Computer Graphics

Lecture 6

Introduction to Ray-Tracing

Cameras and Ray Generation

Goals

- Understand the high-level distinction between object-order rendering and image-order rendering.
- Know the mathematical definition of a ray.
- Know how to generate viewing rays for a canonical perspective camera.

Announcements: Scott is sick edition

- Class probably remote tomorrow, Wednesday; watch for announcements.
- Today's OH on (Zoom? Discord?)

Announcements

Announcements

- HW1 out, due in 1 week

Announcements

- HW1 out, due in 1 week
- A1 is out (later today), due Wednesday 10/12.
  - A1 is (much) bigger than A0. Please get started early.

Announcements

- HW1 out, due in 1 week
- A1 is out (later today), due Wednesday 10/12.
  - A1 is (much) bigger than A0. Please get started early.
- There is one video (13 minutes) to watch for tomorrow.

Announcements

- HW1 out, due in 1 week
- A1 is out (later today), due Wednesday 10/12.
  - A1 is (much) bigger than A0. Please get started early.
- There is one video (13 minutes) to watch for tomorrow.
- Discord communication norms:
  - I will ignore DMs
  - Please don't @mention me unless there's something time-sensitive and widely applicable

Where were we?

Pseudocode for 3D graphics:

```
Create a model of a scene
Render an image of the model
```

Triangle(**a**, **b**, **c)**
Sphere(**c**, r)
meshgen.jl (A1)

Where were we?

Pseudocode for 3D graphics:

```
Create a model of a scene
Render an image of the model
```

```
For each pixel:
   if inside triangle:
    color pixel
```

Two Rendering Algorithms

Two Rendering Algorithms

```
for each object in the scene {
  for each pixel in the image {
    if (object affects pixel) {
      do something
    }
  }
}
```

object order
or
rasterization

Two Rendering Algorithms

```
for each object in the scene {
  for each pixel in the image {
    if (object affects pixel) {
      do something
    }
  }
}
```

object order
or
rasterization

```
for each pixel in the image {
  for each object in the scene {
    if (object affects pixel) {
      do something
    }
  }
}
```

image order
or
ray tracing

Two Rendering Algorithms

Q: Which of these did we do in A0?

Two Rendering Algorithms

```
for each object in the scene {
  for each pixel in the image {
    if (object affects pixel) {
      do something
    }
  }
}
```

A

B

object order
or
rasterization

Q: Which of these did we do in A0?

Two Rendering Algorithms

```
for each object in the scene {
  for each pixel in the image {
    if (object affects pixel) {
      do something
    }
  }
}
```

A

object order
or
rasterization

```
for each pixel in the image {
  for each object in the scene {
    if (object affects pixel) {
      do something
    }
  }
}
```

B

image order
or
ray tracing

Q: Which of these did we do in A0?

Two Rendering Algorithms

Today: starting here.

Two Rendering Algorithms

```
for each object in the scene {
  for each pixel in the image {
    if (object affects pixel) {
      do something
    }
  }
}
```

object order
or
rasterization

Today: starting here.

Two Rendering Algorithms

```
for each object in the scene {
  for each pixel in the image {
    if (object affects pixel) {
      do something
    }
  }
}
```

object order
or
rasterization

```
for each pixel in the image {
  for each object in the scene {
    if (object affects pixel) {
      do something
    }
  }
}
```

image order
or
ray tracing

Today: starting here.

How do we make images?

Color pixels canvas[i,j] = color

blitting(?) into a buffer

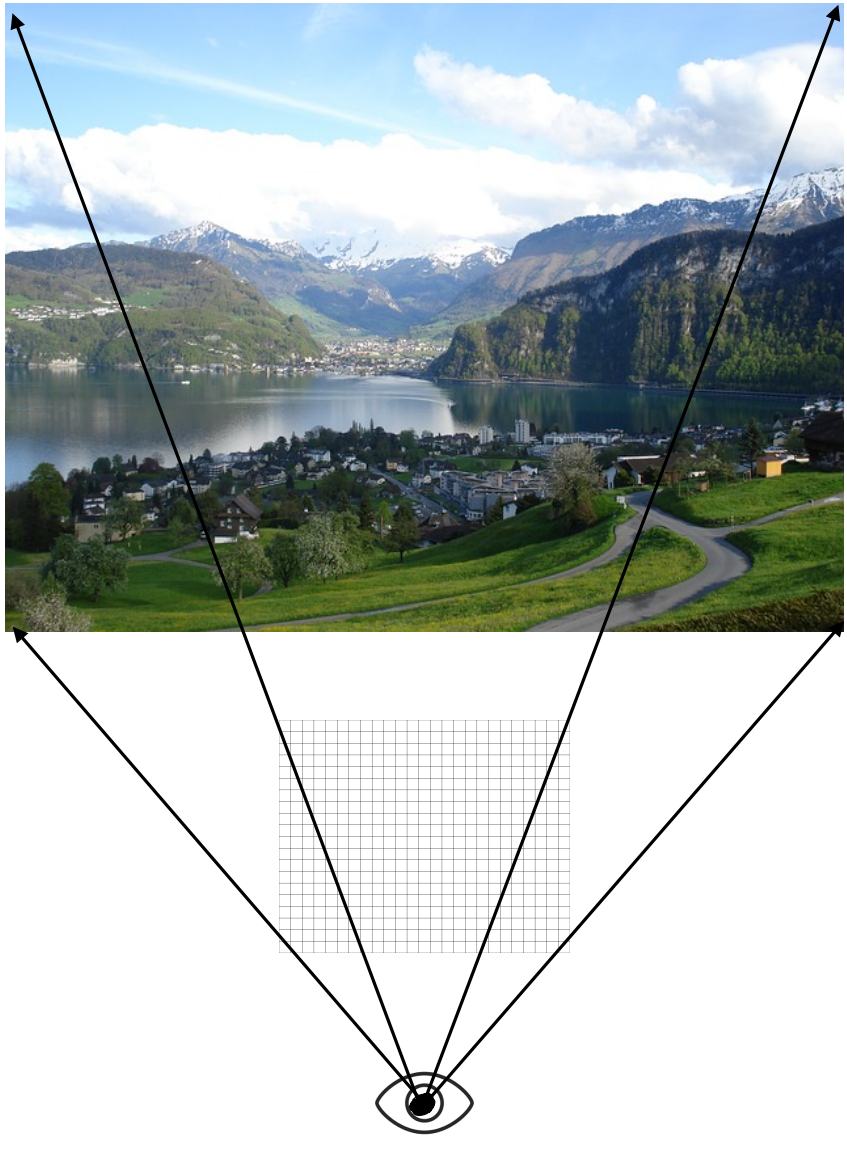
pencils/pens/chalk/paint

math → images

Cameras!

How do we make images?

- IRL:
  - pencils, paintbrushes, watercolors, etc
  - eyes
  - **cameras**
- On computers:
  - MS paint
  - manually writing pixel values into Julia arrays
  - **virtual cameras**



The Camera Conundrum:

The Camera Conundrum:

The world is 3D

The Camera Conundrum:

The world is 3D

Images are 2D

The Camera Conundrum:

The world is 3D

Images are 2D

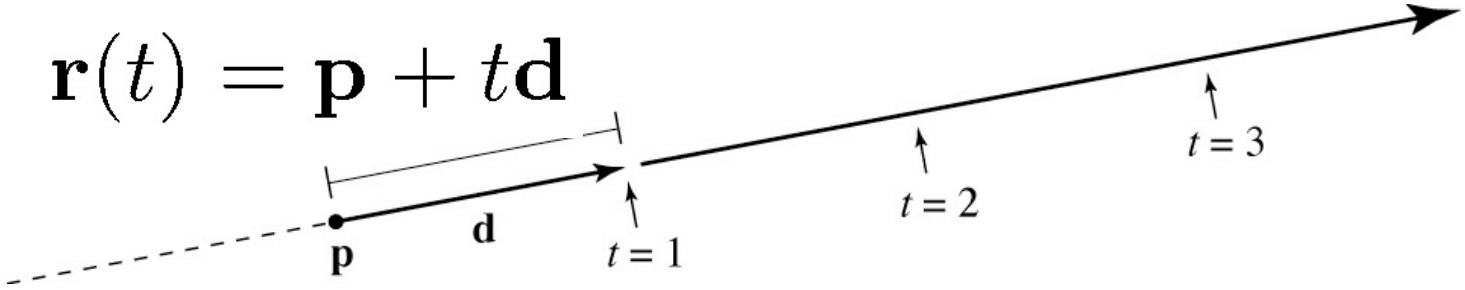
we gotta lose a D somehow

A ray is half a line.

We'll describe rays using:

- An *origin* (**p**) where the ray begins
- A *direction* (**d**) in which the ray goes

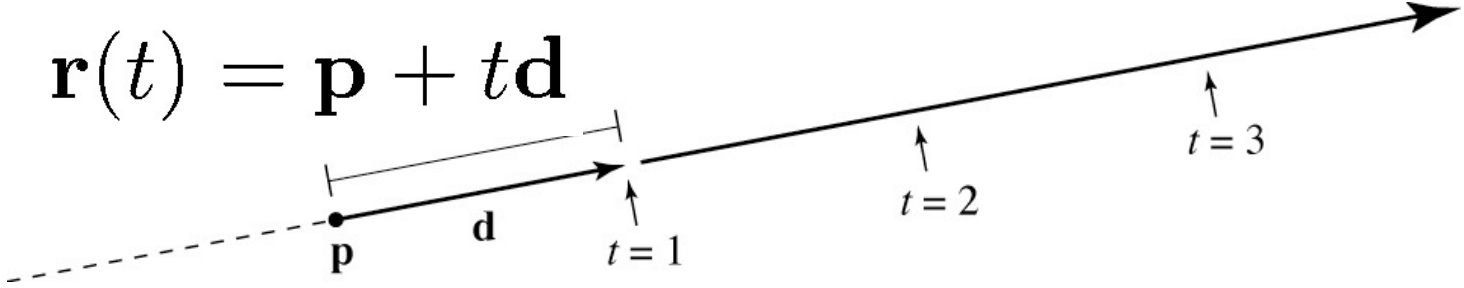
$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$



A ray is half a line.

We'll describe rays using:

- An *origin* (**p**) where the ray begins
- A *direction* (**d**) in which the ray goes

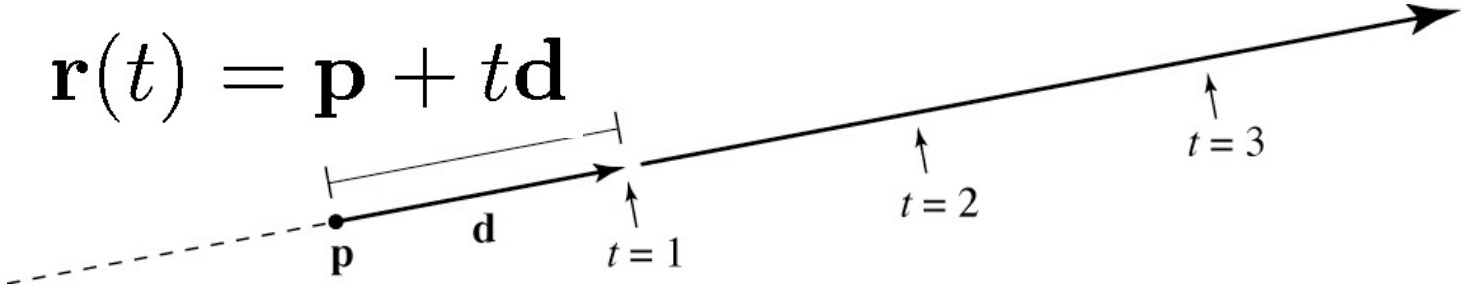


- This is a *parametric equation*: it **generates** points on the line

A ray is half a line.

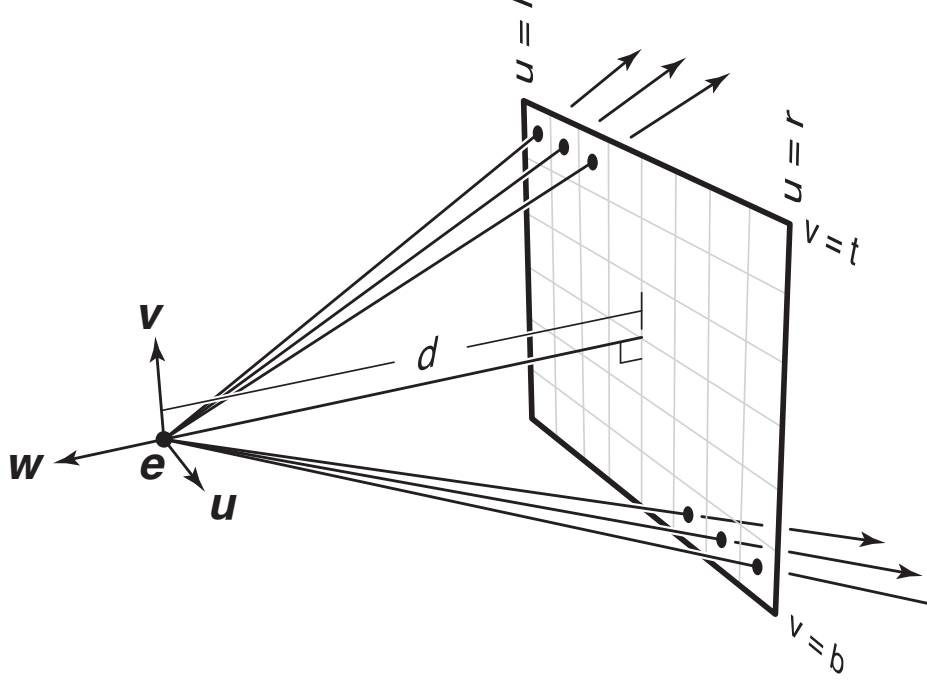
We'll describe rays using:

- An *origin* (**p**) where the ray begins
- A *direction* (**d**) in which the ray goes



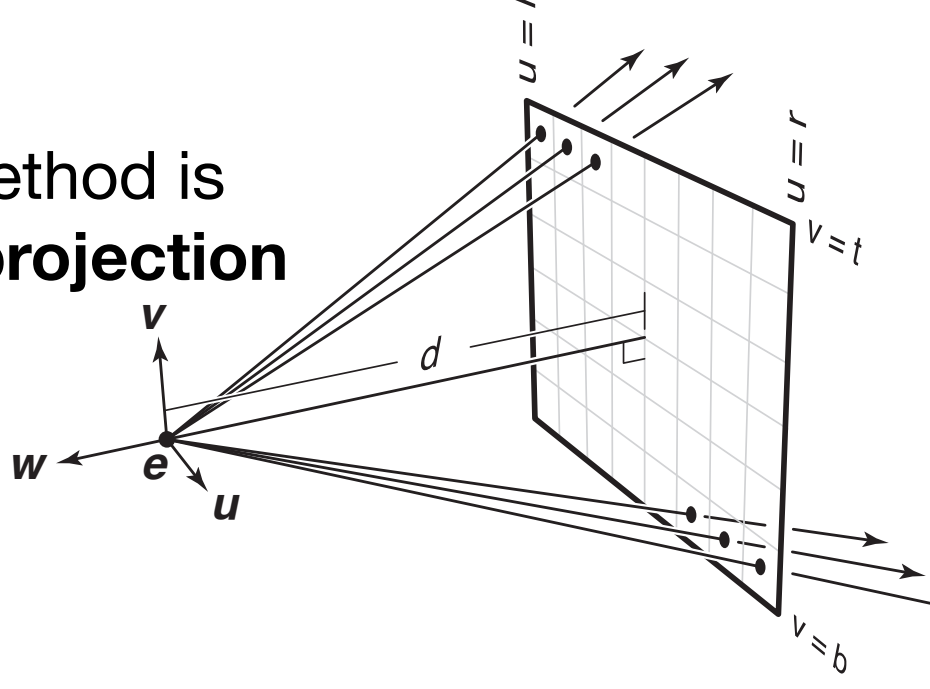
- This is a *parametric equation*: it **generates** points on the line
- The set of points with $t > 0$ gives all points on the ray

Projections: ways to lose a D



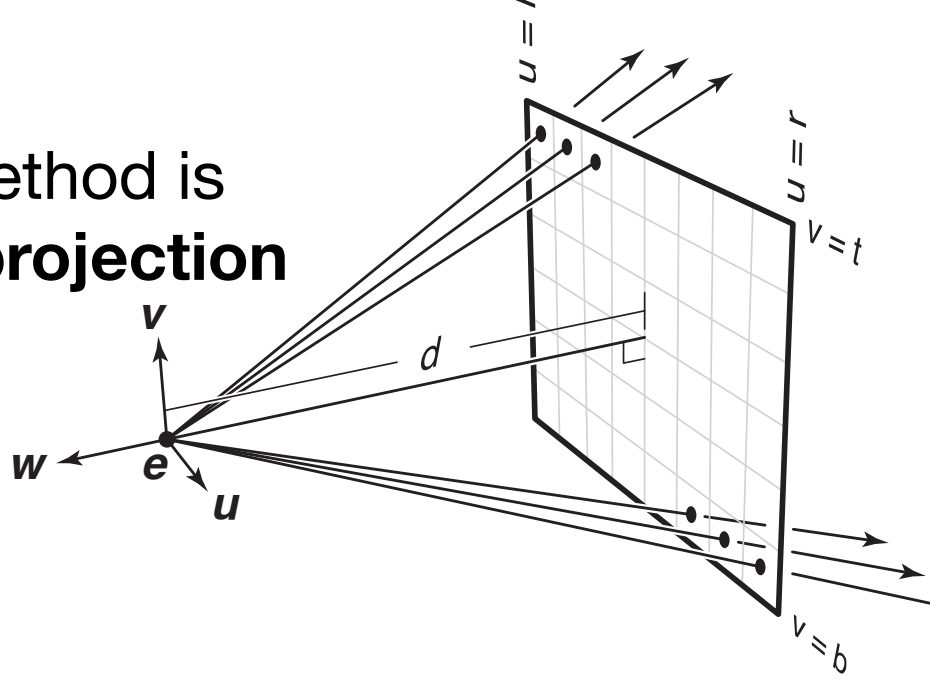
Projections: ways to lose a D

- The picture-frame method is called **perspective projection**



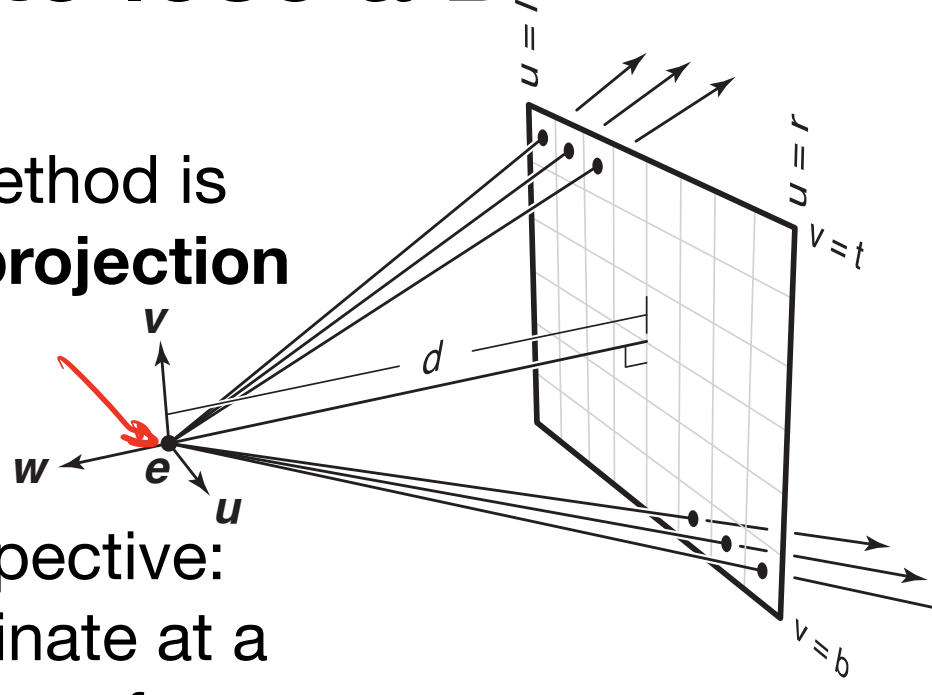
Projections: ways to lose a D

- The picture-frame method is called **perspective projection**



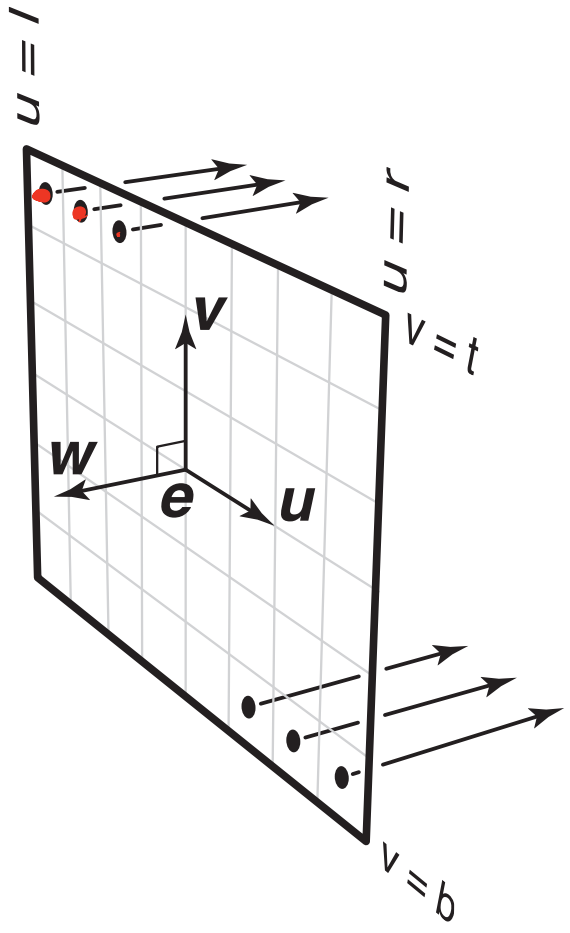
Projections: ways to lose a D

- The picture-frame method is called **perspective projection**



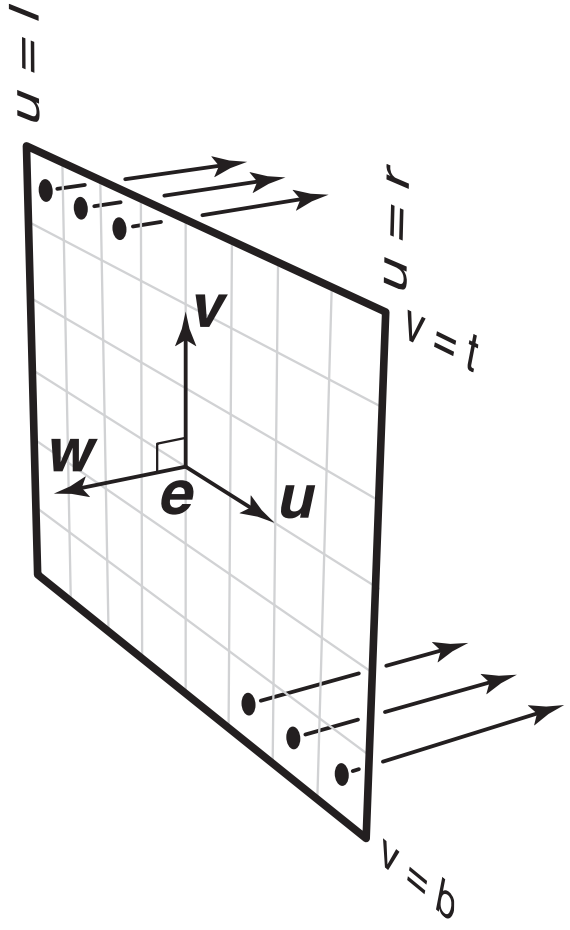
- Key property of perspective: all **viewing rays** originate at a single point, the *center of projection*, or *eye*.

Projections: ways to lose a D



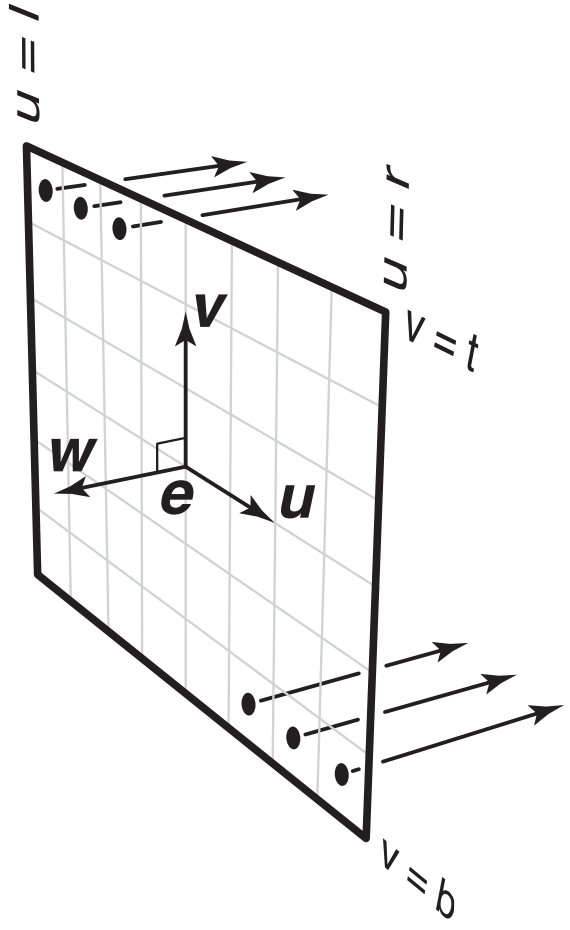
Projections: ways to lose a D

- Another common one is **parallel projection**



Projections: ways to lose a D

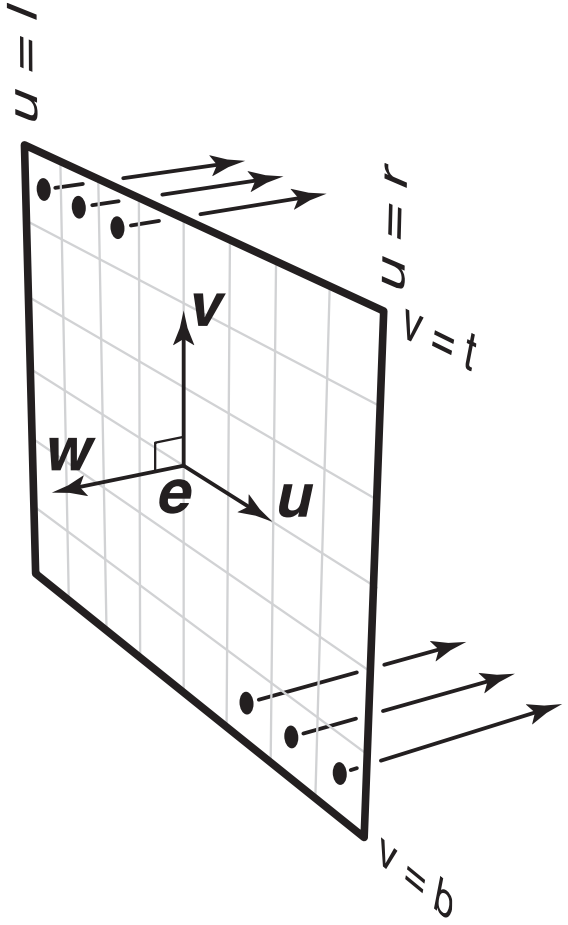
- Another common one is **parallel projection**



Projections: ways to lose a D

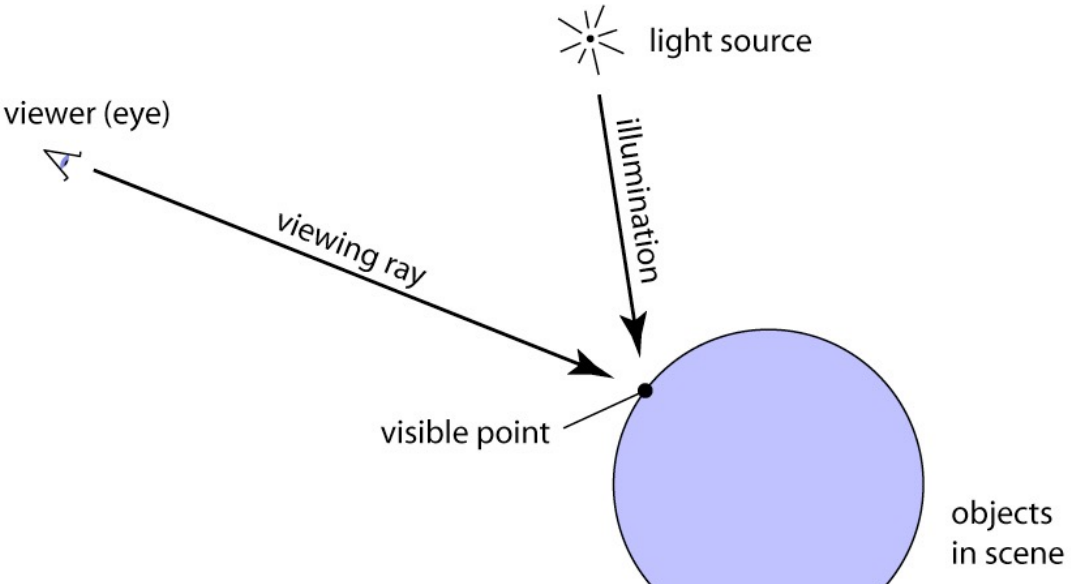
- Another common one is **parallel projection**

- Key property of parallel projections:
  all **viewing rays** are parallel



Ray Tracing: Pseudocode

```
for each pixel:
    generate a viewing ray for the pixel
    find the closest object it intersects
    determine the color of the object
```

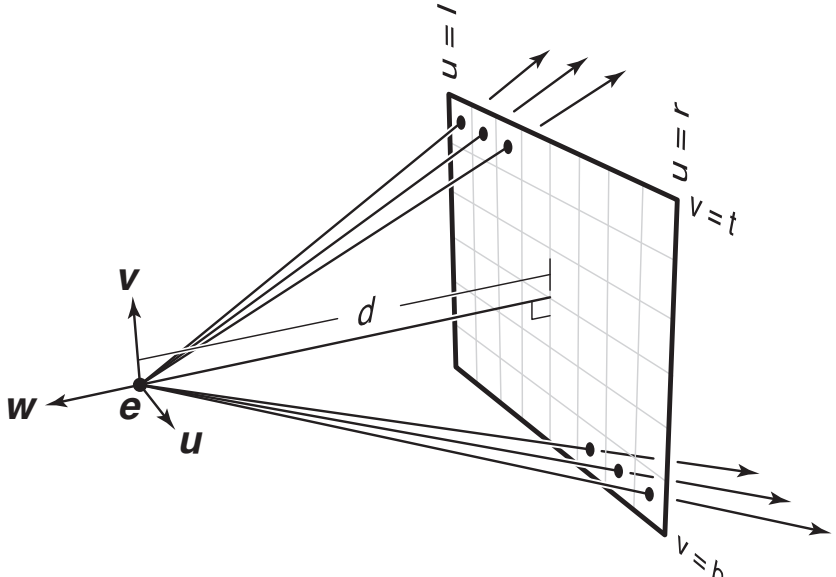


Viewing Rays

are determined by the **position** and **orientation** of the camera

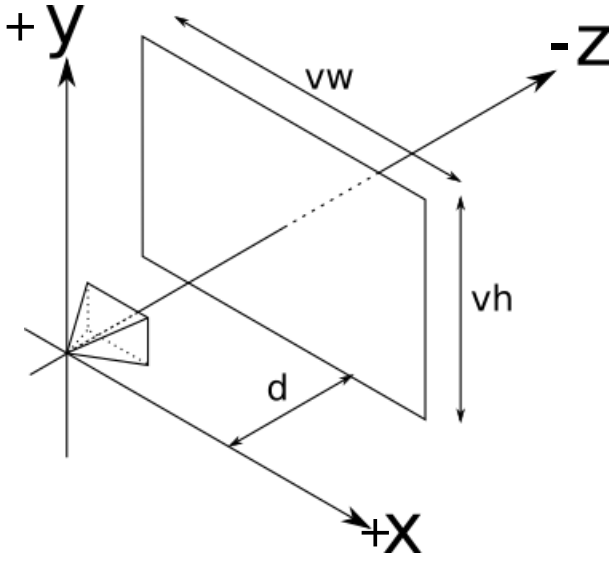
- For perspective projection, viewing rays originate at the **eye**.

- The direction varies depending on the pixel.



Let's start with a simple camera

- Eye is at the origin (0, 0, 0)
- Looking down the **negative** z axis
- Viewport is parallel to the xy plane
- vh = vw = 1
- d = 1



What is the 3D viewing ray for pixel (i, j)?

Whiteboard: (i, j) to (x, y)

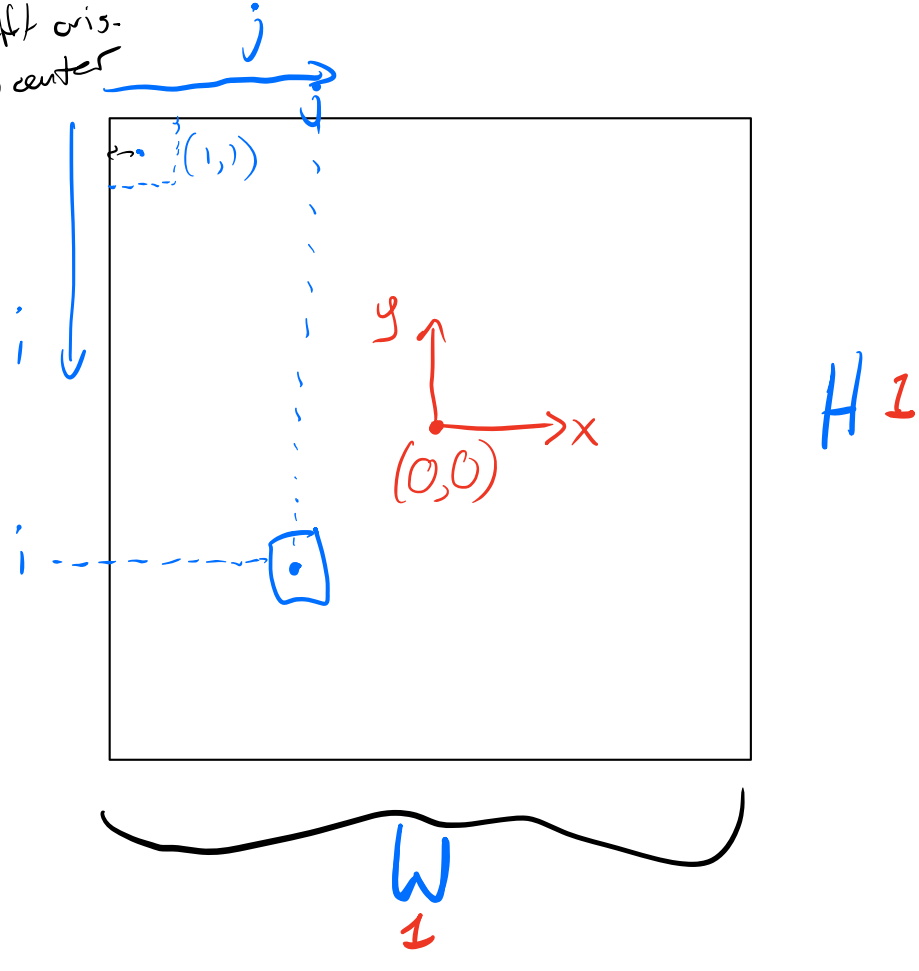
$$x = \frac{j - \frac{1}{2}}{W} - \frac{1}{2}$$

half pixel left

Scale to size 1

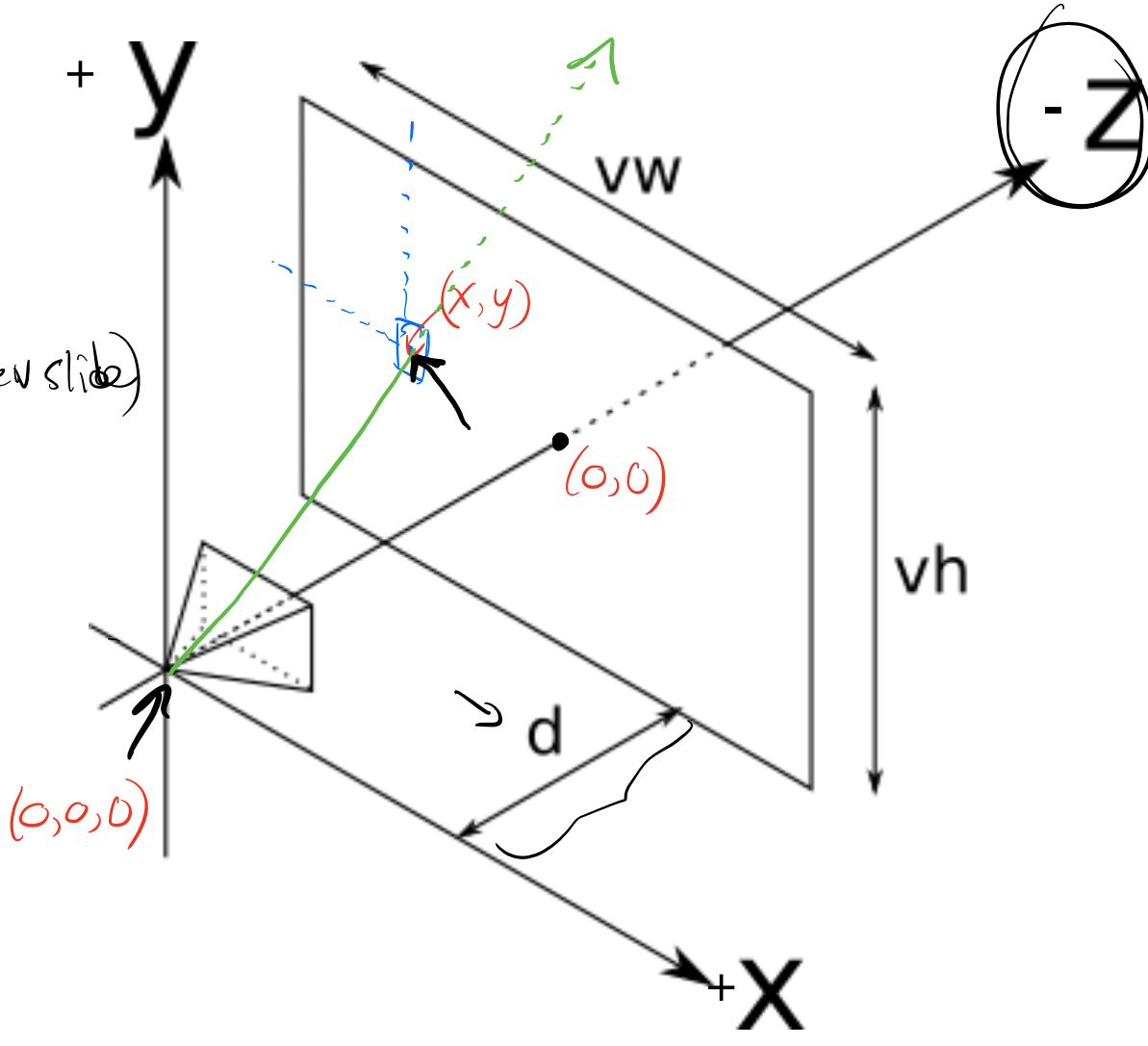
← Shift origin to center

$$y = -\left(\frac{\left(i - \frac{1}{2}\right)}{H} - \frac{1}{2}\right)$$



orig: $\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$

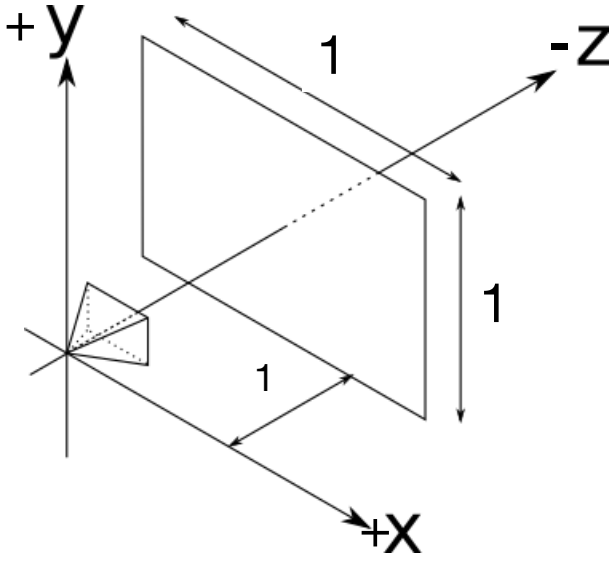
dir: $\begin{bmatrix} x \\ y \\ -1 \end{bmatrix}$ ← (prev slide)



Viewing rays for the canonical camera

$$x = \frac{j - \frac{1}{2}}{W} - \frac{1}{2}$$

$$y = -\left(\frac{i - \frac{1}{2}}{H} - \frac{1}{2} \right)$$



Origin (**p**): (0, 0, 0)
Direction (**d**): (*x*, *y*, -1)

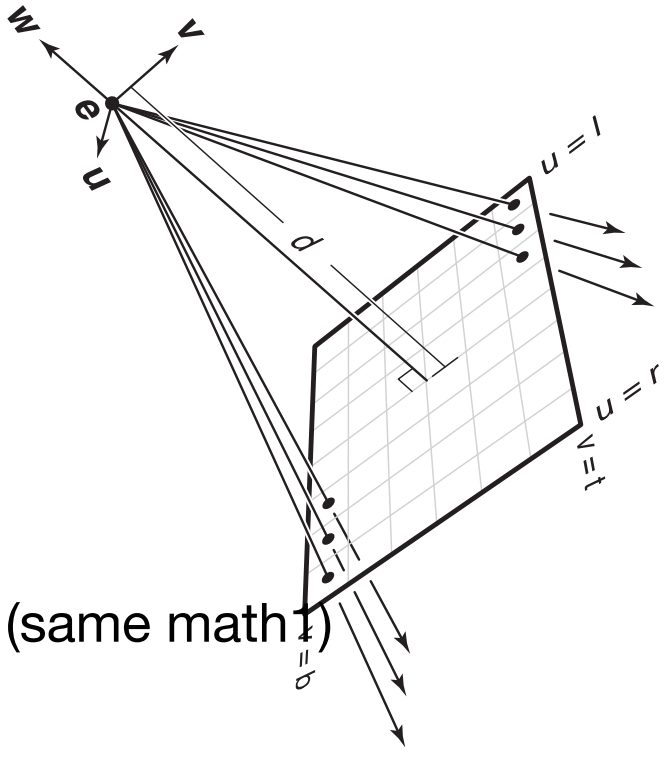
Problems - in groups

1. Generate an example viewing ray
2. Intersect the ray with a plane in the scene
3. Generalize camera model by removing assumptions:
   - Eye is **not** at the origin (0, 0, 0)
   - vh **!=** vw **!=** 1
   - d **!=** 1

What if I want to point the camera somewhere else?

The camera's pose is defined by a **coordinate frame:**

- **u** points right from the eye
- **v** points up from the eye
- **w** points back from the eye



Given this, we can generate a viewing ray as follows:

1. Turn (i,j) into *u, v* instead of x, y (same math!)
2. Viewing ray in (x, y, z) world is:
   origin = eye
   direction = u * **u** + v * **v** + -d * **w**