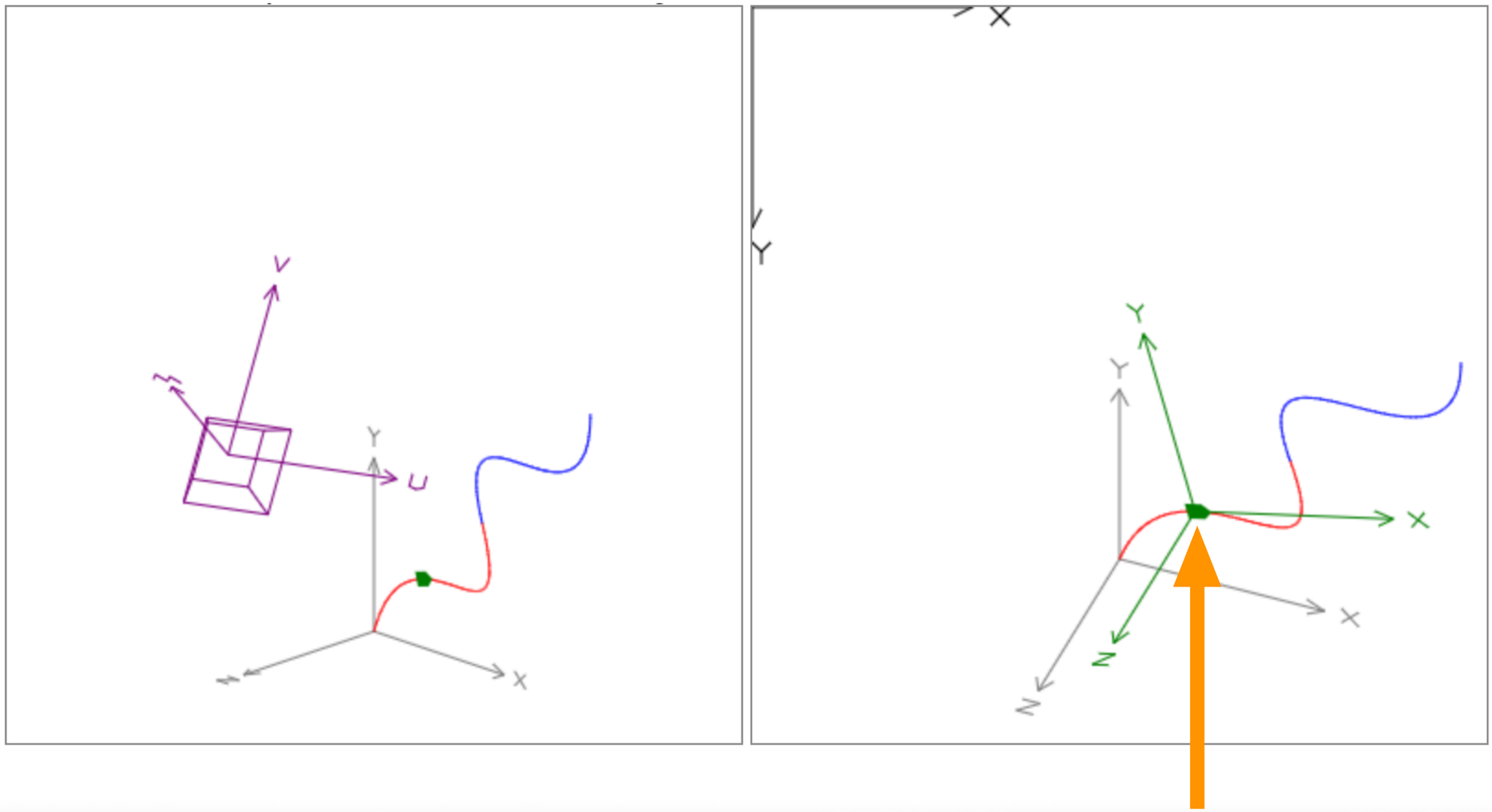
(A side observation …) these two visualizations have been created with a *different lookAt* transform! (the one of the camera used on the right; on the left, the location of the external observer was used to place a fictitious camera there!)

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

Model(ing) transform(s)

jsbin.com/ficoxeh

Week7/Demo4



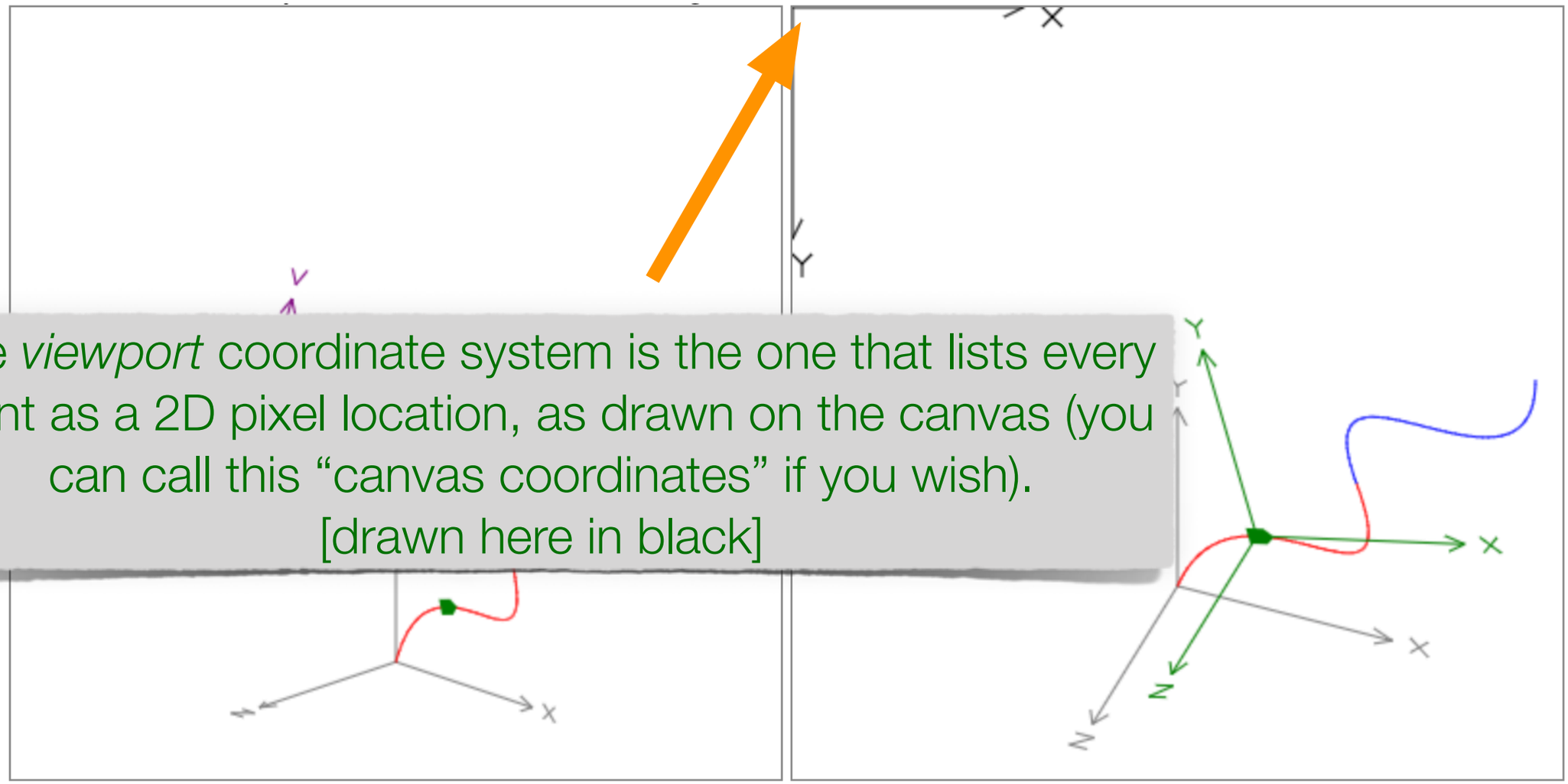
We can also have additional transforms that define coordinate systems (typically for moving objects, or displaced object instances) relative to the world coordinates. (Shown here in green …)

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

Canvas/Viewport coordinates

jsbin.com/ficoxeh

Week7/Demo4



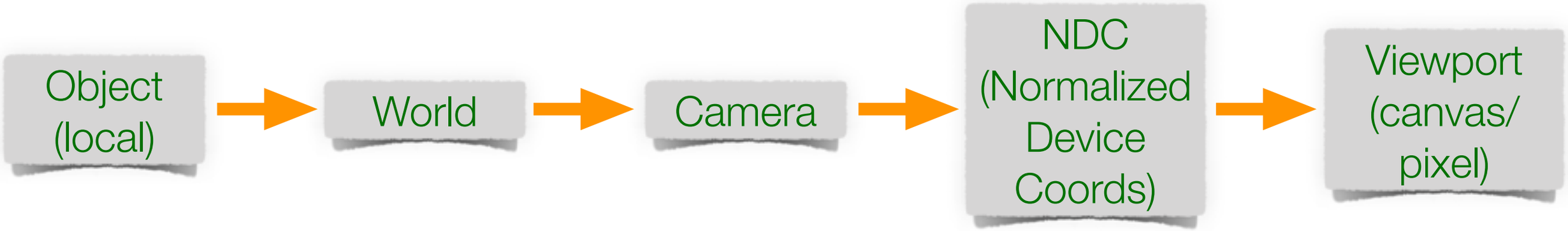
The *viewport* coordinate system is the one that lists every point as a 2D pixel location, as drawn on the canvas (you can call this "canvas coordinates" if you wish).
[drawn here in black]

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

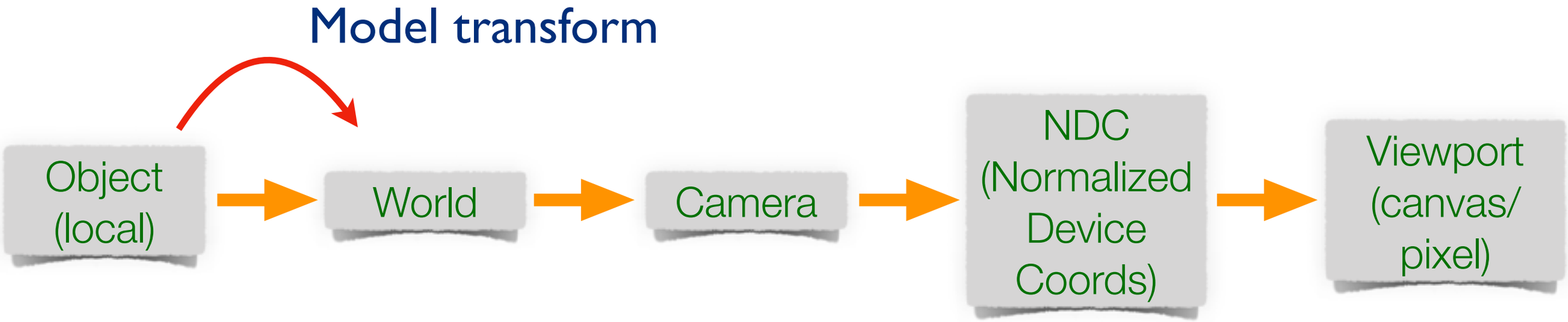
Week7/Demo4



A (viewing) transform pipeline

jsbin.com/ficoxeh

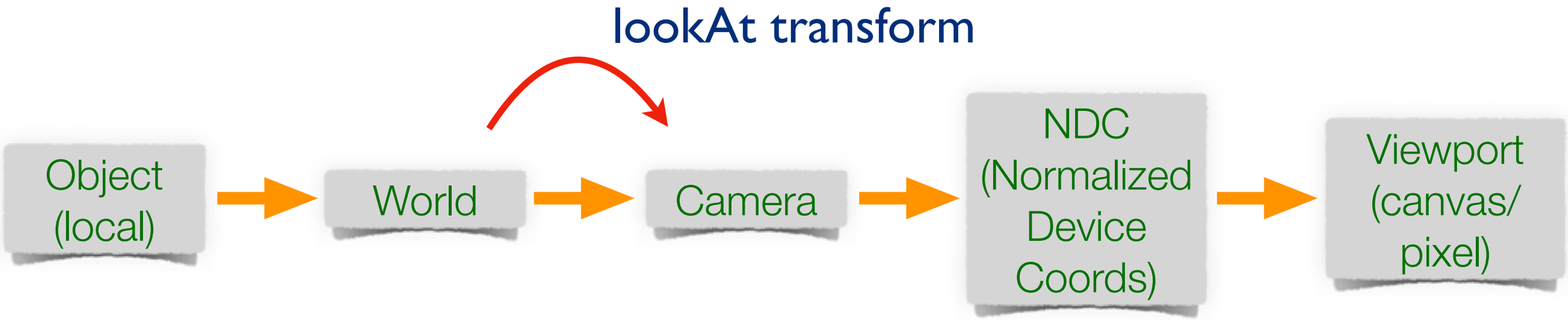
Week7/Demo4



A (viewing) transform pipeline

jsbin.com/ficoxeh

Week7/Demo4



A (viewing) transform pipeline

jsbin.com/ficoxeh

Week7/Demo4

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