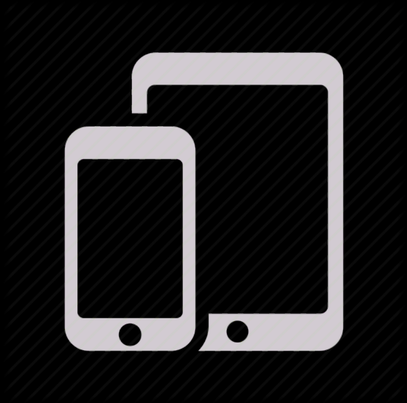
From app to game development

Michel-André Chirita

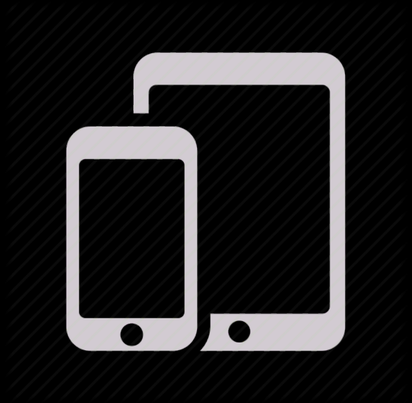
FrenchKit 2022

The goal



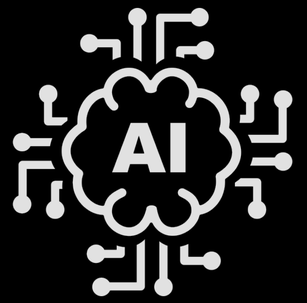
iPhone iPad

The goal



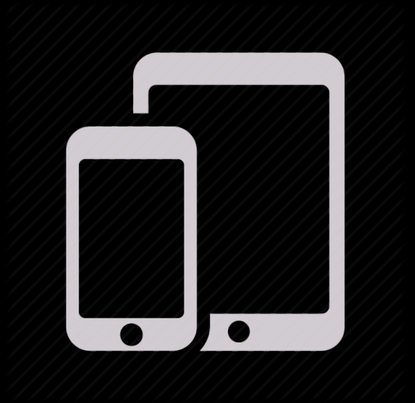
iPhone iPad

+



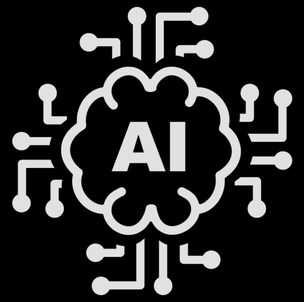
AI

The goal



iPhone iPad

+

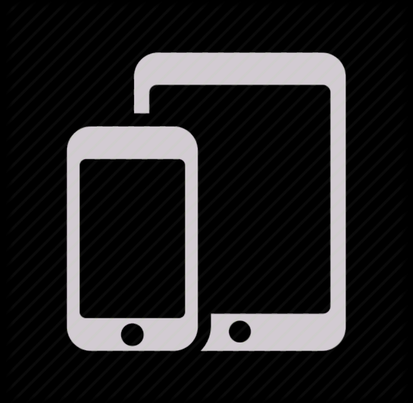


AI

+

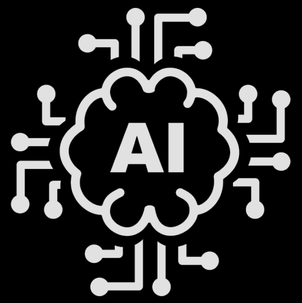
Realtime online multiplayer

The goal



iPhone iPad

+



AI

+

Realtime online multiplayer

+

No backend

The goal

iPhone iPad

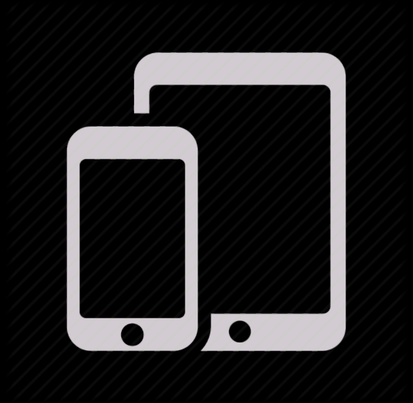
AI

Realtime online multiplayer

No backend

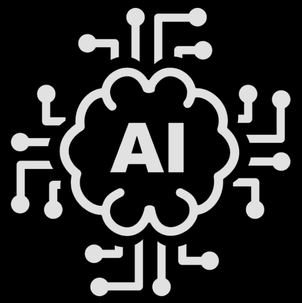
1 hand

The goal



iPhone iPad

+



AI

+

Realtime online multiplayer

+

No backend

Only with

Swift + SpriteKit + GameKit + GameplayKit

?

The result

Learnings

Learnings

Easy things that are not

Hard things that are not

Architecture

Artificial intelligence

Online multiplayer

Hero

Hero (Ndèle)

vs

Hero (Ndèle)

The other guy

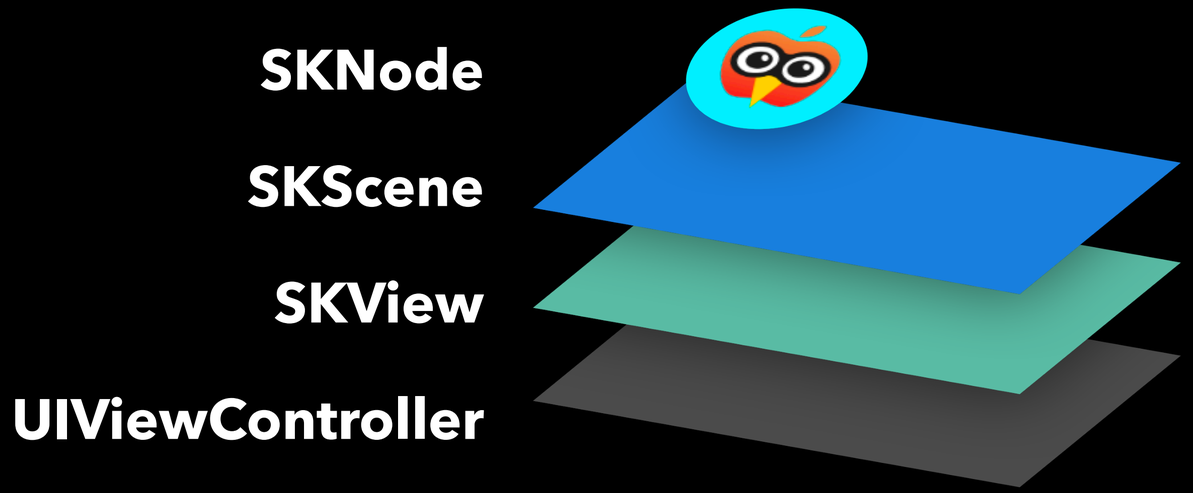
Learning 1

Easy things that are not

Adapt to screen size

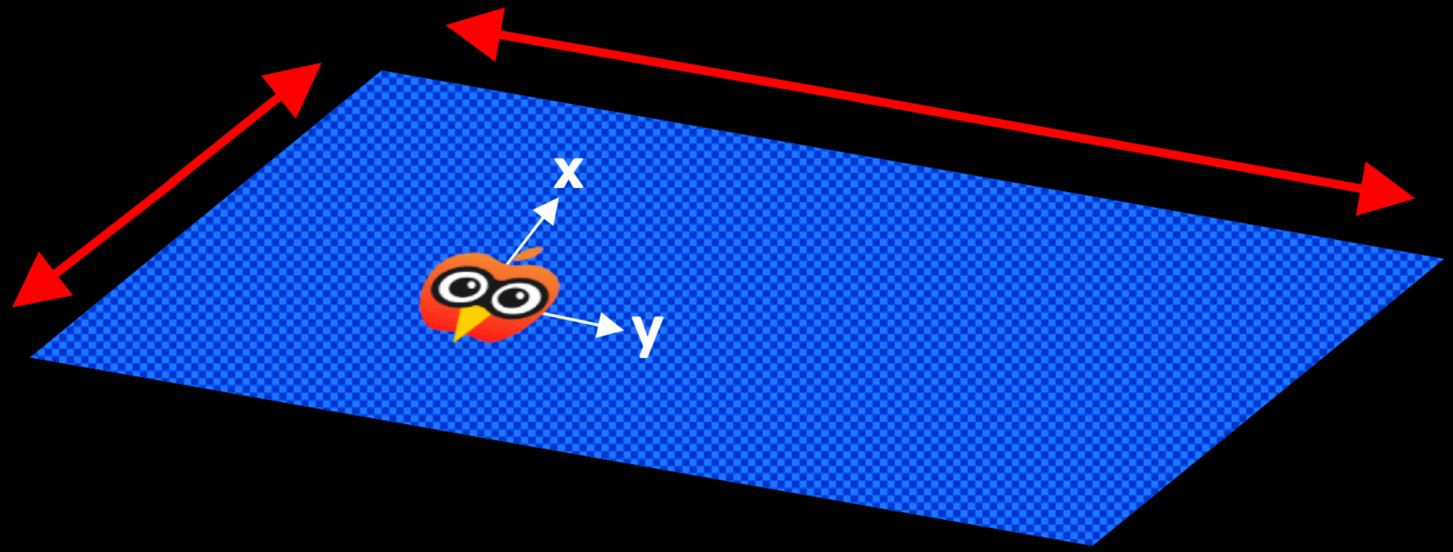
Learning 1

Adapt to screen size



Learning 1

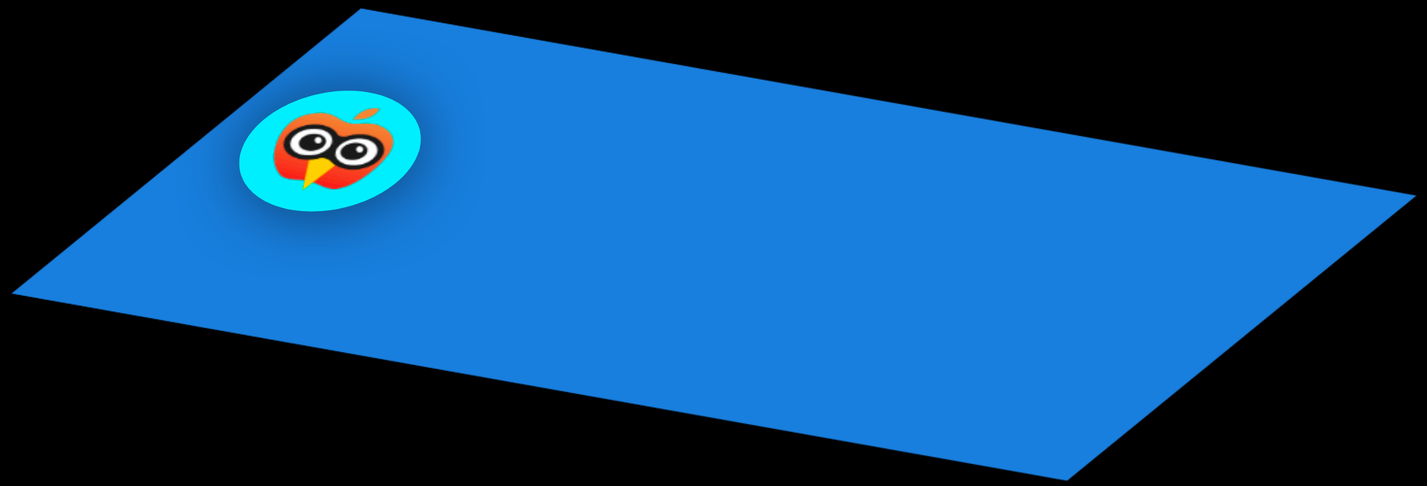
Adapt to screen size



Drag & drop

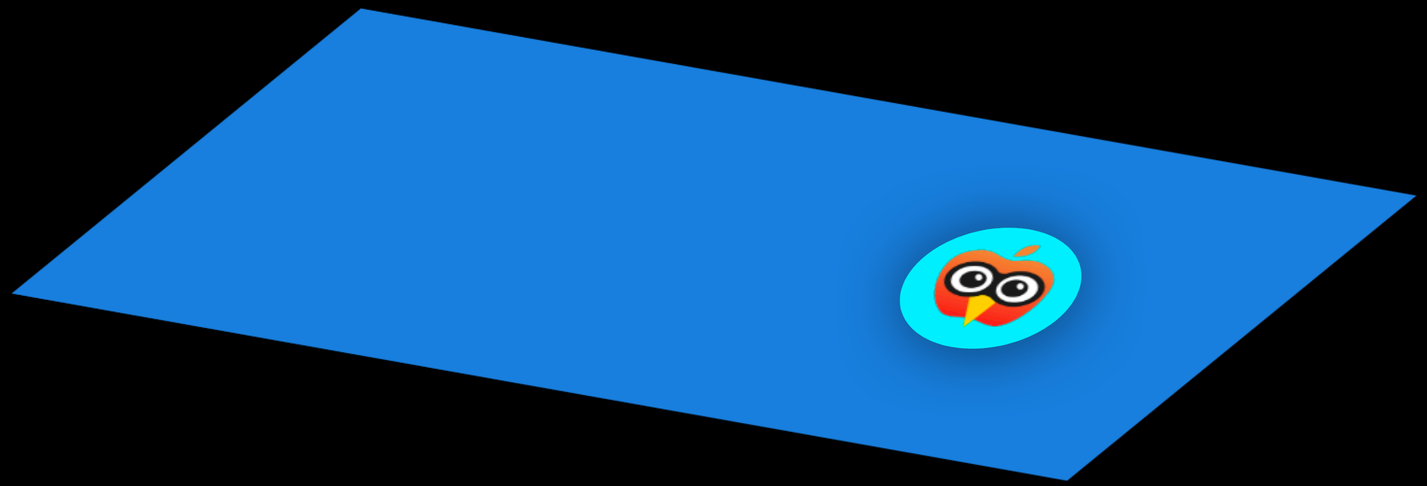
Learning 1

Drag & drop



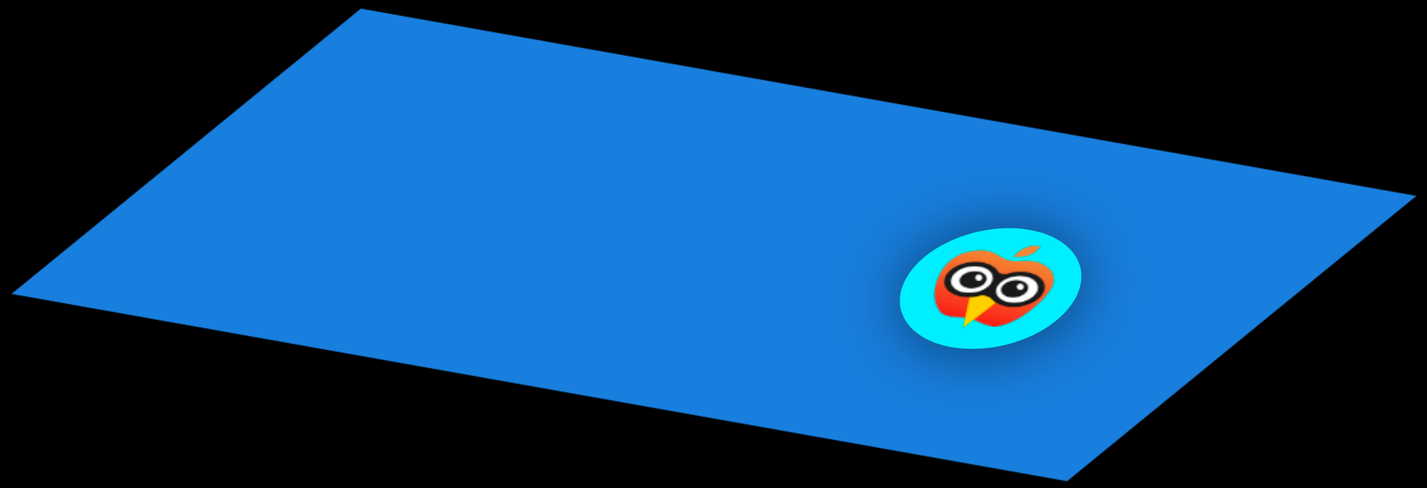
Learning 1

Drag & drop



Learning 1

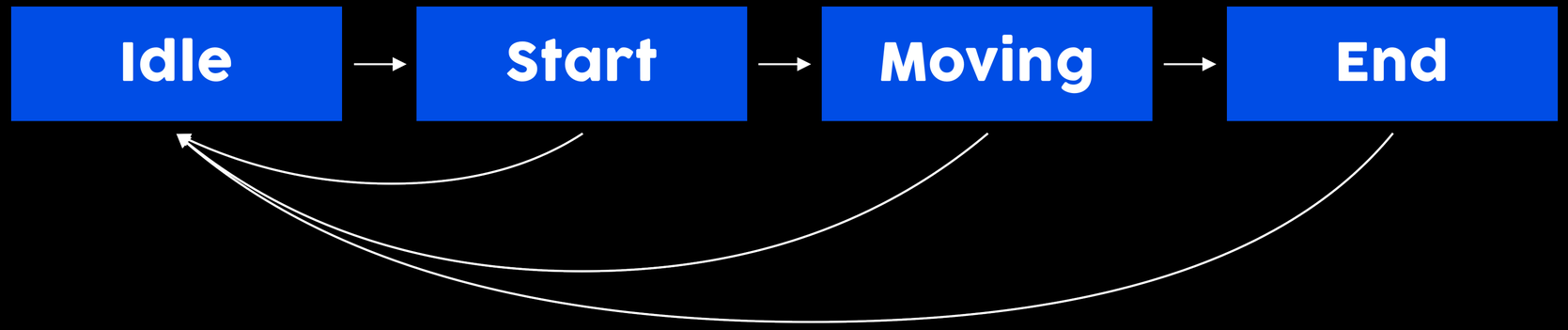
Drag & drop



```
func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) { /*...*/ }
func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) { /*...*/ }
func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) { /*...*/ }
func touchesCancelled(_ touches: Set<UITouch>, with event: UIEvent?) { /*...*/ }
```

Learning 1

Drag & drop

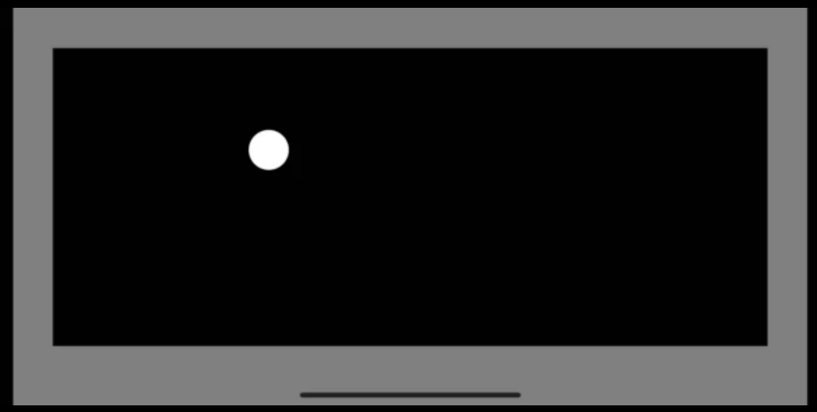



```
var dragStateMachine = GKStateMachine(states: [IdleDragState(),
                                               StartDragState(),
                                               MovingDragState(),
                                               EndDragState()])
```

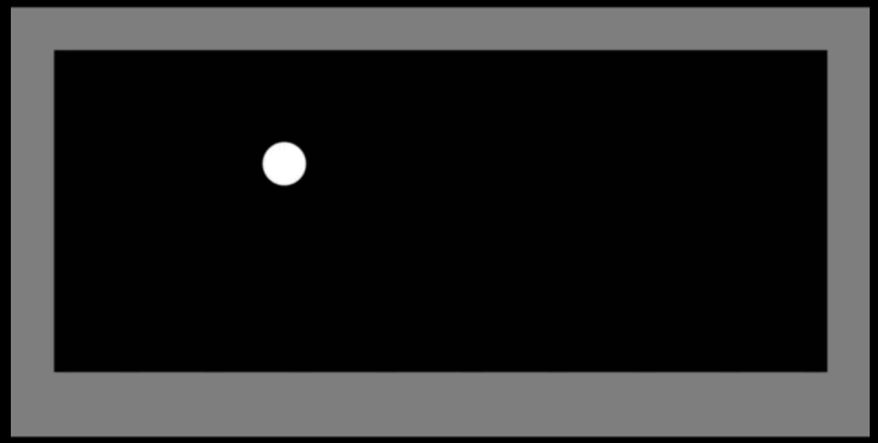
Bouncing ball

Learning 1

Bouncing ball



Device 1

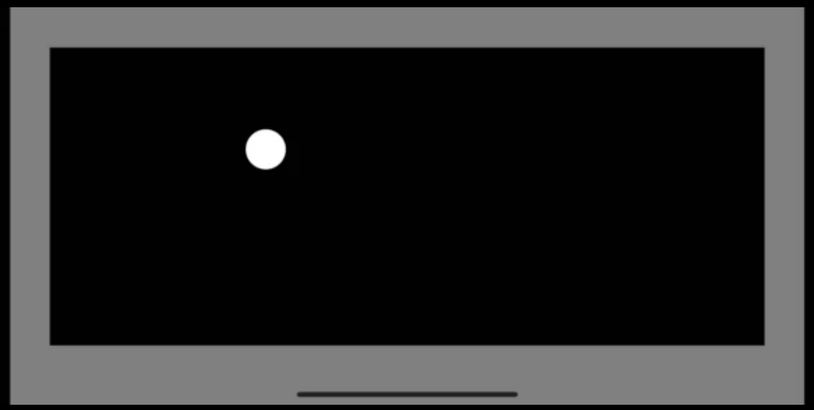


Device 2

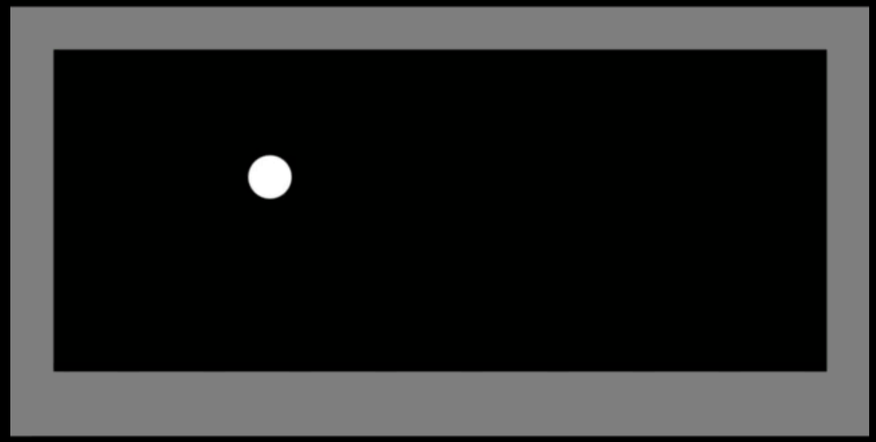
What’s the difference ?

Learning 1

Bouncing ball



Device 1



Device 2

⚠️ **scale affects the mass**

Position

+

Size

+

Scale

+

Physics parameters

=

Deterministic physics

physicsBody.usesPreciseCollisionDetection if needed

Learning 1

Takeaways

- SpriteKit doesn’t include all building blocks
- Deal with screen sizes from the beginning
- Use a State Machine for Drag & Drop
- Physics engine has deterministic results

Learning 2

Hard things that are not

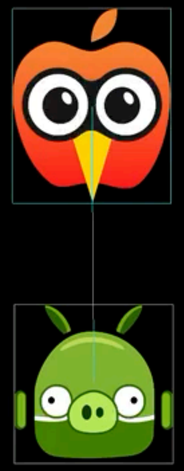
Learning 2

SKAction

```
node.run(.sequence([
  .fadeIn(withDuration: 2),
  .group([ .move(to: position, duration: 2), .rotate(byAngle: angle, duration: 2) ]),
  .scale(to: factor, duration: 1),
  .wait(forDuration: 1),
  .fadeOut(withDuration: 1)
]))
```

Learning 2

Physics



```
let spring = SKPhysicsJointSpring.joint(withBodyA: nodeA.physicsBody!,
                                        bodyB: nodeB.physicsBody!,
                                        anchorA: nodeA.position,
                                        anchorB: nodeB.position)
 spring.frequency = 0.8
 spring.damping = 0.1
 scene.physicsWorld.add(spring)
```

Learning 2

Matchmaking

GKMatchmakerViewController

OR

```
let request = GKMatchRequest()
request.minPlayers = 2
request.maxPlayers = 2

GKMatchmaker.shared().findMatch(for: request, withCompletionHandler: { match, error in
  /*...*/
})
```

Learning 2

Many building blocks

Entities

Components

Role composition

GKRandom

Controlled random value
(source, distribution)

...

Agent

Goals

Behaviors

Autonomous
movement

Graph

Path

Obstacles

Path finding

Learning 2

Takeaways

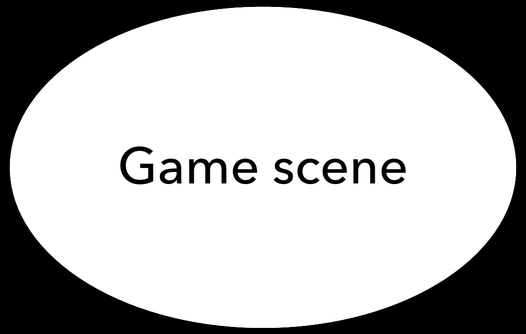
- Nice building blocks
- Most are impressively easy to implement
- Explore documentation before to re-implement something by yourself

Learning 3

Architecture

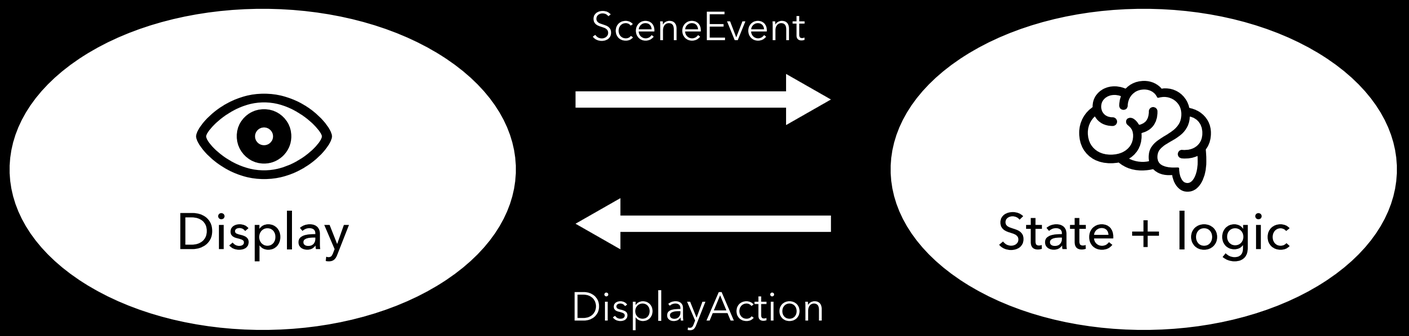
Learning 3

Overview



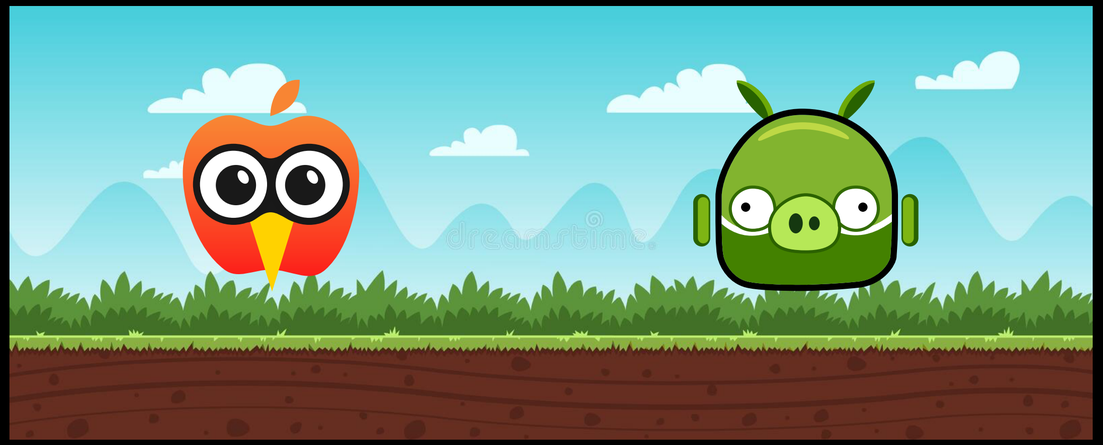
Learning 3

Overview



Learning 3

State



```
struct GameState {
  var gameTime: TimeInterval
  var score: Int
  var heroPosition: CGPoint
  var opponentPosition: CGPoint
}
```

Learning 3

Scene events

```
enum SceneEvent {
  case heroDidMove(to: CGPoint)
  case opponentDidMove(to: CGPoint)
}
```

Learning 3

Logic: state & reducer

```
enum SceneEvent {
  case heroDidMove(to: CGPoint)
  case opponentDidMove(to: CGPoint)
}
```

```
func reduce(event: SceneEvent) {
    switch event {



}
```

Learning 3

Logic: state & reducer

```
enum SceneEvent {
  case heroDidMove(to: CGPoint)
  case opponentDidMove(to: CGPoint)
}
```

```
func reduce(event: SceneEvent) {
    switch event {
    case .heroDidMove(to: let position):
      state.heroPosition = position
     gameUI.display(.moveHero(to: position))



}
```

Learning 3

Logic: state & reducer

```
enum SceneEvent {
  case heroDidMove(to: CGPoint)
  case opponentDidMove(to: CGPoint)
}
```

```
func reduce(event: SceneEvent) {
    switch event {
    case .heroDidMove(to: let position):
      state.heroPosition = position
     gameUI.display(.moveHero(to: position))

    case .opponentDidMove(to: let position):
     […]
   }


}
```

Learning 3

Logic: state & reducer

```
enum SceneEvent {
  case heroDidMove(to: CGPoint)
  case opponentDidMove(to: CGPoint)
}
```

```
func reduce(event: SceneEvent) {
    switch event {
    case .heroDidMove(to: let position):
      state.heroPosition = position
     gameUI.display(.moveHero(to: position))

    case .opponentDidMove(to: let position):
     […]
   }

    if isVictory() {
      gameUI.display(.gameOver(winner: winner))
    }
  }
```

Learning 4

Display action

```
enum DisplayAction {
  case moveHero(to: CGPoint)
  case moveOpponent(to: CGPoint)
  case gameOver(winner: Winner)
}
```

Learning 3

Display action

```
enum DisplayAction {
  case moveHero(to: CGPoint)
  case moveOpponent(to: CGPoint)
  case gameOver(winner: Winner)
}
```

```
func display(_ action: DisplayAction) {
  switch action {



  }
}
```

Learning 3

Display action

```
enum DisplayAction {
  case moveHero(to: CGPoint)
  case moveOpponent(to: CGPoint)
  case gameOver(winner: Winner)
}
```

```
func display(_ action: DisplayAction) {
  switch action {
  case .moveHero(let position):
    heroNode.run(.move(to: position, duration: 1.0))




  }
}
```

Learning 3

Display action

```
enum DisplayAction {
  case moveHero(to: CGPoint)
  case moveOpponent(to: CGPoint)
  case gameOver(winner: Winner)
}
```

```
func display(_ action: DisplayAction) {
  switch action {
  case .moveHero(let position):
    heroNode.run(.move(to: position, duration: 1.0))

  case .moveOpponent(let position):
    opponentNode.run(.move(to: position, duration: 1))


  }
}
```

Learning 3

Display action

```
enum DisplayAction {
  case moveHero(to: CGPoint)
  case moveOpponent(to: CGPoint)
  case gameOver(winner: Winner)
}
```

```
func display(_ action: DisplayAction) {
  switch action {
  case .moveHero(let position):
    heroNode.run(.move(to: position, duration: 1.0))

  case .moveOpponent(let position):
    opponentNode.run(.move(to: position, duration: 1))

  case .gameOver(let winner):
    /* ... */
  }
}
```

Learning 3

Takeaway

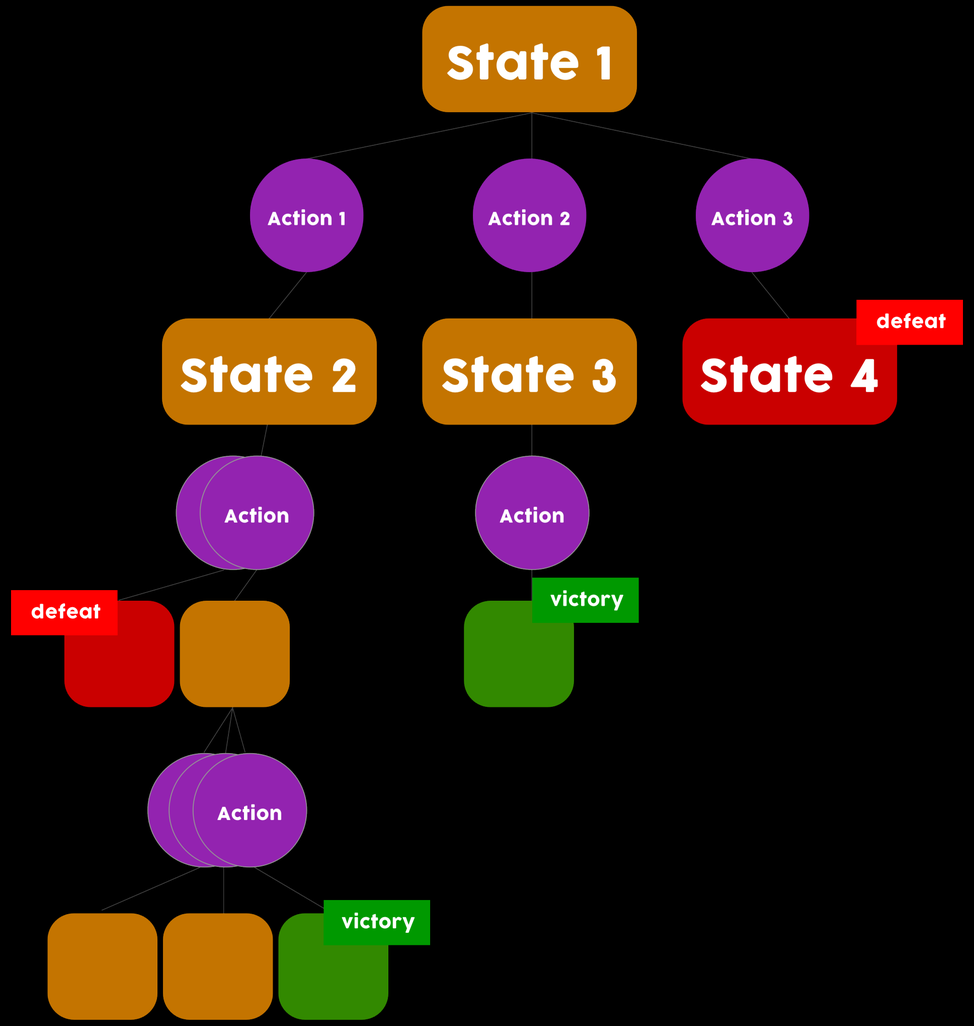
- Separate game logic & display
- Use a global state and enums for defining all interactions
- More readability
- Improve testability
- Helps integrate AI & online multiplayer

Learning 4

Artificial intelligence

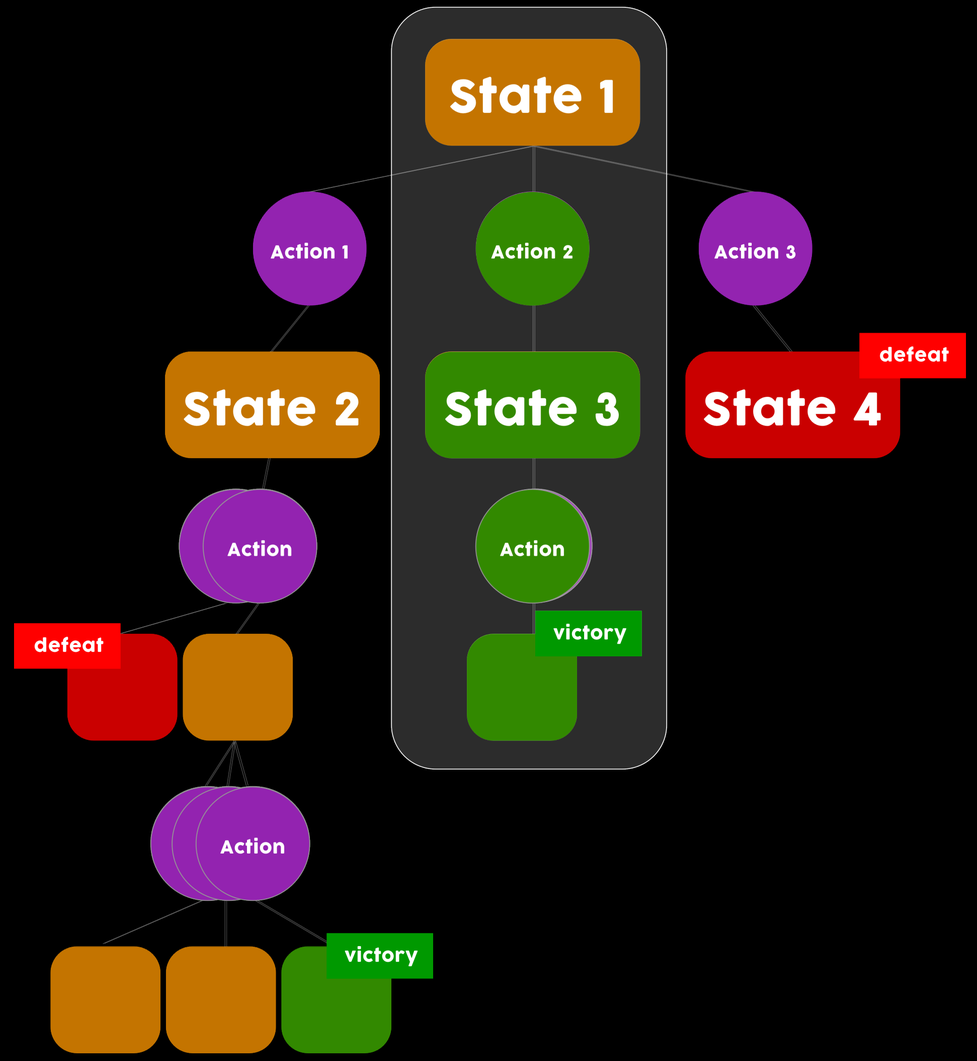
Learning 4

AI decision tree



Learning 4

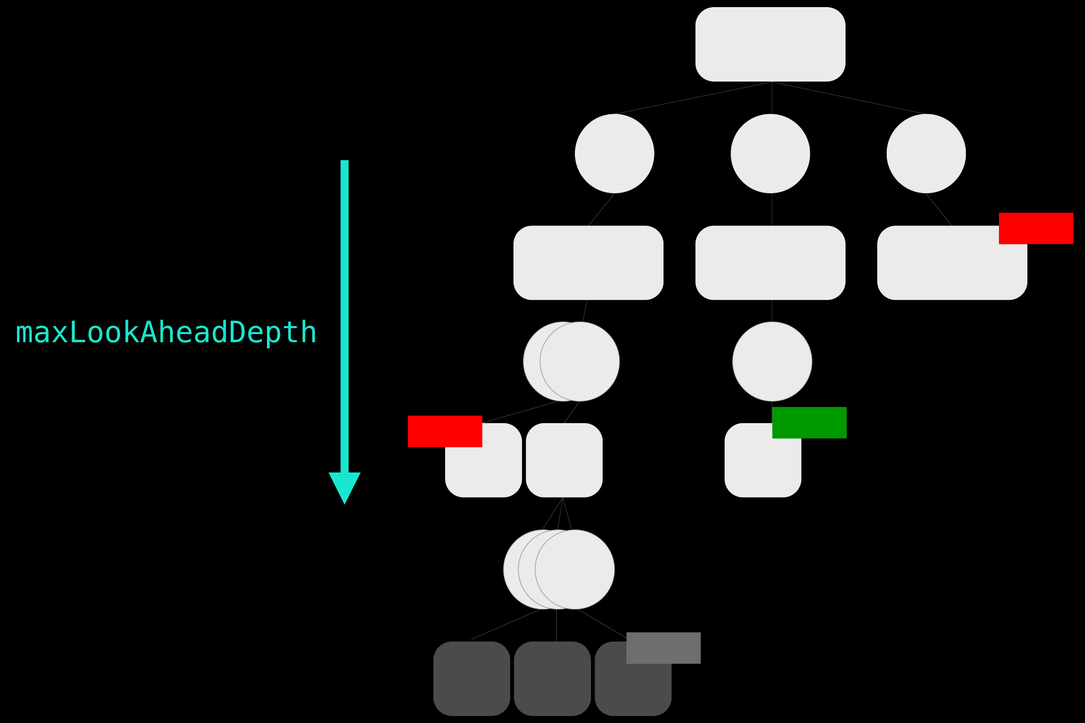
AI decision tree



Learning 4

Strategists

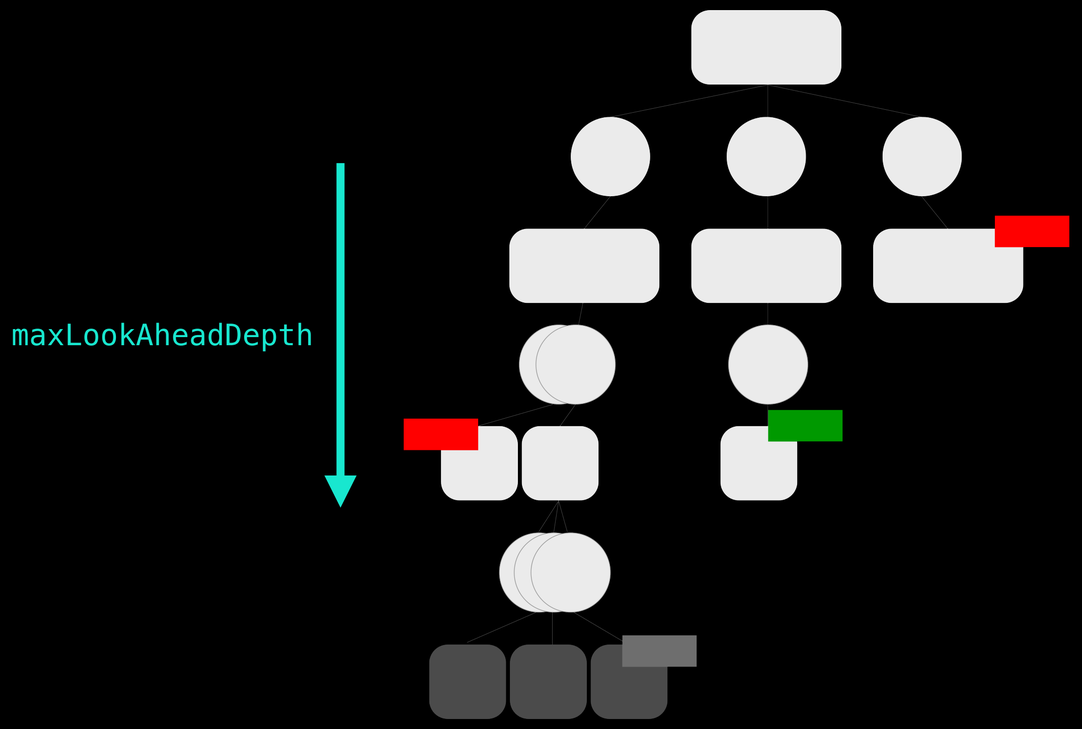
MinMax



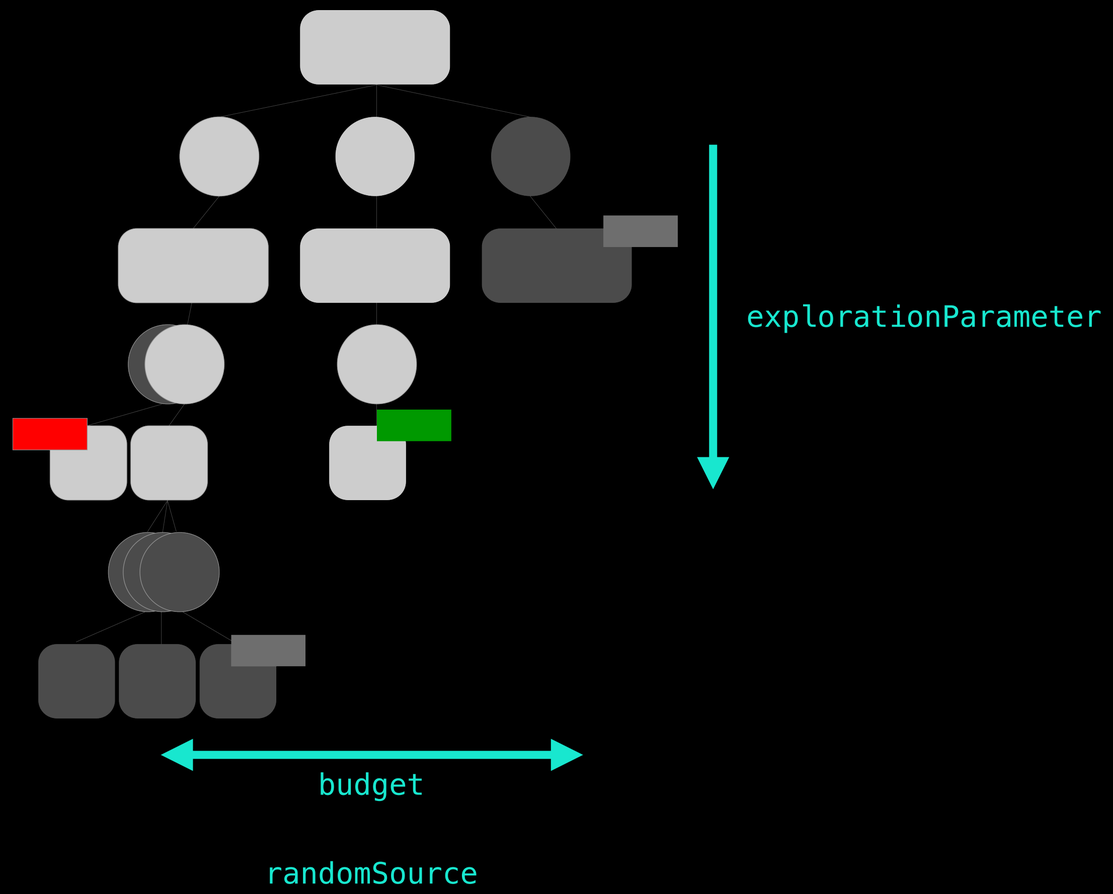
Learning 4

Strategists

MinMax



MonteCarlo



Slower

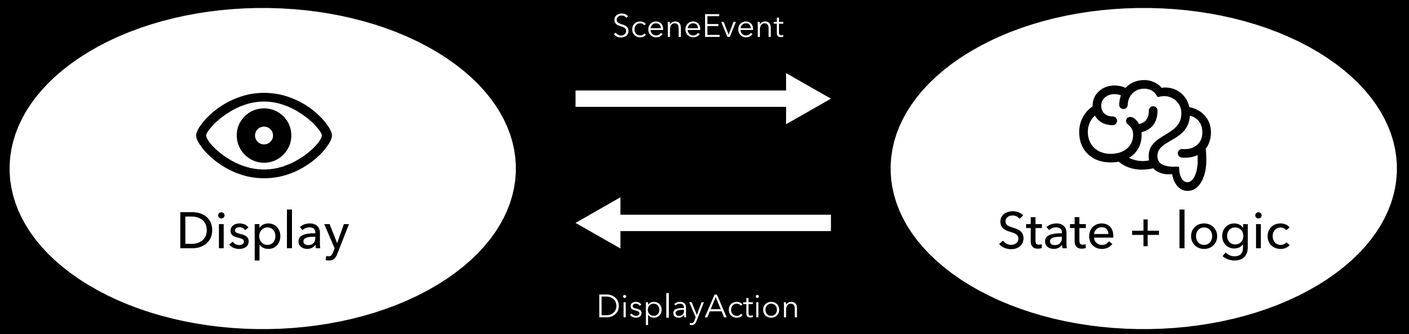
Best result

Faster

Less good results

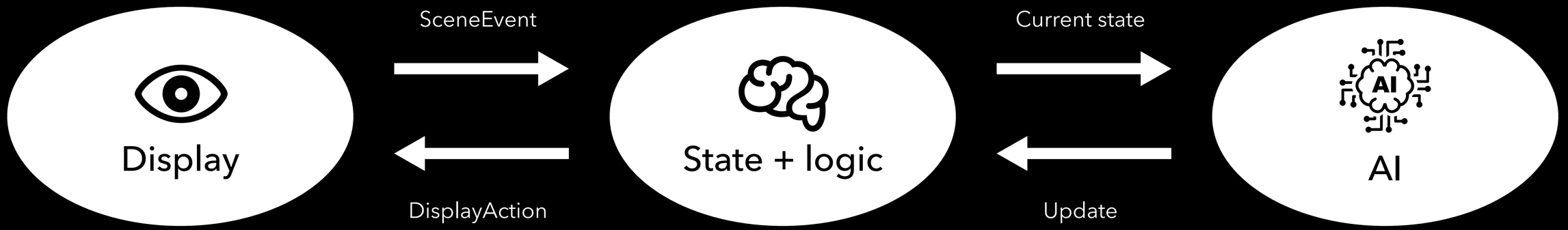
Learning 4

Architecture



Learning 4

Architecture



Learning 4

Game model

```
class State: NSObject, GKGameModel {


}
```

Learning 4

Game model

```
class State: NSObject, GKGameModel {

  // 1: Provide players and who are playing next
   var players: [GKGameModelPlayer]?
   var activePlayer: GKGameModelPlayer?
}
```

Learning 4

Game model

```
class State: NSObject, GKGameModel {

  // 1: Provide players and who are playing next
  var players: [GKGameModelPlayer]?
  var activePlayer: GKGameModelPlayer?

  // 2: Calculate score for player and if there is a winner
  func score(for player: GKGameModelPlayer) -> Int { /* ... */ }
  func isWin(for player: GKGameModelPlayer) -> Bool { /* ... */ }
}
```

Learning 4

Game model

```
class State: NSObject, GKGameModel {

  // 1: Provide players and who are playing next
   var players: [GKGameModelPlayer]?
   var activePlayer: GKGameModelPlayer?

   // 2: Calculate score for player and if there is a winner
  func score(for player: GKGameModelPlayer) -> Int { /* ... */ }
   func isWin(for player: GKGameModelPlayer) -> Bool { /* ... */ }

  // 3: Provide all possible updates for this state
   func gameModelUpdates(for player: GKGameModelPlayer) -> [GKGameModelUpdate]? { /*...*/ }

}
```

Learning 4

Game model

```
class State: NSObject, GKGameModel {

  // 1: Provide players and who are playing next
  var players: [GKGameModelPlayer]?
  var activePlayer: GKGameModelPlayer?

  // 2: Calculate score for player and if there is a winner
  func score(for player: GKGameModelPlayer) -> Int { /* ... */ }
  func isWin(for player: GKGameModelPlayer) -> Bool { /* ... */ }

  // 3: Provide all possible updates for this state
  func gameModelUpdates(for player: GKGameModelPlayer) -> [GKGameModelUpdate]? { /*...*/ }

  // 4: Duplicate the state
  func copy(with zone: NSZone? = nil) -> Any { /* ... */ }
  func setGameModel(_ gameModel: GKGameModel) { /* ... */ }
}
```

Learning 4

Game model

```
class State: NSObject, GKGameModel {

  // 1: Provide players and who are playing next
  var players: [GKGameModelPlayer]?
  var activePlayer: GKGameModelPlayer?

  // 2: Calculate score for player and if there is a winner
  func score(for player: GKGameModelPlayer) -> Int { /* ... */ }
  func isWin(for player: GKGameModelPlayer) -> Bool { /* ... */ }

  // 3: Provide all possible updates for this state
  func gameModelUpdates(for player: GKGameModelPlayer) -> [GKGameModelUpdate]? { /*...*/ }

  // 4: Duplicate the state
  func copy(with zone: NSZone? = nil) -> Any { /* ... */ }
  func setGameModel(_ gameModel: GKGameModel) { /* ... */ }

  // 5: Apply the given update
  func apply(_ gameModelUpdate: GKGameModelUpdate) { /* ... */ }
}
```

Learning 4

Engine

```
let strategist = GKMinmaxStrategist() // GKMonteCarloStrategist()

strategist.gameModel = state

strategist.bestMoveForActivePlayer()
```

Learning 4

Takeaway

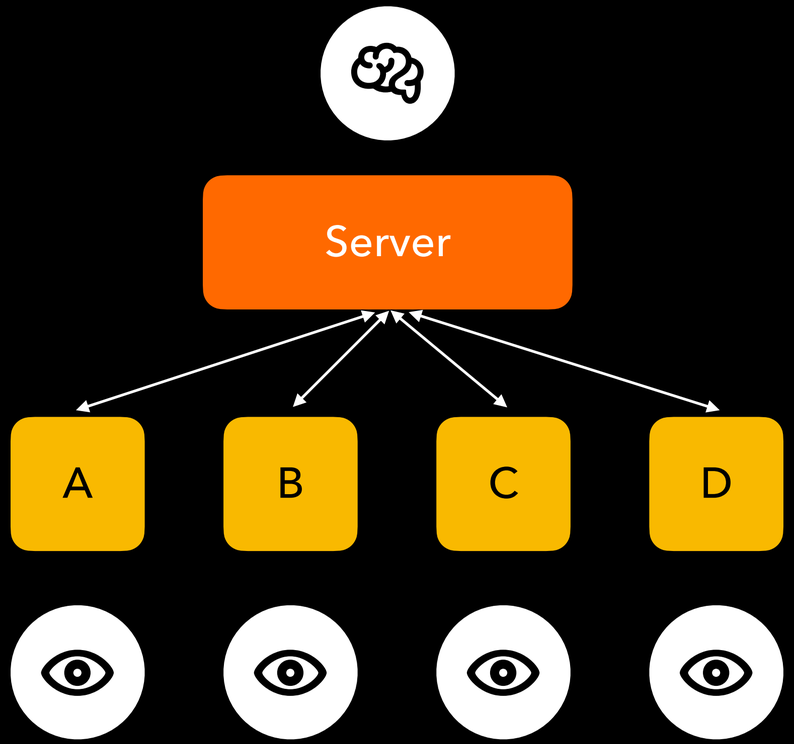
- Super easy to integrate, with the right architecture
- Fast enough for realtime gaming …
- … but choose wisely when to run it
- GKMinmaxStrategist when few possible actions for each state
- else use GKMonteCarloStrategist

Learning 5

Realtime online multiplayer

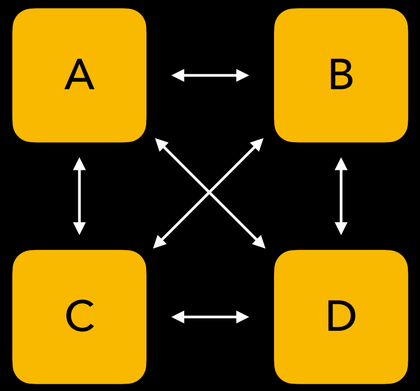
Multiplayer strategies

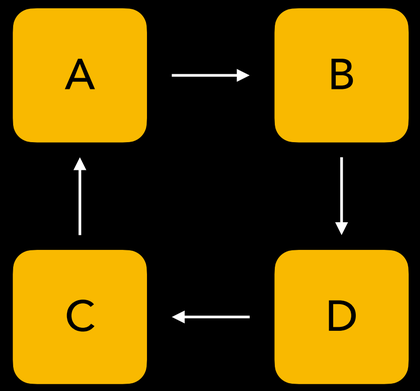
Server hosted



Multiplayer strategies

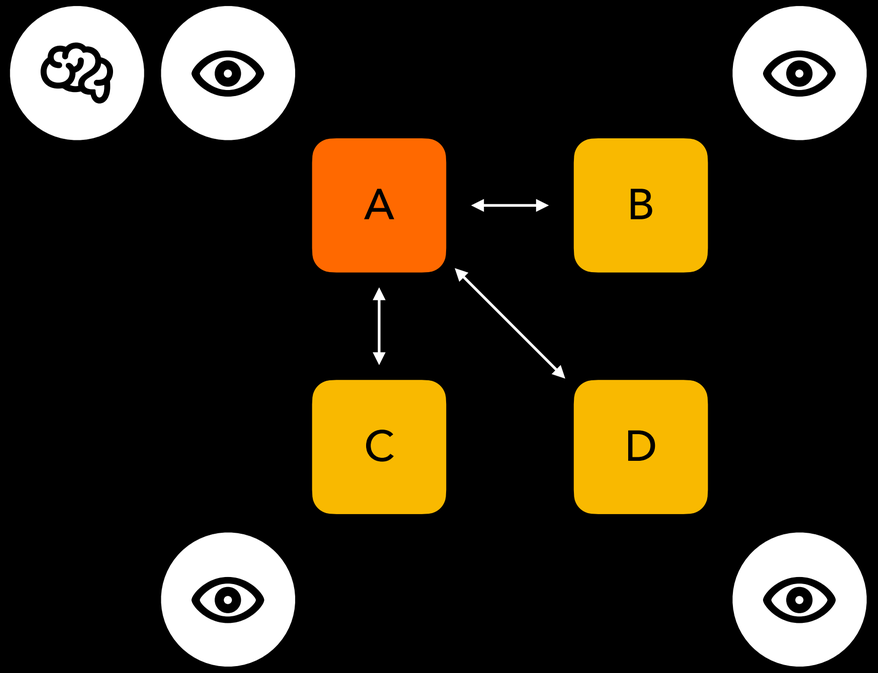
Direct P2P - Ring P2P





Multiplayer strategies

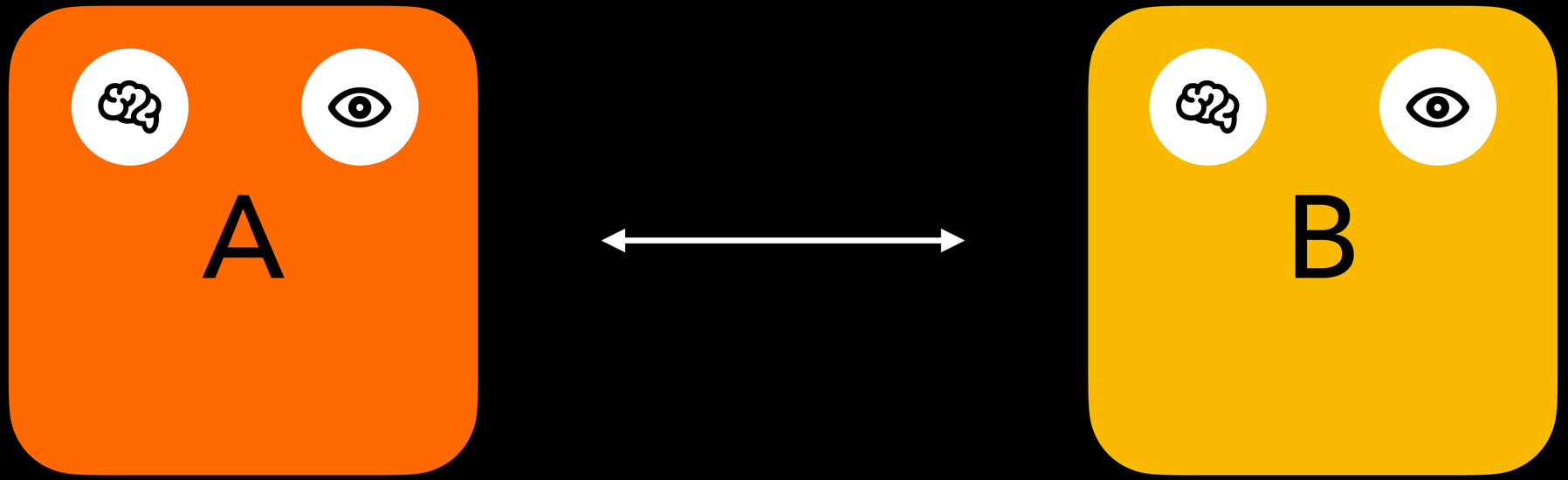
Player-hosted client / server



Multiplayer strategies

Hybrid player-hosted client / server

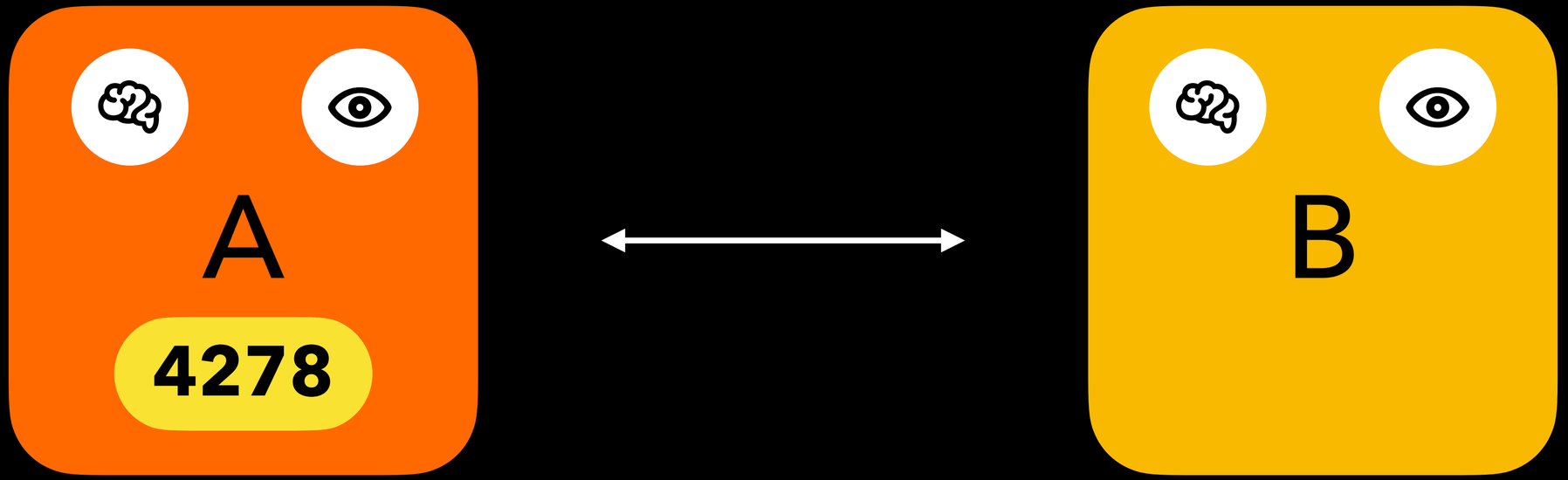
Local logic processing



Multiplayer strategies

Hybrid player-hosted client / server

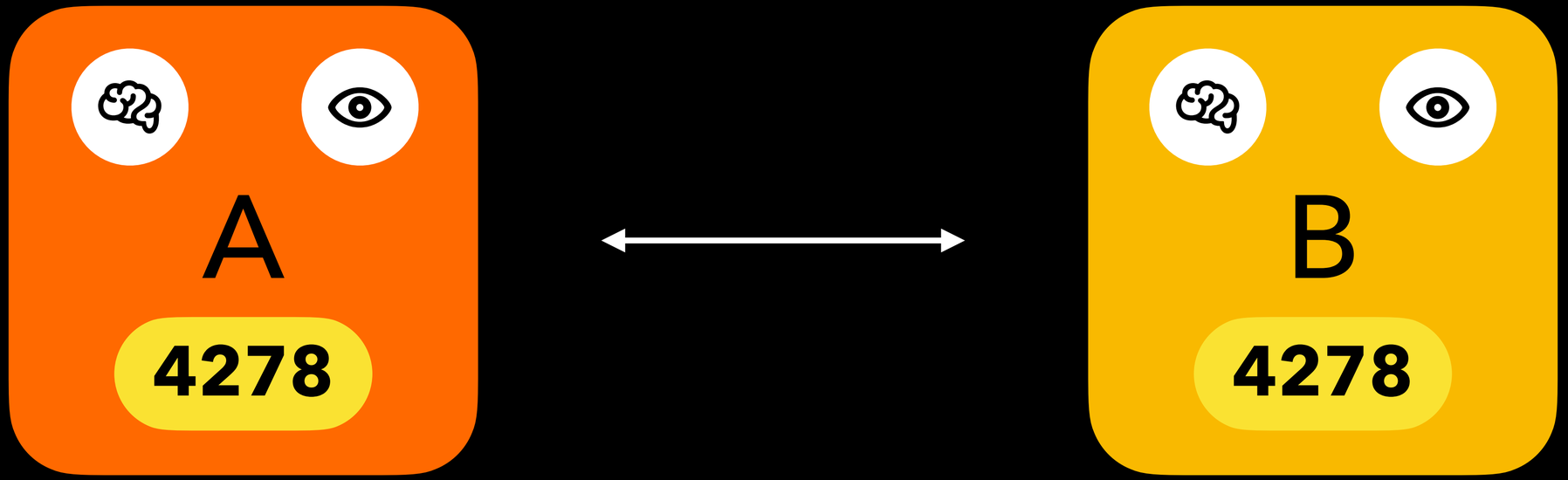
Local logic processing



Multiplayer strategies

Hybrid player-hosted client / server

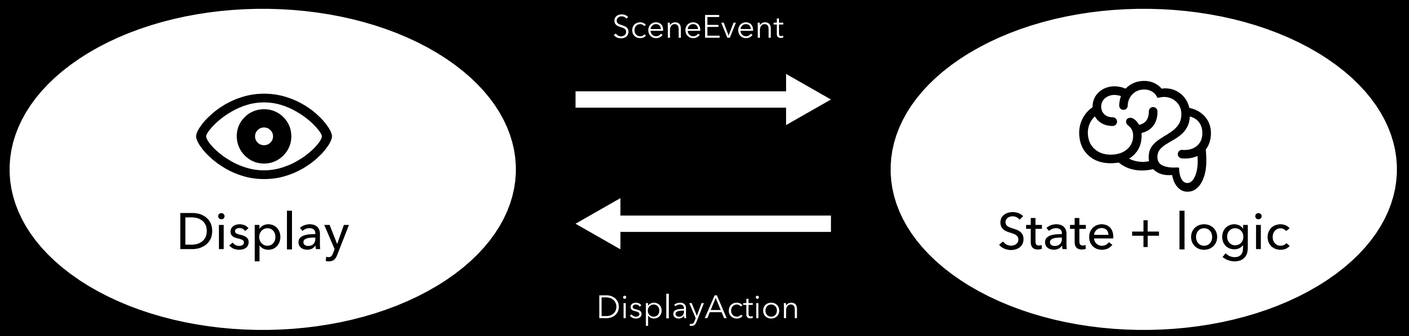
Local logic processing



Single source of truth

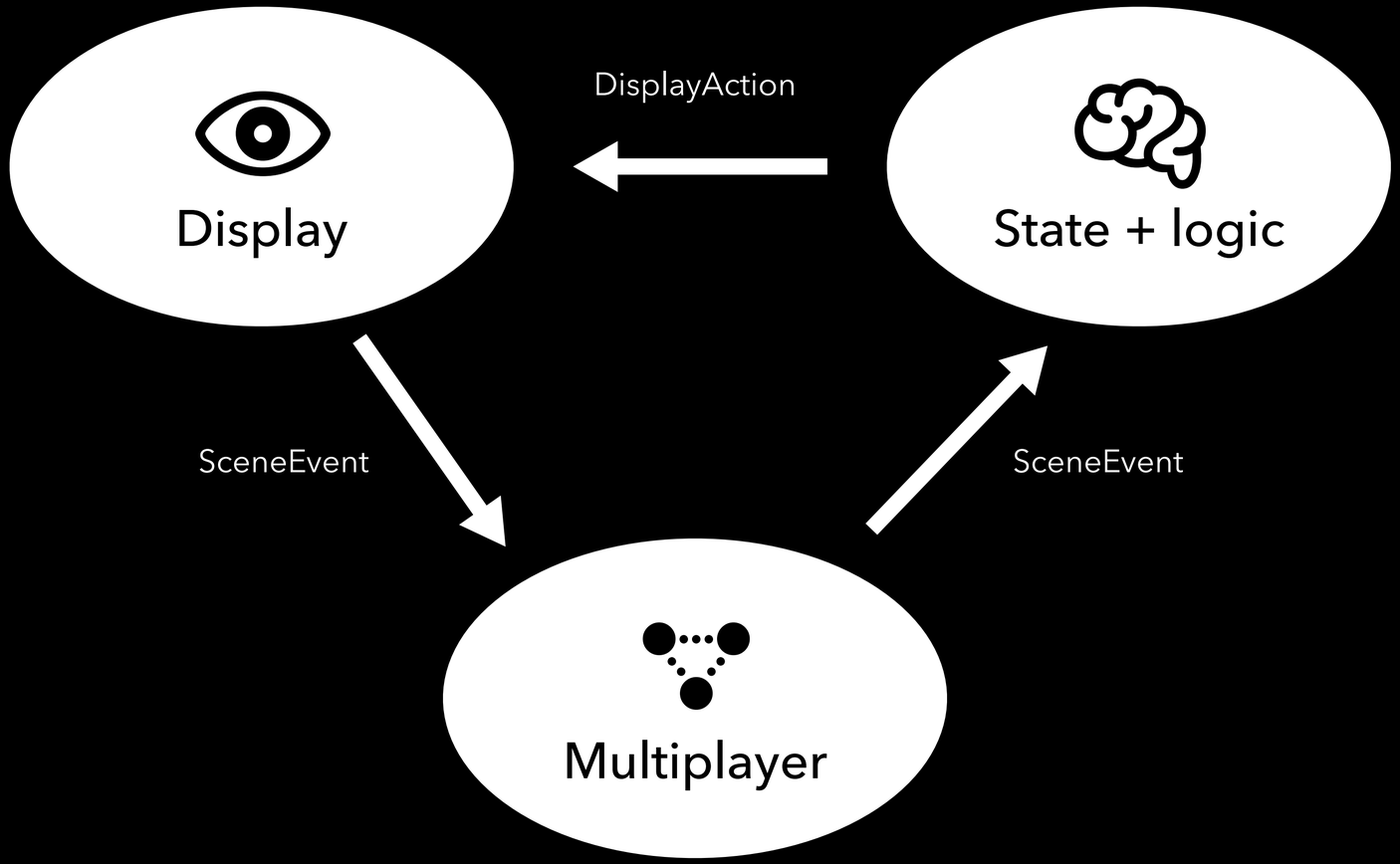
Multiplayer

Architecture



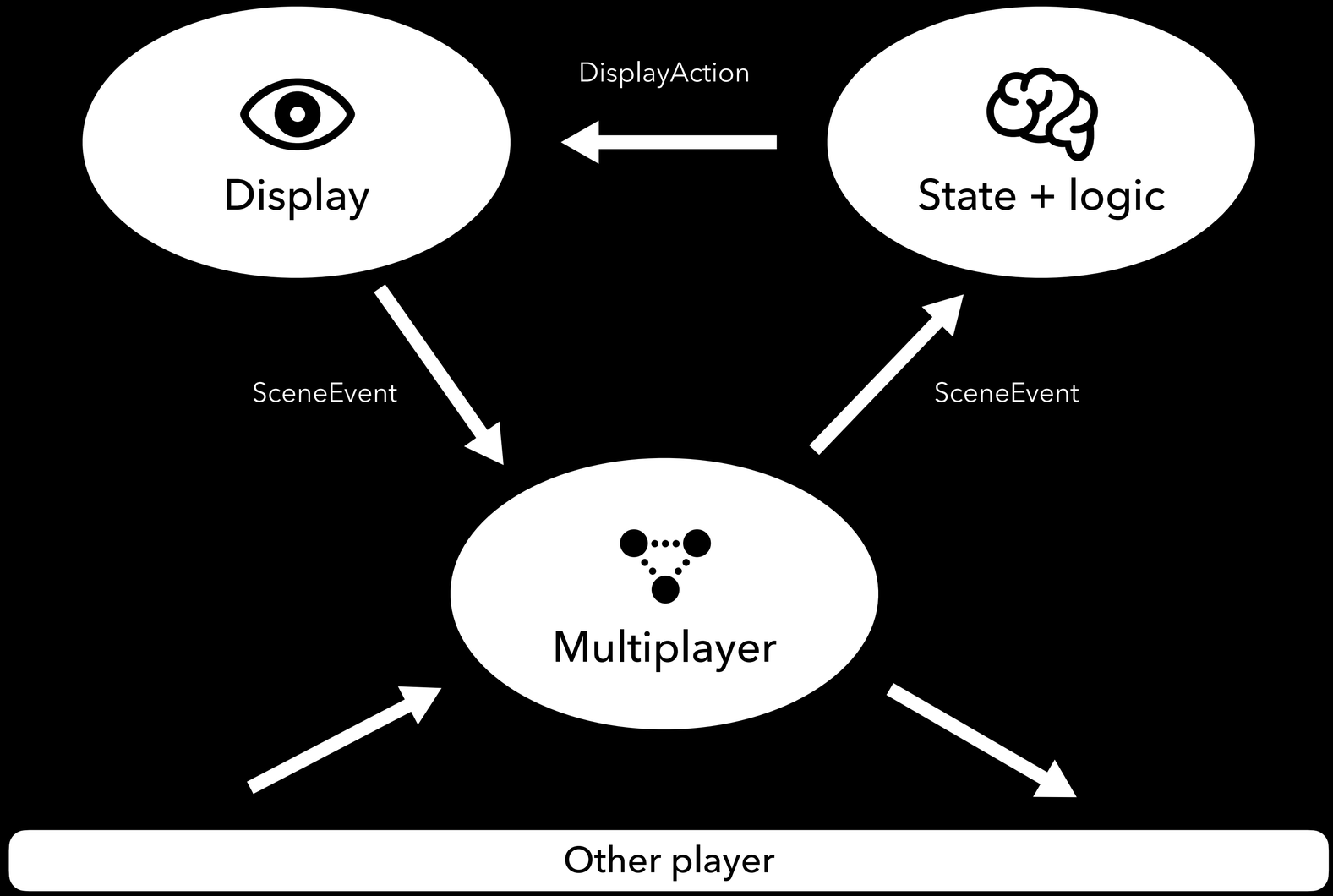
Multiplayer

Architecture



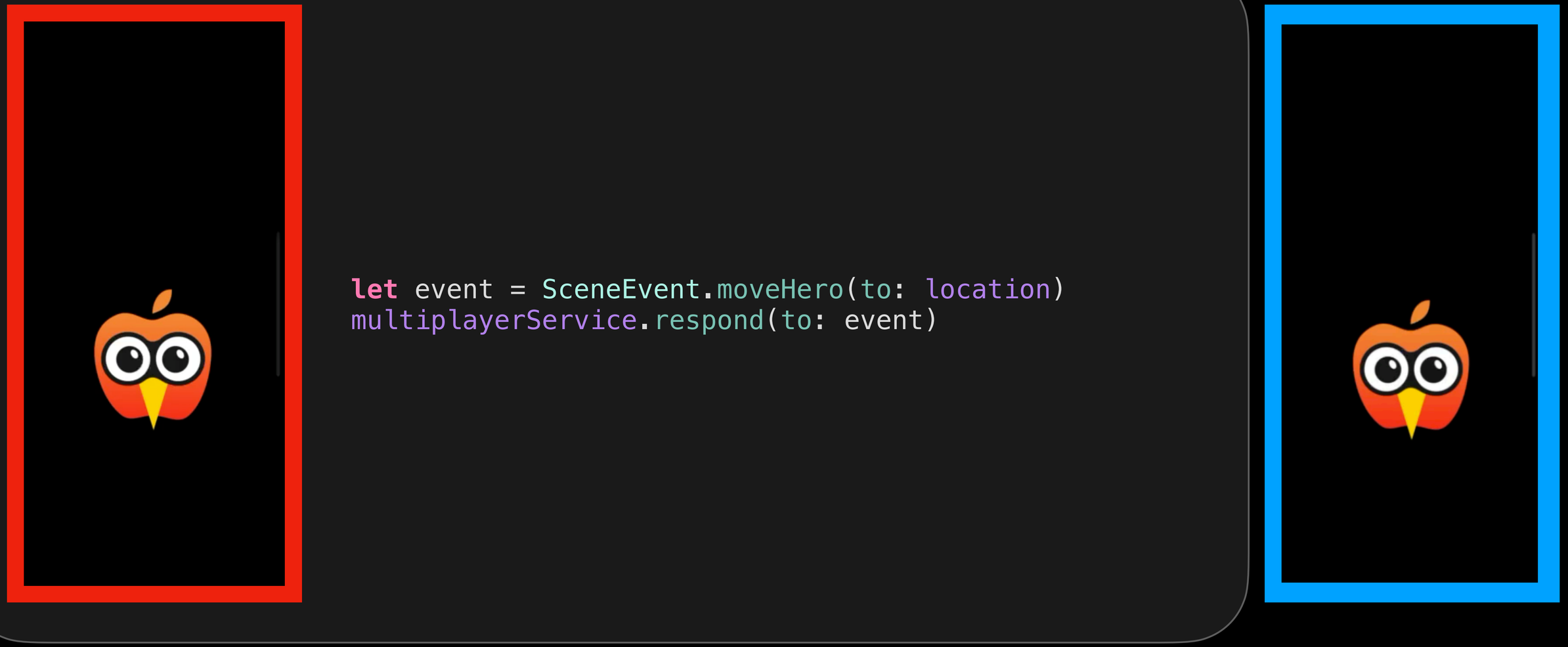
Multiplayer

Architecture



Multiplayer

Exchange events



```
let event = SceneEvent.moveHero(to: location)
multiplayerService.respond(to: event)
```

Multiplayer

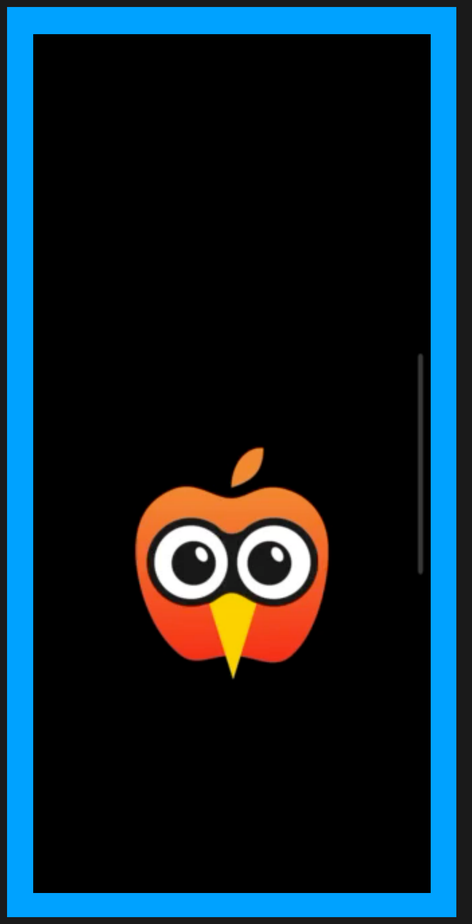
Exchange events

```
var match: GKMatch

func send(_ event: SceneEvent) {

    let encodedEvent = JSONEncoder().encode(event)

    match.send(encodedEvent,
                    to: [opponent],
            dataMode: .reliable)

}
```

Multiplayer

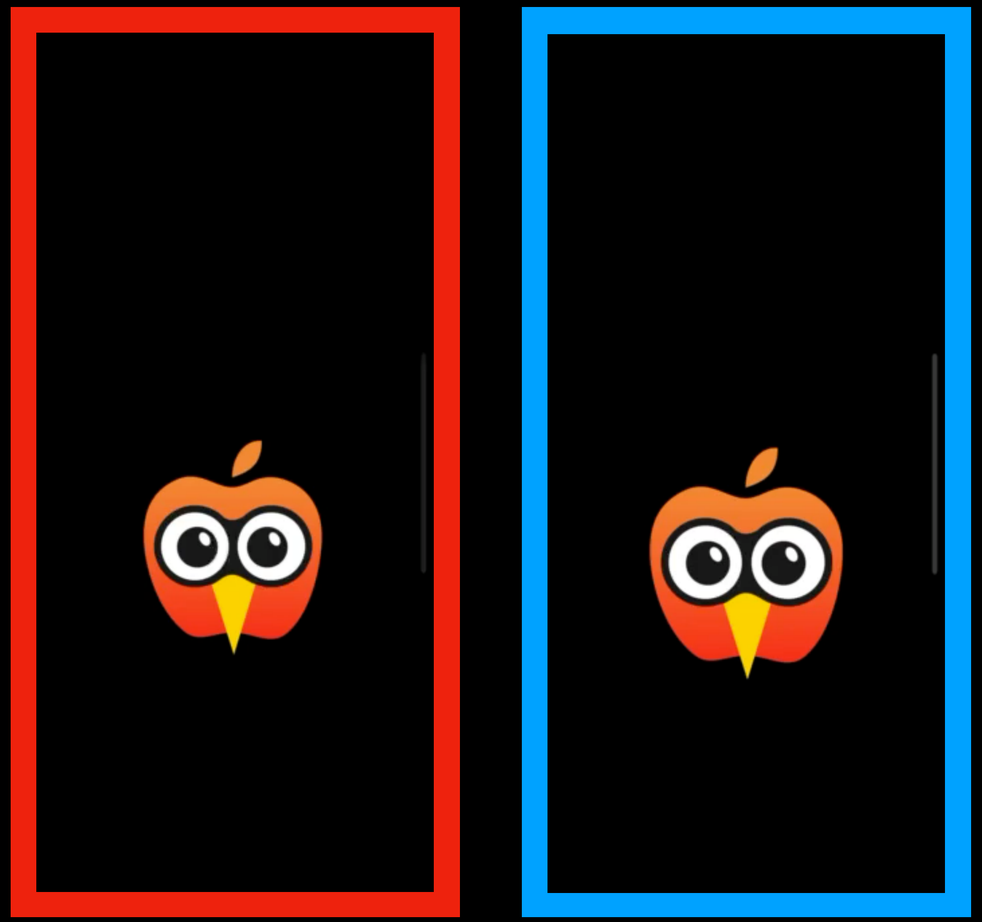
Exchange events

```
extension MultiplayerService: GKMatchDelegate {

  func match(_ match: GKMatch,
     didReceive data: Data,
    fromRemotePlayer player: GKPlayer) {

     let event = decoder.decode(SceneEvent.self,
                                from: data)
     delegate?.didReceive(event)
  }

}
```



Multiplayer

Performances



- Limit the payload size (max 87kb)
- Fast enough for a realtime game (with a good network connection)
- Handle edge cases (connection errors, conflicts…)
- Then you can add some *magic…*

Multiplayer

Optimization

Remote input 1

Multiplayer

Optimization

Remote input 1

Remote input 2

Multiplayer

Optimization

Remote input 1

Remote input 2

Remote input 3

Multiplayer

Optimization

Remote input 1

Remote input 2

Multiplayer

Interpolation

Remote input 1

Remote input 2

Multiplayer

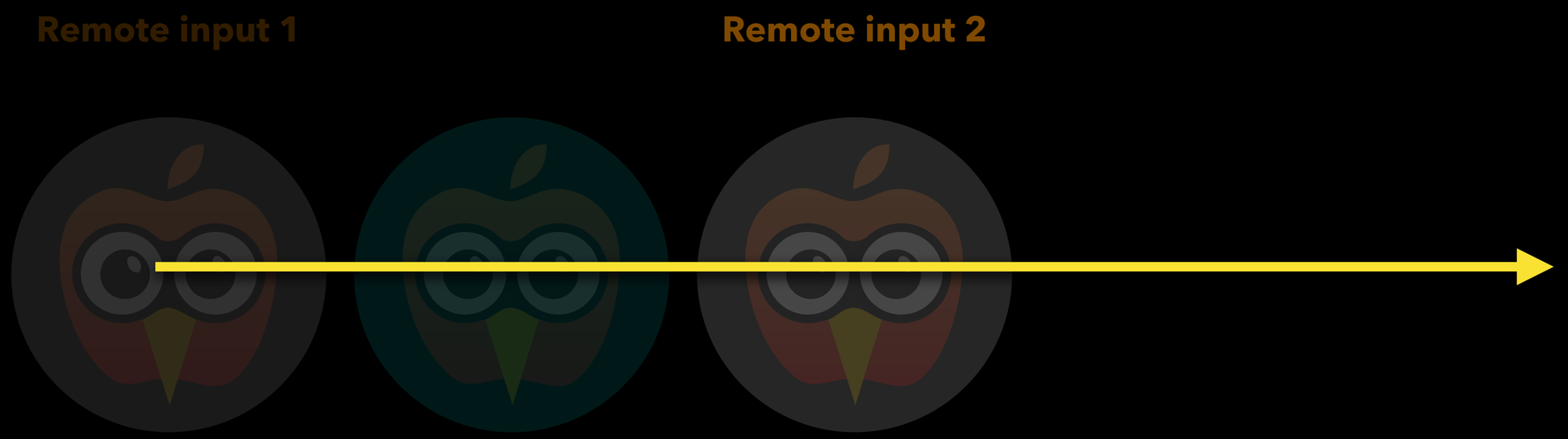
Client-Side Prediction

Remote input 1

Remote input 2

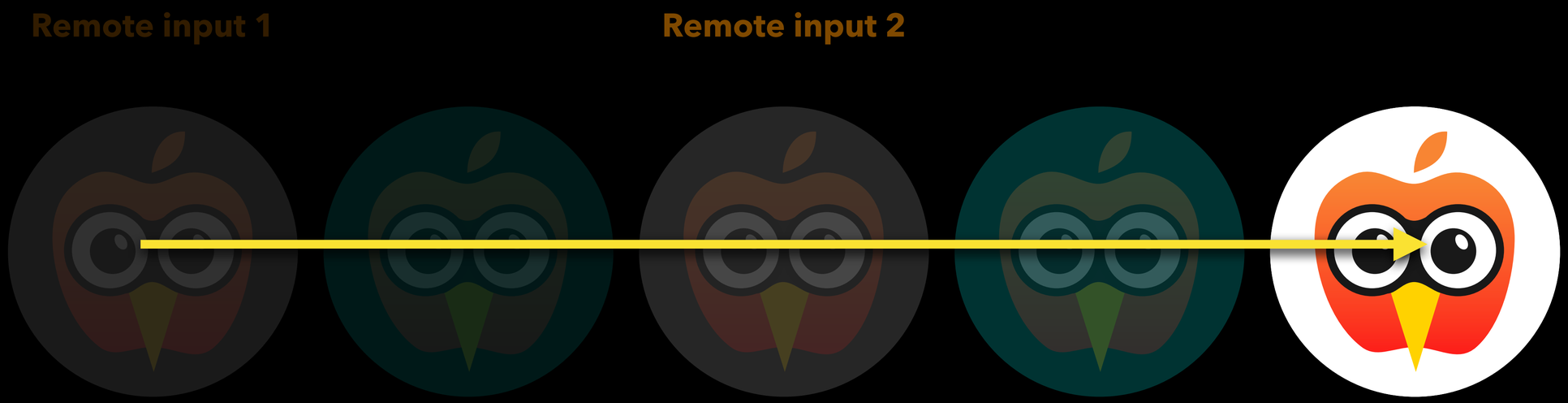
Multiplayer

Client-Side Prediction



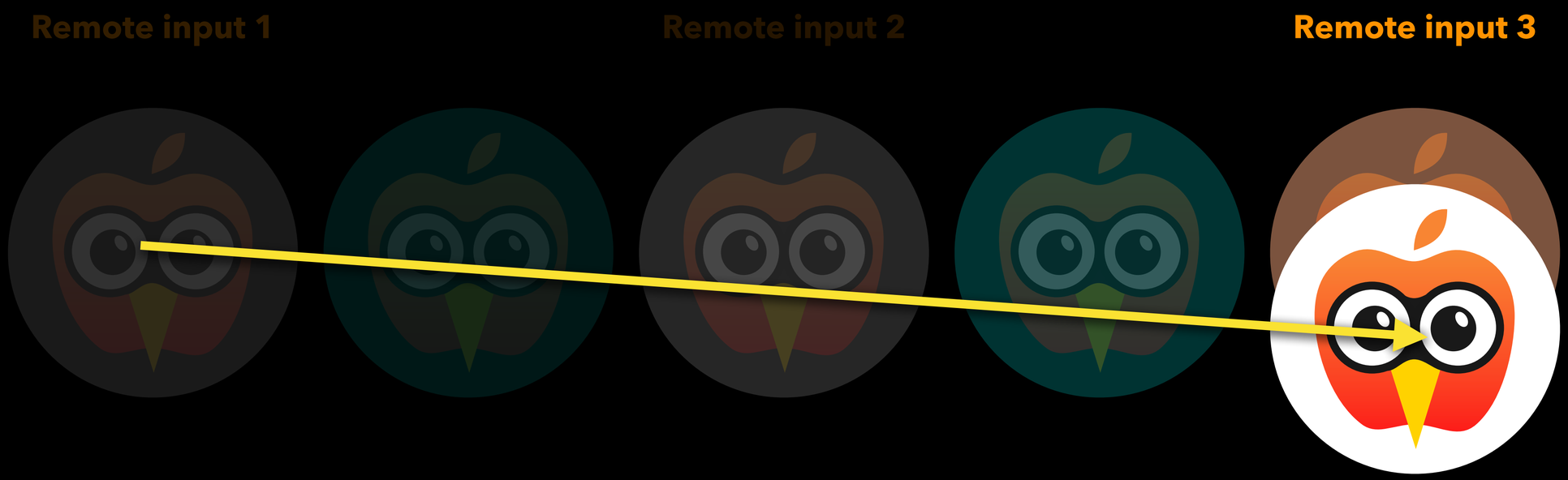
Multiplayer

Client-Side Prediction



Multiplayer

Client-Side Prediction



Learning 5

Takeaway

- Multiplayer connectivity can be easy to implement
- Realtime is possible but with limitations
- First concentrate on robustness
- Then optimize and add smoothness
- Pay attention to edge cases and optimizations

Conclusion

- Solo game dev only with Apple frameworks is possible
- Lots of building blocks, some are missing
- With the right architecture AI and online features are very accessible
- Nice problems to solve
- Game dev is fun !

Thank you !

Thank you !

Michel-André Chirita

@macistador