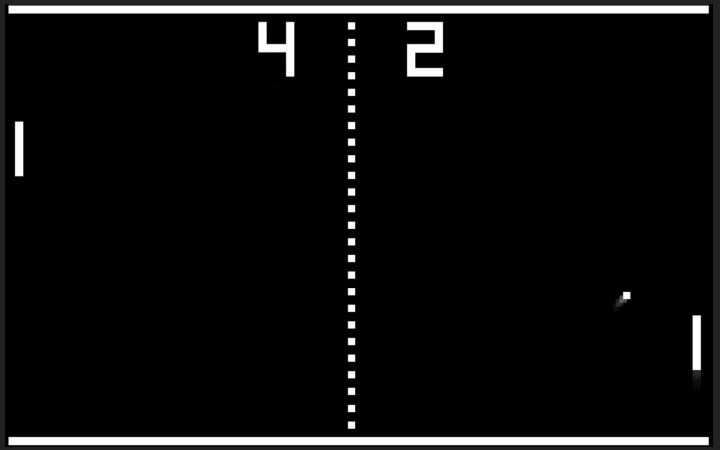
Input


PONG
PLAYER 1
PLAYER 2

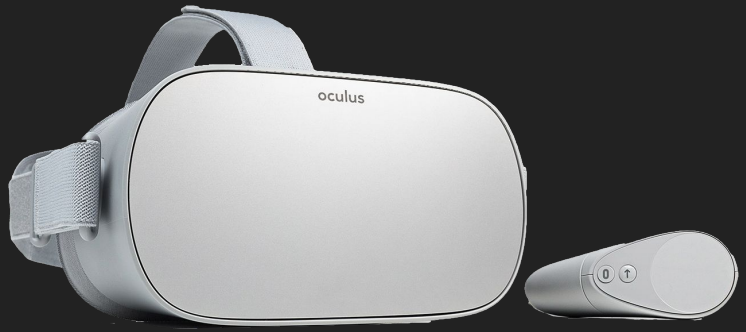

4
2


PONG
ATARI


Nintendo®
SELECT START
B A

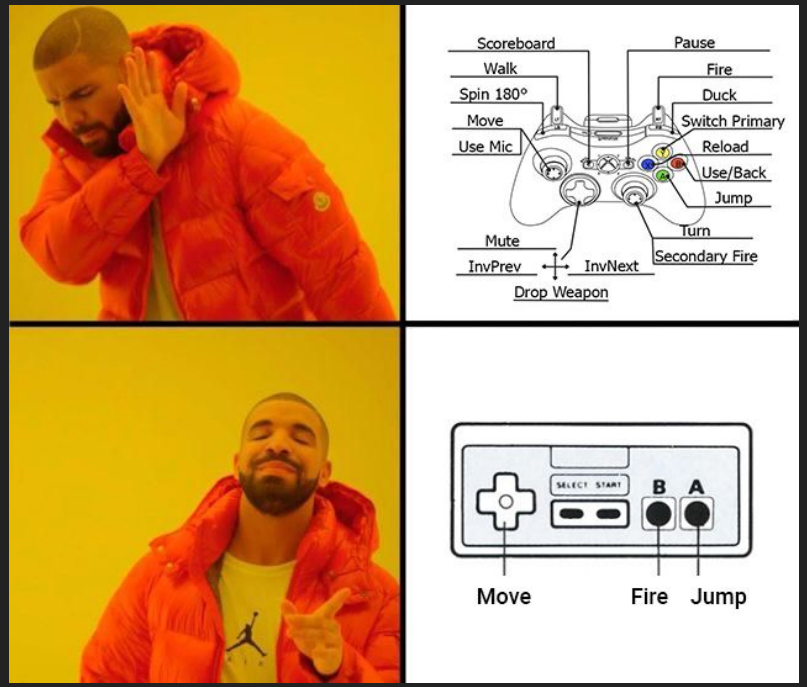

L
R
SONY
SELECT
START


Nintendo



Your Professor 1999

Our Games


Scoreboard
Walk
Spin 180°
Move
Use Mic
Pause
Fire
Duck
Switch Primary
Reload
Use/Back
Jump
Turn
Secondary Fire
Mute
InvPrev
InvNext
Drop Weapon
SELECT START
B
A
Move
Fire
Jump

Keyboard Input

The Game Loop

```
Startup();

while (gameIsRunning) {
    ProcessInput();
    Update();
    Render();
}

Shutdown();
```

ProcessInput();

```
SDL_Event event;
while (SDL_PollEvent(&event)) {
    if (event.type == SDL_QUIT
            || event.type == SDL_WINDOWEVENT_CLOSE) {
            gameIsRunning = false;
    }
}
```

ProcessInput();

(you may want to use a switch/case for event types)

```
switch (event.type) {
    case SDL_QUIT:
    case SDL_WINDOWEVENT_CLOSE:
        gameIsRunning = false;
        break;
}
```

ProcessInput();

SDL_KEYDOWN is when a key is pressed.
SDL_KEYUP is when a key is released.

```
// Check if a key was pressed
case SDL_KEYDOWN:
    switch(event.key.keysym.sym) {
        // ..
    }

    break;
```

ProcessInput();

```
// Check which key was pressed
// https://wiki.libsdl.org/SDL_Scancode

switch(event.key.keysym.sym) {

        case SDLK_RIGHT:
            player_x += 1;
            break;

        case SDLK_SPACE:
            PlayerJump();
            break;
    }
```

```
void ProcessInput() {
    SDL_Event event;
    while (SDL_PollEvent(&event)) {
        switch (event.type) {
            case SDL_QUIT:
            case SDL_WINDOWEVENT_CLOSE:
                gameIsRunning = false;
                break;

            case SDL_KEYDOWN:
                switch (event.key.keysym.sym) {
                    case SDLK_LEFT:
                        // Move the player left
                        break;

                    case SDLK_RIGHT:
                        // Move the player right
                        break;

                    case SDLK_SPACE:
                        // Some sort of action
                        break;
                }
                break; // SDL_KEYDOWN
        }
    }
}
```

SDL_KEYDOWN and SDL_KEYUP

(great for knowing when a key was pressed or released)

Useful for actions such as jumping and shooting.

(but we need something else for a key held down)

SDL_GetKeyboardState

Returns a pointer to an array of key states. A value of 1 means that the key is pressed and a value of 0 means that it is not. Indexes into this array are obtained by using SDL_Scancode values. The pointer returned is a pointer to an internal SDL array. It will be valid for the whole lifetime of the application and should not be freed by the caller.

SDL_GetKeyboardState

```
const Uint8 *keys = SDL_GetKeyboardState(NULL);

if (keys[SDL_SCANCODE_LEFT]) {
    PlayerLeft();
}

if (keys[SDL_SCANCODE_RIGHT]) {
    PlayerRight();
}

// Notice the above use SDL_SCANCODE_ and not SDLK_
// https://wiki.libsdl.org/SDL_Scancode
```

SDL_KEYDOWN and SDL_KEYUP

(used inside of the while loop for processing events)

SDL_GetKeyboardState

(used outside of the while loop for processing events)

Mouse Input

SDL_MOUSEMOTION

(happens when the mouse is moved)

```
case SDL_MOUSEMOTION:
    // event.motion.x        : x position in pixels
    // event.motion.y        : y position in pixels
     break;
```

SDL_MOUSEBUTTONDOWN

```
case SDL_MOUSEBUTTONDOWN:
    // event.button.x       : x position in pixels
    // event.button.y       : y position in pixels
    // event.button.button  : button that was clicked (1,2,3)
    break;
```

Polling the Mouse

```
// put this outside of “while (SDL_PollEvent(&event))”

int x, y;

SDL_GetMouseState(&x, &y);
```

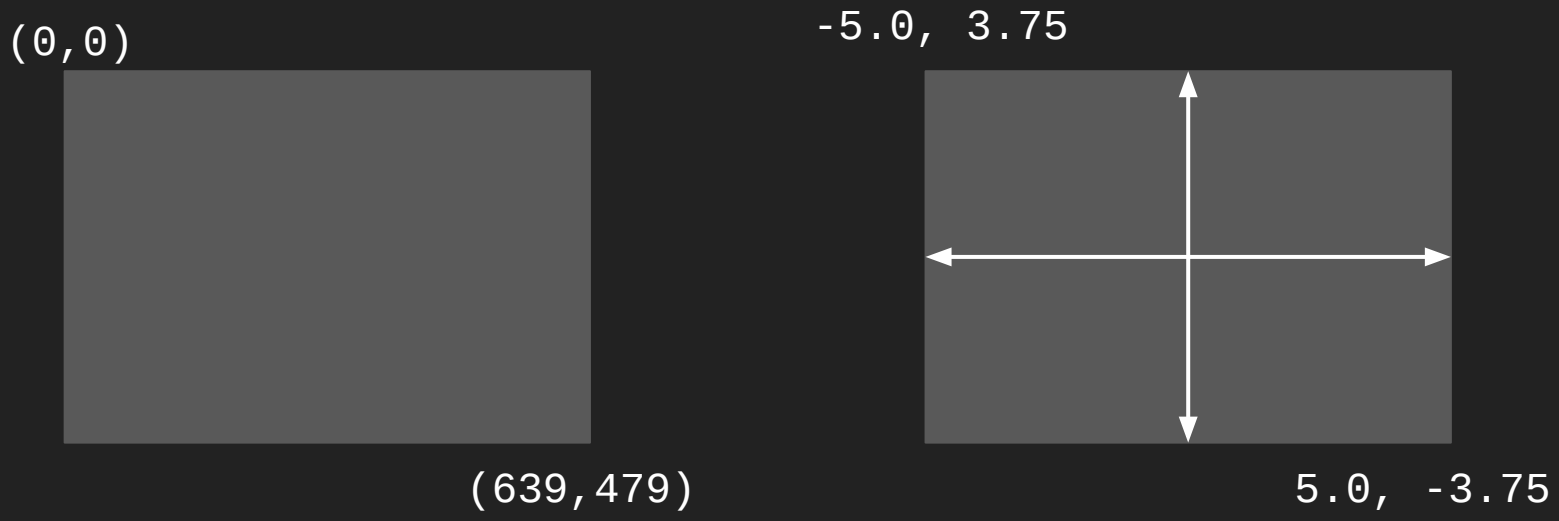
Mouse Coordinates are in Pixels!

(not your world coordinates)


640 pixels
x = 0
y = 0
480 pixels
x = 639
y = 479

We need to convert from pixel coordinates to OpenGL units.

```
glm::ortho(-5.0f, 5.0f, -3.75f, 3.75f, -1.0f, 1.0f);
```



We need to convert from pixel coordinates to OpenGL units.

```
glm::ortho(-5.0f, 5.0f, -3.75f, 3.75f, -1.0f, 1.0f);
```

```
// Convert mouse x, y to world unit x, y
// Assumes we are looking at 0,0 in our world.

unit_x = ((x / width) * ortho_width) - (ortho_width / 2.0);
unit_y = (((height - y) / height) * ortho_height) - (ortho_height / 2.0);
```

Controller Input

Initialization

```
SDL_Joystick *playerOneController;

void Initialize() {

    // Initialize Video and the Joystick subsystem
    SDL_Init(SDL_INIT_VIDEO | SDL_INIT_JOYSTICK);

    // Open the 1st controller found. Returns null on error.
    playerOneController = SDL_JoystickOpen(0);

    // Do the other stuff
}
```

Cleanup

```
SDL_JoystickClose(playerOneController);
```

Checking for Controllers

You can SDL_NumJoysticks() to get the number of controllers.

Axis and Button Events

SDL_JOYAXISMOTION
SDL_JOYBUTTONDOWN
SDL_JOYBUTTONUP

Axes and Buttons

(these might be different on your system/controller/etc.)

Axes

0: Left Stick - X Axis
1: Left Stick - Y Axis

3: Right Stick - X Axis
4: Right Stick - Y Axis

2: Left Trigger
5: Right Trigger

Buttons

0: A
1: B
2: X
3: Y
4: LB
5: RB
6: L3/LS
7: R3/RS
8: Start
9: Select
10: Home
11: DPad Up
12: DPad Down
13: DPad Left
14: DPad Right

Axes
0: Left Stick - X Axis
1: Left Stick - Y Axis

3: Right Stick - X Axis
4: Right Stick - Y Axis

2: Left Trigger
5: Right Trigger

Buttons
0: A
1: B
2: X
3: Y
4: LB
5: RB
6: L3/LS
7: R3/RS
8: Start
9: Select
10: Home
11: DPad Up
12: DPad Down
13: DPad Left
14: DPad Right

```
while (SDL_PollEvent(&event)) {
    switch (event.type) {
        case SDL_QUIT:
        case SDL_WINDOWEVENT_CLOSE:
            gameIsRunning = false;
            break;

        case SDL_JOYAXISMOTION:
            // event.jaxis.which        : Which controller (usually 0)
            // event.jaxis.axis         : Which Axis
            // event.jaxis.value        : -32768 to 32767
            break;

        case SDL_JOYBUTTONDOWN:
            // event.jbutton.which      : Which controller (usually 0)
            // event.jbutton.button     : Which button
            break;
    }
}
```

SDL_JOYAXISMOTION and SDL_JOYBUTTONDOWN

Similar to keyboard events. Great for knowing when something happened, but does not handle sustained usage.

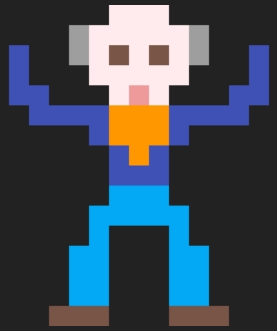
(but we need something else)

Polling the Controller

(used outside of that while loop)

```
SDL_JoystickGetAxis(playerOneController, axisIndex);

SDL_JoystickGetButton(playerOneController, buttonIndex);
```

Movement



The Game Loop

```
Startup();

while (gameIsRunning) {
   ProcessInput();
   Update();
   Render();
}

Shutdown();
```

The Game Loop

ProcessInput()
- Store the player’s intent to move/jump/etc.

Update()
- Test/Apply movement.
- Player, enemies, moving platforms, etc.

Render()
- Draw the current state of the game.

How do we do this?

Vectors!

We can store the player’s position as a vector as well as an intended movement.

```
// Start at 0, 0, 0
glm::vec3 player_position = glm::vec3(0, 0, 0);

// Don’t go anywhere (yet).
glm::vec3 player_movement = glm::vec3(0, 0, 0);
```

Set where we want to go in ProcessInput()

```
player_movement = glm::vec3(0, 0, 0);

const Uint8 *keys = SDL_GetKeyboardState(NULL);

if (keys[SDL_SCANCODE_A]) {
    player_movement.x = -1.0f;
}
else if (keys[SDL_SCANCODE_D]) {
    player_movement.x = 1.0f;
}
```

All movement needs to consider timing.

```
float lastTicks = 0.0f;

void Update() {
    float ticks = (float)SDL_GetTicks() / 1000.0f;
    float deltaTime = ticks - lastTicks;
    lastTicks = ticks;

    // Add (direction * units per second * elapsed time)
    player_position += player_movement * player_speed * deltaTime;

    modelMatrix = glm::mat4(1.0f);
    modelMatrix = glm::translate(modelMatrix, player_position);
}
```

Look Out!

Joysticks are in a circle, however the WASD keys would make a square...

The pythagorean theorem: $A^2 + B^2 = C^2$

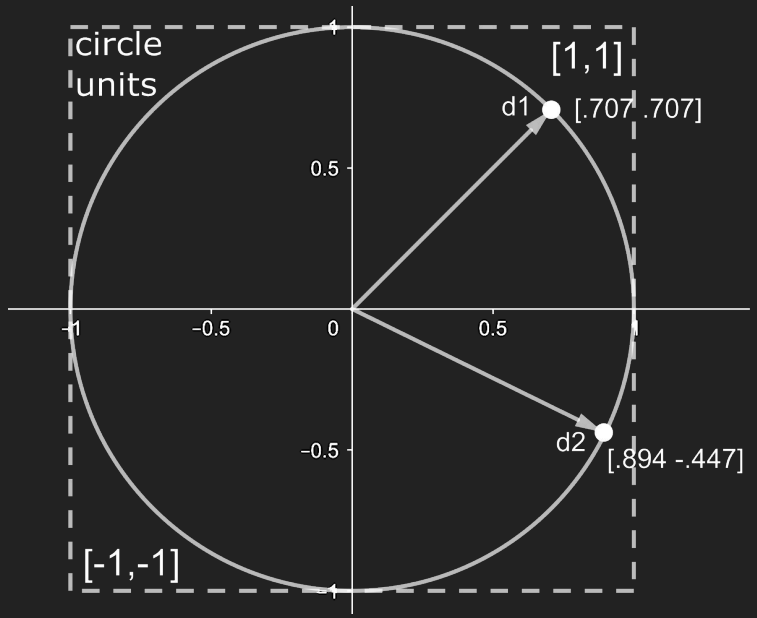
Pressing D movement vector = [1, 0] : What is the magnitude? 1

Pressing W movement vector = [0, 1] : What is the magnitude? 1

Pressing W and D movement vector = [1, 1] : What is the magnitude? 1.414

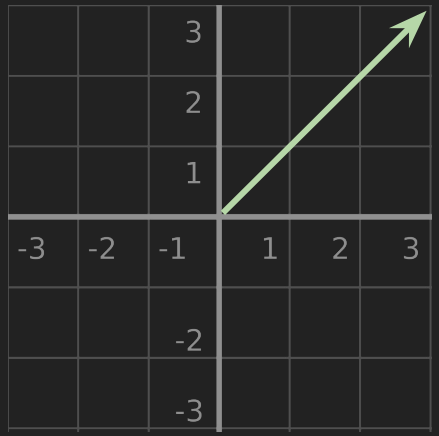
Unit Vector

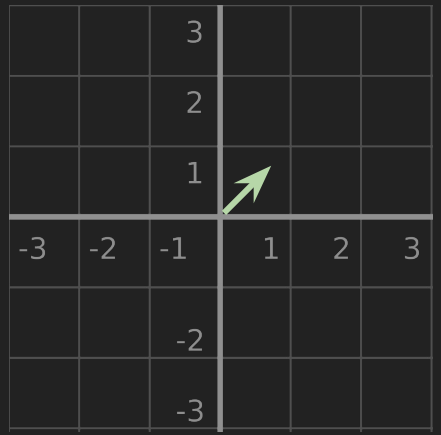
(a vector with a magnitude of 1)



Unit Vector

We can **normalize** a vector to get the unit vector.





You normalize a vector by dividing it by the magnitude.
There is a glm::normalize function. Do not use it if the magnitude = 0

How do we code that?

```
if (glm::length(player_movement) > 1.0f) {
    player_movement = glm::normalize(player_movement);
}
```

Let’s Code!

We’re almost ready to make a game!

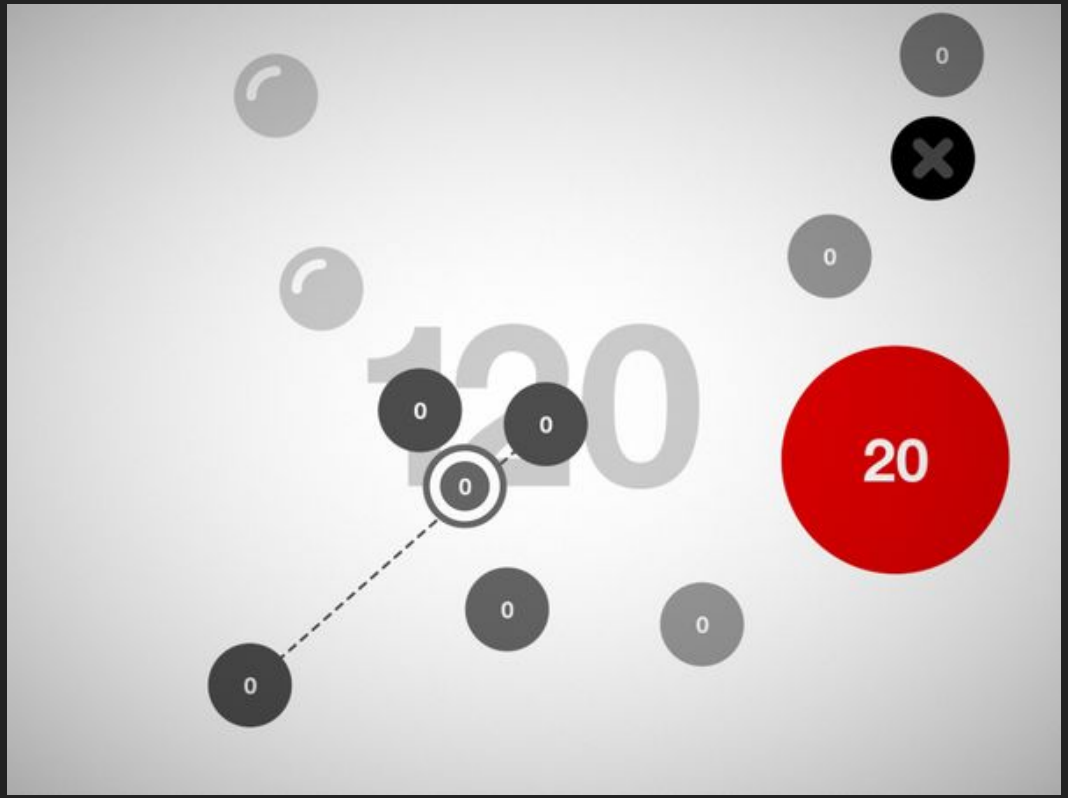
What we've learned so far...

Initializing SDL, OpenGL, creating a window.
Drawing a triangle.
Transformations: translation, rotation, scale.
Handling timing.
Loading images and rendering textures.
Keyboard, mouse and controller input.
Movement based on unit vector, speed and deltaTime.

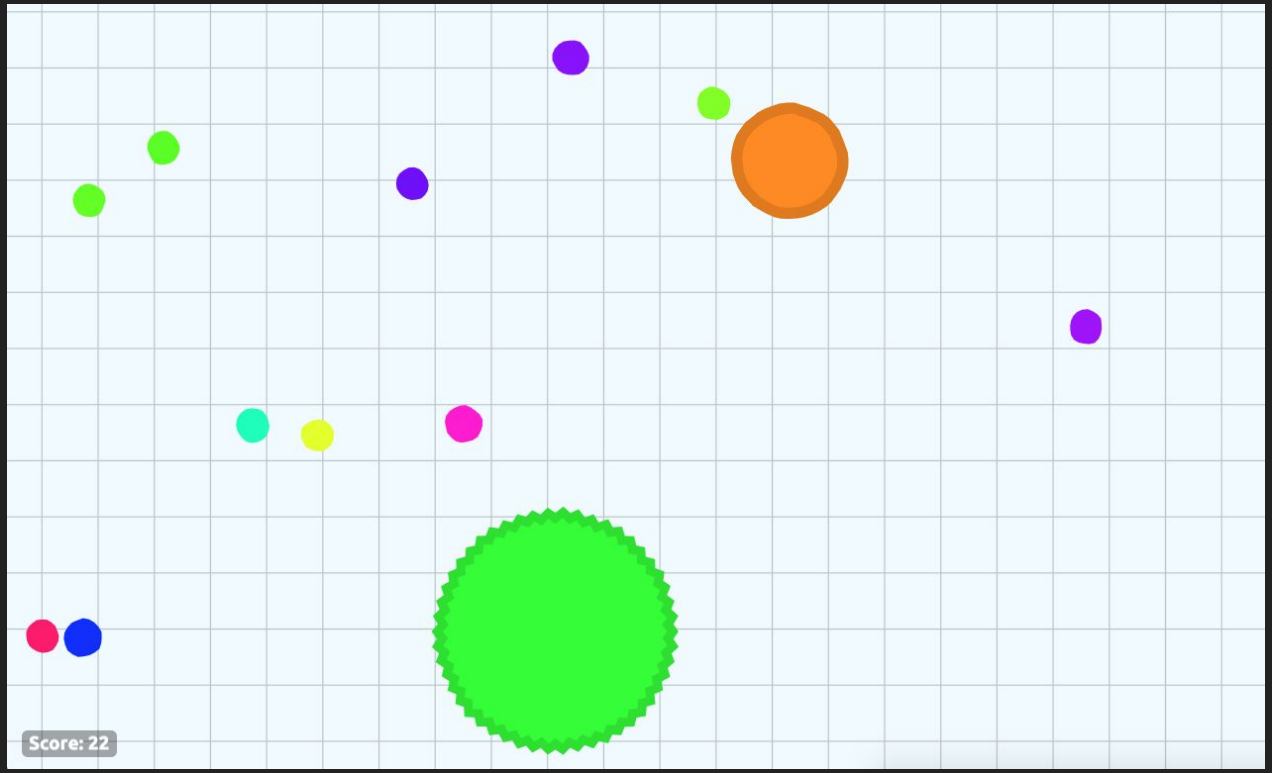
We need one more thing to make our first game...

Collision Detection!

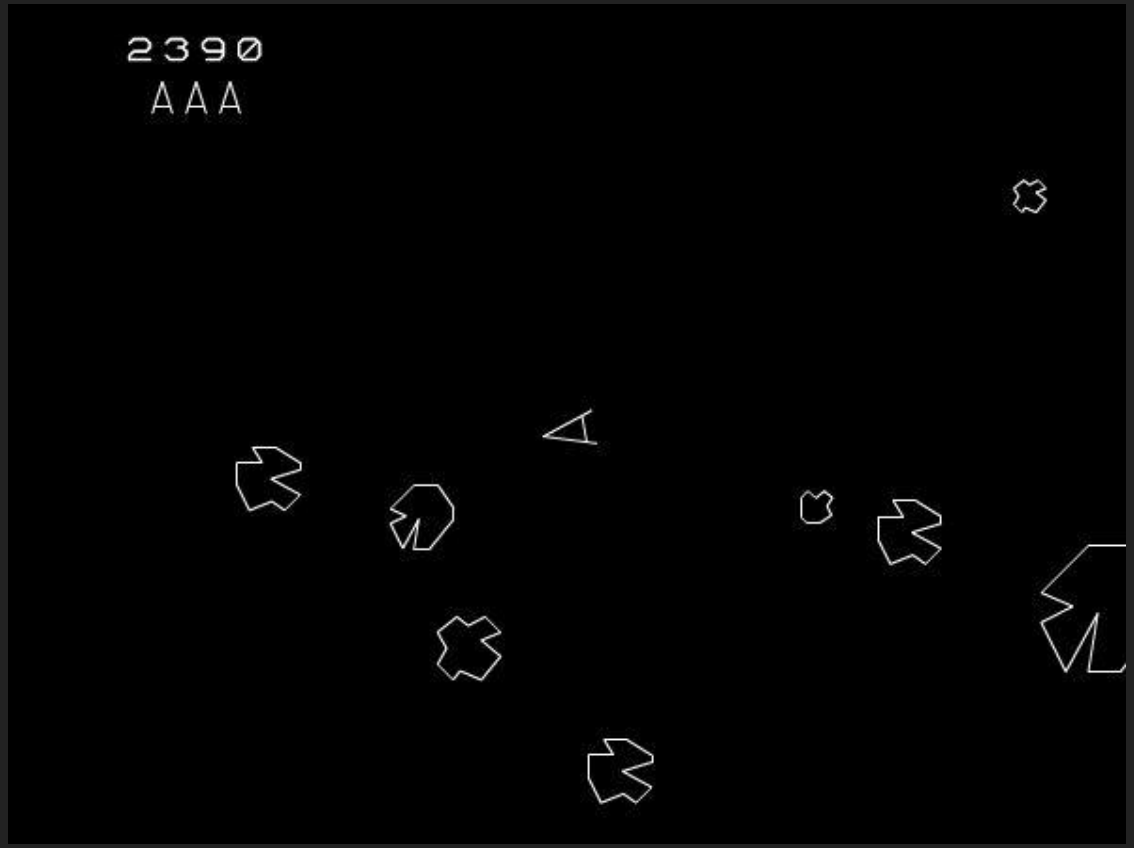
Circle - Circle Collision Detection



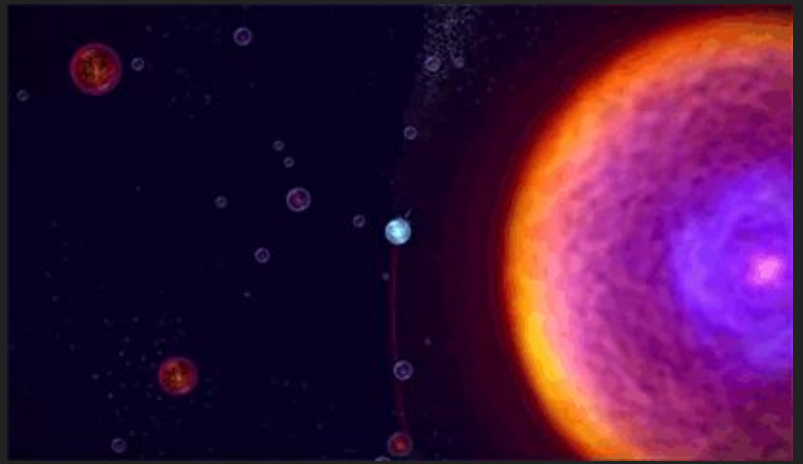
Hundreds (iOS/Android)



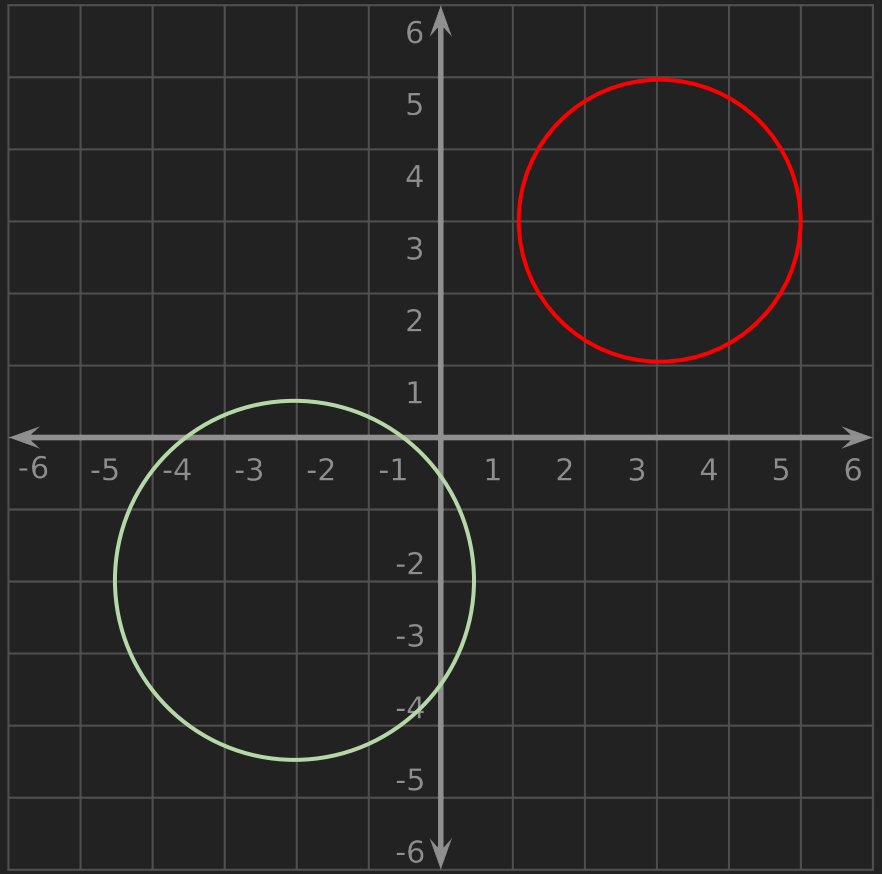
https://agar.io

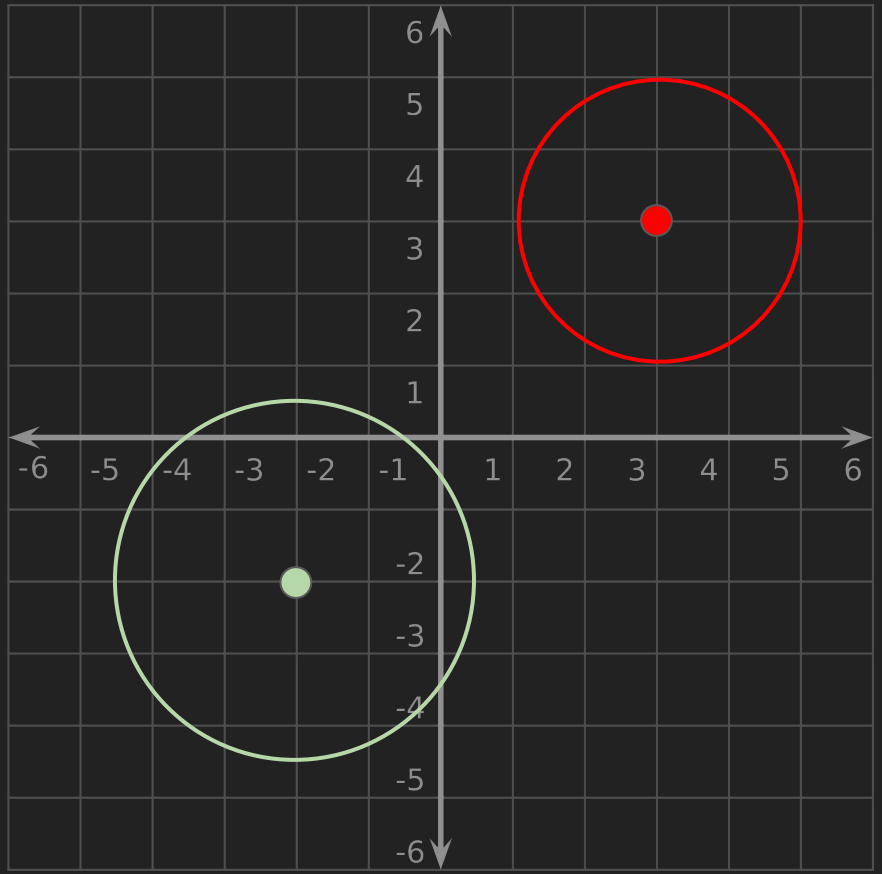


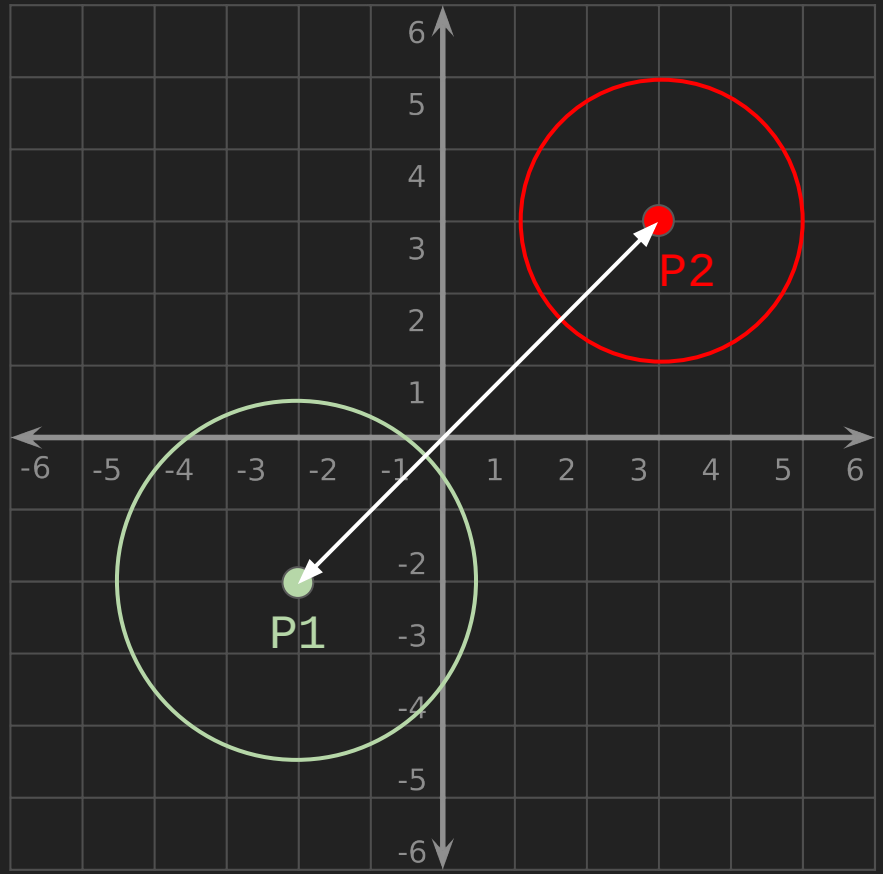
Asteroids - 1979 Arcade



Osmos
by Andy Nealen
NYU Professor!


6
5
4
3
2
1
-6
-5
-4
-3
-2
-1
1
2
3
4
5
6
-2
-3
-4
-5
-6


6
5
4
3
2
1
-6
-5
-4
-3
-2
-1
1
2
3
4
5
6
-2
-3
-4
-5
-6



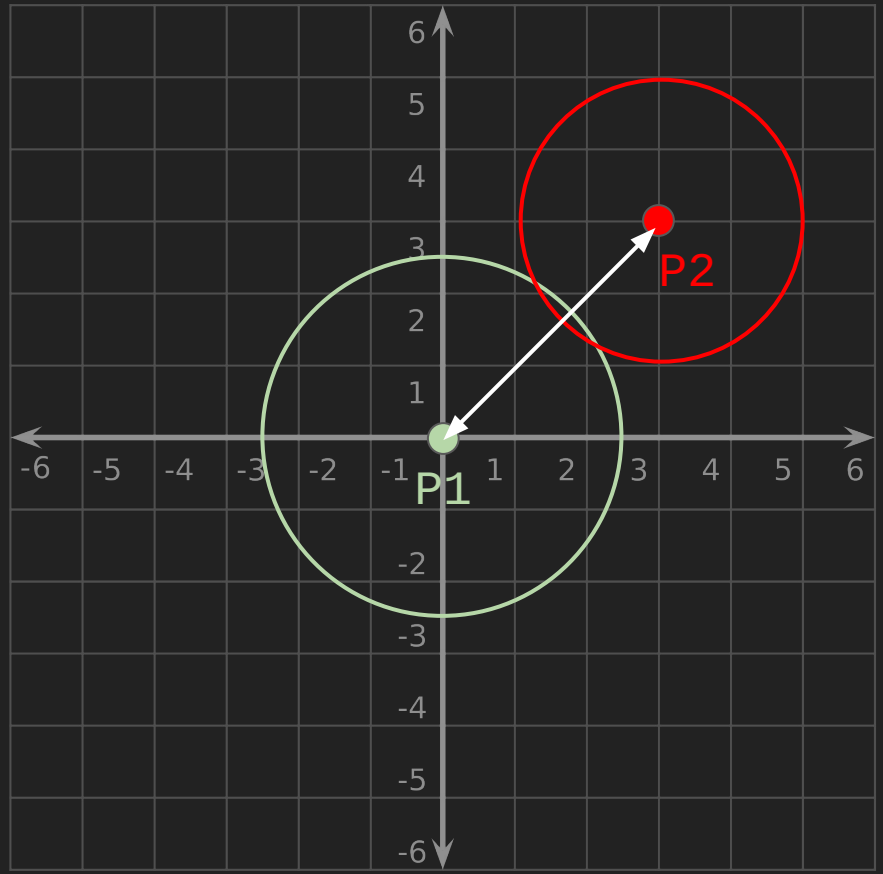
Radius: 3
P1: -2, -2

Radius: 2
P2: 3, 3

Radius + Radius = 5

Distance: 7.07
(Pythagorean Theorem)

distance > (radius + radius)



Radius: 3
P1: 0, 0

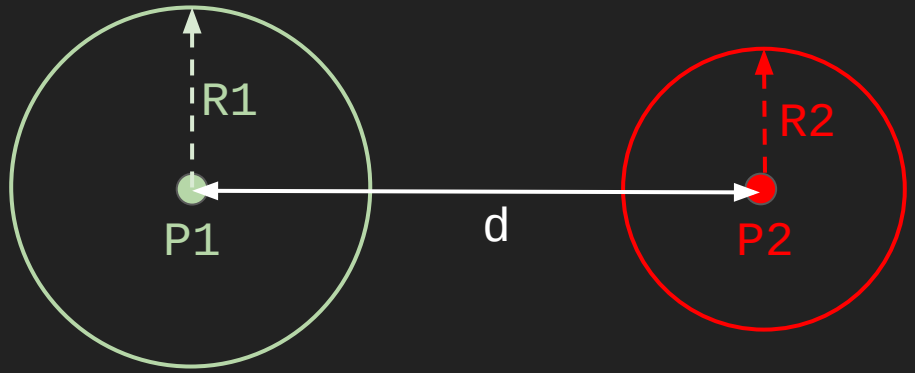
Radius: 2
P2: 3, 3

Radius + Radius = 5

Distance: 4.24
(Pythagorean Theorem)

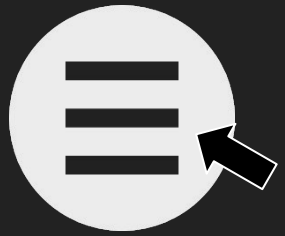
distance < (radius + radius)

Circle - Circle Collision Detection



If the distance between the center of the circles is less than the sum of the radii, they are colliding.

Point - Circle Collision Detection



Click on UI, move the player, select a target.

Point - Circle Collision Detection

If the distance between the point and the circle center is less than the radius, they are colliding.

Box - Box
Collision Detection

The Legend of Zelda (NES)

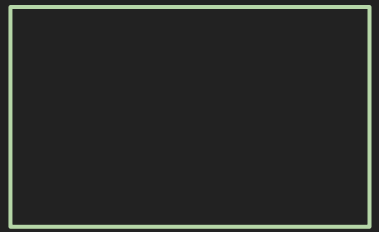
A Link to the Past (SNES)

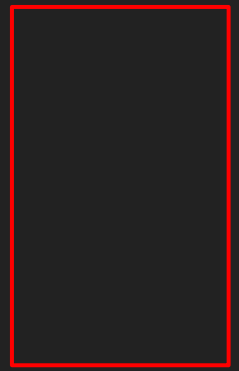

1P 127101 HI 127101 2P 1301
00100
00600
ZANGIEF
39
KEN

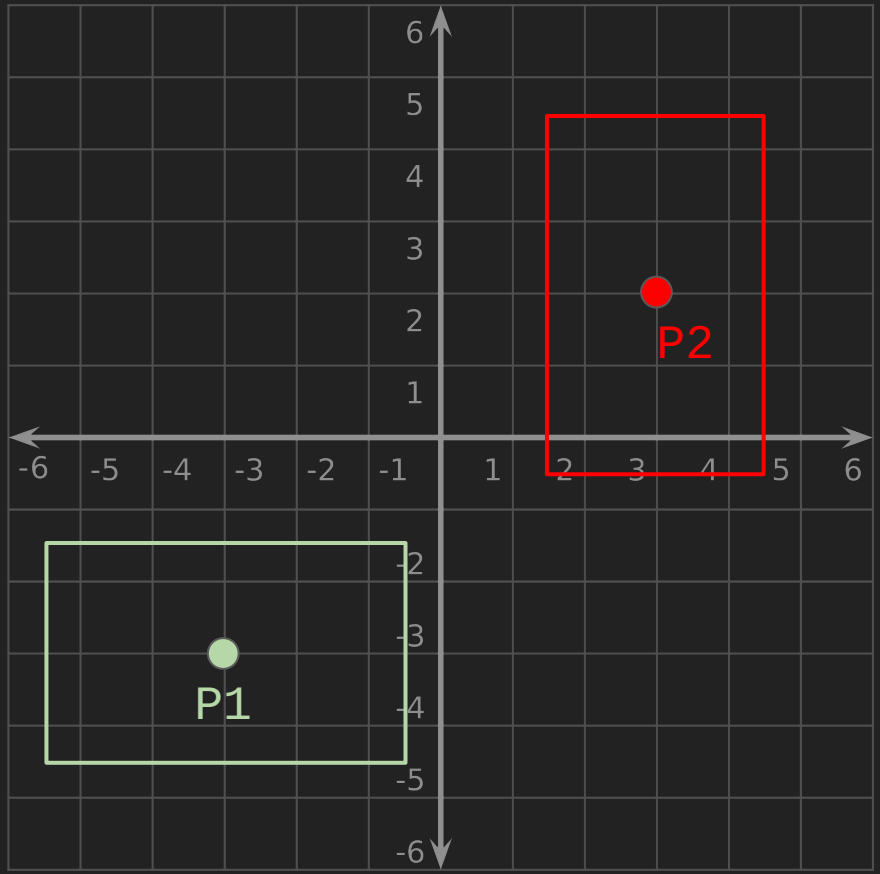

1P 1 POO 50000 2P 1
K.O
Blanka
77
E.Honda

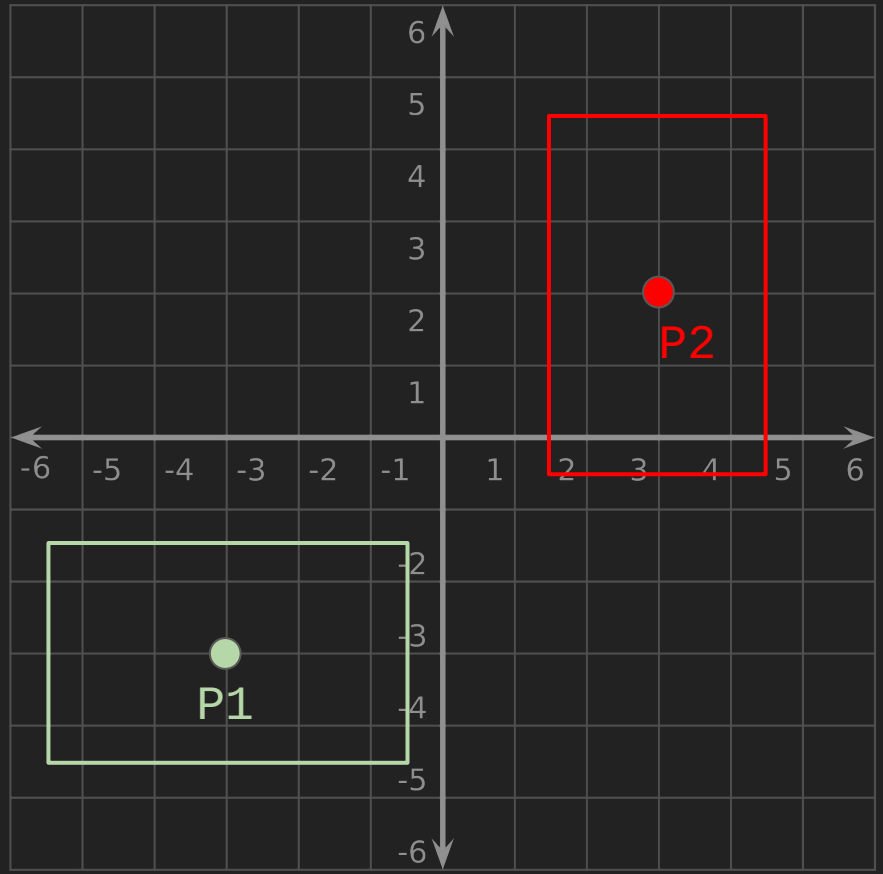

1P 1 POO 50000 2P 1
K.O
95
Sagat
ChunLi


0
98
0
SODOM
SAGAT
LEVEL
LEVEL



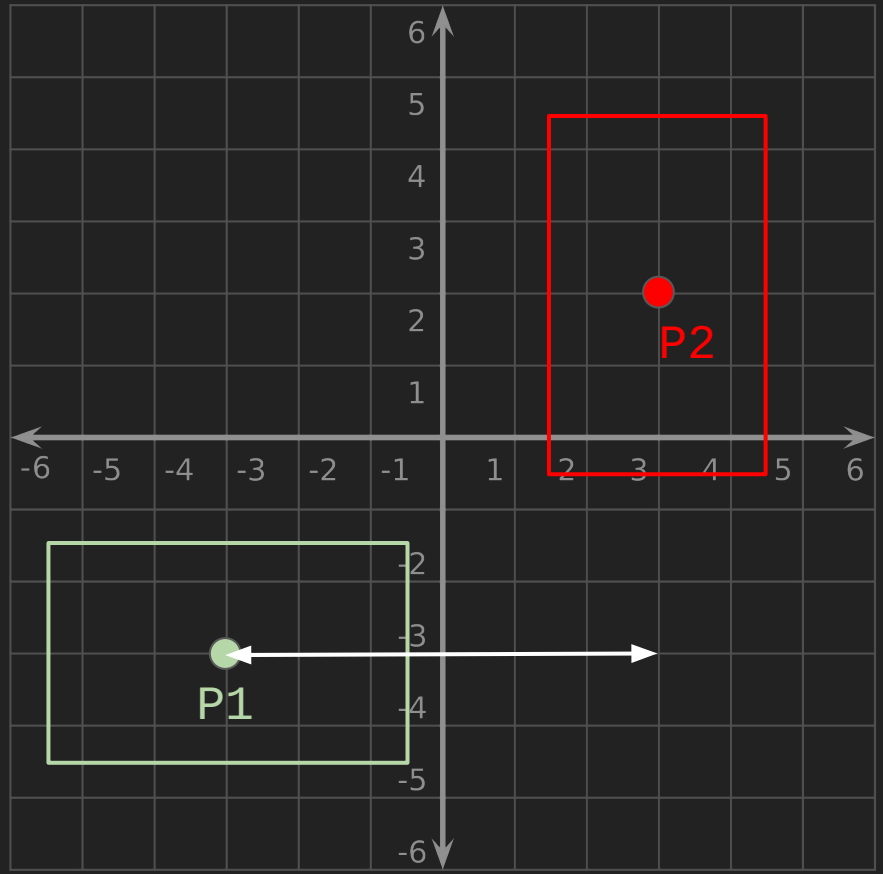



6
5
4
3
2
1
-6
-5
-4
-3
-2
-1
1
2
3
4
5
6
-2
-3
-4
-5
-6
P2
P1



Width: 5 Height: 3
P1: -3, -3

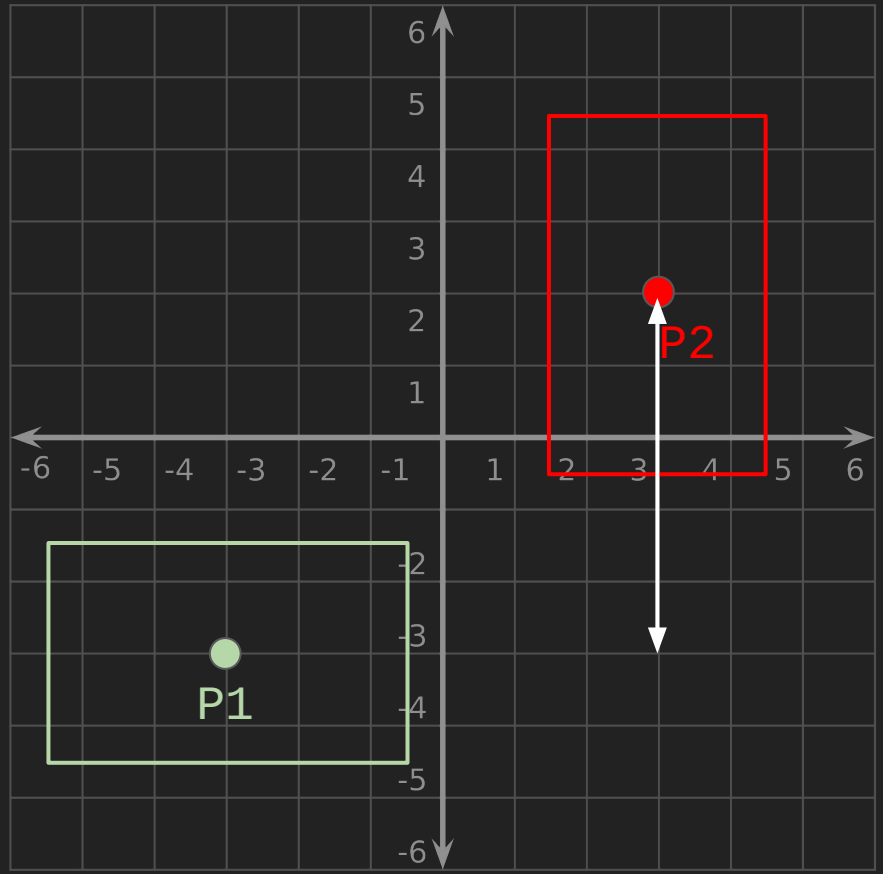
Width: 3 Height: 5
P2: 3, 2



Width: 5 Height: 3
P1: -3, -3

Width: 3 Height: 5
P2: 3, 2

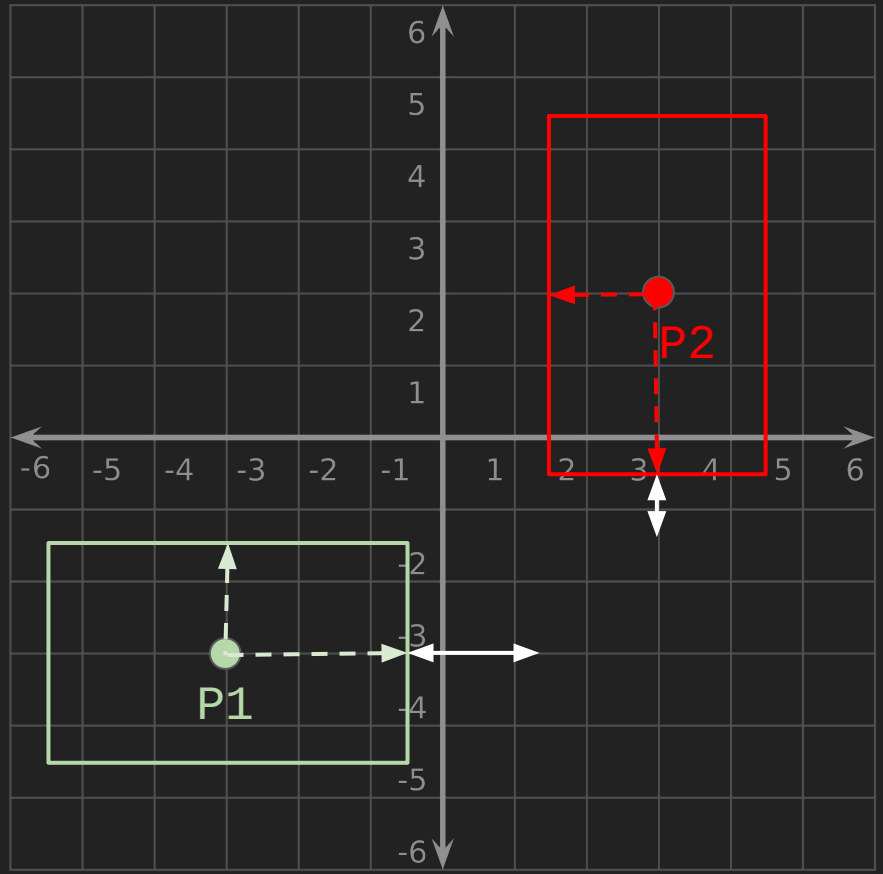
X Diff: 6 fabs(x2 - x1)



Width: 5 Height: 3
P1: -3, -3

Width: 3 Height: 5
P2: 3, 2

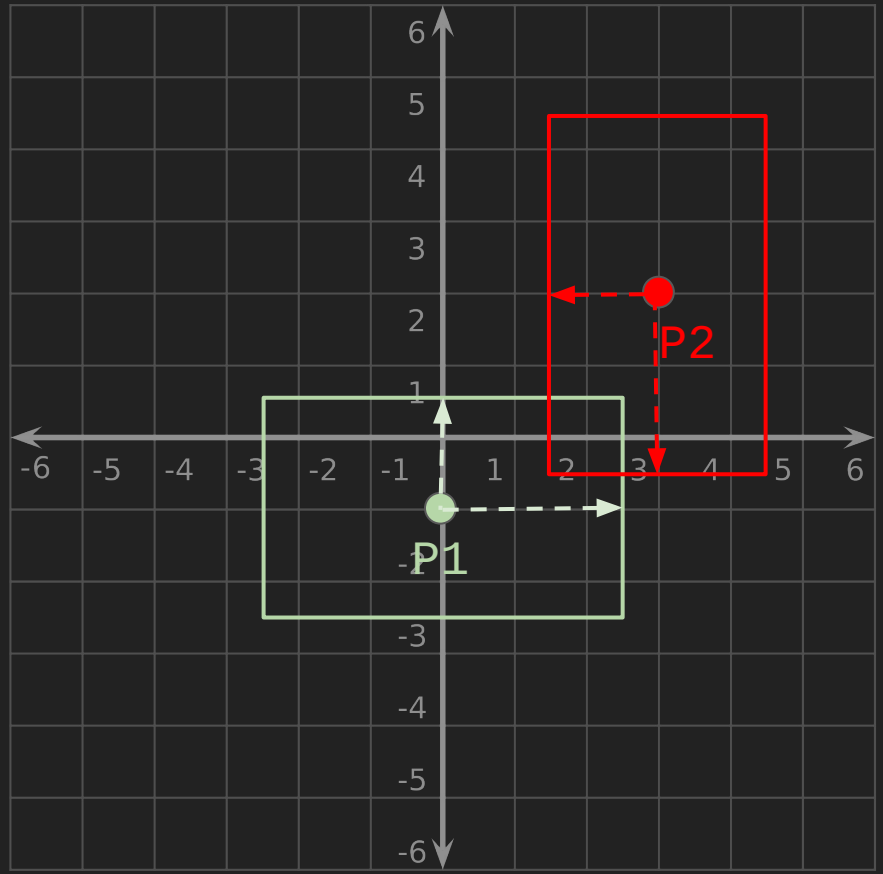
X Diff: 6 fabs(x2 - x1)
Y Diff: 5 fabs(y2 - y1)




```
Width: 5          Height: 3
P1: -3, -3

Width: 3          Height: 5
P2: 3, 2

X Diff: 6         fabs(x2 - x1)
Y Diff: 5         fabs(y2 - y1)
```

X Distance = XDiff - (W1 + W2) / 2
Y Distance = YDiff - (H1 + H2) / 2

X Distance = 2, Y Distance = 1
(both need to be < 0 to be colliding)

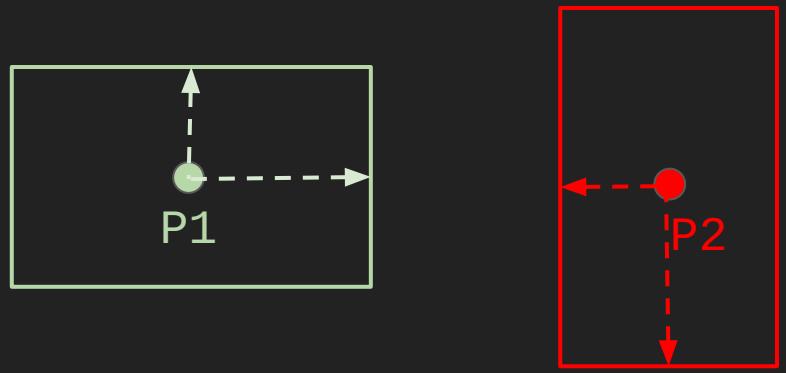



```
Width: 5          Height: 3
P1: 0, -1

Width: 3          Height: 5
P2: 3, 2

X Diff: 3         fabs(x2 - x1)
Y Diff: 3         fabs(y2 - y1)
```

X Distance = XDiff - (W1 + W2) / 2
Y Distance = YDiff - (H1 + H2) / 2

X Distance = -1, Y Distance = -1
(both are < 0 = colliding)

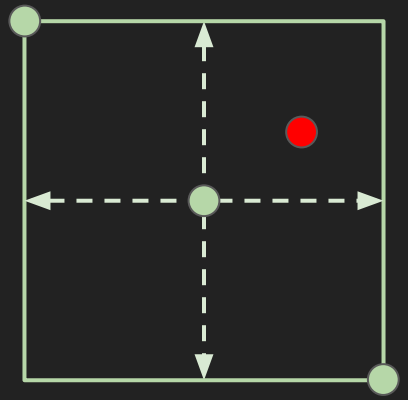
Box - Box Collision Detection




```
float xdist = fabs(x2 - x1) - ((w1 + w2) / 2.0f);
float ydist = fabs(y2 - y1) - ((h1 + h2) / 2.0f);

if (xdist < 0 && ydist < 0) // Colliding!
```

Point - Box Collision Detection

Point - Box Collision Detection



Get the top left and bottom right corners.
Check if the X,Y of the point is inside.

The Game Loop

ProcessInput()
- player.movement = ...

Update()
- Calculate deltaTime
- Test (Collision Detection) / Apply movement
- Update each object

Render()
- Clear the screen
- Draw everything

You’re ready to make your first game!