Universidade Federal do Rio Grande do Sul Instituto de Informática
INF01121 - Modelos de Linguagem de Programação

Marina Fortes Rey 220486
Nicolas Silveira Kagami 194636
Hélène Besancon 250242

Professor Lucas M. Schnorr

Jogo de Torre de Defesa Desenvolvido em Javascript

Porto Alegre/ RS
novembro de 2014

Conteúdo

1 Problema

1.1 Definição

Para o trabalho final da disciplina escolhemos desenvolver um jogo de defesa de torres. A dinâmica de um jogo desse gênero é ter um caminho pelo qual inimigos percorrem para atingir a sua base e, se chegarem lá, a danificam. O jogador perde caso sua base seja destruída e ganha caso, depois de todos os inimigos serem derrotados, sua base continuar de pé. Para derrotar os inimigos, usam-se torres, que os atacam de diversas maneiras diferentes, dependendo de deu tipo. Esses são os requisitos básicos de um jogo desse gênero.

O jogo desenvolvido, nomeado "Chicken Hendemption", trabalhou esses conceitos básicos em um jogo temático. Galinhas são os equivalentes das torres, que, com as suas diferentes habilidades, tentam proteger seus ovos de animais selvagens. Praticamente todos os elementos gráficos do jogo foram desenhados pelo grupo, tornando o jogo único e constante graficamente. Todas as músicas, sons e outras imagens utilizadas estão disponíveis na internet como material "opensource".

Como o jogo foi desenvolvido em javascript, pode ser jogado acessando o endereço "hendemption.fortesrey.net". Note que dependendo da velocidade da conexão com a internet da máquina o tempo de "loading" do jogo pode ser de alguns minutos, visto que precisamos carregas diversas imagens em alta resolução além de arquivos de audio. Também se o computador não possui placa de vídeo o processamento pode ser prejudicado, já que o rendering no computador é feito em WebGL, que usa a placa gráfica para otimizar o processamento.

1.2 Framework

Para desenvolver o jogo, utilizamos o framework "Phaser.js", específico para jogos desenvolvidos para web (Javascript). Ele possui portabildade tanto para computadores quanto para tablets e smartphones, conseguindo adaptar automaticamente a entrada do mouse para a entrada touchscreen, também podendo adaptar o tamanho da tela para os diferentes tamanhos disponíveis no mercado. Além disso, quando possível, é feito o rendering em WebGL apartir da biblioteca herdada "Pixi.js", o que torna o programa ainda mais eficiente. Escolhemos esse Framework graças a sua facilidade de trabalhar com Sprites, conseguindo facilmente detectar diversos tipos de input com qualidade "pixel perfect", e criando animações de forma simples e intuitiva. Outra facilidade é o uso de estados de jogo, conseguindo separar de forma modular os estados de preparação, menu e o jogo e si.

1.3 Projeto

O projeto utilizado para a criação do jogo visa diversos tamanhos de tela possíveis. Para cada tamanho foram geradas imagens de tamanhos adequados, podendo assim jogar o jogo em diversos dispositivos sem sobrecarregá-los com imagens desnecessariamente grandes. Isso foi possível por todas as imagens estarem em formato vetorial, podendo ser aumentadas sem alteração da qualidade.

Separamos o jogo em vários estados distintos, sendo os primeiros puramente para inicialização de componentes básicos. O primeiro estado acionado é o "Boot", que guarda algumas variáveis globais ao jogo, como o volume da música setado pelo usuário, além de redimensionar a tela e testar a orientação do aparelho (no caso de mobile, já que o jogo deve ser jogado com o aparelho na horizontal).

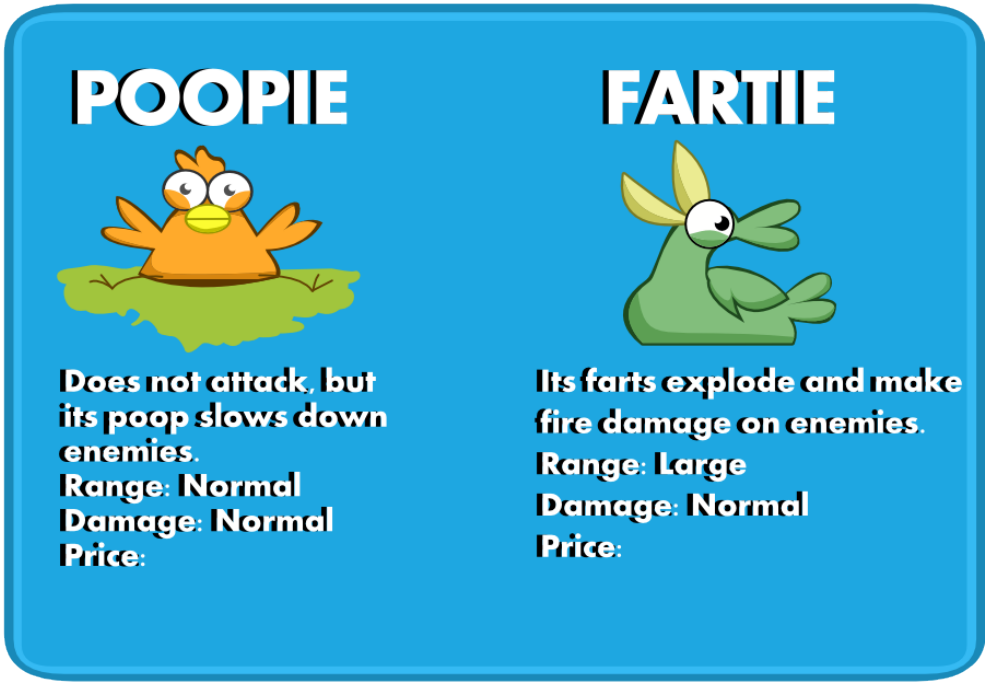
Depois disso, é chamado o estado "Preloader", que carrega todas as imagens, mapas e sons utilizados no jogo, podendo depois serem acessados apenas por uma string de identificação por todas as classes e estados. Por serem diversos arquivos carregados nessa parte, pode haver uma demora de segundos a minutos (dependendo do computador e da conexão à internet) para iniciar o jogo, por isso carregamos no estado de "Boot" uma imagem para avisar o jogador que o programa está carregando arquivos. Quando termina de carregar os arquivos, passa para o estado de "Intro", que é a tela de início de jogo. Dessa tela podemos passar para o estado "MainMenu". Nesse estados temos botões para acessar cada fase desenvolvida e botões para acessar as janelas de informação e de opções do jogo. Quando selecionada uma fase, o jogo passa para o estado "Game", que é onde está implementado o jogo em si.

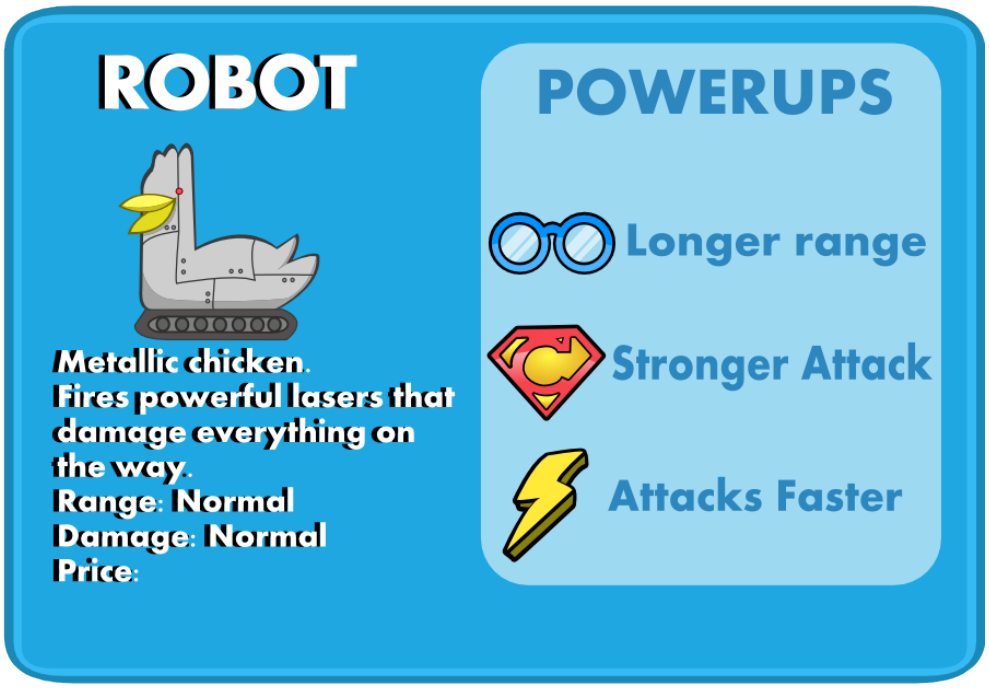
1.4 Funcionalidades

No jogo desenvolvido podemos jogar três fases distintas. Temos um total de cinco galinhas, sendo elas uma galinha comum, uma galinha com um grande pescoço que consegue atingir mais longe que as outras, uma galinha que diminui a velocidade dos inimigos por soltar líquidos de sua cloaca, uma galinha com um poderoso ataque de fogo e, por fim, uma galinha robótica que solta raios laser, atingindo todos os inimigos no seu caminho. Cada ataque possui uma animação distinta além de algumas possuirem efeitos sonoros (os botões do jogo, ao serem clicados, também possuem um efeito), podendo serem desabilitados ou alterados seus volumes pelo menu principal do jogo.

Para informar o jogador sobre as propriedades e habilidades de cada galinha foram criados painéis informativos no menu principal do jogo. Abaixo temos as imagens utilizadas em cada painel. Também foram feitos painéis semelhantes para explicar o funcionamento do jogo assim como informações sobre os inimigos encontrados nele.


NORMAL
Standard chicken.
Range: Normal
Damage: Normal
Price:
LONGIE
High range chicken.
Range: Large
Damage: Normal
Price:


POOPIE
Does not attack, but its poop slows down enemies.
Range: Normal
Damage: Normal
Price:
FARTIE
Its farts explode and make fire damage on enemies.
Range: Large
Damage: Normal
Price:



Ao colocar uma galinha no campo de batalha, uma grid automaticamente aparecerá sinalizando lugares que a galinha pode ser colocada. Quando o cursor se encontra sobre um lugar inválido, a cor que seleciona o local se tornará vermelha, e se o usuário decidir mesmo assim colocar uma galinha naquela posição, nada irá acontecer. Para adicionar essas galinhas no jogo há um custo de milhos, a moeda que utilizamos. Caso o jogador não tenha milhos suficientes para colocar uma galinha, também nada acontecerá.

O contador no canto superior esquerdo da tela do jogo mostra a quantidade de milhos acumulada. Para conseguir mais milhos, é preciso derrotar inimigos, cada um dando uma quantidade específica dessa moeda. Abaixo do contador de milhos há o contador de ovos, que mostra de forma simples o status da base a ser protegida.

Quando há zero ovos na base, o jogo acaba. O ninho com o número de ovos atuais pode ser visto no mapa e, quando um ovo é comido por um inimigo, o ninho é alterado para aparecer exatamente o número de ovos restantes.

Outra funcionalidade implementada foi melhoramentos para as galinhas. Conseguimos aumentar o alcance, ataque e velocidade de ataque de cada uma delas por uma quantidade específica de milhos. Para isso criamos três símbolos para representar cada um dos melhoramentos. O óculos representa um alcance maior, o símbolo da “super galinha” representa maior poder de ataque e o simbolo do raio representa uma melhor velocidade de ataque. Quando comprado, cada melhoramento terá uma representação gráfica na galinha, por exemplo, quando comprado um alcance maior a galinha estará usando óculos. Para acessar a janela de melhoramentos, basta clicar em cima da galinha que deseja obtê-los. Para fechá-la, basta clicar novamente na mesma galinha.

2 Linguagem : Javascript

Javascript é uma linguagem de programação dinâmica, utilizada mais frequentemente em aplicações web para executar no lado do cliente. A linguagem apresenta diversos aspectos que a classificam como uma linguagem multi-paradigma, contendo tipagem dinâmica fraca, programação orientada a protótipos, interpretação por scripts e funções de primeira ordem. Este conjunto de propriedades permite que Javascript seja programado em estilos funcionais, imperativos e orientado a objetos.

O fato de executar no lado do cliente permite tempos de latência de resposta muito melhores que se a entrada tivesse que ser enviada ao servidor para ser processada e retornada. Hoje em dia, praticamente todos os navegadores tem compatibilidade para rodar códigos em Javascript, o que torna a linguagem extremamente portável.

Uma das principais características de Javascript é a liberdade de programação. A linguagem mantém a checagem por erros mínima, sempre tentando interpretar o que o programador tentou fazer mais do que apontar por erros. Isso a torna, por um lado, uma linguagem com mais liberdade ao programador e, por outro, uma linguagem que dificulta a identificação dos erros.

Outro fator que ajuda a geração de erros no código é a falta de tipos explícitos na linguagem. A tipagem de Javascript é dinâmica e fraca, testando pelo tipo a cada atribuição. Uma forma de teste por tipo utilizada é a tipagem de Pato ou Duck Typing, que se refere à ideia de que se algo se comporta como um pato, ele deve ser tratado como um pato.

Javascript suporta bem a orientação a objetos, com herança prototípica e propriedades implementadas através de vetores associativos. Isto quer dizer que os objetos tem propriedades que são acessíveis através de nomes que são chaves de string. Os protótipos são onde características de objetos como métodos são desenvolvidos para serem comunicáveis através de herança.

3 Implementação

3.1 Recursos Necessários

3.1.1 Definição e uso de classes:

Javascript não tem classes no sentido estrito da palavra, mas existem prototypes, que permitem herança e podem simular muitas funcionalidades originadas de classes. Na nossa implementação utilizamos um padrão de desenvolvimento recomendado para programação orientada a objetos em Javascript. O padrão se encontra abaixo:

```
var ClassName = function(constructorParameters...)
    {
        this.classProperties = initialValues;
        constructorProcedures();
        ...
    }
    ClassName.prototype =
    {
        method1: function()
        {
            method1Procedures();
            ...
        }
        ...
    }
```

Trecho de código 1: Definição de classes em Javascript

Este padrão é usado tanto nas estruturas utilitárias quanto para representar as estruturas de dados, visto que sempre que possível escolhemos conceituar as abstrações de forma orientada a objeto.

```
// Standard Chicken
var Chicken = function (Xtile,Ytile,Index,gameContext)
{
    this.index = Index;
    this.gameContext = gameContext;
    this.x = Xtile;
    this.y = Ytile;
    this.lastAttack = 0;
    this.attackSpeed = 10;
    this.damage = 10;
    this.glassesExtraHeight = -15;
    this.glassesExtraWidth = 7;
    this.glasses = "oculos";
    this.sprite = gameContext.add.sprite((Xtile*64),((Ytile*64+5)),'normalP');
    this.rangeSprite = gameContext.add.graphics(0,0);
    this.range = 2*64;

    this.setSprite();
    this.setRange();
    this.cleanRange();
    this.initializeAttackEffect();
    this.setUpgradeButton();
}
Chicken.prototype =
```

```
{
    attack: function(enemy)
      {   /*Implementação da função*/},
    detectEnemies: function()
      {   /*Implementação da função*/},
    initializeAttackEffect: function()
      {   /*Implementação da função*/},
    setSprite: function()
  {   /*Implementação da função*/},
    setRange: function()
  {   /*Implementação da função*/},
    update: function()
  {   /*Implementação da função*/},
    setUpgradeButton: function()
  {   /*Implementação da função*/},
    showUpgradeWindow: function()
  {   /*Implementação da função*/},
    cleanUpgradeWindow: function()
  {   /*Implementação da função*/},
    showRange: function()
  {   /*Implementação da função*/},
    cleanRange: function()
{   /*Implementação da função*/},
};
```

Trecho de código 2: Construtor da estrutura Chickens com uma visão geral de seus métodos

Vemos no exemplo acima a estrutura Chickens, com seu construtor primeiro, inicializando dados e chamando os métodos necessários para configurar a criação de uma nova galinha. A seguir vemos os seus métodos (sem a implementação para economizar espaço) desenvolvidos dentro de seu protótipo. Isto permite a passagens destes por herança para objetos cujo protótipo o referencie em criação.

Este padrão é crucial no desenvolvimento de estados da biblioteca que utilizamos, uma vez que a Phaser organiza a invocação de métodos predeterminados dos estados (preload,create,update,shutdown,etc) de acordo com a inicialização, atualização e transição de estados. Está, portanto, implementado em todos os estados do Chicken Hendemption.

As estruturas de dados que representam entidades como Chickens e Enemies (e derivados) também utilizam deste padrão, sendo particularmente útil para descrever as características e funcionalidades destas entidades, asseverando a representatividade do modelo orientado a objetos.

3.1.2 Espaços de nomes:

Javascript tem clausuras léxicas, que propagam o escopo externo contendo não somente os nomes das variáveis, funções, objetos, etc utilizados, como também as suas referências, mesmo se chamado de um escopo fora daqueles em que foi criado. Isto é uma característica que foi muito útil em vários casos quando tínhamos que executar um código um determinado período tempo após sua invocação. Para isto foi usada a função setTimeout, que mantinha o escopo das **vars** que referenciava, mesmo executando mais tardiamente, fora do escopo de execução onde foi criada.

```
Poopie.prototype.attack = function(enemy)
{
    this.lastAttack = this.gameContext.game.time.now;
    enemy.enemy.speed = enemy.enemy.oldSpeed/4;
    this.explosion.alpha = 0.5;
    this.explosion.animations.play('pooping', 10, false);
```

```
7       var effect = this.explosion;
8       setTimeout(function()
9          {
10             enemy.enemy.speed = enemy.enemy.oldSpeed;
11             effect.alpha = 0;
12         },1500)
13  };
```

Trecho de código 3: Exemplo de espaço de nomes

Tomamos como exemplo o ataque sobrecarregado da Poopie, que usa a funço de timeout para fazer seu "efeito especial" desaparecer depois de um segundo e meio, assim como fazer que o inimigo afetado retome sua velocidade original. Note que precisamos salvar a referência ao efeito numa variável para que este seja acessível dentro da função de timeout, que vai ser executada fora do contexto da Poopie, incapaz de acessar suas propriedades, neste caso, **this.explosion**.

3.1.3 Encapsulamento:

Comumente se encontra código de Javascript com objetos cujas propriedades são públicas.No entanto, a linguagem tem a capacidade de definir getters e setters para suas propriedades. Getters e Setters são funções que são invocadas quando queremos atribuir um valor a uma propriedade (setter) ou queremos obter o valor atual desta (getter).

A contribuição destes para as linguagens modernas inclui não somente o encapsulamento, como a validez semântica obtida através das, somente então, disponíveis capacidades de verificação garantidas em todas atribuições da variável em questão.

Existem múltiplas formas de definir getters e setters em Javascript, abaixo estão algumas delas:

```
1   Object.defineProperty(BasicGame.game,"cornCounter", set: function(value)
2   {
3       if(value >= 0)
4           this.cornCounter = value;
5       else
6           this.cornCounter = 0;
7   });
8
9   BasicGame.Game.prototype =
10  {
11    set cornCounter(value)
12    {
13        if(value >= 0)
14            this.cornCounter = value;
15      else
16          this.cornCounter = 0;
17  },
18    ...
19  }
20
21  Object.__defineSetter__.call(BasicGame.Game.prototype, "cornCounter", function(
        value)
22  {
23      if(value>=0)
24          this.cornCounter = value;
25      else
26          this.cornCounter = 0;
```

```
27 });
```

Trecho de código 4: Três formas de definir um setter em Javascript

Acima podemos ver que os exemplos tentam definir uma função de setter para ser chamada quando uma atribuição é feita a cornCounter, uma váriavel usada no nosso código para contar o número de milhos disponíveis, que representam tanto munição quanto moeda.

Quando executado nenhum erro de sintaxe é detectado, porém, o código não roda corretamente. A razão disto é que pouquíssimos navegadores suportam setters e getters (`http://en.wikipedia.org/wiki/JavaScript#Vendor-specific_extensions`), o que nos levou a abandonar inteiramente este padrão, uma vez que um requisito fundamental do nosso tipo de aplicação e uma das motivações principais que nos levaram a escolher Javascript é justamente a disponibilidade abrangente, juntamente com a conveniência de acesso através do browser.

A nossa implementação incluiu algumas funções com o mesmo propósito de um set. Estas atribuem e inicializam corretamente aquelas propriedades cujas averiguações semânticas eram específicas, ou quando um procedimento específico era necessário.

```
1  Chicken.prototype =
2  {
3      ....
4      setSprite: function()
5      {
6          this.sprite.inputEnabled = true;
7          this.showingUpgradeWindow = false;
8          this.sprite.events.onInputDown.add(function()
9                  {
10                     if(this.showingUpgradeWindow == false)
11                         this.showUpgradeWindow();
12                     else
13                         this.cleanUpgradeWindow();
14                 },this);
15         this.sprite.events.onInputOver.add(this.showRange,this);
16         this.sprite.events.onInputOut.add(this.cleanRange,this);
17         this.gameContext.chickenLayers[this.y].add(this.sprite);
18     },
19     ....
20 }
```

Trecho de código 5: Função setSprite da estrutra Chicken

O código acima mostra a função setSprite da estrutura Chicken. Neste método atribuímos todas as propriedades necessárias para o sprite da galinha. Primeiro indicamos que o sprite aceita input do usuário (através do mouse), a seguir, definimos as respostas para cada tipo de input. Se o usuário clica no sprite da galinha, o menu de upgrades deve aparecer se estiver oculto, senão ele deve desaparecer. Se o usuário passar com o mouse por cima da galinha, um círculo deve aparecer em volta dela, indicando o alcance de seus ataques. Uma vez que o mouse deixe o sprite este círculo deve desaparecer.

Por fim adicionamos o contexto do sprite da galinha a um grupo de sprites de acordo com seu y (variável que representa o deslocamento vertical do tile no qual a galinha se encontra). Cada um desses grupos é uma camada que compartilha a prioridade de impressão na tela, ou seja, as imagens das galinhas nos tiles mais de baixo devem ficar sobre as mais de cima, dando ao usuário um sentimento de profundidade.

3.1.4 Construtores e Destrutores:

- **Construtores:**

  Em Javascript qualquer coisa pode ser um objeto, para que uma função seja um objeto, basta que esta tenha suas propriedades atribuídas. A partir desta condição podemos atribuir um protótipo a este objeto, adicionando métodos herdáveis. Quando criamos um novo objeto deste tipo (com a palavra reservada **new**, por exemplo) a função original que definimos é invocada, sendo, portanto, a função construtora do objeto.

  Construtores estão amplamente difundidos no nosso código, uma vez que estes fazem parte do padrão que utilizamos para programação em Javascript orientado a objetos, descrito previamente em Classes.

```
var Enemies = function (game, path, IndexEnemy, wave)
{
this.game = game;
this.gameContext = this.game.state.getCurrentState();
this.enemy.path = path;
this.enemy.indexEnemy = IndexEnemy;
this.wave = wave;
this.setSprite();
this.setAnim();
this.setCentre();
this.cost = 5;
}
```

Trecho de código 6: Construtor da estrutura Enemies

  O construtor da estrutura geral Enemies exibe um comportamento padrão de construtor, atribuindo propriedades com valores e referências importantes e executando métodos de inicialização.

- **Destrutores:**

  Javascript não tem destrutores no sentido de uma função que é invocada quando um objeto é destruído, se não existem referências a um objeto, ele simplesemente é recolhido. Uma função pode ser desenvolvida tal que ela delete um ou vários objetos, mas nada no nível de um destrutor existe em Javascript.

```
Map.prototype =
{
...
    cleanMap: function()
    {
        for(i=0;i<22*15;i++)
        {
                this.tiles[i].occupied = false;
                this.tiles[i].forbidden = false;
        }
    },
    ...
};
```

Trecho de código 7: Método de limpeza do mapa

  No Chicken Hendemption temos alguns métodos de clean up e de adequação de certas propriedades antes de sua destruição ou reutilização. No exemplo acima temos um o método

de clean up da estrutura Map, cujo propósito é guardar os tiles que estão atualmente ocupados por galinhas e os tiles que não permitem a instalação de nenhuma galinha (como os presentes no meio do caminho ou embaixo da interface de usuário). O método de cleanMap serve para apagar todos os resquícios do mapa para reutilização, o que é uma tarefa mais eficiente e mais responsável em termos de gerenciamento de memória do que a criação de um novo objeto mapa.

```
1 BasicGame.Game.prototype =
2 {
3     ...
4     shutdown: function()
5     {
6         this.stopMusic();
7     },
8     ...
9 }
```

Trecho de código 8: Método de shutdwon da estrutura de estado Game

Uma característica interessante do estrutura de estado fornecido pela Phaser é a função de shutdown, uma função que é chamada sempre **antes** da transição para outro estado, independente do escopo em que foi chamada a transição. Neste caso nós simplesmente fazemos a música de fundo parar de tocar, para não tocar juntamente com a música do próximo estado, seja qual for.

3.1.5 Herança:

O padrão que utilizamos (veja Classe, acima) permite a fácil implementação de herança através dos protótipos. O protótipo guarda a implementação de seus métodos e quando queremos que um objeto filho herde o protótipo de um objeto pai simplesmente fazemos:

```
1 Filho.prototype = Object.create(Pai.prototype);
```

Encontramos esta linha de código diversas vezes no nosso trabalho, mas especialmente no arquivo Chickens.js, que implementa todas as características das galinhas do jogo. Note aqui que temos múltiplos casos de herança por protótipo e múltiplos casos de polimorfismo por sobrecarga em cada implementação da função attack, por exemplo.

```
1 // Standard Chicken
2 var Chicken = function (Xtile,Ytile,Index,gameContext)
3 { /* Função constutora de Chicken */ }
4 Chicken.prototype =
5 { /* Definição dos métodos de Chicken */ };
```

Trecho de código 9: Galinha padrão

A Standard Chicken é uma classe básica, com a implementação de todos os métodos necessários para o funcionamento básico de uma galinha de defesa. Todas as outras galinhas herdam da galinha padrão (direta ou indiretamente), fazendo uso da maioria de seus métodos. Nota-se que o construtor não é herdado, uma vez que a função usada para transmitir o protótipo Object.create, então no construtor da galinha padrão temos apenas o necessário para fazer a galinha Normal funcionar, ou seja, uma galinha cujos ataques não geram projéteis, não tem efeitos colaterais nem forma especial de detectar ou danificar inimigos.

```
1  //AOE Chicken
2  //Stands for Area Of Effect, these chickens have a modified detectEnemies
       functions that targets all enemies in range
3  //Inherits from Standard Chicken
4  var AOEChicken = function (Xtile,Ytile,Index,gameContext)
5  { /* Função constutora de AOEChicken */ }
6  AOEChicken.prototype = Object.create(Chicken.prototype);
7  AOEChicken.prototype.detectEnemies = function()
8  { /* Sobrecarga do método herdado de Chicken: detectEnemies */};
```

Trecho de código 10: MGalinha de efeito em Área

A seguir temos a AOEChicken, que herda da Chicken, mas esta implementa uma forma diferente de detectar inimigos. Enquanto que a Chicken detecta um inimigo e o ataca, a AOEChicken, cujo nome significa Area of Effect Chicken, ataca todos os inimigos que detectar, como o nome indica. Esta é uma classe abstrata, não utilizada especificamente em nenhum momento do jogo, apenas servindo de base para outras galinhas que a herdam e especializam.

```
1  //Longie
2  //Longer range, less damage, spends corn
3  //Inherits from Standard Chicken
4  var Longie = function (Xtile,Ytile,Index,gameContext)
5  { /* Função constutora da Longie */ }
6  Longie.prototype = Object.create(Chicken.prototype);
7  Longie.prototype.attack = function(enemy)
8  { /* Sobrecarga do método herdado de Chicken: attack */};
```

Trecho de código 11: Galinha de alto alcance

A Longie herda da Chicken, especializando seus ataques para que estes sejam enviados como tiros de milhos. Para tanto precisamos sobrecarregar o método attack herdado para que este crie uma nova bullet, que é gerenciada na estrutura de estado Game. Uma vez criada, esta bullet é independente da galinha que a originou, tendo seu próprio destino, velocidade e dano, todos imbuídos pela Longie. A função construtora que inicializa suas propriedades fornece um valor de dano menor para esta galinha, com um alcance maior, para compensar e ser condizente com o tipo de ataque.

```
1  //Poopie
2  //Slows enemies, doesn't deal damage
3  //Inherits from Area of Effect Chicken
4  var Poopie = function (Xtile,Ytile,Index,gameContext)
5  { /* Função constutora da Poopie */ }
6  Poopie.prototype = Object.create(AOEChicken.prototype);
7  Poopie.prototype.initializeExplosion = function()
8  { /* Implementação de um novo método: initializeExplosion */};
9  Poopie.prototype.attack = function(enemy)
10 { /* Sobrecarga do método herdado de Chicken: attack */};
```

Trecho de código 12: Galinha de lentidão

A Poopie herda da AOEChicken (fazendo dela parte do 3º nível de herança), afetando todos os inimigos em seu alcance. Esta galinha serve para desacelerar os inimigos com o seu excremento que se espalha pelo chão, dando mais tempo para as outras galinhas defenderem o galinheiro. Para isso a galinha tem sobrecarregar a função de ataque, que diminui a velocidade de movimento dos inimigos. Ela também deve ter uma função para inicializar seu efeito gráfico de ataque, para que não tenha que ser recriado a cada ataque, apenas tornado visível.

```
//Fartie
//Deals damage to all enemies in range
//Inherits from Area of Effect Chicken
var Fartie = function (Xtile,Ytile,Index,gameContext)
{ /* Função constutora da Fartie */ }
Fartie.prototype = Object.create(AOEChicken.prototype);
Fartie.prototype.initializeExplosion = function()
{ /* Implementação de um novo método: initializeExplosion */};
Fartie.prototype.attack = function(enemy)
{ /* Sobrecarga do método herdado de Chicken: attack */};
```

Trecho de código 13: Galinha de dano em área

A Fartie é a segunda galinha que herda da galinha de efeito em área e também faz parte do terceiro nível de herança. Esta galinha incendeia sua flatulência e causa dano a todos os inimigos à sua volta. Para tanto precisamos sobrecarregar a função de ataque apenas para chamar o efeito especial da explosão, que deve ser inicializado antes e deve ter um tempo de desvanecimento. O ataque também pretendia incendiar os inimigos e fazê-los levar dano com o tempo, mas devido à forma como os inimigos são mortos este aspecto ficou para versões futuras.

```
//Robot Chicken
//Shoots a laser that damages all enemies in a line
//Inherits from Standard Chicken
var Robot = function (Xtile,Ytile,Index,gameContext)
Robot.prototype = Object.create(Chicken.prototype);
Robot.prototype.initializeLaser = function()
{ /* Implementação de um novo método: initializeLaser */};
Robot.prototype.attack = function(enemy)
{ /* Sobrecarga do método herdado de Chicken: attack */};
```

Trecho de código 14: Galinha robô com ataque especial

A última e mais cara galinha é a galinha robô, que ataca com lasers que saem de sua boca. A Robot herda da Chicken normal, não mirando em múltiplos inimigos, mas seu ataque é especial e ela pode (e geralmente vai) causar dano a múltiplos inimigos. O seu ataque é um laser que atravessa o mapa e causa dano a todos inimigos que estiverem em contato. O laser deve escolher apenas um inimigo e mirar o laser nele, por isso herdamos o detectEnemies da Chicken. Precisamos de uma função para inicializar a animação e precisamos sobrecarregar a função de ataque para que este posicione o laser e verifique todos os inimigos que estão no caminho para que estes levem o dano. O ataque também deve colocar um timeout para desvanecer o efeito do laser.

Nota-se também que a função construtora está implementada em todas as galinhas, uma vez que esta não é propagada para as galinhas herdeiras. Mesmo que fossem propagados, a implementação de cada construtor é necessária pois estes que definem as características únicas de cada galinha, como imagem, dano, velocidade de ataque, alcance, etc.

3.1.6 Polimorfismo por Inclusão:

Podemos observar polimorfismo por inclusão no Game.js, na função chickenUpdate(), que atualiza todas as galinhas chamando a função de update interna da galinha que detecta e ataca os inimigos de acordo com a sua função especializada e não a versão original definida na classe geral (Chicken). Esta vinculação dinâmica é uma flexibilidade muito facilitadora da linguagem.

```
chickenUpdate: function()
    {
```

```
3         for(var i=0;i<this.chickens.length;i++)
4         {
5             this.chickens[i].update();
6         }
7     },
```

Trecho de código 15: Método de atualização de galinhas

3.1.7 Polimorfismo paramétrico:

Polimorfismo paramétrico não faz sentido em linguagens dinamicamente tipadas como Javascript. Javascript tem o conceito de Duck Typing, ou seja, tipagem de pato, que referencia a ideia de "Se encontramos um pássaro que anda como um pato, nada como um pato e grasna como um pato, chamamos este pássaro de pato". Considerando que o polimorfismo paramétrico serve para aumentar a expressividade de um tipo estático, não faz sentido aplicá-lo a linguagens com tipagem dinâmica, fraca e implícita como Javascript.

3.1.8 Currying:

Currying é a uma técnica na qual traduzimos a avaliação de uma função com múltiplos parâmetros para diversas aplicações com um parâmetro cada. A partir desta descrição e algum conhecimento de funções em Javascript podemos perceber que é possível implementar Currying em Javascript, uma vez que podemos fazer facilmente uma função retornar outra função. Inclusive é possível construir uma função que opera sobre a função na qual queremos aplicar Currying. A seguir uma possível implementação desta função. (`http://www.crockford.com/javascript/www_svendtofte_com/code/curried_javascript/index.html`)

```
1  function curry(func,args,space) {
2      var n  = func.length - args.length; //arguments still to come
3      var sa = Array.prototype.slice.apply(args); // saved accumulator array
4      function accumulator(moreArgs,sa,n) {
5          var saPrev = sa.slice(0); // to reset
6          var nPrev  = n; // to reset
7          for(var i=0;i<moreArgs.length;i++,n--) {
8              sa[sa.length] = moreArgs[i];
9          }
10         if ((n-moreArgs.length)<=0) {
11             var res = func.apply(space,sa);
12             // reset vars, so curried function can be applied to new params.
13             sa = saPrev;
14             n  = nPrev;
15             return res;
16         } else {
17             return function (){
18                 // arguments are params, so closure bussiness is avoided.
19                 return accumulator(arguments,sa.slice(0),n);
20             }
21         }
22     }
23     return accumulator([],sa,n);
24 }
```

Trecho de código 16: Função genérica curry

Currying é interessante para aplicações cujas funções tenham múltiplos argumentos e estes habitem áreas de código diferente, onde uma aplicação de um desses em currying seja passada para outro. Não conseguimos achar nenhuma forma não extremamente forçada de aplicar currying. No interesse de manter um padrão de programação escolhemos não incluir currying no código, também em parte por sua legibilidade confusa.

3.1.9 Pattern Matching:

No sentido em que pattern matching aplica-se à decisão entre funções/estruturas candidatas polimórficas, este não se aplica a Javascript, uma vez que sua tipagem dinâmica e fraca não consegue verificar os tipos para decidir a candidata. Esta característica da linguagem permite várias situações flexíveis mas enfraquece a checagem.

3.1.10 Delegates:

Entre os conceitos associados com delegates na programação orientada a objetos está a ideia de atribuir dinamicamente a um método a um objeto. A versão 1.8.5 do Javascript introduziu Function.prototype.bind(), que implementa justamente isto, atribuindo o valor do objeto às ocorrências do termo **this** na função.

```
1  setUpgradeButton: function()
2  {
3          this.speedUpgraded = false;
4          this.rangeUpgraded = false;
5          this.damageUpgraded = false;
6          var upRange = UpgradeRange.bind(this);
7          var upSpeed = UpgradeSpeed.bind(this);
8          var upDamage = UpgradeDamage.bind(this);
9
10         this.upgradeWindow = this.gameContext.add.sprite(-200,-200,'upgradeMenu');
11         this.upgradeWindow.inputEnabled = true;
12
13         this.upgradeRangeButton = this.gameContext.add.sprite(-200,-200,'oculos');
14         this.upgradeRangeButton.inputEnabled = true;
15         this.upgradeRangeButton.events.onInputDown.add(function()
16         {
17             upRange(this);
18         },this);
19
20         this.upgradeDamageButton = this.gameContext.add.sprite(-200,-200,'super');
21         this.upgradeDamageButton.inputEnabled = true;
22         this.upgradeDamageButton.events.onInputDown.add(function()
23         {
24             upDamage(this);
25         },this);
26
27         this.upgradeSpeedButton = this.gameContext.add.sprite(-200,-200,'speed');
28         this.upgradeSpeedButton.inputEnabled = true;
29         this.upgradeSpeedButton.events.onInputDown.add(function()
30         {
31             upSpeed(this)
32         },this);
33 },
```

Trecho de código 17: Método setUpgradeButton da estrutura Chicken

A função setUpgradeButton é o único exemplo de aplicação da função bind, uma vez que seria possível implementar as mesmas funcionalidades com o mesmo padrão que seguimos até agora e fazemos questão de mantê-lo. Este método simplesmente associa as funções de upgrade aos seus respectivos botões, após serem vinculados ao objeto chamador.

O poder que este padrão fornece é reproduzível com outras ferramentas de linguagem, mas certamente pode ser bem mais expressivo que os métodos tradicionais em certas ocasiões. A capacidade de atribuir métodos a objetos desconexos é muito poderosa, porém provavelmente poderia gerar confusão e códigos menos seguros, devido ao fato de que se os objetos e métodos não foram feitos um para o outro, é possível que estes tenham falsas expectativas.

É interessante notar que a sintaxe desse padrão, apesar de não adotada por nós ou frequentemente encontrada em implementações de Javascript, tem uma legibilidade decente, diferente de outros padrões de linguagens funcionais aqui descritos (como currying). Claro que devemos levar em consideração que este padrão tem uma implementação embutida na linguagem, enquanto que precisamos implementar nossas próprias funções com currying.

3.1.11 Funções de múltiplas ordens:

Uma função é tratada como um elemento de primeira ordem quando esta é tratada como uma variável qualquer, seja passando para uma função como um argumento, retornando como resultado de outra função ou guardando em uma estrutura de dados. Funções que utilizam outras funções como elementos de primeira ordem são chamadas de funções de ordem maior. Ambos estes conceitos são fundamentais nas linguagens funcionais, portanto não é surpresa que estão presentes em Javascript.

O exemplo mais frequente da utilização de funções como elementos de ordens variadas utilizado no código do Chicken Hendemption é a atribuição de uma função como resposta padronizada a um tipo de evento. Este tipo de capacidade é o fundamento da programação orientada a eventos e é uma forma altamente intuitiva de tratar o problema de resposta à entrada do usuário.

```
1  initializeInterface: function()
2  {
3          var money = this.add.sprite(BasicGame.convertWidth(0),BasicGame.
               convertHeight(0),'counter');
4          money.bringToTop();
5
6          this.inGameOpt = new InGameOptionsPanel(this);
7          this.game.add.existing(this.inGameOpt);
8          //paused = false;
9          opt = this.add.sprite(BasicGame.convertWidth(453),BasicGame.convertHeight
               (5),'opt');
10         opt.inputEnabled = true;
11         opt.bringToTop();
12         opt.events.onInputDown.add(function()
13         {
14             this.clickButtonSound.play();
15             opt.loadTexture('opt_pressed',0);
16         },this);
17         opt.events.onInputUp.add(this.pauseGame,this);
18         BasicGame.optionsPanel.game = this.game;
19         BasicGame.optionsPanel = new OptionsPanel(this);
20         this.game.add.existing(BasicGame.optionsPanel);
```

```
21 },
```

Trecho de código 18: Método initializeInterface da estrutura de estado Game

Vemos no exemplo acima que uma função não nomeada está sendo passada como argumento para ser executada quando o usuário apertar o botão para baixo, tocando um som e recarregando uma textura apropriada para um botão pressionado. Após a liberação do botão chamamos a função de pausa do jogo, o propósito do botão. Este tipo de uso de funções não nomeadas como elementos de primeira ordem, sendo manipuladas em funções de ordem maior é muito comum no nosso código. É um uso superficial, mas aprofundar na complexidade de utilização de funções de ordem maior seria desnecessário e confuso. E a geração de um código padronizado, legível e moldável foi o valor guardado com a maior estima.

A possibilidade de manipular funções de um lado a outro, retornar uma função ou atribuir um função como resposta a um evento é uma ferramenta muito poderosa para desenvolvedores, facilmente aplicável a um a um conjunto abrangente de atividades. A sintaxe utilizada em Javascript para desenvolver tais propósitos é intuitiva, pelo tratamento de funções como verdadeiros elementos de primeira ordem.

3.1.12 Listas e Recursão:

Javascript tem uma implementação de vetores fortemente difundida e muito bem desenvolvida. Listas são coisas raras em Javascript, uma vez que a flexibilidade notoriamente atribuída a listas também se encontra na implementação de vetores de Javascript. A aplicação de manipulação de listas, especificamente dentro de funções de ordem maior, não serviu algum propósito verdadeiro dentro da nossa aplicação. Em parte por que as funções de ordem maior são utilizadas superficialmente.

A recursão como um modo de iteração tem a mesma característica, flexibilidade a custo de desempenho. Faz sentido utilizar recursão para manusear listas em funções de ordem maior.

No entanto, é difícil imaginar um jogo que utilize funções de ordem maior manipulando funções de 1ª ordem com uma complexidade que necessite a utilização de recursos como listas, recursão e currying. E de fato, não utilizamos estes três últimos recursos de linguagens funcionais. Apesar de possível implementar as aplicações sugeridas com várias características funcionais (e, muitas vezes, mais fácil com estas). Uma aplicação que necessite tais recursos aplicados profundamente requer uma complexidade funcional maior, como um analisador de expressões regulares, ou um interpretador de uma linguagem fictícia bem ortogonal.

4 Analise Critica

Simplicidade	8.5	Os conceitos básicos de Javascript são fáceis de entender e utilizar.
Ortogonalidade	7.0	Existem múltiplas formas de realizar as mesmas operações, cada uma com funcionalidades levemente diferentes e diversos construtos da linguagem possuem comportamentos que variam conforme o tipo.
Expressividade	8.5	Poucas linhas de código podem representar complexas operações, especialmente a respeito dos aspectos funcionais.
Adequabilidade e Variedade de estruturas de controle	9.0	Além das mesmas estruturas de controle de C (inclusive com a mesma sintaxe), JS tem suporte a estruturas como for each.
Mecanismos de definição de tipos	*	Tipagem dinâmica e fraca. Métodos de objetos são herdados no protótipo.
Suporte a abstração de dados e de processos	6.0	Capacidade de abstração, mas nada enforçado. Suporte a delegates e encapsulamento por getters/setters, porém estes padrões muitas vezes nem são suportados pela plataforma.
Modelo de tipos	*	Tipagem dinâmica e fraca.
Portabilidade	9.5	Muito portável (suporte ao navegador).
Reusabilidade	7.0	Potencial para reusabilidade, requer programação genérica, dificultado pelo escopo léxico implementado. A implementação de módulos não está bem determinada ainda.
Suporte e documentação	8.0	A documentação é vasta, mas muitos browsers não suportam features específicas.
Tamanho de código	8.8	Relativamente pequeno, com fácil definição de estruturas como vetores, conjuntos e outros.
Generalidade	7.0	Delegates permitem que, no nível de métodos, a aplicação seja mais genérica, porém não há garantias de funcionamento ou de detecção de erro.
Eficiência	7.5	Considerando a plataforma de aplicação, a capacidade de executar código no lado do cliente é muito mais eficiente do que enviar o código ao servidor e ter que sofrer a latência. Ainda seria uma opção carregar um código java para ser executado no cliente, que seria mais eficiente [1]. Javascript geralmente é usado para facilitar este tipo de coisa (validação, ajax, etc).
Custo	9.0	Plataforma de desenvolvimento e documentação amplamente disponíveis.
Confiabilidade	7.0	A utilização de um interpretador sempre permite maior flexibilidade a um custo de confiabilidade.

Conclusões

Ao desenvolver o jogo, a principal dificuldade que encontramos foi o encapsulamento no código em Javascript. Todas as variáveis de todas as classes são visíveis por todas as outras classes do projeto. Outro grande problema que tivemos no desenvolvimento foi a falta de precisão na detecção de erros. Muitas vezes o erro apresentado no console não era o verdadeiro problema, muitas vezes apontando para um trecho de código incompreensível dentro do código do Framework. Este é um dos problemas de se programar em uma linguagem interpretada em relação a uma compilada. O compilador não só detecta os erros sintáticos na hora de compilação como detecta os erros mais semânticos como os de tipos.

A parte gráfica do jogo não trouxe muitas dificuldades, já que o Framework escolhido para o desenvolvimento serve perfeitamente para o nosso caso de uso no projeto. As únicas dificuldades foram na parte da sintaxe, já que, por ser um framework muito atualizado, haviam modos diferentes descritos na internet para a mesma funcionalidade, pertencentes e versões diferentes do "Phaser". Por isso boa parte do desenvolvimento foram tentativas de diversas sintaxes até encontrar o modo correto de operar a funcionalidade. Também, por ser um Framework de alto nível, muitas coisas simples que desejávamos fazer se tornaram mais difíceis do que originalmente achávamos. Por exemplo, desenhar linhas na tela para formar a grid (na hora de adicionar novas galinhas ao jogo) se tornou uma tarefa difícil pelo Framework não possuir um suporte amplo na área de formas simples. Por isso foi necessário buscar funções para isso na biblioteca gráfica que a "Phaser" herda, chamada "PIXI", também muito conhecida no ramo gráfico de Javascript.

O plano inicial para o jogo era fazer ele totalmente compatível tanto para dispositivos mobile quanto para desktop. Não conseguimos implementar isso a tempo por alguns problemas no tratamento do input pela biblioteca. Também para o funcionamento em telas menores precisávamos adaptar todas as imagens utilizadas no jogo (atualmente 102 imagens) para um tamanho adequado (58 delas já foram passadas para todos os 4 outros diferentes tamanhos de resolução previstos pelo projeto). Isso demandaria muito tempo, por isso não ficará pronto até o prazo de entrega do trabalho mas será desenvolvido ao longo do tempo livre dos desenvolvedores.

Conseguimos deixar este trabalho com um grande conhecimento adquirido tanto da biblioteca Phaser quanto de Javascript com suas características peculiares de múltiplos paradigmas. Aprendemos sobre a utilidade de diversos aspectos funcionais que Javascript traz consigo, no contexto de uma aplicação real. Assim como

Referencias

Javascript Game Programming Using Phaser.
Disponível em: http://www.sitepoint.com/javascript-game-programming-using-phaser/
Acesso em: 13 de novembro de 2014.

How do Organize your Javascript Code into Classes Using Phaser HTML5 Game Framework.
Disponível em: http://toastedware.com/?p=258
Acesso em: 13 de novembro de 2014.

Phaser Framework.
Disponível em: http://phaser.io/
Acesso em: 13 de novembro de 2014.

PIXI.js.
Disponível em: http://www.pixijs.com/
Acesso em: 13 de novembro de 2014.

Eloquent Javascript - High-Order Functions. Disponível em: http://eloquentjavascript.net/05_higher_order.html
Acesso em: 13 de novembro de 2014.

http://en.wikipedia.org/wiki/Anonymous_function

Curried Javascript Functions.
Disponível em: http://www.crockford.com/javascript/www_svendtofte_com/code/curried_javascript/index.html
Acesso em: 13 de novembro de 2014.

Javascript V8 vs Java | Computer Language Benchmarks Game.
Disponível em: http://benchmarksgame.alioth.debian.org/u32/benchmark.php?test=all&lang=v8&lang2=java&data=u32
Acesso em: 13 de novembro de 2014.