Урок 8

Змейка

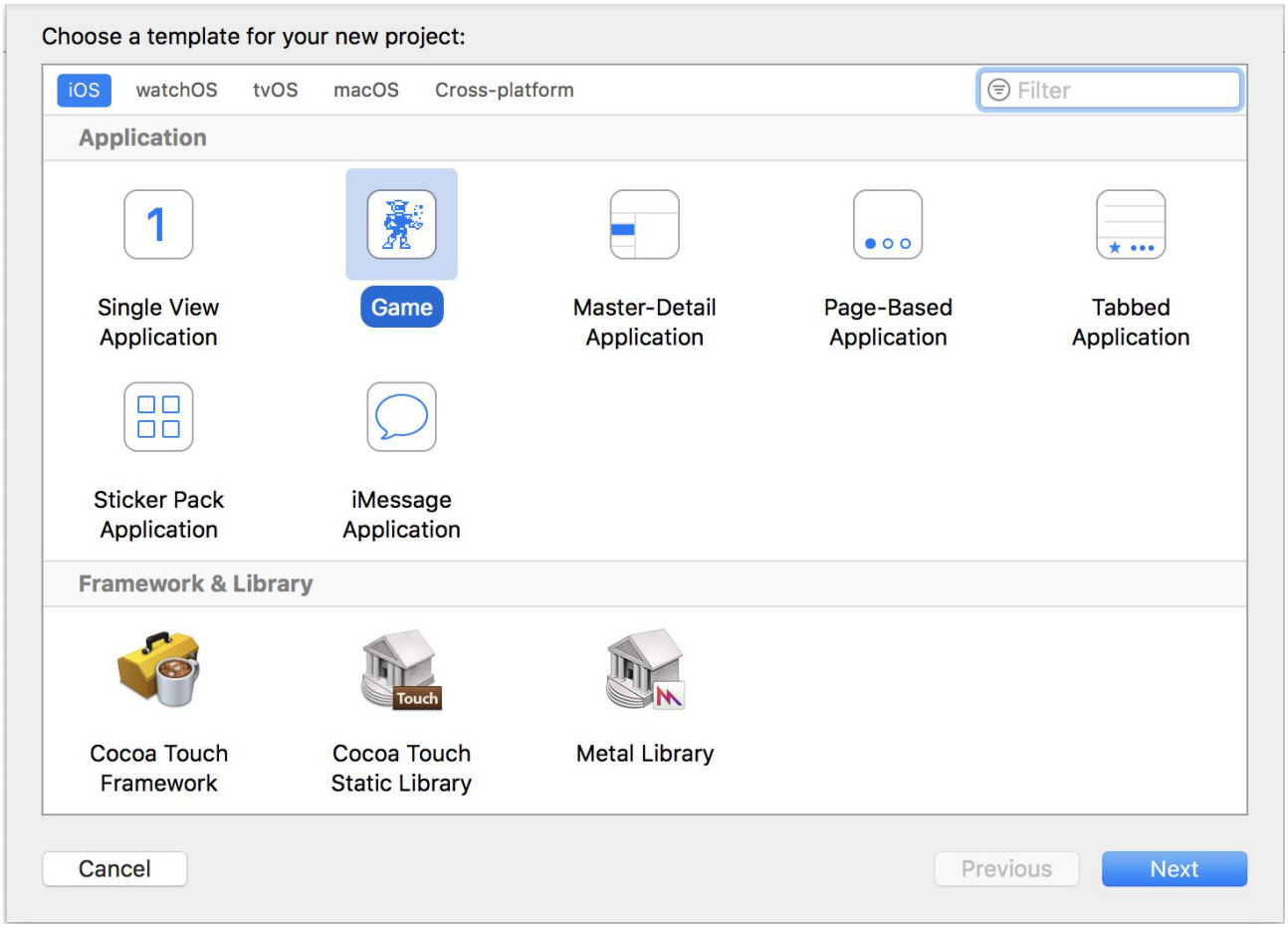
Пишем простую игру с использованием SpriteKit

Змейка

Создание приложения

Сегодня у нас заключительный урок и мы будем применять на практике знания, полученные за время курса.

Для того чтобы создать простую игру, давайте создадим приложение. При выборе шаблона на этот раз возьмем не “Single View application”, а “Game”. В остальном начальные шаги создания игры не отличаются от обычных приложений.



После создания приложения мы увидим достаточно много файлов:

- AppDelegate.swift - позволяет отслеживать события вашего приложения;
- GameScene.sks - файл сцены;
- Actions.sks - файл действия;
- GameScene.swift - класс сцены;
- GameViewController.swift - контроллер экрана;
- Main.storyboard - файл с экранами и переходами между ними;
- Assets.xcassets - каталог с изображениями;
- LaunchScreen.storyboard - файл с заставкой при загрузке приложения;
- Info.plist - настройки приложения.

С большинством из них мы познакомимся в следующем курсе, сейчас мы будем их просто игнорировать. Так, например, нам не нужно изменять AppDelegate.swift, Main.storyboard, Assets.xcassets, LaunchScreen.storyboard, Info.plist.

Настройка сцены

Область, где будут размещены все игровые объекты, называется сценой. Игра может состоять как из одной, так и из нескольких сцен. Например, вы можете разместить каждый игровой уровень на отдельной сцене. Для нашей игры достаточно одной сцены.

Сцену можно создать в графическом редакторе, но для закрепления знаний полученных во время прохождения курса мы будем делать это в коде. Для этого удалим файл GameScene.sks и Actions.sks.

Класс для сцены тоже был создан автоматически, но так как мы удалили файл с ее графическим представлением, нам необходимо вручную создать экземпляр сцены и разместить на экране телефона. За количество экранов и их содержимое отвечают UIViewController`ы. Нам достаточно одного и он тоже был для нас создан, мы просто добавим в него создание сцены. Откройте файл GameViewController.swift и измените метод viewDidLoad как показано ниже.

```
override func viewDidLoad() {
        super.viewDidLoad()
        // создаем экземпляр сцены
        let scene = GameScene(size: view.bounds.size)
        // получаем главную область экрана
        let skView = view as! SKView
        // включаем отображение fps (Количество кадров в секунду)
        skView.showsFPS = true
        // показывать количество объектов на экране
        skView.showsNodeCount = true
        // включает включаем произвольный порядок рендеринга объектов в узле
        skView.ignoresSiblingOrder = true
        // режим отображения сцены, растягивается на все доступное пространство
        scene.scaleMode = .resizeFill
        // добавляем сцену на экран
        skView.presentScene(scene)
    }
```

Метод viewDidLoad вызывается при подготовке к первому показу экрана. В нем мы создали сцену, настроили параметры ее отображения и добавили ее на экран.

Сцена, созданная по умолчанию, содержит много демонстрационного кода, который нам будет только мешать, давайте удалим его. Откройте файл GameScene.swift и измените его так, как показано ниже.

```
import SpriteKit
import GameplayKit
class GameScene: SKScene {
    // вызывается при первом запуске сцены
    override func didMove(to view: SKView) {
    }
    // вызывается при нажатии на экран
    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    }
    // вызывается при прекращении нажатия на экран
    override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
    }
   // вызывается при обрыве нажатия на экран, например ,если телефон примет звонок и свернет приложение
    override func touchesCancelled(_ touches: Set<UITouch>, with event: UIEvent?) {
    }
    // вызывается при обработке кадров сцены
    override func update(_ currentTime: TimeInterval) {
    }
}
```

Теперь у нас гораздо меньше методов и проще понять, что они делают. Обращаю ваше внимание, что все методы описаны в родительском классе, и наш игровой движок сам будет их вызывать. Мы же переопределим их и добавим свои инструкции.

didMove - вызывается в момент запуска сцены. Он предназначен для создания начального состояния и добавления объектов, необходимых нам на старте игры.

touchesBegan - метод обработки начала нажатия на экран. Он срабатывает в момент, когда палец прикоснулся к экрану.

touchesEnded - метод обработки прекращения нажатия на экран. Он срабатывает в момент, когда палец отрывается от экрана.

touchesCancelled - метод внезапного прекращения нажатия на экран. Он срабатывает, когда нажатия завершается не по воле пользователя, а под влиянием внешних факторов, например, если вам позвонят во время игры и игра свернется.

update - метод обновления сцены вызывается при каждом новом кадре.

Давайте еще раз изменим метод didMove, как показано ниже. Этим кодом мы внесем последние настройки в сцену. Добавим поддержку физики, выключим гравитацию, чтобы наша змейка не падала в низ экрана на землю, а могла перемещаться в любом направлении.

```
// вызывается при первом запуске сцены
   override func didMove(to view: SKView) {
// цвет фона сцены
       backgroundColor = SKColor.black
// вектор и сила гравитации
       self.physicsWorld.gravity = CGVector(dx: 0, dy: 0)
// добавляем поддержку физики
       self.physicsBody = SKPhysicsBody(edgeLoopFrom: frame)
// выключаем внешние воздействия на нашу игру
       self.physicsBody?.allowsRotation = false
// включаем отображение отладочной информации
       view.showsPhysics = true
   }
```

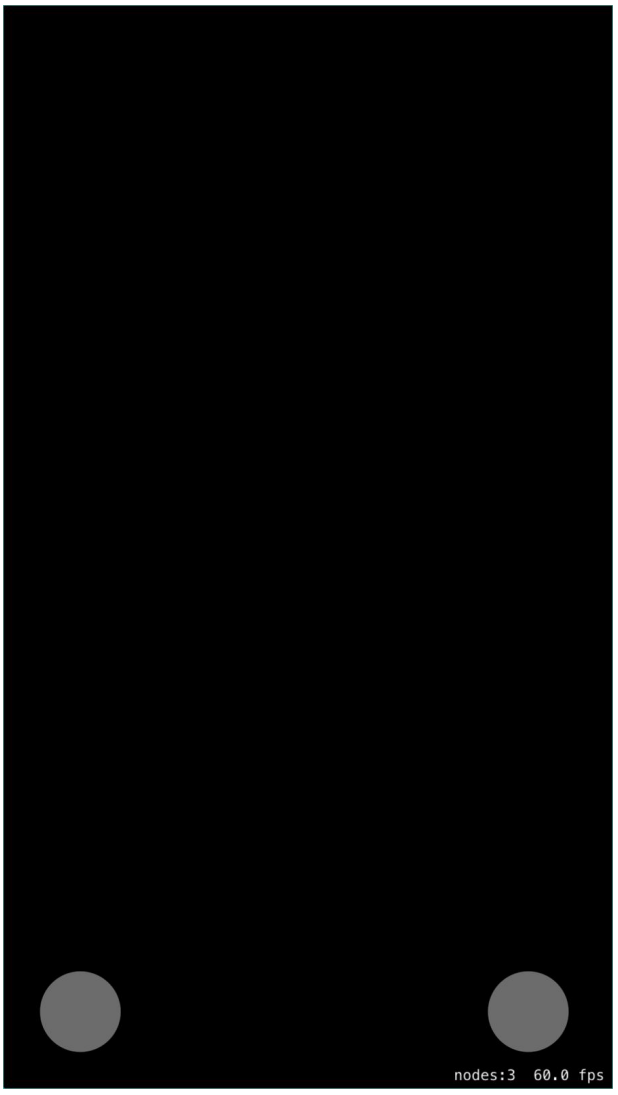
Размещение элементов на сцене.

Можете уже запустить нашу игру, но вы увидите пока только черный экран. Давайте добавим кнопки управления змейкой, одна будет поворачивать по часовой стрелке, вторая против.

```
// вызывается при первом запуске сцены
   override func didMove(to view: SKView) {
// цвет фона сцены
        backgroundColor = SKColor.black
// вектор и сила гравитации
        self.physicsWorld.gravity = CGVector(dx: 0, dy: 0)
// добавляем поддержку физики
        self.physicsBody = SKPhysicsBody(edgeLoopFrom: frame)
// выключаем внешние воздействия на нашу игру
        self.physicsBody?.allowsRotation = false
// включаем отображение отладочной информации
        view.showsPhysics = true
// поворот против часовой стрелки
// создаем ноду (объект)
        let counterClockwiseButton = SKShapeNode()
// задаем форму круга
        counterClockwiseButton.path = UIBezierPath(ovalIn: CGRect(x: 0, y: 0, width: 45, height: 45)).cgPath
// указываем координаты размещения
        counterClockwiseButton.position = CGPoint(x: view.scene!.frame.minX+30, y:
view.scene!.frame.minY+30)
// цвет заливки
        counterClockwiseButton.fillColor = UIColor.gray
// цвет рамки
        counterClockwiseButton.strokeColor = UIColor.gray
// толщина рамки
        counterClockwiseButton.lineWidth = 10
// имя объекта для взаимодействия
        counterClockwiseButton.name = "counterClockwiseButton"
// Добавляем на сцену
        self.addChild(counterClockwiseButton)
// Поворот по часовой стрелке
        let clockwiseButton = SKShapeNode()
        clockwiseButton.path = UIBezierPath(ovalIn: CGRect(x: 0, y: 0, width: 45, height: 45)).cgPath
        clockwiseButton.position = CGPoint(x: view.scene!.frame.maxX-80, y: view.scene!.frame.minY+30)
        clockwiseButton.fillColor = UIColor.gray
        clockwiseButton.strokeColor = UIColor.gray
        clockwiseButton.lineWidth = 10
        clockwiseButton.name = "clockwiseButton"
        self.addChild(clockwiseButton)
    }
```

Обратите внимание, что каждый игровой объект - это объект какого-либо класса. Например, наши кнопки - это объекты класса “SKShapeNode”. Вся настройка осуществляется посредством изменения значений свойств этого объекта.

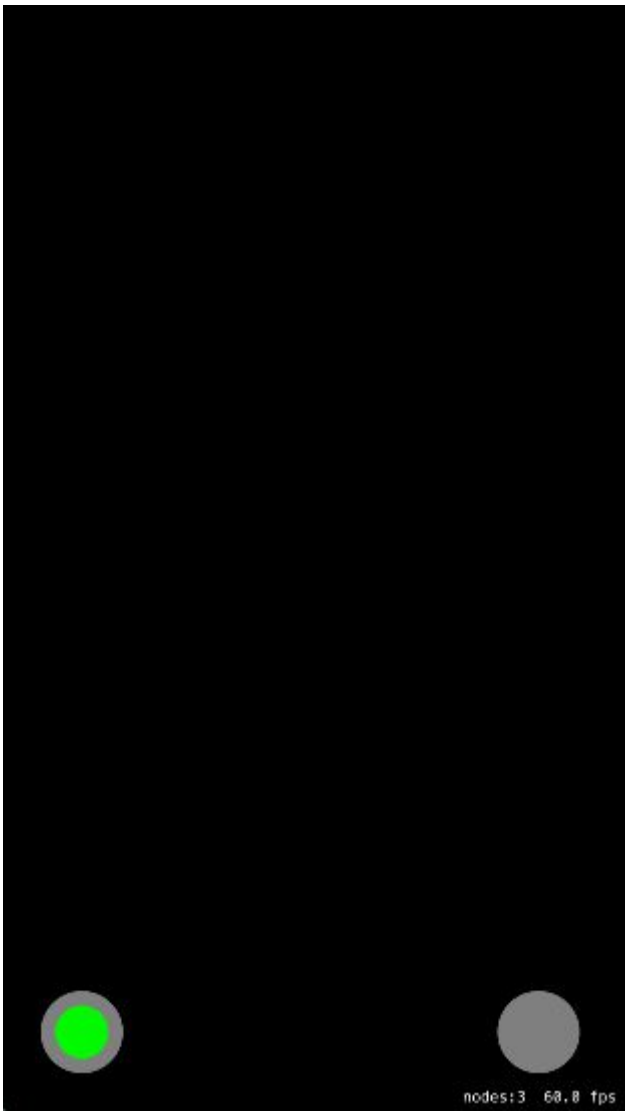
Теперь уже можно запустить приложение и увидеть кнопки. Они будут невзрачные и серые, и если на них нажать, ничего не произойдет.



Давайте добавим изменение цвета кнопок по нажатию, чтобы дать какую-то обратную связь для пользователя. С точки зрения игры наша кнопка это просто круг на экране. У нее нет методов, которые бы вызвали в момент нажатия, кнопка вообще не может определить, нажал ли на нее пользователь или нет. Поэтому мы будет следить за касаниями пальцами экрана. Когда палец будет прикасаться к экрану, мы будем проверять, не находится ли в этом месте наша кнопка. Если находится, то мы закрасим ее середину зеленым. По такому же принципу мы будем отслеживать событие прекращения нажатия пальцем. В момент прекращения касания мы вернем кнопке серый цвет.

```
// вызывается при нажатии на экран
    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
// перебираем все точки, куда прикоснулся палец
        for touch in touches {
// определяем координаты касания для точки
            let touchLocation = touch.location(in: self)
// проверяем, есть ли объект по этим координатам, и если есть, то не наша ли это кнопка
            guard let touchedNode = self.atPoint(touchLocation) as? SKShapeNode,
                touchedNode.name == "counterClockwiseButton" || touchedNode.name == "clockwiseButton"
else {
                return
            }
// если это наша кнопка, заливаем ее зеленой
            touchedNode.fillColor = .green
        }
    }
// вызывается при прекращении нажатия на экран
    override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
// повторяем все то же самое для действия, когда палец отрывается от экрана
        for touch in touches {
            let touchLocation = touch.location(in: self)

            guard let touchedNode = self.atPoint(touchLocation) as? SKShapeNode,
                touchedNode.name == "counterClockwiseButton" || touchedNode.name == "clockwiseButton"
else {
                return
            }
 // но делаем цвет снова серый
            touchedNode.fillColor = UIColor.gray
        }
    }
```



Давайте еще добавим в игру яблоко, которое будет кушать наша змейка. Для этого, во-первых, создадим класс Apple и в конструкторе, определим, как оно будет отрисовываться. Разместите этот код в отдельном файле. Хорошей практикой является классов размещение в отдельных файлах.

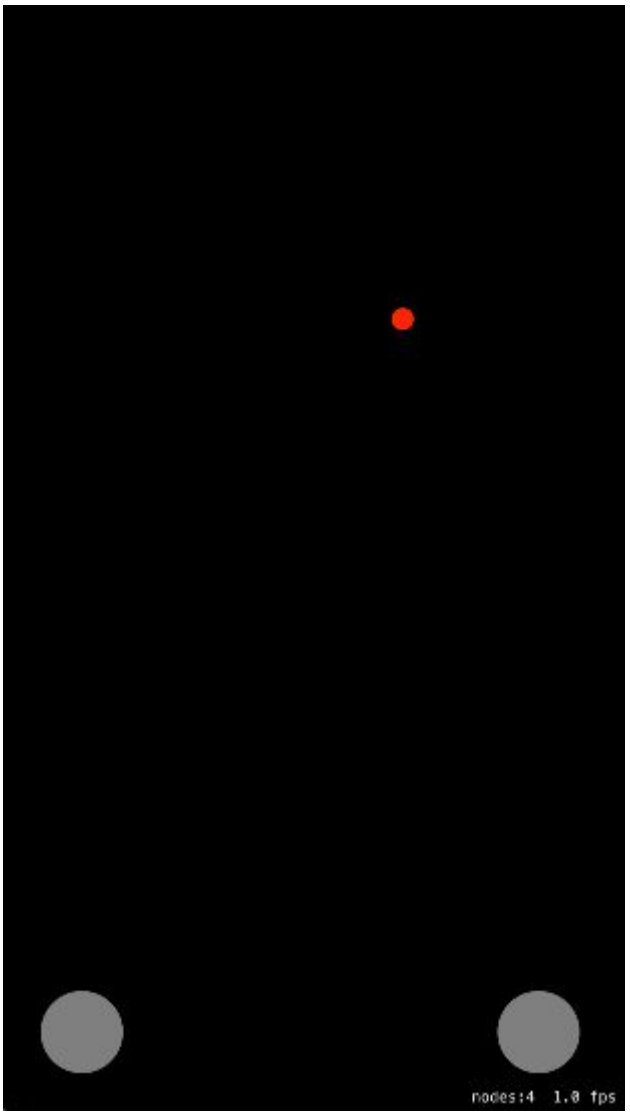
```
import UIKit
import SpriteKit
// Яблоко
class Apple: SKShapeNode {
/ /определяем, как оно будет отрисовываться
    convenience init(position: CGPoint) {
        self.init()
// рисуем круг
        path = UIBezierPath(ovalIn: CGRect(x: 0, y: 0, width: 10, height: 10)).cgPath
// заливаем красным
        fillColor = UIColor.red
// рамка тоже красная
        strokeColor = UIColor.red
// ширина рамки 5 поинтов
        lineWidth = 5
        self.position = position
    }
}
```

Во-вторых, в классе сцены добавим метод, который будет создавать яблоко в случайной точке экрана. Код ниже надо разместить в классе GameScene в файле GameScene.swift

```
// Создаем яблоко в случайной точке сцены
    func createApple(){
// Случайная точка на экране
        let randX  = CGFloat(arc4random_uniform(UInt32(view!.scene!.frame.maxX-5)) + 1)
        let randY  = CGFloat(arc4random_uniform(UInt32(view!.scene!.frame.maxY-5)) + 1)
// Создаем яблоко
        let apple = Apple(position: CGPoint(x: randX, y: randY))
// Добавляем яблоко на сцену
        self.addChild(apple)
    }
```

И добавим вызов этого метода в методе didMove, для создания яблока на старте игры.

```
// вызывается при первом запуске сцены
    override func didMove(to view: SKView) {
        <...>
        createApple()
    }
```



Вот и наше яблоко. Теперь добавим змейку. Она будет состоять из трех классов, размещенных в трех разных файла. Мы опишем сегмент ее тела, голову и всю змейку целиком.

Начнем с сегмента тела. Создайте файл SnakeBodyPart.swift и поместите в него следующий код. Пока он мало чем отличается от яблока.

```
import UIKit
import SpriteKit
class SnakeBodyPart: SKShapeNode {
    let diameter = 10.0
    // добавляем конструктор
    init (atPoint point: CGPoint){
        super.init()
   // рисуем круглый элемент
        path = UIBezierPath(ovalIn: CGRect(x: 0, y: 0, width: CGFloat(diameter), height:
CGFloat(diameter))).cgPath
   // цвет рамки и заливки зеленый
        fillColor = UIColor.green
        strokeColor = UIColor.green
   // ширина рамки 5 поинтов
        lineWidth = 5
   // размещаем элемент в переданой точке
        self.position = point
    }
    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

Теперь зададим файл SnakeHead.swift и разместим там код головы змейки. Он будет наследоваться от класса сегмента тела. И пока ничем не отличается от своего родителя. В дальнейшем мы научим ее кушать яблоки.

```
import UIKit
class SnakeHead: SnakeBodyPart {
    override init(atPoint point: CGPoint){
        super.init(atPoint:point)
    }
    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

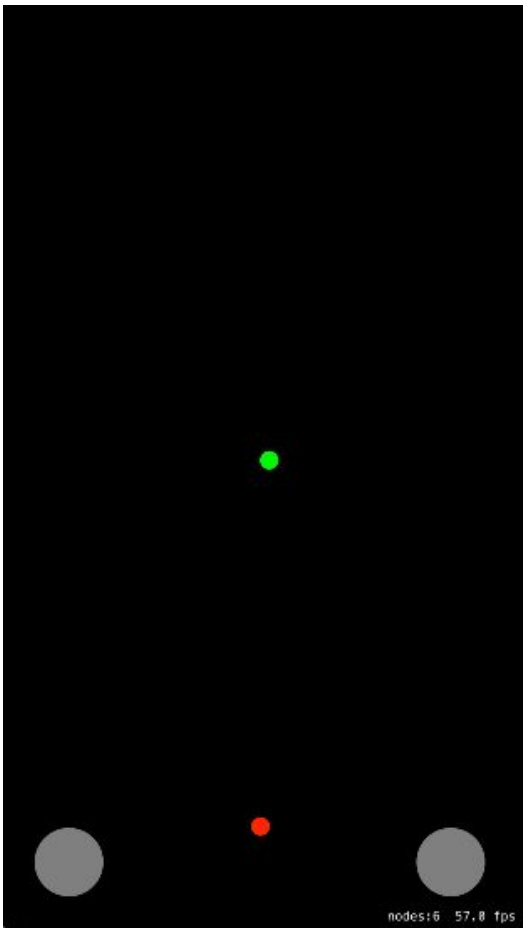
Теперь осталось добавить класс самой змейки. Создаем файл Snake.swift и разместим там следующий код.

```
import UIKit
import SpriteKit
// сама змейка
class Snake: SKShapeNode {
// массив где хранятся сегменты тела
   var body = [SnakeBodyPart]()
// конструктор
   convenience init(atPoint point: CGPoint) {
      self.init()
// змейка начинается с головы, создадим ее
      let head = SnakeHead(atPoint: point)
// и добавим в массив
      body.append(head)
// и сделаем ее дочерним объектом.
      addChild(head)
   }
// метод добавляет еще один сегмент тела
   func addBodyPart(){
// инстанцируем сегмент
      let newBodyPart = SnakeBodyPart(atPoint: CGPoint(x: body[0].position.x, y: body[0].position.y))
// добавляем его в массив
      body.append(newBodyPart)
// делаем дочерним объектом
      addChild(newBodyPart)
   }
}
```

Пока тоже ничего особенного. Это еще одна нода, в которой есть массив, хранящий сегменты змеи. Конструктор, создающий голову змеи, и метод для создания новых сегментов, фактически, этот метод увеличивает нашу змею. Обратите внимание, что самам змейка это нода (SKShapeNode), которую можно разместить на сцене. Голова и сегменты тела, это тоже ноды, но мы не будем добавлять их на сцену напрямую. Мы добавим их внутрь ноды змейки с помощью метода “addChild”, таким образом, они будут существовать в ее координатном пространстве. Когда мы добавим змейку на сцену, все ее дочерние ноды тоже добавляются вместе с ней.

Теперь нам осталось добавить змейку на сцену. Во-первых, давайте заведем свойство для хранения змеи. Во-вторых, добавим в методе didMove ее создание.

```
class GameScene: SKScene {
   // наша змея
   var snake: Snake?
   // вызывается при первом запуске сцены
   override func didMove(to view: SKView) {
     <...>
   // создаем змею по центру экрана и добавляем ее на сцену
      snake = Snake(atPoint: CGPoint(x: view.scene!.frame.midX, y: view.scene!.frame.midY))
      self.addChild(snake!)
   }
```



На этом наша сцена готова. На ней есть кнопки, змея и яблоко? которое она должна съесть, нам осталось только вдохнуть жизнь в нашу игру.

Перемещение объектов

Давайте напишем метод перемещения нашей змейки. Перемещение будет проходить по следующему алгоритму. Для начала мы рассчитаем точку, куда переместится голова змеи. Мы введем две переменных “moveSpeed” - константу, определяющую расстояние смещения и “angle” - определяющую смещение точки для движения в различных направлениях. Вычисляя синус и косинус от заданного угла, мы будем определять направление движения, а умножая на “moveSpeed”, мы определим расстояние, на которое необходимо сдвинуть точку. После расчета мы просто переместим голову в новую точку. Остальные сегменты нашей змеи будут перемещаться намного проще, каждый сегмент будет смещаться к точке, где находится предыдущий. Например, первый сегмент будет смещен к голове, второй к первому и так далее. Когда наш метод будет готов, мы просто добавим его вызов в методе “update” нашей сцены.

Измените класс “Snake” следующим образом.

```
// сама змея
class Snake: SKShapeNode {
// скорость перемещения
   let moveSpeed = 125.0
// угол, необходимый для расчета направления
   var angle: CGFloat = 0.0
   <...>
// перемещаем змейку
   func move(){
// если у змейки нет головы то ничего не перемещаем
      guard !body.isEmpty else { return }
// перемещаем голову
      let head = body[0]
      moveHead(head)
// перемещаем все сегменты тела
      for index in (0..<body.count) where index > 0 {
         let previousBodyPart = body[index-1]
         let currentBodyPart = body[index]
         moveBodyPart(previousBodyPart, c: currentBodyPart)
      }
   }
// перемещаем голову
   func moveHead(_ head: SnakeBodyPart){
// рассчитываем смещение точки
      let dx = CGFloat(moveSpeed) * sin(angle);
      let dy = CGFloat(moveSpeed) * cos(angle);
// смещаем точку назначения головы
      let nextPosition = CGPoint(x: head.position.x + dx, y: head.position.y + dy)
// действие перемещения головы
      let moveAction = SKAction.move(to: nextPosition, duration: 1.0)
// запуск действия перемещения
      head.run(moveAction)
   }
/ /перемещаем сегмента змеи
   func moveBodyPart(_ p: SnakeBodyPart, c: SnakeBodyPart){
      //перемещаем текущий элемент к предыдущему
      let moveAction = SKAction.move(to: CGPoint(x: p.position.x, y: p.position.y), duration: 0.1 )
  // запуск действия перемещения
      c.run(moveAction)
   }
}
```

И теперь вызовем метод движения в методе обновления нашей сцены. Таким образом на каждом новом кадре наша змейка будет менять свою позицию на экране.

```
// вызывается при обработке кадров сцены
   override func update(_ currentTime: TimeInterval) {
      snake!.move()
   }
```

Уже можно запустить игру и посмотреть, как голова нашей змеи улетает за край экрана. Время добавить интерактивности. Добавим два метода для поворота. Фактически, они будут просто менять переменную “angle”. И настроим вызов этих методов при нажатии на кнопки.

Итак, добавим два метода в класс “snake”.

```
// поворот по часовой стрелке
   func moveClockwise(){
// смещаем угол на 45 градусов
      angle += CGFloat(Double.pi/2)
   }
// поворот против часовой стрелки
   func moveCounterClockwise(){
      angle -= CGFloat(Double.pi/2)
   }
```

И дополним код нажатия на кнопку в классе нашей сцены.

```
// вызывается при нажатии на экран
   override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
// перебираем все точки, куда прикоснулся палец
      for touch in touches {
// определяем координаты касания для точки
         let touchLocation = touch.location(in: self)
// проверяем, есть ли объект по этим координатам, и если есть, то не наша ли это кнопка
         guard let touchedNode = self.atPoint(touchLocation) as? SKShapeNode,
               touchedNode.name == "counterClockwiseButton" || touchedNode.name == "clockwiseButton"
else {
               return
         }
// если это наша кнопка, заливаем ее зеленой
         touchedNode.fillColor = .green
// определяем, какая кнопка нажата и поворачиваем в нужную сторону
         if touchedNode.name == "counterClockwiseButton" {
            snake!.moveCounterClockwise()
         } else if touchedNode.name == "clockwiseButton" {
            snake!.moveClockwise()
         }
      }
   }
```

Взаимодействие объектов

Запустим приложение и теперь мы можем управлять нашей змейкой, но если мы попытаемся съесть яблоко, то просто пролетим сквозь него. Тоже самое касается и стен, мы можем легко проходить сквозь стены и уходить за пределы экрана. Дело в том, что мы не обрабатываем взаимодействие наших объектов.

Вся физика сцены сейчас обрабатывается в классе “SKPhysicsWorld”, его экземпляр доступен через свойство сцены “physicsWorld”. Для того чтобы наш физический мир начал обрабатывать соприкосновения, необходимо установить для него “contactDelegate”. В нашем случае в роли делегата будет выступать сама сцена. Для этого имплементируем классу сцены протокол “SKPhysicsContactDelegate”. Этот протокол описывает всего два метода “didBegin” - срабатывает при начале соприкосновения и “didEnd” - срабатывает при завершении соприкосновения. Нам будет достаточно только метода “didBegin”. Давайте добавим его через расширение класса сцены.

```
// Имплементируем протокол
extension GameScene: SKPhysicsContactDelegate {
// Добавляем метод отслеживания начала столкновения
    func didBegin(_ contact: SKPhysicsContact) {
    }
}
```

И установим делегат “contactDelegate” для сцены. Добавим строку в метод didMove в классе сцены.

```
// Вызывается при первом запуске сцены
    override func didMove(to view: SKView) {
        <...>
// Делаем нашу сцену делегатом соприкосновений
        self.physicsWorld.contactDelegate = self
        <...>
    }
```

Теперь мы готовы отслеживать соприкосновения, но наши объекты пока только изображения и не имеют физического тела (“physicsBody”). Необходимо его добавить яблоку, голове и сегментам тела змеи.

```
// Яблоко
class Apple: SKShapeNode {
// Определяем как оно будет рисоваться
    convenience init(position: CGPoint) {
        self.init()
        <...>
 // Добавляем физическое тело, совпадающее с изображением яблока
        self.physicsBody = SKPhysicsBody(circleOfRadius: 10.0, center:CGPoint(x:5, y:5))
    }
}
```

Но физическое тело мало создать, надо еще и присвоить ему категорию “categoryBitMask” и категории, с которыми будет взаимодействовать “contactTestBitMask“.

Оба эти значения - битовые маски. Категории нужны для разделения всех объектов, находящихся на сцене на группы. Можно указывать, какие группы могут взаимодействовать между собой, а какие не могут. Второе свойство “contactTestBitMask“ как раз содержит результат логического ИЛИ (“|”) между несколькими категориями, с которыми будет взаимодействовать наш объект.

Давайте создадим структуру с несколькими категориями для наших объектов. Добавьте код, приведенный ниже в тот же файл, где и хранится наша сцена.

```
// Категория пересечения объектов
struct CollisionCategories{
// Тело змеи
    static let Snake: UInt32 = 0x1 << 0
// Голова змеи
    static let SnakeHead: UInt32 = 0x1 << 1
// Яблоко
    static let Apple: UInt32 = 0x1 << 2
// Край сцены (экрана)
    static let EdgeBody:   UInt32 = 0x1 << 3
}
```

Теперь, когда у нас появились категории, мы можем настроить все объекты. Начнем со сцены.

```
class GameScene: SKScene {
// наша змея
    var snake: Snake?
// вызывается при первом запуске сцены
    override func didMove(to view: SKView) {
        <...>
// устанавливаем категорию взаимодействия с другими объектами
        self.physicsBody?.categoryBitMask = CollisionCategories.EdgeBody
// устанавливаем категории, с которыми будут пересекаться края сцены
        self.physicsBody?.collisionBitMask = CollisionCategories.Snake | CollisionCategories.SnakeHead
        <...>
```

Теперь займемся яблоком и телом змеи.

```
// Яблоко
class Apple: SKShapeNode {
// Определяем, как оно будет рисоваться
    convenience init(position: CGPoint) {
        self.init()
       <...>
// Добавляем физическое тело, совпадающее с изображением яблока
        self.physicsBody = SKPhysicsBody(circleOfRadius: 10.0, center:CGPoint(x:5, y:5))
// Категория - яблоко
        self.physicsBody?.categoryBitMask = CollisionCategories.Apple
    }
}
```

```
class SnakeBodyPart: SKShapeNode {
    let diameter = 10.0
// Добавляем конструктор
    init (atPoint point: CGPoint){
        super.init()
        <...>
// Создаем физическое тело
            self.physicsBody = SKPhysicsBody(circleOfRadius: CGFloat(diameter - 4), center: CGPoint(x: 5,
y:5))
// Может перемещаться в пространстве
        self.physicsBody?.isDynamic = true
// Категория - змея
        self.physicsBody?.categoryBitMask = CollisionCategories.Snake
// пересекается с границами экрана и яблоком
        self.physicsBody?.contactTestBitMask = CollisionCategories.EdgeBody | CollisionCategories.Apple
    }
}
```

И нам осталось модифицировать только голову. Так как голова наследуется от тела, создавать физическое тело не нужно, надо только установить категорию и маску пересечения.

```
class SnakeHead: SnakeBodyPart {
    override init(atPoint point: CGPoint){
        super.init(atPoint:point)
// категория - голова
        self.physicsBody?.categoryBitMask = CollisionCategories.SnakeHead
// пересекается с телом, яблоком и границей экрана
         self.physicsBody?.contactTestBitMask = CollisionCategories.EdgeBody | CollisionCategories.Apple |
CollisionCategories.Snake
    }
    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

После создания физических тел и расстановки масок наши объекты, наконец, готовы к соприкосновениям. Давайте напишем метод делегата “didBegin”.

```
func didBegin(_ contact: SKPhysicsContact) {
// логическая сумма масок соприкоснувшихся объектов
        let bodyes = contact.bodyA.categoryBitMask | contact.bodyB.categoryBitMask
// вычитаем из суммы голову змеи и у нас остается маска второго объекта
        let collisionObject = bodyes ^ CollisionCategories.SnakeHead
// проверяем, что это за второй объект
        switch collisionObject {
        case CollisionCategories.Apple: // проверяем что это яблоко
// яблоко это один из двух объектов, которые соприкоснулись. Используем тернарный оператор, чтобы вычислить какой именно
            let apple = contact.bodyA.node is Apple ? contact.bodyA.node : contact.bodyB.node
// добавляем к змее еще одну секцию
            snake?.addBodyPart()
// удаляем съеденное яблоко со сцены
            apple?.removeFromParent()
// создаем новое яблоко
            createApple()
        case CollisionCategories.EdgeBody: // проверяем, что это стенка экрана
            break                          // соприкосновение со стеной будет домашним заданием
        default:
            break
        }
    }
```

Готово! Можно запустить игру и даже поиграть в нее. Единственное, что в нашу игру невозможно проиграть. Змейка свободно проходит сквозь свой хвост и стенки экрана. Попробуйте самостоятельно реализовать поражение при соприкосновении со стенкой экрана и запуск игры сначала.

Домашнее задание

1. Написать рестарт игры при соприкосновении со стеной.

Дополнительные материалы

1. https://developer.apple.com/spritekit/
2. https://habrahabr.ru/post/225517/

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://developer.apple.com/spritekit/