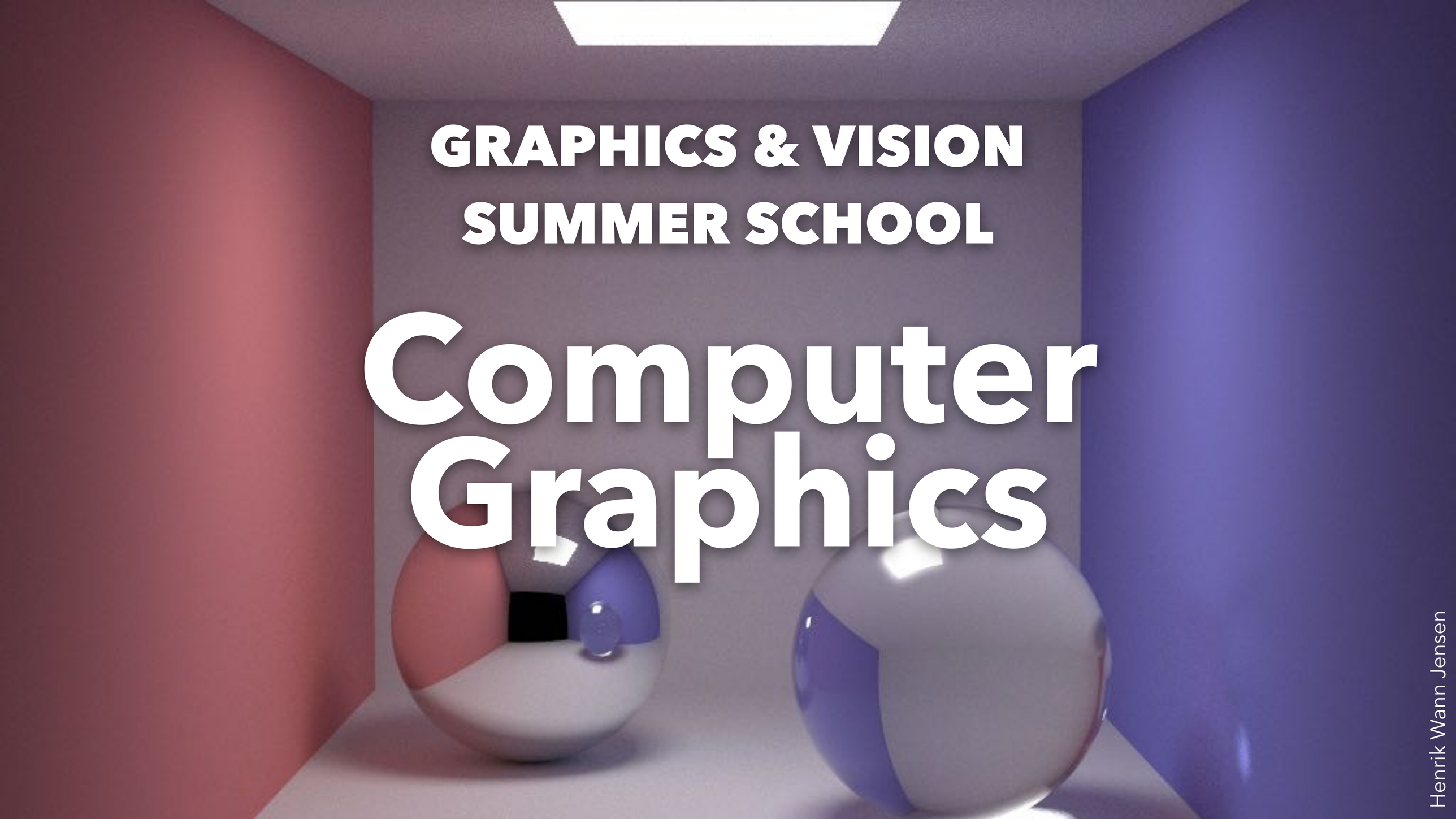
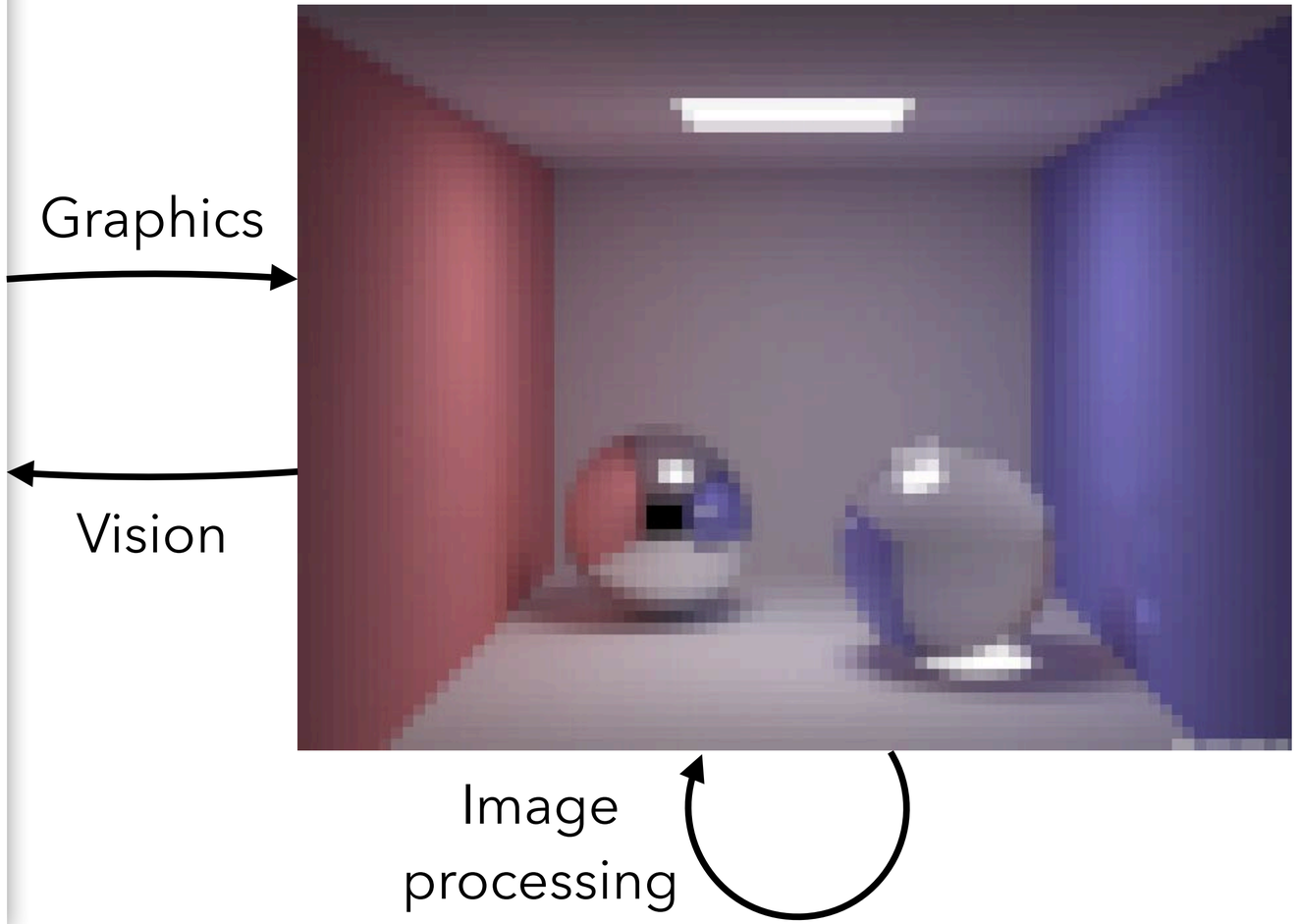
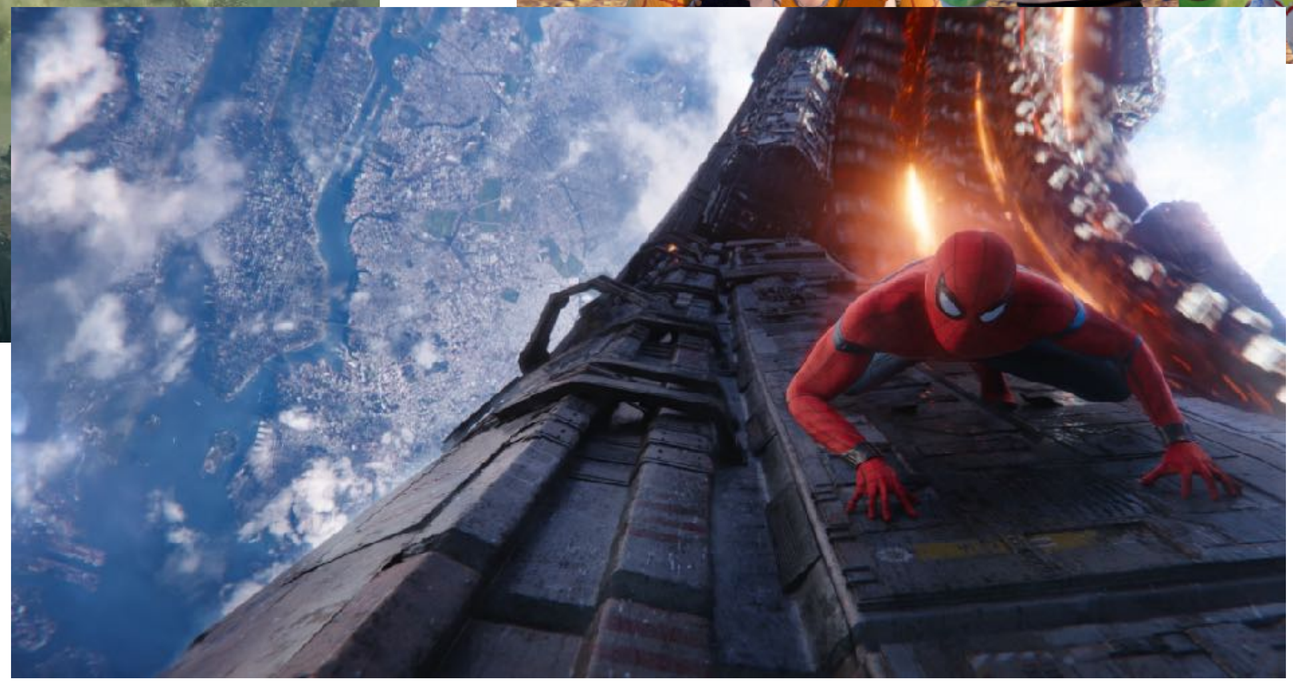

GRAPHICS & VISION
SUMMER SCHOOL
Computer
Graphics
Henrik Wann Jensen

```
light:
  shape: box [-0.5,0.5] …
  intensity: rgb (10,10,10)
sphere:
  center: (-0.5,0.5,-0.5)
  radius: 0.5
  material: mirror
sphere:
  center: (0.5,0.5,0.5)
  radius: 0.5
  material: glass
wall:
  x: -1
  colour: rgb (0.8,0.2,0.2)
…
```

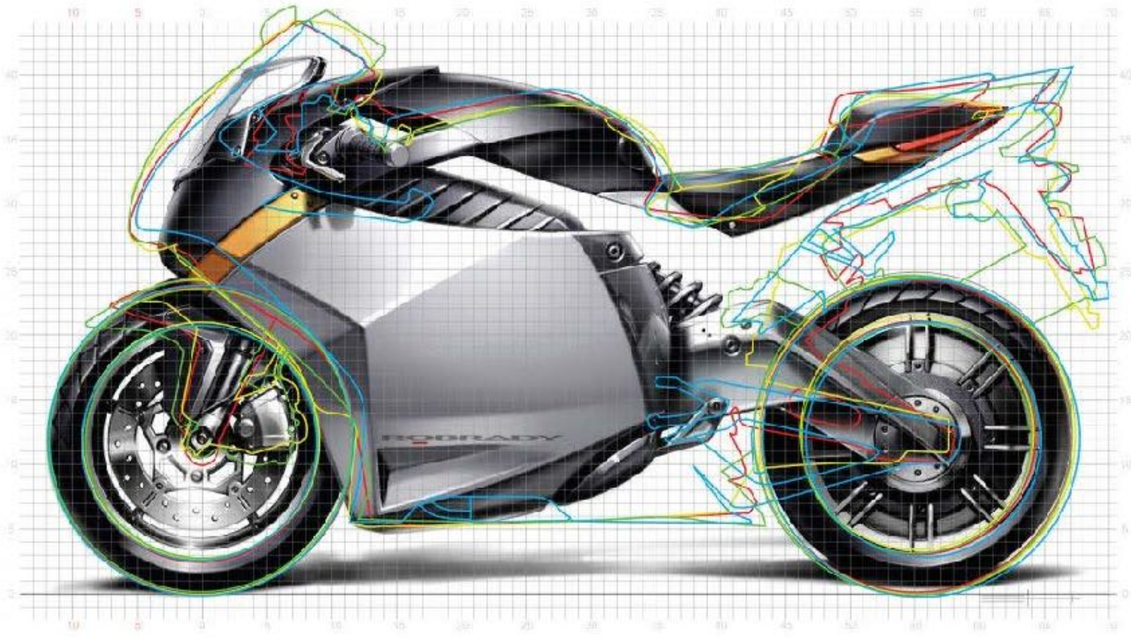


Entertainment



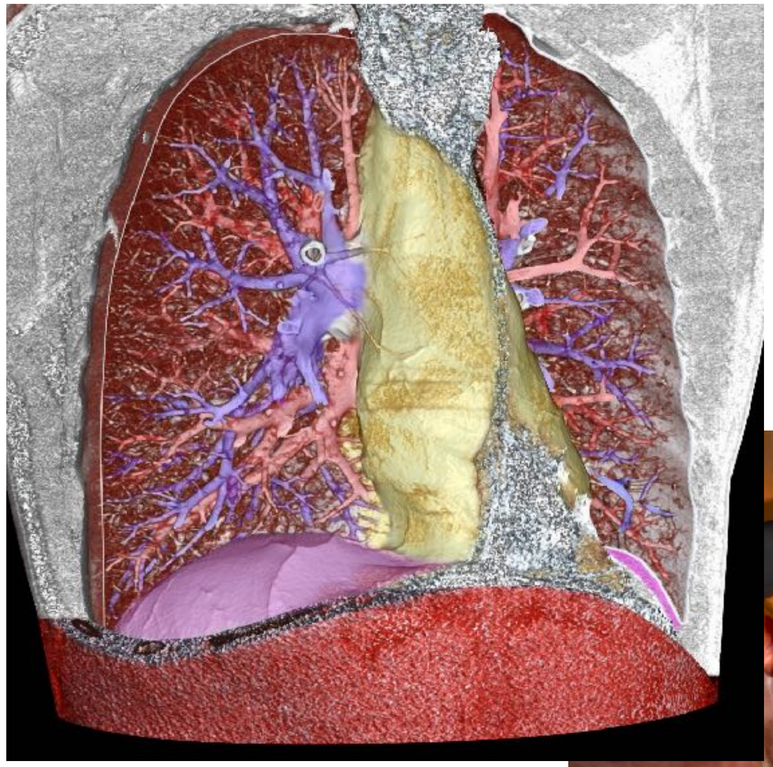
Art, design, architecture

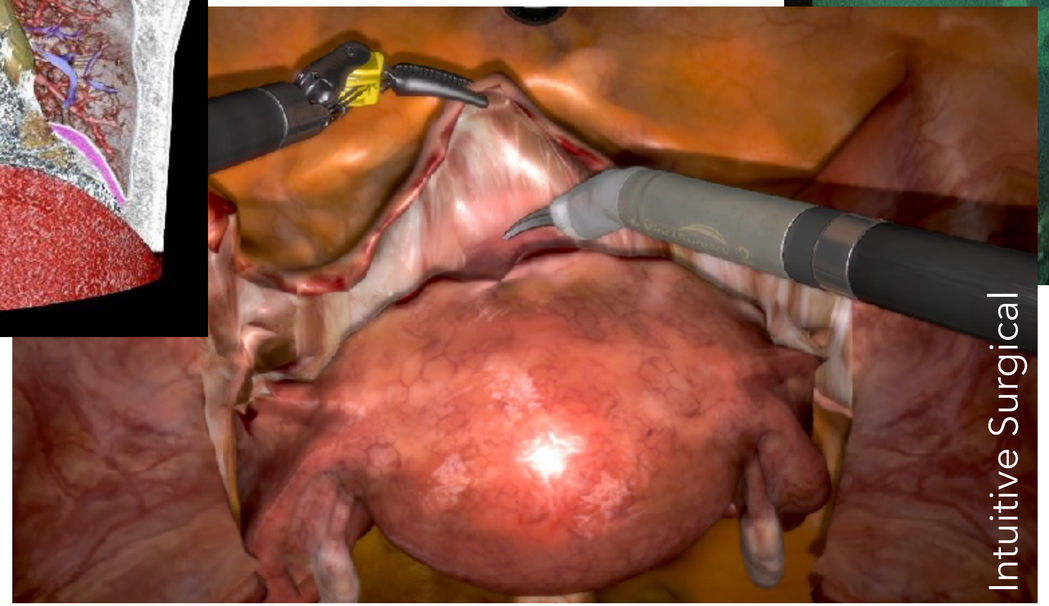

Bathsheba Grossman



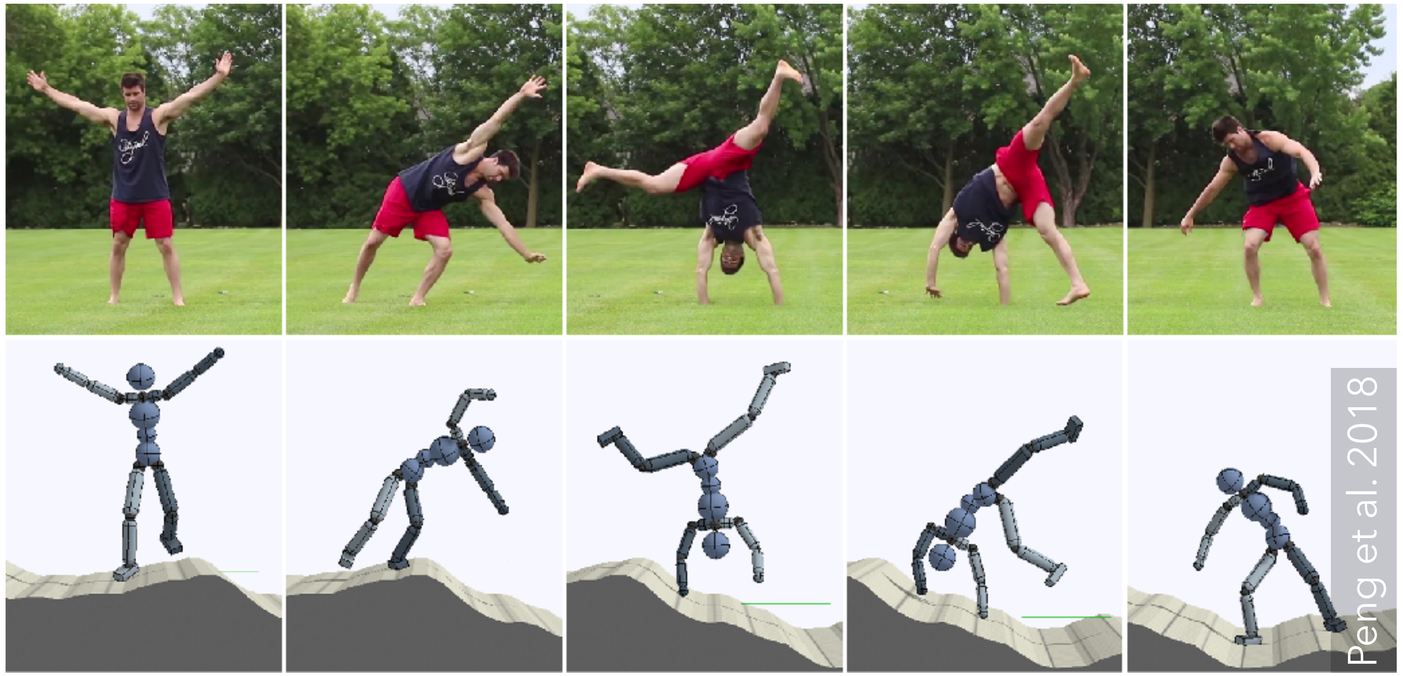

Sage Gateshead

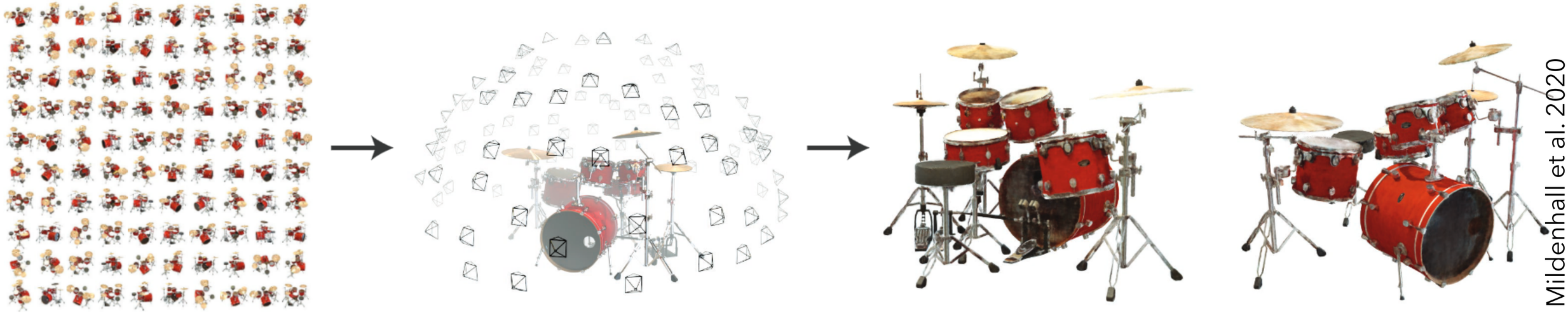
Scientific visualization and training





Inverse graphics

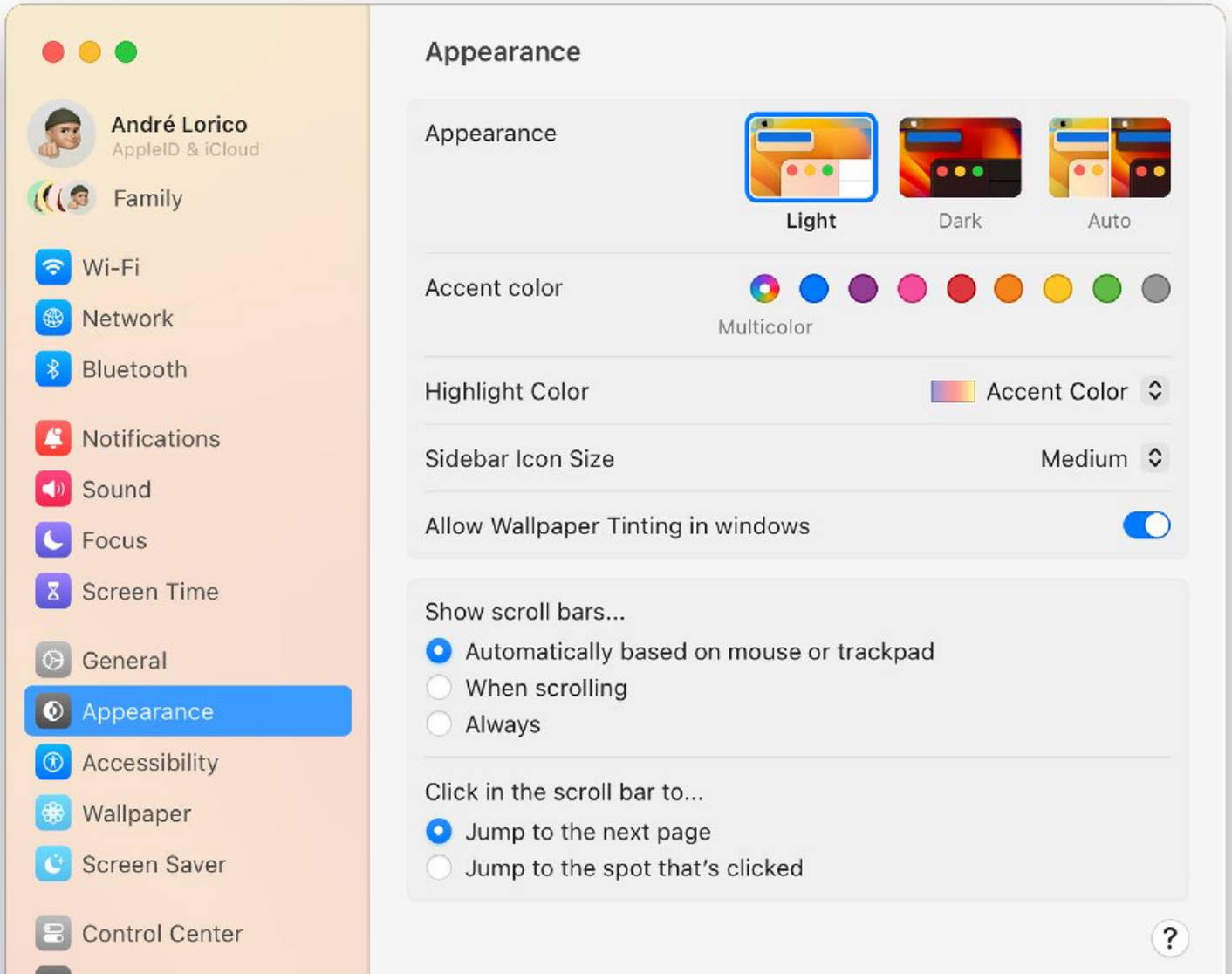




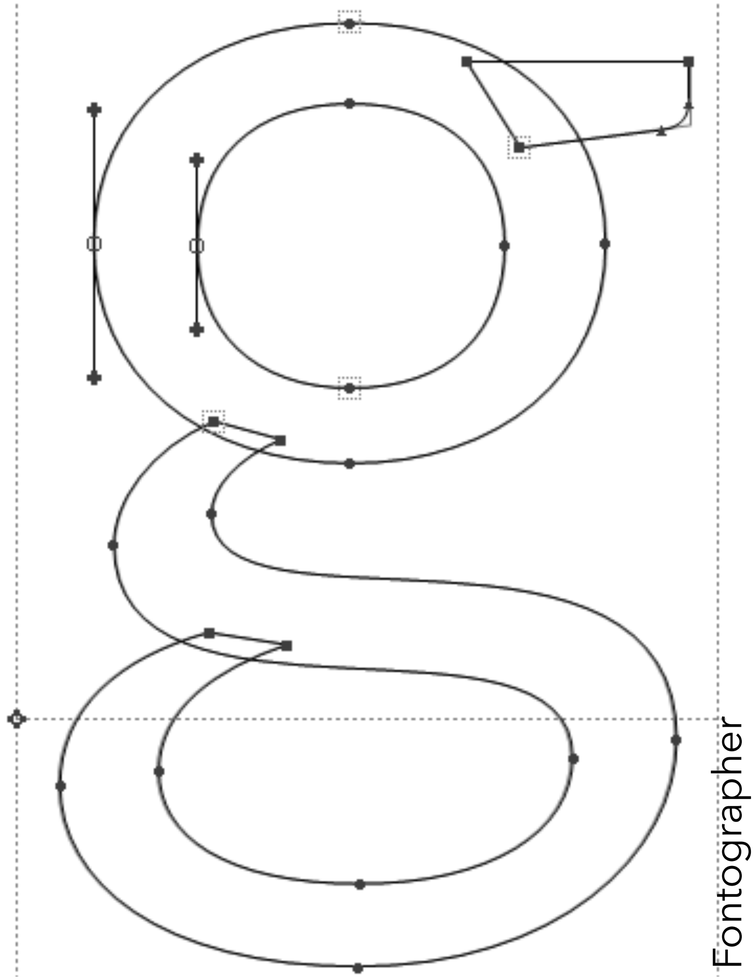
Actually, computer graphics is **omnipresent** in how all of us interact with computers today!

…Wait, how?

Graphical user interfaces, typography

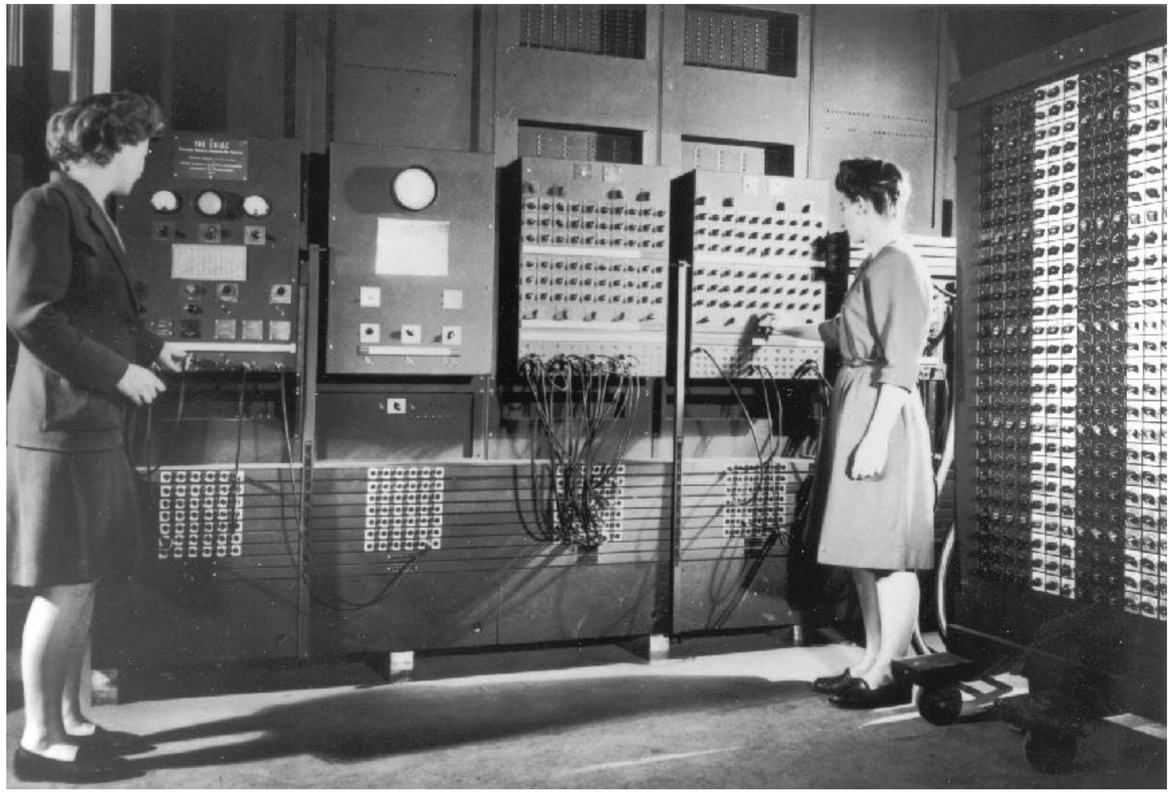


Apple

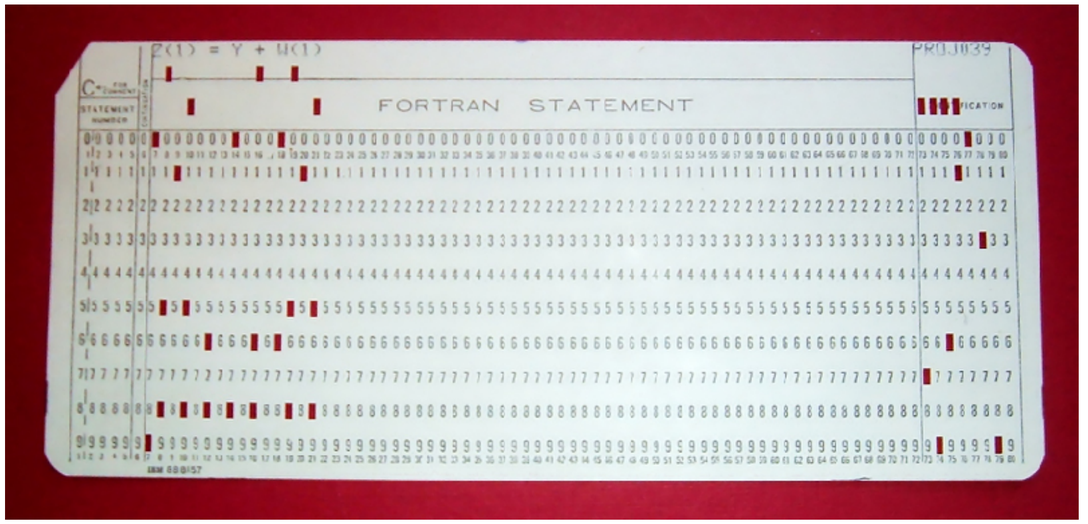


Fontographer

Computing without graphics



ENIAC (1945)

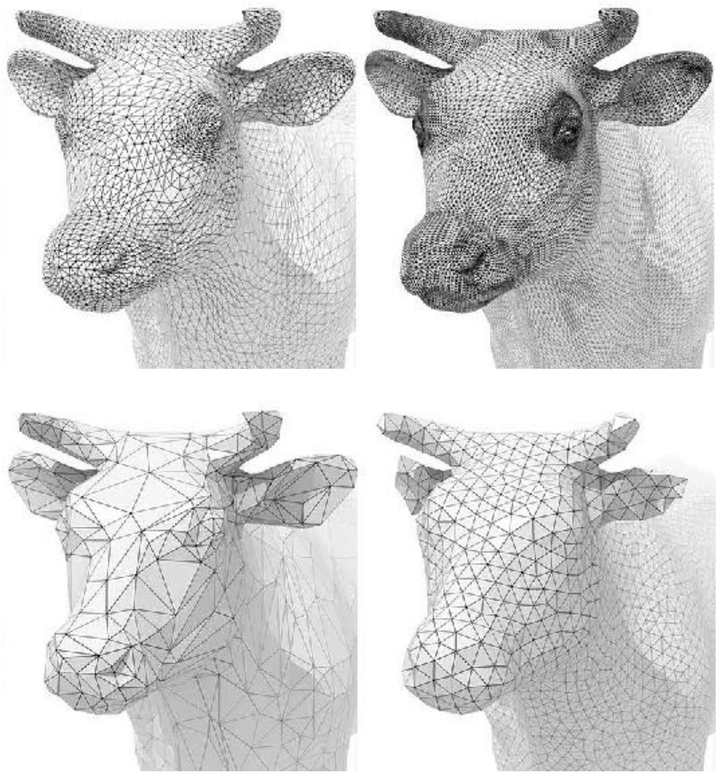


Punched card from a
Fortran program

Computing without graphics

```
[root@localhost ~]# ping -q fa.wikipedia.org
PING text.pmtpa.wikimedia.org (208.80.152.2) 56(84) bytes of data.
^C
--- text.pmtpa.wikimedia.org ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.000/0.000/0.000/0.000 ms
[root@localhost ~]# pwd
/root
[root@localhost ~]# cd /var
[root@localhost var]# ls -la
total 72
drwxr-xr-x. 18 root root 4096 Jul 30 22:43 .
drwxr-xr-x. 23 root root 4096 Sep 14 20:42 ..
drwxr-xr-x.  2 root root 4096 May 14 00:15 account
drwxr-xr-x. 11 root root 4096 Jul 31 22:26 cache
drwxr-xr-x.  3 root root 4096 May 18 16:03 db
drwxr-xr-x.  3 root root 4096 May 18 16:03 empty
drwxr-xr-x.  2 root root 4096 May 18 16:03 games
drwxrwx--T.  2 root gdm  4096 Jun  2 18:39 gdm
drwxr-xr-x. 38 root root 4096 May 18 16:03 lib
drwxr-xr-x.  2 root root 4096 May 18 16:03 local
lrwxrwxrwx.  1 root root   11 May 14 00:12 lock -> ../run/lock
drwxr-xr-x. 14 root root 4096 Sep 14 20:42 log
lrwxrwxrwx.  1 root root   10 Jul 30 22:43 mail -> spool/mail
drwxr-xr-x.  2 root root 4096 May 18 16:03 nis
drwxr-xr-x.  2 root root 4096 May 18 16:03 opt
drwxr-xr-x.  2 root root 4096 May 18 16:03 preserve
drwxr-xr-x.  2 root root 4096 Jul  1 22:11 report
lrwxrwxrwx.  1 root root    6 May 14 00:12 run -> ../run
drwxr-xr-x. 14 root root 4096 May 18 16:03 spool
drwxrwxrwt.  4 root root 4096 Sep 12 23:50 tmp
drwxr-xr-x.  2 root root 4096 May 18 16:03 yp
[root@localhost var]# yum search wiki
Loaded plugins: langpacks, presto, refresh-packagekit, remove-with-leaves
rpmfusion-free-updates                                   | 2.7 kB     00:00
rpmfusion-free-updates/primary_db                        | 206 kB     00:04
rpmfusion-nonfree-updates                                | 2.7 kB     00:00
updates/metalink                                         | 5.9 kB     00:00
updates                                                  | 4.7 kB     00:00
updates/primary_db                 73% [=================          ]  62 kB/s | 2.6 MB     00:15 ETA
```

Aspects of computer graphics



Modelling

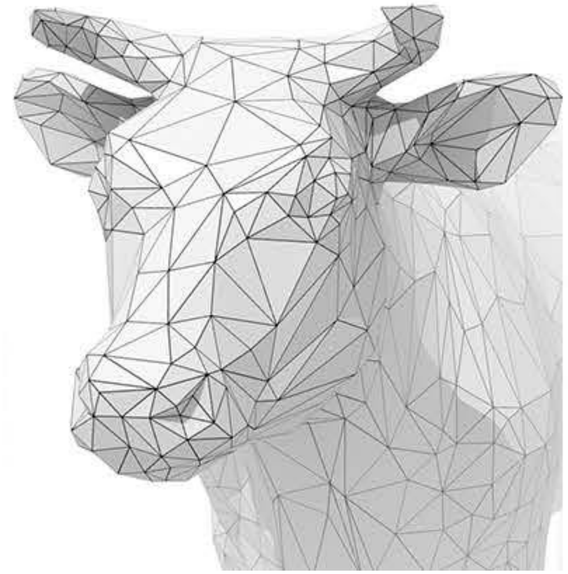
Rendering

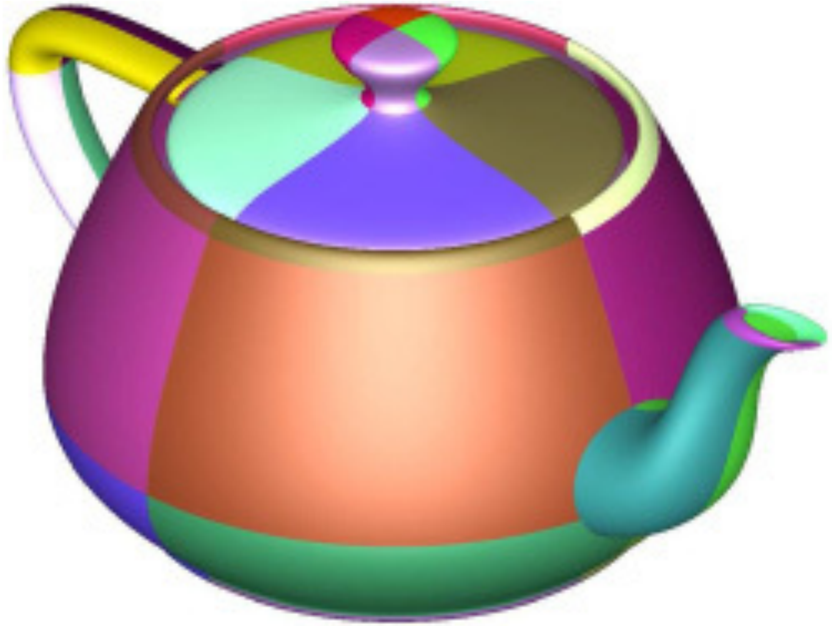
Animation

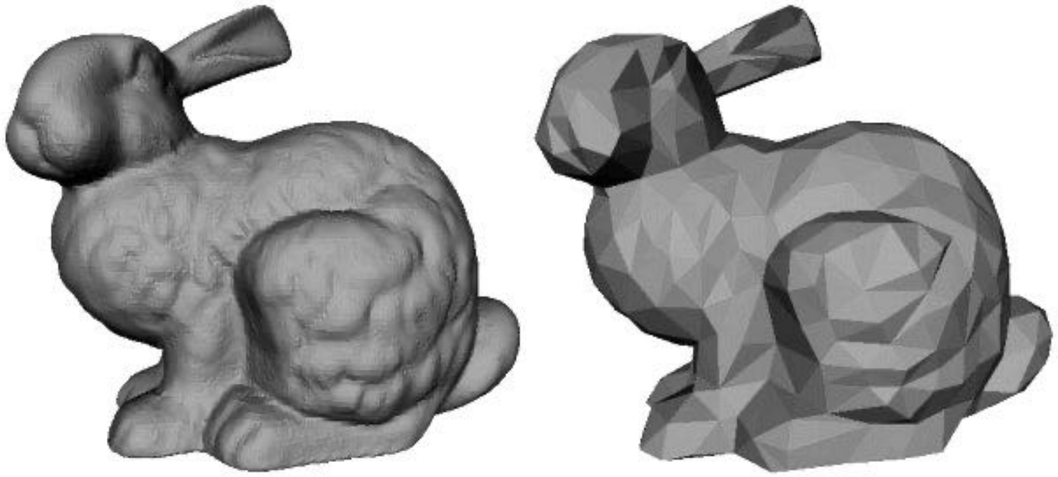
Modelling

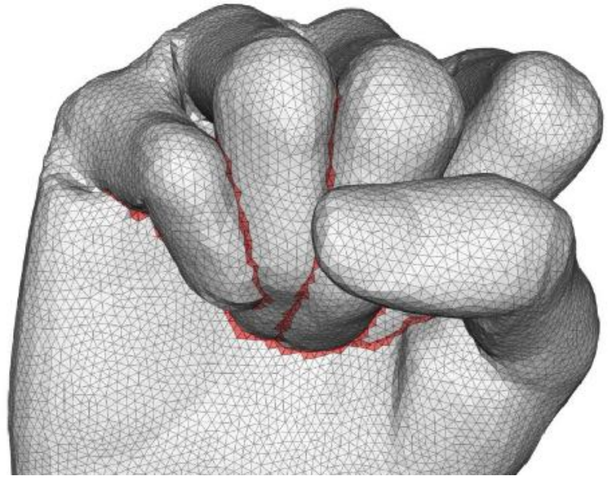
How to work with geometry?

- Representation
- Manipulation and editing
- Geometric queries



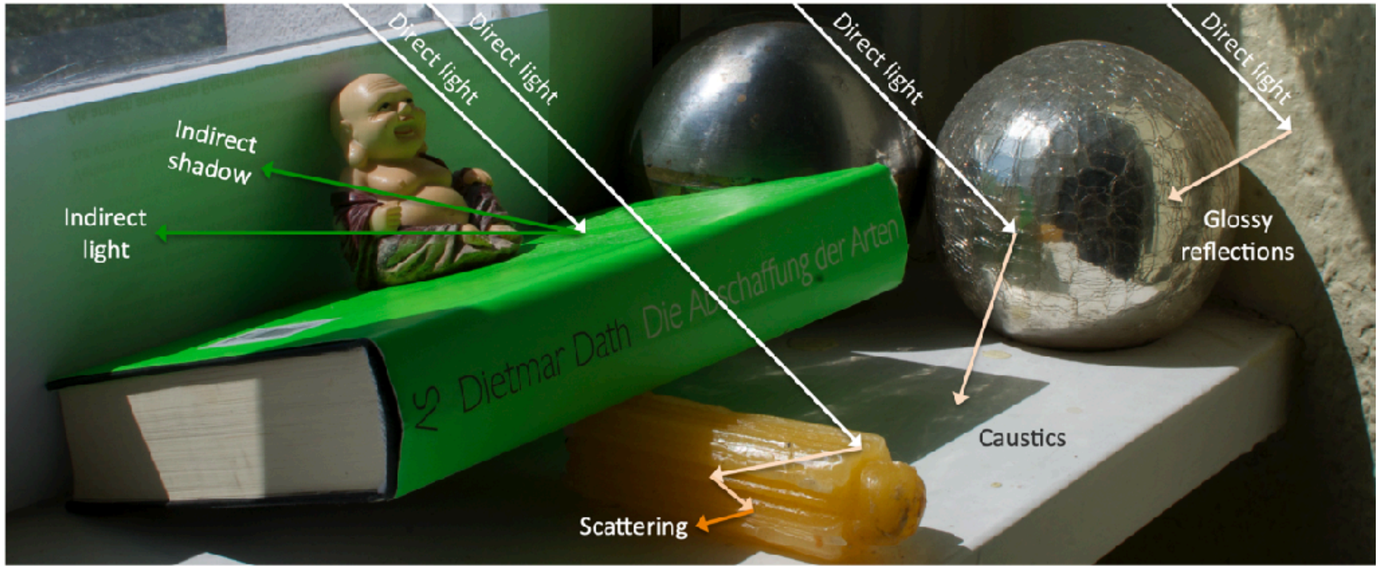






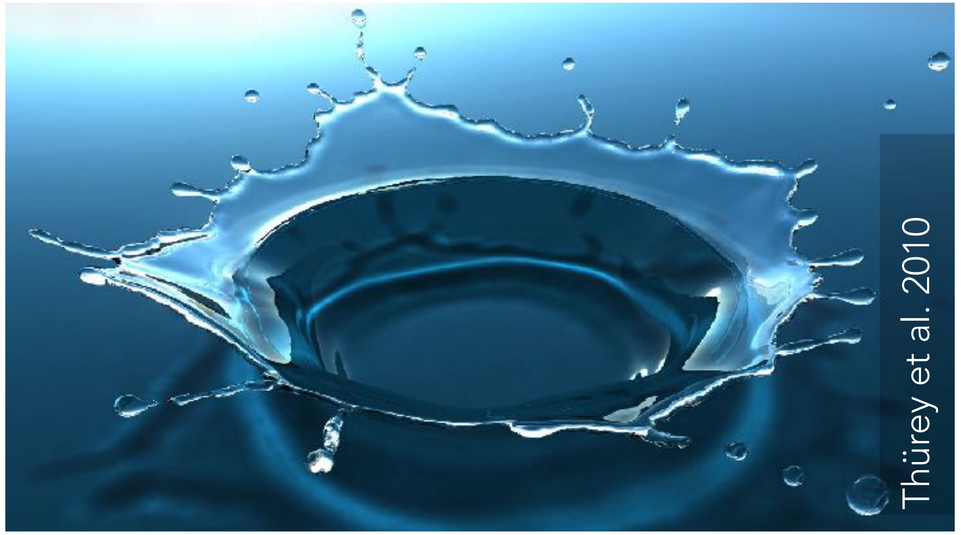
Rendering

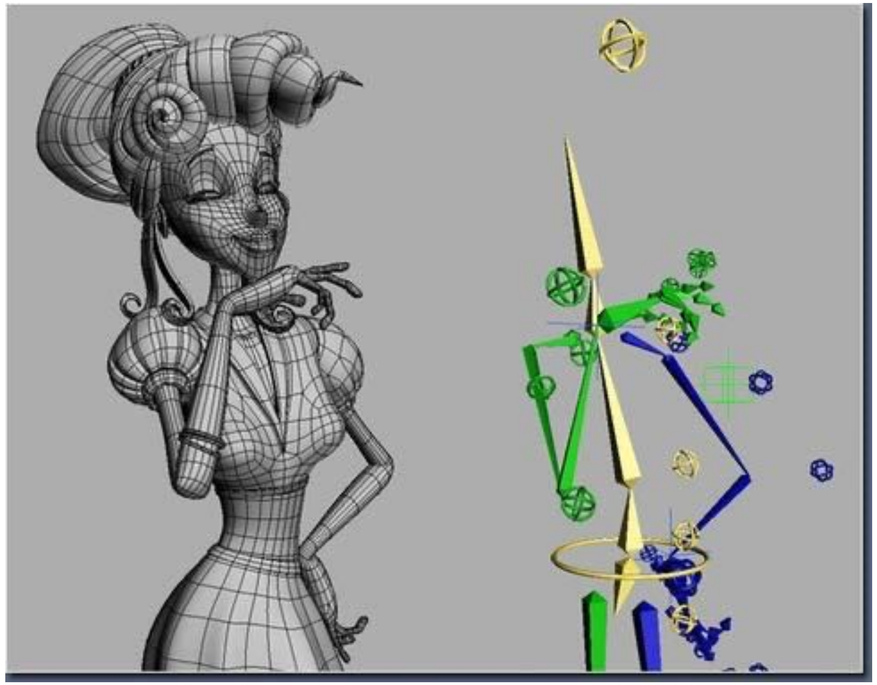
- Quantifying light and materials
- Computing light transport in a scene
- Real-time approximations

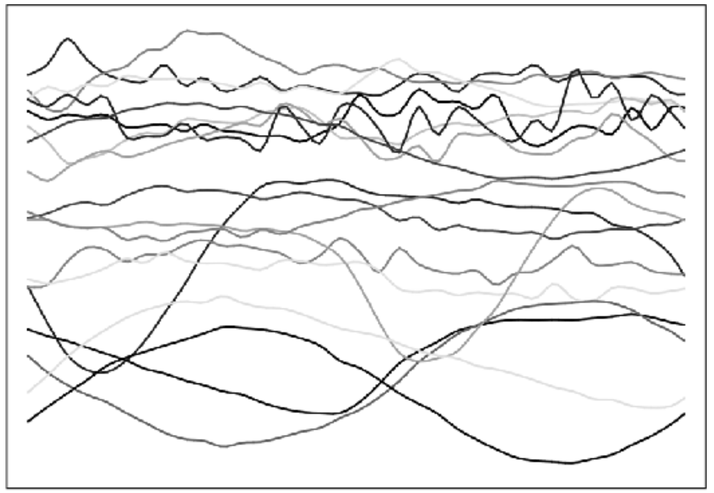


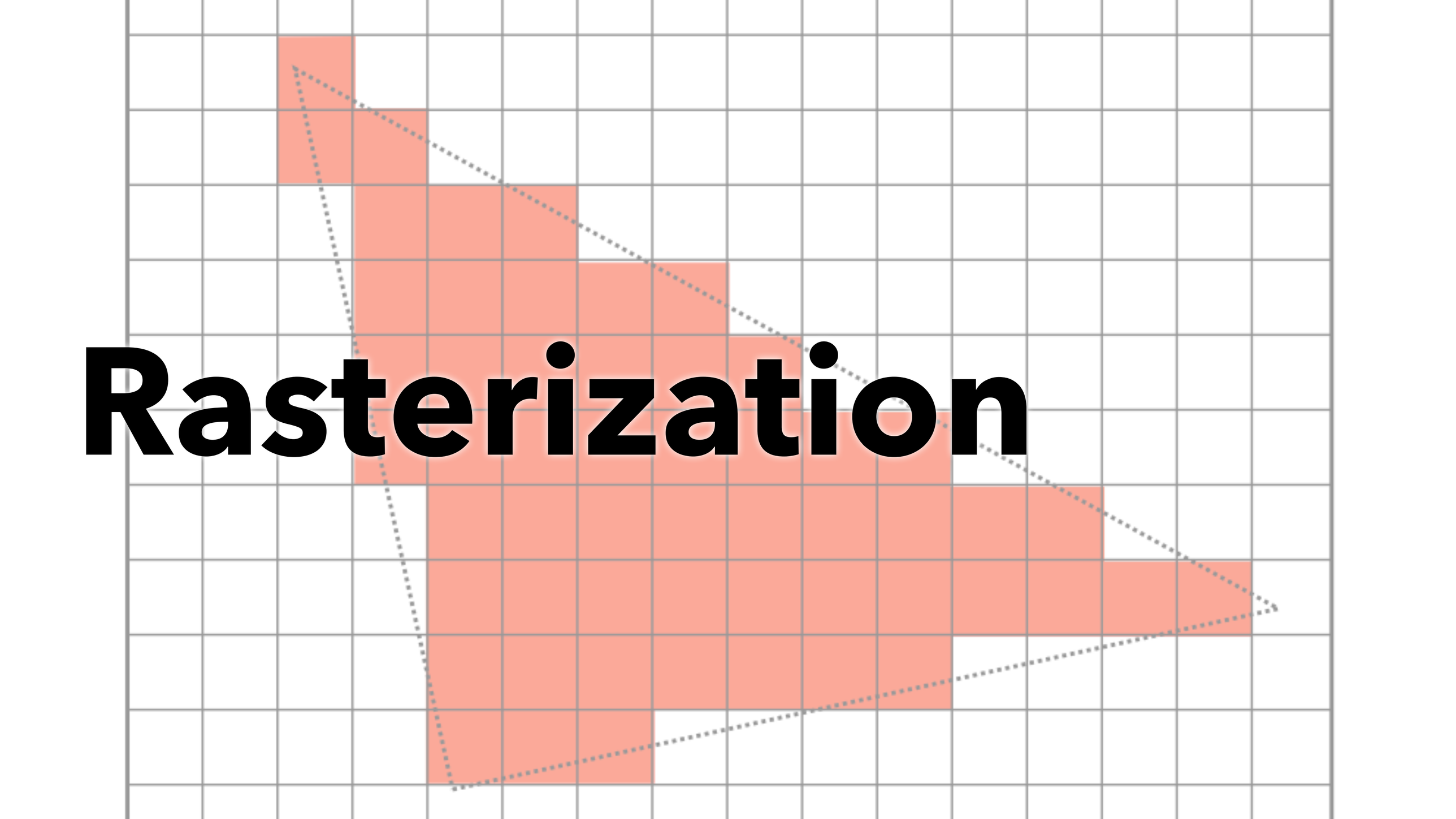
Animation

- Character animation
- Physics-based animation








Rasterization

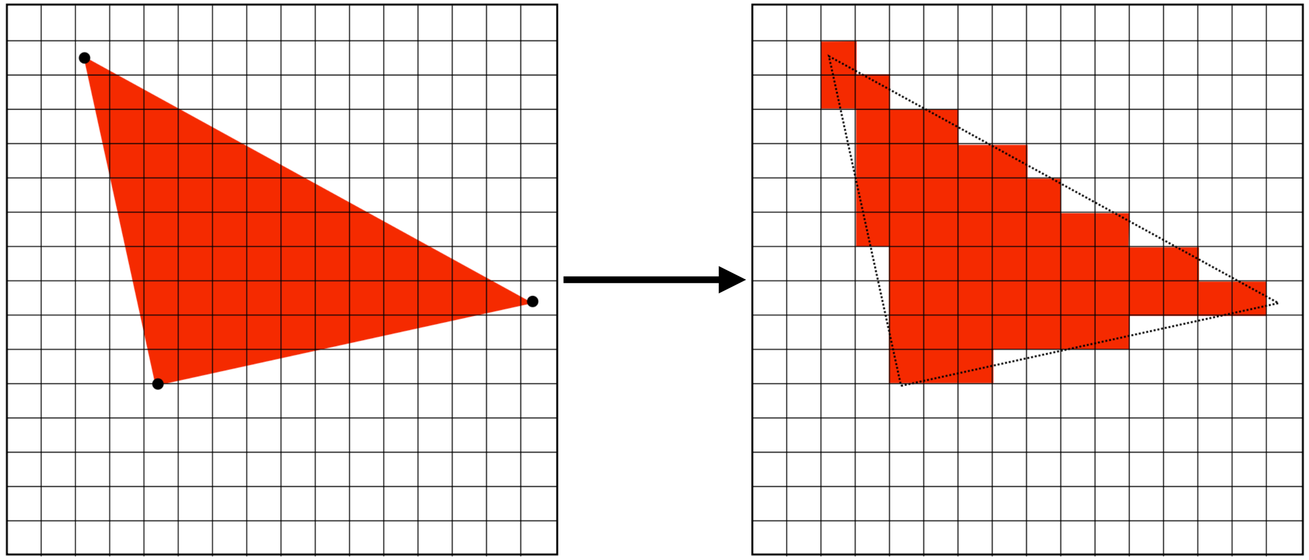
Rasterization

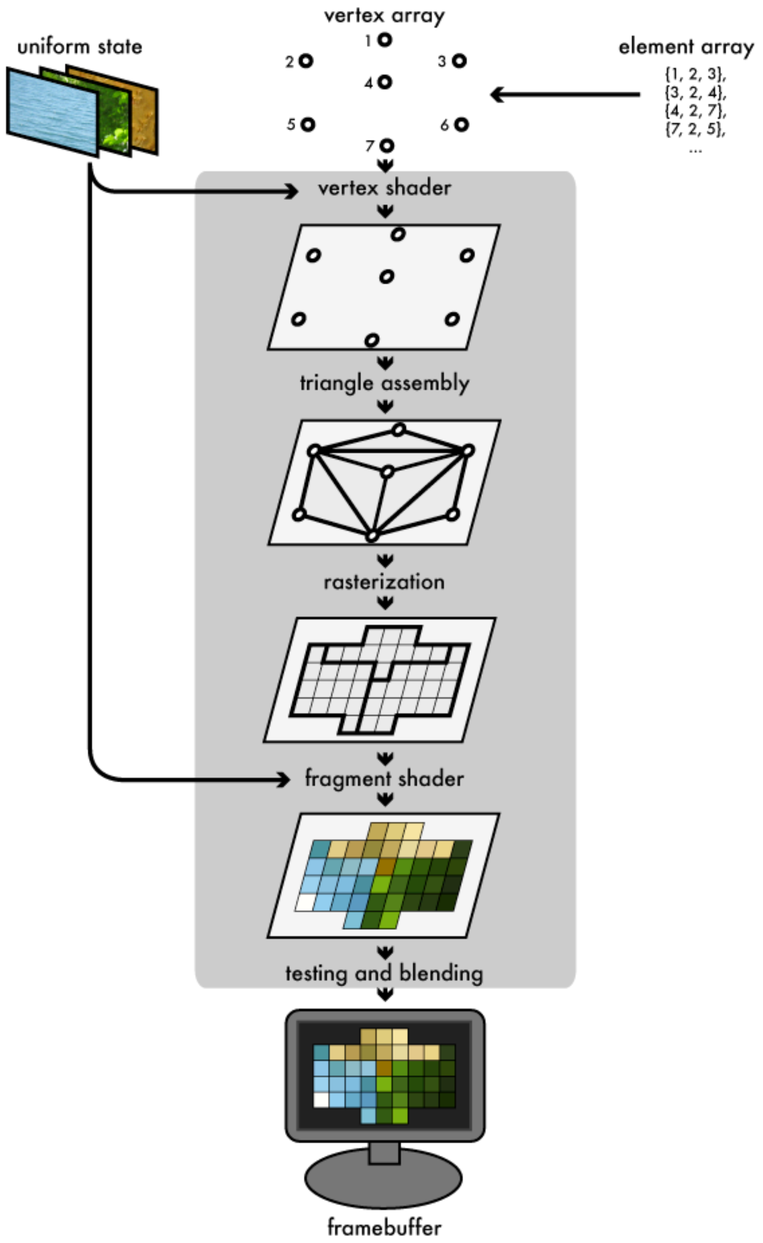
To display any 2D or 3D shape on a pixel display, it needs to be **rasterized**!

Input: Geometrical “primitives” (usually triangles) with attributes (e.g. colour)

Output: Raster image approx-imating the given shape

Usually this is performed by the **graphics processing unit** (GPU)





Preview: The (real-time) graphics pipeline

Even for 3D graphics,

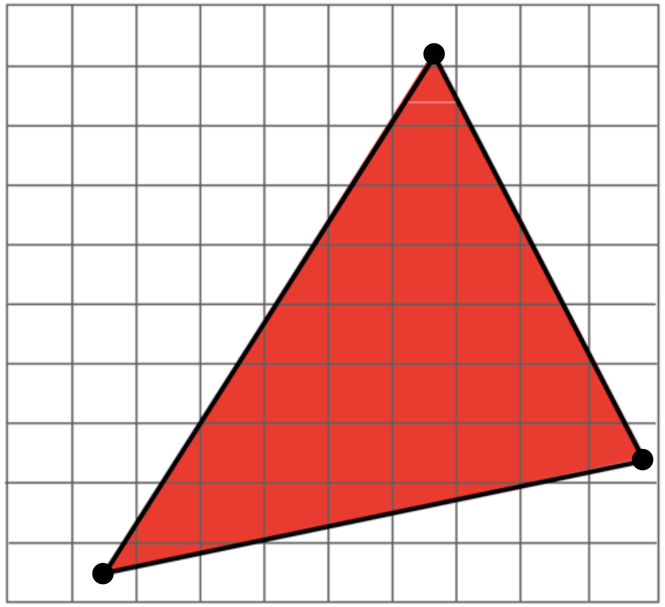
1. first we project the vertices of each 3D triangle to their 2D locations on the screen

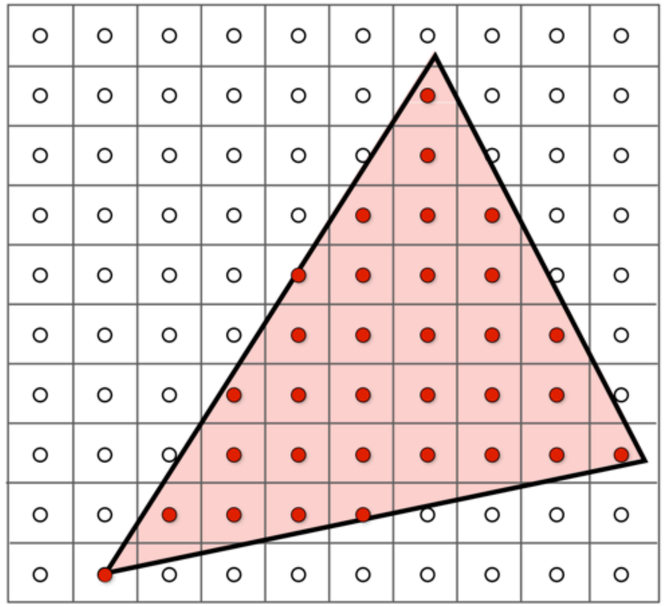
2. then we rasterize the 2D triangle!

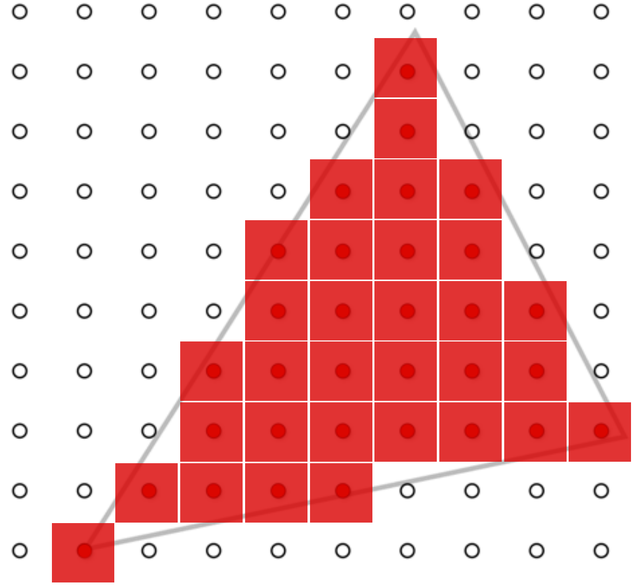
So it makes sense to study rasterization of 2D graphics first.

How to draw an arbitrary triangle on a pixel grid?

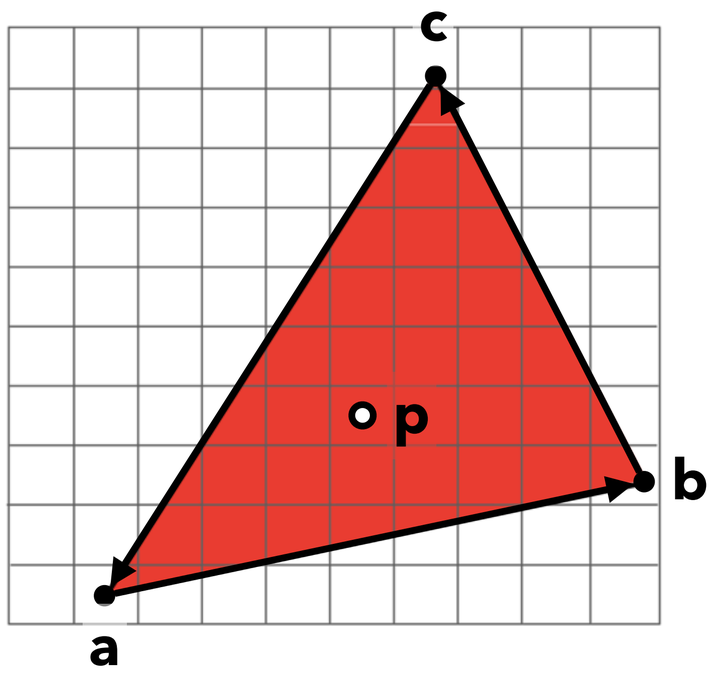
For now, let's pick a sample point at the center of each pixel, and colour the pixel if the sample point lies inside the triangle.





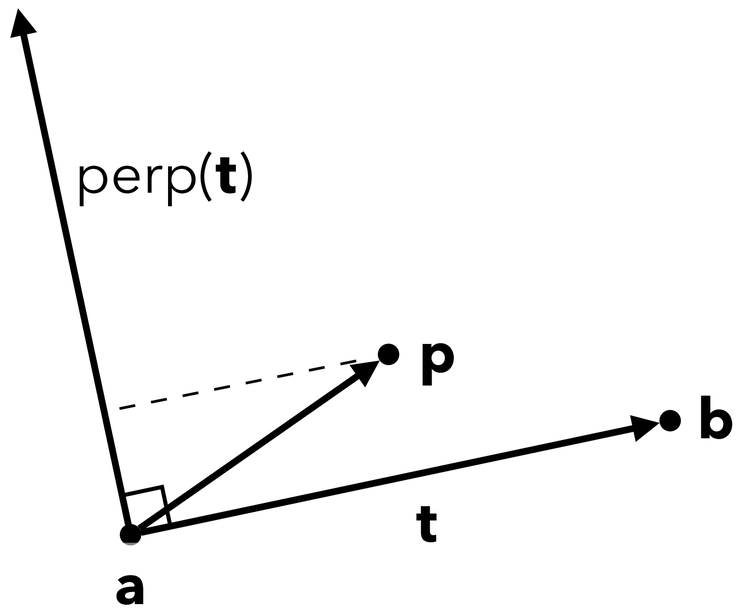


How to check whether a point is inside a triangle?



A point **p** is inside triangle **abc** if:

- **p** is to the left of edge **ab**, and
- **p** is to the left of edge **bc**, and
- **p** is to the left of edge **ca**.



Edge tangent vector:

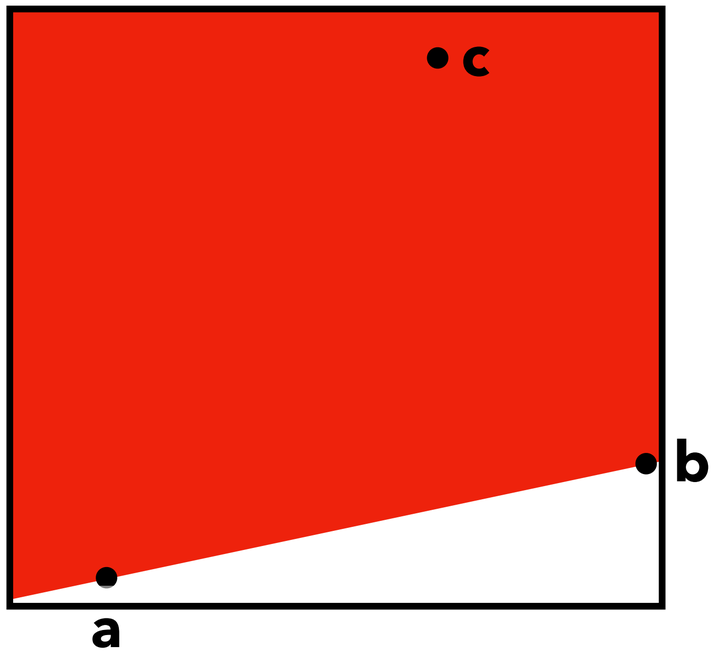
$$\mathbf{t} = \mathbf{b} - \mathbf{a} = (b_x - a_x, b_y - a_y)$$

Edge “normal” vector:

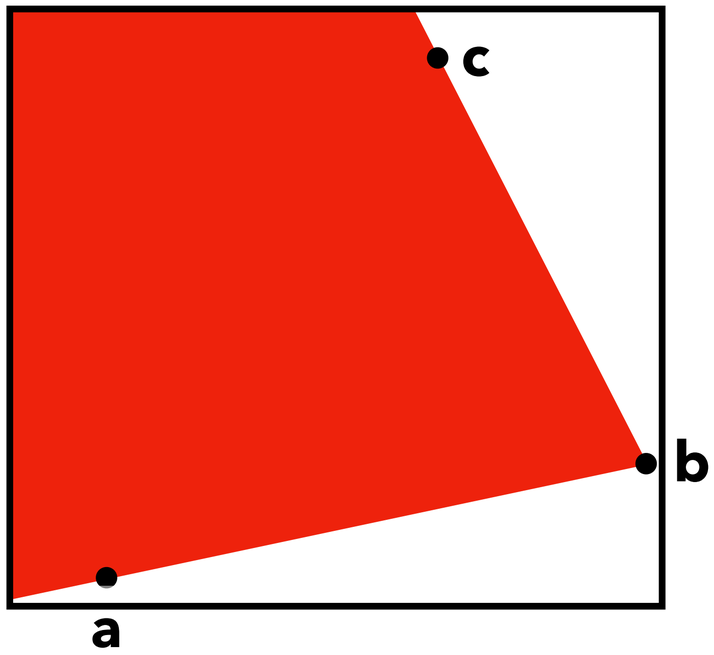
$$\mathbf{n} = \text{perp}(\mathbf{t}) = (-t_y, t_x)$$

p is to the left of **ab** if

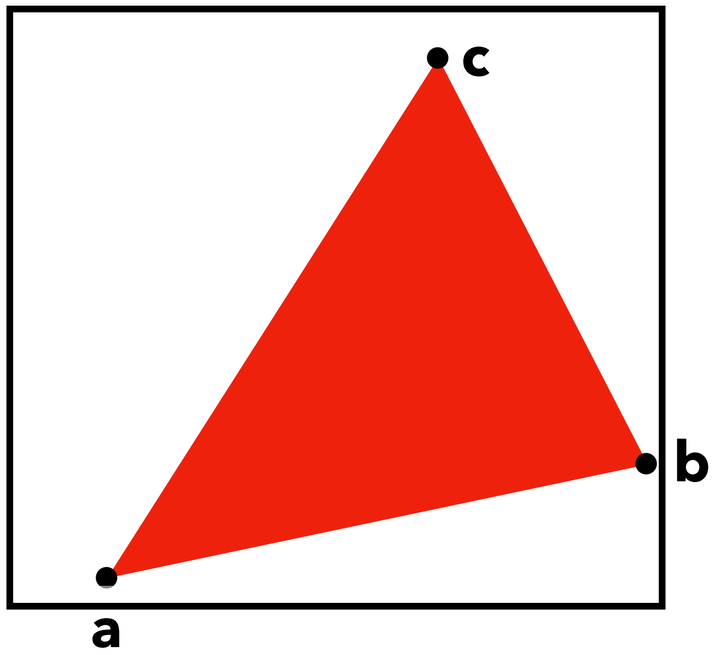
$$\mathbf{n} \cdot (\mathbf{p} - \mathbf{a}) \geq 0$$



Points to the left of **ab**

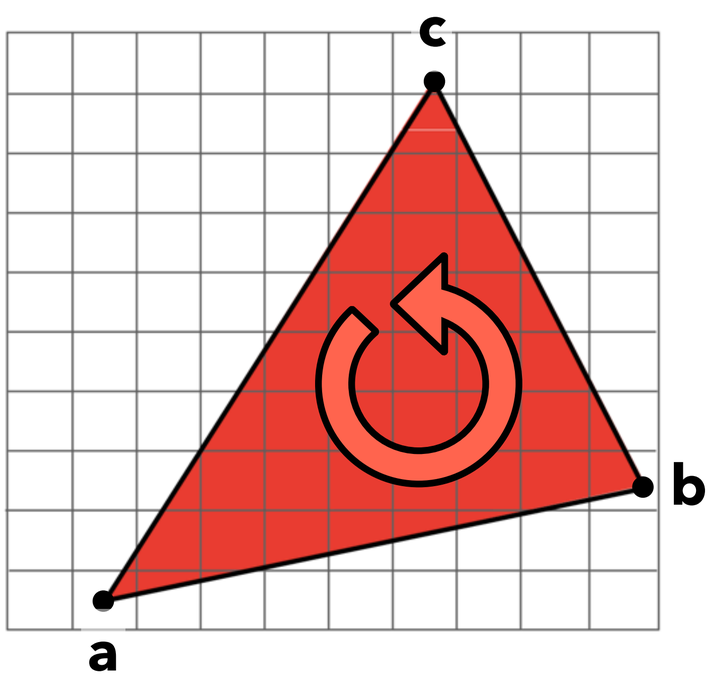


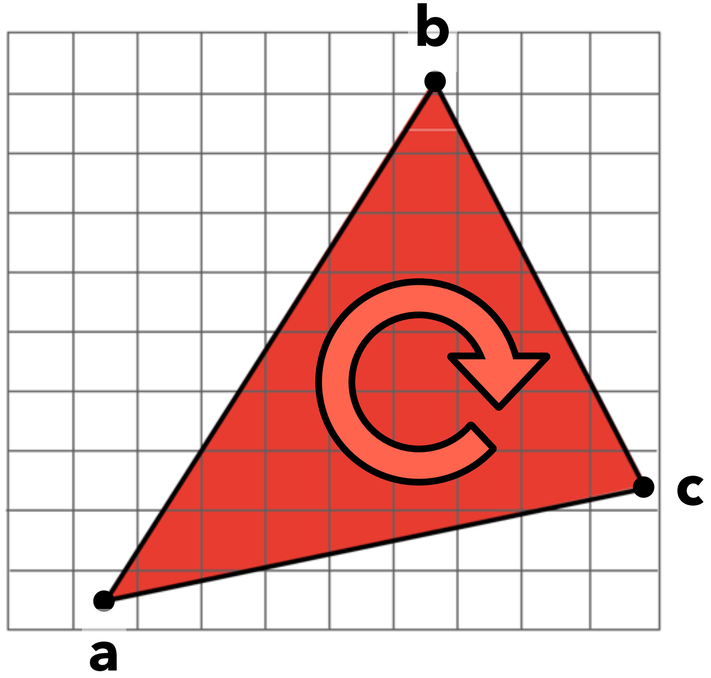
Points to the left of **ab**
and to the left of **bc**



Points to the left of **ab**
and to the left of **bc**
and to the left of **ca**

Would this still work if the vertices were given in **clockwise** order instead?





Easy to fix: First check if **c** is to the left or the right of **ab**.
But, better if you ensure all triangles are anticlockwise in the first place.

So, here's what our rasterization algorithm looks like so far.

```
drawTriangle(triangle, colour):
  for x = 0 ... imageWidth:
    for y = 0 ... imageHeight:
      if isInside(x, y, triangle):
        image[x, y] = colour
```

Is this an efficient algorithm?

How can we make it faster?

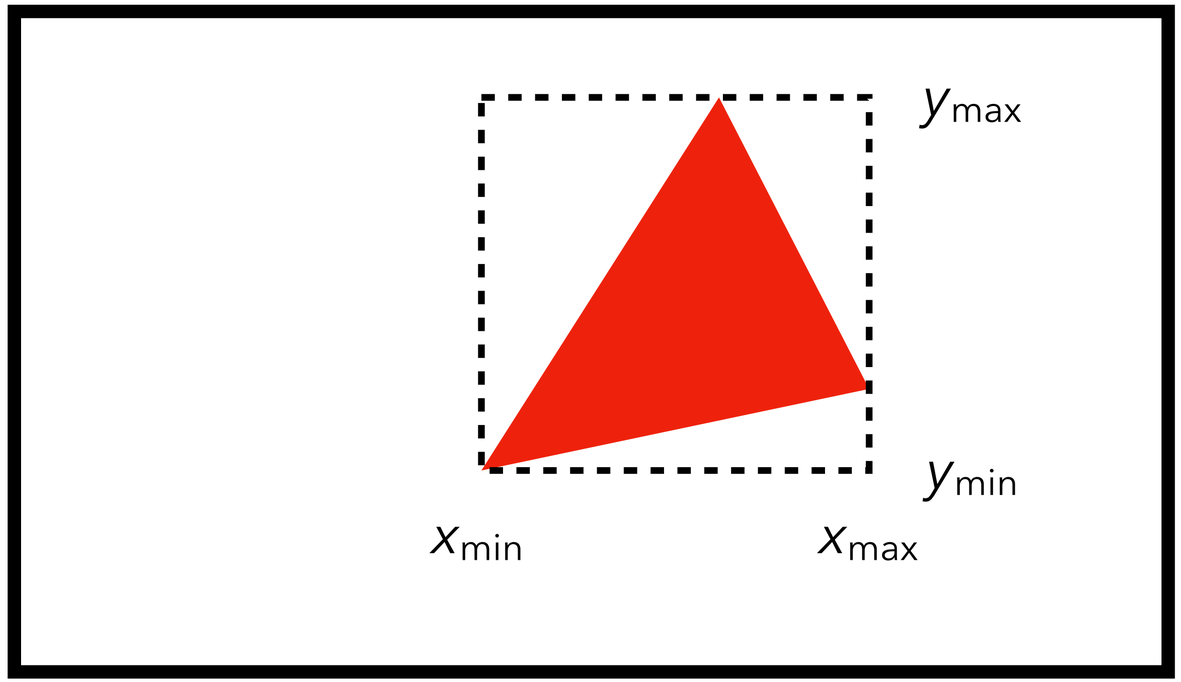
Better to only check the pixels in the **bounding box** of the triangle.

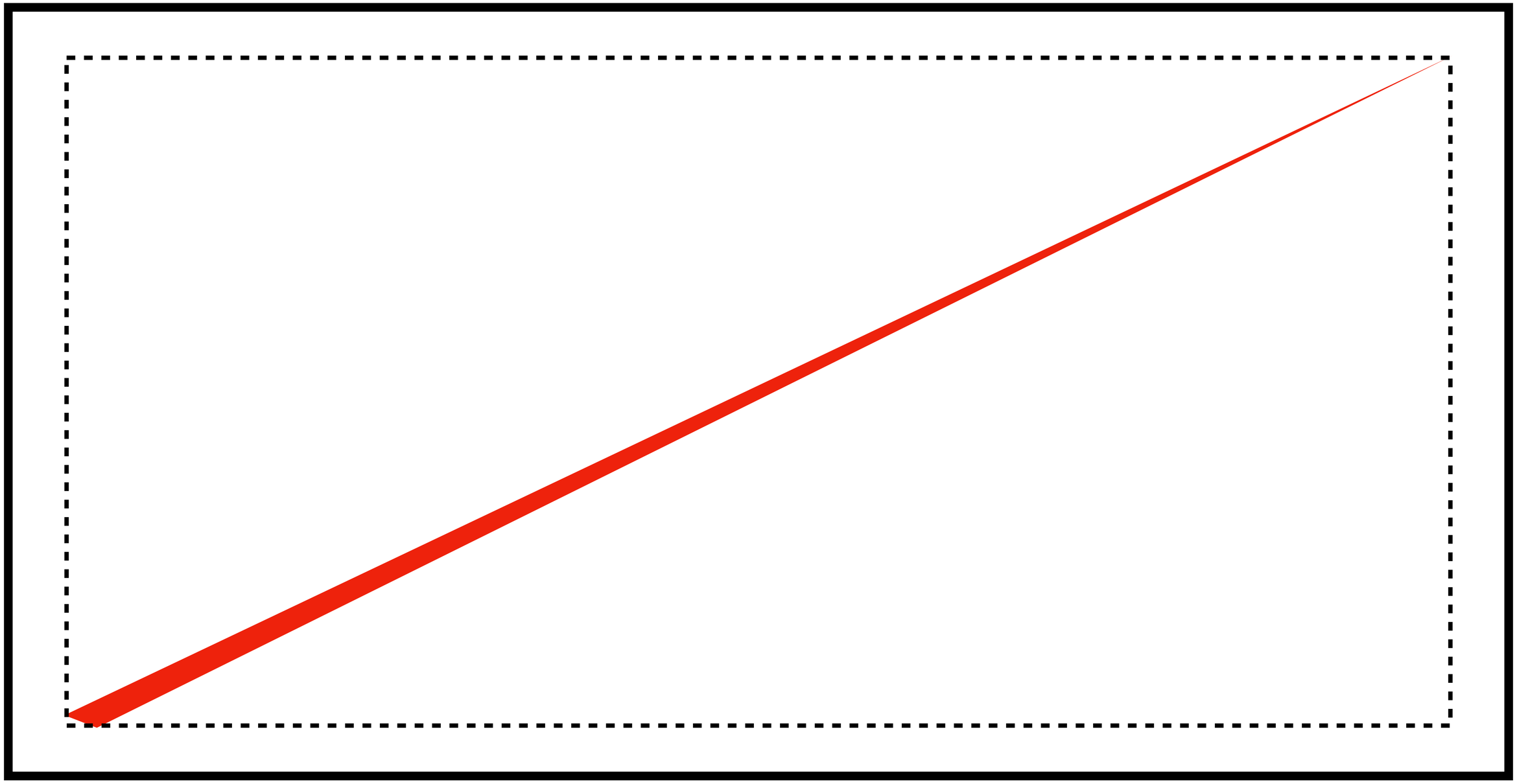
What are the coordinates of this box?

$x_{\text{min}} = \min\{a_x, b_x, c_x\}$,

…

Are there any cases where this is also terribly inefficient?

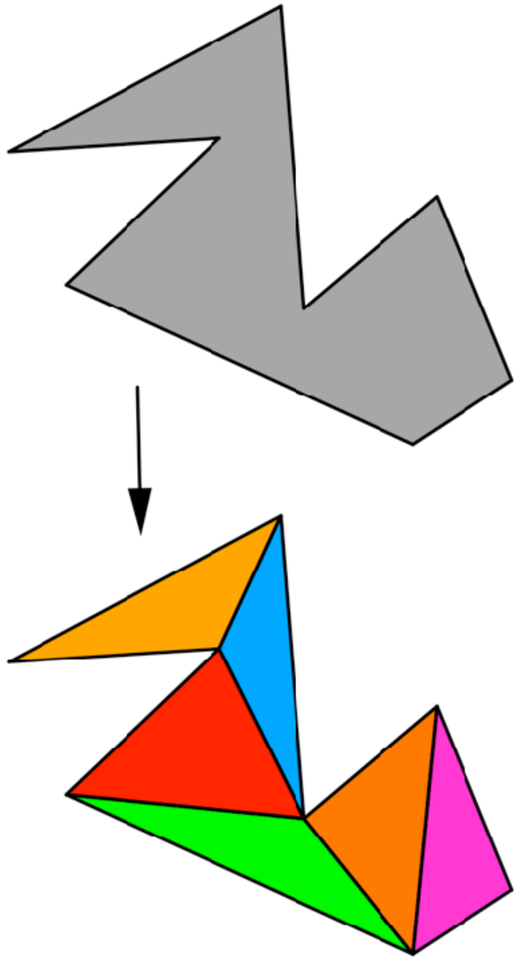




What about more complex shapes?

Split them up into triangles, then draw each triangle

for each *triangle*:
    for each sample (*x*, *y*) it covers:
        image[*x*, *y*] = *triangle*.colour(*x*, *y*)



Something to think about: What if we did this instead?

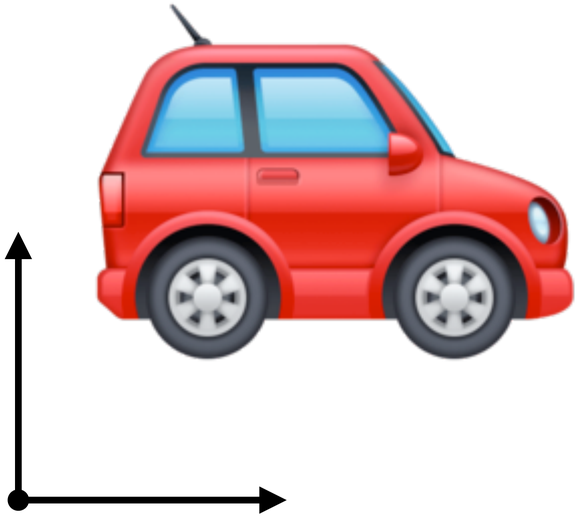
for each sample (*x*, *y*):
    for each *shape* that covers it:
        image[*x*, *y*] = *shape*.colour(*x*, *y*)

Transformations

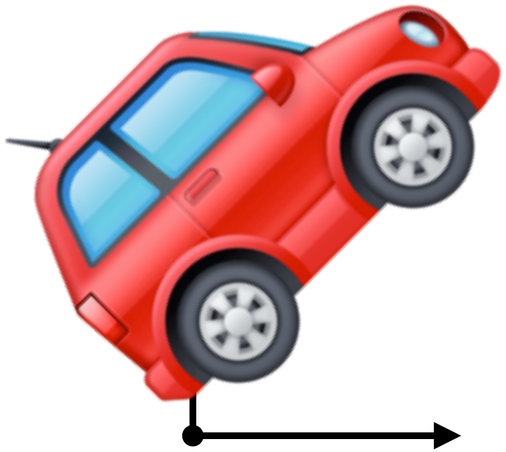
Matthew Wagner

Transformations

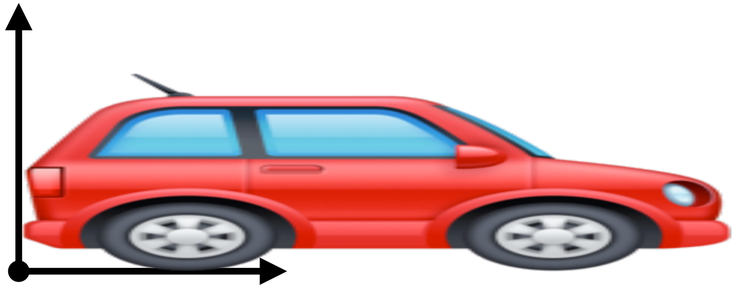
Transformations



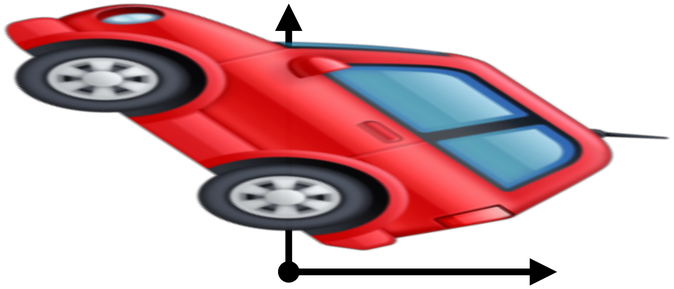
Translation



Rotation



Scaling



???

Applications: Instancing

Star Wars: Episode II – Attack of the Clones (2002)

Applications: Posing

Third Shift Vintage

Applications: Viewing

Cristian Goga

Transformation matrices

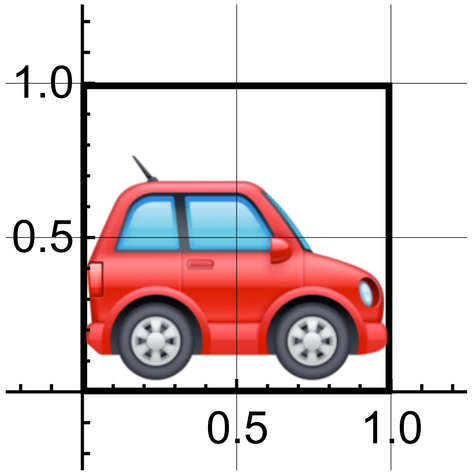
As you probably know, we can represent many transformations by matrices:

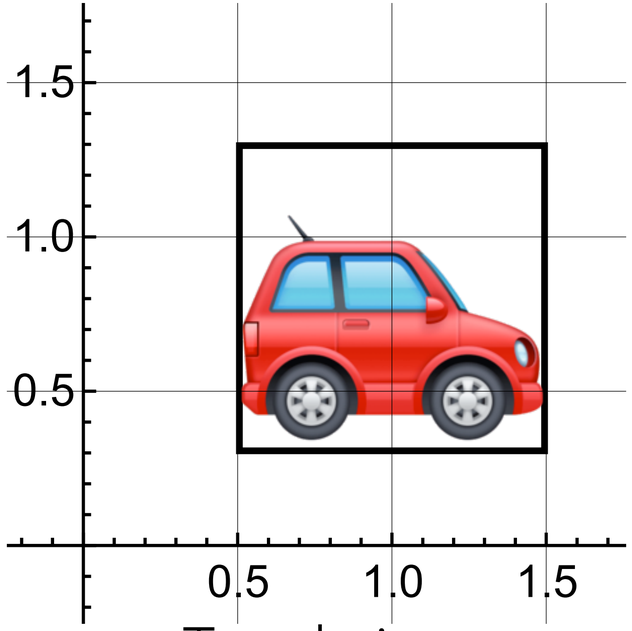
$$\mathbf{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix} \qquad \mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad \mathbf{Av} = \begin{bmatrix} a_{11}v_x + a_{12}v_y \\ a_{21}v_x + a_{22}v_y \end{bmatrix}$$

and similarly in 3D:

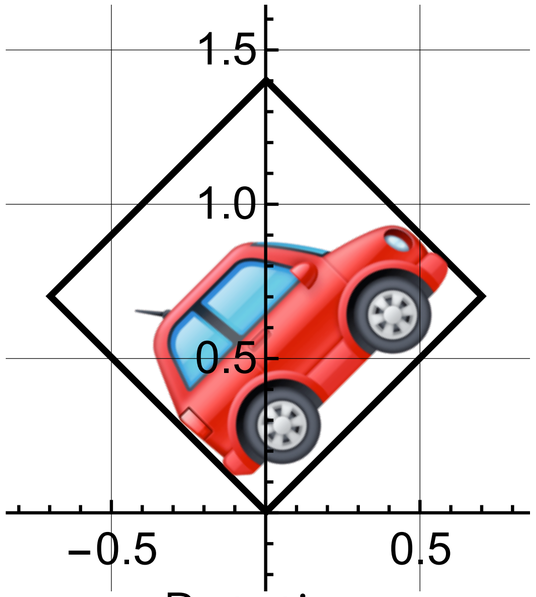
$$\mathbf{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \qquad \mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \qquad \mathbf{Av} = \begin{bmatrix} a_{11}v_x + a_{12}v_y + a_{13}v_z \\ a_{21}v_x + a_{22}v_y + a_{23}v_z \\ a_{31}v_x + a_{32}v_y + a_{33}v_y \end{bmatrix}$$

What are the matrices for these transformations?

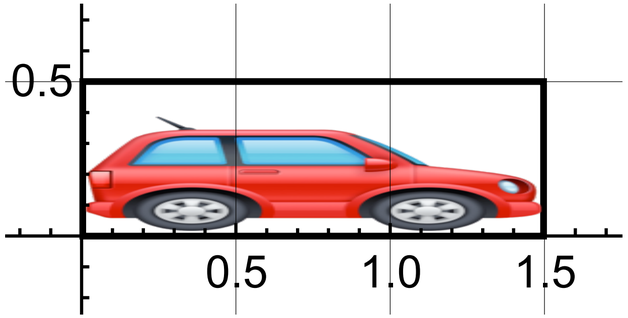




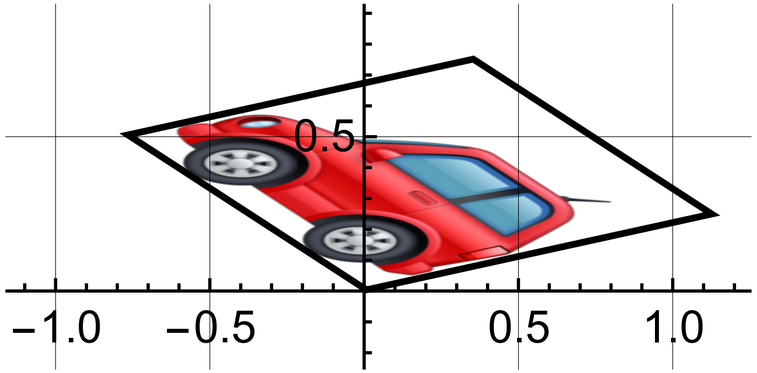
Translation



Rotation



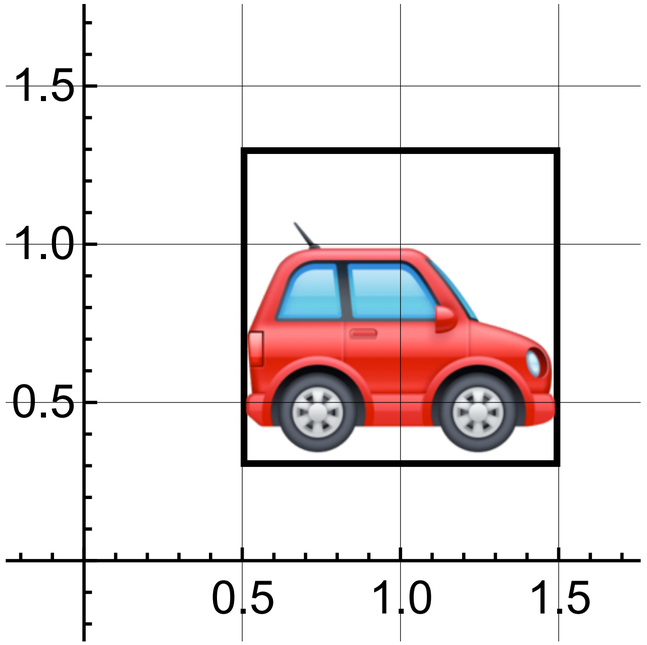
Scaling



???

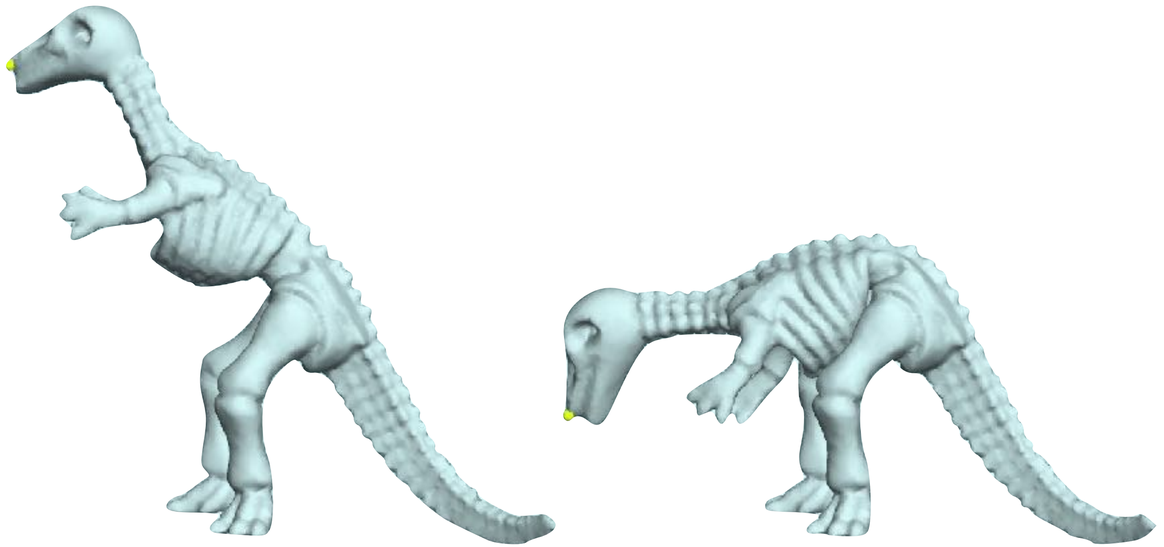
What can’t matrices do?

$$\mathbf{v}_{new} \neq \mathbf{A}\mathbf{v}_{old}$$



Translation

(for now…)



Sorkine & Alexa 2007

Nonlinear deformation

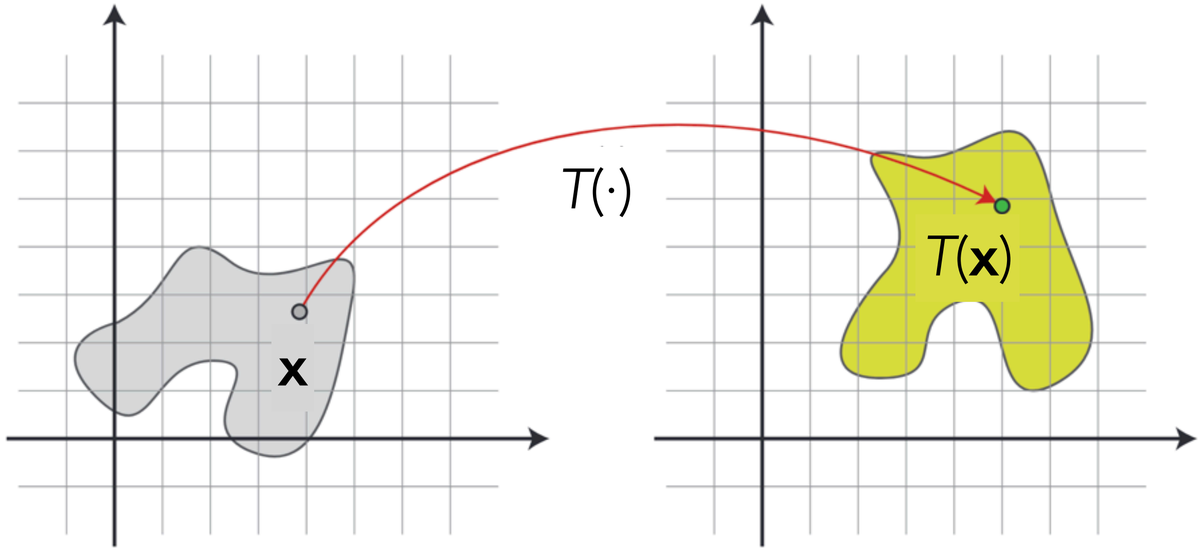
Transformations

A transformation is just a function that map points to points

$$T : \mathbb{R}^n \to \mathbb{R}^n$$

Now: **linear transformations**
(easy to represent with matrices)

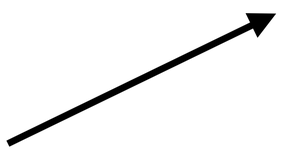
Later: affine transformations
(linear transformations + translation)



Linear algebra

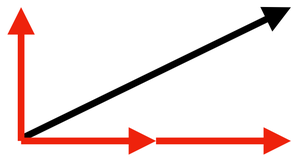
Linear algebra

Linear algebra is not about little lists of numbers!

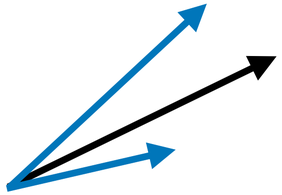


$$\neq \quad \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

A vector only has coordinates once you make an (arbitrary) choice of basis

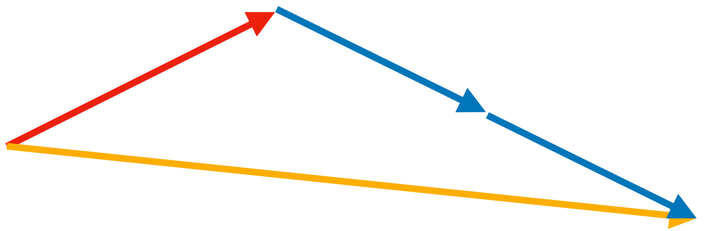


$$\begin{bmatrix} 2 \\ 1 \end{bmatrix}$$



$$\begin{bmatrix} 1 \\ 0.5 \end{bmatrix}$$

Outcomes of operations should be independent of arbitrary choices!



If **a** + 2**b** = **c** in my basis, then it should be true in your basis as well.

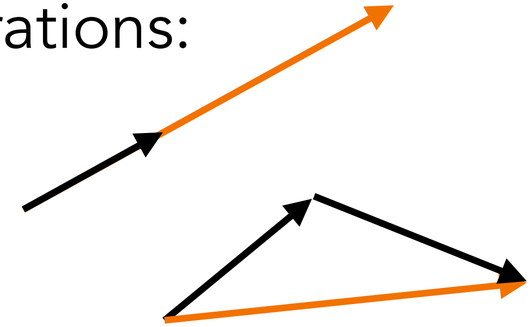
Best to think in a **basis-independent** way as much as possible

Though, to compute anything we will always need a basis in the end…

What are vectors, really?

A vector is an element of a **vector space**.

A vector space over $\mathbb{R}$ is any set V equipped with two operations:

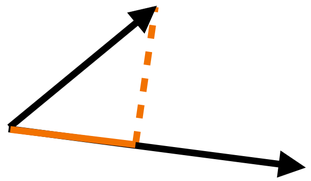


- scalar multiplication: $\mathbb{R} \times V \to V$
- vector addition: $V \times V \to V$

satisfying various identities, e.g. $\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$, $a(\mathbf{u} + \mathbf{v}) = a\mathbf{u} + a\mathbf{v}$, etc.

To do geometry, we also need a third operation:

- dot product / inner product: $V \times V \to \mathbb{R}$



satisfying identities like $\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{u}$, $(a\mathbf{u} + b\mathbf{v}) \cdot \mathbf{w} = a(\mathbf{u} \cdot \mathbf{w}) + b(\mathbf{v} \cdot \mathbf{w})$, etc.

Think of these three operations as the “public API” of the vector data type.

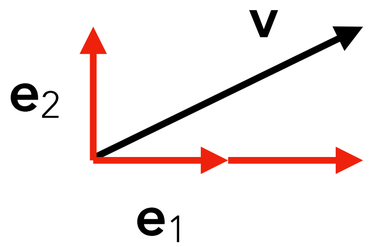
Write your algorithm and code in terms of these, and it will work in 2D, in 3D, and in any n dimensions!

(It will also work for other vector spaces: functions, images, etc. …)

Bases

A basis is just a set of vectors $\{\mathbf{e}_1, \mathbf{e}_2, \ldots\}$ such that **any** vector can be written **uniquely** as a linear combination of them.

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \end{bmatrix} \text{ in this basis} \quad \Leftrightarrow \quad \mathbf{v} = v_1\mathbf{e}_1 + v_2\mathbf{e}_2 + \cdots$$



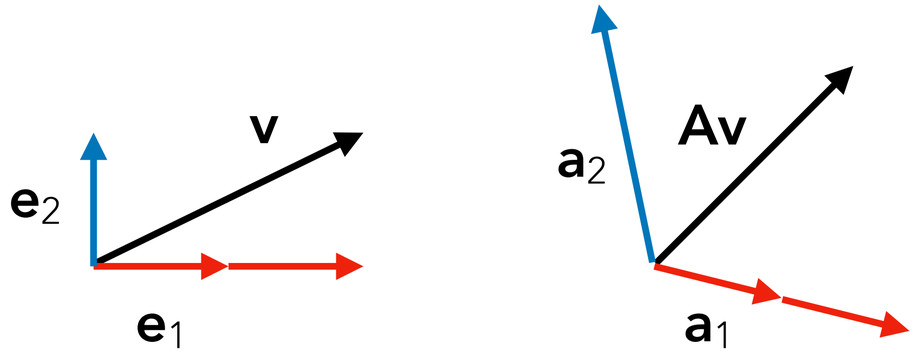
$$\begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

What happens when you apply a matrix **A** to the basis vectors?

$$\mathbf{A}\mathbf{e}_1 = \begin{bmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & a_{22} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ \vdots \end{bmatrix} = \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \end{bmatrix} = \text{1st column of } \mathbf{A}$$

This determines the action of **A** on all other vectors!

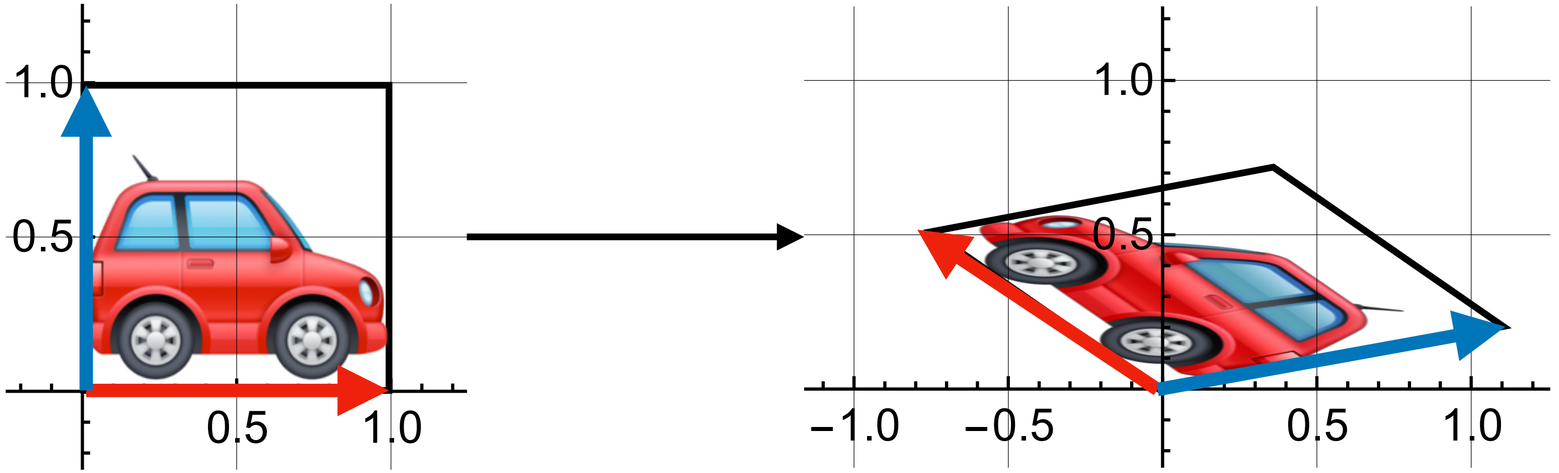
$$\mathbf{Av} = \mathbf{A}(v_1\mathbf{e}_1 + v_2\mathbf{e}_2 + \cdots) = v_1(\mathbf{Ae}_1) + v_2(\mathbf{Ae}_2) + \cdots = v_1\mathbf{a}_1 + v_2\mathbf{a}_2 + \cdots$$



- **Interpretation 1:** A matrix transforms the basis vectors to its columns; all other vectors follow.

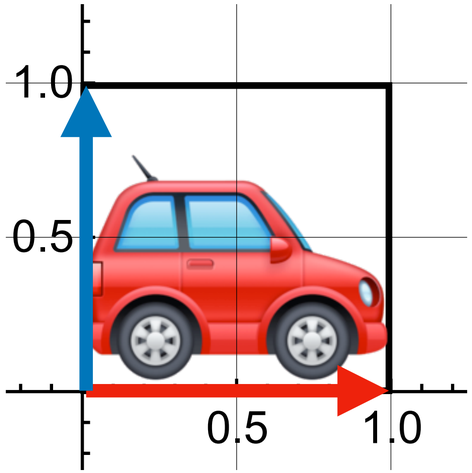
- **Interpretation 2:** Matrix-vector multiplication **Av** produces a linear combination of the columns of **A**, weighted by the components v_1, v_2, …

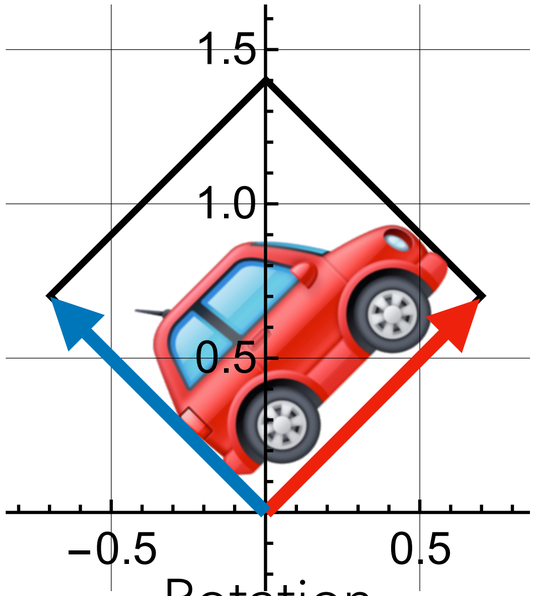
Now, what is the matrix for this transformation?



$\mathbf{a}_1$ = image of $\mathbf{e}_1 \approx \begin{bmatrix} -0.8 \\ 0.5 \end{bmatrix}$, $\mathbf{a}_2$ = image of $\mathbf{e}_2 \approx \begin{bmatrix} 1.1 \\ 0.2 \end{bmatrix}$

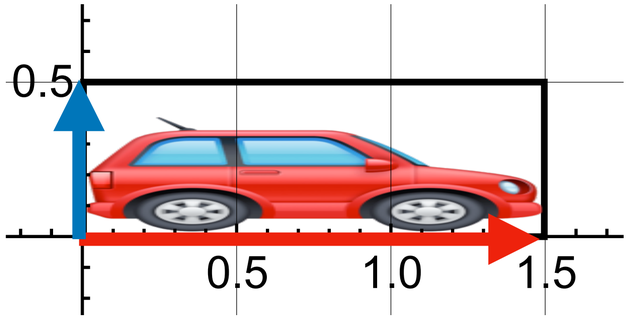
$$\mathbf{A} \approx \begin{bmatrix} -0.8 & 1.1 \\ 0.5 & 0.2 \end{bmatrix}$$





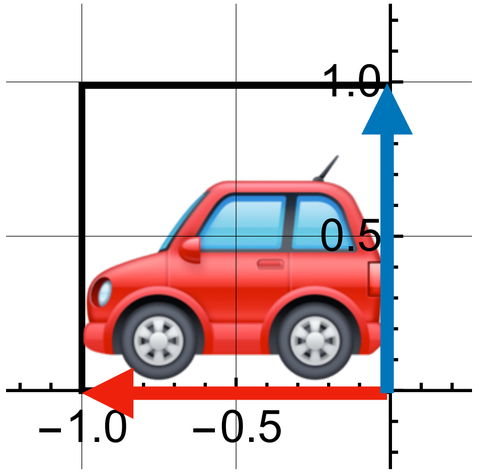
Rotation

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$



Scaling

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$



Reflection

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

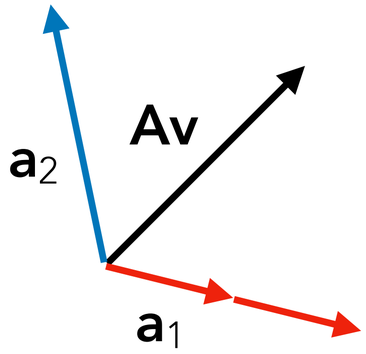
Composition of transformations

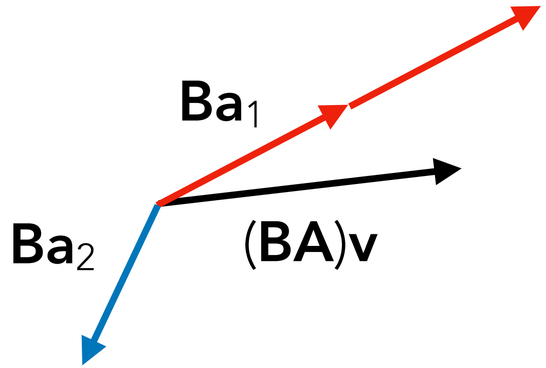
Apply transformation **A** then transformation **B**:

$$\mathbf{v} \quad \rightarrow \quad \mathbf{Av} \quad \rightarrow \quad \mathbf{B(Av)} = \mathbf{(BA)v}$$

Column interpretation:

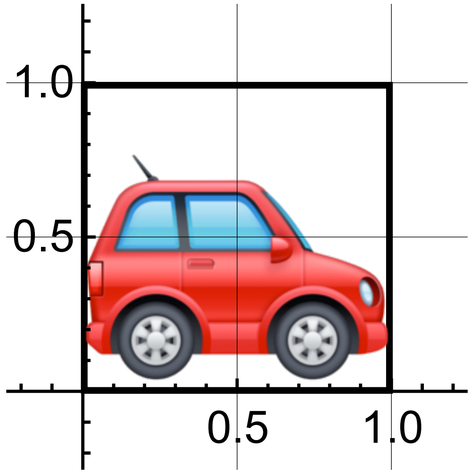
$$\mathbf{B}\begin{bmatrix}\mathbf{a}_1 & \mathbf{a}_2 & \cdots\end{bmatrix} = \begin{bmatrix}\mathbf{Ba}_1 & \mathbf{Ba}_2 & \cdots\end{bmatrix}$$



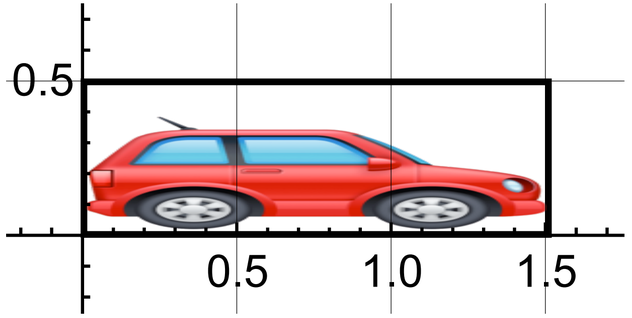


Often, want to apply a sequence of *n* transformations on millions of vertices.
Just compute the product: then only 1 matrix-vector multiplication per vertex.

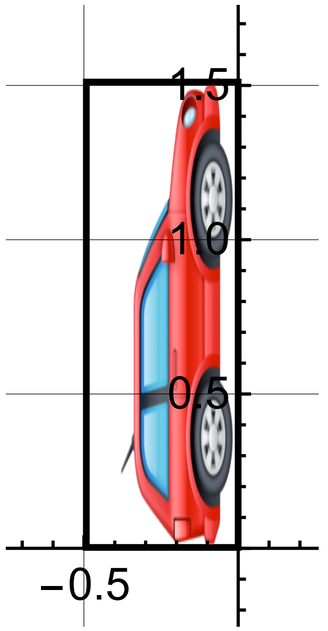
AB ≠ BA

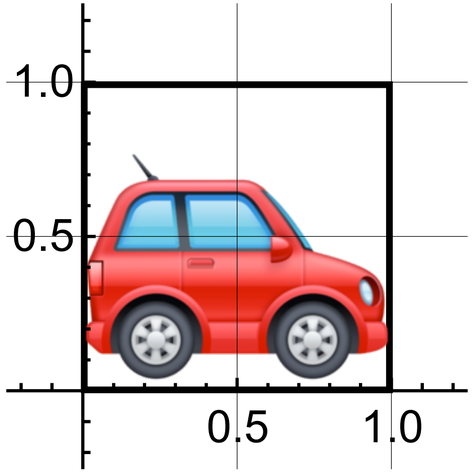


Stretch *x*, shrink *y*

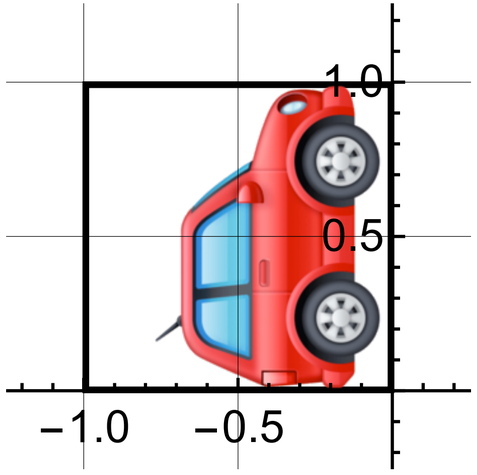


Rotate by 90°

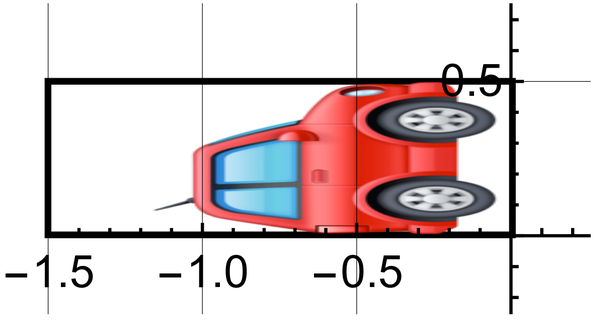




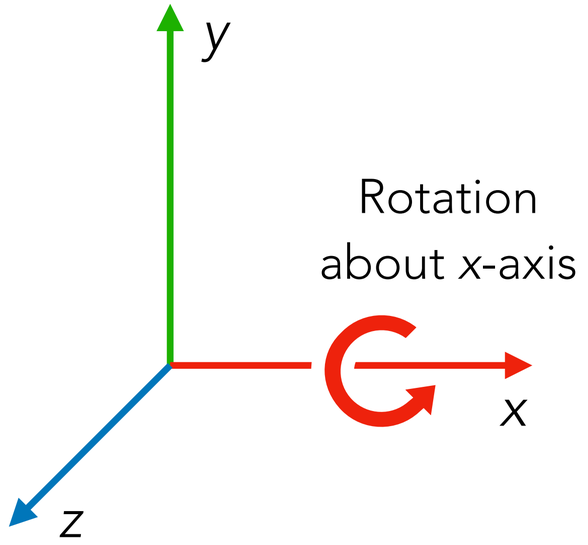
Rotate by 90°



Stretch *x*, shrink *y*



Rotations in 3D



Rotations about the coordinate axes:

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

Rotation about *x*-axis
= Rotation in *yz*-plane

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

Rotation about *y*-axis
= Rotation in ***zx***-plane

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

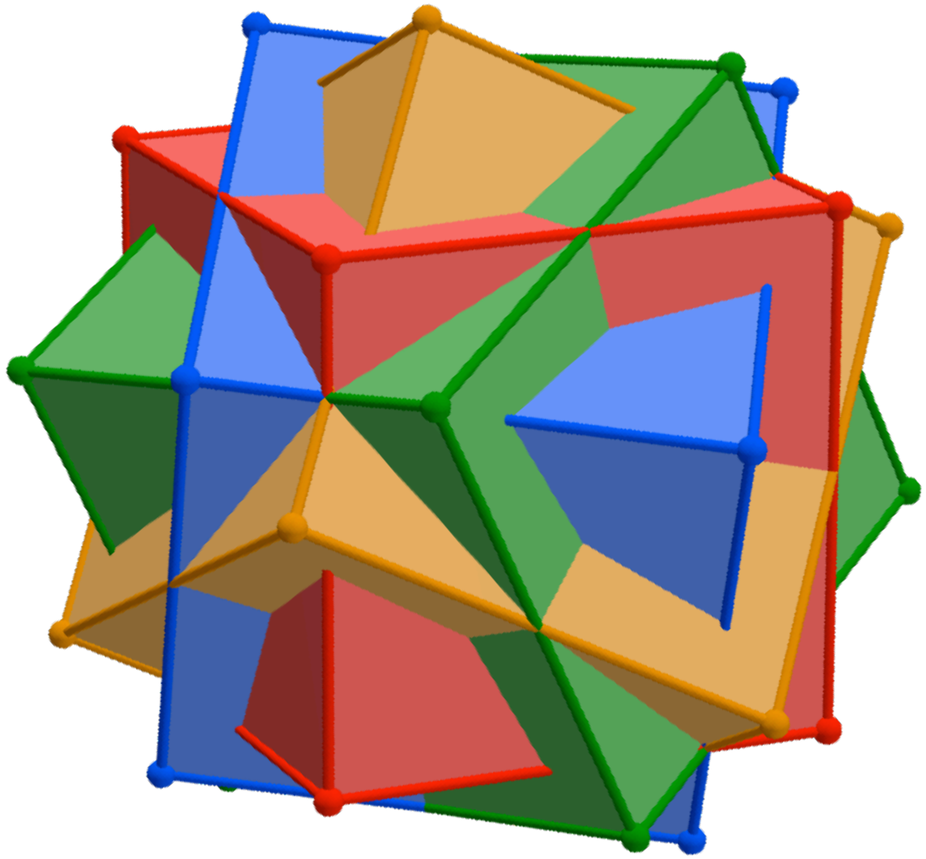
Rotation about *z*-axis
= Rotation in *xy*-plane

Are these all the possible rotations?

Rotations in 3D

Are these all possible rotations?

Not at all!



A rotation is any transformation which:

- preserves distances and angles
- preserves orientation
  (“F” can become “Ⅎ” but not “Ⅎ”)

Equivalently, $\mathbf{R}^T\mathbf{R} = \mathbf{I}$, and $\det \mathbf{R} = 1$

Rodrigues' rotation formula

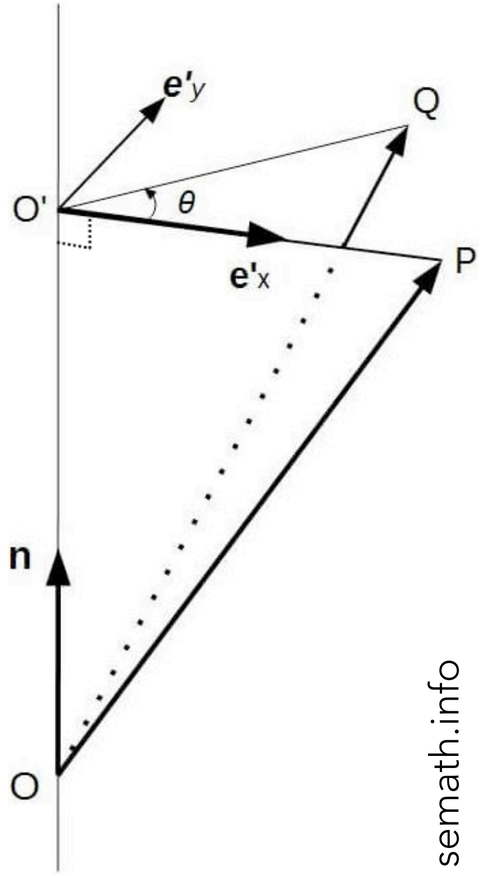
Rotation around an axis **n** by angle θ:

$$\mathbf{R} = \mathbf{I} \cos\theta + [\mathbf{n}]_\times \sin\theta + \mathbf{n}\,\mathbf{n}^\mathsf{T}(1 - \cos\theta)$$

$$\text{where } [\mathbf{n}]_\times = \begin{bmatrix} 0 & -n_z & n_y \\ n_z & 0 & -n_x \\ -n_y & n_x & 0 \end{bmatrix}$$

How? Hints:

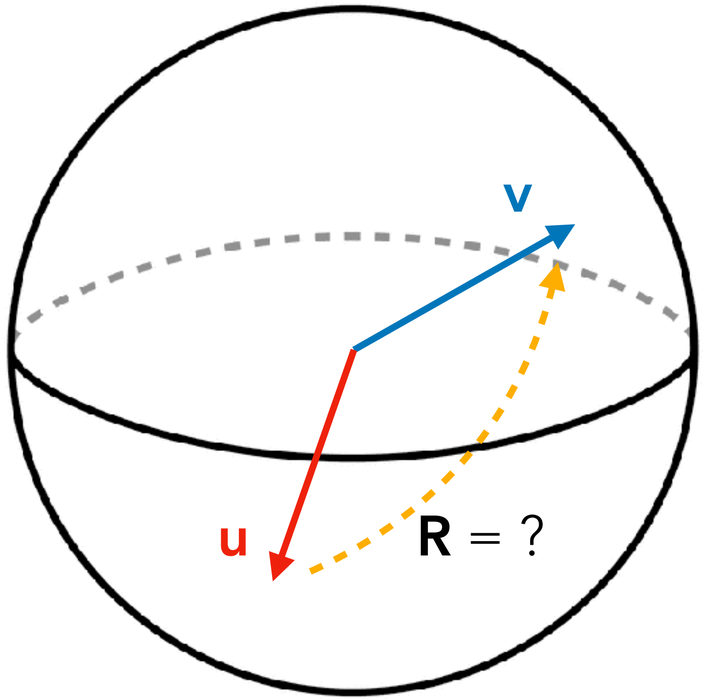
- $[\mathbf{n}]_\times$ is the "cross-product matrix": $[\mathbf{n}]_\times \mathbf{v} = \mathbf{n} \times \mathbf{v}$
- Assume an orthogonal basis **n**, $\mathbf{e}_1$, $\mathbf{e}_2$ and see what **R** does to it



semath.info

Given unit vectors **u** and **v**, find a way to construct a rotation matrix **R** which maps **u** to **v**, i.e. **Ru** = **v**.

Is it unique, or are there many different such rotations?

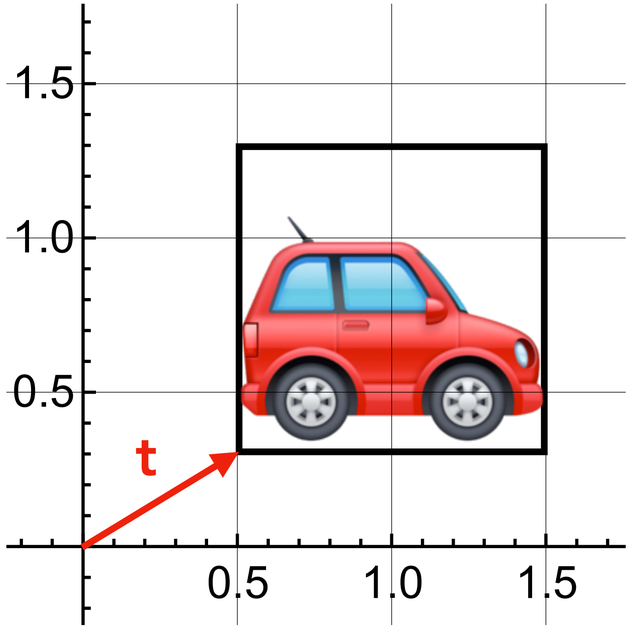


15 minute break

Translations

Move all points by a constant displacement

$$T(\mathbf{p}) = \mathbf{p} + \mathbf{t}$$

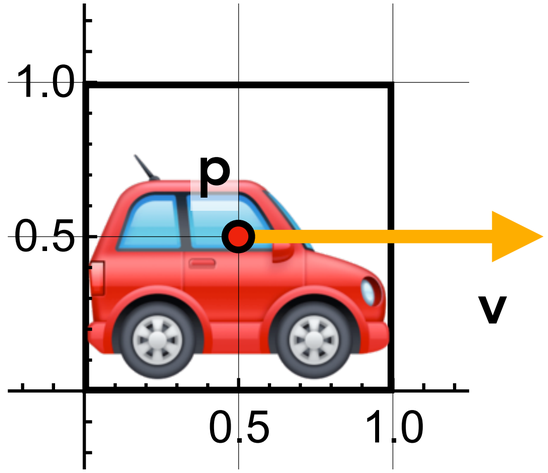


So a linear transformation followed by a translation will be of the form $T(\mathbf{p}) = \mathbf{A}\mathbf{p} + \mathbf{b}$

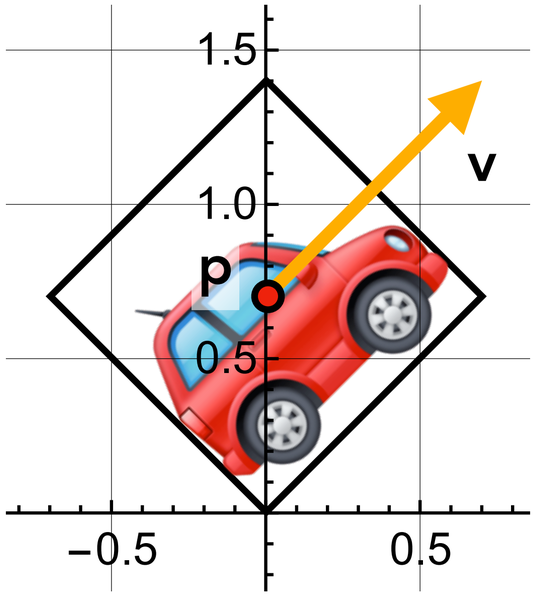
A bit tedious to compose:

$$T_2(T_1(\mathbf{p})) = \mathbf{A}_2(\mathbf{A}_1\mathbf{p} + \mathbf{b}_1) + \mathbf{b}_2 = (\mathbf{A}_2\mathbf{A}_1)\mathbf{p} + (\mathbf{A}_2\mathbf{b}_1 + \mathbf{b}_2)$$

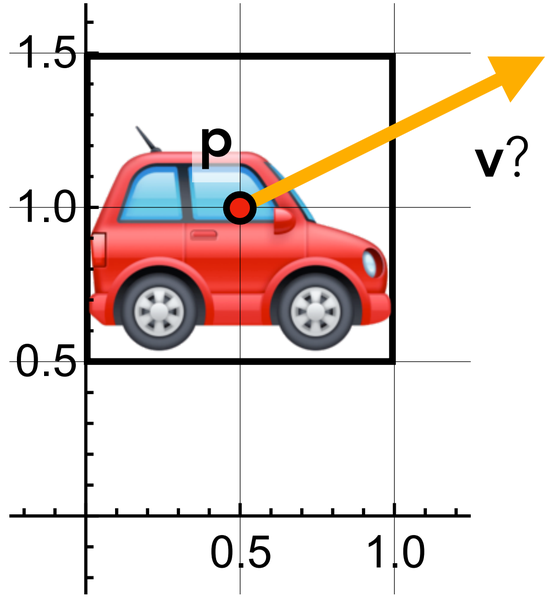
Suppose I have both points and directions/velocities/etc. to transform.



Original:
p = (0.5, 0.5)
v = (1, 0)



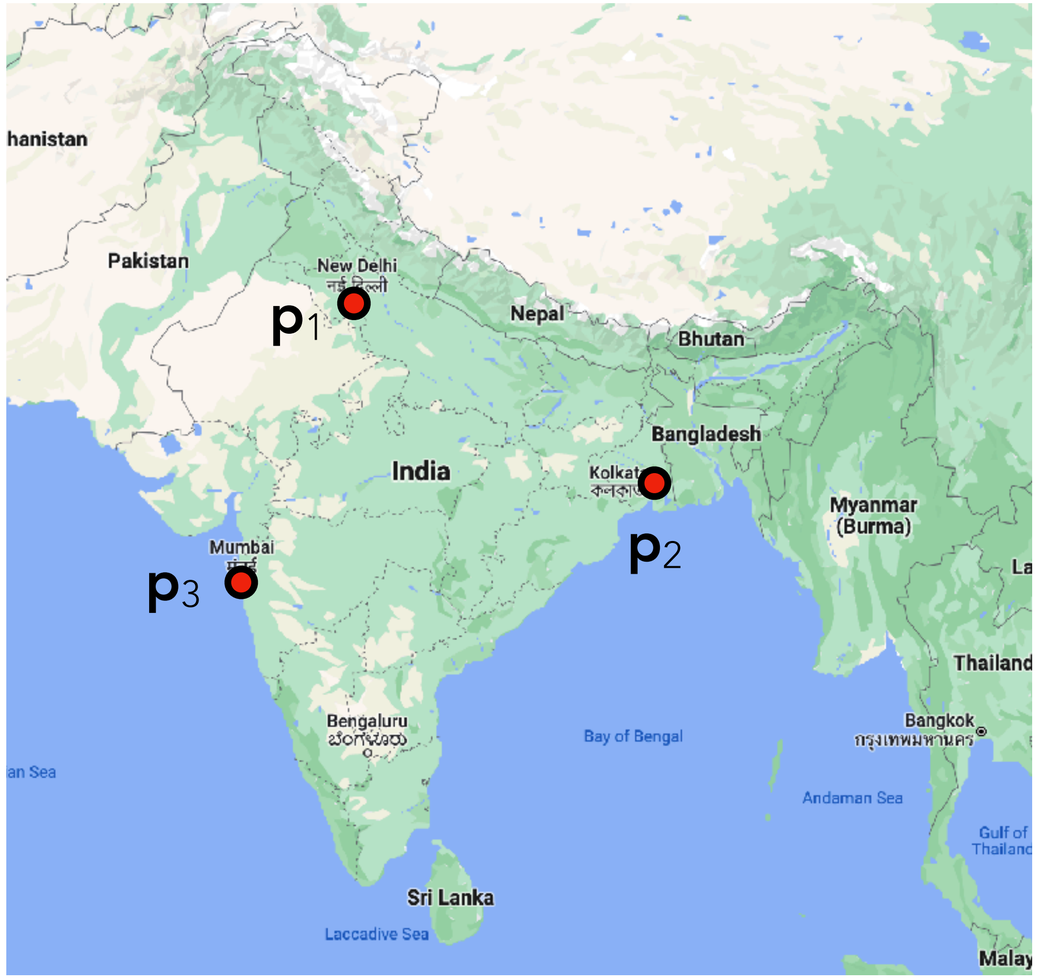
Rotation by 45°:
p = (0, 0.7)
v = (0.7, 0.7)



Translation by (0, 0.5):
p = (0.5, 1)
v = (1, 0.5)?

It seems translation should only affect some things, not others. But why?

Are points really vectors?



$\mathbf{p}_1 + \mathbf{p}_2 = ?$

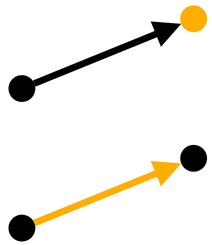
$5\mathbf{p}_3 = ?$

How about I just choose an origin and then add the displacement vectors?

Points vs. vectors

Points form an **affine space** A over the vector space V.

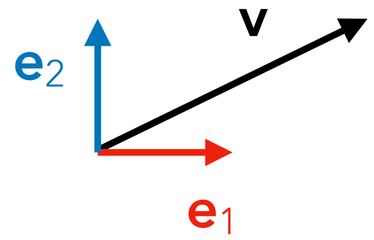
- Point-vector addition: $A \times V \to \boldsymbol{A}$
- Point subtraction: $A \times A \to \boldsymbol{V}$



with the obvious properties e.g. $(\mathbf{p} + \mathbf{u}) + \mathbf{v} = \mathbf{p} + (\mathbf{u} + \mathbf{v})$, $\mathbf{p} + (\mathbf{q} - \mathbf{p}) = \mathbf{q}$, etc.

Coordinate frames

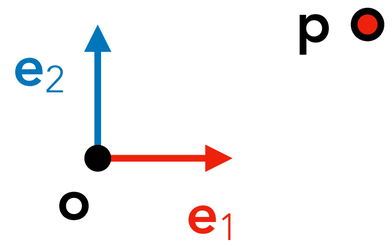
To specify a vector numerically, we need a basis



$$\mathbf{v} = v_1\mathbf{e}_1 + v_2\mathbf{e}_2 + \cdots \quad \Leftrightarrow \quad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \end{bmatrix} \text{ in the basis}$$

To specify a **point** numerically, we need a **coordinate frame**: origin and basis

$$\mathbf{p} = p_1\mathbf{e}_1 + p_2\mathbf{e}_2 + \cdots + \mathbf{o} \quad \text{so maybe} \quad \mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ 1 \end{bmatrix} ?$$



Write a point as an (n+1)-tuple $\mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ 1 \end{bmatrix}$ to mean $\mathbf{p} = p_1\mathbf{e}_1 + p_2\mathbf{e}_2 + \cdots + \mathbf{o}$.

Linear transformations are now $\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & 1 \end{bmatrix}$, mapping $\mathbf{e}_i \rightarrow \mathbf{A}\mathbf{e}_i$ and $\mathbf{o} \rightarrow \mathbf{o}$

$$\text{e.g. } \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} = \begin{bmatrix} s_x p_x \\ s_y p_y \\ 1 \end{bmatrix}$$

Translation by a vector $\mathbf{t}$: $\begin{bmatrix} \mathbf{I} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}$, mapping $\mathbf{e}_i \rightarrow \mathbf{e}_i$ but $\mathbf{o} \rightarrow \mathbf{o} + \mathbf{t}$

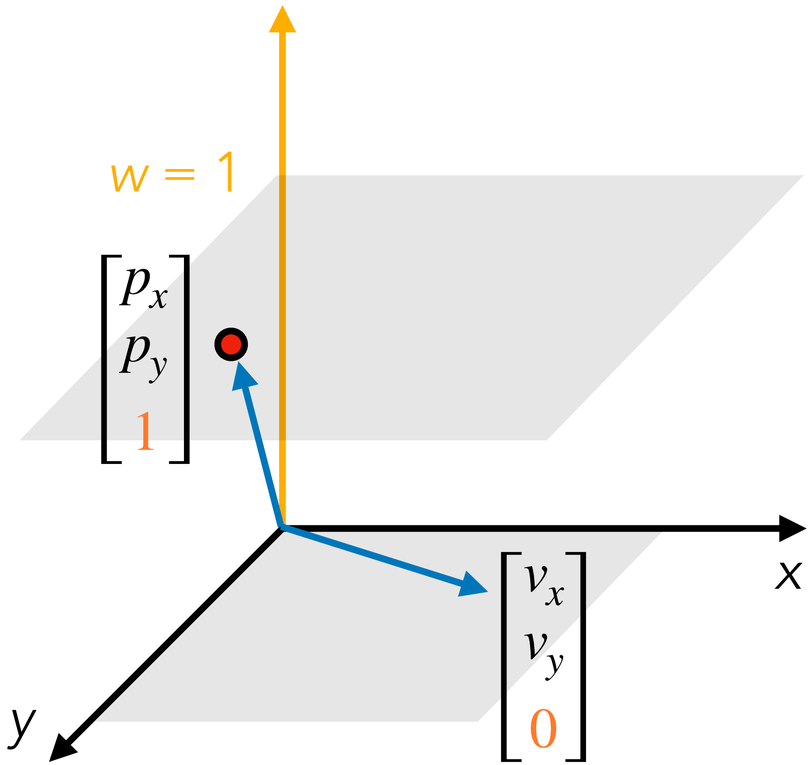
$$\text{e.g. } \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + t_x \\ p_y + t_y \\ 1 \end{bmatrix}$$

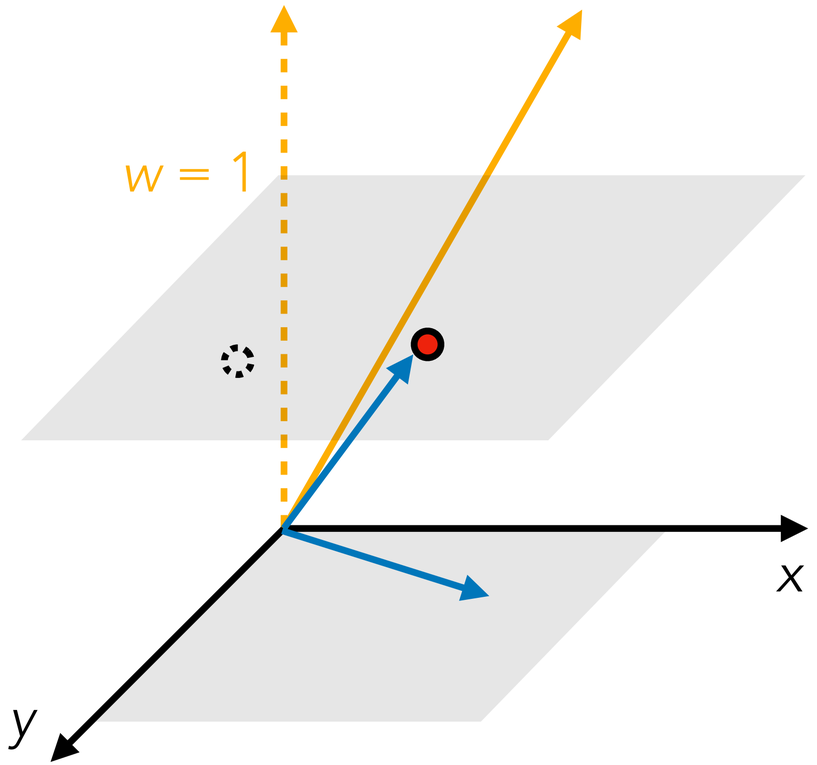
What about vectors?

$$\mathbf{v} = v_1\mathbf{e}_1 + v_2\mathbf{e}_2 + \cdots + 0\mathbf{o} \quad \Leftrightarrow \quad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ 0 \end{bmatrix}$$

Apply a translation:

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ 0 \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ 0 \end{bmatrix}$$


w = 1
$\begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix}$
x
y
$\begin{bmatrix} v_x \\ v_y \\ 0 \end{bmatrix}$


w = 1
x
y

Homogeneous coordinates

Add an extra coordinate w at the end.

- Points: $w = 1$
- Vectors: $w = 0$

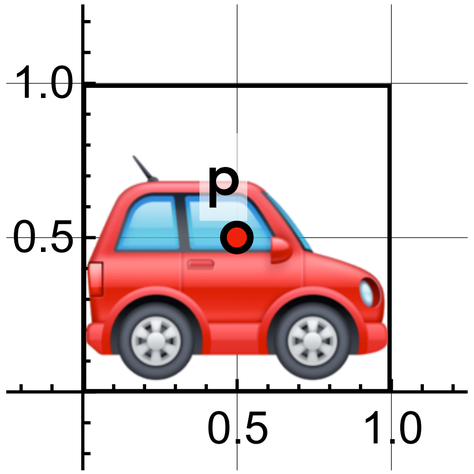
Transformations become $(n+1)\times(n+1)$ matrices

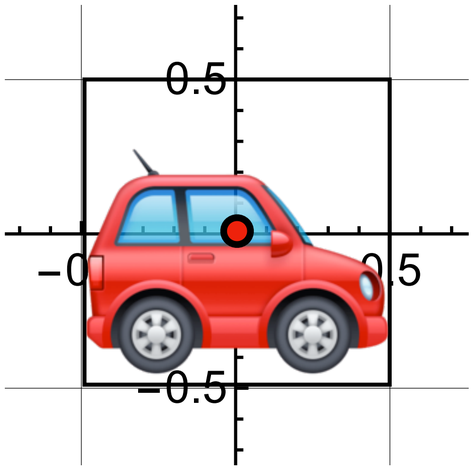
- Linear transformations: $\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & 1 \end{bmatrix}$
- Translations: $\begin{bmatrix} \mathbf{I} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}$

General **affine transformation**: $\begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}$

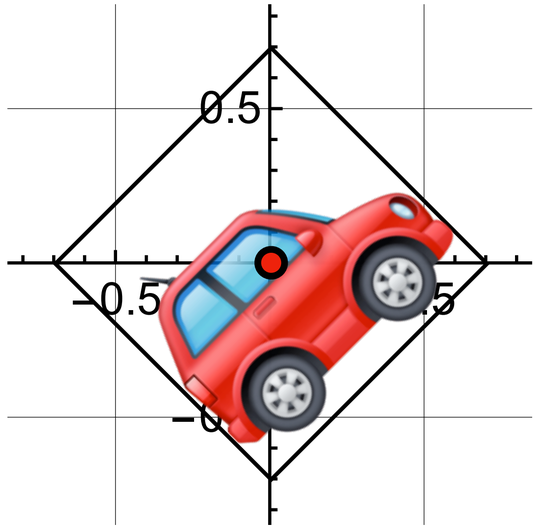
- Corresponds to linearly transforming basis vectors $\mathbf{e}_i \rightarrow \mathbf{A}\mathbf{e}_i$ and translating origin $\mathbf{o} \rightarrow \mathbf{o} + \mathbf{t}$
- Maps parallel lines to parallel lines, but does not preserve the origin
- Composition: just matrix multiplication again.

Example: Rotate by given angle θ about given point **p** (instead of about origin)

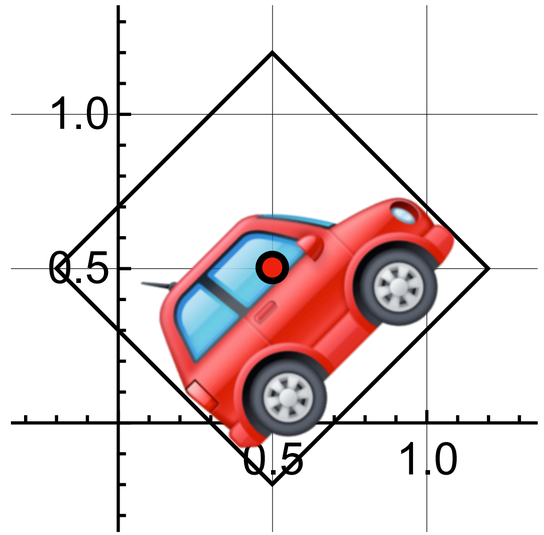




Translate by $-\mathbf{p}$



Rotate by θ
about origin



Translate by **p**

$$\mathbf{M} = \mathbf{T}(\mathbf{p})\,\mathbf{R}(\theta)\,\mathbf{T}(-\mathbf{p})$$

Given coordinates of **p** in frame 1, what are its coordinates in frame 2?

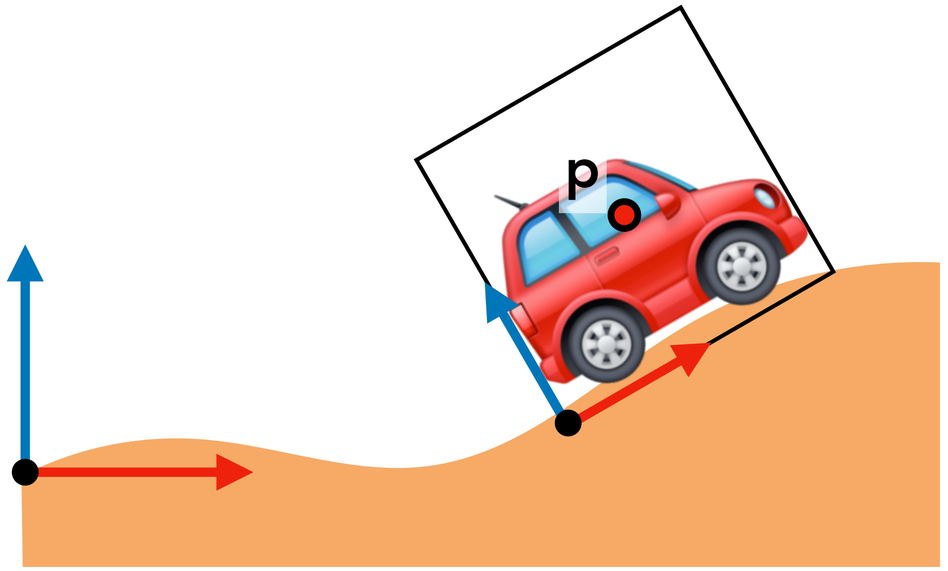
$$\mathbf{p} = p_1\mathbf{e}_1 + p_2\mathbf{e}_2 + \cdots + \mathbf{o}$$

Write coords of $\mathbf{e}_1$, $\mathbf{e}_2$, … and $\mathbf{o}$ in frame 2:

$$\mathbf{e}_i = \begin{bmatrix} \bullet \\ \bullet \\ \vdots \\ 0 \end{bmatrix}, \quad \mathbf{o} = \begin{bmatrix} \bullet \\ \bullet \\ \vdots \\ 1 \end{bmatrix}$$

Then $\mathbf{p} = \begin{bmatrix} \bullet & \bullet & \cdots & \bullet \\ \bullet & \bullet & \cdots & \bullet \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ 1 \end{bmatrix}$

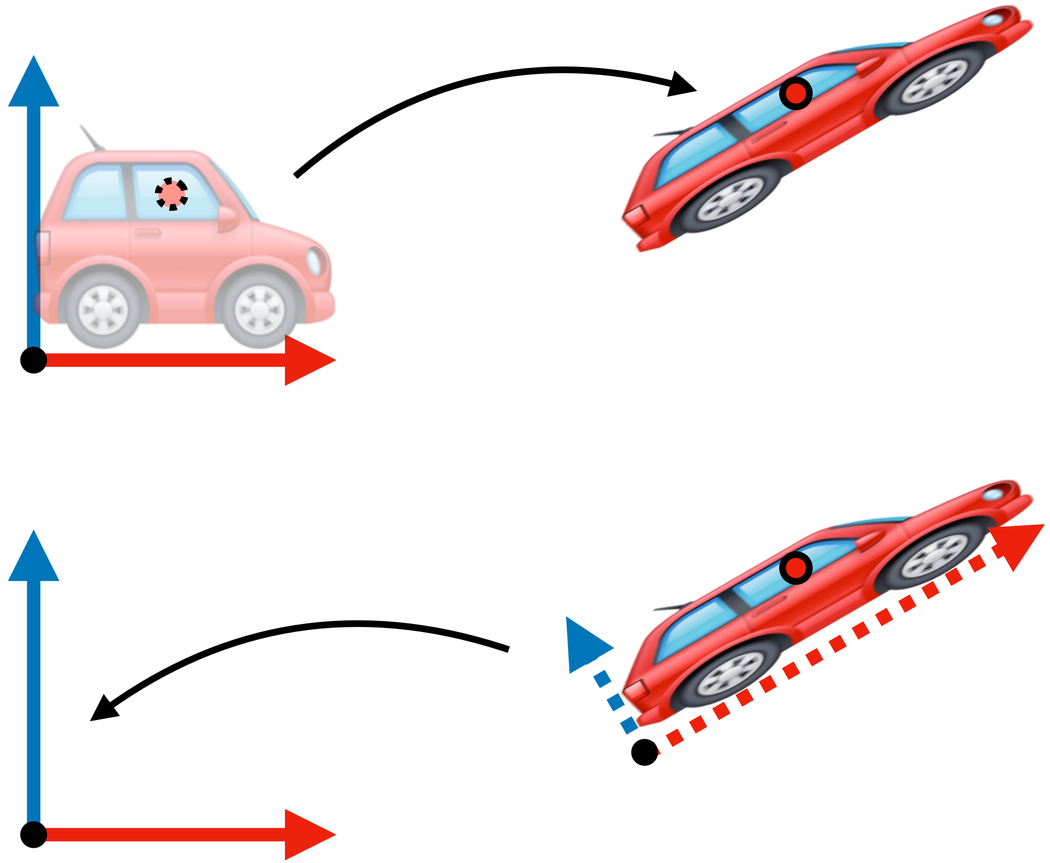
$\mathbf{e}_1$ $\mathbf{e}_2$ $\mathbf{o}$



Change of coordinates looks exactly like a transformation matrix!

Active transformation: Moves points to new locations in the same frame

Change of coordinates (passive transformation): Gives coordinates of the same point in a different frame



Matrices are the same but the meaning is different! **You** have to keep track.

e.g. `world_driver = world_from_car * car_driver`

`Vec3` `Mat3x3` `Vec3`

Perspective Projection

Luciano Testoni

Blender man

So far we know:

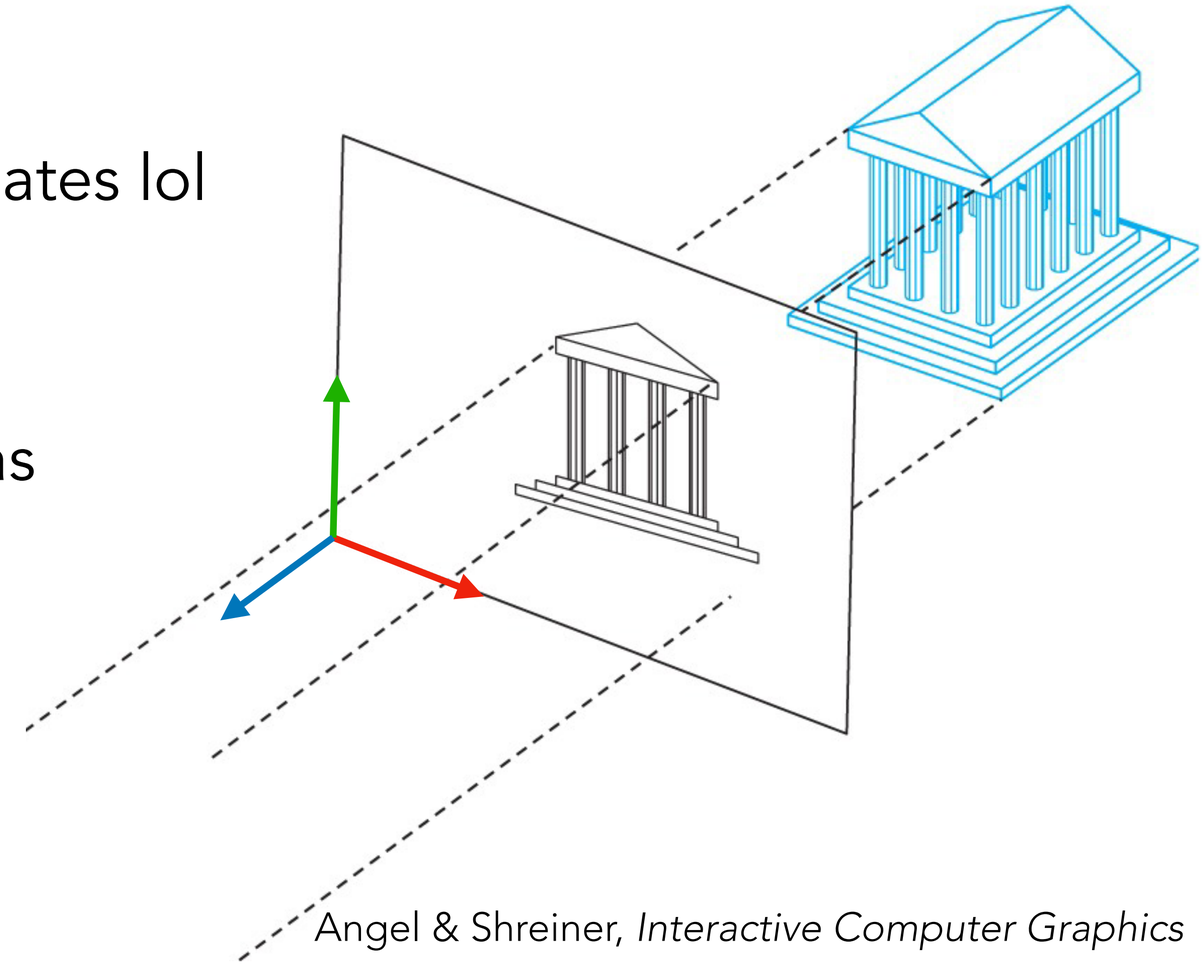
- How to draw 2D shapes
- How to transform 2D and 3D shapes

Today: How to draw 3D shapes on a 2D screen?

Parallel projection

Easy way: Just drop one of the coordinates lol

- Useful for engineering drawings
- Doesn’t match how eyes and cameras actually see things!



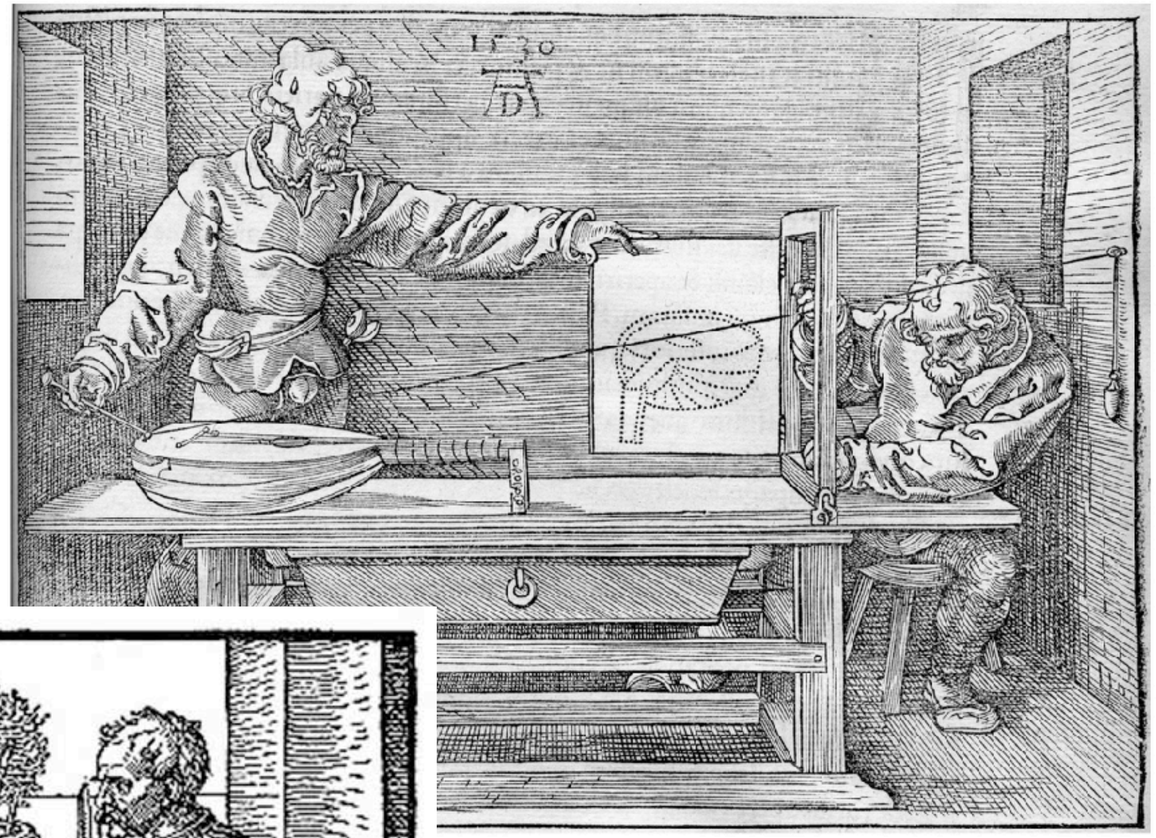
Angel & Shreiner, *Interactive Computer Graphics*

Perspective

Jeff Lynch

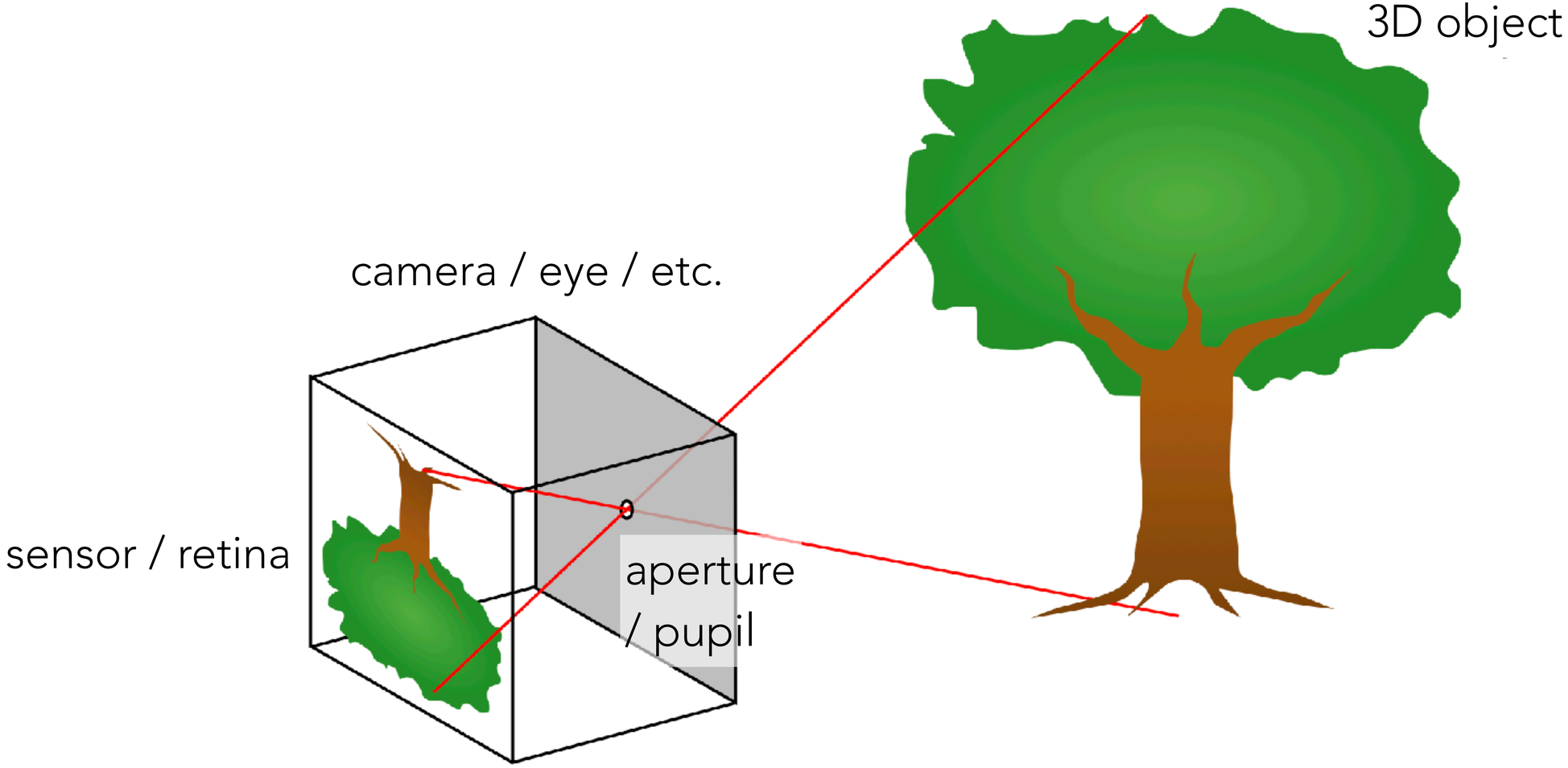
Algorithmic drawing in the 1500s

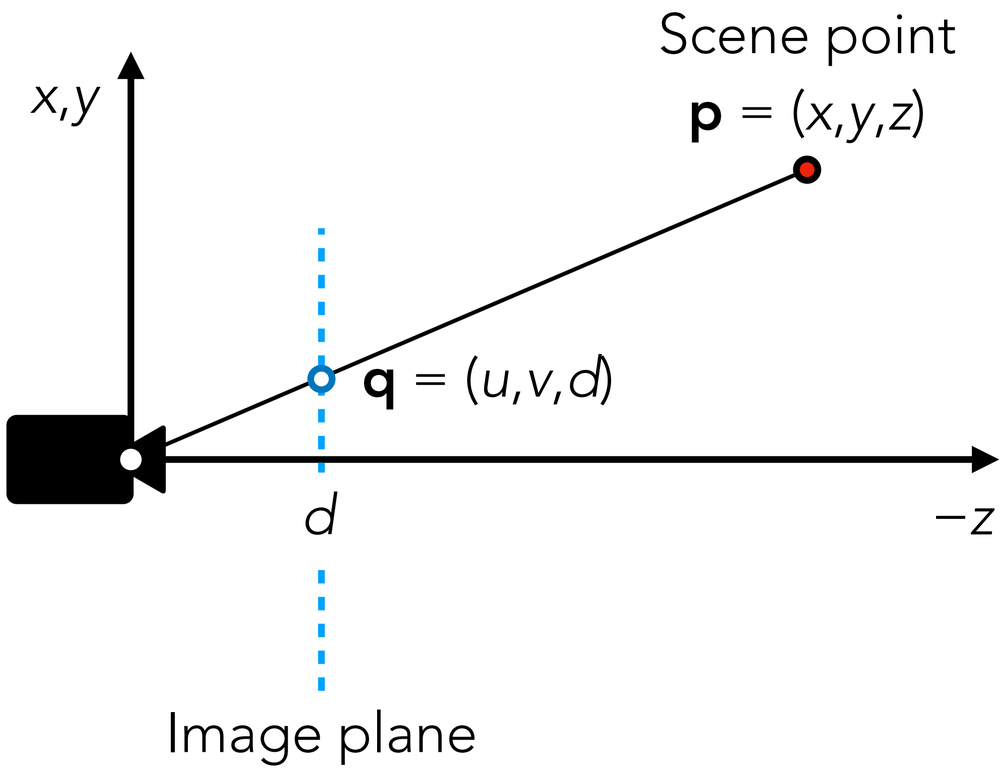
A point is drawn where the ray from the viewpoint meets the image plane.



Albrecht Dürer

Pinhole camera model





Assume camera is at the origin, pointing in the direction $-z$.

Where is the point **p** projected to?

$$\frac{x}{z} = \frac{u}{d}$$

$$u = \frac{xd}{z}$$

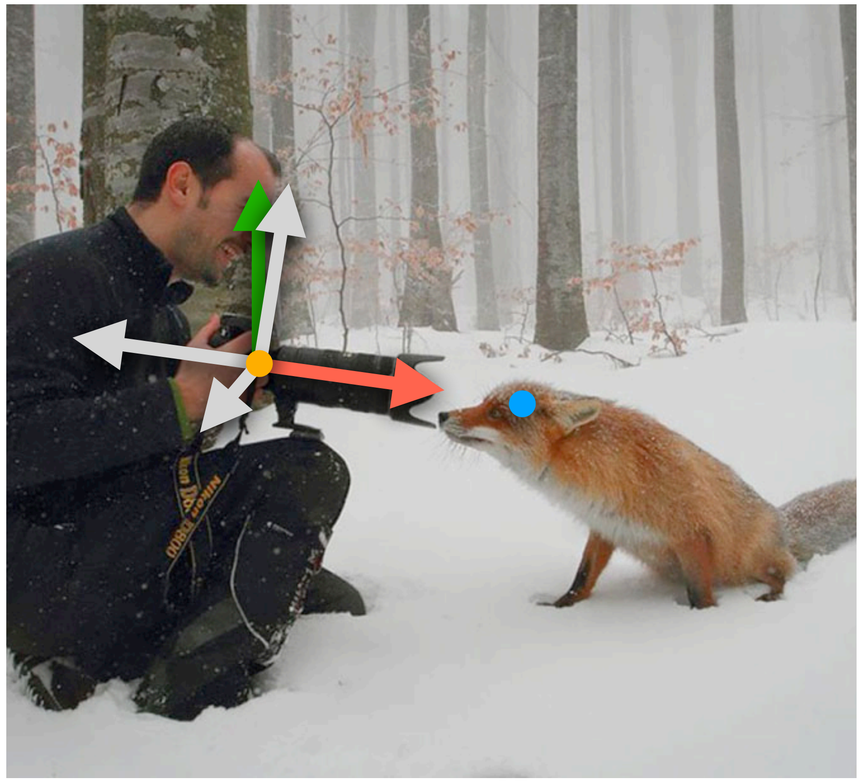
Similarly $v = yd/z$

(W.l.o.g., let's take $d = -1$)

What if the camera is not at the origin and/or not looking along $-z$?

Just change to a coordinate system in which it is.

Viewing transformation

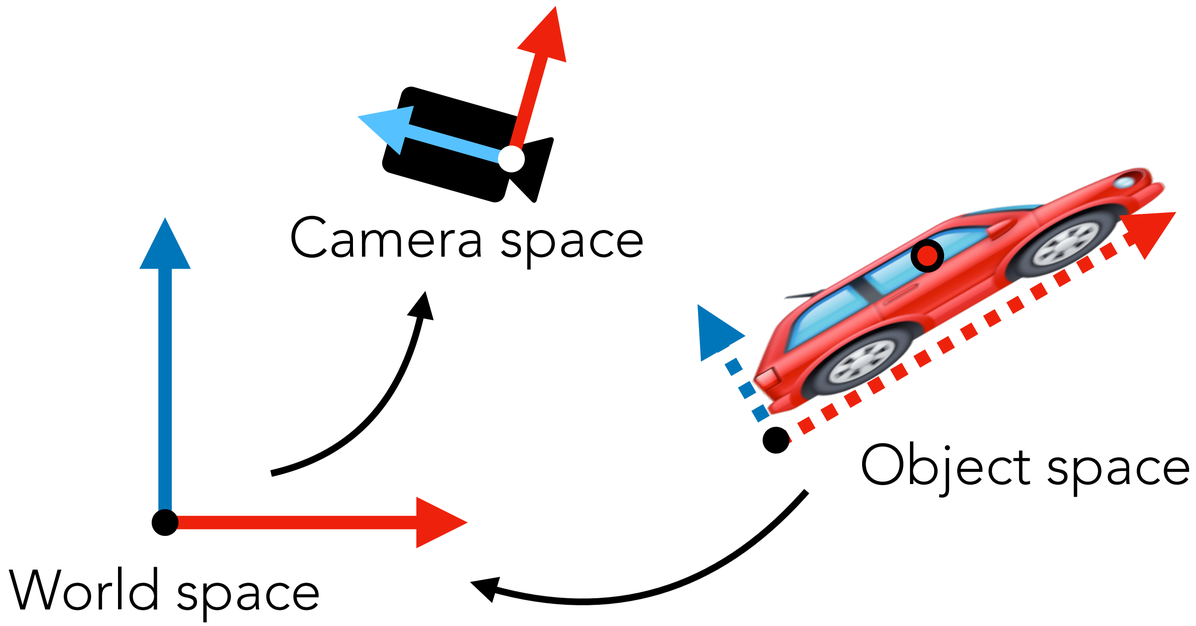


Usually, user specifies:

- center of projection **c**
- target point **t** or view vector **v** = (**t**−**c**)/‖**t**−**c**‖
- “up vector” **u**

Construct orthonormal basis

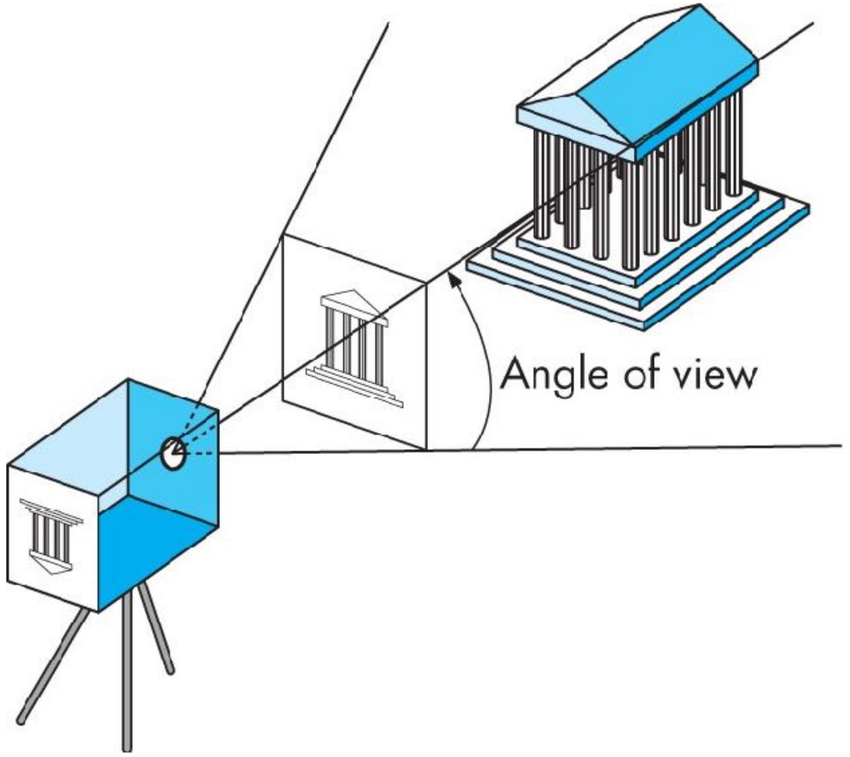
$$\mathbf{e}_2 = (\mathbf{v}\times\mathbf{u})/\|\mathbf{v}\times\mathbf{u}\|$$
$$\mathbf{e}_1 = \mathbf{v}\times\mathbf{e}_2$$
$$\mathbf{e}_3 = -\mathbf{v}$$



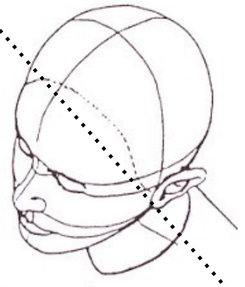
Camera → world: $\mathbf{M} = [\mathbf{e}_1 \;\; \mathbf{e}_2 \;\; \mathbf{e}_3 \;\; \mathbf{c}]$

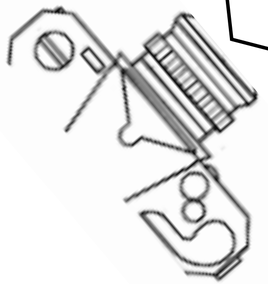
World → camera: $\mathbf{M}^{-1}$

Once point is in camera space, projected point = $\begin{bmatrix} xd/z \\ yd/z \end{bmatrix}$



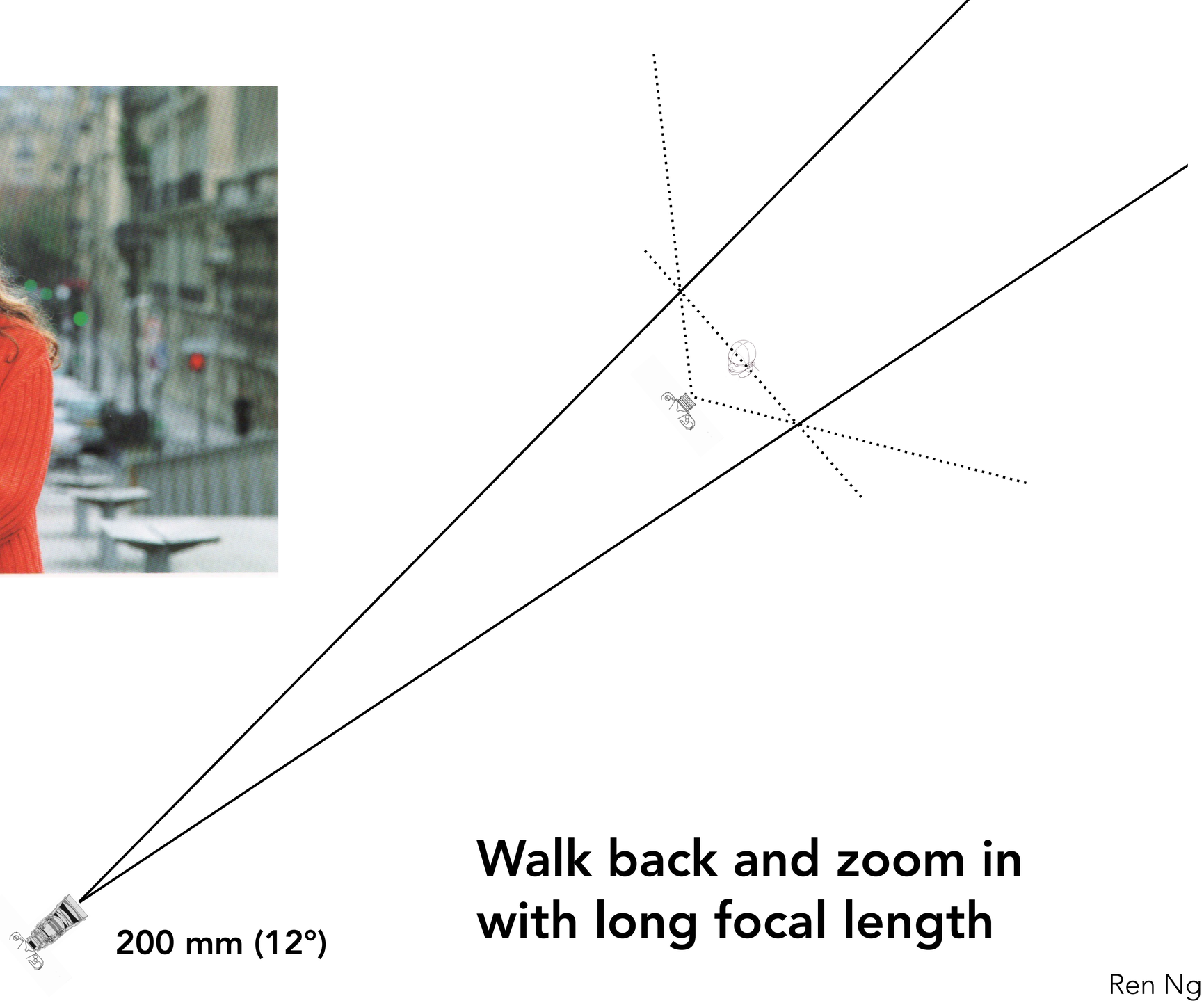
Canon EF Lens Work III





16 mm (110°)

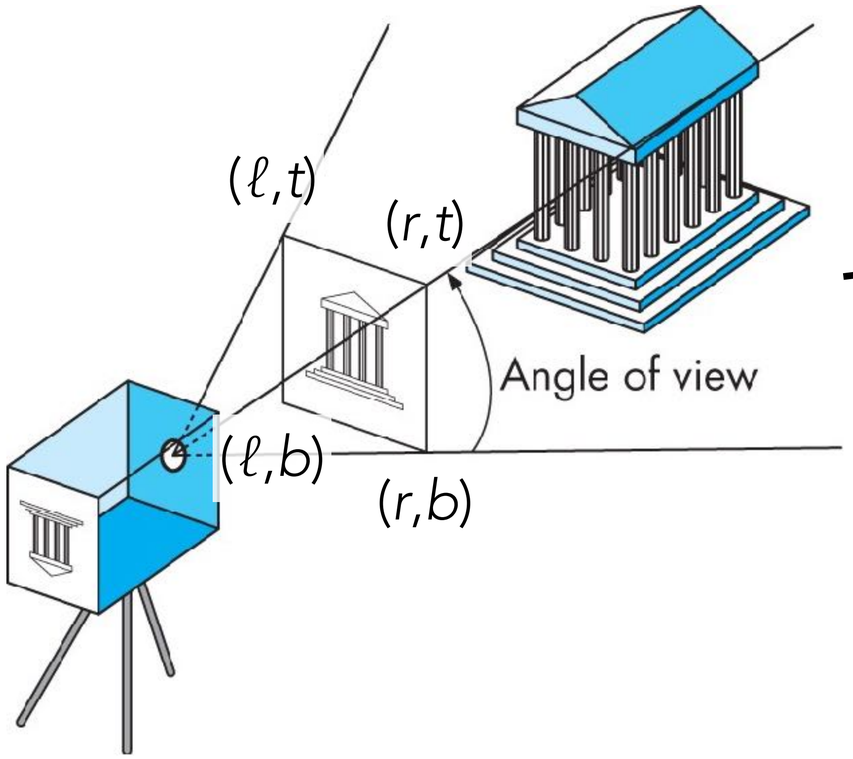
Up close and zoomed wide with short focal length



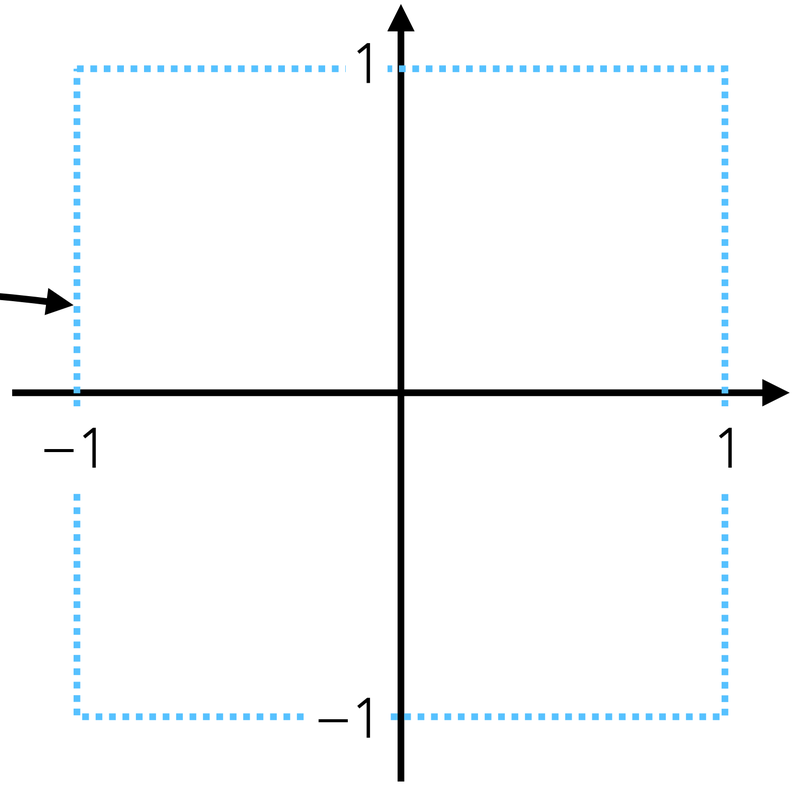
200 mm (12°)

Walk back and zoom in with long focal length

Ren Ng



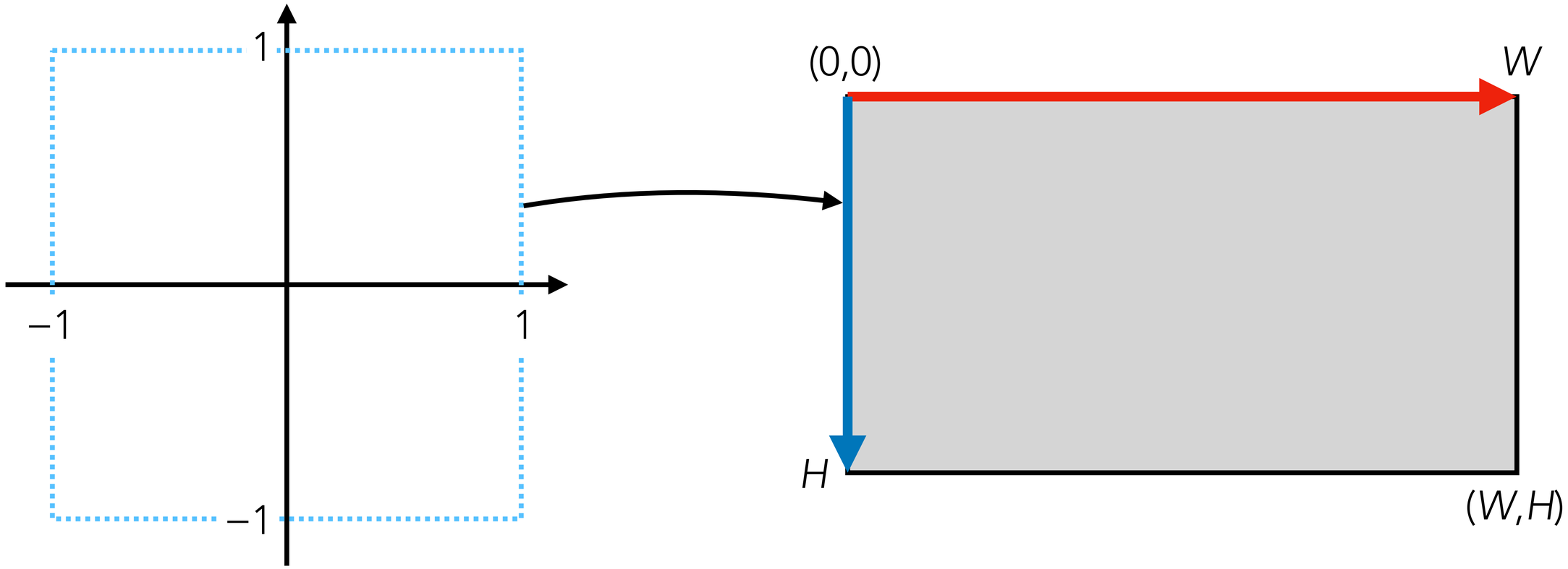
Coordinates after
perspective division



"Normalized
device coordinates"

(Actually there's a bit more in NDC… Will correct later!)

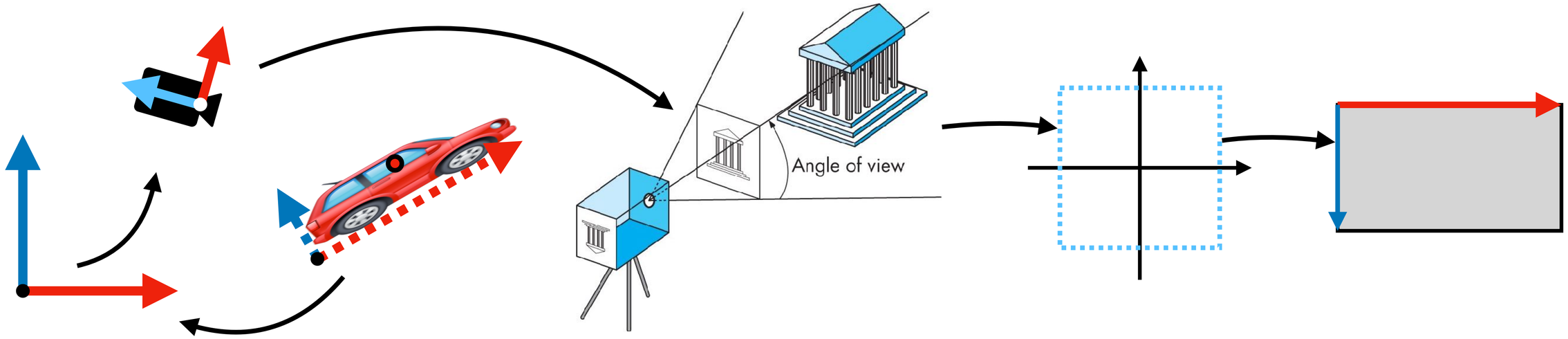
Choose transformation so that points in field of view fall inside $[-1,1] \times [-1,1]$



Normalized
device coordinates

Screen coordinates
(locations of pixels)

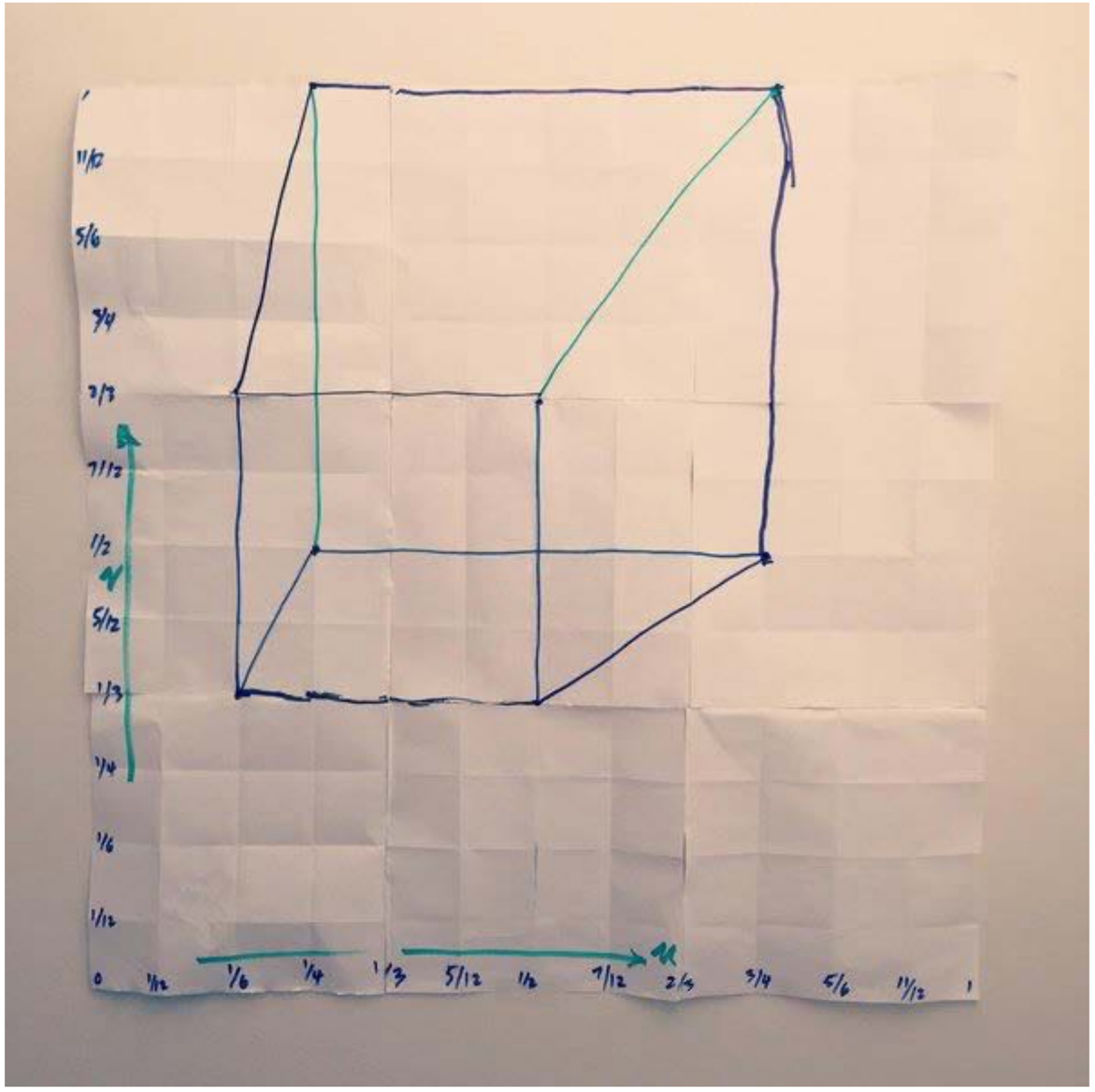
And finally, we can rasterize our triangles!

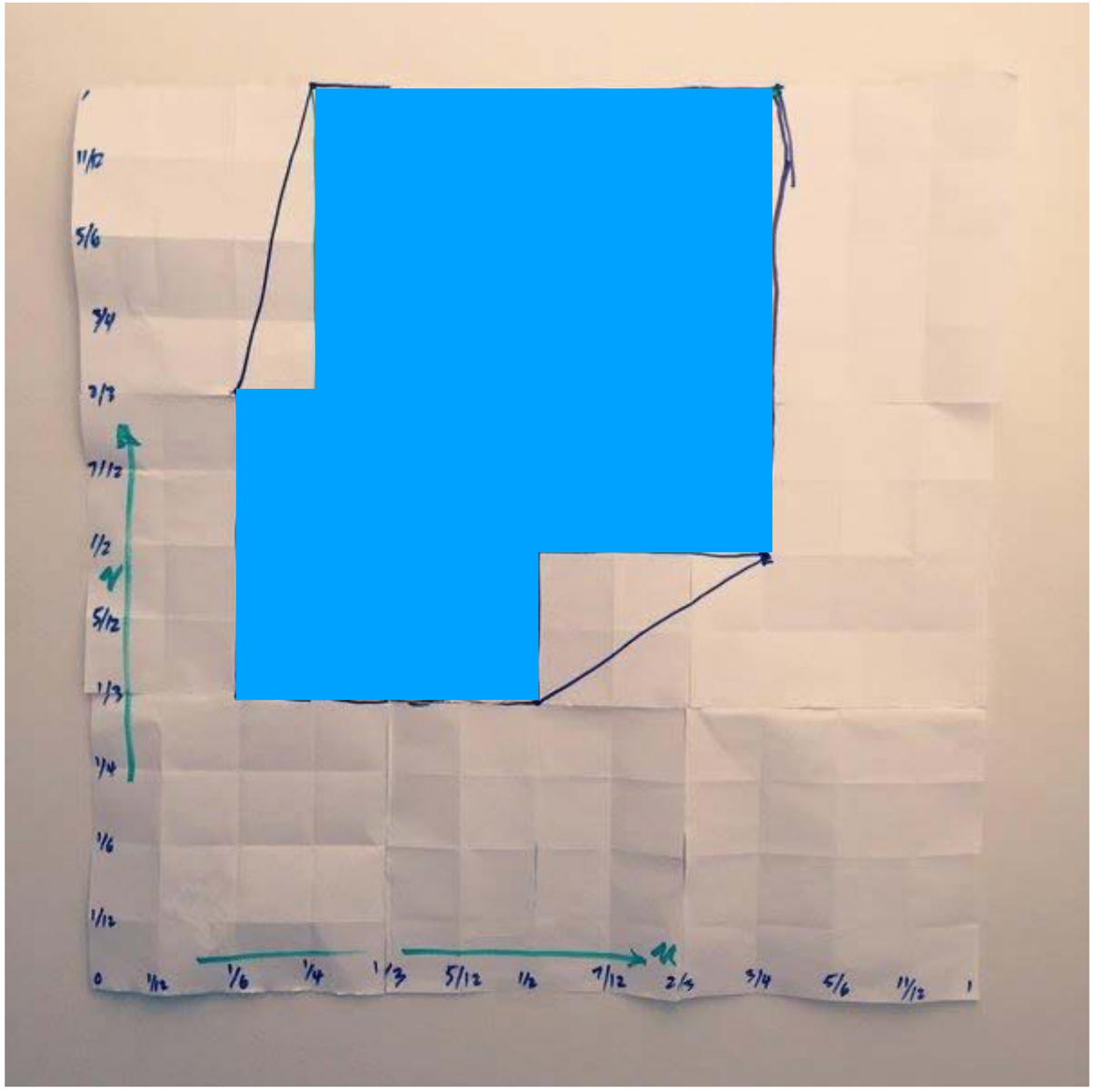


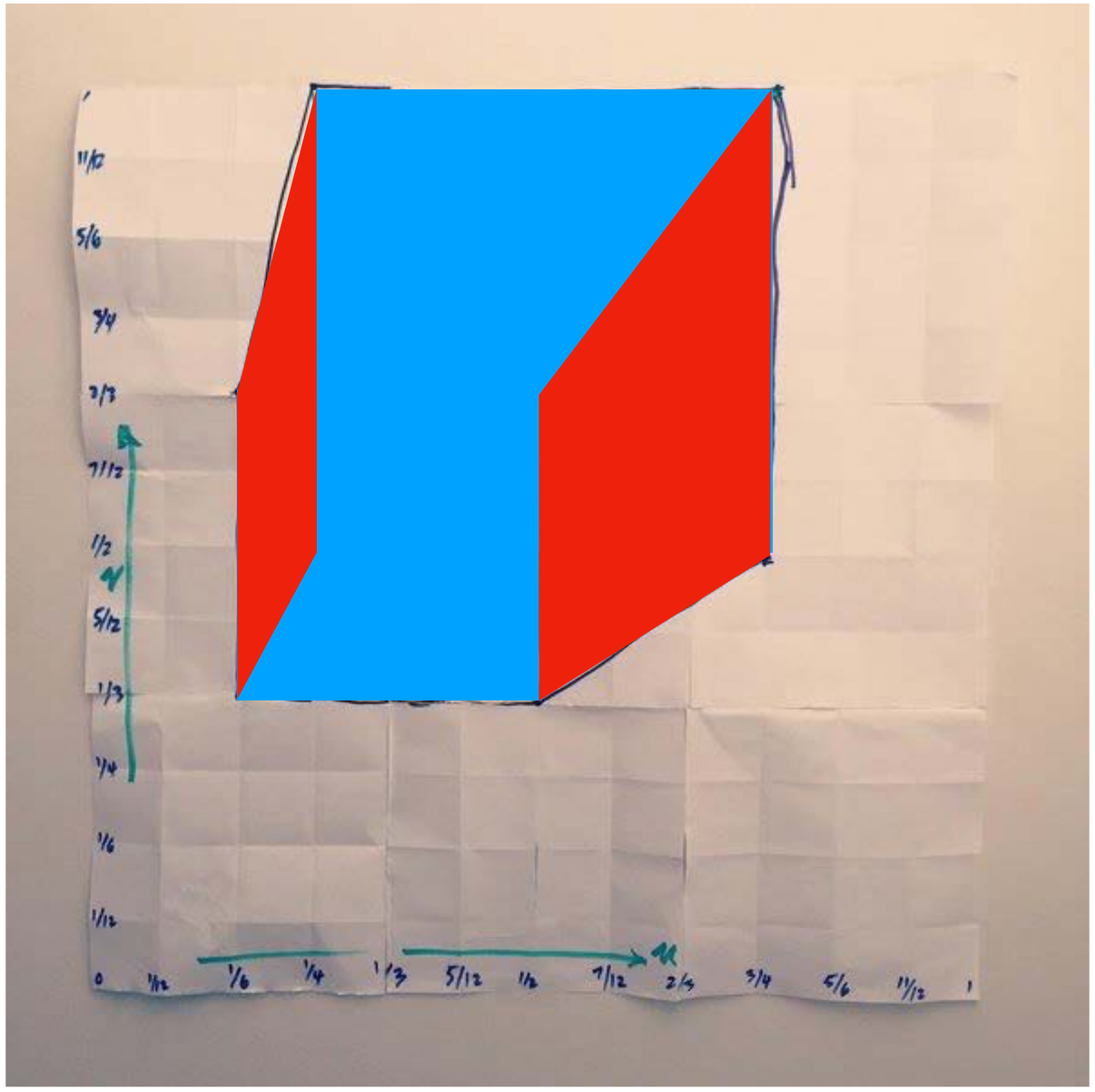
- Object space → world space
- World space → camera space
- Camera space → projection plane (division by *z*)
- Projection plane → NDC
- NDC → screen coordinates

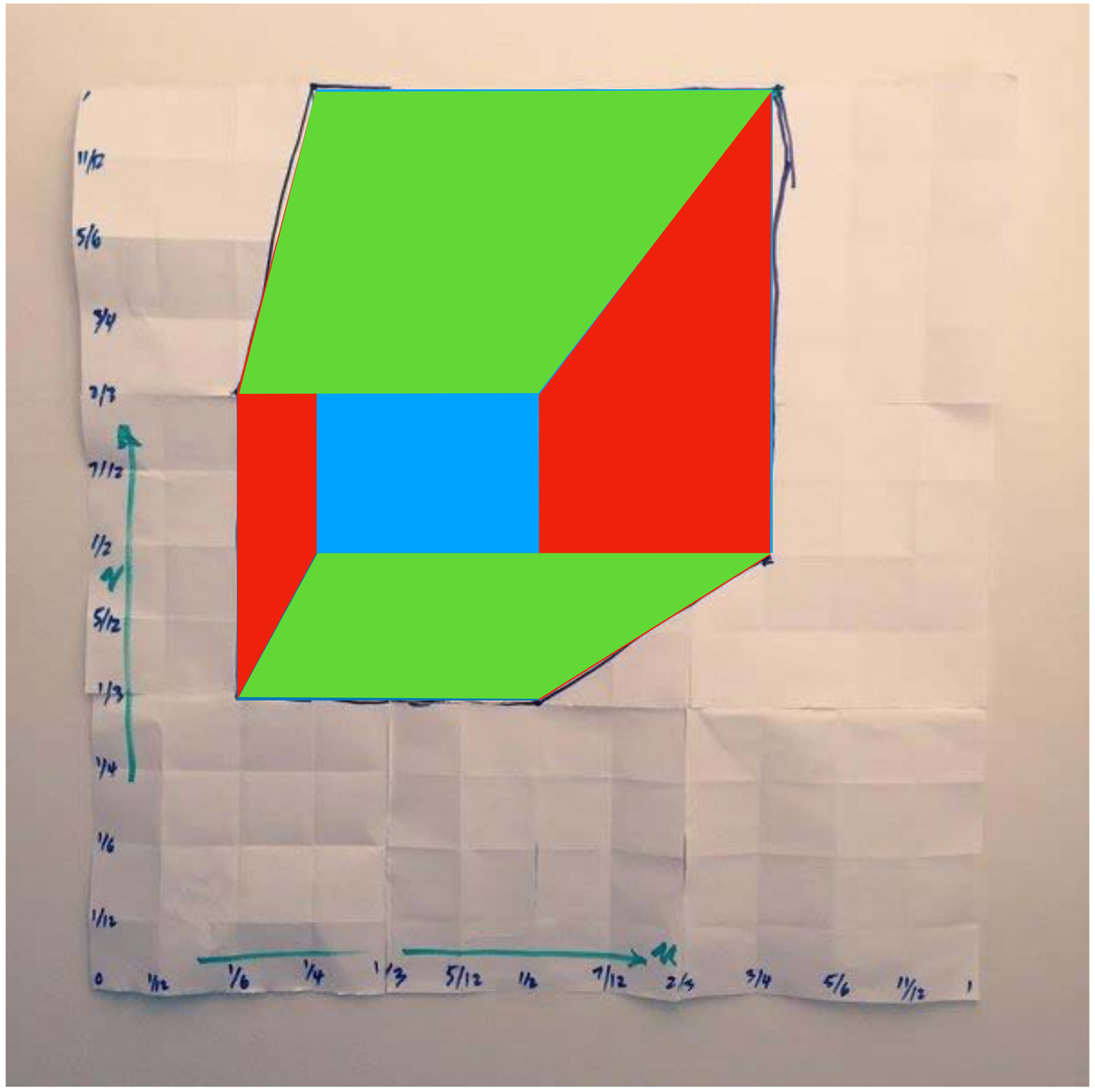
Two problems:

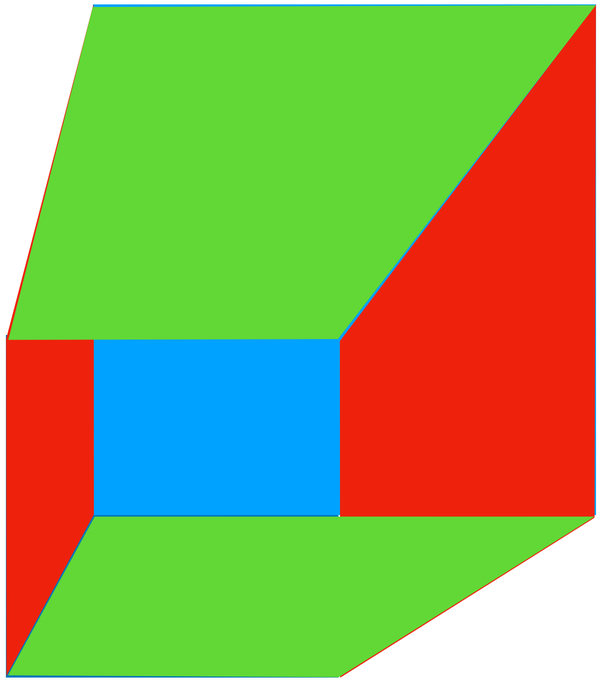
- Every step is a matrix, **except perspective division**.
- Final result has lost depth information (the *z* coordinate): don't know which points are in front of which


11/12
5/6
3/4
2/3
7/12
1/2
5/12
1/3
1/4
1/6
1/12
0
1/12
1/6
1/4
1/3
5/12
1/2
7/12
2/3
3/4
5/6
11/12
1


11/12
5/6
3/4
2/3
7/12
1/2
5/12
1/3
1/4
1/6
1/12
0
1/12
1/6
1/4
1/3
5/12
1/2
7/12
2/3
3/4
5/6
11/12
1

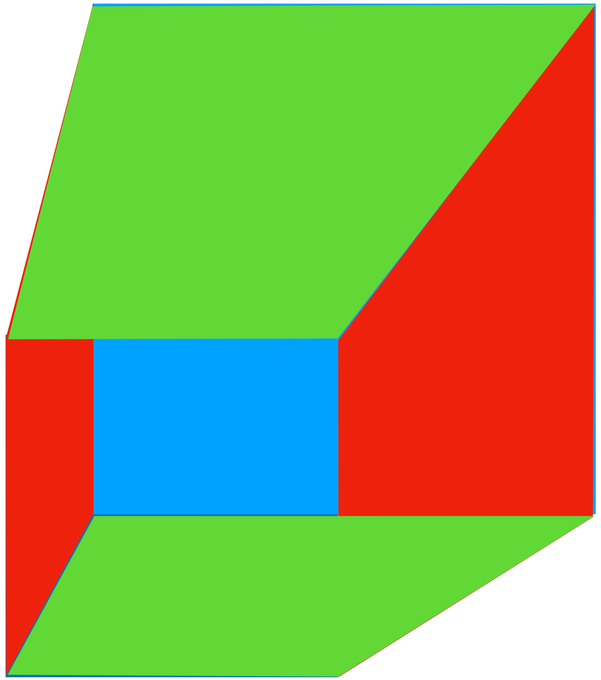

11/12
5/6
3/4
2/3
7/12
1/2
5/12
1/3
1/4
1/6
1/12
0
1/12
1/6
1/4
1/3
5/12
1/2
7/12
2/3
3/4
5/6
11/12
1


11/12
5/6
3/4
2/3
7/12
1/2
5/12
1/3
1/4
1/6
1/12
v
u
0
1/12
1/6
1/4
1/3
5/12
1/2
7/12
2/3
3/4
5/6
11/12
1

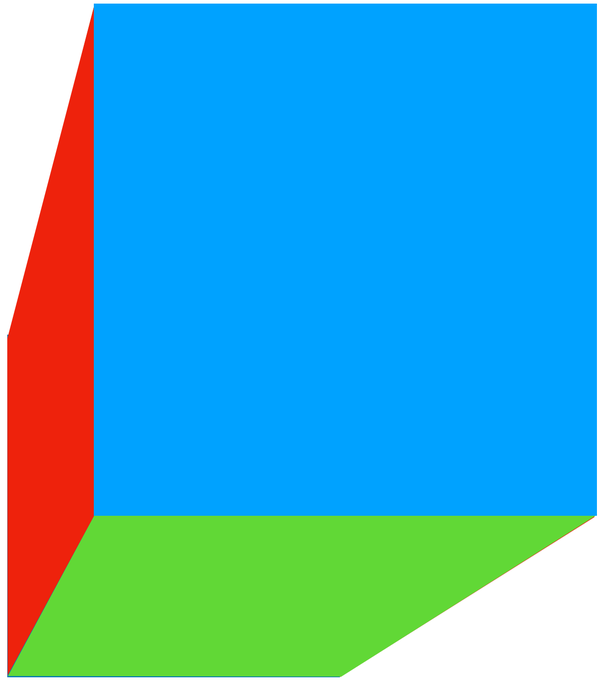


Visibility a.k.a. hidden surface removal

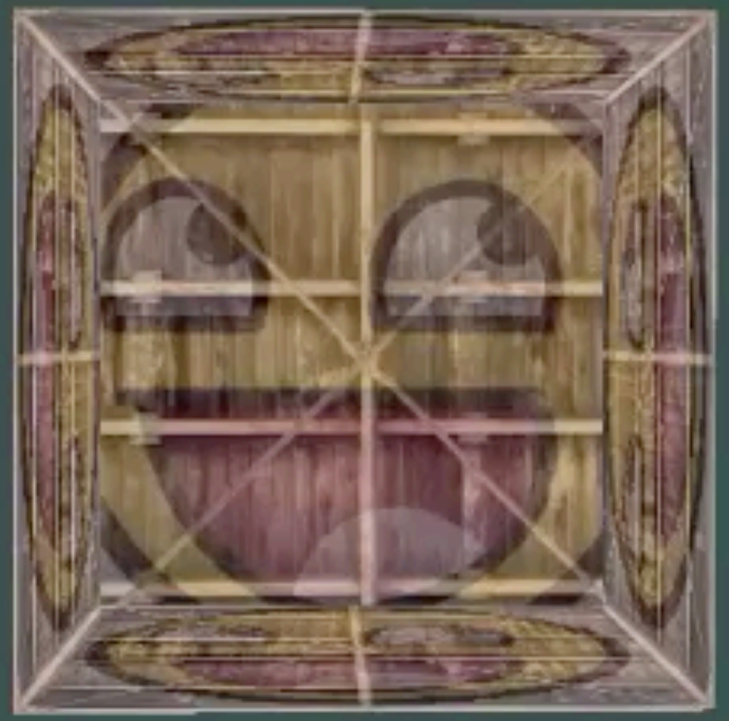
Which surfaces are visible? Those that are not hidden by **nearer** surfaces.



Triangles drawn without considering depth / visibility



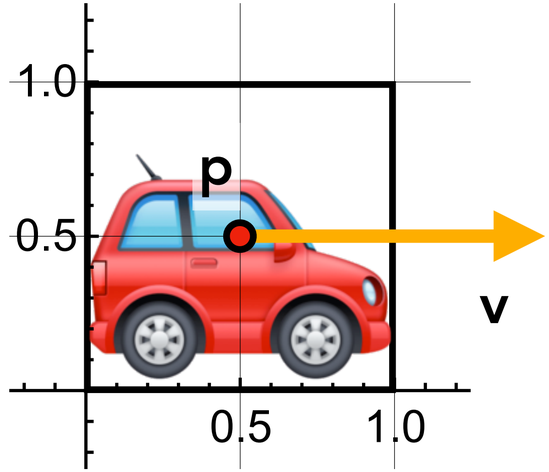
Correct result

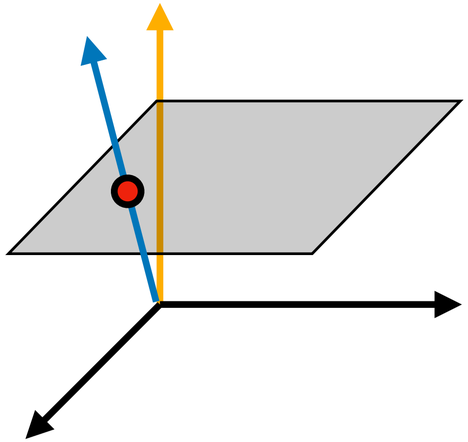


Homogeneous coordinates revisited

Recall points vs. vectors: $\mathbf{p} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$, $\mathbf{v} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$

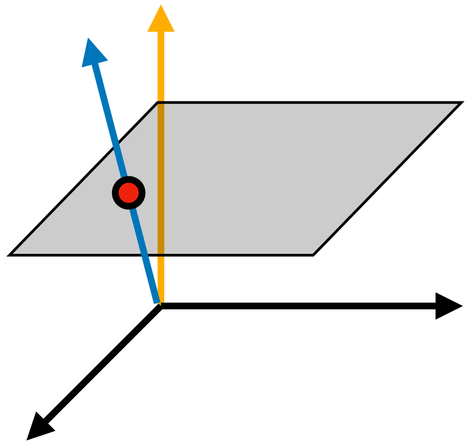
Let's generalize: points can have **any** $w \neq 0$

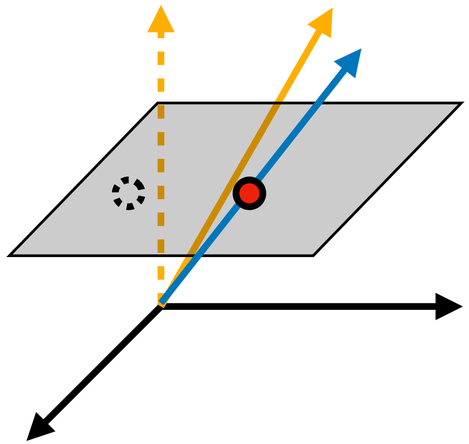




Any point in homogeneous coordinates $\hat{\mathbf{p}} = \begin{bmatrix} x \\ y \\ w \end{bmatrix}$ with $w \neq 0$

corresponds to the 2D point $\mathbf{p} = (x/w, y/w)$





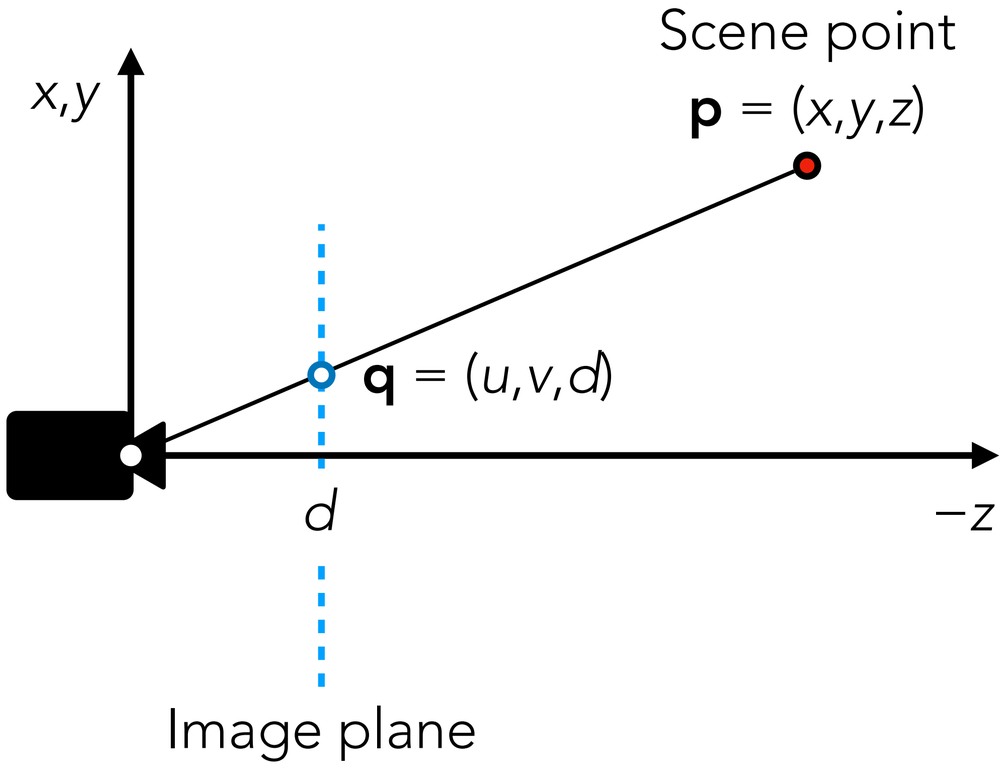
The main idea: Points in 2D correspond to **lines through the origin** in 3D!

All points $\hat{\mathbf{p}} = \begin{bmatrix} cx \\ cy \\ c \end{bmatrix}$ on a line represent the **same** point

$\mathbf{p} = (x, y)$ where the line meets the plane $w = 1$

Analogy: Various tuples (2,4), (−1,−2), (5,10), … all represent the same rational number ½

Linear and affine transformations still work as before!



Perspective projection: $(x,y,z) \rightarrow (xd/z,\ yd/z)$

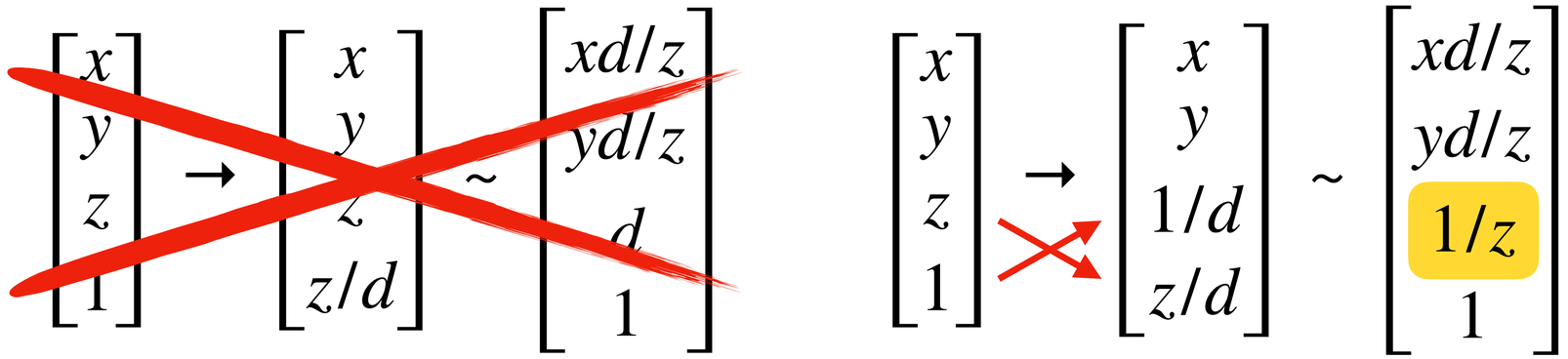
With homogeneous coordinates:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} \sim \begin{bmatrix} xd/z \\ yd/z \\ d \\ 1 \end{bmatrix}$$

Corresponding matrix: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$

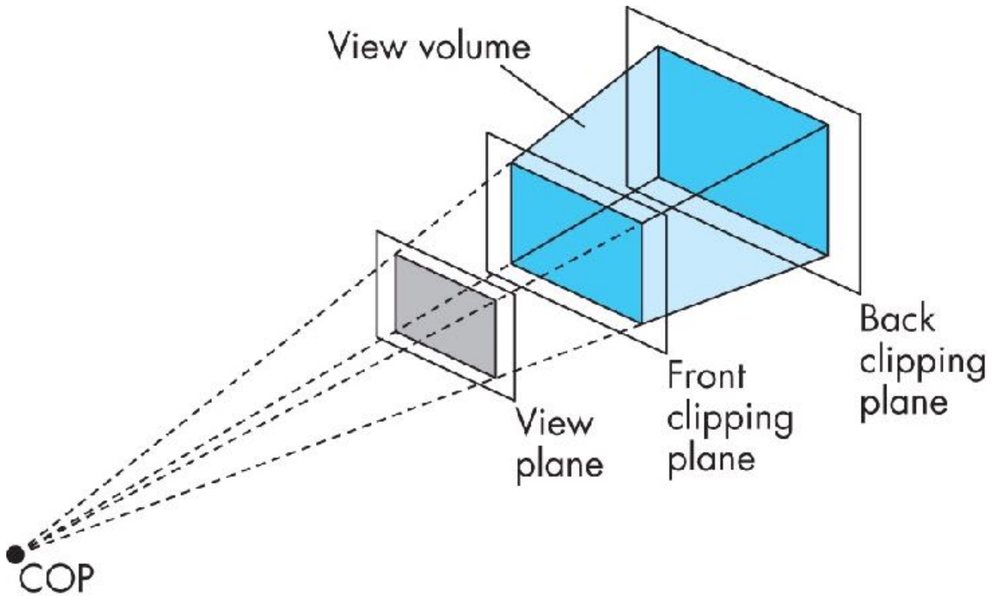
Hang on, we've still lost depth information.

To retain depth information, let's copy *w* into the *z*-coordinate:



Matrix: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/d \\ 0 & 0 & 1/d & 0 \end{bmatrix}$

The view frustum



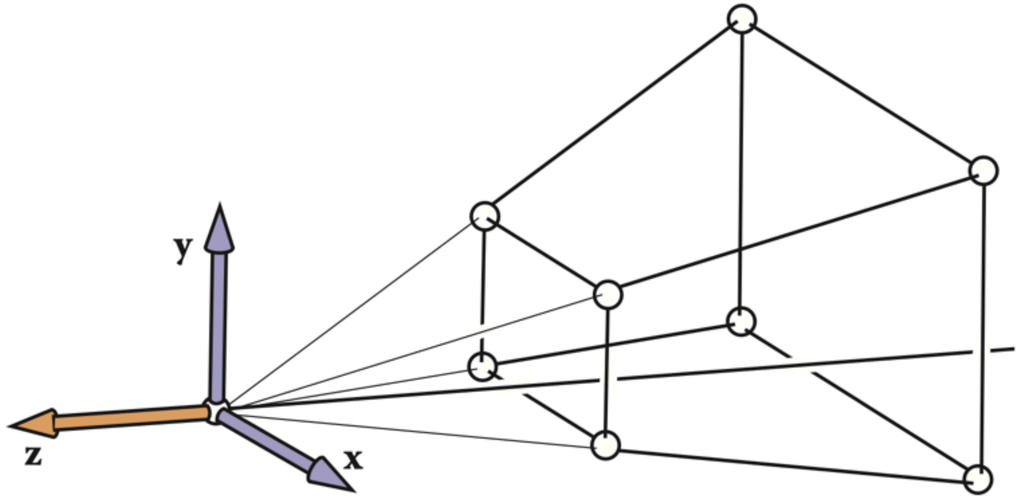
Angel & Shreiner, *Interactive Computer Graphics*

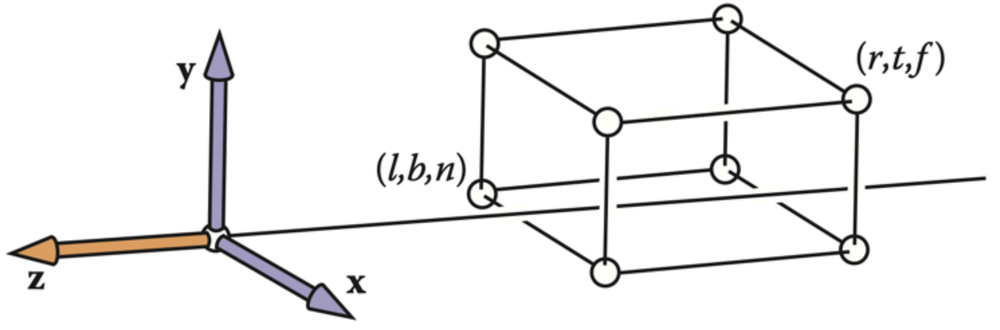
In theory, horizontal and vertical angles of view define an infinite **view cone**

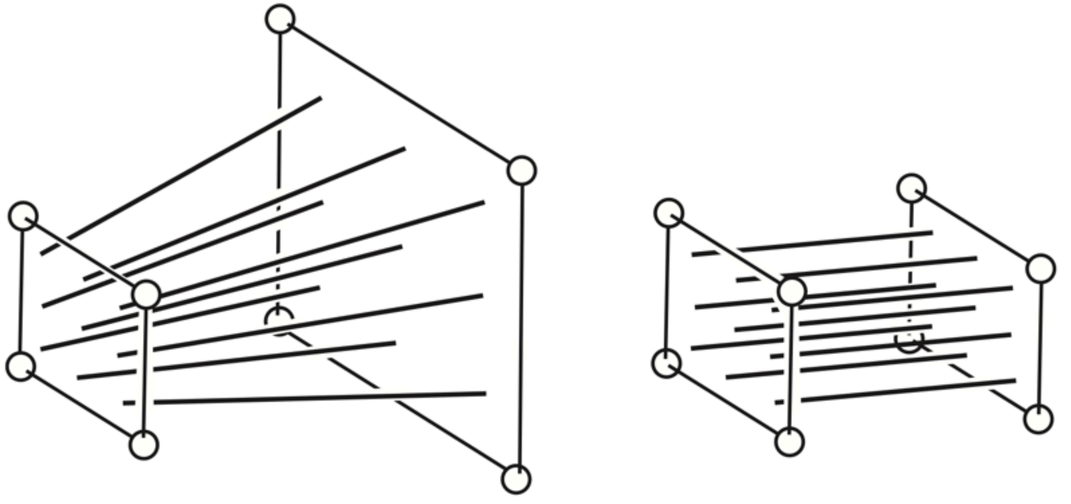
In practice, cut off at near and far "clipping planes": **view frustum**

Why?

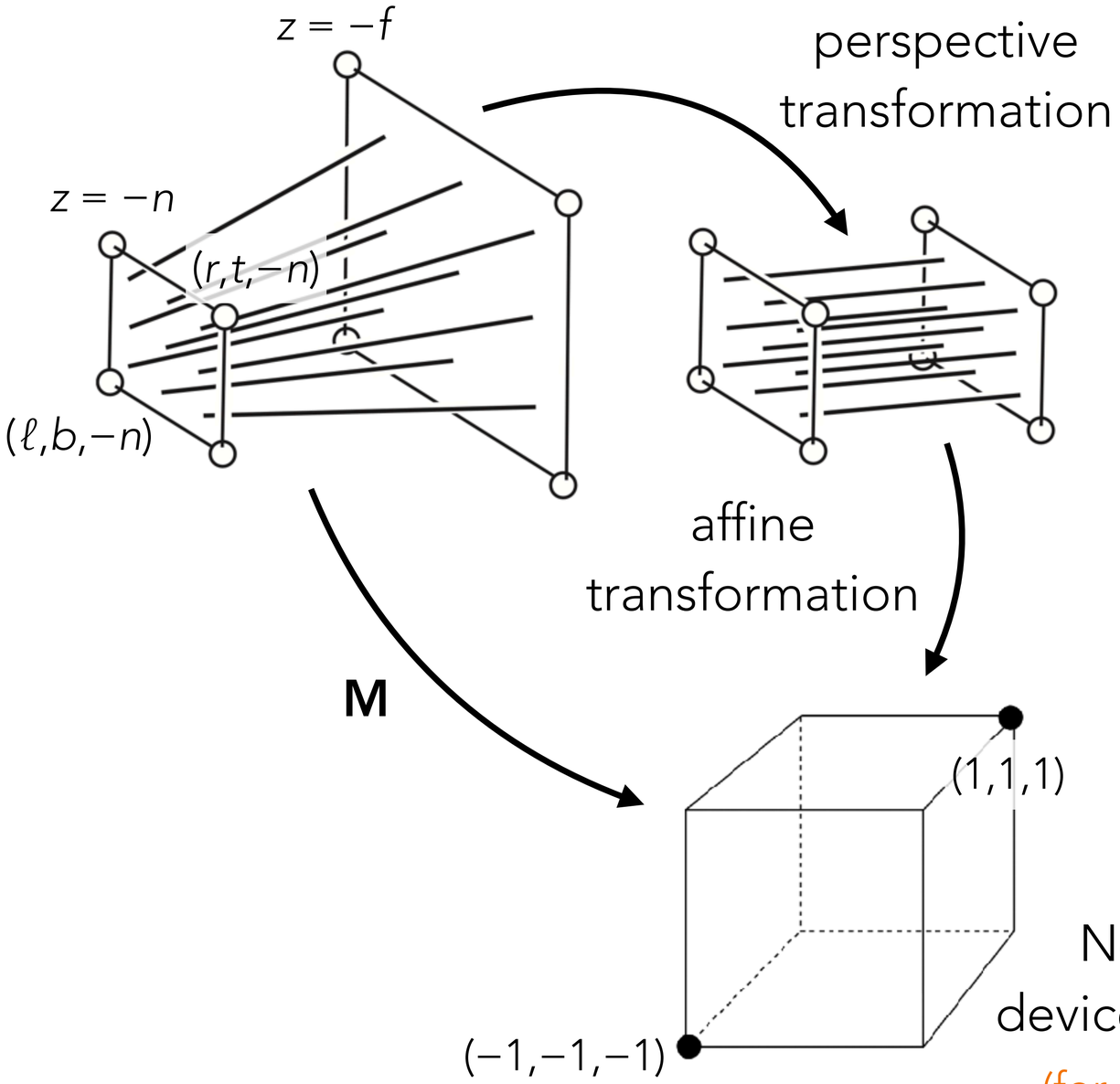
- Exclude objects behind the camera
- Finite precision of depth coordinate (we'll see why later)





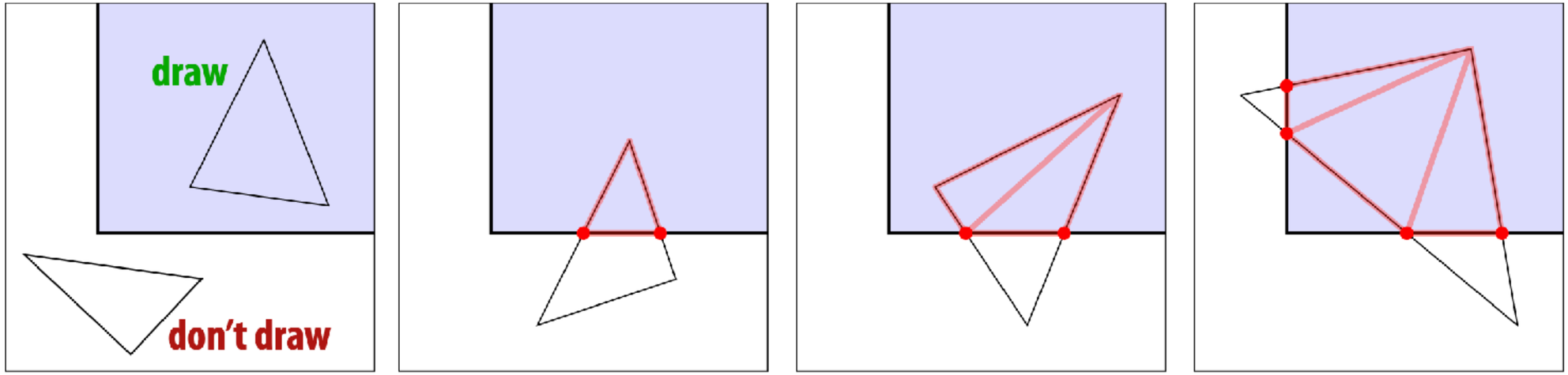


Marschner & Shirley, *Fundamentals of Computer Graphics*



$$\mathbf{M} = \begin{bmatrix} \frac{2|n|}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2|n|}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{|n|+|f|}{|n|-|f|} & \frac{2|n||f|}{|n|-|f|} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Clipping



Keenan Crane

- Discard triangles outside view frustum
- Clip triangles partially intersecting view frustum

Usually implemented in homogeneous coordinates (before division)