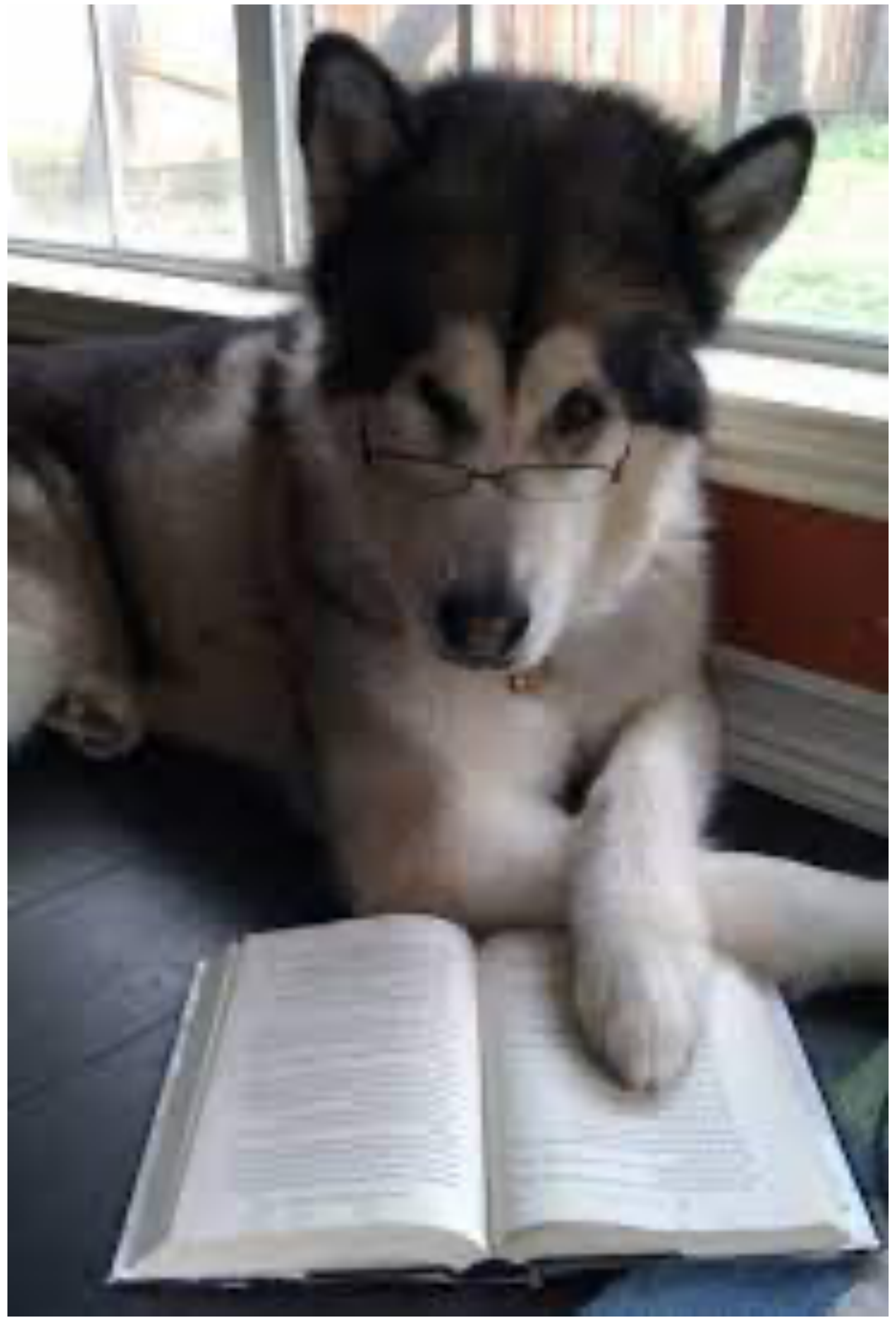
Coordinate Systems

CS 385 - Class 5
8 February 2022

Administrativa

- Reminder to sign up in Piazza
- First assignment has been graded
  - should be visible in Canvas
- Assignment observations
  - Reading is Fundamental
  - So are following directions



Speaking of Assignments ...

- Let's talk cones for a second
- Not what you were expecting, eh?
  - doesn't mean it's wrong
- This exercise has several goals:
  - verifying you have the enough JavaScript to get what we'll need done
    - it's not much, and will only be a little more involved
  - illustrate that there's a lot of topics that we'll need to get to what you're probably imaging

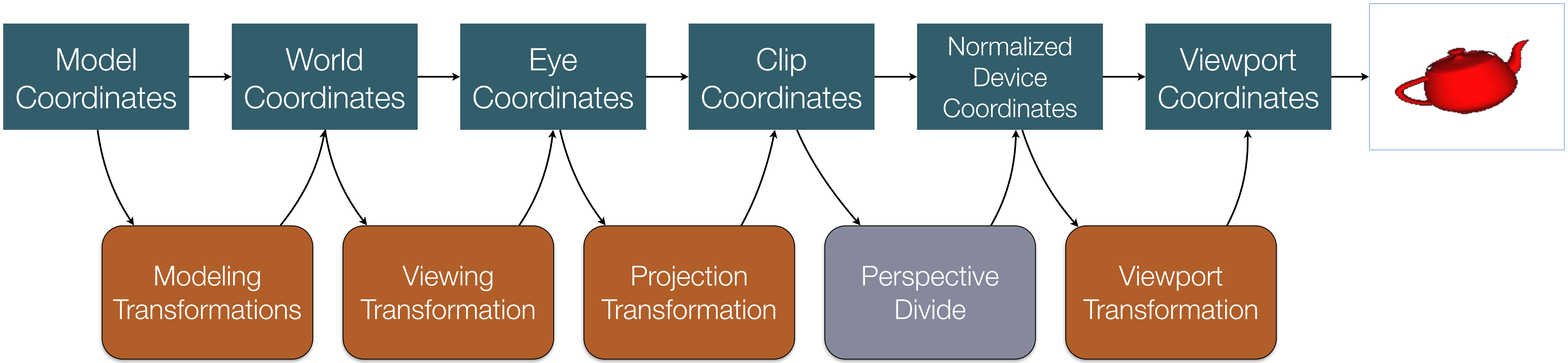
An Important WebGL 2.0 Shader Detail

- WebGL 2.0 shaders require an additional line of source code
- This line must start at the *first character* of the shader
  - That is, it needs to be immediately after the closing `>` of the `script` tag

```
<script id="vertex-shader"
   type="x-shader/x-vertex">#version 300 es
in  vec4 aPosition;
out vec4 vColor;

void main()
{
  vColor = vec4(0.0, 0.0, 1.0, 1.0);
  gl_Position = aPosition;
}
</script>
```

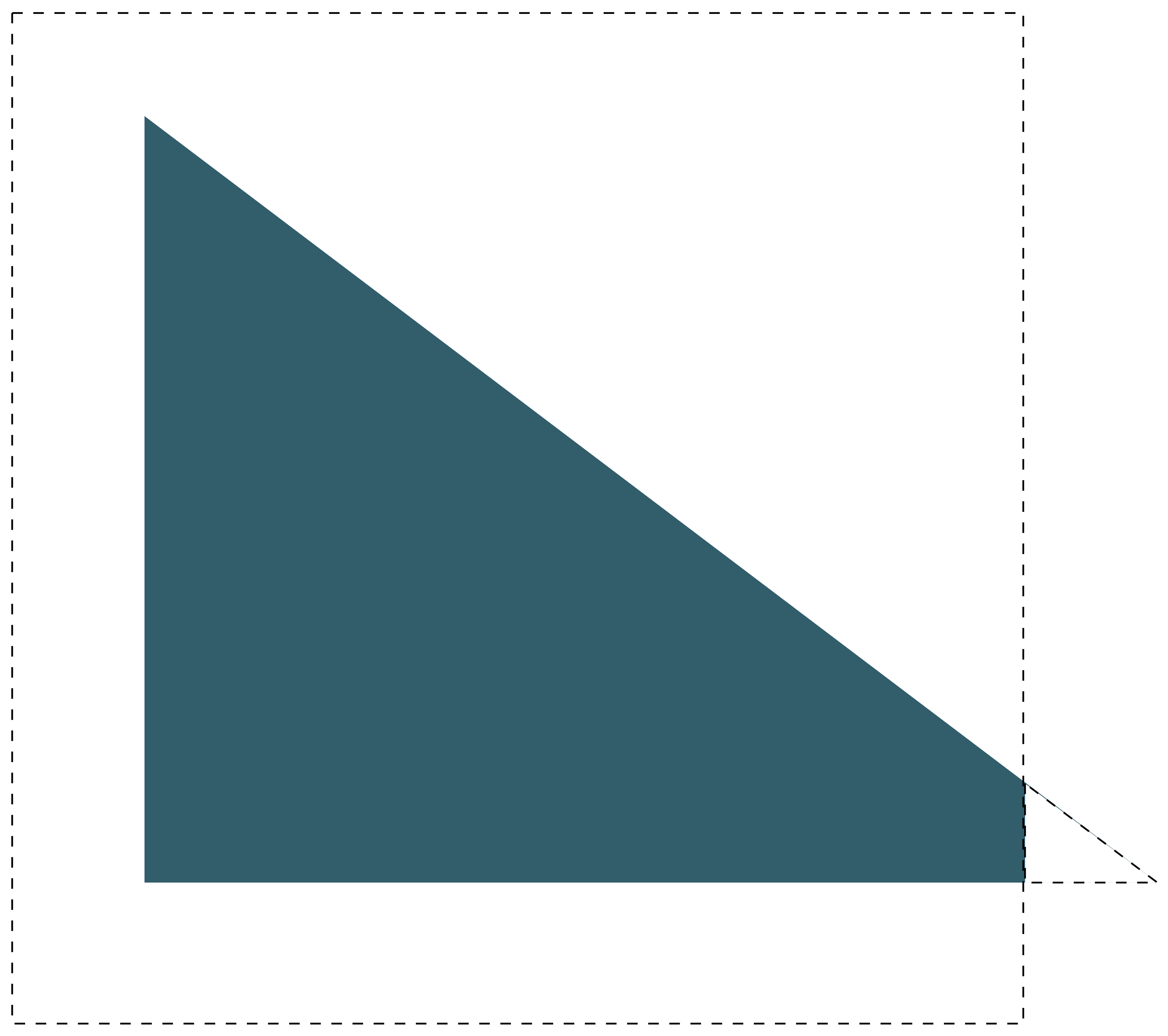
Coordinate Systems

Coordinate Systems in Computer Graphics



Normalized Device Coordinates

- This is the coordinate system you've been working in this far
  - 3D space
  - $x, y, z \in [-1.0, 1.0]$,
- If a vertex is outside of this box, it's *clipped*
  - the primitive associated with the clipped vertex will be modified
- Another example of partitioning the problem
  - clip in the clip coordinates space
  - convert all other spaces to clip coordinates
- That conversion is of course, a *transformation*



Moving Between Coordinate Systems

- A *transformation* moves from one coordinate system to another
- WebGL transforms are represented by 4x4 matrices
  - they are stored in *column-major order*

$$\begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix}$$

Representing all those Coordinate Systems

Transformation	Represenation	Common Types
Projection	mat4	perspective, orthographic
Viewing	mat4	“LookAt”
Model	mat4	scale, rotation, translation
Viewport	(mat2)	—

Introducing MV.js

- Set of JavaScript helper functions and types to simplify all of this
- Includes matrix and vector types
  - `mat2`, `mat3`, `mat4`, `vec2`, `vec3`, `vec4`
- Mathematical and logical functions
  - JavaScript doesn't support operator overloading (yet!)
  - `equal()`, `mult()`, `add()`
- Graphics transformation functions
  - `perspective()`, `ortho()`

Aside: Some Mathematics

- We can encode a line equation (in *slope-intercept form*) into a matrix

$$y = mx + b \Rightarrow \begin{pmatrix} y \\ 1 \end{pmatrix} = \begin{pmatrix} m & b \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ 1 \end{pmatrix}$$

Aspect Ratio

Aspect Ratio

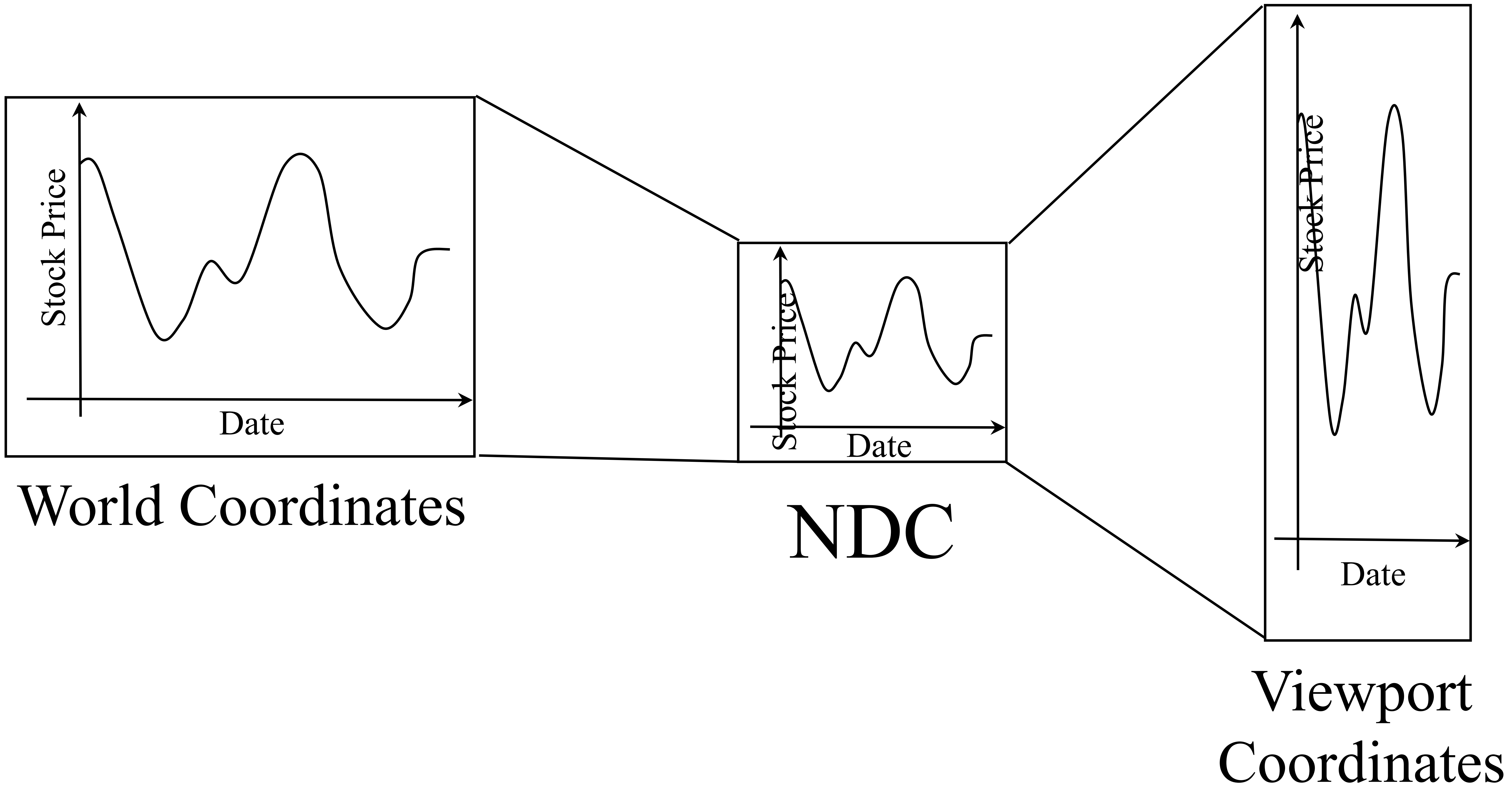
- Simply

$$\text{aspect ratio} = \frac{width}{height}$$

- Various coordinate systems have aspect ratios we'll be concerned with

- We'll need to match aspect ratios between clip and viewport coordinates

An Example


Stock Price
Date
World Coordinates
Stock Price
Date
NDC
Stock Price
Date
Viewport
Coordinates

Specifying the Viewport in WebGL

- The *viewport* is the area in the window where you can draw
- Specify the viewport using

```
gl.viewport(x, y, width, height);
```

- Initial viewport is set to the canvas size
- Update the viewport when the canvas size changes

Handling a Canvas Resize

- When a window changes size, adjust the viewport
  - the canvas's size is adjusted automatically

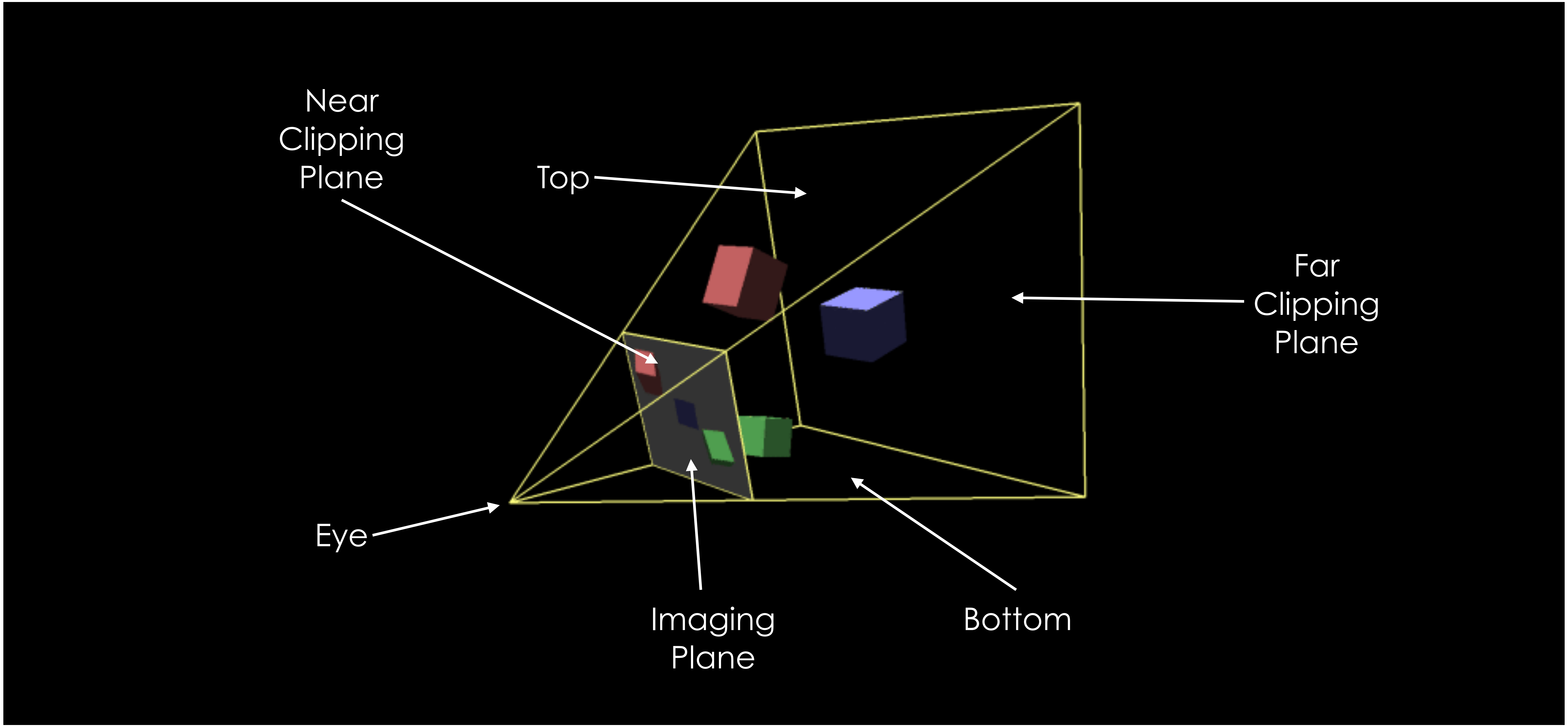
```
var canvas; // initialized in init()

function resize() {
  var w = canvas.clientWidth,
      h = canvas.clientHeight;

  gl.viewport(0, 0, w, h);

  // ... and more things to come
}

window.onresize = resize;
```

Projection Transformations

Terminology

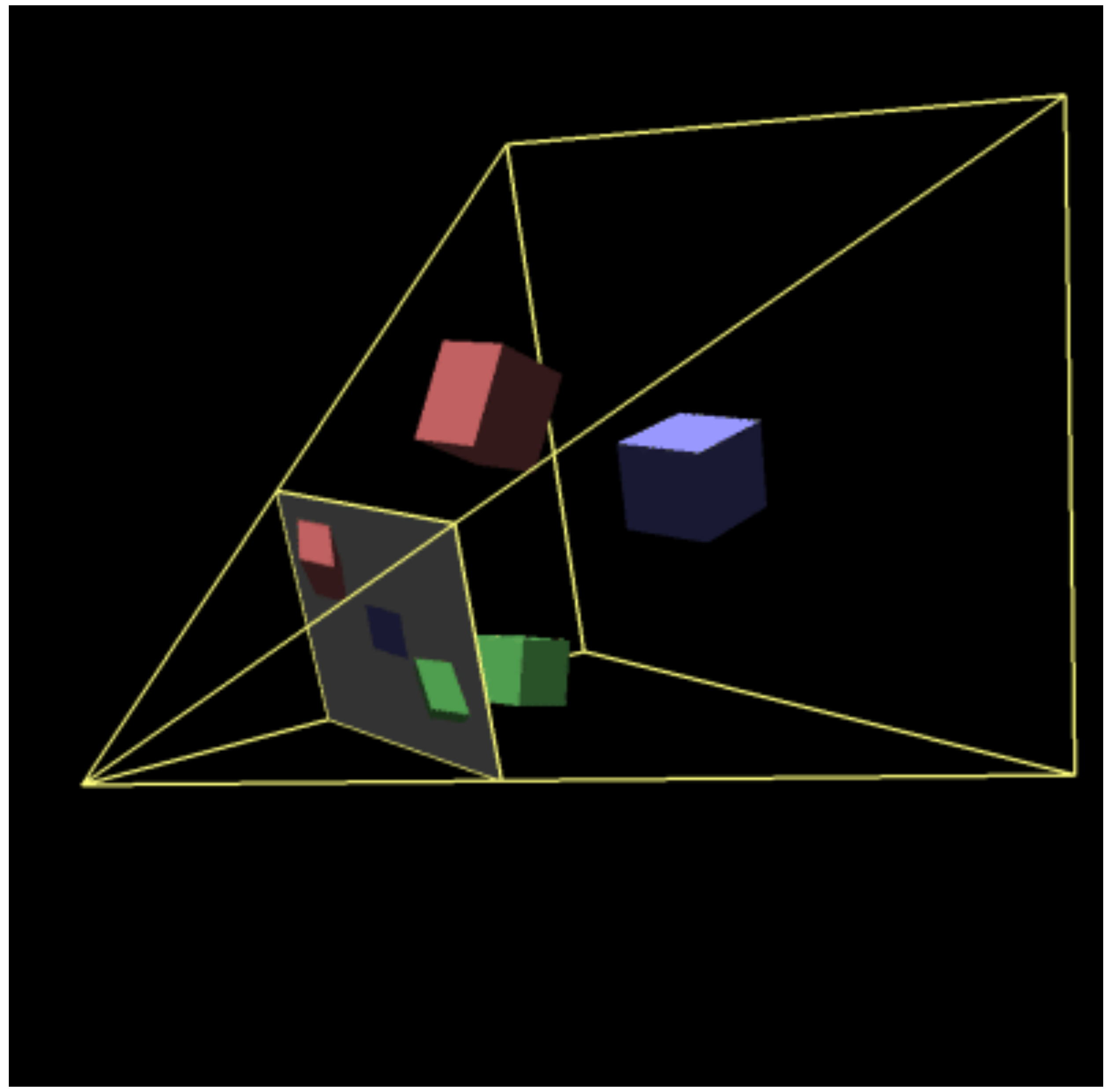
- A *viewing frustum* specifies the region of space where objects can be seen
    - that is, all visible objects in a scene are inside of the frustum
- The *imaging plane* is the plane in space where our scene is “projected”
    - it’s co-incident with the *near-clipping plane*
- The positioning of our viewing frustum controls where the imaging plane is located, and what we see

Anatomy of a Viewing Frustum



Perspective Projections

- "Works" similar to your eyes
  - objects farther from the viewer appear smaller
- Assumptions about perspective projections
  - the eye is located at the origin (apex of the pyramid)
  - line-of-sight is down the negative z-axis
- Use `perspective(fovy, aspect, n, f)` for a view-centered projection



$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(n+f)}{f-n} & \frac{-2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Computing a Perspective Projection

- mapping `perspective()`'s parameters to the matrix elements

$$t = n \cdot \tan\left(\frac{fovy}{2}\right)$$

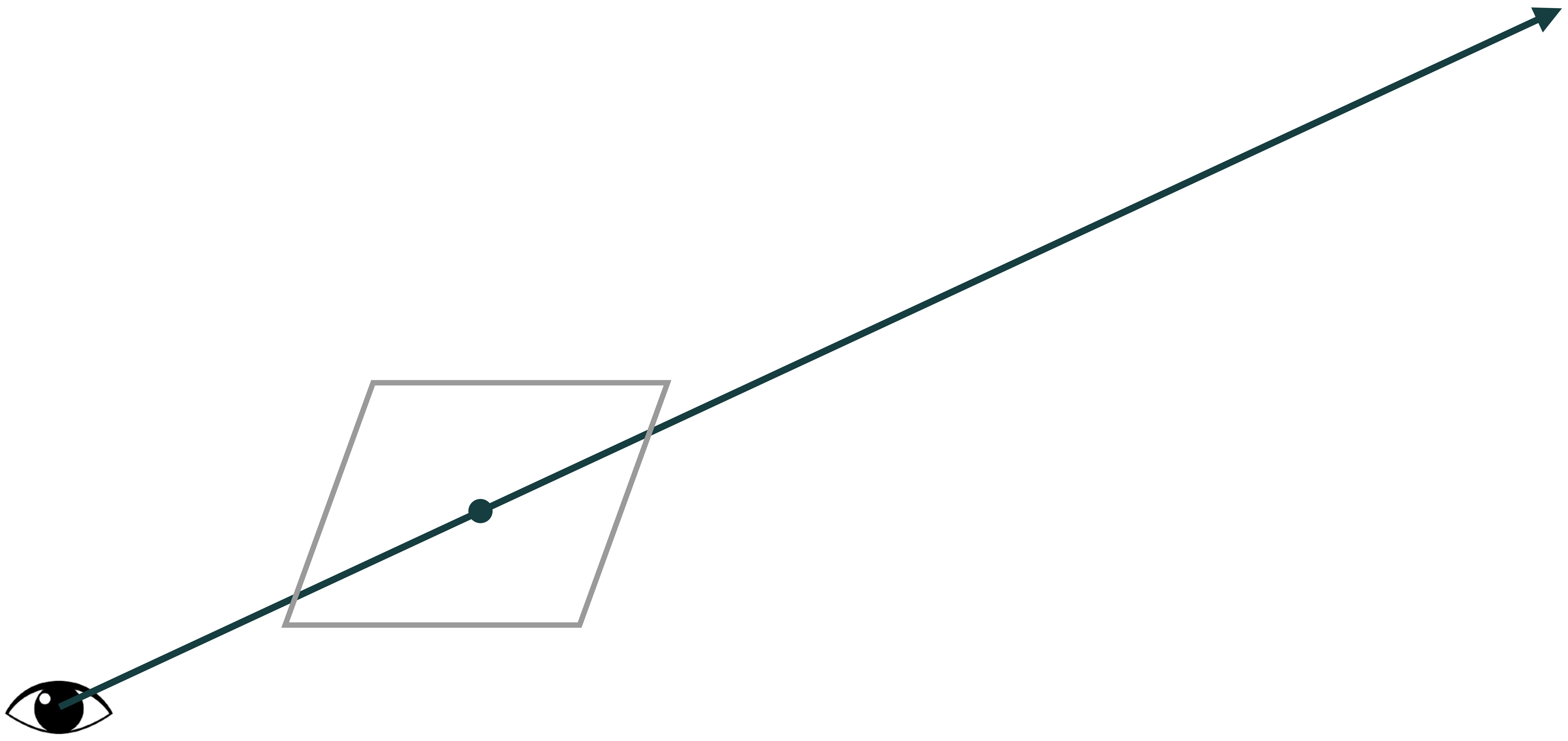
$$b = -t$$

$$r = t \cdot aspect$$

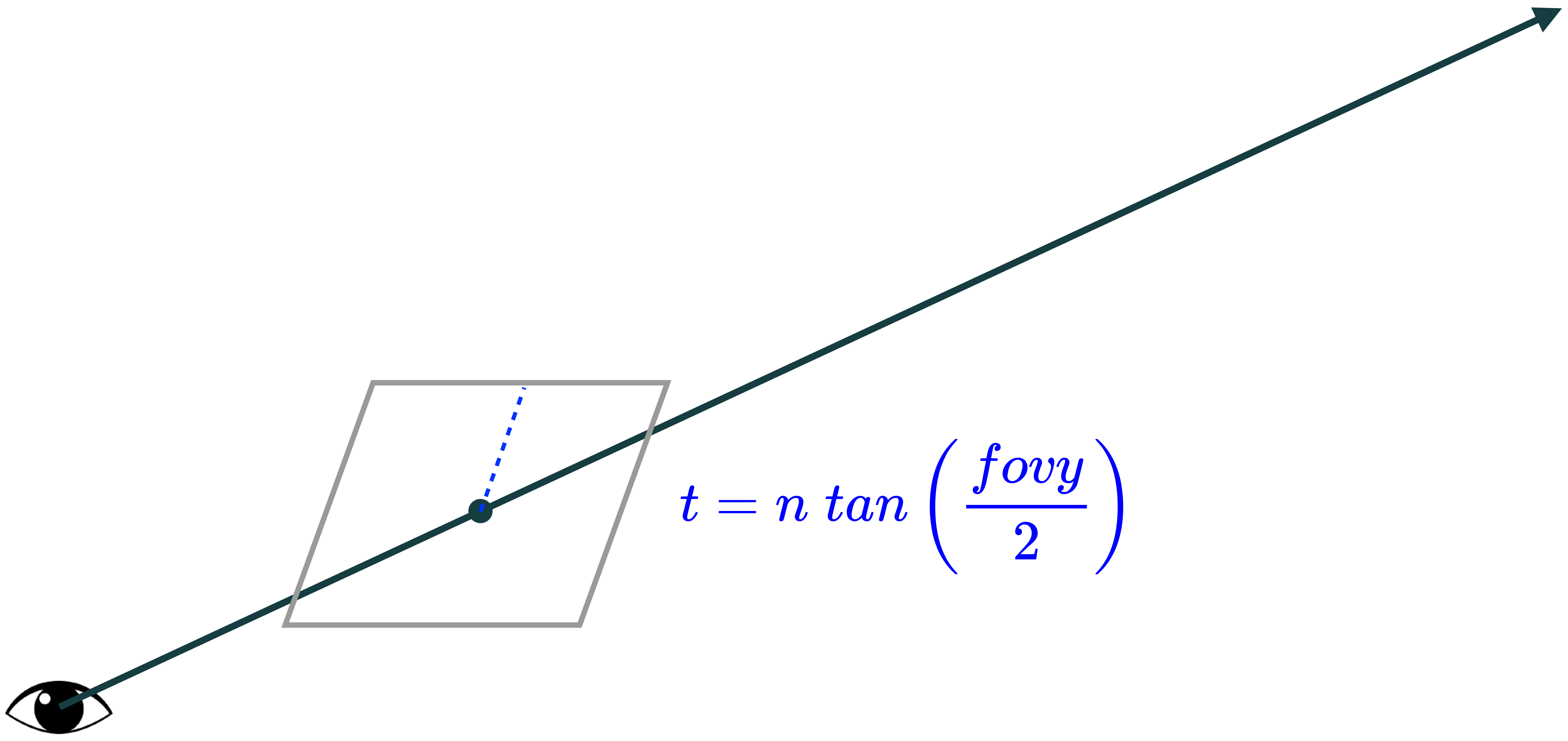
$$l = -r$$

Building a Perspective Viewing Frustum

Building a Perspective Viewing Frustum

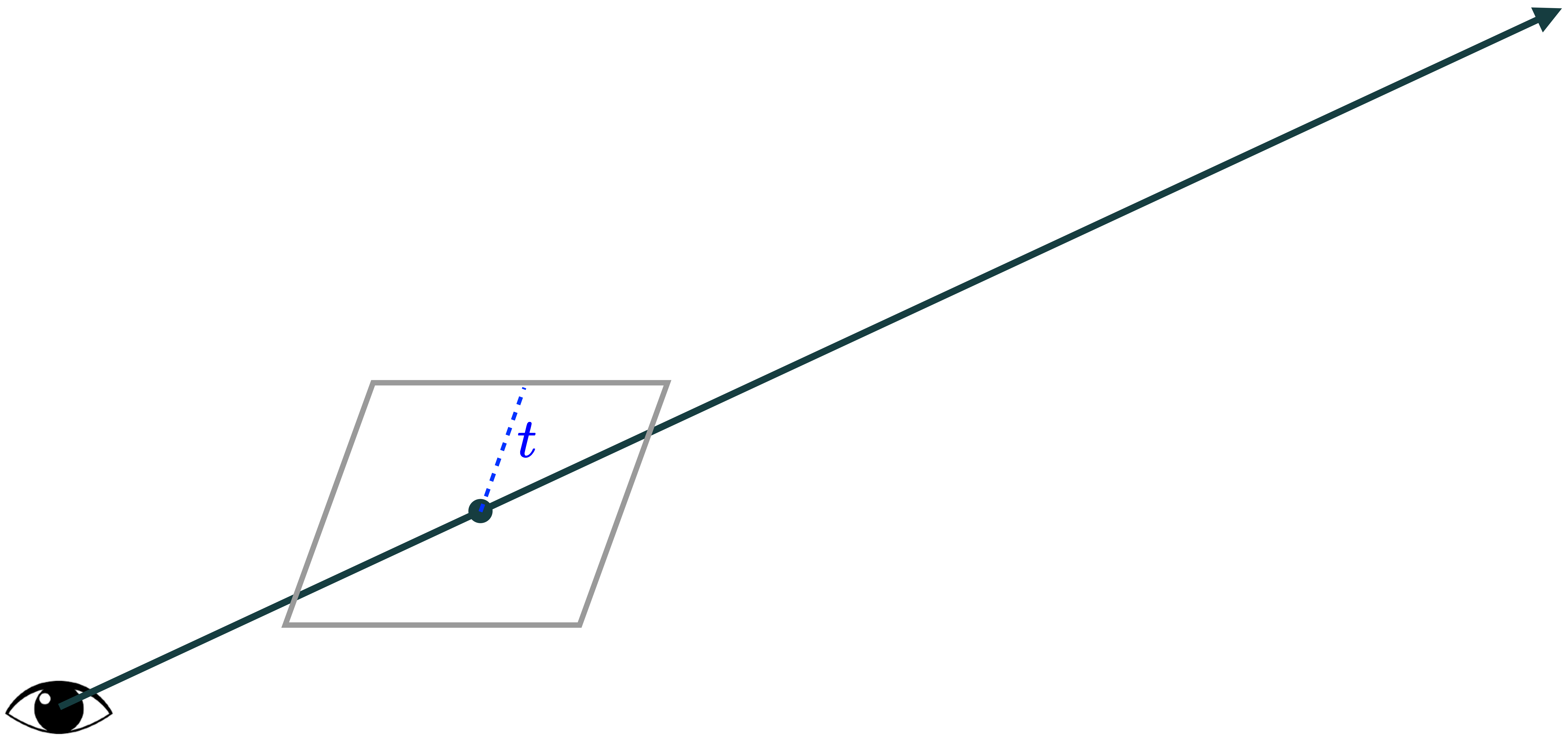


Building a Perspective Viewing Frustum

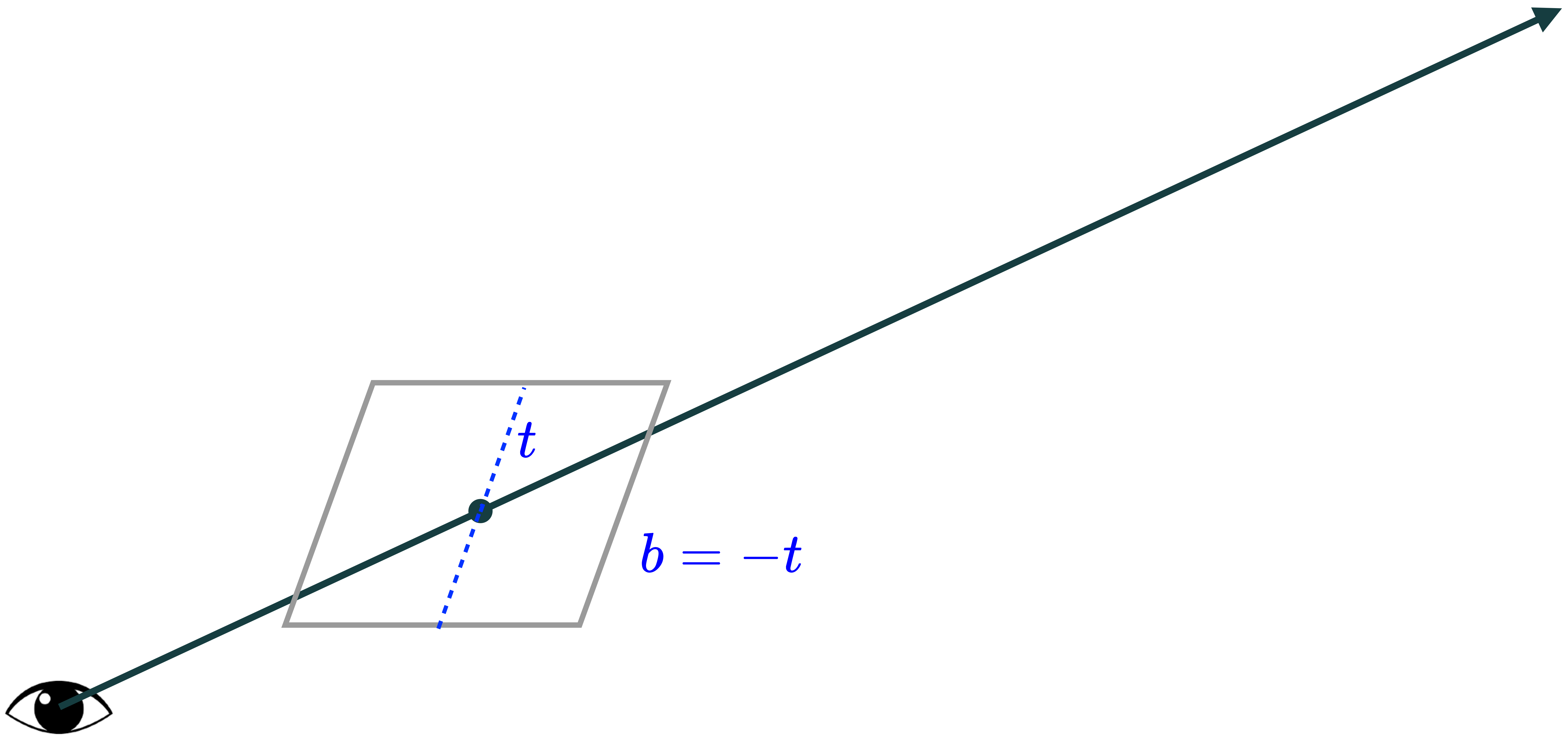


$$t = n\ tan\left(\frac{fovy}{2}\right)$$

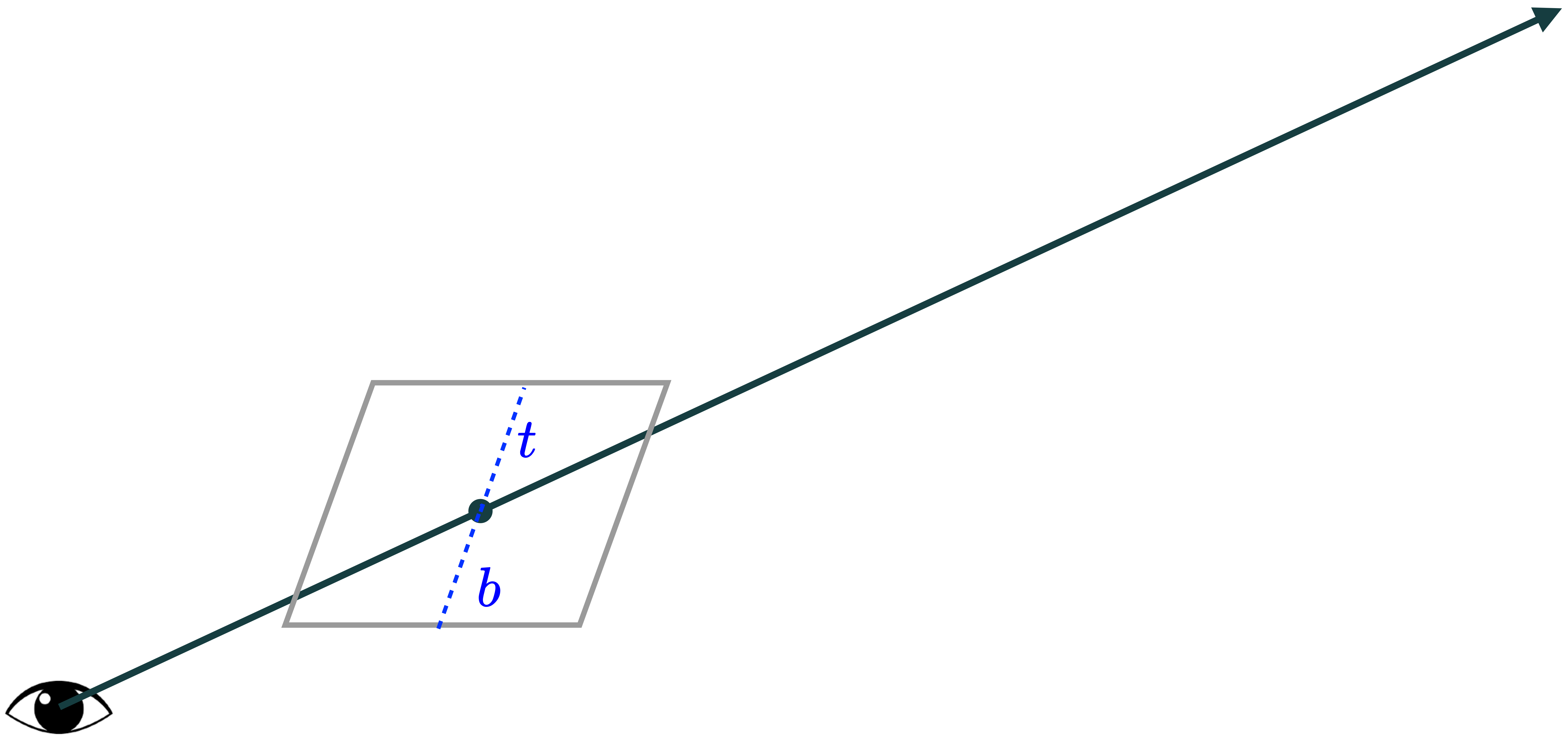
Building a Perspective Viewing Frustum



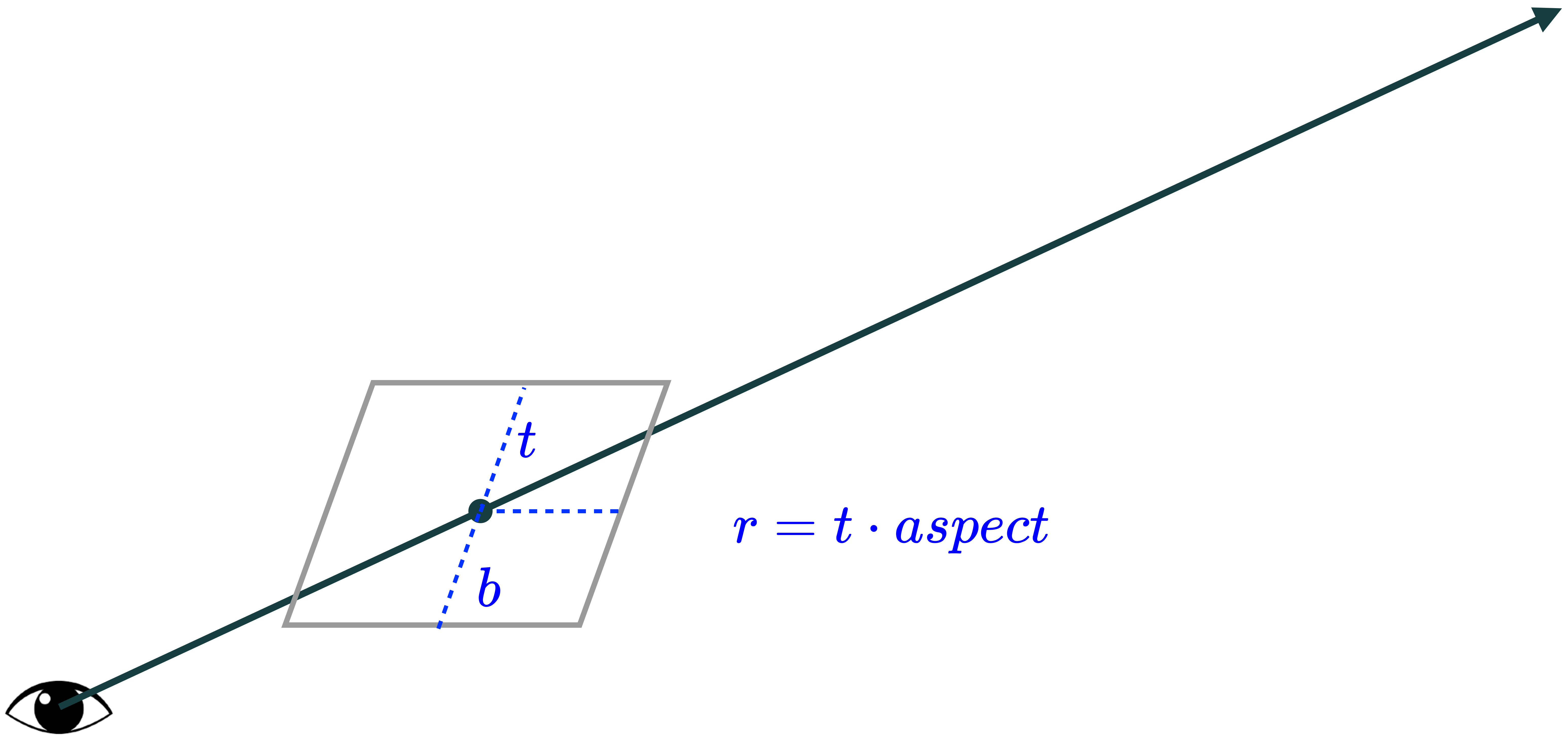
Building a Perspective Viewing Frustum



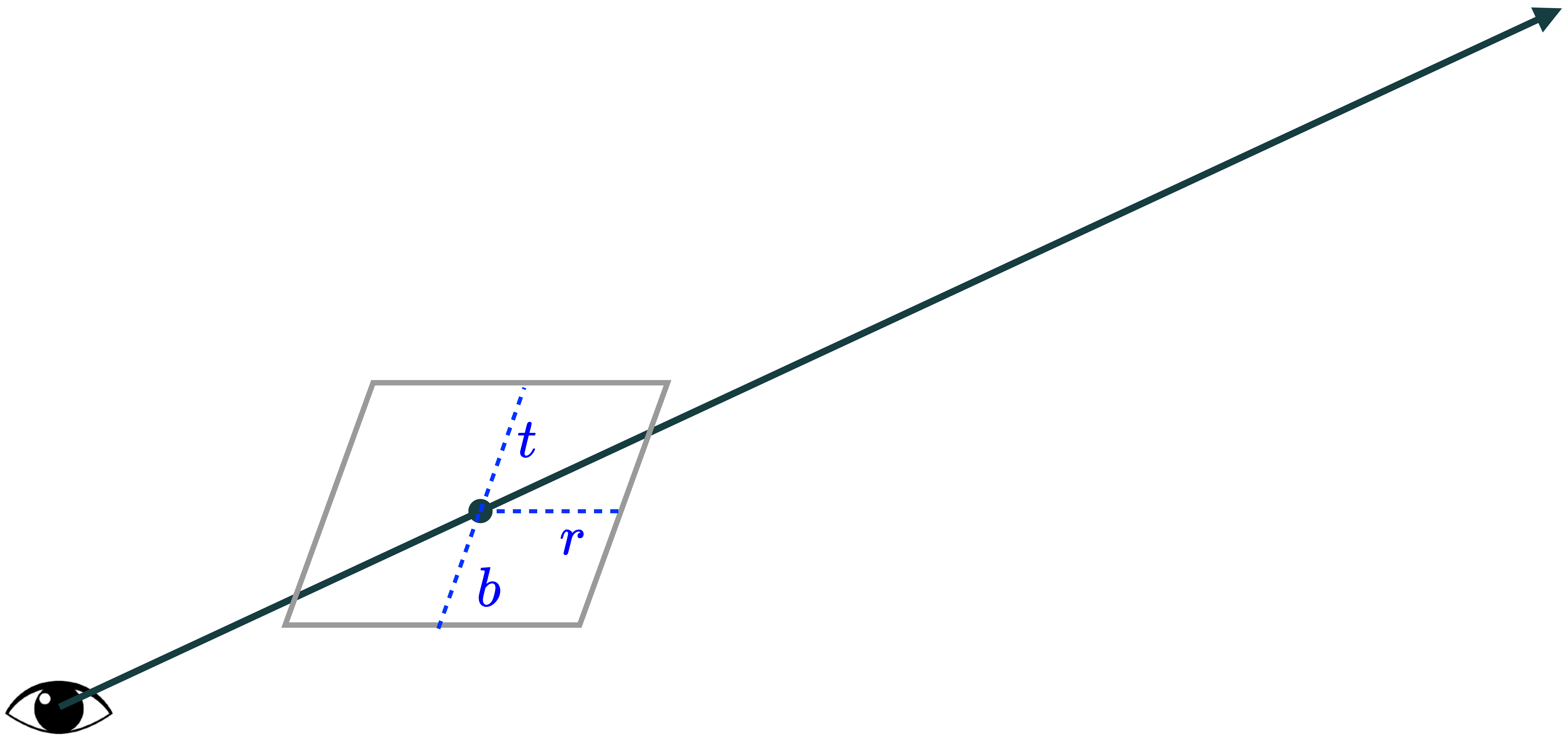
Building a Perspective Viewing Frustum



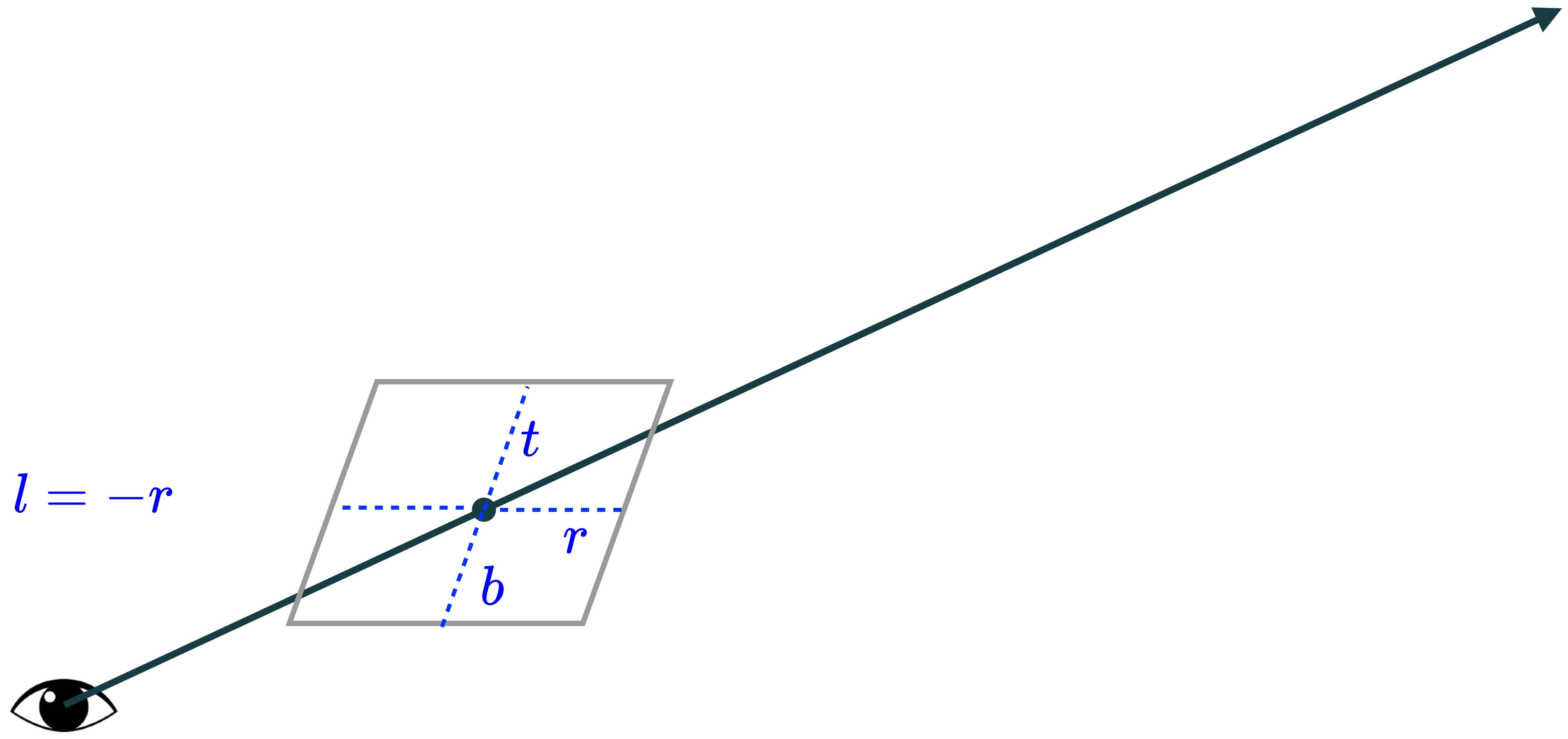
Building a Perspective Viewing Frustum



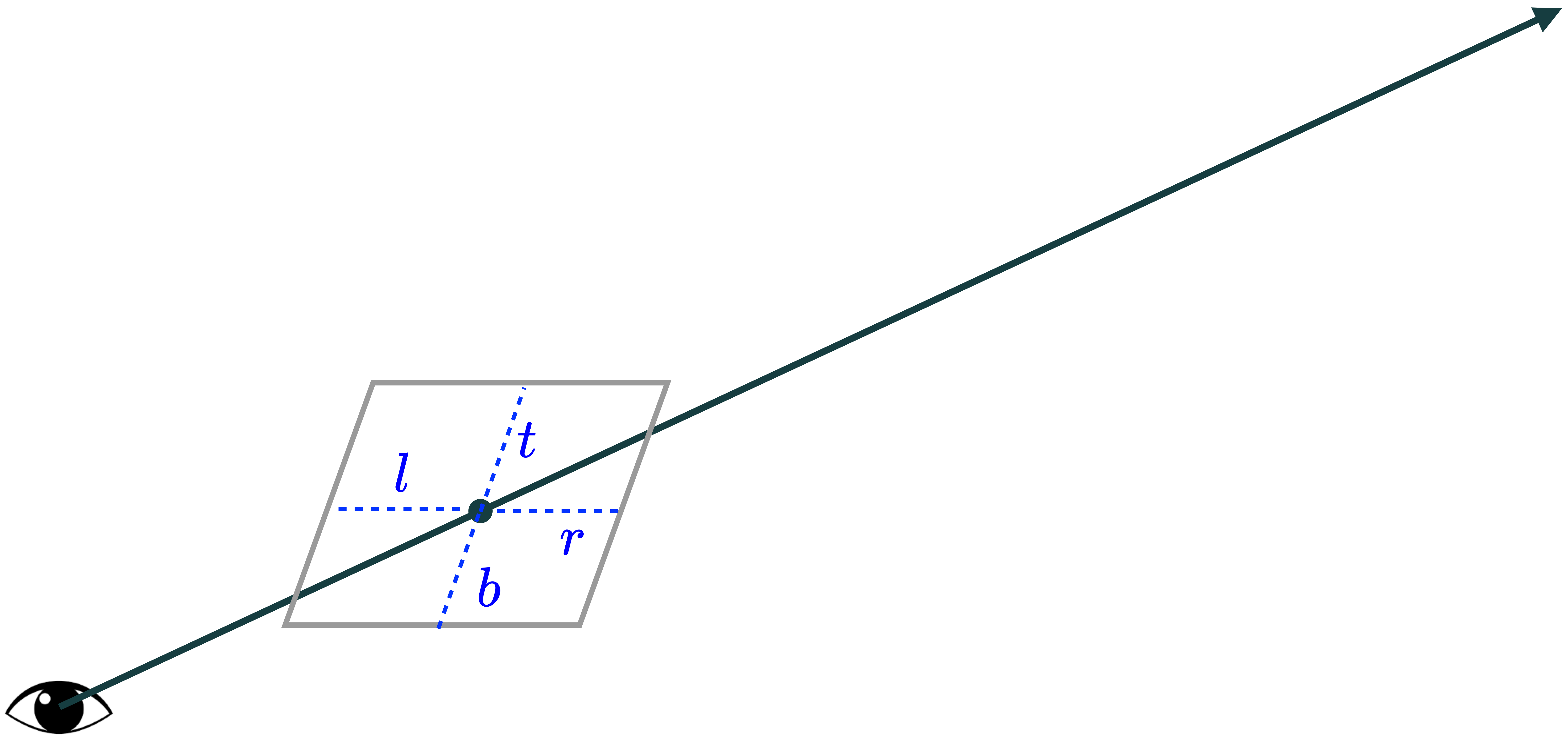
Building a Perspective Viewing Frustum



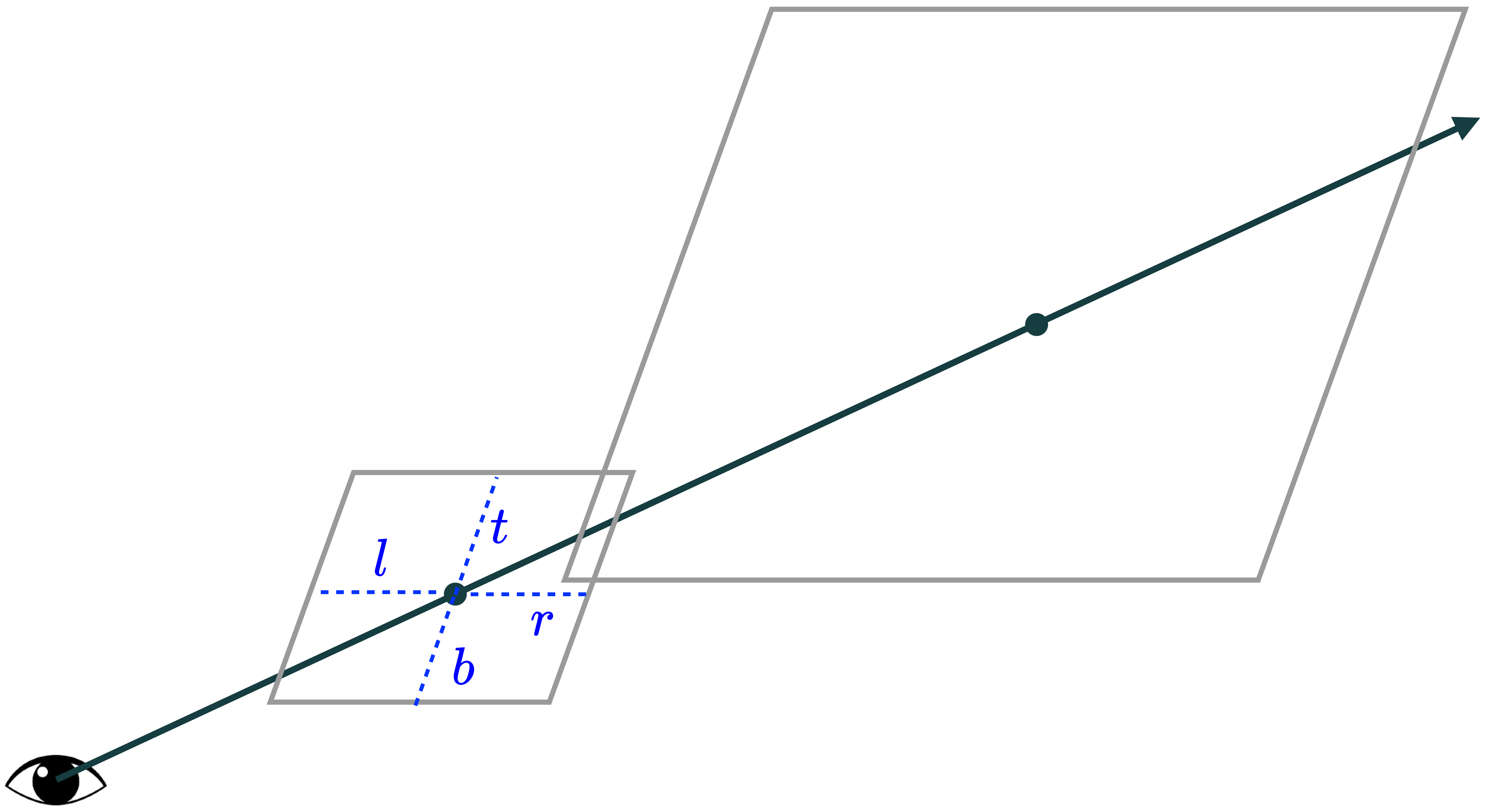
Building a Perspective Viewing Frustum



Building a Perspective Viewing Frustum



Building a Perspective Viewing Frustum



Building a Perspective Viewing Frustum