Curso de Ciência da Computação
Universidade Estadual de Mato Grosso do Sul

DESENVOLVIMENTO DE DOIS JOGOS COMO FERRAMENTAS DE AUXÍLIO PARA O ENSINO DE LÓGICA DE PROGRAMAÇÃO

DANIEL KENZO TAKAGI

Profª Drª Mercedes Rocío Gonzales Márquez (Orientador)

DOURADOS/MS.

2019

DESENVOLVIMENTO DE DOIS JOGOS COMO FERRAMENTAS DE AUXÍLIO PARA O ENSINO DE LÓGICA DE PROGRAMAÇÃO

DANIEL KENZO TAKAGI

Trabalho de Conclusão de Curso para a obtenção do Grau de Bacharel em Ciência da Computação na Área de Ciências Exatas e da Terra da UEMS.

Dourados, 14 de novembro de 2019.

Profª Drª Mercedes Rocío Gonzales Márquez (Orientador)

Curso de Ciência da Computação
Universidade Estadual de Mato Grosso do Sul

DESENVOLVIMENTO DE DOIS JOGOS COMO FERRAMENTAS DE AUXÍLIO PARA O ENSINO DE LÓGICA DE PROGRAMAÇÃO

Daniel Kenzo Takagi

Novembro de 2019

Banca Examinadora:

Dra. Mercedes Rocío Gonzales Márquez (Orientador)

Área de Computação – UEMS

Prof Me. André Chastel Lima

Área de Computação – UEMS

Prof. Me. Yuri Karan Benevides Tomas

Área de Computação – IFMS

AGRADECIMENTOS

.

Agradeço a minha família, em especial minha mãe que cuidou sozinha de mim e de meu irmão.

Agradeço os meus professores pelo conhecimento adquirido na instituição.

Agradeço a minha orientadora que me ajudou na organização e pesquisa deste trabalho.

RESUMO

O aprendizado de uma linguagem de programação é considerado difícil por se tratar da codificação da solução de um problema seguindo um raciocínio lógico que envolve uma abstração de alto nível. Muitas vezes os métodos convencionais de ensino não são eficientes para conseguir a assimilação desta abstração por parte dos alunos, precisando, portanto, formas de ensino inovadoras. Os jogos computacionais educativos são uma dessas propostas inovadoras de ensino.

Neste trabalho apresentam-se dois jogos educativos inéditos como alternativa para auxiliar na solução deste problema. São jogos que objetivam auxiliar no aprendizado de conceitos de linguagem de programação, como são o conhecimento de estruturas de fluxo de execução, tais como estrutura sequencial, estrutura de seleção e estrutura de repetição, além de familiarizar o jogador com o uso correto de funções que comandam ações no jogo. Considerando que estamos diante de um aluno do século 21, muito familiarizado com o uso de jogos de computador, acredita-se que o fator lúdico dos jogos propostos possa ser uma alternativa interessante para conseguir uma interação mais atraente do aluno com o aprendizado de lógica de programação.

Ao decorrer do trabalho, serão mostrados detalhes do desenvolvimento dos jogos, que foram desenvolvidos na ferramenta Unity e com o auxílio de outras ferramentas, como o Blender.

Palavras-chave: jogos, linguagem de programação, ensino, Unity.

ABSTRACT

The learning of a programming language is considered difficult because it is the codification of the solution of a problem following a logical reasoning that involves a high level abstraction.

Often the conventional methods are not efficient to get the assimilation of that abstraction by the students, requiring inovating teaching proposals.

In this work are shown two unprecedented educative games as an alternative to solve this problem. These are games that aim to assist in the learning of programming language concepts, such as the knowledge of execution flow structures such as sequential structure, selection structure and repetition structure, as well as familiarizing the player with the correct use of command functions actions in the game. Considering that we are facing a 21st century student who is very familiar with the use of computer games, it is believed that the playfulness of the proposed games may be an interesting alternative to achieve a more attractive interaction of the student with the learning of computer logic programming.

Key-Words: games, programming language, study, Unity.

SUMÁRIO

AGRADECIMENTOS.....7
RESUMO.....9
ABSTRACT.....11
LISTA DE FIGURAS.....17
1. INTRODUÇÃO.....19
1.1 OBJETIVO.....22
1.1.1 Objetivos específicos.....22
1.2 Jogos educativos ou *serious games*.....22
1.3 A importância de novos métodos de ensino da programação.....22
1.4 Trabalhos correlatos.....22
1.5 Algoritmos: Estruturas de Controle de Fluxo de Execução de Comandos e funções.....24
1.5.1 Estrutura sequencial.....24
1.5.2 Estrutura de seleção.....24
1.5.3 Estrutura de repetição.....26
1.5.4 Chamada de funções.....27
1.6. Unity 3D.....27
1.7 Linguagem C Sharp.....28
1.8 Blender.....28
1.9 Aseprite e Inkscape.....28
1.10 Metodologia.....28
1.11 Organização dos capítulos.....29
2. Game Design Document (GDD).....29
2.1 Primeiro jogo: Illuminate.....29
2.1.1 Conceitos do jogo.....29
2.1.1.1 Características principais.....29
2.1.1.2 Gênero.....30
2.1.1.3 Plataforma.....30
2.1.2 Proposta do jogo.....30
2.1.2.1 Público alvo.....30
2.1.2.2 Comparação de características.....30
2.2.2.3 Recursos estimados (orçamento).....30
2.1.3 Projeto do jogo.....31
2.1.3.1 Elaboração do jogo.....31
2.1.3.2 Mecânica do jogo.....32
2.1.3.3 Fluxo.....32
2.1.3.4 Personagens.....33
2.1.3.5 Elementos do *gameplay*.....33
2.1.3.6 História.....33
2.1.3.7 Efeitos sonoros.....33
2.1.3.8 Artes.....34
2.2 Segundo jogo: Robot.exe.....34
2.2.1 Conceitos do jogo.....34
2.2.1.1 Características principais.....34
2.2.1.2 Gênero.....34
2.2.1.3 Plataforma.....34
2.2.2 Proposta do jogo.....35
2.2.2.1 Público alvo.....35

2.2.2.2 Comparação de características.....35
2.2.2.3 Recursos estimados (orçamento).....35
2.2.3 Projeto do jogo.....35
2.2.3.1 Elaboração do jogo.....35
2.2.3.2 Mecânica do jogo.....36
2.2.3.3 Fluxo.....36
2.2.3.4 Personagens.....36
2.2.3.5 Elementos do *gameplay*.....37
2.2.3.6 História.....38
2.2.3.7 Efeitos sonoros.....39
2.2.3.8 Artes.....39
3. IMPLEMENTAÇÃO.....41
3.1 Implementação do jogo Illuminate.....43
3.1.1 Caixa de texto para entrada de dados.....43
3.1.2 Comandos do usuário.....44
3.1.2.1 Comando de ação andar.....44
3.1.2.2 Comando de ação voltar.....45
3.1.2.3 Comando de ação esquerda.....46
3.1.2.4 Comando de ação direita.....47
3.1.2.5 Comando se.....47
3.1.2.6 Comando repita.....51
3.1.2.7 Sensor de colisões.....52
3.2 Implementação do jogo Robot.exe.....53
3.2.1 Caixa de texto para entrada de dados (Input Field).....54
3.2.2 Histórico de comandos.....55
3.2.3 Comandos do jogador.....57
3.2.3.1 Andar.....58
3.2.3.2 Pular.....59
3.2.3.3 Descer.....59
3.2.3.4 Atacar.....60
3.2.3.5 Defender.....61
3.2.3.5 Ativar.....62
3.2.3.6 Encolher.....63
3.2.3.7 Crescer.....64
3.2.4 Botão de informação.....65
3.2.5 Botão de pausar.....65
3.2.6 Os personagens.....65
3.2.7 Objetos destrutíveis.....66
3.2.8 Botões de interação.....67
3.2.9 Cenários.....67
3.2.9.1 Cena 1.....68
3.2.9.2 Cena 2.....68
3.2.9.3 Cena 3.....69
3.2.9.4 Cena 4.....69
3.2.9.5 Cena 5.....70
3.2.9.6 Cena 6.....71
3.2.10 Câmera.....72
3.2.10.1 Mover a câmera se o cursor do mouse estiver na borda.....72
3.2.10.2 Zoom In e Zoom Out da câmera 74
3.2.11 Cena cinemática.....74

3.2.12 Nome dos objetos....75
3.2.13 Número de comandos....75
4. TESTES E FEEDBACK DO PÚBLICO ALVO....77
5. CONCLUSÃO....83
5.1 Dificuldades encontradas....83
5.2 Trabalhos futuros....83
REFERÊNCIAS BIBLIOGRÁFICAS....85

LISTA DE FIGURAS

Figura 1: Jogo LightBot....22
Figura 2: Jogo Code Combat....22
Figura 3: Em jogo....31
Figura 4: Tela inicial....32
Figura 5: Tela de menu....32
Figura 6: Personagem de Illuminate....33
Figura 7: Personagem Robot....36
Figura 8: Personagem Formiga....37
Figura 9: Personagem Lutador....37
Figura 10: Bibliotecas....41
Figura 11: Classe Monobehaviour....41
Figura 12: Funções Start e Update....42
Figura 13: Componentes do objeto Player....42
Figura 14: Uso da função Split....43
Figura 15: Função TratamentoMinusc()....44
Figura 16: Laço de repetição da sub-rotina _execute()....44
Figura 17: Código da função de ação andar....45
Figura 18: Código do comando de ação voltar....46
Figura 19: Código da função viraEsquerda()....46
Figura 20: Código da atualização da rotação....47
Figura 21: Código da função viraDireita()....47
Figura 22: Código que define a posição do fimse....48
Figura 23: Verificação do == ou !=....48
Figura 24: Chamada da função Condicional....48
Figura 25: Função Condicional....49
Figura 26: Função VerifSeExistem....50
Figura 27: Verificação do "se"....51
Figura 28: Lendo o valor entre parênteses do comando repita....52
Figura 29: Parte principal da estrutura repita....52
Figura 30: Sensores....52
Figura 31: Script do sensor de colisões....53
Figura 32: Código da caixa de texto para entrada de dados....54
Figura 33: Função SendMessageToChat....55
Figura 34: Classe Message....56
Figura 35: Deleta mensagem....56
Figura 36: Alterando cor da mensagem e removendo-a....57
Figura 37: Parte do código da ação de andar....58
Figura 38: Parte do código da ação de pular....59
Figura 39: Código da ação de descer....60
Figura 40: Parte do código da ação de atacar....61
Figura 41: Código que verifica colisão, inserido em um personagem....61
Figura 42: Código da leitura da ação de defender....62
Figura 43: Código de defender insderido em um personagem....62
Figura 44: Código da ação ativar....63
Figura 45: Código da ação encolher....64
Figura 46: Código da ação de crescer....64

Figura 47: Componentes dos objetos destrutíveis...67
Figura 48: Cena 1..68
Figura 49: Cena 2..68
Figura 50: Cena 3..69
Figura 51: Cena 4..70
Figura 52: Cena 5..70
Figura 53: Cena 6..71
Figura 54: Código da instanciação dos inimigos...72
Figura 55: Código da movimentação da câmera...73
Figura 56: Limitação de movimento da câmera...73
Figura 57: Parte do funcionamento do código de cenas cinemáticas..74
Figura 58: Código que atribui nome aos objetos...75
Figura 59: Representação do nome dos objetos...75
Figura 60: Contagem de comandos...76
Figura 61: Desempenho do jogador...76
Figura 62: Gráfico da opinião sobre o design do jogo..77
Figura 63: Gráfico sobre o entendimento da jogabilidade..78
Figura 64: Gráfico sobre a ambientação do jogo...78
Figura 65: Gráfico sobre a monotonia do jogo...79
Figura 66: Gráfico a respeito da história do jogo..79
Figura 67: Gráfico sobre até que fase jogaram...80
Figura 68: Gráfico de notas gerais..80
Figura 69: Gráfico da atuação profissional/escolar dos usuários...81
Figura 70: Sugestões dos jogadores...82

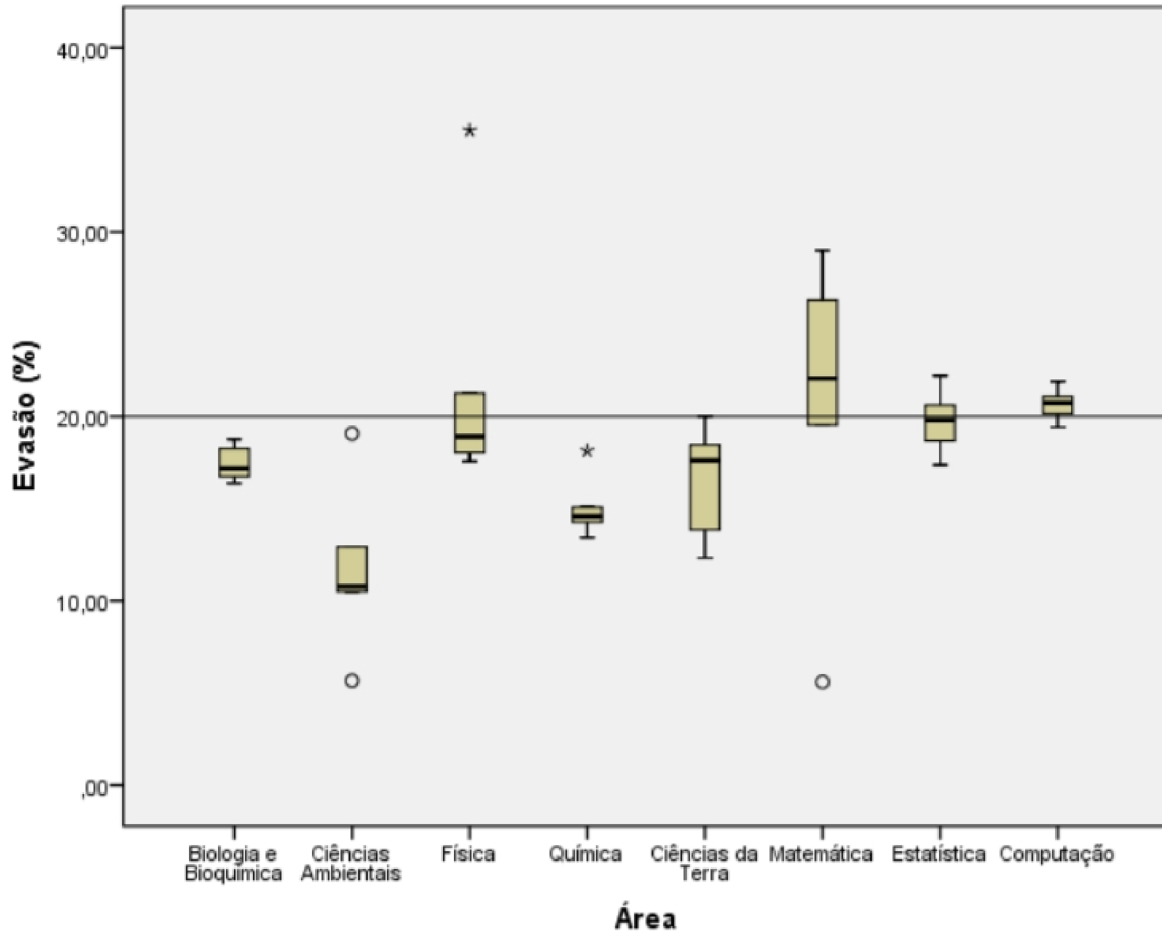
1. INTRODUÇÃO

Um problema que professores têm em sala de aula é o desinteresse e a falta de concentração do aluno para aprender o tema proposto. Além disso, o ensino de uma linguagem de programação se torna desafiador devido à abstração de alto nível presente no raciocínio lógico que envolve a codificação de um programa de computador (ROCHA, 2010). Segundo MOSER existem alguns fatores que contribuem para a dificuldade no aprendizado de uma linguagem de programação. Estes são: Programar é uma habilidade multicamada no sentido que o aprendiz deve passar por vários estágios de aprendizado, primeiro a sintaxe, depois as estruturas e estilo de programação. A experiência de programar não está relacionada com nenhuma experiência cotidiana, isto é, o aprendiz não conhece nenhum paralelo na sua experiência, com o qual possa relacionar o conhecimento novo adquirido. Aprende-se a programar, portanto, em um único contexto, a sintaxe deve ser aprendida e só depois aplicada. O aprendizado destas regras de sintaxe só para uma aplicação posterior na construção de programas, faz com que isto se torne entendiante para os alunos.

Cabe mencionar que MONCLAR(2018) considera como fator importante na dificuldade que os alunos enfrentam para programar, a falta de contato com o conteúdo no ensino básico. Isto foi considerado por países como a Austrália, Estados Unidos e Inglaterra que já perceberam a importância da linguagem de programação no ensino básico, integrando a matéria ao currículo das séries iniciais da formação escolar (COSTA et al, 2017).

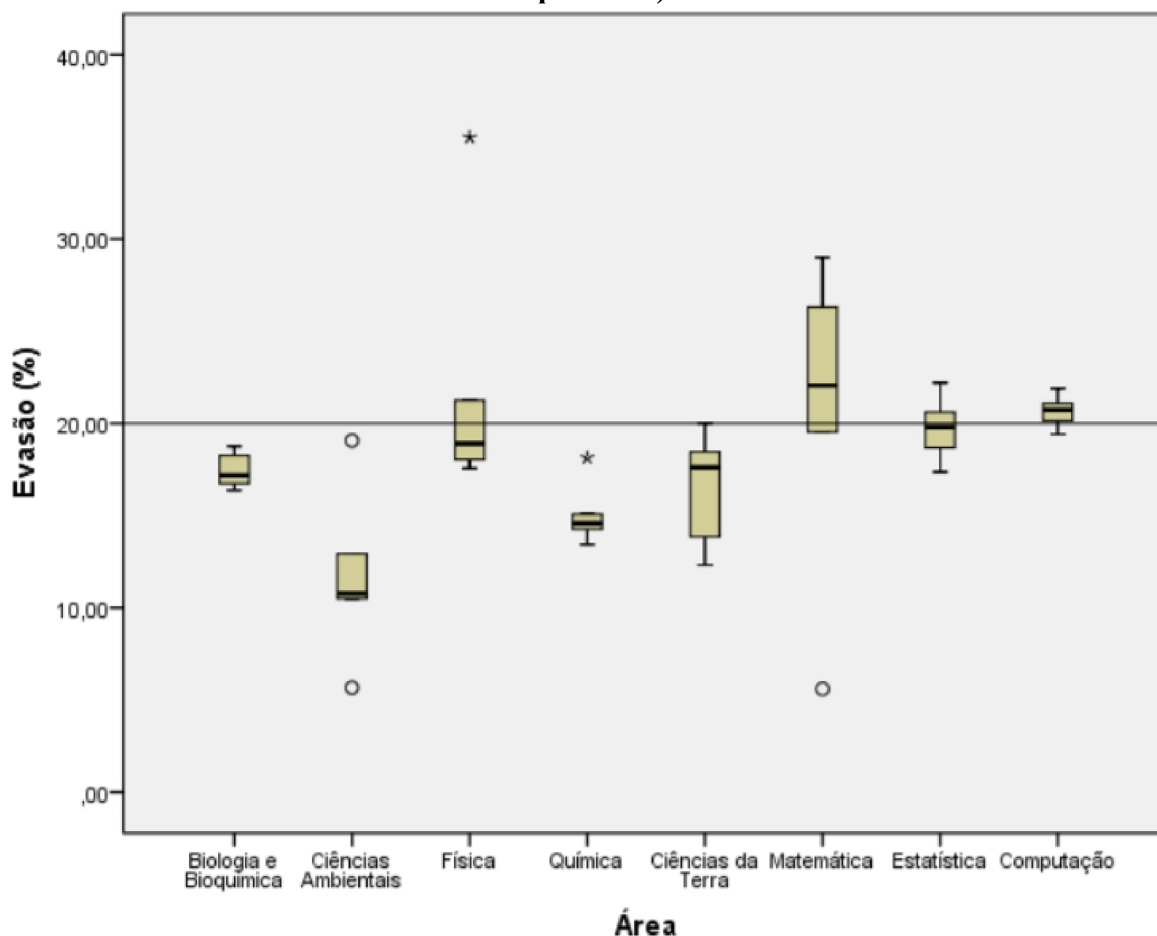
Como consequência disso, tem ocorrido grande evasão de alunos na área da computação, como podemos ver na Figura 1 e 2 (dados retirados dos anos 2010 a 2014):

Figura 1: Evasão na grande área de Ciências, Matemática e Computação (instituições privadas)



Fonte: HOED, 2016, p. 59.

Figura 2: Evasão na grande área de Ciências, Matemática e Computação (instituições privadas)



Fonte: HOED, 2016, p. 60.

O contato com o lúdico é muito importante para aprendizagem "...a aprendizagem lúdica tem tomado espaço de discussão no contexto escolar pelos profissionais da educação, como sendo um facilitador da aprendizagem, pois proporciona entusiasmo e motivação aos educandos na construção do saber." (RODRIGUES, 2013).

Além disso, com o maior acesso da Internet pelas pessoas no geral, os jogos têm se tornado em uma grande indústria do mercado, arrecadando bilhões de dólares a cada ano. Em relação a isso, sabe-se que o número de jogadores (de jogos digitais) também cresceu nos últimos anos. Em 2018, no Brasil, estimava-se cerca de 75 milhões de jogadores ajudando a movimentar o mercado (NEWZOO, 2018).

Tendo em mente a grande quantidade de pessoas que possuem contato com jogos digitais, por que não utilizá-lo na área da aprendizagem? E é isso o que empresas, escolas e universidades têm feito.

Os jogos digitais para aprendizagem tem sido bastante utilizado, podendo ser intitulados *serious games*. Os *serious games*:

> "Facilitam a comunicação de conceitos/fatos – devido à dramatização de problemas e motivação – além de contribuírem para o desenvolvimento de estratégias, a tomada de decisão, o desempenho de papéis, dentre outras vantagens, em um ambiente em que o *feedback* é instituído de maneira ágil" (ABT in FLEURY et al, 2014).

Hoje em dia há muitos *serious games*, pois através deles desenvolvem-se muitas habilidades e conhecimentos, além de ser uma forma mais prazerosa e encantadora de se aprender (GRUBEL, BEZ - 2006).

Diante das dificuldades dos alunos da computação em assimilar os conceitos abstratos envolvidos na lógica de programação foi proposto neste trabalho uma alternativa para a solução do problema: o desenvolvimento de dois jogos inéditos para auxiliar nesta aprendizagem.

O jogo Illuminate, que consiste de apenas uma fase, o qual tem o objetivo de introduzir o aluno no uso das estruturas de fluxo de execução de comandos em uma linguagem de programação. Estas são: Estrutura Sequencial na qual um conjunto de comandos é executado em uma sequência linear de cima para baixo e da esquerda para a direita. Estrutura de seleção simples usada quando precisamos testar uma certa condição antes de executar uma ação. E a estrutura de repetição com variável de controle, a qual repete a execução de um bloco de ações um número predeterminado de vezes (FORBELLONE, 2005).

E o jogo Robot.exe que consiste de seis fases, o qual tem o objetivo de além de treinar a estrutura sequencial, levar o aluno a se familiarizar com o uso de funções e seus parâmetros. O conceito de função é sempre presente na maioria dos programas de computador e o sucesso do seu uso está diretamente ligada ao entendimento e à correta passagem dos seus parâmetros. Este jogo pretende treinar o aluno nesta capacidade.

Acredita-se que a proposta destes jogos inéditos se une à iniciativa da comunidade de desenvolvedores de jogos, de introduzir os jogos como uma ferramenta de auxílio interessante e atraente para o ensino de lógica de programação. Deixa-se em aberto a incorporação de outras funcionalidades que visem aprimorar estas ferramentas.

1.1 OBJETIVO

O objetivo deste trabalho é o desenvolvimento de jogos para auxiliarem no aprendizado da lógica básica de programação. Entre histórias e atividades com elementos gráficos 2D e 3D abordam-se estruturas de controle de fluxo e funções com as quais o jogador interage e comanda para obter a solução de desafios variados.

1.1.1 Objetivos específicos

1. Revisão de conceitos ligados aos jogos na computação e trabalhos correlatos;
2. Estudo detalhado da ferramenta Unity;
3. Estudo sobre os conceitos de lógica de programação e a melhor forma de incorporá-los nos jogos;
4. Elaboração do Game Design Document (GDD);
4. Implementação dos jogos.

1.2 Jogos educativos ou *serious games*

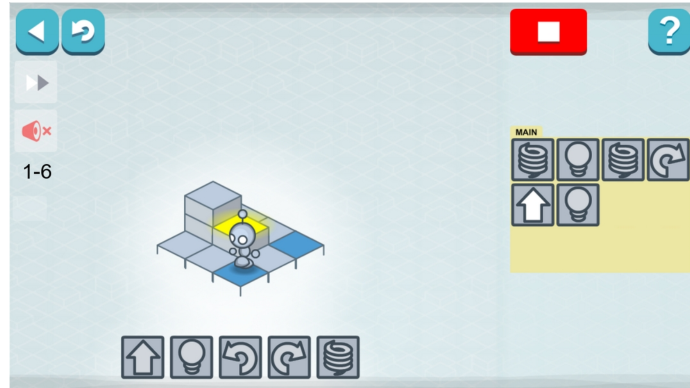
Os jogos educativos são jogos desenvolvidos com o intuito de prover algum conhecimento de aprendizagem. Eles são principalmente utilizados em situações de treinamento de emergência, como treinamentos militares, em escolas, na área de saúde e em vários outros setores da sociedade. (DERRYBERRY, 2008).

1.3 Trabalhos correlatos

Foram observados outros jogos já desenvolvidos, e que possuem uma boa avaliação pela comunidade. Parte de suas jogabilidades inspiraram na criação dos jogos criados pelo autor. Outros jogos foram analisados no trabalho de MONCLAR, mas estes não possuíam uma jogabilidade que atendesse às características necessárias para referência. Dois jogos dos jogos analisados possuem características semelhantes, são eles LIGHTBOT e CODECOMBAT, que serão discutidos a seguir.

LIGHTBOT (Figura 3) é um jogo onde o jogador controla um robô através de algoritmos que ele mesmo constrói, através de ações já predeterminadas pelo sistema. Através desses controles, o usuário aprende como funciona os passos que um algoritmo e função seguem.

Figura 3: Jogo LIGHTBOT



Fonte: Retirada do site oficial da LIGHTBOT *(https://lightbot.com/img/screen_basic.png)*

CODECOMBAT (Figura 4), assim como LIGHTBOT, também ensina como funcionam os algoritmos e sua execução. Porém, diferentemente de LIGHTBOT, ele utiliza linhas de texto em vez de botões (que é como LIGHTBOT funciona). Illuminate também fará uso de linhas de texto para ações do personagem.

Figura 4: Jogo Code Combat

Fonte: Retirada do site Code Combat (*https://br.codecombat.com/images/pages/about/screenshot_forest.png*)

1.4 Algoritmos: Estruturas de Controle de Fluxo de Execução de Comandos e funções

Um algoritmo pode ser definido como uma sequência de passos que visam atingir um objetivo bem definido ou resolver um problema. Já um programa é a codificação de um algoritmo em uma linguagem de programação.

Uma linguagem de programação, portanto, permite a comunicação de instruções para um computador usando variáveis, operadores e comandos básicos e estruturas de controle de fluxo de execução desses comandos. Estas estruturas são: a estrutura sequencial, estrutura de seleção e estrutura de repetição.

No jogo Robot.exe são utilizados os conceitos de chamada de funções com parametrização, que também será explicado nesta seção.

1.4.1 Estrutura sequencial

A estrutura sequencial é a estrutura base que compõe as outras estruturas. Nela, o conjunto de ações primitivas é executada "em uma sequência linear de cima para baixo e da esquerda para a direita, isto é, na mesma ordem em que foram escritas" (FORBELLONE, 2005).

1.4.2 Estrutura de seleção

"Uma estrutura de seleção permite a escolha de um grupo de ações (bloco) a ser executado quando determinadas condições, representadas por expressões lógicas ou relacionais, são ou não satisfeitas" (FORBELLONE, 2005). Ela pode ser separada em categorias: seleção simples, seleção composta e seleção encadeada.

Utiliza-se a seleção simples "quando precisamos testar uma certa condição antes de executar uma ação" (FORBELLONE,2005). Quando a condição, que é dada por uma expressão lógica, for verdadeira, então as ações primitivas localizadas entre a seleção (se) e o fim da seleção (fimse) são executadas, caso contrário nada é executado, encerrando-se a seleção.

```
se (condição)
        C1 //sequência de comandos
        C2 //sequência de comandos
fimse
```

A seleção composta é utilizada “quando tivermos situações em que duas alternativas dependem de uma mesma condição, uma de a condição ser verdadeira e outra de a condição ser falsa” (FORBELLONE, 2005). Quando a condição é verdadeira, então a execução comporta-se como a seleção simples. Caso a condição seja falsa, então executa-se a cláusula do senão.

```
se (condição)
        C1 //sequência de comandos
        C2 //sequência de comandos
        .
        .
        Cn
senão
        C1 //sequência de comandos
        C2 //sequência de comandos
        .
        .
        Cn
fimse
```

Seleção encadeada é utilizada quando “devido à necessidade de processamento, agruparmos várias seleções” (FORBELLONE, 2005). A seleção encadeada, de acordo com Forbellone, pode ser dividida em: seleção encadeada heterogênea e seleção encadeada homogênea, porém será explicado apenas a seleção encadeada homogênea, por ser mais completa em relação a outra.

A seleção encadeada homogênea é utilizada quando “a construção de diversas estruturas de seleção encadeadas seguem um determinado padrão lógico” (FORBELLONE, 2005). Abaixo será representado um exemplo:

```
se (condição 1 e condição 2)
        C1
senão se (condição 3)
        C2
senão
```

C3

fimse

Perceba que entre a condição 1 e a condição 2 o "e" pode ser substituído por um "ou". Pode-se adicionar mais condições em cada seleção, podendo ser verificadas mais facilmente suas validações através da construção de uma tabela verdade. Perceba também que o primeiro "senão" é procedido de um "se", enquanto que o segundo não o possui. Isso ocorre pois a condição do segundo senão é que as outras seleções sejam falsas.

1.4.3 Estrutura de repetição

A estrutura de repetição é utilizada quando precisamos realizar determinadas ações diversas vezes. Utilizando uma ação primitiva como exemplo, se necessitarmos executá-la 100 vezes, em vez de escrevê-la 100 vezes podemos utilizar apenas uma estrutura de repetição, economizando mais tempo, espaço e trabalho. Essa estrutura pode ser dividida em repetição com teste no início, repetição com teste no final e repetição com variável de controle. Porém, é explicado apenas a forma como esta estrutura foi aplicada neste trabalho, sendo uma forma simplificada da estrutura de repetição com variável de controle.

Para facilitação do entendimento do jogador em relação a esta estrutura, foi utilizada a palavra "repita", que é precedida por um número, o qual determina o número de vezes que seu bloco será executado. O término de seu bloco é determinado pelo "fimrepita". Um exemplo de seu uso será mostrado logo abaixo:

```
repita(N)
    C1
    .
    .
    Cn
fimrepita
```

Neste trabalho, não foram implementados estruturas de fluxo aninhadas (por exemplo estrutura de repetição dentro de outra estrutura de repetição), nem combinação de estruturas (por exemplo

estrutura condicional dentro de estrutura de repetição), porém, tal material pode ser implementado em trabalhos futuros.

1.4.4 Chamada de funções

Em linguagem de programação, quando é realizada a chamada de uma função, tal função é responsável por realizar uma determinada ação, e a forma dessa ação ser realizada pode ser controlada através de seus parâmetros. A forma como isso é representado nos jogos desenvolvidos neste trabalho é nomeando as funções com o nome das ações que elas realizam. A distância com que o personagem anda, por exemplo, pode ser determinada através de seu parâmetro, como mostrado a seguir.

andar(nome do personagem, distância)

O “nome do personagem” e a “distância” são parâmetros, que podem ser alterados para realizar resultados diferentes. Alterando o nome do personagem para “robot” e a distância para “10”, a chamada da função pode ser representado da seguinte maneira:

andar(robot, 10)

E é desta forma que o jogo realiza a associação da chamada de funções com as ações do jogo.

1.5 Ferramentas utilizadas

A principal ferramenta utilizada para o desenvolvimento dos dois jogos deste trabalho é a Unity 3D. A Unity é uma ferramenta muito famosa pois ela possui todo o seu funcionamento documentado em um manual online, além de ela ser gratuita.

Vários jogos famosos foram desenvolvidos utilizando esta ferramenta, como HEARTHSTONE (desenvolvido pela Blizzard Entertainment) e CUPHEAD (desenvolvido pelo Studio MDHR). Sendo um motor de jogos, ela é responsável por unir animação, modelagem e codificação, fornecendo suporte para várias plataformas (Sistema operacional Windows, Linux, macOS, Android entre outros). Ela também fornece facilidades para o desenvolvimento de jogos tanto 2D quanto 3D, graças a sua variedade de funcionalidades já implementadas em suas bibliotecas, agilizando o processo de desenvolvimento.

No decorrer deste trabalho foram utilizadas as versões mais recentes na época em que foram iniciadas, não sendo alteradas suas versões para não haver instabilidades. A versão da Unity utilizada para o Robot.exe foi a versão 2018.3.12 e no Illuminate foi utilizada a versão 2019.2.6.

A linguagem C Sharp é uma das duas linguagens de programação suportadas pela Unity, tendo algumas semelhanças com as linguagens C++ e JAVA. “Em comparação com o C++, o C# é mais fácil de aprender. Além disso, é uma linguagem gerenciada, o que significa que faz o gerenciamento de memória automaticamente para o usuário.” (UNITY, 2019).

Com a programação, o desenvolvedor pode controlar todos os objetos e suas propriedades. Com *scripts*, ele pode “implementar sua própria lógica e comportamento de jogo simplesmente adicionando-os aos objetos do jogo”. (UNITY, 2019)

A Blender é uma ferramenta de criação 3D gratuita e de código aberto. Suporta todo conteúdo de animação e modelagem 3D. Usuários avançados utilizam a linguagem Python para customizar a aplicação e criar ferramentas especializadas. Existem outras ferramentas de modelagem 3D, como o Maya, mas que são pagos. O Blender apesar de ser uma ferramenta gratuita, possui funcionalidades tão boas quanto qualquer outra ferramenta profissional de modelagem 3D.

As ferramentas Aseprite e Inkscape são ferramentas de uso artístico. O Aseprite é pago; é utilizado para criação de artes em nível de pixel (baixa escala), além de possibilitar a criação de animações nesse formato. Por outro lado, o Inkscape é uma ferramenta gratuita e de código-fonte aberta, porém, é um editor de gráficos vetoriais, disponível para Linux, Windows e macOS.

1.6 Metodologia

O projeto foi desenvolvido em quatro etapas. Em cada etapa foram feitas reuniões periódicas com a orientadora de forma a acompanhar e avaliar o andamento da pesquisa. A primeira etapa constou na revisão bibliográfica, na qual foram pesquisados artigos sobre jogos existentes e ferramentas para desenvolvimento de jogos.

Na segunda etapa foi trabalhada a idealização do jogo, que no conceito de jogos essa prática é conhecida como *Game Design Document* (GDD). O GDD é uma documentação do trabalho que faz uma previsão de como o jogo será, sendo uma base para o desenvolvimento do jogo. Por isso, o produto final do jogo pode ser diferente do que foi idealizado no começo. No GDD são descritos mecânicas do jogo, sons, artes, história, público-alvo, cronograma, transições e fluxos do jogo.

Na terceira etapa foi realizada a implementação de cada elemento do jogo, mostrando como estão feitos e os funcionamentos detalhados de determinados objetos do jogo.

A quarta e última etapa constou na realização de testes e *feedback* do público-alvo, verificando a receptividade e aprovação do público-alvo. Os testes consistiram em formulários online, que foram respondidos após o teste do jogo.

1.7 Organização dos capítulos

O Capítulo 2 apresenta o GDD dos dois jogos, mostrando cada detalhe dos conceitos do jogo; o Capítulo 3 apresentara a implementação dos dois jogos, na linguagem C sharp; o Capítulo 4 mostra testes e *feedback* do público, sendo mostrados gráficos para melhor visualização; ao final, no Capítulo 5, são mostradas as conclusões sobre o trabalho.

2. Game Design Document (GDD)

O GDD é o documento que detalha como será feito o jogo. É uma base para a produção dele, o projeto final não obrigatoriamente necessita ser igual ao que foi definido neste documento. Ele é relacionado com o design do jogo e suas mecânicas, é através dele que o designer do jogo comunica-se com a equipe de desenvolvimento. Por isso, os desenvolvedores do jogo precisam estar sempre atentos às mudanças no GDD (MOTTA, 2013).

Esse documento deve ser escrito antes mesmo do desenvolvimento do projeto. Ele não possui um padrão a ser seguido.

Há casos em que a produção do jogo é feita apenas por um desenvolvedor, mas, mesmo assim, é importante que tenha este documento feito para que o próprio desenvolvedor tenha suas ideias organizadas e que possa servir de apresentação caso alguém interesse-se por ele, como alguém querendo financiar o projeto ou alguém querendo juntar-se à equipe.

Foram criados dois GDDs, uma seção para cada jogo. Cada uma dessas seções são divididas em três partes: conceitos do jogo, proposta e projeto.

2.1 GDD do jogo Illuminate

O primeiro GDD apresentado será o Illuminate, que é desenvolvido pelo autor.

2.1.1 Conceitos do jogo

Illuminate é um jogo simples e pequeno, sem um roteiro muito detalhado. Illuminate faz uso de caixa de texto, pois comporta mais de uma linha em sua estrutura de entrada.

2.1.1.1 Características principais

Single-player: não fornecerá suporte *multiplayer* ou cooperativo, sendo apenas para um jogador.

Gráficos em 3D: Todos os componentes que não são de interface gráfica serão compostos por objetos 3D. Foi utilizada a ferramenta Blender para modelagem 3D dos objetos.

2.1.1.2 Gênero

Illuminate é um jogo de puzzle, mas utiliza-se de apenas uma única fase para mostrar todas as suas funcionalidades.

2.1.1.3 Plataforma

O jogo está desenvolvido para PC's (*Personal Computer*). Para jogar é necessário apenas o download do mesmo. Ele possui suporte para Linux, Windows e MacOS, pois a Unity possibilita, de acordo com a vontade do jogador, a geração dos executáveis para esses sistemas operacionais.

2.1.2 Proposta do jogo

Illuminate introduz o jogador a um ambiente de programação. Também possui a característica de dar o entendimento da estrutura sequencial ao jogador.
O jogo faz o usuário utilizar estruturas de repetição e estruturas seletivas simples.

2.1.2.1 Público alvo

O jogo é voltado para iniciantes na área de programação e até mesmo para quem não possui experiência alguma na área da programação.

2.1.2.2 Comparação de características

No desenvolvimento das características de Illuminate foram observadas funcionalidades semelhantes em jogos como Lightbot e Code Combat. Detalhes sobre estes jogos podem ser observados na seção 1.4 de trabalhos correlatos.

2.2.2.3 Recursos estimados (orçamento)

Os recursos estimados para o desenvolvimento de Illuminate foram:

1 Desenvolvedor.

Softwares: Unity; Blender.

Hardware a ser utilizado:

o i7 oitava geração

o 8GB RAM

o Windows 10

o Placa de vídeo GeForce MX150

2.1.3 Projeto do jogo

Define os elementos do jogo de forma detalhada, servindo para base de produção.

O jogo tem uma proposta de introduzir ao jogador a estrutura de seleção simples e a estrutura de repetição com variável de controle. A estrutura de seleção simples é "quando precisamos testar uma certa condição antes de executar uma ação" (FORBELLONE, 2005). A estrutura de seleção simples não faz o uso do "senão" no contexto utilizado pelo jogo.

A estrutura de repetição com variável de controle "repete a execução do bloco um número predeterminado de vezes, pois ela não prevê uma condição e possui limites fixos" (FORBELLONE, 2005). Na Figura 5 está representada uma imagem em jogo de Illuminate.

Figura 5: Em jogo

Fonte: Imagem retirada diretamente do programa.

2.1.3.1 Elaboração do jogo

O jogo será desenvolvido com o uso da ferramenta Unity versão 2019.2.6, utilizando a linguagem C#.

2.1.3.2 Mecânica do jogo

Illuminate é um jogo puzzle, em que o jogador deve entrar com uma sequência de comandos em uma caixa de texto que será lida pelo jogo. O jogador pode executar essa sequência apenas uma vez, caso contrário deverá reiniciar a cena.

2.1.3.3 Fluxo

O jogo inicia-se em seu logo, apresentando o título do jogo (Figura 6). Após isso o jogador é levado a um menu que contém duas opções: jogar e sair (Figura 7). Caso o jogador escolha jogar, ele é levado à única fase que o jogo possui; caso escolha sair, o jogo é fechado.

Figura 6: Tela inicial

Fonte: Imagem retirada diretamente do programa

Figura 7: Tela de menu

Fonte: Imagem retirada diretamente do programa

2.1.3.4 Personagens

O único personagem do jogo é o personagem principal, ele é uma luminária (Figura 8) que pula para locomover-se. O personagem é inspirado na luminária que aparece na introdução de filmes da Pixar.

Figura 8: Personagem de Illuminate

Fonte: Imagem retirada diretamente do programa

2.1.3.5 Elementos do *gameplay*

Há 5 elementos de interface de usuário (UI) que o jogador pode interagir, os botões de: ajuda, comandos, menu, caixa de texto e botão compilar.

O botão de ajuda mostra informações úteis que podem sanar as dúvidas do jogador; o botão de comandos lista todos os comandos possíveis e como utilizá-los; o botão de menu abre uma janela que dá as opções de sair do jogo ou voltar ao jogo; a caixa de texto é onde o jogador entra com os comandos de ações do jogo; o botão compilar executa todos os comandos digitados na caixa de texto.

2.1.3.6 História

Illuminate não possui um roteiro de história. Porém o ambiente do jogo consiste na coleta de lâmpadas que permitirão a iluminação de um farol, que é o objetivo final do mapa. Esse farol só é aceso quando todas as lâmpadas são coletadas.

2.1.3.7 Efeitos sonoros

Os efeitos sonoros, assim como o jogo Robot.exe foram retirados do site Freesound

2.1.3.8 Artes

A única parte artística do jogo é referente à modelagem 3D, que foi toda feita no Blender.

2.2 GDD do jogo [Robot.exe]

O segundo GDD apresentado é do jogo Robot.exe.

2.2.1 Conceitos do jogo

Nesta etapa, introduzimos a ideia central do jogo.

[Robot.exe] é um jogo da categoria puzzle, que será desenvolvido utilizando a linguagem C# na Unity, para a plataforma PC. O jogo consiste em utilizar uma caixa de texto que recebe entradas para controlar os personagens ou parte do cenário dentro do jogo.

Sendo um jogo puzzle, o jogador deve desvendar quebra-cabeças e formas de completar cada fase.

2.2.1.1 Características principais

Single-player: não fornece suporte *multiplayer* ou cooperativo, sendo apenas para um jogador.

Gráficos em 2.5D: Utiliza sprites (2D) para o gráfico dos personagens e objetos 3D para elementos do mapa, gerando o que chama-se gráficos 2.5D.

As ferramentas Aseprite e Inkscape foram utilizadas para auxiliarem na parte artística do jogo.

2.2.1.2 Gênero

Robot.exe é um jogo no estilo puzzle, onde o jogador deverá desvendar mistérios ou quebra-cabeças para conseguir concluir o cenário. As entradas de texto para movimentação dos personagens ou cenários serão fundamentais para a conclusão do cenário, além de elas implicitamente auxiliar no aprendizado de execução de funções de programação e introduzir o jogador ao ambiente de programação.

2.2.1.3 Plataforma

O jogo é desenvolvido para PC, como visto anteriormente. Para jogar é necessário apenas o download do mesmo. É fornecido suporte para MacOS, Windows e Linux, pois a Unity disponibiliza a geração de executáveis automaticamente para esses sistemas operacionais.

2.2.2 Proposta do jogo

Introduzir o jogador ao ambiente de programação e dar uma leve introdução ao conceito de estrutura sequencial e execução de funções de programação. Na estrutura sequencial um “conjunto de ações primitivas será executado em uma sequência linear de cima para baixo e da esquerda para a direita” (FORBELLONE,2005).

2.2.2.1 Público alvo

O jogo Robot.exe é voltado para iniciantes na área de programação e até mesmo para quem não possui experiência alguma na área da programação.

2.2.2.2 Comparação de características

As comparações de características foram as mesmas aplicadas ao jogo Illuminate na seção 2.1.2.2.

2.2.2.3 Recursos estimados (orçamento)

Os recursos estimados para o desenvolvimento de Robot.exe são:

1 Desenvolvedor.

Software: Sistema Operacional Windows 7 ou superior; Unity; Aseprite.

Hardware a ser utilizado:

o i5-3317U 1.70GHz

o 8GB RAM

o Windows 10

o Placa de vídeo Intel HD graphics 4000

2.2.3 Projeto do jogo

Define os elementos do jogo de forma detalhada, servindo para base de produção

2.2.3.1 Elaboração do jogo

O jogo será desenvolvido com o uso da ferramenta Unity versão 2018.3.12, utilizando a linguagem C#.

Para começar um jogo, a Unity requisita o local em que será armazenado e para que plataforma o jogo será desenvolvido, 3D ou 2D. Neste caso, será escolhida a plataforma 2D, pois as ações têm mais impacto que na plataforma 3D quando os comandos são realizados através de texto.

No desenvolvimento de um jogo devem ser elaborados uma história, efeitos sonoros (inclui trilha sonora), artes (sendo *sprites* ou texturas) e o que vai unir e executar tudo isso, a programação.

2.2.3.2 Mecânica do jogo

Robot.exe é um jogo do estilo puzzle, onde o jogador deve resolver o problema proposto pelo cenário. Esse tipo de jogo faz mais uso do raciocínio lógico ao invés de agilidade ou reflexos. Em Robot.exe o jogador controlará um robô que movimenta-se através de linhas de comando inseridas em uma caixa de texto. A proposta é que a cada cenário a dificuldade para a conclusão da mesma aumente, através de novos elementos incrementados gradativamente.

2.2.3.3 Fluxo

O jogo inicia-se em um menu básico, o qual o jogador pode escolher qualquer fase que quiser, mas tendo consciência que quanto maior o indicador da fase, maior será sua dificuldade; no menu básico também possuirá a opção para Ajuda, que mostrará explicado cada elemento da *Graphical User Interface* (GUI) presente no jogo.

A primeira fase é o tutorial, que introduz os comandos básicos ao jogador.

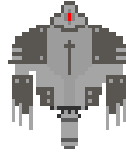
Após escolhido a fase, a fase correspondente é iniciada. Ao concluir a fase, é apresentado um menu mostrando a pontuação do jogador. Em seguida o jogador é direcionado à próxima fase.

2.2.3.4 Personagens

O jogador controlará um robô. Conforme a dificuldade do jogo aumenta, são acrescentados personagens, tendo em mente que o jogador pode mover apenas um personagem por vez.

O personagem principal é Robot, representado na Figura 9, o primeiro que o jogador controlará no início do jogo. Ele não possui nenhuma habilidade ou particularidade especial além de andar, pular, descer e ativar botões.

Figura 9: Personagem Robot



Fonte: Imagem capturada diretamente do programa

Na fase 3 o jogador conhece a personagem Formiga, representado na Figura 10, que possui como característica a habilidade de encolher-se e passar por lugares que outros personagens não conseguem.

Figura 10: Personagem Formiga

Fonte: Imagem capturada diretamente do programa

Na fase 5 o jogador conhece o personagem Lutador, representado na Figura 11, que possui como característica a habilidade de destruir objetos e de criar um escudo que protege-o contra projéteis.
Na última fase são apresentados o hacker e o criador de Robot

Figura 11: Personagem Lutador

Fonte: Imagem capturada diretamente do programa

2.2.3.5 Elementos do *gameplay*

Haverá 3 elementos que o jogador poderá interagir dentro do jogo, sendo eles menu, ajuda e a caixa de texto.

O menu será um botão que mostrará alguns elementos, como voltar ao jogo, recomeçar o jogo e voltar ao menu inicial.

O botão de ajuda irá mostrar elementos novos ou específicos do cenário, que não foram apresentados até o progresso atual do jogador.

A caixa de texto será uma área que o jogador clica sobre ela, podendo entrar com strings que possibilitam a interação com o personagem ou elementos do cenário.

Nessa entrada de texto existem comandos que seguem um padrão, com uma estrutura que possui o propósito de ensinar a execução das funções de programação. Esse padrão é composto pelo primeiro elemento, a função responsável por realizar a ação. Esse primeiro elemento pode ser dentre as opções básicas: pular, descer, andar, atacar e ativar ou as opções que são adicionadas com o decorrer do jogo, sendo: atacar, defender, encolher e crescer. O segundo elemento é composto por parâmetros, que são separados por vírgulas. Esses parâmetros podem ser o nome dos personagens, um valor numérico real ou inteiro, ou nome de botões (que pode ser visualizado no jogo).

2.2.3.6 História

No início do jogo é apresentado um jornal que destaca duas imagens, uma mostrando que ocorreu um assassinato e outra mostrando a retomada de investigações no laboratório onde ocorreu o assassinato.

O jogador é introduzido como um hacker, que invadiu o sistema do laboratório abandonado mencionado no jornal e encontrou um robô. O hacker liga o robô, o qual tem danificado grande parte dos seus dados. O robô está com o seu sistema de movimentação falho, então o jogador (o hacker) deve ajudar a movimentar ele através dos comandos de texto.

Na primeira conversa entre o hacker e o robô, o qual o nome é Robot, Robot menciona que não se lembra de nada e que não conhece o hacker, então o hacker diz para ele sair de lá pois o laboratório logo seria demolido.

Durante o percurso Robot conhece dois personagens, Formiga e Lutador, que ajudam-no na conclusão das fases até sair do laboratório. Na penúltima fase Formiga despede-se dos dois robôs e então eles prosseguem para a fase final, enfrentando complicados obstáculos.

Após a conclusão da última fase, na cena final, os dois robôs saem do laboratório e o hacker aparece dando os parabéns por eles terem conseguido sair de lá. O robô Lutador então escaneia o hacker e seu computador, percebendo que o hacker havia invadido o laboratório no mesmo dia que havia acontecido o assassinato. Após a revelação, Robot começa a lembrar-se do que aconteceu antes dele ser ligado, que o hacker havia hackeado-o e forçado-o a matar o dono do laboratório, que era o seu criador.

O hacker revela que havia feito tudo isso pois ele trabalhava no laboratório e vendia armas ilegais para contrabandistas de armas e pessoas mal intencionadas, e o dono do laboratório tinha ciência disso.

Após todo esse espetáculo, é mostrado através de texto que o hacker foi preso e que Robot e Lutador foram transferidos a um instituto tecnológico para auxiliar no desenvolvimento da tecnologia.

2.2.3.7 Efeitos sonoros

Para os efeitos sonoros serão utilizados sons gratuitos já existentes.

Os efeitos sonoros serão aplicados em objetos destrutíveis e botões. Alguns cenários possuem sons de fundo.

Eles são todos retirados do site https://freesound.org

2.2.3.8 Artes

As artes seguirão o padrão de pixel art. Elas serão produzidas através de softwares que fazem o melhor uso delas, o Aseprite, que é um software para computador designado especificamente para a criação de pixel art.

Os objetos 3D dos cenários utilizam texturas adquiridas em Assets (componentes de auxílio) da própria Unity.

3. IMPLEMENTAÇÃO

A implementação dos jogos foram ambas feitas na Unity, em suas respectivas versões (nas seções 2.1.3.1 e 2.2.3.1), utilizando a linguagem C# (C sharp). Na interface da Unity há a interação com os *scripts*, que por padrão são abertos e editados no Microsoft Visual Studio.

Dentro da Unity há 6 itens que precisam de mais atenção: os *GameObjects*, os componentes dos *GameObjects*, os métodos *start* e *update*, os *assets* e *prefabs*, e as *scenes.*

Os *GameObjects* são os objetos da Unity em si. Eles servem como base na construção de uma hierarquia de componentes.

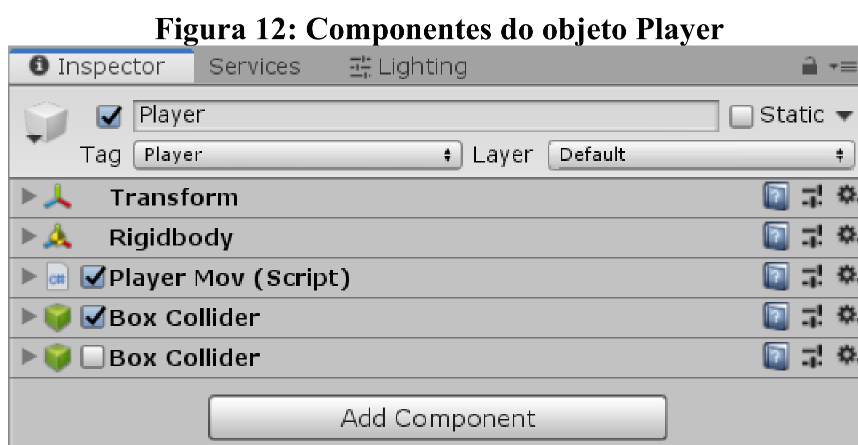
Os componentes são pertencentes aos *GameObjects* e eles são responsáveis por definir as características de um *GameObject.* Um *GameObject* pode possuir vários componentes, desses, três componentes são importantes serem citados: *Camera*, *Transform* e *Script.*

O componente *Transform* é um componente crítico que todo *GameObject* necessita ter. Através dele, é definido a posição, rotação e escala do *GameObject* em relação ao mundo do jogo.

É necessário ter ao menos um *GameObject* com o componente *Camera* no mundo do jogo, para que a parte gráfica seja possível de visualização. Por padrão, na criação de um jogo já vem um *GameObject* intitulado *Main Camera* que possui tais componentes.

O componente *Script* é um componente opcional, mas que é muito importante para que as propriedades dos componentes possam ser manuseadas, ou que novas ações sejam implementadas. É nele que se localiza as codificações de um jogo. Os *scripts* podem controlar os componentes que estava anteriormente inserido em um objeto, assim como eles mesmos podem criar componentes através da codificação. Na Figura 12 um exemplo dos componentes do *GameObject* “Player”:

Figura 12: Componentes do objeto Player



Fonte: Imagem retirada diretamente do programa

Os itens "Transform", "Rigidbody", "Player Mov", "Box Collider" são todos componentes pertencentes ao *GameObject* Player.

A seguir será discutido sobre a implementação de cada jogo, dividindo esta seção em duas partes, uma para cada jogo.

A estrutura básica dos *scripts* na Unity serão descritas a seguir.

- Em suas primeiras linhas são identificadas as bibliotecas que serão usadas:

Figura 13: Bibliotecas

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

Fonte: Imagem retirada diretamente do programa

As bibliotecas mostradas na Figura 13 são as bibliotecas básicas que a Unity faz uso em todos os seus *scripts*.

- Todo *script* da Unity deve derivar da classe base *Monobehaviour* (Figura 14):

Figura 14: Classe Monobehaviour

```
public class Nome_do_Script : MonoBehaviour
```

Fonte: Imagem retirada diretamente do programa

A classe Monobehaviour disponibiliza o uso de vários métodos, que agilizam o processo de codificação do jogo. Duas delas são as mais utilizadas: os métodos *Start* e *Update*. O método *Start* inicializa seu bloco antes do primeiro frame quando o script é iniciado. O método Update executa seu bloco frame-a-frame, rodando a todo o momento, como pode ser visto na Figura 15:

Figura 15: Funções Start e Update

```
// Start is called before the first frame update
void Start()
{

}

// Update is called once per frame
void Update()
{

}
```

Fonte: Imagem retirada diretamente do programa

Há também outros métodoss como o Awake, que executa antes mesmo do método Start; o OnTriggerEnter, que executa quando um colisor entra em contato; o OnTriggerExit, que executa quando um colisor sai do contato; entre vários outros.

O desenvolvedor de um jogo pode utilizar arquivos externos para a Unity, esses são intitulados de *Assets*. Os *Assets* podem ser qualquer tipo de arquivo, um modelo 3D, um arquivo de áudio, imagem, ou qualquer outro tipo de arquivo que a Unity suporte. Muitos desses *Assets* são encontrados à venda na *Internet*, agilizando o processo de desenvolvimento de um jogo. Um exemplo muito claro de agilização é o uso de algum *Asset* que implemente automaticamente a janela de bate papo de um jogo, em que apenas algumas pequenas alterações necessitam serem feitas.

A agilização no desenvolvimento de um jogo também pode ser feita através dos *prefabs*, esses são responsáveis por criar cópias de *GameObjects,* uma cópia de *prefab* pode ser instanciada quando um novo *GameObject* é adicionado a cena, ao invés do desenvolvedor ter que criar um *GameObject* e inserir manualmente cada componente, ele pode estar apenas instanciando o *prefab* para fazer o reuso do *GameObject.* A maioria dos *Assets* vêm com *prefabs*, sendo esses a base da agilidade no desenvolvimento de jogos na Unity.

Um último item da Unity, mas que também é de extrema importância são as *Scenes*, as *Scenes* são importantíssimas pois sem elas os *GameObjects* não existem, pois são nelas que eles são inseridos. Todo o conteúdo de um jogo que pode ser observado por um jogador está inserido nas *Scenes*. Cada *Scene* representa uma fase ou cenário do jogo.

3.1 Implementação do jogo Illuminate

A seguir será apresentado a implementação do segundo jogo: Illuminate.

O jogo pode ser baixado através do link https://kkkkenzo.itch.io/illuminate.

3.1.1 Caixa de texto para entrada de dados

A caixa de texto para entrada de dados é primariamente um componente *TextMeshPro – InputField* (literalmente a própria caixa de texto de entrada de dados), o qual é composto por um *Text Area* (que define a área de digitação), que é composto por um *Placeholder* e um *Text*. Desses componentes, apenas o *TextMeshPro – InputField* e o *Text* (o texto digitado) são mais importantes. Eles são editados em um script chamado “Compilar_v3” que é ativado ao clicar no botão “Compile!”.

A caixa de texto comportas várias linhas, sendo lidos pelo script quando clicado no botão. Um vetor de strings chamada “linhas” recebe o TextMeshPro – InputField separado por cada enter, que é lida pelo “\n”, através da função “string.Split(caracter)”, como pode ser visto na Figura 16:

Figura 16: Uso da função Split, no script Compilar_v3

```
linhas = textBox.text.Split('\n');
```

Fonte: Imagem capturada diretamente do programa

Nessa caixa de texto são guardados comandos entrados pelo jogador, que são responsáveis por mover o personagem.

Esses comandos são lidos na função “Leitura_Comandos(inteiro)”, que é responsável por fazer a verificação se os textos entrados pelo jogador são realmente comandos. Caso uma linha de texto não seja um comando, é apresentado uma mensagem de erro ao jogador, caso contrário a variável de vetor de string “sequence_commands” recebe a linha de texto. É importante lembrar que antes dessa leitura é feito o tratamento para letras minúsculas através da função “Tratamento_Minusc()”. Essa conversão é feita através da função “ToLower()”, aplicada em cada linha de comando (Figura 17).

Figura 17: Função Tratamento_Minusc() do script Compilar_v3

```
public void Tratamento_Minusc()
{
    for (int j = 0; j < linhas.Length; j++)
    {
        linhas[j] = linhas[j].ToLower();
    }
}
```

Fonte: Imagem capturada diretamente do programa

3.1.2 Comandos do usuário

Pode-se dizer que a parte mais importante do programa está neste item, embora os outros também tenha uma importante relevância ao jogo.

Os comandos, como foi visto no item anterior, foram passados à caixa de texto e então tratados. Após o sucesso de todo o processo, é feito o processo de execução desses comandos, que são executados em uma sub-rotina nomeada de "_execute". Todo a codificação fica dentro de um laço de repetição que utiliza numLinha como variável de controle, que é incrementada até ler todos os comandos em sequence_commands (Figura 18).

Figura 18: Laço de repetição da sub-rotina execute() do script Compilar_v3

```
for(int numLinha = 0; numLinha < sequence_commands.Length - 1; numLinha++)
```

Fonte: Imagem capturada diretamente do programa

Caso alguma linha tenha conteúdo nulo, o laço de repetição é quebrado.

A movimentação dos personagens é feita em um *script* separado, que é um componente do personagem e é nomeado de "PlayerMov". Ou seja, a função _execute faz a requisição da ação de movimento do personagem para o script PlayerMov.

3.1.2.1 Comando de ação andar

O primeiro dos 4 comandos básicos, o andar, é utilizado para o personagem mover-se para a frente. Na função _execute, quando chama a função andar, também é chamado a animação de movimentação do personagem. Já na função de ação do personagem, no script PlayerMov, como citado anteriormente, é verificado se tem algum objeto à sua frente através de um sensor, caso

tenha, verifica se esse objeto tem a tag “Wall”. Caso não tenha, é chamado a função “Rigidbody.AddRelativeForce(Vector3)”, que aplica uma força no personagem nas coordenadas y e z. Essa força é definida como uma variável chamada “jumpForce”, que é definida pelo criador do jogo. Caso o objeto à frente do personagem contenha a tag “Wall”, a força é aplica apenas no eixo y, dando a sensação de bloqueio de movimento causado pelo obstáculo à sua frente (Figura 19).

Figura 19: Código da função de ação andar do script PlayerMov

```
public void andar()
{
    if (objFrente.hasObjInside)
    {
        if (objFrente.objectInside.tag != "Wall")
        {
            rgbdPlayer.AddRelativeForce(new Vector3(0, jumpForce, jumpForce));
        }
        else
        {
            rgbdPlayer.AddRelativeForce(new Vector3(0, jumpForce, 0));
        }
    }
    else
    {
        rgbdPlayer.AddRelativeForce(new Vector3(0, jumpForce, jumpForce));
    }
}
```

Fonte: Imagem capturada diretamente do programa

3.1.2.2 Comando de ação voltar

Semelhante ao comando de andar, sua única diferença é que a verificação é feita em um sensor localizado atrás do personagem, e que a força aplicada em z durante o movimento é negativo em vez de positivo (Figura 20).

Figura 20: Código do comando de ação voltar, contido no script PlayerMov

```
public void voltar()
{
    if (objAtras.hasObjInside)
    {
        if (objAtras.objectInside.tag != "Wall")
        {
            rgbdPlayer.AddRelativeForce(new Vector3(0, jumpForce, -jumpForce));
        }
        else
        {
            rgbdPlayer.AddRelativeForce(new Vector3(0, jumpForce, 0));
        }
    }
    else
    {
        rgbdPlayer.AddRelativeForce(new Vector3(0, jumpForce, -jumpForce));
    }
}
```

Fonte: Imagem capturada diretamente do programa

3.1.2.3 Comando de ação esquerda

Quando o comando esquerda é lido na caixa de texto, a função _execute chama a função "viraEsquerda()", que chama a função "Rotate" que aplica uma rotação no personagem de 90 graus negativos. (Figura 21)

Figura 21: Código da função viraEsquerda(), contido no script PlayerMov

```
public void viraEsquerda()
{
    endRotation.transform.Rotate(Vector3.up, -90, Space.World);
}
```

Fonte: Imagem capturada diretamente do programa

Apenas esta função não faz todo o processo de rotação do personagem, também é necessário que na função "Update", que atualiza frame-a-frame, seja atualizado a rotação do personagem , através da função "Quaternion.Lerp", que interpola a rotação inicial do personagem com a rotação final, criando uma animação fluída. O último parâmetro garante que a animação seja realizada em função do tempo (Figura 22).

Figura 22: Código da atualização da rotação, contido no script PlayerMov

```
void Update()
{
    this.transform.rotation = Quaternion.Lerp(this.transform.rotation,
        endRotation.transform.rotation, Time.deltaTime * speed);
}
```

Fonte: Imagem capturada diretamente do programa

3.1.2.4 Comando de ação direita

Semelhante à ação de esquerda, mas o nome de sua função é "viraDireita" e o grau de rotação é 90 graus positivos (Figura 23).

Figura 23: Código da função viraDireita(), no script PlayerMov

```
public void viraDireita()
{
    endRotation.transform.Rotate(Vector3.up, 90, Space.World);
}
```

Fonte: Imagem capturada diretamente do programa

3.1.2.5 Comando se

Quando um comando começa com "se(", ele realiza os passos da estrutura se. Nessa estrutura são definidos respectivamente início e término da estrutura, sendo "se(" e "fimse". O início da estrutura é guardado na variável inteira startIndexSe, que recebe a linha atual.

Para encontrar a posição do término da estrutura, procura-se o primeiro "fimse" criando uma estrutura de repetição que inicia-se na posição do startIndexSe (Figura 24).

Figura 24: Código que define a posição do fim se, no script Compilar_v3

```
for (int kk = startIndexSe; kk < sequence_commands.Length; kk++)
{
    if(sequence_commands[kk] == null)
    {
        //donothing
        break;
    }
    else if (!sequence_commands[kk].StartsWith("fimse"))
    {
        //donothing
        //break;
    }
    else
    {
        endIndexSe = kk;
        contemFimSe = true;
        break;
    }
}
```

Fonte: Imagem capturada diretamente do programa

Em seguida é verificado se o comando possui "==" (igual) ou "!=" (diferente), caso não tenha, é dado um erro (Figura 25).

Figura 25: Verificação do == ou != , no script Compilar_v3

```
if (sequence_commands[numLinha].Contains("=="))...
else if (sequence_commands[numLinha].Contains("!="))...
else
{
    Debug.Log("ERRO");
    Erro();
    break;
}
}
```

Fonte: Imagem capturada diretamente do programa

Após a verificação, são guardados em um vetor de duas posições de *string*, os dois parâmetros que serão comparados. Essa *string* é chamada de "conds" e a função de "Condicional" (Figura 26).

Figura 26: Chamada da função Condicional

```
Condicional(conds, "==", numLinha);
```

Fonte: Imagem capturada diretamente do programa

A função Condicional é responsável por encontrar quais são as condicionais a serem comparadas, para isso, primeiro a função define a primeira condicional, pegando a posição zero, através da função “IndexOf” e localizando onde termina lendo a posição do símbolo de comparação dado (igual ou diferente) menos 1. O início da segunda condicional a ser analisada é pegando a posição do símbolo de comparação e somando com o tamanho do símbolo mais 1. O tamanho desse símbolo é encontrado através da função “Length”. Após atribuídos os valores, eles são retornados (Figura 27).

Figura 27: Função Condicional, no script Compilar_v3

```
public string[] Condicional(string[] condicionais, string etcetc, int j)
{
    int startIndex = linhas[j].IndexOf("(");
    int endIndex = linhas[j].IndexOf(")");

    string cond1;
    int cond1StartIndex = 0;
    int cond1EndIndex;
    cond1StartIndex = startIndex + 1;
    cond1EndIndex = linhas[j].IndexOf(etcetc) - 1;
    cond1 = linhas[j].Substring(cond1StartIndex, cond1EndIndex - cond1StartIndex);
    cond1.Trim();
    condicionais[0] = cond1;

    string cond2;
    int cond2StartIndex;
    int cond2EndIndex;
    cond2StartIndex = linhas[j].IndexOf(etcetc) + etcetc.Length + 1;
    cond2EndIndex = endIndex;
    cond2 = linhas[j].Substring(cond2StartIndex, cond2EndIndex - cond2StartIndex);
    cond2.Trim();
    condicionais[1] = cond2;

    return condicionais;
}
```

Fonte: Imagem capturada diretamente do programa

Após adquiridos os nomes dos objetos a serem comparados, é verificado se existem objetos com esses nomes no jogo, através da função “VerifSeExistem”, demonstrado na Figura 28.

Figura 28: Função VerifSeExistem, no script Compilar_v3

```
public bool VerifSeExistem(string[] listaObjs)
{
    print(listaObjs[0]);
    print(listaObjs[1]);
    if (GameObject.Find(listaObjs[0]) && GameObject.Find(listaObjs[1]))
    {
        return true;
    }
    else
    {
        print("Não existem");
        Erro();
        return false;
    }
}
```

Fonte: Imagem capturada diretamente do programa

Depois de verificados, também é verificado se o nome de um dos dois objetos comparados possuem o nome de “sensor_dentro”, “sensor_atras” ou “sensor_frente”. Esses sensores são responsáveis por verificarem qual objeto está dentro, atrás ou na frente, respectivamente. Caso o tenham, é verificado se o objeto contido no sensor verificado corresponde ao nome do outro objeto, o que significa que a condição pode ser verificada. No caso, se o símbolo de verificação for ==, se os dois objetos forem diferentes, então a linha atual que está sendo lida é enviada para o final do se (localizado no fimse) mais uma posição. Caso o símbolo de verificação seja !=, a linha atual só é enviada para o final do “se” se os dois objetos forem iguais (Figura 29).

Figura 29: Verificação do "se", no script Compilar_v3

```
objeto1 = GameObject.Find(conds[0]);
print("obj1: "+ objeto1);
objeto2 = GameObject.Find(conds[1]);
print("obj2: " + objeto2);
if (objeto1.name == "sensor_frente" || objeto1.name == "sensor_atras" || objeto1.name == "sensor_dentro" ||
    objeto2.name == "sensor_frente" || objeto2.name == "sensor_atras" || objeto2.name == "sensor_dentro")
{
    if (!contemFimSe)
    {
        print("Nao contem fimse");
        Erro();
    }
    if (objeto1.GetComponent<CollisionSensor>().hasObjInside)
    {
        print("objeto1:" +objeto1.GetComponent<CollisionSensor>().objectInside.name);
        print("objeto2:" + objeto2.name);
        if (objeto1.GetComponent<CollisionSensor>().objectInside.name != objeto2.name)
        {
            numLinha = endIndexSe +1;
        }
    }
    else
    {
        numLinha = endIndexSe +1;
    }
}
else
{
    numLinha = endIndexSe +1;
}
```

Fonte: Imagem retirada diretamente do programa.

3.1.2.6 Comando repita

A estrutura do comando repita começa com “repita(numero_de__loops)” e termina com “fimrepita”. Seu funcionamento é feito dentro da leitura dos comandos, que lê os comandos digitados e envia tudo o que está dentro da estrutura repetidamente igual ao número de repetições escolhida. Todos esses comandos são enviados ao vetor de strings “sequence_commands”, que depois é lida na função “_execute”.

Através da função “Substring” é pego o parâmero “numero_de_loops”, localizando onde começa o parênteses e onde ele termina, através da função “IndexOf”. Após isso a variável inteira “numLoops” recebe a string “paramNumLoops” convertida em inteiro, feita através do “int.Parse” (Figura 30).

Figura 30: Lendo o valor entre parênteses do comando repita

```
int startIndex = linhas[j].IndexOf("(");
int endIndex = linhas[j].IndexOf(")");
string paramNumLoops = linhas[j].Substring(startIndex + 1, endIndex - startIndex - 1);
int numLoops = int.Parse(paramNumLoops);
```

Fonte: Imagem retirada diretamente do programa.

Dado o número de repetições (loops), é feita toda a leitura dos comandos básicos do jogo entre o repita e o fimrepita, enviando-os para a variável “sequence_commands”, repetindo enquanto uma variável de controle for menor que o número de repetições, dado pela variável “numLoops” (Figura 31).

Figura 31: Parte principal da estrutura repita

```
for (int z = 0; z < numLoops; z++)
{
    int k = startIndexRepita + 1;
    while (!linhas[k].Contains("fimrepita"))
    {
```

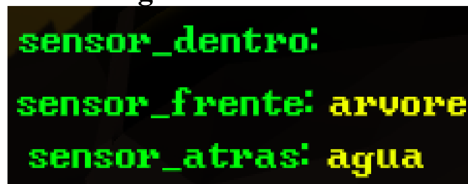
Fonte: imagem retirada diretamente do programa

3.1.2.7 Sensor de colisões

O personagem possui três sensores de colisões, um para identificar qual o objeto à sua frente, outro para o objeto atrás, e um outro para o objeto que está dentro dele. Cada um desses sensores possuem um “Box Collider” com a opção “Is trigger” marcada, que verifica outros “colliders” que encostam nele, marcando o “Is trigger” esse *collider* permite que outros objetos entrem nele.

Os sensores verificam quando um objeto entra em colisão com eles através do “OnTriggerEnter”, e verificam quando um objeto sai da colisão através do “OnTriggerExit”. Através deles é mostrado ao usuário por um “TextMeshProUGUI” quais objetos estão dentro dos sensores (Figura 32).

Figura 32: Sensores



Fonte: imagem retirada diretamente do programa

O parâmetro "Collider other" permite que manipule-se as informações do objeto colisor utilizando a variável "other", assim é possível verificar seu nome ou outras características desejadas. No Update é atualizado as informações para vazio caso não tenha nenhum objeto dentro (Figura 33). Seus comandos estão contidos no script CollisionSensor.

Figura 33: Script do sensor de colisões

```
private void Update()
{
    if(objectInside == null)
    {
        texto.text = "";
        hasObjInside = false;
        objsCounter = 0;
    }
}

private void OnTriggerEnter(Collider other)
{
    objectInside = other.gameObject;
    texto.text = other.gameObject.name;
    hasObjInside = true;
    objsCounter++;
}

private void OnTriggerExit(Collider other)
{
    objsCounter--;
    if (objsCounter == 0)
    {
        objectInside = null;
        texto.text = "";
        hasObjInside = false;
    }
}
```

Fonte: imagem retirada diretamente do programa

3.2 Implementação do jogo Robot.exe

Nessa seção serão apresentados a parte da implementação mais detalhadamente em relação ao GDD.

O jogo pode ser baixado nos links https://kkkkenzo.itch.io/robotexe e https://gamejolt.com/games/robot_exe_br/437192

3.2.1 Caixa de texto para entrada de dados (InputField)

A caixa de texto para entrada de dados é conhecida na Unity como *InputField*, um componente de UI (Interface de usuário; no inglês User Interface).

Sua estrutura fica rodando na função Update, no script principal. A variável timeSinceLastCall é incrementada a todo momento pelo Time.deltaTime, que serve como um contador de segundos.

É feito a verificação se o *InputField* está preenchido, caso esteja e o usuário aperte a tecla Enter e o timeSinceLastCall seja maior que 0.75 segundos, então é transformado todo o texto para minúsculo; chama-se a função leitura_texto(texto), que faz a verificação do comando digitado, também sendo enviado a mensagem ao histórico de comandos. Após a leitura do comando, o *InputField* é esvaziado, o foco nele é setado e o timeSinceLastCall é zerado.

Caso o *InputField* esteja vazio, e a tecla Enter seja pressionada e o *InputField* não esteja sendo focado, então o foco ao *InputField* é setado (Figura 34).

Figura 34: Código da caixa de texto para entrada de dados

```
timeSinceLastCall += Time.deltaTime;
if (chatInput.text != "")
{
    if (Input.GetKeyDown(KeyCode.Return) && timeSinceLastCall >= 0.75f)
    {
        chatInput.text = chatInput.text.ToLower();
        //SendMessageToChat(chatBox.text, Message.MessageType.playerMessage);
        if (leitura_texto(chatInput.text))
        {
            SendMessageToChat(chatInput.text, Message.MessageType.playerMessage);
        }
        chatInput.text = "";
        chatInput.ActivateInputField();
        timeSinceLastCall = 0;
    }
}
else
{
    if (!chatInput.isFocused && Input.GetKeyDown(KeyCode.Return))
    {
        chatInput.ActivateInputField();
    }
}
```

Fonte: Imagem capturada diretamente do programa

Através da função leitura_texto é lido o comando digitado, verificando qual ação realizar. A ação que os personagens realizarão é definida pela primeira palavra precedida do '(', podendo ser pular, descer, andar, ativar, atacar, defender, encolher e crescer. Tudo que estiver entre parênteses serão os

parâmetros que correspondem ao comando, que são divididos por vírgula. Estes comandos podem ser:

- andar(nome_do_personagem, distância)
- pular(nome_do_personagem, altura)
- descer(nome_do_personagem)
- ativar(nome_do_botão)
- atacar(nome_do_personagem)
- defender(nome_do_personagem, verdadeiro_ou_falso)
- encolher(nome_do_personagem)
- crescer(nome_do_personagem)

Quando o jogador aperta a tecla “enter”, o texto é lido pelo script. Se o comando é digitado corretamente, a ação é realizada, senão, é enviada uma mensagem de texto em um histórico que mostra todos os comandos já realizados pelo jogador.

O jogador pode apertar para cima ou para baixo para acessar os comandos armazenados no histórico, permitindo mais agilidade no momento da digitação.

3.2.2 Histórico de comandos

O histórico de comandos é representado por um painel, ele também localiza-se no script principal. Na Unity este painel é chamado de *Scroll View*, que também é uma UI. Ele mostra os comandos já digitados pelo jogador em cor verde (para assemelhar a um terminal de computador) e em vermelho caso seja uma mensagem de informação (exemplo: comandos digitados de forma errada).

Em continuidade, o *Scroll View* recebe a mensagem do *InputField*, o qual verifica o tipo da mensagem para poder representá-lo com a cor desejada.

A função é chamada através da função SendMessageToChat (Figura 35):

Figura 35: Função SendMessageToChat

```
SendMessageToChat(chatInput.text, Message.MessageType.playerMessage);
```

Fonte: Imagem capturada diretamente do programa

O qual recebe o texto do *InputField*, e um objeto da classe *Message*, que foi criado em um *script* separado, também chamado Message (Figura 36).

Figura 36: Classe Message

```
public class Message
{
    public string text;
    public Text textObject;

    public MessageType messageType;

    public enum MessageType
    {
        playerMessage,
        info
    }
}
```

Fonte: Imagem capturada diretamente do programa

A função SendMessageToChat, localizado no script principal, faz uma contagem na quantidade de mensagens já enviadas ao histórico, deletando as primeiras mensagens que foram enviadas caso já se tenha passado o limite de mensagens preestabelecidos (Figura 37):

Figura 37: Deleta mensagem

```
if (messageList.Count >= maxMessages)
{
    Destroy(messageList[0].textObject.gameObject);
    messageList.Remove(messageList[0]);
}
```

Fonte: Imagem capturada diretamente do programa

Após isso, verifica-se o tipo da mensagem, alterando sua cor para verde ou vermelho, dependendo do seu tipo, adicionando a mensagem ao histórico (Figura 38):

Figura 38: Alterando cor da mensagem e removendo-a

```
newMessage.textObject.text = newMessage.text;
newMessage.textObject.color = MessageTypeColor(messageType);

messageList.Add(newMessage);

foreach(string part in Errors)
{
    if (!String.Equals(part,text))
    {
        messageListStored.Add(newMessage);
    }
}
```

Fonte: Imagem capturada diretamente do programa

3.2.3 Comandos do jogador

Os comandos que podem ser realizados pelo jogador são pular, descer, andar, ativar, atacar, defender, encolher e crescer. Todos esses comandos são obtidos através do *Input Field*, verificando se a tecla "enter" foi pressionada e se o *Input Field* não está vazio.

Todos os comandos seguem um padrão, assim como citado ao final do item 2.3.5, a ação do jogador vem precedida do abre parênteses. Dentro dos parênteses, localizam-se os parâmetros que a função necessita. Todas as funções possuem como primeiro parâmetro o personagem alvo, e caso a função necessite de um outro parâmetro, o primeiro parâmetro será sucedido por uma vírgula, e essa vírgula sucedida pelo segundo parâmetro. Exemplo:

nome_da_ação(nome_do_personagem , valor_do_segundo_parâmetro)

Nesse segundo parâmetro pode vir uma variável de qualquer tipo, que geralmente é um valor que corresponda à função. Por exemplo, na ação andar seu segundo parâmetro é equivalente à distância que o personagem alvo percorrerá. Ao final do comando, deve-se possuir um fecha parênteses.

Na caixa de texto, a ação desejada pelo jogador é identificada através da função "string.StartsWith(string)". Seus parâmetros são identificados por uma função que identifica a posição do caractere desejado, através do "string.IndexOf(string)", que é armazenada em uma variável inteira, representando sua posição. O mesmo é feito para encontrar a posição da vírgula (se houver) e do fecha parênteses. Através da função "string.Substring(int posição_inicial, int posição_final)", consegue-se obter o primeiro parâmetro e o segundo parâmetro (se houver).

3.2.3.1 Andar

O comando de andar possui dois parâmetros: o personagem alvo que o jogador deseja movimentar e a distância a ser percorrida. Exemplo:

andar(personagem, distância).

Se o nome do personagem for “robot”, e a distância que deseja percorrer for 10 para a direita, o jogador deve digitar da seguinte forma: andar(robot,10). Se a distância a ser percorrida for positiva, o alvo é movido para a direita, caso seja negativa, ele é movido para a esquerda.

Na implementação do código é colocado um limitador para o segundo parâmetro, sendo de -10 a 10. Caso o valor ultrapasse esses limites, o valor é alterado para o valor limite mais próximo.

No código, que é localizado no script principal, é feito a busca pelo primeiro parâmetro (o personagem), pegando seu componente RigidBody2D e componente *script* chamado “Player_movement”. É verificado no *script* Player_movement se a variável canMove (pode mover-se) é verdadeira, caso seja, é passado o destino o qual o personagem deve ir, ou seja, sua posição atual acrescentado do segundo parâmetro (a distância que o jogador passou para o personagem alvo andar). Se a variável canMove for falsa, então o personagem não faz nada (Figura 39).

Figura 39: Parte do código da ação de andar

```
int startIndex = texto.IndexOf("(");
int virgula = texto.IndexOf(",");
int endIndex = texto.IndexOf(")");
string primeiroParametro = texto.Substring(startIndex + 1, virgula - startIndex - 1);
float segundoParametro = float.Parse(texto.Substring(virgula + 1, endIndex - virgula - 1));
if (Convert.ToSingle(segundoParametro) > 10)
{
    segundoParametro = 10;
}
if(Convert.ToSingle(segundoParametro) < 10)
{
    segundoParametro = -10;
}
player = GameObject.Find(primeiroParametro).GetComponent<Rigidbody2D>();
RoboPrincipal = GameObject.Find(primeiroParametro).GetComponent<Player_movement>();
if(RoboPrincipal.canMove == true)
{
    Vector3 dest = player.transform.position;
    dest.x += segundoParametro;
    RoboPrincipal.StartMove(dest);
}
```

Fonte: Imagem capturada diretamente do programa

3.2.3.2 Pular

O comando de pular possui dois parâmetros: o personagem alvo e a altura que o jogador deseja pular. Exemplo:

pular(personagem, altura)

Se o nome do personagem for “robot”, e a altura que deseja pular for 10, o jogador deve digitar da seguinte forma: pular(robot,10).

Este comando faz as mesmas requisições a outros componentes que o comando andar, também possuindo os mesmos limitadores (Figura 40).

Figura 40: Parte do código da ação de pular (no script principal)

```
//guarda o primeiro e segundo parametro, através da localização de seus indexes("(",",",")")
int startIndex = texto.IndexOf("(");
int virgula = texto.IndexOf(",");
int endIndex = texto.IndexOf(")");
string primeiroParametro = texto.Substring(startIndex + 1, virgula - startIndex - 1);
string segundoParametro = texto.Substring(virgula + 1, endIndex - virgula - 1);
if (segundoParametro.Contains(","))
{
    segundoParametro.Replace(',', '.');
    Debug.Log(segundoParametro);
}
if(Convert.ToSingle(segundoParametro) > 10)
{
    segundoParametro = "10";
}
player = GameObject.Find(primeiroParametro).GetComponent<Rigidbody2D>();
RoboPrincipal = GameObject.Find(primeiroParametro).GetComponent<Player_movement>();
if (RoboPrincipal.isGrounded == true && RoboPrincipal.canMove == true)
{
    Player_movement.jumpHeight = Convert.ToSingle(segundoParametro);

    player.velocity = Vector2.up * Player_movement.jumpHeight;
}
```

Fonte: Imagem capturada diretamente do programa

3.2.3.3 Descer

O comando de descer possui apenas um parâmetro, o personagem alvo. Exemplo: descer(personagem).

Neste comando, o “BoxCollider2D” do personagem alvo é desabilitado por um curto período de tempo. Tempo o suficiente para que atravesse a plataforma que ele estava sobre.

Neste comando, é feito acesso ao *script* Player_movement, verificando se o personagem está sobre uma plataforma através da variável onPlatform (sobre plataforma), desabilitando todos os Colliders2D do personagem através da estrutura de repetição foreach, chamando uma co rotina que habilita os Colliders2D após um tempo estabelecido no parâmetro da co rotina (Figura 41).

Figura 41: Código da ação de descer (no script principal)

```
if(GameObject.Find(primeiroParametro) == true){
    if(GameObject.Find(primeiroParametro).GetComponent<Player_movement>().onPlatform == true)
    {
        foreach (Collider2D col in GameObject.Find(primeiroParametro).GetComponents<Collider2D>())
        {
            col.enabled = false;
            StartCoroutine(Habilita_Collider(0.75f, col));
        }
    }

}
```

Fonte: Imagem capturada diretamente do programa

3.2.3.4 Atacar

No caso de atacar, ao passar o parâmetro de atacar ao personagem, ele realizará a ação de atacar. Somente o personagem Lutador faz uso dele, pois um script específico chamado Atacar está inserido nele. Exemplo: atacar(lutador).

No exemplo dado, o personagem “lutador” está realizando a ação de atacar.

O comando atacar possui uma área de verificação, através do componente “BoxCollider2D”, checando se ele é um “trigger (ativador)” ou não, que é acoplado ao personagem que possui o comando (Figura 43). Quando passa-se o comando de atacar, é verificado se há algum objeto destrutível na área desse componente, então caso haja, é realizado a ação de atacar, destruindo os objetos dentro dessa área, ativando a animação de ataque do personagem e do som de destruição (Figura.,42).

Figura 42: Parte do código da ação de atacar (no script principal)

```
scriptAttack = GameObject.Find(primeiroParametro).GetComponent<Attack>();

if(GameObject.Find(primeiroParametro) == true &&
   GameObject.Find(primeiroParametro).GetComponent<Player_movement>().canMove)
{
    scriptAttack.attack = true;
    StartCoroutine("Attack_Recebe_Falso", 1);
}
```

Fonte: Imagem capturada diretamente do programa

Figura 43: Código Atacar que verifica colisão, inserido em um personagem

```
void OnTriggerStay2D(Collider2D other)
{
    if (attack == true && other.gameObject.tag == tagsD[0] || attack == true && other.gameObject.tag == tagsD[1])
    {
        foreach(MeshRenderer mr in other.GetComponentsInChildren<MeshRenderer>())
        {
            mr.GetComponentInChildren<MeshRenderer>().enabled = false;
        }
        other.gameObject.GetComponent<Renderer>().enabled = false;
        foreach (Collider2D col in other.GetComponents<Collider2D>())
        {
            col.enabled = false;

            Destroy(other.gameObject, 0.5f);
        }
        GetComponent<Animator>().Play("cavaleiro_action");
        foreach(Animator explosao in explosaoEffect)
        {
            explosao.Play("explosao_L");
            explosionSound.Play();
        }
        other.gameObject.GetComponentInChildren<ParticleSystem>().Play();

        other.gameObject.GetComponent<BoxCollider2D>().enabled = false;
        //other.gameObject.GetComponent<BoxCollider2D>().enabled = false;

        attack = false;
    }
}
```

Fonte: Imagem capturada diretamente do programa

3.2.3.5 Defender

O comando de defender possui dois parâmetros, o primeiro parâmetro que corresponde ao nome do personagem que realizará a ação de defender e o segundo parâmetro que corresponde a verdadeiro ou falso, no jogo sendo representados respectivamente por *true* ou *false*. Esse segundo parâmetro define se o comando de defender será ativado, caso seja *true*, a ação é ativada, caso seja false, ela é desativada. Quando o comando de defender é ativado, liga-se um escudo transparente, este escudo

transparente possui formato redondo e um *Circle Collider* 2D, utilizado para bloquear projéteis de inimigos. Enquanto ativado, o personagem não pode mover-se, podendo voltar a realizar outros comandos caso ele seja desativado. Apenas o personagem Lutador utiliza esse comando (Figura 44 e 45).

Exemplo de comando defender ativado: “defender(lutador, true)”.

Exemplo de comando defender desativado: “defender(lutador, false)”.

Figura 44: Código da leitura da ação de defender (no script principal)

```
bool segundoParametro = Convert.ToBoolean(texto.Substring(virgula + 1, endIndex - virgula - 1));
scriptDefender = GameObject.Find(primeiroParametro).GetComponent<Defender>();

if (GameObject.Find(primeiroParametro) == true)
{
    scriptDefender.defend = segundoParametro;
    scriptDefender.shieldSound.Play();
}
```

Fonte: Imagem capturada diretamente do programa

Figura 45: Código Defender inserido em um personagem

```
if (defend)
{
    GetComponent<Player_movement>().canMove = false;
    shield.SetActive(true);
}
if (!defend)
{
    GetComponent<Player_movement>().canMove = true;
    shield.SetActive(false);
}
```

Fonte: Imagem capturada diretamente do programa

3.2.3.5 Ativar

O comando de ativar possui um parâmetro: o nome do objeto que será ativado. Exemplo: ativar(nome_do_botão). Este comando localiza-se no *script* principal.

Se o jogador quiser interagir com o botão “alavanca”, ele deverá fazer com que algum personagem chegue perto do botão e digite: ativar(alavanca) (Figura 46). Assim, o personagem próximo a “alavanca” estará ativando-o, o qual poderá ser utilizado como agente para mover uma parede, destruir um objeto, entre outras diversas ações.

Figura 46: Código da ação ativar

```
if (GameObject.Find(primeiroParametro))
{
    if (GameObject.Find(primeiroParametro).GetComponent<OneTimeClick>())
    {
        GameObject.Find(primeiroParametro).GetComponent<OneTimeClick>().interagir = true;
    }
    else
    {
        GameObject.Find(primeiroParametro).GetComponent<BotaoScript>().interagir = true;
    }

    StartCoroutine(Interagir_Recebe_Falso(1,primeiroParametro));
}
```

Fonte: Imagem capturada diretamente do programa

Existem dois tipos de botões, aqueles que são ativáveis apenas uma vez ou aqueles que possuem o estado de ativado e desativado. Primeiro, é verificado se o botão é do tipo OneTimeClick, ou seja, que ativa apenas uma vez. Caso seja, é trocado a variável interagir do OneTimeClick para verdadeiro, senão chama-se o BotaoScript e então sua variável interagir que é setada como verdadeira.

Após a ação anterior, é chamada a co rotina Interagir_Recebe_Falso, que é responsável por setar a variável interagir para falso.

3.2.3.6 Encolher

O comando de encolher possui apenas um parâmetro: o nome do personagem que irá encolher. Apenas a personagem Formiga faz o uso do comando, pois um *script* chamado Encolher que faz tal ação está inserido nela. Quando o comando é ativado, o personagem alvo encolhe de tamanho, podendo passar por lugares que em seu tamanho normal não conseguiria. Sua gravidade é alterada, ficando mais pesado, não conseguindo pular em lugares que conseguiria em seu tamanho normal (Figura 47).

Exemplo: “encolher(formiga)”.

Figura 47: Código da ação encolher

```
if (encolher == true && encolhido == false)
{
    gameObject.transform.localScale = new Vector3(transform.localScale.x / 1.5f,
                            transform.localScale.y /1.5f, transform.localScale.z);
    efeitos.Play();
    encolher = false;
    encolhido = true;
    GetComponent<Rigidbody2D>().gravityScale = 2;
}
```

Fonte: Imagem capturada diretamente do programa

No *script*, a escala do objeto é alterada, dividindo-se sua escala atual por 1.5, nos eixos X e Y. Também nota-se a alteração da escala de gravidade para 2, deixando o personagem mais pesado, diminuindo o alcance de seu pulo.

3.2.3.7 Crescer

O comando de crescer é semelhante ao de encolher, mas faz o contrário: o personagem volta ao seu tamanho normal. Caso o comando seja utilizado enquanto o personagem está em seu tamanho normal, nada acontece. Assim como o comando encolher, somente o personagem Formiga faz o uso desse comando, pois o script Encolher está inserido nela (Figura 48).
Exemplo: “crescer(formiga)”.

Figura 48: Código da ação de crescer

```
if (crescer == true && GetComponent<Encolher>().encolhido == true)
{
    efeitos.Play();
    gameObject.transform.localScale = new Vector3(transform.localScale.x * 1.5f,
                            transform.localScale.y * 1.5f, transform.localScale.z);
    crescer = false;
    GetComponent<Encolher>().encolhido = false;
    GetComponent<Rigidbody2D>().gravityScale = 1;
}
```

Fonte: Imagem capturada diretamente do programa

Nota-se que, em vez de fazer igual ao encolher, que divide por 1.5, o crescer faz o processo contrário, multiplicando o valor 1.5 pela escala atual. A escala de gravidade é alterada para 1

3.2.4 Botão de informação

O botão de informação é um componente da classe *Button* (botão) que é uma UI. Ele é representado por um botão com o texto "i", representando informação. O botão de informação é responsável por mostrar comandos que o jogador poderá utilizar em determinado cenário em que se encontra. Também introduzirá novos comandos que serão utilizados no cenário.

Ao clicar no botão, será ativado um objeto da classe *Panel* (painel), que também é uma UI e armazenará informação de texto ou imagem (*sprites*), podendo ser desativado ao clicar novamente sobre ele.

3.2.5 Botão de pausar

O botão de pausar é semelhante ao botão de informação. Ele ficará responsável por pausar o jogo e então abrir um *Panel* que mostrará outros botões alocados nele. Esses outros botões são as funções de voltar ao jogo, voltar ao menu, fechar o jogo.

3.2.6 Os personagens

Os personagens possuem alguns itens, requeridos para seu funcionamento. Estes itens são um *Animator* (animador), um *Sprite Renderer* (o *sprite* em si), um *script*, um *Rigid body* 2D (responsável pela física do personagem) e dois *Box Colliders* 2D (responsáveis por verificarem as colisões do personagem).

O *Animator* possui um controle, o qual realiza as animações do personagem. Este controle possui estados, indicando em qual estado o personagem estará em determinado momento. Cada estado possui uma animação, que modificará o *sprite* do personagem.

O *Sprite Renderer* armazena o *sprite* do personagem, em conjunto com sua cor (escolhida pelo desenvolvedor). Um *sprite* pode conter milhares de variações de cor, não havendo a necessidade da criação de um mesmo *sprite* para que corresponda com a cor desejada pelo desenvolvedor.

Através do *Rigid Body* 2D são simulados e realizados as físicas que serão aplicadas sobre o personagem. Serão modificados sua massa, gravidade, e se ele rotacionará no eixo Z (para o personagem não ficar rodando pelo mapa). Qualquer tipo de força aplicada sobre ele será verificado e realizado a ação correspondente (como ser empurrado por um outro componente).

O *Box Collider* 2D é uma área responsável por verificar a área que o personagem ocupa. Através dele também são verificados outros colisores existentes pelo mapa, que ao entrarem em conflito são verificados que ações serão realizadas. É através do *Box Collider* 2D que o personagem pode

manter-se de pé sobre o chão (que também possui um *Box Collider* 2D), interagir com outros objetos, e impedir o bloqueio de passagem de outro personagem. O tamanho do *Box Collider* 2D varia de personagem para personagem.

O *script* base de cada personagem é responsável por verificar se o personagem está no chão, verificar o lado que o personagem está virado, realizar os cálculos de seus movimentos (pular e andar), ignorar colisão com outros personagens..

No método *Update* verifica-se se o personagem está se movendo, caso esteja, atualiza o movimento dele através da função de vetor (Vector3) *Lerp*, que possui como parâmetros a localização atual, o destino e a velocidade com que será realizado o trajeto. Caso a posição atual seja igual a posição do destino, seu estado de movimentação é setado como false e então seu movimento é interrompido. Se houver um obstáculo no caminho, que é verificado através da função “OnCollisionEnter2D”, a movimentação do personagem é interrompida setando o estado para false e modificando sua posição em 0,5 unidades dependendo de que lado o personagem está virado, a fim de evitar *bugs* de colisão. O pulo do personagem é calculado apenas através do produto entre a altura passada pelo jogador e por uma variável “Vector2.up”, que indica a direção para cima. É importante lembrar que todo movimento possui interação com o *script* principal, que faz interação com o *Input Field*.

Dependendo do personagem, ele pode ter *scripts* adicionais que permitem a realização de outras ações. Um deles é o personagem “lutador”, que possui o *scrip*t Attack (atacar).

O *script* Attack verifica quais colisores estão em contato com o personagem, e caso a variável de estado “attack” for verdadeira (*true*) e o objeto o qual está colidindo possui a *tag* (identificador) do tipo “Destructible”, então o objeto colisor será destruído, ativando sua animação de partículas.

O personagem “defensor” possui o *script* Defend (defender).

O *script* Defend verifica se o personagem colide com um projétil, destruindo-o, permitindo que ele haja como um escudo para os seus companheiros.

3.2.7 Objetos destrutíveis

Os objetos destrutíveis são compostos por um *Mesh Renderer*, um *Rigid Body* 2D, e dois *Box Colliders* 2D (Figura 49). Eles devem possuir a tag identificadora “Destructible”. Possuem como “filho” um sistema de partículas (intitulado na Unity como *Particle System*).

O *Mesh Renderer* é responsável por criar a aparência 3D do objeto.

O *Rigid Body* 2D, assim como no personagem, é responsável por simular as físicas aplicadas ao objeto.

Um *Box Collider* 2D é responsável por demarcar a área de colisão, enquanto o outro é responsável por demarcar a área em que ele pode ser destruído. Por exemplo, se for ativado o comando de atacar do personagem lutador e o *Box Collider* 2D responsável por demarcar a área em que ele pode ser destruído estiver em contato, então o objeto será destruído.
Quando o objeto for destruído, seu sistema de partículas, se houver, será acionado.

Figura 49: Componentes dos objetos destrutíveis

Fonte: Imagem capturada diretamente do programa

3.2.8 Botões de interação

Os botões de interação possuem dois estados, um correspondendo ao botão ativado, e o outro correspondendo ao botão desativado. Cada botão possui um *script*, portanto possuindo uma função que pode variar de botão para botão. A ação do botão é definido através uma ou duas animações, que podem ser: destruir objetos, mover objetos.

3.2.9 Cenários

Os cenários são compostos por personagens pré determinados; possuem objetos destrutíveis, paredes, chão e plataformas. As plataformas são um tipo de chão o qual só verifica a colisão na parte de cima, ou seja, o personagem pode subir sobre ela, mas só irá descer caso seja atribuído o comando descer; elas são apresentadas com uma cor mais clara, para que possa se diferenciar do chão ou das paredes.
Ao decorrer dos cenários, haverão objetos que lançam projéteis, que causam dano aos personagens, ocasionando a derrota do jogador. Também haverão armadilhas, como um buraco camuflado ou botões que ativam-se automaticamente ao contato com os personagens.
O primeiro cenário ficará responsável por introduzir os comandos básicos ao jogador. São eles andar, pular e interagir. Ao mesmo tempo, será mostrado e contado a história do jogo através dos elementos dos cenários e do sistema de diálogos.

3.2.9.1 Cena 1

O objetivo da cena 1 (Figura 50) é introduzir a mecânica do jogo ao jogador, sendo um tutorial passo a passo, instruindo o jogador a chegar a determinados pontos a fim de aprender algo a respeito da jogabilidade. Ao final o jogador deve ativar um botão que abre uma fresta, que ao passar por ele mostra o seu desempenho na fase e em seguida envia-o para a próxima fase, a cena 2.

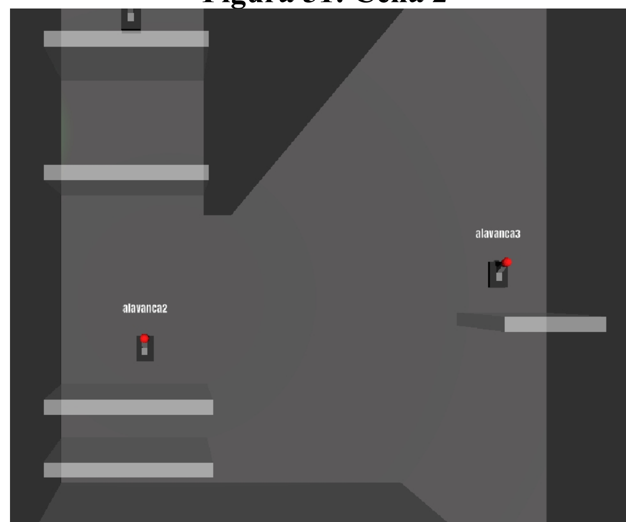
Figura 50: Cena 1

Fonte: Imagem capturada diretamente do programa

3.2.9.2 Cena 2

A cena 2 (Figura 51) foi feita para o usuário treinar os comandos básicos ensinados na cena 1, são eles: andar, pular, descer e ativar botões.

Figura 51: Cena 2



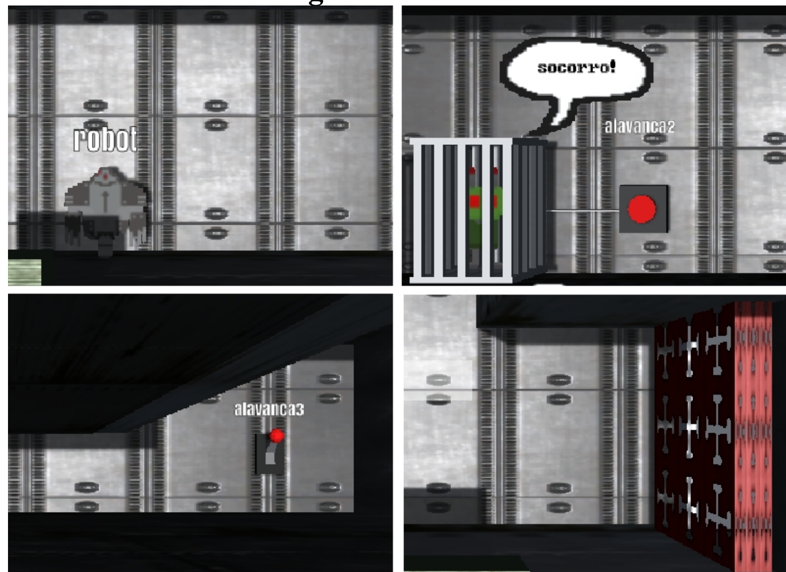
Fonte: Imagem capturada diretamente do programa

3.2.9.3 Cena 3

Na cena 3 (Figura 52) é introduzido um novo personagem, a personagem Formiga. Ao início da fase ela está presa e Robot deve ajudá-la a se soltar. É essencial a ajuda dela para que consiga-se concluir o cenário, pois para chegar à "alavanca3" deve-se passar por uma área em que Robot não consegue, devido ao pequeno espaço, utilizando o comando de encolher para a Formiga.

Quando a alavanca3 é ativada, o portão vermelho se abre, então ambos devem sair por ela e assim concluir o cenário.

Figura 52: Cena 3

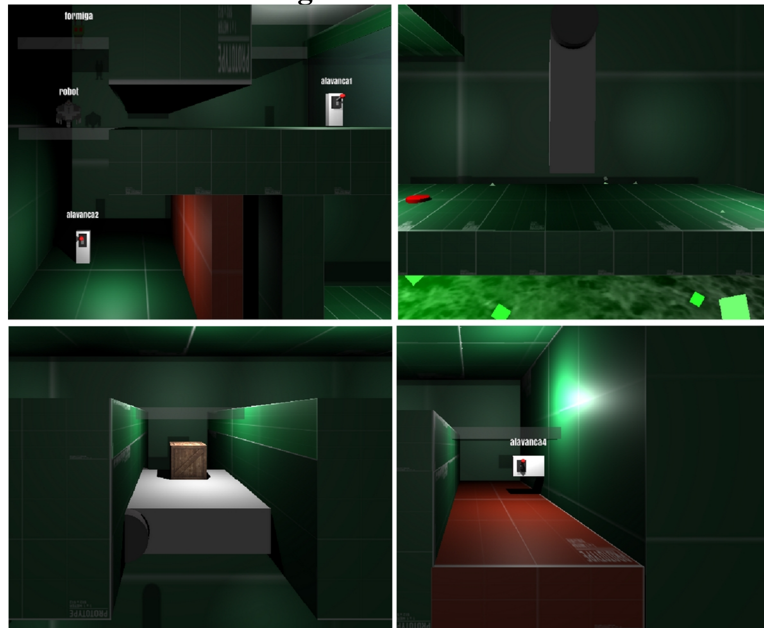


Fonte: Imagem capturada diretamente do programa

3.2.9.4 Cena 4

Nessa cena (Figura 53) é introduzido um novo botão, o botão por pressão, o qual é ativado quando um personagem ou objeto está sobre ele, realizando uma ação. Nesse cenário também é possível fazer com que o personagem morra caso ele caia no ácido. Também é necessário rápida digitação ou acesso aos comandos de andar para a direita, pois uma plataforma é levantada e caso o jogador não consiga levar o personagem rapidamente até ela, ele deverá reiniciar a fase. Também é de suma importância a ordem com que deve-se ativar os botões, pois nesse cenário também é necessário um raciocínio, um verdadeiro quebra-cabeças.

Figura 53: Cena 4

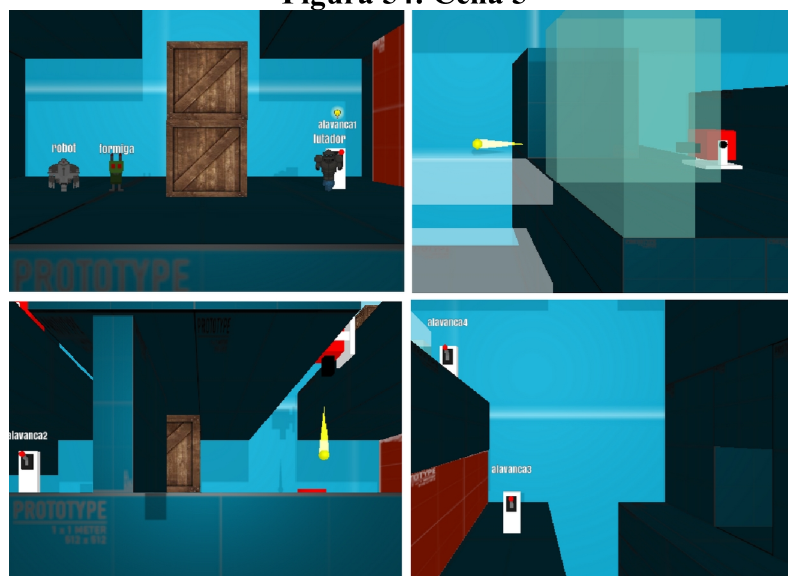


Fonte: Imagem capturada diretamente do programa

3.2.9.5 Cena 5

Nessa cena (Figura 54) é introduzido o personagem Lutador, também sendo apresentados os seus comandos. O jogador terá que fazer uso desses novos comandos do Lutador para que ele destrua os inimigos. Os três personagens devem chegar até o final, sendo necessário um bom raciocínio sobre os passos que devem ser seguidos para a conclusão da fase.

Figura 54: Cena 5

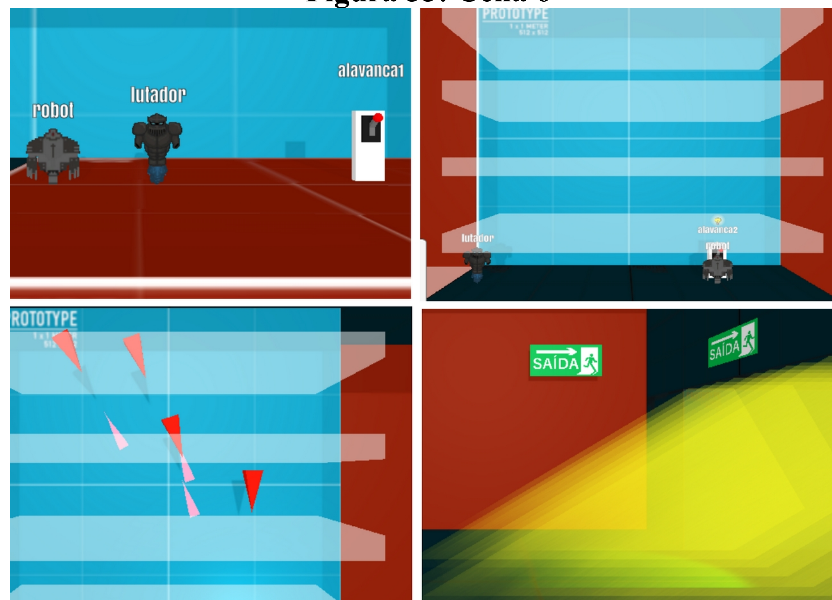


Fonte: Imagem capturada diretamente do programa

3.2.9.6 Cena 6

Na cena 6 (Figura 55), a última, ao ativar a “alavanca2” começam a surgir inimigos. Esses inimigos aparecem a cada 17.5 segundos, e param de nascer quando a animação do elevador terminar. Eles nascem em uma determinada área, sendo chamados através da função *Instantiate*. (Figura 56) Essa área é calculada passando dois “Vector2”, que possuem os eixos X e Y, define-se qual é o máximo e o mínimo, gerando um ponto aleatório entre esses dois, que é onde o inimigo nascerá.

Figura 55: Cena 6



Fonte: Imagem capturada diretamente do programa

Figura 56: Código EnemySpawn da instanciação dos inimigos

```
void Update()
{
    timer += Time.deltaTime;
    instantiateEnemy();
}

void instantiateEnemy()
{
    getCount = GameObject.FindGameObjectsWithTag("Enemy");
    currentEnemies = getCount.Length;
    if (currentEnemies <= maxEnemies && timer >= instantiateTime)
    {
        min = new Vector2(minGO.transform.position.x, minGO.transform.position.y);
        max = new Vector2(maxGO.transform.position.x, maxGO.transform.position.y);
        Vector2 instantiatePos = new Vector2(Random.Range(min.x, max.x), Random.Range(min.y, max.y));
        Instantiate(enemyPrefab, instantiatePos, Quaternion.identity, transform);
        Instantiate(spawnParticle, instantiatePos, Quaternion.identity, transform);
        timer = 0;
    }
}
```

Fonte: Imagem capturada diretamente do programa

Se o jogador ficar apenas na defensiva, mais inimigos surgirão, ficando mais difícil. Ele deve destruí-los caso queira concluir a fase. A fase é concluída indo até a área iluminada em amarelo, que indica a saída do laboratório.

3.2.10 Câmera

A câmera do jogo pode ser movida através do mouse. Ao levar o cursor do mouse até as bordas, se tiver conteúdo do cenário à mostra, a câmera irá movimentar-se para a posição do cursor. Pode-se aumentar ou diminuir o zoom da câmera girando a rolagem do mouse.

3.2.10.1 Mover a câmera se o cursor do mouse estiver na borda

Os comandos localizam-se dentro de um loop. Identifica-se a posição do mouse através do comando "Input.mousePosition"; o tamanho da tela através da classe *Screen.height* ou *Screen.width* (Figura 57).

Figura 57: Código do script CameraEdge da movimentação da câmera

```
if(Input.mousePosition.x > Screen.width - edgeSize)
{
    cameraFollowPosition.x += moveAmount * Time.deltaTime;
}
if (Input.mousePosition.x < edgeSize)
{
    cameraFollowPosition.x -= moveAmount * Time.deltaTime;
}
if (Input.mousePosition.y > Screen.height - edgeSize)
{
    cameraFollowPosition.y += moveAmount * Time.deltaTime;
}
if (Input.mousePosition.y < edgeSize)
{
    cameraFollowPosition.y -= moveAmount * Time.deltaTime;
}
```

Fonte: Imagem capturada diretamente do programa

Localizando a posição do mouse, verifica-se se ele ultrapassou as coordenadas do tamanho da tela, movendo a câmera. São setados os valores máximos e mínimos de X e Y, que limitam a visão da câmera (Figura 58, código CameraEdge).

Figura 58: Limitação de movimento da câmera

```
//Set max and min camera
if (cameraFollowPosition.x < minX)
{
    cameraFollowPosition.x = minX;
}
if (cameraFollowPosition.x > maxX)
{
    cameraFollowPosition.x = maxX;
}
if (cameraFollowPosition.y < minY)
{
    cameraFollowPosition.y = minY;
}
if (cameraFollowPosition.y > maxY)
{
    cameraFollowPosition.y = maxY;
}
```

Fonte: Imagem capturada diretamente do programa

3.2.10.2 Zoom In e Zoom Out da câmera

A câmera possui por padrão o valor “Field of View”, o qual define o distancia de visualização da câmera. Identifica-se a rolagem do mouse 3 através do comando “Input.GetAxis("Mouse ScrollWheel")”, com ele, calcula-se o *Field of View* adicionando a multiplicação por uma constante identificada como “sensibilidade”. Define-se a distância mínima e máxima de visualização através do “Mathf.Clamp(FieldOfView, min_FoV, max_FoV)”.

3.2.11 Cena cinemática

Para o diálogo, será utilizado o “TextMeshPro”, diferente do Text padrão, o TextMeshPro pode fazer alterações mais detalhadas em seu design, o que não é possível no *Text* padrão.

A “Cena cinemática” é feita através de um *script* (Figura 59), onde o roteiro inicia-se com a chamada de uma co rotina, e essa outra co rotina faz a chamada de uma outra co rotina, até chega no seu final. Assim, cada cena cinemática possui um roteiro diferente, necessitando de um *script* diferente para cada uma delas.

Figura 59: Parte do funcionamento do código de cenas cinemáticas

```
public IEnumerator Comeca()
{
    textoHack.text = "";
    textoRobo.text = "Onde... estou?";
    yield return new WaitForSeconds(time);
    StartCoroutine(Pt1());
}

public IEnumerator Pt1()
{
    textoRobo.text = "";
    textoHack.text = "Em um laboratório abandonado";
    yield return new WaitForSeconds(time);
    StartCoroutine(Pt2());
}
```

Fonte: Imagem capturada diretamente do programa

Quando uma cena cinemática é chamada, todos os componentes de interação são desativados, a fim de fazer com que o jogador preste atenção na cena.

3.2.12 Nome dos objetos

Todos os objetos que podem ser utilizados como parâmetro nos comandos utilizados pelo jogador possuem um “TextMeshPro” (Figura 60), responsável por nomear cada objeto através de um texto. Na hierarquia da disposição dos objetos, o TextMeshPro fica no segundo nível, pegando, através de um *script*, o nome de seu “pai” através do comando “transform.parent.name”, alterando o seu texto.

Figura 60: Código NomePai que atribui nome aos objetos

```
GetComponent<TextMeshPro>().text = transform.parent.name;
```

Fonte: Imagem capturada diretamente do programa

Figura 61: Representação do nome dos objetos

Fonte: Imagem capturada diretamente do programa

3.2.13 Número de comandos

O jogo contabilizará o número de comandos que o jogador entrará (Figura 62). Será definido um número mínimo, médio e máximo. Caso o jogador consiga realizar um número de comandos menor ou igual ao número mínimo, ele receberá três estrelas de desempenho (o melhor); caso o número de comandos esteja entre o mínimo e o máximo, ele receberá duas estrelas de desempenho (caso médio); caso o número de comandos seja igual ou maior ao número máximo, ele receberá uma estrela de desempenho (pior caso) (Desempenho do jogador). Representação na Figura 63.

O desempenho do jogador pode ser relacionado com a eficiência dos algoritmos, o qual possuem seus piores casos, casos médios e melhores casos.

Figura 62: Contagem de comandos

Fonte: Imagem capturada diretamente do programa

Figura 63: Desempenho do jogador

Fonte: Imagem capturada diretamente do programa

4. TESTES E FEEDBACK DO PÚBLICO ALVO

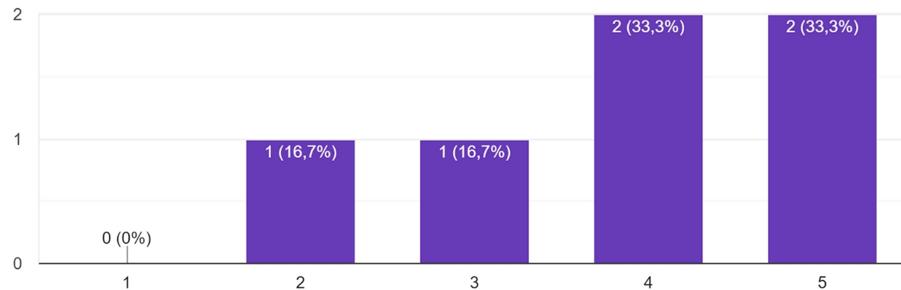
Foi criado um questionário como forma de realização e coleta dos testes de *feedback*. Esse questionário foi disponibilizado em uma plataforma de pesquisa online de formulários da Google. Os jogadores tinham o livre arbítrio de decidir se queriam responder ou não ao questionário.

O jogo Robot.exe foi disponibilizado nos sites: https://kkkkenzo.itch.io/robotexe e https://gamejolt.com/games/robot_exe_br/437192, deixando o link do questionário para que eles o respondessem. Foram feitos 38 downloads no site da Itch.io e no site da Game Jolt foram realizados 16 downloads, totalizando 54 downloads, sendo que desses, apenas 6 jogadores responderam ao questionário. Os itens considerados no questionário foram: design do jogo, entendimento da jogabilidade (ou mecânica), ambientação, monotonia, história, notas gerais, atuação profissional dos usuários e, por fim, sugestões dos jogadores. Esses resultados foram observados no período de duas semanas, do dia 1 ao dia 14 de setembro. Os gráficos que serão mostrados nos itens a seguir são divididos em dois eixos, a horizontal sendo a satisfação do usuário (de 1 a 5) e na vertical a quantidade de votos.

4.1 Opinião sobre o design do jogo

Foi verificado se o design do jogo é agradável, pois a sensação visual que o jogo passa é importante para que mantenha o jogador interessado no mesmo. A interface do sistema assume o papel de conexão entre usuário e conhecimento (NEVES e CANDA). A seguir a Figura 64 em relação ao design do jogo:

Figura 64: Gráfico da opinião sobre o design do jogo



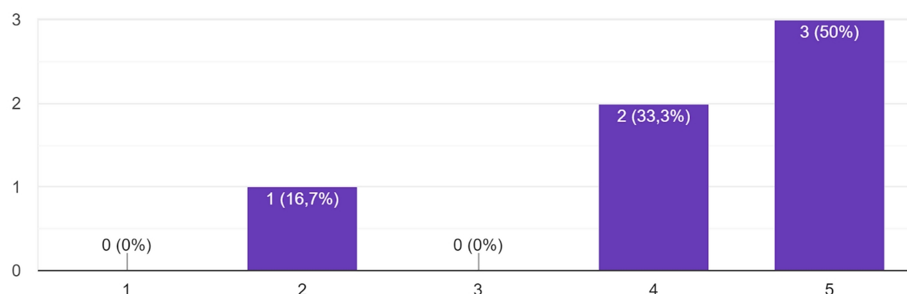
Fonte: Imagem capturada diretamente do programa

O design do jogo está agradável ao público, portanto não seria necessário muitas mudanças a respeito de seu design.

4.2 Entendimento da jogabilidade / mecânica

É importante saber se o jogador conseguiu entender facilmente a jogabilidade. Um jogador que não entende a jogabilidade pode ficar perdido sobre o que deve ser feito no jogo, perdendo seu interesse pelo mesmo e então desistindo. A seguir o gráfico sobre o entendimento da jogabilidade (Figura 65):

Figura 65: Gráfico sobre o entendimento da jogabilidade



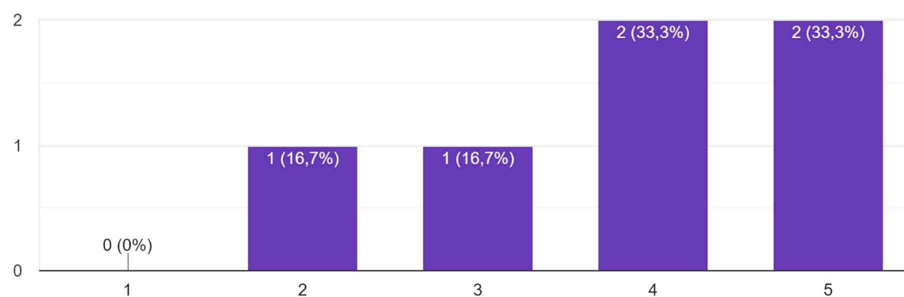
Fonte: Imagem capturada diretamente do programa

Como pode-se ver, a grande maioria entendeu a jogabilidade ou mecânica facilmente, o que significa que nesse aspecto o jogo não peca, e também que a primeira fase, o tutorial, explica bem as mecânicas principais do jogo.

4.3 Ambientação

Foi questionado aos jogadores se o jogo lembra o ambiente de programação. Apesar de não parecer impactante, a assimilação do jogador com o ambiente de programação pode despertar o interesse dele em começar os estudos nessa área. A seguir o gráfico sobre a ambientação do jogo em relação a programação (Figura 65):

Figura 66: Gráfico sobre a ambientação do jogo



Fonte: Imagem capturada diretamente do programa

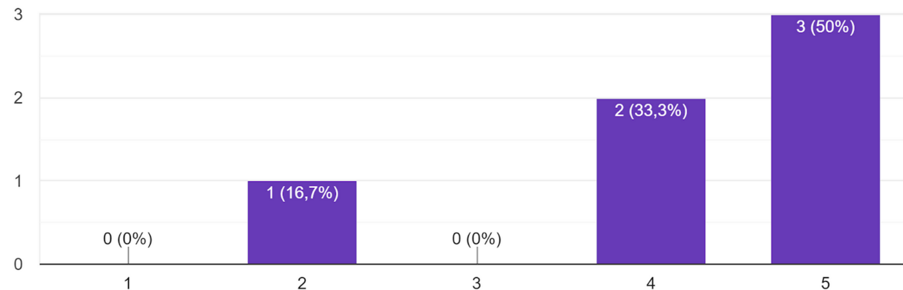
A grande maioria gostou da ambientação relacionada à programação. Com isso, têm-se mais garantia de que os jogadores podem acostumar-se mais facilmente à interface utilizada em blocos de programação.

4.4 Monotonia

A monotonia do jogo deve ser avaliada, pois o interesse do jogador pode ser perdido se um jogo se estagna sempre em uma dificuldade. A elevação e variação dos desafios é importante para manter esse quesito. A seguir o gráfico na Figura 66 em relação a monotonia do jogo:

Figura 67: Gráfico sobre a monotonia do jogo

O jogo evolui num ritmo adequado e não fica monótono – oferece novos obstáculos, situações ou variações de atividades.
6 respostas



Fonte: Imagem capturada diretamente do programa

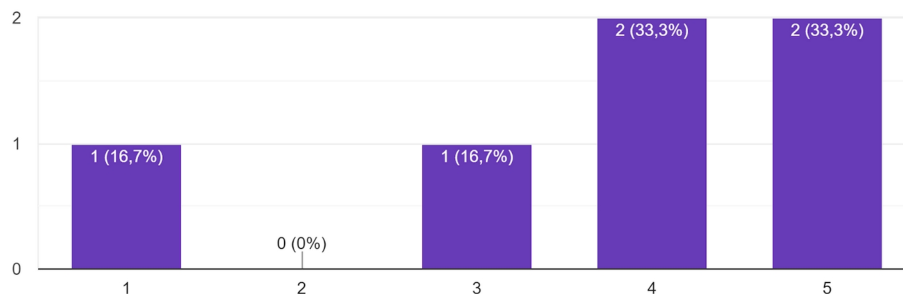
Com os resultados, verifica-se que a grande maioria não achou o jogo monótono, a introdução de novos elementos em cada cenário novo possibilita tal feito.

4.5 História

Verificar se a história é entendível é importante para saber se ela segue num ritmo que tenha contexto. A amostragem de acontecimentos passados pode causar confusão em relação a história. Coletar o *feedback* e ajustar a história de uma forma que fique clara e entendível. A seguir o gráfico da Figura 68 a respeito da história do jogo:

Figura 68: Gráfico a respeito da história do jogo

Consegui acompanhar a história do jogo
6 respostas



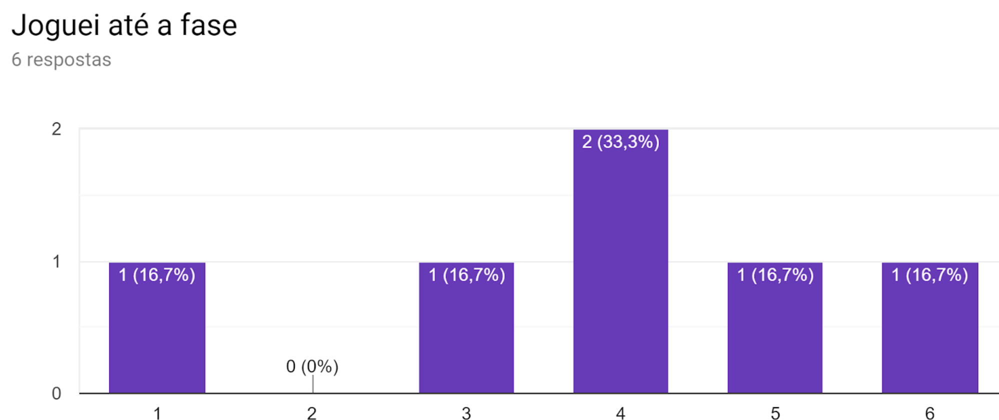
Fonte: Imagem capturada diretamente do programa

Apesar de a maioria ter entendido, houve um que não conseguiu captar a história do jogo. Isso se deve pelo jogador possivelmente não ter conseguido ler os textos de falas, o que significa que algumas mudanças no tempo que cada fala permanece pode ser alterada.

4.6 Até que fase jogaram?

Saber até que fase jogaram nos dá a informação necessária para saber se os jogadores realmente gostaram do jogo ou se alguma fase específica está muito difícil de ser concluída, como podemos ver na Figura 69.

Figura 69: Gráfico sobre até que fase jogaram



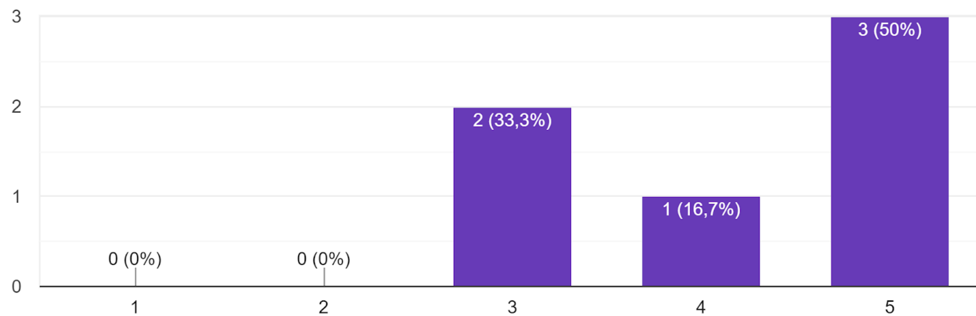
Fonte: Imagem capturada diretamente do programa

Nota-se que a fase 4 foi onde mais jogadores pararam, ou seja, pode ser que ela tenha um nível de dificuldade muito alto para o momento em que ela é apresentada.

4.7 Notas gerais

As notas gerais nos dá uma noção de como o jogo se saiu ao todo. Se é viável ou não para aplicação no auxílio do ensinamento da linguagem de programação. A seguir, na Figura 70, o gráfico com as notas gerais:

Figura 70: Gráfico de notas gerais



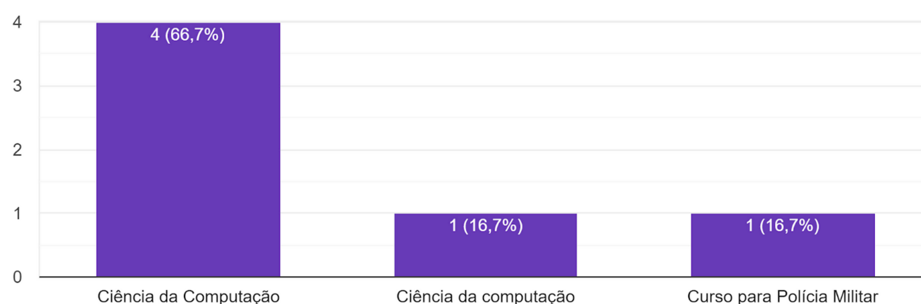
Fonte: Imagem capturada diretamente do programa

Não houve nenhuma avaliação negativa (menor que 2), o que significa que o jogo possui uma boa aceitação e consegue entregar a sua proposta de ensinar de uma maneira mais intuitiva.

4.8 Atuação profissional dos usuários

Saber a área de atuação do jogador nos dá a informação se o jogo é aceito a pessoas fora da área da computação, apesar de apenas um jogador que é fora dessa área tenha dado feedback. Na Figura 71 um gráfico sobre a atuação profissional / escolar dos jogadores:

Figura 71: Gráfico da atuação profissional/escolar dos usuários



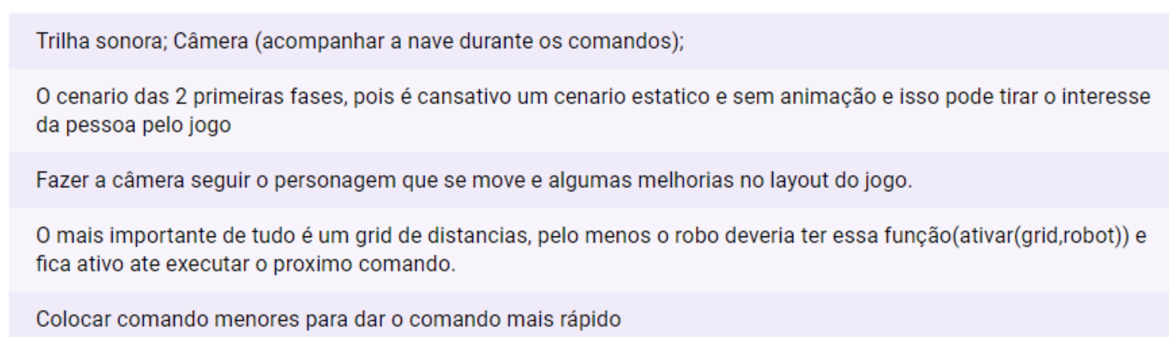
Fonte: Imagem capturada diretamente do programa

Como a grande maioria é da área da computação, tem-se mais certeza a respeito dos resultados obtidos, pois é o público principal do jogo.

4.9 Sugestões

As sugestões dos jogadores contribuem para que os pontos fracos do jogo sejam melhorados, deixando o jogo mais otimizado e agradável aos jogadores. Grandes jogos sempre fazem esse tipo de coleta antes do lançamento oficial deles, para que em seus lançamentos estejam agradáveis ao público. A seguir uma lista de respostas sobre a pergunta "O que você melhoraria no jogo?"

Figura 72: Sugestões dos jogadores



Fonte: Imagem capturada diretamente do programa

Olhando a figura, percebe-se que há muitas melhorias que podem ser feitas no jogo. A correção desses detalhes podem influenciar numa boa experiência para o jogo, mesmo que pequenas.

Com todos os dados coletados, conclui-se que o jogo teve uma boa aceitação, e que sua proposta de ser um jogo educacional voltado a área da computação foi concluída.

Não foram realizados questionários para o jogo Illuminate, por motivos de falta de tempo. Contudo, o jogo pode ser baixado através do link https://kkkkenzo.itch.io/illuminate.

5. CONCLUSÃO

Nos tempos atuais, a programação vem sendo cada vez mais presente em nossas vidas, assim como a matemática. Com o rápido desenvolvimento da tecnologia, as áreas tecnológicas vêm aumentando suas vagas enquanto outras vêm perdendo espaço pela mesma, o que aumenta a demanda por um aprendizado melhor da linguagem de programação.

O projeto atingiu seu principal objetivo, o de introduzir a linguagem de programação ao seu público de uma forma mais divertida e lúdica. Embora o objetivo tenha sido concluído, ainda há muitas melhorias que podem ser aplicadas aos jogos desenvolvidos, e também há muito a ser estudado a respeito do tema.

5.1 Dificuldades encontradas

O processo do desenvolvimento de um jogo envolve muitas áreas, sendo elas arte, música, conhecimentos específicos e programação. Para atender toda essa diversidade a contento precisa-se de uma equipe de desenvolvedores, pois uma única pessoa dificilmente terá dotes em todas essas áreas, resultando em um jogo limitado em alguns aspectos. No caso desse trabalho, por se tratar de um TCC que deve ser desenvolvido por uma única pessoa, o autor teve que fazer o trabalho que uma equipe faria, o qual, naturalmente, resultou em um jogo com artes e música limitados, devido ao autor não ter experiência nessas áreas.

5.2 Trabalhos futuros

Todo jogo pode possuir conteúdos adicionais (mais popularmente conhecido como DLCs – *Downloadable Contents*) para enriquecer no universo de um jogo. No caso de Robot.exe, no início de seu desenvolvimento estava previsto a criação de mais duas fases, mas devido a limitação de tempo foram desenvolvidos apenas seis. O jogo precisa passar também por uma revisão mais apurada.

Assim como Robot.exe, o jogo Illuminate também pode ser criado mais DLCs. Como exemplo, algumas dessas DLCs podem ser mais fases e personagens novos. Alguns ajustes (correção de *bugs*) também precisam ser feitos. Para trabalhos futuros em Illuminate, podem ser introduzidos mais conceitos específicos da linguagem de programação, como estruturas de repetição com teste no início e com teste no final; para as estruturas de seleção podem ser introduzidas seleções compostas;

a implementação da junção das duas estruturas, a de repetição e seleção, agregando mais complexidade ao ensino.

REFERÊNCIAS BIBLIOGRÁFICAS

CUP HEAD. Jogo digital de entretenimento. Disponível em <http://www.cupheadgame.com/ >. Acesso em 19 nov. 2019 11h53min.

CODE COMBAT. Jogo digital educacional. Disponível em <https://br.codecombat.com/ >. Acesso em 19 nov. 2019 às 11h47min.

COSTA, Thaise de Amorim et al. **O Ensino de Linguagem de Programação na Educação Básica Através da Robótica Educacional: Práticas e a Interdisciplinaridade.** VI Congresso Brasileiro de Informática na Educação (CBIE 2017). Disponível em <http://www.br-ie.org/pub/index.php/wie/article/download/7287/5085>. Acesso em 07 jun. 2019 às 11h00min

DERRYBERRY, Anne. "Serious games: online games for learning**"**. **I'm Serious.net** - 2018. Disponível em <https://iktmangud.files.wordpress.com/2014/09/online-games-for-learning.pdf>. Acesso em 07 jun. 2019 às 10h29min

FLEURY A., NAKANO D., CORDEIRO J. H. D.. **Mapeamento da Indústria Brasileira de Jogos Digitais.** São Paulo: Pesquisa do GEDIGames, NPGT, Escola Politécnica, USP, para o BNDES, 2014.

FORBELLONE, Andre Luiz Villar; EBERSPACHER, Henri Frederico. Lógica de Programação: A construção de algoritmos e estruturas de dados. **Pearson**. 3. ed. - São Paulo: Pearson Prentice Hall, 2005.

GALLEGO, J. P.. **A Utilização dos Jogos como Recurso Didático no Ensino-Aprendizagem da Matemática**. Monografia (Graduação em Pedagogia), UNESP – Universidade Estadual Paulista "Júlio de Mesquita Filho" - Campus de Bauru, Bauru, 2007.

GRUBEL, Joceline Mausolff; BEZ, Marta Rosecler. Jogos Educativos. **RENOTE - Revista Novas Tecnologias na Educação** - 2006. Disponível em <https://www.seer.ufrgs.br/renote/article/view/14270/8183>. Acesso em 11 abr. 2019 às 8h42min.

HEARTHSTONE. Jogo digital de entretenimento. Disponível em <https://playhearthstone.com/pt-br/ >. Acesso em 19 nov. 2019 às 11h52min.

HOED, Raphael Magalhães. **Análise de evasão em cursos superiores: o caso da evasão em cursos superiores da área de Computação**. Monografia (Mestrado Profissional em Computação Aplicada), Universidade de Brasília – Brasília, 16 dez. 2016. Disponível em <https://core.ac.uk/download/pdf/80747023.pdf >. Acesso em 12 nov. 2019 às 13h08min.

KAPP, Karl M. **The gamification of learning and instruction: game-based methods and strategies for training and education**. San-Francisco, United States of America: Co-published with ASTD, 2012

LIGHTBOT. Jogo digital educacional. Disponível em <https://lightbot.com/ >. Acesso em 19 nov. 2019 às 11:45.

MONCLAR Rafael Studart; SILVA, Marcelo Arêas; XEXÉO, Geraldo. **Jogos com propósito para o ensino de programação**. SBC – Proceedings of SBGames 2018 — ISSN: 2179-2259. Disponível em <http://www.sbgames.org/sbgames2018/files/papers/EducacaoFull/188132.pdf>. Acesso em 11 abr. 2019 às 8h33min.

MOSER, Robert. **A fantasy adventure game as a learning environment: Why learning to program is so difficult and what can be done about it**. Artigo. University of South Wales. Disponível em <http://darkwynter.com/repo/Personal/Amanda/Thesis/papers/moser_fantasy.pdf >. Acesso em 19 de nov. 2019 às 11:00.

MOTTA, Rodrigo L.; JUNIOR, José Trigueiro. **Short game design document (SGGD)**. SBC – Proceedings of SBGames 2013. Disponível em

<http://www.sbgames.org/sbgames2013/proceedings/artedesign/15-dt-paper_SGDD.pdf >. Acesso em 12 nov. 2019 às 15h40min.

NEVES, Libni Almeida; KANDA, Jorge Yoshio. Desenvolvimento e avaliação de jogos educativos para deficientes intelectuais. **Nuevas Ideas en Informática Educativa, Volumen 12, p. 612 - 617**. Santiago de Chile. Disponível em <http://www.tise.cl/volumen12/TISE2016/612-617.pdf >. Acesso em 27 de nov. 2019 às 13:30

NEWZOO. **Brazil Games Market 2018**. Disponível em : < https://newzoo.com/insights/infographics/brazil-games-market-2018/>. Acesso em 21 mar. 2019 às 9h46min.

ROCHA, Paulo Santana et al. Ensino e aprendizagem de programação: análise da aplicação de proposta metodológica baseada no sistema personalizado de ensino. **RENOTE – Revista Novas Tecnologias na Educação**, v. 8, n.3, dez. 2010. Disponível em <https://www.seer.ufrgs.br/renote/article/view/18061/10649>. Acesso em 12 nov. 2019 às 11h40min.

RODRIGUES, Lídia da Silva. **Jogos e brincadeiras como ferramentas no processo de aprendizagem lúdica na alfabetização.** Dissertação de Mestrado. Faculdade de Educação da Universidade de Brasília. 2013.

SOUZA, Isabel Amorim de; SOUZA, Luciana Vírgilia Amorim de. O USO DA TECNOLOGIA COMO FACILITADORA DA APRENDIZAGEM DO ALUNO NA ESCOLA. **Revista fórum identidades** - 2010. Disponível em <https://seer.ufs.br/index.php/forumidentidades/article/view/1784/1573>. Acesso em 06 de jun. 2019 às 08h30min.

UNITY. Disponível em <https://unity3d.com/pt/programming-in-unity>. Acesso em 12 de out. 2019 às 19h09min.