

# **Programação Web para Jogos**



**Cruzeiro do Sul Virtual**  
Educação a distância



# Material Teórico



Frameworks para Desenvolvimento de Jogos 2D e 3D na Web

**Responsável pelo Conteúdo:**

Prof. Me. Alcides Teixeira Barboza Junior

**Revisão Textual:**

Prof.<sup>a</sup> Me. Luciene Oliveira da Costa Granadeiro



# UNIDADE

## Frameworks para Desenvolvimento de Jogos 2D e 3D na Web



- **Introdução;**
- **Iniciando o Uso do *Phaser*;**
- **Estrutura do Projeto de um jogo Web;**
- **Fluxo de Execução do *Phaser*;**
- **Tratamentos de Eventos com *Phaser*;**
- **Trabalhando com Colisão;**
- **Conceitos Iniciais Para a Criação de um Jogo de Plataforma.**



### OBJETIVO DE APRENDIZADO

- Apresentar os conceitos iniciais para a utilização do framework *Phaser*;
- Demonstrar a estrutura de execução do framework *Phaser*;
- Desenvolver um jogo básico para iniciar o uso do framework;
- Apresentar alguns princípios para a criação de um jogo estilo plataforma com o uso do *Phaser*.





# Orientações de estudo

Para que o conteúdo desta Disciplina seja bem aproveitado e haja maior aplicabilidade na sua formação acadêmica e atuação profissional, siga algumas recomendações básicas:

Determine um horário fixo para estudar.

Mantenha o foco! Evite se distrair com as redes sociais.

Procure manter contato com seus colegas e tutores para trocar ideias! Isso amplia a aprendizagem.

Seja original! Nunca plágie trabalhos.

Aproveite as indicações de Material Complementar.

Conserve seu material e local de estudos sempre organizados.

Não se esqueça de se alimentar e de se manter hidratado.

## Assim:

- ✓ Organize seus estudos de maneira que passem a fazer parte da sua rotina. Por exemplo, você poderá determinar um dia e horário fixos como seu “momento do estudo”;
- ✓ Procure se alimentar e se hidratar quando for estudar; lembre-se de que uma alimentação saudável pode proporcionar melhor aproveitamento do estudo;
- ✓ No material de cada Unidade, há leituras indicadas e, entre elas, artigos científicos, livros, vídeos e sites para aprofundar os conhecimentos adquiridos ao longo da Unidade. Além disso, você também encontrará sugestões de conteúdo extra no item **Material Complementar**, que ampliarão sua interpretação e auxiliarão no pleno entendimento dos temas abordados;
- ✓ Após o contato com o conteúdo proposto, participe dos debates mediados em fóruns de discussão, pois irão auxiliar a verificar o quanto você absorveu de conhecimento, além de propiciar o contato com seus colegas e tutores, o que se apresenta como rico espaço de troca de ideias e de aprendizagem.

# Contextualização

Desenvolvemos o jogo da Unidade 2 utilizando a tag canvas e programação pura em JavaScript, com o objetivo de mostrar como desenvolver jogos com esses recursos. Você achou mais complexa em relação à Unidade 1? Já que o nosso propósito é programação, vamos avançar mais.

Vimos que é possível desenvolver jogos utilizando as tags básicas do HTML ou a tag Canvas do HTML 5, nos dois casos, a programação em JavaScript é essencial. Mas, como você viu, fazer o jogo com a linguagem na sua forma pura, sem auxílio de outras ferramentas, torna nosso trabalho mais complexo e, muitas vezes, mais demorado.

Nesta unidade, iremos conhecer como utilizar um *framework* para desenvolvermos nossos jogos para Web de forma mais rápida e menos complexa. Ainda estaremos utilizando o JavaScript, contudo, iremos introduzir outros recursos fornecidos pelo *framework* chamado *Phaser* para facilitar nosso desenvolvimento. Em questão de conceitos de programação, iremos subir um pouco o nível, mas continuaremos a utilizar os conceitos que foram abordados até agora nas demais unidades.

Vamos começar!

# Introdução

Como estudamos na Unidade 1, existem diferentes formas ou ferramentas para desenvolvermos nossos jogos para Web. Optamos em utilizar tecnologias voltadas para Web como HTML, CSS e JavaScript.

Na Unidade 2 e 3, fizemos exemplos de jogos utilizando essas tecnologias na sua forma mais pura, ou seja, sem nenhuma *engine* ou complemento às linguagens. Como foi possível perceber, utilizando essas tecnologias, o processo se torna um pouco demorado visto que muitos recursos presentes na maioria dos jogos precisam ser recriados nas linguagens selecionadas.

O reaproveitamento de código é um conceito muito importante e que, de certa forma, deu origem às diferentes ferramentas que temos hoje em dia para desenvolvimento de jogos. Você poderia se questionar: “posso criar um recurso como tratamento de eventos para teclados e utilizar esse código em todos os meus jogos?”. A resposta seria: sim, basta você criar uma função genérica em um arquivo JS externo e utilizá-lo em todos os seus projetos.

Mesmo possuindo essa opção de reaproveitar seu código, que é interessante, você deve pensar sobre o que é mais importante no desenvolvimento, recriar uma rotina na linguagem que está utilizando para fazer algo já padrão nos jogos digitais, ou se preocupar com a programação da mecânica do jogo? Antes de responder, vamos ser realistas, é muito gratificante você criar suas próprias rotinas mesmo que saiba que já existe algo pronto.

Agora, respondendo à pergunta, é mais importante nos concentrarmos na lógica principal do jogo. As rotinas padrões podem ser simplesmente reaproveitadas, e isso faz com que o projeto ganhe agilidade, o que é muito importante. Sabemos que o tempo para desenvolver um jogo e lançá-lo no mercado é crucial, pois sempre há vários concorrentes que podem sair na frente com uma ideia igual, ou muito parecida com a nossa.

Seguindo nossa metodologia de usar as tecnologias Web para o desenvolvimento de jogos, e agora que já aprendemos os conceitos do HTML, CSS e JavaScript, precisamos subir um pouco o nível e fazer usos de códigos prontos para agilizarmos nosso trabalho. Vamos, então, entender e usar os chamados *frameworks*.

Um *framework* é um conjunto de códigos considerados genéricos, que são desenvolvidos em alguma linguagem de programação. No nosso caso JavaScript, sua função é disponibilizar funcionalidades específicas para o desenvolvimento da aplicação ou jogo.

Podemos pensar em um *framework* como uma caixa de ferramentas, que possui diversas funcionalidades prontas para serem utilizadas. Esse reuso poupa tempo ao desenvolvedor e trabalho na elaboração de rotinas já consolidadas como, por exemplo, atenção a inputs, carregamento de arquivos, sons, entre outros. Com

isso, perceba que você poderá focar na lógica principal do jogo e não nas funcionalidades que são gerais aos diferentes tipos de jogos.

Embora os *frameworks* ajudem o desenvolvedor, tenha em mente que a dependência total a eles não é algo muito interessante. Outro fator a considerar é analisar as atualizações que o *framework* que você escolheu recebe, pois quanto mais constante, mais garantias você terá em relação à compatibilidade e disponibilização de novos recursos para seus jogos.

Tenha o pensamento aberto, pois hoje você pode estar utilizando um *framework*, amanhã poderá ser outro que fará com que você tenha que reaprender algumas coisas. Isso é parte dessa área de desenvolvimento que está sempre em evolução.

Agora que você sabe o que é um *framework*, nesta unidade, iremos trabalhar com o *framework* chamado *Phaser* desenvolvido pela Photon Storm.

O *Phaser* foi idealizado a partir da biblioteca Flixel, voltada ao desenvolvimento de jogos. É um *framework* também conhecido como uma biblioteca de código, que tem o foco de rodar nos navegadores mais conhecidos em PCs e em dispositivos móveis.

Os desenvolvedores, com a evolução do *framework*, adicionaram ao seu núcleo o suporte ao CocoonJS para criação de jogos nativos que rodem em dispositivos móveis, essa funcionalidade tem como base o Apache Cordova. Não se preocupe com esses termos, não iremos desenvolver os jogos para dispositivos móveis, contudo, deixamos algumas sugestões de links para você pesquisar sobre esses tópicos.

O *Phaser* permite renderizar os jogos em 2D, ou, por meio de WebGL, o *framework* utiliza a biblioteca Pixi.js para renderização dos gráficos. Também possui compatibilidade com Web Audio API e trata eventos para teclado, touch e mouse nos jogos. Vamos, então, começar a trabalhar com o *Phaser* para entender sua estrutura e, assim, criarmos jogos com ele.



Para conhecer mais sobre o surgimento do *Phaser*, acesse o artigo disponível em:  
<https://goo.gl/rmbmsh>



Veja os trabalhos na empresa criado do *Phaser* em <https://goo.gl/grwno>  
Vale a pena você explorar a biblioteca Flixel no site <https://goo.gl/wYBq>  
Para conhecer mais sobre o Cocoon acesse <https://goo.gl/ZKetFj>  
Pesquise um pouco sobre o Apache Cordova para descobrir como rodar seus jogos feitos em HTML5 no celular, acesse <https://goo.gl/WICml>  
Veja o que seria o Pixi.js em <https://goo.gl/sys7oi>

# Iniciando o Uso do Phaser

Antes de criarmos um “Olá Mundo” no *Phaser*, precisamos conhecer alguns pontos importantes. Para criarmos o jogo, iremos carregar diversos arquivos de imagens, sons, arquivos de dados, entre outros. Por questões de segurança, os navegadores atuais bloqueiam a maioria dos acessos a esse tipo de carregamento por meio do protocolo HTTP://, contudo, o *Phaser* usa esse protocolo para carregar os arquivos do jogo. Você se pergunta: “como resolvemos isso?”.

A solução ideal que, inclusive, é colocada no site oficial do *framework* é instalarmos um servidor Web na nossa máquina e executarmos nossos jogos a partir dele. Deixaremos um link sobre esse assunto, e nossa sugestão de servidor Web simples seria o disponibilizado pelo Wamp. Não se preocupe, pois também deixaremos o link dessa ferramenta.

Acreditamos que, em um primeiro momento, você não precisará instalar o servidor, embora seja interessante. Vamos ser mais práticos. Mas, se não o instalarmos, como iremos rodar os jogos? Podemos simplesmente abrir o arquivo HTML no navegador?

Infelizmente, não podemos abrir o arquivo HTML do jogo diretamente no navegador se utilizarmos o protocolo HTTP. Isso deve ser feito por meio de um servidor Web. Se você se lembra das unidades anteriores, comentamos que o Brackets possui um ícone no formato de um relâmpago e, clicando nesse ícone, a ferramenta irá simular um servidor Web e abrir nossa página, então essa seria a solução mais prática para nós.

Caso você tenha problemas na execução por meio do Brackets, a solução seria você instalar o servidor Web e rodar por ele, acompanhe os links indicados no final desta seção caso seja necessário fazer isso em seu computador.

Agora, podemos avançar. O *Phaser* é um *framework* que disponibiliza alguns códigos prontos, isso você já aprendeu. Assim como outros *frameworks* do mercado, o *Phaser* é disponibilizado em alguns formatos para download e também pode ser utilizado fazendo uma referência a um URL (CDN).

Caso você prefira baixar o arquivo do *Phaser*, irá se deparar com quatro opções gerais. A primeira é a opção de fazer um clone do Github; a segunda é baixar o arquivo .js sem compactação; a terceira é baixar o arquivo min.js que seria a versão compactada do *framework* (essa compactação retira as quebras de linhas e comentários do código para reduzir seu tamanho); e a última opção é um arquivo zip. Sugerimos que você baixe a versão normal .js.

Se você optar em utilizar o URL (CDN), as duas opções possíveis são listadas abaixo. Escolha somente uma para inserir no seu código.

1. 

```
<script
src="//cdn.jsdelivr.net/npm/phaser@3.11.0/dist/phaser.js"></script>
```
2. 

```
<script
src="//cdn.jsdelivr.net/npm/phaser@3.11.0/dist/phaser.min.js"></script>
```

A diferença é que a primeira opção é o arquivo normal e a segunda o arquivo compactado como explicado anteriormente. Vamos recriar o famoso “Olá Mundo”, neste caso, do *Phaser*.

Crie uma pasta na sua máquina e dentro dela salve o arquivo *phaser.js* baixado do site oficial. Em seguida, utilizando o Brackets, abra a pasta e crie um arquivo novo com o nome *index.html*.

No arquivo HTML criado, vamos inserir o código da figura 1.

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8">
5      <script src="phaser.js"></script>
6  </head>
7  <body>
8      <script>
9          var config = {
10              type: Phaser.AUTO,
11              width: 800,
12              height: 600,
13              physics: {
14                  default: 'arcade',
15                  arcade: {
16                      gravity: { y: 200 }
17                  }
18              },
19              scene: {
20                  preload: preload,
21                  create: create
22              }
23          };
24
25          var game = new Phaser.Game(config);
26
27          function preload ()
28          {
29              this.load.setBaseURL('http://labs.phaser.io');
30              this.load.image('sky', 'assets/skies/space3.png');
31              this.load.image('logo', 'assets/sprites/phaser3-logo.png');
32          }
33
34          function create ()
35          {
36              this.add.image(400, 300, 'sky');
37              var logo = this.physics.add.image(400, 100, 'logo');
38              logo.setVelocity(100, 200);
39              logo.setBounce(1, 1);
40              logo.setCollideWorldBounds(true);
41          }
42          //código fonte retirado o site oficial Phaser
43      </script>
44  </body>
45  </html>
```

Figura 1 – Olá Mundo - *Phaser*

Após digitar o código apresentado na figura 1, clique no relâmpago para executar seu arquivo no servidor. Seu arquivo deverá apresentar um fundo de um céu escuro e o logo do *Phaser* batendo nos quatro cantos da tela.

O código digitado pode ser dividido em quatro partes. O bloco de código da linha 9, na linha 23, define uma variável com as configurações gerais do jogo. Essas configurações são atribuídas ao objeto *Phaser.Game* na linha 25, na qual se cria um objeto *game* com essas configurações.

A função *preload* (linhas 27 a 32) tem como tarefa carregar os arquivos necessários para a renderização. Já a função *create* (linhas 34 a 41), possui a tarefa de montar os elementos na tela (linha 36) e aplicar a física no logo (linhas 37 a 40).

Procure acessar o site do *framework* para consultar a documentação completa na qual você encontrará as referências mais atualizadas relacionadas aos comandos ou funcionalidades fornecidas pelo *Phaser*.



Para baixar o *Phaser*, acesse o link <https://goo.gl/588Zc4> e clique em Download ou acesse o link <https://goo.gl/Y2Jg2u> para ir direto para a página de download, escolha o arquivo JS ou o arquivo ZIP. Nossa sugestão é baixar somente o JS.

A documentação do *Phaser* 3.0 ainda está em desenvolvimento, contudo, a documentação da versão 2.6.2 pode ser consultada no link <https://goo.gl/bCacPs>



Consulte no site de exemplos do *Phaser* no endereço <https://goo.gl/pyeqd6> para conhecer alguns dos jogos criados com este *framework*.

Existem diversos tutoriais oficiais do *Phaser* disponíveis no endereço <https://goo.gl/UW2hHY>, dê uma espiada lá e tente fazer alguns exemplos.

Quando você tiver alguma dúvida pontual, pesquise no Fórum do *Phaser*, pois outros desenvolvedores podem ajudar – sua dúvida pode ser a mesma de várias pessoas – acesse o fórum do *Phaser* no link <https://goo.gl/bGHEZB>

Um fórum bem interessante para você conhecer alguns exemplos de jogos de pessoas que estão utilizando tecnologias/ferramentas Web está disponível no link <https://goo.gl/xQTuAe>, explore os exemplos e, quando finalizar um jogo com *Phaser*, publique nesse fórum. Os comentários dos demais desenvolvedores serão bastante úteis para seu projeto

Leia o artigo da página oficial do *Phaser* sobre a necessidade de um servidor Web disponível em <https://goo.gl/L1hsZC>

Caso você queira instalar um servidor Web em seu computador, pode utilizar o WampServer disponível no link <https://goo.gl/T3uH>. Esse pacote instalará, em sua máquina, o Apache Web Server, a linguagem PHP que roda no servidor e o banco de dados MySQL.

## Estrutura do Projeto de um Jogo Web

Você já pensou como organizar seus arquivos e pastas quando for desenvolver um jogo? Geralmente seguimos uma estrutura, não que exista algo obrigatório para se definir a organização, mas é interessante seguir algum padrão.

Por exemplo, algumas engines possuem uma organização de pastas padrões entre os desenvolvedores, como o caso da Unity. Nessa engine, você encontrará geralmente algumas pastas padrões, como, por exemplo, Script, Texture, Model, Prefabs, entre outras.

Para o desenvolvimento de jogos para Web, podemos também criar um padrão, mas tem que ficar claro que você poderá definir sua estrutura, daremos duas sugestões referentes à estrutura de organização para seu projeto na figura 2.

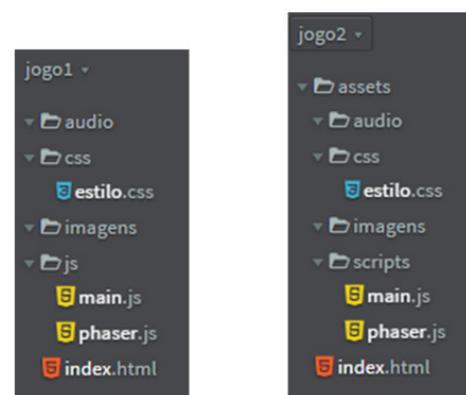


Figura 2 – Sugestões de organização de um projeto de jogo para Web

A figura 2 (a) apresenta uma organização mais semelhante ao padrão de desenvolvimento de aplicações Web, já a figura 2 (b) apresenta uma sugestão de organização para projetos de jogos criados com o *framework Phaser*. Defina o seu padrão e siga o mesmo em todos os seus projetos, procure sempre manter a organização dos arquivos em pastas para facilitar manutenções futuras.

## Fluxo de Execução do *Phaser*

O *Phaser* trabalha com um fluxo de execução que faz uso de algumas funções (também conhecido como métodos em algumas linguagens) que são específicas para cada etapa do processamento do jogo. Assim como um *framework* possui um fluxo de execução, uma engine também possui algo semelhante, cada um desses possui suas particularidades, contudo, são semelhantes na essência.

O fluxo de execução é conhecido também como o loop do jogo, o que, em outras palavras, seria a repetição realizada que permite ao jogo ser atualizado, tanto em lógica quanto em questões de renderização. Geralmente os jogos trabalham com uma taxa de atualização de 60 quadros por segundo (60 fps).

A figura 3 apresenta o loop do jogo que, muitas vezes, é padrão em diferentes *frameworks* ou engines.

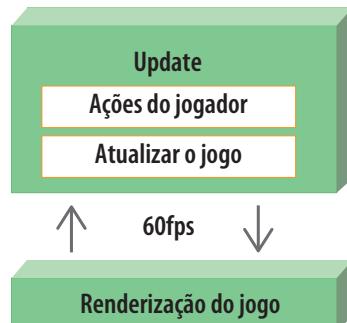


Figura 3 – Loop do Jogo

O *Phaser* inclui esse loop dentro do seu fluxo de execução que é denominado de estados do jogo. Veja alguns desses estados que são executados conforme apresentado na figura 4.

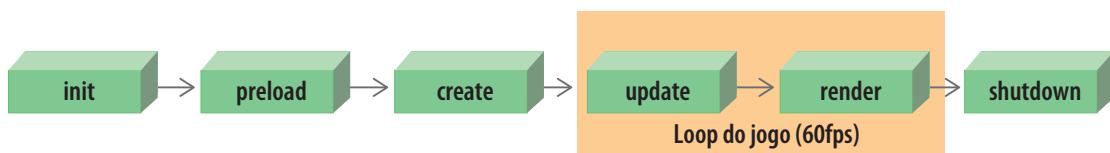


Figura 4 – Principais estados do jogo no *Phaser*

Dos estados apresentados na figura 4, podemos utilizar somente o init, o preload, o create e o update. Claro que os demais são importantes, mas tudo dependerá

da necessidade. Esses quatro estados que citamos são suficientes para criarmos alguns exemplos.

Cada estado possui uma função específica conforme definições a seguir:

- ***init***: método chamado assim que se inicia o jogo, utilizado para configurar algumas variáveis ou atributos do jogo;
- ***preload***: método chamado para fazer o pré-carregamento dos elementos utilizados no jogo; com esse carregamento, caso seja necessário reutilizar um elemento, o mesmo será carregado a partir do cache, o que torna mais rápida a execução;
- ***create***: método chamado após a execução do preload, sua tarefa é criar os elementos do jogo na tela;
- ***update***: método chamado durante o loop do jogo.

Vamos criar um código para verificar a execução dos estados preload, create e update. Crie uma pasta e copie para dentro dela o arquivo do framework *phaser.js*. Em seguida, abra a pasta no Brackets e, dentro de um arquivo .html novo, insira o código exibido na figura 5.

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8">
5      <title> Exemplo </title>
6      <script src="js/phaser.js"></script>
7  </head>
8  <body>
9      <script>
10     var config = {
11         type: Phaser.AUTO,
12         width: 600,
13         height: 600,
14         scene: [
15             preload,
16             create,
17             update
18         ]
19     };
20
21     var game = new Phaser.Game(config);
22
23     function preload() {
24         console.log("Executou a função PRELOAD");
25     }
26
27     function create() {
28         console.log("Executou a função CREATE");
29     }
30
31     function update() {
32         console.log("Executou a função UPDATE");
33     }
34     </script>
35 </body>
36 </html>

```

Figura 5 – Exemplo de código para testar os estados do *Phaser*

Execute o exemplo e, ativando o painel de desenvolvedor do Chrome (tecla F12), visualize as mensagens na aba console. Veja a sequência de execução das funções.

# Tratamentos de Eventos com Phaser

O *Phaser* possui diversos recursos para tratamento de eventos, podemos utilizar, por exemplo, a função `addListener` da classe `KeyboardPlugin`, da classe `Gamepad`, entre outras. Ou podemos usar métodos mais específicos como, por exemplo, o método `createCursorKeys` da classe `KeyboardPlugin`.

O método `createCursorKeys` cria e retorna um objeto com os valores para as setas direcionais, espaço e tecla Shift, com isso, poderíamos criar facilmente a movimentação de um personagem por meio das setas do teclado.

Você deve se perguntar, mas só podemos usar as setas? A resposta é não. O *Phaser* permite utilizar diferentes meios de entrada no jogo como toque em tela para dispositivos móveis (celular, por exemplo), teclado, mouse, entre outros.

Vamos criar um pequeno exemplo para movimentar um personagem pela tela do jogo, utilizando as setas direcionais. Crie uma pasta para esse exemplo e, dentro dela, crie as seguintes subpastas: js e imagens. Na pasta js copie, o `phaser.js` e, na pasta imagens, baixe ou copie algum arquivo do seu interesse que irá representar nosso personagem que terá movimento.

Após finalizar a criação da pasta, subpastas e copiar os arquivos, abra a pasta principal no Brackets, crie um arquivo HTML na pasta principal e um arquivo chamado `main.js` dentro da pasta js. Em seguida, insira o código apresentado, na figura 6, no arquivo HTML criado.

```
1  <!DOCTYPE html>
2 ▼ <html>
3 ▼ <head>
4      <meta charset="utf-8">
5      <title> Exemplo </title>
6      <script src="js/phaser.js"></script>
7      <script src="js/main.js"></script>
8  </head>
9 ▼ <body>
10
11  </body>
12  </html>
```

Figura 6 – Código do arquivo HTML para mover um personagem

A linha 6 do código apresentado na figura 6, importa o arquivo `phaser.js` para nossa página, já a linha 7, faz a importação do arquivo `main.js` que contém a lógica principal do nosso exemplo.

Agora vamos criar o código para mover nosso personagem, abra o arquivo `main.js` e insira no arquivo o código apresentado na figura 7.

```

1 ▼ var config = {
2     type: Phaser.AUTO,
3     width: 400,
4     height: 400,
5     backgroundColor: '#ffffa7d',
6     physics: {
7         default: 'arcade'
8     },
9     scene: [
10         preload,
11         create,
12         update
13     ]
14 };
15
16 var setas;
17 var personagem;
18
19 var game = new Phaser.Game(config);
20
21 ▼ function preload() {
22     this.load.image('pinguim', 'imagens/pinguim.png');
23 }
24
25 ▼ function create() {
26     setas = this.input.keyboard.createCursorKeys();
27     personagem = this.physics.add.sprite(100, 100, 'pinguim');
28 }
29

```

Figura 7 – Código do arquivo main.js para mover um personagem (parte 1)

No código da figura 7, destacamos a linha 6 que adiciona física ao jogo. Adicionando isso, temos a opção de trabalhar com conceitos de físicas nos elementos do jogo. Outra linha que merece destaque é a linha 22, na qual estamos carregando uma imagem que está no caminho “imagens/pinguim.png” e atribuímos o nome de “pinguim”.

A função create (linha 25) irá criar os nossos elementos. Primeiro, configuramos o tratamento de eventos das setas direcionais na variável setas (linha 26). Na linha 27, estamos configurando o nosso personagem como sendo a imagem previamente carregada com o nome de “pinguim” e também já adicionando física ao personagem que nos permitirá mover o mesmo.

Agora, precisamos trabalhar com o loop do jogo, no qual iremos atualizar a posição do personagem na tela conforme as setas pressionadas pelo jogador. Após a função create, insira a função update e o código apresentado na figura 8.

```

30 ▼ function update() {
31     personagem.setVelocity(0);
32
33     if (setas.left.isDown) {
34         personagem.setVelocityX(-300);
35     } else if (setas.right.isDown) {
36         personagem.setVelocityX(300);
37     }
38
39     if (setas.up.isDown) {
40         personagem.setVelocityY(-300);
41     } else if (setas.down.isDown) {
42         personagem.setVelocityY(300);
43     }
44 }

```

Figura 8 – Código do arquivo main.js para mover um personagem (parte 2)

A figura 8 mostra dois blocos de IFs. Lembre-se de que precisamos verificar quando uma tecla referente à seta for pressionada e aplicar a velocidade ao personagem conforme a respectiva direção. Se pressionarmos as setas esquerda (linha 33) ou direta (linha 35), devemos aplicar uma velocidade negativa (linha 34) ou positiva (linha 36) respectivamente ao eixo X.

Já para mover o personagem no eixo Y, devemos verificar se a seta para cima (linha 39) ou para baixo (linha 41) foi pressionada. Caso isso ocorra, também devemos aplicar uma velocidade negativa (linha 40) ou positiva (linha 42) respectivamente no eixo Y.

Execute seu código no navegador e tente mexer o personagem com as setas direcionais. Caso você encontre algum erro no console (F12), revise seu código com o nosso para achar possíveis erros.



Consulte o link da documentação do *Phaser 3* para verificar os meios de entrada possíveis para o seu jogo. O link direto para a parte de Input (entrada) está disponível em <https://goo.gl/eZXR6>

## Trabalhando com Colisão

As colisões são muito importantes nos jogos, elas determinam quando um objeto tocou outro e, com isso, podemos realizar diversas ações como, por exemplo, contar pontos quando o personagem colide com uma moeda, retirar energia do personagem quando ele colidir com um inimigo, fazer o personagem morrer ao colidir com um obstáculo de grande prejuízo, entre outros.

Existem diferentes algoritmos para determinar colisões. Em linhas gerais, é a matemática que utilizamos para determinar quando um objeto toca outro. Não se preocupe, não iremos abordar essa parte matemática, iremos utilizar os recursos que o *Phaser* oferece para determinar colisão.

O *Phaser* possui duas funções principais de colisão. Uma dessas funções é específica para determinar se o objeto tocou as laterais da área do jogo.

A sintaxe dessa função é:

- **objeto.setCollideWorldBounds([true ou false]);**

Já para determinar a colisão entre objetos do jogo, podemos utilizar a função collide, cuja sintaxe é:

- **objeto.collide(object1 [, object2] [, collideCallback]);**

Podemos passar objetos para se verificar quando um colidiu com o outro, ou então um vetor de objetos para verificar a colisão. O collideCallBack é opcional, seria a função executada no caso de ocorrer a colisão entre os objetos.

Vamos fazer um exemplo para testar essas funções, mas perceba que o *Phaser* possui outras e, conforme o mecanismo de física que se está utilizando, essas funções podem ganhar mais recursos e performance.

Nosso exemplo irá consistir de duas bolas (uma imagem de bola de basquete e outra de futebol), que irão ficar colidindo com as laterais da área do jogo, e uma pequena plataforma.

Para começar, crie uma pasta, copie para ela o *framework Phaser*, em seguida baixe a imagem de duas bolas de algum esporte que você goste. Agora Abra a pasta no Brackets para começarmos.

Vamos criar um arquivo .html com o código apresentado na figura 9.

```

1  <!DOCTYPE html>
2 ▼ <html>
3 ▼ <head>
4      <meta charset="utf-8">
5      <title> Colisão </title>
6      <script src="js/phaser.js"></script>
7      <script src="js/main.js"></script>
8  </head>
9 ▼ <body>
10
11 </body>
12 </html>
```

Figura 9 – Código do arquivo HTML de colisão

Precisamos criar a lógica principal desse exemplo, então crie um arquivo main.js. Em nosso exemplo, o arquivo foi criado na pasta js. Dentro desse arquivo, vamos começar pelo código que configura nosso jogo. Digite o código apresentado na figura 10.

```

1 ▼ var config = {
2     type: Phaser.AUTO,
3     width: 800,
4     height: 600,
5     parent: 'colisao',
6     physics: {
7         default: 'arcade',
8         arcade: {
9             gravity: { y: 100 }
10        }
11    },
12    scene: {
13        preload: preload,
14        create: create,
15        update: update
16    }
17 };
18
19 var bola1, bola2, barreira;
20
21 var game = new Phaser.Game(config);
```

Figura 10 – Código do arquivo main.js de colisão (parte 1)

No código apresentado na figura 10, estamos configurando o tamanho do canvas (linhas 3 e 4); indicamos que iremos utilizar o motor de física padrão “arcade”

(linha 7) e que nossa gravidade possui valor de 100 (linha 9). Os estados do jogo, irão seguir o padrão do *Phaser* (linhas 12 a 16).

Como nosso exemplo utiliza 3 objetos, precisamos definir uma variável para cada bola e uma variável para a barreira (linha 19).

Depois do objeto do jogo criado (linha 21), precisamos carregar as imagens e criar os elementos dentro do jogo e, consequentemente, na tela.

Insira no seu arquivo o código apresentado pela figura 11.

```

23  function preload ()
24  {
25      this.load.image('futebol', 'imagens/futebol.png');
26      this.load.image('basketball', 'imagens/basketball.png');
27      this.load.image('tijolo', 'imagens/tijolo.png');
28  }
29
30  function create ()
31  {
32      bola1 = this.add.image(100, 100, 'futebol');
33      bola2 = this.add.image(400, 100, 'basketball');
34      barreira = this.add.image(250, 520, 'tijolo');
35
36      this.physics.world.enable([ bola1, bola2,barreira ]);
37
38      barreira.body.setAllowGravity(false)
39      barreira.body.setImmovable(true)
40
41      bola1.body.setVelocity(50, 100).setBounce(1, 1).setCollideWorldBounds(true);
42      bola2.body.setVelocity(100, 100).setBounce(1, 1).setCollideWorldBounds(true);
43  }
44

```

Figura 11 – Código do arquivo main.js de colisão (parte 2)

A função preload, exibida na figura 10, carrega 3 imagens do tipo png da pasta imagens. Já a função create está criando as imagens na tela (linhas 32 a 34). Para que nossos objetos possam colidir e responder às leis da física, devemos determinar quais irão receber as influências dessas leis. A linha 36 está habilitando a física para os três objetos do jogo.

Como a barreira não precisa de gravidade, é necessário desabilitar essa ação (linha 38) e tornar o objeto estático (linha 39).

Analise as linhas 41 e 42. Nesse ponto, estamos configurando a colisão com as laterais da área do jogo, para isso, utilizamos a função setColliderWorldBounds. Só por curiosidade: a função setBounce está determinando o pulo do objeto, o valor 1 será de 100% e pode variar até 0.

A detecção da colisão com os objetos precisa ser feita uma a uma, também precisa ser verificada a cada ciclo do “jogo”, ou seja, precisamos verificar isso no loop do jogo, consequentemente, iremos fazer essa verificação na função update.

Insira o código apresentado na figura 11 referente à função update do jogo.

```

45  function update ()
46  {
47      this.physics.world.collide(bola1, bola2);
48      this.physics.world.collide([bola1,bola2],barreira, function(){
49          console.log("oi")
50      });
51  }

```

Figura 12 – Código do arquivo main.js de colisão (parte 3)

A função `update` verifica a colisão entre as duas bolinhas (linha 47). Caso ocorra a colisão, por padrão, o sentido dos objetos é alterado. Caso você queira criar sua função específica para ser executada quando ocorrer uma colisão, poderá seguir a sintaxe apresentada na linha 48.

A segunda verificação de colisão (linha 48) checa se cada um dos elementos no vetor delimitado por “[ ]” colidiu com a barreira. Caso isso ocorra, será executada a função anônima. Nesse pequeno exemplo, a função executada na colisão exibe somente um “Olá” no console do navegador (F12).

## Conceitos Iniciais Para a Criação de um Jogo de Plataforma

Vamos agora juntar tudo o que estudamos para criar um exemplo que servirá para o desenvolvimento de um jogo de plataforma. Nesse exemplo, iremos abordar os estados do jogo padrões do *Phaser*, o loop do jogo e principalmente a programação JavaScript.

Antes de iniciarmos, é importante destacar que você deve tomar cuidado com a versão que está utilizando do *Phaser*. Alguns tutoriais na Internet usam a versão antiga, logo, muitas funções dessas versões antigas não irão funcionar na última versão. No nosso exemplo, estaremos utilizando a versão 3.11.

O exemplo que estaremos estudando agora foi baseado nos exemplos disponibilizados no *Phaser Labs*, um repositório com diversos modelos fornecidos pelos desenvolvedores do framework com diversos exemplos prontos.

A figura 13 apresenta a tela do jogo finalizado.

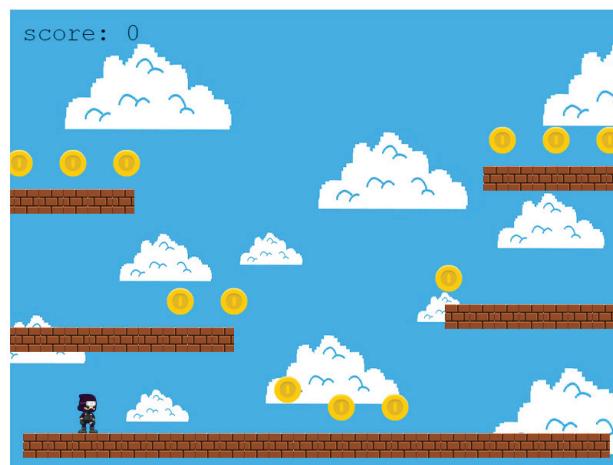


Figura 13 – Exemplo de jogo de plataforma adaptado do site *Phaser Labs*

Para iniciarmos o jogo, baixe os materiais complementares disponíveis no ambiente virtual, abra a pasta fornecida no Brackets e edite o arquivo HTML inserindo a referências aos scripts conforme exibido na figura 14.

```
1  <!DOCTYPE html>
2 ▼ <html>
3 ▼ <head>
4      <meta charset="utf-8">
5      <title> Ninja </title>
6      <script src="js/phaser.js"></script>
7      <script src="js/main.js"></script>
8  </head>
9 ▼ <body>
10
11 </body>
12 </html>
```

Figura 14 – Código do jogo de plataforma Ninja (parte 1)

As linhas 6 e 7 exibidas na figura 14 fazem a importação dos arquivos .js para a página HTML. Lembrando que você deve deixar esses arquivos na pasta chamada js. Caso faça alguma alteração nesse nome, altere também as referências aos mesmos no HTML.

Vamos inserir agora a programação JavaScript no arquivo main.js para que nosso jogo comece a ganhar forma.

O primeiro passo é inserir as configurações iniciais do jogo, como tamanho do canvas, a descrição das funções para cada cena etc. Insira no arquivo main.js o código apresentado na figura 15.

```
1 ▼ var config = {
2     type: Phaser.AUTO,
3     width: 800,
4     height: 600,
5     physics: {
6         default: 'arcade',
7         arcade: {
8             gravity: { y: 300 }
9         }
10    },
11    scene: {
12        preload: preload,
13        create: create,
14        update: update
15    }
16};
17
18 var player;
19 var coins;
20 var platforms;
21 var cursors;
22 var score = 0;
23 var scoreText;
24
25 var game = new Phaser.Game(config);
26
```

Figura 15 – Código do jogo de plataforma Ninja (parte 2)

Nesse código apresentado na figura 15, foi configurado o motor de física do Phaser para “arcade” (linha6) e já configuramos a gravidade para 300 (linha 8). Demais linhas de código são para a criação das variáveis que irão ser processadas durante o jogo como, por exemplo, score, cursores, player etc.

Seguindo os estados do jogo, a primeira função que iremos implementar será a função preload, lembre-se de que essa função possui a tarefa de carregar todos os assets que serão usados no nosso jogo. Vamos inserir quatro imagens normais e um spritesheet. Veja o código na figura 16.

```

27  function preload ()
28  {
29      this.load.image('sky', 'imagens/sky2.png');
30      this.load.image('ground', 'imagens/plataforma.png');
31      this.load.image('coin', 'imagens/coinGold.png');
32      this.load.image('bomb', 'imagens/bomb.png');
33      this.load.spritesheet('ninja', 'imagens/ninja.png',
34                           { frameWidth: 45, frameHeight: 54 });
35  }
36
  
```

Figura 16 – Código do jogo de plataforma Ninja (parte 3)

As linhas do intervalo entre 29 a 32 carregam as imagens básicas do jogo, contudo, perceba que, na linha 33, utilizamos a função spritesheet. Um *spritesheet* pode ser uma imagem com diversos sprites ou animações (quadro a quadro) que podemos inserir no jogo, criando uma única imagem com todos os elementos ou animações, o que facilita o carregamento e reduz o uso da memória. Isso é considerado uma boa prática por muitos desenvolvedores.

Quando carregamos um spritesheet, devemos especificar o tamanho dos quadros, perceba que na linha 34 (continuação da função), definimos que cada sprite terá a largura de 45px e altura de 54px, a figura 16 demonstra um esquema dessas medidas na imagem *ninja.png*.

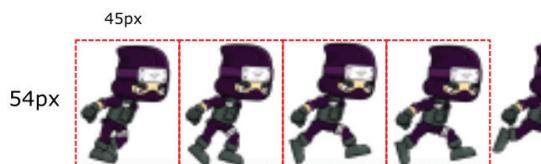


Figura 17 – Spritesheet *ninja.png*

Cada frame possui a mesma medida, com isso, fica mais simples criar uma animação que dará a ideia de que o personagem está andando.

Precisamos agora criar todos os elementos do jogo na tela. Para isso, implemente a função *create* conforme apresentado na figura 18.

```

37  function create ()
38  {
39      this.add.image(400, 300, 'sky');
40
41      platforms = this.physics.add.staticGroup();
42
43      platforms.create(400, 568, 'ground').refreshBody();
44
45      platforms.create(950, 400, 'ground');
46      platforms.create(-220, 250, 'ground');
47      platforms.create(-90, 430, 'ground');
48      platforms.create(1000, 220, 'ground');
49
50      player = this.physics.add.sprite(100, 450, 'ninja');
51
52      player.setCollideWorldBounds(true);
53
54      this.anims.create({
55          key: 'left',
56          frames: this.anims.generateFrameNumbers('ninja',
57                                          { start: 21, end: 31 }),
58          frameRate: 25,
59          repeat: -1
60      });
61
62      this.anims.create({
63          key: 'turn',
64          frames: this.anims.generateFrameNumbers('ninja',
65                                          { start: 11, end: 20 }),
66          frameRate: 10,
67          repeat: -1
68      });
69
70      this.anims.create({
71          key: 'right',
72          frames: this.anims.generateFrameNumbers('ninja',
73                                          { start: 0, end: 10 }),
74          frameRate: 25,
75          repeat: -1
76      });
77
78      cursors = this.input.keyboard.createCursorKeys();
79
80      coins = this.physics.add.group({
81          key: 'coin',
82          repeat: 11,
83          setXY: { x: 12, y: 0, stepX: 70 }
84      });
85
86      coins.children.iterate(function (child) {
87          child.setBounceY(Phaser.Math.FloatBetween(0.4, 0.8));
88      });
89
90      scoreText = this.add.text(16, 16, 'score: 0',
91                               { fontSize: '32px', fill: '#000' });
92
93      this.physics.add.collider(player, platforms);
94      this.physics.add.collider(coins, platforms);
95
96      this.physics.add.overlap(player, coins, collectCoin, null, this);
97  }

```

Figura 18 – Código do jogo de plataforma Ninja (parte 4)

Algumas linhas de código apresentadas, na figura 18, já foram estudadas anteriormente, assim, iremos comentar os blocos de códigos novos. Analise o intervalo de linhas compreendido entre a linha 54 a 76. Nesses blocos, criamos as animações que irão percorrer o spritesheet carregado anteriormente. Nosso arquivo do personagem irá funcionar como um vetor em que cada posição corresponde a um quadro da animação.

Abra seu arquivo `ninja.png` e perceba, por exemplo, que, do quadro 0 ao quadro 10, temos o personagem correndo para a direita; do quadro 11 ao quadro 20, temos o personagem de frente realizando um movimento sutil como se estivesse “vivo”; já do quadro 21 ao 31, temos o personagem correndo para a esquerda.

A criação de cada animação com a função `anims.create` exige uma chave (`key`) para identificar a animação, os quadros (`frames`) que compõem a animação e o `frameRate` que corresponde à velocidade que deverá ser utilizada para percorrer o conjunto de quadros definido. A partir da chave, poderemos chamar e iniciar a animação sempre que necessário.

Outro ponto que merece destaque é relacionado à colisão do jogo. O personagem poderá colidir com as plataformas (linha 93), quando ocorrer essa colisão, o personagem não consegue avançar. Isso é replicado para as moedas quando elas colidem com as plataformas também. Perceba que, na linha 96, a função de colisão que está sendo utilizada se chama `overlap`.

A função overlap também irá detectar a colisão do personagem com as moedas, mas, nesse caso, a colisão permitirá que o personagem continue se movimentando após capturar a moeda que colidiu. A função responsável em contar pontos a cada moeda coletada é camada de collectCoin.

Para encerrar o protótipo do nosso jogo, precisamos implementar as funções update e collectCoin. Insira agora o código apresentado na figura 19.

```

99  function update ()
100 {
101   if (cursors.left.isDown)
102   {
103     player.setVelocityX(-160);
104     player.anims.play('left', true);
105   }
106   else if (cursors.right.isDown)
107   {
108     player.setVelocityX(160);
109     player.anims.play('right', true);
110   }
111   else
112   {
113     player.setVelocityX(0);
114     player.anims.play('turn', true);
115   }
116
117   if (cursors.up.isDown && player.body.touching.down)
118   {
119     player.setVelocityY(-330);
120   }
121 }
122
123 function collectCoin (player, star)
124 {
125   star.disableBody(true, true);
126   score += 10;
127   scoreText.setText('Score: ' + score);
128 }
```

Figura 19 – Código do jogo de plataforma Ninja (parte 4)

Nós já estudamos a função update. Nesse jogo, ela é responsável em verificar se as setas foram pressionadas para movimentar nosso personagem. Adicionamos somente a função play() da animação para que reproduza os quadros do spritesheet sempre que apertarmos as setas para a esquerda (linha 104) ou para a direita (linha 109).

Por fim, a função collectCoin é responsável em adicionar 10 pontos a cada moeda capturada (linha 126) e exibir essa informação na tela do jogo (linha 127).

Agora, sim, chegamos ao final de um jogo um pouco mais completo utilizando o framework Phaser. Teste o jogo clicando no relâmpago do Brackets. Caso não seja inicializado o servidor no seu navegador, volte ao Brackets e pressione a tecla F5, em seguida, repita o processo para verificar se o servidor é inicializado e seu jogo está rodando.

Lembre-se de que, para rodar o jogo, que faz uso do Phaser, é necessário fazer uso de um servidor, isso é solucionado pelo relâmpago do Brackets como já comentado anteriormente.

Continue praticando, procure alterar seu cenário e criar outros elementos como inimigos, por exemplo.



Veja um tutorial passo a passo para criar o jogo Breakout 2d com o Phaser no site MDN web docs (Mozilla) disponível em <https://goo.gl/kmFqph>

Acesse o site Game Art 2D que disponibiliza alguns assets gratuitos para sprites, cenários e GUI. Link: <https://goo.gl/D583Qi>

# Material Complementar

## Indicações para saber mais sobre os assuntos abordados nesta Unidade:



Sites

**W3SCHOOLS HTML: The Language for Building Pages**

<https://goo.gl/gLZ9v>

**Sololearn – Everyone can code**

<https://goo.gl/zAeEe8>

**Codecademy**

<https://goo.gl/OTm7rG>

**Tutoriais MDN Mozilla**

<https://goo.gl/QKTkc7>

**Phaser Framework**

<https://goo.gl/ij8P8H>

# Referências

- BUCHARD, E. **The Web Game Developer's Cookbook:** Using JavaScript and HTML5 to Develop Games. Addison-Wesley, 2013.
- RAMTAL, D.; DOBRE, A. **Physics for JavaScript games, animation, and simulations:** with HTML5 canvas. New York: Apress, 2014.
- SILVA, M. S. **Html 5:** a linguagem de marcação que revolucionou a web. São Paulo: Novatec, 2011.

## Sites visitados

---

WORLD WIDE WEB CONSORTIUM W3C -Disponível em: <https://www.w3.org/html/> Acesso em: 09/06/2018.



**Cruzeiro do Sul**  
Educacional