Criando Labirintos e Masmorras Procedurais Utilizando Game Maker 2

Yuri Barboza Bandeira[1] **, Leandro Rosinak Tibola**[2]

[1] Instituto Federal Farroupilha - Câmpus Frederico Westphalen (IFFAR)

yuri.2021002726@aluno.iffar.edu.br[1],

leandro.tibola@iffarroupilha.edu.br[2]

Abstract. *The digital gaming market is constantly growing, and one of the emerging trends is procedural content generation. This technique enables the creation of game elements randomly, providing a unique experience in every playthrough.*
This study explores the generation of mazes and rooms through different procedural map generation algorithms, detailing their implementation and functionality. Additionally, the GameMaker platform is introduced as a tool for developing these algorithms. Practical tutorials are also provided, demonstrating the step-by-step creation process within GameMaker, facilitating understanding and replication by other developers.

Resumo. *O mercado de jogos digitais está em constante crescimento, e uma das tendências que vem ganhando destaque é a geração procedural de conteúdo. Essa técnica permite a criação de elementos de jogo de forma aleatória, proporcionando uma experiência diversificada a cada nova rodada.*
Neste trabalho, abordaremos a criação de labirintos e salas de forma aleatória, explorando diferentes algoritmos de geração procedural de mapas e descrevendo seus funcionamentos. Além disso, será apresentado o uso da plataforma GameMaker, destacando sua aplicação na implementação desses algoritmos. Também serão desenvolvidos tutoriais práticos demonstrando o processo de criação dos algoritmos dentro do GameMaker, facilitando a compreensão e replicação das técnicas por outros desenvolvedores.

1. INTRODUÇÃO

O mercado de jogos digitais está em constante crescimento, e uma das tendências que vem ganhando destaque é a geração procedural de conteúdo. Essa técnica permite a criação de elementos de jogo de forma aleatória, proporcionando uma experiência diversificada a cada nova rodada do jogo.

Neste trabalho, será apresentado um guia de implementação de Labirintos e Masmorras procedurais, fazendo uso da ferramenta Game Maker 2. Com isso, pretende-se apresentar a implementação de três algoritmos distindos, sendo eles, *Kruskal*, para gerar relevos e mapas, *Backtracking*, para criar labirintos, e *Simple Room Placement*, para criação de masmorras

2. REFERENCIAL TEÓRICO

Esta seção tem como objetivo apresentar uma visão geral dos conceitos fundamentais relacionados ao trabalho em questão. Serão abordados os principais algoritmos utilizados para a geração procedural de labirintos e ambientes, com ênfase nos métodos de *Kruskal*, *Simple Room Placement* e *Backtracking*. Além disso, será discutido o contexto e a aplicação desses algoritmos, destacando sua relevância e os cenários em que são amplamente utilizados. Ao longo da seção, buscamos proporcionar uma compreensão mais profunda dos princípios subjacentes a essas abordagens, suas vantagens, limitações e como elas contribuem para a solução dos problemas propostos.

2.1. O Game Maker 2

O GameMaker é uma plataforma para a criação de jogos 2D amplamente utilizada no cenário de jogos indie (jogos criados por uma equipe pequena, muitas vezes sem um grande investimento), conforme destacado em seu site, que pode ser observado na Figura 1.

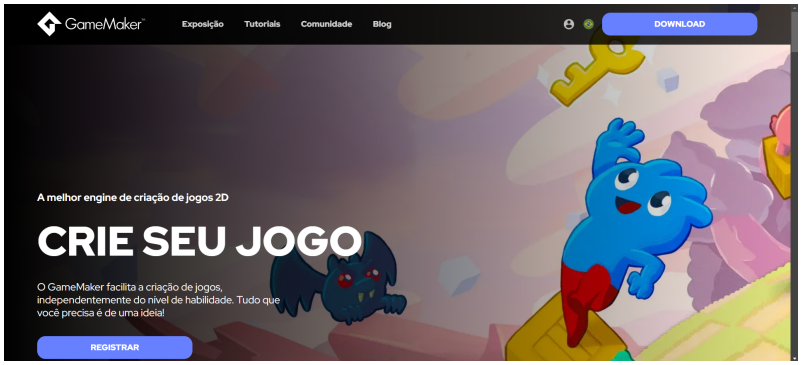

Figura 1. Página inicial do site do GameMaker2

A ferramenta se sobressai por sua acessibilidade, permitindo que desenvolvedores de diferentes níveis de habilidade criem jogos com facilidade. O próprio site [GameMaker 2025] enfatiza essa característica por meio da afirmação: ”O GameMaker facilita a criação de jogos, independentemente do nível de habilidade. Tudo que você precisa é de uma ideia!”. Essa abordagem evidencia a possibilidade de desenvolver jogos sem a necessidade de escrever código, utilizando, em vez disso, a programação baseada em blocos.

Além disso, segundo [GameMaker 2025], a plataforma permite exportação para diversos sistemas, incluindo Windows, macOS, Linux, Android, iOS, HTML5, Xbox, PlayStation e Nintendo Switch.

2.1.1. A Tela do Projeto

Nesta seção, destacamos alguns aspectos importantes da interface do GameMaker2, ilustrada na Figura 2.

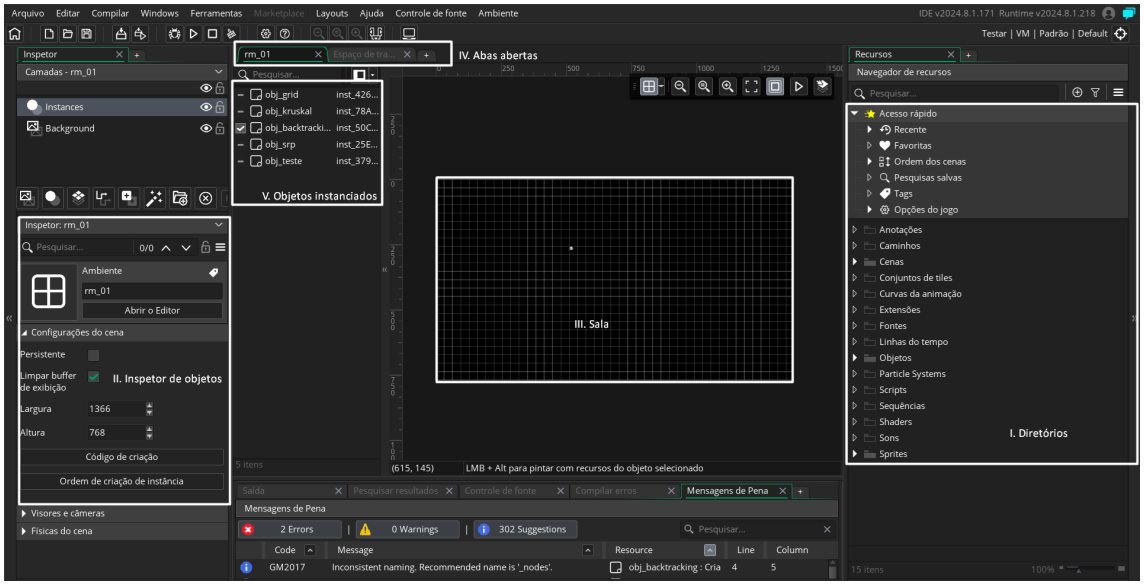

Figura 2. Tela geral do projeto no GameMaker2

Para uma melhor compreensão, descreveremos os principais elementos destacados na imagem.

Item I: representa a estrutura de diretórios do projeto, onde estão armazenadas as pastas contendo os *scripts*, objetos e *sprites* (desenhos) utilizados. Item II: corresponde ao inspetor de objetos, permitindo visualizar e modificar a estrutura e os parâmetros do objeto selecionado. Item III: refere-se à sala, ou room, como é comumente chamada pelos usuários da ferramenta. Nesse espaço, são criadas as instâncias dos objetos. Basicamente, tudo o que aparecer na tela do jogo deve estar presente nessa sala. Item IV: representa as abas abertas. No caso deste projeto, utilizamos apenas a *rm_01* (descrita no Item III) e o espaço de trabalho, onde são criados os *sprites*, *scripts* e objetos. Esse espaço pode ser observado na Figura 3.

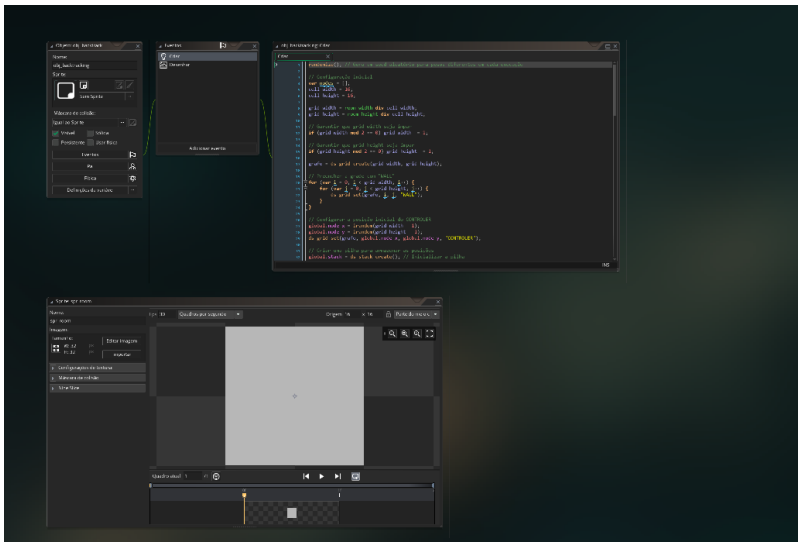

Figura 3. Espaço de trabalho do GameMaker2

Por fim, o Item V exibe todos os objetos instanciados na *rm_01*. Nesse espaço, é possível visualizar, ativar ou desativar os objetos, proporcionando um maior controle sobre os elementos da sala. Aqui podemos destacar que todo objeto contém vários eventos, que podem ser definidos pelo seu menu de eventos, destacados na Figura 4

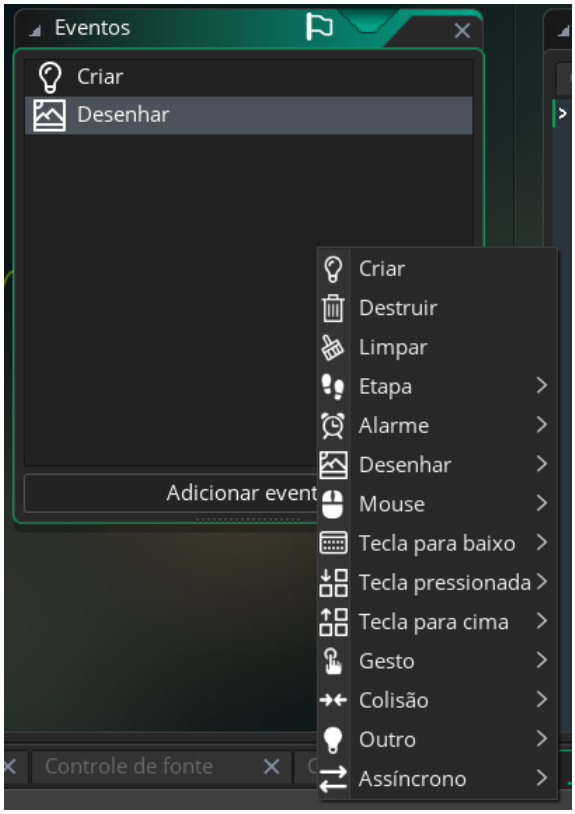

Figura 4. Eventos de um objeto no GameMaker2

Cada evento irá ser executado em um momento específico do seu jogo, como por exemplo o evento *Create*, que é executado assim que o objeto em questão é instanciado.

2.2. Geração Procedural

A geração procedural é uma técnica de criação que utiliza procedimentos, frequentemente implementados por meio de algoritmos especializados, para construir conteúdos de maneira automática. Esses algoritmos são, em sua maioria, randômicos, o que significa que eles utilizam processos de geração de números pseudo-aleatórios para introduzir variação e imprevisibilidade nas saídas geradas. A geração procedural é amplamente usada para criar conteúdos que precisam ser diversificados e únicos a cada execução, como mapas, níveis, paisagens e até mesmo histórias e personagens [Togelius et al. 2011].

A aplicação desses algoritmos está profundamente enraizada na história dos jogos eletrônicos, uma vez que a capacidade de gerar conteúdo de forma automática é especialmente valiosa nesse contexto. Isso é particularmente relevante quando se considera a limitação de espaço de armazenamento e a necessidade de manter o interesse dos jogadores por longos períodos. Uma das primeiras implementações notáveis da geração procedural em jogos é encontrada em Akalabeth: World of Doom, lançado em 1979.

Este jogo é considerado um marco na evolução dos jogos de computador por ter sido um dos primeiros a utilizar algoritmos para criar mundos inteiros de maneira completamente aleatória [Togelius et al. 2011].

Akalabeth utilizava geração de números pseudo-aleatórios para criar as ”seeds”, ou sementes, que são valores iniciais usados para iniciar o processo de geração procedural. Essas seeds garantem que, apesar da aleatoriedade, um mesmo conjunto de dados de entrada (ou seed) sempre produzirá o mesmo resultado, permitindo que mundos gerados aleatoriamente sejam replicáveis. Essa abordagem inovadora para a época, permitiu que os jogadores tivessem experiências diferentes a cada nova partida, mantendo o jogo interessante e desafiador. Atualmente, o uso de seeds é uma prática comum em muitos jogos, como Minecraft, Terraria, Shadows of Doubt, entre outros. Esses jogos permitem que os jogadores compartilhem suas seeds, possibilitando a recriação de mundos específicos e proporcionando uma combinação única de previsibilidade e surpresa [Togelius et al. 2011].

Essa técnica tem evoluído ao longo dos anos e agora se estende além dos jogos, influenciando áreas como design gráfico, criação de conteúdo em mídia digital, inteligência artificial, e até mesmo em simulações científicas. A geração procedural continua a ser uma ferramenta adequada para desenvolvedores que buscam criar experiências diversificadas e ricas, permitindo que vastos universos virtuais sejam criados com relativamente pouca intervenção humana [Togelius et al. 2011].

2.3. Algoritmos utilizados

Entre os diversos algoritmos utilizados para a geração procedural de labirintos e ambientes, foram escolhidos três, sendo eles: *Kruskal*, *Simple Room Placement* e *Backtracking*. Cada um desses métodos foi selecionado devido à sua simplicidade de implementação e eficácia na criação de labirintos dinâmicos, como demonstra a seção 4. O algoritmo de *Kruskal*, com sua abordagem baseada em árvores geradoras mínimas, é conhecido por produzir resultados visualmente interessantes e eficientes. O *Simple Room Placement*, por sua vez, é ideal para gerar ambientes mais estruturados, como os encontrados em jogos de estilo *hack-and-slash*, com uma abordagem focada em salas interconectadas. Já o *Backtracking*, com sua natureza de busca em profundidade, é amplamente utilizado em problemas que exigem soluções mais exploratórias e complexas, sendo eficiente na criação de labirintos com uma estrutura mais orgânica e fluida.

Todas os detalhes de re-implementalção de cada um dos algoritmos citados estão descritos na Seção 2.5

2.3.1. Algoritmo de Kruskal

De acordo com [Li et al. 2017], o Algoritmo Kruskal foi introduzido pela primeira vez em 1956, este algoritmo pertence à categoria dos algoritmos gulosos, que são caracterizados por sempre escolherem a opção mais vantajosa ou de menor custo em cada etapa do processo. A estratégia gulosa é eficaz em muitos casos porque localiza soluções rapidamente sem explorar todas as possibilidades, o que torna esses algoritmos particularmente úteis para problemas de otimização e caminhos mínimos. As principais características

desse algoritmo incluem o fato de que ele começa sua execução com uma aresta previamente ordenada. Esta abordagem ordenada permite que o algoritmo processe de forma eficiente, escolhendo sempre a aresta de menor peso ou custo que ainda não tenha sido incluída no conjunto da árvore geradora mínima. Isso é especialmente vantajoso quando se lida com grafos densos ou de grande escala, onde o número de vértices e arestas pode ser considerável.

Além disso, o algoritmo é flexível em relação à conectividade do grafo no qual opera. Ele não exige que o grafo esteja completamente conectado para funcionar. Em cenários onde o grafo é parcialmente conectado, o algoritmo ainda é capaz de executar sua função, gerando o que é conhecido como uma floresta de abrangência mínima. Essa floresta pode ser entendida como um conjunto de árvores geradoras mínimas, onde cada componente conectado do grafo forma sua própria árvore. Assim, cada componente atua como uma sub-estrutura independente, garantindo que todas as partes do grafo sejam abrangidas pelo menor custo possível, respeitando as restrições de conectividade [Li et al. 2017].

Essa capacidade de lidar com grafos desconectados e ainda produzir uma solução ótima destaca a versatilidade e eficiência do algoritmo, tornando-o uma ferramenta valiosa em várias aplicações, desde redes de computadores até a otimização de rotas de logística e problemas de design de circuitos, [Li et al. 2017].

2.3.2. Simple Room Placement

O algoritmo Simple Room Placement, como descrito por [Amaral 2023], esse algoritmo é utilizado para criar salas ou masmorras de maneira aleatória, o que proporciona uma experiência de jogo única a cada nova execução. O funcionamento do algoritmo é estruturado em três principais etapas, sendo, criação de salas, verificação de salas sobrepostas e distribuição e ligação das salas.

Para começar, o algoritmo gera um conjunto de salas com tamanhos e formas variados, escolhidos aleatoriamente dentro de parâmetros predefinidos. Essa aleatoriedade é fundamental para assegurar que cada sala seja única, contribuindo para a diversidade do mapa criado. Em seguida, o algoritmo realiza uma verificação de sobreposição, onde todas as salas geradas são analisadas para identificar possíveis interseções. Este processo é crucial, pois a sobreposição excessiva de salas pode comprometer a navegabilidade do ambiente e a estética do jogo. Salas que se sobrepõem são identificadas durante uma ”varredura”e, conforme necessário, removidas ou ajustadas para evitar colisões, [Amaral 2023].

Por fim, o algoritmo distribui as salas pelo mapa, garantindo que haja uma distância mínima entre elas. Essa separação não só melhora a clareza visual do ambiente, como também facilita a movimentação do jogador e a implementação de elementos adicionais, como inimigos, itens, e obstáculos. Após o posicionamento final, as salas são conectadas por um ou mais corredores, formando uma rede coesa de entradas e saídas. Esse processo de conexão é vital para a integridade do layout, permitindo que todas as áreas do mapa sejam acessíveis e que o jogador possa explorar o ambiente de forma fluida e contínua. Assim, o Simple Room Placement se torna uma ferramenta poderosa para desenvolvedores que buscam criar mapas complexos e interessantes com um toque

de aleatoriedade, [Amaral 2023].

2.3.3. Algoritmo de Backtracking

Esse tipo de algoritmo, em específico, como cita [Rodrigues and Carvalhaes 2017], é um refinamento da busca por força bruta, onde a exploração exaustiva de possibilidades é sistematizada para encontrar uma solução viável para o problema.

De acordo com [Rodrigues and Carvalhaes 2017], seu funcionamento envolve percorrer uma árvore de decisões de maneira ordenada, começando do topo e avançando sistematicamente de cima para baixo e da esquerda para a direita. A busca continua até que se depare com uma falha ou alcance um nodo terminal, que representa uma possível solução.

Neste ponto, caso não se encontre uma solução satisfatória, o algoritmo emprega uma técnica conhecida como backtracking. Essa técnica é essencial para explorar todas as alternativas, uma vez que ela permite que o algoritmo ”volte atrás”para o último ponto de decisão, retornando pelo mesmo caminho previamente percorrido e buscando novas rotas possíveis. Dessa forma, o algoritmo evita caminhos já explorados que não levam a soluções válidas e continua a busca por alternativas, garantindo que todas as possibilidades sejam consideradas antes de concluir que não há solução ou que uma solução ótima foi encontrada [Van Beek 2006].

Essa abordagem sistemática e exaustiva é especialmente útil em problemas complexos, onde o espaço de busca é vasto e não estruturado, como em problemas de otimização ou na geração de conteúdo aleatório. Ao combinar a força bruta com técnicas inteligentes de exploração e eliminação de caminhos inviáveis, o algoritmo consegue balancear a necessidade de exploração completa com eficiência, evitando um número excessivo de operações desnecessárias [Van Beek 2006].

2.4. Jogos que Utilizam Geração Procedural

Nesta seção, destacamos alguns jogos que fazem uso eficiente da geração procedural na criação de seus mapas e salas. Vale ressaltar que os jogos mencionados apresentam uma aplicação semelhante aos resultados apresentados na Seção 4.

2.4.1. Kruskal

Para o algoritmo de Kruskal, destacamos um jogo do gênero sandbox que se diferencia pela grande liberdade oferecida ao jogador para tomar decisões e realizar tarefas.

O jogo em questão é *WorldBox - God Simulator*, desenvolvido por Maxim Karpenko e lançado em 2021, conforme descrito em sua página na Steam.

Figura 5. Ilha criada no jogo WorldBox

A Figura 5 mostra uma ilha em formato de dragão, cujo contorno foi criado manualmente. No entanto, seu relevo e vegetação foram gerados de forma procedural.

Embora a Figura 5 represente uma ilha desenhada manualmente, o jogo permite a criação de cenários totalmente aleatórios, o que se assemelha bastante aos resultados do algoritmo de Kruskal apresentados na Seção 4.

2.4.2. Backtracking

Os resultados do algoritmo de Backtracking, apresentados na Seção 4, são visualmente semelhantes aos mapas do jogo Labyrinthine, lançado em 2023 pela Valko Game Studios, conforme destacado em sua página na Steam.

Figura 6. Labirinto Apresentado no Jogo Labyrinthine

Labyrinthine é um jogo cooperativo de terror que utiliza seus vastos labirintos para limitar o campo de visão do jogador, criando momentos de surpresa e susto, como ilustrado na Figura 6.

2.4.3. Simple Room Placement

Para o algoritmo Simple Room Placement, compararemos seus resultados com o jogo *The Binding of Isaac*, publicado em 2011 por, Edmund McMillen e Florian Himsl, como apresentado na sua página da Steam.

The Binding of Isaac é um jogo do gênero *Roguelike*, no qual cada nova partida é única, pois os elementos do jogo — desde os itens encontrados até a topologia das salas geradas — são determinados de forma aleatória.

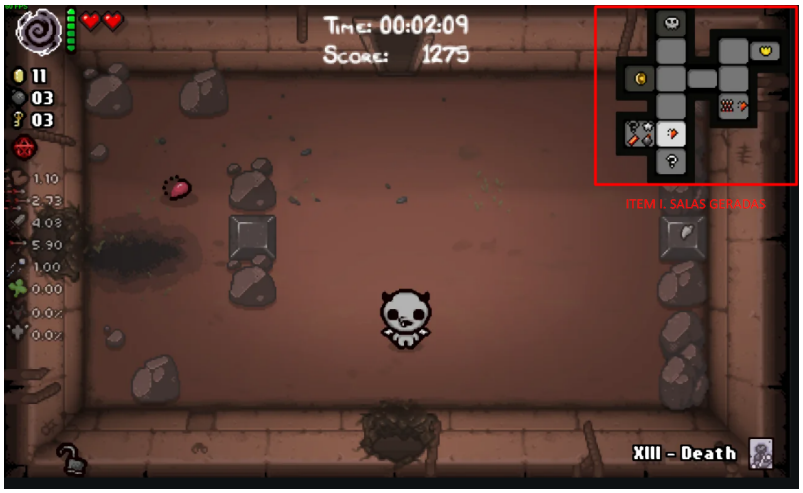

Figura 7. Exemplo de sala no jogo The Binding of Isaac

Como destacado no Item I da Figura 7, o formato da sala apresenta semelhanças com os resultados obtidos na Seção 4 deste trabalho.

2.5. Implementação dos Algoritmos

Nesta seção, são descritos e detalhados os códigos utilizados para a geração dos labirintos pelos algoritmos mencionados na Seção 2.3, cujos resultados são apresentados na Seção 4.

2.5.1. Kruskal

Para criação do labirinto gerado por uma re-implementação de kruskal, primeiro precisamos definir um novo objeto no GameMaker, no exemplo, vamos denominá-lo de *obj_kruskal*. Com o objeto criado e posto na sala, precisamos inserir alguns códigos no seu evento de criação, como mostra a Figura 4.

Código 1: Estrutura de dados para nós e arestas

```
randomize();
// Estruturas de dados para nós e arestas
// Para armazenar os nós
grafo = ds_grid_create(grid_width, grid_height);
arestas = []; // Para armazenar as arestas e seus pesos
// Criar nós e arests
```

```
for (var i = 0; i < grid_width; ++i) {
    for (var j = 0; j < grid_height; ++j) {
        // Posição dos nós
        var x_pos = i * cell_width + 16;
        var y_pos = j * cell_height + 16;

        // Criar um nó fictício na grade
        // ID único baseado na posição
        var node_id = i + j * grid_width;
        grafo[# i, j] = {x: x_pos, y: y_pos, id: node_id};
        // Criar arestas para vizinhos
        // Criando a lista interna

        // Adicionando a lista interna à lista principal
         if (i > 0) { // Vizinho superior
            var vizinho = grafo[# i - 1, j];
            var peso = irandom_range(1, 100); // Peso aleatório
            aresta = []
            array_push(aresta, node_id);
            array_push(aresta, vizinho.id);
            array_push(aresta, peso);
            array_push(arestas, aresta)
        }

        if (j > 0) { // Vizinho esquerdo
            var vizinho = grafo[# i, j - 1];
            var peso = irandom_range(1, 100); // Peso aleatório
            aresta = []
            array_push(aresta, node_id);
            array_push(aresta, vizinho.id);
            array_push(aresta, peso);
            array_push(arestas, aresta)
        }
    }
}
```

Tendo nosso grafo em mãos, podemos começar a construir o objeto de *Kruskal*. Primeiro, vamos ordenar nosso grafo, de acordo com o peso de cada aresta.

Código 2: Ordenação do Grafo

```
    var grafo = obj_grid.arestas;
    // Obtém a quantidade de arestas
    var quantia_arestas = array_length(grafo);
    show_debug_message("Quantidade de arestas: " + string(
        quantia_arestas));
    // Função de comparação para ordenar as arestas pelo peso
    function orderna_arestas(a, b) {
        return a[2] - b[2]; // Compara pelo peso no índice [2]
    }
    // Ordena o grafo (arestas) pelo peso
    array_sort(grafo, orderna_arestas);
```

Após isso iniciamos nossas variáveis de controle, ela nos ajudarão a balancear o

grafo e mapear seus seus vértices representantes.

Código 3: Inicia as estruturas *Union-Find*

```
// Mapeia cada nó ao seu representante
global.parent = ds_map_create();
// Ajuda a balancear as uniões
global.rank = ds_map_create();
```

Então encontramos nossos nós (vértices).

Código 4: Encontra nó *Union-Find*

```
function find(node, parent) {
    if (!ds_map_exists(parent, node)) {
        // Se não existir, é o próprio pai
        ds_map_add(parent, node, node);
        // Inicializa o rank
        ds_map_add(global.rank, node, 0);
    }
    if (parent[? node] != node) {
        // Caminho compressão
        parent[? node] = find(parent[? node], parent);
    }
    return parent[? node];
}
```

Após isso, fazemos a união dos nossos nós, de acordo com as estruturas criadas anteriormente.

Código 5: União dos nós *Union-Find*

```
function union(node1, node2, parent, rank) {
    var root1 = find(node1, parent);
    var root2 = find(node2, parent);
    if (root1 != root2) {
        if (rank[? root1] > rank[? root2]) {
            // root1 se torna o pai de root2
            parent[? root2] = root1;
        } else if (rank[? root1] < rank[? root2]) {
            // root2 se torna o pai de root1
            parent[? root1] = root2;
        } else {
            // root1 se torna pai e aumenta o rank
            parent[? root2] = root1;
            rank[? root1] += 1;
        }
        return true;
    }
    return false;
}
```

Por fim, definimos nossa MST *(Minimum Spanning Tree)*, ou Árvore Geradora Mínima.

Código 6: Definição da MST

```
// Inicializa o array da Árvore Geradora Mínima (MST)
mst = [];
// Itera sobre as arestas do grafo
for (var i = 0; i < quantia_arestas; i++) {
    var aresta = grafo[i];
    var node1 = aresta[0]; // Primeiro nó da aresta
    var node2 = aresta[1]; // Segundo nó da aresta
    var peso = aresta[2];  // Peso da aresta
    // Adiciona a aresta à MST se não formar ciclo
    if (union(node1, node2, global.parent, global.rank)) {
        array_push(mst, aresta); // Adiciona a aresta à MST
    }
}
```

Então agora só precisamos iterar sobre nossa MST para desenhar o resultado. Para isso vamos utilizar o evento Draw ou Desenhar do GameMaker2.

Código 7: Desenhando a MST

```
// Criar um array para armazenar as posições dos nós
var node_positions = array_create(obj_grid.grid_width * obj_grid.
    grid_height);
// Mapeando as posições dos nós
for (var i = 0; i < obj_grid.grid_width; ++i) {
    for (var j = 0; j < obj_grid.grid_height; ++j) {
        var x_pos = i * obj_grid.cell_width + 16;
        var y_pos = j * obj_grid.cell_height + 16;
        //Armazenar as coordenadas do nó (x_pos, y_pos) na
        //lista de posições
        var node_index = i + j * obj_grid.grid_width;
        node_positions[node_index] = [x_pos, y_pos];
        instance_create_depth(x_pos, y_pos, 1, obj_wall);
    }
}
// Iterar pela MST para criar objetos apenas no ponto médio das
//conexões
for (var i = 0; i < array_length(mst); i++) {
    var node1 = mst[i][0];
    var node2 = mst[i][1];
    // Acessar as coordenadas de cada nó a partir do array
    var x1 = node_positions[node1][0];
    var y1 = node_positions[node1][1];
    var x2 = node_positions[node2][0];
    var y2 = node_positions[node2][1];
    // Calcular o ponto médio entre os dois nós
    var mid_x = (x1 + x2) / 2;
    var mid_y = (y1 + y2) / 2;
    // Criar um objeto no ponto médio da conexão
    instance_create_depth(mid_x, mid_y, 0, obj_floor);
}
```

2.5.2. Backtracking

O funcionamento do algoritmo de *Backtracking* é muito semelhante à *Kruskal*, já que também trabalhamos com um grafo, a única diferença, é que aqui percorremos todos os seus pontos, para isso, vamos utilizar um objeto denominado *"Controller"*, acompanhe com os códigos. Até o código 5, todos os *scripts* deverão ser inseridos no evento *Create* do objeto.

Código 1: Cria o Grafo

```
// Gera um seed aleatório para pesos diferentes em cada execução
randomize();
// Configuração inicial
var nodes = [];
cell_width = 16;
cell_height = 16;

grid_width = room_width div cell_width;
grid_height = room_height div cell_height;

// Garantir que grid_width seja ímpar
if (grid_width mod 2 == 0) grid_width -= 1;

// Garantir que grid_height seja ímpar
if (grid_height mod 2 == 0) grid_height -= 1;

grafo = ds_grid_create(grid_width, grid_height);
```

Código 2: Inicia o Objeto Controler e Prepara o Cenário

```
// Preencher a grade com "WALL"
for (var i = 0; i < grid_width; i++) {
    for (var j = 0; j < grid_height; j++) {
        ds_grid_set(grafo, i, j, "WALL");
    }
}
// Configurar a posição inicial do CONTROLER
global.node_x = irandom(grid_width - 1);
global.node_y = irandom(grid_height - 1);
ds_grid_set(grafo, global.node_x, global.node_y, "CONTROLER");
```

Código 3: Cria uma pilha para armazenar as posições visitadas

```
// Inicializar a pilha
global.stack = ds_stack_create();
// Adicionar a posição inicial
ds_stack_push(global.stack, [global.node_x, global.node_y]);
```

Código 4: Verficações e Movimentações do*Controller*

```
// Função para verificar se há paredes ao redor
function walls_around(  x, y) {
    var moves = [
```

```
        [x - 2, y],      // Esquerda
        [x + 2, y],      // Direita
        [x, y - 2],      // Cima
        [x, y + 2]       // Baixo
    ];
    for (var d = 0; d < array_length(moves); d++) {
        var check_x = moves[d][0];
        var check_y = moves[d][1];
        if (check_x >= 0 && check_x < grid_width && check_y >= 0 &&
            check_y < grid_height) {
            if (ds_grid_get(grafo, check_x, check_y) == "WALL") {
                return true;
            }
        }
    }
    return false;
}
// Função para mover o controller
function move_controller() {
    var moved = false;
    var moves = [
        [0, -2], // Cima
        [0, 2],  // Baixo
        [-2, 0], // Esquerda
        [2, 0]   // Direita
    ];
    while (!moved && walls_around(global.node_x, global.node_y)) {
        var move_index = irandom(array_length(moves) - 1);
        var dx = moves[move_index][0];
        var dy = moves[move_index][1];
        var new_x = global.node_x + dx;
        var new_y = global.node_y + dy;
        // Verificar se a nova posição é válida
        if (new_x >= 0 && new_x < grid_width && new_y >= 0 && new_y <
            grid_height) {
            if (ds_grid_get(grafo, new_x, new_y) == "WALL") {
                // Atualizar células
                // Atual posição vira FLOOR
                ds_grid_set(grafo, global.node_x, global.node_y, "FLOOR
                    ");
                // Intermediária vira FLOOR
                ds_grid_set(grafo, (global.node_x + new_x) div 2, (
                    global.node_y + new_y) div 2, "FLOOR");
                // Nova posição vira CONTROLER
                ds_grid_set(grafo, new_x, new_y, "CONTROLER");
                // Adicionar nova posição na pilha
                ds_stack_push(global.stack, [new_x, new_y]);
                // Atualizar posição do controller
                global.node_x = new_x;
                global.node_y = new_y;
                moved = true;
            }
        }
    }
}
```

Código 5: *Looping* principal

```
// Loop principal
while (ds_stack_size(global.stack) > 0) {
    if (walls_around(global.node_x, global.node_y)) {
        // Tenta mover o controller
        move_controller();
    } else {
        // Não há mais paredes ao redor, voltar para a posição anterior
        var previous_position = ds_stack_pop(global.stack);
        if (previous_position != undefined) {
            global.node_x = previous_position[0];
            global.node_y = previous_position[1];
            // Atualizar a posição no grid
            ds_grid_set(grafo, global.node_x, global.node_y, "CONTROLER
                ");
        }
    }
}// Liberar memória da pilha
ds_stack_destroy(global.stack);
for (var i = 0; i < grid_width; i++) {
    for (var j = 0; j < grid_height; j++) {
        if (ds_grid_get(grafo, i, j) != "WALL") {
            ds_grid_set(grafo, i, j, "FLOOR");
        }
    }
}
```

Por fim, para desenhar o labirinto, devemos colocar este código no evento *Draw*.

Código 6: Desenhando o Labirinto

```
for (var i = 0; i < grid_width; i++) {
    for (var j = 0; j < grid_height; j++) {
        var x_pos = i * cell_width + 16;
        var y_pos = j * cell_height + 16;

        // Verifica o valor na grid e cria os objetos correspondentes
        if (ds_grid_get(grafo, i, j) == "WALL") {
            instance_create_depth(x_pos, y_pos, 1, obj_wall     )
        } else if (ds_grid_get(grafo, i, j) == "FLOOR") {
            instance_create_depth(x_pos, y_pos, 1, obj_floor);
        } else if (ds_grid_get(grafo, i, j) == "CONTROLER") {
            instance_create_depth(x_pos, y_pos, 1, obj_controler);
        }
    }
}
```

2.5.3. Simple Room Placement

O algoritmo *Simple Room Placement* se mostrou o de mais fácil implementação, todos os códigos a seguir deverão ser inseridos no evento *Create* do objeto desejado.

Vale ressatar que, dos três algoritmos citados, este é o único que não se baseia em um grafo, mas para seu funcionamento, devemos criar um objeto para os corredores, e um objeto para as salas.

Código 1: Definimos as salas, e colocamos elas nos lugares

```
// Configuração inicial
randomize();
var n_rooms = 50; // Número de salas

// Array para armazenar as salas
var rooms = [];

// Criar as salas
while (array_length(rooms) <= n_rooms) {
    // Gerar posi ção aleatória dentro dos limites do mapa
    var pos_x = irandom_range(64, 1250);
    var pos_y = irandom_range(64, 680);

    // Criar a sala como estrutura
    var new_room = {
        x: pos_x,
        y: pos_y,
    };
    // Verificar se a sala sobrepõe alguma outra já existente
        if (place_free(pos_x, pos_y)){
            // Adicionar sala válida ao array
                array_push(rooms, new_room);
            // Criar a sala no jogo
                instance_create_layer(pos_x, pos_y, "Instances",
                    obj_room);
        }
}
```

Código 2: Conectamos as salas

```
// Conectar salas com corredores
for (var i = 1; i < array_length(rooms); i++) {
    var room_a = rooms[i - 1];
    var room_b = rooms[i];
    // Coordenadas do centro das salas
    var x1 = room_a.x
    var y1 = room_a.y
    var x2 = room_b.x
    var y2 = room_b.y
    // Criar corredor horizontal
    if (x1 != x2) {
        for (var x_pos = min(x1, x2); x_pos <= max(x1, x2); x_pos++) {
            instance_create_layer(x_pos, y1, "Instances", obj_wall);
        }
    }
    // Criar corredor vertical
    if (y1 != y2) {
        for (var y_pos = min(y1, y2); y_pos <= max(y1, y2); y_pos++) {
            instance_create_layer(x2, y_pos, "Instances", obj_wall);
```

```
            }
        }
}
```

3. Metodologia

Nesta seção, descrevemos os métodos utilizados para a geração procedural de mapas e salas, destacando a implementação dos algoritmos empregados e sua comparação com jogos que utilizam técnicas semelhantes.

Inicialmente, foram analisados três algoritmos principais para geração de labirintos e mapas: Simple Room Placement, Backtracking e Kruskal. Cada um desses algoritmos foi implementado e seus resultados foram avaliados em relação à estrutura e organização dos cenários gerados.

Para validar a eficiência e aplicabilidade desses algoritmos, os mapas gerados foram comparados com jogos que fazem uso de técnicas similares. O *Simple Room Placement* foi analisado em relação ao jogo *The Binding of Isaac*, destacando-se as semelhanças na disposição das salas. Já o algoritmo de *Backtracking* demonstrou resultados semelhantes aos labirintos do jogo *Labyrinthine*, onde a geração procedural é utilizada para criar ambientes imersivos e desafiadores. Por fim, o algoritmo de *Kruskal* foi comparado ao jogo *WorldBox - God Simulator*, no qual a geração procedural é empregada na criação de terrenos e biomas.

As implementações foram realizadas utilizando o GameMaker2, que foi descrito na Seção 2.1.1, juntamente com suas estruturas de dados, que se mostraram mais que suficientes para representar os mapas gerados.

4. Resultados Obtidos

Nesta seção, serão apresentados e detalhados os resultados obtidos a partir da execução dos algoritmos mencionados na Seção 2.3. A análise a seguir fornecerá uma visão abrangente do desempenho de cada algoritmo, destacando suas eficiências e características específicas em termos de tempo de execução, utilização de memória e a qualidade dos resultados gerados. Serão consideradas as variáveis relevantes para cada abordagem, com ênfase nas métricas de complexidade computacional e sua aplicabilidade prática, a fim de fornecer uma compreensão clara sobre os pontos fortes e limitações de cada algoritmo no contexto do problema em questão.

Como pode ser observado na Figura 8, as áreas destacadas em ciano ou em tons claros de cinza, representam os caminhos gerados pelo algoritmo. Por outro lado, as áreas em cinza ou em tons escuros de cinza, correspondem às paredes, que delimitam os limites do labirinto e ajudam a formar sua estrutura de maneira clara e organizada. Esse método de geração de labirintos é interessante não apenas por sua simplicidade, mas também pela forma eficiente com que distribui os caminhos, garantindo um resultado funcional e visualmente compreensível.

4.1. Kruskal

O primeiro algoritmo implementado neste trabalho foi o de *Kruskal*, amplamente conhecido por sua eficiência em problemas relacionados à geração de árvores geradoras

mínimas. Esse algoritmo foi utilizado para criar um labirinto, onde os caminhos e as paredes são definidos de maneira aberta, com bastante espaço.

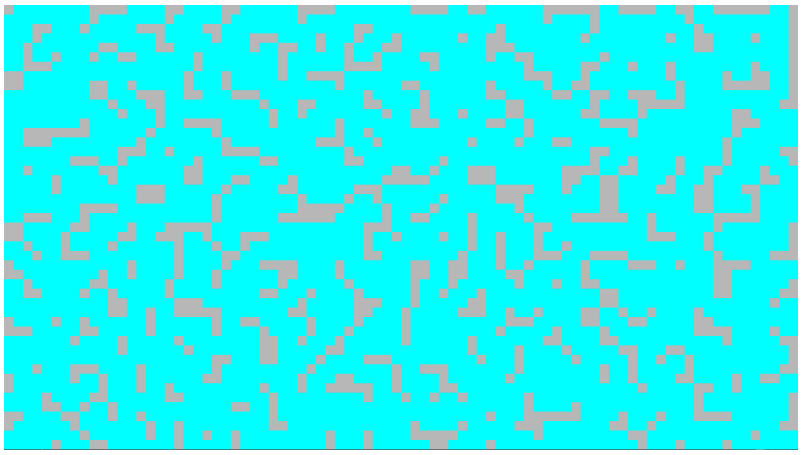

Figura 8. Labirinto gerado com o algoritmo de *Kruskal*

Uma possível aplicação para esse método seria na criação de terrenos 2D em jogos ou simulações, onde a diferenciação entre áreas transitáveis e não transitáveis é essencial. Por exemplo, as áreas em cinza podem ser utilizadas para representar montanhas ou obstáculos naturais, enquanto as áreas em ciano ou em tons claros de cinza poderiam corresponder a caminhos navegáveis ou planícies. Essa abordagem permite criar ambientes dinâmicos e estratégicos, ideais para jogos de aventura ou exploração, ao mesmo tempo que mantém a implementação relativamente simples e flexível para diferentes cenários.

4.2. Backtracking

O algoritmo de *Backtracking* destacou-se como o preferido do autor para implementação, devido à sua simplicidade e aos resultados bem definidos que proporciona. Diferentemente do labirinto gerado pelo algoritmo de Kruskal, apresentado na Seção 4.1, o *Backtracking* oferece uma representação de labirinto que se aproxima mais de sua forma literal, como pode ser visto na Figura 9.

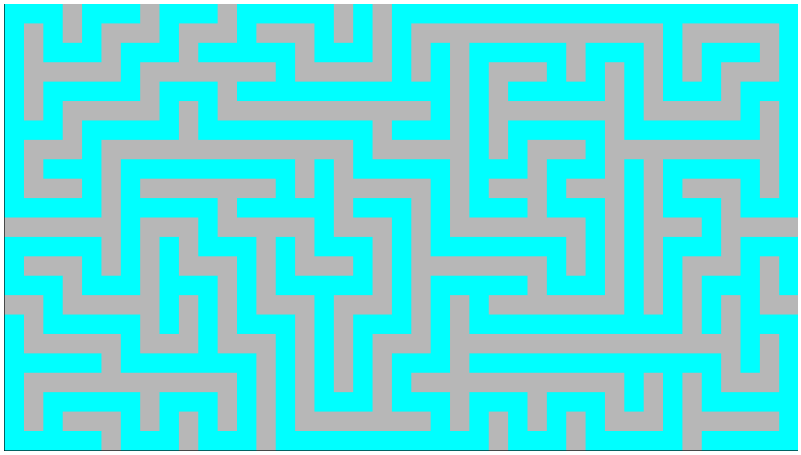

Figura 9. Labirinto gerado com o algoritmo de *Backtraking*

Neste labirinto, seguimos a mesma lógica utilizada no exemplo anterior: as áreas em ciano ou tons claros de cinza representam o piso, enquanto as áreas em cinza ou em tons escuros de cinza correspondem às paredes.

Uma aplicação prática para labirintos gerados com o algoritmo de *Backtracking* seria em jogos eletrônicos, especialmente em desafios de exploração ou solução de quebra-cabeças, como os populares jogos de aventura em 2D. Nesse contexto, o labirinto poderia ser utilizado para criar níveis onde o jogador precisa encontrar uma saída, enfrentando obstáculos representados pelas áreas em cinza e seguindo caminhos disponíveis em ciano ou tonalidades mais claras. Além disso, esse algoritmo poderia ser empregado em simulações de navegação autônoma, onde robôs ou agentes inteligentes necessitam encontrar o caminho ideal em um ambiente delimitado por paredes e caminhos viáveis. Sua simplicidade na construção do labirinto e a garantia de uma solução viável tornam o *Backtracking* uma escolha eficiente para tais aplicações.

4.3. Simple Room Placement

Os labirintos gerados pelo algoritmo *Simple Room Placement* apresentam uma abordagem distinta em relação aos métodos discutidos anteriormente. Como o próprio nome sugere, esse algoritmo trabalha com a geração de salas, distribuindo-as estrategicamente ao longo da área desejada, conforme mencionado na Seção 2.3.2.

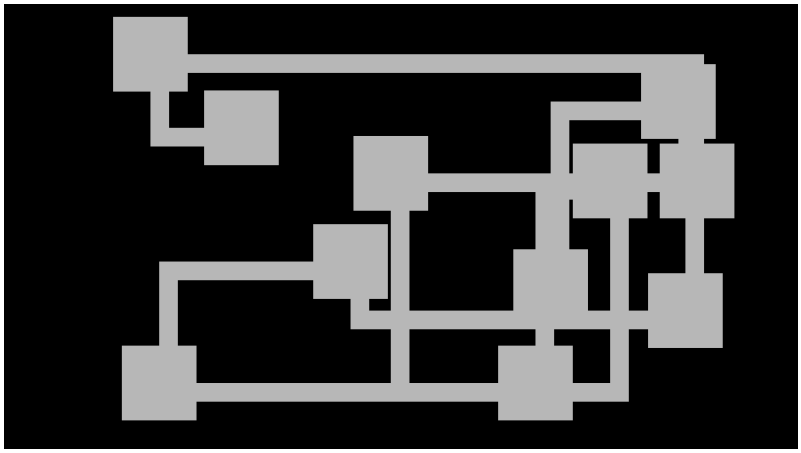

Figura 10. Labirinto gerado com o algoritmo *Simple Room Placement*

A implementação desse algoritmo é bastante satisfatória, especialmente devido à sua capacidade de criar layouts dinâmicos e variados. Suas aplicações práticas são amplamente voltadas para jogos 2D no estilo *Roguelike*, que se caracterizam por múltiplas salas interconectadas e um alto fator de rejogabilidade. A aleatoriedade inerente ao algoritmo permite que cada nova *gameplay* resulte em um mapa diferente, proporcionando uma experiência única a cada partida. Esse método também pode ser aplicado em jogos de exploração e masmorras (*dungeon crawlers*), onde a geração procedural de ambientes enriquece a imersão e a estratégia dos jogadores.

5. Considerações Finais

Com este trabalho, concluímos que os métodos de geração procedural possuem ampla aplicabilidade em diversas áreas, sendo fundamentais para o desenvolvimento de soluções

eficientes na criação de mapas e ambientes dinâmicos. Além disso, destacamos o *GameMaker2* como uma ferramenta versátil para o desenvolvimento de jogos e apresentamos a implementação dos algoritmos citados, proporcionando uma base para sua utilização em projetos futuros.

Referências

Amaral, G. P. d. A. M. d. (2023). *Algoritmo de Grafos Cíclico para Geração de Dungeons*. PhD thesis, Universidade de Lisboa, Faculdade de Ciências.

GameMaker (2025). Gamemaker official website. Acessado em: 17 fev. 2025.

Li, H., Xia, Q., Wang, Y., et al. (2017). Research and improvement of kruskal algorithm. *Journal of Computer and Communications*, 5(12):63.

Rodrigues, R. A. N. and Carvalhaes, M. F. A. (2017). Análise e complexidade de algoritmos: Backtracking e for a bruta. *Revista Ada Lovelace*, 1:45–48.

Togelius, J., Yannakakis, G. N., Stanley, K. O., and Browne, C. (2011). Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186.

Van Beek, P. (2006). Backtracking search algorithms. In *Foundations of artificial intelligence*, volume 2, pages 85–134. Elsevier.