

UNIVERSIDADE REGIONAL DE BLUMENAU
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

UM SIMULADOR 2D PARA A LINGUAGEM ROBOTDY

DIOGO DA SILVA

BLUMENAU
2016

DIOGO DA SILVA

UM SIMULADOR 2D PARA A LINGUAGEM ROBOTUY

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Profa. Joyce Martins, Mestre - Orientadora

**BLUMENAU
2016**

UM SIMULADOR 2D PARA A LINGUAGEM ROBOTDY

Por

DIOGO DA SILVA

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente: _____
Profa. Joyce Martins, Mestre – Orientadora, FURB

Membro: _____
Prof. Dalton Solano dos Reis, Mestre – FURB

Membro: _____
Prof. Aurélio Faustino Hoppe, Mestre – FURB

Blumenau, 30 de junho de 2016

Para minha família que, em meio a discussões acaloradas, é possível sentir a afeição de uns pelos outros.

AGRADECIMENTOS

Primeiramente gostaria de agradecer a mim, pois se eu não existisse, este trabalho também não existiria.

Em segundo lugar a minha família, que sempre me encoraja a correr atrás daquilo que quero com frases acolhedoras como “Estamos aqui se precisar.” ou “Trabalhando duro conseguiremos atingir nossos sonhos.” ou “Vem jantar se não vamos comer tudo!”.

Em seguida agradeço a mulher da minha vida por manter vivo o meu lado humano em meio a este universo exato e computacional.

Por fim, mas não menos importante, a minha orientadora, mestre e amiga. Por aguentar por cinco meses minhas piadas sem graça, e-mails constantes com assinaturas extensas, loucuras de desenvolvimento de simuladores, mas principalmente por me olhar com ar de “Esse rapaz é doido!!!” sempre que eu explicava algum assunto extra ao término das minhas orientações.

Fácil né galera?

Danton Cavalcanti Franco Júnior

RESUMO

Este trabalho descreve a implementação de um simulador 2D para a linguagem Robotoy proposta por Torrens (2014). Para o desenvolvimento do cenário onde as simulações ocorrem foi utilizada a biblioteca *Java Open Graphic Language* (JOGL), que é a extensão da biblioteca *Open Graphic Language* (OpenGL) para Java. Para efetuar as simulações foi criada uma linguagem intermediária em formato de *script* que é interpretada pelo simulador com a finalidade de executar as ações correspondentes aos comandos da linguagem Robotoy. Alguns comandos originais da Robotoy foram suprimidos por não serem estritamente relevantes em um cenário simulado. O simulador dispõe de um compilador de Robotoy para a linguagem intermediária, um editor de cenários 2D dividido em células que também serve como cena para a execução de simulações, além do controle da simulação propriamente dito. Atualmente a ferramenta suporta cenários de até 35x35 células sem apresentar lentidão. Caso os cenários sejam maiores que isso, a ferramenta começa a perder a fluidez nas execuções. Os resultados obtidos apresentaram que o ambiente desenvolvido simula de maneira eficiente os cenários reais que foram testados pela Robotoy de Torrens (2014).

Palavras-chave: Simulação. Robótica. Interpretadores. *Script*. OpenGL.

ABSTRACT

This work describes the implementation of a 2D simulator to the Robotoy programming language suggested by Torrens (2014). The Java Open Graphic Language (JOGL), which is an extension from Open Graphic Language (OpenGL) for Java, was used to develop the scenes where the simulation occurs. In order to make the simulator works properly, a middle script language was built, and the simulator works as a interpreter to execute the script actions. Some original commands from Robotoy were suppressed because they weren't strictly necessary for a simulated scenario. The simulator have a compiler from Robotoy to the script language, a 2D scene editor divided in cells where the simulation happens. Currently the tool supports scenes up to 35x35 tiles with ease, if the scenarios are greater than this, the tool start to show slowness in the simulation process. The results obtained show that the developed environment efficiently simulates the real scenarios tested with Robotoy from Torrens (2014).

Key-words: Simulation. Robotics. Interpreters. Script. OpenGL.

LISTA DE FIGURAS

Figura 1 - Processo de desenvolvimento de uma simulação	14
Figura 2 - Cenário do robô preso na mina	19
Figura 3 - Exemplo da mina na interface do Furbot.....	20
Figura 4 - Ambiente RoboMind FURB	21
Figura 5 - Pond desenvolvido com o Blockly	22
Figura 6 - Tela do simulador 2D Robotoy.....	29
Figura 7 - Diagrama de classes do compilador.....	30
Figura 8 - Diagrama de classe do editor de cenários.....	31
Figura 9 - Diagrama de atividades de uma simulação.....	32
Figura 10 - Exemplo de conversão da posição do <i>mouse</i> para a posição da matriz do cenário.....	38
Figura 11 - Compilador	40
Figura 12 - Editor de cenários	40
Figura 13 – Seleção de células	41
Figura 14 – Controle da simulação.....	42
Figura 15 - Exemplo de cenário para o robô acumulador	43
Figura 16 - Texturas	54

LISTA DE QUADROS

Quadro 1 - Comparação entre JavaScript e Java	17
Quadro 2 - Programa na linguagem Robotoy	18
Quadro 3 - Programa do robô que encontra a saída de uma mina.....	19
Quadro 4 - Programa exemplo do Furbot.....	20
Quadro 5 - Comandos a serem executados pelo robô	24
Quadro 6 - Comandos para declaração e atribuição de valores a variáveis	26
Quadro 7 - Comandos de controle de fluxo e declaração / chamada de rotinas.....	27
Quadro 8 - Método <code>executarProximaAcao</code> da classe <code>ControleSimulacao</code>	33
Quadro 9 - Método <code>executarExpressao</code> da classe <code>ControleSimulacao</code>	37
Quadro 10 - Método <code>setTileAt</code> da classe <code>Cenario2DJOGL</code>	38
Quadro 11 - Solução do exercício do robô acumulador	43
Quadro 12 - Comparativo com os trabalhos correlatos	45
Quadro 13 - Especificação sintática da linguagem intermediária	50
Quadro 14 - Exercício 1: labirinto	52
Quadro 15 - Exercício 2: encontrar água.....	53

LISTA DE ABREVIATURAS E SIGLAS

FURB - Universidade Regional de Blumenau

IDE - *Integrated Development Environment*

JOGL – *Java Open Graphic Language*

OpenGL – *Open Graphic Language*

RF – Requisito Funcional

RNF – Requisito Não Funcional

SUMÁRIO

1 INTRODUÇÃO.....	12
1.1 OBJETIVOS.....	13
1.2 ESTRUTURA.....	13
2 FUNDAMENTAÇÃO TEÓRICA	14
2.1 SIMULAÇÃO	14
2.2 LINGUAGENS INTERPRETADAS E INTEPRETADORES.....	16
2.3 A LINGUAGEM ROBOTY E A APLICAÇÃO DA ROBÓTICA NA EDUCAÇÃO .	17
2.4 TRABALHOS CORRELATOS	19
2.4.1 FURBOT	19
2.4.2 ROBOMIND FURB	20
2.4.3 BLOCKLY	21
3 DESENVOLVIMENTO	23
3.1 REQUISITOS.....	23
3.2 ESPECIFICAÇÃO DA LINGUAGEM INTERMEDIÁRIA	23
3.3 ESPECIFICAÇÃO DO SIMULADOR.....	29
3.4 IMPLEMENTAÇÃO	33
3.4.1 Técnicas e ferramentas utilizadas.....	33
3.4.2 Método responsável por interpretar as ações	33
3.4.3 Método responsável por editar o cenário	37
3.4.4 Operacionalidade da implementação	39
3.5 RESULTADOS E DISCUSSÕES.....	43
4 CONCLUSÕES.....	46
4.1 EXTENSÕES	46
APÊNDICE A – ESPECIFICAÇÃO DA LINGUAGEM INTERMEDIÁRIA	50
APÊNDICE B – LISTA DE EXERCÍCIOS.....	52
ANEXO A – TEXTURAS	54

1 INTRODUÇÃO

No decorrer da elaboração de processos, espera-se poder averiguar o impacto de decisões em um sistema, bem como visualizar os resultados obtidos. A simulação existe para suprir estas necessidades sendo a técnica que estuda comportamentos e reações de modelos (ERLANG.COM.BR, 2015). Nesse sentido, no universo computacional, tornou-se mais fácil criar ambientes virtuais para simular situações específicas como, por exemplo, a de um *call center* para averiguar as melhores escolhas em termos de eficácia de atendimento e resposta financeira.

Dentro da robótica a simulação possui um papel muito importante, como citam Bibby e Necessary (2008), em teoria, é a única maneira de validar algoritmos de controle, de agarrar objetos, de efetuar o planejamento de rotas, entre outras situações específicas. Os autores também afirmam que é perigoso testar algumas funcionalidades diretamente em um hardware, pois não é possível prever o impacto que tais ações causariam e é nesse ponto que simular os cenários e possíveis efeitos se faz essencial.

A robótica é uma das apostas das universidades para atrair alunos para o campo tecnológico. Apresentar-lhes a programação dessa forma motiva a construção de programas e o desenvolvimento criativo (BENITTI et al., 2010). A robótica educacional é um ótimo meio para desenvolver o raciocínio lógico de uma criança, estimulando a sua capacidade de formular e resolver problemas, mesmo que, previamente, essa criança nunca tenha tido qualquer contato com uma linguagem de programação ou um modo lógico de raciocínio (OLIVEIRA; SILVA; ANDRADE, 2013).

Com o intuito de facilitar o desenvolvimento de programas para robôs Mindstorms NXT da Lego, em 2014 foi proposta a linguagem Robotoy, uma linguagem mais simples que pode ser facilmente associada por crianças (TORRENS, 2014). No entanto, um *kit* do Mindstorms NXT possui um custo elevado que pode inviabilizar o uso da linguagem Robotoy por algumas instituições de ensino.

Baseado nesses pressupostos, foi desenvolvida uma ferramenta para simular a execução de programas escritos na linguagem Robotoy, tornando desnecessária a aquisição de um *kit* físico de robótica. Para tanto, deve ser possível criar e editar cenários 2D para a resolução de exercícios, como os propostos por Torrens (2014), com diferentes níveis de dificuldade.

1.1 OBJETIVOS

O objetivo deste trabalho é desenvolver uma ferramenta para simular as ações de um robô em cenários 2D, programadas utilizando a linguagem Robotoy.

Os objetivos específicos do trabalho são:

- a) disponibilizar uma interface para criação e edição dos cenários 2D;
- b) disponibilizar uma interface para a edição de programas na linguagem Robotoy;
- c) executar os programas Robotoy nos cenários previamente criados.

1.2 ESTRUTURA

O trabalho está dividido em quatro capítulos: introdução, fundamentação teórica, desenvolvimento e conclusão. Primeiramente é abordada uma introdução a esse trabalho, em seguida é fundamentada a teoria em que esse trabalho foi embasado. Na sequência são apresentados a especificação e o desenvolvimento do simulador, bem como a especificação da linguagem intermediária. Por fim, são discutidos os resultados e conclusões obtidos.

2 FUNDAMENTAÇÃO TEÓRICA

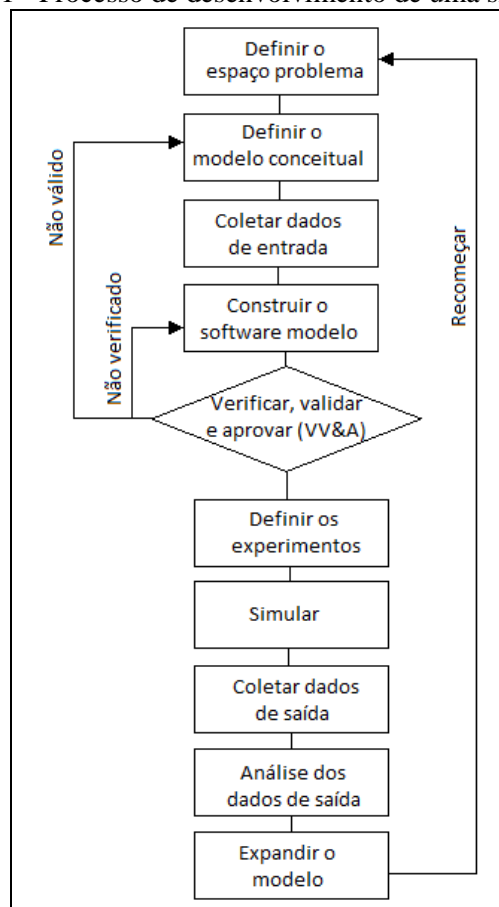
Primeiramente é abordado o tema simulação e sua importância na robótica. Em seguida, são apresentadas linguagens interpretadas e interpretadores, seguidos de uma descrição da linguagem Robotoy e a aplicação da robótica na educação. Por fim, estão relacionados três trabalhos correlatos a este.

2.1 SIMULAÇÃO

A simulação é o processo de modelar um sistema real, permitindo observar seu comportamento e levantar as melhores estratégias para resolvê-lo (SMITH, 1998). A simulação computacional, por sua vez, ganhou grande importância em meados da década de 50 com o surgimento de linguagens orientadas a simulação. A simulação de processos permite que seja feita a análise de um sistema sem interferir no mesmo, interagindo apenas com um modelo computacional que represente a situação real (ERLANG.COM.BR, 2015).

O processo de modelagem e simulação pode ser dividido em uma série de passos ordenados como na Figura 1.

Figura 1 - Processo de desenvolvimento de uma simulação



Fonte: adaptado de Smith (1998).

Conforme Smith (1998), os passos que compõem o processo de modelagem de uma simulação são os seguintes:

- a) definir o espaço problema: o principal objetivo da simulação deve ser definir, explicitamente, o problema para o qual buscam-se resultados;
- b) definir o modelo conceitual: criar o modelo significa bolar os algoritmos que testarão os processos, bem como definir as entradas que serão atribuídas e o que esperar nas saídas. Múltiplos modelos podem ser criados para resolver um mesmo problema;
- c) coletar dados de entrada: após definir os resultados esperados, devem ser coletados os dados de entrada. A melhor opção é coletar dados observáveis em tempo real, reduzindo a margem de erro na entrada de dados (CHUNG, 2003);
- d) construir o software modelo: sugestivamente, é onde se cria o software que irá processar as entradas no modelo definido;
- e) verificar, validar e aprovar (VV&A): é o processo de analisar o comportamento da simulação, verificar se atende as expectativas do modelo e aprovar, ou não, o modelo. Caso o modelo conceitual, testado via software, não atenda à problemática definida, deve-se voltar à fase de definição do modelo conceitual. Esse ciclo é finalizado quando os objetivos definidos são atingidos com o software modelado;
- f) definir os experimentos: aqui deve-se definir os experimentos mais adequados para se obter as respostas esperadas do modelo. Esta parte é ainda mais importante quando a execução das simulações são custosas ou demoradas, tornando a otimização dos experimentos quase obrigatória;
- g) simular: é a execução em si dos experimentos em cima do modelo definido e validado;
- h) coletar dados de saída: neste ponto, são coletados e armazenados os dados gerados pela simulação;
- i) análise dos dados de saída: por vezes os dados coletados podem ser volumosos. Nesse caso, deve-se analisar com cuidado os dados colhidos da melhor maneira possível, via tabelas, gráficos e outras formas específicas;
- j) expandir o modelo: por fim, pode-se optar por expandir o modelo conceitual a partir dos resultados obtidos e recomeçar o processo para melhorar cada vez mais os resultados das simulações.

Segundo Hounsell e Redel (2004), a simulação e a robótica caminham juntas. A importância de testar via software as ações de robôs a fim de preservar seu hardware ou até mesmo baratear os custos de pesquisa são os motivos mais notáveis, sem contar que com o uso de simuladores, não é necessário parar robôs que já operam em modo de produção. Além disso, a simulação permite que os softwares sejam atualizados em paralelo à execução real dos mesmos. Na construção de um simulador para robôs, cinco pontos podem ser levados em consideração, sendo eles:

- a) gatilho: é o evento que dá início a uma ação do robô;
- b) lógica: envolve o processamento de alguma informação e pode gerar ou não uma entrada para o próximo passo, o *timer*;
- c) *timer* (relógio): gera eventos que levam algum tempo para serem completados ou efetua ações até que as mesmas sejam explicitamente interrompidas;
- d) máquina de execução: é responsável por interpretar e executar as ações no robô simulado a partir dos parâmetros recebidos do *timer*;
- e) alvo: os resultados obtidos da máquina de execução alteram as variáveis do cenário onde o robô está inserido e preparam o simulador para um novo ciclo.

2.2 LINGUAGENS INTERPRETADAS E INTEPRETADORES

Com a popularização de linguagens como Java e C#, é comum ver profissionais da área de informática debater o conceito de linguagens interpretadas e compiladas. Basicamente, as linguagens compiladas são transformadas diretamente em código legível aos computadores, como ocorre com o C e o C++. Já as interpretadas geram um código intermediário, interpretado por uma máquina virtual, a qual gera os comandos para o processador (BASTOS, 2008).

A praticidade das linguagens interpretadas, conforme afirmam Batista e Rodriguez (2001), é compatível com o ideal de que: (a) são fáceis de desenvolver, partindo do princípio de que são de alto nível; (b) são reutilizáveis; (c) praticamente todo computador hoje possui pelo menos um interpretador de *scripts*, como um *browser*.

Enquadram-se também nas linguagens interpretadas os *scripts*. Estes, por sua vez, são compostos por uma sequência de comandos e possuem muitos recursos para o processamento de textos. Tornaram-se mais populares com a expansão das páginas de internet. Porém, assim como todas as linguagens interpretadas, tendem a ser mais lentos que as linguagens compiladas (CAMARGO, 2009). Há várias linguagens de *scripts*, como por exemplo Lua, PHP e Ruby, sendo uma das mais populares o JavaScript. Praticamente todo computador tem

um interpretador dessa linguagem, normalmente um *browser*, que é utilizado com frequência (CROCKFORD, 2001). Segundo Crockford (2001), pode-se notar a semelhança sintática do JavaScript com outras linguagens de alto nível, como Java ou C. No Quadro 1 tem-se um comando de seleção `if` codificado em JavaScript e em Java. Nota-se a semelhança entre as linguagens, tanto em termos de construção da expressão associada ao comando quanto nas chamadas de rotina.

Quadro 1- Comparação entre JavaScript e Java

JavaScript		Java	
1	<code>if (horas > 18) {</code>	1	<code>if (horas > 18) {</code>
2	<code> boaNoite();</code>	2	<code> boaNoite();</code>
3	<code>} else {</code>	3	<code>} else {</code>
4	<code> bomDia();</code>	4	<code> bomDia();</code>
5	<code>}</code>	5	<code>}</code>

As linguagens interpretadas, como o nome sugere, precisam passar por um interpretador. Este tem o papel de varrer os programas a fim de interpretá-los, bem como executar as instruções definidas. A funcionalidade dos interpretadores é diferenciada para cada linguagem, portanto não existem fases distintas e específicas que um interpretador precisa ter para ser considerado como tal. Eles não produzem código intermediário e passam todo o seu ciclo de vida lendo o código e executando a ação especificada (GUERBER, 2007).

A construção de interpretadores segue um passo parecido com o da elaboração de compiladores. Ambos analisam um programa de entrada e definem se o programa é válido ou não. Constroem uma estrutura para o programa, dando um significado para ele e armazenam os valores necessários para sua execução, como valores de variáveis e definições ambíguas. A grande diferença entre esses é que os compiladores emitem um programa traduzido em outra linguagem a qual pode obter um resultado, os interpretadores por sua vez produzem resultados através desse processo acima descrito (COOPER; TORCZON, 2014).

2.3 A LINGUAGEM ROBOTROY E A APLICAÇÃO DA ROBÓTICA NA EDUCAÇÃO

A linguagem Robotroy surgiu no âmbito da robótica educacional com o intuito de ensinar às crianças a programação, em específico, de robôs (TORRENS, 2014). A inclusão da robótica no processo educativo traz melhorias em competências básicas como o raciocínio lógico, a criatividade, a autonomia de aprendizado, dentre outras (CASTILHO, 2009).

A intenção da Robotroy, segundo Torrens (2014), é aproximar a programação de uma linguagem comum às crianças, fazendo-as focar nos objetivos e na resolução dos problemas em si, não com sintaxes complicadas e programas difíceis de compreender. Castilho (2009) cita como exemplo a criação de um robô que deve seguir uma fonte de luz e parar quando

próximo dela. Não importa como o programa fora concebido, será avaliado apenas o “produto final”, ou seja, se o robô atingiu ou não o objetivo específico.

A simplificação da linguagem fica por meio de sentenças que são fáceis de serem associadas pelas crianças, pois fazem uso do vocabulário normalmente aprendido aos seis anos de idade (GROLLA, 2006, p. 7). Segundo Torrens (2014), a linguagem Robotoy é dividida em grupos de comandos como:

- a) os direcionados ao robô, divididos em comunicação (emitir sons, escrever no display), orientação (identificar obstáculos), detecção (detectar cores) e locomoção (movimentação do robô);
- b) os de controle de variáveis, para declarar e atribuir valores as mesmas;
- c) os de controle de fluxo, tais como o `se` e o `enquanto`;
- d) os de declaração e invocação de rotinas.

No Quadro 2 abaixo, pode-se verificar a simplicidade de se declarar a variável `tom` do tipo `cor` na linguagem Robotoy (linha 1). Ainda no Quadro 2 (linha 2), verifica-se se o `tom` é diferente (`=/`) de `branco` para que uma sequência de comandos seja executada.

Quadro 2 - Programa na linguagem Robotoy

```

1 cor tom <- cor identificada
2 enquanto tom =/ branco
3     texto tomIdentificado <- "Tom:" . tom
4     escrever tomIdentificado
5     tom <- cor identificada
6 fim do enquanto
7 emitir som

```

Fonte: Torrens (2014, p. 56).

Para validar a linguagem Robotoy foram construídos robôs que possuíam uma medida pré-definida de “passo” (movimentação), cada um correspondente a um quadro de 25cm², de tal forma que “um passo do robô equivale à distância entre uma célula e outra” (TORRENS, 2014, p. 54). Na Figura 2 é possível ver estes quadros de movimentação. Essa figura ilustra um cenário onde o robô deve identificar e se dirigir à saída da “mina”. O programa apresentado no Quadro 3, utilizado para resolver este cenário, consiste em verificar se a passagem do robô está obstruída (linha 1) e, caso não esteja, avançar dois passos e encerrar (linha 4). Do contrário, o robô deve virar à esquerda (linha 2) e verificar novamente (linha 1). Como ainda pode ser observado no Quadro 3, a sintaxe da linguagem Robotoy é definida em alto nível, abstraindo assim os comandos mais complicados, como por exemplo a detecção de obstáculos que, em Java, necessita a inicialização de um sensor ultrassônico bem como a interpretação da leitura do mesmo.

Figura 2 - Cenário do robô preso na mina



Fonte: Torrens (2014, p. 55).

Quadro 3 - Programa do robô que encontra a saída de uma mina

1	enquanto tem obstáculo
2	virar para a esquerda 1
3	fim do enquanto
4	andar para frente 2

Fonte: Torrens (2014, p. 55).

2.4 TRABALHOS CORRELATOS

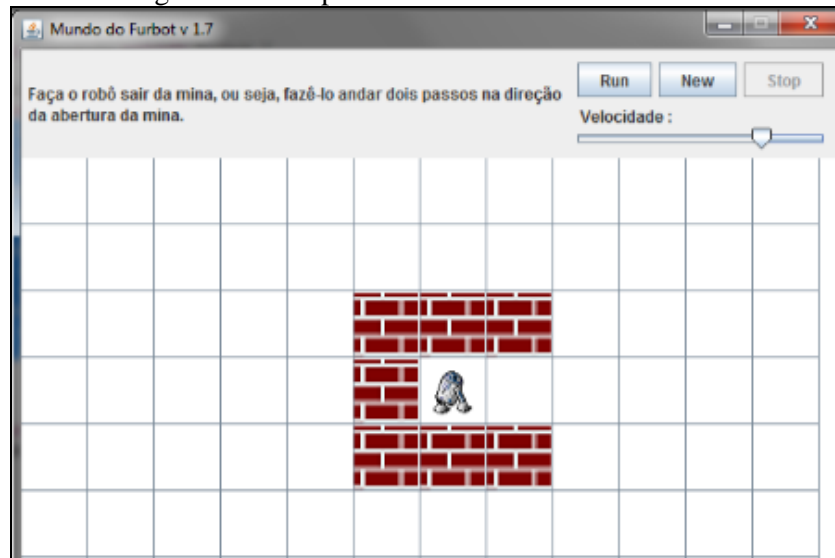
Nesta seção estão descritos os trabalhos correlatos, sendo eles o Furbot e o Robomind FURB, que envolvem simulação de robôs e o Blockly, um ambiente de programação visual.

2.4.1 FURBOT

O Furbot foi concebido para ser uma ferramenta para auxiliar no ensino de lógica de programação. Foi desenvolvido por um grupo de professores da Universidade Regional de Blumenau (FURB) com o propósito de disponibilizar um ambiente, com forte apelo na área de jogos, afim de criar uma atmosfera facilitadora no ensino da programação (VAHLDIK et al., 2008).

O Furbot se resume em um cenário matricial bidimensional, onde personagens, obstáculos e objetivos podem ser definidos. O objetivo de cada situação pode variar, dependendo da pré-configuração do mundo no qual o personagem principal irá atuar. A Figura 3 representa o cenário do robô que precisa sair da mina, sendo que a saída pode aparecer, aleatoriamente, na parte de cima, baixo, esquerda ou direita da posição inicial do personagem.

Figura 3 - Exemplo da mina na interface do Furbot



Fonte: Torrens (2014, p. 22).

Para resolver este problema, basta verificar a possibilidade de mover o personagem para cada um dos lados (linhas 1, 4, 7, 10) e, se possível, deslocar-se duas unidades na mesma direção. O Quadro 4 exemplifica a implementação deste código.

Quadro 4 - Programa exemplo do Furbot

```

1  if (ehVazio(ACIMA)) {
2    andarAcima();
3    andarAcima();
4  } else if (ehVazio(ABAIXO)) {
5    andarAbaixo();
6    andarAbaixo();
7  } else if (ehVazio(DIREITA)) {
8    andarDireita();
9    andarDireita();
10 } else {
11   andarEsquerda();
12   andarEsquerda();
13 }

```

Fonte: Torrens (2014, p. 23).

O Furbot é uma biblioteca Java que pode ser anexada ao *classpath* de qualquer *Integrated Development Environment* (IDE) de desenvolvimento, como o Eclipse ou o NetBeans. Na elaboração de cenários é necessária uma atenção maior, pois os cenários são construídos através de arquivos *eXtensible Markup Language* (XML) sendo necessário que o usuário que desenvolve o cenário esteja bem familiarizado com a biblioteca (VAHLICK et al., 2008).

2.4.2 ROBOMIND FURB

O RoboMind é um ambiente de desenvolvimento onde o programador desenvolve algoritmos na linguagem Robo, nascida da antiga linguagem Logo. O personagem principal dos cenários RoboMind é um robô, que pode se locomover, olhar ao redor, pintar e mover

itens. Isso a torna uma ótima ferramenta para familiarizar iniciantes aos conceitos básicos da programação (ROBOMIND ACADEMY, 2014).

Já o RoboMind FURB é uma adaptação do original. Ele disponibiliza um *plugin* para que os usuários possam testar os programas em robôs físicos montados com o *kit* Mindstorms NXT da Lego. O usuário pode escolher um robô suportado pelo ambiente e então programá-lo utilizando os comandos aceitos por cada modelo específico (BENITTI et al., 2010).

Na Figura 4 é apresentada a interface do RoboMind FURB. Ela é dividida em área de código à esquerda, cenário simulado à direita e uma área de mensagens abaixo. Ao ser pressionado o botão *play* (abaixo da área de código), os programas são compilados e executados no simulador. É possível, além de iniciar a execução, interrompê-la, bem como avançar comando por comando para analisar possíveis problemas com o robô.

Figura 4 - Ambiente RoboMind FURB



2.4.3 BLOCKLY

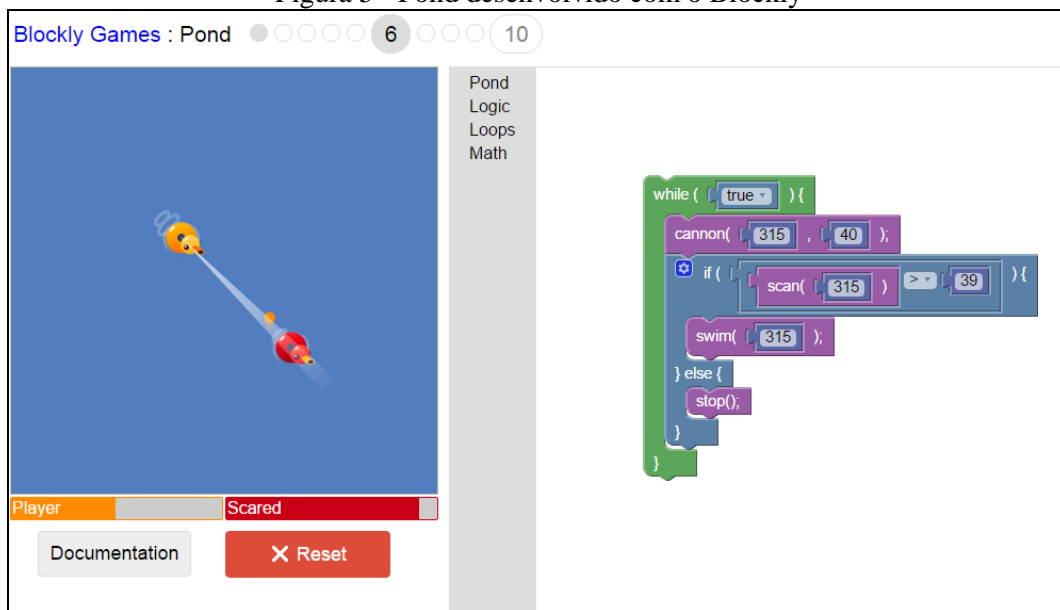
O Blockly é uma iniciativa da Google no desenvolvimento de um ambiente de programação visual. É um ambiente de programação baseado no encaixe de peças que formam os programas que são convertidos para JavaScript, Python e PHP, dentre outras linguagens. O Blockly pode ser usado, gratuitamente, para gerar qualquer linguagem que um desenvolvedor queira (GOOGLE DEVELOPERS, 2015).

Ele foi criado no intuito de ajudar programadores iniciantes a se adequarem com as regras de construção de programas e também ajuda a fazer com que os desenvolvedores

foquem apenas na lógica. Até mesmo alguém não familiarizado à programação pode utilizar o Blockly sem ter grandes problemas (GOOGLE DEVELOPERS, 2015).

A Figura 5 apresenta um dos jogos de lógica criado para mostrar o funcionamento do Blockly, o Pond. O objetivo deste jogo é acertar tiros no pato vermelho através do bloco cannon. Para tanto, deve-se informar o ângulo e a força usada ao disparar o canhão. Neste exemplo, o pato vermelho se desloca sempre que é acertado, forçando o uso de um *loop* (*while*) e dos comandos *swim* e *stop* (locomover e parar) para acertar os disparos.

Figura 5 - Pond desenvolvido com o Blockly



A equipe do Google Developers (2015) ainda cita que vem trabalhando para expandir o Blockly desde 2011, levando-o a:

- rodar em qualquer lugar: para poder incorporar o uso do Blockly em praticamente qualquer IDE ou dispositivo (como *tablets*, por exemplo);
- possuir mais blocos: disponibilizar uma gama maior de comandos reconhecidos pelo ambiente;
- melhorar a interface: seleção de múltiplos blocos, visualização de blocos removidos, dentre outros.

3 DESENVOLVIMENTO

Este capítulo divide-se em cinco partes, sendo elas:

- a) os requisitos principais do simulador;
- b) a especificação da linguagem intermediária e os comandos correspondentes na linguagem Robotoy;
- c) a especificação do simulador;
- d) a implementação, as ferramentas utilizadas e a operacionalidade da aplicação;
- e) os resultados obtidos no simulador.

3.1 REQUISITOS

O simulador deve:

- a) possuir uma interface para a elaboração de programas na linguagem Robotoy (Requisito Funcional - RF);
- b) possuir uma interface para criar e editar cenários 2D a fim de serem a base das simulações (RF);
- c) traduzir os programas Robotoy em um *script* que possa ser interpretado pelo simulador 2D (RF);
- d) executar os programas traduzidos no simulador 2D (RF);
- e) ser implementado na linguagem Java (Requisito Não Funcional - RNF);
- f) ser desenvolvido na IDE Eclipse (RNF);
- g) utilizar a biblioteca OpenGL para Java (JOGL) para modelar os cenários 2D (RNF);
- h) utilizar a linguagem Robotoy como linguagem de alto nível para programar as ações dos robôs que serão simuladas (RNF).

3.2 ESPECIFICAÇÃO DA LINGUAGEM INTERMEDIÁRIA

A ferramenta de programação da linguagem Robotoy, como proposta por Torrens (2014), gera código objeto em Java. Os programas gerados são transferidos aos robôs Lego Mindstorms NXT, os quais executam os comandos.

Entretanto, para que o simulador possa compreender os comandos a serem executados, foi desenvolvida uma linguagem intermediária em formato de *script*. Dessa forma, o simulador age como um interpretador, executando as ações nos cenários previamente desenvolvidos. Sendo assim, cada estrutura sintática da linguagem Robotoy é traduzida para essa linguagem intermediária, cuja definição encontra-se no Apêndice A.

A seguir estão relacionados os comandos da linguagem Robotoy, incluindo um exemplo, o comando correspondente na linguagem intermediária e uma descrição da ação executada pelo simulador / interpretador ou detalhamento do processo de tradução de Robotoy para a linguagem intermediária, quando for o caso. No Quadro 5 são apresentados os comandos executados pelo robô, seguido do Quadro 6 onde são listados os comandos para declaração / atribuição de valores a variáveis. Por fim, são mostrados os comandos para controle de fluxo e declaração / chamada de rotinas no Quadro 7.

Quadro 5 - Comandos a serem executados pelo robô

andar para frente <quantidade>	
<p>Exemplos:</p> <p>– em Robotoy andar para frente 4</p>	<p>– na linguagem intermediária COMANDO_ROBO;EM_FRENTE;4</p>
<p>Descrição: Move o robô para frente uma <quantidade> de vezes. Caso não seja informada a <quantidade>, o robô se move apenas uma unidade. A direção frente é o lado do cenário 2D para o qual o robô se encontra virado no momento da execução.</p>	
andar para trás <quantidade>	
<p>Exemplos:</p> <p>– em Robotoy andar para trás</p>	<p>– na linguagem intermediária COMANDO_ROBO;RECUAR;1</p>
<p>Descrição: Move o robô para trás uma <quantidade> de vezes. Caso não seja informada a <quantidade>, o robô se move apenas uma unidade. A direção trás é o lado contrário no cenário 2D para o qual o robô se encontra virado no momento da execução.</p>	
virar para a esquerda <quantidade>	
<p>Exemplos:</p> <p>– em Robotoy virar para a esquerda 1</p>	<p>– na linguagem intermediária COMANDO_ROBO;VIRAR_ESQUERDA;1</p>
<p>Descrição: Gira o robô para a esquerda uma <quantidade> de vezes. Caso não seja informada a <quantidade>, o robô se vira apenas uma vez. Uma <quantidade> equivale a 90° de rotação, fazendo com que o robô sempre esteja completamente virado para cima, para baixo, para a esquerda ou para a direita no cenário 2D.</p>	
virar para a direita <quantidade>	
<p>Exemplos:</p> <p>– em Robotoy virar para a direita</p>	<p>– na linguagem intermediária COMANDO_ROBO;VIRAR_DIREITA;1</p>
<p>Descrição: Gira o robô para a direita uma <quantidade> de vezes. Caso não seja informada a <quantidade>, o robô se vira apenas uma vez. Uma <quantidade> equivale a 90° de rotação, fazendo com que o robô sempre esteja completamente virado para cima, para baixo, para a esquerda ou para a direita no cenário 2D.</p>	

virar motor multiuso para a esquerda <quantidade>

Exemplos:

- | | |
|---|---|
| <p>– em Robotoy
virar motor multiuso para a esquerda 3</p> | <p>– na linguagem intermediária
COMANDO_ROBO;VIRAR_CABECA_ESQUERDA;3</p> |
|---|---|

Descrição:

Gira apenas a cabeça do robô simulado para a esquerda uma <quantidade> de vezes. Caso não seja informada a <quantidade>, o robô a vira apenas uma vez. Uma <quantidade> equivale a 90° de rotação, fazendo com que a cabeça do robô sempre esteja virada para cima, para baixo, para a esquerda ou para a direita no cenário 2D.

virar motor multiuso para a direita <quantidade>

Exemplos:

- | | |
|--|--|
| <p>– em Robotoy
virar motor multiuso para a direita 2</p> | <p>– na linguagem intermediária
COMANDO_ROBO;VIRAR_CABECA_DIREITA;2</p> |
|--|--|

Descrição:

Gira apenas a cabeça do robô simulado para a direita uma <quantidade> de vezes. Caso não seja informada a <quantidade>, o robô a vira apenas uma vez. Uma <quantidade> equivale a 90° de rotação, fazendo com que a cabeça do robô sempre esteja virada para cima, para baixo, para a esquerda ou para a direita no cenário 2D.

<negação> tem obstáculo

Exemplos:

- | | |
|---|--|
| <p>– em Robotoy
...
se tem obstáculo
 escrever "Obstáculo"
senão
 andar para frente 1
fim do se

enquanto não tem obstáculo
 andar para frente 1
fim do enquanto
escrever "Obstáculo"</p> | <p>– na linguagem intermediária
DECLARAR_VARIAVEL;tem_obstaculo;false
...
TEM_OBSTACULO
IF;tem_obstaculo
ESCREVER;("Obstáculo")
ELSE
COMANDO_ROBO;EM_FRENTE;1
ENDIF

TEM_OBSTACULO
WHILE;! tem_obstaculo
COMANDO_ROBO;EM_FRENTE;1
TEM_OBSTACULO
ENDWHILE
ESCREVER;("Obstáculo")</p> |
|---|--|

Descrição:

O comando TEM_OBSTACULO verifica se há um obstáculo na posição à frente, em relação à cabeça do robô no cenário 2D, e coloca a informação na variável tem_obstaculo, pré-declarada em qualquer código intermediário. A negação é opcional, permitindo verificar se não tem obstáculo. Este comando está diretamente ligado aos comandos de controle de fluxo. Caso seja usado na expressão de um comando <se> da Robotoy, antes do comando IF correspondente ao mesmo no código intermediário, é incluído o comando TEM_OBSTACULO e a variável tem_obstaculo é utilizada na expressão do comando IF. O mesmo ocorre com o comando <enquanto>. Entretanto o comando TEM_OBSTACULO é inserido novamente antes do ENDWHILE.

<code>cor identificada</code>	
Exemplos:	
– em Robotoy <code>cor c <- cor identificada</code>	– na linguagem intermediária <code>DECLARAR_VARIAVEL;cor_identificada;'BRANCO'</code> <code>...</code> <code>COR_IDENTIFICADA</code> <code>DECLARAR_VARIAVEL;c;cor_identificada</code>
Descrição: O comando <code>COR_IDENTIFICADA</code> verifica a cor da posição onde o robô se encontra no cenário 2D e coloca a informação na variável <code>cor_identificada</code> , pré-declarada em qualquer código intermediário. As cores suportadas são amarelo, azul, branco, preto, verde e vermelho. Este comando pode ser usado em comandos de controle de fluxo ou na declaração de variáveis. Caso seja usado em comandos de controle de fluxo, a tradução de Robotoy para a linguagem intermediária é a mesma do comando anterior. Caso seja usado na declaração de uma variável em Robotoy, antes da declaração da variável correspondente no código intermediário, é incluído o comando <code>COR_IDENTIFICADA</code> e a variável <code>cor_identificada</code> é utilizada para atribuir um valor à variável declarada.	
<code>emitir som</code>	
Exemplos:	
– em Robotoy <code>emitir som</code>	– na linguagem intermediária <code>COMANDO_ROBO;EMITIR_SOM;(0)</code>
Descrição: Permite que o robô emita o <i>beep</i> padrão do computador que está executando a simulação.	
<code>escrever <texto></code>	
Exemplos:	
– em Robotoy <code>escrever "Olá Mundo"</code>	– na linguagem intermediária <code>ESCREVER;("Olá Mundo")</code>
Descrição: Permite que o robô escreva mensagens para o usuário.	

Quadro 6 - Comandos para declaração e atribuição de valores a variáveis

<code>cor <nome> <- <valor></code>	
Exemplos:	
– em Robotoy <code>cor c <- azul</code>	– na linguagem intermediária <code>DECLARAR_VARIAVEL;c;'AZUL'</code>
Descrição: Cria uma variável do tipo <code>cor</code> , identificada por <code><nome></code> , e atribui o <code><valor></code> a mesma.	
<code>número <nome> <- <valor></code>	
Exemplos:	
– em Robotoy <code>número n1 <- 5</code> <code>número n2 <- 3 * n1</code>	– na linguagem intermediária <code>DECLARAR_VARIAVEL;n1;(5)</code> <code>DECLARAR_VARIAVEL;n2;(3 * n1)</code>
Descrição: Cria uma variável do tipo <code>número</code> , identificada por <code><nome></code> , e atribui o <code><valor></code> a mesma.	
<code>texto <nome> <- <valor></code>	
Exemplos:	
– em Robotoy <code>texto t <- "Olá""Tudo bem?"</code>	– na linguagem intermediária <code>DECLARAR_VARIAVEL;t;"Olá" + "Tudo bem?"</code>
Descrição: Cria uma variável do tipo <code>texto</code> , identificada por <code><nome></code> , e atribui o <code><valor></code> a mesma.	

<code><nome> <- <valor></code>	
Exemplos:	
<ul style="list-style-type: none"> – em Robotoy <pre>texto t <- "Olá" t <- t . "Tudo bem?"</pre>	<ul style="list-style-type: none"> – na linguagem intermediária <pre>DECLARAR_VARIAVEL;t;"Olá" ATRIBUIR_VALOR_VARIAVEL;t;t + "Tudo bem?"</pre>
Descrição:	
Atribui o <valor> à variável identificada por <nome>.	

O <valor> citado no Quadro 6 representa os valores atribuídos às variáveis, podendo ser uma cor, uma expressão numérica ou uma expressão literal, conforme a especificação da linguagem Robotoy (TORRENS, 2014, p. 65). O interpretador não faz nenhuma validação de tipo ao criar ou atribuir valores para as variáveis. Quando uma expressão estiver sendo avaliada, o interpretador utiliza o valor das constantes e das variáveis e, caso os tipos não sejam compatíveis, retorna erro de inconsistência na resolução da expressão.

Quadro 7 - Comandos de controle de fluxo e declaração / chamada de rotinas

<pre>se <condição> <lista de comandos> senão <lista de comandos> fim do se</pre>	
Exemplos:	
<ul style="list-style-type: none"> – em Robotoy <pre>número horas <- 10 se horas > 18 escrever "Boa noite" senão escrever "Bom dia" fim do se</pre>	<ul style="list-style-type: none"> – na linguagem intermediária <pre>DECLARAR_VARIAVEL;horas;10 IF;(horas > 18) ESCREVER;("Boa noite") ELSE ESCREVER;("Bom dia") ENDIF</pre>
Descrição:	
O comando IF depende de uma <condição>. Se esta for verdadeira, o interpretador executa a <lista de comandos> associada ao IF. Do contrário, a <lista de comandos> do ELSE é executada. Atenta-se ao fato de que o ELSE é opcional.	
<pre>enquanto <condição> <lista de comandos> fim do enquanto</pre>	
Exemplos:	
<ul style="list-style-type: none"> – em Robotoy <pre>número n <- 1 texto t <- "" enquanto n <= 10 t <- "3 x" . n . "=" . (n * 3) escrever t n <- n + 1 fim do enquanto</pre>	<ul style="list-style-type: none"> – na linguagem intermediária <pre>DECLARAR_VARIAVEL;n;1 DECLARAR_VARIAVEL;t;" " WHILE;n <= 10 ATRIBUIR_VALOR_VARIAVEL;t; "3 x" + n + "=" + n * 3 ESCREVER;(t) ATRIBUIR_VALOR_VARIAVEL;n;n + 1 ENDWHILE</pre>
Descrição:	
O comando WHILE opera de forma semelhante ao comando IF, pois precisa avaliar uma <condição>. Entretanto, o WHILE executará a <lista de comandos> até que a <condição> seja falsa.	

```
rotina <nome>
  <lista de comandos>
fim da rotina
```

Exemplos:– **em Robotoy**

```
número v <- 0
virar motor multiuso para a
esquerda 1
enquanto tem obstáculo
  addValor
fim do enquanto
texto t <- "Valor: " . v
escrever t

rotina addValor
  v <- v + 10
  virar motor multiuso para a
  direita 1
  se não tem obstáculo
    andar para frente
    virar motor multiuso para a
    esquerda 1
  fim do se
fim da rotina
```

– **na linguagem intermediária**

```
ROTINA;addValor
ATRIBUIR_VALOR_VARIAVEL;v;v + 10
COMANDO_ROBO;VIRAR_CABECA_DIREITA;( 1 )
TEM_OBSTACULO
IF;! tem_obstaculo
COMANDO_ROBO;EM_FRENTE;( 1 )
COMANDO_ROBO;VIRAR_CABECA_ESQUERDA;( 1 )
ENDIF
ENDROTINA

DECLARAR_VARIAVEL;tem_obstaculo;false
DECLARAR_VARIAVEL;v;0
COMANDO_ROBO;VIRAR_CABECA_ESQUERDA;( 1 )
TEM_OBSTACULO
WHILE;tem_obstaculo
CHAMADA_ROTINA;addValor
TEM_OBSTACULO
ENDWHILE
DECLARAR_VARIAVEL;t;"Valor: " + v
ESCREVER;( t )
```

Descrição:

As rotinas, também conhecidas por métodos, são utilizadas para encapsular trechos de códigos que serão utilizados várias vezes ao longo de um código, melhorando assim a legibilidade do mesmo. As rotinas são identificadas por um <nome> e são compostas por uma <lista de comandos>. Diferentemente da Robotoy, todas as rotinas existentes no código intermediário devem aparecer no começo, pois o interpretador reconhece as rotinas existentes em tempo de execução. A utilização de uma rotina é feita através do seu <nome>.

A <condição> citada no Quadro 7 pode ser uma comparação entre número, cores, textos ou a verificação de obstáculos (TORRENS, 2014, p. 36).

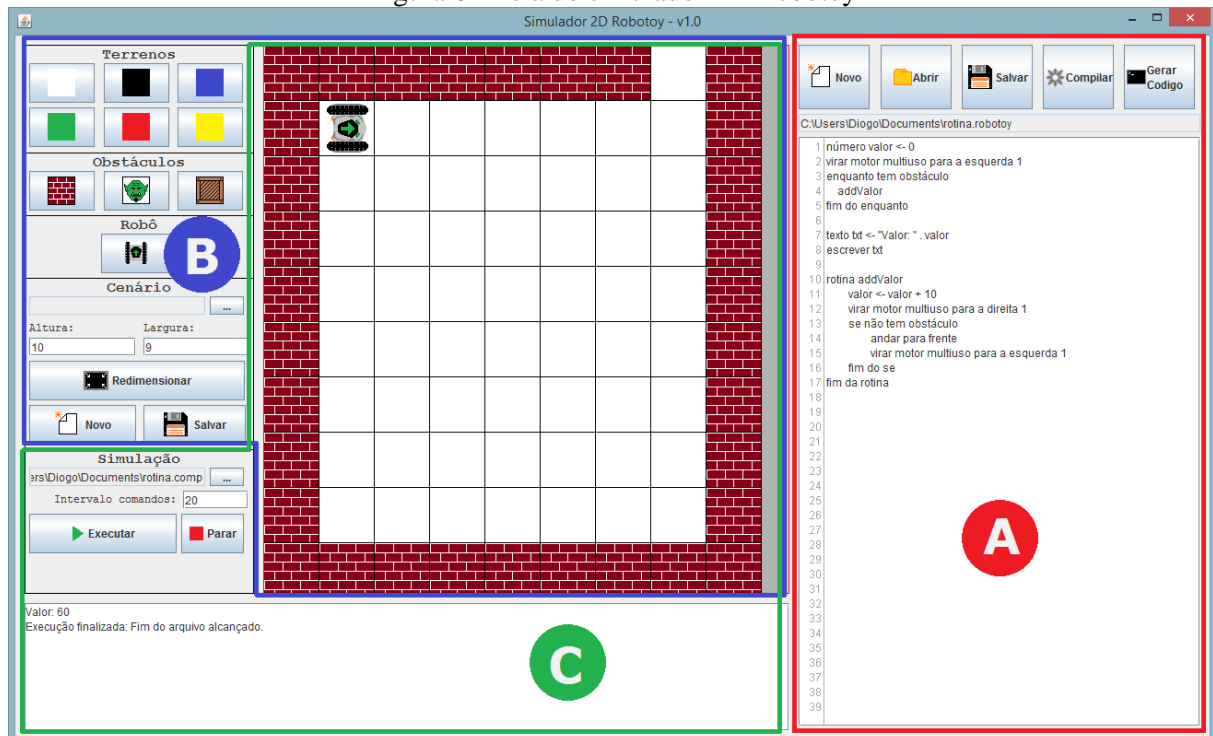
Observa-se que neste trabalho alguns comandos da linguagem Robotoy a serem executados pelo robô foram simplificados, enquanto outros comandos, como girar roda, parar de girar, parar de andar e parar de virar foram suprimidos. Na linguagem Robotoy é permitido usar o comando girar roda <lado> para <direção> a fim de movimentar apenas uma roda do robô por vez. Associado a esse comando, tem-se o comando parar de girar. Esses comandos foram suprimidos porque o robô simulado não possui rodas para efetuar giros independentes, ele possui apenas a direção a qual está virado. Já os comandos andar e virar, apresentados no Quadro 5, em Robotoy podem não ser seguidos de uma quantidade de vezes a executar. Isso implica na execução contínua do comando em questão (andar ou virar) até que um comando parar de andar ou parar de virar (respectivamente) seja encontrado. Na linguagem intermediária, os comandos andar e virar são sempre seguidos de uma quantidade de vezes a executar porque o robô deve sempre saber quantas casas ele irá se locomover, para que o simulador possa representar esta movimentação

e o robô não saia do cenário virtual. Como não é possível andar ou virar constantemente em uma direção, os comandos `parar de andar` e `parar de virar` não são necessários.

3.3 ESPECIFICAÇÃO DO SIMULADOR

A fim de explicar o simulador como um todo, é necessário que ele seja visto como uma composição de três módulos, visualizados na Figura 6: compilador (área destacada em vermelho), editor de cenários (área destacada em azul) e o controlador da simulação (área destacada em verde). O compilador é responsável por traduzir os programas escritos na linguagem Robotoy (arquivos com a extensão `.robotoy`) em *scripts* da linguagem intermediária (arquivos com a extensão `.comp`). Um cenário 2D (componente B) é compartilhado com o editor de cenários e o controlador da simulação, pois ambos fazem uso do mesmo. O editor de cenários apresenta uma barra de ferramentas com botões para edição de cenários e botões para carregar, redimensionar, criar e salvar os cenários (arquivos com a extensão `.cena`) para uso futuro. Por fim, o controlador da simulação é dedicado à simulação, onde é possível selecionar um arquivo de *script* (arquivo `.comp`), executar e parar a simulação. Esse módulo dispõe também de um componente que representa o *display* do robô (componente C).

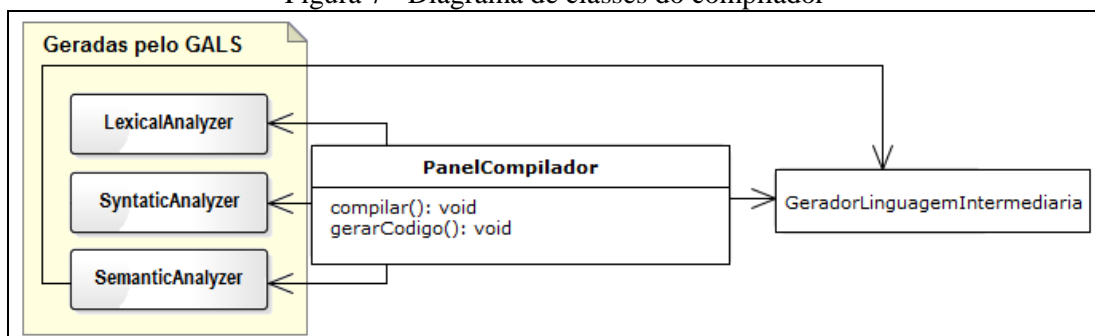
Figura 6 - Tela do simulador 2D Robotoy



As classes que compõem o compilador são apresentadas na Figura 7. As classes responsáveis pelas análises léxica e sintática da Robotoy, `LexicalAnalyzer` e

`SyntaticAnalyzer`, respectivamente, geradas pela ferramenta GALS, utilizada por Torrens (2014), foram mantidas. Já classe `SemanticAnalyzer` sofreu diversas alterações. Na Robotoy, essa classe gera linhas de código Java que são posteriormente adicionadas ao arquivo gerado. Entretanto, para efetuar a simulação de um programa Robotoy, a classe `SemanticAnalyzer` teve que ser modificada para gerar código na linguagem intermediária especificada. Para tal, todos os métodos foram modificados, incluindo os métodos que geram as expressões, os quais foram adaptados gerando código JavaScript. A classe `PanelCompilador` é um painel que controla os analisadores (através do método `compilar`) e é quem instancia a classe `GeradorLinguagemIntermediaria`, a qual possui uma lista de comandos em *script*, traduzidos pelo analisador semântico. Já o método `gerarCodigo` dessa classe é o responsável por delegar ao `GeradorLinguagemIntermediaria` que o mesmo deve gravar a lista de comandos em um arquivo `.comp`, o qual pode ser executado no simulador.

Figura 7 - Diagrama de classes do compilador

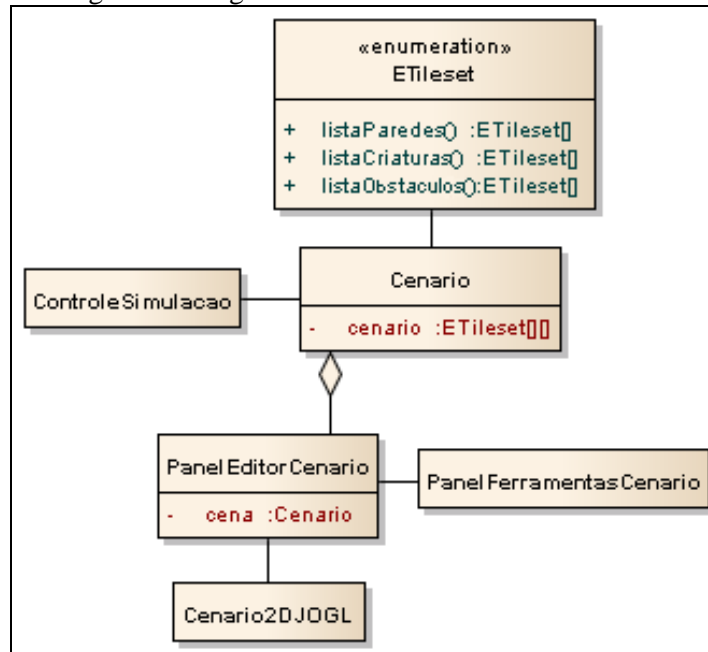


Vale ressaltar que a remoção dos comandos `andar` e `virar` sem limitação (sem parâmetro) fizeram com que uma nova ação semântica (a de número 53, definida pelo método `addQuantidadeVezesPadrao()`) tivesse que ser incluída no `SemanticAnalyzer`. Esse método é chamado sempre que um programa possui algum comando `andar` ou `virar`, para garantir que no código intermediário gerado o parâmetro seja maior ou igual a um. A inclusão de tal regra semântica acarretou na inclusão de uma chamada para esse método na classe `SyntaticAnalyzer`.

O editor de cenário, como já descrito, é composto de uma barra de ferramentas e um cenário 2D. Um cenário é composto por uma matriz de dimensões (x, y) , onde cada célula dessa matriz representa um espaço utilizado para desenhar as texturas e a cena. No diagrama de classes representado na Figura 8 pode-se ver a relação entre as classes que compõem o editor de cenários. O `PanelFerramentasCenario` é a barra de ferramentas, onde se seleciona o que será alterado no `Cenario`, que por sua vez é composto por uma matriz de `ETileset`, guardando as informações de cada uma das células bem como as dimensões da cena. As

modificações efetuadas no cenário são feitas no painel `Cenario2DJOGL`, que ao receber os cliques do *mouse*, transmite as alterações (como a alteração de uma célula ou o posicionamento do robô) ao objeto de `Cenario`. Este se encontra instanciado e em edição no `PanelEditorCenario` e, por fim, o `Cenario2DJOGL` deve redesenhar a cena (baseando-se na matriz de `ETileset` do `Cenario`) para representar a cena atual após quaisquer alterações efetuadas.

Figura 8 - Diagrama de classe do editor de cenários

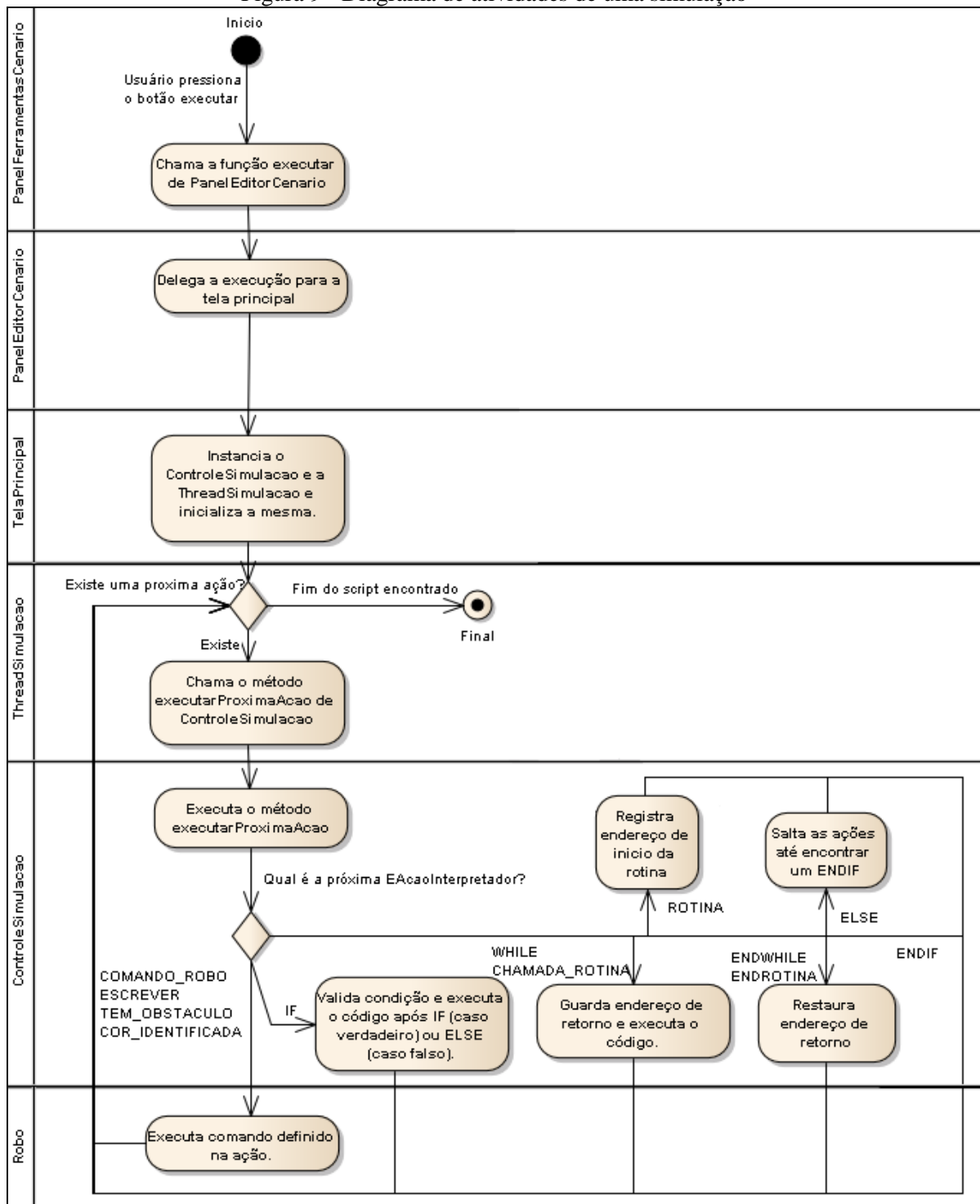


No diagrama representado pela Figura 8, a enumeração `ETileset` define o que cada célula do cenário representa. Cada valor dessa enumeração possui as seguintes características: código, cor, nome, caractere que a representa e se a célula é transponível ou não, sendo, nesse caso, considerada um obstáculo para o robô. Assim, as células podem ser divididas em transponível de cor base (terrenos) ou obstáculo, o qual pode ser subdividido em paredes, criaturas e obstáculos. Essas subdivisões são definidas respectivamente pelos métodos estáticos `listaParedes`, `listaCriaturas` e `listaObstaculos`, pertencentes à classe `ETileset`.

A simulação tem início ao ser pressionado o botão `Executar` (do controle da simulação – Figura 6). Como pode ser visualizada no diagrama de atividades representado pela Figura 9, a função de executar uma simulação é delegada através da tela principal do sistema, a qual implementa o método responsável por instanciar a classe `ControleSimulacao`. Essa é responsável por ler o *script* e interpretar suas ações. Também é inicializada a classe que possui a *thread* onde a simulação é executada, a qual opera

paralelamente ao restante do sistema, tornando possível a edição dos cenários durante a simulação.

Figura 9 - Diagrama de atividades de uma simulação



A `ThreadSimulacao` executa o método `executarProximaAcao` até que não haja mais ações (linhas) no *script*. Cada ação verificada pelo `ControleSimulacao` (na tomada de decisão, após a atividade “Executa o método `executarProximaAcao`”) deve existir na enumeração `EacaoInterpretador` e o `ControleSimulacao` então, baseado nessa ação, toma

as decisões necessárias. Caso seja encontrado `COMANDO_ROBO`, `ESCREVER`, `COR_IDENTIFICADA` ou `TEM_OBSTACULO`, o controle faz com que a classe `Robo` execute a ação correspondente. Do contrário, o próprio interpretador é quem executa as ações.

3.4 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas, a implementação dos métodos responsáveis por interpretar as ações descritas nos *scripts* e por identificar os espaços selecionados do cenário quando ocorrem alterações no mesmo, assim como a operacionalidade do simulador.

3.4.1 Técnicas e ferramentas utilizadas

Para desenvolver o simulador foi utilizada a IDE Eclipse em sua versão Luna. Não foram utilizados *plugins* extras para desenvolver a interface ou quaisquer outras partes. Foi utilizada a classe `ScriptEngine` do Java para criar uma *engine* de *scripts* para que o simulador possa resolver expressões, as quais são escritas em JavaScript. Já na implementação do cenário 2D foi utilizada a biblioteca JOGL, que é a extensão Java da OpenGL. Foi utilizada apenas a visualização 2D da biblioteca em conjunto com as texturas. As texturas foram criadas e editadas no editor de imagens *Microsoft Paint*, presente nos sistemas operacionais da *Microsoft*. Apenas duas texturas presentes no simulador foram baixadas da internet (Anexo A).

3.4.2 Método responsável por interpretar as ações

Conforme explanado na seção 3.3, os *scripts* são interpretados pela classe `ControleSimulacao`, mais precisamente pelo método `executaProximaAcao` que se encontra no Quadro 8.

Quadro 8 - Método `executarProximaAcao` da classe `ControleSimulacao`

```

1  StringBuilder linha = new StringBuilder();
2  Retorno ret = new Retorno();
3
4  ret = proximaLinha(linha);
5  if (!ret.ok()) return ret;
6
7  String line[] = linha.toString().split(";");
8  EAcaoInterpretador todo = EAcaoInterpretador.valueOf(line[0]);
9
10 ResultadoExpressao resultado = new ResultadoExpressao(new Object());
11
12 switch (todo) {
13 case DECLARAR_VARIAVEL:
14 case ATRIBUIR_VALOR_VARIAVEL:
15     ret = executarExpressao(line[2], resultado);
16     mapaVariaveis.put(line[1], resultado.getResultado());

```

```

17     break;
18 case COMANDO_ROBO:
19     ret = executarExpressao(line[2], resultado);
20     if (!ret.ok()) return ret;
21     ret =
22     robo.executarAcao(ESComandoRobo.valueOf(line[1]),
23                     Integer.parseInt(resultado.getResultado().toString()));
24     break;
25 case TEM_OBSTACULO:
26     resultado.setResultado(new Obstaculo(false));
27     ret =
28     robo.executarAcao(ESComandoRobo.DETECTAR_OBSTACULO, resultado.getResultado());
29     mapaVariaveis.put("tem_obstaculo",
30                     ((Obstaculo) resultado.getResultado()).isTemObstaculo());
31     break;
32 case COR_IDENTIFICADA:
33     resultado.setResultado(new Cor(Color.white));
34     ret = robo.executarAcao(ESComandoRobo.DETECTAR_COR, resultado.getResultado());
35     if (resultado.getResultado() != null)
36         mapaVariaveis.put("cor_identificada", resultado.getResultado().toString());
37     break;
38 case IF:
39     ret = executarExpressao(line[1], resultado);
40     if (!(Boolean) resultado.getResultado().booleanValue())
41         while (!(line[0].equals("ELSE") || line[0].equals("ENDIF"))) {
42             proximaLinha(linha);
43             line = linha.toString().split(";");
44             if (line[0].equals("IF")) pulaAteEndif(line);
45         }
46     break;
47 case ELSE:
48     pulaAteEndif(line);
49     break;
50 case ENDIF: // Nada acontece. Feijoadá.
51     break;
52 case WHILE:
53     ret = executarExpressao(line[1], resultado);
54     if ((Boolean) resultado.getResultado().booleanValue()) {
55         try {
56             offsetDosRetornos.push(arquivo.getOffset() - linha.length() - 2);
57         } catch (IOException e) {
58             e.printStackTrace();
59             return new
60             Retorno().erro("Ocorreu um erro ao empilhar o offset do WHILE.");
61         }
62     } else {
63         pulaAteEndWhile(line);
64     }
65     break;
66 case ENDWHILE:
67 case ENDROTINA:
68     try {
69         arquivo.seek(offsetDosRetornos.pop());
70     } catch (IOException e) {
71         e.printStackTrace();
72         return new Retorno().erro("Não foi possível restaurar o offset do WHILE.");
73     }
74     break;
75 case CHAMADA_ROTINA:
76     try {
77         offsetDosRetornos.push(arquivo.getOffset());
78         arquivo.seek(mapaOffsetDasRotinas.get(line[1]));
79     } catch (IOException e) {
80         e.printStackTrace();
81         return new
82         Retorno().erro("Ocorreu um erro ao ajustar o offset do arquivo.");
83     }
84     break;

```

```

85 case ROTINA:
86     try {
87         mapaOffsetDasRotinas.put(line[1], arquivo.getOffset());
88         while (!(line[0].equals("ENDROTINA"))) {
89             proximaLinha(linha);
90             line = linha.toString().split(";");
91         }
92     } catch (IOException e) {
93         e.printStackTrace();
94         return new
95             Retorno().erro("Ocorreu um erro ao tentar obter o offset da rotina.");
96     }
97     break;
98 case ESCREVER:
99     ret = executarExpressao(line[1], resultado);
100    if (!ret.ok()) return ret;
101    ret =
102        robo.executarAcao(EMandoRobo.ESCREVER,
103                        resultado.getResultado().toString());
104    break;
105 default:
106    ret.erro("Comando " + todo + " inexistente.");
107    break;
108 }
109 return ret;

```

Inicialmente (linhas 1 e 2), são instanciados um `StringBuilder` que conterá a próxima linha do *script* a ser analisada e um objeto da classe `Retorno` que conterá o valor que o método `executarProximaAcao` irá retornar. Então é utilizado o método `proximaLinha` (linha 4) para buscar do *script* a próxima linha a ser executada. Os *tokens* são separados uns dos outros pelo caracter ponto e vírgula (;) presente na linha e seus valores distribuídos no vetor `line`, como pode ser observado na linha 7. Em seguida, o interpretador tenta obter uma instância de `EacaoInterpretador` utilizando o primeiro valor do vetor (linha 8).

Na linha 12 inicia o `switch` que decide qual ação tomar dependendo da `EacaoInterpretador` instanciada. Para o interpretador, tanto a declaração de variável (linha 13) quanto a atribuição de valor para uma variável (linha 14) devem ser tratadas da mesma forma: colocar no `mapaVariaveis`, um `HashMap` contendo todas as variáveis que estão sendo utilizadas no *script*, o par chave-valor, sendo a chave o nome da variável contido na segunda posição do vetor `line` e o valor calculado pelo método privado `executarExpressao` (descrito no Quadro 9) a partir do conteúdo da terceira posição do vetor `line`. Caso tenha ocorrido algum erro na avaliação da expressão (linha 15), o método `executarProximaAcao` retorna o erro detectado (linha 109).

Caso a ação seja um `COMANDO_ROBO` (linha 18), o interpretador designa que o `Robo` execute a ação definida pelo segundo valor do vetor `line` (linha 22), porém antes disso utiliza o método `executarExpressao` para validar a quantidade de vezes que a ação deve ser executada, definida pelo conteúdo da terceira posição do vetor.

Se a ação for `TEM_OBSTACULO` ou `COR_DENTIFICADA` (linhas 25 e 32, respectivamente), novamente o interpretador define que o Robo deve executar estas ações. Diferentemente do `COMANDO_ROBO`, estas ações retornam um valor que é atribuído às variáveis pré-declaradas `tem_obstaculo` (linha 29) e `cor_identificada` (linha 36), respectivamente.

No caso do interpretador encontrar um `IF` (linha 38), é executado o método `executarExpressao` considerando que o segundo valor do vetor `line` é a condição do comando `IF`. O resultado da avaliação da condição (linha 40) determina qual a próxima linha do *script* deve ser interpretada. Se a condição for verdadeira, a próxima linha a ser interpretada é a primeira linha da lista de comandos associada ao `IF`. Se a condição for falsa, o interpretador desconsidera as próximas linhas (linhas 41 a 44) até encontrar o `ELSE` ou `ENDIF` correspondente ao `IF` em questão, determinando que a próxima linha a ser interpretada é a primeira linha da lista de comandos associada ao `ELSE` ou a primeira linha após o `ENDIF`, conforme o caso. Portanto, quando o interpretador encontra um `ELSE` (linha 47) significa que ele estava executando as ações associadas a um `IF` cujo resultado da avaliação da condição era verdadeiro. Deve então desconsiderar todas as linhas subsequentes até encontrar o `ENDIF` correspondente, para o qual não é executada nenhuma ação quando encontrado (linha 50).

Quando o interpretador encontra um `WHILE`, executa nas linhas 53 e 54 a mesma validação de condição que o `IF`. Quando o resultado da avaliação da condição é falso, todas as linhas são ignoradas até que um `ENDWHILE` seja encontrado (linha 63) e quando é verdadeiro, o interpretador determina que a próxima linha a ser interpretada é a primeira linha da lista de comandos associada ao `WHILE`. Entretanto, diferente do `IF`, o *offset* da linha do `WHILE` (a posição do arquivo onde o `WHILE` começa) é armazenado na variável `offsetDosRetornos`. Esta é uma pilha que guarda o ponto que deve ser restaurado ao encontrar as ações `ENDWHILE` ou `ENROTINA` (linhas 66 e 67), conforme o caso.

A ação `ROTINA` é a declaração de uma rotina que será executada ao ser encontrada a `CHAMADA_ROTINA` correspondente. Portanto, o interpretador adiciona o *offset* da primeira linha da rotina no `HashMap` nomeado `mapaOffsetDasRotinas`, que contém o nome de cada rotina declarada como chave e o *offset* como valor (linha 87). Feito isso, o interpretador desconsidera todas as instruções (linhas 88 a 90) até encontrar `ENDROTINA`.

A ação `CHAMADA_ROTINA` (linha 75) é a responsável por utilizar os *offsets* do `mapaOffsetDasRotinas` (linha 78) e fazer com que sejam executadas as instruções da `ROTINA` correspondente. Entretanto, antes de restaurar o ponto do arquivo onde a rotina se

encontra, o interpretador salva o *offset* da ação seguinte a `CHAMADA_ROTINA` (linha 77) para que a execução continue a partir desse ponto ao encontrar o `ENDROTINA`.

Concluindo as ações interpretadas, tem-se a ação `ESCREVER` (linha 98), que faz com que o `Robo` escreva em seu *display* (linha 102) a mensagem calculada pelo método `executarExpressao` a partir do conteúdo da segunda posição do vetor `line` (linha 99).

Pode-se observar no Quadro 9 que o método `executarExpressao` faz uso da classe `ScriptEngine` (linha 1), que é instanciada como um motor JavaScript. A expressão é então quebrada em vários termos (linha 6) e então o interpretador tenta substituir quaisquer termos da expressão pelo valor presente no `mapaVariaveis`, fazendo com que o nome das variáveis utilizadas na expressão seja trocado pelo seu devido valor (linha 9). Feito isso, a expressão é avaliada (linha 11) e o resultado armazenado em uma instância de `ResultadoExpressao`, representada pela variável `resultado`. Por fim, o método retorna (linha 16) se a expressão foi concluída com êxito ou se houve falha ao executá-la.

Quadro 9 - Método `executarExpressao` da classe `ControleSimulacao`

```

1  ScriptEngine scriptEngine = new ScriptEngineManager().
2                                getEngineByName("javascript");
3
4  Bindings bind = scriptEngine.getBindings(ScriptContext.ENGINE_SCOPE);
5
6  String possiveisVariaveis[] = expressao.split(" ");
7  for (int i = 0; i < possiveisVariaveis.length; i++)
8      if (mapaVariaveis.containsKey(possiveisVariaveis[i]))
9          bind.put(possiveisVariaveis[i], mapaVariaveis.get(possiveisVariaveis[i]));
10     try {
11         resultado.setResultado(scriptEngine.eval(expressao));
12     } catch (ScriptException e) {
13         e.printStackTrace();
14         return new Retorno().erro("Ocorreu um erro na resolução da formula.");
15     }
16     return new Retorno();

```

3.4.3 Método responsável por editar o cenário

Conforme especificado na seção 3.3, um cenário é composto por uma matriz de `ETileset` que se encontra na classe `Cenario`. Esta por sua vez é editada através da classe `Cenario2DJOGL` que interpreta os cliques do *mouse* e, baseado na posição do mesmo; no deslocamento da cena; e no *zoom*, aplica as alterações no cenário. Pode-se observar no Quadro 10 o método responsável por fazer a relação entre a posição do *mouse* e a posição da matriz do cenário, e por validar a inclusão da célula selecionada na matriz da cena ou adicionar a posição inicial do robô. Vale ressaltar que o cenário é desenhado abaixo do eixo *x* do sistema de referência. Tal decisão foi tomada para que a posição `0,0` da matriz fosse representada na visualização 2D nessas mesmas coordenadas, facilitando a aplicação geral do método de desenho.

Quadro 10 - Método `setTileAt` da classe `Cenario2DJOGL`

```

1  int x = (-deslocamentoViewportX + mouseX) / (int)(tamanhoCelulas / zoom),
2      y = (deslocamentoViewportY + mouseY) / (int)(tamanhoCelulas / zoom);
3
4  Cenario cena = panelEditorCenario.getCenarioEmEdicao();
5
6  if (panelEditorCenario.getTileSelecionado() != ETileset.TILE_COM_ROBO) {
7      if (!(x >= 0 && x < cena.getLargura() && y >= 0 && y < cena.getAltura()))
8          return;
9      if (y == cena.getxInicialRobo() && x == cena.getyInicialRobo())
10         if (!panelEditorCenario.getTileSelecionado().isTransponivel())
11             return;
12     cena.setTileAt(y, x, panelEditorCenario.getTileSelecionado());
13 } else {
14     Tile tile = new Tile(ETileset.defaultTileSet());
15     if (!cena.getTileAt(new int[] {y, x}, tile).ok())
16         return;
17     if (tile.getTile().isTransponivel())
18         cena.setPosicaoInicialRobo(new int[] {y, x});
19 }
20 panelEditorCenario.cenarioEditado();
21 glDrawable.display();

```

Inicialmente (linhas 1 e 2), é efetuada a conversão das posições x e y do *mouse* para as coordenadas da matriz do cenário. As variáveis `xMouse` e `yMouse` são a posição do *mouse* dentro da cena OpenGL apresentada ao usuário. Estes valores são somados com as variáveis `deslocamentoViewportX` negativa¹ e `deslocamentoViewportY`, respectivamente, e então divididos pelo tamanho das células e pelo `zoom`. O `zoom` é um valor do tipo `float` que representa a distância da câmera para com a tela de pintura. Na Figura 10 pode ser observado como são encontrados os valores x e y descritos anteriormente. No exemplo foi considerado que a *viewport* foi deslocada 75 unidades em x e que o *mouse* foi clicado na posição $x=20$ do editor de cenários. O tamanho das células utiliza o valor padrão de 60, enquanto foi considerado um pequeno *zoom* no cenário.

Figura 10 - Exemplo de conversão da posição do *mouse* para a posição da matriz do cenário

$$\begin{array}{l}
 x = \frac{-\text{deslocamentoViewportX} + \text{xMouse}}{\text{tamanhoCelulas} / \text{zoom}} \\
 x = \frac{-75 + 20}{60 / 0.8} \\
 x = \frac{-55}{75} \\
 x = -0.73 \\
 x = 0 \quad \checkmark
 \end{array}$$

Após a obtenção das coordenadas, o simulador busca a instância do `Cenario` que será editado (linha 4). Então efetua uma validação para saber se a célula (robô, terreno, obstáculo)

¹ Os deslocamentos da *viewport* são alterados quando o usuário movimenta a tela de pintura OpenGL para os lados. Ao efetuar a conversão de ponto 2D para posição da matriz, é necessário levar em consideração este deslocamento. No caso do x o valor é negativo, pois ao arrastar a *viewport* para a direita, por exemplo, o ponto 2D onde o mouse clica fica a esquerda de onde originalmente o cenário estava sendo desenhado.

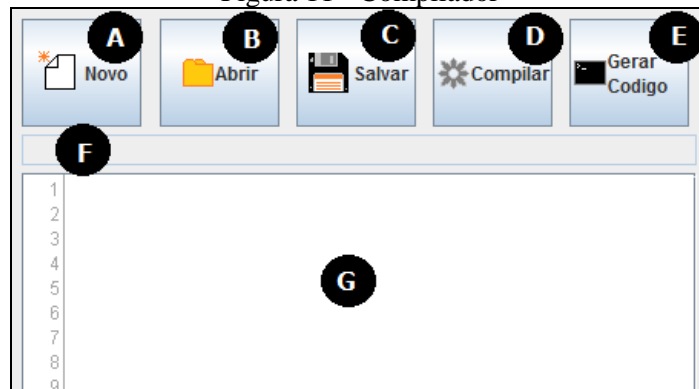
a ser incluída no cenário não é a que representa o robô (linha 6). Caso seja o `TILE_COM_ROBO`, o sistema valida a célula nas coordenadas definidas (linha 15) e, caso não ocorra nenhum erro, verifica se a célula selecionada é transponível (linha 17), definindo a posição inicial do robô no cenário com as coordenadas da seleção (linha 18). Caso a célula a ser incluída não seja o robô, o simulador valida a posição calculada (linha 7) e verifica se a posição inicial do robô é a mesma coordenada selecionada (linha 9). Se assim for, é verificado se a célula a ser incluída é transponível (linha 10). Se não for, o método `setTileAt` é abortado. Entretanto, se a posição inicial do robô for diferente da selecionada ou a célula for transponível, a aplicação altera a matriz do `Cenario` incluindo a nova célula nas coordenadas x e y calculadas (linha 12). Observa-se que os valores passados como parâmetro seguem a ordem y, x . Isso se dá, pois a posição x do cenário OpenGL representa a coluna da matriz e a posição y sua linha.

3.4.4 Operacionalidade da implementação

O simulador, conforme apresentado na Figura 6 da seção 3.3, foi dividido em três módulos. Em seguida é explanado o funcionamento de cada um desses módulos. Primeiro é abordado o compilador, seguido do editor de cenários e, por fim, é apresentada a parte que executa as simulações.

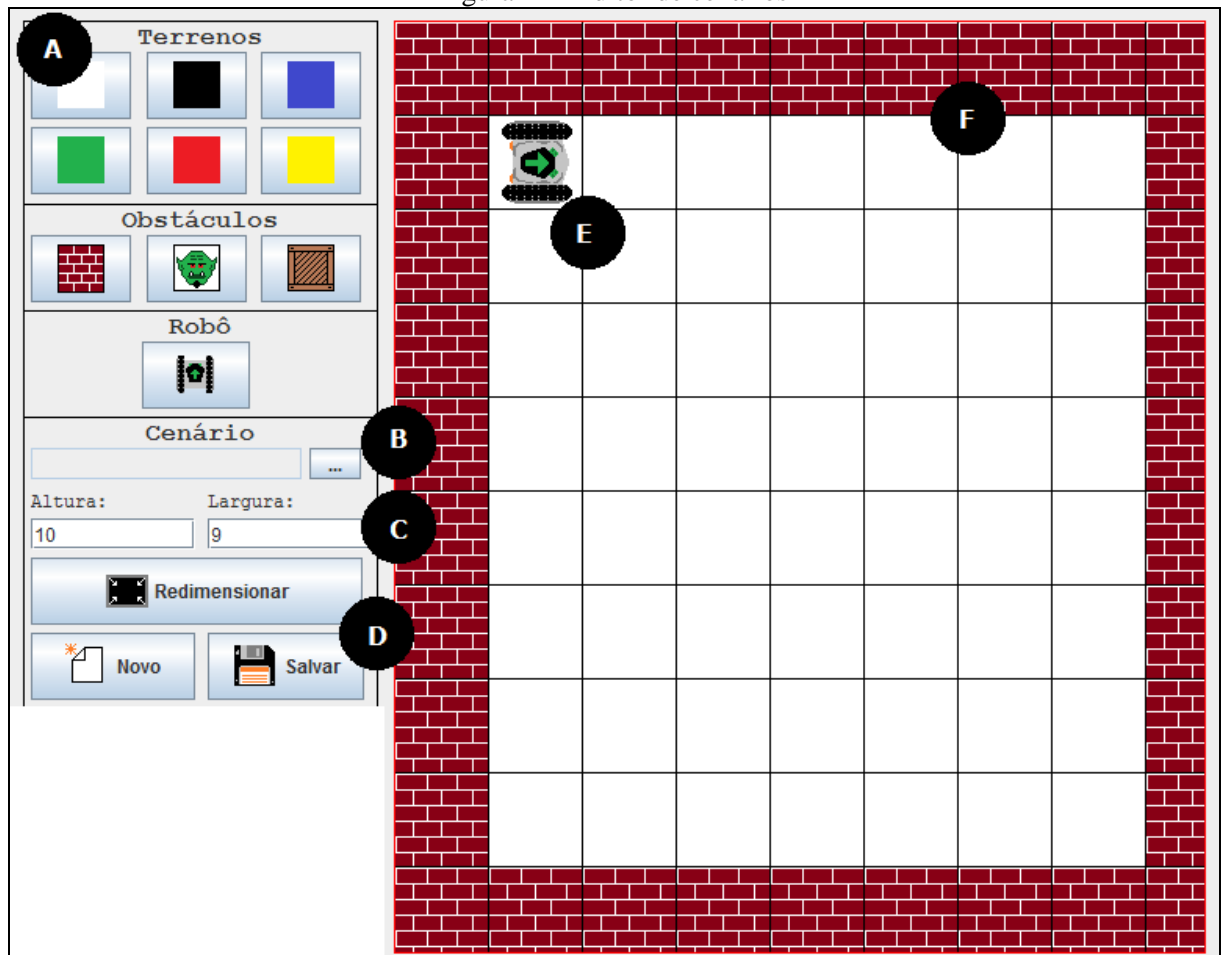
O compilador possui cinco botões com as funções necessárias para gerar programas na linguagem intermediária de *script*. Pode-se observar na Figura 11 a interface do compilador, com os seguintes componentes e respectivas funcionalidades: (A) botão `Novo`, que limpa o editor de programas (G) e, se houver algum texto editado, pergunta ao usuário se ele gostaria de salvar o trabalho; (B) botão `Abrir`, que permite selecionar um arquivo com extensão `.robotoy`, sendo que a pasta onde se encontra o arquivo aberto é apresentada no campo de texto (F) e o conteúdo apresentado no editor de programas (G); (C) botão `Salvar`, que, por sua vez, grava o texto editado em arquivos no formato `.robotoy`; (D) botão `Compilar`, que efetua as análises léxica, sintática e semântica do programa editado; (E) botão `Gerar Código`, que gera um arquivo `.comp` com a representação na linguagem intermediária de *script* do programa em Robotoy.

Figura 11 - Compilador



O editor de cenários, apresentado na Figura 12, possui uma barra de ferramentas com botões para edição de cenários, assim como o cenário 2D propriamente dito.

Figura 12 - Editor de cenários



O cenário em edição pode ser visualizado no componente (F). Para alterar o cenário, primeiro é preciso selecionar um objeto da barra de ferramentas (A), entre *Terrenos*, *Obstáculos* (paredes, criaturas ou outros obstáculos) e *Robô*. Os objetos podem ser incluídos no cenário apenas clicando no objeto desejado e clicando novamente no cenário. Segurar esse clique e arrastar inclui múltiplas células iguais a selecionada no cenário. Todos os objetos em

Terrenos são transponíveis, o que significa que o robô pode andar em cima deles. Em contrapartida, todos os objetos em *Obstáculos* não podem ser atravessados. Por fim, o cenário pode conter apenas um robô (E). Quando o robô é selecionado, é possível rotacionar o robô por inteiro (pressionando a tecla *R*) ou apenas a sua cabeça (pressionando as teclas *Shift+R*), definindo assim a posição inicial do mesmo. O robô pode ser removido apertando a tecla *ESC* quando o botão do mesmo tiver sido pressionado. O botão (B) serve para selecionar arquivos com a extensão *.cena*, que guardam as informações dos cenários salvos. O caminho para o arquivo selecionado é mostrado no campo de texto ao lado desse botão. Os campos altura e largura (C) indicam a quantidade de células do cenário e o botão *Redimensionar* (D) é utilizado para aplicar as alterações no tamanho do mesmo. Caso o tamanho em alguma das medidas seja menor que o tamanho atual, as células excedentes são descartadas. Os botões *Novo* e *Salvar* desempenham funções semelhantes às encontradas no compilador, sendo que um limpa o cenário e o outro salva o cenário em arquivo, respectivamente. É possível efetuar *zoom in* e *zoom out* no cenário utilizando a roda do *mouse*.

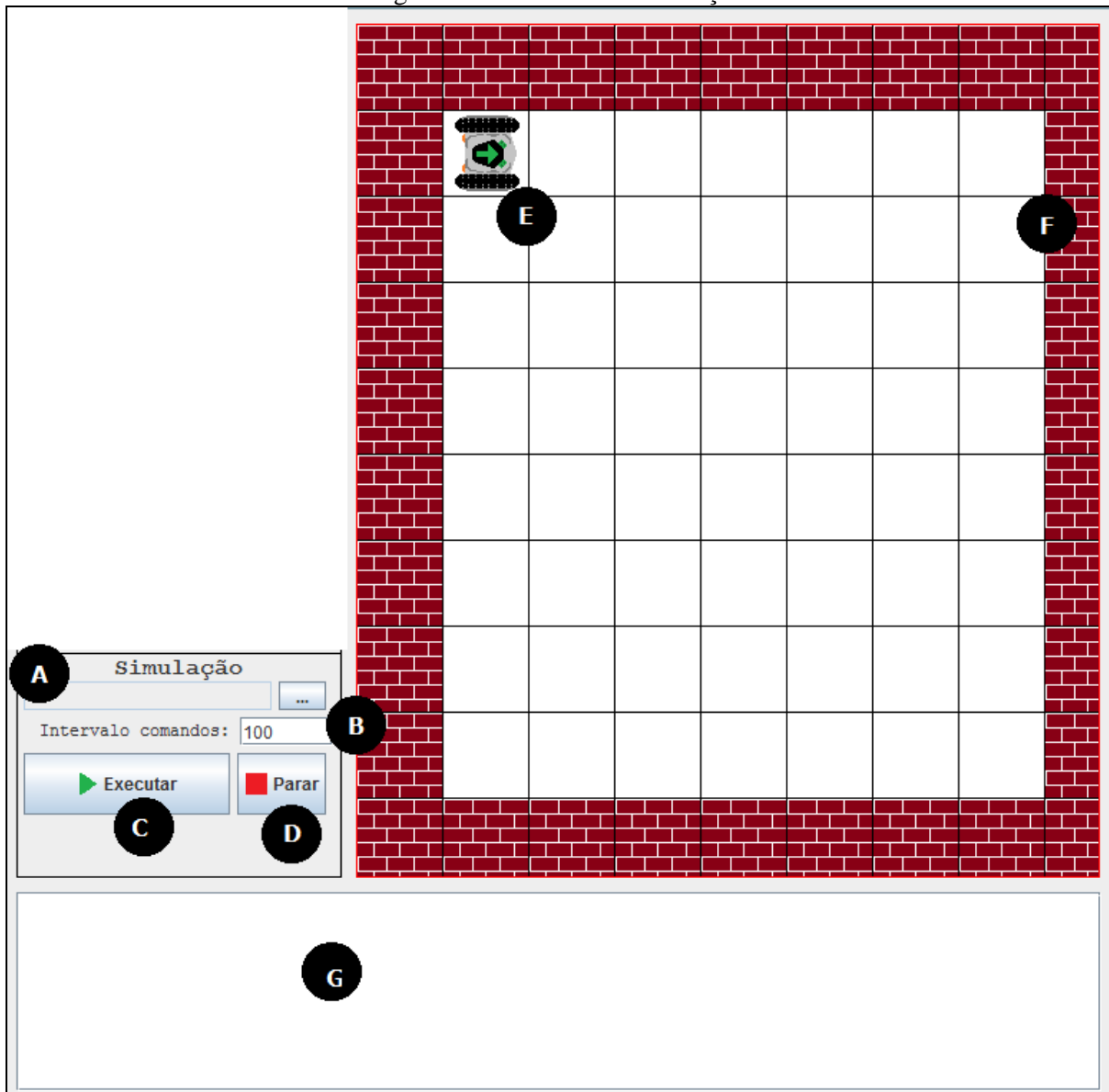
Conforme citado, os objetos a serem inseridos no cenário 2D são selecionados no componente (A) da Figura 12. Quando um dos botões é pressionado, é apresentada uma caixa de seleção, onde o usuário pode selecionar o objeto desejado. Tal caixa encontra-se destacada em vermelho na Figura 13.



Por fim, para realizar uma simulação, tem-se os componentes de interface observados na Figura 14, onde é possível visualizar o cenário (F) e o robô (E). Os arquivos em edição no compilador não são executados diretamente, ou seja, é necessário selecionar manualmente um arquivo *.comp* para ser interpretado. Para tal, deve-se: clicar no botão ao lado da caixa de texto (A); selecionar o arquivo *.comp*, contendo o código intermediário a ser simulado, que deve ser gerado utilizando o compilador previamente apresentado; clicar no botão *Executar* (C). A velocidade na qual a simulação roda depende da quantidade de quadros desenhados para cada ação de movimentação do robô. Tal quantidade pode ser alterada a qualquer momento nos campos presentes em *Intervalo comandos* (B). Este valor representa a quantidade de quadros necessários para executar uma ação de movimentação do robô. Assim

como a taxa de quadros, o cenário também pode ser alterado em tempo real, pois a simulação é controlada por uma *thread* separada do processo principal. Para interromper uma execução basta pressionar o botão `Parar` (D) e a simulação é encerrada. Caso o robô esteja efetuando uma ação de movimentação, o mesmo a finaliza e então a simulação é interrompida. A caixa de texto representada por G é o *display* do robô, onde é possível emitir mensagens através do comando `escrever`.

Figura 14 – Controle da simulação

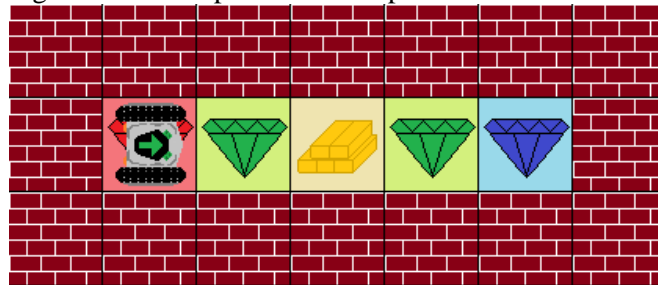


Quando uma simulação acaba, a mensagem “Execução finalizada: Fim do arquivo alcançado.” é apresentada no *display* (G), denotando o sucesso da execução da simulação.

3.5 RESULTADOS E DISCUSSÕES

Para validar o simulador foram elaborados cenários de testes e programas para serem executados nos mesmos. O exemplo apresentado na Figura 15 apresenta um cenário para um robô acumulador. Nesse cenário o robô deve andar por um corredor com pedras preciosas. Conforme identifica sobre qual pedra preciosa ele está posicionado, acumula pontos baseado na cor da mesma, sendo 200 pontos para o rubi (vermelho), 100 para a esmeralda (verde), 50 para o ouro (amarelo) e 250 para a safira (azul). O objetivo é acumular 2000 pontos ou mais. Assim, ao atingir o final do percurso, ele deve verificar se o valor acumulado é igual ou superior a 2000. Em caso negativo, o robô deve voltar pelo caminho e somar os valores novamente, até atingir o valor estipulado. A solução proposta para este problema envolve comandos de movimentação do robô, de identificação de obstáculo e cores, a chamada de rotina e uma estrutura de repetição encadeada. A solução para o problema do robô acumulador pode ser vista no Quadro 11.

Figura 15 - Exemplo de cenário para o robô acumulador



Quadro 11 - Solução do exercício do robô acumulador

Programa em Robotoy	
1	número valor <- 0
2	enquanto valor < 2000
3	enquanto não tem obstáculo
4	somarValor
5	andar para frente 1
6	fim do enquanto
7	somarValor
8	virar para a esquerda 2
9	fim do enquanto
10	texto resultado <- "Valor final: R\$ " . valor
11	escrever resultado
12	
13	rotina somarValor
14	se cor identificada = vermelho
15	valor <- valor + 200
16	fim do se
17	se cor identificada = verde
18	valor <- valor + 100
19	fim do se
20	se cor identificada = amarelo
21	valor <- valor + 50
22	fim do se
23	se cor identificada = azul
24	valor <- valor + 250
25	fim do se
26	fim da rotina

Programa na linguagem intermediária	
-------------------------------------	--

1	ROTINA; somarValor
2	COR_IDENTIFICADA
3	IF; cor_identificada == 'VERMELHO'
4	ATRIBUIR_VALOR_VARIAVEL; valor; valor + 200
5	ENDIF
6	COR_IDENTIFICADA
7	IF; cor_identificada == 'VERDE'
8	ATRIBUIR_VALOR_VARIAVEL; valor; valor + 100
9	ENDIF
10	COR_IDENTIFICADA
11	IF; cor_identificada == 'AMARELO'
12	ATRIBUIR_VALOR_VARIAVEL; valor; valor + 50
13	ENDIF
14	COR_IDENTIFICADA
15	IF; cor_identificada == 'AZUL'
16	ATRIBUIR_VALOR_VARIAVEL; valor; valor + 250
17	ENDIF
18	ENDROTINA
19	DECLARAR_VARIAVEL; tem_obstaculo; false
20	DECLARAR_VARIAVEL; cor_identificada; 'BRANCO'
21	DECLARAR_VARIAVEL; valor; 0
22	WHILE; valor < 2000
23	TEM_OBSTACULO
24	WHILE; ! tem_obstaculo
25	CHAMADA_ROTINA; somarValor
26	COMANDO_ROBO; EM_FRENTE; (1)
27	TEM_OBSTACULO
28	ENDWHILE
29	CHAMADA_ROTINA; somarValor
30	COMANDO_ROBO; VIRAR_ESQUERDA; (2)
31	ENDWHILE
32	DECLARAR_VARIAVEL; resultado; "Valor final: R\$ " + valor
33	ESCREVER; (resultado)

Seguindo o programa Robotoy, na linha 1 tem-se a criação da variável que é utilizada para acumular os valores. Nas linhas 2 e 3 tem-se os comandos enquanto responsáveis por repetir as ações até atingir o valor de 2000 e verificar se existe obstáculos impedindo a movimentação. A chamada da rotina `somarValor`, presente nas linhas 4 e 7, faz com que o robô execute os vários `if` da rotina (linha 13) que é responsável por tratar a cor identificada e o valor a atribuir. As linhas 5 e 8 possuem os comandos de movimentação do robô, para que ele se mova para frente e vire para trás, respectivamente, caso seja necessário mais uma rodada para acumular valores. Por fim, uma variável texto é alimentada (linha 10) com a mensagem final de saída, composta pelo valor total acumulado. Essa mensagem é então apresentada no *display* através do comando `escrever` da linha 11. No Quadro 11 também é possível visualizar o *script* correspondente gerado a partir do programa Robotoy.

Para editar os cenários não é necessário muito treinamento. Fora instruído um usuário, que não possuía conhecimento prévio sobre a ferramenta, a utilizar o *zoom*, a movimentação do cenário, o redimensionamento do mesmo e o posicionamento inicial do robô, o qual é muito importante, pois sem ele a simulação não pode ser iniciada. A partir das instruções iniciais, esse usuário conseguiu criar, salvar e modificar cenários propostos.

Outros exercícios foram elaborados a fim de testar melhor o simulador, sendo que os cenários e as respectivas soluções podem ser encontrados no Apêndice B. O primeiro tem o objetivo de desenvolver a lógica do programador, criando um programa para que o robô encontre a saída de labirintos. Já o segundo visa explorar os comandos de movimentação e detecção do robô. Em ambos os exercícios foi possível desenvolver os cenários e os programas, simulando-os na ferramenta.

Notou-se um problema no simulador: o tamanho dos cenários pode acarretar quedas no desempenho da simulação. Isso ocorre se existirem muitos objetos a serem desenhados no cenário 2D. A queda de desempenho começa a ser observada em cenários 15x15. Entretanto, pode ser contornada reduzindo a taxa de atualização por comando de movimentação do robô. Essa solução deixa de ser eficiente a partir do tamanho de 35x35, quando a execução, mesmo que com uma taxa pequena de quadros por comando, continua lenta.

Com relação aos trabalhos correlatos, o Quadro 12 traz uma comparação entre esses e a ferramenta desenvolvida, trazendo as características desses trabalhos.

Quadro 12 - Comparativo com os trabalhos correlatos

características	Furbot	Robomind FURB	Blockly	Simulador Robotoy
possui linguagem de alto nível simplificada e de fácil assimilação por usuários não programadores		X	X	X
possui linguagem intermediária acessível e que pode ser gerada a partir de qualquer linguagem de alto nível			X	X
possui ambiente de programação próprio		X	X	X
tipo de programação	textual	textual	visual	textual
permite simular os comandos em robôs virtuais	X	X		X
permite executar os comandos em robôs reais		X		
possui interface para editar cenários de simulação		X		X

Vale ressaltar que o item do quadro comparativo “permite simular os comandos em robôs virtuais” significa que a ferramenta permite que seja executada uma simulação onde um robô virtual efetua as ações especificadas por uma linguagem; e o item “permite simular os comandos em robôs reais” significa que a ferramenta possui um meio de direcionar os programas escritos para serem executados em robôs reais, construídos a partir da utilização de *kits* de robôs Lego Mindstorms NXT ou Arduínos, entre outros. Embora o simulador desenvolvido não tenha essa característica, isso é facilmente contornado ao incluir a geração de código em Java, como desenvolvido por Torrens (2014).

4 CONCLUSÕES

Hoje existem diversas metodologias para desenvolver as habilidades cognitivas de uma criança e a robótica é uma delas. A Robotoy surgiu para proporcionar uma maneira mais acessível para que as crianças pudessem desenvolver seus próprios processos de automação de robôs, ajudando nesse desenvolvimento cognitivo. Entretanto, para usar a linguagem Robotoy se fazia necessária a construção de um robô Lego Mindstorms NXT, o que limitava o acesso a este objeto de aprendizado. Baseado nesse pressuposto, decidiu-se elaborar um simulador para a linguagem Robotoy de forma a permitir criar cenários e executar os comandos do robô, dispensando assim a aquisição / o uso de um *kit* de robótica.

Com a execução de vários programas desenvolvidos e a construção de variados cenários, pode-se concluir que os objetivos estabelecidos foram atingidos. O simulador desenvolvido possui um editor de cenários 2D, um editor de programas em Robotoy e nos mesmos cenários que podem ser editados, o simulador pode executar os programas Robotoy compilados. Para simular os programas, foi desenvolvida uma linguagem intermediária em formato de *script*.

Desenvolver esta linguagem foi necessário a fim de facilitar o desenvolvimento da simulação. Primeiro foram definidos os passos que a simulação tomaria durante a execução. Com isso, foi mais fácil criar uma linguagem que atendesse a todos os requisitos que seriam necessários para seguir esses passos, pois adaptar uma linguagem já existente para se adequar às regras do interpretador levaria mais tempo.

No decorrer da implementação houve problemas quanto ao uso da biblioteca JOGL, que por não ter adquirido a documentação da mesma, fora necessário diversos testes em relação a como carregar texturas no cenário 2D. Há dois parâmetros que devem ser passados para o método que carrega as texturas da JOGL que estavam sendo informados de maneira incorreta. Um era o tipo de imagem de entrada (não o formato dos arquivos de imagem, mas sim suas características como: preto e branco, *Red, Green and Blue* (RGB), entre outras) e o outro o tipo interno na JOGL, que por não informá-los corretamente acarretava na geração de texturas cortadas ou em pequenos fragmentos de cores variadas.

4.1 EXTENSÕES

Como sugestão de extensões para o simulador propõe-se:

- a) substituir o editor de cenários 2D por um 3D;
- b) adicionar os comandos da linguagem Robotoy que foram suprimidos, conforme

explanado na seção 3.2, além de incluir comandos novos, como `pegar`, presente no Robomind FURB, gerando o código correspondente na linguagem intermediária e, conseqüentemente, permitindo a simulação dos mesmos;

- c) adicionar outras linguagens ao compilador, para que os usuários possam optar por diferentes linguagens fonte para então gerar o código intermediário;
- d) melhorar o desempenho do simulador em relação a execução em cenários muito grandes, conforme salientado na seção 3.5;
- e) criar uma ferramenta complementar para o simulador com a finalidade de tornar possível que os usuários criem suas próprias texturas e adicionem-nas ao simulador.

REFERÊNCIAS

- BASTOS, Henrique. **Diferenças entre linguagem compilada e linguagem interpretada**. [S.l.], 2008. Disponível em: <<http://henriquebastos.net/diferencas-entre-linguagem-compilada-e-linguagem-interpretada/>>. Acesso em: 27 abr. 2016.
- BATISTA, Thaís V.; RODRIGUEZ, Noemi. LuaSpace: um ambiente para reconfiguração dinâmica de aplicações baseadas em componentes. In: WORKSHOP DE TESES E DISSERTAÇÕES, 16., 2001, Fortaleza. **Anais...** Fortaleza: SBC, 2001. p. 1-8. Disponível em: <<http://www.dimap.ufrn.br/~thais/Publicacoes/ctdfinal.pdf>>. Acesso em: 27 abr. 2016.
- BENITTI, Fabiane B. V. et al. Robótica como elemento motivacional para atração de novos alunos para cursos de computação. In: CONGRESO IBEROAMERICANO DE EDUCACIÓN SUPERIOR EN COMPUTACIÓN, 18., 2010, San Lorenzo, Paraguay. **Proceedings...** San Lorenzo: [s.n.], 2010. Não paginado. Disponível em: <http://www.inf.furb.br/dsc/download/ciesc2010_submission_16.pdf>. Acesso em: 05 set. 2015.
- BIBBY, Joe; NECESSARY, Ryan. **Simulation**. [S.l.], 2008. Disponível em: <<http://robonaut.jsc.nasa.gov/R1/sub/simulation.asp>>. Acesso em: 08 set. 2015.
- CAMARGO, Heloisa A. **Paradigmas de linguagens de programação**. [S.l.], 2009. Disponível em: <http://www2.dc.ufscar.br/~heloisa/PLP2009/LP_geral.pdf>. Acesso em: 27 abr. 2016.
- CASTILHO, Maria I. **Robótica na educação: com que objetivos?** [Porto Alegre], [2009]. Disponível em: <<http://www.pucrs.br/eventos/desafio/2007/mariaines.php#raclog>>. Acesso em: 4 set. 2015
- CHUNG, Christopher A. **Simulation modeling handbook: a practical approach**. Florida: CRC Press, 2003.
- COOPER, Keith D.; TORCZON, Linda. **Construindo compiladores**. Tradução Daniel Vieira. 2.ed. Rio de Janeiro: Elsevier, 2014.
- CROCKFORD, Douglas. **Javascript: the world's most misunderstood programming language**. [S.l.], 2001. Disponível em: <<http://www.crockford.com/javascript/javascript.html>>. Acesso em: 02 maio 2016.
- ERLANG.COM.BR. **Simulação**. [S.l.], 2015. Disponível em: <<http://www.erlang.com.br/simulacao.asp>>. Acesso em: 08 set. 2015.
- GOOGLE DEVELOPERS. **What is Blockly?** [S.l.], 2015. Disponível em: <<https://developers.google.com/blockly/about/faq>>. Acesso em: 05 set. 2015.
- GROLLA, Elaine. **A aquisição da linguagem**. [Florianópolis?], 2006. Disponível em: <<http://stoa.usp.br/egrolla/files/-1/17317/Aquisicao+de+linguagem.pdf>>. Acesso em 05 set. 2015.
- GUERBER, Carlos. **Compiladores e interpretadores**. Mafra, 2007. Disponível em: <<http://www.mfa.unc.br/info/carlosrafael/aco/aula16.pdf>>. Acesso em: 17 maio 2016.
- HOUNSELL, Marcelo S.; REDEL, Rubens. Implementação de simuladores de robôs com o uso da tecnologia de realidade virtual. In: CONGRESSO BRASILEIRO DE COMPUTAÇÃO, 4., 2004, Itajaí. **Anais...** Itajaí: UNIVALI, 2004. p. 396-401. Disponível em: <http://www.niee.ufrgs.br/eventos/CBCOMP/2004/pdf/Robotica/t170100201_3.pdf>. Acesso em: 08 set. 2015.

OLIVEIRA, Thiago H. B.; SILVA, Cherlia; ANDRADE, Mariel J. P. Scratch: utilizando programação para desenvolvimento do raciocínio lógico em crianças. In: JORNADA DE ENSINO, PESQUISA E EXTENSÃO, 13., 2013, Recife. **Anais...** Recife: UFRPE, 2013. Não paginado. Disponível em: <<http://www.eventosufrpe.com.br/2013/cd/resumos/R1503-2.pdf>>. Acesso em: 14 set. 2015.

ROBOMIND ACADEMY. **Introduction:** what is Robo/RoboMind? [S.l.], 2014. Disponível em: <<http://www.robomind.net/en/introduction.htm>>. Acesso em: 05 set. 2015.

SMITH, Roger D. **Simulation article.** [S.l.], 1998. Disponível em: <<http://www.modelbenders.com/encyclopedia/encyclopedia.html>>. Acesso em: 08 set. 2015.

SWH14TEAM6. **Ghost Polaroid.** [S.l.], 2014. Disponível em: <<https://sketchingwithhardware.wordpress.com/2014/04/17/ghost-polaroid/>>. Acesso em: 31 maio 2016.

TORRENS, Maria G. **Robotoy:** ferramenta para uso de robótica no ensino de programação para crianças. 2014, 71 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <http://dsc.inf.furb.br/arquivos/tccs/monografias/2014_2_maria-gabriela-torrens_monografia.pdf>. Acesso em: 08 set. 2015.

VAHLDIK, Adilson et al. **Projeto FURBOT:** introdução. [Blumenau], 2008. Disponível em: <<http://www.inf.furb.br/poo/furbot/introducao.html>>. Acesso em: 05 set. 2015.

APÊNDICE A – Especificação da linguagem intermediária

O Quadro 13 apresenta a especificação da linguagem intermediária.

Quadro 13 - Especificação sintática da linguagem intermediária

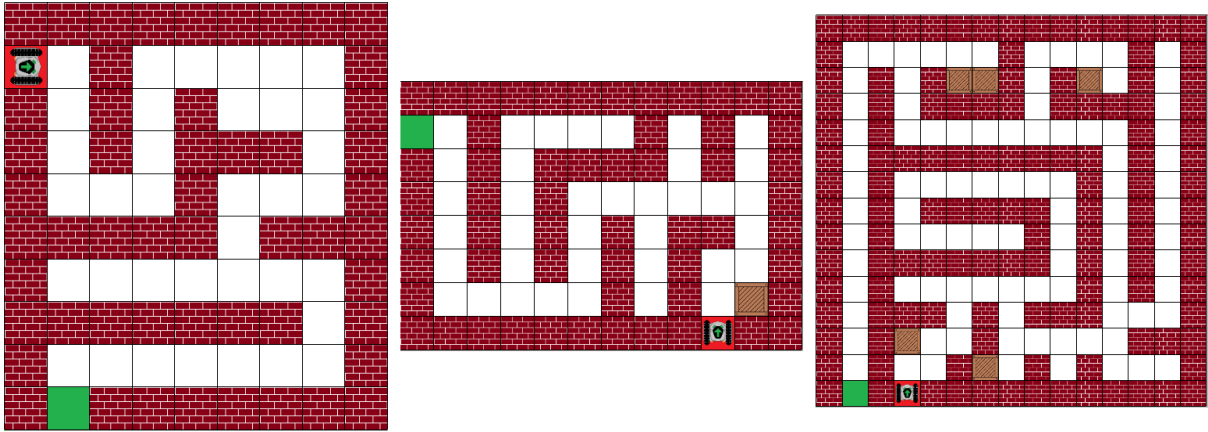
<pre> <programa> ::= <lista_de_rotinas> <lista_de_comandos> <lista_de_rotinas> ::= ε <rotina> <lista_de_rotinas> <rotina> ::= ROTINA; identificador quebra_de_linha <lista_de_comandos> ENDROTINA quebra_de_linha <lista_de_comandos> ::= <comando> <comando> quebra_de_linha <lista_de_comandos> <comando> ::= <controle_de_fluxo> <definição_de_variável_ou_invocação_de_rotina> <ação> </pre>
<pre> <controle_de_fluxo> ::= <se> <enquanto> <se> ::= <tem_obstáculo> <cor_identificada> IF; <condição> quebra_de_linha <lista_de_comandos> <senão> ENDIF <senão> ::= ε ELSE quebra_de_linha <lista_de_comandos> <enquanto> ::= <tem_obstáculo> <cor_identificada> WHILE; <condição> quebra_de_linha <lista_de_comandos> <tem_obstáculo> <cor_identificada> ENDWHILE <tem_obstáculo> ::= TEM_OBSTACULO ε <cor_identificada> ::= COR_IDENTIFICADA ε </pre>
<pre> <definição_de_variável_ou_invocação_de_rotina> ::= ATRIBUIR_VALOR_VARIÁVEL; identificador ; <expressão> DECLARAR_VARIÁVEL; identificador ; <expressão> identificador </pre>
<pre> <ação> ::= <andar> <virar> <escrever> <emitir_som> <detecção> <andar> ::= COMANDO_ROBO; <frente_ou_tras> ; <expressão> <virar> ::= COMANDO_ROBO; <robô_ou_cabeça> ; <expressão> <escrever> ::= ESCREVER; <expressão> <emitir_som> ::= COMANDO_ROBO; EMITIR_SOM; 0 <detecção> ::= TEM_OBSTACULO COR_IDENTIFICADA <frente_ou_tras> ::= EM_FRENTE RECUAR <robô_ou_cabeça> ::= VIRAR_ESQUERDA VIRAR_DIREITA VIRAR_CABECA_ESQUERDA VIRAR_CABECA_DIREITA </pre>
<pre> <condição> ::= <tipo_condição> <condição_extra> <tipo_condição> ::= <verificar_obstáculo> <comparar_numerico_literal> <condição_extra> ::= ε "&&" <condição> " " <condição> <verificar_obstáculo> ::= TEM_OBSTACULO quebra_de_linha <não> tem_obstaculo <não> ::= ε "!" <comparar_numerico_literal> ::= <expressão> <operador_relacional> <expressão> <operador_relacional_numerico> ::= ">" "<" ">=" "<=" "=" "=/ </pre>
<pre> <expressão> ::= <aritmética> <aritmética> ::= <operação> <mais_menos> <mais_menos> ::= ε "+" <operação> <mais_menos> "-" <operação> <mais_menos> <operação> ::= <termo> <vezes_divisão> <vezes_divisão> ::= ε "*" <termo> <vezes_divisão> "/" <termo> <vezes_divisão> "%" <termo> <vezes_divisão> <termo> ::= "+" <termo> "-" <termo> identificador <concatenar_literal> numérico "(" <aritmética> ")" literal <concatenar_literal> </pre>

```
<cor> | <comando_identificar_cor> cor identificada
<cor> ::= branca | branco | preta | preto | vermelha | vermelho |
        amarela | amarelo | verde | azul
<comando_identificar_cor> ::= COR_IDENTIFICADA quebra_de_linha
<concatenar_literal>      ::= ε | "+" <opções_de_concatenação>
<opções_de_concatenação> ::= identificador <concatenar_literal> |
        "(" <aritmética> ")" <concatenar_literal> |
        literal <concatenar_literal>
```

APÊNDICE B – Lista de exercícios

No Quadro 14 e no Quadro 15 são apresentados os exercícios utilizados para validar o simulador e a linguagem intermediária.

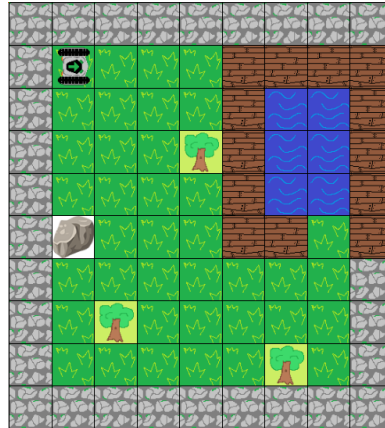
Quadro 14 - Exercício 1: labirinto

Enunciado:
<p>Crie um programa para encontrar a saída do labirinto (célula verde no cenário). Um único programa deve ser desenvolvido para a resolução desse problema em todos os cenários apresentados.</p>
Cenários:

Solução:
<pre> virar motor multiuso para a esquerda 1 número x <- 0 enquanto cor identificada != VERDE resto fim do enquanto escrever "Fim do labirinto" rotina resto se tem obstáculo virar motor multiuso para a direita 1 se tem obstáculo virar para a direita 1 virar motor multiuso para a esquerda 1 senão andar para frente 1 virar motor multiuso para a esquerda 1 fim do se senão virar para a esquerda 1 andar para frente 1 fim do se fim da rotina </pre>

Quadro 15 - Exercício 2: encontrar água

Enunciado:

Desenvolva um programa que leve o robô a encontrar água. Ao chegar no local desejado, o robô deve verificar se encontrou mesmo água e emitir uma mensagem dizendo se em sua posição final há ou não há água.

Cenário:**Solução:**

```
andar para frente 2
virar para a direita 1
andar para frente 5
virar para a esquerda 1
andar para frente 4
virar para a direita 1
andar para trás 3
se cor identificada = azul
    escrever "Água encontrada"
senão
    escrever "Não fora encontrada água"
fim do se
```

ANEXO A – Texturas

A Figura 16 apresenta duas das texturas usadas no simulador (SWH14TEAM6, 2014). Ambas foram redimensionadas a fim de comportarem o tamanho padrão de 60x60 pixels das células dos cenários.

Figura 16 - Texturas

