Título

Assunto

- Classes
  - MCSession - Uma sessão gerencia todas as comunicações entre seus pares associados. Você pode enviar mensagens, arquivos e fluxos por meio de uma sessão e seu representante será notificado quando um deles for recebido de um ponto conectado.
  - MCPeerID - Um ID de mesmo nível permite identificar dispositivos de pares individuais em uma sessão. Ele tem um nome associado a ele, mas tenha cuidado: IDs pares com o mesmo nome não são considerados idênticos
  - MCNearbyServiceAdvertiser - Um anunciante permite que você transmita seu nome de serviço para dispositivos próximos. Isso permite que eles se conectem a você.
  - MCNearbyServiceBrowser - Um navegador permite procurar dispositivos usando o MCNearbyServiceAdvertiser. O uso dessas duas classes juntas permite que você descubra dispositivos próximos e crie suas conexões ponto a ponto.
  - MCBrowserViewController - Isso fornece uma interface do usuário muito básica para navegar por serviços de dispositivos próximos.

```swift
class SenderServiseManeger: NSObject {

    private let senderServiceType = "myString"

    private let myPeerId = MCPeerID(displayName: UIDevice.current.name)

    private let serviceAdvertiser: MCNearbyServiceAdvertiser
    private let serviceBrowser: MCNearbyServiceBrowser
    private var serviceAdviserAssistant: MCAdvertiserAssistant!

    lazy var session: MCSession = {
        let session = MCSession(peer: myPeerId, securityIdentity: nil, encryptionPreference: .required)
        session.delegate = self
        return session
    }()


    var delegate: SenderServiceManagerDelegate?
    var gameDelegate: GameDelegate?


    override init() {
        self.serviceAdvertiser = MCNearbyServiceAdvertiser(peer: myPeerId, discoveryInfo: nil, serviceType: senderServiceType)
        self.serviceBrowser = MCNearbyServiceBrowser(peer: myPeerId, serviceType: senderServiceType)
        super .init()
        self.serviceAdvertiser.delegate = self
        self.serviceAdvertiser.startAdvertisingPeer()

        self.serviceBrowser.delegate = self
        //        self.serviceBrowser.startBrowsingForPeers()

        //self.delegate = GameViewController()
    }

    deinit {
        self.serviceAdvertiser.stopAdvertisingPeer()
        self.serviceBrowser.stopBrowsingForPeers()
    }
```

Mostra como se inicia o serviço de conexão do MPC. Como vimos, precisamos instanciar objetos de cada classe. Um diferencial são as variáveis delegate e gameDelegate, que são uma forma de criar um vinculo com as classes que herdam esses delegates e usar os métodos implementados através de um protocolo.

```
func sendCommand(command : String) {
    //NSLog("%@", "sendColor: \(command) to \(session.connectedPeers.count) peers")

    if session.connectedPeers.count > 0 {
        do {
            try self.session.send(command.data(using: .utf8)!, toPeers: session.connectedPeers, with: .reliable)
        }
        catch let error {
            NSLog("%@", "Error for sending: \(error)")
        }
    }

}

func sendPlayer(player: Player){
    //NSLog("%@", "sendColor: \(player) to \(session.connectedPeers.count) peers")

    if session.connectedPeers.count > 0 {
        do {
            print("BB")
            let dict: Dictionary<String, Player> = ["player": player]
            let encodeDict = try NSKeyedArchiver.archivedData(withRootObject: dict, requiringSecureCoding: true)
            try self.session.send(encodeDict, toPeers: session.connectedPeers, with: .reliable)

        }
        catch let error {
            NSLog("%@", "Error for sending: \(error)")
        }
    }
}

func host(){
    self.serviceAdviserAssistant = MCAdvertiserAssistant(serviceType: senderServiceType, discoveryInfo: nil, session: self.session)
    self.serviceAdviserAssistant.start()
    //self.gameDelegate?.createPlayer(pos: CGPoint(x: CGFloat(Int.random(in: 20...60)), y: CGFloat(Int.random(in: 20...60))), id: "001")

}

func join() -> MCBrowserViewController{
    let mcBrowser = MCBrowserViewController(serviceType: senderServiceType, session: self.session)
    mcBrowser.maximumNumberOfPeers = 1
    mcBrowser.delegate = self
    return mcBrowser
}
```

A função sendCommand é responsável por enviar uma ação para outros dispositivos (No caso do app é apenas pular). Caso condição seja verdadeira ele enviará uma string em forma de dado.

A finção sendPlayer é responsável por criar os avatares no jogo assim que houver uma conexão. Para isso cria um dicionário de player só para poder dar decode, visto pelo método NSKeyedArchiver.archiverData. Depois o objeto decodificado é enviado.

Host e join são funções para estabelecer consoes com outos dispositivos. No caso do join foi limitado que haja apenas um peer.

```
extension SenderServiseManeger: MCNearbyServiceAdvertiserDelegate {

    func advertiser(_ advertiser: MCNearbyServiceAdvertiser, didNotStartAdvertisingPeer error: Error) {
        //NSLog("%@", "didNotStartAdvertisingPeer: \(error)")
    }

    func advertiser(_ advertiser: MCNearbyServiceAdvertiser, didReceiveInvitationFromPeer peerID: MCPeerID, withContext context: Data?,
        invitationHandler: @escaping (Bool, MCSession?) -> Void) {
        //NSLog("%@", "didReceiveInvitationFromPeer \(peerID)")
        invitationHandler(true, self.session)
    }


}

extension SenderServiseManeger: MCNearbyServiceBrowserDelegate{

    func browser(_ browser: MCNearbyServiceBrowser, didNotStartBrowsingForPeers error: Error) {
        //NSLog("%@", "didNotStartBrowsingForPeers: \(error)")
    }

    func browser(_ browser: MCNearbyServiceBrowser, foundPeer peerID: MCPeerID, withDiscoveryInfo info: [String : String]?) {
        //NSLog("%@", "foundPeer: \(peerID)")
        browser.invitePeer(peerID, to: self.session, withContext: nil, timeout: 10)
    }

    func browser(_ browser: MCNearbyServiceBrowser, lostPeer peerID: MCPeerID) {
        //NSLog("%@", "lostPeer: \(peerID)")
    }


}
```

Implementação dos protocolos. No geral segue-se esse modelo.

```swift
extension SenderServiseManeger: MCSessionDelegate{
    func session(_ session: MCSession, peer peerID: MCPeerID, didChange state: MCSessionState) {
        //NSLog("%@", "peer \(peerID) didChangeState: \(state)")
        //self.delegate?.connectedDevicesChanged(manager: self, connectedDevices:
            //session.connectedPeers.map{$0.displayName})
    }

    func session(_ session: MCSession, didReceive data: Data, fromPeer peerID: MCPeerID) {
        //NSLog("%@", "didReceiveData: \(data)")

        let pData = try? NSKeyedUnarchiver.unarchiveTopLevelObjectWithData(data)
        let playerData = pData as? [String : Player]

        if "player" == playerData?.keys.first{
            self.delegate?.passPlayer(player: ((playerData?["player"])!))
            }
        else{
            let str = String(data: data, encoding: .utf8)
            print("A sessao", self.session.myPeerID.displayName)
            self.delegate?.command(maneger: self, action: str!, peerId: peerID.displayName)
        }

        print(session.connectedPeers)

    }

    func session(_ session: MCSession, didReceive stream: InputStream, withName streamName: String, fromPeer peerID: MCPeerID) {
        //NSLog("%@", "didReceiveStream")
    }

    func session(_ session: MCSession, didStartReceivingResourceWithName resourceName: String, fromPeer peerID: MCPeerID, with progress: Progress) {
        //NSLog("%@", "didStartReceivingResourceWithName")
    }

    func session(_ session: MCSession, didFinishReceivingResourceWithName resourceName: String, fromPeer peerID: MCPeerID, at localURL: URL?,
        withError error: Error?) {
        //NSLog("%@", "didFinishReceivingResourceWithName")
    }


}
```

Foco na implementação da sessão que há o didReceive. Ele é sempre acionado quando se manda algo através do método send, como visto. Eu tento encodificar o dado e depois castando como um dicionário, podendo ser nulo. Depois é feita uma comparação se a chave do dicionário recebido é player para depois adicionar os players na cena, através do método passPlayer. Na condição contraria é onde acontece a ação de comando, passando qual string da ação e quem enviou.1

```
extension SenderServiseManeger: MCBrowserViewControllerDelegate{
    func browserViewControllerDidFinish(_ browserViewController: MCBrowserViewController) {
         self.delegate?.dismissView(algo: "oi")
//        print(browserViewController.session.myPeerID)
        let x = CGFloat(Int.random(in: 0...200))
        let y = CGFloat(Int.random(in: 0...100))
        print(x,y)
        self.gameDelegate?.createPlayer(pos: CGPoint(x: x, y: y), id: session.myPeerID.displayName)
//        let x2 = CGFloat(Int.random(in: 200...300))
//        let y2 = CGFloat(Int.random(in: 0...100))
//        print(x2,y2)
//        self.gameDelegate?.createPlayer(pos: CGPoint(x: x2, y: y2), id: "002")

        for peer in browserViewController.session.connectedPeers{
            let x = CGFloat(Int.random(in: 200...300))
            let y = CGFloat(Int.random(in: 0...100))
            self.gameDelegate?.createPlayer(pos: CGPoint(x: x, y: y), id: peer.displayName)
            print(peer.displayName, x, y)
        }
    }

    func browserViewControllerWasCancelled(_ browserViewController: MCBrowserViewController) {
        self.delegate?.dismissView(algo: "oi")
    }


}
```

Basicamente são os métodos acionados quando se aperta cancel ou done na tela de conexão dos aparelhos. No DidFinish se cria os avatares com posições aleatórias e adiciona como filho, cada jogador, na cena.

```
protocol SenderServiceManagerDelegate {
    func dismissView(algo: String)
    func command(maneger: SenderServiseManeger, action: String, peerId: String)
    func passPlayer(player:Player)
}
```

Protocolo do delegate que sera implementado na classe GameViewController.

```swift
import Foundation
import SpriteKit

protocol GameDelegate {
    func createPlayer(pos: CGPoint, id: String)
}
```

Protocolo do gameDelegate que será implementado na classe GameScene.

```swift
import UIKit
import SpriteKit
import GameplayKit
import Foundation

class GameViewController: UIViewController {

    var senderServise = SenderServiseManeger()
    var gameSceneDelegate: GameScene?

    override func viewDidLoad() {
        super.viewDidLoad()

        if let view = self.view as! SKView? {
            // Load the SKScene from 'GameScene.sks'
            if let scene = SKScene(fileNamed: "GameScene") {
                // Set the scale mode to scale to fit the window
                let gameScene = scene as! GameScene
                gameScene.scaleMode = .aspectFill
                gameScene.gameVC = self


                // Present the scene
                view.presentScene(gameScene)
            }

            view.ignoresSiblingOrder = true

            view.showsFPS = true
            view.showsNodeCount = true
        }
        senderServise.delegate = self
    }

    override var shouldAutorotate: Bool {
        return true
    }
```

Na GameViewControler se cria a variável senderService e ja instanciando. Com ele já se cria o objeto para a classe que irá manejar todo processo de conexão e recebimento de dados. gameSceneDelegate é uma variável do tipo GameScene, ela irá permitir instanciar a própria classe ViewController na variável existente na GameScene. Uma abstração seria criar uma ponte de sentido único para a GameScene.

```
    @IBAction func showConnectivityActions(_ sender: Any) {
        let actionSheet = UIAlertController(title: "ToDo Exchange", message: "Do you want to Host or Join a session?", preferredStyle: .actionSheet)

        actionSheet.addAction(UIAlertAction(title: "Host Session", style: .default, handler: { (action:UIAlertAction) in
            self.senderServise.host()
        }))


        actionSheet.addAction(UIAlertAction(title: "Join Session", style: .default, handler: { (action:UIAlertAction) in
            self.present(self.senderServise.join(), animated: true, completion: nil)
        }))

        actionSheet.addAction(UIAlertAction(title: "Disconect", style: .default, handler: { (action:UIAlertAction) in
            self.senderServise.disconect()
        }))


        actionSheet.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: nil))

        self.present(actionSheet, animated: true, completion: nil)

    }

    @IBAction func jumpTapped(_ sender: Any) {
        self.jump()
        senderServise.sendCommand(command: "jump")
    }

    func jump(){
        for (i, player) in (gameSceneDelegate?.players.enumerated())!{
            if player.playerId == self.senderServise.session.myPeerID.displayName {
                print(player.playerId as Any, self.senderServise.session.myPeerID.displayName)
                gameSceneDelegate?.players[i].jump()
            }
        }
    }


}
```

Na função show é feito um alert responsável para guiar o usuário que ação ele gostaria de realizar entre conectar, dar host ou desconectar. Cada escolha resulta numa chamada de função que se encontam na classe SenderService service. A função jumpTapped faz com que realize uma ação de pular para o próprio dispositivo e “fala” para o senderService mandar este comando como jump.
A função jump é para fazer com que o seu avatar pule no dispositivo de quem realizou a ação

```swift
extension GameViewController: SenderServiceManagerDelegate{

    func passPlayer(player: Player) {
        print("dando certo")
        self.gameSceneDelegate?.addChild(player.shape!)
        self.gameSceneDelegate?.players.append(player)
    }

    func command(maneger: SenderServiseManeger, action: String, peerId: String) {
        if action == "jump"{
            print("ah chave eh", peerId)
            for (i, player) in (gameSceneDelegate?.players.enumerated())!{
                if player.playerId == peerId {
                    gameSceneDelegate?.players[i].jump()
                }
            }
        }
    }

    func dismissView(algo: String) {
        self.dismiss(animated: true, completion: nil)
    }
}
```

Implementação dos protocolos na classe GameViewController. A função passPlayer é responsável por plotar os avatares de cada jogador na tela de todos que receberam o dado. Logo em seguida é salvo no array de players na classe GameScene.

A função command verifica qual comando foi realizado e qual avatar vai realizar esta ação. Está função será realizada para todos os dispositivos que receberem algum dado.

A função dismissView serve somente para encerrar a View da classe de busca do próprio MPC.

```swift
class GameScene: SKScene{

    var players: [Player] = []

    var gameVC = GameViewController()



    override func didMove(to view: SKView) {
        gameVC.senderServise.gameDelegate = self
        gameVC.gameSceneDelegate = self
        //players.append(Player(position: CGPoint(x: 0, y: 0), playerId: "123", parent: self))

    }



    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
            print("hi")
            //gameVC?.dismissView(algo: "keke")

        }

    override func update(_ currentTime: TimeInterval) {
        // Called before each frame is rendered
    }


}

extension GameScene: GameDelegate{
    func createPlayer(pos: CGPoint, id: String) {
        print("criando player")
        print("preparando para aplicar as posicoes", pos)
        self.players.append(Player(position: pos, playerId: id, parent: self))
        print("hayaa")
        gameVC.senderServise.sendPlayer(player: players.last!)



    }


}
```

Na variável se instancia um objeto do tipo GameViewController. Porém logo esse objeto será referenciado pela própria ViewController que instancia a GameScene.

Por sua vez a GameScene se referencia para os delegates do SenderService e da ViewController.

No extension se implementa o protocolo do GameDelegate. Nele é salvo o player no dispositivo de quem irá acionar a ação e salvar no array de players. Depois disso é enviado o player recém criado aos outros dispositivos que irão receber este player.

```swift
public class Player: NSObject, NSCoding, NSSecureCoding{
    public static var supportsSecureCoding: Bool{
        return true
    }



    var shape: SKSpriteNode?
    var playerId: String?

//    public init(position: CGPoint, playerId: String ,parent: SKNode?) {
//
//        shape = SKSpriteNode(imageNamed: "player")
//        shape?.position = position
//        shape?.zPosition = 2
//        shape?.size = CGSize(width: 120, height: 125)
//
//        if (parent != nil){
//            parent?.addChild(self.shape!)
//        }
//
//        self.playerId = playerId
//    }

    required convenience public init(coder decoder: NSCoder) {
        self.init()
        self.playerId = decoder.decodeObject(forKey: "playerId") as? String
        self.shape = decoder.decodeObject(forKey: "shape") as? SKSpriteNode
    }

    convenience init(position: CGPoint, playerId: String ,parent: SKNode?) {
        self.init()
        shape = SKSpriteNode(imageNamed: "player")
        shape?.position.x = position.x
        shape?.position.y = position.y
        print("A posicao eh", position.x, position)
        shape?.zPosition = 2
        shape?.size = CGSize(width: 120, height: 125)

        if (parent != nil){
            parent?.addChild(self.shape!)
        }

        self.playerId = playerId
    }
```

A classe player contem o id do jogador e a forma do objeto. O diferente dessa classe é que além de ser um objeto, ela é uma classe codificavel. Required convenience possibilita que cada variável da classe seja codificada.

Convenience init pode ser relacionado como o proprio init da classe.

```
59
60      public func encode(with aCoder: NSCoder) {
61          if let playerId = playerId { aCoder.encode(playerId, forKey: "playerId") }
62          if let shape = shape { aCoder.encode(shape, forKey: "shape") }
63      }
64
65
66      func jump(){
67          let sequence = SKAction.sequence([SKAction.moveBy(x: 0, y: 20, duration: 0.2),SKAction.moveBy(x: 0, y: -20, duration: 0.2)])
68          self.shape?.run(sequence)
69      }
70
71
72  }
```

Encode possibilita o objeto ser traduzido for descodificado, isso através de palavras chaves.

A função jump realiza a ação de pular do avatar.