UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
PRÓ-REITORIA DE GRADUAÇÃO
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE CIÊNCIAS DA COMPUTAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO (2025)

SISTEMA DE GERAÇÃO PROCEDURAL DE MAPAS APLICADO A JOGOS

Vinicius D. de Sousa[1]
Helcio W. da Silva[2]

RESUMO

Este artigo apresenta um guia prático para desenvolvedores interessados na implementação de geração procedural de mapas em jogos 2D. São abordados dois algoritmos com finalidades distintas: o *Perlin Noise*, utilizado para gerar terrenos naturais com transições suaves, e o *Binary Space Partitioning* (BSP), voltado à criação de mapas internos com salas e corredores interligados. Além dos fundamentos teóricos, o trabalho descreve de forma detalhada a implementação de cada algoritmo, incluindo parâmetros ajustáveis, lógica de geração e exemplos visuais. O objetivo é fornecer um material educativo e replicável, que auxilie outros desenvolvedores na construção de ambientes de jogo dinâmicos e personalizáveis.

Palavras-chave: geração procedural; *perlin noise*; bsp; jogos.

ABSTRACT

This article presents a practical guide for developers interested in procedural map generation in 2D games. Two algorithms are addressed: Perlin Noise, widely used for generating smooth natural terrains, and Binary Space Partitioning (BSP), ideal for indoor maps with interconnected rooms and corridors. Beyond the theoretical foundations, the article offers a detailed, step-by-step breakdown of each implementation, including adjustable parameters, generation logic, and visual examples. The goal is to provide a replicable and educational reference that supports the development of dynamic, customizable game environments.

Key words: procedural generation; perlin noise; bsp; games

1 INTRODUÇÃO

A geração procedural é uma técnica computacional que consiste na criação de conteúdo por meio de algoritmos capazes de gerar dados automaticamente, a partir de regras definidas, funções matemáticas ou fontes de aleatoriedade controlada. Essa técnica permite que ambientes, objetos ou desafios sejam produzidos de forma dinâmica, sem a necessidade de intervenção manual em cada instância gerada [Leite e Lima 2015].

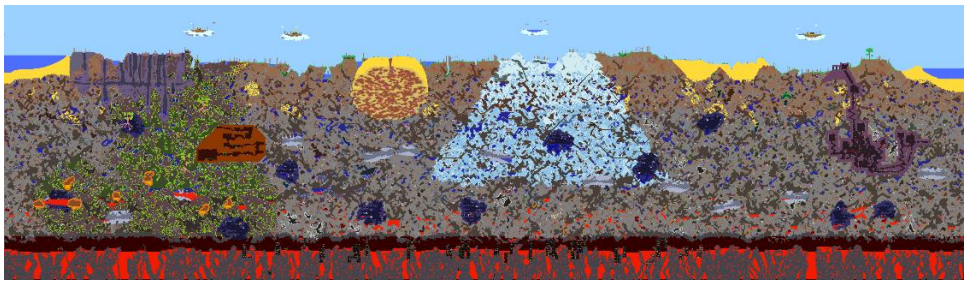
No contexto de desenvolvimento de jogos digitais, a geração procedural tem se consolidado como uma abordagem estratégica para a criação de mapas, terrenos, calabouços,

[1] Graduando do Bacharelado em Ciências da Computação UFERSA
[2] Professor Associado do Departamento de Computação da UFERSA/Mossoró

personagens, itens, narrativas e entre outros elementos. Sua principal característica é a capacidade de produzir conteúdo variado e, consequentemente, aumentar a longevidade de um jogo no mercado, além de acelerar o processo de desenvolvimento [Canedo 2022].

Este artigo tem como objetivo analisar e demonstrar a aplicação de dois algoritmos distintos de geração procedural de mapas em jogos 2D: o *Perlin Noise*, amplamente utilizado para criar terrenos naturais com transições suaves, como florestas, montanhas e oceanos, sendo empregado em jogos como *Terraria* [Re-Logic 2011], representado pela Figura 1, e o *Binary Space Partitioning* (BSP), voltado à geração de mapas internos compostos por salas e corredores interconectados, como observado no jogo *Nethack* [Stephenson 1985], representado pela Figura 2.

Figura 1 - Exemplo de mapa gerado pelo jogo *Terraria* utilizando Perlin Noise



Fonte: https://forums.terraria.org/index.php?threads/flyingsnake-a-world-map-renderer.82400/

Figura 2. Exemplo de mapa gerado pelo jogo *Nethack* utilizando BSP

Fonte: https://dosgames.com/game/nethack/

Como evidenciado pelas Figura 1 e 2, ambos algoritmos possuem características que os tornam mais adequados para diferentes estilos e propósitos de design de mapas.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção, serão apresentados alguns conceitos básicos relacionados ao desenvolvimento de jogos digitais e à geração procedural de mapas, com o objetivo de contextualizar o leitor que não possui familiaridade com o tema.

2.1 Jogos Digitais e Ambientes Gerados Proceduralmente

Jogos digitais podem ser estruturados de diversas formas, desde cenários lineares e pré-definidos até ambientes abertos e gerados dinamicamente. Uma tendência crescente no design de jogos é a geração procedural, que consiste na criação de conteúdo de forma automática por

meio de algoritmos. Essa abordagem permite criar ambientes únicos a cada nova jogada, economizando recursos de desenvolvimento e aumentando a rejogabilidade.

No contexto dos jogos, é comum classificarmos os mapas como abertos ou fechados. Um mapa aberto oferece ao jogador liberdade de locomoção e exploração em grandes áreas contínuas, como florestas, planícies ou montanhas. Já um mapa fechado é composto por espaços internos segmentados, como salas, corredores ou andares de um calabouço, geralmente com rotas mais lineares ou interconectadas.

Algoritmos diferentes são mais adequados para cada um desses tipos. O *Perlin Noise* é amplamente utilizado na geração de mapas abertos, por produzir terrenos naturais e contínuos, ideais para jogos com exploração em áreas externas. Já o algoritmo *Binary Space Partitioning* (BSP) se destaca na criação de mapas fechados, como calabouços, edifícios ou labirintos, onde o ambiente é composto por múltiplos compartimentos conectados de forma organizada.

2.2 O que é "ruído" em geração procedural

No contexto de algoritmos como o *Perlin Noise*, o termo "ruído" não está associado a algo indesejado ou caótico. Pelo contrário, refere-se a padrões matemáticos contínuos que simulam irregularidades naturais, como variações de relevo, nuvens ou texturas orgânicas.

O ruído gerado é controlado, ou seja, segue um padrão que parece natural, mas ainda assim imprevisível. Ele serve como um conjunto de valores que pode representar, por exemplo, diferentes altitudes em um terreno ou variações no tipo de solo. O objetivo é criar mapas abertos com topografia fluida e natural, ideais para jogos com exploração em grandes áreas externas.

3 PERLIN NOISE

O Perlin Noise é um algoritmo de geração de ruído desenvolvido por Ken Perlin, inicialmente com o objetivo de criar padrões de textura mais realistas em computação gráfica. Desde então, tornou-se uma ferramenta fundamental na geração procedural de conteúdo para jogos, especialmente na criação de terrenos naturais, nuvens, água e outros elementos com transições suaves [Biagioli 2014].

Ao contrário de outros algoritmos de ruído, o Perlin Noise proporciona uma variação contínua e suave de valores, criando padrões orgânicos que imitam a natureza [Scratchapixel 2025].

Essa suavidade presente no algoritmo é alcançada ao seguir as etapas:

- Divisão do espaço em células;
- Atribuição de vetores gradientes aos vértices das células;
- Atribuição de vetores distância: Dos vértices ao ponto;
- Cálculo do produto escalar entre os vetores gradientes e distância;
- Interpolação dos valores do produto escalar;
- Normalização do valor final;
- Oitavas e Ruído Composto.

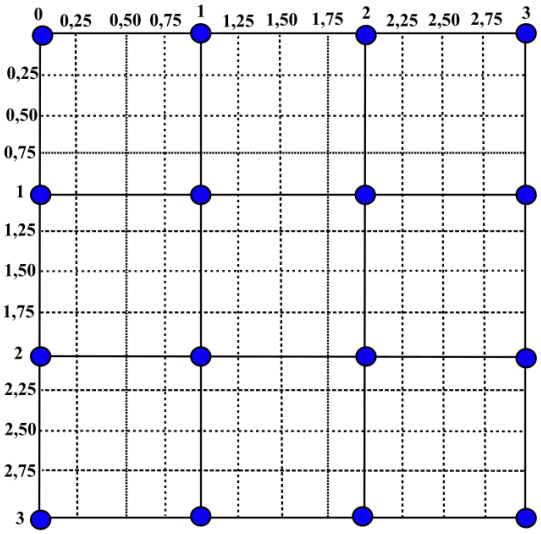
A seguir, detalharemos cada uma dessas etapas conceituais.

3.1 Divisão do Espaço em Células

O algoritmo Perlin Noise é baseado na ideia de calcular um valor de ruído em qualquer ponto decimal dentro de uma grade, de forma contínua e suave [Scratchapixel 2025]. Para isso, o primeiro passo é dividir o espaço da grade em células, que podem ser visualizadas como

quadrados de uma malha bidimensional, semelhante a um papel quadriculado, conforme a Figura 3.

Figura 3 - Grade bidimensional composta por células

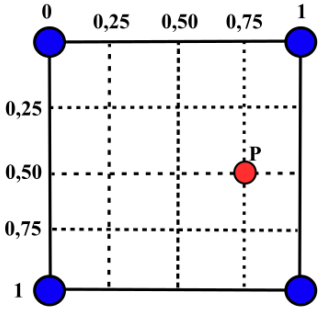


Fonte: Autoria própria (2025).

Cada célula é composta por quatro vértices (cantos), e o objetivo do algoritmo é determinar o valor de ruído para qualquer ponto que esteja dentro dessa célula. Esse valor será influenciado pelos vértices ao redor.

Imagine que temos um ponto de coordenadas decimais P(0.75,0.50). Esse ponto está localizado dentro de uma célula específica, cujos quatro cantos mais próximos ((0,0), (1,0), (0,1) e (1,1)) serão usados para determinar o valor final de ruído para o ponto, como ilustrado pela Figura 4. Essa célula será usada como base para o cálculo do Perlin Noise nas próximas etapas.

Figura 4 - Representação de uma célula com seus quatro vértices e um ponto decimal P em seu interior.



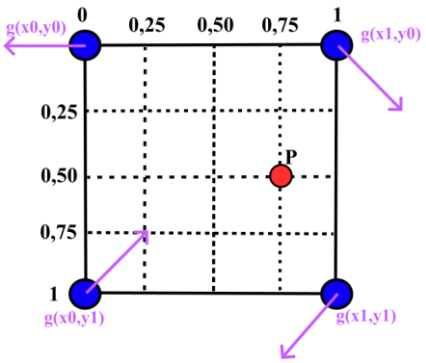
Fonte: Autoria própria (2025).

Essa divisão em células permite que o Perlin Noise opere localmente em pequenas regiões do espaço, garantindo suavidade nas transições entre os valores de ruído.

3.2 Atribuição de Vetores Gradientes aos Vértices das Células

Após a divisão do espaço em células, o próximo passo do algoritmo é associar um vetor gradiente a cada vértice (ou canto) das células, conforme a Figura 5. Esses vetores representam direções e serão responsáveis por influenciar o valor de ruído calculado para o ponto P dentro da célula.

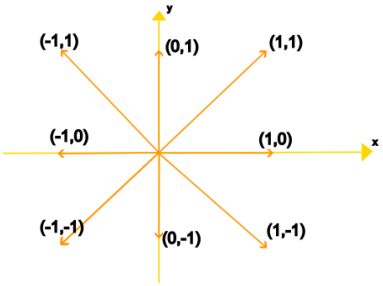
Figura 5 - Exemplo de vetores gradientes nos vértices de uma célula



Fonte: Autoria própria (2025).

Esses vetores gradientes não são gerados de forma completamente aleatória. No caso do Perlin Noise 2D, existem oito possíveis direções para os vetores gradientes. Essas direções correspondem a vetores unitários que apontam ao longo dos eixos cartesianos e das diagonais principais, como ilustrado pela Figura 6.

Figura 6 - Representação das 8 possíveis direções de vetores gradientes



Fonte: Autoria própria (2025).

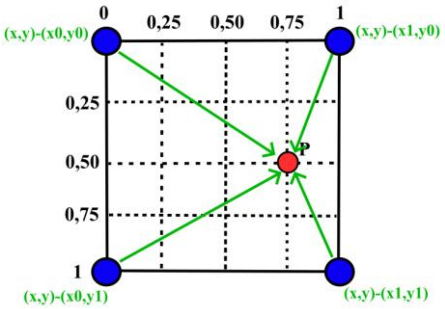
A escolha desses vetores influencia diretamente o comportamento do ruído. Como cada ponto dentro da célula será influenciado pelos vértices ao redor, a direção e o sentido desses vetores determinam a variação local do valor de ruído. Essa abordagem é essencial para gerar padrões com aparência natural e transições suaves.

3.3 Atribuição de Vetores Distância: Dos Vértices ao Ponto

Com os vetores gradientes já definidos nos vértices da célula, o próximo passo é calcular os vetores distância. Esses vetores são obtidos a partir da diferença entre cada vértice da célula e o ponto P(0.75,0.50) onde queremos calcular o valor de ruído.

Em termos simples, para cada um dos quatro cantos da célula, traçamos um vetor que aponta do vértice até o ponto P, como visto na Figura 7. Esses vetores indicam a "posição relativa" de P em relação aos vértices e serão usados para medir a influência que cada canto exerce sobre o valor de ruído final.

Figura 7 - Representação dos vetores distância entre P e os vértices da célula



Fonte: Autoria própria (2025).

Para o ponto P(0.75, 0.50), localizamos sua posição em relação a cada um dos vértices da célula:

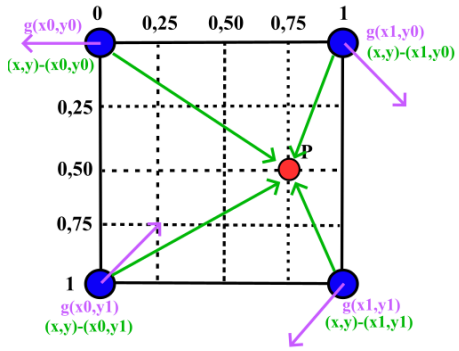
- Do vértice (0,0) até P: vetor distância (0.75, 0.50)
- Do vértice (1,0) até P: vetor distância (-0.25, 0.50)
- Do vértice (0,1) até P: vetor distância (0.75, -0.5)
- Do vértice (1,1) até P: vetor distância (-0.25, -0.5)

Esses vetores distância são fundamentais para a próxima etapa do algoritmo, pois serão combinados com os vetores gradientes por meio de produtos escalares, que ajudam a determinar a intensidade da influência de cada vértice no ponto considerado. Quanto mais alinhados estiverem os vetores distância e gradiente, maior será a influência.

3.4 Cálculo do Produto Escalar entre os Vetores Gradientes e Distância

Após definirmos os vetores gradientes nos vértices e os vetores distância entre esses vértices e o ponto P(0.75, 0.50), o próximo passo é combinar essas informações para medir o grau de influência de cada vértice sobre o ponto. Isso é feito através do produto escalar entre o vetor gradiente e o vetor distância, representado pela Figura 8.

Figura 8 - Célula com seus vetores distância e gradientes associados, indicando o cálculo do produto escalar para P



Fonte: Autoria própria (2025).

O produto escalar entre dois vetores é uma operação matemática que indica o quanto um vetor aponta na direção do outro. Em termos do algoritmo, quanto mais alinhado o vetor distância estiver com o vetor gradiente, maior será o valor do produto escalar, e, portanto, maior a influência daquele vértice sobre o ponto P.

A Equação 1 mostra o cálculo do produto escalar para vetores bidimensionais:

$$\mathbf{g} \cdot \mathbf{d} = g_x \cdot d_x + g_y \cdot d_y \quad (1)$$

Onde:

- $g = (g_x, g_y)$ é o vetor gradiente
- $d = (d_x, d_y)$ é o vetor distância

O resultado dessa operação pode ser interpretado da seguinte forma:

- **Valor positivo:** os vetores estão alinhados, a influência é forte.
- **Valor zero:** os vetores são ortogonais, não há influência.
- **Valor negativo:** os vetores estão em direções opostas, a influência é fraca.

Com base nesse conceito, vamos calcular os produtos escalares dos vetores:

- **Vértice (0,0)**: $(-1, 0) \cdot (-0{,}75, -0{,}5) = -0.75$
- **Vértice (1,0)**: $(1, -1) \cdot (0{,}25, -0{,}5) = -0.75$
- **Vértice (0,1)**: $(1, 1) \cdot (-0{,}75, 0{,}5) = 0.25$
- **Vértice (1,1)**: $(-1, -1) \cdot (0{,}25, 0{,}5) = 0.75$

O produto escalar resultante de cada vértice será utilizado na interpolação final, permitindo que o valor de ruído no ponto seja uma combinação suave e ponderada das influências dos quatro cantos da célula.

3.5 Interpolação dos Valores do Produto Escalar

Depois de calcular o produto escalar entre os vetores gradientes e distância para os quatro vértices da célula, ainda não temos o valor final de ruído para o ponto P(0.75, 0.50). Esses quatro valores precisam ser combinados suavemente para garantir transições naturais entre células vizinhas. É aqui que entra o processo de interpolação.

A interpolação é uma técnica que permite calcular um valor intermediário entre dois pontos, com base em um fator *t* que varia entre 0 e 1. A Equação 2 representa o cálculo da interpolação linear:

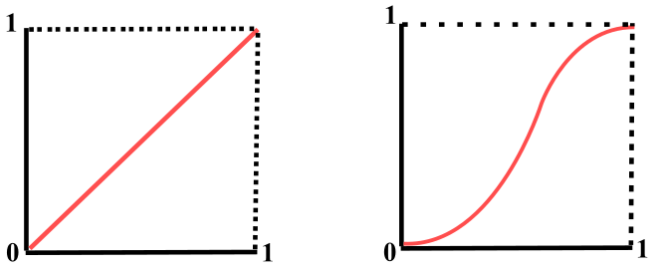
$$\mathbf{lerp(a, b, t) = a + t(b - a)\ (2)}$$

No entanto, essa forma linear simples pode causar transições abruptas, o que prejudicaria a suavidade desejada no Perlin Noise. Para evitar isso, Ken Perlin propôs o uso de uma função de suavização, chamada de *fade function*, que altera o fator *t* antes de aplicá-lo na interpolação [Biagioli 2014]. A função de suavização, vista pela Equação 3, é:

$$\mathbf{f(t) = 6t^5 - 15t^4 + 10t^3\ (3)}$$

Essa função suaviza as extremidades da interpolação, começando e terminando com variações suaves, o que elimina descontinuidades visuais. Podemos observar a diferença entre a função de interpolação linear e a função de interpolação suavizada na Figura 9.

Figura 9 - Comparação entre os gráficos de interpolação linear e interpolação suavizada.



Fonte: Autoria própria (2025).

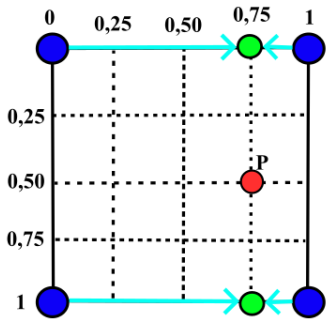
Aplicando a função de suavização para a função de interpolação linear, temos o cálculo representado pela Equação 4:

$$\mathbf{lerp(a, b, f(t)) = a + f(t)(b - a)\ (4)}$$

Com essa função, o processo de interpolação no Perlin Noise 2D é feito em duas etapas, conhecido como interpolação bilinear:

- **Interpolação no eixo X:** Primeiramente, interpolamos os dois produtos escalares dos vértices superiores entre si e, separadamente, os dos vértices inferiores, conforme ilustrado pela Figura 10. Para o eixo X o valor de ***t*** que será suavizado é **0.75**, que vem de P(**0.75**, 0.50).
  - **Vértices superiores (0,0) e (1,0)**: lerp(0.75, 0.75, f(*t*)) = -0.75
  - **Vértices inferiores (0,1) e (1,1)**: lerp(-0.25, -0.75, f(*t*)) = 0.698

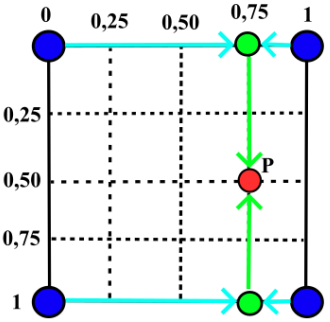
Figura 10 - Interpolação ao longo do eixo X entre os produtos escalares dos vértices superiores e inferiores.



Fonte: Autoria própria (2025).

- **Interpolação no eixo Y:** Depois, interpolamos os dois resultados obtidos anteriormente para calcular o valor final de ruído no ponto P(0.75, 0.50), de acordo com a Figura 11. Para o eixo Y o valor de ***t*** que será suavizado é **0.50**, que vem de P(0.75, **0.50**).
  - **Interpolação final**: lerp(0.75, -0.7, f(*t*)) = -0.025

Figura 11 - Interpolação ao longo do eixo Y entre os resultados obtidos na etapa anterior



Fonte: Autoria própria (2025).

Essa combinação suavizada das influências dos quatro vértices é o que dá ao Perlin Noise sua característica mais marcante: a aparência contínua e orgânica do ruído gerado.

3.6 Normalização do Valor Final

Após todas as etapas anteriores, o algoritmo chega a um valor final de ruído bruto para o ponto P(0.75, 0.50), que é aproximadamente: **-0.025**.

No entanto, esse valor pode estar fora do intervalo [0, 1], o que dificulta sua aplicação direta em representações visuais, como altura de terreno ou intensidade de cor.

Para resolver isso, é feita uma normalização do valor, ajustando-o para o intervalo desejado. Isso é feito a partir do cálculo representado pela Equação 5:

$$\textbf{valor final} = \frac{\textbf{valor interpolado} + 1}{2} \textbf{ (5)}$$

Esse ajuste garante que:

- Valores originalmente em torno de −1 sejam mapeados para próximo de 0
- Valores próximos de 0 fiquem em torno de 0.5
- Valores perto de 1 sejam preservados próximos de 1.

Sendo assim, normalizando o valor de ruído bruto obtido, temos:

$$\textbf{valor final} = \frac{-0.025 + 1}{2} = \textbf{0.487}$$

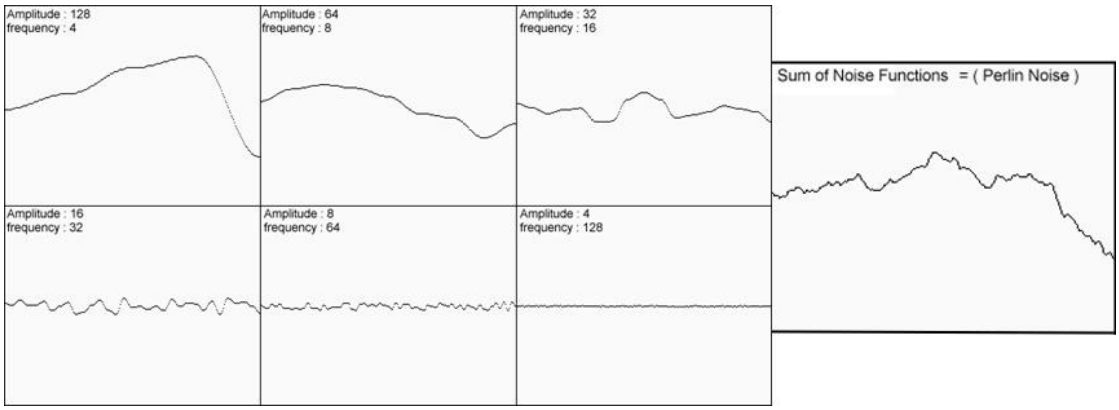
Essa etapa final é essencial para que o ruído gerado possa ser usado de maneira intuitiva na criação de mapas, onde cada ponto do espaço representa um tipo de terreno, com transições suaves e naturais.

3.7 Oitavas e Ruído Composto

Embora o Perlin Noise básico, que acaba no processo anterior, já produza um padrão suave e natural, em muitos casos ele ainda pode parecer simples ou repetitivo. Para enriquecer os detalhes do ruído gerado e criar variações em múltiplas escalas, é comum utilizar uma técnica chamada de oitavas (*octaves*) [Biagioli 2014].

Essa técnica consiste em somar várias camadas de Perlin Noise, cada uma com frequência e amplitude diferentes, conforme a Figura 12.

Figura 12 - Exemplo de composição com oitavas variando frequência e amplitude, para um ruído unidimensional.



Fonte: https://adrianb.io/2014/08/09/perlinnoise.html

Para combinar múltiplas oitavas e formar o ruído final, cada camada recebe valores distintos de frequência e amplitude. A **frequência** de uma camada define quantas variações (oscilações) ocorrem em um determinado espaço. A cada nova oitava adicionada, a frequência é geralmente multiplicada por 2, o que significa que os detalhes ficam mais apertados, aumentando a complexidade do ruído.

Já a **amplitude** determina a variação vertical da camada, ou seja, a diferença entre o valor mínimo e o máximo (valor pico-a-pico) que aquela oitava pode produzir. Quanto maior a amplitude, mais influência aquela camada exerce sobre o relevo. Ao longo das oitavas, essa amplitude vai diminuindo para evitar que os detalhes mais finos distorçam excessivamente a forma geral.

Esse processo é governado por um parâmetro chamado **persistência**, que controla a taxa de decaimento da amplitude nas oitavas subsequentes. A cada nova oitava *i*, os valores são ajustados da seguinte forma [Peter 2025]:

- frequência = 2^i
- amplitude = persistência^i

Com isso, a primeira camada ($i = 0$) tem frequência 1 e amplitude 1, enquanto as demais são ajustadas conforme esses fatores. A persistência, portanto, regula o equilíbrio entre grandes estruturas (baixas frequências e amplitudes altas) e detalhes finos (altas frequências e amplitudes pequenas).

Por fim, o ruído final é construído pela soma ponderada dessas camadas. Como resultado, obtém-se um relevo natural, com grandes formações suavemente enriquecidas por irregularidades menores. Após a soma, é comum normalizar os valores novamente entre 0 e 1 para facilitar seu uso.

4 BINARY SPACE PARTITIONING - BSP

O *Binary Space Partitioning*, ou simplesmente BSP, é um algoritmo originalmente desenvolvido na área de computação gráfica para organização hierárquica de cenas tridimensionais, com o objetivo de otimizar a renderização [Rizzon 2025]. No entanto, sua aplicação foi adaptada com sucesso para o desenvolvimento de jogos digitais, especialmente em sistemas de geração procedural de mapas fechados, como calabouços e labirintos.

A principal ideia por trás do BSP é dividir recursivamente uma área retangular em regiões menores, formando uma estrutura de árvore binária [Shields 2023], onde:

- Cada **nó** da árvore representa uma divisão do espaço.
- Cada **folha** representa uma região final onde será criada uma sala.

Essa estrutura facilita a criação de mapas com salas distribuídas de forma não uniforme, conectadas por corredores, resultando em *layouts* complexos, mas sempre navegáveis e logicamente coerentes.

As etapas principais do BSP são [Roguebasin 2013]:

- Divisão recursiva do espaço;
- Criação das salas;
- Conexão das salas por corredores.

A seguir, detalharemos cada uma dessas etapas.

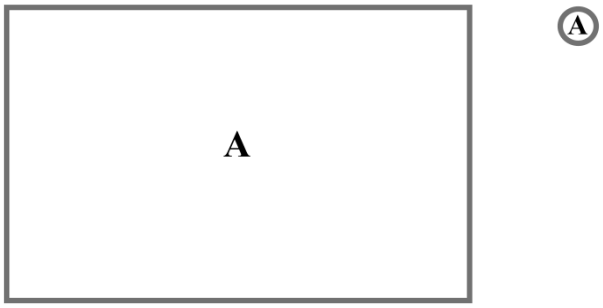
4.1 Divisão Recursiva do Espaço

O processo de geração começa com um retângulo inicial, que representa toda a área disponível do mapa. Esse retângulo é então dividido em duas partes menores, e cada uma dessas partes pode ser novamente subdividida, formando uma estrutura hierárquica em forma de árvore binária.

A divisão recursiva do espaço pode ser feita vertical ou horizontalmente, a escolha sendo feito seguindo critérios como proporção da área atual, tamanho mínimo das regiões e fator aleatório controlado.

Tenha como exemplo um retângulo inicial que representa toda a área disponível do mapa, ilustrado pela Figura 13. Este retângulo é representado pelo nó “A” na figura.

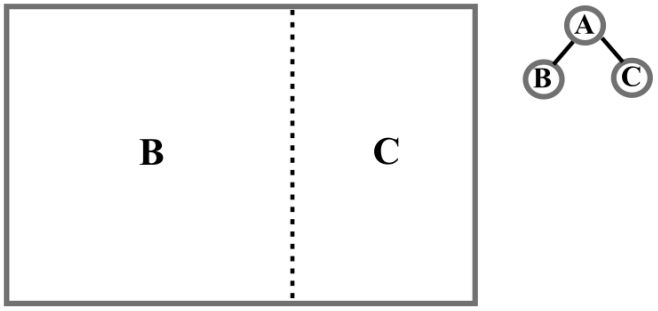
Figura 13 - Área total do mapa antes de qualquer divisão.



Fonte: Autoria própria (2025).

O retângulo é dividido verticalmente em duas partes menores e representam os nós “B” e “C”, ilustrados pela Figura 14. Estes nós são filhos do nó “A”, formando a estrutura de árvore cuja raiz é o nó “A”.

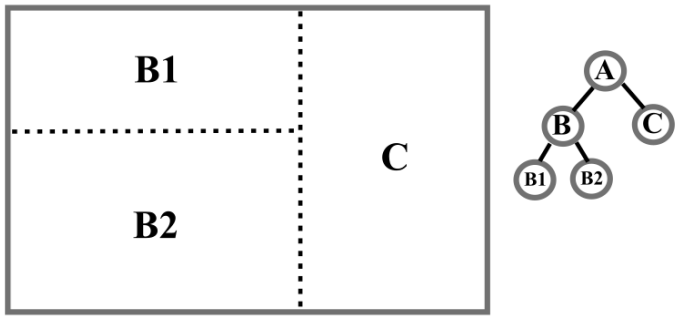
Figura 14 - Divisão vertical realizada sobre a área inicial.



Fonte: Autoria própria (2025).

Após isso, a área representada pelo nó “B” é dividida horizontalmente em duas partes menores, representadas pelos nós “B1” e “B2”, filhos do nó “B”, conforme ilustra a Figura 15.

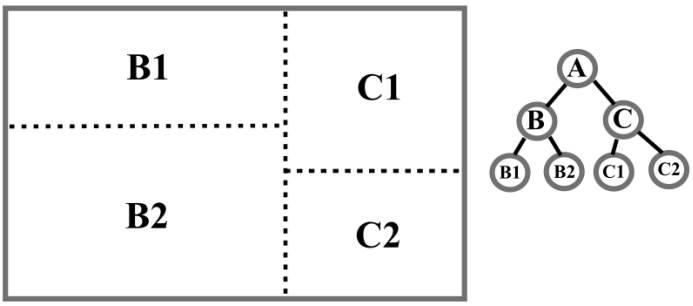
Figura 15 - Divisão horizontal realizada sobre a área do nó “B”.



Fonte: Autoria própria (2025).

Em seguida, a área representada pelo nó “C” é dividida horizontalmente em duas partes menores, representadas pelos nós “C1” e “C2”, filhos do nó “C”, conforme é ilustrado na Figura 16.

Figura 16 - Divisão horizontal realizada sobre a área do nó “C”.



Fonte: Autoria própria (2025).

No final dessa etapa, temos uma segmentação lógica do mapa em retângulos menores, prontos para receber as salas e serem conectados por corredores. A estrutura da árvore BSP facilita a organização espacial e a navegação eficiente do ambiente gerado.

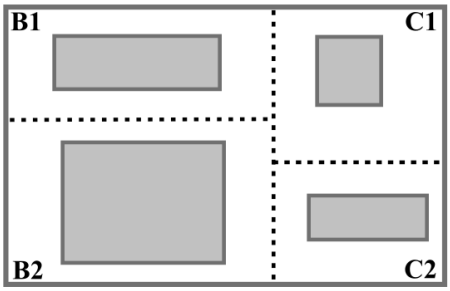
4.2 Criação das Salas

Após a divisão recursiva do espaço, o próximo passo é a criação das salas dentro das folhas da árvore, ou seja, nas regiões finais que não foram mais subdivididas.

Cada sala é posicionada dentro da sua região, mas com uma margem interna proposital, conforme a Figura 17. Essa margem evita que a sala encoste nas bordas da área, o que garante maior flexibilidade na criação dos corredores e evita problemas de sobreposição ou movimentação no jogo.

As dimensões das salas são escolhidas de forma aleatória controlada, respeitando limites mínimos e máximos definidos para largura e altura. Esse controle mantém a diversidade das salas sem comprometer a navegabilidade.

Figura 17 - Salas geradas dentro das folhas da árvore BSP, respeitando margens internas.



Fonte: Autoria própria (2025).

Ao final desta etapa, o mapa possui diversas salas distribuídas em regiões distintas, mas que ainda não estão conectadas entre si. Para isso, é necessário realizar a próxima etapa: a criação dos corredores.

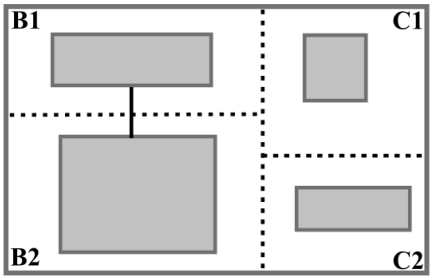
4.3 Conexão das Salas com Corredores

Com as salas devidamente criadas nas folhas da árvore BSP, o próximo passo é garantir que todas estejam conectadas entre si, formando um mapa completamente navegável. Para isso, utiliza-se a própria estrutura da árvore BSP como guia para a construção dos corredores.

O processo ocorre de forma recursiva, de baixo para cima, e garante que todas as salas estejam diretas ou indiretamente interligadas. O algoritmo conecta salas localizadas em subárvores diferentes utilizando o menor número possível de segmentos de linha, variando entre trechos retilíneos e trajetos em “L”.

1. Para cada nó interno da árvore BSP, o algoritmo percorre as subárvores esquerda e direita até encontrar uma sala em cada uma delas. A função de busca retorna a primeira sala encontrada em cada lado, priorizando, por padrão, as folhas mais à esquerda da árvore.
2. Se as salas estiverem alinhadas horizontal ou verticalmente, é criado um corredor reto. Caso contrário, é construído um corredor em "L", formado por dois segmentos. A Figura 18 mostra o primeiro corredor sendo criado entre as salas dos nós irmãos "B1" e "B2".

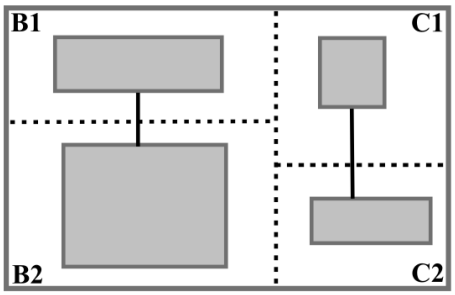
Figura 18 - Primeiro corredor gerado entre as salas irmãs de "B1" e "B2".



Fonte: Autoria própria (2025).

3. O processo continua recursivamente, conectando os pares de salas em cada divisão da árvore. A Figura 19 mostra a adição de um novo corredor entre as salas dos nós irmãos "C1" e "C2".

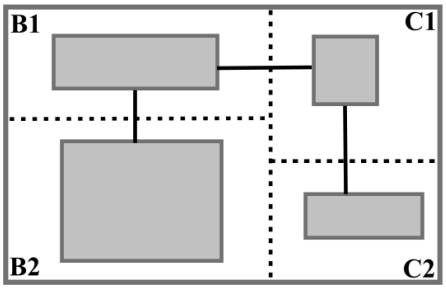
Figura 19 - Segundo corredor gerado entre as salas irmãs de "C1" e "C2".



Fonte: Autoria própria (2025).

4. Ao final, é preciso conectar uma sala da área representada por "B" e uma sala da área representada por "C". Para isso, como mencionado anteriormente na etapa 1, o algoritmo prioriza a primeira sala mais à esquerda de cada área. Sendo assim, é criado um corredor para conectar as salas presentes em "B1" e "C1", conforme ilustra a Figura 20.

Figura 20 - Corredor gerado entre as salas de "B1" e "C1" para que todo o mapa esteja navegável.



Fonte: Autoria própria (2025).

Ao final desta etapa, temos a formação do mapa com suas salas, geradas em diferentes regiões, conectadas por corredores de forma coesa e navegável.

5 O SISTEMA DE GERAÇÃO PROCEDURAL DE MAPAS

Nesta seção, é apresentado um protótipo funcional desenvolvido com o objetivo de demonstrar, na prática, o funcionamento dos algoritmos de geração procedural abordados ao longo do artigo. Para isso, utilizou-se o *GameMaker Studio 2*, um motor de jogos amplamente utilizado no desenvolvimento de jogos 2D tanto por desenvolvedores independentes quanto por grandes estúdios e instituições de ensino ao redor do mundo.

A versão do *GameMaker* utilizada neste protótipo foi a licença gratuita para uso não comercial, que permite exportar projetos para plataformas como Windows, Linux, Android, iOS e navegadores Web. Para projetos comerciais, são oferecidas duas licenças pagas, a Professional e a Enterprise, com a último sendo compatível para exportar até para consoles como Xbox, PlayStation e Nintendo Switch [YoYo Games, 2025].

Semelhante a outros motores, o *GameMaker* estrutura o desenvolvimento de um jogo em *Rooms* compostas por objetos, os quais são configurados com *events* e *scripts* escritos em GML (*GameMaker Language*), a linguagem proprietária do motor.

Para facilitar a experimentação dos algoritmos, foi desenvolvida uma interface gráfica que permite ao usuário alternar entre Perlin Noise e BSP, ajustar os parâmetros de cada algoritmo por meio de *sliders* e gerar novos mapas instantaneamente com um botão. Esta interface pode ser vista através da Figura 21. A abordagem interativa favorece a análise visual dos efeitos de cada parâmetro e torna o sistema mais acessível.

Figura 21 - Interface gráfica para o protótipo descrito nesta seção.

Fonte: Autoria própria (2025).

As subseções a seguir apresentam os parâmetros disponíveis, exemplos visuais de geração de mapas e diagramas que ilustram a estrutura das funções implementadas. O código-fonte completo da aplicação, incluindo a lógica dos algoritmos e da interface interativa, está disponível publicamente no repositório do projeto [Sousa 2025].

5.1 Perlin Noise

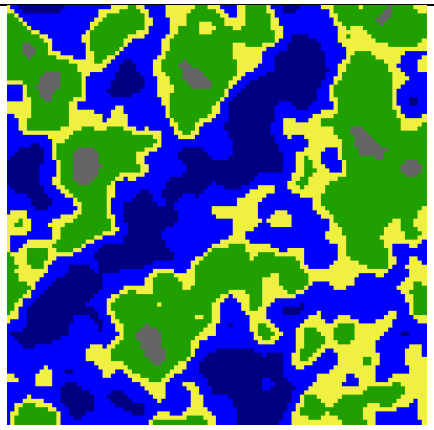
A interface apresentada na Figura 21 permite ao usuário ajustar visualmente os parâmetros essenciais que controlam o comportamento do algoritmo Perlin Noise antes de iniciar a geração do mapa. Esses parâmetros influenciam diretamente a suavidade, o nível de detalhe e a variação do terreno, permitindo explorar diferentes configurações com facilidade.

Os parâmetros disponíveis para ajuste são:

- ***Scale*:** define o espaçamento entre os pontos da grade utilizados para calcular o ruído. Esse espaçamento influencia diretamente a quantidade de variações no mapa: valores menores de *scale* fazem com que os pontos fiquem mais próximos, resultando em transições suaves e grandes áreas contínuas. Já valores maiores aumentam a distância entre os pontos, criando mudanças mais bruscas e um terreno mais fragmentado. Uma forma de visualizar isso seria como olhar para um papel quadriculado com uma lupa: quanto mais você aproxima (menor *scale*), mais detalhes você vê entre os mesmos limites. Mas o papel continua o mesmo, você só está escolhendo com que densidade quer olhar para ele.
- ***Octaves:*** define quantas camadas de ruído (oitavas) serão combinadas para formar o terreno final. Cada nova camada adiciona mais detalhes ao mapa, com frequência maior e amplitude menor.
- ***Persistence:*** indica o quanto da influência de cada oitava será mantida, controlando a redução da amplitude de cada nova oitava. Caso a *persistence* seja 1, todas as oitavas terão igual influência. Se a *persistence* for um valor superior a 1, então todas as oitavas sucessivas serão mais influentes, o que pode resultar em algo próximo a um ruído comum (*white noise*). Já se a *persistence* for um valor menor que 1, o que é o padrão, as oitavas sucessivas menos influentes [Scher 2017].

Ao pressionar o botão "*Generate*", o sistema aplica os valores definidos pelos *sliders* e instancia dinamicamente o algoritmo com as configurações escolhidas, permitindo observar em tempo real os efeitos de cada parâmetro. A Figura 22 apresenta um mapa gerado com os valores padrão para cada parâmetro ao iniciar o protótipo.

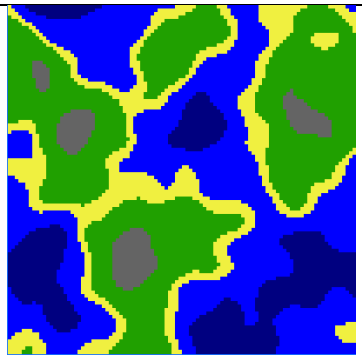
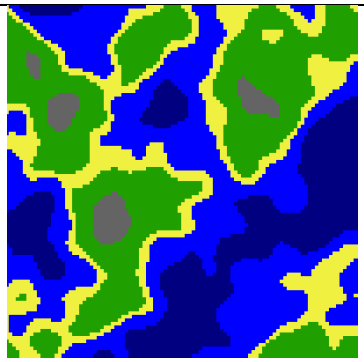
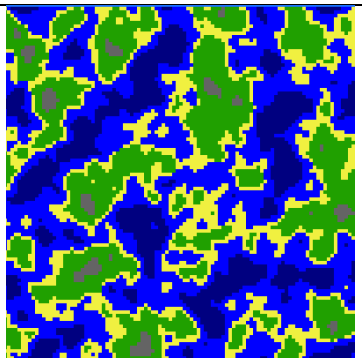
Figura 22 - Exemplo de um mapa gerado com os valores padrão para os parâmetros ao iniciar o protótipo.

Scale = 0.04	Octaves = 3	Persistence = 0.6
		

Fonte: Autoria própria (2025).

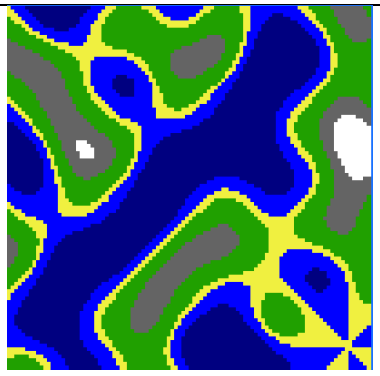
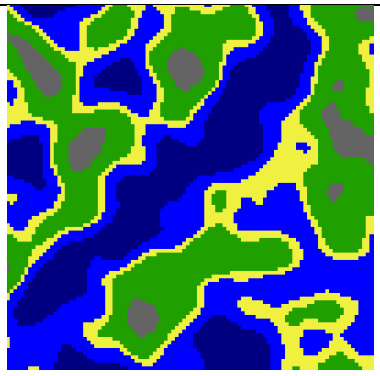
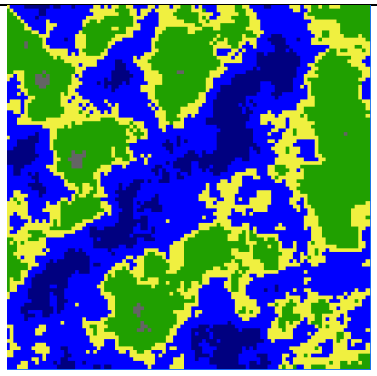
A seguir, é apresentado as Figuras 22A, 22B e 22C, onde é demonstrado o mesmo mapa da Figura 22 com diferentes valores, respectivamente, para *Scale*, *Octaves* e *Persistence*. Considere que apenas um parâmetro está sendo alterado por figura, mantendo os demais com o valor padrão.

Figura 22A - Mapa representado pela Figura 22 com diferentes valores de *Scale*, mantendo os demais parâmetros como padrão.

Scale = 0.02	Scale = 0.03	Scale = 0.06
		

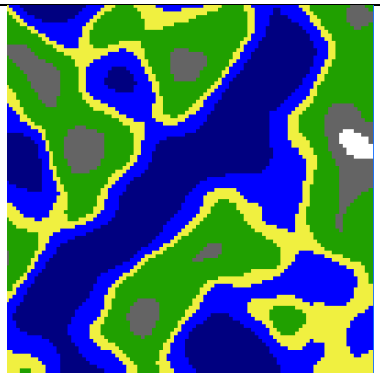
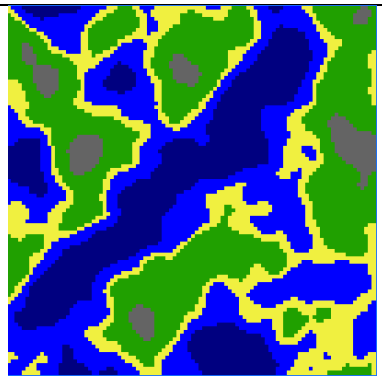
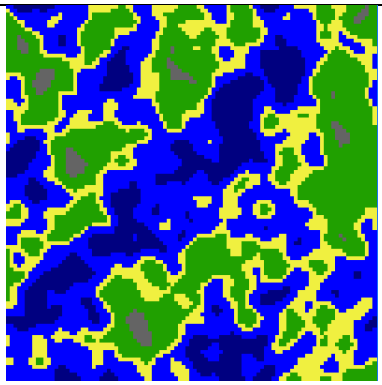
Fonte: Autoria própria (2025).

Figura 22B - Mapa representado pela Figura 22 com diferentes valores de *Octaves*, mantendo os demais parâmetros como padrão.

Octaves = 1	Octaves = 2	Octaves = 7
		

Fonte: Autoria própria (2025).

Figura 22C - Mapa representado pela Figura 22 com diferentes valores de *Persistence*, mantendo os demais parâmetros como padrão.

Persistence = 0.2	Persistence = 0.4	Persistence = 0.8
		

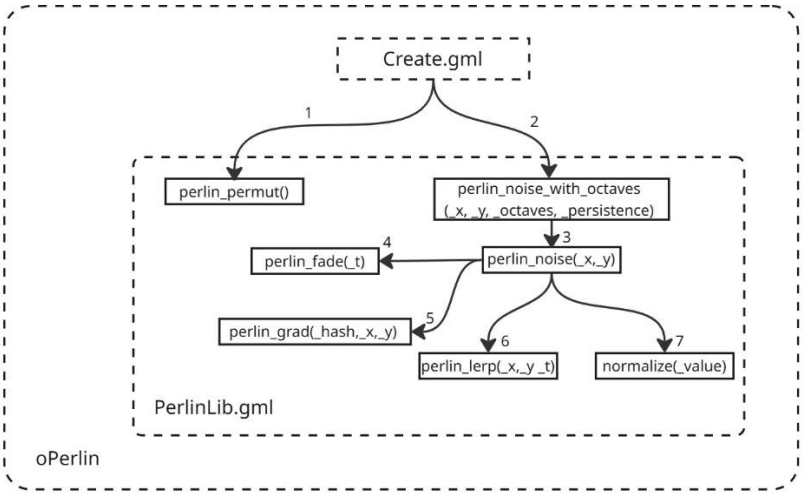
Fonte: Autoria própria (2025).

Na subseção seguinte, será apresentado o diagrama com o fluxo de execução do algoritmo, seguido da explicação das funções que compõem o processo.

5.1.1 Estrutura de Execução e Funções do Algoritmo

A Figura 23 apresenta o fluxo de execução do algoritmo Perlin Noise no contexto do objeto *oPerlin*. O diagrama destaca a ordem das chamadas de função desde a inicialização até o cálculo final do valor de ruído para cada ponto do mapa, separando as funções definidas no evento *Create* do objeto, daquelas localizadas no *script PerlinLib.gml.*

Figura 23 - Diagrama de fluxo das chamadas de funções do algoritmo Perlin Noise.



Fonte: Autoria própria (2025).

A execução tem início no evento *Create*, que ocorre apenas uma vez durante a instancia do objeto *oPerlin*, onde duas funções fundamentais são chamadas: *perlin_permut()* (etapa 1), responsável por gerar e embaralhar a tabela de permutação usada como base para os vetores gradientes, e *perlin_noise_with_octaves(_x,_y,_octaves,_persistence)* (etapa 2), que coordena o cálculo do ruído combinando múltiplas oitavas com diferentes frequências e amplitudes.

A cada iteração dessa função, é feita uma chamada para *perlin_noise(_x,_y)* (etapa 3), que executa o cálculo base do Perlin Noise para um par de coordenadas (x, y). Essa função utiliza três funções auxiliares:

- *perlin_fade(_t)* (etapa 4): aplica uma curva de suavização ao valor fracionário das coordenadas, garantindo que as transições entre células vizinhas ocorram de maneira contínua e sem quebras abruptas.
- *perlin_grad(_hash,_x,_y)* (etapa 5): com base em um valor de *_hash*, vindo da tabela de permutação, seleciona um vetor gradiente pré-definido para cada canto da célula e calcula o produto escalar com o vetor distância que liga esse canto ao ponto de interesse. Essa operação define a influência de cada canto no valor final.
- *perlin_lerp(_a, _b, _t)* (etapa 6): realiza a interpolação linear entre os valores gerados pelos produtos escalares, primeiro na direção horizontal e depois na vertical, finalizando o cálculo do ruído com uma transição suave em ambas as direções.
- *normalize(_value)* (etapa 7): realiza a normalização do valor final do ruído, garantido que ele esteja no intervalo [0,1].

A separação modular dessas funções e sua reutilização ao longo do processo tornam a implementação do algoritmo flexível, compacta e eficiente. O valor final de ruído retornado por *perlin_noise_with_octaves* é normalizado entre 0 e 1 e usado posteriormente para definir a "altura" de cada ponto do mapa, sendo, então, convertido em cor.

5.2 Binary Space Partitioning – BSP

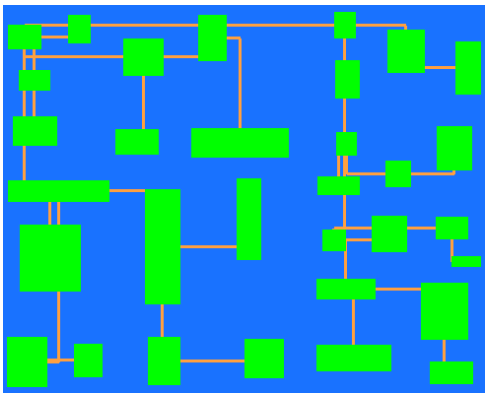
O BSP foi o segundo algoritmo implementado no protótipo e, assim como no Perlin Noise, a interface interativa permite ao usuário controlar os principais parâmetros ajustáveis que influenciam diretamente na configuração final do mapa gerado:

- ***Depth***: define a profundidade máxima da árvore BSP, ou seja, quantas divisões serão feitas no espaço total.
- ***Minimum Partition Size* (MPS)**: determina o tamanho mínimo permitido para cada região dividida, evitando divisões excessivas.
- ***Minimum Room Size* (MRS)**: estabelece o tamanho mínimo que uma sala pode ter dentro de uma partição.
- ***Margin***: define o espaço mínimo entre as bordas da partição e a sala, criando corredores mais definidos e menos colados às paredes.

A cada vez que o botão "*Generate*" é pressionado, o sistema recria a árvore BSP com as configurações escolhidas e redesenha o mapa, possibilitando a análise visual de diferentes combinações de parâmetros. A Figura 24 apresenta um mapa gerado com os valores padrão de cada parâmetro, determinados ao inicializar o protótipo.

Figura 24 - Exemplo de um mapa gerado com os valores padrão para cada parâmetro ao inicializar o protótipo.

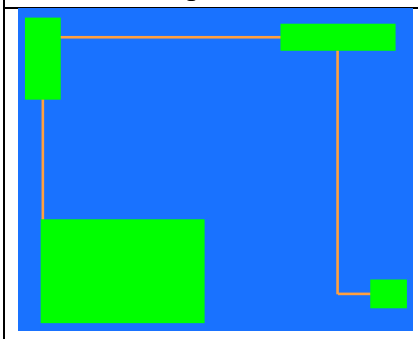
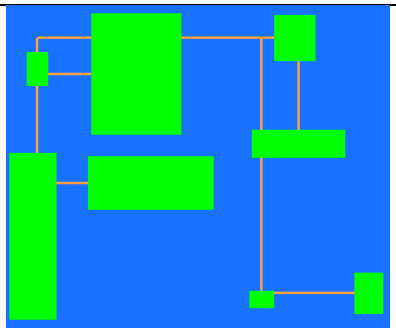
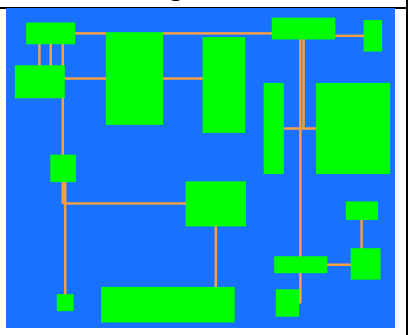
Depth = 5	MPS = 20	MRS = 20	Margin = 10



Fonte: Autoria própria (2025).

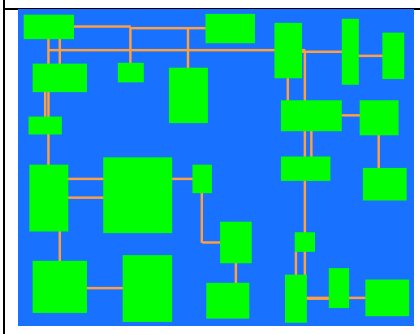
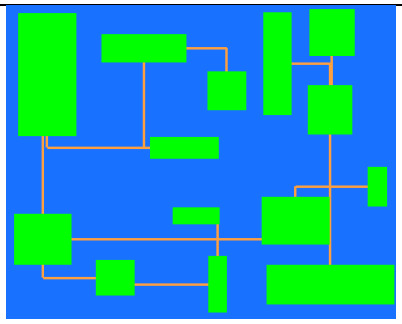
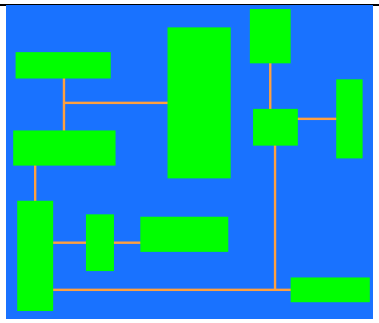
A seguir, é apresentado as Figuras 24A, 24B, 24C e 24D, onde é demonstrado o mapa apresentado na Figura 24 com diferentes valores, respectivamente, para *Depth*, *MPS*, *MRS* e *Margin*. Considere que apenas um parâmetro está sendo alterado por figura, mantendo os demais com o valor padrão.

Figura 24A - Mapa representado pela Figura 24 com diferentes valores de *Depth*, mantendo os demais parâmetros como padrão.

Depth = 2	Depth = 3	Depth = 4
		

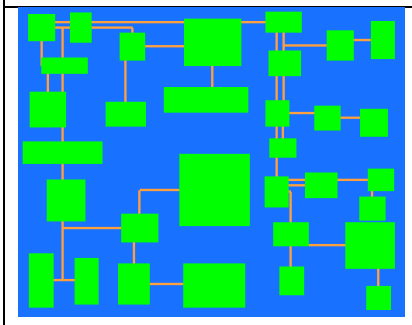
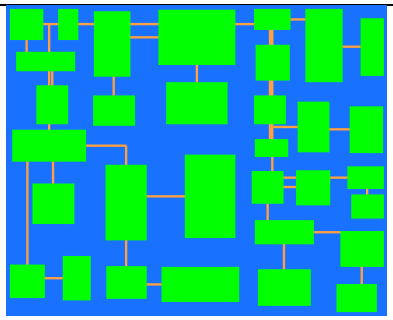
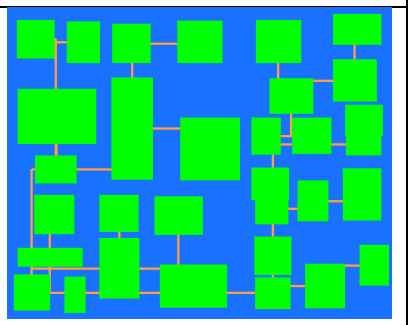
Fonte: Autoria própria (2025).

Figura 24B - Mapa representado pela Figura 24 com diferentes valores de *MPS*, mantendo os demais parâmetros como padrão.

MPS = 40	MPS = 60	MPS = 80
		

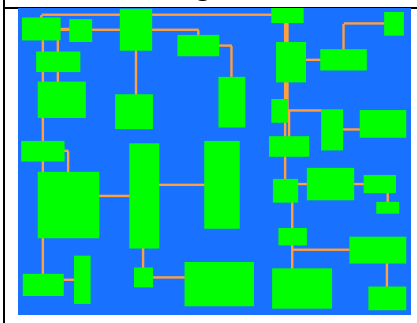
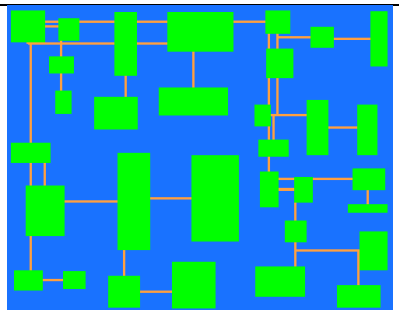
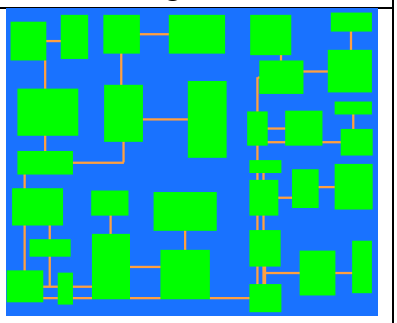
Fonte: Autoria própria (2025).

Figura 24C - Mapa representado pela Figura 24 com diferentes valores de *MRS*, mantendo os demais parâmetros como padrão.

MRS = 30	MRS = 40	MRS = 50
		

Fonte: Autoria própria (2025).

Figura 24D - Mapa representado pela Figura 24 com diferentes valores de *Margin*, mantendo os demais parâmetros como padrão.

Margin = 6	Margin = 8	Margin = 13
		

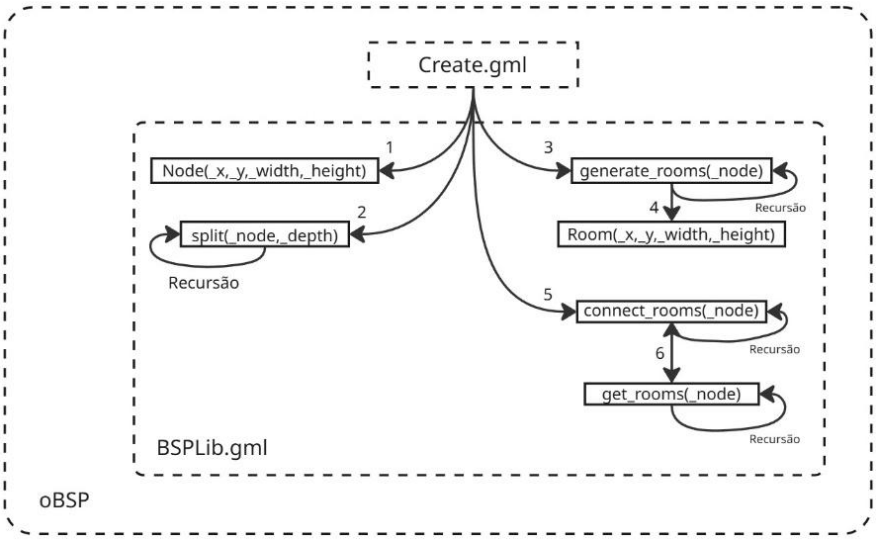
Fonte: Autoria própria (2025).

Na subseção seguinte, será apresentado o diagrama com o fluxo de execução do algoritmo, seguido da explicação das funções que compõem o processo.

5.2.1 Estrutura de Execução e Funções do Algoritmo

A Figura 25 apresenta o fluxo de execução do algoritmo BSP dentro do objeto *oBSP*, detalhando a sequência de chamadas realizadas a partir do evento *Create*. Assim como no algoritmo Perlin Noise, as funções foram organizadas em um *script* separado chamado *BSPLib.gml*, o que facilita a manutenção e a reutilização do código.

Figura 25 - Diagrama de fluxo das chamadas de funções do algoritmo BSP.



Fonte: Autoria própria (2025).

A execução se inicia com a criação de um nó raiz por meio do construtor *Node(_x,_y,_width,_height)* (etapa 1), que define o espaço total que será particionado. Em seguida, a função *split(_node,_depth)* (etapa 2) é chamada para dividir esse espaço recursivamente, formando a estrutura hierárquica característica do algoritmo BSP. A profundidade máxima da divisão é controlada por um dos parâmetros ajustáveis fornecidos na interface interativa.

Após a criação da estrutura de partições, a função *generate_rooms(_node)* (etapa 3) percorre recursivamente a árvore BSP, identificando as folhas e alocando, dentro de cada uma, uma sala de tamanho aleatório por meio do construtor *Room(_x,_y,_width,_height)* (etapa 4).

A posição e o tamanho da sala são calculados com base nas dimensões disponíveis da partição e respeitando margens mínimas pré-definidas.

Na sequência, o processo de conexão das salas é iniciado com a chamada à função *connect_rooms(_node)* (etapa 5). Essa função também é recursiva e, para cada nó interno da árvore, busca conectar uma sala em sua subárvore esquerda com uma sala na subárvore direita. Essa busca é realizada pela função auxiliar *get_rooms(_node)* (etapa 6), que percorre as subárvores até encontrar uma sala válida.

Os corredores gerados podem ser retos, quando os centros das salas estão alinhados, ou em formato de "L" quando não estão, adicionando variedade à topologia do mapa. Ao final da execução, todos os nós da árvore foram processados, todas as salas foram criadas, e todos os pares de subárvores foram conectados de forma lógica e navegável.

5 CONSIDERAÇÕES FINAIS

Este artigo apresentou a geração procedural de mapas 2D com foco na implementação prática dos algoritmos Perlin Noise e Binary Space Partitioning (BSP). Mais do que uma exposição conceitual, o objetivo principal foi oferecer um guia acessível e aplicável, permitindo que desenvolvedores entendam o funcionamento interno desses algoritmos e consigam aplicá-los em seus próprios jogos.

O protótipo desenvolvido permitiu explorar os parâmetros ajustáveis de cada algoritmo por meio de uma interface interativa, facilitando a experimentação e visualização dos efeitos. Diagramas ilustraram a estrutura das funções, reforçando a compreensão da lógica envolvida.

Como continuidade, este trabalho pode ser expandido para incorporar elementos adicionais de jogabilidade, como geração de inimigos, obstáculos, eventos e lógica de caminhos. Além disso, o uso de *tilesets* e melhorias visuais pode tornar os mapas ainda mais próximos de jogos reais.

Espera-se que este artigo sirva como ponto de partida para desenvolvedores, estudantes e entusiastas que desejam adentrar o universo da geração procedural com bases sólidas e ferramentas práticas à disposição. O repositório do projeto permanece aberto para colaboração, permitindo que desenvolvedores experimentem, modifiquem e expandam os algoritmos explorados aqui.

REFERÊNCIAS

Biagioli, A. (2014). "Understanding Perlin Noise". Disponível em: https://adrianb.io/2014/08/09/perlinnoise.html. Acesso em: 10 jun. 2025

Canedo, G. (2022): "Criação de Mapas Utilizando Geração Procedural". Trabalho de Conclusão de Curso. Pontifícia Universidade Católica de Goiás. Disponível a partir de https://repositorio.pucgoias.edu.br/jspui/handle/123456789/5077

Leite, G. e Lima, E. (2015): "Geração Procedural de Mapas para Jogos 2D". Proceedings of SBGames 2015. ISSN: 2179-2259, pp. 244-247. Disponível em: https://www.sbgames.org/sbgames2015/anaispdf/computacao-short/147654.pdf

Peter, A. "Perlin Noise". Disponível em: https://www.arendpeter.com/Perlin_Noise.html#:~:text=In%20this%20case%2C%20the%20amplitude,Creating%20the%20Perlin%20Noise%20Function. Acesso em: 12 jul. 2025

Re-Logic (2011). “Terraria”. Disponível em: https://terraria.org/. Acesso em: 01 ago. 2025

Rizzon, J. “O que é: Binary Space Partitioning”. Disponível em: https://napoleon.com.br/glossario/o-que-e-binary-space-partitioning-2/. Acesso em: 10 jun. 2025

Roguebasin (2013). “Basic BSP Dungeon generation”. Disponível em: https://www.roguebasin.com/index.php?title=Basic_BSP_Dungeon_generation. Acesso em: 10 jun. 2025

Scher, Y. (2017). “Playing with Perlin Noise: Generating Realistic Archipelagos”. Disponível em: https://medium.com/@yvanscher/playing-with-perlin-noise-generating-realistic-archipelagos-b59f004d8401. Acesso em: 18 jul. 2025

Scratchapixel. “Value Noise and Procedural Patterns”. Disponível em: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1/creating-simple-1D-noise.html. Acesso em: 10 jun. 2025

Shields, J. (2023). “Procedural Dungeon Generation in Godot”. Disponível em: https://jonoshields.com/post/bsp-dungeon. Acesso em: 10 jun. 2025

Sousa, V. (2025). “Sistema de Geração Procedural de Mapas”. Disponível em: https://github.com/vinicius-dantasso/procedural-generation. Acesso em: 05 jun. 2025

Stephenson, M. (1985). “Nethack 3.6.7: Nethack Home Page”. Disponível em: https://www.nethack.org/. Acesso em: 01 ago. 2025

YoYo Games. “Faça jogos 2D com GameMaker | Criador de Jogos Gratuito”. Disponível em: https://gamemaker.io/pt-BR. Acesso em: 12 jun. 2025