CRIANDO UM JOGO DE PLATAFORMA COM PHASER

Esta aula apresenta o processo de criação de um jogo de plataforma com Phaser. O conteúdo é apresentado em partes. Ao final, obteremos um pequeno jogo sobre um jogador correndo e pulando por plataformas, colecionando estrelas e evitando os inimigos.

Parte 1: estrutura padrão do jogo

```
<!doctype html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8" />
    <title>Plataforma</title>
    <script
src="//cdn.jsdelivr.net/npm/phaser@3/dist/phaser.min.js">
    </script>
    <style type="text/css">
        body {
            margin: 0;
        }
    </style>
</head>
<body>
    <script type="text/javascript">
        var config = {
            type: Phaser.AUTO,
            width: 800,
            height: 600,
            scene: {
                preload: preload,
                create: create,
                update: update
            }
        };
        var game = new Phaser.Game(config);
            function preload() {
            }
            function create() {
            }
            function update() {
            }
    </script>
</body>
</html>
```

Temos na estrutura:

- definição da variável `config`
- criação da instância do jogo (`Phaser`.Game) informando o config
- definição da função `preload()`
- definição da função `create()`
- definição da função `update`()

A variável (objeto) `config` representa a configuração do jogo. Há muitas configurações disponíveis, mas aqui são definidas as seguintes:

- o atributo `type` indica o tipo do jogo. Pode ser `Phaser.CANVAS`, `Phaser.WEBGL` ou `Phaser.AUTO` (o padrão)
- `width` e `height` definem, respectivamente, largura e altura da área de desenho
- `scene` indica as funções `preload`(), `create`(), e `update`()

Parte 2: carregando e exibindo assets

Substitua o código da função preload() pelo seguinte:

```
function preload ()
{
    this.load.image('sky', 'assets/sky.png');
    this.load.image('ground', 'assets/platform.png');
    this.load.image('star', 'assets/star.png');
    this.load.image('bomb', 'assets/bomb.png');
    this.load.spritesheet('dude','assets/dude.png',
        { frameWidth: 32, frameHeight: 48 }
    );
}
```

O código utiliza o método `image()` do objeto `load` (da classe `Phaser.Game`) para carregar quatro imagens. O primeiro parâmetro é o nome do asset dentro do jogo e o segundo é seu caminho (utilizando como base a pasta atual).

O método `spritesheet()` é utilizado para carregar um sprite. O terceiro parâmetro define as dimensões (32 pixels de largura, 48 pixels de altura).


32px
48px

Os assets foram apenas carregados. Falta exibi-los. Para isso, comece substituindo a função `create()` pelo seguinte:

```
function create () {
    this.add.image(400, 300, 'sky');
}
```

O resultado é que o jogo exibe o asset chamado `sky` (o plano de fundo do jogo). Para isso, usa o objeto `add` e o método `image()` cujos parâmetros são:

- coordenada x
- coordenada y
- nome do asset

Nesse caso, usa como o origem da imagem o seu centro (não o canto superior esquerdo) e o posiciona na coordenada (400, 300) da tela.

Continue mostrando as demais imagens. Para isso, substitua a função create() por:

```
function create() {
    this.add.image(400, 300, 'sky');
    this.add.image(400, 300, 'star');
}
```

O importante a considerar aqui é a ordem em que as imagens são exibidas: seguem a ordem da chamada da função `image()`. Assim, na ordem do código, o asset `sky` é apresentado primeiro, depois o asset `star`.

Parte 3: criando o mundo

O método `image()` cria um objeto de imagem e o adiciona à lista de exibição atual da cena. Esta lista contém todos os objetos do jogo. Você poderia posicionar a imagem em qualquer lugar e o `Phaser` não se importaria. Claro, se estiver fora da região definida na configuração (0x0 a 800x600), você não o verá, porque estará "fora da tela", mas ainda existirá dentro da cena.

A cena em si não tem tamanho fixo e se estende infinitamente em todas as direções. O sistema de câmera controla sua visualização e você pode mover e aplicar zoom na câmera que está ativa conforme necessário. Você também pode criar novas câmeras para outras visualizações da cena.

Vamos construir a cena adicionando uma imagem de fundo e algumas plataformas. Substitua a função `create()` pelo seguinte:

```
var platforms;

function create() {
    this.add.image(400, 300, 'sky');

    platforms = this.physics.add.staticGroup();

    platforms.create(400, 568,'ground').setScale(2).refreshBody();

    platforms.create(600, 400, 'ground');
    platforms.create(50, 250, 'ground');
    platforms.create(750, 220, 'ground');
}
```

Além disso, substitua a configuração do jogo por:

```
var config = {
    type: Phaser.AUTO,
    width: 800,
    height: 600,
    physics: {
        default: 'arcade',
        arcade: {
            gravity: { y: 300 },
            debug: false
        }
    },
    scene: {
        preload: preload,
        create: create,
        update: update
    }
};
```

Primeiro, a configuração do jogo adiciona o atributo `physics`, que determina as configurações da física do jogo. Segundo, a função `create()`, além do passo 2, utiliza `physics.add.staticGroup()` para criar um grupo de objetos armazenado em `platforms` (o grupo das plataformas) e `create()` para inserir objetos no grupo. Os parâmetros para `create()` são: as coordenadas do objeto e o nome do asset (nesse caso, o asset é `ground`).

No Arcade Physics existem dois tipos de corpos físicos: Dinâmico e Estático. Um corpo dinâmico é aquele que pode se movimentar por meio de forças como velocidade ou aceleração. Ele pode saltar e colidir com outros objetos e essa colisão é influenciada pela massa do corpo e outros elementos.

Em contraste, um corpo estático simplesmente tem uma posição e um tamanho. Não é afetado pela gravidade, você não pode ajustar a velocidade dele e quando algo colide com ele, ele nunca se move. Estático por nome, estático por natureza. E perfeito para o chão e plataformas que vamos deixar o jogador correr por aí.

Parte 4: adicionar o jogador

Adicione uma variável player e modifique a função create() para incluir as linhas a seguir:

```
player = this.physics.add.sprite(100, 450, 'dude');

player.setBounce(0.2);
player.setCollideWorldBounds(true);

this.anims.create({
    key: 'left',
    frames: this.anims.generateFrameNumbers('dude’,
                                            { start: 0, end: 3 }),
    frameRate: 10,
    repeat: -1
});
```

```
this.anims.create({
    key: 'turn',
    frames: [{ key: 'dude', frame: 4 }],
    frameRate: 20
});

this.anims.create({
    key: 'right',
    frames: this.anims.generateFrameNumbers('dude’,
                                            { start: 5, end: 8 }),
    frameRate: 10,
    repeat: -1
});
```

A primeira parte do código faz o seguinte:

- cria o sprite usando `physics.add.sprite()`
- define o valor de "ressalto" (quicar) para 0.2 usando `setBounce()`; significa que quando aterrissar depois de pular, ele saltará muito levemente
- configura o sprite para colidir com os limites do mundo usando `setCollideWorldBounds()` (os limites são definidos na configuração do jogo, ou seja, 800x600)

A outra parte do código trata de animações. O asset dude é um sprite. Ele possui 9 quadros: 4 para correr à esquerda, 1 para olhar para a câmera (frente) e 4 para correr à direita. O código cria três animações: `left`, `turn`, `right`, cada uma usando `anim.create()` cujo parâmetro é um objeto com os atributos:

- `key`: o nome da animação
- `frames`: os frames usados na animação
- `frameRate`: a taxa de atualização dos quadros
- `repeat`: o valor -1 indica para a animação fazer um loop

Assim a animação `left` usa os quadros 0, 1, 2 e 3; a animação `turn` usa apenas o quadro 4; a animação `right` usa os quadros 5, 6, 7 e 8.

Parte 5: física

O Phaser tem suporte a uma variedade de diferentes sistemas de física, cada um atua como um plugin disponível para qualquer cena. No presente momento ele dispõe de **Arcade Physics**, **Impact Physics** e **Matter.js Physics**. Aqui estamos usando o sistema **Arcade Physics** para o nosso jogo.

Quando um sprite é criado, ele recebe uma propriedade body, que é uma referência ao Arcade Physics Body. Este representa o sprite como um corpo físico na engine Arcade Physics.

Quando você executar o jogo verá que o player cai sem parar, ignorando as plataformas. A razão para isso é que ainda não estamos testando a colisão entre o solo e o jogador.

Para permitir que o jogador colida com as plataformas, podemos criar um objeto `Collider`. Esse objeto monitora dois objetos físicos (`Groups` também podem ser incluídos) e verifica colisões ou sobreposição entre eles. Se isso ocorrer, é possível opcionalmente, invocar seu *callback*, mas não exigimos isso apenas para colidir com as plataformas. Adicione a linha ao método `create`():

```
this.physics.add.collider(player, platforms);
```

Parte 6: controlando o jogador com o teclado

Isso já sabemos fazer. Em `create()`, antes da linha que cria o *Collider* adicione:

```
cursors = this.input.keyboard.createCursorKeys();
```

E substitua o código da função `update()` por:

```
function update() {
    if (cursors.left.isDown) {
        player.setVelocityX(-160);
        player.anims.play('left', true);
    }
    else if (cursors.right.isDown) {
        player.setVelocityX(160);
        player.anims.play('right', true);
    }
    else {
        player.setVelocityX(0);
        player.anims.play('turn');
    }

    if (cursors.up.isDown && player.body.touching.down) {
        player.setVelocityY(-330);
    }
}
```

A primeira coisa que o código faz é verificar se a tecla esquerda está sendo pressionada. Se for o caso, aplicamos uma velocidade horizontal negativa e iniciamos a animação de execução `'left'` (usando `player.setVelocityX(-160)` e `player.anims.play('left', true)`).

Se a tecla da direita da direita esta sendo pressionada, em vez disso, nós fazemos o oposto (`player.setVelocityX(160)` e `player.anims.play('right', true)`). Já que estamos resetando a velocidade e ajustando-a desta maneira, em cada quadro, cria-se um estilo de movimento "stop-start".

O sprite do jogador se moverá apenas quando uma tecla estiver pressionada e parará imediatamente quando não estiver. O Phaser também permite criar movimentos mais complexos, com impulso e aceleração, mas já temos o efeito que precisamos para este jogo. A parte final da verificação da tecla define a animação para 'turn' e zera a velocidade horizontal caso nenhuma tecla estiver pressionada.

A parte final do código adiciona a capacidade de pular, ou seja, trata a situação em que a tecla up (seta para cima) está pressionada. O cursor para cima é a nossa tecla de salto e testamos se está pressionada. No entanto, também testamos se o jogador está tocando o chão, se não fizermos isto, ele poderá pular no ar.

Se ambas as condições forem atendidas, aplicamos uma velocidade vertical de 330 px/seg (usando `player.setVelocityY(-330)`). O jogador cairá no chão automaticamente por causa da gravidade.

Parte 7: propósito

É hora de dar ao nosso joguinho um propósito. Vamos jogar algumas estrelas na cena e permitir que o jogador as colete. Para conseguir isso, criaremos um novo grupo chamado 'stars' e o preencheremos.

Ajuste o código da função `create()` para conter o seguinte ao final:

```
stars = this.physics.add.group({
    key: 'star',
    repeat: 11,
    setXY: { x: 12, y: 0, stepX: 70 }
});

stars.children.iterate(function (child) {
    child.setBounceY(Phaser.Math.FloatBetween(0.4,
0.8));
});

this.physics.add.collider(player, platforms);
this.physics.add.collider(stars, platforms);

this.physics.add.overlap(player, stars, collectStar,
null, this);
```

O código cria o grupo `stars`, que você já viu como fazer. Depois, o código utiliza `stars.children.iterate()` para criar uma função que itera pelos elementos do grupo de estrelas e aplica um *bounce* diferente para cada um.

Na sequência, são definidas as colisões e, ao final, utilizando `physics.add.overlap()`. Essa função permite executar um código quando ocorrer uma colisão entre o player e uma das estrelas. Nesse caso, o código chama a função `collectStar()`:

```
function collectStar(player, star) {
    star.disableBody(true, true);
}
```

O código chama `star.disableBody()`, que faz com que a estrela tenha sua física desabilitada e seu **Game Object** pai fica inativo e invisível, removendo sua exibição.

Parte 8: pontuação

Para fazer isso, vamos utilizar um **Text Game Object**. Criamos duas novas variáveis, uma para guardar a pontuação real outra para o próprio objeto de texto:

```
var score = 0;
var scoreText;
```

O objeto `scoreText` é instanciado na função `create()`:

```
scoreText = this.add.text(16, 16, 'score: 0’,
                          { fontSize: '32px', fill: '#000' });
```

O texto é criado por meio de add.text(), sendo que os parâmetros são:

- posição na coordenada x
- posição na coordenada y
- o texto, em si
- objeto que representa o estilo do texto

Por fim, o texto da pontuação precisa ser atualizado quando o jogador coletar uma estrela. Para isso, substitua função `collectStars()` por:

```
function collectStar(player, star) {
    star.disableBody(true, true);

    score += 10;
    scoreText.setText('Score: ' + score);
}
```

O código atualiza o valor da variável score e chama o método `setText()` para definir o novo texto (com a pontuação atualizada). Nesse caso, cada estrela coletada vale 10 pontos.

Parte 9: Vilões

Já temos a pontuação do jogo para quando o jogador coletar estrelas. Agora é hora de adicionar alguns vilões, o que aumentará o desafio e dará um novo propósito ao jogo.

A ideia é: Quando você coletar todas as estrelas pela primeira vez, será lançada uma bomba saltitante. A bomba ficará pulando aleatoriamente pela fase e, se você colidir com ela, você morre. Todas as estrelas irão reaparecer para que você as colete de novo, e se você o fizer, uma nova bomba será lançada. Isto irá dar ao jogador um desafio: conseguir a pontuação mais alta sem morrer.

Vamos começar adicionando duas variáveis do script: `bombs`, para conter o grupo de bombas, e `gameOver`, para controlar a lógica de execução do jogo (inicializa com o valor false e recebe true quando o jogador morrer -- termina o jogo). Na sequência, carregamos a imagem da bomba no `preload()` usando `load.image()`.

Depois, criamos um **Group** de física para as bombas (variável `bombs`) e também definimos as colisões. Fazemos isso adicionando o seguinte ao método `create()`:

```
bombs = this.physics.add.group();
this.physics.add.collider(bombs, platforms);
this.physics.add.collider(player, bombs, hitBomb, null, this);
```

As bombas vão quicar nas plataformas e, se o jogador encostar em alguma delas, chamamos a função `hitBomb()` :

```
function hitBomb(player, bomb) {
    this.physics.pause();
    player.setTint(0xff0000);
    player.anims.play('turn');
    gameOver = true;
}
```

O código faz uma pausa no sistema de física (`physics.pause()`), pinta o jogador de vermelho (`setTint()`), toca a animação "turn" (`player.anims.play()`) e indica que o jogo terminou (`gameOver = true`).

Na sequência, modificamos a função `collectStar()` :

```
function collectStar(player, star) {
    star.disableBody(true, true);

    score += 10;
    scoreText.setText('Score: ' + score);

    if (stars.countActive(true) === 0) {
        stars.children.iterate(function (child) {
            child.enableBody(true, child.x, 0, true, true);
        });

        var x = (player.x < 400) ?
            Phaser.Math.Between(400, 800) : Phaser.Math.Between(0, 400);

        var bomb = bombs.create(x, 16, 'bomb');
        bomb.setBounce(1);
        bomb.setCollideWorldBounds(true);
        bomb.setVelocity(Phaser.Math.Between(-200, 200), 20);
        bomb.allowGravity = false;
    }
}
```

O método `countActive()` permite saber quantos elementos do grupo (as estrelas) estão ativas (não foram coletadas ainda). Se não houver uma estrela ativa, então o jogador coletou todas, com isso usamos a função `stars.children.iterate()` para habilitar todas as estrelas do grupo novamente e redefinir suas posições para zero para que elas caiam do topo da tela de novo.

Na sequência, o código cria a bomba. Para isso, define um valor aleatório para a coordenada **x** , sempre de um lado oposto ao do jogador (veja que é possível saber a posição do jogador no eixo usando `player.x`) (experimente desconsiderar isso para verificar se aumenta o nível de dificuldade). O restante do código faz o seguinte:

- cria uma bomba no conjunto de bombas (`bombs.create()`)
- define o quique (`setBounce()`)
- configura para colidir com o mundo (`setCollideWorldBounds()`)
- configura para ter uma velocidade aleatória no eixo **x**
- e constante no eixo **y** (`setVelocity()`)
- configura para ter a posição afetada pela gravidade (`allowGravity`)

Para finalizar, adicione o seguinte no início da função `update()` :

```
if (gameOver) {
    return;
}
```