**Relatório Técnico de Desenvolvimento do Jogo Vampixel**

**Paulo Matos, Rodrigo Braga**

Universidade do Estado do Amazonas (UEA)

Amazonas – AM – Brasil

**{paulojomatos, rodrigogrow}@gmail.com**

***Resumo:*** *Este trabalho tem como objetivo apresentar alguns aspectos encontrados no desenvolvimento do Vampixel, o qual consiste em um jogo de plataforma. O jogo foi desenvolvido utilizando o framework Phaser.io. Dentre as atividades desenvolvidas durante o processo de criação, podemos destacar: a entidade Game Manager, as transformações do personagem e a lógica do boss.*

***Abstract:*** *The work aims to present some aspects encountered in Vampixel development, which consists of a platform game. The game was developed using the Phaser.io framework. Among the activities performed during the creating process of the game, stand out: Game Manager entity, player transformations and the boss logic.*

**1. INTRODUÇÃO**

Nos dias atuais os jogos de computador estão cada vez mais presentes nas vidas das pessoas, quase sempre simplesemente pela diversão. Entretanto, eles não apenas fazem bem para o nosso cérebro, como também preservam nossas faculdades mentais [Ryan Anderson 2016].

Este trabalho apresenta alguns aspectos técnicos do desenvolvimento do jogo de plataforma 2D Vampixel. O jogo é feito para a plataforma web utilizando tecnologias como HTML5 e JavasScript. O framework Phaser.io foi utilizado como *game engine*[[1]](#footnote-2) para ganho de produtividade.

**2. Game Manager**

O Game Manager é a entidade principal da arquitetura do jogo. Assim como todos os módulos, também é uma IIFE (Immediately Invoked Function Expression). Entretanto, ele exporta um objeto que se torna disponível globalmente. Com esse objeto, podemos criar uma nova instância do jogo assim como gerenciar states, sprites, módulos e variáveis globais.

Outra vantagem oferecida pela arquitetura foi o desacoplamento de sprites e states. O melhor exemplo de desacoplamento que temos é o player, que é um único sprite importado em vários states. Quando um sprite é importado como uma injeção de dependência ele não é recriado, ou seja, uma mesma instância é compartilhada entre vários states. Esse comportamento permite o player não ter seus atributos (e.g. vida e capa de invisibilidade) resetados na transição de uma fase para outra.

**2.1 Métodos e atributos**

O Game Manager é composto por uma série de atributos e métodos responsáveis pela sua lógica. Esses elementos são listados a seguir.

**2.1.1 Método addState**

Adiciona um novo state no array privado de states. Esses states serão instanciados ao jogo na fase de criação.

var addState = function (name, theClass) {

states[name] = theClass;

}

2.1.1.1 Parâmetros

|  |  |
| --- | --- |
| **name** | O nome do state |
| **theClass** | A classe contendo a lógica do state |

2.1.1.2 Exemplo de uso

gameManager.addState('level4', Level4State);

**2.1.2 Método Create**

Cria uma nova instância do jogo, vincula essa instância do jogo aos sprites, instancia states ao jogo e executa o método *start* do *mainState*.

var create = function (width, height, renderer, parent, mainState) {

this.createNewGameInstance(width, height, renderer, parent);

setTimeout(function () {

bindGameToSprites();

addAllStatesToGame();

getGameInstance().state.start(mainState);

}, 0);

return this;

}

2.1.2.1 Parâmetros

|  |  |
| --- | --- |
| **width** | Largura da tela do jogo em pixels |
| **height** | Altura da tela do jogo em pixels |
| **renderer** | Renderizador do jogo. No Vampixel usamos Phaser.CANVAS |
| **parent** | Id do elemento HTML de referência para o Canvas do Phaser |
| **mainState** | Nome do state inicial |

2.1.2.2 Exemplo de uso

gameManager.create(800, 600, Phaser.CANVAS, 'phaser-canvas', 'menu');

**2.1.3 Método addSprite**

Adiciona um novo sprite no array privado de sprites. Esses sprites serão vinculados ao jogo na fase de criação.

var addSprite = function (name, Sprite) {

sprites[name] = Sprite;

}

2.1.3.1 Parâmetros

|  |  |
| --- | --- |
| **name** | O nome do state |
| **Sprite** | A classe contendo a lógica do sprite |

2.1.3.2 Exemplo de uso

gameManager.addSprite('boss', Boss);

**2.1.4 Método getSprite**

Recupera um sprite do array privado de sprites.

vargetSprite=function(name) {

if(sprites.hasOwnProperty(name)) {

return sprites[name];

}

else {

throw new ReferenceError("Sprite '"+ name +"' was not found");

}

}

2.1.4.1 Parâmetros

|  |  |
| --- | --- |
| **name** | O nome do sprite |

2.1.4.2 Exemplo de uso

this.player = gameManager.getSprite('player');

**2.1.5 Método addModule**

Adiciona um novo módulo ao array privado de módulos. Esses módulos podem ser injetados em qualquer entidade do jogo através do método *getModule*.

varaddModule=function(name, func) {

modules[name] =func;

}

2.1.5.1 Parâmetros

|  |  |
| --- | --- |
| **name** | O nome do módulo |
| **func** | A função construtora contendo a lógica do módulo |

2.1.5.2 Exemplo de uso

gameManager.addModule('playerPreload', playerPreload);

**2.1.6 Método getModule**

Recupera um módulo do array privado de módulos.

var getModule = function (name) {

if(modules.hasOwnProperty(name)) {

returnmodules[name];

}

else {

throw new ReferenceError("Module '"+ name +"' was not found");

}

}

2.1.6.1 Parâmetros

|  |  |
| --- | --- |
| **name** | O nome do módulo |

2.1.6.2 Exemplo de uso

Player.prototype.preload = gameManager.getModule('playerPreload');

**2.1.7 Método createNewGameInstance**

Cria uma nova instância do jogo Phaser.

var createNewGameInstance = function (w, h, r, p) {

gameInstance = new Phaser.Game(w, h, r, p);

return this;

}

2.1.7.1 Parâmetros

|  |  |
| --- | --- |
| **w** | Largura da tela do jogo em pixels |
| **h** | Altura da tela do jogo em pixels |
| **r** | Renderizador do jogo. No Vampixel usamos Phaser.CANVAS |
| **p** | Id do elemento HTML de referência para o Canvas do Phaser |

2.1.7.2 Exemplo de uso

var create = function (width, height, renderer, parent, mainState) {

this.createNewGameInstance(width, height, renderer, parent);

...

**2.1.8 Método getGameInstance**

Recupera a instância do jogo criada pelo Phaser.

var getGameInstance = function () {

return gameInstance;

}

2.1.8.1 Exemplo de uso

getGameInstance().state.start(mainState);

**2.1.9 Método getSprites**

Recupera todos os sprites registrados no array privado.

var getSprites = function () {

return sprites;

}

2.1.9.1 Exemplo de uso

var sprites = gameManager.getSprites();

**2.1.10 Método getSprites**

Recupera todos os states registrados no array privado.

var getStates = function () {

return states;

}

2.1.10.1 Exemplo de uso

var states = gameManager.getStates();

**2.1.11 Atributo *globals***

Usado como um singleton global para compartilhar variáveis entre os states.

2.1.11.1 Exemplo de uso

gameManager.globals.lives=1;

**3. Transformações**

Dentre as características mais divertidas do jogo, nós temos as transformações em lobo e morcego. Essas transformações são apresentadas a seguir.

**3.1 Lobo**

Quando o jogador precisa passar por obstáculos maiores ou simplesmente se movimentar mais rápido, ele tem como opção a transformação em lobo. Para usar essa transformação do Vampixel basta segurar a tecla *SHIFT*. O código que lida com essa lógica está no arquivo *js/sprites/player/setup.js*:

this.runButton = this.game.input.keyboard.addKey(Phaser.Keyboard.SHIFT);

this.runButton.onDown.add(this.startWolfTransformation, this);

this.runButton.onUp.add(this.cancelWolfTransformation, this);

As implementações dos métodos *startWolfTransformation* e *cancelWolfTransformation* estão no arquivo *js/sprites/player/sprite.js*:

Player.prototype.startWolfTransformation = function () {

if(!this.isDead && !this.isDoubleJumping) {

this.isWolf = true;

this.setNormalOrWolfAnimation('walk', 'wolfRun');

}

}

Player.prototype.cancelWolfTransformation = function () {

if(!this.isDead && !this.isDoubleJumping) {

this.isWolf = false;

this.setNormalOrWolfAnimation('walk', 'wolfRun');

}

}

O método *setNormalOrWolfAnimation* irá basicamente substituir a animação do player baseado no atributo booleano *isWolf*.



**Figura 1. Transformação em lobo**

**3.2 Morcego**

Por ser um vampiro, Vampixel tem entre suas transformações a característica de se transformar em um morcego. O jogador pode usar essa habilidade através do pulo duplo. Um comportamento especial dessa transformação é a queda em *slow fall*. O código que lida com essa lógica está implementado no arquivo *js/sprites/player/sprite.js*:

Player.prototype.jump = function () {

if(this.isDead || !this.canJump) return;

if(this.sprite.body.onFloor() || this.sprite.body.touching.down) {

this.isJumping = true;

this.setNormalOrWolfAnimation('singleJump', 'wolfRun', this.imageJumpName);

this.sprite.events.onAnimationComplete.add(function(){

this.sprite.loadTexture(this.imageName);

this.sprite.anchor.set(0.5);

},this);

return doJump.apply(this);

}

else if((!this.isDoubleJumping && !this.isWolf)) {

this.canJump = false;

this.isDoubleJumping = true;

this.setAnimation('batFly', this.imageBatFlyName);

return doJump.apply(this);

}

function doJump() {

this.sprite.body.velocity.y = this.jumpVelocity || -450;

this.soundJump.play();

}

}



**Figura 2. Transformação em morcego**

**4. BOSS**

Na fase 4 e última, Vampixel enfrenta o monge Lucius. Seu ataque base é o lançamento de crucifixos, mas ao atingir metade do seu total de vida, se transforma em Lúcifer, que tem como ataque base o lançamento de fogo. A lógica do boss está toda implementada no arquivo *js/sprites/boss\_sprite.js*. A seguir alguns métodos serão apresentados.

4.1 Pulo a cada 3 segundos

this.game.time.events.loop(Phaser.Timer.SECOND \* 3, function () {

self.sprite.body.velocity.y = self.jumpHeight;

}, this);

4.2 Método *fire*

Esse método leva em consideração o projétil atual a ser atirado, isso signfica, crucifixo para a transformação em monge e fogo para a transformação em demônio.

Boss.prototype.fire = function () {

if (this.game.time.now > this.bulletTime) {

this.bullet = this.bullets.create(0, 0, this.currentImageBullet);

if (this.bullet) {

if(this.state === 'normal') {

var y = this.sprite.y;

}

else if(this.state === 'demon') {

var y = this.sprite.y + 10;

}

this.bullet.reset(this.sprite.x, y);

if (this.sprite.scale.x == 1) {

this.bullet.body.velocity.x = 300;

this.bulletTime = this.game.time.now + 150;

} else {

this.bullet.body.velocity.x = -300;

this.bulletTime = this.game.time.now + 150;

}

}

}

}

4.3 Método *transform*

Esse método atualiza algumas propriedades do player como velocidade, altura, largura, tamanho do pulo, textura do projétil lançado e a animação para *demon*.

Boss.prototype.transform = function () {

this.state = 'demon';

this.sprite.body.velocity.y = -700;

this.sprite.body.width = 128;

this.sprite.body.height = 200;

this.jumpHeight = -700;

this.currentImageBullet = this.imageBulletFire;

this.sprite.loadTexture(this.imageDemonName);

this.sprite.animations.play('demon');

}



**Figura 3. Monge Lucius**



**Figura 4. Demônio Lúcifer**

**5. Conclusão**

O desenvolvimento do jogo Vampixel foi feito durante o curso de pós-graduação em desenvolvimento de jogos eletrônicos da Universidade do Estado do Amazonas e neste trabalho abordamos alguns aspectos técnicos das principais entidades implementadas bem como seus usos dentro da arquitetura do jogo. Conseguimos trabalhar em equipe utilizando metodologias ágeis e ferramentas como Phaser framework para acelerar na produtividade do processo. O resultado realizado foi satisfatório tendo em vista que o jogo possui elementos essenciais presentes em qualquer jogo, como animações de personagens, HUD, fases, itens bônus, habilidades especiais e um bom enredo.

**6. Trabalhos Futuros**

Durante o processo de desenvolvimento do Vampixel, exploramos pouco outras bibliotecas e ferramentas de terceiros que poderiam nos auxiliar na resolução de problemas ou melhorar funcionalidades já existentes. Um aspecto que com certeza pode ser melhorado é a inteligência do boss. Pretendemos utilizar algoritmos de aprendizagem de máquina usando *reinforcement learning[[2]](#footnote-3)* para esse fim.

**Referências**

[Ryan Anderson] Ryan Anderson (2016). 7 Reasons to Play Computer Games. https://www.psychologytoday.com/us/blog/the-mating-game/201603/7-reasons-play-computer-games. [Online; acessado em 20-Maio-2018].

1. Um framework contendo diversos recursos para auxiliar no desenvolvimento de jogos eletrônicos [↑](#footnote-ref-2)
2. Um modelo de aprendizado onde o agente aprende conforme as ações do ambiente [↑](#footnote-ref-3)