INSTITUTO DE MATEMÁTICA E ESTATÍSTICA - IME
UNIVERSIDADE DE SÃO PAULO - USP

TRABALHO DE FORMATURA SUPERVISIONADO

Arcabouço para Jogos e Jogo Educacional em Ruby

Autor: Victor David Santos
Orientador: Prof. Dr. Marco Dimas Gubitoso

São Paulo, Dezembro de 2014

A educação é a arma mais poderosa que você pode usar para mudar o mundo.

Nelson Mandela

Resumo

Os principais objetivos do projeto aqui apresentado são destacar a importância de métodos alternativos de educação, tendo em vista a sabidamente pouco satisfatória situação dos índices educacionais no país, e apresentar uma ferramenta que pode ser usada como elemento integrante desses métodos: um jogo educacional.

Em caráter secundário, quase que como consequência imediata do trabalho demandado para o desenvolvimento da ferramenta proposta, figura também como parte deste projeto o arcabouço que foi utilizado para facilitar e agilizar a produção do jogo, e que consiste de uma biblioteca razoavelmente genérica, que foi disponibilizada para a comunidade como software livre, podendo auxiliar desenvolvedores na criação de jogos com a linguagem Ruby.

O jogo educacional desenvolvido foca-se em algumas das áreas mais fundamentais do conhecimento: Matemática, Idioma (neste caso, Língua Portuguesa) e Lógica. Ele é voltado para crianças nas primeiras séries do ensino fundamental, que compõem a base da formação educacional do indivíduo [1].

Palavras-chave: educação, jogos, arcabouço

Abstract

The main goals of the hereby presented project are evidencing the importance of conceiving alternative educational methods, given the well-known bad overall situation of educational levels in Brazil, and the presentation of a suitable tool that could be used as integrating part of such methods: an educational game.

Secondarily, as consequence of the efforts demanded for the development of the proposed tool, the framework used during this development is also part of this project. It consists of a fairly generic library, which was made available to the open source community, with the intent of helping developers to create games in Ruby language.

The educational game is focused on some of the most fundamental areas of knowledge: Mathematics, Language (Brazilian Portuguese, in this case) and Logics. It is aimed for children attending the first years of school, which represent the basis of the intellectual formation of individuals [1].

Keywords: education, games, framework

Sumário

I. Parte Objetiva

1. Introdução

A educação é o principal elemento que torna o ser humano tão distinto dos outros animais e também o que torna cada indivíduo tão distinto dos outros que o cercam. Afinal, a educação instiga no indivíduo a necessidade de compreender melhor o ambiente em que ele vive, e como os elementos interagem entre si neste ambiente; leva-o a se questionar sobre o que vê e o que sente, e a ir formando, pouco a pouco, suas conclusões e opiniões sobre tudo isso.

As fundações da sociedade humana estão também diretamente ligadas com a educação: as convenções sociais, as regras e as leis, somente podem ser corretamente interpretadas – e questionadas, quando necessário – por um indivíduo que teve suficiente base educacional, de modo a discernir apropriadamente as diretivas comportamentais que se adéquam a cada situação.

Não é surpreendente, pois, que as sociedades mais bem desenvolvidas no cenário mundial atual sejam marcadas, em geral, por uma forte valorização da educação: Nova Zelândia, Austrália, Noruega e Canadá são países que figuram entre os 8 primeiros, simultaneamente, no ranking de IDH (Índice de Desenvolvimento Humano) e no de educação [2][3]; os chamados Tigres Asiáticos são países que apresentaram um ritmo acelerado de crescimento nas últimas décadas, sendo que os investimentos em educação foram especialmente importantes para impulsionar este fenômeno [4].

Tendo isto em mente, não restam dúvidas do quão relevante é a educação na formação dos indivíduos e de uma sociedade bem organizada, e de como é direta a relação entre educação e desenvolvimento. Porém, investimentos por si só não conferem, necessariamente, sucesso a um sistema educacional. Uma grande parcela do sucesso está relacionada aos métodos utilizados. Nesse sentido, muitos especialistas das áreas de educação e psicologia têm apontado, dentre outras ideias que fogem ao "tradicional", a efetividade de jogos no auxílio ao aprendizado [5][6][7].

Assim, apresenta-se este trabalho como uma aplicação direta de pesquisas na área de educação, e especificamente na direção dos métodos alternativos de aprendizado, no intuito de fornecer uma contribuição real à sociedade neste quesito. Além disso, o arcabouço que também compõe este trabalho representa uma contribuição à comunidade de desenvolvedores de jogos e do software livre.

Entretanto, como ainda se trata de um trabalho de conclusão de curso de Ciência da Computação, não se pode deixar de percorrer os detalhes mais técnicos de todo o desenvolvimento que percorreu estes últimos meses e permitiu a apresentação deste material finalizado. A esta visão técnica se dedica a parte Objetiva do trabalho. Para uma melhor organização cronológica (e também concernindo uma relação de interdependência), serão primeiramente descritos os detalhes do supracitado arcabouço, para em seguida explorarem-se os modos como o mesmo contribuiu na construção de "Aventura do Saber", o jogo educacional.

2. MiniGL: um arcabouço para jogos em Ruby

2.1. A escolha de tecnologias

Dentre um vasto conjunto de linguagens de programação disponíveis, por que escolher Ruby? Esta linguagem apresenta, sabidamente, resultados muito pobres em desempenho quando comparada com outras como C, C++, ou mesmo Java [8], principalmente devido ao fato de ser interpretada, mas também por ser dinâmica e conter *garbage collector*, dentre outras características. Para a execução de tarefas de alta complexidade computacional, certamente esta limitação de desempenho tornaria inviável o uso de Ruby.

Porém, dentre outras várias características positivas desta linguagem, pelas quais passaremos mais adiante, está a própria comunidade de desenvolvedores, e a quantidade de bibliotecas voltadas aos mais diversos propósitos que se pode facilmente encontrar e instalar a partir de um repositório centralizado – o RubyGems (http://rubygems.org).

Felizmente, o interpretador Ruby foi concebido de tal forma que é possível integrar código C a um programa Ruby, através de extensões [9]. Em outras palavras, é possível delegar ao código de máquina altamente eficiente e otimizado, produzido por compiladores C, certas tarefas mais pesadas, enquanto se utiliza a sintaxe altamente amigável de Ruby.

Utilizando este paradigma, e por intermédio da biblioteca Gosu (http://www.libgosu.org), a qual encapsula funções da largamente utilizada biblioteca gráfica OpenGL, chegou-se à conclusão de que seria possível, sim, utilizar Ruby para construir um jogo com interface gráfica amigável e muita interatividade. Assim, verificada a viabilidade, e por várias razões que podem ser melhor esclarecidas durante a exposição das particularidades do código, Ruby foi escolhida; a biblioteca Gosu foi escolhida por ser a mais citada entre vários fóruns sobre desenvolvimento de jogos com Ruby.

Com as tecnologias bem definidas, poder-se-ia partir para o "trabalho pesado", iniciando a programação do jogo. Notou-se, no entanto, que usando a *gem* (nome dado às bibliotecas da linguagem Ruby) Gosu diretamente, certas tarefas se repetiam com frequência, e portanto poderia haver mais um nível de encapsulamento. Isto serviu de motivação para a criação do arcabouço MiniGL.

2.2. Explorando a biblioteca Gosu

Listaremos aqui quais funcionalidades a *gem* Gosu oferece, e quais ela não fornece, mas seriam desejáveis num arcabouço para jogos.

- Fornecidas:
  - gerenciamento da janela do jogo;
  - gerenciamento do laço principal do jogo (detalhes mais à frente);
  - desenho de primitivas (linhas, triângulos, retângulos) na tela;
  - desenho de imagens na tela;
  - desenho de texto na tela;
  - detecção do estado do teclado;
  - detecção do estado do mouse;

    - reprodução de música e sons.
- Desejáveis:
    - controle de animações;
    - controle de eventos específicos do teclado (tecla pressionada, tecla mantida pressionada, tecla solta, etc.);
    - controle de eventos específicos do mouse (botão pressionado, botão solto, duplo clique, etc.);
    - elementos básicos de interface de usuário (botões, campos de texto, etc.);
    - facilidades para desenho de texto (múltiplas linhas, alinhamento, etc.);
    - física básica (colisões, movimento baseado em forças, etc.);
    - gerenciamento de *viewport* (detalhes mais à frente);
    - gerenciamento de memória/recursos (imagens, fontes, sons, etc.);

Esta listagem evidencia alguns dos propósitos para a construção de uma biblioteca auxiliar, adicionando mais um nível de encapsulamento às funções comuns na programação de jogo, tornando o processo consideravelmente mais produtivo. Esclareçamos alguns dos conceitos próprios da área de desenvolvimento de jogos que foram apresentados acima, para melhor compreender de que modo eles afetam a programação propriamente dita:

- Laço principal do jogo: o princípio de funcionamento de todo jogo eletrônico é a execução de um laço infinito, interrompido apenas por alguma ação explícita do usuário para terminar o jogo; neste laço, são executadas essencialmente duas tarefas a cada iteração: atualização e desenho (geralmente associadas a métodos *update* e *draw*, respectivamente). A *gem* Gosu oferece uma interface simples para controle deste laço.
- Viewport: este é o termo comumente usado para se referir ao conteúdo da janela do jogo, ou seja, à parcela visível dos elementos do jogo. Um elemento comum a diversos tipos de jogo é a necessidade de se alterar a perspectiva pela qual o personagem visualiza o cenário e os elementos – algo como movimentar a "câmera" –, que faz sentido sempre que nem todos os elementos cabem simultaneamente na tela. Não há, na biblioteca Gosu, classes ou métodos que permitam uma fácil alteração de perspectiva em relação a um cenário maior.

Todos os elementos listados como "desejáveis" são tratados, com maior ou menor completude, pela biblioteca apresentada como parte deste trabalho, batizada de "MiniGL" (uma abreviação para *Minimal Game Library*).

2.3. Um pouco de MiniGL

A finalidade deste tópico não é servir como um tutorial para se aprender a utilizar a gem MiniGL, mas apresentar os aspectos técnicos que explicam como e por que ela funciona. Estes aspectos serão enumerados e explorados em subtópicos, cada um dizendo respeito a um dos itens da listagem anterior de características "desejáveis". Em geral, serão utilizadas comparações entre trechos de código com e sem a utilização de MiniGL, evidenciando os pontos em que se alcança uma maior sucintez e/ou clareza.

2.3.1. Controle de animações

Um elemento extremamente comum em jogos é a exibição de objetos ou personagens animados, isto é, elementos representados por uma imagem que vai se alterando, sutilmente, com o passar do tempo, conferindo-lhes um aspecto de "movimento".

A utilização deste recurso é especialmente apropriada quando se trata do personagem principal do jogo: em geral, este poderá executar diversas ações, sob comando do jogador, as quais podem ser visualmente representadas por animações. Ao dar o comando de "andar para a frente", por exemplo, o jogador espera ver seu personagem mexer as pernas alternadamente, para a frente e para trás, enquanto se desloca na direção para a qual está voltado.

Considerando, além disso, que o personagem principal pode interagir com diversos outros elementos, os quais devem "responder" de alguma forma à sua interação, o uso de animações se torna frequente para muitos elementos. Por outro lado, sua lógica é sempre muito parecida: desenha-se a imagem atualmente associada ao objeto, e modifica-se esta associação para outras imagens previamente carregadas, a cada intervalo de *n* quadros (um "quadro" corresponde aproximadamente a 1/60 de segundo, ou 16,67 milissegundos); quanto menor for *n*, mais rápida parecerá a animação.

Tendo em vista o exposto acima, verifiquemos um trecho de código que utiliza a biblioteca Gosu para alcançar o efeito de animar constantemente um objeto qualquer, utilizando três imagens diferentes:

```
require 'gosu'
class MyGame < Gosu::Window
  def initialize
    super 800, 600, false # janela de 800 por 600 pixels, não fullscreen
    @imgs = Gosu::Image.load_tiles(self, “imagem.png”, 60, 120)
    # ou
    # @imgs = Gosu::Image.load_tiles(self, “imagem.png”, -3, -2)
    @timer = 0
    @interval = 10
    @indices = [0, 1, 0, 2]
    @index_index = 0
    @index = @indices[@index_index]
  end

  def update
    @timer += 1
    if @timer == @interval
      @index_index += 1
      @index_index = 0 if @index_index == @indices.length
      @index = @indices[@index_index]
      @timer = 0
    end
  end

  def draw
    @imgs[@index].draw 0, 0, 0
  end
end
```

```
game = MyGame.new
game.show
```

No exemplo acima, nota-se o carregamento de um vetor de imagens com `load_tiles`, ao qual é passado o nome de um único arquivo de imagem. Isto porque o arquivo será interpretado como uma *spritesheet*, ou seja, uma única imagem que contém várias "subimagens", cada uma representando um passo da animação. Segue exemplo abaixo:

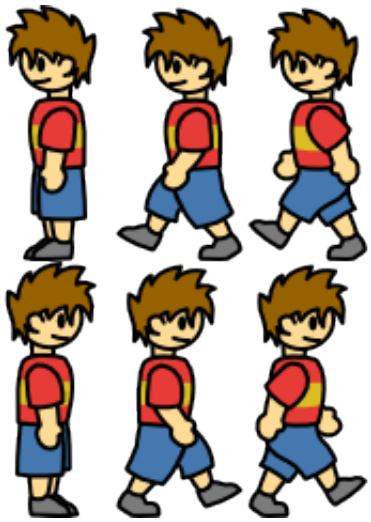

Figura 1: exemplo de spritesheet

Também são passados para `load_tiles` os parâmetros 60 e 120, ou -3 e -2. Isto indica que cada imagem usada na animação terá 60 por 120 pixels, ou que a imagem fornecida será dividida em 3 colunas e 2 linhas (portanto deve ter tamanho total 180 por 240 pixels). Estes valores descrevem como deve ser carregada a imagem acima.

Nota-se também a definição de uma classe que herda de `Gosu::Window`, e implementa métodos `initialize`, `update` e `draw`. Este é o formato padrão de um programa que utiliza Gosu. O padrão é mantido com a utilização de MiniGL, de forma que a estrutura fica familiar a desenvolvedores que porventura já viessem utilizando Gosu:

```
require 'minigl'
include AGL

class MyGame < Gosu::Window
  def initialize
    super 800, 600, false
    Game.initialize self
    @sprite = Sprite.new 0, 0, :imagem, 3, 2
  end

  def update
    # A contagem de índices na spritesheet é feita da esquerda para a
    # direita, de cima para baixo, iniciando em 0.
    @sprite.animate [0, 1, 0, 2], 10
  end
```

```
  def draw
    @sprite.draw
  end
end

game = MyGame.new
game.show
```

O trecho de código acima exposto provê exatamente o mesmo resultado durante a execução – a imagem do menino, virado para a esquerda e movimentando as pernas alternadamente –, e, no entanto, ele é perceptivelmente mais sucinto. Observemos com maior minúcia as diferenças:

- No lugar de um simples vetor de imagens, utiliza-se uma instância da classe `Sprite`.
  - Esta classe guarda a posição do objeto (coordenadas (0, 0), neste caso), de modo que na chamada `@sprite.draw`, nenhum parâmetro é necessário.
  - Esta classe encapsula a lógica anteriormente controlada pelas variáveis `@index`, `@timer`, `@interval`, `@indices` e `@index_index` no método `animate` (cujos parâmetros são a sequência de índices usados na animação e o intervalo entre cada troca de índice). O corpo da função `update` ficou reduzido a uma única linha de código.
- Para permitir o uso das funcionalidades da MiniGL, a chamada `Game.initialize`, que recebe como parâmetro a janela do jogo, se faz necessária (verificaremos mais adiante os motivos).

A *gem* MiniGL oferece ainda, no âmbito de controle de animações, a classe `Effect`. Esta representa, basicamente, uma `Sprite` com sequência de animação e tempo de vida pré-definidos, sendo ideal para a exibição de efeitos visuais na tela. Sua utilização é bastante semelhante à de `Sprite`, mas os parâmetros de animação são fornecidos diretamente no construtor. Mais detalhes podem ser vistos no tutorial da biblioteca (esse e outros recursos são listados no final deste documento).

2.3.2. Controle de eventos específicos do teclado

Em jogos de computador, é fundamental o controle da interação do usuário com o teclado. É, além disso, muito comum a necessidade de que o jogo processe uma ação quando uma tecla é pressionada, mas não a repita no caso de a tecla ser mantida pressionada, por exemplo. Também é possível pensar em cenários em que uma certa ação deve ser tomada quando uma tecla que estava pressionada é solta. Para permitir a fácil detecção de eventos como esse, a classe `KB` da *gem* MiniGL fornece vários métodos.

Vejamos um exemplo das dificuldades enfrentadas para evitar que uma ação seja repetida diversas vezes com um único apertar de tecla do usuário. Vale notar que, mesmo que o usuário aperte a tecla e a solte logo em seguida, é muito provável que o estado da tecla seja considerado como "pressionado" (ou *down*) por vários quadros, já que estes duram apenas cerca de 17 milissegundos. As linhas de importação de biblioteca (`require …`, `include …`), assim como as de declaração, instanciação e exibição da janela (`class MyGame …`, `game = MyGame.new`, `game.show`) serão omitidas deste exemplo em diante, já que não sofrerão alterações:

```
def initialize
  super 800, 600, false
  @key_down = false
  @imgs = Gosu::Image.load_tiles self, “imagem.png”, false, -3, -2
  @index = 0
end

def update
  if button_down? Gosu::KbSpace
    unless @key_down
      @index += 1
      @index = 0 if @index == @imgs.length
      @key_down = true
    end
  else
    @key_down = false
  end
end

def draw
  @imgs[@index].draw 0, 0, 0
end
```

O objetivo do código acima é semelhante ao do exemplo da seção 2.3.1, carregando e exibindo a mesma *spritesheet*, porém em vez de a animação ser feita automaticamente, troca-se a imagem cada vez que o usuário pressiona a barra de espaço.

Não parece muito complicado, mas se imaginarmos fazer esse controle para cada tecla e cada evento que precisa dessa checagem, rapidamente teríamos um código muito longo e repetitivo – a não ser, é claro, que encapsulássemos esta lógica num módulo ou função, e é justamente isso que a classe `KB` de MiniGL faz:

```
def initialize
  super 800, 600, false
  Game.initialize self
  @imgs = Res.imgs “imagem”, 3, 1
  @index = 0
end

def update
  KB.update
  if KB.key_pressed? Gosu::KbSpace
    @index += 1
    @index = 0 if @index == @imgs.length
  end
end

def draw
  @imgs[@index].draw 0, 0, 0
end
```

Ao lado de `key_pressed?`, a classe `KB` fornece também os métodos `key_down?` (que corresponde ao `button_down?` da Gosu), `key_released?` (que retorna `true` apenas no quadro em

que uma tecla que estava pressionada é solta) e `key_held?` (que indica que uma tecla está pressionada já há um certo intervalo de tempo).

2.3.3. Controle de eventos específicos do mouse

Situações parecidas com aquelas descritas no item anterior, que dizem respeito ao teclado, também se aplicam, de modo muito similar, ao mouse. Poderíamos usar dois trechos de código praticamente idênticos aos do exemplo acima para demonstrar como o uso de MiniGL contribuiria para um código mais sucinto – no lugar de `key_pressed?`, por exemplo, seria utilizado o método `button_pressed?`, da classe `Mouse`; a *gem* Gosu utiliza o mesmo método, `button_down?`, para detectar o estado dos botões do mouse, passando como parâmetro constantes diferentes, como `Gosu::MsLeft`.

Assim, para um cenário como o ilustrado acima, o ganho com o uso de MiniGL seria similar àquele conseguido para o cenário do teclado. Porém, adicionalmente, MiniGL oferece ainda a possibilidade de detectar cliques duplos do mouse com uma única chamada de método, enquanto que utilizando as interfaces oferecidas pela Gosu, a lógica seria ainda mais complicada de implementar. Isso pode ser evidenciado pelo trecho de código abaixo, extraído do próprio código fonte da *gem* MiniGL:

```
class Mouse
  def self.initialize
    ...
    @@dbl_click = {}
    @@dbl_click_timer = {}
  end

  def self.update
    ...
    @@dbl_click.clear

    @@dbl_click_timer.each do |k, v|
      if v < Game.double_click_delay; @@dbl_click_timer[k] += 1
      else; @@dbl_click_timer.delete k; end
    end

    k1 = [Gosu::MsLeft, Gosu::MsMiddle, Gosu::MsRight]
    k2 = [:left, :middle, :right]
    for i in 0..2
      if Game.window.button_down? k1[i]
        @@down[k2[i]] = true
        @@dbl_click[k2[i]] = true if @@dbl_click_timer[k2[i]]
        @@dbl_click_timer.delete k2[i]
      elsif @@prev_down[k2[i]]
        @@dbl_click_timer[k2[i]] = 0
      end
    end

    ...
  end
  ...
end
```

Todas estas variáveis e esta lógica são necessárias para permitir que o usuário de MiniGL detecte um duplo clique do botão esquerdo do mouse, por exemplo, com:

```
if Mouse.double_click? :left
  # tratar duplo clique aqui
end
```

É válido notar que o *delay* máximo, em quadros, entre os dois cliques do usuário, para que ainda se considere um duplo clique é um valor configurável através da variável `Game.double_click_delay`. Como veremos mais adiante, há mais alguns elementos que podem ser configurados na biblioteca através da classe `Game`. De fato, a intenção é que haja cada vez mais possibilidades de configuração – esta diretiva será seguida para a continuidade e lançamento das próximas versões da *gem*.

2.3.4. Elementos básicos de interface de usuários

Praticamente todo sistema de computador com interface gráfica possui elementos gráficos através dos quais o usuário pode interagir com o sistema, navegando entre as telas, alterando opções, salvando e carregando arquivos, etc. Estes elementos são muito recorrentes entre os diversos tipos de sistema: é difícil imaginar um software com interface gráfica que não use botões ou campos de texto, por exemplo.

Os jogos eletrônicos não são exceção a essa característica: neles também é tarefa recorrente a necessidade de mostrar botões para o usuário – iniciar o jogo, carregar jogo salvo, exibir as opções e sair do jogo são algumas das ações que o usuário comumente executará clicando em botões.

Como é também frequente em jogos a solicitação de identificação do jogador (pede-se o nome do jogador para, por exemplo, facilitar que vários usuários salvem seus jogos numa mesma máquina e possam continuá-los sem interferir no jogo dos outros), campos de texto são um outro elemento de interface bastante útil – e particularmente complicado de programar "do zero".

Por fim, como muitos jogos apresentam opções do tipo "sim/não", ou "ligado/desligado", mostra-se conveniente, nestas situações, a utilização de um controle que alterna entre dois estados (a que se pode referir por *checkbox* ou *toggle button*).

Levando em conta a recorrência dos elementos descritos acima, decidiu-se por incluir na biblioteca MiniGL classes que permitissem a fácil exposição e manipulação de tais objetos. Vejamos a seguir uma amostra do uso da classe `Button` – neste caso, não faremos comparação com código utilizando apenas a biblioteca Gosu, pois este se mostraria consideravelmente extenso, e a mesma lógica pode ser vista no código-fonte da *gem* MiniGL.

```
def initialize
  ...
  font = Res.font :font1, 20
  @btn = Button.new 0, 0, font, "Clique aqui", :button1 {
    puts "clique"
  }
end
```

```
def update
  @btn.update
end

def draw
  @btn.draw
end
```

Enumeremos passo a passo o significado das linhas de código:

- Primeiro, um objeto do tipo `Gosu::Font` é instanciado a partir da chamada a `Res.font`. Este objeto representa uma fonte, ou seja, é usado para desenhar texto na tela (veremos outros usos para estes objetos mais adiante).
- Em seguida, instancia-se um botão propriamente dito, nas coordenadas (0, 0), utilizando a fonte `font`, com texto "Clique aqui"; o símbolo `:button1` codifica o caminho "data/img/button1.png", onde deve estar uma *spritesheet* contendo imagens para os diferentes estados do botão (ver figura 2); o bloco de código passado para o construtor corresponde à ação que será executada quando o botão for clicado (aqui é notável a contribuição das *closures* de Ruby para permitir esta sintaxe limpa e intuitiva).

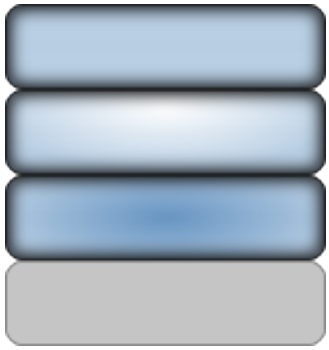

Figura 2: Exemplo de spritesheet *para botão. A primeira imagem corresponde ao estado padrão, a segunda é usada quando o cursor do mouse está sobre o botão, a terceira quando o botão está pressionado e a quarta quando o botão está desabilitado.*

- A chamada `@btn.update` cuida do gerenciamento de todos os eventos do botão.
- A chamada `@btn.draw` desenha o botão na tela.

Essa simples sequência de quatro ações produz como efeito um botão plenamente funcional sendo exibido na tela do jogo. Mas ainda maior é a contribuição dada pela MiniGL no caso de campos de texto. A utilização se dá de modo bastante semelhante à dos botões, instanciando um objeto da classe `TextField` e chamando seus métodos `update` e `draw`. Porém, a lógica encapsulada por esta classe é particularmente complexa, conforme citado anteriormente. O trecho de código a seguir corresponde apenas ao tratamento das interações com o campo de texto através do mouse:

```
def update
  ...
  if Mouse.over? @x, @y, @w, @h
    if not @active and Mouse.button_pressed? :left
      focus
```

```
    end
  elsif Mouse.button_pressed? :left
    unfocus
  end

  return unless @active

  if Mouse.double_click? :left
    if @nodes.size > 1
      @anchor1 = 0
      @anchor2 = @nodes.size - 1
      @cur_node = @anchor2
      @double_clicked = true
    end
    set_cursor_visible
  elsif Mouse.button_pressed? :left
    set_node_by_mouse
    @anchor1 = @cur_node
    @anchor2 = nil
    @double_clicked = false
    set_cursor_visible
  elsif Mouse.button_down? :left
    if @anchor1 and not @double_clicked
      set_node_by_mouse
      if @cur_node != @anchor1; @anchor2 = @cur_node
      else; @anchor2 = nil; end
      set_cursor_visible
    end
  elsif Mouse.button_released? :left
    if @anchor1 and not @double_clicked
      if @cur_node != @anchor1; @anchor2 = @cur_node
      else; @anchor1 = nil; end
    end
  end
  ...
end

def focus
  @active = true
end

def unfocus
  @anchor1 = @anchor2 = nil
  @cursor_visible = false
  @cursor_timer = 0
  @active = false
end

def set_node_by_mouse
  index = @nodes.size - 1
  @nodes.each_with_index do |n, i|
    if n >= Mouse.x
      index = i
```

```
      break
    end
  end
  if index > 0
    d1 = @nodes[index] - Mouse.x; d2 = Mouse.x - @nodes[index - 1]
    index -= 1 if d1 > d2
  end
  @cur_node = index
end
```

Embora não fique muito claro apenas olhando, o código acima é responsável por detectar e tratar as seguintes ações:

- Se o usuário clica fora do campo de texto, ele perde o foco (de modo que ao digitar, o texto não é inserido no campo, e o cursor piscante não é mais exibido).
- Se o usuário clicar no campo de texto, ele ganha o foco, caso já não o tivesse, e o cursor é posicionado junto ao caractere mais próximo da posição do clique.
- Se o usuário dá um duplo clique no campo de texto, todo o texto ali contido é selecionado.
- Se o usuário clica e arrasta o mouse ao longo do campo, o texto vai sendo selecionado conforme é atravessado pelo cursor.

Vale frisar que o código acima, além de não representar nem mesmo metade de toda a lógica encapsulada pela classe `TextField`, é construído já se utilizando das facilidades oferecidas pela classe `Mouse` da biblioteca MiniGL, o que o torna ainda mais distante de uma implementação utilizando apenas a biblioteca Gosu.

Também faz parte deste conjunto de classes de elementos de interface a classe `ToggleButton`, que representa um *checkbox* ou *toggle button*, conforme apresentados anteriormente. Esta é uma classe que herda de `Button`, pois funciona de modo muito semelhante. As diferenças são basicamente:

- O bloco passado para o construtor (que indica a ação a ser executada no evento de clique) receberá sempre um parâmetro cujo valor alterna entre `true` e `false` ao longo de chamadas sucessivas, o que corresponde à alternância entre dois estados do objeto (“sim” e “não”, “ligado” e “desligado”, conforme for o caso).
- As instâncias desta classe expõem um atributo `checked`, que permite a verificação do estado atual do botão.
- A *spritesheet* a ser fornecida para instanciação de um `ToggleButton` deve seguir um padrão um pouco mais complicado que aquele usado para botões comuns – isto porque os estados intrínsecos (“ligado” e “desligado”) combinados com os quatro estados relativos à interação com o mouse resultam em oito estados no total, cada um tendo a possibilidade de ser retratado com uma imagem diferente. Ver figura 3.

Figura 3: Exemplo de spritesheet *para um* toggle button *ou* checkbox. *A primeira coluna corresponde a estados em que* `checked` *retorna* `false`*, a segunda, àqueles em que* `checked` *retorna* `true`*.*

2.3.5. Facilidades para desenho de texto

Claramente, texto é fundamental para completar o tipo de interface gráfica que descrevemos acima – por exemplo, compondo o rótulo dos botões, mostrando o texto atual das caixas de texto, etc. –, assim como pode ser muito útil mesmo nas telas principais do jogo (ou seja, onde as ações do jogador tomam lugar): pode ser usado para exibir o nome e a pontuação do jogador atual, para mostrar instruções de como jogar, explicando elementos da tela, exibir diálogos entre o jogador e outros personagens do jogo, e uma infinidade de outros usos possíveis.

As funcionalidades básicas para exibição de texto na tela são já oferecidas pela biblioteca Gosu, mas há cenários que demandam modos específicos de manipulação do texto, especialmente em se tratando de texto com múltiplas linhas, em que apenas estas funções básicas acabam deixando o trabalho um pouco mais complicado do que o desejável. Mais uma vez, a *gem* MiniGL atua neste sentido para tornar tudo mais simples.

Abaixo pode-se observar um possível exemplo de como criar uma função que exibe texto com múltiplas linhas respeitando as quebras de linha (isto é, caracteres "\n" que apareçam dentro da *string* a ser renderizada). Sem o uso de MiniGL, é necessário criar uma função auxiliar, pois a função `draw` dos objetos `Gosu::Font` (que representam fontes com as quais se pode escrever texto) não aceita quebras de linha:

```
def write_breaking font, text, x, y
  lines = text.split "\n"
  lines.each do |l|
    next if l.empty?
    Font.draw l, x, y, 0
    y += font.height
  end
end

# exemplo de utilização ('font' é uma instância de Gosu::Font):
write_breaking font, "linha 1\nlinha 2\nlinha 3", 0, 0
```

Note-se que processar caracteres "\n" é apenas a parte mais simples das implicações de se escrever texto quebrando linha: em geral, gostaríamos que houvesse um limite horizontal para as linhas, de modo que as quebras fossem feitas automaticamente quando o texto atingisse

este limite, ainda que não houvesse uma quebra explícita na *string* – como quando usamos um editor de texto. Com o uso de MiniGL, o código da função e a chamada acima poderiam ser substituídos pelas seguintes duas linhas:

```
helper = TextHelper.new font
helper.write_breaking “linha 1\nlinha 2\nlinha 3”, 0, 0, 800
```

A classe `TextHelper`, apresentada acima, encapsula não somente o processamento dos caracteres “\n”, mas também o tratamento de limite de tamanho para as linhas (descrito no parágrafo anterior), motivo pelo qual o último parâmetro da chamada a `write_breaking` é necessário: ele especifica qual o limite de tamanho, em pixels, para as linhas do texto. Neste caso, usamos 800, que pode ser a largura total da tela, de modo que o texto ocupará todo o espaço horizontal disponível antes de quebrar linha.

Além do descrito acima, a função `write_breaking` de `TextHelper` tem ainda mais capacidades: ela pode receber parâmetros para a cor e a opacidade do texto (neste caso, não representa grande diferença com relação à função `draw` de `Gosu::Font`) e, mais importante, para o alinhamento. Mais uma vez, fazendo alusão a editores de texto – não editores de texto puro, mas de documentos, com formatação – são comuns os modos de alinhamento à esquerda, à direita, centralizado e justificado. Assim, o quinto parâmetro de `write_breaking` assume o valor padrão `:left`, que especifica alinhamento à esquerda, mas aceita os valores `:right`, `:center` e `:justified`, que especificam cada um dos outros três modos de alinhamento enumerados, respectivamente.

O código a seguir ilustra a exploração de todos os modos de alinhamento aceitos pela classe `TextHelper`:

```
def initialize
  ...
  @font = Res.font :font1, 20
  @th = TextHelper.new @font
end
...
def draw
  # para texto de uma linha, usar 'write_line'
  @th.write_line "Here's some left-aligned text...", 10, 200, :left,
                 0xffffff
  @th.write_line "...and some centered text...", 400, 240, :center,
                 0xffffff
  @th.write_line "...and some right-aligned text too!", 790, 280, :right,
                 0xffffff
  @th.write_breaking "This is a justified paragraph. Yeah, that's right,
you define a width and all lines will have that same width.\nWell, except
for the last line in each paragraph...", 10, 320, 300, :justified, 0xffffff
end
```

Este trecho relativamente curto de código produz o seguinte na janela do jogo:

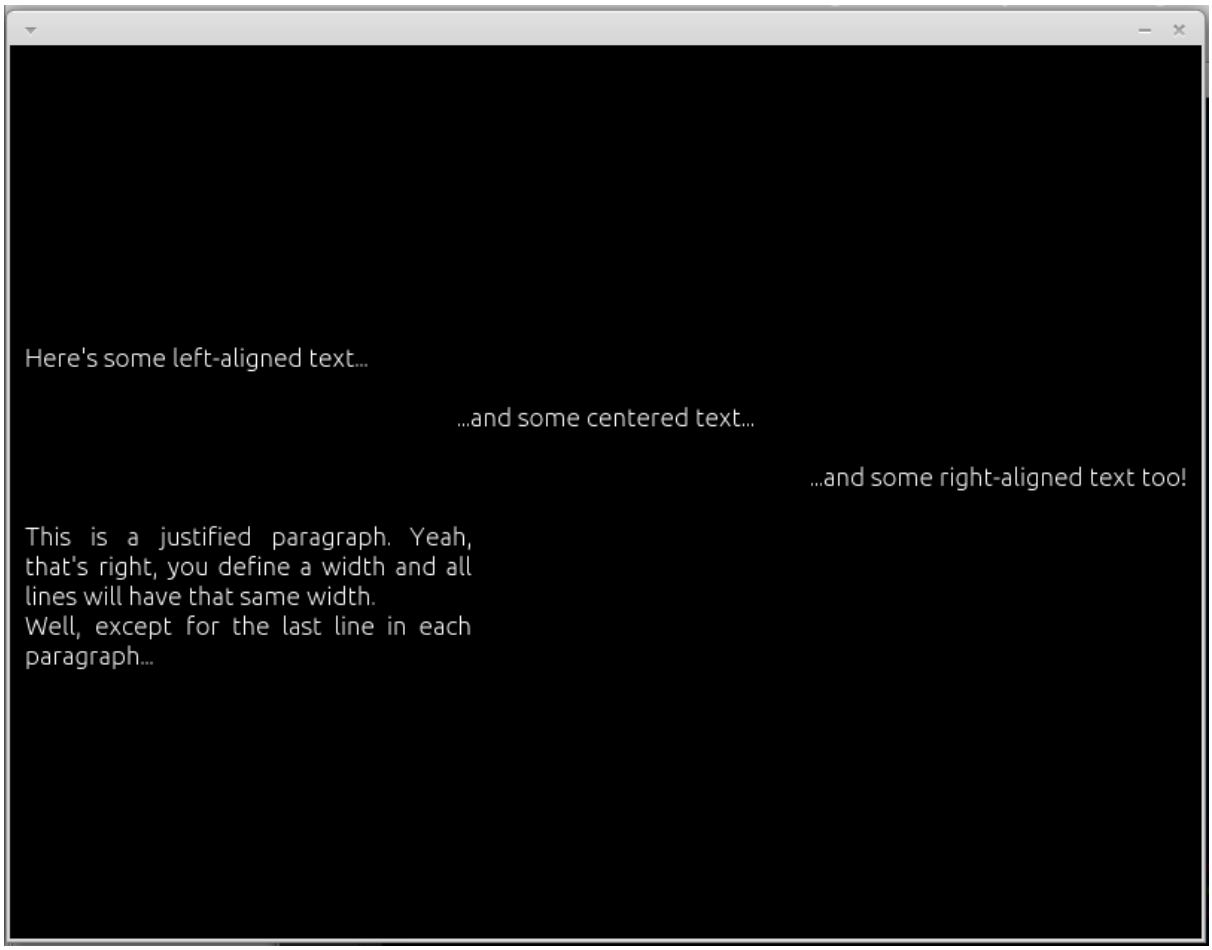

Figura 4: Utilizando a classe `TextHelper` para desenhar texto com diferentes modos de alinhamento.

2.3.6. Física básica

As funcionalidades da biblioteca que serão apresentadas nesta seção são provavelmente as menos gerais, no sentido de que podem ser completamente dispensadas em certos gêneros de jogo; não obstante, numa boa parcela dos gêneros, pode-se tirar proveito dessas funções, uma vez que permitem a manipulação de um fator inerente a praticamente todo jogo que tenta simular de alguma maneira o mundo real: física.

Mais especificamente falando, trata-se da Mecânica – isto é, forças e movimento. É extremamente comum em jogos a necessidade de se detectar colisão entre objetos. Mas, mais do que isso, em muitos casos é necessário apresentar uma resposta adequada à colisão, isto é, não permitir que os objetos que colidiram "atravessem" um ao outro, reposicionando um deles ou ambos de modo que se mantenham em contato, mas respeitando os limites um do outro.

Um modelo simples e bastante utilizado para a checagem de colisões é o de caixas de colisão retangulares. Neste modelo, os limites físicos de todos os objetos são aproximados para retângulos. Embora esta aproximação possa parecer bastante grosseira, na prática podem-se conse-

guir resultados bastante satisfatórios, considerando parâmetros bem comportados para grandezas como as forças, velocidades e tamanhos dos objetos envolvidos.

A figura a seguir demonstra esquematicamente como funciona a detecção e resposta à colisão num modelo de caixas retangulares.

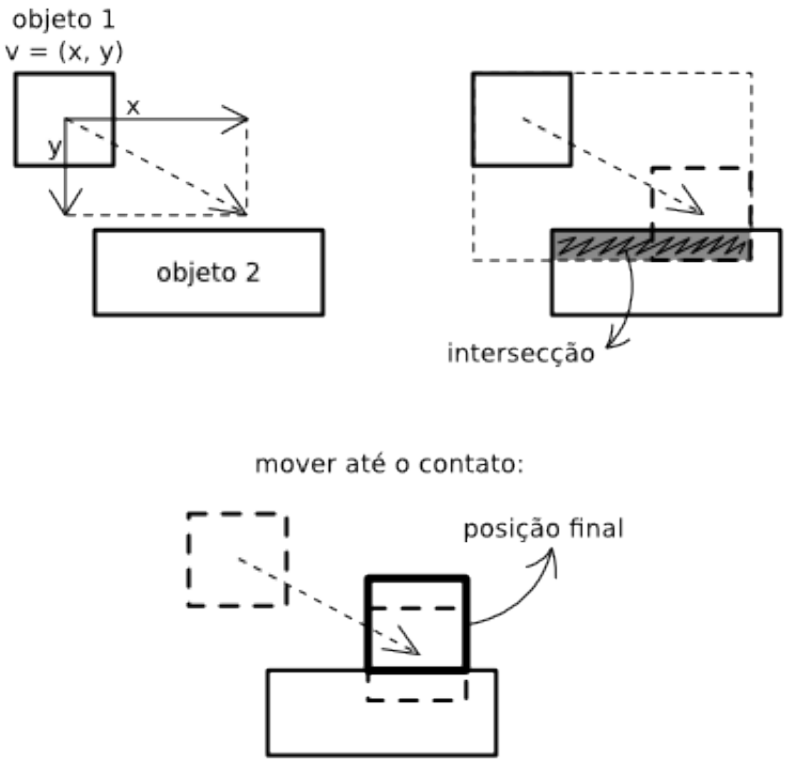

Figura 5: Esquema da checagem e resposta a colisão com caixas retangulares.

Na figura acima, distinguem-se três passos, a saber:

1) Projeta-se a nova posição hipotética do objeto que vai se mover, baseando-se nas componentes da velocidade;

2) Obtém-se um retângulo correspondente a todo o espaço ocupado durante o movimento. Na verdade, o retângulo extrapola um pouco o espaço que de fato seria varrido no movimento, mas, para valores pequenos de velocidade, esta diferença acaba sendo desprezível. Como as checagens de colisão são feitas a cada quadro, e a cada segundo são executados cerca de sessenta quadros, nota-se que, em geral, as velocidades serão de fato baixas – um objeto se movendo a 100 pixels por quadro, por exemplo, atravessa toda a extensão de uma tela horizontal de 800 pixels de largura em pouco mais de 0,13 segundos, ou seja, seu movimento mal é perceptível aos nossos olhos;

3) Se o retângulo projetado intersecta o retângulo corresponde aos limites físicos do outro objeto, verifica-se em qual direção o movimento deve ser limitado para manter o máximo possí-

vel do movimento original, e corrige-se a posição final para apenas "encostar" no outro objeto, na mesma faixa horizontal ou vertical onde ocorreu a intersecção.

Agora que compreendemos os fundamentos teóricos desse modelo de checagens de colisão, vejamos um pouco da utilização prática, com exemplos de código. Anteriormente a isso, todavia, é importante que sejam introduzidas algumas outras classes fornecidas pela biblioteca que desempenham papel de objetos físicos e que serão, portanto, aquelas que serão manipuladas pelo método de checagem de colisão descrito acima.

Primeiramente, apresentemos a classe `GameObject`. Esta classe, derivada de `Sprite`, adiciona à classe base características de corpo físico, como largura e altura (que juntamente com os atributos de posição, x e y, definem o retângulo delimitador do objeto), massa, velocidade, etc. Tais atributos são necessários para que se possa utilizar os métodos de movimentação baseada em forças e com checagem de colisões.

Por outro lado, alguns dos objetos envolvidos na checagem de colisão desempenham mais o papel de obstáculos, isto é, não são móveis, mas ocupam um certo espaço que deve ser respeitado (ou seja, que não pode ser atravessado). Por isso, em segundo, apresentamos a classe `Block`. Esta é uma classe realmente muito simples, cujo papel principal é armazenar os atributos necessários à definição de uma caixa de colisão retangular. Adicionalmente, ela fornece o atributo `passable`, que, quando definido como `true`, define um comportamento especial para o bloco em questão: objetos móveis poderão atravessar o bloco, exceto quando vindo de "cima", que em termos do espaço bidimensional representado na tela do jogo significa que a coordenada y é crescente ao longo do movimento (este comportamento é útil especificamente para jogos do gênero plataforma).

Por fim, participa também deste subconjunto de classes envolvidas em movimento e checagens de colisão a classe `Ramp`. Aqui também, tem-se um elemento primordialmente associado a jogos de plataforma: instâncias da classe `Ramp` representam rampas, isto é, inclinações no terreno através das quais um objeto pode passar de regiões mais "baixas" (maiores coordenadas y) para regiões mais "altas" (menores coordenadas y), sem que precise exercer explicitamente uma força para cima (o que corresponderia, por exemplo, ao personagem dar um pulo).

De posse destas informações preliminares, pode-se agora proceder à observação e detalhamento de um trecho de código que se utiliza de todas estas funcionalidades:

```
def initialize
  ...
  @obj = GameObject.new 0, 0, 50, 50, :obj
  @obstacles = [
    Block.new(0, 500, 700, 10, true),
    Block.new(700, 0, 10, 500, false)
  ]
  @ramps = [
    Ramp.new(600, 450, 100, 50, true)
  ]
end

def update
  forces = Vector.new(0.1, 0)
  @obj.move forces, @obstacles, @ramps
```

```
end

def draw
  @obstacles.each do |o|
    draw_quad o.x, o.y, 0xffffffff,
              o.x + o.w, o.y, 0xffffffff,
              o.x + o.w, o.y + o.h, 0xffffffff,
              o.x, o.y + o.h, 0xffffffff, 0
  end
  @ramps.each do |r|
    draw_triangle (r.left ? r.x + r.w : r.x), r.y, 0xffffffff,
                  r.x + r.w, r.y + r.h, 0xffffffff,
                  r.x, r.y + r.h, 0xffffffff, 0
  end
  @obj.draw
end
```

Ao executar o "jogo" definido pelo código acima, poder-se-ia ver um objeto (visualmente representado pela imagem que estiver sendo identificada por `:obj`), partindo do canto superior esquerdo da tela, "cair" com a força da gravidade até o solo (representado por um retângulo branco), enquanto se move para a direita com velocidade crescente. Ao chegar mais próximo a uma parede, mais à direita da tela (também representada por um retângulo branco), o objeto subiria numa rampa até parar de se mover, em contato com a parede. É notável a sucintez e simplicidade do código que produziu este resultado. Cabem, além disso, as seguintes observações:

- A força da gravidade é sempre adicionada ao vetor de forças fornecido ao método move. Porém, para tornar o método mais genérico, esta força é configurável através do atributo `gravity` da classe `Game`. Definindo-a como zero (`Vector.new(0, 0)`), por exemplo, a movimentação do objeto será determinada unicamente pelas forças passadas como parâmetro.
- A classe `GameObject` aceita alguns parâmetros adicionais no construtor, não mostrados no exemplo acima. São eles:
  - `img_gap`: deve receber um `Vector`, indicando o deslocamento entre a posição física do objeto e a posição da imagem. Quando não é especificado, o vetor (0, 0) é considerado. A importância da utilização deste parâmetro é exemplificada na figura 6.
  - `sprite_cols` e `sprite_rows`: correspondem ao quarto e ao quinto parâmetros do construtor de `Sprite`, ou seja, determinam o número de linhas e o de colunas da *spritesheet* (para mais detalhes, ver tópico 2.3.1).
  - `mass`: massa do objeto, deve ser um número positivo (quando não especificada, o valor é 1). Quanto maior a massa, menor a aceleração do objeto para uma mesma intensidade de força aplicada (conforme a segunda lei de Newton). Traduzindo para o jogo, o que ocorre é que, a cada chamada à função move, à velocidade do objeto é somado o quociente entre a intensidade da força e sua massa.
- O código para o método `draw` ficou um pouco extenso porque se decidiu por desenhar retângulos para tornar visíveis os blocos, e triângulos para as rampas, já que estes objetos não possuem imagem associada. Uma alternativa a isto seria criar classes que

herdassem de `Block` e `Ramp` e pudessem receber uma imagem, disponibilizando também um método `draw`.

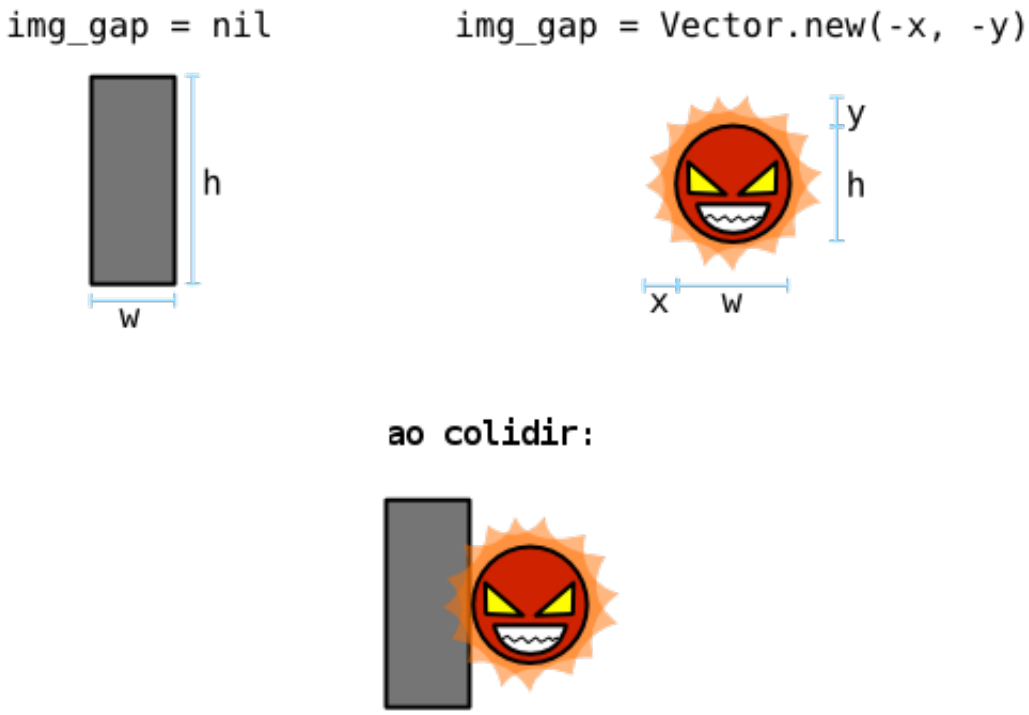

Figura 6: O uso de `img_gap`: nem sempre os limites físicos de um objeto coincidem com os limites da imagem que o representa.

Embora tenhamos discorrido longamente sobre as funcionalidades de movimentação da biblioteca MiniGL, nem tudo foi coberto ainda. Um objeto que inclui o módulo `Movement` ganha também as seguintes possibilidades:

- Movimentação livre. Através do método `move_free`, pode-se colocar um objeto para se mover em direção a um ponto alvo, com uma velocidade fixa, sem checagens de colisão.
- Movimentação "de elevador". Através do método `move_carrying`, pode-se fazer um objeto se mover de modo semelhante ao descrito para `move_free`, mas recebendo uma lista de outros objetos que ele deve "carregar" no caso de colidir com estes "por baixo". Trata-se do comportamento de elevadores, elementos relativamente comuns em jogos de plataforma, e uma contribuição significativa aos desenvolvedores por parte da biblioteca (a implementação de elevadores é consideravelmente complicada).
- Movimentação em ciclos. Pode-se fazer um objeto iniciar um movimento cíclico, passando por uma lista de pontos dados como parâmetro ao método `cycle`. Mais ainda, pode-se combinar esta funcionalidade com o comportamento de elevador, ao fornecer também como parâmetro a lista de objetos que devem ser carregados.

2.3.7. Gerenciamento de *viewport*

Este tópico trata também de um componente dispensável para certos gêneros de jogo, mas muito útil para outros: o *viewport*, que podemos traduzir aproximadamente por "câmera" ou "campo de visão". Em poucas palavras, gerenciamento de *viewport* corresponde ao controle do campo de visão do personagem, considerando-se um cenário amplo, que não pode ser retratado inteiramente na tela do jogo – ou seja, não é possível visualizar todas as partes do cenário simultaneamente, logo uma espécie de "movimentação de câmera" é necessária.

Esta funcionalidade é encapsulada pela classe `Map` da *gem* MiniGL. Esta classe combina facilidades para controle do *viewport* com um outro tipo de elemento comum em jogos: os chamados *tiles*. Estes, por sua vez, correspondem a pequenos retângulos que, combinados em forma de grade, compõem o "mapa" ou o cenário do jogo. Assim, ao definir uma instância de `Map`, podem-se parametrizar a posição da "câmera" e o tamanho e quantidade de *tiles*. Entretanto, o uso de tiles é opcional (pode-se usar a classe `Map` apenas para gerenciar o campo de visão).

Para melhor compreender de que modo podemos combinar estas funcionalidades, obtendo um código elegante que exibe os elementos em suas posições de acordo com a movimentação da câmera, consideremos o seguinte caso: o personagem principal se desloca pelo cenário conforme comandos do jogador, e a câmera deve se mover de modo a sempre mostrar o personagem no centro da tela; o cenário é composto de tiles. A figura 7 ilustra uma possível representação visual de um caso deste tipo.

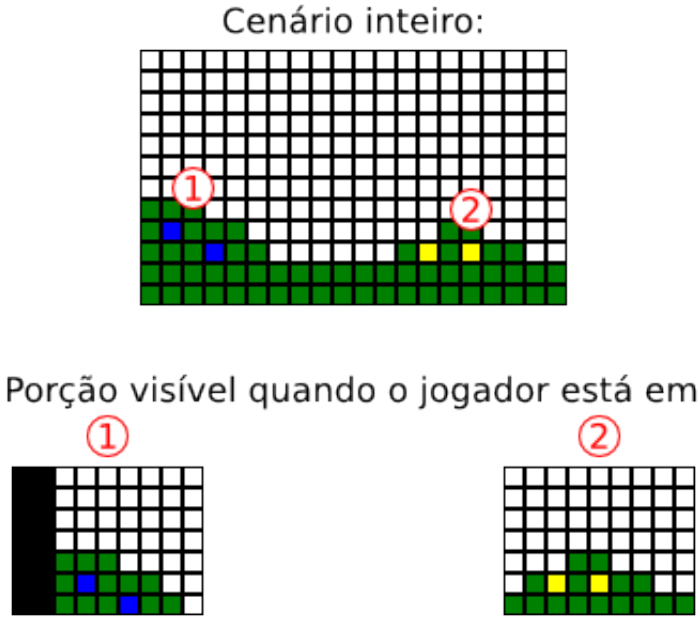

Figura 7: Exemplo do efeito de movimentação de viewport*. As partes do cenário exibidas na tela são sempre os arredores do personagem controlado pelo jogador.*

Agora observemos um trecho de código que, utilizando MiniGL, reproduz aproximadamente o descrito acima:

```
def initialize
  super 90, 70, false
  Game.initialize self
  # matriz representando o cenário da figura 7
  @tiles = [[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
           [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
           [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
           [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
           [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
           [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
           [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
           [1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
           [1,2,1,1,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0],
           [1,1,1,2,1,1,0,0,0,0,0,0,1,3,1,3,1,1,0,0],
           [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
           [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]]
  #          branco      verde       azul        amarelo
  @colors = [0xffffffff, 0xff008000, 0xff0000ff, 0xffffff00]
  @map = Map.new 10, 10, 20, 12, 90, 70
  @player = Sprite.new 0, 0, :img
end

def update
  KB.update
  @player.y -= 1 if KB.key_down? Gosu::KbUp
  @player.x += 1 if KB.key_down? Gosu::KbRight
  @player.y += 1 if KB.key_down? Gosu::KbDown
  @player.x -= 1 if KB.key_down? Gosu::KbLeft
  @map.set_camera @player.x - 40, @player.y - 30
end

def draw
  @map.foreach do |i, j, x, y|
    draw_quad x + 1, y + 1, @colors[@tiles[j][i]],
              x + 9, y + 1, @colors[@tiles[j][i]],
              x + 9, y + 9, @colors[@tiles[j][i]],
              x + 1, y + 9, @colors[@tiles[j][i]], 0
  end
  @player.draw @map
end
```

Notem-se os seguintes detalhes:

- Os parâmetros passados ao construtor de `Map` são, nesta ordem:
  - `t_w` → largura de cada *tile*;
  - `t_h` → altura de cada *tile*;
  - `t_x_count` → contagem de *tiles* horizontalmente;
  - `t_y_count` → contagem vertical de *tiles*;
  - `scr_w` → largura da tela (opcional, assume valor padrão 800)
  - `scr_h` → altura da tela (opcional, assume valor padrão 600)
- Ao executar o código acima, pode-se observar que, ao aproximar o jogador da posição 1 indicada na figura 7, não aparecerá uma borda preta como na imagem; em vez disso,

o jogador será mostrado não ao centro, mas um pouco à esquerda deste na tela. Isto ocorre porque o comportamento padrão da câmera é respeitar os limites do mapa (em geral, mostrar um espaço vazio não é visualmente muito agradável). Este comportamento pode ser alterado passando-se `false` como oitavo parâmetro de `Map.new` (o sétimo parâmetro, também opcional, será explicado mais à frente).

- O método `foreach` do mapa é uma função extremamente útil, que retorna a posição na matriz de *tiles* (parâmetros `i` e `j`) e as coordenadas na tela (parâmetros `x` e `y`) de cada *tile* atualmente visível. Deste modo, evita-se desenhar ou executar lógica relativa a elementos que não estão visíveis na tela. Durante testes da biblioteca, foi constatado que a iteração por listas com alguns milhares de elementos já produzia impacto no desempenho. Portanto, atualizar/desenhar apenas parte dos objetos, num cenário extenso, é fundamental para se manter o jogo fluindo a sessenta quadros por segundo, e o método `foreach` contribui para facilitar este tratamento.

As regras descritas acima também se aplicam, com a simples alteração de um parâmetro no construtor – o sétimo parâmetro, que vale `false` por padrão, mudado para `true` – para a criação de um mapa isométrico, ou seja, um mapa em que a grade de *tiles* é apresentada de forma inclinada. Isto pode ser melhor observado na figura 8. Mapas isométricos são muito úteis e frequentemente usados, por exemplo, em jogos do gênero RTS (*Real Time Strategy*).

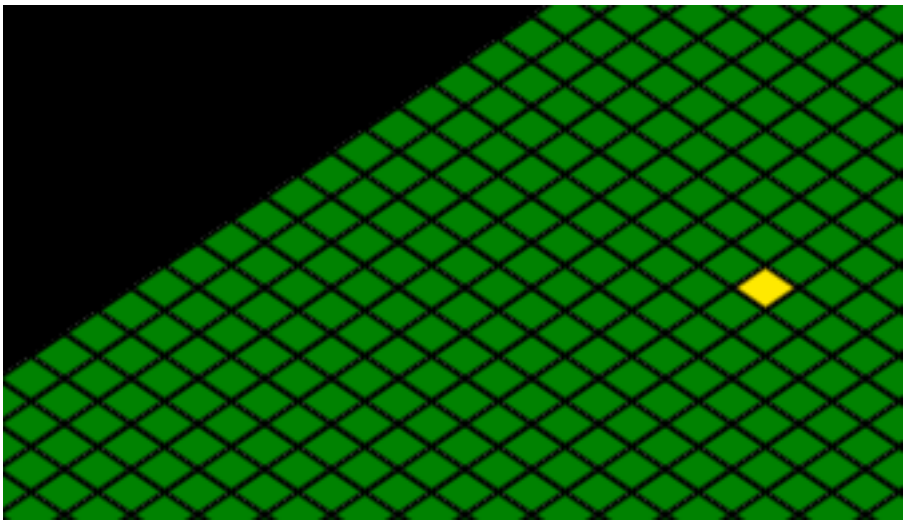

Figura 8: Pedaço de mapa isométrico renderizado com auxílio de MiniGL. O tile amarelo indica a posição sobre o qual o cursor do mouse estava pousado.

Os cálculos envolvidos na transformação de coordenadas de uma grade ortogonal (com eixos paralelos aos lados da tela) para uma grade isométrica correspondem à aplicação de alguns conceitos básicos de álgebra linear, como rotação, translação e multiplicação (por escalares) de vetores.

Cabe observar também que a classe `Map` fornece ainda alguns métodos auxiliares, que permitem a obtenção de coordenadas de *tiles* específicos, ou o processo contrário – obter o *tile* da matriz que corresponde, atualmente, a uma dada posição da tela, método usado para destacar

o *tile* amarelo da figura acima –, assim como a obtenção de certas métricas a respeito do mapa como um todo (tamanho absoluto, posição central, etc.).

2.3.8. Gerenciamento de memória/recursos

Para concluir a análise minuciosa do arcabouço MiniGL, voltemos a atenção ao módulo de gerenciamento de recursos por ele fornecido. Este representa um dos pilares de suporte das funcionalidades da biblioteca: vários outros módulos são construídos utilizando-se já dos métodos de obtenção centralizada de recursos providos pela classe `Res`, que constitui o núcleo do módulo.

Vimos, em seções anteriores, diversas inicializações de imagens e fontes, com `Res.img` e `Res.font`, além de inicializações de objetos como `Sprite` e `GameObject`, que recebiam um símbolo (elemento característico da linguagem Ruby, que corresponde a uma *string* única, ou seja, uma instância de *string* que pode ser reutilizada por diversos módulos de um programa, sem a necessidade de instanciar um novo objeto *string* toda vez), como `:img`, `:obj`, `:font1`, etc. Esta sintaxe de símbolos para carregamento de recursos é característica da classe `Res`.

Para melhor esclarecer, enumeremos os vários métodos oferecidos por esta classe, comparando-os com a sintaxe de carregamento utilizando apenas os métodos oferecidos diretamente pela *gem* Gosu:

- Imagem:
```
Res.img :imagem                              # MiniGL
Image.new win, “data/img/imagem.png”, false # Gosu
```
- Vetor de imagens:
```
Res.imgs :sp_sheet, 3, 2                                        # MiniGL
Image.load_tiles win, “data/img/sp_sheet.png”, -3, -2, false # Gosu
```
- Amostra de som:
```
Res.sound :sound1                        # MiniGL
Sample.new win, “data/sound/sound1.wav” # Gosu
```
- Música:
```
Res.song :song1                      # MiniGL
Song.new win, “data/song/song1.ogg” # Gosu
```
- Fonte:
```
Res.font :font1, 20                      # MiniGL
Font.new win, “data/font/font1.ttf”, 20 # Gosu
```

Nota-se que as diferenças entre o carregamento com MiniGL e com Gosu são praticamente sempre as mesmas: no caso da Gosu, é necessário fornecer uma referência para a janela principal do jogo – representada acima pela variável `win` – na inicialização de todos os recursos. Assim, este é um ponto que obviamente poderia ser melhorado se a referência à janela principal – que não muda durante a execução – fosse mantida em algum tipo de variável global. A biblioteca MiniGL guarda esta referência em `Game.window`, e com isso é possível carregar recursos sem especificar a janela explicitamente.

Além disso, a outra diferença evidente diz respeito à especificação de caminhos de arquivos. Com Gosu pura, é necessário especificar o caminho absoluto ou relativo ao diretório atual, e

como é muito comum que os recursos sejam organizados em subpastas, quase toda vez esses nomes de subpastas têm de ser reescritos no código. Utilizando-se de convenções, conforme dita o padrão moderno *Convention over Configuration* [10], MiniGL permite que os principais nomes de pastas sejam omitidos no carregamento dos arquivos.

As convenções mencionadas acima correspondem simplesmente à organização de cada tipo distinto de recurso (imagem, som, música ou fonte) num diretório com nome intuitivo, diretamente ligado a esse tipo. Para as imagens, este diretório deve ser "data/img"; para efeitos sonoros, "data/sound"; para músicas, "data/song"; e para fontes, "data/font".

Porém, a contribuição da classe `Res` para o desenvolvimento do jogo não se limita a tornar a sintaxe mais limpa e resumida. Ela tem também o papel de repositório centralizado, como foi comentado anteriormente, o que significa que, quando um recurso é carregado, uma referência para ele é mantida; assim, ao se pedir para carregar novamente este recurso, não é necessário executar todo o processamento novamente, apenas retornar a referência; por outro lado, se for do desejo do desenvolvedor desfazer-se das referências num dado momento, para liberar memória, pode-se utilizar o método `Res.clear`.

Neste contexto, há uma diferenciação entre recursos considerados "globais" e recursos "locais": todos os métodos de carregamento admitem um parâmetro adicional, que por padrão vale `false`, mas que quando definido como `true`, indicará para o repositório `Res` que este recurso é global, ou seja, não deve ser limpo da memória enquanto o jogo estiver em execução. Isto é útil para certos elementos de interface que aparecem com muita frequência no jogo (fontes, botões, imagens de fundo do menu, etc.).

Para concluir enfatizando os benefícios trazidos pelo módulo de gerenciamento de recursos da MiniGL, observemos a tabela de comparação de uso de CPU e memória, entre o método nativo da *gem* Gosu e o método da *gem* MiniGL, ao carregar múltiplas vezes uma imagem relativamente pesada (113 kB):

Número de imagens carregadas	**Gosu**			**MiniGL**		
	CPU (média/máximo)		**Memória**	**CPU (méd/máx)**		**Memória**
100	45,0%	99,2%	11,5 MiB	0,6%	17,8%	9,0 MiB
200	47,6%	98,9%	13,4 MiB	0,8%	14,0%	9,0 MiB
400	48,5%	99,1%	13,7 MiB	1,0%	11,9%	9,1 MiB

Tabela 1: Comparação de consumo de recursos computacionais ao carregar imagens entre o método provido pela biblioteca Gosu e aquele provido pela biblioteca MiniGL.

Os resultados acima foram obtidos a partir de dez experimentos para cada quantidade de imagens (totalizando trinta experimentos). Estes experimentos foram realizados numa máquina com as seguintes configurações:

- Processador Intel i3 2310M
- 4 GB de memória RAM
- Placa de vídeo Intel HD Graphics 3000 (integrada)
- Sistema operacional Linux Mint 16 ("Petra")

2.4. Conclusões

Vimos, no decorrer das oito seções anteriores, o quão complexo pode ser o processo de desenvolvimento de jogos. Para desenvolvedores independentes, alta produtividade é indispensável, e isto só pode ser alcançado com a utilização de arcabouços que forneçam funcionalidades abrangentes – ou seja, que possam ser aplicadas para tipos diversos de jogos – e com interfaces fáceis e intuitivas – permitindo código limpo, sucinto e autoexplicativo, até certo ponto.

Tendo isso em vista, há diversas bibliotecas para jogos já disponibilizadas para também diversas linguagens de programação. No caso de Ruby, no entanto, não foi encontrada, durante o período de pesquisas que antecedeu a produção deste projeto, uma biblioteca que atendesse às muitas necessidades de aumento de produtividade que foram sendo identificadas ao longo do desenvolvimento do jogo educativo e de outros jogos, anteriormente a este projeto.

Assim, foi estabelecido como parte do objetivo do projeto a criação de um arcabouço que apresentasse as características desejáveis já citadas anteriormente. O arcabouço foi implementado e conseguiu atender grande parte das necessidades, conforme foi possível observar através do detalhamento dado nas seções anteriores.

Embora haja, certamente, pontos de melhora que possam ser identificados, o arcabouço foi efetivamente aplicado na produção do jogo educativo, e facilitou muito o seu processo de desenvolvimento. Além disso, a biblioteca parece ter tido boa aceitação por parte da comunidade de desenvolvedores, já que alcançou quase 3000 downloads no repositório de bibliotecas Ruby, com apenas 5 meses de existência, e sem nenhum tipo de divulgação.

3. “Aventura do Saber”: um jogo educacional em Ruby

Agora que já passamos por um detalhamento do arcabouço MiniGL, torna-se viável uma observação mais detalhada da implementação do jogo “Aventura do Saber”, cujo código-fonte é ricamente permeado pela utilização de funções da MiniGL. Além disso, esta implementação também se beneficiou de diversas características interessantes da linguagem Ruby, as quais serão destacadas e comentadas conforme aparecem.

Antes de iniciarmos a exploração do código, no entanto, verifiquemos quais foram as ideias e decisões de mais “alto nível” que levaram o jogo a ser implementado como foi, e a apresentar as características que apresenta.

3.1. Decisões iniciais

Primeiramente, por que um jogo educacional? Em verdade, a ideia inicial do projeto – que remonta ao primeiro ou segundo mês deste ano – era apenas a implementação de um jogo comum (não educacional), utilizando um arcabouço, que também faria parte do projeto – esta ideia foi mantida –, com o principal objetivo de demonstrar as facilidades oferecidas, em primeira instância, pela linguagem Ruby, e, em segunda instância, pelo arcabouço.

No entanto, dentre os diálogos que ocorreram nas fases iniciais de projeto, com diversas pessoas conhecidas, algumas delas já tendo atuado profissionalmente na área de educação, foi levantada a questão de como o sistema educacional no país estava necessitado de ideias, métodos e ferramentas novas para trazer maior interatividade, e consequentemente maior interesse por parte das crianças, para as aulas. Rapidamente, isto culminou com a sugestão de que o projeto aqui apresentado fosse voltado para este propósito, produzindo-se um jogo educativo.

De fato, após a sugestão, mais pesquisas foram feitas e ajudaram a constatar que este mercado específico – de jogos educacionais – é ainda incipiente no Brasil. Mais incomuns ainda são as iniciativas de software livre neste sentido [11]. Portanto, identificou-se que um projeto de jogo educativo poderia oferecer uma contribuição sólida para a sociedade.

Tomada a decisão do propósito do jogo, ainda havia a necessidade de definir diversos detalhes, como, por exemplo, o gênero. Foi escolhido o gênero aventura por ser um com o qual o autor deste projeto tem afinidade, além de já ter uma certa experiência vinda de projetos anteriores deste tipo. Além do gênero, era necessário também definir em que tipo e nível de conhecimento o jogo focaria. Decidiu-se por focar nos dois alicerces básicos dos conhecimentos gerais – o idioma e a matemática –, além de um outro alicerce importante, ligado ao raciocínio: a Lógica. O nível seria básico, de modo a direcionar o jogo para alunos de primeiros anos de ensino fundamental, pois estes fazem parte de um período crucial na formação do intelecto das crianças [1].

3.2. Explorando o jogo

Enfim, temos todas as informações necessárias para começar a explorar o jogo em seus detalhes. Num primeiro momento, avaliemos o jogo sob a ótica de um usuário comum, identificando seus principais elementos e características, e refletindo sobre como eles contribuem para alcançar o objetivo do projeto, constituindo um jogo educativo, interativo, atraente, e

que permite a avaliação das facilidades ou dificuldades que o aluno possa vir a enfrentar no decorrer da "aventura".

3.2.1. A interface gráfica

É indiscutível a importância de um bom planejamento de interface gráfica para a criação de um jogo. Segundo Schell [12], a interface, "quando malfeita, torna-se um muro entre o jogador e o mundo do jogo; quando bem-feita, amplifica o poder e controle que o jogador tem sobre o mundo do jogo"[1].

Afora esta constatação, que se aplica a qualquer variedade de jogo, tem de se levar em conta o fato de que o jogo será direcionado a crianças. A simplicidade da interface passa a ser ainda mais importante. Além disso, deve ser atraente, no sentido de incitar o interesse da criança pelo jogo. Com essas diretivas em mente, produziu-se uma interface cujos traços são exemplificados pelas imagens a seguir.

Figura 9: Tela inicial do jogo "Aventura do Saber".

1 Tradução livre de "Done poorly, they become a wall between the player and the game world. Done well, they amplify the power and control a player has in the game world."

Figura 10: Tela de identificação do jogador. O jogador pode escrever seu nome para iniciar um novo jogo ou selecioná-lo dentre os jogos salvos, listados abaixo.

Procurou-se sempre incluir ilustrações mesmo nas telas dos menus, para tornar a experiência mais atrativa e chamativa para o usuário, sendo a tela de identificação, mostrada na figura 9, uma das poucas exceções. Mas, mais importante do que isso, procurou-se utilizar uma interface bastante simples, com o mínimo de informação na tela, pois este pode ser um fator determinante com relação ao conforto que o usuário terá ao interagir com o software e, consequentemente, com relação ao estímulo da continuidade desta interação [13].

As telas das figuras 9 e 10 exemplificam também o uso de componentes de interface de usuário fornecidos pela MiniGL (botões e caixa de texto), e de funcionalidades fornecidas pelo módulo de texto (classe `TextHelper`), como por exemplo o título do jogo, centralizado no topo da tela.

A figura 11, por outro lado, mostra uma situação diferente. Há bastantes informações na tela, e estas informações certamente pareceriam confusas aos olhos de um estudante das primeiras séries do ensino fundamental. Porém, a ideia aqui é outra: esta tela serve para que um adulto – mais especificamente, um educador – possa acompanhar o progresso da(s) criança(s) no jogo. Mais detalhes sobre esse acompanhamento são discutidos na seção seguinte.

Figura 11: Tela de pontuações e resultados. Esta tela é voltada para os olhos de um adulto, e por isso não segue o princípio da simplicidade aplicado a todas as demais telas.

Os exemplos exibidos até aqui ilustram bem os contornos gerais das telas "periféricas" do jogo, isto é, aquelas que não compreendem a ação principal do jogo em si. Estas outras, a que podemos nos referir por telas do "núcleo" do jogo, serão exemplificadas pelas próximas imagens.

Na figura 12, tem-se um exemplo típico de tela do núcleo do jogo, num momento em que o jogador não está interagindo com nenhum elemento do cenário. Mais uma vez, tem-se uma pequena quantidade de informações na tela: o nome e a pontuação do jogador, os itens que ele possui atualmente (se possuir algum) e dicas sobre o modo de jogar. Estas são importantes de início, já que não há uma tela de instruções, mas conforme o jogador se acostuma com os controles, podem-se tornar desnecessárias. Por isso existe a opção de ocultá-las no menu de opções, obtendo-se uma interface ainda mais limpa.

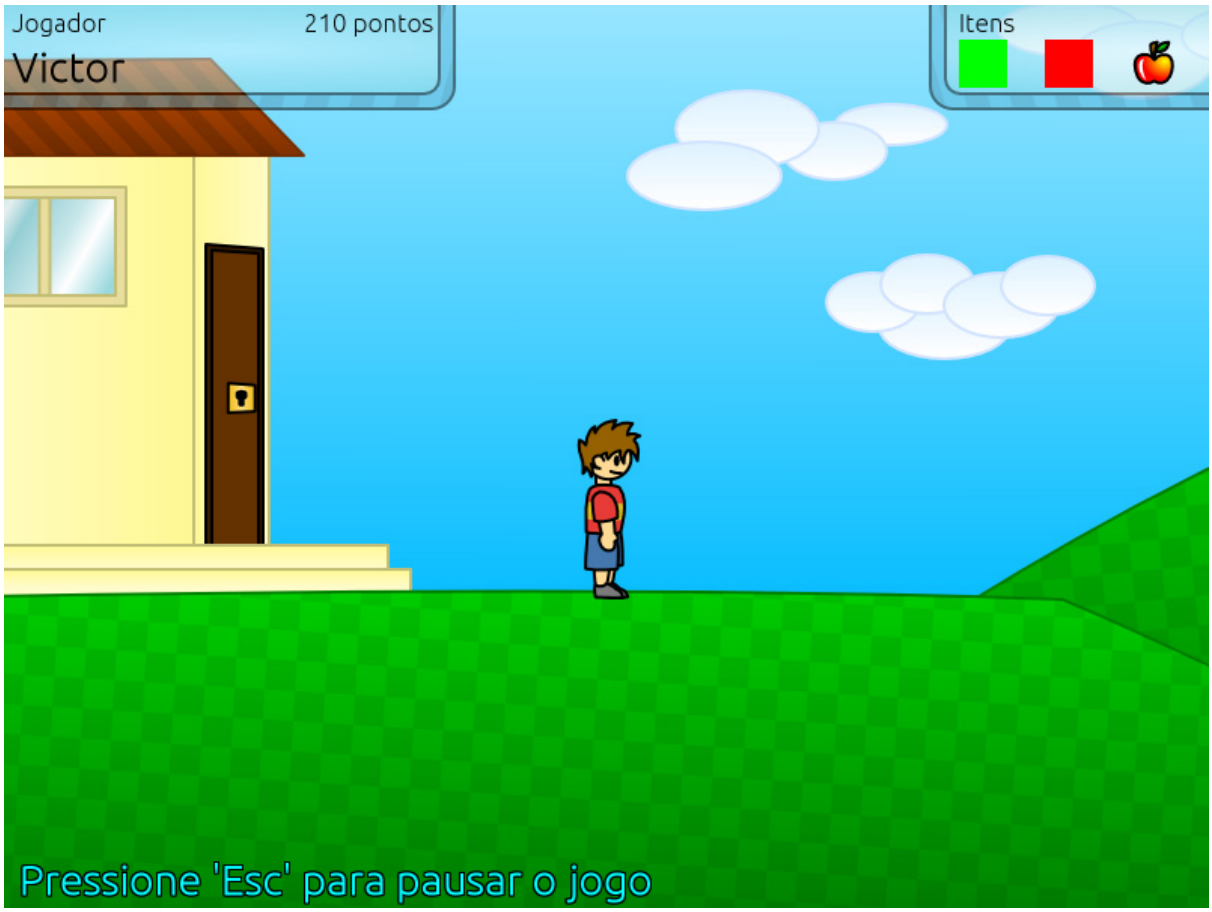

Figura 12: Tela típica do núcleo do jogo.

A ação do jogo se resume basicamente em três tipos de atividade: andar com o personagem pelo cenário, explorando o mundo para encontrar itens a coletar, objetos ou outros personagens, interagir com objetos – o que corresponde a executar uma sequência de ações, às vezes utilizando os itens coletados –, ou interagir com personagens, respondendo a perguntas e fornecendo-lhes itens.

Quando o jogador se encontra numa posição em que pode interagir com um objeto ou personagem, isto é assinalado graficamente por meio de um ponto de exclamação em cima do objeto, ou um balão com reticências, indicando que se pode conversar com aquele personagem, conforme ilustra a figura 13. A figura 14 exemplifica um momento de interação com um personagem, em que uma pergunta é feita e o jogador deve selecionar a resposta correta para prosseguir; a figura 15, por sua vez, exemplifica a interação com um objeto, onde é solicitada a utilização de um item.

Figura 13: Elementos gráficos indicando a possibilidade de interação com um objeto (esquerda) e com um personagem (direita).

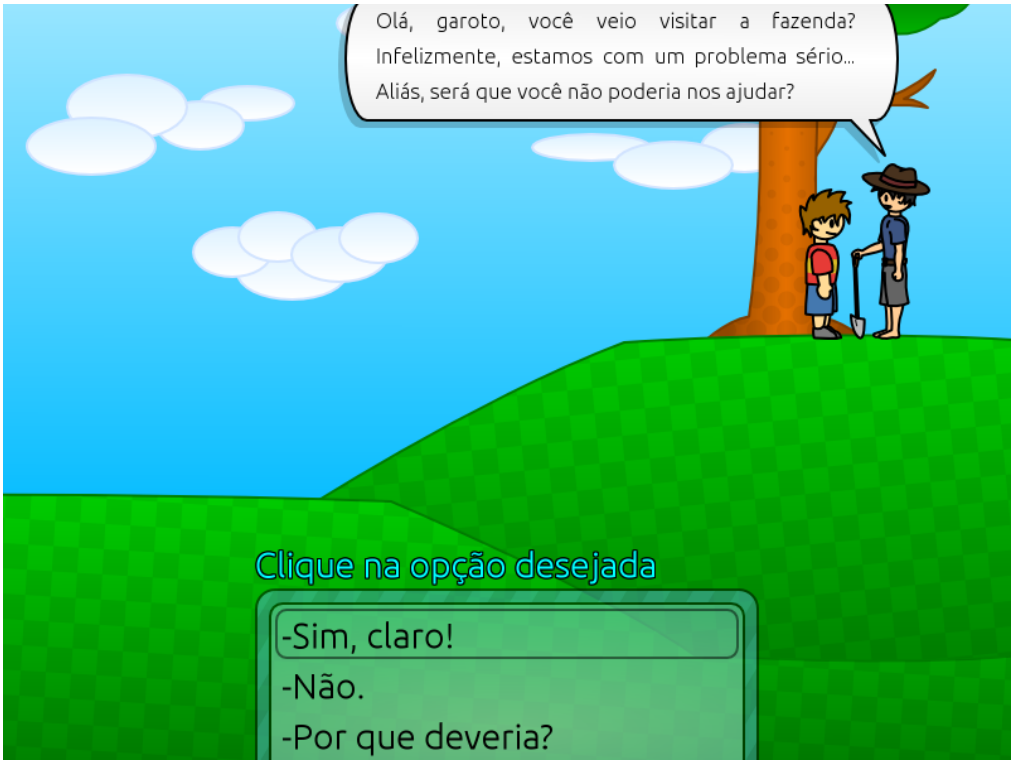

Figura 14: Diálogo entre o jogador e um personagem do cenário. O jogador deve escolher a resposta mais apropriada à pergunta para prosseguir.

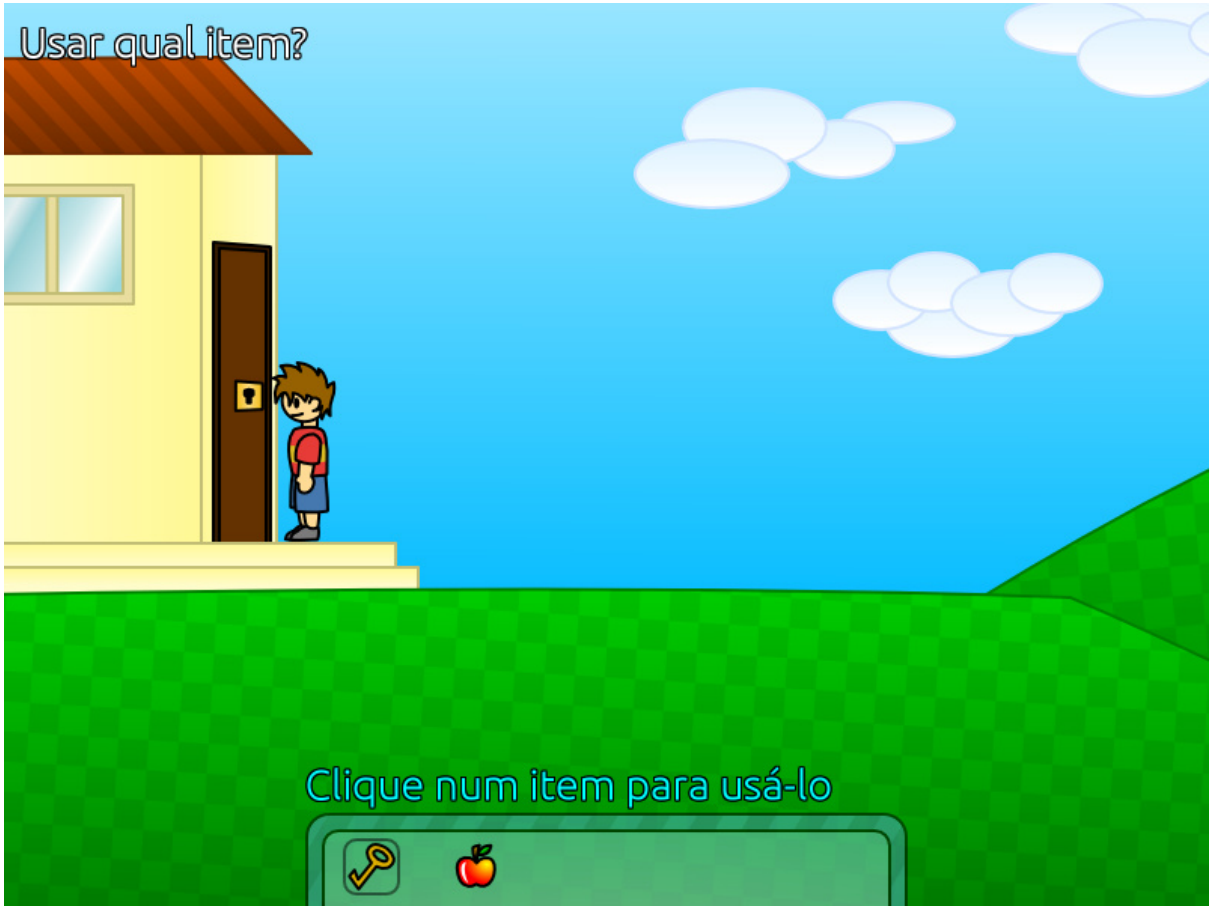

Figura 15: Interação com um objeto. O jogador deve selecionar o item correto para prosseguir.

É válido notar que, quando o jogador está interagindo, são ocultadas as informações que normalmente se encontram fixas no topo da tela (nome do jogador, pontuação e itens). Com isso, mantém-se pouca informação na tela, sempre orientado pela diretriz da simplicidade da interface.

3.2.2. O sistema de pontuação

O sistema de pontuação do jogo é relativamente simples, podendo ser descrito pelas regras a seguir:

- Para cada item coletado no cenário, ganha-se uma quantidade de pontos, que depende do item, de modo a valorizar mais os itens mais difíceis de coletar.
- Para cada resposta correta ou item correto dado a um personagem, ganha-se uma quantidade de pontos também parametrizada de acordo com a dificuldade da pergunta ou do pedido.
- Para cada resposta errada dada a um personagem, perde-se uma pequena quantidade de pontos – aproximadamente 10% do valor da resposta correta. Isto se faz necessário para possibilitar a diferenciação entre um jogador que acertou a pergunta na primeira

tentativa e outro que errou uma ou mais vezes antes de acertar, já que o jogador pode tentar cada uma das alternativas livremente.

- Para cada ação correta ou item correto utilizado junto a um objeto, ganha-se também uma quantidade parametrizada de pontos.

Além da pontuação propriamente dita, que é simplesmente um número que vai sendo alterado conforme o jogo avança, são guardadas algumas outras estatísticas quanto ao progresso do jogador, conforme foi possível vislumbrar na tela detalhada de pontuações, no tópico anterior. Deste modo, torna-se mais simples a tarefa do educador de avaliar as facilidades e dificuldades de cada estudante, e também verificar se a aplicação do jogo como ferramenta auxiliar tem surtido o efeito desejado.

A primeira estatística, "cenas visitadas", fornece uma noção de quanto do cenário já foi explorado pelo jogador. Em geral, é necessário completar certos desafios para ganhar acesso a cada próxima cena, e portanto este número é um bom indicador do nível de progresso do usuário.

A segunda estatística, "respostas corretas", fornece o mesmo tipo de informação que o descrito acima, porém de forma um pouco mais precisa e específica. Dois alunos podem ter explorado um mesmo número de telas, mas um deles pode já ter resolvido alguns desafios a mais que o outro – por exemplo, desafios encontrados na última cena que eles visitaram –, e esta diferença sutil pode ser identificada pelo número de respostas corretas.

Por fim, a terceira estatística, que diz respeito a "respostas erradas", permite uma inferência rápida de qual assunto oferece mais dificuldade para cada estudante. Por outro lado, um número alto de respostas erradas generalizado pode indicar que os alunos não se adaptaram bem ao jogo, caso em que, pelo menos, o jogo permite fazer esse diagnóstico. Veremos mais adiante, no entanto, que é possível, sem a necessidade de programar, adaptar o jogo às crianças de acordo com seu nível de aprendizado.

O gráfico mostrado na parte inferior da tela (ver figura 11) é uma maneira visualmente mais interessante e rápida de detectar as afinidades por assunto de cada estudante: uma reta constante por um número grande de jogadas indica que o aluno enfrentou ou está enfrentando dificuldade no assunto correspondente; quando o aluno está absorvendo bem um determinado tipo de conhecimento, espera-se que o gráfico para aquela área do conhecimento seja constantemente ascendente.

É importante notar que, conforme as informações dadas até aqui a respeito do jogo já podem ter permitido deduzir, o seu principal papel é aplicar e testar conhecimentos já aprendidos em sala de aula ou por meio de outros métodos quaisquer. O jogo não inclui características didáticas, mas sim de avaliação – embora um jogo didático pudesse ser igualmente interessante, a necessidade de limitar o escopo do projeto, que já era amplo ao abarcar tanto um jogo quanto um arcabouço, levou a esta decisão com relação ao desenvolvimento.

3.2.3. O sistema de "missões" e "cenas"

Para finalizar a descrição de "alto nível" do jogo, passemos brevemente por uma explanação acerca do sistema de missões, que é o modo como o jogo se subdivide. Para cada área do conhecimento que o jogo aborda, há uma missão. Há, ainda, uma missão que combina um pou-

co das três áreas – Matemática, Língua Portuguesa e Lógica. A figura 16 mostra a tela de escolha de missões.

Figura 16: A tela de escolha de missões.

Basicamente, uma missão representa um conjunto de cenas – ou telas –, que contêm elementos com os quais o personagem pode interagir. Todas as cenas de uma missão estão interligadas entre si, no sentido de que é necessário passar por todas elas, possivelmente mais de uma vez pela mesma, para se completar a missão. Os itens coletados numa cena são mantidos ao longo de todas elas, e as interações já feitas com objetos ou personagens também ficam gravadas, de modo que, ao retornar a uma cena já visitada, o jogador encontrará tudo do jeito que deixou.

O jogador pode iniciar e manter salvo o progresso para todas as missões do jogo simultaneamente, tornando a experiência mais flexível e menos maçante. Assim, o aluno não precisa completar toda a missão de Matemática para então poder praticar Língua Portuguesa. Em vez disso, pode alternar a prática desta com a daquela, e ainda com um pouco de Lógica. Toda vez que o jogo precisar ser interrompido, não haverá perdas de progresso, desde que ele seja encerrado corretamente – ver figura 17.

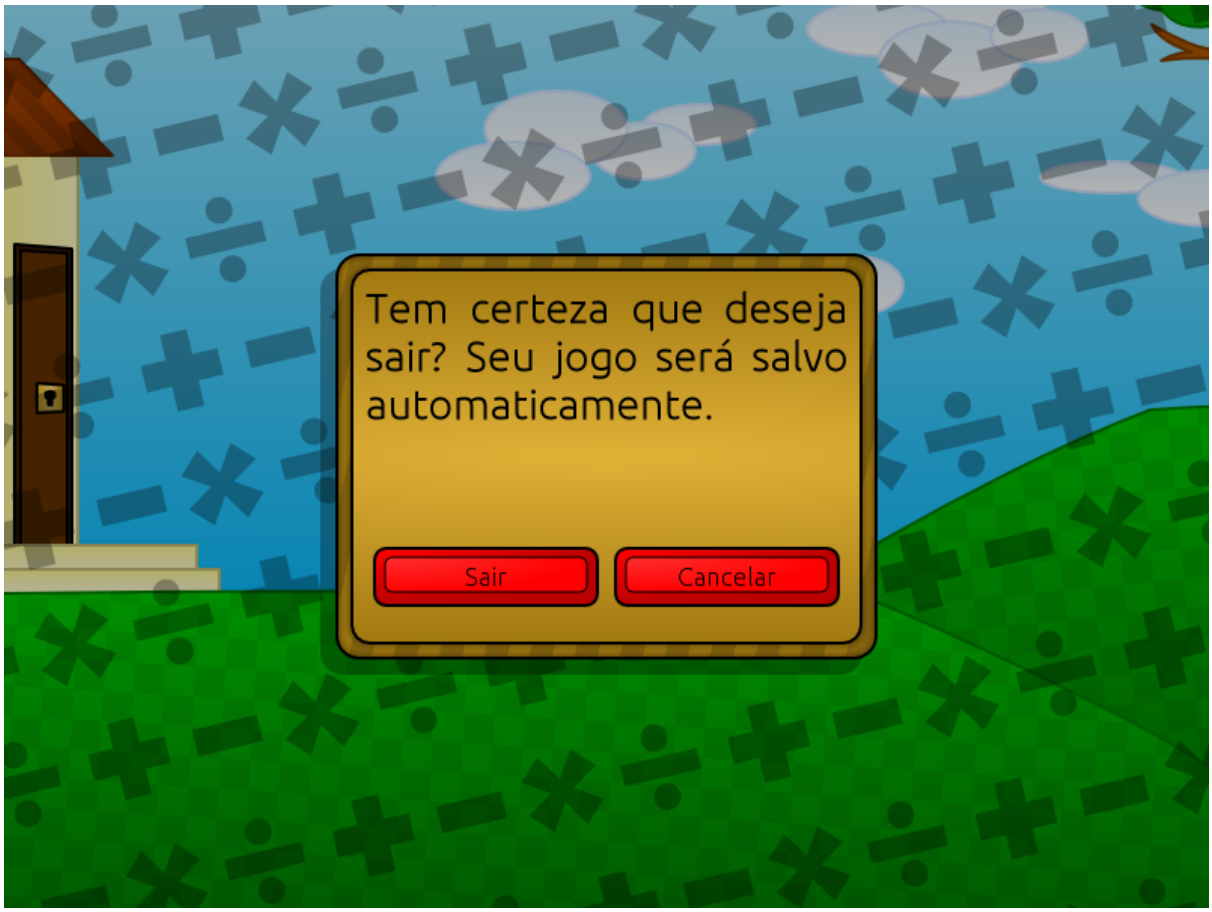

Figura 17: Salvamento automático ao sair do jogo. O jogador não precisa se lembrar de salvar o progresso periodicamente.

3.3. Arquitetura do software

Iniciemos, enfim, uma análise mais técnica do jogo, focando agora nos detalhes da implementação, nas sutilezas do código e nas particularidades da linguagem Ruby que permitiram o desenvolvimento do jogo com alta produtividade e a criação de um código com alta expressividade. Como é natural ao se utilizar esta linguagem, a arquitetura do jogo é fortemente orientada a objetos, e a seguir veremos as principais classes que definem esta arquitetura.

3.3.1. Classe “G”

Esta classe é responsável por expor variáveis e métodos que precisam estar acessíveis a todos os módulos do jogo, ou seja, variáveis e métodos globais. Em vez de utilizar variáveis globais padrão (variáveis que, em Ruby, têm o nome iniciado por “$”), optou-se por encapsular todas elas nesta classe, pois a utilização de variáveis globais é, em geral, considerada uma má prática.

A maioria destas variáveis precisa apenas ser lida pelos outros módulos, raramente sendo modificadas diretamente por estes. Deste modo, tem-se um cenário com diversas variáveis com modificador de acesso *private* – fazendo-se um paralelo com linguagens mais clássicas como

Java –, mas que precisam de um método *getter* correspondente. Com o uso de Ruby, não é necessário declarar todas as variáveis e todos os métodos *getter*. Em vez disso, utilizou-se o poder de linguagem dinâmica de Ruby para criar os métodos *getters* automaticamente para todas as variáveis inicializadas até o momento. Veja abaixo:

```
class G
  def self.initialize hints, sounds, music
    @@win = Game.window
    @@full_screen = @@win.fullscreen?
    @@hints = hints
    @@sounds = sounds
    @@music = music

    @@font = Res.font :UbuntuLight, 20
    @@med_font = Res.font :UbuntuLight, 32
    @@big_font = Res.font :UbuntuLight, 54
    ...
    @@effects = f.readlines
    ...
    @@items = {}
    ...
    @@s_items = {}
    ...
    @@switches = nil
    @@state = :menu

    @@menu = nil
    @@player = nil
    @@scene = nil
    @@scenes = nil
    @@c_answers = nil
    @@w_answers = nil

    class_variables.each do |v|
      define_singleton_method(v.to_s[2..-1]) { class_variable_get v }
    end
    ...
  end
  ...
end
```

Pode parecer pouco relevante, mas, levando-se em conta que se trata de dezenove variáveis, a quantidade de código que deixou de ser escrito graças a essa abordagem dinâmica é considerável (seriam dezenove declarações e dezenove métodos *getter*, portanto várias dezenas de linhas de código). Isto contribui não apenas para a produtividade, mas também, e principalmente, para a limpeza/sucintez do código.

Em termos de arquitetura, esta classe `G` é referenciada por quase todas as outras classes importantes do jogo, exceto aquelas que representam componentes bastante isolados e que não dependem do acesso às variáveis e métodos globais, como é o caso de `Chart` e `Intro`.

Algumas das informações importantes que a classe G disponibiliza aos diversos módulos do jogo são:

- Uma referência à janela do jogo, onde todos os elementos são desenhados.
- Variáveis *booleanas* correspondentes à configuração atual das opções do jogo, que podem estar habilitadas (`true`) ou desabilitadas (`false`). As opções são: tela cheia, sons, música e dicas.
- Três fontes pré-carregadas, que são usadas para escrever todos os textos do jogo.
- Uma referência ao jogador atual – esta referência é nula enquanto o usuário ainda não entrou nas telas do "núcleo" do jogo.
- Uma referência à cena atual em que o jogador se encontra – esta referência é nula enquanto o usuário ainda não entrou nas telas do "núcleo" do jogo.
- O vetor de *switches* da missão atual (mais detalhes sobre os *switches* em seções posteriores).
- O estado atual do jogo, que permite diferenciar se o jogador está jogando uma missão, se está com o jogo pausado, se está nas telas de menu, etc.

A maioria dos métodos expostos pela classe `G` servem para encapsular modificações nestas variáveis globais. Toda a lógica que manipula os elementos globais do jogo, portanto, fica sob responsabilidade desta classe.

3.3.2. Classe "Player"

Esta classe representa o jogador, ou, mais especificamente, o personagem que o jogador controla no cenário, quando está jogando uma missão. Além de representar o aspecto físico do personagem – esta classe herda de `GameObject`, classe oferecida pelo arcabouço –, ela também armazena informações abstratas que dizem respeito ao jogador, como por exemplo o conjunto de itens coletados e a sua pontuação atual.

Uma referência à instância atual de Player é disponibilizada através da classe G, conforme visto anteriormente, porque os outros elementos do cenário podem interagir com o jogador, e portanto precisam estar cientes de sua posição, além de terem a possibilidade de modificar a pontuação e o conjunto de itens do mesmo.

Nesta classe, o destaque vai para dois trechos de código, relativamente curtos, que correspondem a todo o controle de movimentação e animação do personagem. Esta lógica, que, para ser implementada inteiramente, poderia ser bastante complexa e dispendiosa em termos de tempo, foi rapidamente produzida com cerca de 40 linhas de código, conforme abaixo:

```
def update
  ...
  forces = Vector.new 0, 0
  if KB.key_down? Gosu::KbLeft
    set_direction :left if @facing_right
    forces.x -= @bottom ? 0.25 : 0.03
  elsif @speed.x < 0
    forces.x -= 0.1 * @speed.x
  end
  if KB.key_down? Gosu::KbRight
    set_direction :right unless @facing_right
    forces.x += @bottom ? 0.25 : 0.03
  elsif @speed.x > 0
    forces.x -= 0.1 * @speed.x
```

```
  end
  if @bottom
    if @speed.x != 0
      animate @anim_indices, 30 / @speed.x.abs
    elsif @facing_right
      set_animation 6
    else
      set_animation 0
    end
    if KB.key_pressed? Gosu::KbSpace
      forces.y -= 13.7 + 0.4 * @speed.x.abs
    end
  end
  move forces, G.scene.obsts, G.scene.ramps
  ...
end

def set_direction dir
  if dir == :left
    @facing_right = false
    @anim_indices = @anim_indices_left
  else
    @facing_right = true
    @anim_indices = @anim_indices_right
  end
  set_animation @anim_indices[0]
end
```

É válido destacar no código acima a construção em diversos passos de um vetor de forças, que será posteriormente passado como parâmetro para o método `move`, fornecido pelo módulo `Movement` do arcabouço. Este modo de construir o vetor pode ser associado, correta e intuitivamente, com a ideia de somar as diversas forças atuantes no objeto para obter uma força resultante; a única diferença é que a força da gravidade é adicionada automaticamente dentro do método `move` – já que ela atua sobre todos os corpos, seria redundante ficar escrevendo a soma desta força para todos eles.

3.3.3. Classe “Scene”

A classe `Scene` representa, como o nome sugere, as cenas do jogo, ou seja, as “telas” que compõem o núcleo do jogo; representa o cenário, e portanto contém referências para os diversos elementos contidos neste cenário. Sua principal função é coordenar o comportamento de todos estes elementos e sua interação com o cenário propriamente dito.

A inicialização de uma instância da classe `Scene` envolve a leitura de um arquivo de texto que descreve, numa linguagem relativamente simples, os elementos que estarão contidos naquela cena. Avaliando este arquivo, pode-se ter um primeiro vislumbre de por que o jogo é extensível: não apenas as fases, mas todos os elementos, como veremos mais adiante, são definidos por arquivos de texto similares, permitindo a definição de novos elementos ou a alteração de elementos existentes sem a necessidade de programar.

A seguir, um exemplo de arquivo de definição de cena:

```
> 1000,400,l
> 410,10,r
< 0,395,l,2,0,i
< 160,250,x,2,1,n,5
  0,395,709,1
  948,521,252,1
  901,267,306,1
  0,358,253,1
  0,375,268,1
/ 660,267,241,120
\ 709,395,239,126
! 490,30,apple
! $6
! 590,30,item2
? 1070,127,1
* 138,158,1
```

Basicamente, cada linha do arquivo codifica um elemento. O primeiro caractere da linha indica o tipo de elemento, e os valores que seguem são parâmetros adicionais – como as coordenadas do objeto no mapa –, em geral passados diretamente para os construtores das classes correspondentes. A tabela 2 mostra a correspondência entre caracteres e tipos de elementos, assim como a classe associada a esses tipos:

Caractere	Tipo de elemento	Classe
>	Entrada	`Entry`
<	Saída	`Exit`
[espaço em branco]	Obstáculo	`Block`
/	Rampa ascendente para a direita	`Ramp`
\	Rampa ascendente para a esquerda	`Ramp`
!	Item	`Item`
?	Personagem	`NPC`
*	Objeto	`SceneObject`

Tabela 2: codificação de elementos do cenário no arquivo texto lido pela classe `Scene`.

Para a interpretação destes arquivos, também foi notável a contribuição da linguagem Ruby. Por ser uma linguagem interpretada, é possível executar código diretamente a partir de texto, ou, em outras palavras, uma *string* pode ser usada como uma linha de código propriamente dito. Isto pode ser feito através da função `eval`, fornecida na biblioteca padrão da linguagem. Na classe `Scene`, esta função é usada em construções como:

```
eval "@items << Item.new(info[0].to_i, info[1].to_i, :#{info2})"
```

No exemplo acima, a variável `info2` é, na verdade, uma *string* contendo uma série de valores separados por vírgula, que foi lida de um outro arquivo de texto, no qual são definidos os itens do jogo. Ao ser utilizada na função `eval`, essa *string* será interpretada como uma sequência de parâmetros sendo passados para o construtor de `Item`.

Uma vez lidos, interpretados e instanciados todos os elementos do cenário, a instância de `Scene` guardará referências para todos estes objetos, de modo que em seu método `update`, ela chama o método `update` de cada um deles – além de executar algumas ações de controle, como centralizar a câmera em relação ao jogador –, e, no método `draw`, ela chama o método `draw` de cada um deles.

3.3.4. Classes "Item", "NPC" e "SceneObject"

Estas classes foram apresentadas na seção anterior como elementos contidos numa cena, ou seja, elementos do cenário. Como tal, apresentam diversas características em comum:

- São elementos que aparecem fixos numa posição do cenário.
- Interagem com o jogador, de uma maneira mais significativa que os simples obstáculos ou rampas.
- São subclasses de `GameObject` (classe fornecida pelo arcabouço).
- Seus detalhes são definidos em arquivos de texto.

Como se pode esperar, estes objetos têm seus atributos físicos (posição e tamanho), e a interação com o personagem principal (controlado pelo jogador) se dá mediante o contato ou proximidade. Por isso, todos eles recebem como parâmetros as coordenadas x e y e a largura e altura do retângulo delimitador.

Porém, cada uma dessas classes apresenta também suas particularidades, as quais identificaremos melhor conforme observamos exemplos de arquivos de definição. Os itens são os mais simples dentre estes elementos, e na verdade todos são definidos num mesmo arquivo, em particular porque é possível definir cada um em apenas uma linha.

```
apple,30,30,0,0,100
item1,20,20,-6,-6,50
item2,22,32,-4,-4,60
item3,2,1,-15,-31,70
```

O primeiro valor é o identificador do tipo de item (pois podem aparecer várias instâncias de um mesmo tipo de item ao longo das cenas); o segundo e o terceiro são a largura e a altura, conforme explicado acima; o quarto e o quinto são as componentes do `img_gap` do `GameObject` (ver seção 2.3.6); o sexto é a pontuação conferida ao jogador por coletar o item.

Note-se que, no arquivo de definição da cena, há três parâmetros para a definição de um item, que são as coordenadas x e y e o identificador de tipo. As informações contidas no arquivo de definição de itens são aquelas que não variam de instância para instância, por isso as coordenadas x e y não podem ficar neste arquivo.

Agora, observemos um arquivo de definição de um NPC:

```
70,140,2,0,l
Olá, você veio visitar a fazenda? Infelizmente, estamos com um problema sé-
rio... Aliás, será que você não poderia nos ajudar?
-Sim, claro!
-Não.
-Por que deveria?
+1 10 $1
```

```
$1 Ah, que ótimo! Nesse caso... Bom, estamos com uma praga de um inseto que
está atacando as abelhas, .../e com isso a polinização está prejudicada, e
não conseguimos produzir nada./Porém, parece que há uma planta da qual po-
demos extrair um produto para espantar esses insetos./Vá à casa do sr.
José, ele te explicará melhor...
-OK.
+1 0 $2

$2 Ficarei aguardando você retornar com o produto.

$3 Você já o conseguiu?
!apple 30 $4 $5 .

$4 Ótimo! Agora, precisamos aplicar o produto... Siga em frente para chegar
às plantações.

$6 Muito obrigado por toda a ajuda!
```

Na primeira linha, estão atributos semelhantes aos encontrados na definição do item (largura, altura e componentes do `img_gap`), com um atributo final diferenciador, que define se o NPC aparecerá inicialmente "olhando" para a esquerda (com o valor "l") ou para a direita (no caso do valor "r").

A seguir, vemos conjuntos de linhas separados por linhas em branco. Cada um destes conjuntos corresponde a um "estado" do NPC. O NPC vai alterando de estado conforme o jogador interage com ele e/ou são ativadas *switches*. As *switches* são justamente "gatilhos" disparados por determinadas ações do jogador, e que afetam outros elementos do jogo, como NPCs e objetos. Por exemplo, coletar um item pode ativar uma *switch*, a qual vai alterar o estado do NPC que estava aguardando o item para um estado em que ele aceita receber este item (a transição entre o terceiro e o quarto estados do NPC acima é um exemplo deste tipo).

Assim, na primeira linha de cada estado, aparece a mensagem que o NPC fala quando o jogador interage, opcionalmente precedida do número da *switch* que ativa este estado. Esta é a única linha obrigatória para todos os estados; a seguir, virão as opções de resposta, caso a mensagem seja uma pergunta; finalmente, na última linha do estado ficam as indicações de como passar para o próximo estado, quando isto pode ser feito respondendo a uma pergunta ou entregado um item (a resposta correta é indicada pelo caractere "+" seguido do índice da resposta, o item é indicado pelo caractere "!" seguido pelo identificador), e de ações que são disparadas nesta transição (como, por exemplo, ganhar pontos, ativar uma *switch* ou receber um item).

Por fim, observemos um arquivo de definição de um objeto do cenário (`SceneObject`):

```
40,200,8,1
-opção 1
-opção 2
-opção 3
+1 15 $3/certa resposta!/[0]

$3
!item2 30/certa resposta!/[1,2,3,2,1]
```

```
!item1 50 $4 $8/certa resposta!/[1]

$4
-opção 1
-opção 2
+2 80 $5 !apple/certa resposta!/[1,2,3,4,5,6,7]

$5
```

A distribuição dos dados neste arquivo segue uma lógica bastante semelhante àquela observada no arquivo do NPC. A primeira linha contém parâmetros a serem passados para o construtor de `GameObject`, que nesse caso são largura, altura, `sprite_cols` e `sprite_rows` (ver seção 2.3.6). Seguem os estados, cada um especificado por um conjunto de linhas separadas das demais por uma linha em branco.

No caso de `SceneObject`, não há "falas", mas mensagens exibidas na tela conforme a ação correta é tomada (por isso estas mensagens aparecem na última linha de cada estado, que contém informações sobre a transição, assim como no caso de `NPC`); Outra diferença em relação aos NPCs é que, além da mensagem de transição, há uma sequência de números, delimitada por "[" e "]", a qual corresponde a índices da animação a ser executada por ocasião da transição de estados. Mais uma vez, a função `eval` de Ruby mostra-se conveniente, pois permite a interpretação direta desta sequência de números como um *array*.

3.3.5. Classes "Chart" e "Series"

Estas não estão entre as principais classes da arquitetura, mas merecem ser citadas para enfatizar a facilidade trazida pela combinação da linguagem Ruby com o arcabouço MiniGL. Elas são responsáveis – tendo `Series` apenas um papel auxiliar – por renderizar o gráfico de linhas da tela de detalhes de pontuação dos jogadores. E, surpreendente, toda esta lógica está contida em pouco mais de sessenta linhas de código!

Para começar, observemos como é definida a classe `Series`:

```
Series = Struct.new :name, :color, :data
```

Sim, é apenas isso. O código acima utiliza uma classe da biblioteca padrão de Ruby, `Struct`, para criar uma nova classe (`Series`) a partir de três atributos. O resultado é uma classe com um construtor que recebe e armazena estes atributos, e ainda os expõe através de métodos que funcionam como "*getters*" e "*setters*", relembrando a mesma analogia que fizemos na seção referente à classe `G`.

Como já mencionado, esta é apenas uma classe auxiliar para guardar os dados do gráfico, sendo toda a lógica de desenho responsabilidade da classe `Chart`. Nesta última, há alguns pontos a destacar. Observemos o trecho abaixo:

```
@x_max = 0
series.each {|s| @x_max = s.data.length if s.data.length > @x_max}
@y_max = 0
series.each {|s| @y_max = s.data.max if s.data.max and s.data.max > @y_max}
```

Estas quatro linhas permitem a obtenção, com pouquíssimo esforço, da série mais longa e do maior valor dentre todos os pontos de todas as séries, armazenados em `@max_x` e `@max_y`, res-

pectivamente. Estes parâmetros são usados posteriormente para definir a distribuição dos pontos na área destinada ao gráfico.

Por fim, no método `draw`, é possível ver diversas chamadas a `G.win.draw_line` e `G.win.draw_quad`. Conforme foi visto anteriormente, `G.win` é uma referência à janela do jogo, e `draw_line` e `draw_quad` são métodos fornecidos pela biblioteca Gosu para desenhar linhas e quadriláteros na tela.

3.4. Conclusões

Vimos, ao longo deste último capítulo, uma descrição detalhada, em parte apenas sob a visão de um usuário comum, em parte sob uma perspectiva mais técnica, de um jogo educativo feito em Ruby, utilizando a biblioteca Gosu e o arcabouço MiniGL, construído sobre esta biblioteca. O primeiro ponto a ser ressaltado são as sutilezas que foram levadas em conta para se obter o efeito de atratividade desejado, e se seguir o princípio de simplicidade de interface, tendo em vista que o jogo é voltado para crianças.

Mais do que a qualidade do auxílio educacional oferecido pelo jogo, procurou-se enfatizar a arquitetura do código e o projeto da interface gráfica, além de exemplificar os vários usos da biblioteca MiniGL e como ela contribuiu para a produção de código mais sucinto – isto principalmente por se tratar de um projeto de Ciência da Computação, e não de Educação. Esta última é uma área de interesse do autor, mas produzir um jogo comprovadamente eficaz em auxiliar e avaliar o aprendizado das crianças requereria um aprofundamento muito maior na área de Educação, em detrimento dos estudos na área de Computação que tiveram de ser feitos para tornar possível este projeto.

Em relação à efetividade do jogo como ferramenta educativa, pouco pode ser dito, devido ao pouco tempo que se pôde dedicar a expô-lo ao teste de usuários reais – ou seja, crianças e educadores. Além da limitação de tempo, a própria incompletude do conteúdo do jogo dificultou estas avaliações. Por outro lado, numa visão geral, o jogo alcança o objetivo esperado no aspecto computacional: o código-fonte completo do jogo possui apenas cerca de duas mil linhas, um valor significativamente abaixo do esperado, considerando-se a quantidade de *features* apresentadas.

Adiciona-se a isso, ainda, o fato de que o jogo é extensível. Pode-se adaptar os elementos do conteúdo para melhor atender às necessidades educacionais do público de estudantes pretendido, assim como adicionar elementos novos, com a simples configuração de arquivos de texto (para o caso de adicionar elementos novos, seria necessário também incluir imagens nos diretórios de dados do jogo). Nenhuma compilação ou qualquer outro tipo de processo é necessário, as alterações têm efeito imediato. Tudo isso, é claro, ainda é favorecido pelo fato de se tratar de um projeto de código aberto, permitindo, para um grupo mais seleto de pessoas, uma personalização mais profunda do software.

Assim, pode-se dizer que o projeto atingiu suas principais metas de forma satisfatória. Há, claramente, diversos aspectos a aprimorar, assim como no caso do arcabouço. Poder-se-ia, por exemplo, expandir o jogo para abarcar outras áreas do conhecimento, ou ainda criar outros jogos com gêneros diferentes, utilizando meios diferentes de apresentar e avaliar o aprendizado. Porém, apresentam-se aqui estas imperfeições ou incompletudes com a esperança de

que elas possam servir de incentivo para futuros projetos voltados para esta área. A Educação necessita de novas ferramentas e métodos, e qualquer acréscimo, qualquer contribuição que se possa fazer neste meio científico, poderá ser considerada bem-vinda.

4. Referências bibliográficas

[1] TOMONARI, Rachelle F. D. *Stages of Growth Child Development: Early Childhood (Birth to Eight Years), Middle Childhood (Eight to Twelve Years)*. Disponível em <http://education.stateuniversity.com/pages/1826/Child-Development-Stages-Growth.html>. Acesso em: 27 set 2014.

[2] *Wikipedia, the free encyclopedia: Education Index*. Disponível em: <http://en.wikipedia.org/wiki/Education_Index>. Acesso em: 27 set 2014.

[3] *Wikipedia, the free encyclopedia: List of countries by Human Development Index*. Disponível em: <http://en.wikipedia.org/wiki/List_of_countries_by_Human_Development_Index>. Acesso em: 27 set 2014.

[4] *Wikipedia, the free encyclopedia: Four Asian Tigers*. Disponível em: <http://en.wikipedia.org/wiki/Four_Asian_Tigers>. Acesso em: 27 set 2014.

[5] TRYBUS, Jessica. *Game-Based Learning: What it is, Why it Works, and Where it's Going*. Disponível em: <http://www.newmedia.org/game-based-learning--what-it-is-why-it-works-and-where-its-going.html>. Acesso em: 27 set 2014.

[6] *Games for learning*. Disponível em: <http://interactioneducation.com/index_files/games_for_learning.html>. Acesso em: 30 abr 2014.

[7] FLETCHER, J. D.; TOBIAS, Sigmund. Reflections on “A Review of Trends in Serious Gaming”. Disponível em <http://rer.sagepub.com/content/82/2/233.abstract>. Acesso em: 27 set 2014.

[8] *Computer Languages Benchmarks Game: Which programs are fastest?* Disponível em: <http://benchmarksgame.alioth.debian.org/u32q/which-programs-are-best.php>. Acesso em: 27 set 2014.

[9] *RubyGems Guides: Gems with extensions*. Disponível em: <http://guides.rubygems.org/gems-with-extensions/>. Acesso em 27 set 2014.

[10] GAMBLE, Adam; CARNEIRO JR., Cloves; AL BARAZI, Rida. *Beginning Rails 4*. 3 ed. Nova York: Apress, 2013.

[11] SANTOS, Robson Rodrigues dos. *Panorama do Mercado de Jogos Educativos no Brasil*. São Caetano do Sul, 2010.

[12] SCHELL, Jesse. *The Art of Game Design*: A Book of Lenses. Burlington: Morgan Kaufmann, 2008.

[13] CHAMMAS, Adriana; MORAES, Anamaria; TEIXEIRA, Eduardo. *Avaliação Heurística e Avaliação Cooperativa em interfaces de games para crianças: um estudo comparativo*. In: 4º Congresso Sul-Americano de Design de Interação, Brasil, 2012.

II. Parte Subjetiva

1. Desafios e frustrações

Durante o último ano, especialmente o último semestre, a minha "lista de coisas para fazer" parecia simplesmente não diminuir. As pendências eram muitas e frequentemente me levavam a um estado de quase desespero, diante da possibilidade de não cumprir com os prazos – e, com isso, não cumprir a meta de terminar o curso em quatro anos.

O TCC, por si só, representou a ocupação de uma fração relevante de meu tempo, consumindo consistentemente longas horas, por todas as semanas dos últimos quatro ou cinco meses. Mas tudo tornou-se mais complicado devido às atividades extra-classe exigidas por várias das disciplinas que eu estava cursando, simultaneamente. Ainda se somou a isso o fato de eu estar trabalhando, durante todo este tempo, também.

Desta maneira, posso afirmar com certeza que nunca tive que ter tamanha disciplina com a administração do tempo. Passei, mais do que nunca, a valorizar cada hora e cada minuto, tendo em mente que cada minuto de ócio significava um minuto a menos para me ajudar a cumprir com todos os prazos. Enfim, foi um período sufocante, mas por outro lado uma experiência única, da qual se pôde extrair muita coisa boa – dentre as quais provavelmente a mais intensa foi a disciplina.

Em contrapartida, o desenvolvimento do trabalho, propriamente dito, não ofereceu grandes desafios técnicos – escolhi um tema e uma linguagem com os quais já estava familiarizado. De fato, é possível que a escrita da monografia, e de todos os outros materiais escritos envolvidos no projeto (relatórios preliminares, assim como a própria documentação do arcabouço) tenham constituído os maiores desafios.

É claro que não atribuo a relativa tranquilidade na superação de obstáculos apenas à escolha do tema e das tecnologias: posso afirmar, com segurança, que adquiri uma boa base das teorias da Computação durante o curso de Bacharelado. Além disso, houve o apoio do orientador, de colegas e familiares que, com pequenas sugestões, facilitaram a solução de alguns dos impasses que surgiram ao longo do processo – mas, dedicada a estes, há uma seção inteira de agradecimentos mais à frente.

Provavelmente uma das maiores frustrações encontradas no projeto foi a impossibilidade de medir, a tempo hábil e com a devida precisão, o real impacto do uso do jogo para apoiar a educação de crianças. Seria necessário fazê-lo ser utilizado por um grupo maior de alunos, durante um intervalo de tempo mais longo, e fazer uma análise mais detalhada, com metodologias estatísticas corretas, para inferir medidas mais realistas da sua contribuição. Contudo, embora não possa fazer parte dos resultados apresentados aqui, esse estudo pode ser feito num futuro próximo – e para passos futuros também há uma seção dedicada.

2. A contribuição do curso de Computação

Aqui listarei algumas das disciplinas ou conceitos que julgo terem tido papéis mais relevantes na contribuição para possibilitar a produção deste trabalho:

- MAC0110 – Introdução à Computação: Esta disciplina constitui um primeiro passo no aprendizado de Computação, e todos nós precisamos começar do começo; Os conceitos básicos desta ciência foram apresentados de uma forma interessante nesta disciplina.
- MAC0122 – Princípios de Desenvolvimento de Algoritmos: Constitui uma espécie de continuação, com maior aprofundamento e formalidade, da disciplina MAC0110; assim, é também um dos blocos fundamentais na formação de um cientista da Computação.
- MAC0323 – Estruturas de Dados: Não há como desenvolver um sistema computacional eficiente (quando se tem um nível mínimo de complexidade, claramente) sem conhecer os conceitos de estruturas de dados. Elas são usadas, literalmente, o tempo inteiro, e conhecer os diversos tipos e suas vantagens e desvantagens é indispensável.

Também merecem ser enumeradas outras disciplinas que, embora não tenham tido aplicação direta neste projeto, especificamente, considero que tenham um valor inestimável como parte de minha formação:

- MAE0121, MAE0212 – Introdução à Probabilidade e à Estatística I e II: É inegável o vasto campo de aplicações da Estatística. Em Computação, especificamente, os conceitos estatísticos e de probabilidade aparecem com grande frequência, e, mais uma vez, é necessária uma base para se capturar estes conceitos.
- MAC0338 – Análise de algoritmos: Se uma parte do trabalho envolvido na criação de sistemas computacionais eficientes está relacionada ao conhecimento de estruturas de dados e de tecnologias, outra parte igualmente importante diz respeito ao conhecimento de técnicas de programação e dos métodos que permitem avaliar a eficiência das técnicas.
- MAC0438 – Programação Concorrente: A demanda por poder computacional cresce incessantemente, mas nossa tecnologia para aprimoramento do hardware está começando a alcançar seus limites. Assim, torna-se cada vez mais importante a utilização de diversos computadores (ou processadores) para aumentar o poder computacional. Esta disciplina oferece uma boa base dos conceitos intrínsecos – a princípio complicados – à programação para múltiplos processadores.

As muitas outras disciplinas do curso também tiveram, é claro, sua contribuição, constituindo uma vasta riqueza intelectual que será um legado permanente para aqueles que passaram por este curso.

3. Próximos passos

Conforme citado brevemente em seções anteriores, nem tudo acaba aqui! Pretendo levar adiante, pelo menos até uma maior estabilização, ambos os componentes do trabalho – arcabouço e jogo.

Com relação ao arcabouço, ele será mantido disponível para a comunidade, e nisso está atrelado um compromisso, que é o de dar manutenção e disponibilizar material de documentação. Porém, tenho a intenção de não apenas honrar com os compromissos mínimos, mas também manter o progresso da biblioteca, até porque pretendo seguir desenvolvendo jogos com Ruby. Na verdade, já há diversas ideias para serem implementadas numa "MiniGL 2.0".

No que tange a "Aventura do Saber", pretendo, pelo menos, tentar implantar o jogo em alguma instituição de ensino, de modo a viabilizar o estudo de impacto discutido na seção 1. A depender dos resultados deste estudo, eu poderia decidir por aprimorar o projeto atual ou expandi-lo para abranger mais áreas do conhecimento e, possivelmente, mais faixas etárias.

De todo modo, não há como prometer nada, apenas especular. Ainda assim, é certo que as experiências adquiridas ao longo do desenvolvimento deste projeto me serão úteis por longos anos, durante minha atuação profissional e, também, para o desenvolvimento de projetos independentes.

4. Agradecimentos

Agradeço, em primeiro lugar, à minha mãe, cuja contribuição para este projeto foi nada menos do que fornecer a ideia principal (jogo educacional). A ela e aos meus demais familiares, agradeço pelo apoio incondicional, não apenas nessa época conturbada do desenvolvimento do projeto, mas em todas as demais etapas de minha formação.

Agradeço, também, fortemente, aos meus amigos e colegas de curso. A jornada que enfrentamos foi longa e tortuosa, mas nada comparada ao que seria se não houvesse outros na mesma situação, compartilhando alegrias e frustrações.

Não posso deixar de agradecer, é claro, ao meu orientador, Prof. Gubi, pelo apoio fundamental na definição do escopo do projeto, e por ter aceitado supervisioná-lo, acreditando na minha ideia, mesmo sem conhecer as tecnologias que eu pretendia usar.

Por fim, também devo lembrar de todos os outros professores e funcionários do IME e da USP, em geral, que colaboram para criar o ambiente amigável e agradável que temos nesta Universidade.

A todos vocês, meu sincero “Obrigado por tudo!”.

III. Apêndice

1. Tecnologias utilizadas

- Sistema operacional Linux Mint – http://www.linuxmint.com/
- Linguagem de programação Ruby – https://www.ruby-lang.org/pt/
- Biblioteca Gosu – http://www.libgosu.org/
- Inkscape (produção de gráficos) – https://inkscape.org/pt/
- Linux Multimedia Studio (produção de músicas) – http://lmms.sourceforge.net/
- RubyMine (IDE) – https://www.jetbrains.com/ruby/
- gedit (edição de aquivos de texto) – https://wiki.gnome.org/Apps/Gedit

2. Links do projeto

- Página do TCC – https://linux.ime.usp.br/~victords/mac0499/
- Repositório do arcabouço – https://github.com/victords/minigl
- Repositório do jogo – https://github.com/victords/aventura-do-saber

3. Como executar o jogo

Até o momento da entrega do projeto, não foi criada uma versão distribuível do jogo, portanto para testá-lo é necessário instalar algumas bibliotecas e executar o jogo a partir da linha de comando. Seguem abaixo os passos para realizar esta tarefa num sistema Linux:

- Instalar o interpretador Ruby, versão 2.0 ou superior.
- Instalar os pacotes `libsdl2-dev`, `libsdl2-ttf-dev`, `libpango1.0-dev`, `libgl1-mesa-dev`, `libfreeimage-dev`, `libopenal-dev` e `libsndfile-dev`.
- Num terminal, executar `gem install minigl` (o programa `gem` deve ser instalado juntamente com o interpretador Ruby).
- Baixar o código-fonte do jogo a partir do repositório do GitHub, navegar até a pasta onde ele foi baixado, pelo terminal, e executar `ruby game.rb` (os arquivos de código propriamente ditos encontram-se na pasta `src` do projeto).