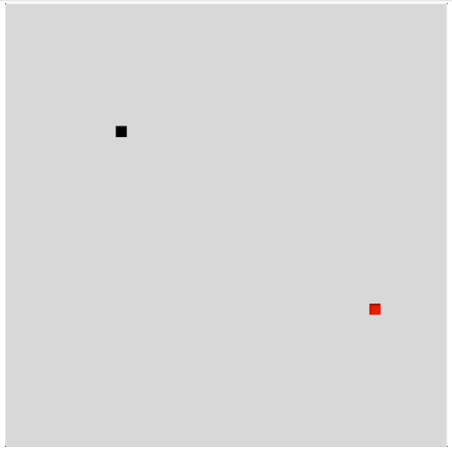
Impact Lab 2025: Programming Fundamentals
Lecture 8: Snake Game

Summer 2025

School of Computing and Data Science

Wentworth Institute of Technology

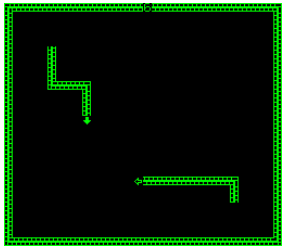


History

- Snake typically refers to a genre of action video games where the player maneuvers the end of a growing line.
- The "snake" can't collide with other obstacles or itself.
- As the game proceeds, the snake typically gets longer and moves faster.

The original game was *Blockade*, from 1976.

It was a multiplayer game where the goal is to survive longer than the other player.



Topics for Today

- **Setup**
- Head of the Snake
- Eating Apples
- Growing the Body
- Death

The lecture today has a lot of code, essentially everything to make the game.

I'll be talking through all of it.

Setup

```
let snake=[];
let food;
let res=10;

function setup() {
  frameRate(5);
  createCanvas(400, 400);
}
```

We will keep the frame rate low for now, just know that it will control the speed of the game.

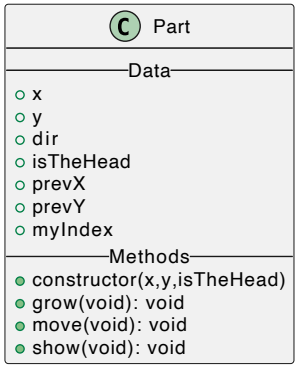
- **`snake`**: An array that will represent the entire snake.
- **`food`**: The single food that will be on the board, it will move once the snake eats it.
- **`res`**: Resolution of the game (the snake will be **`res`** by **`res`** pixels).

Topics for Today

- Setup
- **Head of the Snake**
- Eating Apples
- Growing the Body
- Death

`Part` Class

- The **`Part`** class represents any of the individual parts of the snake (including the head).
- It has the standard data we need to draw the part.
- It also contains some extra information about that we will need to move and grow.



We'll leave **`grow()`** for later, but the other methods are easy to implement.

`Part` Constructor

```
constructor(x,y,isTheHead){
    this.x=x;
    this.y=y;
    this.dir=3;
    this.isTheHead=isTheHead;
    this.prevX=x;
    this.prevY=y;
    this.myIndex=snake.length;
  }
```

Standard constructor that sets all the data

The intention here (illustrated by **`myIndex`**) is that each part will be added to the end of the snake array that we created a few slides ago.

`Part show()`

```
show(){
  fill(0);
  noStroke();
  rect(this.x,this.y,res,res);
}
```

Standard show() function that we've seen before.

Our snake will be a black square with side lengths **`res`**, that we defined at the start.

Most of the methods will be quite small.

This is a common feature of OOP programs that helps us reuse our code and increases maintainability and extensibility.

Part `move()` (Initial Version)

```
move(){
  this.prevX=this.x;
  this.prevY=this.y;
  if(this.isHead==true){
    if(this.dir==1){
      this.y-=res;
    }
    if(this.dir==2){
      this.x+=res;
    }
   if(this.dir==3){
      this.y+=res;
    }
   if(this.dir==4){
      this.x-=res;
    }
  }
}
```

Before we move, save our current position into the previous position variables.

We only move based on user input if we are the head part.

Move **x** or **y** depending on direction (which is updated by the user).

Part `move()` (Initial Version)

```
move(){
  this.prevX=this.x;
  this.prevY=this.y;
  if(this.isHead==true){
    if(this.dir==1){
      this.y-=res;
    }
    if(this.dir==2){
      this.x+=res;
    }
   if(this.dir==3){
      this.y+=res;
    }
   if(this.dir==4){
      this.x-=res;
    }
  }
}
```

Before we move, save our current position into the previous position variables.

We only move based on user input if we are the head part.

depending on
hich is updated
e user).

There will be another part to this if we are *not* the head, but we can save that for later.

Back to `setup()` and `draw()`

```
snake[0]=new Part(100,100,true);
```

Add this to setup, it creates our **Head** piece as the first element of the array (index 0)

We'll keep adding to **`setup()`** and **`draw()`** as we go forward.

```
function draw() {
  background(220);
  for(let i=0;i<snake.length;i++){
    snake[i].move();
    snake[i].show();
  }
}
```

Loop over every snake piece, **`move()`** and **`show()`**.

`keyPressed()` Special Function (Truncated)

```
function keyPressed(){
  if(keyCode==UP_ARROW){
    if(snake[0].dir!=3){
      snake[0].dir=1
    }
  }
  if(keyCode==RIGHT_ARROW){
    if(snake[0].dir!=4){
      snake[0].dir=2;
    }
  }
  …
}
```

Use **`keyCode`** to check which key was pressed.

Why do we have the if statements before setting the direction?

The directions are:

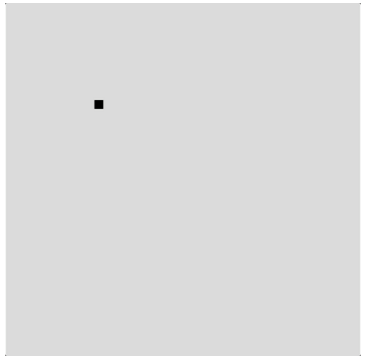
- Up: 1
- Right: 2
- Down: 3
- Left: 4

You can add other key presses too, like one to restart or to test adding parts.

So Far

Our snake moves based on user input (arrow keys).

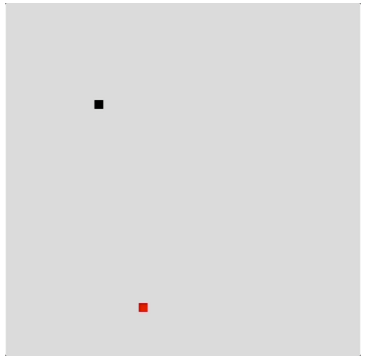
We'll add a game over check later, but for now we can move off the edge of the canvas (and back on if we turn around).



The **`if`** statements in the **`keyPressed()`** function prevents us from moving in the opposite direction that we are currently moving.

Topics for Today

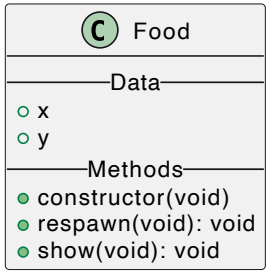
- Setup
- Head of the Snake
- **Eating Apples**
- Growing the Body
- Death



`Food` Class

```
constructor(){
  this.x=0;
  this.y=0;
  this.respawn();
}

show(){
  fill(255,0,0);
  noStroke();
  rect(this.x,this.y,res,res);
}
```



Very simple class

`respawn()` Method

```
respawn(){
    this.x=int(random(0,width/res))*res;
    this.y=int(random(0,height/res))*res;
}
```

Randomly places the "Apple" on the canvas.

In our current code, we generate a random number between 0 and 40, then multiply it by 10.

So, the food will only appear on pixels that are a multiple of 10.

But, we don't want it just anywhere. It should spawn so that it appears to be on a grid, even though there is no actual grid.

Updating `setup()` and `draw()`

```
food=new Food();
```

Add to **`setup()`**

```
food.show();
```

Add to **`draw()`** before the snake loop

```
if(snake[0].x==food.x && snake[0].y==food.y){
  snake[0].grow();
  food.respawn();
}
```

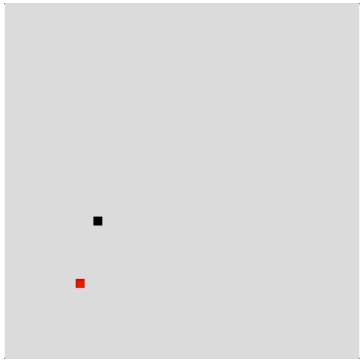
In **`draw()`**, check if the snake "eats" the food.

We haven't implemented **`grow()`** yet, but we'll get there.

We just have to check if the head of the snake is in the exact same place as the food (both **`x`** and **`y`**)

Topics for Today

- Setup
- Head of the Snake
- Eating Apples
- **Growing the Body**
- Death



Growing the Body

In the **`Snake`** class

```
grow(){
  snake.push(new Part(-100,-100,false));
}
```

Notice that we put in a negative position value, so it is not seen on the canvas.

When we grow the snake, we just add a new part to the end of the snake array.

In the **`Snake`** class **`move()`** function

```
else{
  this.x=snake[this.myIndex-1].prevX;
  this.y=snake[this.myIndex-1].prevY;
}
```


Growing the Body

In the **`Snake`** class

```
grow(){
  snake.push(new Part(-100,-100,false));
}
```

Notice th… position v…

The body part will move to the previous position of the part "above" in the array!

When we grow the snake, we just add a new part to the end of the snake array.

…class **`move()`** function

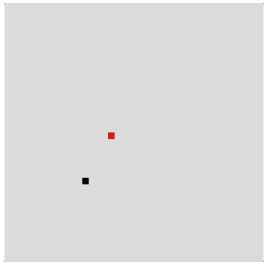
```
else{
  this.x=snake[this.myIndex-1].prevX;
  this.y=snake[this.myIndex-1].prevY;
}
```

Growing the Body

```
if(snake[0].x==food.x && snake[0].y==food.y){
  snake[0].grow();
  food.respawn();
}
```

Update the food check in **draw()**.

Make sure to call **grow()**, on the head piece (index 0) when we "eat" the food.

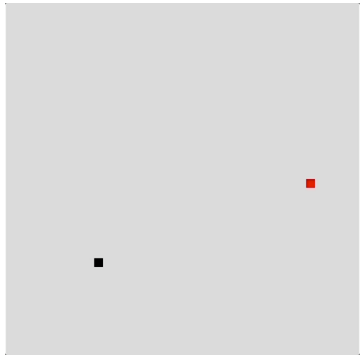


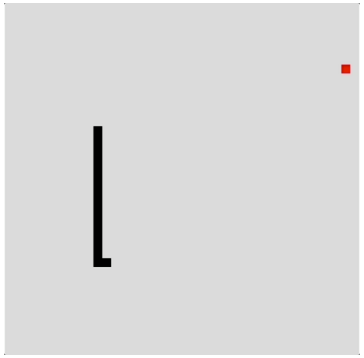
Topics for Today

- Setup
- Head of the Snake
- Eating Apples
- Growing the Body
- **Death**

Death

Death consists of two parts, hitting the boundary or hitting your tail.





Death

In **draw()**:

```
if(snake[0].x<0 || snake[0].x>width
                ||snake[0].y<0 || snake[0].y>height){
   gameOver();
 }
```

If the head of the snake leaves the canvas call **gameOver()**

```
for(let i=1;i<snake.length;i++){
   if(snake[0].x==snake[i].x && snake[0].y==snake[i].y){
     gameOver();
   }
 }
```

Check all the parts (except the head) if the head touches them.

Death

In **draw()**:

```
if(snake[0].x<0 || snake[0].x>width
                ||snake[0].y<0 || snake[0].y>height){
   gameOver();
 }
```

If the head of the snake leaves the canvas call **gameOver()**

```
for(let i=1;i<snake.length;i++){
   if(snake[0].x==snake[i].x && snake[0].y==snake[i].y){
     gameOver();
```

This is similar to checking if we "eat" the apple.

Check all the parts (except the head) if the head touches them.

gameOver()

```
function gameOver(){
  background(255,0,0,100);
  print("Game Over!");
  noLoop();
}
```

Change the background to red, with a little bit of transparency.

You can do anything you want here, like play a sound, show a menu, or just restart the game immediately.

We could also include a use **keyPressed()** to reset when we are in a game over state.

Print to the console that the game is over.

Turn off the **draw()** loop, so that it stops running.

Extra

- **2D Ray Casting**
- Collision Detection

What is Ray Casting

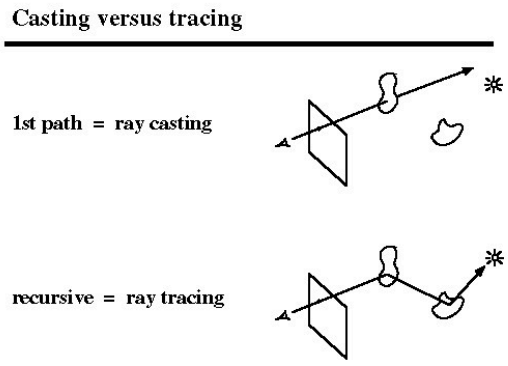
- Technique which has many applications
  - First game to use Ray Casting was released in 1985
- Popular example: Wolfenstein 3D

Ray Casting is not Ray Tracing

Wolfenstein 3D style tutorial:

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

Ray Tracings vs Ray Casting



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

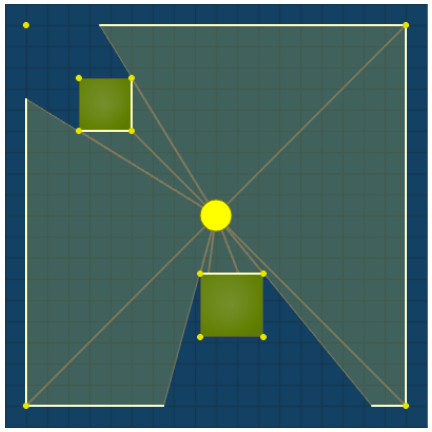
- Draw a line between two points
- Detect if and where a collision occurs

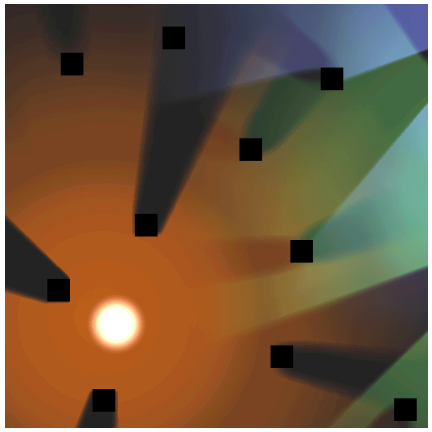
Ray Tracing: How did it bounce?

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

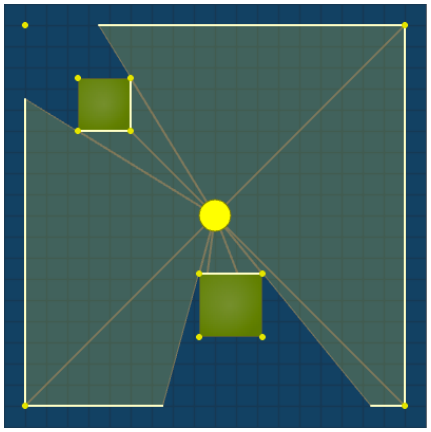
For us, 2D Ray Casting is useful for solving visibility/light problems in our game

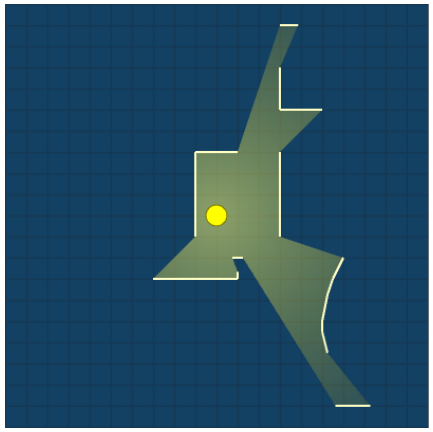
2D Ray Casting: Lighting





2D Ray Casting: Visibility





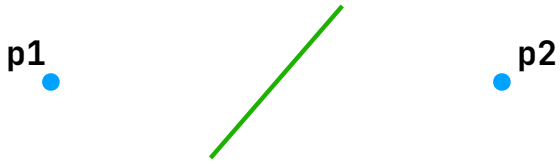
https://www.redblobgames.com/articles/visibility/

2D Ray Casting: Visibility

Monaco: What's Yours Is Mine - Pocketwatch Games

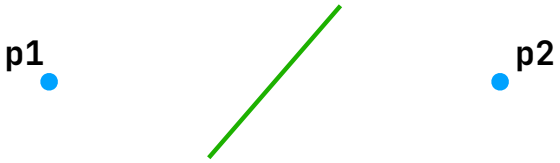
Basics

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



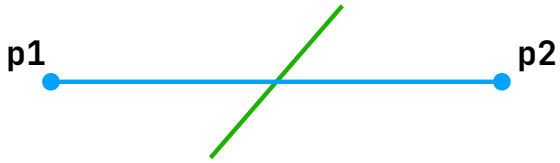
Basics

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



Basics

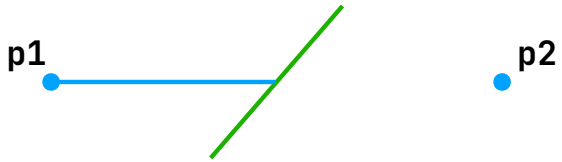
- The path between two points is blocked if the line segment between them intersects with another line segment:



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.



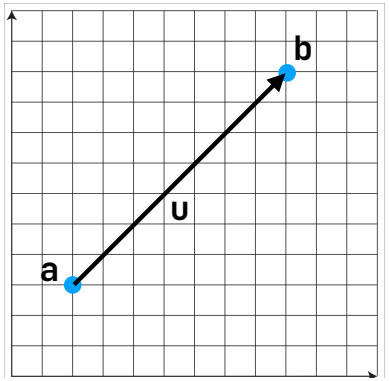
Line Segment Intersection

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



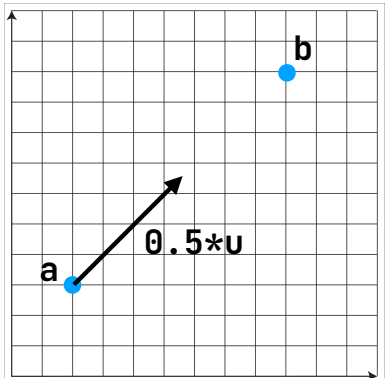
- Our points are stored as **`Vec2`** objects
- Call the vector between the points **`u`**
  - Computed by subtraction

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

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

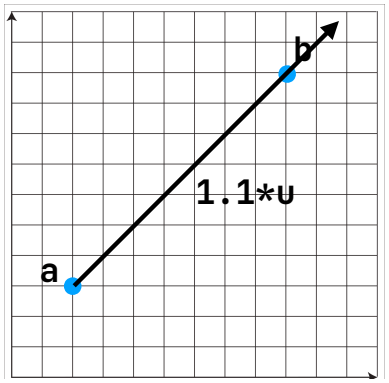
$$\mathbf{u} = \mathbf{b} - \mathbf{a}$$

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

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

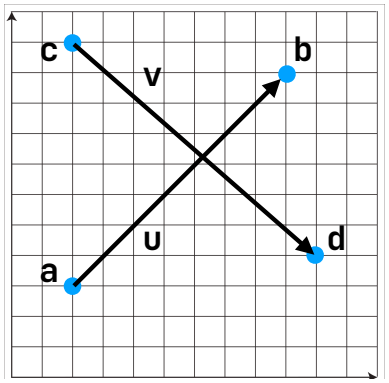
or

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

We can scale the result vector by a scalar

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

Line Segments

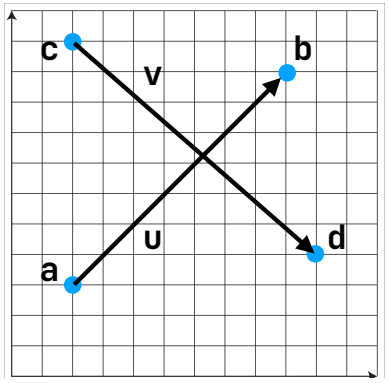


- Our goal is to determine when two lines intersect

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

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

Line Segments

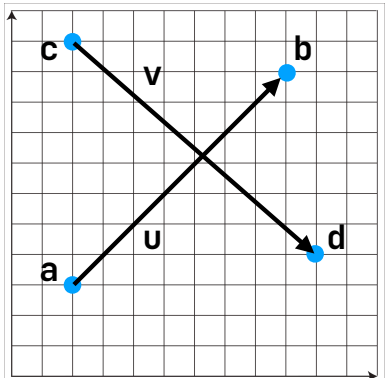


- Use the scaled version of each:

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

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

Line Segments

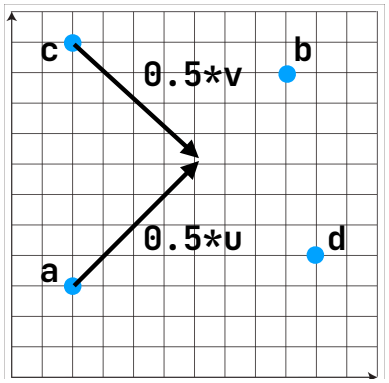


- Use the scaled version of each:

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

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

Line Segments

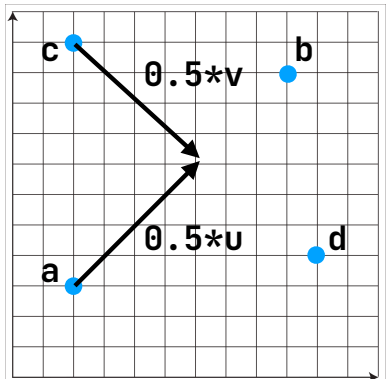


- Use the scaled version of each:

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

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

Line Segments



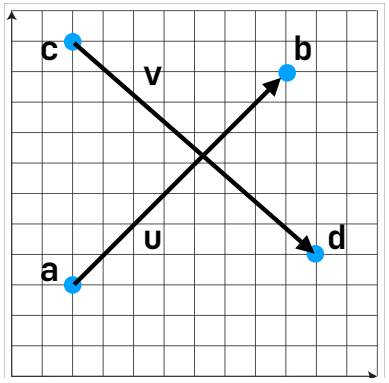
- Use the scaled version of each:

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

- 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:

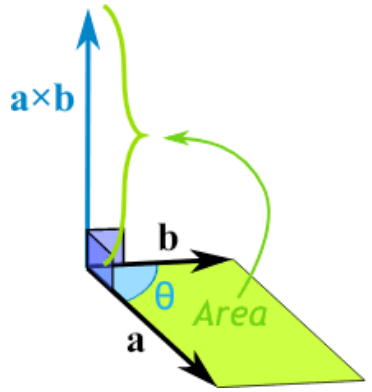
$$\mathbf{a} + t \cdot \mathbf{u} = \mathbf{c} + s \cdot \mathbf{v}$$

- There is an intersection when:

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

But, we have one equation and two unknowns..

Cross Product

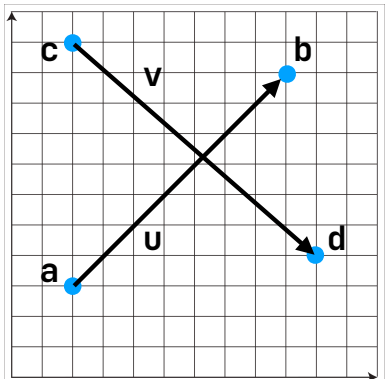


- The standard 3D Cross Product creates a vector that is perpendicular to two other vectors
- This is called the **normal vector**
- Only really makes sense in 3D

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:

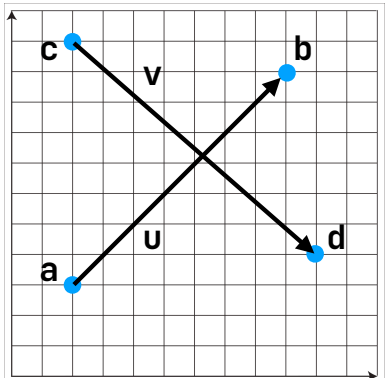
$$t = \frac{(\mathbf{c} - \mathbf{a}) \times \mathbf{v}}{\mathbf{u} \times \mathbf{v}} \qquad s = \frac{(\mathbf{a} - \mathbf{c}) \times \mathbf{u}}{\mathbf{v} \times \mathbf{u}}$$

- 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:

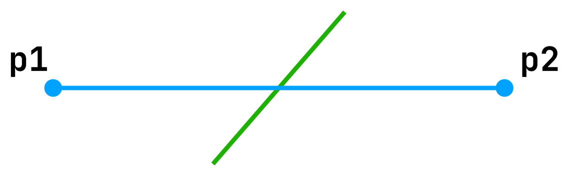
$$\mathbf{p} = \mathbf{a} + t * \mathbf{u}$$

or

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

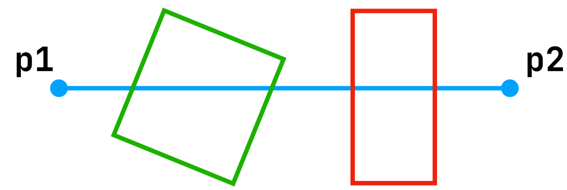
Visibility

- 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:



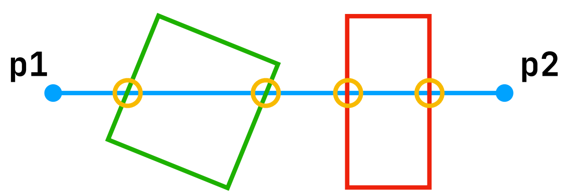
Visibility

- 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:



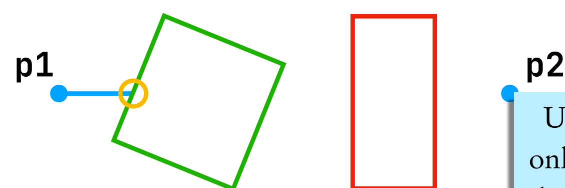
Visibility

- 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/

Extra

- 2D Ray Casting
- **Collision Detection**

Collision Detection (From my Game Engine Class)

- Collisions involve two stages:
  - Detection
    - When there are two objects that can collide, determine if they intersect
    - Geometric problem
  - Resolution
    - If two entities have collided, determine how to resolve the physics

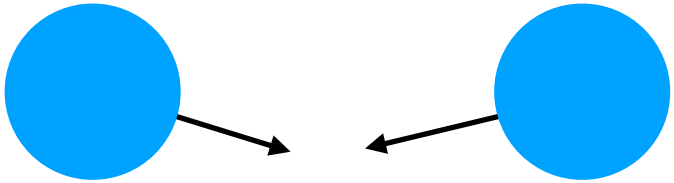
Assumptions

- We will use **`Entity`** to represent any object in the game
- Entities have the following components:
  - Position (x and y)
  - Bounding Shape (circle or rectangle)
  - Velocity (dx and dy)
- Our goal is to learn the math that allows entities to interact

If this wasn't a class on Game Engine Development, you would likely use **Box2D** for your 2D physics.

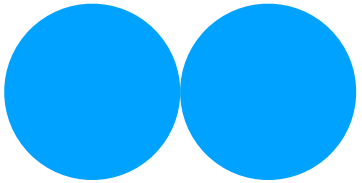
The Collision Problem

- Given two entities, which have a velocity, position, and bounding shape, determine if, when, and where they collide:



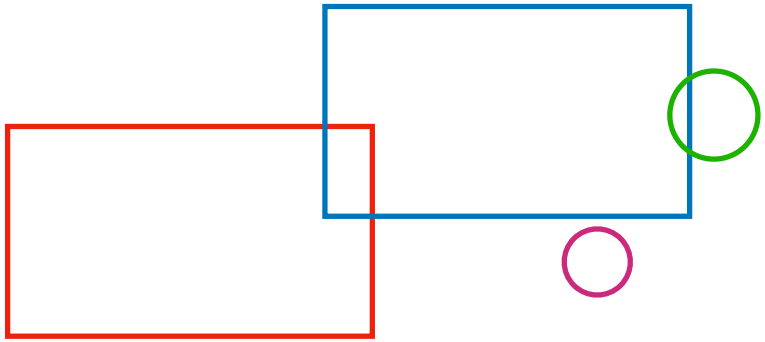
The Collision Problem

- Given two entities, which have a velocity, position, and bounding shape, determine if, when, and where they collide:



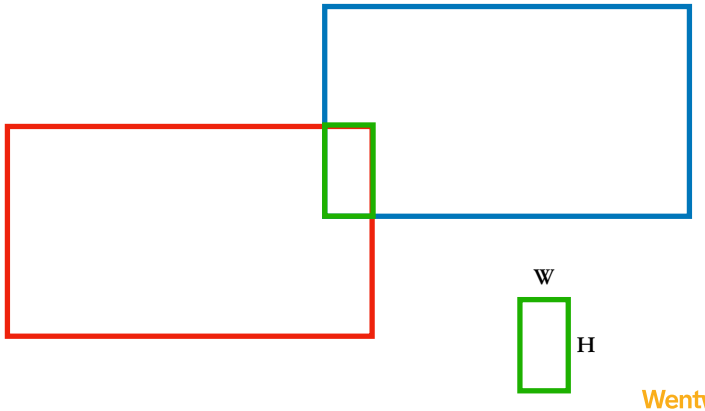
The Collision Problem (Simplifed)

- Given two entities that have a current position (only) do they intersect?



The Collision Problem (Simplifed)

- Given two entities that have a current position (only) do they intersect?

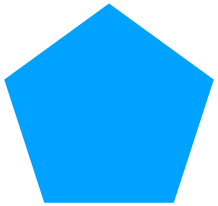


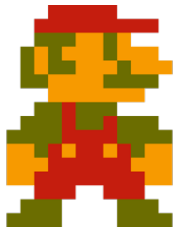
Bounding Shapes

- Objects can have arbitrary shapes/surfaces that interact
- An arbitrary shape is difficult to store and simulate accurately
- In most games, bounding shapes are just primitive objects:
  - 2D: Triangle, Circle, Rectangle, Line, etc.
  - 3D: Sphere, Plane, Prism, Capsule, etc.

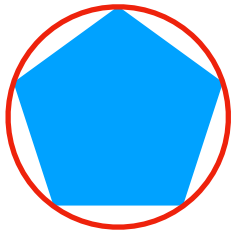
Computing interactions between simple shapes is much easier than with complex shapes

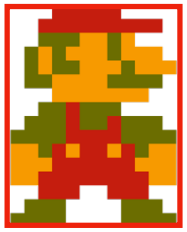
Bounding Shape Example





Bounding Shape Example





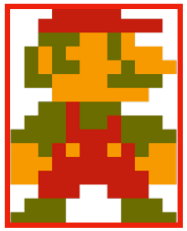
Often, the bounding shape is set to be smaller than the actual graphic.

2D Bounding Shapes

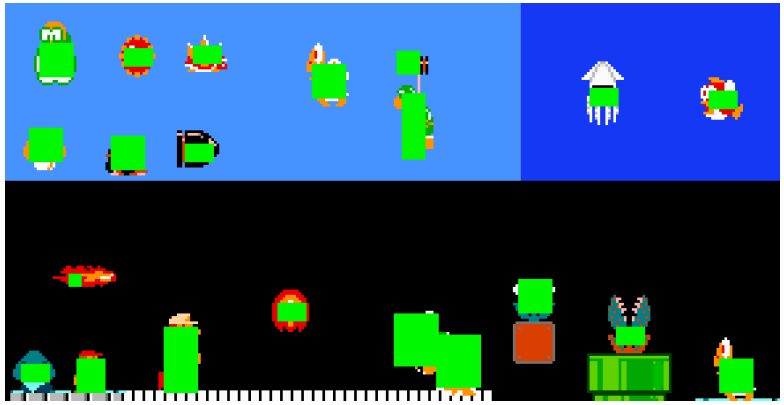
- The simplest 2D bounding shape is the circle.
- We already covered circle intersection
- Other bounding shapes can be constructed with line segments
- The simplest of these is a rectangle
  - Bounding Rectangle/Bounding Box
  - The intersection math is easy, but not trivial

Bounding Box Size

- Most classic 2D games use rectangular sprite graphics.
- Naturally, they also use rectangles for collisions
- Usually, the bounding box is the smallest rectangle that fits the sprite so that we can use the texture size as the bounding box size
  - This is not always the case!

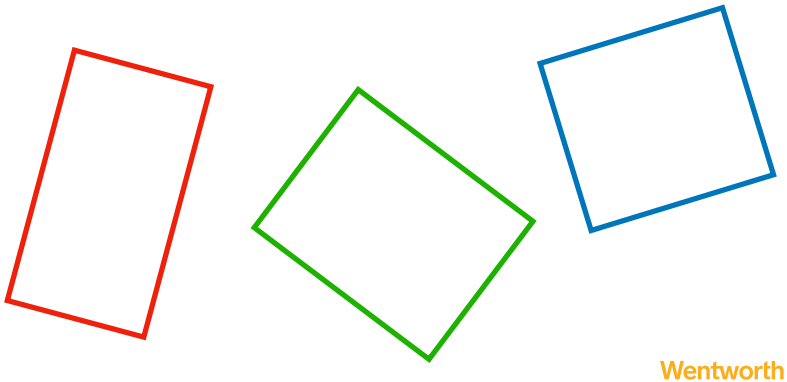


Bounding Box Size



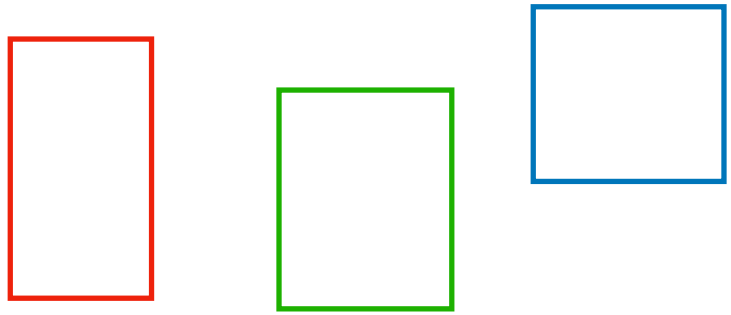
Axis Aligned Bounding Boxes

- Rectangles in 2D can take any orientation as long as the four corners form 90 degree angles:



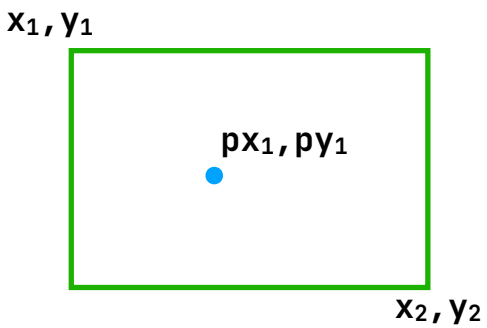
Axis Aligned Bounding Boxes

- By aligning our rectangles with the xy axis, we can do very fast collision checks
- Axis Aligned Bounding Box: **AABB**



Point Inside AABB

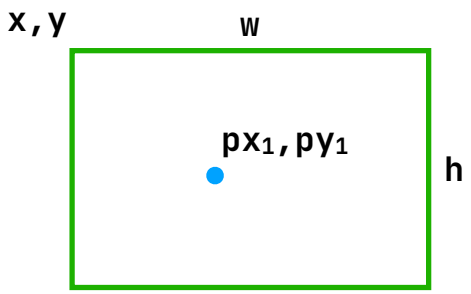
- Assume `(0,0)` is the top left of our window




```
P is inside if:
px1 > x1
px1 < x2
py1 > y1
py1 < y2
```


Point Inside AABB

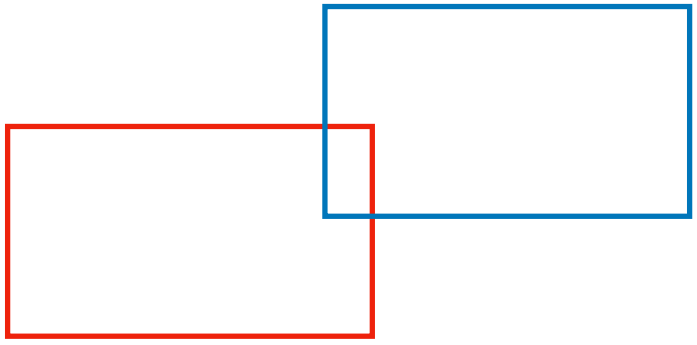
- Assume `(0,0)` is the top left of our window




```
P is inside if:
px1 > x
px1 < x+w
py1 > y
py1 < y+h
```

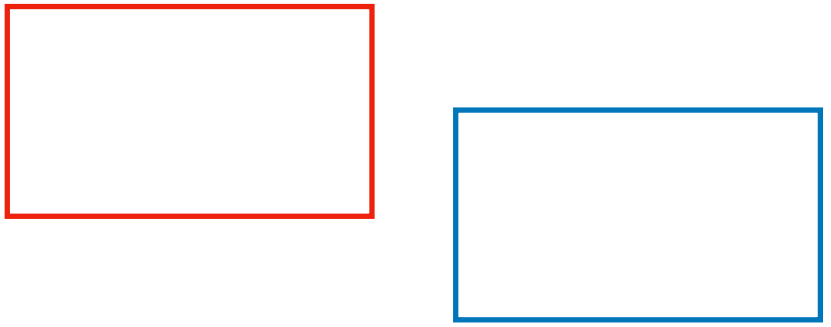

Next Step: AABB Intersection

- Determine if any two AABBs intersect anywhere

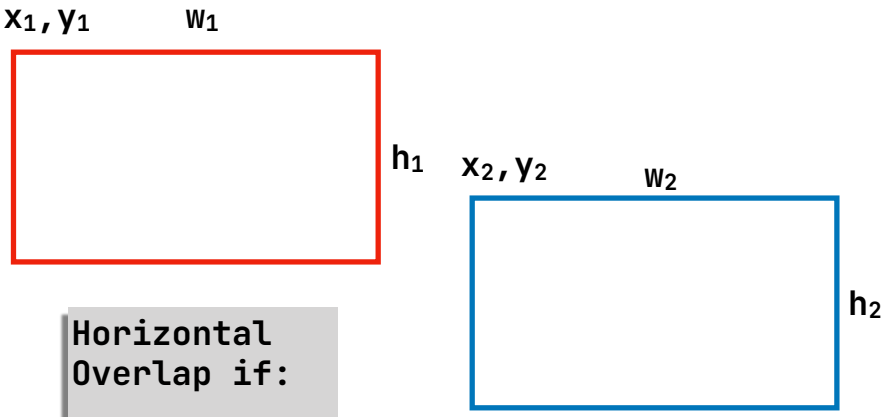


Horizontal Overlap

- Horizontal overlap occurs if the top of each box is higher than the bottom of the other



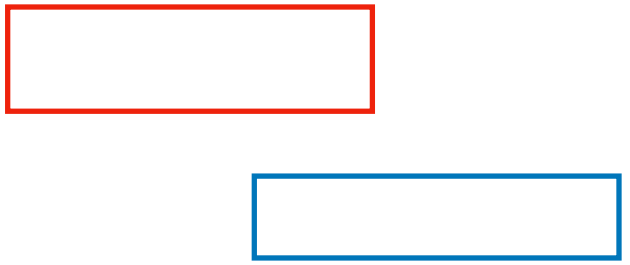
Horizontal Overlap




```
Horizontal
Overlap if:
y1 < y2+h2
y2 < y1+h1
```

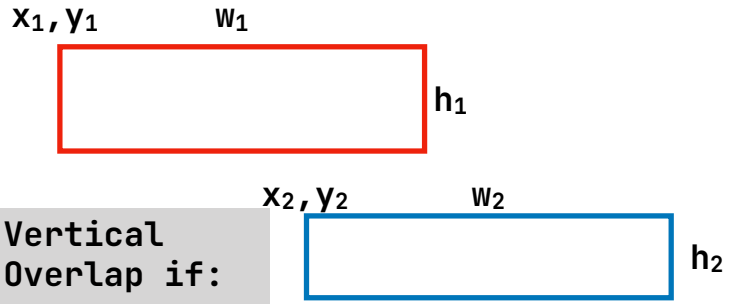

Vertical Overlap

- Vertical Overlap occurs if the left of each box is less than the right of the other



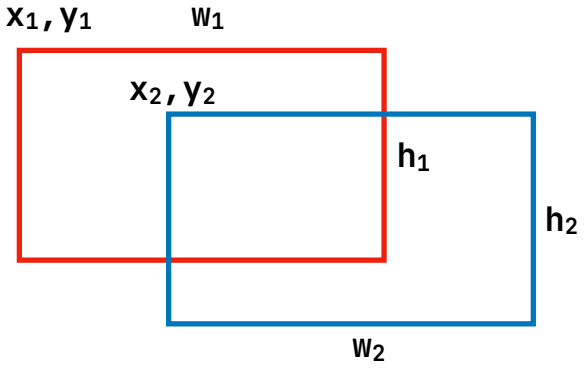
Vertical Overlap

- Vertical Overlap occurs if the left of each box is less than the right of the other




```
Vertical
Overlap if:
x1 < x2+w2
x2 < x1+w1
```

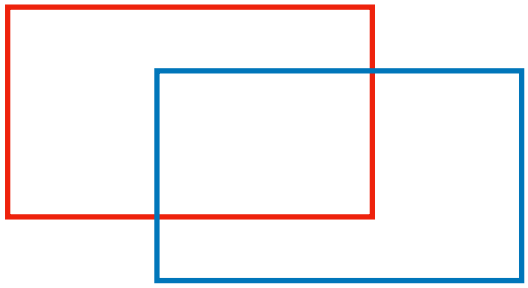

AABB Intersection



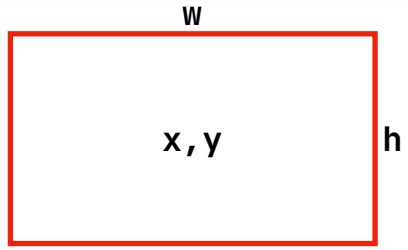

```
Intersection
if:
x1 < x2+w2
x2 < x2+w1
y1 < y2+h2
y2 < y2+h1
```

AABB Intersection

- This is OK, but it only tells us **if** there was a collision, not by how much...
- We're going to look at a centered intersection

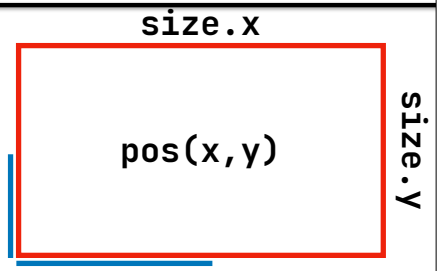


AABB Intersection



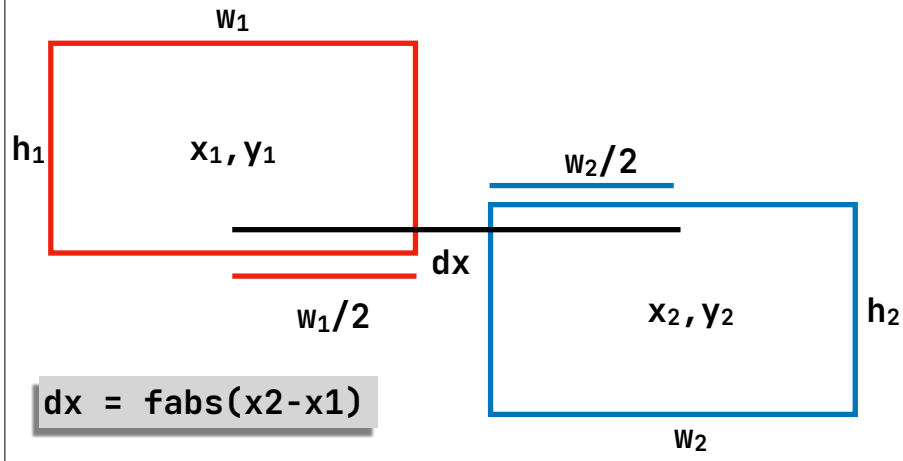
AABB Intersection

We'll store the size of the bounding box and the half size



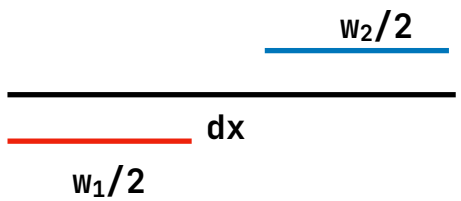

```
class CBBox : public component {
public:
  Vec2 size;
  Vec2 halfSize;
  CBBox(const Vec2& s) : size(s),
halfSize(s.x/2, s.y/2) {}
```


AABB Intersection




```
dx = fabs(x2-x1)
```

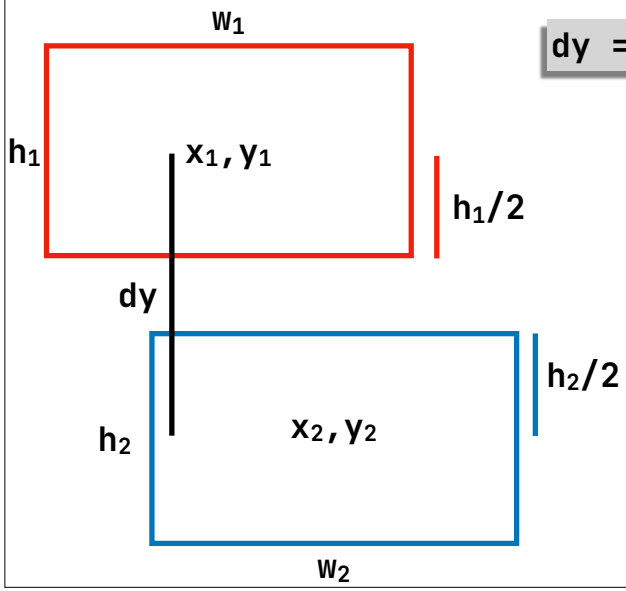
AABB Intersection




```
dx = fabs(x2-x1)
```

```
Overlap in the x
direction:
(w2/2) + (w1/2) - dx
Positive if x overlap
```

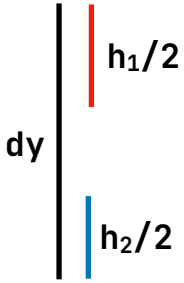

AABB Intersection




```
dy = fabs(y2-y1)
```

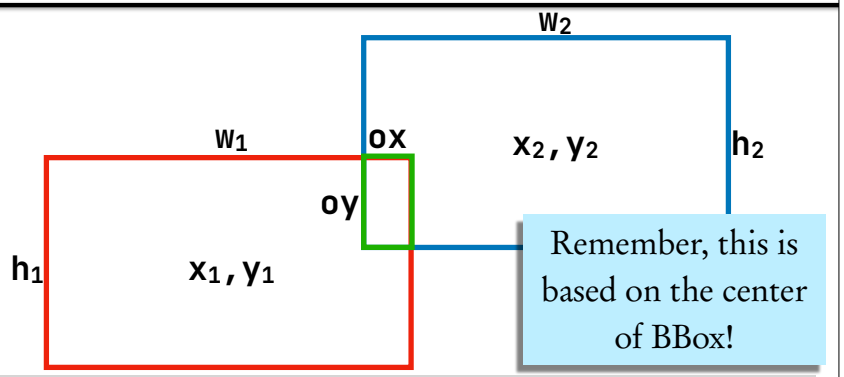

AABB Intersection

```
dy = fabs(y2-y1)
```




```
Overlap in the y
direction:
(h2/2) + (h1/2) - dy
Positive if y overlap
```


AABB Intersection

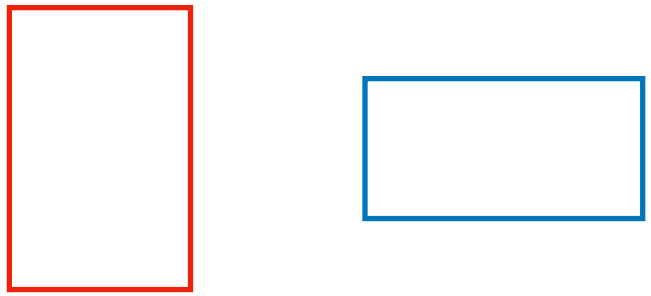



```
delta = [fabs(x1-x2), fabs(y1-y2)]
ox = (w1/2) + (w2/2) - delta.x
oy = (h1/2) + (h2/2) - delta.y
overlap = [ox, oy]
```

Resolving Collisions

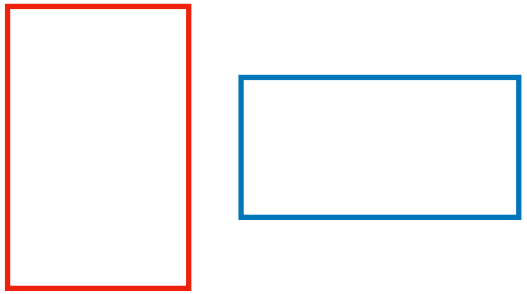
- Now that we can detect collisions between bounding boxes, what do we do?
- Resolution of collisions is highly dependent on what type of physics you want to represent
  - For example: If a player collides with a ground tile, it should be "pushed out" with no overlap

Resolving Collisions



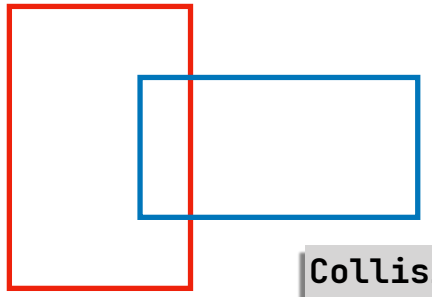
Frame 1

Resolving Collisions



Frame 3

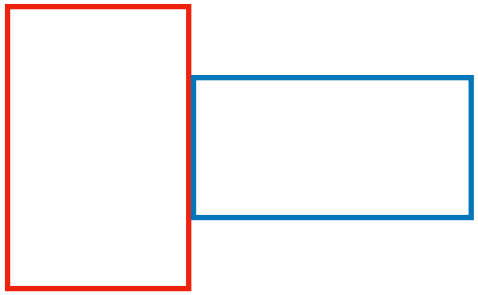
Resolving Collisions



```
Collision Detected!
Resolution:
Blue.pos += overlap
```

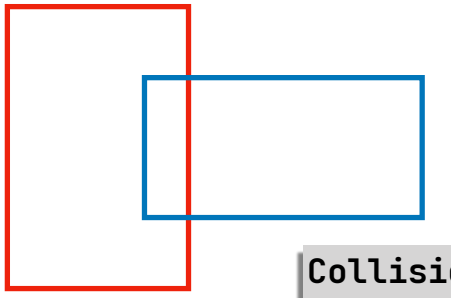
Frame 5

Resolving Collisions



Frame 5

Resolving Collisions

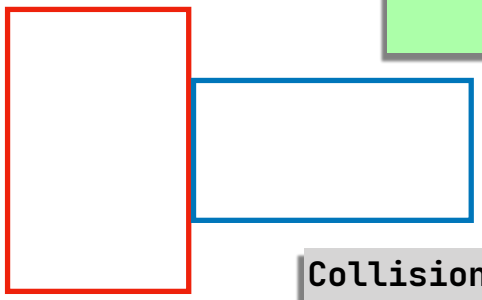


```
Collision Detected!
Resolution:
Blue.pos += overlap
```

Frame 6

Resolving Collisions

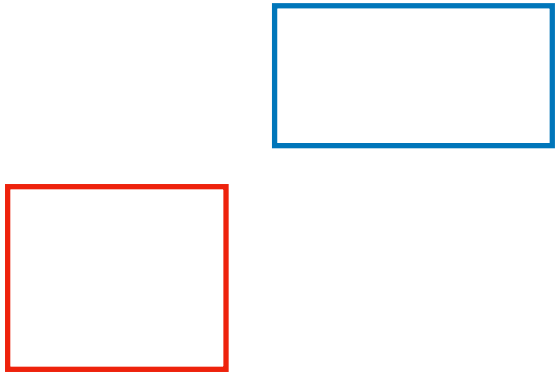
Every frame, we make sure there is no overlap



```
Collision Detected!
Resolution:
Blue.pos += overlap
```

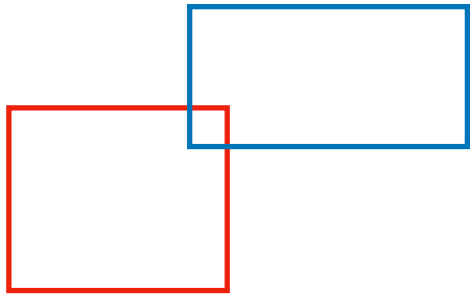
Frame 7

Tricky Collisions

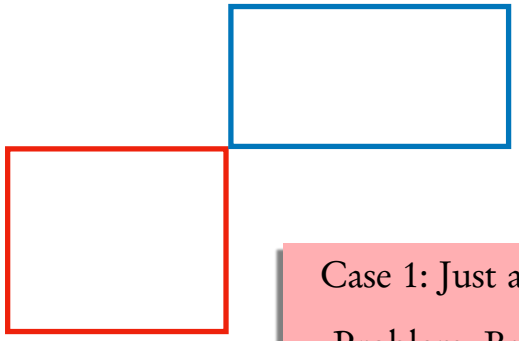


Tricky Collisions

What do we do here?



Tricky Collisions

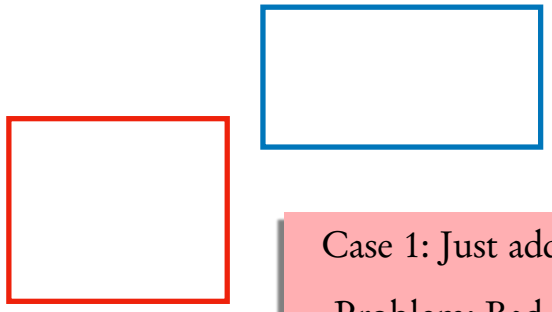


Case 1: Just add the overlap

Problem: Bad physics and doesn't look correct

Tricky Collisions

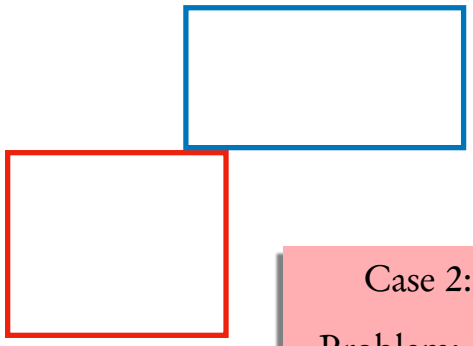
What if we came from the right?



Case 1: Just add the overlap

Problem: Bad physics and doesn't look correct

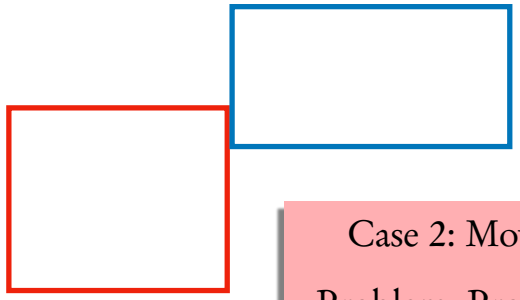
Tricky Collisions



Case 2: Move up only

Problem: Again, what if we came from the right?

Tricky Collisions



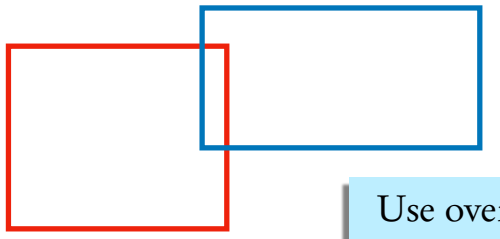
Case 2: Move right only

Problem: Probably the best, but still not good if you came from the top

Detecting Collision Direction

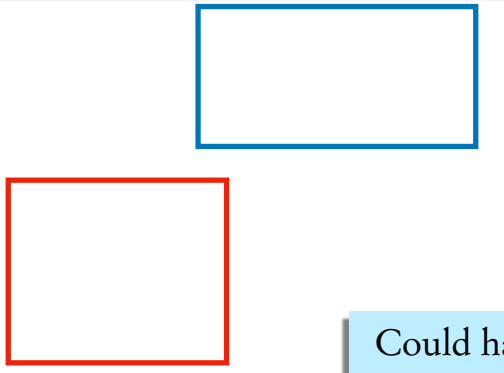
- Hopefully you get the idea of some of the tricky cases where we need more information
- Typically, on the frame before a collision, there is no collision (it hasn't happened yet)
- Can we try to use overlap to detect which direction the movement came from?

Tricky Resoultion Cases



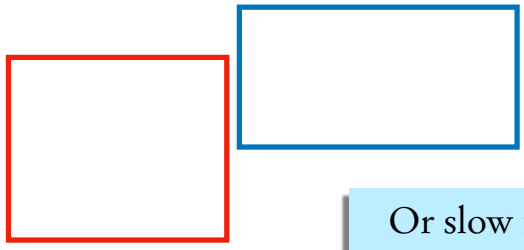
Use overlap to detect direction of movement?

Tricky Resoultion Cases



Could have came fast from the y direction...

Tricky Resoultion Cases



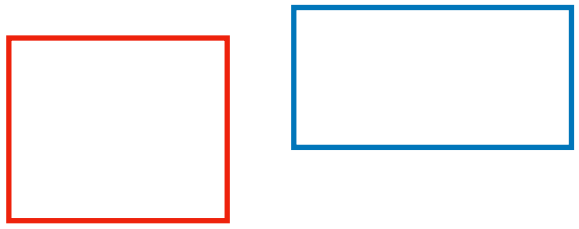
Or slow from the x direction...

Detecting Collision Direction

- Hopefully you get the idea of some of the tricky cases where we need more information
- Typically, on the frame before a collision, there is no collision (it hasn't happened yet)
- Can we try to use overlap to detect which direction the movement came from?
- **Use previous overlap to detect direction**

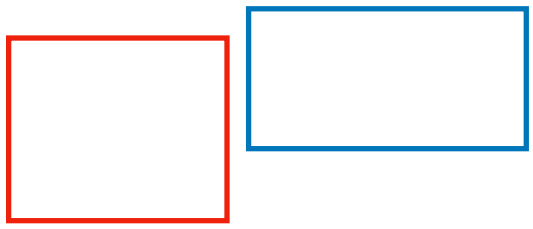
Using Previous Overlap

- If previous frame `overlap.y > 0`, then the movement came from the side (`x`)
- Resolve the collision by pushing in the x direction



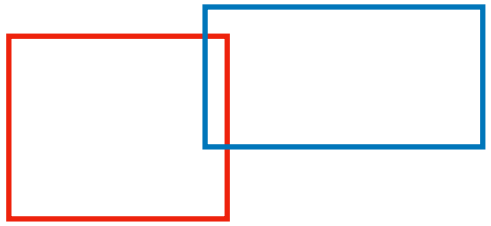
Using Previous Overlap

- If previous frame `overlap.y > 0`, then the movement came from the side (`x`)
- Resolve the collision by pushing in the x direction



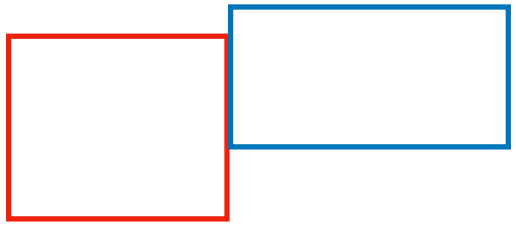
Using Previous Overlap

- If previous frame **`overlap.y > 0`**, then the movement came from the side (**`x`**)
- Resolve the collision by pushing in the x direction



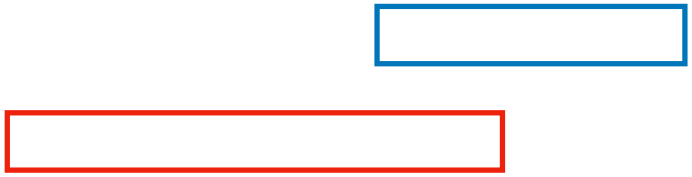
Using Previous Overlap

- If previous frame **`overlap.y > 0`**, then the movement came from the side (**`x`**)
- Resolve the collision by pushing in the x direction



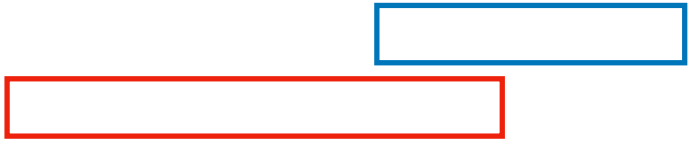
Using Previous Overlap

- If previous frame **`overlap.x > 0`**, then the movement came from the top or bottom (**`y`**)
- Resolve the collision by pushing in the y direction



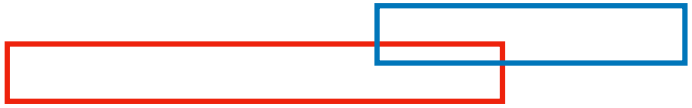
Using Previous Overlap

- If previous frame **`overlap.x > 0`**, then the movement came from the top or bottom (**`y`**)
- Resolve the collision by pushing in the y direction



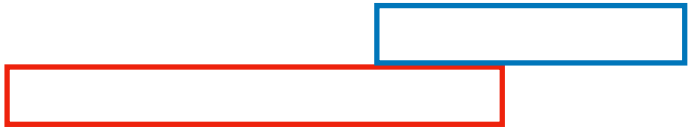
Using Previous Overlap

- If previous frame **`overlap.x > 0`**, then the movement came from the top or bottom (**`y`**)
- Resolve the collision by pushing in the y direction



Using Previous Overlap

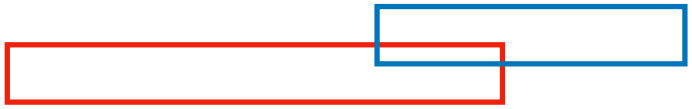
- If previous frame **`overlap.x > 0`**, then the movement came from the top or bottom (**`y`**)
- Resolve the collision by pushing in the y direction



Wentworth
Computing & Data Science

Top/Bottom Detection

- A collision that came from the top has a previous **`overlap.x > 0`** and its **`y`** is higher
- Push in the negative y direction



Wentworth
Computing & Data Science

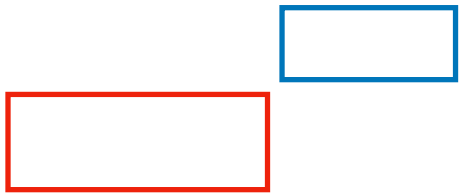
Top/Bottom Detection

- A collision that came from the bottom has a previous **`overlap.x > 0`** and its **`y`** is lower
- Push in the y direction

Wentworth
Computing & Data Science

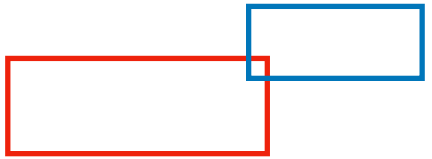
More Tricky Resolutions

- What if the previous frame had no overlap in either x or y?
- We must decide how to resolve this based on previous position or overlap size



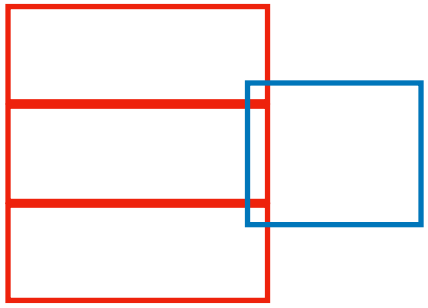
More Tricky Resolutions

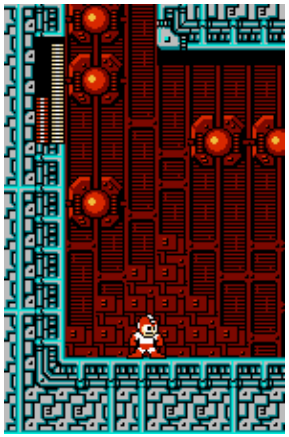
- What if the previous frame had no overlap in either x or y?
- We must decide how to resolve this based on previous position or overlap size



More Tricky Resolutions

- Order of collision checks may determine which resolution happens:





Important Note

- All collision resolution here is for the simple case of avoiding overlap
- Best used for: walking on tiles, not passing through tiles, or when a bullet hits an enemy
- There are many other kinds of resolutions:
  - Elastic or Inelastic collision (you do elastic collisions for A1 and A2 when bouncing off walls)
  - Physics simulations with mass and/or forces

Side Note: Mario Bros

- Super Mario Bros collisions work slightly differently that I presented
- How does Mario kill an enemy only when he jumps on top of them?
- Same AABB overlap calculations, but Mario kills the enemy if he is moving downward (the x direction is ignored)

Fun Resources

- "How is this speedrun possible?"
  - https://www.youtube.com/watch?v=_FQJEzJ_cQw
- "Walls, Floors, and Ceilings"
  - https://www.youtube.com/watch?v=UnU7DJXiMAQ
- "Watch for Rolling Rocks" (SM64)
  - https://www.youtube.com/watch?v=aFx7woWkZbc&t=0s