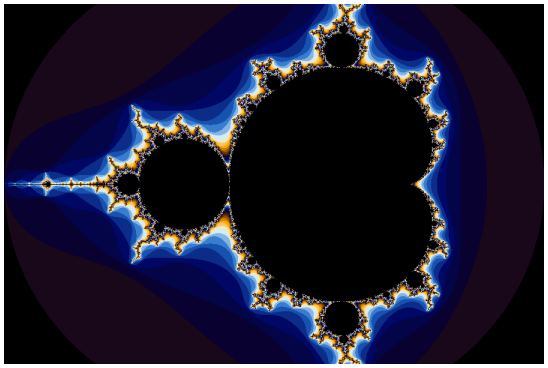
Impact Lab 2025: Programming Fundamentals
Lecture 7: Image Processing and Mandelbrot

Summer 2025

School of Computing and Data Science

Wentworth Institute of Technology



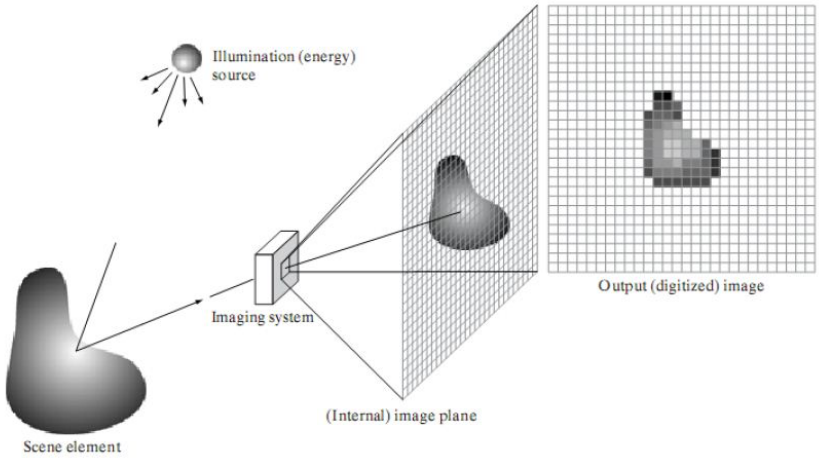
Wentworth
Computing & Data Science

Topics for Today

- **Image Processing**
- Mandelbrot Set

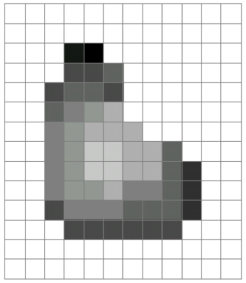
Wentworth
Computing & Data Science

What is an Image?



Wentworth
Computing & Data Science

What is an Image?



=

255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	20	0	255	255	255	255	255	255	255	255
255	255	255	75	75	75	255	255	255	255	255	255	255
255	255	75	95	95	75	255	255	255	255	255	255	255
255	255	96	127	145	175	255	255	255	255	255	255	255
255	255	127	145	175	175	175	255	255	255	255	255	255
255	255	127	145	200	200	175	175	95	255	255	255	255
255	255	127	145	200	200	175	175	95	47	255	255	255
255	255	127	145	145	175	127	127	95	47	255	255	255
255	255	74	127	127	127	95	95	95	47	255	255	255
255	255	255	74	74	74	74	74	74	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255

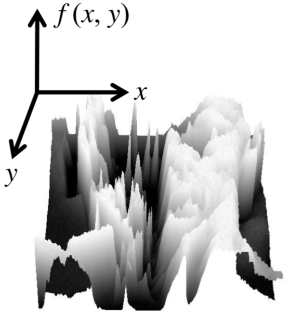
Essentially, we have one byte per pixel (0 = Black, 255 = White)

Wentworth
Computing & Data Science

What is an Image?

You can think of a greyscale image as a function, **f**, from $\mathbf{R}^2$ to **R** (basically a **2D** signal)

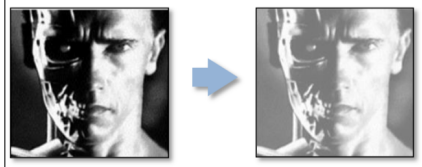
- **f(x,y)** is the intensity of the image!



So, a digital image is just a **discrete** (sampled, quantized) version of the function **f**.

Image Transformation

As you can imagine, you can apply an operator to this image 'function'



$g(x,y) = f(x,y) + 20$

$g(x,y) = f(-x,y)$

We'll talk about a special kind of function called a **convolution**.

Convolution

Let F be the image, H be the kernel and G be the output:

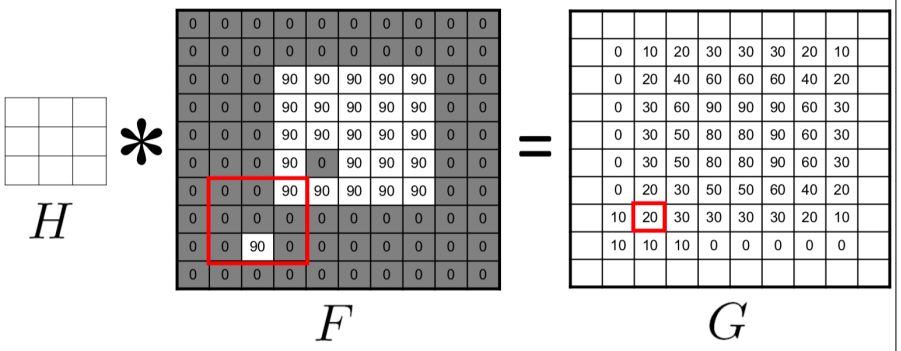
$$G(i,j) = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H(u,v)F(i-u, j-v)$$

The kernel is size (2k+1, 2k+1).

This is called the convolution operation:

$$G = H * F$$

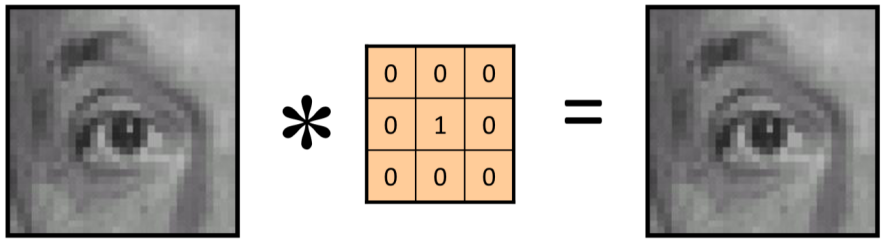
Mean Filtering



A 3x3 kernel is applied to a 3x3 region of the image.

Sum of interior pixels ÷ Size of kernel = Resulting pixel in G

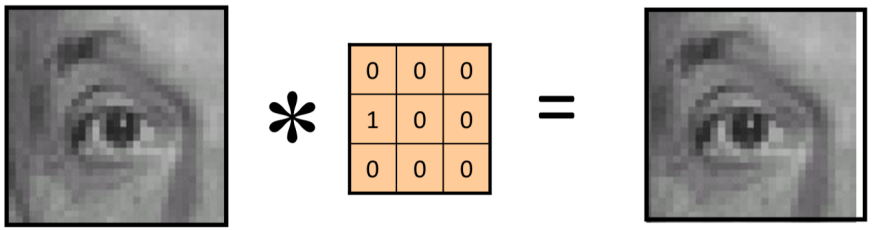
Linear Filter Examples



Original

Identical image

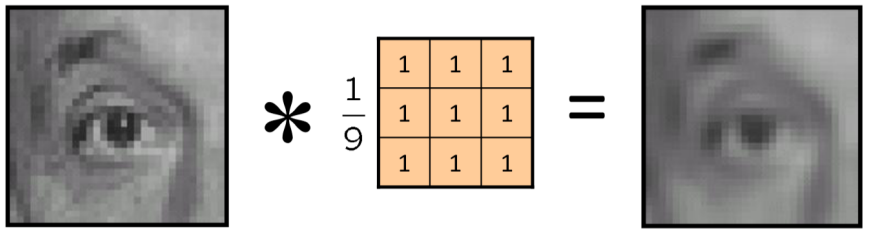
Linear Filter Examples



Original

Shifted left
By 1 pixel

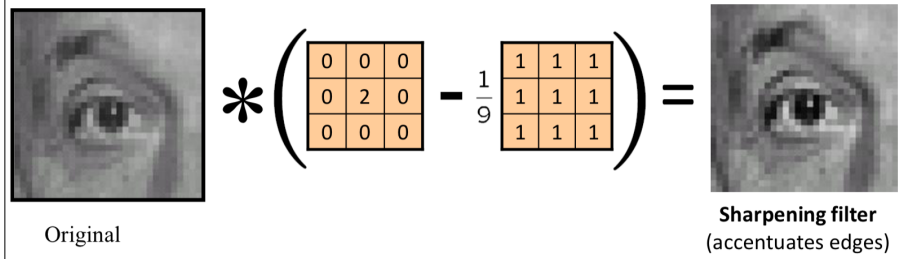
Linear Filter Examples



Original

Blur (with a mean filter)

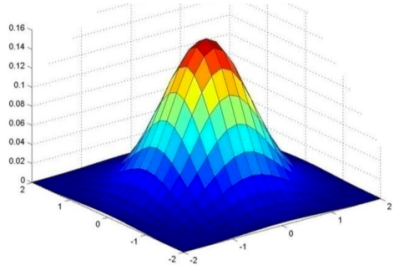
Linear Filter Examples



Original

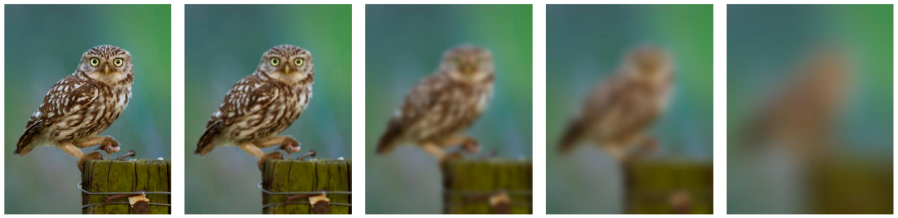
Sharpening filter
(accentuates edges)

Gaussian Kernel



$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

Gaussian Kernel



σ = 1 pixel σ = 5 pixels σ = 10 pixels σ = 30 pixels

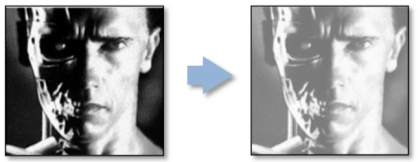
Code it

There are many more filters and techniques to explore, but first, I want you to program a convolution operator and a few of the simple filters.

Code it

Today, we'll just read in an image and store it in a regular matrix in a greyscale format.

Start small, pick an image, make sure you can read it and display it on the canvas. Then try adding a constant value to the image:



$g(x,y) = f(x,y) + 20$

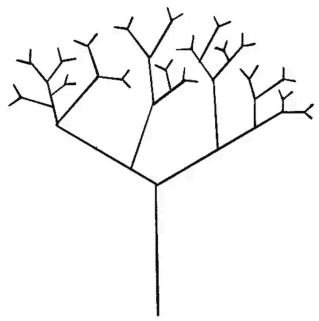
Topics for Today

- Image Processing
- **Mandelbrot Set**

What is a Fractal?

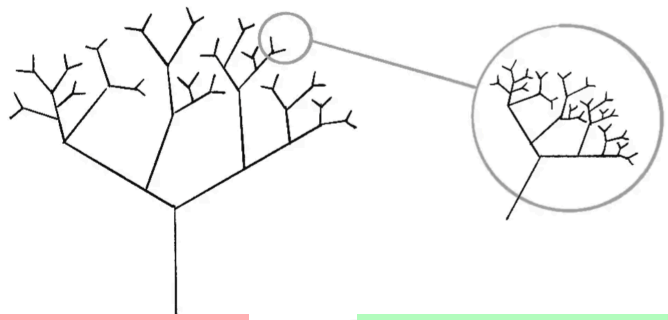
- The term **fractal** comes from the Latin fractus, meaning "broken"
- Coined by Benoit Mandelbrot in 1975 in his work *The Fractal Geometry of Nature*.

"A rough or fragmented geometric shape that can be split into parts, each of with is (at least approximately) a reduced size copy of the whole"



Think about a 2D tree, we have trunk and branches...

What is a Fractal?



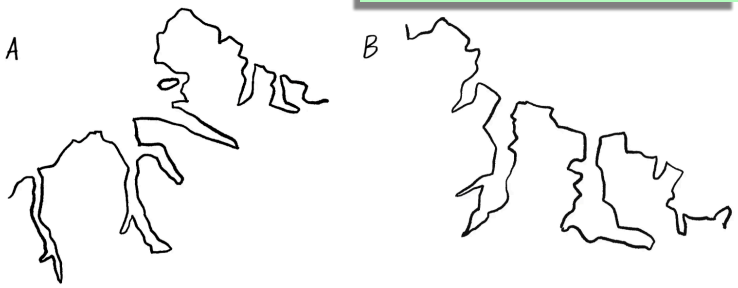
If we pluck one branch from the tree and take a closer look...

...It is an exact replica of the whole, just as Mandelbrot described.

Coastlines

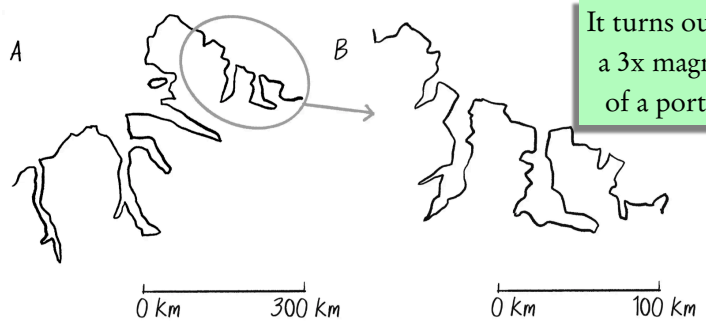
- Not all fractals need to be perfectly self similar:

This shows illustrations of a portion of the coastline of Greenland.



Without a scale for reference, they look very similar. Coastlines exhibit fractal behavior, where they look the same an any scale.

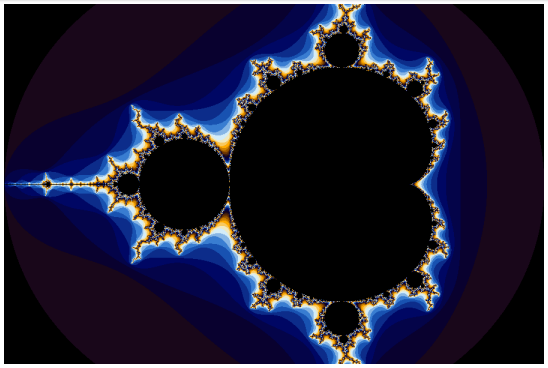
Coastlines



It turns out that B is a 3x magnification of a portion of A

This is a **stochastic** fractal, meaning that it is created from probabilities and randomness. Rather than being exactly self similar, like the tree, it is statistically self similar.

Mandelbrot Set



The most well known fractal is named after Mandelbrot himself: **The Mandelbrot Set**.

It takes a little bit of math to create, but it's the fractal that we'll make today.

It is generated by sending a set of numbers though a simple function and seeing what happens. Does the number go to infinity or does it stay "small".

Mandelbrot Set (The Math)

The Mandelbrot Set is the set of numbers resulting from repeated iterations of the complex function:

$$Z_{n+1} = Z_n^2 + C \quad \text{with } Z_0 = 0$$

$C = x_0 + iy_0$ is part of the Mandelbrot set if $|Z|$ converges

Remember, square of a complex number:

$$Z = x + iy \rightarrow Z^2 = x^2 + i2xy - y^2$$

$$Z^2 = (x^2 - y^2) + i(2xy)$$

Real Part

Imaginary Part

Mandelbrot Set (The Math)

We can separate out the real and imaginary parts:

$$Z^r = x^2 - y^2 + x_0$$

$$Z^i = 2xy + y_0$$

The threshold value is usually:

$$|Z|^2 < 4.0$$

We then set a max number of iterations, N, and iterate until Z falls outside of this threshold, or we reach N.

Generally, we assume that Z will not diverge for higher values of N.

How?

Pick a value for C within your window (usually between -2 and 1 on the real axis and -1 and 1 on the imaginary axis) and test how many iterations before it diverges.

$$f_c(Z) = Z^2 + C$$

Let's try C=1:

$$f_c(0) = 0^2 + 1 = 1$$
$$f_c(1) = 1^2 + 1 = 2$$
$$f_c(2) = 2^2 + 1 = 5$$
$$f_c(5) = 5^2 + 1 = 26$$

Passes our criteria after the third iteration (blows up!)

How?

Let's try C=-1:

$$f_c(0) = 0^2 + -1 = -1$$
$$f_c(-1) = -1^2 + -1 = 0$$
$$f_c(0) = 0^2 + -1 = -1$$
$$f_c(-1) = -1^2 + -1 = 0$$

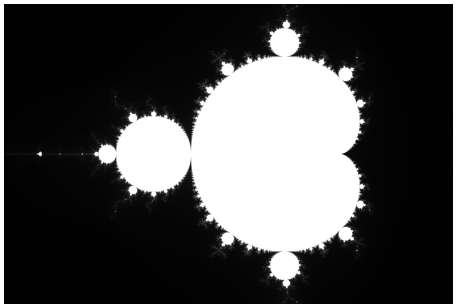
Never diverges, so it will reach our max iterations.

Some values of C will diverge quickly, some slowly and some never at all.

Mandelbrot Set (Plotting)

The Mandelbrot Set is plotted on the complex plane. The x-axis is the real numbers and the y-axis is the complex numbers.

From a practical sense, we'll still just think of it as x and y.



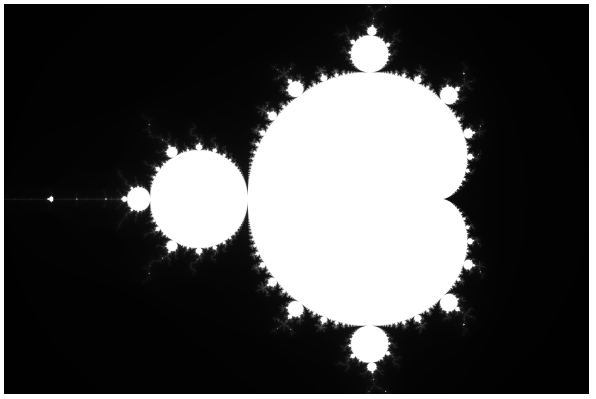
However, our x and y value will come from our choice of C:

$$C = x + iy$$

We will pick many points (many C's) to generate the image.

Mandelbrot Set (The Math)

You can visualize the Mandelbrot set by assigning a color value to the number of iterations it takes to pass that threshold value.



This was made with the code that we will be writing

Complex Equations

Magnitude of a Complex Number:

$$|Z|^2 = x^2 + y^2$$

Real and Imaginary parts: (needed for JavaScript)

$$Z^r = x^2 - y^2 + x_0$$

$$Z^i = 2xy + y_0$$

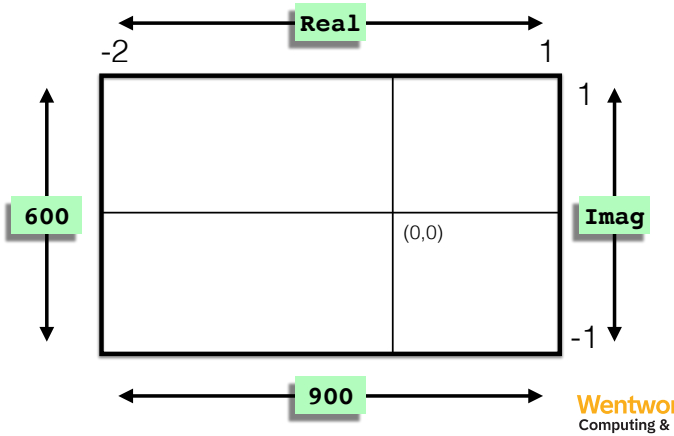
Converting from pixel space to coordinate space (1D):

$$p_{\text{start}} + \left(\frac{i}{n}\right)(p_{\text{end}} - p_{\text{start}})$$

Wentworth
Computing & Data Science

Pixel Space to Coordinate Space?

Our image (matrix) is in pixels, how do we relate that to an actual (x,y) value on the complex plane?



Wentworth
Computing & Data Science

Pixel Space to Coordinate Space?

Let's say we pick a pixel, (i,j), what point in our image does that represent in our complex plane, (x,y)?

$$x = \text{xStart} + \left(\frac{i}{\text{xNum}}\right)(\text{xEnd} - \text{xStart})$$

$$y = \text{yStart} + \left(\frac{j}{\text{yNum}}\right)(\text{yEnd} - \text{yStart})$$

Beginning of the axis in complex space

(i,j) is the pixel, yNum and xNum is the size in pixels

Range in complex space

Wentworth
Computing & Data Science

Pixel Space to Coordinate Space?

In our specific case:

$$x = \text{xStart} + \left(\frac{i}{\text{xNum}}\right)(\text{xEnd} - \text{xStart})$$

$$y = \text{yStart} + \left(\frac{j}{\text{yNum}}\right)(\text{yEnd} - \text{yStart})$$

$$x = -2 + 3\left(\frac{i}{900}\right)$$

$$y = -1 + 2\left(\frac{j}{600}\right)$$

Don't do an integer division here!

Wentworth
Computing & Data Science

Complex Equations

Magnitude of a Complex Number:

$$|Z|^2 = x^2 + y^2$$

Real and Imaginary parts: (needed for Java)

$$Z^r = x^2 - y^2 + x_0$$

$$Z^i = 2xy + y_0$$

Converting from pixel space to coordinate space (1D):

$$p_{\text{start}} + \left(\frac{i}{n}\right)(p_{\text{end}} - p_{\text{start}})$$

Pseudocode

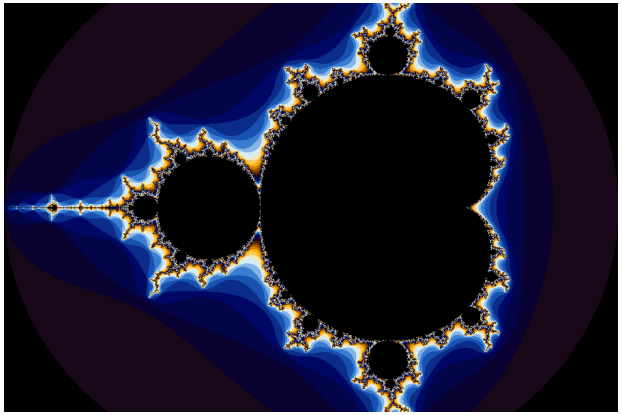
Creating the Mandelbrot set is fairly simple.

```
Decide image resolution (nx,ny) and max iterations
Decide range of the Re-Im plane to view
       (generally between Re=-2..1 and Im=-1..1)

Loop over nx and ny
    Convert nx and ny to points in the Re-Im
    Set C to the point in the Re-Im plane

    loop until max iterations reached
        Z=Z^2+C
        if(|Z|>4)
            stop, record iterations

    store iter in matrix for image plot (opt
```

Code it up!
Useful params:
maxiter = 255
range: Re = -2..1
Im = -1..1

Writing to Image

I've created a color map function that you can use. It uses a 16 color gradient to assign colors to ranges of iterations.



Extra

- **Showing Worlds**
- 2D Ray Casting

Showing Worlds

- In most games, the world is much bigger than we can display in a single screen
- So, we show only a subsection of the game world at any given time
- The part of the world that we show is determined by the camera

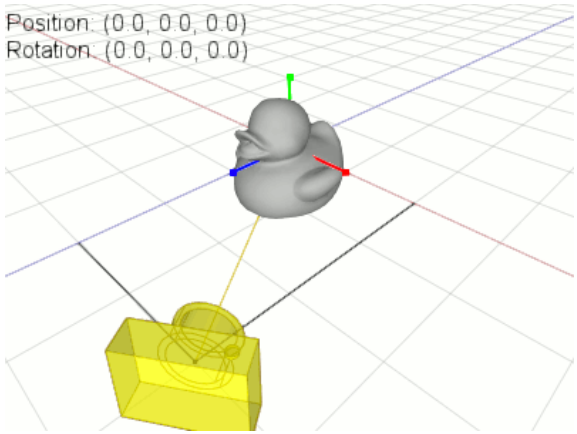
While not an actual camera, how we see the game world can be described by physical properties like zoom, rotation, position, etc.

The Camera

- Using the in-game camera, we can mathematically determine what should be shown to the player
- Intuitively, we define a location in the world for the camera and a target to point at
- Analogous to real-world photography, this determines the final view that players will see in our game

There are many different types of camera systems, while 2D cameras tend to be easier to use, they have many of their own quirks

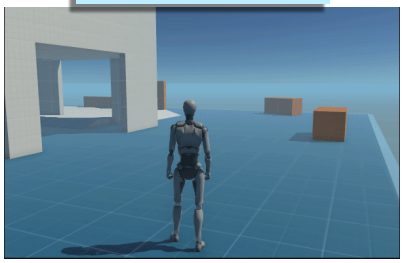
The Camera



https://www.songho.ca/opengl/gl_camera.html

Some 3D Camera Styles

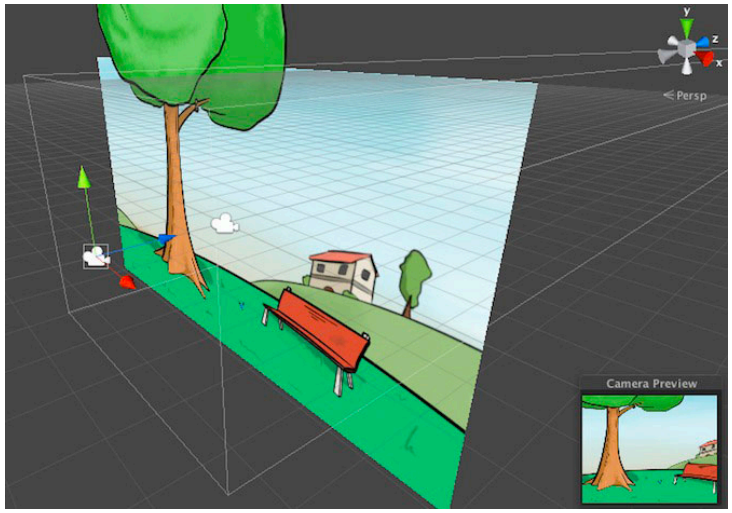
Third person, over the shoulder style



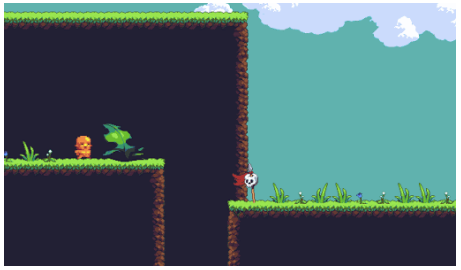
First person

Fixed Position

2D Camera



2D Camera



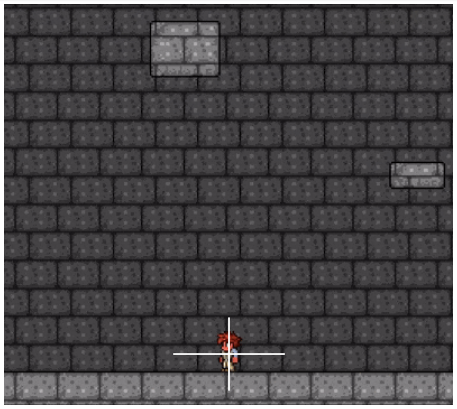
There are many different types of 2D cameras depending on your needs

2D Game Cameras:

https://www.youtube.com/watch?v=l9G6MNhfV7M

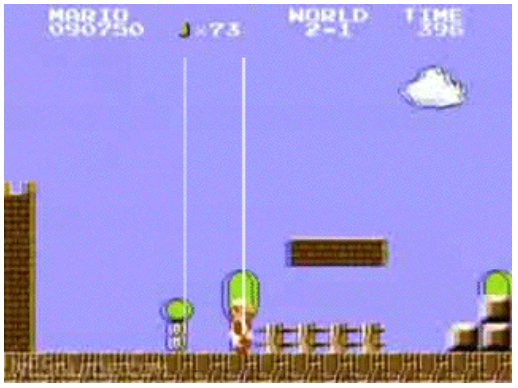
2D Camera Styles: Locked

- Camera is always centered exactly on the player position.
- Best when the game is zoomed out
  - Otherwise can cause motion sickness!
- Not great is most situations



Can be done in both axes or just one

2D Camera Styles: Zones

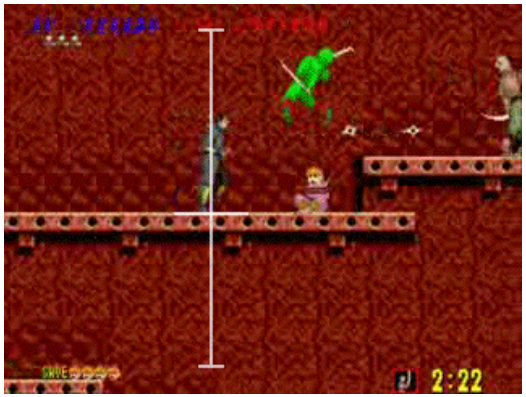


- Define a zone for the camera motion
- Choose to stop or speed up motion based on player's position within the zone.

Probably the best choice for platformers

2D Camera Styles: Zones

2D Camera Styles: Position Snap

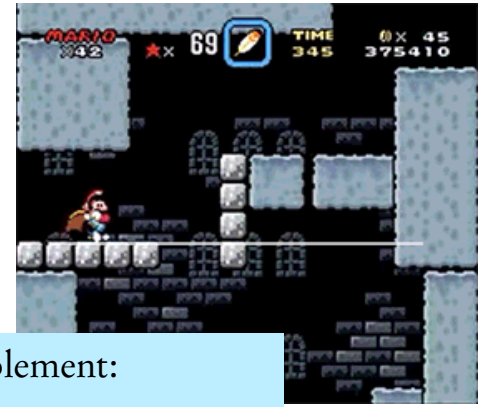


- Lock to one axis
- Change the position of the other axis if the player changes that axis (like hopping on a new platform)
- Gives nice motion overall

Notice: The camera has smooth motion in the y and locked in the x

2D Camera Styles: Platform Snap

- Similar to position snapping
- Based on the level layout
- Only snaps when the player is "grounded"



Easy to implement:

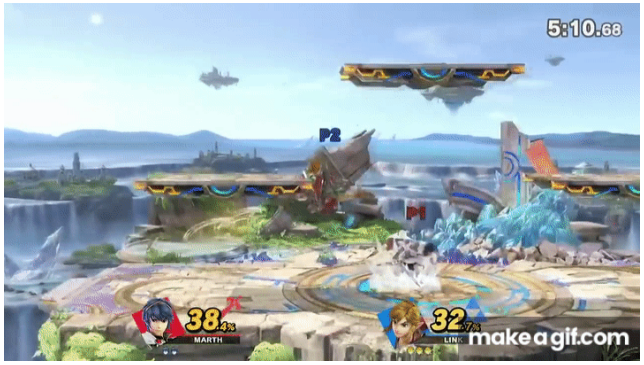
Just lock the camera to the top of the collider that Mario just landed on

2D Camera Styles: Trap

- Define a box around the player
- Camera only moves if the player "pushes" the edge of the box

Great for game with a lot of jerky movement

2D Camera Styles: Zooming



- In a beat'em up, you may want the camera to zoom tight when the players are close together and zoom out when they are far away

Extra

- Showing Worlds
- **2D Ray Casting**

What is Ray Casting

- Technique which has many applications
  - First game to use Ray Casting was released in 1985
- Popular example: Wolfenstein 3D

Ray Casting is not Ray Tracing

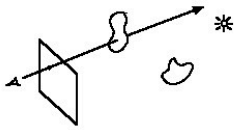
Wolfenstein 3D style tutorial:

https://lodev.org/cgtutor/raycasting.html

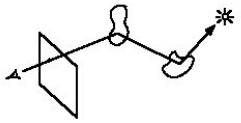
Ray Tracings vs Ray Casting

Casting versus tracing

1st path = ray casting



recursive = ray tracing



https://cs.stanford.edu/people/eroberts/courses/soco/projects/1997-98/ray-tracing/alternatives.html

We can think of ray tracing as many ray casts that represent bouncing light

Ray Casting Idea

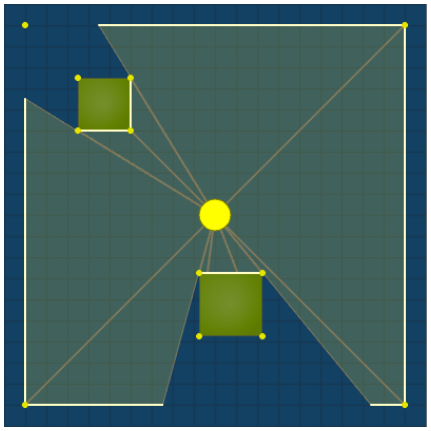
- Draw a line between two points
- Detect if and where a collision occurs

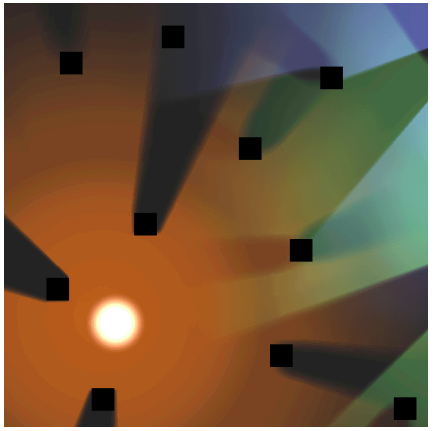
Ray Tracing: How did it bounce?

If there is no collision between the two points, they can "see" each other

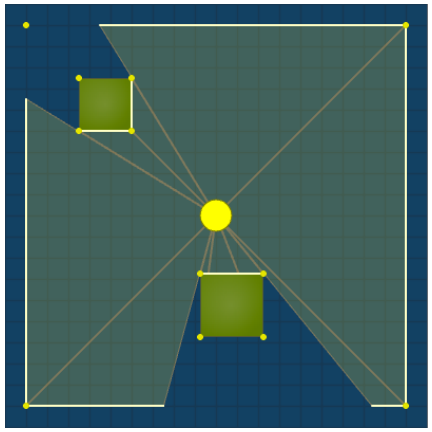
For us, 2D Ray Casting is useful for solving visibility/light problems in our game

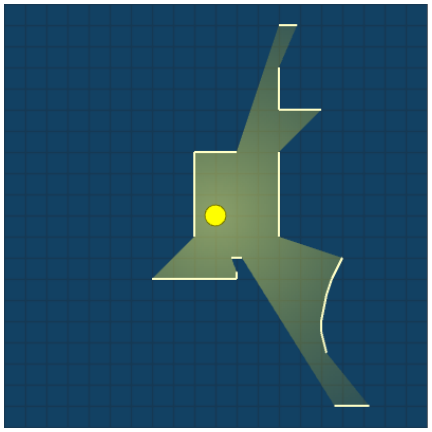
2D Ray Casting: Lighting





2D Ray Casting: Visibility





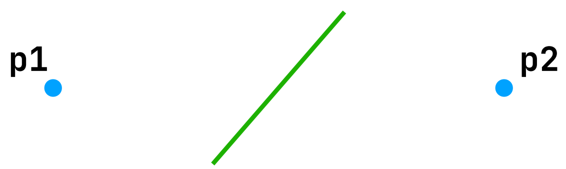
https://www.redblobgames.com/articles/visibility/

2D Ray Casting: Visibility

Monaco: What's Yours Is Mine - Pocketwatch Games

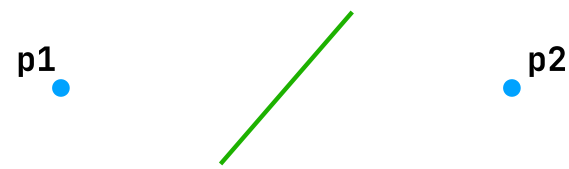
Basics

- A ray cast asks a simple question:
  - Is there any obstacle blocking the path between two points?
  - How do we detect this?



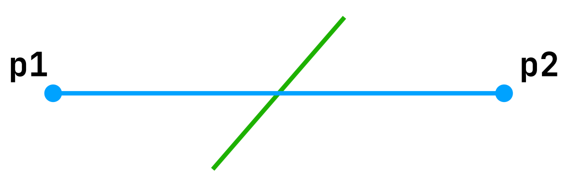
Basics

- The path between two points is blocked if the line segment between them intersects with another line segment:



Basics

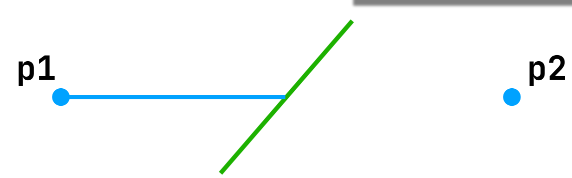
- The path between two points is blocked if the line segment between them intersects with another line segment:



Basics

- The path between two points is blocked if the line segment between them intersects with another line segment:

We may just want to know the first intersection or all intersections between the two points.



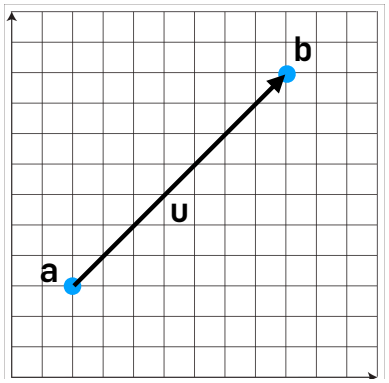
Line Segment Intersection

We usually have a **`Vec2`** class that we can use to represent points in 2D space

A line segment is defined by two **`Vec2`** objects, representing the start and end points

```
Vec2 p1 = Vec2(x1,y1);
Vec2 p2 = Vec2(x2,y2);
```

Line Segments



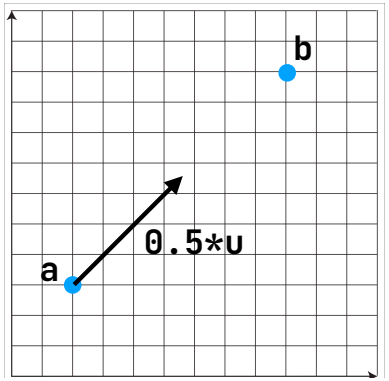
- Our points are stored as **`Vec2`** objects
- Call the vector between the points **`u`**
  - Computed by subtraction

$$\mathbf{u} = \mathbf{b} - \mathbf{a}$$

or

$$\mathbf{b} = \mathbf{a} - \mathbf{u}$$

Line Segments



- Our points are stored as **`Vec2`** objects
- Call the vector between the points **`u`**
  - Computed by subtraction

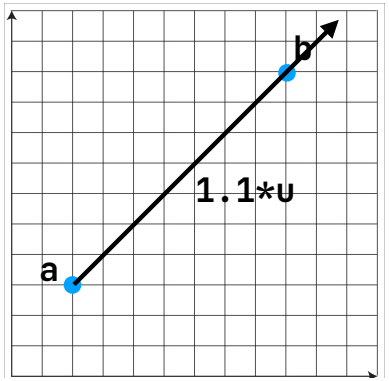
$$\mathbf{u} = \mathbf{b} - \mathbf{a}$$

or

$$\mathbf{b} = \mathbf{a} - \mathbf{u}$$

We can scale the result vector by a scalar

Line Segments



- Our points are stored as **`Vec2`** objects
- Call the vector between the points **`u`**
  - Computed by subtraction

$$\mathbf{u} = \mathbf{b} - \mathbf{a}$$

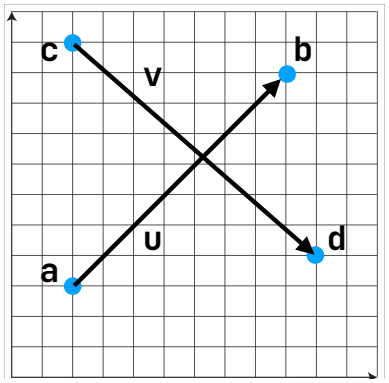
or

$$\mathbf{b} = \mathbf{a} - \mathbf{u}$$

We can scale the result vector by a scalar

$$\mathbf{b} = \mathbf{a} + t \cdot \mathbf{u}$$

Line Segments

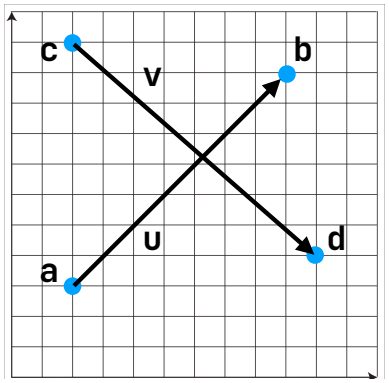


- Our goal is to determine when two lines intersect

$$\mathbf{v} = \mathbf{d} - \mathbf{c}$$
$$\mathbf{u} = \mathbf{b} - \mathbf{a}$$

- When `u` and `v` are the "correct" length, they end at the same point (where they intersect)

Line Segments

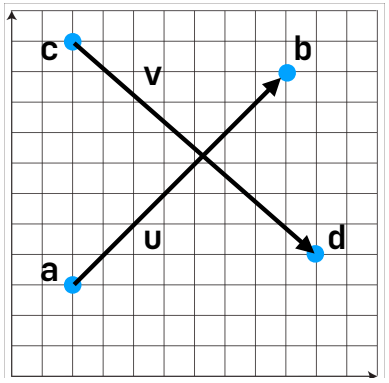


- Use the scaled version of each:

$$\mathbf{b} = \mathbf{a} + t \cdot \mathbf{u}$$
$$\mathbf{d} = \mathbf{c} + s \cdot \mathbf{v}$$

Remember, `a+t*u` is just a vector going part way to `b` (if `t<1`)

Line Segments

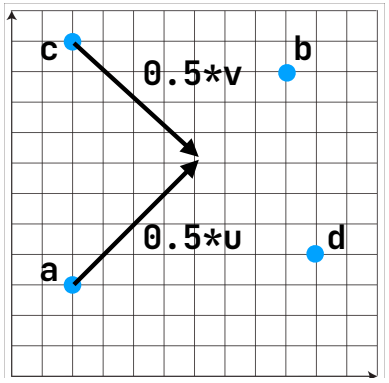


- Use the scaled version of each:

$$\mathbf{b} = \mathbf{a} + t \cdot \mathbf{u}$$
$$\mathbf{d} = \mathbf{c} + s \cdot \mathbf{v}$$

- What if `t=0.5` and `s=0.5`?

Line Segments

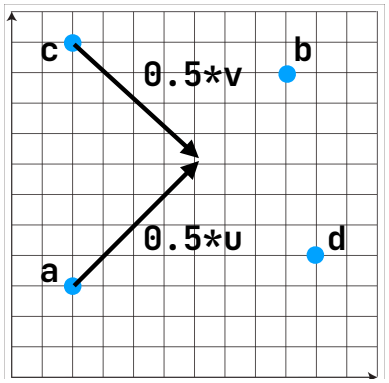


- Use the scaled version of each:

$$\mathbf{b} = \mathbf{a} + t \cdot \mathbf{u}$$
$$\mathbf{d} = \mathbf{c} + s \cdot \mathbf{v}$$

- What if `t=0.5` and `s=0.5`?

Line Segments



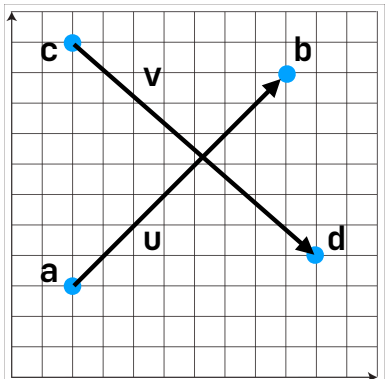
- Use the scaled version of each:

$$\mathbf{b} = \mathbf{a} + t \cdot \mathbf{u}$$
$$\mathbf{d} = \mathbf{c} + s \cdot \mathbf{v}$$

- What if `t=0.5` and `s=0.5`?
- They point to the same place, their intersection

This is our goal: find the values of **t** and **s** so that they point to the same location

Line Segment Intersection



- Start with:

$$\mathbf{b} = \mathbf{a} + t \cdot \mathbf{u}$$
$$\mathbf{d} = \mathbf{c} + s \cdot \mathbf{v}$$

- Set them equal:

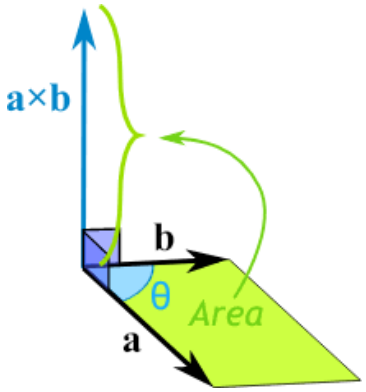
$$\mathbf{a} + t \cdot \mathbf{u} = \mathbf{c} + s \cdot \mathbf{v}$$

- There is an intersection when:

$$t, s \in [0, 1]$$

But, we have one equation and two unknowns..

Cross Product

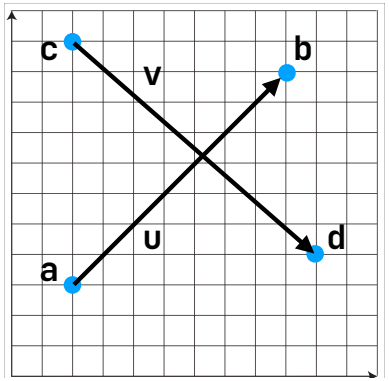


- The standard 3D Cross Product creates a vector that is perpendicular to two other vectors
- This is called the **normal vector**
- Only really makes sense in 3D

Normal vectors are perpendicular to a plane.

Normal vectors are often used in computer graphics (e.g. dynamic mesh generation) and in 3D path finding.

2D Cross Product



- The 2D cross product returns a scalar, not a vector
- Basic calculation:

$$u \times v = u_x v_y - u_y v_x$$

- Properties:

$$u \times u = 0$$
$$(a + b) \times c = a \times c + b \times c$$
$$a \times b = -(b \times a)$$

Solving for t

- Start with:

$$\mathbf{a} + t \cdot \mathbf{u} = \mathbf{c} + s \cdot \mathbf{v}$$

- Apply 2D cross product with **v** to both sides:

$$(\mathbf{a} + t \cdot \mathbf{u}) \times \mathbf{v} = (\mathbf{c} + s \cdot \mathbf{v}) \times \mathbf{v}$$

$$\mathbf{a} \times \mathbf{v} + t \cdot \mathbf{u} \times \mathbf{v} = \mathbf{c} \times \mathbf{v} + s \cdot \mathbf{v} \times \mathbf{v}$$

$$\mathbf{a} \times \mathbf{v} + t \cdot \mathbf{u} \times \mathbf{v} = \mathbf{c} \times \mathbf{v}$$

- Now we can solve for **t**:

$$\boxed{t = \frac{(\mathbf{c} - \mathbf{a}) \times \mathbf{v}}{\mathbf{u} \times \mathbf{v}}}$$

Solve for Intersection

- We can solve for **s** in the same way, in the end we have:

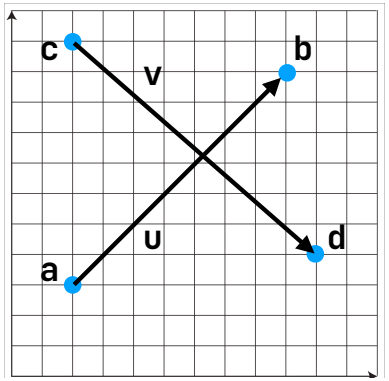
$$t = \frac{(\mathbf{c} - \mathbf{a}) \times \mathbf{v}}{\mathbf{u} \times \mathbf{v}} \qquad s = \frac{(\mathbf{a} - \mathbf{c}) \times \mathbf{u}}{\mathbf{v} \times \mathbf{u}}$$

- We can optimize this a little bit by using the properties of the 2D cross product:

$$\boxed{t = \frac{(\mathbf{c} - \mathbf{a}) \times \mathbf{v}}{\mathbf{u} \times \mathbf{v}}} \qquad \boxed{s = \frac{(\mathbf{c} - \mathbf{a}) \times \mathbf{u}}{\mathbf{u} \times \mathbf{v}}}$$

Compute **c-a** and **uxv** only once, use twice

Finally



$$\boxed{s = \frac{(\mathbf{c} - \mathbf{a}) \times \mathbf{u}}{\mathbf{u} \times \mathbf{v}}} \qquad \boxed{t = \frac{(\mathbf{c} - \mathbf{a}) \times \mathbf{v}}{\mathbf{u} \times \mathbf{v}}}$$

- Intersection if:

$$t, s \in [0, 1]$$

- Intersection point:

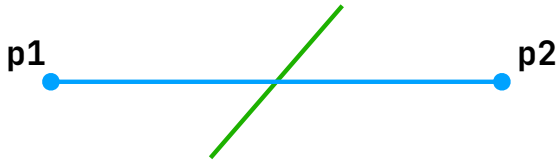
$$\mathbf{p} = \mathbf{a} + t * \mathbf{u}$$

or

$$\mathbf{p} = \mathbf{c} + s * \mathbf{v}$$

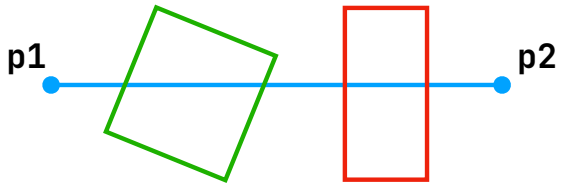
Visibility

- OK, we can compute line intersections, so how do we actually use it for visibility?
- The concept is simple:
  - If the line between two points intersects another line, they are not visible to each other:



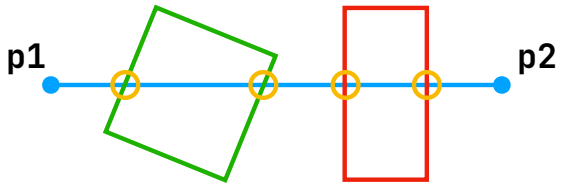
Visibility

- OK, we can compute line intersections, so how do we actually use it for visibility?
- The concept is simple:
  - If the line between two points intersects another line, they are not visible to each other:



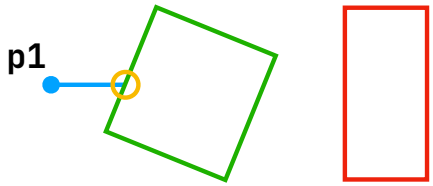
Visibility

- OK, we can compute line intersections, so how do we actually use it for visibility?
- The concept is simple:
  - If the line between two points intersects another line, they are not visible to each other:



Visibility

- OK, we can compute line intersections, so how do we actually use it for visibility?
- The concept is simple:
  - If the line between two points intersects another line, they are not visible to each other:



Usually, we are only interested in the first collision, closest to p1

Lighting Effects

- How do we calculate everywhere light hits?
- Links:
  - https://www.redblobgames.com/articles/visibility/
  - https://ncase.me/sight-and-light/