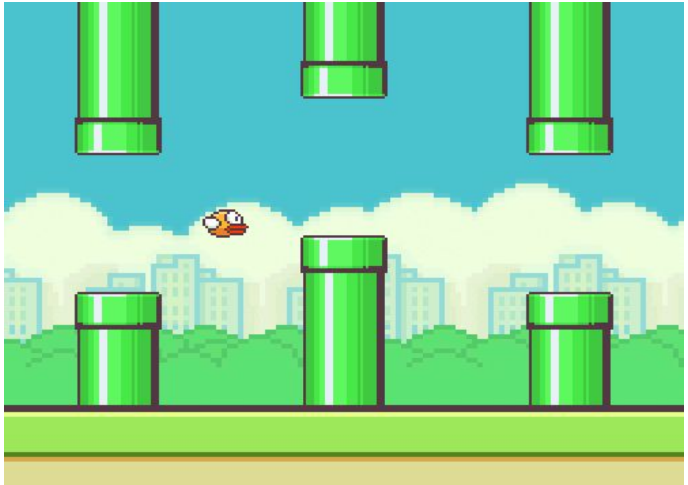
Building an Intelligent Flappy Bird

MADLab

MADLab Mobile Application Development Lab

Agenda

- Intro to mobile development - iOS
- Building the Classic Flappy Bird Game
- Break
- Teaching the Bird How to Fly



History of the Mobile Development

What is mobile development?

- Way to build applications “Apps” for mobile phones
- iOS: Apple phone devices
- Android: Everyone else (other players embargoed from North America i.e HarmonyOS)

iOS Game Dev

Types of mobile development

Native development (Swift, Kotlin, Java)

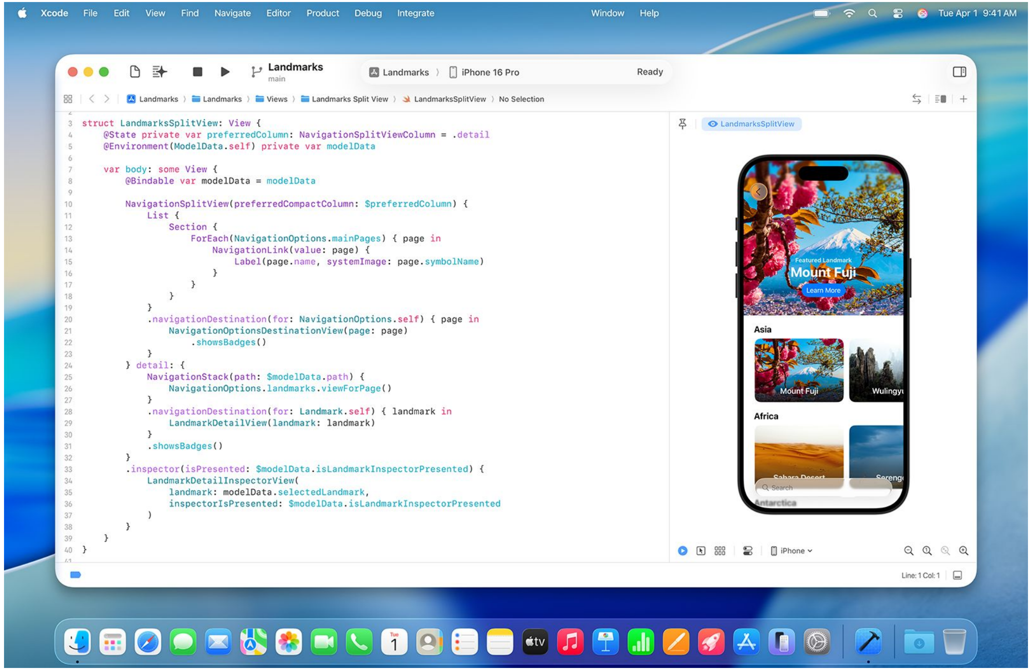
Swift
Kotlin

Cross platform development (Flutter, React Native)

+

iOS Development

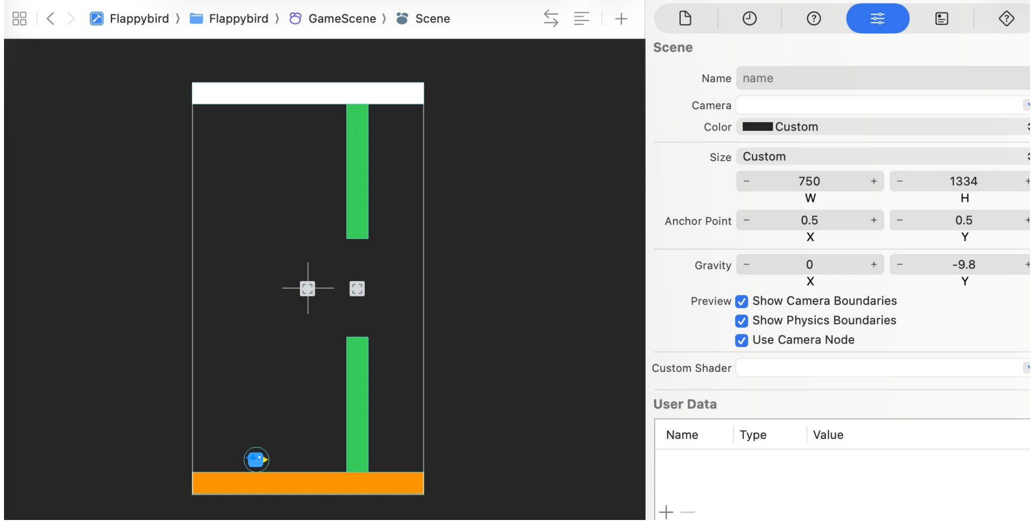
Uses Swift programming language since 2014 (prev. Objective-C)



iOS Game Dev

SpriteKit

- Apple's built-in framework for creating 2D games



Flappy Bird

Dong Nguyen
@dongatory

Following

I am sorry 'Flappy Bird' users, 22 hours from now, I will take 'Flappy Bird' down. I cannot take this anymore.

Reply · Retweet · Favorite · Buffer · More · HootSuite

RETWEET	FAVORITE
51,699	14,501

11:02 AM - 8 Feb 2014

1: Make Flappy Bird

Clone the starter project

1. Open Xcode
2. Click “Clone a repository”
3. Paste the GitHub URL
4. Choose where to save the project
5. Select “Develop” branch
6. Wait for Xcode to open the project

https://github.com/cyn900/FlappyBird

How SpriteKit works

- .sks (SpriteKit Scene) file defines **what exists**
- Swift code defines **what happens**

GameScene
GameScene

```swift
// MARK: - Game Scene
// This is the main game loop controller.
// It owns the world, spawns birds and pipes, handles input, collisions, and scoring.
final class GameScene: SKScene, SKPhysicsContactDelegate {

...
```

Adjustable value

```
    // MARK: Birds
    private var birds = [SKSpriteNode]()
    // ADJUSTABLE: Number of birds to spawn
    private let birdCount = Int(100)

    // MARK: Pipes
    private var pipes = [Pipe]()

    // Gameplay-controlled pipe gap.
    private var pipeGap = CGFloat(130)   // Vertical opening between pipes, overwritten from SKS in didMove()
    // ADJUSTABLE: Leftward (moving to the left) movement speed
    private let pipeSpeed = CGFloat(-2.5)   // Larger absolute value = faster movement.
    // ADJUSTABLE: Distance between pipe spawns
    private let spawnDist = CGFloat(300)

    private var pipeSpawnProgress = CGFloat(0)

    // MARK: Physics tuning

    // ADJUSTABLE: Downward gravity applied to birds.
    private let gravityY = CGFloat(-12)

    // ADJUSTABLE: Upward velocity applied when a bird flaps.
    private var flapVelocityTarget = CGFloat(320)   // Computed again in didMove() to match world scale.
```

Adjustable value

```
// MARK: Birds
private var birds = [SKSpriteNode]()
// ADJUSTABLE: Number of birds to spawn
private let birdCount = Int(100)

// MARK: Pipes
private var pipes = [Pipe]()

// Gameplay-controlled pipe gap.
private var pipeGap = CGFloat(130)   // Vertical opening between pipes, overwritten from SKS in didMove()
// ADJUSTABLE: Leftward (moving to the left) movement speed
private let pipeSpeed = CGFloat(-2.5)   // Larger absolute value = faster movement.
// ADJUSTABLE: Distance between pipe spawns
private let spawnDist = CGFloat(300)

private var pipeSpawnProgress = CGFloat(0)

// MARK: Physics tuning

// ADJUSTABLE: Downward gravity applied to birds.
private let gravityY = CGFloat(-12)

// ADJUSTABLE: Upward velocity applied when a bird flaps.
private var flapVelocityTarget = CGFloat(320)   // Computed again in didMove() to match world scale.
```

Adjustable value

```swift
    // MARK: Birds
    private var birds = [SKSpriteNode]()
    // ADJUSTABLE: Number of birds to spawn
    private let birdCount = Int(100)

    // MARK: Pipes
    private var pipes = [Pipe]()

    // Gameplay-controlled pipe gap.
    private var pipeGap = CGFloat(130)   // Vertical opening between pipes, overwritten from SKS in didMove()
    // ADJUSTABLE: Leftward (moving to the left) movement speed
    private let pipeSpeed = CGFloat(-2.5)   // Larger absolute value = faster movement.
    // ADJUSTABLE: Distance between pipe spawns
    private let spawnDist = CGFloat(300)

    private var pipeSpawnProgress = CGFloat(0)

    // MARK: Physics tuning

    // ADJUSTABLE: Downward gravity applied to birds.
    private let gravityY = CGFloat(-12)

    // ADJUSTABLE: Upward velocity applied when a bird flaps.
    private var flapVelocityTarget = CGFloat(320)   // Computed again in didMove() to match world scale.
```

Adjustable value

```swift
    // MARK: Birds
    private var birds = [SKSpriteNode]()
    // ADJUSTABLE: Number of birds to spawn
    private let birdCount = Int(100)

    // MARK: Pipes
    private var pipes = [Pipe]()

    // Gameplay-controlled pipe gap.
    private var pipeGap = CGFloat(130)   // Vertical opening between pipes, overwritten from SKS in didMove()
    // ADJUSTABLE: Leftward (moving to the left) movement speed
    private let pipeSpeed = CGFloat(-2.5)   // Larger absolute value = faster movement.
    // ADJUSTABLE: Distance between pipe spawns
    private let spawnDist = CGFloat(300)

    private var pipeSpawnProgress = CGFloat(0)

    // MARK: Physics tuning

    // ADJUSTABLE: Downward gravity applied to birds.
    private let gravityY = CGFloat(-12)

    // ADJUSTABLE: Upward velocity applied when a bird flaps.
    private var flapVelocityTarget = CGFloat(320)   // Computed again in didMove() to match world scale.
```

Adjustable value

```swift
    // MARK: Birds
    private var birds = [SKSpriteNode]()
    // ADJUSTABLE: Number of birds to spawn
    private let birdCount = Int(100)

    // MARK: Pipes
    private var pipes = [Pipe]()

    // Gameplay-controlled pipe gap.
    private var pipeGap = CGFloat(130)   // Vertical opening between pipes, overwritten from SKS in didMove()
    // ADJUSTABLE: Leftward (moving to the left) movement speed
    private let pipeSpeed = CGFloat(-2.5)   // Larger absolute value = faster movement.
    // ADJUSTABLE: Distance between pipe spawns
    private let spawnDist = CGFloat(300)

    private var pipeSpawnProgress = CGFloat(0)

    // MARK: Physics tuning

    // ADJUSTABLE: Downward gravity applied to birds.
    private let gravityY = CGFloat(-12)

    // ADJUSTABLE: Upward velocity applied when a bird flaps.
    private var flapVelocityTarget = CGFloat(320)   // Computed again in didMove() to match world scale.
```

How SpriteKit works

```swift
override func didMove(to view: SKView) {
    guard !didInit else { return }
    didInit = true

    physicsWorld.gravity = CGVector(dx: 0, dy: gravityY)
    physicsWorld.contactDelegate = self

    // Prefer // paths so SKS hierarchy changes don't break lookups
    guard let w = childNode(withName: "World") else {
        fatalError("Missing node named World")
    }
    world = w

    guard let bp = w.childNode(withName: "birdPrototype") as? SKSpriteNode else {
        fatalError("Missing SKSpriteNode named birdPrototype")
    }
    birdPrototype = bp
    birdPrototype.removeFromParent()
    ...
```

when the scene is first presented, we set up the scene

How SpriteKit works

```swift
    override func didMove(to view: SKView) {
        guard !didInit else { return }
        didInit = true

        physicsWorld.gravity = CGVector(dx: 0, dy: gravityY)
        physicsWorld.contactDelegate = self

        // Prefer // paths so SKS hierarchy changes don't break lookups
        guard let w = childNode(withName: "World") else {
            fatalError("Missing node named World")
        }
        world = w

        guard let bp = w.childNode(withName: "birdPrototype") as? SKSpriteNode else {
            fatalError("Missing SKSpriteNode named birdPrototype")
        }
        birdPrototype = bp
        birdPrototype.removeFromParent()
        ...
```

Create prototypes for things that are spawned multiple times or reset often.
Don't create prototypes for things that exist once and persist.

Update loop

Called Every Frame

- Runs **once per rendered frame**
- Usually ~60 FPS

What Happens Every Frame

- Spawn pipes
- Move pipes
- Remove off-screen pipes
- Update score
- And more

```swift
// MARK: Game loop
override func update(_ currentTime: TimeInterval) {

    let dt: TimeInterval
    if lastUpdateTime == 0 {
        dt = TimeInterval(1.0 / 60.0)
    } else {
        dt = currentTime - lastUpdateTime
    }
    lastUpdateTime = currentTime

    // Auto-restart after game over
    if gameOver {
        if let t = restartAt, currentTime >= t {
            resetGame()
        }
        return
    }

    // Move pipes left
    for p in pipes { p.move(pipeSpeed) }
```

Pipes

One Pipe = Two Sprites

- Top pipe
- Bottom pipe
- Gap in between

What Happens When a Pipe Spawns

- Choose a **random gap position**
- Resize top & bottom pipes

```
    private func spawnPipe() {
        let worldMinY = groundNode.frame.maxY
        let worldMaxY = ceilingNode.frame.minY

        let minGapCenter = worldMinY + pipeGap * 0.5
        let maxGapCenter = worldMaxY - pipeGap * 0.5

        let gapYWorld: CGFloat
        if minGapCenter < maxGapCenter {
            gapYWorld = CGFloat.random(in: minGapCenter...maxGapCenter)
        } else {
            gapYWorld = (worldMinY + worldMaxY) * 0.5
        }

        let xWorld = sceneXToWorld(frame.maxX + 60)

        let pipe = Pipe(
            template: pipePrototype,
            xPosInWorld: xWorld,
            gapYInWorld: gapYWorld,
            gap: pipeGap,
            worldMinY: worldMinY,
            worldMaxY: worldMaxY
        )

        pipes.append(pipe)
        world.addChild(pipe.node)
    }
```

Game Play

The Bird Is Physics-Driven

- Gravity is **always on**
- No manual position updates
- Physics engine moves the bird

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    flapAll()
}

private func flapAll() {
    for b in birds {
        guard let body = b.physicsBody, body.isDynamic else { continue }
        body.velocity = CGVector(dx: body.velocity.dx, dy: flapVelocityTarget)
    }
}
```

Flapping = Reset Vertical Velocity

- Tap does **not** push the bird
- Tap does **not** animate the bird
- Tap **sets Y velocity**

Collision

- Every physics body has a **category**

```swift
enum PhysicsCategory {
    static let bird: UInt32    = 1 << 0 // 0001
    static let pipe: UInt32    = 1 << 1 // 0010
    static let ground: UInt32  = 1 << 2 // 0100
    static let ceiling: UInt32 = 1 << 3 // 1000
}
```

Three Important Masks

- categoryBitMask → *what am I?*
- collisionBitMask → *what stops me?* → physically stops movement
- contactTestBitMask → *what do I get notified about?* → trigger game over

```swift
body.categoryBitMask = PhysicsCategory.bird
body.contactTestBitMask = PhysicsCategory.pipe | PhysicsCategory.ground |
    PhysicsCategory.ceiling
body.collisionBitMask = PhysicsCategory.pipe | PhysicsCategory.ground |
    PhysicsCategory.ceiling
```

Collision = Death

When a Collision Happens

- SpriteKit notifies us automatically
- Called when **two physics bodies touch**

```
// MARK: Collision
func didBegin(_ contact: SKPhysicsContact) {
    let birdBody: SKPhysicsBody?
    if contact.bodyA.categoryBitMask == PhysicsCategory.bird { birdBody =
        contact.bodyA }
    else if contact.bodyB.categoryBitMask == PhysicsCategory.bird { birdBody =
        contact.bodyB }
    else { birdBody = nil }

    guard let bBody = birdBody, let node = bBody.node as? SKSpriteNode else {
        return }
    killBird(node)
}
```

Game Restart

What “Death” Means in This Game

- Physics is disabled
- Bird is hidden

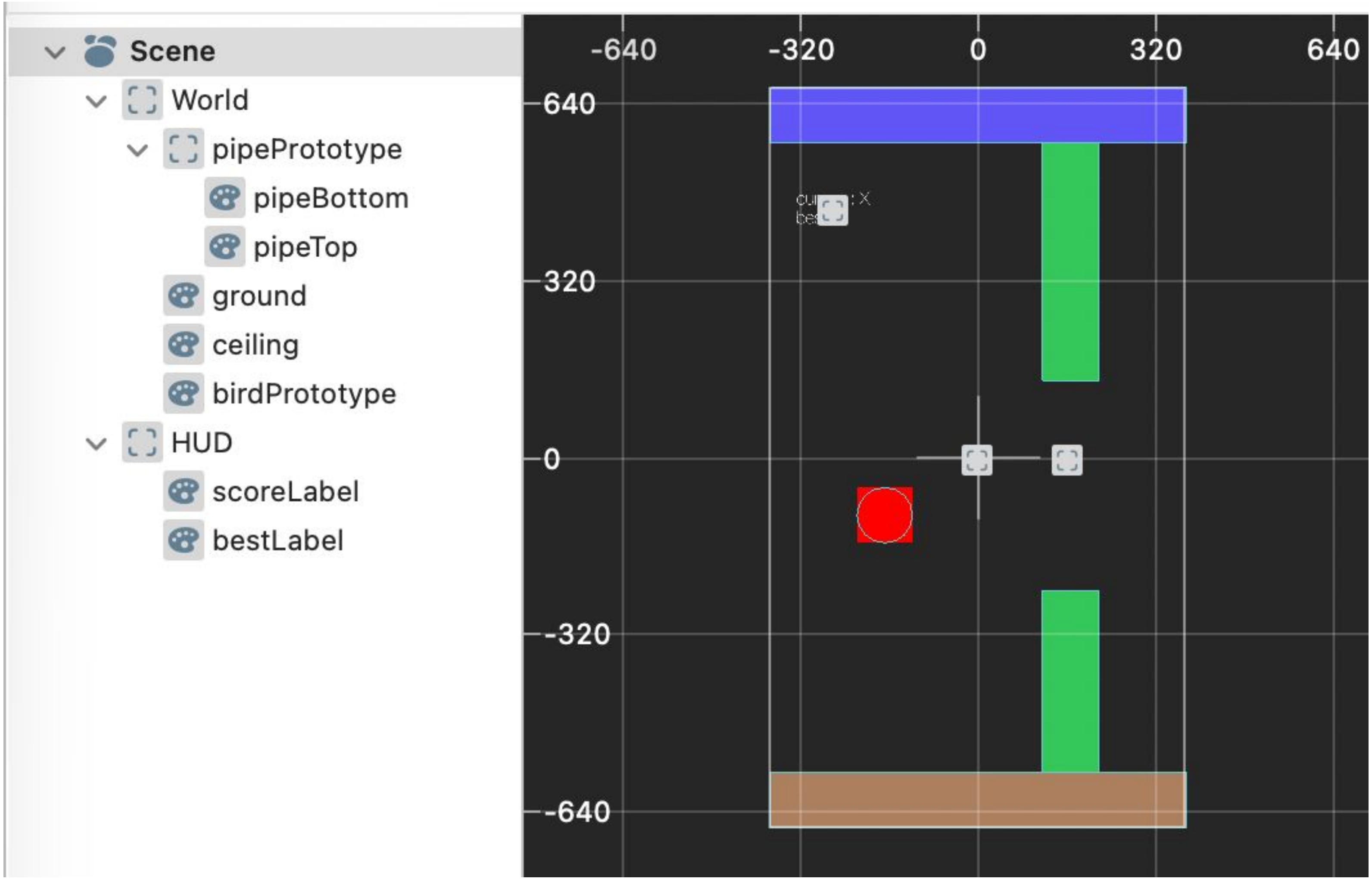
Game Over Condition

- Check if **all birds are inactive**
- Trigger game over once

Auto Restart

- Short delay
- Reset game state
- Start again

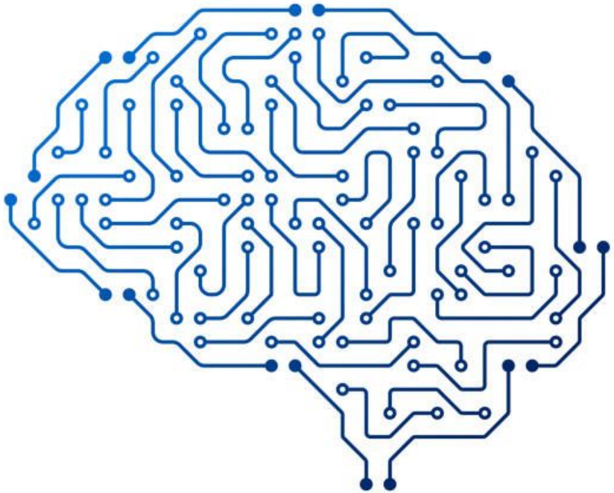
```swift
private func killBird(_ bird: SKSpriteNode) {
    guard let idx = birds.firstIndex(where: { $0 === bird }) else { return }
    let b = birds[idx]
    b.alpha = 0
    b.physicsBody?.isDynamic = false

    // If all birds are dead -> auto restart
    if birds.allSatisfy({ $0.physicsBody?.isDynamic == false }) {
        triggerGameOver()
    }
}

private func triggerGameOver() {
    guard !gameOver else { return }
    gameOver = true
    restartAt = lastUpdateTime + 0.6
}
```


Scene
World
pipePrototype
pipeBottom
pipeTop
ground
ceiling
birdPrototype
HUD
scoreLabel
bestLabel
-640
-320
0
320
640
640
320
0
-320
-640

2: Building the Bird’s Brain

What is AI?

- Process of collecting data
- Storing data
- Using data to make decisions



What are we going to build today?

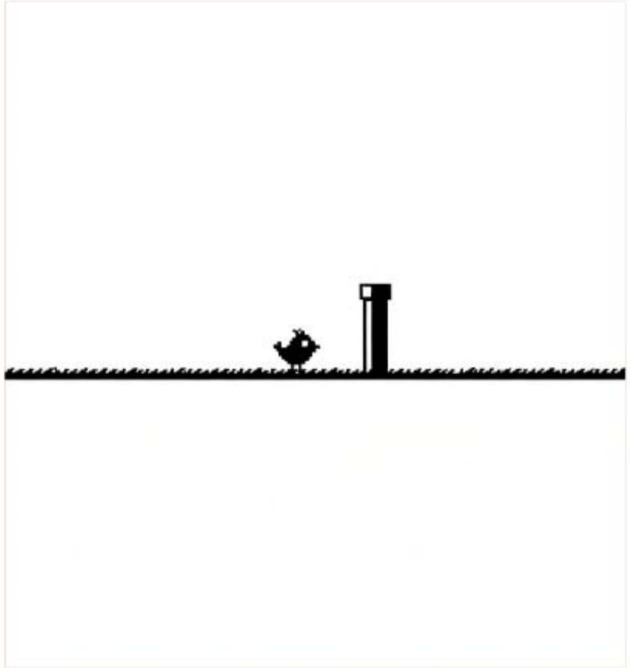
Here's our plan:

1. Make a bird that can decide when to flap
2. Give it a simple brain
3. Let it play the game
4. See why it fails
5. Figure out how to make it better

What Are We Doing?

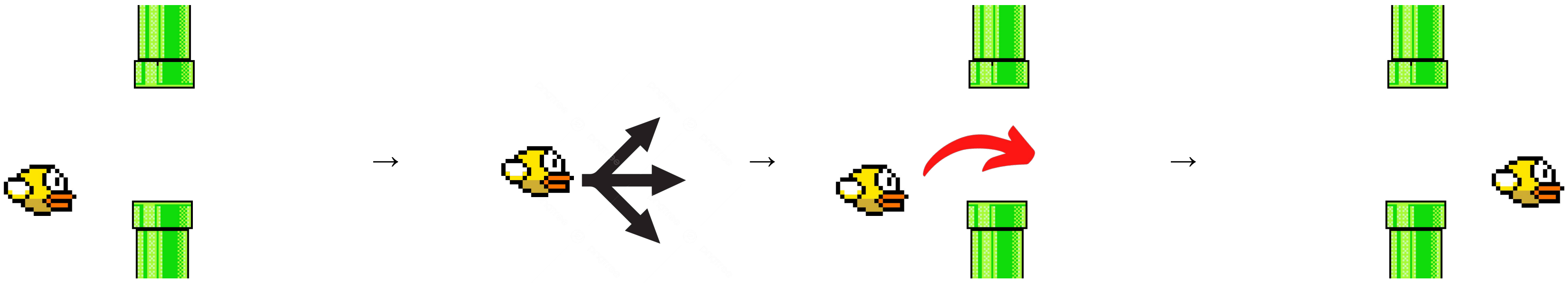
Flap or Not Flap

```
final class NeuralNetwork {
```



From Facts to Action

The brain doesn’t think in words — it processes information.



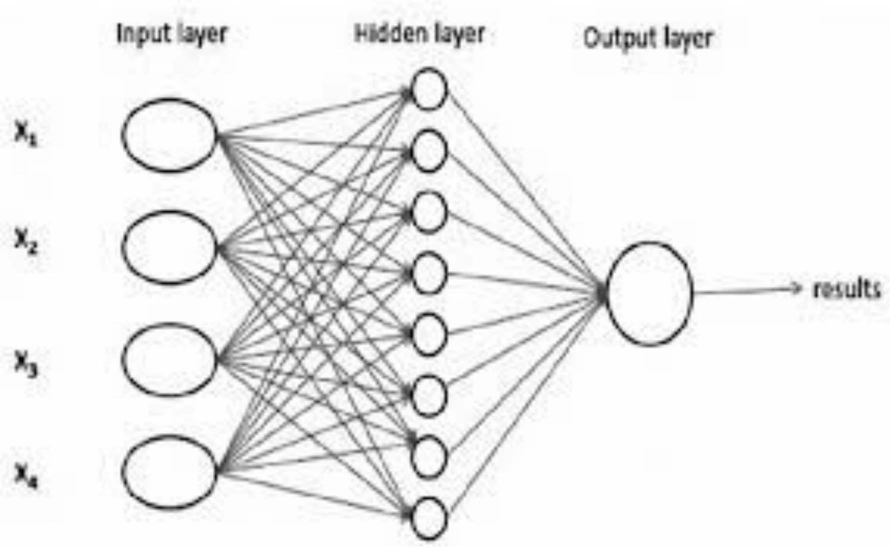
Fact **Analysis** **Decision** **Action**

Brain structure

- **4 inputs** → facts the bird know about the situation
  (height, top pipe, bottom pipe, distance)
  (only raw information, nothing else)

- **8 hidden neurons** → little helpers that mix and think about the facts
  (they do NOT decide)
  (each sees all 4 inputs)
  (each focuses differently)

- **1 output** → final decision



Importance and instinct

Trust level of input

- w1 → tells hidden neurons **how important each fact is**
 (pipe distance may matter more than height, etc.)

- w2 → tells output **how important each hidden neuron is**
 (which helper we trust more)

Instinct

- b1 → small adjustments for hidden neurons
 (personal tendency of each helper)

- b2 → small adjustment for output
 (overall instinct push before decision)

The Bird’s Facts

That’s **all** it gets

- Its current height
- Where the top of the pipe is
- Where the bottom of the pipe is
- How far the next pipe is

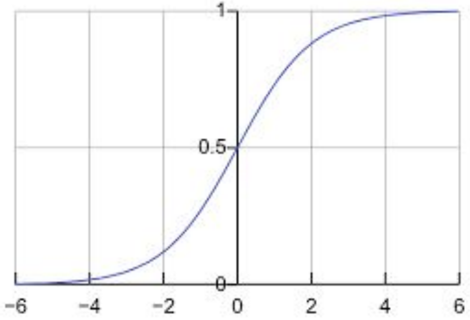
```
let inputs: [Double] = [
    birdY / height,        // bird height
    topY / height,         // top pipe
    botY / height,         // bottom pipe
    min(dist, 600) / 600.0 // distance to next pipe
]
```

How the Brain Analyzes

- Takes facts
    a. Each fact has importance (**weights**)
- Multiply each fact by its importance
- Adds all the values together
    a. Plus a small extra number (**bias**) to help adjust decisions.
- Passed through a filter (**sigmoid**)
- Gets a number from 0 to 1

```
// calculate brain's decision based on input facts
func predict(_ inputs: [Double]) -> Double {
    var hidden = [Double](repeating: 0, count: 8) // store
        hidden neuron outputs

    // compute each hidden neuron
    for i in 0..<8 {
        var sum = b1[i] // start with bias (adjustment)
        for j in 0..<4 {
            sum += inputs[j] * w1[i][j] // each fact × importance
        }
        hidden[i] = sigmoid(sum) // filter result to 0-1
    }

    // compute output neuron
    var out = b2 // start with output bias
    for i in 0..<8 {
        out += hidden[i] * w2[i] // combine all hidden outputs
    }

    return sigmoid(out) // final probability to flap
}
```

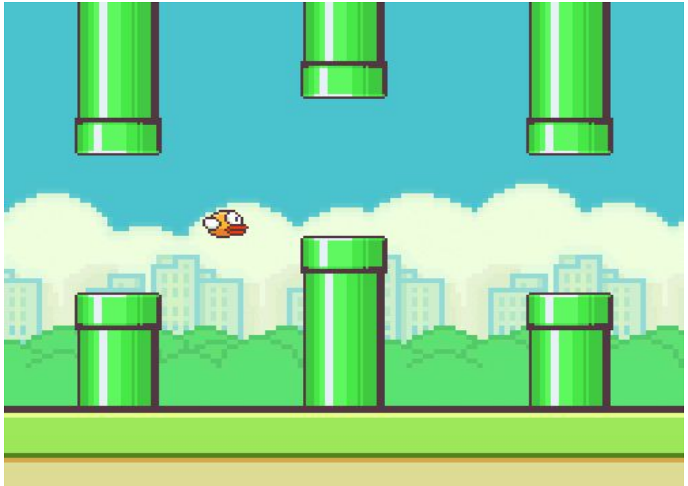
Sigmoid function




```
1.0 / (1.0 + exp(-x))
```

This turns any number into a value between **0 and 1**, which is perfect for decisions.

The Brain’s Output



Analyzes → produces **one number**

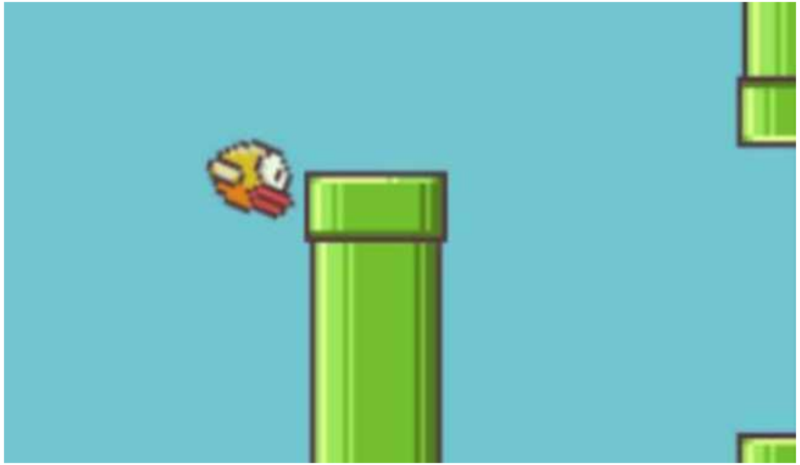
- Big (> 0.5) → flap
- Small → do nothing

```
func shouldFlap(birdIndex: Int, birdY: Double, topY: Double,
    botY: Double, dist: Double, height: Double) -> Bool {
    let g = genomes[birdIndex]
    guard g.alive else { return false } // dead birds don't flap

    // convert game state into normalized facts
    let inputs: [Double] = [
        birdY / height,         // bird height
        topY / height,          // top pipe
        botY / height,          // bottom pipe
        min(dist, 600) / 600.0 // distance to next pipe
    ]
    return g.brain.predict(inputs) > 0.5 // logic: flap if brain
        says yes
}
```

So… does this bird fly well?

Why the First Brain Is Bad

- Starts random → importance chosen **by chance**
- Brain doesn't know what matters yet
- Flaps randomly → crashes



```
// create initial population with random brains
genomes = (0..<popSize).map { _ in Genome(brain:
    NeuralNetwork()) }
}
```

How do we make the brain better?

We have a problem:

- One bird
- One random brain
- No learning

How do we improve the brain without teaching it?

Big Idea: use many birds

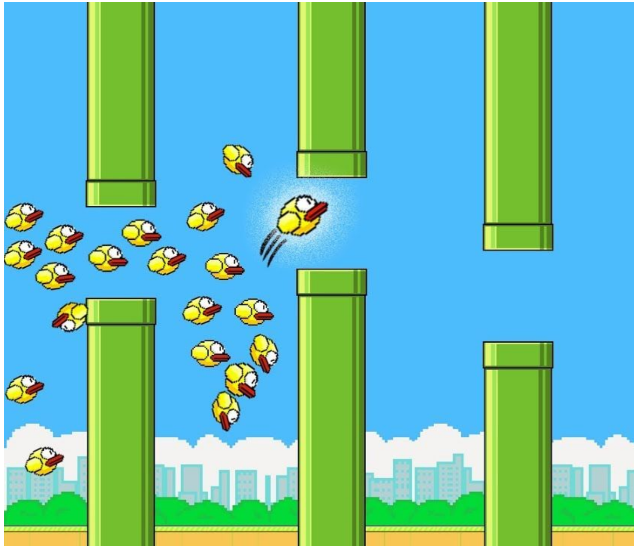
Instead of 1 bird…

we create **100 birds**.

All with:

- the same structure
- different random brains

We use **natural selection** — the best birds survive and become the next generation of 100 birds.



Learning Through Evolution

```
func evolveToNextGen() {
```

The next generation is slightly better than the last.
Over many generations, the brains improve automatically through **natural selection**.

Nature:

- Many are born
- Some survive longer
- The best survive and pass traits

AI:

- 100 birds start
- some crash immediately, some fly longer
- keep and copy the best brains
- slightly change them, sometimes mix two

About the Code

The full project is already pre-coded.

We will briefly explain the main classes and the purpose of each one.

A bird = Genome

Each bird:

- Has a brain
- Can be alive or dead
- Gets points
- Travels distance

```swift
final class Genome {
    let brain: NeuralNetwork // brain of this bird
    var alive: Bool = true   // is it still flying?
    var score: Int = 0       // pipes passed
    var distance: Double = 0 // how far it traveled
```

Fitness = how good the bird is

```swift
var fitness: Double { Double(score) * 1000 + distance }
// logic: combines score and distance into one "how good" number
```

Fitness is:

- Mostly based on score
- Distance matters too
- Higher fitness = better bird

Managing all birds — FlappyAI

```
final class FlappyAI {
```

This class:

- Creates birds
- Tracks them
- Kills them on collision
- Chooses the best
- Creates the next generation

During the game

Each game tick:

- `shouldFlap(...)` → asks the brain if it should flap
- `addScore(i:)` → increases score when passing pipes
- `tickAlive(i:, distance:)` → updates distance
- `kill(i:)` → kills bird if it crashes

Sort by best fitness

Best birds go first.

```
genomes.sort { $0.fitness > $1.fitness } // rank birds by
    fitness
```

Select elites

Take the **top 10%** (at least 2 birds).
They become parents.

```
let eliteCount = max(2, popSize / 10) // keep top 10% as
    parents
let elites = Array(genomes.prefix(eliteCount))
```

Copy elites directly

Best birds go into the next generation **unchanged**.

```
// 1) keep elites exactly
for e in elites {
    newGen.append(Genome(brain: e.brain.copy())) // best
        birds survive unchanged
}
```

Create children

While population is not full:

```
let p1 = elites.randomElement()! // parent 1
let p2 = elites.randomElement()! // parent 2

let child = NeuralNetwork.average(p1.brain, p2.brain) // combine traits
child.tinyMutate(chance: 0.03, amount: 0.08) // slight variation
```

🔀 a) Crossover (averaging weights)

Child’s brain = mix of **two parents’ brains**.

Combines their strengths.

🧬 b) Mutation

With a small chance (3%), some numbers change slightly.

This helps:

- Create new ideas
- Avoid getting stuck

New generation is ready

Now all birds are new, and the cycle repeats.

```
genomes = newGen // replace old population
generation += 1 // increment generation counter
```

Why does this work?

Because:

- Bad birds die
- Good birds reproduce
- Children are slightly different
- Good differences survive

This is **evolution**, just like in nature.

Making Learning Faster

Simple code
→ Training time: ~1 hour
→ Same result
→ Slower learning

Advanced code
→ Training time: ~5 minutes
→ Same result
→ Faster learning

Same goal. Same bird. Just faster training.

IMPORTANT: This is NOT classic machine learning

There is:

❌ No correct answers

❌ No magic math that fixes mistakes automatically (no loss function)

❌ No complex brain training steps (no backpropagation)

✔️ Birds play the game

✔️ We watch who does well

✔️ The best birds get to have children

✔️ Children brains are copied and slightly changed

Back to the code

Where Evolutionary AI Can Be Used

- Self-driving cars → AI decides how to drive safely
- Medical diagnosis → AI finds patterns in patient data
- Recommendations → AI picks the best options for users
- Robots → AI learns how to move or complete tasks
- Trading → AI adapts to market conditions

Anywhere you have **data** → **decision** → **improvement over time**

We didn’t build a smart bird.

We built:

data → analysis → decision → improvement

That’s the foundation of AI.