Ivo Matheus de Goes Lopes

Um Motor de Jogos para Autoria de Aplicações de Hipermídia com NCLua

São Luís

2015

Ivo Matheus de Goes Lopes

Um Motor de Jogos para Autoria de Aplicações de Hipermídia com NCLua

Uma experiência com Motores de Jogos com o fim de incentivar trabalhos com Ginga.

Universidade Federal do Maranhão – UFMA

Curso de Ciências da Computação

Graduação

São Luís

2015

Lista de ilustrações

Lista de tabelas

Lista de quadros

Lista de códigos

Lista de abreviaturas e siglas

GW	Ginga Wings
OO	Programação Orientada a Objetos
SRS	Super Rotation System

Sumário

1 Introdução

2 Trabalhos Relacionados

3 Engine Ginga Wings

3.1 Arquitetura de Implementação

Uma Engine de Jogos é um Software com o propósito de facilitar o desenvolvimento de jogos. Elas foram, a princípio, criadas para facilitar o desenvolvimento com estrutura similares – como por exemplo, a série de jogos de RPG intitulada Baldur's Gate, da BioWare, que se utilizou da Infinity Engine –, deixando para o desenvolvedor apenas a preocupação de criar o jogo em si, e não perder metade do prazo do jogo construindo seu alicerce. A popularidade desse tipo de aplicação aumentou nos anos 90, com o aparecimento de jogos "mods" (nome dado a modificações de partes de um jogo, mas mantendo boa parte da estrutura) do famoso jogo Doom, criado pela Id Software. Isso foi possível graças à separação que a equipe fez entre os componentes do jogo – deixando bem definidos os códigos que controlavam o renderizador gráfico, a colisão de objetos, o sistema de áudio, os recursos de mídia e imagem, as regras de jogo, etc. -, o que permitiu que outros grupos de desenvolvedores modificassem apenas as partes que lhes interessava (basicamente os recursos de áudio e imagens, arquivos de armas, inimigos, fases, etc), utilizando-se de Toolkits liberados pela empresa. Hoje temos Engines bem mais genéricas do que as dos anos 90, que permitiam criar jogos de Doom. Engines como Unreal ou Unity podem ser utilizadas e modificadas para criar uma vasta variedade de jogos para várias plataformas, 3D e 2D. A Engine de Jogos Ginga Wings vem com uma proposta de trazer semelhante funcionalidade para aplicações de TV Digital que rodam o Middleware Ginga.

A GW é um Engine especializada em jogos com gráficos 2D e interação pelo controle remoto. Ela se utiliza de boa parte da API de NCLua - descrita pelo padrão de Sistema Brasileiro de TV Digital (ABNT 2007) – para prover funcionalidades através do Set-Top Box. Ela roda, inteiramente, com código Lua e se comunica com o Documento NCL nativamente, sendo essa uma de suas principais características. Apesar do desenvolvimento ser voltado para Set-Top Boxes, como o Ginga também roda em dispositivos móveis, a Engine pode ser utilizada para jogos nestes dispositivos.

A visão geral da arquitetura da engine pode ser observada na Figura 1, com detalhamentos nas próximas seções.

3.1.1 Hardware

Hardware é onde a Aplicação está sendo executada. Geralmente será um Set-Top Box, mas pode ser um Dispositivo Móvel. Não é necessário detalhamento uma vez que a função do S.O. e do Middleware Ginga é deixar o tratamento do Hardware transparente.

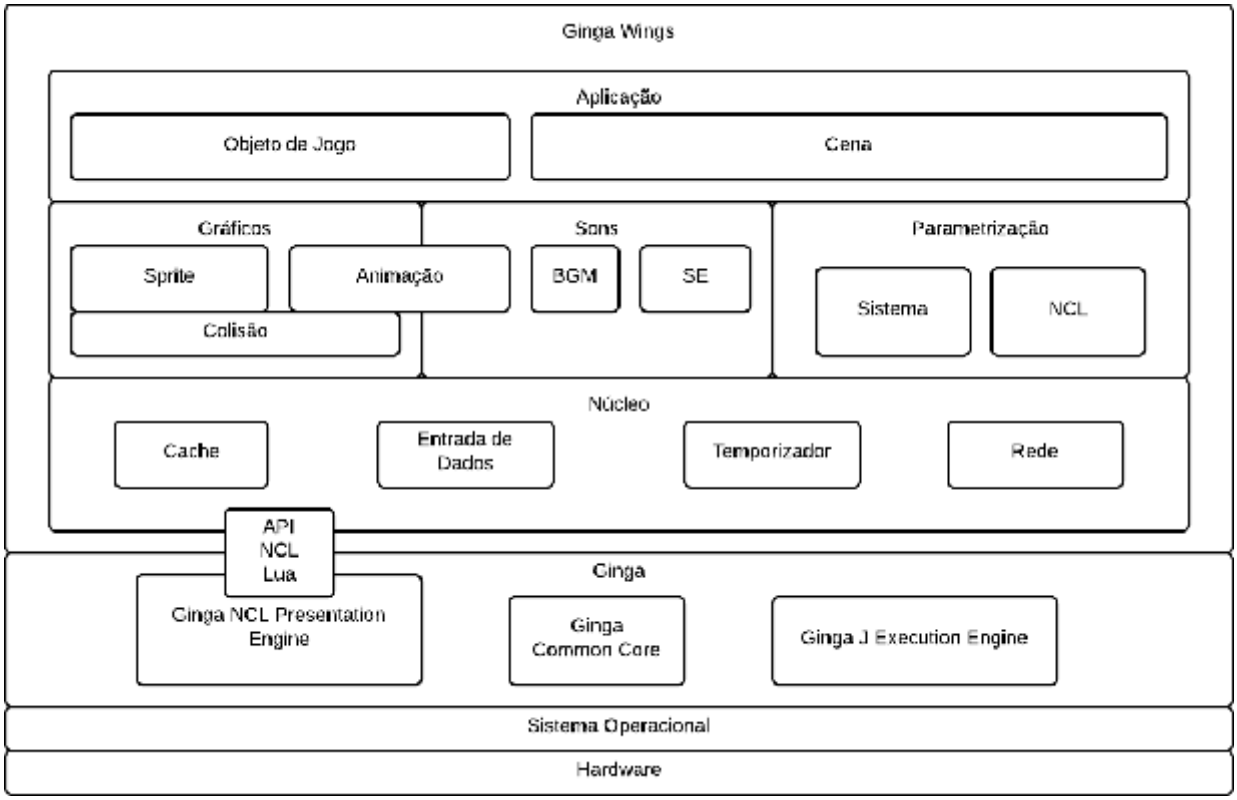

Figura 1 – Arquitetura Ginga Wings

3.1.2 Sistema Operacional

O Sistema Operacional é o principal sistema executado no Hardware, na maioria dos dispositivos. Normalmente em PCs, o SO se ocupa de gerenciar os recursos de Hardware, distribuindo-os para as aplicações. Por isso, em um PC, é de se esperar que nenhuma aplicação tenha total posse dos recursos de Hardware. Dispositivos Móveis e Set-Top Boxes, por outro lado, por terem uma função definida (contraposta ao SO de PC, que tem propósito geral), geralmente alocam praticamente todos os recursos à aplicação. A implementação de referência utiliza o Linux como Sistema Operacional padrão, pelo fato do mesmo ser um Software Livre.

3.1.3 Ginga

O middleware é a camada de software que executa entre o sistema operacional e as aplicações. No caso específico do Sistema Brasileiro de TV Digital (SBTVD), o middleware Ginga foi desenvolvido. A Figura 2 apresenta uma visão geral da organização do Ginga.

Como pode ser visto, ele é formado por diversos subsistemas, os quais foram implementados usando a linguagem C++. O núcleo comum do Ginga (CommonCore) implementa as funcionalidades principais do middleware, e acima dele temos uma camada de serviços específicos que consiste na implementação das APIs utilizadas pelas aplicações que executam sobre o middleware. Dois tipos de aplicações são suportadas pelo middleware

Figura 2 – Visão Geral do Middleware Ginga

brasileiro, as aplicações NCL (Ginga-NCL) e as aplicações Java (Ginga-J). Neste cenário, as aplicações acessam suas APIs específicas e estas APIs utilizam o núcleo comum para efetuarem suas operações.

Em especial, ignoramos a Máquina de Execução em Ginga-J, e consideramos somente a de Apresentação, Ginga-NCL. Esta, em especial, possui o que chamamos de Objetos Imperativos NCLua, o que nada mais é do que a comunicação nativa existente entre o NCL e a Linguagem de scripting Lua. A criação, a ativação, pausa e encerramento do objeto NCLua são todos gerenciados a partir do NCL.

O Middleware disponibiliza uma API que expõe métodos úteis para que o código Lua possa se utilizar dos recursos do sistema, tais como métodos para acesso aos pixels da tela reservada ao script, envio e recebimento de eventos e nós NCL, envio e recebimento de pacotes de rede pela internet, captura de entrada de dados da parte do usuário, entre outros. A Engine se baseia fortemente nestes métodos expostos para a implementação de suas funções.

3.1.4 GingaWings

Por fim, chegamos ao componente Ginga Wings. O primeiro nível da arquitetura é o grupo Núcleo, que possui os componentes Cache, Entrada de Dados, Temporizador e Rede. O componente Cache é responsável pelo gerenciamento básico de memória para sons, músicas e imagens, garantindo que um recurso não será várias vezes carregado na memória, e que recursos não utilizados pelo jogo sejam devidamente descartados da memória. O componente Entrada de Dados fornece várias formas de detecção de entrada do usuário, enriquecendo a API originada do Ginga. O componente Temporizador abusa da API de eventos do Ginga para garantir que o sistema será executado sempre em tempo constante.

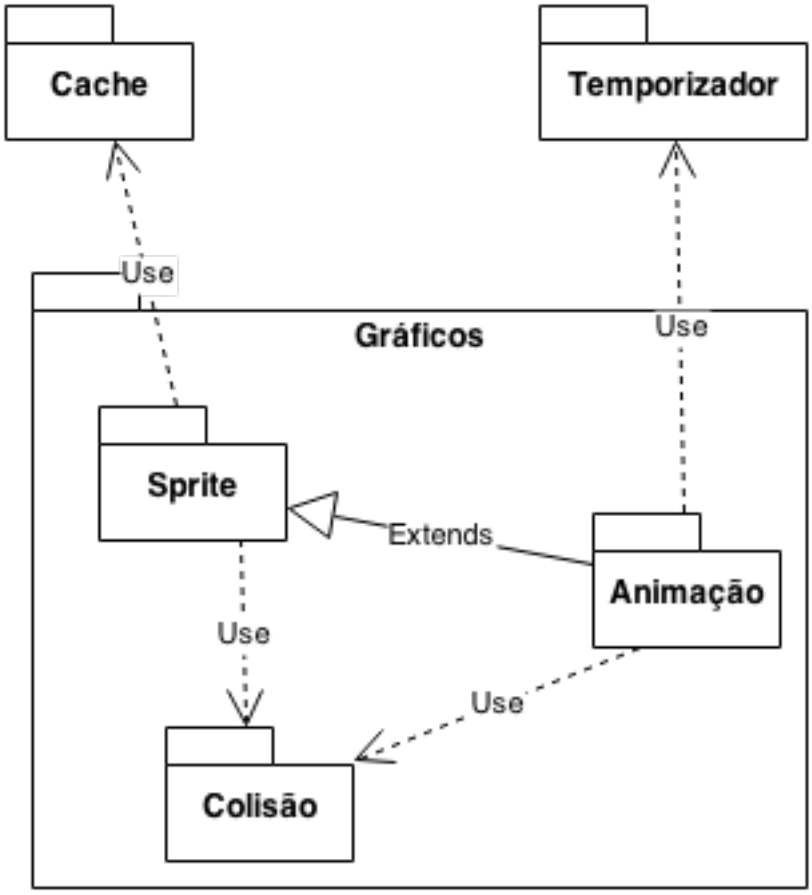

Figura 3 – Diagrama de Pacotes. Ginga Wings – Gráficos

E por fim o componente de Rede oferece uma forma amigável de se enviar e receber pacotes via web.

3.1.4.1 Gráficos

O componente Gráficos pode ser detalhado pela Figura 3, na forma de pacotes. A estrutura básica do pacote é o Sprite. O Sprite é uma representação da posição, alargamento (zoom), e rotação de uma imagem, mas não é uma imagem em si. A imagem fica armazenada no Cache, de quem o Sprite depende. Um Sprite pode ter um objeto de Colisão associado a ele. A GW automaticamente detecta esta colisão e dispara eventos de caso este Sprite colida com outros. O pacote Animação é uma sequência de Sprites, gerenciada pelo Temporizador. Ela pode ter seu próprio objeto de colisão à parte do Sprite. A animação geralmente apenas muda qual Sprite está sendo exibido no momento, sendo a mudança gerenciada por um tempo t, o qual o Temporizador pode oferecer. A Animação também possui sua própria localização, alargamento e rotação à parte dos seus sprites.

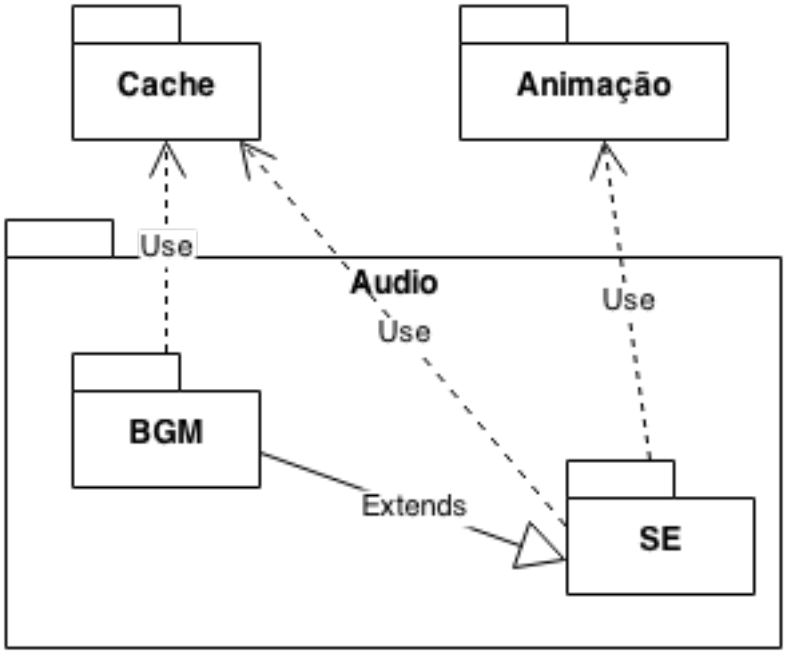

Figura 4 – Diagrama de Pacotes. Ginga Wings – Audio

3.1.4.2 Audio

O componente Audio, conforme ilustrado na Figura 4, também possui dependência com o Cache. Diferentemente do que ocorre com imagens, não há uma exposição de métodos para manipulação de sons diretamente. Por isso é preciso utilizar-se de eventos NCLua para que algum som seja tocado através do Documento NCL, não do código Lua. O objetivo do Cache neste caso é gerar estes eventos. Mais detalhes na seção de API. Existem dois tipos de Áudio, BGM e SE. BGM signifca Backgroung Music, ou música de fundo. Basicamente são músicas que tocam em loop infinito enquanto o jogador manipula o jogo. SE significa Sound Effects, ou Efeitos Sonoros, que são podem ser descritos como Onomatopeias na gramática. As SEs são utilizadas pelas Animações para tocar sons sincronizados.

3.1.4.3 Parametrização

O componente Parametrização é uma das principais características do sistema: permite que o usuário de um Documento NCL envie parâmetros para o código Lua, configurando e controlando o jogo a partir do próprio NCL. Utilizando-se de estruturas como Âncoras e Links, é possível criar uma Engine Básica de jogos e orientá-la totalmente pelo seu código NCL: por exemplo, é possível definir uma fase inteira de Super Mario World utilizando NCL e NCLua. O código Lua teria a Engine básica, com os eventos, cenários, comportamentos de inimigos e itens, etc, enquanto que o código NCL seria o responsável por dizer como a fase seria desenhada, quais inimigos aparecem nela, e onde aparecem. Ele pode utilizar âncoras para determinar o fim de jogo caso o tempo termine, e por aí

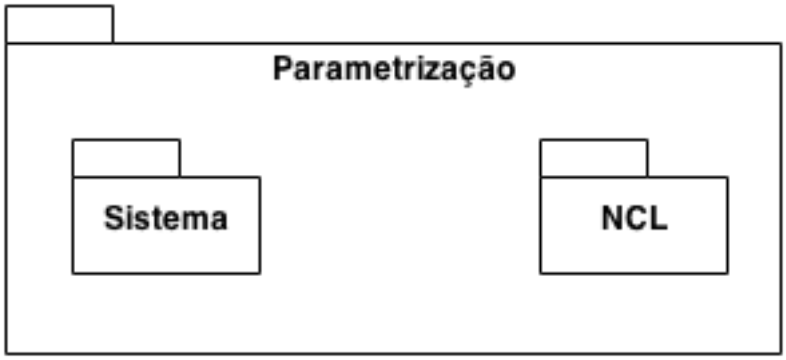

Figura 5 – Diagrama de Pacotes. Ginga Wings – Parametrização

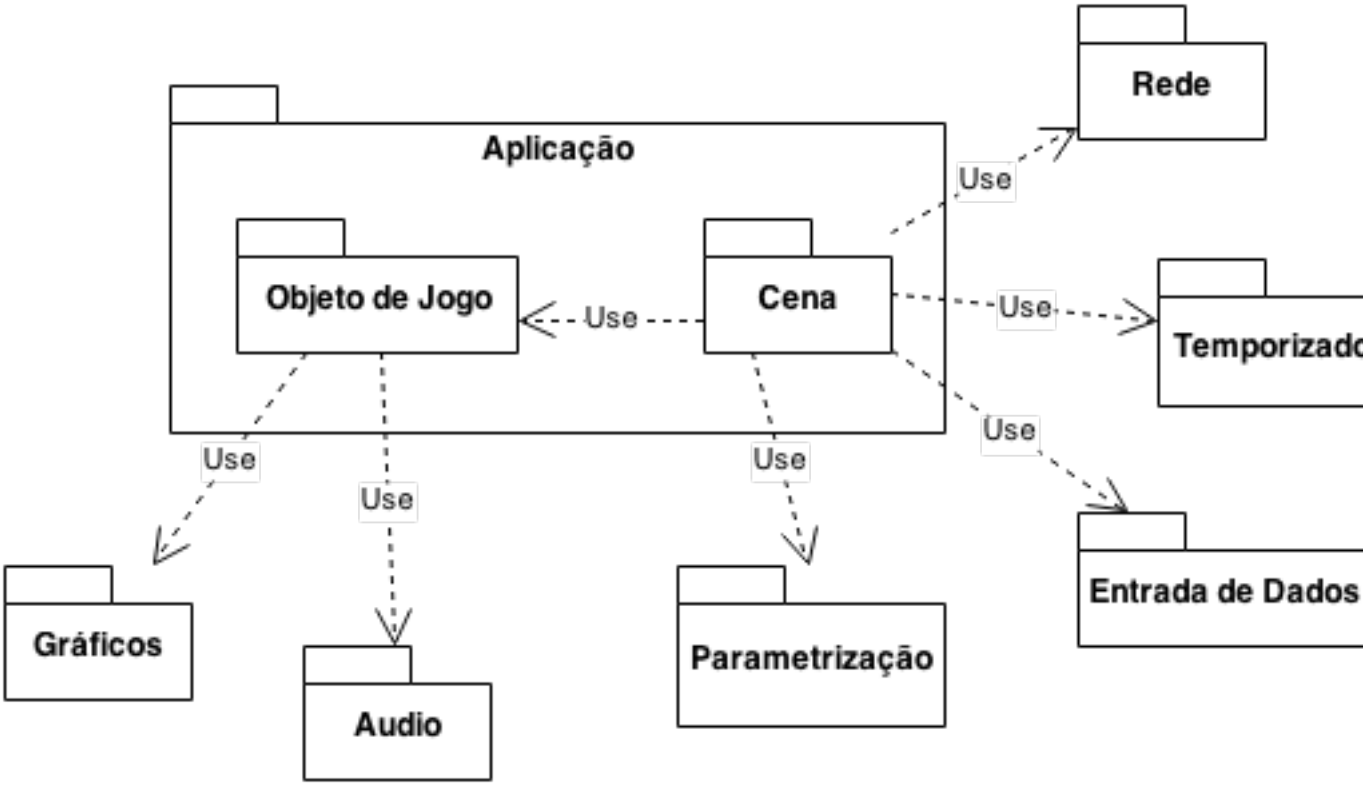

Figura 6 – Diagrama de Pacotes. Ginga Wings – Aplicação

vai. É preciso, no entanto, que o código Lua esteja pronto para receber e interpretar estes parâmetros. A parametrização NCL captura esses parâmetros e os envia para dentro do código Lua, onde classes especializadas devem estar prontas para recebe-los e processá-los. A parametrização de Sistema possui toda e qualquer configuração geral do jogo: qual a resolução base que ele executa, FPS aceitável, entre outras configurações.

3.1.4.4 Aplicação

Por fim temos o componente de Aplicação, que utiliza todos os outros componentes. Seu principal pacote é a Cena, que junta os componentes de Entrada de Dados, Temporizador, Rede e Parametrização em um só lugar. O Temporizador determina, neste pacote,

quantas vezes ele será atualizado por segundo. A Cena não interage diretamente com Músicas e Imagens, agindo através de Objetos de Jogo, estes sim, que encapsulam toda a estrutura de Mídia.

3.2 Detalhamento da API

3.2.1 Class

A criação de jogos geralmente envolve códigos grandes e complexos. Claro, alguns jogos mais simples podem ter apenas algumas dezenas de linhas de código os definindo completamente, desde a aplicação gráfica, a entrada de dados e as respostas do jogo. Porém, para a maioria dos casos um jogo é uma série de códigos grande e complexa. Com o crescimento de qualquer tipo de código, sua manutenção tende a ficar cada vez mais complicada. Uma solução para isto é OO, que permite uma maior organização do código ao agrupar o funcionamento do sistema em objetos, suas possíveis ações e seus relacionamentos com outros objetos. Com isto em vista, tornou-se uma necessidade a implementação de tal funcionalidade, já que Lua não dá suporte a OO, sendo na verdade uma linguagem de protótipos.

Código 3.1 – Exemplo de Criação de Classe

```
1.      GW_Codes.classes.Block = {}
2.      GW_Codes.classes.Block.superclass = "OBJECT"
3.      function GW_Codes.classes.Block:init()
4.              GWClasses.Block().__superclass().init(self)
5.      end
```

O propósito do módulo `Class` é emular a funcionalidade de classes de OO, permitindo definir uma classe, criar instâncias dela, e também ter herança não múltipla. É extremamente fácil criar uma classe utilizando este módulo, como mostra o exemplo no Código 3.1.

Lua funciona utilizando tabelas e metatabelas, que são os objetos nativos responsáveis por emular o funcionamento de classes para a GW. A linha 1 cria uma tabela que armazenará a estrutura da classe `Block`. A segunda linha define qual é a classe pai da classe `Block`. No caso, a classe `OBJECT`, que é a classe padrão, já implementada na GW, que disponibiliza todas as funcionalidades básicas da GW. Todas as classes devem herdar de `OBJECT`. É possível trabalhar sem ela, porém será mais difícil tirar proveito do que a GW tem a oferecer. A linha 3 define uma função que é inicializada toda vez que um objeto dela é instanciado. A linha 4 faz o equivalente a super em linguagens orientadas a objeto, permitindo que a classe pai inicialize seus atributos e métodos junto com a inicialização

da classe filha. A passagem do parâmetro self é obrigatória para que a classe pai possa alterar a metatabela da filha. A linha 5 termina a função.

Para criar uma instância da classe `Block`, basta chamar o método new.

Código 3.2 – Instanciando uma Classe

```
1.        GW_Classes['Block'].new()
```

O método criará uma instância da classe `Block` e passará a referência para a variável block. A tabela `GW_Classes` guarda uma referência para todas as classes instanciáveis – a exemplo, a classe `OBJECT`. O nome da classe será igual ao passado para a variável `GW_Codes` no Código 3.1.

3.2.1.1 Class:__class() : table

Retorna a estrutura de classe do objeto em questão.

3.2.1.2 Class:__classname() : String

Retorna o nome da classe do objeto.

3.2.1.3 Class:__superClass() : metatable

Retorna a estrutura de classe da classe pai do objeto.

3.2.1.4 Class:__is_a(string: nomedaclasse) : boolean

Verifica se o objeto é da mesma classe que a informada no parâmetro. Neste caso, verifica se os nomes são iguais.

3.2.1.5 Class:__is_a(table: classe) : boolean

Verifica se o objeto é da mesma classe que a informada no parâmetro. Neste caso, verifica se a metatable é igual à metatabela de classe do objeto.

3.2.1.6 Class:__respond-to(string: método) : boolean

Verifica se o método passado existe na classe em questão. Por ser uma tabela simulando uma classe, é possível que um objeto tenha mais funções visíveis do que a classe disponibiliza. Estas funções não são inspecionados por esta função.

3.2.2 Game_Object

Como mencionado, existe uma classe base, chamada simplesmente de OBJECT. Ela disponibiliza as funções básicas da GW, por isso é recomendado que qualquer objeto

de jogo herde desta classe. Um objeto que herde de OBJECT automaticamente terá acesso a disparadores de eventos (por exemplo, disparar um evento quando um objeto Block tocar em outro), terá acesso a um comportamento, invocado automaticamente, desenho na tela, automático, funções de acesso à rede e de destruição do objeto. A classe objeto dá a seus filhos acesso a um sprite, uma posição na tela, e uma tabela de variáveis locais.

3.2.2.1 Object:update() : boolean

É a função principal da classe, chamando sua função de comportamento, gerenciando seus eventos e também o desenhando na tela. Caso o objeto tenha sido desposado, não realiza nenhuma dessas ações. Retorna true caso consiga realizar todas as suas operações.

3.2.2.2 Object:dispose() : boolean

Desposa um objeto, ou seja, este objeto não será mais considerado como ativo pela GW. Retorna true caso consiga desposar o objeto.

3.2.2.3 Object:draw()

Desenha o objeto na tela.

3.2.2.4 Object:___create_event(string: event)

Cria um evento como o nome event. O evento criado é uma tabela, que possui a entrada command, uma função que é executada quando o evento é disparado, e outra entrada result, que guarda o resultado do evento gerado para que possa ser devidamente tratado.

3.2.2.5 Object:___consume_event(string: event)

Faz uma chamada à entrada command do event parâmetro caso ele tenha sido disparado. Retorna true caso tenha sucesso. Deve ser invocado por funções interessadas em obter uma funcionalidade do evento, por exemplo, um objeto Gato iria chamar o evento Rato:___consume_event('saiu_da_toca') e ler, em seguida, Rato.events['saiu_da_toca'] para verificar se o rato saiu da toca para que ele possa então caçá-lo. Um evento pode ser consumido até que reinicie o loop. O método command é invocado todas as vezes, portanto é preciso ser cauteloso ao codifica-lo.

3.2.2.6 Object:_____fire_event(string: event)

Dispara o evento event. Todos os objetos que invocarem a função ___consume_event() poderão obter os dados do evento gerado. Deve ser chamado pelo próprio objeto. Aproveitando o exemplo do gato e do rato, é o objeto rato que define Rato.___fire_event('saiu_da_toca'),

para que enfim o gato possa invocar o método ___consume_event() e correr atrás do rato fujão.

3.2.3 Timer

Este módulo, Game_Timer, é o responsável por controlar os eventos temporais na aplicação. Ela armazena o tempo geral da aplicação, a taxa de frame rates (que determina quantas vezes por segundo a tela é renderizada, e portanto, o sistema é atualizado) esperada pela aplicação, e o tempo entre um loop e outro. Por padrão, a taxa de frame rates é 20, tendo em vista que o Set-Top Box não foi projetado para rodar aplicações pesadas, e o olho humano conseguir distinguir imagens fluídas em até 18 frames por segundo. Por causa disso não é possível criar animações muito fluídas sob a configuração padrão. Ela pode ser alterada de acordo com as necessidades do usuário, porém a performance do sistema como um todo deve ser levada em consideração.

3.2.3.1 Game_Timer:sync(number: total_time) : number

Existem duas maneiras de se programar eventos temporais com frame rate: utilizando o frame rate como unidade de tempo; e utilizando o frame rate como medida de tempo. O primeiro cria uma estrutura temporal Dependente do Frame Rate (DFR). Se a taxa de frames for normal (usemos nosso exemplo de 20 frames por segundo), a aplicação irá executar em velocidade normal. Se, por algum motivo, a aplicação contiver um processo pesado e o frame rate cair para 10, a aplicação irá rodar duas vezes mais lento. Por outro lado, caso a aplicação receba atenção total do processador e conseguir rodar a 30 frames, o jogo irá rodar mais rápido. A segunda maneira, mais conhecida como Jogo em Tempo Real ou Independente de Frame Rate (IFR), se utiliza do intervalo entre cada loop do temporizador para calcular quanto tempo realmente passou, atualizando todos os objetos de jogo de acordo com o tempo passado. Desta forma, o jogo mantém sua velocidade de execução mesmo com a variação de frames por segundo – que por outro lado gera "paralizações" no jogo, geralmente chamadas de lag. Por outro lado, algumas aplicações ficam mais complexas, por exemplo, a detecção de colisão – é necessário realizar previsões dos movimentos dos objetos da cena para verificar se eles colidiram durante o intervalo de tempo. Com o devido cuidado, ambas as formas de manipular o tempo são viáveis, e o GW dá suporte a ambas.

Código 3.3 – Exemplo de uso da função sync

```
1.      pixels = Game_Timer:sync(20) --pixels = 20
```

A DFR é como a GW funciona normalmente. Para se trabalhar com a IFR, existe esta função no módulo Timer. Ele recebe um tempo e infere o quanto passou dele no intervalo entre os frames. O parâmetro deve indicar o movimento, através do tempo, por 1

segundo, sob o qual o método vai trabalhar. Por exemplo, se o Herói do jogo se movimenta a 20 pixels por segundo, em um jogo rodando a 20 FPS (frames por segundo), com a aplicação desta função, ele andará 1 pixel por segundo, a cada frame. Se por algum motivo a velocidade do jogo cair para 14 FPS, a velocidade do Herói será de 1.42 pixels por segundo.

3.2.4 Graphics

Módulo responsável pela renderização dos gráficos na tela. É possível informar uma ordem para que a renderização seja realizada. A GW possui uma resolução, nativa, que deve ser informada para o módulo. O jogo é renderizado na resolução nativa e após a renderização, se ajusta ao tamanho da tela da TV.

3.2.4.1 Game_Graphics:insertLayer(number: n, table: obj)

Caso o objeto obj possua o método draw, que é chamado para realizar a renderização do mesmo, ele é incluído na lista de objetos a serem renderizados pelo módulo. Objetos com um n menor são desenhados primeiro, portanto os que tiverem os maiores valores ficarão por cima no resultado final. Não há garantia de ordenação para objetos com valores n iguais.

Código 3.4 – Definindo a ordem de renderização para a classe Board

```
1.  GW_Codes.classes.Board.drawLayer = 1
2.  Game_Graphics:insertLayer(4, board)
```

Para setar um valor n padrão para todos os objetos de uma classe, basta seguir o exemplo no Código 3.4, linha 1. Valores individuais podem ser aplicados, também, como mostrado na linha 2.

3.2.5 Sprite

Módulo básico, que armazena as gráficos da aplicação. Dá suporte tanto a gráficos estáticos quanto animados. Possui também suporte para física básica, na forma de colisões.

3.2.5.1 Sprite.new(string: path) : table

Cria uma instância da classe Sprite com a imagem referenciada pelo caminho path. Caso não encontre a imagem cria uma Sprite sem imagem. O Sprite criado não é subdividido, tem animação ativado (mas não executa nenhuma), e tem uma área de colisão de acordo com o tamanho da imagem.

3.2.5.2 Sprite:set_crop(number: w, number: h)

Determina em quantos pedaços a imagem será repartida. Esta partição não altera o arquivo de imagem utilizado pelo Sprite, apenas determina pedaços de tamanho igual para serem desenhados com a função draw(). Qual parte será renderizada deve ser determinada pelo atributo anim_index. O index é definido a partir do número de linhas e colunas. Por exemplo, se $w = 4$ e $h = 4$, o $index = 9$ representará a parte da imagem localizada na linha 3, coluna 1.

Código 3.5 – Exemplo de uso da função set_crop

```
1.      board = Sprite.new("board.png");
2.      board:set_crop(4,4)
3.      board.anim_index = 6
```

3.2.5.3 Sprite:setup_frames(number: index, number: steady, string: sound, table: offset)

Permite configurar um frame de animação para o Sprite. Index informa qual o índice da animação (os frames são organizados de acordo com esse índice); steady informa quantos milissegundos o frame permanecerá ativo (valores muito baixos podem tornar o frame quase indistinguível, valores muito altos pode "paralisar" a animação); sound informa que o frame deve tocar o som informado pelo campo no início do frame. Um valor vazio ou nulo terão o mesmo efeito, sem som. Offset é uma tabela, com valores x e y, indicando que o Sprite deve mover sua imagem mas sem mover a matriz. Ou seja, a imagem renderizada vai mudar sua posição de pintura na tela, mas o objeto Sprite manterá sua posição original.

Código 3.6 – Exemplo de uso da função setup_frames

```
1.      board = Sprite.new("board.png");
2.      board:setup_frames(0, 4, null, {x=0, y=0})
```

3.2.6 Cache

Módulo responsável por gerenciar os recursos de mídia da aplicação. Gerencia tanto recursos de áudio como de imagem. O gerenciamento de imagens é automático, porém o de áudio, por depender do NCL para ser feito, precisa de configurações mais cuidadosas.

3.2.6.1 Game_Resources.load_image(string: path)

Carrega a imagem referenciada no caminho path na memória e retorna uma referência para ela. Chamadas posteriores a este mesmo path apenas retornarão uma referência. Alterações no arquivo de imagem, portanto, afetarão todos os objetos que chamarem a referenciarem.

3.2.6.2 Game_Resources.load_background(string: path)

Invoca um recurso de imagem do caminho padrão "media\backgrounds\". É um método auxiliar com a única finalidade de facilitar a obtenção de recursos de imagem.

3.2.6.3 Game_Resources.load_character(string: path)

Invoca um recurso de imagem do caminho padrão "media\characters\". É um método auxiliar com a única finalidade de facilitar a obtenção de recursos de imagem.

3.2.6.4 Game_Resources.free_image(string: path)

Informa ao Cache que a imagem em path deve ser liberada da memória. O Cache vai registrar o pedido, porém a imagem só será removida se todos os recursos que pediram a imagem o liberarem.

3.2.6.5 Game_Resources.play_SE(string: label)

Envia uma requisição ao NCL para tocar o áudio label. SE é a sigla para Sound Effects (Efeitos Especiais). SEs são geralmente utilizadas para representar sons ambientes e de HUDs / sistema (Head Up Display, como geralmente são chamadas as telas de interface do jogo, como status, vida, dinheiro, etc). Uma configuração especial para este tipo de som deve ser provida pelo arquivo .ncl que chamada o script do GW.

3.2.6.6 Game_Resources.stop_SE(string: label)

Envia uma requisição ao NCL para parar o áudio label. Uma configuração especial para este tipo de som deve ser provida pelo arquivo .ncl que chamada o script do GW.

3.2.6.7 Game_Resources.pause_SE(string: label)

Envia uma requisição ao NCL para pausar o áudio label. Uma configuração especial para este tipo de som deve ser provida pelo arquivo .ncl que chamada o script do GW.

3.2.6.8 Game_Resources.play_BG(string: label)

Envia uma requisição ao NCL para tocar o áudio label. BG é a sigla para Background Sound (Sons de fundo), ou seja, canções que ficam tocando, em loop, enquanto a aplicação executa. Quando a canção termina sua execução ela é iniciada novamente, criando um loop. Uma configuração especial para este tipo de som deve ser provida pelo arquivo .ncl que chamada o script do GW.

3.2.6.9 Game_Resources.stop_BG(string: label)

Envia uma requisição ao NCL para parar o áudio label. Uma configuração especial para este tipo de som deve ser provida pelo arquivo .ncl que chamada o script do GW.

3.2.6.10 Game_Resources.pause_BG(string: label)

Envia uma requisição ao NCL para pausar o áudio label. Uma configuração especial para este tipo de som deve ser provida pelo arquivo .ncl que chamada o script do GW.

3.2.7 Scene

Módulo que centraliza e controla todos os outros. Os Game_Objects devem ser manipulados junto com a lógica do jogo, utilizando este módulo. Uma instância do GW só tem 1 objeto Scene, que deve gerenciar os vários recursos.

3.2.7.1 GWScene:begin()

Esta função deve ser utilizada para inicializar todos os objetos da aplicação que rodarão junto com a scene.

Código 3.7 – Exemplo de uso da função begin

```
1.      function GWScene:begin()
2.              self.vars.color = 'red'
3.              self.vars.board = GWClasses['Board'].new()
4.              self.vars.score = GWClasses['Score'].new()
5.              table.insert(self.objects, self.vars.board)
6.              Game_Graphics:insertLayer(self.vars.board.
                        drawLayer, self.vars.board)
7.              table.insert(self.objects, self.vars.score)
8.              Game_Graphics:insertLayer(self.vars.score.
                        drawLayer, self.vars.score)
9.              GW_Codes.classes.Block.createBlocks()
10.             self:createToy()
11.             Game_Resources.play_SE('playback')
12.     end
```

3.2.7.2 GWScene:behavior()

Esta função determina o comportamento da Scene, e as interações entre os objetos. A lógica de comportamento interna do objeto não deve ser implementada aqui. O consumo de eventos, do Game Object, deve ser aplicado aqui.

Código 3.8 – Exemplo de uso da função behavior

```
1.      function GWScene:behavior()
2.              if self.vars.toy:__consume_event('endGame') then
3.                      self:End()
4.              end
5.              if self.vars.toy:__consume_event('gotFired') then
6.                      self:deleteObject(self.vars.toy)
7.                      self:createToy()
8.              end
9.              if self.vars.board:
                __consume_event('fieldChanged') then
10.                     local ls = self.vars.board.
                events['fieldChanged'].result
11.                     self.vars.score:
                addScore(ls * 100 + (ls - 1) * 50)
12.             end
13.     end
```

3.2.7.3 GWScene:End()

Finaliza a Scene e a aplicação. Caso algum dado deva ser escrito na finalização, este método deve ser sobrescrito. O Quadro 8 mostra um exemplo da aplicação desta função.

3.2.7.4 GWScene:getObjects(string: className) : table

Obtém todos os objetos registrados na Scene com o nome de classe className. Útil para gerenciar os objetos na Scene durante as chamadas ao behavior.

Código 3.9 – Exemplo de uso da função getObjects

```
1.      blocks = GWScene:getObject('Block')
```

3.2.7.5 GWScene:deleteObject(table: obj)

Deleta o objeto da Scene.

Código 3.10 – Exemplo de uso da função deleteObject

```
1.      blocks = GWScene:getObject('Block')
2.      for k,v in pairs(blocks) do
3.              GWScene:deleteObject(v)
4.      end
```

Figura 7 – Exemplo de Colisão por Área de Quadrado. Não há colisão entre os dois primeiros, mas há colisão nos subsequentes

3.2.8 Parameters

Tem como objetivo servir como a ponte configurável entre o NCL e o código Lua. Através destes objetos se passa argumentos para a aplicação Lua, que determinarão como a aplicação irá funcionar. Deve-se levar em conta que esta funcionalidade é opcional para a execução da aplicação Lua – um jogo pode rodar, tranquilamente, sem depender dos parâmetros vindos do NCL. Ela deve ser utilizada pelo desenvolvedor para permitir que o desenvolvedor NCL possa personalizar o jogo de acordo com suas necessidades.

3.2.8.1 GWInitializer:parse(id: string,par: string)

Método invocado no código principal, recebe todos os parâmetros enviados pelo NCL. Sua funcionalidade deve ser reescrita para atender às necessidades da aplicação. Deve-se levar em conta que, apesar de ser possível enviar parâmetros durante a vida útil da aplicação (Scene), os primeiros parâmetros são enviados antes da Scene iniciar, não existindo, portanto, objetos inerentes à Scene neste momento.

3.2.9 Collision

Tem a função de gerenciar as colisões entre objetos na cena. Um objeto colide com outro caso sua área toque na área do outro objeto. Existem várias abordagens para se obter este efeito. A primeira, mais simples, é a colisão de quadrados. Define-se um quadrado mínimo que cerca o objeto, e verifica-se se este colidiu com o outro.

Este tipo colisão é eficaz em jogos que não necessitam de muita precisão na colisão. Objetos com áreas muito pequenas, e similares à área do quadrado também podem utilizar este modo de colisão. Seu custo de processamento é baixo.

Para casos em que há uma necessidade de maior precisão, mas não o suficiente para gerar um custo de computação, é possível se utilizar de um segundo método de colisão, a colisão por círculos, que simplesmente detecta se o círculo que contém o objeto está na área de um outro. Em jogos 3D, esse tipo de colisão é usada para objetos que não requerem muito controle de colisão, utilizando esferas no lugar de círculos.

Figura 8 – Exemplo de Colisão por Área de Círculo. Não há colisão entre os dois primeiros, mas há colisão nos subsequentes

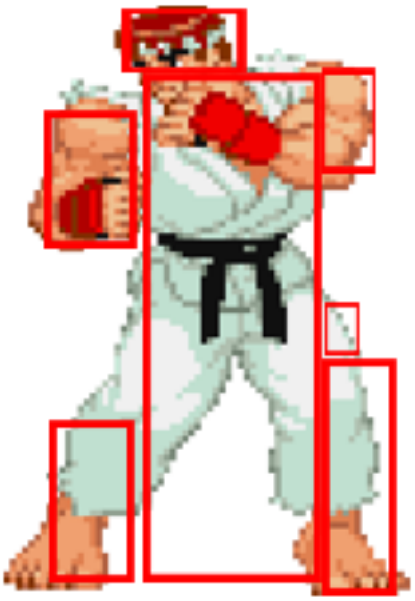

Figura 9 – Exemplo de Configuração de Colisão com Múltiplos Quadrados

Estes dois primeiros casos são suportados pela GW. São simples de implementar e possuem pouco custo. Existe uma terceira maneira, extremamente precisa, utilizada em jogos 2D (que geralmente tem seus gráficos desenhados com pixel-art), que é a Colisão de Pixels Perfeita. Este tipo de colisão lê a matriz de pixels do gráfico de ambos os objetos (não sendo necessário definir a área de colisão), e verifica, de acordo com a posição absoluta do objeto na tela, se algum dos pixels sobrescreve ao outro. Caso sim, existe uma colisão. Este tipo de colisão é custoso e não é oferecido por padrão pela GW. É possível, no entanto, estabelecer um algoritmo que simule a precisão da Colisão de Pixels Perfeita. A GW permite que se aplique várias áreas de colisão por objeto. Utilizando-se a Colisão por Quadrado, pode-se fragmentá-los para que cubram apenas a área desejada no gráfico, criando uma ilusão de Colisão de Pixels Perfeita. A desvantagem dessa abordagem é a configuração, geralmente custosa para o usuário que desenvolverá as animações, por ter que definir os quadros de colisão a cada quadro da animação.

3.2.9.1 ___collision_setup(offx: number, offy: number, wd: number, hg: number, frame: number, ct: number)

Cria uma configuração de colisão para objeto no frame especificado, com duração ct (em milissegundos), com tamanho wd x hg na posição offx vs offy. É possível criar várias caixas num mesmo frame para dar suporte ao modelo explicado acima.

3.2.9.2 left() : number

Retorna o limite mais à esquerda de todas as caixas de colisão no frame atual.

3.2.9.3 right() : number

Retorna o limite mais à dereita de todas as caixas de colisão no frame atual.

3.2.9.4 top() : number

Retorna o limite mais acima de todas as caixas de colisão no frame atual.

3.2.9.5 bottom() : number

Retorna o limite mais abaixo de todas as caixas de colisão no frame atual.

3.2.9.6 colide(obj: table) : boolean

Determina se o parâmetro obj possui caixas em colisão com as do objeto do qual a chamada foi realizada.

3.2.10 Input

Módulo responsável por controlar a entrada de eventos do controle, a fim de interagir com o jogo. Possui basicamente as entradas vermelho, verde, azul, amarelo, cima, baixo, esquerda, direita, menu, sair, sendo esses botões um padrão em controles de TVs Digitais. Todos os métodos retornam um boolean indicando se um botão foi pressionado ou não, diferenciando-se a quantidade de vezes quando o evento é disparado.

3.2.10.1 GW_Input.pressed(key: string) : boolean

Retorna true enquanto o botão key estiver pressionado. Útil para eventos repetitivos em jogos, como por exemplo aceleração em jogos de corrida.

3.2.10.2 GW_Input.trigger(key: string) : boolean

Retorna true apenas no frame em que o botão key é pressionado. Se o botão continuar pressionado, retorna false continuamente até que ele seja solto. Útil para eventos

não repetitivos em jogos, por exemplo, o salto de um personagem de um jogo de side-scrooling.

3.2.10.3 GW_Input.consecutive(key: string) : boolean

Retorna true repetidamente enquanto o botão key estiver pressionado, em pequenos intervalos.

3.2.11 Utils

Módulo agrupados de funções comuns e não associadas à classes e objetos em geral.

3.2.11.1 PROMPT(msg: string)

Exibe a mensagem msg na tela por alguns segundos, servindo como um tipo de prompt para testes no jogo. Não deve ser utilizado para apresentar as mensagens do jogo.

3.3 Aplicação de Exemplo

Como exemplo de uso da ferramenta, será mostrado o passo-a-passo da criação de um jogo básico, um Hello World. O jogo deverá apresentar o uso das principais funcionalidades da ferramenta. Como objetivos nesse exemplo, teremos:

- Apresentação de cenário de fundo e de fase;
- Um personagem controlado pelo controle remoto;
- Funcionamento de física (gravidade e colisões);
- Reprodução de BGM em loop.

Para programar o cenário de fundo e de fase, são necessárias duas classes diferentes. O cenário de fundo (Background) não possui interatividade com o jogador (apesar de em alguns tipos de jogos reagir a ele, como em cenários de fundo que utilizam a técnica de parallax scrolling), e, portanto é mais fácil de implementar. O cenário de fase (Base) interage com o jogador através de colisões, sendo o chão da fase. O jogador, se colidir com a face superior do cenário de fase, deve permanecer nela, não a atravessando.

O personagem de fundo deve responder ao botar vermelho do controle (ação de pular) e às setas esquerda e direita (movimento). Para programar o pulo é necessário implementar gravidade. Para a implementação de gravidade, o personagem deve ter peso, que será considerado 1. Enquanto o personagem não estiver em contato com a face superior de nenhum cenário de fase, deve receber uma aceleração negativa de -9.8m/s. Se ele pular,

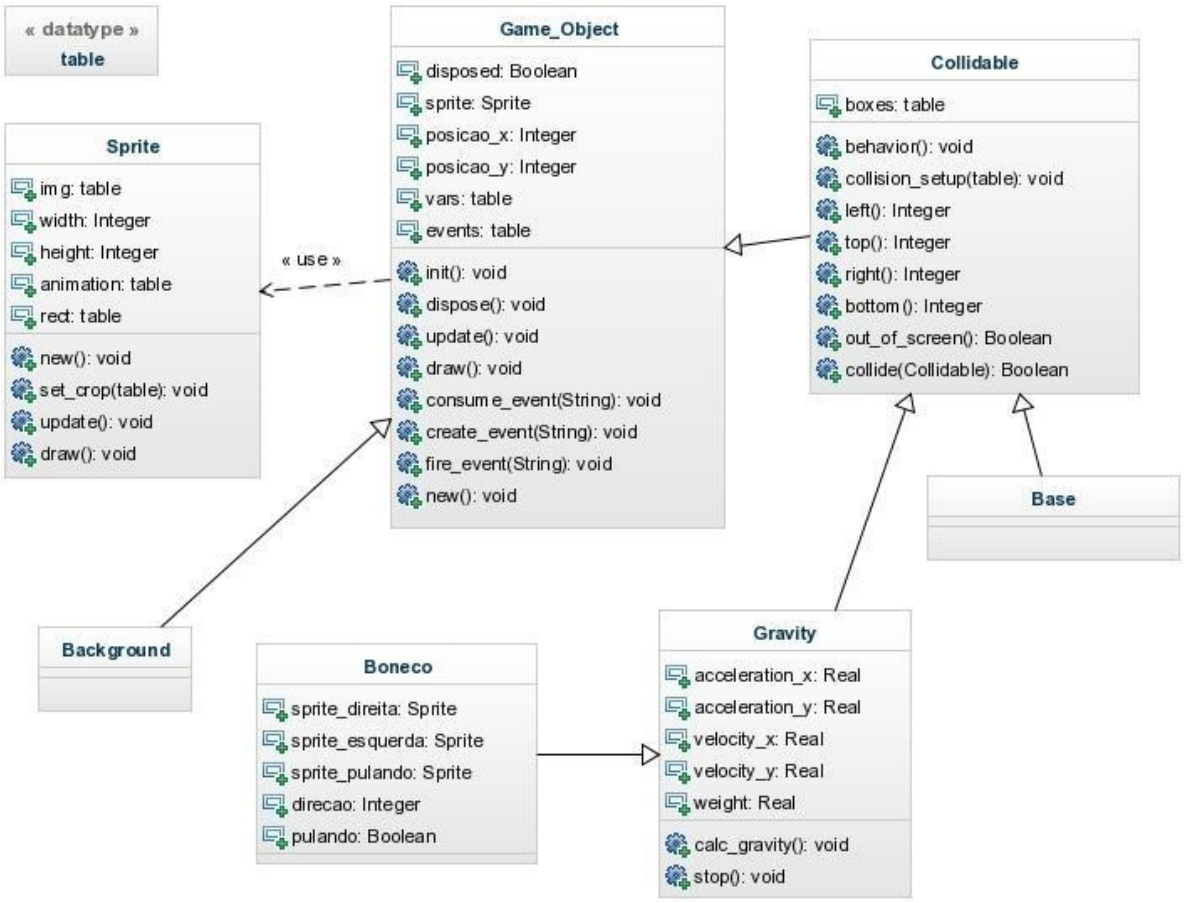

Figura 10 – Diagrama Entidade Relacionamento para a Aplicação de Exemplo

deve receber uma aceleração superior a 9,8 m/s a fim de alcançar uma altura desejada. A Figura 10 deve apresentar o diagrama de classes para o jogo de exemplo.

Todos os objetos da cena (Background, Boneco e Base) herdam de Game_Object, que possui os atributos básicos de um objeto de jogo. As classes Boneco e Base precisam reagir a colisões, portanto também herdam de Collidable (que por sua vez herda de Game_Object). A classe boneco precisa reagir à gravidade, portanto herda de Gravity, que gerencia informações quanto à gravidade.

A implementação da classe Background no Código 3.11, seguido da implementação da classe Base no Código 3.12, da classe Gravity no Código 3.13 e da classe Boneco no Código 3.14.

Código 3.11 – Classe Background

```
1.      GW_Codes.classes.Background = {}
2.      GW_Codes.classes.Background.superclass = "OBJECT"
3.      function GW_Codes.classes.Background:init()
4.              GWClasses.Background.__superClass().init(self)
5.              self.sprite =
```

```
                Sprite.new('/backgrounds/back1.png')
6.      end
```

Código 3.12 – Classe Base

```
1.      GW_Codes.classes.Base = {}
2.      GW_Codes.classes.Base.superclass = "COLLIDABLE"
3.      function GW_Codes.classes.Base:init()
4.              GWClasses.Base.__superClass().init(self)
5.              self.sprite =
                Sprite.new('/characters/base.png')
6.              self.pos.x = 0
7.              self.pos.y = 600-33
8.              self.vars.__collision_data.loop = true
9.              self:__collision_setup(0,0,800,33,100)
10.     end
```

Código 3.13 – Classe Gravity

```
1.      GW_Codes.classes.Gravity = {}
2.      GW_Codes.classes.Gravity.superclass = "COLLIDABLE"
3.      function GW_Codes.classes.Gravity:init()
4.              GWClasses.Gravity.__superClass().init(self)
5.              self.vars.accy = 0.0
6.              self.vars.accx = 0.0
7.              self.vars.velx = 0.0
8.              self.vars.vely = 0.0
9.              self.vars.weight = 1.0
10.     end
11.
12.     function GW_Codes.classes.Gravity:calc_gravity()
13.             local add_accel = {
                        x = Game_Timer:sync(self.vars.accx),
                        y = Game_Timer:sync(self.vars.accy) }
14.             self.vars.velx = self.vars.velx + add_accel.x
15.             self.vars.vely = self.vars.vely + add_accel.y
16.             self.pos.x = self.pos.x + self.vars.velx
17.             self.pos.y = self.pos.y + self.vars.vely
18.     end
```

Código 3.14 – Classe Boneco

```
1.      GW_Codes.classes.Boneco = {}
2.      GW_Codes.classes.Boneco.superclass = "Gravity"
3.      function GW_Codes.classes.Boneco:init()
4.              GWClasses.Boneco.__superClass().init(self)
5.              self.vars.sprites = {}
6.              self.vars.sprites.d =
                Sprite.new('/characters/mario_parado.png')
7.              self.vars.sprites.e =
                Sprite.new('/characters/mario_parado_m.png')
8.              self.sprite = self.vars.sprites.d
9.              self.pos.x = 800/2-16
10.             self.pos.y = 16
11.             self.vars.weight = 2.5
12.             self.vars.lado = 'd'
13.             self.vars.__collision_data.loop = true
14.             self:__collision_setup(9,7,15,19,100)
15.     end
16.
17.     function GW_Codes.classes.Boneco:behavior()
18.             GWClasses.Boneco.__superClass().behavior(self)
19.             local vel = 5.0
20.             self.vars.velx = 0.0
21.             if GW_Input.pressed('CURSOR_RIGHT') then
22.                     self.sprite = self.vars.sprites.d
23.                     self.vars.velx = vel
24.             end
25.             if GW_Input.pressed('CURSOR_LEFT') then
26.                     self.sprite = self.vars.sprites.e
27.                     self.vars.velx = -vel
28.             end
29.             local tempx = Game_Timer:sync(self.vars.velx)
30.             self.pos.x = self.pos.x + self.vars.velx
31.     end
```

As classes Background e Base são estáticas na tela, portanto é seguro aplicar-lhes um posicionamento estático por toda a vida da aplicação.

Tendo sido implementadas as classes de funcionamento básico do jogo, podemos passar para a criação da Scene que irá gerenciar a lógica e recursos. Será dever da Scene instanciar e destruir objetos, invocar BGMs, determinar o fluxo da lógica (muitas vezes

chamada de Máquina Finita de Estados), etc. A GW provê uma Scene já pronta para uso, ela não deve ser instanciada. Em casos da necessidade do uso de mais de uma Scene, o ideal é a implementação de uma nova classe que desempenhe a função, sendo controlada pela Scene provida pela GW.

No caso da aplicação de exemplo, a Scene vai instanciar o Backgroun, a Base e o Boneco. Vai tocar uma BGM em modo infinito (loop) e vai gerenciar os eventos disparados pela interação dos objetos. O código segue no Código 3.15.

Código 3.15 – Código da Cena

```
1.      function GWScene:begin()
2.              print("cena inicializada")
3.              self.vars.background =
                GWClasses['Background'].new()
4.              self.vars.mario = GWClasses['Boneco'].new()
5.              self.vars.base = GWClasses['Base'].new()
6.              table.insert(self.objects, self.vars.background)
7.              Game_Graphics:
                insertLayer(1, self.vars.background)
8.              table.insert(self.objects, self.vars.base)
9.              Game_Graphics:insertLayer(1, self.vars.base)
10.             table.insert(self.objects, self.vars.mario)
11.             Game_Graphics:insertLayer(2, self.vars.mario)
12.             Game_Resources.play_BG('playback', true)
13.             self.vars.gravity = 9.8
14.     end
15.
16.     function GWScene:behavior()
17.             local mario = self.vars.mario
18.             if mario:right() >= Game_Graphics.width then
19.                     mario.pos.x = Game_Graphics.width -
                (mario:frame().offset.x + mario:frame().width)
20.             end
21.             if mario:left() <= 0 then
22.                     mario.pos.x = -mario:frame().offset.x
23.             end
24.             mario:calc_gravity()
25.             if mario:collide(self.vars.base) then
26.                     mario.vars.accy = 0.0
27.                     mario:stop()
```

```
28.                     mario.pos.y = self.vars.base:top() -
                        (32-7)
29.             else
30.                     if mario.vars.accy < 0 then
31.                             mario.vars.accy =
                        mario.vars.accy + self.vars.gravity
32.                     else
33.                             mario.vars.accy =
                                self.vars.gravity
34.                     end
35.             end
36.             if GW_Input.pressed('CURSOR_UP') and
                        mario:collide(self.vars.base) then
37.                     mario.vars.accy = mario.vars.accy -
                        self.vars.gravity * 5
38.             end
39.     end
```

As linhas 2-11 criam os objetos na cena e os registram junto à Scene e ao módulo de gráficos (Game_Graphics). O registro com o Game_Graphics permite que o módulo reconheça os objetos como desenháveis e vai invocar o método draw() de cada objeto registrado para que apareça na tela. A linha 12 faz com que a BGM playback seja tocada. Na linha 13, setamos a gravidade global para o valor 9,8 (m/s).

Na função behavior o objeto mario (Boneco) tem sua gravidade setada para a global. Da linha 18-23 é feito o tratamento para quando o objeto tentar sair da tela pelos lados. Nas linhas 25-35 é feito o tratamento para quando o objeto colide com a base. As linhas 36-38 tratam o pulo do boneco, adiciona aceleração negativa (assim ele irá para cima).

Com a Scene pronta, é necessário agora configurar os sons. A configuração do som exige que se utilize o módulo Parameters. A GW não tem como tocar sons nativamente, nem saber sua duração. Serão utilizados parâmetros para se informar esses dados através do NCL. Apenas uma sobrescrita da função parse é necessária, como mostrado no Código 3.16.

Código 3.16 – Código dos Parâmetros

```
1.      function GWInitializer:parse(id,par)
2.              if string.match(par, "BGM%d+") ~= nil then
3.                      print("Registering BGM "..id)
4.                      local duracao =
```

```
                    string.match(par, "BGM(%d%d%d%d)")
5.                  local minutos =
                    tonumber(string.sub(duracao, 0, 2))
6.                  local segundos =
                    tonumber(string.sub(duracao, 3))
7.                  Game_Resources.
                    register_sound(id,
                            minutos * 60 + segundos)
8.              end
9.              if par == "End" then
10.                 print("BGM "..id.." ended")
11.                 Game_Resources.end_BG(id)
12.             end
13.     end
```

Nesta implementação, a função recebe um parâmetro informando o id da mídia no NCL e sua duração no parâmetro par. A GW reconhece que é um som caso contenha a palavra BGM seguido de 4 inteiros, que representam a duração do som em minutos e segundos. Essa duração é utilizada para saber quando o som terá terminado e mandar tocar de novo. Para que funcione, é preciso que o NCL disponibilize links para inicializar e finalizar uma mídia de som. Todo o trabalho do lado do código Lua está pronto. Agora é necessário criar o código NCL que irá invocar e parametrizar a aplicação Lua. O código segue nos Códigos 3.17 e 3.18.

Código 3.17 – Regiões e Descritores de Exemplo

```
<!-- Regioes -->
<regionBase>
    <region id="screen" width="100%" height="100%"  zIndex="5"/>
    <region id="music" width="100%" height="100%" zIndex="0"/>
</regionBase>
<!-- Descritores -->
<descriptorBase>
    <descriptor id="dsScreen" region="screen" focusIndex="240"/>
    <descriptor id="dsMusic" region="music">
        <descriptorParam name="soundLevel" value="1"/>
    </descriptor>
</descriptorBase>
```

Código 3.18 – Portas, Medias e Links

```
<!-- Porta -->
```

```
<port id="entryPoint" component="lua"/>
<media id="lua" src="codes/main.lua" descriptor="dsScreen">
    <property name="start1"/>
    <property name="playback"/>
</media>
<media id="playback" type="audio/mp3"
        src="media/backtracks/teste.mp3"/>
<media id="programSettings" type="application/x-ginga-settings">
    <property name="service.currentKeyMaster" value="240"/>
</media>
<!-- DEFINICAO DOS LINKS AQUI!!!!!!!! -->
<link xconnector="onBeginSetN">
    <bind role="onBegin" component="lua" />
    <bind role="set" component="lua" interface="start1"/>
    <bind role="set" component="lua" interface="playback">
        <bindParam name="var" value="BGM0036"/>
    </bind>
</link>
<link xconnector="onEndAttributionStart">
    <bind role="onEndAttribution" component="lua"
        interface="playback" />
    <bind role="start" component="playback" />
</link>
```

É criada a mídia playback com o som a ser reproduzido. São inseridas propriedades na mídia Lua, que tem finalidades específicas: a propriedade start1 informa à GW que ela deve esperar por 1 parâmetro pelo menos (além do próprio start1). Isso ocorre porque a passagem de parâmetros entre o NCL e o Lua não segue a ordem estabelecida no código XML e em alguns casos, quando há muitos links, alguns deixam de ser enviados. O parâmetro playback será o id do som na GW. Este parâmetro deve ser setado com o valor BGM0036. Como dito acima, BGM diz à GW (neste exemplo) que o parâmetro é um som, enquanto que 0036 informa que o som tem a duração de 0 minutos e 36 segundos. Com esta configuração, está pronta o Hello World da GW, apresentado nas Figuras 11, 12 e 13.

Figura 11 – Boneco em Queda Livre

Figura 12 – Boneco em Repouso na Base

Figura 13 – Boneco durante Salto

4 Estudo de Caso: Tetris

A fim de testar a capacidade da engine, este capítulo apresenta um estudo de caso de um jogo, nominamente, uma versão de Tetris implementada para TVi. Um jogo, sendo uma aplicação de hipermídia, envolve diversas formas de se comunicar com o usuário. Jogos se utilizam de gráficos, efeitos sonoros, jogabilidade e narrativa para se comunicar com o jogador. Para oferecer todos estes tipos de mídia é necessário hardware capaz de processar esse tipo de dado. O STB em geral (modelos econômicos) não possuem muito poder de hardware, não tendo como função executar aplicações pesadas e/ou complexas. É preciso, portanto, analisar a capacidade de processamento da STB em relação ao jogo para averiguar se o hardware é capaz de entregar uma taxa de frames razoável. A jogabilidade deve ser avaliada através de uma análise da ergonomia e resposta do controle remoto (meio principal de entrada de dados no STB). Controles não foram designados para responder a múltiplos comandos simultâneos, portanto os jogos precisam ser viáveis recebendo uma única entrada de cada vez. Por fim, deve-se analisar qual é o público alvo para o jogo. A maioria da população disposta a jogar jogos casuais o farão utilizando o telefone celular. Para que algum jogo possa ter sucesso em uma TV Digital, é preciso realizar um estudo da população disposta a jogar na TV Digital e o que eles esperam em um jogo. Todos esses fatores influenciam na escolha de um jogo de sucesso na TV Digital utilizando a STB.

Realizar uma triagem do anseio da população disposta a jogar em uma TV Digital está fora do escopo deste trabalho, portanto apresentaremos um jogo pioneiro na história dos jogos, comumente utilizado como aplicação de teste em engines de games por sua (deveras) simplicidade e fama: Tetris.

4.1 Limitações do Set-Top Box

4.2 Tetris

4.2.1 História

Em 1984, enquanto trabalhava para o Centro de Computação da Academia de Ciências da URSS, Alexey Pajitnov anteviu um jogo eletrônico em que os jogadores poderiam arranjar peças de um quebra-cabeças em tempo real, enquanto eles caíram da parte superior do campo de jogo em velocidades cada vez maiores. Usando um computador de 60 Electronika, ele projetou um jogo que apresentava sete distintas peças geométricas, cada um composto por quatro quadrados. Alexey chamou o jogo de "Tetris", uma combinação da palavra "tetra"(palavra grega que significa "quatro") e "tênis"(seu esporte favorito). O

jogo foi portado para o PC IBM e tornou-se um sucesso imediato com seus colegas, se espalhando como fogo em toda a União Soviética.

Em pouco tempo, Tetris começou sua expansão global, lançando em PCs na América do Norte e Europa. O jogo foi apresentado em 1988 na Consumer Electronics Show, em Las Vegas, onde Henk Rogers, designer de jogos e editor, se deparou com Tetris. Ele ficou imediatamente viciado e achou que havia algo especial sobre o jogo. Sua empresa, a Bullet-Proof Software, obteve os direitos para lançamentos de Tetris para PC e NES no Japão, e mais de 2 milhões de cópias foram vendidas. Já em 1989, Henk se encontrou com Alexey, autor do jogo. Henk manteve posse dos direitos de Tetris para portáteis e os licenciou para a Nintendo, que colocou Tetris para lançamento junto com seu novo console portátil Game Boy. Esta associação conseguiu resultou na venda de 35 milhões de cópias, alavancando ainda mais os nomes das duas empresas.

Em 1997, Henk estabeleceu a Blue Planet Software, Inc. como agente exclusivo para a marca Tetris. Pouco depois, The Tetris Company foi formada, tornando-se a fonte de todas as licenças para Tetris. Muitos aspectos do jogo tornaram-se padronizados. Através de empresas como G-mode, Blue Lava Wireless e EA, Tetris é embarcado para sistemas mobile, fazendo com que o jogo fosse reconhecido como pioneiro na indústria de games casuais.

4.2.2 Regras de Tetris

As regras no jogo tetris se definem como especificado a seguir. O site oficial de Tetris não especificam as regras do jogo, e portanto essas regras citadas a seguir foram extraídas através da experiência os jogadores.

O jogo acontece dentro de uma máquina, chamada Tétrion. O Tétrion é formado de dez espaços horizontais e 20 verticais, onde se pode mover os tetraminós. Tetraminós escolhidos aleatoriamente caem do topo do Tétrion, um de cada vez. Cada tetraminó entra no campo com uma orientação e cor dependente de seu formato. Parte do Tétrion, chamada *piece preview*, mostra as próximas peças que irão entrar em campo.

O jogador pode rotacionar o tetraminó que está caindo em noventa graus, considerando que haja espaço para que a peça rotacione. Algumas versões do jogo ajustam a posição do tetraminó para que ele consiga realizar a rotação.

O jogador pode movimentar o tetraminó para os lados (um espaço por vez), considerando que haja espaço para que a peça se mova. As peças não podem ultrapassar paredes ou outros blocos.

No topo-esquerda, ou em outros casos, no fundo-direita do Tétrion, pode existir uma área chamada *hold box*, onde o jogador pode armazenar um tetraminó para uso no futuro. A qualquer momento, enquanto um tetraminó cai, o jogador pode movê-lo para a

hold box, fazendo com que o tetraminó armazenado seja levado ao topo do Tétrion.

Cada tetraminó se move para baixo, devagar. Geralmente, o jogador pode usar algum método para "derrubar"o tetraminó, ou fazê-lo se mover mais rápido para baixo. Quando a peça cai no chão ou em outras peças, ela irá aguardar um pouco antes de se fixar ao campo. Durante este tempo o jogador poderá ainda movê-lo para tentar encaixá-lo em algum lugar. Depois de fixo, o jogador já não pode mover mais o tetraminó.

Quando o tetraminó se fixa e preenche todo o espaço de uma linha, esta linha irá ser limpa (as peças na linha, removidas). Os blocos acima da linha irá se mover para baixo, de acordo com a quantidade de linhas preenchidas.

Se o campo não estiver preenchido com blocos, a próxima peça entra. Se o tetraminó não puder ser alocado no campo, pelo menos parcialmente, devido às regras acima, é considerado fim de jogo.

4.2.3 Diretrizes

A partir da criação da empresa Tetris Company, foi criada uma diretriz para que todos os jogos lançados posteriormente mantivessem um padrão de jogabilidade. Essas diretrizes, assim como as regras, não estão disponíveis oficialmente e foram inferidas através da experimentação dos jogadores.

- Campo de jogo do Tétrion tem tamanho de 10x22 células, sendo as linhas acima da 20ª ocultas (servem para inserir tetraminós parcialmente no campo).
- Cores dos tetraminós
  - I: ciano
  - O: amarelo
  - T: roxo
  - S: verde
  - Z: vermelho
  - J: azul
  - L: laranja
- As peças I e O aparecem nas colunas centrais
- As outras aparecem guidas mais à esquerda
- Os tetraminós aparecem em formato horizontal, com a parte plana apontada para baixo

- O *Super Rotation System* (SRS) especifica a rotação do tetrominó
- Mapeamento padrão para os controles:
  - As setas para cima, para baixo, para direita e para esquerda. Para cima derruba imediatamente a peça (*Hard Drop*), para baixo a derruba mais rapidamente (*Soft Drop*), para os lados, move a peça.
  - O botão de ação da esquerda rotaciona a peça 90º em sentido anti-horário, enquanto que o botão de ação da direita rotaciona em sentido horário.
- Sistema gerador aleatório de peças (*Random Generator*)
- "Segurar a peça": O jogador pode pressionar um botão para enviar o tetraminó que está caindo para a *hold box*, e qualquer tetraminó que já estiver na *hold box* se move para o topo da tela. Este comando não pode ser usado novamente até que a peça trocada se fixe no Tétrion. Jogos com pouca disponibilidade de botões podem ignorar esta funcionalidade.
- O jogo precisa ter a função de *Ghost Piece* (peça fantasma).
- O jogador só pode subir o nível de dificuldade limpando linhas ou relizando o movimento *T-Spin*.
- O jogo precisa utilizar a logo desenvolvida por Roger Dean ou uma variação.
- O jogo precisa incluir a canção clássica de Tétris, Korobeiniki.
- O jogador perde o jogo quando a peça aparecer sobre outra peça já fixa no Tétrion, ou quando ela se fixa completamente fora do escopo de visão do campo.

Pelo fato dessas regras não serem de domínio público, não é possível estabelecer com precisão algumas características, como a velocidade de queda, o sistema de pontuação, entre outros.

4.2.3.1 Super Rotation System

Super Rotation System, ou SRS, é o atual padrão de comportamento dos tetrominós. SRS determina onde e como tetrominós aparecem, como eles rotacionam, e quais *wall kicks* eles podem executar. Todos os tetraminós são definidos dentro de um quadrado, e rotacionam pelo seu centro, a não ser que sejam impedidos. As peças com largura 3 (J, L, S, T, Z) são posicionadas nas duas linhas superiores do quadrado e (para J, L e T) com o lado plano para baixo. I é colocado na linha superior. Todos os tetraminós aparecem nas duas linhas ocultas no topo do campo de jogo. São posicionados no meio dessas linhas, com preferência à esquerda caso não seja possível colocá-los exatamente no centro. Uma

vez que o tetraminó pouse (no chão ou em outro tetraminó), ele não se fixa até que se passe o tempo de espera para fixação (*delay*). Este comportamento, chamado de *Infinity* pela *Tetris Company*, reseta o *delay* quando o tetraminó é movido ou rotacionado. O movimento *Hard Drop* não sofre *delay*, fixando-se automaticamente, na maioria dos casos (já que nem todas as implementações de Tetris seguem à risca as diretrizes).

Um *wall kick* ocorre quando o jogador rotaciona a peça e não há espaço para que tal movimento seja realizado. O jogo então move a peça para uma posição em que seja possível realizar a rotação sem colidir com os tetraminós já fixados no campo. O algoritmo mais simples desse movimento é tentar movimentar a peça um espaço para a direita, e então um para a esquerda, e falhar caso nenhuma das duas opções seja possível.

The Random Generator é o nome dado ao algoritmo utilizado para gerar a sequência de tetraminós. Ele gera uma sequencia de todos as peças, permutadas randomicamente. Essas peças são servidas ao Tétrion antes de ser gerado uma nova sequência. Nos jogos que implementam este gerador, acredita-se que haja probabilidade igual de aparecimento para cada peça, tornando menos provável casos onde se passa muito tempo sem receber uma peça específica. O algoritmo produz uma diferença entre duas peças iguais de, no máximo, 12 e no mínimo, 4 peças. Algumas implementações quebram essa regra, fazendo sempre uma peça na primeira posição (I, J, L ou T).

A *Ghost Piece* é uma representação de onde o tetraminó irá eventualmente cair se não for movido. Tem uma cor geralmente mais transparente, e é desenhado por trás da peça original. Movimentos no tetraminó são refletidos na *Ghost Piece*.

4.3 Documentação da Aplicação

A aplicação tem como objetivo criar um jogo de Tetris para TV Digital, seguindo as diretrizes já expostas (com algumas ressalvas, mais detalhadas à frente), para um jogador, capaz de gerar e manter um *hi-score* para fins auto-comparativos. O jogo terá diversos níveis de dificuldade, sendo que a diferença entre os níveis está na velocidade de queda das peças. Das diretrizes descritas em 4.2.3, este jogo não irá ter a logo, nem utilizar a canção requerida, nem a função de T-Spin como bônus de pontuação.

A aplicação consiste de algumas classes, aproveitando as funcionalidades fornecidas pela GW. Há uma classe representando o Tetraminó, que herda de Game_Object, e que é responsável por gerenciar o comportamento isolado do tetraminó (como rotação, tipo, e cor da peça). Há uma classe representando os blocos que compõe o tetraminó. Para fins de otimização de desempenho, esta classe apenas herda de Sprite, de onde tira as informações de imagem. Como ela não tem funcionalidades intrísecas a ela que sejam servidas pela classe Game_Object, instanciar a classe Block como um Game_Object seria uma perda desnecessária de desempenho. Há uma classe Tétrion, onde se passa todo o

jogo, e que deve implementar as diretrizes especificadas em 4.2.3. Esta classe consome as configurações especificadas na "classe"Tétris, que é responsável por armazenar o modo de funcionamento do jogo. Como não herda de nenhuma classe disponibilizada pela GW, e Lua não disponibiliza a funcionalidade de classes apriori, Tétris é apenas um arquivo .lua que armazena configurações.

É importante frisar que mesmo numa aplicação de exemplo utilizando a GW, nem todos os comportamentos devem ser criados com base nas funcionalidades disponibilizadas pela Game Engine. Em jogos, a performance (nomeadamente, os frames por segundo) é tão importante quanto a jogabilidade e gráficos[1]. Um jogo que tenha todas as características em desenvolvidas (gráficos, jogabilidade, história, replay, sons e músicas, etc), mas que entrega performance ruim ou instável pode ser tão mal recebido quanto um jogo cheio de erros (bugs)[2]. Por isso, otimização é geralmente tratado como um sobrenome para jogos. Em jogos 3D, a otimização é geralmente toda concentrada no aspecto gráfico do jogo, devido a estes rodarem em, geralmente, processadores poderosos, com múltiplos núcleos, e uma placa de vídeo dedicada, configuração supostamente capaz de rodar qualquer aspecto lógico do jogo. As partes gráficas, no entanto, devido à alta complexidade para tentar reproduzir imagens realistas, requerem alto investimento em otimização. Em Set-Top Boxes, por seu baixo poder de processamento, temos que levar em consideração otimizações tanto a nível gráfico quanto a nível de processamento e memória. Por isso, sempre que possível, é bom evitar o uso das classes fornecidas pela GW, pelo fato de adicionarem peso computacional ao objeto. Elas somente devem ser usadas se suas funcionalidades forem essenciais ao funcionamento do objeto, como é o caso do Tetraminó e do Tétrion. A classe Bloco apenas precisa do funcionamento do Sprite, enquanto que o arquivo Tétris não precisa de nenhum dos dois, não herdando portanto nenhuma funcionalidade desnecessária.

Na figura 14 podemos ver um Diagrama de Classes simples que resolve o problema lógico do jogo Tétris. A seguir, explicamos o funcionamento de cada função, levando em consideração as diretrizes descritas em 4.2.3.

A classe mais simples, *Block*, é apenas um wrapper para Sprite. Não adiciona nenhum funcionamento extra, a não ser obter o gráfico dos blocos utilizado no jogo. A classe bloco armazena o sprite de todas as cores de blocos, distribuídos numa única imagem, e apenas escolhe qual parte da imagem será desenhada (o bloco em si). Este método de desenho é mais eficiente do que ter uma imagem para cada cor de bloco, uma vez que é mais rápido apenas escolher uma parte de uma imagem para ser desenhada do que mudar a imagem a ser desenhada.

Se utilizando da classe *Block*, temos a classe Tetraminó. Esta classe é responsável por determinar o formato do tetraminó (atributo *type*) e sua direção (atributo *direction*).

[1] <http://www.eurogamer.net/articles/digitalfoundry-2014-frame-rate-vs-frame-pacing>
[2] <http://www.dsogaming.com/editorial/top-5-worst-optimized-pc-games-of-2014/>

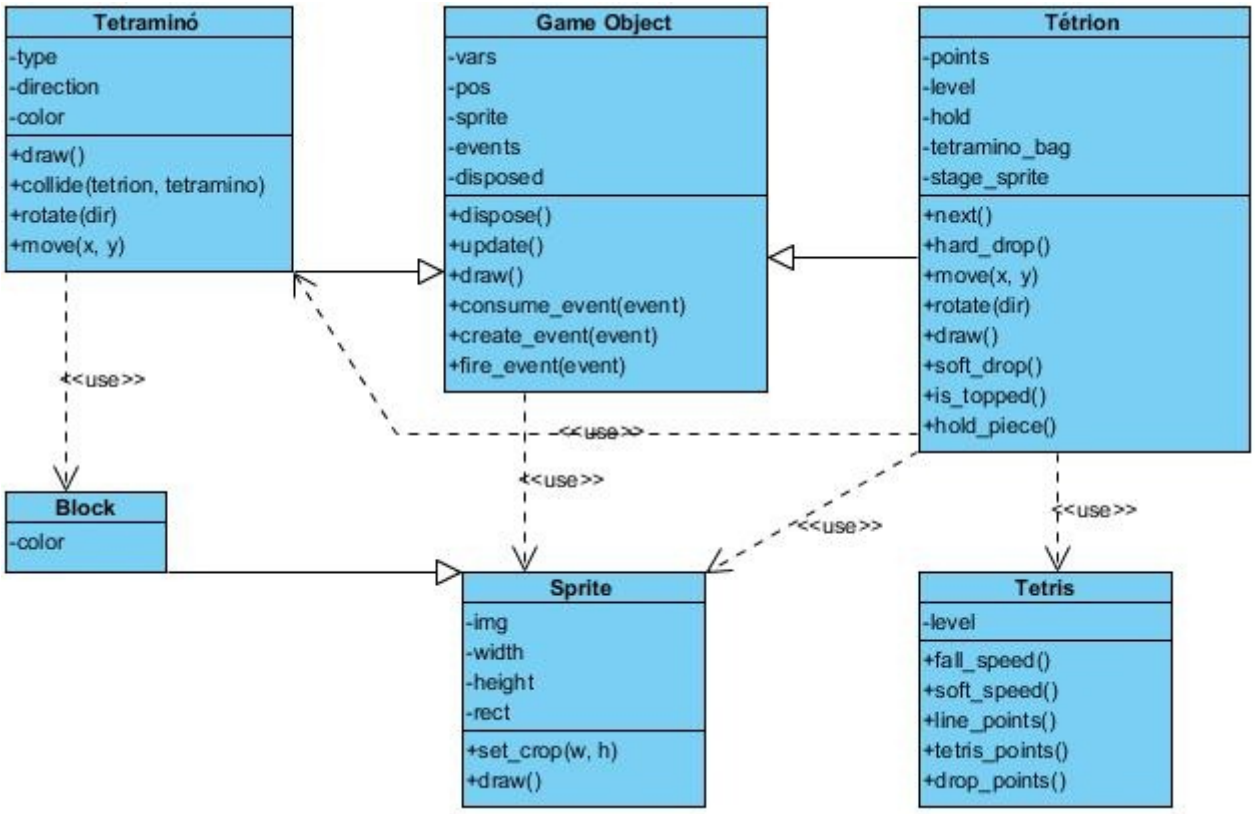

Figura 14 – Diagrama de Classes. Tetris

O formato da peça determina sua cor, e vice-versa. Os formatos são determinados pelas letras, descritos nas diretrizes. As direções são baixo, esquerda, acima, direita, numeradas de 0 a 3, respectivamente, tendo como direção inicial o valor 0, para satisfazer as diretrizes. O método *draw()* desenha blocos na tela, obedecendo o tipo e a direção. O método *collide(tetrion)* determina se este tetraminó colidiu com o tétrion de alguma forma. O método *rotate(dir)*, que recebe um valor positivo ou negativo, rotaciona a peça de acordo com a direção informada. E por fim, o método *move(x, y)*, que também recebe números positivos ou negativos, movimenta o tetraminó levando em conta as direções informadas. As movimentações sempre acontecem em 1 unidade, independente do valor. O que varia é a direção em que essa unidade é avançada. Nota-se que o método não se responsabiliza por validar se o movimento é válido. A estrutura de dados em que o tetraminó é armazenado é uma matriz 4x4, formada por números, 0 e 1, onde 0 significa não haver nenhum bloco ali, e 1 o contrário. Ela é preenchida levando em consideração o tipo e a direção.

A "classe"Tetris tem o propósito de armazenar as configurações gerais do jogo. Possui o atributo *level (nível)* que determina a dificuldade do jogo. O método *fall_speed()* determina o intervalo que a peça espera para cair 1 unidade automaticamente, levando em consideração o nível. O método *soft_speed()* determina a velocidade de queda da peça quando o jogador está utilizando o movimento *Soft Drop*, levando em consideração o nível. O método *line_points()* determina quantos o jogador recebe ao limpar uma linha. O

método *tetris_points()* determina quantos pontos o jogador recebe ao realizar um tetris (limpar 4 linhas de uma só vez). O método *drop_points()* determina uma pontuação para quando o jogador fixar um tetraminó no campo.

Por fim, a classe Tétrion tem como responsabilidade gerir todas essas outras classes de acordo com as diretrizes de jogo. O atributo *points* armazena os pontos obtidos pelo jogador, O atributo *lines* informa quantas linhas o jogador já limpou. O atributo *hold* é utilizado para a função *Hold* em jogos de Tetris, onde é possível guardar um tetraminó para uso tardio. O *tetramino_bag* armazena todas peças geradas randomicamente para serem colocadas em jogo. *stage_sprite* apenas guarda os sprites utilizados para desenhar o fundo. A função *next()* tira o próximo tetraminó de *tetramino_bag* e o coloca em campo. A função *hard_drop()* faz a peça cair automaticamente na mesma posição que está a *Ghost Piece*. A peça se fixa imediatamente após esse comando. O método *move(x, y)* requisita movimento do tetraminó, ao mesmo tempo em que valida sua nova posição, realizando algoritmos descritos no SRS para ajuste de posição. O método *rotate(dir)* requisita um movimento de rotação ao tetraminó, validando sua posição após e ajustando caso haja necessidade, utilizando o SRS. O método *draw()* é sobrescrito para desenhar todos os objetos na tela (o tetraminó, os próximos tetraminós, o *hold*, os pontos, linhas limpas, nível e imagens de fundo). O método *soft_drop()* acelera a velocidade de queda do tetraminó. Em níveis altos, usar esta função pode ter o mesmo efeito de *hard_drop()* por causa de uma velocidade muito alta. O método *is_topped()* verifica se a peça está presa no topo do campo, que é a condição de fim de jogo. O método *hold_piece()* requisita que o tetraminó atual seja colocado no campo *hold*. Somente uma peça que não saiu do *hold* pode ir para lá, para evitar trocas infinitas enquanto o jogador decide onde colocar a peça.

4.4 Resultados

5 Conclusões

Referências