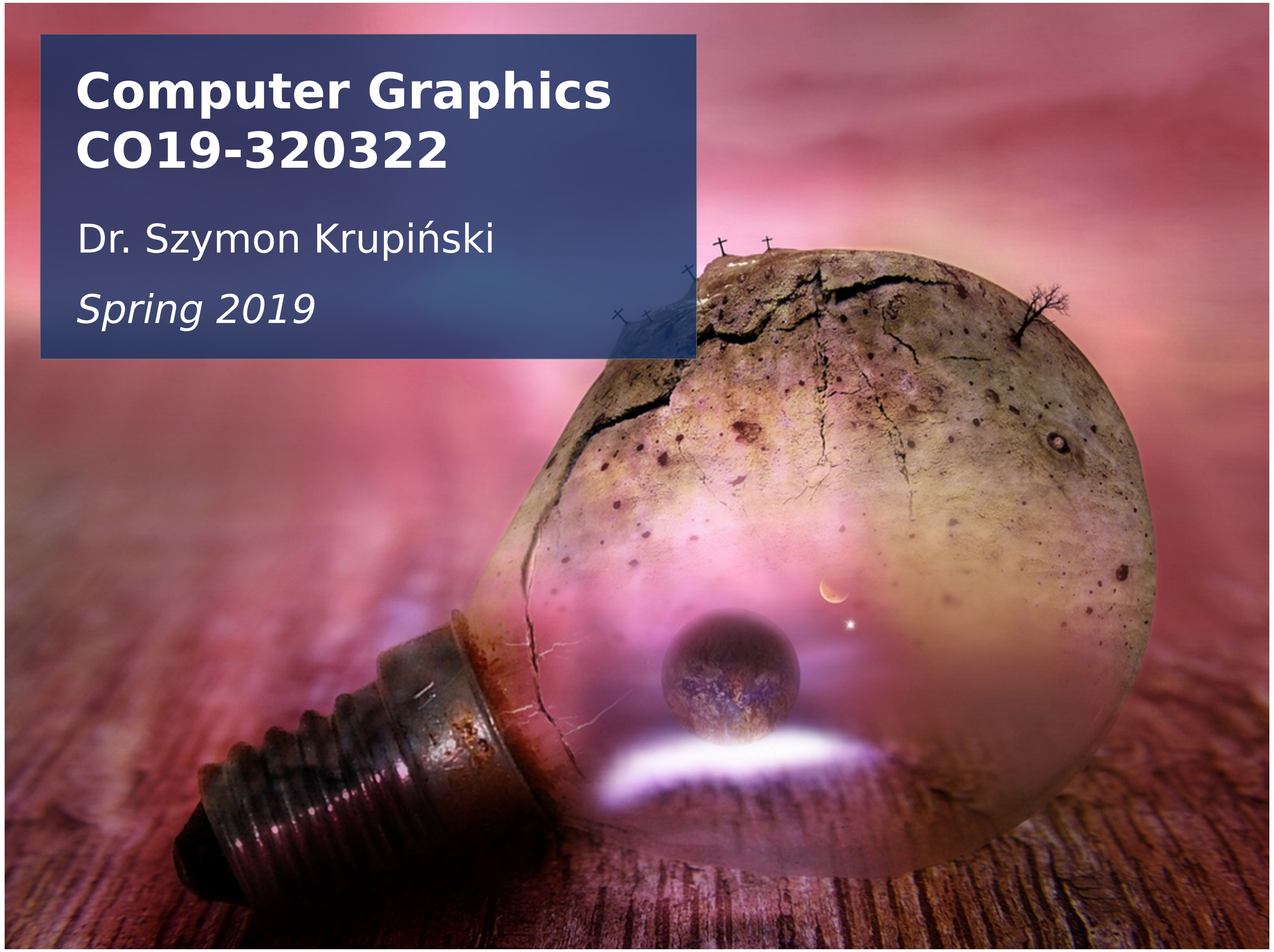
Lecture 3: Projections

JACOBS
UNIVERSITY



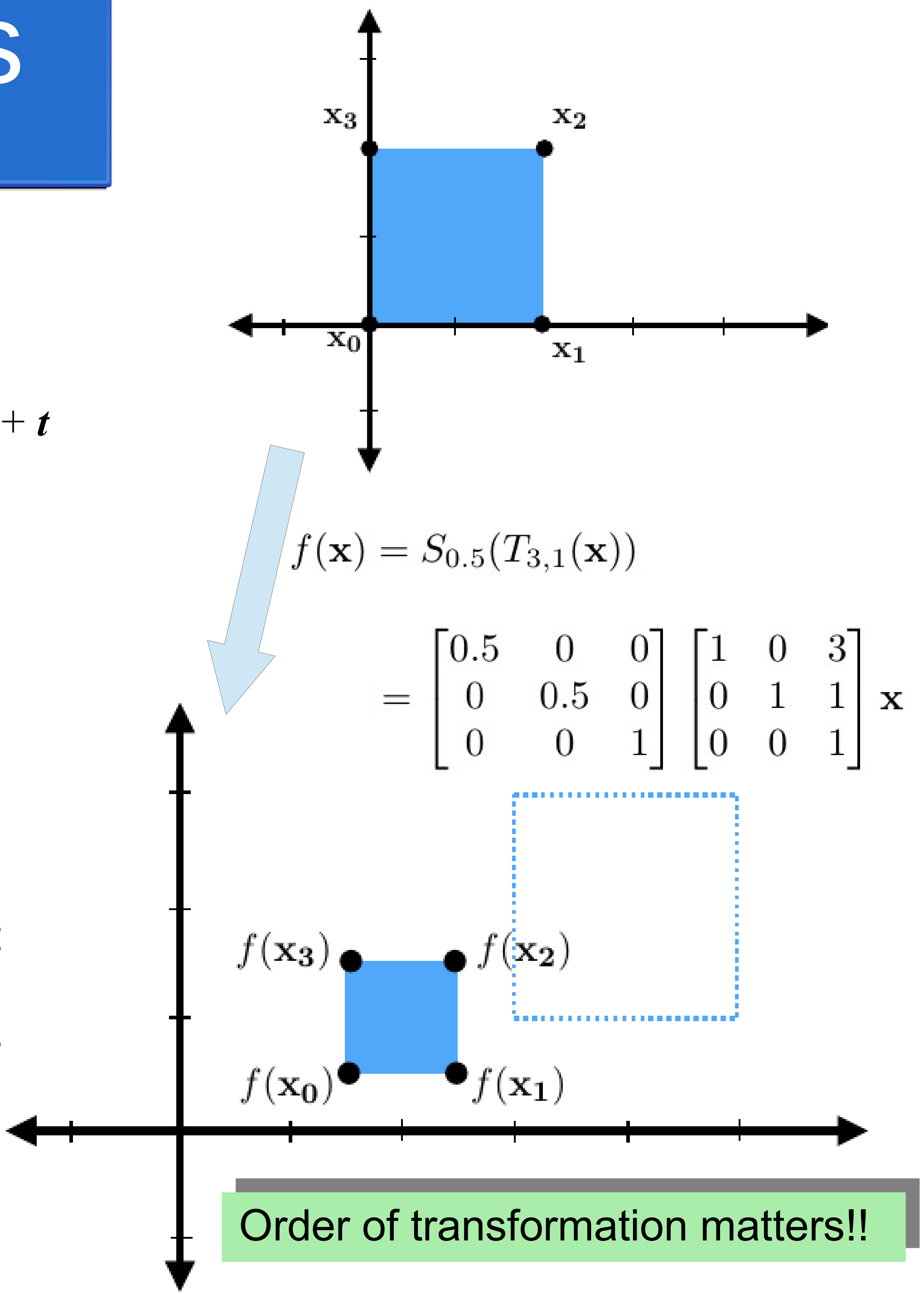
Computer Graphics
CO19-320322

Dr. Szymon Krupiński

Spring 2019

Recap of transformations

- Expressing the position of points in space using homogeneous coordinates is a great trick to represent transformations efficiently (matrix multiplication)
  - ...including affine transformations which take the form $f(\boldsymbol{x}) = \boldsymbol{Ax} + \boldsymbol{t}$ in Cartesian coordinates
- In n-dimensions, we add a n+1th coordinate w. in 3-D: points before: $\boldsymbol{x}_C = (x, y, z)$ and after: $\boldsymbol{x}_H = (x, y, z, 1)$
  - … actually, (wx, wy, wz, w) – the representation is not unique in homogeneous coordinates but always points to the same Cartesian point (x, y, z) (remember the line crossing the $w = 1$ plane?)
  - vectors before: $\boldsymbol{v}_C = (x, y, z)$ and after: $\boldsymbol{v}_H = (x, y, z, 0)$
  - It makes much sense if you think of them as drawn between two points: $\boldsymbol{v}_H = (x_{P1}, y_{P1}, z_{P1}, 1) - (x_{P2}, y_{P2}, z_{P2}, 1) = (x_{P1} - x_{P2}, y_{P1} - y_{P2}, z_{P1} - z_{P2}, 0$
- We can write and combine transformations like so:



$$f(\mathbf{x}) = S_{0.5}(T_{3,1}(\mathbf{x}))$$

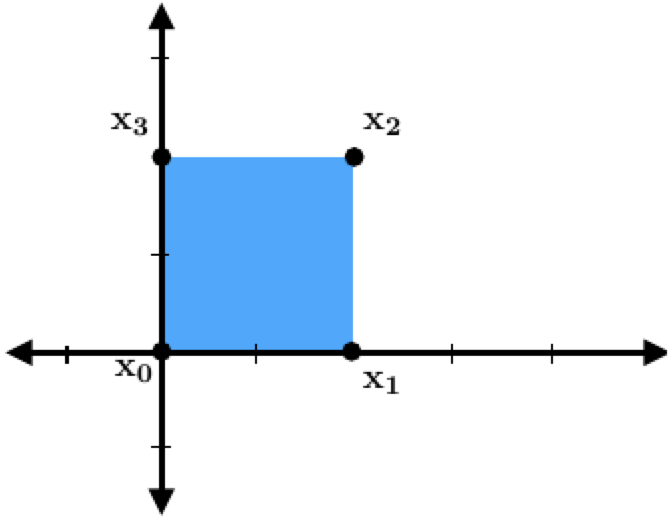
$$= \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{x}$$

Order of transformation matters!!

Transformations

- Every affine transformation can be split into 3 consecutive steps of
  - Scaling
  - Rotation
  - Translation
  - $f(\mathbf{x}) = \boldsymbol{A}\mathbf{x} + \boldsymbol{t} \rightarrow f(x) = \boldsymbol{RS}\mathbf{x} + \boldsymbol{t}$
  - Or, in homogeneous coordinates:

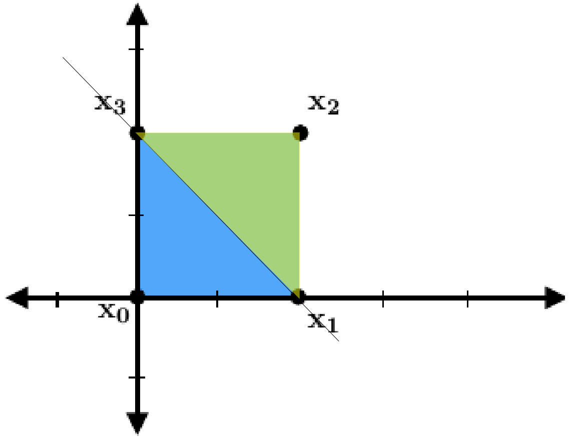
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11}s_x & r_{12} & r_{13} & t_x \\ r_{21} & r_{22}s_y & r_{23} & t_y \\ r_{31} & r_{32} & r_{33}s_z & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

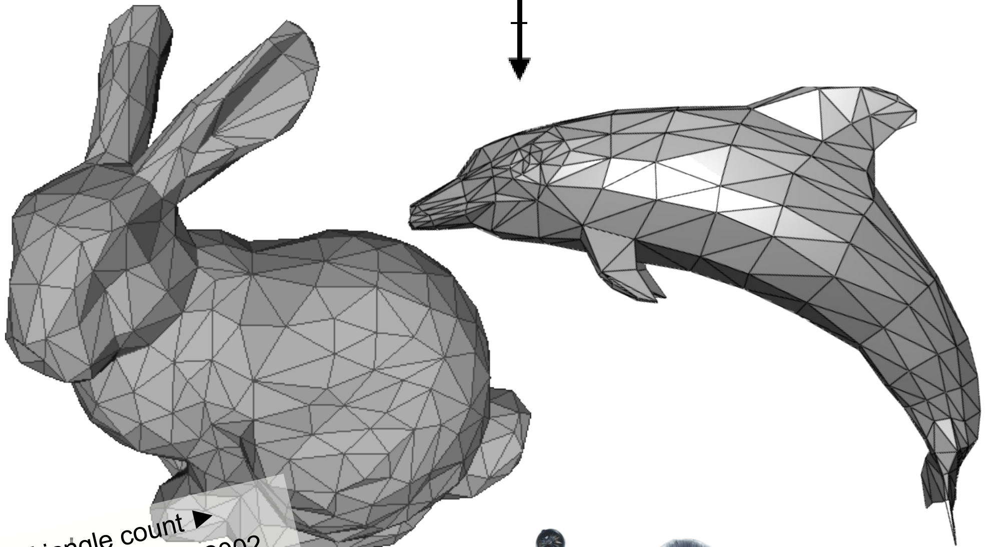


Beyond modelling?

- We are still in the modelling domain – we have been playing with **points** in space
  - Well, we made our favourite blue shape, didn't we? We just implicitly added **edges** and obtained a **polygon**
  - What is the simplest shape which we can make beyond an edge?
  - A **triangle** is a good building block. Any complex 2-D polygon can be triangulated – subdivided into triangles
  - Triangle is one of the fundamental primitive in Computer Graphics
  - Using triangles we can build complex 3-D meshes – it's just a question how many of them we want to use...
  - We will pause our modelling at this stage and see what is awaiting us at the later stages!

- always planar
- well defined interior (convex)
- mathematical tricks apply (barycentric coordinates)





◀ Triangle count ▶

Super Mario Sunshine 2002
Mario - **1500**

Tomb Raider (Lara)
TR1 - **230**
TRIII - **300**
Angel of Darkness - **4400**
Legend - **9800**
Underworld - **32816**

The Witcher
Geralt - **10875**

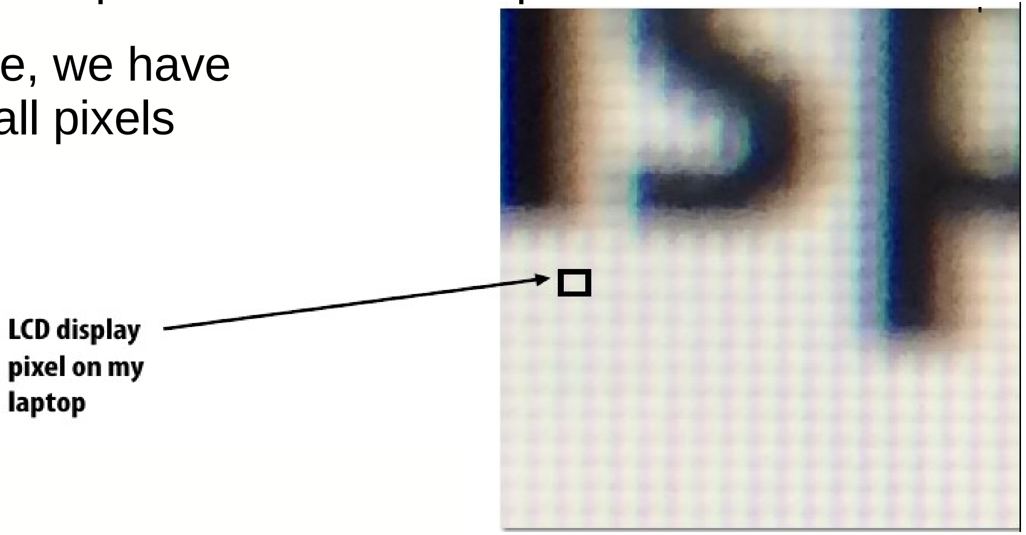
Tekken 6
Jin Kazama – **20240**

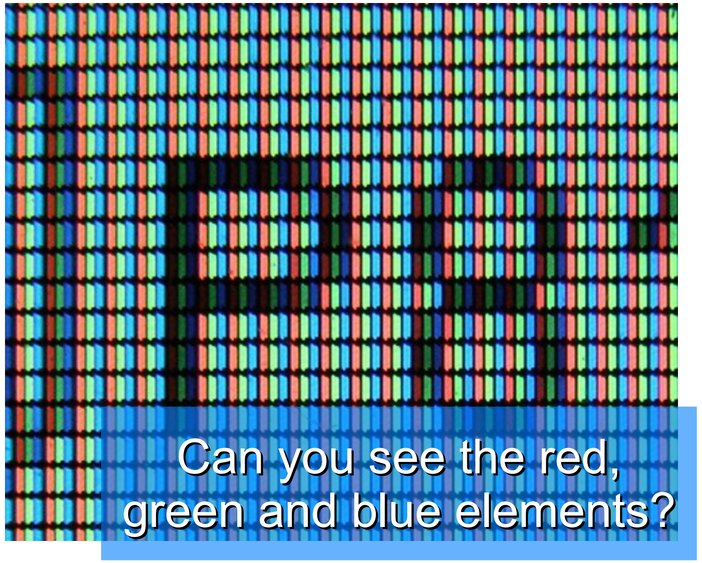
Uncharted 2
Chloe - **45000**, each subsequent LOD has halved count (3 LODs)
Drake - **37000**
Drake's Hair - **4000**

Between 3-D and 2-D

- We live in a 3-D world but so far our computer screens are 2-D. How does one obtain 2-D version of 3-D objects?
- What exactly do we need at the output?
  - A computer screen might have a resolution, let’s say 1920x1080px (or “1080p” or “Full HD” or “FHD” or BT.709…); it is capable of displaying a particular range of colours, for example: “True color” (24-bit)
  - If you look closer, a pixel actually consists of sub-pixels in most of display technologies, which produce colour components
- To obtain a full image, we have to define values for all pixels

In digital imaging, a **pixel** is a physical point in a raster image, or the smallest addressable element in an all points addressable display device; so it is the smallest controllable element of a picture represented on the screen.

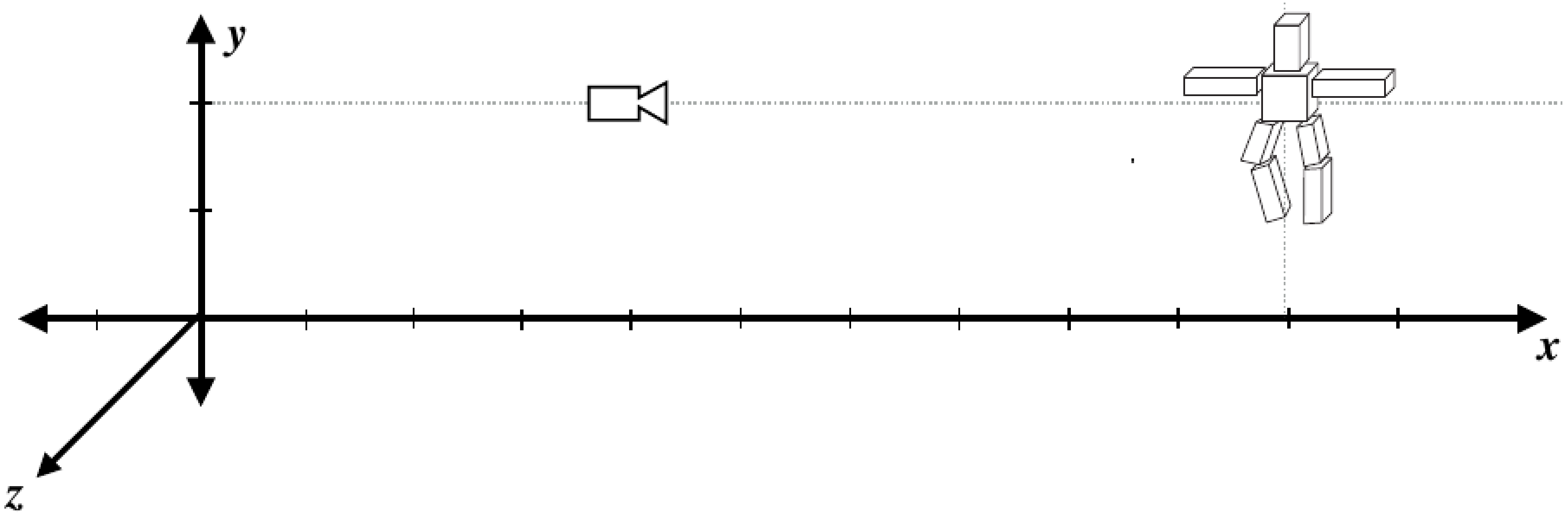

LCD display pixel on my laptop


Can you see the red, green and blue elements?

Between 3-D and 2-D

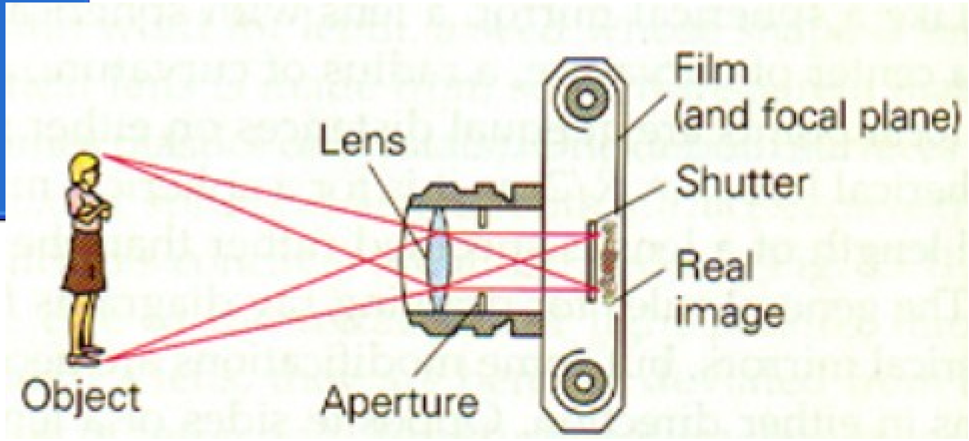
- In a real world we would use a photo or video camera to obtain a 2-D representation

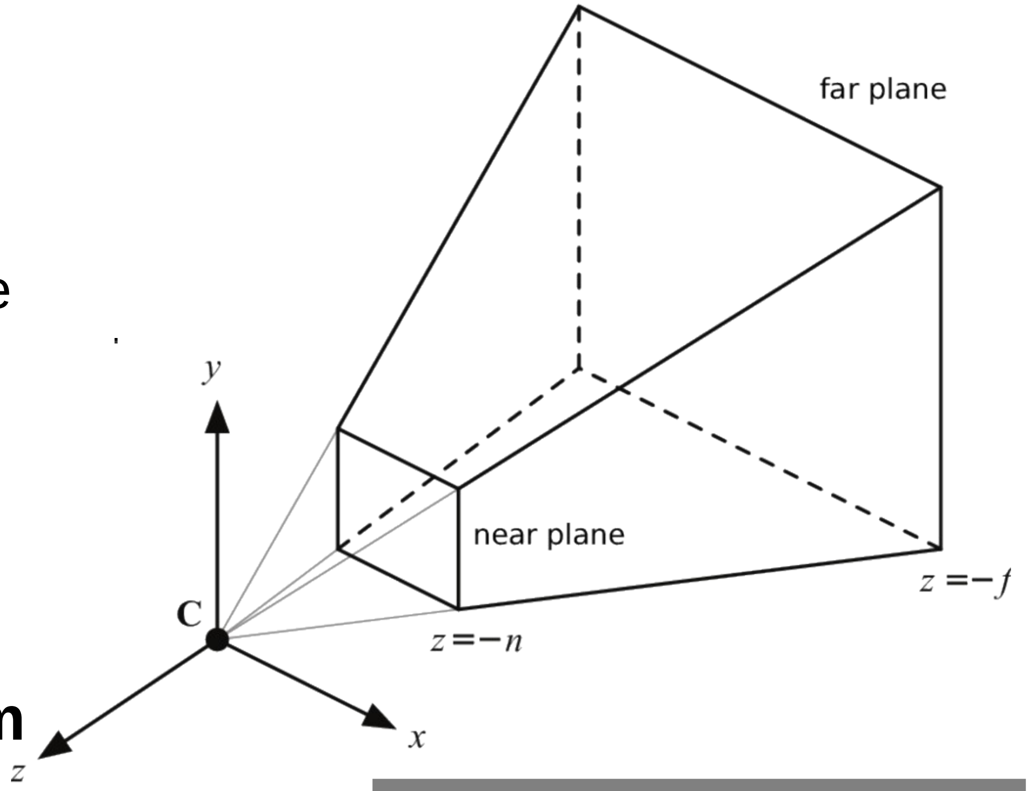
➔ Let us then model a camera in our world of 3-D points!



Between 3-D and 2-D

- In a typical camera light passes through the objective and falls on a CCD sensor or light-sensitive film
- The objective usually defines whether the camera can capture objects standing far apart (short focal length, fisheye) or just a very narrow scene (long focal length, telephoto)
- How wide can the scene be depends on the distance from the camera – if you cannot make a scene fit in the image, you intuitively step back
- Object which are too far can become blurred and the same applies to objects which are too close to the objective. You can change this by picking the right objective (teleophoto lens vs macro photography)
- This set of constraints is called the **camera frustrum**

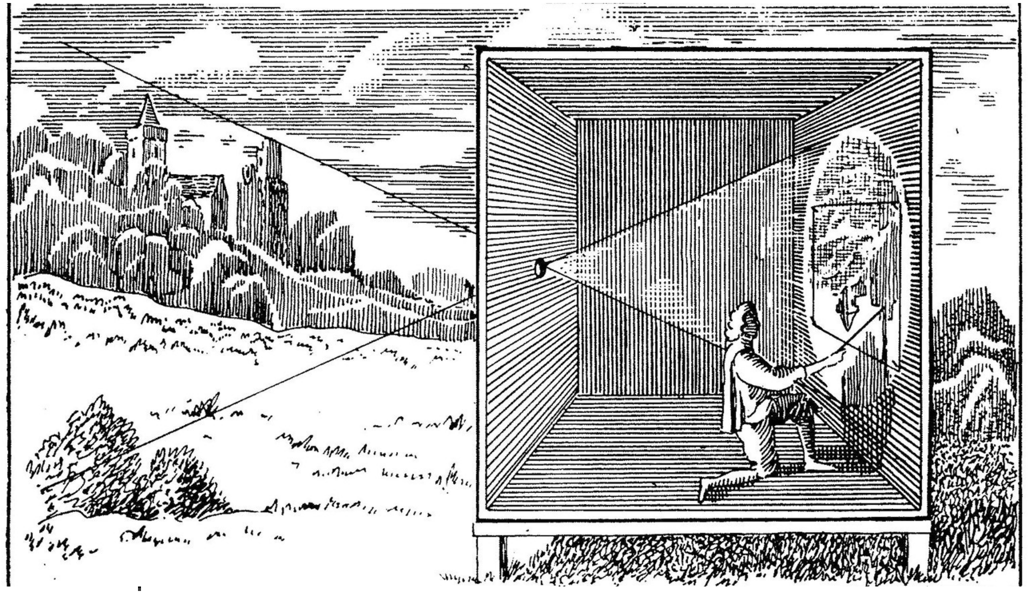




n and f also called “near clipping plane” and “far clipping plane”

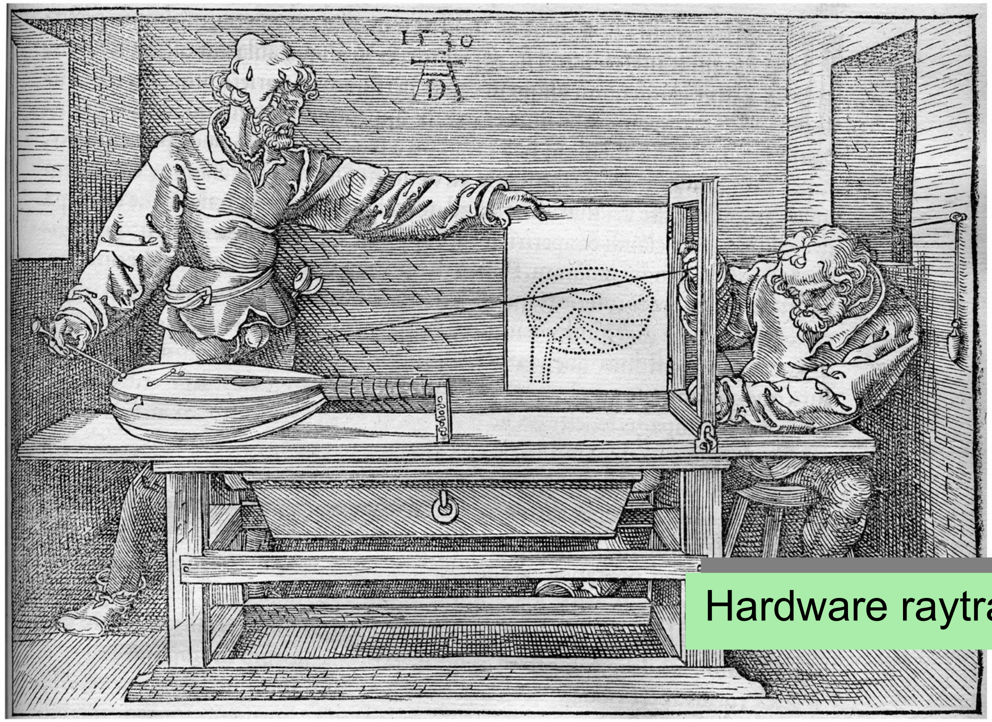
Pinhole camera

- The most popular model of camera in CG is the **pinhole model**
- It uses the same concept that makes the *camera obscura* work. This concept has been known for more than a millenium
- Rays (either produced by a source of light on the object or reflected from it) leaving the points on the objects pass through a very small aperture and form a small inverted image on the wall
- https://www.youtube.com/watch?v=qIp9kItDUh8

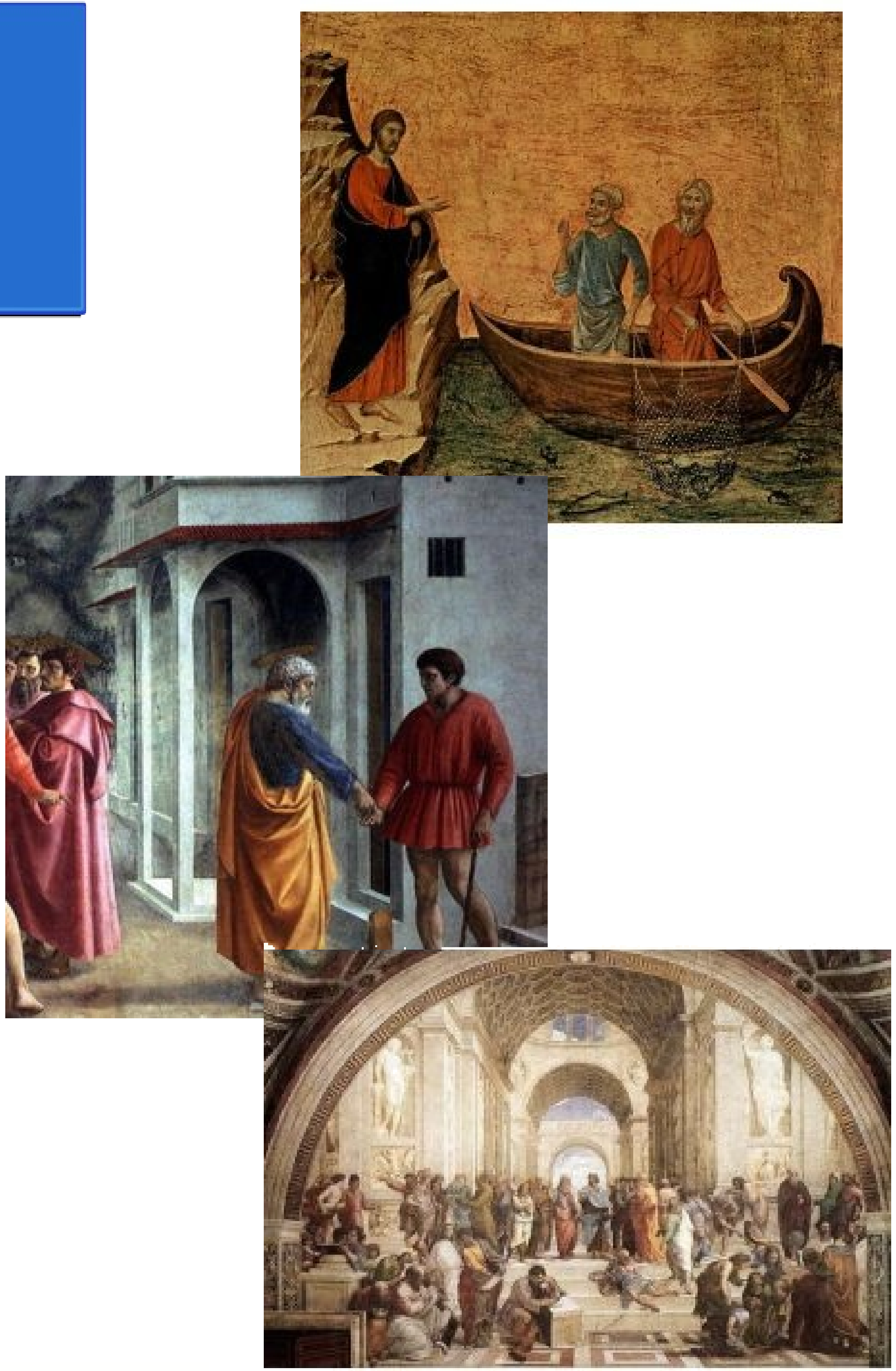


Not a new idea!

- Just like camera obscura, the question of transforming 3-D world to a 2-D image fascinated people for centuries!
- The artistic rules of perspective, for example, became known around the 14th century

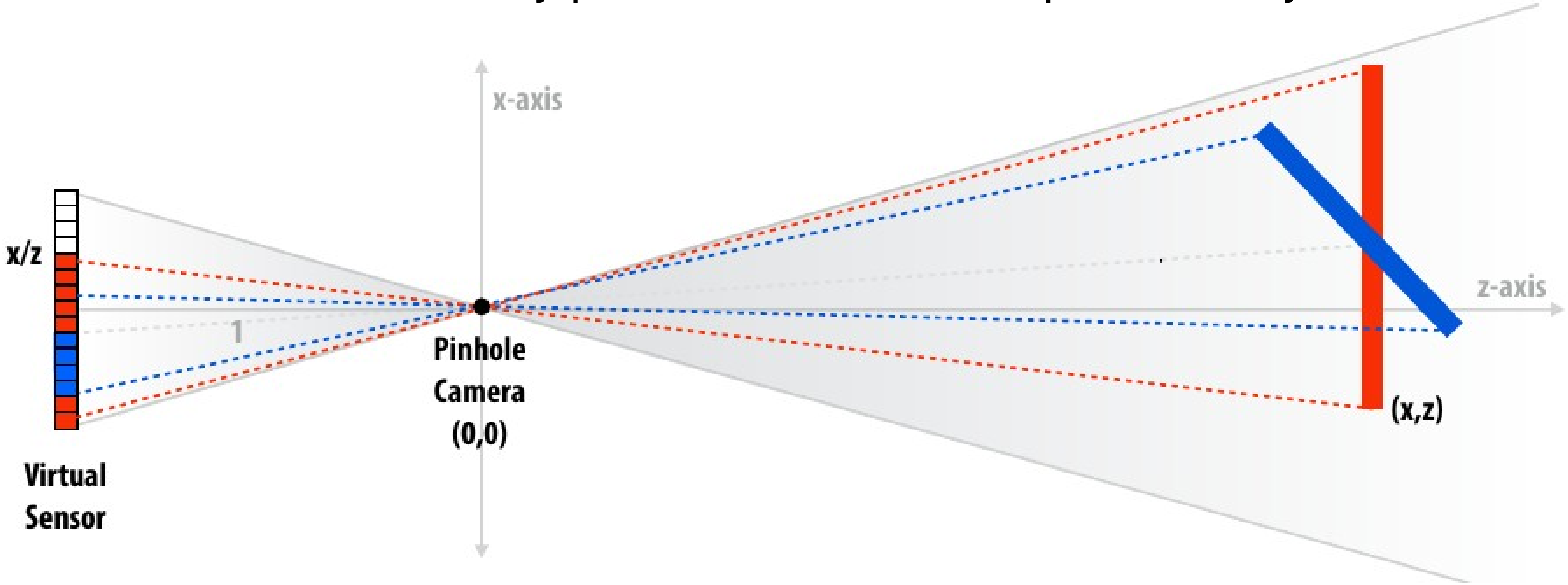


Hardware raytracing!!!



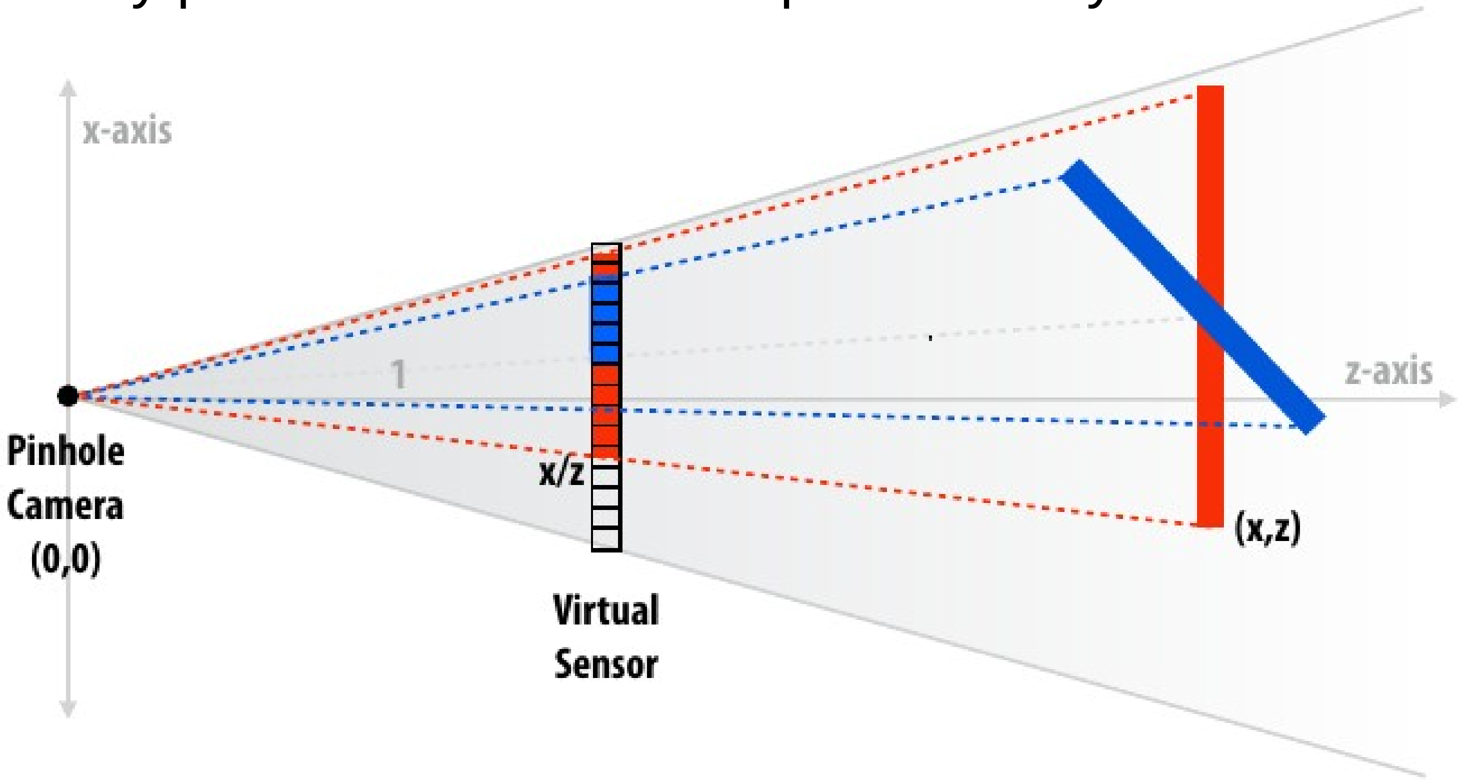
Pinhole camera

- The objective: we want the pixels of the screen light up at points where the light would come from the objects on the way to our eye
- This can be modelled by pinhole model in two equivalent ways:



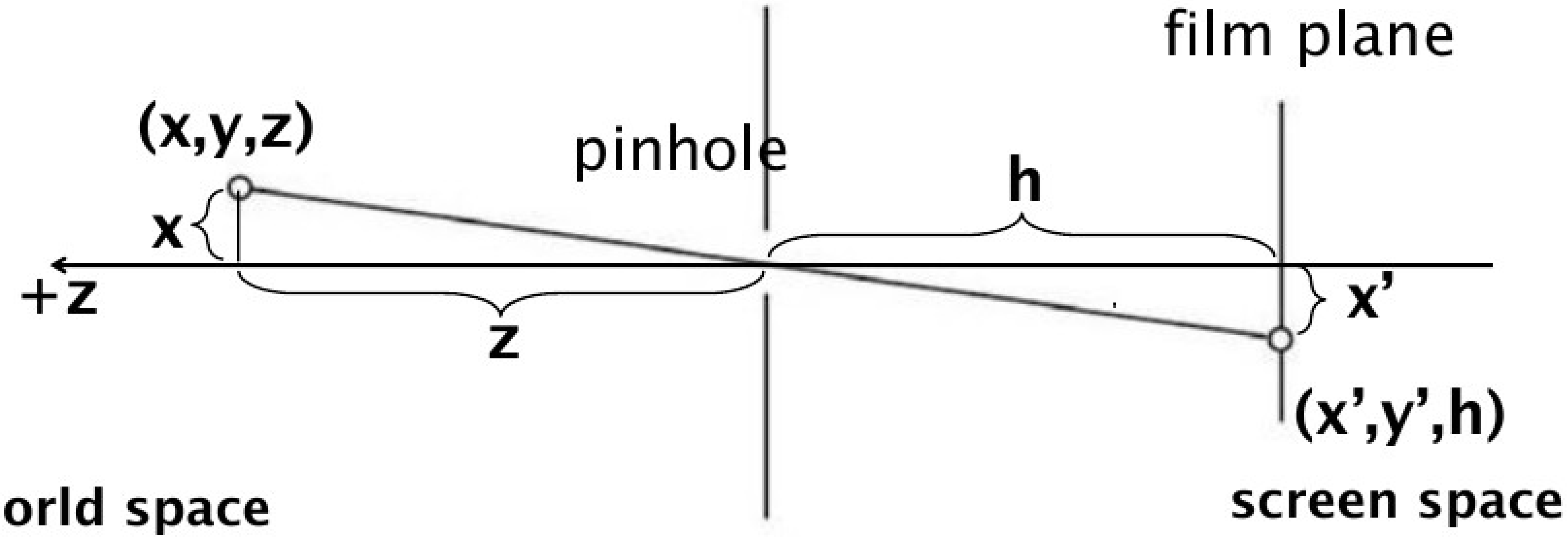
Pinhole camera

- The objective: we want the pixels of the screen light up at points where the light would come from the objects on the way to our eye
- This can be modelled by pinhole model in two equivalent ways:



Perspective projection

- Let’s have a look at the geometry of this model
- By mapping the rays on the virtual image plane, we are applying **perspective projection**



$$\frac{x}{z} = \frac{x'}{h} \qquad \frac{y}{z} = \frac{y'}{h} \quad \Rightarrow \quad x' = h\frac{x}{z} \qquad y' = h\frac{y}{z}$$

Perspective projection

$$x' = h\frac{x}{z} \qquad y' = h\frac{y}{z}$$

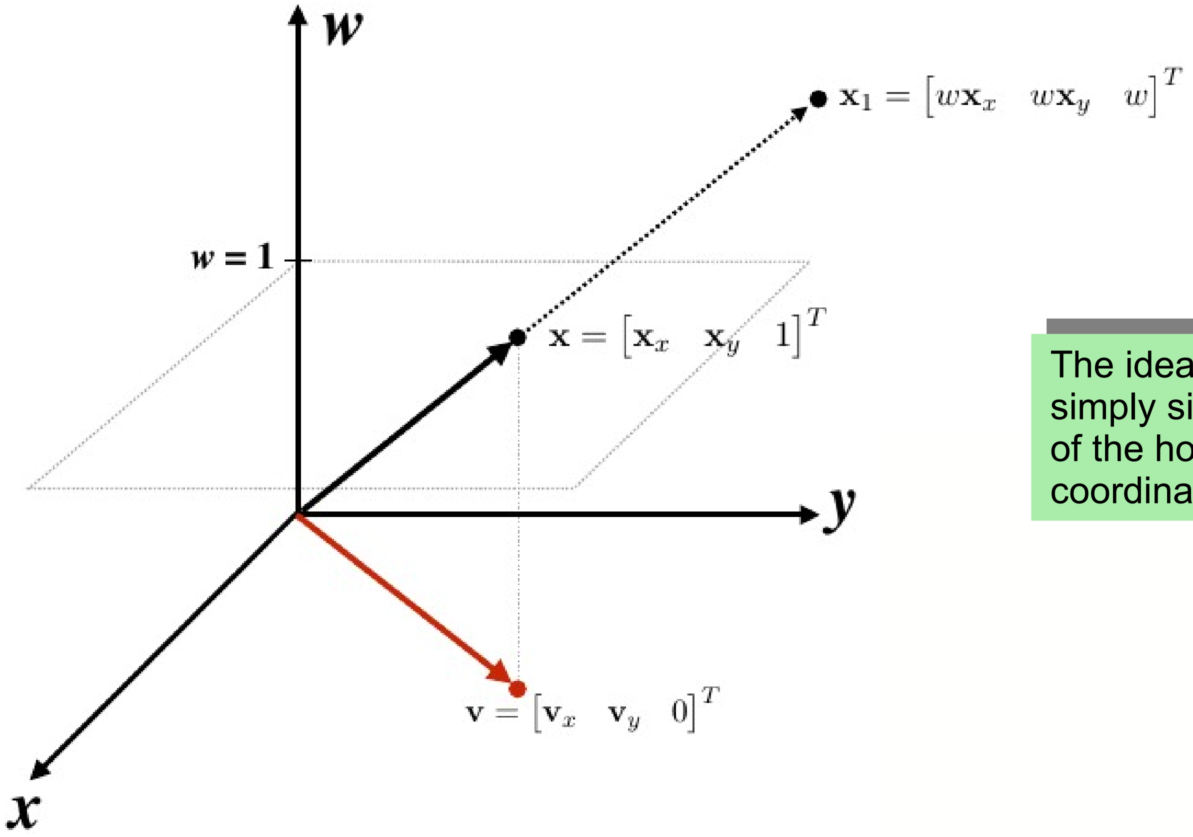
- The division by z definitely a non-linear operation
- We can again turn to homogeneous coordinates to streamline this calculation!
- What is the trick? Normalisation and non-unique character of this type of coordinates

$$\begin{bmatrix} x'w' \\ y'w' \\ z'w' \\ w' \end{bmatrix} = \begin{bmatrix} h & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad \Longrightarrow \quad \begin{aligned} x'w' &= hx \\ y'w' &= hy \\ z'w' &=? \\ w' &= z \end{aligned}$$

Perspective projection

$$x' = h\frac{x}{z} \qquad y' = h\frac{y}{z}$$

- Remember the line?



w
$\mathbf{x}_1 = [w\mathbf{x}_x \quad w\mathbf{x}_y \quad w]^T$
w = 1
$\mathbf{x} = [\mathbf{x}_x \quad \mathbf{x}_y \quad 1]^T$
y
$\mathbf{v} = [\mathbf{v}_x \quad \mathbf{v}_y \quad 0]^T$
x

The idea of projection simply sits at the heart of the homogeneous coordinates!

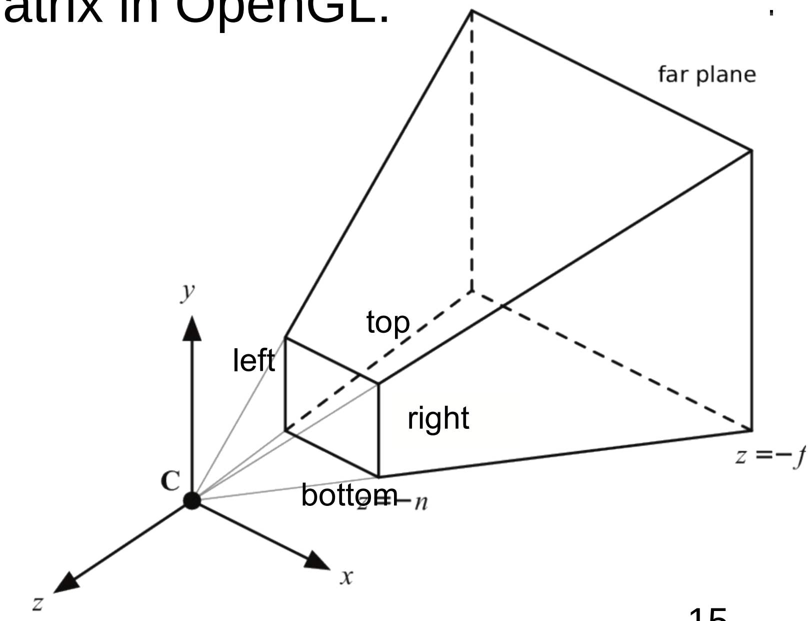
Perspective projection

$$\begin{bmatrix} x'w' \\ y'w' \\ z'w' \\ w' \end{bmatrix} = \begin{bmatrix} h & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Remark: while we need a lot of precision and range in the 3-D point coordinates to represent a big scene and maintain precision in algebraic operations, the result of the projection does not need to be so precise, since we just want to know which pixel to light up. We can get away with 8 bits. How does that affect the depth information?

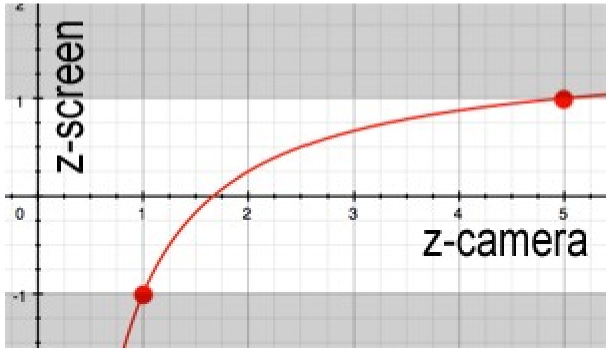
- Of course, we don’t need the four coordinates to mark the 2-D point in our screen space, but this information can be useful. Let’s examine how we can store depth information in this field
- Our matrix so far was intentionally kept simple. The projection matrix in OpenGL:

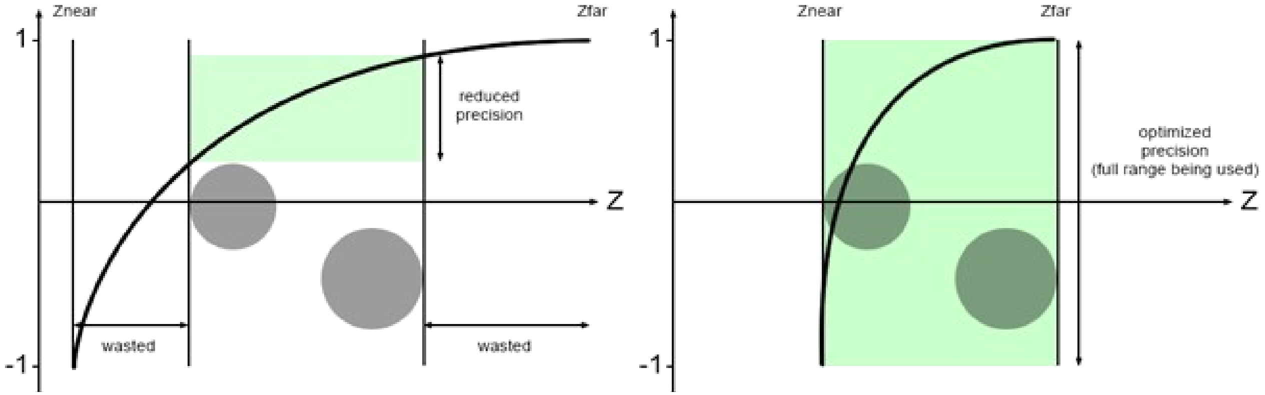
$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$



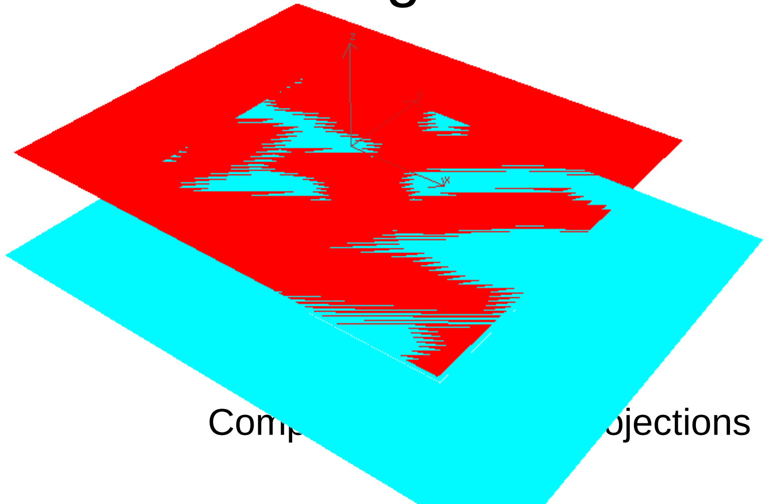
Perspective projection

- Depth mapping – a little insight into the caveats





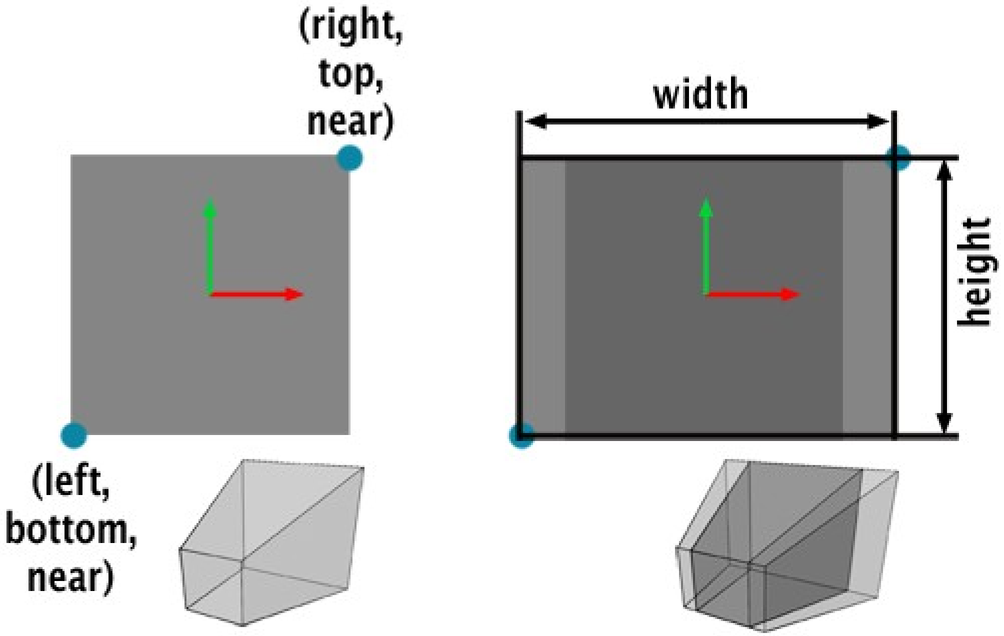
- The depth information is used for deciding which object to draw in case of occlusions – if there is not enough resolution, we run into “z-fighting” - a situation in which it’s impossible to distinguish which object is further!

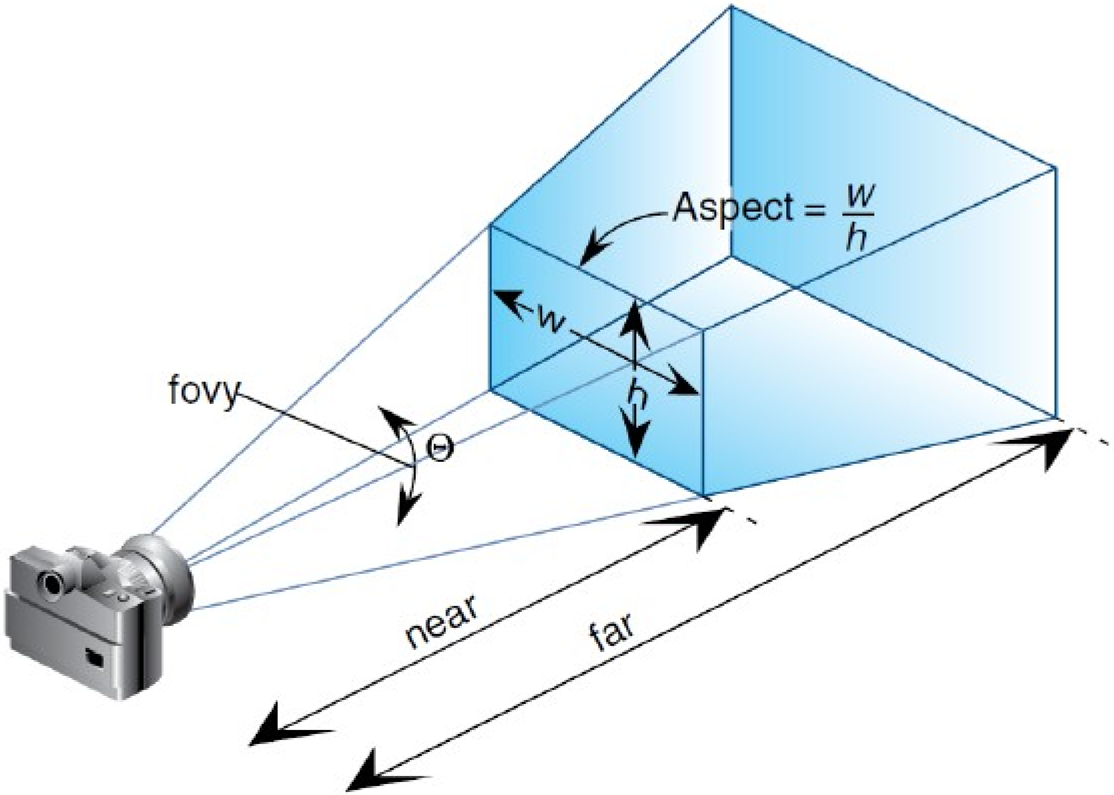


z-fighting is easy to recreate in blender if you add objects which have overlapping facets

Frustrum and projection

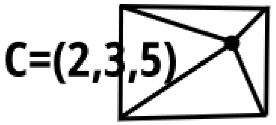
- Lots of hidden details that one needs to get right in practical work

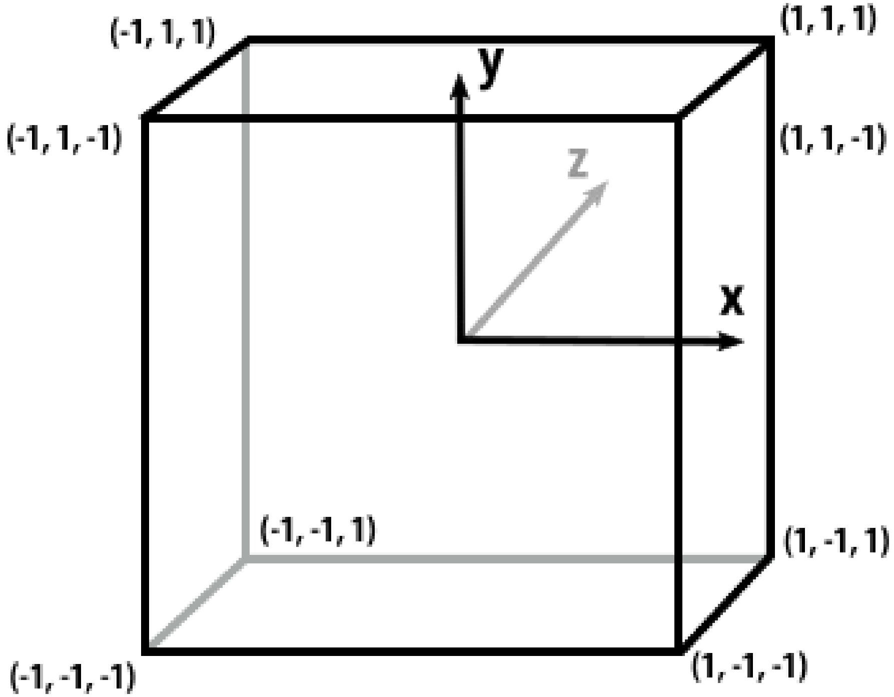


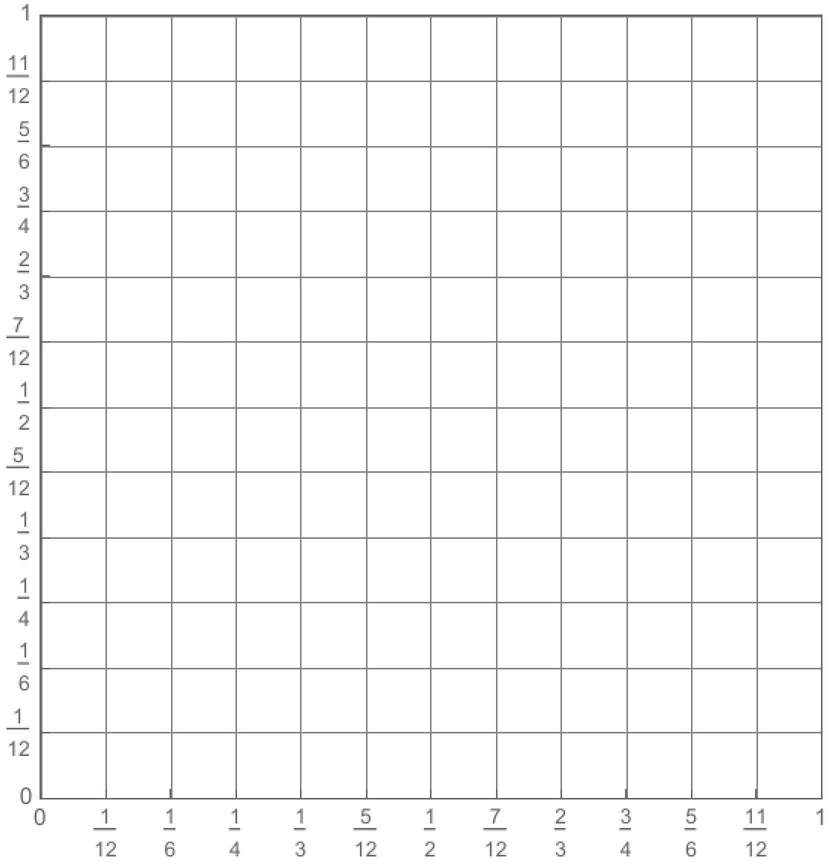


We can do it!

- We have the mathematics to do it
- Let’s assume $h=1$, camera axis aligned with z
- Let’s leave a and b undefined







We can do it!

For point $\boldsymbol{P_1} = (1,-1,1)$:

- Let's start by converting to homogeneous coordinates
- As the projection transformation was formulated in the camera reference frame (= pinhole at $(0,0,0)$), we have to apply translation of the point by $\boldsymbol{-C} = (-2, -3, -5)$
- Now, we can transform the point
- We have to normalise to get $w=1$
- Voilà!
  - it could have been done in one operation!
  - what if we had rotated the camera?

$$P_1^{HC} = \begin{bmatrix}1\\-1\\1\\1\end{bmatrix} \qquad \left.\begin{matrix}T_{(-2,-3,-5)}\\ R_{(0,0,0)}\end{matrix}\right\}\begin{bmatrix}1&0&0&-2\\0&1&0&-3\\0&0&1&-5\\0&0&0&1\end{bmatrix}$$

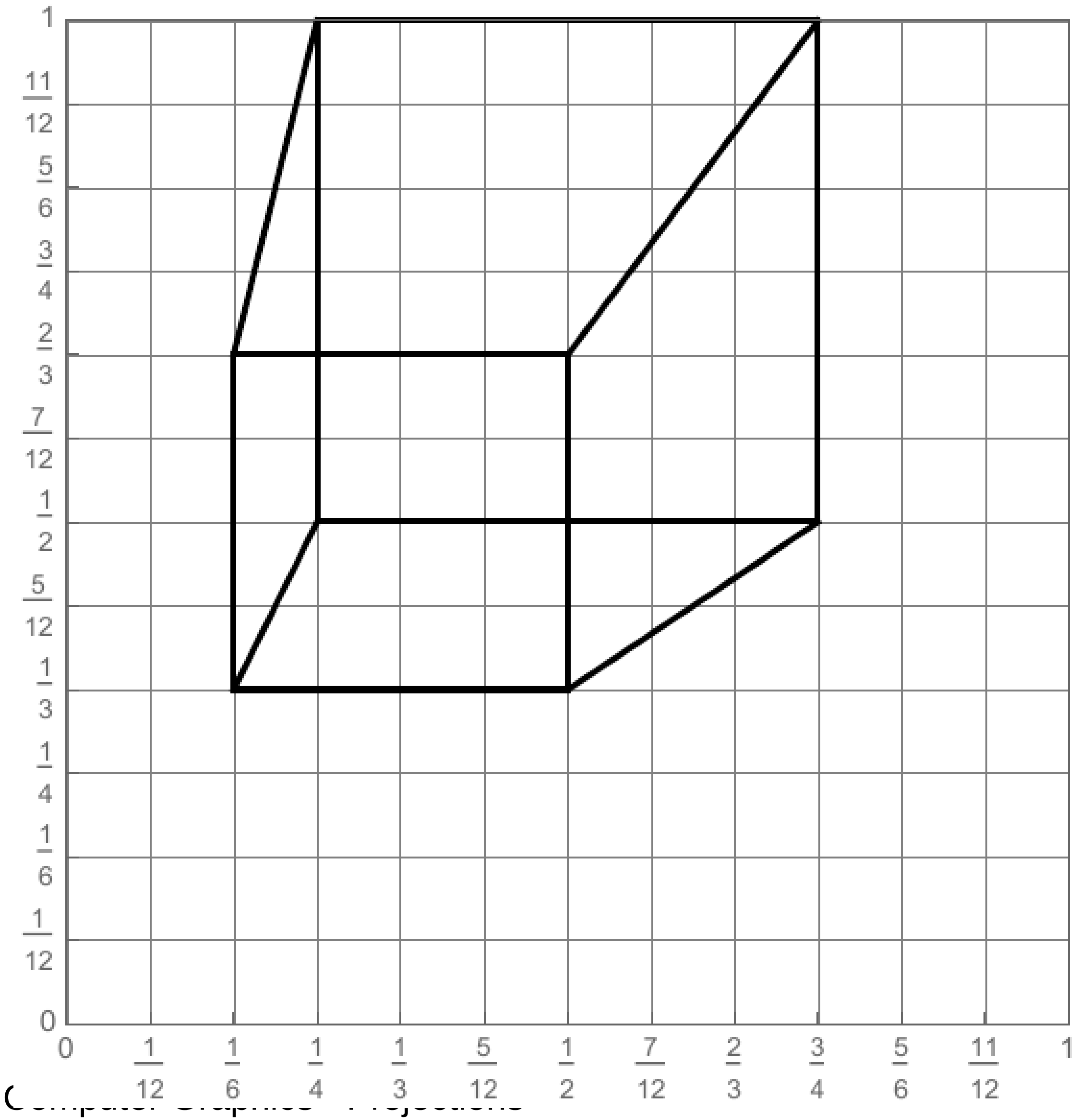
$$P_1' = \begin{bmatrix}1&0&0&-2\\0&1&0&-3\\0&0&1&-5\\0&0&0&1\end{bmatrix}\begin{bmatrix}1\\-1\\1\\1\end{bmatrix} = \begin{bmatrix}-1\\-4\\-4\\1\end{bmatrix}$$

$$P_1'' = \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&a&b\\0&0&1&0\end{bmatrix}\begin{bmatrix}-1\\-4\\-4\\1\end{bmatrix} = \begin{bmatrix}-1\\-4\\-4a+b\\-4\end{bmatrix}$$

$$P"_1 = \begin{bmatrix}\frac{1}{4}\\1\\a-\frac{1}{4}b\\1\end{bmatrix} \rightarrow P_1^{Screen} = \begin{bmatrix}\frac{1}{4}\\1\end{bmatrix}$$

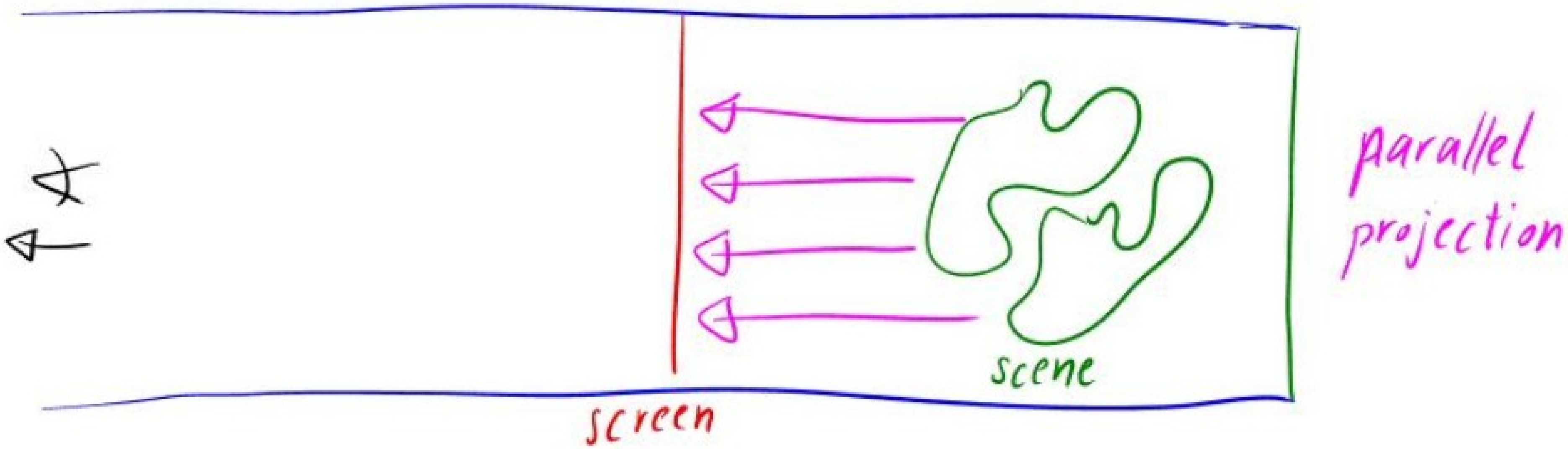
We can do it!

- This is what we should obtain:



More projections?

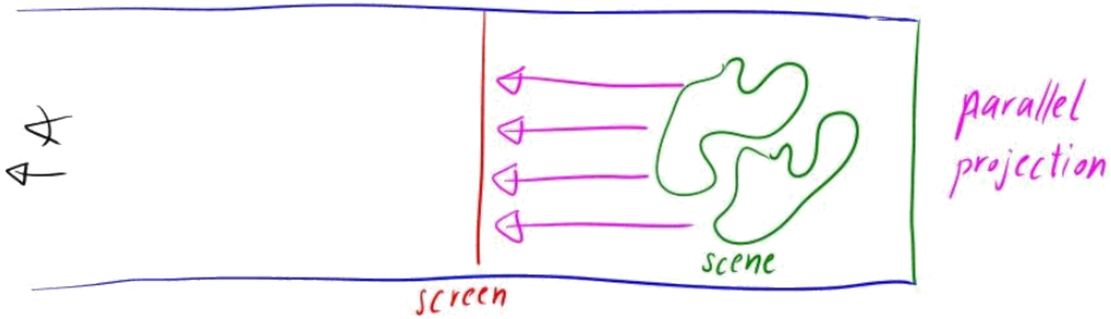
- Actually, perspective projection is only one of the infinite set of possible projection, with a few others also commonly used
- We skipped the simplest one, **parallel projection**:

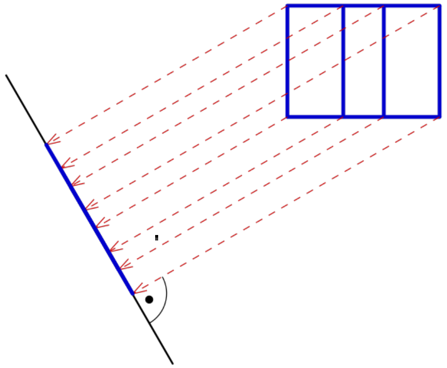


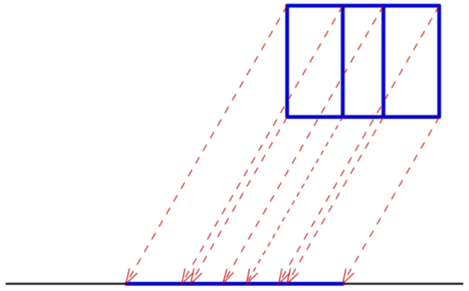
- It’s conceptually equivalent to the perspective projection with the camera at infinity. No kidding – it has serious consequences for photogrammetry!

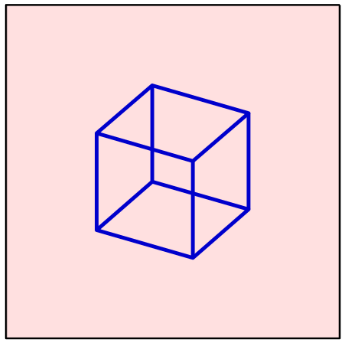
Parallel and orthographic projection

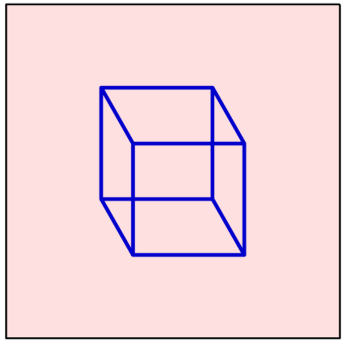
- The diagram actually shows an **orthographic projection** (sometimes called orthogonal), because we chose to have all the rays perpendicular to the screen. A parallel projection does not need to respect this assumption!
- Example easy to see in blender





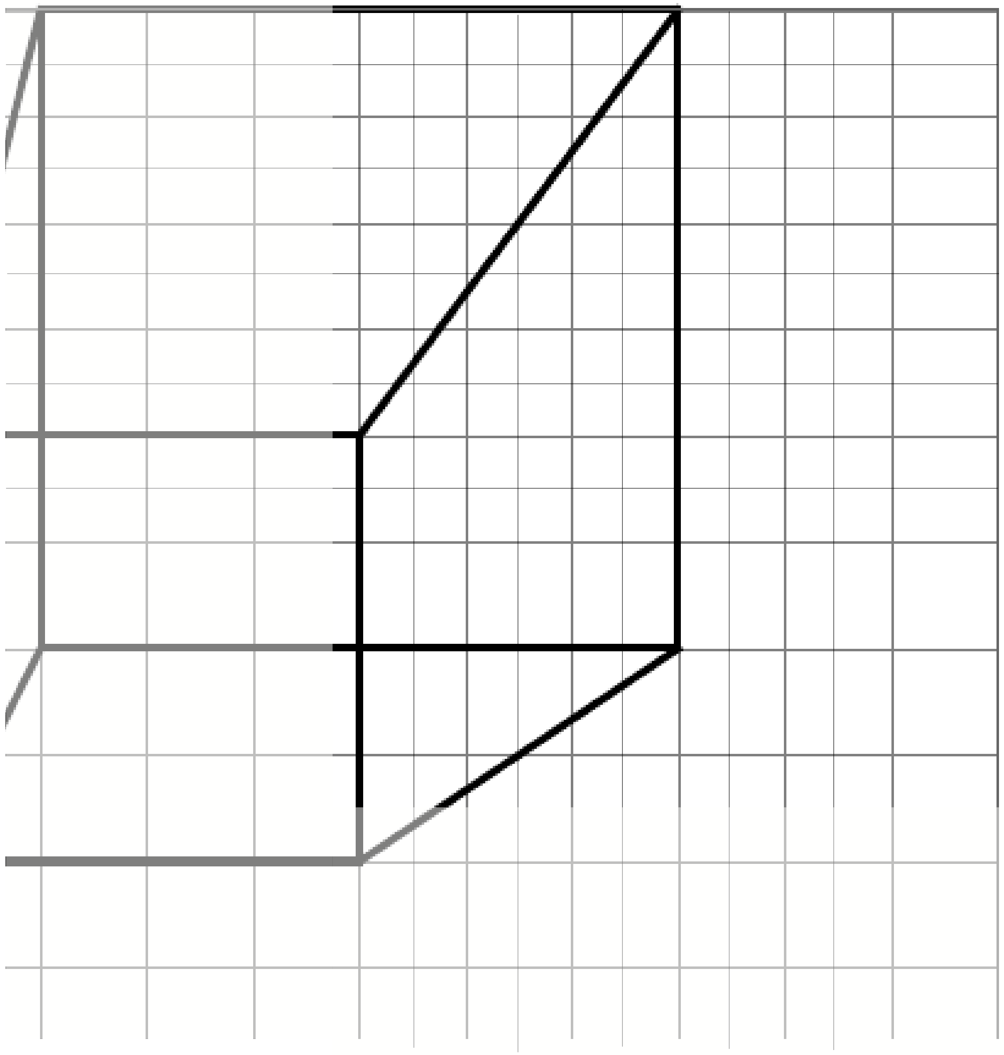






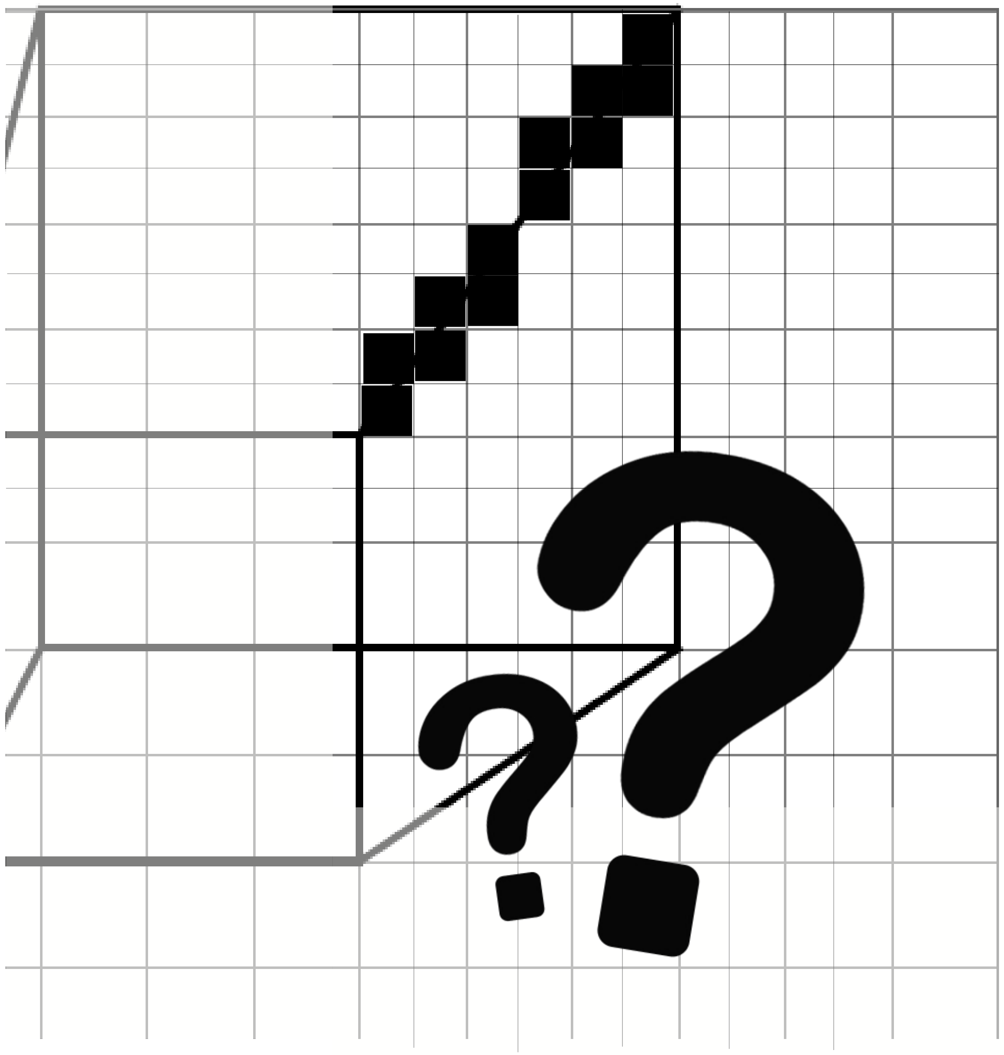
There is more work coming...

- What about our edges? How do we actually draw them when we don’t have a coordinate system but a grid of pixels? How do we do it when they are not horizontal or vertical?



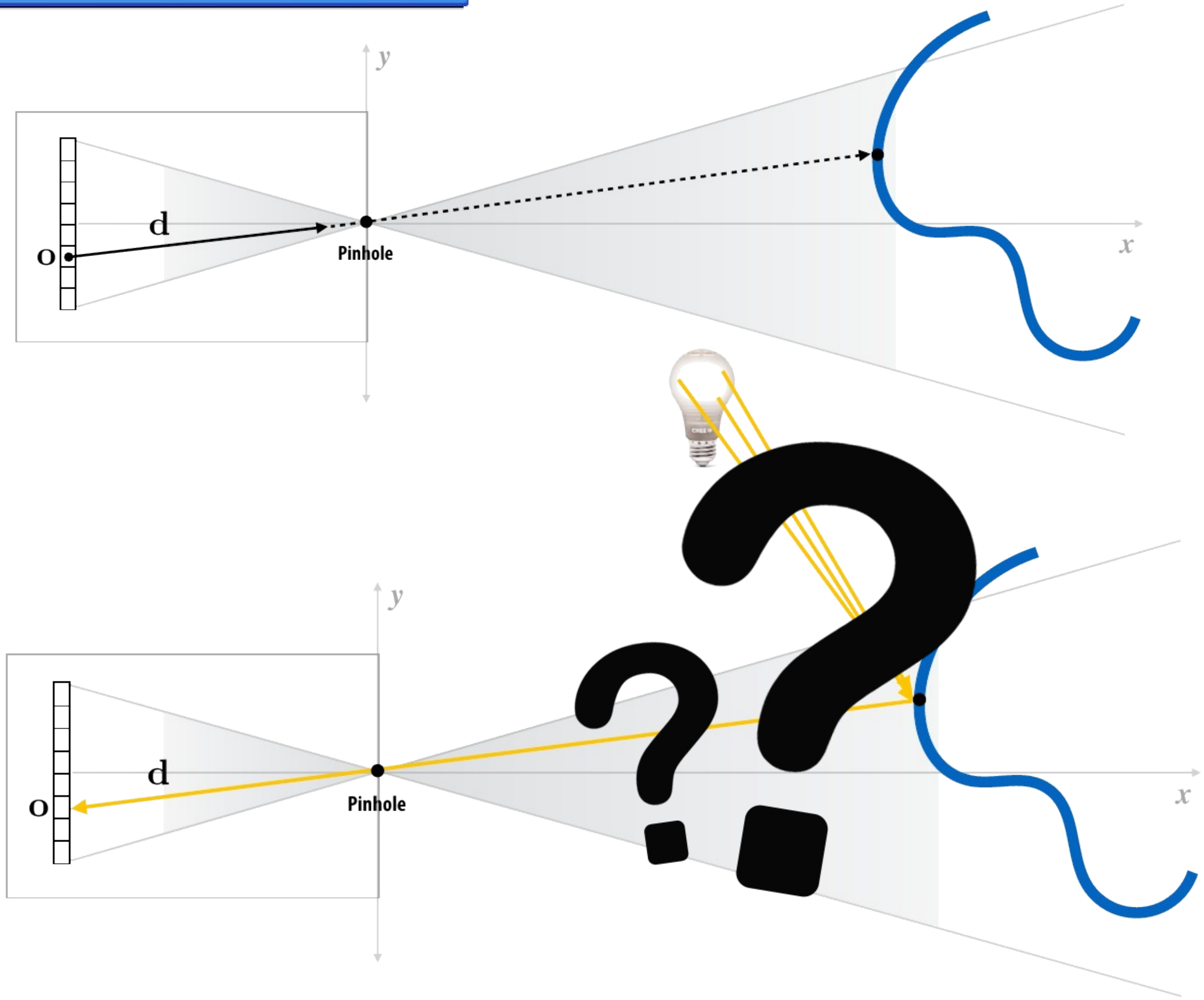
There is more work coming...

- What about our edges? How do we actually draw them when we don’t have a coordinate system but a grid of pixels? How do we do it when they are not horizontal or vertical?



There is MUCH more work coming...

- So far we spoke about the light ray coming from an object but what if we want to render an image of an object which does not produce light but rather reflects it?



Thank you!

- Questions?