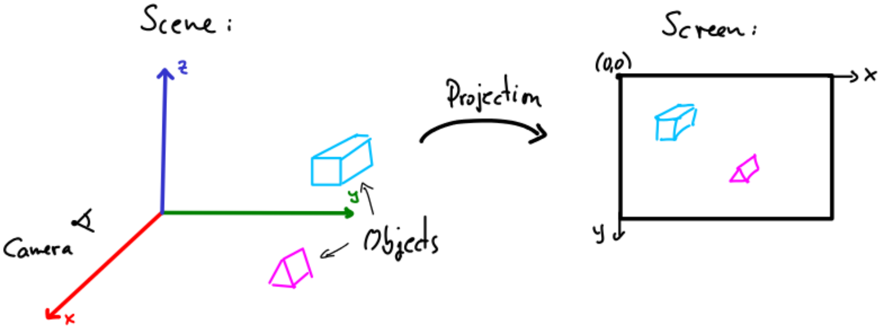
Chapter 9: Camera models and the view pipeline

K. Jünemann
Department Informations- und Elektrotechnik
HAW Hamburg

Content

Camera models and the view pipeline

Key process in 3D graphics: map 3D-scene (assembly of objects in 3D space) to 2D projection plane (screen).

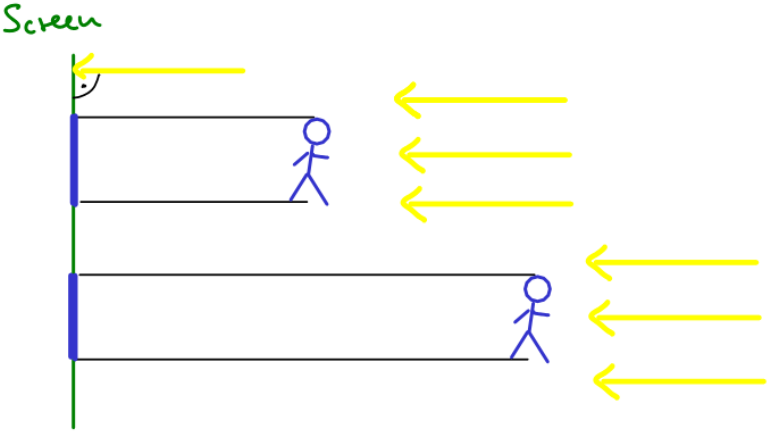


Two important types of projections:

- *Parallel* (a.k.a. *orthographic*) projection
- *Perspective* projection

Parallel Projections

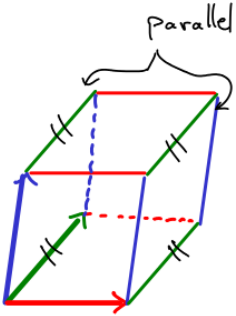
Optical definition:
Shadow of an object by parallel light rays perpendicular to projection screen.



Parallel Projections

Properties:

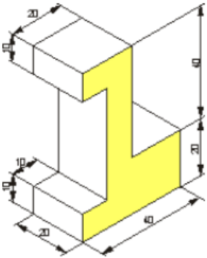
- Size of projection is independant of distance between object and screen.
- Parallel lines in 3D are mapped to parallel lines on screen.



Parallel Projections

Parallel Projections are ...

- popular for technical drawings,

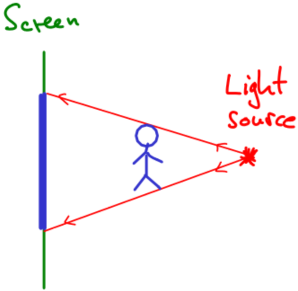


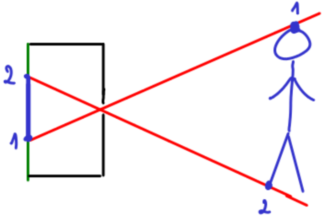
- not well suited for most 3D-graphics applications,
- implemented in **`THREE.OrthographicCamera`**.

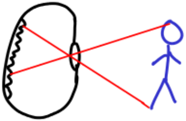
Perspective Projections

Optical definition:
Shadow of an object by light rays emitted from point-like light source.

Similar idea: pinhole camera, eye

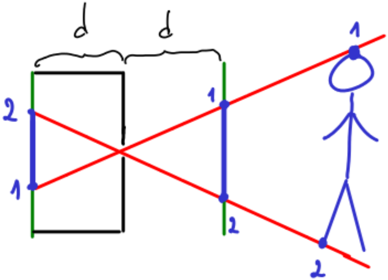






Perspective Projections

Pinhole camera displays image upside down!

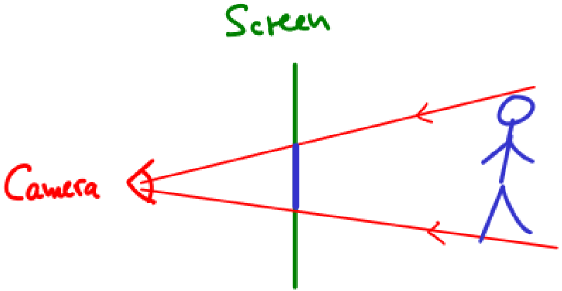


Place screen at same distance between object and hole (same distance from hole):

- Image correctly oriented!
- Apart from that both images are the same.

Perspective Projections

Most common camera model used in 3D graphics:

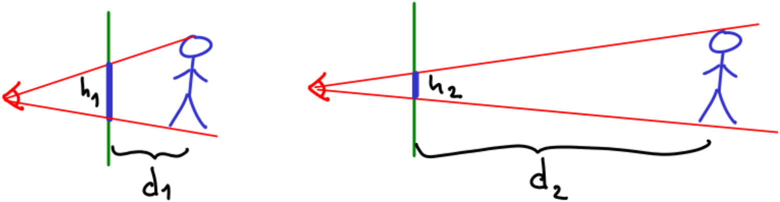


- Camera location is a point.
- Distance between camera and projection screen irrelevant as image gets scaled to screen size anyway.

Perspective Projections

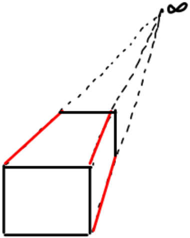
Properties:

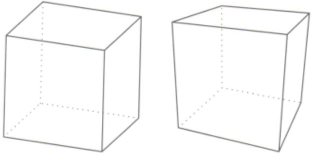
- Parallel projection = perspective projection with camera at infinity.
- Distant objects appear smaller, closer objects larger: $d_1 < d_2 \implies h_1 > h_2$.



Perspective Projections

- Parallel lines in 3D are *not* mapped to parallel lines on screen. Parallel lines intersect at infinity.

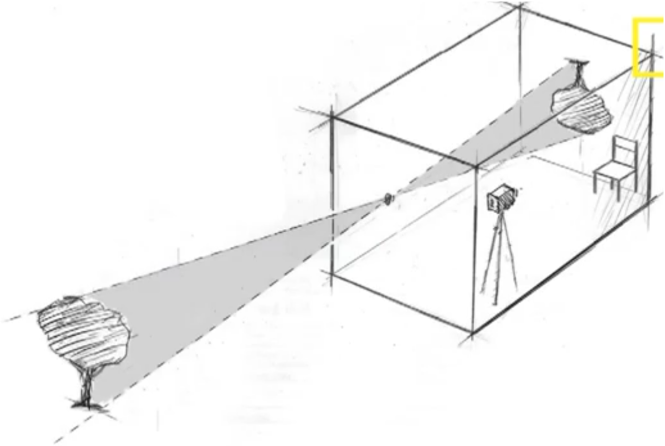




Left: Perspective projection of a cube, the red lines are not parallel.
Right: Comparison of a orthographic and perspective projection of a cube. (from Buss, *3D Computer Graphics*)

Perspective Projections

Nice project: build a room sized pinhole camera:



Find link to video on `README.md` for this chapter.

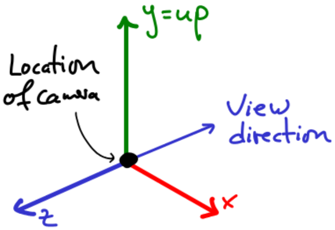
Perspective cameras and the view frustum

Camera space: local coordinate system attached to camera.

- a.k.a. *View space*
- cameras are derived from `Object3D`

Orientation of camera space:

- View direction: along negative z - axis.
- Up direction: along y axis.

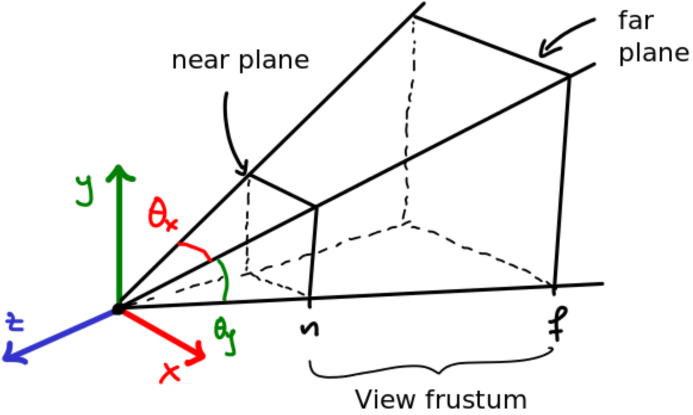


Perspective projection is implemented in **`THREE.PerspectiveCamera`**.

Perspective cameras and the view frustum

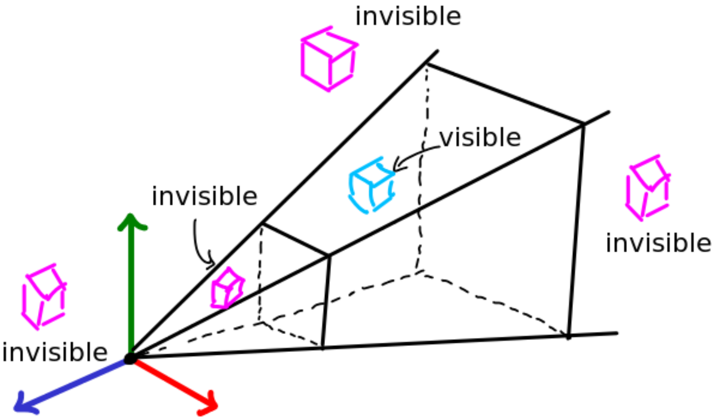
View frustum: visible part of the world.

- Pyramid with top cut off.
- Think of near plane as projection plane.



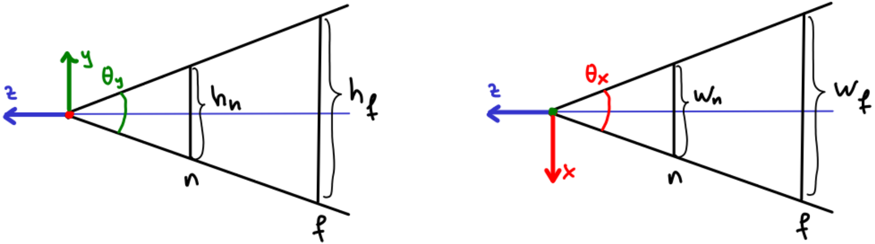
Perspective cameras and the view frustum

Only objects inside the view frustum are visible:



Perspective cameras and the view frustum

Parameters of the view frustum:



Side and top view of frustum.

- n: distance camera to near plane
- f: distance camera to far plane
- θ_x: field of view (fov) in x direction
- θ_y: field of view (fov) in y direction
- w_n, h_n: width and height of near plane
- w_f, h_f: width and height of near plane

Perspective cameras and the view frustum

- Aspect ratio: $a = \frac{w_n}{h_n} = \frac{w_f}{h_f}$
- Relations for near plane (similar for far plane):

$$\frac{w_n}{2n} = \tan\left(\frac{\theta_x}{2}\right) \qquad , \frac{h_n}{2n} = \tan\left(\frac{\theta_y}{2}\right)$$

- 4 independent params:
  `three.js` chooses θ_y, a, n, f

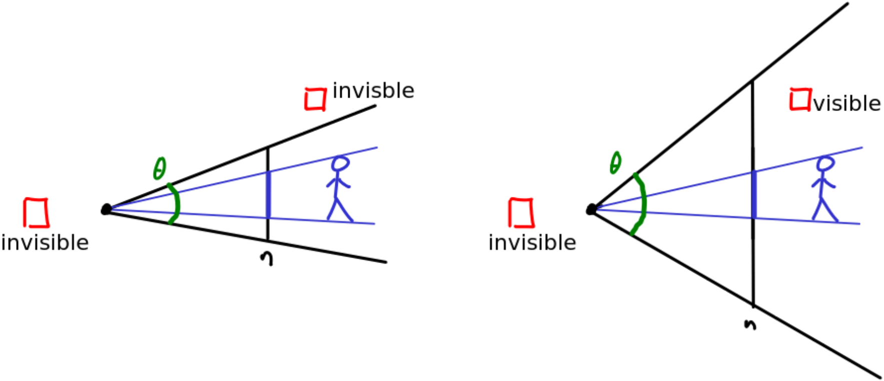
```
const fov = 60;   // degree !!
const a = canvas.width/canvas.height;
const n = 1, f = 1000;
const cam = new THREE.PerspectiveCamera(fov, a,
    n, f);
```

  - a should coincide with canvas aspect ratio
  - n and f set scale of 3D coordinates
  - f usually very large

Perspective cameras and the view frustum

Zooming:

- changing the field of view θ
- keeping camera position fixed

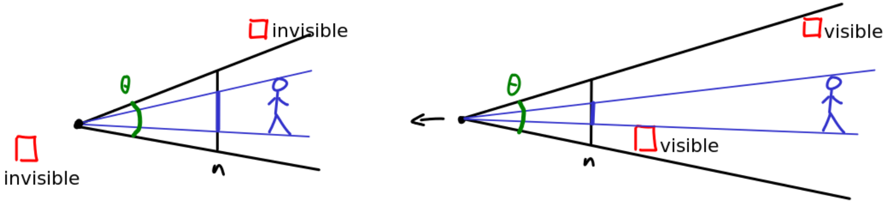


- θ larger: zooming out, objects get smaller
- θ smaller: zooming in, objects get larger

Perspective cameras and the view frustum

Dollying:

- moving camera along its own z direction
- keeping field of view fixed



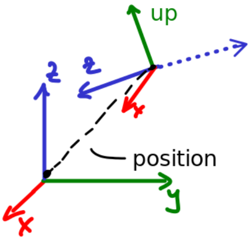
Moving away from object:

- objects get smaller
- objects initially behind camera become visible

Perspective cameras and the view frustum

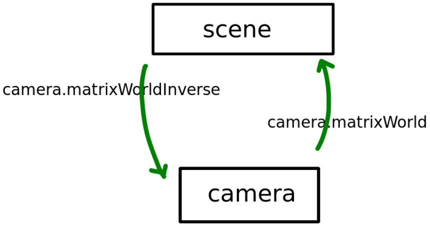
The camera in the scene:

- A camera *is an* `Object3D`,
- can be positioned like any other object,
- *can* be added to any other object,
- `camera.matrix` set by mouse controller



Special field in camera objects:
`matrixWorldInverse`

- maintained for performance reasons.



Exercise 1

Implement a ball reflected inside view frustum:

- (x_c, y_c, z_c): coordinate of ball in camera space!
- h_c, w_c: height and width of frustum at z_c:

$$h_c = 2z_c \tan\left(\frac{\theta_y}{2}\right) \ , \ w_c = a \cdot h_c$$

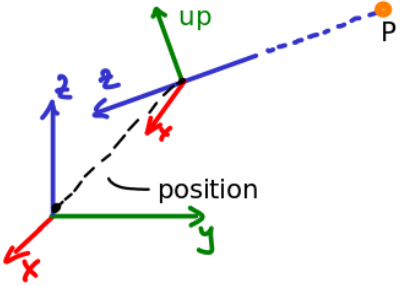
- Collision with frustum plane:

```
if(x_c > wc/2 - radius) {
  // change speed of ball
}
```

Perspective cameras and the view frustum

`Object3D.lookAt`: yet another way to specify orientation of a coordinate system

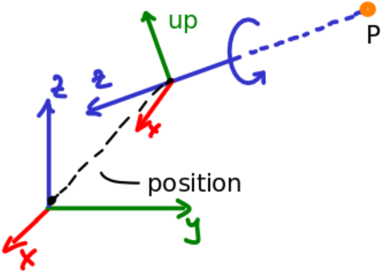
- particularly useful for cameras,
- `camera.lookAt(x,y,z)` rotates camera space such that a point P with *world space* coordinates (x, y, z) sits on *negative* z-axis.
- funny: `Object3D.lookAt` places point on *positive* z-axis



Perspective cameras and the view frustum

The vector `Object3D.up` (default: $(0, 1, 0)$)

- looking at a point *P* leaves one degree of freedom: rotation around camera z-axis
- `lookAt` method 'tries' to align y-axis of *camera space* with up-vector in *world space*.



Exercise 2

Consider the following code:

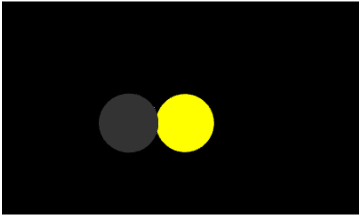
```
const camera = new THREE.PerspectiveCamera(...) ;
camera.position.set(5, 0, 0);
camera.lookAt(0, 0, 0);
```

1. Work out `camera.matrix`
2. Verify with `three.js`

Perspective cameras and the view frustum

Example: Place the camera at the center of the sun-earth-moon example (chapter 8)

1. Look at the sun.
2. Look at the moon.
3. Look at a fixed point in the earth coordinate system.
   (1,0,0) for example



Exercise 3

Upgrade your Snake game by placing the camera on top of the head of the snake.

1. Make the camera look into the direction the snake is moving.

Hint: comment out your mouse controller in case you used one.

Projection as a matrix operation

- Parallel projection is an affine map $\Longrightarrow$ Implementation as matrix multiplication.
  - Straightforward with the concepts of chapter 8
  - Not further needed in this lecture
- Perspective projection is *not* an affine map: Implementation by matrix multiplication and a trick!
  - Perspective projection matrix contained in `camera.projectionMatrix`.
  - Perspective projections use bottom row of 4×4 matrix!
  - Whenever a camera parameter changes, the projection matrix needs to be updated by hand!

```
camera.updateProjectionMatrix();
```

Projection as a matrix operation

Important use case of updating camera parameters: react to screen resizing.

- signalled by browser event *"resize"*.

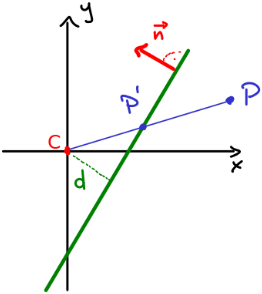
```
window.addEventListener("resize", function() {
  const w = window.innerWidth;
  const h = window.innerHeight;
  renderer.setSize(w,h);
  camera.aspect = w/h;
  camera.updateProjectionMatrix();
});
```

- `window` is global object present in every browser.

Projection as a matrix operation

Projection in camera space: camera (C) located at origin.
P gets projected to P':

- Position vectors: $\vec{p} = \overline{OP}$, $\vec{p}' = \overline{OP'}$
- Projection: $\vec{p}' = \alpha \vec{p}$ for some α
- P' on projection plane: $\vec{p}' \cdot \vec{n} + d = 0$



Solve for α: (first equation $\cdot \vec{n}$) into second:

$$\vec{p}' \cdot \vec{n} = \alpha \vec{p} \cdot \vec{n} = -d \Rightarrow \alpha = -\frac{d}{\vec{p} \cdot \vec{n}}$$

Insert back into first equation

$$\vec{p}' = -\frac{d}{\vec{p} \cdot \vec{n}} \vec{p} \qquad \text{(non-affine!)}$$

Projection as a matrix operation

This expression cannot be implemented as matrix multiplication. What to do?
Consider the expression

$$\begin{pmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ -n_1 & -n_2 & -n_3 & 0 \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} = \begin{pmatrix} d\,p_1 \\ d\,p_2 \\ d\,p_3 \\ -\vec{p}\cdot\vec{n} \end{pmatrix} \overset{\text{p.d.}}{\rightarrow} \begin{pmatrix} -\frac{d}{\vec{p}\cdot\vec{n}}p_1 \\ -\frac{d}{\vec{p}\cdot\vec{n}}p_2 \\ -\frac{d}{\vec{p}\cdot\vec{n}}p_3 \\ 1 \end{pmatrix}$$

- Fourth component 'wrong', but in a useful way $\implies$ just divide by it (*perspective division*)!
- Perspective division looks like a funny trick but is a natural operation in projective geometry.
- Because of perspective division, any scalar multiple of this matrix will do the job.

Projection as a matrix operation

The perspective projection onto a projection plane with normal unit vector $\vec{n}$ at distance d from the camera located at the origin is performed by the following two steps:

1. Multiply with the matrix $\mathbf{Q}_{\vec{n},d} = \left(\begin{array}{c|c} d\mathbf{E} & \vec{0} \\ \hline -\vec{n}^T & 0 \end{array}\right)$
2. Divide the result by its last component (perspective divide)

Remark:

- $\mathbf{Q}_{\vec{n},d} \cdot \mathbf{Q}_{\vec{n},d} = d\mathbf{Q}_{\vec{n},d}$.
  Projecting twice is the same as projecting once (due to perspective division).

Exercise 4

Consider perspective projection in $\mathbb{R}^2$ with camera at the origin onto a line passing through $(1, 0)$ and $(0, -1)$.

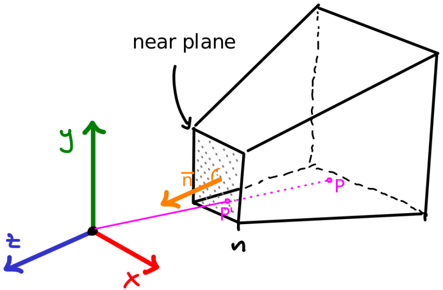
- Write down the 3×3 projection matrix.
- What are the projections of
  - $P = (1, 0)$
  - $Q = (2, -2)$
  - $R = (1, 1)$

Projection as a matrix operation

Important use case: projection onto near plane

- Normal vector:
  $\vec{n} = (0, 0, 1)$
- Distance origin to projection plane: $d = n$

$$\mathbf{Q} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$



Application to point P with view space coordinates (x, y, z):

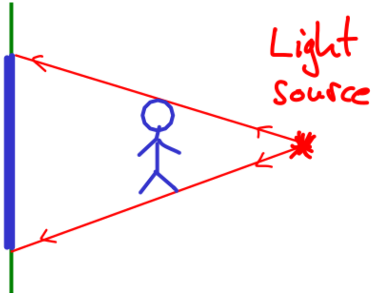
$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} nx \\ ny \\ nz \\ -z \end{pmatrix} \overset{\text{p.d.}}{\rightarrow} \begin{pmatrix} -\frac{nx}{z} \\ -\frac{ny}{z} \\ -n \\ 1 \end{pmatrix}$$

Projection as a matrix operation

Example: self-made shadows

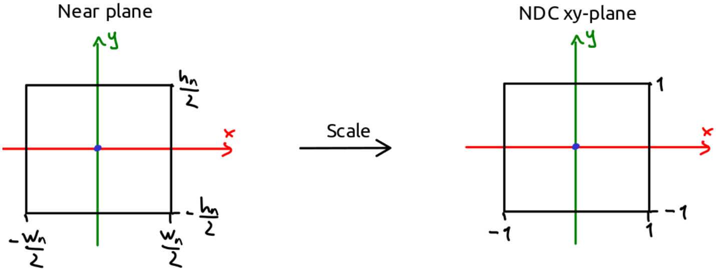
Shadow cast by point light: perspective projection

- replace camera by point light source located at origin
- place projection screen behind object



Normalized Device Coordinates (NDC)

After projection near plane is rescaled to size 2×2:



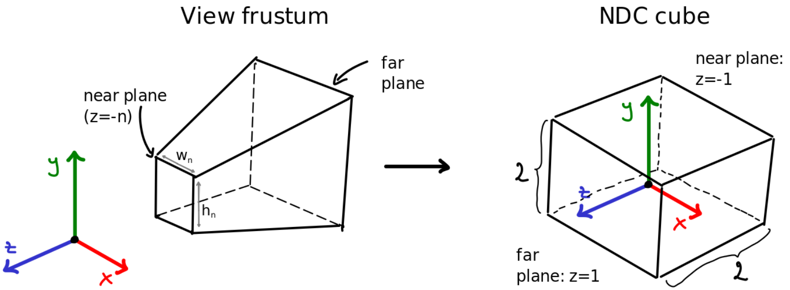
$$\begin{pmatrix} \frac{2}{w_n} & 0 & 0 & 0 \\ 0 & \frac{2}{h_n} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix} = \begin{pmatrix} \frac{2n}{w_n} & 0 & 0 & 0 \\ 0 & \frac{2n}{h_n} & 0 & 0 \\ 0 & 0 & n & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Normalized Device Coordinates (NDC)

Challenge: rendering process needs to keep track of info about z-coordinate (in view space) for hidden object removal!

Solution: Map view frustum to *Normalized Device Coordinates (NDC)*: a cube of size 2: $x, y, z \in [-1, 1]$

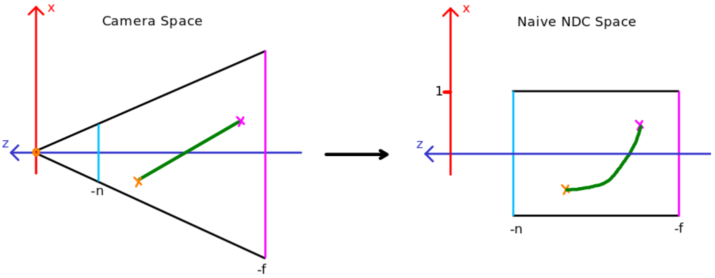
- rescale x, y - coordinates as explained on previous slide.
- don't project all points to $z = -n$ but apply a smarter map.
- near plane mapped to $z = -1$, far plane to $z = 1$.



Normalized Device Coordinates (NDC)

A naive mapping from camera space to NDC space:
(ignore shift and scale of z to $[-1, 1]$ for the moment)

$$(x, y, z) \to \left(-\frac{2nx}{w_n z}, -\frac{2ny}{h_n z}, z\right)$$



Problem: this does ***not*** map straight lines to straight lines!

Normalized Device Coordinates (NDC)

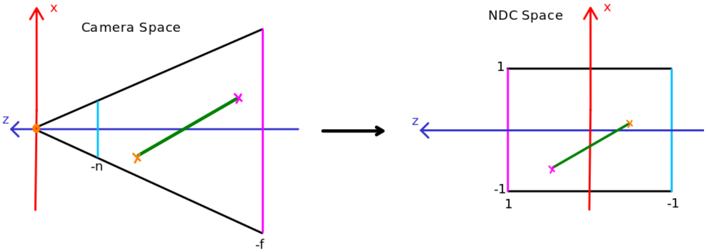
How do we map the z-coordinate from camera to NDC space?

- Monotonous function of z would be ok $\implies$ enough for detection which object is closest to camera.
- Straight lines in camera space should be mapped to straight lines in NDC space!

Normalized Device Coordinates (NDC)

Solution: A map of the form $z \to a + \dfrac{b}{z}$ does the job!

- adapt a and b such that
  - $-n \to -1$
  - $-f \to 1$.



This leads to $z \to \dfrac{f+n}{f-n} + \dfrac{2nf}{f-n} \cdot \dfrac{1}{z}$

- Straight lines are mapped to straight lines! (Ex.: Prove this!)

Exercise 5

1. Verify that the function

$$g(z) = \frac{f+n}{f-n} + \frac{2nf}{f-n} \cdot \frac{1}{z}$$

   has the properties
   - $g(-n) = -1$
   - $g(-f) = 1$
2. Draw $g(z)$ in the intervall $[-f, -n]$ for $f = 3$ and $n = 1$.

Normalized Device Coordinates (NDC)

Implement this map as

1. Multiply by $\mathbf{Q}_{\text{ndc}}$
2. Do perspective division!

$$\mathbf{Q}_{\text{ndc}} = \begin{pmatrix} \frac{2n}{w_n} & 0 & 0 & 0 \\ 0 & \frac{2n}{h_n} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & 2\frac{nf}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Blue: Project and scale x and y coordinates.

Red: Do the right thing with the z coordinate.

Green: Negative normal vector of projection plane.

This matrix is stored in **`camera.projectionMatrix`**.

Normalized Device Coordinates (NDC)

Apply this to camera space coordinates (x_c, y_c, z_c) resulting in normalized device coordinates $(x_{\text{ndc}}, y_{\text{ndc}}, z_{\text{ndc}})$:

$$\begin{pmatrix} \frac{2n}{w_n} & 0 & 0 & 0 \\ 0 & \frac{2n}{h_n} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & 2\frac{nf}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_c \\ y_c \\ z_c \\ 1 \end{pmatrix} = \begin{pmatrix} 2\frac{nx_c}{w_n} \\ 2\frac{ny_c}{h_n} \\ \frac{n+f}{n-f} z_c + 2\frac{nf}{n-f} \\ -z_c \end{pmatrix}$$

$$\overset{\text{p.d.}}{\rightarrow} \begin{pmatrix} -2\frac{nx_c}{w_n z_c} \\ -2\frac{ny_c}{h_n z_c} \\ \frac{f+n}{f-n} + 2\frac{nf}{f-n} \cdot \frac{1}{z_c} \\ 1 \end{pmatrix} = \begin{pmatrix} x_{\text{ndc}} \\ y_{\text{ndc}} \\ z_{\text{ndc}} \\ 1 \end{pmatrix}$$

Note: $z_c < -n$

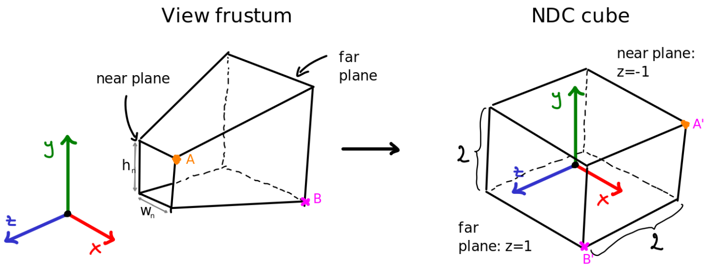
Exercise 6

1. Condsider the following code:

```
const cam = new THREE.PerspectiveCamera(90,2,2,10)
```

   - Work out $\mathbf{Q}_{\text{ndc}}$
   - Compare with `three.js`

2. Check that $\mathbf{Q}_{\text{ndc}}$ maps A to A' and B to B':



The graphics pipeline (1): vertex part

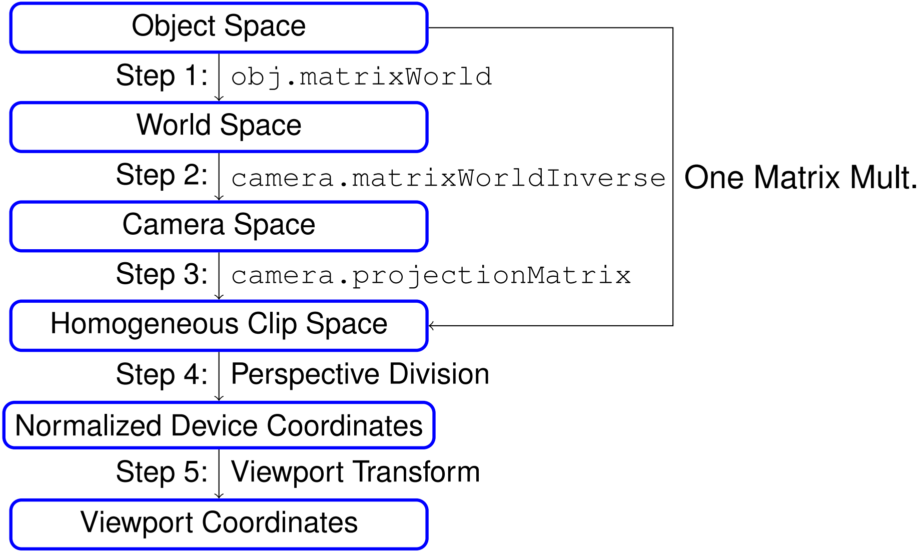
Goal: Calculate vertex positions on computer screen!

Given: Geometry definition (vertices, faces) in local coordinates

Tasks to be done along the way:

- hidden object removal
- Removal of (parts of) objects outside view frustum

The graphics pipeline (1): vertex part



The graphics pipeline (1): vertex part

Steps 1-3:
For each vertex steps 1 to 3 is *one* matrix multiplication by

$$\texttt{camera.projectionMatrix}$$
$$\cdot\texttt{camera.matrixWorldInverse}$$
$$\cdot\texttt{obj.matrixWorld}$$

- This matrix is assembled at Javascript level.
- Matrix multiplication executed by graphics engine!

The graphics pipeline (1): vertex part

After step 3:

Homogeneous clip space = space of coordinates *before* perspective division.

- Assume vertex has clip space coordinates (x, y, z, w): It is outside view frustum if

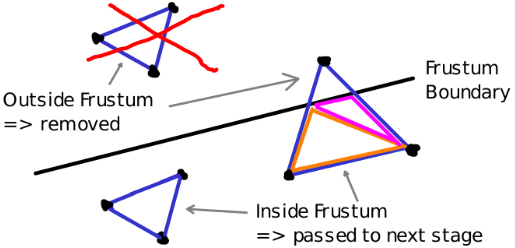
$$\begin{aligned} |x/w| > 1 &\iff |x| > w \\ \text{or } |y/w| > 1 &\iff |y| > w \\ \text{or } |z/w| > 1 &\iff |z| > w \end{aligned}$$

- allows for efficient *face clipping*: removal of vertices outside frustum

The graphics pipeline (1): vertex part

Face clipping: removal of vertices outside frustum

- implemented deep inside graphics engine



The graphics pipeline (1): vertex part

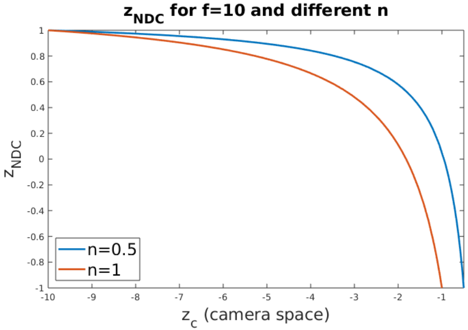
After step 4: *Hidden object removal* in NDC space.
Relation between camera space coordinates (x_c, y_c, z_c) and normalized device coordinates $(x_{\text{ndc}}, y_{\text{ndc}}, z_{\text{ndc}})$:

$$
\begin{aligned}
x_{\text{ndc}} &= -2\frac{nx_c}{w_n z_c} \\
y_{\text{ndc}} &= -2\frac{ny_c}{h_n z_c} \\
z_{\text{ndc}} &= \frac{n+f}{f-n} + 2\frac{nf}{f-n} \cdot \frac{1}{z_c}
\end{aligned}
$$

- x_{ndc} and y_{ndc} are sent to next step in pipeline.
- z_{ndc} is stored in *depth buffer*: the larger the value the further away the vertex from the camera!
- depth buffer is typically a 24 bit fixed point type
  $\implies$ limited resolution.

The graphics pipeline (1): vertex part

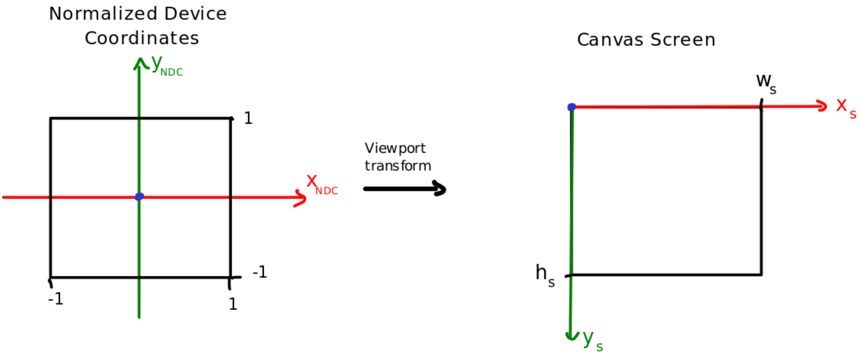
Plot of $z_{\mathrm{ndc}}(z_c)$ for $f = 10$ and $n = 0.5$ or $n = 1$



- The smaller n, the worse the depth resolution far away from the camera.
- Choose n as large as possible to avoid z-fighting!

The graphics pipeline (1): vertex part

Step 5: the viewport transform
Map normalized device coordinates to reserved screen space:



- just translate and scale
- scaling makes position of camera projection screen irrelevant

The graphics pipeline (1): vertex part

The viewport transform:

- Normalized device coordinates $x_{\text{ndc}}, y_{\text{ndc}}$:
  - Origin at center
  - Height and width equal to 2
- Screen:
  - coordinates x_s, y_s (pixel units)
  - height h_s, width w_s (set by canvas element)

$$
\begin{aligned}
x_s &= \frac{w_s}{2} x_{\text{ndc}} + \frac{w_s}{2} \\
y_s &= -\frac{h_s}{2} y_{\text{ndc}} + \frac{h_s}{2}
\end{aligned}
$$