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

GAMEMAKER: STUDIO COMO FERRAMENTA DE APRENDIZAGEM DE PROGRAMAÇÃO

Alex Ferreira Costa

Profa. Dra. Mercedes Rocio Gonzales Márquez (Orientadora).

Dourados-MS
2017

Curso de Ciência da Computação

Universidade Estadual de Mato Grosso do Sul

GAMEMAKER: STUDIO COMO FERRAMENTA DE APRENDIZAGEM DE PROGRAMAÇÃO

Alex Ferreira Costa

Setembro de 2017

Banca Examinadora:

Profa. Dra. Mercedes Rocio Gonzales Márquez (Orientadora)
Área de Computação – UEMS

Profa. MSc. Jéssica Bassani de Oliveira
Área de Computação – UEMS

Prof. Esp. Alcione Ferreira
Área de Computação – UEMS

GAMEMAKER: STUDIO COMO FERRAMENTA DE APRENDIZAGEM DE PROGRAMAÇÃO

ALEX FERREIRA COSTA

Este exemplar corresponde à redação final da monografia da disciplina Projeto Final de Curso devidamente corrigida e defendida por Alex Ferreira Costa e aprovada pela Banca Examinadora, como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação. Título do Trabalho Bacharel em Sistemas de Informação.

Dourados, 27 de outubro de 2017.

Profa. Dra. Mercedes Rocio Gonzales Márquez
(Orientadora)

Dedico este trabalho aos meus familiares e aos meus amigos que estiveram sempre ao meu lado pelo apoio motivacional durante esses anos.

"Não há conhecimento que não seja poder"

(Ralph Waldo Emerson).

AGRADECIMENTOS

Agradeço a Deus por estar comigo ao longo de toda a minha vida, ainda mais durante o desenvolvimento deste trabalho.

Agradeço também minha orientadora, Profª Mercedes Rocio Gonzales Márquez que me auxiliou semana a semana com este trabalho.

A Profª Maria de Fátima Grassi (in memorian) que além de corrigir várias vezes o trabalho me incentivou a não desistir antes de tentar e de estar sempre à disposição para ajudar.

Também não posso esquecer de minha mãe, a dona Maria Clementino que sempre apoiou minhas decisões, e que apesar de passarmos por alguns momentos complicados, nunca deixou faltar o pão na mesa.

Agradeço também aos amigos do fórum TGM Brasil, antiga Game Maker Brasil, onde fiz diversas amizades e conheci a ferramenta que é objeto de estudo deste trabalho.

Aos meus Professores desde o primário até o atual momento pelo conhecimento transmitido.

Por último e não menos importante, minha chefe e também amiga, Célia Maria Cortez que compreendeu algumas ausências e permitiu que eu usasse do tempo livre no trabalho para fazer trabalhos da faculdade.

RESUMO

Programadores em geral, precisam ter o raciocínio lógico apurado e dominar inúmeras técnicas matemáticas para que sua criatividade tome forma e se apresente em um *software*. Por vezes métodos convencionais de ensino não surtem efeito no aprendizado que os ingressantes da área recebem.

Tentar driblar essa questão é o enfoque deste trabalho. Mostrar que criar jogos digitais, pode ser uma forma alternativa de adquirir conhecimento, além de melhorar a percepção dos estudantes quanto a diversos fatores complexos que envolvem a programação.

Para tal objetivo, nos utilizamos de todas as tecnologias necessárias e de fácil alcance, para que o aprendizado seja acessível.

A videoaula em plataformas online vem se mostrando eficiente em diversos pontos, como flexibilidade, portabilidade e acessibilidade. Estudar onde quiser e no momento que quiser é uma bandeira que deve ser levantada, já que o modelo fixo está ficando impraticável para a sociedade imediatista em que vivemos. Neste trabalho são apresentados detalhadamente processos de criação de jogos e programação, através da ferramenta *GameMaker: Studio*, que além de prática, também é robusta, possuindo diversas funcionalidades avançadas.

Palavras-Chave: *jogos*, *programação*, *ensino*.

ABSTRACT

Developers usually need to have a determined logical reasoning and to know many mathematical techniques to your creativity prosper and turn it in software. Sometimes conventional teaching methods have no effect on students that entering in this world.

Try to circumvent this issue is the focus of this work. To show that create digital games can be an alternative way of acquiring knowledge, and improve the perception of the students when the many complex factors that involve programming.

For this purpose, we use in all the necessary technology and within easy reach, so that learning is accessible.

A video lesson on online platforms is proving effective in several points, such as flexibility, portability and accessibility. To study where you want and when you want, is a flag that should be raised since the actual model is becoming impractical for an immediatist society in which we live. This paper presents a detailed process of creating games and programming through GameMaker: Studio tool, which in addition to simple, it is also robust, having many advanced features.

Key-Words: *games, programming, teaching*.

SUMÁRIO

LISTA DE SIGLAS

GMS: GameMaker: Studio

GML: Game Maker Language

HTML5: Hypertext Markup Language 5

GLSL: OpenGL Shading Language

HLSL: High Level Shading Language

MP3: MPEG1 Layer 3

WAV: Waveform Audio File Format

OGG: Ogg Vorbis

RAM: Random Access Memory

CPU: Central Processing Unit

FIFO: First In First Out

LIFO: Last In First Out

LISTA DE FIGURAS

APENDICE A:

APENDICE B:

APENDICE C:

LISTA DE QUADROS

LISTA DE TABELAS

1. INTRODUÇÃO

O impacto que a computação causou nas vidas das pessoas pode ser considerado espantoso levando em conta as últimas duas décadas, porém sua história começa bem antes com fatos muitas vezes esquecidos na linha do tempo dessa ciência. De acordo com Fonseca Filho:

> A evolução tecnológica se nos apresenta abrupta, através de saltos descontínuos, e todo o trabalho que antecede cada etapa aparece coberto por uma camada impenetrável de obsolescência, algo para a paleontologia ou para os museus, como se nada pudesse ser aprendido do passado (2007, p.23).

Mecanismos analógicos que automatizavam determinadas tarefas também podem ser considerados uma forma de computação primitiva, como a Régua de Cálculos criada por William Oughtred em 1633 (GADELHA, 2001).

Apesar de que a Ciência da Computação tem sofrido profundas transformações em decorrência das novas tecnologias de hardware e das novas técnicas e paradigmas de desenvolvimento de software, o ensino desta ciência não acompanha a curva com que ela se desenvolveu, ou seja, os métodos de ensino não têm evoluído o suficiente para ficar à par do dinamismo do avanço desta área de conhecimento.

A área de construção de algoritmos e programação de computadores constitui-se a porta de entrada para esta ciência, porém o ensino ainda segue o molde convencional de muitos anos atrás. Embora mais pessoas programarem hoje em dia, ainda sim é uma parcela ínfima da população. Segundo dados publicados em 2016 pela *Evans Data Corporation*, há 21 milhões de desenvolvedores no mundo atualmente. Ou seja, 0.3% da população mundial. A empresa ainda faz uma projeção para 2020 de 25 milhões de desenvolvedores. Ao lado dos engenheiros, profissionais da área de Tecnologia da Informação (TI) estão no topo da lista dos profissionais que estão mais em falta no mercado (BRESSANE, 2015).

Traçando esse paralelo histórico com a falta de programadores no mercado, se tornou objetivo primordial das grandes potências como os Estados Unidos da América incentivar seus cidadãos a aprenderem programação. Em um discurso feito para a *Computer Science Education Week*, ou "*CSEdWeek*" em 2013, o então presidente dos Estados Unidos, Barack Obama destacou a importância de se aprender programação: "Peço a você que se envolva. Não compre apenas um novo videogame, crie um. Não baixe um aplicativo. Ajude a desenvolvê-lo. Não apenas brinque em seu celular, mas programe-o."

Este trabalho busca propor uma forma alternativa dinâmica, moderna e lúdica de adquirir conhecimento dos diversos fatores complexos que envolvem a programação de computadores, visando melhorar o formato tradicional de ensino desta disciplina tão importante da Ciência da Computação.

Para tanto foi escolhida a ferramenta *GameMaker: Studio*[1], uma *Game Engine*[2] dedicada principalmente a produção de jogos em duas dimensões (2D) que possui uma linguagem de programação bem amigável, a *Game Maker Language* (GML).

1.1. Objetivos

O objetivo deste projeto é ensinar programação com a ferramenta *GameMaker: Studio* através de videoaulas na plataforma de vídeos da *Google*, o *YouTube*. Para isso alguns objetivos específicos devem ser cumpridos:

1.1.1. Objetivos Específicos

1. Revisão de conceitos ligados aos jogos e à computação;
2. Estudo detalhado da ferramenta *GameMaker: Studio* e seus recursos.
3. Introduzir pessoas à programação através da criação de jogos. Isto será realizado através de da gravação de videoaulas e sua publicação no YouTube. Estas aulas terão os seguintes conteúdos:
    a. Ensinar o básico da linguagem *GML:*
        i. Variáveis e Macros;
        ii. Laços de repetição;
        iii. Estruturas condicionais;
        iv. Funções e Scripts;
        v. Estruturas de dados: listas, pilhas e filas.
    b. Ensinar como criar jogos com mecânicas simples.

[1] Ferramenta para desenvolvimento de jogos: http://www.yoyogames.com/

[2] Software com módulos dedicados a criação de jogos.

1.2. Motivação

Um testemunho pessoal: Quando ingressei no curso de Sistemas de Informação um fato que chamou muito a minha atenção foi que, a maioria dos alunos da minha turma tinham grandes dificuldades no aprendizado de tópicos básicos da programação. Também percebi que a mesma situação se repetia em outras instituições.

Por outro lado, antes do meu ingresso no curso, em meados de 2008, conheci uma ferramenta dinâmica para criação de jogos eletrônicos, chamada *Game Maker*, nessa época ainda na versão 7.0. Esse *software* possui uma linguagem de programação em alto nível e orientada a objetos que lembra bastante *JavaScript*[3]. Assim, tenho convicção de que o que me ajudou pode auxiliar outras pessoas a ingressar na área de programação.

1.3. Metodologia

O projeto foi dividido em três partes: Primeiramente foi feita a revisão bibliográfica do assunto.

Na segunda foram feitos vídeos sobre a instalação do software, recursos que compõem um jogo feito no *Game Maker* e a linguagem de programação estudada, no caso a *GML*. No Quadro 1.1 estão listados os vídeos produzidos:

Quadro 1. 1 - Lista de vídeos I

LISTA DE VÍDEOS I
1 - Introdução
- 1.1 O que é Game Maker: Studio?
- 1.2 Instalando o GameMaker: Studio
- 1.3 Visão Geral do programa e interface gráfica
2 - Recursos do programa
- 2.1 Usando Sprites
- 2.2 Objetos, instancias e seus eventos
- 2.3 Rooms e Views
- 2.4 Backgrounds e Tiles
- 2.5 Sons e Músicas
3 - Game Maker Language
- 3.1 Variáveis, Vetores (Arrays) e constantes
- 3.2 Estruturas condicionais: IF ELSE, SWITCH e Expressões
- 3.3 Estruturas de repetição I: WHILE e DO UNTIL
- 3.4 Estruturas de repetição II: FOR e REPEAT
- 3.5 Funções e Scripts
4 – Estruturas de dados
- 4.1 Listas
- 4.2 Grelhas

[3] Linguagem de programação interpretada comumente usada em aplicações web.

- 4.3.1 Filas
- 4.3.2 Filas de prioridade
- 4.4 Pilhas

Fonte: Quadro criado pelo autor

E na terceira parte foram produzidos vídeos ensinando a criar diversos tipos simples de jogos para ajudar o desenvolvimento da lógica de programação. Temos 8 vídeos desenvolvendo jogos (Quadro 1.2):

Quadro 1. 2 - Lista de vídeos I

LISTA DE VÍDEOS II
Criando um Runner com o GM:S:
- CRIANDO UM RUNNER #01: Movimento, Sprites e Gerador de obstáculos
- CRIANDO UM RUNNER #02: Gravando pontuações em um arquivo INI
- CRIANDO UM RUNNER #03: Menu, Músicas, SFX e Correções
Movimentação:
-MOVE #01: Introdução: Movimento Top-Down
-MOVE #02: Sem limites: Movimento suave
-MOVE #03: Plataforma: Pulo, aceleração e velocidade máxima
Exemplos avançados:
-Exemplo de inventário
-Exemplo de SUDOKU

Fonte: Quadro criado pelo autor

1.4. Conceito de Jogos

O conceito de *jogo* em si vem de muito antes do de *jogo eletrônico* no qual este trabalho se sustenta. A etimologia da palavra jogo vem do latim: *iocus*, *iocare* que significa divertimento, passatempo, brinquedo, como também pode simbolizar coisas que formam uma coleção[4].

Um jogo pode ser uma competição consigo mesmo ou com outrem. Geralmente este jogo possui regras definidas e desafia ao praticante, o jogador, a realizar ações para vencer ou organizar uma situação. Sabendo disso Julia, (1996, p. 3) menciona que jogos podem ser até mesmo coisas extremamente naturais como as que ocorrem na vida animal, como as disputas por fêmeas em época de acasalamento e o cachorro que corre atrás de seu próprio rabo.

Feijó et al. destacam a importância dos desafios em um jogo:

> No caso dos videogames, é justamente esse fator desafio que importa: criar desafios para o cérebro, porém em situações totalmente desconectadas da realidade, portanto, dando-nos uma total sensação de segurança, caso não consigamos "vencer". É por isso que passamos muito mal numa situação de assalto à mão armada real, mas

[4] Fonte: http://www.dicionarioetimologico.com.br/jogo/

> podemos nos divertir bastante ao participar de um combate virtual. Poder perder é uma necessidade de um jogo, pois se sempre ganharmos não existe desafio, ou pelo menos ele pode se tornar muito pequeno a ponto de não exercitar nosso cérebro o suficiente. Jogar futebol pode ser muito divertido, mas ficar passando a bola um para outro, sem nenhum propósito, pode se tornar uma tarefa sem graça (FEIJÓ; CLUA; SILVA, 2010, p. 183-184).

A recompensa também é um elemento importante do jogo e está diretamente ligada ao objetivo dentro dele. Assim como na vida onde somos recompensados com dinheiro pelo nosso trabalho, em um jogo isso também se torna essencial, tanto quanto os obstáculos que temos que evitar.

Os jogos eletrônicos por sua vez podem ser entendidos de uma forma mais específica uma vez que são mais recentes. Um jogo eletrônico necessita de uma máquina para ser jogado. Nesse ambiente entram os fliperamas, máquinas caça-níqueis, consoles de videogame, computadores pessoais, maquinas de *Pinball*[5], *Smartphones* [6]e etc.

1.5. A computação e os jogos educacionais

Como o jogo é tratado, muitas vezes como uma atividade infantil, já que advém de diversão e brincadeiras, foi difícil a acepção de jogos para o âmbito do ensino, pois, estudar e brincar sempre foram atividades abordadas de formas diferentes e em locais diferentes.

Wangenheim (2012) faz os seguintes questionamentos:

1. Por que usar jogos para ensinar?
2. É possível usar jogos para ensinar também computação como uma área das ciências exatas?
3. Como, na prática, ensinar computação com jogos?

Fazendo uma reflexão sobre o questionamento inicial, pode-se dizer que os jogos estimulam o pensamento lógico de forma mais ativa do que outras atividades:

> A competição inerente aos jogos garante-lhes o dinamismo, o movimento, propiciando um interesse e envolvimento naturais do aluno e contribuindo para seu desenvolvimento social, intelectual e afetivo. (MORATORI, 2003, p. 16).

[5] É um jogo eletromecânico onde o jogador manipula duas ou mais 'palhetas' de modo a evitar que uma ou mais bolas de metal caiam no espaço existente na parte inferior da área de jogo.
[6] Celular inteligente que possui tecnologias avançadas e executa um sistema operacional equivalente ao computador.

Para responder a segunda questão, é necessário frisar que jogos em sua maioria estão ligados aos números e às regras. Logo, fica fácil deduzir que isso é perfeitamente possível, usar jogos para aprender Computação. Mas isso vai depender muito da didática do educador, do contexto em que ele pretende aplicar e do quanto de tecnologia ele irá usar, pois, dependendo do equipamento, pode se tornar inviável financeiramente tanto para instituição de ensino quando para o discente.

Com a última indagação, Wangenheim (2012) se desafia a mostrar o que pode, de fato, ser feito e mostra isso com a conclusão de sua pesquisa enfatizando a imersão, a diversão e a interatividade dos alunos durante o jogo.

Em uma comparação com este trabalho temos a seguinte diferença: Aqui proponho "Criar jogos para aprender programação" e não "Aprender programação através de jogos". Ou seja, o objetivo não é criar um jogo educativo e sim criar diversos tipos de jogos para adquirir e melhorar as habilidades de programação.

1.6. Trabalhos Relacionados

(FEIJÓ; CLUA; SILVA, 2010) apresenta como desenvolver games com um *Motor de Jogo*[7] feito em *Java*[8] justamente para facilitar a criação de jogos abstraindo o conteúdo. Mesmo assim, boa parte da programação em *Java* é ensinada antes para se desenvolver as habilidades para usar o motor.

Em **(WANGENHEIM, 2013)** pode-se ver como justificar o uso de jogos para o aprendizado de ciências exatas como a computação através da interação e da imersão proporcionada.

(MORATORI, 2003) desmistifica os jogos como apenas diversão e entretenimento, revelando que os jogos educativos podem e devem ser usados como ferramentas de aprendizado.

1.7. Organização dos capítulos

O Capítulo 2 tratará da revisão sobre a ferramenta estudada, já no Capítulo 3 são explanados conceitos básicos de programação no *GameMaker: Studio*, no Capítulo 4 serão dispostos os jogos produzidos, e por fim no Capítulo 5 as conclusões sobre o trabalho.

[7] É um pacote de funcionalidades que são disponibilizadas para facilitar o desenvolvimento de um jogo evitando construí-lo do zero.
[8] Linguagem de programação interpretada e orientada a objetos.

2. GAMEMAKER: STUDIO

Neste capítulo serão descritos conceitos básicos e características da ferramenta *GameMaker: Studio* bem como sua história e suas características.

2.1. História

Em 1999, o holandês Markus Hendrik Overmars, então Professor titular de Ciências da Informação e Computação na Universidade de Utrecht, criou um protótipo de software de animação chamado *Animo* que viria mais tarde a se tornar o *Game Maker*. Sua pesquisa se focava na área de modelagem *3D*, animação computadorizada, personagens virtuais, simulação e aspectos artísticos dos jogos[9].

Inicialmente o *Animo* serviria para criar animações, porém os usuários do programa começaram a criar jogos simples ao invés disso. Dessa forma, Markus Overmars passou a direcionar o desenvolvimento do programa para essa área e trocou o nome do programa para *Game Maker*. No Quadro 2.1 é destacado o histórico de versões do *Game Maker*.

[9] Fonte: https://pt.wikipedia.org/wiki/Game_Maker:_Studio

Quadro 2. 1 - A evolução do Game Maker com o passar dos anos

Ano	Versão	Novidades	Downloads[10]
1999	1.3	Primeira versão dedicada à criação de jogos.	366 / ano
2000	1.4 e 2.0	Melhorias na interface gráfica.	40.000 / ano
2001	3.0 e 4.0	Uso da *DirectX*[11] para desenhos, programa reescrito do zero na versão 4.0.	270.000 / ano
2002	4.2	Versão mais estável.	750.000 / ano
2003	5.0 - 5.2	Sistema de arquivos, *Timelines*[12] e Partículas.	1.700.000 / ano
2004	5.4 e 6.0	Funções gráficas avançadas a partir do melhor uso da *DirectX* com suporte 3D e melhorias no motor de som.	1.600.000 / ano
2005	6.1	Editor de imagem melhorado e uso de *Surfaces*[13].	10.000 / dia
2006	7.0	Beta da versão 7.0 que agora seria paga e distribuída pela empresa holandesa *YoYoGames*, sendo que Markus agora seria um de seus diretores.	Desconhecido
2007	7.0	Versão final.	Desconhecido
2009	8.0	Suporte a imagens com transparência e maior velocidade.	Desconhecido
2011	8.1 e *HTML5*[14]	No início do ano a versão 8.1 deu sinais que teria suporte a outras plataformas além do *Windows*[15]. O que só ocorreu no fim do ano com a versão *HTML5*.	Desconhecido
2012 a 2016	Studio	Exportação para diversas plataformas tais como dispositivos móveis, computadores e videogames. Além de um novo sistema de rede e de Física.	Desconhecido
2017	Studio 2	Exportação para diversas plataformas tais como dispositivos móveis, computadores e videogames. Agora disponível para *MacOS*.	Desconhecido

Fonte: *YoYoGames (wiki.yoyogames.com).*

[10] Transferência de dados de um servidor para um cliente.
[11] Conjunto de bibliotecas gráficas desenvolvidas pela Microsoft.
[12] Recurso do Game Maker que posiciona ações em uma linha temporal.
[13] Recurso Gráfico do Game Maker para criar áreas de desenho personalizadas.
[14] Principal linguagem de marcação utilizada na Web em sua mais recente versão.
[15] Sistema Operacional de Computadores produzido pela Microsoft.

Markus Overmars permaneceu como codiretor da *YoYoGames* até fevereiro de 2015. Atualmente é cofundador da *Fans4Music*, plataforma social onde os músicos podem interagir com seus fãs. Ele também é o criador da *Qlvr*, uma produtora de jogos sérios com temáticas como saúde e educação[16].

2.2. Características

Este subcapítulo irá detalhar das características e funções mais importantes do *GameMaker: Studio*.

2.2.1. Multiplataforma

Com o *GameMaker: Studio* pode-se publicar jogos em diversos dispositivos como Computadores com *Windows*, *Mac OS X*[17] ou *Ubuntu*[18]. Em dispositivos móveis com *Android*[19], *iOS*[20], *Windows Phone*[21] ou *Tizen*[22]. Nos consoles *PlayStation Vita*[23], *PlayStation 3*[24], *PlayStation 4*[25], e *Xbox One*[26]. E ainda pode exportar jogos em *HTML5* para plataformas *Web* e para a *Amazon FireTV*[27]. Na Figura 2.1 são destacadas as plataformas de exportação:

Figura 2. 1 - Plataformas de Exportação

Fonte: Página de recursos do programa (http://www.yoyogames.com/gamemaker/features)

[16] Fonte: https://nl.linkedin.com/in/markovermars
[17] Sistema Operacional para computadores produzido pela Apple.
[18] Sistema Operacional para computadores distribuído pela Canonical.
[19] Sistema Operacional para dispositivos móveis distribuído pela Google.
[20] Sistema Operacional para dispositivos móveis distribuído pela Apple.
[21] Sistema Operacional para dispositivos móveis distribuído pela Microsoft.
[22] Sistema Operacional para dispositivos móveis distribuído pela Samsung.
[23] Console de videogame portátil criado pela Sony.
[24] Console de videogame criado pela Sony.
[25] Console de videogame criado pela Sony.
[26] Console de videogame criado pela Microsoft.
[27] Reprodutor de mídia digital da Amazon.

Ter seu jogo em várias plataformas multiplica o alcance e o torna mais acessível. A maior parte dos módulos de exportação do programa é paga, sendo possível comprar um pacote com todos os módulos e garantir módulos que venham a sair no futuro gratuitamente. De início pode-se testar gratuitamente em computadores com Windows sem a opção de exportar, logo a versão gratuita não atrapalha o uso para fins educativos. A versão 1.4 não está mais a venda, porém comprando a versão 2, ganha a 1.4 como extra. Na Tabela 2.1 tem-se os valores dos módulos:

Tabela 2. 1 - Tabela de preços do *GameMaker: Studio 2* na *Steam*

Módulo	**Preço (Em reais)**
GameMaker: Studio 2 Desktop	R$ 169,99
GameMaker: Studio 2 Mobile	R$ 680,00
GameMaker: Studio 2 Web	R$ 252,99
GameMaker: Studio 2 UWP	R$ 680,00

Fonte: Página de compra na *Steam*[28] - *2017*

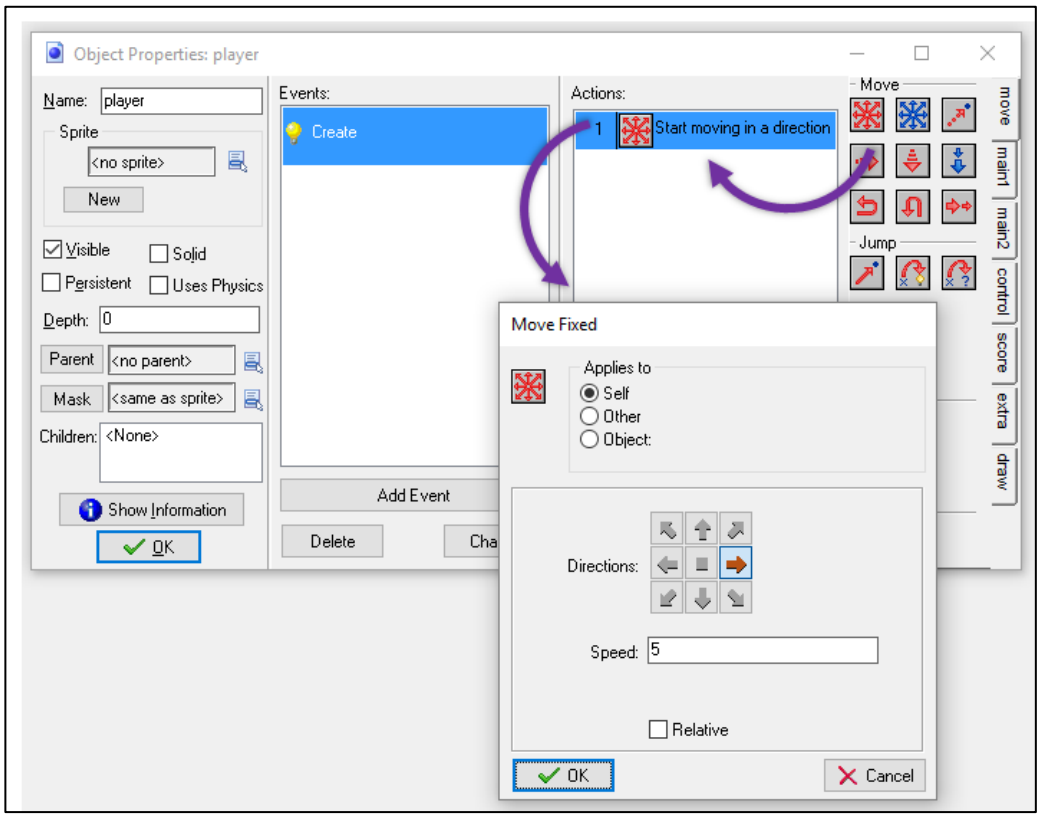
(http://store.steampowered.com/app/585410/GameMaker_Studio_2_Desktop/)

2.2.2. Sistema de arrastar e soltar

Consiste em uma forma de programação estruturada através de sequencias de *blocos de ações*[29], simulando a programação escrita como mostra a Figura 2.2. Indicado para quem nunca teve contato com programação, pois, com o tempo e a complexidade dos códigos pode ficar difícil organizar as ações.

[28] Plataforma de venda de jogos e softwares em formato digital.
[29] São blocos que representam trechos de código visualmente com campos de valores.

Figura 2. 2 - Sistema de arrastar e soltar



Fonte: imagem capturada e editada pelo autor diretamente do programa.

A mesma ação da Figura 2.2 pode ser feita com o seguinte trecho de código (Figura 2.3):

Figura 2. 3 - Trecho de código para movimento horizontal

```
1 direction = 0
2
3 speed = 5
```

Fonte: Imagem capturada pelo autor diretamente do programa.

É uma prática comum não encorajar o uso desse sistema, pois mesmo para tarefas simples acaba se tornando mais objetivo ensinar a linguagem *GML* (CAPÍTULO 3).

2.2.3. A linguagem de programação *GML* (*Game Maker Language*)

A GML é uma linguagem de alto nível orientada a objetos que se assemelha tanto a *C/C++* [30]como a *JavaScript*. Possui estruturas de controle de fluxo condicional, laços de repetição, *tipagem fraca*[31] de variáveis, *vetores dinâmicos*[32] e etc. Na Figura 2.4 vê-se um código simples de movimento para um jogo desenvolvido durante o projeto (Ver CAPITULO 4).

Figura 2. 4 - Exemplo de código de movimento em GML

```
1  /// Movimento
2
3  // Pulo
4  if keyboard_check_pressed(vk_up) and
5  place_free(x, y + 4) == false
6  {
7      vspeed = -14
8      audio_play_sound(sdJump, 1, 0)
9  }
10
11 // Abaixar
12 if keyboard_check(vk_down) and
13 place_free(x, y + 4) == false
14 {
15     duck = true
16 }
17 else
18 {
19     duck = false
20 }
```

Fonte: Imagem capturada pelo autor diretamente do programa.

2.2.4. Sistema de Física Box2D

O *GameMaker: Studio* possui um sistema de física integrado chamado *Box2D*. Esse sistema dá mais realismo aos jogos proporcionando simulações de gravidade, massa, colisões simples e complexas e etc.

O jogo *Angry Cats Space* (Figura 2.5) tem seu código-fonte disponível para *download* e vem como material de auxílio do próprio *GameMaker: Studio* justamente para entender como funcionam as várias funções de física.

[30] Linguagem orientada a objetos derivada da linguagem C.
[31] Variáveis não precisam ser declaradas com tipo especifico e flutuam facilmente de um tipo para outro.
[32] Não é preciso chamar métodos para redimensionar vetores, isso é feito automaticamente ao adicionar um novo valor.

Figura 2. 5 - Jogo de demonstração *Angry Cats Space*

Fonte: Retirada de *Droid.Informer (http://angry-cats-space.android.informer.com)*

2.2.5. IAP (In-App Purchases) – Compras no jogo

Nos últimos anos tem se popularizado os jogos *pay-to-win*, onde você tem um jogo gratuito, mas que possui itens compráveis (Itens de melhoramento do personagem, dinheiro virtual, itens raros e etc.) justificando o nome de 'pagar para vencer', em razão de os itens representarem vantagens no jogo[33] (Figura 2.6).

O *GameMaker: Studio* apresenta o sistema *IAP (In-App Purchases)*[34] para que possa ser feito esse tipo de operação de compra dentro do jogo.

[33] Fonte: http://www.arkade.com.br/entao-quer-dizer-modelo-pay-to-win-realmente-funciona/
[34] Compras dentro do aplicativo.

Figura 2. 6 - Compras no jogo Clash of Clans

Fonte: Retirada do portal *TechTudo (http://www.techtudo.com.br/dicas-e-tutoriais/noticia/2014/05/clash-clans-veja-como-subir-de-nivel-rapido-no-jogo-para-android-e-ios.html).*

2.2.6. Extensões

Caso o desenvolvedor queira expandir as funcionalidades do GameMaker: Studio, há a possibilidade de usar extensões para tal. Em casos mais avançados, onde a GML não dê suporte, para que se crie uma extensão é necessário programar na linguagem nativa (Ou equivalente, mas que produza o arquivo necessário de extensão) da plataforma a qual utilizará a extensão. O Quadro 2.2 lista os tipos das extensões e suas respectivas plataformas:

Quadro 2. 2 - Extensões do GameMaker: Studio

Tipo de extensão	**Plataforma destino**
GML	Feita na linguagem nativa do *GameMaker: Studio* e compatível com todas as plataformas.
Arquivo *.js	Arquivo *JavaScript* que pode ser utilizado no *HTML5*, *Windows 8* e *Tizen*.
Arquivo *.dll	Compatível com as plataformas *Windows*, porém cada uma com seu tipo diferente.
Arquivo *.so	Compatível com *Tizen* e *Linux*, porém cada um com seu tipo diferente.
Arquivo *.prx	Compatível com *PS3* e *PS4*, porém cada um com seu tipo diferente.
Arquivo *.suprx	*PSVita*.

Arquivo *.dylib	*Mac OS X*.
placeholder	Pode ser de qualquer tipo e geralmente é usado em conexões no *Android* e *iOS*.

Fonte: Referência do *GameMaker: Studio*[35]

2.2.7. Shaders

São códigos que permitem que o desenvolvedor altere as características de desenho através das placas de vídeo em um nível baixo de programação. Isso traz a possibilidade de fazer efeitos complexos usando o *hardware*[36] de modo otimizado. Como por exemplo: água (Figura 2.7), fogo, fumaça, explosões e neve realisticas.

Figura 2. 7 - Efeito de água realista em um jogo de Plataforma

Fonte: Imagem retirada do MarketPlace | YoYoGames
(https://docs.yoyogames.com/source/dadiospice/001_advanced%20use/extensions/creating%20extensions.html)

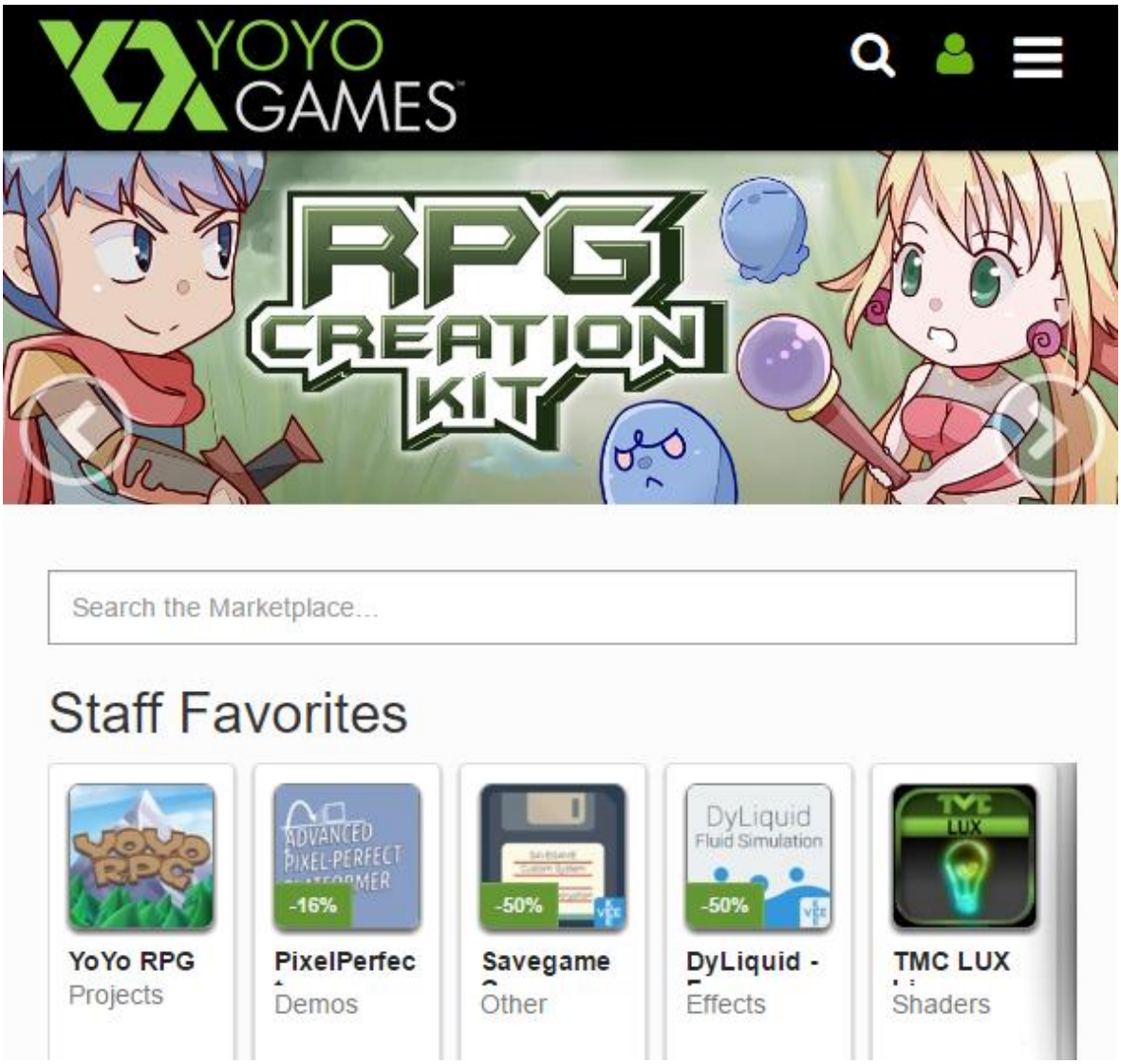
As linguagens para *Shaders* disponíveis são a *GLSL*[37] (Que usa de base a *OpenGL*) e *HLSL* [38](Baseada na biblioteca *DirectX*).

2.2.8. Marketplace – Compra de recursos

[35] Disponível em: < https://docs.yoyogames.com/source/dadiospice/001_advanced%20use/extensions/creating%20extensions.html>
[36] Peças e equipamentos que compõem um computador.
[37] A *OpenGL Shading Language* é uma linguagem usada para programar *Shaders* através da biblioteca *OpenGL*.
[38] A *High Level Shading Language* é uma linguagem usada para programar *Shaders* através da biblioteca *DirectX*.

Os desenvolvedores podem vender recursos como Imagens, efeitos sonoros, músicas, códigos, extensões e até modelos de projetos bem completos em uma loja oficial e de fácil acesso no próprio programa (Figura 8).

Figura 2. 8 - Marketplace, loja de recursos



Fonte: Imagem retirada do MarketPlace | YoYoGames (https://marketplace.yoyogames.com)

3. PROGRAMANDO COM O GAME MAKER – A SÉRIE DE VÍDEOS

Neste capítulo será destacado o conteúdo aplicado nas videoaulas sobre a ferramenta *GameMaker: Studio*. Os vídeos foram divididos em seções para facilitar o aprendizado e até o momento foram desenvolvidas 6 (seis) seções: Introdução, Recursos do *GameMaker: Studio*, *GML*, movimentação e Criando um *Runner*[39]. No total foram produzidos 28 (vinte e oito) vídeos distribuídos entre essas seções totalizando mais de 10 horas de reprodução e 7.124 visualizações.

A plataforma escolhida para distribuir as videoaulas foi o *YouTube*, pois além de ser gratuita também possui diversas ferramentas de edição e publicação de vídeos. Os vídeos estão organizados em *playlists*[40] no canal *Red Screen Soft*[41] onde há outros informativos sobre programação.

3.1. Introdução

Aqui será abordado o básico sobre a ferramenta *GameMaker: Studio*, desde como obter uma cópia do *software* até como instalá-lo na máquina.

3.1.1. O que é GameMaker: Studio?

Figura 3. 1 - Capa do vídeo 'O que é GameMaker: Studio'

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=yTJxPU_HWQ4)

[39] Estilo de jogo onde o personagem percorre um cenário infinito pontuando enquanto estiver vivo.
[40] Lista de reprodução sequencial de algum tipo de mídia (Áudio, vídeo, e etc.).
[41] Nome da produtora do autor deste trabalho.

No vídeo *O que é GameMaker: Studio* (Figura 3.1) é explicado do que se trata a ferramenta *GameMaker: Studio* bem como seus recursos. Observa-se isso em detalhes no Capítulo 2 explanando os recursos do programa e parte de sua história.

3.1.2. Instalando o GameMaker: Studio

Figura 3. 2 - Capa do vídeo 'Instalando o GameMaker: Studio'

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=7TJDruv7vzY)

Nessa videoaula (Figura 3.2) são apontados os procedimentos para a instalação do *software* e criação de conta de usuário. Todos os passos para instalação estão no Apêndice A.

3.1.3. Visão Geral e Interface Gráfica

Figura 3. 3 - Capa do vídeo 'Visão Geral e Interface Gráfica'

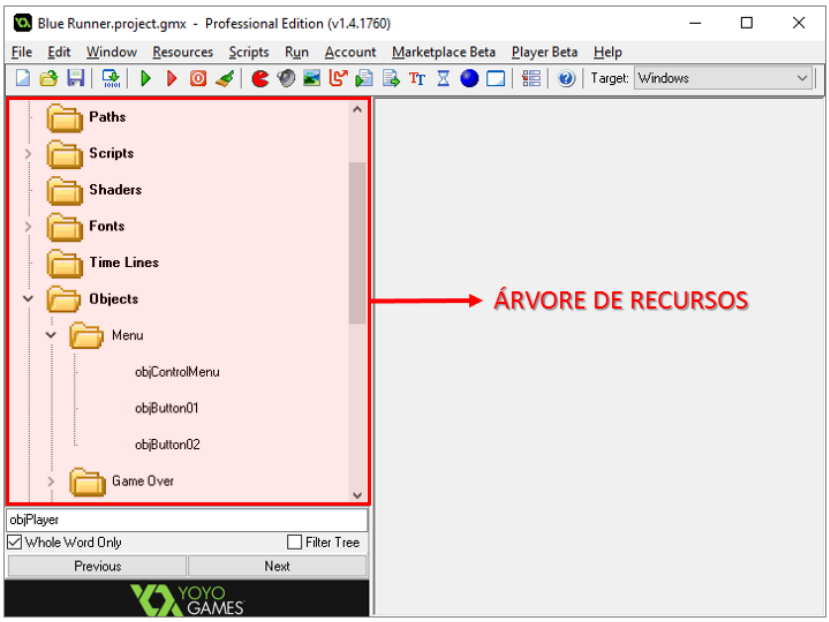
Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=c1b6z2IkD9I)

Nesta parte é mostrada a interface gráfica do programa e sua estrutura organizacional com base em uma árvore de recursos (Figura 3.3).

3.1.3.1. Árvore de recursos

O *GameMaker: Studio* possui um sistema de organização de arquivos comumente usado em diversos programas que trabalham com muitos arquivos, que é a árvore de recursos (Figura 3.4). Nessa árvore ficam organizados os recursos que estarão no jogo como imagens, sons, scripts e etc.

Figura 3. 4 - Árvore de recursos



Fonte: Imagem capturada diretamente do programa

Nessa árvore estão dispostos os seguintes recursos (Quadro 3.1):

Quadro 3. 1 - Recursos do GameMaker: Studio

Recursos	**Descrição**
Sprites	Imagens estáticas ou animadas usadas nos objetos
Sounds	Músicas ou efeitos sonoros
Backgrounds	Imagens grandes estáticas usadas nas *rooms*
Paths	Trajeto pré-determinado para um objeto seguir
Scripts	Trechos de códigos com parâmetros ou não
Shaders	Códigos para controlar o desenho da tela, geralmente para criar efeitos
Fonts	Estilos de textos
Time Lines	Progressão de ações pontuadas pelo tempo

Objects	Elemento base no jogo (Ex: personagem, moeda, controlador de música).
Rooms	Cenários do jogo onde se posicionam os objetos
Included files	Arquivos que serão acoplados ao executável do jogo
Extensions	Expansões de funcionalidades
Macros	Código de entrada substituído por um identificador de saída

Fonte: Referência do *Game Maker* (http://docs.yoyogames.com/source/dadiospice/000_using%20gamemaker/003_gamemaker%20studio%20overview.html)

Alguns desses recursos serão descritos em detalhes na próxima seção (3.2).

3.2. Recursos do GameMaker: Studio

Os recursos listados no Quadro 3.1 serão descritos com detalhes nesta seção, com base nos vídeos produzidos.

3.2.1. Sprites

Figura 3. 5 - Capa do vídeo 'Usando Sprites'

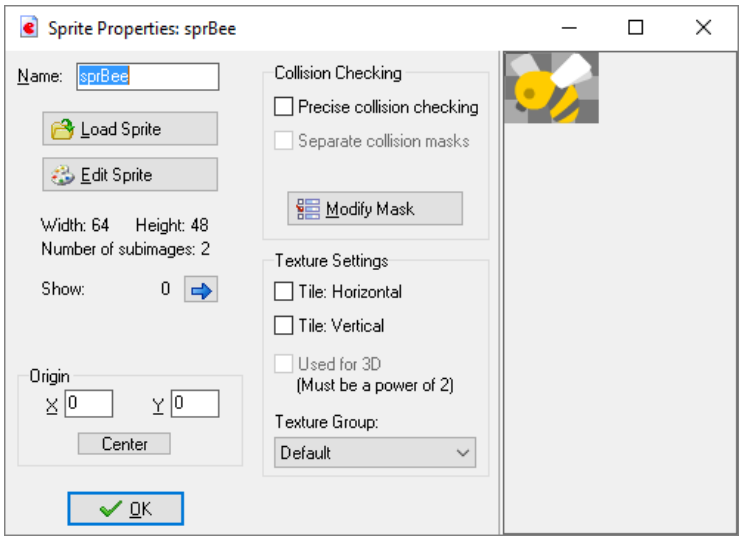
Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=PmUdAn-9wwQ)

No vídeo *Usando Sprites* (Figura 3.5) aprende-se usar a estrutura mais básica de imagens do *GameMaker: Studio*.

Sprites são imagens estáticas ou animadas que são anexadas a objetos para representar elementos gráficos no jogo como personagens, obstáculos, pontuações e etc.

Ao criar uma sprite temos a seguinte configuração:

Figura 3. 6 - Configuração da Sprite



Fonte: Imagem capturada diretamente do programa

Seguindo a Figura 3.6, em *Name* é colocado o identificador das sprites que pode conter letras maiúsculas e minúsculas sem acentos (O cedilha é proibido), números (Desde que não seja o primeiro caractere) e o símbolo sublinhado.

O botão *Load Sprite* serve para carregar uma nova imagem na Sprite. Ao clicar nessa opção o usuário pode escolher qual imagem anexar a partir do seu computador. Vários formatados são suportados (*GIF*, *JPG*, *PNG*, *BMP* e etc.).

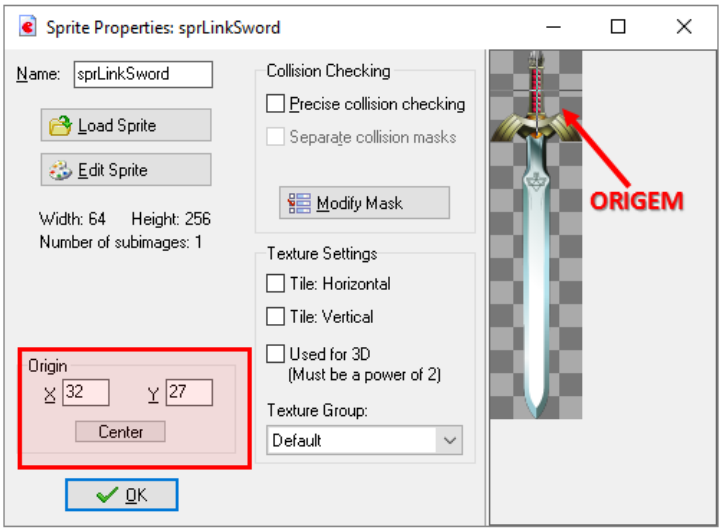
O botão *Edit Sprite* leva o usuário para um editor completo de imagens para auxiliar na criação de animações e efeitos.

Abaixo temos informações básicas como Altura (Em pixels), Largura (Em pixels) e número de *quadros* [42]que compõem a *Sprite*.

Na seção *Origin* definimos um eixo para a *Sprite*, a partir desse ponto a imagem será posicionada, girada e escalonada. Se, por exemplo, houvesse a imagem de uma espada (Figura 3.7) seu eixo deveria ficar no punho da mesma, como na imagem:

[42] Uma imagem que em combinação de reprodução em sequência da a ilusão de movimento criando uma animação.

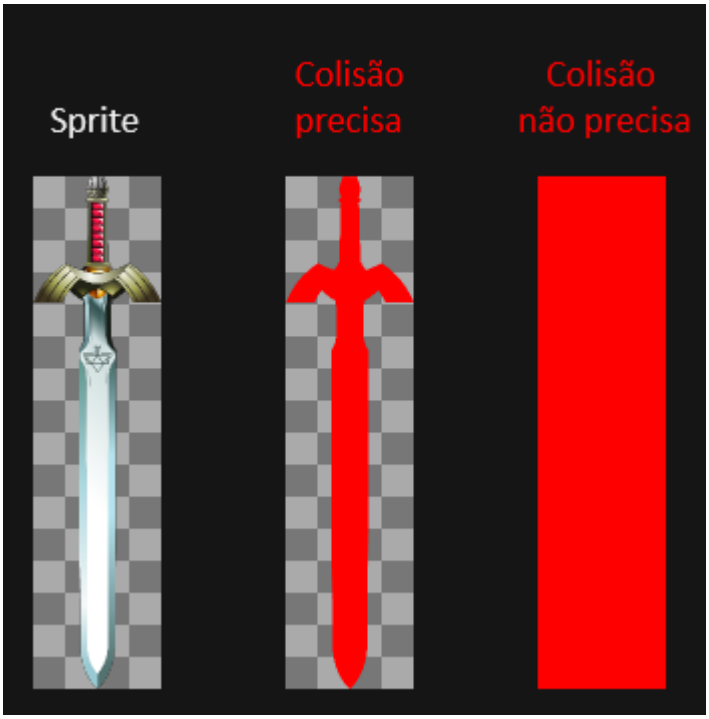
Figura 3. 7 - Origem da *Sprite*



Fonte: Imagem capturada diretamente do programa

Na seção *Collision Checking* é configurado de que forma essa *Sprite* irá se comportar nas colisões. A opção *Precise Collision Checking* faz com que os pixels não transparentes sejam a máscara de colisão, do contrário um retângulo preencherá os limites da *Sprite*. Isso pode ser exemplificado na Figura 3.8:

Figura 3. 8 - Precisão da colisão

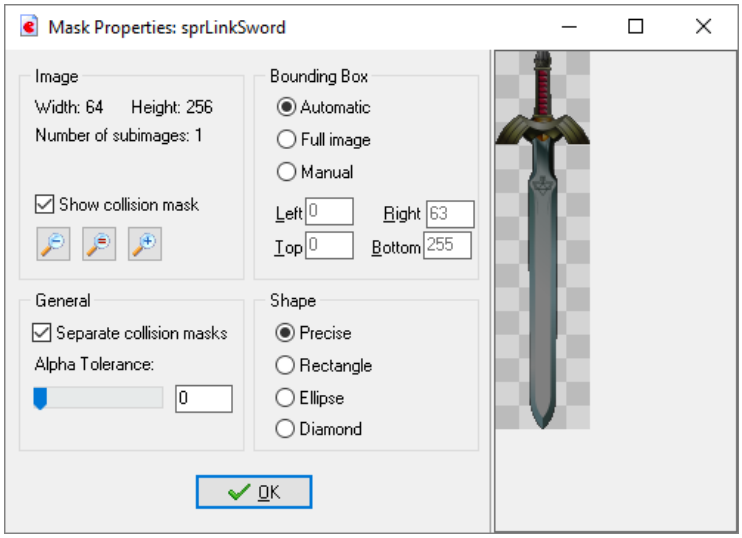


Fonte: Editada pelo autor

A opção *Separete Collision Masks* faz com que cada quadro da imagem tenha uma máscara de colisão diferente que pode ser configurada clicando em *Modify Mask*.

O botão *Modify Mask* (Figura 3.9) permite alterar as dimensões da máscara, grau de transparência para considerar um pixel sólido, bem como o formato da máscara caso a colisão não seja precisa, pois, o padrão é um retângulo. Além dessa, há a forma circular e a que tem formato diamante (Losango).

Figura 3. 9 - Modificando a máscara de colisão



Fonte: Imagem capturada diretamente do programa

3.2.2. Objetos, Instâncias e seus Eventos

Figura 3. 10 - Capa do vídeo 'Objeto, Instâncias e seus Eventos'

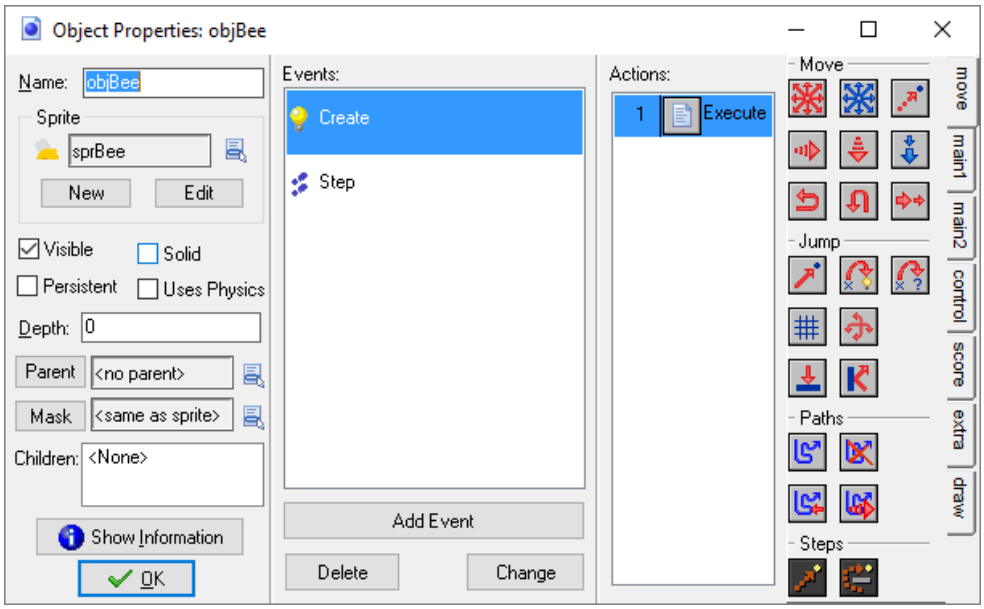
Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=2WsUp7W_dx8)

Nesse vídeo, Figura 3.10, é apresentada a estrutura de dados mais abstrata do programa.

3.2.2.1. Objetos

Um objeto é o tipo de estrutura que pode ser mais abstrato dentro do GameMaker: Studio. Isso porque um objeto pode ser quase qualquer elemento dentro do jogo. Seria equivalente à uma classe do *C++* ou registro do *C*, uma espécie de forma que pode ser replicada em várias instâncias.

Figura 3. 11 - Configuração de um objeto



Fonte: Imagem capturada diretamente do programa

Seguindo a Figura 3.11, o campo *Name* segue as mesmas regras descritas para o campo de mesmo nome para uma *Sprite*, assim como para qualquer outro recurso.

Na seção *Sprite* deve-se escolher qual sprite anexar ao objeto e ainda tem dois atalhos rápidos: Um para criar uma nova *Sprite* e outro para editar a *Sprite* atual.

A opção *Visible* deixa o objeto visível ou não. *Solid* ativa a propriedade de colisão para um objeto que não pode ser atravessado (Ex: paredes). *Persistent* não deixa o objeto ser recriado nas transições de cenários. Já a *Uses Physics* permite que sejam ativadas propriedades físicas avançadas no objeto, desconsiderando o sistema normal.

No campo *Depth* é definida a ordem de desenho para o objeto. Geralmente todos estão em 0, logo as instâncias de objetos mais novos ficam a frente dos velhos. Segue o mesmo princípio de *camadas* [43]dos *softwares* gráficos.

Visto que é um sistema orientado a objetos, há a característica de herança, que aqui recebe o nome de *Parent*. Basta selecionar o objeto que deseja herdar características.

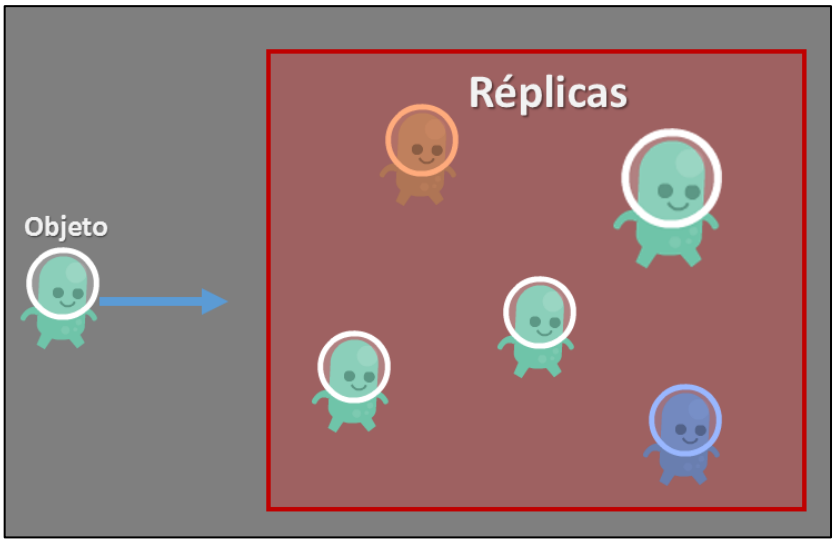
Em *Mask*, é definida qual *Sprite* está sendo usada como máscara de colisão, por padrão é a *Sprite* já anexada anteriormente.

Já em *Childrem* aparecem os objetos que tem esse objeto marcado como parente e, portanto, são seus filhos na herança. *Show Information* mostra um resumo do objeto.

3.2.2.2. Instâncias

Já que um objeto é como se fosse um modelo, as instâncias são resultado da aplicação desse modelo, ou seja, são réplicas (Figura 3.12) com características diferentes (Posição, tamanho, cor, velocidade, etc.), onde cada uma possui um *id*[44] diferente.

Figura 3. 12 - Instâncias: Réplicas de um objeto

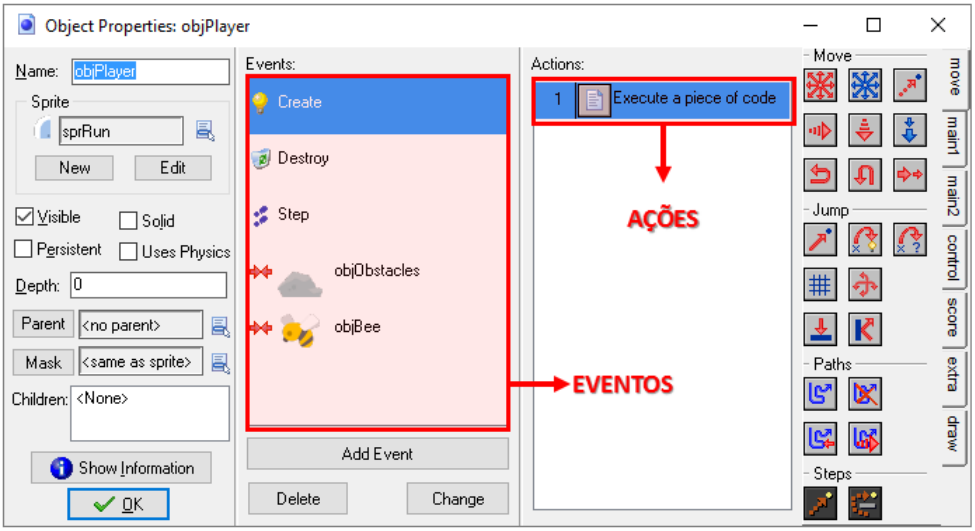


Fonte: Editada pelo autor

[43] Imagem que sobrepõe outra, mas que no momento da edição pode ser destacada para edição individual.
[44] Variável de leitura que guarda um número inteiro único para cada réplica.

3.2.2.3. Eventos

Figura 3. 13 - Eventos e ações



Fonte: Imagem capturada diretamente do programa

Toda instância executa ações, e elas são executadas através de eventos que ocorrem de várias maneiras diferentes (Figura 3.13). Assim, eventos são estruturas temporais que nos dizem quando e onde os códigos serão executados. No Quadro 3.2 abaixo, são apresentados os eventos mais usados:

Quadro 3. 2 - Eventos básicos

Evento	**Descrição**
Create	Ocorre apenas uma vez quando a instância é criada, geralmente usado para iniciar atributos.
Step	Ocorre o tempo todo (Por padrão são 30 vezes por segundo) durante a existência da instância. Serve para criar a lógica fazendo verificações e movimentações.
Draw	Ocorre o tempo todo (Por padrão são 30 vezes por segundo) durante a existência da instância. Neste evento são feitos desenhos (Imagens, textos e etc.).
Collision	Ocorre enquanto o objeto colide com o objeto selecionado.
Destroy	Ocorre apenas uma vez no momento em que o objeto é destruído.

Fonte: Referência do *Game Maker*
(http://docs.yoyogames.com/source/dadiospice/000_using%20gamemaker/events/index.html)

3.2.3. Rooms e Views

Figura 3. 14 - Capa do vídeo ‘Rooms e Views’

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=4BwoHvGkY2s)

Aqui é explanada a estrutura *room* e suas características, principalmente o uso de *views*, Figura 3.14.

3.2.3.1. Rooms

É uma estrutura diretamente ligada à apresentação gráfica do jogo, geralmente exibida em uma *janela* [45]ou em *tela cheia*[46]. Uma *room*, pode ser um mapa, um cenário (Figura 3.15), uma tela de menu, ou qualquer elemento gráfico que se disponha em uma tela, ou seja, no *GameMaker: Studio* é um espaço digital onde são posicionadas as instâncias dos objetos.

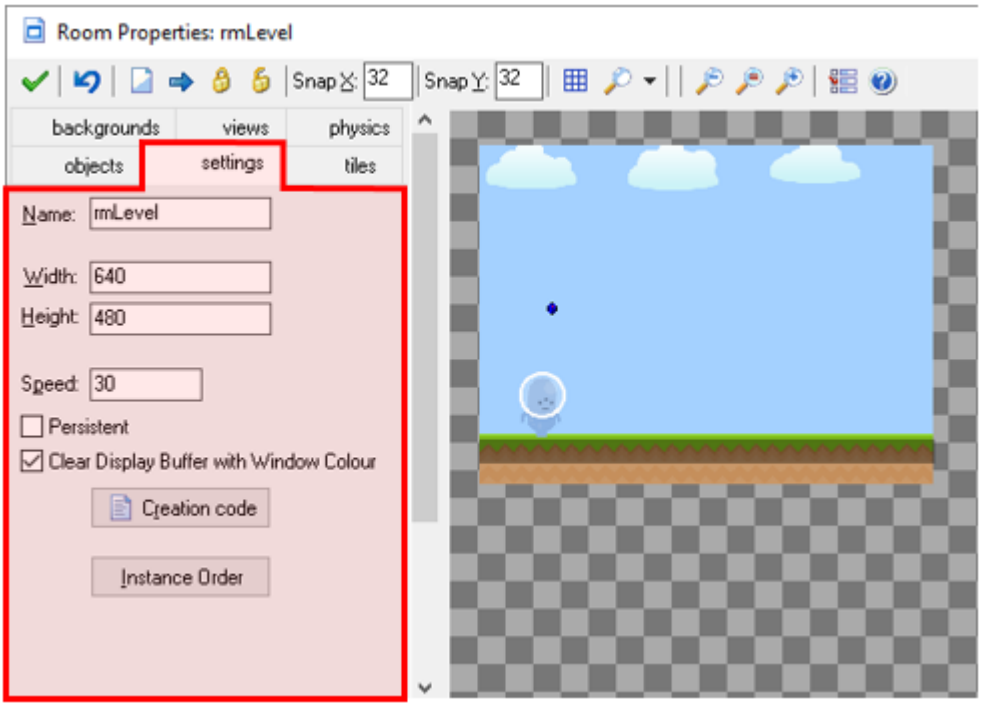
[45] Termo utilizado para retângulos gráficos executando um programa, como no *Microsoft Windows*.
[46] Quando uma aplicação se utiliza de todo espaço de uma tela gráfica para apresentar seu conteúdo.

Figura 3. 15 - Jogo *Bomberman 2*

Fonte: *SydLexia.com* (http://www.sydlexia.com/snes100/snes54.htm)

3.2.3.2. Configurações Gerais

Figura 3. 16 - Configurações da *Room*



Fonte: Imagem capturada diretamente do programa

Seguindo a Figura 3.16, em *Name* é definido um identificador para a mesma. Logo em seguida alteramos suas dimensões *Width* (Largura em pixel) e *Height* (Altura em pixels).

Em *Speed* define-se a taxa de atualização das instâncias, que também é a taxa de atualização dos desenhos na tela. Ou seja, a tela será atualizada 30 vezes por segundo (Em condições ideais, onde o computador não seja tão exigido, caso contrário essa taxa cai).

A opção *Persistent* faz com que a *room* permaneça com seu estado inalterado caso haja troca de *rooms*. Resumindo, as instâncias estarão no mesmo local e com as mesmas configurações caso o jogador seja levado à outra *room* e em seguida volte para a anterior. Recurso usado principalmente para pausa no jogo, onde a *room* do cenário é persistente e a com menu de pausa não.

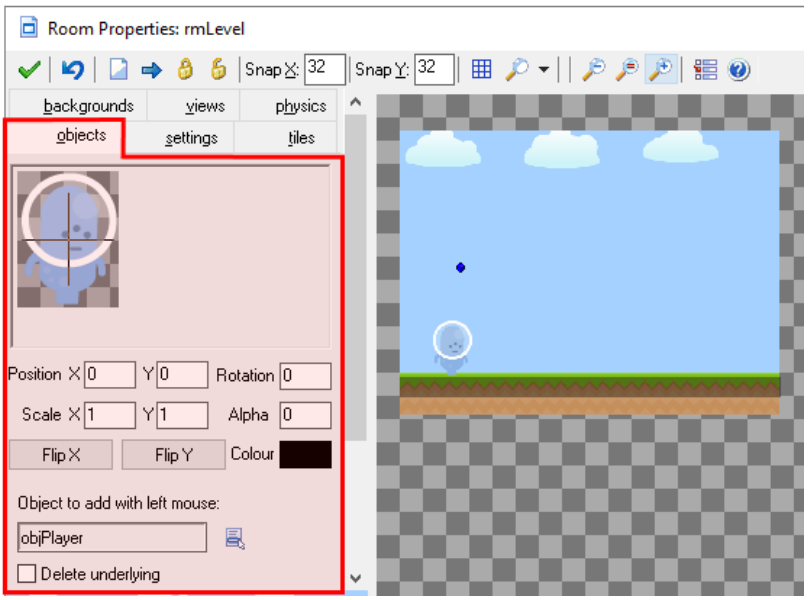
Clear Display Buffer with Window Colour faz com que a tela seja limpa (Toda a tela preenchida com apenas uma cor) automaticamente com a cor de fundo da janela. É recomendado deixar essa opção sempre ativa.

O botão *Creation Code*, executa um trecho de código não vinculado a nenhum objeto e pode ser usado justamente para criá-los, caso tenha feito um algoritmo de criação de instâncias inicial. Também pode ser aproveitado para definir parâmetros globais iniciais.

Instance Order permite trocar a ordem de criação dos objetos posicionados na *room*. Útil para controlar sequências de execução, onde uma instância só executa um código caso outra existir antes, ou ainda para sobreposição de desenhos.

3.2.3.3. Adicionando instâncias na *room*

Figura 3. 17 - Inserindo instâncias na *room*



Fonte: Imagem capturada diretamente do programa

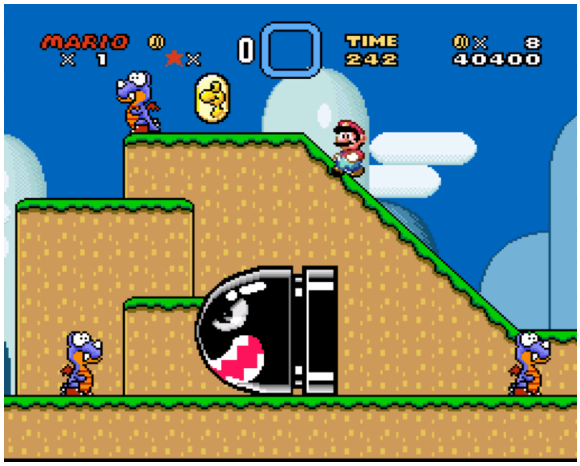
Acompanhando a Figura 3.17, deve ser selecionado um objeto em *Object to add with left mouse*, após isso basta usar o botão esquerdo do mouse para adicionar instâncias clicando na *room*. A opção *Delete underlying* faz com que instâncias não sejam sobrepostas ao clicar em uma posição já ocupada.

Acima dessas opções temos um painel para mudar o estado inicial de uma instância que esteja selecionada na *room* onde há como alterar sua posição, escala, rotação, transparência e cor, além de poder espelhar a sprite verticalmente ou horizontalmente.

3.2.3.4. Ativando Views

Quando queremos cenários maiores do que a tela pode exibir usamos um recurso parecido com uma câmera, chamado aqui de View. Ou seja, um recorte de tela, um bom exemplo é o jogo *Super Mario World*[47] como ilustra a Figura 3.18, onde o personagem se move em todas as direções e a *View* o acompanha:

Figura 3. 18 - Super Mario World

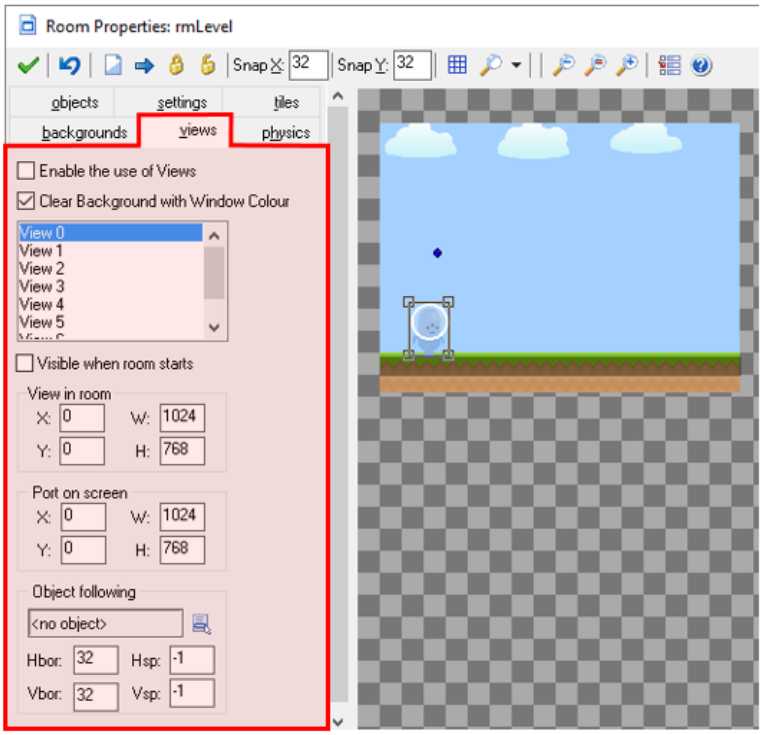


Fonte: *Nintendo Blast* (http://www.nintendoblast.com.br/2015/10/super-mario-world-2-yoshis-island-snes.html)

No *GameMaker: Studio* na aba *Views* ao editar uma *room* o mesmo processo pode ser feito(Figura 3.18).

[47] Jogo do console Super Nintendo.

Figura 3. 19 - Configurando uma View



Fonte: Imagem capturada diretamente do programa.

Seguindo a Figura 3.19, há a opção *Enable the use of Views* que ativa a funcionalidade de usar a câmera. Logo abaixo *Clear Background with Window Colour* que preenche o fundo da tela com uma cor especifica.

São 8 *views* no total (De 0 a 7), selecionando uma temos que configurá-la com as opções inferiores. O uso múltiplo de *views* é comumente utilizado em telas dividas com *multiplayer local*[48] ou para criar mini mapas dos cenários.

Em *Visible when room starts* é apontada que a *view* selecionada deve estar ativa assim que *room* iniciar.

Na seção *View in room* configuramos o tamanho da *view* e sua posição inicial. Em *Port on Screen* é definido o tamanho e a posição da janela, geralmente basta repetir os dados da seção anterior ou especificar um tamanho diferente, onde a *View* que foi projetada será redimensionada.

Object Following define o objeto que a *view* vai seguir, na maioria dos casos é o personagem principal. Nos campos *Hbor* e *Vbor* é definida a borda que ficará envolta do objeto para que quando este estiver próximo as extremidades da tela, ela se mova. Em *Hsp* e *Vsp*, são

[48] Partida onde há mais de um jogador utilizando a mesma tela, mas cada um com controles individuais.

definidas a velocidade em que a *view* se movimenta, os valores –1 indicam que o movimento é instantâneo, ou seja, a *view* estará sempre no objeto.

3.2.4. Backgrounds e Tiles

Figura 3. 20 - Capa do vídeo 'Background e Tiles'

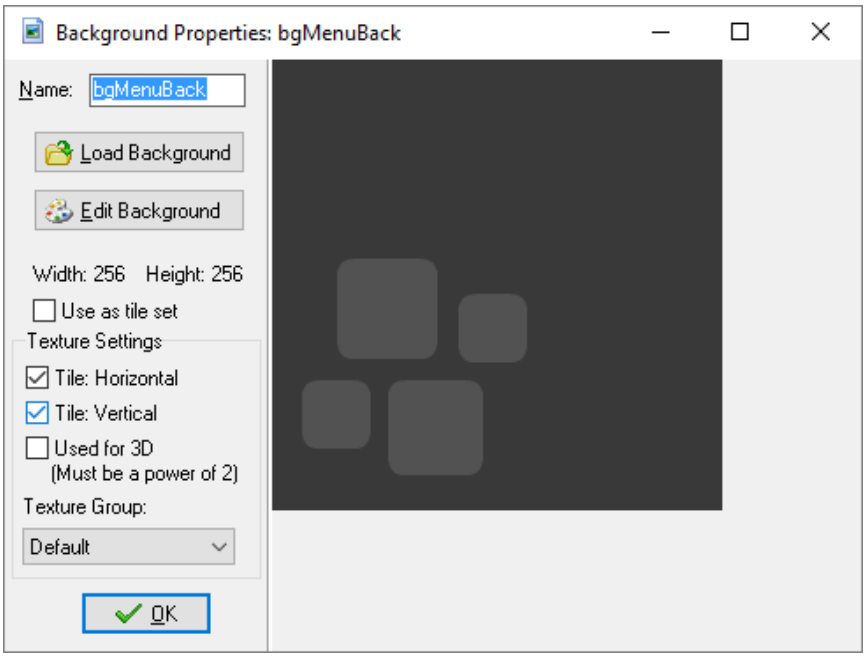
Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=nPTUEwlz5Vk)

No vídeo intitulado *Background e Tiles* (Figura 3.20) tratamos de elucidar como funcionam as seguintes estruturas de imagem: *backgrounds* e as *tiles*.

3.2.4.1. Backgrounds

São imagens, geralmente grandes e estáticas (Sem animação) usadas para preencher os fundos das *rooms*.

Figura 3. 21 - Configuração de um *background*



Fonte: Imagem capturada diretamente do programa

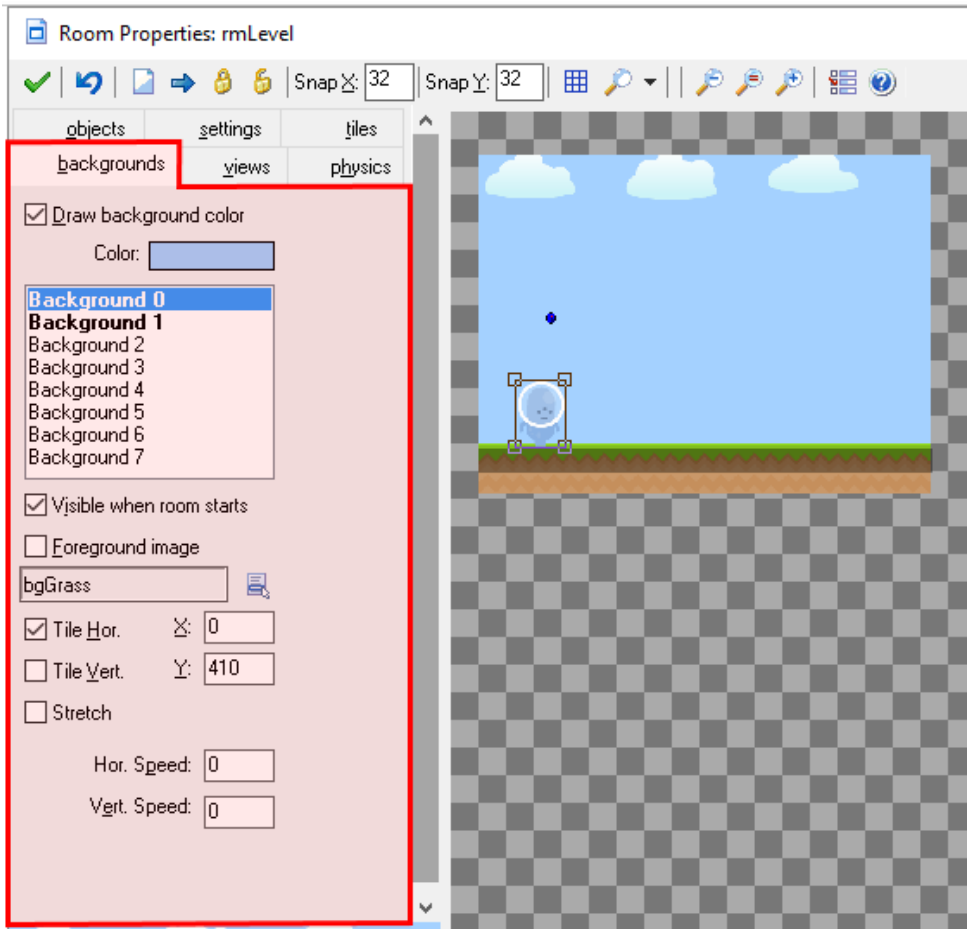
Seguindo a Figura 3.21, no campo *Name* o identificador do *background* é escolhido. Com o botão *Load Background* pode-se carregar uma imagem. Ao clicar nessa opção o usuário pode escolher qual imagem anexar a partir do seu computador. Vários formatados são suportados (*GIF*, *JPG*, *PNG*, *BMP* e etc.). *Edit background*, abre a imagem no editor gráfico.

Abaixo temos as informações de Largura e Altura (Em pixels) e na seção *Texture Settings* é determinada a finalidade de uso. Se a imagem vai se repetir Horizontalmente ou Verticalmente, como uma textura, basta marcar as opções *Tile: Horizontal* e *Tile: Vertical*. E no caso de ser usada para desenhos 3D, a opção *Uded for 3D*, lembrando que duas dimensões devem ser potências de 2.

Para usar um *background* há diversos módulos, porém, o mais básico se encontra no editor de *rooms* (Figura 30).

Seguindo a Figura 3.22, temos como primeira opção *Draw background color*, por padrão essa opção sempre está marcada, basta apenas escolher a cor de fundo. Em seguida deve-se escolher até 8 *backgrounds* por *room* (De 0 a 7) e baixo suas respectivas configurações.

Figura 3. 22 - Usando backgrounds



Fonte: Imagem capturada diretamente do programa

Visible when room starts indica que o *background* será visível desde quando a room iniciar, *Foreground image* faz com que o *background* fique à frente de todos os outros elementos gráficos do jogo e logo abaixo podemos selecionar a imagem que usaremos.

Tile Hor e *Tile Vert* indicam se a imagem irá se repetir na horizontal e/ou na vertical. Em *X* e *Y* trocamos a posição em que o *background* é iniciado na room e em *Stretch* marcamos se a imagem deve ser esticada na *room*.

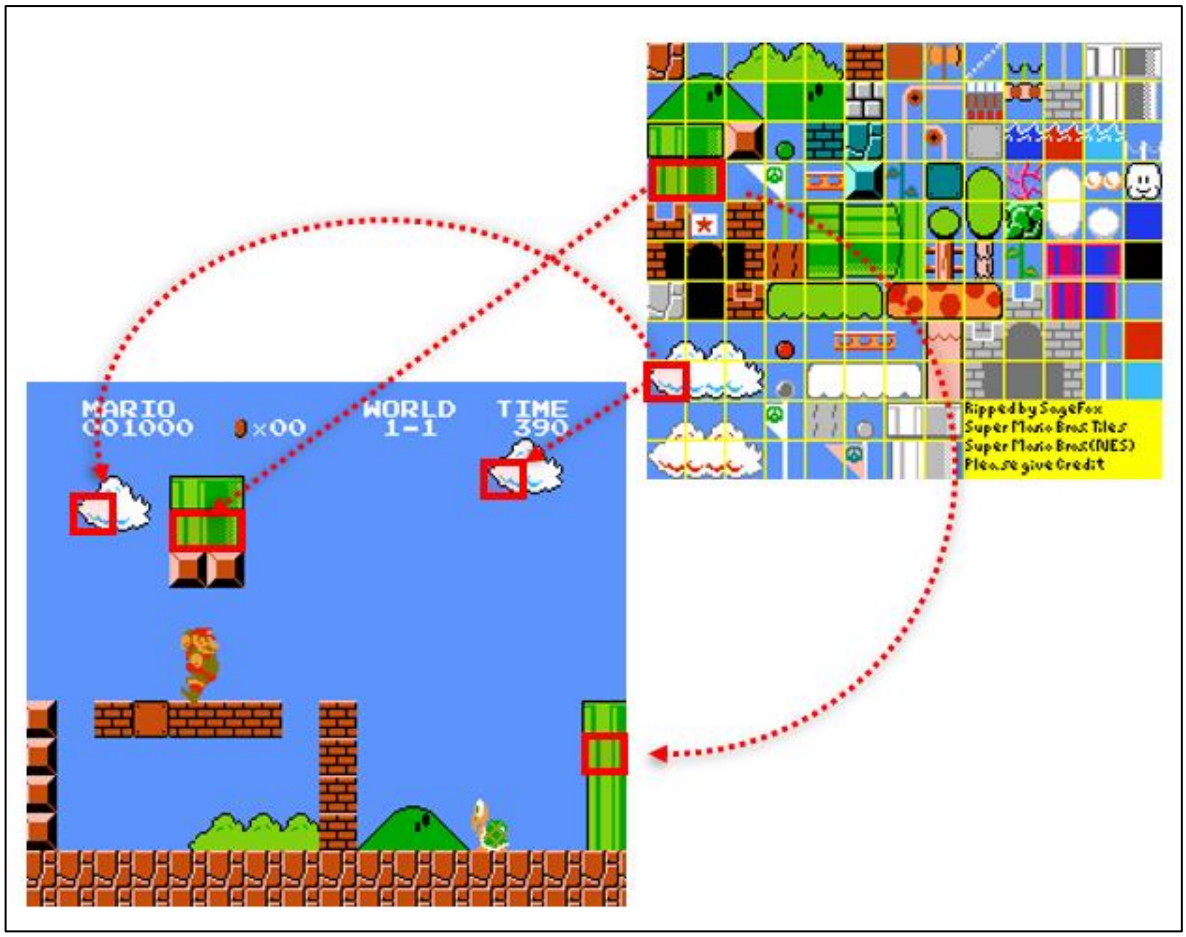
Hor. Speed e *Vert. Speed* definem velocidade de movimento do *background*. Caso haja repetição vertical ou horizontal cria ilusão de movimento, mesmo em uma tela estática.

3.2.4.2. Tiles

Apesar de ter uma função diferente, as imagens desse tipo também ficam na pasta *Backgrounds* da árvore de recursos e isso pode confundir de início, mas todo *background* pode

ser uma *tile*, assim como toda *tile* pode ser um *background*. O que muda é apenas a finalidade. Enquanto com os *backgrounds* usa-se toda a imagem, nas tiles são pedaços recortados delas. Na Figura 31 nota-se que o mesmo pedaço de imagem se repete em várias partes. Isso poderia ser feito usando uma sprite e um objeto posicionado na room, mas como é uma imagem estática e que não necessita de eventos e ações, basta usar *tiles* (Figura 3.23).

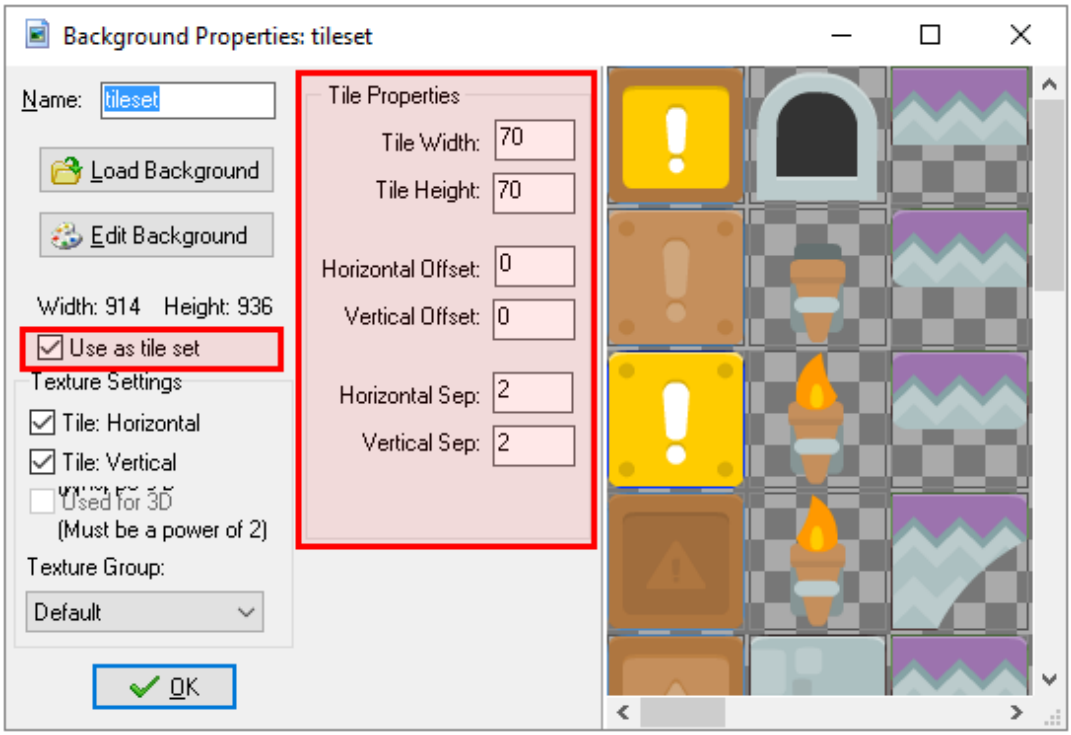
Figura 3. 23 - Exemplificação do uso de tiles



Fonte: Imagem editada pelo autor (www.vizzed.com; www.bhivecanvas.com)

Nota-se na Figura 3.24, além dos mesmos campos e botões de um *background*, caso a opção *Use as tile set* estiver marcada é aberta uma nova seção, a *Tile Properties* onde deve-se configurar como será recortada a imagem. Nos dois primeiros parâmetros são escolhidas as dimensões do recorte, em seguida é possível mover todas as caixas de recorte tanto verticalmente quanto horizontalmente. Por último, destaca-se a separação entre as caixas de recorte, no caso de os elementos não estarem colados.

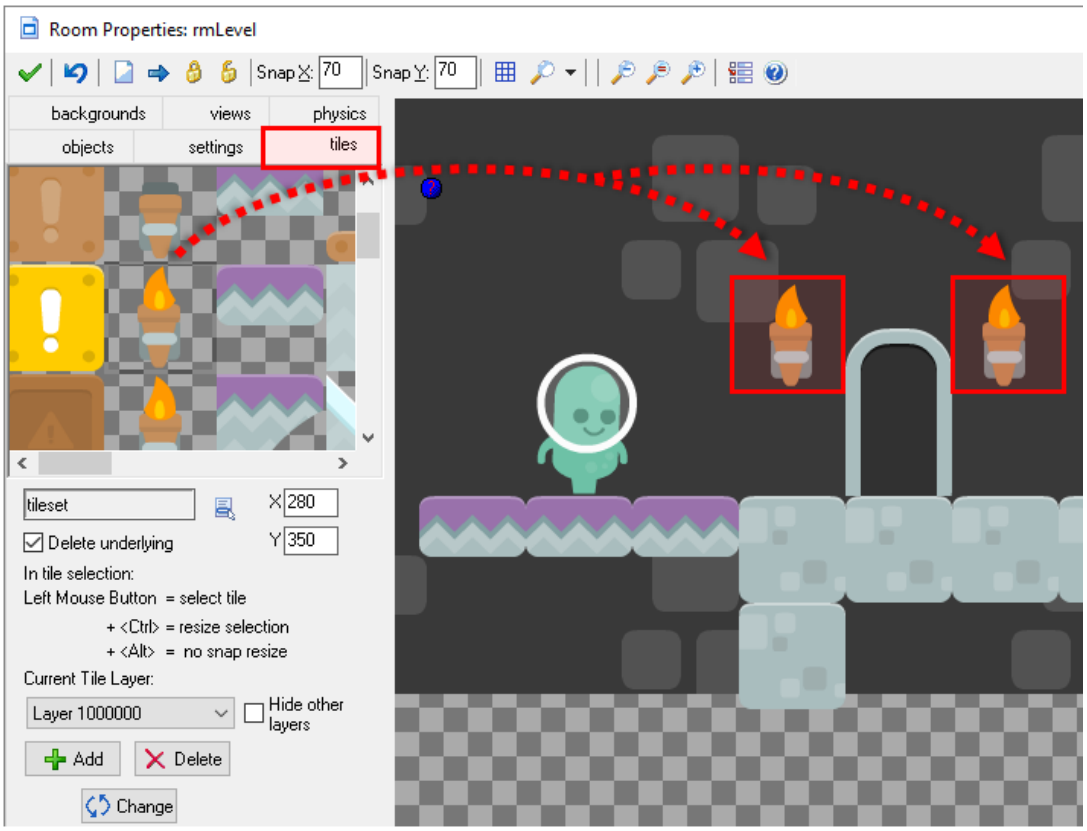
Figura 3. 24 - Configuração da tile



Fonte: Imagem capturada diretamente do programa

Com a *tile* devidamente configurada, basta abrir o editor de *rooms* e posicionar os recortes que foram feitos através da aba *tiles*.

Figura 3. 25 - Inserindo tiles



Fonte: Imagem capturada diretamente do programa

Ao escolher a tile, Figura 3.25, um painel é revelado com a imagem que foi configurada, onde basta clicar e escolher os pedaços que irão ser posicionados na room. Os mesmo comandos e atalhos usados para posicionar os objetos servem aqui também.

A novidade é a seção *Current Tile Layer* onde a estratégia de camadas para agrupar as *tiles*, pode ser utilizada. Por padrão a profundidade de desenho é 100000, ou seja, bem ao fundo da tela, mas a frente dos *backgrounds*. É possível adicionar vários tiles, e caso desejarmos colocar tiles sobrepostos sem o perigo de apagar ou mover o que esteja abaixo, basta criar um novo *Layer* com profundidade menor (Ex: 999999). Enquanto se está adicionando *tiles* em um *Layer* os outros ficam bloqueados, impedindo exclusões e movimentos acidentais. Lembrando também que o *Layer* segue o mesmo princípio do *Depth* dos objetos, logo um pode sobrepor o outro e vice-versa sendo que valores menores aparecerão por cima e os maiores ao fundo.

3.2.5. Músicas e Sons

Figura 3. 26 - Capa do vídeo 'Músicas e sons'

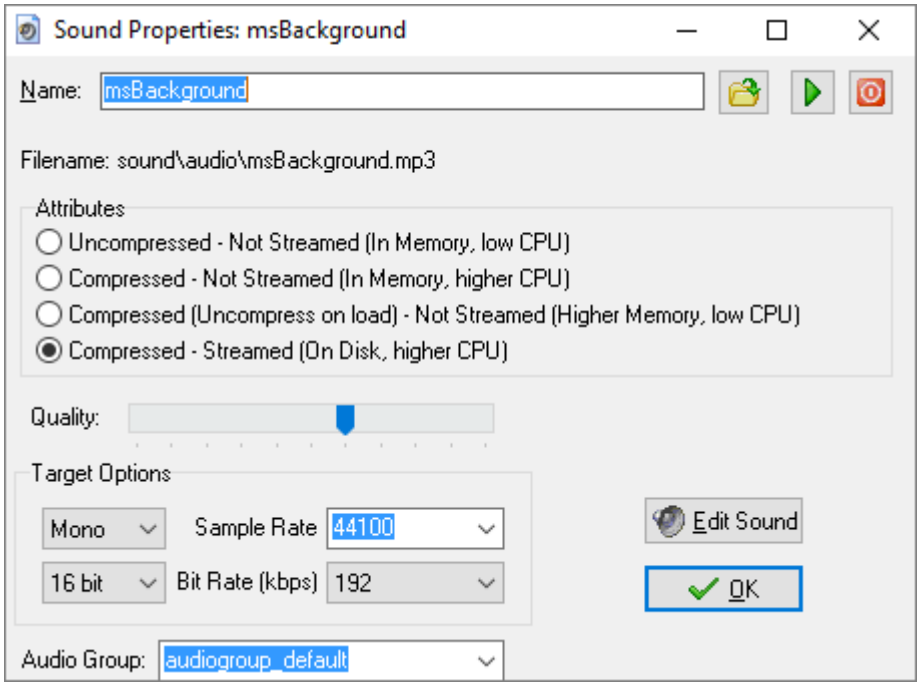
Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=uHaG220aIVk)

No vídeo *Músicas e Sons* (Figura 3.26) é visto como armazenar e reproduzir sons e músicas no *GameMaker: Studio*.

3.2.5.1. Carregando arquivos de áudio

Para o campo *Name* (Figura 3.27) definimos um identificador para o áudio, clicando no ícone com símbolo de uma pasta podemos escolher o arquivo a ser carregado. Também há dois botões de reprodução de áudio para teste logo ao lado.

Figura 3. 27 - Carregando áudio



Fonte: Imagem capturada diretamente do programa

É recomendado importar arquivos *MP3*[49] para músicas de fundo e arquivos *WAV*[50] para efeitos sonoros. Isso não significa que ao rodar o jogo os arquivos estarão nesses formatos. Isso porque ocorre uma conversão antes da compilação. Isso é elucidado na seção *Atribbutes*. Essas opções definem como o *GameMaker: Studio* irá gerenciar os sons. Vejamos o que faz cada opção:

- Não comprimido - Sem *Stream*[51] (É carregado diretamente na *RAM* [52]e usa pouco a *CPU*[53]); O formato de conversão vai ser *WAV*, pois este não precisa de decodificação. E sendo assim o executável terá um tamanho maior;

- Comprimido - Sem *Stream* (É carregado diretamente na *RAM*, mas demanda muita *CPU* para decodificação). O formato de conversão é *OGG*[54], que nada mais é do que *WAV* comprimido. O executável ficará com tamanho menor;

- Comprimido - Sem *Stream* e Descomprimido ao carregar o jogo (O arquivo *OGG* é descomprimido e transformado em *WAV* que é carregado na memória, requer muita *RAM* e pouca *CPU*). O formato em disco é *OGG* e passa a ser *WAV* quando executado. O executável ficará com tamanho menor;

- Comprimido - Com *Stream* (O arquivo é decodificado em pequenas partes diretamente do disco e prontamente executado). O formato de conversão é *OGG* ou *MP3* (Depende da plataforma que está exportando), essa opção é usada para músicas de fundo.

O *GameMaker: Studio* já usa por padrão a primeira opção para arquivos *WAV* e a quarta para *MP3*. Por isso é extremamente recomendado abrir arquivos já nesses formatos.

[49] MPEG1 Layer 3, é um formato de áudio digital compactado.
[50] Waveform Audio File Format, é um formato digital de áudio sem compactação e sem perda de qualidade.
[51] Tecnologia que permite executar um arquivo logo enquanto o mesmo é carregado.
[52] Memória que armazena dados enquanto estiver energizada, porém mais rápida que os Hard Drives.
[53] Central Processing Unit, é o chip responsável por realizar a maior parte dos cálculos do computador.
[54] Ogg Vorbis é um formato de áudio comprimido.

A seção *Quality* modifica a qualidade de reprodução do áudio, assim como a seção *Target Options*, mas está última varia de acordo com a plataforma de destino. Alterar estas opções é recomendado apenas para quem tiver conhecimentos prévios de formatos e configurações de áudio, caso não, deixar como estão.

3.2.5.2. Reproduzindo áudio com GML

Ao contrário dos outros recursos que podem ser testados diretamente em uma *room*, com os arquivos de áudio precisaremos usar códigos para avalia-los em jogo, ainda que *Game Maker Language* (*GML*) só seja detalhadamente estudada no subcapítulo 3.3. A seguir veremos as 4 funções mais básicas para reprodução de áudio:

3.2.5.2.1. Função: audio_play_sound

Figura 3. 28 - Função de reprodução de áudio

```
action
1 audio_play_sound(msBackground, 1, true)
```

Fonte: Imagem capturada diretamente do programa

Essa função é responsável por executar os sons (Figura 3.28). No primeiro parâmetro colocamos o identificador do arquivo. No segundo definimos a prioridade de execução (Geralmente quando há muitos sons para serem tocados e a máquina não suporta executar todos, apenas os de prioridade mais alta são reproduzidos), sendo que não há escala fixa de valor, ou seja, o som com prioridade mais alta é o máximo e o com menor é o mínimo. Por fim, temos o parâmetro de repetição, sinalizando verdadeiro a música irá repetir ao chegar no fim, e sinalizando falso a execução terminará ao fim.

3.2.5.2.2. Função: audio_stop_sound

Figura 3. 29 - Função de interrupção de reprodução

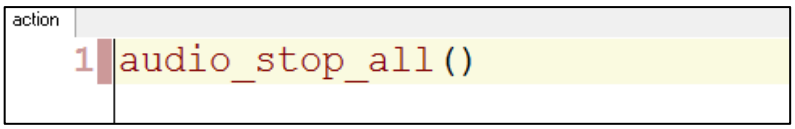
```
action
1 audio_stop_sound(msBackground)
```

Fonte: Imagem capturada diretamente do programa

O código acima interrompe a reprodução do som especificado no primeiro e único parâmetro apenas, mantendo os demais sons em sua execução normal (Figura 3.29).

3.2.5.2.3. Função: audio_stop_all

Figura 3. 30 - Função de interrupção geral de áudio

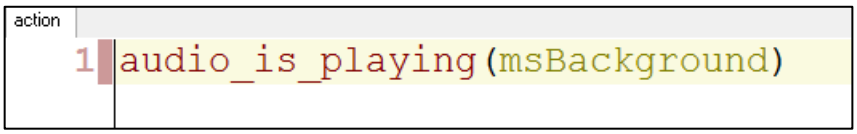


Fonte: Imagem capturada diretamente do programa

Essa instrução cessa a reprodução de todos os sons, sem exceção (Figura 3.30). Não há parâmetros para essa função.

3.2.5.2.4. Função: audio_is_playing

Figura 3. 31 - Função de verificação de reprodução



Fonte: Imagem capturada diretamente do programa

Verifica se o som especificado no primeiro parâmetro está em execução (Figura 3.31). Ao contrário das outras funções supracitadas esta retorna um valor verdadeiro, caso o som esteja tocando no momento da verificação e um valor falso se não.

3.3. Conhecendo a Game Maker Language (GML)

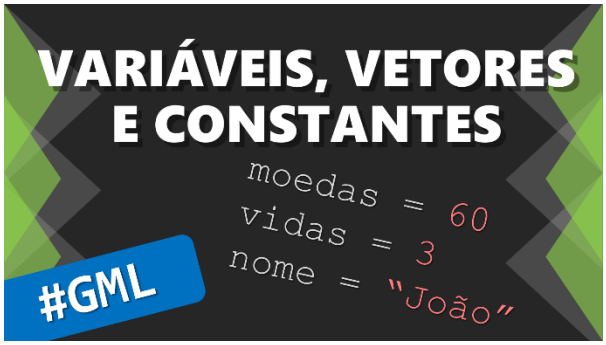
O *GameMaker: Studio* utiliza uma linguagem proprietária de programação chamada *Game Maker Language* (*GML*). Ela é uma linguagem interpretada que se assemelha a JavaScript e C++ que utiliza o paradigma de orientação a objetos. A vantagem de utiliza-la é

escrever o código apenas uma vez e exportar o jogo para diversas plataformas diferentes, especificadas no CAPÍTULO 2.

No decorrer deste subcapítulo esta linguagem e suas estruturas de programação serão particularizadas.

3.3.1. Variáveis, Vetores e Constantes

Figura 3. 32 - Capa do vídeo 'Variáveis, vetores e constantes'



Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=87LlVqHgWTk)

No vídeo (Figura 3.33) são detalhadas as estruturas de dados mais básicas da *GML*.

3.3.1.1. Variáveis

São identificadores que apontam para determinado local na memória. As variáveis são usadas para carregar informações na memória. Essa informação pode ser tanto numérica (Número real) como também textual (Cadeia de caracteres ou *String*[55]).

Um identificador de variável (O nome pelo qual é chamado) pode conter caracteres alfanuméricos desde que respeite algumas limitações (Quadro 3.3). Para aplicar um valor à uma variável basta usar o operador de atribuição (=) seguindo de uma expressão (Valor).

Quadro 3. 3 - Limitações de nomenclatura de identificadores

Identificador de variável	**Comentário**
`moedas = 100`	Correto.
`1Armas = 50`	Errado. Nenhum identificador pode iniciar com números.

[55] Sequência de caracteres alfanuméricos que formam textos.

`_VIDA_ = 1`	Correto. Pode se utilizar o sublinhado em qualquer posição.
`Item05 = 3`	Correto.
`_maçãs = 99`	Errado. Acentos e cedilha são proibidos.
`sprArm = 0`	Errado. Caso sprArm seja um identificador de uma *Sprite*.
`random = 10`	Errado. O identificador é usado por uma função, ou seja, está indisponível.
`Salto duplo = 1`	Errado. Não é possível inserir espaços.

Fonte: Referência da *GML* (https://docs.yoyogames.com/source/dadiospice/002_reference/001_gml%20language%20overview/variables/index.html)

Apesar de haverem vários tipos de variáveis, o *GameMaker: Studio* trabalha com dois tipos gerais mais frequentemente:

Figura 3. 33 - Tipos de variáveis

```
1  // Tipo numérico
2
3  // Inteiro
4  balas = 15
5
6  // Ponto flutuante
7  valor_de_pi = 3.1415
8
9  // Tipo textual
10
11 frase = "O céu está limpo hoje"
```

Fonte: Imagem capturada diretamente do programa

O tipo numérico (*Double*) pode armazenar números inteiros e números de ponto flutuante (Figura 3.33). Já o tipo textual (*String*) pode conter letras, palavras e textos (Figura 3.33), estes valores devem sempre estar entre aspas simples ou duplas.

Tirando esses dois tipos básicos ainda há outros cinco tipos: vetores, *booleanos*[56], *ponteiros*[57], *enumeradores* [58]e *não definidos*[59]. Serão vistos os dois primeiros no decorrer dos capítulos.

Todas as quatro operações matemáticas básicas (Somar, subtrair, multiplicar e dividir) estão disponíveis para atuar nas variáveis e demais valores, com o acréscimo do módulo e da divisão que resulta em um inteiro (Quadro 3.34).

[56] Valores numéricos na álgebra de Boole.
[57] Indicam uma posição na memória.
[58] Valores fixos enumerados em sequência.
[59] Não são de nenhum tipo, geralmente retornam em algum erro.

Figura 3. 34 - Operações matemáticas

Operador	**Código GML (A = 5; B = 3; C = 2)**	**Valor de N**
Soma (+)	`N = 5 + 3`	8
Subtração (-)	`N = A – 2`	3
Multiplicação (*)	`N = C * 2`	4
Divisão (/)	`N = A / C`	2.5
Módulo (mod)	`N = A mod C`	1
Divisão com inteiro (div)	`N = A div C`	2
Várias operações	`N = ((4 * 10) / C) – 10`	10
Soma relativa (+=), onde N = 3	`N += 1`	4
Subtração relativa (-=), onde N = 9	`N -= 4`	5
Multiplicação relativa (+=), onde N = 2	`N *= 5`	10
Divisão relativa (+=), onde N = 30	`N /= 3`	10
Soma rápida, onde N =10	`N ++`	11
Subtração rápida, onde N =10	`N --`	9

Fonte: Quadro criado pelo autor

Também é possível fazer operações com *strings*, sendo a principal delas a concatenação onde cria-se uma nova *string* a partir de outras. Para isso basta utilizar o operador matemático de soma (Quadro 3.4).

Quadro 3. 4 - Concatenação de Strings

Código GML de concatenação	**Valor de S**
`S = "Bom " + "dia!"`	`"Bom dia!"`
`B = "cinco"` `S = "Eu tenho " + B + " balas!"`	`"Eu tenho cinco balas!"`
`B = 5` `S = "Eu tenho " + string(B) + " balas!"`	`"Eu tenho 5 balas!"`

Fonte: Quadro criado pelo autor

Note que no Quadro 3.4 foi utilizada a função *string*, ela faz a conversão de tipos numéricos em *strings* para que seja possível anexa-los a um texto. É importante saber converter os dados para não gerar erros. O Quadro 3.5 possui as conversões mais convencionais:

Quadro 3. 5 - Conversões de dados

Código GML	**Operação de conversão**	**Valor de A**
`A = string(25.6)`	Converte número real em *string*	`"25,6"`
`A = real("5,1")`	Converte *string* em número real	`5.1`
`A = ord("A")`	Converte caractere em número da *tabela ASCII*[60]	`65`

Fonte: Quadro criado pelo autor

Vale notar que nas conversões entre números reais e *strings*, o símbolo que separa as casas decimais é trocado, logo convém usar vírgulas em *strings* que contenham valores reais, antes de converter em valor numérico.

3.3.1.2. Vetores ou Arrays

Estrutura semelhante a variável, porém nesta é possível adicionar vários valores e depois acessá-los através de índices sem mudar o identificador, se assemelha a uma lista e é chamada também de *array* de uma dimensão (Figura 3.35).

Figura 3. 35 - Definindo um vetor

```
1 // Define o valor dos quatro primeiros indices
2 nomes[0] = "João"
3 nomes[1] = "Maria"
4 nomes[2] = "José"
5 nomes[3] = "Ana"
6
7 // Guarda o valor 'José' na variável 'nome_escolhido'
8 nome_escolhido = nomes[2]
```

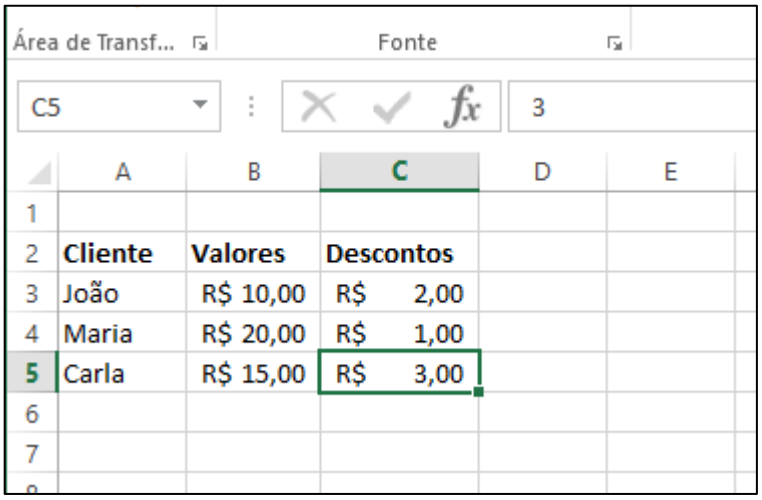
Fonte: Imagem capturada diretamente do programa

[60] Um esquema de codificação que atribui valores numéricos a caracteres visando padronizar a troca de dados entre computadores.

No *GameMaker: Studio* o programador não precisa se preocupar em redimensionar o vetor toda vez que adicionar um valor (Como em *C*, por exemplo), isso é feito dinamicamente. A única preocupação é acessar um índice que ainda não foi definido.

Ainda há suporte para vetores de duas dimensões, onde temos dois índices. Para facilitar o entendimento basta lembrar-se das planilhas do Microsoft Excel[61]. A mudança fica por conta de que há dois índices números no vetor bidimensional e no Excel as colunas são representadas por letras (Figura 3.36).

Figura 3. 36 - Planilha do Excel



Fonte: Imagem capturada diretamente do programa

Um bom exemplo de uso seria para seleção de idiomas dentro do jogo (Figura 3.37), onde guarda-se cada um dos os textos de um idioma em uma coluna diferente. Depois basta guardar um valor seletivo em uma variável que alternará facilmente entre os idiomas.

[61] Programa de gerenciamento de planilhas.

Figura 3. 37 - Seleção de idioma através de um vetor bidimensional

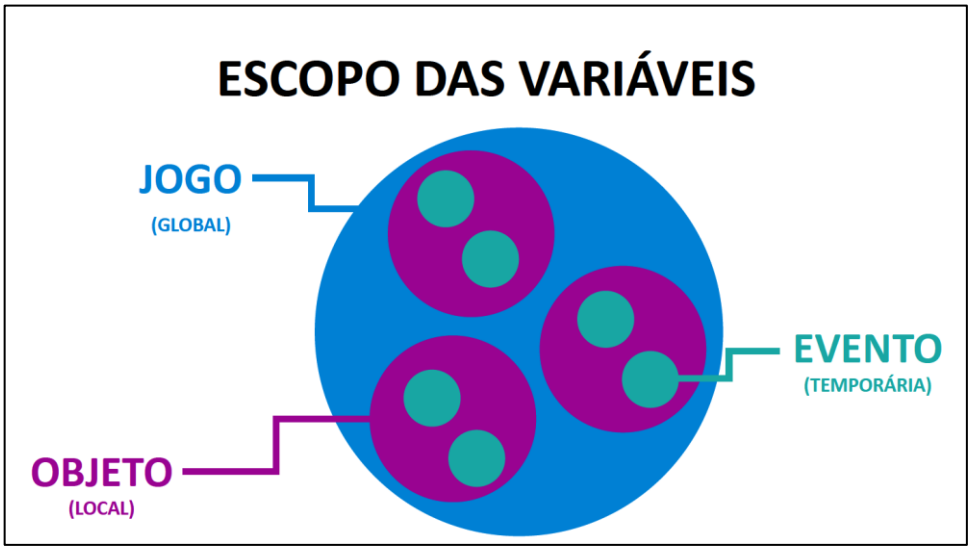
```
1  // Frases em Português - Coluna 0
2  frases[0,0] = "Seja bem-vindo!"
3  frases[0,1] = "Muito Obrigado!"
4
5  // Frases em Inglês - Coluna 1
6  frases[1,0] = "Welcome!"
7  frases[1,1] = "Thank you very much!"
8
9  // Seleciona o idioma, coluna 1, ou seja Inglês
10 idioma = 1
11
12 // Guarda a mensagem em inglês
13 mensagem = frases[idioma,1]
```

Fonte: Imagem capturada diretamente do programa

3.3.1.3. Escopo das variáveis e vetores

Nem toda variável fica o tempo inteiro disponível na memória depois de definida e também não é acessada de qualquer lugar. Dá-se o nome do alcance de uma variável de escopo. Na Figura 3.38 são observados os três escopos para variáveis no GameMaker: Studio.

Figura 3. 38 - Escopo de variáveis



Fonte: Gráfico criado pelo autor

Uma variável do tipo global pode ser acessada de qualquer parte do jogo, de qualquer instância e de qualquer evento. O prefixo usado para esse tipo de variável é *global.* seguido do nome do identificador. Todos os tipos de variáveis e vetores podem ser globais. São exemplos

de variáveis globais a quantidade de moedas coletadas, a pontuação e a saúde do jogador (Figura 3.39). Há variáveis globais embutidas como *score* e *health*, que não possuem o prefixo *global*.. Para lista todas basta ir até o menu *Scripts* e selecionar a opção *Show Built-In Variables* e procurar as que tenha o termo local indicado no início.

Figura 3. 39 - Variáveis globais

```
1 // Incializa saúde do jogador
2 global.saude = 100
3
4 // Pontos iniciais
5 global.pontos = 0
6
7 // Quantidade inicial de moedas
8 global.moedas = 0
```

Fonte: Imagem capturada diretamente do programa

A variável local de instância pode ser acessada apenas pela própria instância onde foi declarada. Para que outras instâncias acessem seu conteúdo é necessário que a instância em questão ainda exista (Caso não, ocorrerão erros), chamando seu *id* (Ou no caso de existir apenas uma instância, o próprio nome do objeto) seguido de um ponto e o identificador da variável (Figura 3.40). Há várias funções para conseguir o *id* de uma instância especifica (Proximidade, colisão, ordem de criação e etc.). São exemplos a posição da instância, saúde de cada réplica de inimigo e etc.

Existem também variáveis locais embutidas como *x*, *y* e *sprite_index*. Para listar todas basta ir até o menu *Scripts* e selecionar a opção *Show Built-In Variables* e procurar as que tenha o termo *local* indicado no início.

Figura 3. 40 - Variáveis locais de instância

```
1 // Muda posição da instância
2 x = 350
3 y = 200
4
5 // Guarda o valor do 'x' do jogador
6 alvo_x = jogador.x
7
8 // Guarda o 'id' da instância de porta mais próxima
9 seguir = instance_nearest(x, y, porta)
```

Fonte: Imagem capturada diretamente do programa

A variável temporária ou de evento permanece pouquíssimo tempo na memória, o tempo de um quadro de atualização por cada evento. Ao fim do evento a mesma é removida da memória. Ou seja, ela só pode ser criada, acessada ou modificada dentro do próprio evento. Ao declarar esse tipo de variável deve-se usar a estrutura *var* antes do identificador (Figura 3.41). Não é possível usar vetores nessa modalidade.

Um exemplo de uso seria um valor calculado diversas vezes durante o mesmo evento, onde para economizar processamento, faça este cálculo apenas uma vez e o guarde em uma variável temporária.

Figura 3. 41 - Variáveis temporárias

```
1 // calcula a posição do meio da tela
2 var meio = display_get_width() / 2
3
4 // Usa a variável duas vezes sem a necessidade de recalcular
5 // para desenhar os textos
6 draw_text(meio, 20, "Warrior Player")
7 draw_text(meio, 40, "PONTOS: " + string(score))
```

Fonte: Imagem capturada diretamente do programa

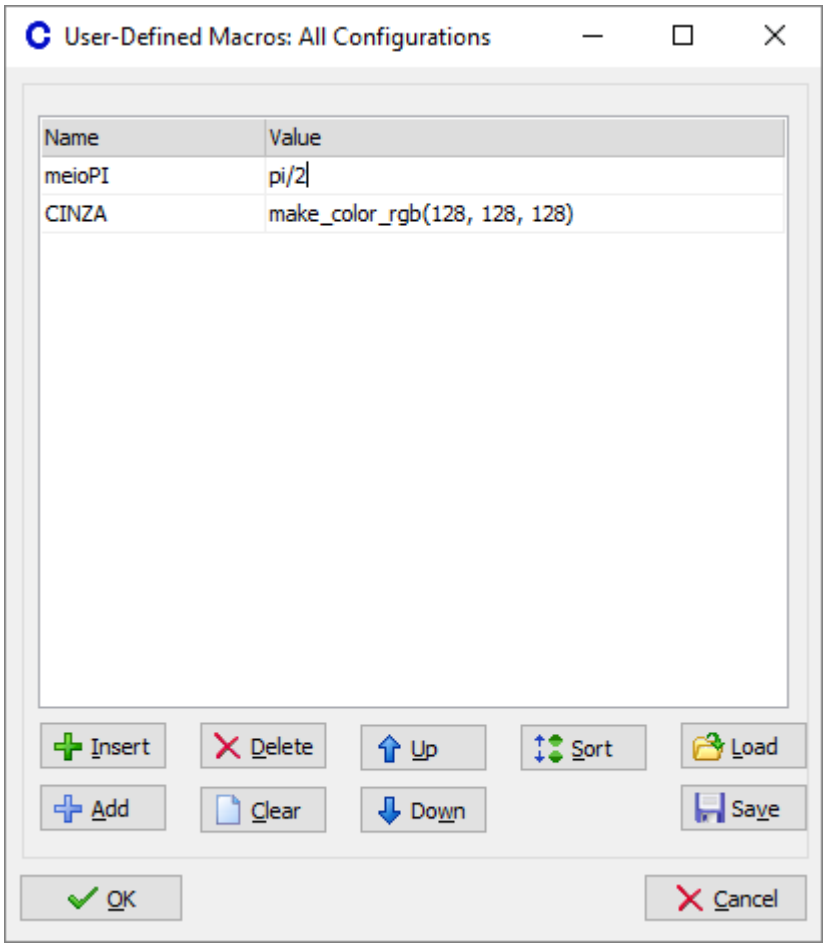
3.3.1.4. Constantes e Macros

Constantes são valores pré-determinados que nunca variam, ou seja, permanece constantes. Há várias constantes embutidas e para listar todas basta ir até o menu *Scripts* e selecionar a opção *Show Constants*. Para ser considerada uma constante não é necessário ter um identificador, pois podem ser valores como o número *15* ou a palavra *carro*, ou seja, o número *15* sempre será *15* e a palavra *carro* sempre será *carro*.

Para definir uma constante com identificador, devemos usar a pasta *Macros* na árvore de recursos. Lá teremos dois arquivos *All Configurations* e *Default*. O primeiro é usado para todas as *configurações*[62]. Ao abrir é apresentada a janela da Figura 3.42.

[62] No *GameMaker: Studio* é possível criar configurações personalizadas excluindo e incluindo recursos.

Figura 3. 42 - Definindo Constantes

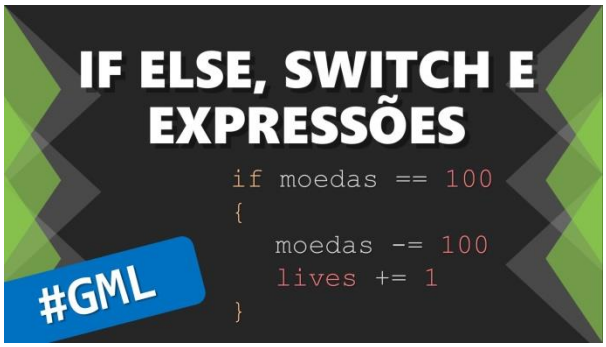


Fonte: Imagem capturada diretamente do programa

É permitido usar funções, cálculos, e outras constantes definidas previamente para atribuir na constante. Basta clicar em *Insert*, nomear um identificador e determinar um valor.

3.3.2. IF, ELSE, SWITCH e Expressões

Figura 3. 43 - Capa do vídeo 'IF ELSE, SWITCH e Expressões'



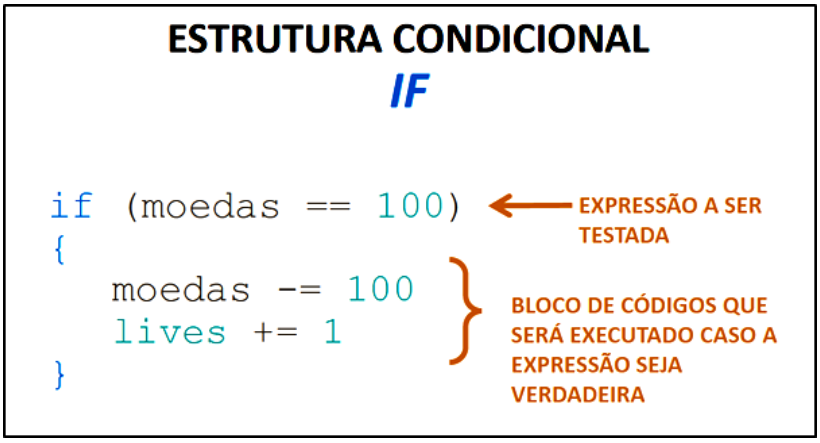
Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=YHqPN6TvMsw)

Apresentando estruturas condicionais no vídeo *IF ELSE, SWITCH e Expressões* (Figura 3.43).

3.3.2.1. Estrutura condicional: IF

É uma estrutura de desvio de execução, onde é testada uma expressão, sendo ela verdadeira um bloco de códigos é executado. Em tradução livre para o português significa *se*. Na Figura 3.44 testa-se a variável *moedas* tem o valor exatamente igual a 100, caso isso seja verdadeiro o bloco de códigos entre chaves é executado.

Figura 3. 44 - Estrutura condicional IF



Fonte: Imagem criada pelo autor

3.3.2.2. Expressões boolenas

Uma expressão pode ser uma ou mais operações que retornam um valor booleano (Verdadeiro ou falso). No Quadro 3.6 observa-se várias expressões e seus valores. E ainda podemos usar os operadores relacionais de igualdade (==), maior que (>), maior ou igual (>=), menor que, menor ou igual (<=) e diferente de (!=).

Quadro 3. 6 - Expressões e operadores relacionais

Expressão	**Valor booleano**
`(5 > 3)`	**`true`** (5 é maior que 3)
`(5 + 3 == 8)`	**`true`** (8 é exatamente igual a 8)
`15`	**`true`** (Qualquer valor acima de 0 é verdadeiro)
`"Olá mundo"`	**`true`** (Qualquer *string* é verdadeira, mesmo vazia)
`("oi" == "olá")`	**`false`** (As *strings* são diferentes)
`(15 <= 20)`	**`false`** (15 não é menor e nem igual a 20)
`(irandom(20) >= 5)`	**`true`** (Se a função retornar um valor maior ou igual a 5, caso não **`false`**)
`(10 != 10)`	**`false`** (10 é igual a 10, (!=) é um operador de diferença)
`!(15 > 10)`	**`false`** (A expressão tem valor invertido com o prefixo (!))

Fonte: Referência do programa (https://docs.yoyogames.com/source/dadiospice/002_reference/001_gml%20language%20overview/401_04_expressions.html)

É importante diferenciar o operador de igualdade (==) do operador de atribuição (=), a primeira é usada em comparações e a segunda para aplicar valores.

Um valor verdadeiro é igual à constante 1, que é também representada pela constante *true*, e o valor falso representado pela constante false que equivale a 0.

Também podemos utilizar operadores lógicos para agrupar várias expressões em uma única expressão (Quadro 3.7). Temos dois operadores lógicos básicos, o *and* (e) e o *or* (ou).

Quadro 3. 7 - Operadores lógicos AND e OR

Expressão	**Valor booleano**
`(moedas == 100 and magia == 2)`	Verdadeiro se as duas expressões forem verdadeiras, mas se apenas uma for falsa, toda expressão se torna falsa.
`(vidas > 3 or arma != 4)`	Verdadeiro se ao menos uma das expressões for verdadeira.

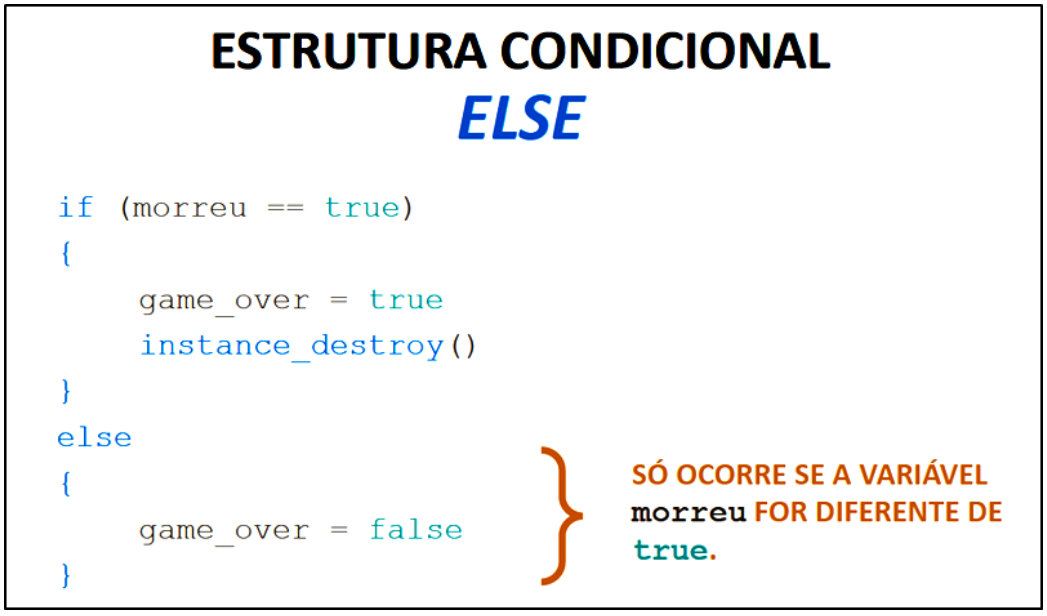
`((balas > 0 and arma == 3) or infinito == true)`	Resolve primeiramente a expressão dos parênteses mais internos (and), depois com o resultado em combinação com o operador *or* obtém o valor da expressão.

Fonte: Referência do programa (https://docs.yoyogames.com/source/dadiospice/002_reference/001_gml%20language%20overview/401_04_expressions.html)

3.3.2.3. Estrutura condicional: ELSE

Significa *senão*, e é um comando utilizado em conjunto com o *if* e desvia o código para outro bloco de código (Figura 3.45), caso a expressão anteriormente testada pelo *if* não seja verdadeira.

Figura 3. 45 - Estrutura condicional ELSE

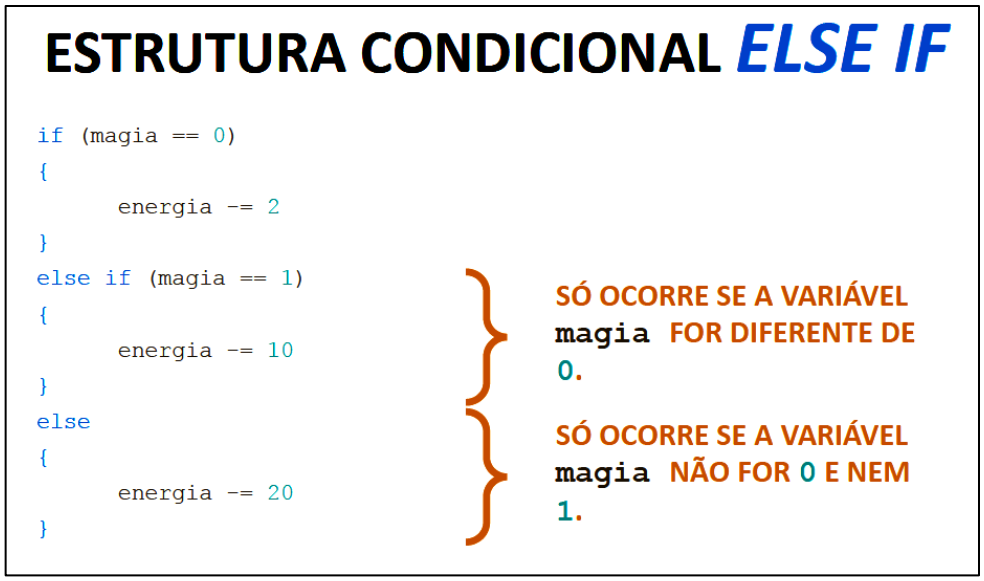


Fonte: Imagem criada pelo autor

3.3.2.4. Estrutura condicional: ELSE IF

Agrupa as funcionalidades das duas estruturas anteriores. Neste caso podemos testar uma nova expressão além da inicial (Figura 3.46). Esse tipo de estrutura evita que o programa teste expressões de forma desnecessária, dado que uma expressão fica dependente de outra.

Figura 3. 46 - Estrutura condicional ELSE IF



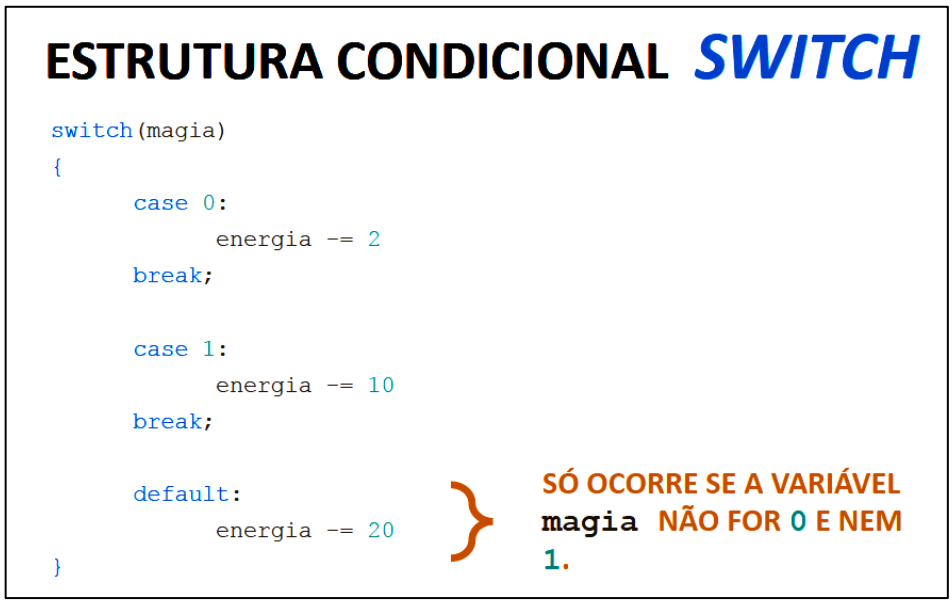
Fonte: Imagem criada pelo autor

3.3.2.5. Estrutura condicional: SWITCH

Tem a mesma funcionalidade da estrutura *else if*, porém sua dinâmica é diferente. O comando *switch* é usado seguido de uma expressão, geralmente essa expressão deve retornar mais de dois valores diferentes (Caso não, compensa utilizar *if else*). Em seguida especifica-se as possibilidades de valores com *case*, onde em seguida colocamos o código a ser executado adicionando *break* ao final do bloco.

Ainda há o comando *default* que nos permite colocar um caso padrão, se os valores especificados antes forem retornados (Equivale ao *else* no final de uma sequência de verificações). Na Figura 3.47 nota-se o mesmo código do exemplo da Figura 3.46, porém usando *switch*.

Figura 3. 47 - Estrutura condicional SWITCH



Fonte: Imagem criada pelo autor

3.3.3. Estruturas de repetição: WHILE e DO UNTIL

Figura 3. 48 - Capa do vídeo WHILE e DO UNTIL

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=Mmex38uMPkA)

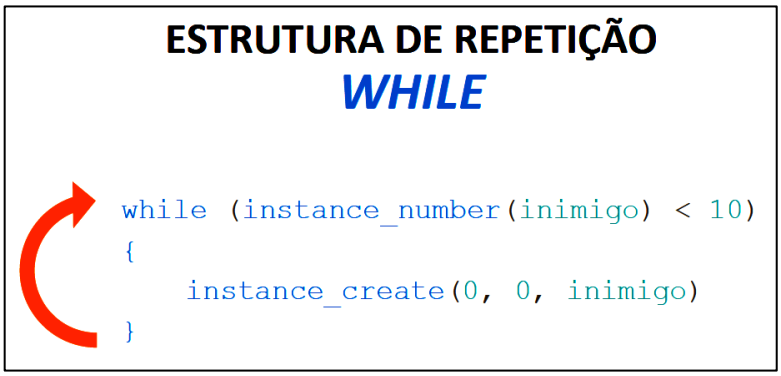
Vídeo (Figura 3.48) dedicado aos laços de repetição *while* e do *until*.

3.3.3.1. Estrutura de repetição: WHILE

Traduzindo para o português significa *enquanto*. Tem a sintaxe semelhante ao *if*, apesar da execução ser diferente. No *if* o desvio sempre ocorre para frente do código, em

contrapartida o *while* além de voltar no código, o executa novamente, por isso o nome laço de repetição.

Figura 3. 49 - Estrutura de repetição WHILE



Fonte: Imagem criada pelo autor

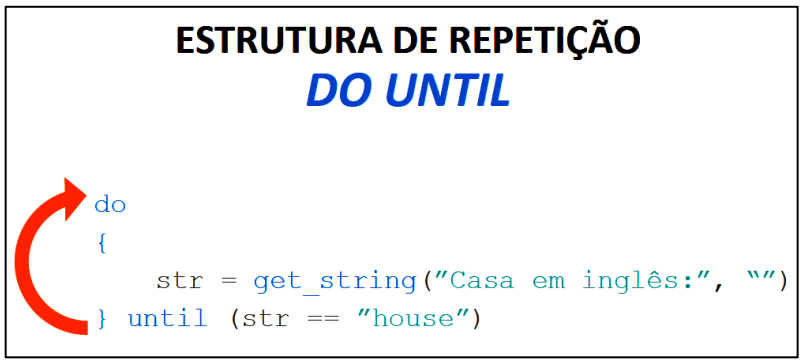
Na Figura 3.49 ocorre a seguinte situação: Enquanto o número de instâncias do objeto *inimigo* for menor que 10, será criada uma nova instância de *inimigo*. A estrutura executa os seguintes passos:

- **Passo 1**: Verifica se a expressão é verdadeira (caso sim, vai para o passo 2 e caso não vai para o passo 3);
- **Passo 2**: Executa o bloco de códigos e volta para o passo 1;
- **Passo 3**: Sai do laço e continua a execução do código.

3.3.3.2. Estrutura de repetição: DO UNTIL

Muito parecido com *while*, só que este executa primeiramente o bloco de códigos e depois verifica a expressão. O processo se repete até que a expressão seja verdadeira. Resumindo: Execute o código até que a expressão seja verdadeira.

Figura 3. 50 - Estrutura de repetição DO UNTIL



Fonte: Imagem criada pelo autor

No código da Figura 3.50 o programa pede uma entrada de texto, guarda esse texto em uma variável e verifica se ela é a palavra *house*. No caso de coincidir, sai do ciclo de repetição e continua a execução do código. Já se a resposta não for adequada, o processo é executado novamente até que o usuário entre com a resposta correta.

3.3.4. Estruturas de repetição: FOR e REPEAT

Figura 3. 51 - Capa do vídeo 'FOR e REPEAT'

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v= ueFb0kHbXC4)

São abordadas no vídeo (Figura 3.51) as estruturas de repetição *for* e *repeat*.

3.3.4.1. Estrutura de repetição: FOR

Das estruturas de repetição, a mais complexa, por envolver mais etapas. Usada geralmente para percorrer índices de estruturas de dados como vetores e listas.

Essa estrutura segue os seguintes passos de execução:

•Passo 1: Executa a inicialização uma vez e vai para o passo 2;

•Passo 2: Verifica se a condição é verdadeira (caso sim, vai para o passo 3, caso não, vai para o passo 5);

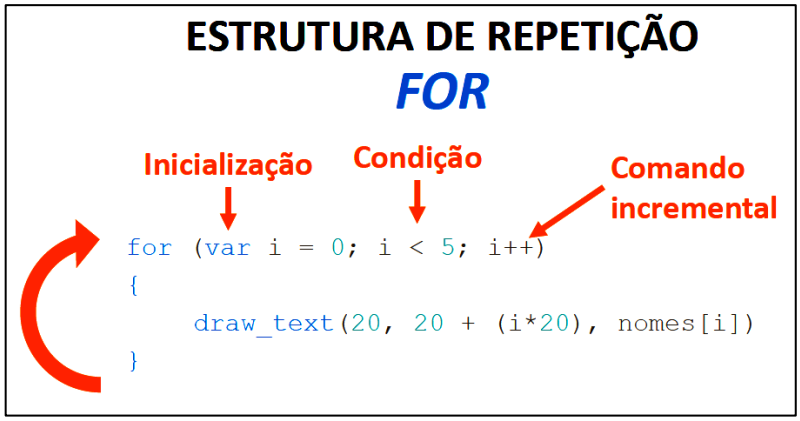
•Passo 3: Executa bloco de código e vai para o passo 4.

•Passo 4: Executa comando incremental e volta para o passo 2.

•Passo 5: Sai do laço e continua a execução do código.

Na Figura 3.52 são desenhados os 5 primeiros valores dentro do vetor *nomes*. Tanto a inicialização quanto o comando incremental são livres para execução de mais comandos. Por convenção é utilizado esse modelo com declaração e incremento de variável.

Figura 3. 52 - Estrutura de repetição FOR

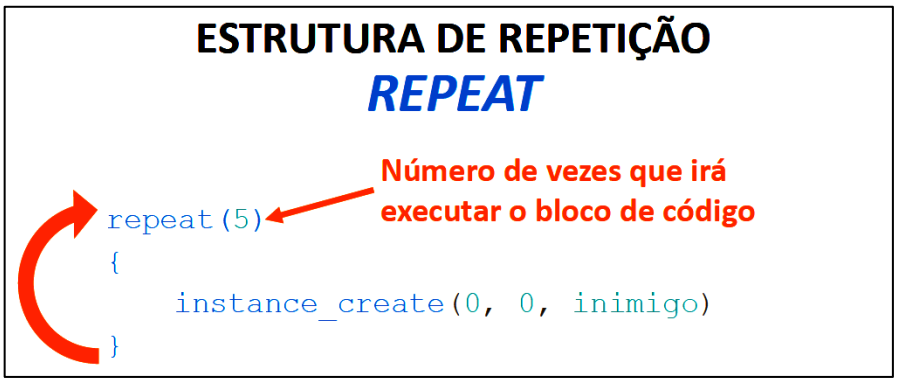


Fonte: Imagem criada pelo autor

3.3.4.2. Estrutura de repetição: REPEAT

Ao contrário do anterior, este é o laço de repetição mais simples. Basta apenas atribuir o número de vezes que um bloco de código será repetido. Na Figura 3.53 o código é repetido 5 vezes.

Figura 3. 53 - Estrutura de repetição REPEAT



Fonte: Imagem criada pelo autor

3.3.5. Funções e Scripts

Figura 3. 54 - Capa do vídeo 'Funções e Scripts'

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=2S5W8Rz4LAE)

No vídeo *Funções e Scripts* destaca-se o modo como o *GameMaker: Studio* trata as funções (Figura 3.54).

3.3.5.1. Funções

Uma função é uma estrutura que permite chamar um bloco de códigos através de um identificador (com ou sem argumentos) podendo ou não retornar um valor. No *GameMaker: Studio*, especialmente, as funções já estão embutidas no programa. Para ver todas as funções que estão embutidas basta acessar o menu *Scripts* e selecionar a opção *Show Built-In Functions*. No Quadro 3.8 estão destacadas algumas das funções mais utilizadas.

Quadro 3. 8 - Funções mais utilizadas

Funções	**Ação realizada**	**Valor de retorno**
`instance_create(x, y, obj)`	Cria uma instância de objeto	*id* da instância criada
`instance_destroy()`	Destrói a instância atual	Nada
`game_end()`	Finaliza o jogo	Nada
`game_restart()`	Reinicia o jogo	Nada
`irandom(val)`	Gera um número aleatório	Valor inteiro
`place_free(x, y)`	Verifica colisão com sólidos	Valor booleano
`draw_text(x, y, str)`	Desenha texto na tela	Nada

Fonte: Quadro criado pelo autor

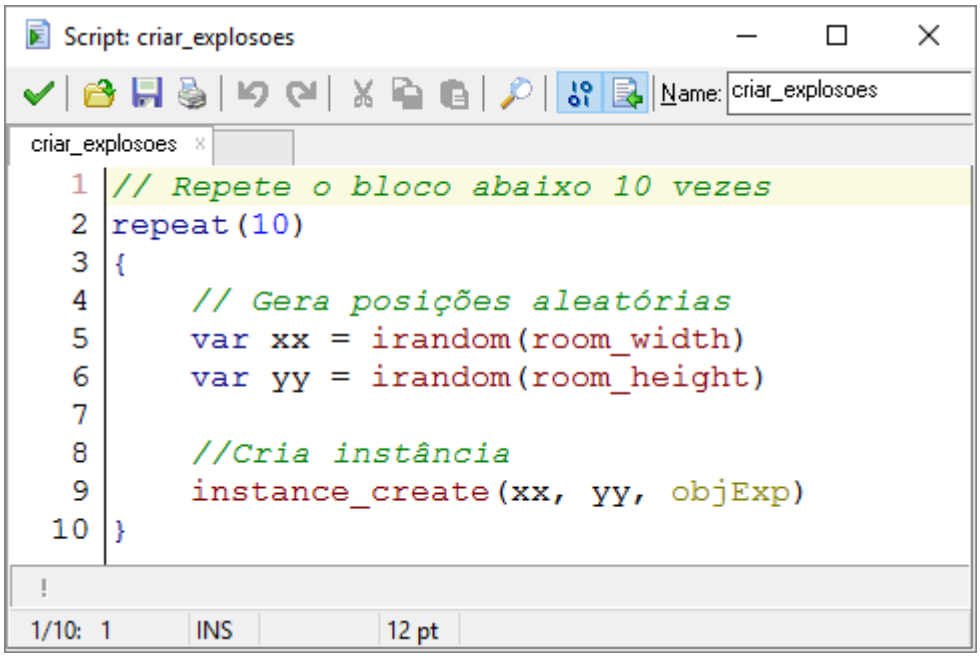
3.3.5.2. Scripts

Funções definidas pelo programador são chamadas de *Scripts* e ficam na árvore de recursos do projeto. Os termos *método* e *procedimento* não são usados para se referir a essas estruturas no *GameMaker: Studio*, apesar de ser possível encaixar a suas definições a estas.

Lembrando que o objetivo de utilizar tais estruturas é evitar o retrabalho e dinamizar o desenvolvimento do código, devemos criar scripts para simplificar algoritmos.

O *script* da Figura 3.55 gera 10 instâncias de um objeto chamado *objExp* em posições aleatórias na *room*.

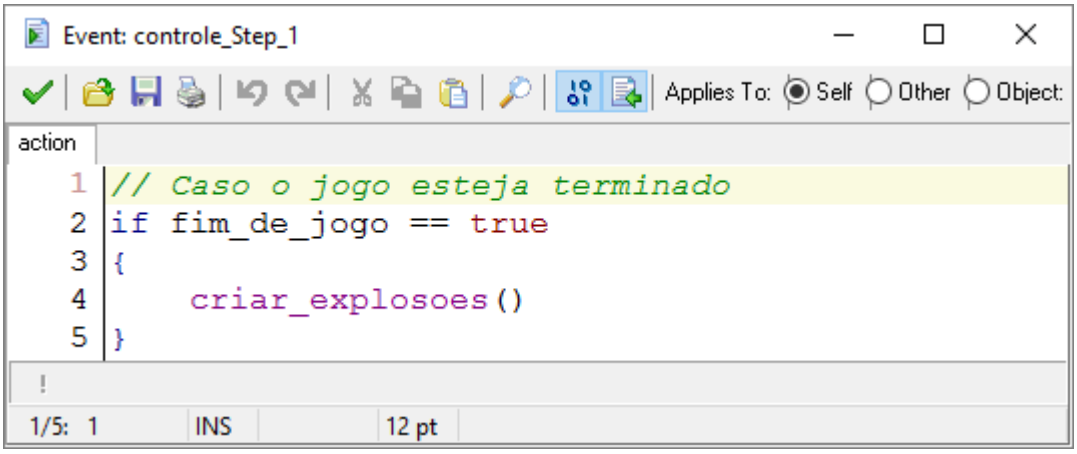
Figura 3. 55 - Script 'criar_explosoes'



Fonte: Imagem capturada diretamente do programa

Depois basta chamar o identificador do *script* quando precisar das explosões (Figura 3.56).

Figura 3. 56 - Criando explosões



Fonte: Imagem capturada diretamente do programa

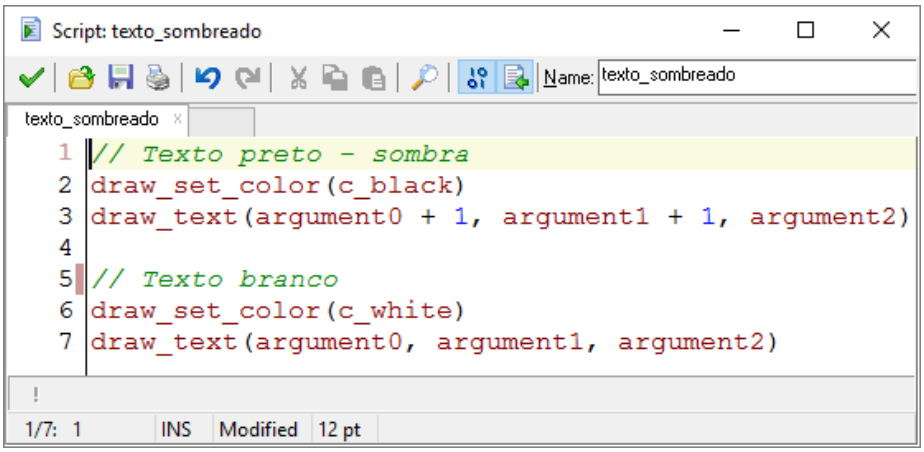
3.3.5.3. Argumentos ou Parâmetros

Para personalizar ainda mais o *script*, há a possibilidade de usar parâmetros, que no *GameMaker: Studio* são comumente chamados de argumentos.

São suportados até 16 argumentos em um mesmo script, e há dois modos de usa-los, um utilizando variáveis com prefixo *argument* seguidos de um número de 0 a 15, e o outro que usa o vetor *argument* com índice de 0 a 15, regido pela variável *argument_count*. Será explicado apenas o primeiro, posto que é o método mais tradicional.

Na Figura 3.57 é criado um *script* para desenhar um texto com uma sombra simples.

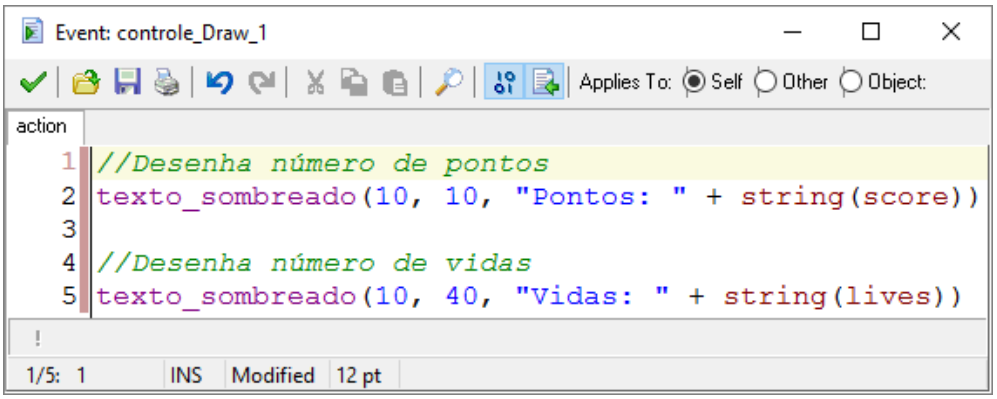
Figura 3. 57 - Script 'Texto com sombra'



Fonte: Imagem capturada diretamente do programa

O primeiro argumento (*argument0*) é a posição horizontal do texto, o segundo (*argument1*) é a posição vertical e o terceiro (*argument2*), uma *string*. Nesse script o texto em preto, que é desenhado primeiro, é deslocado 1 pixel para direita e um pixel para baixo. A Figura 3.58 demonstra o uso desse script.

Figura 3. 58 - Desenhando textos com sombra



Fonte: Imagem capturada diretamente do programa

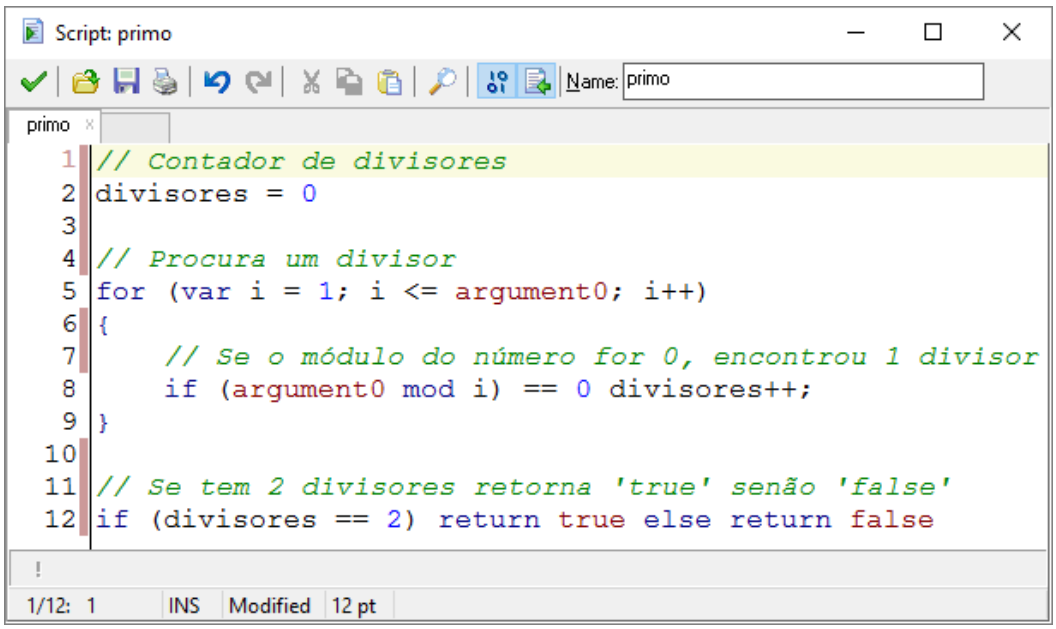
3.3.5.4. Retornando valores

Já foi apresentado algumas funções que retornam um valor, como no caso da *irandom*, que devolve um valor inteiro entre *0* e o parâmetro passado. Para fazer isso com *scripts* usamos o comando *return*.

Uma função ou *script* são desvios no código, pois, são chamados em dado momento desviando a execução para seu escopo, onde executam suas ações e depois retornam para onde foram chamados. Esse retorno pode ser adiantado usando *return*.

Mas essa não é a funcionalidade principal desse comando, pois o *script* pode trazer consigo um valor. Na Figura 3.59 vê-se a checagem de dado número para verificar se ele é um *número primo*[63].

Figura 3. 59 - Script 'primo'



Fonte: Imagem capturada diretamente do programa

Lembrando que o comando *return* encerra a execução do *script*, deve haver cautela ao usá-lo. Se por exemplo colocar na primeira linha e ainda houver código a ser executado, esse código nunca será executado.

3.4. Estruturas de Dados (Data Structures)

[63] Número que é divisível por 1 e por ele mesmo apenas.

Neste subcapitulo iremos tratar das principais estruturas de dados disponíveis no *GameMaker: Studio*, que são: Listas, filas, pilhas e grelhas.

3.4.1. Listas

Figura 3. 60 - Capa do vídeo 'Estruturas de dados: Listas'

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=wa-tnaUt1VU)

No vídeo mencionado acima (Figura 3.60), as listas são explicadas com detalhes, mostrando as funções em um exemplo prático.

As listas são parecidas com vetores/arrays de uma dimensão, porém mais simples de manusear. Elas possuem índices com seus valores correspondentes, onde usamos funções pré-definidas para configurá-las. Um exemplo de uso seria para uma lista de pontuações mais altas e também para ordenar uma lista de nomes alfabeticamente.

3.4.1.1. Função: ds_list_create

Figura 3. 61 - Função para criação de listas

```
1 lista = ds_list_create()
```

Fonte: Imagem capturada diretamente do programa

O código acima cria uma lista e guarda na variável *lista* (Figura 3.61).

3.4.1.2. Função: ds_list_add

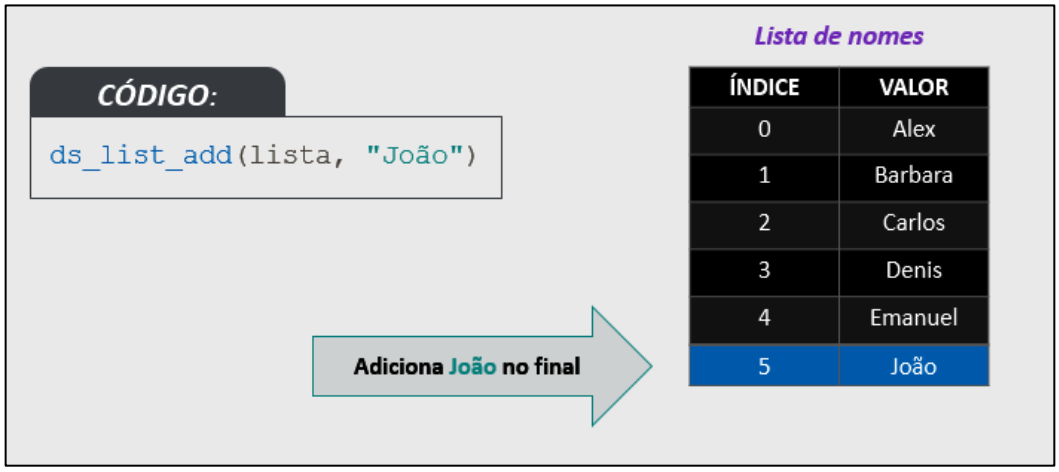
Figura 3. 62 - Função para inserir um elemento no final da lista

```
1 ds_list_add(lista, "João")
```

Fonte: Imagem capturada diretamente do programa

Acima adicionamos o elemento *João* na última posição de *lista* (Figura 3.62). Caso não haja itens, coloca na primeira posição. Listas aceitam valores repetidos, logo podemos ter mais de um valor *João*, por exemplo. O esquema abaixo (Figura 3.63) demonstra o funcionamento dessa função.

Figura 3. 63 – Esquema de inserção no fim da lista



Fonte: Imagem criada pelo autor

3.4.1.3. Função: ds_list_insert

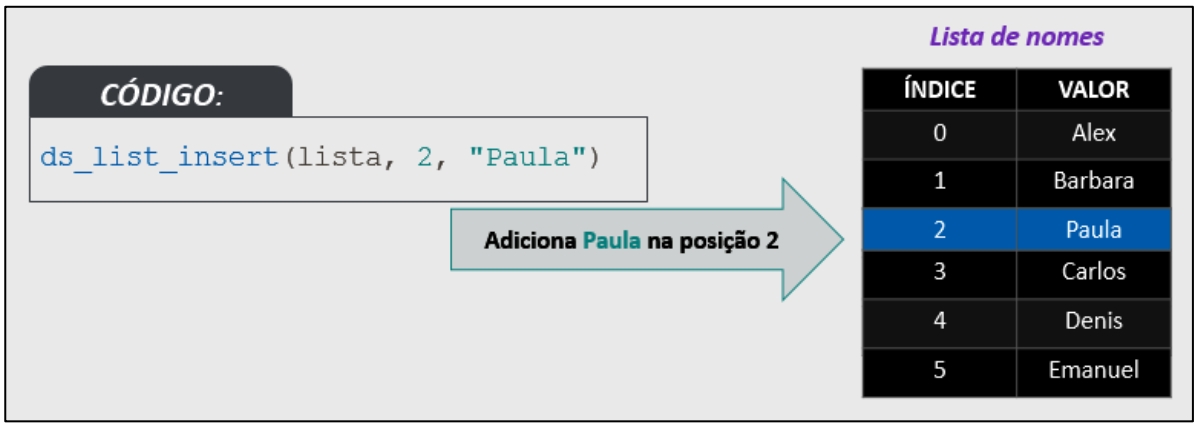
Figura 3. 64 - Função para inserir um elemento em uma posição especifica da lista

```
1 ds_list_insert(lista, 2, "Paula")
```

Fonte: Imagem capturada diretamente do programa

No código acima (Figura 3.64) inserimos o elemento Paula na 3ª posição de lista. O índice inicial é 0, a primeira posição. Os elementos posteriores à posição inserida têm os índices acrescidos em uma posição na lista (Figura 3.65).

Figura 3. 65 – Esquema de inserção na lista



Fonte: Imagem criada pelo autor

3.4.1.4. Função: ds_list_delete

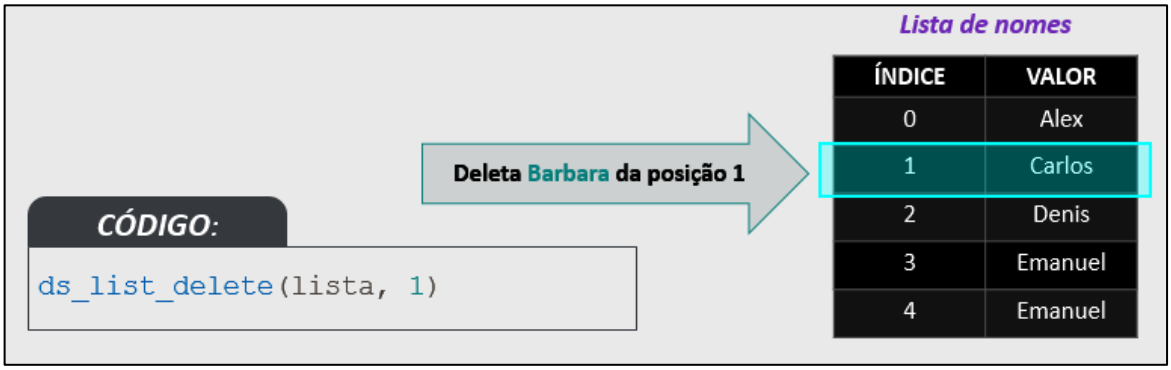
Figura 3. 66 - Função para deletar um elemento em uma posição especifica da lista

```
1 ds_list_delete(lista, 1)
```

Fonte: Imagem capturada diretamente do programa

No código destacado (Figura 3.66) deletamos o elemento da 2ª posição de lista. Os elementos posteriores à posição inserida têm os índices decrescidos em uma posição na lista (Figura 3.67). Caso seja indicada uma posição que não possui valor, nada acontece.

Figura 3. 67 – Esquema de deleção na lista



Fonte: Imagem criada pelo autor

3.4.1.5. Função: ds_list_replace

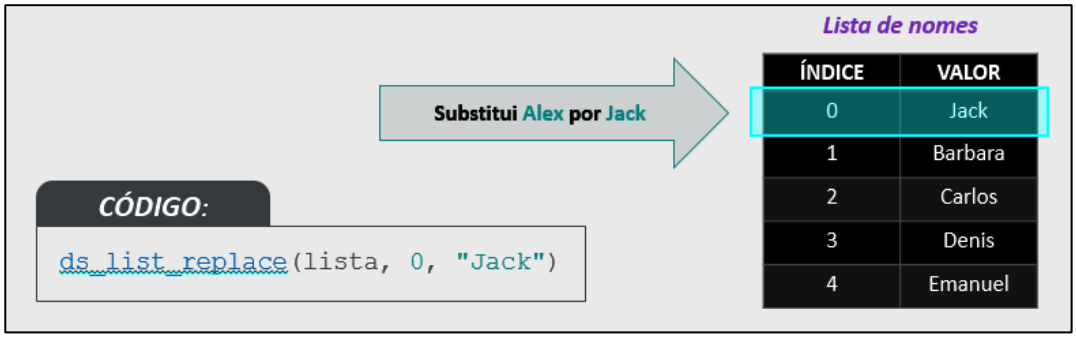
Figura 3. 68 - Função para substituir um elemento em uma determinada posição da lista

```
1 ds_list_replace(lista, 0, "Jack")
```

Fonte: Imagem capturada diretamente do programa

Acima (Figura 3.68) substituímos o elemento da 1ª posição de lista por *Jack*. Conforme podemos ver no esquema abaixo (Figura 3.69). Caso seja indicada uma posição que não possui valor, nada acontece.

Figura 3. 69 – Esquema de substituição na lista



Fonte: Imagem criada pelo autor

3.4.1.6. Função: ds_list_find_value

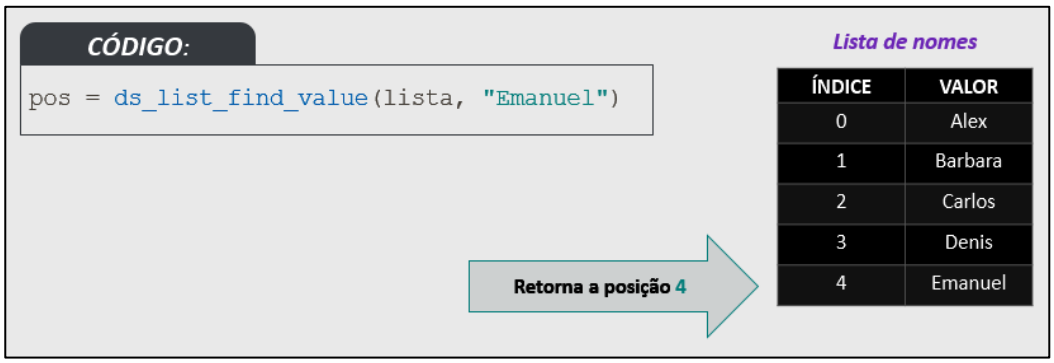
Figura 3. 70 - Função para encontrar a posição de um valor da lista

```
1 pos = ds_list_find_value(lista, "Emanuel")
```

Fonte: Imagem capturada diretamente do programa

Acima (Figura 3.70) procuramos o valor *Emanuel* na lista e salvamos seu índice na variável *pos*. O esquema abaixo (Figura 3.71) mostra o processo. Se o valor não existe na lista um valor indefinido é retornado.

Figura 3. 71 – Esquema de busca de valores na lista



Fonte: Imagem criada pelo autor

3.4.1.7. Função: ds_list_find_index

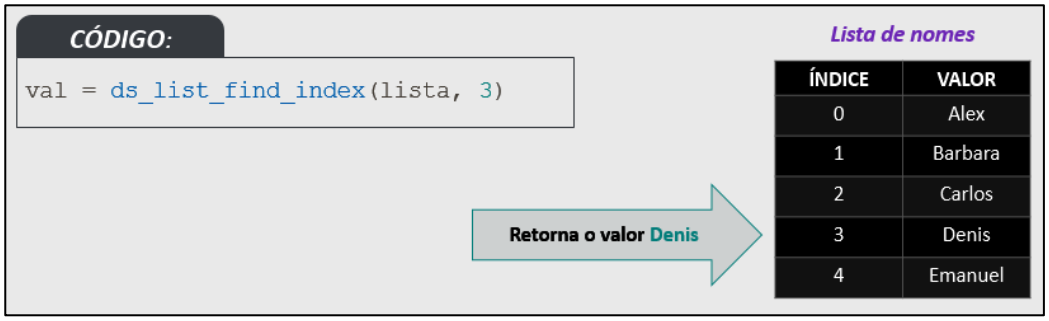
Figura 3. 72 - Função para encontrar a valor de uma dada posição na lista

```
1 val = ds_list_find_index(lista, 3)
```

Fonte: Imagem capturada diretamente do programa

No script acima (Figura 3.72) procuramos a posição *3 (Quarta posição)* da lista e guardamos seu valor na variável *val*. Se o valor não é encontrado na lista o valor -1 é retornado. O esquema abaixo (Figura 3.73) mostra como isso ocorre.

Figura 3. 73 – Esquema de busca de posições na lista



Fonte: Imagem criada pelo autor

3.4.1.8. Função: ds_list_clear

Figura 3. 74 - Função que remove todos os elementos da lista

```
1 ds_list_clear(lista)
```

Fonte: Imagem capturada diretamente do programa

Acima (Figura 3.74) todos os elementos de *lista* são deletados, assim a lista fica vazia, mas ainda existe na memória.

3.4.1.9. Função: ds_list_destroy

Figura 3. 75 - Função que remove a lista da memória

```
1 ds_list_destroy(lista)
```

Fonte: Imagem capturada diretamente do programa

No código mostrado acima (Figura 3.75) a lista é destruída, ou seja, o identificador *lista* não guarda mais nada (Valor Indefinido ou nulo).

3.4.1.10. Função: ds_list_sort

Figura 3. 76 - Função que ordena a lista alfanumericamente

```
1 ds_list_sort(lista, true)
```

Fonte: Imagem capturada diretamente do programa

Acima (Figura 3.76) a lista é ordenada de forma alfanumérica, sendo que o segundo parâmetro determina se é uma ordenação ascendente (Valor *true*) ou descendente (Valor *false*).

3.4.1.11. Função: ds_list_shuffle

Figura 3. 77 - Função que embaralha os elementos da lista

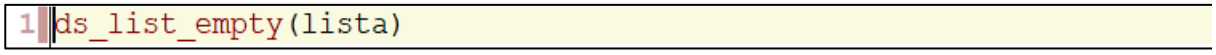
```
1 ds_list_shuffle(lista)
```

Fonte: Imagem capturada diretamente do programa

O código acima (Figura 3.77) faz com que os elementos da lista sejam embaralhados, ou seja, seus índices são trocados de forma aleatória.

3.4.1.12. Função: ds_list_empty

Figura 3. 78 - Função que verifica se a lista está vazia

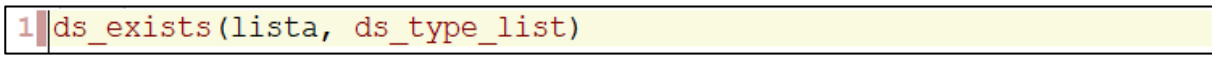


Fonte: Imagem capturada diretamente do programa

No código acima (Figura 3.78) fazemos a verificação com *lista*. A função irá retornar *true* caso a lista esteja vazia, e *false* caso não esteja.

3.4.1.13. Função: ds_exists com parâmetro 'ds_type_list'

Figura 3. 79 - Função que verifica se a lista existe

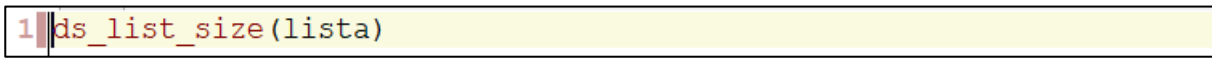


Fonte: Imagem capturada diretamente do programa

Na instrução acima (Figura 3.79) verificamos se *lista* existe. No segundo parâmetro indicamos que a estrutura de dados a ser verificada é uma lista. A função irá retornar *true* caso a lista exista, e *false* caso não esteja. É recomendável fazer essa verificação antes de qualquer operação com listas, evitando erros inesperados.

3.4.1.14. Função: ds_list_size

Figura 3. 80 - Função que retorna o tamanho da lista



Fonte: Imagem capturada diretamente do programa

Na instrução acima (Figura 3.80) é retornada a quantidade de elementos que *lista* possui.

3.4.1.15. Função: ds_list_write

Figura 3. 81 - Função que gera uma *string* codificada da lista

```
1 dados = ds_list_write(lista)
```

Fonte: Imagem capturada diretamente do programa

No trecho acima destacado é gerada uma *string* codificada que é guardada na variável *dados* (Figura 3.81). Essa *string* pode ser carregada novamente em uma lista usando a função *ds_list_read*.

3.4.1.16. Função: ds_list_read

Figura 3. 82 - Função que carrega uma *string* codificada para criar uma lista

```
1 ds_list_read(lista, dados)
```

Fonte: Imagem capturada diretamente do programa

Essa função carrega uma lista a partir da *string* previamente gerada com a função ds_*list_write* e guardada na variável *dados* (Figura 3.82). A lista também deve ter sido criada anteriormente usando *ds_list_create* e assim usada no primeiro parâmetro.

3.4.2. Grelhas

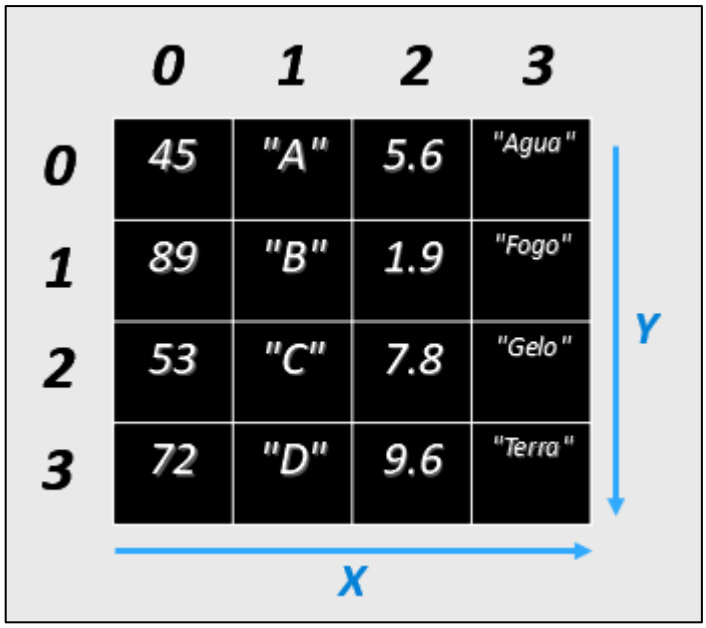
Figura 3. 83 - Capa do vídeo 'Estruturas de dados: Grelhas'

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=ipTM0Vn5u2c)

No vídeo são apresentadas as grelhas ou grades, bem como as funções necessárias para manipulá-las (Figura 3.83).

As grelhas são parecidas com vetores/arrays de duas dimensões, com a vantagem de armazenar qualquer tipo de dados mesclados. Ou seja, você pode ter uma *string* ao lado de um valor número real. Essas estruturas são parecidas com as tabelas do *Microsoft Excel*. Um exemplo de uso seria para criar um inventário de itens para o personagem.

Figura 3. 84 – Exemplo de uma grelha 4x4 com elementos de tipos diferentes



Fonte: Imagem criada pelo autor

3.4.2.1. Função: ds_grid_create

Figura 3. 85 - Função para criação de grelhas

```
1 grelha = ds_grid_create(4, 4)
```

Fonte: Imagem capturada diretamente do programa

O código acima cria uma grelha com 4 células de largura (Primeiro parâmetro) e 4 células de altura (Segundo parâmetro), totalizando 16 células de dados (Figura 3.84). Assim guardamos a grelha na variável *grelha* para usar futuramente. Inicialmente todas as células tem valor 0 (zero).

3.4.2.2. Função: ds_grid_set

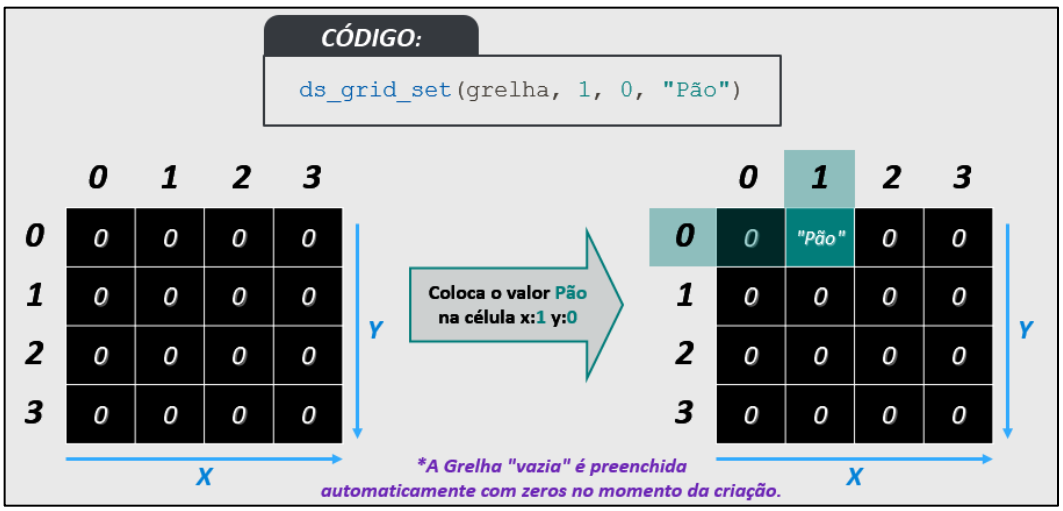
Figura 3. 86 - Função para modificar uma célula da grelha

```
1 ds_grid_set(grelha, 1, 0, "Pão")
```

Fonte: Imagem capturada diretamente do programa

Acima modificamos a célula que se encontra na posição X:1 (Segundo parâmetro) e Y:0 (Terceiro parâmetro) de *grelha*, colocando o valor *Pão* (Figura 3.86). Caso a posição não exista, nada acontece na execução do jogo (É exibido um alerta no console de desenvolvedor). Vejamos com mais detalhes no esquema abaixo (Figura 3.87).

Figura 3. 87 – Esquema de mudança de dados de uma célula



Fonte: Imagem criada pelo autor

3.4.2.3. Função: ds_grid_get

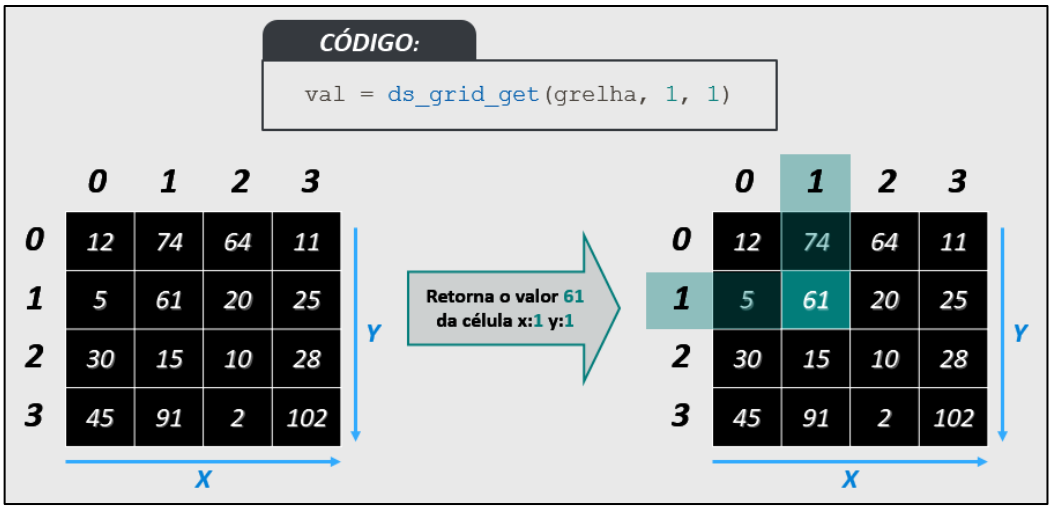
Figura 3. 88 - Função para retornar valor de uma célula da grelha

```
1 val = ds_grid_get(grelha, 1, 1)
```

Fonte: Imagem capturada diretamente do programa

Com esse código resgatamos o valor que se encontra na posição X:1 e Y:1 de *grelha* e guardamos na variável *val* (Figura 3.88). Caso a posição não exista, nada acontece na execução do jogo (É exibido um alerta no console de desenvolvedor). Esse retorno funciona como descrito abaixo (Figura 3.87).

Figura 3. 89 – Esquema de retorno de dados de uma célula



Fonte: Imagem criada pelo autor

3.4.2.4. Função: ds_grid_add

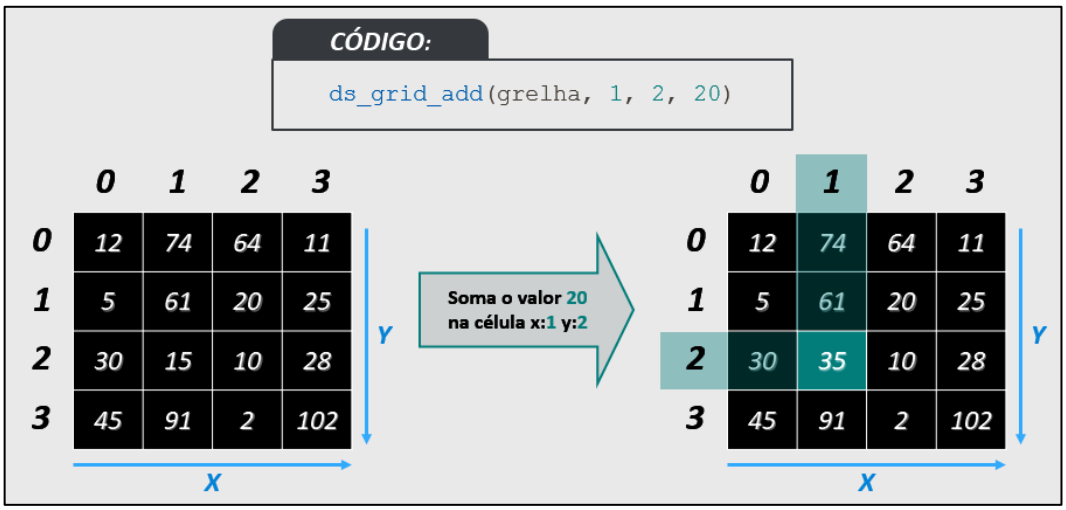
Figura 3. 90 - Função que soma um dado valor à um pré-existente

```
1 ds_grid_add(grelha, 1, 2, 20)
```

Fonte: Imagem capturada diretamente do programa

O código acima acrescenta o valor *20* (Quarto parâmetro) ao valor da posição X:1 e Y:2 de *grelha* (Figura 3.90) como mostra o esquema abaixo (Figura 3.91). Caso a posição não exista, nada acontece na execução do jogo (É exibido um alerta no console de desenvolvedor). Se o valor adicionado for de um tipo diferente, o valor é simplesmente substituído.

Figura 3. 91 – Esquema soma de valores em grelhas



Fonte: Imagem criada pelo autor

3.4.2.5. Função: ds_grid_clear

Figura 3. 92 - Função que remove todos os elementos da grelha

```
1 ds_grid_clear(grelha, 0)
```

Fonte: Imagem capturada diretamente do programa

No código acima (Figura 3.92) todas as células de *grelha* são substituídas pelo valor destacado no segundo parâmetro, ou seja, zero.

3.4.2.6. Função: ds_grid_destroy

Figura 3. 93 - Função que remove a grelha da memória

```
1 ds_grid_destroy(grelha)
```

Fonte: Imagem capturada diretamente do programa

Código que deleta *grelha* da memória deixando seu identificador com valor indefinido (Figura 3.93).

3.4.2.7. Funções: ds_grid_value_exists, ds_grid_value_x e ds_grid_value_yy

Figura 3. 94 – Funções de uso conjunto para buscar valores na grelha

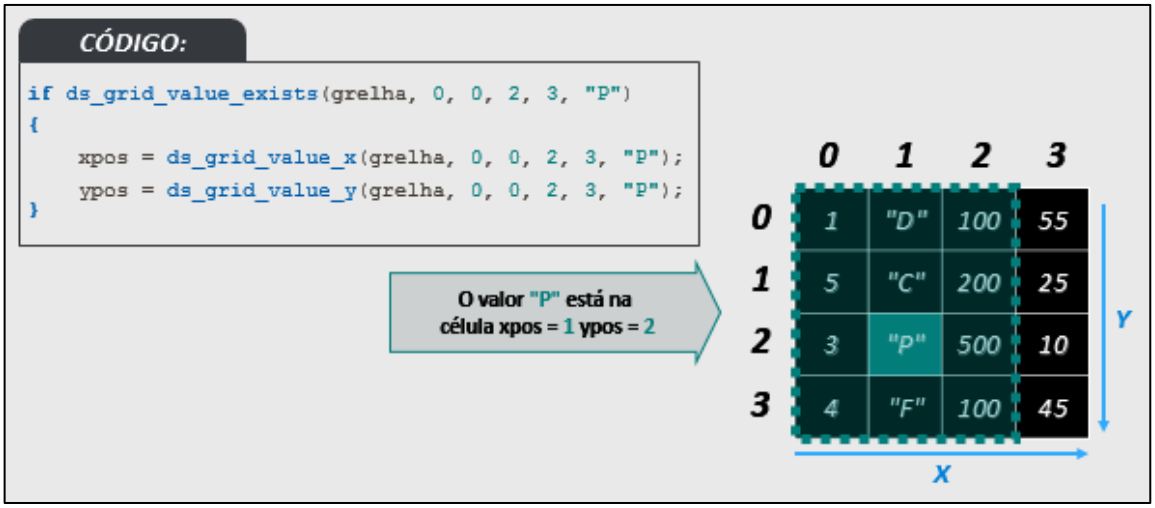
```
if ds_grid_value_exists(grelha, 0, 0, 2, 3, "P")
{
    xpos = ds_grid_value_x(grelha, 0, 0, 2, 3, "P");
    ypos = ds_grid_value_y(grelha, 0, 0, 2, 3, "P");
}
```

Fonte: Imagem capturada diretamente do programa

A função *ds_grid_value_exists* verifica uma área retangular da grelha (Do segundo ao quinto parâmetro) a procura do valor destacado no sexto parâmetro (Figura 3.94). Caso encontre o valor retorna *true*, senão *false*.

Só fazendo essa verificação antes podemos usar as funções *ds_grid_value_x* e *ds_grid_value_y* que retornam a posição da célula que tem o valor procurado. Para ficar mais claro basta observar o esquema da Figura 3.95.

Figura 3. 95 – Esquema de busca de valores na grelha



Fonte: Imagem criada pelo autor

3.4.2.8. Função: ds_grid_resize

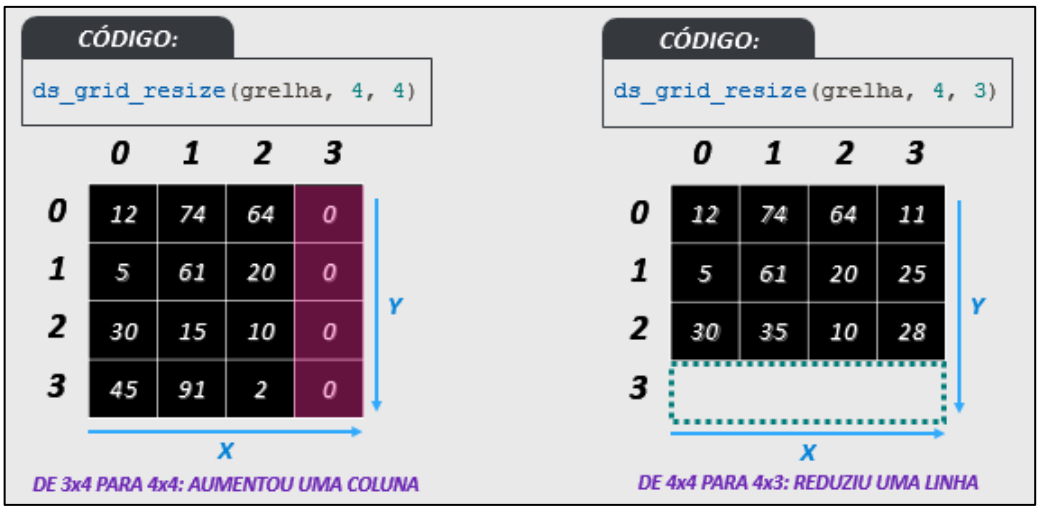
Figura 3. 96 – Função de redimensionamento de grelhas

```
ds_grid_resize(grelha, 4, 3)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_grid_value_resize* define a largura e a altura de uma grelha (Segundo e terceiro parâmetros respectivamente). O código acima redimensiona a grelha para 4x3. Caso reduza o nº de colunas ou linhas, os dados serão perdidos. Caso aumente a nova linha ou coluna será preenchida com zeros. Na Figura 3.97 isso é explicado com mais detalhes.

Figura 3. 97 – Esquema de redimensionamento de grelhas



Fonte: Imagem criada pelo autor

3.4.2.9. Função: ds_grid_sort

Figura 3. 98 – Função para ordenar colunas de grelhas

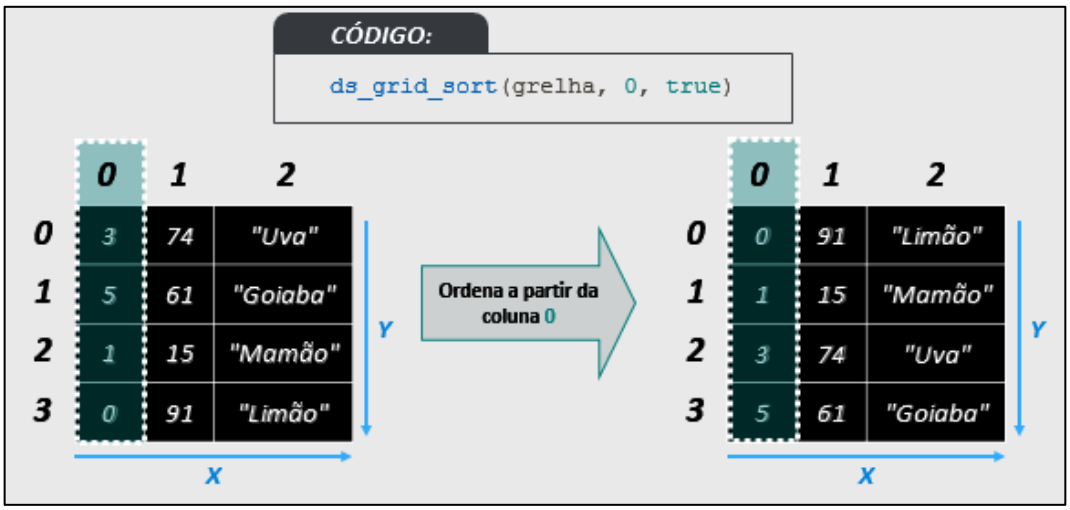
```
1 ds_grid_sort(grelha, 0, true)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_grid_sort* (Figura 3.98) ordena a grelha em ordem alfanumérica crescente ou decrescente escolhendo uma coluna como base. O segundo argumento indica a coluna base para ordenação. O terceiro argumento sendo *true*, ordena de forma crescente, se *false*, de forma decrescente.

A coluna base pode ser melhor explicada olhando o esquema abaixo (Figura 3.99). Nessa ordenação os valores na mesma linha do valor da coluna base também são ordenados. Ou seja, o que ocorre é uma ordenação de linhas, usando uma coluna de base.

Figura 3. 99 – Esquema de ordenação de grelhas



Fonte: Imagem criada pelo autor

3.4.2.10. Função: ds_grid_shuffle

Figura 3. 100 – Função para embaralhar as células da grelha

```
1 ds_grid_shuffle(grelha)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_grid_shuffle* (Figura 3.100) embaralha todas as células da grelha, independente de coluna.

3.4.2.11. Função: ds_exists com parâmetro 'ds_type_grid'

Figura 3. 101 - Função que verifica se a grelha existe

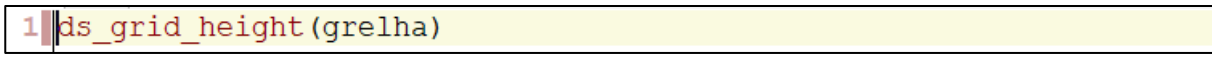
```
1 ds_exists(grelha, ds_type_grid)
```

Fonte: Imagem capturada diretamente do programa

Na instrução acima (Figura 3.101) verificamos se *grelha* existe. No segundo parâmetro indicamos que a estrutura de dados a ser verificada é uma grelha. A função irá retornar *true* caso a lista exista, e *false* caso não esteja.

3.4.2.12. Função: ds_grid_height

Figura 3. 102 – Função que retorna altura da grelha

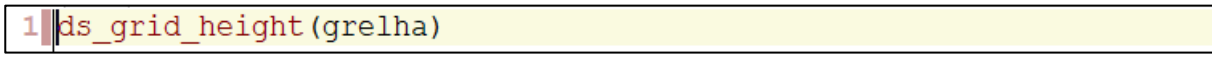


Fonte: Imagem capturada diretamente do programa

A função *ds_grid_height* (Figura 3.102) retorna quantas células há verticalmente na grelha.

3.4.2.13. Função: ds_grid_width

Figura 3. 103 – Função que retorna largura da grelha

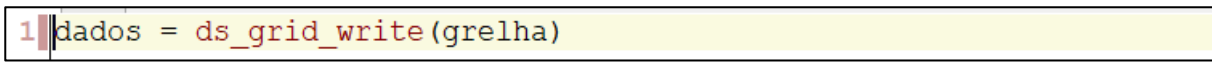


Fonte: Imagem capturada diretamente do programa

A função *ds_grid_width* (Figura 3.102) retorna quantas células há horizontalmente na grelha.

3.4.2.14. Função: ds_grid_write

Figura 3. 104 - Função que gera uma *string* codificada da grelha



Fonte: Imagem capturada diretamente do programa

No trecho acima destacado é gerada uma *string* codificada que é guardada na variável *dados* (Figura 3.104). Essa *string* pode ser carregada novamente em uma grelha usando a função *ds_grid_read*.

3.4.2.15. Função: ds_grid_read

Figura 3. 105 - Função que carrega uma *string* codificada para criar uma grelha

```
1 ds_grid_read(grelha, dados)
```

Fonte: Imagem capturada diretamente do programa

Essa função carrega uma grelha a partir da *string* previamente gerada com a função *ds_grid_write* e guardada na variável *dados* (Figura 3.105). A grelha também deve ter sido criada anteriormente usando *ds_grid_create* e assim usada no primeiro parâmetro.

3.4.3. Filas

Figura 3. 106 - Capa do vídeo 'Estruturas de dados: Filas

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=PvgdcR5pmn0)

No vídeo são apresentadas as filas comuns e suas funções de manipulação (Figura 3.106).

As filas são estruturas ordenadas pela entrada de seus elementos que utilizam o esquema FIFO (First In – First Out). É como em uma fila de banco: O primeiro a chegar é o primeiro a ser atendido.

3.4.3.1. Função: ds_queue_create

Figura 3. 107 - Função para criação de filas

```
1 fila = ds_queue_create();
```

Fonte: Imagem capturada diretamente do programa

A função *ds_queue_create* cria uma fila vazia e retorna um identificador. Na Figura 3.107 a fila é guardada na variável *'fila'*.

3.4.3.2. Função: ds_queue_enqueue

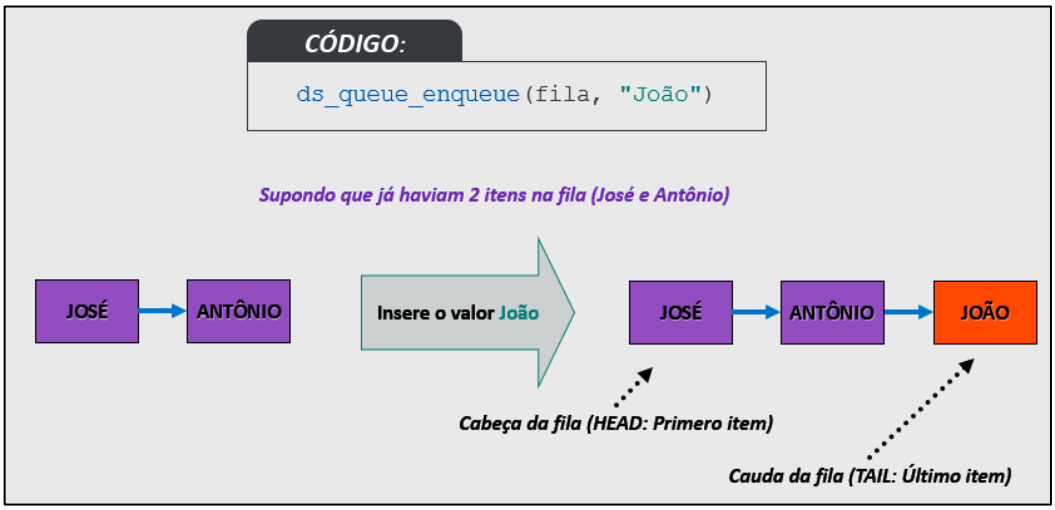
Figura 3. 108 – Função enfileirar itens

```
1 ds_queue_enqueue(fila, "João")
```

Fonte: Imagem capturada diretamente do programa

A função *ds_queue_enqueue* (Figura 3.108) insere dados na cauda da fila, ou seja, por último. Na Figura 3.108 é inserido o valor *João* no final de *fila*. Filas também podem ter valores repetidos, não há restrição quanto a isso. Vejamos como funciona no esquema a seguir (Figura 3.109).

Figura 3. 109 – Esquema enfileiramento de itens



Fonte: Imagem criada pelo autor

3.4.3.3. Função: ds_queue_dequeue

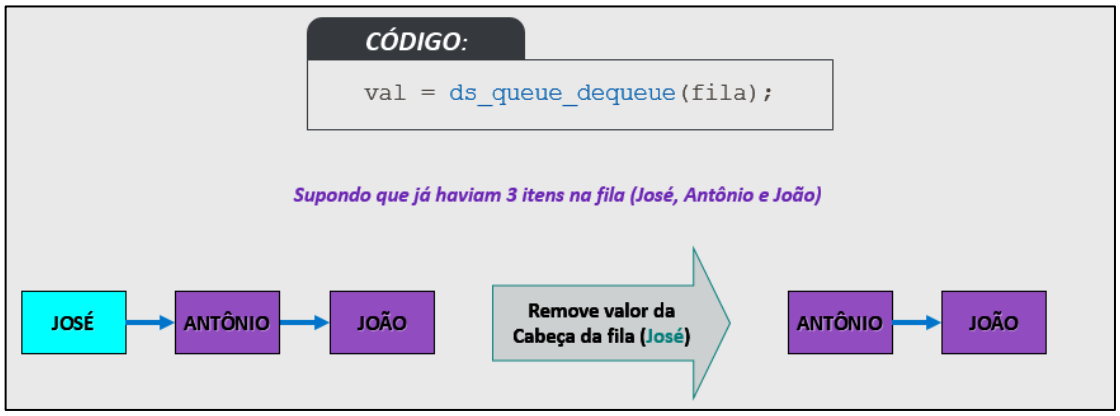
Figura 3. 110 – Função para desenfileirar itens

```
1 val = ds_queue_dequeue(fila);
```

Fonte: Imagem capturada diretamente do programa

A função *ds_queue_dequeue* (Figura 3.110) remove o primeiro elemento (Head) e retorna seu valor. Seu funcionamento pode ser visto no esquema a seguir (Figura 3.111).

Figura 3. 111 – Esquema para desenfileirar de itens



Fonte: Imagem criada pelo autor

3.4.3.4. Funções: ds_queue_head e ds_queue_tail

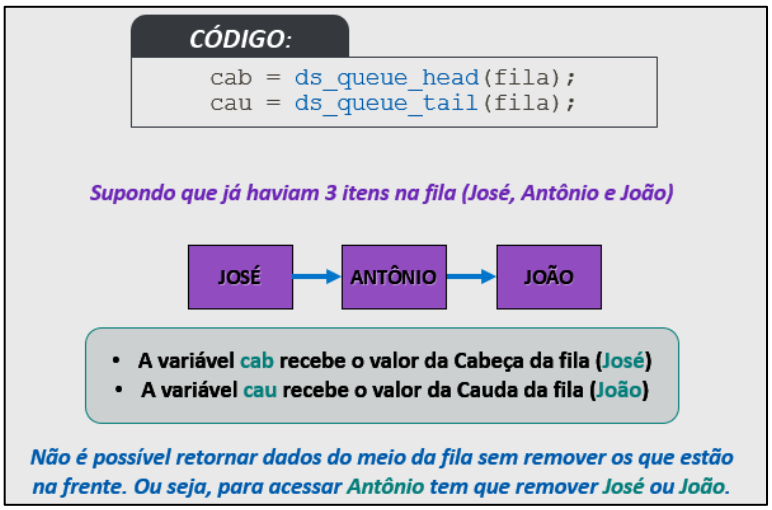
Figura 3. 112 – Funções que retornam valor sem deletar

```
1 cab = ds_queue_head(fila)
2 cau = ds_queue_tail(fila)
```

Fonte: Imagem capturada diretamente do programa

Só é possível saber os valores da cabeça (Primeiro item) e da cauda (Último item) da fila com as funções *ds_queue_head* e *ds_queue_tail* (Figura 3.112). Ou seja, para saber valores intermediários é necessário desenfileirar antes. No esquema a seguir podemos visualizar com maiores detalhes (Figura 3.113).

Figura 3. 113 – Esquema para retornar dados sem deletar



Fonte: Imagem criada pelo autor

3.4.3.5. Função: ds_queue_clear

Figura 3. 114 – Função que remove todos os itens da fila

```
1 ds_queue_clear(fila)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_queue_clear* (Figura 3.114) deleta todos dados da fila, ou seja, a fila fica vazia e pode ser enfileirada novamente.

3.4.3.6. Função: ds_queue_destroy

Figura 3. 115 – Função que remove a fila da memória

```
1 ds_queue_destroy(fila)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_queue_destroy* (Figura 3.115) deleta a fila da memória, ou seja, o identificador usado por essa fila fica com valor indefinido.

3.4.3.7. Função: ds_exists com parâmetro 'ds_type_queue'

Figura 3. 116 - Função que verifica se a fila existe

```
1 ds_exists(fila, ds_type_queue)
```

Fonte: Imagem capturada diretamente do programa

Na instrução acima (Figura 3.116) verificamos se *fila* existe. No segundo parâmetro indicamos que a estrutura de dados a ser verificada é uma fila. A função irá retornar *true* caso a lista exista, e *false* caso não esteja.

3.4.3.8. Função: ds_queue_empty

Figura 3. 117 - Função que verifica se a fila está vazia

```
1 ds_queue_empty(fila)
```

Fonte: Imagem capturada diretamente do programa

No código acima (Figura 3.117) fazemos a verificação com *fila*. A função irá retornar *true* caso a lista esteja vazia, e *false* caso não esteja.

3.4.3.9. Função: ds_queue_size

Figura 3. 118 - Função que retorna o tamanho da fila

```
1 tam = ds_queue_size(fila)
```

Fonte: Imagem capturada diretamente do programa

Na instrução acima (Figura 3.118) é retornada a quantidade de elementos que *fila* possui e guardada na variável *tam*.

3.4.3.10. Função: ds_queue_write

Figura 3. 119 - Função que gera uma *string* codificada da fila

```
1 dados = ds_queue_write(fila)
```

Fonte: Imagem capturada diretamente do programa

No trecho acima destacado é gerada uma *string* codificada que é guardada na variável *dados* (Figura 3.119). Essa *string* pode ser carregada novamente em uma fila usando a função *ds_queue_read.*

3.4.3.11. Função: ds_queue_read

Figura 3. 120 - Função que carrega uma *string* codificada para criar uma fila

```
1 ds_queue_read(fila, dados)
```

Fonte: Imagem capturada diretamente do programa

Essa função carrega uma fila a partir da *string* previamente gerada com a função *ds_queue_write* e guardada na variável *dados* (Figura 3.120). A fila também deve ter sido criada anteriormente usando *ds_queue_create* e assim usada no primeiro parâmetro.

3.4.4. Filas de prioridades

Figura 3. 121 - Capa do vídeo 'Estruturas de dados: Filas de Prioridades'

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=a1uUj-h25kk)

No vídeo são apresentadas as filas de prioridades e suas funções de manipulação (Figura 3.121).

As filas de prioridades são uma mistura de filas com listas. Mas com uma particularidade em especial que é a prioridade que cada valor terá nessa fila. Um exemplo poderia ser a prioridade que idosos e gestantes têm em filas de atendimento.

3.4.4.1. Função: ds_priority_create

Figura 3. 122 - Função para criação de filas de prioridades

```
1 fila = ds_priority_create()
```

Fonte: Imagem capturada diretamente do programa

A função *ds_priority_create* cria uma fila vazia e retorna um identificador. Na Figura 3.122 a fila é guardada na variável *fila*.

3.4.4.2. Função: ds_priority_add

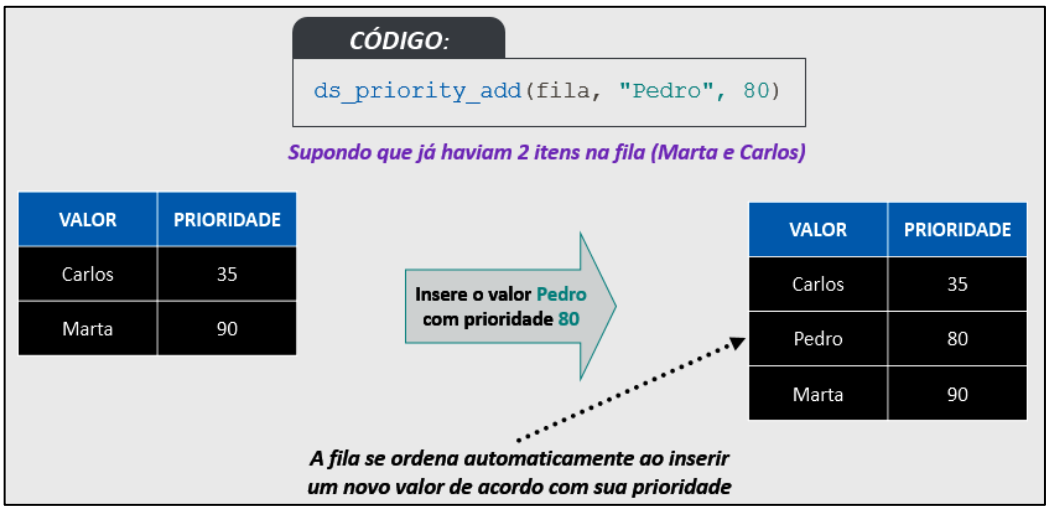
Figura 3. 123 - Função para inserção de valores em filas de prioridades

```
1 ds_priority_add(fila, "Pedro", 80)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_priority_add* (Figura 3.123) insere dados na *fila*, dada uma prioridade (Terceiro parâmetro). A fila se ordena automaticamente após a inserção de novos valores (Figura 3.124). Esse tipo de estrutura também aceita valores repetidos e com prioridades repetidas.

Figura 3. 124 – Esquema para retornar dados sem deletar



Fonte: Imagem criada pelo autor

3.4.4.3. Função: ds_priority_delete_value

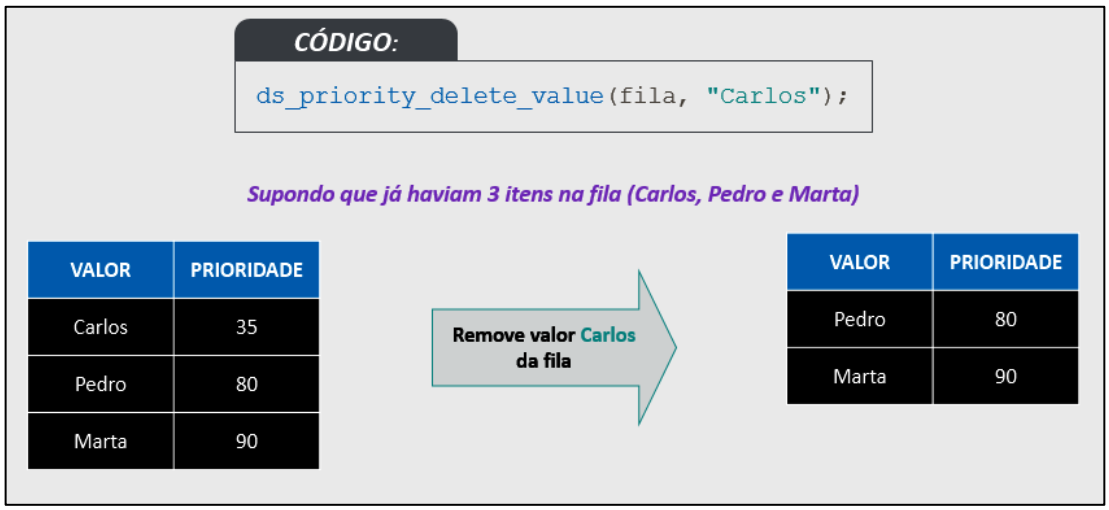
Figura 3. 125 - Função para deletar valores em filas de prioridades

```
1 ds_priority_delete_value(fila, "Carlos")
```

Fonte: Imagem capturada diretamente do programa

A função *ds_priority_delete_value* (Figura 3.125) remove dados na *fila*, dado um valor a ser procurado (Segundo parâmetro). Não há restrições como na fila comum, qualquer valor, desde que exista na fila pode ser removido (Figura 3.126).

Figura 3. 126 – Removendo dados da fila



Fonte: Imagem criada pelo autor

3.4.4.4. Função: ds_priority_delete_max

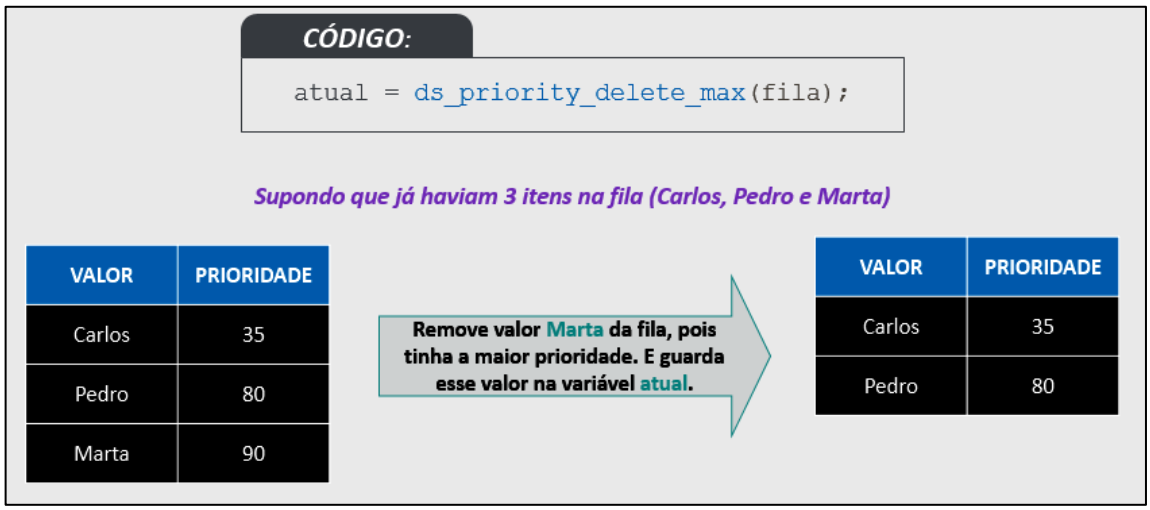
Figura 3. 127 - Função para deletar valor com maior prioridade

```
1 atual = ds_priority_delete_max(fila)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_priority_delete_max* (Figura 3.127) remove o item com maior prioridade na *fila* retornando seu valor, que é guardado na variável *atual*. Vejamos no esquema abaixo (Figura 3.128).

Figura 3. 128 – Removendo valor com maior prioridade



Fonte: Imagem criada pelo autor

3.4.4.5. Função: ds_priority_delete_min

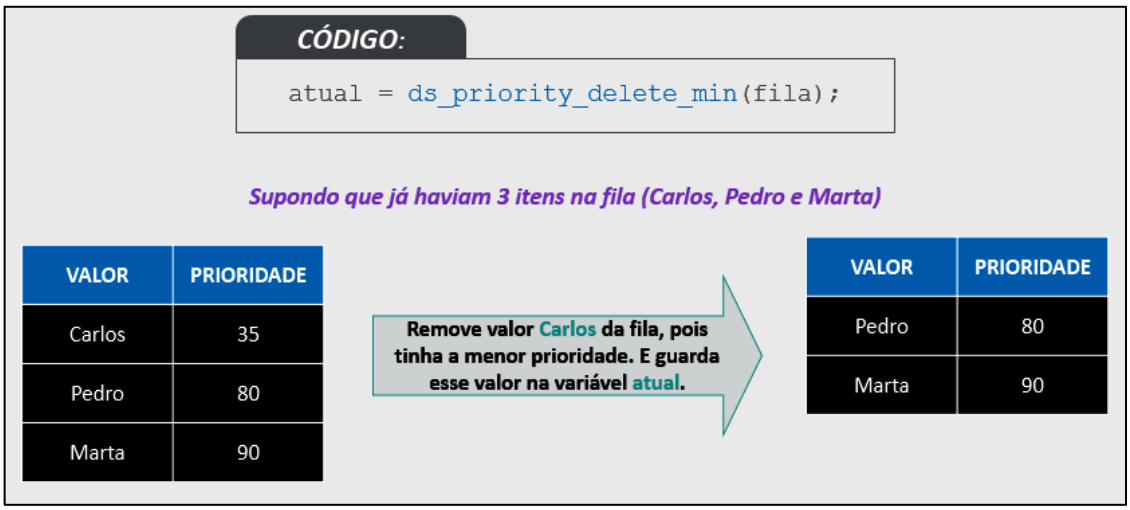
Figura 3. 129 - Função para deletar valor com menor prioridade

```
1 atual = ds_priority_delete_min(fila)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_priority_delete_min* (Figura 3.129) remove o item com menor prioridade na *fila* retornando seu valor, que é guardado na variável *atual*. Vejamos no esquema abaixo (Figura 3.130).

Figura 3. 130 – Removendo valor com menor prioridade



Fonte: Imagem criada pelo autor

3.4.4.6. Função: ds_priority_find_priority

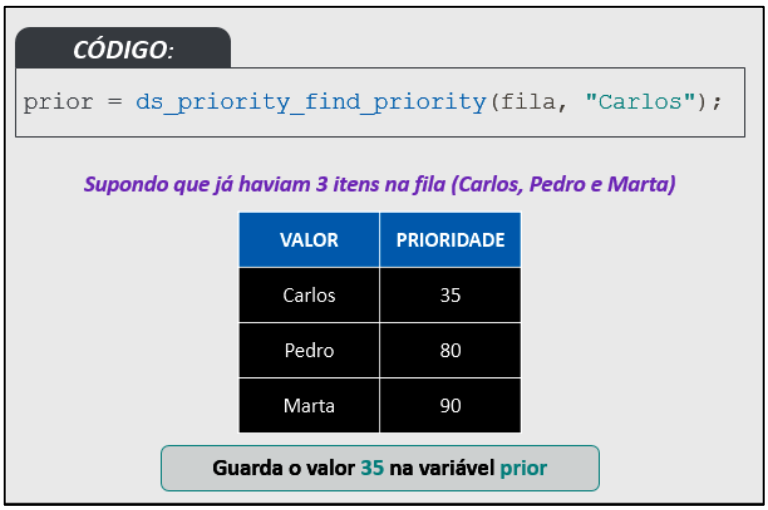
Figura 3. 131 - Função que retorna prioridade de um valor

```
1 prior = ds_priority_find_priority(fila, "Carlos")
```

Fonte: Imagem capturada diretamente do programa

A função *ds_priority_find_priority* (Figura 3.131) procura a prioridade de um dado valor (Segundo parâmetro, *Carlos*), que é guardado na variável *prior*. Caso o valor não exista na fila, retorna um valor indefinido (undefined). Podemos ver um exemplo do funcionamento abaixo (Figura 3.132).

Figura 3. 132 – Procurando prioridade de um valor



Fonte: Imagem criada pelo autor

3.4.4.7. Função: ds_priority_find_max

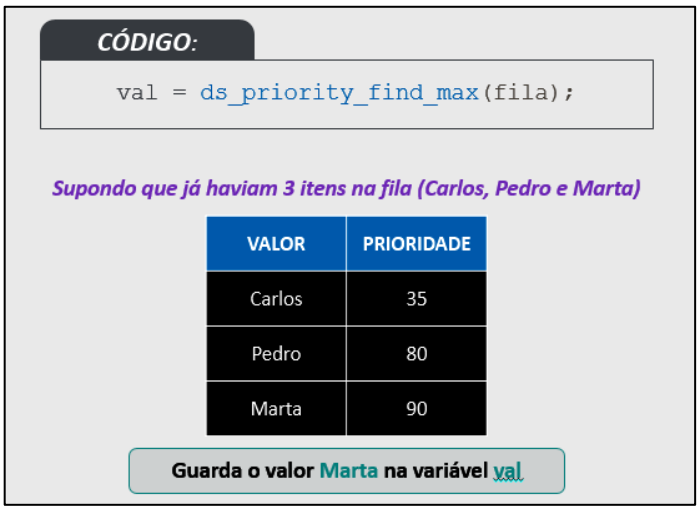
Figura 3. 133 - Função para retornar valor com maior prioridade

```
1 val = ds_priority_find_max(fila)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_priority_find_max* (Figura 3.133) retorna o item com maior prioridade na *fila* sem removê-lo, que é guardado na variável *val*. Vejamos no esquema abaixo (Figura 3.134).

Figura 3. 134 – Retornando valor com maior prioridade



Fonte: Imagem criada pelo autor

3.4.4.8. Função: ds_priority_find_min

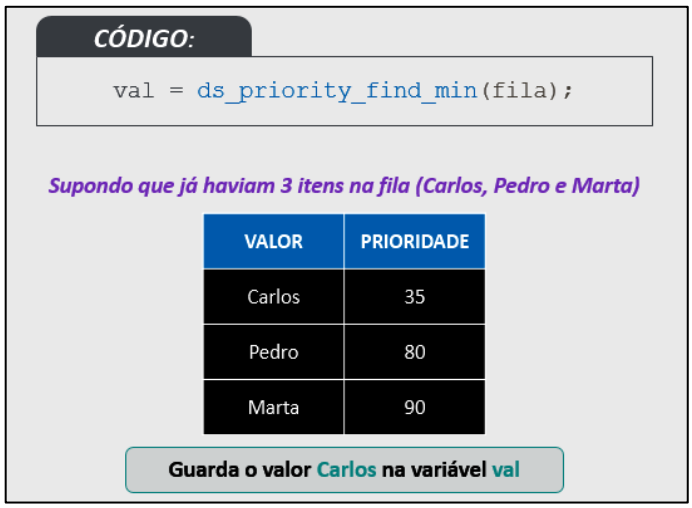
Figura 3. 135 - Função para retornar valor com menor prioridade

```
1 val = ds_priority_find_min(fila)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_priority_find_min* (Figura 3.135) retorna o item com menor prioridade na *fila* sem removê-lo, que é guardado na variável *val*. Vejamos no esquema abaixo (Figura 3.136).

Figura 3. 136 – Retornando valor com menor prioridade



Fonte: Imagem criada pelo autor

3.4.4.9. Função: ds_priority_change_priority

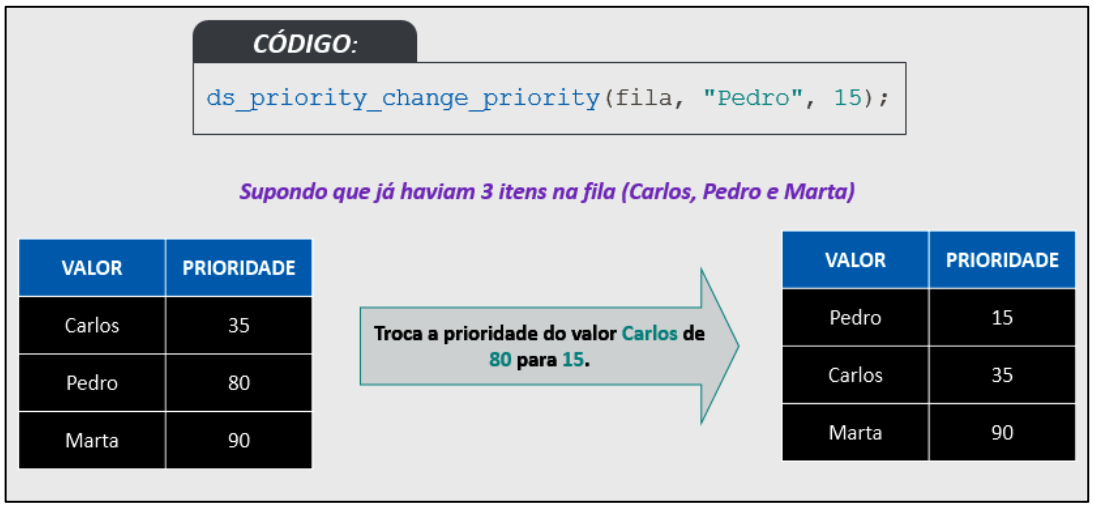
Figura 3. 137 - Função para trocar prioridade de um valor

```
1 ds_priority_change_priority(fila, "Pedro", 15)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_priority_change_priority* (Figura 3.137) troca a prioridade (Terceiro parâmetro) de um item (Segundo parâmetro). Após a mudança a ordenação da fila é atualizada (Figura 3.138).

Figura 3. 138 – Trocando prioridade de um valor



Fonte: Imagem criada pelo autor

3.4.4.10. Função: ds_priority_clear

Figura 3. 139 – Função que remove todos os itens da fila

```
1 ds_priority_clear(fila)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_priority_clear* (Figura 3.139) deleta todos dados da fila, ou seja, a fila fica vazia e pode ser enfileirada novamente.

3.4.4.11. Função: ds_priority_destroy

Figura 3. 140 – Função que remove a fila da memória

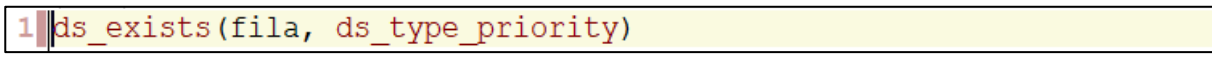
```
1 ds_priority_destroy(fila)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_priority_destroy* (Figura 3.140) deleta a fila da memória, ou seja, o identificador usado por essa fila fica com valor indefinido.

3.4.4.12. Função: ds_exists com parâmetro 'ds_type_priority'

Figura 3. 141 - Função que verifica se a fila existe

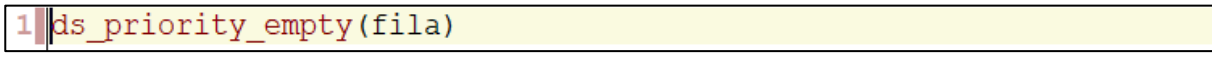


Fonte: Imagem capturada diretamente do programa

Na instrução acima (Figura 3.141) verificamos se *fila* existe. No segundo parâmetro indicamos que a estrutura de dados a ser verificada é uma fila de prioridade. A função irá retornar *true* caso a lista exista, e *false* caso não esteja.

3.4.4.13. Função: ds_priority_empty

Figura 3. 142 - Função que verifica se a fila está vazia

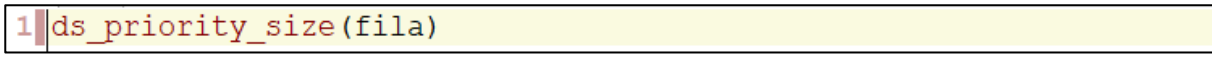


Fonte: Imagem capturada diretamente do programa

No código acima (Figura 3.142) fazemos a verificação com *fila*. A função irá retornar *true* caso a lista esteja vazia, e *false* caso não esteja.

3.4.4.14. Função: ds_priority_size

Figura 3. 143 - Função que retorna o tamanho da fila



Fonte: Imagem capturada diretamente do programa

Na instrução acima (Figura 3.143) é retornada a quantidade de elementos que *fila* possui e guardada na variável *tam*.

3.4.4.15. Função: ds_priority_write

Figura 3. 144 - Função que gera uma *string* codificada da fila

```
1 dados = ds_priority_write(fila)
```

Fonte: Imagem capturada diretamente do programa

No trecho acima destacado é gerada uma *string* codificada que é guardada na variável *dados* (Figura 3.144). Essa *string* pode ser carregada novamente em uma fila usando a função ds_priority_read.

3.4.4.16. Função: ds_priority_read

Figura 3. 145 - Função que carrega uma *string* codificada para criar uma fila

```
1 ds_priority_read(fila, dados)
```

Fonte: Imagem capturada diretamente do programa

Essa função carrega uma fila a partir da *string* previamente gerada com a função *ds_priority_write* e guardada na variável *dados* (Figura 3.145). A fila também deve ter sido criada anteriormente usando *ds_priority_create* e assim usada no primeiro parâmetro.

3.4.5. Pilhas

Figura 3. 146 - Capa do vídeo 'Estruturas de dados: Pilhas'

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=gmxiKlYPbTM)

No vídeo são apresentadas as pilhas e suas funções de manipulação (Figura 3.146).

As pilhas são estruturas ordenadas pela entrada de seus elementos que utilizam o esquema LIFO (Last In – First Out). É como em uma pilha de pratos: O primeiro a limpo é o último a ser guardado.

3.4.5.1. Função: ds_stack_create

Figura 3. 147 - Função para criação de pilhas

```
1 pilha = ds_stack_create()
```

Fonte: Imagem capturada diretamente do programa

A função *ds_stack_create* cria uma pilha vazia e retorna um identificador. Na Figura 3.147 a pilha é guardada na variável *pilha*.

3.4.5.2. Função: ds_stack_push

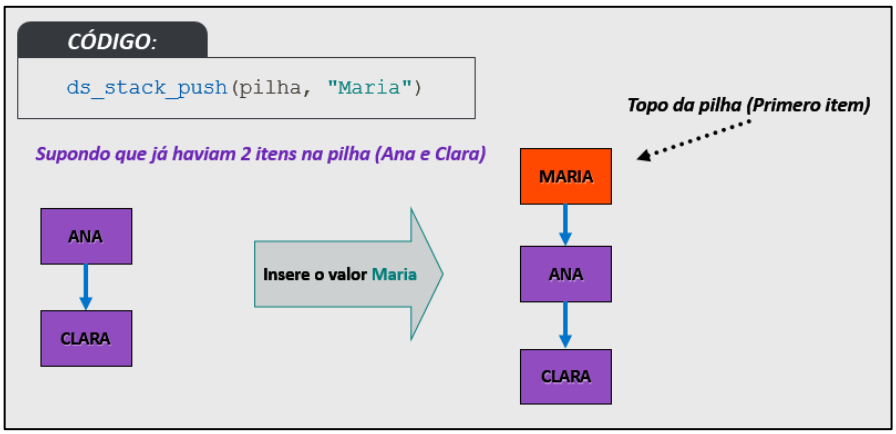
Figura 3. 148 – Função empilhar itens

```
1 ds_stack_push(pilha, "Maria")
```

Fonte: Imagem capturada diretamente do programa

A função *ds_stack_push* (Figura 3.148) insere dados no topo da pilha. Na Figura 3.108 é inserido o valor *Maria* no topo de *pilha*. Pilhas aceitam valores repetidos, já que o que importa é a posição do valor. Vejamos como funciona no esquema a seguir (Figura 3.149).

Figura 3. 149 – Esquema empilhamento de itens



Fonte: Imagem criada pelo autor

3.4.5.3. Função: ds_stack_pop

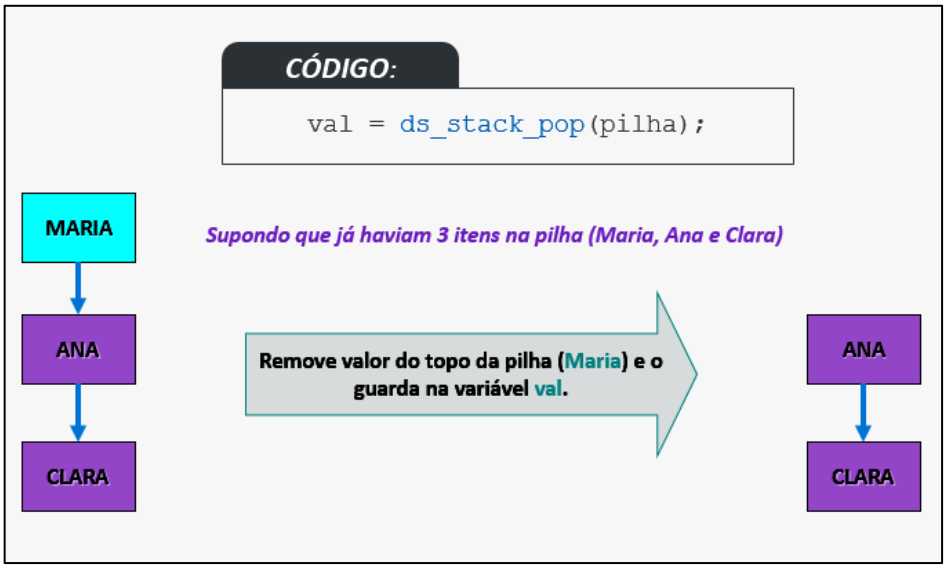
Figura 3. 150 – Função para desempilhar itens

```
1 val = ds_stack_pop(pilha)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_stack_pop* (Figura 3.150) remove o elemento do topo e retorna seu valor. Seu funcionamento pode ser visto no esquema a seguir (Figura 3.151).

Figura 3. 151 – Esquema para desempilhar de itens



Fonte: Imagem criada pelo autor

3.4.5.4. Função: ds_stack_top

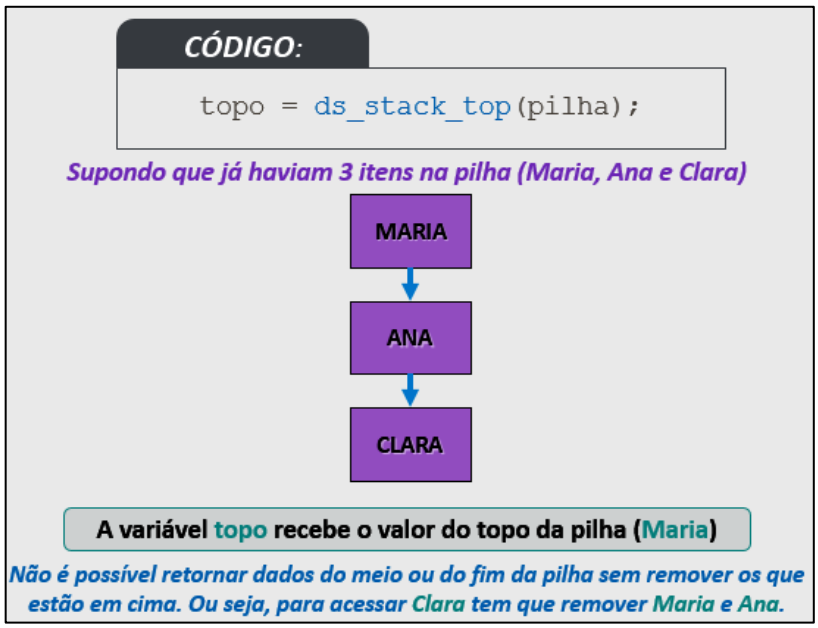
Figura 3. 152 – Função que retorna o valor do topo da pilha

```
1 topo = ds_stack_top(pilha)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_stack_top* (Figura 3.152) retorna o valor do topo da pilha sem removê-lo. No esquema a seguir podemos visualizar com maiores detalhes (Figura 3.153).

Figura 3. 153 – Esquema para retornar dados do topo sem deletar



Fonte: Imagem criada pelo autor

3.4.5.5. Função: ds_stack_clear

Figura 3. 154 – Função que remove todos os itens da pilha

```
1 ds_stack_clear(pilha)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_stack_clear* (Figura 3.154) deleta todos dados da pilha, ou seja, a pilha fica vazia e pode ser empilhada novamente.

3.4.5.6. Função: ds_stack_destroy

Figura 3. 155 – Função que remove a pilha da memória

```
1 ds_stack_destroy(pilha)
```

Fonte: Imagem capturada diretamente do programa

A função *ds_stack_destroy* (Figura 3.155) deleta a pilha da memória, ou seja, o identificador usado por essa pilha fica com valor indefinido.

3.4.5.7. Função: ds_exists com parâmetro ds_type_stack

Figura 3. 156 - Função que verifica se a pilha existe

```
1 ds_exists(pilha, ds_type_stack)
```

Fonte: Imagem capturada diretamente do programa

Na instrução acima (Figura 3.156) verificamos se *pilha* existe. No segundo parâmetro indicamos que a estrutura de dados a ser verificada é uma pilha. A função irá retornar *true* caso a lista exista, e *false* caso não esteja.

3.4.5.8. Função: ds_stack_empty

Figura 3. 157 - Função que verifica se a pilha está vazia

```
1 ds_stack_empty(pilha)
```

Fonte: Imagem capturada diretamente do programa

No código acima (Figura 3.157) fazemos a verificação com *pilha*. A função irá retornar *true* caso a lista esteja vazia, e *false* caso não esteja.

3.4.5.9. Função: ds_stack_size

Figura 3. 158 - Função que retorna o tamanho da pilha

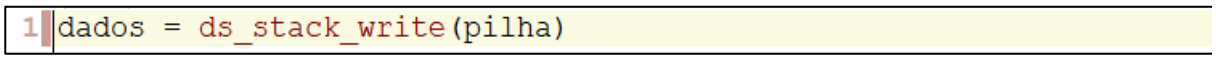
```
1 tam = ds_stack_size(pilha)
```

Fonte: Imagem capturada diretamente do programa

Na instrução acima (Figura 3.158) é retornada a quantidade de elementos que *pilha* possui e guardada na variável *tam*.

3.4.5.10. Função: ds_stack_write

Figura 3. 159 - Função que gera uma *string* codificada da pilha

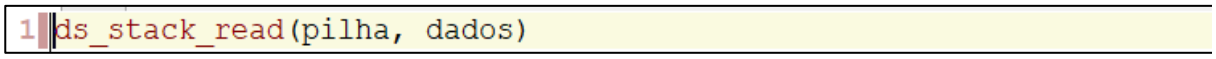


Fonte: Imagem capturada diretamente do programa

No trecho acima destacado é gerada uma *string* codificada que é guardada na variável *dados* (Figura 3.159). Essa *string* pode ser carregada novamente em uma pilha usando a função ds_stack_read.

3.4.5.11. Função: ds_stack_read

Figura 3. 160 - Função que carrega uma *string* codificada para criar uma pilha



Fonte: Imagem capturada diretamente do programa

Essa função carrega uma pilha a partir da *string* previamente gerada com a função ds_stack_write e guardada na variável *dados* (Figura 3.160). A pilha também deve ter sido criada anteriormente usando *ds_stack_create* e assim usada no primeiro parâmetro.

4. JOGOS E EXEMPLOS DESENVOLVIDOS

Este capitulo irá listar todas as produções desenvolvidas durante o projeto afim de especificar os objetivos propostos.

4.1. Blue Runner

Jogo no estilo *runner*, criado com o conhecimento adquirido nas demais aulas (Figura 4.1). O jogo possui gravação local de pontuação, gerador de obstáculos e inimigos, além de um sistema básico de máquina de estados.

Figura 4. 1 - Jogo Blue Runner

Fonte: Imagem capturada diretamente do jogo

No Apêndice B encontram-se os códigos-fonte do projeto bem como a estrutura da árvore de recursos.

Figura 4. 2 - Capa do vídeo 'Criando um Runner – Parte 01'

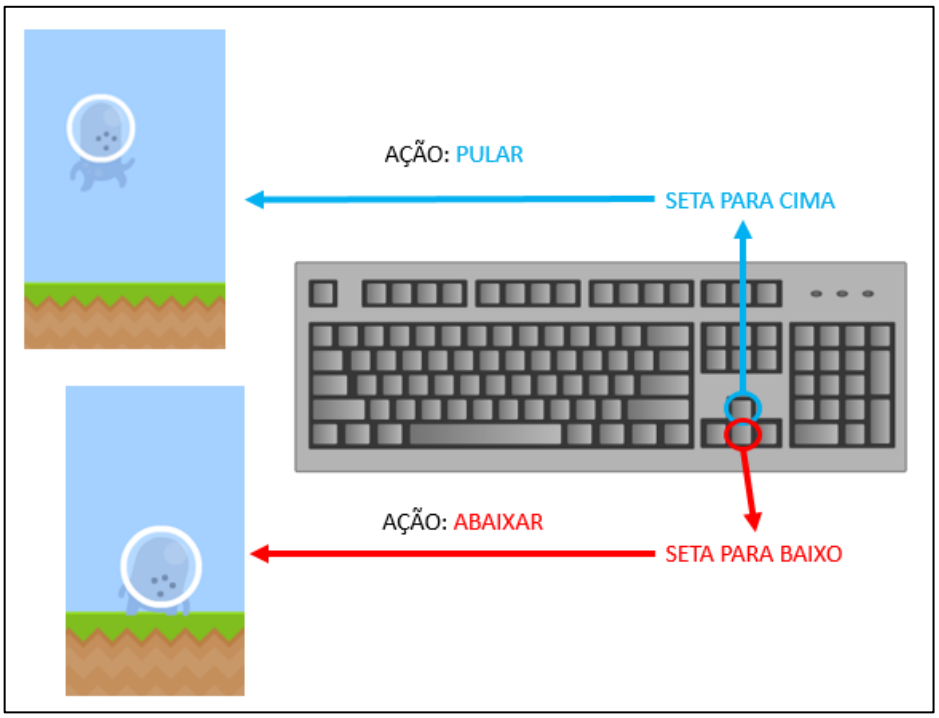
Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=vFz7tNSxr8c)

Nesse vídeo são criados os sistemas básicos do jogo: Movimento, troca de sprites e geração de obstáculos (Figura 4.2).

O objetivo é criar um *Endless Runner* (Corredor infinito, do inglês), onde o personagem segue em frente indefinidamente, desviando de obstáculos. A pontuação é dada pelo tempo que o jogador permanece ‘vivo’ em cena.

Primeiramente é necessário criar um sistema de movimentação que em geral são simples de manusear no *GameMaker*. O personagem tem apenas 3 (três) movimentos: andar, pular e abaixar, porém apenas os dois últimos são controlados pelo jogador via teclado (Figura 4.3).

Figura 4. 3 - Esquema do teclado para ações do jogador



Fonte: Imagem criada pelo autor

Para cada movimento uma imagem diferente deve ser acionada. Como são 3 (três) movimentos, logo são 3 (três) imagens distintas (Figura 4.4) que simbolizam os estados do personagem.

Figura 4.4 - Estados do jogador

Fonte: Imagens sob *Domínio Público*[64] criadas por Kenney (http://www.kenney.nl)

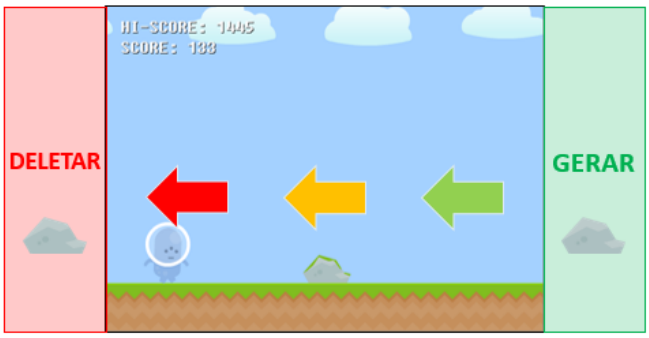
[64] Domínio Público ocorre quando não incidem mais direitos autorais do autor sobre sua obra, podendo, portanto, ser reproduzida livremente por qualquer pessoa.

Os movimentos do personagem são necessários para que ele desvie dos obstáculos que aparecem na tela. A geração desses é feita dinamicamente, já que não existe um cenário infinito, tudo é feito para criar a ilusão de que o percurso não tem fim.

Isso é feito da seguinte maneira: O chão do cenário na verdade é estático. Um *Background* com repetição horizontal e velocidade no eixo X negativa (Esquerda) é que dá a sensação de movimento. Logo é como se fosse uma animação do chão se movendo.

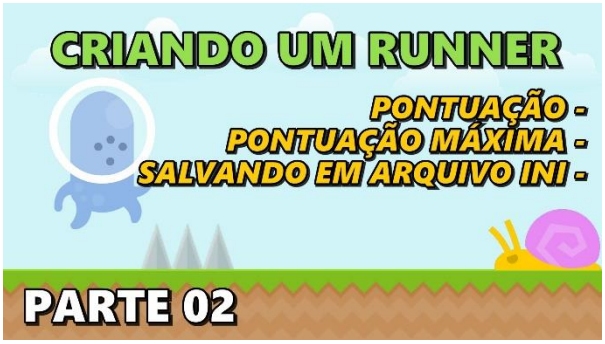
Os obstáculos são gerados na mesma velocidade que o background fora da tela na parte direita. Assim que saem da tela, pela esquerda são removidos da memória para não gerar um *Memory Leak*[65].

Figura 4.5 - Esquema de geração de obstáculos



Fonte: Imagem criada pelo autor

Figura 4.6 - Capa do vídeo 'Criando um Runner - Parte 02'



Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=sE6zxlrjnr8)

Na segunda parte é mostrado como criar um sistema básico de pontuação e como gravar a maior pontuação em arquivo (Figura 4.5).

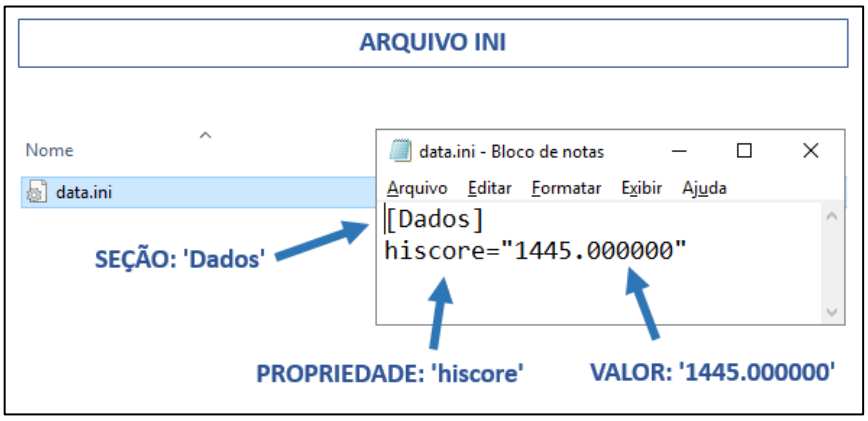
[65] Memory leak, ou estouro de memória, ocorre em sistemas computacionais quando uma parte de memória já alocada não é liberada quando não é mais necessária.

Todo jogo necessita de um sistema que diga ao jogador se ele está vencendo ou perdendo. Utilizamos um sistema simples que conta 30 (Trinta) pontos a cada segundo que o personagem permanece em cena sem perder.

Como as antigas máquinas de Fliperamas, o jogo possui um sistema de pontuação máxima, onde a maior pontuação é gravada em disco para ser exibida em outras execuções.

A gravação é feita através de arquivos do Tipo INI (Figura 4.6), que são arquivos de texto simples com uma estrutura básica composta de "seções" e "propriedades".

Figura 4.7 - Arquivo INI



Fonte: Imagem criada pelo autor

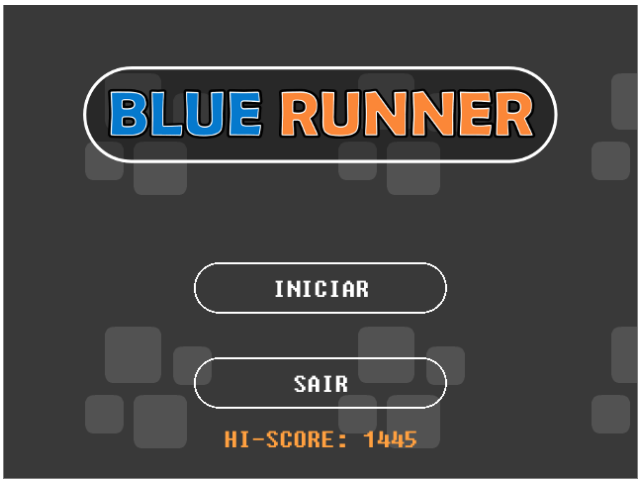
Figura 4.8 - Capa do vídeo 'Criando um Runner - Parte 03'

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v= 8FWdCUJ18B4)

Terceiro vídeo onde o foco fica direcionado para o acabamento do jogo (Figura 4.7), já que os principais sistemas foram criados anteriormente.

A tela de menu é criada com a disposição dos botões e do logotipo centralizados e a informação da maior pontuação na parte inferior (Figura 4.8).

Figura 4.9 - Tela de menu



Fonte: Imagem capturada diretamente do jogo

Ao adicionar as músicas e efeitos sonoros e fazer pequenas correções finalizamos o jogo Blue Runner.

4.2. Exemplos de Movimentação

Foi criada uma série de vídeos sobre movimentação básica, onde foram construídos exemplos básicos de movimentação.

Figura 4.10 - Capa do vídeo 'Introdução: Movimentação Top-Down'

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=IBvG9viYTBg)

Problemas com movimentação são comuns na área de desenvolvimento de jogos. Evitar que esses problemas aconteçam envolve ter um conhecimento mais apurado de ocupação de espaço e ordem de execução dos comandos. O código-fonte do exemplo desenvolvido abaixo está no Apêndice C.

O primeiro passo é conhecer as duas formas mais básicas para se movimentar no GameMaker: Studio, modificado diretamente os eixos X e Y ou utilizando o sistema básico de física.

4.2.1. Movimentação pela alteração da posição dos eixos X e Y

Toda movimentação se baseia nas posições de X e Y, porém alterar diretamente seus valores pode ter suas vantagens como também desvantagens.

A vantagem é criar um sistema simples e rápido com poucas linhas de código, como pode ver no código abaixo (Figura 4.11).

Figura 4.11 – Movimentação Simples

```
1  if keyboard_check(vk_right) and place_free(x + 4, y)
2  {
3      x += 4
4  }
5  if keyboard_check(vk_left) and place_free(x - 4, y)
6  {
7      x -= 4
8  }
9  if keyboard_check(vk_down) and place_free(x, y + 4)
10 {
11     y += 4
12 }
13 if keyboard_check(vk_up) and place_free(x, y - 4)
14 {
15     y -= 4
16 }
```

Fonte: Imagem capturada diretamente do programa

Com apenas isso já é possível ter um movimento visto de cima fluído. Os pontos negativos são as restrições para que esse movimento permaneça assim. São elas:

- Sprites devem ter dimensões que sejam múltiplas do valor em que se da o movimento.
- Somente usar números inteiros para valor do movimento;

No código utilizamos o valor 4, logo as sprites devem ser múltiplos iguais ou maiores a esse valor, pois a checagem não é adaptativa e caso os valores não encaixem o personagem pode ficar preso em paredes ou parar antes de colidir com elas.

4.2.2. Movimentação utilizando o sistema básico de física

A desvantagem desse sistema é que as operações sobre os eixos X e Y ocorrem internamente, sem que o programador possa ter controle total sobre a ação. Porém como é um sistema semiautomático é possível tirar proveito dele e ainda sim fazer uma movimentação coerente.

4.2.3. Movimentação personalizada

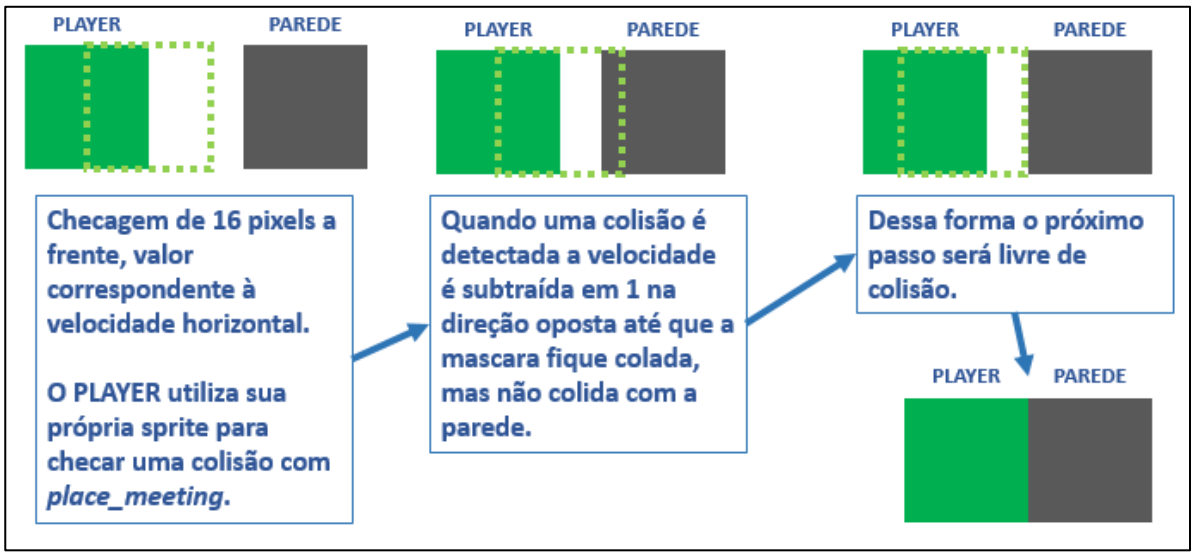
Figura 4.12 - Capa do vídeo 'Sem limites: Movimentação suave'

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=_OjrizQMm3g)

Afim de se livrar das limitações impostas pela movimentação alterando os valores de X e Y, faz-se necessária a criação de um sistema que seja mais dinâmico (Figura 4.12). Para isso é utilizado o sistema básico de física em conjunto com scripts de prevenção de colisão.

A Figura 4.13 descreve como o sistema funciona para evitar que o personagem trave em alguma parede.

Figura 4.13 – Previsão da colisão



Fonte: Imagem criada pelo autor

Tudo funciona como uma previsão de posição futura. Ou seja, sabendo o próximo passa a ser dado podemos verificar se a próxima posição possui colisão com alguma parede, e diminuir a velocidade até que o próximo passo não seja uma colisão.

Dessa maneira não é necessário usar os eventos de colisão que acontecem no momento exato em que uma instância invade a área de outra.

4.2.4. Movimentação em Plataforma

Figura 4.14 - Capa do vídeo 'Plataforma: Pulo, aceleração e velocidade máxima'

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=UGpw3tH14GM)

Com pequenos ajustes é possível transformar o exemplo anterior em um movimento de plataforma com gravidade, pulo, aceleração e limitação de velocidade (Figura 4.14). Isso que torna o sistema interessante, pois tem uso geral.

4.3. Exemplos avançados

Dois exemplos mais complexos foram disponibilizados para estudo: Um sistema de inventário usando grelhas e um exemplo de *SUDOKU*[66].

4.3.1. Sistema de inventário

Figura 4.15 - Capa do vídeo 'Exemplo: Inventário'

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=mR7LSADZ7a4)

Usando o que foi apresentado no vídeo sobre grelhas, foi desenvolvido um exemplo de inventário (Figura 4.15), que é utilizado com frequência em jogos de coleta de itens.

4.3.2. Exemplo de SUDOKU

[66] Jogo de quebra-cabeça.

Figura 4.16 - Capa do vídeo 'SUDOKU – Open Source'

Fonte: Canal *Red Screen Soft* (https://www.youtube.com/watch?v=eFYfraeUFqs)

Esse exemplo (Figura 4.16) faz parte de um trabalho da matéria de Inteligência Artificial do quarto ano, turma de 2017, ministrada pelo Prof. Dr. Cleber Mira na Universidade Estadual do Mato Grosso do Sul. Trata-se de um jogo completo disponibilizado para estudo.

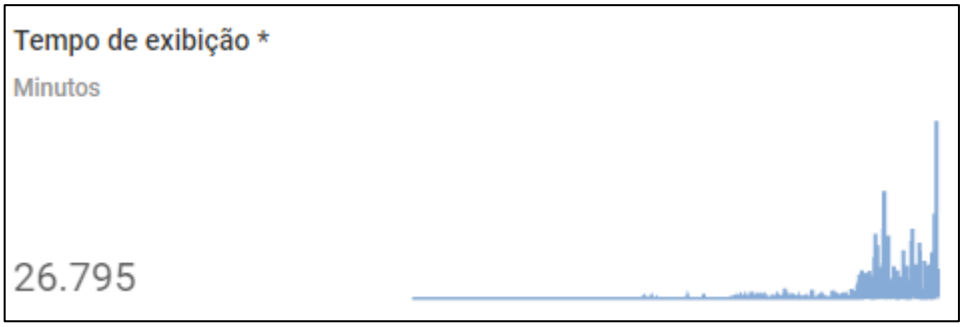
5. RESULTADOS

Para ter em mãos dados que comprovam que o projeto ajudou de alguma maneira foi necessário coletar dados estatísticos do canal, comentários e fazer uma pesquisa com os usuários.

5.1. Estatísticas

Usando a ferramenta *Analytics*[67] do próprio *Youtube*, foi possível constatar que o interesse no assunto é crescente. Vejamos o total de minutos exibidos (Figura 5.1).

Figura 5.1 – Tempo de exibição total



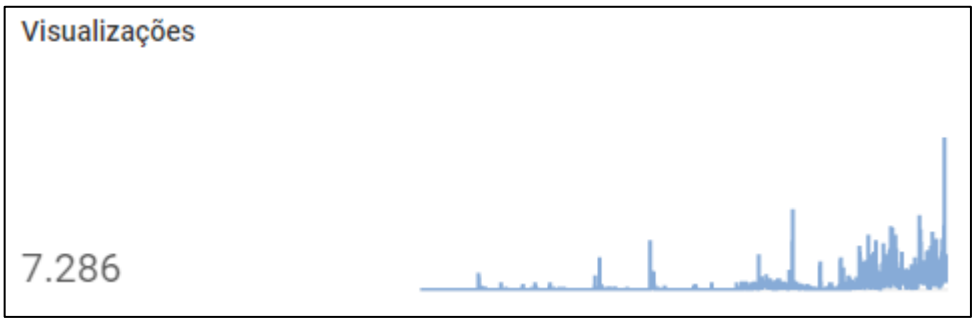
Fonte: Ferramenta Analytics do Youtube

O gráfico mostra um início lento que a partir do último quarto subiu acentuadamente. Isso indica que as pessoas começaram a ficar mais tempo assistindo o vídeo.

O total de visualizações também segue a mesma linha (Figura 5.2).

[67] Coletor de dados estáticos do Google

Figura 5.2 – Tempo de exibição total



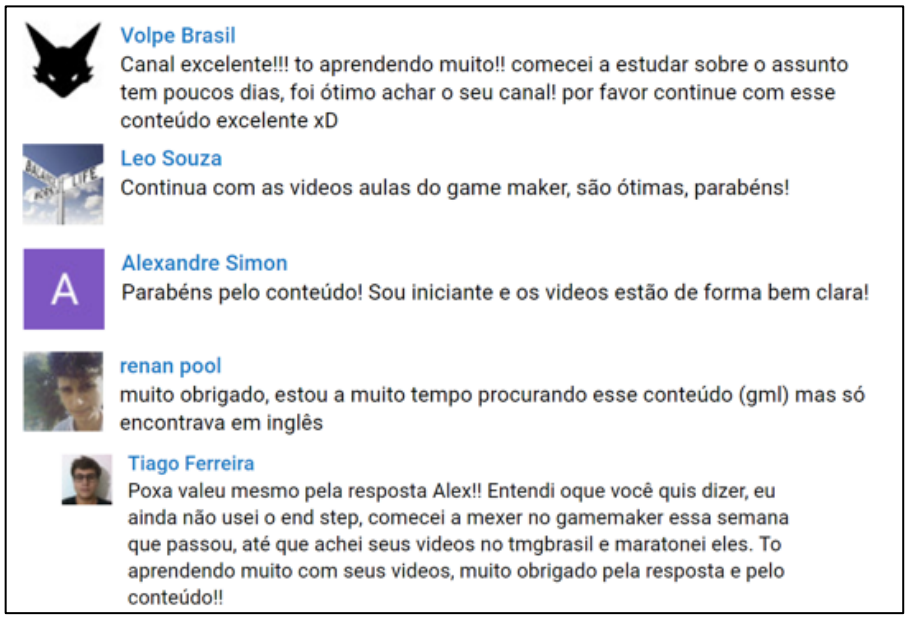
Fonte: Ferramenta Analytics do Youtube

Esse número deve ser colocado em um contexto onde há 184 inscritos e 28 vídeos sobre o assunto. Falando em números de mídias sociais, é pouco, porém devido a especificação do assunto já mencionada os números são até razoáveis.

5.2. Comentários

Outra forma de avaliar é saber o que os usuários espontaneamente comentaram nos vídeos (Figura 5.3).

Figura 5.3 – Comentários



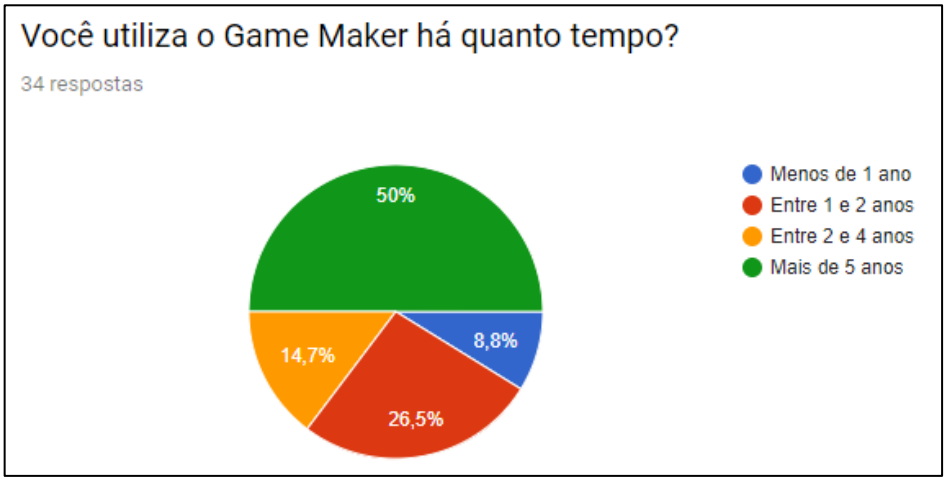
Fonte: Canal *Red Screen Soft* no *Youtube*

5.3. Pesquisa

Utilizando a ferramenta *Forms*[68] do *Google*, pedi aos inscritos do canal que respondessem uma pesquisa. 34 pessoas responderam as perguntas sobre *Game Maker* e programação.

Segundo os dados coletados apenas um quarto dos usuários é iniciante, enquanto metade já conhecia a ferramenta há mais de 5 anos (Figura 5.4).

Figura 5.4 – Pergunta: Você utiliza o Game Maker há quanto tempo?

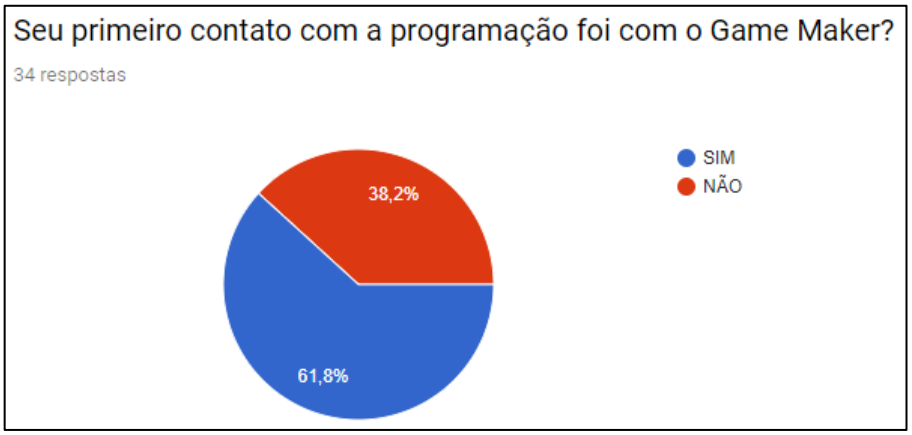


Fonte: Ferramenta *Forms* do *Google*

Mais de 60% dos entrevistados responderam que o primeiro contato com a programação foi utilizando o *Game Maker* (Figura 5.5).

[68] Gerenciador de formulários

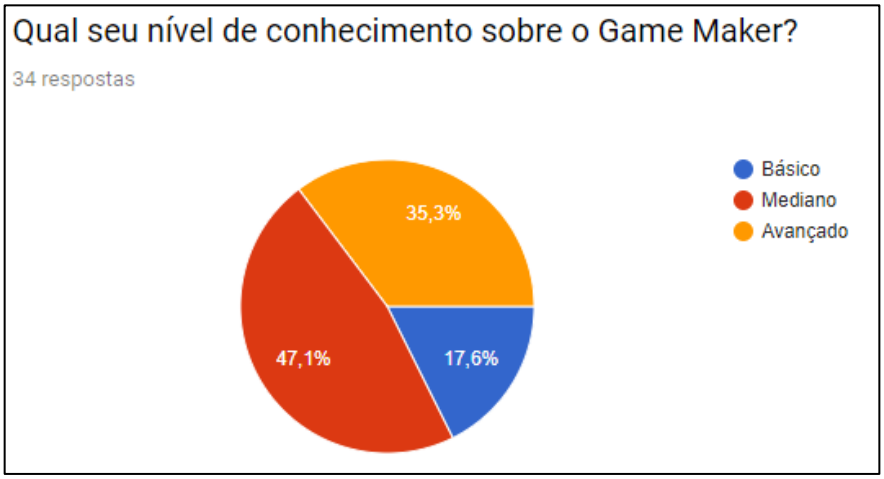
Figura 5.5 – Pergunta: Seu primeiro contato com a programação foi com o Game Maker?



Fonte: Ferramenta *Forms* do *Google*

Quase metade das pessoas classificam seu conhecimento sobre a ferramenta como mediano (Figura 5.6). Isso é bastante relativo, porém é um dado interessante de olhar.

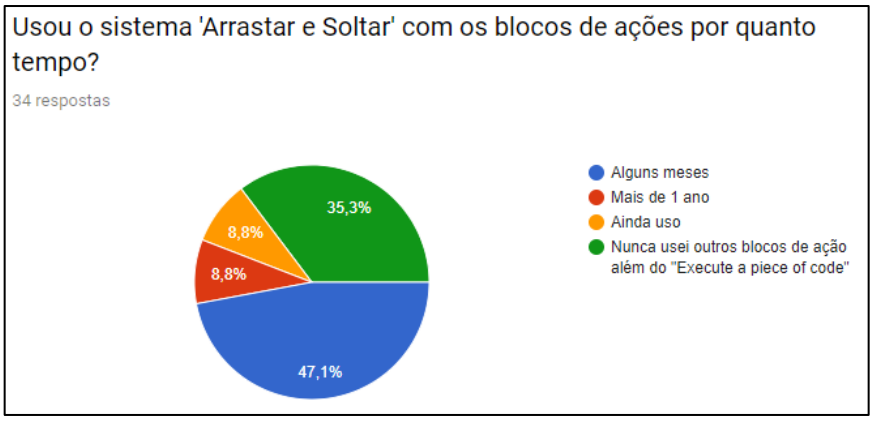
Figura 5.6 – Pergunta: Qual seu nível de conhecimento sobre o Game Maker?



Fonte: Ferramenta *Forms* do *Google*

Entre pessoas que não utilizaram o sistema de arrastar e soltar e as pessoas que utilizaram só por alguns meses soma-se mais de 80% (Figura 5.7). Esse dado demonstra que o sistema é realmente uma introdução à ferramenta.

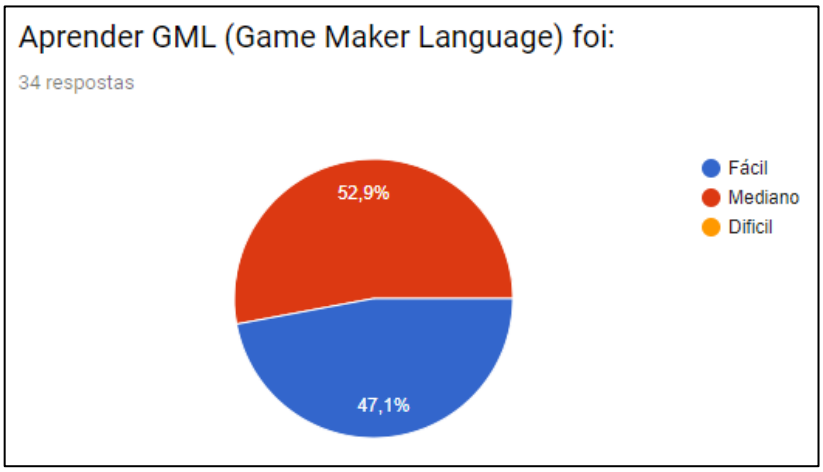
Figura 5.7 – Pergunta: Usou o sistema 'Arrastar e Soltar' com os blocos de ações por quanto tempo?



Fonte: Ferramenta *Forms* do *Google*

Ninguém considerou que aprender GML foi difícil (Figura 5.8).

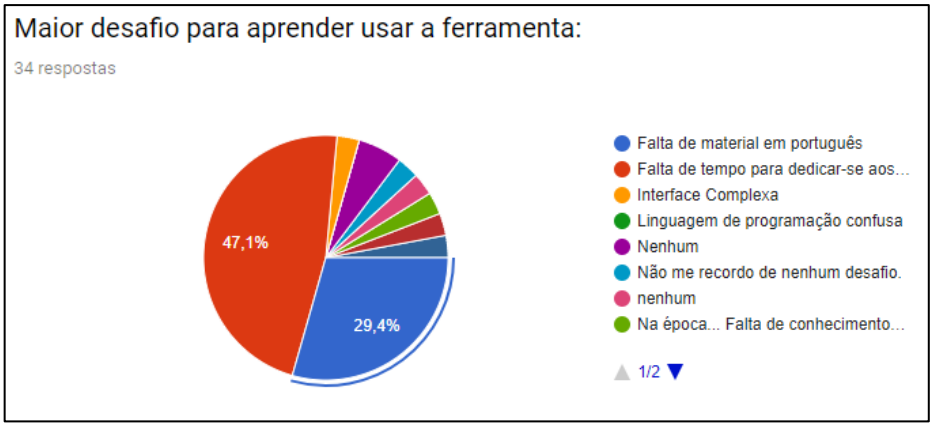
Figura 5.8 – Pergunta: Aprender GML (Game Maker Language) foi:



Fonte: Ferramenta *Forms* do *Google*

Foi apontado por quem respondeu a pesquisa que os dois maiores problemas para aprender usar o *GameMaker: Studio* foram a falta de tempo e a falta de material em português (Figura 5.9).

Figura 5.9 – Pergunta: Maior desafio para aprender usar a ferramenta:



Fonte: Ferramenta *Forms* do *Google*

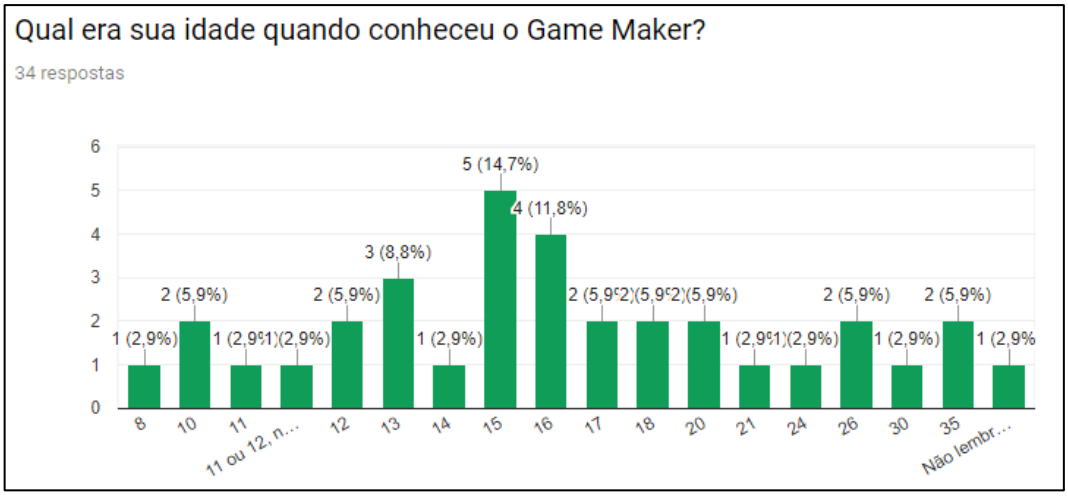
Quase houve unanimidade em recomendar o *GameMaker: Studio* para quem está iniciando (Figura 5.10).

Figura 5.10 – Pergunta: Você recomendaria o Game Maker para alguém que está começando a programar?

Fonte: Ferramenta *Forms* do *Google*

A maioria conheceu a ferramenta entre 15 e 16 anos de idade (Figura 5.11).

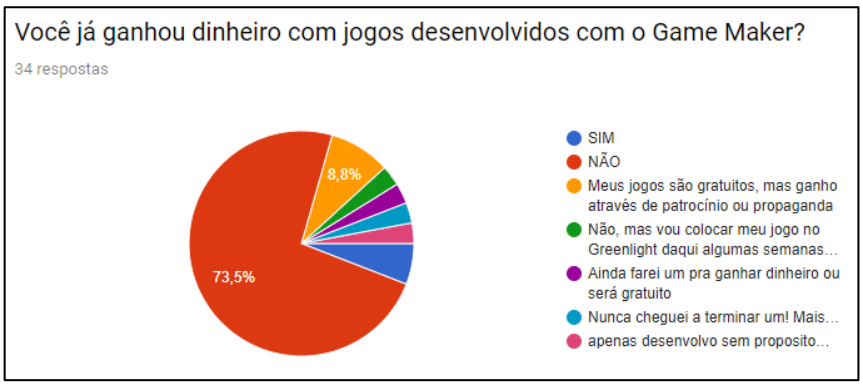
Figura 5.11 – Pergunta: Qual era sua idade quando conheceu o Game Maker?



Fonte: Ferramenta *Forms* do *Google*

A maior parte das pessoas ainda não ganharam dinheiro com jogos feitos no *GM:S* (Figura 5.12).

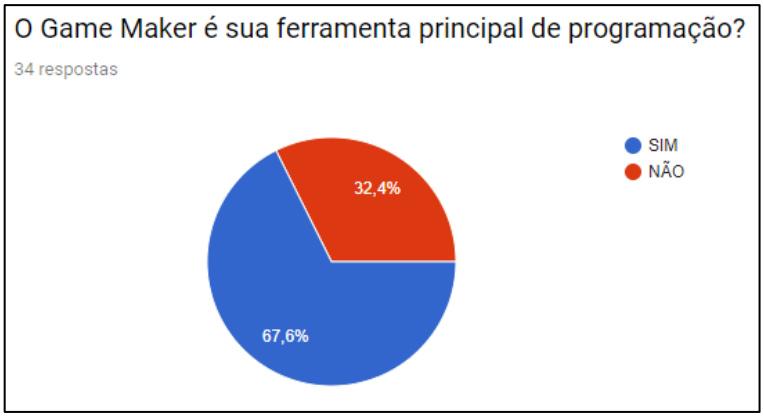
Figura 5.12 – Pergunta: Você já ganhou dinheiro com jogos desenvolvidos com o Game Maker?



Fonte: Ferramenta *Forms* do *Google*

Para quase 70% dos entrevistados o *GameMaker: Studio* é a sua ferramenta principal de programação (Figura 5.13).

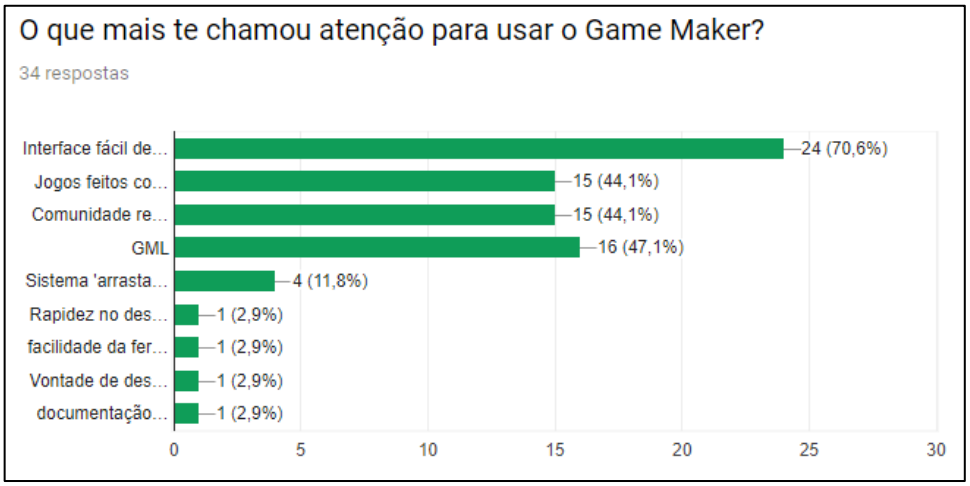
Figura 5.13 – Pergunta: O Game Maker é sua ferramenta principal de programação?



Fonte: Ferramenta *Forms* do *Google*

Os dois itens que mais chamam atenção no *GameMaker: Studio* é a sua interface fácil e sua linguagem GML (Figura 5.14).

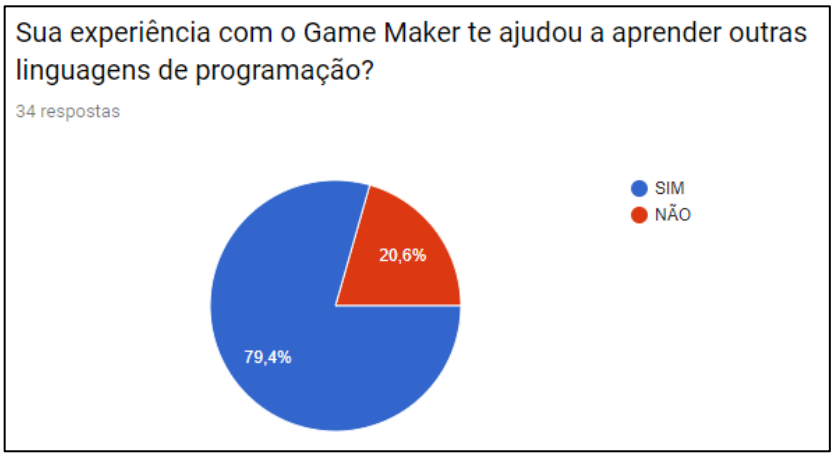
Figura 5.14 – Pergunta: O que mais te chamou atenção para usar o Game Maker?



Fonte: Ferramenta *Forms* do *Google*

O *GameMaker: Studio* ajudou quase 80% das pessoas a aprender outras linguagens de programação (Figura 5.15).

Figura 5.15 – Pergunta: Sua experiência com o Game Maker te ajudou a aprender outras linguagens de programação?



Fonte: Ferramenta *Forms* do *Google*

6. CONCLUSÃO

O tempo em que programar será tão essencial nas vidas das pessoas, quanto realizar operações matemáticas simples, como somar e subtrair está se aproximando. Apesar de parecer um futuro longínquo, é crível que a humanidade chegue próximo a essa realidade.

O ensino de computação ainda tem muito a evoluir para oferecer condições de formar um programador criativo, que não apenas repita técnicas pré-estabelecidos. Um passo deve ser dado por vez e é de se acreditar que esse projeto ajudará, mesmo que minimamente, para que esse processo cresça e se desenvolva.

O projeto atingiu seu objetivo em introduzir a programação à jovens que ainda não ingressaram na faculdade através de um tema chamativo que é o desenvolvimento de jogos. Pode se questionar a quantidade orgânica de pessoas que foram atingidas, porém deve-se levar em consideração que é uma área especifica de conhecimento que ainda tem muito a se desenvolver.

6.1. Dificuldades encontradas

Para trabalhar com mídias sociais digitais convém estar disponibilizando conteúdo com frequência para que haja retenção de público. Como este era o último período da faculdade o tempo foi ficando escasso e o canal chegou a ficar parado dois meses seguidos em duas ocasiões.

6.2. Projetos futuros

Os vídeos produzidos até o momento detalham diversos processos da programação de jogos e estão ajudando pessoas a aprimorarem suas habilidades, como os comentários nos vídeos e nas plataformas (Grupos de redes sociais e fóruns da internet) onde foram divulgados demonstram. Usar jogos para tal tarefa é um desafio, pois é um tema recorrente e que poucas vezes resulta em algo realmente prático ou duradouro. Tendo isso em mente, adaptar o conteúdo apresentado no canal terá que ser uma constante, abordando também outras ferramentas e linguagens, porém mantendo o foco no *GameMaker*.

REFERÊNCIAS

BRESSANE, Andrielle. Pesquisa revela os profissionais mais em falta no mercado mundial, 2015 Disponível em: <http://valejornal.com.br/pesquisa-revela-profissionais-falta-mercado-mundial>. Acesso em: 28 de abr. 2016.

COMPUTER SCIENCE EDUCATION WEEK 2013. The White House. Discurso, 1'18". Disponível em: <https://www.youtube.com/watch?v=yE6IfCrqg3s>. Acesso em: 17 mai. 2016.

FEIJÓ, Bruno; CLUA, Esteban; SILVA, Flávio S. Corrêa da. Introdução à ciência da computação com jogos: aprendendo a programar com entretenimento. Rio de Janeiro: Elsevier Editora Ltda, 2010.

FONSECA FILHO, Cléuzio. História da computação: O caminho do pensamento e da tecnologia. Porto Alegre: Edipucrs, 2007. 204 p.

GADELHA, Julia. A Evolução dos Computadores. Universidade Federal Fluminense, 2001. Pesquisa on-line site Instituto de Computação. Disponível em: <http://www.ic.uff.br/~aconci/evolucao.html> Acesso em 15 de abr. de 2016

JULIA, Paula. Refletindo sobre o jogo, Motriz, v. 2, n. 2, DEZ.1996.

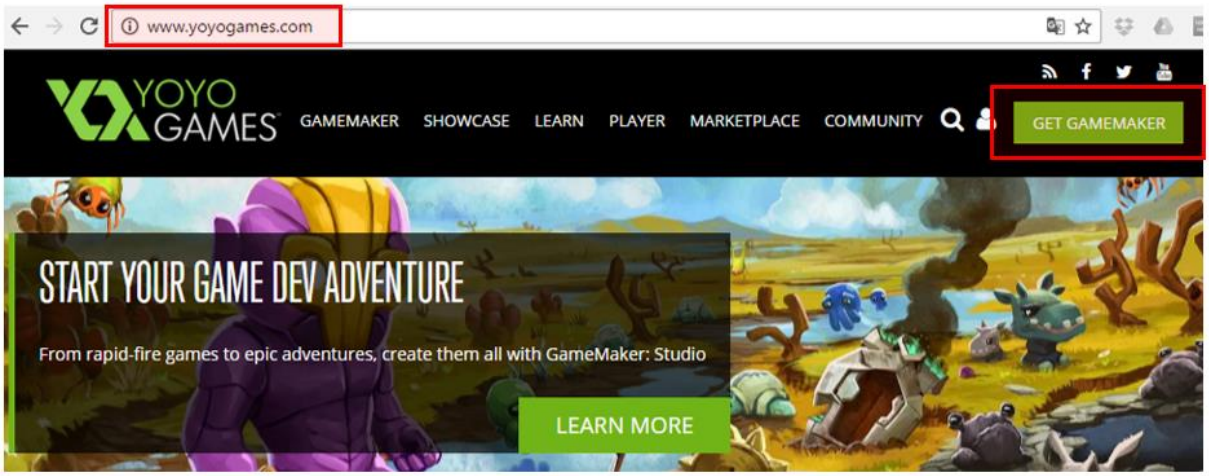
MORATORI, Patrick. POR QUE UTILIZAR JOGOS EDUCATIVOS NO PROCESSO DE ENSINO APRENDIZAGEM? 2003. Disponível em <http://www.virtual.ufc.br/solar/aula_link/lquim/I_a_P/Psicologia_educacao_II/aula_03-7754/imagens/02/Jogos.pdf> Acesso em: 16 jul. 2016.

WANGENHEIM, Christiane Gresse von. Ensinando computação com jogos. Florianópolis: Bookess Editora, 2012.

APÊNDICE A: Instalação do GameMaker: Studio

Acesse o site da produtora *YoYoGames* e clique no botão ***GET GAMEMAKER*** no canto superior direito, como na Figura abaixo:

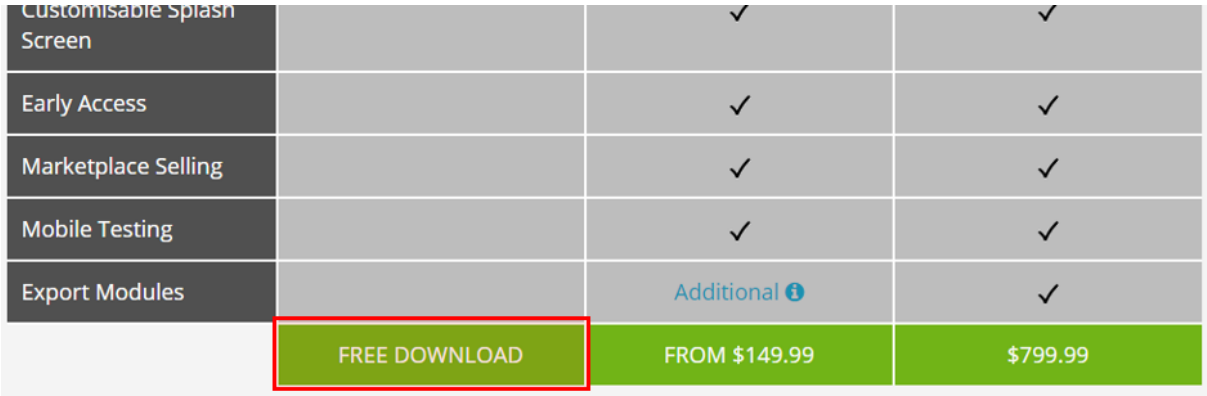
Figura A 1 - Página inicial da YoYoGames



Fonte: Site da *YoYoGames* (https://yoyogames.com/)

Depois serão mostradas as versões disponíveis para aquisição. Clique em ***FREE DOWNLOAD***, como na Figura:

Figura A 2 - Página de download do GameMaker: Studio



Fonte: Site da *YoYoGames* (https://yoyogames.com/)

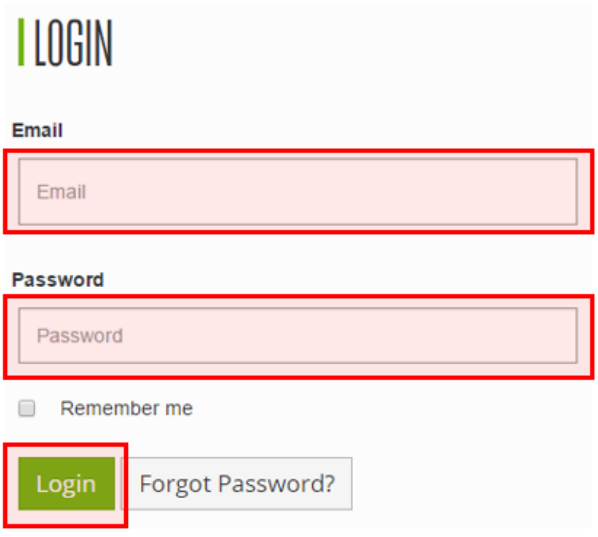
Agora teremos que criar uma conta para realizar o *download*. Preencha os campos corretamente e clique em ***Register***, assim como na Figura:

Figura A 3 - Página de registro da YoYoGames

Fonte: Site da *YoYoGames* (https://yoyogames.com/)

Vá até sua caixa de entrada de e-mails e clique no link de confirmação. Dessa forma você já pode fazer o *login* no site, assim como pode observar na Figura:

Figura A 4 - Página de Login da YoYoGames

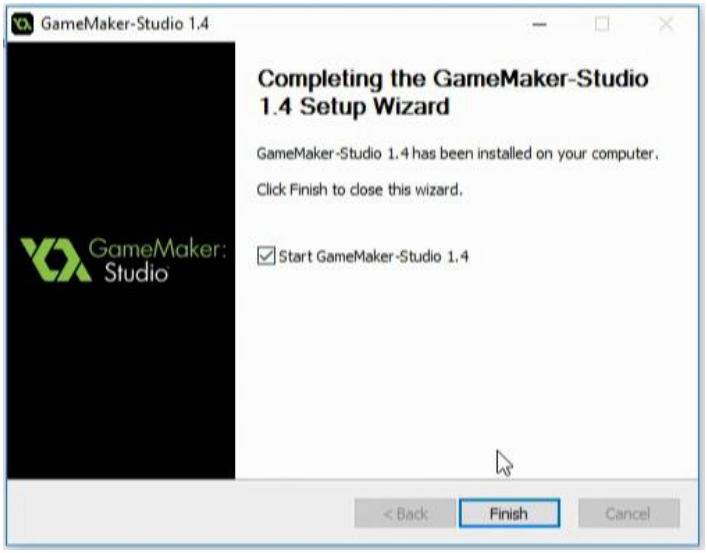


Fonte: Site da *YoYoGames* (https://yoyogames.com/)

Agora apenas clique no botão ***Download GameMaker: Studio 1.4*** e o *download* se iniciará.

Assim podemos iniciar a instalação do *GameMaker: Studio*. Execute o arquivo baixado anteriormente, aceite os termos, escolha o local da instalação e finalize.

Figura A 5 - Completando a instalação



Fonte: Imagem capturada diretamente do programa

Após a extração você será apresentado à esta tela:

Figura A 6 – Fazendo login no programa

Fonte: Imagem capturada diretamente do programa

Basta inserir seu e-mail e senha no quadro central, fechar o programa e executá-lo novamente e estará pronto para uso.

APÊNDICE B: Código-fonte do jogo Blue Runner

SCRIPTS:

Aqui estão listados os scripts personalizados para carregar e salvar pontuações.

Figura B 1 - loadHiscore()

```
/// loadHiscore()

ini_open("data.ini")

    var ss = ini_read_real("Dados", "hiscore", 0)

ini_close()

return ss
```

Figura B 2 - saveHiscore()

```
/// saveHiscore(valor)

ini_open("data.ini")

    ini_write_real("Dados", "hiscore", argument0)

ini_close()
```

OBJECTS:

Abaixo estão listados os códigos de todos os objetos do projeto.

Figura B 3 - objControlMenu - Evento Create

```
audio_play_sound(msMenu, 1, 1)
```

Figura B 4 - objControlMenu - Evento Game Start

```
globalvar hiScore;

hiScore = loadHiscore()

score = 0

randomize()
```

Figura B 5 - objControlMenu - Evento Room End

```
1 audio_stop_sound(msMenu)
```

Figura B 6 - objControlMenu - Evento Draw GUI

```
 1 draw_set_font(fontScore)
 2 draw_set_color(c_orange)
 3
 4 draw_set_halign(fa_center)
 5 draw_set_valign(fa_center)
 6
 7 draw_text(320, 440, "HI-SCORE: " + string(hiScore))
 8
 9 draw_set_halign(fa_left)
10 draw_set_valign(fa_top)
```

Figura B 7 - objButton01 - Evento Mouse => Left Pressed

```
1 room_goto(rmLevel)
2 audio_play_sound(sdSelect, 1, 0)
```

Figura B 8 - objButton01 - Evento Draw

```
 1 draw_self()
 2
 3 draw_set_font(fontScore)
 4 draw_set_color(c_white)
 5
 6 draw_set_halign(fa_center)
 7 draw_set_valign(fa_center)
 8
 9 draw_text(x, y, "INICIAR")
10
11 draw_set_halign(fa_left)
12 draw_set_valign(fa_top)
```

Figura B 9 - objButton02 - Evento Mouse => Left Pressed

```
1 game_end()
2 audio_play_sound(sdSelect, 1, 0)
```

Figura B 10- objButton02 - Evento Draw

```
draw_self()

draw_set_font(fontScore)
draw_set_color(c_white)

draw_set_halign(fa_center)
draw_set_valign(fa_center)

draw_text(x, y, "SAIR")

draw_set_halign(fa_left)
draw_set_valign(fa_top)
```

Figura B 11- objControlGO - Evento Draw GUI

```
var textScore = "HI-SCORE: " + string(hiScore) + "#" + "SCORE: "
 + string(score)

draw_set_font(fontScore)
draw_set_color(c_red)

draw_set_halign(fa_center)
draw_set_valign(fa_center)

draw_text_transformed(320, 70, "GAME OVER", 2, 2, 0)

draw_set_color(c_white)
draw_text(320, 120, textScore)

draw_set_halign(fa_left)
draw_set_valign(fa_top)
```

Figura B 12- objButton03 - Evento Mouse => Left Pressed

```
room_goto(rmLevel)
audio_play_sound(sdSelect, 1, 0)
```

Figura B 13- objButton03 - Evento Draw

```
draw_self()

draw_set_font(fontScore)
draw_set_color(c_white)

draw_set_halign(fa_center)
draw_set_valign(fa_center)

draw_text(x, y, "TENTAR NOVAMENTE")

draw_set_halign(fa_left)
draw_set_valign(fa_top)
```

Figura B 14- objButton04 - Evento Mouse => Left Pressed

```
1 room_goto(rmMenu)
2 audio_play_sound(sdSelect, 1, 0)
```

Figura B 15- objButton04 - Evento Draw

```
 1 draw_self()
 2
 3 draw_set_font(fontScore)
 4 draw_set_color(c_white)
 5
 6 draw_set_halign(fa_center)
 7 draw_set_valign(fa_center)
 8
 9 draw_text(x, y, "IR PARA O MENU")
10
11 draw_set_halign(fa_left)
12 draw_set_valign(fa_top)
```

Figura B 16- objControl - Evento Create

```
 1 background_hspeed[0] = -8
 2 background_hspeed[1] = -4
 3
 4 interval = room_speed * 3
 5
 6 timeCount = 0
 7
 8 deadTime = 0
 9
10 score = 0
11
12 audio_play_sound(msGame, 1, 1)
```

Figura B 17- objControl - Evento Step

```
if instance_exists(objPlayer)
{
    score += 1
}
else
{
    deadTime += 1
}

if deadTime >= 90
{
    if score > hiScore hiScore = score

    saveHiscore(hiScore)

    room_goto(rmGO)
}

timeCount += 1

if timeCount >= interval
{

    if random(10) > 7
    {
        instance_create(room_width, room_height - 190, objBee)
    }
    else
    {
        instance_create(room_width, room_height - 140,
                                                objObstacles)
    }

    interval = room_speed * irandom_range(1.75, 3)

    timeCount = 0
}
```

Figura B 18- objControl - Evento Room End

```
audio_stop_sound(msGame)
```

Figura B 19- objControl - Evento Draw GUI

```
var textScore = "HI-SCORE: " + string(hiScore) + "#" + "SCORE: "
+ string(score)

draw_set_font(fontScore)

draw_set_colour(c_black)

draw_text(21, 21, textScore)

draw_set_colour(c_white)

draw_text(20, 20, textScore)
```

Figura B 20 - objPlayer - Evento Create

```
duck = false
```

Figura B 21 - objPlayer - Evento Destroy

```
audio_play_sound(sdDead, 1, 0)
```

Figura B 22- objPlayer - Evento Step (Gravidade)

```
/// Gravidade
if place_free(x, y + 4)
{
    gravity = 1
}
else
{
    gravity = 0
    vspeed = 0
}
```

Figura B 23- objPlayer - Evento Step (Movimento)

```
/// Movimento

// Pulo
if keyboard_check_pressed(vk_up) and
place_free(x, y + 4) == false
{
    vspeed = -14
    audio_play_sound(sdJump, 1, 0)
}

// Abaixar
if keyboard_check(vk_down) and
place_free(x, y + 4) == false
{
    duck = true
}
else
{
    duck = false
}
```

Figura B 24- objPlayer - Evento Step (Troca de Sprites)

```
/// Troca de sprites
if place_free(x, y + 4)
{
    sprite_index = sprJump
}
else if duck == true
{
    sprite_index = sprDuck
}
else
{
    sprite_index = sprRun
}
```

Figura B 25- objPlayer - Evento Collision => objObstacles

```
instance_destroy()
```

Figura B 26- objPlayer - Evento Collision => objBee

```
instance_destroy()
```

Figura B 27- objObstacles - Evento Create

```
image_speed = 0

image_index = irandom(4)

hspeed = -8
```

Figura B 28- objObstacles - Evento Step

```
1 if x < -200
2 {
3     instance_destroy();
4 }
```

Figura B 29- objBee - Evento Create

```
1 image_speed = 0.25
2
3 hspeed = -8
```

Figura B 30 - objBee - Evento Step

```
1 if x < -200
2 {
3     instance_destroy();
4 }
```

APÊNDICE C: Código-fonte do exemplo de Plataforma

Requisitos: Criação de 2 objetos (*player* e *block*) com suas respectivas sprites.

Figura C 1 - player - Evento Create

```
ground = false;
```

Figura C 2 - player - Evento Step (Parte 1)

```
// Gravidade
gravity = 1.25

// Movimentos horizontais
if keyboard_check(vk_right)
{
    hspeed += 2.25
}
if keyboard_check(vk_left)
{
    hspeed -= 2.25
}

// Pulo
if keyboard_check_pressed(vk_up) and ground == true
{
    vspeed = -15
}
```

Figura C 3 - player - Evento Step (Parte 3)

```

// Limitação da velocidade horizontal
hspeed = min(max(hspeed, -7), 7);

// Fricção
var myFric = 0.5;
if abs(hspeed) >= myFric hspeed -= (sign(hspeed) * myFric) else hspeed = 0;

// Chão
ground = false;
```

Figura C 4 - player - Evento Step (Parte 4)

```
32 // Horizontal
33 i = instance_place(x + hspeed, y, block);
34 
35 while i
36 {
37     if abs(hspeed) < 1 hspeed -= hspeed else hspeed -= sign(hspeed);
38     i = instance_place(x + hspeed, y, block);
39 }
40 // Vertical
41 i = instance_place(x, y + vspeed + gravity, block);
42 if i and (vspeed + gravity) > 0 ground = true;
43 
44 while i
45 {
46     if abs(vspeed + gravity) < 1 vspeed -= (vspeed + gravity) else vspeed -= sign(vspeed + gravity);
47     i = instance_place(x, y + vspeed + gravity, block);
48 }
49 
50 // Diagonal
51 i = instance_place(x + hspeed, y + vspeed + gravity, block);
52 
53 while i
54 {
55     if abs(hspeed) < 1 hspeed -= hspeed else hspeed -= sign(hspeed);
56     if abs(vspeed + gravity) < 1 vspeed -= (vspeed + gravity) else vspeed -= sign(vspeed + gravity);
57     i = instance_place(x + hspeed, y + vspeed + gravity, block);
58 }
```