

Desenvolvimento de Jogos para iOS

Explore sua imaginação com o framework Cocos2D



Casa do
Código

MAURICIO TOLLIN
RODRIGO GOMES
ANDERSON LEITE

Sumário

1	Introdução ao desenvolvimento de jogos no iOS	1
1.1	O que você encontrará neste livro	3
1.2	Que comece a diversão!	8
2	Protótipo de um jogo	9
2.1	Iniciando o projeto	11
2.2	Criando a base do jogo	14
2.3	Desenhando o objeto principal	20
2.4	Captando os comandos do usuário e movendo objetos	24
2.5	Criando o inimigo	29
2.6	Detectando colisões e mostrando resultados	32
2.7	Adicionando um placar	40
2.8	Criando botões de interface do usuário	42
2.9	Adicionando mais vida: imagens da nave e do céu	46
2.10	Conclusão	49
3	História do jogo	51
3.1	14-bis	52
3.2	14-bis VS 100 Meteoros	53
4	Tela inicial: Lidando com Background, logo e botões de menu	57
4.1	Sobre o Cocos2D	59
4.2	Iniciando o projeto	59
4.3	ajustando a orientação	62
4.4	Background	64

4.5	Assets da Tela de abertura	67
4.6	Capturando configurações iniciais do dispositivo	68
4.7	Logo	71
4.8	Botões	72
4.9	Conclusão	74
5	Tela do jogo e objetos inimigos	77
5.1	GameScene	78
5.2	Transição de telas	80
5.3	Engines	81
5.4	Meteor	85
5.5	Tela do game	87
5.6	Conclusão	90
6	Criando o Player	93
6.1	Desenhando o Player	94
6.2	Botões de controle	98
6.3	Atirando	101
6.4	Movendo o player	107
6.5	Conclusão	109
7	Detectando colisões, pontuando e criando efeitos	111
7.1	Detectando colisões	112
7.2	Efeitos	115
7.3	Player morre	122
7.4	Placar	123
7.5	Conclusão	126
8	Adicionando sons e música	127
8.1	Executando sons	128
8.2	Cache de sons	129
8.3	Música de fundo	130
8.4	Conclusão	131

9	Voando com a gravidade!	133
9.1	Usando o Acelerômetro	134
9.2	Controlando a instabilidade	141
9.3	Calibrando a partir da posição inicial do aparelho	142
9.4	Desafios com o acelerômetro	144
9.5	Conclusão	144
10	Tela final e game over	147
10.1	Tela final	148
10.2	Tela Game Over	152
10.3	Conclusão	156
11	Pausando o jogo	157
11.1	Montando a tela de pause	158
11.2	Controlando o Game Loop	161
11.3	Adicionando o botão de pause	162
11.4	A interface entre jogo e pause	163
11.5	Pausando o jogo	163
11.6	Pausando os objetos	168
11.7	Conclusão	170
12	Continuando nosso jogo	173
12.1	Utilizando ferramentas sociais	173
12.2	Highscore	174
12.3	Achievements	176
12.4	Desafios para você melhorar o jogo	177
12.5	Como ganhar dinheiro?	178
12.6	Conclusão	180

CAPÍTULO 1

Introdução ao desenvolvimento de jogos no iOS

River Raid, para Atari, foi provavelmente o primeiro jogo de videogame que muitos jogaram. Nesse clássico game da `Activision` criado em 1982, o jogador controlava uma nave que se movia de baixo para cima na tela, ganhando pontos por matar inimigos, destruir helicópteros, naves e balões. E mais: era possível encher o tanque passando por estações de gás.



Figura 1.1: RIVER RAID no Atari

Incrível como um desenho simples e 2D podia ser tão divertido. Controlar a nave, fazer pontos e passar por obstáculos garantiam horas de diversão.

Com o passar do tempo, novos jogos foram surgindo e se tornaram cada vez mais sofisticados. Apesar de todos os conceitos dos jogos antigos terem sido mantidos, um jogo de Playstation 3, por exemplo, pode envolver dezenas de desenvolvedores.

Atualmente, com o crescimento dos *casual gamers*, os celulares e tablets se tornaram plataformas de sucessos e disputadas. Com eles, o desenvolvimento de um jogo não precisa mais de uma quantidade enorme de desenvolvedores. Uma ideia interessante e bem implementada pode ser o suficiente para seu jogo obter sucesso. Só depende de você.

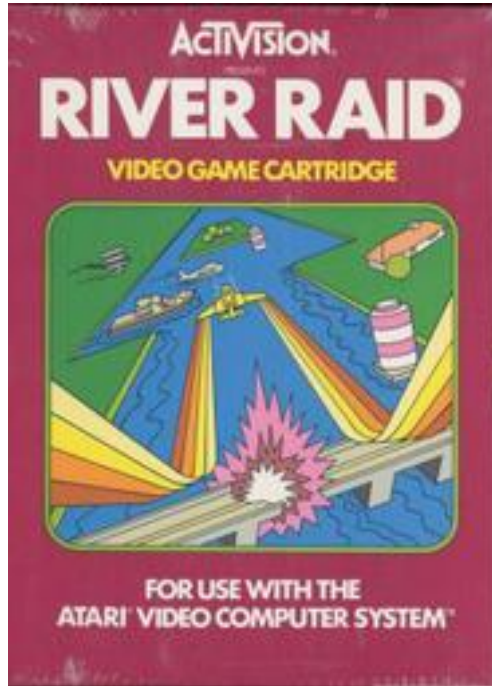


Figura 1.2: Capa do jogo RIVER RAID de 1982

1.1 O QUE VOCÊ ENCONTRARÁ NESTE LIVRO

Este livro é escrito para desenvolvedores que já conhecem o básico de desenvolvimento de aplicativos para iOS. Ele é dividido em 3 partes principais:

- Um protótipo inicial
- Um estudo do jogo que será desenvolvido
- Um jogo desenvolvido com Cocos2D

A ideia é que seja um bom guia para todos aqueles que querem iniciar no desenvolvimento de games, seja profissionalmente, para evoluir seus conhecimentos ou mesmo por pura diversão.

O que é um Desenvolvedor Apple?

O Xcode é a plataforma de desenvolvimento de aplicativos iOS. Qualquer pessoa pode baixá-lo e começar a desenvolver um aplicativo, testando-o no simulador que vem junto do próprio Xcode. Entretanto, para executar um aplicativo em seu aparelho iPhone / iPad ou publicá-lo na App Store, deve-se ser um Desenvolvedor Apple registrado no “iOS Developer Program” (mais em <https://developer.apple.com/programs/ios/>).

No capítulo 2 deste livro falaremos sobre como baixar e instalar o Xcode.

Um protótipo inicial

No início do livro, será desenvolvido um jogo simples, programado com apenas 2 classes. O objetivo é se familiarizar e ter uma noção geral dos conceitos básicos no desenvolvimento de games. Esses conceitos aparecem em quase todos os jogos, sejam eles simples ou avançados.

Nesse capítulo não será utilizado nenhum *framework* adicional de desenvolvimento, apenas os *frameworks* padrões de qualquer aplicativo de iOS, incluídos automaticamente pelo Xcode na criação de um novo projeto. Mesmo assim, chegaremos a um resultado bem interessante, como esse:

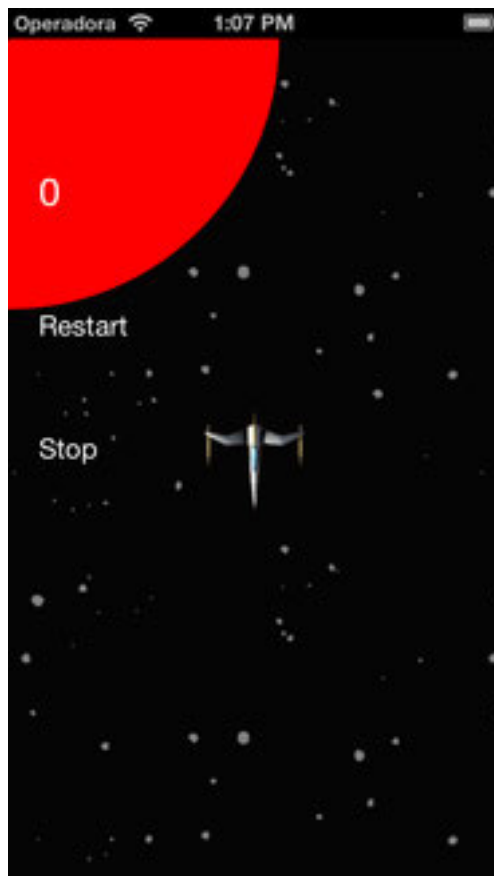


Figura 1.3: Imagem do nosso protótipo.

O código do nosso protótipo pode ser encontrado aqui:

https://github.com/BivisSoft/jogos_ios_prototipo

Um estudo do jogo que será desenvolvido

Programação é apenas uma parte do desenvolvimento de games. Empresas focadas em desenvolvimento de jogos possuem roteiristas para criar a história dos games, designers para definir o melhor visual do jogo, profissionais de som para a trilha sonora e efeitos, designers de interface para definir como será a experiência do jogador no game, entre outros. O marketing e divulgação são casos à parte.

Teremos um capítulo especial para planejar um pouco a história do jogo, determinar as transições de tela e estudar o visual do jogo a ser desenvolvido, que será nessa direção:

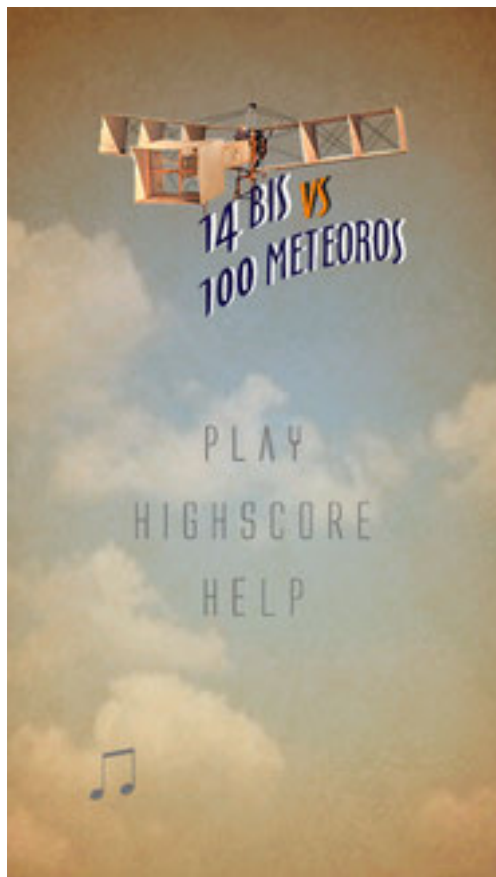


Figura 1.4: 14 bis VS 100 Meteoros

Também veremos um pouco sobre como deixar o jogo viciante e poder ganhar dinheiro com itens, missões e upgrades.

Um jogo desenvolvido com Cocos2D

Quando os principais conceitos já tiverem sido passados e a história e planejamento do jogo finalizada, iniciaremos o desenvolvimento do nosso jogo principal. Para ele, utilizaremos um *framework* chamado Cocos2D, que facilita e otimiza di-

versas questões usuais no desenvolvimento de jogos.

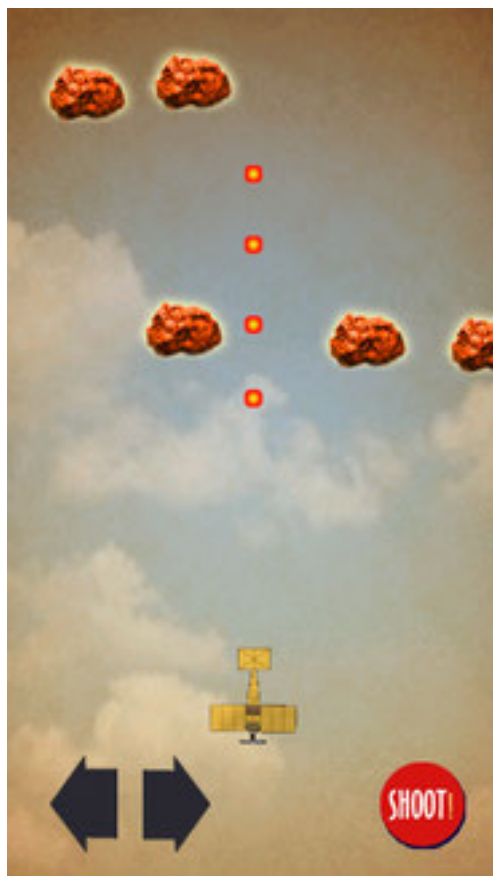


Figura 1.5: 14 bis VS 100 Meteoros

O código do jogo completo com Cocos2D está disponível em:

https://github.com/BivisSoft/jogos_ios_14bis

Grupo de Discussão

Existe um grupo de discussão focado exclusivamente para os exemplos que serão desenvolvidos aqui. Caso você tenha dúvidas em algum passo, ou mesmo venha a implementar modificações e criar o seu próprio jogo com o que aprendeu, compartilhe!

<https://groups.google.com/group/desenvolvimento-de-jogos-para-ios>

Caso tenha uma conta de Desenvolvedor Apple, você também pode utilizar o fórum de Desenvolvedores Apple para resolver suas dúvidas:

<https://developer.apple.com/>

1.2 QUE COMECE A DIVERSÃO!

Este livro vai te dar a base para criar um jogo! Você saberá por onde começar e terá os principais conceitos e a forma de pensar necessária para desenvolver um game 2D ao final desta leitura. A partir disso, é a sua própria criatividade e determinação que poderão fazer de suas ideias o novo jogo de sucesso no mundo dos games!

CAPÍTULO 2

Protótipo de um jogo

Vamos começar a desenvolver um jogo! Este será um capítulo fundamental para todo o livro, focado em conceitos importantes, ilustrando com muita prática. Nele percorreremos as principais etapas que precisamos ter em mente ao desenvolver um jogo.

Com os conceitos desse capítulo poderemos desenvolver jogos bem interessantes, porém, o objetivo agora é explorarmos as mecânicas por trás dos games e sermos apresentados à forma de pensar necessária.

Para percorrer esse caminho, iniciaremos criando um protótipo. Criar um protótipo será bom pelas seguintes razões:

- Conseguiremos um rápido entendimento da visão geral necessária para desenvolver um game.
- Não precisaremos nos preocupar com criar diversas telas que um jogo pode ter, permitindo focar apenas nos conceitos importantes.

- Permitirá entrar em detalhes mais complexos quando de fato iniciarmos nosso game.

Nosso protótipo terá as funcionalidades básicas encontradas nos games, vamos conhecer os objetivos.

Funcionalidades do protótipo

Pense em um jogo 2D tradicional como Super Mario Bros ou mesmo Street Fighter. Eles possuem uma série de semelhanças. Em ambos você controla algum elemento, que podemos chamar de Player. O player recebe algum tipo de estímulo (*input*) para executar movimentos na tela, como teclado, joystick ou mouse. Após os *inputs* o player pode ganhar pontos se algo acontecer, normalmente associado a encostar em outro objeto do jogo, o que faz com que algum placar seja atualizado. Em determinado momento o player pode ganhar ou perder o jogo, por diversos motivos, como superar um tempo, ultrapassar uma marca de pontos ou encostar em algum outro objeto do game.

Essas são as mecânicas básicas de qualquer jogo. Pense em outro jogo com as características semelhantes e tente fazer esse paralelo. No protótipo que criaremos nesse capítulo, implementaremos essas mecânicas, entendendo como desenvolvê-las em um aplicativo iOS.

Nosso jogo terá as seguintes funcionalidades:

- Um player que será representado por uma circunferência verde, posteriormente, a nave.
- Mover o player de acordo com um estímulo, no caso, o toque na tela (*input*).
- Um inimigo que será representado por uma circunferência que aumentará com o passar do tempo.
- Um placar que será atualizado de acordo com o tempo no qual o player não é capturado pelo inimigo.
- Game Over quando o inimigo encostar no player
- Opções de *restart* e *exit*

Ao fim desse capítulo, teremos o protótipo abaixo.

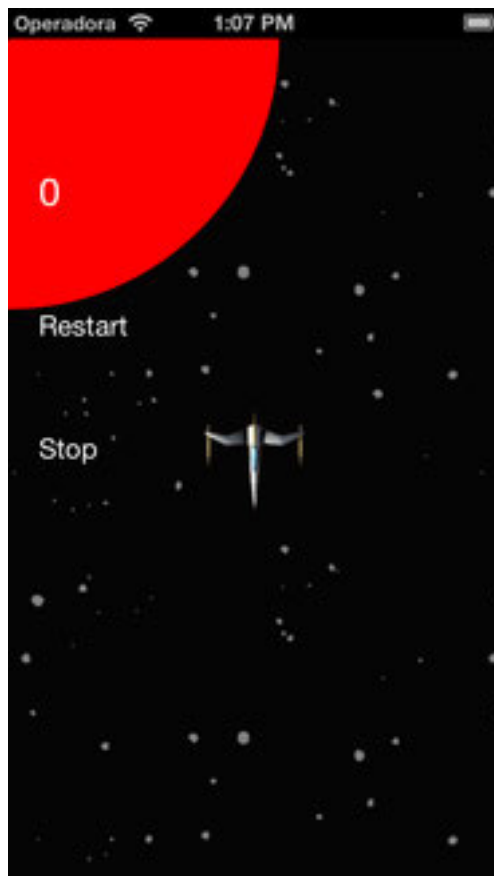


Figura 2.1: Imagem do jogo.

Temos muito para percorrer nesse protótipo. Repare que ao entender a lógica por trás de um jogo, poderemos criar qualquer tipo de game. Vamos ao protótipo!

2.1 INICIANDO O PROJETO

Vamos iniciar criando um projeto comum de iOS. Como você já criou algum aplicativo iOS, perceberá que o procedimento é o mesmo. Não é necessário configurar nada específico para este protótipo ao criar o projeto. Lembre-se que você precisa ter o Xcode instalado, que pode ser baixado diretamente pela App Store de seu computador ou utilizando o seguinte link:

<http://itunes.apple.com/br/app/xcode/id497799835>

Esse é um livro focado em quem já conhece o básico do desenvolvimento de aplicativos iOS, mas mesmo assim passaremos passo a passo em alguns pontos e revisaremos conceitos chave, para facilitar seu acompanhamento.

Abra o Xcode e vá em `File`, acesse as opções em `New` e selecione `Project...` para criar um novo projeto. A tela para escolher um template de projeto será aberta. No menu à esquerda, na seção `iOS` selecione a opção `Application`. À direita, selecione `Empty Application` e clique em `Next`.

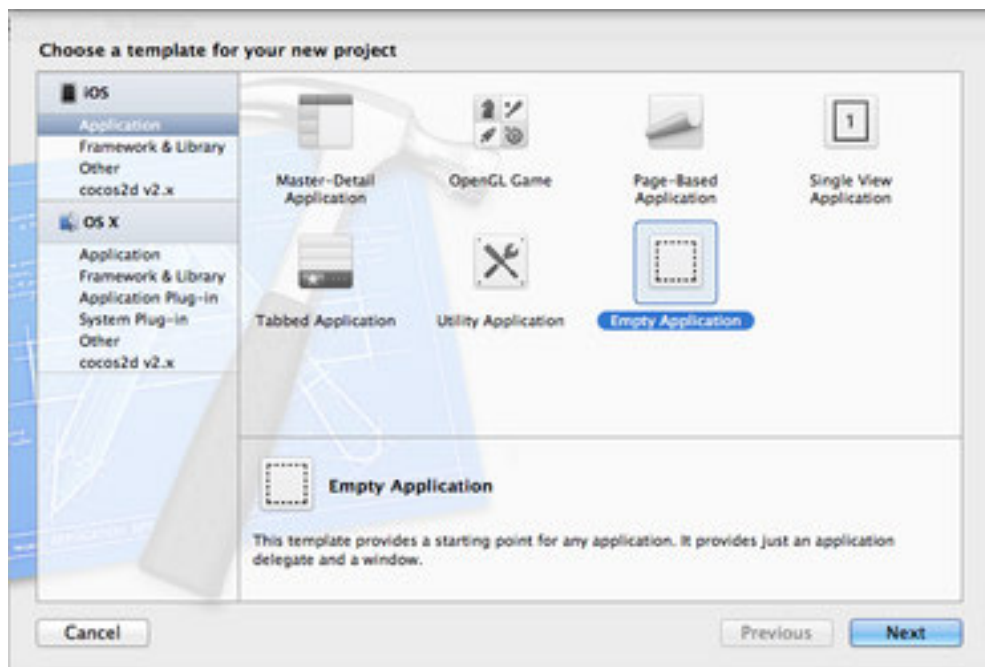


Figura 2.2: Novo projeto de aplicativo iOS.

Criaremos um projeto chamado *Impossible*. Por que esse nome? Se você reparou na lista de funcionalidades, nosso protótipo nunca tem um final feliz!

Coloque o nome do projeto como `impossible`, no campo `Product Name`. Como nosso protótipo será apenas para iPhone, no campo `Device` escolha `iPhone`. Nos demais campos, deixe as opções conforme a figura:

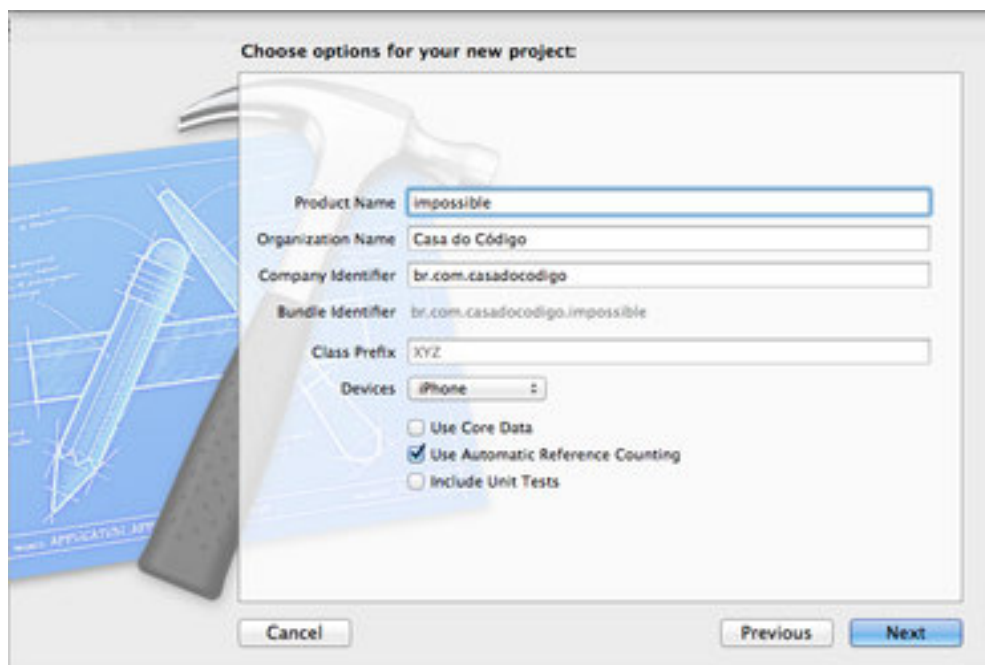


Figura 2.3: Criando o projeto.

Clique em `Next` e escolha o local onde o projeto será gravado.

Nesse momento temos o projeto iniciado. Você já pode executá-lo: selecione *iPhone x.x Simulator* em *scheme* e clique em *Run*. Ao executá-lo, será aberto o simulador de iPhone com uma tela em branco:

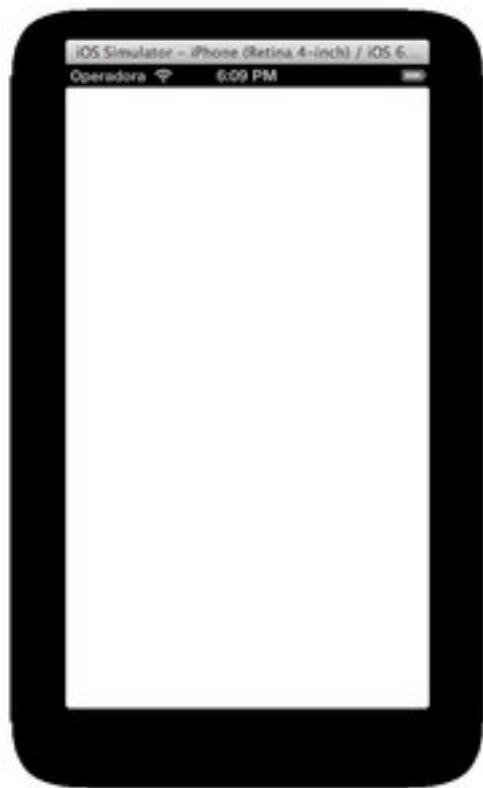


Figura 2.4: Simulador com tela em branco.

Como desenvolveremos nosso protótipo para o iPhone de tela retina de 4 polegadas, no simulador clique em `Hardware`, `Device` e escolha a opção `iPhone (Retina 4-inch)`. O simulador irá mudar para como se fosse um iPhone retina de 4 polegadas. Caso o simulador fique muito grande em sua tela, clique em `Window`, `Scale` e escolha um percentual de zoom que seja melhor para que você visualize o simulador inteiro.

2.2 CRIANDO A BASE DO JOGO

Hora de começar! Para esse protótipo tentaremos manter as coisas simples e diretas, sempre pensando em aplicar os conceitos necessários para criar o jogo nos próximos capítulos.

Teremos apenas duas classes: uma `UIViewController`, na qual trataremos

as ações do jogador, e uma outra classe que terá a lógica do jogo e será responsável por desenhar os objetos na tela. Além dessas duas classes, teremos ainda a `AppDelegate`, criada por padrão em qualquer projeto iOS.

A lógica de um jogo normalmente é dividida em diversas classes. Aqui a orientação a objeto se faz realmente necessária para uma boa utilização dos elementos do game. Nesse primeiro capítulo, manteremos as coisas simples, concentrando a lógica em apenas uma classe. Nos próximos capítulos, quando já tivermos passado pelos conceitos importantes, definiremos a arquitetura do nosso jogo, assim como dividiremos as responsabilidades em diversas classes.

Game ViewController

A classe `GameViewController` começará simples. Essa classe é a porta de entrada do nosso jogo, então ela será a `rootViewController` de nosso aplicativo. Vamos criar esta classe como qualquer `UIViewController` é criada, clicando em `File`, `New` e selecionando `File`. Na tela para escolher o tipo de arquivo a ser criado, vamos escolher `Cocoa Touch` na seção `iOS`, e selecionar `Objective-C Class`:

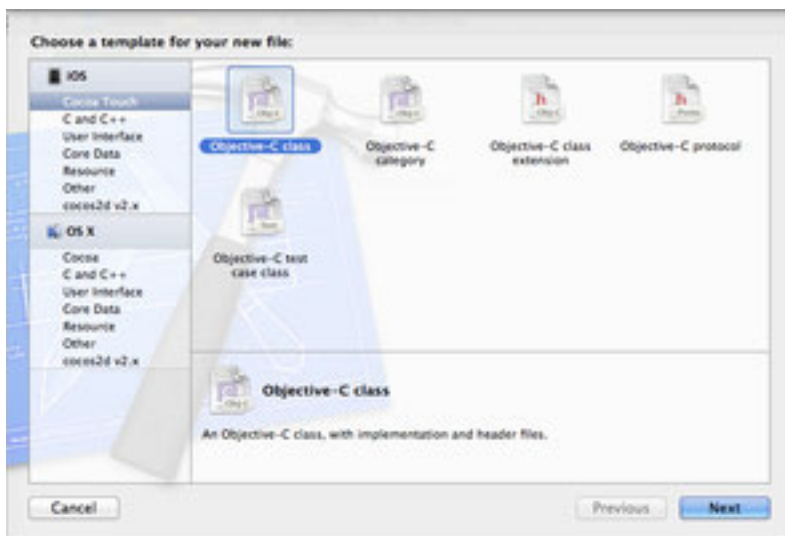


Figura 2.5: Criando nova Classe.

Vamos dar o nome da nossa classe como `GameViewController`, subclasse de uma `UIViewController`:

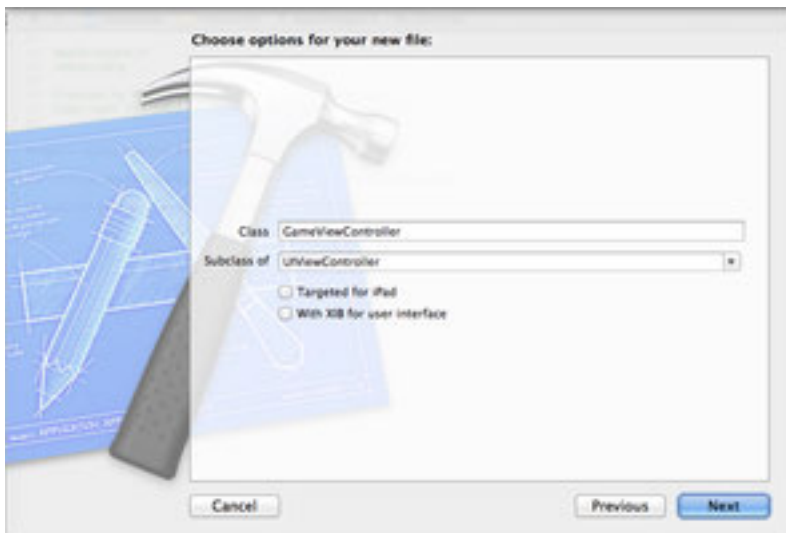


Figura 2.6: Criando a GameViewController.

No nosso arquivo `GameViewController.m` vamos remover o que o Xcode gerou de código padrão, com exceção do método `viewDidLoad`, e deixar a classe assim:

```
#import "GameViewController.h"

@implementation GameViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
}

@end
```

Agora, vamos alterar a classe `AppDelegate` para que use nossa `GameViewController` como `rootViewController`. Para isto, primeiramente devemos importar a `GameViewController` e depois alterar o método `application:didFinishLaunchingWithOptions:`. Altere o arquivo `AppDelegate.m`.

```
#import "GameViewController.h"
```

```
// ...
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:
        [[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.window.backgroundColor = [UIColor whiteColor];

    GameViewController *gameVC = [[GameViewController alloc] init];
    self.window.rootViewController = gameVC;

    [self.window makeKeyAndVisible];
    return YES;
}
```

A lógica do Impossible: criando um loop infinito

A classe que conterá a lógica se chamará `Impossible`. Essa classe deve herdar de `UIView`. Deixe o header `Impossible.h` com o código padrão criado pelo Xcode.

```
#import <UIKit/UIKit.h>
```

```
@interface Impossible : UIView
```

```
@end
```

E limpe todo o código padrão gerado no *implementation* `Impossible.m`.

```
#import "Impossible.h"
```

```
@implementation Impossible
```

```
@end
```

No `iOS` um objeto do tipo `UIView` permite que desenhos sejam executados sobre a superfície em vez de trabalhar com um `Xib` ou `Storyboard`.

Agora que já temos a `GameViewController`, que é a porta de entrada, e a classe `Impossible`, que representa a lógica do jogo, vamos criar o link entre elas. Para isso, na classe `GameViewController.h` criaremos uma propriedade que armazenará a `Impossible` e a chamaremos de `impossibleView`.

```
#import <UIKit/UIKit.h>
#import "Impossible.h"

@interface GameViewController : UIViewController

@property (nonatomic, strong) Impossible *impossibleView;

@end
```

Na `GameViewController.m` inicializaremos nossa `impossibleView` e adicionaremos ela à tela:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    // Instancia um objeto do tipo Impossible
    self.impossibleView = [[Impossible alloc] init];

    // Define o tamanho dele, com base no tamanho da tela
    self.impossibleView.frame = CGRectMake(0.0f,
                                           0.0f,
                                           self.view.frame.size.width,
                                           self.view.frame.size.height);
    [self.view addSubview:self.impossibleView];
}
```

Pode rodar seu projeto novamente, mas ainda não temos bons resultados! Parece que um jogo ainda está longe de se concretizar, mas até o final do capítulo você já terá um protótipo para mostrar.

Game Loop

Normalmente os jogos têm um primeiro conceito importante: um `loop` infinito, conhecido como *game loop* ou *main loop*.

O jogo é uma série de iterações nesse *loop* infinito. Nele, o jogo define posições de elementos, desenha-os na tela, atualiza valores como placar, verifica colisões entre elementos. Isso tudo é realizado diversas vezes por segundo, em que cada tela desenhada é chamada de *frame*. Uma boa analogia são os desenhos feitos em blocos de papel, onde cada desenho (*frame*) é um pedaço da animação. Ao passar tudo rapidamente temos a impressão de movimento.

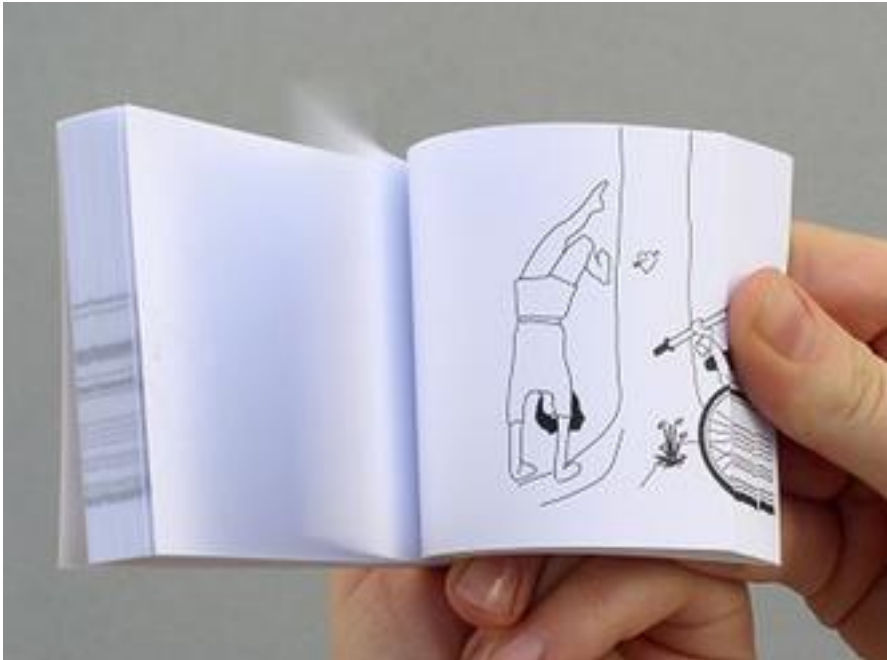


Figura 2.7: Desenho animado em bloco de papel.

Esse conceito é extremamente importante e, com o passar do tempo, difícil de lidar no desenvolvimento de um jogo. Com vários objetos tendo diversas possibilidades na tela a chance de perder o controle é grande. Por isso, tente sempre criar métodos pequenos, com pouca responsabilidade. Dessa forma, encontrar o que pode estar errado fica muito mais fácil.

Vamos colocar o nosso *loop* infinito dentro da classe `Impossible`. Para isto, criaremos um timer que chamará um método 60 vezes por segundo, repetidamente. Também teremos a propriedade `running` que controlará se o jogo está em execução ou não. Primeiramente, inclua o código abaixo no `Impossible.h`:

```
@property (nonatomic, strong) NSTimer *gameRunTimer;  
@property (nonatomic) BOOL running;
```

Então altere também o `Impossible.m`:

```
- (id)init  
{  
    self = [super init];
```



```

if (self) {
    // Timer que executa o Game Loop ("run") 60 vezes por segundo
    self.gameRunTimer = [NSTimer
        scheduledTimerWithTimeInterval:1.0f/60.0f
        target:self
        selector:@selector(run)
        userInfo:nil
        repeats:YES];

    // Variável para controlar a execução do jogo
    self.running = YES;
}
return self;
}

// Game Loop
- (void)run
{
    if (self.running == YES) {
        NSLog(@"Impossible Running...!");
    }
}
}

```

Como criamos nosso *timer* dentro do `init`, no momento em que a `Impossible` for criada ela já começará a repetir o `run` 60 vezes por segundo.

Rode o projeto novamente. Temos nossa primeira saída, ainda não muito empolgante. O console exibe `Impossible Running...!` infinitamente.

2.3 DESENHANDO O OBJETO PRINCIPAL

O motor do nosso jogo já está ligado, funcionando a todo vapor, porém nada acontece na tela. Nosso próximo passo será definir o objeto principal, que chamaremos de `Player`. Nosso `player` será bem simples, apenas um elemento gráfico, no caso um círculo. Pode parecer simples mas jogos 2D são objetos triviais que são animados muitas vezes por segundo. No nosso caso temos um círculo, mas podemos trocar por qualquer recurso ou imagem melhor trabalhado para definir um personagem interessante.

Com a popularização dos smartphones e dos jogos casuais, esse tipo de `player` simples de jogos 2D reapareceu com muita força. O nome atribuído a esses objetos

é `Sprite`, que nada mais são que imagens, normalmente retangulares ou mesmo quadradas, com fundos transparentes.

Você pode encontrar `Sprites` de diversos jogos clássicos na internet. Procure no Google por ‘sprites’ mais o nome de um jogo que você goste. Comece a imaginar como esse jogo funciona com o que falamos até aqui.



Figura 2.8: Sprites do famoso Mario Bros.

Utilizando iOS UIView e CGContext

Para desenhar elementos na tela do jogo no iPhone, temos algumas opções. Poderíamos criar os objetos pelo próprio Xib ou StoryBoard do Xcode. Porém, para que possamos entender como funciona o desenho dos objetos de um game, realizaremos todo o desenho através de códigos.

Quando desenhamos na vida real, precisamos de ferramentas como pincéis e um lugar para utilizá-las, como papel ou telas. O elemento `UIView` no iOS representa essa tela, na qual podemos desenhar diversas formas ou mesmo `Sprites`. Para ter acesso a esse elemento, podemos declarar nossa classe como sendo uma tela, por meio da `UIView`.

Ao utilizar uma `UIView`, temos um tipo de `View` especializado em exibir desenhos na tela. O principal propósito da `UIView` é fornecer o que precisamos para que renderizar as atualizações do jogo a todo momento. Uma `UIView` desenha os objetos em seu método `drawRect:`. Este método é chamado toda vez que o iOS identificar que a nossa `UIView` deve ser redenhada.

Para desenhar vamos usar as funções da `CGContext`, presente no *framework* `CoreGraphics` da Apple. Com ela, conseguiremos definir elementos como textos, linhas, figuras geométricas, cores e tudo que for referente a colocar os elementos do jogo na tela. Chamamos de `context` o contexto atual onde o iOS está desenhando objetos.

Vamos começar sobrescrevendo o método `drawRect:` em nossa classe `Impossible.m`, e declarando uma variável para buscar o contexto atual onde estão sendo desenhados os objetos:

```
- (void)drawRect:(CGRect)rect
{
    CGContextRef context = UIGraphicsGetCurrentContext();

    // desenha o player
    // o que fazer aqui??? já vamos aprender
}
```

No momento em que sobrescrevemos o método `drawRect:` da `UIView`, temos total controle do que iremos desenhar ou não em nossa tela.

Finalmente vamos desenhar o player! Utilizaremos nossos pincéis, no caso, as funções do `CGContext`. Para não deixar o método `drawRect:` muito longo, vamos criar um outro método:

```
- (void)drawPlayerInContext:(CGContextRef)context
{
    UIGraphicsPushContext(context);
    CGContextBeginPath(context);
    CGContextAddArc(context,
                    160,
                    275,
                    25,
                    0,
                    (2 * M_PI),
                    YES); // Circulo de 360° (0 ~ 2pi)
```

```
CGContextSetRGBFillColor(context, 0.0f, 0.9f, 0.0f, 1.0f);
CGContextFillPath(context);
UIGraphicsPopContext();
}
```

Para desenhar nosso player, criamos um círculo completo e preenchemos ele com uma cor.

Agora basta invocar o método `drawPlayer` de dentro do nosso `drawRect`. Repare que só precisamos alterar uma única linha, a que estava com um comentário:

```
- (void)drawRect:(CGRect)rect
{
    CGContextRef context = UIGraphicsGetCurrentContext();

    [self drawPlayerInContext:context];
}
```

Rode novamente o seu projeto. Obtivemos nosso primeiro resultado!

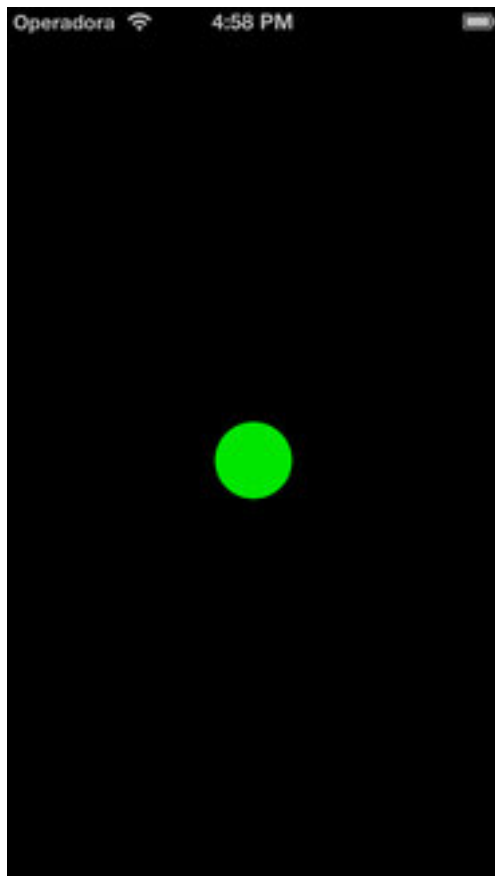


Figura 2.9: Player na tela.

2.4 CAPTANDO OS COMANDOS DO USUÁRIO E MOVENDO OBJETOS

Existem diversas maneiras de interagir com um jogo e com o player principal. Mover o mouse e clicar, utilizar o teclado, tocar a tela ou mesmo capturar o movimento de um aparelho, usando por exemplo, o acelerômetro. No protótipo, utilizaremos o toque na tela para mover o player. Vamos capturar cada toque como sendo um *input* do usuário e, a cada vez que isso ocorrer, iremos dar um comando ao nosso jogo.

Nesse momento vamos explorar esse conceito de *inputs* do usuário no iOS, e novamente reparar na importância da `UIViewController` principal como

porta de entrada do jogo. Utilizaremos um objeto do próprio iOS chamado `UITapGestureRecognizer` para identificar os toques na tela. Toda vez que um toque for detectado, o `UITapGestureRecognizer` chamará um método passando as coordenadas tocadas na superfície da tela. E de posse dessas coordenadas, poderemos tomar ações sobre os objetos na tela do jogo.

Nesse momento, ao detectar um toque, moveremos para baixo nosso player. Repare que aqui poderíamos utilizar a informação que recebemos para tomar ações interessantes no jogo, como mover para um lado ou para o outro, mover mais rápido etc. Para fim de entendimento de conceito e prototipação, seremos simples nessa implementação.

Antes de mais nada, precisamos saber em que posição nosso player está. Declare o atributo `playerY` no `Impossible.h`:

```
@property (nonatomic) int playerY;
```

Defina também um valor inicial para ele quando nossa classe for criada, no método `init` da `Impossible.m`:

```
- (id)init
{
    self = [super init];
    if (self) {
        // Timer que executa o Game Loop (método "run") 60 vezes por segundo
        self.gameRunTimer = [NSTimer
                               scheduledTimerWithTimeInterval:1.0f/60.0f
                               target:self
                               selector:@selector(run)
                               userInfo:nil
                               repeats:YES];

        // Posição inicial do jogador
        self.playerY = 275;

        // Variável para controlar a execução do jogo
        self.running = YES;
    }
    return self;
}
```

E, toda vez que invocarem o `drawPlayerInContext:`, vamos desenhá-lo nessa altura, em vez daquele número fixo. Na `Impossible.m` altere:

```
// Desenha o Player
- (void)drawPlayerInContext:(CGContextRef)context
{
    UIGraphicsPushContext(context);
    CGContextBeginPath(context);
    CGContextAddArc(context,
                    160.0f,
                    self.playerY,
                    25.0f,
                    0,
                    (2 * M_PI),
                    YES); // Círculo de 360° (0 ~ 2pi)
    CGContextSetRGBFillColor(context, 0.0f, 0.9f, 0.0f, 1.0f);
    CGContextFillPath(context);
    UIGraphicsPopContext();
}
```

Ainda na `Impossible.m` teremos um método que pode ser invocado para mover o player para baixo (a tela do iPhone possui a posição `0,0` no canto superior esquerdo):

```
- (void)moveDown:(int)pixels
{
    if (self.running == YES) {
        self.playerY += pixels;
    }
}
```

Como este será um método público que deverá ser chamado pela nossa `GameViewController`, temos também que declará-lo na `Impossible.h`:

```
- (void)moveDown:(int)pixels;
```

Para podermos receber a informação de que a tela foi tocada, em nossa `GameViewController.m` criaremos nosso objeto para reconhecer toques na tela:

```
- (void)viewDidLoad
{
    //...

    // Instancia um objeto para reconhecimento de gestos do tipo "Tap"
    // e a ação a ser executada quando o usuário realizar o gesto
```

```

UITapGestureRecognizer *tapGesture =
    [[UITapGestureRecognizer alloc] initWithTarget:self
                                             action:@selector(handleTapGesture)];

// Adiciona o reconhecimento de gestos à view com a qual o
// usuário irá interagir
[self.impossibleView addGestureRecognizer:tapGesture];
}

```

A classe ainda não vai funcionar pois está faltando o método de *callback* de nosso `UITapGestureRecognizer`, que definimos como `handleTapGesture:`. Vamos implementá-lo:

```

- (void)handleTapGesture:(UITapGestureRecognizer *)gesture
{
    if (gesture.state == UIGestureRecognizerStateEnded) {
        [self.impossibleView moveDown:10];
    }
}

```

Rode o jogo. Mas temos um problema: ao tocar na tela, nosso player ainda não está se movendo. Isto acontece porque não estamos pedindo para nossa `Impossible` redesenhar a tela.

Vamos corrigir este problema! No `run` da `Impossible.m` vamos informar que a tela precisa ser redesenhada:

```

- (void)run
{
    if (self.running == YES) {
        NSLog(@"Impossible Running...!");

        // Informa ao iOS que a tela deve ser redesenhada
        [self setNeedsDisplay];
    }
}

```

Rode o jogo. Algo já deve ocorrer ao tocar na tela: nosso player deve ter sua posição alterada. Mas agora temos um outro problema, veja como o player se movimenta e o que acontece com a tela:

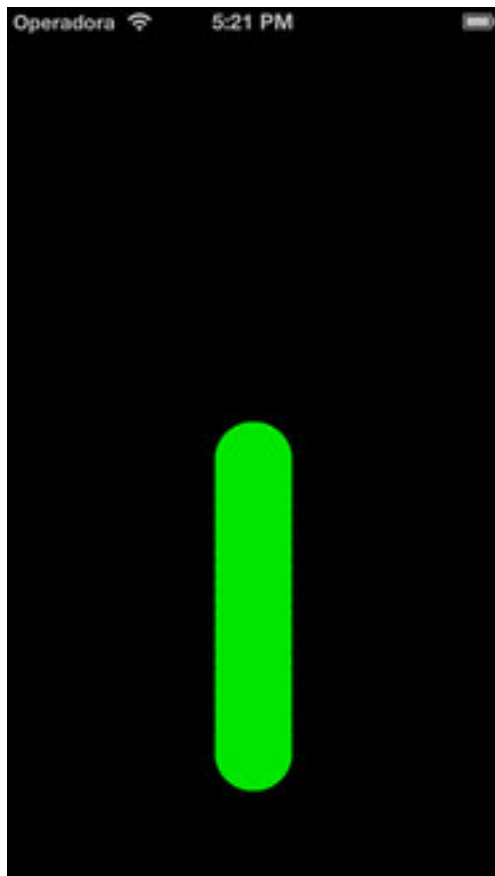


Figura 2.10: Player movendo na tela, mas temos um problema.

Importante: Limpando a tela

Lembra da relação de um jogo com os desenhos em blocos de papel? O que dá a impressão de movimento em um bloco como esse é que cada imagem é desenhada com uma pequena variação de sua posição anterior. Nenhum desenho, se visto individualmente, dá a impressão de movimento ou continuidade do desenho anterior. É como se cada folha do bloco fosse totalmente redesenhada a cada frame.

O que faremos para que o player se mova na tela é zerar a tela toda vez que formos renderizar um novo frame. Como nosso protótipo é simples e não tem um fundo com imagens se movendo, podemos apenas iniciar o frame com um fundo preto. Para jogos com backgrounds mais complexos a estratégia de limpar a tela será

mais complexa.

Na classe `Impossible.m`, altere o método `drawRect:` para pintar o background da tela de preto, chamando o novo método `drawBackgroundInContext:` que iremos criar:

```
- (void)drawRect:(CGRect)rect
{
    CGContextRef context = UIGraphicsGetCurrentContext();

    [self drawBackgroundInContext:context];
    [self drawPlayerInContext:context];
}

// Desenha o Plano de Fundo
- (void)drawBackgroundInContext:(CGContextRef)context
{
    UIGraphicsPushContext(context);
    CGContextSetRGBFillColor(context, 0.0, 0.0, 0.0, 1.0);
    CGContextFillRect(context, CGRectMake(0, 0,
                                           self.frame.size.width, self.frame.size.height));
    UIGraphicsPopContext();
}
```

Rode agora e toque na tela. O player se moverá a cada toque!

2.5 CRIANDO O INIMIGO

Chegamos à parte perigosa! São os inimigos que fazem um jogo ser mais desafiador, que nos instigam a superar algo. O inimigo em um jogo pode estar representado de diversas maneiras. Pode ser o tempo, pode ser uma lógica complexa a ser resolvida ou mesmo outros objetos e personagens.

A partir dos inimigos podemos conhecer diversos conceitos importantes para um jogo funcionar. Assim como o player principal, os inimigos possuem seus próprios movimentos, porém, diferentemente do player, os movimentos do inimigo costumam ser definidos por lógicas internas do jogo. O interessante é que, por mais que os *inputs* do usuário não determinem diretamente o movimento do inimigo, quanto mais inteligente ele for de acordo com a movimentação do player, mais interessante e desafiador pode ser o jogo.

Para o protótipo do jogo, nosso inimigo será um outro círculo, porém, como falamos anteriormente, esse círculo terá sua própria lógica. Ele crescerá com o tempo,

ou seja, de acordo com o passar do jogo, seu raio irá aumentando e consequentemente, ocupando cada vez mais a região do jogo.

Vamos criar, na classe `Impossible.h`, uma propriedade que representa o raio do inimigo:

```
@property (nonatomic) int enemyRadius;
```

Assim como temos um método separado que desenha o player, na classe `Impossible.m` teremos um que desenha o inimigo:

```
- (void)drawRect:(CGRect)rect
{
    CGContextRef context = UIGraphicsGetCurrentContext();

    [self drawBackgroundInContext:context];
    [self drawPlayerInContext:context];
    [self drawEnemyInContext:context];
}

//...

// Desenha o Inimigo
- (void)drawEnemyInContext:(CGContextRef)context
{
    UIGraphicsPushContext(context);
    CGContextBeginPath(context);
    CGContextAddArc(context,
                    100,
                    100,
                    self.enemyRadius,
                    0,
                    (2 * M_PI),
                    YES); // Círculo de 360° (0 ~ 2pi)
    CGContextSetRGBFillColor(context, 0.4f, 0.4f, 0.4f, 1.0f);
    CGContextFillPath(context);
    UIGraphicsPopContext();
}
```

Para este game, queremos que a cada novo *frame* o raio do inimigo cresça em 1. Para isto, vamos alterar o `run`:

```
- (void)run
{
    if (self.running == YES) {
        // Aumenta raio do inimigo
        self.enemyRadius++;

        // Informa ao iOS que a tela deve ser redenhada
        [self setNeedsDisplay];
    }
}
```

Ao rodar o jogo, nosso inimigo cresce sozinho e o player se afasta com o *touch* na tela!

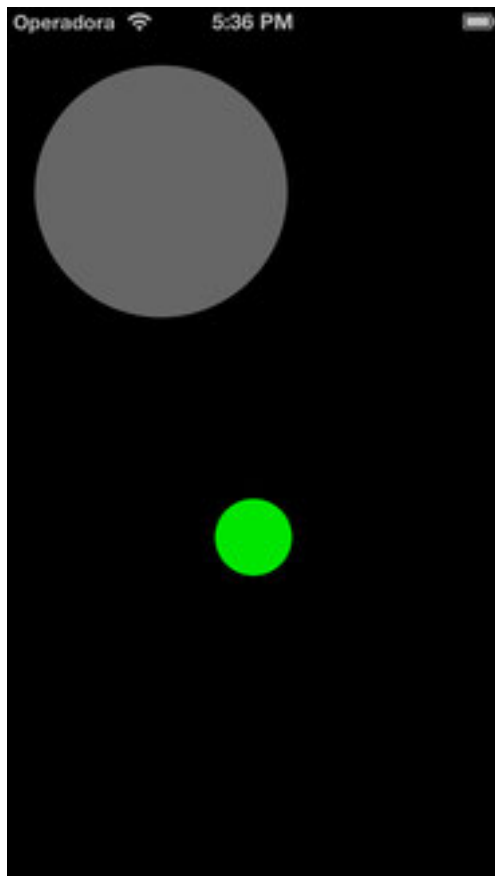


Figura 2.11: Inimigo aparece na tela.

Nesse momento conseguimos mover o player principal e tentar se afastar do inimigo que cresce cada vez mais com o passar do tempo. Agora precisamos detectar a colisão!

2.6 DETECTANDO COLISÕES E MOSTRANDO RESULTADOS

Já passamos pelo conceito de mover objetos, no caso, pelo toque na tela e já também por ter um outro objeto que representa o inimigo e tem sua própria inteligência. A graça do jogo agora é conseguir identificar quando uma determinada situação acontece, situação essa que o player está “lutando contra”.

No nosso caso, o player não pode encostar no círculo que cresce cada vez mais.

Repare que aqui ainda não temos uma história para que essa colisão faça realmente sentido em ser evitada no jogo, porém, é aí que a imaginação faz o jogo se tornar divertido. Jogos antigos, em 2D, não possuíam gráficos incríveis, mas sim, ideias interessantes representadas por objetos simples na tela.

Poderíamos estar desenvolvendo um jogo no qual o player está fugindo de algo. Como um vulcão entrou em erupção e nosso herói (player) deve salvar os habitantes dessa vila, por exemplo. Ou seja, sabendo os conceitos, iremos incrementar o visual para que represente uma história interessante.

Detectando colisões

Precisamos então reconhecer que o círculo maior, que representa o inimigo, conseguiu encostar no círculo menor, movido pelo usuário, que representa o player. Detectar colisões é um assunto muito amplo. Existem diversos tipos de detecção de colisões possíveis.

Uma maneira bem tradicional é considerar que cada elemento é um quadrado ou retângulo e verificar através de geometria se um elemento sobrepõe o outro. Essa forma considera mesmo elementos que não contornam um objeto como parte do mesmo. Na imagem abaixo, uma nave de jogos de tiro. Para detectar que algo colide com ela, a área analisada pode ser generalizada para um quadrado ao redor dela.

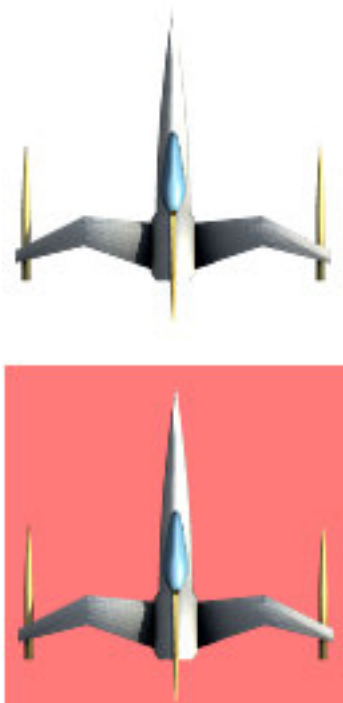


Figura 2.12: Região detectada pelo jogo.

Pode-se questionar se esse método é bom. Será que, se algo encostar na quina do quadrado, que não faz parte da nave, uma colisão será detectada? Em muitos casos essa aproximação é feita por dois motivos.

- Simplificação para detectar a colisão.
- Menor exigência computacional.

Simplificar a detecção por conta de simplificar o algoritmo da colisão é uma prática bem comum. Além disso, é bem mais barato computacionalmente do que ter que analisar cada item real de uma imagem de um player.

Colisões no protótipo

Chegamos a um dos conceitos mais importantes no desenvolvimento de um game! Precisamos identificar a colisão entre o player e o inimigo. Esse é o item

chave do nosso protótipo e normalmente na maioria dos jogos. Existem diversas formas de pontuar, e muitas delas utilizam a colisão entre dois ou mais objetos para isso. Jogos de tiro pontuam pela colisão do tiro com o objeto atirado. Jogos como Super Mario Bros e Street Fighter pontuam pelas colisões do player com moedas ou com inimigos.

Existem diversas formas de detectar colisões, algumas mais complexas outras mais simples. Para o nosso protótipo, utilizaremos a colisão de duas circunferências.

A colisão de dois círculos é uma das mais simples, porém, é relacionada a alguns conceitos matemáticos como o Teorema de Pitágoras.

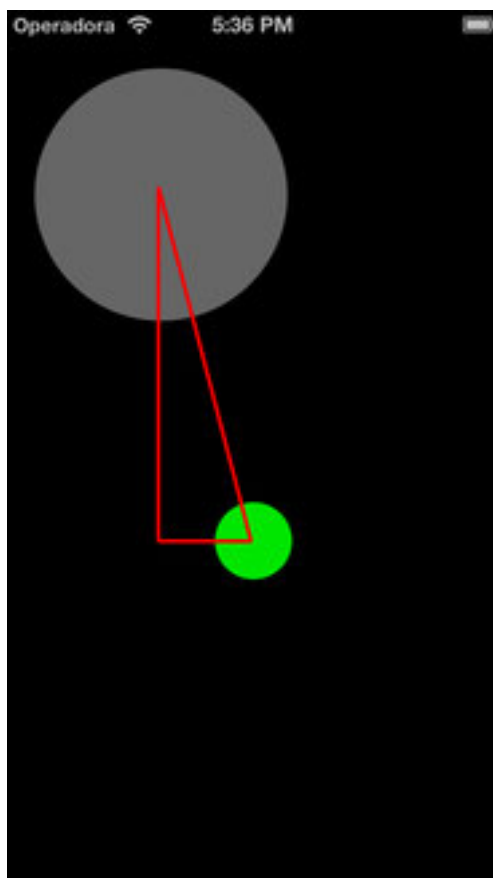


Figura 2.13: Teorema de Pitágoras.

Na figura anterior, existe uma maneira matematicamente simples de determinar

se as circunferências estão sobrepostas. Precisamos identificar os valores a seguir:

- Soma dos raios das duas circunferências
- Valor da hipotenusa, ou distância entre os dois raios

De posse das duas informações acima, conseguimos identificar se a soma dos raios é maior que a hipotenusa gerada. Se for maior, não existe colisão.

Vamos ao código!

Primeiro criaremos algumas propriedades para esse cálculo. As variáveis se referem às posições X e Y de ambas as circunferências, tanto do player quanto a do inimigo.

Altere sua classe `Impossible.h` para ter todos esses atributos. Repare que já possuíamos `enemyRadius` e `playerY`:

```
@property (nonatomic) int playerX;
@property (nonatomic) int playerY;
@property (nonatomic) int playerRadius;

@property (nonatomic) int enemyX;
@property (nonatomic) int enemyY;
@property (nonatomic) int enemyRadius;
```

No `init` da `Impossible.m`, vamos definir os valores padrão destas propriedades:

```
- (id)init
{
    //...
    // Posição inicial do jogador
    self.playerRadius = 25;
    self.playerX = 160;
    self.playerY = 275;

    // Posição inicial do inimigo
    self.enemyX = 0;
    self.enemyY = 0;
    self.enemyRadius = 0;
    //...
}
```

Altere os métodos de desenho do player e inimigo para que utilizem as variáveis que foram criadas:

```
// Desenha o Player
- (void)drawPlayerInContext:(CGContextRef)context
{
    UIGraphicsPushContext(context);
    CGContextBeginPath(context);
    CGContextAddArc(context,
                    self.playerX,
                    self.playerY,
                    self.playerRadius,
                    0,
                    (2 * M_PI),
                    YES); // Círculo de 360° (0 ~ 2pi)
    CGContextSetRGBFillColor(context, 0.0f, 0.9f, 0.0f, 1.0f);
    CGContextFillPath(context);
    UIGraphicsPopContext();
}

// Desenha o Inimigo
- (void)drawEnemyInContext:(CGContextRef)context
{
    UIGraphicsPushContext(context);
    CGContextBeginPath(context);
    CGContextAddArc(context,
                    self.enemyX,
                    self.enemyY,
                    self.enemyRadius,
                    0,
                    (2 * M_PI),
                    YES); // Círculo de 360° (0 ~ 2pi)
    CGContextSetRGBFillColor(context, 0.4f, 0.4f, 0.4f, 1.0f);
    CGContextFillPath(context);
    UIGraphicsPopContext();
}
```

O método que identifica a colisão segue a matemática que já vimos e utiliza as funções `pow` (*power*) para potenciação e a função `sqrt` (*square root*) para raiz quadrada.

```
// Verifica Colisões
```

```
- (void)checkCollision
{
    double distance = 0.0f;

    // Teorema de Pitágoras
    distance = pow(self.playerY - self.enemyY, 2)
        + pow(self.playerX - self.enemyX, 2);
    distance = sqrt(distance);

    if (distance <= (self.playerRadius + self.enemyRadius)) {
        self.running = NO;
    }
}
```

Adicione a chamada ao método que detectará a colisão, dentro do nosso `run`:

```
// Game Loop
- (void)run
{
    if (self.running == YES) {
        // Aumenta raio do inimigo
        self.enemyRadius++;

        // Checa colisões
        [self checkCollision];

        // Informa ao iOS que a tela deve ser redesenhada
        [self setNeedsDisplay];
    }
}
```

Rode o jogo, quando houver colisão entre o inimigo e nosso player, o jogo irá parar! Isto acontece porque quando ocorre a colisão nós alteramos a propriedade `running` para falso, que é checada no início dos métodos `run` e `moveDown`.

Seria mais bonito um Game Over mais convincente, não? Vamos colocar na tela uma mensagem de Game Over. Para os textos de nosso game utilizaremos a classe `UILabel` própria do iOS.

Crie uma propriedade que exibirá o texto de game over na `Impossible.h`.

```
@property (nonatomic, strong) UILabel *gameOverLabel;
```

No init da `Impossible.m` vamos instanciar o *label* `gameOverLabel`, só que sem nenhum texto para que ele não apareça na tela neste momento.

```
- (id)init
{
    //...
    // Criação do Label que exibirá a mensagem de "Game Over!"
    self.gameOverLabel = [[UILabel alloc]
        initWithFrame:CGRectMake(20.0f, 40.0f, 300.0f, 50.0f)];
    self.gameOverLabel.font = [UIFont systemFontOfSize:40.0f];
    self.gameOverLabel.textColor = [UIColor lightGrayColor];
    self.gameOverLabel.backgroundColor = [UIColor clearColor];
    self.gameOverLabel.text = @"";
    [self addSubview:self.gameOverLabel];
    //...
}
```

Ainda na `Impossible.m`, altere o método `checkCollision` para que exiba a mensagem de Game Over quando ocorrer uma colisão.

```
// Verifica Colisões
- (void)checkCollision
{
    double distance = 0.0f;

    // Teorema de Pitágoras
    distance = pow(self.playerY - self.enemyY, 2)
        + pow(self.playerX - self.enemyX, 2);
    distance = sqrt(distance);

    if (distance <= (self.playerRadius + self.enemyRadius)) {
        self.gameOverLabel.text = @"GAME OVER!";
        self.running = NO;
    }
}
```

A partir daqui, o jogo já detecta as colisões e para em Game Over caso o inimigo alcance o player. Essa é uma maneira de se pensar em games que foi utilizada por muito tempo. Atualmente, com o avanço do hardware e com velocidades de processamento cada vez maiores, detecções de colisões muito mais complexas foram criadas.

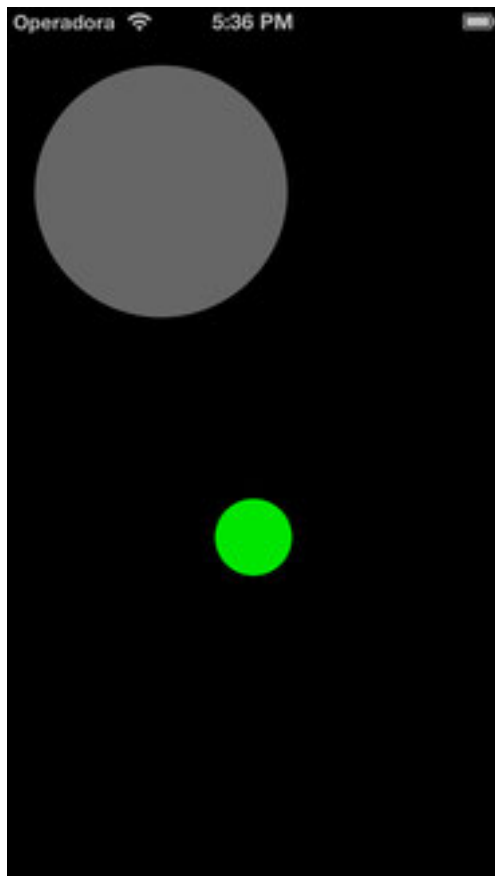


Figura 2.14: Player aparece na tela.

2.7 ADICIONANDO UM PLACAR

Vamos ver agora um próximo conceito importante para a maioria dos jogos, a atualização do placar. A maioria dos jogos atualiza alguma forma de pontuação de tempos em tempos. Essa atualização de placar pode ocorrer de diversas formas, como pela detecção de colisões entre objetos, por exemplo. No protótipo o placar será simples, a cada toque na tela o player se afasta do inimigo e se movimenta, ganhando pontos.

Crie a propriedade `score` na `Impossible.h`, além de um `UILabel` que exibirá o texto do score. Crie também um método que aumenta o score:

```
@property (nonatomic) int score;
```

```
@property (nonatomic, strong) UILabel *gameScoreLabel;

- (void)increaseScore:(int)points;
```

Na `Impossible.m` vamos instanciar o *label* do score e implementar o método que aumenta o score:

```
- (id)init
{
    //...
    // Criação do Label que exibirá o score do jogador
    self.gameScoreLabel = [[UILabel alloc]
                           initWithFrame:CGRectMake(20.0f, 85.0f, 300.0f, 30.0f)];
    self.gameScoreLabel.font = [UIFont systemFontOfSize:25.0f];
    self.gameScoreLabel.textColor = [UIColor whiteColor];
    self.gameScoreLabel.backgroundColor = [UIColor clearColor];
    self.gameScoreLabel.text = @"0";
    [self addSubview:self.gameScoreLabel];
    //...
}

// Soma Pontos
- (void)increaseScore:(int)points
{
    if (self.running == YES) {
        self.score += points;
        self.gameScoreLabel.text = [NSString stringWithFormat:@"%d",
                                     self.score];
    }
}
```

No método que recebe o evento de toque, incrementaremos os pontos. Atualize o método `handleTapGesture:` do arquivo `GameViewController.m`:

```
- (void)handleTapGesture:(UITapGestureRecognizer *)gesture
{
    if (gesture.state == UIGestureRecognizerStateEnded) {
        [self.impossibleView moveDown:10];
        [self.impossibleView increaseScore:100];
    }
}
```

O jogo deve estar assim:

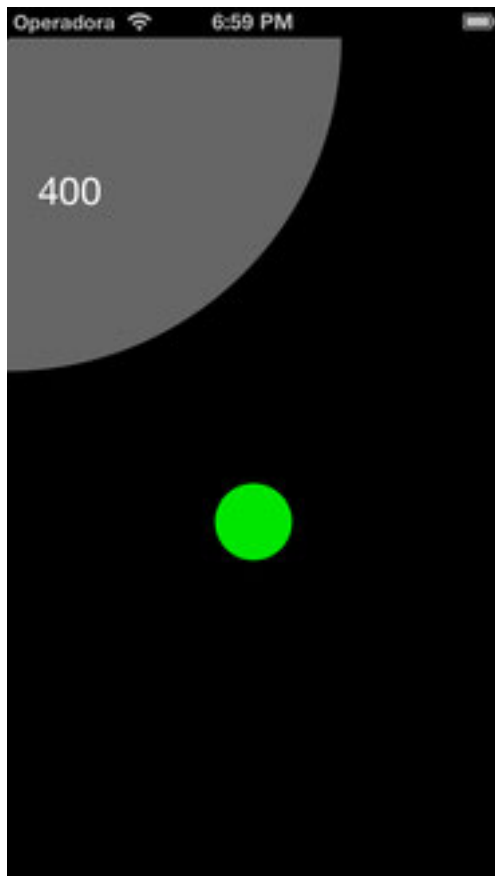


Figura 2.15: Score aparece na tela.

2.8 CRIANDO BOTÕES DE INTERFACE DO USUÁRIO

Outro conceito importante em jogos é a possibilidade de interagir com elementos de configuração do game. Poder pausar o jogo, entrar em uma tela para mudar, por exemplo, a dificuldade ou mesmo fechar e sair do jogo são partes importante do desenvolvimento.

No protótipo, ilustraremos esse conceito com duas opções na tela, para *Restart* (reiniciar) e *Stop* (parar). Simplificaremos para entender o conceito, e depois, poderemos converter para botões mais interessantes.

No arquivo `GameViewController.m` crie os botões e métodos para o *Restart* e *Stop*:

```
- (void)viewDidLoad
{
    //...
    // Botão para Reiniciar o jogo
    UIButton *buttonRestart =
        [UIButton buttonWithType:UIButtonTypeCustom];
    [buttonRestart setTitle:@"Restart"
                   forState:UIControlStateNormal];
    buttonRestart.contentHorizontalAlignment =
        UIControlContentHorizontalAlignmentLeft;
    buttonRestart.frame = CGRectMake(20.0f, 170.0f, 80.0f, 35.0f);
    [buttonRestart addTarget:self
                      action:@selector(restart:)
                      forControlEvents:UIControlEventTouchUpInside];
    [self.view addSubview:buttonRestart];

    // Botão para Parar o jogo
    UIButton *buttonStop =
        [UIButton buttonWithType:UIButtonTypeCustom];
    [buttonStop setTitle:@"Stop" forState:UIControlStateNormal];
    buttonStop.contentHorizontalAlignment =
        UIControlContentHorizontalAlignmentLeft;
    buttonStop.frame = CGRectMake(20.0f, 250.0f, 80.0f, 35.0f);
    [buttonStop addTarget:self
                   action:@selector(stop:)
                   forControlEvents:UIControlEventTouchUpInside];
    [self.view addSubview:buttonStop];
}

- (void)restart:(id)sender
{
    // Reinicia o jogo
    // Já vamos aprender o que fazer aqui!
}

- (void)stop:(id)sender
{
    // Para o jogo
```



```
// Já vamos aprender o que fazer aqui!  
}
```

Rode o jogo e veja que os botões de *Restart* e *Stop* já aparecem na tela.

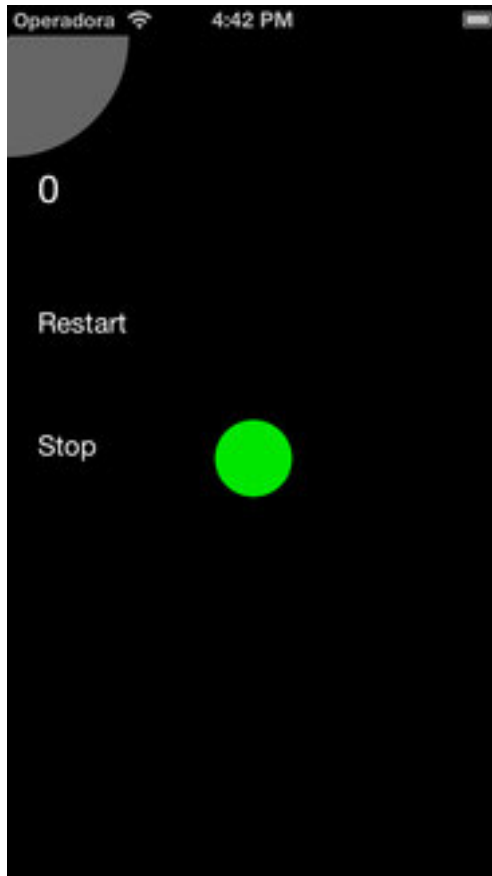


Figura 2.16: Botões de Restart e Stop na tela.

Agora, precisaremos de uma forma de parar e reinicializar as propriedades do game. No arquivo `Impossible.h` defina os métodos `restart` e `stopGame`:

```
- (void)restart;  
- (void)stopGame;
```

No arquivo `Impossible.m` implemente estes métodos. Repare que, para o *restart*, utilizaremos uma abordagem simples de reinicialização.

```
// Para o Jogo
- (void)stopGame
{
    self.running = NO;
}

// Reinicia o jogo, redefinindo as variáveis
- (void)restart
{
    self.enemyX = 0;
    self.enemyY = 0;
    self.enemyRadius = 0;

    self.playerRadius = 25;
    self.playerX = 160;
    self.playerY = 275;

    self.score = 0;
    self.gameScoreLabel.text = @"0";
    self.gameOverLabel.text = @"";

    self.running = YES;
}
```

Precisamos agora alterar para que o toque dos botões chame estes métodos da classe `Impossible`. Na `GameViewController.m`, altere os métodos `restart:` e `stop:`:

```
- (void)restart:(id)sender
{
    [self.impossibleView restart];
}

- (void)stop:(id)sender
{
    [self.impossibleView stopGame];
}
```

A tela final do protótipo deve estar assim:

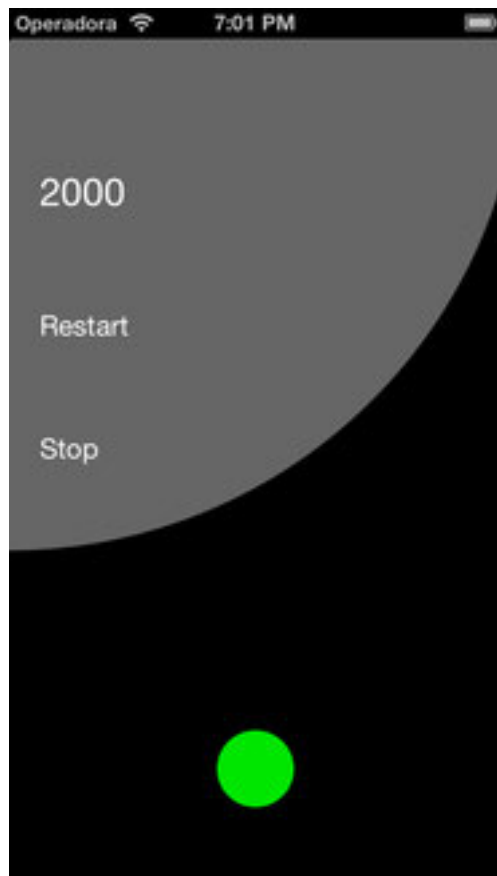


Figura 2.17: Protótipo final.

2.9 ADICIONANDO MAIS VIDA: IMAGENS DA NAVE E DO CÉU

Temos toda a lógica do protótipo rodando e já podemos, finalmente, alterar alguns elementos visuais para finalizar o protótipo do jogo e fechar os conceitos básicos.

Vamos desenhar um *background* que simule um céu escuro com estrelas. Para isso, utilizaremos a imagem `sky.png`. Um ponto importante é reparar que quanto mais sprites forem adicionados ao jogo, mais esforço computacional, o que pode tornar o jogo mais lento. Mais a frente, utilizaremos *frameworks* que otimizam essa questão.

Adicione os arquivos `sky.png` e `nave.png` no diretório do projeto. Você pode

fazer isto pelo Xcode clicando na pasta `Impossible` do projeto, e acessando o menu `File, Add Files to "Impossible"....`

Vamos alterar algumas linhas para utilizar as imagens. No método `drawBackgroundInContext:` da classe `Impossible` adicione as linhas que desenharam uma imagem como *background* do game.

```
// Desenha o Plano de Fundo
- (void)drawBackgroundInContext:(CGContextRef)context
{
    UIGraphicsPushContext(context);
    CGContextSetRGBFillColor(context, 0.0, 0.0, 0.0, 1.0);
    CGContextFillRect(context, CGRectMake(0, 0,
        self.frame.size.width,
        self.frame.size.height));
    UIGraphicsPopContext();

    // Utiliza uma imagem do projeto e a desenha em um
    // determinado ponto da tela
    UIImage *image = [UIImage imageNamed:@"sky.png"];
    [image drawAtPoint:CGPointMake(0.0f, 0.0f)];
}
```

Altere o método `drawPlayerInContext:` para renderizar a imagem da nave.

```
// Desenha o Player
- (void)drawPlayerInContext:(CGContextRef)context
{
    UIGraphicsPushContext(context);
    CGContextBeginPath(context);
    CGContextAddArc(context,
        self.playerX,
        self.playerY,
        self.playerRadius,
        0, (2 * M_PI),
        YES); // Círculo de 360° (0 ~ 2pi)
    CGContextSetRGBFillColor(context, 0.0f, 0.9f, 0.0f, 1.0f);
    CGContextFillPath(context);
    UIGraphicsPopContext();

    // Utiliza uma imagem do projeto e a desenha em um
    // determinado ponto da tela
}
```

```
UIImage *image = [UIImage imageNamed:@"nave.png"];
[image drawAtPoint:CGPointMake(
    (self.playerX - (image.size.width / 2)),
    (self.playerY - (image.size.height / 2)))];
}
```

Altere a cor do inimigo para vermelho.

```
// Desenha o Inimigo
- (void)drawEnemyInContext:(CGContextRef)context
{
    UIGraphicsPushContext(context);
    CGContextBeginPath(context);
    CGContextAddArc(context,
        self.enemyX,
        self.enemyY,
        self.enemyRadius,
        0,
        (2 * M_PI),
        YES); // Círculo de 360° (0 ~ 2pi)
    //CGContextSetRGBFillColor(context, 0.4f, 0.4f, 0.4f, 1.0f);
    CGContextSetRGBFillColor(context, 1.0f, 0.0f, 0.0f, 1.0f);
    CGContextFillPath(context);
    UIGraphicsPopContext();
}
```

Pode rodar o jogo, o protótipo está com sprites e conceitos fundamentais de um jogo!

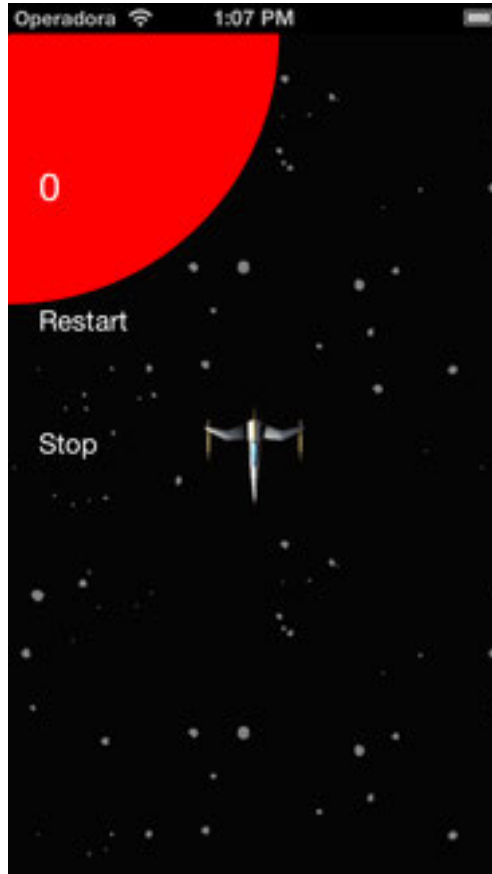


Figura 2.18: Imagem do jogo.

2.10 CONCLUSÃO

Um jogo possui diversos conceitos bem específicos, que não são comuns em outros tipos de projetos de *software* como *web* ou *desktop*. Para desenvolver um jogo é interessante ter bem esclarecidas as partes principais que compõem o quebra-cabeça do mundo da programação para games.

Um jogo ocorre normalmente em um `loop infinito`, no qual *inputs*, ou seja, entradas de comandos, são interpretados e utilizados para executar as lógicas do game. O movimento do player, normalmente uma imagem chamada *sprite* costuma ser dado a partir desses *inputs*, assim como o movimento dos inimigos, que

indiretamente, também é calculado.

Conceitos periféricos como atualização da tela, limpeza da tela e botões de comandos também são importantes e devem ser todos muito bem pensados na execução do jogo.

Com isso em mente, podemos planejar nosso jogo e iniciar seu desenvolvimento.

CAPÍTULO 3

História do jogo

Jogos são feitos de magia, de ilusão, de fantasia. São histórias que envolvem as pessoas de uma forma poderosa, na qual o usuário se sente o protagonista estando no comando das ações.

No começo do livro falamos sobre um jogo fantástico chamado *River Raid*, além de desenvolver um protótipo de jogo de avião no capítulo anterior. Pois bem, chegou a hora!

Mas se criarmos um jogo de nave, qual apelo ele terá? O que o diferenciara dos mil outros jogos que podemos encontrar na App Store? O que fará prender a atenção do jogador?

O enredo, os personagens e a carisma são peças fundamentais que vestem um jogo. É realmente importante considerar um tema chamativo que seja diferente do que já estamos acostumados. Como fazer algo um pouco diferente em um jogo de naves? Como trazer isso para um contexto com o qual os nossos jogadores estejam familiarizados?

Criaremos um jogo também com a temática de aviões, como uma homenagem

a um importante brasileiro que participou do início dessa revolução aérea.

Em 1873 nascia Alberto Santos Dumont, um piloto, atleta e inventor brasileiro.

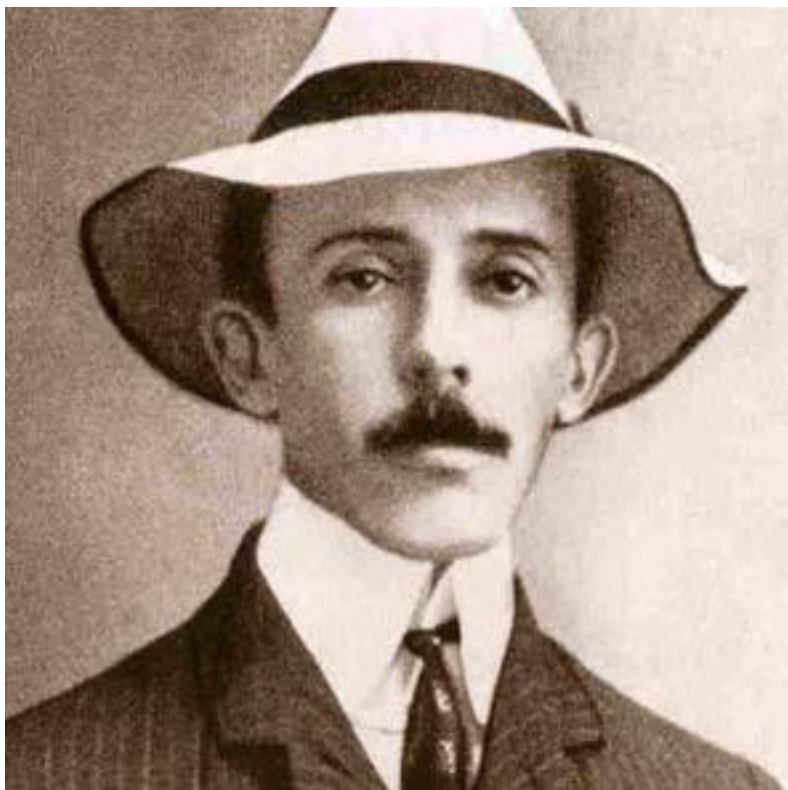


Figura 3.1: Alberto Santos Dumont

Santos Dumont projetou, construiu e voou os primeiros balões dirigíveis com motor a gasolina, conquistando o Prêmio Deutsch em 1901, quando contornou a Torre Eiffel. Se tornou então uma das pessoas mais famosas do mundo durante o século XX.

3.1 14-BIS

Em 1906, Santos Dumont criou um avião híbrido chamado 14 bis, considerado o primeiro objeto mais pesado que o ar a superar a gravidade terrestre.

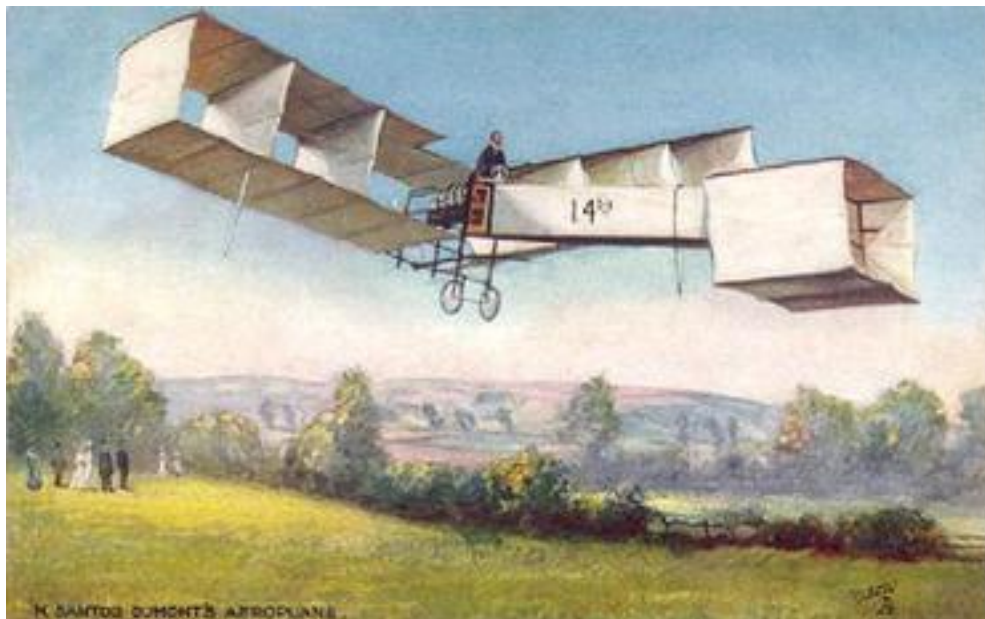


Figura 3.2: 14-bis do brasileiro Santos Dumont

O pesadelo de Santos Dumont

Em agosto de 1914 começava a Primeira Guerra Mundial e os aviões começaram a ser utilizados em combates aéreos. A guerra ficou cada vez mais violenta, com metralhadoras e bombas. Santos Dumont viu seu sonho se transformar em pesadelo.

Em 1932, um conflito entre o estado de São Paulo e o governo de Getúlio Vargas foi iniciado e aviões atacaram a cidade. Essa visão causou muita angustia a Santos Dumont, que cometeu suicídio.

3.2 14-BIS VS 100 METEOROS

Santos Dumont inventou o 14-bis não com o intuito de guerra. Nesse jogo, homenagearemos Dumont e sua invenção utilizando sua aeronave para salvar o planeta!

Depois do meteoro Shoemaker-Levy-9 que caiu em Júpiter, depois do meteoro que caiu na Rússia em 2013, tudo indicava que o fim estava próximo. O exército brasileiro detectou a presença de 100 meteoros entrando na órbita terrestre! Esses meteoros acabarão com a existência de toda forma de vida que conhecemos caso não sejam detidos urgentemente.



Figura 3.3: 14-bis VS 100 Meteoros

O planeta está em apuros! Todas as partes do mundo só falam nesse assunto e buscam formas de evitar o fim. Eis que surge um brasileiro, com seu invento 14-bis, para enfrentar o perigo e tentar salvar a todos nós. Munido de uma arma poderosa para destruir os meteoros que caem sobre a Terra, você comandará a aeronave 14-bis nessa aventura!

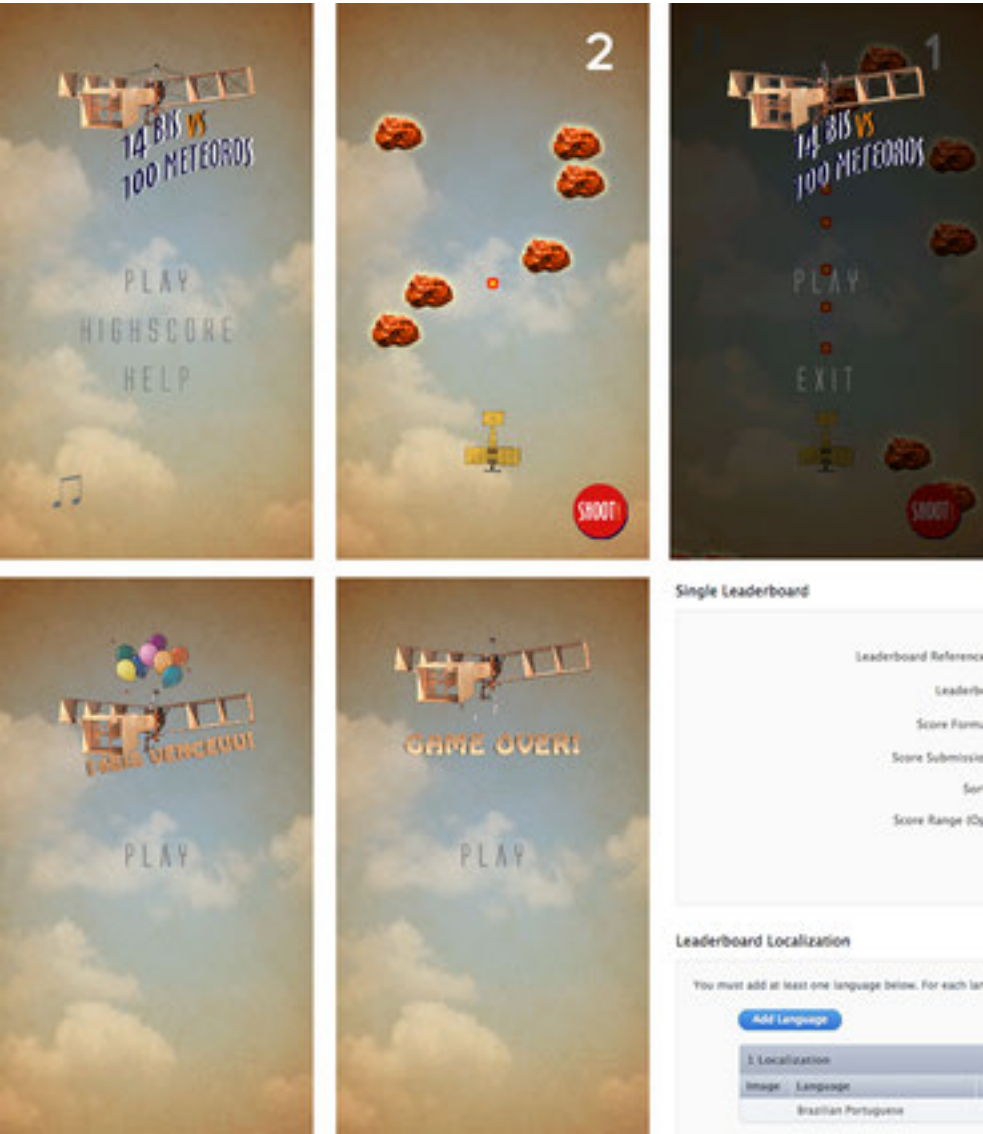


Figura 3.4: Fluxo do game 14-bis VS 100 Meteoros

CAPÍTULO 4

Tela inicial: Lidando com Background, logo e botões de menu

Hora de começar o jogo de verdade! Agora que já passamos pelos conceitos básicos de desenvolvimento de jogos como *game loop*, *sprites*, colisões e *inputs*, podemos ir um pouco mais a fundo no desenvolvimento.

Vamos criar um jogo baseado no game do capítulo anterior, porém dessa vez utilizando um *framework* de desenvolvimento de jogos chamado Cocos2D. O motivo de utilizar um *framework* daqui pra frente é otimizar diversos aspectos, dentre eles:

- Não se preocupar com a posição exata em pontos/pixels dos objetos, como botões, por exemplo
- Utilizar comportamentos já implementados para *sprites*, para não ter problemas com posicionamento das imagens
- Eliminar da lógica a questão da limpeza de tela, deixando isso como responsabilidade do *framework*

- Conseguir efeitos interessantes já implementados pelo Cocos2D
- Trabalhar mais facilmente com sons e efeitos

Nesse capítulo criaremos a tela inicial, que será composta por um logo, um background, e quatro botões. Veremos aqui como posicionar cada um desses elementos na tela e como detectar os *inputs* dos botões utilizando o Cocos2D. Ao fim do capítulo, devemos ter a tela inicial como a seguir:

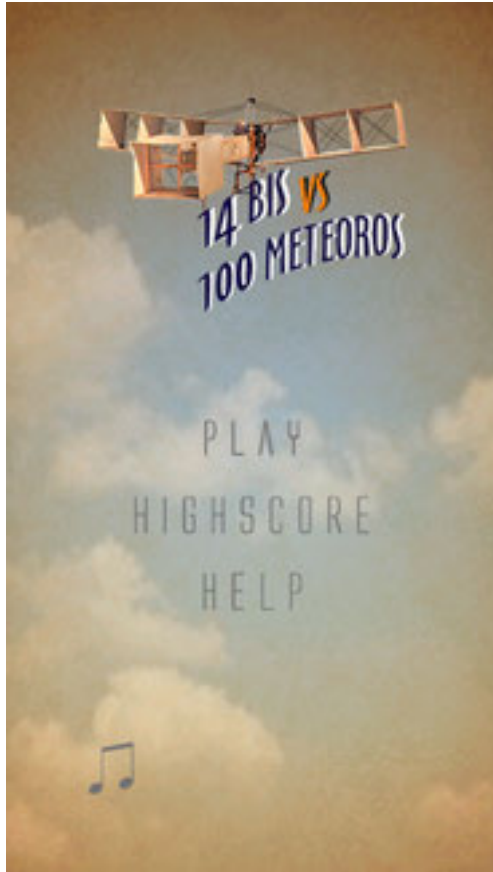


Figura 4.1: Tela de abertura.

Você poderá encontrar o código completo do jogo, junto com algumas melhorias, no GitHub:

https://github.com/BivisSoft/jogos_ios_14bis

Mas prefira você mesmo escrevê-lo! Será menos trabalhoso do que você imagina, e certamente ajudará muito no seu aprendizado.

4.1 SOBRE O COCOS2D

O Cocos2D é *framework open source* de desenvolvimento de jogos. A versão original foi criada em Python e desde então foi portada para diversas linguagens como C++, JavaScript, Objective-C e Java. É muito poderoso e simples.

Para utilizar a versão para iOS, basta baixá-lo no seguinte endereço:

<http://www.cocos2d-iphone.org/download/>

Após baixado, você deve instalar os templates do Cocos2D para que possa iniciar um projeto já utilizando o *framework*. Extraia o arquivo baixado, abra o Terminal e navegue até o diretório onde o Cocos2D foi extraído. Depois, execute o comando `./install-templates.sh -f`.

```
$ cd cocos2d-iphone
$ ./install-templates.sh -f
```

4.2 INICIANDO O PROJETO

Vamos iniciar criando um novo projeto no Xcode, porém, desta vez utilizaremos um template do Cocos2D como base. Abra o Xcode e vá em `File`, acesse as opções em `New` e selecione `Project...` para criar um novo projeto.

A tela para escolher um template de projeto será aberta. No menu à esquerda, na seção `iOS` selecione a opção `cocos2d v2.x`. À direita, selecione `cocos2d iOS` e clique em `Next`.

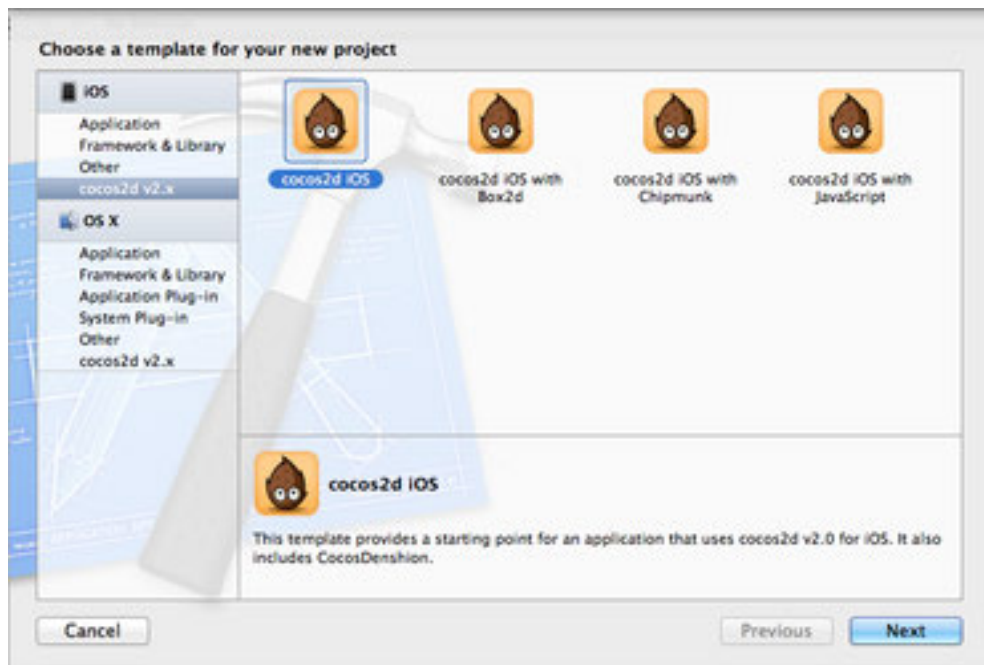


Figura 4.2: Novo projeto de Cocos2D.

Criaremos um projeto chamado *Bis*. O nome do bundle ficará à seu critério, mas você pode seguir a sugestão de usar `br.com.casadocodigo.bis`. Dessa forma você poderá sempre acompanhar com facilidade o código fonte completo que está no GitHub em https://github.com/BivisSoft/jogos_ios_14bis. Como nosso jogo será apenas para iPhone, no campo `Device` escolha `iPhone`.

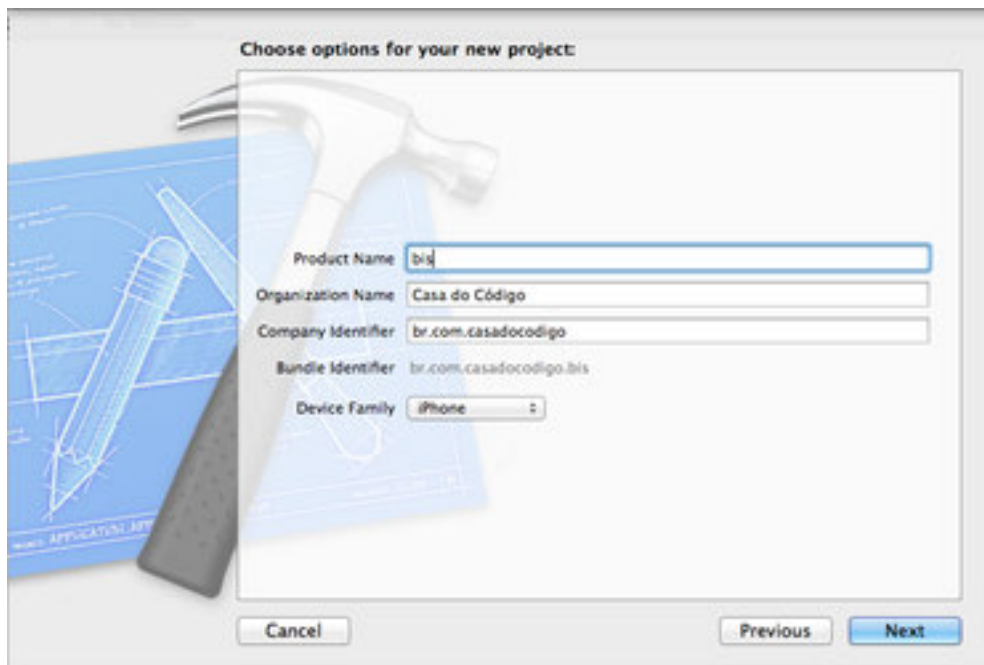


Figura 4.3: Criando o projeto Cocos2D.

Clique em `Next` e escolha o local onde o projeto será gravado.

Nesse momento temos o projeto iniciado. Execute-o e o simulador deverá exibir uma tela escrito *Hello World*:



Figura 4.4: Simulador com Hello World do Cocos2D.

Você percebeu os números que aparecem no canto inferior esquerdo do simulador? Estes números indicam o consumo de memória, tempo entre um *game loop* e outro, e a quantidade de *frames* por segundo. Estas informações são bastante úteis caso você perceba que seu jogo está muito lento e queira verificar a que velocidade ele está sendo executado.

Para nosso game, vamos retirar estes números da tela. Para isto, no arquivo `AppDelegate.`, no método `application:didFinishLaunchingWithOptions:` vamos alterar a opção de exibir FPS para *não*:

```
//...
// Display FSP and SPF
[director_ setDisplayStats:NO];
//...
```

4.3 AJUSTANDO A ORIENTAÇÃO

Por padrão, o Cocos2D inicia um novo projeto com a orientação do aparelho em modo paisagem. Para nosso game, alteraremos o projeto para rodar o jogo em modo retrato. Abra as configurações do projeto em `Supported Interfaces Orientations` e marque apenas a opção `Portrait`:



Figura 4.5: Configurando projeto em modo retrato.

No `AppDelegate.m`, dentro do `@implementation` da `MyNavigationController`, altere os seguintes métodos padrões para que a tela não gire quando o aparelho mudar de posição:

```
- (NSUInteger)supportedInterfaceOrientations
{
    return UIInterfaceOrientationMaskPortrait;
}
```

```
- (BOOL)shouldAutorotateToInterfaceOrientation:  
    (UIInterfaceOrientation)interfaceOrientation  
{  
    return interfaceOrientation == UIInterfaceOrientationPortrait;  
}
```

Rode o game e agora o Hello World deverá aparecer em modo retrato:



Figura 4.6: Simulador com Hello World em modo retrato.

4.4 BACKGROUND

A primeira tela do game é a tela de abertura, e no Cocos2D, utilizamos uma classe chamada `CCLayer` para identificar cada tela do jogo. Ao herdar dessa classe do *framework*, ganhamos alguns reconhecimentos do Cocos2D, como conseguir fazer a transição entre as telas com apenas uma linha de código.

Uma classe que herda de `CCLayer` não precisa ter muitos códigos do *framework*, podemos criar nossa tela inicial como bem entendermos, apenas utilizando esse comportamento para informar ao *framework* que tipo de objeto estamos representando.

Layers

Criar telas com o `CCLayer` do Cocos2D é criar telas pensando em camadas que se sobrepõem. Essas camadas são transparentes, a menos quando definidas de outra forma, e quando colocadas uma sobre as outras definem a tela final.

Na tela de abertura, podemos pensar em camadas para a imagem de *background*, para o logo e para o menu.

Criaremos uma classe chamada `TitleScreen`. Você pode criá-la no diretório ou grupo do projeto que desejar. No nosso projeto, utilizamos o diretório `Scenes`. Lembre-se de organizar suas classes em grupos significativos.

Nesta classe, utilizaremos um segundo componente do Cocos2D. Para instanciar uma tela no *framework*, utilizamos a classe `CCScene`, que é devolvida já pronta para utilizarmos quando invocamos o método `node`.

Scenes

Outro objeto importante do Cocos2D são as `Scenes`. Com elas, conseguimos inicializar telas do jogo. Um jogo pode ter quantas `Scenes` forem necessárias, porém apenas uma estará ativa por vez.

Por exemplo, no nosso jogo teremos a tela de abertura, a tela do jogo, a tela de ajuda, a tela de pause etc. Cada uma delas é um `Scene`.

Vamos ao código inicial da tela de abertura. Precisamos de uma classe que saiba trabalhar com camadas e de uma tela. Criaremos a classe `TitleScreen` que receberá essas definições de camadas e a adicionaremos em uma `Scene`, formando a base da tela inicial.

Criaremos o método `scene`, responsável por instanciar nossa classe e retorná-la dentro de uma `Scene`. Você perceberá ao longo do livro que todas as `scenes` que

criarmos terão este método.

No header `TitleScreen.h` iremos declarar o método `scene`:

```
#import "cocos2d.h"

@interface TitleScreen : CCLayer

+ (CCScene *)scene;

@end
```

E no *implementation* `TitleScreen.m` vamos implementar o método. Você pode copiá-lo da classe `HelloWorldLayer`, criada pelo Cocos2D.

```
+ (CCScene *)scene
{
    // 'scene' is an autorelease object.
    CCScene *scene = [CCScene node];

    // 'layer' is an autorelease object.
    TitleScreen *layer = [TitleScreen node];

    // add layer as a child to scene
    [scene addChild:layer];

    // return the scene
    return scene;
}
```

O código anterior prepara a tela para utilização e posicionamento dos elementos, no nosso caso, esses elementos serão *background*, logo e botões.

Vamos iniciar configurando o *background* do game. Assim como botões ou logo, o *background* também é um objeto representado por uma imagem. Lembre-se que para manipular imagens temos o conceito de *Sprites*, que basicamente é um objeto associado à uma figura.

Sprites

Um `Sprite` no Cocos2D é como qualquer outro `Sprite`, ou seja, uma imagem 2D que pode ser movida, rotacionada, animada, ter sua escala alterada etc. Uma das

vantagens de utilizar `Sprites` como objetos do Cocos2D é que ganhamos algumas possibilidades de animação, que veremos mais à frente.

Criaremos então um `sprite` que representa nosso *background* e iremos adicioná-lo à tela de abertura. Para isto, instanciamos um objeto do tipo `CCSprite` informando qual a imagem desejada e configuramos sua posição. Aqui, utilizaremos o tamanho da tela, tanto largura quanto altura, para posicionar o *background* de forma centralizada. Faremos isso com um elemento muito importante do Cocos2D, o `CCDirector`, que será aprofundado mais à frente. Vamos adicionar o *background* na tela de abertura, instanciando-o no `init` da `TitleScreen.m`:

```
- (id)init
{
    self = [super init];
    if (self) {
        // Imagem de Background
        CCSprite *background =
            [CCSprite spriteWithFile:@"background.png"];
        background.position =
            ccp([CCDirector sharedDirector].winSize.width / 2.0f,
               [CCDirector sharedDirector].winSize.height / 2.0f);
        [self addChild:background];
    }
    return self;
}
```

Da mesma forma que acontece nos demais aplicativos de iOS, podemos ter uma imagem para aparelhos de tela não-retina e outra com o dobro de resolução para aparelhos com tela retina. Entretanto, ao invés de utilizar arquivos com final “@2x”, o Cocos2D utiliza arquivos com final “-hd” para identificar uma imagem para tela retina.

O Cocos2D já vem habilitado para utilizar imagens retina e reconhecê-las como arquivos com final “-hd”. Estas configurações vêm definidas como padrão no `AppDelegate.m`. Basta apenas incluir as imagens em ambas resoluções no projeto e o Cocos2D vai utilizar uma ou outra automaticamente.

Precisamos então do `background.png` e `background-hd.png` do nosso jogo. Criamos algumas imagens e utilizamos outras de fontes gratuitas para nosso jogo. Você pode baixar um zip que as contém nesse endereço:

https://github.com/bivissoft/jogos_ios_14bis

Coloque os arquivos `background.png` e `background-hd.png` dentro do diretório `Resources` do seu projeto. Você precisará repetir esse procedimento para outras imagens, sons e arquivos dos quais precisaremos no decorrer do desenvolvimento de nosso jogo.

4.5 ASSETS DA TELA DE ABERTURA

A tela de abertura do game terá 6 *assets* (arquivos como imagens e figuras) que serão utilizados para compor logo e menus.

Para começar, vamos organizar na classe `Assets` os arquivos de imagens que utilizaremos no game. Crie a classe `Assets`, subclasse de um `NSObject`, e em seu header declare as imagens que utilizaremos:

```
@interface Assets : NSObject
#define kBACKGROUND    @"background.png"
#define kLOGO           @"logo.png"
#define kPLAY           @"play.png"
#define kHIGHSCORE      @"highscore.png"
#define kHELP           @"help.png"
#define kSOUND          @"sound.png"
@end
```

Para facilitar, importaremos a classe `Assets` em nosso arquivo `Prefix.pch`. Desta forma, todas as classes de nosso projeto automaticamente importarão a `Assets`. Também incluiremos na `Prefix.pch` o *import* da `cocos2d.h`, outra classe que será bastante utilizada em nosso projeto.

```
//...
#ifdef __OBJC__
#import <UIKit/UIKit.h>
#import <Foundation/Foundation.h>
#import "cocos2d.h"
#import "Assets.h"
#endif
```

Sempre após alterar o arquivo `Prefix.pch`, precisamos recompilar nosso projeto para que as classes identifiquem as alterações. Você pode fazer isto acessando o menu `Product` segurando a tecla *option* e selecionando `Clean Build Folder`. Após isto, acesse novamente o menu `Product` e selecione `Build`.

Agora, altere a linha da `TitleScreen.m` que chamava o `background.png` para utilizar a constante da classe `Assets`:

```
//...
CCSprite *background = [CCSprite spriteWithFile:kBACKGROUND];
//...
```

Utilizaremos essa classe para adicionar outros *assets* posteriormente, quando os objetos inimigos e o player forem desenhados. É importante ter uma classe como essa para não espalhar suas imagens pelo código. Por exemplo, imagine que você queira alterar a imagem da nave principal. É melhor alterá-la em apenas um lugar, e fazer referência a essa variável nas classes necessárias.

4.6 CAPTURANDO CONFIGURAÇÕES INICIAIS DO DISPOSITIVO

Existem diversos dispositivos iOS atualmente, com diferentes tamanhos de tela. Existem algumas técnicas para tentar limitar esse problema durante o desenvolvimento do jogo. Utilizaremos aqui uma técnica simples para adaptar nosso conteúdo aos diversos dispositivos, capturando as medidas e utilizando-os sempre que for necessário lidar com essa questão.

Para iniciar as configurações de tela e criar a tela inicial, criaremos algumas macros:

- `SCREEN_WIDTH()`: Retorna a largura da tela
- `SCREEN_HEIGHT()`: Retorna a altura da tela
- `WIN_SIZE()`: Retorna o tamanho (largura e altura) da tela

O Cocos2D nos ajuda nesse momento, pois já possui objetos preparados para executar essa função. Podemos utilizar a classe `CCDirector` para conseguir os parâmetros da tela.

A vantagem de utilizar uma macro, é que caso haja necessidade, poderemos alterá-la para responder por diferentes tamanhos de tela para cada tipo de aparelho.

Director

O `CCDirector` é o componente principal que executa o game. É ele quem controla o FPS, tamanho da tela, resolução, e também cuida das transições entre scenes, ou seja, transições de telas do jogo. Ele é um `Singleton` que sabe qual tela está ativa no momento e gerencia uma pilha de telas, aguardando suas chamadas para fazer as transições.

Vamos criar a classe `DeviceSettings`, responsável por acessar o `CCDirector` e retornar as medidas e configurações do dispositivo.

Crie a classe `DeviceSettings`, subclasse de um `NSObject`, e no header dela declare as seguintes macros:

```
#define SCREEN_WIDTH() \
    [CCDirector sharedDirector].winSize.width

#define SCREEN_HEIGHT() \
    [CCDirector sharedDirector].winSize.height

#define WIN_SIZE() \
    [CCDirector sharedDirector].winSize
```

Como importaremos a `DeviceSettings` em diversas outras classes do projeto, vamos aproveitar e incluí-la em nosso arquivo `Prefix.pch`:

```
//...
#ifdef __OBJC__
#import <UIKit/UIKit.h>
#import <Foundation/Foundation.h>
#import "cocos2d.h"
#import "Assets.h"
#import "DeviceSettings.h"
#endif
```

Com isso, podemos refatorar o posicionamento do *background* no `init` da `TitleScreen.m` para ficar como a seguir.

```
//...
background.position = ccp(SCREEN_WIDTH() / 2.0f, SCREEN_HEIGHT() / 2.0f);
//...
```

O `CCDirector` é também responsável pela inicialização da tela de abertura.

Iniciando a tela de abertura

Tela inicial preparada! Agora precisamos fazer a transição, ou seja, devemos informar ao Cocos2D para iniciar a tela de abertura.

Sempre após iniciar o game, o Cocos2D irá chamar a scene padrão `IntroLayer`. Esta scene é uma tela bem simples e leve, que é rapidamente carregada ao abrir o aplicativo, evitando que a tela “pisque” a transição entre nossa *Splash Screen* e a tela inicial do game.

Primeiramente, vamos ajustar a orientação da imagem exibida em nossa `IntroLayer`. Altere o método `init` da `IntroLayer.m` para não girar nossa imagem de abertura:

```
-(id) init
{
    if( (self=[super init])) {

        // ask director for the window size
        CGSize size = [[CCDirector sharedDirector] winSize];

        CCSprite *background;

        if( UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPhone ) {
            background =
                [CCSprite spriteWithFile:@"Default-568h@2x.png"];
            //background = [CCSprite spriteWithFile:@"Default.png"];
            //background.rotation = 90;
        } else {
            background =
                [CCSprite spriteWithFile:@"Default-Landscape~ipad.png"];
        }
        background.position = ccp(size.width/2, size.height/2);

        // add the label as a child to this Layer
        [self addChild: background];
    }

    return self;
}
```

Agora vamos alterar a `IntroLayer.m` para que chame nossa `TitleScreen` após abertura do game. Um objeto muito importante do Cocos2D será utilizado para

esse controle. Utilizaremos o `CCDirector` novamente, dessa vez para apresentar uma nova tela, utilizando o método `replaceScene:` e passando como parâmetro a `TitleScreen`, que é nossa tela inicial.

```
#import "TitleScreen.h"
//...
@implementation IntroLayer
//...
-(void) onEnter
{
    [super onEnter];
    [[CCDirector sharedDirector] replaceScene:
        [CCTransitionFade transitionWithDuration:1.0
                                                scene:[TitleScreen scene] ]];
}
@end
```

Já é possível rodar o projeto e ver a tela de abertura com a *background* configurado! Faça o teste.

4.7 LOGO

Vamos utilizar a mesma ideia e colocar um logo do jogo no topo da tela.

O logo é uma imagem simples e imagens são coordenadas por objetos que chamamos de `Sprites`. Criaremos um `Sprite` de forma simples para posicionar o logo e utilizaremos o método `setPosition:` para que o Cocos2D saiba onde colocar o elemento.

Pra finalizar, basta adicionar o logo à tela inicial com o método `addChild:.` Mude o `init` de sua `TitleScreen.m`:

```
- (id)init
{
    self = [super init];
    if (self) {
        //...

        // Imagem de Logo
        CCSprite *title = [CCSprite spriteWithFile:kLOGO];
        title.position =
            ccp(SCREEN_WIDTH() / 2.0f, SCREEN_HEIGHT() - 130.0f);
```

```

        [self addChild:title];

    }
    return self;
}
@end

```

Ao rodar o projeto já temos as imagens de background e logo do jogo posicionados.

4.8 BOTÕES

Os botões são partes importantíssimas do jogo. É a partir deles que o usuário interage com o game e que recebemos comandos para transicionar as telas e, mais à frente, mover o player, atirar etc.

Utilizando o Cocos2D o trabalho com *inputs* de botões fica bem mais simples, não precisando detectar a posição do toque na tela e comparar com o posicionamento dos *Sprites*. Esse trabalho será feito pelo *framework* e o jogo pode se preocupar com a lógica em si.

No Cocos2D, a classe *CCMenuItem* e suas subclasses representam os botões, e a classe *CCMenu* representa o menu onde estes botões estarão posicionados. Com estas classes, o Cocos2D já sabe se estamos tocando em um botão, se estamos segurando-o, ou se o soltamos.

O que precisamos agora é:

- Criar os 4 botões: Play, Highscore, Help e Sound.
- Configurar suas posições
- Adicioná-los à tela inicial

No `init` da `TitleScreen.m`, criaremos nosso menu com os botões:

```

- (id)init
{
    self = [super init];
    if (self) {
        //...
    }
}

```

```
// Cria os botões
CCMenuItemSprite *playButton = [CCMenuItemSprite
    initWithNormalSprite:[CCSprite spriteWithFile:kPLAY]
    selectedSprite:[CCSprite spriteWithFile:kPLAY]
    target:self
    selector:@selector(playGame:)];
CCMenuItemSprite *highscoreButton = [CCMenuItemSprite
    initWithNormalSprite:[CCSprite spriteWithFile:kHIGHSCORE]
    selectedSprite:[CCSprite spriteWithFile:kHIGHSCORE]
    target:self
    selector:@selector(viewHighscore:)];
CCMenuItemSprite *helpButton = [CCMenuItemSprite
    initWithNormalSprite:[CCSprite spriteWithFile:kHELP]
    selectedSprite:[CCSprite spriteWithFile:kHELP]
    target:self
    selector:@selector(viewHelp:)];
CCMenuItemSprite *soundButton = [CCMenuItemSprite
    initWithNormalSprite:[CCSprite spriteWithFile:kSOUND]
    selectedSprite:[CCSprite spriteWithFile:kSOUND]
    target:self
    selector:@selector(toggleSound:)];

// Define as posições dos botões
playButton.position = ccp(0.0f, 0.0f);
highscoreButton.position = ccp(0.0f, -50.0f);
helpButton.position = ccp(0.0f, -100.0f);
soundButton.position = ccp((SCREEN_WIDTH() / -2.0f) + 70.0f,
    (SCREEN_HEIGHT() / -2.0f) + 70.0f);

// Cria o menu que terá os botões
CCMenu *menu = [CCMenu menuWithItems:playButton,
    highscoreButton,
    helpButton,
    soundButton,
    nil];

[self addChild:menu];

}
return self;
}
@end
```

Perceba que nossos botões foram criados utilizando a classe `CCMenuItemSprite`, que representa um botão com uma imagem (`sprite`). Existem diversos tipos de botões, tais como o `CCMenuItemLabel` para textos, o `CCMenuItemImage` para `UIImage`s, entre outros. Por padrão, os botões incluídos no menu serão centralizados no meio da tela. Ou seja, ao posicionar um botão na posição `(0, 0)`, ele aparecerá no centro da tela.

Quando criamos os botões, informamos os parâmetros `target` e `selector`, que indicam os métodos chamados cada vez que um botão for selecionado. Faltava apenas criar estes métodos em nossa `TitleScreen.m`:

```
- (void)playGame:(id)sender
{
    NSLog(@"Botão selecionado: Play");
}

- (void)viewHighscore:(id)sender
{
    NSLog(@"Botão selecionado: Highscore");
}

- (void)viewHelp:(id)sender
{
    NSLog(@"Botão selecionado: Help");
}

- (void)toggleSound:(id)sender
{
    NSLog(@"Botão selecionado: Som");
}
```

Rode o projeto e confira os botões recebendo os *inputs* no *console*.

4.9 CONCLUSÃO

O jogo deve estar como mostrado na tela abaixo, com *background* e logo configurados. Além disso, 4 botões foram implementados: Play, Highscore, Help e controle de Som.



Figura 4.7: Tela de abertura.

O *core* do desenvolvimento de um jogo não é fácil. Repare na evolução do protótipo para o jogo real que estamos criando e perceberá que partes complexas foram encapsuladas pelo Cocos2D.

Nesse capítulo, fomos um pouco mais a fundo em questões como telas (`CCScene`), camadas (`CCLayer`), menus (`CCMenu`) e botões (`CCMenuItem`). Utilizamos também um importante elemento do Cocos2D, o `CCDirector`.

A seguir, faremos a transição para a tela do jogo e teremos nossos primeiros elementos do game.

CAPÍTULO 5

Tela do jogo e objetos inimigos

Hora de adicionar perigo ao nosso game! Nesse capítulo iremos entrar na tela do jogo de fato, onde toda a ação ocorrerá. Essa será a principal parte do jogo e por isso trataremos em alguns capítulos.

Para iniciar, passaremos pela transição da tela de abertura para a tela de jogo. Além disso, colocaremos os inimigos na tela. Alguns conceitos importantes do `Cocos2D` serão utilizados nesse capítulo, cujo objetivo é ter a tela do game rodando, com alguns inimigos surgindo.

Utilizaremos muito do que já foi visto até aqui, como `CCLayers` para representar camadas de uma tela, `CCSprites` para controlar objetos e `CCScene` para criar a tela do jogo. Além disso o `CCDirector` será utilizado novamente.

No fim desse capítulo teremos a transição entre tela de abertura e tela do game, além dos objetos inimigos aparecendo na tela.



Figura 5.1: Meteoros inimigos.

5.1 GAMESCENE

Precisamos de uma tela para o jogo, para conter os elementos principais de interação do game como player, inimigos e controles. Assim como anteriormente, criaremos uma tela herdando da classe `CCLayer` do Cocos2D, para que possamos ter diversas camadas atuantes, como botões, inimigos, player, score etc.

Também como anteriormente, a definição de uma tela é criada através de um `CCScene`, que saberá lidar com as camadas da nossa classe.

O Maestro

Idealmente, essa classe não deve ter muitas responsabilidades, mas sim funcionar como um orquestrador de todos os elementos, ou seja, um maestro em uma orquestra, que dirige e comanda o que todos os outros elementos fazem e como eles interagem entre si.

Ela será a classe que inicializa objetos no jogo, que coloca objetos na tela, porém o comportamento de cada um deles será representado individualmente em cada classe correspondente.

Algumas das responsabilidades da `GameScene`, a classe maestro do jogo, devem ser:

- Iniciar a tela do game e organizar as camadas
- Adicionar objetos como player, inimigos e botões à essas camadas
- Inicializar cada um desses objetos
- Checar colisões entre objetos

A classe `GameScene` tem muita responsabilidade, porém não detém regras e lógicas de cada elemento. Outra função importante dessa classe é aplicar um dos conceitos vistos anteriormente, do *game loop*.

Vamos então criar a classe `GameScene` já colocando um background como fizemos anteriormente na tela de abertura. O *header* `GameScene.h` ficará assim:

```
@interface GameScene : CCLayer
```

```
+ (CCScene *)scene;
```

```
@end
```

E o *implementation* `GameScene.m`

```
#import "GameScene.h"
```

```
@implementation GameScene
```

```
+ (CCScene *)scene
```

```
{
```

```
    // 'scene' is an autorelease object.
```

```

    CCScene *scene = [CCScene node];

    // 'layer' is an autorelease object.
    GameScene *layer = [GameScene node];

    // add layer as a child to scene
    [scene addChild:layer];

    // return the scene
    return scene;
}

- (id)init
{
    self = [super init];
    if (self) {
        // Imagem de Background
        CCSprite *background = [CCSprite spriteWithFile:kBACKGROUND];
        background.position =
            ccp(SCREEN_WIDTH() / 2.0f, SCREEN_HEIGHT() / 2.0f);
        [self addChild:background];
    }
    return self;
}

@end

```

5.2 TRANSIÇÃO DE TELAS

Para que o jogo comece, precisamos fazer o link entre a tela de abertura e a tela do game!

Aqui utilizaremos o `CCDirector` que sabe manter uma `CCScene` ativa por vez. Além de trocar de uma tela para outra, o Cocos2D nos permite escolher e configurar detalhes dessa transição.

Utilizaremos o método `replaceScene:` que fará uma transição com o tempo de pausa entre uma tela e outro, gerando um efeito suave.

Para isso, na classe `TitleScreen`, mudamos o `playGame:` para que botão de *play* comece o jogo. Importe a `GameScene` na `TitleScreen.h`

```
#import "GameScene.h"
```

E mude o método `playGame::`

```
//...
- (void)playGame:(id)sender
{
    NSLog(@"Botão selecionado: Play");
    [[CCDirector sharedDirector]
     replaceScene:[CCTransitionFade transitionWithDuration:1.0
                                                         scene:[GameScene scene]]];
}
```

Rode o jogo e clique no botão *play*. O que acontece? Por enquanto, só temos a tela de *background*!

5.3 ENGINES

Temos a classe que orquestrará os objetos do game e criaremos agora classes responsáveis por gerenciar outros elementos. O primeiro elemento que teremos serão os inimigos. Nossos inimigos serão meteoros que cairão e precisarão ser destruídos pelo player.

Criaremos uma nova camada, um novo *layer* para representar esses inimigos. Como utilizado anteriormente, camadas são representadas por heranças ao `CCLayer` do Cocos2D.

Engine de objetos inimigos

Nossa camada de objetos inimigos, os meteoros, será responsável por criar inimigos e enviar à tela do jogo. Essa *engine* de meteoros não é responsável pelo movimento do meteoro em si, mas sim de controlar a aparição deles na tela e fazer o link entre objeto `Meteoro` e tela do Game.

precisaremos manter o link entre a tela principal do game e a *engine* de inimigos. Essa é uma parte complexa do desenvolvimento de games. Não é simples coordenar objetos com ciclos de vida diferentes que rodam pela aplicação. O que faremos aqui é utilizar o Design Pattern de *delegate* para auxiliar na comunicação entre os objetos.

Delegates são muito utilizados em games e aplicações iOS. Você já deve ter visto este conceito em diversas outras classes do iOS, como a `UITableView` e `UIScrollView` por exemplo.

A *engine* de inimigos sabe como e quando criar os inimigos. Mas apenas isto.

Nossa tela principal é que sabe em qual camada incluir o inimigo, além de conhecer todos os demais objetos (players, tiros etc) para poder checar a colisão entre eles.

Em nosso game, a tela principal do jogo será o *delegate* da *engine* de inimigos. Ou seja, a tela principal do jogo será responsável por “escutar” e tratar as instruções da *engine* de inimigo. Pense da seguinte maneira: a *engine* de inimigos irá dizer “Ei delegate! Eu quero criar o inimigo!”, e a tela principal irá responder “OK! Vou criar o inimigo. Deixe comigo que daqui pra frente eu cuido dele!”.

É importante que uma *Engine* saiba quando é o momento de colocar um novo elemento no jogo. Muitas vezes, principalmente para objetos inimigos, utilizamos números randômicos para definir a hora de colocar um novo inimigo na tela.

Essa ideia foi muito utilizada por jogos em que o nível mais difícil era apenas uma equação na qual o número randômico gerado satisfazia uma condição de entrada em um `if`. No código da nossa *engine* abaixo, faremos exatamente isso.

Vale citar que a *engine* é o código responsável por manter o loop de objetos, e com o Cocos2D, utilizamos métodos de agendamento para isso. Ou seja, criaremos um `schedule` para que a *engine* analise se deve ou não atualizar e incluir um novo objeto inimigo na tela.

Abaixo, o código da primeira *Engine* do game, a classe `MeteorsEngine`. No *header* `MeteorsEngine.h` iremos definir um *protocolo*, que é o método que o *delegate* (a `GameScene`) terá que implementar para responder às requisições da *engine* de inimigos:

```
@protocol MeteorsEngineDelegate;

@interface MeteorsEngine : CCLayer

@property (nonatomic, assign) id<MeteorsEngineDelegate>delegate;

+ (MeteorsEngine *)meteorEngine;

@end

@protocol MeteorsEngineDelegate <NSObject>
- (void)meteorsEngineDidCreateMeteor:(CCNode *)meteor;
@end
```

No *implementation* `MeteorsEngine.m`, vamos escrever a lógica de criação de meteoros:

```
#import "MeteorsEngine.h"

@implementation MeteorsEngine

+ (MeteorsEngine *)meteorEngine
{
    return [[[MeteorsEngine alloc] init] autorelease];
}

- (id)init
{
    self = [super init];
    if (self) {
        [self schedule:@selector(meteorsEngine:) interval:(1.0f/10.0f)];
    }
    return self;
}

- (void)meteorsEngine:(float)dt
{
    // sorte: 1 em 30 gera um novo meteoro!
    if(arc4random_uniform(30) == 0) {
        if ([self.delegate respondsToSelector:
            @selector(meteorsEngineDidCreateMeteor:)]){
            // Pedes para o delegate criar o meteoro
            // (por enquanto, não estamos informando qual o meteoro)
            [self.delegate meteorsEngineDidCreateMeteor:nil];
        }
    }
}

@end
```

Para fechar o *link* entre ambas as camadas, implementaremos o protocolo `MeteorsEngineDelegate` na tela do jogo, o que obrigará ela a ter um método para receber os objetos criados por essa `Engine` e colocá-los na tela.

Primeiramente, vamos informar na `GameScene.h` que ela deverá implementar o protocolo:

```
#import "MeteorsEngine.h"
```



```
@interface GameScene : CCLayer < MeteorsEngineDelegate>

+ (CCScene *)scene;

@end
```

E na `GameScene.m` crie o método que será responsável pelos meteoros que criaremos em seguida.

```
- (void)meteorsEngineDidCreateMeteor:(CCNode *)meteor
{
    // Aqui incluiremos o meteoro na tela
}
```

Mantendo as referências

Um outro ponto importante para o controle do jogo e toda a orquestração é manter todos os objetos criados de uma forma fácil para que possam ser analisados depois. Um exemplo nesse caso é a comparação com o tiro ou com o próprio player para detectar colisões, que serão tratados mais à frente.

Vamos guardar a referência de cada meteoro criado em uma propriedade `&&NSMutableArray` na classe `GameScene`, e também uma propriedade com referência à `engine` de inimigos. Declare estas propriedades na `GameScene.h`

```
#import "MeteorsEngine.h"

@interface GameScene : CCLayer < MeteorsEngineDelegate>

+ (CCScene *)scene;

// Engines
@property (nonatomic, retain) MeteorsEngine *meteorsEngine;

// Arrays
@property (nonatomic, retain) NSMutableArray *meteorsArray;

@end
```

O Cocos2D não suporta ARC (*Automatic Reference Counting*), por isso, todas as propriedades que criarmos como `retain` deverão ser retiradas da memória no método `dealloc` de nossas classes.

Crie o método `dealloc` no final da `GameScene.m` e inclua:

```
- (void)dealloc
{
    [_meteorsEngine release];
    [_meteorsArray release];
    [super dealloc];
}
```

Lembre-se de incluir o `dealloc` em todas as demais classes que criarmos com propriedades declaradas com `retain`.

5.4 METEOR

Chegamos ao objeto inimigo propriamente dito. As principais responsabilidades desse objeto são:

- Carregar imagem (`CCSprite`)
- Posicionar elemento na tela
- Guardar a posição do objeto para que o mesmo possa ser movimentado com o tempo

Esse é o primeiro objeto de jogo realmente que criaremos. Até o momento, criamos telas preenchendo-as com elementos, botões de menu e classes de *engine* para dar a base a esses elementos principais do jogo.

A primeira coisa a se fazer é adicionar o *asset* do meteoro na `Assets.h`.

```
@interface Assets : NSObject
#define kMETEOR @"meteor.png"
@end
```

Vamos criar a classe `Meteor`. Inicialmente, cada meteoro nasce no topo da tela (é o valor de `SCREEN_HEIGHT()`), e numa posição *x* randômica. Declare no *header* `Meteor.h` duas propriedades para guardar a posição *x* e *y* do meteoro, o método construtor `meteorWithImage:` e o método `start`, que será implementado posteriormente:

```
#import "CCSprite.h"

@interface Meteor : CCSprite

@property (nonatomic, assign) float positionX;
@property (nonatomic, assign) float positionY;

+ (Meteor *)meteorWithImage:(NSString *)image;

- (void)start;

@end
```

No *implementation* `Meteor.h`, nosso construtor irá sortear uma posição inicial do meteoro:

```
#import "Meteor.h"

@implementation Meteor

+ (Meteor *)meteorWithImage:(NSString *)image
{
    Meteor *meteor = [Meteor spriteWithFile:image];

    meteor.positionX = arc4random_uniform(SCREEN_WIDTH());
    meteor.positionY = SCREEN_HEIGHT();
    meteor.position = ccp(meteor.positionX, meteor.positionY);

    return meteor;
}

@end
```

Repare que o objeto meteoro permanece vivo na memória por um bom tempo. Ele é criado e, a cada *frame*, renderizado em uma posição diferente, dando a impressão de movimento.

Aqui mais uma vez o *framework* nos ajuda. Para que cada *frame* seja renderizado durante o jogo, e a posição do objeto mude com o passar do tempo, o Cocos2D nos permite escolher um método que será invocado de tempo em tempo. Isso será definido no `start`, fazendo `scheduleUpdate`, que invocará o método `update`.

```
//...
- (void)start
{
    [self scheduleUpdate];
}

- (void)update:(float)dt
{
    self.positionY -= 1;
    self.position = ccp(self.positionX, self.positionY);
}
```

VARIÁVEL DT DOS UPDATES

O Cocos2D vai **tentar** invocar o seu método `update` de x em x milissegundos, isso é, em cada frame. Mas, por uma série de motivos, o processador pode estar ocupado com outras coisas, fazendo com que essa chamada agendada não ocorra quando você queria. Ele pode demorar mais. Nesse caso, vai dar uma impressão que seu jogo está lento, já que a tela será renderizada como se só tivesse passado o tempo de um *frame*, mas na verdade pode ter passado alguns segundos.

Num jogo profissional, você deve guardar essa informação para decidir corretamente quantos *pixels* cada objeto deve mudar. No nosso caso, se o `dt` for maior que o de 1 *frame*, deveríamos descer o meteoro mais que 1 pixel, fazendo a regra de 3.

5.5 TELA DO GAME

Agora que temos nossa classe `Meteor`, vamos alterar o protocolo da `MeteorsEngine.h` para que passe um objeto meteoro como parâmetro:

```
#import "Meteor.h"
//...
@protocol MeteorsEngineDelegate <NSObject>
- (void)meteorsEngineDidCreateMeteor:(Meteor *)meteor;
@end
```

Na `MeteorsEngine.m` passaremos um novo meteoro como parâmetro para o `delegate`:

```
- (void)meteorsEngine:(float)dt
{
    // sorte: 1 em 30 gera um novo meteoro!
    if(arc4random_uniform(30) == 0) {
        if ([self.delegate respondsToSelector:
            @selector(meteorsEngineDidCreateMeteor:))]{
            // Pedes para o delegate criar o meteoro
            [self.delegate meteorsEngineDidCreateMeteor:
                [Meteor meteorWithImage:kMETEOR]];
        }
    }
}
```

Para fechar e criar o *link* entre a classe da tela do jogo, com a *engine* de meteoros e com os objetos meteoros criados, modificaremos a classe `GameScene`.

Primeiramente, vamos criar um método que conterà a inicialização dos objetos de jogo. Crie o método `addGameObjects`:

```
- (void)addGameObjects
{
    // Inicializa os Arrays
    self.meteorsArray = [NSMutableArray array];

    // Inicializa a Engine de Meteoros
    self.meteorsEngine = [MeteorsEngine meteorEngine];
}
```

No construtor, criaremos um *layer* especialmente para os meteoros e adicionaremos a tela do jogo via `addChild`. Declare a propriedade `meteorsLayer` na `GameScene.h`:

```
//...
// Layers
@property (nonatomic, retain) CCLayer *meteorsLayer;
```

No `init` da `GameScene.m` criaremos a *layer* de meteoros e invocaremos o `addGameObjetos`. Ficará assim:

```
- (id)init
{
    self = [super init];
```

```

if (self) {
    // Imagem de Background
    CCSprite *background = [CCSprite spriteWithFile:kBACKGROUND];
    background.position =
        ccp(SCREEN_WIDTH() / 2.0f, SCREEN_HEIGHT() / 2.0f);
    [self addChild:background];

    // CCLayer para os Meteoros
    self.meteorsLayer = [CCLayer node];
    [self addChild:self.meteorsLayer];

    [self addGameObjects];
}
return self;
}

```

E agora podemos alterar o método `meteorsEngineDidCreateMeteor:` para incluir o meteoro na tela e iniciá-lo:

```

- (void)meteorsEngineDidCreateMeteor:(Meteor *)meteor
{
    // A Engine de Meteoros indicou que um novo Meteoro foi criado
    [self.meteorsLayer addChild:meteor];
    [meteor start];
    [self.meteorsArray addObject:meteor];
}

```

Um método importante que utilizaremos aqui é o `onEnter`. Ele é invocado pelo Cocos2D assim que a tela do game está pronta para orquestrar os objetos do jogo. Ele será a porta de entrada do jogo. Por enquanto, simplesmente adicionaremos o `meteorsEngine` e indicaremos o `delegate` como `self`, para sermos avisados quando novos meteoros forem criados:

```

//...
- (void)onEnter
{
    [super onEnter];
    [self startGame];
}

- (void)startGame

```

```
{  
    [self startEngines];  
}  
  
- (void)startEngines  
{  
    [self addChild:self.meteorsEngine];  
    self.meteorsEngine.delegate = self;  
}
```

5.6 CONCLUSÃO

Temos agora a tela de abertura e a tela de jogo (onde o *game loop* é executado) em funcionamento. Nosso *game loop* inicializa os inimigos. Precisamos de um player para jogar contra eles, é o que veremos a seguir!

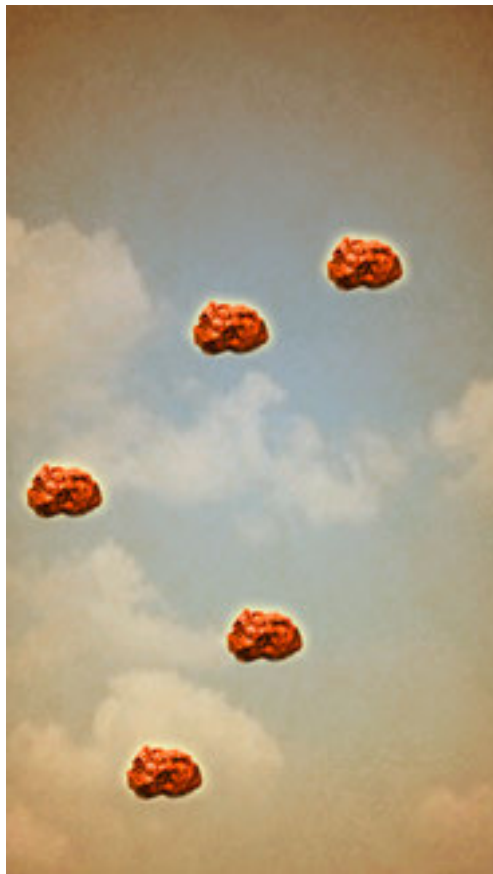


Figura 5.2: Meteoros inimigos.

CAPÍTULO 6

Criando o Player

Tela do jogo preparada e inimigos aparecendo! Cenário perfeito para iniciarmos o desenvolvimento do player! Essa é uma parte bem interessante no desenvolvimento de jogos, pois programaremos o objeto que será controlado pelos *inputs* do usuário.

Para isso, utilizaremos a maioria dos elementos do *framework* Cocos2D que vimos até agora para trabalhar com o player. Utilizaremos camadas, sprites e os conceitos vistos anteriormente.

Nossa tela de jogo precisará de mais uma camada, utilizaremos `Sprites` para o Player e detectaremos *inputs* do usuário para movê-lo.

Resumidamente, nesse capítulo faremos:

- Colocar o player na tela
- Movimentar o player
- E atirar!

Daremos um passo importante na construção do jogo nesse capítulo, o objetivo final é ter a cena a seguir, ainda sem detectar colisões.

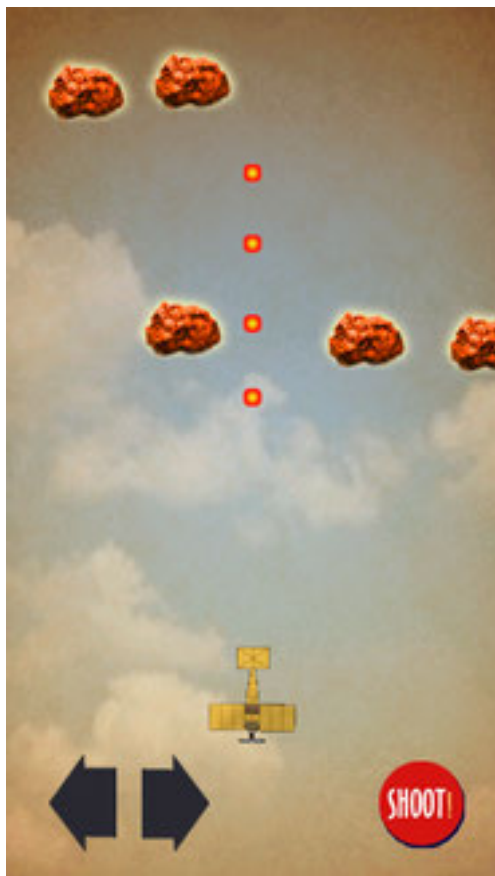


Figura 6.1: 14 bis atirando contra os meteoros.

6.1 DESENHANDO O PLAYER

Iniciaremos pela imagem, adicionando a figura do player no `Assets.h`.

```
@interface Assets : NSObject
# define kNAVE          @"nave.png"
@end
```

Criaremos o objeto principal e, como anteriormente, controlamos figuras e imagens herdando do `CCSprite` do Cocos2D.

Utilizaremos o método que retorna a largura da tela para centralizar o Player. Precisamos de variáveis que guardem essas posições pois precisaremos alterá-las mais à frente.

Como já utilizado pelas outras classes, manteremos o link entre tela de abertura e player utilizando um delegate.

Na `Player.h` iremos declarar nossas propriedades de posição do player, e o construtor.

```
@protocol PlayerDelegate;

@interface Player : CCSprite

@property (nonatomic, assign) id<PlayerDelegate>delegate;

@property (nonatomic, assign) float positionX;
@property (nonatomic, assign) float positionY;

+ (Player *)player;

@end

@protocol PlayerDelegate <NSObject>
// Vamos criar os métodos de delegate mais pra frente
@end
```

A `Player.m` será iniciada da seguinte maneira:

```
#import "Player.h"

@implementation Player

+ (Player *)player
{
    Player *player = [Player spriteWithFile:kNAVE];

    // Posiciona o Player recém criado
    player.positionX = SCREEN_WIDTH() / 2.0f;
    player.positionY = 120.0f;
    player.position = ccp(player.positionX, player.positionY);
}
```

```

    return player;
}

```

O objeto `Player` já está pronto para ser inicializado, mas ainda não existe uma camada na tela do jogo responsável por mostrá-lo. Para que o player apareça na tela do jogo, temos que adicionar mais uma camada. Essa camada terá o nome de `playerLayer`.

Na `GameScene.h` vamos declarar as propriedades da *layer* e do player, além de informar que ela implementará o protocolo `PlayerDelegate`:

```

#import "Player.h"
//...
@interface GameScene : CCLayer < MeteorsEngineDelegate, PlayerDelegate >
//...
@property (nonatomic, retain) CCLayer *playerLayer;
@property (nonatomic, retain) Player *player;

```

Na `GameScene.m` é necessário adicionar a variável de *layer* e iniciá-la no construtor. Após isso, adicione a camada criada através do método `addGameObjects`.

```

- (id)init
{
    self = [super init];
    if (self) {
        //...

        // CCLayer para o Jogador
        self.playerLayer = [CCLayer node];
        [self addChild:self.playerLayer];

        [self addGameObjects];
    }
    return self;
}

- (void)addGameObjects
{
    // Inicializa os Arrays
    self.meteorsArray = [NSMutableArray array];
}

```

```
// Inicializa a Engine de Meteoros
self.meteorsEngine = [MeteorsEngine meteorEngine];

// Cria o Player
self.player = [Player player];
self.player.delegate = self;
[self.playerLayer addChild:self.player];
}
```



Figura 6.2: 14 bis pronto para ação.

6.2 BOTÕES DE CONTROLE

Já temos o player aparecendo na tela. Além disso, ele já está em uma camada da tela do game, o que faz com que seja renderizado durante o jogo.

Vamos agora adicionar outros elementos à tela, para que o player possa ser comandado pelos *inputs* do usuário. Para isso, precisaremos de novas imagens para esses controles.

Iniciaremos adicionando 3 imagens, duas para movimentar o player entre direita e esquerda e outra que será o botão de atirar. Essas imagens serão incluídas no arquivo `Assets.h`.

```
@interface Assets : NSObject
#define kLEFTCONTROL @"left.png"
#define kRIGHTCONTROL @"right.png"
#define kSHOOTBUTTON @"shootButton.png"
@end
```

Para os botões, poderíamos utilizar novamente a classe `CCMenuItemSprite`. Entretanto, os objetos do tipo `CCMenuItemSprite` são ativados somente quando o usuário retira o dedo do botão. Este comportamento é aceitável para botões de menu, porém, para jogos o ideal é que o botão execute a ação no momento que o usuário tocá-lo, e não após retirar o dedo.

Para isto, criaremos a classe `CCMenuItemGameButton` que irá sobrescrever alguns métodos padrões da `CCMenuItemSprite`, a fim de que as ações dos botões sejam executadas logo quando o botão for tocado.

Crie a classe `CCMenuItemGameButton`, subclasse da `CCMenuItemSprite`. No arquivo header `CCMenuItemGameButton.h` não é necessário nenhuma alteração. Vamos apenas alterar os métodos de ativação e seleção do botão na `CCMenuItemGameButton.m`:

```
@implementation CCMenuItemGameButton

- (void)selected
{
    // Quando está "selected", aciona o "activate" para disparar o botão
    // antes de o jogador tirar o dedo da tela
    [super activate];
}
```

```
- (void)activate
{
    // Não chama mais o super "activate" aqui, já que este foi
    // acionado no método "selected"
    // [super activate];
}
```

@end

Pronto, temos nosso botão customizado criado. Agora vamos incluir os botões na tela alterando a classe `GameScene`. Vamos importar a classe de botões e definir sua camada, a declarando na `GameScene.h`:

```
#import "CCMenuItemGameButton.h"
//...
@property (nonatomic, retain) CCLayer *gameButtonsLayer;
```

No arquivo de implementação `GameScene.m`, vamos instanciar a camada dos botões e criar o menu. Perceba que os botões serão criados da mesma forma que fizemos na `TitleScreen`, mas desta vez utilizaremos nossa classe `CCMenuItemGameButton`:

```
- (id)init
{
    self = [super init];
    if (self) {
        //...
        // CCLayer para os Botões
        self.gameButtonsLayer = [CCLayer node];
        [self addChild:self.gameButtonsLayer];

        // Cria os botões
        CCMenuItemGameButton *leftControl = [CCMenuItemGameButton
            initWithNormalSprite:[CCSprite spriteWithFile:kLEFTCONTROL]
            selectedSprite:[CCSprite spriteWithFile:kLEFTCONTROL]
            target:self
            selector:@selector(moveLeft:)];
        CCMenuItemGameButton *rightControl = [CCMenuItemGameButton
            initWithNormalSprite:[CCSprite spriteWithFile:kRIGHTCONTROL]
            selectedSprite:[CCSprite spriteWithFile:kRIGHTCONTROL]
            target:self
```



```

        selector:@selector(moveRight:)]];
CCMenuItemGameButton *shootButton = [CCMenuItemGameButton
    initWithNormalSprite:[CCSprite spriteWithFile:kSHOOTBUTTON]
    selectedSprite:[CCSprite spriteWithFile:kSHOOTBUTTON]
    target:self
    selector:@selector(shoot:)]];

// Define as posições dos botões
leftControl.position = ccp(-110.0f,
    (SCREEN_HEIGHT() / -2.0f) + 50.0f);
rightControl.position = ccp(-50.0f,
    (SCREEN_HEIGHT() / -2.0f) + 50.0f);
shootButton.position = ccp((SCREEN_WIDTH() / 2.0f) - 50.0f,
    (SCREEN_HEIGHT() / -2.0f) + 50.0f);

// Cria o menu que terá os botões
CCMenu *menu = [CCMenu menuWithItems:leftControl,
    rightControl,
    shootButton,
    nil];
[self.gameButtonsLayer addChild:menu];

[self addGameObjects];
}
return self;
}

```

Por fim, é necessário criar os métodos que receberão os *inputs* dos botões:

```

- (void)moveLeft:(id)sender
{
    NSLog(@"Botão selecionado: Esquerda");
}

- (void)moveRight:(id)sender
{
    NSLog(@"Botão selecionado: Direita");
}

- (void)shoot:(id)sender
{

```

```
NSLog(@"Botão selecionado: Atirar");  
}
```

Rode o jogo e aperte os botões. Veja que eles respondem logo quando você os toca, e não quando retira o dedo da tela!

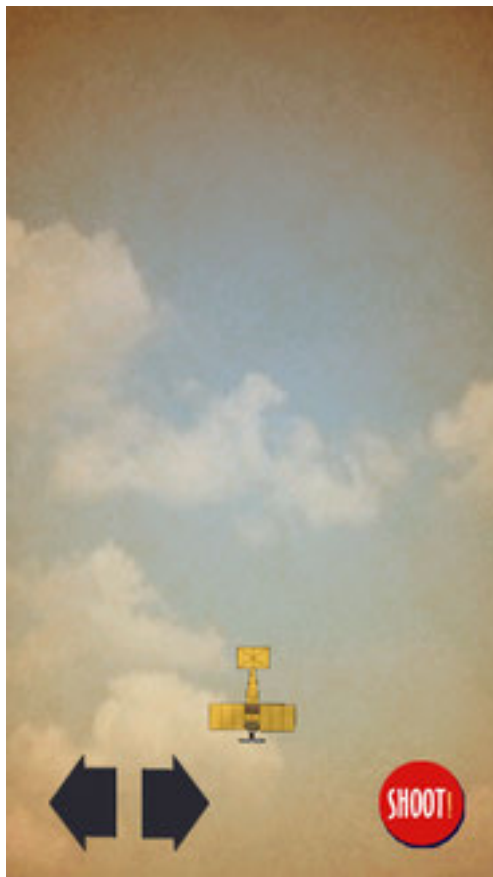


Figura 6.3: Controles de direção e tiro.

Imagens posicionadas e preparadas. Hora de começar a ação!

6.3 ATIRANDO

Falaremos agora de uma parte do jogo que pode parecer simples a princípio, mas tem impacto em muitas outras partes para que funcione: o tiro! Para que a nave

atire, precisaremos de uma série de coisas, portanto, vamos primeiro listar o que será necessário para organizar o pensamento antes de ir para o código.

- Um novo *asset*, ou seja, imagem do tiro
- Uma classe que represente o tiro como um `CCSprite`
- Definir o posicionamento do tiro na tela
- Uma engine responsável por criar um tiro
- Uma camada na tela do jogo para os tiros
- Associar o tiro e o player, para que o tiro saia do player

Há muita coisa para que o tiro de fato aconteça, porém, se listarmos cada uma dessas dependências podemos simplificar o desenvolvimento. Iniciaremos adicionando a figura do tiro na classe `Assets.h`.

```
@interface Assets : NSObject
#define kSHOOTBUTTON @"shootButton.png"
@end
```

Podemos iniciar a programação do tiro. Antes de pensar em fazer a nave atirar ou algo assim, vamos tentar pensar em “o que é o tiro?”. O tiro é um *sprite*, ou seja, uma imagem, que anda pela tela de baixo para cima. Assim, uma vez que um tiro aparece na tela, precisamos movimentá-lo para cima com o passar do tempo.

Para gerar essa sensação de movimento e controlar esses *updates* do posicionamento do tiro no eixo vertical, criaremos um método com o nome `update:`. Esse método será executado pelo Cocos2D a cada iteração. O *framework* manda como parâmetro um tempo de execução, para que possa ser analisado se algo deve ou não ser executado desde a última vez que ele invocou esse método. No nosso caso, esse parâmetro não será utilizado, pois a lógica do tiro é pelo toque no botão e não por uma regra de tempo.

A classe de tiro precisa manter o link com a tela de jogo, então utilizaremos o *delegate*. Criaremos também um método chamado `start`, que será utilizado para indicar que o tiro está funcionando.

Declare as propriedades e métodos na `Shoot.h`:

```
@interface Shoot : CCSprite

@property (nonatomic, assign) float positionX;
@property (nonatomic, assign) float positionY;

+ (Shoot *)shootWithPositionX:(float)positionX
    andPositionY:(float)positionY;

- (void)start;

@end
```

Vamos implementar os métodos na `Shoot.m`:

```
@implementation Shoot

+ (Shoot *)shootWithPositionX:(float)positionX
    andPositionY:(float)positionY
{
    Shoot *shoot = [Shoot spriteWithFile:kSHOOT];

    // Posiciona o Tiro recém criado no ponto indicado
    shoot.positionX = positionX;
    shoot.positionY = positionY;
    shoot.position = ccp(shoot.positionX, shoot.positionY);

    return shoot;
}

- (void)start
{
    // Inicia a Animação / Movimentação do Tiro
    [self scheduleUpdate];
}

- (void)update:(float)dt
{
    // Move o Tiro para cima
    self.positionY += 2;
    self.position = ccp(self.positionX, self.positionY);
}
```

@end

Tiro e tela de jogo

Até aqui, a classe de tiro foi definida. Agora vamos à tela do jogo para adicionar esse novo elemento. Duas coisas são necessárias nesse momento. A primeira é a camada do Cocos2D para que os tiros apareçam. A segunda é um `NSMutableArray` que guardará os tiros para que possamos detectar a colisão com os meteoros. Altere a `GameScene.h`:

```
#import "Shoot.h"
//...
@property (nonatomic, retain) CCLayer *shootsLayer;
@property (nonatomic, retain) NSMutableArray *shootsArray;
```

Além de criar as propriedades, é necessário inicializá-las. No construtor criaremos a camada.

```
- (id)init
{
    self = [super init];
    if (self) {
        //...
        // CCLayer para os Tiros
        self.shootsLayer = [CCLayer node];
        [self addChild:self.shootsLayer];
    }
    return self;
}
```

E no método `addGameObjects` criaremos o `array`.

```
- (void)addGameObjects
{
    //...
    self.shootsArray = [NSMutableArray array];
    //...
}
```

Atirando!

Já temos a classe do tiro e também a preparação na tela de jogo para o *link* entre esses dois objetos. O que fizemos até aqui foi preparar a estrutura para que o tiro aconteça. Vamos nos concentrar agora na relação entre o tiro e o Player.

Nesse momento, iremos definir o método que o player notificará o *delegate* de que ele está atirando. Inclua o método no `PlayerDelegate` da `Player.h`:

```
#import "Shoot.h"
//...
@protocol PlayerDelegate <NSObject>
- (void)playerDidCreateShoot:(Shoot *)shoot;
@end
```

Nesse momento, implementaremos a interface `PlayerDelegate` na `GameScene`. Dessa forma, a tela de jogo saberá o que deve fazer quando for requisitada para atirar. A interface obriga a criação do método `playerDidCreateShoot:`. Nele, um novo tiro, que é recebido como parâmetro, é adicionado à camada e ao *array* de tiros. Além disso, chama o método `start` da classe `Shoot`, permitindo que ela controle o que for necessário lá dentro.

Primeiramente, implemente o método `playerDidCreateShoot:` na `GameScene.m`

```
- (void)playerDidCreateShoot:(Shoot *)shoot
{
    // O Player indicou que um novo Tiro foi criado
    [self.shootsLayer addChild:shoot];
    [shoot start];
    [self.shootsArray addObject:shoot];
}
```

Criaremos na classe `Player` um método chamado `shoot` para disparar os tiros. Precisamos disso por um fato muito importante, que é o posicionamento inicial do tiro. Lembre-se que o tiro deve sair da nave, portanto o `Player` e seu posicionamento são muito importantes. O método `shoot` conterà todas as variáveis de posicionamento da nave na hora do tiro.

Declare o método na `Player.h`:

```
- (void)shoot;
```

Implemente o código de tiro na `Player.m`:

```
- (void)shoot
{
    // Aqui será disparado o tiro!
}
```

Apertando o botão!

A `GameScene.m` é quem tem o menu de botões e recebe o *input* do usuário, ou seja, o momento em que o botão é pressionado, e consequentemente o tiro disparado.

Vamos alterar o método `shoot:`, disparado pelo botão de tiro, para chamar o método de atirar do `Player`:

```
- (void)shoot:(id)sender
{
    NSLog(@"Botão selecionado: Atirar");
    [self.player shoot];
}
```

Player atirando

Tudo pronto para atirar a partir do botão de tiro pressionado! Temos a classe de tiro definida, o botão de tiro programado e a tela de jogo com a camada e métodos necessários prontos. Precisamos que alguém dê o comando de atirar e nada melhor que o próprio `Player` para fazer isso! Vamos alterar o método `shoot` da `Player.m` para que capture o posicionamento da nave e chame seu *delegate* para criação do tiro. Bang!

```
- (void)shoot
{
    // Atira
    if ([self.delegate respondsToSelector:
        @selector(playerDidCreateShoot:)]) {
        [self.delegate playerDidCreateShoot:
            [Shoot shootWithPositionX:self.positionX
                               andPositionY:self.positionY]];
    }
}
```

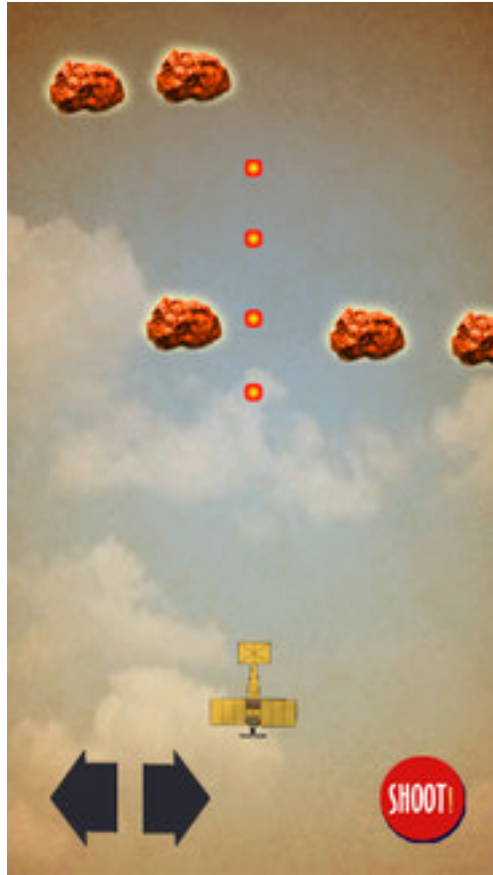


Figura 6.4: Player atirando.

6.4 MOVENDO O PLAYER

Vamos fechar esse capítulo com uma das partes mais importantes do jogo. Após atirar, moveremos o Player para esquerda e direita. Utilizaremos a mesma estratégia de atirar, mas agora as coisas serão mais simples.

Para iniciar, o `Player` deve saber se mover. Movimentar o player é fazer com que sua posição X seja atualizada quando um evento for detectado.

Na classe `Player` adicionaremos dois novos métodos, que quando chamados, mudam a posição horizontal da nave. Declare os métodos na `Player.h`.

```
- (void)moveLeft;
```



```
- (void)moveRight;
```

Vamos implementar os métodos de movimentação na `Player.m`:

```
- (void)moveLeft
{
    // Move o Player para a Esquerda
    if (self.positionX > 30.0f) {
        self.positionX -= 10.0f;
    }
    self.position = ccp(self.positionX, self.positionY);
}

- (void)moveRight
{
    // Move o Player para a Direita
    if (self.positionX < SCREEN_WIDTH() - 30.0f) {
        self.positionX += 10.0f;
    }
    self.position = ccp(self.positionX, self.positionY);
}
```

Na `GameScene.m` alteraremos os métodos para que essas ações sejam chamadas pelos botões.

```
- (void)moveLeft:(id)sender
{
    NSLog(@"Botão selecionado: Esquerda");
    [self.player moveLeft];
}

- (void)moveRight:(id)sender
{
    NSLog(@"Botão selecionado: Direita");
    [self.player moveRight];
}
```

Ao rodar o projeto, devemos ter a nave se movimentando a partir dos toques no comando de controle da nave. Além disso ela já atira de acordo com a posição do Player.

6.5 CONCLUSÃO

Esse é um capítulo muito importante para o desenvolvimento do jogo. Além de usar diversos elementos do *framework* Cocos2D, como camadas, sprites e agendamento (*update*), diversos conceitos de jogos foram usados, além de práticas como *delegates* e *engines*.

O jogo deve estar como na figura abaixo.

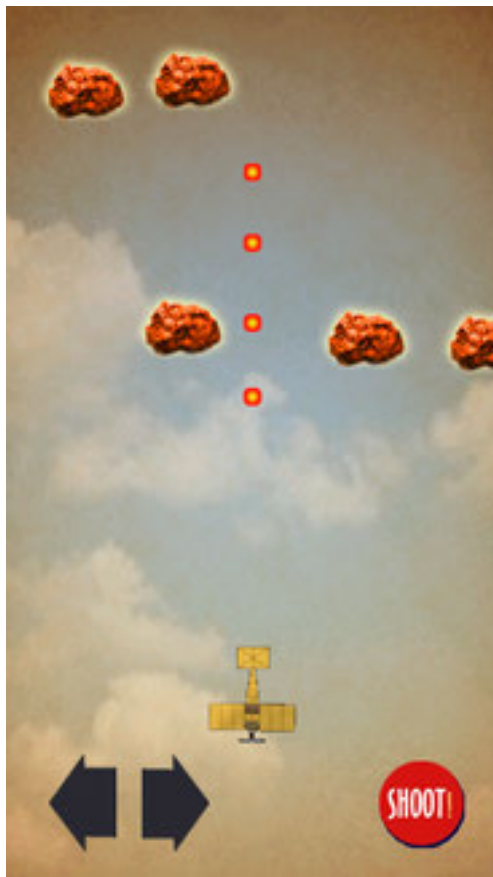


Figura 6.5: 14 bis atirando contra os meteoros.

É hora de verificar as colisões!

CAPÍTULO 7

Detectando colisões, pontuando e criando efeitos

Colisões são o coração dos games. Seja um soco de um personagem no outro, seja o personagem principal capturando algum elemento, ou como no jogo que estamos programando, um tiro que atinge um meteoro.

Detectar que um elemento encostou em outro é um assunto que pode ser muito complexo. Como mostrado no capítulo do protótipo que desenvolvemos, podemos detectar colisões considerando figuras geométricas em volta dos elementos para facilitar.

No capítulo atual, detectaremos colisões em duas situações.

- Quando um tiro atinge um meteoro
- Quando um meteoro atinge o avião (game over)

Veremos aqui mais uma vez que utilizar um *framework* de desenvolvimento de jogos como o Cocos2D ajuda muito nesse trabalho.

Uma vez detectadas as colisões, utilizaremos os efeitos do Cocos2D para gerar uma animação quando a detecção ocorrer. Em outra parte importante desse capítulo, falaremos sobre a atualização do placar. Essa deve ser uma parte tranquila pois utilizará conceitos já vistos anteriormente, como *camadas* e *sprites*.

Esse capítulo trará novos códigos e mais utilização do *framework* Cocos2D, sendo um capítulo chave para o desenvolvimento do jogo.

7.1 DETECTANDO COLISÕES

A primeira coisa que precisamos para identificar a colisão entre objetos no game é definir uma estratégia para isso. Nesse jogo, a estratégia será analisar um grupo de objetos em um *array*, com cada objeto de um outro grupo. Ou seja, dados dois *arrays*, verificar se algum elemento de um *array* sobrepõe outro.

Na `GameScene.h` criaremos um `NSMutableArray` para o player que, inicialmente, terá um único jogador.

```
@property (nonatomic, retain) NSMutableArray *playersArray;
```

E adicionaremos esse player no *array* de objetos na `GameScene.m`. Mais pra frente você poderá evoluir o jogo e controlar mais de um player.

```
- (void)addGameObjects
{
    //...
    // Insere o Player no array de Players
    self.playersArray = [NSMutableArray array];
    [self.playersArray addObject:self.player];
}
```

Definindo as bordas

Precisamos agora de uma forma de definir as bordas ou limites do tiro, da nave e dos meteoros, para que seja possível fazer a detecção da colisão.

No protótipo, utilizamos a estratégia de círculos. Para o jogo atual, utilizaremos uma estratégia de quadrados ou retângulos.

Como estamos utilizando o Cocos2D e seus `Sprites` podemos mais facilmente conseguir essas informações.

Precisaremos de um método do `Sprite` que devolva um retângulo, que contenha as bordas do elemento. Para isso, utilizaremos um método que existe nos pró-

prios *Sprites* chamado `boundingBox`. Esse método devolve um tipo `CGRect`, que representa os contornos da figura mapeados em forma retangular.

Com esse método, sabemos as coordenadas do posicionamento do objeto a ser analisado.

Checando a colisão

Uma vez que já conseguimos os valores de um elemento do jogo, suas bordas e posição na tela, podemos utilizar a estratégia que falamos no começo do capítulo para identificar se um elemento colide com outro durante o jogo.

Criaremos um método importante para checar as colisões, que precisa de alguns parâmetros para funcionar. Os dois primeiros são *arrays* de objetos a serem verificados. Ou seja, se queremos checar se os tiros estão colidindo com os meteoros, passaremos esses dois arrays. Outro ponto importante é passar uma referência da tela de jogo, no caso a `GameScene`. Precisamos disso para poder executar algum método caso a colisão seja detectada, porém como diversas colisões podem ser detectadas, como nave com tiro ou tiro com meteoro, isso será decidido em tempo de execução. É exatamente por isso que precisamos, por último, de um parâmetro a mais, que recebe o nome do método a ser executado caso a colisão aconteça.

O que teremos então é uma verificação de cada elemento do primeiro array, com cada elemento do segundo array. Caso a detecção aconteça, faremos um log por enquanto, e mostraremos quais elementos tiveram a colisão.

Na `GameScene.m` teremos o código a seguir.

```
- (BOOL)checkRadiusHitsOfArray:(NSArray *)array1
    againstArray:(NSArray *)array2
    withSender:(id)sender
    andCallbackMethod:(SEL)callbackMethod
{
    BOOL result = NO;

    for (int i = 0; i < [array1 count]; i++) {
        // Pega objeto do primeiro array
        CCSprite *obj1 = [array1 objectAtIndex:i];
        CGRect rect1 = obj1.boundingBox;

        for (int j = 0; j < [array2 count]; j++) {
            // Pega objeto do segundo array
            CCSprite *obj2 = [array2 objectAtIndex:j];
```

```

        CGRect rect2 = obj2.boundingBox;

        // Verifica colisão
        if (CGRectIntersectsRect(rect1, rect2)) {
            NSLog(@"Colisão Detectada: %@",
                  NSStringFromSelector(callbackMethod));
            result = YES;

            // Se encontrou uma colisão, sai dos 2 loops
            i = [array1 count] + 1;
            j = [array2 count] + 1;
        }
    }
}

return result;
}

```

Tendo o método que identifica a colisão como fizemos, podemos utilizá-lo para a verificação de dois *arrays* quaisquer. Faremos a chamada a ele para detectar colisões entre tiros e meteoros e entre meteoros e a nave. Esse método deve ser chamado de tempos em tempos pelo *Cocos2D*, portanto, utilizaremos mais uma vez o agendamento do *framework*, passando esse método para o `schedule::`.

Ainda na `GameScene.m` faremos essas chamadas.

```

- (void)checkHits:(float)dt
{
    // Checa se houve colisão entre Meteoros e Tiros
    [self checkRadiusHitsOfArray:self.meteorsArray
        againstArray:self.shootsArray
        withSender:self
        andCallbackMethod:@selector(meteorHit:withShoot:)];

    // Checa se houve colisão entre Jogador(es) e Meteoros
    [self checkRadiusHitsOfArray:self.playersArray
        againstArray:self.meteorsArray
        withSender:self
        andCallbackMethod:@selector(playerHit:withMeteor:)];
}

```

Como citado anteriormente, agendaremos o método de checagem de colisões.

Para isso, o método `schedule:` receberá o método `checkHits:` como parâmetro.

```
- (void)startGame
{
    // Ao entrar na GameScene, inicia a checagem de colisões
    // e inicia as Engines do jogo
    [self schedule:@selector(checkHits:)];
    [self startEngines];
}
```

Rode o jogo e repare no console do Xcode. A colisão já está sendo detectada, e portanto, podemos partir para a atualização do placar e criação dos efeitos necessários!

7.2 EFEITOS

No momento que detectamos que dois objetos do jogo colidiram, algumas coisas devem ser feitas. Vamos listar para facilitar o fluxo a seguir.

- Executar uma animação, como explosão ou similar
- Remover os elementos dos *arrays*
- Atualizar o placar ou mostrar tela de game over

Vamos começar pela animação. Existem diversas possibilidades de animação utilizando o Cocos2D. O *framework* oferece alguns efeitos tradicionais como *fade* e *scale*. A ideia é configurar uma série de ações, que juntas e em um espaço curto de tempo, criam uma animação.

Para o primeiro exemplo, vamos animar o meteoro quando é atingido pelo tiro. Nesse momento, animaremos o meteoro para que fique pequeno, dando a impressão que sumiu por ser atingido.

O que faremos abaixo é:

- Reduzir a escala de tamanho da imagem
- Retirar da tela com um efeito leve, chamado *fade out*
- Rodar ambas em sequência

Após rodar a sequência de animações, precisamos retirar o meteoro do array e da memória, pois ele não existe mais. Precisamos também parar o agendamento desse objeto, que roda de tempos em tempos atualizando sua posição e gerando o movimento.

Teremos um novo método na `Meteor`, que será invocado quando houver a colisão. Defina-o na `Meteor.h`:

```
- (void)gotShot;
```

Agora implemente o método na `Meteor.m`:

```
- (void)gotShot
{
    // Para o agendamento
    [self unscheduleUpdate];

    // Cria efeitos
    float dt = 0.2f;
    CCScaleBy *a1 = [CCScaleBy initWithDuration:dt scale:0.5f];
    CCFadeOut *a2 = [CCFadeOut initWithDuration:dt];
    CCSpawn *s1 = [CCSpawn arrayWithArray:
        [NSArray arrayWithObjects:a1, a2, nil]];

    // Método a ser executado após efeito
    CCCallFunc *c1 = [CCCallFunc initWithTarget:self
        selector:@selector(removeMe)];

    // Executa efeito
    [self runAction:[CCSequence arrayWithArray:
        [NSArray arrayWithObjects:s1, c1, nil]]];
}
```

No método acima cancelamos o agendamento da atualização de posição e criamos as ações que juntas farão o efeito de sumir do meteoro.

O Cocos2D possui ainda um método que pode ser invocado para que o objeto seja liberado da memória. Esse método é o `removeFromParentAndCleanup`: e será invocado logo após a animação acabar. Fazemos isso via `CCCallFunc`, pois queremos que o método `removeMe`: seja invocado depois de a animação terminar.

```
- (void)removeMe:(id)sender
{
```

```
// Quando o Meteoro é removido, limpa a memória utilizada pelo mesmo
[self removeFromParentAndCleanup:YES];
}
```

Animando o tiro

Utilizaremos o mesmo pensamento para animar o tiro quando colidir com um meteoro. Para isso, criaremos o método `explode` na classe `Shoot`, alterando apenas alguns parâmetros.

Esse método terá que executar algumas ações:

- Parar o agendamento da movimentação do meteoro
- Animar a explosão do meteoro
- Limpar o objeto da memória

Vamos declarar o método na `Meteor.h`:

```
- (void)explode;
```

Agora implemente o método na `Meteor.m`:

```
- (void)explode
{
    // Para o agendamento
    [self unscheduleUpdate];

    // Cria efeitos
    float dt = 0.2f;
    CCScaleBy *a1 = [CCScaleBy initWithDuration:dt scale:2.0f];
    CCFadeOut *a2 = [CCFadeOut initWithDuration:dt];
    CCSpawn *s1 = [CCSpawn initWithArray:
        [NSArray arrayWithObjects:a1, a2, nil]];

    // Método a ser executado após efeito
    CCCallFunc *c1 = [CCCallFunc initWithTarget:self
        selector:@selector(removeMe)];

    // Executa efeito
    [self runAction:[CCSequence initWithArray:
        [NSArray arrayWithObjects:s1, c1, nil]]];
}
```

Para limpar o objeto da memória, chamaremos novamente o método do `CoCos2D` chamado `removeFromParentAndCleanup:`.

```
- (void)removeMe:(id)sender
{
    // Quando o Tiro é removido, limpa a memória utilizada pelo mesmo
    [self removeFromParentAndCleanup:YES];
}
```

Já temos o código que o meteoro deve executar quando uma colisão for detectada. Agora precisamos fazer a chamada a ele no momento de colisão. Voltando à classe `GameScene` temos o método `checkRadiusHitsOfArray:againstArray:withSender:andCallbackMethod:`, que percorre dois arrays e verifica intersecções.

Nesse método, quando dois objetos estiverem colidindo, algo deve ser feito. Essa é uma parte importante do jogo e precisa de muito cuidado. Repare que o nosso método de detecção é genérico, ou seja, ele recebe dois *arrays* de objetos e analisa se ocorre colisão entre eles. No momento que essa colisão é detectada, ele precisará executar algo que pode ser a explosão da nave ou a explosão de um meteoro.

É importante perceber que a decisão de qual método rodar após detectada a colisão é em tempo de execução, e precisaremos invocá-lo via `performSelector:`.

```
No      arquivo      GameScene.m      altere      o      método
checkRadiusHitsOfArray:againstArray:withSender:andCallbackMethod:..

//...
// Verifica colisão
if (CGRectIntersectsRect(rect1, rect2)) {
    NSLog(@"Colisão Detectada: %@",
          NSStringFromSelector(callbackMethod));
    result = YES;

    // Se o sender possui o método indicado na chamada do método,
    // executa o mesmo com os objetos encontrados nos arrays
    if ([sender respondsToSelector:callbackMethod]) {
        [sender performSelector:callbackMethod
                  withObject:[array1 objectAtIndex:i]
                  withObject:[array2 objectAtIndex:j]];
    }

    // Se encontrou uma colisão, sai dos 2 loops
```

```

    i = [array1 count] + 1;
    j = [array2 count] + 1;
}

```

A partir daqui, basta fazer as chamadas aos métodos `gotShot` e `explode`. Na `GameScene.m` adicione o método `meteorHit:withShoot::`

```

- (void)meteorHit:(id)meteor withShoot:(id)shoot
{
    // Quando houve uma colisão entre Meteoro e Tiro, indica que
    // o Meteoro foi atingido e que o Tiro deve explodir
    if ([meteor isKindOfClass:[Meteor class]]) {
        [(Meteor *)meteor gotShot];
    }
    if ([shoot isKindOfClass:[Shoot class]]) {
        [(Shoot *)shoot explode];
    }
}

```

Rode o projeto e teste o que fizemos até aqui!

Removendo objetos

Precisamos lembrar que embora os objetos não estejam mais aparecendo na tela eles continuam alocados na memória, ou seja, ainda não foram removidos. Criaremos agora o código que eliminará os objetos dos arrays após colisão entre tiro e meteoro.

Primeiramente, vamos criar um *delegate* para nossa classe `Meteor` para que ela avise quando deverá ser removida. Declare o protocolo e *delegate* na `Meteor.h`:

```

@protocol MeteorDelegate;

@interface Meteor : CCSprite

@property (nonatomic, assign) float positionX;
@property (nonatomic, assign) float positionY;

@property (nonatomic, assign) id<MeteorDelegate>delegate;

+ (Meteor *)meteorWithImage:(NSString *)image;

```

```

- (void)start;
- (void)gotShot;

@end

@protocol MeteorDelegate <NSObject>
- (void)meteorWillBeRemoved:(Meteor *)meteor;
@end

```

Altere o método `gotShot` para que o meteoro notifique seu delegate de que ele será removido:

```

- (void)gotShot
{
    //...
    // Notifica delegate
    if ([self.delegate respondsToSelector:
        @selector(meteorWillBeRemoved:)]) {
        [self.delegate meteorWillBeRemoved:self];
    }
}

```

Faremos a mesma coisa para a classe `Shoot`. Declare o protocolo e delegate na `Shoot.h`:

```

@protocol ShootDelegate;

@interface Shoot : CCSprite

@property (nonatomic, assign) id<ShootDelegate>delegate;

@property (nonatomic, assign) float positionX;
@property (nonatomic, assign) float positionY;

+ (Shoot *)shootWithPositionX:(float)positionX
    andPositionY:(float)positionY;

- (void)start;
- (void)explode;

@end

```

```
@protocol ShootDelegate <NSObject>
- (void)shootWillBeRemoved:(Shoot *)shoot;
@end
```

Altere o método `explode` para que o meteoro notifique seu *delegate* de que ele será removido:

```
- (void)explode
{
    //...
    // Notifica delegate
    if ([self.delegate respondsToSelector:
        @selector(shootWillBeRemoved:)]) {
        [self.delegate shootWillBeRemoved:self];
    }
}
```

Agora, na nossa `GameScene`, toda vez que criarmos um novo `Meteor` ou `Shoot`, queremos possibilitar que eles avisem a própria `GameScene` de que serão removidos. Em outras palavras, a `GameScene` será o *delegate* do `Meteor` e do `Shoot`. Informe na `GameScene.h` de que ela implementará os protocolos `MeteorDelegate` e `ShootDelegate`:

```
@interface GameScene : CCLayer < MeteorsEngineDelegate,
                                PlayerDelegate,
                                MeteorDelegate,
                                ShootDelegate>
```

Agora altere os métodos `playerDidCreateShoot:` e `meteorsEngineDidCreateMeteor:` na `GameScene.m` para que ele defina os *delegates*:

```
- (void)playerDidCreateShoot:(Shoot *)shoot
{
    // O Player indicou que um novo Tiro foi criado
    [self.shootsLayer addChild:shoot];
    shoot.delegate = self;
    [shoot start];
    [self.shootsArray addObject:shoot];
}
```

```
- (void)meteorsEngineDidCreateMeteor:(Meteor *)meteor
{
    // A Engine de Meteoros indicou que um novo Meteoro foi criado
    [self.meteorsLayer addChild:meteor];
    meteor.delegate = self;
    [meteor start];
    [self.meteorsArray addObject:meteor];
}
```

Feito isso, basta fazer a remoção dos objetos na `GameScene.m`, implementando os métodos `meteorWillBeRemoved:` e `shootWillBeRemoved:` dos protocolos:

```
- (void)meteorWillBeRemoved:(Meteor *)meteor
{
    // Após atingido, um Meteoro notifica a GameScene
    // para que seja removido do Array
    meteor.delegate = nil;
    [self.meteorsArray removeObject:meteor];
}

- (void)shootWillBeRemoved:(Shoot *)shoot
{
    // Após explodir, um Tiro notifica a GameScene
    // para que seja removido do Array
    shoot.delegate = nil;
    [self.shootsArray removeObject:shoot];
}
```

7.3 PLAYER MORRE

Utilizando a mesma estratégia vamos animar o player quando um meteoro colidir com ele. Para isso, na classe `Player` teremos o método `explode` como abaixo. Declare-o na `Player.h`:

```
- (void)explode;
```

E o implemente na `Player.m`:

```
- (void)explode
{
```

```

// Para o agendamento
[self unscheduleUpdate];

// Cria efeitos
float dt = 0.2f;
CCScaleBy *a1 = [CCScaleBy initWithDuration:dt scale:2.0f];
CCFadeOut *a2 = [CCFadeOut initWithDuration:dt];
CCSpawn *s1 = [CCSpawn initWithArray:
               [NSArray arrayWithObjects:a1, a2, nil]];

// Executa efeito
[self runAction:s1];
}

```

Para que seja executado, crie o método `playerHit:withMeteor:` na `GameScene.m`. Repare que este é o método de *callback* que definimos quando chegamos as colisões entre os *arrays* de *players* e *meteoros*.

```

- (void)playerHit:(id)player withMeteor:(id)meteor
{
    // Quando houve uma colisão entre Player e Meteoro,
    // indica que ambos foram atingidos
    if ([player isKindOfClass:[Player class]]) {
        [(Player *)player explode];
    }
    if ([meteor isKindOfClass:[Meteor class]]) {
        [(Meteor *)meteor gotShot];
    }
}

```

Rode o projeto e verifique a tela do jogo!

7.4 PLACAR

Para fechar esse capítulo, vamos adicionar uma pontuação para cada meteoro destruído após ser atingido por um tiro. Utilizaremos alguns conceitos já vistos anteriormente.

Primeiramente, trataremos o valor de pontos que aparece na tela como uma nova camada, e camadas para o Cocos2D são classes que herdam de `CCLayer`.

Essa camada apresentará os pontos em número para o jogador. Para isso, precisamos de duas variáveis. A primeira é do tipo inteiro, que guarda os pontos atuais. A segunda é um campo de texto para colocar esse valor na tela.

Além disso, veremos como utilizar um tipo de letra (*font type*) diferente para fazer isso. Existe no Cocos2D uma classe chamada `CCLabelBMFont`. Essa classe possui um método chamado `labelWithString:fntFile:` que recebe uma *string* e uma fonte a ser usada. Depois disso, basta configurar o tamanho da letra e posicionamento.

Alteraremos o valor do `score` com um método chamado `increase`. Esse método poderá ser chamado sempre que uma colisão entre tiro e meteoro for detectada.

Crie a classe `Score` e defina as propriedades e métodos em seu *header* `Score.h`.

```
@interface Score : CCLayer

@property (nonatomic, assign) int score;
@property (nonatomic, retain) CCLabelBMFont *text;

+ (Score *)score;

- (void)increase;

@end
```

No `init` da `Score.m` crie o *label* de texto:

```
+ (Score *)score
{
    return [[[Score alloc] init] autorelease];
}

- (id)init
{
    self = [super init];
    if (self) {
        // Inicializa a pontuação com o valor "0"
        self.score = 0;

        // Posiciona o Placar recém criado
        self.position =
            ccp(SCREEN_WIDTH() - 50.0f, SCREEN_HEIGHT() - 50.0f);
    }
}
```

```

        // Adiciona o Player na tela, como um texto para o jogador
        self.text = [CCLabelBMFont labelWithString:
                    [NSString stringWithFormat:@"%d", self.score]
                    fntFile:@"UniSansSemiBold_Numbers_240.fnt"];
        self.text.scale = (float)(240.0f / 240.0f);
        [self addChild:self.text];
    }
    return self;
}

```

Implemente também o método `increase`. Seu código é simples, apenas incrementa a variável `score` e configura novamente o texto do placar.

```

- (void)increase
{
    // Aumenta a pontuação e atualiza o Placar
    self.score++;
    self.text.string = [NSString stringWithFormat:@"%d", self.score];
}

```

Agora que temos a camada do placar preparada, vamos adicioná-la à tela principal. Para isso, criaremos dois objetos: um do tipo `CCLayer` e outro do tipo `Score`.

Na `GameScene.h` adicione:

```

#import "Score.h"
//...
@property (nonatomic, retain) CCLayer *scoreLayer;
@property (nonatomic, retain) Score *score;

```

É necessário iniciar a camada e adicioná-la a tela. No `init` da `GameScene.m` adicione:

```

- (id)init
{
    self = [super init];
    if (self) {
        //...
        // CCLayer para o Placar
        self.scoreLayer = [CCLayer node];
        [self addChild:self.scoreLayer];
    }
}

```

```

    return self;
}

```

Para finalizar, basta criar o objeto do tipo `Score` e adicionar a camada correspondente.

Ainda na classe `GameScene` adicione essa chamada ao método `addGameObjects`:

```

- (void)addGameObjects
{
    //...
    // Cria o Placar
    self.score = [Score score];
    [self.scoreLayer addChild:self.score];
}

```

Agora altere o método `meteorHit:withShoot:` para que aumente o score quando houver colisão entre um tiro e um meteoro:

```

- (void)meteorHit:(id)meteor withShoot:(id)shoot
{
    //...
    // Aumenta a pontuação
    [self.score increase];
}

```

7.5 CONCLUSÃO

Detectar colisões de forma manual, como feito no capítulo do protótipo, não é tão simples e envolve muitos cálculos matemáticos. Porém, utilizando um *framework* como o Cocos2D as coisas são facilitadas.

Nesse capítulo passamos pelo que pode ser considerado o coração do jogo, a detecção de colisões. A partir delas, executamos efeitos e atualizamos a tela para o jogador.

O próximo capítulo tratará de uma parte muito importante para dar vida aos jogos, os sons e efeitos.

CAPÍTULO 8

Adicionando sons e música

Os sons são de fundamental importância no desenvolvimento de um game. Hoje existem profissões como *sound designers* que trabalham especificamente criando os sons dos games. Muitos jogos utilizam orquestras para executar sua trilha sonora.

A música dá vida ao jogo, torna-o mais divertido e dá respostas ao jogador para as partes importantes.

Existem duas formas principais de sons no mundo dos games: música e efeitos.

Quando o jogo começa, uma música de fundo normalmente dá o clima do jogo. Essa música é normalmente executada em *background* e se repete inúmeras vezes ao longo do game. Além dela, existem os efeitos de som gerados em momentos importantes, como quando uma colisão é detectada ou quando o placar é alterado

Para o nosso jogo, utilizaremos sons encontrados gratuitamente no site <http://www.freesound.org/>. Você pode buscar diversos tipos de sons nesse site para o seu próximo game!

8.1 EXECUTANDO SONS

Nessa primeira etapa utilizaremos o *framework* Cocos2D para adicionar som a 3 eventos do jogo. Utilizaremos 3 arquivos de sons diferentes:

- Disparo de um tiro
- Colisão do tiro com um meteoro
- Colisão entre meteoro e avião

Podemos utilizar os formatos mais comuns para adicionar sons ao nosso jogo, e aqui o formato escolhido será `wav`. Colocaremos os sons no diretório `Resources/Sounds` do nosso projeto.

Você pode encontrar todos os sons que serão utilizados nesse capítulo nesse *link*: https://github.com/bivissoft/jogos_ios_14bis

SoundEngine

Para lidar com sons, o Cocos2D disponibiliza uma classe chamada `SimpleAudioEngine`, que possui diversos métodos que possibilitam trabalhar com sons e música no game. Para utilizar essa classe não é necessário criar uma instância, mas sim utilizar um `Singleton` disponibilizado pelo *framework*. Para isso executamos `[SimpleAudioEngine sharedEngine]` tendo acesso às opções de sons de que precisamos.

Com esse acesso, podemos executar músicas e sons, parar e iniciar arquivos de áudio, aumentar e diminuir o volume etc. Nesse momento, iniciaremos executando 3 sons utilizando o método `playEffect:`. Esse método recebe como parâmetro o nome do arquivo de áudio.

Para facilitar, importaremos a classe `SimpleAudioEngine` no arquivo `Prefix.pch`:

```
#ifdef __OBJC__  
//...  
#import "SimpleAudioEngine.h"  
#endif
```

O primeiro efeito de som, o tiro, será colocado na classe `Shoot`. Adicione o código abaixo ao método `start` da `Shoot.m`:

```
- (void)start
{
    //...
    // Som do Tiro
    [[SimpleAudioEngine sharedEngine] playEffect:@"shoot.wav"];
}
```

O próximo som será executado quando ocorrer a colisão entre meteoro e tiro. No arquivo `Meteor.m` adicione o código abaixo no início do método `gotShot`:

```
- (void)gotShot
{
    // Som ao ser atingido
    [[SimpleAudioEngine sharedEngine] playEffect:@"bang.wav"];
    //...
}
```

Para finalizar, adicionaremos um som quando um meteoro atingir o avião. No arquivo `Player.m` adicione o código abaixo no início do método `explode`:

```
- (void)explode
{
    // Som ao explodir
    [[SimpleAudioEngine sharedEngine] playEffect:@"over.wav"];
    //...
}
```

Agora o game já executará os sons quando um dos 3 eventos acima (tiro e colisões) acontecerem!

8.2 CACHE DE SONS

Vimos como executar sons utilizando o Cocos2D. Porém, você deve ter reparado em um problema grave: a primeira vez que um dos sons precisa ser tocado, ele demora muito. Isso se deve ao fato do *framework* inicializar o som apenas no momento que foi necessário. Existe uma estratégia e boa prática para solucionar esse problema, o cache de sons.

Para colocar um som no cache devemos iniciá-lo já no início do game. Criaremos então um método com essa responsabilidade na `GameScene.m` chamado `preloadCache`. Esse método fará o cache dos 3 sons que estamos utilizando até aqui.

```
- (void)preloadCache
{
    // Cache de sons do Jogo
    [[SimpleAudioEngine sharedEngine] preloadEffect:@"shoot.wav"];
    [[SimpleAudioEngine sharedEngine] preloadEffect:@"bang.wav"];
    [[SimpleAudioEngine sharedEngine] preloadEffect:@"over.wav"];
}
```

Adicione a chamada do método `preloadCache` no construtor da `GameScene.m`, na última linha do método:

```
- (id)init
{
    self = [super init];
    if (self) {
        //...
        // Cache de sons do Jogo
        [self preloadCache];
    }
    return self;
}
```

Rode novamente o projeto e verifique que agora os sons respondem perfeitamente ao momento dos tiros e colisões!

8.3 MÚSICA DE FUNDO

Agora que já trabalhamos com os sons do game, vamos ver como lidar com a música do jogo. A música será iniciada quando o jogador entrar na tela de jogo. Para isso, usaremos a mesma classe `SimpleAudioEngine` porém com o método `playBackgroundMusic:loop:`.

No construtor da `GameScene.m` adicione a chamada a música.

```
- (id)init
{
    self = [super init];
    if (self) {
        //...
        // Música do Jogo
        [[SimpleAudioEngine sharedEngine]
         playBackgroundMusic:@"music.wav" loop:YES];
    }
}
```

```
    }  
    return self;  
}
```

Outro método importante é o que para a música. Esse método é chamado `pauseBackgroundMusic`. Faremos isso logo após o efeito de explosão entre meteoro e avião na classe `Player.m`:

```
- (void)explode  
{  
    //...  
    // Pausa a música de fundo  
    [[SimpleAudioEngine sharedEngine] pauseBackgroundMusic];  
}
```

Rode o projeto e veja como o som adiciona vida ao jogo. Tiro, colisões e música devem estar funcionando nesse momento!

8.4 CONCLUSÃO

Sons e música são muito importantes para os jogos. Repare nos jogos famosos ou mesmo os que você mais gosta e repare na trilha sonora. Mesmo os jogos antigos possuíam sons muito especiais para cada jogo.

É importantíssimo definir uma boa trilha sonora e efeitos para o seu próximo game!

CAPÍTULO 9

Voando com a gravidade!

O mundo dos games foi totalmente revitalizado com os `smartphones`. Jogos que até então eram simples e já não atraíam mais tanto a atenção dos jogadores foram remodelados com as novas possibilidades de experiência que os aparelhos modernos podem proporcionar.

A maioria dos jogos que fazem sucesso nos celulares hoje faz uso de algum recurso do aparelho, como `touch screen`, arrastando objetos na tela ou utilizando a gravidade com o acelerômetro, e movimentando os elementos de uma maneira diferente dos jogos para consoles.

Nesse capítulo trocaremos a movimentação do avião feita por botões pelo controle baseado na movimentação do celular! A experiência será levada a outro nível, não mais sendo um simples jogo com botões, mas sim, utilizando um recurso nativo do aparelho que faz com que o game tenha uma melhor jogabilidade!

Para este capítulo, para poder rodar o jogo no seu próprio aparelho e testar o acelerômetro, você precisará ser um “*Desenvolvedor Apple Registrado*”. Ao fim do capítulo, nosso avião poderá percorrer a tela, ficando como na figura a seguir:

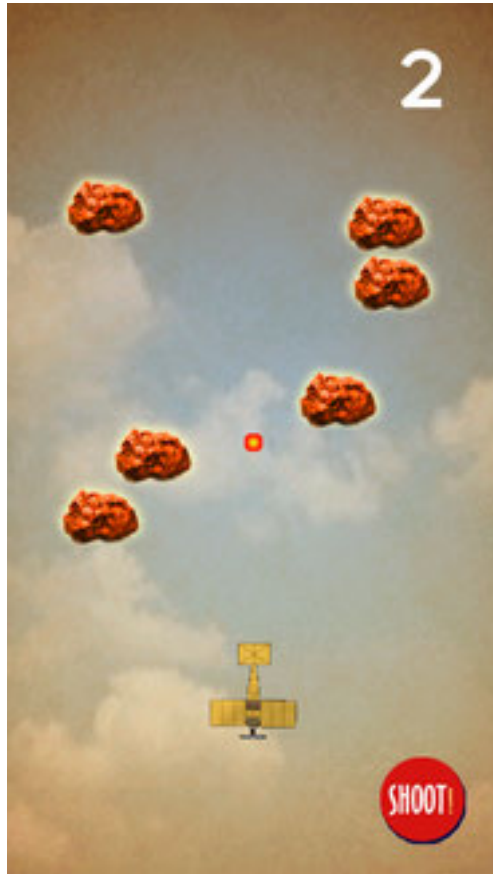


Figura 9.1: Controle por acelerômetro.

9.1 USANDO O ACELERÔMETRO

Vamos iniciar planejando o uso do acelerômetro e sabendo onde queremos chegar. Nosso objetivo é retirar os botões que movimentam o avião para esquerda e direita. Sem esses botões, moveremos o avião a partir das coordenadas que o celular captar de movimento no aparelho. Dividiremos esse trabalho em 3 partes principais.

- Capturar as coordenadas de movimentação horizontal e vertical do aparelho
- Controlar a instabilidade do movimento do avião
- Calibrar essas coordenadas para o controle funcionar em posições diferentes

Capturando as coordenadas

A primeira coisa que precisamos saber é como o `iOS` pode nos enviar informações do acelerômetro. Isso é feito através da classe `CMMotionManager` do *framework* `CoreMotion`, próprio do `iOS`. O que ocorre quando usamos esta classe?

Sempre que iniciamos a `CMMotionManager`, ela começa a colher as informações do acelerômetro do aparelho em um intervalo de tempo que definirmos. Assim, a cada movimentação do dispositivo, as informações de aceleração são atualizadas no método `startAccelerometerUpdatesToQueue:withHandler:`, onde o bloco do *handler* possui dois objetos, sendo um deles um `CMAccelerometerData`, que em nosso código chamaremos de `accelerometerData`.

Essa é uma parte muito importante, e você deve reparar que a cada chamada deste bloco pelo `CMMotionManager`, o objeto `accelerometerData` é atualizado, com os novos valores de posição do aparelho. Esses valores serão informados em 3 variáveis, que representam os eixos X, Y e Z do aparelho, como demonstrado na figura a seguir.

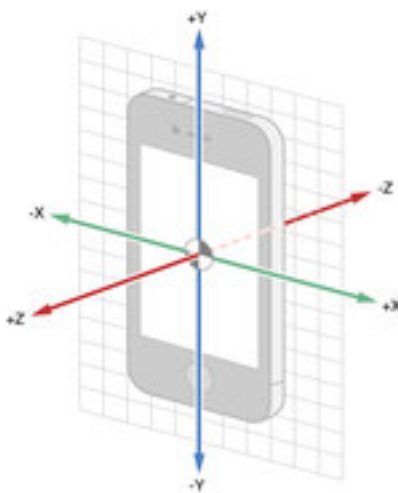


Figura 9.2: Eixos X, Y e Z.

De posse dessas informações, conseguiremos saber se o aparelho se moveu para um determinado lado e atualizar a posição do avião. Essa é uma conclusão importante, ou seja, quando percebemos que o celular está inclinado para um determinado lado, atualizamos essa posição do avião.

A classe Accelerometer

Vamos criar uma classe responsável por instanciar um `CMMotionManager` e capturar seus valores. Mais à frente, veremos como fazer o link entre ela e a movimentação do avião.

Para começar, precisamos importar o *framework* `CoreMotion` para nosso projeto. Para importar um *framework*, no menu à esquerda selecione seu projeto e depois clique no *target* bis. Na aba `Build Phases`, em `Link Binary With Libraries` clique no botão de adicionar e escolha o `CoreMotion.framework`. Pronto! temos o *framework* importado em nosso projeto.

Agora criaremos a classe `Accelerometer` bem simples, como subclasse da `NSObject`. Nesta classe, vamos instanciar o `CMMotionManager` e criar métodos para iniciar e parar o mesmo. Também vamos ver como capturar as posições X e Y do aparelho. Para isso, criaremos 2 propriedades que vão guardar as informações do acelerômetro. O *header* `Accelerometer.h` ficará assim:

```
#import <CoreMotion/CoreMotion.h>

@interface Accelerometer : NSObject

@property (nonatomic, retain) CMMotionManager *motionManager;

- (void)startAccelerometerUpdates;
- (void)stopAccelerometerUpdates;

@property (nonatomic, assign) float currentAccelerationX;
@property (nonatomic, assign) float currentAccelerationY;

@end
```

Na `Accelerometer.m` vamos instanciar a `CMMotionManager` no `init` e criar os métodos que retornam as acelerações X e Y:

```
@implementation Accelerometer

- (id)init
{
    self = [super init];
    if (self) {
        self.motionManager =
```

```
        [[[CMMotionManager alloc] init] autorelease];
        self.motionManager.accelerometerUpdateInterval = 0.01f; // 100Hz
    }
    return self;
}

- (void)startAccelerometerUpdates
{
    // Verifica se o dispositivo possui acesso ao Acelerômetro e o inicia
    if ([self.motionManager isAccelerometerAvailable]) {
        // Zera as propriedades
        self.currentAccelerationX = 0.0f;
        self.currentAccelerationY = 0.0f;

        // Inicia atualizações do acelerômetro
        [self.motionManager
         startAccelerometerUpdatesToQueue:[NSOperationQueue mainQueue]
         withHandler:^(CMAccelerometerData *accelerometerData,
                      NSError *error)
        {
            self.currentAccelerationX = accelerometerData.acceleration.x;
            self.currentAccelerationY = accelerometerData.acceleration.y;
        }]];
    }
}

- (void)stopAccelerometerUpdates
{
    // Quando os dados do Acelerômetro não são mais necessários,
    // deve-se parar as atualizações
    if ([self.motionManager isAccelerometerActive]) {
        [self.motionManager stopAccelerometerUpdates];
    }
}

@end
```

O que temos até aqui? Uma classe que recebe informações de movimentação do aparelho. Ela não foi inicializada ainda, mas logo mais poderá ser utilizada para atualizar a posição da nave.

Para continuar, faremos que essa classe se comporte como um `Singleton`, tendo apenas um objeto do tipo `Accelerometer` no jogo. Para isso, no arquivo `Accelerometer.h` declare um método estático que retornará a instância única da classe:

```
+ (Accelerometer *)sharedAccelerometer;
```

No arquivo `Accelerometer.m` adicione o seguinte código:

```
static Accelerometer *sharedAccelerometer = nil;
+ (Accelerometer *)sharedAccelerometer
{
    if (!sharedAccelerometer) {
        sharedAccelerometer = [[Accelerometer alloc] init];
    }
    return sharedAccelerometer;
}
```

Dessa forma, não corremos riscos de mais de um objeto desse tipo ser instanciado no jogo. Agora, já podemos pensar em fazer o *link* com a classe `Player`.

Configurando o player

Toda vez que o acelerômetro atualizar as posições devemos mover o avião. Isso ocorrerá em uma quantidade muito grande de vezes por segundo, dando a impressão de movimento. Como em outras classes, iremos agendar o método `update` para que o player busque as acelerações X e Y da classe `Accelerometer` e atualize sua posição.

Criaremos o método `monitorAccelerometer` para que o player inicie a classe de acelerômetro e agende seu próprio `update`. Defina o método na `Player.h`:

```
#import "Accelerometer.h"
//...
- (void)monitorAccelerometer;
```

Na `Player.m` implemente este método e o `update`:

```
- (void)monitorAccelerometer
{
    // Inicia a Atualização do Acelerômetro
    [[Accelerometer sharedAccelerometer] startAccelerometerUpdates];
```

```
// Inicia a Animação / Movimentação do Player
[self scheduleUpdate];
}

- (void)update:(float)dt
{
    NSLog(@"X: %f", [[Accelerometer sharedAccelerometer]
                    currentAccelerationX]);
    NSLog(@"Y: %f", [[Accelerometer sharedAccelerometer]
                    currentAccelerationY]);
}
```

Agora vamos alterar o método `startGame` da `GameScene.m` para que peça ao player para monitorar o acelerômetro quando o jogo for iniciado:

```
- (void)startGame
{
    //...
    // Ao entrar na GameScene, inicia o monitoramento do Acelerômetro
    [self.player monitorAccelerometer];
}
```

Você pode executar o projeto agora e olhar no console do Xcode as coordenadas sendo impressas.

O que temos neste momento? A classe `Accelerometer` que recebe coordenadas do aparelho e a classe `Player` preparada para ler esses valores.

Movendo o player

Agora que já estamos lendo as coordenadas do acelerômetro na classe `Player` já podemos movimentá-lo. O método `update` é quem altera a posição do avião, então, vamos alterá-lo.

A ideia aqui será mover uma das quatro possíveis coordenadas do avião, seja horizontal (direita ou esquerda) ou vertical (cima ou baixo). As coordenadas enviadas pelo acelerômetro seguem o padrão dos eixos, fazendo com que movimentações para um lado sejam números positivos e para o lado oposto números negativos. O mesmo ocorre para o eixo X, enviando números positivos para uma direção e negativos para a posição oposta.

Com base nessa informação, analisaremos as coordenadas que o acelerômetro enviou e alteraremos a posição do avião.

Na classe `Player` altere o método `update::`

```
- (void)update:(float)dt
{
    if ([[Accelerometer sharedAccelerometer]
        currentAccelerationX] < -0.0f) {
        self.positionX--;
    }
    if ([[Accelerometer sharedAccelerometer]
        currentAccelerationX] > 0.0f) {
        self.positionX++;
    }
    if ([[Accelerometer sharedAccelerometer]
        currentAccelerationY] < -0.0f) {
        self.positionY--;
    }
    if ([[Accelerometer sharedAccelerometer]
        currentAccelerationY] > 0.0f) {
        self.positionY++;
    }

    // Checa limites da tela
    if (self.positionX < 30.0f) {
        self.positionX = 30.0f;
    }
    if (self.positionX > SCREEN_WIDTH() - 30.0f) {
        self.positionX = SCREEN_WIDTH() - 30.0f;
    }
    if (self.positionY < 30.0f) {
        self.positionY = 30.0f;
    }
    if (self.positionY > SCREEN_HEIGHT() - 30.0f) {
        self.positionY = SCREEN_HEIGHT() - 30.0f;
    }

    // Configura posição do Avião
    self.position = ccp(self.positionX, self.positionY);
}
```

Já é possível rodar o jogo e ver o avião se movendo a partir da movimentação do aparelho! Faça o teste.

Experimente deixar a tela paralela a uma mesa. Algo estranho, não? Parece que ele está enfrentando uma certa “turbulência”.

9.2 CONTROLANDO A INSTABILIDADE

Você deve ter percebido que o controle do avião ficou instável. Isso ocorre porque não estamos usando nenhuma tolerância para mover o avião, ou seja, movemos sempre independente de ser uma movimentação realmente válida do aparelho. O acelerômetro não é perfeito, além de que ele pega movimentações minúsculas, sempre gerando eventos!

Para melhorar isso, vamos usar um limite. Em vez de comparar com zero, utilizaremos uma constante de tolerância. Chamaremos essa constante de `kNOISE`. Ela definirá o valor mínimo que o acelerômetro deve ser alterado para realmente mover o avião.

Crie a constante e altere o método `update:` na `Player.m`.

```
#define kNOISE 0.15f
//...
- (void)update:(float)dt
{
    if ([[Accelerometer sharedAccelerometer]
        currentAccelerationX] < -kNOISE) {
        self.positionX--;
    }
    if ([[Accelerometer sharedAccelerometer]
        currentAccelerationX] > kNOISE) {
        self.positionX++;
    }
    if ([[Accelerometer sharedAccelerometer]
        currentAccelerationY] < -kNOISE) {
        self.positionY--;
    }
    if ([[Accelerometer sharedAccelerometer]
        currentAccelerationY] > kNOISE) {
        self.positionY++;
    }
    //...
}
```

A partir deste momento, devemos ter um bom controle do avião, podendo movimentá-lo por toda a tela. O único inconveniente é que não temos uma calibração para utilizar o acelerômetro, ou seja, ele funciona bem para a posição inicial zero, que é a aquela quando deixamos o aparelho parado em uma mesa, por exemplo.

9.3 CALIBRANDO A PARTIR DA POSIÇÃO INICIAL DO APARELHO

Vamos utilizar uma estratégia de calibração no jogo! A ideia é não se basear apenas na posição enviada pelo acelerômetro para mover o player, mas fazer antes algumas contas para entender a posição que o jogador está segurando o aparelho e considerá-la como a posição ZERO, ou posição inicial.

Para isso faremos algumas alterações na classe `Accelerometer`. Criaremos novas propriedades que serão responsáveis por guardar informações sobre a calibração. Essas propriedades guardarão as informações iniciais e se já temos a calibração concluída.

Na `Accelerometer.h` crie as seguintes propriedades:

```
@property (nonatomic, assign) int calibrated;
@property (nonatomic, assign) float calibratedAccelerationX;
@property (nonatomic, assign) float calibratedAccelerationY;
```

Alteraremos também o método `startAccelerometerUpdates` na `Accelerometer.m`. Nele, calibraremos o acelerômetro recebendo as 30 primeiras informações do acelerômetro. Com esses 30 primeiros valores guardaremos a posição inicial do aparelho.

A partir disso, faremos uma alteração no valor que é enviado para mover o avião. Ao invés de enviar o valor diretamente informado pelo acelerômetro, vamos tirar a posição inicial, para ter apenas a mudança relativa àquela movimentação.

```
#define kCALIBRATIONCOUNT 30
```

```
- (void)startAccelerometerUpdates
{
    // Verifica se o dispositivo possui acesso ao Acelerômetro e o inicia
    if ([self.motionManager isAccelerometerAvailable]) {
        // Zera as propriedades
        self.currentAccelerationX = 0.0f;
```

```
self.currentAccelerationY = 0.0f;
self.calibrated = 0;
self.calibratedAccelerationX = 0.0f;
self.calibratedAccelerationY = 0.0f;

// Inicia atualizações do acelerômetro
[self.motionManager
    startAccelerometerUpdatesToQueue:[NSOperationQueue mainQueue]
    withHandler:^(CMAccelerometerData *accelerometerData,
                  NSError *error)
{
    if (self.calibrated < kCALIBRATIONCOUNT) {
        // Soma posições do acelerômetro
        self.calibratedAccelerationX +=
            accelerometerData.acceleration.x;
        self.calibratedAccelerationY +=
            accelerometerData.acceleration.y;

        self.calibrated++;
        if (self.calibrated == kCALIBRATIONCOUNT) {
            // Calcula a média das calibrações
            self.calibratedAccelerationX /= kCALIBRATIONCOUNT;
            self.calibratedAccelerationY /= kCALIBRATIONCOUNT;
        }
    } else {
        // Atualiza aceleração atual
        self.currentAccelerationX =
            accelerometerData.acceleration.x -
            self.calibratedAccelerationX;
        self.currentAccelerationY =
            accelerometerData.acceleration.y -
            self.calibratedAccelerationY;
    }
}];
}
```

Por que 30 vezes? O jogador acabou de apertar o botão play e está ajustando sua melhor posição. Então vamos dar uma tolerância para ler 30 atualizações do acelerômetro e tirar uma média delas. Com isso, conseguiremos definir qual a posição inicial do aparelho.

Você pode manter os `logs` por um tempo para entender os valores enviados e depois apagá-los. Nesse momento o avião já deve estar sendo controlado pela movimentação do aparelho!

Retirando os botões

Provavelmente você não vai querer mover mais o avião utilizando os botões esquerda e direita. Uma forma simples de removê-los é não incluí-los no menu de botões, alterando as linhas a seguir, no método `init` da `GameScene.m`:

```
// CCMenu *menu = [CCMenu menuWithItems:leftControl,
//                                     rightControl,
//                                     shootButton,
//                                     nil];
CCMenu *menu = [CCMenu menuWithItems:shootButton,
                                     nil];
```

9.4 DESAFIOS COM O ACELERÔMETRO

Utilizar recursos como acelerômetro pode tornar o jogo muito mais divertido e engajador, sendo um daqueles detalhes que fazem a experiência do game ser totalmente única. Que tal melhorar ainda mais essa experiência com as sugestões abaixo?

- Controle de velocidade: Você pode fazer que, quanto mais inclinado, o avião deslize mais. Se ele estiver pouco inclinado, em vez de incrementar a variável `x` em 1, você incrementaria em 0.5 ou algo proporcional à aceleração indicada pelo acelerômetro.
- O acelerômetro pode mandar sinais invertidos de acordo com a inclinação. Dependendo da calibração, você precisa detectar se isso está ocorrendo, para não mudar a orientação do avião repentinamente! Isso dá um certo trabalho...

9.5 CONCLUSÃO

Esse é um capítulo trabalhoso, mas muito gratificante. Utilizar recursos dos aparelhos é um dos principais apelos da revolução dos jogos para celular. Saber trabalhar bem com esses poderosos recursos pode elevar o jogo a um nível de jogabilidade muito mais interessante!

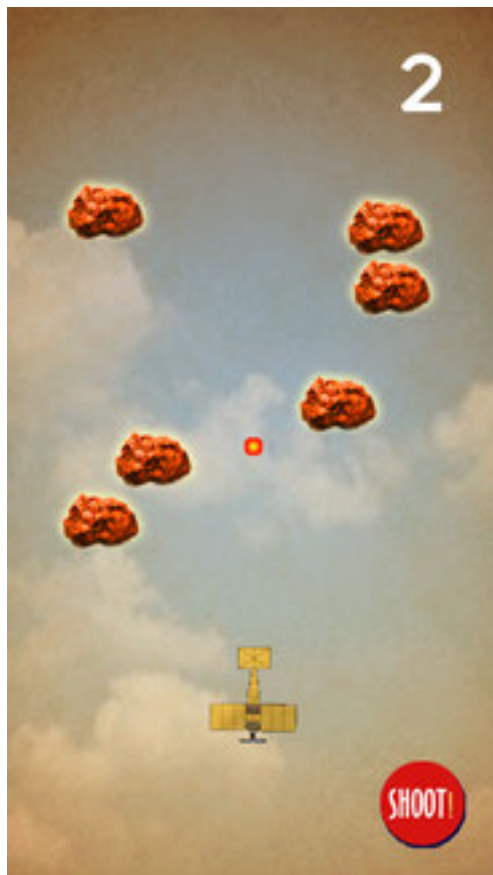


Figura 9.3: Controle por acelerômetro.

CAPÍTULO 10

Tela final e game over

Todo o fluxo do jogo está bem encaminhado, desde a tela de abertura, passando pelas colisões, efeitos, sons e acelerômetro. Agora criaremos duas telas que fecharão o ciclo principal do game.

A primeira tela a ser criada será a tela mostrada quando o jogo acabar, ou seja, quando o 14bis vencer os 100 meteoros.

Essa tela será composta por uma música final, por uma imagem nova e um botão de início do jogo.

Ao final, deveremos ter a tela como abaixo.



Figura 10.1: Tela do final do jogo.

10.1 TELA FINAL

Para montar a tela de final de jogo precisaremos de mais uma imagem, a `finalend.png` (e sua versão HD `finalend-hd.png`). Importe essas imagens para o projeto e inclua a definição padrão na `Assets.h`.

```
#define kFINALEND      @"finalend.png"
```

Vamos criar a classe que representará a tela final. Ela será uma nova `CCLayer` e seu nome será `FinalScreen`.

Como toda scene, declare no `FinalScreen.h` o construtor:

```
@interface FinalScreen : CCLayer

+ (CCScene *)scene;

@end
```

Na `FinalScreen.m` implementaremos o método de criação da `scene`. Aproveitaremos o `init` para inicializar os objetos *background*, som e imagem de logo. Faremos isso como já fizemos em outras telas:

```
- (id)init
{
    self = [super init];
    if (self) {
        // Imagem de Background
        CCSprite *background = [CCSprite spriteWithFile:kBACKGROUND];
        background.position =
            ccp(SCREEN_WIDTH() / 2.0f, SCREEN_HEIGHT() / 2.0f);
        [self addChild:background];

        // Som
        [[SimpleAudioEngine sharedEngine] playEffect:@"finalend.wav"];

        // Imagem
        CCSprite *title = [CCSprite spriteWithFile:kFINALEND];
        title.position =
            ccp(SCREEN_WIDTH() / 2.0f, SCREEN_HEIGHT() - 130.0f);
        [self addChild:title];
    }
    return self;
}
```

Configuraremos, ainda no `init`, o botão que inicia o jogo novamente:

```
//...
// Cria o botão para reiniciar o jogo
CCMenuItemSprite *beginButton = [CCMenuItemSprite
    itemWithNormalSprite:[CCSprite spriteWithFile:kPLAY]
    selectedSprite:[CCSprite spriteWithFile:kPLAY]
    target:self
    selector:@selector(restartGame:)];
```

```
// Define a posição do botão
beginButton.position = ccp(0.0f, 0.0f);

// Cria o menu que terá o botão
CCMenu *menu = [CCMenu menuWithItems:beginButton, nil];
[self addChild:menu];
```

Quando o jogador selecionar o botão de reiniciar o jogo, ele retornará à tela inicial `TitleScreen`. Vamos importar a tela inicial na `FinalScreen.h`:

```
#import "TitleScreen.h"
```

Voltando à `FinalScreen.m`, o botão chamará o método `restartGame`: quando selecionado, para que retorne à tela inicial de nosso game. Vamos implementar este método.

```
- (void)restartGame:(id)sender
{
    // Pausa a música de fundo
    [[SimpleAudioEngine sharedEngine] pauseBackgroundMusic];

    // Transfere o Jogador para a TitleScreen
    [[CCDirector sharedDirector] replaceScene:
        [CCTransitionFade transitionWithDuration:1.0
        scene:[TitleScreen scene]]];
}
```

Para que essa classe possa ser vista no jogo, a classe `GameScene` deve estar ciente e inicializá-la. Para isso, vamos importar a `FinalScreen` em nossa `GameScene.h`

```
#import "FinalScreen.h"
```

Agora na `GameScene.m` teremos o método `startFinalScreen` que faz a transição para a tela final.

```
- (void)startFinalScreen
{
    // Transfere o Jogador para a FinalScreen
    [[CCDirector sharedDirector] replaceScene:
        [CCTransitionFade transitionWithDuration:1.0
        scene:[FinalScreen scene]]];
}
```

E quando teremos a tela de final do jogo? Faremos isso quando 100 meteoros forem destruídos! Porém, para facilitar os testes, mostraremos a tela final quando 5 meteoros forem destruídos.

Ainda na `GameScene.m`, no método `meteorHit:withShoot:` vamos incluir a chamada da tela final após aumentar o score, caso este seja maior ou igual a 5:

```
- (void)meteorHit:(id)meteor withShoot:(id)shoot
{
    //...
    // Verifica o máximo score para vencer o jogo
    if (self.score.score >= 5) {
        [self startFinalScreen];
    }
}
```

Ao destruir 5 meteoros já devemos ter a tela final com o som da vitória!



Figura 10.2: Tela do final do jogo.

10.2 TELA GAME OVER

A tela de game over seguirá a mesma lógica da tela final, porém deverá ser inicializada em um outro momento. Quando? Simples, quando um meteoro atingir o avião!

Vamos ao código. Primeiro, importe a imagem `gameover.png` (e sua versão HD `gameover-hd.png`) e inclua a definição padrão na `Assets.h`.

```
#define kGAMEOVER @"gameover.png"
```

Crie a classe `GameOverScreen` que é uma `CCLayer`. Nessa classe, utilizaremos a mesma ideia da tela de final de jogo. A `GameOverScreen.h` ficará assim:

```
#import "TitleScreen.h"

@interface GameOverScreen : CCLayer

+ (CCScene *)scene;

@end
```

O `init` da `GameOverScreen.m` também é bem simples, parecido com o que já conhecemos:

```
+ (CCScene *)scene
{
    // 'scene' is an autorelease object.
    CCScene *scene = [CCScene node];

    // 'layer' is an autorelease object.
    GameOverScreen *layer = [GameOverScreen node];

    // add layer as a child to scene
    [scene addChild:layer];

    // return the scene
    return scene;
}

- (id)init
{
    self = [super init];
    if (self) {
        // Imagem de Background
        CCSprite *background = [CCSprite spriteWithFile:kBACKGROUND];
        background.position =
            ccp(SCREEN_WIDTH() / 2.0f, SCREEN_HEIGHT() / 2.0f);
        [self addChild:background];

        // Som
        [[SimpleAudioEngine sharedEngine] playEffect:@"finalend.wav"];

        // Imagem
        CCSprite *title = [CCSprite spriteWithFile:kGAMEOVER];
        title.position =
```

```

        ccp(SCREEN_WIDTH() / 2.0f, SCREEN_HEIGHT() - 130.0f);
[self addChild:title];

// Cria o botão para reiniciar o jogo
CCMenuItemSprite *beginButton = [CCMenuItemSprite
    initWithNormalSprite:[CCSprite spriteWithFile:kPLAY]
    selectedSprite:[CCSprite spriteWithFile:kPLAY]
    target:self
    selector:@selector(restartGame:)];

// Define a posição do botão
beginButton.position = ccp(0.0f, 0.0f);

// Cria o menu que terá o botão
CCMenu *menu = [CCMenu menuWithItems:beginButton, nil];
[self addChild:menu];
}
return self;
}

```

Para que o botão de reiniciar o jogo funcione, implemente o método `restartGame:`.

```

- (void)restartGame:(id)sender
{
    // Pausa a música de fundo
    [[SimpleAudioEngine sharedEngine] pauseBackgroundMusic];

    // Transfere o Jogador para a TitleScreen
    [[CCDirector sharedDirector] replaceScene:
        [CCTransitionFade transitionWithDuration:1.0
            scene:[TitleScreen scene]]];
}

```

A chamada a essa tela deve ser feita quando a colisão entre meteoro e avião for detectada. Para isso, importe a `GameOverScreen` na `GameScene.h`:

```
#import "GameOverScreen.h"
```

Agora adicione a transição ao método `playerHit:withMeteor:` da `GameScene.m`:

```
- (void)playerHit:(id)player withMeteor:(id)meteor
{
    //...
    // Ao ser atingido, o Jogador é transferido à GameOverScreen
    [[CCDirector sharedDirector] replaceScene:
        [CCTransitionFade transitionWithDuration:1.0
            scene:[GameOverScreen scene]]];
}
```

A tela de game over deve estar aparecendo quando o meteoro colide com o avião, como mostrado a seguir.



Figura 10.3: Tela de game over.

10.3 CONCLUSÃO

Esse é um capítulo simples, pois já conhecemos tudo que é necessário para criação de telas e transições. Você pode usar sua imaginação e criar diversas novas telas no game!

CAPÍTULO 11

Pausando o jogo

Nesse capítulo falaremos de mais uma parte importantíssima de um jogo, a tela de pause. Essa tela não costuma ser das mais divertidas de ser desenvolvida, até mesmo pela falsa impressão de que pode ser uma tela simples. Porém, tenha atenção aqui! Teremos muitos conceitos importantes nesse momento.

Para não se enganar, vamos à lista de funcionalidades que uma tela de pause deve ter.

- Construir uma nova camada para a tela de pause
- Criar uma classe que entenderá se o jogo está em pause ou rodando
- Criar mais um botão na tela de jogo, o botão pause
- Fazer o *link* entre a tela de pause e tela de jogo
- Parar realmente os objetos na tela

Veja como a tela de pause pode enganar. São muitas coisas a serem feitas para que tudo funcione bem.

Repare que, sempre que possível, fazer uma lista de funcionalidades esperadas na tela pode ajudar a ver com mais profundidade o trabalho que será necessário desenvolver.

Ao final desse capítulo a tela deverá estar como abaixo:

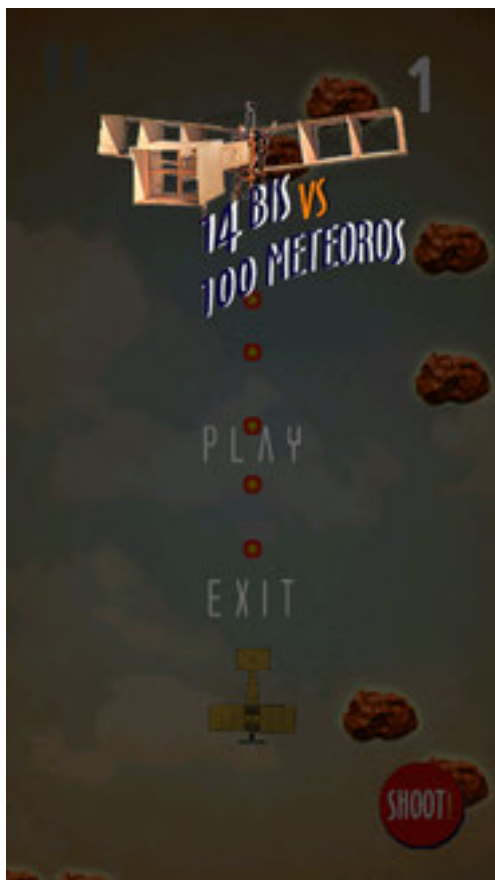


Figura 11.1: Tela de pause.

11.1 MONTANDO A TELA DE PAUSE

Vamos começar de forma simples e com o que já vimos até o momento. A tela de pause é na verdade mais uma camada dentro de uma tela, ou seja, não mudaremos

de cena (`CCScene`). Como anteriormente, toda nova camada no Cocos2D pode ser representada pela classe `CCLayer`.

Uma parte importante dessa tela é que ela terá 2 botões: o “continuar”, que volta pro jogo, e o “sair”, que vai para a tela de abertura.

Criaremos os dois botões nessa camada com a classe já utilizada em outras partes do game, a `CCMenuItemSprite`.

Crie a classe `PauseScreen`, subclasse da `CCLayer`, com o header abaixo:

```
@interface PauseScreen : CCLayer
```

```
+ (PauseScreen *)pauseScreen;
```

```
@end
```

A próxima etapa é adicionar a seguinte lista nessa tela:

- Definir um `background`, nesse caso um preto translúcido
- Colocar o logo, como na tela de abertura
- Adicionar os botões
- Posicionar os botões

Importe a imagem `exit.png` (e sua versão HD `exit-hd.png`) para o projeto e inclua a definição padrão na `Assets.h`:

```
#define kEXIT @"exit.png"
```

Agora, na `PauseScreen.m`, implemente o método `pauseScreen` e adicione os botões no `init`, como abaixo:

```
+ (PauseScreen *)pauseScreen
{
    return [[[PauseScreen alloc] init] autorelease];
}
```

```
- (id)init
{
    self = [super init];
    if (self) {
```

```

// Cor de Background
CCLayerColor *background =
    [CCLayerColor layerWithColor:ccc4(0, 0, 0, 175)
                      width:SCREEN_WIDTH()
                      height:SCREEN_HEIGHT()];

[self addChild:background];

// Imagem de Logo
CCSprite *title = [CCSprite spriteWithFile:kLOGO];
title.position =
    ccp(SCREEN_WIDTH() / 2.0f, SCREEN_HEIGHT() - 130.0f);
[self addChild:title];

// Cria os botões
CCMenuItemSprite *resumeButton = [CCMenuItemSprite
    itemWithNormalSprite:[CCSprite spriteWithFile:kPLAY]
    selectedSprite:[CCSprite spriteWithFile:kPLAY]
    target:self
    selector:@selector(resumeGame:)];
CCMenuItemSprite *quitButton = [CCMenuItemSprite
    itemWithNormalSprite:[CCSprite spriteWithFile:kEXIT]
    selectedSprite:[CCSprite spriteWithFile:kEXIT]
    target:self
    selector:@selector(quitGame:)];

// Define as posições dos botões
resumeButton.position = ccp(0.0f, 0.0f);
quitButton.position = ccp(0.0f, -100.0f);

// Cria o menu que terá os botões
CCMenu *menu = [CCMenu menuWithItems:resumeButton,
                                           quitButton,
                                           nil];

[self addChild:menu];
}

return self;
}

```

Nesse momento temos a tela de pause, porém essa é só a primeira parte. Repare que criar uma tela de pause não é algo diferente das telas anteriores do jogo, porém, fazer o controle que ela precisa demandará novos conceitos. Vamos a eles.

11.2 CONTROLANDO O GAME LOOP

Lembra da analogia de desenhos em blocos de papel do capítulo sobre protótipos? Falamos que um jogo pode ser comparado a uma sequência de imagens que são desenhadas a cada mudança no posicionamento dos objetos do jogo, gerando a impressão de movimento.

O que isso importa para a tela de pause? Precisamos ter um controle do que o jogo está rodando, ou seja, objetos realmente do jogo e não em telas de abertura, por exemplo. Além disso, precisamos saber se o jogo está pausado ou não naquele momento.

Com uma variável de controle de estado, podemos definir se devemos mostrar a tela de pause, se devemos paralisar os objetos na tela, retomar os movimentos dos objetos etc.

Faremos esse controle com uma classe que terá apenas essa responsabilidade. Crie a classe `Runner`, subclasse da `NSObject`, e defina a variável de controle em seu *header* `Runner.h`:

```
@interface Runner : NSObject

@property (nonatomic, assign, getter = isGamePaused) BOOL gamePaused;

+ (Runner *)sharedRunner;

@end
```

Essa classe terá algumas peculiaridades. Uma delas é que ela só terá uma instância ativa no projeto, para que não ocorram confusões entre os estados. Ou seja, ela será um `Singleton`.

Para isso, defina o método `sharedRunner` na `Runner.h` para que retorne a instância única de nossa classe:

```
+ (Runner *)sharedRunner;
```

Na `Runner.m` implemente este método:

```
static Runner *sharedRunner = nil;
+ (Runner *)sharedRunner
{
    if (!sharedRunner) {
        sharedRunner = [[Runner alloc] init];
    }
}
```

```

    }
    return sharedRunner;
}

```

Sempre que precisarmos fazer a verificação do estado do game, chamaremos o método `sharedRunner` que nos devolverá a referência que está cuidando disso. Caso ainda não tenha sido criada, será criada nesse momento.

Com a classe `Runner` sendo um `Singleton`, podemos a qualquer momento acessar a propriedade `gamePaused` para saber o estado do game.

Essa classe pode parecer simples. Um `Singleton` com uma única propriedade. Porém, sua funcionalidade é de extrema importância no game. Vamos utilizá-la em vários momentos.

Como utilizaremos o `Runner` em diversas classes, importe-o na `Prefix.pch`:

```
#import "Runner.h"
```

11.3 ADICIONANDO O BOTÃO DE PAUSE

A próxima etapa é relativamente simples. Iremos adicionar o botão pause na tela do jogo. Para isso precisamos de um novo arquivo, a imagem do botão pause. Importe-o para o projeto e adicione-o no arquivo `Assets.h`.

```
#define kPAUSE @"pause.png"
```

Adicionaremos esse novo botão no menu da classe `GameScene`. O objetivo é criar uma nova variável do tipo `CCMenuItemSprite`, adicioná-la na camada de botões, configurar o método de *callback* e posicioná-la na tela.

Vamos criar o botão no `init` da `GameScene.m`, junto com os demais:

```

- (id)init
{
    self = [super init];
    if (self) {
        //...

        // Cria os botões
        //...
        CCMenuItemGameButton *pauseButton = [CCMenuItemGameButton
                                              initWithNormalSprite:[CCSprite spriteWithFile:kPAUSE]
                                              selectedSprite:[CCSprite spriteWithFile:kPAUSE]

```

```

        target:self
        selector:@selector(pauseGame:));

    // Define as posições dos botões
    //...
    pauseButton.position = ccp(-120.0f,
                               (SCREEN_HEIGHT() / 2.0f) - 30.0f);

    // Cria o menu que terá os botões
    //...
    CCMenu *menu = [CCMenu menuWithItems:shootButton,
                                           pauseButton,
                                           nil];

    [self.gameButtonsLayer addChild:menu];
}
return self;
}

```

Para fechar, vamos criar o método `pauseGame:`, chamado quando o botão `pause` for tocado na `GameScene.m`:

```

- (void)pauseGame:(id)sender
{
    NSLog(@"Botão selecionado: Pausa");
}

```

11.4 A INTERFACE ENTRE JOGO E PAUSE

De fato uma tela de pause não é tão simples como pode parecer, certo? Recapitulando, nesse capítulo criamos até aqui 3 coisas: a tela de pause, uma classe de controle de estados e adicionamos o botão de pause na tela de jogo.

É sempre bom parar e analisar o que já foi feito em tarefas que são grandes como essas e envolvem diversas classes.

Tendo a classe da tela de pause e os botões, precisamos fazer o jogo entender quando ele deve mostrá-la.

11.5 PAUSANDO O JOGO

Começaremos a fazer o *link* entre tudo que foi visto nesse capítulo agora! A principal ideia é orquestrar tudo que fizemos pela classe `GameScene`, que é a tela do jogo,

receberá o evento de pause e ativará a `PauseScreen`.

Para que a tela de pause possa ser vista, precisaremos adicionar uma nova camada na `GameScene`. Criaremos uma camada do tipo `CCLayer` e um objeto do tipo `PauseScreen`.

Na `GameScene.h` crie as propriedades:

```
#import "PauseScreen.h"
//...
@property (nonatomic, retain) CCLayer *topLayer;
@property (nonatomic, retain) PauseScreen *pauseScreen;
```

No `init` da `GameScene.m` iniciaremos a camada e vamos adicionar a mesma à cena atual.

```
- (id)init
{
    self = [super init];
    if (self) {
        // CCLayer para o exibição da tela de Pause
        self.topLayer = [CCLayer node];
        [self addChild:self.topLayer];
    }
}
```

Feito isso, uma nova camada existe na tela de jogo, mas ainda sem relação que queremos com a tela de pause.

Métodos de pause

Vamos seguir fazendo esse *link*. Faremos a tela de jogo saber que alguns métodos de pause são necessários, nesse caso, *pause*, *resume* e *quit*.

Vamos implementar cada um dos 3 métodos de pause agora. O primeiro será o `pauseGame:`, já criado. O que esse método deve fazer é verificar se o game está rodando, ou seja, se o jogo não está em pause. Caso isso seja verdadeiro, configuramos a variável de pause para `YES`.

Na `GameScene.m` altere o método `pauseGame::`

```
- (void)pauseGame:(id)sender
{
    NSLog(@"Botão selecionado: Pausa");
```

```

    if ([Runner sharedRunner].isGamePaused == NO) {
        [Runner sharedRunner].gamePaused = YES;
    }
}

```

Agora vamos fazer a tela de pause avisar a tela de jogo quando o botão de *resume* ou *quit* foi selecionado. Faremos isto através de *delegate*. A nossa tela de jogo será o *delegate* da tela de pause.

Na `PauseScreen.h` declare o protocolo `PauseScreenDelegate` e a propriedade `delegate`:

```

@protocol PauseScreenDelegate;

@interface PauseScreen : CCLayer

@property (nonatomic, assign) id<PauseScreenDelegate>delegate;

+ (PauseScreen *)pauseScreen;

@end

@protocol PauseScreenDelegate <NSObject>
- (void)pauseScreenWillResumeGame:(PauseScreen *)pauseScreen;
- (void)pauseScreenWillQuitGame:(PauseScreen *)pauseScreen;
@end

```

Vamos informar que nossa `GameScene` implementará os métodos do protocolo da tela de pause, alterando o arquivo `GameScene.h`:

```

@interface GameScene : CCLayer < MeteorsEngineDelegate,
                                PlayerDelegate,
                                MeteorDelegate,
                                ShootDelegate,
                                PauseScreenDelegate>

```

O próximo passo é criar os métodos do protocolo na `GameScene.m`. Primeiro, o método que remove a tela de pause para continuar o jogo. Nele, configuramos o `pause` para `NO`. Adicione o método `pauseScreenWillResumeGame` na `GameScreen.m`:

```
- (void)pauseScreenWillResumeGame:(PauseScreen *)pauseScreen
{
    if ([Runner sharedRunner].isGamePaused == YES) {
        // Continua o jogo
        self.pauseScreen.delegate = nil;
        self.pauseScreen = nil;

        [Runner sharedRunner].gamePaused = NO;
    }
}
```

Para fechar, implementaremos o método `pauseScreenWillQuitGame:`, o mais simples entre os 3. Nele, vamos parar os sons que estamos tocando. Além disso, faremos uma transição para a tela de abertura.

```
- (void)pauseScreenWillQuitGame:(PauseScreen *)pauseScreen
{
    [SimpleAudioEngine sharedEngine].effectsVolume = 0.0f;
    [SimpleAudioEngine sharedEngine].backgroundMusicVolume = 0.0f;

    // Transfere o Jogador para a TitleScreen
    [[CCDirector sharedDirector] replaceScene:
        [CCTransitionFade transitionWithDuration:1.0
        scene:[TitleScreen scene]]];
}
```

Iniciando tudo

Precisamos configurar a variável de controle de estado logo no começo do jogo. No método `startGame` da `GameScreen.m` vamos adicionar a configuração de pause abaixo:

```
- (void)startGame
{
    // Configura o status do jogo
    [Runner sharedRunner].gamePaused = NO;
    //...
}
```

Ajustaremos agora o método responsável por iniciar a tela de pause, que será acionado pelo botão de pause. Para isto, altere o método `pauseGame:` na classe `GameScreen.m`:

```
- (void)pauseGame:(id)sender
{
    NSLog(@"Botão selecionado: Pausa");

    if ([Runner sharedRunner].isGamePaused == NO) {
        [Runner sharedRunner].gamePaused = YES;
    }

    if ([Runner sharedRunner].isGamePaused == YES &&
        self.pauseScreen == nil) {

        self.pauseScreen = [PauseScreen pauseScreen];
        self.pauseScreen.delegate = self;
        [self.topLayer addChild:self.pauseScreen];
    }
}
```

Agora que já temos os métodos de pause criados na `GameScreen`, vamos voltar à classe `PauseScreen.m` e referenciar as ações dos botões, para que notifiquem o *delegate*:

```
- (void)resumeGame:(id)sender
{
    if ([self.delegate respondsToSelector:
        @selector(pauseScreenWillResumeGame:)]) {
        [self.delegate pauseScreenWillResumeGame:self];
        [self removeFromParentAndCleanup:YES];
    }
}

- (void)quitGame:(id)sender
{
    if ([self.delegate respondsToSelector:
        @selector(pauseScreenWillQuitGame:)]) {
        [self.delegate pauseScreenWillQuitGame:self];
    }
}
```

Ao rodar o projeto, a tela de pause deve aparecer mas ainda temos trabalho a fazer.

11.6 PAUSANDO OS OBJETOS

Temos toda a arquitetura preparada para o pause, mas algo muito importante ainda não foi feito. Ao rodar o projeto e apertar o pause, temos a tela aparecendo, porém, não temos os objetos parando.

Precisamos usar a classe `Runner` que controla o estado do game para isso. Será através dos controles dessa classe que poderemos definir se os objetos como meteoros, tiros e avião devem se mover no `game loop`.

A lógica para isso será sempre igual. Apenas movimentaremos um objeto na tela de jogo se o jogo não estiver em pause. Além disso, só podemos criar novos elementos se essa condição também for satisfatória.

Na `MeteorsEngine.m` adicione a verificação abaixo:

```
- (void)meteorsEngine:(float)dt
{
    // Checa se o jogo está em execução
    if ([Runner sharedRunner].isGamePaused == NO) {
        // Sorte: 1 em 30 gera um novo meteoro!
        if(arc4random_uniform(30) == 0) {
            if ([self.delegate respondsToSelector:
                @selector(meteorsEngineDidCreateMeteor:)]) {
                [self.delegate meteorsEngineDidCreateMeteor:
                    [Meteor meteorWithImage:kMETEOR]];
            }
        }
    }
}
```

Na `Meteor.m` adicione a verificação abaixo durante o `update`:

```
- (void)update:(float)dt
{
    // Checa se o jogo está em execução
    if ([Runner sharedRunner].isGamePaused == NO) {
        // Move o Meteoro para baixo
        self.positionY -= 1.0f;
        self.position = ccp(self.positionX, self.positionY);
    }
}
```

Na `Shoot.m` adicione a verificação abaixo:

```
- (void)update:(float)dt
{
    // Checa se o jogo está em execução
    if ([Runner sharedRunner].isGamePaused == NO) {
        // Move o Tiro para cima
        self.positionY += 2;
        self.position = ccp(self.positionX, self.positionY);
    }
}
```

Na classe `Player` adicione as verificações abaixo:

```
- (void)update:(float)dt
{
    // Checa se o jogo está em execução
    if ([Runner sharedRunner].isGamePaused == NO) {
        //... Demais códigos
    }
}

- (void)moveLeft
{
    // Checa se o jogo está em execução
    if ([Runner sharedRunner].isGamePaused == NO) {
        //... Demais códigos
    }
}

- (void)moveRight
{
    // Checa se o jogo está em execução
    if ([Runner sharedRunner].isGamePaused == NO) {
        //... Demais códigos
    }
}

- (void)shoot
{
    // Checa se o jogo está em execução
    if ([Runner sharedRunner].isGamePaused == NO) {
        //... Demais códigos
    }
}
```

```
}  
}
```

Muito trabalho, não? A tela de pause pode enganar, mas como foi visto, é um ponto chave no desenvolvimento dos games. Rode o projeto e verifique o comportamento dessa nova tela no jogo.

11.7 CONCLUSÃO

Fazer uma tela de pause é normalmente uma parte que os desenvolvedores deixam de lado na construção de um game e depois percebem a complexidade de sua implementação. Essa tela usa conceitos que se propagam por todo o game e deve ser feita com cuidado e planejamento.

A notícia boa é que se você chegou até aqui você já tem os conceitos principais para desenvolver um jogo! Imagens, *loop*, camadas, sons, colisões etc. Não são coisas fáceis, mas com a prática viram conceitos que se repetem por todo o game.

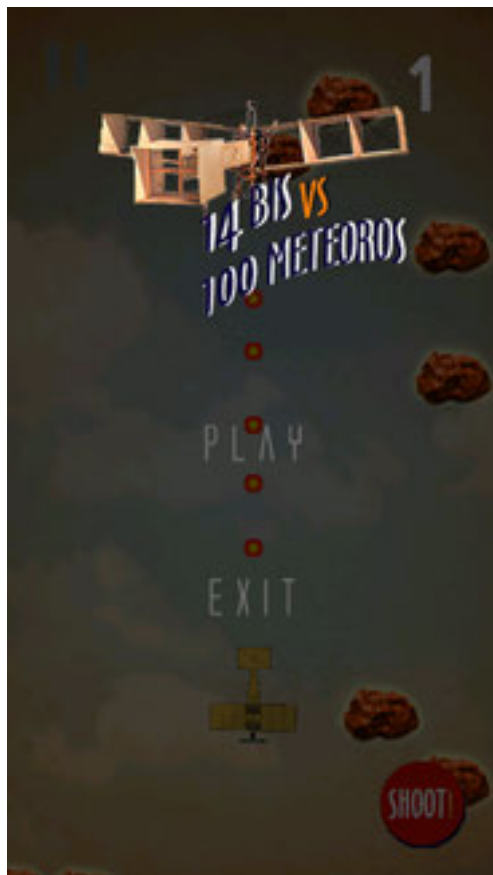


Figura 11.2: Tela de pause.

CAPÍTULO 12

Continuando nosso jogo

Depois de saber como criar a base de um jogo e sua estrutura, montar os cenário e iterações e criar telas de fluxo do game, muita coisa pode ser desenvolvida de acordo com a sua imaginação. A parte mais legal de agora em diante será pensar em ideias que possam tornar o jogo cada vez mais divertido e motivante. O mundo dos games é sempre um mundo de novidades, de inovações, em que cada nova ideia dá margem para milhares de novos jogos.

Esse capítulo mostrará sugestões de como tornar o game mais social, dinâmico e, quem sabe, rentável.

12.1 UTILIZANDO FERRAMENTAS SOCIAIS

Para tornar o jogo mais engajador podemos adicionar diversas funcionalidades como *rankings*, onde os usuários disputam quem é o melhor, *achievements* que servem como medalhas para provar conquistas durante o jogo e até monetizar o aplicativo com itens especiais.

A Apple fornece estas ferramentas de forma nativa, simplificando muito todo esse desenvolvimento, com os *frameworks* GameKit (para o Game Center) e StoreKit (para os In-App Purchases, que são as vendas dentro do jogo).

12.2 HIGHSCORE

Os *rankings* ou *highscores* no Game Center são chamados de *Leaderboards*. A ideia é criar um mural onde todos os usuários são ordenados para saber quem são os melhores.

A implementação é simples e consiste basicamente nos passos exemplificados a seguir. Primeiro, deve-se autenticar o usuário no Game Center:

```
GKLocalPlayer *localPlayer = [GKLocalPlayer localPlayer];
localPlayer.authenticateHandler = ^(UIViewController *viewController,
                                     NSError *error)
{
    if (viewController != nil) {
        [self showAuthenticationDialogWhenReasonable:viewController];
    } else if (localPlayer.isAuthenticated) {
        [self authenticatedPlayer:localPlayer];
    } else {
        [self disableGameCenter];
    }
};
```

Após a autenticação, pode-se enviar a pontuação para o Game Center, indicando em qual *Leaderboard* a mesma será registrada:

```
GKScore *scoreReporter = [[GKScore alloc]
                           initWithCategory:br.com.casadocodigo.bis.meteoros];
scoreReporter.value = meteorosDestruidos;

[scoreReporter reportScoreWithCompletionHandler:^(NSError *error)
{
    // Código executado após reportar o score
}];
```

Para exibir o Game Center dentro do seu jogo, pode-se implementar o código abaixo:

```

GKGameCenterViewController *gameCenterVC =
    [[GKGameCenterViewController alloc] init];
if (gameCenterVC != nil) {
    gameCenterVC.gameCenterDelegate = self;
    gameCenterVC.viewState = GKGameCenterViewControllerStateLeaderboards;
    gameCenterVC.leaderboardTimeScope = GKLeaderboardTimeScopeToday;
    gameCenterVC.leaderboardCategory = br.com.casadocodigo.bis.meteoros;
    [self presentViewController:gameCenterVC
        animated:YES
        completion:nil];
}

```

Single Leaderboard

Leaderboard Reference Name:

Leaderboard ID:

Score Format Type:

Score Submission Type: ☒ Best Score ☐ Most Recent Score

Sort Order: ☐ Low to High ☒ High to Low

Score Range (Optional): To

Leaderboard Localization

You must add at least one language below. For each language, provide a score format and a leaderboard name.

[Add Language](#)

Image	Language	Leaderboard Name	Score Format	
	Brazilian Portuguese	Meteoros	Integer (100,000,123)	Delete

Figura 12.1: Criando Leaderboards.

Mais detalhes sobre o Game Center e *Leaderboards* podem ser encontrados na documentação da Apple:

https://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/GameKit_Guide/Introduction/Introduction.html

12.3 ACHIEVEMENTS

Imagine se após destruir 10 meteoros o jogador ganhasse o título de `Piloto Pró`, e após destruir 50 ganhasse o título de `Piloto Master`. Que tal, ao fim do jogo, entregar o *Achievement* Santos Dumont ao jogador? Essas criações tornam o jogo mais atrativo, com mais objetivos, e inclusive podem ser compartilhadas em redes sociais.

A criação de *Achievements* usando o `Game Center` também é fácil. Você apenas define os *Achievements* do jogo e informa a porcentagem completada através do `Game Center`.

```
if (meteorosDestruídos == 50) {
    GKAchievement *achievement = [[GKAchievement alloc]
                                   initWithIdentifier:br.com.casadocodigo.bis.pilotomaster];
    if (achievement) {
        achievement.percentComplete = 100;
        [achievement reportAchievementWithCompletionHandler:
         ^(NSError *error) {
             if (error != nil) {
                 NSLog(@"Erro: %@", error);
             }
         }];
    }
}
```

Crie ideias interessantes de *Achievements* e implemente-as com o `Game Center`.

Achievement

Achievement Reference Name ?

Achievement ID ?

Point Value ?

170 of 1000 Points Remaining

Hidden Yes ☐ No ☒ ?

Achievable More Than Once Yes ☐ No ☒ ?

Achievement Localization

These are the languages in which your achievements will be available for display in Game Center. You must add at least one language.

[Add Language](#)


Image	Language	Title	
	Brazilian Portuguese	Piloto com mais de 50 meteoros destruídos!	Delete

Figura 12.2: Criando Achievements.

Mais detalhes sobre o `Game Center` e *Achievements* podem ser encontrados na documentação da Apple:

https://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/GameKit_Guide/Introduction/Introduction.html

12.4 DESAFIOS PARA VOCÊ MELHORAR O JOGO

Aqui vão algumas sugestões que visam desafiar o leitor a solidificar os conhecimentos adquiridos com o livro e ampliar o engajamento do jogo 14 bis.

Novos tiros

Todo jogador adora incrementar sua munição. Crie um tiro duplo para o 14 bis! Esses tiros devem sair não mais em linha reta mas sim formando um angulo de 45 graus para cada lado. Interessante fazer esse tiro ser dado ao jogador após alguma conquista do game, como destruir um número de meteoros, ou atirar em um elemento especial que dê esse poder!

Diferentes meteoros

Quem disse que todos os meteoros são iguais? Que tal implementar tamanhos diferentes de meteoros, que pontuam de forma diferente, de acordo com seu tamanho? Comece simples, meteoros maiores valem 2 pontos e meteoros menores continuam valendo 1. Com isso implementado, mude a imagem dos meteoros e até coloque esporadicamente outros elementos que podem valer 5 ou 10 pontos, mas que apareçam com uma frequência menor. Vale também fazer com que os meteoros tenham velocidades diferentes entre si!

Armaduras

Morrer com apenas um meteoro é chato, mas podemos capturar um elemento que fortifique o avião! Crie um elemento que funcione como uma armadura e permita a colisão entre um meteoro e o avião. Lembre-se de mostrar ao jogador que ele está equipado com esse elemento, mudando a cor do avião, por exemplo.

Efeitos nos sprites

Atualmente, os objetos são estáticos no nosso game. Que tal adicionar efeitos como fazer o meteoro descer girando ou luzes no avião? Esses pequenos detalhes fazem o jogo parecer muito mais atraente.

12.5 COMO GANHAR DINHEIRO?

O mundo dos games move um valor enorme e é hoje uma das maiores movimentações financeiras do mundo. Esse livro não visa trazer análises sobre esse mercado financeiro, porém hoje as cifras dos games superam os números do cinema.

Para monetizar o game você pode seguir por diversas abordagens, como estabelecer um valor de venda quando o usuário baixá-lo.

Uma outra forma bem interessante de ganhar dinheiro com o jogo é deixando-o gratuito ou cobrando um valor bem baixo, com o qual o usuário vai instalar o jogo sem muito esforço. Após isso, o jogo deve cativar o usuário e então pode-se oferecer itens a serem comprados que melhorem a experiência e performance do jogo.

O *framework* `StoreKit` da Apple oferece uma forma bem interessante e fácil de aplicar para monetizar o game. Você pode adicionar esses itens com poucas linhas de código, como descrito a seguir.

Inicie a loja interna de seu aplicativo já no `AppDelegate.m`, garantindo que quaisquer transações pendentes sejam processadas.

```
[[SKPaymentQueue defaultQueue] addTransactionObserver:self];
```

Busque no iTunes Connect todos os seus produtos, obtendo então quais estão disponíveis (ativos) e seus preços atuais para a loja do país do usuário.

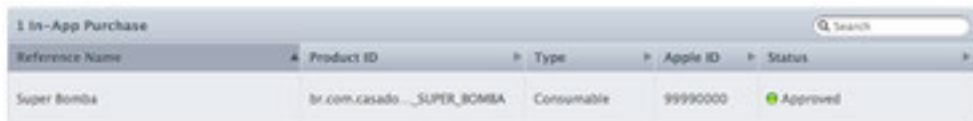
```
SKProductsRequest *request = [[SKProductsRequest alloc]
                               initWithProductIdentifiers:(NSSet *)productsID];
request.delegate = self;
[request start];
```

Quando o jogador escolher um item para a compra na loja, como uma “Super Bomba”, o jogo deve comunicar-se com o iTunes Connect, solicitando a compra do item.

```
SKPayment *payment = [SKPayment
                       paymentWithProduct:br.com.casadocodigo.bis.SUPER_BOMBA];
[[SKPaymentQueue defaultQueue] addPayment:payment];
```

O iTunes solicita o Apple ID e a senha do usuário automaticamente, caso necessário. Após processar a compra, o jogo é notificado e, caso tenha sido efetivada com sucesso, você deve liberar para o jogador o que ele comprou, como a “Super Bomba” do exemplo. Então você deve remover essa transação da fila de pagamentos, para que não seja processada novamente.

```
[[SKPaymentQueue defaultQueue] finishTransaction:transaction];
```



Reference Name	Product ID	Type	Apple ID	Status
Super Bomba	br.com.casado..._SUPER_BOMBA	Consumable	99990000	Approved

Figura 12.3: Exemplo de itens pagos.

Detalhes sobre `In-App Purchase` podem ser encontrados na documentação da Apple:

<https://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/StoreKitGuide/Introduction/Introduction.html>

12.6 CONCLUSÃO

Desenvolver jogos é uma tarefa complexa e ao mesmo tempo divertida, além de ser uma ótima maneira de elevar o conhecimento de programação. Atualmente, os celulares modernos trouxeram uma oportunidade única para que desenvolvedores possam ter essa experiência, criando seus próprios games, uma revolução similar à que a web e os computadores pessoais geraram alguns anos atrás.

Além disso, criar jogos é um exercício de criatividade que permite explorar a nossa imaginação e criar histórias interativas e únicas. Esse livro tenta compartilhar esses conhecimentos e ideias, esperando ser de fato útil a todos aqueles que estão ingressando nesse mágico mundo de jogos para iOS.

Fica novamente o convite para você participar da nossa lista de discussão:

<https://groups.google.com/group/desenvolvimento-de-jogos-para-ios>

Boa diversão!